DERS KİTABI

C ile Veri Yapıları ve Algoritmalar

Sercan Külcü

Giresun Üniversitesi · Bilgisayar Mühendisliği Bölümü

C ile Veri Yapıları ve Algoritmalar

Sercan Külcü
Giresun Üniversitesi, Bilgisayar Mühendisliği Bölümü
Giresun, Türkiye

Bu kitap, şu adreste yayımlanan ders notlarının basılı hâlidir:
`sercankulcu.github.io/teaching/data-structures-and-algorithms-with-c/`

Sunumlar ve çalışma soruları o sayfadadır.

© Sercan Külcü. Tüm hakları saklıdır.

Notların 13 Kasım 2026 tarihli hâlinden üretilmiştir.

Önsöz

Bu kitap veri yapılarını ve algoritmaları bellekteki gösterimleri ve C ile gerçeklenmeleri üzerinden inceler. Her konu soyut veri tipinden başlar; işaretçiler, dinamik bellek ve yapılarla çalışan bir C99/C11 gerçeklemesine ve onun karmaşıklık çözümlemesine ulaşır. Kitap, Giresun Üniversitesi Bilgisayar Mühendisliği Bölümü'nde verilen C ile Veri Yapıları ve Algoritmalar dersinin notlarından oluşur ve bölümleri dersteki sırayla izler.

Kitap beş kısımdır. Kısımlar sırasıyla şunlardır: **Temeller** (Bölüm 1–3), **Doğrusal Yapılar** (Bölüm 4–9), **Ağaçlar** (Bölüm 10–13), **Hash, Çizgeler ve Kümeler** (Bölüm 14–18) ve **Sıralama** (Bölüm 19–20).

Her bölüm aynı düzeni izler. Başta öğrenme çıktıları yer alır. Biçimsel tanımlar, teoremler ve çözülmüş örnekler adlandırılmış kutulardadır. *Sık yapılan hata* kutuları, öğrencilerin en sık düştüğü yanılgıları gösterir. Kod izlemeleri, bir programın ya da algoritmanın adım adım nasıl çalıştığını gösterir. Bölüm; bir özet, bir kavram listesi, cevaplarıyla birlikte *Kendini Dene* soruları, ileri okuma önerileri ve kaynakçayla kapanır. Kitabın sonundaki terim sözlüğü, her İngilizce terim için notlarda kullanılan tek Türkçe karşılığı verir.

Sunumlar ve çalışma soruları ders sayfasındadır.

Sercan Külcü
Giresun

İçindekiler

Dersin Haritası 1

KISIM I · TEMELLER

1 Giriş ve Algoritma Analizi 4

1.1 Ders kitapları ve meslek sevgisi 4
1.2 Sistem yaşam döngüsü 5
1.3 Algoritma 7
1.4 Soyut veri tipi 7
1.5 Seçimli sıralama 12
1.6 Performans analizi 17
1.7 Asimptotik gösterim 24
1.8 Sihirli kare 29
1.9 Performans ölçümü 31
1.10 Özyinelemeli yordamlar 35
1.11 Karmaşıklık özeti 36
1.12 Sık yapılan hatalar 37
1.13 Özet 37
1.14 Kavram listesi 37
1.15 Kendini Dene 38
1.16 İleri okuma 39
1.17 Kaynaklar 39

2 Diziler ve Yapılar 40

2.1 Dizi soyut veri tipi 40
2.2 C'de tek boyutlu dizi 41
2.3 Dinamik diziler 43
2.4 Çok boyutlu diziler ve adres hesabı 48
2.5 Üçgen matrisler 52
2.6 Kuşak matrisler 55
2.7 Yapılar ve birlikler 60
2.8 Dizgiler 63
2.9 Matris işlemleri 66
2.10 Karmaşıklık özeti 68
2.11 Sık yapılan hatalar 68
2.12 Özet 68
2.13 Kavram listesi 69
2.14 Kendini Dene 69
2.15 İleri okuma 71
2.16 Kaynaklar 71

3 Özyineleme, İkili Arama ve Geri İzleme 72

3.1 Özyinelemeli algoritmalar 72
3.2 Çağrı yığını 74
3.3 Özyineleme mi, döngü mü? 76
3.4 Kuyruk özyinelemesi 77
3.5 İkili arama 78
3.6 Hanoi kuleleri 83
3.7 Permütasyon üretimi 86
3.8 Kaba kuvvet ve geri izleme 89
3.9 Sağlanabilirlik (SAT) problemi 91
3.10 n-vezir problemi 96
3.11 At gezisi ve Warnsdorff kuralı 99
3.12 Karmaşıklık özeti 101
3.13 Sık yapılan hatalar 101
3.14 Özet 102
3.15 Kavram listesi 102
3.16 Kendini Dene 102
3.17 İleri okuma 104

3.18 Kaynaklar 104

KISIM II · DOĞRUSAL YAPILAR

4 Bağlı Listeler 106
4.1 Dizi yetmediğinde 106
4.2 C'de işaretçi ve dinamik bellek 108
4.3 Tek yönlü bağlı liste: düğüm ve gezinme 111
4.4 Ekleme 112
4.5 Silme, arama ve listeyi bırakma 117
4.6 Klasik yordamlar 123
4.7 Dairesel liste ve başlık düğümü 128
4.8 Çift yönlü bağlı liste 131
4.9 Kullanılabilir düğüm listesi 134
4.10 Karmaşıklık özeti ve ölçüm 136
4.11 Sık yapılan hatalar 137
4.12 Alıştırmalar ve çözümleri 137
4.13 Özet 140
4.14 Kavram listesi 140
4.15 Kendini Dene 140
4.16 İleri okuma 142
4.17 Kaynaklar 142

5 Polinomlar ve Seyrek Matrisler 143
5.1 Polinom soyut veri tipi 143
5.2 Polinomu dizide tutmak 144
5.3 Polinom toplama: padd 146
5.4 Dizi gösterimiyle çarpım ve çok polinomlu işlemler 150
5.5 Bağlı liste ile polinom 151
5.6 Seyrek matris 156
5.7 Transpoz alma 158
5.8 Seyrek matris çarpımı 164
5.9 Öteki gösterimler 167
5.10 Karmaşıklık özeti 168
5.11 Sık yapılan hatalar 168
5.12 Özet 169
5.13 Kavram listesi 169
5.14 Kendini Dene 169
5.15 İleri okuma 171
5.16 Kaynaklar 171

6 Yığınlar 172
6.1 Yığın: son giren ilk çıkar 172
6.2 Yığın soyut veri tipi 174
6.3 Dizi ile yığın 176
6.4 Yığını bir nesne gibi tasarlamak 180
6.5 Bağlı yığın 184
6.6 Dizi tabanlı mı, bağlı mı? 188
6.7 Çağrı yığını 190
6.8 Çoklu yığın 192
6.9 Hangi çıkış sırası mümkün? 195
6.10 Karmaşıklık özeti 195
6.11 Sık yapılan hatalar 196
6.12 Alıştırmalar ve çözümleri 196
6.13 Özet 198
6.14 Kavram listesi 199
6.15 Kendini Dene 199
6.16 İleri okuma 200
6.17 Kaynaklar 200

7 Kuyruklar 201
7.1 Kuyruk nedir? 201
7.2 Dizi ile doğrusal kuyruk 203

7.3 Dairesel kuyruk 207
7.4 Bağlı kuyruk 214
7.5 Deque ve uygulamalar 218
7.6 Karmaşıklık özeti ve ölçüm 225
7.7 Sık yapılan hatalar 226
7.8 Alıştırmalar ve çözümleri 226
7.9 Özet 228
7.10 Kavram listesi 228
7.11 Kendini Dene 228
7.12 İleri okuma 230
7.13 Kaynaklar 230

8 Yığın ve Kuyruk Uygulamaları 231

8.1 Yığın ve geri izleme 231
8.2 Labirent problemi 232
8.3 İfade gösterimleri 238
8.4 Sonek ifadeyi hesaplama 242
8.5 Ara ekten soneke çevirme 248
8.6 Önek gösterim 252
8.7 Parantez denetimi 255
8.8 Yığın ve kuyruğun başka uygulamaları 257
8.9 Karmaşıklık özeti 260
8.10 Sık yapılan hatalar 260
8.11 Alıştırmalar ve çözümleri 261
8.12 Özet 263
8.13 Kavram listesi 263
8.14 Kendini Dene 264
8.15 İleri okuma 265
8.16 Kaynaklar 265

9 Liste ADT ve Genelleştirilmiş Listeler 266

9.1 Liste soyut veri tipi 266
9.2 Dizi ile gerçekleme 268
9.3 Bağlı gerçekleme 272
9.4 Genelleştirilmiş listeler 275
9.5 Özyinelemeli temel yordamlar 281
9.6 Derinlik 287
9.7 Ters çevirme 291
9.8 Ağacı listeyle kurmak 293
9.9 Paylaşım, referans sayma ve çöp toplama 295
9.10 Sınav soruları 297
9.11 Karmaşıklık özeti ve ölçüm 299
9.12 Sık yapılan hatalar 300
9.13 Alıştırmalar ve çözümleri 300
9.14 Özet 302
9.15 Kavram listesi 302
9.16 Kendini Dene 302
9.17 İleri okuma 304
9.18 Kaynaklar 304

KISIM III · AĞAÇLAR

10 Ağaçlar 306

10.1 Ağaç veri yapısı 306
10.2 Terminoloji 307
10.3 İkili ağaç 309
10.4 İkili ağacın bellekte gösterimi 313
10.5 Özyinelemeli gezinmeler 317
10.6 Yığınla gezinme 323
10.7 Düzey düzey gezinme 329
10.8 Diğer ağaç yordamları 333
10.9 Gezinmelerden ağacı yeniden kurmak 337
10.10 Karmaşıklık özeti 338

10.11 Sık yapılan hatalar 339
10.12 Alıştırmalar ve çözümleri 339
10.13 Özet 341
10.14 Kavram listesi 341
10.15 Kendini Dene 341
10.16 İleri okuma 342
10.17 Kaynaklar 343

11 Arama Ağaçları 344

11.1 İkili arama ağacı 344
11.2 Ekleme 347
11.3 Arama 353
11.4 Derinlik 356
11.5 Silme 360
11.6 Dengeli ağaçlar 366
11.7 Sınav soruları 374
11.8 Karmaşıklık özeti 377
11.9 Sık yapılan hatalar 378
11.10 Alıştırmalar ve çözümleri 378
11.11 Özet 380
11.12 Kavram listesi 380
11.13 Kendini Dene 380
11.14 İleri okuma 381
11.15 Kaynaklar 382

12 Kılavuzlu İkili Ağaçlar 383

12.1 Boş işaretçiler ve kılavuz fikri 383
12.2 Kılavuzları kurmak 385
12.3 Kök ortada gezinme 390
12.4 Düğüm ekleme 396
12.5 Kazanç, bedel ve karmaşıklık özeti 402
12.6 Sık yapılan hatalar 402
12.7 Alıştırmalar ve çözümleri 403
12.8 Özet 404
12.9 Kavram listesi 404
12.10 Kendini Dene 405
12.11 İleri okuma 406
12.12 Kaynaklar 406

13 Öncelikli Kuyruk ve Heap 407

13.1 Öncelikli kuyruk 407
13.2 Heap yapısı 409
13.3 Ekleme 413
13.4 Silme 418
13.5 Gerçeklemeyi geliştirmek 423
13.6 Diğer yapılarla karşılaştırma 426
13.7 Aşağıdan yukarı heap kurma 429
13.8 d-ary heap 431
13.9 Uygulamalar ve heap sort 432
13.10 Karmaşıklık özeti 433
13.11 Sık yapılan hatalar 433
13.12 Sınav tarzı soru 434
13.13 Alıştırmalar ve çözümleri 434
13.14 Özet 435
13.15 Kavram listesi 435
13.16 Kendini Dene 436
13.17 İleri okuma 437
13.18 Kaynaklar 437

KISIM IV · HASH, ÇİZGELER VE KÜMELER

14 Hash Tabloları 439

14.1 Hash fikri 439

14.2 Hash fonksiyonları 442
14.3 Açık adresleme 448
14.4 Zincirleme 454
14.5 Silme, yük faktörü ve yeniden boyutlandırma 458
14.6 Zincirleme mi, açık adresleme mi? 464
14.7 Karmaşıklık özeti 464
14.8 Sık yapılan hatalar 465
14.9 Alıştırmalar ve çözümleri 465
14.10 Özet 466
14.11 Kavram listesi 466
14.12 Kendini Dene 467
14.13 İleri okuma 468
14.14 Kaynaklar 468

15 Çizgeler 469

15.1 Giriş: köprüler, haritalar ve ağlar 469
15.2 Tanımlar ve terimler 471
15.3 Çizgeyi bellekte gerçeklemek 474
15.4 Gezinmeler 489
15.5 Bileşenler, döngüler ve iki parçalılık 495
15.6 Karmaşıklık özeti 501
15.7 Sık yapılan hatalar 501
15.8 Alıştırmalar ve çözümleri 502
15.9 Özet 503
15.10 Kavram listesi 504
15.11 Kendini Dene 504
15.12 İleri okuma 505
15.13 Kaynaklar 505

16 Kümeler ve Union-Find 506

16.1 Ayrık kümeler 506
16.2 Dizi ile gerçekleme 508
16.3 İlk sürümün analizi 512
16.4 Ağırlıklı kural 513
16.5 Azaltma kuralı 517
16.6 Tarjan–van Leeuwen sonucu ve Ackermann fonksiyonu 520
16.7 Uygulamalar 523
16.8 Sınav sorusu: ebeveyn dizisinin içeriği 526
16.9 Karmaşıklık özeti 527
16.10 Sık yapılan hatalar 527
16.11 Alıştırmalar ve çözümleri 528
16.12 Özet 529
16.13 Kavram listesi 529
16.14 Kendini Dene 529
16.15 İleri okuma 530
16.16 Kaynaklar 530

17 Minimum Kapsayan Ağaçlar 531

17.1 Kapsayan ağaçlar 531
17.2 Minimum maliyetli kapsayan ağaç 535
17.3 Kruskal algoritması 537
17.4 Prim algoritması 544
17.5 Öteki algoritmalar ve karşılaştırmalar 550
17.6 Sınav tarzı soru 552
17.7 Ödev: Kruskal'ı veri yapısıyla kodlamak 553
17.8 Karmaşıklık özeti 553
17.9 Sık yapılan hatalar 554
17.10 Alıştırmalar 554
17.11 Özet 555
17.12 Kavram listesi 555
17.13 Kendini Dene 556
17.14 İleri okuma 557

17.15 Kaynaklar 557

18 En Kısa Yol ve Aktivite Ağları 558

18.1 En kısa yol problemi 558
18.2 Kaba kuvvet çözümü 559
18.3 Dijkstra algoritması 561
18.4 Yolu bulmak: öncüller dizisi 566
18.5 Negatif kenarlar ve tüm çiftler 568
18.6 Aktivite ağları ve topolojik sıra 573
18.7 Kenarda aktivite ağları ve kritik yol 580
18.8 En erken ve en geç zamanlar 582
18.9 Karmaşıklık özeti 589
18.10 Sık yapılan hatalar 589
18.11 Alıştırmalar ve çözümleri 590
18.12 Özet 590
18.13 Kavram listesi 591
18.14 Kendini Dene 591
18.15 İleri okuma 592
18.16 Kaynaklar 592

KISIM V · SIRALAMA

19 Temel Sıralama Algoritmaları 594

19.1 Sıralama kavramları 594
19.2 Seçimli sıralama 598
19.3 Eklemeli sıralama 600
19.4 Kabarcık sıralaması 605
19.5 Ters çiftler: ortak ölçü 608
19.6 Saymalı sıralama 609
19.7 Shell sıralaması 612
19.8 Kova sıralaması 614
19.9 Karşılaştırmalı sıralamanın alt sınırı 616
19.10 Algoritmaları karşılaştırmak 618
19.11 Sık yapılan hatalar 619
19.12 Alıştırmalar ve çözümleri 620
19.13 Özet 621
19.14 Kavram listesi 621
19.15 Kendini Dene 621
19.16 İleri okuma 622
19.17 Kaynaklar 622

20 İleri Sıralama Algoritmaları 624

20.1 Sıralamayı hızlandırmak: böl ve fethet 624
20.2 Hızlı sıralama 625
20.3 Hızlı sıralamanın analizi 629
20.4 Hızlı sıralamayı iyileştirmek 631
20.5 C kitaplığı ve hibrit sıralamalar 636
20.6 Birleştirme sıralaması 638
20.7 Bağlı birleştirme sıralaması 643
20.8 Heap sort 646
20.9 Basamak sıralaması 650
20.10 Karşılaştırma ve seçim 654
20.11 Sık yapılan hatalar 655
20.12 Alıştırmalar ve çözümleri 655
20.13 Özet 656
20.14 Kavram listesi 656
20.15 Kendini Dene 656
20.16 İleri okuma 657
20.17 Kaynaklar 657

EKLER

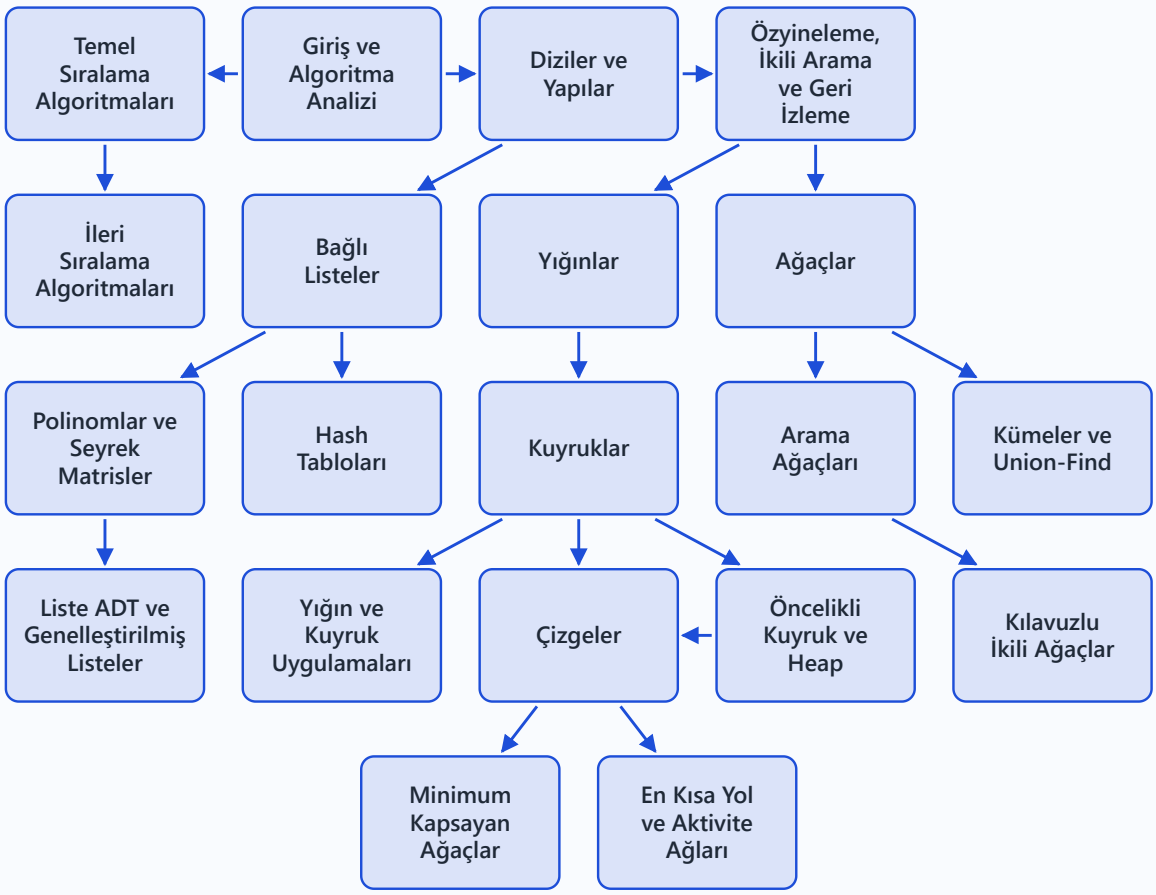
Ek A Terim Sözlüğü 658

Ek B C ile Veri Yapıları İçin Ön Bilgiler 663

Ek C C ile Veri Yapılarında Önemli İsimler 676

Dersin Haritası

Bu sayfa dersin bütün bölümlerini tek bir haritada birleştirir. Her bölüm bir düğümdür; bir ok, sonraki konunun önceki konunun hangi fikrine dayandığını gösterir. Derse başlamadan önce okunursa konuların nereye gittiğini, ders sırasında okunursa bulunulan yeri gösterir.



Şekil 1: Dersin haritası. Oklar, bir konunun hangi konunun üzerine kurulduğunu gösterir. Dizi ve özyinelemeden çıkan yollar doğrusal yapılara, ağaçlara, çizgelere ve sıralamaya ayrılır.

Şekil 1 konular arasındaki bağları gösterir. Her düğüm bir bölümdür; bir ok, sonraki konunun öncekinin hangi fikrine dayandığını belirtir. Haritada yalnızca en yakın bağlar çizilmiştir; bir bölümün dayandığı bütün bölümler Tablo 1'dedir.

Konu (bölüm)	Dayandığı konu	Bağ
Giriş ve Algoritma Analizi (1)	—	Yazılımın yaşam döngüsü, soyut veri tipinin C başlık dosyasıyla ifadesi ve asimptotik analiz.
Diziler ve Yapılar (2)	Giriş ve Algoritma Analizi	Dizi ADT'si ve dizi ile işaretçi ilişkisi; malloc ile dinamik diziler, yapılar.
Özyineleme, İkili Arama ve Geri İzleme (3)	Giriş ve Algoritma Analizi, Diziler ve Yapılar	Özyinelemeyi yığın çerçeveleriyle izlemek; sıralı dizide ikili arama, Hanoi ve permütasyon.
Bağlı Listeler (4)	Diziler ve Yapılar	Dizinin yetersiz kaldığı yerde malloc ve free ile kurulan düğümler.

Konu (bölüm)	Dayandığı konu	Bağ
Polinomlar ve Seyrek Matrisler (5)	Diziler ve Yapılar, Bağlı Listeler	Aynı ADT'nin iki gösterimi: katsayı ve terim dizileri, bağlı liste; seyrek matrisler.
Yığınlar (6)	Diziler ve Yapılar, Özyineleme, İkili Arama ve Geri İzleme, Bağlı Listeler	LIFO yapısı: dizi ve struct ile push ve pop; çağrı yığını özyinelemenin yığınıdır.
Kuyruklar (7)	Yığınlar	FIFO yapısı: doğrusal kuyruğun kayma sorunu dairesel kuyrukla çözülür.
Yığın ve Kuyruk Uygulamaları (8)	Yığınlar, Kuyruklar	Labirentte geri izleme, infix ifadeyi postfix'e çevirme ve değerlendirme.
Liste ADT ve Genelleştirilmiş Listeler (9)	Bağlı Listeler, Polinomlar ve Seyrek Matrisler	Liste ADT'sinin sözleşmesi; çok değişkenli polinomlar genelleştirilmiş listeye götürür.
Ağaçlar (10)	Özyineleme, İkili Arama ve Geri İzleme, Bağlı Listeler	İkili ağaç, dizi ve bağlı gösterim; gezinmeler özyinelemeyle yazılır.
Arama Ağaçları (11)	Özyineleme, İkili Arama ve Geri İzleme, Ağaçlar	Ağaca sıralama koşulu: ekleme ve arama ikili aramanın ağaçtaki karşılığıdır.
Kılavuzlu İkili Ağaçlar (12)	Ağaçlar, Arama Ağaçları	Boş kalan n+1 işaretçi kılavuza dönüşür: yığınsız kök ortada gezinme.
Öncelikli Kuyruk ve Heap (13)	Kuyruklar, Ağaçlar	Kuyruğun öncelikli biçimi; tam ikili ağacın dizide saklanmasıyla heap.
Hash Tabloları (14)	Diziler ve Yapılar, Bağlı Listeler	Doğrudan adreslemeden hash'e; açık adresleme dizi, zincirleme bağlı liste kullanır.
Çizgeler (15)	Bağlı Listeler, Kuyruklar, Öncelikli Kuyruk ve Heap	Komşuluk listesi bağlı listedir; BFS kuyrukla, DFS yığınla gezinir.
Kümeler ve Union-Find (16)	Ağaçlar	Ayrık kümeler ata dizisiyle ağaç olarak saklanır; ağırlıklı birleşim ve yol kısaltma.
Minimum Kapsayan Ağaçlar (17)	Öncelikli Kuyruk ve Heap, Çizgeler, Kümeler ve Union-Find	Açgözlü kapsayan ağaç: Kruskal union-find, Prim heap kullanır.
En Kısa Yol ve Aktivite Ağları (18)	Öncelikli Kuyruk ve Heap, Çizgeler	Dijkstra, Bellman-Ford ve Floyd-Warshall; AoV ve AoE ağlarında topolojik sıra ve kritik yol.
Temel Sıralama Algoritmaları (19)	Giriş ve Algoritma Analizi, Diziler ve Yapılar	İlk örnek olan seçimli sıralama genellenir: eklemeli ve kabarcık sıralaması, kararlılık.
İleri Sıralama Algoritmaları (20)	Özyineleme, İkili Arama ve Geri İzleme, Öncelikli Kuyruk ve Heap, Temel Sıralama Algoritmaları	Böl ve fethet ile hızlı ve birleştirme sıralaması; heap sıralaması ve basamağa göre sıralama.

Tablo 1: Haritadaki bağlar.

Haritanın ana omurgası şöyle okunabilir: **dizi ve işaretçi → bağlı yapı → ağaç → çizge**. Her konu soyut veri tipinden çalışan bir C gerçeklemesine kadar götürülür; bunun iki aracı dizi ile malloc ve free üzerine kurulan bağlı listedir. Yığın ve kuyruk her iki gerçeklemeyle yazılır ve uygulamalarda birleşir. Özyineleme ağaçlara götürür; heap ve ayrık kümeler ağacın dizideki biçimleridir. Çizgeler bu yapıların hepsini birlikte kullanır: kapsayan ağaç için heap ve union-find, en kısa yol için öncelikli kuyruk. Sıralama ise ilk bölümdeki seçimli sıralamadan başlar ve son bölümde böl ve fethet ile heap'e dayanır.

KISIM I

Temeller

Bölüm 1–3

KISIM I · BÖLÜM 1

Giriş ve Algoritma Analizi

Bir yazılım, gereksinimlerin yazılmasından doğrulamaya kadar uzanan bir süreçte ortaya çıkar. Bu sürecin iki temel taşı veri ve o veri üzerinde çalışan algoritmadır. Not, önce bu süreci ve algoritma ile soyut veri tipi kavramlarını anlatır; C'de bir soyut veri tipinin başlık dosyası ve opak tiple nasıl kurulduğunu gösterir. Ardından seçimli sıralama üzerinden ilk C kodlarını, takas işlemini ve işaretçiyle parametre geçirmeyi ele alır. Notun büyük kısmı bir programın ne kadar bellek ve zaman harcadığını hesaplamaya ayrılmıştır: program adımlarını saymak, sonucu O, Ω ve Θ gösterimleriyle sınıflamak, büyüme hızlarını karşılaştırmak. Sihirli kare problemi doğru algoritmanın donanımdan önce geldiğini gösterir. Son kesimler `clock()` ile gerçek süre ölçümünü, ölçümün tuzaklarını ve özyinelemeli yordamların maliyetini anlatır.

Kod (C): `github.com/sercankulcu/data-structures-and-algorithms-c/Bolum01`

ÖĞRENME ÇIKTILARI

Süreci tanımak
Sistem yaşam döngüsünün beş aşamasını ve her aşamada veri yapısı kararlarının yerini açıklamak.

Soyutlamak
Algoritma ve soyut veri tipi kavramlarını tanımlamak; C'de başlık dosyası ve opak tiple bir arayüz kurmak.

Saymak
Bir C yordamının bellek gereksinimini ve program adımı sayısını çıkarmak (`sum` , `rsum` , `add`).

Sınıflandırmak
Adım sayısını O, Ω ve Θ ile ifade etmek; en iyi, ortalama ve en kötü durumu ayırmak.

Karşılaştırmak
Alışılmış karmaşıklık sınıflarının *n* büyüdükçe nasıl ayrıştığını yorumlamak.

Ölçmek
`clock()` ile işletim süresini ölçmek; saat vuruşu ve çıktı işlemleri gibi ölçüm tuzaklarını tanımak.

1.1 Ders kitapları ve meslek sevgisi

Veri yapıları ve algoritmalar, bilgisayar mühendisliğinin geri kalanının üzerine kurulduğu temel derslerdendir. İşletim sistemi, derleyici, veri tabanı, ağ yazılımı ya da oyun motoru yazan her mühendis, verisini bellekte nasıl tutacağına ve onu hangi yöntemle işleyeceğine karar verir. Bu derste her yapı C diliyle gerçeklenir. C, belleği doğrudan gösterdiği için bir yapının bellekte nasıl durduğunu, bir işaretçinin neyi gösterdiğini ve ayrılan belleğin ne zaman geri verildiğini açıkça görmeyi sağlar.

1.1.1 Ders kitapları

Dersin ana kitabı Horowitz, Sahni ve Anderson-Freed'in *Fundamentals of Data Structures in C* adlı kitabıdır. Bu notun konu sırası, örnekleri ve kodlarının çoğu bu kitabın birinci bölümünden gelir. Kitabın aslında üç sürümü vardır. İlk sürümdeki kodlar SPARKS adlı bir sözde dille yazılmıştı. Sonraki sürümlerde aynı içerik C ve C++ ile yeniden yazıldı. Konular ve sıralanışları hemen hemen aynıdır; değişen, kodların dilidir. C++ sürümü sınıflar ve şablonlar nedeniyle en kapsamlı olanıdır, ama bu derste C sürümü izlenir (Şekil 1.1).



Şekil 1.1: Ders kitabının üç sürümü. Soldan sağa: SPARKS sözde diliyle yazılan ilk sürüm, dersin ana kitabı olan C sürümü ve en kapsamlı olan C++ sürü-

mü.

Kitaplığın ikinci sırası Kernighan ve Ritchie'nin *The C Programming Language* kitabıdır. Dennis Ritchie C dilinin tasarımcısıdır; kitap da dilin her ayrıntısını kısa ve kesin biçimde anlatır. Bir işlecin önceliği, bir standart kütüphane fonksiyonunun dönüş değeri ya da bir bildirimin anlamı konusunda kuşku doğduğunda başvurulacak yer burasıdır. Bu yüzden bir başucu kitabı olarak el altında durmalıdır.

Ek kaynak olarak iki kitap önerilir. Cormen, Leiserson, Rivest ve Stein'ın *Introduction to Algorithms* kitabı, baş harflerinden dolayı kısaca CLRS diye anılır. Asimptotik gösterimi, en kötü ve ortalama durum analizini ve amortize analizi matematiksel olarak en sağlam biçimde anlatan kaynaktır. Sedgewick'in *Algorithms in C* kitabı ise aynı algoritmaları C koduyla ve çok sayıda ölçümle verir.

1.1.2 Mesleği sevmek

Bir işin iyi yapılması büyük ölçüde o işin sevilmesine bağlıdır. Arkeolog bugün yaşar, ama aklı beş bin beş yüz yıl öncesindedir. Bir mezar odasına bakıp firavunun karısının oraya hangi sebeple gömülmüş olabileceğini tartışabilir. Astronom bir kara deliğin varlığını göstermek için yıllarca fotoğraf çeker ve bir yıldızın görüntüde bir piksel oynamasına sevinir. Bir futbol taraftarı sekiz saat yol gider, yüz elli liralık bilet alır, taş ve sopa tehlikesini göze alır. Aynı yolu bir bilimsel konferans için gideni ise çoğu kişi tuhaf karşılar. Üç örnekte de insanları zahmete katlandıran şey, yaptıkları işe duydukları ilgidir.

Programlamada da durum farklı değildir. Mesleğini seven kişi anlattığı şeyi içtenlikle anlatır ve dinlenir; işi baştan savma yapanı kimse dinlemez. Kod yazarken de kodun içinde yaşıyormuş gibi her satırda ne olup bittiği anlaşılmalıdır. Bir değişkenin o anda hangi değeri taşıdığı, bir işaretçinin hangi bellek hücresini gösterdiği, bir döngünün kaçıncı turda olduğu bilinmelidir. Bu notta kodların yanında verilen izleme tabloları tam olarak bu alışkanlığı kazandırmak içindir. Bir Çin atasözü bunu kısaca söyler: bir ömür mutlu olmanın yolu, işini sevmektir.

1.2 Sistem yaşam döngüsü

Büyük bir yazılım bir oturuşta yazılmaz. Bir fikirden çalışan ve doğruluğuna güvenilen bir programa giden yol birbirini izleyen aşamalardan oluşur. Bu aşamaların bütününe **sistem yaşam döngüsü** (*system life cycle*) denir. Veri yapısı ve algoritma kararları bu döngünün belirli aşamalarında verilir; o yüzden döngüyü bilmek, bu derste öğrenilenlerin gerçek bir projede nereye oturduğunu görmeyi sağlar.

1.2.1 Beş aşama

Horowitz ve Sahni yaşam döngüsünü beş aşamaya ayırır (Şekil 1.2):

1. **Gereksinimler.** Her proje, amacını belirten şartlarla başlar. Programcıya hangi girdilerin verileceği ve hangi çıktıların üretileceği bu aşamada yazılır. Örneğin bir öğrenci işleri programının girdisi öğrenci kayıtları ve notlar, çıktısı da transkriptler ve not ortalamalarıdır.
2. **Çözümleme** (*analysis*). Problem yönetilebilir parçalara ayrılır. Bunun iki yolu vardır: aşağıdan yukarı ve yukarıdan aşağı.
3. **Tasarım.** Problemin veri nesneleri ve bu nesneler üzerindeki işlemler belirlenir. Veri nesnelerinin tanımı bir *soyut veri tipine*, işlemlerin tanımı bir *algoritmaya* karşılık gelir. İkisi de programlama dilinden bağımsızdır. Tasarıma ayrılan zaman arttıkça sonraki aşamalarda çıkan hata azalır.
4. **İnceltme ve kodlama.** Veri nesneleri için gösterimler, yani somut veri yapıları seçilir ve algoritmalar bu yapıların üzerinde kodlanır. Verinin nasıl gösterildiği algoritmanın ne kadar etkin olacağını belirler. Aynı öğrenci listesi bir dizide ya da bir bağlı listede tutulabilir; aramanın ve eklemenin maliyeti bu seçime göre değişir.
5. **Doğrulama.** Programın doğruluğu ispat ve sınamayla gösterilir; bulunan hatalar giderilir. Bu üç etkinliğin her biri başlı başına bir araştırma alanıdır.



Şekil 1.2: Sistem yaşam döngüsünün beş aşaması. Doğrulamada bir hata bulunursa kesikli okla gösterildiği gibi tasarım ya da kodlama kararlarına geri dönülür.

Şekildeki kesikli ok, döngünün tek yönlü bir boru hattı olmadığını gösterir. Doğrulama aşamasında bulunan bir hata kimi zaman yalnız bir satırın düzeltilmesini, kimi zaman bir veri yapısının baştan değiştirilmesini gerektirir. Hata ne kadar erken aşamadan kaynaklanıyorsa düzeltmesi o kadar pahalıdır. Gereksinimlerdeki bir yanlış anlama, kodlamadaki bir yazım hatasından çok daha fazla iş çıkarır.

1.2.2 Çözümleme: aşağıdan yukarı, yukarıdan aşağı

Çözümleme aşamasında problemi parçalamanın iki yolu vardır.

Aşağıdan yukarı (*bottom-up*) yaklaşımda problem doğrudan parçalara bölünür ve her parçaya ayrı ayrı odaklanılır. Modüller baştan görülür; problem modül modül çözülür, sonra parçalar birleştirilir. Bu yol, benzer bir projeyi daha önce yapmış ve hangi modüllerin gerektiğini bilen ekipler için hızlıdır. Proje konusunda deneyim varsa tercih edilir.

Yukarıdan aşağı (*top-down*) yaklaşımda önce bir ana plan yapılır. Amacın ne olduğu ve son ürünün neye benzeyeceği sorulur. Problem bu plana göre alt problemlere, alt problemler de daha küçük parçalara bölünür. Bu yol daha çok zaman alır, ama geri dönüş ve hata olasılığı daha düşüktür. Konuda deneyim yoksa tercih edilir; çünkü parçalar bütünün içindeki yerleri bilinerek tanımlanır.

Örnek 1.1

Bir öğrenci işleri programı yukarıdan aşağı çözümlensin. En üst düzeyde program üç işten oluşur: kayıt yönetimi, not girişi ve raporlama. Raporlama da transkript üretme ile ortalama hesaplama diye ikiye ayrılır. Ortalama hesaplama, bir öğrencinin bütün derslerini bulmayı ve kredilerle ağırlıklı ortalama almayı gerektirir. Her düzeyde parçalar bir üst düzeydeki işe hizmet edecek biçimde tanımlanır. Aşağıdan yukarı çözümlemede ise doğrudan "öğrenci kaydı ekle", "not gir", "ortalama hesapla" gibi modüller yazılır ve sonra bir menüyle birleştirilir.

1.2.3 Tasarım ve gerçeklemeyi ertelemek

Tasarım aşamasında verilen kararlar dilden bağımsızdır. Örneğin öğrenci kütüğünün hangi alanları içereceğine (numara, ad, bölüm, not listesi) bu aşamada karar verilir. Ama bu kütüğün bir dizi olarak mı, bir bağlı liste olarak mı, yoksa bir ağaç olarak mı gerçekleneceği ertelenebilir. Erteleme bir tembellik değil, bilinçli bir tercihtir. Programın hangi işlemleri ne sıklıkla yapacağı netleştikçe en uygun gerçekleme daha iyi seçilir. Kayıtlar çoğunlukla numarayla aranıyorsa sıralı bir dizi ya da bir arama ağacı uygundur. Sık sık ekleme ve silme yapılıyorsa bağlı liste öne çıkar.

Aynı düşünce kodlamanın kendisi için de geçerlidir: kodlama ne kadar geciktirilirse, arka planda o kadar iyi bir yöntem yakalanabilir. Problemi düşünmeden klavyeye oturmak, ilk akla gelen ve çoğu zaman en iyi olmayan yöntemle yola çıkmak demektir.

1.2.4 Kodlama ve doğrulama

Doğrulama dört etkinliği kapsar.

- **Doğruluk ispatı.** Programın doğruluğu matematiksel tekniklerle ispatlanabilir. Küçük bir yordam için bu mümkündür, ama büyük bir projeyi baştan sona ispatlamak neredeyse imkânsızdır. Pratik yol, doğruluğu daha önce ispatlanmış algoritmaları kullanmaktır; böylece hata olasılığı azalır.
- **Sınama** (*testing*). Sınama için çalışan bir kod ve bütün senaryoları kapsayan test verisi gerekir. Boş girdi, tek öğeli girdi, en büyük girdi, sıralı ve ters sıralı girdi, tekrarlı değerler bu senaryoların başında gelir. Derleyicinin sözdizimi hatası vermemesi ve programın bir örnekte doğru çıktı üretmesi, programın doğru olduğunu göstermez.
- **Hata giderme.** İspat ya da testler bir hata gösterirse hata, tasarım ve kodlama kararları göz önüne alınarak ayıklanır. İyi belgelenmiş bir tasarım bu işi büyük ölçüde kolaylaştırır.
- **İşletim zamanı.** Hatasız ama yavaş çalışan bir programın da fazla değeri yoktur. Doğruluk ile verimlilik ayrı sorulardır; ikisine de ayrı ayrı cevap verilmelidir.

Sık yapılan hata

"Derlendi ve çalıştı, o hâlde doğru" diye düşünmek. Bir sıralama programı rastgele on sayıyla denendiğinde doğru sonuç verebilir, ama boş dizide çökebilir ya da tekrarlı değerlerde yanlış sıralayabilir. Test verisi yalnız "olağan" girdiyi değil, uç durumları da kapsamalıdır.

1.3 Algoritma

Tasarım aşamasında işlemler bir algoritma olarak tanımlanır. Günlük dilde “bir işi yapmanın yolu” anlamında kullanılan bu sözcüğün bilgisayar biliminde kesin bir tanımı vardır.

1.3.1 Tanım ve beş ölçüt

Tanım 1.1

Algoritma (*algorithm*), belirli bir görevi yerine getiren, sonlu sayıda komuttan oluşan bir dizidir. Her algoritma şu beş ölçütü sağlar: girdi, çıktı, kesinlik, sonluluk ve etkinlik.

Beş ölçüt Tablo 1.1'de özetlenmiştir.

Ölçüt	Anlamı
Girdi	Dışarıdan sıfır ya da daha fazla nicelik alınır.
Çıktı	En az bir nicelik üretilir.
Kesinlik (*definiteness*)	Her komut açık ve tek anlamlıdır.
Sonluluk (*finiteness*)	Her durumda, sonlu sayıda adımdan sonra sona erer.
Etkinlik (*effectiveness*)	Her komut, kâğıt kalemle yapılabilecek kadar temel ve uygulanabilirdir.

Tablo 1.1: Bir algoritmanın sağlaması gereken beş ölçüt.

Ölçütleri tek tek açalım. **Girdi** ölçütü sıfır girdiye izin verir: “ilk on asal sayıyı yaz” diyen bir algoritma dışarıdan hiçbir şey almaz. **Çıktı** ölçütü ise en az bir sonuç ister; hiçbir şey üretmeyen bir yöntemin işe yarar bir görevi yoktur.

Kesinlik her komutun tek bir biçimde anlaşılmasını ister. “x'e 6 ya da 7 ekle” komutu kesin değildir, çünkü hangisinin ekleneceğini söylemez. “Sayıyı yeterince büyük bir değerle çarp” da kesin değildir. C gibi bir programlama dili bu ölçütü kendiliğinden sağlar: dilin her yapısının anlamı tanımlıdır. Doğal dille yazılan bir algoritmada ise kesinliğe ayrıca dikkat etmek gerekir.

Sonluluk, algoritmanın her girdi için bir noktada durmasını ister. “1'den başla, sayıyı ikiyle çarp, sonuç negatif olunca dur” komutu matematikte hiç durmaz. **Etkinlik** ise her komutun gerçekten yapılabilir olmasını ister. İki tam sayıyı toplamak etkin bir işlemdir. “π sayısının bütün basamaklarını yaz” komutu kesindir, ama sonlu zamanda yapılamadığı için etkin değildir.

1.3.2 Algoritma ile hesaplama yöntemi

Sonluluk ölçütü ilginç bir sonuç doğurur: hiç sonlanmaması gereken programlar algoritma değildir. Bir işletim sistemi bilgisayar açık kaldığı sürece çalışır; bir görevi bekler, onu yürütür ve yeniden beklemeye geçer. Sona ermediği için tanım gereği bir algoritma sayılmaz. Bu tür programlara **hesaplama yöntemi** (*computational procedure*) denir. İşletim sisteminin içindeki parçalar ise, örneğin bir dosyayı diskten okuyan ya da bekleyen işler arasından sıradakini seçen yordam, sona erdikleri için birer algoritmadır.

Bir algoritma doğal dille, bir akış şemasıyla ya da bir programlama diliyle ifade edilebilir. Doğal dil kolay okunur ama kesinliği sağlamakta zorlanır. Akış şeması küçük algoritmalarda açıklayıcıdır, büyük algoritmalarda ise karmaşık bir çizime dönüşür. Bu derste algoritmalar C ile yazılır. Yer yer, bir fikri kodun ayrıntısına girmeden göstermek için C'ye benzeyen kısa bir sözde kod kullanılır.

1.4 Soyut veri tipi

Tasarım aşamasında veri nesneleri dilden bağımsız olarak tanımlanır. Bu tanımın aracı soyut veri tipidir. Soyut veri tipi, bir verinin “ne yaptığını” onun “nasıl yapıldığından” ayırır. Bu ayrım, bu derste gerçeklenecek her yapının (yığın, kuyruk, liste, ağaç, çizge) ortak çerçevesidir.

1.4.1 Veri tipi ve soyut veri tipi

C'deki `int` tipini düşünelim. Bu tip iki şeyden oluşur: belirli bir aralıktaki tam sayılar ve bu tam sayılar üzerinde tanımlı işlemler (`+`, `-`, `*`, `/`, `%`, karşılaştırmalar, atama). Bir `int` değişkenini kullanan programcı, sayının bellekte kaç bitle ve hangi kodlamayla tutulduğunu bilmek zorunda değildir. Toplama işleminin işlemcide nasıl yapıldığını da bilmez. Yalnız işlemlerin ne yaptığını bilir. Genel olarak bir **veri tipi**, nesnelerden ve bu nesneler üzerinde tanımlı işlemlerden oluşan bir topluluktur.

> **Tanım 1.2**
>
> **Soyut veri tipi** (*abstract data type*, ADT), nesnelerin ve işlemlerin **belirtiminin** (*specification*), nesnelerin bellekteki gösteriminden ve işlemlerin **gerçeklemesinden** (*implementation*) ayrıldığı veri tipidir.

Bu tanımda iki kavram karşı karşıya durur.

- **Belirtim, "NE yapılır?" sorusunun cevabıdır.** İşlemlerin adlarını, parametrelerinin tiplerini, sonuçlarının tipini ve her işlemin ne yaptığını söyler. ADT'yi kullanan programcı yalnız bunu bilir.
- **Gerçekleme, "NASIL yapılır?" sorusunun cevabıdır.** Verinin bellekteki gösterimini (bir `struct`, bir dizi, bir bağlı liste) ve işlemleri yapan kodu içerir. Kullanıcıdan gizlenir.

Ayrımın yararı somuttur. Gerçekleme gizli olduğu için, belirtim değişmediği sürece gerçekleme rahatça değiştirilebilir. Dizi ile yazılmış bir yığın daha sonra bağlı listeyle yeniden yazılabilir; yığını kullanan programın tek satırı bile değişmez. Kullanıcı da gerçeklemenin iç ayrıntılarına bağımlı kod yazamaz, çünkü o ayrıntıları göremez. Aynı ADT birden çok veri yapısıyla gerçeklenebilir ve hangisinin seçileceği, yaşam döngüsünün inceltme aşamasına bırakılabilir.

Bir ADT'nin işlemleri görevlerine göre gruplanır:

- **Kurucu** (*constructor*) işlemler türün yeni bir örneğini yaratır.
- **Dönüştürücü** (*transformer*) işlemler var olan örneklerden yeni bir değer üretir.
- **Gözlemci** ya da **raporlayıcı** (*observer, reporter*) işlemler örnek hakkında bilgi verir, onu değiştirmez.

1.4.2 Kesir ADT'sinin belirtimi

Soyut veri tipi kavramını küçük ama eksiksiz bir örnekle somutlaştıralım: kesirler. Bir kesir bir pay ve sıfırdan farklı bir paydadan oluşur. Kesirler üzerinde yaratma, toplama, eşitlik denetimi, yazdırma ve silme işlemleri tanımlansın. Belirtim, dilden bağımsız bir gösterimle şöyle yazılır:

```
ADT Kesir
  nesneler: pay/payda, payda ≠ 0
  işlemler:
    olustur(p, q)  → Kesir   /* kurucu */
    topla(a, b)    → Kesir   /* dönüştürücü */
    esit(a, b)     → Boolean /* gözlemci */
    yaz(a)                   /* raporlayıcı */
    sil(a)
```

Belirtimin ilk satırı nesneleri tanımlar: pay ve paydadan oluşan, paydası sıfır olmayan değerler. Sonraki satırlar işlemleri ve her birinin sonuç tipini verir. `olustur(p, q)` bir kurucudur. `topla(a, b)` iki kesirden yeni bir kesir ürettiği için dönüştürücüdür. `esit(a, b)` bir doğruluk değeri döndüren gözlemcidir; `yaz(a)` ise kesri ekrana yazan raporlayıcıdır. `sil(a)` kesrin kapladığı kaynağı geri verir. Belirtimde kesrin bellekte nasıl tutulacağına dair tek bir sözcük yoktur; olması da gerekmez.

Belirtim aynı zamanda bir **ön koşul** içerir: payda sıfır olamaz. Bu koşul, `olustur` işleminin paydası sıfır olan bir istek aldığında ne yapacağını belirlemeyi gerektirir. Bu soru gerçeklemede cevaplanır.

1.4.3 C'de arayüz: başlık dosyası ve opak tip

C'nin sınıf kavramı yoktur, ama belirtim ile gerçeklemeyi ayırmak için iki araç sunar: **başlık dosyası** (*header file*) ve **eksik tip** (*incomplete type*). Başlık dosyası (`.h`), tip ve fonksiyon bildirimlerini taşır ve `#include` ile başka dosyalara eklenir. Gerçekleme ise ayrı bir kaynak dosyada (`.c`) durur. Kesir ADT'sinin arayüzü şudur:

```c
/* kesir.h : arayuz (NE?) */
#ifndef KESIR_H
#define KESIR_H
typedef struct kesir Kesir; /* opak */

Kesir *kesir_olustur(int pay, int payda);
Kesir *kesir_topla(const Kesir *a,
                   const Kesir *b);
int  kesir_esit(const Kesir *a,
                const Kesir *b);
void kesir_yaz(const Kesir *k);
void kesir_sil(Kesir *k);
#endif
```

Dosyanın satırları şöyle çalışır:

- `#ifndef KESIR_H`, `#define KESIR_H` ve en sondaki `#endif` bir **içerme koruması**dır. Bu üç satır önişlemciye yöneliktir. Dosya aynı derleme biriminde ikinci kez `#include` edilirse `KESIR_H` zaten tanımlı olduğu için içerik atlanır ve aynı bildirimler iki kez yapılmaz.
- `typedef struct kesir Kesir;` satırı iki iş yapar. Önce `struct kesir` adlı bir yapının var olduğunu bildirir, ama alanlarını söylemez. Sonra bu yapıya `Kesir` diye kısa bir ad verir. Alanları bilinmeyen bir yapı **eksik tip**tir: derleyici boyunu bilmez. Bu yüzden `kesir.h`'yi kullanan bir dosya `Kesir` tipinde bir değişken tanımlayamaz, `k->pay` gibi bir alana erişemez. Yalnız `Kesir *` tipinde bir işaretçi tutabilir. Alanları kullanıcıdan gizlenen böyle bir tipe **opak tip** (*opaque type*) denir.
- Beş fonksiyon bildirimi ADT'nin beş işlemine karşılık gelir. Kesirler hep işaretçiyle taşınır. Kesri yalnız okuyan işlemlerin parametreleri `const Kesir *` tipindedir; bu, fonksiyonun gösterilen kesri değiştirmeyeceğine dair derleyicinin denetlediği bir sözdür.

Gerçeklemenin ilk kısmı yapının alanlarını ve kurucuyu içerir:

```c
/* kesir.c : gercekleme (NASIL?) */
#include <stdio.h>
#include <stdlib.h>
#include "kesir.h"

struct kesir { int pay, payda; };

Kesir *kesir_olustur(int pay, int payda) {
  Kesir *k;
  if (payda == 0) return NULL;
  k = malloc(sizeof *k);
  if (k == NULL) return NULL;
  k->pay = pay;
  k->payda = payda;
  return k;
}
```

`struct kesir { int pay, payda; };` satırı yapıyı tamamlar. Bu satır yalnız `kesir.c` içinde göründüğü için alanlar yalnız bu dosyada bilinir. Opak tip yaklaşımı, ADT'nin "gerçekleme gizlidir" ilkesinin C'deki karşılığıdır.

`kesir_olustur` fonksiyonu satır satır şunu yapar:

1. Payda sıfırsa kesir yaratılamaz; fonksiyon `NULL` döndürür. `NULL`, hiçbir nesneyi göstermeyen **boş işaretçi**dir (*null pointer*) ve burada "başarısız" anlamı taşır.
2. `malloc(sizeof *k)` çağrısı, bir `struct kesir` büyüklüğünde bellek ayırır. `malloc`, istenen bayt sayısında bir bellek bloğunu **heap** bölgesinden ayırır ve bloğun başlangıç adresini döndürür. `sizeof *k` ifadesi, `k`'nin gösterdiği tipin, yani `struct kesir`'in boyunu verir. Tip adını yeniden yazmak yerine `*k` kullanmak, ileride `k`'nin tipi değişirse boyun da kendiliğinden doğru kalmasını sağlar. `sizeof` işleneni değerlendirmez; `k` henüz bir değer almamış olsa da ifade güvenlidir.
3. Bellek kalmamışsa `malloc` `NULL` döndürür. Bu durum denetlenir ve fonksiyon da `NULL` döndürür. Denetim yapılmasaydı bir sonraki satır boş işaretçinin gösterdiği yere yazmaya çalışır ve program çökerdi.
4. `k->pay = pay;` satırındaki `->` **ok işlecidir**: işaretçinin gösterdiği yapının alanına erişir. `k->pay`, `(*k).pay` ile aynı anlamdadır.
5. Son olarak yeni kesrin adresi döndürülür.

Fonksiyonun yerel değişkeni `k`, fonksiyon dönünce yok olur. Ama `malloc` ile ayrılan blok yaşamaya devam eder; çünkü heap'teki bir blok, `free` ile geri verilene kadar ayrılmış kalır. Fonksiyonun döndürdüğü adres bu bloğu gösterdiği için çağıran taraf kesri kullanabilir. Bir yerel değişkenin adresi döndürülseydi durum farklı olurdu: fonksiyonun çerçevesi yok olduktan sonra o adres geçersiz kalır.

> **Sık yapılan hata**
>
> `malloc`'un dönüş değerini denetlememek. Küçük deneme programlarında bellek hemen hiç tükenmez ve hata görünmez. Ama büyük girdilerle çalışan ya da uzun süre açık kalan bir programda `malloc` `NULL` döndürebilir. Denetimsiz kod bu durumda `NULL->pay` gibi bir yazmayla çöker. Her `malloc` çağrısından sonra sonuç `NULL` ile karşılaştırılmalıdır.

1.4.4 Gerçeklemenin kalanı

`kesir.c` dosyasının devamında kalan dört işlem yer alır:

```
Kesir *kesir_topla(const Kesir *a,
                   const Kesir *b) {
  return kesir_olustur(
      a->pay * b->payda
        + b->pay * a->payda,
      a->payda * b->payda);
}
int kesir_esit(const Kesir *a,
               const Kesir *b) {
  return a->pay * b->payda
      == b->pay * a->payda;
}
void kesir_yaz(const Kesir *k) {
  printf("%d/%d\n", k->pay, k->payda);
}
void kesir_sil(Kesir *k) { free(k); }
```

`kesir_topla` ortak payda formülünü uygular: $a/b + c/d = (a{\cdot}d + c{\cdot}b) / (b{\cdot}d)$. Sonucu kendisi ayırmaz, `kesir_olustur`'u çağırır. Böylece payda denetimi ve bellek ayırma tek bir yerde kalır. İki payda da sıfırdan farklı olduğu için çarpımları da sıfırdan farklıdır; `kesir_olustur` ancak bellek tükenirse `NULL` döndürür. Gerçekleme sonucu sadeleştirmez: `1/2 + 1/2` işleminin sonucu `4/4` olur. Sadeleştirme istenirse, sonucu en büyük ortak bölene bölen bir satır eklenebilir; bu ekleme arayüzü değiştirmez.

`kesir_esit` iki kesrin eşitliğini çapraz çarpımla denetler: $a/b = c/d$ ancak ve ancak $a{\cdot}d = c{\cdot}b$ ise. Bu yöntem sadeleştirilmemiş kesirlerde de doğru çalışır; `2/4` ile `1/2` eşit bulunur. C'de karşılaştırma işleci `int` tipinde 1 ya da 0 üretir; bu yüzden fonksiyon doğrudan karşılaştırmanın sonucunu döndürür. Belirtimdeki `Boolean`, C'de bu `int` değerle temsil edilir.

`kesir_yaz` kesri `pay/payda` biçiminde yazar. `kesir_sil` ise kesrin bloğunu `free` ile heap'e geri verir. `free`, `malloc` ile ayrılmış bir bloğu sisteme iade eder. Standart C, `free(NULL)` çağrısının hiçbir şey yapmadığını garanti eder; bu yüzden `kesir_sil` boş işaretçiyle de güvenle çağrılabilir.

> **Sık yapılan hata**
>
> Çok büyük pay ve paydalarla çalışmak. `a->pay * b->payda` gibi çarpımlar `int` aralığını aşabilir. İşaretli tam sayı taşması C'de *tanımsız davranıştır*: sonuç rastgele bir değer olabilir, program beklenmedik biçimde davranabilir. Küçük örneklerde sorun görünmez; büyük değerlerle çalışan bir gerçeklemede `long long` kullanmak ya da sonucu her adımda sadeleştirmek gerekir.

1.4.5 Arayüzü kullanmak: sözleşme ve sahiplik

Kesirleri kullanan program yalnız arayüzü bilir:

```c
/* main.c : yalnizca arayuzu bilir */
#include "kesir.h"
#include <stddef.h> /* NULL */
int main(void) {
  Kesir *a = kesir_olustur(1, 2);
  Kesir *b = kesir_olustur(1, 3);
  Kesir *t = NULL;
  if (a && b) t = kesir_topla(a, b);
  if (t) kesir_yaz(t);  /* 5/6 */
  kesir_sil(a); kesir_sil(b);
  kesir_sil(t);  /* free(NULL) zararsiz */
  return 0;
}
```

`main.c` yalnız `kesir.h`'yi ve `NULL` sabiti için `<stddef.h>` başlığını içerir. `struct kesir`'in alanlarını görmez. Program üç dosyadan oluşur ve derlenirken iki kaynak dosya birlikte verilir; örneğin `cc kesir.c main.c`. Derleyici her `.c` dosyasını ayrı derler, bağlayıcı da `main.c`'deki çağrıları `kesir.c`'deki tanımlara bağlar.

Programın akışı Tablo 1.2'de izlenmiştir. Tablodaki adresler, bloğun yerini anlatmak için seçilmiş örnek değerlerdir; gerçek adresler her çalıştırmada farklı olabilir.

Satır	Olay	`a`	`b`	`t`
`a = kesir_olustur(1, 2)`	heap'te {1, 2} bloğu ayrılır	0x2000 → 1/2	—	—
`b = kesir_olustur(1, 3)`	heap'te {1, 3} bloğu ayrılır	0x2000	0x2010 → 1/3	—
`t = NULL`	henüz sonuç yok	0x2000	0x2010	NULL
`if (a && b)`	ikisi de NULL değil: doğru	0x2000	0x2010	NULL
`t = kesir_topla(a, b)`	pay 1·3 + 1·2 = 5, payda 2·3 = 6; yeni blok	0x2000	0x2010	0x2020 → 5/6
`if (t) kesir_yaz(t)`	ekrana `5/6` yazılır	0x2000	0x2010	0x2020
`kesir_sil(a); kesir_sil(b); kesir_sil(t)`	üç blok heap'e döner	geçersiz	geçersiz	geçersiz

Tablo 1.2: `main.c` programının izlemesi (adresler örnektir).

Programın son satırlarından sonra `a`, `b` ve `t` değişkenleri hâlâ eski adresleri taşır, ama bu adresler artık geçerli bir nesneyi göstermez. Böyle işaretçilere **sarkan işaretçi** (*dangling pointer*) denir. Onları kullanmak tanımsız davranıştır. Program hemen sona erdiği için burada bir sorun çıkmaz.

Arayüzü güvenle kullanmak için iki kural açıkça yazılmalıdır.

- **Sözleşme.** `kesir_olustur`, payda sıfırsa ya da bellek kalmamışsa `NULL` döndürür. Çağıran taraf bu sonucu denetlemekle yükümlüdür. `main`'deki `if (a && b)` ve `if (t)` denetimleri bu sözleşmenin gereğidir. Denetim yapılmasaydı `kesir_topla(a, b)` boş bir işaretçinin alanını okumaya çalışabilirdi.
- **Sahiplik.** `kesir_olustur` ve `kesir_topla`'nın döndürdüğü her kesir çağırana aittir. Çağıran, kesirle işi bitince onu `kesir_sil` ile bırakmalıdır. Bırakılmayan blok program bitene kadar heap'te kalır. Artık hiçbir işaretçiyle erişilemeyen ama geri verilmemiş belleğe **bellek sızıntısı** (*memory leak*) denir. Döngü içinde her turda bir kesir yaratıp silmeyi unutan bir program, bellek bitene kadar büyür.

`t`'nin başlangıçta `NULL` yapılmasının nedeni de sahiplik kuralıdır. `a` ya da `b` yaratılamazsa `kesir_topla` hiç çağrılmaz ve `t` `NULL` kalır. Programın sonundaki `kesir_sil(t)` çağrısı bu durumda `free(NULL)`'a dönüşür ve zararsızdır. Başlangıç değeri verilmeseydi `t` rastgele bir adres taşıyabilir ve `free` çağrısı programı çökertebilirdi.

> **Sık yapılan hata**
>
> `main.c` içinde `a->pay` yazmaya ya da `Kesir k;` diye bir değişken tanımlamaya çalışmak. İkisi de derleme hatası verir, çünkü `main.c` için `struct kesir` eksik bir tiptir. Bu hata bir kusur değil, opak tipin amacıdır: kullanıcı yalnız arayüzdeki fonksiyonlarla çalışmaya zorlanır. Bir başka hata da aynı kesri iki kez silmektir. İkinci `free`, artık ayrılmış olmayan bir bloğu geri vermeye çalışır;

bu da tanımsız davranıştır.

1.5 Seçimli sıralama

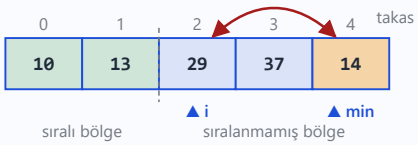
Algoritma kavramını ve C ile yazılışını ilk kez somut bir problem üzerinde görelim: bir tam sayı dizisini küçükten büyüğe sıralamak. **Sıralama** (*sorting*), öğeleri anahtarlarına göre artan ya da azalan düzene koymaktır. Bu problemin pek çok çözümü vardır. Bu kesimde en kolay anlaşılanlardan biri olan seçimli sıralama, önce bir fikir olarak, sonra sözde kodla, en sonunda da C koduyla verilir. Bir fikrin adım adım koda dönüşmesi, algoritma tasarımının olağan yoludur.

1.5.1 Fikir

n öğeli sıralı olmayan bir `list` dizisi verilsin. **Seçimli sıralama** (*selection sort*) şu fikre dayanır: dizinin sıralanmamış kısmındaki en küçük öğe bulunur ve bu kısmın başına konur. Sonra sıralanmamış kısım bir öğe kısalır ve aynı iş tekrarlanır. İlk turda bütün dizinin en küçüğü `list[0]` 'a, ikinci turda kalanların en küçüğü `list[1]` 'e yerleşir. Genel olarak *i*. turda `list[i]` … `list[n-1]` aralığının en küçüğü `list[i]` 'ye taşınır. Fikir C'ye benzeyen bir sözde kodla şöyle yazılır:

```
for (i = 0; i < n; i++) {
    list[i] .. list[n-1] arasini incele;
    en kucuk tamsayi list[min] olsun;
    list[i] ile list[min]'i takas et;
}
```

Bu sözde kodun ikinci ve üçüncü satırları henüz C değildir; “aralığı incele” ve “en küçük öğe bulunsun” gibi ifadeler bir C derleyicisinin anlayacağı komutlar değildir. Ama fikir açıktır ve bu satırların her biri kolayca C'ye çevrilir. En küçük öğeyi bulmak için aralık soldan sağa taranır; o ana kadar görülen en küçüğün indisi bir `min` değişkeninde tutulur. Takas işlemi de üç atamayla yapılır.



Şekil 1.3: Seçimli sıralamanın bir ara durumu. Yeşil hücreler (10, 13) sıralı bölgedir ve son yerlerindedir. `i = 2` için sıralanmamış bölgede (29, 37, 14) en küçük öğe 14'tür ve indis 4'te durur; `list[2]` ile `list[4]` takas edilir.

Şekil 1.3, algoritmanın her turda koruduğu durumu gösterir. Dizi kesikli çizgiyle iki bölgeye ayrılır. Soldaki **sıralı bölge** dizinin en küçük *i* öğesini sıralı biçimde tutar; bu öğeler bir daha yer değiştirmez. Sağdaki **sıralanmamış bölge** kalan öğeleri tutar. Her tur sıralı bölgeyi bir öğe büyütür. Bu durum her turun başında doğrudur ve algoritmanın doğruluğunun gerekçesidir. *n − 1* tur sonunda sıralı bölgede en küçük *n − 1* öğe vardır; kalan tek öğe de en büyük olduğu için son hücrede durur. Dizi sıralanmıştır.

Yazılmış bir algoritma olduğu gibi kabul edilmek zorunda değildir. Onu anlamak, zayıf yanlarını görmek ve gerekirse daha iyisini önerebilmek, bu dersin ulaşmak istediği düzeydir. Seçimli sıralama da bu gözle incelenecektir: kaç karşılaştırma yapar, kaç takas yapar, girdinin sıralı olması işini kolaylaştırır mı?

1.5.2 C gerçeklemesi

Önce sıralama fonksiyonunu kullanan ana program:

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 101
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
void sort(int list[], int n);

int main(void) {
  int i, n, list[MAX_SIZE];
  printf("Dizi boyunu giriniz: ");
  if (scanf("%d", &n) != 1 || n < 1 || n > MAX_SIZE) {
    printf("Hatali n degeri!\n");
    return 1;
  }
  for (i = 0; i < n; i++) list[i] = rand() % 100;
  sort(list, n);
  for (i = 0; i < n; i++) printf("%d ", list[i]);
  return 0;
}
```

Ana program satır satır şöyle çalışır:

- `#define MAX_SIZE 101` dizinin en büyük boyunu bir sabitle verir. Sabit değiştirilince dizinin boyu da, sınır denetimi de kendiliğinden değişir.
- `#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))` bir **makrodur**. Makro, `#define` ile tanımlanan ve derlemeden önce metin olarak yerine konan bir addır. `SWAP(list[i], list[min], temp)` yazılan yere önişlemci `((temp) = (list[i]), (list[i]) = (list[min]), (list[min]) = (temp))` metnini koyar. Virgül işleci üç atamayı soldan sağa sırayla yapar.
- `void sort(int list[], int n);` bir **fonksiyon bildirimi**dir (prototip). `sort`'un tanımı `main`'den sonra geldiği için derleyiciye önceden fonksiyonun parametre tipleri ve dönüş tipi bildirilir.
- `scanf("%d", &n)` kullanıcıdan dizinin boyunu okur. `&n`, `n`'nin adresidir; `scanf` okuduğu değeri bu adrese yazar. `scanf` başarıyla okuduğu öğe sayısını döndürür. Kullanıcı sayı yerine harf girerse 1 dışında bir değer döner.
- Koşul üç durumu reddeder: okuma başarısız, `n < 1` ya da `n > MAX_SIZE`. Son denetim önemlidir. `list` dizisinde `MAX_SIZE` hücre vardır; daha büyük bir `n` kabul edilseydi döngüler dizinin sonundan sonraki belleğe yazardı. C bu tür taşmaları denetlemez.
- `rand() % 100` 0 ile 99 arasında sözde rastgele bir sayı üretir. `srand` çağrılmadığı için program her çalıştırmada aynı diziyi üretir; bu, sonuçların tekrarlanabilmesi için bir kolaylıktır.
- `sort(list, n)` çağrısında dizinin kendisi değil, ilk öğesinin adresi gönderilir. `sort` içindeki `list[i]` ifadesi bu adresten başlayan belleğe erişir; bu yüzden `sort`'un dizide yaptığı değişiklikler `main`'deki diziyi değiştirir.

Sıralama fonksiyonu sözde kodun doğrudan C karşılığıdır:

```c
void sort(int list[], int n) {
  int i, j, min, temp;
  for (i = 0; i < n - 1; i++) {
    min = i;
    for (j = i + 1; j < n; j++)
      if (list[j] < list[min])
        min = j;
    SWAP(list[i], list[min], temp);
  }
}
```

- Dış döngü `i = 0`'dan `n - 2`'ye kadar döner. Sözde kodda döngü `n` kez dönüyordu; son tur gereksizdir, çünkü tek öğelik aralık zaten sıralıdır. Bu yüzden döngü koşulu `i < n - 1` olarak yazılır.
- `min = i;` aralığın ilk öğesini geçici olarak en küçük kabul eder.
- İç döngü `j = i + 1`'den `n - 1`'e kadar aralığın kalan öğelerini tarar. `list[j] < list[min]` ise daha küçük bir öğe bulunmuştur ve `min` güncellenir. İç döngü bittiğinde `min`, aralığın en küçük öğesinin indisidir.
- `SWAP(list[i], list[min], temp)` bu öğeyi aralığın başına taşır. `min` zaten `i`'ye eşitse takas bir öğeyi kendisiyle değiştirir; sonuç değişmez, yalnız üç gereksiz atama yapılır.

Program `n = 10` girdisiyle derlenip çalıştırıldığında rastgele üretilen on sayıyı sıralı olarak yazar. Bu notun hazırlandığı ortamda çıktı `0 24 34 41 58 62 64 67 69 78` oldu. `rand()` 'ın ürettiği dizi C kütüphanesine bağlıdır; başka bir derleyicide sayılar farklı çıkar, ama hep artan sırada yazılır.

1.5.3 Kod izleme: {29, 10, 14, 37, 13}

`sort` fonksiyonunu beş öğeli {29, 10, 14, 37, 13} dizisiyle izleyelim. Tablo 1.3 her turda iç döngünün bulduğu `min` değerini ve takastan sonraki diziyi verir.

`i`	İç döngünün karşılaştırmaları	`min`	Takas	Dizi
0	10 < 29 evet (min = 1); 14 < 10 hayır; 37 < 10 hayır; 13 < 10 hayır	1	list[0] ↔ list[1]	{**10**, 29, 14, 37, 13}
1	14 < 29 evet (min = 2); 37 < 14 hayır; 13 < 14 evet (min = 4)	4	list[1] ↔ list[4]	{**10**, **13**, 14, 37, 29}
2	37 < 14 hayır; 29 < 14 hayır	2	kendisiyle (değişmez)	{**10**, **13**, **14**, 37, 29}
3	29 < 37 evet (min = 4)	4	list[3] ↔ list[4]	{**10**, **13**, **14**, **29**, 37}

Tablo 1.3: `sort` fonksiyonunun {29, 10, 14, 37, 13} dizisi üzerinde izlemesi.

Birinci turda en küçük öğe 10 indis 1'de bulunur ve 29 ile yer değiştirir. İkinci turda tarama önce 14'ü, sonra 13'ü en küçük kabul eder; 13, indis 1'deki 29 ile takas edilir. Üçüncü turda 14 zaten aralığın en küçüğüdür; `SWAP` çalışır ama bir şey değiştirmez. Dördüncü ve son turda 29 ile 37 yer değiştirir. Dış döngü `i = 4` olduğunda `i < n - 1` koşulu yanlış çıkar ve fonksiyon biter. Toplam *4 + 3 + 2 + 1 = 10* karşılaştırma yapılmış, `SWAP` dört kez çağrılmış, bunların üçünde iki farklı hücre gerçekten yer değiştirmiştir.

1.5.4 Analiz

Seçimli sıralamanın yaptığı karşılaştırma sayısı girdinin değerlerine bağlı değildir; yalnız n'ye bağlıdır. i. turda iç döngü $j = i + 1, \ldots, n - 1$ değerlerini alır, yani $n - 1 - i$ karşılaştırma yapar. Bütün turların toplamı:

> $$(n - 1) + (n - 2) + \ldots + 1 = n(n - 1) / 2$$

Bu toplam, ilk $n - 1$ doğal sayının toplamıdır. Formül şöyle doğrulanır: toplamı bir kez baştan, bir kez sondan yazıp terim terim toplayınca her çift n eder ve $n - 1$ çift vardır; toplamın iki katı $n(n - 1)$'dir. $n = 5$ için $5 \cdot 4 / 2 = 10$ karşılaştırma bulunur; izleme tablosundaki sayı da budur. $n = 1000$ için yaklaşık yarım milyon karşılaştırma gerekir. Karşılaştırma sayısı n^2 ile orantılı büyüdüğü için seçimli sıralamanın zamanı $O(n^2)$'dir. $O(n^2)$ yazımı, büyük n için adım sayısının n^2'nin sabit bir katını aşmadığını anlatır.

Takas sayısı ise azdır: her turda tek bir `SWAP` yapılır, toplam $n - 1$ kez. Bu yüzden takas en fazla $n - 1$ kez gerçekleşir. Öğeler büyük kayıtlarsa ve taşınmaları pahalıysa bu özellik seçimli sıralamayı çekici kılar. Algoritma ek bellek olarak yalnız `i`, `j`, `min` ve `temp` değişkenlerini kullanır. Dizi kendi yerinde sıralanır.

1.5.5 Hatalı bir sort() sürümü

Seçimli sıralama yazılmak istendiğinde sık rastlanan bir kod şudur:

```
void sort(int list[], int n) {
  int i, j, min, temp;
  for (i = 0; i < n; i++) {
    min = i;
    for (j = 0; j < n; j++)
      if (list[j] > list[min]) {
        min = j;
        SWAP(list[i], list[min], temp);
      }
  }
}
```

Bu kod şaşırtıcı biçimde diziyi yine sıralar. Bu notu hazırlarken $n \le 7$ için bütün permütasyonlar denendi; hepsinde sonuç sıralı çıktı. Ama kod sözde koddaki seçimli sıralama değildir. Üç fark vardır:

1. İç döngü `j = i + 1` 'den değil `j = 0` 'dan başlar; her turda bütün diziyi tarar.

2. Koşul `<` değil `>`'dir; en küçük değil, o ana kadarki en büyük öğe aranır.
3. Takas iç döngüden sonra bir kez değil, koşul her doğru çıktığında iç döngünün içinde yapılır.

Sonuç doğru olsa da maliyet çok farklıdır. Tablo 1.4, iki sürümün aynı girdilerde yaptığı karşılaştırma ve takas sayılarını verir. Değerler iki sürüme sayaç eklenerek ölçülmüştür. Takas sütunu yalnız iki farklı hücrenin gerçekten yer değiştirdiği `SWAP`'ları (`i ≠ min`) sayar. Rastgele satır, `rand() % 100` ile üretilen yüz öğeli bir örnek diziden alınmıştır; başka bir rastgele dizide takas sayıları biraz değişir, karşılaştırma sayıları değişmez.

Girdi	Sürüm	Karşılaştırma	Takas
{29, 10, 14, 37, 13}	hatalı	25	7
	seçimli	10	3
100 rastgele	hatalı	10000	2561
	seçimli	4950	94
500, 499, ..., 1 (azalan)	hatalı	250000	124750
	seçimli	124750	250

Tablo 1.4: Hatalı sürüm ile seçimli sıralamanın karşılaştırma ve takas sayıları.

Hatalı sürüm her turda n karşılaştırma yapar ve n tur döner: toplam n^2. Bu, seçimli sıralamanın $n(n-1)/2$ karşılaştırmasının iki katından fazladır. Asıl fark takaslardadır. Azalan 500 öğeli dizide hatalı sürüm 124.750 takas yapar; seçimli sıralama yalnız 250. Azalan dizide seçimli sıralamanın her takası iki öğeyi birden son yerine koyar. Örneğin ilk takas 1 ile 500'ü yer değiştirir. Bu yüzden 250 takas yeterlidir. Bu girdide hatalı sürümün takas sayısı da ters sıradaki öğe çiftlerinin sayısına, yani $500 \cdot 499 / 2 = 124.750$'ye eşit çıkar.

Bu örnek önemli bir dersi gösterir: bir programın doğru sonuç vermesi, istenen algoritmayı gerçeklediğini göstermez. Algoritma analizi, sonucun yanında o sonuca ulaşmanın maliyetini de sorar. Doğru sürüm, yani `j = i + 1`'den başlayan ve takası iç döngüden sonra yapan sürüm, bu bölümde kullanılan sürümdür.

1.5.6 SWAP: makro mu, fonksiyon mu?

İki değişkenin değerlerini yer değiştirme işlemine **takas** (*swap*) denir. Takas C'de iki biçimde yazılabilir. Birincisi, programda kullanılan makrodur:

```
#define SWAP(x, y, t) \
        ((t) = (x), (x) = (y), (y) = (t))

int temp;
SWAP(list[i], list[min], temp);
```

Satır sonundaki ters bölü (`\`), makro tanımının bir sonraki satırda sürdüğünü önişlemciye bildirir. Makronun yararı, bir fonksiyon çağrısı maliyeti taşımamasıdır. Önişlemci makro metnini çağrının yerine koyar; çalışma anında bir çağrı, parametre kopyalama ya da dönüş olmaz. Buna karşılık geçici değişken `t` makroya dışarıdan verilmelidir; bu yüzden `sort` içinde `temp` ayrıca tanımlanır.

İkinci yol bir fonksiyondur:

```
void swap(int *x, int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}

swap(&list[i], &list[min]);
```

Fonksiyonun parametreleri tam sayı değil, tam sayıyı gösteren işaretçidir (`int *`). C'de parametreler **değerle** geçirilir: fonksiyon, gönderilen değerin bir kopyasını alır. `swap(int x, int y)` diye yazılmış bir fonksiyon yalnız kendi kopyalarını takas eder; çağıran taraftaki değişkenler değişmez. Bu yüzden değişkenlerin kendileri değil, adresleri gönderilir: `swap(&list[i], &list[min])`. Fonksiyonun içinde `*x`

ifadesi, `x`'in gösterdiği yerdeki değeri verir. `*` burada **içerik işlecidir** (*dereference*). `*x = *y` ataması da `y`'nin gösterdiği değeri `x`'in gösterdiği yere yazar.

Fonksiyonun yararı, derleyicinin tip denetimi yapmasıdır: `swap`'a `int` olmayan bir adres verilirse derleyici uyarır. Geçici değişken `temp` fonksiyonun yerel değişkenidir; çağıranın bir şey tanımlaması gerekmez. Fonksiyonun adı `swap` olarak küçük harfle verilmiştir. Fonksiyona da `SWAP` adı verilseydi önişlemci her `SWAP(` görünümünü makro olarak açmaya çalışır ve iki tanım çakışırdı.

Makronun gizli bir tehlikesi vardır: argümanlarını birden çok kez değerlendirir. `SWAP(a[i++], a[j], t)` yazıldığında önişlemci şu metni üretir:

```
((t) = (a[i++]), (a[i++]) = (a[j]), (a[j]) = (t))
```

`i++` iki kez işletilir. `a = {10, 20, 30, 40}`, `i = 0` ve `j = 3` ise önce `t = a[0] = 10` olur ve `i` 1'e çıkar. Sonra `a[1] = a[3] = 40` yazılır ve `i` 2'ye çıkar. En sonunda `a[3] = 10` olur. Sonuç `{10, 40, 30, 10}` ve `i = 2`'dir: hiçbir takas yapılmamış, 20 değeri kaybolmuştur. Aynı çağrı `swap(&a[i++], &a[j])` biçiminde fonksiyonla yapılsaydı `i++` bir kez işletilir ve `a[0]` ile `a[3]` doğru biçimde takas edilirdi. Bu sonuçlar, örnek derlenip çalıştırılarak doğrulanmıştır.

> **Sık yapılan hata**
>
> Makroya yan etkili bir argüman vermek: `SWAP(a[i++], a[j], t)`, `SWAP(x, f(), t)` gibi. Makro bir fonksiyon değildir; argümanlar metin olarak kopyalanır ve her geçtikleri yerde yeniden değerlendirilir. Makro argümanları yalın değişkenler ya da yan etkisiz ifadeler olmalıdır. Makro tanımındaki her parametrenin parantez içine alınması da bu yüzdendir: `SWAP(a[k], b[k + 1], t)` gibi ifadeler doğru öbeklenir.

1.5.7 swap() bellekte adım adım

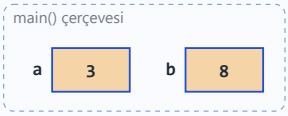
İşaretçiyle parametre geçirmenin bellekte nasıl çalıştığını küçük bir programla izleyelim:

```c
void swap(int *x, int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}

int main(void) {
  int a = 3, b = 8;
  swap(&a, &b);
  printf("%d %d\n", a, b);
}
```

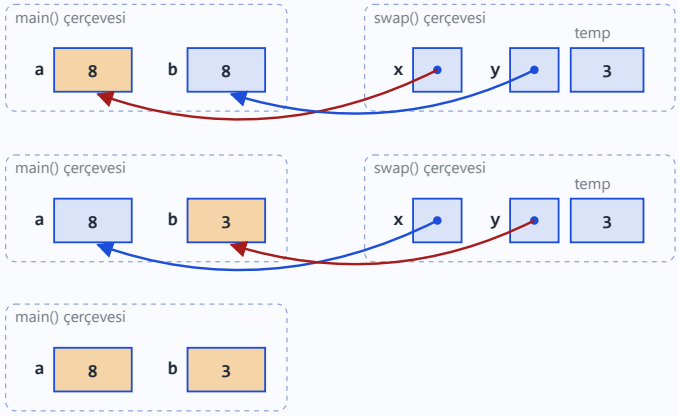
Program çalışınca `8 3` yazar. Bu sonuca nasıl ulaşıldığını görmek için belleği adım adım izleyelim. Her fonksiyon çağrısı, **çağrı yığını** (*call stack*) üzerinde bir **çerçeve** (*stack frame*) açar. Çerçeve, o çağrının parametrelerini ve yerel değişkenlerini tutar; çağrı bitince yok olur.

1. **Başlangıç.** `main`'in çerçevesinde `a = 3` ve `b = 8` vardır (Şekil 1.4a).
2. **Çağrı.** `swap(&a, &b)` çağrılınca `swap` için yeni bir çerçeve açılır. Parametrelere değerler değil adresler kopyalanır: `x`, `a`'yı; `y`, `b`'yi gösterir. `temp` henüz bir değer almamıştır (Şekil 1.4b).
3. `int temp = *x;` `x`'in gösterdiği yerdeki değer, yani 3 okunur ve `temp`'e yazılır.
4. `*x = *y;` `y`'nin gösterdiği değer (8), `x`'in gösterdiği yere, yani `main`'deki `a`'ya yazılır. Şu an `a = 8`, `b = 8`'dir (Şekil 1.5a).
5. `*y = temp;` `temp`'teki 3, `y`'nin gösterdiği `b`'ye yazılır. Takas `main`'in kendi değişkenlerinde gerçekleşmiştir (Şekil 1.5b).
6. **Dönüş.** `swap`'ın çerçevesi yok olur; `x`, `y` ve `temp` artık yoktur. Ama yaptıkları değişiklik `main`'in çerçevesinde kalır: `a = 8`, `b = 3` (Şekil 1.5c).





Şekil 1.4: `swap(&a, &b)` çağrısından önce ve hemen sonra bellek. (a) `main`'in çerçevesinde `a = 3`, `b = 8`. (b) `swap`'ın çerçevesi açılmış; `x` ve `y` sırasıyla `a` ile `b`'yi gösteriyor.



Şekil 1.5: Takasın son adımları. (a) `*x = *y` sonrası: `a` 8 oldu, `temp` eski değeri (3) saklıyor. (b) `*y = temp` sonrası: `b` 3 oldu. (c) `swap` döndü; çerçevesi yok oldu, ama `a = 8`, `b = 3` kaldı.

Aynı izleme Tablo 1.5'te değişken değerleriyle verilmiştir. Tabloda adresler örnek olarak `a` için 1000, `b` için 1004 seçilmiştir; bir `int` dört bayt yer kaplar.

Adım	`a` (1000)	`b` (1004)	`x`	`y`	`temp`
çağrıdan önce	3	8	—	—	—
çağrı	3	8	1000	1004	?
`temp = *x`	3	8	1000	1004	3
`*x = *y`	**8**	8	1000	1004	3
`*y = temp`	8	**3**	1000	1004	3
dönüş	8	3	—	—	—

Tablo 1.5: `swap(&a, &b)` çağrısının izlemesi (adresler örnektir).

Sık yapılan hata

`swap(a, b)` yazmak, yani adres yerine değer göndermek. Fonksiyon `int *` beklediği için derleyici bir uyarı ya da hata verir; uyarı göz ardı edilirse `x` 3 adresini, `y` 8 adresini gösterir ve `*x` geçersiz bir belleğe erişir. Program büyük olasılıkla çöker. Fonksiyonu `void swap(int x, int y)` diye yazmak ise sessiz bir hatadır: program derlenir ve çalışır, ama `main`'deki değerler değişmez.

1.6 Performans analizi

Seçimli sıralamanın karşılaştırma sayısını hesaplarken bir programın maliyetini, onu çalıştırmadan, yalnız koda bakarak bulduk. Bu kesim bu yaklaşımı genelleştirir. Bir programın ne kadar bellek ve zaman gerektirdiği, girdinin büyüklüğüne bağlı bir fonksiyon olarak hesaplanır.

1.6.1 Bir programı değerlendirmek

Bir programın iyi olup olmadığına karar vermek için şu sorular sorulur:

1. Programdan isteneni karşılıyor mu?
2. Doğru çalışıyor mu?
3. Nasıl işletileceği ve kullanılacağı, açıklama satırları da dâhil, belgelenmiş mi?
4. Mantıksal sonuç üretmek için fonksiyonları etkili kullanıyor mu?
5. Kod okunabilir mi?
6. Ana ve ikincil belleği etkili kullanıyor mu?
7. İşletim zamanı kabul edilebilir mi?

Son iki soru performans değerlendirmesinin konusudur. İlk beşi de en az onlar kadar önemlidir: hızlı ama yanlış sonuç veren, belgesiz ve okunamayan bir program bakımı yapılamadığı için kısa sürede değerini yitirir. Ancak ilk beş sorunun cevabı büyük ölçüde deneyim ve özen işidir. Son iki soru ise sayılarla cevaplanabilir.

Performans değerlendirmesi iki yolla yapılır.

- **Performans analizi** (*performance analysis*) *makineden bağımsızdır*. Program çalıştırılmaz; zaman ve bellek gereksinimi koda bakılarak tahmin edilir. Sonuç bir karmaşıklık fonksiyonudur.
- **Performans ölçümü** (*performance measurement*) *makineye bağlıdır*. Program belirli bir makinede, belirli bir derleyiciyle çalıştırılır ve gerçek işletim süresi ölçülür.

İki yol birbirini tamamlar. Analiz, bir algoritmanın girdi büyüdükçe nasıl davranacağını önceden söyler. Ölçüm ise bu tahminin gerçek bir makinede tutup tutmadığını gösterir ve analizin göz ardı ettiği sabitleri ortaya çıkarır.

1.6.2 Alan karmaşıklığı

Tanım 1.3

Bir P programının **alan karmaşıklığı** (*space complexity*), programın işini tamamlaması için gereken bellek miktarıdır:

> $S(P) = c + S_P(I)$

Burada c, girdinin ve çıktının büyüklüğünden bağımsız **sabit kısımdır**; $S_P(I)$ ise I girdisinin özelliklerine bağlı **değişken kısımdır**.

Sabit kısım programın kodunu, basit değişkenleri, sabit boyutlu yapıları ve sabitleri kapsar. Bu bellek, girdi ne olursa olsun aynıdır. Değişken kısım ise girdiye bağlı olarak değişen bellektir: boyu girdiye göre ayrılan diziler, özyinelemeli çağrıların açtığı çerçeveler, `malloc` ile girdiye göre ayrılan bloklar. Girdinin bu kısmı en çok etkileyen özelliği genellikle boyutudur: dizinin öğe sayısı, matrisin satır ve sütun sayısı. Çözümlemede yalnız değişken kısımla ilgilenilir, çünkü programlar arasındaki farkı girdi büyüdükçe o belirler.

Örnek 1.2

Üç sayıdan bir ifade hesaplayan şu fonksiyonu ele alalım:

```
float abc(float a, float b, float c) {
  return a + b + b * c
         + (a + b - c) / (a + b) + 4.00;
}
/* yalniz sabit bellek: S_abc(I) = 0 */
```

Fonksiyon üç `float` parametre alır ve tek bir değer döndürür. Girdinin büyüklüğü diye bir şey yoktur; her çağrıda aynı üç sayı ve aynı dönüş adresi saklanır. Bütün bellek sabit kısımdadır: $S_{abc}(I) = 0$. `abc(1, 2, 3)` çağrısı $1 + 2 + 2 \cdot 3 + (1 + 2 - 3)/(1 + 2) + 4 = 13$ sonucunu verir.

Bellek kullanımı konusunda bir alışkanlık baştan kazanılmalıdır: kod, her şey kısıtlıymış gibi yazılmalıdır. Bugünün bilgisayarlarında bellek bol görünür, ama aynı kod bir gömülü sistemde, bir telefonda ya da milyonlarca kaydı aynı anda işleyen bir sunucuda çalışabilir. Bellek nasıl olsa bol diye boşa bellek harcanmaz.

1.6.3 sum ve rsum'un belleği

Bir dizinin öğelerini toplayan iki fonksiyonu karşılaştıralım. `sum` bir döngü, `rsum` ise özyineleme kullanır:

```
float sum(float list[], int n) {
  float tempsum = 0;
  int i;
  for (i = 0; i < n; i++)
    tempsum += list[i];
  return tempsum;
}

float rsum(float list[], int n) {
  if (n)
    return rsum(list, n - 1) + list[n - 1];
  return 0;
}
```

Özyineleme (*recursion*), bir fonksiyonun kendini daha küçük bir girdiyle çağırmasıdır. `rsum(list, n)`, ilk *n* öğenin toplamını şöyle tanımlar: *n = 0* ise toplam 0'dır; değilse ilk *n − 1* öğenin toplamına `list[n-1]` eklenir. Kendini çağırmadan doğrudan sonuç veren *n = 0* durumuna **taban durum** (*base case*) denir. Her çağrı *n*'yi bir azalttığı için zincir sonunda taban duruma ulaşır.

İki fonksiyonun değişken belleğini bulalım.

- `sum`'a dizinin öğeleri değil, **ilk öğesinin adresi** gönderilir. C'de bir dizi parametresi aslında bir işaretçidir; `float list[]` ile `float *list` parametre olarak aynı anlamdadır. Bu yüzden dizi ne kadar büyük olursa olsun `sum` yalnız bir adres, bir `int` ve iki yerel değişken saklar: $S_{sum}(n) = 0$. Öğeler değerle kopyalansaydı, örneğin Pascal'daki gibi bir dizi değer parametresi olarak geçirilseydi, $S_{sum}(n) = n$ olurdu.
- `rsum(list, n)` toplam *n + 1* kez çağrılır: *n, n − 1, ..., 1, 0* değerleri için birer kez. Her çağrı, yığında bir çerçeve açar ve bu çerçeveler taban duruma ulaşılıncaya kadar birlikte yaşar. Bu yüzden değişken bellek *n* ile orantılıdır: $S_{rsum}(n) \propto n$.



Şekil 1.6: `rsum(list, 3)` çağrısının en derin anında sistem yığını. Dört çerçeve üst üste durur; her biri `list` işaretçisini, `n`'yi ve dönüş adresini saklar.

Şekil 1.6 bir çerçevenin içeriğini gösterir: dizinin adresi (`list`), `n` parametresi ve fonksiyon bitince programın nereden devam edeceğini gösteren dönüş adresi. Ders kitabının varsaydığı 16 bitlik bir makinede bu üç değerin her biri 2 bayttır ve bir çerçeve 6 bayt tutar. *n = 1000* için 1001 çerçeve, yani en az 6006 bayt gerekir. 64 bitlik bugünkü makinelerde adresler 8 bayttır ve derleyiciler çerçeveye hizalama ve yazmaç saklama alanı da ekler. Bir çerçeve tipik olarak en az 24 bayttır, çoğu zaman daha fazlası.

Bu farkın pratik bir sonucu vardır. Çağrı yığını için ayrılan bellek sınırlıdır; Windows'ta varsayılan değer 1 MB civarındadır. `rsum` çok büyük bir *n* ile çağrılırsa çerçeveler bu sınırı aşar ve program **yığın taşması** (*stack overflow*) ile çöker. Bu notu hazırlarken yapılan denemede `rsum`, *n = 100.000* için doğru sonucu verdi, *n = 1.000.000* için ise program çöktü. Aynı girdiyle `sum` sorunsuz çalıştı. Sınırın tam yeri derleyiciye, işletim sistemine ve derleme seçeneklerine göre değişir.

1.6.4 Zaman karmaşıklığı ve program adımı

Bir *P* programının harcadığı zaman *T(P)*, derleme zamanı ile işletim zamanının toplamıdır. Derleme bir kez yapılır ve program sonra defalarca çalıştırılabilir. Bu yüzden ilgilenilen yalnız işletim zamanıdır; bu zaman t_P ile gösterilir.

İşletim zamanı, programın yaptığı her temel işlemin sayısı ile o işlemin süresinin çarpımlarının toplamı olarak yazılabilir:

> $t_P(n) = c_a \cdot ADD(n) + c_s \cdot SUB(n) + c_l \cdot LDA(n) + c_{st} \cdot STA(n) + \ldots$

Burada *ADD(n)*, *n* boyutlu bir girdi için yapılan toplama sayısını, c_a ise bir toplamanın süresini gösterir. Çıkarma (SUB), bellekten yükleme (LDA) ve belleğe yazma (STA) için de birer terim vardır. Formül kesindir ama pratikte kullanışsızdır. *c* sabitleri derleyiciye, işlemciye ve önbelleğin durumuna bağlıdır ve önceden bilinmesi zordur. Bir derleyicinin bir C deyimini kaç makine komutuna çevireceği de belli değildir.

Bu yüzden zaman, makine komutları yerine **program adımlarıyla** ölçülür.

> **Tanım 1.4**
>
> **Program adımı** (*program step*), işletim süresi girdinin boyutundan bağımsız olan, sözdizimi ya da anlam bakımından anlamlı bir program parçasıdır. Bir programın **zaman karmaşıklığı** (*time complexity*), girdinin boyutuna bağlı olarak işlettiği program adımı sayısıdır.

Tanımın can alıcı noktası "girdinin boyutundan bağımsız" ifadesidir. `a = b + c * d;` gibi bir atama tek adım sayılır; birkaç makine komutuna dönüşse de süresi *n*'ye bağlı değildir. Bir döngü ise tek adım sayılamaz, çünkü kaç kez döneceği *n*'ye bağlıdır. Döngünün her test edilişi bir adım, gövdesindeki her deyimin her işletilişi ayrı adımlardır. Bildirimler (`int i;` gibi) çalışma anında iş yapmadığı için adım sayılmaz. Fonksiyon başlığı ve kapanan süslü parantez de adım değildir.

Adım sayısı makineden bağımsızdır, ama gerçek süreyle ilişkisini korur: adım sayısı iki katına çıkarsa süre de yaklaşık iki katına çıkar. Bir adımın süresi makineden makineye değişse de bu oran değişmez.

Performansın neden bu kadar önemli olduğunu uç bir örnek gösterir. Bir savaş uçağının bombalama sistemi programlanıyor olsun. Bu program hedefi 3 saniyede bulup bombayı bırakıyor, karşı tarafınki aynı işi 2 saniyede yapıyorsa, savaş baştan kaybedilmiş demektir. Gerçek zamanlı sistemlerde, büyük veri işleyen sunucularda ve oyunlarda bir saniyelik fark, işe yarayan bir programla işe yaramayan bir program arasındaki farktır.

1.6.5 Adım sayma: sum()

Bir fonksiyonun adım sayısını bulmanın düzenli bir yolu **adım sayma tablosudur**. Tabloda her deyim için üç sayı yazılır. **s/e** (*steps per execution*), deyimin bir kez işletilmesinin kaç adım olduğudur. **Sıklık**, deyimin kaç kez işletildiğidir. **Toplam** ise bu ikisinin çarpımıdır. Fonksiyonun adım sayısı, toplam sütununun toplamıdır. `sum` fonksiyonunun tablosu Tablo 1.6'da verilmiştir.

Deyim	s/e	Sıklık	Toplam
`float sum(float list[], int n) {`	0	0	0
`float tempsum = 0;`	1	1	1
`int i;`	0	0	0
`for (i = 0; i < n; i++)`	1	*n + 1*	*n + 1*
`tempsum += list[i];`	1	*n*	*n*
`return tempsum;`	1	1	1
`}`	0	0	0
Toplam			*2n + 3*

Tablo 1.6: `sum` fonksiyonunun adım sayma tablosu.

Tablonun en dikkat isteyen satırı `for` satırıdır. Döngü gövdesi *n* kez işletilir, ama döngü koşulu *n + 1* kez test edilir. *i = 0, 1, …, n − 1* değerlerinde test doğru çıkar ve gövdeye girilir; bunlar *n* test eder. *i = n* olduğunda test bir kez daha yapılır, yanlış çıkar ve döngü biter. Bu son test de bir adımdır. Atama satırı ve `return` birer kez işletilir. Toplam *1 + (n + 1) + n + 1 = 2n + 3* adımdır.

1.6.6 Sayaçla doğrulama

Tablonun doğruluğu, programa bir sayaç eklenerek denetlenebilir. Global bir `count` değişkeni sıfırdan başlar ve her program adımında bir artırılır:

```
float sum(float list[], int n) {
  float tempsum = 0; count++;
  int i;
  for (i = 0; i < n; i++) {
    count++;             /* for testi */
    tempsum += list[i]; count++;
  }
  count++;  /* for'un son testi */
  count++;  return tempsum;
}
```

Sayaç artırımları tablodaki adımların yerine konmuştur: atamadan sonra bir, her döngü testinde bir, her gövde işletiminde bir, döngünün son testinde bir ve `return`'de bir. Döngü testinin sayacı gövdenin başına konmuştur, çünkü gövdeye her girişten önce bir test yapılmıştır. Döngüden çıkış testini saymak için döngüden sonra ayrıca bir artırım vardır.

`list = {3, 1, 4}` ve `n = 3` için izleme Tablo 1.7'de verilmiştir.

Olay	`i`	`tempsum`	`count`
başlangıç	—	—	0
`tempsum = 0`	—	0	1
test 0 < 3: doğru	0	0	2
`tempsum += list[0]`	0	3	3
test 1 < 3: doğru	1	3	4
`tempsum += list[1]`	1	4	5
test 2 < 3: doğru	2	4	6
`tempsum += list[2]`	2	8	7
test 3 < 3: yanlış	3	8	8
`return tempsum`	3	8	9

Tablo 1.7: Sayaçlı `sum(list, 3)` çağrısının izlemesi; `list = {3, 1, 4}`.

Sonunda `count = 9 = 2 · 3 + 3` olur; tablodaki formül doğrulanmıştır. Fonksiyon *3 + 1 + 4 = 8* değerini döndürür. Sayaçlı fonksiyon *n = 0, 1, ..., 6* için de çalıştırıldı ve her seferinde `count` değeri *2n + 3*'e eşit çıktı.

Sayaçlı kodda dikkat çeken bir ayrıntı vardır: `float tempsum = 0; count++;` satırından sonra `int i;` bildirimi gelir. Bir bloğun ortasında bildirim yapmak C89'da yasaktı, C99'dan beri serbesttir. Bu yüzden kod C99 ya da daha yeni bir standartla derlenmelidir.

1.6.7 Adım sayma: rsum()

Özyinelemeli bir fonksiyonun adım sayısı biraz farklı bulunur: her deyimin sıklığı, fonksiyonun kaç kez çağrıldığına göre hesaplanır. `rsum(list, n)` toplam *n + 1* kez çağrılır. Adım sayma tablosu Tablo 1.8'de verilmiştir.

Deyim	s/e	Sıklık	Toplam
`float rsum(float list[], int n) {`	0	0	0
`if (n)`	1	*n + 1*	*n + 1*
`return rsum(list, n-1) + list[n-1];`	1	*n*	*n*
`return 0;`	1	1	1
`}`	0	0	0

Deyim	s/e	Sıklık	Toplam
Toplam			*2n + 2*

Tablo 1.8: `rsum` fonksiyonunun adım sayma tablosu.

`if (n)` testi her çağrıda bir kez yapılır, yani *n + 1* kez. *n* sıfırdan farklıyken yapılan *n* çağrıda ikinci satır işletilir: özyinelemeli çağrının sonucu ile bir öğe toplanır ve döndürülür. Bu satır bir adım sayılır; çağrılan fonksiyonun kendi adımları, o çağrının kendi satırlarında ayrıca sayılır. Taban durumda bir kez `return 0` işletilir. Toplam *(n + 1) + n + 1 = 2n + 2*'dir.

Bu sonuç ilk bakışta `rsum`'u `sum`'dan (*2n + 3*) daha hızlı gösterir. Ama adım sayısı burada yanıltıcıdır. Bir adımın süresi girdiden bağımsızdır, ama bütün adımlar aynı sürede işlemez. Bir özyinelemeli çağrı yeni bir çerçeve açmayı, parametreleri kopyalamayı, dönüş adresini saklamayı ve dönüşte çerçeveyi kaldırmayı gerektirir. Bu iş, bir döngü turundaki bir karşılaştırma ve artırmadan çok daha pahalıdır. Gerçek makinede `rsum` genellikle `sum`'dan yavaştır ve fazladan bellek harcar.

Adım sayısının asıl söylediği şey başkadır: *n* büyüdükçe adım sayısının nasıl büyüdüğü. *2n + 3* ile *2n + 2*, *n* ile doğru orantılı büyür; ikisi de **doğrusaldır** ve ikisi de *O(n)*'dir. *n* iki katına çıktığında iki fonksiyonun da süresi yaklaşık iki katına çıkar.

1.6.8 rsum() ve çağrı yığını

`rsum`'a da sayaç eklenebilir:

```
float rsum(float list[], int n) {
  count++;              /* if testi */
  if (n) {
    count++;            /* toplama + return */
    return rsum(list, n - 1) + list[n - 1];
  }
  count++;
  return 0;
}
```

`rsum(list, 3)` çağrısının izlemesi, çağrı yığınının her adımdaki durumuyla birlikte Tablo 1.9'da verilmiştir. Yığın sütununda en sağdaki çağrı, o anda çalışan çağrıdır.

Olay	Çağrı yığını (alttan üste)	`count`
`rsum(list, 3)`: `if` testi, 3 ≠ 0	r(3)	1
toplama ve `return` adımı; önce `rsum(list, 2)` gerekli	r(3)	2
`rsum(list, 2)`: `if` testi, 2 ≠ 0	r(3), r(2)	3
önce `rsum(list, 1)` gerekli	r(3), r(2)	4
`rsum(list, 1)`: `if` testi, 1 ≠ 0	r(3), r(2), r(1)	5
önce `rsum(list, 0)` gerekli	r(3), r(2), r(1)	6
`rsum(list, 0)`: `if` testi, *n = 0*: taban durum	r(3), r(2), r(1), r(0)	7
`return 0`; yığın en derin hâlinde	r(3), r(2), r(1), r(0)	8
0 + `list[0]` = 3; `rsum(list, 1)` 3 döndürür	r(3), r(2), r(1)	8
3 + `list[1]` = 4; `rsum(list, 2)` 4 döndürür	r(3), r(2)	8
4 + `list[2]` = 8; `rsum(list, 3)` 8 döndürür	r(3)	8

Tablo 1.9: Sayaçlı `rsum(list, 3)` çağrısının izlemesi; `list = {3, 1, 4}`.

İzlemede iki evre açıkça görülür. **İniş evresinde** her çağrı `if` testini yapar, `count`'u bir kez daha artırır ve hemen yeni bir çağrı açar. Toplama henüz yapılamaz, çünkü toplamın ilk terimi olan özyinelemeli çağrının sonucu henüz bilinmez. Yığın dört çerçeveye kadar büyür. **Dönüş evresinde** taban durumdan başlayarak her çağrı, aldığı değere kendi öğesini ekleyip bir alttaki çağrıya döner. Toplamalar bu evrede, `list[0]`'dan başlayarak yapılır. Sayaç dönüş evresinde artmaz, çünkü toplama ve `return` adımları iniş evresinde sayılmıştır. Son değer *count = 8 = 2 · 3 + 2*'dir.



Şekil 1.7: `rsum(list, 3)` izlemesinin en derin anı. Yığında dört çerçeve vardır; en üstteki `rsum(list, 0)` taban duruma ulaşmış ve 0 döndürmek üzeredir.

1.6.9 Matris toplama

İç içe döngülerin adım sayısı da aynı yolla bulunur. İki matrisi toplayan `add` fonksiyonunun tablosu Tablo 1.10'da görülür. Fonksiyon `c[i][j] = a[i][j] + b[i][j]` işlemini her hücre için yapar.

Deyim	Sıklık
`void add(int a[][MAX_SIZE], int b[][MAX_SIZE],` `  int c[][MAX_SIZE], int rows, int cols) {`	0
`int i, j;`	0
`for (i = 0; i < rows; i++)`	*rows + 1*
`  for (j = 0; j < cols; j++)`	*rows · (cols + 1)*
`  c[i][j] = a[i][j] + b[i][j];`	*rows · cols*
Toplam	*2·rows·cols + 2·rows + 1*

Tablo 1.10: Matris toplama fonksiyonu `add`'in adım sayma tablosu (her deyim için s/e = 1).

Dış döngünün testi, `sum`'daki gibi, *rows + 1* kez yapılır. İç döngü dış döngünün her turunda baştan başlar; her başlayışında testi *cols + 1* kez yapılır. Dış döngü *rows* tur döndüğü için iç döngü testinin toplam sıklığı *rows · (cols + 1)*'dir. Atama her hücre için bir kez, toplam *rows · cols* kez işletilir. Hepsi toplanınca:

> *(rows + 1) + rows · (cols + 1) + rows · cols = 2·rows·cols + 2·rows + 1*

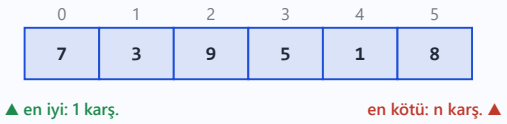
Örneğin *rows = 2*, *cols = 4* için *16 + 4 + 1 = 21* adım bulunur. Formül, fonksiyona sayaç eklenerek dört farklı satır ve iki farklı sütun sayısı için denendi; hepsinde sayaç formülle aynı değeri verdi.

Sonucun büyüme hızı *O(rows · cols)*'dir; kare bir matriste (*rows = cols = n*) bu $O(n^2)$ olur. Formüldeki *2·rows* terimi dış döngünün hangi boyutta döndüğüne bağlıdır. Satır sayısı sütun sayısından çok büyükse (*rows ≫ cols*) döngülerin yerleri değiştirilebilir: dış döngü sütunlarda, iç döngü satırlarda döner. Bu durumda sayım *2·rows·cols + 2·cols + 1* olur ve küçük terim azalır. Baskın terim ise değişmez. Matris çarpımı ise üç iç içe döngü gerektirir ve $O(n^3)$'tür.

> **Sık yapılan hata**
>
> Matris toplamanın karmaşıklığını "O(rows, cols)" diye yazmak. O gösteriminin içine bir fonksiyon yazılır; iki boyutu virgülle ayırmak bir anlam taşımaz. İşlem sayısı iki boyutun çarpımıyla büyüdüğü için doğrusu *O(rows · cols)*'dir.

1.6.10 En iyi, ortalama ve en kötü durum

Şu ana kadar incelenen fonksiyonlarda adım sayısı yalnız girdinin boyutuna bağlıydı: `sum` her *n* öğeli dizide *2n + 3* adım atar. Ama birçok algoritmada aynı boyuttaki iki girdi farklı sayıda adım gerektirir. **Doğrusal arama** (*linear search*) bunun en basit örneğidir: bir değer, dizinin öğeleriyle baştan sona tek tek karşılaştırılır ve bulununca durulur.



Şekil 1.8: Altı öğeli bir dizide doğrusal arama. Aranan değer ilk hücredeyse (7) tek karşılaştırma yeter; son hücredeyse (8) ya da dizide yoksa *n* karşılaştırma gerekir.

Şekil 1.8'deki {7, 3, 9, 5, 1, 8} dizisinde 7 aranırsa ilk karşılaştırmada bulunur. 8 aranırsa altı karşılaştırma gerekir. 6 aranırsa da altı karşılaştırmadan sonra bulunamadığı anlaşılır. Bu üç duruma ad verilir:

Durum	Ne zaman?	Karşılaştırma
En iyi	Aranan ilk hücrede	$1 \to \Theta(1)$
Ortalama	Aranan dizide var ve her konumda olması eşit olasılıklı	$(n + 1) / 2 \to \Theta(n)$
En kötü	Aranan son hücrede ya da dizide hiç yok	$n \to \Theta(n)$

Tablo 1.11: *n* öğeli dizide doğrusal aramanın en iyi, ortalama ve en kötü durumu.

- **En iyi durum** (*best case*), aynı boyuttaki girdiler arasında en az adım gerektiren girdinin maliyetidir.
- **En kötü durum** (*worst case*), en çok adım gerektiren girdinin maliyetidir.
- **Ortalama durum** (*average case*), girdilerin belirli bir olasılık dağılımına göre beklenen maliyettir.

Ortalama değer şöyle bulunur. Aranan değer dizide var olsun ve *n* konumun her birinde bulunma olasılığı *1/n* olsun. Değer *k*. konumdaysa (1'den sayarak) *k* karşılaştırma yapılır. Beklenen karşılaştırma sayısı:

> $(1/n) \cdot (1 + 2 + \ldots + n) = (1/n) \cdot n(n + 1)/2 = (n + 1)/2$

Altı öğeli dizide bu 3,5 eder: ortalamada dizinin yarısı taranır. Yarısı da olsa tarama *n* ile doğru orantılıdır; bu yüzden ortalama durum da $\Theta(n)$'dir. $\Theta(n)$ yazımı, adım sayısının *n* ile aynı hızda büyüdüğünü, yani *n*'nin iki sabit katı arasında kaldığını anlatır.

Analizde genellikle en kötü durum verilir. Bunun iki nedeni vardır. Birincisi, en kötü durum her girdi için geçerli bir garantidir: algoritma hiçbir girdide bundan fazla adım atmaz. Gerçek zamanlı bir sistemde ya da bir kullanıcının beklediği bir işlemde bu garanti ortalamadan daha değerlidir. İkincisi, ortalama durumu hesaplamak girdinin olasılık dağılımını bilmeyi gerektirir. Bu dağılım çoğu zaman bilinmez; varsayılan dağılım da gerçeğe uymayabilir. En iyi durum ise nadiren işe yarar bilgi verir: neredeyse her algoritma bazı şanslı girdilerde hızlıdır.

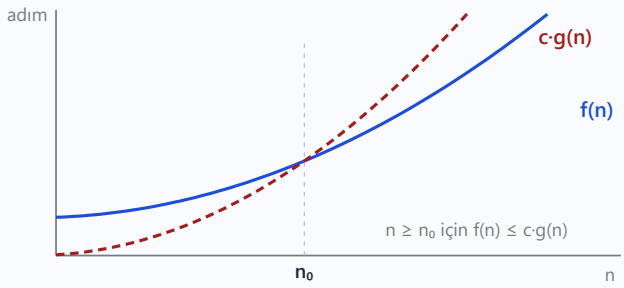
1.7 Asimptotik gösterim

Adım sayısını tam olarak bulmak her zaman kolay değildir ve çoğu zaman gerekli de değildir. `sum`'un $2n + 3$ adım attığını bilmek güzeldir, ama bu sayının $2n + 3$ mü, $3n + 2$ mi olduğu pek önemli değildir. Bir adımın gerçek süresi zaten bilinmez; adım sayısındaki sabit çarpanlar bu bilinmezliğin içinde kaybolur. Önemli olan, *n* büyüdükçe adım sayısının *nasıl* büyüdüğüdür: doğrusal mı, karesel mi, üstel mi? Bu soruyu sabitlerden ve küçük terimlerden arındırarak cevaplayan dile **asimptotik gösterim** (*asymptotic notation*) denir.

1.7.1 Büyük O

> **Tanım 1.5**
>
> Bütün $n \geq n_0$ değerleri için $f(n) \leq c \cdot g(n)$ olacak biçimde pozitif c ve n_0 sabitleri varsa $f(n) = O(g(n))$ yazılır ve "$f(n)$, $g(n)$ derecesindedir" diye okunur. Bu gösterime **büyük O** (*Big-O*) denir.

Tanım iki serbestlik tanır. Birincisi *c* sabitidir: $g(n)$'nin kendisi değil, istenen bir sabit katı $f(n)$'nin üstünde kalmalıdır. Bu serbestlik sabit çarpanları önemsiz kılar. İkincisi n_0 sabitidir: eşitsizlik yalnız yeterince büyük *n* değerleri için aranır. Küçük *n*'lerde ne olduğu önemsenmez. Şekil 1.9 bu durumu gösterir: n_0'dan önce $f(n)$ eğrisi $c \cdot g(n)$'nin üstünde olabilir, ama n_0'dan sonra hep altında kalır.



Şekil 1.9: Büyük O'nun tanımı. n_0'dan sonra $f(n)$ eğrisi, $c \cdot g(n)$ eğrisinin altında kalır.

`sum`, `rsum` ve `add` için bulunan adım sayıları bu gösterimle şöyle yazılır:

- $T_{sum}(n) = 2n + 3 = O(n)$. $n \geq 3$ için $2n + 3 \leq 3n$ olduğundan $c = 3$, $n_0 = 3$ seçilebilir.
- $T_{rsum}(n) = 2n + 2 = O(n)$. $n \geq 2$ için $2n + 2 \leq 3n$.
- $T_{add} = 2{\cdot}rows{\cdot}cols + 2{\cdot}rows + 1 = O(rows \cdot cols)$; kare matriste $O(n^2)$.

Gösterimdeki "=" işareti alışılmış eşitlik değildir. $O(g(n))$, $g(n)$'nin bir sabit katıyla üstten sınırlanan bütün fonksiyonların kümesidir; $f(n) = O(g(n))$ ifadesi de "$f(n)$ bu kümenin bir elemanıdır" anlamına gelir. Bu yüzden ifade tek yönlü okunur: $2n + 3 = O(n)$ yazılır, ama $O(n) = 2n + 3$ yazılmaz. Büyük O bir **üst sınırdır**; fonksiyonun en fazla ne hızla büyüdüğünü söyler.

Bu derste en çok zaman karmaşıklığı hesaplanır. Bellek de önemlidir, ama çoğu problemde zaman daha kıt kaynaktır: eksik bellek bazen eklenebilir, geçen zaman ise geri gelmez.

1.7.2 Örnek: $3n^3 + 6n^2 + 7n$

Örnek 1.3

$f(n) = 3n^3 + 6n^2 + 7n$ olsun. $f(n) = O(n^3)$ olduğunu tanıma uygun c ve n_0 bularak gösterelim. Yöntem, baskın olmayan terimleri adım adım bir üstteki kuvvetle sınırlamaktır.

1. En küçük terim $7n$'dir. $n \geq 7$ için $7n \leq n \cdot n = n^2$. Bu yüzden $f(n) \leq 3n^3 + 6n^2 + n^2$.
2. Kareli terimler birleşir: $6n^2 + n^2 = 7n^2$. Şimdi $f(n) \leq 3n^3 + 7n^2$.
3. Aynı yol bir kez daha: $n \geq 7$ için $7n^2 \leq n \cdot n^2 = n^3$. Böylece $f(n) \leq 3n^3 + n^3 = 4n^3$.
4. $c = 4$ ve $n_0 = 7$ bulunmuştur: bütün $n \geq 7$ için $f(n) \leq 4n^3$. Tanım sağlanır.

```
3n³ + 6n² + 7n ≤ 3n³ + 6n² + n²  (n ≥ 7 için 7n ≤ n²)
3n³ + 6n² + 7n ≤ 3n³ + 7n²
3n³ + 6n² + 7n ≤ 3n³ + n³  (n ≥ 7 için 7n² ≤ n³)
3n³ + 6n² + 7n ≤ 4n³ ⇒ f(n) = O(n³)  (c = 4, n₀ = 7)
```

Şekil 1.10: $3n^3 + 6n^2 + 7n \leq 4n^3$ eşitsizlik zinciri. Her satırda baskın olmayan bir terim, $n \geq 7$ koşuluyla bir üst kuvvete yükseltilir.

Bulunan c ve n_0 tek seçenek değildir. Örneğin $n \geq 1$ için her terim n^3'ün bir katıyla sınırlanabilir: $f(n) \leq 3n^3 + 6n^3 + 7n^3 = 16n^3$. Bu da $c = 16$, $n_0 = 1$ verir. Tanım yalnız *bazı* sabitlerin var olmasını ister; hangilerinin seçildiği önemli değildir. $c = 4$ için ise $n_0 = 7$ gerçekten en küçük değerdir: bir betikle denendiğinde $n = 6$'da $f(6) = 906 > 4 \cdot 216 = 864$ çıkar.

Sonucun iki pratik okuması vardır.

- **Üst sınır.** İşletim zamanı $f(n)$ adımla veriliyorsa, $n \geq 7$ için program hiçbir zaman $4n^3$ adımı aşmaz.
- **Kübik büyüme.** n üç katına çıktığında süre en fazla $3^3 = 27$ katına çıkar. Bu algoritma 100 sayıyı 2 saniyede sıralıyorsa, 300 sayıyı sıralaması en fazla $2 \cdot 27 = 54$ saniye sürer; 6 saniye değil.

Örnekteki yöntem genelleştirilebilir.

Teorem 1.1 (Polinomun derecesi)

$f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$ bir polinom ise $f(n) = O(n^m)$'dir.

İspat. $n \geq 1$ olsun. Her $i \leq m$ için $n^i \leq n^m$'dir. Katsayıların mutlak değerleriyle sınırlayalım: $f(n) \leq |a_m| n^m + \ldots + |a_1| n + |a_0| \leq (|a_m| + \ldots + |a_1| + |a_0|) \cdot n^m$. $c = |a_m| + \ldots + |a_0|$ ve $n_0 = 1$ seçilirse tanım sağlanır. □

Teorem, bir polinomun büyüme hızını en yüksek dereceli teriminin belirlediğini söyler. Bu yüzden $2n + 3$, $100n$ ve $n + 10^6$ aynı sınıftadır: hepsi $O(n)$'dir.

1.7.3 Ω ve Θ

Büyük O yalnız bir üst sınır verir ve bu sınır gevşek olabilir. $3n + 2 = O(n^2)$ ifadesi doğrudur: $n \geq 2$ için $3n + 2 \leq 4n \leq 4n^2$. Hatta $3n + 2 = O(2^n)$ da doğrudur. Ama bu ifadeler fonksiyon hakkında pek bilgi vermez. Bir fonksiyonun büyüme hızını tam olarak söylemek için alt sınır ve sıkı sınır kavramları da gerekir.

Tanım 1.6

Bütün $n \geq n_0$ için $f(n) \geq c \cdot g(n)$ olacak biçimde pozitif c ve n_0 varsa $f(n) = \Omega(g(n))$ yazılır (**büyük Omega**, *Big Omega*). Bütün $n \geq n_0$ için $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ olacak biçimde pozitif c_1, c_2 ve n_0 varsa $f(n) = \Theta(g(n))$ yazılır (**büyük Teta**, *Big Theta*).

Gösterim	Tanım	Anlamı
$f(n) = O(g(n))$	$f(n) \leq c \cdot g(n)$	g, f'nin üst sınırı
$f(n) = \Omega(g(n))$	$f(n) \geq c \cdot g(n)$	g, f'nin alt sınırı
$f(n) = \Theta(g(n))$	$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	hem O hem Ω: sıkı sınır

Tablo 1.12: Üç asimptotik gösterim (eşitsizlikler bütün $n \geq n_0$ için).

Θ, O ile Ω'nın kesişimidir: $f(n) = \Theta(g(n))$ ancak ve ancak $f(n) = O(g(n))$ ve $f(n) = \Omega(g(n))$ ise. Θ, fonksiyonun $g(n)$ ile *aynı hızda* büyüdüğünü söyler.

Örnek 1.4

$3n + 2 = \Theta(n)$ olduğunu gösterelim. Alt sınır için: her $n \geq 1$ için $3n + 2 \geq 3n$; yani $c_1 = 3$. Üst sınır için: $n \geq 2$ ise $2 \leq n$ ve $3n + 2 \leq 3n + n = 4n$; yani $c_2 = 4$. $n_0 = 2$ alınınca bütün $n \geq 2$ için $3n \leq 3n + 2 \leq 4n$ sağlanır. $n = 1$'de üst sınır tutmaz ($5 > 4$); n_0'ın 2 seçilmesi bu yüzdendir.

Asimptotik gösterim kullanılırken mümkün olan en küçük üst sınır yazılır. $3n + 2 = O(n^2)$ doğrudur ama yanıltıcıdır; okuyan, algoritmanın karesel olduğunu sanabilir. Sıkı bir sınır biliniyorsa Θ ile yazılır. Seçimli sıralama, girdi ne olursa olsun $n(n-1)/2$ karşılaştırma yapar. $n \geq 2$ için $n^2/4 \leq n(n-1)/2 \leq n^2/2$ olduğundan seçimli sıralama $\Theta(n^2)$'dir. Doğrusal aramada ise en kötü durum $\Theta(n)$, en iyi durum $\Theta(1)$'dir; her girdi için geçerli tek bir sınır istenirse $O(n)$ yazılır.

Bu gösterimler, bir problemin istenen sınırdan daha iyi çözülebildiği durumlarda da düşünmeyi kolaylaştırır. Bir sınav sorusu "$O(n + m)$ sürede çözün" diye sorulmuş olsun. Bir öğrenci problemi $O(n)$'de çözmüş, sonra sorunun sözüne uymak için kodun sonuna m kez dönen boş bir döngü eklemiş. Bu gereksizdir. $O(n)$ süreli bir çözüm zaten $O(n + m)$'dir, çünkü büyük O bir üst sınırdır. İstenen sınırdan daha iyisi yapılabiliyorsa ne âlâ.

1.7.4 Alışılmış karmaşıklıklar

Algoritmaların zaman karmaşıklıkları çoğunlukla birkaç tanıdık sınıftan birine düşer. Tablo 1.13 bu sınıfları, adlarını ve küçük n değerleri için aldıkları değerleri verir. Tabloda $\lg n$, $\log_2 n$ anlamındadır.

Karmaşıklık	Adı	n=1	n=2	n=4	n=8	n=16	n=32
1	sabit	1	1	1	1	1	1
lg n	logaritmik	0	1	2	3	4	5
n	doğrusal	1	2	4	8	16	32
n lg n	log-doğrusal	0	2	8	24	64	160
n^2	karesel	1	4	16	64	256	1.024
n^3	kübik	1	8	64	512	4.096	32.768
2^n	üstel	2	4	16	256	65.536	$4{,}29 \cdot 10^9$
n!	faktöriyel	1	2	24	40.320	$2{,}09 \cdot 10^{13}$	$2{,}63 \cdot 10^{35}$

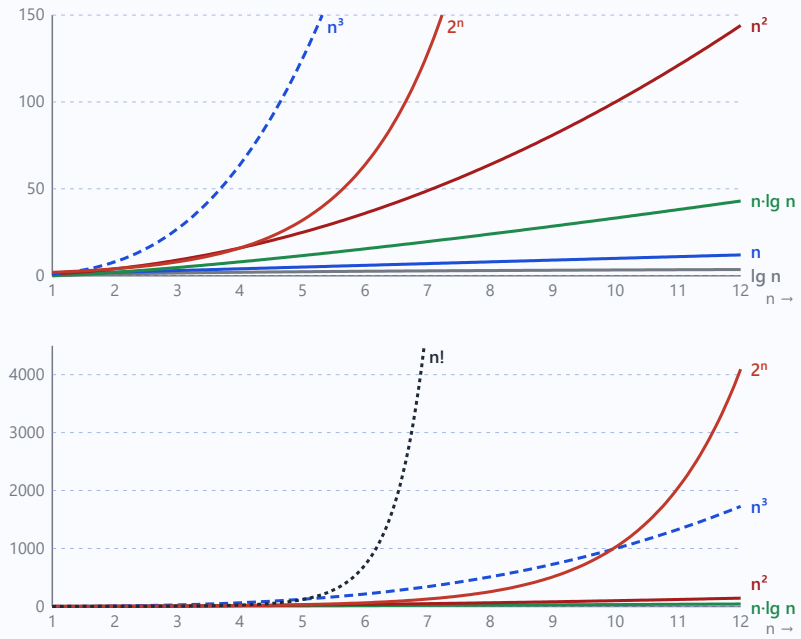
Tablo 1.13: Alışılmış karmaşıklık fonksiyonlarının küçük *n* değerlerindeki büyüklüğü.

Her sınıfın tipik bir kaynağı vardır. **Sabit** süreli bir işlem, dizinin bir öğesine indisle erişmek gibi, girdinin boyutuna bakmaz. **Logaritmik** süre, problemi her adımda yarıya indiren algoritmalarda görülür; sıralı bir dizide ikili arama bunun örneğidir. **Doğrusal** süre girdiye bir kez bakan algoritmaların, örneğin `sum`'un ve doğrusal aramanın süresidir. **Log-doğrusal** süre iyi sıralama algoritmalarında ortaya çıkar. **Karesel** süre iki iç içe döngüden, **kübik** süre üç iç içe döngüden gelir; seçimli sıralama ve matris çarpımı bunlara örnektir. **Üstel** ve **faktöriyel** süre ise bütün alt kümeleri ya da bütün sıralanışları deneyen algoritmaların süresidir.

Tablonun küçük *n* değerleri yanıltıcı olabilir. *n = 4*'te $2^n = n^2 = 16$'dır; iki fonksiyon aynı görünür. *n = 8*'de üstel fonksiyon öne geçmiştir: $2^n = 256$, $n^2 = 64$. *n = 32*'de fark milyarlarla ifade edilir. Belli bir noktadan sonra üstel ve faktöriyel terimler, polinom terimlerin hepsini ezer.

1.7.5 Büyüme grafiği

Aynı fonksiyonların *n = 1* ile *n = 12* arasındaki grafiği farkı daha da açık gösterir. Şekil 1.11'in üst grafiğinde dikey ölçek 0 ile 150 arasındadır. Bu ölçekte *lg n*, *n* ve *n lg n* birbirinden ayrışır; n^2 hızla tırmanır ve *n = 12*'de 144'e ulaşır. n^3 ve 2^n ise grafiğin dışına çoktan çıkmıştır.



Şekil 1.11: Karmaşıklık fonksiyonlarının *n = 1 … 12* arasındaki büyümesi. Üstte dikey ölçek 0–150, altta 0–4500'dür. Altta *n!*, *n = 7*'de grafiği terk eder; doğrusal fonksiyonlar tabana yapışmış görünür.

Alt grafikte ölçek 0 ile 4500 arasına genişletilmiştir. *n = 12*'de $n^3 = 1728$ ve $2^n = 4096$ olur. 2^n, *n = 10* civarında n^3'ü geçer ve ondan sonra hep önde kalır. *n!* ise *n = 7*'de 5040 değeriyle grafikten çıkar; *n = 6*'da ise yalnız 720'dir. Bu ölçekte *n* ve *n lg n* yatay eksenden ayırt edilemez.

1.7.6 Saniyede bir milyar komut

Karmaşıklığın gerçek süreye etkisini görmek için saniyede 10^9 komut işleyen, yani her adımı 1 nanosaniyede yapan bir makine düşünelim. Tablo 1.14, *f(n)* adım atan bir algoritmanın bu makinede ne kadar süreceğini gösterir. Hesapta 1 yıl 365 gün, 1 ay 30 gün alınmıştır. Tablodaki bütün değerler bir betikle yeniden hesaplanmış ve doğrulanmıştır.

n	n	lg n	n lg n	n^2	n^3	n^4	n^{10}	2^n
10	0,01 µs	0,00332 µs	0,0332 µs	0,1 µs	1 µs	10,0 µs	10,0 sn	1,02 µs
20	0,02 µs	0,00432 µs	0,0864 µs	0,4 µs	8 µs	160 µs	2,84 saat	1,05 ms
30	0,03 µs	0,00491 µs	0,147 µs	0,9 µs	27,0 µs	810 µs	6,83 gün	1,07 sn
40	0,04 µs	0,00532 µs	0,213 µs	1,6 µs	64,0 µs	2,56 ms	4,05 ay	18,3 dk
50	0,05 µs	0,00564 µs	0,282 µs	2,5 µs	125 µs	6,25 ms	3,1 yıl	13,0 gün
100	0,1 µs	0,00664 µs	0,664 µs	10,0 µs	1 ms	100 ms	31,7 asır	$4{,}02 \cdot 10^{11}$ asır
1000	1 µs	0,00997 µs	9,97 µs	1 ms	1 sn	16,7 dk	$3{,}17 \cdot 10^{11}$ asır	$3{,}40 \cdot 10^{282}$ asır
10000	10,0 µs	0,0133 µs	133 µs	100 ms	16,7 dk	3,86 ay	$3{,}17 \cdot 10^{21}$ asır	$6{,}33 \cdot 10^{2991}$ asır

Tablo 1.14: Saniyede 10^9 komut işleyen bir makinede *f(n)* adımın süresi.

Bir hücrenin nasıl hesaplandığını görmek için $n = 40$ ve $f(n) = 2^n$ alalım. $2^{40} \approx 1{,}1 \cdot 10^{12}$ adım, nanosaniye cinsinden $1{,}1 \cdot 10^{12}$ ns, yani yaklaşık 1100 saniyedir. Bu da 18,3 dakika eder. n 10 artıp 50 olduğunda adım sayısı $2^{10} = 1024$ katına çıkar ve süre 13 güne ulaşır. Üstel algoritmalarda girdiye birkaç öğe eklemek süreyi katlar.

Tablodan üç sonuç çıkar. Birincisi, logaritmik ve doğrusal algoritmalar on bin öğede bile mikrosaniyeler içinde biter. İkincisi, polinom algoritmalar arasında da büyük farklar vardır: on bin öğede n^2 100 milisaniye, n^3 16,7 dakika, n^4 ise 3,86 ay sürer. Üçüncüsü, üstel algoritmalar pratikte yalnız çok küçük girdilerde kullanılabilir. $n = 100$ için 2^n adım $4 \cdot 10^{11}$ asır sürer; evrenin yaşı ise yaklaşık $1{,}4 \cdot 10^8$ asırdır. $n = 10.000$ için süre yaklaşık 10^{2992} asırdır.

1.7.7 Amortize analiz

En kötü durum analizi her işlemi tek başına ele alır. Ama bazı veri yapılarında işlemler bazen pahalı, çoğunlukla ucuzdur. Böyle bir yapıda her işlemin en kötü maliyetini toplamak, toplam maliyeti olduğundan çok büyük gösterir. **Amortize analiz** (*amortized analysis*), n işlemlik bir dizinin toplam maliyetini hesaplar ve n'ye böler. Sonuç, işlem başına düşen ortalama maliyettir. Bu ortalama bir olasılık varsayımına dayanmaz; her işlem dizisi için geçerli bir garantidir.

Tipik örnek **dinamik dizidir** (*dynamic array*). Dinamik dizi belirli bir **kapasiteyle** başlar ve öğeleri sona ekler. Dizi dolduğunda `realloc` ile kapasitesi iki katına çıkarılır. `realloc`, ayrılmış bir bloğu büyütür; yerinde büyütemezse yeni bir blok ayırır ve eski öğeleri oraya kopyalar. Bir eklemenin maliyeti, bir yazma artı varsa kopyalanan öğe sayısıdır.

> **Örnek 1.5**
>
> Kapasitesi 1 olan boş bir diziye 9 öğe eklensin; her doluşta kapasite ikiye katlansın. 2. eklemede dizi doludur: 1 öğe kopyalanır, kapasite 2 olur. 3. eklemede 2 öğe kopyalanır, kapasite 4 olur. 5. eklemede 4 öğe kopyalanır, kapasite 8 olur. 9. eklemede 8 öğe kopyalanır, kapasite 16 olur. Toplam kopya $1 + 2 + 4 + 8 = 15$, toplam yazma 9'dur. 9 eklemenin toplam maliyeti 24'tür; ekleme başına $24 / 9 \approx 2{,}7$ işlem düşer. Tek tek bakıldığında 9. ekleme 9 işlem tutmuştur, ama ortalama 3'ün altında kalır.

Genel durumda n eklemede kopyalanan öğe sayısı $1 + 2 + 4 + \ldots + 2^{k'}$dır; burada $2^k < n$'dir. Bu geometrik toplam $2^{k+1} - 1 < 2n$'dir. Yazmalarla birlikte toplam maliyet $3n$'den küçüktür. Tek bir ekleme $O(n)$ sürebilir, ama ekleme başına **amortize maliyet** $O(1)$'dir. Bu sınır, 1 ile 1000 arasındaki farklı n değerleri için bir betikle de denetlenmiştir; toplam maliyet her seferinde $3n$'nin altında kaldı.

Kapasite iki katına çıkarılmak yerine her seferinde sabit bir miktar, örneğin 10 artırılsaydı sonuç çok farklı olurdu. Bu durumda her 10 eklemede bir bütün dizi kopyalanır ve kopya sayısı $10 + 20 + 30 + \ldots \approx n^2/20$ olur. Ekleme başına amortize maliyet $O(n)$'ye çıkar. Katlayarak büyüme, dinamik dizilerin ve birçok kütüphane yapısının dayandığı fikirdir.

1.8 Sihirli kare

Asimptotik gösterimin pratik önemini çarpıcı biçimde gösteren bir problem sihirli karedir. Aynı problem için biri kaba kuvvetle, biri akıllı bir kuralla çalışan iki yöntem karşılaştırılınca, doğru algoritmanın hiçbir donanım hızlanmasıyla telafi edilemeyecek bir fark yarattığı görülür.

1.8.1 Problem

Tanım 1.7

$n \times n$ boyutlu bir **sihirli kare** (*magic square*), 1'den n^2'ye kadar sayıların, her satırın, her sütunun ve iki ana köşegenin toplamı eşit olacak biçimde yerleştirildiği bir tablodur.

Bu ortak toplam n'ye bağlı olarak hesaplanabilir. 1'den n^2'ye kadar sayıların toplamı $n^2(n^2+1)/2$'dir. Bu sayılar n satıra dağılmıştır ve her satırın toplamı eşittir. Bu yüzden bir satırın toplamı:

$$n^2(n^2+1)/2 \div n = n(n^2+1)/2$$

$n = 3$ için toplam $3 \cdot 10 / 2 = 15$, $n = 5$ için $5 \cdot 26 / 2 = 65$, $n = 7$ için $7 \cdot 50 / 2 = 175$'tir. Şekil 1.12 5 × 5 boyutlu bir sihirli kareyi gösterir.

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

Şekil 1.12: 5 × 5 sihirli kare. Her satır, sütun ve ana köşegenin toplamı 65'tir. Kare, Coxeter kuralıyla üretilmiştir: 1 ilk satırın ortasındadır.

Şekildeki kareyi denetleyelim. İlk satır $15 + 8 + 1 + 24 + 17 = 65$, ilk sütun $15 + 16 + 22 + 3 + 9 = 65$, ana köşegen $15 + 14 + 13 + 12 + 11 = 65$, diğer köşegen $17 + 5 + 13 + 21 + 9 = 65$'tir. Kalan satır ve sütunlar da 65 tutar.

1.8.2 Kaba kuvvet ve daha hızlı makine

Problemi çözmenin akla gelen ilk yolu **kaba kuvvettir** (*brute force*): bütün olası çözümleri tek tek denemek. 1'den n^2'ye kadar sayılar n^2 hücreye $(n^2)!$ farklı biçimde dizilebilir. Her diziliş için satır, sütun ve köşegen toplamları denetlenir. 5 × 5 kare için $25! \approx 1{,}55 \cdot 10^{25}$ diziliş vardır. Saniyede 10^9 diziliş deneyen bir makine bunların hepsini $1{,}55 \cdot 10^{16}$ saniyede dener. Bir asır yaklaşık $3{,}15 \cdot 10^9$ saniye olduğundan bu süre yaklaşık 4,9 milyon asırdır.

Bu hesaba sık yapılan bir itiraz vardır: on kat daha hızlı, örneğin 10 GHz'lik yeni bir bilgisayar bu problemi hemen çözer; algoritmayla uğraşmaya gerek yoktur. Hesap bu itirazı çürütür:

$$25! / (10^{10} \cdot 3{,}15 \cdot 10^9) \approx 4{,}9 \cdot 10^5 \text{ asır}$$

On kat hızlı makine 4,9 milyon asrı yaklaşık 490 bin asra indirir. Sonuç hâlâ umutsuzdur. Bin kat hızlı bir makine bile süreyi ancak 4900 asra indirir. Karmaşıklığı $(n^2)!$ olan bir algoritmada n'yi 5'ten 6'ya çıkarmak ise adım sayısını $36!/25! \approx 2{,}4 \cdot 10^{16}$ katına çıkarır; hiçbir donanım ilerlemesi bunu karşılayamaz.

Doğru algoritma ise aynı kareyi yalnız 25 adımda, bir mikrosaniyenin çok altında kurar. Algoritma donanımdan önce gelir: yavaş bir algoritmayı hızlı bir makinede çalıştırmak, hızlı bir algoritmayı yavaş bir makinede çalıştırmaktan neredeyse her zaman daha kötüdür.

1.8.3 Coxeter kuralı ve C kodu

n tek sayı olduğunda sihirli kareyi doğrudan kuran basit bir kural vardır. Horowitz ve Sahni bu kuralı Coxeter'e dayandırır; Siyam yöntemi ya da de la Loubère yöntemi adlarıyla da bilinir. Kural şudur:

1. İlk satırın ortasına 1 yazılır.
2. Sonraki sayı için hep sol üst çapraza gidilir: bir satır yukarı, bir sütun sola.
3. Tablonun dışına taşılırsa karşı kenardan dönülür. Yukarı taşan bir hareket en alt satıra, sola taşan bir hareket en sağ sütuna gider. Tablo, karşılıklı kenarları birbirine yapışık bir simit (torus) gibi düşünülür.
4. Sol üst çapraz doluysa oraya gidilmez; bulunulan karenin hemen altına inilir.

Kural her sayıyı tek adımda yerleştirir ve n^2 adımda biter: zaman karmaşıklığı $O(n^2)$'dir. Tablonun kendisi zaten n^2 hücre içerdiği için daha hızlı bir algoritma olamaz; her hücreye en az bir kez yazmak gerekir. Bu yüzden kural $\Theta(n^2)$ süreli ve en iyi olabilecek bir algoritmadır. Kuralın C ile yazılışı şudur:

```c
void sihirli(int n, int k[][MAX]) {
  int i = 0, j = (n - 1) / 2, s, r, c;
  k[i][j] = 1;        /* k sifirlanmis olmali */
  for (s = 2; s <= n * n; s++) {
    r = (i - 1 < 0) ? n - 1 : i - 1; /* yukari */
    c = (j - 1 < 0) ? n - 1 : j - 1; /* sol */
    if (k[r][c]) i = (i + 1) % n; /* in */
    else { i = r; j = c; }
    k[i][j] = s;
  }
}
```

Kod satır satır şöyle çalışır:

- `k[][MAX]`, satır uzunluğu `MAX` olan iki boyutlu bir tam sayı dizisidir. C'de iki boyutlu bir dizi parametresinin ikinci boyutu yazılmalıdır, çünkü derleyici `k[i][j]` adresini `i · MAX + j` olarak hesaplar. `MAX`, programda `#define` ile tanımlanan bir sabittir ve *n*'den küçük olamaz.
- `i = 0`, `j = (n - 1) / 2` ilk satırın ortasıdır. *n = 5* için *j = 2*. Oraya 1 yazılır. Kod, bütün hücrelerin sıfır olduğunu varsayar; çağıran tarafın diziyi sıfırla başlatması gerekir. Bir hücrenin dolu olup olmadığı, sıfırdan farklı olup olmadığına bakılarak anlaşılır.
- Döngü *s = 2*'den n^2'ye kadar her sayıyı yerleştirir.
- `r` ve `c` sol üst çaprazın satır ve sütunudur. `i - 1 < 0` ise satır yukarı taşmıştır; `r` en alt satıra, `n - 1`'e gider. Sütun için de aynı denetim yapılır. Koşul işleci `? :` bu dönüşü tek satırda yazar.
- `k[r][c]` sıfırdan farklıysa çapraz doludur. Bu durumda bulunulan karenin altına inilir: `i = (i + 1) % n`. Mod işlemi alt kenardan taşmayı en üst satıra döndürür. `j` değişmez.
- Çapraz boşsa oraya geçilir: `i = r; j = c;`. Sonra `k[i][j] = s` ile sayı yazılır.

Kod derlenip *n = 3*, 5 ve 7 için çalıştırıldı ve her satır, sütun ve iki köşegenin toplamı bir program parçasıyla denetlendi. Üç karede de toplamlar sırasıyla 15, 65 ve 175 çıktı. 5 × 5 karenin çıktısı Şekil 1.12 ile aynıdır.

1.8.4 Kod izleme: n = 3

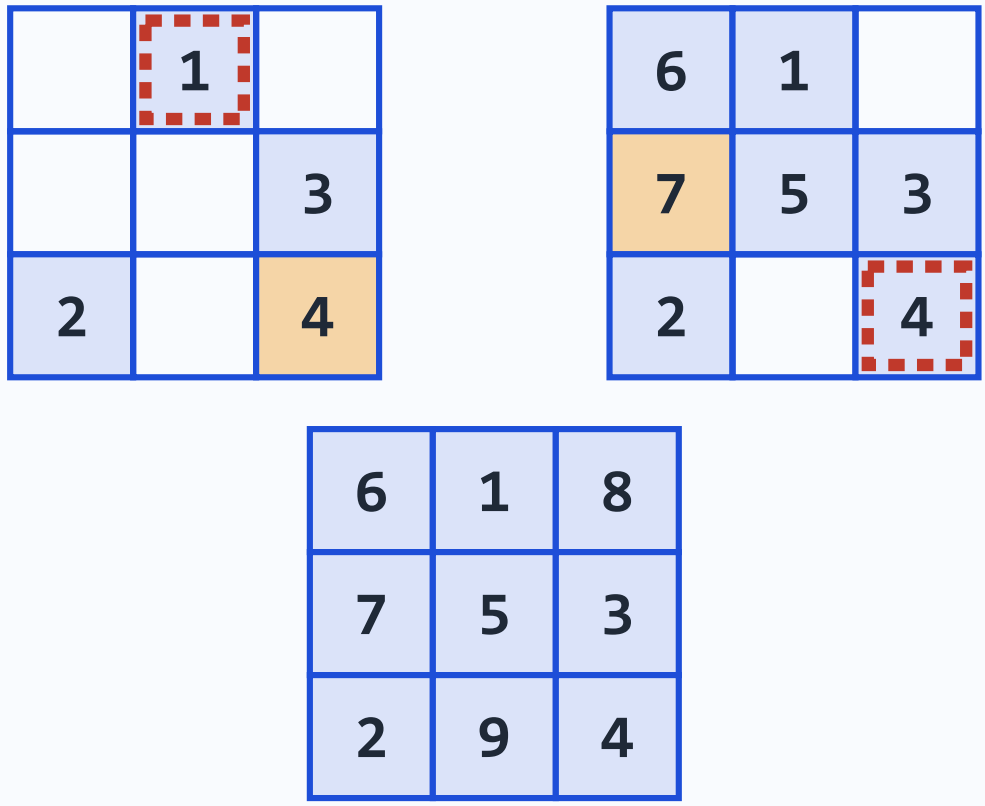
`sihirli(3, k)` çağrısını izleyelim. Kare 3 × 3 olduğu için satır ve sütun indisleri 0, 1 ve 2 değerlerini alır. Tablo 1.15 her sayının nereye yerleştiğini ve nedenini verir.

`s`	Önceki (i, j)	(r, c)	Çapraz	Yeni (i, j)
1	—	—	ilk satırın ortası	(0, 1)
2	(0, 1)	(2, 0)	boş; satır taştı → en alt satır	(2, 0)
3	(2, 0)	(1, 2)	boş; sütun taştı → en sağ sütun	(1, 2)
4	(1, 2)	(0, 1)	dolu (1 var) → bir aşağı	(2, 2)
5	(2, 2)	(1, 1)	boş	(1, 1)
6	(1, 1)	(0, 0)	boş	(0, 0)

`s`	Önceki (i, j)	(r, c)	Çapraz	Yeni (i, j)
7	(0, 0)	(2, 2)	dolu (4 var) → bir aşağı	(1, 0)
8	(1, 0)	(0, 2)	boş; sütun taştı → en sağ sütun	(0, 2)
9	(0, 2)	(2, 1)	boş; satır taştı → en alt satır	(2, 1)

Tablo 1.15: `sihirli(3, k)` çağrısının izlemesi. (r, c) sol üst çaprazdır.

İzlemenin dikkat isteyen iki satırı vardır. *s = 4*'te sol üst çapraz (0, 1)'dir ve orada 1 vardır; bu yüzden 3'ün altına, (2, 2)'ye inilir. *s = 7*'de bulunulan kare (0, 0)'dır. Sol üst çapraz hem satırdan hem sütundan taşar ve (2, 2) olur; orada 4 vardır. Bu yüzden (0, 0)'ın altına, (1, 0)'a inilir. *s = 10 > 9* olunca döngü biter.



Şekil 1.13: `sihirli(3, k)` izlemesinden üç an. (a) *s = 4*: çapraz dolu olduğu için 4, 3'ün altına yazıldı. (b) *s = 7*: 7, 6'nın altına yazıldı. (c) Bitmiş kare.

Sonuçta ortaya çıkan kare şudur: ilk satır 6, 1, 8; ikinci satır 7, 5, 3; üçüncü satır 2, 9, 4. Her satırın, her sütunun ve iki köşegenin toplamı *15 = 3 · (9 + 1)/2*'dir. Örneğin ana köşegen *6 + 5 + 4 = 15*, diğer köşegen *8 + 5 + 2 = 15*'tir.

1.9 Performans ölçümü

Performans analizi, bir algoritmanın büyüme hızını makineden bağımsız olarak söyler. Ama gerçek bir makinede gerçek bir girdiyle kaç milisaniye süreceğini söylemez. Bunu öğrenmenin yolu programı çalıştırıp süresini ölçmektir. Ölçüm, analizin göz ardı ettiği sabitleri, önbellek etkilerini ve derleyici iyileştirmelerini de içerir. Bu yüzden iki algoritmanın aynı karmaşıklık sınıfında olduğu durumlarda hangisinin daha hızlı olduğunu ancak ölçüm söyler.

1.9.1 clock() ve time()

Standart C kütüphanesi, `<time.h>` başlığında süre ölçmek için iki fonksiyon sunar. Tablo 1.16 ikisinin kullanılışını karşılaştırır.

	1. İşlemci zamanı: `clock()`	2. Takvim zamanı: `time()`
Başlangıç	`start = clock();`	`start = time(NULL);`
Bitiş	`stop = clock();`	`stop = time(NULL);`
Tip	`clock_t`	`time_t`
Sonuç (saniye)	`(double)(stop - start) / CLOCKS_PER_SEC`	`difftime(stop, start)`

Tablo 1.16: C'de süre ölçmenin iki yolu.

`clock()`, programın başlangıcından beri harcanan işlemci zamanını **saat vuruşu** (*clock tick*) cinsinden döndürür. Bir saniyedeki vuruş sayısı `CLOCKS_PER_SEC` sabitiyle verilir. İki ölçüm arasındaki fark bu sabite bölünerek saniyeye çevrilir. Bölmeden önce `(double)` dönüşümü yapılmalıdır; yoksa iki tam sayının bölümü küsuratı atar ve 0,4 saniye 0 olarak çıkar. `clock()` 'un çözünürlüğü sistemin vuruş aralığıyla sınırlıdır: iki vuruş arasında biten bir iş 0 süre almış görünür.

`time()` ise duvar saatini, yani takvim zamanını saniye cinsinden verir. `difftime` iki zaman arasındaki farkı `double` olarak döndürür. Çözünürlüğü bir saniyedir; dakikalarca süren işler için uygundur, kısa işler için kabadır. `time()` ayrıca bilgisayarda o sırada çalışan başka programların süresini de içerir.

Eski kitaplarda ve kodlarda `CLK_TCK` adına rastlanır. Bu ad standart dışıdır ve eski derleyicilerden kalmadır; bugünkü derleyicilerin çoğu onu tanımaz. Standart C'de `CLOCKS_PER_SEC` kullanılır.

Ölçüm yaparken bir kural daha vardır: çevre birimleriyle yapılan işlemler, yani ekrana yazma, diske yazma ya da yazıcıya gönderme, işlemcinin hızına göre çok yavaştır. Ölçülen bölgede bir `printf` bırakılırsa ölçülen süre büyük ölçüde `printf` 'in süresi olur; algoritmanın süresi onun gölgesinde kalır. `printf` 'ler ölçülen bölgenin dışında tutulmalıdır.

1.9.2 Seçimli sıralamayı ölçmek

Seçimli sıralamanın gerçek süresini farklı boyutlu dizilerde ölçen program şudur:

```c
#include <stdio.h>
#include <time.h>
#define MAX_SIZE 5001
#define ITERATIONS 20
void sort(int list[], int n);   /* secmeli siralama */

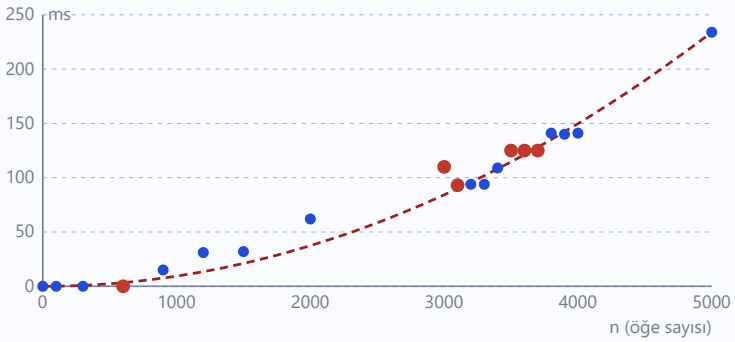
int main(void) {
  int i, j, list[MAX_SIZE];
  int sizelist[] = { 0, 100, 300, 600, 900, 1200, 1500, 2000,
    3000, 3100, 3200, 3300, 3400, 3500, 3600, 3700, 3800,
    3900, 4000, 5000 };
  clock_t start, stop;  double ms;
  for (i = 0; i < ITERATIONS; i++) {
    for (j = 0; j < sizelist[i]; j++)
      list[j] = sizelist[i] - j;   /* azalan dizi */
    start = clock();
    sort(list, sizelist[i]);
    stop = clock();
    ms = 1000.0 * (stop - start) / CLOCKS_PER_SEC;
    printf("%4d eleman: %6.1f ms\n", sizelist[i], ms);
  }
  return 0;
}
```

- `sizelist` dizisi ölçüm yapılacak 20 boyutu tutar: 0'dan 5000'e. Boyutlar 3000 ile 4000 arasında sık seçilmiştir; böylece komşu boyutların süreleri karşılaştırılabilir.
- Her boyut için dizi `list[j] = sizelist[i] - j` ile büyükten küçüğe doldurulur. Seçimli sıralamanın karşılaştırma sayısı girdiden bağımsız olduğu için azalan dizi özel bir durum değildir; ama her ölçümün aynı tür girdiyle yapılmasını sağlar.
- İki `clock()` çağrısının arasında yalnız `sort` çağrısı vardır. Diziyi doldurmak ve sonucu yazmak ölçülen bölgenin dışındadır.
- `ms = 1000.0 * (stop - start) / CLOCKS_PER_SEC` süreyi milisaniyeye çevirir. `1000.0` bir `double` sabittir; bu yüzden bütün ifade `double` aritmetiğiyle hesaplanır. Eski kodlarda bu satır `(int)(1000*(stop-start)/CLK_TCK)` biçiminde görülür; tam sayı bölmesi küsuratı atar ve `CLK_TCK` standart değildir. Bu yüzden `1000.0` ile `double` aritmetiği daha doğrudur.

- `sort`'un tanımı programda verilmez; seçimli sıralama fonksiyonu ve `SWAP` makrosu aynı dosyaya eklenmelidir. `MAX_SIZE` 5001'dir; en büyük ölçüm boyu olan 5000 öğe sığar.

1.9.3 Sonuçlar ve tutarsızlıklar

Programın eski bir makinedeki örnek çıktısı Şekil 1.14'te çizilmiştir. Noktalar ölçülen süreleri, kesikli eğri ise bir $c \cdot n^2$ eğrisini gösterir. Genel eğilim analizle uyumludur: süre n^2 ile büyür. Ama ayrıntıda tuhaflıklar vardır.



Şekil 1.14: Seçimli sıralamanın tek seferlik ölçüm sonuçları. Noktalar örnek ölçümleri, kesikli eğri $c \cdot n^2$ eğilimini gösterir. Kırmızı noktalar tutarsız görünen ölçümlerdir.

- 600 öğeye kadar bütün ölçümler 0 ms çıkmıştır. Sıralama gerçekten zaman almıştır, ama ölçülemeyecek kadar kısa sürmüştür.
- 3000 öğe 110 ms sürmüş, 3100 öğe ise 93 ms. Daha büyük girdi daha kısa sürmüş gibi görünür.
- 3500, 3600 ve 3700 öğenin üçü de 125 ms sürmüştür; aralarında hiç fark yoktur.

Bu tuhaflıkların nedeni `clock()`'un çözünürlüğüdür. Sayaç ancak bir saat vuruşunda artar. Sıralama iki vuruş arasında başlayıp biterse `stop - start = 0` olur. Vuruş aralığı yaklaşık 15,6 ms olan bir sistemde bütün sonuçlar 15–16 ms'nin katlarına yuvarlanır: 93, 110 ve 125 ms, sırasıyla altı, yedi ve sekiz vuruşa karşılık gelir. Gerçek süre bir vuruş kadar sapabilir. Bu yüzden 3000 ile 3100 öğe arasındaki "ters" fark gerçek değildir. Ayrıca işletim sistemi programı ölçüm sırasında kısa süreliğine durdurup başka bir işi çalıştırabilir; bu da tek ölçümleri bozar.

Program bu notun hazırlandığı bugünkü bir makinede de çalıştırıldı. Sonuçlar Tablo 1.17'de seçilmiş boyutlar için verilmiştir. Kod iyileştirmesiz (`-O0`) derlenmiştir; bu makinede `clock()`'un çözünürlüğü 1 ms'dir. Süreler makineye, derleyiciye ve derleme seçeneklerine göre değişir.

n	300	600	1200	2000	3000	3500	3600	4000	5000
ms	0	1	4	11	23	40	37	38	64

Tablo 1.17: Seçimli sıralamanın bugünkü bir makinede tek seferlik ölçümü (ms; makineye göre değişir).

Makine eskisinden kat kat hızlıdır, ama tuhaflık sürer: 3500 öğe 40 ms, 3600 öğe 37 ms sürmüştür. 1 ms'lik çözünürlük kısa ölçümlerde hâlâ kabadır; tek seferlik ölçümler de işletim sisteminin araya girmesine açıktır. Bu sorunun çözümü her ölçümü çok kez tekrarlamak ve ortalamasını almaktır.

> **Sık yapılan hata**
>
> Kısa bir işi tek seferlik `clock()` ölçümüyle ölçmek. Sonuç ya 0 çıkar ya da bir vuruş aralığının katlarına yuvarlanır. Bir arama gibi mikrosaniyeler süren bir iş, ölçülen bölgede binlerce kez tekrarlanmalı ve toplam süre tekrar sayısına bölünmelidir.

1.9.4 Bekçili doğrusal arama

Tekrarlı ölçümü doğrusal arama üzerinde görelim. Arama fonksiyonu küçük bir hile kullanır:

```c
int seqsearch(int list[], int searchnum, int n) {
  int i;
  list[n] = searchnum;          /* bekci */
  for (i = 0; list[i] != searchnum; i++)
    ;
  return (i < n) ? i : -1;
}
```

Sıradan bir doğrusal aramanın döngüsünde her turda iki test yapılır: `i < n` (dizinin sonuna gelindi mi?) ve `list[i] != searchnum` (aranan bulundu mu?). `seqsearch` ilk testi kaldırır. Aranan değeri dizinin hemen sonuna, `list[n]`'ye yazar. Bu hücreye **bekçi** (*sentinel*) denir. Aranan değer dizide olmasa bile döngü en geç bekçide durur; bu yüzden sınır denetimine gerek kalmaz. Döngü durduğunda `i < n` ise değer gerçekten dizide bulunmuştur ve indisi döndürülür. `i == n` ise döngü bekçide durmuştur ve `-1` döndürülerek değerin bulunamadığı bildirilir.

Bekçi hilesinin bir bedeli vardır: dizide n öğe için $n + 1$ yer ayrılmalıdır. Ölçüm programında `MAX_SIZE = 5001` seçilmesinin nedeni budur. Bekçi ayrıca diziyi değiştirir; `list[n]`'deki eski değer kaybolur. Bu yüzden ölçüm programı her boydan sonra bekçiyi siler.

En kötü durumda, yani aranan değer dizide yoksa, `seqsearch` $n + 1$ karşılaştırma yapar: n öğe ve bekçi. Zaman $O(n)$'dir; en iyi durum, aranan ilk hücredeyken, $\Theta(1)$'dir. Bekçi karşılaştırma sayısını değiştirmez, ama her turdaki test sayısını ikiden bire indirir; döngü yaklaşık iki kat hızlanabilir.

1.9.5 Kod izleme: seqsearch()

`seqsearch(list, 5, 3)` çağrısını, `list = {4, 8, 2}` dizisinde izleyelim. Dizide üç öğe vardır, ama `list[3]` için de yer ayrılmıştır. 5 aranmaktadır.

1. `list[3] = 5`: bekçi yerine konur. Döngü artık en geç burada duracaktır (Şekil 1.15a).
2. `i = 0`: `list[0] = 4 ≠ 5`, `i` 1 olur. Tek test yapılır; sınır denetimi yoktur.
3. `i = 1`: `list[1] = 8 ≠ 5`, `i` 2 olur.
4. `i = 2`: `list[2] = 2 ≠ 5`, `i` 3 olur.
5. `i = 3`: `list[3] = 5`; döngü durur. Ama bu bulunan değer bekçidir (Şekil 1.15b).
6. `i = 3 < n` yanlıştır; fonksiyon `-1` döndürür: 5 dizide yoktur. Toplam $n + 1 = 4$ karşılaştırma yapılmıştır.



Şekil 1.15: `seqsearch({4, 8, 2}, 5, 3)`. (a) Bekçi `list[3]`'e yazıldı. (b) Döngü `i = 3`'te, bekçide durdu; sonuç `-1`.

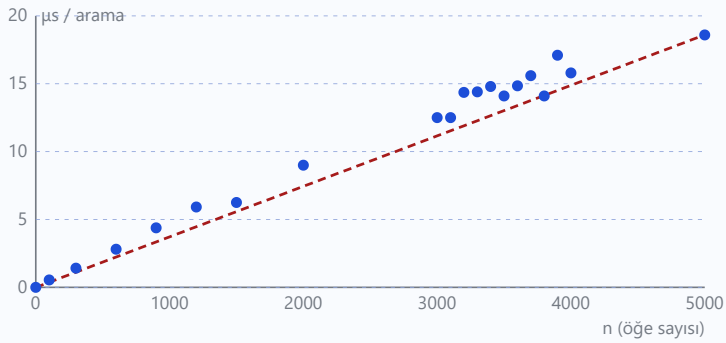
Aynı dizide 8 aranırsa döngü `i = 1`'de durur; $1 < 3$ olduğu için fonksiyon 1 döndürür. Bu iki çağrı derlenip çalıştırılmış ve sonuçlar `-1` ile 1 olarak doğrulanmıştır.

1.9.6 Tekrarlı ölçüm

Arama tek başına mikrosaniyeler sürer; tek seferlik bir ölçümle süresi bulunamaz. Ölçüm programı bu yüzden her boy için aramayı çok kez tekrarlar. Programın ölçüm bölümü şudur:

```
start = clock();
for (j = 0; j < runtimes[i]; j++)
  pos = seqsearch(list, -1, sizelist[i]);
stop = clock();
total = 1000.0 * (stop - start) / CLOCKS_PER_SEC;
ort = 1000.0 * total / runtimes[i];  /* mikro sn */
list[sizelist[i]] = sizelist[i];  /* bekciyi sil */
```

- Her n için arama `runtimes[i]` kez tekrarlanır. Tekrar sayısı küçük boylarda büyük (500.000), büyük boylarda küçüktür (5000). Böylece her boyun toplam ölçüm süresi birkaç vuruştan çok daha uzun olur, ama ölçüm de dakikalar sürmez.
- `-1` aranır. Dizide yalnız 0 ile $n - 1$ arası sayılar olduğu için `-1` hiç bulunmaz; her arama en kötü durumdadır ve tam $n + 1$ karşılaştırma yapar.
- `total` milisaniye cinsinden toplam süredir. `ort = 1000.0 * total / runtimes[i]`, tek bir aramanın ortalama süresini mikrosaniye cinsinden verir: milisaniyeyi 1000 ile çarpmak mikrosaniyeye çevirir.
- Son satır bekçiyi siler: `list[n]`'ye eski değeri, yani n'yi geri yazar. Dizinin `list[j] = j` ile doldurulduğu varsayılır.



Şekil 1.16: Tekrarlı ölçümle bulunan ortalama arama süreleri (µs / arama). Noktalar ölçülen ortalamaları, kesikli doğru doğrusal eğilimi gösterir.

Şekil 1.16 sonuçları gösterir. Ortalamalar artık 0 değildir ve n ile doğrusal büyür; bu, $O(n)$ beklentisiyle uyumludur. 3000'den sonraki dalgalanmanın nedeni tekrar sayısının azalmasıdır: `runtimes` küçüldükçe toplam süre de kısalır ve ölçüm yine kabalaşır. Ölçümün yapıldığı makinede saniyede 1000 vuruş vardı; yani bir vuruş 1 ms'dir.

Aynı ölçüm bu notu hazırlarken bugünkü bir makinede de yapıldı. 100 öğede arama başına yaklaşık 0,4 µs, 1000 öğede yaklaşık 3,4 µs, 5000 öğede yaklaşık 18 µs ölçüldü. Değerler makineye göre değişir, ama oranları aynı kalır: n on katına çıkınca süre de yaklaşık on katına çıkar. Kısa işlerin ölçümünde izlenecek yol özetle şudur: iş çok kez tekrarlanır, toplam süre ölçülür ve tekrar sayısına bölünür.

1.10 Özyinelemeli yordamlar

`rsum` fonksiyonu, bir problemin kendisinin daha küçük bir kopyasıyla nasıl çözülebileceğini gösterdi. Aynı yaklaşım matematikte özyinelemeli olarak tanımlanan birçok fonksiyonu doğrudan koda çevirmeyi sağlar. Her özyinelemeli yordamın iki parçası vardır: kendini çağırmadan sonuç veren bir **taban durum** ve problemi daha küçük bir probleme indirgeyen bir **özyineleme adımı**.

1.10.1 Faktöriyel, Fibonacci ve binom katsayısı

Üç klasik fonksiyonu ele alalım:

- **Faktöriyel.** $n! = n \cdot (n-1)!$ ve $0! = 1! = 1$. Taban durum $n \le 1$, adım $n \cdot fakt(n-1)$'dir.
- **Fibonacci dizisi.** $f_0 = 0$, $f_1 = 1$ ve $i > 1$ için $f_i = f_{i-1} + f_{i-2}$. Taban durum $n \le 1$'dir ve sonuç n'nin kendisidir; adım $fibo(n-1) + fibo(n-2)$'dir.
- **Binom katsayısı.** $C(n, m) = n! / (m! \cdot (n-m)!)$, n öğeden m tanesini seçmenin yollarının sayısıdır. Pascal üçgeninin bağıntısı $C(n, m) = C(n-1, m) + C(n-1, m-1)$'dir. Taban durum $m = 0$ ya da $m = n$'dir; ikisinde de sonuç 1'dir.

Tanımlar C'ye neredeyse kelimesi kelimesine çevrilir:

```c
long fakt(long n) {
  if (n <= 1) return 1;
  return n * fakt(n - 1);
}

long fibo(long n) {
  if (n <= 1) return n;   /* f0 = 0, f1 = 1 */
  return fibo(n - 1) + fibo(n - 2);
}

long komb(int n, int m) {
  if (m == 0 || m == n) return 1;
  return komb(n - 1, m) + komb(n - 1, m - 1);
}
```

Kodlar derlenip çalıştırıldığında `fakt(5) = 120`, `fakt(10) = 3628800`, `fibo(10) = 55`, `komb(5, 2) = 10` ve `komb(10, 3) = 120` verdi. `fakt(3)` çağrısının çalışmasını izleyelim. `fakt(3)`, $3 \cdot$ `fakt(2)`'yi hesaplamak ister; `fakt(2)`, $2 \cdot$ `fakt(1)`'i hesaplamak ister; `fakt(1)` taban durumdur ve 1 döndürür. Dönüşte `fakt(2)` $2 \cdot 1 = 2$, `fakt(3)` de $3 \cdot 2 = 6$ döndürür. Yığın, `rsum`'da olduğu gibi önce büyür, sonra küçülür.

> **Sık yapılan hata**
>
> Fibonacci adımında `fibo(n - 1) * fibo(n - 2)` yazmak. Taban değerler 0 ve 1 olduğu için çarpım hep 0 ya da 1 verir: $f_2 = 1 \cdot 0 = 0, f_3 = 0 \cdot 1 = 0$ ve bundan sonra hep 0. Tanım gereği adım bir toplamadır. Binom katsayısında sık görülen hata ise fonksiyonu `long komb()` diye parametresiz yazmaktır; n ve m parametre olmalıdır, çünkü her özyinelemeli çağrı farklı değerlerle yapılır.

1.10.2 Maliyet

`fakt(n)` kendini $n - 1$ kez çağırır ve her çağrıda sabit iş yapar. Süresi $O(n)$'dir; yığın derinliği de n'dir. Aynı iş bir döngüyle de $O(n)$ sürede ama $O(1)$ bellekle yapılabilir.

`fibo` ve `komb` ise bu hâlleriyle **üstel** zaman harcar. Neden açıktır: aynı alt problemler tekrar tekrar çözülür. `fibo(5)`, `fibo(4)` ile `fibo(3)`'ü çağırır; `fibo(4)` de `fibo(3)`'ü yeniden çağırır. `fibo(3)` iki kez, `fibo(2)` üç kez hesaplanır. Toplam çağrı sayısı $2f_{n+1} - 1$'dir. Sayaçla yapılan denemede `fibo(10)` 177, `fibo(20)` 21.891, `fibo(30)` ise 2.692.537 çağrı yaptı. Her 10 artışta çağrı sayısı yaklaşık 123 katına çıkar. Buna karşılık yığın derinliği yalnız n'dir, çünkü aynı anda yığında en fazla n çerçeve bulunur. Fibonacci sayıları bir döngüyle, son iki değeri tutarak $O(n)$ sürede hesaplanabilir. Tekrarlanan alt problemlerin sonuçlarını saklayarak üstel süreyi doğrusal ya da polinom süreye indirmek, algoritma tasarımının önemli fikirlerinden biridir.

1.11 Karmaşıklık özeti

Bu notta incelenen yordamların zaman ve ek bellek gereksinimleri Tablo 1.18'de toplanmıştır.

Yordam	Zaman	Ek bellek	Açıklama
`sort` (seçimli sıralama)	$\Theta(n^2)$	$O(1)$	$n(n-1)/2$ karşılaştırma, en fazla $n-1$ takas
`sum`	$\Theta(n)$	$O(1)$	$2n+3$ adım
`rsum`	$\Theta(n)$	$O(n)$	$2n+2$ adım, $n+1$ çerçeve
`add` (matris toplama)	$\Theta(rows \cdot cols)$	$O(1)$	kare matriste $\Theta(n^2)$
`seqsearch`	$O(n)$	$O(1)$	en iyi $\Theta(1)$, en kötü $n+1$ karşılaştırma
`sihirli` (Coxeter)	$\Theta(n^2)$	$O(1)$	kaba kuvvet: $(n^2)!$ diziliş
dinamik dizi ekleme	$O(1)$ amortize	$O(n)$	tek bir ekleme en kötü $O(n)$
`fibo` (özyinelemeli)	üstel	$O(n)$	yığın derinliği n

Tablo 1.18: Bu notta incelenen yordamların karmaşıklıkları.

Tablodaki "ek bellek" sütunu girdinin kendisini saymaz. `sort`, `sum` ve `add` yalnız birkaç yerel değişken kullanır. `rsum` ve `fibo` ise çağrı yığınında *n* ile orantılı yer kaplar. Dinamik dizinin *O(n)* belleği öğeleri tutan bloğun kendisidir: kapasite öğe sayısının en fazla iki katıdır ve yeniden ayırma sırasında eski ile yeni blok kısa bir süre birlikte bulunur.

1.12 Sık yapılan hatalar

1. **"Derlendi ve çalıştı, o hâlde doğru" sanmak.** Bir program bir örnekte doğru sonuç verebilir, ama boş girdide, tek öğede ya da tekrarlı değerlerde yanlış çalışabilir. Test verisi bütün senaryoları kapsamalıdır. Hatalı `sort` sürümü de doğru sonuç verdiği hâlde istenen algoritma değildir.
2. **Adım sayısını sabitlerle karıştırmak.** *2n + 3* ile *100n* aynı sınıftadır: ikisi de *O(n)*'dir. Asimptotik gösterim sabit çarpanları ve küçük terimleri bilerek atar.
3. **Gevşek bir üst sınırı en iyi sınır sanmak.** $3n + 2 = O(n^2)$ doğrudur ama bilgi vermez. Mümkün olan en küçük *g(n)* yazılır; sıkı sınır biliniyorsa Θ kullanılır.
4. **Kısa bir işi tek sefer `clock()` ile ölçmek.** Sonuç 0 ya da vuruş aralığının bir katı çıkar. İş çok kez tekrarlanmalı ve ortalaması alınmalıdır.
5. **Ölçülen bölgede `printf` bırakmak.** Giriş/çıkış süresi algoritmanın süresini gölgeler.
6. **Makroya yan etkili argüman vermek.** `SWAP(a[i++], a[j], t)` `i`'yi iki kez artırır ve bir değeri yok eder.
7. `malloc`**'un sonucunu denetlememek ve ayrılan belleği geri vermemek.** İlki boş işaretçi erişimine, ikincisi bellek sızıntısına yol açar.

1.13 Özet

- **Süreç.** Yazılım gereksinim, çözümleme, tasarım, kodlama ve doğrulama aşamalarından geçer. Tasarımda veri nesneleri ADT, işlemler algoritma olarak tanımlanır. Gerçeklemeyi ertelemek daha iyi bir seçim yapmayı sağlar.
- **Algoritma.** Girdi, çıktı, kesinlik, sonluluk ve etkinlik ölçütlerini sağlayan sonlu komut dizisidir. Sonlanmayan programlar hesaplama yöntemidir.
- **Soyutlama.** ADT, belirtimi gerçeklemeden ayırır. C'de bunun aracı başlık dosyası ve opak tiptir: kullanıcı yalnız `Kesir *` tutar, alanlar `.c` dosyasında gizlidir. Arayüzün sözleşmesi ve sahiplik kuralı açıkça yazılmalıdır.
- **Seçimli sıralama.** Her turda kalan kısmın en küçüğü başa taşınır. Her girdide $n(n-1)/2$ karşılaştırma yapar ve $\Theta(n^2)$'dir; takas sayısı en fazla $n-1$'dir. Takas, değişkenlerin adresleri gönderilerek bir fonksiyonla ya da bir makroyla yapılır.
- **Sayma.** Bellek $S(P) = c + S_P(I)$'dır. Zaman, program adımlarıyla sayılır: `sum` $2n+3$, `rsum` $2n+2$, `add` $2 \cdot rows \cdot cols + 2 \cdot rows + 1$ adım atar. Özyineleme her çağrıda bir çerçeve açar ve bellek harcar.
- **Sınıflama.** O üst, Ω alt, Θ sıkı sınırdır. Genellikle en kötü durum verilir, çünkü her girdi için bir garantidir. Amortize analiz bir işlem dizisinin toplam maliyetine bakar; katlanarak büyüyen dinamik dizide ekleme amortize *O(1)*'dir.
- **Büyüme.** Küçük *n* yanıltır. 2^n ve $n!$ kısa sürede umutsuzlaşır. Sihirli karede kaba kuvvet milyonlarca asır, Coxeter kuralı mikrosaniyeler sürer: algoritma donanımdan önce gelir.
- **Ölçüm.** `clock()` ile ölçülür; kısa işler tekrarlanır ve ortalaması alınır; giriş/çıkış ölçülen bölgenin dışında tutulur. Bekçi, doğrusal aramada her turdaki sınır testini kaldırır.

1.14 Kavram listesi

sistem yaşam döngüsü *system life cycle* · aşağıdan yukarı *bottom-up* · yukarıdan aşağı *top-down* · algoritma *algorithm* · soyut veri tipi *abstract data type (ADT)* · gerçekleme *implementation* · başlık dosyası *header file* · eksik tip *incomplete type* · opak tip *opaque type* · struct (yapı) *struct* · typedef (tip adı) *typedef* · ok işleci *arrow operator (->)* · malloc *malloc* · free *free* · boş işaretçi *null pointer (NULL)* · bellek sızıntısı *memory leak* · sarkan işaretçi *dangling pointer* · sıralama *sorting* · seçimli sıralama *selection sort* · takas *swap* · makro *macro* · önişlemci *preprocessor* · içerik işleci *dereference (*)* · performans analizi *performance analysis* · alan karmaşıklığı *space complexity* · zaman karmaşıklığı *time complexity* · program adımı *program step* · özyineleme *recursion* · taban durum *base case* · çağrı yığını *call stack* · yığın çerçevesi *stack frame* · yığın taşması *stack overflow* · en iyi / ortalama / en kötü durum *best / average / worst case* · asimptotik gösterim *asymptotic notation* · büyük O *Big O* · büyük Omega *Big Omega* · büyük Teta *Big Theta* · üstel *exponential*

amortize analiz *amortized analysis* · dinamik dizi *dynamic array* · kapasite *capacity* · realloc *realloc* · sihirli kare *magic square* · kaba kuvvet *brute force* · doğrusal arama *linear (sequential) search* · bekçi *sentinel*

1.15 Kendini Dene

1. `for (i = 0; i < n; i++) for (j = i; j < n; j++) x++;` parçasında `x++` kaç kez işletilir? $n = 4$ için değer nedir ve parçanın zamanı hangi sınıftadır?

   Cevap

   `i` . turda iç döngü $n - i$ kez döner. Toplam $n + (n-1) + \dots + 1 = n(n+1)/2$'dir. $n = 4$ için $4 \cdot 5 / 2 = 10$. Parça $\Theta(n^2)$'dir.

2. $5n^2 + 20n + 50 = O(n^2)$ eşitliğinde $c = 6$ seçilirse en küçük n_0 kaçtır?

   Cevap

   $5n^2 + 20n + 50 \le 6n^2$ eşitsizliği $20n + 50 \le n^2$'ye denktir. $n = 22$'de $490 > 484$, $n = 23$'te $510 \le 529$'dur ve bundan sonra hep sağlanır. En küçük n_0 23'tür. $c = 75$ seçilirse $n_0 = 1$ yeter.

3. Bir dizinin en büyük öğesini bulan şu fonksiyon en kötü ve en iyi durumda kaç program adımı atar? `int maks(int a[], int n) { int m = a[0]; int i; for (i = 1; i < n; i++) if (a[i] > m) m = a[i]; return m; }`

   Cevap

   Atama 1 adım, `for` testi n kez, `if` testi $n - 1$ kez, `return` 1 adım. `m = a[i]` ataması en kötü durumda (dizi artan sıralıysa) $n - 1$ kez, en iyi durumda (en büyük öğe başta) hiç işletilmez. En kötü durum $3n$, en iyi durum $2n + 1$ adımdır; ikisi de $\Theta(n)$. Sayaçlı sürüm $n = 6$ için 18 ve 13 verir.

4. Seçimli sıralama {5, 4, 3, 2, 1} dizisinde kaç karşılaştırma ve kaç gerçek takas (`i ≠ min`) yapar?

   Cevap

   Karşılaştırma sayısı girdiden bağımsızdır: $5 \cdot 4 / 2 = 10$. Takaslar: `i = 0` 'da 5 ile 1, `i = 1` 'de 4 ile 2 yer değiştirir; dizi {1, 2, 3, 4, 5} olur. `i = 2` ve `i = 3` 'te `min = i` 'dir. Toplam 2 gerçek takas yapılır.

5. Bekçili `seqsearch` , beş öğeli bir dizide, dizide bulunmayan bir değeri ararken kaç karşılaştırma yapar? Aranan değer indis 2'deyse kaç karşılaştırma yapar?

   Cevap

   Değer yoksa beş öğe ve bekçi ile $n + 1 = 6$ karşılaştırma yapılır. Değer indis 2'deyse `list[0]` , `list[1]` ve `list[2]` ile 3 karşılaştırma yapılır; döngü `i = 2` 'de durur ve $2 < 5$ olduğu için 2 döndürülür.

6. Kapasitesi 4 olan boş bir dinamik diziye, her doluşta kapasite ikiye katlanarak 20 öğe ekleniyor. Toplam kaç öğe kopyalanır? Yazmalarla birlikte toplam maliyet nedir?

   Cevap

   5. eklemede 4, 9. eklemede 8, 17. eklemede 16 öğe kopyalanır: toplam 28. Yazmalarla toplam maliyet $28 + 20 = 48$'dir; ekleme başına 2,4 işlem düşer ve $3n = 60$ sınırının altında kalır.

7. Özyinelemeli `fibo(5)` çağrısı, ilk çağrı dâhil, toplam kaç kez `fibo` çağırır? En derin anda yığında kaç `fibo` çerçevesi bulunur?

   Cevap

   Çağrı sayısı $2f_6 - 1 = 2 \cdot 8 - 1 = 15$'tir. Elle de sayılabilir: $T(n) = T(n-1) + T(n-2) + 1$, $T(0) = T(1) = 1$ bağıntısından $T(2) = 3$, $T(3) = 5$, $T(4) = 9$, $T(5) = 15$. En derin zincir `fibo(5)` → `fibo(4)` → `fibo(3)` → `fibo(2)` → `fibo(1)` olduğu için yığında en fazla 5 çerçeve bulunur.

8. $\Theta(n^2)$ süreli bir algoritma 1000 öğede 1 saniye sürüyor. 4000 öğede yaklaşık ne kadar sürer? On kat hızlı bir makinede 1 saniyede kaç öğe işlenebilir?

   Cevap

   n dört katına çıkınca süre $4^2 = 16$ katına çıkar: yaklaşık 16 saniye. On kat hızlı makinede 1 saniyede on kat fazla adım atılır; n^2 on katına çıkarsa n, $\sqrt{10} \approx 3{,}16$ katına çıkar. Yaklaşık 3160 öğe işlenir. Karesel bir algoritmada makineyi on kat hızlandırmak girdiyi ancak üç kat büyütür.

9. 5 × 5 sihirli karede Coxeter kuralıyla 2 sayısı hangi hücreye yazılır? Neden?

   Cevap

   1, ilk satırın ortasına, (0, 2)'ye yazılır. Sol üst çapraz (−1, 1)'dir; satır yukarı taştığı için en alt satıra döner ve (4, 1) olur. Hücre boştur; 2 oraya yazılır. Şekil 1.12'de alt satırın ikinci hücresinde 2 görülür.

10. `main.c` içinde `Kesir k; k.pay = 3;` yazılırsa ne olur? Neden?

    Cevap

    Derleme hatası olur. `main.c` yalnız `kesir.h`'yi görür ve orada `struct kesir` eksik bir tiptir: derleyici boyunu ve alanlarını bilmez. Bu yüzden bu tipte bir değişken tanımlanamaz ve alanlarına erişilemez. Kullanıcı kesirlerle yalnız `Kesir *` işaretçisi ve arayüzdeki fonksiyonlar aracılığıyla çalışabilir; opak tipin amacı da budur.

1.16 İleri okuma

- E. Horowitz, S. Sahni ve S. Anderson-Freed, *Fundamentals of Data Structures in C* (2. baskı), Bölüm 1: bu notun temelidir; sistem yaşam döngüsünü, ADT'yi, adım sayma tablolarını, asimptotik gösterimi, sihirli kareyi ve performans ölçümünü daha fazla alıştırmayla verir.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (3. baskı), Bölüm 2–3 ve 17: algoritma analizini, O, Ω ve Θ gösterimlerini kesin tanımlar ve ispatlarla, amortize analizi de dinamik tablo örneğiyle ayrıntılı anlatır.
- R. Sedgewick, *Algorithms in C*: algoritma analizinin ilkelerini C koduyla ve çok sayıda ölçümle anlatır; kuramsal tahminlerin deneylerle nasıl karşılaştırılacağını gösterir.
- B. W. Kernighan ve D. M. Ritchie, *The C Programming Language* (2. baskı): işaretçiler, yapılar, önişlemci ve standart kütüphane (`<stdlib.h>`, `<time.h>`) için başvuru kaynağıdır.

1.17 Kaynaklar

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, Bölüm 1.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3. baskı: Bölüm 2–3 (analiz, asimptotik gösterim), Bölüm 17 (amortize analiz).
3. B. W. Kernighan, D. M. Ritchie, *The C Programming Language*, 2. baskı.
4. R. Sedgewick, *Algorithms in C*.

KISIM I · BÖLÜM 2

Diziler ve Yapılar

Dizi, programlamanın en eski ve en çok kullanılan veri yapısıdır. Öğelerine indisle, sabit sürede erişilir. Bu hız, dizinin bellekte ardışık durmasından ve her öğenin adresinin basit bir formülle hesaplanmasından gelir. Not, diziyi önce soyut bir veri tipi olarak tanımlar, sonra C'deki gerçeklemesine iner. Dizi ile işaretçi arasındaki ilişki, `malloc`, `realloc` ve `free` ile büyüyen diziler ve ikiye katlamanın amortize maliyeti bunu izler. Çok boyutlu dizilerin tek boyutlu belleğe nasıl dizildiği ve öğe adresinin nasıl hesaplandığı ayrıntılı biçimde ele alınır. Aynı düşünce, alt üçgen ve kuşak matrisleri yalnız gerekli öğeleri saklayarak tutmakta da kullanılır. Son kesimler farklı tipteki alanları bir araya getiren `struct` ve `union` tiplerini, C dizgilerini ve matris çarpımının analizini anlatır.

Kod (C): `github.com/sercankulcu/data-structures-and-algorithms-c/Bolum02`

ÖĞRENME ÇIKTILARI

Diziyi soyut ve somut görmek
Dizi ADT'sini tanımlamak ve C dizisinin bellekteki ardışık yerleşimini, dizi adı ile işaretçi arasındaki ilişkiyi açıklamak.

Adres hesaplamak
Satır ve sütun öncelikli yerleşimde iki, üç ve n boyutlu dizilerin öğe adresini bulmak; Horner yöntemiyle çarpma sayısını azaltmak.

Bellek tasarrufu
Üçgen ve kuşak matrisleri tek boyutlu bir dizide saklamak ve her öğeye $O(1)$ sürede erişen fonksiyonları yazmak.

Dinamik bellek
`malloc`, `realloc` ve `free` ile büyüyen dizi ve iki boyutlu dizi kurmak; ikiye katlamanın amortize maliyetini gerekçelendirmek.

Kayıt türleri
`struct`, `union` ve `'\0'` ile biten dizgileri doğru ve güvenli kullanmak; hizalamanın yapı boyutuna etkisini açıklamak.

Matris işlemleri
Matris toplama ve çarpımının adım sayısını çıkarmak ve çarpmanın $O(n^3)$ olduğunu göstermek.

2.1 Dizi soyut veri tipi

Bir veri yapısını iki düzeyde düşünmek mümkündür. Soyut düzeyde yalnız yapının ne iş gördüğü, yani hangi değerleri tuttuğu ve hangi işlemleri sunduğu konuşulur. Somut düzeyde ise bu işlemlerin bellekte hangi düzenle ve hangi kodla yapıldığı ele alınır. Birinci düzeye **soyut veri tipi** (*abstract data type*, ADT), ikinci düzeye **gerçekleme** (*implementation*) denir. Dizi, bu iki düzeyin çoğu zaman birbirine karıştırıldığı bir yapıdır: “dizi” denince akla hemen bellekte yan yana duran hücreler gelir. Oysa yan yana durmak dizinin tanımı değil, en yaygın gerçeklemesidir.

2.1.1 Dizi bir eşlemedir

Soyut bakışta dizi, indislerle değerler arasında kurulmuş bir ilişkidir. Her geçerli indise tek bir değer karşılık gelir. Bir sınıftaki öğrencilerin sıra numarasına göre notları, bir yılın günlerine göre sıcaklıklar ya da bir satranç tahtasının karelerine göre taşlar bu türden ilişkilerdir. İndis kümesi bir sayı aralığı olabileceği gibi, satranç tahtasındaki gibi sayı çiftlerinden de oluşabilir.

Tanım 2.1

Dizi (*array*), ⟨indis, değer⟩ çiftlerinden oluşan bir kümedir. Tanımlı her indis için kümede o indisle başlayan tam olarak bir çift bulunur. Bu ilişkiye **eşleme** (*mapping*) denir: dizi, indis kümesinden değer kümesine bir eşlemedir.

Bir ADT yalnız değerleriyle değil, işlemleriyle de tanımlanır. Horowitz, Sahni ve Anderson-Freed'in ders kitabı dizi ADT'si için üç işlem verir. `Create` boş bir dizi kurar, `Retrieve` bir indisteki değeri okur, `Store` ise bir indise değer yazar. Tablo 2.1'de bu işlemlerin anlamı görülür.

İşlem	Anlamı
`Create(j, liste)`	*j* boyutlu, indis kümesi `liste` olan boş bir dizi döndürür.
`Retrieve(A, i)`	*i* geçerli bir indisse *A[i]* değerini, değilse hata döndürür.
`Store(A, i, x)`	*i* geçerliyse *A*'da *i* ile *x*'i eşleştirir ve yeni diziyi döndürür; değilse hata döndürür.

Tablo 2.1: Dizi ADT'sinin işlemleri.

Tanımda iki ayrıntı önemlidir. Birincisi, `Retrieve` ve `Store` geçersiz bir indiste hata verir. ADT düzeyinde indis denetimi işlemin bir parçasıdır. İkincisi, `Store` "yeni diziyi" döndürür. Soyut düzeyde bir yazma işlemi eski diziyi bozmaz, yeni bir eşleme üretir. C gibi bir dilde ise yazma, var olan hücrenin içeriğini değiştirir. Bu iki fark, gerçeklemenin ADT'den neden ayrı düşünülmesi gerektiğini gösterir. C dizisi indisi hiç denetlemez ve yazdığı yeri yerinde değiştirir. Bu iki özellik hız kazandırır, karşılığında güvenlikten ödün verir.

2.1.2 Hızın kaynağı ve bedeli

Dizinin en yaygın gerçeklemesi, öğeleri bellekte ardışık hücrelere yerleştirmektir. Her öğe aynı tipte olduğundan aynı sayıda bayt kaplar. Bu sayıya *s*, ilk öğenin adresine α denirse *i* indisli öğenin adresi doğrudan hesaplanır:

$$\text{adres}(A[i]) = \alpha + i \cdot s$$

Bu hesap bir çarpma ve bir toplamadan ibarettir. Dizinin uzunluğu ne olursa olsun işlem sayısı değişmez. Bu yüzden indisle erişim *O(1)* sürer. Bin öğeli bir dizinin 999. öğesine erişmek, ilk öğesine erişmek kadar kısa sürer. Bağlı liste gibi yapılarda aynı öğeye ulaşmak için baştan başlayıp öğe öğe ilerlemek gerekir. Dizinin ayırt edici gücü budur.

Ardışık yerleşimin iki bedeli vardır. Birincisi, dizinin boyutu baştan bellidir. Bellekte ayrılan blok sonradan kendiliğinden uzamaz; daha büyük bir dizi gerekirse yeni bir blok ayrılıp eski öğeler oraya taşınır. İkincisi, araya ekleme ve silme pahalıdır. *n* öğeli bir dizinin başına yeni bir öğe eklemek için bütün öğelerin birer hücre sağa kaydırılması gerekir. Silmede de boşalan yeri kapatmak için sonraki öğeler sola kaydırılır. Kaydırılan öğe sayısı en kötü durumda *n* olduğundan araya ekleme ve silme *O(n)* sürer. Dizinin sonuna ekleme ise kaydırma gerektirmez; bu işlemin maliyeti dinamik dizilerde ayrıca incelenir.

2.2 C'de tek boyutlu dizi

C, dizi ADT'sini doğrudan dilin içinde gerçekler. Bir dizi bildirildiğinde derleyici ardışık bir bellek bloğu ayırır ve ilk öğenin adresini akılda tutar. Bu kesim bildirimi, adres hesabını, `sizeof` işlecini ve dizi adının işaretçiyle olan yakın ilişkisini ele alır. Konuyu izlemek için işaretçiyle ilgili iki işleci bilmek yeter: `&x`, `x` nesnesinin adresini verir; `*p` ise `p` işaretçisinin gösterdiği nesneye ulaşır.

2.2.1 Bildirim, adres ve sizeof

Aşağıdaki bildirim iki dizi tanımlar. `list` beş tam sayı, `plist` ise beş tam sayı işaretçisi tutar.

```
int list[5], *plist[5];
/* list : 5 tamsayı       */
/* plist: 5 tamsayı işaretçisi */

list[i]  /* adresi: α + i * sizeof(int) */
sizeof list   /* 5 * sizeof(int) bayt */
```

Derleyici `list` için beş `int` 'lik ardışık bir blok ayırır ve `list[0]` 'ın adresini, yani α'yı tutar. Program `list[i]` yazdığında öğenin adresi çalışma anında $\alpha + i \cdot$ `sizeof(int)` olarak hesaplanır. `sizeof` , bir tipin ya da nesnenin bayt cinsinden boyunu veren bir işleçtir. Değeri derleme sırasında bilinir. Yerel bir dizide `sizeof list` bütün dizinin boyunu verir: `int` 4 bayt ise $5 \cdot 4 = 20$ bayt. `plist` ise beş işaretçi tutar ve 64 bitlik bir sistemde her işaretçi 8 bayt olduğu için `sizeof plist` 40'tır. Bu değerler not için yazılan sürücü programda ölçülmüştür; `int` ve işaretçi boyları derleyiciye ve sisteme göre değişebilir.



Şekil 2.1: Dört tam sayılık bir dizinin bellekteki görünümü. `sizeof(int) = 4` varsayımıyla öğeler α, $\alpha + 4$, $\alpha + 8$ ve $\alpha + 12$ adreslerinde durur.

Şekil 2.1 bu yerleşimi gösterir. Her kutu bir öğedir; kutunun üstünde indisi, altında adresi yazar. Adresler dörder bayt artar, çünkü her `int` dört bayt kaplar. İndisler 0'dan başlar: n öğeli bir dizinin geçerli indisleri $0, 1, \ldots, n - 1$'dir. Sıfırdan başlamanın nedeni adres formülüdür. İlk öğenin adresi $\alpha + 0 \cdot s = \alpha$ olur ve formülde ayrıca bir düzeltme terimi gerekmez.

C, indisin geçerli olup olmadığını denetlemez. ADT'deki `Retrieve` ve `Store` geçersiz indiste hata verirken C, `list[5]` ifadesinde de $\alpha + 5 \cdot 4$ adresini hesaplar ve oraya erişir. Bu adreste dizinin bir parçası bulunmaz; başka bir değişken, dönüş adresi ya da hiç ayrılmamış bir bölge olabilir. C standardı bu durumu **tanımsız davranış** (*undefined behavior*) sayar. Program çökebilir, yanlış sonuç üretebilir ya da görünüşte doğru çalışabilir. En tehlikelisi sonuncusudur, çünkü hata ancak başka bir makinede ya da başka bir girdide ortaya çıkar.

> **Sık yapılan hata**
>
> n öğeli bir dizide döngüyü `for (i = 0; i <= n; i++)` biçiminde yazmak. Son turda `a[n]`'ye erişilir ve dizinin bir hücre ötesi okunur ya da ezilir. Derleyici bu hatayı yakalamaz. Doğru koşul `i < n`'dir; "n öğe, son indis $n - 1$" kuralı her döngüde akılda tutulmalıdır.

2.2.2 Dizi adı ve işaretçi

İşaretçi (*pointer*), bir nesnenin bellek adresini tutan değişkendir. C'de dizi adı çoğu ifadede ilk öğenin adresine dönüşür. Örneğin `list` yazıldığında derleyici bunu `&list[0]` olarak, yani `int *` tipinde bir adres olarak işler. Bu kural dizi ile işaretçiyi birbirine çok yaklaştırır. Tablo 2.2 birbirine eşdeğer ifadeleri listeler.

İfade	Eşdeğeri
`list`	`&list[0]`
`list + i`	`&list[i]` ($i \cdot$ `sizeof(int)` bayt ileri)
`*(list + i)`	`list[i]`
`p = list;` sonra `p[i]`	`list[i]` ile aynı hücre

Tablo 2.2: Dizi ve işaretçi ifadelerinin eşdeğerleri (`int list[5]; int *p;`).

Tablonun ikinci satırı **işaretçi aritmetiğinin** (*pointer arithmetic*) temel kuralını içerir. Bir işaretçiye tam sayı eklendiğinde birim bayt değil, gösterilen tipin bir öğesidir. `p` bir `int *` ise `p + 1` bir sonraki `int`'i gösterir; adres 1 değil, 4 bayt artar. `double *` için artış 8 bayt, `char *` için 1 bayttır. Derleyici çarpmayı kendisi yapar. Bu sayede `*(list + i)` yazımı, adres formülü $\alpha + i \cdot s$'nin C'deki doğrudan karşılığı olur. Gerçekte `list[i]` yazımı da standartta `*(list + i)` olarak tanımlanır; köşeli parantez bu toplama ve içerik alma işleminin kısa yazılışıdır.

Dizi adı ile işaretçi değişkeni yine de aynı şey değildir. İki fark önemlidir:

- **Dizi adına atama yapılamaz.** `list = p;` derlenmez. Dizi adı bir değişken değil, derleyicinin bildiği sabit bir adrestir. `p = list;` ise geçerlidir, çünkü `p` bir değişkendir ve başka bir adres tutabilir.
- `sizeof` **farklı sonuç verir.** Yerel bir dizide `sizeof list` bütün dizinin boyunu (20 bayt) verir. Bir işaretçide ise `sizeof p` yalnız işaretçinin kendi boyudur (64 bitlik sistemde 8 bayt).

İkinci fark fonksiyon parametrelerinde sık hataya yol açar. C'de bir fonksiyona dizi geçirildiğinde dizi kopyalanmaz; yalnız ilk öğesinin adresi geçer. Bu yüzden `void f(int a[])` ile `void f(int *a)` bildirimleri tamamen aynıdır. Parametre `int a[10]` diye yazılsa bile `a` fonksiyonun içinde bir işaretçidir ve `sizeof a` 8 verir. Not için yazılan sürücüde derleyici bu kullanımda `-Wsizeof-array-argument` uyarısını vermiştir. Dizinin uzunluğu fonksiyona ayrı bir parametreyle gönderilir. Standart kütüphanedeki `qsort`, `memcpy` ve `fwrite` gibi fonksiyonların hepsinin bir uzunluk parametresi almasının nedeni budur.

Sık yapılan hata

Fonksiyon içinde dizi uzunluğunu `sizeof a / sizeof a[0]` ile hesaplamak. Bu kalıp yalnız dizinin bildirildiği yerde doğru çalışır. Parametre olarak gelen `a` bir işaretçi olduğundan sonuç 64 bitlik bir sistemde *8 / 4 = 2* çıkar ve dizinin gerçek uzunluğuyla ilgisi yoktur.

2.2.3 Kod izleme: yazdir()

Aşağıdaki program dört öğeli bir diziyi `yazdir` fonksiyonuna geçirir. Fonksiyon her öğenin adresini ve değerini işaretçi aritmetiğiyle yazdırır.

```
void yazdir(int *p, int n) {
  int i;
  for (i = 0; i < n; i++)
    printf("%p %d\n", (void *)(p + i), *(p + i));
}

int main(void) {
  int dizi[] = {10, 20, 30, 40};
  yazdir(dizi, 4);
  return 0;
}
```

Program satır satır şöyle çalışır:

- `main`, `dizi`'yi yığında (*stack*) dört ardışık `int` olarak ayırır. Boyut başlangıç değerlerinin sayısından çıkarılır. İlk öğenin adresine α diyelim.
- `yazdir(dizi, 4)` çağrısında dizi kopyalanmaz. `dizi` adı `&dizi[0]`'a dönüşür ve bu adres, yani α, `p` parametresine kopyalanır. Uzunluk ayrı parametreyle (`n = 4`) gelir.
- Döngünün her turunda `p + i`, $\alpha + i \cdot 4$ adresini hesaplar. `*(p + i)` bu adresteki değeri okur; `p[i]` yazmakla aynıdır.
- `%p` biçimi bir `void *` bekler. Bu yüzden adres `(void *)` ile dönüştürülür. Dönüşüm yazılmazsa davranış standarda göre tanımsızdır.

`i`	`i < n`	`p + i`	`*(p + i)`	Yazılan satır
0	0 < 4, doğru	$\alpha + 0$	10	α 10
1	1 < 4, doğru	$\alpha + 4$	20	$\alpha + 4$ 20
2	2 < 4, doğru	$\alpha + 8$	30	$\alpha + 8$ 30
3	3 < 4, doğru	$\alpha + 12$	40	$\alpha + 12$ 40
4	4 < 4, yanlış	—	—	döngü biter, `yazdir` döner

Tablo 2.3: `yazdir(dizi, 4)` çağrısının izlemesi (`sizeof(int) = 4`).

Gerçek çıktıda α yerine onaltılık bir adres görünür. Not için yapılan bir çalıştırmada adresler `…F910`, `…F914`, `…F918` ve `…F91C` çıkmıştır. Son iki onaltılık basamak 10, 14, 18, 1C biçiminde dörder artar. Adresin kendisi her çalıştırmada değişebilir; değişmeyen, ardışık iki öğe arasındaki 4 baytlık farktır. Fonksiyon dizinin her öğesine bir kez uğrar ve her turda sabit iş yapar. Süresi $\Theta(n)$, ek belleği $O(1)$'dir.

2.3 Dinamik diziler

`int list[5];` gibi bir bildirimde boyut derleme sırasında bellidir. Pek çok programda ise boyut ancak çalışma anında öğrenilir: kullanıcının girdiği öğrenci sayısı, bir dosyadaki satır sayısı ya da bir ağ bağlantısından gelen paket sayısı. Bu durumda bellek çalışma anında istenir. Buna **dinamik bellek ayırma** (*dynamic memory allocation*) denir. C'de bu iş `<stdlib.h>` başlığındaki `malloc`, `calloc`, `realloc` ve `free` fonksiyonlarıyla yapılır.

2.3.1 malloc ile dizi

Aşağıdaki parça önce dizinin boyutunu okur, sonra o kadar `int`'lik bir blok ayırır, bloğu dizi gibi kullanır ve en sonunda geri verir.

```c
int n, i, *a;
scanf("%d", &n);
a = malloc(n * sizeof *a);   /* n int */
if (a == NULL) {
  fprintf(stderr, "Yetersiz bellek\n");
  exit(EXIT_FAILURE);
}
for (i = 0; i < n; i++)
  a[i] = i * i;              /* dizi gibi */
free(a);                     /* geri ver */
a = NULL;          /* sarkan işaretçi olmasın */
```

Kod satır satır şöyle çalışır:

- `malloc(n * sizeof *a)`, *n* tam sayıya yetecek bayt sayısını ister ve bloğun başlangıç adresini döndürür. Dönüş tipi `void *`'dır; C bu adresi `int *` tipindeki `a`'ya dönüşüm yazmadan atar.
- Bellek yetmezse `malloc` `NULL` döndürür. **Boş işaretçi** (*null pointer*) hiçbir nesneyi göstermez. Bu durum denetlenmeden `a[i]`'ye yazılırsa program çöker. Denetim başarısız olursa bir ileti yazılır ve program `exit(EXIT_FAILURE)` ile sonlanır.
- Döngü `a[i] = i * i` ile bloğu doldurur. `a` bir işaretçi olsa da `a[i]` yazımı dizideki gibi çalışır, çünkü `a[i]` zaten `*(a + i)` demektir.
- `free(a)` bloğu sisteme geri verir. Ardından `a = NULL` yazılır. `free`'den sonra `a` hâlâ eski adresi tutar, ama o adresteki bellek artık programa ait değildir. Böyle bir işaretçiye **sarkan işaretçi** (*dangling pointer*) denir. `NULL` atamak, olası bir yanlış kullanımı belirsiz bir bozulma yerine hemen fark edilen bir hataya çevirir.

Yerel bir dizi ile `malloc` bloğunun ömrü farklıdır. Fonksiyon içinde bildirilen dizi, çağrının **yığın** çerçevesinde durur ve fonksiyon dönünce yok olur. `malloc` ile alınan blok ise **heap bellekte** (*heap memory*) durur ve `free` edilene kadar yaşar. Bu yüzden bir fonksiyon, içinde ayırdığı bir bloğun adresini güvenle döndürebilir, ama yerel bir dizinin adresini döndüremez. Yerel dizinin adresi fonksiyon döndüğü anda sarkan işaretçi olur.

Boyut ifadesindeki `sizeof *a` kalıbı bilinçli bir seçimdir. `sizeof *a`, `a`'nın gösterdiği tipin boyunu verir. `a`'nın tipi sonradan `double *` yapılsa bile ifade kendiliğinden doğru boyu hesaplar. `sizeof(int)` yazmak da doğrudur, ama tip değişince güncellenmesi unutulabilir. Daha tehlikeli hata, `sizeof(int *)` ile `sizeof(int)`'i karıştırmaktır: 64 bitlik bir sistemde ilki 8, ikincisi 4 bayttır. Yanlış olan küçük kalırsa bloğun sonundan taşılır, büyük kalırsa bellek boşa gider.

Kütüphanenin iki fonksiyonu daha vardır. `calloc(n, s)`, *n* tane *s* baytlık öğe için yer ayırır ve bloğun bütün baytlarını sıfırlar. `malloc` ise bloğu sıfırlamaz; ilk içerik belirsizdir. `realloc(p, yeni)`, daha önce ayrılmış bir bloğu büyütür ya da küçültür. Blok yerinde büyüyemezse `realloc` yeni bir blok ayırır, eski içeriği oraya kopyalar ve eski bloğu serbest bırakır. `p` `NULL` ise `realloc` tam olarak `malloc` gibi davranır. Bu son özellik, boş başlayan büyüyen dizilerde işe yarar.

> **Sık yapılan hata**
>
> `malloc`'un dönüşünü denetlememek ya da `free`'den sonra işaretçiyi kullanmaya devam etmek. Birincisinde bellek yetmediğinde program `NULL` adrese yazmaya çalışır. İkincisinde ise geri verilen blok başka bir istek için yeniden ayrılmış olabilir ve bir yazma başka bir verinin üstüne düşer. Her `malloc`'tan sonra `NULL` denetimi, her `free`'den sonra işaretçiye `NULL` ataması alışkanlık hâline getirilmelidir.

2.3.2 Büyüyen dizi: ekle()

Kaç öğe geleceği hiç bilinmiyorsa dizi büyüyebilmelidir. **Dinamik dizi** (*dynamic array*), dolduğunda `realloc` ile büyütülen dizidir. Bir dinamik diziyi tanımlamak için üç bilgi gerekir: öğelerin durduğu bloğun adresi, dizide şu anda kaç öğe bulunduğu ve bloğun en çok kaç öğe alabildiği. Son değere **kapasite** (*capacity*) denir. Aşağıdaki kodda bu üç bilgi `DinDizi` yapısının `veri`, `boyut` ve `kap` alanlarında durur.

```c
typedef struct { int *veri; int boyut, kap; } DinDizi;

int ekle(DinDizi *d, int x) {
  if (d->boyut == d->kap) {              /* dolu */
    int yeni = d->kap ? 2 * d->kap : 1;
    int *t = realloc(d->veri, yeni * sizeof *t);
    if (t == NULL) return 0;    /* eski blok sağlam */
    d->veri = t;
    d->kap = yeni;
  }
  d->veri[d->boyut++] = x;
  return 1;
}
```

Kod satır satır şöyle çalışır:

- `d` bir `DinDizi *`'dır. Yapının kendisi kopyalanmasın ve fonksiyon `boyut` ile `kap`'ı kalıcı olarak değiştirebilsin diye işaretçi alınır. `d->boyut` yazımı `(*d).boyut`'un kısaltmasıdır.
- `d->boyut == d->kap` ise dizi doludur ve önce büyütülmelidir. Yeni kapasite eskisinin iki katıdır. Kapasite 0 ise iki katı da 0 olacağından `yeni` 1 alınır. `?:` koşul işleci bu iki durumu tek satırda ayırır.
- `realloc`'un sonucu önce geçici `t` değişkenine alınır. Başarısızlıkta `realloc` `NULL` döndürür ama eski bloğa dokunmaz. Sonuç doğrudan `d->veri`'ye yazılsaydı eski bloğun adresi kaybolurdu: öğeler hem erişilemez hem geri verilemez olurdu. Bu, bir **bellek sızıntısıdır** (*memory leak*). Kodda başarısızlık durumunda 0 döndürülür ve dizi eski hâliyle sağlam kalır.
- Büyütme başarılıysa `d->veri` ve `d->kap` güncellenir.
- `d->veri[d->boyut++] = x;` satırı öğeyi ilk boş hücreye yazar. `boyut++` son ek biçimidir: ifadede önce eski değer kullanılır, sonra değer bir artırılır. Bu yüzden `x`, `veri[boyut]`'a yazılır ve ardından `boyut` bir artar.

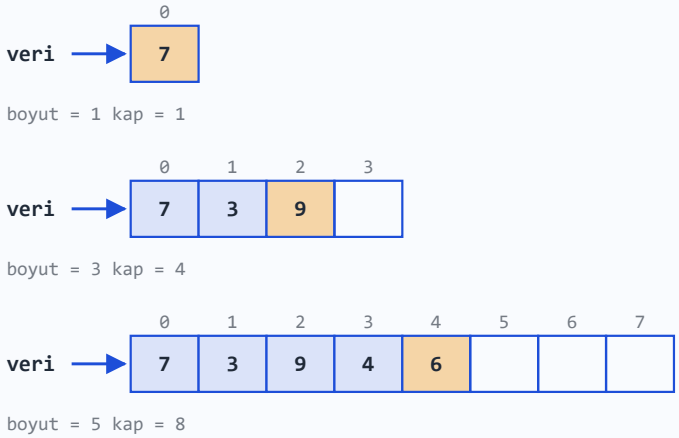
Örnek 2.1

Boş bir dinamik diziye (`{NULL, 0, 0}`) sırayla 7, 3, 9, 4 ve 6 eklenir. Her eklemede dizinin dolu olup olmadığına bakılır; doluysa kapasite ikiye katlanır ve eski öğeler yeni bloğa kopyalanır. Tablo 2.4 bütün adımları verir.

`x`	`boyut == kap`	`yeni`	**Kopya (toplam)**	`kap`	`boyut`	`veri`
7	0 == 0, dolu	1	0 (0)	1	1	[7]
3	1 == 1, dolu	2	1 (1)	2	2	[7, 3]
9	2 == 2, dolu	4	2 (3)	4	3	[7, 3, 9, _]
4	3 == 4, yer var	—	0 (3)	4	4	[7, 3, 9, 4]
6	4 == 4, dolu	8	4 (7)	8	5	[7, 3, 9, 4, 6, _, _, _]

Tablo 2.4: Boş diziye 7, 3, 9, 4, 6 eklenirken `ekle`'nin izlemesi. “Kopya”, büyütmede yeni bloğa taşınan öğe sayısıdır.

İlk eklemede `veri` `NULL`'dır ve `realloc(NULL, …)` bir `malloc` gibi davranır; kopyalanacak öğe yoktur. Sonraki büyütmelerde sırayla 1, 2 ve 4 öğe kopyalanır. Beş ekleme için toplam yalnız 7 kopya yapılmıştır. Dördüncü eklemede (4) dizide yer vardır ve öğe doğrudan yazılır. Not için yazılan sürücü programda `realloc` çağrılarının sayısı ve kopya toplamı bu tabloyla birebir aynı çıkmıştır. Burada “kopya”, `realloc`'un en kötü durumda taşıdığı öğe sayısıdır. Blok yerinde büyüyebilirse `realloc` hiç kopyalamaz; analiz bu yüzden bir üst sınır verir.



Şekil 2.2: Dinamik dizinin üç anı. (a) İlk ekleme: kapasite 1, öğe 7. (b) Üçüncü ekleme: kapasite 4'e katlandı, 9 yazıldı; bir hücre boş. (c) Beşinci ekleme: kapasite 8, beş öğe dolu, üç hücre boş.

2.3.3 Amortize maliyet

Tek bir `ekle` çağrısının maliyeti büyük ölçüde değişir. Dizide yer varsa çağrı bir yazmadan ibarettir ve $O(1)$ sürer. Dizi doluysa `realloc` bütün öğeleri kopyalayabilir; bu çağrı $O(n)$ sürer. Yalnız en kötü duruma bakılırsa n ekleme için $O(n^2)$ gibi karamsar bir sonuç çıkar. Oysa pahalı çağrılar seyrektir: kapasite her seferinde ikiye katlandığı için büyütme yalnız 1, 2, 4, 8, … öğeden sonra gerekir. Uzun bir işlem dizisinin toplam maliyetini işlem sayısına bölerek işlem başına maliyeti bulmaya **amortize analiz** (*amortized analysis*) denir.

> **Teorem 2.1 (İkiye katlamanın maliyeti)**
>
> Boş bir dinamik diziye kapasiteyi ikiye katlayarak $n \geq 1$ öğe eklenirse toplam kopya sayısı $2n$'den azdır. Yazmalar da sayıldığında toplam iş $3n$'den azdır. Bu yüzden ekleme başına amortize maliyet $O(1)$'dir.

İspat. Büyütmeler 1, 2, 4, …, 2^k öğe dolduğunda olur; burada 2^k, n'den küçük en büyük ikinin kuvvetidir. Her büyütmede o an dizide bulunan öğeler kopyalanır. Toplam kopya geometrik bir seridir: $1 + 2 + 4 + \dots + 2^k = 2^{k+1} - 1$. $2^k < n$ olduğundan bu toplam $2n - 1$'den küçüktür. Her öğe ayrıca bir kez yazılır; bu da n iş eder. Toplam $3n$'den azdır ve n'ye bölününce ekleme başına sabit bir sayı kalır. ∎

Kapasiteyi ikiye katlamak yerine her seferinde bir hücre artırmak ilk bakışta daha tutumlu görünür. Ama bu durumda her eklemede büyütme gerekir. i. eklemede $i - 1$ öğe kopyalanır ve toplam kopya $0 + 1 + \dots + (n-1) = n(n-1)/2$ olur. Bu $O(n^2)$'dir. Sabit bir c kadar artırmak da durumu düzeltmez: büyütme sayısı n/c olur ve toplam kopya yine n^2 ile orantılı kalır. Önemli olan artışın sabit değil, mevcut kapasiteyle orantılı olmasıdır. Tablo 2.5 iki stratejiyi birkaç n için karşılaştırır.

n ekleme	Katlama: kopya	Kopya / n	+1 büyütme: kopya	Kopya / n
8	7	0,88	28	3,5
64	63	0,98	2.016	31,5
1.024	1.023	1,00	523.776	511,5

Tablo 2.5: Boş diziye n ekleme yapıldığında toplam kopya sayısı. Değerler bir sayma programıyla üretilmiştir.

Katlamada ekleme başına kopya sayısı 1'in altında kalır; n ikinin kuvveti olduğunda kopya tam $n - 1$'dir. Bir artırmada ise ekleme başına kopya $(n-1)/2$'dir ve n ile birlikte büyür. Bin öğede fark beş yüz kattan fazladır.

Amortize sonuç, her tek eklemenin hızlı olduğu anlamına gelmez. Büyütme anına denk gelen ekleme yine $O(n)$ sürer. Amortize analiz uzun bir işlem dizisinin ortalamasını garanti eder; olasılığa dayanmaz, en kötü işlem dizisi için de geçerlidir. Gerçek zamanlı bir sistemde tek bir işlemin süresi önemliyse bu ayrım dikkate alınmalıdır.

Katlamanın bir de bellek bedeli vardır. Büyütmeden hemen sonra kapasitenin yaklaşık yarısı boştur. Kapasite hiçbir zaman boyutun iki katını aşmaz; yani en kötü durumda hücrelerin yarısı kullanılmaz. Bazı kütüphaneler bu payı azaltmak için 1,5 gibi daha küçük bir çarpan kullanır. Çarpan 1'den büyük kaldığı sürece toplam kopya yine $O(n)$'dir; yalnız sabit değişir.

> **Sık yapılan hata**
>
> `a = realloc(a, yeni * sizeof *a);` yazmak. `realloc` başarısız olursa `NULL` döndürür ve bu değer `a`'ya yazılır. Eski blok hâlâ ayrılmış durumdadır ama adresi artık hiçbir değişkende yoktur: blok sızar ve içindeki veri kaybolur. Sonuç önce geçici bir değişkene alınmalı, `NULL` değilse asıl işaretçiye atanmalıdır.

Not için yazılan bir denemede `malloc` ve `realloc` sayaçlı bir sarmalayıcıyla değiştirildi ve üçüncü `realloc` bilerek başarısız yapıldı. `ekle` 0 döndürdü, dizi iki öğesiyle (0, 1) sağlam kaldı ve `free`'den sonra hiç ayrılmış blok kalmadı.

2.3.4 İki boyutlu dinamik dizi: dizi2b()

Satır ve sütun sayısı çalışma anında belli olan bir matris için en yaygın yöntem, bir **işaretçi dizisi** kurmaktır. Önce her satır için bir işaretçi tutan bir dizi ayrılır, sonra her satır ayrı bir blok olarak ayrılır ve adresi bu diziye yazılır. Matrisin adresi, işaretçi dizisinin adresidir. Bu adresin tipi `int **`, yani **işaretçiye işaretçidir** (*pointer to pointer*).

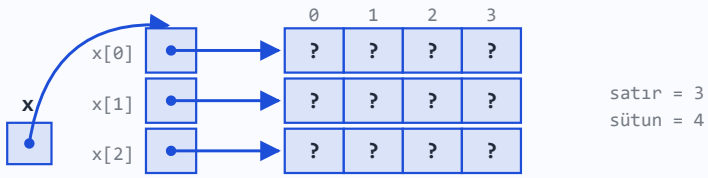
```
int **dizi2b(int satir, int sutun) {
  int i, **x = malloc(satir * sizeof *x);
  if (x == NULL) return NULL;
  for (i = 0; i < satir; i++) {
    x[i] = malloc(sutun * sizeof **x);
    if (x[i] == NULL) {               /* geri al */
      while (--i >= 0) free(x[i]);
      free(x); return NULL;
    }
  }
  return x;
}
```

Kod satır satır şöyle çalışır:

- `malloc(satir * sizeof *x)`, `satir` tane işaretçilik bir blok ayırır. `x` bir `int **` olduğu için `*x` bir `int *`'dır ve `sizeof *x` işaretçi boyunu verir. Bu blok ayrılamazsa fonksiyon hemen `NULL` döndürür.
- Döngü her satır için `sutun` tane `int` ayırır. `**x` bir `int` olduğundan `sizeof **x` tam sayı boyudur. İki `sizeof` ifadesi birbirinden farklıdır ve karıştırılmaması gerekir.
- Bir satır ayrılamazsa o ana kadar ayrılanlar geri verilir. `while (--i >= 0)` önce `i`'yi azaltır, sonra denetler; böylece başarısız olan `i`. satır atlanır ve *i − 1, …, 0* satırları serbest bırakılır. En son işaretçi dizisi geri verilir ve `NULL` döner. Bu adım yazılmazsa başarısızlıkta ayrılmış satırlar sızar.
- Bütün satırlar ayrıldıysa `x` döner. Çağıran taraf `x[i][j]` yazımını sıradan bir iki boyutlu dizideki gibi kullanabilir.

Adım	İşlem	Sonuç
1	`x = malloc(3 * sizeof(int *))`	3 hücrelik işaretçi dizisi; içerik belirsiz
2	`x == NULL` ?	hayır, devam
3	`i = 0` : `x[0] = malloc(4 * sizeof(int))`	`x[0]` 4 `int`'lik bloğu gösterir
4	`i = 1` : `x[1] = malloc(…)`	`x[1]` ikinci bloğu gösterir
5	`i = 2` : `x[2] = malloc(…)`	`x[2]` üçüncü bloğu gösterir
6	`i = 3`, `3 < 3` yanlış	döngü biter, `x` döner

Tablo 2.6: `dizi2b(3, 4)` çağrısının izlemesi.



Şekil 2.3: `dizi2b(3, 4)` 'ün kurduğu yapı. `x`, üç işaretçilik bir diziyi gösterir; her işaretçi dört `int` 'lik ayrı bir satır bloğunu gösterir. Satırların içeriği henüz belirsizdir (?).

Şekil 2.3 yapının bellekteki görünümünü verir. `x[i][j]` ifadesi iki adımda çözülür: `x[i]`, yani `*(x + i)`, *i*. satırın adresini getirir; `[j]` bu adresten *j* öğe ilerler. Tam açık yazılışı `*(*(x + i) + j)` 'dir. Sürücü programda `x[2][3] = 23` ve `*(*(x + 1) + 2) = 12` değerleri, satırlara *10i + j* yazıldıktan sonra beklendiği gibi okunmuştur.

Bu yapı ile `int a[3][4];` gibi gerçek bir iki boyutlu dizi arasında önemli bir fark vardır. Gerçek dizide on iki öğenin hepsi tek bir ardışık blokta durur. İşaretçi dizisinde ise her satır ayrı bir `malloc` çağrısıyla ayrıldığından satırlar bellekte ardışık olmak zorunda değildir. Bu iki sonuç doğurur. Birincisi, satırlar farklı uzunlukta olabilir; alt üçgen bir matrisin *i*. satırı için yalnız *i + 1* hücre ayırmak mümkündür. İkincisi, yapı bir adım daha fazla dolaylı erişim içerir ve bütün matrisi tek bir `memcpy` ile kopyalamak mümkün değildir.

Böyle bir matris geri verilirken sıra önemlidir. Önce her `x[i]` satırı, sonra `x` işaretçi dizisi `free` edilir. Sıra tersine çevrilirse `x` serbest bırakıldıktan sonra `x[i]` okunmuş olur; bu, geri verilmiş belleğe erişimdir. Yalnız `free(x)` yazılırsa satırların adresleri kaybolur ve hepsi sızar.

Sık yapılan hata

İki boyutlu dinamik diziyi yalnız `free(x)` ile bırakmak. Üç satırlık bir matriste bu, üç bloğu sızdırır. Doğru sıra: `for (i = 0; i < satir; i++) free(x[i]); free(x);`. Not için yazılan sayaçlı denemede `dizi2b` 'nin her ayırmada başarısız olduğu durumlar tek tek denendi; her durumda fonksiyon `NULL` döndürdü ve geride ayrılmış blok kalmadı.

Maliyet açısından `dizi2b` *satir + 1* tane `malloc` çağrısı yapar. Her çağrının süresi sabit kabul edilirse fonksiyon *O(satir)* sürer. Ayrılan bellek *satir · sutun* tam sayı ile *satir* işaretçinin toplamıdır. Öğelere erişim, iki dolaylı adım içerse de *O(1)*'dir.

2.4 Çok boyutlu diziler ve adres hesabı

Bir tablo, bir resim ya da bir oyun tahtası iki indisle; bir hacim ya da bir video üç indisle doğal biçimde ifade edilir. Öğelerine birden çok indisle erişilen diziye **çok boyutlu dizi** (*multidimensional array*) denir. Bellek ise tek boyutludur: her baytın tek bir adresi vardır. Bu kesim, çok boyutlu bir dizinin tek boyutlu belleğe nasıl dizildiğini ve bir öğenin adresinin nasıl hesaplandığını inceler.

2.4.1 Bellek tek boyutludur

$A[u_0][u_1]\ldots[u_{n-1}]$ olarak bildirilen *n* boyutlu bir dizide $u_0 \cdot u_1 \cdot \ldots \cdot u_{n-1}$ öğe vardır. Örneğin `A[10][10][10]` bin öğe tutar. Derleyici bu öğeleri belleğe bir sıraya koymak zorundadır. İki temel sıra vardır.

Tanım 2.2

Satır öncelikli düzende (*row-major order*) önce 0. satırın bütün öğeleri, sonra 1. satırın öğeleri yerleştirilir; en sağdaki indis en hızlı değişir. **Sütun öncelikli düzende** (*column-major order*) önce 0. sütunun bütün öğeleri, sonra 1. sütunun öğeleri yerleştirilir; en soldaki indis en hızlı değişir.

C, C++ ve Python'daki NumPy kütüphanesi varsayılan olarak satır öncelikli düzeni kullanır. Fortran, MATLAB ve R sütun önceliklidir. İki dil arasında matris aktaran bir programda bu farkın gözden kaçması, matrisin farkında olmadan transpozunun alınmasıyla sonuçlanır.

Hangi düzen seçilirse seçilsin adres hesabı aynı soruya dayanır: *A[i][j] öğesinden önce bellekte kaç öğe vardır?* Bu sayıya *k* denirse öğenin adresi $\alpha + k \cdot s$ olur. Burada α dizinin başlangıç adresi, *s* bir öğenin bayt cinsinden boyudur. Tek boyutlu dizide $k = i$'dir; çok boyutlu dizide *k* birkaç terimin toplamıdır.

C'de `int x[3][4];` bildirimi aslında dizilerden oluşan bir dizidir: her biri dört `int` 'lik üç öğe. `x[1]` ifadesi 1. satırın, yani dört öğelik bir dizinin başını gösterir. Sürücü programda `x[1] - x[0]` farkı 4 öğe çıkmıştır: iki satır başı arasında tam bir satır vardır. Bu yapı, satır öncelikli düzenin dilin tip sisteminde doğrudan görünmesidir.

2.4.2 Satır ve sütun öncelikli formüller

`int x[3][4]` dizisini ele alalım; satır sayısı $u_0 = 3$, satır uzunluğu $u_1 = 4$'tür. Satır öncelikli düzende $x[i][j]$'den önce i tam satır bulunur. Her satırda u_1 öğe olduğundan bu satırlar $i \cdot u_1$ öğe eder. Öğenin kendi satırında ise solunda j öğe vardır. Toplam:

$$\text{satır öncelikli: } \&x[i][j] = \alpha + (i \cdot u_1 + j) \cdot s$$

Sütun öncelikli düzende ise önce j tam sütun gelir. Her sütunda u_0 öğe vardır; bunlar $j \cdot u_0$ öğe eder. Öğenin sütunu içinde üstünde i öğe bulunur:

$$\text{sütun öncelikli: } \&x[i][j] = \alpha + (j \cdot u_0 + i) \cdot s$$



Şekil 2.4: `int x[3][4]` dizisinin iki yerleşimi. Matris hücrelerinin köşesindeki küçük sayılar, öğenin doğrusal dizideki konumudur. Üstte satır öncelikli düzen: önce 0. satırın dört öğesi gelir. Altta sütun öncelikli düzen: önce 0. sütunun üç öğesi gelir.

Şekil 2.4 aynı öğenin iki düzende farklı konuma düştüğünü gösterir. x_{12} satır öncelikli düzende $1 \cdot 4 + 2 = 6$. konumda, sütun öncelikli düzende $2 \cdot 3 + 1 = 7$. konumdadır. İki formül aynı mantığı izler; yalnız hangi indisin "dış", hangisinin "iç" olduğu değişir.

Formüldeki parantez içindeki ifade öğe birimindedir. Bayt adresine geçmek için öğe boyu s ile çarpılır. C'de işaretçi aritmetiği bu çarpmayı kendisi yaptığından `&x[0][0] + 6` zaten $6 \cdot s$ bayt ileridedir. Elle bayt adresi hesaplanırken ise s unutulmamalıdır.

> **Sık yapılan hata**
>
> Satır öncelikli formülde satır sayısını (u_0) kullanmak. i tam satırın kapladığı yer, satır *uzunluğuna* (u_1, yani sütun sayısına) bağlıdır. Formülde satır sayısı hiç geçmez. Kare matrislerde iki değer eşit olduğu için bu hata gizli kalır; 3×4 gibi kare olmayan bir matriste hemen yanlış adres verir.

2.4.3 Kod izleme: adres()

Satır öncelikli formül küçük bir fonksiyonla yazılabilir. Fonksiyon başlangıç adresini, iki indisi, satır uzunluğunu ve öğe boyunu alır, bayt adresini döndürür.

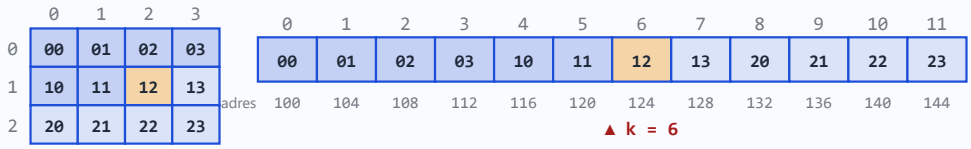
```c
/* int x[3][4]: satır öncelikli bayt adresi */
long adres(long alfa, int i, int j,
           int ust1, int s) {
  long k = (long)i * ust1 + j;   /* öncekiler */
  return alfa + k * s;
}

int main(void) {
  printf("%ld\n", adres(100, 1, 2, 4, 4));
  printf("%ld\n", adres(100, 2, 3, 4, 4));
  return 0;
}
```

`k`, öğeden önceki öğelerin sayısıdır. `(long)i` dönüşümü çarpmanın `long` türünde yapılmasını sağlar; çok büyük dizilerde `int` çarpımı taşabilir. Fonksiyon `alfa + k * s`'yi döndürür. `main` iki öğenin adresini sorar: *x[1][2]* ve *x[2][3]*; dizi 100 adresinden başlar, $u_1 = 4$ ve $s = 4$'tür.

Çağrı	`k = i · ust1 + j`	Dönüş `alfa + k · s`	Çıktı
`adres(100, 1, 2, 4, 4)`	$1 \cdot 4 + 2 = 6$	$100 + 6 \cdot 4 = 124$	124
`adres(100, 2, 3, 4, 4)`	$2 \cdot 4 + 3 = 11$	$100 + 11 \cdot 4 = 144$	144

Tablo 2.7: `adres` fonksiyonunun iki çağrısı ($\alpha = 100$, $u_1 = 4$, $s = 4$).

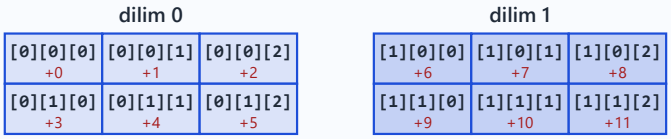


Şekil 2.5: *x[1][2]*'nin adresi. Öğeden önce bir tam satır (4 öğe) ve kendi satırında 2 öğe vardır; $k = 6$, adres $100 + 6 \cdot 4 = 124$.

İkinci çağrıdaki *x[2][3]* dizinin son öğesidir. $k = 11$, yani on iki öğeli dizide son konum çıkar. Program derlenip çalıştırıldığında 124 ve 144 yazdırır. Sütun öncelikli düzen kullanılsaydı *x[1][2]* için $k = 2 \cdot 3 + 1 = 7$ ve adres 128 olurdu. Sürücü programda gerçek bir `int x[3][4]` dizisinde `&x[1][2] - &x[0][0]` farkı da 6 öğe çıkmıştır; C'nin satır öncelikli olduğu böylece doğrudan görülür. Fonksiyon girdiye bağlı hiçbir döngü içermez; süresi *O(1)*'dir.

2.4.4 Üç boyutlu dizi

Üç boyutlu bir dizi, iki boyutlu dizilerden oluşan bir dizidir. `G[2][2][3]` dizisinde iki **dilim** vardır; her dilim 2 satır ve 3 sütunluk bir matristir. Satır öncelikli düzende önce 0. dilimin altı öğesi, sonra 1. dilimin altı öğesi gelir. Bir dilimin içi de satır satır dizilir.



Şekil 2.6: `G[2][2][3]` dizisinin iki dilimi. Her hücrede öğenin indisleri ve altında doğrusal konumu (*+0* … *+11*) yazar.

G[i][j][k]'dan önceki öğeler yine katman katman sayılır. Önce i tam dilim vardır; her dilim $u_1 \cdot u_2$ öğe eder. Sonra kendi diliminde j tam satır, her biri u_2 öğe. En sonda kendi satırında k öğe. Tablo 2.8 bu terimlerin adım adım nasıl eklendiğini gösterir; adresler öğe biriminde dir.

Öğe	Adres
[0][0][0]	α
[i][0][0]	$\alpha + i \cdot u_1 \cdot u_2$

Öğe	Adres
[i][j][0]	$\alpha + i \cdot u_1 \cdot u_2 + j \cdot u_2$
[i][j][k]	$\alpha + i \cdot u_1 \cdot u_2 + j \cdot u_2 + k$

Tablo 2.8: Üç boyutlu dizide öğe adresinin oluşumu (öğe birimi).

Son satırdaki ifadede üç çarpma ve üç toplama vardır. Çarpma toplamadan daha pahalı bir işlem olduğundan derleyici ifadeyi ortak çarpanı paranteze alarak yeniden yazar:

$$\alpha + (i \cdot u_1 + j) \cdot u_2 + k$$

Bu biçimde iki çarpma ve üç toplama kalır. `G[1][0][2]` için hesap *(1 · 2 + 0) · 3 + 2 = 8* verir. Şekil 2.6 ile karşılaştırınca *[1][0][2]* hücresinin altında gerçekten *+8* yazdığı görülür. Sürücü programda gerçek bir `int G[2][2][3]` dizisinde `&G[1][0][2] - &G[0][0][0]` farkı da 8 çıkmıştır.

2.4.5 n boyutlu genelleme

Aynı düşünce her boyut sayısına genişler. *A[u₀][u₁]…[uₙ₋₁]* dizisinde *A[i₀][i₁]…[iₙ₋₁]*'den önce, en dıştaki indis için i_0 tam "dilim" vardır ve her biri $u_1\ u_2 \ldots u_{n-1}$ öğe eder. Bir içteki indis için i_1 tane $u_2 \ldots u_{n-1}$ öğelik blok vardır. En içteki indis yalnız i_{n-1} öğe ekler.

Öğe	Adres
A[0][0]…[0]	α
A[i_0][0]…[0]	$\alpha + i_0 \cdot u_1\ u_2 \ldots u_{n-1}$
A[i_0][i_1]…[0]	önceki satır + $i_1 \cdot u_2 \ldots u_{n-1}$
A[i_0]…[i_{n-1}]	önceki satır + i_{n-1}

Tablo 2.9: *n* boyutlu dizide adres terimleri (öğe birimi).

Terimler bir toplam olarak yazılabilir. *j*. indisin katsayısı, ondan sonra gelen bütün boyutların çarpımıdır:

$$\&A[i_0]\ldots[i_{n-1}] = \alpha + \Sigma_{j=0}^{n-1}\, i_j \cdot a_j, \quad a_j = u_{j+1} \cdot u_{j+2} \cdot \ldots \cdot u_{n-1}\ (0 \le j < n-1), \quad a_{n-1} = 1$$

a_j katsayıları yalnız boyutlara bağlıdır, indislere bağlı değildir. Derleyici onları bir kez, derleme sırasında hesaplar. Formülün dikkat çekici bir sonucu, u_0'ın hiçbir katsayıda geçmemesidir. En dıştaki boyutun uzunluğu adres hesabına girmez. C'de bir fonksiyon parametresi `int a[][4]` diye yazılabilir ama `int a[3][]` diye yazılamaz; nedeni budur. İlk boyut adres için gereksizdir, sonrakiler ise zorunludur.

Her terimi ayrı hesaplamak çok sayıda çarpma gerektirir. **Horner yöntemi** (*Horner's method*), bir polinomu iç içe çarpımlarla hesaplayan yöntemdir ve burada da işe yarar. Adres terimi şöyle yeniden yazılabilir:

$$(\ldots((i_0 \cdot u_1 + i_1) \cdot u_2 + i_2) \cdot u_3 + \ldots) \cdot u_{n-1} + i_{n-1}$$

Hesap $t = i_0$ ile başlar ve her boyut için $t = t \cdot u_j + i_j$ yapılır. Döngü $n - 1$ kez döner, her turda bir çarpma ve bir toplama yapılır. Toplam $n - 1$ çarpma yeter. Üç boyutlu dizideki "paranteze alma" bu yöntemin $n = 3$ için özel hâlidir.

2.4.6 Kod izleme: ofset()

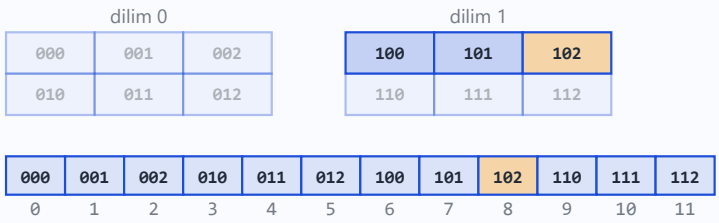
Horner yöntemi, indis dizisini ve boyut dizisini alan genel bir fonksiyonla yazılır. Fonksiyon öğe biriminde bağıl konumu döndürür; bayt adresi `alfa + t * s` ile bulunur.

```c
/* ind: indisler, ust: boyutlar, n: boyut sayısı */
int ofset(const int ind[], const int ust[], int n) {
  int j, t = ind[0];
  for (j = 1; j < n; j++)
    t = t * ust[j] + ind[j];            /* Horner */
  return t;               /* adres = alfa + t*s */
}
```

Parametrelerdeki `const`, fonksiyonun dizileri değiştirmeyeceğini belgeler. `t` en dıştaki indisle başlar. Döngü `j = 1`'den başladığı için `ust[0]` hiç okunmaz. En dıştaki boyutun adres hesabına girmediği kodda da böylece görülür.

Adım	`j`	İşlem	`t`	Anlamı
başlangıç	—	`t = ind[0]`	1	1 numaralı dilim
1	1	$t = 1 \cdot 2 + 0$	2	dilimler boyunca sayılan “genel satır” numarası (dilim 1, satır 0)
2	2	$t = 2 \cdot 3 + 2$	8	öğeden önce 8 öğe var
son	3	`3 < 3` yanlış	8	döngü biter, 8 döner

Tablo 2.10: `G[2][2][3]` dizisinde `ofset` ile *G[1][0][2]*'nin konumu (`ind = {1, 0, 2}`, `ust = {2, 2, 3}`, `n = 3`).



Şekil 2.7: `ofset`'in sonucu. *G[1][0][2]* (turuncu) doğrusal dizinin 8 numaralı hücresidir.

Ara değer $t = 2$'nin anlamı ilginçtir. İki dilimin satırları art arda sayılırsa (dilim 0'ın satırları 0 ve 1, dilim 1'in satırları 2 ve 3) aranan öğe 2 numaralı satırdadır. Son adımda bu satır numarası satır uzunluğuyla (3) çarpılıp sütun indisi eklenince öğenin konumu çıkar. Hesap toplam iki çarpma kullanır, yani $n - 1$. Doğrudan formül de aynı sonucu verir: $1 \cdot 2 \cdot 3 + 0 \cdot 3 + 2 = 8$. Sürücü programda `A[4][5][6]` dizisi için *[2][3][4]*'ün bağıl konumu da denenmiş ve fonksiyonun sonucu (82) gerçek dizideki farkla aynı çıkmıştır. `ofset`'in süresi boyut sayısıyla doğrusaldır, $\Theta(n)$. Boyut sayısı bir dizi için sabit olduğundan erişim, öğe sayısına göre $O(1)$'dir.

2.5 Üçgen matrisler

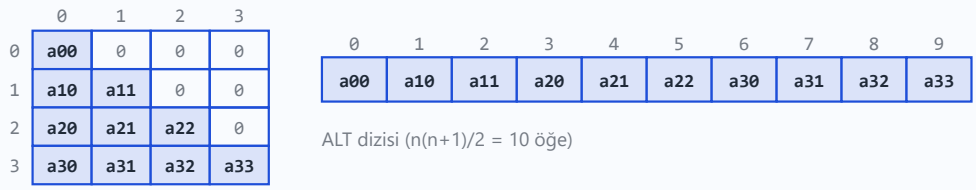
Adres formülü yalnız derleyicinin işine yaramaz. Programcı da belleği kendi düzenleyip öğelerin yerini bir formülle bulabilir. Bu fikrin en sade uygulaması, yarısı boş olan matrisleri yalnız dolu yarılarıyla saklamaktır.

2.5.1 Alt üçgen matris

Kare bir matris ana köşegeniyle ikiye bölünür. Bazı uygulamalarda yarılardan biri yeter. Şehirler arası uzaklık tablosunda *i*'den *j*'ye uzaklık *j*'den *i*'ye uzaklıkla aynıdır; tablonun yarısı öbür yarısının tekrarıdır. Doğrusal denklem sistemlerinin çözümünde de köşegenin üstü sıfır olan matrisler sık çıkar.

> **Tanım 2.3**
>
> **Alt üçgen matris** (*lower triangular matrix*), ana köşegenin üstündeki bütün öğeleri sıfır olan kare matristir: $i < j$ ise $a_{ij} = 0$.

Sıfırları saklamaya gerek yoktur. Matrisin geri kalanı tek boyutlu bir `ALT` dizisinde satır satır tutulur. Kullanıcıya ise iki boyutlu bir matrisle çalışıyormuş gibi bir arayüz sunulur: a_{ij} istendiğinde fonksiyon onun `ALT`'taki konumunu hesaplar.



Şekil 2.8: *4 × 4* alt üçgen matris ve onu satır satır tutan 10 öğeli `ALT` dizisi. Köşegenin üstündeki altı sıfır saklanmaz.

Formülü bulmak için yine aynı soru sorulur: a_{ij}'den önce `ALT`'ta kaç öğe vardır? *i*. satırda *i + 1* öğe saklanır (sütun 0'dan *i*'ye kadar). Önceki satırlar $1 + 2 + \dots + i = i(i+1)/2$ öğe eder. Öğenin kendi satırında solunda *j* öğe vardır:

$$k = i(i+1)/2 + j \quad (j \le i)$$

$n \times n$ bir alt üçgen matriste toplam $1 + 2 + \dots + n = n(n+1)/2$ öğe saklanır. Tam matris n^2 hücre ister. Kazanç $n(n-1)/2$ hücredir ve büyük n'de belleğin yaklaşık yarısı demektir. Şekil 2.8 üzerinde formül denenebilir: a_{21} için $k = 2 \cdot 3/2 + 1 = 4$ bulunur ve `ALT[4]` gerçekten a_{21}'dir.

2.5.2 Kod izleme: AltUcgenOku()

Aşağıdaki fonksiyon $n \times n$ bir alt üçgen matrisin saklanan öğelerini standart girdiden satır satır okur ve `alt` dizisine yerleştirir.

```
#define MAX_SIZE 100
void AltUcgenOku(int alt[], int n) {
  int i, j, k;
  if (n * (n + 1) / 2 > MAX_SIZE) {
    printf("Alt dizi boyutu yetersiz!\n"); exit(1);
  }
  for (i = 0; i < n; i++) {
    k = (i + 1) * i / 2;      /* öncekiler */
    for (j = 0; j <= i; j++)  /* j <= i */
      scanf("%d", &alt[k + j]);
  }
}
```

Kod satır satır şöyle çalışır:

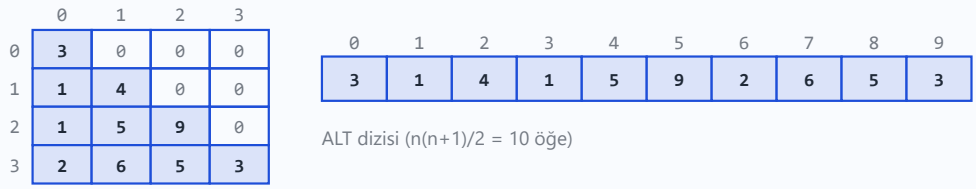
- Önce saklanacak öğe sayısı *n(n + 1)/2*, dizinin kapasitesi `MAX_SIZE` ile karşılaştırılır. Sığmazsa ileti yazılır ve program sonlanır. Bu denetim olmasaydı büyük bir *n* dizinin sonundan taşmaya yol açardı.
- Dış döngü satırları gezer. `k = (i + 1) * i / 2`, *i*. satırdan önce yerleşmiş öğe sayısıdır; yani satırın `alt`'taki başlangıcıdır. Bölme çarpımdan sonra yapılır; *i(i + 1)* her zaman çift olduğundan bölme tamdır.
- İç döngü yalnız `j <= i` için döner; köşegenin üstündeki sıfırlar okunmaz. Her değer `alt[k + j]`'ye yazılır.

> **Örnek 2.2**
>
> *n = 4* için girdi `3 1 4 1 5 9 2 6 5 3` olsun. Önce $4 \cdot 5/2 = 10 \le 100$ denetimi geçilir. Tablo 2.11 her değerin nereye yazıldığını gösterir.

`i`	`k`	`j`	Konum `k + j`	Okunan değer	Matris öğesi
0	$1 \cdot 0/2 = 0$	0	0	3	a_{00}
1	$2 \cdot 1/2 = 1$	0	1	1	a_{10}
1	1	1	2	4	a_{11}
2	$3 \cdot 2/2 = 3$	0	3	1	a_{20}
2	3	1	4	5	a_{21}

`i`	`k`	`j`	Konum `k + j`	Okunan değer	Matris öğesi
2	3	2	5	9	a_{22}
3	$4 \cdot 3/2 = 6$	0	6	2	a_{30}
3	6	1	7	6	a_{31}
3	6	2	8	5	a_{32}
3	6	3	9	3	a_{33}

Tablo 2.11: `AltUcgenOku(alt, 4)` çağrısının izlemesi.



Şekil 2.9: Okuma bittiğinde matris ve `ALT` dizisi. On değer okundu; üst üçgendeki altı sıfır hiç saklanmadı.

Fonksiyonun gövdesi her saklanan öğe için bir `scanf` yapar. İç döngünün toplam tur sayısı $1 + 2 + \ldots + n = n(n + 1)/2$'dir. Süre saklanan öğe sayısıyla orantılıdır, $\Theta(n^2)$. Tam matrisi okumak n^2 tur sürerdi; üçgen gösterim sabit çarpanı yarıya indirir ama büyüme hızını değiştirmez. Sürücü programda `alt` dizisi okumadan sonra `3 1 4 1 5 9 2 6 5 3` olarak yazdırılmış ve tablo doğrulanmıştır.

2.5.3 Erişim ve hata kodları: AltUcgenEris()

Okumadan sonra matrise erişmek için (i, j) çiftini `ALT` içindeki konuma çeviren bir fonksiyon gerekir. Fonksiyon iki istisnayı da ayırt etmelidir: indisler matrisin dışında olabilir ya da istenen öğe üst üçgende, yani saklanmayan sıfırlı bölgede olabilir.

```
int AltUcgenEris(int i, int j, int n) {
  if (i < 0 || i >= n || j < 0 || j >= n) {
    printf("Gecersiz indis\n");
    return -2;
  } else if (i < j) return -1;    /* üst üçgen: 0 */
  else return (i + 1) * i / 2 + j;   /* geçerli */
}
/* main: */
printf("%d\n", AltUcgenEris(2, 1, 4));
printf("%d\n", AltUcgenEris(1, 3, 4));
printf("%d\n", AltUcgenEris(4, 0, 4));
```

Fonksiyon önce sınırları denetler. İndislerden biri $[0, n - 1]$ aralığının dışındaysa ileti yazar ve −2 döndürür. Sonra $i < j$ olup olmadığına bakar. Öyleyse öğe üst üçgendedir; değeri sıfırdır ama `ALT`'ta yeri yoktur ve fonksiyon −1 döndürür. Geriye kalan durumda öğe alt üçgendedir ve konum formülle hesaplanır.

Çağrı	Sınır denetimi	`i < j` ?	Dönüş	Çıktı
`AltUcgenEris(2, 1, 4)` , a_{21}	geçerli	2 < 1, hayır	$3 \cdot 2/2 + 1 = 4$	4
`AltUcgenEris(1, 3, 4)` , a_{13}	geçerli	1 < 3, evet	−1 (üst üçgen)	-1
`AltUcgenEris(4, 0, 4)` , a_{40}	$i = 4 \geq n$, geçersiz	—	−2	Gecersiz indis, -2

Tablo 2.12: `AltUcgenEris` fonksiyonunun üç çağrısı ($n = 4$).

Sürücü programın çıktısı `4` , `-1` , `Gecersiz indis` ve `-2` satırlarından oluşur. Dönüş değerlerinin seçimi bilinçlidir. Geçerli konumlar 0, 1, 2, … gibi negatif olmayan sayılardır. Hata kodları {−1, −2} bu kümeyle kesişmez. Bu yüzden çağıran taraf dönüş değerine bakarak durumu her zaman ayırt edebilir: negatif değer bir istisna, sıfır ya da pozitif değer gerçek bir konumdur. Hata kodu olarak örneğin 0 seçilseydi, a_{00}'ın konumuyla karışırdı.

Fonksiyon girdiye bağlı döngü içermez: birkaç karşılaştırma ve bir formül hesabı yapar. Süresi *O(1)*'dir. Tam matristeki `a[i][j]` erişimi kadar hızlıdır, ama yarı bellekle çalışır.

Sık yapılan hata

Konum formülünü `i * (i + 1) / 2 + j` yerine `i * i / 2 + j` ya da `(i + 1) * (i + 2) / 2 + j` yazmak. İlki satır başını eksik verir ($i = 2$'de 3 yerine 2), ikincisi bir sonraki satırın başını verir. Formül, küçük bir matriste köşe öğeleriyle denenerek kolayca sınanır: $a_{00} \to 0$, $a_{10} \to 1$, $a_{n-1,n-1} \to n(n+1)/2 - 1$.

2.5.4 Öteki üçgenler ve simetrik matris

Saklanan yarı, köşegenin hangi yanında olduğuna göre dört biçimde olabilir. Her birinde konum formülü aynı yöntemle çıkarılır: önceki satırlardaki öğeler toplanır, sonra satır içindeki sıra eklenir. Tablo 2.13 dört durumun sonucunu verir.

Saklanan bölge	*i*. satırdaki öğe	Konum *k*
$j \le i$ (alt sol)	$i + 1$	$i(i+1)/2 + j$
$j \ge i$ (üst sağ)	$n - i$	$i \cdot n - i(i-1)/2 + (j - i)$
$i + j \le n - 1$ (sol üst)	$n - i$	$i \cdot n - i(i-1)/2 + j$
$i + j \ge n - 1$ (sağ alt)	$i + 1$	$i(i+1)/2 + j - (n - 1 - i)$

Tablo 2.13: Dört üçgen türünde satır öncelikli konum formülü ($n \times n$ matris).

Üst sağ üçgende r. satırda $n - r$ öğe vardır. i. satırdan önceki toplam $n + (n-1) + \dots + (n - i + 1) = i \cdot n - i(i-1)/2$'dir. Satırın ilk saklanan öğesi köşegendedir ($j = i$), bu yüzden satır içindeki sıra $j - i$'dir. Sol üst üçgende satır uzunlukları aynıdır ama satır 0. sütundan başlar; sıra doğrudan j olur. Sağ alt üçgende i. satır $i + 1$ öğe tutar ve $j = n - 1 - i$ sütununda başlar; sıra $j - (n - 1 - i)$'dir. Not için yazılan bir döngü, $n = 1$'den 8'e kadar her matriste dört formülü denemiştir. Formüllerin verdiği konumlar, satır satır sayılan konumlarla hep aynı çıkmıştır.

Gerçek bir **simetrik matriste** ($a_{ij} = a_{ji}$) üst üçgen sıfır değil, alt üçgenin aynasıdır. Bu durumda `AltUcgenEris` üst üçgen için −1 döndürmek yerine indisleri yer değiştirir: $i < j$ ise $k = j(j+1)/2 + i$. Uzaklık tablosu ya da bir çizgenin yönsüz komşuluk matrisi böyle saklanabilir. Bellek yine yarıya iner ve hiçbir bilgi kaybolmaz.

2.6 Kuşak matrisler

Üçgen matriste saklanan bölge köşegenin bir yanıydı. Pek çok sayısal problemde ise sıfırdan farklı öğeler köşegenin iki yanında, dar bir şeritte toplanır. Isı iletimi ya da titreşim gibi fizik problemlerinin sayısal çözümünde her bilinmeyen yalnız komşularıyla ilişkilidir; ortaya çıkan matrisin köşegenden uzak öğeleri sıfırdır. Böyle bir matrisi tam saklamak belleğin büyük kısmını sıfırlara harcamak demektir.

2.6.1 Tanım ve öğe sayısı

Tanım 2.4

Kuşak matris (*band matrix*), anlamlı öğeleri ana köşegen çevresindeki bir şeritte toplanan kare matristir. Şeridin dışındaki öğeler kullanılmaz ya da hepsi aynı sabit c değerindedir (çoğu zaman $c = 0$). Matris üç sayıyla betimlenir: n kenar uzunluğu, a köşegen dâhil alt kuşak sayısı, b köşegen dâhil üst kuşak sayısı.



Şekil 2.10: *n = 4, a = 3, b = 2* olan kuşak matris. Solda matris (kuşak dışı öğeler *c*), sağda öğeleri kuşak kuşak tutan `kusak` dizisi ve her kuşağın başlangıcını tutan `ara` dizisi.

Şekil 2.10'daki matriste köşegen ve altındaki iki kuşak (*a = 3*) ile köşegenin üstündeki bir kuşak (*b = 2*, köşegen dâhil) saklanır. d_{02}, d_{03}, d_{13} ve d_{30} kuşak dışında kalır.

Saklanan öğe sayısını bulmak için kuşaklar tek tek sayılır. Ana köşegende *n* öğe vardır. Köşegenden bir uzak kuşakta (altta ya da üstte) *n − 1*, iki uzak kuşakta *n − 2* öğe bulunur. Genel olarak köşegene uzaklığı *t* olan bir kuşakta *n − |t|* öğe vardır. Toplam *a + b − 1* kuşak saklanır, çünkü köşegen hem *a*'da hem *b*'de sayılmıştır. Her kuşak *n* öğeli olsaydı toplam *n(a + b − 1)* olurdu. Alt kuşaklardaki eksikler *1 + 2 + … + (a − 1) = a(a − 1)/2*, üst kuşaklardaki eksikler *b(b − 1)/2*'dir:

$$\text{öğe sayısı} = n(a + b - 1) - a(a-1)/2 - b(b-1)/2$$

Örnekte $4 \cdot 4 - 3 - 1 = 12$ öğe saklanır; tam matris 16 hücre isterdi. Kazanç küçük görünür, çünkü matris küçüktür. *n = 1000, a = b = 3* olan bir matriste ise tam matris bir milyon hücre isterken kuşak gösterimi $1000 \cdot 5 - 3 - 3 = 4994$ hücreyle yetinir.

Saklama sorunu böylece çözülür. Asıl soru, saklanan öğelere hızlı erişilip erişilemeyeceğidir. Öğeler sıkıştırıldığında konum hesabı da karmaşıklaşır. Kuşak matris gösteriminin değeri, erişimin hâlâ sabit sürede yapılabilmesine bağlıdır.

2.6.2 Kuşak numarası ve ARA dizisi

Erişimin anahtarı, bir öğenin hangi kuşakta olduğunu indislerinden hemen anlamaktır. Aynı kuşaktaki öğelerde *j − i* farkı sabittir. Örneğin d_{20} ve d_{31} aynı kuşaktadır: *0 − 2 = 1 − 3 = −2*. Bu farka **kuşak numarası** denir. Ana köşegenin numarası 0'dır; negatif numaralar alt kuşakları, pozitif numaralar üst kuşakları gösterir.

Geçerli kuşak numaraları *[1 − a, b − 1]* kapalı aralığındadır. Örnekte bu aralık *[−2, 1]*'dir. Numarası bu aralığın dışında kalan öğeler sabit bölgededir ve saklanmaz. Böylece bir öğenin kuşak içinde mi dışında mı olduğu tek bir çıkarma ve iki karşılaştırmayla anlaşılır.

Öğeler `kusak` dizisinde kuşak kuşak, en alttaki kuşaktan başlayarak saklanır. Bir öğenin konumu iki parçadan oluşur: kuşağının `kusak` içindeki başlangıcı ve öğenin kuşak içindeki sırası.

- **Kuşak başlangıcı.** Her kuşağın başlangıcı `ara` (ARA) dizisinde tutulur. Dizide *a + b − 1* öğe vardır, yani kuşak başına bir öğe. C'de dizi indisi negatif olamayacağı için kuşak numarası *a − 1* kaydırılarak kullanılır: *t* numaralı kuşağın başlangıcı `ara[t + a - 1]`'dedir. Örnekte `ara = {0, 2, 5, 9}` olur; −2 numaralı kuşak 0'dan, −1 numaralı kuşak 2'den, köşegen 5'ten, +1 numaralı kuşak 9'dan başlar.
- **Kuşak içindeki sıra.** Bir alt kuşağın ilk öğesi 0. sütundadır; kuşak boyunca her adımda sütun bir artar. Bu yüzden alt kuşakta sıra *j*'dir. Bir üst kuşağın ilk öğesi 0. satırdadır; sıra *i*'dir. Köşegende *i = j* olduğundan ikisi de aynı sonucu verir.

Tasarımın arkasındaki düşünce şudur: her şeyi kısıtlıymış gibi inceleyip erişim mekanizmasını bu kısıtlardan çıkarmak. Kuşak numarasının sabit olması, kuşak uzunluklarının *n − |t|* olması ve ilk öğenin yerinin bilinmesi birer kısıttır. Üçü birlikte, karmaşık görünen bir sıkıştırmayı birkaç aritmetik işleme indirger.

2.6.3 Kod izleme: KusakMatrisKur()

Aşağıdaki fonksiyon kuşak matrisin öğelerini standart girdiden okur. Girdi kuşak kuşak, en alttaki kuşaktan başlayarak gelir. Fonksiyon her kuşağın başlangıcını `ara`'ya yazar ve öğeleri `kusak`'a yerleştirir. `abs` fonksiyonu `<stdlib.h>` başlığındadır.

```c
void KusakMatrisKur(int kusak[], int ara[],
                    int n, int a, int b) {
  int i, k, toplamoge = 0;   /* önceki kuşaklar */
  if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
    printf("Bellek yetersiz!\n"); exit(-1);
  }
  for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
    ara[i + a - 1] = toplamoge;     /* kuşak başı */
    for (k = 0; k <= n - abs(i) - 1; k++)
      scanf("%d", &kusak[ara[i + a - 1] + k]);
    toplamoge += n - abs(i);
  }
}
```

Kod satır satır şöyle çalışır:

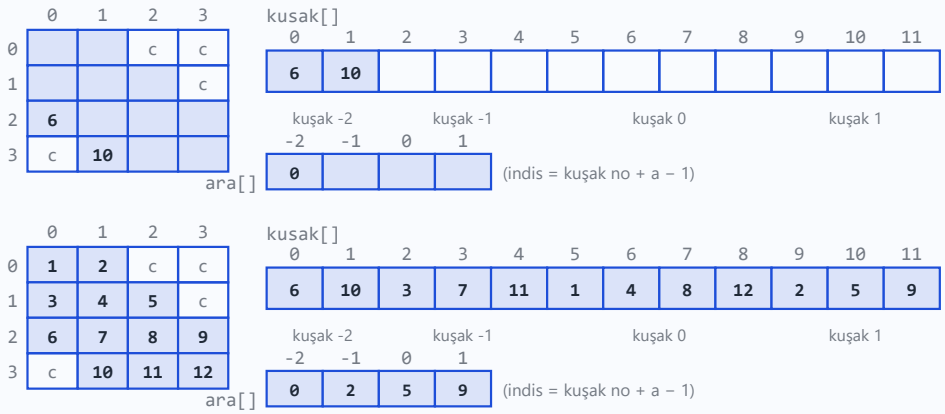
- `toplamoge`, o ana kadar yerleşmiş öğe sayısıdır; yani sıradaki kuşağın başlangıcıdır.
- Önce saklanacak öğe sayısı `MAX_SIZE` ile karşılaştırılır. Formül, kuşak uzunluklarının toplamından çıkan öğe sayısıdır.
- Dış döngü kuşak numarası `i`'yi $1 - a$'dan $b - 1$'e kadar gezer. Burada `i` bir satır indisi değil, kuşak numarasıdır.
- `ara[i + a - 1] = toplamoge` kuşağın başlangıcını kaydeder.
- İç döngü kuşağın $n - |i| - 1$ öğesini okur: `k`, 0'dan $n - |i| - 1$'e kadar gider. Her değer `kusak[ara[i + a - 1] + k]`'ye yazılır.
- Kuşak bitince `toplamoge`, kuşağın uzunluğu kadar artar.

Örnek 2.3

$n = 4$, $a = 3$, $b = 2$ için matrisin değerleri $d_{00} = 1$, $d_{01} = 2$, $d_{10} = 3$, $d_{11} = 4$, $d_{12} = 5$, $d_{20} = 6$, $d_{21} = 7$, $d_{22} = 8$, $d_{23} = 9$, $d_{31} = 10$, $d_{32} = 11$, $d_{33} = 12$ olsun. Kuşak kuşak verilen girdi `6 10 · 3 7 11 · 1 4 8 12 · 2 5 9`'dur. Denetim $4 \cdot 4 - 3 - 1 = 12 \le$ `MAX_SIZE` ile geçilir. Tablo 2.14 her kuşağın işlenişini verir.

Kuşak `i`	Uzunluk $n - \|i\|$	`ara[i + a - 1]`	Yazılan konumlar ← değer (öğe)	`toplamoge` sonra
−2 (alt)	2	`ara[0] = 0`	$0 \leftarrow 6$ (d_{20}), $1 \leftarrow 10$ (d_{31})	2
−1 (alt)	3	`ara[1] = 2`	$2 \leftarrow 3$ (d_{10}), $3 \leftarrow 7$ (d_{21}), $4 \leftarrow 11$ (d_{32})	5
0 (köşegen)	4	`ara[2] = 5`	$5 \leftarrow 1$ (d_{00}), $6 \leftarrow 4$ (d_{11}), $7 \leftarrow 8$ (d_{22}), $8 \leftarrow 12$ (d_{33})	9
1 (üst)	3	`ara[3] = 9`	$9 \leftarrow 2$ (d_{01}), $10 \leftarrow 5$ (d_{12}), $11 \leftarrow 9$ (d_{23})	12

Tablo 2.14: `KusakMatrisKur(kusak, ara, 4, 3, 2)` çağrısının izlemesi.



Şekil 2.11: Kurulumdan iki an. (a) İlk kuşak (−2) yerleşti: `kusak[0] = 6`, `kusak[1] = 10`, `ara[0] = 0`. (b) Bütün kuşaklar yerleşti: `kusak = {6, 10, 3, 7, 11, 1, 4, 8, 12, 2, 5, 9}`, `ara = {0, 2, 5, 9}`.

Sonuçta `toplamoge` 12'ye ulaşır; bu, öğe sayısı formülünün verdiği değerdir. Sürücü programın yazdırdığı `ara` ve `kusak` dizileri tabloyla aynıdır. Fonksiyon her saklanan öğe için bir `scanf` yapar ve dış döngü $a + b - 1$ kez döner. Süre *O(a + b + saklanan öğe sayısı)*'dır; kuşak sayısı öğe sayısını aşamayacağından bu *O(saklanan öğe sayısı)*'dır. Ek bellek `ara` dizisinin $a + b - 1$ hücresidir.

2.6.4 Kod izleme: KusakMatrisEris()

Erişim fonksiyonu *(i, j)* çiftini `kusak` içindeki konuma çevirir. Kuşak dışındaki öğeler için iki ayrı hata kodu döndürür; böylece üst ve alt sabit bölge birbirinden ayrılabilir.

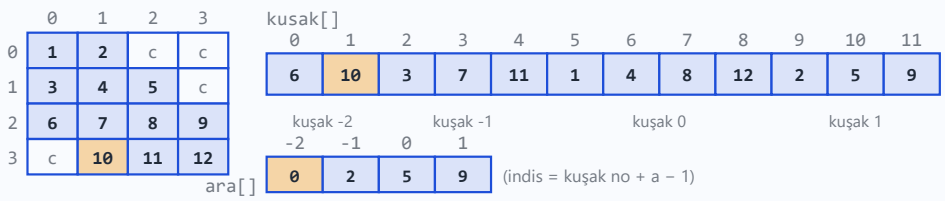
```
int KusakMatrisEris(int i, int j, int n, int a,
                    int b, int ara[]) {
  if (i < 0 || j < 0 || i >= n || j >= n) {
    printf("Indis tasmasi\n"); return -3;
  } else if (j - i >= 1) {          /* üst üçgen */
    if (j - i < b)                  /* geçerli kuşak */
      return ara[j - i + a - 1] + i;
    else return -1;                 /* üst sabit bölge */
  } else if (i - j < a)             /* köşegen / alt */
    return ara[j - i + a - 1] + j;
  else
    return -2;                      /* alt sabit bölge */
}
```

Fonksiyonun mantığı dört duruma ayrılır:

- İndislerden biri matrisin dışındaysa ileti yazılır ve −3 döner.
- $j - i \geq 1$ ise öğe köşegenin üstündedir. Kuşak numarası b'den küçükse öğe geçerli bir üst kuşaktadır ve konum `ara[j - i + a - 1] + i`'dir. Değilse öğe üst sabit bölgededir; −1 döner.
- Aksi hâlde öğe köşegende ya da altındadır. $i - j < a$ ise geçerli bir alt kuşaktadır (köşegen dâhil) ve konum `ara[j - i + a - 1] + j`'dir.
- Hiçbiri değilse öğe alt sabit bölgededir; −2 döner.

Öğe	Kuşak no $j - i$	Karar	Dönüş	Değer
d_{31}	−2	$i - j = 2 < 3$: alt kuşak	`ara[0] + j` = 0 + 1 = 1	`kusak[1]` = 10
d_{12}	1	$1 \geq 1$: üst; $1 < b = 2$: geçerli	`ara[3] + i` = 9 + 1 = 10	`kusak[10]` = 5
d_{03}	3	$3 \geq 1$: üst; $3 \geq b$: kuşak dışı	−1	*c*
d_{30}	−3	$i - j = 3 < 3$ yanlış: kuşak dışı	−2	*c*

Tablo 2.15: `KusakMatrisEris` ile dört sorgu ($n = 4$, $a = 3$, $b = 2$, `ara = {0, 2, 5, 9}`).



Şekil 2.12: d_{31} sorgusu. Öğe −2 numaralı kuşağın ikinci öğesidir; konumu `ara[0] + 1 = 1` ve değeri 10'dur.

Sürücü programda dört sorgu aynı sonuçları vermiştir. Ayrıca bütün *(i, j)* çiftleri için erişim fonksiyonu çağrılıp matris yeniden kurulmuş, kuşak dışı öğelere 0 yazılmıştır. Ortaya çıkan matris örnekteki değerlerle aynıdır. Fonksiyon her durumda sabit sayıda karşılaştırma ve aritmetik işlem yapar; girdiye bağlı bir döngüsü yoktur. Süresi *O(1)*'dir.

> **Sık yapılan hata**
>
> Üst kuşakta konumu `ara[…] + j` ile hesaplamak. Üst kuşağın ilk öğesi 0. satırdadır ama 0. sütunda değildir; *t* numaralı üst kuşak

t. sütundan başlar. Sıra sütundan değil satırdan sayılmalıdır. d_{12} için hatalı formül *9 + 2 = 11* verir ve d_{23}'ün değerini (9) döndürür.

2.6.5 ARA dizisi mi, formül mü?

`KusakMatrisEris` sabit sürede çalışır, çünkü kuşak başlangıçları önceden `ara` dizisine yazılmıştır. Bu dizi olmasaydı her erişimde önceki kuşakların uzunluklarını bir döngüyle toplamak gerekirdi ve erişim *O(a + b)*'ye çıkardı. `ara` bunu önler ama bedeli vardır: fazladan bellek harcar ve matrisin parametreleri değişirse yeniden kurulmalıdır. Bu da programı dinamik olmaktan uzaklaştırır.

Burada genel bir tasarım ilkesi görülür. Performansı gerçekten artırıyorsa, makul ölçüde fazladan bellek kullanmakta sakınca yoktur. `ara` yalnız *a + b − 1* hücredir; saklanan öğelerin yanında çok küçüktür. Yine de aynı sonucu hiç ek bellek kullanmadan almak mümkündür: kuşak başlangıcı da kapalı bir formülle hesaplanabilir.

```
/* t: kuşak no (1-a..b-1); ara[t+a-1] ile aynı */
int kusak_basi(int t, int n, int a) {
  if (t <= 0)                    /* alt kuşaklar */
    return (t + a - 1) * n
           - (a * (a - 1) + t * (1 - t)) / 2;
  return (a - 1) * n - a * (a - 1) / 2  /* t=0 başı */
         + t * n - t * (t - 1) / 2;
}
```

Formül iki durumda çıkarılır.

- **Alt kuşaklar ve köşegen (*t ≤ 0*).** *t*'den önce *1 − a, …, t − 1* numaralı *t + a − 1* kuşak vardır. Hepsi *n* öğeli olsaydı toplam *(t + a − 1) · n* olurdu. Eksikler, bu kuşakların numaralarının mutlak değerleridir: *(−t + 1) + … + (a − 1) = a(a − 1)/2 − t(t − 1)/2*. Kodda bu ifade `(a * (a - 1) + t * (1 - t)) / 2` olarak yazılmıştır; iki biçim aynıdır.
- **Üst kuşaklar (*t > 0*).** Köşegenin başlangıcı, yukarıdaki formülde *t = 0* konarak bulunur: *(a − 1) · n − a(a − 1)/2*. Ona köşegen ve *1, …, t − 1* numaralı kuşakların uzunlukları eklenir: *n + (n − 1) + … + (n − t + 1) = t · n − t(t − 1)/2*.

n = 4, a = 3 için *t = −2, −1, 0, 1* değerleri sırayla 0, 2, 5 ve 9 verir; `ara` ile aynıdır. Not için yazılan bir döngü *n ≤ 11* ve bütün *a, b ≤ n* değerleri için 4.356 kuşak başlangıcını tek tek saymıştır. Formülle hiç fark çıkmamıştır. Formülle erişim `kusak_basi(j - i, n, a) + (j > i ? i : j)` olur. Bu da *O(1)*'dir; birkaç çarpma fazladan yapılır ama dizi gerekmez. Hangisinin seçileceği, belleğin mi işlem süresinin mi daha değerli olduğuna bağlıdır.

2.6.6 && işlecinin değerlendirme sırası

Erişim fonksiyonlarındaki koşullar `i < 0 || i >= n || …` gibi birleşik ifadelerdir. Bu tür ifadeler için sık duyulan bir kanı vardır. Buna göre derleyicinin ifadeyi soldan sağa mı sağdan sola mı taradığı bilinemez. Bu yüzden aşağıdaki iki biçimden ikincisi daha güvenlidir.

```
/* 1 */ if (i < n && a[i] != 0) ...

/* 2 */ if (i < n)
          if (a[i] != 0) ...
```

Bu kanı yanlıştır. C standardı `&&` ve `||` işleçleri için soldan sağa değerlendirmeyi ve **kısa devre değerlendirmeyi** (*short-circuit evaluation*) garanti eder (C99 §6.5.13–14). `&&`'nin sol tarafı yanlışsa sonuç zaten yanlıştır ve sağ taraf hiç hesaplanmaz. `||`'de sol taraf doğruysa sağ taraf hesaplanmaz. Birinci biçimde `i < n` yanlışsa `a[i]` hiç okunmaz. İki biçim aynı derecede güvenlidir; birincisi C'de yaygın ve doğru bir kalıptır. Sürücü programda üç öğeli bir dizide `i = n = 3` iken birinci biçim denenmiş; koşul `a[3]` okunmadan yanlış çıkmıştır.

Asıl tuzaklar başka yerdedir. `&` (bit düzeyi VE) ve `|` kısa devre yapmaz; `i < n & a[i] != 0` yazılırsa iki taraf da hesaplanır ve `a[n]` okunur. Ayrıca `f(x) + g(x)` gibi sıradan bir ifadede işlenenlerin hangi sırayla hesaplanacağı standartta belirtilmemiştir. `f` ve `g` aynı küresel değişkeni değiştiriyorsa sonuç derleyiciye göre farklı çıkabilir. Değerlendirme sırası yalnız `&&`, `||`, `?:` ve virgül işlecinde garanti altındadır.

2.7 Yapılar ve birlikler

Dizinin bütün öğeleri aynı tiptedir. Gerçek dünyadaki kayıtlar ise farklı tipte parçalardan oluşur: bir öğrencinin numarası tam sayı, adı karakter dizisi, not ortalaması ondalıklı sayıdır. C bu tür kayıtları `struct` ile tanımlar. `union` ise birkaç alanın aynı belleği paylaştığı özel bir kayıt türüdür. Bu kesim iki yapıyı, bellekteki görünümlerini ve hizalamanın boyut üzerindeki etkisini ele alır.

2.7.1 struct

Tanım 2.5

`struct` **(yapı)**, farklı tipteki alanları tek bir kayıtta toplayan C tipidir. Alanlar bellekte bildirim sırasıyla yerleşir. Bir alana `değişken.alan` yazımıyla, bir yapı işaretçisi üzerinden ise `işaretçi->alan` yazımıyla erişilir.

```c
typedef struct {
  char ad[10];
  int yas;
  float maas;
} insan;

insan k1, k2;
strcpy(k1.ad, "Ali");
k1.yas = 21;
k1.maas = 3500.0f;
k2 = k1;          /* alan alan kopyalanır */
/* if (k1 == k2) -> DERLENMEZ */
```

Kod satır satır şöyle çalışır:

- `typedef struct { … } insan;` adsız bir yapı tanımlar ve ona `insan` adını verir. `typedef`, var olan bir tipe yeni bir ad veren bildirimdir. Bu sayede değişken bildirirken her seferinde `struct` yazmak gerekmez: `insan k1, k2;` yeterlidir.
- `k1.ad` bir `char` dizisidir. Dizilere atama yapılamadığı için ad `strcpy` ile kopyalanır. `k1.yas` ve `k1.maas` sıradan değişkenler gibi atanır. `3500.0f` sonundaki `f`, sabitin `float` olduğunu belirtir.
- `k2 = k1;` bütün yapıyı alan alan kopyalar. İlginç olan şudur: yapının içindeki `ad` dizisi de kopyalanır. Tek başına bir diziye atama yapılamaz, ama dizi bir yapının alanıysa yapı ataması onu da taşır.
- `k1 == k2` derlenmez. C yapılar için eşitlik işleci tanımlamaz. Nedeni hizalamadır: alanlar arasında anlamsız **dolgu baytları** (*padding*) bulunabilir ve bayt bayt karşılaştırma yanlış sonuç verebilir. İki yapıyı karşılaştırmak için alan alan karşılaştıran bir fonksiyon yazılır; `char` dizisi alanları `strcmp` ile karşılaştırılır.

Yapılar fonksiyonlara çoğu zaman işaretçiyle geçirilir. Yapı değerle geçirilirse bütün alanları kopyalanır; büyük yapılarda bu pahalıdır. İşaretçiyle geçirmek yalnız bir adres kopyalar. `insan *p = &k2;` tanımından sonra `p->yas` ile `(*p).yas` aynı alanı verir. Parantez gereklidir, çünkü `.` işlecinin önceliği `*`'dan yüksektir: `*p.yas`, "`p.yas`'ın gösterdiği yer" anlamına gelir ve derlenmez. **Ok işleci** (*arrow operator*) bu yazımı kısaltmak için vardır.

Sürücü programda `sizeof(insan)` 20 çıkmıştır. Alanların toplamı *10 + 4 + 4 = 18* bayttır. Aradaki 2 bayt hizalamadan gelir: `yas` alanı 4'ün katı olan bir adreste başlamalıdır ve `ad` 10. baytta bittiği için iki dolgu baytı eklenir.

2.7.2 İç içe ve kendine başvuran yapılar

Bir yapının alanı başka bir yapı olabilir. Ayrıca bir yapı, kendi türünden bir nesneyi gösteren bir işaretçi alanı taşıyabilir.

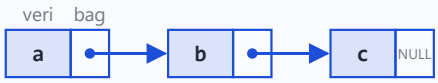
```c
typedef struct {
  int gun, ay, yil;
} tarih;

typedef struct {
  char ad[10];
  tarih dogum;          /* iç içe yapı */
} kisi;
/* k.dogum.yil = 2004; */

typedef struct liste {
  char veri;
  struct liste *bag;    /* kendine başvuru */
} liste;
```

`kisi` yapısının `dogum` alanı bir `tarih` yapısıdır. İç içe alanlara nokta işleci art arda uygulanarak erişilir: `k.dogum.yil = 2004;`. Sürücü programda `sizeof(kisi)` 24 çıkmıştır: 10 baytlık ad, 2 dolgu baytı ve 12 baytlık tarih.

`liste` yapısı farklıdır. `bag` alanının tipi `struct liste *`, yani yapının kendi türüne işaretçidir. Bu tür yapılara **kendine başvuran yapı** (*self-referential struct*) denir. Tanımda yapıya bir etiket adı (`struct liste`) verilmesi zorunludur. `typedef` ile verilen `liste` adı bildirim bitene kadar henüz tanımlı değildir; alanın tipi ancak etiketle yazılabilir.



Şekil 2.13: Kendine başvuran yapılardan kurulmuş üç düğümlü bir bağlı liste. Her düğümün `veri` alanında bir karakter, `bag` alanında sonraki düğümün adresi durur; son düğümün bağı `NULL`'dır.

Şekil 2.13 böyle üç yapının birbirine bağlanmasını gösterir. Her düğüm sonrakinin adresini tutar; zincirin sonu `NULL` ile belirtilir. Sürücü programda üç düğüm `a`, `b`, `c` değerleriyle kurulmuş ve `for (q = &a1; q != NULL; q = q->bag)` döngüsüyle sırayla yazdırılmıştır. Bu yapı, bağlı listelerin, ağaçların ve çizgelerin temelidir. Dizinin aksine düğümler bellekte ardışık olmak zorunda değildir; araya ekleme yalnız birkaç bağın değiştirilmesiyle yapılır.

Bir yapı kendi türünden bir *alan* içeremez. `struct liste { char veri; struct liste sonraki; };` gibi bir tanımda yapının boyu kendi boyunu içermek zorunda kalırdı; bu sonsuz bir boyut demektir ve derlenmez. Kendi türüne *işaretçi* ise sorun yaratmaz, çünkü işaretçinin boyu gösterdiği tipin boyundan bağımsızdır (64 bitlik sistemde 8 bayt). Sürücüde `sizeof(liste)` 16 çıkmıştır: 1 baytlık karakter, 7 dolgu baytı ve 8 baytlık işaretçi.

2.7.3 union ve hizalama

> **Tanım 2.6**
>
> `union` **(birlik)**, bütün alanları aynı bellek bölgesini paylaşan C tipidir. Birliğin boyu en büyük alanı kadardır (hizalama için biraz büyüyebilir). Aynı anda yalnız bir alan geçerli bir değer taşır.

Birlik, bir kaydın türüne göre farklı bilgiler taşıması gerektiğinde bellek kazandırır. Ders kitabındaki örnekte bir kişinin cinsiyetine göre iki farklı bilgi tutulur. İki alan hiçbir zaman aynı anda gerekmediği için aynı belleği paylaşabilir.

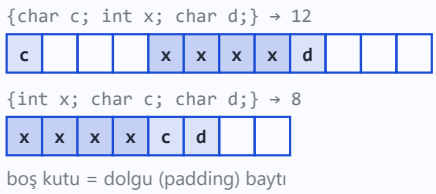
```c
typedef struct {
  enum { KADIN, ERKEK } tur;   /* etiket */
  union {
    int cocuk_sayisi;          /* KADIN ise */
    int sakal;                 /* ERKEK ise */
  } u;
} cinsiyet;

cinsiyet c;
c.tur = ERKEK;
c.u.sakal = 0;   /* yalnız etiketle uyumlu alanı oku */
```

`tur` alanının tipi bir **sayım tipidir** (*enumeration*, `enum`): `KADIN` ve `ERKEK`, 0 ve 1 değerlerini alan adlandırılmış sabitlerdir. Bu alan, birliğin hangi alanının geçerli olduğunu gösteren bir *etikettir*. Böyle bir yapıya **etiketli union** (*tagged union*) denir. Kod, etiketi `ERKEK` yapıp ardından yalnız onunla uyumlu `sakal` alanını kullanır.

Etiketin doğru tutulması tamamen programcının işidir. C, birliğin hangi alanına en son yazıldığını izlemez. `c.u.sakal = 0;` yazıldıktan sonra `c.u.cocuk_sayisi` okunursa aynı dört bayt bu kez başka bir anlamla yorumlanır. Burada iki alan da `int` olduğu için sonuç 0 çıkar, ama alanlar farklı tipte olsaydı anlamsız bir değer okunurdu. Doğru kalıp, birliğe her erişimden önce etikete bakmaktır: `if (c.tur == ERKEK) … c.u.sakal …`. Sürücü programda `sizeof c.u` 4, `sizeof c` ise 8 bayt çıkmıştır: 4 baytlık etiket ve 4 baytlık birlik. Ayrı iki alan kullanılsaydı yapı 12 bayt olurdu.

Yapı boyunu belirleyen asıl etken **hizalamadır** (*alignment*). İşlemciler çoğu tipi kendi boyunun katı olan adreslerden en hızlı biçimde okur; bazı işlemciler hizasız okumaya hiç izin vermez. Bu yüzden derleyici her alanı uygun bir adres katına yerleştirir ve gerekirse araya dolgu baytları ekler. Yapının toplam boyu da en büyük hizalama gereğinin katına yuvarlanır; böylece yapı dizilerinde her öğe doğru hizada başlar.



Şekil 2.14: Alan sırasının yapı boyuna etkisi (4 baytlık `int`, tipik 32/64 bitlik derleyici). Üstte `{char c; int x; char d;}` 12 bayt, altta `{int x; char c; char d;}` 8 bayt tutar. Boş kutular dolgu baytlarıdır.

Şekil 2.14 aynı üç alanın iki farklı sırasını karşılaştırır. Birinci sırada `c` 0. bayta yerleşir; `x` 4'ün katında başlamak zorunda olduğu için 1, 2, 3. baytlar dolgu olur ve `x` 4–7. baytları kaplar. `d` 8. baytta durur. Yapının boyu 4'ün katı olmalıdır, bu yüzden sona üç dolgu baytı eklenir: toplam 12. İkinci sırada `x` 0–3, `c` 4, `d` 5. baytları kaplar; sona iki dolgu baytı eklenir ve toplam 8 olur. Aynı veri üçte bir daha az bellekle tutulur. Sürücü programda bu iki yapının boyları 12 ve 8, `offsetof` ile ölçülen `x` konumu ise birincisinde 4 çıkmıştır.

Pratik kural, alanları büyükten küçüğe doğru sıralamaktır. Hizalama kuralları derleyiciye ve işlemciye bağlıdır; burada verilen değerler yaygın 32 ve 64 bitlik sistemler için tipiktir. Dosyaya ya da ağa ham bayt olarak yazılan yapılarda dolgu baytları taşınabilirlik sorunu çıkarır. Bu yüzden böyle durumlarda alanlar tek tek yazılır.

2.7.4 Kod izleme: sinif_ort()

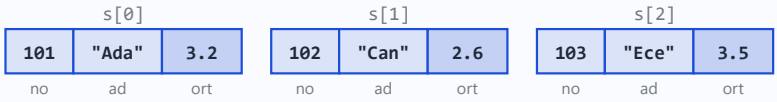
Yapılar dizilerde toplanabilir. Aşağıdaki fonksiyon bir sınıfın öğrenci kayıtlarını içeren diziyi alır ve not ortalamalarının ortalamasını döndürür. Bir veri yapısının bütün öğelerine belirli bir sırayla birer kez uğramaya **gezinme** (*traversal*) denir.

```c
typedef struct {
  int no;
  char ad[16];
  double ort;
} Ogrenci;

double sinif_ort(const Ogrenci s[], int n) {
  double top = 0.0;
  int i;
  for (i = 0; i < n; i++)
    top += s[i].ort;              /* . ile alan */
  return n > 0 ? top / n : 0.0;    /* boş sınıf */
}
```

Kod satır satır şöyle çalışır:

- `const Ogrenci s[]` parametresi bir işaretçidir; dizi kopyalanmaz. `const`, fonksiyonun kayıtları değiştirmeyeceğini belgeler ve yanlışlıkla bir alana yazılırsa derleyicinin hata vermesini sağlar.
- `s[i].ort` önce *i*. kaydı bulur ($\alpha + i \cdot$ `sizeof(Ogrenci)` adresi), sonra onun `ort` alanını okur. Yapı dizisinde de adres formülü aynen geçerlidir; yalnız öğe boyu yapının boyudur.
- `n > 0 ? top / n : 0.0`, boş sınıfta sıfıra bölmeyi önler. Kayan noktalı sayılarda sıfıra bölme programı durdurmaz ama sonsuz ya da NaN üretir; bu değer sessizce sonraki hesaplara yayılır.



Şekil 2.15: Üç öğrenci kaydından oluşan yapı dizisi. Kayıtlar bellekte art arda durur; her kayıt `sizeof(Ogrenci)` bayttır.

`i`	`i < n`	`s[i].ort`	`top` **sonra**
0	0 < 3, doğru	3,2	3,2
1	1 < 3, doğru	2,6	5,8
2	2 < 3, doğru	3,5	9,3
3	3 < 3, yanlış	—	döngü biter; *n > 0*, dönüş *9,3 / 3 = 3,10*

Tablo 2.16: Şekil 2.15'teki üç kayıtla `sinif_ort(s, 3)` çağrısının izlemesi.

Fonksiyon her kayda bir kez uğrar; süresi $\Theta(n)$, ek belleği $O(1)$'dir. Sürücü programda sonuç `%.2f` biçimiyle 3.10 olarak yazdırılmıştır. Ara toplamların tam değerleri ise 3.2000000000000002, 5.8000000000000007 ve 9.3000000000000007'dir. 3,2 gibi ondalık sayılar ikili tabanda tam gösterilemez ve küçük yuvarlama hataları birikir. Bu yüzden kayan noktalı sonuçlar `==` ile karşılaştırılmaz; ya belirli bir basamakla yazdırılır ya da küçük bir tolerans içinde karşılaştırılır.

`Ogrenci` yapısı da hizalamanın etkisini gösterir. 4 baytlık `no` ve 16 baytlık `ad` 20 bayt eder. `double` 8'in katında başlamalıdır; bu yüzden 4 dolgu baytı eklenir ve `ort` 24. baytta başlar. Toplam boy 32 bayttır. Sürücü programda `offsetof(Ogrenci, ort)` 24, `sizeof(Ogrenci)` 32 çıkmıştır.

2.8 Dizgiler

Metin, programların en çok işlediği veri türlerinden biridir. Pek çok dil metin için ayrı bir tip sunar. C'de ise metin, özel bir kurala uyan sıradan bir `char` dizisidir. Bu sadelik hız kazandırır ama dikkatsiz kullanıldığında en yaygın güvenlik açıklarının da kaynağı olur.

2.8.1 '\0' ile biten char dizisi

Tanım 2.7

C'de **dizgi** (*string*), değeri sıfır olan `'\0'` karakteriyle biten bir `char` dizisidir. Bu karaktere **dizgi sonu karakteri** (*null character*) denir. Dizginin uzunluğu, ilk `'\0'` 'dan önceki karakter sayısıdır.

```c
char s[] = "dog";      /* 4 bayt: d o g \0 */
char t[10];
char *p = "dog";       /* salt okunur sabit */

strlen(s)   /* 3 : \0'a kadar */
sizeof s    /* 4 : dizinin boyutu */

strcpy(t, s);                /* t yeterince büyük mü? */
snprintf(t, sizeof t, "%s!", s);   /* güvenli */
if (strcmp(s, "dog") == 0) ...     /* == değil! */
```



Şekil 2.16: `char s[] = "dog";` bildiriminin bellekteki görünümü: üç karakter ve sonlandırıcı `'\0'`, toplam 4 bayt.

Kod satır satır şöyle çalışır:

- `char s[] = "dog";` dört baytlık bir dizi ayırır. Derleyici dizgiyi kopyalar ve sona `'\0'` ekler; boyutu bu yüzden 3 değil 4'tür (Şekil 2.16).
- `char t[10];` on karakterlik boş bir alandır; içeriği henüz belirsizdir.
- `char *p = "dog";` bir dizi ayırmaz. `p`, programın salt okunur bölgesindeki bir dizgi sabitini gösterir. `p[0] = 'D';` yazmak tanımsız davranıştır ve çoğu sistemde programı çökertir. Bu yüzden böyle işaretçiler `const char *p` olarak bildirilmelidir; o zaman derleyici yazma girişimini yakalar.
- `strlen(s)` 3, `sizeof s` 4 verir. `strlen` karakterleri `'\0'` 'a kadar sayar; `sizeof` ise dizinin kapladığı bellektir.
- `strcpy(t, s)`, `s` 'yi sonlandırıcısıyla birlikte `t` 'ye kopyalar. `strcpy` hedefin boyunu bilmez; `t` yeterince büyük değilse sonundan taşar. Burada 10 bayt 4 baytlık dizgi için yeterlidir.
- `snprintf(t, sizeof t, "%s!", s)` hedefin boyunu ikinci parametre olarak alır ve en çok `sizeof t - 1` karakter yazıp sonuna her zaman `'\0'` koyar. Taşma imkânsızdır. Sürücüde `t` 'nin yeni değeri `dog!` olmuştur.
- `strcmp(s, "dog") == 0` iki dizginin içeriğini karakter karakter karşılaştırır. `s == "dog"` yazılsaydı içerikler değil adresler karşılaştırılırdı ve sonuç büyük olasılıkla yanlış çıkardı.

`<string.h>` başlığı dizgiler için bir dizi fonksiyon sunar. Tablo 2.17 en çok kullanılanları özetler. Bu fonksiyonların hepsi dizginin sonunu `'\0'` 'dan anlar. Sonlandırıcısı olmayan bir dizi bu fonksiyonlara verilirse, belleğin ilerisinde rastlantısal bir sıfır bulunana kadar okumaya devam ederler.

Fonksiyon	İş	Süre (n: dizgi uzunluğu)
`strlen`	uzunluk (`'\0'` hariç)	$O(n)$
`strcpy` / `strncpy`	kopyala	$O(n)$
`strcat`	sona ekle	$O(n + m)$; önce hedefin sonu aranır
`strcmp`	sözlük sırasıyla karşılaştır; eşitse 0, ilki küçükse negatif, büyükse pozitif	$O(n)$
`strchr` / `strstr`	karakter / alt dizgi ara; bulamazsa `NULL`	$O(n)$ / basit gerçeklemede $O(n \cdot m)$

Tablo 2.17: Sık kullanılan `<string.h>` fonksiyonları.

`strncpy`'nin bir tuzağı vardır. Kaynak, verilen sınırdan uzunsa `strncpy` hedefe `'\0'` koymaz; sonuç dizgi değildir. Bu yüzden güvenli kopya için `snprintf` ya da sınır bir eksik tutularak sonradan elle `'\0'` yazılan `strncpy` tercih edilir.

> **Sık yapılan hata**
>
> Dizgi için sonlandırıcıya yer ayırmamak. `char ad[3] = "Ali";` C'de derlenir (C++'ta derlenmez), ama dizi yalnız üç karakteri tutar ve `'\0'` dışarıda kalır. `strlen(ad)` ya da `printf("%s", ad)` dizinin sonundan okumaya devam eder. *n* karakterlik bir dizgi için her zaman *n + 1* bayt gerekir. Dinamik ayırmada da aynısı geçerlidir: `malloc(strlen(s) + 1)`.

2.8.2 Kod izleme: uzunluk()

Aşağıdaki fonksiyon `strlen`'in yaptığı işi açıkça yazar. Bir işaretçiyi `'\0'`'a gelene kadar ilerletir ve başlangıçtan ne kadar uzaklaştığını döndürür.

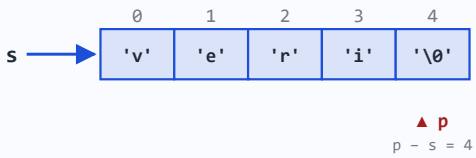
```
size_t uzunluk(const char *s) {
  const char *p = s;
  while (*p != '\0')     /* sonlandırıcıya dek */
    p++;
  return (size_t)(p - s);  /* işaretçi farkı */
}

/* uzunluk("veri") */
```

`p` başlangıçta `s` ile aynı yeri gösterir. Döngü her turda `*p`'yi sonlandırıcıyla karşılaştırır; farklıysa `p`'yi bir karakter (1 bayt) ilerletir. Döngü bittiğinde `p` sonlandırıcıyı gösterir. İki işaretçinin farkı `p - s`, aralarındaki öğe sayısıdır; bu da dizginin uzunluğudur. Fark `ptrdiff_t` tipindedir ve `size_t`'ye dönüştürülerek döndürülür.

Tur	`p - s`	`*p`	`*p != '\0'`	İşlem
1	0	`'v'`	doğru	`p++`
2	1	`'e'`	doğru	`p++`
3	2	`'r'`	doğru	`p++`
4	3	`'i'`	doğru	`p++`
5	4	`'\0'`	yanlış	döngü biter; dönüş 4

Tablo 2.18: `uzunluk("veri")` çağrısının izlemesi.



Şekil 2.17: Döngü bittiğinde `p` sonlandırıcıyı gösterir; *p − s = 4*.

Uzunluğu bulmak için dizgi baştan sona taranır: süre $\Theta(n)$'dir. Dizi gibi uzunluğu ayrıca tutulan yapılardan farklı olarak C dizgisinin uzunluğu her seferinde yeniden hesaplanır. Bu, sık rastlanan bir verimsizliğe yol açar:

```
for (i = 0; i < strlen(s); i++)   /* her turda strlen! */
    ...
```

Döngü koşulu her turda yeniden hesaplanır ve her hesap *n* karakteri tarar. *n* tur için toplam $n \cdot n$ karakter okunur; döngü $O(n^2)$ olur. Bir milyon karakterlik bir metinde bu, bir milyon yerine bir trilyon okuma demektir. Doğrusu uzunluğu döngüden önce bir kez hesaplamaktır: `size_t len = strlen(s); for (i = 0; i < len; i++)`. Döngü gövdesi dizgiyi değiştirmiyorsa iyi derleyiciler bu dönüşümü kendileri yapabilir, ama buna güvenmemek gerekir.

2.9 Matris işlemleri

İki boyutlu diziler en çok matris hesaplarında kullanılır. Bu kesim matris toplamayı ve çarpmayı inceler; özellikle çarpmanın adım sayısını tam olarak çıkarır. Örneklerde `N` derleme sırasında tanımlı bir sabittir ve matrisler `int a[][N]` biçiminde geçirilir. İkinci boyutun yazılması zorunludur, çünkü adres hesabı satır uzunluğuna ihtiyaç duyar.

2.9.1 Matris toplama

Aynı boyutlu iki matrisin toplamında her hücre, iki matrisin aynı konumdaki hücrelerinin toplamıdır. $m \times N$ matrisler için kod iki iç içe döngüden oluşur.

```
void topla(int a[][N], int b[][N], int c[][N], int m) {
    int i, j;
    for (i = 0; i < m; i++)
        for (j = 0; j < N; j++)
            c[i][j] = a[i][j] + b[i][j];
}
```

İç döngü gövdesi her hücre için bir kez, toplam $m \cdot N$ kez çalışır. Döngü denetimleri de bu sayıyla orantılıdır. Süre $\Theta(mN)$'dir; $n \times n$ kare matrislerde $\Theta(n^2)$. Sonuç matrisi dışında ek bellek gerekmez. Her hücre bir kez okunup bir kez yazıldığı için bundan daha hızlı bir toplama algoritması yoktur: girdinin her hücresine en az bir kez bakmak gerekir. Sürücüde 2×3 iki matrisin toplamı denenmiş ve beklenen sonuç alınmıştır.

2.9.2 Kod izleme: mult()

A ve B iki $n \times n$ matris olsun. Çarpım $C = AB$'nin her hücresi, A'nın bir satırı ile B'nin bir sütununun iç çarpımıdır:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}$$

```
#define N 2
void mult(int a[][N], int b[][N], int c[][N]) {
  int i, j, k;
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++) {
      c[i][j] = 0;
      for (k = 0; k < N; k++)
        c[i][j] += a[i][k] * b[k][j];
    }
}
```

Dış iki döngü sonuç matrisinin hücrelerini gezer. Her hücre önce sıfırlanır, sonra en içteki döngü n çarpımı toplar. $A = [[1, 2], [3, 4]]$ ve $B = [[5, 6], [7, 8]]$ için izleme Tablo 2.19'da verilmiştir.

`i`	`j`	`k`	İşlem	`c[i][j]` sonra
0	0	0	$0 + a[0][0] \cdot b[0][0] = 1 \cdot 5$	5
0	0	1	$5 + a[0][1] \cdot b[1][0] = 5 + 2 \cdot 7$	19
0	1	0	$0 + a[0][0] \cdot b[0][1] = 1 \cdot 6$	6
0	1	1	$6 + a[0][1] \cdot b[1][1] = 6 + 2 \cdot 8$	22
1	0	0	$0 + a[1][0] \cdot b[0][0] = 3 \cdot 5$	15
1	0	1	$15 + a[1][1] \cdot b[1][0] = 15 + 4 \cdot 7$	43
1	1	0	$0 + a[1][0] \cdot b[0][1] = 3 \cdot 6$	18
1	1	1	$18 + a[1][1] \cdot b[1][1] = 18 + 4 \cdot 8$	50

Tablo 2.19: `mult` fonksiyonunun $N = 2$ için izlemesi.



Şekil 2.18: *2 × 2* matrislerin çarpımı: *C = [[19, 22], [43, 50]]*. En içteki satır $N^3 = 8$ kez çalıştı.

Sürücü program sonucu `[[19, 22], [43, 50]]` olarak yazdırmıştır. En içteki atama sekiz kez, yani N^3 kez çalışmıştır. n büyüdükçe bu sayı küp olarak büyür; matris boyu iki katına çıkınca iş sekiz katına çıkar.

2.9.3 Adım sayısı, dikdörtgen matrisler ve transpoz

Çarpmanın tam adım sayısı her satırın kaç kez çalıştığı sayılarak bulunur. Bir `for` döngüsünün koşulu, gövdenin çalışma sayısından bir fazla denetlenir: son denetim yanlış çıkar ve döngü biter. Tablo 2.20 satır satır sayımı verir.

Satır	Çalışma sayısı
`for (i …)` koşulu	$n + 1$
`for (j …)` koşulu	$n(n + 1)$
`c[i][j] = 0`	n^2
`for (k …)` koşulu	$n^2(n + 1)$
`c[i][j] += …`	n^3
Toplam	$2n^3 + 3n^2 + 2n + 1$

Tablo 2.20: `mult`'un $n \times n$ matrislerde adım sayısı.

Toplam $(n + 1) + (n^2 + n) + n^2 + (n^3 + n^2) + n^3$ terimlerinin toplanmasıyla bulunur. Baskın terim $2n^3$'tür; süre $\Theta(n^3)$'tür. Yapılan çarpma sayısı tam n^3'tür. Not için yazılan sayaçlı bir sürümde adımlar $n = 1$'den 9'a kadar sayılmış ve formülle aynı çıkmıştır (örneğin $n = 2$'de 33, $n = 3$'te 88).

Matrisler kare olmak zorunda değildir. $m \times n$ bir matris ile $n \times p$ bir matrisin çarpımı $m \times p$ boyutludur; yani $m \cdot p$ hücresi vardır. Her hücre n çarpımın toplamıdır. Toplam çarpma sayısı ve süre $\Theta(m \cdot n \cdot p)$'dir. Bu sonuç, art arda çarpılan matrislerde çarpma sırasının neden önemli olduğunu da açıklar. $(AB)C$ ile $A(BC)$ aynı sonucu verir ama çok farklı sayıda çarpma gerektirebilir.

Bir matrisin **transpozu** (*transpose*), satırlarıyla sütunlarının yer değiştirmiş hâlidir: a_{ij} ile a_{ji} takas edilir. Kare bir matriste bu iş ek dizi kullanmadan, **yerinde** (*in place*) yapılabilir.

```
void transpoz(int a[][N], int n) {
    int i, j, t;
    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++) { /* yalnız üst üçgen */
            t = a[i][j];
            a[i][j] = a[j][i];
            a[j][i] = t;
        }
}
```

İç döngü `j = i + 1`'den başlar; yalnız köşegenin üstündeki hücreler gezilir. Her takas, bir üst üçgen hücresini karşılığındaki alt üçgen hücresiyle değiştirir. Döngü `j = 0`'dan başlasaydı her çift iki kez takas edilirdi: ilk takas transpozu alır, ikincisi geri çevirir ve matris hiç değişmemiş olurdu. Köşegen hücreleri kendi yerlerinde kalır. Takas sayısı $(n - 1) + (n - 2) + \ldots + 0 = n(n - 1)/2$'dir; süre $\Theta(n^2)$, ek bellek tek bir geçici değişkendir. Sürücüde 3×3 bir matris üç takasla transpoz edilmiştir. Kare olmayan bir matrisin yerinde transpozu çok daha zordur, çünkü satır uzunluğu değişir ve öğeler dizide çevrimler hâlinde yer değiştirir.

2.10 Karmaşıklık özeti

Tablo 2.21 bölümdeki işlemlerin süre ve ek bellek maliyetlerini bir arada verir. Süreler en kötü durum içindir; amortize olanlar ayrıca belirtilmiştir.

İşlem	Zaman	Ek bellek
A[i] erişimi; *n* boyutlu adres (*n − 1* çarpma)	*O(1)*	*O(1)*
Dinamik diziye sona ekleme (katlama)	amortize *O(1)*, en kötü *O(n)*	kapasite $\leq 2 \cdot$ boyut
Araya ekleme / silme	*O(n)*	*O(1)*
Alt üçgen / kuşak matris erişimi	*O(1)*	$n(n+1)/2$ / kuşak öğeleri
`KusakMatrisKur` , `AltUcgenOku`	*O*(saklanan öğe)	ARA: $a + b - 1$
`strlen` / `uzunluk`	*O(n)*	*O(1)*
Matris toplama ($m \times n$)	*O(mn)*	*O(1)*
Matris çarpımı ($n \times n$)	$O(n^3)$	*O(1)*

Tablo 2.21: Bölümdeki işlemlerin karmaşıklığı.

Tablonun ortak bir dersi vardır: bellekteki yerleşim bilindiğinde erişim bir formüle iner ve sabit sürede yapılır. Tek boyutlu dizi, çok boyutlu dizi, üçgen ve kuşak matrislerin hepsinde aynı soru sorulmuştur: “aranan öğeden önce kaç öğe var?” Maliyeti artıran işlemler ise öğelerin yerini değiştirenlerdir: araya ekleme, büyütme ve tarama.

2.11 Sık yapılan hatalar

Bölümdeki hataların çoğu C'nin bilinçli bir tercihinden doğar: dil indisleri, bellek ayırmanın sonucunu ve dizgi sonunu denetlemez. Aşağıdaki listede her hata, doğru kalıbıyla birlikte verilmiştir.

1. **Dizi sınırı dışına yazmak.** `a[n]` 'ye erişmek tanımsız davranıştır ve C bunu denetlemez. *n* öğeli bir dizinin son indisi *n − 1*'dir; döngü koşulu `i < n` yazılır.
2. `a = realloc(a, …)` **yazmak.** Başarısızlıkta eski bloğun adresi kaybolur ve blok sızar. Sonuç önce geçici bir değişkene alınır, `NULL` değilse asıl işaretçiye atanır.
3. `sizeof(int *)` **ile** `sizeof(int)` **'i karıştırmak; parametrede** `sizeof a` **kullanmak.** İlki yanlış boyda bellek ayırır. İkincisinde parametre bir işaretçi olduğundan dizinin değil işaretçinin boyu çıkar. `malloc(n * sizeof *a)` kalıbı ve uzunluğun ayrı parametreyle geçirilmesi bu iki hatayı önler.
4. **İki boyutlu dinamik dizide satırları** `free` **etmeden yalnız işaretçi dizisini bırakmak.** Bütün satırlar sızar. Önce her `x[i]` , sonra `x` serbest bırakılır.
5. **Satır ve sütun öncelikli formülleri karıştırmak; öğe boyunu (*s*) unutmak.** Satır öncelikli formülde satır uzunluğu (u_1) kullanılır; bayt adresi için öğe sayısı *s* ile çarpılır.
6. **Dizgiler için** `==` **kullanmak ya da** `'\0'` **için yer ayırmamak.** `==` adresleri karşılaştırır; içerik için `strcmp` kullanılır. *n* karakterlik bir dizgi *n + 1* bayt ister.
7. **Hizalamayı hesaba katmamak.** Bir yapının boyu alanlarının toplamından büyük olabilir. Boy her zaman `sizeof` ile alınır; elle toplanmaz.

2.12 Özet

- **Dizi** soyut düzeyde ⟨indis, değer⟩ çiftlerinden oluşan bir **eşlemedir**; işlemleri `Create` , `Retrieve` ve `Store` 'dur. C gerçeklemesinde öğeler ardışık durur ve *A[i]*'nin adresi $\alpha + i \cdot s$ ile *O(1)*'de hesaplanır. Araya ekleme ve silme kaydırma gerektirdiği için *O(n)*'dir.
- C indisleri denetlemez; `a[n]` 'ye erişim **tanımsız davranıştır**. Dizi adı çoğu ifadede ilk öğenin adresine dönüşür: `list[i] ≡ *(list + i)` . **İşaretçi aritmetiği** öğe sayar, bayt değil. Dizi adına atama yapılamaz; fonksiyona geçirilen dizi işaretçiye dönüşür ve uzunluğu ayrıca gönderilir.
- `malloc` heap bellekten blok ayırır; dönüşü `NULL` için denetlenir, iş bitince `free` edilir ve işaretçiye `NULL` atanır. `sizeof *a` kalıbı tip değişse de doğru boyu verir.

- **Dinamik dizi** dolunca kapasitesini **ikiye katlar**. n eklemede toplam kopya $2n$'den azdır; ekleme başına **amortize** maliyet $O(1)$'dir. Sabit artışla büyütmek $O(n^2)$ kopya demektir. `realloc`'un sonucu önce geçici bir değişkene alınır.
- İki boyutlu dinamik dizi bir **işaretçi dizisi** ve ayrı satır bloklarından oluşur (`int **`). Satırlar ardışık olmak zorunda değildir; bırakırken önce satırlar, sonra işaretçi dizisi `free` edilir.
- C **satır öncelikli** düzeni kullanır: $\&x[i][j] = \alpha + (i \cdot u_1 + j) \cdot s$. Sütun öncelikli düzende $\alpha + (j \cdot u_0 + i) \cdot s$'dir. n boyutlu dizide adres $\alpha + \Sigma\, i_j\, a_j$'dir; **Horner yöntemi** bunu $n-1$ çarpmayla hesaplar. En dıştaki boyut adres hesabına girmez.
- **Alt üçgen matris** $n(n+1)/2$ öğeyle saklanır; a_{ij}'nin konumu $i(i+1)/2 + j$'dir. Erişim fonksiyonu geçerli konumlarla kesişmeyen hata kodları ($-1, -2$) döndürür.
- **Kuşak matriste** aynı kuşaktaki öğelerin $j - i$ farkı sabittir. Öğe sayısı $n(a+b-1) - a(a-1)/2 - b(b-1)/2$'dir. Kuşak başlangıçları ARA dizisinde tutulur ya da kapalı bir formülle hesaplanır; her iki yolla erişim $O(1)$'dir.
- `&&` ve `||` soldan sağa ve **kısa devreyle** değerlendirilir; `i < n && a[i] != 0` güvenlidir. `&` ve `|` kısa devre yapmaz.
- `struct` farklı tipteki alanları toplar; atanabilir ama `==` ile karşılaştırılamaz. **Kendine başvuran yapı** bağlı yapıların temelidir. `union`'ın alanları aynı belleği paylaşır; geçerli alan bir etiketle izlenir. **Hizalama** dolgu baytları ekler; alanlar büyükten küçüğe dizilerek yapı küçültülebilir.
- C **dizgisi** `'\0'` ile biten bir `char` dizisidir; n karakter $n+1$ bayt ister. `strlen` $O(n)$'dir; döngü koşulunda çağrılırsa döngü $O(n^2)$ olur. İçerik `strcmp` ile karşılaştırılır; taşmaya karşı `snprintf` kullanılır.
- Matris toplama $\Theta(mn)$, kare matris çarpımı $2n^3 + 3n^2 + 2n + 1$ adımla $\Theta(n^3)$, dikdörtgen matris çarpımı $\Theta(mnp)$'dir. Kare matrisin yerinde transpozu $n(n-1)/2$ takasla yapılır.

2.13 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

soyut veri tipi *abstract data type (ADT)* · gerçekleme *implementation* · dizi *array* · eşleme *mapping* · indis *index* · işaretçi *pointer* · işaretçi aritmetiği *pointer arithmetic* · işaretçiye işaretçi *pointer to pointer* · boş işaretçi *null pointer (NULL)* · sarkan işaretçi *dangling pointer* · sizeof işleci *sizeof* · dinamik bellek ayırma *dynamic memory allocation* · heap bellek *heap memory* · malloc / calloc / realloc / free · bellek sızıntısı *memory leak* · dinamik dizi *dynamic array* · kapasite *capacity* · ikiye katlama *doubling* · amortize analiz *amortized analysis* · çok boyutlu dizi *multidimensional array* · satır öncelikli düzen *row-major order* · sütun öncelikli düzen *column-major order* · Horner yöntemi *Horner's method* · alt üçgen matris *lower triangular matrix* · kuşak matris *band matrix* · kısa devre değerlendirme *short-circuit evaluation* · struct (yapı) *struct* · typedef (tip adı) *typedef* · ok işleci *arrow operator (->)* · kendine başvuran yapı *self-referential struct* · union (birlik) *union (C)* · etiketli union *tagged union* · sayım tipi *enumeration (enum)* · hizalama *alignment* · gezinme *traversal* · dizgi *string* · dizgi sonu karakteri *null character ('\0')* · transpoz *transpose* · yerinde *in place*

2.14 Kendini Dene

1. `int a[6];` dizisi 1000 adresinden başlıyor ve `sizeof(int) = 4`. `&a[4]` ve `a + 4` hangi adresi verir? `*(a + 4)` hangi öğedir?

   Cevap

   İkisi de $1000 + 4 \cdot 4 = 1016$ adresini verir. İşaretçi aritmetiği öğe saydığı için `a + 4`, 4 bayt değil 4 öğe, yani 16 bayt ileridedir. `*(a + 4)`, `a[4]`'ün kendisidir.

2. `short s[4][7];` dizisi 500 adresinden başlıyor ve `sizeof(short) = 2`. Satır öncelikli düzende `&s[2][5]` kaçtır? Dizi sütun öncelikli saklansaydı ne olurdu?

   Cevap

   Satır öncelikli: öğeden önce 2 tam satır ($2 \cdot 7 = 14$ öğe) ve satırında 5 öğe vardır; $500 + 19 \cdot 2 = 538$. Sütun öncelikli: önce 5 tam sütun ($5 \cdot 4 = 20$ öğe) ve sütununda 2 öğe; $500 + 22 \cdot 2 = 544$. Satır öncelikli formülde satır uzunluğu (7), sütun öncelikli formülde sütun uzunluğu (4) kullanılır.

3. `int A[3][4][5];` dizisinde *A[2][1][3]*'ün Horner yöntemiyle bulunan bağıl konumu nedir? Hesapta kaç çarpma yapılır?

   Cevap

   t = 2; *t = 2 · 4 + 1 = 9*; *t = 9 · 5 + 3 = 48*. İki çarpma yapılır (*n − 1*, *n = 3*). İlk boyut olan 3 hesaba hiç girmez. Gerçek bir C dizisinde `&A[2][1][3] - &A[0][0][0]` da 48'dir.

4. *6 × 6* bir alt üçgen matris `ALT` dizisinde saklanıyor. Dizide kaç öğe vardır? a_{52} hangi konumdadır? `AltUcgenEris(2, 5, 6)` ne döndürür?

   Cevap

   6 · 7/2 = 21 öğe. a_{52}: *5 · 6/2 + 2 = 17*. `AltUcgenEris(2, 5, 6)` üst üçgendeki a_{25}'i sorar; *2 < 5* olduğu için −1 döner (değer 0, saklanmıyor).

5. *n = 5*, *a = 2*, *b = 2* olan bir kuşak matris (üç köşegenli matris) kaç öğe tutar? ARA dizisi nedir? d_{34} ve d_{43} `kusak` dizisinin hangi konumlarındadır?

   Cevap

   5 · 3 − 1 − 1 = 13 öğe. Kuşaklar −1, 0, 1 ve uzunlukları 4, 5, 4'tür; ARA = [0, 4, 9]. d_{34}: kuşak *4 − 3 = 1*, üst kuşak; konum *ara[1 + 1] + i = 9 + 3 = 12*. d_{43}: kuşak −1, alt kuşak; konum *ara[0] + j = 0 + 3 = 3*.

6. Boş başlayıp kapasitesini ikiye katlayan bir dinamik diziye 20 öğe eklenirse toplam kaç kopya yapılır ve son kapasite kaç olur? Kapasite her seferinde 1 artırılsaydı kopya sayısı ne olurdu?

   Cevap

   Büyütmeler 1, 2, 4, 8 ve 16 öğe dolduğunda olur: *1 + 2 + 4 + 8 + 16 = 31* kopya, son kapasite 32. Bu, *2 · 20 = 40*'tan azdır. 1 artırmada *0 + 1 + … + 19 = 190* kopya yapılır.

7. `struct { char a; int b; char c; double d; }` tipik bir 64 bitlik sistemde kaç bayt tutar? Alanlar yeniden sıralanarak boy kaça indirilebilir?

   Cevap

   `a` 0. bayt, 3 dolgu, `b` 4–7, `c` 8. bayt, 7 dolgu, `d` 16–23: toplam 24 bayt. `{ double d; int b; char a, c; }` sırasında *8 + 4 + 1 + 1 = 14* bayta 2 dolgu eklenir ve boy 16 olur. Sürücü programda iki değer de ölçülmüştür.

8. `for (i = 0; i < strlen(s); i++) if (s[i] == ' ') say++;` döngüsü 10.000 karakterlik bir dizgide yaklaşık kaç karakter okur? Döngü nasıl düzeltilir?

   Cevap

   Koşul her turda yeniden hesaplanır ve her `strlen` 10.000 karakter tarar. 10.001 denetimde yaklaşık 10^8 karakter okunur; döngü $O(n^2)$'dir. Uzunluk döngüden önce bir kez hesaplanır (`size_t len = strlen(s);`) ya da döngü `s[i] != '\0'` koşuluyla yazılır; ikisi de *O(n)*'dir.

9. *2 × 3* bir matris ile *3 × 4* bir matris çarpılıyor. Sonuç kaç hücrelidir ve kaç çarpma yapılır?

   Cevap

   Sonuç *2 × 4*, yani 8 hücrelidir. Her hücre 3 çarpımın toplamıdır: *2 · 3 · 4 = 24* çarpma.

10. `size_t f(int a[10]) { return sizeof a; }` fonksiyonu 64 bitlik bir sistemde ne döndürür? Neden?

    Cevap

    8 döndürür. Parametre `int a[10]` diye yazılsa da derleyici onu `int *a` olarak işler; `sizeof a` işaretçinin boyudur, 40 değildir. Dizinin uzunluğu fonksiyona ayrı bir parametreyle gönderilmelidir. Derleyiciler bu kullanım için uyarı verir.

2.15 İleri okuma

- E. Horowitz, S. Sahni ve S. Anderson-Freed, *Fundamentals of Data Structures in C* (2. baskı), Bölüm 2: dizi ADT'si, yapılar, polinom ve seyrek matris gösterimleri, çok boyutlu dizilerin adres hesabı ve dizgiler; bu notun izlediği ders kitabıdır.
- B. W. Kernighan ve D. M. Ritchie, *The C Programming Language* (2. baskı), Bölüm 5–6: işaretçiler ve diziler arasındaki ilişkiyi, işaretçi dizilerini, çok boyutlu dizileri ve yapıları dilin tasarımcılarının anlatımıyla verir.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (3. baskı), Kısım 17.4: dinamik tabloların büyütülmesini ve küçültülmesini toplama, muhasebe ve potansiyel yöntemleriyle amortize olarak analiz eder.

2.16 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3. baskı, MIT Press, 2009.
2. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, Silicon Press, 2008.
3. B. W. Kernighan, D. M. Ritchie, *The C Programming Language*, 2. baskı, Prentice Hall, 1988.

KISIM I · BÖLÜM 3

Özyineleme, İkili Arama ve Geri İzleme

Bazı problemler kendi içinde daha küçük bir kopyasını taşır. Bir sayının faktöriyeli, bir eksiğinin faktöriyeline dayanır. Sıralı bir dizide arama, dizinin yarısında yapılan aramaya indirgenir. Üç diski taşımak, iki diski taşımayı bilmeyi gerektirir. Bu not, böyle problemleri çözen özyinelemeli algoritmaları anlatır. Önce özyinelemenin makinede nasıl çalıştığını, yani çağrı yığınını ve yığın çerçevesini inceler. Ardından ikili aramayı döngüyle ve özyinelemeyle yazar, Hanoi kulelerinden yineleme bağıntısını çıkarır ve permütasyon üretir. Son kısım geri izleme yöntemine ayrılmıştır: alt küme üretimi, sağlanabilirlik (SAT), n-vezir ve at gezisi problemleri bu yöntemle çözülür. Her kod C ile verilir, adım adım izlenir ve zaman ile bellek maliyeti gerekçesiyle hesaplanır.

Kod (C): `github.com/sercankulcu/data-structures-and-algorithms-c/Bolum03`

ÖĞRENME ÇIKTILARI

Özyinelemeli düşünmek
Bir problemi taban durum ve küçülen alt problem olarak ifade etmek; özyinelemeyi döngüyle karşılaştırmak.

Yığını izlemek
Her çağrının yığın çerçevesini (parametreler, yerel değişkenler, dönüş adresi) çizmek ve özyinelemenin bellek maliyetini hesaplamak.

İkili arama
Yinelemeli ve özyinelemeli ikili aramayı C ile yazmak, bir dizi üzerinde izlemek ve $O(\log n)$ sınırını göstermek.

Yineleme bağıntısı
Hanoi kuleleri için $h(n) = 2h(n-1) + 1$ bağıntısını kurup çözmek; permütasyon üretiminin $O(n \cdot n!)$ olduğunu göstermek.

Geri izleme
Kaba kuvvet ile geri izlemeyi ayırmak; alt küme üretimi, SAT ve n-vezir problemlerini bu yöntemlerle çözmek.

Sezgisel yöntem
At gezisinde Warnsdorff kuralının geri izlemeden neden çok daha hızlı olduğunu açıklamak.

3.1 Özyinelemeli algoritmalar

Programlamaya yeni başlayan biri bir işi tekrarlamak için önce döngüyü öğrenir. Döngü, aynı gövdeyi bir koşul sağlandığı sürece yeniden çalıştırır. Özyineleme tekrarı başka bir yoldan elde eder: fonksiyon, işin küçük bir parçasını yapmak için kendini çağırır. İlk karşılaşmada tuhaf görünen bu fikir, matematikte çok eskidir. Tümevarımla yapılan tanımlar ve ispatlar aynı yapıdadır.

3.1.1 Özyineleme nedir?

Bir problemin bütününde kullanılan çözüm mantığı, problemin parçalarında da geçerliyse özyinelemeli bir algoritma yazılabilir. Faktöriyel bunun en yalın örneğidir. $n! = n \cdot (n-1)!$ eşitliği, n'nin faktöriyelini bir küçük sayının faktöriyeline bağlar. $(n-1)!$ de aynı kuralla $(n-2)!$'ye bağlanır. Zincir $1! = 1$'de durur.

> **Tanım 3.1**
>
> **Özyineleme** (*recursion*), bir fonksiyonun kendini daha küçük bir girdiyle çağırmasıdır. Kendini çağıran fonksiyona **özyinelemeli** (*recursive*) fonksiyon denir. Fonksiyon kendini doğrudan çağırıyorsa **doğrudan özyineleme** vardır: `fakt` → `fakt`. Fonksiyon başka bir fonksiyonu çağırıyor, o da zincirin sonunda ilk fonksiyonu yeniden çağırıyorsa **dolaylı özyineleme** vardır: `A` → `B` → `A`.

Dolaylı özyinelemenin tanıdık bir örneği, ifade çözümleyicilerdir. Bir aritmetik ifadeyi çözümleyen `ifade()` fonksiyonu terimleri okumak için `terim()` fonksiyonunu çağırır. Parantez içinde yeni bir ifade başlayınca `terim()` yeniden `ifade()`'yi çağırır. İki fonksiyon kendilerini hiç doğrudan çağırmaz, ama zincir kapalı bir döngü oluşturur. C'de dolaylı özyineleme yazarken ikinci fonksiyonun bildirimi (prototipi) birinci fonksiyondan önce verilmelidir; derleyici aksi hâlde henüz görmediği bir adı tanımaz.

3.1.2 Taban durum ve özyinelemeli adım

Kendini çağıran bir fonksiyonun bir yerde durması gerekir. Faktöriyel zinciri *1! = 1*'de durur, çünkü bu değer başka bir faktöriyele başvurmadan bilinir. Her özyinelemeli algoritma bu iki parçadan oluşur.

Tanım 3.2

Taban durum (*base case*), özyinelemeye başvurmadan doğrudan çözülen en küçük örnektir. **Özyinelemeli adım** (*recursive step*) ise problemi aynı türden daha küçük bir örneğe indirger ve o örneğin çözümünden asıl çözümü kurar. Her özyinelemeli çağrı taban duruma biraz daha yaklaşmalıdır.

Tanımdaki son cümle, özyinelemenin doğruluğu için şarttır. “Daha küçük” sözü bir ölçüyle ifade edilir: faktöriyelde *n*, ikili aramada aralıktaki öğe sayısı, Hanoi kulelerinde disk sayısı. Ölçü her çağrıda azalmalı ve taban durumda en küçük değerine ulaşmalıdır. Ölçü azalmıyorsa çağrılar sonsuza dek sürer. Gerçek bir makinede “sonsuza dek” uzun sürmez, çünkü her çağrı bellekte yer tutar ve ayrılan bellek kısa sürede biter.

Özyinelemeli bir fonksiyon yazarken iki soru sorulur. Birincisi, en küçük örnek hangisidir ve cevabı nedir? İkincisi, bir boy küçük örneğin cevabı bilinseydi asıl cevap nasıl kurulurdu? İkinci soru, özyinelemeli düşünmenin özüdür. Küçük örneğin nasıl çözüldüğü düşünülmez; çözüldüğü kabul edilir. Bu kabul, tümevarım ispatındaki “*n − 1* için doğru olsun” varsayımının aynısıdır.

Sık yapılan hata

Taban durumu unutmak ya da ona hiç ulaşmayan bir adım yazmak. `long f(int n) { return n * f(n - 1); }` fonksiyonunda taban durum yoktur; çağrılar *n = 0*'ı geçip negatif sayılarla sürer ve program **yığın taşması** (*stack overflow*) ile çöker. Taban durum yazılmış ama adım ona yaklaşmıyorsa sonuç aynıdır: `if (n == 1) return 1;` taban durumu `f(0)` çağrısını hiç yakalamaz. Faktöriyelde `n <= 1` koşulu hem 0'ı hem 1'i kapsadığı için daha güvenlidir.

Yeni başlayanların sık sorduğu bir soru vardır: özyinelemeli çözüm çok bellek harcamaz mı? Bu soruya cevap vermek için önce özyinelemeli kodu yazabilmek, sonra o kodun karmaşıklığını hesaplayabilmek gerekir. Özyinelemeli kod yazmak öğrenilmeden yapılan bir “çok bellek harcar” yargısı temelsiz kalır. Bu notun sırası da bu yüzden böyledir: önce kod, sonra ölçüm. Bellek maliyeti çağrı yığını incelenince somut bir sayıya dönüşür.

3.1.3 İlk örnek: faktöriyel

Faktöriyelin özyinelemeli tanımı, C'ye neredeyse birebir aktarılır:

```c
long fakt(int n) {
  if (n <= 1)
    return 1;              /* taban durum */
  return n * fakt(n - 1);
}

int main(void) {
  printf("%ld\n", fakt(3));
  return 0;
}
```

Fonksiyon `long` türünde bir değer döndürür, çünkü faktöriyel çok hızlı büyür. İlk satırdaki `if (n <= 1)` taban durumu yakalar ve 1 döndürür. Son satır özyinelemeli adımdır: `n * fakt(n - 1)` ifadesi önce `fakt(n - 1)`'i hesaplatır, dönen değeri `n` ile çarpar. `main` fonksiyonu `fakt(3)`'ü çağırıp sonucu `%ld` biçimiyle yazar; çıktı 6'dır.

Kodun okunuşu tanımın okunuşuyla aynıdır: “*n* en çok 1 ise faktöriyel 1'dir, değilse *n* çarpı bir eksiğinin faktöriyelidir.” Döngülü bir sürümde bu cümle yoktur; onun yerine bir sayaç ve bir birikim değişkeni bulunur. Özyinelemenin en büyük kazancı bu okunabilirliktir.

Kodun doğruluğu tümevarımla gösterilir. $n \le 1$ için fonksiyon 1 döndürür ve $0! = 1! = 1$'dir. $n > 1$ için `fakt(n - 1)` 'in $(n-1)!$ döndürdüğü kabul edilirse, fonksiyon $n \cdot (n-1)! = n!$ döndürür. Taban durum ve adım birlikte her $n \ge 0$ için doğruluğu verir.

`long` türünün boyu sisteme göre değişir ve bu fark faktöriyelde hemen görünür. 64 bit Linux ve macOS'ta `long` 64 bittir; en büyük değeri $2^{63} - 1 \approx 9{,}2 \cdot 10^{18}$'dir ve $20! \approx 2{,}4 \cdot 10^{18}$ bu sınıra sığar. Windows'ta ise `long` 32 bittir; en büyük değeri $2^{31} - 1 = 2.147.483.647$'dir. $12! = 479.001.600$ sığar, $13! = 6.227.020.800$ sığmaz. İşaretli tam sayı taşması C'de **tanımsız davranıştır**: program yanlış bir sayı yazabilir, beklenmedik biçimde davranabilir. Bu notun kodları `-fsanitize=undefined` seçeneğiyle derlenip Windows'ta denendiğinde `fakt(13)` çağrısı tam da bu hatayı bildirdi. Büyük faktöriyeller için `long long` (en az 64 bit) ya da büyük sayı kitaplıkları kullanılır.

3.2 Çağrı yığını

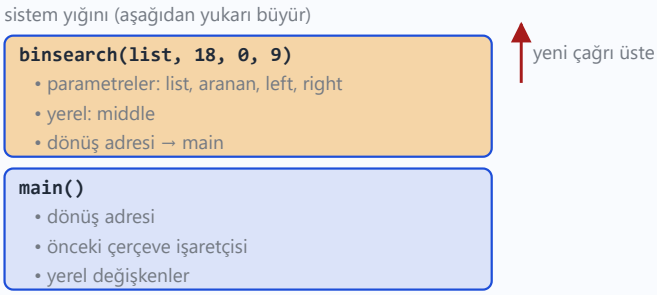
Özyineleme kâğıt üzerinde bir tanımdır, ama bilgisayarda gerçek bir mekanizmayla çalışır. Bu mekanizmayı anlamak hem özyinelemeli kodu izleyebilmek hem de bellek maliyetini hesaplayabilmek için gerekir. Mekanizmanın adı çağrı yığınıdır ve her fonksiyon çağrısında, özyinelemeli olsun olmasın, kullanılır.

3.2.1 Yığın çerçevesi

Bir fonksiyon çağrıldığında denetim hemen fonksiyonun gövdesine geçmez. Önce bir hazırlık yapılır. Fonksiyon bittiğinde programın nereye döneceği, yani **dönüş adresi** saklanır. Parametrelerin değerleri fonksiyonun erişebileceği bir yere kopyalanır. Fonksiyonun yerel değişkenleri için yer ayrılır. Bütün bu bilgiler bellekte art arda duran tek bir kayıtta toplanır.

> **Tanım 3.3**
>
> **Yığın çerçevesi** (*stack frame, activation record*), bir fonksiyon çağrısının parametrelerini, yerel değişkenlerini, dönüş adresini ve bir önceki çerçeveye işaretçiyi tutan kayıttır. Etkin çağrıların çerçeveleri **çağrı yığını** (*call stack*) denen bellek bölgesinde, çağrılma sırasıyla üst üste durur.

Çağrı yığını, adından da anlaşılacağı gibi bir yığın yapısıdır: son giren ilk çıkar. Bir fonksiyon çağrılınca çerçevesi yığının üstüne eklenir. Fonksiyon `return` ile bitince çerçevesi üstten kaldırılır ve denetim, çerçevedeki dönüş adresine geçer. Program başladığında yığında yalnız `main` 'in çerçevesi vardır. Bu ilk çerçevede de dönüş adresi ve önceki çerçeveye işaretçi bulunur; `main` bittiğinde denetim işletim sistemine döner.



Şekil 3.1: `main` , `binsearch(list, 18, 0, 9)` 'u çağırdığında sistem yığını. Yeni çağrının çerçevesi üste eklenir; parametreleri, yerel `middle` değişkenini ve `main` 'e dönüş adresini tutar.

Şekil 3.1 bu düzeni gösterir. Yığın şekilde aşağıdan yukarı büyür; en alttaki çerçeve `main` 'indir. `binsearch` çağrılınca onun çerçevesi üste eklenir. Bu çerçevede dört parametre (`list` , `aranan` , `left` , `right`), bir yerel değişken (`middle`) ve `main` 'e dönüş adresi vardır. `list` parametresinin dizinin kendisi olmadığına dikkat etmek gerekir. C'de dizi parametresi, dizinin ilk öğesinin adresini taşıyan bir işaretçiye dönüşür. Çerçevede bu adres durur; dizinin on öğesi `main` 'in çerçevesinde kalır ve kopyalanmaz.

Gerçek bir derleyicinin çerçeve düzeni işlemciye ve derleyiciye göre değişir. Bazı parametreler bellek yerine işlemci yazmaçlarında taşınabilir. Yine de mantık her yerde aynıdır: her etkin çağrının kendi çerçevesi vardır ve bu çerçeveler bir yığın düzeninde tutulur. Özyinelemeyi anlamak için bu mantık yeterlidir.

Özyinelemenin özel bir durumu yoktur. Fonksiyon kendini çağırdığında da yeni bir çerçeve açılır. Aynı fonksiyonun birden çok çağrısı aynı anda yığında bulunabilir ve her birinin kendi parametre ve yerel değişken kopyaları olur. Bir çağrıdaki `n`, öteki çağrıdaki `n`'den tamamen ayrıdır.

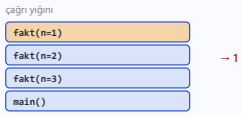
3.2.2 Kod izleme: fakt(3)

`fakt(3)` çağrısının yığını nasıl değiştirdiği adım adım izlenebilir. Tablo 3.1, her adımda çalışan satırı, o anki `n` değerini ve yığının içeriğini verir. Yığın soldan sağa yazılmıştır; en sağdaki çerçeve yığının üstüdür.

Adım	Çalışan satır	n	Yığın (alttan üste)	Olay
1	`printf(…, fakt(3))`	—	main	`main`, `fakt(3)`'ü çağırır.
2	`if (n <= 1)`	3	main | fakt(3)	*3 > 1*, taban durum değil.
3	`return n * fakt(n - 1);`	3	main | fakt(3)	Çarpma bekler; önce `fakt(2)` çağrılır.
4	`if (n <= 1)`	2	main | fakt(3) | fakt(2)	*2 > 1*, taban durum değil.
5	`return n * fakt(n - 1);`	2	main | fakt(3) | fakt(2)	Çarpma bekler; önce `fakt(1)` çağrılır.
6	`if (n <= 1) return 1;`	1	main | fakt(3) | fakt(2) | fakt(1)	Taban durum: 1 döner. Yığın en derin hâlde.
7	`return n * fakt(n - 1);`	2	main | fakt(3) | fakt(2)	*2 · 1 = 2* döner, çerçeve kapanır.
8	`return n * fakt(n - 1);`	3	main | fakt(3)	*3 · 2 = 6* döner, çerçeve kapanır.
9	`printf(…, fakt(3))`	—	main	6 yazılır.

Tablo 3.1: `fakt(3)` izlemesi ve çağrı yığını.

İzlemenin ilk yarısı bir iniştir. Adım 3'te `fakt(3)`, `3 * fakt(2)` ifadesini hesaplamaya başlar ama çarpmayı yapamaz, çünkü ikinci çarpan henüz bilinmez. Bu yüzden `fakt(2)`'yi çağırır ve kendi çerçevesi yığında beklemeye geçer. Aynı şey adım 5'te `fakt(2)` için olur. Adım 6'da `fakt(1)` taban duruma ulaşır ve hiçbir çağrı yapmadan 1 döndürür. Bu anda yığında dört çerçeve vardır: `main` ve üç `fakt` çağrısı.



Şekil 3.2: `fakt(1)` taban duruma ulaştığında çağrı yığını. Üç `fakt` çerçevesinin her birinde ayrı bir `n` vardır.

İkinci yarı bir çıkıştır. `fakt(1)`'in çerçevesi kaldırılır ve denetim `fakt(2)`'nin bıraktığı yere, yarım kalan çarpmaya döner. `fakt(2)`'nin çerçevesinde `n` hâlâ 2'dir. Şekil 3.2'deki her kutunun kendi `n` kopyası olması bu yüzden önemlidir. `fakt(1)` çalışırken `n` 1 olmuştu, ama bu değişiklik yalnız `fakt(1)`'in çerçevesindeydi. Çarpma *2 · 1 = 2* sonucunu verir ve `fakt(2)` biter. Aynı biçimde `fakt(3)` *3 · 2 = 6* hesaplar. Son olarak `main` 6 yazar ve yığında yalnız `main` kalır.

Bu izleme özyinelemeli fonksiyonların genel davranışını gösterir. Çağrılar önce taban duruma kadar iner; asıl hesaplama çoğunlukla dönüş yolunda yapılır. Faktöriyelde çarpmaların hepsi çıkış sırasında, en içteki çağrıdan dışa doğru yapılır: önce *2 · 1*, sonra *3 · 2*.

3.2.3 Özyinelemenin bellek maliyeti

Her özyinelemeli çağrı yeni bir çerçeve demektir. Çerçeveler ancak çağrı bitince kalkar. Dolayısıyla bir anda yığında duran çerçeve sayısı, özyineleme zincirinin o anki uzunluğuna eşittir. Bu uzunluğun en büyük değerine **özyineleme derinliği** denir.

> **Teorem 3.1 (Özyinelemenin yığın maliyeti)**
>
> Özyineleme derinliği *d* olan ve her çerçevesi sabit boyda olan bir fonksiyonun çağrı yığınında kullandığı ek bellek *O(d)*'dir.

İspat. Bir çerçevenin boyu parametre ve yerel değişken sayısına bağlıdır; bu sayılar koddan bellidir ve girdiyle değişmez. Çerçeve boyu *c* bayt olsun. Herhangi bir anda yığında en çok *d* özyinelemeli çerçeve bulunur, çünkü özyineleme derinliği, çağrı zincirinin ulaşabildiği en büyük uzunluktur. Kullanılan bellek en çok *c · d* bayttır, yani *O(d)*'dir. □

`fakt(n)` için derinlik *n*'dir: `fakt(n)`, `fakt(n - 1)`, …, `fakt(1)` zinciri *n* çerçeve açar. Ek bellek *O(n)*'dir. Zaman maliyeti de *Θ(n)*'dir, çünkü *n* çağrının her biri sabit sayıda işlem (bir karşılaştırma, bir çarpma, bir dönüş) yapar.

Çağrı yığınının boyu sınırlıdır. İşletim sistemi her iş parçacığına sabit bir yığın alanı ayırır; bu alan Windows'ta varsayılan olarak 1 MB, Linux'ta çoğunlukla 8 MB'tır. Çerçeve başına birkaç on bayt harcayan bir fonksiyon bile yüz binlerce derinliğe inerse bu alanı doldurur ve program yığın taşmasıyla çöker. Bu yüzden derinliği *n* olan özyineleme büyük *n* için tehlikelidir; derinliği *log n* olan özyineleme ise pratikte hiç sorun çıkarmaz. Bir milyar öğeli dizide ikili arama bile 30 çerçeveden derine inmez.

> **Sık yapılan hata**
>
> Özyinelemeli bir fonksiyonun yalnız zaman maliyetini hesaplayıp yığın maliyetini unutmak. `fakt` ile döngülü faktöriyel aynı *Θ(n)* zamanda çalışır, ama özyinelemeli sürüm *O(n)*, döngülü sürüm *O(1)* ek bellek kullanır. Özyinelemeli bir algoritmanın yığın maliyeti, en derin çağrı zincirindeki çerçevelerin toplam boyudur.

3.3 Özyineleme mi, döngü mü?

Aynı problemi hem özyinelemeyle hem döngüyle çözmek çoğu zaman mümkündür. Hangisinin seçileceği, okunabilirlik, bellek ve hız arasında bir dengeye bağlıdır.

3.3.1 İki faktöriyel

Faktöriyelin iki sürümü yan yana şöyledir:

```c
/* ozyinelemeli */
long fakt(int n) {
  return n <= 1 ? 1 : n * fakt(n - 1);
}

/* dongulu */
long fakt_d(int n) {
  long f = 1;
  while (n > 1)
    f *= n--;
  return f;
}
```

Özyinelemeli sürüm tek satıra inmiştir. `?:` koşul işleci, `if-else` yapısının ifade biçimidir: koşul doğruysa iki noktadan önceki, yanlışsa sonraki değeri verir. Döngülü sürüm `f` adında bir birikim değişkeni kullanır. `f *= n--;` satırı iki iş yapar: önce `f`'yi `n` ile çarpar, sonra `n`'yi bir azaltır. Sonek `--` işleci, ifadede eski değeri kullanıp değişkeni sonra azaltır. `fakt_d(4)` çağrısında `f` sırasıyla 4, 12, 24 olur ve *n = 1* olunca döngü durur.

Döngülü sürümde `n` parametresinin değiştirilmesi çağıranı etkilemez. C'de parametreler değerle aktarılır: fonksiyon, çağıranın değişkeninin bir kopyasıyla çalışır. `fakt_d(x)` çağrısından sonra `x` değişmeden kalır.

3.3.2 Karşılaştırma

İki yaklaşımın farkları Tablo 3.2'de toplanmıştır.

Ölçüt	Özyinelemeli	Döngülü
Okunabilirlik	Tanıma çok yakın	Durum değişkenleri gerekir
Ek bellek	*O(derinlik)* yığın	*O(1)*
Çağrı maliyeti	Her adımda bir çerçeve	Yok
Risk	Derin özyinelemede yığın taşması	Sonsuz döngü
Doğal olduğu yer	Ağaç, böl ve fethet, geri izleme	Düz tarama, sayaçlar

Tablo 3.2: Özyineleme ile döngünün karşılaştırması.

Tablonun "çağrı maliyeti" satırı hızla ilgilidir. Her çağrıda çerçeve kurulur, parametreler kopyalanır, dönüşte çerçeve kaldırılır. Bu işler birkaç makine komutudur, ama döngünün bir turundan pahalıdır. Faktöriyel gibi basit bir işte özyinelemeli sürüm döngülü sürümden biraz yavaş çalışır. Asimptotik mertebe ise aynıdır; fark yalnız sabit çarpandadır.

"Doğal olduğu yer" satırı daha önemlidir. Faktöriyelde döngü de özyineleme kadar açıktır. Ağaç gezinmesinde, Hanoi kulelerinde ya da geri izlemede durum böyle değildir. Bu problemlerde bir çağrı birden çok alt çağrı yapar ve her alt çağrının nerede kaldığını hatırlamak gerekir. Özyineleme bu kaydı çağrı yığınına bırakır; döngülü sürüm aynı kaydı programcının elle tutmasını ister.

3.3.3 Özyinelemeyi döngüye çevirmek

Her özyinelemeli algoritma döngüye çevrilebilir ve her döngü özyinelemeye çevrilebilir. Çeviri yöntemi çağrı yığınını taklit etmektir: programcı bir dizi ya da yığın yapısı tanımlar, özyinelemenin saklayacağı bilgileri oraya koyar ve bir döngüyle işler. Aşağıdaki fonksiyon faktöriyeli bu yolla hesaplar:

```
long fakt_y(int n) {
  int yigin[64], ust = 0;         /* cagri yiginini taklit eden dizi */
  long f = 1;
  while (n > 1)                   /* inis: carpanlari yigina it */
    yigin[ust++] = n--;
  while (ust > 0)                 /* donus: bekleyen carpmalari yap */
    f *= yigin[--ust];
  return f;
}
```

İlk döngü, özyinelemenin iniş aşamasına karşılık gelir. Her turda bekleyen bir çarpanı ($n, n-1, \ldots, 2$) diziye iter. İkinci döngü çıkış aşamasıdır: çarpanları ters sırayla alıp birikimle çarpar. `yigin` dizisinin boyu 64 seçilmiştir; faktöriyel $n = 21$'de zaten 64 bitlik sayıya sığmadığından bu boy yeterlidir. Özyinelemeli `fakt`'ın çağrı yığınında tuttuğu bilgi tam olarak bu dizidir. Faktöriyelde bu çeviri gereksiz bir zahmettir, çünkü çarpmalar sıradan bağımsızdır ve tek bir değişken yeter. Ağaç gezinmesi gibi problemlerde ise açık yığın, özyinelemesiz çözümün en doğal yoludur.

Seçim bir dengedir. Kod okunabilirliği öndeyse ve derinlik küçükse (örneğin $O(\log n)$) özyineleme iyi bir seçimdir. Derinlik büyük olabiliyorsa ya da her çağrının maliyeti önemliyse döngü tercih edilir.

3.4 Kuyruk özyinelemesi

Bazı özyinelemeli fonksiyonlar özel bir biçimdedir: özyinelemeli çağrı, fonksiyonun yaptığı son iştir. Bu biçimdeki fonksiyonlar döngüye mekanik olarak çevrilebilir ve derleyici bu çeviriyi kendisi yapabilir.

3.4.1 Tanım ve birikimli parametre

> **Tanım 3.4**
>
> Özyinelemeli çağrı fonksiyonun son işiyse, yani çağrı döndükten sonra yapılacak hiçbir iş kalmıyorsa, bu özyinelemeye **kuyruk özyinelemesi** (*tail recursion*) denir. Böyle bir çağrıya **kuyruk çağrısı** adı verilir.

`fakt` kuyruk özyinelemeli değildir. `return n * fakt(n - 1);` satırında özyinelemeli çağrı döndükten sonra bir çarpma yapılır. Bu çarpma için `n`'nin değeri saklanmalıdır; dolayısıyla çerçeve, alt çağrı bitene kadar yığında kalmak zorundadır. Faktöriyelin kuyruk özyinelemeli sürümü, bekleyen çarpmayı ortadan kaldırmak için ek bir parametre kullanır:

```
long fakt_k(int n, long acc) {
  if (n <= 1) return acc;
  return fakt_k(n - 1, n * acc);  /* son is: cagri */
}
/* ilk cagri: fakt_k(n, 1) */
```

`acc` (*accumulator*) **birikimli parametre**dir ve o ana kadar hesaplanan kısmi çarpımı taşır. Çarpma, özyinelemeli çağrıdan *önce*, argüman hesaplanırken yapılır: `n * acc`. Çağrı döndüğünde yapılacak iş kalmaz; dönen değer olduğu gibi yukarı iletilir. Taban durumda birikim tam sonuçtur ve doğrudan döndürülür. İlk çağrıda birikim 1'dir, çünkü 1 çarpmanın etkisiz öğesidir.



Şekil 3.3: `fakt_k(4, 1)` çağrı zinciri. Kısmi çarpım parametrede ilerler; en içteki çağrının döndürdüğü 24 hiçbir işlem görmeden yukarı iletilir.

Örnek 3.1

`fakt_k(4, 1)` çağrısı izlenirse:

1. `fakt_k(4, 1)` : $n = 4 > 1$; `fakt_k(3, 4 · 1) = fakt_k(3, 4)` çağrılır.
2. `fakt_k(3, 4)` : `fakt_k(2, 3 · 4) = fakt_k(2, 12)` çağrılır.
3. `fakt_k(2, 12)` : `fakt_k(1, 2 · 12) = fakt_k(1, 24)` çağrılır.
4. `fakt_k(1, 24)` : taban durum; `acc = 24` döner.
5. 24, hiçbir işlem görmeden üç çerçeveden geçerek ilk çağırana ulaşır.

`fakt` 'ta çarpmalar dönüş yolunda, içten dışa yapılıyordu: $2 \cdot 1, 3 \cdot 2, 4 \cdot 6$. `fakt_k` 'da iniş yolunda, dıştan içe yapılır: $4 \cdot 1, 3 \cdot 4, 2 \cdot 12$. Çarpma değişmeli ve birleşmeli olduğu için sonuç aynıdır.

3.4.2 Derleyici ve kuyruk çağrısı

Kuyruk çağrısından sonra çerçevede bekleyen iş olmadığı için çerçeveyi saklamanın bir anlamı yoktur. Derleyici yeni çerçeve açmak yerine mevcut çerçevedeki parametreleri yeni değerlerle günceller ve fonksiyonun başına **atlar**. Bu dönüşüme kuyruk çağrısı eniyilemesi (*tail call optimization*) denir. Sonuç, döngülü sürümle aynı makine kodudur ve yığın maliyeti $O(1)$'e iner.

C standardı bu eniyilemeyi **garanti etmez**. GCC ve Clang, `-O2` düzeyinde kuyruk çağrılarını çoğu zaman atlamaya çevirir; eniyileme kapalı derlemede (`-O0`) ise her çağrı yeni bir çerçeve açar. Dolayısıyla derinliği çok büyük olabilecek bir kuyruk özyinelemesine C'de güvenmek doğru değildir. Böyle bir fonksiyonu elle döngüye çevirmek kolaydır: parametreler döngü değişkeni olur, özyinelemeli çağrı yerine parametrelere yeni değerler atanır, taban durum döngünün çıkış koşulu olur. `fakt_k` bu yolla `fakt_d` 'ye dönüşür.

Sık yapılan hata

Çağrının satırın sonunda durmasını kuyruk özyinelemesi sanmak. `return n * fakt(n - 1);` satırında çağrı yazı olarak sonda görünür, ama son yapılan iş çarpmadır. Ölçüt yazım sırası değil, çağrı döndükten sonra iş kalıp kalmadığıdır. `return fakt_k(n - 1, n * acc);` satırında çağrının sonucu olduğu gibi döndürülür; bu bir kuyruk çağrısıdır.

3.5 İkili arama

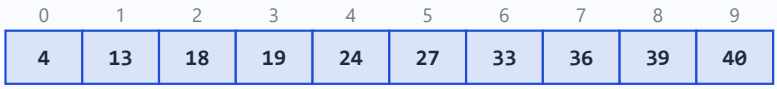
Arama, bir veri kümesinde istenen değeri bulma işidir. Sırasız bir dizide yapılabilecek en iyi şey öğelere tek tek bakmaktır; buna doğrusal arama denir ve en kötü durumda n karşılaştırma ister. Dizi sıralıysa çok daha iyisi yapılabilir. Sıralı bir dizide tek bir karşılaştırma, aranan değerin hangi yarıda olduğunu söyler. İkili arama bu bilgiyi her adımda kullanır.

3.5.1 Problem ve COMPARE makrosu

Tanım 3.5

İkili arama (*binary search*), artan sırada dizilmiş bir dizide aranan değeri bulan yordamdır. Her adımda aralığın ortasındaki öğeye bakar. Ortadaki öğe aranana eşitse indisini döndürür. Arananan küçükse sol yarıyı, büyükse sağ yarıyı atar ve kalan yarıda aramayı sürdürür. Aralık boşalırsa değer dizide yoktur ve sonuç -1'dir.

Bu bölümdeki bütün ikili arama örnekleri aynı test dizisini kullanır:



Şekil 3.4: Test dizisi `list` : on öğe, artan sırada. Kutuların üstündeki sayılar indislerdir.

Yordam aranan değerin kendisini değil, **indisini** döndürür. Bunun iki nedeni vardır. Birincisi, aranan değer zaten bilinir; asıl merak edilen onun dizideki yeridir. İndis bilinince o konumdaki kaydın başka alanlarına da ulaşılabilir. İkincisi, "bulunamadı" bilgisinin değerlerle karışmaması gerekir. Yordam `list[i]` döndürseydi, hata işareti olarak seçilen *−1* dizideki gerçek bir *−1* öğesiyle karışabilirdi. C'de dizi indisleri 0'dan başladığı için *−1* hiçbir geçerli indise denk gelmez ve "yok" anlamını güvenle taşır.

Karşılaştırma sonucu üç durumdan biridir: küçük, eşit ya da büyük. Kodu sade tutmak için bu üç durumu bir sayıya çeviren bir makro tanımlanır:

```
#define COMPARE(x, y) ((x) < (y) ? -1 : (y) < (x) ? 1 : 0)
```

`COMPARE(x, y)` , *x < y* ise *−1*, *x > y* ise 1, eşitse 0 verir. Makro iç içe iki koşul işleciyle yazılmıştır. İlk koşul `(x) < (y)` doğruysa sonuç *−1*'dir. Değilse ikinci koşul `(y) < (x)` sınanır; doğruysa 1, değilse 0 çıkar. Makro parametrelerinin her geçişte parantez içine alınması önişlemcinin metin değiştirme kuralından kaynaklanır. `COMPARE(a + 1, b)` gibi bir çağrıda parantezler, ifadenin işleç önceliği yüzünden yanlış gruplanmasını önler.

Makronun dönüşü, `switch` deyiminde üç ayrı dala bağlanabilir. Sayısal karşılaştırma için `x - y` farkı da kullanılabilirdi, ama büyük sayılarda fark taşabilir. Makro, çıkarma yapmadığı için bu riski taşımaz.

3.5.2 Yinelemeli binsearch()

İkili aramanın döngüyle yazılmış sürümü şöyledir:

```
int binsearch(int list[], int aranan,
              int left, int right) {
  while (left <= right) {
    int middle = (left + right) / 2;
    switch (COMPARE(list[middle], aranan)) {
      case -1: left = middle + 1; break;
      case 0: return middle;
      case 1: right = middle - 1;
    }
  }
  return -1;
}
```

Fonksiyon dört parametre alır. `list` sıralı dizinin başlangıç adresidir, `aranan` bulunacak değerdir. `left` ve `right` , aramanın sürdüğü aralığın sol ve sağ uç indisleridir. Tüm dizide aramak için `binsearch(list, x, 0, n - 1)` çağrılır. Aralık uçları kapalıdır: `left` ve `right` indislerindeki öğeler de aralığa dahildir.

`while (left <= right)` koşulu aralığın boş olmadığını denetler. Kapalı uçlu bir aralıkta *left = right* durumu tek öğeli bir aralıktır ve hâlâ aranmalıdır; bu yüzden koşul `<` değil `<=` ile yazılır. Döngünün her turunda orta indis `middle = (left + right) / 2` ile hesaplanır. C'de iki tam sayının bölümü, kesirli kısmı atılmış bir tam sayıdır; *(0 + 9)/2* sonucu 4'tür.

`switch` deyimi `COMPARE(list[middle], aranan)` sonucuna göre üç dala ayrılır:

- `case -1` : ortadaki öğe arananından küçüktür. Aranan, varsa, `middle` 'ın sağındadır. Sol uç `middle + 1` yapılır; `middle` artık aralıkta değildir, çünkü ona bakıldı. `break` , `switch` 'ten çıkıp döngünün başına dönmeyi sağlar.
- `case 0` : eşitlik. Aranan bulunmuştur ve indisi döndürülür. `return` , hem `switch` 'ten hem döngüden hem fonksiyondan çıkar.
- `case 1` : ortadaki öğe arananından büyüktür. Aranan, varsa, `middle` 'ın solundadır; sağ uç `middle - 1` yapılır. Bu son dal olduğundan `break` yazılmasa da `switch` zaten biter.

Döngü, aralık boşalınca (*left > right*) sona erer ve fonksiyon *−1* döndürür. Döngünün her turunda aralık en az bir öğe küçülür: ya bir dal `return` ile biter ya da `left` artar ya da `right` azalır. Bu yüzden döngü her zaman sonlanır.

> **Sık yapılan hata**
>
> Uçları `left = middle` ya da `right = middle` diye güncellemek. Aralıkta iki öğe kaldığında, örneğin *left = 2*, *right = 3* iken, `middle` 2 olur. Aranan sağdaysa `left = middle` ataması `left` 'i 2'de bırakır ve döngü aynı aralıkla sonsuza dek döner. *±1* eklemek hem sonlanmayı garanti eder hem de bakılmış öğeyi aralıktan çıkarır.

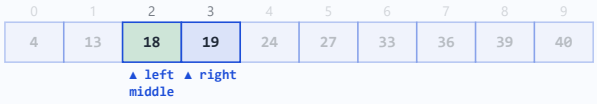
3.5.3 Kod izleme: 18 aranıyor

`binsearch(list, 18, 0, 9)` çağrısı izlenirse aralığın nasıl daraldığı görülür. Tablo 3.3 her turdaki değişken değerlerini verir.

Tur	left	right	middle	list[middle]	COMPARE	İşlem
1	0	9	4	24	1	`right = 3`
2	0	3	1	13	*−1*	`left = 2`
3	2	3	2	18	0	`return 2`

Tablo 3.3: `binsearch(list, 18, 0, 9)` izlemesi.

Birinci turda aralık bütün dizidir. Orta indis *(0 + 9)/2 = 4*, ortadaki öğe 24'tür. *24 > 18* olduğu için `COMPARE` 1 verir: 18 varsa 24'ün solundadır. Sağ yarı, yani 4–9 indisleri, tek karşılaştırmayla atılır ve `right = 3` olur. İkinci turda orta *(0 + 3)/2 = 1*, öğe 13'tür. *13 < 18* olduğundan sol taraf atılır: `left = 2` . Üçüncü turda orta *(2 + 3)/2 = 2*'dir ve `list[2] = 18` aranana eşittir. Fonksiyon 2 döndürür. On öğeli dizide değer üç karşılaştırmayla bulunmuştur.



Şekil 3.5: Üçüncü tur: `left = middle = 2` , `right = 3` . Gri hücreler atılmış kısımlardır; `list[2] = 18` bulunur.

3.5.4 Kod izleme: 25 dizide yok

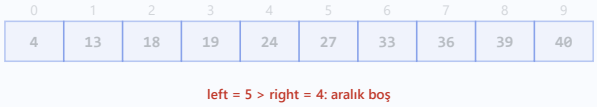
Başarısız bir arama, döngünün nasıl sona erdiğini gösterir. `binsearch(list, 25, 0, 9)` çağrısı Tablo 3.4'teki gibi ilerler.

Tur	left	right	middle	list[middle]	COMPARE	İşlem
1	0	9	4	24	*−1*	`left = 5`
2	5	9	7	36	1	`right = 6`
3	5	6	5	27	1	`right = 4`
—	5	4	—	—	—	*left > right*: `return -1`

Tablo 3.4: `binsearch(list, 25, 0, 9)` izlemesi.

Birinci turda *24 < 25* çıkar ve arama sağ yarıya geçer: `left = 5` . İkinci turda orta *(5 + 9)/2 = 7*'dir; *36 > 25* olduğundan `right = 6` olur. Üçüncü turda orta *(5 + 6)/2 = 5*, öğe 27'dir; *27 > 25* olduğundan `right = 4` olur. Artık *left = 5 > right = 4*'tür ve aralık boştur. Döngü koşulu yanlış çıkar, fonksiyon *−1* döndürür.

Döngü bittiğinde `left` 'in gösterdiği yer de bir bilgi taşır. *left = 5*, 25 dizide olsaydı sıralamayı bozmadan yerleşeceği indistir: 24 ile 27 arası. Döngü boyunca `left` 'in solundaki her öğe aranandan küçük, `right` 'ın sağındaki her öğe büyük kalır. Aralık boşaldığında bu iki bölge yan yana gelir ve `left` ilk büyük öğeyi gösterir. Sıralı bir diziye ekleme yapan programlar bu gözlemi kullanır.



Şekil 3.6: Başarısız aramanın sonu: `left = 5 > right = 4`, aralık boştur ve fonksiyon *−1* döndürür.

3.5.5 Özyinelemeli binsearch()

İkili arama özyinelemeli bir yapıdadır: aralığın yarısında yapılan arama da bir ikili aramadır. Döngü yerine aralığı küçülterek kendini çağıran sürüm şöyledir:

```
int binsearch(int list[], int x, int l, int r) {
  int m;
  if (l > r) return -1;          /* aralik bos */
  m = (l + r) / 2;
  switch (COMPARE(list[m], x)) {
    case -1: return binsearch(list, x, m + 1, r);
    case 0:  return m;
    case 1:  return binsearch(list, x, l, m - 1);
  }
  return -1;                     /* ulasilmaz */
}
```

Kodu kısa tutmak için parametre adları kısaltılmıştır: `x` aranan değer, `l` ve `r` aralığın sol ve sağ ucu, `m` orta indistir. Fonksiyonun iki taban durumu vardır. Birincisi aralığın boş olmasıdır (*l > r*); değer bulunamamıştır ve *−1* döner. İkincisi `case 0` dalıdır; değer bulunmuştur ve indisi döner. Öteki iki dal özyinelemeli adımdır. Ortadaki öğe küçükse arama *[m + 1, r]* aralığında, büyükse *[l, m − 1]* aralığında sürer.

İki özyinelemeli çağrı da `return` ile başlar: alt çağrının sonucu hiç değiştirilmeden yukarı iletilir. Bu çağrılar birer kuyruk çağrısıdır. Fonksiyonun sonundaki `return -1;` satırına hiçbir zaman ulaşılmaz, çünkü `COMPARE` yalnız *−1*, 0 ya da 1 verir ve üç dal da `return` ile biter. Satırın yazılma nedeni derleyicidir: `switch`'in her olasılığı kapsadığını bilemeyen derleyici, değer döndürmeden biten bir yol görür ve uyarı verir. Bu satır o uyarıyı önler.

Sık yapılan hata

Özyinelemeli sürümde `if` yerine `while` yazmak. `while (l <= r)` diye başlayan bir gövdede her dal `return` ile çıktığı için döngü yalnız bir kez döner ve sonuç doğru çıkar. Yine de kod yanıltıcıdır: okuyan kişi bir döngü bekler, oysa tekrar özyinelemeyle sağlanır. Özyinelemeli sürümde aralık denetimi `if` ile yapılmalıdır.

Sık yapılan hata

Özyinelemeli çağrının sonucunu döndürmemek. `case -1: binsearch(list, x, m + 1, r); break;` yazılırsa alt çağrı değeri doğru bulur ama dönen değer atılır. Fonksiyon sonra `return -1;` satırına ulaşır ve aranan dizide olsa bile *−1* döndürür. Özyinelemeli çağrının sonucu ya döndürülmeli ya da bir değişkende kullanılmalıdır.

3.5.6 Kod izleme: özyinelemeli sürüm ve çağrı yığını

Özyinelemeli `binsearch(list, 18, 0, 9)` çağrısı, yinelemeli sürümle aynı orta indisleri (4, 1, 2) dener. Fark, her turun ayrı bir çerçevede yapılmasıdır. Tablo 3.5 çağrı yığınının değişimini gösterir.

Adım	Yığın (alttan üste)	`l`, `r`, `m`	Olay
1	main | bs(0, 9)	0, 9, 4	`list[4] = 24`
2	main | bs(0, 9)	0, 9, 4	*24 > 18*, `case 1:` `binsearch(list, 18, 0, 3)`
3	main | bs(0, 9) | bs(0, 3)	0, 3, 1	`list[1] = 13`
4	main | bs(0, 9) | bs(0, 3)	0, 3, 1	*13 < 18*, `case -1:` `binsearch(list, 18, 2, 3)`

Adım	Yığın (alttan üste)	`l, r, m`	Olay
5	main | bs(0, 9) | bs(0, 3) | bs(2, 3)	2, 3, 2	`list[2] = 18`
6	main | bs(0, 9) | bs(0, 3) | bs(2, 3)	2, 3, 2	`case 0`: `return 2`
7	main | bs(0, 9) | bs(0, 3)	0, 3, 1	2, `case -1` çerçevesinden olduğu gibi döner
8	main | bs(0, 9)	0, 9, 4	2, `case 1` çerçevesinden olduğu gibi döner
9	main	—	`main`'e 2 döner

Tablo 3.5: Özyinelemeli `binsearch(list, 18, 0, 9)` izlemesi.

Tabloda `bs(l, r)`, `binsearch(list, 18, l, r)` çağrısının kısaltmasıdır. İniş aşamasında her çağrı aralığı yarıya indirip yeni bir çerçeve açar. En derin noktada, adım 5 ve 6'da, yığında `main`'in üstünde üç `binsearch` çerçevesi vardır. Taban durum 2 döndürünce çıkış başlar. `bs(2, 3)`, 2'yi `case -1` dalında bekleyen `bs(0, 3)`'e döndürür. O da hiçbir işlem yapmadan 2'yi `case 1` dalında bekleyen `bs(0, 9)`'a döndürür. Son olarak 2, `main`'e ulaşır.



Şekil 3.7: Özyinelemeli ikili aramanın en derin anı. Sağdaki üç çerçeve *[0, 9]*, *[0, 3]* ve *[2, 3]* aralıklarını tutar; en üstteki çerçeve $m = 2$'de değeri bulur.

Çıkış aşamasında hiçbir hesap yapılmaz. Faktöriyelde dönüş yolunda çarpmalar vardı; burada dönen değer olduğu gibi aktarılır. Bu fark, iki fonksiyonun kuyruk özyinelemeli olup olmamasından gelir. Kuyruk çağrısı eniyilemesi açıksa derleyici bu üç çerçeveyi tek çerçeveye indirir ve kod, yinelemeli sürümle aynı biçimde çalışır.

Başarısız aramada derinlik bir fazla olabilir. `binsearch(list, 25, 0, 9)` çağrısı *[0, 9]*, *[5, 9]*, *[5, 6]* ve *[5, 4]* aralıklarıyla dört çerçeve açar. Son çerçeve boş aralığı görür ve hemen *−1* döndürür. Yinelemeli sürümde bu son denetim, döngü koşulunun yanlış çıkmasına karşılık gelir.

3.5.7 Karmaşıklık

İkili aramanın hızı, her turda aralığın en az yarıya inmesinden gelir. Aralıktaki öğe sayısı n'den başlayarak $n/2$, $n/4$, … diye küçülür ve en geç 1'e indiğinde son karşılaştırma yapılır.

> **Teorem 3.2 (İkili aramanın tur sınırı)**
>
> $n \geq 1$ öğeli sıralı bir dizide ikili arama, değer bulunsa da bulunmasa da en çok $\lfloor \log_2 n \rfloor + 1$ tur döner. Zaman maliyeti $O(\log n)$'dir.

İspat. Bir turun başında aralıkta s öğe olsun. Ortadaki öğe çıkarılınca solda $\lfloor (s-1)/2 \rfloor$, sağda $\lceil (s-1)/2 \rceil = \lfloor s/2 \rfloor$ öğe kalır. Hangi yarıya gidilirse gidilsin yeni aralık en çok $\lfloor s/2 \rfloor$ öğelidir. Başlangıçta $s = n$ olduğundan, k turdan sonra aralıkta en çok $\lfloor n/2^k \rfloor$ öğe kalır. $(k+1)$. turun yapılabilmesi için aralık boş olmamalıdır: $n/2^k \geq 1$, yani $k \leq \log_2 n$. Tur numarası $k+1$ en çok $\lfloor \log_2 n \rfloor + 1$ olabilir. Her turda sabit sayıda işlem yapıldığından zaman $O(\log n)$'dir. □

Aynı sonuç bir yineleme bağıntısıyla da elde edilir. $T(n)$, n öğeli bir aralıkta aramanın maliyeti olsun. Bir tur sabit bir c maliyetiyle aralığı yarıya indirir:

$$T(n) = T(n/2) + c,\quad T(1) = c$$

Bağıntı açılırsa $T(n) = T(n/4) + 2c = T(n/8) + 3c = \dots = T(n/2^k) + kc$ bulunur. $n/2^k = 1$ olduğunda, yani $k = \log_2 n$ iken, $T(n) = c(\log_2 n + 1)$ çıkar. Bu, $O(\log n)$'dir.

Logaritmik büyümenin ne demek olduğunu Tablo 3.6 gösterir. Doğrusal arama en kötü durumda dizideki her öğeye bakar; ikili arama ise yalnız birkaç öğeye.

n	İkili arama (en çok tur)	Doğrusal arama
10	4	10
1.000	10	1.000
1.000.000	20	1.000.000
1.000.000.000	30	1.000.000.000

Tablo 3.6: En kötü durumda tur (karşılaştırma) sayıları.

Test dizisinde $n = 10$'dur ve en çok $\lfloor \log_2 10 \rfloor + 1 = 3 + 1 = 4$ tur gerekir. 18 için üç tur yetti. Dört tur gerektiren bir örnek 40'tır: orta indisler 4, 7, 8 ve 9 sırasıyla denenir. Bu sayılar, test dizisinde 0'dan 41'e kadar bütün değerler aranarak doğrulanmıştır; hiçbir arama dört turu aşmaz. Bir milyon öğeli bir dizide de bütün başarılı ve başarısız aramalar denendiğinde en çok tur 20 çıkar.

Bellek maliyeti iki sürümde farklıdır. Yinelemeli sürüm yalnız birkaç değişken kullanır; ek bellek $O(1)$'dir. Özyinelemeli sürüm her tur için bir çerçeve açar; derinlik tur sayısı kadardır ve ek bellek $O(\log n)$'dir. Bir milyar öğede bu 30 çerçeve demektir; pratikte önemsizdir.

İkili aramanın iki ön koşulu vardır. Birincisi, dizi sıralı olmalıdır; sırasız bir dizide yarıyı atma kararı yanlış olur ve var olan değer bulunamayabilir. İkincisi, herhangi bir indisteki öğeye $O(1)$ zamanda erişilebilmelidir. Dizi bu koşulu sağlar: `list[middle]`, başlangıç adresine `middle` öğe boyu eklenerek tek adımda bulunur. Bağlı listede ise ortadaki düğüme ulaşmak için baştan itibaren düğüm düğüm ilerlemek gerekir. Her tur $O(n)$ adım alır ve ikili aramanın kazancı kaybolur.

Sık yapılan hata

Orta indisi çok büyük dizilerde `(left + right) / 2` ile hesaplamak. `int` 32 bit iken iki indisin toplamı $2^{31} - 1$'i aşabilir; bu işaretli taşmadır ve tanımsız davranıştır. Bu ancak bir milyardan fazla öğeli dizilerde olur, ama olduğunda bulunması zor bir hata çıkar. Güvenli yazım `left + (right - left) / 2` 'dir; fark hiçbir zaman taşmaz ve sonuç aynıdır.

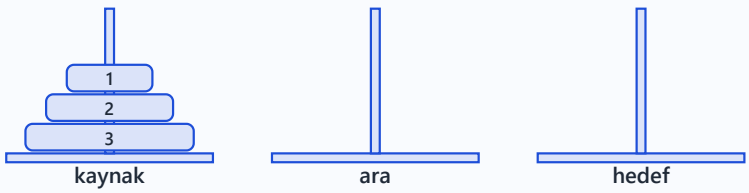
3.6 Hanoi kuleleri

Faktöriyel ve ikili arama, her adımda tek bir özyinelemeli çağrı yapar. Hanoi kuleleri, iki özyinelemeli çağrı yapan ilk örnektir. İki çağrı, çağrı sayısını hızla büyütür ve algoritmanın maliyetini hesaplamak için yeni bir araç gerektirir: yineleme bağıntısı.

3.6.1 Problem

Üç kule (çubuk) vardır: kaynak, ara ve hedef. Kaynak kulede n disk durur. Disklerin boyları farklıdır ve her disk, üzerindekinden büyüktür; en büyük disk en altta, en küçüğü en üsttedir. Amaç bütün diskleri, ara kuleden yardım alarak, hedef kuleye aynı düzende taşımaktır. İki kural vardır:

1. Her hamlede yalnız bir disk taşınır; bir kulenin en üstündeki disk alınıp başka bir kulenin en üstüne konur.
2. Bir disk kendinden küçük bir diskin üzerine konamaz.



Şekil 3.8: Üç diskli Hanoi kuleleri, başlangıç durumu. Diskler kaynak kulede, büyükten küçüğe dizilidir.

Bir ya da iki diskle problem kolaydır. Tek disk doğrudan hedefe taşınır. İki diskte küçük disk araya, büyük disk hedefe, küçük disk de hedefe taşınır: üç hamle. Üç diskte deneme yanılmayla çözüm bulunabilir, ama on diskte bu yol işe yaramaz. Disk sayısından bağımsız, sistemli bir yönteme ihtiyaç vardır.

3.6.2 Özyinelemeli çözüm

Çözümün anahtarı en büyük diske odaklanmaktır. En büyük disk bir noktada hedef kuleye taşınmalıdır. O anda en büyük diskin üstünde hiçbir disk olamaz, çünkü yalnız en üstteki disk taşınabilir. Hedef kule de boş olmalıdır, çünkü en büyük disk hiçbir diskin üzerine konamaz. Geriye kalan $n-1$ diskin hepsi o anda ara kulede durmalıdır. Bu gözlem çözümü üç adıma böler:

1. Üstteki $n-1$ disk, hedef kuleden yardım alınarak kaynaktan araya taşınır.
2. En büyük disk kaynaktan hedefe taşınır.
3. $n-1$ disk, kaynak kuleden yardım alınarak aradan hedefe taşınır.

Birinci ve üçüncü adım, aynı problemin $n-1$ diskli bir örneğidir; yalnız kulelerin rolleri değişmiştir. Bu adımlarda en büyük disk ya kaynakta ya da hedefte en altta durur. Her iki durumda da öteki bütün disklerden büyük olduğu için hiçbir hamleyi engellemez; sanki orada yokmuş gibi davranılabilir. Taban durum tek disktir ve bir hamleyle çözülür. Algoritmanın C kodu şöyledir:

```c
void hanoi(int n, char *kaynak, char *ara,
           char *hedef) {
  if (n == 1)
    printf("%s -> %s\n", kaynak, hedef);
  else {
    hanoi(n - 1, kaynak, hedef, ara);
    printf("%s -> %s\n", kaynak, hedef);
    hanoi(n - 1, ara, kaynak, hedef);
  }
}
/* main: hanoi(3, "A", "B", "C"); */
```

Kuleler adlarıyla (`char *` türünde dizgiler) temsil edilir. Kod diskleri bellekte gerçekten taşımaz; yalnız yapılacak hamleleri yazar. `printf("%s -> %s\n", kaynak, hedef)`, kaynak kulenin en üstündeki diskin hedef kuleye taşındığını bildirir. $n = 1$ ise tek hamle yazılır. Değilse ilk özyinelemeli çağrı `hanoi(n - 1, kaynak, hedef, ara)` yapılır. Bu çağrıda ikinci ve dördüncü argümanın yer değiştirdiğine dikkat etmek gerekir: $n-1$ diskin hedefi asıl problemin ara kulesidir, yardımcı kule de asıl hedeftir. Ardından en büyük diskin hamlesi yazılır. İkinci çağrı `hanoi(n - 1, ara, kaynak, hedef)`, $n-1$ diski aradan hedefe, kaynağı yardımcı kullanarak taşır.

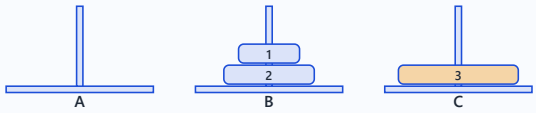
3.6.3 Kod izleme: hanoi(3, "A", "B", "C")

Üç diskli problemde A kaynak, B ara, C hedeftir. Çağrıların sırası ve yazılan hamleler Tablo 3.7'de verilmiştir. Tablodaki çağrılar `hanoi(n, kaynak, ara, hedef)` biçimindedir.

Adım	Etkin çağrı	Yığın derinliği	Olay	Yazılan
1	`hanoi(3, A, B, C)`	1	2 disk A → B için `hanoi(2, A, C, B)` çağrılır	
2	`hanoi(2, A, C, B)`	2	1 disk A → C için `hanoi(1, A, B, C)` çağrılır	
3	`hanoi(1, A, B, C)`	3	taban durum	`A -> C`
4	`hanoi(2, A, C, B)`	2	disk 2 taşınır; sonra `hanoi(1, C, A, B)`	`A -> B`
5	`hanoi(1, C, A, B)`	3	taban durum	`C -> B`
6	`hanoi(3, A, B, C)`	1	disk 3 taşınır; sonra `hanoi(2, B, A, C)`	`A -> C`
7	`hanoi(2, B, A, C)`	2	1 disk B → A için `hanoi(1, B, C, A)`	
8	`hanoi(1, B, C, A)`	3	taban durum	`B -> A`
9	`hanoi(2, B, A, C)`	2	disk 2 taşınır; sonra `hanoi(1, A, B, C)`	`B -> C`
10	`hanoi(1, A, B, C)`	3	taban durum	`A -> C`

Tablo 3.7: `hanoi(3, "A", "B", "C")` izlemesi.

Yığın derinliği `main` sayılmadan verilmiştir. Program yedi satır yazar: `A -> C`, `A -> B`, `C -> B`, `A -> C`, `B -> A`, `B -> C`, `A -> C`. İlk üç hamle küçük iki diski B'ye taşır. Dördüncü hamle en büyük diski C'ye götürür. Son üç hamle B'deki iki diski C'ye, en büyük diskin üstüne taşır. Her hamle kurala uygundur; bu, programın çıktısı adım adım tahtada oynanarak da doğrulanabilir.

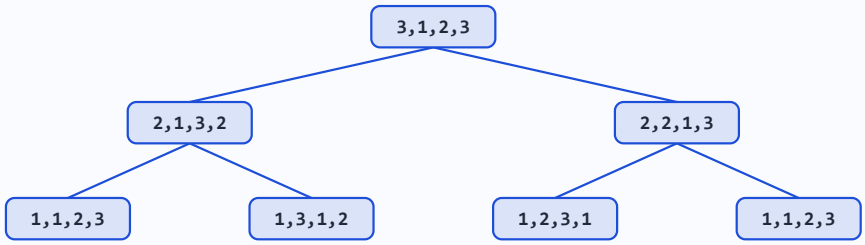


Şekil 3.9: Dördüncü hamleden sonraki durum (adım 6): disk 1 ve 2 ara kulede (B), en büyük disk 3 hedef kulede (C).

Tabloda iki sayı dikkat çeker. Toplam çağrı sayısı yedidir ve her çağrı tam bir `printf` yapar; taban durumlar birer, öteki çağrılar ortadaki birer hamleyi yazar. Yığın derinliği ise hiçbir zaman 3'ü geçmez. *n* diskli problemde bir çağrı zinciri *n, n − 1, …, 1* diye iner; derinlik *n*'dir ve yığın maliyeti *O(n)*'dir.

3.6.4 Çağrı ağacı

Bir özyinelemeli algoritmanın bütün çağrıları bir ağaç olarak çizilebilir. Her düğüm bir çağrıdır; bir çağrının yaptığı alt çağrılar onun çocuklarıdır. Hanoi'de her çağrının iki çocuğu vardır: sol çocuk ilk özyinelemeli çağrı, sağ çocuk ikincisidir. Şekil 3.10, kuleleri 1, 2 ve 3 diye adlandırılmış `hanoi(3, "1", "2", "3")` çağrısının ağacını verir. Her kutuda sırasıyla `n`, kaynak, ara ve hedef yazılıdır.



Şekil 3.10: `hanoi(3, "1", "2", "3")` çağrı ağacı. Her kutu bir çağrının çerçevesidir: `n, kaynak, ara, hedef`.

Ağaç, çağrı yığınının bütün tarihçesini taşır. Herhangi bir anda yığının içeriği, kökten o an çalışan düğüme giden yoldur. Örneğin en soldaki yaprak (1, 1, 2, 3) çalışırken yığında kök (3, 1, 2, 3), onun sol çocuğu (2, 1, 3, 2) ve yaprağın kendisi vardır. Yolun uzunluğu en çok ağacın yüksekliği kadardır; bu da yığın derinliğinin *n* olduğunu bir kez daha gösterir.

Hamlelerin sırası, ağacın **kök ortada** (*inorder*) gezinmesiyle elde edilir: önce sol alt ağaç, sonra düğümün kendisi, sonra sağ alt ağaç. Kodda da sıra böyledir: ilk özyinelemeli çağrı, `printf`, ikinci özyinelemeli çağrı. Her düğümün hamlesi "kaynak → hedef"tir ve sol alt ağacı bittikten sonra yazılır. Bu gezinme şu yedi hamleyi verir:

> 1→3 · 1→2 · 3→2 · 1→3 · 2→1 · 2→3 · 1→3

Program aynı çağrıyla çalıştırıldığında da bu sıra çıkar.

3.6.5 Yineleme bağıntısı: kaç hamle?

n diskli problemin hamle sayısı *h(n)* olsun. Algoritma *n − 1* diski iki kez taşır ve arada bir hamle yapar. Taban durum tek hamledir:

> $h(1) = 1,\ h(n) = h(n-1) + 1 + h(n-1) = 2\cdot h(n-1) + 1$

Tanım 3.6

Yineleme bağıntısı (*recurrence relation*), bir fonksiyonun değerini aynı fonksiyonun daha küçük girdilerdeki değerleriyle ifade eden eşitliktir. Bağıntıyı **çözmek**, fonksiyonu kendine başvurmayan kapalı bir formülle yazmaktır.

Özyinelemeli algoritmaların maliyeti doğal olarak yineleme bağıntısıyla ifade edilir, çünkü algoritmanın kendisi de aynı biçimdedir. En basit çözüm yöntemi bağıntıyı tekrar tekrar yerine koyarak açmaktır:

```
h(1) = 1
h(n) = 2·h(n−1) + 1
     = 4·h(n−2) + 2 + 1
     = 8·h(n−3) + 4 + 2 + 1
     = …
     = 2^{n−1}·h(1) + 2^{n−2} + … + 1
     = 2^n − 1
```

Her satırda $h(n-k)$ yerine $2\cdot h(n-k-1)+1$ konur. k adım sonra katsayı 2^k olur ve sağda $2^{k-1}+\dots+2+1$ toplamı birikir. $k=n-1$'de taban duruma ulaşılır: $h(n)=2^{n-1}\cdot h(1)+2^{n-2}+\dots+2+1$. $h(1)=1$ olduğundan bu, $2^{n-1}+2^{n-2}+\dots+1$ geometrik toplamıdır ve 2^n-1'e eşittir. Toplamın değeri ikili sayılarla kolayca görülür: n tane 1 bitinden oluşan sayı, 2^n'nin bir eksiğidir.

> **Teorem 3.3 (Hanoi hamle sayısı)**
>
> Özyinelemeli algoritma n diski 2^n-1 hamlede taşır. Kurallara uyan hiçbir yöntem daha az hamleyle bitiremez.

İspat. *Hamle sayısı.* Tümevarım: $n=1$ için $h(1)=1=2^1-1$. $h(n-1)=2^{n-1}-1$ kabul edilirse $h(n)=2(2^{n-1}-1)+1=2^n-1$ olur.

Alt sınır. $m(n)$, n diski taşımak için gereken en az hamle sayısı olsun; $m(1)=1$'dir. En büyük disk en az bir kez taşınmalıdır. İlk taşınışından hemen önce öteki $n-1$ diskin hepsi üçüncü kulede toplanmış olmalıdır; bu en az $m(n-1)$ hamle ister. En büyük diskin son taşınışından sonra o $n-1$ disk hedefe, onun üstüne getirilmelidir; bu da en az $m(n-1)$ hamle ister. Dolayısıyla $m(n)\ge 2\cdot m(n-1)+1$'dir ve tümevarımla $m(n)\ge 2^n-1$ bulunur. Algoritma tam bu sayıda hamle yaptığı için en iyidir. □

n	Hamle
1	1
2	3
3	7
4	15
10	1.023
20	1.048.575
64	18.446.744.073.709.551.615

Tablo 3.8: Disk sayısına göre hamle sayısı 2^n-1.

Tablo 3.8 üstel büyümenin hızını gösterir. Her yeni disk hamle sayısını iki katına çıkarır. Efsaneye göre bir tapınakta rahipler 64 diskli kuleyi taşıyor ve iş bitince dünya sona erecektir. $2^{64}-1\approx 1{,}8\cdot 10^{19}$ hamle, saniyede bir hamleyle yaklaşık 585 milyar yıl sürer. Bu süre evrenin bugünkü yaşının kırk katından fazladır.

Algoritmanın zaman maliyeti $\Theta(2^n)$'dir. Çağrı sayısı da hamle sayısına eşittir, çünkü her çağrı tam bir hamle yazar ve $c(n)=2c(n-1)+1$ bağıntısını sağlar. Yığın maliyeti $O(n)$'dir. Teorem gereği bundan iyisi yoktur: daha akıllı bir program yazmak hamle sayısını azaltamaz, çünkü bu sayı problemin kendisinden gelir. Hanoi, çıktısının boyu üstel olan bir problemdir ve hiçbir algoritma çıktıyı yazmadan bitemez.

Kimi problemlerin ancak özyinelemeyle çözülebileceği söylenir. Bu sözün kuramsal karşılığı tam olarak böyle değildir: her özyineleme, açık bir yığınla döngüye çevrilebilir. Sözün asıl vurgusu, Hanoi gibi problemlerde özyinelemenin tek doğal ifade olmasıdır. Hanoi'nin döngülü çözümleri de vardır, ama hiçbiri yukarıdaki üç satırlık fikir kadar açık değildir.

3.7 Permütasyon üretimi

Hanoi'de çıktı üstel boydaydı. Permütasyon üretiminde çıktı daha da hızlı büyür: n öğenin $n!$ dizilişi vardır. Bu problem, özyinelemeyi ilk kez bir döngünün içinde kullanır ve geri izlemenin temel fikrini, yapılan seçimi geri almayı, tanıtır.

3.7.1 Problem ve özyinelemeli fikir

Tanım 3.7

Bir kümenin öğelerinin belli bir sırayla dizilişine **permütasyon** (*permutation*) denir. $n \geq 1$ öğeli bir kümenin $n! = n \cdot (n-1) \cdot \ldots \cdot 1$ farklı permütasyonu vardır. Permütasyon problemi, bu dizilişlerin hepsini yazdırmaktır.

n! sayısı basit bir sayma kuralından gelir. Birinci konuma *n* öğeden herhangi biri gelebilir. İkinci konuma kalan $n - 1$ öğeden biri, üçüncüye kalan $n - 2$ öğeden biri gelir. Seçenekler çarpılır: $n \cdot (n-1) \cdot \ldots \cdot 1$. Örneğin {a, b, c} kümesinin $3! = 6$ dizilişi vardır: abc, acb, bac, bca, cab, cba.

Aynı sayma kuralı özyinelemeli bir algoritma verir. {a, b, c, d} kümesinin dizilişleri dört gruba ayrılır:

- a ile başlayan ve ardından {b, c, d}'nin bir dizilişi gelenler,
- b ile başlayan ve ardından {a, c, d}'nin bir dizilişi gelenler,
- c ile başlayan ve ardından {a, b, d}'nin bir dizilişi gelenler,
- d ile başlayan ve ardından {a, b, c}'nin bir dizilişi gelenler.

Her grup, üç öğeli bir kümenin dizilişlerini ister. Üç öğeli problem aynı biçimde iki öğeli problemlere, onlar da tek öğeli problemlere iner. Tek öğenin tek dizilişi vardır; bu taban durumdur.

3.7.2 Takasla gerçekleme: perm()

Fikri koda dökmek için “kalan öğeler” kümesini ayrı bir dizide tutmak gerekmez. Bütün öğeler tek bir karakter dizisinde durur. `i` indisi, dizinin hangi konumunun şu an doldurulduğunu gösterir. `list[0..i-1]` konumları sabitlenmiş önektir; `list[i..n-1]` konumlarında henüz yerleştirilmemiş öğeler bulunur. `i`. konuma sırayla her aday **takasla** getirilir, kalan konumlar özyinelemeyle dizilir, sonra **geri takasla** dizi eski hâline döndürülür.

Takas için üç argümanlı bir makro kullanılır:

```
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
```

Makro, `t` geçici değişkeni üzerinden `x` ile `y`'nin değerlerini değiştirir. Üç atama virgül işleciyle tek bir ifadede birleştirilmiştir; virgül işleci ifadeleri soldan sağa sırayla hesaplar. Makronun fonksiyon yerine tercih edilme nedeni, C'de bir fonksiyonun çağıranın değişkenlerini değiştirmek için adres (işaretçi) alması gerekmesidir. Makro ise önişlemcide metin olarak yerine konur ve doğrudan `list[i]` ile `list[j]` üzerinde çalışır. Permütasyon fonksiyonu şöyledir:

```
void perm(char *list, int i, int n) {
  int j, temp;
  if (i == n) {                  /* bir dizilis */
    for (j = 0; j < n; j++) printf("%c", list[j]);
    printf("\n");
  } else
    for (j = i; j < n; j++) {
      SWAP(list[i], list[j], temp);
      perm(list, i + 1, n);
      SWAP(list[i], list[j], temp);
    }
}
```

`perm(list, i, n)`, `list` dizisinin ilk `i` konumu sabitken kalan konumların bütün dizilişlerini yazar. `i == n` taban durumdur: bütün konumlar sabitlenmiştir ve dizi bir permütasyondur. Bu durumda ilk `for` döngüsü `n` karakteri yazar, ardından satır sonu gelir. Taban durumda değilse ikinci `for` döngüsü `j`'yi `i`'den `n - 1`'e kadar götürür. Her turda üç iş yapılır:

1. `SWAP(list[i], list[j], temp)`: `j`. konumdaki öğe `i`. konuma getirilir. *j = i* iken öğe kendisiyle takas edilir ve dizi değişmez.
2. `perm(list, i + 1, n)`: `i`. konum artık sabittir; kalan konumların bütün dizilişleri yazılır.
3. `SWAP(list[i], list[j], temp)`: aynı takas tekrarlanır ve dizi, döngü turunun başındaki hâline döner.

`temp` değişkeni `int` olarak bildirilmiştir; içine `char` konup yeniden `char`'a atanır. Bu dönüşüm değer kaybettirmez, çünkü her `char` değeri `int`'e sığar. Fonksiyonun ilk çağrısı, bütün diziyi düzenlemek için `perm(list, 0, n)` biçimindedir. `list` değiştirilebilir bir dizi olmalıdır: `char s[] = "abc";` bildirimi böyle bir dizi oluşturur, ama `perm("abc", 0, 3)` biçiminde bir dizgi sabiti vermek tanımsız davranıştır. Dizgi sabitleri salt okunur bellekte durabilir ve takas programı çökertebilir.

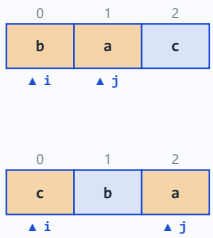
3.7.3 Kod izleme: perm("abc", 0, 3)

`char list[] = "abc"` dizisiyle yapılan `perm(list, 0, 3)` çağrısı altı diziliş yazar. Tablo 3.9 izlemenin bütün adımlarını verir. Yığın sütununda `p(i)`, `perm(list, i, 3)` çağrısının kısaltmasıdır.

Adım	Yığın	`i, j`	İşlem	`list`	Yazılan
1	p(0)	0, 0	kendisiyle takas, `perm(i = 1)`	abc	
2	p(0) p(1)	1, 1	kendisiyle takas, `perm(i = 2)`	abc	
3	p(0) p(1) p(2) p(3)	3, —	taban durum	abc	abc
4	p(0) p(1)	1, 2	`list[1] ⇔ list[2]`, `perm(i = 2)`	acb	
5	p(0) p(1) p(2) p(3)	3, —	taban durum	acb	acb
6	p(0) p(1)	1, 2	geri takas `list[1] ⇔ list[2]`	abc	
7	p(0)	0, 1	`list[0] ⇔ list[1]`, `perm(i = 1)`	bac	
8	p(0) p(1)	1, 1	kendisiyle takas, `perm(i = 2)`	bac	
9	p(0) p(1) p(2) p(3)	3, —	taban durum	bac	bac
10	p(0) p(1)	1, 2	`list[1] ⇔ list[2]`, `perm(i = 2)`	bca	
11	p(0) p(1) p(2) p(3)	3, —	taban durum	bca	bca
12	p(0) p(1)	1, 2	geri takas `list[1] ⇔ list[2]`	bac	
13	p(0)	0, 1	geri takas `list[0] ⇔ list[1]`	abc	
14	p(0)	0, 2	`list[0] ⇔ list[2]`, `perm(i = 1)`	cba	
15	p(0) p(1)	1, 1	kendisiyle takas, `perm(i = 2)`	cba	
16	p(0) p(1) p(2) p(3)	3, —	taban durum	cba	cba
17	p(0) p(1)	1, 2	`list[1] ⇔ list[2]`, `perm(i = 2)`	cab	
18	p(0) p(1) p(2) p(3)	3, —	taban durum	cab	cab
19	p(0) p(1)	1, 2	geri takas `list[1] ⇔ list[2]`	cba	
20	p(0)	0, 2	geri takas `list[0] ⇔ list[2]`	abc	

Tablo 3.9: `perm("abc", 0, 3)` izlemesi.

Tabloda `p(2)` çağrısının kendi döngüsü kısaltılmıştır. *i = 2* iken `j` yalnız 2 değerini alır; kendisiyle takas yapılır, `p(3)` çağrılır ve geri takas yine kendisiyle olur. Bu adımlar diziyi değiştirmez.



Şekil 3.11: Dış döngünün iki takası. (a) *i = 0, j = 1*: a ile b yer değiştirir, dizi `bac` olur (adım 7). (b) *i = 0, j = 2*: a ile c yer değiştirir, dizi `cba` olur (adım 14).

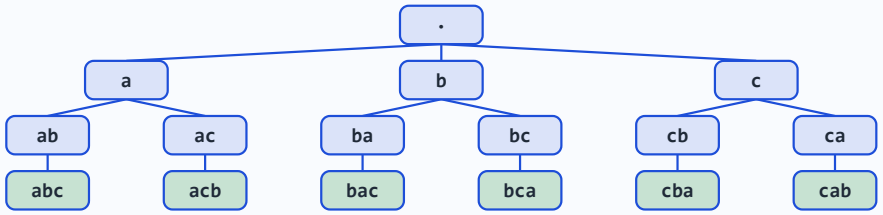
Çıktı sırası abc, acb, bac, bca, cba, cab'dir. Son iki satır sözlük sırasına uymaz: cab'den önce cba gelir. Bunun nedeni adım 14'teki takastır. *j = 2* turunda a ile c yer değiştirir ve dizi `cba` olur; ikinci konumda b, üçüncüde a durur. Alt çağrı bu sırayla başladığı için önce cba yazılır. Dizilişlerin hepsi tam bir kez çıkar; yalnız sıraları farklıdır.

Adım 13 ve 20'deki geri takaslar algoritmanın doğruluğu için şarttır. Dış döngünün her turu, diziyi `abc` hâlinde bulmayı bekler. Geri takaslar hiç yapılmasaydı *j = 1* turu diziyi `cba` olarak bırakırdı ve *j = 2* turu `list[0]` ile `list[2]` 'yi, yani c ile a'yı takas ederdi. Bu durumda b ile başlayan dizilişler hiç üretilmez, a ile başlayanlar ikinci kez üretilirdi. Program çalıştırılıp çağrıdan sonra dizi yazdırıldığında `abc` görülür: her çağrı, diziyi aldığı gibi bırakır.

> **Sık yapılan hata**
>
> İkinci `SWAP` 'ı unutmak. Kod derlenir ve altı satır yazar, ama bazı dizilişler tekrar eder, bazıları hiç çıkmaz. Hata küçük girdilerde gözden kaçabilir. Geri izleme yapan her algoritmada, özyinelemeli çağrıdan önce yapılan değişiklik çağrıdan sonra geri alınmalıdır.

3.7.4 Özyineleme ağacı ve karmaşıklık

`perm` 'in çağrıları bir ağaç oluşturur. Kökte hiçbir konum sabit değildir. Her düzey bir konumu daha sabitler ve düğümün çocukları, o konuma gelebilecek öğelerdir. Yapraklar tamamlanmış dizilişlerdir.



Şekil 3.12: `perm("abc", 0, 3)` özyineleme ağacı. Kökte sabitlenmiş önek yoktur; her düzey bir konumu sabitler. Yapraklar, çıktıdaki sırayla altı dizilişi verir.

Şekil 3.12'deki ağaçta kökün üç çocuğu (a, b, c), her birinin iki çocuğu (ab, ac, …), onların da birer çocuğu vardır. Kardeş dallar aynı başlangıç dizisini görmelidir; ikinci `SWAP` diziyi bu yüzden onarır. Ağacın her düzeyindeki düğüm sayısı çağrı sayısını verir. *k.* düzeyde ilk *k* konum sabittir ve bu konumlar $n \cdot (n-1) \cdot \ldots \cdot (n-k+1)$ farklı biçimde doldurulabilir:

> çağrı sayısı = 1 + n + n(n − 1) + … + n!

n = 3 için *1 + 3 + 6 + 6 = 16*, *n = 4* için *1 + 4 + 12 + 24 + 24 = 65* çağrı yapılır. Bu sayılar, fonksiyona bir sayaç eklenerek programla da doğrulanmıştır. Toplamın son iki terimi *n!*'dir ve öncekiler hızla küçülür; toplam yaklaşık $e \cdot n! \approx 2{,}72 \cdot n!$'dir. Yaprak sayısı tam *n!*'dir.

Zamanı belirleyen yapraklardaki yazma işidir. Her yaprak *n* karakter yazar; *n!* yaprak toplam $n \cdot n!$ karakter eder. İç düğümlerde yapılan işler (takaslar ve döngü denetimleri) çağrı başına sabittir ve toplamı $O(n!)$'dir. Toplam zaman $\Theta(n \cdot n!)$'dir. Permütasyon problemi kimi kaynaklarda kısaca $O(n!)$ diye anılır; bu yazım, diziliş sayısını anlatır ama her dizilişi yazmanın *n* çarpanını gözden kaçırır. Yığın maliyeti derinlik kadardır: *n + 1* çerçeve, yani $O(n)$.

Faktöriyel büyümesi üstel büyümeden de hızlıdır. *n = 10*'da *10! = 3.628.800* diziliş vardır ve yazılacak karakter sayısı 36 milyonu geçer. *n = 20*'de diziliş sayısı $2{,}4 \cdot 10^{18}$'dir; saniyede bir milyar diziliş üreten bir makine bile bu işi 77 yılda bitirir. Bütün dizilişleri sırayla denemeye dayanan bir çözüm bu yüzden ancak çok küçük *n* için kullanılabilir.

3.8 Kaba kuvvet ve geri izleme

Permütasyon ve alt küme üretimi, bir problemin bütün olası çözüm adayları üzerinde sistemli biçimde gezinmenin yollarıdır. Pek çok problem bu gezinmeyle çözülebilir: adaylar üretilir, her biri kurallara göre sınanır. Bu bölümün geri kalanı, bu yaklaşımı ve onun akıllı biçimini, geri izlemeyi, inceler.

3.8.1 İki yaklaşım

Tanım 3.8

Kaba kuvvet (*brute force*) yaklaşımında olası bütün aday çözümler üretilir ve her biri ayrı ayrı sınanır. **Geri izleme** (*backtracking*) yaklaşımında çözüm adım adım kurulur. Kısmi çözüm bir kuralı bozduğu anda o daldaki bütün adaylar **budanır**: son seçim geri alınır ve sıradaki seçenek denenir.

Kaba kuvvet bilgisayarın hızına güvenir. Kurması kolaydır ve her zaman doğru cevabı verir, çünkü hiçbir adayı atlamaz. Sorunu aday sayısıdır. n evet/hayır kararından oluşan bir problemde 2^n, n öğenin sıralanmasını isteyen bir problemde $n!$ aday vardır. Her ikisi de n birkaç düzineyi geçince hiçbir makinenin bitiremeyeceği sayılara ulaşır.

Geri izleme aynı adaylar üzerinde gezinir, ama bir farkla: bir aday tamamlanmadan önce sınanır. Kısmi bir çözümün kuralı bozduğu görülürse, o kısmi çözümle başlayan adayların hiçbiri üretilmez. Örneğin sekiz veziri yerleştirirken ilk iki vezir birbirini tehdit ediyorsa, öteki altı vezirin bütün yerleşimleri tek seferde elenir. Bu eleme işlemi ağacın bir dalını kesmeye benzediği için budama adını alır. En kötü durum maliyeti çoğu zaman kaba kuvvetle aynı mertebede kalır, ama pratikte denenen aday sayısı çok küçülür.

Bir problemi çözerken seçilen veri yapısından ve yöntemden asla tam emin olunmamalıdır. Kaba kuvvetle çalışan bir çözüm, daha iyisinin aranmasını gerektiren bir başlangıç noktasıdır. Bu tutum, “daha iyi bir çözüm yöntemi vardır da biz bulamıyoruz” cümlesiyle özetlenebilir. Geri izleme ve at gezisindeki sezgisel kural, böyle bir aramanın ürünleridir.

3.8.2 Geri izlemenin genel kalıbı

Geri izleme algoritmalarının hepsi aynı iskeleti izler. Aşağıdaki sözde kod bu iskeleti verir:

```
void geri_izle(kismi_cozum) {
  if (tam_cozum_mu) { yaz; return; }
  for (her aday secim)
    if (kurallara_uygun) { sec; geri_izle(...); secimi_geri_al; }
}
```

Fonksiyon bir kısmi çözüm alır. Kısmi çözüm tamamlanmışsa yazılır ve fonksiyon döner; bu taban durumdur. Tamamlanmamışsa kısmi çözümü bir adım uzatabilecek her aday seçim sırayla denenir. Kurallara uyan seçim kısmi çözüme eklenir (`sec`), fonksiyon kendini çağırır ve çağrı dönünce seçim geri alınır (`secimi_geri_al`). Kurallara uymayan seçimler için özyinelemeli çağrı hiç yapılmaz; budama budur.

Permütasyon kodu bu kalıba tam uyar. Kısmi çözüm sabitlenmiş önektir. Aday seçimler `list[i..n-1]` öğeleridir. Seçmek ilk `SWAP` 'tır, seçimi geri almak ikinci `SWAP` 'tır. Permütasyonda her seçim kurallara uyar, dolayısıyla budama yoktur. n-vezir probleminde ise `kurallara_uygun` sınaması gerçek bir iş yapar ve dalların büyük kısmını keser.

Geri izleme kimi zaman “senaryoyu sondan başa geri sarmak” diye yanlış anlaşılır. Yöntem çözümü sondan başa kurmaz. Kısmi çözümü ileriye doğru kurar; çıkmaza girince yalnız son seçimi geri alır ve aynı noktadan ileriye doğru başka bir seçimle devam eder. Geri gidilen tek şey son karar noktasıdır.

3.8.3 Alt küme üretimi

Bir kümenin bütün alt kümelerini yazdırmak, en sade geri izleme örneğidir. n öğeli bir kümenin 2^n alt kümesi vardır, çünkü her öğe için iki seçenek bulunur: alt kümeye alınır ya da alınmaz. Algoritma bu kararları sırayla verir:

```c
void altkume(char *s, int n, int i,
             int sec[]) {
  int j;
  if (i == n) {                  /* karar bitti */
    printf("{");
    for (j = 0; j < n; j++)
      if (sec[j]) printf("%c", s[j]);
    printf("}\n");
    return;
  }
  sec[i] = 1; altkume(s, n, i + 1, sec);
  sec[i] = 0; altkume(s, n, i + 1, sec);
}
```

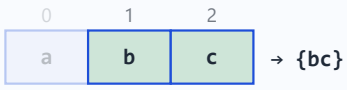
`s` öğeleri tutan karakter dizisi, `n` öğe sayısıdır. `sec` dizisi kararları tutar: `sec[j]` 1 ise `j`. öğe alınmış, 0 ise alınmamıştır. `i`, sıradaki kararın hangi öğe için verileceğini gösterir. `i == n` taban durumdur: bütün kararlar verilmiştir ve `sec` dizisinde 1 olan öğeler küme parantezleri içinde yazılır. Taban durumda değilse iki çağrı yapılır. Önce `sec[i] = 1` ile `i`. öğe alınır ve kalan kararlar için fonksiyon çağrılır. Sonra `sec[i] = 0` ile öğe çıkarılır ve aynı çağrı tekrarlanır.

Kodda "seçimi geri al" adımı ayrı bir satır olarak görünmez. `sec[i] = 0` ataması iki işi birden yapar: birinci dalın seçimini geri alır ve ikinci dalın seçimini yapar. Fonksiyondan çıkarken `sec[i]` 0 kalır, ama bunun bir zararı yoktur. Bir üst çağrı, `sec[i]`'yi okumadan önce her zaman yeniden atar; taban durumdaki yazdırma ise yalnız o ana kadar atanmış değerleri okur.

Sıra	a	b	c	`sec`	Yazılan
1	al	al	al	[1, 1, 1]	{abc}
2	al	al	alma	[1, 1, 0]	{ab}
3	al	alma	al	[1, 0, 1]	{ac}
4	al	alma	alma	[1, 0, 0]	{a}
5	alma	al	al	[0, 1, 1]	{bc}
6	alma	al	alma	[0, 1, 0]	{b}
7	alma	alma	al	[0, 0, 1]	{c}
8	alma	alma	alma	[0, 0, 0]	{}

Tablo 3.10: `altkume("abc", 3, 0, sec)` çıktısı ve kararlar.

Tablo 3.10'da kararlar, bir ikili sayacın geriye doğru saymasına benzer: `sec` dizisi 111, 110, 101, ..., 000 diye ilerler. Bunun nedeni "al" kararının "alma"dan önce denenmesidir. Son satırdaki `{}` boş kümedir; boş küme de her kümenin bir alt kümesidir.



Şekil 3.13: Beşinci yaprak: a alınmamış, b ve c alınmıştır; `{bc}` yazılır.

Karmaşıklık hesabı Hanoi'ye benzer. Her çağrı iki çağrı yapar ve derinlik n'dir. Çağrı ağacı $n + 1$ düzeyli mükemmel bir ikili ağaçtır: 2^n yaprağı ve toplam $2^{n+1} - 1$ düğümü vardır. $n = 3$ için 15, $n = 4$ için 31 çağrı yapılır. Her yaprak `sec` dizisinde baştan sona gezinerek $O(n)$ iş yapar. Toplam zaman $\Theta(n \cdot 2^n)$, yığın maliyeti $O(n)$'dir. `sec` dizisini çağıran fonksiyon bir kez ayırır ve bütün çağrılar aynı diziyi kullanır; her çerçevede yalnız diziye işaretçi durur.

3.9 Sağlanabilirlik (SAT) problemi

Alt küme üretimi, her öğe için iki seçenek üzerinde geziniyordu. Aynı gezinme, mantık değişkenlerine doğru ya da yanlış değeri atamaya da uygulanabilir. Bu bölümün problemi, bilgisayar biliminin en ünlü problemlerinden biridir ve kaba kuvvetin sınırlarını açıkça gösterir.

3.9.1 Problem

Tanım 3.9

$x_1, x_2, \ldots, x_n$ doğru (T) ya da yanlış (F) değer alan mantık değişkenleri olsun. Bu değişkenlerden ∧ (ve), ∨ (veya) ve ¬ (değil) işleçleriyle bir formül kurulur. **Sağlanabilirlik problemi** (*satisfiability*, kısaca SAT) şunu sorar: değişkenlere, formülü doğru yapan bir değer ataması var mı?

Bu bölümde incelenen formül şudur:

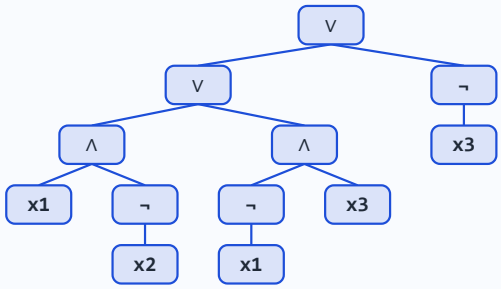
$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$

Formül üç terimin veyasıdır ve terimlerden biri doğruysa formül doğrudur. Örneğin $x_3 = F$ ise son terim $\neg x_3$ doğrudur ve formül de doğrudur. Böylece formülün sağlanabilir olduğu hemen görülür. Soru, bir bilgisayar programının bunu nasıl bulacağıdır.

Kaba kuvvet çözümü bütün atamaları denemektir. Üç değişkenin her biri iki değer aldığından $2^3 = 8$ atama vardır: {F, F, F}, {F, F, T}, …, {T, T, T}. n değişkende atama sayısı 2^n'dir. Her atamada formül bir kez değerlendirilir; formülün boyu $|F|$ ise bu $O(|F|)$ zaman alır. Toplam maliyet $O(2^n \cdot |F|)$'dir.

Üstel maliyet paralel donanımla da aşılamaz. İş m işlemciye eşit bölünse bile maliyet $O(2^n/m)$ olur. 100 değişkenli bir formülde $2^{100} \approx 1{,}3 \cdot 10^{30}$ atama vardır; bir milyon işlemci bile bu sayıyı ancak 10^{24} mertebesine indirir.

SAT için polinom zamanlı bir algoritma bilinmiyor. Problem, **NP-tam** (*NP-complete*) denen problem sınıfının ilk üyesidir. Bu sınıftaki problemlerden biri için polinom zamanlı bir algoritma bulunursa, sınıftaki bütün problemler polinom zamanda çözülebilir. Bugüne kadar böyle bir algoritma bulunamamıştır, ama olmadığı da kanıtlanamamıştır. Her alt cümlesinde (*clause*) en çok iki değişken bulunan özel biçim (2-SAT) polinom zamanda çözülür; alt cümleler üç değişkenli olunca (3-SAT) problem yeniden NP-tam olur. Problemlerin bu biçimde sınıflandırılması hesaplama kuramı (*theory of computation*) dersinin konusudur.



Şekil 3.14: $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$ formülünün ağacı. İç düğümler işleçler, yapraklar değişkenlerdir.

3.9.2 Formül ağacı ve kök sonda değerlendirme

Bir formülü programda değerlendirmenin doğal yolu onu bir ağaç olarak saklamaktır. Şekil 3.14'teki ağaçta her iç düğüm bir işleçtir. ∧ ve ∨ düğümlerinin iki çocuğu vardır. ¬ düğümünün tek işleneni vardır ve sağ çocukta durur. Yapraklar değişkenlerdir; aynı değişken birden çok yaprakta görünebilir. Ağacın C'deki tanımı ve değerlendirme fonksiyonu şöyledir:

```
typedef enum { DEGIL, VE, VEYA, DOGRU, YANLIS } mantik;
typedef struct agac *agac_gost;
struct agac {
  agac_gost solcocuk, sagcocuk;
  mantik veri;
  short deger;
};

void kok_sonda_degerlendir(agac_gost L) {
  if (L) {
    kok_sonda_degerlendir(L->solcocuk);
    kok_sonda_degerlendir(L->sagcocuk);
    switch (L->veri) {
      case DEGIL: L->deger = !L->sagcocuk->deger; break;
      case VE:   L->deger = L->solcocuk->deger
                         && L->sagcocuk->deger; break;
      case VEYA: L->deger = L->solcocuk->deger
                         || L->sagcocuk->deger; break;
      case DOGRU:  L->deger = 1; break;
      case YANLIS: L->deger = 0; break;
    }
  }
}
```

İlk satır, düğüm türlerini adlandıran bir sayım tipi (*enumeration*) tanımlar. `DEGIL`, `VE`, `VEYA` işleç düğümleridir. `DOGRU` ve `YANLIS` yapraklar içindir: bir atama yapıldığında her yaprağın türü, değişkenin o atamadaki değerine göre bu ikisinden biri olur. İkinci satır `agac_gost` adını "`struct agac`'a işaretçi" anlamında tanımlar. Yapının kendisi üçüncü bildirimdedir: iki çocuk işaretçisi, düğümün türü (`veri`) ve düğümün hesaplanan değeri (`deger`).

`typedef`'in yapı tanımından önce gelmesi C'de geçerlidir. `struct agac` bu noktada henüz tamamlanmamış (*incomplete*) bir türdür, ama ona işaretçi tanımlamak için türün içeriğini bilmek gerekmez. Bu sayede yapı, kendi türünden düğümlere işaret eden alanlar içerebilir. Bir yapının kendi türünden bir alanı (işaretçi değil, değerin kendisi) olamaz; bu, sonsuz büyüklükte bir yapı demek olurdu.

Adların seçiminde iki ayrıntı vardır. `enum { not, and, or, true, false }` gibi İngilizce adlar sorun çıkarır. `true` ve `false`, C99'un `<stdbool.h>` başlığında makro olarak tanımlıdır ve C23'te anahtar sözcük olmuştur; `not`, `and`, `or` ise `<iso646.h>` başlığında işleç yerine geçen makrolardır. Bu başlıklardan biri eklenirse sayım türü derlenmez. İkinci ayrıntı, `typedef struct agac {…};` gibi sonuna ad yazılmamış bir `typedef`'in hiçbir ad tanımlamamasıdır. Derleyici bu satırı kabul eder ama uyarı verir. Yukarıdaki kodda ASCII ve çakışmasız adlar kullanılmıştır.

Düğümün bellekteki görünümü de incelenmeye değer. 64 bitlik bir sistemde iki işaretçinin her biri 8 bayt tutar ve yapının ilk 16 baytını oluşturur. `veri` alanı (bir `enum`, 4 bayt) 16. bayttan, `deger` alanı (`short`, 2 bayt) 20. bayttan başlar. Derleyici yapının sonuna 2 baytlık dolgu ekler ve boyu 8'in katı olan 24 bayta tamamlar. Bu dolgu, yapılardan oluşan bir dizide her işaretçinin 8'in katı bir adreste durmasını sağlar. Bu sayılar `sizeof` ve `offsetof` ile ölçülmüştür; başka bir derleyicide ya da 32 bitlik bir sistemde farklı çıkabilir.

Ağacın düğümleri çalışma zamanında `malloc` ile oluşturulur. Sunumdaki kodda bu kısım gösterilmez; tam bir program için aşağıdaki iki yardımcı fonksiyon yeterlidir:

```
agac_gost yeni_dugum(mantik v, agac_gost sol, agac_gost sag) {
  agac_gost p = malloc(sizeof *p);
  if (p == NULL) {              /* bellek ayrilamadi */
    fprintf(stderr, "bellek yok\n");
    exit(EXIT_FAILURE);
  }
  p->veri = v;
  p->solcocuk = sol;
  p->sagcocuk = sag;
  p->deger = 0;
  return p;
}

void agac_sil(agac_gost L) {     /* kok sonda: once cocuklar */
  if (L) {
    agac_sil(L->solcocuk);
    agac_sil(L->sagcocuk);
    free(L);
  }
}
```

`yeni_dugum`, bir düğüm büyüklüğünde bellek ister. `sizeof *p` ifadesi, `p`'nin gösterdiği türün boyunu verir; tür adı yazılmadığı için `p`'nin türü değişse bile ifade doğru kalır. `malloc` bellek ayıramazsa `NULL` döndürür. Bu durum denetlenmeden `p->veri` yazılırsa program boş işaretçiye yazar ve çöker; bu yüzden dönüş değeri hemen sınanır. `agac_sil`, ağacın bütün düğümlerini geri verir. Önce iki çocuğu siler, sonra düğümün kendisini. Sıra önemlidir: önce düğüm serbest bırakılsaydı, `L->solcocuk` alanı serbest bırakılmış bellekten okunurdu ve bu tanımsız davranıştır. Ağaç silinmeden program biterse işletim sistemi belleği zaten geri alır, ama uzun çalışan bir programda her atamada yeni ağaç kurulup silinmezse **bellek sızıntısı** oluşur.

Formül ağacı ve atama işlemi şöyle kurulur:

```
agac_gost y[5];   /* yapraklar: x1, x2, x1, x3, x3 */

agac_gost formul_kur(void) {
  int k;
  for (k = 0; k < 5; k++)
    y[k] = yeni_dugum(YANLIS, NULL, NULL);
  /* (x1 VE DEGIL x2) VEYA (DEGIL x1 VE x3) VEYA DEGIL x3 */
  return yeni_dugum(VEYA,
           yeni_dugum(VEYA,
             yeni_dugum(VE, y[0], yeni_dugum(DEGIL, NULL, y[1])),
             yeni_dugum(VE, yeni_dugum(DEGIL, NULL, y[2]), y[3])),
           yeni_dugum(DEGIL, NULL, y[4]));
}

void yapraklari_yerlestir(agac_gost kok, int x1, int x2, int x3) {
  (void)kok;            /* yapraklara y[] ile dogrudan erisilir */
  y[0]->veri = x1 ? DOGRU : YANLIS;
  y[1]->veri = x2 ? DOGRU : YANLIS;
  y[2]->veri = x1 ? DOGRU : YANLIS;
  y[3]->veri = x3 ? DOGRU : YANLIS;
  y[4]->veri = x3 ? DOGRU : YANLIS;
}
```

Beş yaprak `y` dizisinde tutulur. Ağaç kurulurken bu yaprakların adresleri ilgili işleç düğümlerine çocuk olarak verilir. `yapraklari_yerlestir` bir atamayı ağaca yerleştirir: her yaprağın türünü, temsil ettiği değişkenin değerine göre `DOGRU` ya da `YANLIS` yapar. Ağacın yapısı değişmez; yalnız yapraklar değişir.

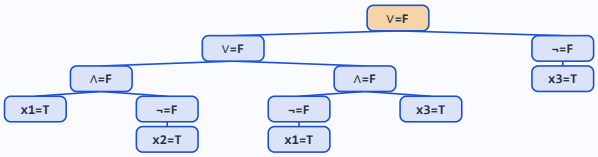
`kok_sonda_degerlendir`, ağacın değerini **kök sonda** (*postorder*) gezinmeyle hesaplar: önce sol alt ağaç, sonra sağ alt ağaç, en son düğümün kendisi. Bu sıra zorunludur, çünkü bir işleç düğümünün değeri çocuklarının değerine bağlıdır ve çocuklar ondan önce hesaplanmalıdır. Fonksiyon `if (L)` ile başlar; boş bir alt ağaç (`NULL`) için hiçbir şey yapmaz. Bu denetim hem yapraklarda hem de ¬ düğümünün boş sol çocuğunda özyinelemeyi durdurur. `switch` deyimi düğümün türüne göre değeri hesaplar. `DEGIL` düğümü sağ çocuğun değerini tersine çevirir. `VE` ve `VEYA` düğümleri iki çocuğun değerini C'nin `&&` ve `||` işleçleriyle birleştirir. Yapraklar türlerine göre 1 ya da 0 değerini alır.

Örnek 3.2

$x_1 = x_2 = x_3 = T$ atamasında ağaç alttan üste şöyle değerlenir:

1. Yapraklar: $x_1 = T, x_2 = T, x_1 = T, x_3 = T, x_3 = T$.
2. ¬ düğümleri: $\neg x_2 = F$, $\neg x_1 = F$, $\neg x_3 = F$.
3. ∧ düğümleri: $x_1 \wedge \neg x_2 = T \wedge F = F$, $\neg x_1 \wedge x_3 = F \wedge T = F$.
4. İçteki ∨: $F \vee F = F$. Kökteki ∨: $F \vee \neg x_3 = F \vee F = F$.

Kök yanlış çıkar: bu atama formülü sağlamaz.



Şekil 3.15: 111 atamasında değerlenmiş ağaç. Hiçbir terim doğru değildir; kök yanlıştır.

3.9.3 Kaba kuvvetle bütün atamalar

Sekiz atamada gezinmek için iç içe üç döngü yazılabilir. Daha kısa bir yol, 0'dan 7'ye kadar sayan tek bir döngü kullanmak ve sayacın bitlerini değişkenlere dağıtmaktır:

```
int m, x1, x2, x3;
for (m = 0; m < 8; m++) {    /* 2^3 atama */
  x1 = (m >> 2) & 1;
  x2 = (m >> 1) & 1;
  x3 = m & 1;
  yapraklari_yerlestir(kok, x1, x2, x3);
  kok_sonda_degerlendir(kok);
  if (kok->deger)
    printf("%d%d%d\n", x1, x2, x3);
}
```

`m`'nin üç bitlik ikili gösterimi bir atamadır. `(m >> 2) & 1` ifadesi `m`'yi iki bit sağa kaydırır ve en sağdaki biti alır; bu `m`'nin 2 numaralı bitidir ve x_1'e atanır. Aynı biçimde 1 numaralı bit x_2'ye, 0 numaralı bit x_3'e gider. $m = 5$ için ikili gösterim 101'dir ve $x_1 = 1, x_2 = 0, x_3 = 1$ çıkar. Her turda atama ağaca yerleştirilir, ağaç değerlendirilir ve kök doğruysa atama yazılır.

`m`	$x_1x_2x_3$	$x_1 \wedge \neg x_2$	$\neg x_1 \wedge x_3$	$\neg x_3$	Kök
0	000	F	F	T	T
1	001	F	T	F	T
2	010	F	F	T	T
3	011	F	T	F	T
4	100	T	F	T	T
5	101	T	F	F	T
6	110	F	F	T	T
7	111	F	F	F	F

Tablo 3.11: Sekiz atamada formülün değeri.

Tablo 3.11, sekiz atamanın yedisinin formülü sağladığını gösterir. Program 000, 001, 010, 011, 100, 101 ve 110 satırlarını yazar; yalnız $x_1 = x_2 = x_3 = 1$ ataması formülü sağlamaz. Bu çıktı, kod tam bir programa dönüştürülüp çalıştırılarak doğrulanmıştır.

SAT sorusu yalnız "bir atama var mı?" diye sorar. Bu soruyu cevaplamak için bütün atamaları yazmak gerekmez; doğru çıkan ilk atamada `return` eden bir yordam yeterlidir. Böyle bir yordam bu formülde ilk atama olan 000'da durur. Formül sağlanamaz olsaydı yordam sekiz atamanın hepsini denemek zorunda kalırdı. Kaba kuvvetin en kötü durumu, cevabın "hayır" olduğu durumdur.

3.9.4 SAT türleri ve ön hesaplama

Sağlanabilirlik sorusu farklı biçimlerde sorulabilir:

Devre sağlanabilirliği (CIRCUIT-SAT)
: Verilen bir mantık devresinin çıkışını 1 yapacak bir giriş ataması var mı? Devre, kapılardan (VE, VEYA, DEĞİL) oluşan bir ağdır.

Formül sağlanabilirliği (Formula SAT)
: Bu bölümdeki gibi ∧, ∨ ve ¬ ile kurulmuş bir önerme formülünün doğru yapılıp yapılamayacağı.

3-CNF-SAT
: Formül, alt cümlelerin ∧'sidir; her alt cümle üç değişkenin (ya da değillerinin) ∨'sidir. Örnek: $(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$. CNF, "birleşimli normal biçim" (*conjunctive normal form*) demektir.

Kaba kuvvet çözümünde dikkat çekici bir verimlilik ayrıntısı vardır. Formül ağacı yalnız **bir kez** kurulur. Her atamada ağacın yapısı aynı kalır ve yalnız yapraklar değişir. Ağaç her atamada baştan kurulsaydı sekiz kez `malloc` ve `free` zinciri çalışırdı. Defalarca yapılacak bir iş için bir kez hazırlanan veri yapısına **ön hesaplama** (*precomputing*) denir. Ön hesaplama, algoritmanın mertebesini değiştirmeyebilir ama sabit çarpanını çok küçültür.

Ön hesaplama metin aramada da kullanılır. Bir metinde bir sözcük ararken, sözcüğün kendi yapısı önceden incelenip bir tabloya yazılırsa arama sırasında geri dönmek gerekmez. Knuth–Morris–Pratt algoritması bu fikrin bilinen bir örneğidir. Algoritma analizi kitaplarındaki açgözlü algoritmalar (*greedy algorithms*), dal ve sınır (*branch and bound*) ve ön hesaplama konuları, kaba kuvvetten daha iyisini aramanın yollarını anlatır.

3.10 n-vezir problemi

Permütasyon ve alt küme üretiminde her seçim geçerliydi; geri izleme yalnız seçimi geri almaktan ibaretti. n-vezir probleminde seçimlerin çoğu kuralları bozar ve budama gerçek bir kazanç sağlar.

3.10.1 Problem ve gösterim

> **Tanım 3.10**
>
> **n-vezir problemi** (*n-queens problem*): $n \times n$ boyutlu bir satranç tahtasına n veziri, hiçbiri diğerini tehdit etmeyecek biçimde yerleştirmek. Satrançta vezir bulunduğu satır, sütun ve iki çapraz boyunca her kareyi tehdit eder. Dolayısıyla aynı satırda, aynı sütunda ya da aynı çaprazda iki vezir olamaz.

İki vezir aynı satırda duramayacağına göre n vezir n satıra birer birer dağılır. Bu gözlem gösterimi sadeleştirir: çözüm, n öğeli bir `q` dizisidir ve `q[r]`, r. satırdaki vezirin sütun numarasıdır. Satır kuralı bu gösterimle kendiliğinden sağlanır. Sütun kuralı, `q`'daki değerlerin hepsinin farklı olmasıdır. Çapraz kuralı için bir sınama gerekir.

(i, q[i]) ve *(r, c)* karelerindeki iki vezir aynı çaprazdaysa satır farkı ile sütun farkı eşittir. Satırlar arasındaki uzaklık kadar sütunda da ilerlemek, 45 derecelik bir çizgide yürümektir. Yeni vezir her zaman daha aşağıdaki bir satıra konduğundan $r > i$'dir ve satır farkı $r - i$'dir. Sütun farkı ise iki yönde olabilir; mutlak değer alınır:

> $|q[i] - c| = r - i$

Kaba kuvvet çözümünde her satırın veziri n sütundan birine konur ve n^n aday çıkar. $n = 8$ için bu $8^8 = 16.777.216$, yani yaklaşık 16,8 milyon adaydır. Sütun kuralını baştan uygulayıp yalnız permütasyonları denemek sayıyı $8! = 40.320$'ye indirir. Geri izleme bundan da azını dener.

3.10.2 uygun() ve yerlestir()

Geri izleme çözümü iki fonksiyondan oluşur. İlki, bir karenin önceki satırlardaki vezirlerle çakışıp çakışmadığını sınar:

```
int q[8];  /* q[r]: r. satirdaki vezirin sutunu */

int uygun(int r, int c) {
  int i;
  for (i = 0; i < r; i++)
    if (q[i] == c || abs(q[i] - c) == r - i)
      return 0;         /* ayni sutun ya da capraz */
  return 1;
}
```

`q` dizisi küresel (*global*) tanımlanmıştır; böylece iki fonksiyon da ona parametre almadan erişir. Dizinin boyu 8'dir, dolayısıyla kod en çok *n = 8* için çalışır. `uygun(r, c)`, *r*. satırın *c*. sütununa vezir konup konamayacağını söyler. Döngü yalnız *0 … r − 1* satırlarına bakar, çünkü alt satırlarda henüz vezir yoktur. Her önceki vezir için iki koşul sınanır: `q[i] == c` aynı sütunu, `abs(q[i] - c) == r - i` aynı çaprazı yakalar. Çakışma varsa fonksiyon hemen 0 döndürür. Döngü çakışma bulmadan biterse kare uygundur. `abs` fonksiyonu `<stdlib.h>` başlığındadır; bu başlık eklenmezse derleyici bildirilmemiş fonksiyon için uyarı ya da hata verir.

İkinci fonksiyon veziri satır satır yerleştirir:

```
int yerlestir(int r, int n) {
  int c;
  if (r == n) return 1;             /* cozum */
  for (c = 0; c < n; c++)
    if (uygun(r, c)) {
      q[r] = c;
      if (yerlestir(r + 1, n)) return 1;
    }
  return 0;                 /* geri don */
}
```

`yerlestir(r, n)`, *r*. satırdan başlayarak kalan vezirleri yerleştirmeye çalışır. Başarırsa 1, başaramazsa 0 döndürür. `r == n` taban durumdur: bütün satırlar doldurulmuştur ve bir çözüm bulunmuştur. Taban durumda değilse döngü sütunları soldan sağa dener. Uygun bir sütun bulununca vezir oraya konur (`q[r] = c`) ve bir sonraki satır için fonksiyon kendini çağırır. Alt çağrı 1 döndürürse çözüm tamamlanmıştır ve 1 hemen yukarı iletilir. Alt çağrı 0 döndürürse bu yerleşim bir çıkmaza varmıştır; döngü sıradaki sütunla devam eder. Hiçbir sütun işe yaramazsa fonksiyon 0 döndürür ve bir üst satır kendi sıradaki sütununu dener.

Kodda seçimi geri alan açık bir satır yoktur. Bunun nedeni gösterimdir: `q[r]` bir sonraki denemede üzerine yazılır ve `uygun` yalnız *r*'den küçük satırlara bakar. Başarısız bir denemeden kalan değerler hiçbir zaman okunmaz. Geri alma işi, `r`'nin yığındaki çerçevelerde saklanmasıyla örtük olarak yapılır.

3.10.3 Kod izleme: 4-vezir

`yerlestir(0, 4)` çağrısı Tablo 3.12'deki gibi ilerler. Kareler *(satır, sütun)* biçiminde yazılmıştır ve satırlar ile sütunlar 0'dan başlar.

Adım	Satır	Denenen sütunlar	Sonuç	`q`
1	0	0	*(0, 0)* uygun	[0, –, –, –]
2	1	0 ×, 1 ×, 2	*(1, 2)* uygun	[0, 2, –, –]
3	2	0 ×, 1 ×, 2 ×, 3 ×	sütun yok, `return 0`	[0, 2, –, –]
4	1	3	*(1, 3)* uygun	[0, 3, –, –]
5	2	0 ×, 1	*(2, 1)* uygun	[0, 3, 1, –]
6	3	0 ×, 1 ×, 2 ×, 3 ×	sütun yok, `return 0`	[0, 3, 1, –]
7	2	2 ×, 3 ×	sütun yok, `return 0`	[0, 3, –, –]
8	1	—	kalan sütun yok, `return 0`	[0, –, –, –]
9	0	1	*(0, 1)* uygun	[1, –, –, –]
10	1	0 ×, 1 ×, 2 ×, 3	*(1, 3)* uygun	[1, 3, –, –]
11	2	0	*(2, 0)* uygun	[1, 3, 0, –]

Adım	Satır	Denenen sütunlar	Sonuç	`q`
12	3	0 ×, 1 ×, 2	*(3, 2)* uygun	[1, 3, 0, 2]
13	4	—	*r* = *n*: çözüm, `return 1`	[1, 3, 0, 2]

Tablo 3.12: `yerlestir(0, 4)` izlemesi.

İzlemenin ilk kısmı, sol üst köşeye konan vezirin çözüme götürmediğini gösterir. Adım 2'de ikinci satırın 0. sütunu birinci vezirle aynı sütunda, 1. sütunu aynı çaprazdadır; ilk uygun sütun 2'dir. Adım 3'te üçüncü satırın her sütunu tehdit altındadır: 0 ve 2 sütun kuralına, 1 ve 3 de *(1, 2)*'deki vezirin çaprazlarına takılır. Fonksiyon 0 döndürür ve ikinci satır sıradaki sütunu, 3'ü dener.



Şekil 3.16: 4-vezir. (a) Adım 3: *(0, 0)* ve *(1, 2)*'deki vezirler üçüncü satırın bütün karelerini tehdit eder (×). (b) Adım 13: bulunan çözüm, *q = [1, 3, 0, 2]*.

Adım 4–7 ikinci denemenin de çıkmaza vardığını gösterir. Dördüncü satırda hiçbir kare kalmaz, üçüncü satırın kalan sütunları da tehdit altındadır. Adım 8'de ikinci satır, son sütunu denemiş olduğundan 0 döndürür. Artık birinci satırın veziri yer değiştirmelidir. Adım 9'da vezir *(0, 1)*'e konur ve bu kez her satırda uygun bir kare bulunur. Adım 13'te *r = 4 = n* olur ve çözüm *q = [1, 3, 0, 2]* bulunur. Dönen 1, dört çerçeveden geçerek ilk çağırana ulaşır.

Tablodaki `q[r] = c` atamaları sayılırsa sekiz yerleştirme yapıldığı görülür: *(0, 0)*, *(1, 2)*, *(1, 3)*, *(2, 1)*, *(0, 1)*, *(1, 3)*, *(2, 0)*, *(3, 2)*. Kaba kuvvet $4^4 = 256$ adayı sınardı. Bu sayılar, `yerlestir`'e bir sayaç eklenerek program çalıştırılıp doğrulanmıştır.

Sık yapılan hata

Çapraz sınamasını yalnız bir yönde yazmak. `q[i] - c == r - i` koşulu sağ yukarıdan sol aşağıya giden çaprazı yakalar, öteki yöndeki çaprazı kaçırır. `abs` ile mutlak değer almak iki yönü birden kapsar. Bir başka hata, `uygun` döngüsünü $i \le r$'ye kadar götürmektir; bu durumda `q[r]`'deki eski bir değer okunur ve doğru kareler reddedilebilir.

3.10.4 Budamanın kazancı

Geri izlemenin en kötü durum maliyeti için bir üst sınır kolayca bulunur. Sütun kuralı yüzünden *r*. satırda en çok *n* − *r* sütun aday olabilir; önceki vezirlerin sütunları kapalıdır. Çağrı ağacının birinci düzeyinde en çok *n*, ikinci düzeyinde en çok *n(n − 1)* düğüm vardır ve yaprak sayısı en çok *n!* olur. Maliyet için verilen *O(n!)* üst sınırı buradan gelir; her düğümde yapılan `uygun` denetimleri bu sınıra yalnız bir polinom çarpanı ekler. Bu kaba bir sınırdır: çapraz kuralı ağacın büyük kısmını erkenden budar ve gerçek maliyet *n!*'nin çok altında kalır. Yığın maliyeti derinlik kadar, *O(n)*'dir.

n	İlk çözüm `q`	Yerleştirme	Çözüm sayısı
4	[1, 3, 0, 2]	8	2
5	[0, 2, 4, 1, 3]	5	10
6	[1, 3, 5, 0, 2, 4]	31	4
7	[0, 2, 4, 6, 1, 3, 5]	9	40
8	[0, 4, 7, 5, 2, 6, 1, 3]	113	92

Tablo 3.13: Küçük *n* değerleri için ilk çözüm, yerleştirme sayısı ve toplam çözüm sayısı.

Tablo 3.13'teki değerler programla hesaplanmıştır. Yerleştirme sütunu, ilk çözüme kadar yapılan `q[r] = c` atamalarını sayar. Sekiz vezir için geri izleme yalnız 113 yerleştirmeyle bir çözüm bulur; kaba kuvvetin 16,8 milyon adayıyla karşılaştırınca fark çarpıcıdır. Çözüm sayısı sütunu, algoritma ilk çözümde durmayıp bütün çözümleri sayacak biçimde değiştirilerek bulunmuştur. 4 için 2, 8 için 92 çözüm vardır. Yerleştirme sayısı n ile düzenli büyümez: 5 ve 7 için ilk denemeler hemen başarılı olur, 6 için çok sayıda çıkmaza girilir. Geri izlemenin maliyeti girdinin yapısına sıkı sıkıya bağlıdır.

3.11 At gezisi ve Warnsdorff kuralı

Geri izleme her zaman çözümü bulur, ama aday sayısı büyüdükçe yavaşlar. Kimi problemlerde iyi bir seçim kuralı, geri dönmeye hiç gerek bırakmadan çözüme götürür. At gezisi bu durumun klasik örneğidir.

3.11.1 Problem

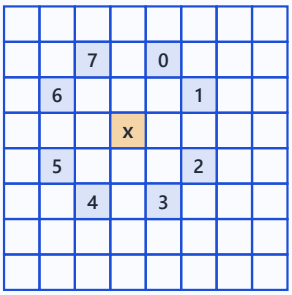
Tanım 3.11

At gezisi (*knight's tour*): satranç atı, bir tahtanın her karesinden tam bir kez geçerek bütün kareleri gezecektir. At L biçiminde hareket eder: iki kare yatay ve bir kare dikey, ya da iki kare dikey ve bir kare yatay.

Tahtanın ortasındaki bir at sekiz kareye gidebilir. Kenarlarda ve köşelerde bu sayı azalır; köşedeki at yalnız iki kareye ulaşır. Hamleler, satır değişimi Δi ve sütun değişimi Δj olarak bir tabloda tutulur:

Hamle	0	1	2	3	4	5	6	7
Δi	−2	−1	+1	+2	+2	+1	−1	−2
Δj	+1	+2	+2	+1	−1	−2	−2	−1

Tablo 3.14: Atın sekiz hamlesi.



Şekil 3.17: x konumundaki atın gidebileceği sekiz kare; numaralar hamle tablosundaki sıradır.

Tablo, hamleleri saat yönünde dolaşır: 0. hamle iki satır yukarı ve bir sütun sağa, 2. hamle bir satır aşağı ve iki sütun sağa götürür. Tabloyla çalışmak, sekiz ayrı `if` yazmak yerine tek bir döngüyle bütün hamleleri denemeyi sağlar. (i, j) karesinden k. hamle $(i + \Delta i[k], j + \Delta j[k])$ karesine gider; karenin tahta içinde ve henüz gezilmemiş olması gerekir.

3.11.2 Geri izleme ve sezgisel yöntem

Geri izleme çözümü n-vezire benzer. At bir kareye konur ve sırayla her hamle denenir. Atın gidebileceği boş kare kalmamışsa ve bütün kareler gezilmemişse bir çıkmaz vardır; son hamle geri alınır ve sıradaki hamle denenir. Bu yöntem bir çözüm varsa onu her zaman bulur. Ama kaba kuvvet niteliğindedir. Her karede sekize kadar seçenek vardır. 8×8 tahtanın 64 karesinde yanlış bir erken seçim, çok sonra fark edilen bir çıkmaza yol açabilir. Algoritma o noktaya kadar kurduğu milyonlarca kısmi geziyi tek tek geri almak zorunda kalır.

Tanım 3.12

Sezgisel yöntem (*heuristic*), her adımda "akıllıca" görünen bir seçim yapan kurala dayanır. Çoğu durumda iyi ya da en iyiye yakın bir çözüm bulur, ama bunu garanti etmez.

Sezgisel kurallar günlük hayatta da kullanılır. Mayın tarlası oyununda oyuncu, açık karelerdeki sayılara bakarak en güvenli görünen kareyi seçer. Bu seçim çoğu zaman doğrudur, ama bazen şansa kalır. Sezgisel bir algoritma da benzer biçimde çalışır: hızlıdır ve genellikle başarılıdır, ama başarısız olduğu girdiler bulunabilir.

3.11.3 Warnsdorff kuralı: 5×5 tahtada izleme

At gezisi için bilinen en eski sezgisel kural H. C. von Warnsdorff'un 1823'te önerdiği kuraldır. Kural, atın bir sonraki gideceği yerin, ilerleme sayısı sıfır olmayan ama en az olan kare olmasını söyler. Bir karenin **ilerleme sayısı**, o kareden doğrudan gidilebilecek boş kare sayısıdır.

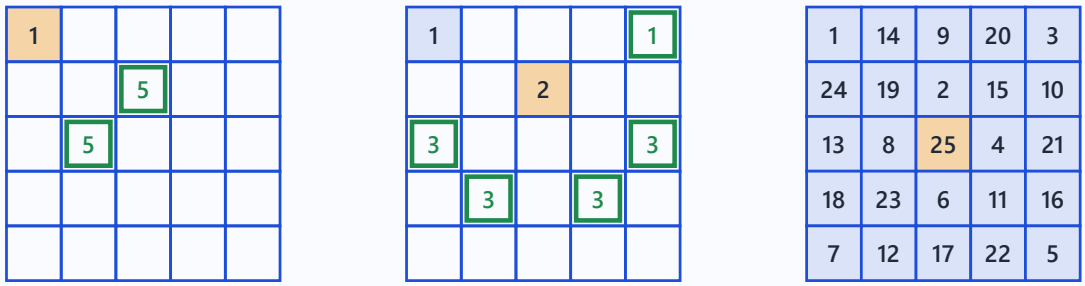
Kuralın mantığı, ulaşılması zor kareleri önce tüketmektir. Köşe ve kenar kareleri az sayıda kareden ulaşılabilir. At bu karelere uğramadan uzaklaşırsa, geri dönebileceği yollar zamanla dolar ve kare ulaşılamaz hâle gelir. En az ilerleme sayılı kareye gitmek bu kareleri erkenden ziyaret eder ve çıkmazı sona bırakmaz. Ortadaki kareler ise çok yoldan ulaşılabildiği için sona kalsa da sorun çıkmaz.

5×5 tahtada sol üst köşeden, yani *(0, 0)* karesinden başlayan bir gezi izlenebilir. Eşitlik durumunda hamle tablosundaki sıra (0'dan 7'ye) belirleyicidir. Tablo 3.15 gezinin önemli adımlarını verir. Adaylar, atın o kareden gidebileceği boş karelerdir; yanlarındaki sayı ilerleme sayısıdır.

Kare no	Konum	Adaylar (ilerleme sayısı)	Seçilen
1	*(0, 0)*	*(1, 2)*: 5, *(2, 1)*: 5	*(1, 2)*, eşitlikte tablo sırası
2	*(1, 2)*	*(0, 4)*: 1, *(2, 4)*: 3, *(3, 3)*: 3, *(3, 1)*: 3, *(2, 0)*: 3	*(0, 4)*
3	*(0, 4)*	*(2, 3)*: 5	*(2, 3)*
4	*(2, 3)*	*(4, 4)*: 1, *(4, 2)*: 3, *(3, 1)*: 2, *(1, 1)*: 3, *(0, 2)*: 3	*(4, 4)*
5	*(4, 4)*	*(3, 2)*: 5	*(3, 2)*
9	*(0, 2)*	*(1, 4)*: 2, *(1, 0)*: 2	*(1, 4)*, eşitlikte tablo sırası
16	*(3, 4)*	*(4, 2)*: 1, *(2, 2)*: 4	*(4, 2)*
24	*(1, 0)*	*(2, 2)*: 0	*(2, 2)*, son boş kare

Tablo 3.15: 5×5 tahtada Warnsdorff gezisinin seçilmiş adımları.

İkinci karede kuralın etkisi açıkça görülür. Beş aday arasından sağ üst köşe *(0, 4)* seçilir, çünkü oradan yalnız bir boş kareye gidilebilir. Köşe şimdi ziyaret edilmezse, ona ulaşan tek yol da dolduğunda köşe sonsuza dek ulaşılamaz kalırdı. Dördüncü karede aynı mantık sağ alt köşeyi, *(4, 4)*'ü seçtirir. Yirmi dördüncü karede tek aday kalır ve onun ilerleme sayısı 0'dır. Kuralın "sıfır olmayan" şartı burada uygulanmaz: bu kare tahtanın son boş karesidir ve gezi onunla tamamlanır.



Şekil 3.18: Warnsdorff gezisi. (a) 1. kare: iki aday, ikisinin de ilerleme sayısı 5. (b) 2. kare: köşe *(0, 4)*'ün ilerleme sayısı 1'dir ve seçilir. (c) Tamamlanmış gezi: sayılar ziyaret sırasıdır; 25. kare tahtanın ortasıdır.

Tamamlanmış gezide 25 kare, hiç geri dönmeden gezilmiştir. Ziyaret sırası satır satır şöyledir: 1 14 9 20 3 / 24 19 2 15 10 / 13 8 25 4 21 / 18 23 6 11 16 / 7 12 17 22 5. Köşeler 1, 3, 5 ve 7. adımlarda, yani gezinin başında tüketilmiştir; orta kare en sona kalmıştır. Bu gezi, kural küçük bir programla uygulanarak her adımın adayları ve ilerleme sayılarıyla birlikte üretilmiş ve sunumdaki değerlerle karşılaştırılmıştır.

Warnsdorff kuralının maliyeti çok düşüktür. $n \times n$ tahtada n^2 hamle yapılır. Her hamlede en çok 8 aday incelenir ve her adayın ilerleme sayısı en çok 8 komşuya bakılarak hesaplanır. Hamle başına iş sabittir (en çok $8 \times 8 = 64$ kare denetimi); toplam iş $O(n^2)$'dir. Bellek, tahtanın kendisi için $O(n^2)$'dir. Geri izleme ise en kötü durumda üstel sayıda kısmi gezi dener.

Kural her zaman başarılı olmaz. Eşitliklerin nasıl bozulduğuna ve başlangıç karesine bağlı olarak at bazen çıkmaza girer. Bu durumda Warnsdorff kuralı geri izlemeyle birleştirilebilir: adaylar ilerleme sayısına göre sıralanıp bu sırayla denenir, çıkmazda geri dönülür. Kural yalnız denemelerin sırasını belirlediği için çözüm garantisi korunur. Pratikte geri dönmeye çok nadiren gerek kalır.

3.12 Karmaşıklık özeti

Bu notta incelenen algoritmaların zaman ve bellek maliyetleri Tablo 3.16'da bir arada verilmiştir. Bellek sütunu, çağrı yığınının ve algoritmanın kullandığı ek yapıların toplamıdır; girdinin kendisi sayılmamıştır.

Algoritma	Zaman	Yığın / ek bellek	Açıklama
`fakt` **(özyinelemeli)**	$\Theta(n)$	$O(n)$	kuyruk sürümü $O(1)$'e indirilebilir
İkili arama (yinelemeli)	$O(\log n)$	$O(1)$	dizi sıralı olmalı
İkili arama (özyinelemeli)	$O(\log n)$	$O(\log n)$	kuyruk özyinelemesi
Hanoi	$\Theta(2^n)$	$O(n)$	$2^n - 1$ hamle, en iyisi
Permütasyon	$\Theta(n \cdot n!)$	$O(n)$	$n!$ diziliş
Alt küme üretimi	$\Theta(n \cdot 2^n)$	$O(n)$	2^n alt küme
SAT (kaba kuvvet)	$O(2^n \cdot \lvert F\rvert)$	$O(\lvert F\rvert)$	$\lvert F\rvert$: formül ağacının boyu
n-vezir (geri izleme)	$O(n!)$ üst sınır	$O(n)$	budama pratikte çok daha az dener
At gezisi (Warnsdorff)	$O(n^2)$	$O(n^2)$	sezgisel; geri izleme üstel

Tablo 3.16: Bölümdeki algoritmaların karmaşıklığı.

Tablo iki gruba ayrılır. İlk üç satırdaki algoritmalar polinom zamanlıdır ve büyük girdilerde rahatça çalışır. Geri kalanların çoğu üstel ya da faktöriyel zamanlıdır. Hanoi, permütasyon ve alt küme üretiminde bu maliyet kaçınılmazdır, çünkü çıktının kendisi o boydadır. SAT ve n-vezirde ise çıktı küçüktür (bir atama ya da bir yerleşim), ama onu bulmak için bilinen en iyi genel yöntemler üstel zaman alır. Warnsdorff kuralı, üstel bir aramayı sezgisel bir seçimle polinom zamana indirmenin örneğidir; karşılığında çözüm garantisinden vazgeçer.

Yığın sütununda dikkat çeken bir düzen vardır: özyinelemeli algoritmaların hepsinde yığın maliyeti, özyineleme derinliğidir. Zaman üstel bile olsa derinlik çoğu zaman n'dir. Bu yüzden Hanoi ya da permütasyon gibi algoritmalar yığın taşmasıyla değil, zaman yetmezliğiyle sınırlanır.

3.13 Sık yapılan hatalar

1. **Taban durumu unutmak ya da ona hiç ulaşmayan bir adım yazmak.** Çağrılar durmaz ve program yığın taşmasıyla çöker. Her özyinelemeli fonksiyonda, her çağrıda azalan bir ölçü ve o ölçünün en küçük değerini yakalayan bir taban durum bulunmalıdır.
2. **Özyinelemeli çağrının sonucunu döndürmemek.** `binsearch(...)` yazılıp dönen değer atılırsa fonksiyon doğru cevabı bulduğu hâlde yanlış değer döndürür. Kuyruk çağrısı `return binsearch(...)` biçiminde yazılmalıdır.
3. **İkili aramayı sırasız dizide ya da bağlı listede kullanmak.** Sırasız dizide yarıyı atma kararı yanlıştır ve var olan değer bulunamayabilir. Bağlı listede ortaya ulaşmak doğrusal zaman alır ve logaritmik kazanç kaybolur.
4. **Bulunan öğenin değerini döndürmek.** Değer, hata işareti -1 ile karışabilir ve konum bilgisi kaybolur. İkili arama indis döndürmelidir.
5. **Geri izlemede seçimi geri almayı unutmak.** Permütasyondaki ikinci `SWAP` ya da alt küme üretimindeki `sec[i] = 0` yazılmazsa kardeş dallar bozuk bir başlangıç durumu görür. Bazı çözümler tekrar eder, bazıları hiç çıkmaz.
6. **Üstel bir algoritmayı "makine hızlı" diye büyük n ile çalıştırmak.** 64 diskli Hanoi yüz milyarlarca yıl, 20 öğeli permütasyon on yıllar sürer. Algoritma seçilmeden önce büyüme mertebesi hesaplanmalıdır.
7. **Derin özyinelemenin yığın maliyetini hesaba katmamak.** Derinliği n olan bir özyineleme, n yüz binler mertebesindeyse birkaç megabaytlık çağrı yığınını doldurur. Kuyruk özyinelemesi C'de güvence değildir; böyle durumlarda döngü tercih edilir.
8. **Taşmayı gözden kaçırmak.** `long` Windows'ta 32 bittir ve `fakt(13)` taşar. İkili aramada `(left + right) / 2` çok büyük dizilerde taşabilir. İşaretli taşma tanımsız davranıştır ve sessizce yanlış sonuç verebilir.

3.14 Özet

- **Özyineleme:** bir fonksiyonun kendini daha küçük bir girdiyle çağırmasıdır. Taban durum özyinelemesiz çözülen en küçük örnektir; özyinelemeli adım problemi küçültür ve her çağrıda taban duruma yaklaşır.
- **Çağrı yığını:** her çağrı, parametreleri, yerel değişkenleri ve dönüş adresini tutan bir yığın çerçevesi açar. Derinliği d olan özyineleme $O(d)$ ek bellek kullanır; aynı değişkenin her çerçevede ayrı kopyası vardır.
- **Özyineleme ve döngü:** her özyineleme açık bir yığınla döngüye çevrilebilir. Özyineleme tanıma yakındır; döngü $O(1)$ bellekle ve çağrı maliyeti olmadan çalışır.
- **Kuyruk özyinelemesi:** özyinelemeli çağrı son iştir; birikimli parametreyle elde edilir ve derleyici bunu atlamaya çevirebilir. C standardı bu eniyilemeyi garanti etmez.
- **İkili arama:** sıralı dizide aralığı her turda yarıya indirir ve en çok $\lfloor \log_2 n \rfloor + 1$ tur döner: $O(\log n)$. İndis döndürür, -1 "yok" demektir. Yinelemeli sürüm $O(1)$, özyinelemeli sürüm $O(\log n)$ ek bellek kullanır.
- **Hanoi kuleleri:** $h(n) = 2h(n-1) + 1$ bağıntısının çözümü $2^n - 1$'dir ve bu sayı en iyisidir. Çağrı ağacı yığının tarihçesidir; hamleler kök ortada gezinmeyle sıralanır.
- **Permütasyon:** takas, özyineleme ve geri takasla $n!$ diziliş üretilir. Çağrı sayısı yaklaşık $e \cdot n!$, zaman $\Theta(n \cdot n!)$'dir.
- **Geri izleme:** kısmi çözüm adım adım kurulur, bir kural bozulunca dal budanır ve son seçim geri alınır. Alt küme üretimi ve n-vezir bu yöntemle çözülür; SAT ise kaba kuvvetle bütün atamalar denenerek çözülmüştür.
- **SAT:** bir mantık formülünü doğru yapan atama aranır. Kaba kuvvet 2^n atamayı dener; formül ağacı bir kez kurulur ve kök sonda gezinmeyle değerlendirilir. Problem NP-tamdır.
- **Sezgisel yöntem:** Warnsdorff kuralı atı her adımda en az ilerleme sayılı kareye götürür ve gezi çoğunlukla hiç geri dönmeden $O(n^2)$ zamanda tamamlanır. Daha iyi bir çözüm aramaktan vazgeçilmez.

3.15 Kavram listesi

özyineleme *recursion* · özyinelemeli *recursive* · taban durum *base case* · çağrı yığını *call stack* · yığın çerçevesi *stack frame* · yığın taşması *stack overflow* · kuyruk özyinelemesi *tail recursion* · ikili arama *binary search* · yineleme bağıntısı *recurrence relation* · permütasyon *permutation* · takas *swap* · kaba kuvvet *brute force* · geri izleme *backtracking* · alt küme *subset* · sağlanabilirlik *satisfiability (SAT)* · kök sonda *postorder* · n-vezir problemi *n-queens problem* · sezgisel *heuristic*

3.16 Kendini Dene

1. `fakt_k(5, 1)` çağrısı hangi çağrıları yapar ve ne döndürür? Kuyruk çağrısı eniyilemesi kapalıyken yığında en çok kaç `fakt_k` çerçevesi bulunur?

   Cevap

   Çağrılar `fakt_k(5, 1)`, `fakt_k(4, 5)`, `fakt_k(3, 20)`, `fakt_k(2, 60)`, `fakt_k(1, 120)` biçimindedir; son çağrı taban durumdur ve 120 döner. 120 hiçbir işlem görmeden yukarı iletilir. Eniyileme kapalıyken beş çerçeve aynı anda yığındadır; açıkken tek çerçeve yeter.

2. Test dizisinde (`{4, 13, 18, 19, 24, 27, 33, 36, 39, 40}`) yinelemeli `binsearch` 39'u ararken hangi orta indisleri dener ve kaç tur döner?

   Cevap

   Orta indisler 4 (*24 < 39*, `left = 5`), 7 (*36 < 39*, `left = 8`) ve 8'dir (`list[8] = 39`). Fonksiyon üç turda 8 döndürür.

3. Özyinelemeli `binsearch(list, 5, 0, 9)` aynı dizide kaç çağrı yapar? Hangi aralıklarla çağrılır ve sonuç nedir?

   Cevap

   Dört çağrı yapılır: *[0, 9]* (orta 24), *[0, 3]* (orta 13), *[0, 0]* (orta 4) ve *[1, 0]*. Son aralık boştur ve -1 döner; bu değer üç çerçeveden geçerek iletilir. En derin anda yığında dört `binsearch` çerçevesi vardır.

4. `hanoi(5, "A", "B", "C")` kaç satır yazar, fonksiyon kendini toplam kaç kez çağırır (ilk çağrı dahil) ve yığın en çok kaç `hanoi` çerçevesi derinleşir?

> **Cevap**
>
> $2^5 - 1 = 31$ satır yazılır. Her çağrı tam bir `printf` yaptığından çağrı sayısı da 31'dir. Derinlik disk sayısı kadardır: 5 çerçeve.

5. `char s[] = "abcd";` için `perm(s, 0, 4)` çıktısının 7–10. satırları nelerdir? Toplam kaç çağrı yapılır?

> **Cevap**
>
> İlk altı satır a ile başlar (abcd, abdc, acbd, acdb, adcb, adbc). Sonra dizi `bacd` olur ve 7–10. satırlar bacd, badc, bcad, bcda'dır. Çağrı sayısı $1 + 4 + 12 + 24 + 24 = 65$'tir.

6. `altkume("abcd", 4, 0, sec)` kaç satır yazar ve beşinci satır nedir? Fonksiyon toplam kaç kez çağrılır?

> **Cevap**
>
> $2^4 = 16$ satır yazılır. Sıra {abcd}, {abc}, {abd}, {ab}, {acd}, … diye ilerler; beşinci satır {acd}'dir. Çağrı ağacı 5 düzeyli mükemmel ikili ağaçtır: $2^5 - 1 = 31$ çağrı.

7. $(x_1 \vee \neg x_2) \wedge x_3$ formülünü sağlayan atamalar ($x_1x_2x_3$ sırasıyla) hangileridir? Formül ağacında kaç düğüm vardır?

> **Cevap**
>
> $x_3 = 1$ olmalı ve $x_1 = 1$ ya da $x_2 = 0$ olmalıdır: 001, 101 ve 111. Ağaçta altı düğüm vardır: kökte ∧, onun sol çocuğu ∨, ∨'nin çocukları x_1 ve ¬, ¬'nin sağ çocuğu x_2, kökün sağ çocuğu x_3.

8. `yerlestir(0, 6)` çağrısının bulduğu ilk çözüm nedir? 6×6 tahtada toplam kaç çözüm vardır?

> **Cevap**
>
> İlk çözüm $q = [1, 3, 5, 0, 2, 4]$'tür; ona ulaşılana kadar 31 yerleştirme yapılır. 6×6 tahtada 4 çözüm vardır; 5×5'te 10 çözüm olmasına karşın 6×6'da daha azdır.

9. Windows'ta `fakt(13)` neden yanlış sonuç verebilir? 64 bit Linux'ta `fakt` hangi n'ye kadar doğru çalışır?

> **Cevap**
>
> Windows'ta `long` 32 bittir ve en büyük değeri 2.147.483.647'dir. $13! = 6.227.020.800$ bu sınırı aşar; işaretli taşma tanımsız davranıştır. 64 bit Linux'ta `long` 64 bittir; $20! \approx 2{,}43 \cdot 10^{18}$ sığar, $21! \approx 5{,}1 \cdot 10^{19}$ sığmaz. Doğru sonuç $n \le 20$ için alınır.

10. Bir dizinin en büyük öğesini bulan özyinelemeli fonksiyon nasıl yazılır? Yığın derinliği ne olur?

> **Cevap**
>
> Taban durum $n = 1$'dir ve tek öğe döner; özyinelemeli adım ilk $n - 1$ öğenin en büyüğünü bulup son öğeyle karşılaştırır:
>
> ```
> int en_buyuk(int a[], int n) {
>   int m;
>   if (n == 1) return a[0];        /* taban durum */
>   m = en_buyuk(a, n - 1);         /* ilk n-1 ogenin en buyugu */
>   return a[n - 1] > m ? a[n - 1] : m;
> }
> ```
>
> Her çağrı n'yi bir azaltır; derinlik n, yığın maliyeti $O(n)$'dir. Test dizisinde fonksiyon 40 döndürür. Çağrı kuyruk çağrısı değildir, çünkü dönüşten sonra bir karşılaştırma yapılır.

3.17 İleri okuma

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C* (2. baskı), Kesim 1.2: bu notun ikili arama, permütasyon ve SAT örneklerinin kaynağıdır; özyinelemeli algoritmaları ve analizlerini aynı C diliyle verir.
- R. Sedgewick, *Algorithms in C*, Bölüm 5: özyinelemeyi, böl ve fethet yöntemini ve çağrı ağaçlarını ayrıntılı çizimlerle anlatır.
- T. H. Cormen ve diğerleri, *Introduction to Algorithms* (3. baskı), Bölüm 4 ve 34: yineleme bağıntılarını çözme yöntemlerini ve NP-tamlık kuramını, SAT'ın bu kuramdaki yerini verir.
- N. Wirth, *Algorithms + Data Structures = Programs* (1976): geri izlemenin genel kalıbını Pascal ile kuran ve sekiz vezir gibi problemlerle örnekleyen klasik bir ders kitabıdır.

3.18 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3. baskı, MIT Press, 2009 (Bölüm 4: yineleme bağıntıları; Bölüm 34: NP-tamlık, SAT).
2. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, Silicon Press, 2008 (Bölüm 1.2: özyinelemeli algoritmalar).
3. R. Sedgewick, *Algorithms in C*, Addison-Wesley (Bölüm 5: özyineleme).
4. N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.

KISIM II

Doğrusal Yapılar

Bölüm 4–9

KISIM II · BÖLÜM 4

Bağlı Listeler

Dizi, öğeleri bellekte yan yana tutar. Bu düzen indisle erişimi çok hızlı yapar, ama ortaya bir öğe eklemek ya da ortadan bir öğe silmek için arkadaki bütün öğelerin kaydırılması gerekir. Bağlı liste bu sorunu başka bir düzenle çözer. Her öğe ayrı bir düğümde durur ve her düğüm bir sonrakinin adresini taşır. Sıra artık bellekteki konumdan değil, bu adreslerden gelir. Not, C'de bu yapıyı kurmak için gereken işaretçi ve dinamik bellek bilgisiyle başlar. Ardından tek yönlü bağlı listede gezinme, ekleme, silme, arama ve listeyi serbest bırakma işlemlerini kod izlemeleriyle anlatır. Ters çevirme, birleştirme, orta düğümü bulma ve Floyd'un döngü tespiti gibi klasik yordamlar bunları izler. Son kesimler dairesel listeyi, başlık düğümünü, çift yönlü bağlı listeyi ve silinen düğümleri yeniden kullanan kullanılabilir düğüm listesini ele alır.

Kod (C): `github.com/sercankulcu/data-structures-and-algorithms-c/Bolum04`

ÖĞRENME ÇIKTILARI

Düğüm kurmak
Kendine başvuran bir `struct` tanımlamak, `malloc` ile düğüm ayırıp sonucunu denetlemek ve işi biten düğümü `free` ile iade etmek.

Bağ disiplini
Başa, sona, araya ve sıralı eklemeyi hiçbir bağı kaybetmeden, doğru sırayla yapmak; baş işaretçisi değişen yordamlara listeyi adresle aktarmak.

Güvenli silme
Bellek sızıntısı ve sarkan işaretçi oluşturmadan düğüm silmek ve bütün bir listeyi serbest bırakmak.

Klasik yordamlar
Ters çevirme, birleştirme, orta düğümü bulma ve Floyd döngü tespitini *O(1)* ek bellekle yazmak ve maliyetlerini gerekçelendirmek.

Liste türleri ve bellek yönetimi
Dairesel, başlık düğümlü ve çift yönlü bağlı listeleri amaca göre seçmek; kullanılabilir düğüm listesiyle düğümleri yeniden kullanmak.

4.1 Dizi yetmediğinde

Bir veri yapısı seçilirken önce hangi işlemlerin sık yapılacağı sorulur. Dizi, belirli işlemler için çok iyi, başka işlemler için ise oldukça kötüdür. Bağlı liste de dizinin zayıf olduğu yerde güçlü, güçlü olduğu yerde zayıftır. Bu kesim iki yapıyı aynı ekleme işlemi üzerinde karşılaştırır ve bağlı listenin temel fikrini, yani düğüm ve bağ kavramlarını tanıtır.

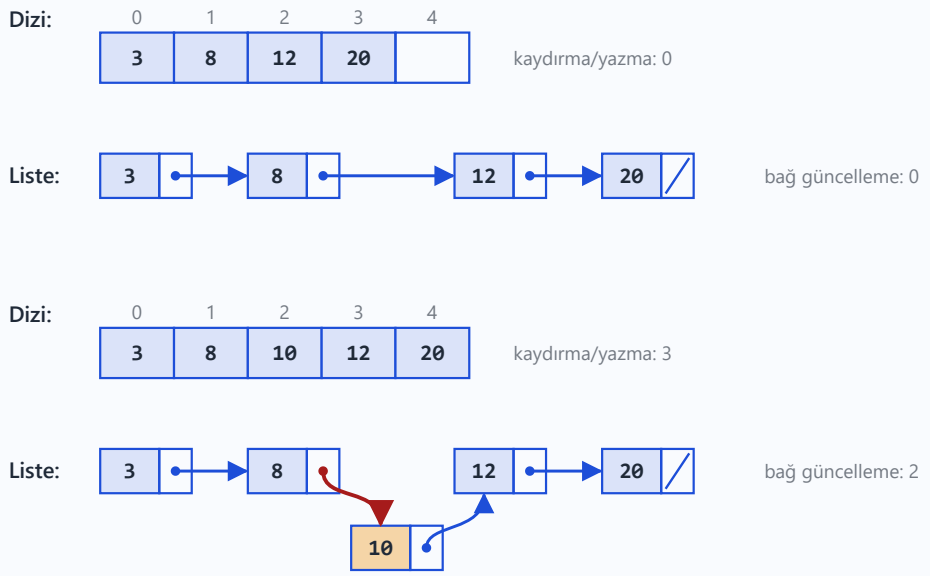
4.1.1 Dizide ekleme ve silmenin maliyeti

C'de bir dizi, aynı tipten öğelerin bellekte art arda dizildiği tek bir bloktur. `int a[5]` tanımı yirmi baytlık bir blok ayırır; `a[0]` bu bloğun ilk dört baytında, `a[1]` sonraki dört baytında durur. Bu yerleşimin büyük bir yararı vardır. `a[i]`'nin adresi tek bir hesapla bulunur: başlangıç adresine $i \cdot sizeof(int)$ eklenir. Bu yüzden dizide herhangi bir öğeye erişmek, dizi ne kadar uzun olursa olsun sabit sürer, yani *O(1)*'dir. Buna **rastgele erişim** (*random access*) denir.

Aynı yerleşim ekleme ve silmede bir yüke dönüşür. Öğeler arasında boşluk bırakılamaz; sıra, konumla belirlenir. Sıralı bir dizinin ortasına yeni bir değer eklemek için önce ona yer açmak gerekir. Yer açmanın tek yolu, eklenecek konumdan sonraki bütün öğeleri birer hücre sağa kaydırmaktır. Silmede de tersi olur: silinen öğenin bıraktığı boşluk, arkadaki öğeler birer hücre sola kaydırılarak kapatılır.

Şekil 4.1 bu farkı somut bir örnekle gösterir. Sıralı 3, 8, 12, 20 dizisine 8'den sonra gelecek biçimde 10 eklenecektir. Dizide önce 20 bir sağa, sonra 12 bir sağa taşınır. Ancak bundan sonra 10, boşalan 2 numaralı hücreye yazılabilir. Toplam iki taşıma ve bir yazma yapılır. Listede ise hiçbir öğe yerinden oynamaz. Yeni düğüm bellekte boş bulunan herhangi bir yere konur ve iki bağ değiştirilir: yeni düğüm

12'yi, 8 ise yeni düğümü gösterir.



Şekil 4.1: 3, 8, 12, 20 dizisine ve listesine 8'den sonra 10 eklemek. Üstteki çizim başlangıç durumudur. Alttaki çizimde dizi 20'yi ve 12'yi birer hücre sağa kaydırmış, 10'u boşalan hücreye yazmıştır (üç kaydırma/yazma). Listede yeni düğüm 12'yi, 8 de yeni düğümü gösterir; yalnız iki bağ güncellenmiştir.

Bu örnek dört öğelidir ve fark küçük görünür. Ama dizide kaydırılacak öğe sayısı, eklemenin yapıldığı konumdan sonraki öğe sayısına eşittir. n öğeli bir dizinin başına ekleme yapılırsa n öğenin hepsi kaydırılır. Bu yüzden dizide ekleme ve silme en kötü durumda $O(n)$'dir. Listede ise eklemenin yapılacağı düğüm biliniyorsa yalnız iki bağ yazılır. Liste ne kadar uzun olursa olsun bu iş değişmez; maliyet $O(1)$'dir.

Dizinin ikinci zayıflığı boyutudur. Durağan bir dizinin boyutu derleme sırasında sabitlenir. Dinamik bir dizi `realloc` ile büyütülebilir, ama büyütme bütün öğelerin yeni bir bloğa kopyalanmasını gerektirebilir. Bağlı liste ise her öğe için ayrı bir küçük blok ister. Liste bir düğüm büyüyüp bir düğüm küçülür; önceden bir kapasite kestirmek gerekmez.

Bağlı listenin bedeli de vardır. Düğümler bellekte dağınık durduğu için i. düğümün adresi hesaplanamaz. Ona ulaşmanın tek yolu, ilk düğümden başlayıp bağları tek tek izlemektir. Bu yüzden listede i. öğeye erişim $O(i)$ sürer. Ayrıca her düğüm, verinin yanında bir adres de taşır; bu da ek bellek demektir. Kısacası iki yapı birbirinin tersidir. Sık erişim ve az değişiklik varsa dizi, sık ekleme ve silme varsa bağlı liste daha uygundur.

4.1.2 Düğüm ve bağ

Tanım 4.1

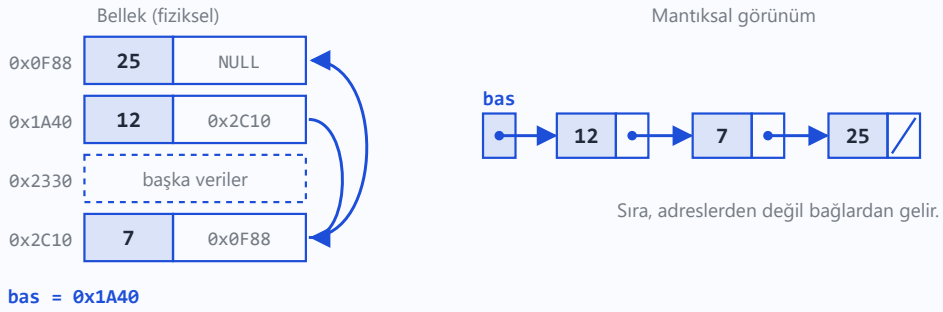
Bağlı liste (*linked list*), öğeleri **düğüm** (*node*) adı verilen ayrı bellek bloklarında tutan doğrusal bir veri yapısıdır. Her düğüm iki alandan oluşur: öğenin değerini tutan **veri alanı** ve bir sonraki düğümün adresini tutan **bağ** (*link*) alanı. Son düğümün bağ alanında `NULL` bulunur. Listenin ilk düğümünün adresini tutan işaretçiye listenin **başı** (*head*) denir. Hiç düğümü olmayan **boş liste**, başı `NULL` olan listedir.

Bu tanımda her düğüm yalnız bir sonrakini bilir. Böyle bir listeye **tek yönlü bağlı liste** (*singly linked list*) denir. Düğümün kendinden önceki düğümü bilmemesi, birçok işlemin biçimini belirler. Örneğin bir düğümü silmek için ondan önceki düğümün bağını değiştirmek gerekir, ama tek yönlü listede öncekine geri dönmenin yolu yoktur.

Bağlı listeyi bir hazine avına benzetmek mümkündür. Her ipucu kâğıdı bir değer ve bir sonraki kâğıdın nerede saklandığını söyler. Kâğıtların evin neresinde durduğu önemli değildir; sırayı kâğıtların üzerindeki yön bilgisi kurar. İlk kâğıdın yerini bilen biri bütün kâğıtları sırasıyla bulabilir. İlk kâğıdın yerini unutan biri ise, kâğıtlar hâlâ evde dursa da hiçbirine ulaşamaz. Bağlı listede baş işaretçisinin kaybolması tam olarak budur: düğümler bellekte durur, ama artık hiçbir değişken onları göstermez.

4.1.3 Bellekteki görünüm

Bağlı listenin en önemli özelliği, düğümlerin bellekte yan yana durmak zorunda olmamasıdır. Şekil 4.2 üç düğümlü bir listenin hem bellekteki gerçek yerleşimini hem de mantıksal görünümünü verir. Sol tarafta bellek, adresleriyle birlikte bir şerit olarak çizilmiştir. 12 değerli düğüm `0x1A40` adresinde, 7 değerli düğüm `0x2C10` adresinde, 25 değerli düğüm ise `0x0F88` adresindedir. Aralarında başka veriler de vardır.



Şekil 4.2: Düğümler bellekte dağınık durur. `bas` değişkeni ilk düğümün adresini (`0x1A40`) tutar. 12'nin bağ alanında 7'nin adresi (`0x2C10`), 7'nin bağ alanında 25'in adresi (`0x0F88`), 25'in bağ alanında `NULL` vardır. Sağdaki mantıksal görünüm aynı listeyi kutu ve oklarla çizer.

Adreslere bakıldığında sıra 0x0F88, 0x1A40, 0x2C10 gibi görünür. Bu sırayla okunsaydı liste 25, 12, 7 olurdu. Oysa listenin sırası 12, 7, 25'tir. Sıranın kaynağı adreslerin büyüklüğü değil, bağlardır: `bas` 12'yi, 12 7'yi, 7 de 25'i gösterir. Şeklin sağ yanındaki mantıksal görünüm bu bilgiyi kutu ve oklarla çizer. Not boyunca kullanılan bütün çizimler bu kutu-ok gösterimindedir. Bir kutunun sol yarısı veri alanını, sağ yarısı bağ alanını gösterir. Bağ alanından çıkan ok, o alanda adresi bulunan düğüme gider. Bağ alanındaki eğik çizgi `NULL` anlamına gelir.

Çizimlerdeki oklar bir kolaylıktır. Bellekte ok yoktur; yalnız sayılar vardır. Bir bağ alanı, öteki düğümün başlangıç adresini bir tam sayı olarak saklar. Okun "gösterdiği" düğüme gitmek, o sayıyı adres olarak kullanıp oradaki belleği okumak demektir. Bu ayrım, işaretçi hatalarını anlamak için önemlidir. Örneğin bir düğüm `free` ile iade edildiğinde onu gösteren başka bir bağdaki sayı kendiliğinden silinmez. Çizimde ok hâlâ oradadır ama artık geçersiz bir yeri gösterir.

Adres kavramı da zaman içinde değişmiştir. Eski x86 işlemcilerinde bir adres iki parçadan oluşurdu: bir bölüt (*segment*) ve bu bölütün içindeki konum. Bu düzene **bölütlü bellek** denir ve o dönemin C derleyicilerinde "yakın" ve "uzak" işaretçi gibi ayrımlar doğururdu. Bugünkü işletim sistemlerinde her programın tek parça, düz bir adres uzayı vardır. Bir işaretçi bu uzayda bir sayıdan ibarettir. 64 bitlik bir sistemde işaretçi 8 bayt yer kaplar.

4.2 C'de işaretçi ve dinamik bellek

Bağlı liste, C'nin iki aracı üzerine kuruludur: başka bir nesnenin adresini tutan işaretçi ve çalışma sırasında bellek isteyip geri vermeyi sağlayan `malloc` ile `free`. Bu iki araç iyi oturmadan bağlı liste kodu ezberden öteye geçmez. Bu konu genellikle ikinci sınıfta kazanılmazsa sonraki yıllarda da eksik kalır; dersin bağlı listeye geçmeden önce C'ye ve işaretçilere dönmesinin nedeni budur. İşaretçiyi bildiğini düşünen bir öğrencinin liste kodunda yine de takılması olağandır. Aşağıdaki kesim bu yüzden temel kavramları kısaca ama eksiksiz yeniden kurar.

4.2.1 Adres ve işaretçi

Tanım 4.2

İşaretçi (*pointer*), bir nesnenin bellek adresini tutan değişkendir. `T *p` bildirimi, `T` tipinden bir nesnenin adresini tutabilen `p` işaretçisini tanımlar. `&x` ifadesi `x`'in adresini verir. `*p` ifadesi ise `p`'nin gösterdiği nesnenin kendisidir; okunabilir ve üzerine yazılabilir. Hiçbir nesneyi göstermeyen işaretçinin değeri `NULL`'dır.

Sunumdaki kısa kod parçası bu işleçleri ve bir düğümün ayrılıp bırakılmasını tek yerde toplar:

```
int x = 5;
int *p = &x;       // p, x'in adresini tutar
*p = 7;            // p'nin gösterdiğine yaz: x == 7

oge_gost q = NULL; // henüz hiçbir düğüm yok
q = malloc(sizeof *q);
if (q != NULL) {
    q->deger = 3;  // (*q).deger = 3
    q->bag = NULL;
}
free(q);           // iade; q artık sarkan işaretçi
q = NULL;
```

İlk üç satır yalnız tam sayı ve işaretçiyle ilgilidir. `int x = 5;` bellekte dört baytlık bir yer ayırır ve oraya 5 yazar. `int *p = &x;` satırı `p` adında bir işaretçi tanımlar ve içine `x`'in adresini koyar. Artık `p` ile `x` iki ayrı değişkendir: `x` bir sayı, `p` bir adres taşır. `*p = 7;` satırı `p`'nin gösterdiği yere, yani `x`'e 7 yazar. `x` hiç adıyla anılmadığı hâlde değeri değişmiştir. Bir fonksiyonun çağıranın değişkenini değiştirebilmesinin yolu da budur: fonksiyona değişkenin kendisi değil adresi verilir.

`*` işareti iki ayrı yerde iki ayrı iş görür ve bu, yeni başlayanlar için sık bir karışıklık kaynağıdır. Bildirimde (`int *p`) "`p` bir işaretçidir" anlamına gelir. Bir ifadenin içinde (`*p = 7`) ise "`p`'nin gösterdiği nesne" anlamına gelir ve **içerik işleci** diye anılır. `&` işleci de **adres işlecidir**. İkisi birbirinin tersidir: `*&x`, `x`'in kendisidir.

Satır	`x`	`p`	`q`	Açıklama
`int x = 5;`	5	—	—	`x` için 4 bayt ayrıldı.
`int *p = &x;`	5	&x	—	`p`, `x`'in adresini tutar.
`*p = 7;`	7	&x	—	`p` üzerinden `x`'e yazıldı.
`oge_gost q = NULL;`	7	&x	NULL	Henüz düğüm yok.
`q = malloc(sizeof *q);`	7	&x	0x2C10	Heap bellekte 16 baytlık blok ayrıldı; alanlar çöp değer taşır.
`q->deger = 3; q->bag = NULL;`	7	&x	0x2C10	Düğüm: `deger = 3`, `bag = NULL`.
`free(q);`	7	&x	0x2C10	Blok iade edildi; `q` hâlâ eski adresi tutar (sarkan).
`q = NULL;`	7	&x	NULL	Sarkan değer silindi.

Tablo 4.1: İşaretçi parçasının izlemesi. Adresler örnek değerlerdir; gerçek adresler her çalıştırmada değişir.

4.2.2 Yapıya işaretçi ve `->` işleci

Parçanın ikinci yarısı bir düğüm üzerinde çalışır. `oge_gost`, bir sonraki kesimde tanımlanacak olan "düğüme işaretçi" tipidir. `q` bir düğümün adresini tuttuğuna göre, düğümün `deger` alanına ulaşmak için önce `*q` ile düğümün kendisine, sonra nokta işleciyle alana gidilir: `(*q).deger`. Buradaki parantez zorunludur. C'de nokta işlecinin önceliği `*`'dan yüksektir. Parantez konmazsa `*q.deger` ifadesi `*(q.deger)` diye okunur; `q` bir yapı değil işaretçi olduğu için bu ifade derlenmez.

Bu yazım hem uzun hem de hataya açık olduğu için C bir kısaltma sunar: `q->deger`. Ok işleci `->`, solundaki işaretçinin gösterdiği yapının sağındaki alanına gider. `q->deger` ile `(*q).deger` tamamen aynıdır. Bağlı liste kodunda ok işleci sürekli zincirlenir. `p->bag->deger`, `p`'nin gösterdiği düğümün sonrakinin değeridir; `p->bag->bag` ise iki sonraki düğümün adresidir. Böyle bir zinciri okurken her okun önündeki işaretçinin `NULL` olmadığından emin olmak gerekir. `NULL` bir işaretçinin alanına erişmek tanımsız davranıştır ve çoğu sistemde programın çökmesiyle sonuçlanır.

4.2.3 `malloc` ile düğüm ayırmak

Fonksiyon içinde tanımlanan yerel değişkenler, fonksiyon dönünce yok olur. Bağlı listenin düğümleri ise onları oluşturan fonksiyondan sonra da yaşamalıdır. Bu yüzden düğümler **heap bellekten** (*heap memory*) ayrılır. Heap bellekteki bir blok, `malloc` ile istenir ve program onu `free` ile geri verene kadar yaşar. `malloc`, istenen bayt sayısı kadar bir blok ayırır ve bloğun başlangıç adresini `void *` tipinde döndürür. C'de `void *` her işaretçi tipine kendiliğinden dönüşür; bu yüzden dönüş değerini tip dönüştürmesi (*cast*) yapmadan atamak yeterlidir. Bellek kalmamışsa `malloc` `NULL` döndürür.

Sunum, düğüm ayırmayı tek bir yardımcı fonksiyonda toplar:

```
oge_gost yeni_oge(int x) {
    oge_gost p = malloc(sizeof *p);
    if (p == NULL) {
        fprintf(stderr, "Bellek yetmiyor yetmiyoor...\n");
        exit(1);
    }
    p->deger = x;
    p->bag = NULL;
    return p;
}
```

- `malloc(sizeof *p)` ifadesi, `p` 'nin gösterdiği tipin, yani `struct oge` 'nin boyutu kadar yer ister. `sizeof *p` derleme sırasında hesaplanır; `p` 'nin o anki değerine bakılmaz. Bu yüzden `p` henüz ilk değer almamışken bile güvenlidir. Tip adını tekrar yazmamak, sonradan `p` 'nin tipi değişirse boyutun yanlış kalmasını önler.
- Dönüş değeri hemen `NULL` 'a karşı denetlenir. Denetim yapılmazsa bellek bittiğinde bir sonraki satır `NULL` bir adrese yazmaya çalışır. Burada seçilen tepki, standart hata akışına bir ileti yazıp programı `exit(1)` ile bitirmektir. Sunumdaki "Bellek yetmiyor yetmiyoor..." iletisi bu durumu esprili bir dille duyurur.
- Yeni düğümün alanları `malloc` 'tan çöp değerle gelir. `p->deger = x` ve `p->bag = NULL` atamaları düğümü tutarlı bir duruma getirir: veri yazılmıştır ve düğüm henüz hiçbir yere bağlı değildir.
- Fonksiyon düğümün adresini döndürür. Düğüm heap bellekte durduğu için fonksiyon dönünce yok olmaz; yalnız yerel `p` değişkeni yok olur.

Tek bir düğüm ne kadar yer kaplar? 64 bitlik bir Windows ya da Linux sisteminde `int` 4 bayt, işaretçi 8 bayttır. İşaretçinin 8'in katı olan bir adreste durması gerektiği için derleyici `deger` alanından sonra 4 baytlık bir dolgu ekler. Sonuçta `sizeof(struct oge)` 16 bayttır (bu değer notun doğrulama programında ölçülmüştür). Dört baytlık veriyi saklamak için 16 bayt harcanmıştır. Üstelik `malloc` her bloğun yanında kendi kayıtları için birkaç bayt daha tutar. Bağlı listenin bellek bedeli küçük öğelerde bu yüzden belirgindir.

Bu noktada çok yaygın bir yanlış anlamayı düzeltmek gerekir. Her işaretçi değişkeni için `malloc` çağırmak gerekmez. `malloc` yalnız *yeni bir düğüm* gerektiğinde çağrılır. Liste üzerinde gezinmek için kullanılan `p` gibi işaretçiler zaten var olan düğümleri gösterir; onlara yer ayırmak hem gereksizdir hem de bellek sızıntısına yol açar.

Sık yapılan hata

Gezinme işaretçisine yer ayırmak: `oge_gost p = malloc(sizeof *p); p = liste;` . İlk satır bir düğüm ayırır, ikinci satır `p` 'nin üzerine listenin başını yazar. Ayrılan bloğun adresi artık hiçbir değişkende yoktur; blok program bitene kadar boşa yer kaplar. Doğrusu yalnız `oge_gost p = liste;` yazmaktır.

4.2.4 `free` ve bellek hataları

`free(p)` , `p` 'nin gösterdiği bloğu belleğe iade eder. Kural basittir: `malloc` 'un döndürdüğü her adres için tam bir kez `free` çağrılır. Bu kuralın iki yönde çiğnenmesi iki farklı hata türü doğurur.

Tanım 4.3

Bellek sızıntısı (*memory leak*), artık hiçbir işaretçiyle erişilemeyen ama `free` ile iade edilmemiş bellektir. **Sarkan işaretçi** (*dangling pointer*) ise `free` edilmiş ya da ömrü bitmiş bir nesnenin adresini tutmaya devam eden işaretçidir.

Sızıntı, `free` 'nin hiç çağrılmamasından doğar. Kısa çalışan bir programda fark edilmeyebilir, çünkü işletim sistemi program bitince bütün belleği geri alır. Ama sürekli çalışan bir sunucuda ya da bir döngü içinde liste kurup bırakan bir programda sızan bellek birikir ve sonunda bellek tükenir. Sarkan işaretçi ise tam tersinden, belleğin erken iade edilmesinden doğar. `free(q)` çağrısı `q` 'nun değerini değiştirmez; `q` eski adresi tutmaya devam eder. Bu adresten okumak ya da bu adrese yazmak tanımsız davranıştır. İade edilen bloğu `malloc` başka bir amaçla yeniden vermiş olabilir; o durumda yazılan değer, programın bambaşka bir verisini bozar. Aynı adresi ikinci kez `free` etmek de tanımsız davranıştır ve `malloc` 'un iç kayıtlarını bozabilir.

Sunumdaki parçanın son iki satırı bu yüzden birlikte yazılmıştır: önce `free(q)` , hemen ardından `q = NULL` . İkinci satır zorunlu değildir, ama iyi bir alışkanlıktır. `NULL` bir işaretçinin yanlışlıkla kullanılması çoğu sistemde programı hemen çökertir ve hata hemen görünür. Sarkan bir işaretçinin kullanılması ise çoğu zaman sessizce "çalışır" ve hata çok sonra, alakasız bir yerde ortaya çıkar. Ayrıca `free(NULL)` çağrısı C standardına göre hiçbir şey yapmaz; bu yüzden `NULL` 'a çekilmiş bir işaretçiyi yanlışlıkla bir kez daha bırakmak zararsızdır.

Sık yapılan hata

Bir düğümü bıraktıktan sonra onun alanını okumak: `free(p); p = p->bag;` . İkinci satır iade edilmiş bloktan okur. Program çoğu zaman beklenen sonucu verir, çünkü blok henüz başka bir işe verilmemiştir; ama bu bir şanstır, doğruluk değildir. Doğru sıra, önce gereken adresi saklamak, sonra bırakmaktır: `q = p->bag; free(p); p = q;` . Bu kural notun silme ve serbest bırakma kodlarının hepsinde karşımıza çıkar.

4.3 Tek yönlü bağlı liste: düğüm ve gezinme

Bu kesimde bağlı listenin C'deki tanımı kurulur ve listedeki her düğümü sırayla ziyaret eden gezinme kalıbı anlatılır. Gezinme, ekleme ve silme dahil neredeyse bütün liste işlemlerinin iskeletidir.

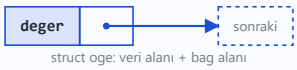
4.3.1 Kendine başvuran yapı

Bir düğüm, bir tam sayı ve bir sonraki düğümün adresi olmak üzere iki alan taşır. İkinci alanın tipi "düğüme işaretçi"dir. Yani düğüm yapısı, kendi tipinden bir nesneyi gösteren bir alan içerir. Böyle bir yapıya **kendine başvuran yapı** (*self-referential struct*) denir. Sunum bu yapıyı Horowitz'in kitabındaki biçimde, önce işaretçi tipine bir ad vererek tanımlar:

```c
typedef struct oge *oge_gost;  // önce işaretçi türü

struct oge {
    int deger;       // veri alanı
    oge_gost bag;    // sonraki düğümün adresi
};

oge_gost liste = NULL;  // boş liste
```



Şekil 4.3: Bir düğüm: `deger` veri alanı ve `bag` alanı. `bag` 'den çıkan ok, sonraki düğümü (kesik çizgili kutu) gösterir.

İlk satır, `struct oge` tanımlanmadan önce ona işaretçi tipine `oge_gost` (öğe göstergesi) adını verir. Bu satır derlenir, çünkü C, boyutu henüz bilinmeyen bir yapıya işaretçi tanımlamaya izin verir. Böyle bir yapıya **eksik tip** (*incomplete type*) denir. Eksik tipin değişkeni tanımlanamaz, çünkü derleyici ona ne kadar yer ayıracağını bilmez. Ama eksik tipe işaretçi tanımlanabilir, çünkü her işaretçinin boyutu, gösterdiği tipten bağımsız olarak bellidir. Ardından gelen `struct oge` tanımı yapıyı tamamlar. İçindeki `bag` alanı, az önce tanımlanan `oge_gost` tipindendir.

Aynı yapı `typedef` olmadan da yazılabilir: `struct oge { int deger; struct oge *bag; };` . Bu yazımda yapı kendi adına, tanımının içinde zaten başvurabilir. Ad seçimi bir alışkanlık meselesidir. Kitaplarda ve eski sınav sorularında aynı tip için `list_gosterge` , `list_gost` ya da `oge_gost` gibi adlar görülür. Bu not sunumla uyumlu olarak `oge_gost` 'u kullanır; sınav sorusunda geçen `list_gost` da aynı tiptir.

Son satır `oge_gost liste = NULL;` boş bir liste oluşturur. Bağlı listenin kendisi için ayrı bir tip yoktur. Liste, ilk düğümün adresini tutan tek bir işaretçiyle temsil edilir. Bu işaretçi `NULL` ise liste boştur. Her işlem bu yüzden boş listeyi ayrıca düşünmek zorundadır.

Sık yapılan hata

`typedef struct liste { … };` yazıp sona tip adını koymamak. `typedef` bir tipe yeni bir ad verir; ad yazılmazsa bildirim boşa gider ve çoğu derleyici "typedef requires a name" uyarısı verir. Ya `typedef` sözcüğü atılmalı ya da kapanış parantezinden sonra tip adı yazılmalıdır: `typedef struct liste { … } liste_t;` .

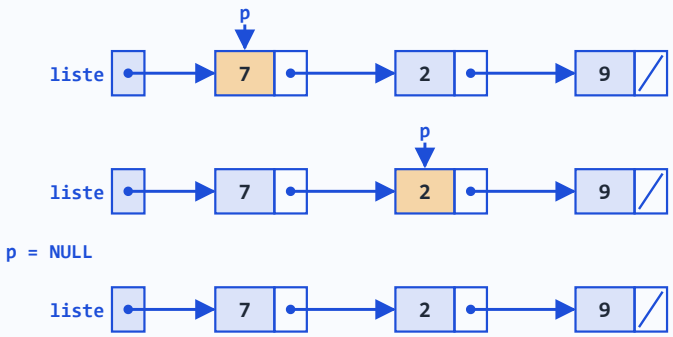
4.3.2 Gezinme

Listedeki bütün düğümleri baştan sona sırayla ziyaret etmeye **gezinme** (*traversal*) denir. Gezinmenin tek aracı bir işaretçidir. İşaretçi ilk düğümden başlar, her adımda bulunduğu düğümün bağ alanını okuyarak bir sonrakine geçer ve `NULL`'a ulaşınca durur. Listenin bütün değerlerini yazdıran fonksiyon bu kalıbın en sade örneğidir:

```
void yazdir(oge_gost p) {
    for (; p != NULL; p = p->bag)
        printf("%d ", p->deger);
    printf("\n");
}
```

`for` döngüsünün ilk bölümü boştur, çünkü `p` zaten parametre olarak ilk düğümün adresiyle gelir. Koşul `p != NULL`, `p`'nin geçerli bir düğümü gösterdiği sürece döngünün sürmesini sağlar. Gövde o düğümün değerini yazar. Artırma bölümü `p = p->bag` ise `p`'yi bir sonraki düğüme taşır. Bu atama bağlı listede dizideki `i++`'nın karşılığıdır. Son düğümün bağı `NULL` olduğu için, son düğüm yazıldıktan sonra `p` `NULL` olur ve döngü biter.

`yazdir(liste)` çağrısında önemli bir ayrıntı vardır. C'de bütün argümanlar **değerle** aktarılır. Fonksiyona `liste` değişkeninin kendisi değil, içindeki adresin bir kopyası gider. Fonksiyon bu kopyayı (`p`) döngüde ilerletir ve sonunda `NULL` yapar. `main`'deki `liste` ise hiç değişmez ve hâlâ ilk düğümü gösterir. Bu yüzden gezinmeyi doğrudan parametre üzerinde yapmak güvenlidir. Aynı döngü `main`'in içinde `liste` değişkeni üzerinde yazılsaydı listenin başı kaybolurdu.



Şekil 4.4: `yazdir` ile 7 → 2 → 9 listesinde gezinme. Üstte `p` ilk düğümde, 7 yazılır. Ortada `p = p->bag` sonrası ikinci düğümde, 2 yazılır. Altta üçüncü düğümün bağı izlenmiş, `p` `NULL` olmuştur; döngü biter.

Şekil 4.4 7 → 2 → 9 listesinde gezinmenin üç anını gösterir. Döngü üç kez döner ve çıktı `7 2 9` olur. Her turda sabit sayıda iş yapılır: bir karşılaştırma, bir yazdırma ve bir atama. *n* düğümlü listede döngü tam *n* tur döner, koşul ise *n + 1* kez denetlenir. Gezinmenin süresi bu yüzden $\Theta(n)$'dir. Ek bellek yalnız `p` işaretçisidir, yani $O(1)$'dir.

Liste uzunluğunu bulmak da aynı kalıpla yapılır; yazdırma yerine bir sayaç artırılır: `for (; p != NULL; p = p->bag) n++;`. Dizide uzunluk ya bir değişkende saklanır ya da derleme sırasında bellidir. Bağlı listede ise uzunluk ayrıca tutulmadıkça her sorulduğunda bütün liste gezilir ve $\Theta(n)$ sürer. Uzunluğun sık gerektiği uygulamalarda listeyle birlikte bir sayaç tutmak bu yüzden yaygın bir çözümdür.

> **Sık yapılan hata**
>
> Döngü koşulunu `p->bag != NULL` yazmak. Bu koşul son düğüme gelindiğinde döngüyü bitirir ve son düğümün değeri hiç yazdırılmaz. Üstelik liste boşsa `p` `NULL`'dır ve `p->bag` ifadesi programı çökertir. `p->bag != NULL` koşulu yalnız son düğümü *bulmak* istendiğinde doğrudur; o zaman da boş liste ayrıca denetlenir.

4.4 Ekleme

Bağlı listeye düğüm eklemenin dört biçimi vardır: başa, sona, bilinen bir düğümün arkasına ve sıralı listede doğru yere. Dördünde de iş, yeni bir düğüm ayırıp iki bağı doğru sırayla yazmaktan ibarettir. Fark, eklemenin yapılacağı yerin nasıl bulunduğunda ve listenin başının değişip değişmediğindedir.

Bütün ekleme işlemlerinde tek bir kural geçerlidir: **önce koru, sonra bağla**. Bir bağ alanının üzerine yazılmadan önce, o alanın gösterdiği adres başka bir yerde saklanmış olmalıdır. Yoksa o adresten sonraki bütün düğümler kaybolur. Bu kural, aşağıdaki her kodda yeni düğümün bağının listedeki bağdan *önce* yazılmasının nedenidir.

4.4.1 Başa ekleme ve adresle aktarım

Başa ekleme en basit eklemedir: yeni düğüm eski ilk düğümü gösterir ve listenin başı yeni düğüm olur. Basit olmasına rağmen C'ye özgü önemli bir soruyu gündeme getirir. Liste başı `main`'deki bir değişkendir ve fonksiyon bu değişkeni değiştirmek zorundadır. Sunumdaki kod bu yüzden listeyi adresle alır:

```
bool basa_ekle(oge_gost *bas, int x) {
    oge_gost yeni = malloc(sizeof *yeni);
    if (yeni == NULL) return false;
    yeni->deger = x;
    yeni->bag = *bas;
    *bas = yeni;
    return true;
}

oge_gost liste = NULL;
basa_ekle(&liste, 2);   // 2
basa_ekle(&liste, 7);   // 7 -> 2
```

Fonksiyonun ilk parametresi `oge_gost *bas`, yani "düğüme işaretçiye işaretçi"dir. Çağrıda `&liste` yazılır ve fonksiyona `liste` değişkeninin adresi verilir. Fonksiyonun içinde `*bas`, `main`'deki `liste` değişkeninin kendisidir. `*bas`'ı okumak `liste`'yi okumak, `*bas`'a yazmak `liste`'ye yazmaktır. Bu tekniğe **adresle aktarım** denir ve bir işaretçinin adresini tutan `bas`'a **işaretçiye işaretçi** (*pointer to pointer*) denir.

Parametre `oge_gost bas` diye değerle alınsaydı ne olurdu? Fonksiyon `liste`'nin bir kopyasını alırdı. `bas = yeni` ataması yalnız bu yerel kopyayı değiştirirdi. Fonksiyon dönünce kopya yok olurdu, `main`'deki `liste` ise eski ilk düğümü göstermeye devam ederdi. Yeni düğüm ayrılmış ve doğru bağlanmış olurdu, ama ona hiçbir değişken ulaşamazdı: hem eklenmemiş hem de sızmış bir düğüm. Derleyici bu hatayı yakalamaz, çünkü kod sözdizimi bakımından tamamen geçerlidir.

Burada iyi bir kural işe yarar. Bir yordamın parametre listesinde `*bas` biçiminde görünen değişken, yordamın içinde de `*bas` biçiminde kullanılır. Kodda `bas` çıplak hâlde, yıldızsız görünüyorsa ya bilinçli bir iş yapılıyordur ya da bir hata vardır. Bu basit denetim, adresle aktarım hatalarının çoğunu okurken yakalar.

Fonksiyonun gövdesi beş adımdan oluşur. `malloc` ile yeni düğüm istenir. Bellek ayrılamamışsa fonksiyon `false` döndürür; liste hiç değişmemiştir ve bozulmamıştır. Ardından değer yazılır. `yeni->bag = *bas` satırı eski başı korur: yeni düğüm, o ana kadar ilk olan düğümü gösterir. Son olarak `*bas = yeni` ile baş değiştirilir. `bool` dönüş tipi ve `true` / `false` değerleri `<stdbool.h>` başlık dosyasından gelir.

Son iki satırın sırası hayati önemdedir. Önce `*bas = yeni` yazılsaydı listenin başı yeni düğüm olurdu. Ardından gelen `yeni->bag = *bas` ise yeni düğümün bağına yine yeni düğümün adresini yazardı. Yeni düğüm kendini gösterir ve eski listenin tamamı kaybolurdu. Ona işaret eden tek değişken `*bas`'tı ve üzerine yazılmıştı.

Başa eklemenin maliyeti listenin uzunluğuna bağlı değildir. Bir `malloc`, bir karşılaştırma ve üç atama yapılır. Süre *O(1)*, ek bellek yeni düğümün kendisidir. Sunumdaki örnek boş listeden başlar: `basa_ekle(&liste, 2)` listeyi 2 yapar, `basa_ekle(&liste, 7)` ise 7 → 2 yapar. Başa ekleme sırayı tersine çevirir; son eklenen ilk sırada durur. Boş listede `*bas` `NULL`'dır ve `yeni->bag = NULL` olur. Böylece boş liste için ayrı bir dal yazmak gerekmez.

4.4.2 Kod izleme: `basa_ekle(&liste, 5)`

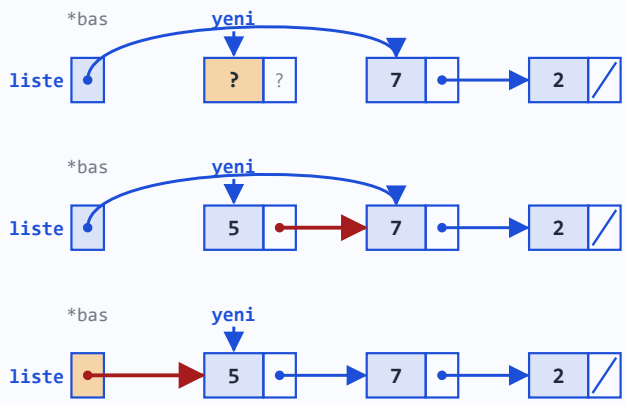
Sunum aynı fonksiyonu, yorumları açılmış hâliyle adım adım çalıştırır:

```
bool basa_ekle(oge_gost *bas, int x) {
    oge_gost yeni = malloc(sizeof *yeni);
    if (yeni == NULL) return false;  // bellek yok
    yeni->deger = x;
    yeni->bag = *bas;    // önce eski başı koru
    *bas = yeni;         // sonra başı değiştir
    return true;
}
```

Başlangıçta liste 7 → 2'dir. Tablo 4.2 her satırdan sonra bellek durumunu verir; Şekil 4.5 ise en önemli üç anı çizer.

Satır	`yeni`	`yeni->deger`	`yeni->bag`	`*bas` (= `liste`)	Liste
Çağrı	?	—	—	&düğüm(7)	7 → 2
`yeni = malloc(…)`	&yeni düğüm	? (çöp)	? (çöp)	&düğüm(7)	7 → 2
`if (yeni == NULL)`	Yanlış: bellek ayrıldı, devam.				
`yeni->deger = x`	&yeni düğüm	5	? (çöp)	&düğüm(7)	7 → 2
`yeni->bag = *bas`	&yeni düğüm	5	&düğüm(7)	&düğüm(7)	7 → 2 (yeni düğüm 7'yi gösteriyor)
`*bas = yeni`	&yeni düğüm	5	&düğüm(7)	&düğüm(5)	5 → 7 → 2
`return true`	Fonksiyon döner; `main`'deki `liste` artık 5'i gösterir.				

Tablo 4.2: `basa_ekle(&liste, 5)` izlemesi; başlangıçta liste 7 → 2.



Şekil 4.5: Başa eklemenin üç anı. Üstte `malloc` yeni düğümü ayırmıştır; alanları çöp değer (?) taşır. Ortada değer yazılmış ve `yeni->bag = *bas` ile yeni düğüm 7'yi göstermiştir; `liste` hâlâ 7'dedir. Altta `*bas = yeni` ile `main`'deki `liste` 5'e geçmiştir: 5 → 7 → 2.

Tablonun beşinci satırında ilginç bir ara durum vardır. Yeni düğüm 7'yi gösterir, ama liste hâlâ 7'den başlar. Bu anda listenin içeriği değişmemiştir; yeni düğüm yalnız hazırlanmıştır. Liste yapısı tek bir atamayla, `*bas = yeni` satırında değişir. Böyle bir düzen hata durumunda da güvenlidir. `malloc` başarısız olsaydı fonksiyon ilk denetimde dönerdi ve liste hiç dokunulmamış kalırdı.

4.4.3 Sona ekleme

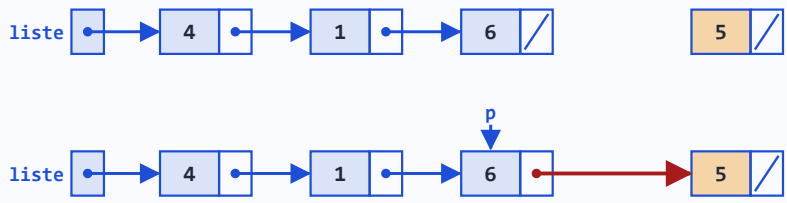
Sona eklemede yeni düğüm son düğüm olacağı için bağı `NULL`'dır. Asıl iş, mevcut son düğümü bulup onun bağını yeni düğüme çevirmektir. Liste yalnız başından tutulduğu için son düğüm ancak baştan sona yürüyerek bulunur:

```
bool sona_ekle(oge_gost *bas, int x) {
    oge_gost p, yeni = malloc(sizeof *yeni);
    if (yeni == NULL) return false;
    yeni->deger = x;
    yeni->bag = NULL;              // yeni son düğüm
    if (*bas == NULL) { *bas = yeni; return true; }
    for (p = *bas; p->bag != NULL; p = p->bag)
        ;                          // son düğümü bul
    p->bag = yeni;
    return true;
}
```

- İlk iki satır yeni düğümü hazırlar: ayırır, denetler, değerini yazar ve bağını `NULL` yapar.
- Liste boşsa (`*bas == NULL`) son düğüm yoktur. Yeni düğüm hem ilk hem son düğüm olur ve baş değişir. Bu durum yüzünden fonksiyon da listeyi adresle alır.

- Liste boş değilse `for` döngüsü `p`'yi baştan başlatır ve `p->bag != NULL` olduğu sürece ilerletir. Döngünün gövdesi boştur; tek başına duran noktalı virgül bunu açıkça gösterir. Döngü bittiğinde `p`, bağı `NULL` olan düğümü, yani son düğümü gösterir.
- `p->bag = yeni` yeni düğümü zincirin sonuna takar.

Buradaki döngü koşulu gezinmedekinden farklıdır: `p != NULL` değil, `p->bag != NULL`. Amaç listenin dışına çıkmak değil, son düğümde durmaktır. Bu koşul `p`'nin `NULL` olmadığını varsayar; boş liste denetiminin döngüden önce yapılmasının nedeni budur.



Şekil 4.6: 4 → 1 → 6 listesinin sonuna 5 eklemek. Üstte yeni düğüm hazırdır ve bağı `NULL`'dır. Altta `p` son düğümü (6) bulmuş ve `p->bag = yeni` yazılmıştır: 4 → 1 → 6 → 5.

Sona eklemenin maliyeti son düğümü bulma yürüyüşünden gelir. *n* düğümlü listede döngü *n − 1* adım atar; süre *Θ(n)*'dir. Boş listeye ekleme ise *O(1)*'dir. Bir listeyi boştan başlayıp sona ekleyerek *n* düğüme çıkarmak bu yüzden *0 + 1 + 2 + … + (n − 1) = n(n − 1)/2* adım, yani $\Theta(n^2)$ sürer. Bu maliyet sona eklemenin sık yapıldığı uygulamalarda kabul edilemez.

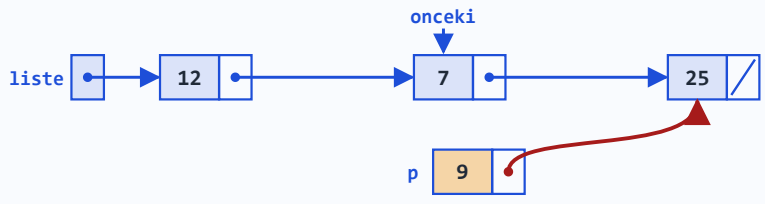
Çözüm basittir: başın yanında son düğümün adresini de tutan ikinci bir işaretçi (`son`). Son düğüm her an bilindiği için sona ekleme *O(1)* olur: `son->bag = yeni; son = yeni;`. Bedeli, her işlemde bu işaretçiyi güncel tutmaktır. Örneğin son düğüm silinirse `son` da güncellenmelidir. Bağlı kuyruk tam bu yolla gerçeklenir: öğeler sona eklenir, baştan çıkarılır ve iki işlem de *O(1)* sürer. Dairesel liste ise aynı kazancı tek bir işaretçiyle sağlar.

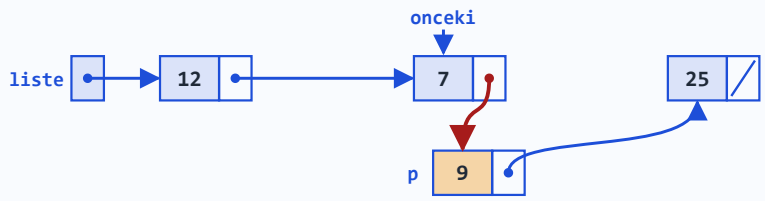
4.4.4 Araya ekleme

Eklenecek yerin hemen öncesindeki düğüm biliniyorsa ekleme yürüyüş gerektirmez. Sunumdaki fonksiyon, `onceki` düğümünün arkasına `x` değerli yeni bir düğüm ekler:

```
// onceki'nin peşine x değerli yeni düğüm ekler
oge_gost sonrasina_ekle(oge_gost onceki, int x) {
    oge_gost p = yeni_oge(x);  // malloc + denetim
    if (onceki != NULL) {      // araya ya da sona
        p->bag = onceki->bag;
        onceki->bag = p;
    }                          // yoksa tek başına
    return p;
}
```

Yeni düğüm `yeni_oge` ile ayrılır; bellek denetimi o fonksiyonun içindedir. `onceki` `NULL` değilse iki bağ yazılır. `p->bag = onceki->bag` satırı, `onceki`'nin o ana kadar gösterdiği düğümün adresini yeni düğüme kopyalar. Bu, "önce koru" adımıdır. `onceki->bag = p` ise yeni düğümü zincire takar. `onceki` son düğümse `onceki->bag` `NULL`'dır ve yeni düğüm sona eklenmiş olur; aynı kod iki durumu da kapsar. `onceki` `NULL` ise yeni düğüm hiçbir listeye bağlanmaz ve tek başına döner. Bu durumda çağıranın, dönen düğümü yeni bir listenin başı olarak kullanması beklenir.





Şekil 4.7: 12 → 7 → 25 listesinde 7'nin arkasına 9 eklemek. Üstte `p->bag = onceki->bag` ile 25'in adresi yeni düğüme kopyalanmıştır; şu anda iki bağ birden 25'i gösterir. Altta `onceki->bag = p` ile 7, yeni düğümü göstermiştir: 12 → 7 → 9 → 25.

Şekil 4.7'nin üst çizimi önemli bir ara anı gösterir: 7 ve 9 aynı anda 25'i gösterir. Bu bir hata değildir; bir düğümü birden çok bağın göstermesi serbesttir. Bu anda liste hâlâ 12 → 7 → 25'tir, çünkü baştan gezinen biri 9'a hiç uğramaz. İkinci atamadan sonra 9 zincire girer. Atamaların sırası ters olsaydı, yani önce `onceki->bag = p` yazılsaydı, 25'in adresi kaybolurdu. Ardından gelen `p->bag = onceki->bag` ise yeni düğümün bağına kendi adresini yazardı ve 9 kendini gösteren bir döngüye dönüşürdü.

Fonksiyon yalnız iki bağ yazar ve hiç döngü içermez; süresi *O(1)*'dir. Ama `onceki`'nin nasıl bulunduğu ayrı bir sorudur. Değere göre bulunacaksa bir arama gerekir ve toplam maliyet aramanın *O(n)*'i olur. Bu fonksiyon, polinom çarpımında kullanılan `YeniTerimEkle(Ust, Katsayi, Onceki)` yordamının genelleştirilmiş hâlidir: orada da yeni terim, bir önceki terimin arkasına eklenir ve önceki terim her adımda bilindiği için ekleme sabit sürer.

4.4.5 Sıralı ekleme

Değerleri küçükten büyüğe sıralı tutulan bir listeye yeni bir değer eklenirken, sıranın bozulmaması için yeni düğüm doğru yere konmalıdır. Doğru yer, yeni değerden küçük olan son düğümün hemen arkasıdır. Tek yönlü listede bir düğümün arkasına ekleme yapmak için o düğümün kendisi gerekir. Bu yüzden arama sırasında iki işaretçi birlikte yürür: `p` incelenen düğümü, `onceki` ise `p`'nin bir gerisindeki düğümü gösterir.

```
void sirali_ekle(oge_gost *bas, oge_gost yeni) {
    oge_gost onceki = NULL, p = *bas;
    while (p != NULL && p->deger < yeni->deger) {
        onceki = p;
        p = p->bag;
    }
    yeni->bag = p;
    if (onceki == NULL) *bas = yeni;  // başa
    else onceki->bag = yeni;          // araya/sona
}
```

Fonksiyon yeni düğümü hazır olarak alır; düğümün ayrılması ve değerinin yazılması çağıranın işidir. `onceki` `NULL`'dan, `p` ilk düğümden başlar. `while` döngüsü `p` listenin dışına çıkmadığı ve `p`'nin değeri yeni değerden küçük olduğu sürece iki işaretçiyi birlikte bir adım ilerletir. Önce `onceki = p`, sonra `p = p->bag` yazılır; bu sıra `onceki`'nin her zaman `p`'nin tam bir gerisinde kalmasını sağlar. Döngü bittiğinde yeni düğümün yeri `onceki` ile `p`'nin arasıdır.

Döngüden sonra önce `yeni->bag = p` yazılır. Böylece `p`'nin adresi, o düğüm ister gerçek bir düğüm ister `NULL` olsun, korunmuş olur. Ardından `onceki`'ye bakılır. `onceki` hâlâ `NULL` ise döngü hiç dönmemiştir. Yani ya liste boştur ya da yeni değer ilk düğümden küçük ya da ona eşittir; her iki durumda da yeni düğüm başa gelir ve `*bas` değişir. `onceki` `NULL` değilse yeni düğüm araya ya da sona eklenir: `onceki->bag = yeni`.

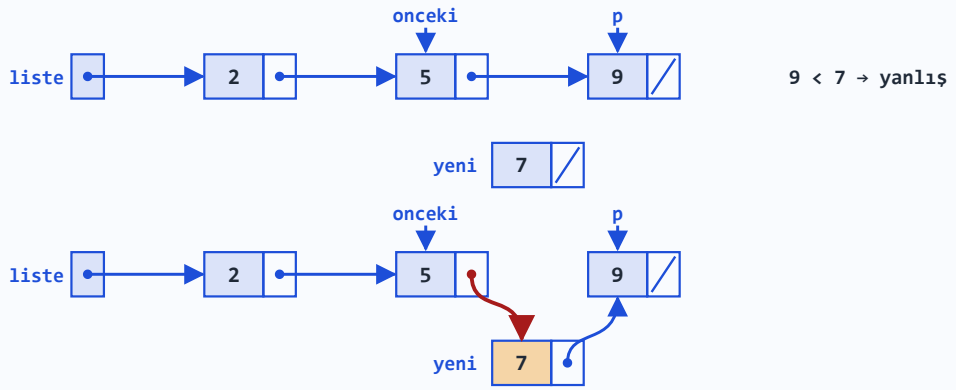
Örnek 4.1

Sıralı 2 → 5 → 9 listesine 7 eklenecektir. Yeni düğüm `malloc` ile ayrılmış, değeri 7 yazılmıştır. Tablo 4.3 döngünün her denetimini ayrı satırda verir.

Adım	`onceki`	`p`	Koşul	İşlem
Başlangıç	NULL	&düğüm(2)	—	—
1. denetim	NULL	&düğüm(2)	2 < 7: doğru	`onceki = p`, `p = p->bag`
2. denetim	&düğüm(2)	&düğüm(5)	5 < 7: doğru	`onceki = p`, `p = p->bag`

Adım	`onceki`	`p`	Koşul	İşlem
3. denetim	&düğüm(5)	&düğüm(9)	9 < 7: yanlış	Döngü biter; yer 5 ile 9 arası.
`yeni->bag = p`	&düğüm(5)	&düğüm(9)	—	7, 9'u gösterir.
`onceki == NULL` ?	&düğüm(5)	&düğüm(9)	yanlış	Baş değişmez.
`onceki->bag = yeni`	&düğüm(5)	&düğüm(9)	—	Liste: 2 → 5 → 7 → 9.

Tablo 4.3: `sirali_ekle` ile 2 → 5 → 9 listesine 7 eklemenin izlemesi.



Şekil 4.8: Sıralı eklemenin iki anı. Üstte döngü 9 < 7 yanlış olduğu için durmuştur; `onceki` 5'te, `p` 9'dadır ve yeni düğüm henüz bağlı değildir. Altta `yeni->bag = p` ve `onceki->bag = yeni` yazılmıştır: 2 → 5 → 7 → 9.

Aynı kod öteki durumları da doğru karşılar (bunlar doğrulama programında denenmiştir). 1 eklenirse ilk denetim 2 < 1 yanlış olduğu için döngü hiç dönmez; `onceki` `NULL` kalır ve yeni düğüm başa geçer. 10 eklenirse döngü bütün düğümleri geçer ve `p` `NULL` olunca durur; `yeni->bag = NULL` olur ve yeni düğüm son düğümün arkasına eklenir. Var olan bir değere eşit bir değer, örneğin 5 eklenirse, koşul `5 < 5` yanlış olduğu için döngü eski 5'in önünde durur ve yeni 5 eskisinin önüne girer. Koşul `<=` yazılsaydı yeni 5 eskisinin arkasına girerdi. Eşit değerlerin göreli sırası önemli olduğunda bu ayrıntı belirleyicidir.

Sıralı eklemenin süresi döngünün tur sayısıyla belirlenir. En iyi durumda yeni değer en küçüktür, döngü dönmez ve süre *O(1)*'dir. En kötü durumda yeni değer en büyüktür ve bütün liste geçilir: *Θ(n)*. Sıralı bir dizide doğru yer ikili aramayla *O(log n)*'de bulunabilir, ama ardından gelen kaydırma yine *O(n)*'dir. Listede ise yer bulma *O(n)*, bağlama *O(1)*'dir. Toplam maliyet iki yapıda da doğrusaldır, ama kaynağı farklıdır.

> **Sık yapılan hata**
>
> Döngü koşulunu `p->deger < yeni->deger && p != NULL` sırasıyla yazmak. `&&` işleci soldan sağa değerlendirilir. Liste sonuna gelindiğinde `p` `NULL` 'dır ve ilk koşul `NULL` 'ın içine bakmaya çalışır; program çöker. Doğru sırada (`p != NULL &&` …) ilk koşul yanlış olunca ikincisi hiç değerlendirilmez. Bu davranışa **kısa devre değerlendirme** (*short-circuit evaluation*) denir ve bağlı liste döngülerinin hepsi buna dayanır.

4.5 Silme, arama ve listeyi bırakma

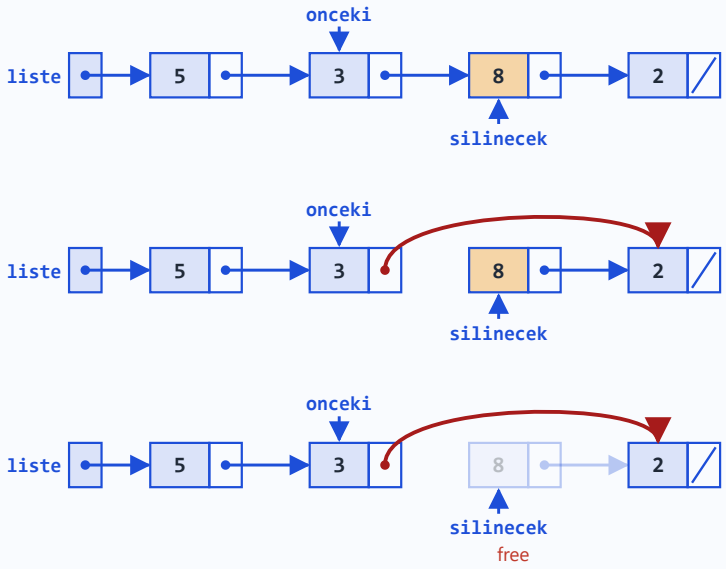
Silme, eklemenin tersidir ama bir adımı fazladır: düğüm zincirden çıkarıldıktan sonra belleği de `free` ile iade edilmelidir. Bu kesim önce sonraki düğümü silmeyi, sonra değere göre silmeyi anlatır. Ardından arama ve konuma erişim, en sonda da bütün listeyi serbest bırakan özyinelemeli ve yinelemeli yordamlar gelir.

4.5.1 Sonraki düğümü silme

Tek yönlü bağlı listede bir düğümü silmek, onu gösteren bağı değiştirmek demektir. O bağ, silinecek düğümün kendisinde değil, ondan önceki düğümdedir. Bu yüzden silme işlemi silinecek düğümün öncesini ister. Sunumdaki ilk silme fonksiyonu bu gereği doğrudan arayüzüne taşır: silinecek düğümü değil, öncesini alır.

```c
// onceki'den sonra gelen düğümü listeden ve
// bellekten siler
void sonrakini_sil(oge_gost onceki) {
    oge_gost silinecek = onceki->bag; // adresi tut
    if (silinecek == NULL) return; // silinecek yok
    onceki->bag = silinecek->bag;  // bağı düzelt
    free(silinecek);               // kalıntıyı sil
}
```

İlk satır silinecek düğümün adresini `silinecek` değişkenine alır. Bu adres saklanmasaydı, bağ değiştirildikten sonra düğüme ulaşmanın yolu kalmaz ve `free` çağrılamazdı. `onceki` son düğümse ardından gelen düğüm yoktur; fonksiyon hiçbir şey yapmadan döner. Aksi hâlde `onceki->bag = silinecek->bag` satırı silinecek düğümün üzerinden atlar: `onceki` artık silinecek düğümün sonrakini gösterir. Bu satırdan sonra düğüm listeden çıkmıştır ama hâlâ bellekte durur. Son satır onu belleğe iade eder.



Şekil 4.9: 5 → 3 → 8 → 2 listesinde 3'ten sonraki düğümü (8) silmek. Üstte `silinecek = onceki->bag` ile 8'in adresi saklanmıştır. Ortada `onceki->bag = silinecek->bag` ile 3 artık 2'yi gösterir; 8 zincirden çıkmıştır. Altta `free(silinecek)` ile 8'in belleği iade edilmiştir: 5 → 3 → 2.

Fonksiyon iki atama ve bir `free` yapar; süresi *O(1)*'dir. Ama bir sınırı vardır: baş düğüm silinemez, çünkü baş düğümün öncesi yoktur. Baş düğümü silmek için listenin başını tutan değişken değiştirilmelidir. Bu sınır iki yolla aşılır. Birincisi, fonksiyonun listeyi adresle alıp baş için ayrı bir dal içermesidir; değere göre silme fonksiyonu böyledir. İkincisi, listenin başına hiç silinmeyen bir başlık düğümü koymaktır. O zaman her gerçek düğümün bir öncesi olur ve özel durum ortadan kalkar.

4.5.2 Değere göre silme

Çoğu uygulamada silinecek düğüm adresiyle değil değeriyle bilinir: "listeden 1'i sil". Bu durumda önce düğüm bulunur, sonra silinir. Arama sırasında, sıralı eklemedeki gibi, bir geriden gelen `onceki` işaretçisi tutulur.

```c
bool sil(oge_gost *bas, int x) {
    oge_gost onceki = NULL, p = *bas;
    while (p != NULL && p->deger != x) {
        onceki = p;
        p = p->bag;
    }
    if (p == NULL) return false;        // x yok
    if (onceki == NULL) *bas = p->bag;  // baştaki
    else onceki->bag = p->bag;          // atla
    free(p);
    return true;
}
```

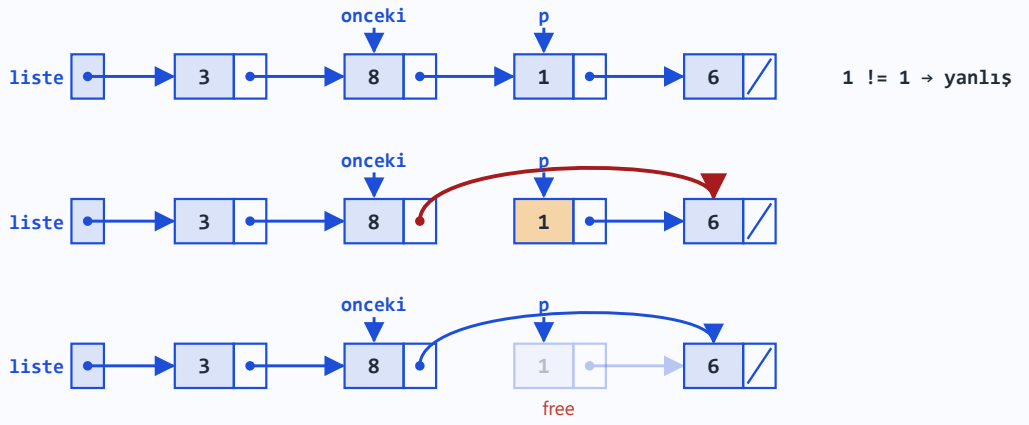
- Döngü, `p` listenin sonuna gelmediği ve `p` 'nin değeri `x` 'ten farklı olduğu sürece iki işaretçiyi birlikte ilerletir.
- Döngü `p == NULL` ile bittiyse `x` listede yoktur; fonksiyon `false` döndürür ve liste değişmez.
- `onceki == NULL` ise döngü hiç dönmemiştir, yani `x` ilk düğümdedir. Baş değişir: `*bas = p->bag` . Liste tek düğümlüyse `p->bag` `NULL` 'dır ve liste boşalır.
- Aksi hâlde `onceki->bag = p->bag` ile `p` 'nin üzerinden atlanır.
- Her iki dalda da `p` artık listede değildir ve `free(p)` ile bırakılır. Bağ değişikliği `free` 'den önce yapılır, çünkü `p->bag` 'in okunması gerekir.

Örnek 4.2

Liste 3 → 8 → 1 → 6'dır ve `sil(&liste, 1)` çağrılır. Tablo 4.4 her adımı verir. Döngü iki kez döner, üçüncü denetimde `1 != 1` yanlış olduğu için durur. Bu anda `p` 1'i, `onceki` 8'i gösterir. `onceki` `NULL` olmadığı için baş değişmez; 8'in bağı 6'ya çevrilir ve 1'in düğümü bırakılır. Sonuç 3 → 8 → 6'dır.

Adım	`onceki`	`p`	Koşul / işlem
Başlangıç	NULL	&düğüm(3)	—
1. denetim	NULL	&düğüm(3)	3 != 1: doğru → `onceki = p` , `p = p->bag`
2. denetim	&düğüm(3)	&düğüm(8)	8 != 1: doğru → `onceki = p` , `p = p->bag`
3. denetim	&düğüm(8)	&düğüm(1)	1 != 1: yanlış → döngü biter
`p == NULL` ?	&düğüm(8)	&düğüm(1)	yanlış: `x` listede var
`onceki == NULL` ?	&düğüm(8)	&düğüm(1)	yanlış: baş silinmiyor
`onceki->bag = p->bag`	&düğüm(8)	&düğüm(1)	8 artık 6'yı gösterir; 1 zincirden atlandı
`free(p)`	&düğüm(8)	(sarkan)	Düğüm iade edildi; liste 3 → 8 → 6, sonuç `true`

Tablo 4.4: `sil(&liste, 1)` izlemesi; başlangıçta liste 3 → 8 → 1 → 6.



Şekil 4.10: `sil(&liste, 1)` çağrısının üç anı. Üstte döngü 1 != 1 yanlış olduğu için durmuştur; `onceki` 8'de, `p` 1'dedir. Ortada `onceki->bag = p->bag` ile 8 artık 6'yı gösterir. Altta `free(p)` sonrası liste 3 → 8 → 6'dır.

Fonksiyonun maliyeti aramadan gelir. `x` , *k*. sıradaki düğümdeyse (0'dan sayarak) döngü *k* tur döner; `x` listede yoksa *n* tur döner. En kötü durumda süre $\Theta(n)$'dir. Bulduktan sonraki bağ değişikliği ve `free` sabit sürer. Doğrulama programında aynı listeden önce 1, sonra baştaki 3 silinmiş, en sonda listede olmayan 42 aranmıştır. Üç çağrı sırasıyla 2, 0 ve 2 tur dönmüş; ilk ikisi `true` , sonuncusu `false` döndürmüştür. Son çağrıda liste 8 → 6 olduğu için 42'yi aramak iki tur sürmüştür.

`free(p)` çağrısı unutulursa ne olur? Düğüm listeden çıkmıştır ve onu gösteren tek değişken yerel `p` 'dir. Fonksiyon dönünce `p` da yok olur. Düğüm bellekte kalır ama hiçbir işaretçi ona ulaşamaz: bu, tanımdaki bellek sızıntısıdır. Liste kodunda sızıntıların büyük kısmı tam olarak silme fonksiyonlarındaki unutulmuş bir `free` 'den doğar.

4.5.3 Arama ve konuma erişim

Arama, gezinmenin erken biten bir biçimidir. Değer bulunduğu anda durulur ve düğümün adresi döndürülür. Konuma erişim ise belirli sayıda adım atar. Sunumdaki iki fonksiyon bu iki işi yapar:

```
oge_gost ara(oge_gost p, int x) {
    while (p != NULL && p->deger != x)
        p = p->bag;
    return p;               // bulunamazsa NULL
}

oge_gost konum(oge_gost p, int i) {  // 0 tabanlı
    while (p != NULL && i-- > 0)
        p = p->bag;
    return p;               // i >= uzunluk ise NULL
}
```

`ara`, `p` listenin sonuna gelmediği ve aranan değeri taşımadığı sürece ilerler. Döngü iki nedenle biter. `p` bir düğümde durmuşsa o düğüm aranan değeri taşır ve adresi döner. `p` `NULL`'a ulaşmışsa değer listede yoktur ve `NULL` döner. Çağıran bu yüzden dönüş değerini kullanmadan önce `NULL` olup olmadığına bakmalıdır.

Koşuldaki sıra burada da belirleyicidir: önce `p != NULL`, sonra `p->deger != x`. Sıra ters olsaydı, değer listede yokken son düğümden sonra `p` `NULL` olur ve `p->deger` okunmaya çalışılırdı. Kısa devre değerlendirme sayesinde doğru sırada ilk koşul yanlış olunca ikincisi hiç değerlendirilmez.

`konum` fonksiyonu 0 tabanlı `i`. düğümü bulur. `i-- > 0` ifadesi önce `i`'yi 0 ile karşılaştırır, sonra `i`'yi bir azaltır. `i = 2` ile çağrılırsa karşılaştırmalar 2 > 0, 1 > 0 ve 0 > 0 olur; ilk ikisi doğru olduğu için `p` iki adım ilerler. Liste `i`'den kısaysa `p` yolda `NULL` olur ve fonksiyon `NULL` döndürür. Doğrulama programında 4 → 1 → 6 → 5 listesinde `konum(liste, 0)` ile `konum(liste, 3)` sırasıyla 4'ü ve 5'i, `konum(liste, 4)` ise `NULL`'ı vermiştir.

İki fonksiyonun maliyeti, bağlı listenin diziye göre en zayıf yanını açıkça gösterir. `ara` en kötü durumda $\Theta(n)$, `konum(p, i)` ise $\Theta(i)$ sürer. Dizide `a[i]` tek bir adres hesabıyla $O(1)$'de okunur. Bu yüzden "*i*. öğeyi getir" işleminin sık yapıldığı bir programda liste kötü bir seçimdir. Örneğin bir döngüde `konum(liste, i)` ile $i = 0, 1, \ldots, n-1$ için bütün öğeleri okumak $0 + 1 + \ldots + (n-1)$, yani $\Theta(n^2)$ adım sürer. Aynı iş tek bir gezinmeyle $\Theta(n)$'de yapılabilir.

Liste sıralıysa arama biraz iyileştirilebilir. `p->deger > x` görüldüğü anda durulur, çünkü daha ilerideki bütün değerler de `x`'ten büyüktür. Değer listede yoksa bu, ortalamada aramayı yarıya indirir. Ama en kötü durum yine $\Theta(n)$'dir. Sıralı dizide kullanılan ikili arama ise bağlı listede işe yaramaz. İkili arama her adımda ortadaki öğeye bakar. Listede ortadaki öğeye ulaşmak için bile yarım listeyi yürümek gerekir. Toplam iş $n/2 + n/4 + \ldots$, yani yine $\Theta(n)$ olur ve karşılaştırma sayısının azalması bir şey kazandırmaz.

4.5.4 Listeyi serbest bırakma: bir sınav sorusu

Bir listeyle iş bittiğinde listenin bütün düğümleri belleğe iade edilmelidir. Bunun için her düğüm tek tek bırakılır. Sunum bu işi eski bir sınav sorusu üzerinden ele alır. Soruda iki yordam verilir ve hangisinin doğru olduğu, öteki yordamın çağıran fonksiyondaki listeye ne yaptığı sorulur. Soldaki sürüm öğrencilerin sık yazdığı hatalı biçimdir:

```
void serbest_birak( list_gost *list ){
    oge_gost p = *list;
    if( *list ){
        p = *list->bag;
        free(*list);
        serbest_birak(&p);
}}
```

Sağdaki sürüm doğrudur:

```
void serbest_birak( list_gost *list ){
    oge_gost p = *list;
    if( p ){
        *list = p->bag;
        free(p);
        serbest_birak(list);
}}
```

Soldaki sürümün iki ayrı sorunu vardır. Birincisi bir yazım hatasıdır: `*list->bag` ifadesi derlenmez. `->` işlecinin önceliği `*`'dan yüksektir, bu yüzden ifade `*(list->bag)` diye okunur. `list` bir düğümü değil, bir işaretçiyi gösterir; `list->bag` anlamsızdır. Derleyici "member reference base type is not a structure or union" hatası verir (doğrulama programında bu hata alınmıştır). Kastedilen ifade `(*list)->bag`'dir.

Yazım hatası düzeltilse bile ikinci, daha derin bir sorun kalır. Soldaki sürüm özyinelemeli çağrıyı `&p` ile, yani *yerel* `p` değişkeninin adresiyle yapar. İkinci çağrı bu yüzden `main`'deki `liste`'yi değil, birinci çağrının yerel `p`'sini görür. Bütün düğümler yine bırakılır, ama `main`'deki `liste` hiç değişmez. Çağrılardan dönüldüğünde `liste` hâlâ eski ilk düğümün adresini tutar. O düğüm artık iade edilmiştir: `liste` sarkan bir işaretçi olmuştur. Doğrulama programında düzeltilmiş soldaki sürüm 4 → 9 listesiyle çağrılmış, dönüşte `liste == NULL` denetimi yanlış çıkmış ve `liste->deger` okunabilmiştir. Bu okuma tanımsız davranıştır; o çalıştırmada eski değer olan 4'ü vermesi, programın doğru olduğunu değil, hatanın gizli kaldığını gösterir.

Sağdaki sürüm her çağrıda aynı `list` adresini kullanır. `*list = p->bag` satırı `main`'deki `liste`'yi doğrudan bir sonraki düğüme ilerletir; ardından eski baş bırakılır. Son çağrıda `*list` `NULL`'dır ve özyineleme durur. Dönüşte `main`'deki `liste` `NULL`'dır; ortada sarkan işaretçi kalmaz.

Bu soru, kod yazan kişinin kendi koduna fazla güvenmesinin ne kadar kolay olduğunu da gösterir. Soldaki sürüm ilk bakışta makul görünür ve yazım hatası düzeltilince sorunsuz çalışıyor izlenimi verir. Hatayı görmenin yolu, her işaretçinin hangi değişkenin adresini tuttuğunu tek tek izlemektir. Bir sonraki alt kesimdeki izleme bunu doğru sürüm için yapar.

4.5.5 Kod izleme: `serbest_birak(&liste)`

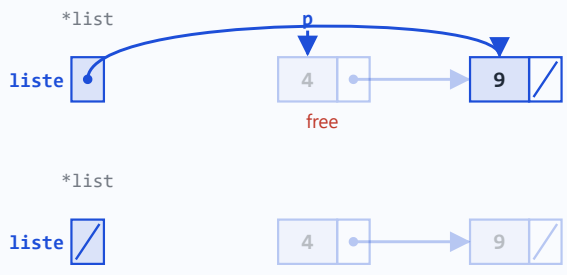
Sunum, sağdaki sürümü düzenlenmiş ve yorumlanmış hâliyle izler:

```
void serbest_birak(oge_gost *list) {
    oge_gost p = *list;
    if (p) {
        *list = p->bag;      // önce başı ilerlet
        free(p);             // sonra eski başı bırak
        serbest_birak(list); // aynı adresle devam
    }
}
```

Liste 4 → 9'dur. Tablo 4.5 üç çağrının her birinde değişkenlerin durumunu verir. Her çağrının kendi `p`'si vardır, ama üç çağrının `list` parametresi aynı adresi, `main`'deki `liste`'nin adresini tutar.

Çağrı	Satır	`p`	`*list` (= `liste`)	Açıklama
1	`p = *list`	&düğüm(4)	&düğüm(4)	Bırakılacak düğüm var.
1	`*list = p->bag`	&düğüm(4)	&düğüm(9)	`main`'deki `liste` ilerledi.
1	`free(p)`	(sarkan)	&düğüm(9)	Düğüm(4) iade edildi; `p` artık kullanılmaz.
2	`p = *list`	&düğüm(9)	&düğüm(9)	Aynı adresle ikinci çağrı.
2	`*list = p->bag`	&düğüm(9)	NULL	`liste` artık `NULL`.
2	`free(p)`	(sarkan)	NULL	Düğüm(9) iade edildi.
3	`p = *list`	NULL	NULL	`if (p)` yanlış: taban durumu, çağrı döner.
2, 1	dönüş	—	NULL	Yapılacak iş kalmadı; çağrılar sırayla döner.

Tablo 4.5: `serbest_birak(&liste)` izlemesi; başlangıçta liste 4 → 9.



Şekil 4.11: Serbest bırakmanın iki anı. Üstte birinci çağrıda `liste` 9'a ilerlemiş, düğüm(4) `free` ile iade edilmiştir. Altta bütün düğümler bırakılmış ve `main`'deki `liste` `NULL` olmuştur.

Fonksiyonun gövdesindeki sıra, silme fonksiyonlarındaki kuralın aynısıdır. Önce `*list = p->bag` ile bir sonraki düğümün adresi güvenli bir yere alınır, sonra eski baş bırakılır. Sıra ters olsaydı, yani önce `free(p)` yazılsaydı, ardından gelen `p->bag` okuması iade edilmiş bellekten yapılırdı. `free`'den sonra `p` artık kullanılmamalıdır; izleme tablosunda bu yüzden "sarkan" diye işaretlenmiştir.

n düğümlü bir liste için fonksiyon $n + 1$ kez çağrılır: her düğüm için bir çağrı ve `NULL`'ı gören son çağrı. Her çağrı sabit iş yapar; süre $\Theta(n)$'dir. Ama özyineleme bedava değildir. Her çağrı, dönüş adresini, parametreyi ve yerel `p`'yi tutan bir çerçeve için çağrı yığınında yer açar. En derin anda $n + 1$ çerçeve birden yığındadır ve ek bellek $\Theta(n)$ olur.

4.5.6 Döngüyle boşaltma

Çağrı yığını sınırlı bir bellek bölgesidir. Boyutu işletim sistemine ve derleyici ayarlarına göre değişir, ama genellikle birkaç megabaytı geçmez. Bir milyon düğümlü bir listede özyinelemeli sürüm bir milyon çerçeve açar ve yığını aşabilir. Buna **yığın taşması** (*stack overflow*) denir. Doğrulama programında bu durum denenmiştir. Kod eniyilemesiz (`-O0`) derlendiğinde 100.000 düğümlü liste sorunsuz bırakılmış, 1.000.000 düğümlü liste ise yığın taşması hatasıyla (`STATUS_STACK_OVERFLOW`) çökmüştür. Aynı kod `-O2` ile derlendiğinde bir milyon düğüm sorunsuz bırakılmıştır. Bunun nedeni, özyinelemeli çağrının fonksiyonun son işi olmasıdır. Derleyici böyle bir **kuyruk özyinelemesini** (*tail recursion*) bir döngüye çevirebilir. Ama C standardı bu dönüşümü garanti etmez; güvenli yol, döngüyü açıkça yazmaktır.

```c
void liste_bosalt(oge_gost *list) {
    while (*list != NULL) {
        oge_gost p = *list;
        *list = p->bag;     // önce ilerle
        free(p);            // sonra bırak
    }
}
```

Döngü, liste boşalana kadar döner. Her turda baştaki düğümün adresi yerel `p`'ye alınır. `*list = p->bag` ile liste bir düğüm ilerler. Ardından `p` bırakılır. Bu, özyinelemeli sürümün gövdesinin bir döngünün içine konmuş hâlidir. Süre yine $\Theta(n)$'dir, ama ek bellek yalnız `p`'dir: $O(1)$. Döngü bittiğinde `*list` `NULL`'dır, yani çağıranın değişkeni de temizlenmiştir.

Özyinelemeli ve yinelemeli sürümlerin ortak kuralı şudur: `free`'den önce sonraki adres alınır; `free` edilmiş bir düğümün bağ alanı asla okunmaz. Bu kural notun doğrulama programında bütün silme ve bırakma fonksiyonlarına uygulanmış, programın sonunda `malloc` ve `free` çağrıları sayılmıştır. 84 `malloc`'a karşılık 84 `free` yapılmış, yani hiçbir düğüm sızmamıştır.

> **Sık yapılan hata**
>
> Boşaltma döngüsünü `while (*list != NULL) { free(*list); *list = (*list)->bag; }` diye yazmak. İkinci satır, az önce bırakılan düğümün bağ alanını okur. Program çoğu çalıştırmada beklenen sonucu verir, çünkü iade edilen blok hemen başka bir işe verilmemiştir. Ama başka bir iş parçacığı ya da `malloc`'un iç kayıtları o bloğa yazmışsa döngü rastgele bir adrese atlar. Bu tür hatalar adres denetleyici (*AddressSanitizer*) gibi araçlarla yakalanır; gözle fark edilmesi zordur.

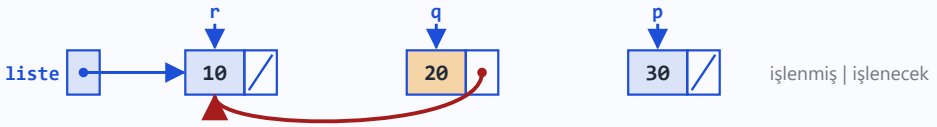
4.6 Klasik yordamlar

Bağlı listelerle ilgili bazı problemler sınavlarda, mülakatlarda ve gerçek kodda o kadar sık çıkar ki adları klasikleşmiştir. Bu kesimdeki dört yordam aynı ortak özelliği taşır: listeyi en çok bir kez gezer, yeni düğüm ayırmaz ve yalnız birkaç işaretçiyle, yani *O(1)* ek bellekle çalışır. Hepsinin özü, birden çok işaretçiyi belirli bir düzende birlikte yürütmektir.

4.6.1 Listeyi ters çevirme

Sunum bu yordamı eski bir sınav sorusuyla tanıtır: `list` işaretçisiyle gösterilen listenin bağ alanlarını tersine çeviren `liste_devrik_al` yordamı yazılacaktır. 10 → 20 → 30 listesi 30 → 20 → 10 olmalıdır. Sorunun anahtar sözcüğü "bağ alanları"dır. Değerleri bir diziye kopyalayıp ters sırayla geri yazmak ya da yeni bir liste kurup düğümleri başa ekleyerek taşımak da sonuç verir. Ama istenen bu değildir. İstenen, hiçbir düğüm ayırmadan, hiçbir değeri taşımadan, yalnız bağların yönünü çevirmektir. Yeni düğüm, kopya, `malloc` ya da ek bellek yoktur; işi yalnız işaretçiler görür.

Fikir şudur: liste baştan sona bir kez gezilir ve her düğümün bağı, bir önceki düğümü gösterecek biçimde çevrilir. Ama bir düğümün bağı çevrildiği anda o düğümden sonraki düğüme giden yol kaybolur. Bu yüzden çevirmeden önce sonraki düğümün adresi saklanmalıdır. Ayrıca çevrilen bağın yazılacağı değer, yani önceki düğümün adresi de elde tutulmalıdır. Toplam üç işaretçi gerekir.



Şekil 4.12: Ters çevirmenin bir ara durumu. `r` ile `q` arasındaki bağ çevrilmiştir (işlenmiş kısım); `p` 'den sonrası henüz eski yönündedir (işlenecek kısım).

Şekil 4.12 bu üç işaretçinin rolünü gösterir. `r` en gerideki işaretçidir ve tersine çevrilmiş kısmın başını, yani çevrilen bağın göstereceği düğümü tutar. `q` ortadadır ve bu turda bağı çevrilecek düğümü gösterir. `p` en öndedir ve henüz işlenmemiş listenin başını tutar. Her turda bir bağ çevrilir ve üç işaretçi birlikte bir adım ilerler.

```
void liste_devrik_al(oge_gost *list) {
    oge_gost r, p = *list, q = NULL;
    while (p) {
        r = q;          // r, q'yu takip eder
        q = p;          // q, p'yi takip eder
        p = p->bag;     // p'yi ilerlet
        q->bag = r;     // bağı tersine çevir
    }
    *list = q;          // son öğe liste başı
}
```

Başlangıçta `p` listenin başını, `q` `NULL` 'ı gösterir; `r` 'nin değeri henüz önemsizdir. `q` 'nun `NULL` olması rastgele değildir: ilk düğüm ters listede son düğüm olacağı için bağı `NULL` olmalıdır ve bu değer ilk turda `r` üzerinden ona yazılır. Döngünün her turu dört atama yapar:

1. `r = q` : `r` , bir önceki turda çevrilen düğüme geçer. Bu düğüm, bu turda çevrilecek bağın yeni hedefidir.
2. `q = p` : `q` , bu turda bağı çevrilecek düğüme geçer.
3. `p = p->bag` : `p` bir ilerler. Bu atama bağ çevrilmeden *önce* yapılır; yoksa kalan liste kaybolurdu.
4. `q->bag = r` : `q` 'nun bağı geriye, `r` 'ye çevrilir.

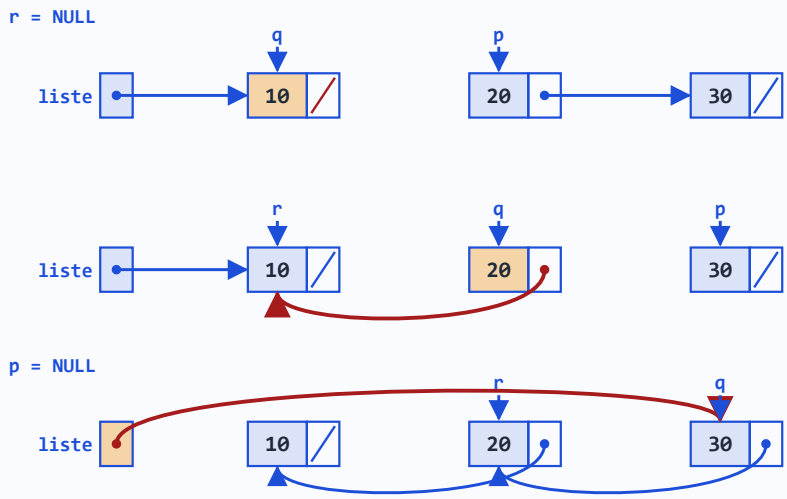
Döngü `p` `NULL` olunca biter. Bu anda `q` eski listenin son düğümünü gösterir. Bu düğüm ters listenin ilk düğümüdür ve `*list = q` ile listenin başı yapılır. Listenin başı değiştiği için fonksiyon listeyi adresle, `oge_gost *list` olarak alır.

4.6.2 Kod izleme: `liste_devrik_al(&liste)`

Liste 10 → 20 → 30'dur. Tablo 4.6 her turun sonundaki durumu verir. Tablonun son sütunu, o ana kadar çevrilmiş bağları gösterir.

An	`r`	`q`	`p`	Bu turda çevrilen bağ	Bağların durumu
Başlangıç	?	NULL	&düğüm(10)	—	10 → 20 → 30
1. tur	NULL	&düğüm(10)	&düğüm(20)	10 → NULL	NULL ← 10 · 20 → 30
2. tur	&düğüm(10)	&düğüm(20)	&düğüm(30)	20 → 10	NULL ← 10 ← 20 · 30
3. tur	&düğüm(20)	&düğüm(30)	NULL	30 → 20	NULL ← 10 ← 20 ← 30
`*list = q`	&düğüm(20)	&düğüm(30)	NULL	—	Liste: 30 → 20 → 10

Tablo 4.6: `liste_devrik_al` izlemesi; başlangıçta liste 10 → 20 → 30. Değerler her turun sonundaki durumdur.



Şekil 4.13: Ters çevirmenin üç anı. Üstte birinci turun sonunda düğüm(10) `NULL`'ı gösterir. Ortada ikinci turun sonunda düğüm(20) düğüm(10)'u gösterir; `p` 30'dadır. Altta `*list = q` ile eski son düğüm yeni baş olmuştur: 30 → 20 → 10.

Birinci turda `r` `NULL` değerini alır ve 10'un bağına yazılır. Böylece 10, ters listenin son düğümü olur. İkinci turda 20'nin bağı 10'a, üçüncü turda 30'un bağı 20'ye çevrilir. Her turun ortasında bir an vardır ki liste iki parçaya ayrılmıştır: `q`'dan geriye doğru çevrilmiş kısım ve `p`'den ileriye doğru çevrilmemiş kısım. İki parçanın da başı bir işaretçide tutulduğu için hiçbir düğüm kaybolmaz. Doğrulama programında fonksiyon aynı listeye iki kez uygulanmış ve liste ilk hâline dönmüştür. Boş listede döngü hiç dönmez ve `*list = q` ataması `NULL` yazar; boş liste boş kalır.

n düğümlü listede döngü n tur döner ve her tur dört atama yapar. Süre $\Theta(n)$, ek bellek üç işaretçi, yani $O(1)$'dir. Yordam **yerinde** (*in-place*) çalışır: yeni düğüm ayırmaz, kopya oluşturmaz. Bu yüzden bellek yetmezliğinden başarısız olamaz; ters çevirme bellek dolu olsa bile yapılabilir.

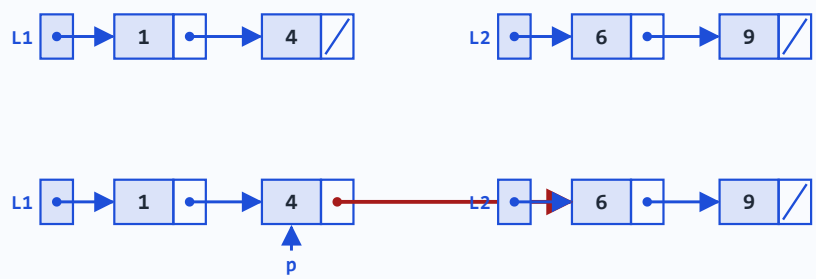
> **Sık yapılan hata**
>
> Döngüdeki `p = p->bag` ile `q->bag = r` satırlarının yerini değiştirmek. `q` ile `p` bu anda aynı düğümü gösterir. Önce `q->bag = r` yazılırsa `p->bag` de değişmiş olur; ardından gelen `p = p->bag`, `p`'yi ileri değil geriye, `r`'ye götürür. İlk turda `r` `NULL` olduğu için döngü hemen biter ve liste yalnız ilk düğümden ibaret kalır. Geri kalan bütün düğümler kaybolur.

4.6.3 İki listeyi uç uca birleştirme

İki listeyi uç uca eklemek (*concatenate*), birinci listenin son düğümünü ikinci listenin ilk düğümüne bağlamaktan ibarettir. Hiçbir düğüm kopyalanmaz.

```
// L1'in sonuna L2'yi bağlar, başı döndürür
oge_gost birlestir(oge_gost L1, oge_gost L2) {
    oge_gost p;
    if (L1 == NULL) return L2;
    if (L2 != NULL) {
        for (p = L1; p->bag != NULL; p = p->bag)
            ;                   // L1'in son düğümü
        p->bag = L2;
    }
    return L1;
}
```

Birinci liste boşsa sonuç ikinci listedir ve fonksiyon `L2`'yi döndürür. İkinci liste boşsa yapılacak iş yoktur. İkisi de boş değilse `for` döngüsü `p`'yi `L1`'in son düğümüne götürür ve `p->bag = L2` yazılır. Fonksiyon sonuç listesinin başını döndürür. Bu tasarımın sona ekleme fonksiyonundan farkı, listeyi adresle değil değerle alıp yeni başı dönüş değeriyle vermesidir. Çağıran yeni başı kendi değişkenine atar: `liste = birlestir(L1, L2);`. Her iki tasarım da doğrudur; adresle aktarım daha az yazım, dönüş değeri ise daha açık bir arayüz sağlar.



Şekil 4.14: `L1` = 1 → 4 ve `L2` = 6 → 9 listelerini birleştirmek. Üstte iki ayrı liste vardır. Altta `p` `L1`'in son düğümünü (4) bulmuş ve `p->bag = L2` yazılmıştır: 1 → 4 → 6 → 9.

Maliyet, birinci listenin son düğümünü bulmaktan gelir: $\Theta(|L1|)$. İkinci listenin uzunluğu önemli değildir. İki dizinin birleştirilmesinde ise öğelerin yeni bir diziye kopyalanması gerekir ve süre $\Theta(n + m)$ olur. Birinci listenin son düğümü bir `son` işaretçisinde tutuluyorsa birleştirme $O(1)$'e iner. Dairesel listede bu iş yine $O(1)$'dir, çünkü dairesel liste zaten son düğümden tutulur.

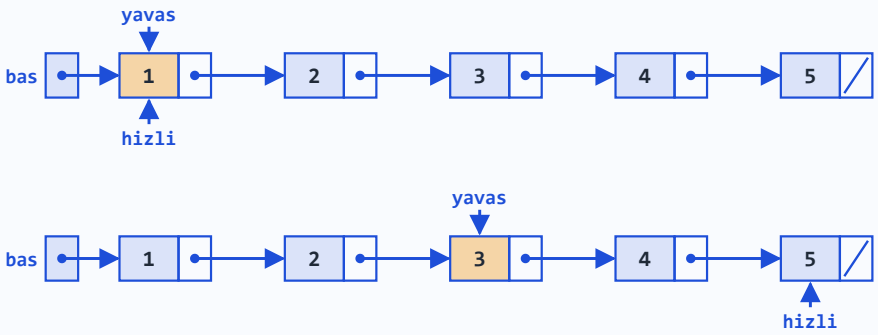
Birleştirmenin bir yan etkisi vardır: işlemden sonra `L2`'nin düğümleri iki listenin ortak malıdır. `L2` değişkeni hâlâ 6'yı gösterir ve `L2` üzerinden yapılan bir değişiklik birleşik listeyi de değiştirir. Özellikle bırakma sırasında dikkatli olmak gerekir. Birleşik liste bırakıldıktan sonra `L2` de bırakılmaya çalışılırsa aynı düğümler ikinci kez `free` edilir. Birleştirmeden sonra `L2`'yi `NULL` yapmak bu hatayı önler.

4.6.4 Orta düğüm

Bir listenin ortadaki düğümünü bulmanın ilk akla gelen yolu iki geçiştir: önce uzunluk n sayılır, sonra baştan $n/2$ adım yürünür. Bu yol doğrudur ve $\Theta(n)$ sürer. Ama aynı iş tek geçişte de yapılabilir. Bunun için iki işaretçi baştan birlikte yola çıkar: **yavaş işaretçi** her turda bir adım, **hızlı işaretçi** iki adım atar. Hızlı işaretçi listenin sonuna vardığında yavaş işaretçi yolun tam yarısındadır.

```
oge_gost orta(oge_gost bas) {
    oge_gost yavas = bas, hizli = bas;
    while (hizli != NULL && hizli->bag != NULL) {
        yavas = yavas->bag;        // 1 adım
        hizli = hizli->bag->bag;   // 2 adım
    }
    return yavas;
}
```

Döngü koşulu hızlı işaretçinin iki adım atabilmesini güvenceye alır. `hizli != NULL` denetimi çift uzunluktaki listede hızlı işaretçinin son düğümü aşıp `NULL`'a düşmesini yakalar. `hizli->bag != NULL` ise tek uzunluktaki listede hızlı işaretçinin son düğümde durmasını yakalar. İkinci koşul, `hizli->bag->bag` okumasının `NULL` bir işaretçi üzerinden yapılmasını da önler.



Şekil 4.15: 1 → 2 → 3 → 4 → 5 listesinde orta düğüm. Üstte iki işaretçi de baştadır. Altta `hizli` son düğüme (5) varmıştır; `hizli->bag == NULL` olduğu için döngü biter ve `yavas` ortadaki düğümü (3) gösterir.

Beş düğümlü 1 → 2 → 3 → 4 → 5 listesinde döngü iki tur döner. Birinci turda `yavas` 2'ye, `hizli` 3'e geçer. İkinci turda `yavas` 3'e, `hizli` 5'e geçer. 5'in bağı `NULL` olduğu için döngü biter ve fonksiyon 3'ü döndürür. Çift uzunlukta iki orta vardır. 1 → 2 → 3 → 4 listesinde döngü yine iki tur döner: `yavas` 2'ye ve 3'e, `hizli` 3'e ve `NULL`'a geçer. Fonksiyon ikinci ortayı, 3'ü döndürür. İlk orta (2) istenirse koşula `hizli->bag->bag != NULL` eklenir; o zaman hızlı işaretçi bir adım önce durur. Bu değerlerin dördü de doğrulama programında elde edilmiştir.

n düğümlü listede döngü $\lfloor n/2 \rfloor$ tur döner. Hızlı işaretçi toplam yaklaşık n, yavaş işaretçi $n/2$ adım atar; toplam bağ okuma sayısı iki geçişli çözümle aynı mertebededir. Kazanç tek geçiştir: liste bir kez gezilir, uzunluğun önceden bilinmesi gerekmez. Süre $\Theta(n)$, ek bellek $O(1)$'dir. Aynı "yavaş ve hızlı işaretçi" fikri, bir sonraki problemde çok daha güçlü bir sonuç verir.

4.6.5 Floyd'un döngü tespiti

Doğru kurulmuş bir bağlı listede son düğümün bağı `NULL`'dır. Hatalı bir bağ bu düzeni bozabilir. Örneğin son düğümün bağı yanlışlıkla ortadaki bir düğüme yazılırsa (`son->bag = ortadaki`) liste bir **döngüye** (*cycle*) girer. Böyle bir listede `p = p->bag` adımları hiçbir zaman `NULL`'a ulaşmaz. `yazdir` sonsuza kadar yazar, uzunluk sayan döngü hiç bitmez. Bir listede döngü olup olmadığını anlamaya **döngü tespiti** (*cycle detection*) denir.

İlk akla gelen çözüm, ziyaret edilen düğümlerin adreslerini bir kümede tutmaktır. Her adımda yeni düğümün kümede olup olmadığına bakılır; varsa döngü bulunmuştur. Bu çözüm çalışır ama $O(n)$ ek bellek ister. Robert W. Floyd'a atfedilen yöntem aynı işi $O(1)$ ek bellekle yapar. Yöntem "tavşan ve kaplumbağa" adıyla da bilinir: kaplumbağa (`yavas`) her turda bir adım, tavşan (`hizli`) iki adım atar.

```
bool dongu_var(oge_gost bas) {
    oge_gost yavas = bas, hizli = bas;
    while (hizli != NULL && hizli->bag != NULL) {
        yavas = yavas->bag;         // 1 adım
        hizli = hizli->bag->bag;    // 2 adım
        if (yavas == hizli) return true;
    }
    return false;
}
```

Kodun iskeleti orta düğüm fonksiyonunun aynısıdır; tek fark, her turun sonundaki `yavas == hizli` denetimidir. Burada düğümlerin değerleri değil adresleri karşılaştırılır. İki ayrı düğümün değeri aynı olabilir; ama iki işaretçi aynı adresi tutuyorsa aynı düğümdedir. Liste döngüsüzse tavşan kaplumbağadan hızlı olduğu için ondan önce `NULL`'a varır ve fonksiyon `false` döndürür. Liste döngülüyse iki işaretçi de döngüye girer ve orada sonsuza kadar döner. Döngünün içinde tavşan her turda kaplumbağaya bir adım yaklaşır; bu yüzden mutlaka buluşurlar.

Teorem 4.1 (Tavşan ile kaplumbağanın buluşması)

Listenin başından döngünün ilk düğümüne kadar μ adım olsun ve döngü λ düğümden oluşsun. `dongu_var` fonksiyonunda iki işaretçi, döngüye girişten sonra en çok λ tur içinde aynı düğümde buluşur. Toplam tur sayısı $\mu + \lambda$'yı aşmaz; bu da listedeki farklı düğüm sayısı n'ye eşittir.

Kanıt. t turdan sonra kaplumbağa baştan t, tavşan $2t$ adım uzaktadır. $t \geq \mu$ olduğu anda ikisi de döngünün içindedir. Döngü içinde konumlar λ'ya göre kalanla ölçülür. Tavşanın kaplumbağanın önündeki uzaklığı, döngü boyunca ileri sayılarak d olsun; $0 \leq d < \lambda$. Her turda tavşan iki, kaplumbağa bir adım attığı için bu uzaklık 1 azalır. d turdan sonra uzaklık 0 olur ve iki işaretçi aynı düğümdedir. $d < \lambda$ olduğundan buluşma, kaplumbağanın döngüye girişinden en çok $\lambda - 1$ tur sonra gerçekleşir. Kaplumbağa döngüye μ. turda girer; toplam tur sayısı bu yüzden $\mu + \lambda$'dan küçüktür. Döngü yoksa tavşan en çok $n/2$ turda `NULL`'a varır. Her iki durumda da süre $O(n)$'dir. ■

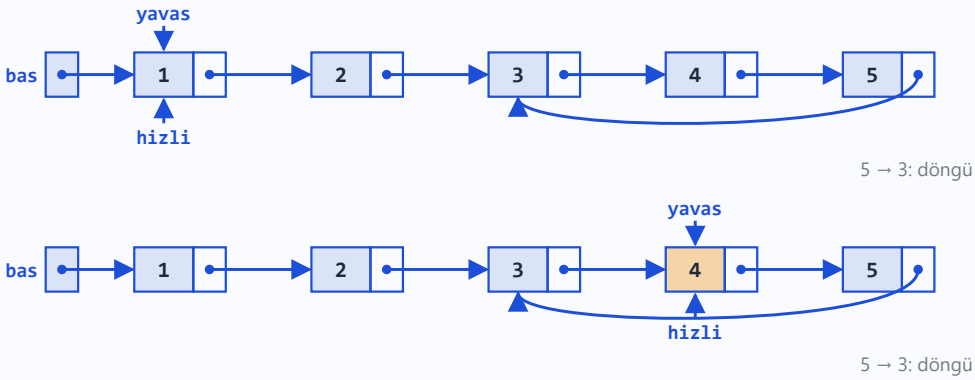
Kanıttaki önemli nokta, tavşanın kaplumbağanın "üzerinden atlayamamasıdır". Tavşan her turda iki adım atar, ama aradaki uzaklık her turda tam 1 azalır. Uzaklık 1 iken bir tur sonra 0 olur; 2 iken önce 1'e, sonra 0'a iner. Uzaklık hiçbir zaman negatife geçmez. Tavşan üç adım atsaydı uzaklık 2'şer azalırdı ve tek uzaklıkta buluşma kaçabilirdi.

Örnek 4.3

Liste 1 → 2 → 3 → 4 → 5'tir ve 5'in bağı yanlışlıkla 3'ü gösterir. Döngü 3 → 4 → 5 → 3'tür; $\mu = 2$, $\lambda = 3$. İki işaretçi de 1'den başlar. Tablo 4.7 her turun sonunu verir. Üçüncü turda iki işaretçi de 4'tedir ve fonksiyon `true` döndürür. Tur sayısı 3'tür; teoremdeki $\mu + \lambda = 5$ sınırının altındadır.

Tur	`yavas`	`hizli`	Döngü koşulu	`yavas == hizli`
Başlangıç	1	1	—	—
1	2	3	1 ve 1'in sonrası NULL değil	yanlış
2	3	5	3 ve 3'ün sonrası NULL değil	yanlış
3	4	4 (5 → 3 → 4)	5 ve 5'in sonrası NULL değil	doğru: `return true`

Tablo 4.7: 1 → 2 → 3 → 4 → 5 → (3) döngülü listede `dongu_var` izlemesi.



Şekil 4.16: Döngülü listede tavşan ve kaplumbağa. Üstte ikisi de baştadır; 5'ten 3'e dönen bağ döngüyü kurar. Altta üçüncü turun sonunda tavşan döngüde tur atıp kaplumbağaya yetişmiştir; ikisi de 4'tedir.

Aynı liste döngüsüz olsaydı, yani 5'in bağı `NULL` olsaydı, birinci turda tavşan 3'e, ikinci turda 5'e varırdı. Üçüncü denetimde `hizli->bag == NULL` olduğu için döngü biter ve fonksiyon `false` döndürürdü. Doğrulama programı iki durumu da çalıştırmış, tablodaki konumları birebir vermiştir.

4.6.6 Döngünün başı

Döngünün varlığını bilmek çoğu zaman yetmez; hatayı düzeltmek için döngünün hangi düğümde başladığı da bilinmelidir. Floyd'un yöntemi bunu da $O(1)$ ek bellekle verir. Buluşmadan sonra işaretçilerden biri listenin başına alınır. Ardından ikisi de her turda *birer* adım ilerletilir. Yeniden buluştukları düğüm döngünün ilk düğümüdür. Doğrulama programında bu yöntem şöyle yazılmıştır:

```
oge_gost dongu_basi(oge_gost bas) {
    oge_gost yavas = bas, hizli = bas;
    while (hizli != NULL && hizli->bag != NULL) {
        yavas = yavas->bag;
        hizli = hizli->bag->bag;
        if (yavas == hizli) {          // döngü var
            yavas = bas;               // biri başa döner
            while (yavas != hizli) {   // ikisi birer adım
                yavas = yavas->bag;
                hizli = hizli->bag;
            }
            return yavas;              // döngünün ilk düğümü
        }
    }
    return NULL;                       // döngü yok
}
```

Teorem 4.2 (Döngünün başı)

Buluşmadan sonra bir işaretçi başa alınıp iki işaretçi birer adım ilerletilirse, ikisi tam μ adım sonra döngünün ilk düğümünde buluşur.

Kanıt. Buluşma t. turda olsun. O anda kaplumbağa baştan t, tavşan $2t$ adım uzaktadır ve ikisi aynı düğümdedir. İki konum aynı düğümü gösterdiğine göre aradaki fark t, döngü uzunluğu λ'nın bir katıdır: tavşan fazladan attığı t adımda döngüyü tam sayıda kez dolaşmıştır. Şimdi baştan başlayan işaretçi μ adım sonra döngünün ilk düğümüne varır. Buluşma noktasından başlayan işaretçi ise μ adım sonra baştan $t + \mu$ adım uzaktadır. Döngünün ilk düğümü baştan μ adım uzaktadır ve t, λ'nın katı olduğu için $t + \mu$ adım da aynı düğüme düşer. Yani iki işaretçi μ. adımda döngünün başında buluşur. Daha önce buluşamazlar, çünkü baştan gelen işaretçi μ. adımdan önce döngünün dışındadır, öteki işaretçi ise hep döngünün içindedir. ■

Örnekte buluşma 3. turda, 4'te olmuştu. Baştan gelen işaretçi $1 \to 2 \to 3$, buluşma noktasından gelen işaretçi $4 \to 5 \to 3$ yolunu izler. İkisi de iki adımda ($\mu = 2$) 3'e varır ve 3 döngünün başı olarak bulunur. Döngünün uzunluğunu bulmak ise daha da kolaydır: buluşma düğümünden başlayan bir işaretçi, aynı düğüme dönene kadar sayılarak yürütülür. Bu, notun sonundaki alıştırmalardan biridir.

Floyd'un yöntemi bağlı listenin dışında da kullanılır. Her değeri bir öncekinden aynı kuralla hesaplanan her dizi, örneğin bir rastgele sayı üretecinin çıktısı, sonunda bir döngüye girer. Yöntem bu döngünün başını ve uzunluğunu dizinin elemanlarını saklamadan bulur. Knuth, yöntemi *The Art of Computer Programming*'in ikinci cildinde rastgele sayı üreteçleriyle ilgili bir alıştırmada (3.1-6) Floyd'a atfeder.

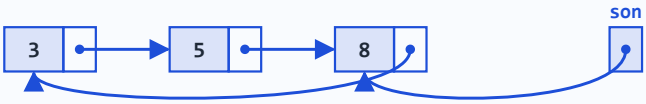
4.7 Dairesel liste ve başlık düğümü

Tek yönlü bağlı listenin iki tekrarlayan sıkıntısı vardır. Sona ekleme ve birleştirme son düğümü aramak zorundadır, ekleme ve silme kodları ise baş düğüm için ayrı bir dal içerir. Bu kesimdeki iki küçük değişiklik bu sıkıntıları giderir. Dairesel listede son düğüm ilk düğümü gösterir ve liste son düğümden tutulur. Başlık düğümünde ise listenin başına veri taşımayan, hiç silinmeyen bir düğüm konur.

4.7.1 Dairesel liste

Tanım 4.4

Dairesel bağlı liste (*circular linked list*), son düğümünün bağ alanı `NULL` yerine ilk düğümü gösteren bağlı listedir. Liste genellikle son düğümü gösteren bir `son` işaretçisiyle tutulur. Boş dairesel listede `son` `NULL`'dır; tek düğümlü listede düğüm kendini gösterir.



Şekil 4.17: 3 → 5 → 8 dairesel listesi. Son düğümün (8) bağı ilk düğüme (3) döner. Liste `son` işaretçisiyle tutulur; ilk düğüm `son->bag` 'dir.

Listenin başı yerine sonunu tutmak ilk bakışta tuhaf görünür. Ama son düğümü bilen biri ilk düğümü de bilir: ilk düğüm `son->bag` 'dir ve tek adımda bulunur. Başı bilen biri ise sonu ancak bütün listeyi yürüyerek bulabilir. `son` işaretçisi böylece iki ucu birden $O(1)$'de erişilebilir kılar. Bu yüzden dairesel listede hem başa hem sona ekleme sabit sürer.

Dairesel listede hiçbir bağ `NULL` değildir. Bu, gezinmenin bitiş koşulunu değiştirir. `p != NULL` koşuluyla yazılmış bir döngü dairesel listede hiç bitmez. Gezinme, başlangıç düğümüne geri dönülünce durur. Başlangıç düğümü döngünün daha ilk turunda koşulu sağlayacağı için koşul döngünün sonunda denetlenir; bu yüzden `do-while` kullanılır. Sunumdaki uzunluk fonksiyonu bu kalıbı gösterir:

```
int d_uzunluk(oge_gost son) {
    int n = 0;
    oge_gost p = son;
    if (son == NULL) return 0;   // boş liste
    do {
        n++;
        p = p->bag;
    } while (p != son);          // tur tamamlandı
    return n;
}
```

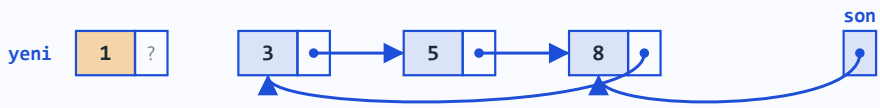
Boş liste ayrıca denetlenir, çünkü `do-while` gövdesi en az bir kez çalışır ve boş listede `p->bag` okunamaz. Liste boş değilse `p` son düğümden başlar. Gövde önce sayacı artırır, sonra `p` 'yi ilerletir. Koşul, `p` yeniden son düğüme dönene kadar döngünün sürmesini sağlar. 3 → 5 → 8 listesinde `p` sırasıyla 3'e, 5'e ve 8'e geçer; üçüncü turdan sonra `p == son` olur ve fonksiyon 3 döndürür. Süre $\Theta(n)$'dir. Aynı döngü `while (p != son)` biçiminde yazılsaydı, `p` başlangıçta zaten `son` 'a eşit olduğu için gövde hiç çalışmaz ve sonuç 0 olurdu.

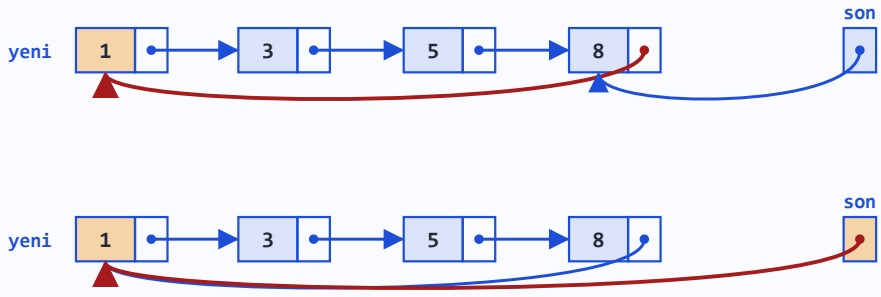
4.7.2 Dairesel listeye ekleme

Dairesel listede başa ekleme, yeni düğümü son düğüm ile ilk düğüm arasına koymaktır:

```
void d_basa_ekle(oge_gost *son, oge_gost yeni) {
    if (*son == NULL) {          // boş liste
        *son = yeni;
        yeni->bag = yeni;        // kendini gösterir
    } else {
        yeni->bag = (*son)->bag;  // eski ilk düğüm
        (*son)->bag = yeni;
    }
}
```

Liste boşsa yeni düğüm tek düğüm olur. `son` onu gösterir ve düğümün bağı kendine döner: tek düğümlü dairesel liste. Liste boş değilse iki bağ yazılır. `yeni->bag = (*son)->bag` satırı yeni düğümü eski ilk düğüme bağlar. `(*son)->bag = yeni` ise son düğümü yeni düğüme bağlar; yeni düğüm artık ilk düğümdür. Burada da sıra "önce koru, sonra bağla" kuralına uyar. `(*son)->bag` yazımındaki parantez zorunludur; `*son->bag` , `*(son->bag)` diye okunur ve derlenmez.





Şekil 4.18: 3 → 5 → 8 dairesel listesine 1 eklemek. Üstte yeni düğüm hazırdır. Ortada `yeni->bag = son->bag` ve `son->bag = yeni` ile 1 başa eklenmiştir: 1 → 3 → 5 → 8. Altta ayrıca `*son = yeni` yazılmış, böylece 1 sona eklenmiş olur: 3 → 5 → 8 → 1.

Sona ekleme için ayrı bir fonksiyon gerekmez. Dairesel listede "ilk" ve "son", yalnız `son` işaretçisinin nerede durduğuna bağlıdır. Yeni düğüm başa eklendikten sonra `son` bir düğüm ilerletilirse, yani `*son = yeni` yazılırsa, aynı düğüm sona eklenmiş olur: `d_basa_ekle(son, yeni); *son = yeni;`. Başa ve sona ekleme aynı bağ işlemidir; fark yalnız `son` işaretçisindedir. İkisi de *O(1)*'dir. Doğrulama programında 3 → 5 → 8 listesine başa ekleme 1 → 3 → 5 → 8, sona ekleme 3 → 5 → 8 → 1 sonucunu vermiştir.

Dairesel liste, üzerinde sırayla tekrar tekrar gezinilen veriler için doğal bir yapıdır. İşletim sistemindeki çevrimsel (*round-robin*) iş sıralayıcı, hazır süreçleri dairesel bir listede tutar; her zaman diliminde bir sonraki sürece geçer ve listenin sonuna gelince kendiliğinden başa döner. Çok oyunculu bir oyunda sıranın kimde olduğu da aynı biçimde tutulabilir.

4.7.3 Başlık düğümü

Önceki kesimlerdeki ekleme ve silme kodlarının hepsinde aynı ayrım vardı: değişiklik baş düğümdeyse `*bas` değişir, değilse bir önceki düğümün bağı değişir. Bu ayrım her fonksiyonda bir `if (onceki == NULL)` dalı, her çağrıda bir adresle aktarım gerektirir. Kökeni basittir: baş düğümün öncesi yoktur. Başlık düğümü bu eksikliği giderir.

> **Tanım 4.5**
>
> **Başlık düğümü** (*header node*), listenin başında duran, veri taşımayan ve hiç silinmeyen bir düğümdür. Liste değişkeni her zaman başlık düğümünü gösterir. Boş liste yalnız başlık düğümünden oluşur. Bu tür bir düğüme, sınır denetimlerini gereksiz kıldığı için bazı kaynaklarda **bekçi** (*sentinel*) düğüm de denir.



Şekil 4.19: Başlık düğümlü liste. Solda dolu liste: `liste` başlığı (H) gösterir, gerçek düğümler H'den sonra gelir (H → 3 → 8). Sağda boş liste: yalnız H vardır.

Başlık düğümünün kazancı şudur: her gerçek düğümün bir öncesi vardır. İlk gerçek düğümün öncesi başlıktır. Liste değişkeni başlığı gösterir ve başlık hiç silinmediği için bu değişken hiç değişmez. Fonksiyonlara listenin adresini vermek gerekmez; başlığın adresi yeter. Sıralı ekleme fonksiyonu bu yüzden çok kısalır:

```
// başlıklı listede sıralı ekleme: özel durum yok
void b_sirali_ekle(oge_gost baslik, oge_gost yeni) {
    oge_gost onceki = baslik;       // asla NULL değil
    while (onceki->bag != NULL &&
           onceki->bag->deger < yeni->deger)
        onceki = onceki->bag;
    yeni->bag = onceki->bag;
    onceki->bag = yeni;
}
```

`onceki` başlıktan başlar ve hiçbir zaman `NULL` olmaz. Döngü, `onceki` 'nin sonraki düğümü var olduğu ve yeni değerden küçük olduğu sürece ilerler. Burada tek bir işaretçi yeterlidir, çünkü incelenen düğüm her zaman `onceki->bag` 'dir. Döngü bittiğinde yeni düğüm `onceki` 'nin arkasına eklenir. Başa ekleme, araya ekleme, sona ekleme ve boş listeye ekleme, dördü de aynı iki satırla yapılır. `if (onceki == NULL)` özel durumu ortadan kalkmıştır. Doğrulama programında başlıklı boş listeye sırasıyla 8, 3, 5, 1 ve 9 eklenmiş, sonuç 1 → 3 → 5 → 8 → 9 olmuştur. Bu sırada 3 ve 1 başa, 9 sona eklenmiştir.

Başlık düğümünün bedeli tek bir düğümün belleğidir. Bazı uygulamalarda başlığın veri alanı da boşa gitmez; listenin uzunluğu gibi bir özet bilgi tutmak için kullanılır. Gezinme ve yazdırma kodlarında ise başlığın atlanması gerekir: gerçek düğümler `liste->bag` 'den başlar. Başlık düğümü en büyük yararını çift yönlü ve dairesel listelerle birlikte kullanıldığında gösterir.

4.8 Çift yönlü bağlı liste

Tek yönlü listede bir düğümden geriye gitmek mümkün değildir. Bir düğümü silmek ya da önüne ekleme yapmak için öncesi gerekir ve öncesi ancak baştan aranarak bulunur. Her düğüme önceki düğümün adresini tutan ikinci bir bağ eklenirse bu kısıt ortadan kalkar.

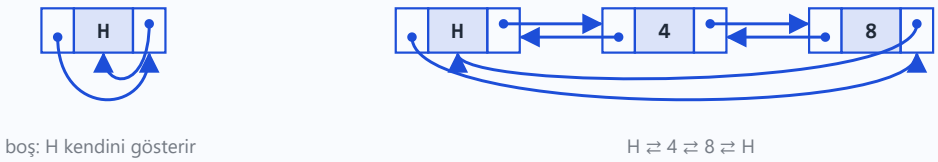
4.8.1 Yapı ve değişmez

Tanım 4.6

Çift yönlü bağlı liste (*doubly linked list*), her düğümün hem sonraki hem önceki düğümün adresini tuttuğu bağlı listedir. Sunumdaki gerçeklemede önceki düğümü gösteren alan `sol` (*llink*), sonrakini gösteren alan `sag` (*rlink*) adını taşır.

```
typedef struct doge *doge_gost;
struct doge {
    doge_gost sol;   // onceki (llink)
    int deger;
    doge_gost sag;   // sonraki (rlink)
};
// değişmez: p == p->sol->sag == p->sag->sol
```

Sunum çift yönlü listeyi başlık düğümlü ve dairesel olarak kurar. Başlık düğümünün `sag` bağı ilk gerçek düğümü, `sol` bağı son gerçek düğümü gösterir. Son gerçek düğümün `sag` bağı ve ilk gerçek düğümün `sol` bağı başlığa döner. Böylece listede hiç `NULL` bağ yoktur. Boş listede başlığın iki bağı da kendini gösterir.



Şekil 4.20: Başlıklı dairesel çift yönlü bağlı liste. Solda boş liste: H'nin iki bağı da kendini gösterir. Sağda H ⇄ 4 ⇄ 8 ⇄ H; her düğümün `sag` bağı sonrakini, `sol` bağı öncekini gösterir ve iki uç başlığa döner.

Bu yapıda her düğüm için geçerli olan bir eşitlik vardır. Bir düğümün solundaki düğümün sağı, düğümün kendisidir; sağındaki düğümün solu da yine kendisidir. Kısaca `p == p->sol->sag == p->sag->sol`. Bu eşitliğe yapının **değişmezi** denir: her işlemden önce ve sonra bütün düğümler için doğru olmalıdır. Başlıklı ve dairesel yapıda eşitlik başlık için de, uçtaki düğümler için de geçerlidir; hiçbir düğüm özel değildir. Bir ekleme ya da silme kodu yazılırken sorulacak soru şudur: işlem bittiğinde değişmez bütün düğümlerde yeniden sağlanıyor mu? Notun doğrulama programı her işlemden sonra bu eşitliği bütün düğümler için denetlemiştir.

Çift yönlü listenin üç önemli kazancı vardır. Liste iki yönde gezilebilir: `p = p->sag` ileri, `p = p->sol` geri gider. Adresi bilinen bir düğüm, öncesi aranmadan *O(1)*'de silinir. Bir düğümün önüne de arkasına da *O(1)*'de ekleme yapılır. Bedeli de üçtür. Her düğüm iki işaretçi taşır; 64 bitlik sistemde bu, düğüm başına fazladan 8 bayttır. Her ekleme ve silme dört bağı günceller. Bağ sayısı arttığı için bağ hatası yapmak da kolaylaşır.

4.8.2 Ekleme

Sunumdaki ekleme fonksiyonu, `p` düğümünün sağına yeni bir düğüm ekler:

```
// p düğümünün sağına yeni düğümü ekler
void cift_ekle(doge_gost p, doge_gost yeni) {
    yeni->sol = p;
    yeni->sag = p->sag;
    p->sag->sol = yeni;
    p->sag = yeni;
}
```

Dört satırın her biri bir bağ yazar. İlk iki satır yalnız yeni düğümü hazırlar: `yeni->sol = p` yeni düğümün solunu `p`'ye, `yeni->sag = p->sag` yeni düğümün sağını `p`'nin o anki sağına bağlar. Bu iki satırdan sonra listenin kendisi hiç değişmemiştir; baştan ya da sondan gezinen biri yeni düğümü görmez. Üçüncü satır `p->sag->sol = yeni`, `p`'nin sağındaki düğümün solunu yeni düğüme çevirir. Dördüncü satır `p->sag = yeni` ise `p`'nin sağını yeni düğüme çevirir.

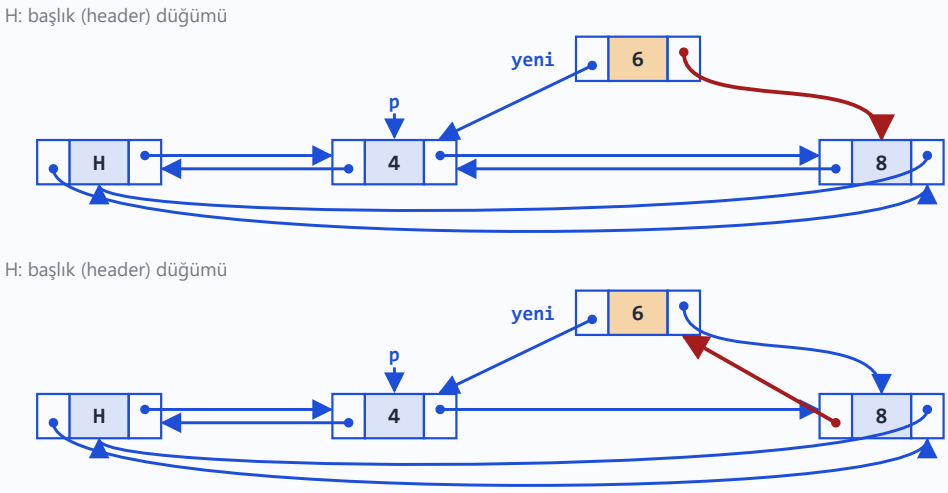
Üçüncü ve dördüncü satırın sırası değiştirilemez. Üçüncü satır, `p`'nin sağındaki düğüme `p->sag` üzerinden ulaşır. Dördüncü satır önce çalışsaydı `p->sag` artık yeni düğüm olurdu. `p->sag->sol = yeni` ise yeni düğümün solunu kendine çevirirdi. Asıl sağ komşunun solu ise hâlâ `p`'yi gösterirdi; değişmez bozulurdu. İlk iki satır ise herhangi bir sırada yazılabilir, çünkü yalnız yeni düğüme yazarlar.

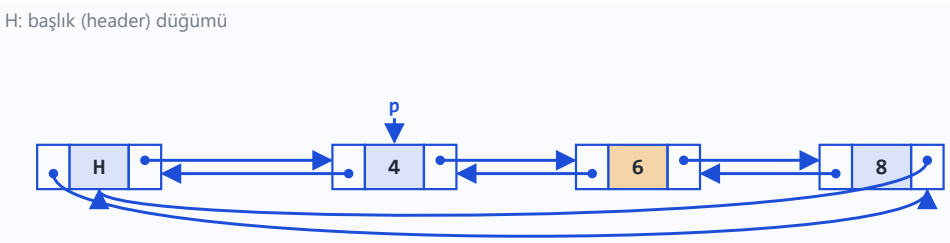
Örnek 4.4

Liste H ⇄ 4 ⇄ 8 ⇄ H'dir. 6 değerli yeni düğüm 4'ün sağına eklenecektir; `p` düğüm(4)'tür. Tablo 4.8 her satırdan sonra değişen bağı verir.

Satır	Değişen bağ	Açıklama
`yeni->sol = p`	6.sol → 4	Yeni düğümün sol bağı 4'ü gösterir.
`yeni->sag = p->sag`	6.sag → 8	Sağ bağı 8'i gösterir. Liste henüz değişmedi.
`p->sag->sol = yeni`	8.sol → 6	`p->sag` hâlâ 8 olduğu için 8'in soluna yazılır.
`p->sag = yeni`	4.sag → 6	Son bağ; liste H ⇄ 4 ⇄ 6 ⇄ 8 ⇄ H.

Tablo 4.8: `cift_ekle(p, yeni)` izlemesi; `p` = düğüm(4), `yeni` = düğüm(6), başlangıç H ⇄ 4 ⇄ 8 ⇄ H.





Şekil 4.21: Çift yönlü eklemenin üç anı. Üstte yeni düğüm hazırlanmıştır: solu 4'ü, sağı 8'i gösterir. Ortada `p->sag->sol = yeni` ile 8'in solu 6'ya çevrilmiştir. Altta `p->sag = yeni` ile ekleme tamamlanmıştır: H ⇄ 4 ⇄ 6 ⇄ 8 ⇄ H.

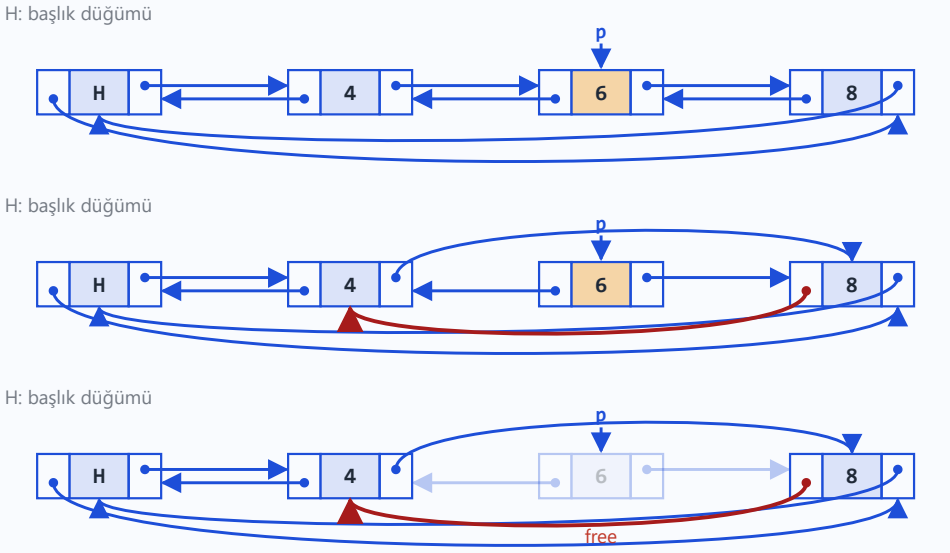
Fonksiyonda hiç `if` yoktur. `p` başlık olsa da (başa ekleme), son düğüm olsa da (sona ekleme), liste boş olsa da aynı dört satır doğru çalışır. Boş listede `p` başlıktır ve `p->sag` da başlıktır. Yeni düğümün iki bağı da başlığı gösterir, başlığın iki bağı da yeni düğümü gösterir. Bu genellik başlık düğümünün ve dairesel yapının ortak kazancıdır. Süre dört atama, yani *O(1)*'dir.

4.8.3 Silme

Çift yönlü listede silinecek düğümün öncesi zaten eldedir: `p->sol`. Bu yüzden silme, düğümün iki komşusunu birbirine bağlamak ve düğümü bırakmaktan ibarettir:

```c
void cift_sil(doge_gost baslik, doge_gost p) {
    if (p == baslik) {       // başlık silinmez
        fprintf(stderr, "Baslik silinemez\n");
        return;
    }
    p->sol->sag = p->sag;
    p->sag->sol = p->sol;
    free(p);
}
```

Başlık düğümü hiçbir zaman silinmemelidir; silinirse liste değişkeni sarkan bir işaretçiye dönüşür. Fonksiyon bu durumu denetler ve bir hata iletisiyle döner. Ardından iki bağ yazılır. `p->sol->sag = p->sag` satırı soldaki komşunun sağını, sağdaki komşuya çevirir. `p->sag->sol = p->sol` satırı sağdaki komşunun solunu, soldaki komşuya çevirir. Bu iki satırın sırası önemli değildir, çünkü ikisi de `p`'nin kendi bağlarını okur ve `p`'nin bağları değişmez. Son olarak düğüm bırakılır.



Şekil 4.22: H ⇄ 4 ⇄ 6 ⇄ 8 ⇄ H listesinden 6'yı silmek. Üstte `p` düğüm(6)'yı gösterir; öncesi `p->sol` ile elde. Ortada iki komşu birbirine bağlanmıştır: 4'ün sağı 8, 8'in solu 4. Altta `free(p)` ile düğüm iade edilmiştir.

Silme işleminde arama yoktur; süre *O(1)*'dir. Tek yönlü listede aynı işlem, öncesini bulmak için *O(n)* sürerdi. Başlık düğümü sayesinde ilk ve son düğüm için de ayrı kod gerekmez. İlk gerçek düğüm silinirken `p->sol` başlıktır ve başlığın sağı güncellenir. Son gerçek düğüm silinirken `p->sag` başlıktır ve başlığın solu güncellenir. Doğrulama programında H ⇄ 4 ⇄ 6 ⇄ 8 ⇄ H listesinden önce 6, sonra 4 ve 8 silinmiş; her adımda değişmez sağlanmış ve sonunda başlığın iki bağı da kendini göstermiştir. Başlığı silme denemesi ise “Baslik silinemez” iletisiyle reddedilmiştir.

> **Sık yapılan hata**
>
> Çift yönlü listede bağların yalnız bir yönünü güncellemek. Örneğin eklemede `p->sag->sol = yeni` satırını unutmak. İleri gezinme doğru çalışır ve hata fark edilmez. Ama geri gezinmede 8'den sola gidilince 6 atlanır ve doğrudan 4'e varılır. İki yönde farklı listeler gören bir yapıda bir sonraki silme bağları tamamen dağıtır. Her işlemden sonra değişmezi (`p == p->sol->sag == p->sag->sol`) bütün düğümlerde denetleyen küçük bir test fonksiyonu bu hatayı hemen yakalar.

4.9 Kullanılabilir düğüm listesi

Bazı programlar çalışma süresi boyunca sürekli düğüm ekler ve siler. Örneğin bir polinom işleme programı her toplama ve çarpmada yeni terim düğümleri oluşturur, ara sonuçların düğümlerini de hemen bırakır. Her eklemede `malloc`, her silmede `free` çağırmak doğrudur ama ucuz değildir. Bu iki fonksiyon, bellekte uygun boyutta boş blok aramak ve kendi kayıtlarını güncellemek zorundadır. Horowitz, Sahni ve Anderson-Freed'in kitabındaki çözüm, silinen düğümleri bırakmak yerine bir kenarda biriktirip yeni düğüm gerektiğinde önce oradan almaktır.

4.9.1 `get_node` ve `ret_node`

> **Tanım 4.7**
>
> **Kullanılabilir düğüm listesi** (*available space list*), silinen düğümlerin `free` edilmek yerine eklendiği ve yeni düğüm gerektiğinde önce başvurulan bağlı listedir. Listenin başı geleneksel olarak `avail` adlı genel bir değişkende tutulur.

```
oge_gost avail = NULL;        // kullanılabilir düğümler

oge_gost get_node(void) {
    oge_gost p;
    if (avail != NULL) {         // önce listeden al
        p = avail;
        avail = avail->bag;
    } else {                     // yoksa sistemden iste
        p = malloc(sizeof *p);
        if (p == NULL) exit(1);
    }
    return p;
}

void ret_node(oge_gost p) {      // free yerine
    p->bag = avail;
    avail = p;
}
```

`get_node`, `malloc`'un yerine geçer. `avail` boş değilse listenin ilk düğümü alınır ve `avail` bir düğüm ilerletilir. Bu durumda `malloc` hiç çağrılmaz. `avail` boşsa düğüm sistemden istenir; bellek yoksa program biter. `ret_node` ise `free`'nin yerine geçer: verilen düğüm `avail`'in başına eklenir. İki fonksiyon da sabit sayıda atama yapar ve *O(1)* sürer.

Dikkatli bakıldığında `avail`'in bir yığın olduğu görülür. `ret_node` listenin başına ekler, `get_node` listenin başından alır. Son iade edilen düğüm ilk yeniden kullanılır: son giren ilk çıkar. Yani `get_node` bağlı yığının çıkarma işlemi (*pop*), `ret_node` ise üste ekleme işlemi (*push*) ile aynıdır. Başa ekleme ve baştan silme bağlı listede sabit süren iki işlem olduğu için bu seçim tesadüf değildir.

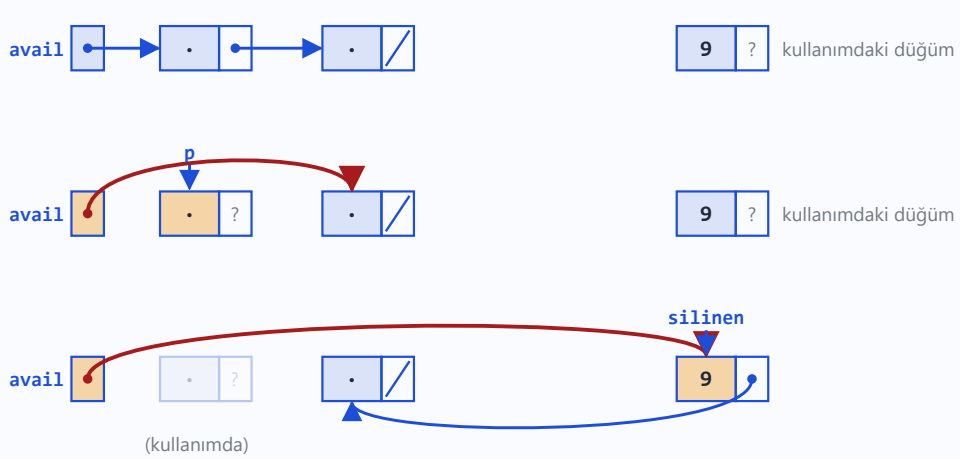
Yöntemin bir sınırlaması vardır: `avail`'deki düğümler program bitene kadar heap belleğe geri dönmez. Program bir an çok sayıda düğüm kullanıp sonra çoğunu silerse, silinen düğümler `avail`'de bekler ve başka bir veri yapısı bu belleği kullanamaz. Ayrıca `avail` yalnız tek bir düğüm tipi için işe yarar. Farklı boyutta düğümler kullanan bir program her tip için ayrı bir `avail` tutmak zorundadır. Yöntem

bu yüzden düğüm tipinin tek olduğu ve ekleme-silmenin çok sık yapıldığı durumlar için uygundur.

4.9.2 Adım adım

Şekil 4.23 `avail` zincirinde iki boş düğümün beklediği bir durumdan başlar. Sağda kullanımda olan bir düğüm (9) vardır.

1. `get_node()` çağrılır. `avail` boş olmadığı için `p = avail` yapılır; `malloc` çağrılmaz.
2. `avail = avail->bag` ile zincirin başı ilerler. Alınan düğüm artık çağırana aittir; `avail` 'de bir düğüm kalmıştır.
3. Daha sonra 9'un düğümü silinecektir. `free` yerine `ret_node(silinen)` çağrılır ve önce `silinen->bag = avail` yazılır.
4. `avail = silinen` ile düğüm zincirin başına geçer ve yeniden kullanıma hazırdır.



Şekil 4.23: `avail` listesiyle düğüm alma ve iade. Üstte `avail` zincirinde iki boş düğüm vardır. Ortada `get_node` ilk düğümü `p` 'ye vermiş, `avail` ilerlemiştir. Altta silinen düğüm `ret_node` ile zincirin başına eklenmiştir.

Doğrulama programında bu davranış sayaçlarla gözlenmiştir. `avail` 'de bekleyen düğümler varken yapılan `get_node` çağrıları `malloc` sayacını artırmamış ve son iade edilen düğümü geri vermiştir. Düğümler zincirde beklerken veri alanlarında eski değerleri taşıyabilir. `get_node` , `malloc` gibi, alanları ilklendirmez; bu iş çağıranındır.

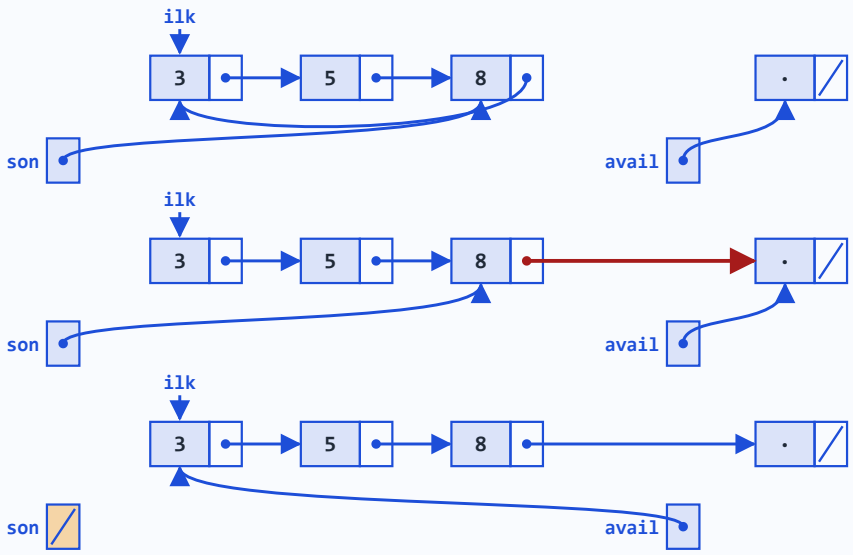
> **Sık yapılan hata**
>
> Bir düğümü `ret_node` ile iade ettikten sonra kullanmaya devam etmek. `free` 'den farklı olarak bu hata hiçbir araçla yakalanmaz, çünkü düğümün belleği gerçekten geçerlidir. Ama düğümün bağ alanı artık `avail` zincirinin bir parçasıdır. Ona yazılan her değer `avail` 'i bozar ve bir sonraki `get_node` çağrısı rastgele bir adresi düğüm diye verebilir.

4.9.3 Dairesel listeyi tek hamlede iade

Bir listenin bütün düğümlerini `avail` 'e iade etmek, tek yönlü düz listede her düğüm için bir `ret_node` ya da en azından son düğümü bulmak için bir yürüyüş, yani $\Theta(n)$ ister. Dairesel listede ise bu iş uzunluktan bağımsız olarak $O(1)$'dir. Horowitz'in kitabında `cerase` adıyla geçen yordam, halkayı tek bir noktadan açıp bütün listeyi `avail` 'in önüne ekler:

```
// dairesel listeyi O(1)'de avail'e iade et
void d_iade(oge_gost *son) {
    oge_gost ilk;
    if (*son != NULL) {
        ilk = (*son)->bag;     // ilk düğüm
        (*son)->bag = avail;   // halkayı aç
        avail = ilk;
        *son = NULL;
    }
}
```

Liste boş değilse önce ilk düğümün adresi `ilk` 'e alınır (`(*son)->bag`). Sonra son düğümün bağı `avail` 'e çevrilir. Bu atama halkayı açar: son düğüm artık ilk düğümü değil, `avail` zincirinin başını gösterir. Bu anda `ilk` 'ten başlayıp ilerleyen biri bütün liste düğümlerinden geçer ve `avail` 'in eski düğümlerine ulaşır. `avail = ilk` ataması bu zincirin tamamını `avail` yapar. En sonda çağıranın `son` değişkeni `NULL` 'a çekilir; liste artık boştur ve sarkan işaretçi kalmaz.



Şekil 4.24: 3 → 5 → 8 dairesel listesini `avail` 'e iade etmek. Üstte `son` = 8, `ilk` = 3 ve `avail` 'de bir düğüm vardır. Ortada `son->bag = avail` ile halka açılmış, 8 artık `avail` 'in eski başını gösterir. Altta `avail = ilk` ve `*son = NULL` ile bütün liste `avail` 'in önüne geçmiştir.

Yordam yalnız dört atama yapar ve hiç döngü içermez; liste üç düğümlü de olsa üç milyon düğümlü de olsa süre aynıdır. Bunu mümkün kılan, dairesel listenin iki ucunu da *O(1)*'de vermesidir: `son` son düğümü, `son->bag` ilk düğümü gösterir. Tek yönlü düz listede aynı iş için son düğümü bulmak *Θ(n)* sürerdi. Doğrulama programında sekiz düğümlü bir `avail` zincirinden üç düğüm alınmış ve bunlarla 3 → 5 → 8 dairesel listesi kurulmuştur; `avail` 'de beş düğüm kalmıştır. `d_iade` çağrısından sonra `avail` 3, 5, 8 diye başlamış ve uzunluğu yeniden sekize çıkmıştır.

4.10 Karmaşıklık özeti ve ölçüm

Tablo 4.9 bölümdeki işlemlerin maliyetini dört yapı için yan yana verir. Tablodaki her değer, ilgili kesimde gerekçesiyle açıklanmıştır. Tek yönlü liste yalnız baş işaretçisiyle, dairesel liste `son` işaretçisiyle, çift yönlü liste ise başlık düğümüyle tutulur.

İşlem	Dizi	Tek yönlü (baş)	Dairesel (son)	Çift yönlü (başlıklı)
***i*. öğeye erişim**	*O(1)*	*O(i)*	*O(i)*	*O(i)*
Başa ekleme	*O(n)*	*O(1)*	*O(1)*	*O(1)*
Sona ekleme	*O(1)**	*O(n)*	*O(1)*	*O(1)*
Bilinen düğümden sonra ekleme	*O(n)*	*O(1)*	*O(1)*	*O(1)*
Bilinen düğümü silme	*O(n)*	*O(n)*†	*O(n)*†	*O(1)*
Değer arama	*O(n)*	*O(n)*	*O(n)*	*O(n)*
Birleştirme	*O(n + m)*	*O(n)*	*O(1)*	*O(1)*
Düğüm başına ek bellek	—	1 işaretçi	1 işaretçi	2 işaretçi

Tablo 4.9: Dizi ve bağlı liste türlerinde işlem maliyetleri (*n* ve *m* liste uzunlukları).

* Dizinin kapasitesi doluysa önce yeniden boyutlandırma gerekir; bu adım *O(n)*'dir. † Silinecek düğümün öncesini bulmak için gezinme gerekir; öncesi biliniyorsa silme *O(1)*'dir. Ters çevirme, orta düğüm ve Floyd döngü tespiti ise her liste türünde *O(n)* zaman ve *O(1)* ek bellek ister.

Tablodaki "tek yönlü listede sona ekleme *O(n)*" satırının pratikteki anlamını görmek için küçük bir ölçüm yapılmıştır. Ölçüm programı *n* öğeyi üç yolla eklemiştir: diziye her seferinde başa (öncekileri `memmove` ile kaydırarak), listeye `basa_ekle` ile ve listeye `sona_ekle` ile. Program `-O2` ile derlenmiş ve birkaç kez çalıştırılmıştır. Tablo 4.10 çalıştırmalardan birini verir. Süreler makineye ve çalıştırmaya göre değişir; önemli olan sayıların büyüme biçimidir.

n	Dizi, başa (kaydırarak)	Liste, `basa_ekle`	Liste, `sona_ekle`
10.000	2,4	0,42	232
20.000	18,8	0,72	768
40.000	57,5	1,33	3.039

Tablo 4.10: *n* öğeyi ekleme süresi (milisaniye). Süreler makineye göre değişir.

Listeye başa ekleme her öğe için sabit iş yapar; toplam süre *Θ(n)*'dir ve *n* iki katına çıkınca yaklaşık iki katına çıkar. Öteki iki sütun karesel büyür: *n* iki katına çıkınca süre kabaca dört katına çıkar. Dizide her ekleme o ana kadarki bütün öğeleri kaydırır; listede her `sona_ekle` o ana kadarki bütün düğümleri yürür. İkisi de toplamda *n(n − 1)/2* birim iş yapar.

Ölçümün şaşırtıcı yanı, iki karesel işlemin arasındaki farktır. Aynı sayıda adım atmalarına rağmen listede sona ekleme, dizide başa eklemeden elli kat kadar yavaştır. Nedeni bellek erişim biçimidir. `memmove` art arda duran baytları büyük bloklar hâlinde taşır ve işlemcinin önbelleğini çok verimli kullanır. Liste yürüyüşü ise her adımda bellekte başka bir yere atlar; bir sonraki düğümün adresi ancak o anki düğüm okunduktan sonra bilinir. Bu yüzden asimptotik maliyet aynı olsa bile dizi işlemleri pratikte çoğu zaman daha hızlıdır. Bağlı liste, asimptotik farkın gerçekten büyük olduğu durumlarda, örneğin konumu bilinen bir yerde sık ekleme ve silmede tercih edilmelidir.

4.11 Sık yapılan hatalar

Bağlı liste kodundaki hataların çoğu birkaç kalıba indirgenir. Aşağıdaki liste bunları bir arada verir; her birinin ayrıntısı ilgili kesimde anlatılmıştır.

1. **Bağı korumadan üzerine yazmak.** Araya eklemede `onceki->bag = yeni` önce yazılırsa `onceki`'nin eski sonrakinin adresi kaybolur ve listenin kalanı kopar. Kural: önce koru, sonra bağla.
2. **Baş değişen yordama listeyi değerle vermek.** `basa_ekle(oge_gost bas, …)` gibi bir tanımda yalnız yerel kopya değişir; `main`'deki `liste` güncellenmez ve yeni düğüm sızar. Baş değişecekse liste adresle aktarılır: `oge_gost *bas` ve çağrıda `&liste`.
3. **`malloc` sonucunu denetlememek; `free`'yi unutmak ya da iki kez çağırmak.** Denetimsiz `malloc` bellek bittiğinde `NULL`'a yazar. Unutulan `free` sızıntı, ikinci `free` tanımsız davranış doğurur.
4. **Bırakılmış düğümün bağ alanını okumak.** `free(p); p = p->bag;` iade edilmiş bellekten okur. Önce sonraki saklanır, sonra bırakılır.
5. **Sınır durumlarını sınamamak.** Boş liste, tek düğümlü liste, baş düğüm ve son düğüm üzerinde yapılan işlemler ayrı ayrı denenmelidir. Hataların çoğu bu dört durumdan birinde ortaya çıkar.
6. **Dairesel listede `p != NULL` ile gezinmek.** Dairesel listede hiçbir bağ `NULL` değildir; döngü hiç bitmez. Bitiş koşulu başlangıç düğümüne geri dönmektir ve `do-while` ile yazılır.
7. **Koşul sırasını ters yazmak.** `p->deger != x && p != NULL` liste sonunda `NULL`'ın içine bakar. Kısa devre değerlendirmeden yararlanmak için `NULL` denetimi her zaman önce gelir.

Bu hataların çoğu derleyicinin gözünden kaçar ve program çoğu girdide doğru sonuç verir. Bu yüzden liste kodu, sınır durumlarını kapsayan küçük testlerle ve mümkünse bellek hatalarını yakalayan araçlarla sınanmalıdır. Linux ve macOS'ta derleyicinin `-fsanitize=address` seçeneği, `free` sonrası erişimi ve sızıntıyı çalışma sırasında raporlar. Ödev ve sınavlarda istenen de çalışan bir kodun ötesinde, her işaretçinin her an neyi gösterdiğini açıklayabilmektir.

4.12 Alıştırmalar ve çözümleri

Sunumun son bölümündeki beş alıştırma, bölümde öğrenilen kalıpları yeni problemlere uygular. Aşağıda her alıştırmanın çözüm fikri ve tam C kodu verilmiştir. Kodların hepsi derlenip örnek listelerle çalıştırılmıştır. Düğüm tipi ve `avail` fonksiyonları bölümdekilerle aynıdır.

4.12.1 İki sıralı listeyi düğüm ayırmadan birleştirmek

Sıralı iki liste tek bir sıralı listede birleştirilecektir; yeni düğüm ayrılmayacak, var olan düğümlerin bağları yeniden düzenlenecektir. Fikir şudur: iki listenin başındaki düğümler karşılaştırılır, küçük olan sonuç listesinin sonuna bağlanır ve o listede bir adım ilerlenir. Listelerden biri bittiğinde öteki liste olduğu gibi sona eklenir; kalan düğümler zaten sıralıdır ve hepsi sonuçtakilerden büyük ya da onlara eşittir.

```
oge_gost sirali_birlestir(oge_gost a, oge_gost b) {
    struct oge bas;             // geçici başlık düğümü
    oge_gost son = &bas;
    while (a != NULL && b != NULL) {
        if (a->deger <= b->deger) { son->bag = a; a = a->bag; }
        else                      { son->bag = b; b = b->bag; }
        son = son->bag;
    }
    son->bag = (a != NULL) ? a : b;  // kalan liste olduğu gibi eklenir
    return bas.bag;
}
```

Kodun incelikli yanı `bas` adlı yerel bir `struct oge` değişkenidir. Bu değişken geçici bir başlık düğümü görevi görür. Sonuç listesinin ilk düğümü onun bağına yazılır; böylece “sonuç listesi henüz boş mu?” özel durumu ortadan kalkar. `son` işaretçisi sonuç listesinin son düğümünü tutar ve her bağlamadan sonra bir ilerler. Fonksiyon sonunda geçici başlığın bağını, yani gerçek sonuç listesinin başını döndürür. Geçici başlık yığında durduğu için `malloc` ve `free` gerektirmez. Karşılaştırmada `<=` kullanıldığı için eşit değerlerde birinci listenin düğümü önce gelir. $1 \to 4 \to 7$ ile $2 \to 3 \to 8 \to 9$ listeleri $1 \to 2 \to 3 \to 4 \to 7 \to 8 \to 9$ olur. Her turda bir düğüm sonuca eklendiği için süre $O(n + m)$, ek bellek $O(1)$'dir.

4.12.2 Sondan *k*. düğüm

Listenin sondan k. düğümü tek geçişte bulunacaktır ($k = 1$ son düğümdür). İki işaretçi kullanılır. Hızlı işaretçi önce k adım öne alınır. Sonra iki işaretçi birlikte, birer adım yürütülür. Hızlı işaretçi `NULL`'a vardığında yavaş işaretçi ondan tam k düğüm geridedir, yani sondan k. düğümdedir.

```
oge_gost sondan_k(oge_gost bas, int k) {
    oge_gost hizli = bas, yavas = bas;
    for (int i = 0; i < k; i++) {
        if (hizli == NULL) return NULL;   // liste k'dan kısa
        hizli = hizli->bag;
    }
    while (hizli != NULL) {
        hizli = hizli->bag;
        yavas = yavas->bag;
    }
    return yavas;
}
```

Liste k'dan kısaysa hızlı işaretçi öne alınırken `NULL`'a düşer ve fonksiyon `NULL` döndürür. $1 \to 2 \to 3 \to 4 \to 7 \to 8 \to 9$ listesinde $k = 1$ için 9, $k = 3$ için 7, $k = 7$ için 1 bulunmuş; $k = 8$ için `NULL` dönmüştür. Hızlı işaretçi toplam n, yavaş işaretçi $n - k$ adım atar; süre $\Theta(n)$, ek bellek $O(1)$'dir.

4.12.3 Sıralı listede tekrarları silmek

Sıralı listede eşit değerler yan yana durur. Bu yüzden her düğüm yalnız sonrakiyle karşılaştırılır. Eşitse sonraki düğüm silinir ve `p` yerinde kalır, çünkü yeni sonraki de aynı değeri taşıyabilir. Eşit değilse `p` ilerler. Silinen düğümler `free` yerine `ret_node` ile `avail`'e iade edilir.

```
void sonrakini_iade(oge_gost onceki) {
    oge_gost silinecek = onceki->bag;
    if (silinecek == NULL) return;
    onceki->bag = silinecek->bag;
    ret_node(silinecek);
}
void tekrarlari_sil(oge_gost bas) {
    oge_gost p = bas;
    while (p != NULL && p->bag != NULL) {
        if (p->deger == p->bag->deger) sonrakini_iade(p);  // p yerinde kalır
        else p = p->bag;
    }
}
```

`sonrakini_iade`, bölümdeki `sonrakini_sil` fonksiyonunun `free` yerine `ret_node` çağıran biçimidir. 1 → 1 → 2 → 3 → 3 → 3 → 5 listesi 1 → 2 → 3 → 5 olur ve silinen üç düğüm `avail`'e geçer. Her denetim ya bir düğüm siler ya da `p`'yi ilerletir; toplam denetim sayısı $n - 1$'dir ve süre $O(n)$'dir.

4.12.4 Özyinelemeli ters çevirme

`liste_devrik_al`'ın özyinelemeli bir sürümü şu düşünceye dayanır: ilk düğümden sonraki liste özyinelemeli olarak ters çevrilir, ardından ilk düğüm bu ters listenin sonuna takılır. Ters listenin sonu, eski ikinci düğümdür ve ilk düğümün bağı hâlâ onu gösterir.

```
oge_gost ters_oz(oge_gost p) {
    oge_gost yeni_bas;
    if (p == NULL || p->bag == NULL) return p;  // 0 ya da 1 düğüm
    yeni_bas = ters_oz(p->bag);                 // kalanı çevir
    p->bag->bag = p;                            // eski sonraki bana dönsün
    p->bag = NULL;
    return yeni_bas;
}
```

Taban durumu boş ya da tek düğümlü listedir; böyle bir liste tersine çevrilmiş hâline eşittir. Aksi hâlde kalan liste çevrilir ve yeni baş `yeni_bas`'ta saklanır. `p->bag->bag = p` satırı, eski ikinci düğümün bağını `p`'ye çevirir; `p->bag = NULL` ise `p`'yi yeni son düğüm yapar. 10 → 20 → 30 → 40 listesi 40 → 30 → 20 → 10 olur. Süre yine $\Theta(n)$'dir, ama her düğüm için bir çağrı çerçevesi açıldığından ek bellek $\Theta(n)$'dir. Yinelemeli sürümün $O(1)$ ek belleği bu yüzden daha iyidir. Özyinelemeli çağrı fonksiyonun son işi olmadığı için derleyici onu döngüye de çeviremez.

4.12.5 Döngünün uzunluğu

Floyd'un yöntemiyle döngü bulunduktan sonra uzunluğu hesaplanacaktır. Buluşma düğümü döngünün içindedir. Oradan başlayan bir işaretçi, aynı düğüme geri dönene kadar sayılarak yürütülür.

```
int dongu_uzunlugu(oge_gost bas) {
    oge_gost yavas = bas, hizli = bas;
    while (hizli != NULL && hizli->bag != NULL) {
        yavas = yavas->bag;
        hizli = hizli->bag->bag;
        if (yavas == hizli) {               // buluşma döngünün içinde
            int n = 1;
            for (oge_gost p = yavas->bag; p != yavas; p = p->bag) n++;
            return n;
        }
    }
    return 0;                               // döngü yok
}
```

1 → 2 → 3 → 4 → 5 listesinde 5'in bağı 3'e yönlendirildiğinde fonksiyon 3, 1'e yönlendirildiğinde 5, 5'in kendisine yönlendirildiğinde 1 döndürmüştür; döngüsüz listede 0 döner. Buluşmaya kadar en çok n tur, sayma için λ adım atılır; süre $O(n)$, ek bellek $O(1)$'dir.

4.13 Özet

- **Bağlı liste**, öğeleri bellekte dağınık duran düğümlerde tutar; sıra her düğümün **bağ** alanıyla kurulur. Liste ilk düğümü gösteren bir işaretçiyle temsil edilir; **boş liste** `NULL`'dır.
- Düğüm, **kendine başvuran bir yapıdır**: `typedef struct oge *oge_gost;` ile işaretçi tipi önce adlandırılır, yapı kendi tipinden bir bağ alanı taşır.
- Düğümler `malloc(sizeof *p)` ile heap bellekten ayrılır; sonuç `NULL`'a karşı denetlenir. Her `malloc` için tam bir `free` çağrılır. Unutulan `free` **bellek sızıntısı**, erken ya da çift `free` **sarkan işaretçi** ve tanımsız davranış doğurur.
- **Bağ disiplini**: önce korunur, sonra bağlanır, en son bırakılır. Listenin başı değişecekse liste adresle (`oge_gost *bas`, çağrıda `&liste`) aktarılır.
- Tek yönlü listede başa ekleme *O(1)*, sona ekleme *O(n)*'dir; bilinen düğümden sonra ekleme ve sonrakini silme *O(1)*, değere göre silme ve arama *O(n)*, *i*. öğeye erişim *O(i)*'dir. İkili arama bağlı listede işe yaramaz.
- Liste özyinelemeli ya da döngüyle bırakılabilir; özyinelemeli sürüm *Θ(n)* yığın kullanır ve çok uzun listede yığın taşmasına yol açabilir.
- **Ters çevirme** (üç işaretçi), **birleştirme**, **orta düğüm** (yavaş ve hızlı işaretçi) ve **Floyd döngü tespiti** tek geçişte, *O(1)* ek bellekle çalışır. Floyd'un yönteminde buluşmadan sonra bir işaretçi başa alınıp ikisi birer adım yürütülürse döngünün başı bulunur.
- **Dairesel liste** son düğümden tutulur; başa ve sona ekleme *O(1)*'dir, gezinme `do-while` ile yapılır. **Başlık düğümü** baş düğüm özel durumunu kaldırır. **Çift yönlü bağlı liste** iki yönde gezinmeyi ve bilinen düğümü *O(1)*'de silmeyi sağlar; değişmezi `p == p->sol->sag == p->sag->sol`'dur.
- **Kullanılabilir düğüm listesi** (`avail`) silinen düğümleri yeniden kullanır; `get_node` ve `ret_node` bir yığının çıkarma ve ekleme işlemleridir. Dairesel liste `avail`'e *O(1)*'de iade edilir.

4.14 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

bağlı liste *linked list* · tek yönlü bağlı liste *singly linked list* · düğüm *node* · bağ *link* · baş *head* · işaretçi *pointer* · işaretçiye işaretçi *pointer to pointer* · boş işaretçi *null pointer (NULL)* · kendine başvuran yapı *self-referential struct* · eksik tip *incomplete type* · typedef *typedef* · dinamik bellek ayırma *dynamic memory allocation* · heap bellek *heap memory* · malloc *malloc* · free *free* · bellek sızıntısı *memory leak* · sarkan işaretçi *dangling pointer* · kısa devre değerlendirme *short-circuit evaluation* · yığın taşması *stack overflow* · kuyruk özyinelemesi *tail recursion* · listeyi ters çevirme *reversing a list* · döngü *cycle* · döngü tespiti *cycle detection* · dairesel bağlı liste *circular linked list* · başlık düğümü *header node* · bekçi *sentinel* · çift yönlü bağlı liste *doubly linked list* · kullanılabilir düğüm listesi *available space list* · push (üste ekleme) *push*

4.15 Kendini Dene

1. Boş bir listeye sırasıyla `sona_ekle(&liste, 3)`, `sona_ekle(&liste, 6)`, `sona_ekle(&liste, 9)`, `basa_ekle(&liste, 1)`, `sona_ekle(&liste, 12)` ve `sil(&liste, 6)` uygulanıyor. `yazdir(liste)` ne yazar?

   Cevap

   Üç sona eklemeden sonra liste 3 → 6 → 9'dur. Başa ekleme 1 → 3 → 6 → 9, sona ekleme 1 → 3 → 6 → 9 → 12 verir. `sil` 6'yı bulur (öncesi 3) ve 3'ün bağını 9'a çevirir. Çıktı `1 3 9 12` olur.

2. `yazdir(oge_gost p)` fonksiyonu `p`'yi `NULL`'a kadar ilerlettiği hâlde `main`'deki `liste` neden değişmez? `basa_ekle` neden `&liste` ile çağrılır?

   Cevap

   C argümanları değerle aktarır. `yazdir`'a `liste`'deki adresin bir kopyası gider ve fonksiyon yalnız bu kopyayı ilerletir. `basa_ekle` ise listenin başını değiştirmek zorundadır. Değerle çağrılsaydı yalnız yerel kopya yeni düğümü gösterir, `main`'deki `liste` eski kalır ve yeni düğüm sızardı. `&liste` ile fonksiyona değişkenin adresi verilir ve `*bas = yeni` doğrudan `liste`'yi değiştirir.

3. Sıralı 2 → 5 → 9 listesine `sirali_ekle` ile ikinci bir 5 ekleniyor. Döngü bittiğinde `onceki` nedir ve yeni 5 eskisinin önüne mi arkasına mı girer?

> **Cevap**
>
> İlk denetim 2 < 5 doğrudur; `onceki` 2'ye, `p` eski 5'e geçer. İkinci denetim 5 < 5 yanlıştır ve döngü biter. `onceki` düğüm(2)'dir. Yeni 5 eski 5'in önüne girer: 2 → 5 (yeni) → 5 (eski) → 9. Koşul `<=` olsaydı arkasına girerdi.

4. `orta` fonksiyonu 1 → 2 → 3 → 4 → 5 → 6 listesinde hangi düğümü döndürür? Kaç tur döner?

> **Cevap**
>
> Üç tur döner. Turlarda `yavas` 2, 3, 4'e; `hizli` 3, 5 ve `NULL`'a geçer. `hizli == NULL` olunca döngü biter ve fonksiyon 4'ü, yani ikinci ortayı döndürür.

5. 1 → 2 → 3 → 4 → 5 → 6 listesinde 6'nın bağı 4'ü gösteriyor. `dongu_var` kaçıncı turda, hangi düğümde buluşma bulur? Döngünün başını bulma adımı kaç adım sürer?

> **Cevap**
>
> Turların sonunda (`yavas` , `hizli`) sırasıyla (2, 3), (3, 5), (4, 4)'tür; buluşma 3. turda 4'tedir. Burada $\mu = 3$ ve $\lambda = 3$'tür. Bir işaretçi başa alınıp ikisi birer adım yürütülünce 3 adım sonra (1 → 2 → 3 → 4 ve 4 → 5 → 6 → 4) ikisi de 4'te buluşur. Döngünün başı 4'tür.

6. `liste_devrik_al` içindeki döngünün gövdesinde `p = p->bag` ile `q->bag = r` satırlarının yeri değiştirilirse 10 → 20 → 30 listesi ne olur?

> **Cevap**
>
> Birinci turda `r = NULL` , `q = p` = düğüm(10) olur. `q->bag = r` 10'un bağını `NULL` yapar. `q` ve `p` aynı düğümü gösterdiği için ardından gelen `p = p->bag` , `p`'yi `NULL` yapar. Döngü biter ve `*list = q` listeyi yalnız 10'dan ibaret bırakır. 20 ve 30 hiçbir işaretçiyle gösterilmez; sızmışlardır.

7. `cift_ekle` fonksiyonunun son iki satırı yer değiştirirse, yani önce `p->sag = yeni` , sonra `p->sag->sol = yeni` yazılırsa H ⇄ 4 ⇄ 8 ⇄ H listesinde 4'ün sağına 6 eklemek ne sonuç verir?

> **Cevap**
>
> `p->sag = yeni` 4'ün sağını 6 yapar. Ardından `p->sag` artık 6 olduğu için `p->sag->sol = yeni` 6'nın solunu 6'ya çevirir. 8'in solu hâlâ 4'tür. İleri gezinme H, 4, 6, 8 sırasını doğru verir; geri gezinme 8'den 4'e atlar ve 6'yı görmez. 6 için `p->sol->sag == p` eşitliği bozulmuştur (6'nın solu kendisidir ve sağı 8'dir).

8. Bir program `avail` boşken üç kez `get_node` , sonra iki kez `ret_node` , sonra yine üç kez `get_node` çağırıyor. `malloc` toplam kaç kez çağrılır?

> **Cevap**
>
> İlk üç çağrıda `avail` boştur; üç `malloc` yapılır. İki `ret_node` iki düğümü `avail`'e koyar. Sonraki üç `get_node` çağrısının ilk ikisi bu düğümleri alır, üçüncüsünde `avail` yine boştur ve bir `malloc` daha yapılır. Toplam dört `malloc` .

9. Tek yönlü bağlı listede yalnız baş işaretçisi tutuluyor. Son düğümü silmenin maliyeti nedir? Ayrıca bir `son` işaretçisi tutmak bu maliyeti düşürür mü? Çift yönlü başlıklı listede durum nedir?

> **Cevap**
>
> Son düğümü silmek için öncesi, yani sondan ikinci düğüm gerekir ve o ancak baştan yürünerek bulunur: $\Theta(n)$. `son` işaretçisi sona eklemeyi $O(1)$ yapar, ama sondan silmeyi hızlandırmaz; silmeden sonra `son`'u güncellemek için de yine sondan ikinci düğüm gerekir. Çift yönlü başlıklı listede son düğüm `baslik->sol`'dur, öncesi de `p->sol`'dur; silme $O(1)$'dir.

4.16 İleri okuma

- E. Horowitz, S. Sahni ve S. Anderson-Freed, *Fundamentals of Data Structures in C* (2. baskı), Bölüm 4: bu notun izlediği C gerçeklemelerinin, `avail` listesinin, dairesel ve çift yönlü bağlı listelerin ve bağlı listelerle polinom işlemlerinin kaynağıdır.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms*, Kesim 10.2: çift yönlü bağlı listeyi ve bekçi düğümünü sözde kodla, kısa ve kesin biçimde anlatır.
- D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Kesim 2.2.3–2.2.5: bağlı yerleşimi, dairesel listeleri ve çift yönlü bağlı listeleri, kullanılabilir düğüm listesiyle birlikte tarihsel kökenleriyle inceler.
- D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Kesim 3.1, Alıştırma 6: Floyd'un döngü bulma yöntemini rastgele sayı üreteçlerinin periyodu bağlamında verir.

4.17 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, MIT Press, Bölüm 10.2 (bekçili listeler).
2. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, Bölüm 4 (listeler, `avail`, dairesel ve çift yönlü bağlı listeler).
3. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3. baskı, Addison-Wesley, 1997, Kesim 2.2.3–2.2.5.
4. D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Kesim 3.1, Alıştırma 6 (Floyd döngü tespiti).

KISIM II · BÖLÜM 5

Polinomlar ve Seyrek Matrisler

Bir veri yapısının değeri, aynı bilgiyi bellekte nasıl tuttuğuna bağlıdır. Bu not, bu gerçeği iki tanıdık matematik nesnesi üzerinden işler: polinom ve matris. İkisinde de değerlerin çoğu sıfır olabilir. $5x^{100} + 1$ polinomunun yüz bir katsayısından yalnız ikisi sıfırdan farklıdır. Bilimsel hesapta kullanılan büyük matrislerde de dolu öğeler binde birin altında kalabilir. Bütün sıfırları saklayan gösterim hem belleği hem zamanı boşa harcar. Not önce polinomu katsayı dizisiyle ve terim dizisiyle tutar, iki sıralı terim dizisini birleştirerek toplar ve polinomları bağlı listeyle çarpar. Ardından seyrek matrisi üçlülerle gösterir, transpozunu iki yolla alır ve iki seyrek matrisi çarpar. Son kısım sıkıştırılmış satır gösterimini ve bağlı seyrek matrisi tanıtır. Her kod C ile verilir, adım adım izlenir ve maliyeti gerekçesiyle hesaplanır.

Kod (C): `github.com/sercankulcu/data-structures-and-algorithms-c/Bolum05`

ÖĞRENME ÇIKTILARI

Gösterim seçmek
Polinom için katsayı dizisi ile terim dizisini bellek ve zaman açısından karşılaştırmak.

Birleştirme mantığı
Sıralı iki terim dizisini `padd` ile $O(n + m)$ zamanda toplamak ve bu sınırı göstermek.

Bağlı liste ile çarpım
Polinomları bağlı listeyle çarpmak; aynı üslü terimleri birleştirmek, sıfır olanları silmek ve belleği geri vermek.

Seyrek matris
Matrisi ⟨satır, sütun, değer⟩ üçlüleriyle tutmak ve bu gösterimin hangi doluluk oranında kazandırdığını hesaplamak.

Transpoz almak
`transpose` ve `fast_transpose` fonksiyonlarını izlemek; $O(cols \cdot terms)$ ile $O(cols + terms)$ maliyetlerini karşılaştırmak.

Çarpmak
Seyrek matris çarpımını transpoz yardımıyla kurmak ve analiz etmek; sıkıştırılmış satır gösterimini tanımak.

5.1 Polinom soyut veri tipi

Polinom, lise matematiğinden tanıdık bir nesnedir. Bilgisayarda da sık karşımıza çıkar: hata denetim kodları, şifreleme, eğri uydurma ve sayısal integral polinomlarla çalışır. Bu nesneyi bir programda kullanmak için önce onunla yapılacak işlemleri, sonra bu işlemleri hızlı yapan bellek düzenini belirlemek gerekir. Birinci adım bir **soyut veri tipi** (*abstract data type*, ADT) tanımıdır. Soyut veri tipi, bir veri türünün değerlerini ve bu değerler üzerindeki işlemleri anlatır, ama bellekte nasıl tutulacağını söylemez. Bellek düzeni ikinci adımda, gerçeklemede seçilir.

5.1.1 Polinom, terim ve derece

Tanım 5.1

Polinom (*polynomial*), $A(x) = a_n x^n + \ldots + a_1 x + a_0$ biçiminde yazılan bir ifadedir. Her $a_i x^i$ parçası bir **terim**dir; a_i terimin **katsayısı**, i ise **üssü**dür. Polinomun **derecesi**, katsayısı sıfırdan farklı olan en büyük üstür. Soyut düzeyde polinom, ⟨üs, katsayı⟩ çiftlerinden oluşan bir kümedir; her üs kümede en çok bir kez geçer.

Tanımdaki küme bakışı önemlidir. $A(x) = 3x^{20} + 2x^5 + 4$ polinomu {⟨20, 3⟩, ⟨5, 2⟩, ⟨0, 4⟩} kümesidir. Sıfır katsayılı terimler kümeye girmez: x^{19}'dan x^6'ya kadar bütün terimlerin katsayısı sıfırdır ve hiçbiri yazılmaz. Polinomun derecesi 20'dir. Sabit terim $4 = 4x^0$ olduğundan üssü 0'dır. Hiç terimi olmayan polinom **sıfır polinomudur**; boş kümeye karşılık gelir.

Aynı üssün iki kez geçmemesi kuralı, toplama ve çarpma kodlarının da temel koşuludur. $2x^3 + 5x^3$ gibi bir yazım kümeye uygun değildir; bu iki terim tek bir $7x^3$ terimine indirgenmelidir. Çarpım kodunun en çok uğraştığı iş bu indirgemedir.

5.1.2 İşlemler

Horowitz ve arkadaşlarının ders kitabı, polinom soyut veri tipini Tablo 5.1'deki işlemlerle tanımlar. İşlemlerin adları kitaptaki İngilizce adlardır; anlamları Türkçe verilmiştir.

İşlem	Anlamı
`Zero()`, `IsZero(p)`	sıfır polinomu kurar / p'nin sıfır polinomu olup olmadığını sınar
`Coef(p, e)`	e üslü terimin katsayısını verir; böyle bir terim yoksa 0
`LeadExp(p)`	en büyük üssü, yani dereceyi verir
`Attach(p, c, e)`, `Remove(p, e)`	$c{\cdot}x^e$ terimini ekler / e üslü terimi çıkarır
`SingleMult(p, c, e)`	$p \cdot c{\cdot}x^e$ çarpımını verir
`Add(p1, p2)`, `Mult(p1, p2)`	iki polinomun toplamını / çarpımını verir

Tablo 5.1: Polinom soyut veri tipinin işlemleri.

Bu liste, polinomla yapılacak her işi kapsar. `Add` ve `Mult` öteki işlemlerle yazılabilir. Toplama, iki polinomun en büyük üslerini `LeadExp` ile karşılaştırıp uygun terimi `Attach` ile sonuca ekleyerek ilerler. Çarpma, birinci polinomun her terimi için `SingleMult` çağırıp ara sonuçları toplar. Soyut veri tipinin yararı burada görülür. Toplama algoritması, polinomun bellekte dizi mi bağlı liste mi olduğunu bilmeden yazılabilir. Gerçeklemeyi değiştiren programcı yalnız bu işlemleri yeniden yazar.

5.1.3 Toplama ve çarpma kuralları

Toplamada aynı üslü terimlerin katsayıları toplanır:

> $$A(x) + B(x) = \Sigma\ (a_i + b_i) \cdot x^i$$

Bir üs yalnız bir polinomda varsa, öteki polinomun o üsteki katsayısı 0 sayılır ve terim olduğu gibi sonuca geçer. İki katsayının toplamı 0 ise terim sonuçtan düşer. Örneğin $(4x^3 + 2x^2 + 5) + (3x^3 - 2x^2 + x) = 7x^3 + x + 5$'tir; x^2 terimleri birbirini götürür.

Çarpmada her terim öteki polinomun her terimiyle çarpılır. İki terimin çarpımında **üsler toplanır, katsayılar çarpılır**: $a_i x^i \cdot b_j x^j = a_i b_j\, x^{i+j}$.

> $$A(x) \cdot B(x) = \Sigma_i\ (\ a_i x^i \cdot \Sigma_j\ b_j x^j\ )$$

n terimli bir polinom m terimli bir polinomla çarpılınca $n \cdot m$ çarpım oluşur. Bu çarpımların bazılarının üssü aynı çıkabilir. $(x + 1)(x - 1)$ çarpımında dört terim oluşur: x^2, $-x$, x ve -1. Ortadaki iki terimin üssü 1'dir; toplamları 0 olduğundan ikisi de kaybolur ve sonuç $x^2 - 1$ olur. Çarpım kodunun asıl zorluğu, aynı üslü terimleri bulup birleştirmek ve sıfırlananları atmaktır.

5.2 Polinomu dizide tutmak

C'de polinomu tutmanın en kolay yolu bir dizidir. Ama dizinin neyi tutacağı konusunda iki farklı seçim vardır: her üs için bir hücre ayırmak ya da yalnız var olan terimleri yazmak. Bu iki seçim, aynı polinom için çok farklı bellek ve zaman maliyetleri doğurur.

5.2.1 Birinci yöntem: katsayı dizisi

Birinci yöntem bütün katsayıları üsse göre dizinlenmiş bir dizide tutar. `coef[i]`, x^i'nin katsayısıdır:

```
#define MAX_DEGREE 101   /* azami derece + 1 */
typedef struct {
  int degree;               /* en büyük üs */
  float coef[MAX_DEGREE];   /* coef[i]: x^i katsayısı */
} polynomial;
polynomial P;
```

`MAX_DEGREE` makrosu en büyük derecenin bir fazlasıdır; 101 hücreli dizi 0'dan 100'e kadar üsleri tutar. Yapının `degree` alanı, katsayısı sıfırdan farklı en büyük üssü saklar. Bu alan sayesinde derece her istendiğinde dizi sondan taranmaz. Yapı tanımı `typedef` ile `polynomial` adını alır; `polynomial P;` satırı bu türden bir değişken bildirir.

Bu gösterimde işlemler çok basittir. `Coef(p, e)` doğrudan `p.coef[e]` 'dir ve *O(1)* sürer. `LeadExp(p)` , `p.degree` 'dir. Toplama, iki dizinin aynı indisli hücrelerini toplar. Bu notta katsayı dizisiyle toplamayı gösteren kısa bir fonksiyon yazılmıştır:

```
/* katsayı dizisiyle toplama: d = a + b */
void dizi_topla(const polynomial *a, const polynomial *b,
                polynomial *d) {
  int i, n = a->degree > b->degree ? a->degree : b->degree;
  for (i = 0; i <= n; i++) {
    float x = i <= a->degree ? a->coef[i] : 0;
    float y = i <= b->degree ? b->coef[i] : 0;
    d->coef[i] = x + y;
  }
  d->degree = n;
  while (d->degree > 0 && d->coef[d->degree] == 0)
    d->degree--;               /* baştaki sıfırları at */
}
```

Fonksiyon iki polinomun derecelerinden büyük olanına kadar döner. Bir polinomun derecesi aşılmışsa o polinomun katsayısı 0 alınır. Toplamın derecesi önce *n* kabul edilir; sonra baştaki sıfırlar atılır. *(2x³ + 1) + (−2x³ + x + 4)* toplamında x^3 katsayısı 0 çıkar ve son döngü dereceyi 3'ten 1'e indirir; sonuç *x + 5*'tir. Zaman maliyeti iki polinomun derecelerinin büyüğüyle orantılıdır: *O(max(deg A, deg B))*.

Sorun bellektedir. Katsayısı 0 olan terimler de tutulur. *P(x) = 5x¹⁰⁰* polinomunun tek terimi vardır, ama dizi 101 hücre ayırır ve `coef[0..99]` hücrelerinin hepsi 0'dır (Şekil 5.1). Bir `float` 4 bayt olduğundan bu yapı, `degree` alanıyla birlikte 408 bayt kaplar. Anlamlı bilgi yalnız iki sayıdır: üs 100 ve katsayı 5. Toplama da aynı israfı zamana taşır: iki tek terimli polinomu toplamak için 101 hücrede gezinilir.



P(x) = 5x¹⁰⁰ için coef[0..99] boşuna sıfır tutar: 101 float, 1 anlamlı

Şekil 5.1: *5x¹⁰⁰* polinomunun katsayı dizisi. Yüz bir hücreden yalnız sonuncusu anlamlıdır; geri kalanı boşuna sıfır tutar.

Katsayı dizisinin bir sınırı daha vardır. Derece `MAX_DEGREE - 1` 'i aşamaz. x^{1000} terimli bir polinom için dizi bin bir hücreli olmalıdır; bu durumda bütün polinomlar, tek terimli olsalar bile, dört kilobayt yer kaplar. Seyrek polinom, yani terimlerinin çoğunun katsayısı 0 olan polinom için bu yöntem uygun değildir. Katsayıların çoğu doluysa, örneğin $x^4 + 3x^3 - x^2 + 2x + 7$ gibi bir polinomda, katsayı dizisi hem sade hem hızlıdır.

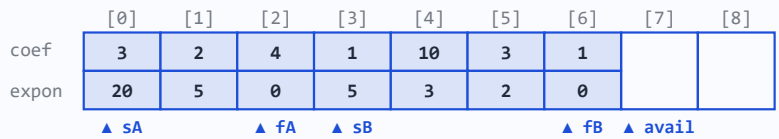
5.2.2 İkinci yöntem: terim dizisi

İkinci yöntem yalnız sıfırdan farklı terimleri saklar. Her terim bir katsayı ve bir üs çiftidir. Bütün polinomların terimleri tek bir ortak dizide durur:

```
#define MAX_TERMS 101   /* azami terim sayısı */
typedef struct {
  float coef;           /* katsayı */
  int expon;            /* üs */
} polynomial;
polynomial P[MAX_TERMS]; /* tüm polinomlar burada */
int avail = 0;          /* ilk boş konum */
```

Yapının adı yine `polynomial` 'dır, ama artık tek bir terimi temsil eder: `coef` katsayıyı, `expon` üssü tutar. `P` dizisi programdaki bütün polinomların terimlerini barındırır. `avail` değişkeni dizideki ilk boş konumu gösterir; yeni bir terim her zaman `P[avail]` 'e yazılır ve `avail` bir artar.

Bir polinom, `P` içinde ardışık bir aralıktır. Polinomu bulmak için iki indis yeter: ilk teriminin indisi `start` ve son teriminin indisi `finish`. Terimler **büyük üsten küçüğe** sıralı tutulur. Bu sıra, toplama algoritmasının hızlı çalışmasının anahtarıdır. Şekil 5.2, $A(x) = 3x^{20} + 2x^5 + 4$ ve $B(x) = x^5 + 10x^3 + 3x^2 + 1$ polinomlarının ortak dizideki yerini gösterir.



Şekil 5.2: İki polinomun terimlerini tutan ortak `P` dizisi. A `P[0..2]`'de (`sA = 0`, `fA = 2`), B `P[3..6]`'da (`sB = 3`, `fB = 6`) durur; ilk boş konum `avail = 7`'dir.

Bu gösterimde $5x^{100}$ polinomu tek bir hücre kaplar: ⟨5, 100⟩. Bir terim `float` katsayı ile `int` üsten oluşur ve 8 bayttır. Katsayı dizisinin 408 baytı yerine 8 bayt yeter. Derecenin bir üst sınırı da kalmaz; üs `int`'e sığdığı sürece her değeri alabilir. Sınırlı olan, bütün polinomların toplam terim sayısıdır: `MAX_TERMS` = 101.

Sıralı terimlerin değerini bir benzetme açıklar. Alfabetik sıralı iki öğrenci listesini tek bir sıralı listeye dönüştürmek için iki listenin başından aynı anda yürümek yeter. Her adımda iki başın önce geleni yeni listeye yazılır. Listeler sırasız olsaydı her isim için öteki listenin tamamını taramak gerekirdi. Bu iki işaretçili yürüyüşe **birleştirme** (*merge*) denir. Dersin hocası da aynı noktayı vurgular: birleştirme mantığı, sıralı yapılar üzerinde daha etkin algoritmalar üretmeyi sağlar. Polinom toplamanın tamamı bu fikre dayanır.

5.2.3 İki gösterimin karşılaştırması

Hangi gösterimin daha iyi olduğu polinomun yapısına bağlıdır. Tablo 5.2, iki gösterimi bellek ve temel işlemler açısından karşılaştırır. Tabloda d derece, t sıfırdan farklı terim sayısıdır.

Ölçüt	Katsayı dizisi	Terim dizisi
Bellek	`MAX_DEGREE` hücre (sabit)	t terim (her biri 8 bayt)
`Coef(p, e)`	$O(1)$	$O(t)$ (ya da ikili aramayla $O(\log t)$)
`LeadExp(p)`	$O(1)$ (`degree` alanı)	$O(1)$ (ilk terim)
Toplama	$O(d)$	$O(t_A + t_B)$
Derece sınırı	`MAX_DEGREE - 1`	yok (yalnız toplam terim sınırı)
Uygun olduğu durum	dolu, düşük dereceli polinom	seyrek polinom

Tablo 5.2: Katsayı dizisi ile terim dizisinin karşılaştırması.

Terim dizisinde `Coef` yavaşlar, çünkü istenen üs dizide aranmalıdır. Ama polinom işlemlerinin çoğu, tek tek katsayıları sorgulamak yerine terimler üzerinde sırayla gezinir. Toplama ve çarpma böyle işlemlerdir ve terim dizisinde terim sayısıyla orantılı zaman alır. $3x^{20} + 2x^5 + 4$ polinomunun katsayı dizisiyle toplanması için 21 hücrede gezinilir; terim dizisinde 3 terim yeter.

> **Sık yapılan hata**
>
> Terim dizisine sıfır katsayılı terim yazmak ya da aynı üssü iki kez eklemek. Gösterimin bütün algoritmaları her üssün en çok bir kez geçtiğini ve katsayıların sıfırdan farklı olduğunu varsayar. $0x^5$ gibi bir terim yer harcar, polinomun derecesini yanlış gösterir ve `IsZero` sınamasını bozar: tek terimi 0 olan bir polinom sıfır polinomu olduğu hâlde boş görünmez.

5.3 Polinom toplama: padd

Terim dizisi gösteriminde iki polinomu toplayan fonksiyonun adı `padd`'dir (*polynomial add*). Fonksiyon `P[sa..fa]`'daki A ile `P[sb..fb]`'deki B'yi toplar ve sonucu `avail`'den başlayarak `P`'nin boş kısmına yazar.

5.3.1 İki işaretçiyle birleştirme

İki polinomun terimleri büyük üsten küçüğe sıralıdır. `sa` ve `sb` indisleri iki polinomun henüz işlenmemiş ilk terimini gösterir; bunlara **işaretçi** denir. C işaretçisi değil, dizi indisidirler. Her adımda iki işaretçinin gösterdiği terimlerin üsleri karşılaştırılır. Üç durum vardır:

- **Üsler eşit.** Katsayılar toplanır. Toplam sıfırdan farklıysa sonuca bu üsle eklenir; sıfırsa terim kaybolur. İki işaretçi de bir ilerler.
- **A'nın üssü büyük.** A'daki bu üsse sahip bir terim B'de yoktur. B'nin kalan terimlerinin hepsi B'nin şu anki teriminden küçük üslüdür, şu anki terim de A'nınkinden küçüktür. Öyleyse bu üs B'nin hiçbir yerinde geçmez. A'nın terimi olduğu gibi sonuca eklenir ve yalnız A'nın işaretçisi ilerler.
- **B'nin üssü büyük.** Durum simetriktir: B'nin terimi eklenir, yalnız B'nin işaretçisi ilerler.

Polinomlardan birinin terimleri bitince döngü durur. Öteki polinomda kalan terimlerin üsleri, sonuca yazılmış bütün terimlerden küçüktür; bu terimler olduğu gibi sonuca kopyalanır.

İkinci durumun gerekçesi algoritmanın doğruluğunun özüdür ve yalnız sıralılık sayesinde geçerlidir. Terimler sırasız olsaydı, A'nın bir teriminin B'de karşılığı olup olmadığını anlamak için B'nin tamamını taramak gerekirdi. Sıralı dizide B'nin şu anki terimi bu soruya tek başına cevap verir.

5.3.2 COMPARE ve attach

`padd` iki yardımcı kullanır. Birincisi üç yönlü karşılaştırma yapan bir makro, ikincisi sonuca terim ekleyen bir fonksiyondur:

```
#define COMPARE(x, y) \
  ((x) < (y) ? -1 : (y) < (x) ? 1 : 0)

void attach(float coef, int expon) {
  if (avail >= MAX_TERMS) {
    fprintf(stderr, "Dizi tasmasi!\n"); exit(1);
  }
  P[avail].coef = coef;
  P[avail++].expon = expon;
}
#define EK(k) attach(P[k].coef, P[k].expon)
```

`COMPARE(x, y)` , $x < y$ ise -1, $x > y$ ise 1, eşitse 0 verir. İç içe iki koşul işleciyle yazılmıştır. Makro parametreleri her geçişte parantez içindedir, çünkü önişlemci makroyu metin olarak yerine koyar. `COMPARE(a + 1, b)` gibi bir çağrıda parantezler, işleç önceliğinin ifadeyi yanlış gruplamasını önler. Sonucun üç değerli olması, `padd` 'in `switch` ile yazılmasını sağlar.

`attach` , bir terimi `P[avail]` 'e yazar ve `avail` 'i bir artırır. Yazmadan önce dizinin dolup dolmadığını denetler; doluysa standart hata akışına bir ileti yazıp programı sonlandırır. Bu denetim olmasaydı `P[101]` 'e yazılırdı. Dizinin sınırı dışına yazmak C'de tanımsız davranıştır: komşu değişkenler bozulabilir ya da program çökebilir. `P[avail++].expon = expon;` satırı son ek artırma işlecini kullanır: önce `P[avail]` 'in üssü yazılır, sonra `avail` artar.

`EK(k)` makrosu, `P[k]` terimini olduğu gibi sonuca kopyalamanın kısa yazımıdır. `EK(sb)` , önişlemciden sonra `attach(P[sb].coef, P[sb].expon)` olur.

> **Sık yapılan hata**
>
> `attach` 'ın katsayı parametresini `int` bildirmek: `void attach(int coef, int expon)` . Bu durumda `float` katsayı fonksiyona geçerken tam sayıya kırpılır ve *2,5* katsayısı 2 olur. Derleyici bu dönüşümü kendiliğinden yapar; `-Wall -Wextra` ile bile uyarı vermez. Uyarı ancak `-Wconversion` seçeneğiyle çıkar. Parametrenin türü, saklanan alanın türüyle aynı olmalıdır: `float coef` .

5.3.3 padd() kodu

```
void padd(int sa,int fa,int sb,int fb,int*sd,int*fd){
  float c; *sd = avail;              /* D'nin başı */
  while (sa <= fa && sb <= fb)
    switch (COMPARE(P[sa].expon, P[sb].expon)) {
    case -1: EK(sb); sb++; break;     /* B büyük */
    case  1: EK(sa); sa++; break;     /* A büyük */
    case  0: c = P[sa].coef + P[sb].coef;
      if (c) attach(c, P[sa].expon); /* 0: ekleme */
      sa++; sb++;
    }
  for (; sa <= fa; sa++) EK(sa);      /* A'da kalan */
  for (; sb <= fb; sb++) EK(sb);      /* B'de kalan */
  *fd = avail - 1;
}
```

Fonksiyon altı parametre alır. İlk dördü girdi aralıklarıdır: A `P[sa..fa]`'da, B `P[sb..fb]`'de durur. Son ikisi çıktıdır: `sd` ve `fd`, sonucun başlangıç ve bitiş indislerinin yazılacağı değişkenlerin adresleridir. C'de bir fonksiyon tek değer döndürebilir. İki değer döndürmek için çağıran taraf iki değişkenin adresini verir; fonksiyon bu adreslere yazar. `*sd = avail;` ifadesi, çağıranın değişkenine `avail`'in o anki değerini koyar.

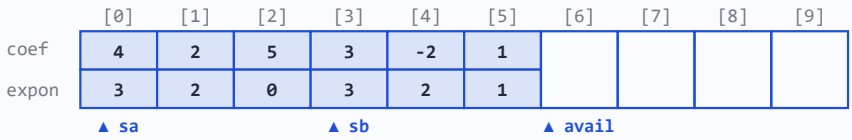
Kodun yapısı şöyledir:

1. **2. satır:** Sonucun ilk terimi `P[avail]`'e yazılacağından `*sd` şimdiden belirlenir.
2. **3–10. satırlar:** İki polinomda da terim kaldıkça döngü döner. `switch`, üslerin karşılaştırma sonucuna göre üç daldan birini seçer. `case -1`, B'nin üssü büyük olduğunda B'nin terimini ekler. `case 1`, A'nınkini ekler. `case 0`, katsayıları toplar; toplam 0 değilse `attach` çağrılır. `if (c)` koşulu `c != 0` anlamındadır.
3. **11–12. satırlar:** Döngü bitince iki polinomdan en çok birinde terim kalmıştır. İki `for` döngüsünden yalnız biri iş yapar; öteki koşulu baştan yanlış bulur.
4. **13. satır:** Son eklenen terim `P[avail - 1]`'dedir; `*fd` bu indisi alır.

`case 0` dalının sonunda `break` yoktur. Bu dal `switch`'in son dalı olduğundan kontrol zaten `switch`'in sonuna düşer; bir sonraki dala geçiş olmaz. Yine de son dala da `break` yazmak iyi bir alışkanlıktır: sonradan altına yeni bir dal eklenirse kod istemeden o dala akmaz.

5.3.4 Kod izleme: A + B

$A(x) = 4x^3 + 2x^2 + 5$ polinomu `P[0..2]`'de, $B(x) = 3x^3 - 2x^2 + x$ polinomu `P[3..5]`'te durur ve `avail = 6`'dır (Şekil 5.3). `padd(0, 2, 3, 5, &sd, &fd)` çağrısı Tablo 5.3'teki gibi ilerler.



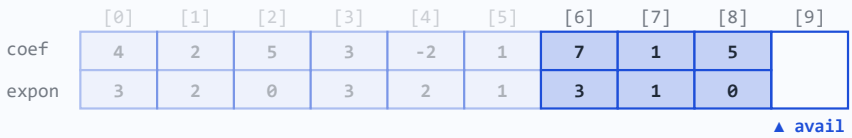
Şekil 5.3: `padd`'in başlangıcı. `sa = 0` A'nın, `sb = 3` B'nin ilk terimini gösterir; sonuç `avail = 6`'dan başlayacaktır.

Adım	`sa`	`sb`	Üsler	`COMPARE`	İşlem	`avail`
0	0	3	—	—	`*sd = 6`	6
1	0	3	3, 3	0	$c = 4 + 3 = 7$; `attach(7, 3)`: `P[6] = 7x³`; ikisi de ilerler	7
2	1	4	2, 2	0	$c = 2 + (-2) = 0$; eklenmez; ikisi de ilerler	7
3	2	5	0, 1	−1	`EK(5)`: `P[7] = 1x¹`; yalnız `sb` ilerler	8
4	2	6	—	—	`sb = 6 > fb = 5`: döngü biter	8
5	3	6	—	—	A'da kalan: `EK(2)`: `P[8] = 5x⁰`	9

Adım	sa	sb	Üsler	COMPARE	İşlem	avail
6	3	6	—	—	`*fd = 9 - 1 = 8`	9

Tablo 5.3: `padd(0, 2, 3, 5, &sd, &fd)` izlemesi.

Tabloda `sa` ve `sb` sütunları adımın başındaki değerleri, `avail` sütunu adımın sonundaki değeri gösterir. İlk iki turda üsler eşittir. Birincide katsayılar toplanıp $7x^3$ yazılır. İkincide $2x^2$ ile $-2x^2$ birbirini götürür; `c` sıfır olduğu için `attach` çağrılmaz ve `avail` değişmez. Üçüncü turda A'nın kalan terimi $5x^0$, B'nin kalan terimi x^1'dir. `COMPARE(0, 1) = -1` olduğu için B'nin terimi eklenir. Bu adımdan sonra B biter ve döngü koşulu yanlış olur. A'da kalan 5, ilk `for` döngüsüyle kopyalanır.



Şekil 5.4: `padd` bittiğinde: sonuç $D(x) = 7x^3 + x + 5$ `P[6..8]` 'de durur, `avail = 9` 'dur. Soluk hücreler girdi polinomlarıdır.

Sonuç $D(x) = 7x^3 + x + 5$'tir ve `P[6..8]` aralığındadır: `sd = 6`, `fd = 8`. Program bu girdiyle çalıştırıldığında `attach` üç kez çağrılır ve `avail` 9 olur. Girdi polinomları yerinde kalır; `padd` onları değiştirmez.

Örnek 5.1

Şekil 5.2'deki polinomlar toplanırsa: $A = 3x^{20} + 2x^5 + 4$ (`P[0..2]`), $B = x^5 + 10x^3 + 3x^2 + 1$ (`P[3..6]`), `avail = 7`.

1. 20 ile 5: A büyük; $3x^{20} \to$ `P[7]`.
2. 5 ile 5: eşit; $2 + 1 = 3$, $3x^5 \to$ `P[8]`.
3. 0 ile 3: B büyük; $10x^3 \to$ `P[9]`.
4. 0 ile 2: B büyük; $3x^2 \to$ `P[10]`.
5. 0 ile 0: eşit; $4 + 1 = 5 \to$ `P[11]`. İki polinom da biter; kalan kopyalama döngüleri iş yapmaz.

Sonuç $3x^{20} + 3x^5 + 10x^3 + 3x^2 + 5$'tir ve `P[7..11]` 'de durur. Beş tur döner, beş `attach` yapılır. Katsayı dizisinde aynı toplama 21 hücrede gezinirdi.

5.3.5 Karmaşıklık

Teorem 5.1 (padd'in maliyeti)

A'da n, B'de m terim varsa `padd` $\Theta(n + m)$ zamanda çalışır ve en çok $n + m$ terim yazar.

İspat. *Üst sınır.* `while` döngüsünün her turunda `sa` ya da `sb` 'den en az biri bir artar. `sa` en çok n, `sb` en çok m kez artabilir. Öyleyse döngü en çok $n + m$ tur döner. Döngüden sonra kalan terimleri kopyalayan iki `for` döngüsü de aynı işaretçileri artırır. İşaretçilerin toplam artışı $n + m$'yi geçemez. Öyleyse bütün döngülerin toplam tur sayısı da en çok $n + m$'dir. Her turda sabit sayıda işlem yapılır (bir karşılaştırma, en çok bir toplama ve bir `attach`). Zaman $O(n + m)$'dir.

Alt sınır. Her tur en çok iki işaretçiyi birer ilerletir. Bütün terimlerin işlenmesi için işaretçilerin toplam $n + m$ ilerlemesi gerekir; bu da en az $(n + m)/2$ tur demektir. Zaman $\Omega(n + m)$'dir. İki sınır birlikte $\Theta(n + m)$ verir. □

En iyi durum, bütün üslerin ortak olduğu durumdur ($n = m$). Her tur `case 0` dalına girer ve iki işaretçi birlikte ilerler; döngü n tur döner. Bu durumda maliyet $\Omega(n)$'dir; $n = m$ olduğundan bu da $\Theta(n + m)$ ile aynı derecedir. En kötü durum hiç ortak üs olmamasıdır. O zaman her terim ayrı ayrı yazılır ve $n + m$ `attach` yapılır. On terimli iki polinomun toplamı bu yüzden en çok 20 `attach` çağrısı yapar. Bu sayı programla da doğrulanmıştır. Üsleri tek sayı olan on terimli bir polinom, üsleri çift sayı olan on terimli bir polinomla toplanınca 20 çağrı sayılır. İki eş polinom toplanınca 10 çağrı sayılır.

Terimler sırasız tutulsaydı ne olurdu? A'nın her terimi için B'de aynı üslü terim aranmalı, bulunamazsa terim sonuca uygun yere yerleştirilmeliydi. Arama terim başına $O(m)$ sürer ve n terim için $O(nm)$ eder. Sonucu sıralı tutmak için araya ekleme yapılırsa her ekleme dizideki terimleri kaydırır; bu da toplamda $O((n+m)^2)$ düzeyine çıkar. Hocanın deyişiyle, istenen $O(n+m)$ ise ve daha iyisi yapılabiliyorsa ne âlâ. Ama girdinin her terimine en az bir kez bakmak gerektiğinden toplama için $n+m$'den hızlı bir algoritma olamaz; `padd` en iyi derecededir.

5.3.6 Ortak dizinin bedeli

Ortak `P` dizisi basit bir bellek yöneticisidir: `avail` yalnız ileri gider. Her `padd` çağrısı sonucunu dizinin sonuna ekler. Bir ara sonuçla işi biten polinomun terimleri dizide kalır ve yerleri geri kazanılmaz. Uzun bir hesapta dizi bu artıklarla dolar ve `attach` "Dizi tasmasi!" iletisiyle programı durdurur, oysa dizideki canlı polinomların toplam boyu küçük olabilir.

Bu sorunun iki çözümü vardır. Birincisi **sıkıştırma**dır (*compaction*). Dizi dolunca hâlâ kullanılan polinomlar dizinin başına doğru kaydırılır ve başlangıç ile bitiş indisleri güncellenir. Sıkıştırma, canlı terim sayısıyla orantılı zaman alır ve bütün polinomların indislerini değiştirdiği için dikkatli yazılmalıdır. İkinci çözüm, polinomu bağlı listeyle tutmaktır. Her terim kendi düğümünde durur; işi biten düğüm `free` ile hemen geri verilir.

5.4 Dizi gösterimiyle çarpım ve çok polinomlu işlemler

`padd` iki polinomu toplar. Çarpım, çok sayıda polinomun toplamı ve çarpımı da aynı dizi gösterimiyle yazılabilir. Sunumda ödev olarak verilen bu üç sorunun çözümleri aşağıda ayrıntılı olarak verilir.

5.4.1 pmult(): tek terimle çarpıp toplamak

Çarpma kuralı bir algoritma önerir. A'nın her $a_i x^{e_i}$ terimi için B, bu tek terimle çarpılır (`SingleMult`). Ortaya çıkan m terimli polinom ara sonuca `padd` ile eklenir. B'nin terimleri sıralı olduğundan tek terimle çarpım da sıralı bir polinom verir: her üse aynı e_i eklenir ve sıra bozulmaz. Bu notta ödevin çözümü olarak yazılan fonksiyon şöyledir:

```
/* A = P[sa..fa], B = P[sb..fb]; çarpım P[*sd..*fd] */
void pmult(int sa, int fa, int sb, int fb,
           int *sd, int *fd) {
  int i, j, ss = avail, sf = avail - 1;  /* boş ara sonuç */
  int ts, tf, ys, yf;
  for (i = sa; i <= fa; i++) {
    ts = avail;                      /* B · a_i x^e_i */
    for (j = sb; j <= fb; j++)
      attach(P[i].coef * P[j].coef,
             P[i].expon + P[j].expon);
    tf = avail - 1;
    padd(ss, sf, ts, tf, &ys, &yf);  /* ara sonuca ekle */
    ss = ys;  sf = yf;
  }
  *sd = ss;  *fd = sf;
}
```

Ara sonuç `P[ss..sf]` 'dir. Başta boştur: `sf = ss - 1` olduğu için aralıkta terim yoktur. Dış döngünün her turunda `ts` ile `tf` arasına $B \cdot a_i x^{e_i}$ yazılır. Sonra `padd` eski ara sonuçla bu geçici polinomu toplar ve yeni ara sonucu dizinin sonuna koyar. İlk turda ara sonuç boş olduğundan `padd` 'in ana döngüsü hiç dönmez ve geçici polinom olduğu gibi kopyalanır.

> **Örnek 5.2**
>
> $A = x + 1$ `P[0..1]` 'de, $B = x - 1$ `P[2..3]` 'te; `avail = 4` , ara sonuç boş (`ss = 4` , `sf = 3`).
>
> 1. $i = 0$ (terim x): $x \cdot B = x^2 - x \rightarrow$ `P[4..5]` . `padd(4, 3, 4, 5)` boş ara sonuca ekler; yeni ara sonuç `P[6..7]` $= x^2 - x$.
> 2. $i = 1$ (terim 1): $1 \cdot B = x - 1 \rightarrow$ `P[8..9]` . `padd(6, 7, 8, 9)` : x^2 eklenir, $-x + x = 0$ atlanır, -1 kopyalanır. Ara sonuç `P[10..11]` = $x^2 - 1$.
>
> Fonksiyon `sd = 10` , `fd = 11` döndürür ve `avail` 12 olur. On iki konumun yalnız ikisi sonucu tutar; `P[4..9]` artık kullanılmayan ara polinomlardır.

Zaman analizi için A'da n, B'de m terim olsun. i. turda geçici polinom m terimlidir ve $O(m)$ zamanda yazılır. Ara sonucun boyu en çok $i \cdot m$'dir, çünkü şimdiye kadar $i \cdot m$ çarpım üretilmiştir. `padd` bu turda $O(im + m)$ zaman alır. Bütün turların toplamı:

> $\Sigma_{i=1}^{n} O(im + m) = O(m \cdot n(n + 1)/2 + nm) = O(n^2 m)$

Sunumdaki $O(n \cdot (nm + m)) = O(n^2 m)$ yazımı, ara sonucu her turda en büyük boyuyla, yani nm ile sınırlayarak aynı sonuca ulaşır. Sonuç simetrik değildir: n'nin karesi, m'nin birinci kuvveti vardır. Küçük polinom dış döngüye alınırsa maliyet $O(min(n, m)^2 \cdot max(n, m))$ olur. 2 terimli bir polinom 100 terimli bir polinomla çarpılsın. İki terimli olanı dışa almak yaklaşık $2^2 \cdot 100 = 400$ birimlik iş yapar; tersi $100^2 \cdot 2 = 20.000$ birim.

Bellek maliyeti daha da kötüdür. Her tur m geçici terim ve en çok im yeni ara sonuç terimi yazar ve bunların hiçbiri geri verilmez. Toplam tüketim de $O(n^2 m)$'dir. Ortak dizinin bu bedeli, uzun hesaplarda sıkıştırmayı ya da bağlı listeyi zorunlu kılar. Programda $(3x^{20} + 2x^5 + 4)(x^5 + 10x^3 + 3x^2 + 1)$ çarpımı 11 terimli doğru sonucu verir, ama `avail` 7'den 42'ye çıkar: 35 konumun yalnız 11'i sonuçtur.

5.4.2 n polinomu toplamak

Çok sayıda polinomla çalışan bir program, her polinomun yerini iki tabloda tutabilir. `start[k]`, k. polinomun `P` içindeki ilk teriminin, `finish[k]` son teriminin indisidir. n polinomu toplamanın en kolay yolu art arda `padd` çağırmaktır: önce ilk ikisi toplanır, sonra sonuca üçüncüsü eklenir ve böyle sürer.

Bütün polinomlardaki toplam terim sayısı N olsun. Art arda toplamada ara sonuç büyür ve her `padd` onun üzerinde baştan gezinir. En kötü durumda ilk polinomun terimleri $n - 1$ kez kopyalanır. Toplam maliyet $O(n \cdot N)$'dir. Örneğin her biri 100 terimli 100 polinom, ortak üsleri yoksa, yaklaşık $100 \cdot 10.000 / 2 = 500.000$ terim kopyalaması gerektirir.

Daha iyi bir yol, n polinomun hepsini aynı anda birleştirmektir. Her polinom için bir işaretçi tutulur. Her adımda n işaretçinin gösterdiği terimler arasından en büyük üslü olanı seçilir; aynı üslü olanlar toplanır. En büyüğü n terim arasından düz arayarak bulmak her adımda $O(n)$ sürer ve toplam $O(n \cdot N)$ eder; bir kazanç yoktur. En büyük üssü **öncelikli kuyruk** (*priority queue*) ile, örneğin bir heap ile seçmek her adımı $O(\log n)$'ye indirir. Toplam maliyet $O(N \log n)$ olur. Aynı sonuç, polinomları bir turnuva ağacı gibi ikişer ikişer toplayarak da elde edilir. Her turda polinom sayısı yarıya iner; her tur toplam $O(N)$ iş yapar ve $\lceil \log_2 n \rceil$ tur vardır.

5.4.3 n polinomu çarpmak

Çarpımda ara sonuçlar hızla büyür. n polinom soldan sağa çarpılırsa, P_1P_2 çarpımı P_3 ile, sonuç P_4 ile çarpılır ve böyle sürer. Her adımda büyük bir ara sonuç küçük bir polinomla çarpılır. `pmult`'un maliyeti dış döngüdeki polinomun terim sayısının karesiyle büyüdüğü için büyük ara sonuç her zaman iç döngüye alınmalıdır.

İkişer ikişer, ağaç biçiminde çarpmak boyutları dengeler. Önce P_1P_2, P_3P_4, … çarpılır; sonra bu çarpımlar ikişer ikişer çarpılır. Her düzeyde çarpılan polinomlar benzer boydadır. Ara sonuçları tutmak için aynı `start` / `finish` tabloları kullanılabilir; ama dizi gösteriminde artıklar hızla biriktiğinden bağlı liste daha uygun bir seçimdir.

> **Sık yapılan hata**
>
> Çarpım sonucunu doğrudan dizinin sonuna terim terim yazıp aynı üslü terimleri birleştirmemek. $(x + 1)(x - 1)$ için $x^2 - x + x - 1$ yazılırsa dizi sıralı kalmaz ve aynı üs iki kez geçer. Sonraki her `padd` çağrısı bu polinomla yanlış sonuç verir, çünkü birleştirme mantığı her üssün tek ve sıralı olduğunu varsayar. `pmult` bu yüzden her tek terimli çarpımı `padd` ile ekler.

5.5 Bağlı liste ile polinom

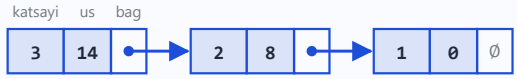
Dizi gösteriminin iki zayıf noktası vardır: toplam terim sayısının önceden bilinmesi gerekir ve işi biten terimlerin yeri geri kazanılmaz. Bağlı liste ikisini de çözer. Her terim, `malloc` ile ayrılan bir düğümde durur. Düğümler birbirine işaretçilerle bağlanır ve terim sayısı programın çalışması sırasında serbestçe büyüyüp küçülür. Bir terim artık gerekmediğinde düğümü `free` ile sisteme geri verilir.

5.5.1 Düğüm yapısı

```c
typedef struct pol_oge *pol_gosterge;
typedef struct pol_oge {
  int katsayi;
  int us;
  pol_gosterge bag;
} pol_oge;
```

İlk satır, `struct pol_oge` türüne işaret eden bir işaretçi türünü `pol_gosterge` adıyla tanımlar. Bu satır yapının tanımından önce gelir. C buna izin verir, çünkü bir yapıya işaretçi bildirmek için yapının içeriğini bilmek gerekmez; adın bir yapı olduğunu bilmek yeter. Ardından gelen tanım düğümün üç alanını verir: `katsayi`, `us` ve listedeki sonraki düğümün adresini tutan `bag`. Yapı kendine işaret eden bir alan içerdiği için **kendine dönüşlü yapı** (*self-referential structure*) adını alır. Bu gösterimde katsayılar tam sayıdır.

Liste, terimleri yine üsse göre azalan sırada tutar ve yalnız sıfırdan farklı terimleri içerir. Listenin ilk düğümünün adresi bir `pol_gosterge` değişkeninde saklanır; son düğümün `bag` alanı `NULL`'dır. Şekil 5.5, $a = 3x^{14} + 2x^{8} + 1$ polinomunun listesini gösterir.



Şekil 5.5: $a = 3x^{14} + 2x^{8} + 1$ polinomunun bağlı listesi. Her düğüm katsayı, üs ve sonraki düğümün adresini tutar; son düğümün bağı `NULL`'dır (∅).

Bir düğüm, 64 bitlik bir sistemde 16 bayt kaplar: iki `int` için 8 bayt, işaretçi için 8 bayt. Terim dizisindeki 8 baytlık terime göre iki kat yer tutar; bağın bedeli budur. Buna karşılık düğümler bellekte ardışık olmak zorunda değildir ve ortak bir dizinin sınırı yoktur. Polinomun terim sayısı ancak sistemin belleği bittiğinde sınırlanır.

Listenin iki önemli kazancı vardır. Birincisi, araya ekleme ve silme yalnız birkaç işaretçinin değişmesiyle, *O(1)* zamanda yapılır; elbette yeri bulunduktan sonra. Dizide araya ekleme, sonraki bütün terimleri bir konum kaydırmayı gerektirirdi. İkincisi, toplama yine birleştirmedir: iki listenin başından aynı anda yürünür ve `padd`'deki üç durum aynen uygulanır. Bağlı listeyle toplama da *O(n + m)* zamanda çalışır. Tek fark, sonuç düğümlerinin `malloc` ile ayrılması ve iş bitince `free` ile geri verilmesidir.

5.5.2 YeniTerimEkle() ve SonrakiTerimiSil()

Çarpım kodu iki yardımcıya dayanır. Birincisi yeni bir terim düğümü oluşturur ve isteğe bağlı olarak verilen bir düğümün peşine bağlar:

```c
/* terim oluştur; Onceki varsa peşine ekle */
pol_gosterge YeniTerimEkle(int Us, int Katsayi,
                           pol_gosterge Onceki) {
  pol_gosterge p = malloc(sizeof *p);
  if (!p) { printf("Bellek yetmiyor!\n"); exit(1); }
  p->us = Us;  p->katsayi = Katsayi;
  if (Onceki) {              /* araya ya da sona */
    p->bag = Onceki->bag;    /* önce yeni düğüm */
    Onceki->bag = p;         /* sonra önceki */
  } else p->bag = NULL;      /* tek başına */
  return p;
}
```

`malloc(sizeof *p)`, `p`'nin gösterdiği türün, yani `pol_oge`'nin boyu kadar bellek ayırır. `sizeof(pol_oge)` yazmak da doğrudur; ama `sizeof *p` biçimi, `p`'nin türü değişirse kendiliğinden doğru kalır. `malloc` bellek bulamazsa `NULL` döndürür. Kod bu durumu hemen denetler ve programı bir iletiyle sonlandırır. Denetim yapılmasaydı bir sonraki satır `NULL` işaretçisinin gösterdiği yere yazmaya çalışırdı; bu tanımsız davranıştır ve çoğu sistemde program çöker.

Düğümün alanları doldurulduktan sonra iki durum ayrılır. `Onceki` verilmişse yeni düğüm onun peşine bağlanır. Sıra önemlidir: önce yeni düğümün bağı `Onceki`'nin eski sonrakine yöneltilir, sonra `Onceki`'nin bağı yeni düğüme. İki satır yer değiştirseydi `Onceki->bag` önce yeni düğüme yazılırdı; eski sonraki düğümün adresi kaybolur ve listenin geri kalanı erişilemez hâle gelirdi. `Onceki` `NULL` ise yeni düğüm tek başına bir liste olur; bağı `NULL`'dır. Fonksiyon her durumda yeni düğümün adresini döndürür.

İkinci yardımcı, verilen düğümden sonraki düğümü siler:

```
/* Onceki'den sonrakini sil */
void SonrakiTerimiSil(
        pol_gosterge Onceki) {
  pol_gosterge Silinecek =
        Onceki->bag;
  if (!Silinecek) return;
  Onceki->bag =
        Silinecek->bag;
  free(Silinecek);
}
```

Silinecek düğümün adresi önce `Silinecek` değişkeninde saklanır. Sonra `Onceki`'nin bağı, silinecek düğümün sonrakine yöneltilir; böylece düğüm listeden çıkar. En son düğümün belleği `free` ile geri verilir. Adres saklanmadan bağ güncellenseydi düğüme bir daha ulaşılamaz ve belleği hiçbir zaman geri verilemezdi. Böyle kaybolan belleğe **bellek sızıntısı** (*memory leak*) denir. Hocanın ifadesiyle, bellekteki kalıntı temizlenmelidir. Ters sıra da yanlıştır: önce `free(Onceki->bag)` çağrılıp sonra `Onceki->bag = Onceki->bag->bag` yazılırsa, serbest bırakılmış bir düğümün alanı okunur. Geri verilmiş belleği gösteren bu işaretçiye **sarkan işaretçi** (*dangling pointer*) denir ve onu kullanmak tanımsız davranıştır.

Fonksiyon, silinecek düğüm yoksa (`Onceki` son düğümse) hiçbir şey yapmadan döner. Fonksiyonun ilk düğümü silemediğine dikkat edilmelidir: silme her zaman bir önceki düğüm üzerinden yapılır. Çarpım kodu bu kısıtla yaşayabilir, çünkü çarpım listesinin ilk terimi hiçbir zaman silinmez.

5.5.3 PolCarp() ve yerlestir()

İki polinomun çarpımında A'nın her terimi B'nin her terimiyle çarpılır. Üretilen her çarpım, sıralı sonuç listesindeki yerine konur. Aynı üslü bir terim zaten varsa katsayı güncellenir; güncellenen katsayı 0 olursa terim silinir. İç döngünün bu gövdesi okunaklılık için ayrı bir fonksiyona alınmıştır:

```
pol_gosterge yerlestir(pol_gosterge r,
                       int us, int kat) {
  while (r->bag && r->bag->us > us) r = r->bag;
  if (!r->bag || r->bag->us < us)
    YeniTerimEkle(us, kat, r);        /* yeni üs */
  else if ((r->bag->katsayi += kat) == 0)
    SonrakiTerimiSil(r);              /* 0 oldu */
  return r;
}
```

`yerlestir`, `r` düğümünden başlayıp yeni terimin yerini arar. Varsayım şudur: `r`'nin üssü yeni terimin üssünden büyüktür, yani yeni terim `r`'den sonra bir yere gelecektir. `while` döngüsü, sonraki düğümün üssü yeni üsten büyük olduğu sürece ilerler. Döngü bittiğinde `r`, yeni terimden önce gelecek son düğümdür. Üç durum kalır:

- `r`'den sonra düğüm yoksa (`!r->bag`) ya da sonraki düğümün üssü yeni üsten küçükse, bu üs listede yoktur. `YeniTerimEkle(us, kat, r)` yeni düğümü `r`'nin peşine bağlar.
- Sonraki düğümün üssü yeni üsse eşitse katsayıya `kat` eklenir. `(r->bag->katsayi += kat) == 0` ifadesi önce toplamayı yapar, sonra sonucu 0 ile karşılaştırır.
- Toplam 0 olduysa `SonrakiTerimiSil(r)` bu düğümü siler.

Fonksiyon `r`'yi döndürür. Bu dönüş değeri bir sonraki aramanın başlangıcıdır. B'nin terimleri azalan sırada geldiği için aynı p ile üretilen çarpımlar da azalan sıradadır. Bir sonraki çarpım, listede öncekinin yerinden sonra gelir; aramayı listenin başından yeniden başlatmaya gerek yoktur.

Çarpım fonksiyonunun kendisi şöyledir:

```
pol_gosterge PolCarp(pol_gosterge pBas,
                     pol_gosterge qBas) {
  pol_gosterge p, q, r = NULL, bas = NULL, rBas=NULL;
  for (p = pBas; p; p = p->bag) {
    r = bas;              /* aramaya buradan başla */
    for (q = qBas; q; q = q->bag) {
      int us = p->us + q->us;
      int kat = p->katsayi * q->katsayi;
      if (rBas) r = yerlestir(r, us, kat);
      else rBas=r=bas=YeniTerimEkle(us, kat, NULL);
      if (q == qBas) bas = r;  /* sonraki p için */
    }
  }
  return rBas;
}
```

`pBas` ve `qBas`, iki polinomun listelerinin başıdır. `rBas`, sonuç listesinin başını tutar; fonksiyon sonunda onu döndürür. `r`, iç döngüde sonuç listesinde gezinen konumdur. `bas` ise bir sonraki p için aramanın başlayacağı düğümdür.

`bas` değişkeni bir gözleme dayanır. p ile B'nin ilk terimi `qBas`'ın çarpımı, p'nin üretebileceği en büyük üslü terimdir. Sıradaki p' terimi p'den küçük üslüdür. p"nin bütün çarpımları, p'nin `qBas` ile çarpımından küçük üslü olur ve listede onun önüne düşemez. Öyleyse p' için arama, $p \cdot qBas$ çarpımının yerleştiği yerden başlayabilir. Kodda bu, `q == qBas` iken dönen `r`'nin `bas`'a kaydedilmesiyle yapılır. Dış döngünün her turu `r = bas` ile başlar.

Sonuç listesi boşken (`rBas == NULL`) `yerlestir` çağrılamaz, çünkü `r` geçerli bir düğüm değildir. Bu durumda ilk çarpım `YeniTerimEkle(us, kat, NULL)` ile tek başına bir düğüm olur ve `rBas`, `r` ve `bas` bu düğüme yöneltilir. `rBas=r=bas=…` zincirleme atamasında atama işleci sağdan sola çalışır: önce `bas`, sonra `r`, en son `rBas` değeri alır.

Listenin ilk düğümü A'nın ilk terimiyle B'nin ilk teriminin çarpımıdır. Bu terimin üssü bütün çarpımların en büyüğüdür ve tek olarak elde edilir: başka hiçbir terim çifti aynı üssü veremez. Bu yüzden ilk düğüm hiçbir zaman güncellenmez ve silinmez; `SonrakiTerimiSil`'in ilk düğümü silememesi bir sorun yaratmaz. Aynı nedenle bir aramanın başlangıcı olan `r` düğümünün üssü her zaman yeni terimin üssünden büyüktür ve `yerlestir`'in varsayımı korunur.

5.5.4 Kod izleme: (x + 1)(x − 1)

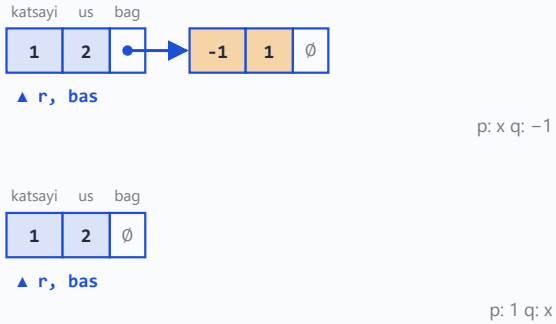
$A = x + 1$ ve $B = x - 1$ listeleri ikişer düğümlüdür. Düğümler (katsayı, üs) çifti olarak yazılır: $A = (1, 1) \to (1, 0)$, $B = (1, 1) \to (-1, 0)$. Beklenen sonuç $x^2 - 1$'dir. Tablo 5.4, dört çarpımın her birini izler. Tabloda `&(1, 2)`, katsayısı 1 ve üssü 2 olan düğümün adresidir.

Adım	p, q	`us`, `kat`	İşlem	Sonuç listesi	`r`, `bas`
1	x, x	2, 1	liste boş: `YeniTerimEkle(2, 1, NULL)`; `q == qBas` → `bas = r`	x^2	&(1, 2), &(1, 2)
2	x, -1	1, −1	`yerlestir`: `r`'den sonra düğüm yok → yeni düğüm $-x$	$x^2 - x$	&(1, 2), &(1, 2)
3	1, x	1, 1	`r = bas`; sonraki düğümün üssü 1 = 1 → katsayı $-1 + 1 = 0$ → `SonrakiTerimiSil(r)`; `bas = r`	x^2	&(1, 2), &(1, 2)
4	1, -1	0, −1	`yerlestir`: `r`'den sonra düğüm yok → yeni düğüm -1	$x^2 - 1$	&(1, 2), &(1, 2)

Tablo 5.4: `PolCarp(A, B)` izlemesi, $A = x + 1$, $B = x - 1$.

Birinci adımda liste boştur ve ilk düğüm doğrudan oluşturulur. İkinci adımda `yerlestir`, `r = &(1, 2)`'den başlar. Sonraki düğüm olmadığı için döngü hiç dönmez ve $-x$ düğümü x^2'nin peşine eklenir (Şekil 5.6a). Fonksiyon yeni düğümü değil, onun öncülünü döndürür; `r` yine `&(1, 2)`'dir.

Üçüncü adım çarpımın ilginç anıdır. Dış döngü $p = 1$'e geçer ve `r = bas` ile başlar. Bu örnekte `bas` listenin ilk düğümüdür; daha uzun polinomlarda arama listenin ortasından başlayabilir. Yeni çarpım $1 \cdot x = x$'tir. Sonraki düğüm $-x$'in üssü de 1'dir. Katsayılar toplanır: $-1 + 1 = 0$. Terim sıfırlandığı için `SonrakiTerimiSil(r)` onu listeden çıkarır ve belleğini geri verir (Şekil 5.6b). Dördüncü adımda $1 \cdot (-1) = -1$ sabit terimi listenin sonuna eklenir.



Şekil 5.6: `PolCarp` 'ın iki anı. (a) Adım 2: $-x$ düğümü x^2'nin peşine eklenmiştir. (b) Adım 3: $+x$ eklenince katsayı 0 olur ve düğüm silinir; liste yeniden tek düğümdür.

Sonuç $x^2 - 1$'dir. $n \cdot m = 4$ çarpım yapılmış, aynı üslü $-x$ ile $+x$ birleşip silinmiştir. Fonksiyon `rBas` 'ı döndürür. Program, sonuçla işi bitince listenin bütün düğümlerini tek tek `free` ile geri vermelidir; girdiler için de aynısı geçerlidir. Test sürücüsünde bunu yapan kısa bir döngü vardır: sıradaki düğümün adresi saklanır, şu anki düğüm serbest bırakılır ve saklanan adrese geçilir.

Örnek 5.3

$(x^2 + 1)(x^2 - 1)$ çarpımında hangi terim silinir?

1. $p = x^2$: $x^2 \cdot x^2 = x^4$ ilk düğüm olur. $x^2 \cdot (-1) = -x^2$ onun peşine eklenir. Liste: $x^4 - x^2$.
2. $p = 1$: arama `bas` $= x^4$ düğümünden başlar. $1 \cdot x^2 = +x^2$; aynı üslü düğüm vardır ve katsayı $-1 + 1 = 0$ olur. x^2 terimi silinir.
3. $1 \cdot (-1) = -1$ sona eklenir.

Sonuç $x^4 - 1$'dir. Program da aynı listeyi üretir.

5.5.5 Karmaşıklık ve ölçüm

A'da n, B'de m terim olsun. Çarpım sayısı tam nm'dir. Her çarpımın kendi işi sabittir (bir toplama, bir çarpma, en çok bir `malloc` ya da `free`). Asıl maliyet `yerlestir` 'in `while` döngüsündeki yürüyüştür.

Bir p terimi için iç döngü, `bas` 'tan başlayıp listede yalnız ileri gider. `yerlestir` her çağrıda kaldığı yerden sürer, geri dönmez. Bu yüzden bir p için toplam yürüyüş, listenin `bas` 'tan sonraki uzunluğunu geçemez. Sonuç listesi en çok nm düğümlüdür. Bir p'nin maliyeti $O(m + nm)$, bütün p'lerin maliyeti $O(n^2 m)$'dir. Bu sınır `pmult` 'unkiyle aynıdır, ama bağlı listede ara sonuçlar için ek bellek harcanmaz: sonuç listesi en çok nm düğümlüdür ve sıfırlanan düğümler hemen geri verilir.

Sınırın gerçekten erişilip erişilmediği terimlerin üslerine bağlıdır. Bunu görmek için `yerlestir` 'in döngüsüne bir sayaç eklenmiş ve $n = m$ terimli iki polinom iki farklı dizilimle çarpılmıştır. İki durumda da sonuç nm farklı üslü terimden oluşur:

- **Seyrek A, sık B.** A'nın üsleri $(n-1)m, \ldots, m, 0$; B'ninkiler $m-1, \ldots, 0$. Her p'nin çarpımları bir blok oluşturur ve bir sonraki p'nin bloğu tamamen onun arkasına gelir.
- **Sık A, seyrek B.** A'nın üsleri $n-1, \ldots, 0$; B'ninkiler $(m-1)n, \ldots, n, 0$. Her p'nin çarpımları, öncekilerin arasına serpiştirilir.

$n = m$	Seyrek A, sık B	Sık A, seyrek B	$n^2 m / 2$
10	178	502	500
20	758	4.007	4.000
40	3.118	32.017	32.000
80	12.638	256.037	256.000
160	50.878	2.048.077	2.048.000

Tablo 5.5: `yerlestir` döngüsünün toplam adım sayısı ($n = m$, sonuçta nm terim).

İlk dizilimde adım sayısı n iki katına çıkınca dört katına çıkar; yaklaşık $2nm$'dir, yani $O(nm)$. `bas` burada işini yapar: her p araması bir önceki bloğun başından başlar ve listede yalnız bir kez gezinilir. İkinci dizilimde adım sayısı sekiz katına çıkar ve $n^2m/2$'ye neredeyse eşittir. Her p, eklediği terimler önceki terimlerin arasına dağıldığı için listenin büyük kısmında yeniden gezinir. Bu, $O(n^2m)$ sınırının gerçekten erişilebildiğini gösterir. En kötü durum maliyeti $\Theta(n^2m)$'dir.

Ölçümden çıkan pratik ders şudur: n ile m farklıysa küçük polinom dış döngüye alınmalıdır. Dış döngüdeki polinomun terim sayısının karesi maliyete girdiği için 2 terimli bir polinomu dışa almak, 100 terimliyi dışa almaktan yaklaşık elli kat ucuzdur.

Kodun doğruluğu da sınanmıştır. Rastgele üretilen 20.000 polinom çiftinin `PolCarp` sonucu, katsayıları bir dizide toplayan kaba kuvvet çarpımıyla karşılaştırılmıştır. Her sonuç listesinin azalan üslü, sıfır katsayısız ve kaba kuvvet sonucuyla aynı olduğu görülmüştür.

> **Sık yapılan hata**
>
> Katsayısı 0'a inen terimi listede bırakmak. $(x + 1)(x - 1)$ çarpımı $x^2 + 0x - 1$ listesi olarak kalırsa liste fazladan bir düğüm taşır, derece ve terim sayısı yanlış görünür ve sonraki işlemler bu düğüme gereksiz yere uğrar. Sıfırlanan düğüm hem listeden çıkarılmalı hem de `free` ile geri verilmelidir; yalnız bağı değiştirmek bellek sızıntısına yol açar.

5.5.6 Problemin öteki çözümleri

İki polinomu çarpmanın tek yolu `PolCarp` değildir. Tablo 5.6, sunumda listelenen yöntemleri karmaşıklıklarıyla verir. A'da n, B'de m terim vardır.

	Yöntem	Karmaşıklık
1	Hepsi çarpılır, her çarpım sona eklenir; sonra sıralanır ve birleştirilir	$O(n^2m^2)$
2	Hepsi çarpılır; her çarpım için sonuç baştan taranıp uygun konuma eklenir	$O(n^2m^2)$
3	A'nın her terimiyle çarpım için bir liste kurulur, her adımda sonuçla birleştirilir	$O(n^2m)$
4	3. yöntem, küçük polinom dışta	$O(\min(n, m)^2 \cdot \max(n, m))$
5	A'nın her terimi için ayrı liste; en sonda hepsi ikişer ikişer birleştirilir	$O(m \cdot n \cdot \lg n)$
6	5. yöntem, küçük polinom dışta	$O(m \cdot n \cdot \lg \min(m, n))$
7	Bir dizi açılır, k. indis k. üssün katsayısını tutar	—

Tablo 5.6: Polinom çarpımının farklı çözümleri.

Birinci yöntem nm çarpımı sırasız bir listeye yazar, sonra listeyi sıralar ve komşu aynı üsleri birleştirir. Sıralama basit bir karesel algoritmayla, örneğin eklemeli sıralamayla yapılırsa $(nm)^2 = n^2m^2$ karşılaştırma gerekir. İkinci yöntem her çarpım için sonucu baştan tarar; sonuç nm terime kadar büyüyebildiği için yine $O(n^2m^2)$ eder. Üçüncü yöntem `pmult`'un liste sürümüdür; `PolCarp` da `bas` sayesinde bu sınıra iner.

Beşinci yöntem birleştirmeli sıralamanın (*merge sort*) birleştirme aşamasına benzer. n liste, her biri m terimli olarak üretilir. Sonra ikişer ikişer birleştirilir: n liste $n/2$'ye, $n/2$ liste $n/4$'e iner. Her düzeyde toplam en çok nm terim işlenir ve $\lceil \lg n \rceil$ düzey vardır. Toplam maliyet $O(nm \lg n)$'dir. Küçük polinomun terimleriyle liste kurmak, logaritmanın içine küçük sayıyı koyar.

Yedinci yöntem sonucu katsayı dizisinde biriktirir: her çarpımın katsayısı `d[us]`'ye eklenir. Çarpımın kendisi $O(nm)$ sürer ve son derece basittir. Ama dizinin boyu üslerin büyüklüğüne bağlıdır. $x^{1000000}$ içeren iki seyrek polinom için iki milyon hücreli bir dizi gerekir. Sonuç dizisini terim listesine çevirmek de bütün diziyi taramayı ister. Dersin hocası bu fikre kısa ve sert bir cevap verir: n. indisin n. üssü tuttuğu bir dizi asla. Seyrek veri için tasarlanan bir yapının içine seyrek olmayan bir dizi koymak, yapının bütün kazancını geri verir. Bu notun test programı yedinci yöntemi yalnız doğruluk denetimi için, küçük üslerle kullanmıştır.

5.6 Seyrek matris

Polinomdaki sorun, iki boyutlu yapılarda daha büyük ölçekte karşımıza çıkar. Bir şehrin yol ağını tutan komşuluk matrisi, bir sosyal ağdaki arkadaşlık ilişkileri ya da bir mühendislik benzetimindeki denklem sistemi, binlerce satır ve sütunlu matrislerdir. Bu matrislerin öğelerinin büyük çoğunluğu sıfırdır: her kavşak birkaç kavşağa bağlanır, her kişinin arkadaş sayısı toplam nüfusun yanında küçüktür.

5.6.1 Tanım

Tanım 5.2

Seyrek matris (*sparse matrix*), öğelerinin çoğu sıfır olan matristir. Sıfırdan farklı öğelere **dolu öğe** denir. Bir matrisin seyrek sayılması için kesin bir eşik yoktur. Pratikte dolu öğe sayısı t, toplam öğe sayısı $rows \cdot cols$'un küçük bir kesriyse matris seyrek kabul edilir.

Şekil 5.7'deki 6×6 matrisin 36 hücresinden yalnız 8'i doludur. Bu notun seyrek matris örneklerinin çoğu bu matristen türetilir. Matris C'de iki boyutlu bir diziyle, `int a[6][6]` olarak tutulabilir. Bu dizi, 8 anlamlı sayı için 36 hücre ayırır. 1000×1000 boyutlu ve her satırında yalnız beş dolu öğe bulunan bir matris, dört baytlık tam sayılarla 4 MB tutar; dolu öğeler ise toplam 5.000 tanedir.

	0	1	2	3	4	5
0	**15**	**0**	**0**	**22**	**0**	**-15**
1	**0**	**11**	**3**	**0**	**0**	**0**
2	**0**	**0**	**0**	**-6**	**0**	**0**
3	**0**	**0**	**0**	**0**	**0**	**0**
4	**91**	**0**	**0**	**0**	**0**	**0**
5	**0**	**0**	**28**	**0**	**0**	**0**

Şekil 5.7: 6×6 seyrek matris. 36 hücreden yalnız 8'i doludur (koyu hücreler).

Bellek israfı tek sorun değildir. İki boyutlu dizide transpoz, toplama ve çarpma gibi işlemler bütün hücrelerde gezinir. 1000×1000 matrisin transpozu bir milyon hücre kopyalar; oysa yalnız 5.000 öğenin yeri değişir. Seyrek matris gösterimlerinin amacı, hem belleği hem de işlem süresini dolu öğe sayısına bağlamaktır.

5.6.2 Üçlü gösterim

Her dolu öğe üç bilgiyle tam olarak belirlenir: satırı, sütunu ve değeri. Bu üç bilgiyi bir yapıda toplamak, terim dizisindeki ⟨üs, katsayı⟩ çiftinin iki boyutlu karşılığıdır.

Tanım 5.3

Bir **üçlü** (*triple*), seyrek matristeki sıfırdan farklı bir öğeyi tutan ⟨satır, sütun, değer⟩ kaydıdır. **Üçlü gösterimde** matris, üçlülerden oluşan bir dizidir. Dizinin 0 numaralı öğesi başlıktır: satır sayısını, sütun sayısını ve dolu öğe sayısını tutar. Üçlüler önce satıra, aynı satırdakiler sütuna göre artan sırada dizilir.

```c
#define MAX_TERMS 101
#define MAX_COL 50   /* azami sütun */
typedef struct {
  int row, col, value;   /* satır, sütun, değer */
} term;
term M[MAX_TERMS];
```

`term` yapısının üç `int` alanı vardır ve yapı 12 bayt kaplar. `M[0]` başlıktır: `M[0].row` satır sayısını, `M[0].col` sütun sayısını, `M[0].value` dolu öğe sayısını tutar. Dolu öğeler `M[1]`'den `M[M[0].value]`'ya kadar dizilir. `MAX_COL`, transpoz fonksiyonlarındaki yardımcı dizilerin boyudur; matrisin sütun sayısı bu değeri aşamaz. Başlığın da aynı türden bir üçlü olarak tutulması küçük bir hiledir: ayrı bir başlık yapısı gerekmez ve matris tek bir diziyle fonksiyonlara geçirilir.

	row	col	value
M[0]	6	6	8
M[1]	0	0	15
M[2]	0	3	22
M[3]	0	5	-15
M[4]	1	1	11
M[5]	1	2	3
M[6]	2	3	-6
M[7]	4	0	91
M[8]	5	2	28

Şekil 5.8: Şekil 5.7'deki matrisin üçlü tablosu. `M[0] = (6, 6, 8)` başlıktır; sekiz üçlü satır, sonra sütun sırasıyla dizilmiştir.

Sıra koşulu, satır öncelikli düzenin (*row-major order*) üçlülere uygulanmasıdır. İki boyutlu dizi bellekte satır satır durur; üçlü tablo da dolu öğeleri aynı sırayla okur. Şekil 5.8'de 0. satırın üç öğesi (sütun 0, 3, 5) önce gelir, sonra 1. satırın iki öğesi (sütun 1, 2) gelir. 3. satırda dolu öğe olmadığı için tabloda 3 satır numaralı üçlü yoktur. Bu sıra sayesinde bir satırın öğeleri tabloda ardışık durur ve bir satırı okumak, tablonun bir aralığını okumaktır.

Horowitz'in kitabı seyrek matris soyut veri tipini dört işlemle tanımlar: `Create(max_row, max_col)` boş bir matris kurar, `Transpose(a)` transpozu verir, `Add(a, b)` ve `Multiply(a, b)` toplamı ve çarpımı verir. Üçlü gösterim bu işlemlerin yalnız biridir; bağlı liste ve sıkıştırılmış satır gösterimi gibi başka gerçeklemeler de vardır.

5.6.3 Üçlü gösterim ne zaman kazandırır?

İki boyutlu dizi *rows · cols* tam sayı tutar. Üçlü gösterim, başlık dahil *t + 1* üçlü, yani *3(t + 1)* tam sayı tutar. Üçlü gösterimin daha az yer kaplaması için

> 3(t + 1) < rows · cols, yani t < rows · cols / 3 − 1

olmalıdır. Kabaca, matrisin üçte birinden azı doluysa üçlü gösterim kazandırır. Üçte biri dolu olduğunda iki gösterimin bellek maliyeti yaklaşık eşittir; daha doluysa üçlü gösterim daha çok yer tutar ve tercih edilmez.

Örnek 5.4

Şekil 5.7'deki matris için: *t = 8*; iki boyutlu dizi 36, üçlü tablo *3 · 9 = 27* tam sayı tutar. Üçlü gösterim kazançlıdır. Eşik *3(t + 1) < 36*, yani *t ≤ 10*'dur; *t = 11*'de iki gösterim 36'şar tam sayıyla eşitlenir.

5 × 5 bir matriste dizi 25 tam sayı tutar. *3(t + 1) > 25* koşulu *t ≥ 8*'de sağlanır: *t = 7*'de üçlüler 24, *t = 8*'de 27 tam sayı tutar. Sekiz ve daha çok dolu öğede üçlü gösterim daha çok yer kaplar.

Bu hesap yalnız belleği karşılaştırır. Zaman açısından tablo farklıdır: üçlü gösterimde bir öğeye `a[i][j]` gibi doğrudan ulaşılamaz. *(i, j)* öğesini okumak için tablo aranmalıdır. Sıralılık sayesinde ikili arama *O(log t)*'de bulur, ama yine de *O(1)* değildir. Seyrek gösterim, tek tek öğe okumaktan çok bütün dolu öğelerde gezinen işlemler için uygundur. Transpoz ve çarpım böyle işlemlerdir.

Sık yapılan hata

`M[0]`'ı bir dolu öğe sanmak. Döngüler `j = 1`'den başlayıp `j <= M[0].value` koşuluyla biter. `j = 0`'dan başlayan bir döngü başlığı da işler: 6 × 6 matrisin başlığı (6, 6, 8), "6. satır, 6. sütundaki değer 8" diye okunur. Böyle bir hücre matriste yoktur ve dizinin sınırları dışına yazmaya yol açabilir.

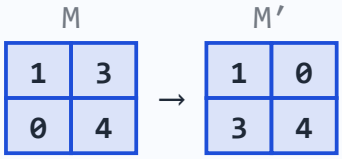
5.7 Transpoz alma

Transpoz, matris işlemlerinin en basitlerinden biridir ve seyrek gösterimin hem gücünü hem tuzağını iyi gösterir.

5.7.1 Transpoz ve sıra sorunu

Tanım 5.4

Bir *M* matrisinin **transpozu** (*transpose*) *M′*, satırları *M*'nin sütunları olan matristir: her *i, j* için *M′[j][i] = M[i][j]*. *rows × cols* boyutlu bir matrisin transpozu *cols × rows* boyutludur.



Şekil 5.9: *2 × 2* bir matris ve transpozu. Birinci satır (1, 3), transpozda birinci sütun olur.

İki boyutlu dizide transpoz iki iç içe döngüdür: `t[j][i] = m[i][j]`. Üçlü gösterimde ilk akla gelen, her üçlünün `row` ve `col` alanlarını yer değiştirmektir. Bu, öğelerin doğru konumunu verir, ama sırayı bozar. Şekil 5.7'deki matriste `M[1] = (0, 0, 15)`, `M[2] = (0, 3, 22)`, `M[3] = (0, 5, -15)` 'tir. Alanlar değiştirilince (0, 0, 15), (3, 0, 22), (5, 0, −15) elde edilir. Bunların ardından gelen `M[4] = (1, 1, 11)`, (1, 1, 11) olur. Sonuç tablosu 0, 3, 5, 1, … diye ilerler ve artık satır sırasında değildir. Satır sırasına güvenen bütün algoritmalar, örneğin çarpım ve ikili arama, bu tabloyla yanlış çalışır.

Sırayı düzeltmenin iki kaba yolu vardır. Alanlar değiştirilir ve tablo sonradan sıralanır; ya da her üçlü, sonuç tablosunda doğru yere araya eklenir ve sonraki üçlüler bir konum kaydırılır. İkisi de pahalıdır. Dersin hocası bunu şöyle söyler: kaydırma, taşıma kullanılıyorsa veri yapısının hakkı verilmiyor demektir. Daha iyi bir çözüm, transpozun tanımından çıkar. Transpozun satırları *M*'nin sütunlarıdır. Öyleyse *M* sütun sütun taranırsa, transpozun satırları sırayla üretilir.

5.7.2 transpose(): sütun sütun tarama

```
void transpose(term M[], term T[]) {
  int n = M[0].value, i, j, cur;
  T[0].row = M[0].col;  T[0].col = M[0].row;
  T[0].value = n;
  if (n > 0) {
    cur = 1;                        /* T'de sıradaki */
    for (i = 0; i < M[0].col; i++)     /* her sütun */
      for (j = 1; j <= n; j++)         /* her terim */
        if (M[j].col == i) {
          T[cur].row = M[j].col; T[cur].col = M[j].row;
          T[cur].value = M[j].value;  cur++;
        }
  }
}
```

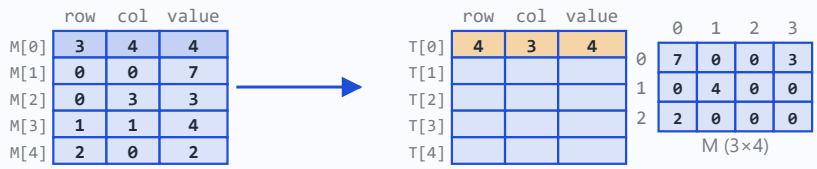
Fonksiyon önce başlığı yazar: transpozun satır sayısı *M*'nin sütun sayısı, sütun sayısı *M*'nin satır sayısıdır. Dolu öğe sayısı değişmez. Matris boş değilse iç içe iki döngü çalışır. Dış döngü *M*'nin her sütunu `i` için bir tur döner. İç döngü *M*'nin bütün üçlülerini tarar ve sütunu `i` olanları `T` 'ye yazar. `cur`, `T` 'de sıradaki boş konumdur.

Sonucun neden sıralı çıktığı iki gözleme dayanır. Birincisi, dış döngü sütunlarda artan sırada gezindiği için `T` 'ye önce *M*'nin 0. sütununun öğeleri, sonra 1. sütununun öğeleri yazılır. Bunlar `T` 'nin 0. ve 1. satırlarıdır; satır sırası sağlanır. İkincisi, aynı sütundaki öğeler *M*'de satır sırasıyla durduğu için iç döngü onları satır sırasıyla bulur. `T` 'de bu satır numaraları sütun numarası olur; aynı satır içinde sütun sırası da sağlanır.

İç döngünün `j = 1` 'den başladığına dikkat edilmelidir. `j = 0` 'dan başlasaydı başlık `M[0]` da taranırdı. `M[0].col` sütun sayısıdır ve hiçbir `i` değerine eşit olmaz, çünkü `i` en çok sütun sayısının bir eksiğine çıkar. Sonuç bu yüzden bozulmazdı, ama her sütun için gereksiz bir karşılaştırma yapılırdı. Doğrusu `j = 1` 'dir.

5.7.3 Kod izleme ve analiz

İzleme için küçük bir matris kullanılır: *3 × 4* boyutlu ve dört dolu öğeli *M*. Dolu öğeler `M[1] = (0, 0, 7)`, `M[2] = (0, 3, 3)`, `M[3] = (1, 1, 4)` ve `M[4] = (2, 0, 2)`'dir. Üçlülerin sütunları sırasıyla 0, 3, 1, 0'dır.

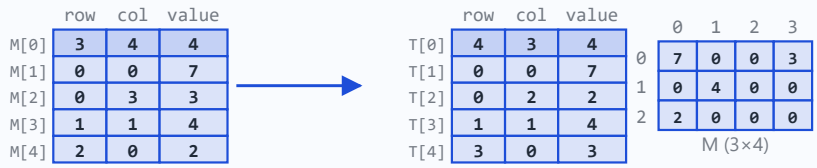


Şekil 5.10: `transpose`'un başlangıcı. Solda *M*'nin üçlüleri, ortada başlığı yazılmış `T`, sağda *M*'nin iki boyutlu görünümü.

`i`	`M[1..4].col` ile karşılaştırma	Eşleşen `j`	Yazılan	`cur` (sonra)
—	başlık	—	`T[0] = (4, 3, 4)`	1
0	0 = 0, 3 ≠ 0, 1 ≠ 0, 0 = 0	1, 4	`T[1] = (0, 0, 7)`, `T[2] = (0, 2, 2)`	3
1	0 ≠ 1, 3 ≠ 1, 1 = 1, 0 ≠ 1	3	`T[3] = (1, 1, 4)`	4
2	0 ≠ 2, 3 ≠ 2, 1 ≠ 2, 0 ≠ 2	—	—	4
3	0 ≠ 3, 3 = 3, 1 ≠ 3, 0 ≠ 3	2	`T[4] = (3, 0, 3)`	5

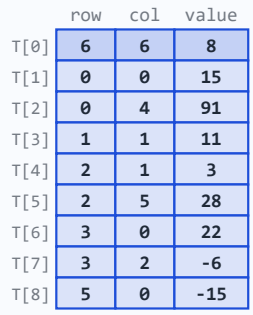
Tablo 5.7: `transpose(M, T)` izlemesi, sütun başına özet.

Dış döngü dört tur, iç döngü her turda dört adım döner: *4 · 4 = 16* karşılaştırma yapılır. Bunların yalnız dördü eşleşir; on ikisi boşa gider. 2. sütunda hiç dolu öğe yoktur, ama yine de bütün üçlüler taranır. Sonuç `T` satır sırasına göre kendiliğinden sıralıdır: (0, 0, 7), (0, 2, 2), (1, 1, 4), (3, 0, 3).



Şekil 5.11: `transpose` bittiğinde `T`. Transpoz *4 × 3* boyutludur; üçlüler satır, sonra sütun sırasındadır.

Fonksiyon Şekil 5.7'deki *6 × 6* matrise de uygulanmıştır. Sonuç tablosu (0, 0, 15), (0, 4, 91), (1, 1, 11), (2, 1, 3), (2, 5, 28), (3, 0, 22), (3, 2, −6), (5, 0, −15)'tir (Şekil 5.12). Burada *6 · 8 = 48* karşılaştırma yapılır.



Şekil 5.12: *6 × 6* örnek matrisin transpozu `T`. *M*'deki 91 (4. satır, 0. sütun) transpozda 0. satır, 4. sütundadır.

Karmaşıklık iç içe döngülerden okunur. Dış döngü *cols*, iç döngü *terms* kez döner; iç gövde sabit iş yapar. Zaman *O(cols · terms)*'dir. Bu maliyet, matrisin ne kadar seyrek olduğuna bağlıdır. Matris doluysa *terms = rows · cols* olur ve maliyet $O(cols^2 \cdot rows)$'a çıkar. İki boyutlu dizide transpoz *O(rows · cols)* sürer. Dolu matriste `transpose` bir *cols* çarpanı kadar daha yavaştır.

Yani üçlü gösterim bellekten kazandırırken zamandan kaybettirmiştir. Bu takas ancak *terms* ≪ *rows* · *cols* olduğunda anlamlıdır. Sorunun kaynağı açıktır: aynı *M* dizisi her sütun için baştan sona yeniden taranır. Her terimin `T`'de nereye gideceği önceden bilinseydi, *M* bir kez taranır ve her terim doğrudan yerine konurdu. Örneğin transpozun 0. satırında 2, 1. satırında 3 öğe olacağı bilinsin. 0. satır `T[1]`'den, 1. satır `T[3]`'ten, 2. satır `T[6]`'dan başlar. Hızlı transpoz bu bilgiyi önceden hesaplar.

5.7.4 fast_transpose(): önce say, sonra yerleştir

Tanım 5.5

Hızlı transpoz (*fast transpose*), seyrek matrisin transpozunu iki geçişte alan yöntemdir. Birinci geçiş her sütundaki dolu öğe sayısını sayar ve transpozun her satırının tabloda başlayacağı konumu hesaplar. İkinci geçiş her üçlüyü doğrudan bu konuma yazar.

```
void fast_transpose(term M[], term T[]) {
  int i, j, nc = M[0].col, nt = M[0].value;
  int row_terms[MAX_COL], start_pos[MAX_COL];
  T[0].row = nc;  T[0].col = M[0].row;  T[0].value = nt;
  if (nt <= 0) return;                    /* boş matris */
  for (i = 0; i < nc; i++) row_terms[i] = 0;
  for (i = 1; i <= nt; i++) row_terms[M[i].col]++;
  start_pos[0] = 1;
  for (i = 1; i < nc; i++)
    start_pos[i] = start_pos[i-1] + row_terms[i-1];
  for (i = 1; i <= nt; i++) {
    j = start_pos[M[i].col]++;       /* yerini al, artır */
    T[j].row = M[i].col;  T[j].col = M[i].row;
    T[j].value = M[i].value;
  }
}
```

Fonksiyon iki yardımcı dizi kullanır. `row_terms[i]`, transpozun `i`. satırındaki, yani *M*'nin `i`. sütunundaki öğe sayısıdır. `start_pos[i]`, transpozun `i`. satırının `T`'deki ilk konumudur. Kodun dört döngüsü vardır:

1. `row_terms` **sıfırlanır.** C'de yerel diziler ilk değer almaz; içlerindeki değer belirsizdir. Sıfırlama atlanırsa sayaçlar rastgele değerlerden başlar.
2. **Sayma.** *M*'nin her üçlüsü için kendi sütununun sayacı bir artar: `row_terms[M[i].col]++`.
3. **Başlangıç konumları.** 0. satır `T[1]`'den başlar. Her satır, bir önceki satırın başlangıcına önceki satırın öğe sayısı eklenerek bulunur: `start_pos[i] = start_pos[i-1] + row_terms[i-1]`. Bu bir **önek toplamıdır** (*prefix sum*): `start_pos[i]`, 1 ile 0'dan *i* − *1*'e kadar bütün sayaçların toplamıdır.
4. **Yerleştirme.** *M* baştan sona bir kez taranır. Her üçlü, kendi sütununun `start_pos` değerinin gösterdiği yere yazılır ve bu değer bir artar. Aynı sütundan gelen sonraki üçlü bir sonraki konuma düşer.

Dördüncü döngünün `j = start_pos[M[i].col]++;` satırı yoğun bir ifadedir. Önce `M[i].col` okunur; bu, dizinin indisidir. Son ek `++` işleci önce `start_pos[M[i].col]`'un eski değerini `j`'ye verir, sonra diziyi bir artırır. Ön ek `++start_pos[M[i].col]` yazılsaydı önce artırma yapılır ve `j` bir fazla olurdu: ilk üçlü `T[2]`'ye yazılır, son konuma ait üçlü dizinin dışına taşardı. Hocanın uyarısı bu satır içindir: programcı, program yazdığı dilin işleç önceliklerini iyi bilmelidir. Burada köşeli parantez ve `.` işleçleri, son ek `++`'dan önce uygulanır; artırılan, `M[i].col` değil, `start_pos` dizisinin o hücresidir.

Üçlülerin sırası neden korunur? *M* satır sırasıyla tarandığı için aynı sütundaki öğeler satır sırasıyla gelir. Her biri kendi satırının bir sonraki boş konumuna yazılır. Transpozun her satırında öğeler *M*'deki satır numaralarına, yani transpozdaki sütun numaralarına göre artan sıradadır. Satırların kendi sırası da `start_pos` değerlerinin artan olmasından gelir.

5.7.5 Kod izleme: iki aşama

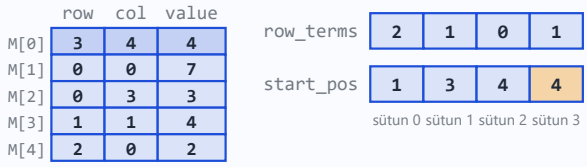
Aynı *3 × 4* matris kullanılır. Birinci aşama sayar ve başlangıç konumlarını bulur (Tablo 5.8).

Adım	İşlem	`row_terms`	`start_pos`
1	başlık: `T[0] = (4, 3, 4)`; sayaçlar sıfırlanır	[0, 0, 0, 0]	—
2	`M[1]` sütun 0'da	[1, 0, 0, 0]	—

Adım	İşlem	`row_terms`	`start_pos`
3	`M[2]` sütun 3'te	[1, 0, 0, 1]	—
4	`M[3]` sütun 1'de	[1, 1, 0, 1]	—
5	`M[4]` sütun 0'da	[2, 1, 0, 1]	—
6	`start_pos[0] = 1`	[2, 1, 0, 1]	[1, ·, ·, ·]
7	`start_pos[1] = 1 + 2 = 3`	[2, 1, 0, 1]	[1, 3, ·, ·]
8	`start_pos[2] = 3 + 1 = 4`	[2, 1, 0, 1]	[1, 3, 4, ·]
9	`start_pos[3] = 4 + 0 = 4`	[2, 1, 0, 1]	[1, 3, 4, 4]

Tablo 5.8: `fast_transpose` birinci aşama: sayma ve önek toplamı.

Son adımdaki eşitlik dikkat çeker: `start_pos[2]` ile `start_pos[3]` ikisi de 4'tür. 2. sütunda hiç öğe olmadığı için transpozun 2. satırı boştur ve 3. satır, 2. satırın başlayacağı yerden başlar. Boş bir satır, ardışık iki eşit başlangıç konumu olarak görünür.



Şekil 5.13: Birinci aşamanın sonu: `row_terms = [2, 1, 0, 1]`, `start_pos = [1, 3, 4, 4]`.

İkinci aşama *M*'yi bir kez tarar (Tablo 5.9).

`i`	`M[i]`	`j = start_pos[M[i].col]++`	Yazılan	`start_pos` (sonra)
1	(0, 0, 7)	`start_pos[0]` : `j = 1`	`T[1] = (0, 0, 7)`	[2, 3, 4, 4]
2	(0, 3, 3)	`start_pos[3]` : `j = 4`	`T[4] = (3, 0, 3)`	[2, 3, 4, 5]
3	(1, 1, 4)	`start_pos[1]` : `j = 3`	`T[3] = (1, 1, 4)`	[2, 4, 4, 5]
4	(2, 0, 2)	`start_pos[0]` : `j = 2`	`T[2] = (0, 2, 2)`	[3, 4, 4, 5]

Tablo 5.9: `fast_transpose` ikinci aşama: yerleştirme.

Üçlüler `T` 'ye sırasız yazılır: önce `T[1]`, sonra `T[4]`, `T[3]` ve `T[2]`. Ama her biri doğru yere gider ve sonunda `T`, `transpose` 'un ürettiği tabloyla aynıdır. Aşama bittiğinde `start_pos[i]`, `i`. satırın son öğesinden sonraki konumu gösterir; yani `start_pos[i]` artık `i + 1`. satırın başıdır. 2. sütun boş olduğu için `start_pos[2]` hiç değişmemiştir.



Şekil 5.14: İkinci aşamanın ikinci adımı: `M[2] = (0, 3, 3)` doğrudan `T[4]` 'e yazılmıştır; `T[2]` ve `T[3]` henüz boştur.

6 × 6 örnek matriste birinci aşama `row_terms = [2, 1, 2, 2, 0, 1]` ve `start_pos = [1, 3, 4, 6, 8, 8]` üretir. 4. sütun boş olduğu için `start_pos[4]` ile `start_pos[5]` eşittir. Bu değerler programla doğrulanmıştır.

5.7.6 Analiz, tek dizili sürüm ve ölçüm

`fast_transpose` 'un dört döngüsü sırasıyla *cols*, *terms*, *cols − 1* ve *terms* tur döner. Hiçbiri iç içe değildir. Döngü dışındaki sabit sayıda işlemle birlikte toplam iş

> 2 · terms + 2 · cols + sabit = O(cols + terms)

olur. Dolu matriste *terms = rows · cols* olduğundan maliyet *O(rows · cols)*'tur: iki boyutlu dizideki transpozla aynı derece. Seyrek matriste maliyet dolu öğe sayısıyla orantılıdır. Bedeli, *cols* boyunda iki yardımcı dizidir: *O(cols)* ek bellek.

Teorem 5.2 (İki transpozun maliyeti)

rows × cols boyutlu ve *t* dolu öğeli bir seyrek matris için `transpose` *θ(cols · t)*, `fast_transpose` *θ(cols + t)* zamanda çalışır. İkisi de sırası doğru bir transpoz üretir.

Hızlı transpoz tek bir yardımcı diziyle de yazılabilir. Hocanın sorusu bunu hatırlatır: hızlı transpoz tek bellekle olur denmişti ya, işte bu o. Sayma bittikten sonra sayaçlara bir daha gerek yoktur; aynı dizi yerinde başlangıç konumlarına çevrilir:

```
/* row_terms'ü yerinde start_pos'a çevir */
for (i = 0, pos = 1; i < nc; i++) {
  int c = row_terms[i];  /* bu sütundaki terim sayısı */
  row_terms[i] = pos;  /* artık başlangıç konumu */
  pos += c;
}
for (i = 1; i <= nt; i++) {
  j = row_terms[M[i].col]++;
  T[j].row = M[i].col;  T[j].col = M[i].row;
  T[j].value = M[i].value;
}
```

Bu kodda `row_terms`, önceki sürümdeki gibi sayılmış dizidir. İlk döngü her sayacı okur, yerine o satırın başlangıç konumunu yazar ve `pos`'u sayaç kadar ilerletir. Sayacın eski değeri `c` yerel değişkeninde saklanmazsa, üzerine yazılan değer okunur ve `pos` yanlış artar. İkinci döngü önceki sürümün yerleştirme döngüsüyle aynıdır. Bu biçim, bu notun test programında tam bir fonksiyon olarak derlenmiş ve iki yardımcı dizili sürümle aynı sonucu verdiği görülmüştür.

Tek dizili sürüm, **saymalı sıralamanın** (*counting sort*) ta kendisidir. Saymalı sıralama, küçük bir aralıktan anahtar alan öğeleri sıralar: önce her anahtardan kaç tane olduğunu sayar, önek toplamıyla her anahtarın başlangıç konumunu bulur, sonra öğeleri sırayla yerlerine koyar. Hızlı transpozda anahtar sütun numarasıdır. Aynı anahtarlı öğelerin girdideki göreli sırası korunduğu için sıralama kararlıdır ve satır sırası bozulmaz.

Aradaki fark büyük matrislerde çarpıcıdır. Tablo 5.10, sunumdaki iki fonksiyonun yalnız `MAX_TERMS` ve `MAX_COL` sabitleri büyütülerek `-O2` ile derlenmiş sürümlerinin ölçümünü verir. Her satırda on dolu öğe vardır. Süreler makineye göre değişir; önemli olan büyüme oranıdır.

Boyut	Dolu öğe	`transpose`	`fast_transpose`
1000 × 1000	10.000	≈ 4	≈ 0,04
2000 × 2000	20.000	≈ 14	≈ 0,08
4000 × 4000	40.000	≈ 60	≈ 0,26
8000 × 8000	80.000	≈ 254	≈ 0,36

Tablo 5.10: Transpoz süreleri (milisaniye; makineye göre değişir).

Boyut iki katına çıkınca hem sütun sayısı hem dolu öğe sayısı iki katına çıkar. `transpose`'un süresi *cols · t* ile orantılı olduğu için yaklaşık dört katına çıkar: 4, 14, 60, 254. `fast_transpose`'un süresi *cols + t* ile orantılıdır ve yaklaşık iki katına çıkar. En büyük matriste iki yöntem arasında yedi yüz kata yakın fark vardır. Küçük süreler ölçüm duyarlılığının sınırına yakındır; bu yüzden hızlı transpoz elli kez çalıştırılıp ortalama alınmıştır.

Sık yapılan hata

Transpozda yalnız `row` ile `col`'u değiştirip tabloyu sıralamayı unutmak. Sonuç doğru öğeleri içerir ama satır sırasında değildir. Bu tabloyla yapılan çarpım, aynı satırın öğelerinin ardışık durduğunu varsaydığı için yanlış sonuç verir. Hata küçük örneklerde gö-

rünmeyebilir: tek satırlık ya da köşegen bir matris, alanlar değiştirildiğinde de sıralı kalır.

5.8 Seyrek matris çarpımı

Matris çarpımı, seyrek gösterimin en çok işe yaradığı ve en dikkatli yazılması gereken işlemdir. Bu kesimdeki kod, Horowitz'in `mmult` programının sade bir uyarlamasıdır.

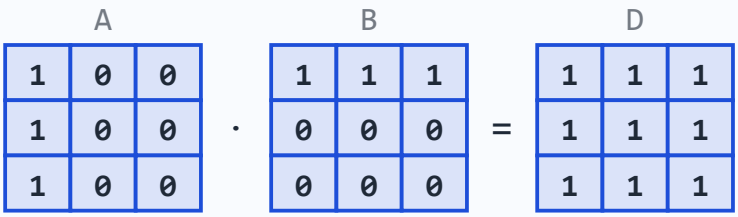
5.8.1 Satır çarpı sütun

A matrisi $m \times n$, B matrisi $n \times p$ boyutlu olsun. Çarpımları $D = A \cdot B$, $m \times p$ boyutludur ve her öğesi

$$d_{ij} = \Sigma_{k=0}^{n-1} a_{ik} \cdot b_{kj}$$

formülüyle bulunur. A'nın i. satırı ile B'nin j. sütunu öğe öğe çarpılıp toplanır. Seyrek matrislerde bu toplamın çoğu terimi sıfırdır. $a_{ik} \cdot b_{kj}$ çarpımı ancak iki öğe de doluysa sıfırdan farklıdır. Öyleyse yalnız aynı k değerinde dolu olan öğe çiftleri çarpılmalıdır.

İki seyrek matrisin çarpımı seyrek olmak zorunda değildir. Şekil 5.15'teki iki matrisin her birinde dokuz hücreden üçü doludur, ama çarpımlarının dokuz hücresinin hepsi doludur. A'nın bütün dolu öğeleri 0. sütunda, B'nin bütün dolu öğeleri 0. satırdadır; her (i, j) çifti $k = 0$'da buluşur.



Şekil 5.15: İki seyrek matrisin çarpımı dolu olabilir. A'da ve B'de dokuz hücreden üçü doludur; D'nin dokuz hücresinin hepsi 1'dir.

Üçlü gösterimde A'nın i. satırı ardışık durur. Ama B'nin j. sütunu tabloya dağılmıştır: her satırdan en çok bir öğe, tablonun farklı yerlerinde. Bir sütunu toplamak için bütün tabloyu taramak gerekir. Çözüm, önce B'nin transpozunu almaktır. `fast_transpose` ile elde edilen B^T'de B'nin sütunları ardışık satırlar olur. Sonra A'nın her satırı ile B^T'nin her satırı, `col` alanına göre birleştirilerek çarpılır. A'nın i. satırındaki bir öğenin sütunu k'dır; B^T'nin j. satırındaki bir öğenin sütunu da B'deki satırı, yani yine k'dır. İki satırın aynı `col` değerli öğeleri çarpılacak çiftlerdir.

Transpozu alınan matrisin hangisi olduğu önemlidir. "Önce ilk matrisin transpozu alınır" demek yanlıştır. Horowitz'in `mmult`'unda ve mantık olarak transpozu alınan ikinci matris, B'dir. A'nın satırları zaten ardışıktır; sorun B'nin sütunlarındadır.

5.8.2 Yardımcılar: son(), carp(), sakla()

Çarpım üç küçük yardımcıyla yazılır. İlk ikisi şunlardır:

```
/* s'deki terimin satırının bittiği yer */
int son(term t[], int s) {
  int e = s;
  while (e <= t[0].value &&
         t[e].row == t[s].row) e++;
  return e;
}
/* a[x..ex) satırı · b[y..ey) satırı */
int carp(term a[], int x, int ex,
         term b[], int y, int ey) {
  int sum = 0;
  while (x < ex && y < ey)
    if (a[x].col < b[y].col) x++;
    else if (a[x].col > b[y].col) y++;
    else sum += a[x++].value * b[y++].value;
  return sum;
}
```

`son(t, s)`, `t[s]` 'deki üçlüyle aynı satırda olan üçlülerin bittiği yeri bulur. `e`, `s` 'den başlar ve satır numarası değişene ya da tablo bitene kadar ilerler. Döndürülen `e`, satırın son öğesinden bir sonraki indistir. Satır, yarı açık *[s, e)* aralığıdır: `s` dahil, `e` hariç. Bu yazım C'deki dizi döngülerinin alışılmış biçimidir; boş aralık *s = e* olarak görünür ve aralığın boyu *e − s*'dir. Koşuldaki sıra önemlidir: `e <= t[0].value` önce sınanır. `&&` işleci kısa devre yaptığı için tablonun sonunu geçen bir `e` ile `t[e]` hiç okunmaz.

`carp`, `a` 'nın *[x, ex)* satırı ile `b` 'nin *[y, ey)* satırının nokta çarpımını hesaplar. Bu, `padd` gibi iki işaretçili bir birleştirmedir. İki satırın öğeleri sütuna göre sıralıdır. Sütunlar farklıysa küçük olan işaretçi ilerler, çünkü o sütunun öteki satırda eşi yoktur. Sütunlar eşitse iki değer çarpılıp toplama eklenir ve iki işaretçi de ilerler. Döngü, satırlardan biri bitince durur; kalan öğelerin eşi olamaz. Maliyet iki satırın öğe sayılarının toplamıyla orantılıdır.

Üçüncü yardımcı sonucu yazar:

```
void sakla(term d[], int *nd,
      int r, int c, int s) {
  if (s == 0) return;
  if (++*nd >= MAX_TERMS) {
    fprintf(stderr, "Cok terim\n");
    exit(1);
  }
  d[*nd].row = r;
  d[*nd].col = c;
  d[*nd].value = s;
}
```

`sakla`, toplam 0 ise hiçbir şey yazmaz; sıfır öğeler seyrek tabloya girmez. Değilse sayaç önce artırılır, sonra üçlü yazılır. `++*nd` ifadesinde ön ek `++` ve `*` aynı öncelikte ve sağdan sola bağlanır: `++(*nd)` anlamına gelir. Önce `nd` 'nin gösterdiği sayaç bir artar, sonra yeni değer sınırla karşılaştırılır. Sonucun `MAX_TERMS` 'ı aşması bir hata durumudur. Çarpımın seyrek kalacağının bir garantisi olmadığı için bu denetim süs değildir.

5.8.3 smult() ve kod izleme

```
void smult(term a[], term bt[], term d[]) {
  int ra, ea, rb, eb, nd = 0;
  for (ra = 1; ra <= a[0].value; ra = ea) {
    ea = son(a, ra);          /* A satırı [ra,ea) */
    for (rb = 1; rb <= bt[0].value; rb = eb) {
      eb = son(bt, rb);       /* B sütunu [rb,eb) */
      sakla(d, &nd, a[ra].row, bt[rb].row,
            carp(a, ra, ea, bt, rb, eb));
    }
  }
  d[0].row = a[0].row;  d[0].col = bt[0].row;
  d[0].value = nd;
}
```

Fonksiyon *A*'yı ve *B*'nin transpozu `bt`'yi alır. Dış döngü *A*'nın satırlarında gezinir: `ra` satırın ilk üçlüsü, `ea = son(a, ra)` satırın sonudur. Bir sonraki tur `ra = ea` ile başlar, yani bir sonraki satırdan. İç döngü aynı biçimde `bt`'nin satırlarında, yani *B*'nin sütunlarında gezinir. Her satır çifti için `carp` nokta çarpımı hesaplar ve `sakla` sıfır değilse sonuca yazar. Sonuç üçlüsünün satırı `a[ra].row`, sütunu `bt[rb].row`'dur: B^T'nin satır numarası *B*'nin sütun numarasıdır.

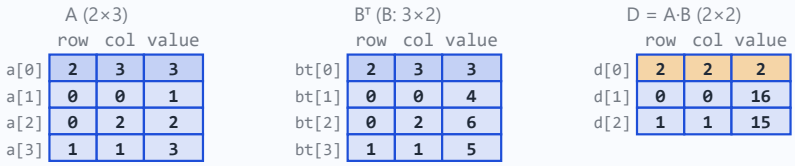
Sonuç üçlüleri kendiliğinden doğru sıradadır. Dış döngü *A*'nın satırlarında, iç döngü *B*'nin sütunlarında artan sırada gezinir. Bu, satır öncelikli sıranın ta kendisidir. Başlık en sona yazılır, çünkü dolu öğe sayısı ancak döngüler bitince bilinir. Boş satırlar ve boş sütunlar hiç ziyaret edilmez; döngüler yalnız tabloda bulunan satırlarda gezinir.

İzleme için *A = [[1, 0, 2], [0, 3, 0]]* (*2 × 3*) ve *B = [[4, 0], [0, 5], [6, 0]]* (*3 × 2*) kullanılır. Tablolar şöyledir: `a` = (2, 3, 3) | (0, 0, 1), (0, 2, 2), (1, 1, 3). *B*'nin transpozu *2 × 3*'tür: `bt` = (2, 3, 3) | (0, 0, 4), (0, 2, 6), (1, 1, 5).

Adım	*A* satırı	*B* sütunu	`carp`	`sakla`	`nd`
1	0: `a[1..2]` (sütun 0, 2)	0: `bt[1..2]` (sütun 0, 2)	*1·4 + 2·6 = 16*	`d[1] = (0, 0, 16)`	1
2	0: `a[1..2]`	1: `bt[3..3]` (sütun 1)	ortak sütun yok: 0	saklanmaz	1
3	1: `a[3..3]` (sütun 1)	0: `bt[1..2]`	ortak sütun yok: 0	saklanmaz	1
4	1: `a[3..3]`	1: `bt[3..3]`	*3·5 = 15*	`d[2] = (1, 1, 15)`	2
5	başlık: `d[0] = (2, 2, 2)`				2

Tablo 5.11: `smult(a, bt, d)` izlemesi.

Birinci adımda `carp`, *A*'nın 0. satırının sütunları {0, 2} ile *B*'nin 0. sütununun satırları {0, 2}'yi birleştirir. İki eşleşme vardır: *k = 0*'da *1 · 4*, *k = 2*'de *2 · 6*. Toplam 16'dır. İkinci adımda *A*'nın sütunları {0, 2}, *B*'nin 1. sütununun satırları {1}'dir; ortak değer yoktur ve toplam 0 olduğu için hiçbir şey yazılmaz. Sonuç *D = [[16, 0], [0, 15]]*'tir.



Şekil 5.16: `smult` bittiğinde: *A*, B^T ve sonuç *D*'nin üçlü tabloları. *D*'nin başlığı en son yazılmıştır: *2 × 2*, iki dolu öğe.

Kodun doğruluğu küçük örneklerin ötesinde de sınanmıştır. Test programı, boyutları 1 ile 6 arasında rastgele seçilen ve yaklaşık dörtte biri dolu 3.000 matris çifti üretmiştir. Her çiftin `fast_transpose` + `smult` sonucu, üç iç içe döngüyle hesaplanan klasik çarpımla aynı çıkmıştır. Aynı testte `transpose`, `fast_transpose` ve tek dizili sürüm de her matriste aynı transpozu vermiştir. Şekil 5.15'teki örnek, çarpımın dokuz öğeli dolu bir tablo ürettiğini gösterir.

5.8.4 Karmaşıklık

A'nın dolu öğe sayısı t_A, *B*'ninki t_B olsun. Maliyet üç parçadan oluşur:

- **Transpoz.** `fast_transpose(B)` : $O(cols_B + t_B)$.
- **B^T'nin gezilmesi.** *A*'nın her satırı için `bt`'de baştan sona gezinilir; `son` çağrıları ve `carp`'ın `bt` tarafındaki adımları her satır için toplam $O(t_B)$ eder. *A*'nın en çok $rows_A$ satırı için: $\Sigma_i\, t_B = rows_A \cdot t_B$.
- ***A* satırlarının gezilmesi.** Her *(i, j)* çifti için `carp`, *A*'nın *i*. satırında da gezinir. *j* en çok $cols_B$ değer alır; her *j* için *A*'nın satırları toplam t_A öğedir: $\Sigma_j\, t_A = cols_B \cdot t_A$.

> **Teorem 5.3 (Seyrek çarpımın maliyeti)**
>
> Üçlü gösterimle verilen *A* (t_A dolu öğe) ve *B* (t_B dolu öğe) matrislerinin çarpımı, transpoz dahil $O(cols_B \cdot t_A + rows_A \cdot t_B)$ zamanda hesaplanır.

Transpozun $O(cols_B + t_B)$ maliyeti bu toplamın içinde kalır. Klasik çarpım $O(rows_A \cdot cols_A \cdot cols_B)$ sürer: üç iç içe döngü. Matrisler doluysa $t_A = rows_A \cdot cols_A$ ve $t_B = cols_A \cdot cols_B$ olur ve seyrek sürüm de $O(rows_A \cdot cols_A \cdot cols_B)$ düzeyine çıkar. Derece aynıdır, ama sabit daha büyüktür: her öğe için üç alan okunur, karşılaştırmalar yapılır.

Seyrek sürüm, t_A ve t_B matris boyutlarının küçük bir kesri olduğunda değer. Örneğin 1000×1000 boyutlu iki matrisin her satırında beş dolu öğe olsun: $t_A = t_B = 5000$. Klasik çarpım 10^9 çarpma yapar. Seyrek çarpımın sınırı $1000 \cdot 5000 + 1000 \cdot 5000 = 10^7$'dir; yüz kat daha azdır. Sonuç D'nin seyrek olacağı ise garanti değildir. Şekil 5.15'teki gibi dolu bir sonuç `MAX_TERMS`'ı aşabilir; `sakla`'daki denetim bu yüzden vardır.

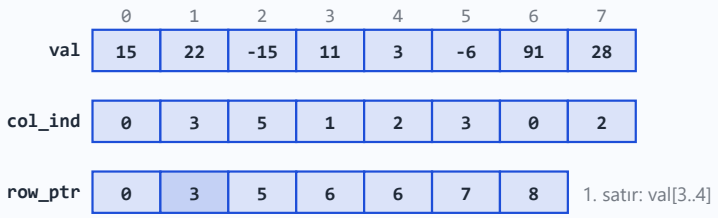
> **Sık yapılan hata**
>
> Çarpımda A'nın transpozunu almak. A'nın satırları zaten ardışıktır; transpozu almak onları dağıtır ve sorunu çözmez. Transpozu alınması gereken, sütunları dağınık duran B'dir. Bir başka hata, sonucun seyrek kalacağını varsayıp sonuç tablosunu küçük ayırmaktır.

5.9 Öteki gösterimler

Üçlü gösterim öğretici ve basittir, ama tek seçenek değildir. Bilimsel hesapta en çok kullanılan biçim sıkıştırılmış satır gösterimidir. Sık ekleme ve silme gereken uygulamalarda ise bağlı gösterim öne çıkar.

5.9.1 Sıkıştırılmış satır gösterimi (CSR)

> **Tanım 5.6**
>
> **Sıkıştırılmış satır gösterimi** (*compressed sparse row*, CSR), seyrek matrisi üç diziyle tutar. `val` dolu öğelerin değerlerini, `col_ind` sütun numaralarını satır öncelikli sırayla tutar; ikisinin de boyu t'dir. `row_ptr` dizisinin boyu $rows + 1$'dir: `row_ptr[i]`, i. satırın ilk öğesinin `val`'deki indisidir ve `row_ptr[rows] = t`'dir.



Şekil 5.17: 6×6 örnek matrisin CSR gösterimi. 1. satır `val[3..4]`'tür; boş olan 3. satır, art arda iki eşit `row_ptr` değeriyle (6, 6) görünür.

Üçlü gösterimde her öğenin satır numarası ayrı ayrı yazılır. Aynı satırdaki öğeler ardışık durduğu için bu bilgi gereksiz yere tekrarlanır. CSR, satır numaralarını atıp her satırın nerede başladığını tutar. i. satırın öğeleri `val[row_ptr[i] .. row_ptr[i+1] - 1]` aralığındadır; aralığın boyu `row_ptr[i+1] - row_ptr[i]`'dir.

6×6 örnekte `row_ptr = [0, 3, 5, 6, 6, 7, 8]`'dir. 0. satır `val[0..2]`'dir: 15, 22, −15. 3. satırda öğe yoktur; `row_ptr[3] = row_ptr[4] = 6`. Bellek: `val` 8, `col_ind` 8, `row_ptr` 7 tam sayı, toplam 23. Üçlü gösterim 27 tam sayı tutuyordu. Fark büyük matrislerde belirginleşir: üçlüler $3t$, CSR $2t + rows + 1$ tam sayı tutar.

CSR'nin en önemli kullanımı matris-vektör çarpımıdır: $y = M \cdot x$. Yinelemeli denklem çözücüleri bu işlemi binlerce kez yapar.

```
/* y = M * x  (matris-vektör çarpımı) */
for (i = 0; i < rows; i++) {
  y[i] = 0;
  for (k = row_ptr[i]; k < row_ptr[i+1]; k++)
    y[i] += val[k] * x[col_ind[k]];
}
```

Dış döngü satırlarda gezinir. İç döngü *i.* satırın dolu öğelerinde gezinir: her öğenin değeri, vektörün aynı sütundaki öğesiyle çarpılıp `y[i]` 'ye eklenir. `x[col_ind[k]]` ifadesi dolaylı erişimdir: önce sütun numarası okunur, sonra vektörün o konumu. Dış döngü *rows*, iç döngüler toplamda *t* tur döner. Maliyet *O(rows + t)*'dir. Örnek matris *x = (1, 2, 3, 4, 5, 6)* ile çarpılınca *y = (13, 31, −24, 0, 91, 84)* çıkar; ilk öğe *15 · 1 + 22 · 4 − 15 · 6 = 13*'tür.

`row_ptr`, `fast_transpose`'taki `start_pos` ile aynı fikirdir: her satırın başlangıç konumu, önceki satırların öğe sayılarının önek toplamıdır. Üçlü tablodan CSR kurmak da sayarak yapılır: her satırın öğeleri sayılır, önek toplamı alınır. Sütun öncelikli eşi **CSC**'dir (*compressed sparse column*): satırlar yerine sütunlar sıkıştırılır. Bir matrisin CSR gösterimi, transpozunun CSC gösterimiyle aynı dizilerdir.

5.9.2 Bağlı seyrek matris

Üçlü dizi ve CSR, sabit bir matris için çok iyidir, ama matris değiştikçe zorlanır. Bir öğe eklemek ya da silmek, sıralı diziyi kaydırmayı gerektirir ve *O(t)* sürer. Üçlü dizinin `MAX_TERMS` sınırı da vardır.

Bağlı gösterimde her dolu öğe bir düğümdür. Düğüm satırı, sütunu, değeri ve iki bağ tutar: aynı satırdaki sonraki öğeye ve aynı sütundaki sonraki öğeye. Aynı satırdaki düğümler bir **dairesel bağlı liste** (*circular linked list*) oluşturur; aynı sütundakiler de öyle. Her satırın ve her sütunun bir **başlık düğümü** (*header node*) vardır. Başlık düğümleri de kendi aralarında bağlıdır ve bütün matriste tek bir giriş noktasından gezinilebilir. Horowitz bu yapıyı kitabının bağlı listeler bölümünde ayrıntılı olarak kurar.

Bu gösterimin kazancı esnekliktir. Yeri bulunduktan sonra bir öğe eklemek ya da silmek birkaç bağ değişikliğiyle, *O(1)* zamanda yapılır. Hem satıra hem sütuna göre gezinmek kolaydır: bir satırda ya da bir sütunda gezinmek, ilgili dairesel listede gezinmektir. Bu yüzden çarpımda transpoza gerek kalmaz. Bedeli, öğe başına iki işaretçilik ek bellektir; 64 bitlik bir sistemde üç tam sayının 12 baytına 16 bayt daha eklenir.

Seyrek matrisin bağlı gerçeklemesi, bağlı listelerin doğrudan bir uygulamasıdır. Bir düğümü iki listeye birden bağlamak, ekleme ve silmede iki listenin bağlarını tutarlı güncellemek ve bütün düğümleri sızıntısız serbest bırakmak, bağlı liste becerisinin iyi bir sınavıdır. Dersin hocası bu çalışmanın her öğrencinin kendi emeği olması gerektiğini özellikle vurgular: iki ödev kâğıdı üst üste konup ışığa tutulduğunda aynı görünmemelidir.

5.10 Karmaşıklık özeti

Tablo 5.12, bu notta incelenen işlemlerin maliyetlerini bir arada verir.

İşlem	Zaman	Açıklama
Katsayı dizisiyle toplama	*O(d)*	*d*: derece; seyrek polinomda israf
`padd` (*n*, *m* terim)	*Θ(n + m)*	sıralı terimler + birleştirme
`pmult` (dizi, art arda `padd`)	$O(n^2 m)$	küçük polinom dışta: $O(min^2 \cdot max)$; artıklar birikir
`PolCarp` (bağlı liste)	$O(n^2 m)$ en kötü	`bas` ile arama kısalır; ölçümde dizilime bağlı
`transpose`	*O(cols · terms)*	dolu matriste $O(cols^2 \cdot rows)$
`fast_transpose`	*O(cols + terms)*	*O(cols)* ek bellek; saymalı sıralama
Seyrek çarpım (`smult` / `mmult`)	$O(cols_B \cdot t_A + rows_A \cdot t_B)$	önce B^T
CSR ile $M \cdot x$	*O(rows + terms)*	satır başı dizisi

Tablo 5.12: Polinom ve seyrek matris işlemlerinin maliyetleri.

Tablonun ortak bir dersi vardır. Bütün hızlı algoritmalar sıralılığa ve birleştirmeye dayanır: `padd` iki sıralı terim dizisini, `carp` iki sıralı satırı birleştirir; `fast_transpose` ise saymalı sıralamayla sıralı bir tablo üretir. Seyrek gösterimin bellek kazancı ancak bu sıra korunursa zaman kazancına dönüşür.

5.11 Sık yapılan hatalar

Bu notun kodlarında en sık rastlanan hatalar şunlardır:

1. **Terimleri sırasız tutup `padd`'i yine de iki işaretçiyle yazmak.** Birleştirme yalnız sıralı girdide doğrudur. Sırasız girdide aynı üslü iki terim birbirini hiç görmeyebilir ve sonuçta iki ayrı terim olarak kalır.

2. **Toplamı 0 olan terimi sonuca eklemek ya da listeden silmemek.** `padd`'deki `if (c)` denetimi ve `yerlestir`'deki `SonrakiTerimiSil` çağrısı bu yüzden vardır. Silinen düğümün belleği `free` ile geri verilmelidir.
3. `attach`**'ta** `float` **katsayıyı** `int` **parametreyle kırpmak;** `avail` **taşmasını denetlememek.** Kırpma derleyici uyarısı olmadan olur. Taşma denetimi olmadan dizinin sonundan sonrası bozulur.
4. **Transpozda yalnız** `row` **ve** `col` **alanlarını değiştirip sıralamayı unutmak;** `M[0]` **başlığını terim sanmak.** Sırasız transpoz, çarpım gibi sıraya dayanan işlemleri bozar. Döngüler `1`'den başlamalıdır.
5. `start_pos[M[i].col]++` **ifadesinde ön ek ile son ek farkını karıştırmak.** Son ek, eski değeri kullanıp sonra artırır; ön ek kullanılırsa her üçlü bir konum kayar ve son konuma ait olan üçlü tablonun dışına yazılır.
6. **Seyrek çarpımda *A*'nın transpozunu almak; sonucun seyrek kalacağını varsaymak.** Transpozu alınacak matris *B*'dir. Sonuç dolu olabilir; sonuç tablosunun sınırı denetlenmelidir.
7. **Bağlı listede silme sırasını karıştırmak.** Düğümün adresi saklanmadan bağ güncellenirse bellek sızar; bağ güncellenmeden düğüm serbest bırakılırsa sarkan işaretçi kalır.
8. **Yerel sayaç dizilerini sıfırlamamak.** `row_terms` gibi yerel diziler C'de belirsiz değerle başlar; sayma döngüsünden önce sıfırlanmalıdır.

5.12 Özet

- **Gösterim önemlidir.** Katsayı dizisi dolu ve düşük dereceli polinomda basit ve hızlıdır, ama seyrek polinomda belleği ve zamanı sıfırlara harcar. Terim dizisi yalnız sıfırdan farklı terimleri, büyük üsten küçüğe sıralı olarak tutar.
- **Sıralılık ve birleştirme.** `padd`, sıralı iki terim dizisini iki işaretçiyle yürüyerek $\Theta(n + m)$ zamanda toplar. Toplamı 0 olan terim sonuca yazılmaz. Ortak `P` dizisinde artıkların yeri geri kazanılmaz; sıkıştırma ya da bağlı liste gerekir.
- **Dizi gösterimiyle çarpım.** `pmult`, her terimle tek terimli çarpım yapıp sonucu `padd` ile ekler: $O(n^2 m)$. Küçük polinom dış döngüye alınmalıdır. Çok sayıda polinom öncelikli kuyrukla ya da ikişer ikişer birleştirilir.
- **Bağlı liste.** Terim sayısı bilinmiyorsa esnektir. `PolCarp` çarpımları sıralı listeye yerleştirir, aynı üsleri birleştirir, sıfırlananları siler ve belleğini geri verir. `bas` değişkeni aramayı kısaltır; en kötü durum yine $\Theta(n^2 m)$'dir.
- **Üçlüler.** Seyrek matris ⟨satır, sütun, değer⟩ üçlüleri ve `M[0]` başlığıyla tutulur. Üçlüler satır, sonra sütun sırasındadır. Gösterim $3(t + 1) < rows \cdot cols$ iken, kabaca matrisin üçte birinden azı doluyken kazançlıdır.
- **Transpoz.** Alanları değiştirmek sırayı bozar. `transpose` sütun sütun tarayarak $O(cols \cdot t)$, `fast_transpose` önce sayıp sonra yerleştirerek $O(cols + t)$ zamanda çalışır. Hızlı transpoz, sütun numarasını anahtar alan saymalı sıralamadır.
- **Çarpım.** Önce *B*'nin transpozu alınır; sonra *A*'nın her satırı B^T'nin her satırıyla birleştirilerek çarpılır: $O(cols_B \cdot t_A + rows_A \cdot t_B)$. Sonucun seyrek kalacağı garanti değildir.
- **Öteki gösterimler.** CSR, satır numaraları yerine satır başlangıçlarını tutar ve matris-vektör çarpımını $O(rows + t)$'de yapar; `row_ptr`, `start_pos` ile aynı önek toplamı fikridir. Bağlı seyrek matris, ekleme ve silmeyi $O(1)$'e indirir, ama öğe başına iki işaretçi harcar.

5.13 Kavram listesi

polinom *polynomial* · soyut veri tipi *abstract data type* · birleştirme *merge* · öncelikli kuyruk *priority queue* · bellek sızıntısı *memory leak* · sarkan işaretçi *dangling pointer* · seyrek matris *sparse matrix* · üçlü *triple* · satır öncelikli düzen *row-major order* · transpoz *transpose* · hızlı transpoz *fast transpose* · saymalı sıralama *counting sort* · anahtar *key* · sıkıştırılmış satır gösterimi *compressed sparse row* · dairesel bağlı liste *circular linked list* · başlık düğümü *header node*

5.14 Kendini Dene

1. $A = 5x^6 - 2x^3 + 4$ ve $B = 2x^6 + 2x^3 + x - 4$ için `padd`'in ana döngüsü kaç tur döner, kaç `attach` yapılır ve sonuç nedir?

> **Cevap**
>
> Döngü dört tur döner: (6, 6) eşit, $5 + 2 = 7$ yazılır; (3, 3) eşit, $-2 + 2 = 0$ yazılmaz; (0, 1) *B* büyük, *x* yazılır; (0, 0) eşit, $4 - 4 = 0$ yazılmaz. İki polinom da aynı anda biter. İki `attach` yapılır ve sonuç $7x^6 + x$'tir.

2. `avail = 5` iken $A = x^2 + 1$ ile $B = x^2 + x + 1$, `pmult` ile çarpılıyor. Sonuç nedir ve iş bitince `avail` kaç olur? Kullanılan konumların kaçı sonucu tutar?

> **Cevap**
>
> Sonuç $x^4 + x^3 + 2x^2 + x + 1$'dir (5 terim). Birinci turda 3 geçici terim yazılır ve `padd` onları boş ara sonuca kopyalar (3 terim daha). İkinci turda 3 geçici terim ve 5 terimlik yeni ara sonuç yazılır. Toplam $3 + 3 + 3 + 5 = 14$ konum kullanılır; `avail` 19 olur. 14 konumun yalnız 5'i sonuçtur.

3. $(x^2 + x)(x - 1)$ çarpımında `PolCarp` hangi terimi siler? Sonuç nedir?

> **Cevap**
>
> $p = x^2$ turunda x^3 ve $-x^2$ eklenir. $p = x$ turunda $x \cdot x = +x^2$ gelir; katsayı $-1 + 1 = 0$ olur ve x^2 düğümü silinir. Son çarpım $-x$ sona eklenir. Sonuç $x^3 - x$'tir.

4. `PolCarp`'ta sonuç listesinin ilk düğümü neden hiçbir zaman silinmez? Bu özellik hangi yardımcının kısıtını zararsız kılar?

> **Cevap**
>
> İlk düğüm, iki polinomun en büyük üslü terimlerinin çarpımıdır. Üssü öteki bütün çarpımlardan büyüktür ve başka hiçbir terim çifti aynı üssü üretemez; katsayısı hiç güncellenmez ve 0 olamaz. `SonrakiTerimiSil` yalnız verilen düğümden sonrakini silebilir, ilk düğümü silemez. İlk düğüm silinmediği için bu kısıt sorun yaratmaz.

5. 100×100 bir matriste üçlü gösterim, iki boyutlu diziden en çok kaç dolu öğeye kadar daha az yer kaplar?

> **Cevap**
>
> Koşul $3(t + 1) < 10.000$'dir. $t = 3332$'de üçlüler $3 \cdot 3333 = 9999$ tam sayı tutar ve daha küçüktür; $t = 3333$'te 10.002 tam sayı tutar. Sınır $t \le 3332$'dir, yani matrisin yaklaşık üçte biri.

6. 4×5 boyutlu bir matrisin dolu öğeleri (0, 2, 1), (1, 0, 3), (1, 2, −1), (2, 4, 6), (3, 2, 2)'dir. `fast_transpose`'un `row_terms` ve `start_pos` dizileri nedir? Transpozun üçlüleri hangi sırayla yazılır? `transpose` kaç karşılaştırma yapar?

> **Cevap**
>
> Sütunlar 2, 0, 2, 4, 2'dir: `row_terms = [1, 0, 3, 0, 1]`, `start_pos = [1, 2, 2, 5, 5]`. Transpoz (0, 1, 3), (2, 0, 1), (2, 1, −1), (2, 3, 2), (4, 2, 6)'dır. 1. ve 3. satırlar boştur; bu yüzden `start_pos[1] = start_pos[2]` ve `start_pos[3] = start_pos[4]`. `transpose`, $5 \cdot 5 = 25$ karşılaştırma yapar.

7. Aynı 4×5 matrisin CSR gösteriminde `row_ptr` nedir? Üç dizi toplam kaç tam sayı tutar?

> **Cevap**
>
> `row_ptr = [0, 1, 3, 4, 5]`'tir: 0. satırda bir, 1. satırda iki, 2. ve 3. satırlarda birer öğe vardır. `val` ve `col_ind` beşer, `row_ptr` beş tam sayı tutar: toplam 15. Üçlü gösterim $3 \cdot 6 = 18$ tam sayı tutardı.

8. `fast_transpose`'ta `j = start_pos[M[i].col]++;` yerine `j = ++start_pos[M[i].col];` yazılırsa 3×4 izleme matrisinde ne olur?

> **Cevap**
>
> Ön ek önce artırır, sonra yeni değeri verir. İlk üçlü (0, 0, 7) `T[1]` yerine `T[2]`'ye yazılır. Her üçlü kendi doğru yerinin bir ilerisine gider ve son üçlü (2, 0, 2) `T[3]`'e, (0, 3, 3) ise `T[5]`'e düşer. `T[1]` boş kalır. `T[5]`, dört dolu öğeli tablonun son geçerli konumu `T[4]`'ün ötesindedir; dizi tam beş üçlülük ayrılmış olsaydı sınır aşılırdı.

9. 1000×1000 boyutlu iki matrisin her birinde 3.000 dolu öğe var. Seyrek çarpımın işlem sınırı nedir? Klasik çarpımın çarpma sayısının kaçta biridir?

> **Cevap**
>
> Seyrek çarpımın sınırı $cols_B \cdot t_A + rows_A \cdot t_B = 1000 \cdot 3000 + 1000 \cdot 3000 = 6 \cdot 10^6$'dır. Klasik çarpım $1000^3 = 10^9$ çarpma yapar. Seyrek yöntem yaklaşık 167 kat daha az iş yapar.

5.15 İleri okuma

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C* (2. baskı), Kesim 2.4–2.5 ve 4.4, 4.7: bu notun `padd`, `transpose`, `fast_transpose` ve `mmult` kodlarının kaynağıdır; bağlı polinomu ve dairesel listelerle kurulan bağlı seyrek matrisi ayrıntılı verir.
- Y. Saad, *Iterative Methods for Sparse Linear Systems* (2. baskı), Kesim 3.4: CSR, CSC ve öteki seyrek depolama biçimlerini, bilimsel hesaptaki kullanımlarıyla birlikte anlatır.
- T. H. Cormen ve diğerleri, *Introduction to Algorithms* (3. baskı), Kesim 8.2: hızlı transpozun dayandığı saymalı sıralamayı, kararlılığını ve doğrusal zaman analizini verir.

5.16 Kaynaklar

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, Silicon Press, 2008 (Kesim 2.4–2.5: polinom ve seyrek matris; 4.4 ve 4.7: bağlı polinom ve seyrek matris).
2. Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2. baskı, SIAM, 2003 (Kesim 3.4: CSR/CSC biçimleri).
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3. baskı, MIT Press, 2009 (Kesim 8.2: saymalı sıralama).

KISIM II · BÖLÜM 6

Yığınlar

Yığın, eklemenin ve silmenin yalnız bir uçtan yapıldığı en basit doğrusal yapıdır. Son eklenen öğe ilk çıkar. Bu kısıt ilk bakışta yapıyı zayıflatıyor gibi görünür. Oysa "en son açılan iş ilk kapanır" kuralına uyan her sorunda yığın tam aranan araçtır: fonksiyon çağrıları, parantez denetimi, geri alma komutu ve geri izleme bu kurala uyar. Not, yığını önce soyut bir veri tipi olarak tanımlar. Ardından aynı arayüzü üç farklı biçimde gerçekler: sabit bir dizi ve `top` indisiyle, gerektikçe büyüyen bir diziyle ve bağlı listeyle. Her gerçeklemede bellekte neler olduğu çizimlerle, kodun her satırı da izleme tablolarıyla verilir. Son kesimler bilgisayarın kendi yığınını, yani çağrı yığınını, tek bir bellek dizisinde birden çok yığın tutmayı ve hangi çıkış sıralarının yığınla elde edilebildiğini ele alır.

Kod (C): `github.com/sercankulcu/data-structures-and-algorithms-c/Bolum06`

ÖĞRENME ÇIKTILARI

LIFO'yu tanımlamak
Yığını $S = \{a_0, \ldots, a_{n-1}\}$ listesi olarak tanımlamak, alt ve üst öğeyi göstermek ve bir işlem dizisinden sonra yığının durumunu bulmak.

Dizi ile gerçeklemek
`top` indisiyle ekleme ve çıkarma yordamlarını yazmak; taşma (*overflow*) ve alt taşma (*underflow*) durumlarını denetlemek; `++*top` ile `*top--` arasındaki farkı açıklamak.

Bağlı yığın kurmak
Düğümü `malloc` ile ayırıp bağlamak, çıkarılan düğümü `free` ile iade etmek ve bellek sızıntısı bırakmadan yığını boşaltmak.

Çağrı yığınını tanımak
Fonksiyon çağrılarının bir yığın gibi çalıştığını, her çağrının bir çerçeve ittiğini ve çağrı yığınının neden taştığını açıklamak.

Tasarım kararı vermek
Sabit dizi, büyüyen dizi ve bağlı liste arasında zaman, bellek ve hata riskine göre seçim yapmak; amortize maliyeti gerekçelendirmek.

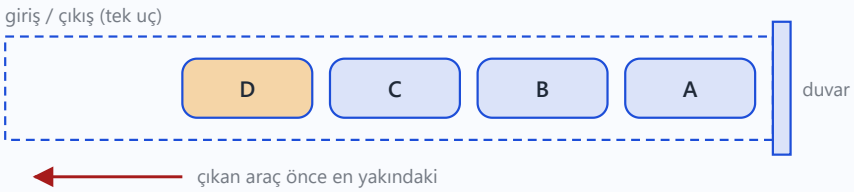
Çoklu yığını yönetmek
Tek bir bellek dizisinde n yığını `top[i]` ve `sinir[i]` dizileriyle paylaştırmak ve bu düzenin sınırlarını görmek.

6.1 Yığın: son giren ilk çıkar

Dizi ve bağlı liste, öğelere her yerden erişmeye izin verir. Dizide herhangi bir indise yazılabilir; listede herhangi bir düğümün arkasına düğüm eklenebilir. Yığın bu özgürlüğü bilerek kısıtlar. Öğeler yalnız bir uçtan girer ve yalnız aynı uçtan çıkar. Bu kesim yığını tanımlar ve bir işlem dizisi boyunca nasıl büyüyüp küçüldüğünü gösterir.

6.1.1 Tanım

Gündelik hayatta yığına benzeyen pek çok düzen vardır. Mutfakta üst üste konmuş tabaklardan önce en üsttekini alırız. Masadaki kâğıt destesine yeni gelen evrak en üste konur ve ilk okunan da odur. Şekil 6.1 daha öğretici bir örnek verir: tek araç genişliğinde, bir ucu duvarla kapalı bir çıkmaz sokak. Araçlar sokağa A, B, C, D sırasıyla girmiştir. Sokaktan ilk çıkabilecek araç, girişe en yakın olan D'dir. A'nın çıkabilmesi için önce D'nin, sonra C'nin, sonra B'nin çıkması gerekir. Sokağın bir ucu kapalı olduğu için girişin tersi sırada çıkış zorunludur.



Şekil 6.1: Tek arabalık çıkmaz sokak. Araçlar tek uçtan girer ve aynı uçtan çıkar. Son giren D, girişe en yakın olduğu için ilk çıkmak zorundadır; ilk giren A en son çıkar.

Tanım 6.1

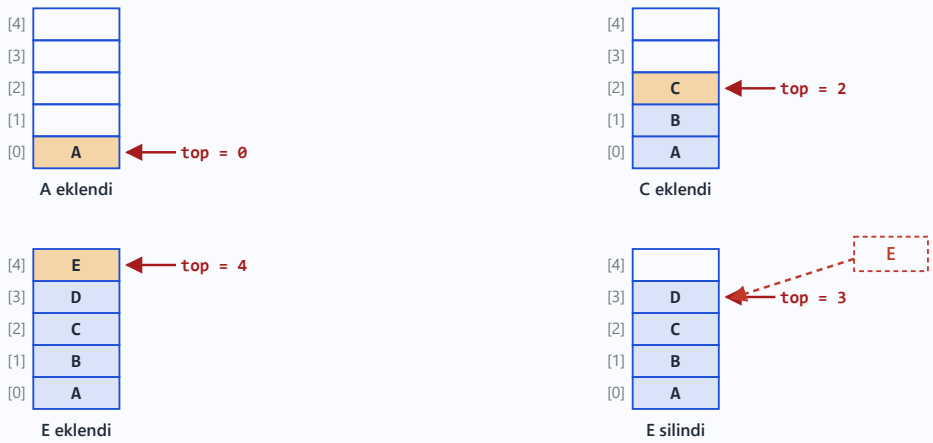
Yığın (*stack*), ekleme (*insertion*) ve silmenin (*deletion*) yalnız bir uçtan yapıldığı sıralı bir listedir. $S = \{a_0, a_1, \ldots, a_{n-1}\}$ yığınının **alt** (*bottom*) öğesi a_0'dır. **Üst** (*top*) öğesi a_{n-1}'dir. a_i öğesi a_{i-1}'in üstündedir ($0 < i < n$). Yeni öğe her zaman a_{n-1}'in üstüne konur; çıkarılan öğe her zaman o anki üst öğedir. Bu yüzden yığından ilk çıkan, ona son giren öğedir. Bu ilkeye **son giren ilk çıkar** (*last-in-first-out, LIFO*) denir.

Tanımdaki önemli sözcük "yalnız"dır. Yığının içinde öğeler sıralıdır, ama bu sıraya dışarıdan yalnız tek bir noktadan dokunulur. Yığına öğe eklemeye **push** (üste ekleme), üstteki öğeyi çıkarmaya **pop** (üstten çıkarma) denir. Bu iki sözcük Türkçe metinlerde de çoğunlukla İngilizce hâliyle kullanılır; not da bu kullanımı izler.

Türkçe kaynaklarda *stack* için hem "yığın" hem "yığıt" sözcükleri geçer. Bu derste *stack* için "yığın" kullanılır. Öncelikli kuyrukların gerçeklendiği *heap* veri yapısı için ise "heap" sözcüğü korunur. C programlarında yığın fonksiyonlarının adları ders kitabı geleneğiyle `yigit_ekle`, `yigit_al` biçiminde kalmıştır. Bu adlar yalnız tanımlayıcıdır; kavramın adı yığındır.

6.1.2 Yığının büyümesi ve küçülmesi

Boş bir yığına sırasıyla A, B, C, D ve E eklensin. Yığın dizi üzerinde tutulsun ve üst öğenin indisi `top` adlı bir değişkende saklansın. Boş yığında `top` −1'dir. Her eklemede `top` bir artar ve yeni öğe bu indise yazılır. Şekil 6.2 bu sürecin üç anını ve ardından yapılan ilk silmeyi gösterir.



Şekil 6.2: Boş yığına A, B, C, D, E eklenirken üç an ve ilk silme. A eklenince `top` = 0, C eklenince `top` = 2, E eklenince `top` = 4 olur. Silme yalnız üstten yapılır: ilk çıkan öğe en son eklenen E'dir ve `top` yeniden 3'e iner; üst öğe artık D'dir.

Her eklemeden sonra üst öğe değişir: A eklenince üst A'dır, B eklenince B'dir ve bu böyle sürer. E eklendiğinde yığında beş öğe vardır ve `top` 4'tür. Öğe sayısı her zaman `top + 1` 'dir. Ardından yapılan ilk silme E'yi verir, çünkü üstte E vardır. Bütün öğeler tek tek çıkarılsaydı çıkış sırası E, D, C, B, A olurdu. Bu, giriş sırasının tam tersidir.

Yığının ortasındaki bir öğeye, örneğin B'ye, doğrudan ulaşılamaz. B'yi almak için üstündeki C, D ve E'nin önce çıkarılması gerekir. Bu kısıt bir kusur değil, bir sözleşmedir. Yığını kullanan program, "en son açılan iş ilk kapanır" kuralının geçerli olduğu bir sorunla uğraşır. Yapının bu kuraldan başka bir şeye izin vermemesi programı yalınlaştırır ve hatayı zorlaştırır. Yığının kullanıldığı her yerde bu kural vardır.

Örnek 6.1

Boş bir yığına şu işlemler uygulanıyor: push(3), push(8), pop, push(5), push(1), pop, pop, push(9). Her adımdan sonra yığın (alttan üste) ve çıkan öğe şöyledir:

1. push(3): [3]
2. push(8): [3, 8]
3. pop: 8 çıkar; [3]
4. push(5): [3, 5]
5. push(1): [3, 5, 1]
6. pop: 1 çıkar; [3, 5]
7. pop: 5 çıkar; [3]
8. push(9): [3, 9]

Çıkan öğeler sırasıyla 8, 1, 5'tir. Sonunda yığında iki öğe vardır; alt öğe 3, üst öğe 9'dur ve dizi gerçeklemesinde `top` = 1'dir. 3 hiç çıkmamıştır, çünkü işlem dizisi boyunca hep en altta kalmıştır.

6.2 Yığın soyut veri tipi

Bir veri yapısını önce "ne yapar" sorusuyla, sonra "nasıl yapar" sorusuyla ele almak, programı değiştirilebilir kılar. **Soyut veri tipi** (*abstract data type, ADT*), bir değerler kümesini ve bu değerler üzerinde tanımlı işlemleri, nasıl gerçeklendiklerinden bağımsız olarak tanımlar. Yığın ADT'si, yığının hangi işlemleri sunduğunu ve bu işlemlerin hangi koşullarda hata verdiğini söyler. Diziyle mi, listeyle mi gerçeklendiğini söylemez.

6.2.1 İşlemler ve ön koşullar

Tablo 6.1 yığın ADT'sinin altı işlemini Horowitz–Sahni'nin ders kitabındaki adlarla verir. İşlemlerin yanındaki `push`, `pop` ve `peek` sözcükleri aynı işlemlerin yaygın İngilizce adlarıdır.

İşlem	Anlamı	Ön koşul / hata
`yigit_olustur(max)`	En çok `max` öğe alan boş yığın kurar	—
`yigit_dolu(S)`	Öğe sayısı `max` ise doğru döner	—
`yigit_ekle(S, x)` · push	`x`'i üste koyar	Yığın doluysa taşma
`yigit_bos(S)`	Öğe yoksa doğru döner	—
`yigit_al(S)` · pop	Üstteki öğeyi çıkarıp döndürür	Yığın boşsa alt taşma
`yigit_ust(S)` · peek	Üstteki öğeyi çıkarmadan okur	Yığın boşsa hata

Tablo 6.1: Yığın soyut veri tipi.

Tablodaki iki hata durumunun adları vardır. Dolu bir yığına öğe eklenmek istenmesine **taşma** (*overflow*), boş bir yığından öğe alınmak istenmesine **alt taşma** (*underflow*) denir. Taşma, yığının bir kapasitesi olduğu varsayımına dayanır. Kâğıt üzerindeki soyut yığın sınırsızdır; ama her gerçek bellek sonludur. Bu yüzden ADT, `yigit_olustur` işlemine bir `max` değeri verir ve `yigit_dolu` sorgusunu tanımlar.

Alt taşma ise kapasiteden bağımsızdır. Boş bir yığının üst öğesi yoktur; ondan öğe istemek anlamsızdır. `yigit_al` ve `yigit_ust` işlemlerinin ön koşulu "yığın boş değil"dir. Bu ön koşulu kimin denetleyeceği bir tasarım kararıdır. Çağıran taraf her `pop`'tan önce `yigit_bos`'u sorabilir. İşlemin kendisi de denetleyip bir hata kodu döndürebilir. İkinci yol daha güvenlidir, çünkü denetimi unutmak mümkün olmaz.

ADT'nin asıl değeri, kullanan kodla gerçekleyen kodu ayırmasıdır. Aynı altı işlem sabit bir diziyle, gerektikçe büyüyen bir diziyle ya da bağlı listeyle gerçeklenebilir. Yalnız ADT'deki işlemleri çağıran bir program, gerçekleme değiştiğinde değişmez. Notta bu üç gerçekleme sırayla kurulur ve aynı işlemlerin maliyetleri karşılaştırılır.

Sık yapılan hata

`pop` ile `peek`'i karıştırmak. `pop` üst öğeyi yığından çıkarır ve yığın bir öğe küçülür. `peek` (`yigit_ust`) yalnız üst öğenin değerini okur, yığın değişmez. Üstteki değere bakıp karar verecek bir döngüde `peek` yerine `pop` çağrılırsa her bakışta bir öğe kaybolur.

6.2.2 Yığının kullanıldığı yerler

Yığın, bilgisayar bilimindeki en çok kullanılan yapılardan biridir. Kullanım alanlarının ortak noktası, işlerin iç içe açılıp kapanmasıdır. İçteki iş dıştakinden önce biter. Aşağıdaki örneklerin her birinde yığının üstü "şu an açık olan en içteki işi" tutar.

Çağrı yığını
: Her fonksiyon çağrısı, dönüş adresi ve yerel değişkenleriyle birlikte bir çerçeveyi yığına iter. Fonksiyon dönünce çerçeve çıkarılır. En son çağrılan fonksiyon ilk biter.

İfade değerlendirme
: Derleyiciler `a + b * c` gibi ara ek (*infix*) bir ifadeyi önce sonek (*postfix*) gösterime çevirir. Bu ifadenin sonek biçimi `a b c * +`'dır. Derleyici sonra bu gösterimi bir yığınla hesaplar. Sonek ifadede her sayı yığına itilir, her işleç üstteki iki değeri alıp sonucu geri iter.

Geri izleme
: Labirentte yol arayan bir program her yol ayrımını yığına koyar. Çıkmaza girince yığının üstündeki, yani en son geçilen ayrıma döner ve oradan başka bir yol dener.

Parantez denetimi
: Bir ifadedeki her kapanan parantez, en son açılan ve henüz kapanmamış parantezle eşleşmelidir. "En son açılan" sözü yığını doğrudan çağrıştırır.

Geri al
: Bir metin düzenleyici yapılan her işlemi bir yığına koyar. Geri al komutu (Ctrl+Z) yığının üstündeki, yani en son yapılan işlemi geri alır. İkinci basış bir önceki işlemi geri alır.

Derinlik öncelikli arama
: Bir çizgede derinlik öncelikli arama (*depth-first search, DFS*), ziyaret edilecek düğümleri bir yığında tutar. Böylece en son keşfedilen yoldan olabildiğince derine gidilir.

Sunumdaki etkileşimli yığın aracı bu kuralı tek düğmeyle gösterir. Dört kez push, ardından dört kez pop yapıldığında çıkan sıra, giren sıranın tersidir. Beşinci pop boş yığına yapılır ve araç bunu alt taşma olarak bildirir.

6.2.3 Parantez denetimi

Kullanım alanlarından biri, yığının neden doğal bir seçim olduğunu kısa bir kodla gösterir. Bir dizgideki `(`, `[` ve `{` parantezlerinin dengeli olup olmadığı sorulsun. Dengeli bir dizgide her kapanan parantezin bir açanı vardır, türleri uyar ve sonunda açık parantez kalmaz. `{a[i] = (b + c) * d;}` dengelidir. `([)]` dengeli değildir: sayılar tutar, ama `)` geldiğinde en son açılan parantez `[`'dir.

Aşağıdaki fonksiyon, bir dizi ile üst indisini aynı yapıda tutan `Yigit` tipini kullanır. `push` ve `pop` başarı durumunu `bool` olarak döndürür; `pop` çıkardığı değeri ikinci argümanın gösterdiği yere yazar.

```
/* (, [ ve { için denge denetimi: her kapanan, en son açılanla eşleşmeli */
bool dengeli(const char *s) {
    Yigit y;  int ac;
    yigit_baslat(&y);
    for (int i = 0; s[i] != '\0'; i++) {
        char c = s[i];
        if (c == '(' || c == '[' || c == '{') {
            if (!push(&y, c)) return false;       /* çok derin */
        } else if (c == ')' || c == ']' || c == '}') {
            if (!pop(&y, &ac)) return false;      /* açılmamış kapanış */
            if ((c == ')' && ac != '(') || (c == ']' && ac != '[') ||
                (c == '}' && ac != '{'))
                return false;                     /* tür uyuşmuyor */
        }
    }
    return yigit_bos(&y);                         /* açık kalan var mı? */
}
```

Fonksiyon dizgiyi soldan sağa bir kez tarar. Açan bir parantez görünce onu yığına iter. Kapanan bir parantez görünce yığının üstündeki açanı çeker. Yığın o anda boşsa, kapanan parantezin açanı yoktur ve sonuç yanlıştır. Yığından çıkan açanın türü kapananınkiyle uyumuyorsa sonuç yine yanlıştır. Parantez olmayan karakterler atlanır. Tarama bittiğinde yığında öğe kalmışsa bazı parantezler hiç kapanmamıştır. Bu yüzden fonksiyon en sonda yığının boş olup olmadığını döndürür.

Fonksiyon şu girdilerle denendi: `{a[i] = (b + c) * d;}` ve `x = f(a[2], {1, 2});` için 1 (doğru), `([)]`, `(()`, `())` ve `]` için 0 (yanlış) döndü. Boş dizgi dengelidir ve 1 döner. Her karakter en çok bir push ve bir pop doğurur. Push ve pop sabit sürdüğü için n karakterlik dizgide süre $\Theta(n)$'dir. En kötü durumda bütün karakterler açan parantezdir ve yığın n öğe tutar; ek bellek $O(n)$'dir.

6.3 Dizi ile yığın

Yığını gerçeklemenin en kısa yolu tek boyutlu bir dizidir. Dizinin 0 numaralı hücresi yığının altıdır. Öğeler 1, 2, 3 numaralı hücrelere doğru yukarı dizilir. Üst öğenin indisi ayrı bir tam sayı değişkende, `top`'ta tutulur. Bu kesim önce `top` değişkeninin anlamını ve iki sınır durumunu, sonra Horowitz–Sahni'nin C kodunu satır satır ele alır.

6.3.1 `top` indisi, taşma ve alt taşma

Dizi yığınında bütün bilgi iki şeydedir: dizinin içeriği ve `top`'un değeri. `top`, yığının üstündeki öğenin indisidir. Yığında k öğe varsa bu öğeler `yigit[0]`'dan `yigit[k-1]`'e kadar olan hücrelerdedir ve `top` $= k - 1$'dir. Yığın boşken hiçbir öğe yoktur; bu durumda `top` $= -1$ seçilir. -1 geçerli bir indis değildir, ama ilk eklemede `top` bir artırılınca tam olarak 0 olur. Bu seçim kodu yalınlaştırır.



Şekil 6.3: Beş hücrelik bir dizi yığınının iki sınır durumu. Solda boş yığın: `top` $= -1$, yani 0 numaralı hücrenin bile altındadır; bu durumda `yigit_al` alt taşma verir. Sağda dolu yığın: A'dan E'ye beş öğe vardır ve `top` $= 4 =$ `MAX_STACK_SIZE - 1`'dir; bu durumda bir ekleme daha taşma olur.

Şekil 6.3 iki sınır durumunu yan yana gösterir. Yığının boş olup olmadığı `top < 0` sınamasıyla anlaşılır. Dizinin `MAX_STACK_SIZE` hücresi varsa son geçerli indis `MAX_STACK_SIZE - 1`'dir. `top` bu değere ulaştığında yığın doludur ve `top >= MAX_STACK_SIZE - 1` sınaması doğru olur. Öğe sayısı her zaman `top + 1`'dir. `top` ile öğe sayısını karıştırmak, "bir eksik ya da bir fazla" (*off-by-one*) hatalarının en bilinen kaynağıdır. Örneğin dolu denetimini `top >= MAX_STACK_SIZE` diye yazan bir program, dizinin son hücresinin bir ötesine yazar.

C, dizi sınırlarını çalışma sırasında denetlemez. `yigit[MAX_STACK_SIZE]` ifadesi derlenir ve çalışır; ama dizinin hemen arkasındaki bellek hücresine erişir. Orada başka bir değişken varsa onun değeri bozulur. Program hemen çökmeyebilir ve hata çok sonra, ilgisiz bir yerde ortaya çıkar. Bu yüzden taşma ve alt taşma denetimi yığın kodunun isteğe bağlı bir süsü değildir. Denetimsiz bir push, komşu belleğin üzerine yazar.

6.3.2 C tanımları

Horowitz–Sahni'nin ders kitabındaki ilk gerçekleme, yığını küresel bir dizi olarak tanımlar:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_STACK_SIZE 101

typedef struct {
    int deger;
    /* diğer alanlar */
} element;

element yigit[MAX_STACK_SIZE];

void yigit_dolu(void) {
    printf("Yigit dolu!!!\n"); exit(-1);
}
void yigit_bos(void) {
    printf("Yigit bos!!!\n"); exit(-2);
}
```

`#define MAX_STACK_SIZE 101` satırı yığının kapasitesini sabitler. Önişlemci, kodda geçen her `MAX_STACK_SIZE`'ı derlemeden önce 101 ile değiştirir. Kapasiteyi değiştirmek için bu satırı düzeltmek yeter.

`element` tipi, yığında saklanacak kaydı tanımlar. Şimdilik tek alanı vardır: `deger`. Yorum satırı, gerçek bir uygulamada bu kayda başka alanların da ekleneceğini hatırlatır. Örneğin bir labirent programı yığına satır, sütun ve yön bilgisini birlikte koyar. Öğeyi `int` yerine bir `struct` olarak tanımlamak, yığın kodunu öğenin içeriğinden bağımsız kılar. Kayda alan eklendiğinde yığın yordamları değişmez, çünkü onlar yalnız `element` değerlerini kopyalar.

`element yigit[MAX_STACK_SIZE];` satırı yığının kendisidir. Bu dizi her fonksiyonun dışında tanımlandığı için bir **küresel değişkendir** (*global variable*). Programın bütün fonksiyonları ona doğrudan erişebilir. Küresel diziler sıfırla başlatılır ve program boyunca yaşar. `top` ise küresel değildir: çağıran fonksiyonda (çoğunlukla `main`'de) yerel bir değişken olarak tanımlanır ve yığın yordamlarına adresiyle aktarılır.

Son iki fonksiyon hata yordamlarıdır. `yigit_dolu` bir mesaj basıp programı `exit(-1)` ile, `yigit_bos` ise `exit(-2)` ile sonlandırır. `exit`, `<stdlib.h>`'de bildirilir ve hangi fonksiyonun içinden çağrılırsa çağrılsın bütün programı bitirir. Verilen sayı programın çıkış kodudur ve işletim sistemine iletilir. Unix türü sistemlerde kabuk −1'i 255 olarak görür; Windows'ta bu kod 4294967295 olarak raporlanır. Bu iki fonksiyonun adı ADT'deki `yigit_dolu` ve `yigit_bos` sorgularıyla aynıdır, ama işleri farklıdır: sorgu yapmaz, hata bildirir. Ad çakışması bu ilk gerçeklemenin kusurlarından biridir; daha iyi bir tasarımda bu adlar gerçek sorgulara verilir.

Ders kitaplarında bu tanımlar tek bir dosyada, küresel dizi ve küresel `top` ile verilir. Dersin sınavlarında ve ödevlerinde ise küresel değişken yasaktır. Bunun nedeni basittir. Küresel bir değişkeni programın her fonksiyonu değiştirebilir. Bir hata çıktığında değişkeni kimin bozduğunu bulmak için bütün programı okumak gerekir. Ayrıca aynı programda iki yığın gerektiğinde küresel tek dizi işe yaramaz. Bu bölümde küresel dizi yalnız ders kitabındaki kodu tanımak için kullanılır; ödevlerde yığın, kuyruk ya da başka bir yapı neyse yerel olarak tanımlanıp parametreyle aktarılır.

6.3.3 `yigit_ekle` ve `yigit_al`

Ekleme ve çıkarma yordamları üçer satırdır:

```
void yigit_ekle(int *top, element item) {
    if (*top >= MAX_STACK_SIZE - 1)  /* yer var mı? */
        yigit_dolu();         /* uyar ve çık */
    yigit[++*top] = item;     /* önce işaretçiyi artır */
}                             /* sonra öğeyi yerleştir */

element yigit_al(int *top) {
    if (*top < 0)
        yigit_bos();          /* boşsa uyar ve çık */
    return yigit[(*top)--];   /* döndür, sonra azalt */
}
```

`yigit_ekle` iki parametre alır. `int *top`, çağırandaki `top` değişkeninin adresidir. `element item`, eklenecek öğenin bir kopyasıdır. İlk satır yer olup olmadığını sınar. `*top`, `top` işaretçisinin gösterdiği tam sayıdır, yani çağırandaki üst indisidir. Bu değer `MAX_STACK_SIZE − 1`'e ulaşmışsa yığın doludur ve `yigit_dolu()` programı bitirir. Yer varsa son satır çalışır: `yigit[++*top] = item;`. Bu satır iki iş yapar. Önce `*top` bir artar, sonra öğe yeni indise yazılır. Boş yığında `*top` −1'den 0'a çıkar ve öğe `yigit[0]`'a yazılır.

`yigit_al` yalnız `top`'un adresini alır. Yığın boşsa, yani `*top < 0` ise `yigit_bos()` programı bitirir. Değilse `return yigit[(*top)--];` satırı çalışır. Bu satır da iki iş yapar, ama ters sırada. Önce eski indisteki öğe okunur, sonra `*top` bir azalır. Okunan öğe fonksiyonun dönüş değeridir. Örneğin `*top` 2 iken `yigit[2]` döner ve `*top` 1 olur.

Çıkarılan öğe diziden silinmez. `yigit[2]` hücresinde eski değer durmaya devam eder. Ama artık yığının parçası değildir, çünkü yığının içeriğini `top` belirler: geçerli öğeler 0'dan `top`'a kadar olan hücrelerdir. Bir sonraki ekleme bu hücrenin üzerine yazar. Silmenin bir hücreyi temizlemek değil, sınırı aşağı çekmek olması, dizi yığınını hızlı yapan şeydir.

`top` neden adresle aktarılır? C'de bütün argümanlar **değerle çağrı** (*call by value*) kuralıyla aktarılır. `yigit_ekle(top, item)` diye bir çağrı, `top`'un değerinin bir kopyasını fonksiyona verirdi. Fonksiyon bu kopyayı artırır, ama çağırandaki `top` değişmezdi. Sonraki ekleme aynı hücreye yazardı. Çağıranın değişkenini değiştirmek isteyen bir fonksiyona o değişkenin adresi verilir: `yigit_ekle(&top, item)`. Fonksiyon `*top` üzerinden çağırandaki asıl değişkene erişir.

Bu noktada bir yazım kuralı önemlidir. Fonksiyonun parametre listesinde `*top` diye görünen bir parametre, fonksiyonun içinde de her yerde `*top` olarak kullanılmalıdır. İçeride yalnız `top` yazılırsa işaretçinin kendisi, yani bir adres kullanılmış olur. `if (top >= MAX_STACK_SIZE - 1)` gibi bir sınama adresi bir tam sayıyla karşılaştırır. Derleyici en iyi durumda bir uyarı verir; program ise anlamsız bir karşılaştırma yapar. Sınavlarda bu hata çok sert değerlendirilir, çünkü işaretçi ile gösterdiği değer arasındaki farkı bilmemek, bu dersteki bütün yapıları yanlış kurmak demektir.

İki yordamın maliyeti de *O(1)*'dir. Her biri bir karşılaştırma, bir indis değişikliği ve bir dizi erişimi yapar. Bu işlerin sayısı yığındaki öğe sayısına bağlı değildir. Ek bellek de sabittir: yalnız parametreler kullanılır.

6.3.4 Öncelik tuzağı: `*` ile `++` ve `--`

`yigit[++*top]` ve `yigit[(*top)--]` ifadeleri kısa ama dikkat ister. İlkinde parantez yoktur, ikincisinde vardır. Bu farkın nedeni C'deki işleç önceliğidir. Tablo 6.2 dört yazımı karşılaştırır.

İfade	C bunu nasıl okur	Sonuç
`++*top`	`++(*top)`: gösterilen tam sayıyı artır, yeni değeri kullan	Doğru (push)
`(*top)--`	gösterilen tam sayının değerini kullan, sonra azalt	Doğru (pop)
`*top--`	`*(top--)`: işaretçiyi azalt, eski adresteki değeri oku	Hatalı: `*top` değişmez
`*top++`	`*(top++)`: işaretçiyi ilerlet, eski adresteki değeri oku	Hatalı

Tablo 6.2: `top` bir `int *` iken dört yazımın anlamı.

Kural şöyledir. Son ek `++` ve `--` işleçleri, tekli `*` işlecinden daha önceliklidir. Bu yüzden `*top--` ifadesinde `--` önce `top`'a bağlanır: azalan şey işaretçinin kendisidir. Ön ek `++` ile tekli `*` ise aynı öncelik düzeyindedir ve sağdan sola bağlanır. `++*top` ifadesinde önce sağdaki `*` uygulanır, `++` onun sonucuna, yani gösterilen tam sayıya bağlanır. Bu yüzden push'ta parantez gerekmez, pop'ta gerekir.

`(*top)--` yerine `*top--` yazılırsa ne olur? İfade `*(top--)` olarak okunur. `top--`, işaretçinin eski değerini verir ve işaretçiyi bir `int` geriye kaydırır. `*` bu eski adresi okur. Okunan değer çağırandaki üst indisidir; dizi doğru hücreden okunur. Ama azalan şey yerel `top` işaretçisidir. Fonksiyon bitince bu yerel kopya yok olur. Çağırandaki `top` hiç değişmemiştir. Bu tür bir hata bu yüzden çok sinsidir: ilk `pop` doğru değeri döndürür.

Örnek 6.2

Yığında yalnız A vardır (`top` = 0). `yigit_al`'ın son satırı yanlışlıkla `return yigit[*top--];` diye yazılmıştır. İki çağrı arka arkaya yapılıyor.

1. Birinci çağrı: `*(top--)` çağırandaki 0'ı okur. `yigit[0]` = A döner. Azalan yerel işaretçidir; çağırandaki `top` 0 kalır.
2. İkinci çağrı: `*top < 0` yanlıştır, alt taşma yakalanmaz. Yine `yigit[0]` = A döner, `top` yine 0'dır.

Bu davranış derlenip çalıştırılarak doğrulandı: iki satır da "al: A top=0" yazdı ve derleyici `-Wall -Wextra` ile bile uyarı vermedi. Yığını boşaltmak için `while (!bos) al` biçiminde bir döngü yazan program sonsuz döngüye girer. `*top++` hatası eklemede daha kötüdür: `yigit[*top++] = item` boş yığında `yigit[-1]`'e, yani dizinin dışına yazar.

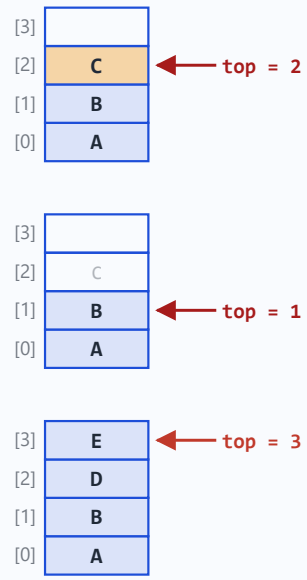
Bu tuzaktan kaçınmanın en sağlam yolu, işaretçi üzerinden yapılan artırma ve azaltmayı ayrı bir satıra almaktır: önce `(*top)++;`, sonra `yigit[*top] = item;`. Kod bir satır uzar, ama okuyan herkes ne olduğunu hemen anlar. Ders kitabındaki kısa yazım da doğrudur; yeter ki parantezin neden orada olduğu bilinsin.

6.3.5 Kod izleme

Yordamların davranışı küçük bir kapasiteyle daha iyi görülür. İzlemede `MAX_STACK_SIZE` = 4 alınır; dolu denetimi bu yüzden `*top ≥ 3`'tür. `main`'de `top` = −1 iken sırasıyla A, B, C eklenir, bir öğe alınır, sonra D, E ve F eklenmeye çalışılır. Tablo 6.3 her çağrıdan sonraki durumu verir.

Çağrı	Sınama	Yapılan	`top`	`yigit[0..top]`
başlangıç	—	—	−1	(boş)
`yigit_ekle(&top, A)`	−1 ≥ 3? Hayır	`++*top`, `yigit[0] = A`	0	A
`yigit_ekle(&top, B)`	0 ≥ 3? Hayır	`yigit[1] = B`	1	A B
`yigit_ekle(&top, C)`	1 ≥ 3? Hayır	`yigit[2] = C`	2	A B C
`yigit_al(&top)`	2 < 0? Hayır	`yigit[2]` = C döner, `top` azalır	1	A B
`yigit_ekle(&top, D)`	1 ≥ 3? Hayır	`yigit[2] = D` (C'nin üzerine)	2	A B D
`yigit_ekle(&top, E)`	2 ≥ 3? Hayır	`yigit[3] = E`	3	A B D E
`yigit_ekle(&top, F)`	3 ≥ 3? **Evet**	`yigit_dolu()`: "Yigit dolu!!!", `exit(-1)`	3	A B D E

Tablo 6.3: `MAX_STACK_SIZE` = 4 iken dizi yığını izlemesi.



Şekil 6.4: İzlemenin üç anı. Üstte C eklenmiştir: `top` = 2. Ortada C alınmıştır: `top` = 1'e inmiştir; C'nin değeri hücrede durur ama artık yığının parçası değildir. Altta D ve E eklenmiştir: `top` = 3 ve dizi doludur; F eklenmek istenince `yigit_dolu()` çağrılır.

Tablonun beşinci satırı dizinin silmeyi nasıl yaptığını gösterir. `yigit_al` C'yi döndürür ve `top`'u 1'e indirir. 2 numaralı hücrede C hâlâ durur. Altıncı satırda D eklenince aynı hücreye D yazılır ve C'nin izi kalmaz. Son satırda yer yoktur. `yigit_dolu` mesajı basar ve `exit(-1)` ile bütün programı bitirir. Bu davranış derlenip çalıştırılarak doğrulandı: program "ekle F ... Yigit dolu!!!" yazdı ve Windows çıkış kodunu 4294967295, yani −1 olarak raporladı.

Taşma anında bütün programı bitirmek kaba bir tepkidir. Yığın küçük bir kütüphane parçasıdır; programın geri kalanının ne yapacağını bilemez. Belki çağıran, yığın dolunca daha büyük bir yığın kuracaktı. Belki kullanıcıya bir mesaj gösterip işlemi iptal edecekti. `exit` bu kararları elinden alır. Ayrıca açık dosyaların düzgün kapatılması, ayrılmış belleklerin bırakılması gibi temizlik işleri de yapılamadan program biter. Daha iyi tasarım, hatayı çağırana bildirip kararı ona bırakmaktır.

6.4 Yığını bir nesne gibi tasarlamak

Ders kitabındaki ilk gerçeklemenin üç zayıflığı vardır. Yığın küresel bir dizidir; bu yüzden programda yalnız bir yığın olabilir. `top` diziden ayrı yaşar; çağıran onu ayrıca tanımlayıp her çağrıya ayrıca vermek zorundadır. Hata durumunda program `exit` ile biter. Bu kesim üç sorunu birlikte çözen bir tasarım kurar: dizi ve üst indisi aynı yapıda durur, yordamlar yığını parametre olarak alır ve hata bir dönüş değeriyle bildirilir.

6.4.1 Küresel dizi yerine `struct`

```
#include <stdbool.h>
#define KAP 100

typedef struct {
    int veri[KAP];
    int ust;              /* boşken -1 */
} Yigit;

void yigit_baslat(Yigit *y) { y->ust = -1; }
bool yigit_bos(const Yigit *y)  { return y->ust < 0; }
bool yigit_dolu(const Yigit *y) { return y->ust == KAP - 1; }
int  yigit_boyut(const Yigit *y) { return y->ust + 1; }
bool yigit_ust(const Yigit *y, int *x) {
    if (yigit_bos(y)) return false;
    *x = y->veri[y->ust];  return true;
}
```

`Yigit` yapısı iki alan taşır: `KAP` öğelik `veri` dizisi ve üst indisini tutan `ust`. Bir yığın artık tek bir değişkendir. `Yigit a, b;` tanımı birbirinden bağımsız iki yığın verir; her birinin kendi dizisi ve kendi üst indisi vardır. Biri dolarken öteki etkilenmez. Küresel değişken yoktur.

Her yordam yığını `Yigit *y` parametresiyle, yani adresiyle alır. Bunun iki nedeni vardır. Birincisi, `push` ve `pop` yığını değiştirir; değerle aktarılan bir kopya değişseydi çağırandaki yığın değişmezdi. İkincisi, `Yigit` büyük bir yapıdır: `KAP` = 100 iken yaklaşık 404 bayt tutar. Değerle aktarılsaydı her çağrıda bu 404 bayt kopyalanırdı. Adres ise 8 bayttır. Yapının alanlarına adres üzerinden ok işleciyle (`->`) erişilir: `y->ust` , `(*y).ust` 'un kısa yazımıdır.

Yığını değiştirmeyen sorgular `const Yigit *y` alır. `const` niteleyicisi, fonksiyonun gösterilen yapıyı değiştirmeyeceğine söz verir. Fonksiyonun içinde `y->ust = 0;` gibi bir atama yazılırsa derleyici hata verir. Bu söz hem okuyana bilgi verir hem de yanlışlıkla yapılan değişiklikleri derleme sırasında yakalar.

Yordamların her biri tek satırdır. `yigit_baslat` , `ust` 'u −1 yaparak boş yığın kurar. Yerel bir `Yigit` değişkeninin alanları başlangıçta belirsiz değerler taşır; bu yüzden her yığın kullanılmadan önce bir kez başlatılmalıdır. `yigit_bos` ve `yigit_dolu` artık gerçek sorgulardır ve `bool` döner. `yigit_boyut` öğe sayısını, yani `ust + 1` 'i verir. `yigit_ust` , ADT'deki peek işlemidir. Yığın boşsa `false` döner ve `*x` 'e dokunmaz. Değilse üst öğeyi `*x` 'e yazar ve `true` döner. Değerin bir çıktı parametresiyle verilmesinin nedeni, dönüş değerinin başarı bilgisine ayrılmış olmasıdır. `bool` tipi ve `true` , `false` adları `<stdbool.h>` başlığından gelir.

Ekleme ve çıkarma da aynı düzenle yazılır:

```
bool push(Yigit *y, int x) {
  if (y->ust == KAP - 1) return false; /* taşma */
  y->veri[++y->ust] = x;
  return true;
}
bool pop(Yigit *y, int *x) {
  if (y->ust < 0) return false;   /* alt taşma */
  *x = y->veri[y->ust--];
  return true;
}
```

`push` önce taşmayı sınar. `ust == KAP - 1` ise yer yoktur ve fonksiyon `false` döner. Program sürer; taşmayı öğrenen çağıran ne yapacağına kendisi karar verir. Yer varsa `y->veri[++y->ust] = x;` satırı önce `ust` 'u artırır, sonra değeri yeni indise yazar. Burada öncelik tuzağı yoktur: `->` işleci `++` 'dan önce uygulanır, yani `++y->ust` ifadesi `++(y->ust)` demektir.

`pop` önce alt taşmayı sınar. Yığın boşsa `false` döner ve `*x`'e dokunmaz. Değilse `*x = y->veri[y->ust--];` satırı üst öğeyi `*x`'e kopyalar ve ardından `ust`'u bir azaltır. `y->ust--` ifadesi de `(y->ust)--` olarak okunur; azalan şey yapının alanıdır.

Bu sürüm hata durumunu dönüş değeriyle bildirdiği için kullanımı çok rahattır. Örneğin yığını boşaltırken her çıkan öğeyi yazdıran döngü tek satırdır: `while (pop(&y, &x)) printf("%d ", x);`. Döngü, `pop` ilk kez `false` döndüğünde, yani yığın boşaldığında durur. İşlemlerin hepsi yine *O(1)*'dir.

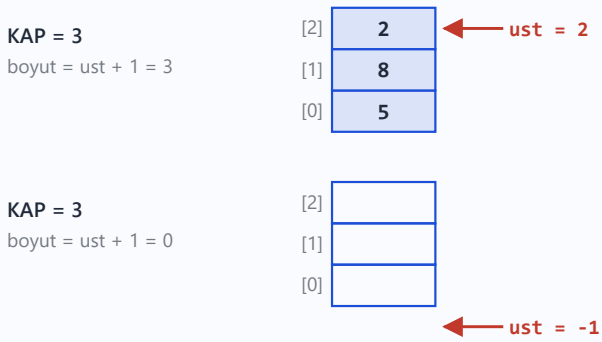
> **Sık yapılan hata**
>
> `bool` döndüren bir yordamın sonucunu yok saymak. `push(&y, x);` yazıp dönüş değerine bakmayan bir program, yığın dolduğunda öğeyi sessizce kaybeder. `pop(&y, &x);` yazıp sonucu denetlemeyen bir program, yığın boşken `x`'in eski ya da belirsiz değerini kullanmaya devam eder. Hata bildiren bir arayüz ancak bildirim okunursa işe yarar.

6.4.2 Kod izleme: `bool` döndüren `push` ve `pop`

İzlemede `KAP` = 3 alınır; dolu denetimi `ust == 2`'dir. `yigit_baslat(&y)` ile boş yığın kurulur. Ardından sırasıyla 5, 8, 2 ve 9 itilir, sonra dört kez `pop` çağrılır. Tablo 6.4 her çağrıyı ve dönüş değerini verir.

Çağrı	Sınama	Yapılan	Döner	`ust`	`veri[0..ust]`
`yigit_baslat(&y)`	—	`ust = -1`	—	−1	(boş)
`push(&y, 5)`	−1 == 2? Hayır	`veri[0] = 5`	true	0	5
`push(&y, 8)`	0 == 2? Hayır	`veri[1] = 8`	true	1	5 8
`push(&y, 2)`	1 == 2? Hayır	`veri[2] = 2`	true	2	5 8 2
`push(&y, 9)`	2 == 2? **Evet**	taşma, hiçbir şey yazılmaz	false	2	5 8 2
`pop(&y, &x)`	2 < 0? Hayır	`x = veri[2]`	true, `x` = 2	1	5 8
`pop(&y, &x)`	1 < 0? Hayır	`x = veri[1]`	true, `x` = 8	0	5
`pop(&y, &x)`	0 < 0? Hayır	`x = veri[0]`	true, `x` = 5	−1	(boş)
`pop(&y, &x)`	−1 < 0? **Evet**	alt taşma, `x`'e dokunulmaz	false	−1	(boş)

Tablo 6.4: `KAP` = 3 iken `push` / `pop` izlemesi.



Şekil 6.5: İzlemenin iki anı. Üstte üç push sonrası yığın doludur: `ust` = 2 = `KAP - 1`, boyut 3'tür; dördüncü push `false` döner. Altta dört pop sonrası yığın boştur: `ust` = −1 ve son pop `false` dönmüştür.

İzleme bir test programıyla doğrulandı. Her push'un dönüş değerini yazdıran program "push 5:1 8:1 2:1 9:0" çıktısını verdi; 9 eklenemedi. Pop'lar sırasıyla 2, 8, 5 döndürdü, dördüncü pop `false` oldu. Aradaki ara sorgular da beklendiği gibi çalıştı: üç push sonrası `yigit_boyut` 3, `yigit_dolu` doğru ve `yigit_ust` 2 verdi.

İki gerçeklemenin farkı taşma satırında görülür. Küresel dizi sürümü bu noktada programı bitirirdi. `bool` sürümü ise `false` döner ve program kaldığı yerden sürer. Örnekte 9'un eklenemediğini çağıran öğrenir. İsterse kullanıcıya bir uyarı basar, isterse 9'u başka bir yere koyar. Tek bir yordamın bütün programın kaderini belirlemesi böylece önlenir.

6.4.3 Arayüz ve gerçekleme

`struct` sürümü de hâlâ bir sorun taşır. `Yigit` yapısının alanları onu kullanan her koda açıktır. Kullanan bir programcı `y.ust = 5;` yazıp yığını bozabilir ya da `y.veri[0]` 'a doğrudan erişip yığın kuralını delebilir. Daha büyük bir sorun da şudur: yığının gerçeklemesi değiştiğinde, örneğin dizi bağlı listeye çevrildiğinde, alanlara doğrudan erişen bütün kodlar bozulur.

C'de bu sorun, kodu bir **başlık dosyası** (*header file*) ve bir kaynak dosyası olarak ikiye bölerek çözülür. Başlık dosyası **arayüzü** (*interface*), yani dışarıya açılan tipleri ve fonksiyon bildirimlerini taşır. Kaynak dosyası **gerçeklemeyi** (*implementation*), yani bu fonksiyonların gövdelerini ve yapının içini taşır.

```
/* yigit.h: kullanıcı yalnız bunu görür */
#ifndef YIGIT_H
#define YIGIT_H
#include <stdbool.h>
typedef struct Yigit Yigit;  /* opak */
Yigit *yigit_olustur(int kapasite);
void yigit_yok_et(Yigit *y);
bool yigit_ekle(Yigit *y, int x);
bool yigit_al(Yigit *y, int *x);
bool yigit_bos(const Yigit *y);
#endif
```

Başlık dosyasının ilk ve son satırları **çoklu ekleme korumasıdır** (*include guard*). Büyük bir programda aynı başlık bir kaynak dosyasına dolaylı yollardan iki kez eklenebilir. `#ifndef YIGIT_H` sınaması, `YIGIT_H` adı henüz tanımlı değilse dosyanın içeriğini işler ve hemen ardından bu adı tanımlar. İkinci eklemede ad tanımlı olduğu için içerik atlanır. Böylece aynı tipin iki kez tanımlanması hatası önlenir.

Asıl önemli satır `typedef struct Yigit Yigit;` satırıdır. Bu satır `struct Yigit` adında bir yapının var olduğunu söyler, ama alanlarını söylemez. Böyle yalnız adı bildirilen bir tipe **eksik tip** (*incomplete type*) denir. Alanları kullanıcıdan gizlenen bu tipe **opak tip** (*opaque type*) adı da verilir. Kullanan kod eksik bir tipin değişkenini tanımlayamaz, çünkü derleyici onun kaç bayt tuttuğunu bilmez. Alanlarına da erişemez. Yapabileceği tek şey, bu tipe bir işaretçi tutmak ve işaretçiyi başlıktaki fonksiyonlara vermektir. Yığın bu yüzden `yigit_olustur` ile oluşturulur ve `yigit_yok_et` ile yok edilir.

```
/* yigit.c: gerçekleme */
struct Yigit { int *veri; int ust, kap; };

Yigit *yigit_olustur(int kap) {
  Yigit *y = malloc(sizeof *y);
  if (y == NULL) return NULL;
  y->veri = malloc(kap * sizeof *y->veri);
  if (y->veri == NULL) {
    free(y);  return NULL;
  }
  y->ust = -1;  y->kap = kap;
  return y;
}
void yigit_yok_et(Yigit *y) {
  if (y) { free(y->veri); free(y); }
}
```

Kaynak dosyasında yapının tam tanımı vardır. Bu sürümde dizi yapının içinde sabit boyutlu değildir. `veri` bir işaretçidir ve dizinin kendisi `yigit_olustur` sırasında, istenen kapasiteyle heap bellekten ayrılır. Böylece her yığının kapasitesi ayrı ayrı seçilebilir.

`yigit_olustur` iki ayırma yapar. Önce yapının kendisi için `malloc(sizeof *y)` çağrılır. `sizeof *y` , `y` 'nin gösterdiği tipin, yani `struct Yigit` 'in boyutudur. Bu yazım, tip adı değişse bile doğru kalır. Ayırma başarısızsa `NULL` döner. Sonra `kap` öğelik dizi için `malloc(kap * sizeof *y->veri)` çağrılır. Bu ikinci ayırma başarısızsa önemli bir ayrıntı vardır: ilk ayrılan yapı bloğu `free(y)` ile iade edilir ve ancak ondan sonra `NULL` döner. Bu satır unutulsaydı yapının bloğu hiçbir değişkenin gösteremediği bir bellek parçası olarak kalırdı; yani bir **bellek sızıntısı** (*memory leak*) oluşurdu. İki ayırma da başarılıysa `ust` −1, `kap` istenen kapasite yapılır ve yapının adresi döner.

`yigit_yok_et` ayırma sırasının tersini izler. Önce dizi, sonra yapı iade edilir. Sıra önemlidir: önce yapı iade edilseydi `y->veri` alanı artık geçersiz bir bellekten okunurdu. `if (y)` sınaması fonksiyonun `NULL` ile de güvenle çağrılabilmesini sağlar. C standardına göre `free(NULL)` zaten hiçbir şey yapmaz; ama `y->veri` 'yi okumak için `y` 'nin geçerli olması gerekir.

Sunumdaki gerçekleme dosyası kısalık için başındaki `#include` satırlarını ve üç yordamı göstermez. Tam bir `yigit.c` şu satırlarla başlar: `#include <stdlib.h>` (`malloc` ve `free` için) ve `#include "yigit.h"` (bildirimlerle tanımların uyuştuğunu derleyicinin denetlemesi için). Eksik yordamlar, `struct` sürümündeki `push`, `pop` ve `yigit_bos`'un aynısıdır; tek fark, dolu denetiminin sabit `KAP` yerine yapının `kap` alanıyla yapılmasıdır: `if (y->ust == y->kap - 1) return false;`. Bu tamamlanmış hâliyle iki ayrı yığın (kapasite 2 ve 5) kuran bir test programı derlenip çalıştırıldı. Kapasitesi 2 olan yığına üçüncü ekleme `false` döndü; iki yığın da yok edildikten sonra sayaçlı bir `malloc` / `free` sarmalayıcısı dört ayırma ve dört iade saydı. Sızıntı yoktur.

Bu bölünmenin değeri, gerçekleme değiştiğinde görülür. `yigit.c` diziden bağlı listeye çevrilsin. `struct Yigit`'in içi tamamen değişir; `veri` ve `kap` alanları kalkar, yerine bir düğüm işaretçisi gelir. Ama `yigit.h` aynı kalır. Bu yüzden yığını kullanan bütün dosyalar hiç değiştirilmeden, yalnız yeniden bağlanarak çalışmaya devam eder. Soyut veri tipinin "ne" ile "nasıl"ı ayırması C'de böyle somutlaşır.

6.4.4 Yığınla dizgi ters çevirme

Yığının en basit uygulaması bir diziyi ters çevirmektir. Öğeler sırayla yığına itilir, sonra sırayla çekilir. Yığın son gireni ilk verdiği için çıkış sırası girişin tersidir. Aşağıdaki fonksiyon bir dizgiyi bu yolla yerinde ters çevirir; `struct` sürümünün `push` ve `pop` yordamlarını kullanır.

```
void ters_cevir(char *s) {
  Yigit y;  int c, i;
  yigit_baslat(&y);
  for (i = 0; s[i] != '\0'; i++)
    push(&y, s[i]);     /* KAP yeterli varsayıldı */
  for (i = 0; pop(&y, &c); i++)
    s[i] = (char)c;
}
```

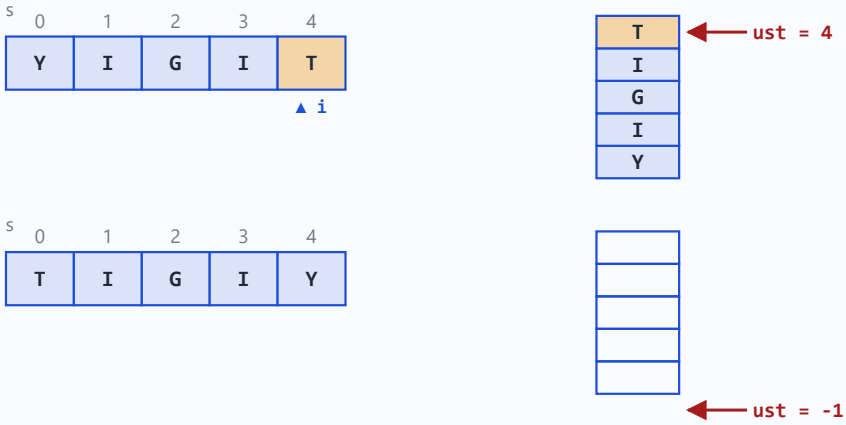
Fonksiyonun yerel bir `Yigit y` değişkeni vardır. Yığın, fonksiyonun çerçevesinde yaşar ve fonksiyon dönünce kendiliğinden yok olur; `malloc` ve `free` gerekmez. İlk döngü, dizginin sonundaki `'\0'` karakterine kadar her karakteri yığına iter. `push`'un dönüş değeri burada denetlenmez; yorumdaki varsayım, dizginin `KAP` = 100 karakterden kısa olduğudur. İkinci döngü, `pop` başarılı olduğu sürece çekilen karakteri dizginin `i`. konumuna yazar ve `i`'yi artırır. Yığın boşalınca `pop` `false` döner ve döngü biter.

`c` değişkeni `int`'tir, çünkü `pop` ikinci argüman olarak `int *` bekler. `push(&y, s[i])` çağrısında `char` değeri otomatik olarak `int`'e genişler. Dizgiye geri yazarken `(char)c` dönüşümü bu değeri yeniden `char`'a indirir. Karakterler zaten `char` aralığından geldiği için bilgi kaybolmaz.

Girdi `"YIGIT"` olsun. Tablo 6.5 iki döngünün her adımını verir.

Döngü	`i`	İşlem	Yığın (alttan üste)	`s`
başlangıç	—	`yigit_baslat`	(boş)	YIGIT
1	0	push('Y')	Y	YIGIT
1	1	push('I')	Y I	YIGIT
1	2	push('G')	Y I G	YIGIT
1	3	push('I')	Y I G I	YIGIT
1	4	push('T')	Y I G I T	YIGIT
2	0	pop → 'T', `s[0] = 'T'`	Y I G I	TIGIT
2	1	pop → 'I', `s[1] = 'I'`	Y I G	TIGIT
2	2	pop → 'G', `s[2] = 'G'`	Y I	TIGIT
2	3	pop → 'I', `s[3] = 'I'`	Y	TIGIT
2	4	pop → 'Y', `s[4] = 'Y'`	(boş)	TIGIY
2	5	pop → `false`, döngü biter	(boş)	TIGIY

Tablo 6.5: `ters_cevir` izlemesi, girdi `"YIGIT"`.



Şekil 6.6: Ters çevirmenin iki anı. Üstte birinci döngü bitmiştir: beş karakter yığındadır, üstte T vardır (`ust` = 4). Altta ikinci döngü bitmiştir: yığın boştur (`ust` = −1) ve dizgi TIGIY olmuştur.

Tablodaki ara dizgiler ilk bakışta şaşırtabilir. Üç adım boyunca `s` "TIGIT" olarak görünür. Bunun nedeni, "YIGIT" kelimesinin ortadaki üç harfinin (I, G, I) kendi içinde simetrik olmasıdır. Yazılan harfler zaten orada bulunan harflerle aynıdır. Yalnız ilk ve son adım dizgiyi gerçekten değiştirir. Program çalıştırılarak doğrulandı: `"YIGIT"` için TIGIY, `"ab"` için ba verdi; boş dizgi boş kaldı.

Maliyet açıktır. *n* karakterlik bir dizgide birinci döngü *n* push, ikinci döngü *n* pop ve bir başarısız pop yapar. Her işlem $O(1)$ olduğundan toplam süre $\Theta(n)$'dir. Yığın *n* karakteri birden tuttuğu için ek bellek de $\Theta(n)$'dir. Aynı iş iki indisle, baştan ve sondan karakter takas ederek $O(1)$ ek bellekle de yapılabilir. Yığınlı sürümün değeri, yığının sırayı tersine çevirme yeteneğini açıkça göstermesidir. Aynı yöntem, sırası tersine çevrilmek istenen ama uzunluğu baştan bilinmeyen her akış için de çalışır.

> **Sık yapılan hata**
>
> Kapasite varsayımını unutmak. `ters_cevir`, dizginin `KAP` karakterden kısa olduğunu varsayar. 150 karakterlik bir dizgide ilk 100 karakter yığına girer, kalan 50 karakter için push `false` döner ve bu karakterler sessizce kaybolur. İkinci döngü yalnız 100 karakteri geri yazar. Sonuç, ilk 100 karakterin tersiyle kalan 50 karakterin özgün hâlinden oluşan anlamsız bir dizgidir. Güvenli bir sürüm ya push sonucunu denetleyip hata döndürür ya da kapasiteyi dizginin uzunluğuna göre seçer.

6.5 Bağlı yığın

Dizi yığınının kapasitesi baştan seçilir. Kapasite küçük seçilirse yığın taşar; büyük seçilirse kullanılmayan hücreler boşa yer kaplar. Bağlı liste bu ikilemi ortadan kaldırır. Her öğe için ayrı bir düğüm ayrılır ve yığın tam olarak öğe sayısı kadar düğüm tutar. Bu kesim yığını tek yönlü bir bağlı listeyle kurar.

6.5.1 Üst neden liste başıdır

Bağlı listede her öğe bir **düğüm**dür (*node*). Düğüm iki alan taşır: öğenin değerini tutan `deger` ve bir sonraki düğümün adresini tutan `bag`. Düğüm tipi kendi tipinden bir işaretçi içerdiği için kendine başvuran bir yapıdır:

```
typedef struct dugum {
    int deger;
    struct dugum *bag;
} dugum;
```

Yığının hangi ucu listenin hangi ucuna denk gelmelidir? Tek yönlü bir listede iki uç eşit değildir. Liste başına ekleme ve liste başından silme $O(1)$'dir: yalnız baş işaretçisi ve bir bağ değişir. Liste sonuna ekleme de son düğüm ayrıca tutulursa $O(1)$'dir. Ama liste sonundan silmek için sondan bir önceki düğümün bağını `NULL` yapmak gerekir. Tek yönlü listede bir düğümden öncekine geri dönülemez; sondan bir önceki düğüm ancak baştan yürüyerek, $O(n)$ sürede bulunur.

Yığında ekleme ve silme aynı uçtan yapıldığı için bu ucun liste başı olması gerekir. Yığının üstünü gösteren `top` işaretçisi bu yüzden listenin ilk düğümünü gösterir. Son eklenen öğe listenin başında, ilk eklenen öğe listenin sonundadır. Boş yığın, `top`'u `NULL` olan yığındır. Özetle iki seçenek vardır: ya iki işaretçi tutulup son düğüm de izlenir ve yine de silme yavaş kalır, ya da tek işaretçiyle baş tutulur ve her şey hızlı olur. Doğru seçim ikincisidir.



Şekil 6.7: A, B, C sırasıyla eklenmiş bağlı yığın. `top` liste başını, yani son eklenen C'yi gösterir; C B'yi, B A'yı gösterir. A'nın bağı `NULL`'dır: A yığının altıdır.

Şekil 6.7 A, B, C sırasıyla eklenmiş bir bağlı yığını gösterir. Okların yönüne dikkat etmek gerekir. Oklar üstten alta doğrudur: her düğüm kendinden önce eklenmiş düğümü gösterir. Bir pop, `top`'u bir ok ileri kaydırır ve C'yi listeden çıkarır. Kaydırma yoktur, kapasite sınırı yoktur. Yığın ancak `malloc` yeni bir düğüm için yer bulamadığında taşar.

6.5.2 Bağlı `push`

```
bool push(dugum **top, int x) {
  dugum *yeni = malloc(sizeof *yeni);
  if (yeni == NULL) return false;  /* bellek yok */
  yeni->deger = x;
  yeni->bag = *top;      /* eski üstü göster */
  *top = yeni;           /* yeni üst */
  return true;
}
```

Fonksiyon `dugum **top` parametresi alır. Bu bir **işaretçiye işaretçidir**: çağırandaki `top` değişkeninin adresidir. Push, çağırandaki `top`'u yeni düğüme çevirecektir. Değerle aktarılan bir kopya değişseydi çağıranın `top`'u eski düğümü göstermeye devam ederdi ve yeni düğüm kaybolurdu. Bu yüzden çağrı `push(&top, 10)` biçiminde yapılır. Fonksiyonun içinde `*top`, çağırandaki `top` değişkeninin kendisidir.

Gövde dört adımdan oluşur:

1. `dugum *yeni = malloc(sizeof *yeni);` heap bellekten bir düğümlük yer ayırır. Ayırma başarısızsa `malloc` `NULL` döner ve fonksiyon `false` ile çıkar. Yığın bu durumda hiç değişmemiştir.
2. `yeni->deger = x;` değeri yeni düğüme yazar. `malloc`'un verdiği bellek temizlenmemiştir; alanlar bu atamaya kadar belirsiz değerler taşır.
3. `yeni->bag = *top;` yeni düğümün bağını eski üste çevirir. Yığın boşsa eski üst `NULL`'dır ve yeni düğümün bağı da `NULL` olur.
4. `*top = yeni;` çağırandaki `top`'u yeni düğüme çevirir. Yeni düğüm artık yığının üstüdür.

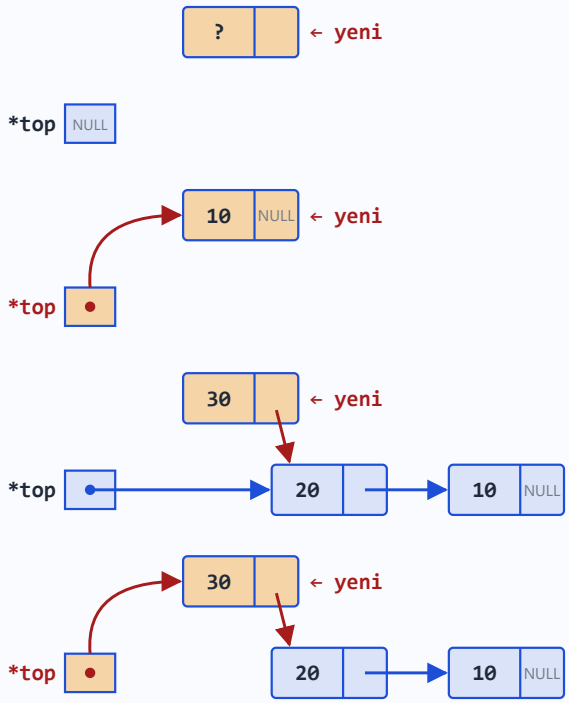
Üçüncü ve dördüncü adımın sırası kritiktir. Önce bağlanır, sonra üst değiştirilir. Sıra ters olsaydı `*top = yeni;` satırı eski üstün adresini kaybettirirdi. Ardından gelen `yeni->bag = *top;` yeni düğümü kendisine bağlardı. Eski listenin bütün düğümleri hiçbir değişkenin gösteremediği bir hâlde, sızmış bellek olarak kalırdı.

Boş bir yığına sırasıyla 10, 20, 30 itilsin. Tablo 6.6 her satırdan sonraki durumu verir. Tabloda `düğüm(10)`, değeri 10 olan düğümün adresi anlamına gelir.

Çağrı	Satır	`yeni`	`yeni->bag`	`*top`	Yığın
başlangıç	`dugum *top = NULL;`	—	—	NULL	(boş)
`push(&top, 10)`	`malloc`	düğüm(?)	?	NULL	(boş)
	`yeni->deger = x`	düğüm(10)	?	NULL	(boş)
	`yeni->bag = *top`	düğüm(10)	NULL	NULL	(boş)
	`*top = yeni`	düğüm(10)	NULL	düğüm(10)	10

Çağrı	Satır	`yeni`	`yeni->bag`	`*top`	Yığın
`push(&top, 20)`	`yeni->bag = *top`	düğüm(20)	düğüm(10)	düğüm(10)	10
	`*top = yeni`	düğüm(20)	düğüm(10)	düğüm(20)	20 → 10
`push(&top, 30)`	`yeni->bag = *top`	düğüm(30)	düğüm(20)	düğüm(20)	20 → 10
	`*top = yeni`	düğüm(30)	düğüm(20)	düğüm(30)	30 → 20 → 10

Tablo 6.6: Boş yığına 10, 20, 30 itilirken bağlı `push` izlemesi.



Şekil 6.8: Bağlı push'un dört anı. Birincide `malloc` ilk düğümü ayırmıştır; alanları belirsizdir (?) ve `*top` hâlâ `NULL`'dır. İkincide 10 yığının tek öğesidir. Üçüncüde 30'luk yeni düğüm hazırlanmış ve eski üst 20'ye bağlanmıştır, ama `*top` henüz 20'yi gösterir. Dördüncüde `*top = yeni` ile yığın 30 → 20 → 10 olmuştur.

Üçüncü çizimdeki ara durum öğreticidir. Yeni düğüm hazırdır ve 20'yi gösterir, ama yığın hâlâ 20 → 10'dur. Yığının yapısı tek bir atamayla, `*top = yeni` satırında değişir. Bu düzen hata durumunda da güvenlidir: `malloc` başarısız olsaydı fonksiyon ilk denetimde dönerdi ve yığına hiç dokunulmamış olurdu. Her push sabit sayıda iş yapar: bir ayırma, üç atama. Yığının uzunluğu bu sayıyı etkilemez; push *O(1)*'dir. `malloc`'un kendi süresi de pratikte sabit kabul edilir, ama bir dizi hücresine yazmaktan çok daha pahalıdır.

6.5.3 Bağlı `pop`

```
bool pop(dugum **top, int *x) {
  dugum *eski = *top;      /* malloc yok! */
  if (eski == NULL) return false;  /* bos */
  *x = eski->deger;
  *top = eski->bag;        /* üstü ilerlet */
  free(eski);              /* belleği bırak */
  return true;
}
```

Pop da `dugum **top` alır, çünkü çağırandaki `top`'u bir sonraki düğüme ilerletecektir. Çıkarılan değer `int *x` çıktı parametresiyle verilir; dönüş değeri başarıyı bildirir. Gövde şu adımlardan oluşur:

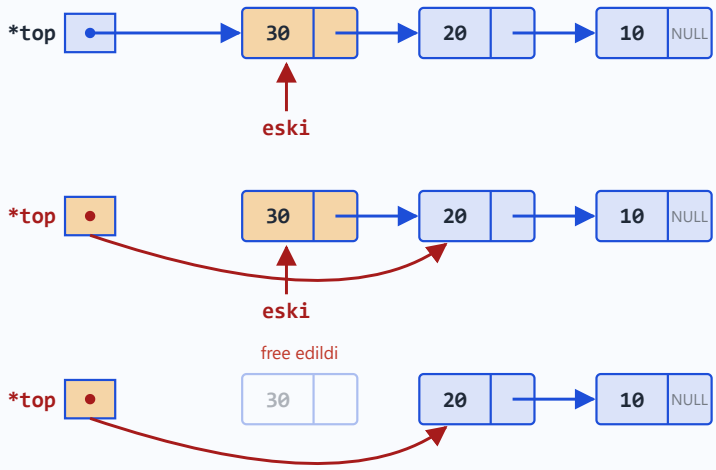
1. `dugum *eski = *top;` üst düğümün adresini yerel bir işaretçide saklar. Bu satır yeni bellek ayırmaz; yalnız bir adresi kopyalar.

2. `if (eski == NULL) return false;` yığın boşsa alt taşmayı bildirir.
3. `*x = eski->deger;` değeri çağırana kopyalar. Bu satır düğüm iade edilmeden önce çalışmalıdır.
4. `*top = eski->bag;` çağırandaki `top`'u bir sonraki düğüme ilerletir. Bu da iadeden önce yapılmalıdır, çünkü bağ alanı iade edilecek düğümün içindedir.
5. `free(eski);` düğümün belleğini sisteme iade eder.

30 → 20 → 10 yığınına dört kez pop uygulansın ve her başarılı sonuç yazdırılsın. Tablo 6.7 çağrıları özetler.

Çağrı	`eski`	`*x`	`*top` (sonra)	`free`	Döner	Yığın
1. pop	düğüm(30)	30	düğüm(20)	düğüm(30)	true	20 → 10
2. pop	düğüm(20)	20	düğüm(10)	düğüm(20)	true	10
3. pop	düğüm(10)	10	NULL	düğüm(10)	true	(boş)
4. pop	NULL	(dokunulmaz)	NULL	—	false	(boş)

Tablo 6.7: 30 → 20 → 10 yığınından dört pop.



Şekil 6.9: İlk pop'un üç anı. Önce `eski` üst düğümü (30) gösterir; yeni bellek ayrılmamıştır. Sonra `*top = eski->bag` ile üst 20'ye ilerler; 30'luk düğüm artık listede değildir ama `eski` onu hâlâ gösterir. En sonda `free(eski)` ile düğüm iade edilmiştir; `main` 30 yazar.

Test programı 10, 20, 30 itip dört kez pop çağırdı ve "30 20 10 false" yazdı. Çıkış sırası girişin tersidir. Pop da sabit sayıda iş yapar: bir karşılaştırma, iki atama ve bir `free`. Maliyeti *O(1)*'dir.

Pop'taki adımların sırası, push'takinden de önemlidir. `free(eski)` çağrısından sonra `eski`'nin gösterdiği bellek artık programa ait değildir. `eski->deger` ya da `eski->bag` bu andan sonra okunursa davranış tanımsızdır. Program çoğu zaman doğru değeri okur, çünkü iade edilen bellek hemen başka bir işe verilmez. Bu da hatayı daha tehlikeli yapar: testlerde görünmez, ama başka bir `malloc` araya girdiğinde değer bozulur. Böyle iade edilmiş bir belleği gösteren işaretçiye **sarkan işaretçi** (*dangling pointer*) denir. Sağlam kural şudur: önce değeri ve bağı kurtar, en son `free` çağır.

6.5.4 Bellek sorumluluğu ve yığını boşaltma

Bağlı yığında her düğümü `malloc` ayırır ve her düğümü bir `free` geri vermelidir. C'de bu sorumluluk tamamen programcıdadır. Program bitmeden yığında kalan düğümler de iade edilmelidir. Bu iş için bir temizleme yordamı yazılır:

```c
typedef struct dugum {
    int deger;
    struct dugum *bag;
} dugum;

void yigit_temizle(dugum **top) {
    while (*top != NULL) {
        dugum *eski = *top;
        *top = eski->bag;
        free(eski);
    }
}                /* sonunda *top == NULL */
```

`yigit_temizle`, yığın boşalana kadar üst düğümü çıkarır. Döngünün her turunda üç satır çalışır: üst düğümün adresi `eski`'ye alınır, `*top` bir sonraki düğüme ilerletilir, sonra `eski` iade edilir. Sıra yine aynıdır: bağ kurtarılmadan düğüm bırakılmaz. Döngü bittiğinde `*top` `NULL`'dır; çağırandaki `top` böylece boş yığını gösterir ve sarkan bir işaretçi olarak kalmaz. *n* düğümlü bir yığında döngü *n* tur döner; maliyet $\Theta(n)$'dir. Ek bellek yalnız `eski` işaretçisidir, yani $O(1)$.

Bu kodda dikkat çeken bir nokta, `eski` değişkeninin bir işaretçi değişkeni olmasıdır, yeni bir düğüm değil. Öğrenci kodlarında sık görülen bir yanlış, her işaretçiye `malloc` ile yer ayırmaktır: `dugum *eski = malloc(sizeof *eski); eski = *top;`. İlk satır bir düğümlük yer ayırır; ikinci satır `eski`'nin üzerine üst düğümün adresini yazar. Ayrılan bloğun adresi artık hiçbir yerde yoktur. Her pop'ta bir düğüm sızar. İşaretçiye `malloc` ancak işaretçinin göstereceği yeni bir nesne gerektiğinde yapılır. Var olan bir düğümü göstermek için yalnız atama yeter.

Bağlı yığının sızıntısız çalıştığı sayaçlı bir sarmalayıcıyla doğrulandı. Sarmalayıcı, `malloc` ve `free` çağrılarını bir makro yardımıyla kendi fonksiyonlarına yönlendirir ve her başarılı çağrıyı sayar. Test programı 10, 20, 30'u itip dört kez pop yaptı, sonra 1'den 5'e beş öğe daha itip `yigit_temizle` çağırdı. Sayaç sekiz `malloc` ve sekiz `free` gösterdi; temizlemeden sonra `top` `NULL` oldu. Aynı programda `sizeof(dugum)` 16 çıktı. Düğümün alanları 4 baytlık `int` ve 8 baytlık işaretçidir; işaretçinin 8'in katı bir adreste durması için derleyici araya 4 baytlık bir dolgu (*padding*) koyar. Bu yüzden 64 bitlik bir sistemde bağlı yığın, her `int` için dizinin dört katı bellek kullanır; `malloc`'un her blok için tuttuğu kendi kayıtları da buna eklenir.

> **Sık yapılan hata**
>
> Push'ta önce üstü değiştirmek: `*top = yeni; yeni->bag = *top;`. İkinci satır çalıştığında `*top` zaten `yeni`'dir; düğüm kendini gösterir ve eski yığının bütün düğümleri kaybolur. Yığının üstüne bakan her döngü bundan sonra aynı düğümde sonsuza kadar döner. Doğru sıra: önce `yeni->bag = *top;`, sonra `*top = yeni;`.

6.6 Dizi tabanlı mı, bağlı mı?

Şu ana kadar iki gerçekleme kuruldu: sabit kapasiteli dizi ve bağlı liste. Arada üçüncü bir seçenek daha vardır: dolduğunda kendini büyüten dizi. Bu kesim önce bu üçüncü gerçeklemeyi kurar, sonra maliyetini amortize analizle gerekçelendirir ve üç gerçeklemeyi karşılaştırır.

6.6.1 Büyüyen dizi

Sabit dizinin sorunu kapasitenin baştan bilinmesi gerekmesidir. Çözüm, diziyi heap bellekte tutmak ve dolduğunda `realloc` ile daha büyük bir bloğa taşımaktır. Dolduğunda `realloc` ile büyütülen diziye **dinamik dizi** (*dynamic array*) denir. Asıl soru, ne kadar büyütüleceğidir. Her dolmada kapasite iki katına çıkarılırsa push'ların ortalama maliyeti sabit kalır. Aşağıdaki gerçekleme `struct` sürümünün düzenini izler:

```c
typedef struct {
    int *veri;
    int ust, kap;           /* boşken ust = -1 */
} DYigit;

bool dy_baslat(DYigit *y, int kap) {
    y->veri = malloc(kap * sizeof *y->veri);
    if (y->veri == NULL) return false;
    y->ust = -1;  y->kap = kap;
    return true;
}
bool dy_push(DYigit *y, int x) {
    if (y->ust == y->kap - 1) {          /* dolu: kapasiteyi ikiye katla */
        int *yeni = realloc(y->veri, 2 * y->kap * sizeof *yeni);
        if (yeni == NULL) return false;  /* eski blok hâlâ geçerli */
        y->veri = yeni;
        y->kap *= 2;
    }
    y->veri[++y->ust] = x;
    return true;
}
bool dy_pop(DYigit *y, int *x) {
    if (y->ust < 0) return false;
    *x = y->veri[y->ust--];
    return true;
}
void dy_yok_et(DYigit *y) {
    free(y->veri);
    y->veri = NULL;  y->ust = -1;  y->kap = 0;
}
```

`DYigit` yapısı üç alan taşır: heap bellekteki dizinin adresi `veri`, üst indisi `ust` ve o anki kapasite `kap`. `dy_baslat` istenen başlangıç kapasitesiyle diziyi ayırır. `dy_pop`, `struct` sürümündeki `pop`'un aynısıdır.

Fark `dy_push`'tadır. Yığın doluysa yordam hata döndürmez; kapasiteyi iki katına çıkarmayı dener. `realloc(y->veri, yeni_boyut)`, eski bloğu yeni boyuta getirir. Eski bloğun hemen arkasında yer varsa blok yerinde büyüyebilir. Yoksa `realloc` yeni bir blok ayırır, eski içeriği oraya kopyalar, eski bloğu iade eder ve yeni bloğun adresini döndürür. Kopyalanan öğe sayısı o anki kapasitedir.

`realloc`'un sonucu doğrudan `y->veri`'ye yazılmaz; önce `yeni` adlı geçici bir işaretçiye alınır. Bunun nedeni hata durumudur. `realloc` başarısız olursa `NULL` döner ve eski bloğa dokunmaz. Sonuç doğrudan `y->veri`'ye yazılsaydı eski bloğun adresi kaybolur, içindeki bütün öğeler sızardı. Geçici işaretçi sayesinde başarısızlıkta yığın eski hâliyle geçerli kalır ve fonksiyon `false` döner. Başarılıysa `veri` ve `kap` güncellenir, ardından öğe her zamanki gibi yazılır.

Başlangıç kapasitesi 2 olan bir yığına 10, 20, 30, 40, 50 itildiğinde kapasite şöyle değişir: 10 ve 20 yer bulur (`kap` = 2). 30 itilirken yığın doludur; kapasite 4'e çıkar, 2 öğe kopyalanır. 40 yer bulur. 50 itilirken kapasite 8'e çıkar, 4 öğe kopyalanır. Bu değerler test programında yazdırılarak doğrulandı: her push'tan sonra (`ust`, `kap`) çiftleri (0, 2), (1, 2), (2, 4), (3, 4), (4, 8) oldu. Sayaçlı sarmalayıcı iki `malloc` ve iki `free` saydı.

6.6.2 Amortize maliyet

Büyüyen dizide tek bir push'un maliyeti duruma göre çok değişir. Çoğu push bir hücreye yazar ve *O(1)*'dir. Ama kapasitenin dolduğu anda yapılan push bütün öğeleri kopyalar ve *O(n)*'dir. En kötü durum analizi bu yüzden "push *O(n)*" der. Bu doğru, ama yanıltıcıdır: pahalı push'lar çok seyrektir. Bir işlem dizisinin toplam maliyetini işlem sayısına bölerek işlem başına ortalama maliyeti bulmaya **amortize analiz** (*amortized analysis*) denir. Buradaki ortalama olasılığa dayanmaz; en kötü işlem dizisi için bile geçerlidir.

Teorem 6.1 (İki katına büyütmenin amortize maliyeti)

Kapasitesi 1 olan boş bir büyüyen dizi yığınına art arda n push yapılsın ve her dolmada kapasite iki katına çıkarılsın. Bütün büyütmelerde kopyalanan toplam öğe sayısı $2n$'den küçüktür. Bu yüzden n push'un toplam maliyeti $O(n)$, push başına amortize maliyet $O(1)$'dir.

Kanıt. Büyütme, yığın dolu iken push yapıldığında olur. Kapasite 1, 2, 4, 8, … diye ilerlediği için büyütmeler kapasite 1, 2, 4, …, 2^k iken yapılır. Kapasite c iken yapılan büyütme c öğe kopyalar. Son büyütme kapasite 2^k iken olmuşsa, bu anda yığında 2^k öğe vardır ve bir öğe daha itilir; dolayısıyla $2^k < n$'dir. Toplam kopya sayısı $1 + 2 + 4 + \dots + 2^k = 2^{k+1} - 1$'dir. $2^k \le n - 1$ olduğundan bu toplam en çok $2n - 3$'tür; n'nin bir sabit katıdır. Her push ayrıca bir yazma yapar. Toplam iş $n + (2n - 3) < 3n$ ile sınırlıdır ve push başına 3 birimden azdır. ■

Kanıttaki kopya sınırı daha sıkı da verilebilir. Kopya toplamı $2^{k+1} - 1$ ve son kapasite 2^{k+1}'dir; yani toplam kopya, son kapasiteden bir eksiktir. Son kapasite de $2n$'den küçüktür. Bu sayılar bir test programıyla doğrulandı: kapasitesi 1 olan yığına 1000 push yapıldığında 10 büyütme oldu, toplam 1023 öğe kopyalandı ($2n$ = 2000'den az) ve son kapasite 1024'e ulaştı.

Büyütme oranının iki olması şart değildir; önemli olan oranın 1'den büyük sabit bir çarpan olmasıdır. Çarpan 1,5 seçilirse kopyalar biraz artar, boşa duran kapasite azalır; amortize maliyet yine $O(1)$ kalır. Buna karşılık kapasite her dolmada sabit bir miktar, örneğin 10 hücre artırılırsa amortize maliyet bozulur. n push boyunca yaklaşık $n/10$ büyütme olur ve her biri o anki bütün öğeleri kopyalar. Toplam kopya $10 + 20 + \dots + n \approx n^2/20$'dir; push başına maliyet $\Theta(n)$'ye çıkar.

> **Sık yapılan hata**
>
> `y->veri = realloc(y->veri, …);` yazmak. `realloc` başarısız olursa `NULL` döner ve bu atama eski bloğun tek adresini siler. Hem yığının bütün içeriği kaybolur hem de eski blok sızar. Sonuç önce geçici bir işaretçiye alınmalı ve `NULL` olmadığı görüldükten sonra asıl alana yazılmalıdır.

6.6.3 Karşılaştırma ve ölçüm

Tablo 6.8 üç gerçeklemeyi altı ölçüte göre karşılaştırır.

Ölçüt	Sabit dizi	Büyüyen dizi	Bağlı liste
Kapasite	Derleme anında sabit	Gerektikçe ×2	Bellek bitene kadar
push / pop	$O(1)$	$O(1)$ amortize	$O(1)$, her push'ta `malloc`
Öğe başına ek bellek	Yok	Yok (boş kapasite hariç)	Bir işaretçi (8 bayt) ve dolgu
Önbellek	Çok iyi	Çok iyi	Zayıf (düğümler dağınık)
Taşma	Mümkün	Yalnız `realloc` başarısızsa	Yalnız `malloc` başarısızsa
Hata riski	Sınır denetimi	`realloc` sonucu	`free` unutmak, sarkan işaretçi

Tablo 6.8: Sabit dizi, büyüyen dizi ve bağlı liste ile yığın.

Tablodaki "önbellek" satırı açıklama ister. İşlemci belleğe her erişimde tek bir sayıyı değil, çevresindeki 64 baytlık bir bloğu **önbelleğe** (*cache*) getirir. Dizi yığınında ardışık öğeler yan yana durduğu için bir bloğa birçok öğe sığar ve sonraki erişimler önbellekten karşılanır. Bağlı yığında her düğüm ayrı bir `malloc` bloğudur ve bellekte nerede duracağı belli değildir. Ardışık düğümlere erişim çoğu zaman yeni bir bloğun getirilmesini gerektirir.

Asimptotik maliyetler aynıdır, ama sabit çarpanlar çok farklıdır. Bunu görmek için üç gerçeklemede 10^7 push ve ardından 10^7 pop yapan bir program `-O2` ile derlenip üç kez çalıştırıldı. Sabit dizi 25–35 ms, büyüyen dizi (başlangıç kapasitesi 16) 30–63 ms, bağlı yığın 680–806 ms sürdü. Bu değerler makineye göre değişir; ama oran tutarlıdır: bağlı yığın, dizi yığınlarından yaklaşık yirmi kat yavaştır. Farkın büyük bölümü her push'taki `malloc` ve her pop'taki `free` çağrısından, kalanı önbellek kayıplarından gelir.

Seçim için pratik bir kural şöyledir. Yığının alabileceği en büyük boyut biliniyorsa sabit dizi en yalın ve en hızlı çözümdür. Bilinmiyorsa büyüyen dizi pratikte en hızlısıdır. Bağlı yığın, düğümlerin zaten başka yapılarda yaşadığı ya da öğelerin tek tek yaratılıp silindiği durumlar için uygundur. Örneğin her öğesi büyük bir kayıt olan ve kayıtları başka listelerle paylaşan bir programda düğümleri yığına bağlamak, onları bir diziye kopyalamaktan ucuzdur.

6.7 Çağrı yığını

Bilgisayarın en çok kullanılan yığını programcının yazdığı bir yığın değildir. Her C programı, fonksiyon çağrılarını yönetmek için derleyicinin ve işletim sisteminin kurduğu bir yığın kullanır. Buna **çağrı yığını** (*call stack*) denir. Bu kesim çağrı yığınının nasıl çalıştığını, nasıl taştığını ve özyinelemeli bir fonksiyonun bu yığın yerine kendi yığınımızla nasıl yazılabileceğini anlatır.

6.7.1 Çerçeve

Bir fonksiyon çağrıldığında, çağrının yaşaması için gereken bilgiler bellekte bir bloğa yazılır. Bu bloğa **çerçeve** (*stack frame, activation record*) denir. Çerçevenin ilk parçası **dönüş adresidir** (*return address*): fonksiyon bitince çağırandaki hangi komuttan devam edileceğini gösterir. Çerçeve ayrıca fonksiyonun parametrelerini, yerel değişkenlerini ve çağıranın çerçevesine geri dönmek için gereken bilgiyi tutar. Çağrı, yeni bir çerçeveyi çağrı yığınına iter. `return` bu çerçeveyi çeker ve denetim dönüş adresine geçer. En son çağrılan fonksiyon ilk biter; çağrı yığını bu yüzden gerçek bir yığındır.

Özyinelemeli bir fonksiyon bu düzeni açıkça gösterir:

```
long fakt(int n) {
    if (n <= 1) return 1;          /* taban durum */
    return n * fakt(n - 1);        /* çağrı dönünce çarpma yapılır */
}
```

`main`, `fakt(5)`'i çağırsın. `fakt(5)`, sonucunu hesaplamak için `fakt(4)`'ün sonucunu bekler; `fakt(4)` `fakt(3)`'ü, o da `fakt(2)`'yi, o da `fakt(1)`'i çağırır. Her çağrının kendi `n`'si vardır ve bu değerler birbirine karışmaz, çünkü her biri ayrı bir çerçevededir. `fakt(1)` taban duruma ulaşır ve 1 döndürür. O anda yığında en çok çerçeve vardır: `main` ile birlikte altı çerçeve. Sonra çerçeveler ters sırada çekilir: `fakt(2)` 2·1 = 2, `fakt(3)` 3·2 = 6, `fakt(4)` 4·6 = 24, `fakt(5)` 5·24 = 120 döndürür. Tablo 6.9 bu süreci özetler.

An	Çağrı yığını	Çerçeve sayısı
`main` başlar	main	1
`fakt(5)` çağrılır	main | fakt(5)	2
`fakt(4)` çağrılır	main | fakt(5) | fakt(4)	3
`fakt(1)` çağrılır	main | fakt(5) | fakt(4) | fakt(3) | fakt(2) | fakt(1)	6 (en çok)
`fakt(1)` 1 döndürür	main | fakt(5) | fakt(4) | fakt(3) | fakt(2)	5
`fakt(2)` 2 döndürür	main | fakt(5) | fakt(4) | fakt(3)	4
`fakt(5)` 120 döndürür	main	1

Tablo 6.9: `fakt(5)` sırasında çağrı yığını (sağdaki çerçeve üsttedir).

Çerçevelerin bellekte gerçekten üst üste dizildiği bir deneyle görülebilir. `fakt`'ın her çağrısında `n` parametresinin adresi yazdırıldı (`-O0`, Windows x86-64). Her çağrıda adres bir öncekinden 96 bayt küçüktü: `main`'deki bir yerel değişkenden `fakt(5)`'in `n`'sine 80 bayt, `fakt(1)`'in `n`'sine 464 bayt vardı. Bu sayılar derleyiciye, eniyileme düzeyine ve işlemciye göre değişir. Değişmeyen gözlem şudur: x86 işlemcilerde çağrı yığını yüksek adreslerden düşük adreslere doğru büyür ve her çağrı sabit büyüklükte bir çerçeve ekler. `fakt(n)` bu yüzden $\Theta(n)$ çerçeve, yani $\Theta(n)$ yığın belleği kullanır.

6.7.2 Çağrı yığını nasıl taşar

Çağrı yığınının kapasitesi sınırlıdır ve programın başında sabitlenir. Tipik değerler birkaç megabayttır: Linux'ta 8 MB, Windows'ta Microsoft derleyicisiyle 1 MB. Bu çalışmadaki `zig cc` derleyicisi Windows programlarına 16 MB ayırıyordu. Yığın dolunca program **yığın taşması** (*stack overflow*) hatasıyla çöker. Aşağıda iki tipik neden ve aynı yığın düzeninden doğan bir hata verilmiştir.

Sonsuz özyineleme
: Taban durumu olmayan ya da taban durumuna hiç ulaşmayan bir özyineleme her çağrıda bir çerçeve iter ve hiçbirini çekmez. Örneğin `fakt`'taki koşul `n == 1` yazılsaydı `fakt(0)` çağrısı *−1*, *−2*, … diye sonsuza dek inerdi. Denemede her çağrısında 64 baytlık bir yerel dizi tutan sonsuz bir özyineleme yaklaşık 104.000 çağrı derinliğinde Windows'un `STATUS_STACK_OVERFLOW` (0xC00000FD) hatasıyla durdu.

Büyük yerel diziler
: `int a[1000000];` gibi bir yerel dizi çerçevenin içindedir ve tek başına 4 MB tutar. 1 MB'lık bir yığını tek başına taşırır. Denemede 16 MB'lık yığında bu dizi sığdı, 5 milyon öğelik (20 MB) dizi ise programı aynı hatayla çökertti. Büyük veriler `malloc` ile heap bellekte tutulur ya da `static` tanımlanır.

Yerel adres döndürme
: `return &yerel;` yazmak, taşma değildir ama aynı yığın düzeninden doğan bir hatadır. Fonksiyon dönünce çerçevesi çekilir. O bellek bölgesi bir sonraki çağrının çerçevesi için yeniden kullanılır. Döndürülen adres artık başka bir çağrının yerel değişkenlerini gösterir; bir sarkan işaretçidir.

Yığın taşması bu dersin konusu olan her özyinelemeli yordamda göz önünde tutulmalıdır. Örneğin bağlı bir listeyi özyinelemeli olarak serbest bırakan bir fonksiyon, her düğüm için bir çerçeve iter. Bir milyon düğümlü bir listede bu, bir milyon çerçeve demektir ve yığını taşırabilir.

6.7.3 Özyinelemeyi döngüye çevirmek

Çağrı yığını bir yığın olduğuna göre, her özyinelemeli algoritma çağrı yığınının yaptığı işi kendi yığınımızla yaparak bir döngüye çevrilebilir. Özyinelemeli bir çağrı, "şimdilik yarım bırakılan işi" bir çerçeveye koyar. Döngülü sürümde bu yarım iş açık bir yığına itilir. Faktöriyel için yarım kalan iş, "dönüşte `n` ile çarp" bilgisidir:

```
long fakt_dongu(int n) {
    Yigit y;  int k;  long sonuc = 1;
    yigit_baslat(&y);
    while (n > 1)                  /* "çağrıları" kendi yığınına it */
        push(&y, n--);
    while (pop(&y, &k))            /* "dönüşlerde" çarp */
        sonuc *= k;
    return sonuc;
}
```

Birinci döngü, özyinelemeli sürümün aşağı inen çağrılarını taklit eder: `n`'yi 1'e inene kadar yığına iter. 5 için yığın alttan üste 5, 4, 3, 2 olur. İkinci döngü dönüşleri taklit eder: yığından çekilen her değerle sonucu çarpar. Çarpımlar 2, 3, 4, 5 sırasıyla yapılır; özyinelemeli sürümdeki dönüş sırasının aynısıdır. Sonuç 120'dir. `fakt_dongu(0)` ve `fakt_dongu(1)` 1, `fakt_dongu(10)` 3628800 verdi; özyinelemeli `fakt` ile aynıdır.

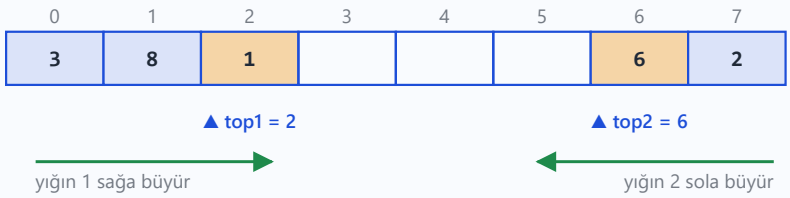
Faktöriyelde bu çeviri gereksizdir; basit bir `for` döngüsü yeter. Ama ağaç gezinme, derinlik öncelikli arama ya da geri izleme gibi algoritmalarda yarım kalan iş tek bir sayı değildir. Yarım kalan iş bir durumdur ve yığına bu durum itilir. Kendi yığınımızı kullanmanın iki yararı vardır. Birincisi, kendi yığınımız heap bellekte büyüyebilir; çağrı yığınının birkaç megabaytlık sınırına takılmaz. İkincisi, yığına yalnız gereken bilgi itilir; çerçevenin dönüş adresi ve kaydedilmiş yazmaçlar gibi ek yükleri yoktur.

6.8 Çoklu yığın

Bir program aynı anda birden çok yığın kullanabilir. Her yığına ayrı bir dizi vermek kolaydır, ama bellek israf edilir: bir yığın dolduğunda ötekilerde boş yer olsa bile o yere erişilemez. Bu kesim birden çok yığını tek bir bellek dizisinde paylaştırmayı ele alır.

6.8.1 Tek dizide iki yığın

İki yığın için çok güzel bir çözüm vardır. m hücrelik dizide birinci yığın 0. hücreden yukarı, ikinci yığın $m - 1$. hücreden aşağı büyür. İki yığının üstleri birbirine doğru ilerler.



Şekil 6.10: Sekiz hücrelik dizide iki yığın. Yığın 1 0. hücreden sağa büyür ve üç öğe tutar (3, 8, 1; `top1` = 2). Yığın 2 7. hücreden sola büyür ve iki öğe tutar (2, 6; `top2` = 6). Aradaki üç hücre iki yığının ortak boş alanıdır.

Boş durumda `top1` = −1 ve `top2` = m'dir; ikisi de dizinin dışındaki hayali bir hücreyi gösterir. Yığın 1'e ekleme `top1`'i artırır, yığın 2'ye ekleme `top2`'yi azaltır. Dizi, iki üst yan yana geldiğinde, yani `top1 + 1 == top2` olduğunda doludur. Bu düzenin yararı, hiçbir hücrenin boşa gitmemesidir. Bir yığın az, öteki çok öğe tutsa bile dizi ancak toplam m öğe olduğunda dolar. Üç ve daha fazla yığın için bu hile işlemez, çünkü bir dizinin yalnız iki ucu vardır.

6.8.2 Tek bellekte *n* yığın

n yığın için Horowitz–Sahni'nin çözümü, *m* hücrelik `bellek` dizisini *n* kesime (*segment*) bölmektir. Her yığın kendi kesiminde büyür. İki yardımcı dizi kullanılır. `top[i]`, *i*. yığının üst öğesinin indisidir. `sinir[i]`, *i*. yığının kesiminin hemen altındaki hücrenin indisidir. Yığın *i*, `bellek[sinir[i]+1]` 'den `bellek[sinir[i+1]]` 'e kadar olan hücreleri kullanır. Son yığının üst sınırını da aynı biçimde yazabilmek için `sinir[n]` = *m* − *1* eklenir.

Başlangıçta bellek eşit parçalara bölünür ve bütün yığınlar boştur: *i = 0, …, n − 1* için `top[i] = sinir[i] = (m/n)·i − 1`. Bir yığının boş olması, üstünün kendi alt sınırına eşit olmasıdır: `top[i] == sinir[i]`. Dolu olması ise üstünün bir sonraki yığının alt sınırına ulaşmasıdır: `top[i] == sinir[i+1]`. Ekleme ve çıkarma bu iki sınamayla yazılır:

```
/* yığın i: bellek[sinir[i]+1 .. sinir[i+1]] */
bool ekle(int i, int x) {
  if (top[i] == sinir[i + 1]) return false; /* dolu */
  bellek[++top[i]] = x;
  return true;
}
bool al(int i, int *x) {
  if (top[i] == sinir[i]) return false;     /* boş */
  *x = bellek[top[i]--];
  return true;
}
```

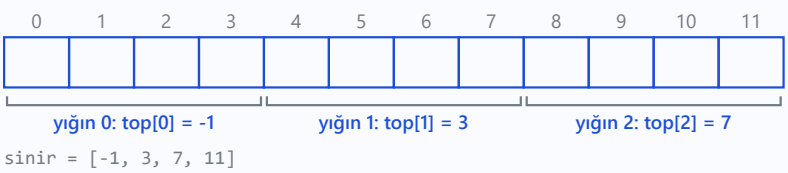
Bu kod `bellek`, `top` ve `sinir` dizilerinin küresel olarak tanımlandığını varsayar; ders kitabındaki gibi kısa tutulmuştur. Gerçek bir programda bu üç dizi bir `struct` 'ta toplanır ve yordamlara adresle verilir. Yordamların yapısı tek yığınınkiyle aynıdır, yalnız sabit sınırların yerini `sinir` dizisi alır. `ekle`, yığın doluysa `false` döner; değilse `top[i]` 'yi artırıp değeri yazar. `al`, yığın boşsa `false` döner; değilse üst değeri `*x` 'e yazıp `top[i]` 'yi azaltır. İkisi de *O(1)*'dir.

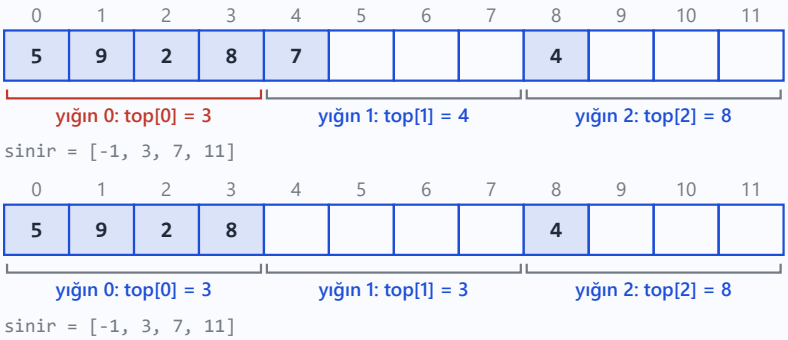
6.8.3 Kod izleme

m = 12 ve *n* = 3 olsun. Her yığına dört hücre düşer. Başlangıçta `top` = [−1, 3, 7] ve `sinir` = [−1, 3, 7, 11]'dir. Yığın 0 hücre 0–3'ü, yığın 1 hücre 4–7'yi, yığın 2 hücre 8–11'i kullanır. Sırasıyla `ekle(0, 5)`, `ekle(1, 7)`, `ekle(0, 9)`, `ekle(2, 4)`, `ekle(0, 2)`, `ekle(0, 8)`, `ekle(0, 6)` ve `al(1, &x)` çağrılıyor. Tablo 6.10 her çağrıyı verir.

Çağrı	Sınama	Yapılan	`top`
başlangıç	—	—	[−1, 3, 7]
`ekle(0, 5)`	−1 == 3? Hayır	`bellek[0] = 5`	[0, 3, 7]
`ekle(1, 7)`	3 == 7? Hayır	`bellek[4] = 7`	[0, 4, 7]
`ekle(0, 9)`	0 == 3? Hayır	`bellek[1] = 9`	[1, 4, 7]
`ekle(2, 4)`	7 == 11? Hayır	`bellek[8] = 4`	[1, 4, 8]
`ekle(0, 2)`	1 == 3? Hayır	`bellek[2] = 2`	[2, 4, 8]
`ekle(0, 8)`	2 == 3? Hayır	`bellek[3] = 8`	[3, 4, 8]
`ekle(0, 6)`	3 == 3? **Evet**	`false` : yığın 0 dolu	[3, 4, 8]
`al(1, &x)`	4 == 3? Hayır	`x = bellek[4]` = 7	[3, 3, 8]

Tablo 6.10: *m* = 12, *n* = 3 iken çoklu yığın izlemesi; `sinir` = [−1, 3, 7, 11] değişmez.





Şekil 6.11: On iki hücrelik bellekte üç yığın. Üstte başlangıç: bütün yığınlar boştur. Ortada yedinci çağrı: yığın 0'ın dört hücresi dolmuştur (5, 9, 2, 8) ve `ekle(0, 6)` reddedilir, oysa bellekte altı hücre boştur. Altta `al(1, &x)` 7'yi çıkarmıştır; yığın 1 yeniden boştur.

İzleme bir test programıyla doğrulandı; her çağrıdan sonraki `top` dizisi tablodakiyle aynı çıktı. Yedinci çağrı düzenin zayıflığını gösterir. Yığın 0 dolmuştur, çünkü üstü bir sonraki yığının sınırına (`sinir[1]` = 3) ulaşmıştır. Oysa bellekteki 12 hücreden yalnız altısı doludur: yığın 0'da dört, yığın 1'de bir, yığın 2'de bir öğe vardır. Kalan altı hücre boştur. Yine de `ekle`, "dolu" der.

6.8.4 Yer açma ve değerlendirme

Horowitz–Sahni bu sorunu bir `stack_full(i)` yordamıyla çözer. Yordam, *i*. yığın dolduğunda çağrılır. Önce sağdaki yığınlar arasında boş yeri olan ilkini arar. Bulursa arada kalan yığınları birer hücre sağa kaydırır ve böylece *i*. yığına bir hücre açar. Sağda yer yoksa aynı aramayı soldaki yığınlarda yapar ve *i*. yığının kendisi dahil arada kalanları birer hücre sola kaydırır. Hiçbir yerde boş hücre yoksa bellek gerçekten doludur. Aşağıda yalnız sağa kaydıran yarı, sadeleştirilerek verilmiştir:

```
/* yığın i doluyken sağda boş yer arar; bulursa arada kalan
   yığınları birer hücre sağa kaydırır (Horowitz'in fikri, sadeleştirilmiş) */
bool yer_ac_sag(int i) {
    int j = i + 1, k;
    while (j < N && top[j] == sinir[j + 1])   /* boş yeri olan ilk yığın */
        j++;
    if (j == N) return false;                 /* sağda yer yok */
    for (k = top[j]; k > sinir[i + 1]; k--)   /* bellek[sinir[i+1]+1 .. top[j]] */
        bellek[k + 1] = bellek[k];            /* bir sağa kayar */
    for (k = i + 1; k <= j; k++) {            /* sınırlar ve üstler de kayar */
        sinir[k]++;
        top[k]++;
    }
    return true;
}
```

Fonksiyon `i` 'nin sağındaki yığınları sırayla dener. `top[j] == sinir[j+1]` olan, yani kendisi de dolu olan yığınları atlar ve boş yeri olan ilk yığında durur. Hiçbiri bulunmazsa `false` döner. Bulunursa `sinir[i+1]+1` 'den `top[j]` 'ye kadar olan hücreler birer hücre sağa kopyalanır. Bu hücreler *i + 1*'den *j*'ye kadar bütün yığınların öğeleridir. Kopyalamanın sağdan başlaması gerekir; soldan başlansaydı her hücre bir sonrakinin üzerine yazar ve değerler kaybolurdu. Son olarak bu yığınların hem alt sınırları hem üstleri bir artırılır. Yığın *i*'nin üst sınırı `sinir[i+1]` de bir artmış olur; yığın *i* bir hücre kazanır.

İzlemedeki duruma uygulanınca: yığın 0 doludur; sağındaki yığın 1'de yer vardır (`top[1]` = 4, `sinir[2]` = 7). Yığın 1'in tek öğesi olan 7, `bellek[4]` 'ten `bellek[5]` 'e kayar. `sinir[1]` 4'e, `top[1]` 5'e çıkar. Bundan sonra `ekle(0, 6)` başarılıdır ve 6, `bellek[4]` 'e yazılır. Test programı bu adımları doğruladı: yer açmadan sonra `top` = [3, 5, 8], `sinir` = [−1, 4, 7, 11] oldu; ekleme sonrası yığın 0 "5 9 2 8 6", yığın 1 "7", yığın 2 "4" içeriyordu.

Bu yaklaşımın artıları ve eksileri açıktır. Artıları: tek bir dizi ve tek bir `malloc` yeter, yönetim kolaydır; push ve pop yine *O(1)*'dir; aynı fikir *n* kuyruğu tek dizide tutmak için de kullanılabilir. Eksileri: bir kesim dolunca, bellekte yer olsa bile "dolu" hatası alınır; yer açmak için yığınları kaydırmak pahalıdır. Kaydırılan öğe sayısı *m*'ye kadar çıkabilir. Bellek neredeyse doluyken her ekleme bir kaydırma gerektirebilir ve en kötü durumda her ekleme *O(m)* olur. Kesim boyutlarını önceden iyi kestirmek de zordur. Yığınların büyüme hızları farklıysa eşit bölme sık sık kaydırmaya yol açar.

6.9 Hangi çıkış sırası mümkün?

Yığının giriş sırasını nasıl değiştirebildiğini anlamak için klasik bir soru sorulur. 1, 2, 3, 4 sayıları bu sırayla push edilecektir. Push'ların arasına istenen sayıda pop konabilir ve sonunda yığın boşaltılır. Pop'ların çıkardığı sayıların sırası, yani **çıkış sırası**, hangi permütasyonlar olabilir?

Birkaç sıra kolayca elde edilir. 1 2 3 4 için her push'tan hemen sonra bir pop yapılır. 4 3 2 1 için dört push, ardından dört pop yapılır. 2 4 3 1 için şu dizi yeter: push 1, push 2, pop (2), push 3, push 4, pop (4), pop (3), pop (1). Peki 3 1 4 2 elde edilebilir mi? İlk çıkan 3 olacaksa 3 itilmeden önce 1 ve 2 de itilmiş olmalıdır; 3 çıktığı anda yığında alttan üste 1 ve 2 vardır. Sonraki çıkacak öğe ya üstteki 2 ya da henüz itilmemiş 4 olabilir. 1, 2'nin altında kaldığı için 2'den önce çıkamaz. 3 1 4 2 bu yüzden imkânsızdır.

Bu akıl yürütme genel bir kurala dönüşür. Bir x çıktığı anda, x'ten önce itilmiş ve henüz çıkmamış bütün öğeler yığındadır ve itildikleri sıranın tersiyle çıkmak zorundadır. Kural, yasak bir desen olarak da söylenebilir.

Teorem 6.2 (Yığınla elde edilebilen sıralar)

1, 2, …, n sırasıyla push edilirse bir $p_1\,p_2\,\dots\,p_n$ çıkış sırası ancak ve ancak şu koşulla elde edilebilir: $i < j < k$ olup $p_j < p_k < p_i$ sağlayan üç konum yoktur. Başka bir deyişle sırada "büyük, küçük, orta" deseni bulunmaz.

Kanıt. Önce desenin imkânsız olduğu gösterilir. p_i çıktığında, ondan küçük olan p_j ve p_k çoktan itilmiştir, çünkü sayılar artan sırada itilir. İkisi de henüz çıkmamıştır, çünkü çıkış sırasında p_i'den sonra gelirler. Demek ki ikisi de yığındadır ve $p_j < p_k$ olduğu için p_k daha sonra itilmiş, yani p_j'nin üstündedir. Bu durumda p_j, p_k'dan önce çıkamaz; oysa sırada önce gelir. Çelişki.

Ters yön için açgözlü bir yöntem kullanılır: sıradaki hedef p ise, p itilene kadar push yapılır, sonra üst öğe çekilir. Yöntem ancak hedef o anda yığının üstünde değilse ve zaten itilmişse takılır. Bu durumda hedef p_j'nin üstünde, ondan büyük ve henüz çıkmamış bir q vardır. q, sırada p_j'den sonra bir k konumundadır. Yığındaki öğelerin hepsi o ana dek çıkmış en büyük değerden, yani daha önceki bir konumdaki bir p_i'den küçüktür. Böylece $p_j < q = p_k < p_i$ deseni bulunur. Desen yoksa yöntem hiç takılmaz ve sıra elde edilir. ■

3 1 4 2 sırasında 3, 1, 2 konumları bu deseni oluşturur: 3 büyük, 1 küçük, 2 orta. Bu yüzden imkânsızdır. 2 4 3 1'de böyle bir üçlü yoktur. Elde edilebilen sıraların sayısı da bilinir. n öğe için bu sayı n. **Catalan sayısıdır**: $C_n = (2n)! / ((n+1)! \cdot n!)$. İlk değerler 1, 2, 5, 14, 42, 132'dir. $n = 4$ için 24 permütasyonun yalnız 14'ü elde edilebilir. $n = 3$ için altı permütasyondan yalnız biri, 3 1 2, imkânsızdır. Bu sayılar, bütün permütasyonları açgözlü yöntemle deneyen bir betikle $n = 1, \dots, 6$ için doğrulandı; aynı betik, desen koşuluyla açgözlü yöntemin $n \le 7$ için her permütasyonda aynı sonucu verdiğini de denetledi.

Örnek 6.3

1, 2, 3, 4, 5 sırasıyla itilecek. 3 5 4 2 1 ve 5 3 4 1 2 sıraları elde edilebilir mi?

3 5 4 2 1: push 1, push 2, push 3, pop (3); yığın [1, 2]. push 4, push 5, pop (5); yığın [1, 2, 4]. pop (4), pop (2), pop (1). Her hedef çekildiği anda üstteydi; sıra elde edilir. Desen aranırsa da "büyük, küçük, orta" üçlüsü bulunmaz.

5 3 4 1 2: 5'in çıkması için beş sayının hepsi itilmelidir; 5 çıkınca yığın alttan üste [1, 2, 3, 4] olur. Sıradaki hedef 3'tür, ama üstte 4 vardır. Sıra elde edilemez. Desen olarak 5, 3, 4 üçlüsü (büyük, küçük, orta) bulunur.

6.10 Karmaşıklık özeti

Tablo 6.11 bölümdeki işlemlerin maliyetlerini üç gerçekleme için yan yana verir. Her değer, ilgili kesimde gerekçesiyle açıklanmıştır.

İşlem	Sabit dizi	Büyüyen dizi	Bağlı
push	$O(1)$	$O(1)$ amortize, en kötü $O(n)$	$O(1)$
pop	$O(1)$	$O(1)$	$O(1)$
peek / boş / dolu	$O(1)$	$O(1)$	$O(1)$
***n* öğelik bellek**	MAX (sabit)	$\le 2n$ hücre	$n \cdot$ (öğe + işaretçi)
Temizleme	$O(1)$	$O(1)$ (tek `free`)	$\Theta(n)$
`fakt(n)` **çağrı yığını**	$\Theta(n)$ çerçeve (gerçeklemeden bağımsız)		

İşlem	Sabit dizi	Büyüyen dizi	Bağlı
Çoklu yığın ekle	*O(1)*; yer açma *O(m)*	—	—

Tablo 6.11: Yığın gerçeklemelerinde maliyetler (*n* öğe sayısı, *m* çoklu yığında bellek boyu).

Tablodaki temizleme satırı bir farkı vurgular. Dizi yığınları tek bir blokta durur; sabit dizide temizlemek için `ust = -1` yapmak, büyüyen dizide tek bir `free` yeter. Bağlı yığında ise her düğüm ayrı bir bloktur ve *n* düğüm için *n* kez `free` çağrılır. Bellek satırındaki "$\le 2n$" sınırı, kapasite 1'den başlayıp iki katına çıkarıldığında geçerlidir; kapasite hiç küçültülmezse çok sayıda pop'tan sonra boş kapasite *n*'nin çok üstünde kalabilir.

6.11 Sık yapılan hatalar

Yığın kodunda görülen hataların çoğu birkaç kalıba girer. Aşağıdaki liste, bölümdeki kodların her birinde karşılaşılabilecek hataları bir araya getirir.

1. **Sınır denetimini atlamak.** Boş yığından pop ya da dolu yığına push, diziyi sınırlarının dışında okur ya da yazar. C bunu denetlemez; komşu bellek bozulur ve hata çok sonra, ilgisiz bir yerde görünür.
2. `*top--` **yazmak.** İfade `*(top--)` olarak okunur ve işaretçiyi azaltır; çağırandaki üst indisi değişmez. Doğrusu `(*top)--`'dir. Derleyici uyarı vermez.
3. `top` **ile öğe sayısını karıştırmak.** `top` son öğenin indisidir; öğe sayısı `top + 1`'dir. Dolu denetimi `top == MAX - 1`'dir, `top == MAX` değildir.
4. **Bağlı push'ta önce** `*top = yeni` **yazmak.** Eski listenin adresi kaybolur, yeni düğüm kendini gösterir.
5. **Pop'ta değeri** `free`**'den sonra okumak ya da düğümü hiç bırakmamak.** İlki sarkan işaretçiyle tanımsız davranış, ikincisi bellek sızıntısıdır.
6. **Geçici işaretçiye gereksiz** `malloc` **yapmak.** `dugum *eski = malloc(…); eski = *top;` her çağrıda bir düğüm sızdırır. Var olan bir düğümü göstermek için atama yeter.
7. `realloc` **sonucunu doğrudan asıl işaretçiye yazmak.** Başarısızlıkta eski bloğun adresi kaybolur.
8. `bool` **sonuçlarını yok saymak.** Başarısız bir push'un öğesi sessizce kaybolur; başarısız bir pop'tan sonra eski değer kullanılmaya devam eder.

6.12 Alıştırmalar ve çözümleri

Sunumun alıştırma slaytındaki altı soru aşağıda çözümleriyle verilmiştir. Çözümlerin hepsi derlenip küçük test programlarıyla çalıştırıldı.

6.12.1 Hata döndüren `yigit_ekle`

Soru. Küresel dizi yığınında `yigit_ekle`'yi `exit` yerine `bool` döndürecek biçimde yeniden yazmak.

```
bool yigit_ekle(int *top, element item) {
    if (*top >= MAX_STACK_SIZE - 1)
        return false;          /* taşma: karar çağıranın */
    yigit[++*top] = item;
    return true;
}
bool yigit_al(int *top, element *item) {
    if (*top < 0)
        return false;          /* alt taşma */
    *item = yigit[(*top)--];
    return true;
}
```

Taşma durumunda fonksiyon `false` döner ve yığına dokunmaz. Yer varsa öğeyi yazıp `true` döner. `yigit_al`'ın da aynı biçimde yazılması gerekir; öğeyi dönüş değeri olarak vermek artık mümkün olmadığı için bir çıktı parametresiyle, `element *item` ile verir. Kapasite 3 iken 10, 20, 30, 40 eklemeye çalışan test programı "1 1 1 0" yazdı: dördüncü ekleme reddedildi. Ardından yığını boşaltan döngü 30, 20, 10 verdi ve `top` −1'e döndü. Program hiçbir noktada `exit` ile bitmedi.

6.12.2 Tek dizide iki yığın

Soru. Tek dizide iki yığın için `ekle(k, x)` ve `al(k)` yordamlarını yazmak ($k = 1$ ya da 2).

```
#define M 10
int dizi[M];
int top1 = -1, top2 = M;     /* ikisi de boş */

bool ekle(int k, int x) {
    if (top1 + 1 == top2) return false;   /* ortada buluştular: dolu */
    if (k == 1) dizi[++top1] = x;         /* yığın 1 yukarı büyür */
    else        dizi[--top2] = x;         /* yığın 2 aşağı büyür */
    return true;
}
bool al(int k, int *x) {
    if (k == 1) {
        if (top1 == -1) return false;     /* yığın 1 boş */
        *x = dizi[top1--];
    } else {
        if (top2 == M) return false;      /* yığın 2 boş */
        *x = dizi[top2++];
    }
    return true;
}
```

Boş durumda `top1` = −1, `top2` = `M` 'dir. Dolu denetimi iki yığın için ortaktır: `top1 + 1 == top2`. Yığın 1'e ekleme `dizi[++top1]`, yığın 2'ye ekleme `dizi[--top2]` ile yapılır; yığın 2 aşağı büyüdüğü için üstü azalır. Çıkarmada boş denetimleri ayrıdır: yığın 1 için `top1 == -1`, yığın 2 için `top2 == M`. Yığın 2'den çıkarma `dizi[top2++]` ile yapılır.

`M` = 10 iken yığın 1'e yedi, yığın 2'ye beş ekleme denendi. Yığın 1'in yedisi de girdi; yığın 2'nin ilk üçü girdi, sonraki ikisi reddedildi, çünkü toplam on hücre dolmuştu (`top1` = 6, `top2` = 7). Bir yığının çok, ötekinin az öğe tutması hiçbir hücreyi boşa harcamadı. Bütün işlemler *O(1)*'dir.

6.12.3 Boyutunu bilen bağlı yığın

Soru. Bağlı yığını, `top` ile birlikte bir `boyut` alanı da tutan bir `struct` içine taşımak.

```
typedef struct {
    dugum *top;
    int boyut;
} BYigit;

void by_baslat(BYigit *y) { y->top = NULL;  y->boyut = 0; }

bool by_push(BYigit *y, int x) {
    if (!push(&y->top, x)) return false;  /* bağlı push yeniden kullanılır */
    ++y->boyut;
    return true;
}
bool by_pop(BYigit *y, int *x) {
    if (!pop(&y->top, x)) return false;
    --y->boyut;
    return true;
}
void by_temizle(BYigit *y) {
    yigit_temizle(&y->top);
    y->boyut = 0;
}
```

`BYigit`, üst işaretçisiyle öğe sayısını birlikte tutar. Yordamlar bağlı yığının `push`, `pop` ve `yigit_temizle` yordamlarını yeniden kullanır; onlara `&y->top`, yani yapının içindeki işaretçi alanının adresini verir. `boyut` yalnız başarılı işlemlerden sonra değişir: başarısız bir push (bellek yok) ya da başarısız bir pop (yığın boş) sayacı bozmaz. Böylece öğe sayısı, listeyi saymaya gerek kalmadan *O(1)*'de bilinir. Altı push ve bir pop sonrası `boyut` 5 oldu; temizlemeden sonra 0 oldu ve sayaçlı sarmalayıcı altı `malloc`'a karşı altı `free` saydı.

6.12.4 Yığınla dizgi ters çevirme

Soru. Bir dizgiyi yığın kullanarak ters çeviren `void ters(char *s)` yordamını yazmak.

Çözüm. Bu yordam, Tablo 6.5'te izlenen `ters_cevir`'in aynısıdır: karakterler sırayla push edilir, sonra pop edilerek aynı diziye geri yazılır. Süre $\Theta(n)$, ek bellek $\Theta(n)$'dir. Sağlam bir sürüm, dizginin uzunluğunu önce `strlen` ile bulup yığını bu kapasiteyle (örneğin `yigit_olustur(n)` ile) kurar; böylece `KAP` varsayımına gerek kalmaz.

6.12.5 En alttaki öğe

Soru. Yalnız push ve pop kullanarak bir yığının en alttaki öğesini döndüren yordamı yazmak; yardımcı bir yığın kullanılabilir.

```
/* y boş değilse en alttaki öğeyi *x'e yazar; y'yi eski hâline getirir */
bool en_alt(Yigit *y, int *x) {
    Yigit yard;  int v;
    if (yigit_bos(y)) return false;
    yigit_baslat(&yard);
    while (pop(y, &v))          /* hepsini yardımcı yığına aktar */
        push(&yard, v);
    *x = v;                     /* son çıkan, en alttakiydi */
    while (pop(&yard, &v))      /* geri aktar: sıra korunur */
        push(y, v);
    return true;
}
```

Yordam, yığındaki bütün öğeleri yardımcı yığına aktarır. Aktarma sırasında en son çekilen öğe, asıl yığının en alttaki öğesidir; `v` değişkeni döngü bittiğinde onu tutar. Sonra öğeler yardımcı yığından asıl yığına geri aktarılır. İki aktarma da sırayı ters çevirdiği için asıl yığın ilk hâline döner. Yığın baştan boşsa `v` hiç atanmayacağı için yordam en başta `false` döner. Alttan üste 4, 7, 1, 9 tutan yığında yordam 4'ü verdi ve yığın üstten okunduğunda yine 9, 1, 7, 4 çıktı. n öğeli yığında $2n$ push ve $2n + 2$ pop yapılır; süre $\Theta(n)$, ek bellek $\Theta(n)$'dir.

6.12.6 `fakt(5)` çağrısında çerçeve sayısı

Soru. `fakt(5)` çağrısında yığında en çok kaç çerçeve olur? `main` dahil çizmek.

Çözüm. Taban durum `n <= 1` olduğundan çağrılar `fakt(5)`, `fakt(4)`, `fakt(3)`, `fakt(2)`, `fakt(1)`'dir. `fakt(1)` çalışırken bu beş çerçeve ile `main`'in çerçevesi birlikte yığındadır: en çok **6** çerçeve. Çizim, Tablo 6.9'un "`fakt(1)` çağrılır" satırıdır: alttan üste main, fakt(5), fakt(4), fakt(3), fakt(2), fakt(1). Taban durum `n == 0` olarak yazılsaydı `fakt(0)` da çağrılır ve sayı 7 olurdu.

6.13 Özet

- **Yığın**, ekleme ve silmenin yalnız üst uçtan yapıldığı sıralı bir listedir. Son giren ilk çıkar (**LIFO**). Ekleme **push**, çıkarma **pop**, çıkarmadan okuma **peek** adını alır.
- Dolu yığına ekleme **taşma**, boş yığından çıkarma **alt taşma**dır. Yığın ADT'si işlemleri ve bu hata durumlarını gerçeklemeden bağımsız tanımlar.
- **Dizi yığınında** `top` üst öğenin indisidir: `top` = −1 boş, `top` = MAX − 1 dolu; öğe sayısı `top + 1`'dir. Push `yigit[++*top] = item`, pop `return yigit[(*top)--]` ile yazılır. `*top--` işaretçiyi azaltır ve hatalıdır.
- `exit` ile biten yordamlar yerine `bool` döndüren yordamlar kararı çağırana bırakır. Dizi ve üst indisi bir `struct`'ta tutulur; küresel değişken kullanılmaz. Başlık dosyasındaki **opak tip**, gerçeklemeyi kullanan koddan gizler.
- **Bağlı yığında** `top` liste başıdır; push ve pop $O(1)$'dir. Push'ta önce bağlanır, sonra üst değişir; pop'ta önce değer ve bağ kurtarılır, en son `free` çağrılır. Her `malloc`'un bir `free`'si olmalıdır; yığın `yigit_temizle` ile $\Theta(n)$'de boşaltılır.
- **Büyüyen dizi** dolduğunda kapasitesini `realloc` ile iki katına çıkarır. Tek bir push $O(n)$ olabilir, ama n push'ta toplam kopya $2n$'den azdır; push'un **amortize** maliyeti $O(1)$'dir. Pratikte dizi yığınları, her push'ta `malloc` yapan bağlı yığından çok daha hızlıdır.
- **Çağrı yığını**, her çağrı için dönüş adresi, parametreler ve yerellerden oluşan bir **çerçeve** iter. Sonsuz özyineleme ve büyük yerel diziler onu taşırır; yerel bir değişkenin adresini döndürmek sarkan işaretçi doğurur. Her özyinelemeli algoritma kendi yığınını tutan bir döngüye çevrilebilir.
- **Çoklu yığında** tek bir bellek dizisi `top[i]` ve `sinir[i]` ile paylaşılır. İki yığın uçlardan büyüyerek hiç hücre harcamaz. n yığında bir kesim dolunca yığınlar kaydırılarak yer açılır; bu $O(m)$ sürebilir.

- 1, …, n sırasıyla itilince elde edilebilen çıkış sıraları, "büyük, küçük, orta" deseni içermeyen sıralardır; sayıları Catalan sayısıdır (n = 4 için 14).

6.14 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

yığın *stack* · son giren ilk çıkar *LIFO (last in, first out)* · üst *top* · push (üste ekleme) *push* · pop (üstten çıkarma) *pop* · taşma *overflow* · alt taşma *underflow* · soyut veri tipi *abstract data type (ADT)* · gerçekleme *implementation* · arayüz *interface* · başlık dosyası *header file* · çoklu ekleme koruması *include guard* · opak tip *opaque type* · eksik tip *incomplete type* · küresel değişken *global variable* · değerle çağrı *call by value* · işleç önceliği *operator precedence* · birleşme yönü *associativity* · const niteleyicisi *const qualifier* · bağlı liste *linked list* · düğüm *node* · malloc *malloc* · free *free* · realloc *realloc* · bellek sızıntısı *memory leak* · sarkan işaretçi *dangling pointer* · dinamik dizi *dynamic array* · kapasite *capacity* · amortize analiz *amortized analysis* · önbellek *cache* · çağrı yığını *call stack* · yığın çerçevesi *stack frame (activation record)* · dönüş adresi *return address* · yığın taşması *stack overflow* · özyineleme *recursion* · taban durum *base case* · parantez denetimi *balanced parentheses* · ifade değerlendirme *expression evaluation* · sonek *postfix* · geri izleme *backtracking* · geri al *undo* · derinlik öncelikli arama *depth-first search (DFS)*

6.15 Kendini Dene

1. `KAP` = 3 olan `struct` yığınına sırasıyla `push 4`, `push 7`, `pop`, `push 1`, `push 6`, `push 2` uygulanıyor; sonra yığın `while (pop(&y, &x))` döngüsüyle boşaltılıyor. Hangi push `false` döner ve döngü hangi değerleri çıkarır?

   Cevap

   4 ve 7 girer; pop 7'yi çıkarır, yığın [4] olur. 1 ve 6 girer; yığın [4, 1, 6] ile doludur (`ust` = 2). Bu yüzden `push 2` `false` döner. Döngü 6, 1, 4 çıkarır ve dördüncü pop'ta durur.

2. `yigit_al`'da `return yigit[(*top)--];` yerine `return yigit[*top--];` yazılmıştır. Yığın boşalana kadar `yigit_al` çağıran bir döngü ne yapar?

   Cevap

   `*top--`, `*(top--)` olarak okunur: azalan şey fonksiyonun yerel işaretçisidir, çağırandaki `top` değişmez. Her çağrı aynı üst öğeyi döndürür ve yığın hiç boşalmaz; döngü sonsuza dek sürer. Derleyici bu hatayı uyarıyla bildirmez.

3. 1, 2, 3, 4, 5 sırasıyla itiliyor. 3 5 4 2 1 ve 5 3 4 1 2 çıkış sıralarından hangisi elde edilebilir?

   Cevap

   3 5 4 2 1 elde edilebilir: push 1, 2, 3, pop (3), push 4, 5, pop (5), pop (4), pop (2), pop (1). 5 3 4 1 2 elde edilemez: 5 çıkınca yığın [1, 2, 3, 4] olur ve 3, 4'ün altında kalır. Desen olarak 5, 3, 4 üçlüsü "büyük, küçük, orta"dır.

4. Boş bir bağlı yığına `push 10`, `push 20`, `push 30`, `pop`, `push 40` uygulanıyor. Yığın baştan sona nasıl görünür? O ana kadar kaç `malloc` ve kaç `free` yapılmıştır? `yigit_temizle`'den sonra toplam `free` sayısı kaç olur?

   Cevap

   Yığın 40 → 20 → 10 → NULL'dır. Dört push dört `malloc`, bir pop bir `free` yapmıştır. `yigit_temizle` kalan üç düğümü iade eder; toplam `free` 4 olur ve `malloc` sayısına eşitlenir.

5. Kapasitesi 1 olan boş bir büyüyen dizi yığınına 9 push yapılıyor; her dolmada kapasite iki katına çıkıyor. Kaç büyütme olur, toplam kaç

öğe kopyalanır ve son kapasite kaçtır?

Cevap

Büyütmeler 2., 3., 5. ve 9. push'ta, kapasite 1, 2, 4 ve 8 iken olur: 4 büyütme. Kopyalanan öğe sayısı 1 + 2 + 4 + 8 = 15'tir ($2n$ = 18'den az). Son kapasite 16'dır.

6. Bağlı pop'ta `free(eski);` satırı `*x = eski->deger;` satırından önceye alınıyor. Program testlerde çoğu zaman doğru çalışıyor. Neden yine de hatalıdır?

Cevap

`free`'den sonra `eski` sarkan bir işaretçidir ve `eski->deger`'i okumak tanımsız davranıştır. İade edilen bellek hemen başka bir işe verilmediği için eski değer çoğu zaman yerinde durur; bu yüzden testler geçer. Araya başka bir `malloc` girdiğinde ya da bellek ayırıcı bloğa kendi kayıtlarını yazdığında değer bozulur.

7. m = 12, n = 3 olan çoklu yığın başlangıç durumundayken `ekle(1, ·)` arka arkaya kaç kez başarılı olur? Bu sırada `al(0, &x)` ne döndürür?

Cevap

Yığın 1 hücre 4–7'yi kullanır; `top[1]` 3'ten 7'ye çıkana kadar dört ekleme başarılıdır, beşincisi `top[1] == sinir[2]` (7 == 7) yüzünden reddedilir. Yığın 0 boştur (`top[0] == sinir[0]` = −1); `al(0, &x)` `false` döndürür.

8. Tek dizide iki yığın (`M` = 10) boşken yığın 1'e 4, yığın 2'ye 6 öğe ekleniyor. Bir sonraki `ekle(1, x)` çağrısı ne döndürür? Neden?

Cevap

`top1` = 3, `top2` = 4 olur. `top1 + 1 == top2` doğru olduğu için dizi doludur ve çağrı `false` döndürür. On hücrenin hepsi doludur.

9. Kapasite her dolmada iki katına çıkarılmak yerine 10 hücre artırılsaydı n push'un toplam maliyeti ne olurdu?

Cevap

Yaklaşık $n/10$ büyütme olur ve her biri o anki öğelerin hepsini kopyalar: $10 + 20 + \ldots \approx n^2/20$. Toplam maliyet $\Theta(n^2)$, push başına amortize maliyet $\Theta(n)$ olur. Amortize $O(1)$ için kapasitenin sabit bir çarpanla büyümesi gerekir.

6.16 İleri okuma

- E. Horowitz, S. Sahni ve S. Anderson-Freed, *Fundamentals of Data Structures in C* (2. baskı), Bölüm 3.1–3.2, 3.7 ve 4.3: bu notun izlediği dizi yığını, dinamik diziyle yığın, çoklu yığın ve bağlı yığın kodlarının kaynağıdır.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (3. baskı), Kesim 10.1 ve Bölüm 17: yığını sözde kodla kısaca tanımlar; dinamik tabloların amortize analizini toplama, muhasebe ve potansiyel yöntemleriyle ayrıntılı verir.
- R. Sedgewick, *Algorithms in C, Parts 1–4*, Bölüm 4: yığını bir soyut veri tipi örneği olarak C'de, arayüz ve gerçekleme ayrımıyla ele alır.
- D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Kesim 2.2.1–2.2.2: yığınların ardışık bellekte gerçeklenmesini, çoklu yığınların kaydırmalı yönetimini ve yığınla elde edilebilen permütasyonları alıştırmalarıyla inceler.

6.17 Kaynaklar

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, Bölüm 3.1–3.2, 3.7 ve 4.3.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3. baskı, MIT Press, Bölüm 10.1 (yığınlar) ve 17 (amortize analiz).
3. R. Sedgewick, *Algorithms in C, Parts 1–4*, 3. baskı, Addison-Wesley, Bölüm 4.
4. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3. baskı, Addison-Wesley, 1997, Kesim 2.2.1–2.2.2.

KISIM II · BÖLÜM 7

Kuyruklar

Yığında ekleme ve çıkarma aynı uçtan yapılır; en son giren öğe ilk çıkar. Gündelik hayatta ise çoğu sıra bunun tersine işler: bankada, yemekhanede, yazıcıda önce gelen önce hizmet alır. Bu kurala uyan veri yapısına kuyruk denir. Kuyrukta yeni öğe arkadan girer, işlenecek öğe önden çıkar. Not, kuyruğu önce soyut bir veri tipi olarak tanımlar. Ardından dizi üzerinde en basit gerçeklemeyi kurar ve bu gerçeklemenin "sahte doluluk" sorununu kod izlemesiyle gösterir. Sorunun çözümü dairesel kuyruktur: dizinin sonu başına bağlanmış gibi düşünülür ve indisler kalan işlemiyle sarılır. Notun öteki konuları dairesel kuyrukta dolu ile boşu ayırt etmenin üç yolu ve bağlı listeyle kurulan kuyruktur. Son kesim iki uçtan da çalışan çift uçlu kuyruğu, işletim sistemlerindeki zaman dilimli zamanlamayı ve iki yığınla kuyruk kurmayı ele alır.

Kod (C): `github.com/sercankulcu/data-structures-and-algorithms-c/Bolum07`

ÖĞRENME ÇIKTILARI

İlk giren ilk çıkar kuralı
Kuyruğu $Q = \{a_0, \ldots, a_{n-1}\}$ biçiminde tanımlamak, ön ve arka uçlarını göstermek ve kuyruk soyut veri tipinin işlemlerini hata durumlarıyla birlikte saymak.

Doğrusal kuyruğun sorunu
Dizi ile kurulan doğrusal kuyrukta önde boşalan hücrelerin neden yeniden kullanılamadığını açıklamak ve kaydırma çözümünün maliyetini hesaplamak.

Dairesel kuyruk
`% MAX` ile indisleri sarmak, öğe sayısını ve bir geri adımı doğru hesaplamak, dolu ile boşu üç yöntemle ayırt etmek.

Bağlı kuyruk
`on` ve `arka` işaretçileriyle ekleme ve çıkarmayı $O(1)$ sürede yapmak; boş kuyruğa ilk eklemeyi ve son öğenin çıkışını bellek hatası olmadan yönetmek.

Deque ve uygulamalar
Çift uçlu kuyruğu dairesel dizide gerçeklemek; zaman dilimli iş zamanlamasında kuyruğu kullanmak; iki yığınla kuyruk kurup amortize maliyetini gerekçelendirmek.

7.1 Kuyruk nedir?

Yığın ve kuyruk, bir öğe topluluğuna erişimi bilerek kısıtlayan iki temel yapıdır. İkisinde de yalnız belirli bir uçtaki öğeye ulaşılır; ortadaki öğeler görünmez. Bu kısıtlama bir eksiklik değildir. Erişim kuralı ne kadar dar olursa yapı o kadar basit ve hızlı gerçeklenir, programın davranışı da o kadar kolay öngörülür. İki yapıyı ayıran tek şey, çıkarılacak öğenin hangisi olduğudur. Yığında en son eklenen öğe çıkar. Kuyrukta ise en uzun süredir bekleyen, yani en önce eklenen öğe çıkar.

7.1.1 İlk giren ilk çıkar

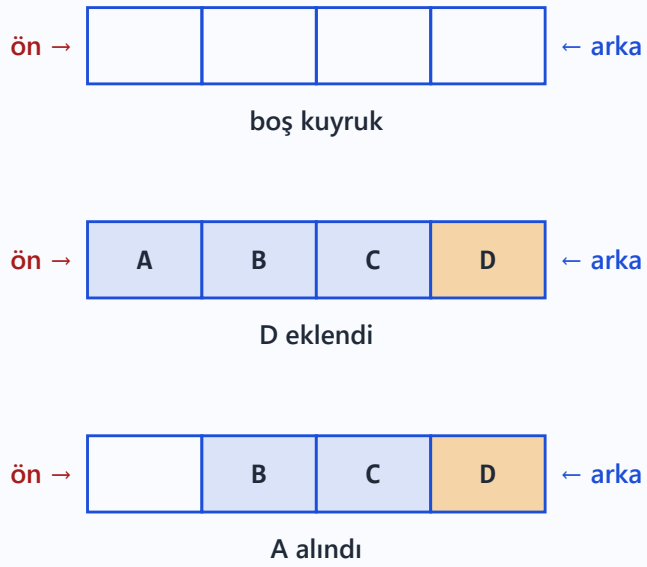
Bir otobüs durağındaki sıra kuyruğun en sade örneğidir. Yeni gelen yolcu sıranın sonuna geçer. Otobüs gelince sıranın başındaki yolcu biner. Sıranın ortasına girmek ya da ortadan birini almak kurala aykırıdır. Bu düzende yolcular sıraya hangi sırayla girdiyse otobüse de o sırayla biner. Kuyruk veri yapısı bu davranışı birebir taşır.

Tanım 7.1

Kuyruk (*queue*), eklemenin bir uçtan, silmenin öteki uçtan yapıldığı sıralı bir listedir. $Q = \{a_0, a_1, \ldots, a_{n-1}\}$ kuyruğunda a_0 kuyruğun **önünde** (*front*), a_{n-1} ise **arkasında** (*rear*) bulunur. Her i için a_i, a_{i+1}'den önce eklenmiştir. Yeni öğe arkaya eklenir, çıkarma her zaman önden yapılır. Bu kurala **ilk giren ilk çıkar** (*FIFO, first-in-first-out*) denir.

Tanımdaki son cümle kuyruğun bütün davranışını belirler. Öğeler kuyruğa a_0, a_1, a_2 sırasıyla girdiyse kuyruktan da aynı sırayla çıkar. Yığında aynı ekleme sırası ters çıkış sırası verirdi: a_2, a_1, a_0. Yığın kuralına bu yüzden **son giren ilk çıkar** (*LIFO*) denir. Kısaca söylemek gerekirse yığın sırayı ters çevirir, kuyruk ise korur.

Şekil 7.1 boş bir kuyruğa sırasıyla A, B, C ve D'nin eklenmesini, ardından bir çıkarmayı gösterir. Boş kuyrukta ön ve arka aynı yerdedir; ortada öğe yoktur. Her eklemede arka bir hücre ilerler. Çıkarma işlemi önde duran A'yı verir, çünkü kuyruğa ilk giren odur. A çıktıktan sonra öndeki hücre boşalır ve kuyruğun yeni önü B olur. Şekilde dikkat çeken bir ayrıntı vardır: çıkarma, kalan öğeleri yerinden oynatmaz. Ön uç bir hücre sağa kayar, öğeler olduğu yerde kalır. Bu ayrıntı, dizi ile kurulan kuyruğun bütün sorunlarının ve çözümlerinin kaynağıdır.



Şekil 7.1: İlk giren ilk çıkar. Üstte boş kuyruk; ön ve arka aynı yerdedir. Ortada A, B, C ve D arkaya eklenmiştir. Altta bir çıkarma yapılmış ve ilk eklenen A kuyruktan alınmıştır; öndeki hücre boşalmıştır, öteki öğeler yerinden oynamamıştır.

7.1.2 Kuyruk soyut veri tipi

Bir veri yapısını gerçeklemeye geçmeden önce onunla hangi işlemlerin yapılacağını ve her işlemin ne anlama geldiğini belirlemek gerekir. Bu belirleme, yapının iç düzeninden bağımsızdır. Kuyruğu kullanan bir program, öğelerin dizide mi bağlı listede mi durduğunu bilmek zorunda değildir; yalnız ekleme ve çıkarmanın ilk giren ilk çıkar kuralına uyduğunu bilmesi yeter. İşlemlerin bu biçimde, gerçeklemeden ayrı olarak verilmesine **soyut veri tipi** (*abstract data type, ADT*) denir. Tablo 7.1 kuyruk soyut veri tipinin altı işlemini verir.

İşlem	Anlamı	Hata durumu
`kuyruk_olustur(max)`	En çok `max` öğe alabilen boş bir kuyruk kurar.	—
`kuyruk_dolu(Q)`	Kuyrukta yer kalmadıysa doğru döndürür.	—
`kuyruk_ekle(Q, x)` (*enqueue*)	`x`'i kuyruğun arkasına ekler.	Kuyruk doluysa **taşma**.
`kuyruk_bos(Q)`	Kuyrukta öğe yoksa doğru döndürür.	—
`kuyruk_al(Q)` (*dequeue*)	Öndeki öğeyi kuyruktan çıkarır ve döndürür.	Kuyruk boşsa **alt taşma**.
`kuyruk_on(Q)` (*peek*)	Öndeki öğeyi çıkarmadan okur.	Kuyruk boşsa hata.

Tablo 7.1: Kuyruk soyut veri tipinin işlemleri. Parantez içindeki İngilizce adlar, kaynaklarda sık geçen karşılıklardır.

Tablodaki iki hata durumunun adları yığından tanıdıktır. Dolu bir yapıya ekleme yapılmak istenirse **taşma** (*overflow*), boş bir yapıdan öğe alınmak istenirse **alt taşma** (*underflow*) olur. "Dolu" kavramı yalnız sabit boyutlu gerçeklemelerde anlam taşır. Bağlı listeyle kurulan bir kuyruk, bellek bitene kadar dolmaz. Yine de soyut veri tipi `kuyruk_dolu` işlemini içerir. Böylece aynı arayüz hem dizi hem bağlı liste gerçeklemesini kapsar.

Yığın ile kuyruğun farkı tabloda tek bir sözcüğe sığar: uç. Yığının tek bir etkin ucu vardır. Ekleme de çıkarma da *üstte* yapılır. Kuyruğun ise iki ucu vardır ve her işlem kendi ucunu kullanır: ekleme *arkada*, çıkarma *önde* yapılır. Bu fark gerçeklemeye de yansır. Yığını tek bir indisle (`top`) yönetmek yeter. Kuyruk ise iki indis ister: biri önü, öteki arkayı izler. Bu iki indisin birbirine göre konumu, notun geri kalanının asıl konusudur.

7.1.3 Kuyruğun kullanıldığı yerler

Kuyruk, "sırası gelen hizmet alır" ilkesinin geçerli olduğu her yerde karşımıza çıkar. Bunların bir kısmı gerçek hayattaki sıraların bilgisayardaki karşılığıdır, bir kısmı ise algoritmaların iç işleyişinde kullanılır.

- **İşletim sistemi.** Çalışmaya hazır süreçler işlemciyi, yazıcıya gönderilen belgeler yazıcıyı, disk istekleri diski kuyrukta bekler. Bir isteğin sonradan gelip öndekileri geçmesi adaletsiz olurdu; kuyruk bu adaleti kendiliğinden sağlar.
- **Tampon.** Klavyeden gelen tuşlar, ağdan gelen paketler, ses kartına giden örnekler bir **tampon** (*buffer*) içinde bekler. Tampon, veriyi üreten taraf ile tüketen taraf arasında duran bir kuyruktur. İki tarafın hızı farklı olabilir: klavye bir anda birkaç tuş gönderebilir, program onları biraz sonra okur. Bu tamponlar neredeyse her zaman sabit boyutlu bir dairesel kuyruktur.
- **Genişlik öncelikli arama.** Bir çizgede ya da ağaçta düğümleri başlangıca uzaklık sırasıyla ziyaret eden **genişlik öncelikli arama** (*breadth-first search, BFS*) kuyruğa dayanır. İşlenen düğümün komşuları ya da çocukları kuyruğun arkasına eklenir, sıradaki düğüm önden alınır. Ağaçta aynı yöntem düğümleri düzey düzey, soldan sağa verir; buna **düzey sırası** (*level-order*) gezinme denir.
- **Benzetim.** Bir banka gişesinin, bir trafik ışığının ya da bir üretim hattının bilgisayarda canlandırılmasında gelen her müşteri ya da araç kuyruğa girer. Benzetim, ortalama bekleme süresi gibi soruları sistemi kurmadan cevaplamaya yarar.
- **Mesaj kuyrukları.** Büyük yazılım sistemlerinde sunucular birbirine işleri bir mesaj kuyruğu üzerinden iletir. İşler gönderiliş sırasıyla işlenir ve alıcı o an meşgul olsa bile kaybolmaz.

Bu listeye sık karıştırılan bir yapı da eklenmelidir. **Öncelikli kuyruk** (*priority queue*) adında "kuyruk" sözcüğü geçer, ama ilk giren ilk çıkar kuralına uymaz. Öncelikli kuyrukta her öğenin bir önceliği vardır ve her çıkarmada en yüksek öncelikli öğe alınır. Acil servisteki sıra buna iyi bir örnektir: ağır hasta, kendisinden önce gelmiş hafif hastaların önüne geçer. Öncelikli kuyruğun verimli gerçeklemesi heap adlı ağaç yapısıyla yapılır. Bu notta "kuyruk" sözcüğü, aksi söylenmedikçe, ilk giren ilk çıkar kuralına uyan sıradan kuyruğu anlatır.

Sunumda bir de etkileşimli kuyruk bulunur. Orada "ekle" düğmesi kuyruğun arkasına yeni bir öğe koyar, "al" düğmesi öndeki öğeyi çıkarır. Birkaç denemeden sonra aynı gözlem tekrar tekrar doğrulanır: çıkan öğelerin sırası, giren öğelerin sırasıyla aynıdır. Yığında bu sıra tam tersiydi.

7.2 Dizi ile doğrusal kuyruk

Kuyruğu C'de kurmanın en kısa yolu, sabit boyutlu bir dizi ve iki tam sayı indis kullanmaktır. Bu kesim önce bu en basit gerçeklemeyi verir. Ardından kod izlemesiyle önemli bir kusurunu ortaya çıkarır ve kusuru gidermenin ilk, pahalı yolunu inceler.

7.2.1 Gösterim ve C kodu

Yığında tek bir `top` indisi yeterliydi. Kuyrukta iki uç olduğu için iki indis tutulur: `on` ve `arka`. Öğeler dizinin içinde, `on`'dan sonraki hücreden `arka`'nın gösterdiği hücreye kadar yan yana durur. Aşağıdaki kod Horowitz, Sahni ve Anderson-Freed'in kitabındaki gerçeklemenin Türkçe adlandırılmış biçimidir:

```
#define MAX_QUEUE_SIZE 100
element kuyruk[MAX_QUEUE_SIZE];
int on = -1;        /* ilk öğenin bir öncesi */
int arka = -1;      /* son öğe */

void kuyruk_ekle(int *arka, element item) {
    if (*arka >= MAX_QUEUE_SIZE - 1)   /* yer var mı? */
        kuyruk_dolu();         /* uyar ve çık */
    kuyruk[++*arka] = item;    /* önce artır, sonra yaz */
}
element kuyruk_al(int *on, int arka) {
    if (*on == arka)
        kuyruk_bos();          /* boşsa uyar ve çık */
    return kuyruk[++*on];      /* artır ve öndekini döndür */
}
```

İlk satırlar kuyruğun belleğini kurar. `MAX_QUEUE_SIZE` dizinin hücre sayısıdır. `element`, kuyrukta saklanacak öğenin tipidir. Yığın gerçeklemesinde olduğu gibi bu tip programa göre bir `struct` olarak tanımlanır; örneğin tek bir `int deger` alanı taşıyabilir. `kuyruk` dizisi ve iki indis küresel değişkendir. Başlangıçta ikisi de −1'dir.

İki indisin anlamı dikkatle okunmalıdır, çünkü bütün kod bu anlamlara dayanır. `arka`, son öğenin bulunduğu hücreyi gösterir. `on` ise ilk öğeyi değil, **ilk öğenin bir öncesini** gösterir. Bu seçim ilk bakışta tuhaf görünür, ama iki işlemi de aynı biçime sokar: her iki işlem de önce kendi indisini bir artırır, sonra o hücreyi kullanır. Kuyruktaki öğeler bu yüzden `kuyruk[on+1]`, …, `kuyruk[arka]` hücrelerindedir ve öğe sayısı `arka - on`'dur. İki indis eşitse aralık boştur: **boş kuyruğun koşulu `on == arka`'dır**. Başlangıçtaki −1 ve −1 değerleri de bu koşulu sağlar.

`kuyruk_ekle` önce yer olup olmadığına bakar. `arka` dizinin son hücresine (`MAX_QUEUE_SIZE - 1`) ulaştıysa arkasına yazılacak hücre kalmamıştır ve `kuyruk_dolu` çağrılır. Yer varsa `++*arka` önce `arka`'yı bir artırır, ifadenin değeri de artırılmış indis olur; öğe o hücreye yazılır. `kuyruk_al` önce boşluğa bakar. Kuyruk boş değilse `++*on` ile ön indisi, çıkarılacak öğenin hücresine ilerler ve o hücredeki öğe döndürülür.

Parametrelerin türleri bilinçli seçilmiştir. `kuyruk_ekle` `arka`'yı değiştirdiği için onu adresle, `int *arka` olarak alır; çağrı `kuyruk_ekle(&arka, x)` biçimindedir. `kuyruk_al` ise `on`'u değiştirir, `arka`'yı yalnız okur. Bu yüzden `on`'u adresle, `arka`'yı değerle alır. Parametre adları küresel değişkenlerin adlarıyla aynıdır. C'de bu geçerlidir: fonksiyonun içinde parametre adı küresel değişkeni gölgeler. Kodu okurken `*arka`'nın küresel `arka`'yı, çıplak `arka`'nın ise parametrenin kopyasını anlattığı akılda tutulmalıdır.

Kod, hata durumunda `kuyruk_dolu` ve `kuyruk_bos` fonksiyonlarını çağırır. Bu fonksiyonlar yığın gerçeklemesindeki karşılıkları gibi bir uyarı yazıp programı bitirir:

```
void kuyruk_dolu(void) {
    printf("Kuyruk Dolu!!!\n");  exit(-1);
}
void kuyruk_bos(void) {
    printf("Kuyruk Bos!!!\n");   exit(-2);
}
```

Programı bitirmek kaba bir hata yönetimidir, ama kodu kısa tutar ve öğrenmenin ilk aşamasında asıl düşünceyi öne çıkarır. Kuyruk dolu ya da boşken işleme devam edilmez; dolayısıyla `kuyruk_ekle` ve `kuyruk_al` içinde hata sonrası için ayrı bir dal gerekmez. Güvenli sürümler hata durumunda `false` döndürür ve kararı çağırana bırakır. Bu sürüm dairesel kuyrukta ele alınır.

Her iki işlem de bir karşılaştırma, bir artırma ve bir dizi erişimi yapar. Kuyrukta kaç öğe olduğundan bağımsız olarak sabit iş yapıldığı için iki işlemin de zaman maliyeti *O(1)*'dir. Kuyruk sabit boyutlu bir dizi kullanır; bellek maliyeti `MAX_QUEUE_SIZE` öğedir ve öğe sayısına bağlı değildir.

> **Sık yapılan hata**
>
> `kuyruk[++*arka]` yerine `kuyruk[*arka++]` yazmak. Sonek `++` işleci önceliği yüksek olduğu için `*arka++` ifadesi `*(arka++)` olarak okunur. Bu ifade `arka`'nın gösterdiği tam sayıyı değil, işaretçinin kendisini ilerletir. Küresel `arka` değişmez; öğe de yanlış hücreye yazılır. Önek biçim `++*arka` ise önce `*arka`'ya erişir, sonra onu artırır. Öncelikten şüphe edilen yerde `++(*arka)` yazmak en güvenli yoldur.

7.2.2 Kod izleme: sahte doluluk

Doğrusal kuyruğun kusuru, kodu birkaç adım çalıştırınca görünür. İzlemede dizi küçük tutulur: `MAX_QUEUE_SIZE = 4`. Başlangıçta `on = arka = -1`'dir. İşlem sırası şöyledir: A, B ve C eklenir; iki kez öğe alınır; sonra D ve E eklenmek istenir. Öğeler tek harfli değerler olduğu için izlemede `element`, `char` olarak düşünülebilir.

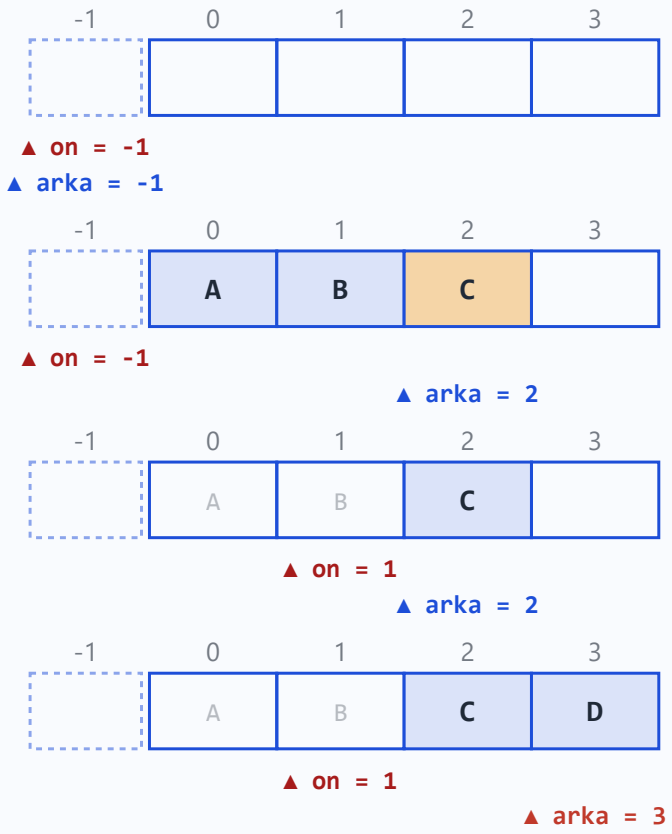
İşlem	Denetim	`on`	`arka`	Dizi [0..3]	Sonuç
Başlangıç	—	−1	−1	`. . . .`	Boş
`kuyruk_ekle(&arka, 'A')`	−1 ≥ 3? Hayır	−1	0	`A . . .`	`kuyruk[0] = A`
`kuyruk_ekle(&arka, 'B')`	0 ≥ 3? Hayır	−1	1	`A B . .`	`kuyruk[1] = B`
`kuyruk_ekle(&arka, 'C')`	1 ≥ 3? Hayır	−1	2	`A B C .`	`kuyruk[2] = C`
`kuyruk_al(&on, arka)`	−1 == 2? Hayır	0	2	`. B C .`	A döner

İşlem	Denetim	on	arka	Dizi [0..3]	Sonuç
`kuyruk_al(&on, arka)`	0 == 2? Hayır	1	2	`. . C .`	B döner
`kuyruk_ekle(&arka, 'D')`	2 ≥ 3? Hayır	1	3	`. . C D`	`kuyruk[3] = D`
`kuyruk_ekle(&arka, 'E')`	3 ≥ 3? **Evet**	1	3	`. . C D`	"Kuyruk Dolu!!!", program biter

Tablo 7.2: `MAX_QUEUE_SIZE = 4` iken doğrusal kuyruğun izlemesi. Dizi sütununda nokta, kuyrukta öğe tutmayan hücreyi gösterir.

İlk üç ekleme sorunsuzdur. Her birinde denetim yer olduğunu söyler, `arka` bir artar ve harf yeni hücreye yazılır. İki çıkarmada `on` önce 0'a, sonra 1'e ilerler ve sırasıyla A ile B döner. Burada önemli olan, çıkarmanın diziyi değiştirmemesidir. A ve B hâlâ 0 ve 1 numaralı hücrelerde yazılı durur, ama artık kuyruğun parçası değildir: kuyruk `on+1`'den başlar. Bu iki hücre fiilen boştur.

D eklendiğinde `arka` 3'e, yani dizinin son hücresine ulaşır. Kuyrukta yalnız iki öğe (C ve D) vardır; öğe sayısı `arka - on = 3 - 1 = 2`'dir. Buna karşın E eklenmek istendiğinde denetim `3 ≥ 3` sonucunu verir ve kod kuyruğu dolu sayar. Program "Kuyruk Dolu!!!" yazıp biter. Oysa dört hücreden ikisi boştur. Bu duruma **sahte doluluk** denir. Sorunun kaynağı, iki indisin de yalnız ileri gitmesidir. `on` ilerledikçe arkada bıraktığı hücreler bir daha hiç kullanılmaz. Doğrusal kuyruk, ömrü boyunca toplam `MAX_QUEUE_SIZE` ekleme yapabilir; aynı anda kaç öğe tuttuğu fark etmez. Örneğin 100 hücreli bir kuyruğa 60 öğe eklenip 60'ı da alınırsa kuyruk boştur. Yine de en çok 40 ekleme daha sığar; 41. ekleme "dolu" hatası verir.



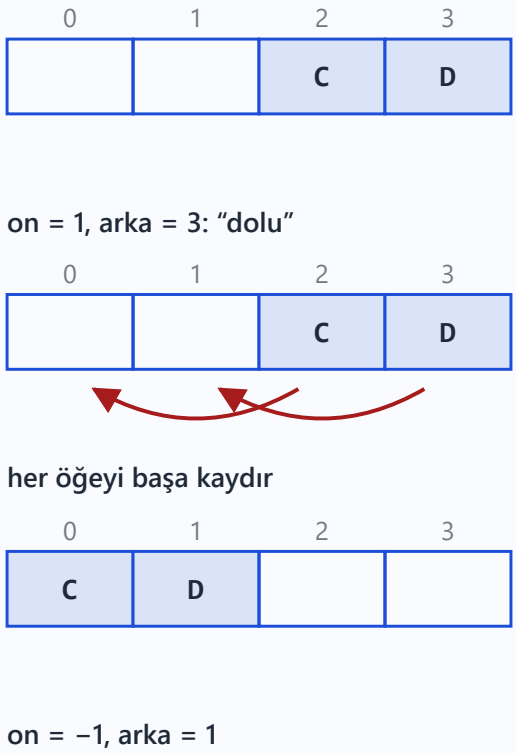
Şekil 7.2: Doğrusal kuyruğun izlemesinden dört kare. Başta iki indis de −1'dedir (soldaki kesik çizgili kutu, dizinin dışındaki −1 konumudur). A, B, C eklenince `arka = 2` olur. İki çıkarmadan sonra `on = 1`'dir; A ve B soluk yazılmıştır, çünkü kuyruğa ait değildir. D eklenince `arka = 3` olur ve sonraki ekleme, 0 ve 1 numaralı hücreler boş olduğu hâlde dolu sayılır.

Sık yapılan hata

Doğrusal kuyrukta `arka == MAX_QUEUE_SIZE - 1` koşulunu gerçek doluluk sanmak. Bu koşul yalnız "arkada yer kalmadı" anlamına gelir. Kuyrukta gerçekten kaç öğe olduğu `arka - on` ile bulunur ve bu sayı çok küçük olabilir. Kuyruk ancak `on == -1` iken `arka` sona dayanmışsa gerçekten doludur.

7.2.3 Kaydırma: pahalı çözüm

Sahte doluluğun ilk akla gelen çözümü, öğeleri dizinin başına kaydırmaktır. `arka` sona dayandığında önde boş hücre varsa bütün öğeler sola taşınır, `on` −1'e, `arka` da öğe sayısının bir eksiğine çekilir. Şekil 7.3 izlemedeki son durumdan başlar: `on = 1`, `arka = 3`, öğeler C ve D. Kaydırmadan sonra C 0 numaralı, D 1 numaralı hücreye geçer; `on = −1`, `arka = 1` olur ve arkada iki hücre açılır.



Şekil 7.3: Kaydırma. Üstte `on = 1`, `arka = 3` durumu “dolu” görünür; önde iki boş hücre vardır. Ortada her öğe bir kez başa doğru taşınır. Altta `on = −1`, `arka = 1` 'dir ve arkada yer açılmıştır.

Kaydırmayı kuyruk koduna eklemek kolaydır. Aşağıdaki iki fonksiyon önceki kesimdeki küresel `kuyruk`, `on` ve `arka` üzerinde çalışır:

```
void kaydir(void) {            /* öğeleri dizinin başına taşı */
    int n = arka - on;         /* öğe sayısı */
    for (int i = 0; i < n; i++)
        kuyruk[i] = kuyruk[on + 1 + i];
    on = -1;
    arka = n - 1;
}
void kuyruk_ekle_kaydir(element item) {
    if (arka == MAX_QUEUE_SIZE - 1) {
        if (on == -1) kuyruk_dolu();   /* gerçekten dolu */
        kaydir();                      /* önde yer var: O(n) */
    }
    kuyruk[++arka] = item;
}
```

`kaydir` önce öğe sayısını (`n = arka − on`) hesaplar. Döngü `i` 'nci öğeyi, yani `kuyruk[on+1+i]` hücresindeki öğeyi, `kuyruk[i]` 'ye taşır. Taşıma soldan sağa yapılır; hedef hücre her zaman kaynak hücrenin solunda olduğu için henüz okunmamış bir öğenin üzerine yazılmaz. Son iki atama indisleri yeni konuma çeker. `kuyruk_ekle_kaydir` yalnız `arka` sona dayandığında devreye girer. O anda `on` da −1 ise önde boş hücre yoktur ve kuyruk gerçekten doludur. Aksi hâlde önce kaydırma yapılır, sonra öğe her zamanki gibi eklenir. Bu sürüm sınandığında izlemedeki durum (C, D; `on = 1`, `arka = 3`) kaydırmayla `on = −1`, `arka = 1` 'e geçer. Ardından E ve F sığar, G ise gerçek doluluk nedeniyle reddedilir.

Çözüm doğrudur, ama bedeli yüksektir. Kaydırma kuyruktaki n öğenin her birini bir kez taşır; maliyeti $\Theta(n)$'dir. Kaydırma her eklemede yapılmaz, yalnız `arka` sona dayandığında yapılır. Ne var ki kuyruk sürekli neredeyse dolu çalışıyorsa (bir tampon için olağan bir durumdur) hemen her ekleme bir kaydırma tetikler. Kuyrukta $M - 1$ öğe varken bir ekleme ile bir çıkarma art arda yapılsın. Her çıkarma önde tek bir hücre açar. Bir sonraki eklemede `arka` yine sona dayanır ve $M - 1$ öğe taşınır. Böylece ekleme işlemi $O(1)$'den $O(n)$'ye çıkar.

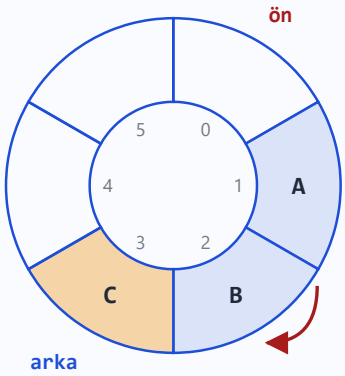
Bu tahmin ölçümle de doğrulandı. Kuyrukta sürekli $M - 1$ öğe tutulurken 200.000 ekleme ve çıkarma çifti yapıldı. $M = 1000$ için toplam taşıma sayısı 199.799.001, $M = 2000$ için 399.798.001'dir; yani her eklemede yaklaşık M öğe taşınmıştır. Süreler Tablo 7.15'te verilmiştir: kaydırmalı kuyruğun süresi M ile doğru orantılı artar, dairesel kuyruğun süresi ise değişmez. Sunumda bu durum hocanın bir sözüyle anlatılır: kaydırma, bilgisayar bilimlerinde yüksek maliyetin adıdır. Kaydırmadan kurtulmanın yolu, öğeleri yerinde bırakıp indislerin dönüp dolaşmasına izin vermektir. Bu yapıya **dairesel kuyruk** denir.

7.3 Dairesel kuyruk

Doğrusal kuyruğun sorunu, indislerin dizinin sonuna gelince durmasıdır. Dairesel kuyruk bu sınırı kaldırır: son hücreden sonra yeniden ilk hücreye geçilir. Böylece önde boşalan hücreler arkaya eklenen öğeler için kullanılır ve hiçbir öğe yerinden taşınmaz. Bu kesim fikri, onu mümkün kılan kalan işlemini, kodu ve dairesel kuyrukta dolu ile boşu ayırt etme sorununu sırasıyla ele alır.

7.3.1 Dizinin sonunu başına bağlamak

Altı hücreli bir dizi düz bir şerit yerine bir halka olarak düşünülsün. 0'dan 5'e kadar numaralanmış hücreler halkanın dilimleridir ve 5 numaralı dilimin hemen ardından yeniden 0 gelir. Bellekte hiçbir şey değişmez; dizi yine düz bir bloktur. Değişen yalnız indislerin ilerleme kuralıdır. Şekil 7.4 bu halkayı gösterir: kuyrukta A, B ve C vardır, `on` 0'ı, `arka` 3'ü gösterir.



Şekil 7.4: Altı hücreli dairesel dizi. İç halkadaki sayılar hücre indisleridir. `on = 0`, `arka = 3` 'tür; kuyrukta 1, 2 ve 3 numaralı hücrelerdeki A, B ve C vardır. 5 numaralı hücreden sonra 0 gelir.

Tanım 7.2

Dairesel kuyruk (*circular queue*), `MAX` hücreli bir dizide indislerin `MAX - 1` 'den sonra 0'a döndüğü kuyruk gerçeklemesidir. İndisler `arka = (arka + 1) % MAX` ve `on = (on + 1) % MAX` ile ilerler. Doğrusal kuyruktaki gibi `on` ilk öğenin bir öncesini, `arka` son öğeyi gösterir; başlangıçta `on = arka = 0` 'dır. Kuyruk `on == arka` iken **boştur**, `(arka + 1) % MAX == on` iken **doludur**.

Tanımdaki dolu koşulu şunu söyler: `arka` bir ilerlediğinde `on` 'un üzerine gelecekse yer yoktur. Bu koşulun bir sonucu, `MAX` hücreden birinin her zaman boş kalmasıdır. Kuyruk en çok `MAX - 1` öğe tutar. Neden böyle olduğu, yani son hücrenin neden doldurulmadığı, dolu ile boşu ayırt etme sorununun konusudur. Şimdilik şu kadarı yeterlidir: son hücre de doldurulsaydı `arka`, `on` 'a eşit olurdu. Bu da boş kuyruğun koşuluyla aynıdır ve kod dolu bir kuyruğu boş sanırdı.

7.3.2 Modüler aritmetik ve sarma

İndisin halka üzerinde dönmesini sağlayan araç, C'deki kalan işleci `%` 'dir. `a % m` ifadesi, `a` 'nın `m` 'ye bölümünden kalanı verir. Negatif olmayan `a` için sonuç her zaman 0 ile `m - 1` arasındadır. Sayıların bu biçimde bir m değerine göre kalanlarıyla hesaplanmasına **modüler aritmetik** (*modular arithmetic*) denir. Saat bunun en tanıdık örneğidir: 11'den üç saat sonrası 14 değil, $(11 + 3) \bmod 12 = 2$'dir.

Dairesel kuyrukta `arka`'nın bir sonraki değeri `(arka + 1) % MAX` ile hesaplanır. `arka` son hücrede değilse bu ifade sıradan `arka + 1`'e eşittir. `arka` son hücredeyse `arka + 1 = MAX` olur ve kalan 0'dır: indis başa döner. Bu olaya **sarma** (*wrap-around*) denir. Tablo 7.3 `MAX = 6` için dört ardışık adımı verir.

`arka`	`arka + 1`	`(arka + 1) % 6`	**Anlamı**
3	4	4	normal ilerleme
4	5	5	son hücre
5	6	0	**sarma**: başa dönüş
0	1	1	yeniden normal ilerleme

Tablo 7.3: `MAX = 6` iken `arka`'nın ilerlemesi.

Kalan işleciyle ilgili C'ye özgü bir tuzak vardır. C standardı (C99'dan beri) tam sayı bölmesini sıfıra doğru keser. Bunun sonucu olarak negatif bir sayının kalanı da negatiftir: `-1 % 6` ifadesinin değeri 5 değil, −1'dir; `-2 % 6` ise −2'dir. Matematikteki *mod* işlemi her zaman negatif olmayan bir sonuç verir, C'nin `%` işleci vermez. İndisi bir *geri* almak gerektiğinde bu fark önem kazanır. `(i - 1) % MAX` ifadesi `i = 0` için −1 verir ve dizinin dışına, `kuyruk[-1]`'e erişilir. Doğru yazım `(i - 1 + MAX) % MAX`'tır. `MAX` eklemek değeri negatif olmaktan kurtarır, kalan işlemi de sonucu yine 0 ile `MAX - 1` arasına indirir: `(0 - 1 + 6) % 6 = 5`. Kuyrukta indis yalnız ileri gider; geri adım, iki uçtan da çalışan çift uçlu kuyrukta gerekir.

Modüler aritmetiğin ikinci kullanımı öğe sayısını bulmaktır. Doğrusal kuyrukta öğe sayısı `arka - on` idi. Dairesel kuyrukta `arka` sarıp `on`'un gerisine düşebilir; o zaman fark negatif olur. Aynı düzeltme burada da işe yarar:

Örnek 7.1

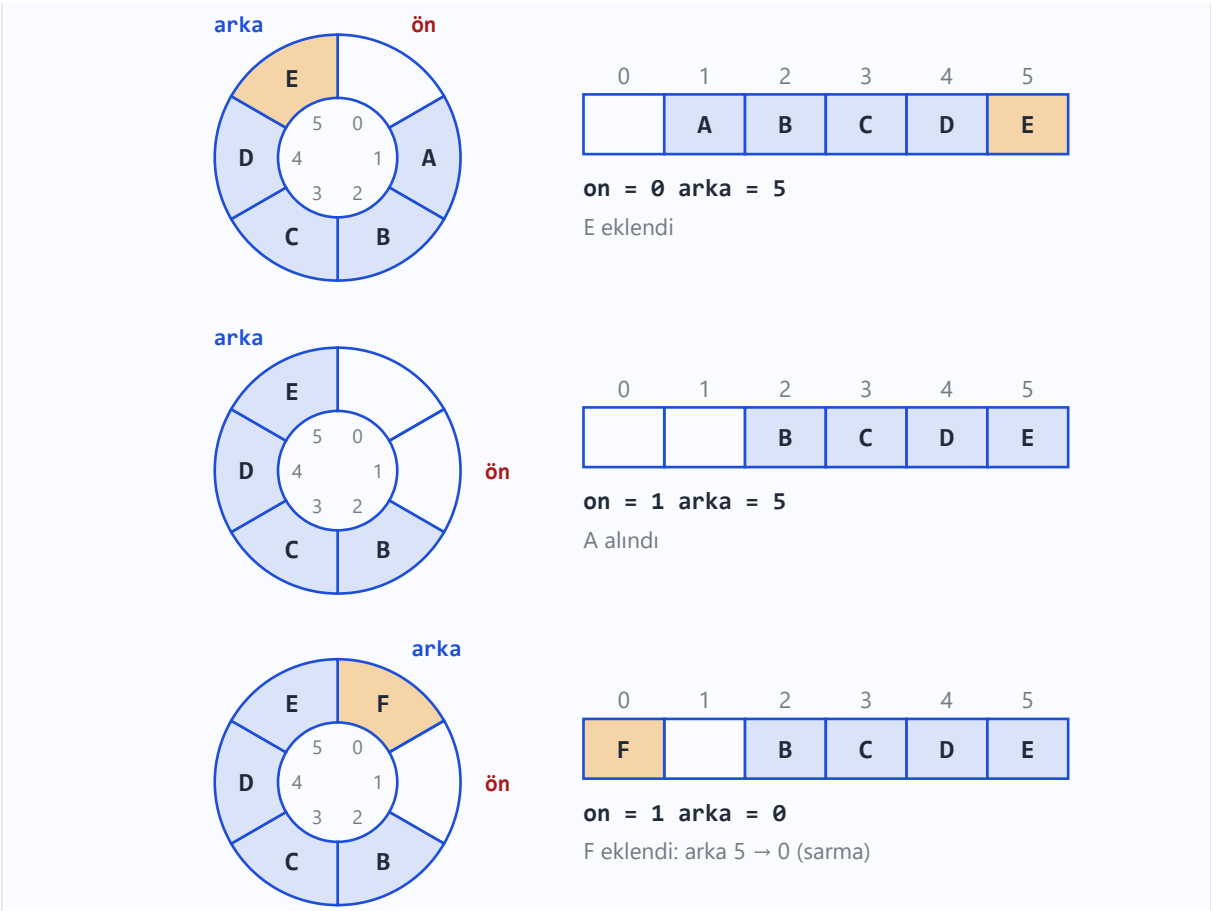
`MAX = 6`, `on = 4`, `arka = 1` olsun. Öğe sayısı `(arka - on + MAX) % MAX = (1 - 4 + 6) % 6 = 3 % 6 = 3`'tür. Öğeler `on`'un bir sonrasından `arka`'ya kadar, sarmayı da hesaba katarak sayılır: 5, 0 ve 1 numaralı hücreler. `arka - on` farkı ise −3 verirdi ve yanlış olurdu. `on` ile `arka` eşitse formül 0 verir; bu da boş kuyruğun koşuluyla tutarlıdır.

7.3.3 A...F ile sarma

Sarmanın kuyrukta nasıl işlediğini adım adım görmek için altı hücreli bir dairesel kuyruk ele alınsın. Başlangıçta `on = arka = 0`'dır. A, B, C, D ve E sırasıyla eklenir. Her eklemede `arka` önce ilerler, sonra öğe yeni hücreye yazılır. Bu yüzden A 0 numaralı hücreye değil, 1 numaralı hücreye girer. E eklendikten sonra `arka = 5`'tir ve 0 numaralı hücre boştur.

Bu noktada bir ekleme daha denenseydi `(5 + 1) % 6 = 0` değeri `on`'a eşit çıkardı ve kuyruk dolu sayılırdı. Altı hücrede beş öğe vardır; bir hücre bilerek boş bırakılmıştır. Sonra A alınır: `on = (0 + 1) % 6 = 1` olur ve A döner. Artık 0 ve 1 numaralı hücrelerin ikisi de boştur. Ardından F eklenir. `arka = (5 + 1) % 6 = 0` olur; F, önde boşalan 0 numaralı hücreye yazılır. Doğrusal kuyruğun kullanamadığı hücre burada arkadaymış gibi kullanılmıştır. Kuyruk şimdi yine doludur: `(0 + 1) % 6 = 1 = on`. Kuyrukta B, C, D, E, F vardır ve sıraları korunmuştur. Bir sonraki çıkarma B'yi verir, çünkü `on` 1'den 2'ye ilerler.





Şekil 7.5: Altı hücreli dairesel kuyrukta sarma. Her karede solda halka, sağda aynı dizinin düz görünümü vardır. Üstten alta: boş kuyruk (`on = arka = 0`); A...E eklenmiş (`arka = 5`); A alınmış (`on = 1`); F, `arka = (5 + 1) % 6 = 0` hücresine yazılmış. Son durumda kuyruk doludur: altı hücrede beş öğe.

7.3.4 C kodu ve ince bir nokta

Dairesel kuyruğun kodu, doğrusal kuyruğun kodundan yalnız iki noktada ayrılır: indisler `% MAX_QUEUE_SIZE` ile ilerler ve dolu koşulu değişir. Aşağıdaki sürüm yine Horowitz, Sahni ve Anderson-Freed'in kitabındaki gerçeklemeye dayanır:

```
void kuyruk_ekle(int on, int *arka, element item) {
    *arka = (*arka + 1) % MAX_QUEUE_SIZE; /* ilerlet */
    if (on == *arka) kuyruk_dolu();       /* dolu mu? */
    kuyruk[*arka] = item;                 /* yerleştir */
}
element kuyruk_al(int *on, int arka) {
    if (*on == arka) kuyruk_bos();        /* boş mu? */
    *on = (*on + 1) % MAX_QUEUE_SIZE;     /* ilerlet */
    return kuyruk[*on];                   /* döndür */
}
```

`kuyruk_ekle` artık `on` 'u da parametre olarak alır, çünkü dolu koşulu `on` 'a bağlıdır. Fonksiyon önce `*arka` 'yı sarmalı biçimde bir ilerletir. İlerlemiş `arka` `on` 'a eşitse kuyruk doludur ve `kuyruk_dolu` çağrılır. Değilse öğe yeni hücreye yazılır. `kuyruk_al` doğrusal sürümdeki gibi önce boşluğa bakar, sonra `*on` 'u sarmalı biçimde ilerletip o hücredeki öğeyi döndürür. Her iki fonksiyon da sabit sayıda işlem yapar ve hiçbir öğeyi taşımaz; ikisinin de maliyeti *O(1)*'dir.

Bu koddaki ince nokta, `kuyruk_ekle` 'nin önce `*arka` 'yı ilerletmesi, doluluğa *sonra* bakmasıdır. Kuyruk doluysa `*arka` çoktan `on` 'a eşitlenmiştir. Sunumdaki sürümde bunun zararı yoktur, çünkü `kuyruk_dolu` programı bitirir ve bozulan değer bir daha kullanılmaz. Hata durumunda programı bitirmeyen, yalnız uyarı verip dönen bir sürüm yazıldığında ise sorun ortaya çıkar. Kuyruk dolu iken yapılan bir ekle-

me denemesinden sonra `on == arka` olur ve kuyruk **boş** görünür. İçindeki bütün öğeler bir anda kaybolmuştur. Bu davranış küçük bir deneyle doğrulandı: altı hücreli kuyruğa A…E eklendikten sonra (`on = 0`, `arka = 5`) F eklenmek istendi. Uyarı yazılıp dönüldükten sonra `arka = 0` ve `on == arka` oldu; kuyruk boş göründü.

Kitaptaki gerçekleme bu yüzden dolu durumunda `arka`'yı eski değerine geri alır. Daha temiz bir yol, yeni konumu önce geçici bir değişkende hesaplamak ve `arka`'yı ancak yer olduğu kesinleşince değiştirmektir. Bu yaklaşım bir genel ilkenin örneğidir: **başarısız olabilen bir işlem, başarısız olduğunda veri yapısını değiştirmemelidir**.

Sunumda bu slaytın yanında hocanın bir sözü de yer alır. Söz, iyi bir üniversitede okumanın güvence değil sorumluluk olduğunu anlatır: öğrenci, başaramazsa okuldan atılacağını bilerek çalışmalıdır. Aynı tutum kod yazarken de geçerlidir. Bir kodun “genellikle çalışması” yetmez; dolu kuyruğa ekleme gibi kenar durumlar da bilerek düşünülmelidir.

7.3.5 Güvenli sürüm ve kod izleme

Aşağıdaki sürüm programı bitirmez. Hata durumunda `false`, başarıda `true` döndürür ve kararı çağırana bırakır. `kuyruk_al` çıkan öğeyi dönüş değeriyle değil, bir çıkış parametresiyle (`element *x`) verir; çünkü dönüş değeri başarı bilgisine ayrılmıştır. `bool` tipi için `<stdbool.h>` eklenmelidir.

```
bool kuyruk_ekle(int on, int *arka, element x) {
  int yeni = (*arka + 1) % MAX_QUEUE_SIZE;
  if (yeni == on) return false;       /* dolu */
  *arka = yeni;  kuyruk[yeni] = x;
  return true;
}
bool kuyruk_al(int *on, int arka, element *x) {
  if (*on == arka) return false;      /* boş */
  *on = (*on + 1) % MAX_QUEUE_SIZE;
  *x = kuyruk[*on];  return true;
}
```

`kuyruk_ekle` yeni konumu `yeni` adlı yerel değişkende hesaplar. `yeni == on` ise kuyruk doludur ve fonksiyon `arka`'ya hiç dokunmadan `false` döner. Yer varsa önce `*arka = yeni` yapılır, sonra öğe yazılır. `kuyruk_al` boş kuyrukta `false` döner. Aksi hâlde `on`'u ilerletir, öğeyi `*x` üzerinden çağırana iletir ve `true` döner. Çağıran taraf `if (!kuyruk_ekle(on, &arka, x))` gibi bir denetimle hatayı yakalayabilir.

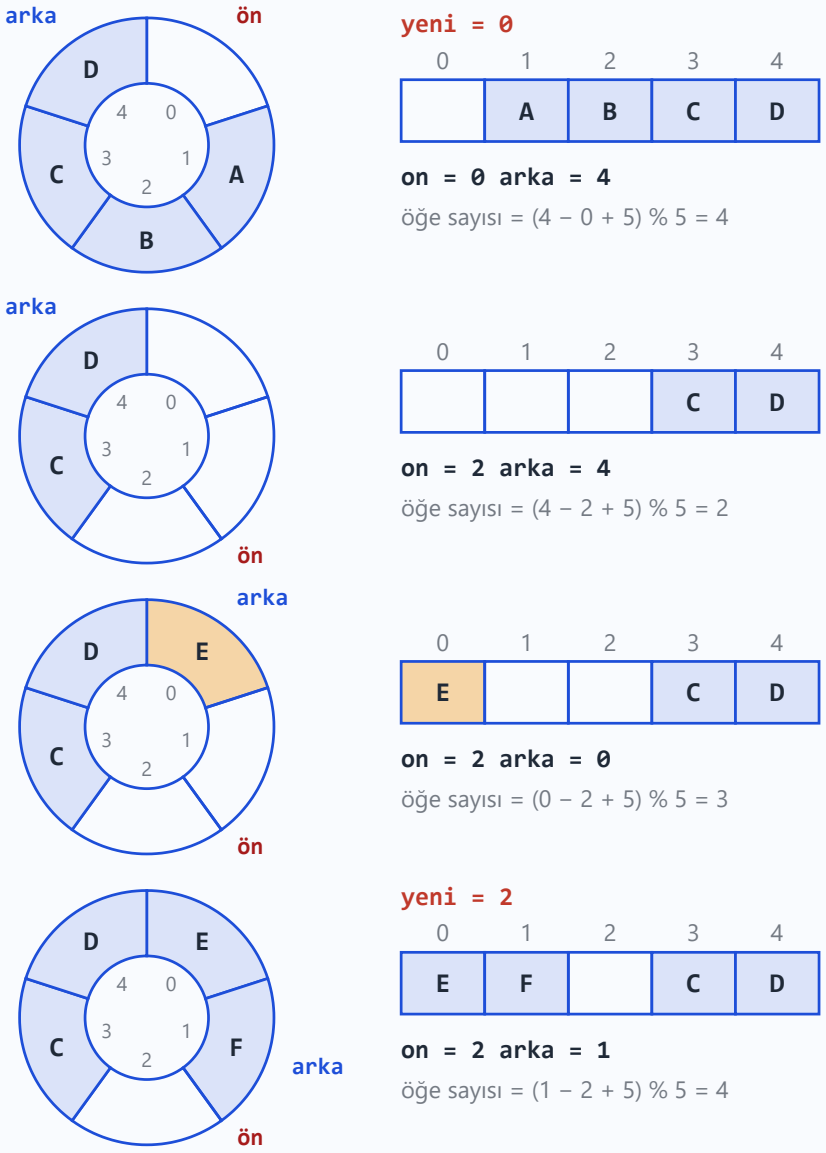
> **Örnek 7.2**
>
> `MAX_QUEUE_SIZE = 5`, başlangıçta `on = arka = 0`. Sırasıyla A, B, C, D ve E eklenmek istenir; iki öğe alınır; sonra E, F ve G eklenmek istenir. Tablo 7.4 her çağrının denetimini ve sonucunu verir.

Çağrı	Denetim	`on`	`arka`	Dizi [0..4]	Dönüş
Başlangıç	—	0	0	`. . . . .`	—
ekle A	`yeni` = 1 ≠ 0	0	1	`. A . . .`	`true`
ekle B	`yeni` = 2 ≠ 0	0	2	`. A B . .`	`true`
ekle C	`yeni` = 3 ≠ 0	0	3	`. A B C .`	`true`
ekle D	`yeni` = 4 ≠ 0	0	4	`. A B C D`	`true`
ekle E	`yeni` = (4 + 1) % 5 = 0 = `on`	0	4	`. A B C D`	`false` (dolu)
al	0 == 4? Hayır	1	4	`. . B C D`	`true`, `*x` = A
al	1 == 4? Hayır	2	4	`. . . C D`	`true`, `*x` = B
ekle E	`yeni` = (4 + 1) % 5 = 0 ≠ 2	2	0	`E . . C D`	`true` (sarma)
ekle F	`yeni` = 1 ≠ 2	2	1	`E F . C D`	`true`

Çağrı	Denetim	`on`	`arka`	Dizi [0..4]	Dönüş
ekle G	`yeni` = 2 = `on`	2	1	`E F . C D`	`false` (dolu)

Tablo 7.4: Güvenli dairesel kuyruğun izlemesi (5 hücre). Dizi sütununda nokta, kuyrukta öğe tutmayan hücreyi gösterir.

Tablonun iki satırı özellikle önemlidir. İlk E denemesinde kuyrukta dört öğe vardır ve beş hücreden biri (0 numaralı) boştur. Yine de `yeni` 0 olur ve `on`'a eşittir; kuyruk dolu sayılır. Güvenli sürüm burada `arka`'yı 4'te bırakır. Kuyruk bozulmamıştır ve iki çıkarma A ile B'yi doğru sırayla verir. İkinci E denemesinde `arka` 4'ten 0'a sarar ve E, doğrusal kuyruğun asla kullanamayacağı 0 numaralı hücreye yazılır. G denemesinde yine dört öğe vardır (C, D, E, F) ve boş kalan 2 numaralı hücre, `on`'un gösterdiği hücredir. Bu durumdaki kuyruğu boşaltan çıkarmalar C, D, E, F sırasını verir; bu da ilk giren ilk çıkar kuralıyla tutarlıdır.



Şekil 7.6: Güvenli dairesel kuyruk izlemesinden dört kare. Üstten alta: A...D eklendikten sonra E denemesi (`yeni = 0 = on`, dolu); iki çıkarmadan sonra `on = 2`; E'nin sarmayla 0 numaralı hücreye yazılması; G denemesinde yeniden doluluk (`yeni = 2 = on`). Her karede "öğe sayısı" satırı `(arka − on + 5) % 5` formülünü uygular.

Sık yapılan hata

Dairesel kuyrukta `% MAX_QUEUE_SIZE`'ı bir yerde unutmak, örneğin `kuyruk_al`'da `*on = *on + 1` yazmak. Program ilk sarmaya kadar doğru çalışır. `on` son hücreden sonra `MAX_QUEUE_SIZE` değerini alınca dizinin dışındaki bir adres okunur. Bu tanımsız davranıştır ve hata çoğu zaman ancak uzun bir çalışmadan sonra, ilgisiz bir yerde belirti verir. İndisi ilerleten her satırda kalan işlemi bulunmalıdır.

7.3.6 Dolu mu, boş mu? Üç yöntem

Önceki iki izlemede kuyruk bir hücresi boşken dolu sayıldı. Bu bir dikkatsizlik değil, zorunluluktur. Yalnız `on` ve `arka` ile dolu ve boş kuyruklar birbirinden ayırt edilemez. Durumu kesin biçimde ifade etmek için sayma yeter.

Teorem 7.1 (İki indis yetmez)

`MAX` hücreli bir dairesel kuyruğun durumu yalnız `on` ve `arka` indisleriyle temsil ediliyorsa, kuyruğun 0, 1, …, `MAX` öğe tuttuğu `MAX + 1` durumun hepsini birbirinden ayırt etmek mümkün değildir. En az iki farklı öğe sayısı aynı indis ilişkisine düşer.

Kanıt. Kuyruktaki öğe sayısı yalnız iki indisin farkına bağlıdır: öğeler `on`'un bir sonrasından başlayıp `arka`'ya kadar uzanır. Fark `(arka - on + MAX) % MAX` biçiminde ölçülür ve bu ifade yalnız 0, 1, …, `MAX - 1` değerlerini alabilir; yani `MAX` farklı değer vardır. Öğe sayısı ise 0'dan `MAX`'a kadar `MAX + 1` farklı değer alabilir. `MAX + 1` durumu `MAX` kutuya dağıtmak gerekir. Güvercin yuvası ilkesine göre en az bir kutuya iki durum düşer. Somut olarak, boş kuyruk ile `MAX` öğeli dolu kuyruk aynı farkı (0) verir: her ikisinde de `on == arka`'dır. ■

Teorem iki çıkış yolu bırakır. Ya ayırt edilemeyen durumlardan biri kullanılmaz ya da indislerin yanında ek bilgi tutulur. Tablo 7.5 bu iki yolun üç uygulamasını karşılaştırır.

Yöntem	Boş	Dolu	Kapasite	Bedeli
1. Bir hücreyi boş bırakmak	`on == arka`	`(arka+1)%MAX == on`	`MAX - 1`	Bir hücre israf edilir.
2. Sayaç tutmak	`sayac == 0`	`sayac == MAX`	`MAX`	Her işlemde sayaç güncellenir.
3. Bayrak tutmak	`on == arka` ve son işlem alma	`on == arka` ve son işlem ekleme	`MAX`	Her işlemde bayrak güncellenir.

Tablo 7.5: Dairesel kuyrukta dolu ve boş durumunu ayırt etmenin üç yöntemi.

Birinci yöntem buraya kadarki kodların kullandığı yöntemdir. `MAX` öğeli durumdan vazgeçilir, böylece `on == arka` yalnız boş kuyruğu anlatır. Bedeli tek bir hücredir. Öğeler büyük yapılar ise bu hücre kayda değer bir bellek demektir, ama çoğu uygulamada önemsizdir. Kodun sadeliği bu bedeli karşılar.

İkinci yöntem öğe sayısını ayrı bir `sayac` değişkeninde tutar. Boş ve dolu doğrudan bu sayaçtan okunur ve `MAX` hücrenin hepsi kullanılır. Sayacın bir yan yararı daha vardır: öğe sayısı her an hazırdır ve *O(1)*'de okunur. Üstelik sayaç varken `arka`'yı tutmaya gerek kalmaz. Son öğenin yeri `(on + sayac - 1) % KAP` ile, bir sonraki boş hücre ise `(on + sayac) % KAP` ile hesaplanır.

Üçüncü yöntem yalnız bir bitlik bilgi ekler: son yapılan işlemin ekleme mi alma mı olduğu. `on == arka` durumu ancak iki biçimde ortaya çıkar. Ya bir alma işlemi kuyruğun son öğesini çıkarmıştır ya da bir ekleme işlemi son boş hücreyi doldurmuştur. Bayrak bu iki olasılığı birbirinden ayırır. Bayrak yönteminin kodu alıştırmalar kesiminde verilir.

7.3.7 Sayaçlı dairesel kuyruk

Sayaçlı sürüm kuyruğun bütün durumunu bir `struct` içinde toplar. Böylece birden çok kuyruk aynı programda kullanılabilir ve küresel değişkene gerek kalmaz. Yapının tanımı `typedef struct { int veri[KAP]; int on, sayac; } Kuyruk;` biçimindedir. Bu sürümde `on`, bir öncesini değil, **ilk öğenin kendisini** gösterir; boş bir kuyruk `on = 0`, `sayac = 0` ile başlar. İşlemler şöyledir:

```c
bool ekle(Kuyruk *k, int x) {
  if (k->sayac == KAP) return false;       /* dolu */
  k->veri[(k->on + k->sayac) % KAP] = x;
  k->sayac++;
  return true;
}
bool al(Kuyruk *k, int *x) {
  if (k->sayac == 0) return false;         /* boş */
  *x = k->veri[k->on];
  k->on = (k->on + 1) % KAP;
  k->sayac--;
  return true;
}
```

`ekle` önce doluluğa bakar: `sayac == KAP` ise yer yoktur. Yer varsa yeni öğe, son öğenin bir sonrasına, yani `(on + sayac) % KAP` hücresine yazılır ve sayaç bir artar. `al` boş kuyrukta `false` döner. Boş değilse öndeki öğeyi `*x` 'e kopyalar, `on` 'u sarmalı biçimde bir ilerletir ve sayacı bir azaltır. İki işlem de sabit iş yapar; maliyetleri *O(1)*'dir. Fonksiyonlar kuyruğu `Kuyruk *k` işaretçisiyle alır. Yapı değere göre aktarılsaydı fonksiyon yapının bir kopyası üzerinde çalışır ve çağıranın kuyruğu değişmezdi. `k->sayac` yazımı, `(*k).sayac` 'ın kısaltmasıdır.

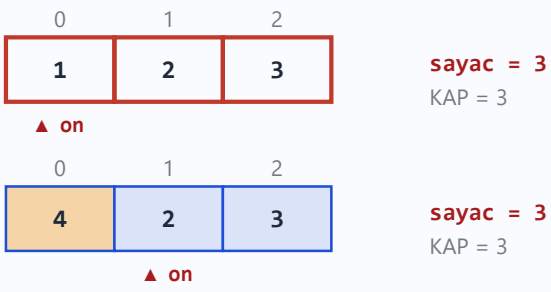
Örnek 7.3

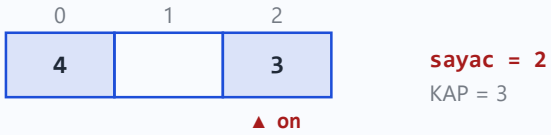
`KAP = 3`, `on = 0`, `sayac = 0`. Sırasıyla 1, 2, 3 ve 4 eklenmek istenir; bir öğe alınır; 4 eklenir; bir öğe daha alınır. Tablo 7.6 her adımı verir.

Çağrı	Hesap	on	sayac	veri[0..2]	Dönüş
Başlangıç	—	0	0	`. . .`	—
`ekle(&k, 1)`	konum = (0 + 0) % 3 = 0	0	1	`1 . .`	`true`
`ekle(&k, 2)`	konum = (0 + 1) % 3 = 1	0	2	`1 2 .`	`true`
`ekle(&k, 3)`	konum = (0 + 2) % 3 = 2	0	3	`1 2 3`	`true`
`ekle(&k, 4)`	`sayac == KAP`	0	3	`1 2 3`	`false` (dolu)
`al(&k, &x)`	`*x = veri[0]`	1	2	`. 2 3`	`true`, `x` = 1
`ekle(&k, 4)`	konum = (1 + 2) % 3 = 0	1	3	`4 2 3`	`true` (sarma)
`al(&k, &x)`	`*x = veri[1]`	2	2	`4 . 3`	`true`, `x` = 2

Tablo 7.6: Sayaçlı dairesel kuyruğun izlemesi (`KAP = 3`).

Bu kez üç hücrenin üçü de dolar ve hiçbir hücre israf edilmez. Dördüncü ekleme ancak sayaç `KAP` 'a ulaştığı için reddedilir. Bir çıkarmadan sonra eklenen 4, sarmayla 0 numaralı hücreye yazılır. Son durumda kuyruğun mantıksal sırası `on` 'dan başlayarak 3, 4'tür: 3 ikinci hücrede, 4 sıfırıncı hücrededir. Dizideki fiziksel sıra (4, ·, 3) ile kuyruğun sırası farklıdır. Dairesel kuyrukta sıra hiçbir zaman dizinin düz okunuşundan çıkarılmaz; her zaman `on` 'dan başlayıp sarılarak okunur.





Şekil 7.7: Sayaçlı dairesel kuyruk (`KAP = 3`). Üstte üç hücrenin üçü de doludur ve dördüncü ekleme reddedilir. Ortada 1 alınmış, 4 sarmayla 0 numaralı hücreye yazılmıştır. Altta 2 de alınmıştır: `on = 2` , `sayac = 2` .

Sık yapılan hata

İki farklı gösterimi karıştırmak. Sunumdaki dizi tabanlı kodlarda `on` , ilk öğenin bir öncesini gösterir ve işlemler "önce ilerlet, sonra kullan" sırasıyla çalışır. Sayaçlı sürümde `on` ilk öğenin kendisidir ve `al` "önce kullan, sonra ilerlet" sırasıyla çalışır. Bir gösterimin koduyla öbürünün başlangıç değerleri birlikte kullanılırsa kuyruk ilk öğeyi atlar ya da var olmayan bir öğe döndürür. Her gerçeklemede indislerin ne gösterdiği kodun başında bir yorumla yazılmalıdır.

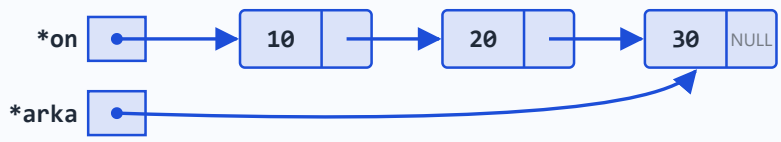
7.4 Bağlı kuyruk

Dizi tabanlı her kuyruğun bir kapasitesi vardır. Kapasite programı yazarken seçilir ve sonradan değişmez. Kuyruğa en çok kaç öğe gireceği önceden bilinmiyorsa iki kötü seçenek kalır: büyük bir dizi ayırıp belleği boşa harcamak ya da küçük bir dizi ayırıp taşma riskini almak. Bağlı liste bu ikilemi ortadan kaldırır. Her öğe kendi düğümünde durur, düğüm gerektiğinde `malloc` ile ayrılır, işi bitince `free` ile iade edilir.

7.4.1 Bağlı liste ile kuyruk

Bağlı listenin her düğümü bir değer ve bir sonraki düğümün adresini (bağ) taşır. Düğüm tipi bu notta `typedef struct dugum { int deger; struct dugum *bag; } dugum;` biçiminde tanımlanır. Yapının içinde kendi tipinden bir işaretçi bulunur; bu yüzden `struct dugum` adı `typedef` bitmeden kullanılır. Son düğümün bağı `NULL` 'dır.

Tek yönlü bağlı listenin başına eklemek ve başından silmek kolaydır, çünkü ilk düğüm elde tutulur. Sona eklemek için ise son düğüme ulaşmak gerekir. Yalnız ilk düğümün adresi tutulursa son düğüm ancak bütün liste baştan sona yürünerek bulunur ve bu *O(n)* sürer. Kuyrukta ekleme bir uçtan, çıkarma öbür uçtan yapıldığı için iki ucun da elde olması gerekir. Bu yüzden bağlı kuyruk iki işaretçiyle temsil edilir: `on` ilk düğümü, `arka` son düğümü gösterir. Boş kuyrukta ikisi de `NULL` 'dır.



Şekil 7.8: 10, 20 ve 30'dan oluşan bağlı kuyruk. `*on` ilk düğümü (10), `*arka` son düğümü (30) gösterir. Son düğümün bağı `NULL` 'dır. Yıldızlı adlar, işaretçilerin fonksiyona adresle aktarıldığını hatırlatır.

Hangi ucun ön, hangisinin arka olacağı rastgele seçilmez. Ekleme listenin *sonuna* yapılır: `arka->bag = yeni; arka = yeni;` ve iş biter. Çıkarma listenin *başından* yapılır: `on = on->bag;` ile ilk düğüm atlanır. İki işlem de sabit sayıda atama yapar ve *O(1)*'dir. Tersi seçilseydi, yani başa ekleyip sondan silinseydi, ekleme yine kolay olurdu ama silme zorlaşırdı. Son düğümü silmek için sondan bir önceki düğümün bağını `NULL` yapmak gerekir. Tek yönlü listede bir düğümden öncekine geri dönülemez; sondan bir önceki düğüm de ancak baştan yürünerek, *O(n)* sürede bulunur. Kural kısaca şöyledir: tek yönlü bağlı kuyrukta **sona eklenir, baştan silinir**.

Bu kesim işaretçi bilgisine dayanır. Sunumda hocanın bir anısı, öğrencinin bu konudaki duygusunu iyi anlatır. Hoca iki dersi işaretçilere ayırmıştır. Bir öğrenci işaretçileri iyi bildiğini, ama anlatılanı yine de anlamadığını söyler. Hoca da bunun olabileceğini kabul eder. İşaretçili kodu kavramanın en güvenli yolu, her atamadan sonra kutu ve okları kâğıda çizmektir. Aşağıdaki iki izleme bu çizimi adım adım yapar.

7.4.2 Kod izleme: `kuyruk_ekle`

Ekleme fonksiyonu kuyruğun iki işaretçisini de değiştirebilir. Boş kuyruğa yapılan ilk eklemede hem `on` hem `arka` yeni düğümü göstermelidir. Bu yüzden iki işaretçi de adresle, yani işaretçinin işaretçisi olarak (`dugum **`) aktarılır. Çağrı `kuyruk_ekle(&on, &arka, 10)` biçimindedir.

```
bool kuyruk_ekle(dugum **on, dugum **arka, int x) {
  dugum *yeni = malloc(sizeof *yeni);
  if (yeni == NULL) return false;
  yeni->deger = x;  yeni->bag = NULL;
  if (*arka == NULL) *on = yeni;   /* boş kuyruk */
  else (*arka)->bag = yeni;        /* sona bağla */
  *arka = yeni;
  return true;
}
```

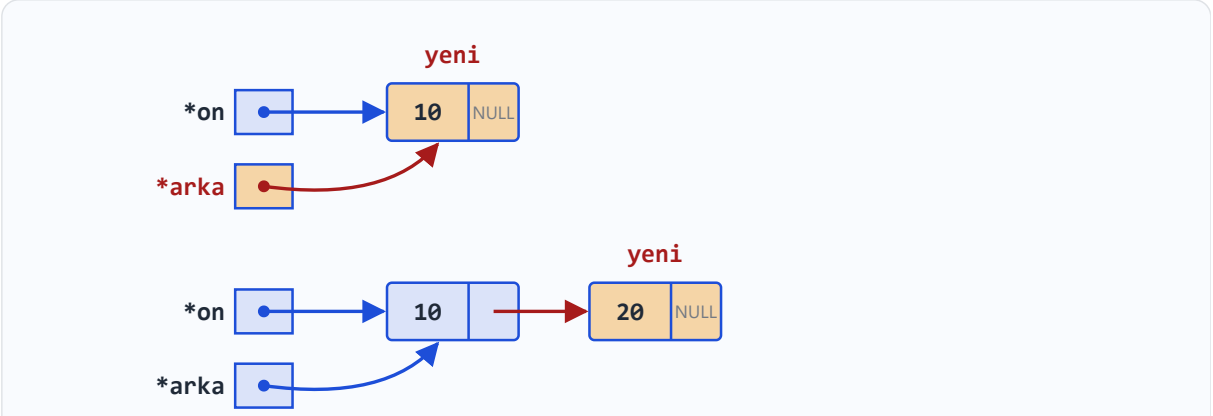
İlk satır yeni düğüm için bellek ayırır. `sizeof *yeni` , `yeni` 'nin gösterdiği tipin, yani `dugum` 'un boyutudur. Tip adını tekrar yazmamak, sonradan tip değişirse kodu güncel tutar. `malloc` bellek bulamazsa `NULL` döner. Bu durumda fonksiyon kuyruğa dokunmadan `false` döndürür. Ardından düğümün alanları doldurulur. Yeni düğüm kuyruğun son düğümü olacağı için bağı `NULL` yapılır.

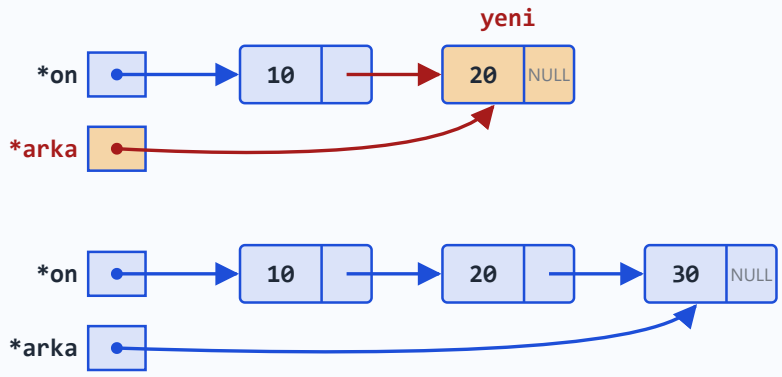
Sonraki `if` iki durumu ayırır. `*arka == NULL` ise kuyruk boştur; yeni düğüm aynı zamanda ilk düğümdür ve `*on = yeni` yapılır. Kuyruk boş değilse yeni düğüm eski son düğümün arkasına bağlanır: `(*arka)->bag = yeni` . Buradaki parantez gereklidir. `->` işleci `*` 'dan önce uygulanır; `*arka->bag` yazılsaydı `*(arka->bag)` olarak okunur ve derlenmezdi, çünkü `arka` bir yapıyı değil, bir işaretçiyi gösterir. Her iki durumda da son satır `*arka = yeni` ile arkayı yeni düğüme taşır.

Çağrı	Adım	`*on`	`*arka`	Kuyruk
Başlangıç	—	NULL	NULL	(boş)
`kuyruk_ekle(…, 10)`	`malloc` ; `deger = 10` , `bag = NULL`	NULL	NULL	(boş)
	`*arka == NULL` → `*on = yeni`	düğüm(10)	NULL	10
	`*arka = yeni`	düğüm(10)	düğüm(10)	10
`kuyruk_ekle(…, 20)`	`malloc` ; `deger = 20` , `bag = NULL`	düğüm(10)	düğüm(10)	10
	`(*arka)->bag = yeni` : 10 artık 20'yi gösterir	düğüm(10)	düğüm(10)	10 → 20
	`*arka = yeni`	düğüm(10)	düğüm(20)	10 → 20
`kuyruk_ekle(…, 30)`	`(*arka)->bag = yeni` : 20 artık 30'u gösterir	düğüm(10)	düğüm(20)	10 → 20 → 30
	`*arka = yeni`	düğüm(10)	düğüm(30)	10 → 20 → 30

Tablo 7.7: Boş bağlı kuyruğa 10, 20 ve 30 eklemenin izlemesi.

İzleme, `on` 'un yalnız ilk eklemede değiştiğini gösterir. Sonraki eklemelerde `on` hep 10'u gösterir; değişen, eski son düğümün bağı ve `arka` 'dır. Her ekleme bir `malloc` , birkaç atama ve bir karşılaştırma yapar. Kuyruğun uzunluğu hiçbir adımı etkilemez; ekleme $O(1)$'dir. `arka` işaretçisi tutulmasaydı son düğüm her eklemede baştan yürünerek bulunur ve n eklemenin toplam maliyeti $1 + 2 + \ldots + n = \Theta(n^2)$ olurdu.





Şekil 7.9: `kuyruk_ekle` izlemesinden dört kare. Üstten alta: 10 eklendikten sonra `*on` ve `*arka` aynı düğümü gösterir; 20 eklenirken önce 10'un bağı yeni düğüme çevrilir (kırmızı ok); sonra `*arka` 20'ye taşınır; 30 da eklendikten sonraki son durum.

Sık yapılan hata

Boş kuyruğa ilk eklemede `on`'u güncellemeyi unutmak, yani `if` dalını atlayıp her durumda `(*arka)->bag = yeni` yazmak. Kuyruk boşken `*arka` `NULL`'dır ve `NULL->bag`'e yazmak programı çökertir. Yalnız `NULL` denetimi eklenip `*on` atlanırsa da program çökmez, ama `on` `NULL` kalır. Kuyruk boş görünür ve eklenen öğeler hiçbir zaman alınamaz; hepsi sızar.

7.4.3 Kod izleme: `kuyruk_al`

Çıkarma işlemi öndeki düğümü listeden ayırır, değerini çağırana verir ve düğümün belleğini iade eder:

```
bool kuyruk_al(dugum **on, dugum **arka, int *x) {
  dugum *eski = *on;
  if (eski == NULL) return false;      /* boş */
  *x = eski->deger;
  *on = eski->bag;
  if (*on == NULL) *arka = NULL;  /* son öğe gitti */
  free(eski);
  return true;
}
```

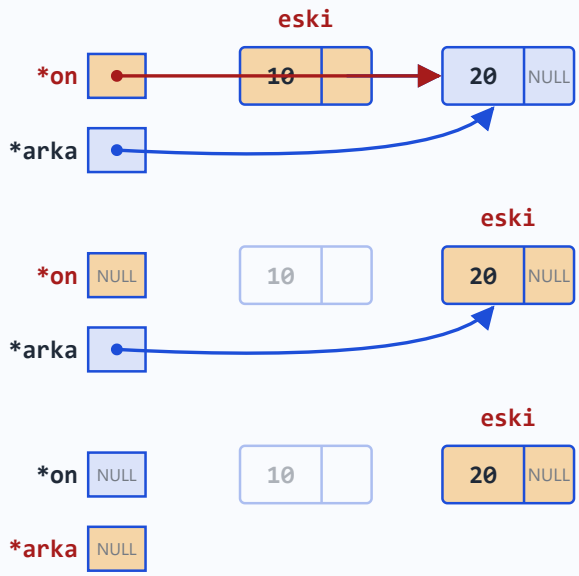
Fonksiyon önce öndeki düğümün adresini `eski` adlı yerel değişkende saklar. Bu saklama zorunludur. `*on` bir sonraki düğüme ilerletildikten sonra eski düğüme ulaşmanın başka yolu kalmaz; adresi saklanmadan ilerlenirse düğüm `free` edilemez ve sızar. `eski` `NULL` ise kuyruk boştur ve `false` döner. Değilse değer `*x`'e kopyalanır ve `*on = eski->bag` ile ön, ikinci düğüme geçer.

Sonraki satır bağlı kuyruğun en kolay unutulan ayrıntısıdır. Kuyruktaki son öğe çıkarıldıysa `*on` artık `NULL`'dır. Bu anda `*arka` hâlâ az sonra serbest bırakılacak düğümü gösterir. `*arka = NULL` yapılmazsa `arka` bir **sarkan işaretçi** (*dangling pointer*), yani artık geçerli olmayan bir bellek bölgesini gösteren işaretçi olur. Bir sonraki `kuyruk_ekle` çağrısı `*arka`'yı `NULL` görmez, kuyruğu boş saymaz ve serbest bırakılmış düğümün bağ alanına yazar. Bu tanımsız davranıştır. Üstelik `on` `NULL` kaldığı için yeni öğe de kuyrukta görünmez. Son olarak `free(eski)` ile düğüm iade edilir. `free`, değer okunduktan ve bağ izlendikten *sonra* çağrılmalıdır; serbest bırakılmış bir düğümün alanlarını okumak da tanımsız davranıştır.

Çağrı	Adım	`eski`	`*on`	`*arka`	Sonuç
1. `kuyruk_al`	`eski = *on`	düğüm(10)	düğüm(10)	düğüm(20)	—
	`*x = 10; *on = eski->bag`	düğüm(10)	düğüm(20)	düğüm(20)	`*on ≠ NULL`: arka değişmez
	`free(eski)`	—	düğüm(20)	düğüm(20)	`true`, 10
2. `kuyruk_al`	`eski = *on`	düğüm(20)	düğüm(20)	düğüm(20)	—
	`*x = 20; *on = eski->bag`	düğüm(20)	NULL	düğüm(20)	`*on == NULL`
	`*arka = NULL; free(eski)`	—	NULL	NULL	`true`, 20

Çağrı	Adım	`eski`	`*on`	`*arka`	Sonuç
3. `kuyruk_al`	`eski = *on = NULL`	NULL	NULL	NULL	`false` (boş)

Tablo 7.8: 10 → 20 kuyruğundan üç kez `kuyruk_al` çağrısının izlemesi.



Şekil 7.10: `kuyruk_al` izlemesinden üç kare. Üstte ilk çağrıda `*on` 20'ye geçmiştir; 10'un düğümü `eski` ile tutulur. Ortada ikinci çağrıda `*on` `NULL` olmuştur ama `*arka` hâlâ serbest bırakılacak düğümü gösterir. Altta `*arka = NULL` yapılmıştır; düğüm artık güvenle `free` edilebilir.

İki fonksiyon birlikte sınandı. Boş kuyruğa 10, 20, 30 eklenip hepsi alındığında değerler 10, 20, 30 sırasıyla çıktı ve son çıkarmadan sonra `on` ile `arka` 'nın ikisi de `NULL` oldu. Boşalan kuyruğa yeni bir ekleme (99) her iki işaretçiyi de doğru biçimde yeni düğüme yöneltti. 100.000 öğelik bir deneme de öğelerin ekleme sırasıyla çıktığını doğruladı. Programdaki `malloc` ve `free` çağrıları sayıldı: 100.006 ayırmanın her biri için bir iade yapıldı, sızıntı yoktur. Bu derleyici ve platformda bir düğüm 16 bayttır: 4 baytlık `int` , 4 baytlık hizalama boşluğu ve 8 baytlık işaretçi.

Sık yapılan hata

Bağlı kuyruktan son öğe alınınca `arka = NULL` yapmamak. Hata hemen belirti vermez; kuyruk boşaldıktan sonraki ilk eklemede ortaya çıkar. Program bazen çalışır gibi görünür, bazen ilgisiz bir yerde çöker. Bağlı kuyrukta iki işaretçinin değişmezi şudur: ya ikisi de `NULL` 'dır ya da ikisi de geçerli bir düğümü gösterir.

7.4.4 Dizi mi, bağlı liste mi?

Kuyruğun üç gerçeklemesi aynı soyut işlemleri sunar, ama maliyetleri ve kısıtları farklıdır. Tablo 7.9 bunları yan yana koyar.

Ölçüt	Doğrusal dizi	Dairesel dizi	Bağlı liste
Ekle / al	*O(1)*, ama "sahte dolu"	*O(1)*	*O(1)* (`malloc` / `free` ile)
Kaydırma gerekir mi?	Evet: *O(n)*	Hayır	Hayır
Kapasite	`MAX` (ömür boyu toplam ekleme)	`MAX - 1` (sayaçla `MAX`)	Bellek kadar
Ek bellek	Yok	Yok	Öğe başına bir işaretçi
Özel durum	`arka` sona dayanır	Sarma ve dolu/boş ayrımı	Boşalınca `arka = NULL`

Tablo 7.9: Kuyruk gerçeklemelerinin karşılaştırması.

Doğrusal kuyruk yalnız öğretim amacı taşır; gerçek programlarda kullanılmaz. Dairesel dizi ile bağlı liste arasındaki seçim ise uygulamaya bağlıdır. Dairesel dizi belleği tek blok olarak kullanır. Öğeler bellekte yan yana durduğu için işlemci önbelleğinden iyi yararlanır ve her işlemde `malloc` çağırmaz. Bağlı kuyruk ise üst sınır gerektirmez, ama her öğe için bir işaretçi taşır ve her ekleme bir bellek ayırma çağrısıdır. `int` tutan bir kuyrukta bu, öğe başına 16 bayt demektir; dizide 4 bayt yeter.

Pratikte en çok iki yapı kullanılır: sabit kapasiteli tamponlar için dairesel dizi ve üst sınırı bilinmeyen kuyruklar için **büyüyen dairesel dizi**. Büyüyen sürümde dizi dolunca iki katı büyüklükte yeni bir dizi ayrılır ve öğeler oraya taşınır. Taşıma `realloc` ile yapılamaz. `realloc` eski bloğu olduğu gibi kopyalar; kuyruk sarmış durumdaysa öğelerin bir kısmı dizinin sonunda, bir kısmı başındadır ve büyüyen dizide arada boş hücreler kalır. Öğeler bu yüzden `on`'dan başlayarak, mantıksal sırayla tek tek kopyalanır.

```c
typedef struct {
  int *veri;
  int kap, on, sayac;
} DKuyruk;

bool buyut(DKuyruk *k) {
  int yeni_kap = 2 * k->kap;
  int *yeni = malloc(yeni_kap * sizeof *yeni);
  if (yeni == NULL) return false;
  for (int i = 0; i < k->sayac; i++)              /* sırayı düzelterek kopyala */
    yeni[i] = k->veri[(k->on + i) % k->kap];
  free(k->veri);
  k->veri = yeni;  k->kap = yeni_kap;  k->on = 0;
  return true;
}
bool d_ekle(DKuyruk *k, int x) {
  if (k->sayac == k->kap && !buyut(k)) return false;
  k->veri[(k->on + k->sayac) % k->kap] = x;
  k->sayac++;
  return true;
}
```

`buyut` yeni diziyi ayırır, `i`'nci öğeyi `veri[(on + i) % kap]` hücresinden alıp yeni dizinin `i`'nci hücresine yazar, eski diziyi iade eder ve `on`'u 0'a çeker. `d_ekle` yalnız kuyruk doluyken büyütür; büyütme başarısız olursa kuyruğa dokunmadan `false` döner. Örneğin dört hücreli kuyruğa 1…4 eklenip iki öğe alındıktan sonra 5 ve 6 eklensin. Dizi fiziksel olarak 5, 6, 3, 4 içerir; `on = 2`'dir. 7 eklenince dizi sekiz hücreye büyür ve öğeler 3, 4, 5, 6 sırasıyla başa dizilir; 7 de beşinci hücreye girer. Sırayı bozan bir kopyalama, kuyruktan 3, 4, 5, 6, 7 yerine başka bir sıra çıkarırdı.

Büyütme tek başına $\Theta(n)$ sürer, ama seyrek yapılır. Kapasite her seferinde iki katına çıktığı için n eklemede kopyalanan toplam öğe sayısı $1 + 2 + 4 + \ldots < 2n$'dir. Böylece ekleme başına ortalama maliyet sabit kalır; ekleme **amortize** $O(1)$'dir. Deneyde kapasitesi 1'den başlayan kuyruğa bir milyon öğe eklendi: 20 büyütme yapıldı, toplam 1.048.575 öğe kopyalandı ve bu sayı $2n$'nin altında kaldı. Bütün öğeler ekleme sırasıyla çıktı; 23 `malloc` çağrısının hepsi `free` ile iade edildi.

Bağlı kuyruk, öğeler zaten düğüm olarak yaşıyorsa doğal seçimdir. Örneğin bir işletim sistemi, süreç tanımlayıcı yapılarını kendi bağ alanlarıyla kuyruğa dizer. Bu durumda kuyruk ayrı bir bellek ayırmaz; yalnız var olan bağları düzenler.

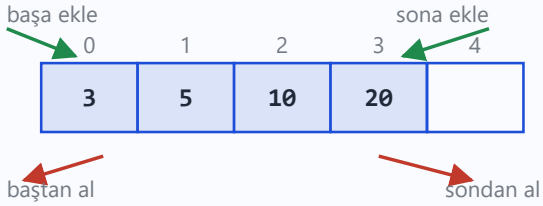
7.5 Deque ve uygulamalar

Kuyruk ve yığın, erişimi bir ya da iki uçla sınırlar. Bu kesim önce iki ucun da hem ekleme hem çıkarma için açık olduğu genel yapıyı, çift uçlu kuyruğu tanıtır. Ardından kuyruğun üç uygulamasına geçer: tek dizide birden çok kuyruk tutmak, işletim sistemlerindeki zaman dilimli zamanlama ve yalnız iki yığınla kuyruk kurmak.

7.5.1 Çift uçlu kuyruk

Tanım 7.3

Çift uçlu kuyruk ya da kısaca **deque** (*double-ended queue*, “dek” diye okunur), ekleme ve silmenin her iki uçta da yapılabildiği sıralı listedir. Dört temel işlemi vardır: başa ekleme, sona ekleme, baştan alma ve sondan alma. Uygun bir gerçeklemede dördü de $O(1)$ sürer.



Şekil 7.11: Deque'nin dört işlemi. Yeşil oklar eklemeyi, kırmızı oklar çıkarmayı gösterir; her iki uç da iki işleme açıktır.

Deque, yığın ile kuyruğu kapsayan genel bir yapıdır. Yalnız bir uç kullanılırsa, örneğin hep sona eklenip hep sondan alınırsa, deque bir yığın gibi davranır: 1, 2, 3 eklenince 3, 2, 1 çıkar. Bir uçtan eklenip öbür uçtan alınırsa kuyruk olur: aynı ekleme sırası 1, 2, 3 çıkışını verir. Bu iki davranış, aşağıdaki gerçeklemeyle sınanarak doğrulandı. İki kısıtlı türü de vardır. **Girdisi kısıtlı** deque'de ekleme yalnız bir uçta, silme iki uçta yapılır. **Çıktısı kısıtlı** deque'de ise ekleme iki uçta, silme yalnız bir uçta yapılır.

Deque birçok algoritmada iş görür. **Kayan pencere** sorusunda bir dizi üzerinde sabit genişlikte bir pencere kaydırılır ve her konumda penceredeki en büyük değer istenir. Deque, aday değerlerin indislerini tutar. Pencereden çıkan indis baştan, yeni gelenden küçük kalan adaylar sondan atılır. Böylece bütün dizi *O(n)* sürede işlenir. Çok çekirdekli zamanlayıcılarda kullanılan **iş çalma** (*work stealing*) yönteminde her çekirdek kendi işlerini bir deque'de tutar. Çekirdek kendi işlerini bir uçtan alır, boşta kalan başka bir çekirdek ise öbür uçtan iş "çalar". Metin düzenleyicilerdeki geri alma geçmişi de deque ile tutulabilir. Yeni işlemler sona eklenir ve geri alma sondan yapılır. Geçmiş belirli bir uzunluğu aşınca en eski kayıt baştan atılır.

7.5.2 Dairesel dizide deque

Deque, sayaçlı dairesel kuyruğa küçük bir eklemeyle kurulur. Yapı aynıdır: `typedef struct { int veri[KAP]; int on, sayac; } Deque;`. `on` ilk öğeyi gösterir, `sayac` öğe sayısıdır. Sona ekleme ve baştan alma, sayaçlı kuyruğun `ekle` ve `al` fonksiyonlarıyla aynıdır. Yeni olan iki işlem başa ekleme ve sondan almadır:

```
bool basa_ekle(Deque *d, int x) {
  if (d->sayac == KAP) return false;
  d->on = (d->on - 1 + KAP) % KAP;    /* bir geri */
  d->veri[d->on] = x;  d->sayac++;
  return true;
}
bool sondan_al(Deque *d, int *x) {
  if (d->sayac == 0) return false;
  *x = d->veri[(d->on + d->sayac - 1) % KAP];
  d->sayac--;  return true;
}
```

`basa_ekle` yeni öğeyi mevcut ilk öğenin bir gerisine koyar. Bunun için `on` bir geri alınır. Geri adım, modüler aritmetiğin negatif kalan tuzağına düşmemek için `(on - 1 + KAP) % KAP` ile hesaplanır. `on = 0` iken bu ifade `KAP - 1` verir: 0'ın bir gerisi dizinin son hücresidir. `sondan_al` son öğenin konumunu `(on + sayac - 1) % KAP` ile bulur, o hücredeki değeri okur ve yalnız sayacı azaltır. Son öğe çıkarıldığında `on` değişmez, çünkü ilk öğe yerinde kalmıştır. Bütünlük için öteki iki işlem de aşağıda verilmiştir:

```
bool sona_ekle(Deque *d, int x) {
  if (d->sayac == KAP) return false;
  d->veri[(d->on + d->sayac) % KAP] = x;   /* son öğenin bir sonrası */
  d->sayac++;
  return true;
}
bool bastan_al(Deque *d, int *x) {
  if (d->sayac == 0) return false;
  *x = d->veri[d->on];
  d->on = (d->on + 1) % KAP;            /* bir ileri */
  d->sayac--;
  return true;
}
```

Dört fonksiyonun hiçbiri döngü içermez; her biri birkaç aritmetik işlem ve bir dizi erişimi yapar. Hepsinin maliyeti *O(1)*'dir. Bellek maliyeti `KAP` öğedir.

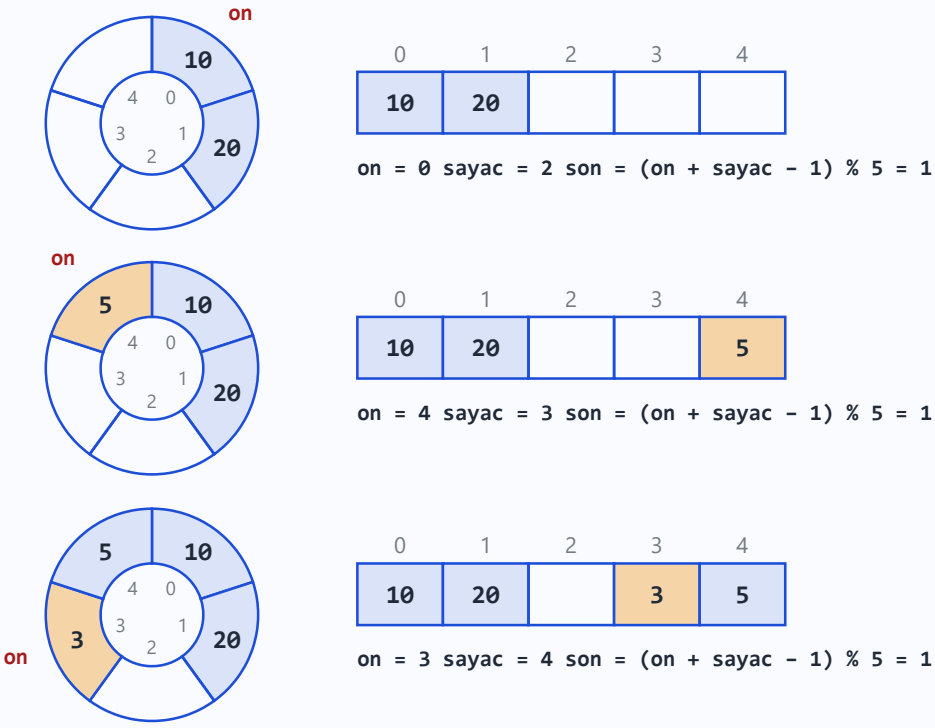
Örnek 7.4

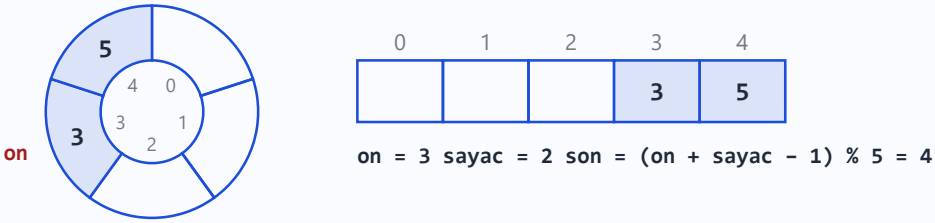
`KAP = 5`. `sona_ekle` ile 10 ve 20 konmuştur; `on = 0`, `sayac = 2`. Sırasıyla `basa_ekle(&d, 5)`, `basa_ekle(&d, 3)` ve iki kez `sondan_al` çağrılır. Tablo 7.10 her adımı verir.

Çağrı	Hesap	`on`	`sayac`	`veri[0..4]`	Sıra
Başlangıç	—	0	2	`10 20 . . .`	10 20
`basa_ekle(&d, 5)`	`on` = (0 − 1 + 5) % 5 = 4	4	3	`10 20 . . 5`	5 10 20
`basa_ekle(&d, 3)`	`on` = (4 − 1 + 5) % 5 = 3	3	4	`10 20 . 3 5`	3 5 10 20
`sondan_al`	konum = (3 + 4 − 1) % 5 = 1; `*x` = 20	3	3	`10 . . 3 5`	3 5 10
`sondan_al`	konum = (3 + 3 − 1) % 5 = 0; `*x` = 10	3	2	`. . . 3 5`	3 5

Tablo 7.10: Dairesel dizide deque izlemesi (`KAP = 5`). Son sütun deque'nin baştan sona mantıksal sırasıdır.

İlk `basa_ekle`, sarmanın geri yönde de işlediğini gösterir. `on` 0'dan 4'e geçer ve 5, dizinin son hücresine yazılır. Mantıksal sırada ise 5, 10'un önündedir. Dizi düz okunduğunda 10, 20, ·, ·, 5 görünür; deque'nin sırası ise `on`'dan başlayıp sarılarak okunur: 5, 10, 20. İki `sondan_al` çağrısı en son giren öğeleri değil, deque'nin *sonundaki* öğeleri verir: önce 20, sonra 10. Bu noktada bir `bastan_al` çağrılsaydı 3 dönerdi ve `on` 4'e ilerlerdi.





Şekil 7.12: Deque izlemesinden dört kare. Üstten alta: başlangıç (10, 20); 5 başa eklenmiş, `on` 0'dan dizinin sonuna (4) sarmış; 3 de başa eklenmiş (`on = 3`); iki `sondan_al` sonrasında yalnız 3 ve 5 kalmış. Alttaki satır her karede son öğenin konumunu hesaplar.

> **Sık yapılan hata**
>
> Geri adımı `(on - 1) % KAP` diye yazmak. C'de `(0 - 1) % 5` ifadesinin değeri −1'dir; matematikteki gibi 4 değildir. Kod `veri[-1]` 'e yazar. Bu adres dizinin hemen önündeki belleğe, çoğu zaman yapının başka bir alanına ya da başka bir değişkene denk gelir. Derleyici bunu yakalamaz. Geri adımlarda her zaman `+ KAP` eklenmelidir.

7.5.3 Tek dizide birden çok kuyruk

Bazı programlar aynı anda çok sayıda kuyruk kullanır. Örneğin bir işletim sistemi her öncelik düzeyi için ayrı bir hazır süreç kuyruğu tutabilir. Her kuyruğa ayrı ve büyük bir dizi ayırmak belleği boşa harcar, çünkü kuyrukların hepsi aynı anda dolmaz. Yığınlarda kullanılan bir yöntem burada da denenebilir: tek boyutlu bir `memory[]` dizisi n kesime bölünür ve her kuyruğa bir kesim verilir. i'nci kuyruk için üç değer tutulur: `on[i]` , `arka[i]` ve kesimin bittiği yeri gösteren `sinir[i]` .

İki yığın tek diziye çok verimli yerleşir: biri dizinin başından sağa, öbürü sonundan sola doğru büyür ve ikisi ancak bütün dizi dolunca karşılaşır. Bu "uçlardan büyütme" hilesi kuyrukta işe yaramaz. Yığının alt ucu sabittir, kuyruğun ise iki ucu da hareket eder. Kuyruk ekleme ve çıkarmalarla kesim içinde sürekli kayar. Her kesim kendi içinde dairesel kuyruk gibi yönetilirse kaydırma gerekmez. Ama bir kesim dolduğunda, başka kesimlerde boş yer olsa bile, kuyruklardan birini taşıyıp yer açmak gerekir. Bu da m dizinin boyu olmak üzere $O(m)$ sürer. Bu karmaşıklık yüzünden günümüzde birden çok kuyruk için genellikle her kuyruğa ayrı bir büyüyen dizi ya da bağlı liste kullanılır.

7.5.4 Round robin iş zamanlama

Tek işlemcili bir bilgisayarda aynı anda birçok süreç çalışmak ister. İşletim sistemi işlemciyi süreçler arasında paylaştırır. En bilinen paylaştırma yöntemi **zaman dilimli zamanlama**dır; İngilizce adıyla **round robin** de denir. Hazır süreçler bir kuyrukta bekler. Zamanlayıcı öndeki süreci alır ve onu en çok Q birim, yani bir zaman dilimi kadar çalıştırır. Süreç bu sürede işini bitirirse sistemden ayrılır. Bitiremezse kalan işiyle birlikte kuyruğun sonuna geri döner. Böylece hiçbir süreç işlemciyi tekeline alamaz ve her süreç düzenli aralıklarla sıra bulur.

Benzetimde süreçler 0, 1, 2 numaralarıyla, kalan iş miktarları `kalan[]` dizisiyle temsil edilir. Kuyruk olarak sayaçlı dairesel kuyruk kullanılır; fonksiyon adları bu kez `kuyruk_ekle` ve `kuyruk_al` 'dır:

```
#define KAP 16
typedef struct { int veri[KAP]; int on, sayac; } Kuyruk;

bool kuyruk_ekle(Kuyruk *k, int x) {
  if (k->sayac == KAP) return false;
  k->veri[(k->on + k->sayac) % KAP] = x;
  k->sayac++;
  return true;
}
bool kuyruk_al(Kuyruk *k, int *x) {
  if (k->sayac == 0) return false;
  *x = k->veri[k->on];
  k->on = (k->on + 1) % KAP;
  k->sayac--;
  return true;
}
```

Başlangıçta üç süreç de kuyruktadır. P0'ın 5, P1'in 2, P2'nin 4 birim işi vardır ve zaman dilimi $Q = 2$'dir:

```
int kalan[] = {5, 2, 4};      /* P0, P1, P2 iş miktarları */
int Q = 2, zaman = 0, p;
Kuyruk k = { {0}, 0, 0 };     /* boş sayaçlı kuyruk */
for (int i = 0; i < 3; i++)
  kuyruk_ekle(&k, i);         /* başta hepsi kuyrukta */
```

Zamanlayıcının kendisi tek bir döngüdür:

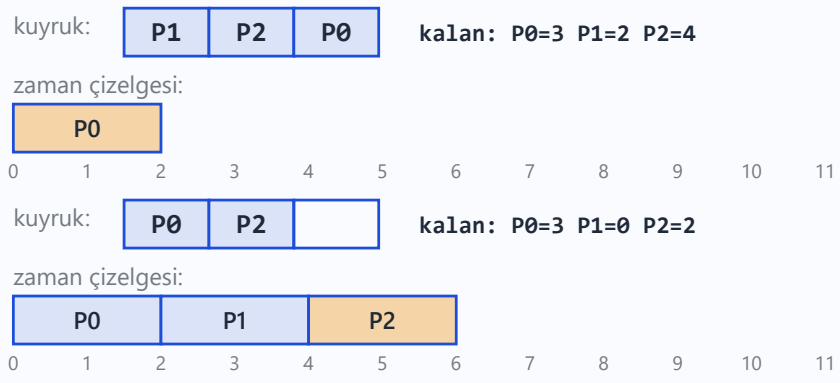
```
/* k: süreç numaraları kuyruğu; Q: zaman dilimi */
while (kuyruk_al(&k, &p)) {
  int dilim = kalan[p] < Q ? kalan[p] : Q;
  zaman += dilim;
  kalan[p] -= dilim;
  if (kalan[p] > 0)
    kuyruk_ekle(&k, p);           /* sona geri */
  else
    printf("P%d biter: t=%d\n", p, zaman);
}
```

Döngü kuyruk boşalana kadar sürer; `kuyruk_al` boş kuyrukta `false` döndürdüğü için koşul aynı zamanda bitiş denetimidir. Her turda öndeki süreç `p` 'ye alınır. Sürecin bu turda çalışacağı süre, kalan işi ile zaman diliminden küçük olanıdır; koşullu ifade (`? :`) bunu tek satırda hesaplar. Saat (`zaman`) bu kadar ilerler, sürecin kalan işi de bu kadar azalır. İş kaldıysa süreç kuyruğun sonuna eklenir, kalmadıysa bitiş anı yazdırılır.

Tur	Alınan	`dilim`	Çalışma aralığı	`kalan` (P0, P1, P2)	Sonuç	Kuyruk (önden arkaya)
—	—	—	—	5, 2, 4	—	P0 P1 P2
1	P0	2	0–2	3, 2, 4	sona döner	P1 P2 P0
2	P1	2	2–4	3, 0, 4	**P1 biter: t=4**	P2 P0
3	P2	2	4–6	3, 0, 2	sona döner	P0 P2
4	P0	2	6–8	1, 0, 2	sona döner	P2 P0
5	P2	2	8–10	1, 0, 0	**P2 biter: t=10**	P0
6	P0	1	10–11	0, 0, 0	**P0 biter: t=11**	(boş)

Tablo 7.11: P0 = 5, P1 = 2, P2 = 4 birim iş ve *Q = 2* ile round robin izlemesi.

Program sırasıyla `P1 biter: t=4` , `P2 biter: t=10` ve `P0 biter: t=11` yazar. Son turda P0'ın yalnız 1 birim işi kaldığı için dilim 2 değil 1'dir. Toplam süre işlerin toplamına (5 + 2 + 4 = 11) eşittir, çünkü işlemci hiç boş kalmaz. Zamanlamanın sonucu bitiş sırasıdır. En kısa iş (P1) önce, en uzun iş (P0) en son biter. Oysa süreçler önce gelen önce hizmet alır kuralıyla, araya girmeden çalıştırılsaydı bitiş anları P0 için 5, P1 için 7, P2 için 11 olurdu. Kısa iş, uzun bir işin arkasında beklemek zorunda kalırdı.





Şekil 7.13: Round robin benzetiminden üç kare. Her karede üstte kuyruk ve kalan işler, altta zaman çizelgesi vardır. Üstten alta: P0 ilk dilimden sonra sona dönmüştür; P2 ilk dilimden sonra sona dönmüştür ($t = 6$); bütün süreçler bitmiştir ($t = 11$).

Maliyet bakımından her tur bir `kuyruk_al` ve en çok bir `kuyruk_ekle` yapar; ikisi de $O(1)$'dir. Bir sürecin toplam işi w birimse süreç $\lceil w/Q \rceil$ kez kuyruktan alınır. Benzetimin toplam maliyeti bu sayıların toplamıyla orantılıdır. Örnekte bu toplam 3 + 1 + 2 = 6 turdur. Zaman dilimi küçüldükçe süreçler daha sık sıra bulur, ama tur sayısı artar. $Q = 1$ ile aynı örnek 11 tur sürer. Gerçek sistemlerde her tur bir bağlam değiştirme bedeli taşıdığı için dilim çok küçük seçilmez.

7.5.5 İki yığınla kuyruk

Yalnız yığın işlemleri sunan bir ortamda kuyruk gerekirse iki yığın yeter. Fikir, yığının sırayı ters çevirmesine dayanır. Bir dizi öğe bir yığına konup oradan başka bir yığına aktarılırsa iki kez ters çevrilmiş olur, yani ilk sırasına döner. Kuyruk iki yığından oluşur: `giris` ve `cikis`. Yığın tipi `Yigit`, `veri` dizisi ve boşken −1 olan `ust` indisiyle tanımlıdır. `push`, `pop` ve `yigit_bos` bilinen yığın işlemleridir; `pop`, yığın boşsa `false` döndürür. Kuyruk tipi `typedef struct { Yigit giris, cikis; } Kuyruk2;` biçimindedir.

```
void ekle(Kuyruk2 *q, int x) {
  push(&q->giris, x);
}
bool al(Kuyruk2 *q, int *x) {
  int t;
  if (yigit_bos(&q->cikis))       /* çıkış boşsa */
    while (pop(&q->giris, &t))    /* hepsini aktar */
      push(&q->cikis, t);
  return pop(&q->cikis, x);
}
```

`ekle` öğeyi her zaman giriş yığınına koyar; bunun maliyeti tek bir `push`'tur. `al`, çıkış yığını boşsa önce girişteki bütün öğeleri tek tek çıkış yığınına aktarır. Aktarma sırayı ters çevirir: girişin en altındaki, yani en eski öğe çıkışın en üstüne gelir. Sonra çıkış yığınından bir `pop` yapılır. Çıkış yığını boş değilse aktarma yapılmaz, çünkü çıkıştaki her öğe girişteki her öğeden daha eskidir. Çıkış boşalmadan aktarma yapılsaydı yeni öğeler eskilerin üstüne biner ve sıra bozulurdu. İki yığın da boşsa son `pop` `false` döndürür ve `al` da `false` döner. `ekle` ise `push`'un dönüş değerini yok sayar. Giriş yığını dolu olursa öğe sessizce kaybolur; güvenli bir sürüm bu değeri çağırana iletmelidir.

İşlem	Ne olur?	`giris`	`cikis`	Dönen
ekle 1, 2, 3	üç `push`	1 2 3	—	—
al	çıkış boş: 3, 2, 1 aktarılır; sonra `pop`	—	3 2	1
ekle 4	`push`	4	3 2	—
al	çıkış dolu: yalnız `pop`	4	3	2
al	çıkış dolu: yalnız `pop`	4	—	3
al	çıkış boş: 4 aktarılır; sonra `pop`	—	—	4

Tablo 7.12: İki yığınla kuyruk izlemesi. Yığınlar alttan üste doğru yazılmıştır.

Çıkış sırası 1, 2, 3, 4'tür; ekleme sırasıyla aynıdır. 4, 2 ve 3'ten sonra eklendiği hâlde onlardan önce çıkmaz. Çünkü 4 eklendiğinde 2 ve 3 zaten çıkış yığınındadır ve 4 ancak çıkış boşaldıktan sonra aktarılır.



Şekil 7.14: İki yığınla kuyruk. Üstten alta: 1, 2, 3 giriş yığınındadır; üçü de çıkış yığınına aktarılmış ve en eski öğe 1 en üste gelmiştir; 1 alınmıştır, çıkışta 2 ve 3 kalmıştır; en sonda çıkış boşalınca 4 de aktarılmıştır ve alınacak tek öğedir.

Tek bir `al` çağrısı pahalı olabilir: çıkış boşken girişte n öğe varsa n aktarma yapılır ve çağrı $\Theta(n)$ sürer. Ama böyle pahalı çağrılar seyrektir. Bir dizi işlemin toplam maliyetine bakıldığında işlem başına maliyet sabittir.

Teorem 7.2 (İki yığınla kuyruğun amortize maliyeti)

Boş bir kuyrukla başlayan ve m tane `ekle` ya da `al` çağrısından oluşan her işlem dizisi en çok $4m$ yığın işlemi (`push` ya da `pop`) yapar. Dolayısıyla her iki işlemin amortize maliyeti $O(1)$'dir.

Kanıt. Tek bir öğenin kuyruktaki ömrü izlensin. Öğe `ekle` ile giriş yığınına bir kez konur (bir `push`). Bir aktarma sırasında girişten bir kez çıkar ve çıkışa bir kez konur (bir `pop`, bir `push`). Çıkış yığınından giriş yığınına hiçbir zaman geri dönmez; bu yüzden ikinci kez aktarılmaz. Son olarak `al` ile çıkış yığınından bir kez çıkar (bir `pop`). Her öğe böylece en çok dört yığın işlemine katılır. Bunlara ek olarak her `al` çağrısı, yığınlar boşken başarısız olan en çok bir `pop` ve bir `yigit_bos` denetimi yapar. Dizide en çok m öğe eklenebildiği için öğelere ait işlemler en çok $4m$'dir; başarısız işlemler de sabit bir çarpanla sınırlıdır. Toplam iş $O(m)$, işlem başına ortalama iş $O(1)$'dir. ■

Bu kanıttaki yöntem, **amortize analizin** (*amortized analysis*) en sade biçimidir. Pahalı bir işlemin maliyeti, onu mümkün kılan ucuz işlemlere paylaştırılır. Burada her `ekle`, öğesinin sonradan aktarılma bedelini de önceden "öder". Teorem bir deneyle de sınandı. Rastgele seçilmiş 60 işlemlik bir dizide 37 ekleme yapıldı ve eklenen öğelerin hepsi sonunda alındı. Toplam 74 `push` ve 74 `pop` sayıldı; bu, öğe başına tam iki `push` ve iki `pop` demektir. Bütün öğeler ekleme sırasıyla çıktı. Burada dikkat edilmesi gereken bir nokta vardır: amortize $O(1)$, her çağrının hızlı olduğunu söylemez. Bir uygulama her işlemin belirli bir süreyi aşmamasını istiyorsa (örneğin gerçek zamanlı bir ses tamponu) dairesel kuyruk daha uygundur.

7.5.6 Yığın mı, kuyruk mu?

Bir problemde yığın mı kuyruk mu kullanılacağını belirlemek için tek bir soru sorulur: işlenmeyi bekleyen öğelerden hangisi önce ele alınmalıdır, en yenisi mi en eskisi mi? En yenisiyse yığın, en eskisiyse kuyruk uygundur. Ne en yeni ne en eski, başka bir ölçüte göre en önemli öğe önce alınacaksa öncelikli kuyruk gerekir. Tablo 7.13 sunumdaki örnekleri bu soruya göre sınıflandırır.

Problem	Yapı	Gerekçe
Yazıcıya gönderilen belgeler	Kuyruk	Belgeler gönderiliş sırasıyla basılır.
Metin düzenleyicide Ctrl+Z	Yığın	En son yapılan işlem ilk geri alınır.
Ağacı düzey düzey yazdırmak	Kuyruk	Düzey sırası gezinme (BFS): önce keşfedilen düğüm önce işlenir.
İç içe parantezlerin denetimi	Yığın	Kapanan parantez, en son açılanla eşleşir.
Fonksiyon çağrıları	Yığın	Çağrı yığını: en son çağrılan fonksiyon ilk döner.
Acil serviste en ağır hasta önce	Öncelikli kuyruk	Sıra gelişe değil önceliğe bağlıdır; FIFO değildir.

Tablo 7.13: Problemlere uygun yapılar.

Son satır, adındaki "kuyruk" sözcüğüne bakılarak yanlış sınıflandırılabilir. Sunumdaki hocanın notu da bu karışıklığa dikkat çeker. Günlük dildeki kuyrukta sondan eklenir, baştan alınır. Öncelikli kuyruk ise bu kurala uymaz; adındaki sözcük yalnız "bekleme yeri" anlamını taşır.

7.6 Karmaşıklık özeti ve ölçüm

Tablo 7.14 notta incelenen dört gerçeklemenin işlem maliyetlerini bir araya getirir. Tablodaki her değerin gerekçesi ilgili kesimde verilmiştir. Burada yalnız ortak noktalar özetlenir.

İşlem	Doğrusal dizi	Dairesel dizi	Bağlı	Deque (dairesel)
Ekle	*O(1)**	*O(1)*	*O(1)*	*O(1)*, iki uçta
Al	*O(1)*	*O(1)*	*O(1)*	*O(1)*, iki uçta
Ön / boş / dolu	*O(1)*	*O(1)*	*O(1)*	*O(1)*
Öğe sayısı	`arka - on`	`(arka-on+MAX)%MAX`	*O(n)* ya da sayaç	`sayac`
Bellek	`MAX`	`MAX`	n · (öğe + işaretçi)	`KAP`

Tablo 7.14: Kuyruk gerçeklemelerinin karmaşıklık özeti. n öğe sayısıdır.

Yıldızlı değer bir uyarı taşır. Doğrusal kuyrukta ekleme kendi başına *O(1)*'dir, ama sahte doluluk kaydırmayla giderilirse ekleme *O(n)*'ye çıkar. Bağlı kuyrukta öğe sayısı ayrı bir sayaç tutulmazsa ancak liste baştan sona yürünerek bulunur. Bir sayaç değişkeni eklemek, bu işlemi de *O(1)*'e indirir. Bütün gerçeklemelerde ekleme ve alma sabit sürdüğü için kuyruğu kullanan algoritmaların maliyeti, yaptıkları kuyruk işlemi sayısıyla belirlenir. Round robin benzetimi ve düzey sırası gezinme, n öğe için *O(n)* kuyruk işlemi yapar.

Kaydırmalı doğrusal kuyruk ile dairesel kuyruk arasındaki fark ölçümle de gösterildi. İki kuyruk da M hücrelik bir dizi kullandı. Kaydırmalı kuyrukta sürekli $M - 1$, dairesel kuyrukta bir hücre boş kalacağı için $M - 2$ öğe tutuldu. Bu durumda 200.000 kez bir ekleme ve bir alma yapıldı. Program `-O2` ile derlendi ve süreler `timespec_get` ile ölçüldü. Sonuçlar Tablo 7.15'te verilmiştir. Mutlak süreler makineye göre değişir; önemli olan, sütunların M ile nasıl değiştiğidir.

M	Kaydırmalı: taşınan öğe	Kaydırmalı: süre	Dairesel: süre
1000	199.799.001	7–8 ms	≈ 1 ms
2000	399.798.001	13–15 ms	≈ 1 ms
4000	799.796.001	31–33 ms	≈ 1 ms
8000	1.599.792.001	63–76 ms	≈ 1 ms

Tablo 7.15: Neredeyse dolu kuyrukta 200.000 ekleme ve alma çifti. Süreler iki çalıştırmanın değer aralığıdır ve makineye göre değişir.

Taşınan öğe sayısı *M* her iki katına çıktığında iki katına çıkar; her ekleme yaklaşık *M − 1* öğe taşır. Süreler de aynı oranda artar. Dairesel kuyruğun süresi ise *M*'den bağımsızdır. Kaydırmalı sürümün süreleri, taşınan öğe sayısına göre şaşırtıcı biçimde kısadır. Bunun nedeni, derleyicinin kaydırma döngüsünü bellek bloklarını toplu kopyalayan hızlı bir işleme çevirmesidir. Yine de büyüme doğrusaldır ve kuyruk büyüdükçe fark açılır.

7.7 Sık yapılan hatalar

Kuyruk kodlarında en sık görülen altı hata aşağıda toplanmıştır. Her biri önceki kesimlerde ayrıntısıyla ele alındı; burada belirtisi ve çaresi birlikte verilir.

1. **Doğrusal kuyrukta `arka == MAX - 1`'i gerçek doluluk sanmak.** Koşul yalnız arkada yer kalmadığını söyler; öğe sayısı `arka - on`'dur ve çok küçük olabilir. Çaresi dairesel kuyruktur.
2. **Dairesel kuyrukta `% MAX`'ı unutmak.** İndis ilk sarmada dizinin dışına çıkar. Program uzun süre doğru çalışıp sonra beklenmedik bir yerde bozulabilir. İndisi değiştiren her satırda kalan işlemi aranmalıdır.
3. **Önce `arka`'yı ilerletip kuyruk doluysa geri almamak.** Hata döndüren bir sürümde dolu kuyruğa yapılan ekleme denemesi `arka`'yı `on`'a eşitler ve kuyruk boş görünür. Yeni konum önce geçici bir değişkende denenmelidir.
4. **Geri adımda `(i - 1) % MAX` yazmak.** C'de negatif sayının kalanı negatiftir; `i = 0` için sonuç −1 olur. Doğrusu `(i - 1 + MAX) % MAX`'tır.
5. **Bağlı kuyruktan son öğe alınınca `arka = NULL` yapmamak.** `arka` serbest bırakılmış düğümü gösterir; sonraki ekleme bu belleğe yazar ve yeni öğe kuyrukta görünmez.
6. **Boş bağlı kuyruğa ilk eklemede `on`'u güncellemeyi unutmak.** Ya `NULL` işaretçi üzerinden yazılır ve program çöker ya da `on` `NULL` kalır ve eklenen öğeler hiçbir zaman alınamaz.

Bu hataların ortak kökü, kenar durumlardır: boş kuyruk, dolu kuyruk, tek öğeli kuyruk ve dizinin son hücresi. Bir kuyruk gerçeklemesi sınanırken en az şu senaryolar denenmelidir: boş kuyruktan alma, kuyruğu tam doldurup bir ekleme daha deneme, en az bir sarma, kuyruğu tümüyle boşaltıp yeniden kullanma.

7.8 Alıştırmalar ve çözümleri

Bu kesim sunumdaki beş alıştırmayı ayrıntılı çözümleriyle verir. Kod içeren çözümler derlenip sınanmıştır.

7.8.1 Öğe sayısı ve konumları

Soru. `MAX = 6`, `on = 4`, `arka = 1` olan dairesel kuyrukta kaç öğe vardır ve öğeler hangi indislerdedir?

Çözüm. Öğe sayısı `(arka - on + MAX) % MAX = (1 - 4 + 6) % 6 = 3`'tür. `on` ilk öğenin bir öncesini gösterdiği için öğeler `on`'un bir sonrasından başlar: `(4 + 1) % 6 = 5`. Ardından sarmayla 0 ve 1 gelir. Öğeler 5, 0 ve 1 numaralı hücrelerdedir ve kuyruğun önündeki öğe 5 numaralı hücrededir. 2, 3 ve 4 numaralı hücreler boştur; 4 numaralı hücre, birinci yöntemin boş bıraktığı hücredir.

7.8.2 Bayrak yöntemiyle dairesel kuyruk

Soru. Dolu ile boşu bir bayrakla ayırt eden dairesel kuyruğun ekleme ve alma yordamlarını yazın.

Çözüm. `son_ekleme` adlı mantıksal değişken, son yapılan işlemin ekleme olup olmadığını tutar. `on == arka` durumunda bu değişken doğruysa kuyruk dolu, yanlışsa boştur. Başlangıçta kuyruk boş olduğu için değeri `false`'tur.

```
#define MAX 4
int kuyruk[MAX];
int on = 0, arka = 0;
bool son_ekleme = false;      /* son işlem ekleme miydi? */

bool ekle(int x) {
    if (on == arka && son_ekleme) return false;   /* dolu */
    arka = (arka + 1) % MAX;
    kuyruk[arka] = x;
    son_ekleme = true;
    return true;
}
bool al(int *x) {
    if (on == arka && !son_ekleme) return false;  /* boş */
    on = (on + 1) % MAX;
    *x = kuyruk[on];
    son_ekleme = false;
    return true;
}
```

Ekleme, dolu denetiminden sonra sunumdaki kodla aynı biçimde çalışır, ama son olarak bayrağı kaldırır. Alma ise bayrağı indirir. Bayrak yalnız `on == arka` iken okunur. Kuyrukta 1 ile `MAX - 1` arasında öğe varken iki indis zaten farklıdır ve bayrağın değeri önemsizdir. Bu sürüm sınandığında dört hücreli boş kuyruğa dört ekleme sığdı (`on = arka = 0` , bayrak doğru), beşinci ekleme reddedildi. Bir alma (10) ve bir ekleme (99) sonrasında kuyruk yine doldu ve öğeler 11, 12, 13, 99 sırasıyla çıktı. Hiçbir hücre israf edilmez; bedeli, her işlemde bir atama ve dolu ya da boş denetiminde bir mantıksal karşılaştırmadır.

7.8.3 İki yığınla kuyruğun maliyeti

Soru. İki yığınla bir kuyruk yazın. `al` işleminin amortize maliyeti nedir?

Çözüm. Kod, sunumdaki `ekle` ve `al` fonksiyonlarıdır: ekleme giriş yığınına `push` yapar; alma, çıkış yığını boşsa girişi tümüyle aktarır, sonra çıkıştan `pop` yapar. Teorem 7.2'nin kanıtında gösterildiği gibi her öğe en çok iki kez yığına konur ve iki kez yığından çıkar. m işlemlik bir dizinin toplam maliyeti $O(m)$'dir. `al` 'ın amortize maliyeti bu yüzden $O(1)$'dir. En kötü durumdaki tek çağrı ise $\Theta(n)$ sürer. Örneğin 1, 2, 3, 4 eklenip bir öğe alınsın, 5 eklensin ve dört öğe daha alınsın. Çıkış 1, 2, 3, 4, 5'tir. İlk alma dört öğeyi, son alma yalnız 5'i aktarır; toplam `push` sayısı 5 ekleme ve 5 aktarma olmak üzere 10'dur.

7.8.4 Yalnız arka işaretçili dairesel bağlı kuyruk

Soru. Kuyruğu, yalnız `arka` işaretçisi tutulan dairesel bir bağlı listeyle yazın.

Çözüm. Dairesel bağlı listede son düğümün bağı `NULL` değil, ilk düğümdür. Bu yüzden ayrı bir `on` işaretçisine gerek kalmaz: ön düğüm her zaman `arka->bag` 'dir. Tek düğümlü kuyrukta düğüm kendini gösterir. Boş kuyrukta `arka` `NULL` 'dır.

```
bool ekle(dugum **arka, int x) {
  dugum *yeni = malloc(sizeof *yeni);
  if (yeni == NULL) return false;
  yeni->deger = x;
  if (*arka == NULL) yeni->bag = yeni;      /* tek düğüm kendini gösterir */
  else {
    yeni->bag = (*arka)->bag;               /* yeni, önü gösterir */
    (*arka)->bag = yeni;                    /* eski son, yeniyi gösterir */
  }
  *arka = yeni;
  return true;
}
bool al(dugum **arka, int *x) {
  if (*arka == NULL) return false;          /* boş */
  dugum *on = (*arka)->bag;                 /* ön, arkanın bir sonrası */
  *x = on->deger;
  if (on == *arka) *arka = NULL;            /* son düğüm de gitti */
  else (*arka)->bag = on->bag;
  free(on);
  return true;
}
```

Ekleme, yeni düğümü son düğüm ile ön düğüm arasına yerleştirir. Önce yeni düğüm ön düğümü gösterir (`yeni->bag = arka->bag`), sonra eski son düğüm yeni düğümü gösterir, en son `arka` yeni düğüme taşınır. Bağların bu sırası önemlidir: ikinci atama önce yapılsaydı ön düğümün adresi kaybolurdu. Alma, ön düğümü (`arka->bag`) çıkarır. Kuyrukta tek düğüm varsa ön ile arka aynı düğümdür ve kuyruk boşalır. Aksi hâlde son düğümün bağı ikinci düğüme çevrilir. İki işlem de *O(1)*'dir ve tek işaretçi tutulur. Sınamada 10, 20, 30 eklenip alındığında sıra korundu; 1003 `malloc` çağrısının hepsi `free` ile iade edildi.

7.8.5 Round robin, *Q = 3*

Soru. Round robin örneğini (P0 = 5, P1 = 2, P2 = 4) *Q = 3* ile çalıştırın. Süreçler ne zaman biter?

Çözüm. P0 0–3 arasında çalışır ve 2 birimi kalır; sona döner. P1 3–5 arasında çalışır ve **t = 5**'te biter. P2 5–8 arasında çalışır ve 1 birimi kalır; sona döner. P0 8–10 arasında kalan 2 birimi çalışır ve **t = 10**'da biter. P2 10–11 arasında son birimini çalışır ve **t = 11**'de biter. Program `P1 biter: t=5` , `P0 biter: t=10` , `P2 biter: t=11` yazar. *Q = 2*'ye göre bitiş sırası değişmiştir: P0 artık P2'den önce biter. Tur sayısı da 6'dan 5'e iner. Dilim büyüdükçe zamanlama, önce gelen önce hizmet alır düzenine yaklaşır; *Q ≥ 5* için bitiş anları 5, 7 ve 11 olur.

7.9 Özet

- **Kuyruk**, eklemenin arkadan, çıkarmanın önden yapıldığı sıralı listedir; kuralı **ilk giren ilk çıkar**'dır (FIFO). Yığın sırayı ters çevirir, kuyruk korur. Temel işlemler `kuyruk_ekle` (taşma), `kuyruk_al` (alt taşma), `kuyruk_on` , `kuyruk_bos` ve `kuyruk_dolu` 'dur.
- **Doğrusal kuyruk** iki indis kullanır: `on` ilk öğenin bir öncesini, `arka` son öğeyi gösterir; boş koşulu `on == arka` 'dır. İndisler yalnız ileri gittiği için önde boşalan hücreler kullanılamaz (**sahte doluluk**). Kaydırma bu sorunu çözer ama eklemeyi *O(n)* yapar.
- **Dairesel kuyruk** indisleri `% MAX` ile sarar. Boş: `on == arka` ; dolu: `(arka + 1) % MAX == on` . Öğe sayısı `(arka - on + MAX) % MAX` , geri adım `(i - 1 + MAX) % MAX` 'tır; C'de negatif sayının kalanı negatiftir.
- İki indis `MAX + 1` doluluk durumunu ayırt edemez. Çare ya bir hücreyi boş bırakmak (kapasite `MAX - 1`) ya da bir **sayaç** veya **bayrak** tutmaktır. Sayaçlı sürüm öğe sayısını *O(1)*'de verir ve `arka` 'yı hesaplar.
- Başarısız olabilen bir işlem, başarısız olduğunda yapıyı değiştirmemelidir: yeni konum önce geçici bir değişkende denenir.
- **Bağlı kuyruk** `on` ve `arka` işaretçileriyle sona ekler, baştan siler; iki işlem de *O(1)*'dir. Boş kuyruğa ilk eklemede `on` , son öğe çıkınca `arka = NULL` güncellenir; aksi hâlde sızıntı ya da sarkan işaretçi oluşur.
- Pratikte sabit kapasiteli tamponlar için dairesel dizi, üst sınırı bilinmeyen kuyruklar için iki katına büyüyen dairesel dizi (amortize *O(1)*) kullanılır; büyütmede öğeler `on` 'dan başlayarak sırayla kopyalanır.
- **Deque** iki uçta da ekleme ve çıkarma yapar; dairesel dizide dört işlemin hepsi *O(1)*'dir. Yığını da kuyruğu da kapsar.
- **Round robin** zamanlamada süreçler kuyrukta bekler, en çok bir zaman dilimi çalışır ve işi kalırsa sona döner. **İki yığınla kuyrukta** her öğe en çok bir kez aktarılır; işlemler amortize *O(1)*'dir.

7.10 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

kuyruk *queue* · ilk giren ilk çıkar *FIFO (first in, first out)* · son giren ilk çıkar *LIFO (last in, first out)* · ön *front* · arka *rear* · enqueue (arkaya ekleme) *enqueue* · dequeue (önden çıkarma) *dequeue* · soyut veri tipi *abstract data type* · taşma *overflow* · alt taşma *underflow* · dairesel kuyruk *circular queue* · modüler aritmetik *modular arithmetic* · tampon *buffer* · genişlik öncelikli arama *breadth-first search (BFS)* · düzey sırası *level-order* · öncelikli kuyruk *priority queue* · sarkan işaretçi *dangling pointer* · bellek sızıntısı *memory leak* · çift uçlu kuyruk *deque (double-ended queue)* · amortize analiz *amortized analysis* · ikiye katlama *doubling* · yığın *stack*

7.11 Kendini Dene

1. Güvenli dairesel kuyrukta `MAX_QUEUE_SIZE = 4` ve başlangıçta `on = arka = 0` 'dır. Sırasıyla 1, 2, 3, 4 eklenmek istenir, bir öğe alınır

ve 4 yeniden eklenir. Hangi ekleme reddedilir? Sonunda `on`, `arka` ve öğe sayısı nedir? Sonraki `kuyruk_al` hangi değeri verir?

Cevap

1, 2, 3 sırasıyla 1, 2, 3 numaralı hücrelere girer. 4 için `yeni = (3 + 1) % 4 = 0 = on` olur ve ekleme reddedilir. Alma `on`'u 1'e ilerletir ve 1'i verir. Yeniden eklenen 4, `arka = 0` hücresine yazılır. Sonunda `on = 1`, `arka = 0`, öğe sayısı `(0 − 1 + 4) % 4 = 3`'tür. Sonraki alma 2'yi verir.

2. `MAX_QUEUE_SIZE = 5` olan doğrusal kuyruğa üç öğe eklenip üçü de alınıyor. Kuyruk boş olduğu hâlde en çok kaç ekleme daha yapılabilir? Neden?

Cevap

Üç ekleme ve üç almadan sonra `on = arka = 2`'dir. Ekleme `arka` 4 olana kadar yapılabilir: iki ekleme. Üçüncü denemede `arka ≥ 4` olur ve kuyruk dolu sayılır. Nedeni sahte doluluktur: 0, 1 ve 2 numaralı hücreler boştur ama indisler yalnız ileri gittiği için bir daha kullanılmaz.

3. Bir programcı deque'de geri adımı `d->on = (d->on - 1) % KAP;` diye yazmıştır. `KAP = 8` ve `on = 0` iken `basa_ekle` çağrılırsa ne olur?

Cevap

C'de `(0 − 1) % 8` ifadesinin değeri −1'dir. `on` −1 olur ve öğe `veri[-1]`'e, yani dizinin dışına yazılır. Bu tanımsız davranıştır; yapının başka bir alanı ya da komşu bir değişken bozulabilir. Doğru ifade `(on − 1 + KAP) % KAP`'tır ve 7 verir.

4. Bağlı kuyruğun `kuyruk_al` fonksiyonundan `if (*on == NULL) *arka = NULL;` satırı silinmiştir. Boş kuyruğa 5 ve 7 eklenir, iki öğe alınır, sonra 9 eklenir. Ne olur?

Cevap

İkinci almadan sonra `on` `NULL`'dır, ama `arka` serbest bırakılmış 7 düğümünü göstermeye devam eder. 9 eklenirken `*arka != NULL` olduğu için kod kuyruğu boş saymaz ve `(*arka)->bag = yeni` ile serbest bırakılmış belleğe yazar; bu tanımsız davranıştır. `on` güncellenmediği için `NULL` kalır. Sonraki `kuyruk_al` kuyruğu boş bulur; 9 hiçbir zaman alınamaz ve düğümü sızar.

5. Sayaçlı dairesel kuyrukta `KAP = 4`, `on = 3`, `sayac = 3`'tür. Öğeler hangi hücrelerdedir? Bir sonraki `ekle` öğeyi hangi hücreye yazar?

Cevap

Öğeler `on`'dan başlayarak 3, 0 ve 1 numaralı hücrelerdedir. Son öğenin yeri `(3 + 3 − 1) % 4 = 1`'dir. Bir sonraki ekleme `(3 + 3) % 4 = 2` numaralı hücreye yazar ve sayaç 4 olur; kuyruk dolar.

6. `KAP = 6` olan boş bir deque'ye (`on = 0`, `sayac = 0`) sırasıyla `basa_ekle 1`, `basa_ekle 2`, `sona_ekle 3` uygulanır. Hangi değer hangi hücrededir? `bastan_al` ne verir?

Cevap

İlk `basa_ekle` `on`'u `(0 − 1 + 6) % 6 = 5`'e alır ve 1'i 5 numaralı hücreye yazar. İkincisi `on`'u 4'e alır ve 2'yi 4 numaralı hücreye yazar. `sona_ekle 3`, `(4 + 2) % 6 = 0` numaralı hücreye yazar. Deque'nin sırası 2, 1, 3'tür; `bastan_al` 2'yi verir.

7. Round robin benzetiminde üç sürecin her birinin 3 birim işi vardır ve $Q = 2$'dir. Süreçler hangi anlarda biter?

Cevap

İlk turda P0 0–2, P1 2–4, P2 4–6 arasında çalışır; her birinin 1 birimi kalır ve sona döner. İkinci turda P0 6–7, P1 7–8, P2 8–9 arasında çalışır. Bitiş anları P0 için 7, P1 için 8, P2 için 9'dur.

8. İki yığınla kurulan kuyruğa 1, 2, 3, 4 eklenir; bir öğe alınır; 5 eklenir; dört öğe daha alınır. Çıkış sırası nedir? Toplam kaç `push` yapılır?

   Cevap

   İlk almada çıkış yığını boştur; 4, 3, 2, 1 aktarılır ve 1 alınır. 5 giriş yığınına konur. Sonraki üç alma çıkış yığınından 2, 3, 4'ü verir. Son almada çıkış boştur; 5 aktarılır ve alınır. Çıkış sırası 1, 2, 3, 4, 5'tir. `push` sayısı 5 ekleme ve 5 aktarma olmak üzere 10'dur.

9. Birinci yöntemde (bir hücreyi boş bırakmak) `MAX` hücreli kuyruk neden en çok `MAX - 1` öğe tutar? Kuyruğun `MAX` öğe tutmasına izin verilseydi ne olurdu?

   Cevap

   `MAX`'ıncı öğe eklenseydi `arka` ilerleyip `on`'a eşit olurdu. `on == arka` ise boş kuyruğun koşuludur. Kod dolu kuyruğu boş sanır: `kuyruk_al` alt taşma bildirir, `kuyruk_ekle` ise öğelerin üzerine yazar. İki indisin farkı yalnız `MAX` değer alır, oysa öğe sayısı `MAX + 1` değer alabilir; bir durumdan vazgeçmek gerekir.

7.12 İleri okuma

- E. Horowitz, S. Sahni ve S. Anderson-Freed, *Fundamentals of Data Structures in C* (2. baskı), Kesim 3.3 ve 4.3: bu notun izlediği dizi tabanlı ve bağlı kuyruk gerçeklemelerinin, dairesel kuyruğun ve tek dizide birden çok kuyruğun kaynağıdır.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (3. baskı), Kesim 10.1: kuyruğu sözde kodla, dairesel dizi üzerinde kısa ve kesin biçimde verir; deque ve iki yığınla kuyruk alıştırma olarak yer alır.
- R. Sedgewick, *Algorithms in C*, Kısım 1–4, Bölüm 4: soyut veri tiplerini, yığın ve kuyruk arayüzlerini ve bunların dizi ve bağlı liste gerçeklemelerini karşılaştırmalı olarak işler.

7.13 Kaynaklar

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, Bölüm 3.3 ve 4.3.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3. baskı, MIT Press, Bölüm 10.1 (kuyruklar, deque alıştırmaları).
3. R. Sedgewick, *Algorithms in C*, Kısım 1–4, Bölüm 4.

KISIM II · BÖLÜM 8

Yığın ve Kuyruk Uygulamaları

Yığın, en son konan öğeyi ilk veren basit bir yapıdır. Bu sadelik onu programlamanın pek çok köşesinde vazgeçilmez kılar. Bu not yığının iki klasik uygulamasını ayrıntısıyla işler. İlki labirentte çıkış yolu aramaktır: her yol ayrımı yığına konur, çıkmaza girilince son ayrıma geri dönülür. İkincisi aritmetik ifadelerin değerlendirilmesidir. Derleyici `a / b - c + d * e - a * c` gibi bir ifadeyi önce işleçlerin önceliğine göre sonek gösterime çevirir. Sonra bu ifadeyi tek geçişte hesaplar. İki iş de bir yığınla yapılır. Not, parantez denetimi, geri al/yinele, özyinelemesiz ağaç gezinmesi gibi başka uygulamalarla ve kuyruğun aynı problemlerde getirdiği farkla sona erer. Her yordamın C kodu, satır satır anlatımı, kod izlemesi ve karmaşıklık analizi verilir.

Kod (C): `github.com/sercankulcu/data-structures-and-algorithms-c/Bolum08`

ÖĞRENME ÇIKTILARI

Geri izlemeyi kodlamak
Labirentte yığınla yol aramak, çıkmazdan son yol ayrımına geri dönmek ve bu süreci adım adım izlemek.

En kötü durumu kurmak
Bir algoritmanın en çok iş yaptığı girdiyi tasarlamak ve maliyetini bu girdiyle gerekçelendirmek.

Gösterimleri çevirmek
Ara ek, önek ve sonek gösterimler arasında elle ve yığınla çeviri yapmak.

Sonek ifadeyi hesaplamak
`eval` yordamını işlenen yığınıyla adım adım çalıştırmak; işlenenlerin alınma sırasını doğru kurmak.

Önceliği kodlamak
`isp` / `icp` tablosuyla ara ek ifadeyi sonek gösterime çeviren yordamı yazmak ve eşit öncelikte doğru kararı vermek.

Yığını genellemek
Parantez denetiminde, geri al/yinele gibi problemlerde ve özyinelemesiz gezinmede yığını; en kısa yol aramada kuyruğu kullanmak.

8.1 Yığın ve geri izleme

Bu notun bütün yordamları tek bir yapının üzerine kurulur. Bu yüzden önce yığını kısaca hatırlayalım. **Yığın** (*stack*), öğelerin yalnız bir uçtan eklenip yine aynı uçtan çıkarıldığı doğrusal bir yapıdır. Bu uca **üst** (*top*) denir. Ekleme işlemi "üste koymak" (`push`), çıkarma işlemi "üstten almak" (`pop`) diye adlandırılır. En son konan öğe ilk çıkar. Bu kurala **son giren ilk çıkar** (*last in, first out*, LIFO) denir.

> **Tanım 8.1**
>
> **Yığın**, ekleme ve çıkarmanın yalnız üst denen uçta yapıldığı sıralı bir öğe topluluğudur. Dizi ile gerçeklenen yığında öğeler `yigit[0]` , `yigit[1]` , … hücrelerinde durur; `top` değişkeni en üstteki öğenin indisini tutar. Boş yığında `top == -1` 'dir. Üste koyma `yigit[++top] = x` , üstten alma `x = yigit[top--]` ile yapılır; ikisi de *O(1)* sürer.

Dizi ile gerçeklemede iki sınır durumu vardır. Dizi doluyken yeni öğe konamaz; buna **taşma** (*overflow*) denir. Boş yığından öğe alınamaz; buna da **alttan taşma** (*underflow*) denir. Bu notun kısa kodları çoğu yerde bu denetimleri yazmaz. Örneğin `eval` yordamı iyi kurulmuş bir ifadede yığının hiç boşalmayacağını varsayar. Doğrulama programlarında kullanılan `yigit_ekle` ve `yigit_al` fonksiyonları ise her iki durumu da denetler. Gerçek bir programda bu denetimler atlanmamalıdır.

Yığının "son giren ilk çıkar" davranışı bir problem türüne doğal olarak uyar. Bir dizi karar verilerek ilerlenen ve bazen yanlış bir kararın geri alınması gereken her problemde en son verilen karar ilk geri alınır. Örneğin bir labirentte ilerlerken her yol ayrımında bir yön seçilir. Seçilen yol çıkmaza çıkarsa en yakın, yani en son geçilen yol ayrımına dönülür ve orada başka bir yön denenir. Bu yöntemin adı **geri izle-**

medir (*backtracking*).

> **Tanım 8.2**
>
> **Geri izleme**, kısmi bir çözümü adım adım genişleten, çıkmaza girdiğinde son seçimi geri alıp bir sonraki seçeneği deneyen arama yöntemidir. Seçimler bir yığında tutulur: yeni seçim üste konur, geri dönüş üstteki seçimin alınmasıyla yapılır.

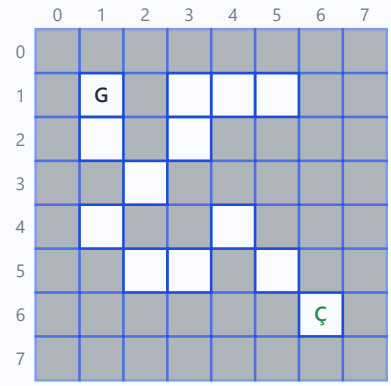
Geri izleme özyinelemeyle de yazılabilir. O durumda yığını programcı kurmaz; her özyinelemeli çağrı **çağrı yığınına** (*call stack*) bir çerçeve ekler ve çağrının dönmesi geri adımı oluşturur. Labirent yordamı ise yığını açıkça, bir dizi olarak kurar. Böylece her an yığında ne olduğu görülebilir ve izlenebilir. Bu, yığının geri izlemedeki rolünü anlamak için en iyi yoldur.

8.2 Labirent problemi

Labirent problemi, yığınla geri izlemenin klasik örneğidir. Bir giriş, bir çıkış ve aralarında duvarlarla bölünmüş geçitler vardır. Girişten çıkışa giden bir yol aranır. Problem basit görünür, ama ayrıntıları öğreticidir. Yönler nasıl kodlanacak, aynı hücreye iki kez girmek nasıl önlenecek, geri dönüşte nereden devam edilecek? Bu soruların cevapları veri yapısı seçiminin önemini gösterir.

8.2.1 Labirentin matrisle gösterimi

Labirent iki boyutlu bir tam sayı matrisiyle gösterilir. Matrisin her hücresi labirentin bir karesidir. `0` değeri açık yolu, `1` değeri duvarı simgeler. Şekil 8.1 notta kullanılan labirenti gösterir. Gri kareler duvar, beyaz kareler açık yoldur. Giriş `[1][1]` hücresidir ve G harfiyle, çıkış `[6][6]` hücresidir ve Ç harfiyle işaretlenmiştir. Labirentin boyutu *m* × *p* ile anılır: *m* satır, *p* sütun. Burada iç kısım 6 × 6'dır.



Şekil 8.1: Labirent matrisi. Gri kareler duvar (1), beyaz kareler açık yol (0). Giriş G = `[1][1]`, çıkış Ç = `[6][6]`. Dış çerçevedeki satır 0, satır 7, sütun 0 ve sütun 7 tümüyle duvardır.

Şekildeki matris 8 × 8'dir, oysa labirentin kendisi 6 × 6'dır. Fazladan iki satır ve iki sütun bilinçli bir hiledir. Matrisin çevresi baştan sona `1` ile, yani duvarla çevrilir. Bunun nedeni şudur. Kenardaki bir hücreden, örneğin `[1][1]`'den kuzeye bakıldığında `[0][1]` hücresine gidilir. Çerçeve olmasaydı bu hücre matrisin dışında kalırdı: 6 × 6'lık bir dizide `[-1][0]` gibi geçersiz bir indis oluşurdu. C, dizi sınırını denetlemez. Dizi dışına okuma tanımsız davranıştır; program ya çöker ya da rastgele bir bellek değerini duvar ya da yol sanar.

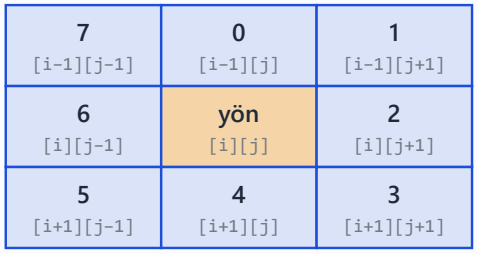
Çerçeve olmadan her adımda "matrisin dışına çıkıyor muyum?" diye dört koşul denetlemek gerekirdi. Çerçeve bu denetimi gereksiz kılar. Dış halkadaki her hücre duvardır ve duvara asla girilmez. Bu yüzden algoritma iç kısımdan hiçbir zaman dışarı adım atmaz ve her komşu indisi geçerli kalır. Kenar durumlarını ortadan kaldıran bu tür ek hücrelere genel olarak bekçi (*sentinel*) denir. Bedeli çok küçüktür: *(m + 2)(p + 2)* hücrelik bir matris.

Hareket serbestliği de problemin bir parçasıdır. Bu labirentte yatay ve dikey hareketin yanında çapraz hareket de serbesttir. Bir hücreden sekiz komşusuna gidilebilir. Çapraz hareket, labirentteki dar köşeleri aşmayı sağlar. Örneğin `[2][1]`'den `[3][2]`'ye ancak çapraz bir adımla geçilebilir, çünkü aradaki `[3][1]` ve `[2][2]` duvardır.

Algoritmanın iki ihtiyacı daha vardır. Birincisi, aynı hücreye ikinci kez girilmemelidir. Aksi hâlde iki hücre arasında sonsuza kadar gidip gelinebilir. Bu yüzden ziyaret edilen hücreler ayrı bir `gezildi` matrisinde işaretlenir. İkincisi, çıkmaza girilince son yol ayrımına geri dönülmelidir. Son geçilen ayrım ilk dönülecek yerdir; bu da yığının tam tanımıdır.

8.2.2 Sekiz yön ve `hareket` tablosu

Sekiz yön 0'dan 7'ye numaralanır. Yön 0 kuzeydir; numaralar saat yönünde artar. Yön 1 kuzeydoğu, 2 doğu, 3 güneydoğu, 4 güney, 5 güneybatı, 6 batı ve 7 kuzeybatıdır. Şekil 8.2 bir hücrenin sekiz komşusunu ve her komşunun indisini gösterir.



Şekil 8.2: Sekiz hareket yönü. Ortadaki hücre `[i][j]`'dir. Yön 0 kuzey (`[i-1][j]`), saat yönünde sonraki yönler 1…7'dir.

Bir yöne gitmek, satır ve sütun indisine birer küçük sayı eklemek demektir. Kuzeye gitmek satırı bir azaltır, sütunu değiştirmez: *(−1, 0)*. Güneydoğuya gitmek ikisini de bir artırır: *(+1, +1)*. Bu sekiz çift Tablo 8.1'de toplanmıştır.

yön	Ad	`hareket[yön].dikey`	`hareket[yön].yatay`
0	K (kuzey)	−1	0
1	KD (kuzeydoğu)	−1	1
2	D (doğu)	0	1
3	GD (güneydoğu)	1	1
4	G (güney)	1	0
5	GB (güneybatı)	1	−1
6	B (batı)	0	−1
7	KB (kuzeybatı)	−1	−1

Tablo 8.1: `hareket` dizisi. Yeni konum `[i + dikey][j + yatay]`'dır.

Bu tabloya **arama tablosu** (*look-up table*) denir. Değeri her seferinde koşullarla hesaplamak yerine önceden hesaplanmış bir diziden okumak anlamına gelir. Tablo olmasaydı sekiz ayrı `if` yazılırdı: "yön 0 ise satırı bir azalt, yön 1 ise satırı azalt ve sütunu artır…" Tablo bu sekiz dalı tek bir satıra indirir:

```
y_satir = satir + hareket[yon].dikey;
y_sutun = sutun + hareket[yon].yatay;
```

Bu sayede yön sıradan bir döngü değişkenine dönüşür. Sekiz yönü denemek için `yon`'u 0'dan 7'ye kadar artırmak yeter. Yönlerin deneme sırasını değiştirmek de kolaylaşır: yalnız tablonun satırlarının sırası değişir, kod aynı kalır.

8.2.3 Veri tanımları

Yordamın kullandığı bütün sabitler, tipler ve küresel değişkenler sunumdaki şu bildirimlerde toplanmıştır:

```
#define MAX_STACK_SIZE 100
#define CIKIS_SATIR 6
#define CIKIS_SUTUN 6
typedef struct { short int dikey, yatay; } yonler;
yonler hareket[8] = {{-1, 0}, {-1, 1}, {0, 1}, {1, 1},
                     {1, 0}, {1, -1}, {0, -1}, {-1, -1}};
typedef struct { short int satir, sutun, yon; } element;
element yigit[MAX_STACK_SIZE];
int top = -1;
short int gezildi[8][8];
short int labirent[8][8] = {{1,1,1,1,1,1,1,1},
    {1,0,1,0,0,0,1,1}, {1,0,1,0,1,1,1,1},
    {1,1,0,1,1,1,1,1}, {1,0,1,1,0,1,1,1},
    {1,1,0,0,1,0,1,1}, {1,1,1,1,1,1,0,1},
    {1,1,1,1,1,1,1,1}};
```

- `MAX_STACK_SIZE` yığın dizisinin boyudur. `CIKIS_SATIR` ve `CIKIS_SUTUN` çıkış hücresinin koordinatlarıdır.
- `yonler` yapısı bir yönün iki bileşenini tutar. `hareket[8]` dizisi Tablo 8.1'deki tablonun kendisidir; ilk çift `{-1, 0}` yön 0'a (kuzey) karşılık gelir. Değerler küçük olduğu için `short int` yeterlidir.
- `element`, yığına konan kayıttır. Üç alanı vardır: bir hücrenin satırı, sütunu ve o hücreye geri dönülürse denemeye hangi yönden devam edileceği (`yon`). Bu üçüncü alan çok önemlidir. Yığında yalnız konum tutulsaydı, geri dönüşte o hücrenin yönleri yeniden 0'dan denenirdi. Yön alanı "bu hücrede şu yöne kadar baktım" bilgisini saklar.
- `yigit` dizisi yığının kendisidir; `top = -1` yığının boş olduğunu belirtir.
- `gezildi` ziyaret edilen hücreleri işaretler. Küresel bir dizi olduğu için C onu program başlarken sıfırla doldurur; ayrıca ilk değer vermeye gerek yoktur. Yerel bir dizi olsaydı içeriği belirsiz olurdu.
- `labirent` çevresi 1 ile çevrili 8 × 8 matristir. Her iç küme parantezi bir satırdır; ilk satır tümüyle 1'dir.

Yığının en fazla kaç öğe tutacağı da önceden kestirilebilir. Bir hücre yığına ancak o hücreden yeni ve ziyaret edilmemiş bir hücreye geçilirken konur. Her açık hücre en çok bir kez ziyaret edildiğine göre, yığındaki öğe sayısı açık hücre sayısını aşamaz. Bu labirentin iç kısmında 13 açık hücre vardır (doğrulama programı da 13 saydı). `MAX_STACK_SIZE` değeri olan 100 bu labirent için fazlasıyla yeterlidir. Genel bir $m \times p$ labirent için güvenli üst sınır $m \cdot p$'dir.

8.2.4 `izbul` yordamı

Yol bulma yordamının adı `izbul` 'dur. Sunum kodu okunur kalsın diye iki parçada verir; aşağıda iki parça art arda gelir:

```
void izbul(void) {
  short int satir, sutun, y_satir, y_sutun;
  int i, yon, bulundu = FALSE;
  element konum;
  yigit[0].satir = yigit[0].sutun = 1;
  yigit[0].yon = 1;     /* ilk konum yığına */
  gezildi[1][1] = 1;  top = 0;
  while (top > -1 && !bulundu) {
    konum = yigit_al(&top);   /* yeni konum */
    satir = konum.satir;
    sutun = konum.sutun;
    yon = konum.yon;          /* hangi yöne */
    while (yon < 8 && !bulundu) {
      y_satir = satir + hareket[yon].dikey;
      y_sutun = sutun + hareket[yon].yatay;
```

```
      if (y_satir == CIKIS_SATIR &&
          y_sutun == CIKIS_SUTUN)
        bulundu = TRUE;       /* çıkış mı? */
      else if (!labirent[y_satir][y_sutun] &&
               !gezildi[y_satir][y_sutun]) {
        gezildi[y_satir][y_sutun] = 1;
        konum.satir = satir;  /* nereden */
        konum.sutun = sutun;
        konum.yon = ++yon;    /* dönüşte yön */
        yigit_ekle(&top, konum);
        satir = y_satir;  sutun = y_sutun;
        yon = 0;
      } else ++yon;           /* başka yön */
    }                         /* geri adım */
  }
```

Kod iki iç içe döngüden oluşur. Dıştaki döngü yığından bir konum alır; içteki döngü o konumdan ilerleyebildiği kadar ilerler. Satır satır bakalım.

- Başlangıçta giriş hücresi `(1, 1)` yığına konur. Yön alanı 1'dir. Yön 0 kuzeydir ve `[0][1]` çerçevedeki bir duvardır; denemeye gerek yoktur. Giriş hücresi `gezildi` matrisinde işaretlenir ve `top = 0` olur.
- Dış döngü yığın boş olmadığı ve çıkış bulunmadığı sürece döner. Her turda yığından bir `konum` alınır. `satir`, `sutun` ve `yon` bu kayıttan okunur. Yani "şu hücredeyim ve şu yönden denemeye başlayacağım".
- İç döngü `yon` 8'e ulaşmadığı ve çıkış bulunmadığı sürece döner. Her turda bir komşu hesaplanır: `(y_satir, y_sutun)`.
- Komşu çıkış hücresiyse `bulundu` doğru olur ve iki döngü de biter.
- Komşu açık yolsa (`!labirent[…]`) ve daha önce ziyaret edilmemişse (`!gezildi[…]`) o hücreye geçilir. Önce komşu işaretlenir. Sonra şimdiki hücre yığına konur, ama yön alanına `++yon` yazılır. Bu "geri dönersem bir sonraki yönden devam edeyim" demektir. Ardından `satir` ve `sutun` yeni hücreye çevrilir ve `yon = 0` ile yeni hücrenin bütün yönleri baştan denenir.
- Komşu duvarsa ya da ziyaret edilmişse yalnız `++yon` ile bir sonraki yöne geçilir.
- İç döngü `yon == 8` ile biterse o hücrenin sekiz yönü de tükenmiştir; hücre bir çıkmazdır. Kod bu durumda hiçbir şey yapmaz. Dış döngü bir sonraki turda yığından son kaydı alır ve böylece bir önceki hücreye geri dönülür. Yorumdaki "geri adım" sözü bu anı anlatır.

Birkaç ayrıntı dikkat ister. Birincisi, çıkış denetimi duvar denetiminden önce yapılır. Bu labirentte çıkış hücresi açıktır ve sıra önemli değildir. Ama çıkış hücresi `gezildi` matrisinde hiçbir zaman işaretlenmez; görüldüğü anda arama biter. İkincisi, yığına konan kayıt yeni hücre değil, terk edilen hücredir. Yeni hücre `satir` ve `sutun` değişkenlerinde durur. Bu yüzden çıkış bulunduğunda tam yol üç parçadan oluşur: yığındaki konumlar (`yigit[0]` 'dan `yigit[top]` 'a), o anda bulunulan hücre ve çıkış hücresi. Sunumun belirttiği gibi yordamın son kısmı bu üç parçayı sırayla yazdırır; çıkış bulunamadıysa "Cikis yolu bulunamadi!" iletisi basılır. Doğrulama programında bu son kısım şöyle tamamlandı:

```
if (bulundu) {
  printf("satir sutun\n");
  for (i = 0; i <= top; i++)        /* yığındaki yol */
    printf("%2d%5d\n", yigit[i].satir, yigit[i].sutun);
  printf("%2d%5d\n", satir, sutun);      /* son konum */
  printf("%2d%5d\n", CIKIS_SATIR, CIKIS_SUTUN);
} else printf("Cikis yolu bulunamadi!\n");
```

Üçüncüsü, `bulundu`, `TRUE` ve `FALSE` C99'un `bool` tipinden önceki bir alışkanlıktır: `#define FALSE 0` ve `#define TRUE 1` ile tanımlanan tam sayılardır. `yigit_ekle` ve `yigit_al` ise yığın gerçeklemesinin bilinen iki fonksiyonudur; `top` 'u adresle alırlar, çünkü onu değiştirirler. Dördüncüsü, `konum.yon = ++yon` satırı hem kayda yazılacak değeri hesaplar hem de yerel `yon` 'u değiştirir. Hemen ardından `yon = 0` yazıldığı için yerel değişikliğin bir etkisi kalmaz; önemli olan kayda yazılan değerdir.

> **Sık yapılan hata**
>
> Yeni hücreyi işaretlemeyi unutmak. `gezildi[y_satir][y_sutun] = 1` satırı yazılmazsa algoritma `(1, 1)` 'den `(2, 1)` 'e geçer, oradan kuzeye bakar ve yeniden `(1, 1)` 'e döner. İki hücre arasında gidip gelir, her geçişte yığına bir kayıt ekler. Yığın sonunda taşar ya da program sonsuz döngüde kalır. Ziyaret işareti, sonlanmanın garantisidir.

Sunumun adım adım izleme slaytı, iç döngünün değişken adlarını kısaltarak (`ys` , `yy`) ve yeni kaydı tek satırda kurarak aynı mantığı daha kısa yazar:

```
while (top > -1 && !bulundu) {
  konum = yigit_al(&top);
  satir = konum.satir;  sutun = konum.sutun;
  yon = konum.yon;
  while (yon < 8 && !bulundu) {
    ys = satir + hareket[yon].dikey;
    yy = sutun + hareket[yon].yatay;
    if (ys == CIKIS_SATIR && yy == CIKIS_SUTUN)
      bulundu = TRUE;
    else if (!labirent[ys][yy] &&
             !gezildi[ys][yy]) {
      gezildi[ys][yy] = 1;
      konum = (element){satir, sutun, ++yon};
      yigit_ekle(&top, konum);
      satir = ys;  sutun = yy;  yon = 0;
    } else ++yon;
} }
```

Buradaki `(element){satir, sutun, ++yon}` ifadesi C99'la gelen **bileşik değişmezdir** (*compound literal*). Adı olmayan geçici bir `element` nesnesi kurar ve alanlarını sırayla doldurur. Üç ayrı atamanın yaptığı işi tek satırda yapar. Bu sürüm de doğrulama programında ayrı olarak çalıştırıldı ve aynı yığın hareketlerini üretti.

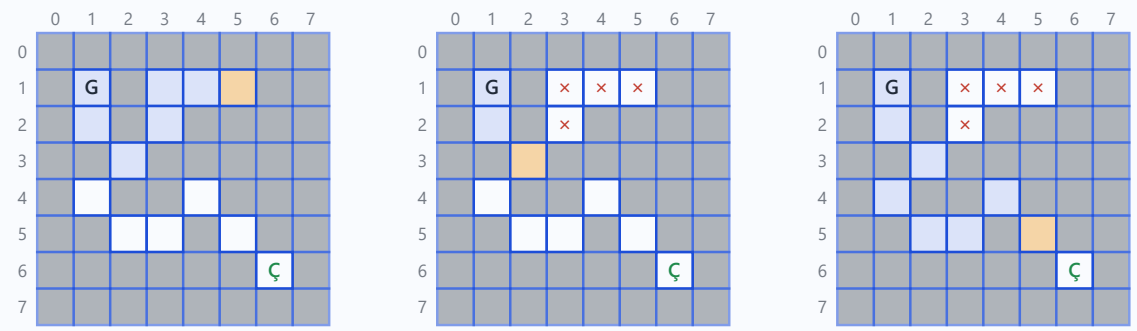
8.2.5 Kod izleme: yığınla geri izleme

Yordamı Şekil 8.1'deki labirent üzerinde çalıştıralım. Her adımda yığının içeriği `satir,sutun,yon` üçlüleriyle, alttan üste doğru yazılır. Tablo 8.2 yığın işlemlerini sırasıyla verir; satırlar doğrulama programının ürettiği çıktıyla birebir aynıdır.

Adım	Olay	Geçiş	Yığın (alttan üste)	`top`
0	Başlangıç	—	1,1,1	0
1	pop (1,1), yön 1'den başla	—	(boş)	−1
2	push (1,1, yön 5)	(1,1) yön 4 (G) → (2,1)	1,1,5	0
3	push (2,1, yön 4)	(2,1) yön 3 (GD) → (3,2)	1,1,5 · 2,1,4	1
4	push (3,2, yön 2)	(3,2) yön 1 (KD) → (2,3)	… · 3,2,2	2
5	push (2,3, yön 1)	(2,3) yön 0 (K) → (1,3)	… · 3,2,2 · 2,3,1	3
6	push (1,3, yön 3)	(1,3) yön 2 (D) → (1,4)	… · 2,3,1 · 1,3,3	4
7	push (1,4, yön 3)	(1,4) yön 2 (D) → (1,5)	… · 1,3,3 · 1,4,3	5
8	(1,5) çıkmaz; pop (1,4)	yön 3…7 kapalı	… · 2,3,1 · 1,3,3	4
9	pop (1,3)	yön 3…7 kapalı	… · 3,2,2 · 2,3,1	3
10	pop (2,3)	yön 1…7 kapalı	1,1,5 · 2,1,4 · 3,2,2	2
11	pop (3,2)	yön 2'den devam	1,1,5 · 2,1,4	1
12	push (3,2, yön 6)	(3,2) yön 5 (GB) → (4,1)	1,1,5 · 2,1,4 · 3,2,6	2
13	push (4,1, yön 4)	(4,1) yön 3 (GD) → (5,2)	… · 3,2,6 · 4,1,4	3
14	push (5,2, yön 3)	(5,2) yön 2 (D) → (5,3)	… · 4,1,4 · 5,2,3	4
15	push (5,3, yön 2)	(5,3) yön 1 (KD) → (4,4)	… · 5,2,3 · 5,3,2	5
16	push (4,4, yön 4)	(4,4) yön 3 (GD) → (5,5)	… · 5,3,2 · 4,4,4	6
17	Çıkış bulundu	(5,5) yön 3 (GD) → (6,6) = Ç	yığın + (5,5) + (6,6)	6

Tablo 8.2: `izbul` izlemesi. "Geçiş" sütunu o anda bulunulan hücreden açık bulunan ilk yönü ve varılan hücreyi verir.

İlk adımlar düz ilerler. `(1,1)`'de yön 1, 2 ve 3 duvara çıkar; yön 4 (güney) `(2,1)`'e açılır. Bu geçişte `(1,1)` yön 5 ile yığına konur: geri dönülürse güneybatıdan aramaya devam edilecektir. `(2,1)`'den güneydoğuya, oradan kuzeydoğuya, oradan kuzeye çıkılır. Algoritma `(1,3)`, `(1,4)` ve `(1,5)` hücreleriyle labirentin üst kenarındaki koridora girer.



Şekil 8.3: İzlemenin üç anı. Solda algoritma `(1,5)`'e varmıştır; bu hücrenin sekiz komşusundan hiçbiri açık ve ziyaret edilmemiş değildir. Ortada dört geri adımdan sonra `(3,2)`'ye dönülmüş, arama yön 2'den sürer. Sağda çıkış bulunmuştur; yol yığındaki konumlar, `(5,5)` ve `(6,6)`'dır.

`(1,5)` bir çıkmazdır. Sekiz komşusunun yedisi duvar, biri (`(1,4)`) ziyaret edilmiştir. İç döngü `yon == 8` ile biter ve dış döngü yığından `(1,4, yön 3)`'ü alır. Bu, bir adım geri gitmektir. `(1,4)`'te yön 3'ten itibaren kalan yönler denenir: güneydoğu `(2,5)` ve güney `(2,4)` duvardır, güneybatı `(2,3)` ve batı `(1,3)` ziyaret edilmiştir, kuzeybatı duvardır. Hiçbir yön açılmaz; yeniden geri gidilir. Aynı durum `(1,3)` ve `(2,3)` için de tekrarlanır. Dört geri adımdan sonra `(3,2)`'ye varılır ve yön 2'den aramaya devam edilir.

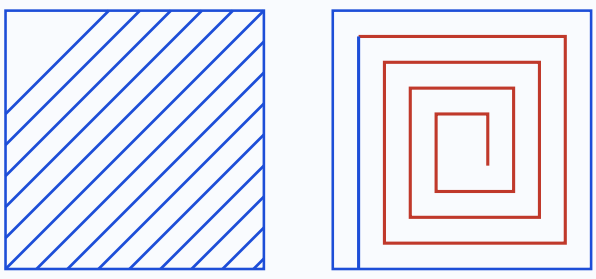
Burada yön alanının değeri ortaya çıkar. `(3,2)`'de yön 0 ve yön 1 daha önce denenmiştir; yön 1 zaten çıkmaz koridora götürmüştü. Kayıttaki 2 değeri bu iki yönün tekrar denenmesini önler. Yön 2, 3 ve 4 duvardır; yön 5 (güneybatı) `(4,1)`'e açılır. Buradan sonra yol çıkışa doğru tek bir koridorla ilerler: `(4,1)` → `(5,2)` → `(5,3)` → `(4,4)` → `(5,5)`. `(5,5)`'te yön 3 (güneydoğu) çıkış hücresine denk gelir ve arama biter.

Bulunan yol şudur: `(1,1)`, `(2,1)`, `(3,2)`, `(4,1)`, `(5,2)`, `(5,3)`, `(4,4)`, `(5,5)`, `(6,6)`. Toplam 9 hücredir. İlk yedisi yığından okunur (`top = 6`, yani 7 kayıt), sekizincisi o anki konum, dokuzuncusu çıkıştır. Algoritma boyunca 11 kez üste koyma, 5 kez üstten alma yapıldı. Yığın en çok 7 kayda ulaştı. Çıkmaz koridordaki dört hücre (`(2,3)`, `(1,3)`, `(1,4)`, `(1,5)`) ziyaret edildi ama yolda yer almadı. Çıkış dışında 12 açık hücrenin hepsi gezildi.

8.2.6 En kötü durum

Labirent programı bir **kaba kuvvet** (*brute force*) aramasıdır. Hiçbir zekâ kullanmaz: çıkışın hangi yönde olduğuna bakmaz, yönleri hep aynı sırayla dener. Sekizli arama tablosu kodu kısaltmış ve işi kolaylaştırmıştır, ama aramanın kendisi körüdür. Böyle bir algoritmanın ne kadar iş yapacağı büyük ölçüde girdiye bağlıdır. Bazı labirentlerde çıkış birkaç adımda bulunur, bazılarında labirentin neredeyse tamamı gezilir.

Bir algoritmanın en kötü durum maliyetini söylemek için yalnız bir üst sınır hesaplamak yetmez; o sınıra gerçekten ulaşan bir girdinin var olduğunu da göstermek gerekir. Bu girdiye **en kötü veri grubu** denir. Labirentte en kötü veri grubunu belirleyen şey, yönlerin tarama sırasıdır. Algoritma her hücrede önce kuzeye, sonra kuzeydoğuya... bakar. Bu sırayla ilk denenen yol çok uzunsa, algoritma onu sonuna kadar izler.



Şekil 8.4: İki uç girdi. Girdi 1'de ilk denenen yol uzun ama doğrudur. Girdi 2'de ilk denenen yol (kırmızı sarmal) uzun ve yanlıştır; çıkışa giden kısa yol (mavi) ancak sarmal bütünüyle gezildikten sonra denenir.

Şekil 8.4 iki girdiyi karşılaştırır.

- **Girdi 1**: İlk denenen yol uzundur ama doğrudur; çıkışa götürür. Algoritma bu yolda hiç geri dönmez. Yalnız üste koyma yapılır, üstten alma yapılmaz. İş miktarı yolun uzunluğuyla orantılıdır.
- **Girdi 2**: İlk denenen yol uzundur ve yanlıştır; bir sarmal gibi labirentin içine kıvrılır ve bir çıkmazda biter. Algoritma bu yolu sonuna kadar gider ve yığın sarmalın her hücresi için bir kayıtla dolar. Çıkmaza varınca yığın bir bir boşaltılır; her geri adımda o hücrenin kalan yönleri de denenir. Ancak sarmalın tamamı tükenince başlangıca dönülür ve kısa doğru yol denenir. Doğru yol dışındaki her yer gezildikten sonra çözüme ulaşılır.

Girdi 2 en kötü veri grubudur: algoritmanın yapabileceği bütün işi yaptırır. Her açık hücre bir kez ziyaret edilir, bir kez yığına konur ve bir kez yığından alınır. Bu analiz bir ilkeyi de gösterir. Bir algoritmayı "yavaş" ya da "hızlı" diye değerlendirmeden önce onun hangi girdide nasıl davrandığı incelenmelidir. Analizi yapılmamış bir algoritma hakkında yargıya varmak doğru olmaz.

8.2.7 Karmaşıklık ve kuyrukla karşılaştırma

`izbul` 'un maliyetini sayarak bulalım. Labirentin boyutu $m \times p$ olsun.

- **Zaman.** Bir hücre yığına ancak ondan ziyaret edilmemiş bir hücreye geçilirken konur ve her hücre `gezildi` sayesinde en çok bir kez ziyaret edilir. Bu yüzden toplam üste koyma sayısı en çok açık hücre sayısı kadardır. Üstten alma sayısı da bunu aşamaz. Her hücrede iç döngü en çok sekiz yön dener; yön alanı sayesinde geri dönüşte daha önce denenmiş yönler tekrar denenmez. Bir hücre için toplam yön denemesi bu yüzden en çok 8'dir, artı her alma için bir tur. Sonuç: $O(8 \cdot m \cdot p) = O(m \cdot p)$. En kötü veri grubunda bu sınıra ulaşılır, yani en kötü durum $\Theta(m \cdot p)$'dir.
- **Bellek.** Yığın en çok açık hücre sayısı kadar büyür: $O(m \cdot p)$. `gezildi` matrisi de $(m + 2)(p + 2)$ hücre tutar, yani $O(m \cdot p)$'dir.

Yığınla yapılan bu arama, **derinlik öncelikli aramanın** (*depth-first search*, DFS) bir örneğidir. Bir yöne gidilir ve olabildiğince derine inilir; geri dönüş ancak çıkmazda olur. Bu aramanın önemli bir zayıflığı vardır: **bulunan yol en kısa olmak zorunda değildir**. Algoritma ilk bulduğu yolu döndürür. O yol, yönlerin deneme sırası yüzünden labirentin uzak bir köşesinden dolanıyor olabilir. Bu labirentte arama `(2,3)` → `(1,3)` → `(1,4)` → `(1,5)` çıkmazına girip döndü ve 9 hücrelik bir yol buldu.

Yığının yerine **kuyruk** kullanılırsa aramanın karakteri tümüyle değişir. Kuyruk "ilk giren ilk çıkar" (*first in, first out*, FIFO) kuralıyla çalışır. Başlangıç hücresi kuyruğa konur. Her adımda kuyruğun önünden bir hücre alınır ve onun ziyaret edilmemiş bütün açık komşuları kuyruğun arkasına eklenir. Böylece hücreler girişe uzaklık sırasıyla açılır: önce uzaklığı 1 olanlar, sonra uzaklığı 2 olanlar... Bu yönteme **genişlik öncelikli arama** (*breadth-first search*, BFS) denir.

> **Tanım 8.3**
>
> **Derinlik öncelikli arama** (DFS), bir komşuya gidip olabildiğince derine inen ve çıkmazda en son ayrıma dönen aramadır; yığınla ya da özyinelemeyle yapılır. **Genişlik öncelikli arama** (BFS), düğümlere başlangıçtan uzaklık sırasıyla uğrayan aramadır; kuyrukla yapılır. Bütün adımların maliyeti eşitse BFS'nin bir hücreye ilk ulaştığı yol, o hücreye giden **en kısa yoldur**.

BFS'nin en kısa yolu neden garanti ettiği kısaca şöyle açıklanır. Kuyrukta hücreler her zaman uzaklıklarına göre sıralıdır: önce uzaklığı d olanların hepsi, sonra uzaklığı $d + 1$ olanlar. Bir hücreye ilk kez ulaşıldığında ona bir uzaklık yazılır ve hücre bir daha kuyruğa girmez. Uzaklığı d olan bir hücreye daha kısa bir yoldan ulaşılabilseydi, o yolun bir önceki hücresi daha erken kuyruktan çıkardı ve hücreye ondan önce ulaşılırdı. Bu yüzden ilk ulaşma her zaman en kısa yoldandır.

Bu labirentte BFS de 9 hücrelik yolu bulur. Yani burada iki arama aynı uzunlukta yol verdi; doğrulama programındaki BFS bunu 9 hücre olarak hesapladı. Bunun nedeni labirentin dar olmasıdır: çıkışa giden tek bir koridor vardır. Genel durumda DFS'nin bulduğu yol çok daha uzun olabilir. BFS'nin bedeli de vardır: yolu geri kurabilmek için her hücrenin hangi hücreden ulaşıldığını saklamak gerekir. İki aramanın zaman ve bellek maliyeti aynı mertebededir: $O(m \cdot p)$. BFS ile en kısa yolu bulan tam kod alıştırma çözümlerinde verilir.

Bu iki algoritmanın kodlarını en iyi inceleyebilecek kişi onları yazandır. Bir başkası, kodu adım adım izlemeden onun ne yaptığını tam olarak analiz edemez. Kod izleme tabloları bu yüzden önemlidir: hem yazana kodunu sınama fırsatı verir hem de okuyana kodun davranışını görünür kılar.

8.3 İfade gösterimleri

Yığının en eski ve en yaygın uygulamalarından biri aritmetik ve mantıksal ifadelerin değerlendirilmesidir. Bir derleyici `x = a / b - c + d * e - a * c` satırını okuduğunda, bu ifadeyi hangi sırayla hesaplayacağına karar vermek zorundadır. Bu kesim sorunu tanımlar ve çözümün dayandığı üç gösterimi tanıtır.

8.3.1 İfade değerlendirme sorunu

İfade değerlendirme (*evaluation of expressions*), bir ifadenin değerini işleçlerin önceliğine ve birleşme yönüne uyarak hesaplamaktır. İfade iki tür öğeden oluşur. **İşleçler** (*operator*) `+`, `-`, `*`, `/` gibi yapılacak işlemi belirten simgelerdir. **İşlenenler** (*operand*) ise işlemin uygulandığı değerlerdir: `a`, `b` gibi değişkenler ya da `4`, `2` gibi sabitler. Parantezler de ifadenin bir parçasıdır ve hesaplama sırasını değiştirir.

Sorunun neden zor olduğunu bir örnekle görelim. `a = 4`, `b = 2`, `c = 2`, `d = 3`, `e = 3` olsun. `a / b - c + d * e - a * c` ifadesi matematikte şöyle okunur: önce bölme ve çarpmalar, sonra soldan sağa toplama ve çıkarmalar. Yani `((4 / 2) - 2) + (3 * 3) - (4 * 2)`. Bu da *2 − 2 + 9 − 8 = 1* eder. Aynı ifadeyi soldan sağa, önceliğe bakmadan hesaplayan biri ise *((((4/2) − 2) + 3) · 3 − 4) · 2 = 10* bulurdu. Doğru sonuç 1'dir.

İnsan ifadeye bütün olarak bakar ve önce çarpmaları seçer. Bilgisayar ise ifadeyi soldan sağa, karakter karakter okur. Okurken bir işleci gördüğü anda onu uygulayamaz, çünkü sağında daha öncelikli bir işleç olabilir. `a + b * c`'de `+` görüldüğünde `a + b` hesaplanırsa yanlış olur; önce `b * c` gerekir. Parantezler işi daha da karıştırır: `a / (b - c) + d * (e - a) * c` ifadesinde parantez içindeki çıkarmalar önce yapılmalıdır.

Derleyicilerin yıllardır kullandığı çözüm iki aşamalıdır:

1. İfade, işleç önceliğini ve parantezleri hesaba katarak **sonek** gösterime çevrilir. Bu gösterimde parantez ve öncelik kalmaz; işleçler uygulanacakları sırada dizilir.
2. Sonek ifade soldan sağa **tek geçişte** hesaplanır.

İki aşama da bir yığınla yapılır. Bu yeni bir buluş değildir. Bu problemi pek çok kişi defalarca çözmüştür ve iyi bilinen bir çözümü vardır. Bir problemle karşılaşınca önce bu tür bilinen çözümler araştırılmalı, en hızlı yol bulunmalıdır. Problem gerçekten çözülmemişse keşif ancak o zaman başlar.

8.3.2 Öncelik ve birleşme yönü

Bir ifadede hangi işlecin önce uygulanacağını iki kural belirler.

> **Tanım 8.4**
>
> **İşleç önceliği** (*operator precedence*), farklı işleçler arasında hangisinin önce uygulanacağını belirleyen sıralamadır: C'de `*`, `+`'dan önce gelir. **Birleşme yönü** (*associativity*), aynı öncelikteki işleçlerin soldan sağa mı sağdan sola mı uygulanacağını belirler: `a - b - c`, `(a - b) - c` demektir, çünkü `-` soldan sağa birleşir.

Tablo 8.3 C'nin öncelik tablosunun bu not için gereken kısmını özetler. Satırlar yukarıdan aşağı önceliği azalan sırada verir.

Öncelik	İşleçler	Birleşme
en yüksek	`() [] -> .`	soldan sağa
	tekli `- ! ++ -- * &`	sağdan sola
	`* / %`	soldan sağa
	`+ -`	soldan sağa
	`< <= > >=`	soldan sağa
	`== !=`	soldan sağa
	`&&`	soldan sağa
en düşük	`||`	soldan sağa

Tablo 8.3: C'de işleç önceliği (özet). Üstteki satırlar önce uygulanır.

Tablo okunurken önce öncelik, öncelik eşitse birleşme yönü karar verir. Parantez her ikisini de geçersiz kılar: parantez içindeki kısım her zaman önce hesaplanır. İkinci satırdaki tekli işleçler (eksi işareti, mantıksal değil, artırma ve azaltma, içerik ve adres işleçleri) sağdan sola birleşir. Örneğin `*p++`, `*(p++)` demektir. Tablo C'nin bütün işleçlerini içermez; kaydırma, bit düzeyi, koşul ve atama işleçleri burada yer almaz.

Örnek 8.1

`a / b - c + d * e - a * c` ifadesini kurallara göre parantezleyelim. Önce en öncelikli işleçler: `/` ve iki `*`. Bunlar `(a / b)`, `(d * e)` ve `(a * c)` olur. Geriye üç toplama/çıkarma kalır: `(a/b) - c + (d*e) - (a*c)`. Hepsi aynı önceliktedir ve soldan sağa birleşir. Önce soldaki `-`, sonra `+`, en son sağdaki `-` uygulanır. Sonuç: `(((a / b) - c) + (d * e)) - (a * c)`.

Bu notun algoritmaları yalnız beş ikili işleçle (`+ - * / %`) ve parantezle sınırlıdır. Tekli eksi, karşılaştırma ve mantıksal işleçler yöntemi değiştirmez; yalnız öncelik tablosu genişler. Horowitz, Sahni ve Anderson-Freed'in kitabındaki tablo (Şekil 3.12) bu işleçlerin tamamını içerir.

Sık yapılan hata

Birleşme yönünü unutup aynı öncelikteki işleçleri sağdan sola hesaplamak. `8 - 4 - 2` soldan sağa *(8 − 4) − 2 = 2*'dir. Sağdan hesaplanırsa *8 − (4 − 2) = 6* bulunur. Toplama ve çarpmada bu hata sonucu değiştirmez, çünkü bu işlemler birleşme özelliğine sahiptir; çıkarma ve bölmede ise değiştirir.

8.3.3 Ara ek, önek ve sonek

Bir ifade, işlecin işlenenlere göre yerine bağlı olarak üç biçimde yazılabilir. Gösterimlerin adları İngilizcede yaygın olarak *infix*, *prefix* ve *postfix* diye geçer; sunum başlıklarında ve fonksiyon adlarında (`postfix`, `prefix_eval`) bu kısa adlar kullanılır.

Tanım 8.5

Ara ek gösterim (*infix*): işleç iki işlenenin arasına yazılır, `a + b`. Hesaplamak için parantezleri ve öncelikleri dikkate almak gerekir. **Önek gösterim** (*prefix*, Polonya gösterimi): işleç işlenenlerden önce yazılır, `+ a b`. **Sonek gösterim** (*postfix*, ters Polonya gösterimi): işleç işlenenlerden sonra yazılır, `a b +`. Önek ve sonek gösterimde parantez gerekmez; ifade önceliğe bakılmadan hesaplanır.

"Polonya gösterimi" adı, önek gösterimi 1920'lerde mantık formülleri için öneren Polonyalı mantıkçı Jan Łukasiewicz'den gelir. Sonek gösterim onun ters çevrilmiş biçimidir. Ters Polonya gösterimi bir zamanlar bazı hesap makinelerinde de kullanılırdı: kullanıcı önce iki sayıyı girer, sonra işlem tuşuna basardı.

Tablo 8.4 dört ifadenin üç gösterimdeki karşılığını verir. Tablodaki her satır doğrulama programındaki `postfix` ve `prefix` fonksiyonlarıyla üretildi.

No	Önek	Ara ek	Sonek
1	`*34`	`3*4`	`34*`
2	`+2*34`	`2+3*4`	`234*+`
3	`-*+abc/da`	`(a+b)*c-d/a`	`ab+c*da/-`
4	`**/a+-bcd-eac`	`(a/(b-c+d))*(e-a)*c`	`abc-d+/ea-*c*`

Tablo 8.4: Aynı ifadenin üç gösterimi.

Tablonun ikinci satırı önceliğin sonek gösterimde nasıl kaybolduğunu gösterir. `2+3*4`'te çarpma önce yapılmalıdır. Sonek karşılığı `234*+`'dır. Soldan okuyan biri önce `2`, `3` ve `4`'ü görür, sonra `*` işlecini görür ve onu en son iki sayıya, yani `3` ile `4`'e uygular. Ardından `+`, `2` ile `12`'yi toplar. Önceliği bilmeye gerek kalmamıştır: işleçlerin sırası, uygulanma sırasının kendisidir.

Üçüncü satırda parantez kaybolur. `(a+b)*c`'de toplama önce yapılmalıdır. Sonek karşılığında `+`, `*`'dan önce yazılır: `ab+c*`. Sonek gösterimde parantezin işlevini işleçlerin konumu üstlenir.

Dördüncü satır daha karmaşıktır. `b-c+d` soldan sağa birleştiği için `(b-c)+d`'dir ve sonek karşılığı `bc-d+` olur. Bölme bunun önüne `a`'yı ekler: `abc-d+/`. Sonra `(e-a)` ile çarpılır (`ea-*`) ve en son `c` ile çarpılır (`c*`). İki çarpma soldan sağa birleştiği için önce soldaki yapılır.

8.3.4 Elle çevirme

Ara ek bir ifadeyi elle sonek gösterime çevirmenin üç adımlı basit bir yolu vardır. Tablo 8.5 bu adımları `a / b - c + d * e - a * c` ifadesine uygular.

Adım	İfade
Başlangıç	`a / b - c + d * e - a * c`
1. Tam parantezle	`((((a / b) - c) + (d * e)) - (a * c))`
2. İşleci sağ kapanışa taşı	`((((a b /) c -) (d e *) +) (a c *) -)`
3. Parantezleri sil	`a b / c - d e * + a c * -`

Tablo 8.5: Elle çevirme: parantezle, taşı, sil.

1. **Tam parantezleme.** Öncelik ve birleşme yönüne göre her işlem kendi parantezine alınır (Örnek 8.1). Her işlecin tam olarak bir parantez çifti olur.
2. **Taşıma.** Her işleç kendi parantez çiftinin sağ kapanışına taşınır. `(a / b)` çifti `(a b /)` olur.
3. **Silme.** Parantezler silinir. Kalan dizi sonek ifadedir: `a b / c - d e * + a c * -`.

Önek gösterim için ikinci adımda işleç sağ kapanışa değil, kendi çiftinin **sol açılışına** taşınır. Aynı ifadenin önek karşılığı `- + - / a b c * d e * a c`'dir.

Çevirinin sonucunda önemli bir gözlem yatar: **işlenenlerin sırası değişmez**. Ara ek ifadede işlenenler `a b c d e a c` sırasındadır; sonek ifadede de aynı sıradadır. Yalnız işleçlerin yeri değişir. Bu gözlem yığınlı algoritmanın temelidir. İşlenen geldiği anda doğrudan çıktıya yazılabilir. İşleçler ise uygun an gelene kadar, yani kendilerinden sonra gelen ve daha öncelikli olan işlemler bitene kadar bir yığında bekler.

> **Sık yapılan hata**
>
> Elle çevirirken tam parantezleme adımını atlayıp yalnız ifadede görünen parantezlerle çalışmak. `a / b - c`'de hiç parantez yoktur; taşınacak bir kapanış da yoktur. Önce `((a / b) - c)` diye bütün işlemleri parantezlemek gerekir. Aksi hâlde sonuç rastgele bir dizilişe dönüşür.

8.3.5 Sonekten ara eke yığınla

Ters yöndeki çeviri, yani sonek ifadeden tam parantezli ara ek ifade elde etmek, yığının nasıl çalıştığını görmek için iyi bir başlangıçtır. Kural kısadır. İfade soldan sağa okunur:

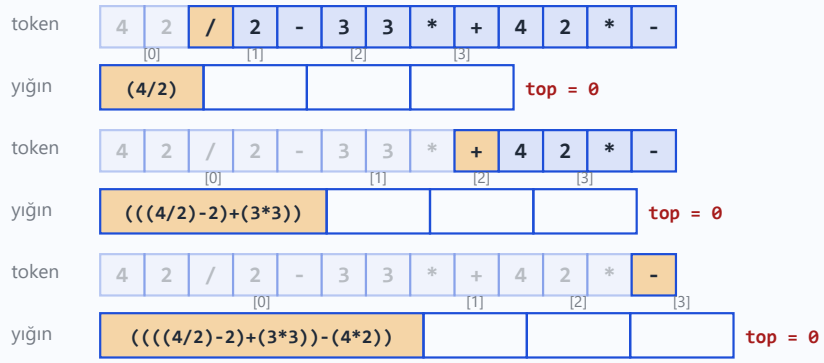
- İşlenen gelirse yığına konur.
- İşleç gelirse yığından iki öğe alınır. İlk alınan sağ işlenen, ikinci alınan sol işlenendir. İkisi, araya işleç konarak ve bir parantez çiftine alınarak birleştirilir. Oluşan dizgi yığına geri konur.
- İfade bitince yığında tek bir öğe kalır: tam parantezli ara ek ifade.

Sunumdaki örnek `42/2-33*+42*-` ifadesidir. Bu, `a / b - c + d * e - a * c` ifadesine `a = 4`, `b = 2`, `c = 2`, `d = 3`, `e = 3` değerleri konarak elde edilen sonek ifadedir. Tablo 8.6 her adımdan sonra yığını verir.

Belirteç	İşlem	Yığın
`4`	işlenen, yığına	4
`2`	işlenen, yığına	4 · 2
`/`	2 ve 4 alınır, `(4/2)` konur	(4/2)
`2`	işlenen, yığına	(4/2) · 2
`-`	`((4/2)-2)` konur	((4/2)−2)
`3`, `3`	iki işlenen yığına	((4/2)−2) · 3 · 3
`*`	`(3*3)` konur	((4/2)−2) · (3*3)

Belirteç	İşlem	Yığın
`+`	`(((4/2)-2)+(3*3))` konur	(((4/2)−2)+(3*3))
`4` , `2`	iki işlenen yığına	(((4/2)−2)+(3*3)) · 4 · 2
`*`	`(4*2)` konur	(((4/2)−2)+(3*3)) · (4*2)
`-`	son birleştirme	((((4/2)−2)+(3*3))−(4*2))

Tablo 8.6: `42/2-33*+42*-` ifadesinin ara eke çevrilmesi. Yığın soldan sağa, alttan üste yazılmıştır.



Şekil 8.5: Sonekten ara eke çevirmenin üç anı. Üstte ilk `/` işlendi ve `(4/2)` yığının dibinde duruyor. Ortada `+` işlendi; yığında tek dizgi var. Altta son `-` işlendi; yığındaki tek öğe sonuçtur.

İki `*` adımında yığının durumu ilgi çekicidir. İlk `*` işlendikten sonra yığında iki öğe vardır: `((4/2)-2)` ve `(3*3)`. Yani `[1]` hücresi boş kalmaz; orada `(3*3)` durur ve `top = 1` 'dir. İkinci `*` 'dan sonra da aynı hücrede bu kez `(4*2)` durur. Yığın hiçbir anda üç öğeyi geçmez. Bu ifadede en derin an `3 3` ya da `4 2` çiftinin yığına konduğu andır.

Bu yöntem C'de yazılırken yığında karakter değil, dizgi tutulur. Her birleştirme yeni bir dizgi için bellek ayırmayı ve artık gereksiz olan iki parçayı serbest bırakmayı gerektirir. Bu yüzden `malloc` ve `free` 'nin dikkatli kullanılması gerekir. Tam kod alıştırma çözümlerinde verilir.

8.4 Sonek ifadeyi hesaplama

Sonek ifadenin hesaplanması, iki aşamalı çözümün ikinci aşamasıdır ama anlaşılması daha kolay olanıdır. Bu yüzden önce onu inceleyelim. Kural, sonekten ara eke çevirmedeki kuralın neredeyse aynısıdır. Tek fark, yığında dizgiler yerine sayıların durmasıdır.

- İşlenen gelirse değeri yığına konur.
- İşleç gelirse yığından iki değer alınır, işlem yapılır ve sonuç yığına konur.
- İfade bitince yığında kalan tek değer sonuçtur.

8.4.1 Belirteç: `token_al`

Hesaplayıcı ifadeyi karakter karakter okur. Her okumada bir **belirteç** (*token*) elde edilir. Belirteç, bir ifadenin anlam taşıyan en küçük parçasıdır: bir işleç, bir parantez, bir işlenen ya da ifadenin sonu. Gerçek derleyicilerde bir belirteç birden çok karakterden oluşabilir; `123` ya da `toplam` tek bir belirteçtir. Bu notta işi sade tutmak için her işlenenin tek basamaklı bir sayı olduğu varsayılır. Böylece her karakter bir belirteçtir.

```
typedef enum { sol_paran, sag_paran, arti, eksi,
    carpma, bolme, mod, eos, operand } oncelik;

oncelik token_al(const char *expr, char *sembol, int *n) {
    *sembol = expr[(*n)++];
    switch (*sembol) {
        case '(':  return sol_paran;
        case ')':  return sag_paran;
        case '+':  return arti;
        case '-':  return eksi;
        case '/':  return bolme;
        case '*':  return carpma;
        case '%':  return mod;
        case '\0': return eos;
        default:   return operand;
    }
}
```

Kodun ilk kısmı bir **sayım tipi** (*enumeration*, `enum`) tanımlar. `typedef enum { … } oncelik;` bildirimi dokuz adlandırılmış tam sayı sabiti üretir: `sol_paran` 0, `sag_paran` 1, `arti` 2 … `operand` 8. Bu sabitler belirtecin türünü etiketler. Kodun geri kalanında `'+'` karakteri yerine `arti` gibi okunur bir ad kullanılır. `typedef enum` ile etiketleme yapmak kodu kolaylaştırır: hem okunurluk artar hem de derleyici, bir `switch` içinde unutulan etiketler için uyarı verebilir.

`token_al` fonksiyonu üç parametre alır. `expr` hesaplanan ifadedir. `sembol`, okunan karakterin yazılacağı yerin adresidir; böylece fonksiyon hem belirtecin türünü döndürür hem de karakterin kendisini çağırana iletir. `n` ise ifadede o anda okunan konumun adresidir. `*sembol = expr[(*n)++]` satırı `n`'nin gösterdiği konumdaki karakteri okur ve konumu bir ilerletir. Sonraki çağrı bir sonraki karakteri okur. Ardından `switch` karakterin türünü belirler. İşleç ve parantez karakterleri kendi etiketlerini, `'\0'` ifadenin sonunu (`eos`, *end of string*), geri kalan her şey `operand` etiketini döndürür.

Doğrulama programı `"(7%3)"` ifadesini belirteç belirteç okuttu ve sırasıyla `sol_paran` (0), `operand` (8), `mod` (6), `operand` (8), `sag_paran` (1) ve `eos` (7) türlerini aldı.

İşlenen bir karakterdir; hesap için onu sayıya çevirmek gerekir. Bunun yolu `sembol - '0'` ifadesidir. C'de bir karakter, aslında kodunu tutan küçük bir tam sayıdır. ASCII tablosunda rakamlar art arda dizilir: `'0'`'ın kodu 48, `'1'`'inki 49, `'2'`'ninki 50'dir. Bu yüzden `'2' - '0'` işlemi *50 − 48 = 2* verir. Bir rakam karakterinden `'0'` çıkarmak onun sayısal değerini verir.

İfadenin `expr` adıyla küresel bir dizide tutulması da mümkündü. Horowitz ve arkadaşlarının kitabında böyledir: `get_token` fonksiyonu yalnız iki parametre alır ve `expr`'i küresel değişkenden okur. Sunumdaki sürüm ifadeyi parametre olarak alır. Bu küçük fark, bir sonraki kesimde tartışılan "bileşen gibi yazma" ilkesinin ilk adımıdır.

8.4.2 `eval`: küresel sürüm

Sonek ifadeyi hesaplayan yordamın ilk sürümü, kitaptaki gibi küresel bir yığın ve küresel bir ifade kullanır:

```
float eval(void) {
    char sembol;  oncelik token;
    float op1, op2;
    int n = 0, top = -1;
    token = token_al(expr, &sembol, &n);
    while (token != eos) {
        if (token == operand)
            yigit_ekle(&top, sembol - '0');
        else {
            op2 = yigit_al(&top);   /* önce sağ işlenen */
            op1 = yigit_al(&top);
            yigit_ekle(&top, islem(token, op1, op2));
        }
        token = token_al(expr, &sembol, &n);
    }
    return yigit_al(&top);   /* kalan tek öğe sonuç */
}
```

İfadenin kendisi (`expr`) ve yığın dizisi (`yigit`) küreseldir; `top` ise yereldir ve yığın fonksiyonlarına adresiyle verilir. Yordamın akışı şöyledir:

- `n = 0` ifadenin başını, `top = -1` boş yığını gösterir. İlk belirteç döngüden önce okunur.
- Döngü belirteç `eos` olmadığı sürece döner. Her turun sonunda bir sonraki belirteç okunur. Bu, "önce oku, sonra döngüde işle ve bir sonrakini oku" kalıbıdır.
- Belirteç işlenince, `sembol - '0'` ile sayısal değeri yığına konur.
- Belirteç işleçse yığından iki değer alınır. **İlk alınan** `op2` **'dir, yani sağ işlenen.** Sonra sol işlenen `op1` alınır. `islem` fonksiyonu işlemi yapar ve sonuç yığına konur.
- Döngü bitince yığında kalan tek değer döndürülür.

`islem` fonksiyonu işleç türünü bir `switch` ile ayırır:

```
float islem(oncelik token, float op1, float op2) {
    switch (token) {
        case arti:   return op1 + op2;
        case eksi:   return op1 - op2;
        case carpma: return op1 * op2;
        case bolme:  return op1 / op2;
        case mod:    return (int)op1 % (int)op2;
        default:     return 0;   /* işleç değil: buraya gelinmez */
    }
}
```

İşlenenlerin alınma sırası bu yordamın en önemli ayrıntısıdır. Sonek ifadede sol işlenen yığına önce konur, sağ işlenen sonra. Yığın son konanı ilk verdiği için ilk alınan değer sağ işlenendir. `6 2 -` ifadesinde yığına önce 6, sonra 2 konur. `-` geldiğinde ilk alınan 2'dir (`op2`), ikinci alınan 6'dır (`op1`). Sonuç *op1 − op2 = 6 − 2 = 4*'tür. İşlenenler ters alınsaydı *2 − 6 = −4* bulunurdu. Toplama ve çarpmada sıra sonucu değiştirmez, ama çıkarma, bölme ve kalan işlemlerinde değiştirir.

`mod` durumundaki `(int)op1 % (int)op2` ifadesi bir C kuralından kaynaklanır. Yığın `float` değer tutar, çünkü bölme sonucu kesirli olabilir. `%` işleci ise `float` ile kullanılamaz; yalnız tam sayılar için tanımlıdır. Bu yüzden iki işlenen önce `int` 'e çevrilir. Kesirli değerler için kalanı hesaplamak gerekirse standart kitaplıktaki `fmod` fonksiyonu kullanılır.

`islem` 'deki `default` dalı da gereklidir. Sayım tipi üzerinde yapılan bir `switch` , etiketlerin bir kısmını karşılamazsa derleyici `-Wswitch` uyarısı verir. Burada `sol_paran` , `sag_paran` , `eos` ve `operand` bu fonksiyona hiç gelmez; `default` dalı bunu açıkça belirtir ve uyarıyı giderir.

Bu yordam doğrulama programında küresel `expr = "42/2-33*+42*-"` ile çalıştırıldı ve 1 değerini döndürdü. Sunumdaki ilk sürümde `token_al` iki argümanla, `token_al(&sembol, &n)` biçiminde çağrılıyordu. Bu çağrı, üç parametreli `token_al` tanımıyla derlenmez ("too few arguments"). Doğru çağrı, küresel `expr` 'i de veren `token_al(expr, &sembol, &n)` 'dir.

8.4.3 Bileşen gibi yazmak

Küresel sürüm çalışır, ama ciddi bir tasarım sorunu taşır. Yordam dış dünyaya bağımlıdır:

```
/* kuresel surum: disa bagimli */
char expr[] = "42/2-33*+42*-";
float yigit[MAX_SIZE];
float eval(void);
```

`eval` parametre almaz; hangi ifadeyi hesaplayacağını küresel `expr` 'den, nereye yazacağını küresel `yigit` 'ten öğrenir. Bu bağımlılığın üç sonucu vardır. Birincisi, aynı anda iki ifade hesaplanamaz: ikinci ifadeyi hesaplamak için `expr` 'in içeriğini değiştirmek gerekir. İkincisi, yordam başka bir programa taşınamaz; onu kullanmak isteyen her program aynı adlı küresel değişkenleri tanımlamak zorundadır. Üçüncüsü, yordam yan etkilidir: çalışırken programın başka bir yerinin de kullandığı bir diziyi değiştirir. Programın bir yerindeki hata, bambaşka bir yerde ortaya çıkabilir.

Bileşen gibi yazılmış sürümde her şey parametre ya da yerel değişkendir:

```
/* bileşen: her şey parametre/yerel */
float eval(const char *expr);
/* yığın eval'in içinde yerel dizi */
/* main içinde: */
float a = eval("23*54*+9-");
float b = eval("42/2-33*+42*-");
```

İfade parametre olarak gelir; yığın `eval`'in içinde yerel bir dizidir ve her çağrıda yeniden kurulur. Aynı yordam her yerde, her ifadeyle ve yan etkisiz çalışır. `main`'deki iki çağrı birbirini hiç etkilemez. Kod izlemelerinde bu sürüm kullanılır.

Bu ilke yalnız bu yordam için değil, yazılan her fonksiyon için geçerlidir. Fonksiyonlar bir **bileşen** (*component*) gibi yazılmalıdır: ihtiyaç duyduğu her şeyi parametreyle almalı, sonucunu dönüş değeriyle vermelidir. Küresel değişkenler, fonksiyonun içinden okunan `#define` sabitleri gibi kullanımlar dışa bağımlılık yaratır. Ekip hâlinde çalışılan bir projede herkes böyle bağımsız parçalar yazar. "Benim değişkenlerimi `main`'in üstüne koyun, ben böyle çalışıyorum" diyen bir programcının kodu başkalarının koduyla birleştirilemez. Kişiye özgü bir tarz, ortak çalışmanın önünde engeldir.

Bileşen sürümünün tam kodu şudur:

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Kod küresel sürümden daha kısadır. `token_al` yerine doğrudan `for` döngüsü ifadenin karakterlerini gezer; döngü `'\0'` karakterinde durur. `isdigit(c)` (`<ctype.h>`) karakterin bir rakam olup olmadığını sınar. Yığın işlemleri fonksiyon çağrısı yerine dizi ifadeleriyle yazılmıştır: `y[++top] = …` üste koyma, `y[top--]` üstten almadır. `op2 = y[top--]; op1 = y[top--];` satırı yine önce sağ işleneni alır. `switch` bu kez karakterin kendisi üzerindedir ve dört işleci karşılar. `MAX_SIZE` yerel yığının boyudur; ifadedeki işlenen sayısından büyük olmalıdır.

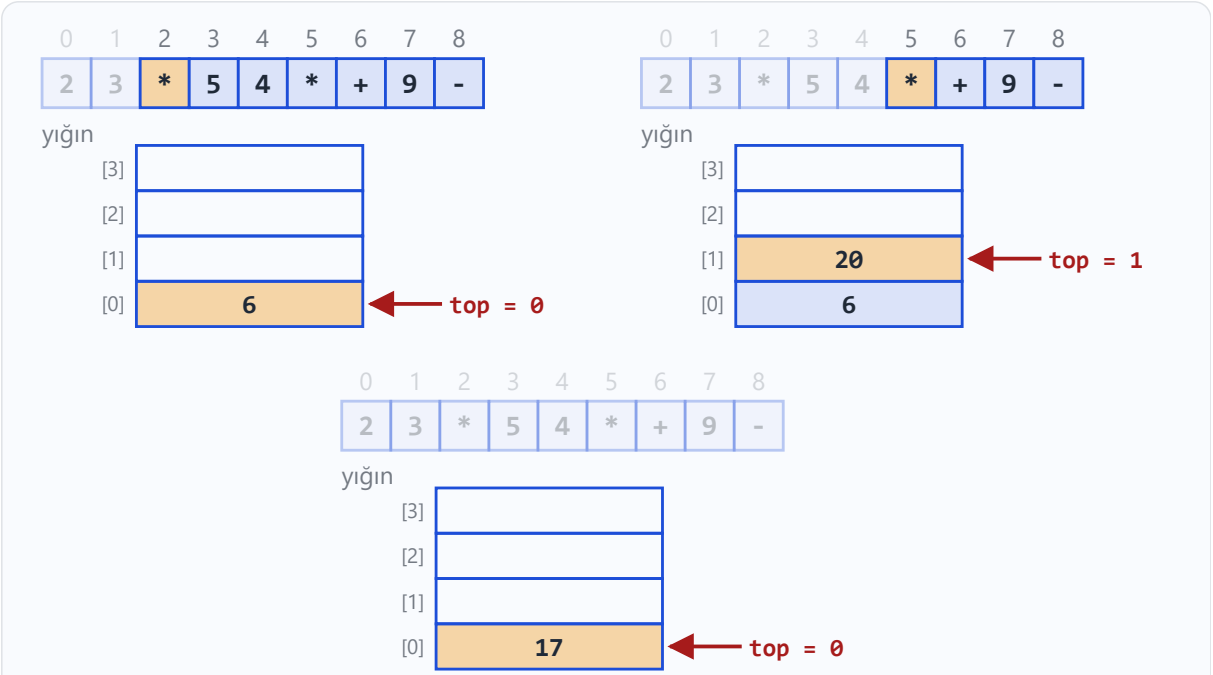
8.4.4 Kod izleme: `eval("23*54*+9-")`

İlk örnek `23*54*+9-`'dir. Bu, ara ek gösterimde `2*3 + 5*4 - 9` ifadesinin sonek karşılığıdır. Değeri *6 + 20 − 9 = 17* olmalıdır. Tablo 8.7 her karakter işlendikten sonra yığının içeriğini verir.

`n`	`c`	İşlem	Yığın	`top`
0	`'2'`	`'2' - '0'` = 2 yığına	2	0
1	`'3'`	3 yığına	2 · 3	1
2	`'*'`	`op2` = 3, `op1` = 2 → 2 * 3 = 6	6	0
3	`'5'`	5 yığına	6 · 5	1
4	`'4'`	4 yığına	6 · 5 · 4	2
5	`'*'`	`op2` = 4, `op1` = 5 → 5 * 4 = 20	6 · 20	1
6	`'+'`	`op2` = 20, `op1` = 6 → 6 + 20 = 26	26	0
7	`'9'`	9 yığına	26 · 9	1

n	c	İşlem	Yığın	top
8	`'-'`	`op2` = 9, `op1` = 26 → 26 − 9 = 17	17	0
9	`'\0'`	döngü biter, `y[0]` döner	17	0

Tablo 8.7: `eval("23*54*+9-")` izlemesi. Yığın alttan üste yazılmıştır.



Şekil 8.6: `eval("23*54*+9-")` izlemesinin üç anı. İlk çizimde ilk `*` işlendi: yığında 6. İkincide ikinci `*` işlendi: yığında 6 ve 20, `top = 1`. Üçüncüde ifade bitti: yığındaki tek değer 17'dir.

İzlemede yığının en derin anı `n = 4`'tür: üç değer (6, 5, 4) aynı anda yığındadır. Bu an, `5*4` çarpımının `2*3` ile toplanmayı beklediği andır. Sonek ifadede bekleyen ara sonuçları yığın saklar.

Maliyet açıktır. İfadenin uzunluğu n olsun. Her karakter bir kez okunur. İşlenen için bir üste koyma, işleç için iki üstten alma, bir işlem ve bir üste koyma yapılır. Bunların her biri $O(1)$'dir. Toplam süre $\Theta(n)$'dir. Yığın en kötü durumda bütün işlenenleri aynı anda tutar (örneğin `1234+++`); ek bellek $O(n)$'dir.

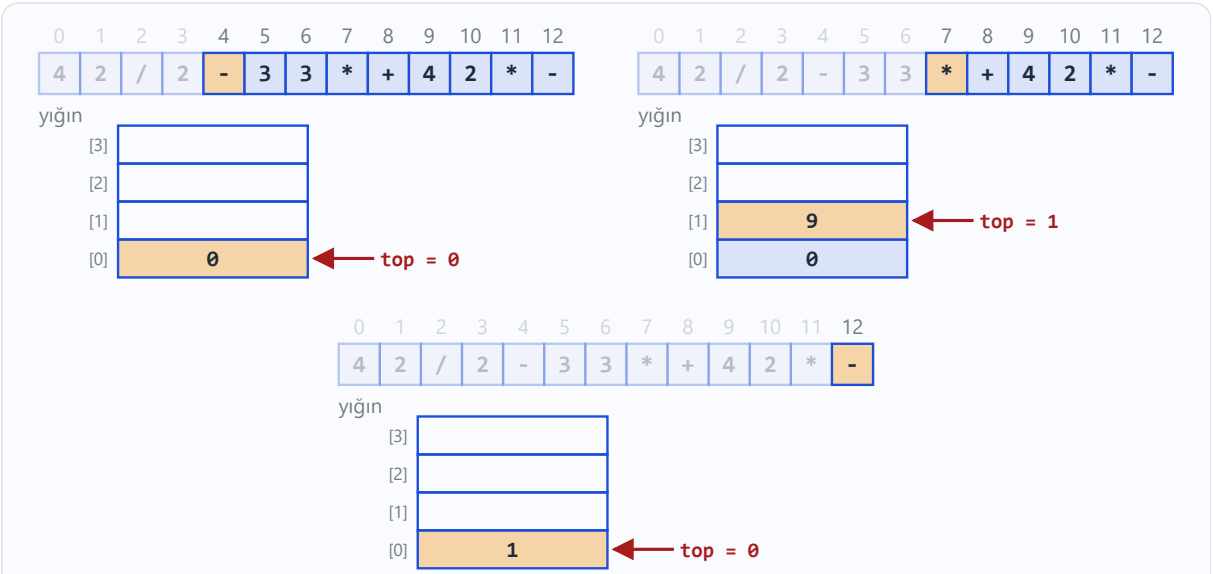
8.4.5 Kod izleme: `eval("42/2-33*+42*-")`

İkinci örnek, ifade değerlendirme sorununu tanıtan ifadenin sonek karşılığıdır: `42/2-33*+42*-`. Beklenen değer 1'dir. Bu ifadenin ara ek karşılığı zaten yığınla bulunmuştu: `((((4/2)-2)+(3*3))-(4*2))`. Şimdi aynı yığın, dizgiler yerine sayılarla çalışır.

c	İşlem	Yığın
`'4'`, `'2'`	4 ve 2 yığına	4 · 2
`'/'`	`op2` = 2, `op1` = 4 → 4 / 2 = 2	2
`'2'`	2 yığına	2 · 2
`'-'`	`op2` = 2, `op1` = 2 → 2 − 2 = 0	0
`'3'`, `'3'`	3 ve 3 yığına	0 · 3 · 3
`'*'`	`op2` = 3, `op1` = 3 → 9	0 · 9
`'+'`	`op2` = 9, `op1` = 0 → 0 + 9 = 9	9
`'4'`, `'2'`	4 ve 2 yığına	9 · 4 · 2
`'*'`	`op2` = 2, `op1` = 4 → 8	9 · 8

`c`	İşlem	Yığın
`'-'`	`op2` = 8, `op1` = 9 → 9 − 8 = 1	1

Tablo 8.8: `eval("42/2-33*+42*-")` izlemesi.



Şekil 8.7: `eval("42/2-33*+42*-")` izlemesinden üç an. İlk çizimde ilk `-`'den sonra yığında 0 vardır. İkincide `3*3` hesaplanmış, 0'ın üstünde 9 durur. Üçüncüde son `-` işlenmiş, sonuç 1'dir.

Bu izleme, ilk karşılaştırmadaki "soldan sağa hesaplayan" yanlış sonucun neden çıkmadığını gösterir. `+` işleci geldiğinde yığında 0 ve 9 vardır; yani `3*3` çarpımı, toplamadan önce hesaplanmıştır. Önceliği bilen, çeviri aşamasıdır. Hesaplayıcı yalnız işleçleri geldikleri sırayla uygular.

Doğrulama programı bu iki ifadeyi ve alıştırmadaki `623+-382/+*` ifadesini hesapladı: sonuçlar 17, 1 ve 7'dir. Programın ara yığın dökümü tablolardaki her satırla aynıdır.

8.4.6 Hatalı ifadeler

İyi kurulmuş bir sonek ifadede işlem sonunda yığında tam olarak bir değer kalır. Bu kural bir geçerlilik testi olarak da kullanılabilir. İfade boyunca bir sayaç tutulsun: işlenen sayacı bir artırır, ikili işleç bir azaltır (iki değer alıp bir değer koyar). İfade geçerliyse her işleç geldiğinde sayaç en az 2'dir ve sonunda tam 1 olur.

- **İşlenen eksikse** (örneğin `2*`), işleç geldiğinde yığında tek değer vardır. Sunumdaki kısa kod `y[top--]`'yu iki kez çalıştırır ve `top` −1'e düştükten sonra `y[-1]`'i okumaya kalkar. Bu, dizi dışına okumadır ve tanımsız davranıştır.
- **İşleç eksikse** (örneğin `234+`), ifade bittiğinde yığında iki değer kalır. Kod üstteki değeri döndürür ve hatayı fark etmez.

Sağlam bir yordam her üstten almadan önce `top >= 0` denetimi yapmalı, sonda da `top == 0` olduğunu sınamalıdır. Sunumdaki `yigit_al` fonksiyonu boş yığında hata iletisi vererek bu denetimi üstlenir. Sıfıra bölme de ayrıca ele alınmalıdır: `float` ile yapılan bölmede sonuç sonsuz ya da NaN (*not a number*) çıkar, `%` işleminde ise bölen sıfırken kalanı bulmak tanımsız davranıştır.

> **Sık yapılan hata**
>
> `op1` ile `op2`'yi ters almak: `op1 = y[top--]; op2 = y[top--];`. Bu kodla `62-` ifadesi 4 yerine −4, `82/` ifadesi 4 yerine 0,25 verir. Toplama ve çarpma içeren sınama ifadeleri bu hatayı yakalamaz; çıkarma ya da bölme içeren bir ifadeyle sınamak gerekir.

8.5 Ara ekten soneke çevirme

Çözümün birinci aşaması, ara ek ifadeyi sonek gösterime çevirmektir. Elle çevirmede gözlendiği gibi işlenenlerin sırası değişmez; değişen yalnız işleçlerin yeridir. Algoritma bu gözlemi doğrudan kullanır: işlenenler geldikleri anda çıktıya yazılır, işleçler ise bir yığında sıralarını bekler.

8.5.1 Algoritma ve `isp` / `icp`

Bir işlecin ne zaman çıktıya yazılacağını öncelik belirler. `a + b * c`'de `+` yığına girer ve bekler. `*` geldiğinde `+` hâlâ yığındadır. `*` daha öncelikli olduğu için önce uygulanmalıdır, yani çıktıya `+`'dan önce yazılmalıdır. Bu yüzden `*`, `+`'nın üstüne biner. Tersine, `a * b + c`'de `+` geldiğinde yığında `*` vardır. `*` daha önceliklidir ve artık uygulanabilir; `+` yığına girmeden önce `*` çıktıya alınır.

Algoritmanın kuralları şunlardır. İfade soldan sağa okunur.

1. **İşlenen** doğrudan çıktıya yazılır.
2. **İşleç** gelince, yığının üstündeki işlecin önceliği gelen işlecin önceliğinden büyük ya da ona eşit olduğu sürece üstteki işleç çıktıya alınır. Sonra gelen işleç yığına konur. Eşitlikte de çıktıya almak, soldan sağa birleşmeyi sağlar.
3. **Sol parantez** `(` her zaman yığına girer. Yığındayken en düşük öncelikli işleç gibi davranır; üstüne her işleç binebilir.
4. **Sağ parantez** `)` gelince `(` görülene kadar yığındaki işleçler çıktıya alınır. Sonra `(` yığından atılır. Parantezler çıktıya hiç yazılmaz.
5. **İfade bitince** yığında kalan işleçlerin hepsi çıktıya alınır.

Sol parantezin iki farklı davranışı tek bir öncelik değeriyle ifade edilemez. Gelirken hiçbir işleci yığından çıkarmamalıdır; bu, en yüksek öncelik demektir. Yığındayken ise üstüne her işleç binmelidir; bu da en düşük öncelik demektir. Çözüm, her işlece iki öncelik vermektir.

> **Tanım 8.6**
>
> **Yığın içi öncelik** (*in-stack precedence*, `isp`), bir işlecin yığında beklerken taşıdığı önceliktir. **Gelen öncelik** (*incoming precedence*, `icp`), bir işlecin girdiden okunduğu andaki önceliğidir. Gelen işleç `x` için, yığının üstündeki `t` işleci $isp(t) \geq icp(x)$ olduğu sürece çıktıya alınır.

Belirteç	`isp` (yığında)	`icp` (gelen)
`(`	0	20
`* / %`	13	13
`+ -`	12	12
`#` (dip işareti)	−1	—

Tablo 8.9: Öncelik değerleri. Büyük sayı yüksek önceliktir.

Tablodaki sayıların kendisi önemli değildir; yalnız aralarındaki sıra önemlidir. Horowitz ve arkadaşları C'nin tam öncelik tablosuna uymak için 12 ve 13 gibi değerler kullanır. `(`'ın gelen önceliği 20'dir ve hiçbir `isp` değeri 20'ye ulaşmaz; bu yüzden `(` geldiğinde yığından hiçbir şey çıkmaz. Yığın içi önceliği 0'dır ve her işlecin gelen önceliği 0'dan büyüktür; bu yüzden `(`'ın üstüne her işleç biner. Aynı zamanda hiçbir işleç `(`'ı yığından çıkaramaz, çünkü $0 \geq 12$ hiçbir zaman doğru değildir. `(`'ı yalnız `)` kaldırır.

Yığının dibindeki `#` bir bekçidir. Yığın içi önceliği −1'dir; hiçbir işlecin gelen önceliğinden büyük ya da ona eşit değildir. Bu yüzden "üstteki işleci çıktıya al" döngüsü `#`'a ulaşınca kendiliğinden durur. Böylece döngü koşulunda yığının boş olup olmadığını ayrıca denetlemeye gerek kalmaz. Labirentin çevresindeki duvar çerçevesiyle aynı fikirdir.

Doğrulama programında öncelikler şu iki küçük fonksiyonla tanımlandı:

```
int isp(char c) {           /* yığındaki öncelik */
  switch (c) {
    case '(': return 0;
    case '*': case '/': case '%': return 13;
    case '+': case '-': return 12;
    default: return -1;     /* '#': yığının dibi */
  }
}
int icp(char c) {           /* gelen öncelik */
  if (c == '(') return 20;
  return isp(c);
}
```

8.5.2 `postfix` yordamı

```
void postfix(const char *e, char *cikti) {
  char y[MAX] = {'#'};  int top = 0, k = 0;
  for (; *e; e++) {
    if (isalnum(*e)) cikti[k++] = *e;
    else if (*e == ')') {
      while (y[top] != '(')
        cikti[k++] = y[top--];
      top--;                /* '(' at */
    } else {
      while (isp(y[top]) >= icp(*e))
        cikti[k++] = y[top--];
      y[++top] = *e;
    }
  }
  while (top > 0) cikti[k++] = y[top--];
  cikti[k] = '\0';
}
```

- `y` işleç yığınıdır. `char y[MAX] = {'#'}` bildirimi ilk hücreye `#` yazar (geri kalan hücreler sıfırlanır). `top = 0` dip işaretini gösterir; yani yığın mantıksal olarak boştur ama dipte bekçi vardır. `k`, çıktı dizgisinde bir sonraki yazılacak konumdur.
- `for (; *e; e++)` döngüsü işaretçiyi ifadenin sonuna, `'\0'`'a kadar ilerletir. `e` parametresi bir kopya olduğu için onu ilerletmek çağıranın dizgisini etkilemez.
- `isalnum(*e)` harf ya da rakam sınar. İşlenen doğrudan `cikti`'ya yazılır.
- `)` gelirse iç döngü yığının üstü `(` olana kadar işleçleri çıktıya alır. Sonra `top--` ile `(` atılır; çıktıya yazılmaz.
- Öteki her karakter (dört işleç, `%` ve `(`) için "büyük ya da eşit oldukça al" döngüsü çalışır. Ardından gelen karakter yığına konur. `(` için döngü hiç dönmez, çünkü *icp('(') = 20*'dir.
- Döngüden sonra `top > 0` olduğu sürece, yani dipteki `#`'a gelinceye kadar kalan işleçler çıktıya alınır. En son çıktı dizgisi `'\0'` ile kapatılır.

Fonksiyon sonucu çağıranın verdiği `cikti` dizisine yazar. Çağıran bu diziyi yeterince büyük ayırmalıdır. Sonek ifade parantez içermediği için hiçbir zaman ara ek ifadeden uzun olmaz; ifade uzunluğu kadar yer artı `'\0'` yeter.

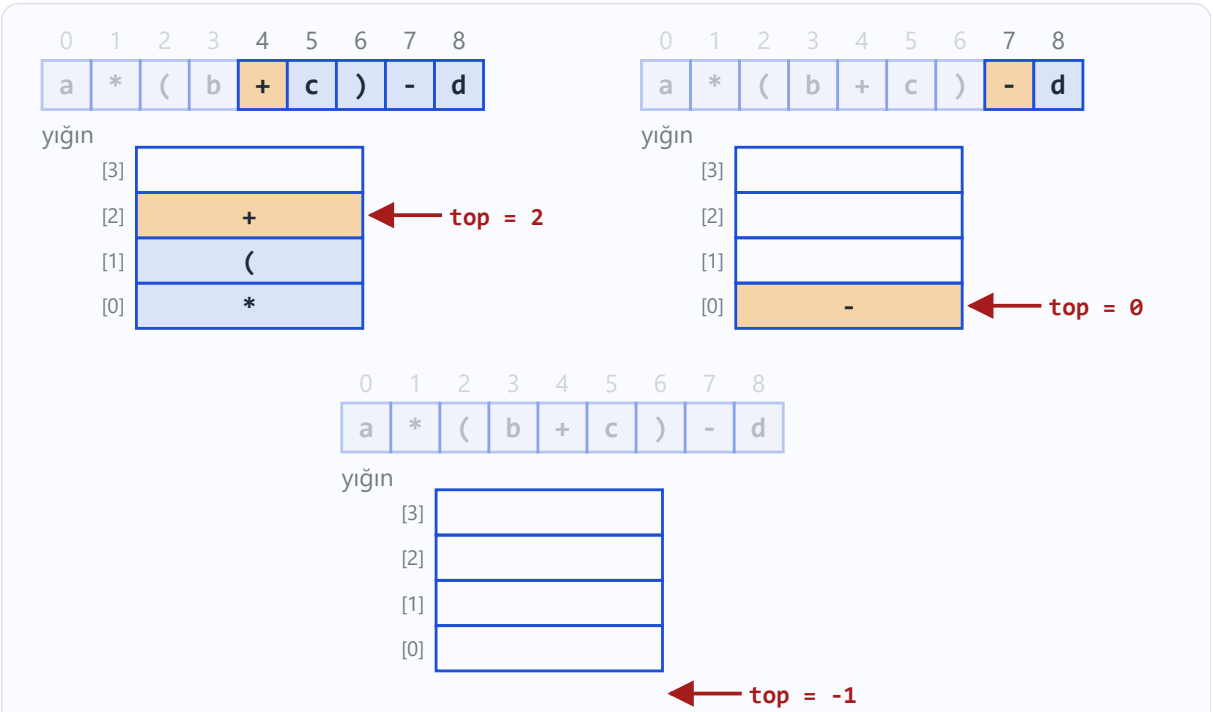
8.5.3 Kod izleme: `a*(b+c)-d`

Yordamı `a*(b+c)-d` üzerinde çalıştıralım. Yığının dibinde her zaman `#` bulunur; tabloda ve şekillerde gösterilmez.

Belirteç	Karar	Yığın	`top`	Çıktı
`a`	işlenen → çıktıya	(boş)	0	a
`*`	*isp(#) = −1 < icp(*) = 13* → yığına	*	1	a
`(`	*isp(*) = 13 < icp(() = 20* → yığına	* (	2	a
`b`	işlenen → çıktıya	* (	2	ab
`+`	*isp(() = 0 < icp(+) = 12* → yığına	* (+	3	ab

Belirteç	Karar	Yığın	`top`	Çıktı
`c`	işlenen → çıktıya	* (+	3	abc
`)`	`(` 'a kadar al: `+` çıktıya, `(` atılır	*	1	abc+
`-`	*isp(*) = 13 ≥ icp(−) = 12* → `*` çıktıya; *isp(#) < 12* → `-` yığına	−	1	abc+*
`d`	işlenen → çıktıya	−	1	abc+*d
son	kalan `-` çıktıya	(boş)	0	abc+*d−

Tablo 8.10: `postfix("a*(b+c)-d")` izlemesi. Yığın sütununda dipteki `#` gösterilmez; `top` kodun değişkenidir.

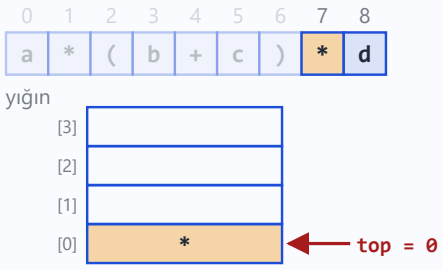


Şekil 8.8: `a*(b+c)-d` çevirisinin üç anı. İlk çizimde `+`, yığındaki `(`'ın üstüne binmiştir. İkincide `-` gelmiş, önceliği eşit ya da yüksek olan `*`'ı çıktıya almış ve kendisi yığına girmiştir. Üçüncüde girdi bitmiş, sonuç `abc+*d-`'dir.

İzlemenin kilit anı `)`'dır. Parantez içindeki `b+c` işlemi kapanış parantezi gelene kadar tamamlanamaz; `+` bu yüzden yığında bekler. `)` geldiğinde `+` çıktıya alınır. Böylece parantez içi, dışarıdaki `*`'dan önce uygulanacak biçimde çıktıya geçer. Sonuç `abc+*d-`'dir. Bu ifade "önce `b+c`, sonra `a` ile çarp, sonra `d`'yi çıkar" diye okunur.

8.5.4 Eşit öncelik: `a*(b+c)*d`

İkinci örnek yalnız bir karakterle ayrılır: `-` yerine `*`. İzleme `)`'a kadar aynıdır. Fark, ikinci `*` geldiğinde ortaya çıkar. Yığının üstünde ilk `*` vardır ve *isp(*) = 13 ≥ icp(*) = 13*'tür. Eşitlik de çıktıya almak için yeterlidir. İlk `*` çıktıya alınır, ikinci `*` yığına girer. Sonuç `abc+*d*`'dir.



Şekil 8.9: `a*(b+c)*d` çevirisinde ikinci `*` geldiği an: eşit öncelikteki ilk `*` çıktıya alınmış, ikinci `*` yığına girmiştir.

Eşitlikte çıktıya almanın anlamı, **soldan sağa birleşmedir**. `a*(b+c)*d` ifadesi `(a*(b+c))*d` demektir: önce soldaki çarpma yapılır. Soldaki çarpma çıktıya önce yazılırsa sonek ifadede de önce uygulanır. Çarpma birleşmeli olduğu için burada sonuç değişmez, ama çıkarmada değişir. `a-b-c` doğru kuralla `ab-c-` 'ye, yani `(a-b)-c` 'ye çevrilir. Eşitlikte almayan bir kod (`>` kullanan) ise `abc--` üretir. Bu `a-(b-c)` demektir. *a = 9*, *b = 5*, *c = 2* için doğru sonuç 2, yanlış kodun sonucu 6'dır. Doğrulama programı iki sürümü de çalıştırdı ve bu iki değeri buldu.

> **Sık yapılan hata**
>
> Karşılaştırmayı `isp(y[top]) > icp(*e)` yazmak. Kod çoğu sınama ifadesinde doğru sonuç verir; yalnız art arda gelen eşit öncelikli işleçlerde yanılır. Bu yüzden hata uzun süre fark edilmeyebilir. `a-b-c` ve `a/b/c` gibi ifadeler mutlaka sınanmalıdır.

8.5.5 İzleme tablosu: `a+b*c`

Sunumdaki son örnek `a+b*c` 'dir. Tablo 8.11 Horowitz ve arkadaşlarının kitabındaki biçimi izler: yığın hücreleri `[0]` ve `[1]` ayrı sütunlardır ve dipteki işaret sayılmaz. Bu yüzden tablodaki `top` değerleri koddaki değişkenden bir eksiktir: kodda `y[0]` hücresinde `#` durur, `+` `y[1]` 'e yazılır.

Belirteç	[0]	[1]	`top`	Çıktı
`a`			−1	a
`+`	+		0	a
`b`	+		0	ab
`*`	+	*	1	ab
`c`	+	*	1	abc
son			−1	abc*+

Tablo 8.11: `a+b*c` → `abc*+` izleme tablosu (dip işareti sayılmadan).

`*` geldiğinde yığının üstünde `+` vardır. *isp(+) = 12 < icp(*) = 13* olduğu için `+` çıktıya alınmaz; `*` onun üstüne biner. Girdi bitince yığın üstten boşaltılır: önce `*`, sonra `+` çıkar. Daha öncelikli işleç önce uygulanır.

Bu tabloda sık sorulan bir soru şudur: “Son işlenen `c` çıktıya basılmıyor mu?” Basılır. `postfix` yordamı işlenenleri geldikleri anda çıktıya yazar; `c` de son satırdan önce çıktıya geçmiştir. Sonda yalnız yığında bekleyen işleçler boşaltılır.

8.5.6 Maliyet

İfadenin uzunluğu *n* olsun. Dış döngü her karakteri bir kez işler. İç döngüler ilk bakışta maliyeti karesel yapacakmış gibi görünür: tek bir `)` ya da tek bir işleç, yığındaki birçok öğeyi çıktıya alabilir. Ama her işleç yığına en çok bir kez girer ve en çok bir kez çıkar. Bütün iç döngülerin toplam tur sayısı bu yüzden yığına giren işleç sayısını geçemez, yani *n*'den küçüktür. Bu tür sayıma **toplu analiz** (*aggregate analysis*) denir: tek bir adımın değil, bütün adımların toplam maliyeti hesaplanır. Toplam süre *Θ(n)*'dir. Yığın ve çıktı dizisi *O(n)* ek bellek kullanır.

Çeviri ve hesaplama birlikte de doğrusaldır: önce $\Theta(n)$'de sonek ifade üretilir, sonra $\Theta(n)$'de hesaplanır. Derleyiciler bir ifadeyi ayrıştırırken genellikle bu iki adımı birleştirir ya da sonek ifade yerine bir ifade ağacı kurar, ama temel fikir aynıdır.

8.6 Önek gösterim

Önek gösterim sonek gösterimin ayna görüntüsüdür. İşleç işlenenlerinden önce gelir. Çevirme ve hesaplama yöntemleri de soneke benzer; fark, ifadenin hangi yönden okunduğundadır.

8.6.1 Ara ekten öneke

Ara ek ifadeyi öneke çevirmenin kolay bir yolu, zaten elde olan `postfix` algoritmasını kullanmaktır. Yöntem dört adımdan oluşur. Tablo 8.12 adımları `(a+b)*c-d/a` ifadesine uygular.

Adım	İfade
Başlangıç	`(a+b)*c-d/a`
1. Ters çevir	`a/d-c*)b+a(`
2. `(` ile `)`'yı değiştir	`a/d-c*(b+a)`
3. Soneke çevir (eşitte alma)	`ad/cba+*-`
4. Yeniden ters çevir	`-*+abc/da`

Tablo 8.12: Ters çevirme yoluyla ara ekten öneke.

Yöntemin mantığı şudur. İfade ters çevrilince her işlecin solundaki işlenen sağa, sağındaki sola geçer. Ters çevrilmiş ifadenin soneki bulunduğunda işleç işlenenlerinin sonuna yazılır. Sonuç yeniden ters çevrilince işleç işlenenlerinin önüne geçer; işlenenler de asıl sıralarına döner. Ters çevirme parantezleri de çevirir: `(a+b)` , `)b+a(` olur. Bu yüzden ikinci adımda parantezler yer değiştirir.

Üçüncü adımda önemli bir değişiklik vardır: eşit öncelikte yığından **alınmaz**; karşılaştırma `>` ile yapılır. Ters çevrilmiş ifadede birleşme yönü de ters döner. `a-b-c` , yani `(a-b)-c` , ters çevrilince `c-b-a` olur. Bu ifadenin doğru öneke dönüşmesi için soldaki `-` 'nin yığında kalması, sağdaki `-` 'nin onun üstüne binmesi gerekir. Doğrulama programı bu kuralla `a-b-c` için `--abc` 'yi, yani `-(-(a, b), c)` 'yi buldu; bu doğru önek karşılığıdır.

Yöntemin C kodu şöyle yazılabilir (aynı `isp` / `icp` fonksiyonlarını kullanır):

```
/* infix -> prefix: ters çevir, parantezleri değiştir,
   eşitlikte almadan (>) postfix'e çevir, yeniden ters çevir */
void prefix(const char *e, char *cikti) {
  char t[MAX], y[MAX] = {'#'};
  int n = strlen(e), top = 0, k = 0;
  for (int i = 0; i < n; i++) {        /* ters çevir */
    char c = e[n - 1 - i];
    t[i] = c == '(' ? ')' : c == ')' ? '(' : c;
  }
  t[n] = '\0';
  for (char *p = t; *p; p++) {
    if (isalnum(*p)) cikti[k++] = *p;
    else if (*p == ')') {
      while (y[top] != '(') cikti[k++] = y[top--];
      top--;
    } else {
      while (isp(y[top]) > icp(*p))    /* > : eşitte alma */
        cikti[k++] = y[top--];
      y[++top] = *p;
    }
  }
  while (top > 0) cikti[k++] = y[top--];
  cikti[k] = '\0';
  for (int i = 0, j = k - 1; i < j; i++, j--) {   /* yeniden ters çevir */
    char g = cikti[i]; cikti[i] = cikti[j]; cikti[j] = g;
  }
}
```

Sonuç Tablo 8.4'ün üçüncü satırıyla aynıdır. Fonksiyon dört ifadenin hepsinde tablodaki önek karşılığını üretti. Ters çevirmeler, yığın işlemleri ve çeviri ifadenin her karakterine sabit sayıda işlem uygular; toplam süre $\Theta(n)$'dir.

Doğrudan bir yol da vardır. İfade sağdan sola taranır. İşlenenler çıktının başına eklenir; işleçler aynı yığın kuralıyla yönetilir, yine eşitlikte alınmadan. Parantezlerin rolü de yer değiştirir: sağdan sola okurken önce `)` görülür ve yığına girer, `(` görülünce yığın boşaltılır. İki yol da $O(n)$'dir.

8.6.2 Önek ifadeyi hesaplama

Önek ifade sağdan sola okunarak, sonek hesaplamayla aynı yığın tekniğiyle hesaplanır:

```
float prefix_eval(const char *e) {
  float y[MAX], op1, op2;  int top = -1;
  for (int i = strlen(e) - 1; i >= 0; i--) {
    if (isdigit(e[i])) y[++top] = e[i] - '0';
    else {
      op1 = y[top--];  op2 = y[top--];
      y[++top] = uygula(e[i], op1, op2);
    }
  }
  return y[top];
}
```

`i` son karakterden başlar ve 0'a kadar azalır. Rakam yığına konur. İşleç gelince iki değer alınır ve `uygula` fonksiyonu işlemi yapar. Bu fonksiyon dört işleci bir `switch` ile ayırır:

```
float uygula(char islec, float op1, float op2) {
  switch (islec) {
    case '+': return op1 + op2;
    case '-': return op1 - op2;
    case '*': return op1 * op2;
    case '/': return op1 / op2;
    default:  return 0;
  }
}
```

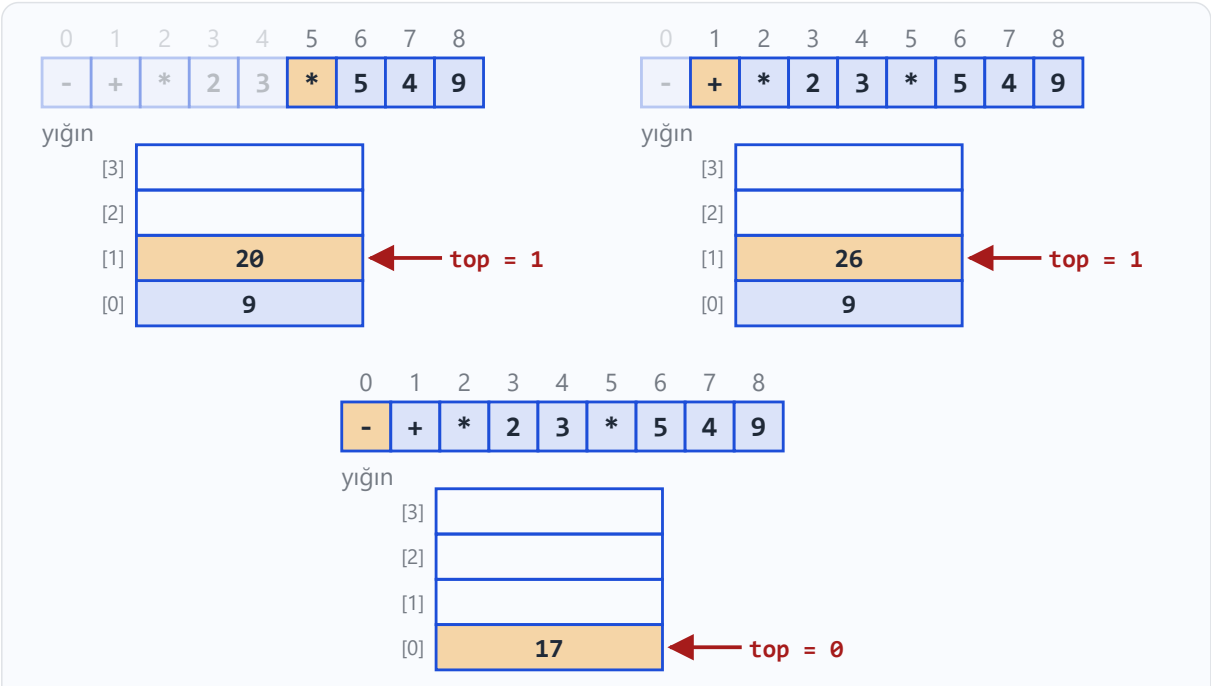
Kodun can alıcı farkı alınma sırasıdır. Önek ifadede bir işlecin hemen sağında önce sol işlenen, sonra sağ işlenen yazılır. Sağdan sola okunduğunda sağ işlenen önce görülür ve yığına önce konur. Sol işlenen sonra konur ve üstte durur. Bu yüzden **ilk alınan değer sol işlenendir**: `op1 = y[top--]; op2 = y[top--];` . Sonek hesaplamada ise ilk alınan sağ işlenendi.

`strlen(e) - 1` ifadesi bir tip ayrıntısı içerir. `strlen` işaretsiz `size_t` döndürür. Boş dizgide `0 - 1` işaretsiz aritmetikte çok büyük bir sayıya dönüşür. Kod bu sonucu `int i` 'ye atadığı için uygulamada −1 elde edilir ve döngü hiç dönmez. Döngü değişkeni `size_t` yapılsaydı `i >= 0` koşulu her zaman doğru olurdu ve döngü sonsuza giderdi.

Sunumdaki örnek `-+*23*549` 'dur. Bu, `2*3+5*4-9` ifadesinin önek karşılığıdır; değeri sonek örnekteki gibi 17 olmalıdır.

`i`	`e[i]`	İşlem	Yığın
8	`9`	yığına	9
7	`4`	yığına	9 · 4
6	`5`	yığına	9 · 4 · 5
5	`*`	`op1` = 5 (sol), `op2` = 4 → 20	9 · 20
4	`3`	yığına	9 · 20 · 3
3	`2`	yığına	9 · 20 · 3 · 2
2	`*`	`op1` = 2, `op2` = 3 → 6	9 · 20 · 6
1	`+`	`op1` = 6, `op2` = 20 → 26	9 · 26
0	`-`	`op1` = 26, `op2` = 9 → 26 − 9 = 17	17

Tablo 8.13: `prefix_eval("-+*23*549")` izlemesi. İfade sağdan sola okunur.

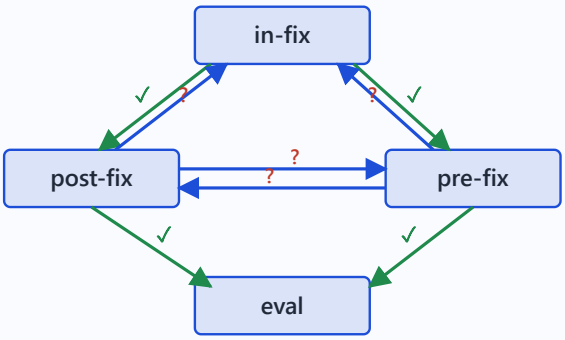


Şekil 8.10: Önek hesaplamanın üç anı. İlk çizimde ilk `*` : ilk alınan 5 sol işlenendir, sonuç 20. İkincide `+` : 6 + 20 = 26. Üçüncüde `-` : 26 − 9 = 17.

Son adımda alınma sırası sonucu belirler. Yığında 9 altta, 26 üsttedir. İlk alınan 26 sol işlenendir ve sonuç *26 − 9 = 17* olur. Sonek kuralı burada uygulansaydı *9 − 26 = −17* bulunurdu. Doğrulama programı `-62` için 4'ü, `*-623` için *(6 − 2) · 3 = 12*'yi buldu.

8.6.3 Öteki çeviriler

Üç gösterim ve hesaplama dört köşeli bir şema oluşturur. Şekil 8.11 bu şemayı verir. Yeşil oklar bu notta kodu verilen yolları, kırmızı soru işaretleri ise tasarlanacak çevirileri gösterir.



Şekil 8.11: Gösterimler arası çeviriler. ✓: bu notta yapıldı (ara ekten soneke, ara ekten öneke ters çevirme yoluyla, sonek ve önek hesaplama). ?: tasarlanacak çeviriler.

Kalan çevirilerin hepsi aynı yığın düşüncesiyle tasarlanabilir:

- **Sonekten ara eke**: elle yapılan çevirinin C ile yazılması. Yığında dizgiler durur; çözüm alıştırmalarda verilir.
- **Sonekten öneke**: sonekten ara eke çevirmenin aynısı, yalnız birleştirme `op a b` biçiminde yapılır. **Önekten soneke**: ifade sağdan sola okunur, birleştirme `a b op` biçimindedir.
- **Ara ekten öneke doğrudan**: ifade sağdan sola taranır (önceki kesimde anlatıldı).
- **Önekten ara eke**: sağdan sola okunur, ilk alınan sol işlenendir, birleştirme `(a op b)` biçimindedir.
- **Ara ek ifadeyi doğrudan hesaplamak**: iki yığın kullanılır, biri işlenenler biri işleçler için. Çeviri algoritması bir işleci çıktıya yazacağı anda, o işleç işlenen yığınından iki değer alıp sonucu geri koyar. Böylece sonek ifade hiç üretilmeden doğrudan değer hesaplanır.

8.7 Parantez denetimi

Ara ekten soneke çeviri, ifadedeki parantezlerin doğru kurulduğunu varsayar. Bu varsayım ayrı bir algoritmayla denetlenebilir. **Parantez denetimi** (*balanced parentheses*), açılan her parantezin doğru türde ve doğru sırada kapandığını sınar. Problem yığının en temiz uygulamalarından biridir.

> **Tanım 8.7**
>
> Bir dizgideki parantezler şu üç koşul sağlanırsa **dengelidir**: her kapanan parantezin kendisinden önce açılmış ve henüz kapanmamış bir eşi vardır; bu eş, kapanmamış açanların **en son açılanıdır** ve kapananla aynı türdendir (`()` , `[]` , `{}`); dizgi bitince kapanmamış açan kalmaz.

Tanımın ortasındaki “en son açılan” ifadesi yığını işaret eder. `{a*(b+c)-[d]}` dizgisinde `(` , `{` 'dan sonra açılmıştır ve ondan önce kapanmalıdır. Açanlar bir yığına konursa, kapanan geldiğinde eşi her zaman yığının üstündedir.

8.7.1 `dengeli` yordamı

```
bool dengeli(const char *e) {
  char y[MAX];  int top = -1;
  for (; *e; e++) {
    if (*e=='(' || *e=='[' || *e=='{')
      y[++top] = *e;
    else if (*e==')' || *e==']' || *e=='}') {
      if (top < 0 || !eslesir(y[top], *e))
        return false;  /* fazla/yanlış */
      top--;
    }
  }
  return top == -1;       /* açık kalmadı mı */
}
```

- Açan bir parantez (`(` , `[` , `{`) gelirse yığına konur.
- Kapanan bir parantez gelirse iki durum denetlenir. Yığın boşsa (`top < 0`) bu kapananın eşi yoktur: fazla kapanan. Yığın boş değilse üstteki açanın kapananla eşleşip eşleşmediğine bakılır. Eşleşmiyorsa türler karışmıştır: yanlış eşleşme. İki durumda da fonksiyon hemen `false` döndürür. Eşleşirse açan yığından atılır (`top--`).
- Parantez olmayan karakterler atlanır.
- Dizgi bitince yığın boşsa (`top == -1`) her açan kapanmıştır ve sonuç `true` olur. Boş değilse kapanmamış açan vardır.

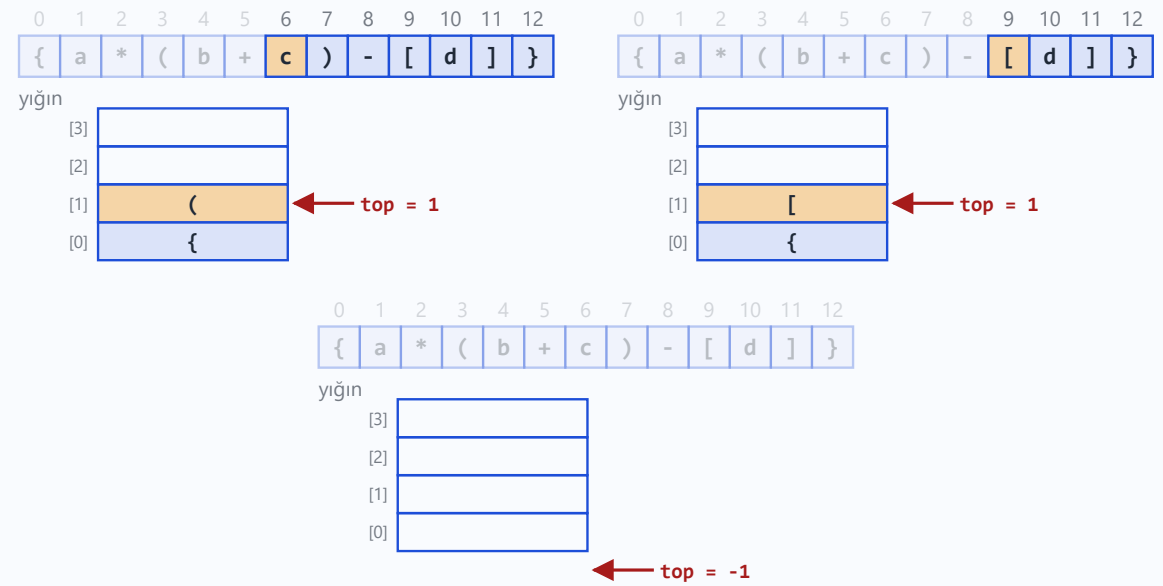
Koşuldaki `top < 0 || !eslesir(y[top], *e)` ifadesi kısa devre değerlendirmeye dayanır. `||` işleci önce sol tarafı değerlendirir; sol taraf doğruysa sağ tarafa hiç bakmaz. Yığın boşken `y[top]` , yani `y[-1]` okunmaz. Koşullar ters sırayla yazılsaydı boş yığında dizi dışına okunurdu. `eslesir` fonksiyonu sunumda verilmemiştir; doğrulama programında şöyle yazıldı:

```
bool eslesir(char acan, char kapanan) {
  return (acan == '(' && kapanan == ')') ||
         (acan == '[' && kapanan == ']') ||
         (acan == '{' && kapanan == '}');
}
```

8.7.2 Kod izleme: `{a*(b+c)-[d]}`

Karakter	İşlem	Yığın	`top`
`{`	açan → yığına	{	0
`a` , `*`	atlanır	{	0
`(`	açan → yığına	{ (	1
`b` , `+` , `c`	atlanır	{ (	1
`)`	üstteki `(` ile eşleşti → çıkar	{	0
`-`	atlanır	{	0
`[`	açan → yığına	{ [	1
`d`	atlanır	{ [	1
`]`	üstteki `[` ile eşleşti → çıkar	{	0
`}`	üstteki `{` ile eşleşti → çıkar	(boş)	−1
son	`top == -1` → `true`	(boş)	−1

Tablo 8.14: `dengeli("{a*(b+c)-[d]}")` izlemesi. Parantez olmayan karakterler tek satırda toplanmıştır.



Şekil 8.12: Parantez denetiminin üç anı. İlk çizimde `{` ve `(` yığındadır. İkincide `(` kapanmış, `[` açılmıştır. Üçüncüde girdi bitmiş ve yığın boştur: dizgi dengelidir.

Yığın en çok iki açan tuttu. Bu, iç içe geçmenin derinliğidir: `(` ve `[` her biri `{`'nın içinde ama birbirinden bağımsızdır.

8.7.3 Üç hata türü

Dengesiz bir dizgi üç farklı biçimde ortaya çıkar. Tablo 8.15 her türden bir örnek verir; üç örnek de doğrulama programında `false` sonucu verdi.

Girdi	Ne olur?	Hata
`[(a+b]*c)`	`]` gelince üstte `(` var	Yanlış eşleşme
`(a+b))`	İkinci `)` geldiğinde yığın boş	Fazla kapanan
`((a+b)`	Girdi bitince yığında `(` kaldı	Kapanmamış açan

Tablo 8.15: Parantez denetiminde üç hata türü.

Tek türde parantez için yığına gerek yoktur; bir sayaç yeter. Açanda sayaç bir artar, kapananda bir azalır. Sayaç hiçbir zaman sıfırın altına inmez ve sonda sıfır olursa dizgi dengelidir. Birden çok parantez türü olduğunda sayaç yetmez. `[(a+b]*c)` dizgisinde açan ve kapanan sayıları eşittir, ama türler çaprazlanmıştır. Bunu yakalamak için "en son açılan hangisiydi?" sorusunu yanıtlamak gerekir. Bu bilgiyi yığın tutar.

Bu algoritma pek çok yazılımın içinde çalışır. Derleyiciler kaynak koddaki parantez ve küme parantezlerini böyle denetler. HTML ve XML denetleyicileri açılan ve kapanan etiketleri aynı yöntemle eşler; orada yığına parantez değil etiket adı konur. Metin düzenleyicilerin imleç bir parantezin üzerine gelince eşini vurgulaması da bu algoritmanın bir uygulamasıdır.

Maliyet: dizginin uzunluğu n ise her karakter bir kez incelenir ve her işlem $O(1)$'dir; süre $O(n)$'dir. En kötü durumda bütün karakterler açandır ve hepsi yığına girer; ek bellek $O(n)$'dir. Hata erken bulunursa fonksiyon dizginin sonuna gitmeden döner.

8.8 Yığın ve kuyruğun başka uygulamaları

Labirent, ifade değerlendirme ve parantez denetimi ortak bir düşünceyi paylaşır: "en son açılan ilk kapanır". Bu düşünce görülen her yerde yığın kullanılabilir. "Sırayla", yani "ilk gelen ilk işlenir" görülen yerde ise kuyruk kullanılır. Bu kesim bu ilkenin birkaç başka uygulamasını kısaca tanıtır.

8.8.1 Geri al ve yinele

Metin düzenleyicilerdeki **geri al** (*undo*) ve **yinele** (*redo*) komutları iki yığınla çalışır. Kullanıcı bir işlem yaptığında işlem "geri al" yığınına konur ve "yinele" yığını boşaltılır. Geri al komutu "geri al" yığınının üstündeki işlemi alır, tersini uygular ve onu "yinele" yığınına koyar. Yinele komutu "yinele" yığınının üstündeki işlemi alır, yeniden uygular ve "geri al" yığınına koyar.

Örneğin kullanıcı sırayla A, B ve C işlemlerini yapsın. "Geri al" yığını A · B · C olur. İki kez geri alınırsa önce C, sonra B geri alınır. "Geri al" yığınında A, "yinele" yığınında C · B kalır; B üsttedir. Bir kez yinelenirse B yeniden uygulanır. Bu anda kullanıcı yeni bir D işlemi yaparsa "yinele" yığınındaki C silinir. Yeni bir dal başlamıştır ve eski geleceğe dönülemez. Web tarayıcılarının geri ve ileri düğmeleri de aynı iki yığınla çalışır.

8.8.2 Özyinelemesiz ağaç gezinmesi

İkili ağacın düğümlerini sırayla ziyaret etmeye **gezinme** denir. Gezinme doğal olarak özyinelemeyle yazılır: kök ortada gezinmede önce sol alt ağaç gezilir, sonra kök ziyaret edilir, sonra sağ alt ağaç gezilir. Özyineleme, bekleyen işleri çağrı yığınında saklar. Aynı iş açık bir yığınla da yapılabilir; böylece özyinelemeli çağrıların yükü ortadan kalkar ve yığının boyu programcının denetimine girer.

Kodlar için şu düğüm tipi kullanılır: her düğüm bir karakter ve iki alt ağaç işaretçisi taşır.

```c
typedef struct dugum *dugum_gost;
typedef struct dugum {
  char veri;
  dugum_gost sol, sag;
} dugum;

void kok_ortada(dugum_gost p) {          /* inorder, özyinelemesiz */
  dugum_gost y[MAX];  int top = -1;
  for (;;) {
    for (; p; p = p->sol) y[++top] = p;  /* sola inerken it */
    if (top < 0) break;                  /* yığın boş: bitti */
    p = y[top--];
    printf("%c", p->veri);
    p = p->sag;                          /* sağ alt ağaca geç */
  }
}

void kok_basta(dugum_gost p) {           /* preorder, özyinelemesiz */
  dugum_gost y[MAX];  int top = -1;
  if (p == NULL) return;
  y[++top] = p;
  while (top >= 0) {
    p = y[top--];
    printf("%c", p->veri);
    if (p->sag) y[++top] = p->sag;       /* önce sağ itilir, */
    if (p->sol) y[++top] = p->sol;       /* sol üstte kalır */
  }
}

void kok_sonda(dugum_gost p) {           /* postorder, iki yığınla */
  dugum_gost y1[MAX], y2[MAX];  int t1 = -1, t2 = -1;
  if (p == NULL) return;
  y1[++t1] = p;
  while (t1 >= 0) {
    p = y1[t1--];
    y2[++t2] = p;                        /* kök, sağ, sol sırası */
    if (p->sol) y1[++t1] = p->sol;
    if (p->sag) y1[++t1] = p->sag;
  }
  while (t2 >= 0) printf("%c", y2[t2--]->veri);   /* ters: sol, sağ, kök */
}
```

`kok_ortada` fonksiyonu kök ortada (*inorder*) gezinmeyi yapar. İç `for` döngüsü `p`'den başlayıp sol bağları izleyerek aşağı iner ve geçtiği her düğümü yığına koyar. Sol uca varılınca yığının üstündeki düğüm alınır, yazdırılır ve onun sağ alt ağacına geçilir. Yığın boşalınca ve `p` `NULL` olunca gezinme biter. Yığında her an kökten şimdiki düğüme giden yol üzerindeki "sağına henüz geçilmemiş" düğümler durur. Bu yüzden yığının boyu ağacın yüksekliğini aşmaz.

`kok_basta` (*preorder*) daha basittir. Kök yığına konur. Her turda üstteki düğüm alınır ve yazdırılır; sonra önce sağ, sonra sol çocuğu yığına konur. Sol çocuk üstte kaldığı için önce o işlenir. `kok_sonda` (*postorder*) iki yığınla yazılmıştır. Birinci yığın düğümleri "kök, sağ, sol" sırasında ikinci yığına aktarır. İkinci yığın ters sırayla boşaltılınca "sol, sağ, kök" sırası, yani kök sonda gezinme elde edilir.

Doğrulama programı `(a+b)*c-d/a` ifadesinin ağacını kurdu: kök `-`, sol alt ağaç `(a+b)*c`, sağ alt ağaç `d/a`. Üç gezinme sırasıyla `a+b*c-d/a` (parantezsiz ara ek), `-*+abc/da` (önek) ve `ab+c*da/-` (sonek) çıktılarını verdi. Bir ifade ağacının üç gezinmesi, ifadenin üç gösterimidir. Ağaç kurmak ve bırakmak için yapılan 9 `malloc` çağrısının hepsine bir `free` karşılık geldi.

Maliyet: n düğümlü ağaçta her düğüm yığına bir kez girer ve bir kez çıkar; süre $O(n)$'dir. Kök ortada ve kök önde gezinmede yığın en çok ağacın yüksekliği h kadar düğüm tutar (kök önde için tam sınır $h + 1$ mertebesindedir); ek bellek $O(h)$'dir. Dengeli bir ağaçta $h = O(\log n)$, tek kola dönmüş bir ağaçta $h = n - 1$'dir. İki yığınlı kök sonda gezinmede ikinci yığın bütün düğümleri tuttuğu için ek bellek $O(n)$'dir. Ağacı düzey düzey işleyen **düzey sıralı gezinme** (*level-order traversal*) ise kuyrukla yapılır: kök kuyruğa konur, her adımda önden bir düğüm alınır ve çocukları arkaya eklenir.

8.8.3 Sonraki büyük öğe

Bir dizinin her öğesi için sağındaki ilk büyük öğe bulunmak istensin. İki iç içe döngü bunu $O(n^2)$'de yapar. Yığın aynı işi tek geçişte yapar:

```
/* her öğenin sağındaki ilk büyük öğe; yoksa -1 */
void sonraki_buyuk(const int a[], int n, int sonuc[]) {
  int y[MAX], top = -1;                 /* yığında indisler */
  for (int i = 0; i < n; i++) {
    while (top >= 0 && a[y[top]] < a[i])
      sonuc[y[top--]] = a[i];           /* a[i], bekleyenin cevabı */
    y[++top] = i;
  }
  while (top >= 0) sonuc[y[top--]] = -1; /* sağında büyük yok */
}
```

Yığında cevabını henüz bulamamış öğelerin indisleri durur. Bu öğelerin değerleri alttan üste azalan sıradadır; bu yüzden yapıya **azalan yığın** (*monotonic stack*) denir. Yeni bir öğe `a[i]` gelince, üstteki bekleyen öğe ondan küçük olduğu sürece `a[i]` o öğenin cevabıdır; öğe yığından alınır. Sonra `i` yığına konur. Dizi bitince yığında kalan öğelerin sağında büyük öğe yoktur; cevapları −1'dir.

Doğrulama programı `{4, 5, 2, 25, 7, 3, 8}` dizisi için `5 25 25 -1 8 8 -1` sonucunu verdi. Örneğin 25 geldiğinde yığında 5 ve 2'nin indisleri vardır; ikisi de 25'ten küçüktür ve ikisinin cevabı da 25 olur. Her indis yığına bir kez girip bir kez çıktığı için toplam süre $O(n)$'dir; ek bellek en kötü durumda (azalan dizi) $O(n)$'dir.

8.8.4 Kayan pencere en büyüğü

n öğeli bir dizide k genişlikli bir pencere soldan sağa kaydırılsın ve her konumda penceredeki en büyük öğe istensin. Her pencereyi ayrı taramak $O(n \cdot k)$ sürer. **Çift uçlu kuyruk** (*deque*), yani iki ucundan da ekleme ve çıkarma yapılabilen kuyruk, işi $O(n)$'e indirir:

```
/* k genişlikli her pencerenin en büyüğü; d: indis tutan deque */
void kayan_maks(const int a[], int n, int k, int sonuc[]) {
  int d[MAX], on = 0, arka = -1;        /* d[on..arka] */
  for (int i = 0; i < n; i++) {
    if (on <= arka && d[on] <= i - k) on++;          /* pencereden çıktı */
    while (on <= arka && a[d[arka]] <= a[i]) arka--;  /* küçükler atılır */
    d[++arka] = i;
    if (i >= k - 1) sonuc[i - k + 1] = a[d[on]];     /* en büyük önde */
  }
}
```

Çift uçlu kuyrukta indisler durur ve değerleri önden arkaya azalan sıradadır. Yeni öğe gelince önce önden, pencerenin dışına düşmüş indis atılır. Sonra arkadan, yeni öğeden büyük olmayan bütün indisler atılır: bu öğeler yeni öğe pencerede kaldığı sürece hiçbir zaman en büyük olamaz. Yeni indis arkaya eklenir. Penceredeki en büyük öğe her zaman öndedir. `{1, 3, -1, -3, 5, 3, 6, 7}` dizisi ve $k = 3$ için sonuç `3 3 5 5 6 7` 'dir. Her indis bir kez eklenir ve en çok bir kez atılır; süre $O(n)$, ek bellek $O(k)$'dır.

8.8.5 Kuyrukla en kısa yol ve öteki geri izleme problemleri

Labirentte yığının yerine kuyruk kullanılırsa ilk bulunan yol en kısa yoldur. Bu, genişlik öncelikli aramanın temel özelliğidir ve labirentin ötesinde, adımların eşit maliyetli olduğu her ağda geçerlidir. Bir oyunda karakterin en az hamleyle hedefe ulaşması, bir sosyal ağda iki kişi arasındaki en kısa tanışıklık zinciri aynı yöntemle bulunur.

Geri izlemenin öteki klasik örnekleri de yığınla, ya da özyinelemeyle, çözülür. **Hanoi kuleleri** problemi diskleri bir çubuktan ötekine taşırken bekleyen alt problemleri bir yığında tutar. **n-vezir** problemi satranç tahtasına birbirini tehdit etmeyen n vezir yerleştirir. Her satıra bir vezir konur; çıkmaza girilirse son konan vezir kaldırılıp bir sonraki sütun denenir. **Sudoku** çözücüsü boş hücrelere sırayla rakam dener, çelişki çıkınca son denemeyi geri alır. Üçü de labirent yordamının iskeletini taşır: seçim yığına konur, çıkmazda yığından alınır. Bu problemler genellikle özyinelemeyle yazılır ama labirentteki gibi açık bir yığınla da yazılabilir.

8.9 Karmaşıklık özeti

Tablo 8.16 notun algoritmalarının maliyetlerini bir arada verir. Her satırın gerekçesi ilgili kesimde açıklandı.

Algoritma	Yapı	Zaman	Ek bellek
Labirent (`izbul`)	yığın	$O(m \cdot p)$	$O(m \cdot p)$
Labirent en kısa yol	kuyruk	$O(m \cdot p)$	$O(m \cdot p)$
Sonek / önek hesaplama	yığın	$O(n)$	$O(n)$
Ara ekten soneke / öneke	yığın	$O(n)$	$O(n)$
Parantez denetimi	yığın	$O(n)$	$O(n)$
Özyinelemesiz kök ortada	yığın	$O(n)$	$O(h)$
Düzey sıralı gezinme	kuyruk	$O(n)$	$O(n)$
Sonraki büyük öğe	yığın	$O(n)$	$O(n)$
Kayan pencere en büyüğü	çift uçlu kuyruk	$O(n)$	$O(k)$

Tablo 8.16: Karmaşıklık özeti. n: ifade uzunluğu ya da düğüm sayısı; $m \times p$: labirent boyutu; h: ağaç yüksekliği.

Tablodaki doğrusal sürelerin hepsi aynı gerekçeye dayanır: her öğe yapıya en çok bir kez girer ve en çok bir kez çıkar, her giriş ve çıkış $O(1)$'dir. İç içe döngüler bu sınırı bozmaz, çünkü iç döngülerin toplam tur sayısı çıkarılan öğe sayısıyla sınırlıdır. Bu sayım yöntemi yığın ve kuyruk algoritmalarını analiz ederken en sık kullanılan araçtır.

8.10 Sık yapılan hatalar

1. **Sonek hesaplamada `op1` ile `op2` 'yi ters almak.** İlk alınan sağ işlenendir. Ters alınırsa `6 2 -`, 4 yerine −4 olur.
2. **Önek hesaplamada sonek sırasını kullanmak.** Önek ifade sağdan sola okunur ve ilk alınan sol işlenendir.
3. **Ara ekten soneke çevirmede eşit öncelikte yığından almamak.** `>` ile karşılaştıran kod `a-b-c` 'yi `abc--` 'ye, yani `a-(b-c)` 'ye çevirir.
4. **`)` gelince `(` 'ı yığında unutmak ya da çıktıya yazmak.** `(` atılmazsa yığında kalır ve sonraki işleçlerin önünde engel olur; sonda boşaltılırken çıktıya da karışır.
5. **Labirentte `gezildi` işaretini unutmak.** Algoritma iki hücre arasında gidip gelir, sonsuz döngüye girer ya da yığını taşırır.
6. **Labirentin çevresini 1 ile çevirmemek.** Kenardaki hücrelerin komşuları dizi dışına düşer; C bunu denetlemez ve tanımsız davranış oluşur.
7. **Parantez denetiminde yığının boş olup olmadığına bakmadan üstteki öğeyi okumak.** `(a+b))` 'de ikinci `)` boş yığında `y[-1]` 'i okur.
8. **Yığın taşmasını denetlememek.** Sabit boyutlu dizi yığınında `MAX` aşılırsa öğeler dizinin dışına yazılır. Ya boyut girdiye göre güvenli seçilmeli ya da her üste koymada denetim yapılmalıdır.

8.11 Alıştırmalar ve çözümleri

Sunumdaki altı alıştırmanın çözümleri aşağıda ayrıntılı olarak verilir. Kodların hepsi doğrulama programlarında derlenip çalıştırıldı.

8.11.1 Bir ifadeyi soneke ve öneke çevirmek

İfade `a/(b-c)+d*(e-a)*c` 'dir. Önce elle çevirelim. Tam parantezli biçim `((a/(b-c)) + ((d*(e-a))*c))` 'dir. Bölme ve iki çarpma, toplamadan önceliklidir; iki çarpma soldan sağa birleşir.

- **Sonek**: her işleç sağ kapanışa taşınır. `(b-c)` → `bc-` , `a/…` → `abc-/` , `(e-a)` → `ea-` , `d*…` → `dea-*` , `…*c` → `dea-*c*` . Toplam: `abc-/dea-*c*+` .
- **Önek**: her işleç sol açılışa taşınır. Sonuç `+/a-bc**d-eac` 'dir.

Aynı sonuç `postfix` yordamıyla da bulunur. Kısaca: `a` çıktıya; `/` yığına; `(` yığına; `b` , `c` çıktıya, `-` `(` 'ın üstüne; `)` gelince `-` çıktıya. `+` gelince *isp(/) = 13 ≥ 12* olduğu için `/` çıktıya, `+` yığına. `d` çıktıya; `*` , `+` 'nın üstüne biner. `(e-a)` aynı biçimde `ea-` üretir. İkinci `*` gelince eşit öncelikteki ilk `*` çıktıya alınır. `c` 'den sonra girdi biter; yığındaki `*` ve `+` sırayla çıktıya geçer. Doğrulama programının `postfix` ve `prefix` fonksiyonları da `abc-/dea-*c*+` ve `+/a-bc**d-eac` sonuçlarını verdi.

8.11.2 Bir sonek ifadeyi hesaplamak

İfade `6 2 3 + - 3 8 2 / + *` 'dır. Yığın soldan sağa şöyle değişir: 6; 6 · 2; 6 · 2 · 3; `+` → *2 + 3 = 5*, yığın 6 · 5; `-` → *6 − 5 = 1*, yığın 1; 1 · 3; 1 · 3 · 8; 1 · 3 · 8 · 2; `/` → *8 / 2 = 4*, yığın 1 · 3 · 4; `+` → *3 + 4 = 7*, yığın 1 · 7; `*` → *1 · 7 = 7*. Sonuç 7'dir. Yığın en çok dört değer tuttu. `eval("623+-382/+*")` de 7 döndürdü.

8.11.3 Sonekten ara eke çeviren C yordamı

Yığında artık karakter değil, dinamik olarak ayrılmış dizgiler durur. Yöntem sonekten ara eke elle çevirmenin aynısıdır.

```
/* postfix -> tam parantezli infix; yığında dinamik dizgiler durur */
char *postfix_infix(const char *e) {
  char *y[MAX];  int top = -1;
  for (; *e; e++) {
    if (isalnum((unsigned char)*e)) {
      char *s = malloc(2);
      if (s == NULL) return NULL;
      s[0] = *e;  s[1] = '\0';
      y[++top] = s;
    } else {
      char *b = y[top--], *a = y[top--];       /* önce sağ işlenen */
      char *s = malloc(strlen(a) + strlen(b) + 4);
      if (s == NULL) return NULL;
      sprintf(s, "(%s%c%s)", a, *e, b);
      free(a);  free(b);                       /* parçalar artık gereksiz */
      y[++top] = s;
    }
  }
  return y[top];             /* çağıran free etmeli */
}
```

- İşlenen gelince iki baytlık bir dizgi ayrılır: karakterin kendisi ve `'\0'` . Dizginin adresi yığına konur.
- İşleç gelince iki dizgi alınır; ilk alınan `b` sağ işlenendir. Yeni dizginin boyu iki parçanın uzunluğu artı dört bayttır: iki parantez, işleç ve `'\0'` . `sprintf` parçaları `"(a op b)"` biçiminde birleştirir.
- Parçalar yeni dizgiye kopyalandıktan sonra artık gereksizdir ve `free` ile bırakılır. Bu satır unutulursa her işleçte iki blok sızar.
- Dönüş değeri yığındaki son dizgidir. Bu dizgiyi bırakmak çağıranın görevidir.

Doğrulama programı dört ifadeyi çevirdi. `42/2-33*+42*-` için `((((4/2)-2)+(3*3))-(4*2))` , `ab+c*da/-` için `(((a+b)*c)-(d/a))` elde edildi. `malloc` ve `free` çağrılarını sayan sarmalayıcı fonksiyonlarla 44 ayırma ve 44 bırakma sayıldı; sızıntı yoktur. Kodun eksik bir yanı da vardır: `malloc` başarısız olursa fonksiyon `NULL` döndürür ama yığında kalan dizgileri bırakmaz. Sağlam bir sürüm dönmeden önce yığını boşaltmalıdır. Maliyet açısından her birleştirme iki parçayı kopyalar; dizgiler uzadıkça kopyalama da uzar. n uzunluklu bir ifadede toplam kopyalama en kötü durumda $O(n^2)$'dir (örneğin `ab+c+d+…` gibi sola yaslanan bir zincirde her adım bütün önceki dizgiyi kopyalar).

8.11.4 Bulunan yolu labirentte işaretlemek

Çıkış bulunduğunda yol, yığındaki konumlar, son konum ve çıkıştan oluşur. Bu hücrelere `2` yazmak yolu labirentte görünür kılar:

```
/* izbul çıkışı bulduktan sonra çağrılır: yolu labirentte 2 ile işaretler */
void yolu_isaretle(int top, short int satir, short int sutun) {
  for (int i = 0; i <= top; i++)
    labirent[yigit[i].satir][yigit[i].sutun] = 2;
  labirent[satir][sutun] = 2;                 /* son konum */
  labirent[CIKIS_SATIR][CIKIS_SUTUN] = 2;     /* çıkış */
}
```

`izbul`'un sonunda `if (bulundu) yolu_isaretle(top, satir, sutun);` satırıyla çağrılır. Fonksiyon yığını değiştirmez; yalnız okur. Sonuçta labirentin satırları şöyle görünür (1 duvar, 0 gezilmeyen yol, 2 bulunan yol):

```
11111111
12100011
12101111
11211111
12112111
11221211
11111121
11111111
```

Çıkmaz koridordaki `(1,3)`, `(1,4)`, `(1,5)` ve `(2,3)` hücreleri 0 olarak kalır: ziyaret edildiler ama yola girmediler. İşaretleme *O(top)*, yani yol uzunluğunda süre alır.

8.11.5 Labirentte en kısa yolu kuyrukla bulmak

Kuyruğa hücrelerin `(satir, sutun)` çiftleri konur. Her hücre için hangi hücreden ulaşıldığı `once` matrisinde saklanır. Çıkışa ulaşılınca bu bilgi geriye doğru izlenerek yol kurulur.

```
/* labirentte en kısa yol: kuyruk + her hücrenin geldiği hücre */
typedef struct { short int satir, sutun; } hucre;
int en_kisa_yol(void) {
  hucre q[64], once[8][8];               /* 6x6 iç kısım: en çok 36 hücre */
  int bas = 0, son = 0;
  short int gordu[8][8] = {{0}};
  q[son++] = (hucre){1, 1};  gordu[1][1] = 1;
  while (bas < son) {
    hucre h = q[bas++];                  /* kuyruğun önünden al */
    if (h.satir == CIKIS_SATIR && h.sutun == CIKIS_SUTUN) {
      int uzunluk = 1;                   /* çıkıştan geriye izle */
      while (h.satir != 1 || h.sutun != 1) {
        labirent[h.satir][h.sutun] = 2;  /* yolu işaretle */
        h = once[h.satir][h.sutun];
        uzunluk++;
      }
      labirent[1][1] = 2;
      return uzunluk;
    }
    for (int yon = 0; yon < 8; yon++) {
      int s = h.satir + hareket[yon].dikey;
      int t = h.sutun + hareket[yon].yatay;
      if (!labirent[s][t] && !gordu[s][t]) {
        gordu[s][t] = 1;
        once[s][t] = h;
        q[son++] = (hucre){s, t};        /* kuyruğun arkasına ekle */
      }
    }
  }
  return 0;                              /* yol yok */
}
```

- Kuyruk basit bir dizidir: `bas` önden alınacak, `son` arkaya yazılacak konumdur. Her hücre en çok bir kez kuyruğa girdiği için dairesel yapıya gerek yoktur; dizi iç kısmın hücre sayısı kadar yer tutsa yeter.
- Bir hücre kuyruğa girerken işaretlenir (`gordu`), çıkarken değil. Böylece aynı hücre iki farklı komşudan iki kez kuyruğa giremez.
- Önden alınan hücre çıkışsa `once` zinciri girişe kadar izlenir. Her adımda hücre `2` ile işaretlenir ve yol uzunluğu bir artar.

Doğrulama programında fonksiyon 9 hücrelik yolu buldu ve labirenti yığınlı aramayla aynı biçimde işaretledi; bu labirentte en kısa yol tektir. Kuyruğa en çok 13 hücre girer. Süre $O(m \cdot p)$, ek bellek (`once`, `gordu`, kuyruk) $O(m \cdot p)$'dir.

8.11.6 Özyinelemesiz kök önde ve kök sonda gezinme

Çözüm, özyinelemesiz ağaç gezinmesini anlatan kesimdeki `kok_basta` ve `kok_sonda` fonksiyonlarıdır. Kök önde gezinmede kök yığına konur; her turda üstteki düğüm alınır, yazdırılır, önce sağ sonra sol çocuğu yığına konur. Kök sonda gezinme iki yığınla yapılır: birinci yığın "kök, sağ, sol" sırasını üretir, ikinci yığın bu sırayı ters çevirir. Tek yığınla da yazılabilir; o zaman her düğüm için sağ alt ağacının gezilip gezilmediğini gösteren bir ziyaret bayrağı ya da son yazdırılan düğümün adresi tutulur.

8.12 Özet

- **Yığın** son giren ilk çıkar kuralıyla çalışır. Dizi gerçeklemesinde üste koyma `y[++top] = x`, üstten alma `y[top--]`'dır; ikisi de $O(1)$'dir.
- **Geri izleme**: labirentte her yol ayrımı yığına konur; çıkmazda yığından alınarak son ayrıma dönülür. Kayıttaki yön alanı, dönüşte daha önce denenmiş yönlerin atlanmasını sağlar. Çevredeki duvar çerçevesi sınır denetimini, `gezildi` matrisi sonsuz döngüyü önler.
- **En kötü durum**, uzun ve yanlış bir ilk yoldur: doğru yol dışındaki bütün bölge gezilir. `izbul` $O(m \cdot p)$ zaman ve bellek kullanır; bulduğu yol en kısa olmak zorunda değildir. Kuyrukla yapılan genişlik öncelikli arama en kısa yolu garanti eder.
- **Gösterimler**: ara ek gösterim öncelik ve parantez ister; önek ve sonek gösterim parantezsiz ve önceliksiz hesaplanır. Üç gösterimde de işlenenlerin sırası aynıdır.
- **Sonek hesaplama**: işlenen yığına konur; işleçte önce `op2` (sağ), sonra `op1` (sol) alınır, sonuç yığına konur. $\Theta(n)$. Önek hesaplamada ifade sağdan sola okunur ve ilk alınan sol işlenendir.
- **Ara ekten soneke**: işlenen çıktıya yazılır; yığındaki işleç $isp \geq icp$ oldukça çıktıya alınır. `(` gelirken en yüksek, yığında en düşük önceliklidir; `)` gelince `(`'a kadar boşaltılır. Eşitlikte almak soldan birleşmeyi sağlar. $\Theta(n)$.
- **Parantez denetimi**: açan yığına konur, kapanan üstteki açanla eşleşmelidir; sonda yığın boş olmalıdır. Üç hata türü: yanlış eşleşme, fazla kapanan, kapanmamış açan.
- **Genel ilke**: "en son açılan ilk kapanır" görülen her yerde yığın, "sırayla" görülen yerde kuyruk. Geri al/yinele, özyinelemesiz gezinme, sonraki büyük öğe ve kayan pencere bu ilkenin başka uygulamalarıdır.
- Fonksiyonlar **bileşen gibi** yazılmalıdır: girdiler parametreyle gelir, sonuç dönüş değeriyle gider, küresel değişkenlere bağımlılık kurulmaz.

8.13 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

yığın *stack* · son giren ilk çıkar *LIFO* · geri izleme *backtracking* · labirent *maze* · arama tablosu *look-up table* · kaba kuvvet *brute force* · en kötü durum *worst case* · derinlik öncelikli arama *depth-first search (DFS)* · genişlik öncelikli arama *breadth-first search (BFS)* · en kısa yol *shortest path* · ifade değerlendirme *expression evaluation* · işleç önceliği *operator precedence* · birleşme yönü *associativity* · ara ek *infix* · önek *prefix (Polish notation)* · sonek *postfix (reverse Polish notation)* · belirteç *token* · sayım tipi *enumeration (enum)* · yığın içi öncelik *in-stack precedence (isp)* · gelen öncelik *incoming precedence (icp)* · küresel değişken *global variable* · parantez denetimi *balanced parentheses* · kısa devre değerlendirme *short-circuit evaluation* · geri al *undo* · çift uçlu kuyruk *deque (double-ended queue)* · ifade ağacı *expression tree* · çağrı yığını *call stack*

8.14 Kendini Dene

1. `a*b-(c+d)/e` ifadesinin sonek ve önek karşılıkları nedir?

   Cevap

   Tam parantezli biçim `((a*b) - ((c+d)/e))`'dir. Sonek: `ab*cd+e/-`. Önek: `-*ab/+cde`. `postfix` yordamında `-` gelince yığındaki `*` çıktıya alınır; `(` `-`'nin üstüne biner; `)` gelince `+` çıktıya geçer; `/` `-`'nin üstüne biner. Sonda `/` ve `-` çıkar.

2. `82/3-42*+` sonek ifadesinin değeri nedir? Yığın en çok kaç değer tutar?

   Cevap

   8 / 2 = 4, 4 − 3 = 1, 4 · 2 = 8, 1 + 8 = 9. Sonuç 9'dur. Yığın en çok üç değer tutar: `4` ve `2` konduğu anda yığında 1 · 4 · 2 vardır.

3. `prefix_eval("+*23-84")` hangi değeri döndürür? `-` işlenirken `op1` ve `op2` nedir?

   Cevap

   Sağdan sola: 4, 8 yığına. `-`: ilk alınan `op1` = 8, `op2` = 4, sonuç 4. Sonra 3 ve 2 yığına; `*`: `op1` = 2, `op2` = 3, sonuç 6. `+`: *6 + 4 = 10*. Sonuç 10'dur.

4. `dengeli("{[}]")` ne döndürür? Hangi karakterde ve neden?

   Cevap

   `false`. `{` ve `[` yığına konur. Üçüncü karakter `}` geldiğinde yığının üstünde `[` vardır; `eslesir('[', '}')` yanlıştır. Yanlış eşleşme hatasıdır. Açan ve kapanan sayıları eşit olduğu için tek bir sayaç bu hatayı yakalayamazdı.

5. `postfix` yordamında karşılaştırma `>=` yerine `>` yazılırsa `a-b-c` nasıl çevrilir? `a = 9`, `b = 5`, `c = 2` için iki sonucun değeri nedir?

   Cevap

   İkinci `-` geldiğinde ilk `-` yığından alınmaz; ikinci `-` onun üstüne biner. Çıktı `abc--` olur, yani `a-(b-c)`. Değeri *9 − 3 = 6*'dır. Doğru çeviri `ab-c-`'nin değeri *(9 − 5) − 2 = 2*'dir.

6. `izbul`'da başlangıç kaydının yön alanı 1 yerine 0 olsaydı bulunan yol değişir miydi?

   Cevap

   Değişmezdi. Yön 0, `(1,1)`'den kuzeye, yani çerçevedeki `[0][1]` duvarına bakar. Bu yön denenir, duvar olduğu için `++yon` ile atlanır ve arama yön 1'den aynı biçimde sürer. Yalnız bir gereksiz deneme eklenir.

7. `isp('(')` 0 değil de 20 olsaydı `a*(b+c)` nasıl çevrilirdi?

   Cevap

   `+` geldiğinde *isp(() = 20 ≥ icp(+) = 12* olur ve `(` çıktıya alınır; ardından `*` da çıktıya alınır (*13 ≥ 12*). Çıktıya parantez karışır ve sıra bozulur. `)` geldiğinde ise yığında `(` bulunmaz; iç döngü dip işaretini de geçip dizi dışına okur. `(`'ın yığında en düşük öncelikli olması bu yüzden zorunludur.

8. `(a+b)*(c-d)/e` ifadesinin soneki nedir? `a = 5`, `b = 3`, `c = 4`, `d = 2`, `e = 1` için değeri kaçtır?

   Cevap

   Sonek: `ab+cd-*e/`. `*` ve `/` eşit önceliklidir ve soldan birleşir; `/` geldiğinde `*` çıktıya alınır. Değer: *5 + 3 = 8, 4 − 2 = 2, 8 · 2 = 16, 16 / 1 = 16*.

9. Labirentte yığın yerine kuyruk kullanılırsa neden ilk bulunan yol en kısa olur? Kuyruğa giren bir hücre neden girerken işaretlenir?

Cevap

Kuyruk hücreleri girişe uzaklık sırasıyla verir: uzaklığı *d* olan bütün hücreler, uzaklığı *d + 1* olanlardan önce işlenir. Bir hücreye ilk ulaşan yol bu yüzden en kısa yoldur. Hücre girerken işaretlenmezse, kuyruktan çıkmadan önce başka bir komşu onu ikinci kez kuyruğa ekleyebilir; kuyruk gereksiz yere büyür ve `once` bilgisi yanlış bir yolla üzerine yazılabilir.

8.15 İleri okuma

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, Kesim 3.5 ve 3.6. Labirent yordamı, `isp` / `icp` tablosu ve ifade değerlendirme bu notta izlenen biçimiyle oradadır.
- R. Sedgewick, *Algorithms in C*, Kısım 1–4, Kesim 4.3. Sonek hesaplama ve ara ekten soneke çevirme, yığın soyut veri tipinin ilk uygulamaları olarak ele alınır; özyineleme ile yığın arasındaki ilişki 5. bölümde işlenir.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3. baskı, MIT Press, 2009, Bölüm 10 ve 22. Yığın ve kuyruğun dizi ile gerçeklenmesi, genişlik ve derinlik öncelikli aramanın doğruluk kanıtlarıyla birlikte anlatılır.

8.16 Kaynaklar

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. bs., Silicon Press, 2008. Bölüm 3.5 (labirent), 3.6 (ifade değerlendirme).
2. R. Sedgewick, *Algorithms in C*, Kısım 1–4, 3. bs., Addison-Wesley, 1998. Bölüm 4.3 (sonek), 5 (özyineleme ve yığın).
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3. bs., MIT Press, 2009.

Liste ADT ve Genelleştirilmiş Listeler

Bir listeyle ne yapılabileceği ile listenin bellekte nasıl tutulduğu iki ayrı sorudur. Bu not önce birinci soruyu cevaplar: liste soyut veri tipinin işlemlerini, ön ve son koşullarıyla bir sözleşme olarak yazar. Ardından aynı sözleşmeyi bir diziyle ve bir bağlı listeyle gerçekler, iki gerçeklemeyi maliyet açısından karşılaştırır. İkinci yarıda liste kavramı genişler. Öğeleri yalnız tek değerler değil, başka listeler de olabilen genelleştirilmiş listeler tanıtılır. Bu yapı üzerinde kopyalama, eşitlik, gezinme, derinlik ve ters çevirme yordamları özyinelemeli olarak yazılır ve adım adım izlenir. Son kesimler bir ağacın tek satırlık bir metinle nasıl saklandığını, paylaşılan alt listelerin doğurduğu sorunları, referans saymayı ve çöp toplamayı ele alır.

Kod (C): `github.com/sercankulcu/data-structures-and-algorithms-c/Bolum09`

ÖĞRENME ÇIKTILARI

Sözleşme
Liste soyut veri tipini işlemleri, ön ve son koşullarıyla tanımlamak; opak tiple arayüzü gerçeklemeden ayırmak.

Karşılaştırma
Dizi ve bağlı gerçeklemeyi erişim, ekleme, silme ve bellek açısından karşılaştırmak; dinamik dizide büyütmenin amortize maliyetini açıklamak.

Genelleştirilmiş liste
Atom ve alt liste içeren listeleri tanımlamak, C'de etiketli bir `union` ile temsil etmek ve bellek görünümünü çizmek.

Özyinelemeli yordamlar
Kopyalama, eşitlik, gezinme, derinlik ve ters çevirmeyi özyinelemeli yazmak, izlemek ve maliyetlerini gerekçelendirmek.

Ağaç gösterimi
Bir ağacı tek satırlık doğrusal forma çevirmek ve ikili ağaçta bu yöntemin sınırını açıklamak.

Paylaşım
Paylaşımlı ve özyinelemeli listelerdeki tuzakları, referans saymayı ve çöp toplama fikrini açıklamak.

9.1 Liste soyut veri tipi

Bir program listeyi kullanırken iki şey bilmek ister: listeye hangi işlemlerin uygulanabileceğini ve her işlemin sonunda listenin ne durumda olacağını. Listenin bellekte bir dizi mi yoksa birbirine bağlı düğümler mi olduğu, kullanan kodun işi değildir. Bu ayrım yazılım mühendisliğinin temel fikirlerinden biridir ve bu kesimin konusudur.

9.1.1 Doğrusal liste ve sözleşme

Tanım 9.1

Doğrusal liste, $n \geq 0$ öğenin sıralı bir dizisidir: $L = (a_0, a_1, \ldots, a_{n-1})$. n listenin **uzunluğudur**; $n = 0$ ise liste boştur. Her $i < n - 1$ için a_i, a_{i+1}'den önce gelir. Bir öğenin listedeki sırası, yani i, onun **konumudur**.

Tanım yalnız sıradan söz eder. Öğelerin bellekte yan yana mı durduğu, bir öğeden ötekine nasıl geçildiği tanımda yoktur. Haftanın günleri, bir sınıfın öğrenci numaraları ya da bir metnin satırları bu tanıma uyar. Bu listelerle yapılacak işler de bellidir: i. öğeyi okumak, bir konuma öğe eklemek, bir konumdaki öğeyi silmek, bir değeri aramak ve uzunluğu öğrenmek.

İşlemleri ve davranışlarını tanımlayıp gerçeklemeyi gizleyen tipe **soyut veri tipi** (*abstract data type*, ADT) denir. **Liste ADT**, konuma göre ekleme, silme ve erişim sunan bir soyut veri tipidir. C'de bir soyut veri tipinin dışa açık yüzü, genellikle bir **başlık dosyasına** (*header file*) yazılan bildirimlerdir. Sunum liste ADT'sinin arayüzünü şöyle verir:

```c
typedef struct liste Liste;          // opak tür

Liste *liste_olustur(void);
void   liste_yok_et(Liste *L);
int    liste_uzunluk(const Liste *L);
bool   liste_getir(const Liste *L, int i, int *x);
bool   liste_ekle(Liste *L, int i, int x);  // 0<=i<=n
bool   liste_sil(Liste *L, int i, int *x);  // 0<=i<n
int    liste_ara(const Liste *L, int x);    // yoksa -1
```

Her fonksiyonun adı yaptığı işi söyler. `liste_olustur` boş bir liste kurup adresini döndürür; `liste_yok_et` listeye ait bütün belleği iade eder. `liste_getir(L, i, &x)` i. öğeyi `x`'e yazar, `liste_ekle(L, i, x)` `x`'i i. konuma ekler, `liste_sil(L, i, &x)` i. öğeyi çıkarıp `x`'e yazar. `liste_ara` aranan değerin konumunu, bulamazsa −1 döndürür. Değer döndüren işlemler sonucu bir işaretçi parametresiyle verir; böylece dönüş değeri işlemin başarılı olup olmadığını bildirmek için boşta kalır.

Bir fonksiyonun adı ve parametreleri tek başına yeterli değildir. `liste_ekle(L, 7, x)` çağrısı uzunluğu 3 olan bir listede ne yapmalıdır? Bu soruların cevabı, her işlem için yazılan **ön koşul** ve **son koşuldur**. Ön koşul, işlem çağrılmadan önce doğru olması gereken durumdur. Son koşul, ön koşul sağlanmışsa işlemin bitiminde doğru olacağı garanti edilen durumdur. İkisi birlikte işlemin **sözleşmesini** oluşturur. `liste_ekle` için sözleşme şudur:

- **Ön koşul:** $0 \le i \le n$. $i = n$ sona eklemek demektir; bu yüzden üst sınır $n - 1$ değil n'dir.
- **Son koşul:** `x` i. konumdadır, uzunluk $n + 1$ olmuştur ve eski $a_i, \ldots, a_{n-1}$ öğeleri birer konum sağa kaymıştır. Yani eski a_k artık $k + 1$. konumdadır. i'den önceki öğeler yerinde kalır.

`liste_sil` için ön koşul $0 \le i < n$'dir; boş listeden silme bu yüzden her i için geçersizdir. Son koşulda uzunluk bir azalır ve i'den sonraki öğeler bir sola kayar. Kod parçasındaki `// 0<=i<=n` ve `// 0<=i<n` açıklamaları bu ön koşulların kısa yazımıdır.

Ön koşul çiğnenirse ne olacağı da sözleşmenin parçasıdır. Bu arayüzde fonksiyon hiçbir şeyi değiştirmeden `false` döndürür. Başka bir tasarım ön koşulu çağıranın sorumluluğuna bırakabilir ve denetim yapmayabilir. Bu daha hızlıdır ama yanlış bir çağrı tanımsız davranışa yol açar. Hangi yol seçilirse seçilsin, belgelenmesi gerekir. Sözleşmesi yazılmamış bir fonksiyonun doğru mu yanlış mı çalıştığı tartışılamaz.

9.1.2 Opak tip: arayüzü gerçeklemeden ayırmak

Arayüzdeki ilk satır `typedef struct liste Liste;` yapının yalnız adını bildirir, alanlarını vermez. Böyle bir bildirimden sonra `struct liste` bir **eksik tiptir** (*incomplete type*): derleyici adını tanır ama boyutunu bilmez. Eksik tipten değişken tanımlanamaz, ama ona işaretçi tanımlanabilir. Bütün işaretçilerin boyutu aynıdır ve derleyicinin bilmesi gereken tek şey budur. Arayüzdeki bütün fonksiyonlar bu yüzden `Liste` değil, `Liste *` alır.

Yapının alanları yalnız gerçeklemeyi içeren `liste.c` dosyasında yazılır. Başlık dosyasını kullanan bir program `L->n` gibi bir ifade yazamaz, çünkü `n` diye bir alanın varlığından haberi yoktur. Listeye ancak arayüzdeki fonksiyonlar aracılığıyla dokunabilir. İçi dışarıdan görülmeyen böyle bir tipe **opak tip** (*opaque type*) denir. Standart kütüphanedeki `FILE` buna tanıdık bir örnektir: programlar `FILE *` ile çalışır ama yapının alanlarını kullanmaz.

Aşağıdaki program listeyi yalnız arayüz üzerinden kullanır. Önce başa beş değer ekler, sonra 2. konumdaki öğeyi siler:

```c
#include <stdio.h>
#include "liste.h"                    // yalnız arayüz

int main(void) {
    Liste *L = liste_olustur();
    int x, k;
    if (L == NULL) return 1;
    for (k = 1; k <= 5; k++)
        liste_ekle(L, 0, 10 * k);     // hep başa: 50 40 30 20 10
    liste_sil(L, 2, &x);              // 30 çıkar
    printf("%d %d %d\n", liste_uzunluk(L), x, liste_ara(L, 20));
    liste_yok_et(L);
    return 0;
}
```

Her ekleme başa yapıldığı için döngüden sonra liste *(50, 40, 30, 20, 10)* olur. `liste_sil(L, 2, &x)` 30'u çıkarır ve liste *(50, 40, 20, 10)*'a döner. 20'nin konumu artık 2'dir. Program `4 30 2` yazar. Bu program, biri dizi, öteki bağlı liste kullanan iki ayrı `liste.c` ile derlenip çalıştırılmıştır; iki gerçekleme de bu notta yer alır. İki durumda da çıktı aynıdır ve `main`'in tek satırı bile değişmemiştir. Opak tipin kazancı budur. Gerçekleme değiştiğinde yalnız `liste.c` yeniden derlenir; listeyi kullanan yüzlerce dosya olduğu gibi kalır.

> **Sık yapılan hata**
>
> Kullanan dosyada `Liste L;` ya da `malloc(sizeof(Liste))` yazmak. `Liste` o dosyada eksik bir tiptir ve derleyici boyutunu bilmez; iki satır da derleme hatası verir. Listeyi ancak boyutunu bilen `liste_olustur` ayırabilir. Bu kural bir kısıtlama değil, opak tipin amacıdır.

9.2 Dizi ile gerçekleme

Liste ADT'sini gerçeklemenin en doğrudan yolu, öğeleri bir dizide sırasıyla tutmaktır. a_i dizinin i. hücresine yazılır. Bu düzene **sıralı yerleşim** denir: mantıksal sıra ile bellekteki sıra aynıdır.

9.2.1 Sıralı yerleşim ve silme

Sunumdaki ilk gerçekleme sabit boyutlu bir dizi ve bir öğe sayacı kullanır:

```
#define KAPASITE 100
typedef struct {
    int a[KAPASITE];
    int n;                  // öğe sayısı
} DiziListe;

bool dizi_sil(DiziListe *L, int i, int *x) {
    int k;
    if (i < 0 || i >= L->n) return false;
    *x = L->a[i];
    for (k = i; k < L->n - 1; k++)
        L->a[k] = L->a[k + 1];      // sola kaydır
    L->n--;
    return true;
}
```

`DiziListe` iki alanlıdır. `a` en çok `KAPASITE` öğe alabilen dizidir; `n` o anda kaç hücrenin dolu olduğunu tutar. Dolu hücreler her zaman `a[0]`'dan `a[n-1]`'e kadar olanlardır; arada boşluk bırakılmaz. Bu kural yapının **değişmezidir** ve her işlemin sonunda korunmalıdır. `int` 4 bayt olduğundan yapı 404 bayt yer tutar. Listede bir öğe de olsa yüz öğe de olsa bu boyut değişmez.

`dizi_sil` önce ön koşulu denetler. i geçerli bir konum değilse hiçbir şeye dokunmadan `false` döndürür. Geçerliyse silinecek değeri `*x`'e kopyalar. Ardından i'den sonraki her öğeyi bir sola taşır: `a[i] = a[i+1]`, `a[i+1] = a[i+2]` ve böyle devam ederek `a[n-2] = a[n-1]`'e kadar. Döngünün `k < L->n - 1` koşulu bu yüzdendir; `k` $n-1$'e ulaşsaydı `a[n]` okunurdu ve o hücrede geçerli bir öğe yoktur. En sonda `n` bir azalır. Son hücrede eski değer kalır ama artık `n`'nin dışında olduğu için listeye ait sayılmaz.

a[i]'ye erişim tek bir adres hesabıdır: dizinin başlangıç adresine $i \cdot sizeof(int)$ eklenir. Bu hesap i'nin büyüklüğünden bağımsızdır; `liste_getir` bu yüzden $O(1)$'dir. Silmede ise taşınan öğe sayısı $n-1-i$'dir. $i = n-1$, yani sondan silme hiç taşıma gerektirmez ve $O(1)$'dir. $i = 0$, yani baştan silme $n-1$ taşıma yapar ve $\Theta(n)$'dir. Bütün konumlar eşit olasılıkla silinirse ortalama taşıma sayısı $(n-1)/2$'dir; ortalama durum da $\Theta(n)$'dir.

9.2.2 Kod izleme: `dizi_ekle(&L, 1, 9)`

Ekleme silmenin aynası gibidir. Silmede boşluk kapatılırken, eklemede önce yer açılır:

```c
bool dizi_ekle(DiziListe *L, int i, int x) {
    int k;
    if (L->n == KAPASITE || i < 0 || i > L->n)
        return false;          // dolu/geçersiz
    for (k = L->n; k > i; k--)
        L->a[k] = L->a[k - 1];     // sağa kaydır
    L->a[i] = x;
    L->n++;
    return true;
}
```

Fonksiyon üç durumda reddeder: dizi doluysa, i negatifse ya da $i > n$ ise. `||` işleci soldan sağa ve kısa devreyle değerlendirilir; ilk doğru koşulda öteki koşullara bakılmaz. Ardından döngü `k = n`'den başlayıp geriye doğru ilerler. Her turda `a[k-1]`'i `a[k]`'ye taşır. Döngü `k == i` olduğunda durur. Bu anda `a[i]`'nin eski değeri `a[i+1]`'e kopyalanmıştır ve hücre yeni değeri almaya hazırdır.

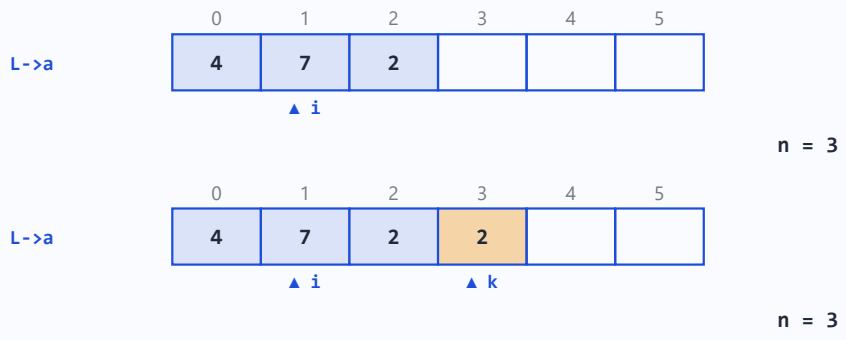
Taşımanın yönü kritiktir. Döngü soldan sağa, yani `k = i`'den başlasaydı, `a[i+1] = a[i]` ataması `a[i+1]`'in eski değerini silerdi. Bir sonraki turda o hücreden okunan değer artık `a[i]`'nin kopyası olurdu ve bütün sağ taraf aynı değerle dolardı. Sağdan sola taşımada her hücre, üzerine yazılmadan önce sağa kopyalanmış olur.

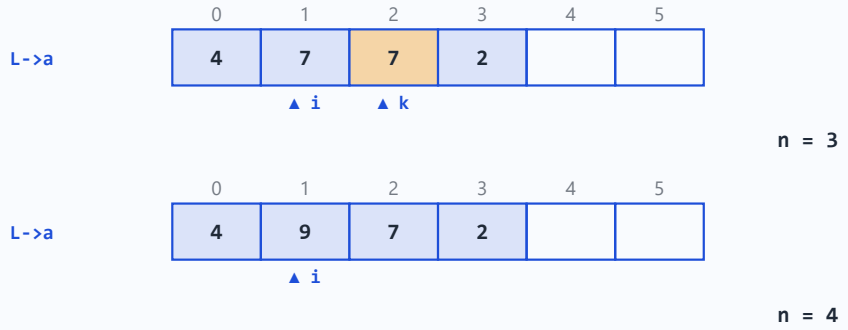
Örnek 9.1

Liste *[4, 7, 2]*, *n = 3*'tür. 9 değeri 1. konuma eklenecektir (konumlar 0'dan başlar). Tablo 9.1 her satırın etkisini, Şekil 9.1 dizinin dört anını gösterir.

Satır	k	Koşul / işlem	a[0..3]	n
Giriş	?	`i = 1`, `x = 9`	4, 7, 2, —	3
`if (…)`	?	dolu değil, *0 ≤ 1 ≤ 3*: devam	4, 7, 2, —	3
`for`	3	*3 > 1* doğru	4, 7, 2, —	3
`a[k] = a[k-1]`	3	`a[3] = a[2]` : 2 sağa kaydı	4, 7, 2, 2	3
`for`	2	*2 > 1* doğru	4, 7, 2, 2	3
`a[k] = a[k-1]`	2	`a[2] = a[1]` : 7 sağa kaydı	4, 7, 7, 2	3
`for`	1	*1 > 1* yanlış: döngü biter	4, 7, 7, 2	3
`a[i] = x`	1	boşalan konuma 9	4, 9, 7, 2	3
`n++`	—	liste *[4, 9, 7, 2]*	4, 9, 7, 2	4

Tablo 9.1: `dizi_ekle(&L, 1, 9)` izlemesi. Dizi sütunu `a[0..3]`'ü gösterir; `a[3]` başlangıçta listeye ait değildir (—).





Şekil 9.1: `dizi_ekle(&L, 1, 9)` sırasında dizinin dört anı. Yukarıdan aşağıya: başlangıç; 2'nin `a[3]` 'e kopyalanması; 7'nin `a[2]` 'ye kopyalanması (bu anda 7 iki hücrede birden durur); 9'un `a[1]` 'e yazılıp `n` 'nin 4 olması.

Tablodaki iki ara durum dikkat ister. `a[3] = a[2]` 'den sonra 2 değeri iki hücrededir; `a[2] = a[1]` 'den sonra aynı durum 7 için olur. Bu geçici kopyalar bir hata değildir. Yerine yazılacak değer her zaman bir sağdaki hücreye önceden taşınmıştır. Döngünün sonunda yalnız `a[i]` fazladan bir kopya taşır ve hemen ardından `x` onun üzerine yazılır.

Kaydırma sayısı $n - i$'dir. Sona ekleme ($i = n$) hiç kaydırma yapmaz; başa ekleme ($i = 0$) bütün n öğeyi kaydırır. En kötü durum bu yüzden $\Theta(n)$'dir. Boş bir diziye yüz kez başa ekleme yapan bir deneme bunu doğrular: toplam kaydırma sayısı $0 + 1 + \dots + 99 = 4950$ çıkmıştır. Yani n kez başa ekleme toplamda $n(n-1)/2$ kaydırma, $\Theta(n^2)$ iş demektir. Dolu dizide yapılan yüz birinci ekleme `false` döner.

9.2.3 Kapasite ve iki katına büyütme

Sabit dizinin iki sakıncası vardır. Kapasite derleme zamanında seçilir ve aşılamaz. Seçilen kapasite büyükse, kısa listeler de aynı büyük alanı boşuna tutar. Çözüm, diziyi heap bellekte ayırıp gerektiğinde `realloc` ile büyütmektir. Böyle bir diziye **dinamik dizi** (*dynamic array*) denir. Asıl soru ne kadar büyütüleceğidir.

Dizi her dolduğunda bir hücre büyütülürse, her ekleme bir `realloc` çağırır. `realloc` bloğu yerinde büyütemezse yeni bir blok ayırır ve eski öğelerin hepsini oraya kopyalar. n eklemede kopyalanan öğe sayısı en kötü durumda $1 + 2 + \dots + (n-1)$, yani $\Theta(n^2)$ olur. Kapasite her seferinde iki katına çıkarılırsa tablo değişir. Aşağıdaki `liste.c`, arayüzü bu yöntemle gerçekler. Kapasite 4 ile başlar ve dizi her dolduğunda iki katına çıkar:

```c
// liste.c: struct liste yalnız burada tanımlıdır (opak tür)
struct liste {
    int *a;                          // heap'teki dizi
    int n, kap;                      // öğe sayısı, kapasite
};

Liste *liste_olustur(void) {
    Liste *L = malloc(sizeof *L);
    if (L == NULL) return NULL;
    L->n = 0;
    L->kap = 4;
    L->a = malloc(L->kap * sizeof *L->a);
    if (L->a == NULL) { free(L); return NULL; }
    return L;
}

void liste_yok_et(Liste *L) {
    if (L) { free(L->a); free(L); }
}

int liste_uzunluk(const Liste *L) { return L->n; }

bool liste_getir(const Liste *L, int i, int *x) {
    if (i < 0 || i >= L->n) return false;
    *x = L->a[i];
    return true;
}

bool liste_ekle(Liste *L, int i, int x) {
    int k;
    if (i < 0 || i > L->n) return false;
    if (L->n == L->kap) {            // dolu: iki katına çıkar
        int *yeni = realloc(L->a, 2 * L->kap * sizeof *L->a);
        if (yeni == NULL) return false; // eski dizi yerinde kalır
        L->a = yeni;
        L->kap *= 2;
    }
    for (k = L->n; k > i; k--)
        L->a[k] = L->a[k - 1];
    L->a[i] = x;
    L->n++;
    return true;
}

bool liste_sil(Liste *L, int i, int *x) {
    int k;
    if (i < 0 || i >= L->n) return false;
    *x = L->a[i];
    for (k = i; k < L->n - 1; k++)
        L->a[k] = L->a[k + 1];
    L->n--;
    return true;
}

int liste_ara(const Liste *L, int x) {
    for (int k = 0; k < L->n; k++)
        if (L->a[k] == x) return k;
    return -1;
}
```

Yapı artık dizinin kendisini değil, heap'teki bir dizinin adresini tutar. `liste_olustur` iki ayrı blok ayırır: biri yapı, biri dizi için. İkinci ayırma başarısız olursa birincisi de iade edilir; aksi hâlde yapı sızardı. `liste_yok_et` blokları ters sırayla bırakır. Önce dizi, sonra yapı iade edilir, çünkü dizinin adresi yapının içindedir.

`liste_ekle`'deki `realloc` çağrısı sonucu önce ayrı bir değişkene alır. `L->a = realloc(L->a, …)` yazılsaydı ve bellek yetmeseydi, `realloc` `NULL` döndürür, ama eski bloğu bırakmaz. `L->a`'ya `NULL` yazıldığı anda eski bloğun adresi kaybolur ve bütün liste sızar. `yeni` değişkeni bu durumda eski diziyi korur; fonksiyon `false` döner ve liste bozulmadan kalır.

Bu gerçeklemeyle yukarıdaki örnek program çalıştırıldığında beşinci eklemede tek bir `realloc` çağrısı görülmüştür: kapasite 4'ten 8'e, blok 16 bayttan 32 bayta çıkmıştır. Program iki `malloc` ve iki `free` yapmıştır; sızıntı yoktur.

> **Teorem 9.1 (İki katına büyütmenin amortize maliyeti)**
>
> Kapasitesi $c_0 \geq 1$ ile başlayan ve her dolduğunda iki katına çıkan dinamik diziye sona n ekleme yapılsın. Büyütmeler sırasında kopyalanan toplam öğe sayısı $2n$'den azdır. Dolayısıyla sona eklemenin **amortize maliyeti** $O(1)$'dir.

Kanıt. Büyütme, öğe sayısı kapasiteye eşitken yeni bir ekleme geldiğinde olur ve o anda bütün öğeler kopyalanır. Kapasiteler $c_0, 2c_0, 4c_0, \dots$ diye gider. Son büyütme kapasite $c_0 2^k$ iken yapılmış olsun; bu büyütme $c_0 2^k$ öğe kopyalar ve o anda dizide bu kadar öğe vardır. Demek ki $c_0 2^k < n$'dir. Toplam kopya sayısı geometrik bir toplamdır: $c_0(1 + 2 + \dots + 2^k) = c_0(2^{k+1} - 1) < 2 \cdot c_0 2^k < 2n$. Hiç büyütme olmadıysa toplam sıfırdır. n eklemenin kendisi n yazma yapar. Kopyalarla birlikte toplam iş $3n$'den azdır ve ekleme başına düşen pay sabittir. ■

Amortize maliyet, bir işlemler dizisinin toplam maliyetinin işlem sayısına bölümüdür. Tek tek eklemelerin hepsi ucuz değildir. Büyütmeye denk gelen ekleme $\Theta(n)$ sürer; ama böyle eklemeler o kadar seyrektir ki toplam iş doğrusal kalır. Büyütme oranı 2 yerine 1,5 gibi başka bir sabit de olabilir; kanıt aynı kalır, yalnız sabit değişir. Sabit bir miktar eklemek ise (her seferinde 10 hücre gibi) yeterli değildir. O durumda büyütme sayısı $n/10$ olur ve toplam kopya yine $\Theta(n^2)$'dir.

> **Sık yapılan hata**
>
> Amortize $O(1)$'i "her ekleme $O(1)$" diye okumak. Gerçek zamanlı bir sistemde tek bir eklemenin bile uzun sürmesi kabul edilemiyorsa, ortalama değil en kötü durum önemlidir. O zaman dinamik dizi yerine bağlı liste ya da önceden ayrılmış bir dizi seçilir. Ayrıca bu teorem yalnız sona eklemeyi anlatır. Başa ekleme, büyütme olmasa da kaydırma yüzünden $\Theta(n)$ kalır.

9.3 Bağlı gerçekleme

Bağlı gerçeklemede her öğe ayrı bir düğümde durur ve her düğüm bir sonrakinin adresini taşır. Ekleme ve silme artık öğe taşımaz, yalnız bağ değiştirir. Bu kesimde kullanılan düğüm, bir tam sayı değer ve bir bağdan oluşan klasik tek yönlü bağlı liste düğümüdür.

9.3.1 İşaretçiye işaretçi ile ekleme

```c
typedef struct dugum *dugum_gost;
struct dugum { int deger; dugum_gost bag; };
typedef struct { dugum_gost bas; int n; } BListe;

bool bl_ekle(BListe *L, int i, int x) {
    dugum_gost *pp = &L->bas, yeni;
    if (i < 0 || i > L->n) return false;
    while (i-- > 0)
        pp = &(*pp)->bag;        // i. bağa yürü
    yeni = malloc(sizeof *yeni);
    if (yeni == NULL) return false;
    yeni->deger = x;
    yeni->bag = *pp;
    *pp = yeni;                  // baş ya da ara: aynı
    L->n++;
    return true;
}
```

`BListe` yapısı listenin başını ve uzunluğunu birlikte tutar. Uzunluk ayrı bir alanda saklandığı için `liste_uzunluk` listeyi saymak zorunda kalmaz ve $O(1)$'de cevap verir. Bunun bedeli, her ekleme ve silmede `n`'yi güncellemeyi unutmamaktır.

Tek yönlü listede i. konuma eklemek için, yeni düğümü gösterecek bağı değiştirmek gerekir. $i = 0$ ise değişecek bağ listenin başı, yani `L->bas`'tır. $i > 0$ ise değişecek bağ $(i-1)$. düğümün `bag` alanıdır. Klasik yazımda bu iki durum ayrı ayrı ele alınır: başa ekleme için bir dal, araya ekleme için önceki düğümü bulan başka bir dal yazılır.

`bl_ekle` iki durumu tek bir fikirle birleştirir. Değiştirilecek bağın kendisini değil, **adresini** tutar. `dugum_gost *pp` bir düğüm işaretçisinin adresini tutan bir değişkendir; bu yüzden ona **işaretçiye işaretçi** (*pointer to pointer*) denir. Başlangıçta `pp = &L->bas`'tır. Döngünün her turunda `pp = &(*pp)->bag` ile bir sonraki bağın adresine geçilir. Bu ifade içten dışa okunur: `*pp` şu anki bağın değeri, yani bir düğü-

mün adresidir; `(*pp)->bag` o düğümün bağ alanı, `&(*pp)->bag` ise bu alanın adresidir. *i* tur sonra `pp`, *i*. konumdaki düğümü gösteren bağın adresini tutar.

Bağlama iki satırdır. `yeni->bag = *pp` yeni düğümü, o konumda şu an duran düğüme bağlar. `*pp = yeni` ise o bağı yeni düğüme çevirir. `pp` `&L->bas`'ı gösteriyorsa listenin başı değişir; bir düğümün `bag` alanını gösteriyorsa o düğümün bağı değişir. Kod iki durumu ayırt etmez, çünkü ikisi de "bir işaretçinin değerini değiştirmek"tir. Satırların sırası önemlidir. Önce `*pp = yeni` yazılsaydı, `yeni->bag = *pp` yeni düğümü kendine bağlardı ve listenin geri kalanı kaybolurdu.

Satır	`i`	`pp`	`*pp`	Liste
Giriş	1	`&B.bas`	&düğüm(4)	4 → 7 → 2
`while (i-- > 0)`	0	*1 > 0* doğru; `i` 0 oldu		
`pp = &(*pp)->bag`	0	&düğüm(4)->bag	&düğüm(7)	4 → 7 → 2
`while (i-- > 0)`	−1	*0 > 0* yanlış; döngü biter		
`yeni = malloc(…)`, değer	−1	&düğüm(4)->bag	&düğüm(7)	yeni düğüm: 9, bağ çöp
`yeni->bag = *pp`	−1	&düğüm(4)->bag	&düğüm(7)	9 → 7 (4 hâlâ 7'yi gösterir)
`*pp = yeni`	−1	&düğüm(4)->bag	&düğüm(9)	4 → 9 → 7 → 2
`L->n++`	`n = 4`; `true` döner			

Tablo 9.2: `bl_ekle(&B, 1, 9)` izlemesi; başlangıçta liste 4 → 7 → 2, `n = 3`.

Döngü koşulundaki `i-- > 0` önce `i`'nin eski değerini 0 ile karşılaştırır, sonra `i`'yi bir azaltır. Bu yüzden döngü tam *i* kez döner ve çıkışta `i` −1 olur. Fonksiyon `i`'yi bundan sonra kullanmadığı için bu değer bir sorun yaratmaz. Aynı çağrı *i = 0* ile yapılsaydı döngü hiç dönmez, `pp` `&B.bas`'ta kalır ve `*pp = yeni` listenin başını değiştirirdi. Başa ekleme böylece özel bir durum olmaktan çıkar.

Maliyet iki parçadan oluşur. Konuma yürümek *i* tur, yani *O(i)*'dir. Bağlamak sabit sayıda atama, yani *O(1)*'dir. Toplam *O(i)* olur: başa ekleme *O(1)*, sona ekleme *O(n)*. Dizide bunun tersi geçerliydi. Bağlı listede hiçbir öğe yerinden oynamaz ve kapasite sınırı yoktur; liste, `malloc` bellek verdikçe büyür. Denemede 7, 4 ve 2 sırasıyla 0., 0. ve 2. konumlara eklenmiş, liste 4 → 7 → 2 olmuştur. Ardından `bl_ekle(&B, 1, 9)` 4 → 9 → 7 → 2'yi vermiş, geçersiz konumlar (6 ve −1) `false` ile reddedilmiştir. Tek düğüm 16 bayttır: 4 baytlık `int`, 4 bayt hizalama boşluğu ve 8 baytlık işaretçi.

9.3.2 Aynı arayüz, başka gerçekleme

Opak tipin gücünü görmek için aynı arayüzün bağlı bir gerçeklemesi yazılabilir. Yapı yalnız `liste.c` içinde değişir:

```c
// liste.c'nin bağlı sürümü: liste.h ve kullanan kod değişmez
typedef struct dugum *dugum_gost;
struct dugum { int deger; dugum_gost bag; };
struct liste { dugum_gost bas; int n; };

Liste *liste_olustur(void) {
    Liste *L = malloc(sizeof *L);
    if (L) { L->bas = NULL; L->n = 0; }
    return L;
}

void liste_yok_et(Liste *L) {
    dugum_gost p, q;
    if (L == NULL) return;
    for (p = L->bas; p; p = q) { q = p->bag; free(p); }
    free(L);
}

int liste_uzunluk(const Liste *L) { return L->n; }

static dugum_gost *bag_adresi(Liste *L, int i) {
    dugum_gost *pp = &L->bas;           // i. bağın adresi
    while (i-- > 0) pp = &(*pp)->bag;
    return pp;
}

bool liste_getir(const Liste *L, int i, int *x) {
    dugum_gost p = L->bas;
    if (i < 0 || i >= L->n) return false;
    while (i-- > 0) p = p->bag;
    *x = p->deger;
    return true;
}

bool liste_ekle(Liste *L, int i, int x) {
    dugum_gost *pp, yeni;
    if (i < 0 || i > L->n) return false;
    pp = bag_adresi(L, i);
    yeni = malloc(sizeof *yeni);
    if (yeni == NULL) return false;
    yeni->deger = x;
    yeni->bag = *pp;
    *pp = yeni;
    L->n++;
    return true;
}

bool liste_sil(Liste *L, int i, int *x) {
    dugum_gost *pp, silinen;
    if (i < 0 || i >= L->n) return false;
    pp = bag_adresi(L, i);
    silinen = *pp;
    *x = silinen->deger;
    *pp = silinen->bag;                 // bağ atlanır
    free(silinen);
    L->n--;
    return true;
}

int liste_ara(const Liste *L, int x) {
    int k = 0;
    for (dugum_gost p = L->bas; p; p = p->bag, k++)
        if (p->deger == x) return k;
    return -1;
}
```

`bag_adresi` yardımcı fonksiyonu `bl_ekle`'deki yürüyüşün kendisidir. `static` olduğu için yalnız bu dosyadan görülür ve arayüzün parçası değildir. `liste_sil` aynı yürüyüşü silme için kullanır. `*pp = silinen->bag` ile silinecek düğüm atlanır, ardından düğüm iade edilir. Baştan silme de aradan silme de aynı üç satırdır. `liste_yok_et` düğümlerde gezinirken bir sonrakinin adresini `free`'den önce `q`'ya alır.

İade edilmiş bir düğümün `bag` alanını okumak tanımsız davranıştır.

Örnek program bu dosyayla da derlenmiş ve aynı `4 30 2` çıktısını vermiştir. Bu kez altı `malloc` yapılmıştır: beşi düğümler, biri yapının kendisi için. Altı `free` ile hepsi iade edilmiştir. Kullanan kod için iki gerçekleme arasında görünen tek fark hızdır.

9.3.3 Dizi mi bağlı liste mi?

İki gerçeklemenin hangisinin daha iyi olduğu sorusunun tek bir cevabı yoktur. Cevap, programın hangi işlemleri sık yaptığına bağlıdır. Tablo 9.3 sunumdaki karşılaştırmayı verir.

Ölçüt	Dizi (sıralı)	Tek yönlü bağlı liste
i. öğeye erişim	*O(1)*	*O(i)*
Başa ekleme / silme	*O(n)* kaydırma	*O(1)*
Sona ekleme	*O(1)** (yer varsa)	*O(n)*, son işaretçisiyle *O(1)*
Bilinen konumda ekleme	*O(n)*	*O(1)* (önceki biliniyorsa)
Arama (sırasız)	*O(n)*	*O(n)*
Arama (sıralı)	*O(log n)* ikili arama	*O(n)*
Bellek	Yalnız veri; kapasite boşluğu	Düğüm başına işaretçi
Önbellek	Bitişik: çok iyi	Dağınık: zayıf

Tablo 9.3: Sıralı (dizi) ve bağlı gerçeklemenin karşılaştırması.

Tablodaki yıldız, dinamik dizide sona eklemenin amortize *O(1)* olduğunu hatırlatır. “Bilinen konumda ekleme” satırı ince bir fark taşır. Dizide konum bilinse bile ekleme kaydırma yüzünden *O(n)*'dir. Bağlı listede ise eklenecek yerden önceki düğüm elde varsa iki bağ değişir ve iş biter. Bu düğüme ulaşmak için baştan yürümek gerekiyorsa maliyet yine *O(i)*'dir.

İkili arama, ortadaki öğeye *O(1)*'de erişebilmeye dayanır. Bağlı listede ortaya gitmek bile *O(n)* sürdüğü için sıralı bir bağlı listede ikili arama hiçbir kazanç sağlamaz; doğrusal arama kadar sürer.

Son iki satır asimptotik gösterimin yakalamadığı farkları anlatır. Bellek açısından dizi her öğe için yalnız veriyi tutar, ama kapasitenin dolmayan kısmı boşa gider. Bağlı liste boş hücre tutmaz, ama her düğüm bir işaretçi taşır. `int` tutan bir listede bu, 4 baytlık veri için 16 baytlık düğüm demektir. İşlemci açısından fark daha da büyüktür. İşlemci belleği tek tek bayt değil, 64 baytlık bloklar hâlinde **önbelleğe** (*cache*) getirir. Dizide bir hücre okunduğunda komşu hücreler de gelir ve sonraki erişimler çok hızlıdır. Bağlı listenin düğümleri bellekte dağınık durabilir; her `p = p->bag` yeni bir bloğun getirilmesini gerektirebilir. Aynı *O(n)* gezinme dizide bağlı listedekinden birkaç kat hızlı olabilir.

Sunumun özetlediği kural şudur: sık erişim ya da ikili arama gerekiyorsa dizi; sık araya ekleme ve silme yapılıyorsa ya da boyut önceden kestirilemiyorsa bağlı liste. Opak bir arayüz bu kararı ertelemeyi de sağlar. Program önce bir gerçeklemeyle yazılır, ölçülür; gerekirse yalnız `liste.c` değiştirilir.

9.4 Genelleştirilmiş listeler

Doğrusal listede her öğe tek bir değerdir: bir sayı, bir harf ya da bir kayıt. Pek çok veri ise iç içedir. Dosya sisteminde bir dizinin içinde alt dizinler, bir menünün içinde alt menüler, bir matematik ifadesinin içinde parantezli alt ifadeler bulunur. Bu tür verileri doğal biçimde tutmak için listenin öğelerinin de liste olabilmesine izin vermek yeterlidir.

9.4.1 Tanım, baş ve kuyruk

Tanım 9.2

$A = (\alpha_0, \alpha_1, \ldots, \alpha_{n-1})$ sonlu bir öğe dizisi olsun. Her α_i ya bir **atom** (bölünmeyen tek bir değer) ya da yine bu biçimde bir listedir. Böyle bir A'ya **genelleştirilmiş liste** (*generalized list*) denir. Liste olan öğelere A'nın **alt listeleri** denir. A listenin adı, n uzunluğudur. $n \geq 1$ ise α_0 listenin **başı** (*head*), $(\alpha_1, \ldots, \alpha_{n-1})$ ise **kuyruğudur** (*tail*).

Tanım kendine başvurur: bir genelleştirilmiş liste, öğeleri genelleştirilmiş liste olabilen bir listedir. Bu tür tanımlara özyinelemeli tanım denir. Böyle bir yapıyı işleyen yordamların da özyinelemeli olması bu yüzden doğaldır. Doğrusal liste, bütün öğeleri atom olan özel bir genelleştirilmiş listedir.

Baş ile kuyruk arasındaki asimetriye dikkat etmek gerekir. Baş bir öğedir; atom da olabilir, liste de. Kuyruk ise her zaman bir listedir, gerekirse boş liste. *(a)* listesinin başı *a* atomu, kuyruğu *()*'dır. Bu iki işlem, 1958'de John McCarthy'nin tasarladığı LISP dilinin temelidir. LISP'te baş `car`, kuyruk `cdr` adını taşır. LISP'in programları da aynı biçimde, iç içe listeler olarak yazılır: `(+ 1 (* 2 3))` ifadesi hem bir program hem de üç öğeli bir genelleştirilmiş listedir.

Liste	Uzunluk	Açıklama
D = ()	0	boş liste
A = (a, (b, c))	2	ilk öğe atom, ikinci öğe *(b, c)* alt listesi; baş(*A*) = *a*, kuyruk(*A*) = *((b, c))*
B = (A, A, ())	3	paylaşımlı alt liste: ilk iki öğe aynı *A*; üçüncüsü boş liste
C = (a, C)	2	özyinelemeli liste: kendini içerir, *C = (a, (a, (a, ...)))*

Tablo 9.4: Genelleştirilmiş liste örnekleri.

Tablodaki her satır ayrı bir ders taşır. *A*'nın uzunluğu 2'dir, 3 değil. Uzunluk yalnız en üst düzeydeki öğeleri sayar; *(b, c)* tek bir öğedir. *A*'nın kuyruğu *(b, c)* değil *((b, c))*'dir. Kuyruk, baş çıkarıldıktan sonra *kalan liste*dir ve bu liste tek öğelidir; o tek öğe de *(b, c)* listesidir.

B'nin üç öğesinden ikisi aynı *A* listesidir. Bu, *A*'nın iki kopyası anlamına gelmez. Bellekte tek bir *A* vardır ve *B*'nin iki öğesi onu gösterir. Böyle listelere **paylaşımlı** liste denir. *B*'nin üçüncü öğesi boş listedir. Boş liste de bir listedir; *B*'nin uzunluğunu bir artırır.

C kendini öğe olarak içerir. Kâğıt üzerinde açılmaya çalışılırsa *(a, (a, (a, ...)))* diye sonsuza kadar gider. Bellekte ise sonlu bir yapıdır: ikinci öğe, listenin kendi başını gösterir. Bu tür listelere **özyinelemeli** liste denir. Paylaşım ve özyineleme, genelleştirilmiş listelerin gücüdür; ama bu notta yazılan yordamların çoğu ikisini de içermeyen listeler için doğrudur. Sınırlar her yordamın yanında ayrıca belirtilir.

9.4.2 Örnek: çok değişkenli polinom

Tek değişkenli bir polinom, her düğümü bir katsayı ve bir üs taşıyan bağlı listeyle kolayca tutulur. Birden çok değişken içeren polinomlarda iş zorlaşır. Her terimde her değişkenin üssünü tutan sabit bir yapı, değişken sayısı arttıkça büyür ve çoğu alanı boş kalır. Genelleştirilmiş liste bunun yerine polinomu iç içe gruplar.

Sunumdaki örnek şu polinomdur:

$$P(x, y, z) = x^{10}y^3z^2 + 2x^8y^3z^2 + 3x^8y^2z^2 + x^4y^4z + 6x^3y^4z + 2yz$$

Polinom önce *z*'nin kuvvetlerine göre gruplanır. z^2 içeren üç terim ve *z* içeren üç terim vardır:

$$P = (x^{10}y^3 + 2x^8y^3 + 3x^8y^2)\,z^2 + (x^4y^4 + 6x^3y^4 + 2y)\,z$$

Parantez içindeki katsayılar artık yalnız *x* ve *y* içeren polinomlardır. Bunlar da *y*'ye göre gruplanır:

$$P = ((x^{10} + 2x^8)\,y^3 + 3x^8\,y^2)\,z^2 + ((x^4 + 6x^3)\,y^4 + 2y)\,z$$

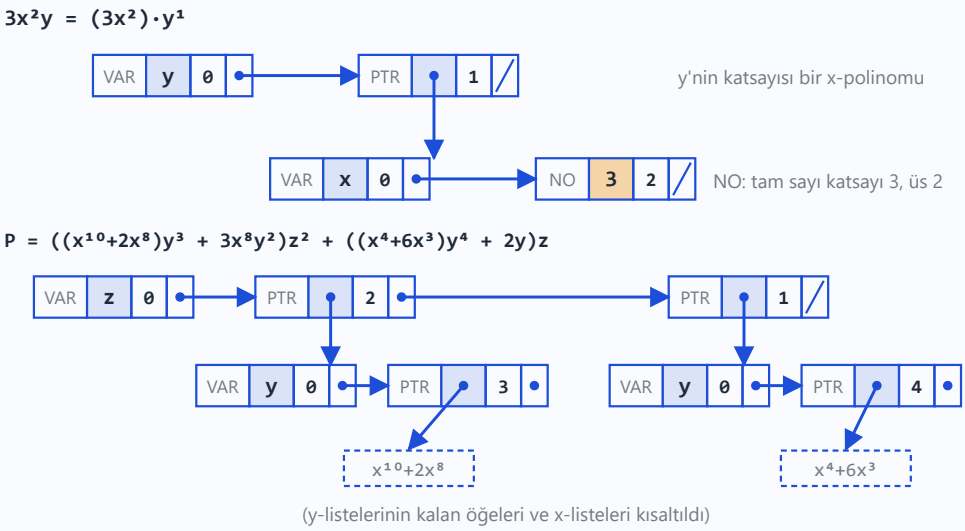
En içteki katsayılar yalnız *x* içerir ve sayısal katsayılı sıradan polinomlardır. Her düzey aynı biçimdedir: bir değişken ve o değişkenin kuvvetleriyle çarpılan katsayılar. Katsayı ya bir tam sayıdır ya da bir alt düzeyin polinomudur. Bu, genelleştirilmiş listenin tanımına birebir uyar. Sunum her öğeyi şu yapıyla tutar:

```c
typedef enum { VAR, PTR, NO } uclu_t;
typedef struct poge *poge_gost;
struct poge {
    uclu_t uclu;            // üçlü
    union {
        char deg;           // VAR: değişkenin adı
        poge_gost L_bag;    // PTR: katsayı bir liste
        int kat;            // NO : tamsayı katsayı
    };
    int us;                 // üs bilgisi
    poge_gost bag;
};
```

`uclu` alanı öğenin türünü söyler ve üç değer alır. `VAR` öğesi bir düzeyin ilk öğesidir ve o düzeyin değişken adını `deg`'de taşır; üssü 0'dır ve kullanılmaz. `PTR` öğesi bir terimdir ve katsayısı bir alt listedir; `L_bag` o alt listeyi gösterir. `NO` öğesi de bir terimdir, ama katsayısı `kat`'ta duran bir tam sayıdır. İki terim türünde de `us`, düzeyin değişkeninin üssüdür. `bag` aynı düzeydeki sonraki öğeyi gösterir.

Üç bilgi aynı anda gerekmez: bir öğe ya bir değişken adı ya bir alt liste ya da bir tam sayı taşır. Bu yüzden üç alan bir `union` içine konmuştur. `union`'ın bütün alanları aynı bellek adresinden başlar; birliğin boyutu en büyük alanınki kadardır. Hangi alanın geçerli olduğu `uclu`'dan anlaşılır. Bir türü gösteren alanla birlikte kullanılan bu düzene **etiketli union** (*tagged union*) denir. Etiket yanlış okunursa, örneğin bir `NO` öğesinin `L_bag` alanı izlenirse, bir tam sayı adres sanılır ve program çöker. Etiketi her erişimden önce denetlemek programcının sorumluluğudur.

Şekil 9.2 iki örneğin bellek görünümünü verir. Üstteki çizim $3x^2y$'dir. Bu terim $(3x^2)\cdot y^1$ diye okunur. Dış düzey y'nin düzeyidir ve tek terimi vardır: üssü 1 olan, katsayısı bir alt liste olan bir `PTR` öğesi. Alt liste x'in düzeyidir ve tek terimi, katsayısı 3, üssü 2 olan bir `NO` öğesidir.



Şekil 9.2: Çok değişkenli polinomların genelleştirilmiş liste gösterimi. Üstte $3x^2y$: `VAR y` düzeyinin tek terimi `PTR`, katsayısı `VAR x` ile başlayan alt listedir; orada `NO 3 2`, yani $3x^2$ durur. Altta $P(x, y, z)$'nin üst iki düzeyi: z düzeyinde üsleri 2 ve 1 olan iki terim, her birinin katsayısı bir y-polinomudur. En içteki x-polinomları kesik çizgili kutularla kısaltılmıştır.

Alttaki çizimde z düzeyi üç öğelidir: `VAR z` ve iki `PTR`. Birinci `PTR`'nin üssü 2'dir ve katsayısı $(x^{10} + 2x^8)\,y^3 + 3x^8\,y^2$ polinomudur. Bu polinomun listesi `VAR y` ile başlar ve üsleri 3 ve 2 olan iki `PTR` içerir. Çizim yer kazanmak için y-listelerinin yalnız ilk terimini ve x-polinomlarını kısaltılmış olarak gösterir. İkinci y-listesindeki $2y$ teriminin katsayısı yalnız 2 sayısıdır; bu terim için alt liste gerekmez ve bir `NO` öğesi yeter. Böylece bütün yapı 17 öğedir: z düzeyinde 3, iki y listesinde 3'er, üç x listesinde toplam 8. x86-64'te her öğe 32 bayttır: dört baytlık `uclu` ve `us` alanlarının her birinden sonra, 8 baytlık işaretçilerin hizası için dört baytlık boşluk kalır.

Böyle bir yapının değerini hesaplayan yordam, yapının kendisi kadar düzenlidir. Her düzeyde katsayı ya doğrudan okunur ya da alt listenin değeri olarak özyinelemeli hesaplanır; ardından değişkenin kuvvetiyle çarpılıp toplanır:

```
// P: VAR öğesiyle başlayan liste; d['x'-'x'], d['y'-'x'], d['z'-'x']: değişken değerleri
long pdeger(poge_gost P, const long d[]) {
    long top = 0, v = d[P->deg - 'x'];  // bu düzeyin değişkeninin değeri
    long kat, kuv;
    int k;
    for (poge_gost p = P->bag; p; p = p->bag) {
        kat = (p->uclu == PTR) ? pdeger(p->L_bag, d)   // katsayı bir liste
                               : p->kat;               // NO: tamsayı
        for (kuv = 1, k = 0; k < p->us; k++) kuv *= v;
        top += kat * kuv;
    }
    return top;
}
```

Fonksiyon `VAR` öğesinden değişkenin adını okur ve değerini `d` dizisinden alır; `'x'` 0., `'y'` 1., `'z'` 2. hücrededir. Sonra `bag` boyunca terimleri gezer. `PTR` teriminde katsayı, alt listenin aynı noktadaki değeridir. Örneğin $x = 2, y = 3, z = 1$ için en dış düzey z^2 ve z terimlerini toplar; $z = 1$ olduğundan sonuç iki y-polinomunun değerlerinin toplamıdır. Birinci y-polinomu $(2^{10} + 2 \cdot 2^8) \cdot 27 + 3 \cdot 2^8 \cdot 9 = 1536 \cdot 27 + 6912 = 48384$, ikincisi $(16 + 48) \cdot 81 + 2 \cdot 3 = 5190$ verir. Toplam 53574'tür. Bu değer polinomun açık hâliyle de aynıdır. Liste ayrıca $-3 \le x, y, z \le 3$ aralığındaki 343 noktanın hepsinde açık formülle karşılaştırılmış, hiçbir fark bulunmamıştır.

Bu gösterim tek seçenek değildir. Aynı polinom değişken sırası farklı seçilerek (önce x'e göre) ya da her terimi bir üs vektörüyle tutan düz bir listeyle de saklanabilir. Önemli olan, seçilen gösterime dönüşüm yordamlarının da yazılabilmesidir: metinden yapıya, yapıdan metne, bir gösterimden ötekine. Bir gösterim ancak bu yordamlarla birlikte bir çözüm sayılır.

9.4.3 Öğe yapısı ve bellek görünümü

Polinom örneği üç türlü öğe kullanır. Genel amaçlı bir genelleştirilmiş listede iki tür yeter: atom ve alt liste. Notun geri kalanındaki bütün yordamlar şu yapıyı kullanır:

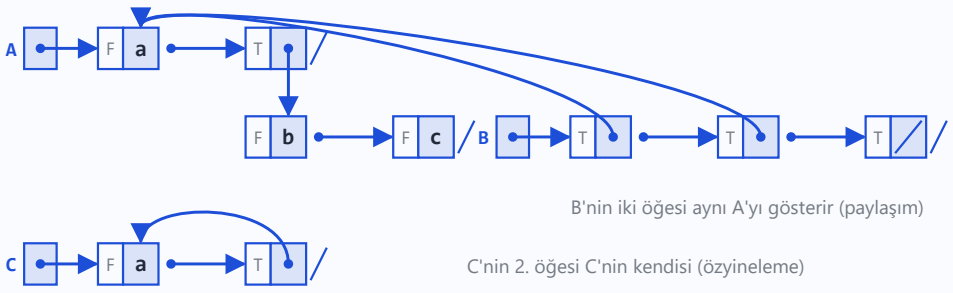
```
typedef enum { FALSE, TRUE } boolean;
typedef struct oge *oge_gost;
struct oge {
    boolean isaret;       // FALSE: atom
    union {
        char veri;        // atomik değer
        oge_gost L_bag;   // alt listenin başı
    };
    oge_gost bag;         // aynı düzeydeki sonraki
};
```

`isaret` alanı etikettir. `FALSE` ise öğe bir atomdur ve değeri `veri` 'dedir. `TRUE` ise öğe bir alt listedir ve `L_bag` alt listenin ilk öğesini gösterir. `bag` her iki durumda da aynı düzeydeki bir sonraki öğeyi gösterir. Kısaca bir öğe *[isaret | veri/L_bag | bag]* biçimindedir. Çizimlerde atom öğesi F, alt liste öğesi T harfiyle gösterilir.

Yapı kendi tipini kullandığı için önce `oge_gost` işaretçi tipi adlandırılır; `struct oge` o anda henüz eksik bir tiptir, ama ona işaretçi tanımlamak serbesttir. `boolean` tipi, sunumun kendi tanımladığı iki değerli bir sayımdır (*enum*): `FALSE` 0, `TRUE` 1 değerini alır. C99'dan beri gelen `<stdbool.h>` 'deki `bool` ile karıştırılmamalıdır. İkisi aynı programda yan yana kullanılabilir, çünkü adları farklıdır: `true` küçük, `TRUE` büyük harflidir.

`union` 'a bir ad verilmemiştir. **Adsız birlik** (*anonymous union*) C11 ile standarda girmiştir ve alanlarına doğrudan `L->veri`, `L->L_bag` diye erişilir. C99 derleyicisinde birliğe bir ad vermek gerekir, örneğin `u`; o zaman erişim `L->u.veri` ve `L->u.L_bag` biçiminde yazılır. Notun bütün kodları `-std=c11` ile uyarısız derlenmiştir.

Birlik bellekten tasarruf sağlar. `char` 1, işaretçi 8 bayttır ve birlik 8 bayt tutar. Etiket 4, birlik 8, bağ 8 bayttır; işaretçiler 8'in katı adreslerde durmak zorunda olduğu için etiketten sonra 4 bayt boşluk kalır. x86-64'te `sizeof(struct oge)` 24 bayttır. Birlik olmasaydı, yani `veri` ve `L_bag` ayrı alanlar olsaydı, öğe 32 bayta çıkardı.



Şekil 9.3: *A = (a, (b, c))*, *B = (A, A, ())* ve *C = (a, C)*'nin bellek görünümü. Her kutu bir öğedir: sol hücre etiket (F atom, T alt liste), orta hücre değer ya da alt liste bağı, sağ hücre aynı düzeydeki sonraki öğenin bağı. *B*'nin ilk iki öğesi aynı *A*'yı gösterir; üçüncüsünün alt liste bağı boştur. *C*'nin ikinci öğesi *C*'nin kendi başını gösterir.

Şekil 9.3 bu üç listeyi bellekte gösterir. *A* iki öğelidir: *a* atomu ve bir T öğesi. T öğesinin orta hücresinden çıkan ok, *(b, c)* alt listesinin ilk öğesine iner. *b* ve *c* aynı satırda, birbirine `bag` ile bağlıdır. Aynı düzeydeki öğeler hep bir satır boyunca izlenir; bir alt listeye inmek bir satır aşağı inmek demektir. Bu çizim kuralı, iç içe yapıları okumayı çok kolaylaştırır.

B'nin üç T öğesi vardır. İlk ikisinin oku aynı yere, *A*'nın ilk öğesine gider. Üçüncüsünün orta hücresinde eğik çizgi, yani `NULL` vardır: bu öğe boş bir alt listedir. Boş liste ile "öğe yok" arasındaki fark burada görünür. *()* öğesi var olan bir T öğesidir ve *B*'nin uzunluğunu bir artırır; yalnız gösterdiği liste boştur. *C*'de ikinci öğenin oku geri döner ve listenin ilk öğesine ulaşır. Bu ok izlenerek yapılan her gezinme, sonsuza kadar aynı iki öğe arasında döner.

> **Sık yapılan hata**
>
> Etiketi denetlemeden birliğin bir alanını okumak. `if (L->L_bag)` yazmak, atom öğesinde de doğru çıkabilir: `veri` alanına yazılan `'a'` karakteri, birliğin ilk baytını 97 yapar ve bu baytı içeren 8 baytlık değer sıfırdan farklıdır. Önce `L->isaret`'e bakılmalı, birliğin hangi alanı geçerliyse yalnız o okunmalıdır.

9.4.4 Neden genelleştirilmiş liste?

Genelleştirilmiş listenin üç yararı öne çıkar. Birincisi yerdir. Bir ağacın bütün yapısı, örneğin *A(B(E(K, L), F), C(G), D(H(M), I, J))* gibi tek satırlık bir metinle yazılabilir. Bu metin, yapıyı tekrar kurmaya yetecek bütün bilgiyi taşır. Ağacın resmi saklanabilir, ya da bellekteki görüntüsü olduğu gibi diske dökülebilir. İkisi de çalışır, ama ikisi de metinden çok daha fazla yer tutar. Metin bir tür sıkıştırmadır. Bu kadar küçük bir verinin saklanmasında bile bilgisayar mühendisi bu farkı görmelidir.

İkincisi doğallıktır. Dosya sistemindeki dizinler, açılır menüler, XML ve JSON belgelerinin iç içe öğeleri, LISP ifadeleri zaten hiyerarşiktir. Bir JSON belgesindeki `[1, [2, 3], []]` dizisi, *(1, (2, 3), ())* genelleştirilmiş listesinin ta kendisidir.

Üçüncüsü algoritmalarla ilgilidir. Veri iyi bilinen bir yapıya taşındığında, o yapı için yazılmış bütün yordamlar kullanılabilir hâle gelir. Mühendislikte bu fikir sık görülür. Sinyal işlemede bir sinyal Fourier ya da Z dönüşümüyle başka bir uzaya taşınır, işlem orada yapılır ve sonuç geri dönüştürülür. Burada da ağaç gibi bir veri listeye taşınır; kopyalama, karşılaştırma, gezinme ve yazdırma gibi liste yordamları ondan sonra hazırdır.

9.4.5 Doğrusal formdan liste kurmak

Genelleştirilmiş listeyi elle, öğe öğe `malloc` ederek kurmak hem uzun hem hataya açıktır. *(a, (b, c))* gibi bir metne listenin **doğrusal formu** denir. Doğrusal formu okuyup listeyi kuran bir yordam, hem programların veriyi dışarıdan almasını sağlar hem de yordamları denemeyi kolaylaştırır. Notta izlenen bütün örnek listeler bu yordamla kurulmuştur. Yordam, her öğe için bir öğe ayıran şu yardımcıyı kullanır:

```
oge_gost ogeal(void) {
    oge_gost p = malloc(sizeof *p);
    if (p == NULL) {                    // bellek kalmadı
        fprintf(stderr, "ogeal: bellek yetersiz\n");
        exit(EXIT_FAILURE);
    }
    return p;                           // alanlar ilklendirilmedi
}
```

`ogeal`, `malloc(sizeof *p)`'nin `NULL` denetimli bir sarmalayıcısıdır. `sizeof *p` yazımı, `p`'nin gösterdiği tipin boyutunu verir; tip adını tekrar yazmaya gerek bırakmaz. Bellek kalmamışsa program bir hata iletisiyle durur. Bu kaba ama dürüst bir seçimdir: yordamların her birinde `NULL` denetimi yapmak yerine hata tek bir yerde ele alınır. Döndürülen öğenin alanları ilklendirilmemiştir; her alanı çağıran doldurmalıdır.

Metin karakter karakter okunur. Dilbilgisi basittir: bir liste `(` ile başlar, öğeleri virgülle ayrılır ve `)` ile biter. Her öğe ya tek bir harftir (atom) ya da yine bir listedir. Dilbilgisi özyinelemeli olduğu için okuyucu da özyinelemelidir:

```
// *s, '(' karakterini gösterirken çağrılır; liste kurulunca
// *s, eşleşen ')' karakterinin bir sonrasını gösterir.
oge_gost ayristir(const char **s) {
    oge_gost bas = NULL, *son = &bas, q;
    (*s)++;                             // '(' atlanır
    while (**s != ')') {
        q = ogeal();
        if (**s == '(') {               // alt liste
            q->isaret = TRUE;
            q->L_bag = ayristir(s);
        } else {                        // atom
            q->isaret = FALSE;
            q->veri = **s;
            (*s)++;
        }
        q->bag = NULL;
        *son = q;                       // sona bağla
        son = &q->bag;
        if (**s == ',') (*s)++;         // ayırıcı atlanır
    }
    (*s)++;                             // ')' atlanır
    return bas;
}
```

Fonksiyon metindeki konumu `const char **s` ile alır. Konumun adresi verilir, çünkü bir alt liste okunduktan sonra çağıran, okumayı alt listenin bittiği yerden sürdürmelidir. İç çağrı `*s`'yi ilerletir ve bu ilerleme dışarıdan da görünür. Bu, `bl_ekle`'deki işaretçiye işaretçi fikrinin bir başka kullanımıdır.

Döngü `)` görülene kadar sürer. Her turda bir öğe ayrılır. Sıradaki karakter `(` ise öğe bir alt listedir ve `L_bag` alanı özyinelemeli çağrının kurduğu listeyle doldurulur. Değilse karakter bir atomdur. Öğe sona bağlanır. Bunun için `son` değişkeni, son öğenin `bag` alanının adresini tutar; boş listede ise `bas`'ın adresini. `*son = q` ilk öğede `bas`'ı, sonrakilerde bir önceki öğenin bağını doldurur. Yine iki durum tek satıra inmiştir. Öğeden sonra virgül varsa atlanır. Döngüden çıkınca kapanış parantezi atlanır ve liste döndürülür.

Okuyucu hatalı metni denetlemez. Kapanmayan bir parantez, `'\0'` karakterinden sonrasını okumaya yol açar. Gerçek bir programda her karakterin beklenen karakter olup olmadığı denetlenmeli, hata durumunda o ana kadar ayrılan öğeler bırakılmalıdır. Not boyunca okuyucu yalnız doğru metinlerle kullanılmıştır. Ayrıca listeyi metne geri çeviren `dogrusal_yaz` yordamıyla birlikte 20.000 rastgele metin üzerinde denenmiştir: her metin okunup geri yazıldığında aynı metin elde edilmiştir.

Örnek 9.2

`"(a,(b,c))"` okunsun. Dış çağrı `(`'yi atlar ve `a`'yı görür: bir atom öğesi kurulur, `bas` ona bağlanır, virgül atlanır. Sıradaki karakter `(`'dir. Yeni bir öğe ayrılır, `isaret = TRUE` yapılır ve iç çağrı başlar. İç çağrı `b` ve `c` atomlarını kurar, `)`'yi görünce durur, bu parantezi atlar ve *(b, c)*'nin ilk öğesini döndürür. Dış çağrı bu adresi T öğesinin `L_bag`'ine yazar ve öğeyi *a*'nın arkasına bağlar. Sırada dış listenin `)`'si vardır; dış döngü de biter. Dört öğe ayrılmış, Şekil 9.3'teki *A* kurulmuştur.

9.5 Özyinelemeli temel yordamlar

Genelleştirilmiş liste üzerinde çalışan yordamların çoğu aynı iskeleti izler. Bir öğeye gelindiğinde önce türüne bakılır. Öğe bir atomsa işlenir: kopyalanır, karşılaştırılır ya da yazdırılır. Öğe bir alt listeyse, aynı yordam o alt liste için çağrılır. Ardından `bag` ile aynı düzeydeki sonraki öğeye geçilir; bu geçiş de ya bir döngüyle ya da yine özyinelemeli bir çağrıyla yapılır. Kısaca: *atomsa işle, alt listeyse in, sonra devam et*. Bu kesimdeki dört yordam bu kalıbın dört farklı kullanımıdır.

Özyinelemeli bir yordamı yazmanın ve okumanın yolu, her çağrıya küçük bir sözleşme gözüyle bakmaktır. Sözleşme "Bu çağrıya şu liste verilirse şunu döndürür." biçimindedir. Yordam yazılırken iç çağrıların bu sözleşmeye uyduğu varsayılır ve yalnız bu çağrının kendi işi düşünülür. Bu varsayım, matematikteki tümevarımın programdaki karşılığıdır. Taban durumu (boş liste) doğru ele alınmışsa ve her çağrı daha küçük bir listeyle yapılıyorsa, yordam bütün sonlu listeler için doğrudur.

9.5.1 Kopyalama

Sunumdaki ilk soru, paylaşımsız ve özyinelemesiz bir listenin kopyasını çıkaran yordamı ister. Kopya, aynı yapıda ama tamamen yeni öğelerden oluşan bir liste olmalıdır: kopyada yapılan bir değişiklik asıl listeyi etkilememelidir.

```
oge_gost kopyala(oge_gost L) {
    oge_gost q = NULL;
    if (L) {                    // L NULL değil mi
        q = ogeal();            // bir öğe al
        q->isaret = L->isaret;
        if (!L->isaret) q->veri = L->veri; // atom
        else q->L_bag = kopyala(L->L_bag);
        q->bag = kopyala(L->bag);
    }
    return q;
}
```

Yordamın sözleşmesi şudur: `kopyala(L)`, `L`'nin gösterdiği listenin, yani `L` öğesinin ve ondan `bag` ile ulaşılan bütün öğelerin bağımsız bir kopyasının ilk öğesini döndürür. Boş liste için `NULL` döndürür.

İlk satırda `q`, `NULL` ile ilklendirilir. `L` boşsa `if` bloğu atlanır ve bu `NULL` döner: boş listenin kopyası boş listedir. Taban durumu bu kadardır. `L` boş değilse `ogeal` ile yeni bir öğe alınır ve etiket kopyalanır. Atom öğesinde değer olduğu gibi kopyalanır. Alt liste öğesinde ise `L_bag` kopyalanmaz; kopyalansaydı yeni öğe, eski alt listeyi gösterirdi ve kopya ile asıl liste aynı alt listeyi paylaşırdı. Bunun yerine alt listenin kendi kopyası istenir ve yeni öğe onu gösterir. Son satır listenin kalanını, yani `L->bag`'den başlayan listeyi kopyalar ve yeni öğeyi bu kopyaya bağlar.

Yordamın iki özyinelemeli çağrısı vardır ve ikisi de daha küçük bir listeyle yapılır. `L->L_bag`'deki alt liste ve `L->bag`'den başlayan kalan liste, `L`'nin öğelerinin bir kısmını içerir. Bu yüzden çağrılar sonunda boş listeye ulaşır ve durur. Bu akıl yürütme, listenin özyinelemesiz olduğu varsayımına dayanır. *C = (a, C)* gibi bir listede alt liste, listenin kendisidir ve küçülmez.

Yeni öğe ayrıldığı anda alanları çöp değer taşır. Kod her alanı mutlaka doldurur: `isaret` hemen, `veri` ya da `L_bag` türe göre, `bag` en sonda. `bag` alanına değer, ancak kalan listenin kopyası tamamlandığında yazılabilir. Öğeler baştan sona doğru *alınır*, bağlar ise sondan başa doğru *güncellenir*. Bu sıra, bu kesimdeki bütün özyinelemeli yordamlarda görülür. Özyinelemeli düşünmekte zorlanan bir öğrenci aynı işi bir kuyrukla yapmayı önerebilir. Yapılabilir, ama kod uzar ve alt listelerin kopyalarını doğru yere bağlamak için ayrıca kayıt tutmak gerekir. Özyinelemeli sürüm, yapının tanımını doğrudan koda çevirir.

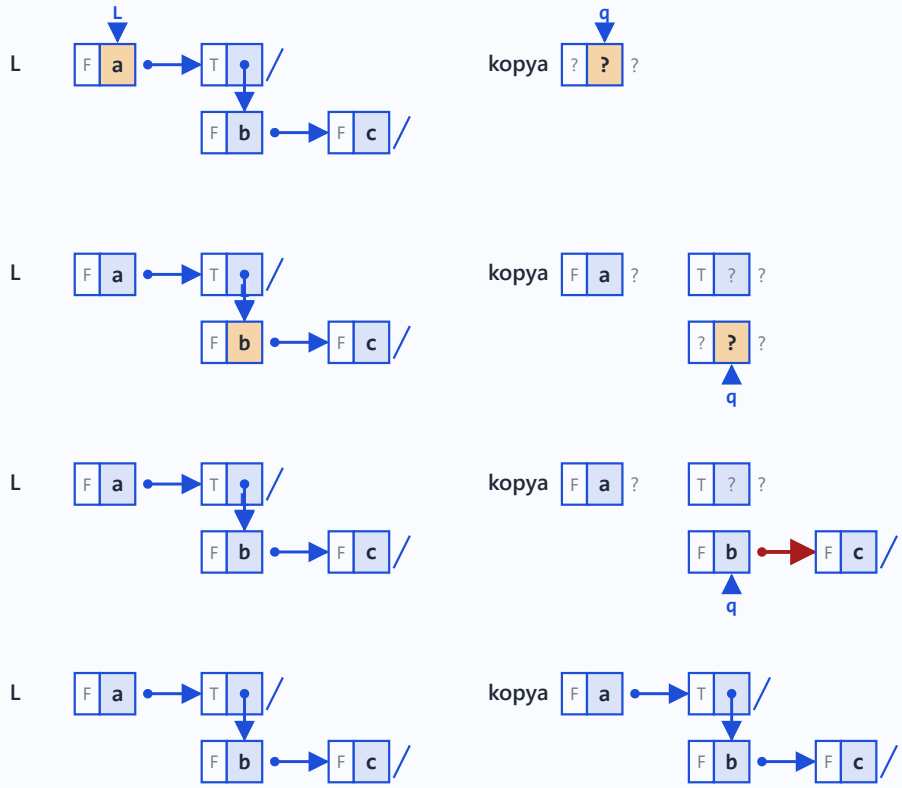
9.5.2 Kod izleme: `kopyala(L)`

L = (a, (b, c)) kopyalansın. Sunumun 20 adımlık izlemesi Tablo 9.5'te dokuz satıra toplanmıştır. Yığın sütunu, o anda çalışan çağrıyı en sağda olacak biçimde çağrı yığınını gösterir. *k(x)*, *x* öğesinin kopyası demektir.

Adım	Yığın (sağdaki çalışıyor)	İşlem
1	kopyala(a)	`ogeal` : *k(a)* ayrıldı; `isaret = FALSE`, `veri = 'a'`. `q->bag = kopyala(L->bag)` çağrısı başlar.
2	kopyala(a) · kopyala((b,c))	*k((b, c))* ayrıldı; `isaret = TRUE`. Önce alt liste: `q->L_bag = kopyala(L->L_bag)`.
3	… · kopyala((b,c)) · kopyala(b)	*k(b)* ayrıldı; `veri = 'b'`. Kalan için çağrı.

Adım	Yığın (sağdaki çalışıyor)	İşlem
4	… · kopyala(b) · kopyala(c)	*k(c)* ayrıldı; `veri = 'c'`. Kalan için çağrı.
5	… · kopyala(c) · kopyala(NULL)	`L == NULL`: hiçbir şey ayırmadan `NULL` döner. *k(c)* `->bag = NULL`; `kopyala(c)`, *k(c)*'yi döndürür.
6	… · kopyala((b,c)) · kopyala(b)	*k(b)* `->bag` ← *k(c)*; *k(b)* döner.
7	kopyala(a) · kopyala((b,c))	*k((b, c))* `->L_bag` ← *k(b)*. Şimdi kalan için çağrı.
8	… · kopyala((b,c)) · kopyala(NULL)	`NULL` döner; *k((b, c))* `->bag = NULL`; *k((b, c))* döner.
9	kopyala(a)	*k(a)* `->bag` ← *k((b, c))*; *k(a)* döner. Kopya tamamdır.

Tablo 9.5: `kopyala(L)` izlemesi, *L = (a, (b, c))*.



Şekil 9.4: `kopyala(L)` sırasında asıl liste (solda) ve kopya (sağda). Yukarıdan aşağıya: *a*'nın kopyası ayrılmış, bağı henüz çöp (?); *b*'nin kopyası ayrılmış, üstteki T öğesinin alt liste bağı henüz boş; dönüşte *k(b)*'nin bağı *k(c)*'ye yazılmış; kopya tamamlanmış.

İzlemenin en önemli gözlemi, çağrı yığınının en çok dört çağrı derinliğine inmesidir. Yol şudur: *a*'dan *(b, c)*'ye `bag` ile, oradan *b*'ye `L_bag` ile, oradan *c*'ye yine `bag` ile inilir, sonra boş liste gelir. Yığın derinliği yalnız iç içeliğe değil, aynı düzeydeki öğe sayısına da bağlıdır, çünkü `bag` boyunca ilerlemek de bir çağrıdır. Bu ayrıntı uzun listelerde önem kazanır.

Bu çağrılar sayılarak yordamın maliyeti bulunur. *(a, (b, c))* için `kopyala` altı kez çağrılır: dört öğe için birer kez ve `NULL` için iki kez. Bu sayı rastlantı değildir.

Teorem 9.2 (Çağrı sayısı)

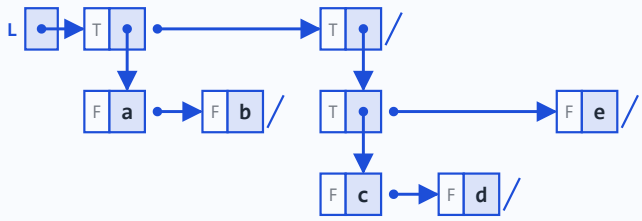
Paylaşımsız ve özyinelemesiz bir genelleştirilmiş listede toplam *n* öğe bulunsun ve bunların *s* tanesi alt liste öğesi olsun. `kopyala` bu liste için tam *1 + n + s* kez çağrılır. Çağrıların *n* tanesi bir öğeyle, *1 + s* tanesi `NULL` ile yapılır. Yordamın süresi bu yüzden *Θ(n)*'dir.

Kanıt. Her öğe tam bir kez bir çağrının parametresi olur. En üst düzeyin ilk öğesi dışarıdan yapılan çağrıyla, öteki öğeler ise kendisinden önceki öğenin `kopyala(L->bag)` çağrısıyla ya da bulunduğu alt listenin sahibi olan T öğesinin `kopyala(L->L_bag)` çağrısıyla gelir. Paylaşım olmadığı için hiçbir öğeye iki yoldan ulaşılmaz. Öğeyle yapılan her çağrı, `bag` için bir çağrı yapar; T öğeleri ayrıca `L_bag` için bir çağrı yapar. Böylece dışarıdan gelen çağrıya ek olarak $n + s$ çağrı daha oluşur ve toplam $1 + n + s$'dir. Bunlardan n'si öğelere gittiğine göre kalan $1 + s$ çağrı `NULL` ile yapılmıştır. Her çağrı, iç çağrılar dışında sabit iş yapar. $s \le n$ olduğundan toplam iş $\Theta(n)$'dir. ■

Teorem bir denemeyle de doğrulanmıştır. Sayaç eklenmiş bir kopyayla *(a, (b, (c)))* için 8 ($n = 5, s = 2$), *((a, b), ((c, d), e))* için 12 ($n = 8, s = 3$), boş liste için 1 çağrı sayılmıştır. Kopyalama ayrıca n kez `malloc` çağırır ve kopyanın kendisi için $\Theta(n)$ bellek ayırır. Yığın için gereken ek bellek ise en derin çağrı zincirinin uzunluğu kadardır.

9.5.3 Kopyalamanın sınırları ve listeyi bırakmak

Sunumun önerdiği test listesi *L = ((a, b), ((c, d), e))*'dir. Bu liste üç düzeylidir, ilk öğesi bir atom değildir ve bir alt listenin ilk öğesi de alt listedir. Bu yüzden basit örneklerde görünmeyen hataları yakalar. Şekil 9.5 listenin bellek görünümünü verir. Deneme bu liste için 8 `malloc` yapıldığını, kopyanın asıl listeye eşit olduğunu ve iki liste arasında ortak hiçbir öğe bulunmadığını göstermiştir.



Şekil 9.5: Test listesi *L = ((a, b), ((c, d), e))*. En üst düzeyde iki T öğesi vardır. İkinci T'nin alt listesi yine bir T öğesiyle başlar; *(c, d)* en alt düzeydedir.

Yordamın iki sınırı vardır. Birincisi paylaşımdır. *B = (A, A, ())* kopyalandığında `kopyala`, *B*'nin ilk iki öğesinin aynı listeyi gösterdiğini bilmez. Her T öğesi için ayrı bir `kopyala(L->L_bag)` çağrısı yapılır ve *A* iki kez kopyalanır. *A = (a, (b, c))* alındığında deneme 11 `malloc` saymıştır: *B*'nin üç öğesi ve *A*'nın iki kopyası için 4'er öğe. Kopyanın değeri asıl listeye eşittir, ama yapısı farklıdır. Asıl listede ilk iki öğe aynı listeyi gösterirken, kopyada iki ayrı liste gösterir. Kopyada *A*'nın birinci kopyasına yapılan bir değişiklik ikinciye yansımaz. Paylaşım kaybolmuştur. Paylaşımı koruyan bir kopyalama, daha önce kopyalanmış alt listeleri bir tabloda tutar; böyle bir sürüm alıştırma çözümleri arasında yer alır.

İkinci sınır özyinelemedir. *C = (a, C)* için `kopyala(C)` önce *a*'yı kopyalar, sonra ikinci öğe için alt listenin kopyasını ister. Alt liste *C*'nin kendisidir; aynı çağrı baştan başlar. Çağrılar hiç bitmez. Her çağrı yığında yer tuttuğu için bir süre sonra çağrı yığını taşar ve işletim sistemi programı durdurur. Bu duruma **yığın taşması** (*stack overflow*) denir. Windows'ta yapılan denemede program `0xC00000FD` (STATUS_STACK_OVERFLOW) koduyla sonlanmıştır.

Kopyalanan bir liste işi bitince bırakılmalıdır. Paylaşımsız ve özyinelemesiz bir listeyi bırakan yordam, kopyalamanın tersidir:

```
// Paylaşımsız, özyinelemesiz listenin bütün öğelerini bırakır.
void liste_bosalt(oge_gost L) {
    oge_gost q;
    while (L) {
        if (L->isaret)
            liste_bosalt(L->L_bag);     // önce alt liste
        q = L;
        L = L->bag;                     // bağ, free'den önce okunur
        free(q);
    }
}
```

Yordam aynı düzeyi bir döngüyle gezer; yalnız alt listeler için kendini çağırır. Her öğede önce alt liste bırakılır. Bir sonraki öğenin adresi `free`'den önce okunur; iade edilmiş bir öğenin `bag` alanı artık okunamaz. Notun bütün denemeleri bu yordamla temizlenmiş, sayaçlı `malloc` ve `free` sarmalayıcılarıyla sızıntı olmadığı denetlenmiştir. Ana doğrulama programında 216.150 `malloc` ve aynı sayıda `free` sayılmıştır.

> **Sık yapılan hata**
>
> Paylaşımlı bir listeyi `liste_bosalt` ile bırakmak. *B = (A, A, ())* bırakılırken *A* önce birinci T öğesi için iade edilir. İkinci T öğesi aynı adresleri bir kez daha `free`'ye verir. Aynı bloğu iki kez iade etmek tanımsız davranıştır ve heap'i bozar. Paylaşımlı listeleri güvenle bırakmanın yolu referans saymadır.

9.5.4 Eşitlik

İki liste ne zaman eşittir? Yapıları aynı olduğunda ve karşılıklı atomları aynı değeri taşıdığında. Yapının aynı olması, iki listenin aynı uzunlukta olması ve her konumda ikisinde de ya atom ya da alt liste bulunması demektir. Alt listeler için aynı tanım özyinelemeli olarak uygulanır. Sunumun sorusu bu tanımı paylaşımsız ve özyinelemesiz listeler için koda çevirmeyi ister:

```
boolean esitmi(oge_gost T, oge_gost L) {
    boolean x = TRUE;
    if (!T && !L) return TRUE;
    if (T && L && T->isaret == L->isaret) {
        if (!T->isaret)         // atomik mi?
            x = (T->veri == L->veri) ? TRUE : FALSE;
        else                    // alt listeler
            x = esitmi(T->L_bag, L->L_bag);
        if (x == TRUE) return esitmi(T->bag, L->bag);
    }
    return FALSE;
}
```

İlk koşul taban durumudur: iki liste de boşsa eşittirler. İkinci koşul, iki öğenin de var olduğunu ve etiketlerinin aynı olduğunu denetler. İkisinden biri boşsa ya da biri atom, öteki alt listeyse bu koşul yanlış olur, blok atlanır ve fonksiyon `FALSE` döndürür. Blok içinde önce bu öğeler karşılaştırılır. Atomlarda `veri` alanları, alt listelerde alt listelerin kendisi özyinelemeli olarak karşılaştırılır. Sonuç `x`'tedir. `x` doğruysa cevap kalan listelerin eşitliğine bağlıdır ve o çağrının sonucu olduğu gibi döndürülür. `x` yanlışsa blok sona erer ve yine `FALSE` döner.

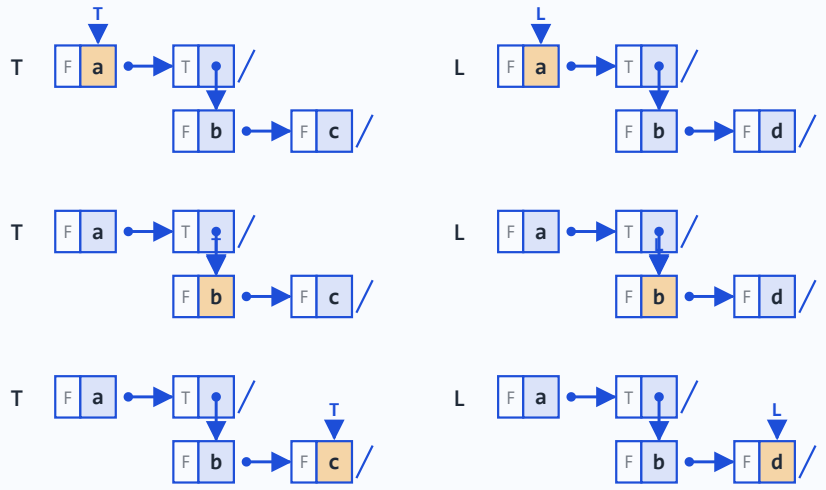
Kod doğru çalışır ama sade değildir. `x` değişkeni gereksizdir; `(T->veri == L->veri) ? TRUE : FALSE` yazımı, karşılaştırmanın zaten 0 ya da 1 verdiği düşünülürse uzundur. Üç ayrı `return` akışı izlemeyi zorlaştırır. Daha nitelikli bir sürüm de yazılabilir. Bir de her iki sürüm için geçerli bir kısıt vardır: *C = (a, C)* gibi özyinelemeli iki liste karşılaştırılamaz. İkinci öğede alt listeler karşılaştırılırken aynı çağrı yeniden başlar ve kopyalamadaki gibi yığın taşar.

9.5.5 Kod izleme: `esitmi(T, L)`

T = (a, (b, c)) ve *L = (a, (b, d))* karşılaştırılsın. Fark en derindeki son atomdadır. Tablo 9.6 karşılaştırmanın nasıl derine inip sonucu yukarı taşıdığını gösterir.

Çağrı	Karşılaştırılan	İşlem	Sonuç
1	*a* / *a*	ikisi de var, ikisi de atom; `'a' == 'a'`, `x = TRUE`; kalanlar için 2. çağrı	2.'nin sonucu
2	*(b, c)* / *(b, d)*	ikisi de T; önce alt listeler: 3. çağrı	`x` = 3.'nün sonucu
3	*b* / *b*	atom; `'b' == 'b'`, `x = TRUE`; kalanlar için 4. çağrı	4.'nün sonucu
4	*c* / *d*	atom; `'c' == 'd'` yanlış, `x = FALSE`; kalanlara bakılmaz	`FALSE`
3	4.'den gelen `FALSE` aynen döndürülür		`FALSE`
2	`x = FALSE`; blok sona erer, `return FALSE`		`FALSE`
1	2.'den gelen `FALSE` aynen döndürülür		`FALSE`

Tablo 9.6: `esitmi(T, L)` izlemesi, *T = (a, (b, c))*, *L = (a, (b, d))*.



Şekil 9.6: `esitmi(T, L)` sırasında karşılaştırılan öğe çiftleri. Yukarıdan aşağıya: ilk atomlar *a*/*a*; alt listelerin ilk atomları *b*/*b*; farkın bulunduğu *c*/*d*.

Fonksiyon toplam dört kez çağrılmıştır. Bu sayı, sayaç eklenmiş bir kopyayla da ölçülmüştür. İlk farkta durmanın etkisi buradadır: 2. çağrı, alt listeler eşit çıkmadığı için *T* ve *L*'nin kalan öğelerine hiç bakmaz. İki liste eşitse bütün öğeler gezilir ve çağrı sayısı `kopyala`'daki gibi *1 + n + s* olur. Süre en kötü durumda iki listeden kısa olanın boyutuyla doğrusaldır: *O(n)*. Deneme ayrıca şu durumları da doğrulamıştır: *(a, (b, c))* kendisinin ayrı kurulmuş bir eşiyle eşittir; *(a, (b, c), d)* ile eşit değildir (uzunluk farkı); *(a, b, c)* ile eşit değildir (ikinci öğede tür farkı); iki boş liste eşittir.

9.5.6 Daha kısa sürüm ve paylaşımlı listeler

```
bool esit(oge_gost T, oge_gost L) {
    if (T == L) return true;        // aynı düğüm/ikisi NULL
    if (!T || !L || T->isaret != L->isaret) return false;
    if (T->isaret ? !esit(T->L_bag, L->L_bag)
                  : T->veri != L->veri)
        return false;
    return esit(T->bag, L->bag);
}
```

Bu sürüm C'nin `bool` tipini kullanır ve her koşul bir satırdır. İlk satır yeni bir kısayol taşır: `T == L` ise iki işaretçi aynı öğeyi gösteriyordur ya da ikisi de `NULL`'dır. İki durumda da listeler eşittir ve içlerine inmeye gerek yoktur. İkinci satır ilk farkları eler: biri boşsa ya da etiketler farklıysa sonuç yanlıştır. Üçüncü satır öğelerin içeriğini karşılaştırır. Koşul işleci burada değer değil koşul seçer: T öğesinde alt listelerin eşit olmaması, atom öğesinde değerlerin farklı olması yanlış sonuç verir. Son satır kalanlara geçer.

Son satırın bir özelliği daha vardır. `return esit(T->bag, L->bag);` çağrısı fonksiyonun yaptığı son iştir; dönüşünden sonra yapılacak bir hesap kalmaz. Böyle çağrılara **kuyruk çağrısı** denir. Eniyileme açıkken derleyiciler kuyruk çağrısını bir sıçramaya çevirebilir ve `bag` boyunca ilerlemek yığında yer tutmaz. C standardı bunu garanti etmez; ama yazım buna izin verir.

Kısayolun asıl değeri paylaşımlı listelerde görülür. Özyinelemesiz ama paylaşımlı listelerde iki yordam da doğru sonuç verir. Paylaşım eşitliği bozmaz; yalnız aynı alt liste birden çok kez gezilir. `esit`, iki tarafta da aynı adresi gördüğü yerde içeri inmez. Örneğin *B* kendisiyle karşılaştırıldığında ilk satır hemen doğru döndürür. *B* paylaşımı koruyan bir kopyasıyla karşılaştırılırken de kopyanın içinde paylaşılan alt liste bir kez gezilir.

Programın direnç noktası, onu kırmaya çalışan girdilerle bulunur. “Tuşa basıldı, çalıştı” yeterli bir sınama değildir. Bir yordamın hangi listelerde doğru, hangilerinde yanlış olduğunu bilmek, onu verilen bir örnekte çalıştırmaktan çok daha geniş bir bilgi ister. Bir kanıt yazmak bu bilgiyi açıkça ortaya koymaktır. `esitmi` için bu bilgi şudur: paylaşımsız ve özyinelemesiz listelerde doğru; paylaşımlı listelerde doğru ama gereksiz iş yapar; özyinelemeli listelerde sonlanmaz.

9.5.7 Gezip yazdırma ve hatalı bir yazım

Üçüncü soru, listeyi gezip atomları sırayla yazan yordamı ister. Sınavlarda sık görülen hatalı bir yazım şudur:

```
void dolas_yaz( oge_gost L ){
    if(L)
        if(!L->isaret)
            printf("%c", L->veri);
        else
            dolas(L->L_bag);
        dolas( L->bag );
}
```

Bu kodda iki hata vardır. Birincisi addır: yordamın adı `dolas_yaz`, ama özyinelemeli çağrılar `dolas`'ı çağırır. Böyle bir fonksiyon tanımlı değildir. C99'dan beri bildirilmemiş bir fonksiyonu çağırmak hatadır. Denemede derleyici "call to undeclared function 'dolas'" hatasıyla durmuştur.

İkinci hata daha sinsidir ve ad düzeltilince ortaya çıkar. `if (L)`'den sonra süslü parantez yoktur. Süslü parantez olmayan bir `if` yalnız kendisinden sonraki *tek* deyimi kapsar. Burada o deyim, iç `if-else`'in tamamıdır. Son satırdaki `dolas( L->bag );` çağrısı ise `if (L)`'nin dışındadır; girintisi onu içeride gösterse de C girintiye bakmaz. Bu satır `L == NULL` iken de çalışır ve `NULL->bag` okunur. Derleyici buna iki uyarıyla işaret eder: "misleading indentation" (yanıltıcı girinti) ve "add explicit braces to avoid dangling else". Bu uyarılar `-Wall` ile görünür; `-Wall` açmadan derlemek bu yüzden kendine zarar vermektir.

Ad hatası düzeltilip kod *(a, (b), c)* ile çalıştırıldığında ekrana `ab` yazılmış, ardından program çökmüştür. *b* yazıldıktan sonra `dolas_yaz(b->bag)`, yani `dolas_yaz(NULL)` çağrılmış; bu çağrının son satırı `NULL` işaretçiden `bag` okumaya çalışmıştır. UBSan bu erişimi "member access within null pointer of type 'struct oge'" iletisiyle yakalamıştır. Derleyici ayrıca "all paths through this function will call itself" uyarısını da vermiştir. Bu uyarıya göre yordamın her yolu kendini en az bir kez çağırır, yani hiçbir çağrı özyinelemesiz bitemez. Bu uyarı, taban durumunun eksik olduğunu doğrudan söyler. Doğru yazım iki çağrıyı da `if (L) { … }` içine alır:

```
void dolas_yaz(oge_gost L) {
    if (L) {
        if (!L->isaret)
            printf("%c", L->veri); // atomikse yaz
        else
            dolas_yaz(L->L_bag);
        dolas_yaz(L->bag);
    }
}
```

Sık yapılan hata

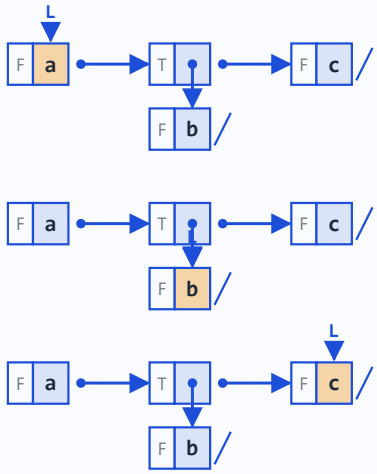
Girintiye güvenmek. C'de blok süslü parantezle kurulur; girinti yalnız okuyan içindir. Tek deyimlik `if` gövdelerine bile süslü parantez koymak, sonradan eklenen bir satırın yanlışlıkla dışarıda kalmasını önler. `-Wall -Wextra` ile derlemek, bu türden hataların çoğunu daha çalıştırmadan gösterir.

9.5.8 Kod izleme: `dolas_yaz(L)`

L = (a, (b), c) gezilsin. İkinci öğe tek öğeli bir alt listedir. Tablo 9.7 çağrıları ve çıktının nasıl büyüdüğünü gösterir.

Adım	Yığın (sağdaki çalışıyor)	İşlem	Çıktı
1	dolas_yaz(a)	atom: `'a'` yazılır; sonraki öğe için çağrı	`a`
2	dolas_yaz(a) · dolas_yaz((b))	alt liste: önce içine dalınır (`L->L_bag`)	`a`
3	… · dolas_yaz((b)) · dolas_yaz(b)	atom: `'b'` yazılır; sonraki öğe için çağrı	`ab`
4	… · dolas_yaz(b) · dolas_yaz(NULL)	`L == NULL` : düzeyin sonu, hemen dönülür	`ab`
5	dolas_yaz(a) · dolas_yaz((b)) · dolas_yaz(c)	alt listeden dönüldü; `(b)`'nin bağı ile *c*'ye geçildi; `'c'` yazılır	`abc`
6	… · dolas_yaz(c) · dolas_yaz(NULL)	düzeyin sonu; bütün çağrılar sırayla döner	`abc`

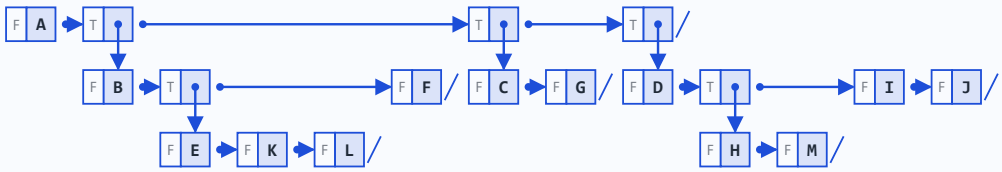
Tablo 9.7: `dolas_yaz(L)` izlemesi, *L = (a, (b), c)*.



Şekil 9.7: `dolas_yaz` 'ın *(a, (b), c)* üzerindeki üç anı: *a* yazılırken, alt listeye inilip *b* yazılırken ve aynı düzeyde *c*'ye geçilmişken. `L` etiketi o anda çalışan çağrının parametresini gösterir.

5. adımda *c*'ye geçiş, `(b)` öğesinin çağrısı içinden yapılır. Yığında `dolas_yaz(b)` artık yoktur; o çağrı `NULL` ile döndükten sonra bitmiştir. Ama `dolas_yaz((b))` hâlâ çalışır ve sıradaki işi, kendi `bag`'ini izlemektir. Atomlar derinlikten bağımsız olarak soldan sağa yazılır; çıktı `abc`'dir. Her öğe bir kez ziyaret edilir; `kopyala`'daki sayımla çağrı sayısı $1 + n + s = 1 + 4 + 1 = 6$'dır ve süre $\Theta(n)$'dir.

Aynı yordam bir ağaç biçimindeki listeye uygulandığında ilginç bir şey olur. Şekil 9.8, *A(B(E(K, L), F), C(G), D(H(M), I, J))* ağacını tutan listeyi gösterir. Ağacın her düğümü bir alt listedir: alt listenin başı düğümün etiketi, kalan öğeleri çocuklarıdır. Yaprak olan çocuklar doğrudan atom olarak, çocuğu olan çocuklar yine alt liste olarak yazılır. Liste doğrusal formda *(A, (B, (E, K, L), F), (C, G), (D, (H, M), I, J))*'dir.



Şekil 9.8: *A(B(E(K, L), F), C(G), D(H(M), I, J))* ağacının genelleştirilmiş liste görünümü. Her alt listenin ilk atomu bir ağaç düğümünün etiketidir, kalan öğeler o düğümün çocuklarıdır. Aynı düzeydeki öğeler bir satır boyunca izlenir.

`dolas_yaz` bu liste için `ABEKLFCGDHMIJ` yazar. Bu sıra rastlantı değildir. Her alt liste önce kendi başını, yani düğümün etiketini yazar, sonra çocuklarını soldan sağa aynı kuralla gezer. Bu, ağaçların **kök önde** (*preorder*) gezinmesidir: önce kök, sonra alt ağaçlar sırayla. Liste yordamı ağaçlar hakkında hiçbir şey bilmeden bir ağaç algoritması çalıştırmış olur. Verinin iyi bilinen bir yapıya taşınmasının kazancı tam olarak budur.

9.6 Derinlik

Bir genelleştirilmiş listenin ne kadar iç içe olduğu, onu işleyen özyinelemeli yordamların yığında ne kadar yer tutacağını belirler. Bir ağacı tutan listede aynı ölçü ağacın yüksekliğine karşılık gelir. Bu kesimde derinlik önce tanımlanır, sonra özyinelemeli ve yinelemeli iki yordamla hesaplanır.

9.6.1 Özyinelemeli derinlik

Tanım 9.3

Boş listenin **derinliği** 0'dır. Boş olmayan bir L listesinin derinliği, 1 ile L'nin alt listelerinin derinliklerinin en büyüğünün toplamıdır; alt listesi yoksa bu en büyük değer 0 alınır. Atomlar derinliğe katkı yapmaz.

Tanıma göre *(a, b)*'nin derinliği 1'dir: alt listesi yoktur. *(a, (b, c))*'nin derinliği *1 + 1 = 2*, *(a, (b, (c)))*'nin derinliği 3'tür. *(())* ilginç bir örnektir. Bu liste boş değildir; tek öğesi, boş bir alt listedir. Alt listenin derinliği 0 olduğundan derinlik *1 + 0 = 1* çıkar. Aynı yoldan *((()))*'nin derinliği 2'dir. Sunumun sorusu bu tanımı özyinelemeli olarak koda çevirmeyi ister:

```
int derinlik(oge_gost L) {
    oge_gost p = L;  int m = 0;    // m: en büyük
    if (!L) return 0;
    while (p) {
        if (p->isaret) {
            int n = derinlik(p->L_bag);
            if (m < n) m = n;
        }
        p = p->bag;
    }
    return m + 1;
}
```

Yordam tanımı neredeyse kelimesi kelimesine izler. Boş liste için 0 döner. Boş olmayan listede aynı düzeyin öğeleri bir `while` döngüsüyle gezilir ve `m` o ana kadar görülen en derin alt listenin derinliğini tutar. Atom öğeleri atlanır. Her alt liste öğesinde alt listenin derinliği özyinelemeli olarak hesaplanır ve gerekirse `m` güncellenir. Döngü bitince `m + 1` döndürülür: en derin alt liste artı bu düzey.

Bu yordam, kopyalamadan farklı olarak `bag` boyunca özyineleme yapmaz; aynı düzeyi bir döngüyle gezer. Özyinelemeli çağrı yalnız bir alt listeye inerken yapılır. Bu yüzden çağrı sayısı *1 + s*'dir: dışarıdan bir çağrı ve her alt liste için bir çağrı. Her öğe döngü içinde tam bir kez ziyaret edilir ve süre *Θ(n)*'dir. Aynı anda yığında bulunan çağrı sayısı en çok listenin derinliği kadardır. Yordamın ek belleği bu yüzden *O(derinlik)*'tir ve uzun ama sığ listelerde yığın sorunu çıkmaz. Denemede bir milyon atomluk düz bir listenin derinliği sorunsuz hesaplanmıştır; aynı liste `kopyala`'yı yığın taşmasına sürüklemiştir.

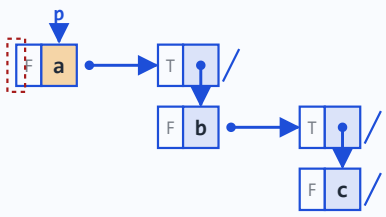
Sunumdaki örnekler denemeyle doğrulanmıştır: *(a, b)* → 1, *(a, (b, c))* → 2, *(a, (b, (c)))* → 3, *(())* → 1. Ek olarak *()* → 0, *((()))* → 2 ve sınav sorusu listesi *(d, (c, (e, (m, k), ()), z, y, (x, (r), b, a, ())))* → 5 bulunmuştur.

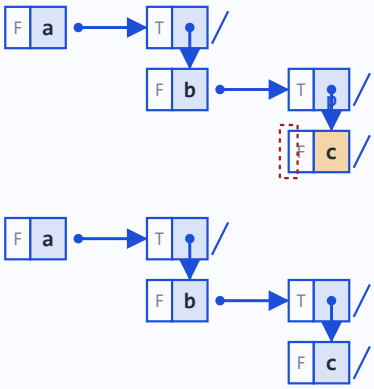
9.6.2 Kod izleme: `derinlik(L)`

L = (a, (b, (c))) için yordam üç düzeye iner. Her düzeyin çağrısı kendi `p` ve `m` değişkenlerine sahiptir; bir düzeydeki `m`, öteki düzeylerin `m`'sinden bağımsızdır. Tablo 9.8 her çağrının yerel değişkenlerini ayrı sütunlarda gösterir.

Adım	Çalışan çağrı	İşlem	`m` (1. / 2. / 3. düzey)
1	1. düzey	`p` = *a*; atom, atlanır	0 / — / —
2	1. düzey	`p` = *(b, (c))*; alt liste: 2. düzey çağrılır	0 / — / —
3	2. düzey	`p` = *b*; atom. `p` = *(c)*; alt liste: 3. düzey çağrılır	0 / 0 / —
4	3. düzey	`p` = *c*; atom. `p = p->bag` → `NULL` : düzey bitti	0 / 0 / 0
5	3. düzey	`return m + 1` = 1	0 / 0 / —
6	2. düzey	`n = 1` ; `m < n` : `m = 1` . `p` → `NULL` ; `return 2`	0 / 1 / —
7	1. düzey	`n = 2` ; `m = 2` . `p` → `NULL` ; `return 3`	2 / — / —

Tablo 9.8: `derinlik(L)` izlemesi, *L = (a, (b, (c)))*. Her çağrının kendi `m`'si vardır.





Şekil 9.9: `derinlik(L)` sırasında `p` 'nin gösterdiği öğe. Üstte 1. düzey çağrı *a*'dadır. Ortada 3. düzey çağrı *c*'dedir. Altta bütün düzeyler bitmiş, 1. düzey çağrı 3 döndürür.

İzlemenin anlattığı şey, özyinelemenin değerleri nasıl taşıdığıdır. Bilgi aşağı doğru yalnız bir adres olarak iner: hangi alt listeye bakılacağı. Cevap ise yukarı doğru, dönüş değeri olarak çıkar. Her düzey yalnız kendi işini bilir: alt listelerinden gelen sayıların en büyüğünü bulup bir ekler. Kimse toplam derinliği bir yerde tutmaz, ama sonuç en üstte doğru biçimde oluşur.

9.6.3 Hatalı bir yinelemeli çözüm

Aynı soruya bir sınavda verilmiş yinelemeli bir cevap şudur:

```
int derinlik( agac_gost L ){
    agac_gost yigit_al(); void yigit_ekle(agac_gost);
    int max = 0;
    for( int current = 1, p = L ; p ; p->bag ){
        if( p->isaret ){ // alt listeye gir
            yigit_ekle(p);
            p = p -> L_bag;
            current++;
        }
        if( !p->bag ){   // liste sonu, üste çık
            p = yigit_al();
            if( max < current ) max = current;
            current--;
        }
    }
    return max;
}
```

Fikir şudur: `current` o anki düzeyi, `max` görülen en büyük düzeyi tutar. Alt listeye inerken T öğesi bir yığına konur ve düzey artar; bir düzeyin sonuna gelince yığından bir öğe alınıp üste çıkılır. Fikirde yanlış yoktur; doğru bir yinelemeli çözüm de aynı yolu izler. Yanlış olan ayrıntılardır ve hepsi önemlidir.

Kod derlenmez. `int current = 1, p = L` bildirimi iki değişkeni de `int` yapar; virgülle ayrılan bildirimlerde tip, satırın başındaki tiptir. `p` bir tam sayı olunca `p->isaret`, `p->L_bag` ve `p->bag` ifadelerinin hiçbiri geçerli değildir. Derleyici yedi hata vermiştir; ilki "incompatible pointer to integer conversion initializing 'int'" iletisidir. `for` 'un üçüncü kısmındaki `p->bag` da bir atama değildir. Değer hesaplanır ve atılır; `p` hiç ilerlemez. Bu hata tek başına düzeltilse bile döngü sonsuza kadar aynı öğede kalırdı. Doğrusu `p = p->bag` 'dir.

Bunlar düzeltilse de kod uç durumlarda çöker. Boş bir alt listeye inildiğinde `p = p->L_bag` `NULL` olur ve hemen ardından `p->bag` okunur. Liste bittiğinde yığın boş olsa da `yigit_al` çağrılır. Yığından alınan T öğesinin kendisine dönülür, ama onun kardeşine, yani `T->bag` 'e geçilmez. Bu yüzden aynı alt listeye yeniden inilir. Ayrıca yığın işlemleri dışarıdaki bir küresel yığına dayanır.

Bu çözüm iki ayrı açıdan eleştirilir. Birincisi beklenen çözüm özyinelemelidir. `current` ve `max` sayaçlarıyla döngü kurmak, birinci sınıf düzeyinde bir yaklaşımdır ve yapının özyinelemeli tanımından uzaklaşır. İkincisi küresel değişkenler yasaktır. Küresel bir yığın ya da küresel bir sayaçla yazılan yordam, tek başına denenirken çalışır. Ama aynı yordam başka bir bileşenin parçası olabilir. Örneğin iki listenin derinliği iç içe hesaplandığında ya da iki iş parçacığı aynı anda çağırdığında ortak değişken bozulur. Yordamın bütün durumu kendi yerel değişkenlerinde olmalıdır.

9.6.4 Yerel yığınla yinelemeli derinlik

Yinelemeli çözüm doğru yazılabilir. Gereken şey, özyinelemenin derleyiciye bıraktığı çağrı yığınını açıkça, yerel bir dizide tutmaktır:

```
int derinlik_d(oge_gost L) {
    oge_gost yigit[100];       // yerel yığın (derinlik<100)
    int ust = 0, d = 1, m = 0;
    oge_gost p = L;
    if (!L) return 0;
    for (;;) {
        if (p == NULL) {             // düzey bitti
            if (ust == 0) return m;
            p = yigit[--ust]->bag;   // T'den sonraki
            d--;
        } else {
            if (d > m) m = d;
            if (p->isaret && p->L_bag) {
                yigit[ust++] = p;    // alt listeye in
                p = p->L_bag;
                d++;
            } else p = p->bag;
        }
    }
}
```

`yigit` dizisi, içine inilmiş alt listelerin T öğelerini tutar; `ust` dizideki öğe sayısıdır. `d` o anki düzey, `m` görülen en büyük düzeydir. Döngünün her turunda iki durumdan biri vardır.

- `p` boş değilse bir öğenin üzerindeyiz. Bu öğe `d`. düzeydedir, dolayısıyla `m` en az `d` olur. Öğe boş olmayan bir alt listeyse T öğesi yığına konur, `p` alt listenin ilk öğesine geçer ve `d` bir artar. Öğe atomsa ya da boş bir alt listeyse `p` aynı düzeydeki sonraki öğeye geçer.
- `p` boşsa bir düzeyin sonuna gelinmiştir. Yığın da boşsa en üst düzey bitmiştir ve `m` döndürülür. Değilse en son inilen T öğesi yığından alınır ve onun *kardeşine*, `T->bag`'e geçilir; `d` bir azalır.

Hatalı çözümün bütün eksikleri bu yazımda giderilmiştir. `p` bir işaretçidir. Boş alt listeye hiç inilmez, çünkü `p->L_bag` ayrıca denetlenir. Yığından alma yalnız yığın boş değilken yapılır. Yığından alınan öğenin kendisine değil kardeşine geçilir. Küresel değişken yoktur. Boş alt listenin derinliğe katkısı da doğrudur. *(())* listesinin tek öğesi `d = 1`'de görülür ve `m` 1 olur; içine inilmediği için 2 olmaz.

Tur	p	d	m	yigit	İşlem
1	a	1	0	boş	`m = 1`; atom: `p = p->bag`
2	T_1	1	1	boş	alt liste: T_1 yığına; `p` = b, `d = 2`
3	b	2	1	T_1	`m = 2`; atom: sonraki
4	T_2	2	2	T_1	alt liste: T_2 yığına; `p` = c, `d = 3`
5	c	3	2	T_1, T_2	`m = 3`; atom: `p = NULL`
6	`NULL`	3	3	T_1, T_2	düzey bitti: T_2 alınır, `p = T2->bag = NULL`, `d = 2`
7	`NULL`	2	3	T_1	T_1 alınır, `p = T1->bag = NULL`, `d = 1`
8	`NULL`	1	3	boş	yığın boş: `return 3`

Tablo 9.9: `derinlik_d(L)` izlemesi, *L = (a, (b, (c)))*. Her satır döngünün bir turunun başındaki durumdur; T_1 = *(b, (c))*, T_2 = *(c)* öğesi.

Tablonun değerleri, yordamın döngü başına bir yazdırma satırı eklenmiş kopyasının çıktısıyla aynıdır. Yordamın doğruluğu ayrıca iki yoldan sınanmıştır. On iki örnek listede ve 20.000 rastgele listede `derinlik_d` ile `derinlik` aynı sonucu vermiştir. Süre yine $\Theta(n)$'dir: her öğe bir kez ziyaret edilir, her T öğesi yığına bir kez konur ve bir kez alınır.

Yığın dizisinin sabit boyutu bu çözümün zayıf noktasıdır. 100 hücre, en çok 100 iç içe alt listeye inmeye yeter; yani derinliği 101'e kadar olan listeler doğru hesaplanır. Daha derin bir listede `yigit[ust++] = p` dizinin dışına yazar. Denemede derinliği 102 olan bir zincirde UBSan "index 100 out of bounds for type 'oge_gost[100]'" hatasıyla programı durdurmuştur. Sağlam bir sürüm ya `ust == 100` olduğunda bir hata döndürür ya da yığını `malloc` ve `realloc` ile büyütür. Özyinelemeli sürümün de bir sınırı vardır, ama o sınır çağrı yığınının boyutudur ve çok daha yüksektir: aynı denemede derinliği 5000 olan zincirde `derinlik` sorunsuz çalışmıştır.

Sık yapılan hata

Yinelemeli gezinmede yığından alınan T öğesinin kendisine dönmek. T öğesinin alt listesi zaten bitmiştir. Gezinme, T'nin kardeşinden, yani `T->bag` 'den sürmelidir. T'nin kendisine dönen bir kod aynı alt listeye yeniden iner ve döngüden hiç çıkamaz.

9.6.5 Özyinelemeli düşünmek ve uç örnekler

Derinlik sorusunun iki çözümü yan yana konunca fark açıkça görülür. Özyinelemeli sürüm on satırdır ve tanımı birebir izler. Yinelemeli sürüm, özyinelemenin kendiliğinden yaptığı işi, yani hangi alt listede olunduğunu ve oradan nereye dönüleceğini, elle yapmak zorundadır. Bu sürüm daha uzundur, sabit bir sınır taşır ve yazarken hata yapmak çok daha kolaydır.

Özyinelemeli düşünmek bir alışkanlıktır ve kazanılması zaman alır. Bu altyapı için bir dönemlik ders yetmez. Her özyinelemeli çağrı için "Buna şu listeyi verirsem bana ne döndürür?" sorusu sorulmalı, cevap yazılmadan yordamın geri kalanına geçilmemelidir. Kodu bir hata ayıklayıcıyla adım adım çalıştırmak ve her dönüşte hangi değerin nereye gittiğini izlemek, bu sorunun cevabını görmenin en iyi yoludur. Bu notun izleme tabloları aynı işi kâğıt üzerinde yapar.

Yalnız verilen örnekte çalışan kod yeterli değildir. Bir yordam, verilen örnekte doğru sonuç verirken başka örneklerde çökebilir. Doğru çalışan kod, uç örneklerde de doğru çalışan koddur. Genelleştirilmiş listeler için uç örnekler bellidir. Tablo 9.10 bu örnekleri ve denemelerde görülen davranışı toplar.

Liste	Neyi sınar	Gözlenen
()	boş liste; `NULL` denetimi	`derinlik` 0; `kopyala` `NULL` ; `dogrusal_yaz` `()`
(a)	tek öğe	derinlik 1; ters çevirme aynı listeyi verir
(())	boş alt liste; `L_bag == NULL` olan T öğesi	derinlik 1; `dogrusal_yaz` `(())`
((()))	iç içe boş alt listeler	derinlik 2
tek düzeyli uzun liste	`bag` boyunca özyineleme	$5 \cdot 10^4$ atomda bütün yordamlar çalıştı; 10^5 atomda `kopyala` ve `esitmi` , 10^6 atomda `dolas_yaz` yığın taşmasıyla durdu; `derinlik` ve `devrik_al` 10^6 atomda çalıştı
derin tek zincir *(((...(a)...)))*	`L_bag` boyunca özyineleme; sabit yığın dizisi	derinlik 5000'de `derinlik` çalıştı; `derinlik_d` 101'den derinde dizi dışına taştı

Tablo 9.10: Uç örnek listeleri ve yordamların davranışı (deneme: `-O0` , Windows, 1 MB varsayılan yığın).

Tablonun beşinci satırı önemli bir ayrıntıyı gösterir. `kopyala` , `esitmi` ve `dolas_yaz` , `bag` boyunca da özyineleme yapar; bu yüzden uzun bir düzeyde her öğe yığında bir çağrı tutar. `derinlik` ve `devrik_al` ise aynı düzeyi döngüyle gezer ve yığında yalnız iç içelik kadar yer tutar. Hangi yordamın hangi listelerde güvenle kullanılabileceği, karmaşıklık tablosunda ek bellek sütunuyla da özetlenir. `dolas_yaz` 'ın `kopyala` 'dan daha uzun listelere dayanması, her çağrısının yığında daha az yer tutmasındandır.

Veri yapıları dersi, bilgisayar biliminin bel kemiğidir. "Liste, ağaç falan değil mi?" diye özetlenebilecek bir ders değildir; beyin cerrahisini "şu damarlar" diye özetlemek gibi bir haksızlık olur. Listeyi ve ağacı tanımak, veri yapısını öğrenmiş olmak demek değildir. Öğrenmenin ölçüsü, o yapıyı kuran, kopyalayan, karşılaştıran ve gezen yordamları yazabilmektir. Bunun için sağlam bir matematik altyapısı da gerekir: ayrık matematik, olasılık ve istatistik, rastgele süreçler. Programlama ödevleriyle erken başlayan bir eğitim hızlı kod yazmayı öğretir, ama kuramı ihmal etme tehlikesi taşır. Bölümün dersleri her şeyi veremez; eksik görülen konular erken fark edilip matematik ya da istatistik bölümlerinden alınacak derslerle tamamlanabilir. R. Johnsonbaugh'un *Discrete Mathematics* kitabı bu altyapı için önerilen kaynaklardandır.

9.7 Ters çevirme

Bir genelleştirilmiş listeyi ters çevirmek, her düzeydeki öğelerin sırasını tersine çevirmektir. Ters çevirme her düzeye uygulanır: *L = (a, (b, c))* listesinin tersi *((c, b), a)*'dır. Yalnız üst düzey çevrilseydi sonuç *((b, c), a)* olurdu. Bu işlem, bir ağaç ya da iç içe bir şekil tutan listede görüntünün aynadaki yansımasını almaya karşılık gelir.

9.7.1 Üç işaretçi ve iki ek satır

Tek yönlü bağlı listeyi yerinde ters çeviren klasik yordam üç işaretçi kullanır. `p` henüz çevrilmemiş kısmın başını, `q` çevrilmiş kısmın başını, `r` ise bir önceki `q`'yu tutar. Her turda `p`'deki öğe çevrilmiş kısmın başına alınır. Genelleştirilmiş liste için bu yordama yalnız iki satır eklenir:

```
void devrik_al(oge_gost *L) {
    oge_gost r, p = *L, q = NULL;
    while (p) {
        if (p->isaret)              // atomik değilse
            devrik_al(&p->L_bag);
        r = q;
        q = p;
        p = p->bag;
        q->bag = r;
    }
    *L = q;
}
```

Döngünün dört ataması klasik yordamın aynısıdır. `r = q` bir önceki turda çevrilen öğeyi saklar. `q = p` bu turda çevrilecek öğeye geçer. `p = p->bag` kalan listeye ilerler; bu atama bağ değiştirilmeden önce yapılmalıdır, yoksa kalan liste kaybolur. `q->bag = r` öğenin bağını geriye çevirir. Döngü bitince `q` eski listenin son öğesini, yani yeni listenin ilk öğesini gösterir ve `*L = q` ile listenin başı yapılır.

Eklenen iki satır, alt liste öğesine gelindiğinde önce alt listenin kendisini ters çevirir. Burada `devrik_al`'a `p->L_bag`'in değeri değil adresi verilir. Bunun nedeni, ters çevirmenin listenin başını değiştirmesidir. *(b, c)* çevrilince alt listenin ilk öğesi *b* değil *c* olur. T öğesinin `L_bag` alanı bu yeni başı göstermelidir. İç çağrı son satırında `*L = q` yazar. `L`, T öğesinin `L_bag` alanının adresi olduğu için bu atama doğrudan o alanı günceller. Dış çağrının dönüşten sonra yapacak bir işi kalmaz. Değer verilseydi iç çağrı kendi yerel kopyasını güncellerdi. T öğesi eski başı, yani artık son öğe olan *b*'yi göstermeye devam ederdi ve *c* erişilemez hâle gelip sızardı.

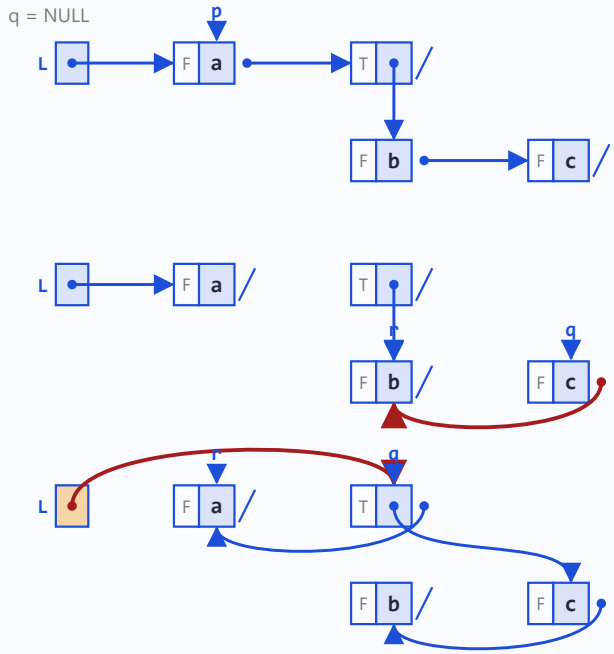
Aynı nedenle dışarıdan yapılan çağrı da adresle yapılır: `devrik_al(&L)`. En üst düzeyin başı da değişir ve çağıranın `L` değişkeni güncellenmelidir. Baş değişen bir yordama listeyi değerle vermek, bağlı listelerin en sık görülen hatalarından biridir.

9.7.2 Kod izleme: `devrik_al(&L)`

L = (a, (b, c)) ters çevrilsin. Tablo 9.11 sunumun on iki adımını verir. T, ikinci üst düzey öğe, yani alt liste öğesidir.

Adım	Çağrı	İşlem	`r`	`q`	`p`
1	dış	`p = *L`, `q = NULL`	?	NULL	*a*
2	dış	*a* atom; dört atama: *a*'nın bağı `NULL` oldu	NULL	*a*	T
3	dış	T alt liste: `devrik_al(&p->L_bag)`	NULL	*a*	T
4	iç	`p = *L` = *b*, `q = NULL`	?	NULL	*b*
5	iç	*b*'nin bağı `NULL` oldu	NULL	*b*	*c*
6	iç	*c*'nin bağı *b*'ye çevrildi	*b*	*c*	NULL
7–8	iç	`p == NULL`: düzey bitti; `*L = q`: T'nin `L_bag`'i artık *c*	*b*	*c*	NULL
9	dış	iç çağrıdan dönüldü; T zaten yeni başı gösteriyor	NULL	*a*	T
10	dış	T'nin bağı *a*'ya çevrildi	*a*	T	NULL
11–12	dış	`p == NULL`; `*L = q`: listenin başı T	*a*	T	NULL

Tablo 9.11: `devrik_al(&L)` izlemesi, *(a, (b, c)) → ((c, b), a)*.



Şekil 9.10: `devrik_al(&L)` 'nin üç anı. Üstte başlangıç: $L \to a \to$ T, T'nin altında $b \to c$. Ortada üst düzeyde a'nın bağı kesilmiş, alt listede c'nin bağı b'ye çevrilmiştir. Altta sonuç: $L \to$ T $\to a$; T'nin alt listesi $c \to b$.

Yordam hiçbir öğe ayırmaz ve hiçbir öğeyi bırakmaz; yalnız bağları değiştirir. Denemede ters çevirme sırasında `malloc` sayacı hiç artmamıştır. Süre $\Theta(n)$'dir: her öğe bir kez `p` olur. Çağrı sayısı $1 + s$'dir ve aynı düzey döngüyle gezildiği için ek bellek $O(derinlik)$'tir. Deneme şu sonuçları vermiştir: *((a, b), ((c, d), e))* → *((e, (d, c)), (b, a))*; *(a, ((), b)* → *(b, ((), a)*; *(a, (b, (c)))* → *(((c), b), a)*; *()* ve *(a)* değişmez. Her listeyi iki kez ters çevirmek ilk listeyi geri vermiştir; bu özellik 20.000 rastgele listede de sınanmıştır.

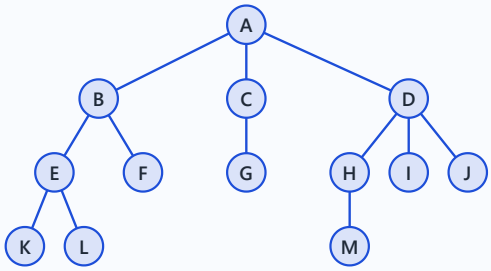
Bu çözüm, iyi bilinen bir yordamın küçük bir eklemeyle genelleştirilmesidir. Aynı sonuca kaba kuvvetle de varılabilirdi. Örneğin her düzeyin öğelerini bir diziye toplayıp ters sırayla yeniden bağlamak ya da öğeleri tek tek yeni bir listenin başına kopyalamak bu işi görür. Bu yollar da ters çevirir, ama gereksiz bellek ve iş harcar; bir fırtına gibi önündeki her şeyi yıkıp yeniden kurar. Üç işaretçili yordam yerinde çalışır ve yapıdaki her bağa yalnız bir kez dokunur.

9.8 Ağacı listeyle kurmak

Bir ağaç, bir kök düğümden ve her biri yine bir ağaç olan sıfır ya da daha çok alt ağaçtan oluşur. Bu tanım da özyinelemelidir ve genelleştirilmiş listenin tanımına çok benzer. Bu benzerlik, bir ağacın tek satırlık bir metinle ve bu metinden kurulan bir listeyle saklanmasını sağlar.

9.8.1 Ağacın doğrusal formu

Ağacın doğrusal formunda her düğüm etiketiyle yazılır; çocukları varsa etiketin hemen arkasından parantez içinde, virgülle ayrılarak sıralanır. *C(G)*, C'nin G'nin ebeveyni olduğunu söyler. *A(B..., C..., D...)*, A'nın B, C ve D'nin ebeveyni olduğunu söyler. Şekil 9.11 sunumdaki ağacı ve doğrusal formunu verir.



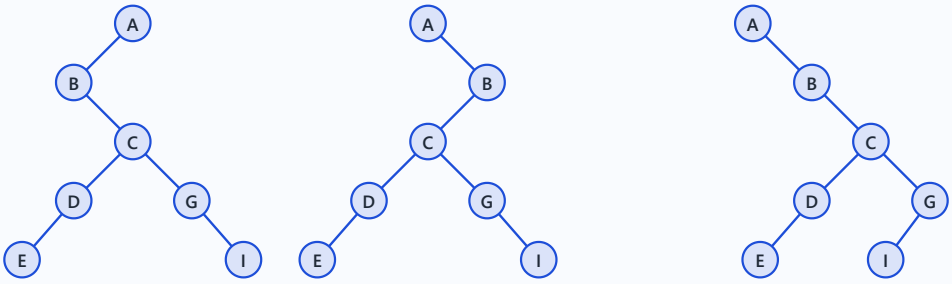
Şekil 9.11: Doğrusal formu *A(B(E(K, L), F), C(G), D(H(M), I, J))* olan ağaç. A'nın üç çocuğu vardır; K, L, F, G, M, I ve J yapraktır.

Metin, ağacın bütün yapısını taşır. Hangi düğümün kimin çocuğu olduğu ve kardeşlerin sırası metinden tek anlamlı olarak okunur. Bu kadar veri için tek satırlık bir metin yeter. Notasyon da bir seçimdir: virgül yerine nokta, normal parantez yerine süslü parantez kullanılabilir. Önemli olan, metinden yapıya ve yapıdan metne geçen yordamların yazılabilmesidir. Bu iki yön sağlandığında programın içindeki yapıyı dışarıdan kimsenin bilmesi gerekmez.

Metinden listeye geçiş için ağaç, ilk öğesi kök etiketi olan bir listeyle tutulur: *A(B, C)* ağacı *(A, B, C)* listesi olur. Çocuğu olan bir çocuk, yine bu biçimde bir alt listedir. Böylece sunumdaki ağaç *(A, (B, (E, K, L), F), (C, G), (D, (H, M), I, J))* listesine dönüşür. Bu dönüşüm ağaç metnindeki her *X(* parçasını *(X,* ile değiştirmekten ibarettir ve doğrusal form okuyucusu listeyi doğrudan kurar.

9.8.2 İkili ağaçta sağ–sol sorunu

Genel bir ağaçta çocukların sırası yeterli bilgidir: birinci çocuk, ikinci çocuk ve bu böyle gider. **İkili ağaçta** (*binary tree*) ise her düğümün en çok iki çocuğu vardır ve bunlar *sol* ve *sağ* diye ayrılır. Tek çocuklu bir düğümde bu çocuğun sol mu sağ mı olduğu yapının bir parçasıdır. İki ağaç yalnız bu bakımdan farklıysa, farklı ağaçlardır.



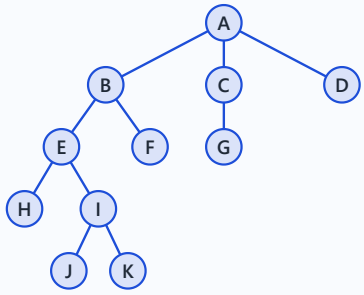
Şekil 9.12: Doğrusal formu aynı, *A(B(C(D(E), G(I))))* olan üç farklı ikili ağaç. Soldakinde B, A'nın sol çocuğu ve C, B'nin sağ çocuğudur. Ortadakinde B sağ, C sol çocuktur. Sağdakinde B ve C sağ çocuklardır; I de G'nin sol çocuğudur.

Şekil 9.12 sorunu gösterir. Üç ağacın da doğrusal formu *A(B(C(D(E), G(I))))*'dir. Metin, B'nin A'nın tek çocuğu olduğunu söyler ama hangi yanda olduğunu söyleyemez. Aynı belirsizlik C, E ve I için de vardır. Doğrusal form ikili ağaçta bilgi kaybeder.

Çözüm, eksik bilgiyi metne eklemektir. Ya her düğümün yanında L ya da R işareti tutulur, ya da boş çocuk yerine özel bir simge yazılır. İkinci yolda soldaki ağaç *A(B(, C(...)),)* diye yazılabilir: B'nin sol çocuğu boştur, bu yüzden virgülden önce hiçbir şey yazılmamıştır. Sunumdaki *A(B(, C(...)))* kısaltması bu fikri gösterir. Böylece her düğümün çocuk listesi tam iki konumlu olur ve bilgi kaybolmaz.

9.8.3 Ağaçtan metne: `agac_yaz`

Sunumun ödevlerinden ikisi bu konuyla ilgilidir. Birincisi *A(B(E(H, I(J, K)), F), C(G), D)* listesinin bellek görünümünü ve ağaç yapısını çizmeyi ister. İkincisi, ağaç yapısı verildiğinde liste yazımını çıktı veren yordamı ister.



Şekil 9.13: Birinci ödevin ağacı: *A(B(E(H, I(J, K)), F), C(G), D)*. I'nın iki çocuğu (J, K) vardır; D bir yapraktır.

Ağaç yapısı Şekil 9.13'te verilmiştir. Listesi *(A, (B, (E, H, (I, J, K)), F), (C, G), D)*'dir. Bellek görünümünde en üst satır A, T, T, D öğelerinden oluşur. İlk T'nin altında B, T, F; o T'nin altında E, H, T; en altta I, J, K bulunur. İkinci T'nin altında C ve G vardır. Bu liste *4 + 3 + 3 + 3 + 2 = 15* öğedir ve derinliği 4'tür. İkinci ödevin çözümü şudur:

```
void agac_yaz(oge_gost L) {   // L: (kök, alt ağaçlar...)
    printf("%c", L->veri);          // kök etiketi
    if (L->bag) {
        printf("(");
        for (oge_gost p = L->bag; p; p = p->bag) {
            if (p->isaret) agac_yaz(p->L_bag);
            else printf("%c", p->veri);   // yaprak
            if (p->bag) printf(",");
        }
        printf(")");
    }
}
```

Yordamın parametresi, ilk öğesi kök etiketi olan bir listedir. Önce kök etiketi yazılır. Listede kökten sonra öğe varsa, yani düğümün çocukları varsa, bir açılış parantezi yazılır ve çocuklar sırayla gezilir. Alt liste olan çocuk bir alt ağaçtır ve aynı yordamla yazılır. Atom olan çocuk bir yapraktır ve yalnız etiketi yazılır. Son çocuk dışındaki her çocuğun arkasına virgül konur. En sonda parantez kapatılır.

Bu yordam, `dolas_yaz`'dan farklı olarak parantezleri ve virgülleri de yazar. Ama parantezi her alt listenin başında değil, kök etiketinin *arkasında* açar. Bu küçük fark, liste notasyonunu ağaç notasyonuna çevirir. Denemede sunumdaki ağacın listesi `A(B(E(K,L),F),C(G),D(H(M),I,J))`, ödevdeki ağacın listesi `A(B(E(H,I(J,K)),F),C(G),D)` çıktısını vermiştir. İkisi de ağaçların doğrusal formuyla aynıdır. Tek düğümlü *(A)* listesi için yalnız `A` yazılır. Yordam boş liste için tanımlı değildir; boş ağaç için `L->veri` okunur ve program çöker. Çağıranın, ağacın boş olmadığını denetlemesi gerekir.

Süre $\Theta(n)$'dir: her öğe bir kez yazılır. Aynı düzey döngüyle gezildiği için yığın derinliği ağacın yüksekliği kadardır.

9.9 Paylaşım, referans sayma ve çöp toplama

Paylaşımlı listeler bellekten tasarruf sağlar; aynı alt liste birçok yerde kopyalanmadan kullanılır. Ama bu tasarrufun bir bedeli vardır. Paylaşılan bir alt liste ne zaman bırakılabilir? Onu gösteren yerlerden biri artık onu kullanmıyor diye bırakılırsa, öteki yerler iade edilmiş belleği gösterir. Hiç bırakılmazsa bellek sızar.

9.9.1 Başlık düğümü ve referans sayısı

Horowitz ve Sahni'nin önerdiği çözümde her liste bir **başlık düğümüyle** (*header node*) başlar. Başlık düğümü listenin bir öğesi değildir; listenin kimliğidir. Listeyi gösteren herkes, ilk öğeyi değil başlık düğümünü gösterir. Başlıkta bir `sayac` alanı tutulur. Bu alan, başlığı gösteren işaretçi sayısıdır; buna listenin **referans sayısı** (*reference count*) denir. Örnekte öğe yapısına `int sayac;` alanı eklenmiştir.

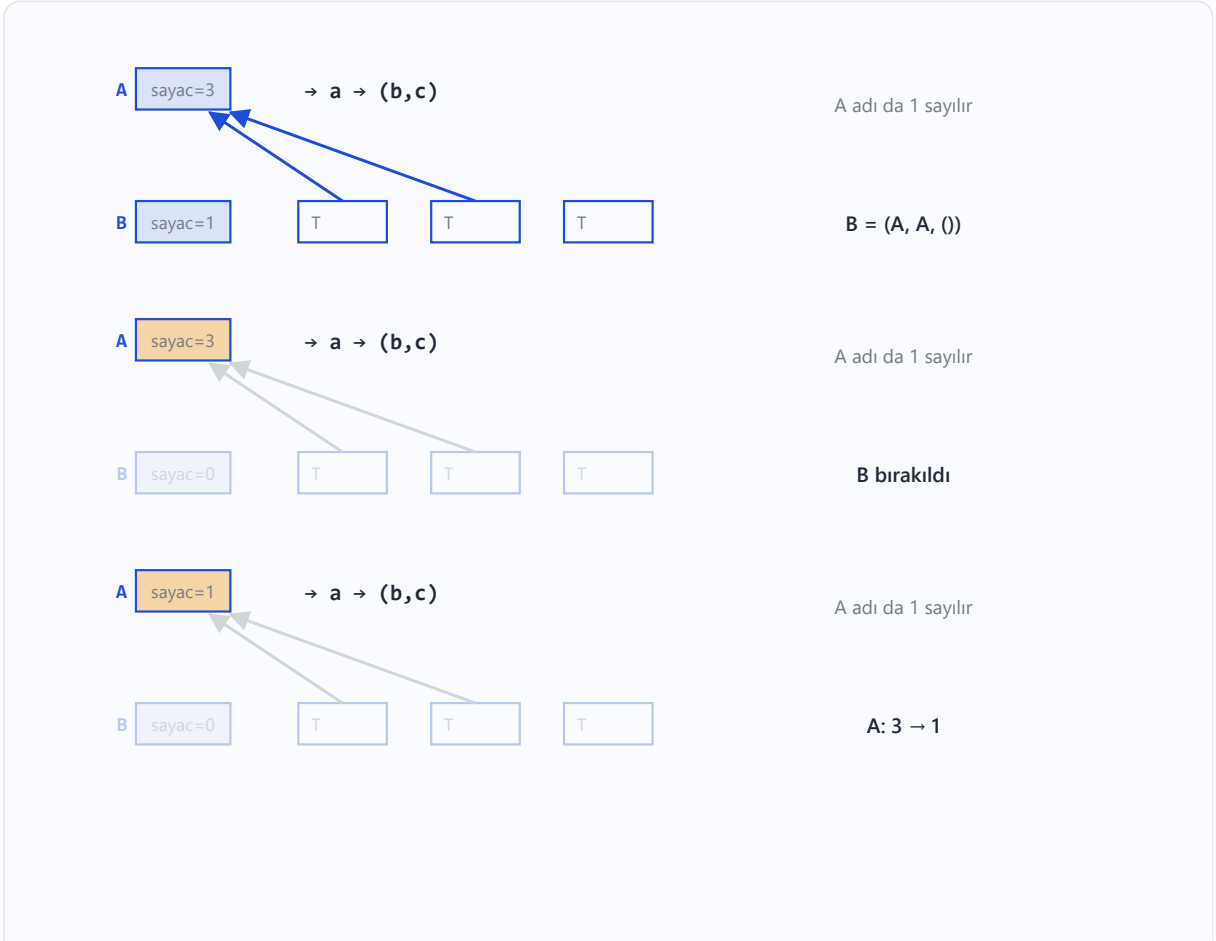
```
// başlıkta sayac: bu listeyi gösteren sayısı
void liste_birak(oge_gost *L) {
    oge_gost p, q;
    if (*L == NULL) return;
    if (--(*L)->sayac > 0) {    // başkası kullanıyor
        *L = NULL; return;
    }
    p = (*L)->bag;              // ilk gerçek öğe
    free(*L);
    while (p) {
        if (p->isaret) liste_birak(&p->L_bag);
        q = p; p = p->bag; free(q);
    }
    *L = NULL;
}
```

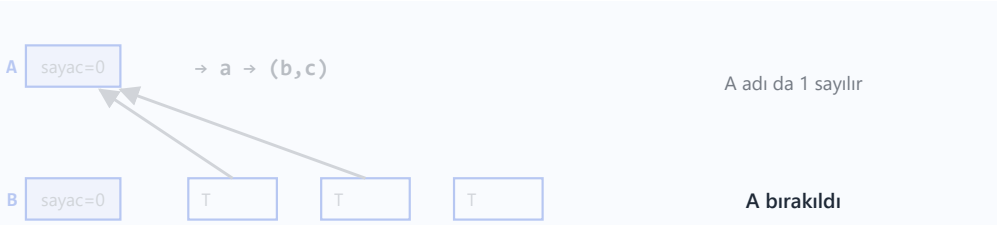
`liste_birak`, listeyi gösteren bir işaretçinin artık kullanılmayacağını bildirir. Önce sayacı bir azaltır. Sayaç hâlâ sıfırdan büyükse listeyi başkaları da kullanıyor demektir. Yordam yalnız çağıranın işaretçisini `NULL` yapar ve döner. Sayaç sıfıra inmişse listeyi gösteren kimse kalmamıştır. Başlık bırakılır, ardından öğeler sırayla bırakılır. Alt liste öğelerinde alt listenin kendisi aynı yordamla bırakılır. Bu bir zincirleme kuraldır: alt listenin de başka göstereni varsa sayacı yalnız azalır, bellekte kalır.

Yordamın üç ayrıntısı önemlidir. Birincisi, ilk öğenin adresi başlık bırakılmadan önce `p`'ye alınır; `free(*L)`'den sonra `(*L)->bag` okunamaz. İkincisi, döngüde her öğenin bağı `free`'den önce okunur. Üçüncüsü, parametre `oge_gost *L`'dir ve yordam her durumda `*L = NULL` yazar. Böylece çağıranın işaretçisi sarkan bir işaretçi olarak kalmaz. Sayacın doğru kalması programcıya bağlıdır: listeyi gösteren yeni bir işaretçi oluşturan her kod sayacı bir artırmalı, işaretçiyi bırakan her kod `liste_birak`'ı çağırmalıdır.

9.9.2 Adım adım: `B = (A, A, ())`

A = (a, (b, c)) olsun ve *B = (A, A, ())*. *A*'nın başlığını üç işaretçi gösterir: `A` adlı değişken ve *B*'nin ilk iki T öğesi. Sayacı bu yüzden 3'tür. *B*'nin başlığını yalnız `B` değişkeni gösterir ve sayacı 1'dir. *B*'nin üçüncü öğesinin gösterdiği boş listenin de kendi başlığı vardır ve sayacı 1'dir.





Şekil 9.14: Referans sayma adım adım. Yukarıdan aşağıya: başlangıçta *A*'nın sayacı 3, *B*'ninki 1; `liste_birak(&B)` *B*'nin sayacını 0'a indirir; *B*'nin iki T öğesi *A*'nın sayacını 3'ten 1'e indirir, *A* bırakılmaz; `liste_birak(&A)` sayacı 0'a indirir ve *A* bırakılır.

1. `liste_birak(&B)` : *B*'nin sayacı 1'den 0'a iner. Başlık bırakılır ve öğeler gezilmeye başlar.
2. Birinci T öğesi `liste_birak(&p->L_bag)` çağırır. *A*'nın sayacı 3'ten 2'ye iner; sıfır olmadığı için *A*'ya dokunulmaz, yalnız T öğesinin `L_bag`'i `NULL` olur. T öğesi bırakılır.
3. İkinci T öğesi aynı işi yapar: *A*'nın sayacı 2'den 1'e iner.
4. Üçüncü T öğesi boş listenin başlığı için çağrı yapar. O başlığın sayacı 1'den 0'a iner; başlık bırakılır, ilk öğesi olmadığı için döngü hiç dönmez. T öğesi bırakılır. `B` artık `NULL`'dır.
5. Program *A*'yı da bırakınca, `liste_birak(&A)` sayacı 1'den 0'a indirir. *A*'nın başlığı, *a*, T öğesi ve *(b, c)*'nin başlığı, *b* ve *c* bırakılır.

Bu senaryo, sayaçlı `malloc` / `free` sarmalayıcılarıyla birlikte çalıştırılmıştır. Yapı 11 blok kullanır: *A* için 6 (iki başlık ve dört öğe), boş liste için 1 başlık, *B* için 1 başlık ve 3 öğe. `liste_birak(&B)` 5 blok bırakmış, *A*'nın sayacını 1'de bırakmıştır. `liste_birak(&A)` kalan 6 bloğu bırakmıştır. Toplam 11 ayırma ve 11 bırakma sayılmıştır; aynı blok iki kez bırakılmamış ve sızıntı olmamıştır. Paylaşım güvenle yönetilmiştir.

Aynı fikir kopyalamaya da uygulanabilir. Paylaşımlı bir alt listenin kopyasını çıkarmak yerine onun sayacı bir artırılır ve yeni liste aynı başlığı gösterir. Kopyalama böylece hem hızlanır hem bellek kazanır. Bunun karşılığı, listelerin değiştirilmesine dikkat etmektir: paylaşılan bir listede yapılan değişiklik, onu gösteren her yerde görünür.

9.9.3 Referans saymanın sınırı ve çöp toplama

Referans sayma, özyinelemeli listelerde yetersiz kalır. *C = (a, C)*'nin başlığını iki işaretçi gösterir: `C` değişkeni ve listenin kendi T öğesi. Sayaç 2'dir. Program `liste_birak(&C)` çağırdığında sayaç 1'e iner ve liste bırakılmaz. Ama artık bu listeye programın hiçbir değişkeninden ulaşılamaz. Sayacı sıfıra indirebilecek tek işaretçi listenin kendi içindedir. Liste bellekte sonsuza kadar kalır. Denemede bu durum gözlenmiştir: `liste_birak(&C)`'den sonra `C` `NULL` olmuş, sayaç 1'de kalmış ve ayrılan 3 bloğun hiçbiri bırakılmamıştır. Aynı sorun, birbirini gösteren iki liste gibi her döngüde ortaya çıkar.

Döngülerin de toplanabilmesi için başka bir yöntem gerekir: **çöp toplama** (*garbage collection*). En bilinen biçimi **işaretle–süpür** (*mark and sweep*) yöntemidir ve iki aşamadan oluşur:

1. *İşaretleme.* Programın değişkenlerinden, yani **kök işaretçilerden** başlanır. Bunlardan erişilebilen bütün öğeler gezilir ve her birinin işaret biti doldurulur. Gezinme bir öğeyi ikinci kez gördüğünde durur; bu yüzden döngüler sorun olmaz.
2. *Süpürme.* Öğelerin bulunduğu bellek baştan sona taranır. İşaretlenmemiş her öğe, hiçbir yerden erişilemeyen bir çöptür ve kullanılabilir düğüm listesine eklenir. İşaretli öğelerin işareti bir sonraki toplama için silinir.

C = (a, C) örneğinde, `C` değişkeni silindikten sonra kök işaretçilerden hiçbiri listeye ulaşmaz. İşaretleme aşaması listenin öğelerine hiç uğramaz ve süpürme onları toplar. Döngü, referans saymanın aksine, burada bir engel değildir.

Çöp toplamanın da bedeli vardır. Toplayıcı çalışırken program durur ya da yavaşlar; büyük bir bellekte bu duraklama fark edilir. Her öğede bir işaret biti için yer gerekir. LISP, Java ve Python bu fikrin geliştirilmiş türevlerini kullanır. Python örneğin önce referans sayar, döngüleri ise ayrı bir toplayıcıyla bulur. C'de otomatik çöp toplama yoktur. Kopyalama, paylaşım ve bırakma sorumluluğu tamamen programcıdadır. Bu yüzden bir C programcısı, yazdığı her yordamın paylaşımlı ve özyinelemeli listelerde nasıl davrandığını bilmek zorundadır.

9.10 Sınav soruları

Sunumun son kesimi, genelleştirilmiş listeler üzerine sorulmuş üç sınav sorusunu verir. Üçü de bu notta anlatılan kalıbın uygulamasıdır.

9.10.1 Öğe sayısı ve alt liste sayaçları

Soru. Öğe yapısına `int sayac` alanı eklenmiştir. İşaret alanı `TRUE` olan her öğenin sayacına altındaki öğe sayısını yazan ve toplam öğe sayısını döndüren yordamı özyinelemeli olarak yazınız.

```
int oge_say(oge_gost L) {
    int miktar = 0;             // boş liste 0 öğe içerir
    for (; L; L = L->bag) {
        if (L->isaret) {        // alt liste var mı
            L->sayac = oge_say(L->L_bag);
            miktar += L->sayac; // alt listenin öğeleri de sayılır
        }
        miktar++;               // şu an gezilen öğe
    }
    return miktar;
}
```

Yordam aynı düzeyi bir `for` döngüsüyle gezer ve `miktar`'da öğeleri toplar. Boş listede döngü hiç dönmez ve 0 döner. Her öğe kendisi için bir sayılır. Alt liste öğesinde ayrıca alt listedeki öğe sayısı özyinelemeli olarak bulunur, öğenin `sayac` alanına yazılır ve toplama eklenir. Böylece T öğesinin sayacı yalnız altındaki öğeleri sayar; T öğesinin kendisi bir üst düzeyde sayılır.

(a, (b, c)) için T öğesinin sayacı 2, dönüş değeri 4'tür: *a*, T öğesi, *b* ve *c*. Deneme bunu ve daha karmaşık listeleri doğrulamıştır. *((a, b), ((c, d), e))* için sonuç 8'dir ve T öğelerinin sayaçları 2, 4 ve 2'dir. *(())* için sonuç 1, tek T öğesinin sayacı 0'dır. Sunumun son sınav sorusundaki listede 19 öğe bulunur.

9.10.2 Doğrusal formda yazdırma

Soru. Bellekte bulunan genelleştirilmiş listeyi doğrusal formda yazdıran özyinelemeli yordamı yazınız.

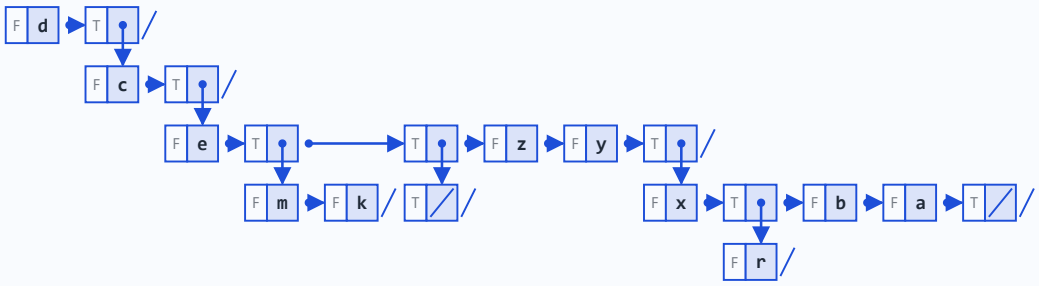
```
void dogrusal_yaz(oge_gost L) {
    printf("(");
    for (; L; L = L->bag) {
        if (L->isaret) dogrusal_yaz(L->L_bag);
        else printf("%c", L->veri);
        if (L->bag) printf(",");
    }
    printf(")");
}
```

Yordamın her çağrısı bir listeyi yazar, bu yüzden açılış ve kapanış parantezi her çağrıda vardır. Arada öğeler gezilir: alt liste öğesi aynı yordamla, atom ise yalnız değeriyle yazılır. Son öğe dışındakilerin arkasına virgül konur. Boş alt liste için çağrı `NULL` ile yapılır; döngü hiç dönmez ve `()` yazılır. Bu yüzden boş alt listeler kaybolmaz: *(a, (()), b)* aynen geri üretilir.

Bu yordam ile doğrusal form okuyucusu birbirinin tersidir. Okunan bir metin yazdırılınca aynı metin çıkmalıdır. Bu özellik 20.000 rastgele metinde sınanmış, hiçbir fark bulunmamıştır. Bir yapıyı metne çeviren ve metinden geri kuran yordam çifti, o yapıyı kaydetmenin, ağ üzerinden göndermenin ve sınamanın temelidir.

9.10.3 Liste çizimi

Soru. L = (d, (c, (e, (m, k), ()), z, y, (x, (r), b, a, ())))) doğrusal gösterimine karşılık gelen genelleştirilmiş liste görünümünü çiziniz. Aynı düzeyde peş peşe gelen öğeler bir satır boyunca izlenebilmelidir.



Şekil 9.15: Sınav sorusunun çözümü. Her satır bir düzeydir; aşağı inen oklar alt listelere iner. *(())* öğesinin altında, alt liste bağı boş olan tek bir T öğesi vardır. En alttaki düzeyin son öğesi *()*, alt liste bağı boş olan bir T öğesidir.

Çizim parantezler sayılarak yapılır. En dış listenin iki öğesi vardır: *d* ve bir alt liste. O alt listenin de iki öğesi vardır: *c* ve bir alt liste. Üçüncü düzey en kalabalıktır: *e*, *(m, k)*, *(())*, *z*, *y* ve son bir alt liste olmak üzere altı öğe. *(m, k)* iki atomlu bir alt listedir. *(())* ise tek öğeli bir listedir ve o tek öğe boş listedir. Çizimde bir T öğesinin altına alt liste bağı boş olan bir T öğesi iner. Dördüncü düzeyde *x*, *(r)*, *b*, *a* ve *()* vardır. *()* son öğedir ve bir T öğesidir; alt liste bağı boştur.

Bu soruda en sık yapılan hata, *()* ile *(())*'yi karıştırmaktır. *()* bir öğedir ve alt liste bağı `NULL` olan bir T kutusu çizilir. *(())* de bir öğedir, ama alt liste bağı boş değildir: bir öğelik bir listeyi gösterir ve o listenin tek öğesi *()*'dir. Öğeleri sayınca çizimin doğruluğu denetlenebilir: listede 11 atom ve 8 T öğesi, toplam 19 öğe vardır; derinlik 5'tir.

9.11 Karmaşıklık özeti ve ölçüm

Tablo 9.12 notta incelenen yordamların maliyetlerini toplar. *n* listedeki toplam öğe sayısıdır; alt listelerin öğeleri de sayılır.

Yordam	Zaman	Ek bellek	Açıklama
`kopyala`	*Θ(n)*	*O(derinlik + uzunluk)**	paylaşımsız, özyinelemesiz; kopya için ayrıca *Θ(n)*
`esitmi` / `esit`	*O(n)*	*O(derinlik + uzunluk)**	ilk farkta durur
`dolas_yaz`	*Θ(n)*	*O(derinlik + uzunluk)**	kök önde sırası
`derinlik` (özyinelemeli)	*Θ(n)*	*O(derinlik)*	düzey döngüyle, alt liste özyinelemeyle
`derinlik_d` (yinelemeli)	*Θ(n)*	*O(derinlik)*	yerel yığın; derinlik en çok 101
`devrik_al`	*Θ(n)*	*O(derinlik)*	yerinde, `malloc` yok
`dizi_ekle` / `dizi_sil`	*O(n)*	*O(1)*	kaydırma; sonda *O(1)*
`bl_ekle` (konum *i*)	*O(i)*	*O(1)*	kaydırma yok

Tablo 9.12: Yordamların zaman ve ek bellek maliyeti.

Yıldızlı satırlar bir inceliği belirtir. `bag` üzerinden de özyinelemeli çağrı yapan yordamlarda yığın derinliği yalnız iç içeliğe değil, aynı düzeydeki öğe sayısına da bağlıdır. *(a, (b, c))* gibi küçük bir listede bu fark görünmez, ama tek düzeyli bir milyon öğelik listede yığın bir milyon çağrı derinliğine iner. `bag` için döngü kullanmak, yani `derinlik` ve `devrik_al`'daki gibi yalnız alt listelere özyinelemeli inmek, ek belleği *O(derinlik)*'e indirir.

Zaman sütunu bir ölçümle de sınanmıştır. Her biri *k* atomlu *k* alt listeden oluşan listeler kurulmuş ($n = k^2 + k$) ve yordamlar `-O2` ile derlenmiş bir programda üçer kez çalıştırılmıştır. Tablo 9.13 tipik değerleri verir. Değerler makineye göre değişir; önemli olan büyüme biçimidir.

n	`kopyala`	`derinlik`	`devrik_al`
≈ 250 bin	12–17	≈ 2	≈ 2
≈ 1 milyon	45–57	5–8	7–9

n	`kopyala`	`derinlik`	`devrik_al`
≈ 4 milyon	180–220	28–29	31–34

Tablo 9.13: Ölçülen süreler (ms, üç çalıştırmanın yaklaşık değeri; makineye göre değişir).

n dört katına çıktıkça süreler de yaklaşık dört katına çıkar; bu doğrusal büyümedir. `kopyala` ötekilerden altı-yedi kat yavaştır, çünkü her öğe için `malloc` çağırır. Bellek ayırmak, bir bağı izlemekten çok daha pahalı bir işlemdir. Aynı $\Theta(n)$ sınıfındaki iki yordam arasında bu kadar fark olabilir.

9.12 Sık yapılan hatalar

1. **Paylaşımlı listeyi paylaşımsız varsaymak.** `kopyala` paylaşılan alt listeyi her gösterildiği yerde ayrı kopyalar; `liste_bosalt` ise onu iki kez iade eder. İkincisi tanımsız davranıştır.
2. **Özyinelemeli listede kopyalama ya da eşitlik çağırmak.** $C = (a, C)$ için çağrılar bitmez ve yığın taşar.
3. **`if`'lerde süslü parantezi unutmak.** Hatalı `dolas_yaz`'da son çağrı `NULL` için de çalışır ve program `NULL->bag` okurken çöker.
4. **Baş değişen yordama listeyi değerle vermek.** Ters çevirme `oge_gost *L` ister; alt listeler için de `&p->L_bag` verilir.
5. **Boş alt liste $()$ ile boş listeyi (`NULL`) karıştırmak.** $()$ bir öğedir: T öğesi vardır, yalnız `L_bag`'i `NULL`'dır.
6. **Küresel değişkenle derinlik saymak.** Yordam başka bir bileşenin parçası olunca ya da iç içe çağrılınca ortak değişken bozulur.
7. **Etiketi denetlemeden birliğin alanını okumak.** Atom öğesinin `L_bag`'i bir adres değildir.
8. **Yinelemeli gezinmede yığından alınan T öğesine dönmek.** Sürmesi gereken yer T'nin kardeşi, `T->bag`'dir.
9. **`realloc`'un sonucunu doğrudan aynı işaretçiye yazmak.** Bellek yetmezse eski blok kaybolur.

9.13 Alıştırmalar ve çözümleri

Sunumun alıştırmaları ve ipuçları aşağıda tam çözümlere dönüştürülmüştür. Bütün çözümler derlenmiş ve örnek listelerle denenmiştir.

9.13.1 Atom sayısı

Soru. Genelleştirilmiş listedeki atom sayısını bulan yordamı yazınız. Sunumun önerdiği yol: atomsa 1, alt listeyse özyinelemeli sonuç; `bag` boyunca topla.

```
int atom_say(oge_gost L) {
    int k = 0;
    for (; L; L = L->bag)
        k += L->isaret ? atom_say(L->L_bag) : 1;
    return k;
}
```

Yordam `oge_say`'ın daha sade bir kardeşidir. Alt liste öğesinin kendisi sayılmaz, yalnız içindeki atomlar sayılır. Sınav sorusundaki listede 11 atom, ağaç listesinde 13 atom (ağacın 13 düğümü) bulunmuştur. Süre $\Theta(n)$, ek bellek $O(derinlik)$'tir.

9.13.2 Doğrusal formdan liste kurmak

Soru. `"(a,(b,c))"` gibi bir doğrusal form metnini okuyup listeyi kuran yordamı yazınız. Sunumun önerdiği yol: `(` görünce alt liste kur, `,` ile ilerle, `)` ile dön.

Bu sorunun çözümü `ayristir` yordamıdır ve genelleştirilmiş listelerin tanıtıldığı kesimde ayrıntısıyla anlatılmıştır. Yordam metindeki konumu `const char **` ile taşır, her alt liste için kendini çağırır ve öğeleri `son` işaretçisiyle sona bağlar. Listeyi kuran yordam, sınavlarda kısa sorular için çok uygun bir konudur. Listeyi kuran, kopyalayan ve yazan yordamların nasıl yazılacağı üzerine sürekli düşünmek, bu konuyu kalıcı kılar.

9.13.3 Paylaşımı koruyan kopyalama

Soru. `kopyala` paylaşımı koruyacak biçimde nasıl değiştirilir? Sunumun önerdiği yol: kopyalanmış alt listeleri (eski adres → yeni adres) bir tabloda tut.

```
#define TABLO 100
typedef struct { oge_gost eski, yeni; } cift;

// t[0..*n-1]: kopyası alınmış alt listeler (eski baş -> yeni baş)
oge_gost kopyala_p(oge_gost L, cift t[], int *n) {
    oge_gost bas = NULL, *son = &bas, q;
    int k;
    for (; L; L = L->bag) {
        q = ogeal();
        q->isaret = L->isaret;
        if (!L->isaret) q->veri = L->veri;
        else {
            for (k = 0; k < *n && t[k].eski != L->L_bag; k++)
                ;                          // tabloda ara
            if (k < *n) q->L_bag = t[k].yeni;   // paylaşılan
            else {
                q->L_bag = kopyala_p(L->L_bag, t, n);
                if (*n < TABLO) {
                    t[*n].eski = L->L_bag;
                    t[*n].yeni = q->L_bag;
                    (*n)++;
                }
            }
        }
        *son = q;
        son = &q->bag;
    }
    *son = NULL;
    return bas;
}
```

Yordam aynı düzeyi döngüyle gezer ve öğeleri `son` işaretçisiyle sona ekler. Alt liste öğesinde önce tabloya bakılır. Alt listenin eski başı tabloda varsa, yeni öğe tablodaki kopyayı gösterir ve alt liste yeniden kopyalanmaz. Yoksa alt liste kopyalanır ve çift tabloya eklenir. *B = (A, A, ())* için yordam 7 öğe ayırmıştır (`kopyala` 11 ayırmıştı). Kopyanın ilk iki öğesi aynı listeyi gösterir ve `esit` kopyayı asıl listeye eşit bulur. Tabloda 3 çift vardır: *A*, *(b, c)* ve boş liste.

Çözümün sınırları açıktır. Tablo doğrusal arandığı için her alt liste için *O(t)* iş yapılır; çok sayıda alt listede bir hash tablosu daha uygundur. Tablo dolarsa paylaşım sessizce kaybolur. Özyinelemeli listelerde yordam yine sonlanmaz, çünkü alt listenin kaydı ancak kopyası bittikten sonra yapılır. Bunu çözmek için yeni başın, alt liste kopyalanmaya başlamadan önce tabloya yazılması gerekir; bu da başlık düğümlü bir gösterimle kolaylaşır.

9.13.4 Atom arama

Soru. Bir *x* atomunun listede geçip geçmediğini bulan `icerir(L, x)` yordamını yazınız. Sunumun önerdiği yol: atomda karşılaştır, alt listede özyinelemeli ara; ilk bulunuşta dön.

```
bool icerir(oge_gost L, char x) {
    for (; L; L = L->bag)
        if (L->isaret ? icerir(L->L_bag, x) : L->veri == x)
            return true;               // ilk bulunuşta dön
    return false;
}
```

Koşul işleci, öğenin türüne göre iki sorudan birini seçer: alt listede *x* var mı, ya da bu atom *x* mi. Cevap evetse arama hemen biter. Sınav sorusu listesinde *r* (en derindeki atomlardan biri) ve *d* bulunmuş, *q* bulunmamıştır. En kötü durumda, yani atom yoksa, bütün öğeler gezilir: *O(n)*.

9.13.5 Büyüyen dizi

Soru. Dizi tabanlı listeyi dolunca iki katına büyüyen biçimde yeniden yazınız. Sunumun önerdiği yol: `realloc(a, 2*kap*sizeof *a)`; ortalama *O(1)* ekleme.

Çözüm, liste ADT'sinin dizi gerçeklemesi olarak notun başında verilmiştir. `struct liste` heap'teki dizinin adresini, öğe sayısını ve kapasiteyi tutar; `liste_ekle` dizi doluyken kapasiteyi iki katına çıkarır. `realloc`'un sonucu ayrı bir değişkene alınır. İki katına büyütmenin sona eklemede amortize *O(1)* maliyet verdiği, geometrik toplamla kanıtlanmıştır.

9.14 Özet

- **Liste ADT** işlemlerin sözleşmesidir: `liste_ekle` için ön koşul *0 ≤ i ≤ n*, son koşul "`x` *i.* konumda, uzunluk *n + 1*"dir. **Opak tip** (`typedef struct liste Liste;`) gerçeklemeyi gizler; gerçekleme değişince kullanan kod değişmez.
- **Dizi** gerçeklemede erişim *O(1)*, ekleme ve silme kaydırma yüzünden *O(n)*'dir. Dinamik dizi iki katına büyütülünce sona ekleme amortize *O(1)* olur.
- **Bağlı** gerçeklemede konuma yürümek *O(i)*, bağlamak *O(1)*'dir. **İşaretçiye işaretçi** (`dugum_gost *pp`), başa ve araya eklemeyi tek koda indirir.
- **Genelleştirilmiş liste**nin öğeleri atom ya da alt listedir. Öğe *[isaret | veri/L_bag | bag]* biçimindedir; `isaret` etiket, `veri` / `L_bag` bir **etiketli union**dur. Liste **paylaşımlı** ya da **özyinelemeli** olabilir.
- Özyinelemeli kalıp: atomsa işle, alt listeyse in, sonra `bag` ile devam et. `kopyala`, `esitmi` ve `dolas_yaz` bu kalıpla *Θ(n)*'de çalışır ve *1 + n + s* çağrı yapar. Paylaşımsız ve özyinelemesiz listelerde doğrudurlar.
- **Derinlik**: boş liste 0, değilse 1 + alt listelerin en büyük derinliği. Özyinelemeli yazım kısa ve doğrudur; yinelemeli yazım yerel bir yığın ister, küresel değişken kullanmaz.
- **Ters çevirme** üç işaretçili yordama iki satır ekler: alt liste `devrik_al(&p->L_bag)` ile önce çevrilir. Yerinde çalışır, `malloc` yapmaz.
- Bir ağaç, ilk öğesi kök etiketi olan iç içe listelerle ve tek satırlık bir **doğrusal formla** saklanır. `dolas_yaz` böyle bir listede **kök önde** gezinme yapar. İkili ağaçta doğrusal form sol–sağ bilgisini kaybeder; L/R işareti ya da boş çocuk simgesi eklenmelidir.
- **Referans sayma**, başlık düğümündeki `sayac` ile paylaşımlı listeleri güvenle bırakır; ama döngüde sayaç sıfıra inmez. **Çöp toplama** (işaretle–süpür) döngüleri de toplar. C'de çöp toplama yoktur.

9.15 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

soyut veri tipi *abstract data type (ADT)* · liste ADT *list ADT* · arayüz *interface* · gerçekleme *implementation* · başlık dosyası *header file* · opak tip *opaque type* · eksik tip *incomplete type* · dinamik dizi *dynamic array* · kapasite *capacity* · amortize analiz *amortized analysis* · önbellek *cache* · işaretçiye işaretçi *pointer to pointer* · genelleştirilmiş liste *generalized list* · baş *head* · kuyruk *tail (of a generalized list)* · çok değişkenli polinom *multivariate polynomial* · union (birlik) *union (C)* · etiketli union *tagged union* · derinlik *depth* · gezinme *traversal* · kök önde *preorder* · ikili ağaç *binary tree* · alt ağaç *subtree* · yığın taşması *stack overflow* · kuyruk özyinelemesi *tail recursion* · başlık düğümü *header node* · referans sayısı *reference count* · çöp toplama *garbage collection* · bellek sızıntısı *memory leak* · sarkan işaretçi *dangling pointer*

9.16 Kendini Dene

1. *L = (a, ((b), c), ())* listesinin uzunluğu, derinliği, toplam öğe sayısı ve atom sayısı nedir? `kopyala(L)` kaç kez çağrılır?

> **Cevap**
>
> Üst düzeyde *a*, *((b), c)* ve *()* vardır; uzunluk 3'tür. *(b)*'nin derinliği 1, *((b), c)*'ninki 2, *L*'ninki 3'tür. Öğeler: üst düzeyde 3, *((b), c)* içinde 2, *(b)* içinde 1; toplam 6. Atomlar *a*, *b*, *c*: 3. Alt liste öğesi sayısı *s = 3* olduğundan `kopyala` *1 + 6 + 3 = 10* kez çağrılır.

2. *((a, b), c)* listesinin başı ve kuyruğu nedir? *(a)*'nın kuyruğu nedir?

Cevap

Baş, ilk öğe olan *(a, b)* listesidir. Kuyruk, kalan öğelerden oluşan *(c)* listesidir; *c* atomu değil. *(a)*'nın kuyruğu boş liste *()*'dır.

3. 4 → 7 listesinde `bl_ekle(&B, 0, 5)` çağrılıyor. Döngü kaç kez döner, `pp` neyi gösterir ve `*pp = yeni` hangi işaretçiyi değiştirir?

Cevap

i = 0 olduğu için döngü hiç dönmez ve `pp`, `&B.bas`'ta kalır. `yeni->bag = *pp` 5'i 4'e bağlar; `*pp = yeni` ise `B.bas`'ı 5'e çevirir. Liste 5 → 4 → 7 olur, `n = 3`. Başa ekleme için ayrı bir dal gerekmez.

4. Boş bir `DiziListe`'ye on kez `dizi_ekle(&L, 0, v)` yapılıyor. Toplam kaç kaydırma yapılır? Aynı on değer hep sona eklenseydi?

Cevap

k. eklemede listede *k − 1* öğe vardır ve hepsi kayar: *0 + 1 + … + 9 = 45*. Sona eklemede *i = n* olduğu için döngü hiç dönmez: 0 kaydırma.

5. `esitmi` ile *(a, (b))* ve *(a, b)* karşılaştırılıyor. Sonuç nedir, fonksiyon kaç kez çağrılır?

Cevap

1. çağrıda *a*/*a* eşittir ve kalanlar için 2. çağrı yapılır. 2. çağrıda bir öğe T, öteki atomdur; etiketler farklı olduğu için blok atlanır ve `FALSE` döner. Sonuç `FALSE`, toplam 2 çağrıdır.

6. `devrik_al(&L)` *L = (a, (b, (c, d)), e)* listesini neye dönüştürür? Kaç `malloc` yapılır?

Cevap

Her düzey ters çevrilir: *(c, d) → (d, c)*, *(b, (c, d)) → ((d, c), b)*, üst düzey → *(e, ((d, c), b), a)*. Yordam yalnız bağları değiştirir; hiç `malloc` yapılmaz.

7. `devrik_al`'ın içinde `devrik_al(&p->L_bag)` yerine yeni yerel bir değişkenle `oge_gost t = p->L_bag; devrik_al(&t);` yazılırsa *(a, (b, c))* için ne olur?

Cevap

İç çağrı alt listeyi doğru çevirir ve yeni başı *c*'yi `t`'ye yazar. Ama T öğesinin `L_bag` alanı güncellenmez; hâlâ *b*'yi gösterir. *b*'nin bağı artık `NULL` olduğundan liste *((b), a)* görünür; *c*'ye erişilemez ve sızar. Düzeltme ya `p->L_bag = t` eklemek ya da alanın adresini doğrudan vermektir.

8. *A = (x)* ve *B = (A, A, A)* referans sayaçlı başlık düğümleriyle kurulmuştur. Başta *A*'nın sayacı kaçtır? `liste_birak(&B)`'den sonra kaç olur ve *A* bırakılır mı?

Cevap

A'yı `A` değişkeni ve *B*'nin üç T öğesi gösterir: sayaç 4'tür. *B* bırakılırken üç T öğesi sayacı 4 → 3 → 2 → 1'e indirir. Sayaç sıfır olmadığı için *A* bırakılmaz; `A` değişkeni hâlâ onu gösterir.

9. *(A, (B, (E, H, (I, J, K)), F), (C, G), D)* listesi için `dolas_yaz` ve `agac_yaz` ne yazar?

Cevap

`dolas_yaz` atomları kök önde sırasıyla yazar: `ABEHIJKFCGD`. `agac_yaz` ağacın doğrusal formunu yazar: `A(B(E(H,I(J,K)),F),C(G),D)`.

9.17 İleri okuma

- E. Horowitz, S. Sahni ve S. Anderson-Freed, *Fundamentals of Data Structures in C* (2. baskı), Bölüm 4: genelleştirilmiş listelerin, çok değişkenli polinom gösteriminin ve referans sayaçlı başlık düğümünün bu notta izlenen kaynağıdır.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms*, Bölüm 10 ve Bölüm 17 (amortize analiz, dinamik tablolar): liste gerçeklemelerini ve dinamik tablonun iki katına büyütülmesinin amortize maliyetini anlatır.
- J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I", *Communications of the ACM* 3(4), 1960: LISP'i, `car` / `cdr` işlemlerini ve çöp toplamayı tanıtan makaledir.
- R. Johnsonbaugh, *Discrete Mathematics*: özyineleme, tümevarım ve ağaçlar için gereken matematik altyapısını verir.

9.18 Kaynaklar

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, Bölüm 4.9 (genelleştirilmiş listeler, referans sayma).
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, MIT Press, Bölüm 10 (liste gerçeklemeleri).
3. R. Johnsonbaugh, *Discrete Mathematics*.
4. J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I", *Communications of the ACM*, 3(4), 184–195, 1960.

KISIM III

Ağaçlar

Bölüm 10–13

KISIM III · BÖLÜM 10

Ağaçlar

Dizi, bağlı liste, yığın ve kuyruk doğrusal yapılardır: her öğenin en çok bir öncülü ve bir ardılı vardır. Gerçek dünyadaki verilerin önemli bir kısmı ise hiyerarşiktir. Bir klasör birçok klasör içerir, bir yönetici birçok çalışana bağlıdır, bir aritmetik işlem iki alt işlemin sonucunu birleştirir. Bu ilişkileri doğal biçimde tutan yapı ağaçtır. Not önce ağaç terimlerini bir örnek üzerinde tanımlar. Sonra ikili ağacı ve onun düğüm sayısına ilişkin temel sonuçlarını kanıtlar. Ardından ikili ağacın bellekte iki gösterimini, diziyi ve işaretçili düğümleri, C kodlarıyla kurar. Notun büyük bölümü gezinmeye ayrılmıştır: kök önde, kök ortada ve kök sonda gezinmeler önce özyinelemeli, sonra yığınla yinelemeli yazılır; düzey düzey gezinme için kuyruk kullanılır. Son kesimler ağacı kopyalayan, iki ağacı karşılaştıran ve ağacın aynasını alan kısa özyinelemeli yordamları ve gezinme sıralarından ağacı yeniden kurmayı ele alır. Her yordamın zaman ve bellek maliyeti gerekçesiyle verilir.

Kod (C): `github.com/sercankulcu/data-structures-and-algorithms-c/Bolum10`

ÖĞRENME ÇIKTILARI

Terminolojiyi kullanmak
Kök, ebeveyn, çocuk, kardeş, ata, soy, derece, düzey, derinlik ve yaprak kavramlarını bir ağaç üzerinde göstermek; derece ile derinliği ayırmak.

İkili ağacı çözümlemek
Bir düzeydeki en fazla düğüm sayısının 2^{i-1}, k derinlikli ağaçtaki en fazla düğüm sayısının $2^k - 1$ olduğunu kanıtlamak; mükemmel, tam ve tam olmayan ağaçları ayırmak; $n_0 = n_2 + 1$ eşitliğini kullanmak.

İki gösterimi kurmak
İkili ağacı diziyle (ebeveyn $\lfloor i/2 \rfloor$, çocuklar $2i$ ve $2i + 1$) ve `agac_gost` işaretçili düğümlerle göstermek; iki gösterimin bellek maliyetini karşılaştırmak.

Gezinmeleri yazmak
Kök önde, kök ortada ve kök sonda gezinmeleri özyinelemeli ve yığınla yinelemeli yazmak; kuyrukla düzey düzey gezmek; bir gezinmenin çıktısını elle bulmak.

Ağaç yordamları kurmak
Uç düğümleri yazan, ağacı kopyalayan, iki ağacın eş biçimli olup olmadığını denetleyen ve sol–sağ çocukları değiştiren özyinelemeli yordamlar yazmak; bellek sızıntısı bırakmadan ağacı silmek.

Maliyeti hesaplamak
Her yordamın $O(n)$ zamanını ve derinlikle orantılı $O(h)$ belleğini gerekçelendirmek; dengeli ve çarpık ağaçta bu sınırların ne olduğunu söylemek.

10.1 Ağaç veri yapısı

Bağlı listede her düğüm bir sonraki düğümü gösterir. Düğüm bir yerine birden çok sonraki düğüm gösterebilseydi ne olurdu? Liste dallanır ve ortaya ağaç çıkar. Bu kesim ağacı tanımlar, kullanıldığı yerleri sayar ve ağacı daha genel bir yapı olan çizgeden ayıran özelliği gösterir.

10.1.1 Tanım ve kullanım alanları

Günlük hayatta ağaç biçimli pek çok düzen vardır. Soy ağacında her kişinin altında çocukları yer alır. Bir kurumun yönetim şemasında genel müdürün altında müdürler, onların altında şefler bulunur. Bilgisayardaki dosya sistemi de bir ağaçtır: kök klasörün içinde klasörler, onların içinde başka klasörler ve dosyalar vardır. Her dosyanın tek bir üst klasörü olur. Bu düzenlerin ortak yanı, en üstte tek bir başlangıç noktasının bulunması ve her öğenin tam olarak bir üst öğeye bağlanmasıdır.

Bilgisayar biliminde ağaç, arama ve sıralama yapılarının, veri sıkıştırmanın ve derleyicilerin temelidir. Derleyici bir programı okurken önce söz dizimini çözümler (*syntax analysis*) ve programın yapısını bir ağaç olarak kurar. Kod iyileştirme ve ara kod üretimi bu ağaç üzerinde çalışır. Aritmetik ifadeler, karar süreçleri, oyunlardaki hamle seçenekleri ve veri tabanı dizinleri de ağaçla tutulur.

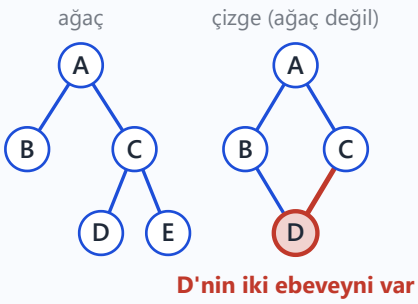
Tanım 10.1

Ağaç (*tree*), bir ya da daha çok düğümden oluşan sonlu bir kümedir. Düğümlerden biri **kök** (*root*) olarak ayrılmıştır. Kalan düğümler $n \geq 0$ tane ayrık kümeye bölünür: $T_1, \ldots, T_n$. Bu kümelerin her biri yine bir ağaçtır ve kökün **alt ağacı** (*subtree*) adını alır.

Tanım özyinelemelidir: ağacı, yine ağaç olan parçalar cinsinden tanımlar. Kökü çıkarınca geriye birkaç küçük ağaç kalır; onların kökleri çıkarılınca daha küçük ağaçlar kalır. Bu süreç tek düğümlü ağaçlara kadar iner. Tanımın bu biçimi, ağaç üzerinde çalışan yordamların neden doğal olarak özyinelemeli yazıldığını açıklar. Bir yordam kökte küçük bir iş yapar, sonra aynı yordamı alt ağaçların her biri için çağırır. Alt ağaçlar ayrık olduğu için hiçbir düğüm iki kez işlenmez. Bu nottaki bütün gezinme, kopyalama ve karşılaştırma yordamları bu kalıba uyar.

10.1.2 Ağaç ile çizge

Düğümler ve onları bağlayan kenarlardan oluşan en genel yapı **çizgedir** (*graph*). Ağaç, çizgenin özel bir türüdür. Ağaçta kök dışındaki her düğümün **tek bir ebeveyni** vardır. Çizgede ise bir düğüme birden çok düğümden kenar gelebilir. Şekil 10.1 iki yapıyı yan yana gösterir. Soldaki yapıda A'nın altında B ve C, C'nin altında D ve E vardır; her düğüme yukarıdan tek bir kenar iner. Sağdaki yapıda D'ye hem B'den hem C'den kenar gelir. D'nin iki ebeveyni olduğu için bu yapı ağaç değildir.



Şekil 10.1: Solda beş düğümlü bir ağaç, sağda ağaç olmayan bir çizge. Sağdaki yapıda D'ye iki ayrı yoldan ulaşılır; D'nin iki ebeveyni vardır.

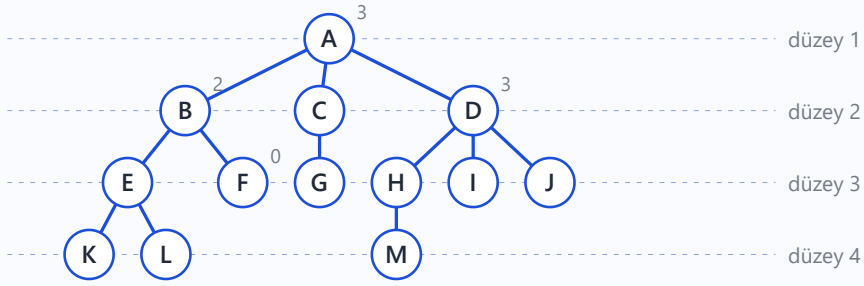
Tek ebeveyn koşulu iki önemli sonuç doğurur. Birincisi, kökten herhangi bir düğüme giden yol tektir. İkincisi, ağaçta döngü yoktur: bir düğümden başlayıp kenarları izleyerek aynı düğüme geri dönülemez. n düğümlü bir ağaçta kök dışındaki her düğüme tam olarak bir kenar girdiği için kenar sayısı $n - 1$'dir. İkili ağaçların düğüm sayısı ilişkileri de bu sayıya dayanır.

Sık yapılan hata

Ağaç yordamlarını çizgeye uygulamak. Ağaçta bir düğüme tek yoldan ulaşıldığı için ziyaret işareti gerekmez. Çizgede ise işaret tutulmazsa düğümler tekrar tekrar işlenir; döngü varsa yordam hiç bitmez.

10.2 Terminoloji

Ağaçlarla ilgili terimlerin çoğu soy ağacından gelir: ebeveyn, çocuk, kardeş, ata, torun. Kalan terimler (derece, düzey, derinlik) düğümleri ve ağacı sayısal olarak betimler. Bu kesim bütün terimleri tek bir örnek ağaç üzerinde tanımlar. Şekil 10.2 bu ağacı gösterir: on üç düğüm, dört düzey. Düğümlerin sağ üstündeki küçük sayılar o düğümün derecesidir.



Şekil 10.2: A–M harfli on üç düğümlü genel ağaç. Kök A'dır. A'nın üç çocuğu (B, C, D), B'nin iki çocuğu (E, F), D'nin üç çocuğu (H, I, J) vardır. Yatay çizgiler düzeyleri gösterir: kök 1. düzeydedir, K, L ve M 4. düzeydedir.

10.2.1 Ebeveyn, çocuk, kardeş, ata ve soy

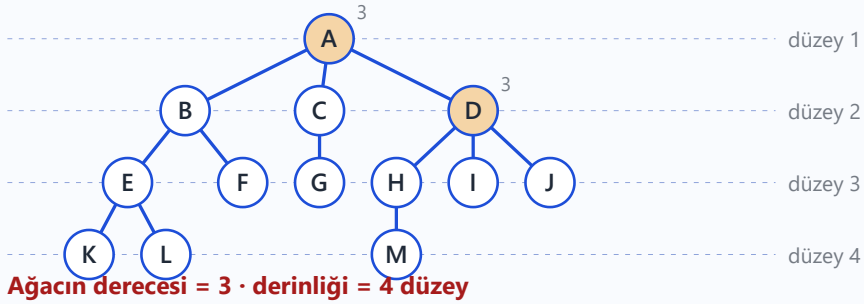
Ağacın en üstündeki, ebeveyni olmayan düğüm **köktür**. Örnek ağaçta kök A'dır. Alt ağaçları olan bir düğüm, o alt ağaçların köklerinin **ebeveynidir** (*parent*). Alt ağaçların kökleri de o düğümün **çocuklarıdır** (*child*). B, E ve F'nin ebeveynidir; E ve F, B'nin çocuklarıdır. Aynı ebeveynin çocukları birbirinin **kardeşidir** (*sibling*): E ile F kardeştir, H, I ve J de kardeştir. Terminoloji soy ağacındaki gibi büyükbaba ve torunla kadar genişler. D, M'nin büyükbabasıdır, çünkü M'nin ebeveyni H'nin ebeveyni D'dir.

Bir düğümün **ataları** (*ancestor*), kökten o düğüme giden yol (*path*) üzerindeki bütün düğümlerdir. M'nin ataları A, D ve H'dir. Kök hiçbir düğümün soyundan değildir; kökün atası yoktur. Bir düğümün alt ağacındaki bütün düğümler onun **soyudur** (*descendant*). B'nin soyu E, F, K ve L'dir. Çocukların çocukları torundur: E, F, G, H, I ve J, A'nın torunlarıdır. İki tanım birbirinin tersidir: X, Y'nin atasıysa Y de X'in soyundandır. Bu ilişki gezinme yordamlarında önemlidir. Bir düğümü işleyen özyinelemeli yordam, o düğümün bütün soyunu da işler, atalarına ise hiç dokunmaz.

10.2.2 Derece, yaprak, düzey ve derinlik

Bir düğümün **derecesi** (*degree*), o düğümün alt ağaç sayısıdır, yani çocuk sayısıdır. A'nın derecesi 3, B'nin 2, C'nin 1, F'nin 0'dır. Derecesi 0 olan düğüme **yaprak** (*leaf*) ya da **uç düğüm** (*terminal node*) denir. Örnek ağacın yaprakları K, L, F, G, M, I ve J'dir. En az bir çocuğu olan düğümler **iç düğümdür** (*internal node*): A, B, C, D, E ve H.

Düğümlerin **düzeyi** (*level*) kökten başlayarak numaralanır. Bu derste kök 1. düzeydedir; bir düğümün düzeyi ebeveyninin düzeyinden bir fazladır. B, C ve D 2. düzeyde, E, F, G, H, I ve J 3. düzeyde, K, L ve M 4. düzeydedir. Ağacın kendisi için de iki sayı tanımlanır. **Ağacın derecesi**, düğüm derecelerinin en büyüğüdür; örnek ağaçta 3'tür (A ve D). **Ağacın derinliği** (*depth*), düğümlerin en büyük düzey numarasıdır; örnek ağaçta 4'tür. Şekil 10.3 iki sayıyı aynı çizimde gösterir.



Şekil 10.3: Ağacın derecesi, derecesi en büyük düğümlerin (A ve D) derecesidir: 3. Ağacın derinliği en büyük düzey numarasıdır: 4. İki sayı birbirinden bağımsızdır.

Derece ve derinlik farklı soruların cevabıdır. Derece ağacın ne kadar *geniş* dallandığını, derinlik ne kadar *aşağı* indiğini ölçer. Bir zincir gibi uzanan ağacın derecesi 1, derinliği düğüm sayısı kadardır. Kökün altına doğrudan yüz yaprak bağlanmış bir ağacın derecesi 100, derinliği 2'dir. Gezinme yordamlarının bellek maliyeti derinliğe bağlıdır; ikili ağaçlarda derece zaten en çok 2'dir.

Örnek 10.1

Şekil 10.2 üzerinde H düğümü için bütün terimler şöyledir. H'nin ebeveyni D'dir; tek çocuğu M'dir. Kardeşleri I ve J'dir. Ataları kökten başlayarak A ve D'dir. Soyu yalnız M'dir. Derecesi 1'dir, dolayısıyla H bir iç düğümdür. Düzeyi 3'tür: A 1., D 2., H 3. düzeydedir. H'yi kök kabul eden alt ağaç iki düğümlüdür (H ve M) ve derinliği 2'dir.

Sık yapılan hata

Derece ile derinliği aynı tanımla vermek: "ağacın derinliği düğüm derecelerinin en büyüğüdür" gibi. Bu cümle ağacın derecesini tanımlar, derinliğini değil. Örnek ağaçta iki değer 3 ve 4'tür; bir zincirde 1 ve n'dir. Sınav sorularında ikisi ayrı ayrı sorulur ve birbirinin yerine kullanılamaz.

10.2.3 Derinlik tanımları: kitaplar farklı sayar

Düzey ve derinlik kavramları kaynaktan kaynağa bir kayma gösterir. Bu dersin ana kaynağı Horowitz, Sahni ve Anderson-Freed'in kitabıdır. Bu kitapta kök 1. düzeydedir ve ağacın derinliği en büyük düzeydir, yani kökten en uzak yaprağa giden yoldaki *düğüm* sayısıdır. Cormen ve arkadaşlarının kitabı (CLRS) ile Sedgewick'in kitabı ise kökün derinliğini 0 kabul eder. Bu kitaplarda ağacın **yüksekliği** (*height*) kökten en uzak yaprağa giden yoldaki *kenar* sayısıdır. Aynı ağaç için iki tanım bir farkla ayrılır. Tablo 10.1 örnek ağaçtaki değerleri karşılaştırır.

Kavram	Bu ders (Horowitz–Sahni)	CLRS / Sedgewick
Kökün düzeyi	1	0
M'nin düzeyi	4	3
Ağacın derinliği / yüksekliği	4 (derinlik)	3 (yükseklik)
Tek düğümlü ağaç	1	0
Boş ağaç	0	−1

Tablo 10.1: İki kaynak geleneğinde düzey ve derinlik (Şekil 10.2 üzerindeki değerler).

İki tanımdan biri doğru, öteki yanlış değildir; yalnız ölçülen şey farklıdır. Biri düğüm, öteki kenar sayar. Bir yoldaki kenar sayısı her zaman düğüm sayısından bir eksiktir. Bu yüzden formüller de bir kayar: bu derste k derinlikli ikili ağaçta en çok $2^k - 1$ düğüm vardır; yükseklikle yazıldığında aynı sınır $2^{h+1} - 1$ olur. Sınavda ya da bir kaynakta formül kullanılırken önce hangi tanımın geçerli olduğuna bakılmalıdır.

Bu notta derinlik bir fonksiyon olarak da kullanılır: *derinlik(NULL) = 0*, tek düğümlü ağacın derinliği 1'dir. Boş olmayan bir ağacın derinliği, iki alt ağacın derinliklerinden büyük olanın bir fazlasıdır. Bu tanım doğrudan bir C fonksiyonuna dönüşür:

```
int derinlik(agac_gost L) {      /* boş ağaç 0, tek düğüm 1 */
  if (!L) return 0;
  int s = derinlik(L->solcocuk);
  int r = derinlik(L->sagcocuk);
  return 1 + (s > r ? s : r);
}
```

Fonksiyon önce boş ağacı yakalar ve 0 döndürür. Boş olmayan ağaçta iki alt ağacın derinliğini özyinelemeli olarak hesaplar ve büyük olana 1 ekler. Eklenen 1 kökün kendisidir. Bütün düğümlere bir kez uğradığı için zamanı $O(n)$'dir. Sunumdaki ikili ağaçlarla yapılan denemelerde fonksiyon el hesabıyla aynı değerleri verdi; boş ağaç için 0 döndürdü.

10.3 İkili ağaç

Genel ağaçta bir düğümün çocuk sayısı sınırsızdır. Bu esneklik bellekte bir sorun yaratır: her düğüme kaç işaretçi alanı ayrılacağı önceden bilinmez. Çocuk sayısı en çok iki ile sınırlanınca hem bellek düzeni sabitlenir hem de yordamlar yalınlaşır. Bu kesim ikili ağacı tanımlar ve onun düğüm sayısıyla derinliği arasındaki ilişkileri kanıtlar.

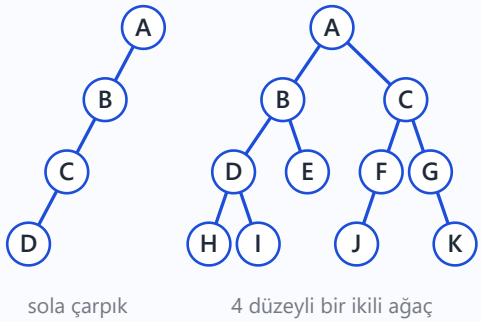
10.3.1 Tanım

Tanım 10.2

İkili ağaç (*binary tree*), sonlu bir düğüm kümesidir. Bu küme ya boştur ya da bir kökten ve **sol alt ağaç** ile **sağ alt ağaç** adı verilen, birbirinden ayrık iki ikili ağaçtan oluşur.

Tanımın iki ayrıntısı genel ağaçtan farklıdır. Birincisi, ikili ağaç boş olabilir; genel ağaç tanımında en az bir düğüm vardır. Boş ağaç sayesinde "sol alt ağaç yok" durumu "sol alt ağaç boş" diye ifade edilir. Bütün yordamlar da tek bir taban durumla, boş ağaçla, biter. İkincisi, sol ile sağ ayrıdır. Tek çocuklu bir düğümde çocuğun solda mı sağda mı olduğu önemlidir. Kökü A, tek çocuğu B olan iki ağaç düşünelim: birinde B soldadır, ötekinde sağdadır. Genel ağaç olarak ikisi aynı ağaçtır; ikili ağaç olarak iki farklı ağaçtır. Bu yüzden düğüm dereceleri 0, 1 ya da 2 olabilir, ama 1 dereceli bir düğümün çocuğunun yönü de bilgi taşır.

Kümeler arasında bir içerme ilişkisi vardır: her ikili ağaç (boş olanı dışında) bir ağaçtır, her ağaç da bir çizgedir. Şekil 10.4 iki ikili ağaç gösterir. Soldaki ağaçta her düğümün yalnız sol çocuğu vardır; bu tür ağaca **çarpık ağaç** (*skewed tree*) denir. Sağdaki ağaç dört düzeylidir ve düğümler düzeylere dengeli dağılmıştır.



Şekil 10.4: İki ikili ağaç. Solda dört düğümlü sola çarpık ağaç: her düğümün yalnız sol çocuğu var, derinlik 4. Sağda on bir düğümlü, dört düzeyli bir ikili ağaç: derinlik yine 4, ama düğüm sayısı neredeyse üç katı.

Şekildeki iki ağacın derinliği aynıdır; düğüm sayıları ise 4 ve 11'dir. Aynı derinliğe çok farklı sayıda düğüm sığabilir. Derinlik, gezinme yordamlarının bellek maliyetini belirlediği için "n düğümlü bir ağacın derinliği en az ve en çok ne olabilir?" sorusu önemlidir. Sonraki iki alt kesim bu sorunun cevabını verir.

10.3.2 Bir düzeyde ve bir ağaçta en fazla kaç düğüm

İkili ağaçta her düğümün en çok iki çocuğu olduğu için her düzeydeki düğüm sayısı bir öncekinin en çok iki katıdır. Bu gözlem iki sınır verir.

Teorem 10.1 (Düzey ve ağaç başına en fazla düğüm)

(i) Bir ikili ağacın i. düzeyinde en çok 2^{i-1} düğüm bulunur ($i \geq 1$). (ii) Derinliği k olan bir ikili ağaçta en çok $2^k - 1$ düğüm bulunur ($k \geq 1$).

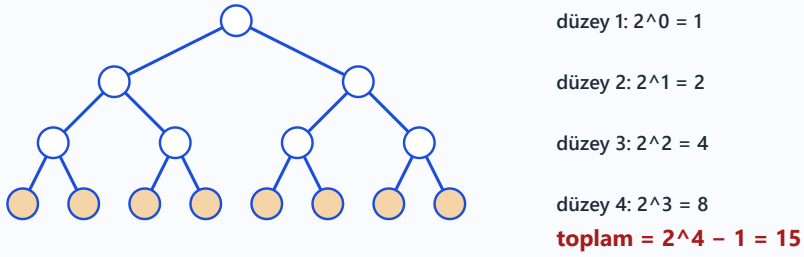
Kanıt. (i) i üzerinde tümevarımla kanıtlanır. Taban durum $i = 1$'dir: 1. düzeyde yalnız kök vardır ve $2^0 = 1$'dir. Tümevarım adımı için $(i-1)$. düzeyde en çok 2^{i-2} düğüm bulunduğunu varsayalım. i. düzeydeki her düğüm, $(i-1)$. düzeydeki bir düğümün çocuğudur ve her düğümün en çok iki çocuğu vardır. Bu yüzden i. düzeyde en çok $2 \cdot 2^{i-2} = 2^{i-1}$ düğüm bulunur.

(ii) Derinliği k olan ağaçta düğümler 1'den k'ya kadar olan düzeylere dağılmıştır. (i) her düzeyin sınırını verir. Toplam düğüm sayısı en çok

$$\Sigma_{i=1..k}\, 2^{i-1} = 2^0 + 2^1 + \ldots + 2^{k-1} = 2^k - 1$$

olur. Son eşitlik geometrik toplamdır: toplam S ile gösterilirse $2S - S = 2^k - 2^0$ bulunur. ■

Ders kitabı bu sonucu “lemma” olarak adlandırır. Lemma, doğruluğu kolayca görülen ve daha büyük bir teoremin kanıtında basamak olarak kullanılan küçük bir önermedir. Şekil 10.5 dört düzeyli ağaçta düzey düzey sayıları gösterir: 1, 2, 4 ve 8 düğüm, toplam 15. Derinlik 5 olsaydı beşinci düzeye 16 düğüm daha eklenir, toplam 31 olurdu.



Şekil 10.5: Derinliği 4 olan ve bütün düzeyleri dolu ikili ağaç. Düzeylerdeki düğüm sayıları 2^0, 2^1, 2^2 ve 2^3'tür; toplam $2^4 - 1 = 15$.

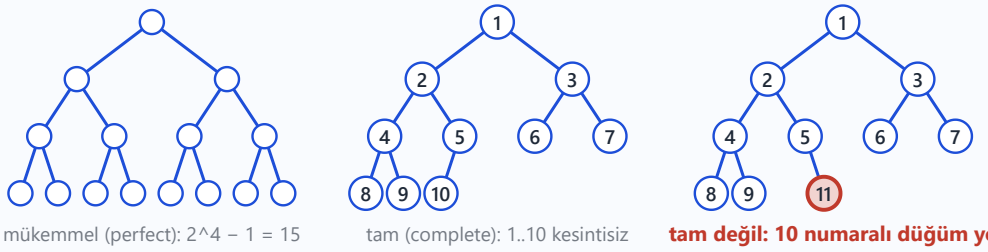
10.3.3 Mükemmel ve tam ikili ağaç

Teorem 10.1'in sınırına ulaşan ağaçlar özel bir ad alır. Bir de bu sınıra en çok yaklaşan, son düzeyi eksik olabilen ağaçlar vardır. İki kavram adlarının benzerliği yüzünden sık karıştırılır.

Tanım 10.3

Derinliği k olan ve tam olarak $2^k - 1$ düğümü bulunan ikili ağaca **mükemmel ikili ağaç** (*perfect binary tree*) denir: bütün düzeyleri doludur. Mükemmel ağacın düğümleri kökten başlayarak, her düzeyde soldan sağa 1, 2, 3, … diye numaralanır. Bir ikili ağacın n düğümü, aynı derinlikteki mükemmel ağacın 1–n numaralı düğümleriyle birebir eşleşebilir. Bu durumda ağaca **tam ikili ağaç** (*complete binary tree*) denir.

Numaralandırmanın kuralı basittir: bir düğümün numarası i ise sol çocuğunun numarası $2i$, sağ çocuğunun numarası $2i + 1$'dir. Kök 1'dir; kökün çocukları 2 ve 3'tür; 2'nin çocukları 4 ve 5'tir, 3'ün çocukları 6 ve 7'dir. Tam ağaç, bu numaralar 1'den n'ye kadar atlamasız kullanıldığında elde edilen ağaçtır. Tanımın gündelik dildeki karşılığı şudur: son düzey dışındaki bütün düzeyler doludur ve son düzeydeki düğümler soldan sağa, arada boşluk bırakmadan dizilmiştir.



Şekil 10.6: Solda 15 düğümlü mükemmel ağaç. Ortada 10 düğümlü tam ağaç: numaralar 1'den 10'a kesintisiz. Sağda tam olmayan bir ağaç: 11 numaralı düğüm var, 10 numaralı düğüm yok.

Şekil 10.6'daki sağdaki ağaç on düğümlüdür, ama tam değildir. 5 numaralı düğümün sol çocuğu yoktur, sağ çocuğu vardır. Bu yüzden numaralar 1–9 ve 11'dir; 10 atlanmıştır. Ortadaki ağaçta ise 5'in yalnız sol çocuğu vardır ve numaralar 1'den 10'a kesintisizdir. Her mükemmel ağaç aynı zamanda tamdır; tersi doğru değildir.

Tam ağacın önemi iki noktadan gelir. Birincisi, tam ağaç bir diziye boşluksuz yerleşir: i numaralı düğüm dizinin i. gözüne konur ve hiçbir göz boş kalmaz. İkincisi, n düğümlü ikili ağaçlar arasında derinliği en küçük olan tam ağaçtır. Heap yapısı bu iki özelliğin üzerine kurulur.

Sık yapılan hata

“Tam ağaç” ile “her düğümün ya 0 ya 2 çocuğu olan ağaç” kavramını karıştırmak. İkinci kavramın adı dolu ikili ağaçtır (*full binary tree*); bütün düzeyleri dolu ağaç ise mükemmel ikili ağaçtır. Ortadaki ağaçta 5 numaralı düğümün tek çocuğu vardır, yine de ağaç

tamdır. Bir ağacın tam olup olmadığı düğüm numaralarına bakılarak anlaşılır: 1'den n'ye kadar her numara kullanılmış mı?

10.3.4 Derinlik ve düğüm sayısı ilişkileri

k derinlikli bir ikili ağaçta her düzeyde en az bir düğüm vardır; yoksa derinlik k olmazdı. Bu yüzden düğüm sayısı en az k'dır. Teorem 10.1'den en çok $2^k - 1$'dir. Birlikte yazılınca

$$k \le n \le 2^k - 1$$

elde edilir. Bu eşitsizlik derinlik cinsinden de yazılabilir. Sağ taraftan $n + 1 \le 2^k$, yani $\lg(n+1) \le k$ çıkar; k tam sayı olduğu için $\lceil \lg(n+1) \rceil \le k$ olur. Sol taraf ise $k \le n$'dir. Sonuç olarak

$$\lceil \lg(n+1) \rceil \le k \le n$$

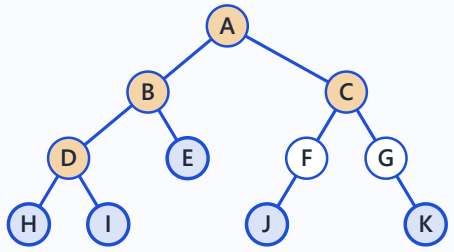
bulunur. Alt sınıra tam ağaç ulaşır: tam ağacın derinliği $\lceil \lg(n+1) \rceil = \lfloor \lg n \rfloor + 1$'dir. Üst sınıra çarpık ağaç ulaşır: her düzeyde tek düğüm vardır ve derinlik n'dir. Örneğin 20 düğümlü bir ikili ağacın derinliği 5 ile 20 arasında olabilir; 20 düğümlü tam ağacın derinliği $\lceil \lg 21 \rceil = 5$'tir. Bu iki uç, "dengeli ağaçta $O(\lg n)$, çarpık ağaçta $O(n)$" biçimindeki bütün maliyet sonuçlarının kaynağıdır.

Düğüm sayısıyla ilgili ikinci ilişki yaprakları ve iki çocuklu düğümleri bağlar. İkili ağaçta derecesi 0, 1 ve 2 olan düğümlerin sayıları sırasıyla n_0, n_1 ve n_2 ile gösterilsin.

Teorem 10.2 (Yaprak sayısı)

Boş olmayan her ikili ağaçta $n_0 = n_2 + 1$'dir: yaprak sayısı, iki çocuklu düğüm sayısından bir fazladır.

Kanıt. Düğüm sayısı iki yoldan sayılır. Her düğümün derecesi 0, 1 ya da 2 olduğu için $n = n_0 + n_1 + n_2$'dir. Kök dışındaki her düğüme yukarıdan tam olarak bir kenar girer; kenar sayısı $n - 1$'dir. Kenarlar düğümlerden çıkarak da sayılabilir: 1 dereceli düğümden bir, 2 dereceli düğümden iki kenar çıkar. Bu yüzden $n - 1 = n_1 + 2n_2$'dir. İlk eşitlikten ikincisi çıkarılırsa $1 = n_0 - n_2$, yani $n_0 = n_2 + 1$ bulunur. ■



n = 11: n0 = 5 yaprak, n1 = 2, n2 = 4 → 5 = 4 + 1

Şekil 10.7: On bir düğümlü ağaçta iki çocuklu düğümler (A, B, D, C) koyu, yapraklar (H, I, E, J, K) çerçeveli gösterilmiştir. $n_0 = 5$, $n_1 = 2$, $n_2 = 4$ ve $5 = 4 + 1$.

Şekil 10.7 eşitliği bir örnekle doğrular. Tek çocuklu iki düğüm (F ve G) sonucu etkilemez: bir zincire tek çocuklu düğüm eklemek yaprak sayısını da iki çocuklu düğüm sayısını da değiştirmez. Yeni bir yaprak ancak var olan bir düğüme ikinci çocuk eklenerek kazanılır; bu işlem n_2'yi de bir artırır.

Teoremin C gerçeklemesi için önemli bir sonucu vardır. Bağlı gösterimde her düğüm iki işaretçi alanı taşır; n düğümde toplam $2n$ alan vardır. Bunların $n - 1$'i bir kenarı, yani bir çocuğu gösterir. Geriye kalan $2n - (n - 1) = n + 1$ alan `NULL`'dır. On bir düğümlü örnekte 22 alanın 10'u dolu, 12'si boştur. Boş işaretçilerin sayısı dolu olanlardan fazladır. Bu boş alanlar, kök ortada sıradaki önceki ve sonraki düğümü gösterecek biçimde kullanılırsa **kılavuzlu ikili ağaç** (*threaded binary tree*) elde edilir. Aynı sayı, özyinelemeli gezinmelerin `NULL` ile kaç kez çağrıldığını da verir.

10.4 İkili ağacın bellekte gösterimi

Bir ikili ağaç bellekte iki biçimde tutulabilir. Birincisinde düğümler bir diziye, numaralarına göre yerleştirilir; ebeveyn ve çocuklar indis hesabıyla bulunur. İkincisinde her düğüm ayrı bir `struct`'tır ve çocuklarına işaretçilerle bağlanır. Bu kesim iki gösterimi C koduyla kurar ve hangisinin ne zaman seçileceğini tartışır.

10.4.1 Dizi ile gösterim

Tam ağacın numaralandırması doğrudan bir dizi düzeni verir: i numaralı düğüm dizinin i indisli gözüne konur. Formüllerin yalın kalması için indisler 1'den başlar; C dizisinin 0. gözü kullanılmaz. Ebeveyn ve çocuk indisleri şöyle hesaplanır:

- *ebeveyn(i)* = $\lfloor i/2 \rfloor$ (kodda `baba(i)`). $i = 1$ ise düğüm köktür ve ebeveyni yoktur.
- *solcocuk(i)* = $2i$. $2i > n$ ise sol çocuk yoktur.
- *sagcocuk(i)* = $2i + 1$. $2i + 1 > n$ ise sağ çocuk yoktur.

```c
#define MAX_DUGUM 63   /* 2^k - 1 */
char agac[MAX_DUGUM + 1];  /* [0] kullanılmaz */
int  n;                    /* düğüm sayısı */

int baba(int i)     { return i / 2; }
int solcocuk(int i) { return 2 * i; }
int sagcocuk(int i) { return 2 * i + 1; }
/* çocuk var mı?  solcocuk(i) <= n */
```

`MAX_DUGUM`, dizinin tutabileceği en büyük düğüm sayısıdır. Derinliği k olan bir ağacı her biçimde tutabilmek için $2^k - 1$ gözlük yer açılmalıdır; burada $k = 6$ için 63 seçilmiştir. Dizinin boyu `MAX_DUGUM + 1`'dir, çünkü 0. göz boş bırakılır. `n` o anki düğüm sayısını tutar. Üç fonksiyon tek bir aritmetik işlemle sonuç verir: C'de iki pozitif tam sayının bölümü aşağı yuvarlandığı için `i / 2` zaten $\lfloor i/2 \rfloor$'dir. Bir çocuğun var olup olmadığı indisin `n` ile karşılaştırılmasıyla anlaşılır: `solcocuk(i) <= n` ise sol çocuk vardır.

Formüllerin doğruluğu numaralandırmadan görülür. i. düğümün düzeyi 2^j'den başlasın. Sonraki düzey 2^{j+1}'den başlar ve i'nin solundaki her düğüm oraya iki çocuk verir. Sol çocuğun numarası $2^{j+1} + 2(i - 2^j) = 2i$, sağ çocuğunki $2i + 1$ olur. $2i$ de $2i + 1$ de ikiye bölünüp aşağı yuvarlanınca i verir.

i	`agac[i]`	`baba(i)`	`solcocuk(i)`	`sagcocuk(i)`
1	A	— (kök)	2 (B)	3 (C)
2	B	1 (A)	4 (D)	5 (E)
3	C	1 (A)	6 (F)	7 (G)
4	D	2 (B)	8 (H)	9 (I)
5	E	2 (B)	10 (J)	— (11 > 10)
6	F	3 (C)	— (12 > 10)	—
7–10	G, H, I, J	3, 4, 4, 5	—	—

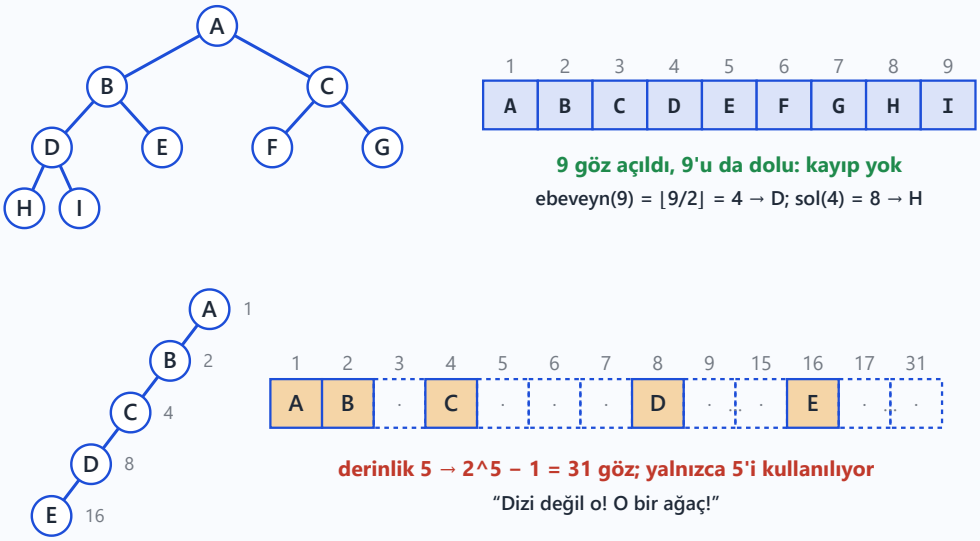
Tablo 10.2: On düğümlü tam ağaçta (A–J harfleri 1–10 numaralı gözlerde) ebeveyn ve çocuk indisleri. "—" ilgili düğümün olmadığını gösterir.

Tablo 10.2'deki değerler kodun küçük bir test programıyla üretildi. 5 numaralı E'nin sol çocuğu 10 numaralı J'dir; sağ çocuğu olacak 11 numaralı göz `n` = 10'u aştığı için yoktur. 6 numaralı ve sonraki düğümler yapraktır. Bir düğümün yaprak olup olmadığı da yalnız indisle anlaşılır: $2i > n$ ise i yapraktır. On düğümlü ağaçta yapraklar 6'dan 10'a kadar olan düğümlerdir; genel olarak tam ağaçta $\lfloor n/2 \rfloor + 1$'den n'ye kadar olan düğümler yapraktır.

> **Sık yapılan hata**
>
> Kökü 0. göze koyup 1 tabanlı formülleri kullanmak. Kök 0'daysa `solcocuk(0)` = 0 yine kökü verir; ağaç kendi üzerine katlanır. 0 tabanlı düzende formüller $2i + 1$, $2i + 2$ ve $\lfloor (i - 1)/2 \rfloor$ olur. İki düzenden biri seçilmeli ve formüller ona göre yazılmalıdır; bu derste 0. göz boş bırakılır.

10.4.2 Dizide bellek kaybı

Dizi gösterimi her ikili ağaç için kullanılabilir: ağaç, aynı derinlikteki mükemmel ağacın içine yerleştirilmiş gibi düşünülür ve var olmayan düğümlerin gözleri boş bırakılır. Ancak bu boş gözler belleği boşa harcar. Şekil 10.8 iki uç durumu karşılaştırır.



Şekil 10.8: Üstte dokuz düğümlü tam ağaç: dokuz göz açılır, dokuzu da dolar. Örneğin *ebeveyn(9) = ⌊9/2⌋ = 4* (D) ve *sol(4) = 8* (H). Altta derinliği 5 olan sola çarpık ağaç: düğümler 1, 2, 4, 8 ve 16 numaralı gözlere düşer; 31 gözün yalnız beşi kullanılır.

Tam ağaçta bellek kaybı ya hiç yoktur ya da son düzeydeki birkaç gözle sınırlıdır. Ağaç tam olmaktan uzaklaştıkça boş göz sayısı artar. En kötü durum çarpık ağaçtır. Sola çarpık ağaçta her düğüm bir öncekinin sol çocuğudur; numaralar 1, 2, 4, 8, ... diye iki katına çıkar. k düğümlü sola çarpık ağaç $2^k - 1$ göz ister ve bunların yalnız k'sını kullanır. $k = 5$ için 31 gözden 26'sı boştur. $k = 30$ olsaydı bir milyardan fazla göz gerekirdi; otuz düğümlü bir liste için bu kabul edilemez. Dizi gösteriminde ağaç değişirken de sorun çıkar: bir düğümün ortaya eklenmesi ya da silinmesi, altındaki bütün alt ağacın başka indislere taşınmasını gerektirir.

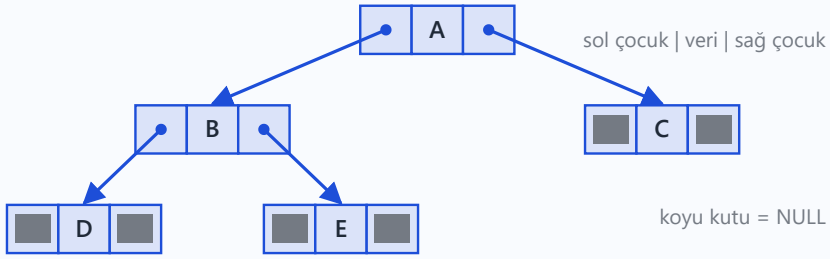
Dersin hocası bu durumu şöyle özetler: depolama ve erişim bilgisayar biliminin iki ana konusudur; ağaç tam ikili ağaçsa ve ağaca ekleme ya da çıkarma yapılmayacaksa diziyle gerçeklemek uygundur. Başka her durumda ağaç bağlı gösterimle tutulur.

10.4.3 Bağlı gösterim

Bağlı gösterimde her düğüm, verisiyle birlikte iki işaretçi taşıyan bir `struct`'tır. Sol işaretçi sol alt ağacın kökünü, sağ işaretçi sağ alt ağacın kökünü gösterir. Alt ağaç boşsa işaretçi `NULL`'dır. Bu derste kullanılan tanım şudur:

```
typedef struct agac *agac_gost;
struct agac {
    agac_gost solcocuk;   /* sol alt ağaç */
    char      veri;
    agac_gost sagcocuk;   /* sağ alt ağaç */
};
```

İlk satır, `struct agac`'ı gösteren işaretçi tipine `agac_gost` adını verir. Bu satır `struct agac` tanımlanmadan önce yazılabilir, çünkü C bir yapının içeriğini bilmeden ona işaretçi tanımlamaya izin verir. Yapının içinde `agac_gost` tipinde iki alan vardır; yapı kendisiyle aynı türden nesneleri gösterir. Bu tür yapılara kendine başvuran yapı denir. Bağlı listedeki düğüm yapısının tek bağ alanı vardı; burada iki bağ alanı vardır. Ağacın kendisi kök düğümün adresiyle temsil edilir: `agac_gost kok;` bildirimi bir ağaç değişkenidir. `kok == NULL` boş ağaç demektir.



Şekil 10.9: A(B(D, E), C) ağacının bağlı gösterimi. Her düğüm üç gözlüdür: sol çocuk, veri, sağ çocuk. Koyu küçük kutular `NULL` işaretçilerdir. Beş düğümde on işaretçi alanı vardır; dördü dolu, altısı ($n + 1$) `NULL` 'dır.

Düğümler `malloc` ile heap bellekten ayrı ayrı ayrılır ve bellekte herhangi bir yere düşer. Ağaç yapısı yalnız işaretçilerle kurulur. Tablo 10.3 Şekil 10.9'daki ağacın bellekteki olası bir görünümünü verir. Adresler örnek olarak seçilmiştir; gerçek adresler her çalıştırmada değişir. Burada önemli olan, düğümlerin sırasının ağacın şekliyle hiçbir ilgisinin olmamasıdır.

Adres	`solcocuk`	`veri`	`sagcocuk`
0x1040	NULL	'E'	NULL
0x10a0	0x2020	'B'	0x1040
0x1f00 (`kok`)	0x10a0	'A'	0x2060
0x2020	NULL	'D'	NULL
0x2060	NULL	'C'	NULL

Tablo 10.3: A(B(D, E), C) ağacının bellekte olası bir yerleşimi (adresler örnektir).

Bir düğümün bellekte kapladığı yer derleyiciye ve makineye bağlıdır. 64 bitlik bir makinede her işaretçi 8 bayttır ve `veri` 1 bayttır. Derleyici ikinci işaretçiyi 8'in katı olan bir adrese hizalamak için `veri`'den sonra 7 bayt dolgu ekler. Testte kullanılan Windows x86-64 derlemesinde `sizeof(struct agac)` 24 bayt çıktı: 16 baytlık iki işaretçi, 1 baytlık veri ve 7 baytlık dolgu. Alanların sırası `solcocuk`, `sagcocuk`, `veri` olsaydı dolgu sona kayardı ve boyut yine 24 olurdu. n düğümlü ağaç n düğüm kadar yer kaplar; ağacın şekli belleği etkilemez. Dizi gösteriminin aksine çarpık ağaç da dengeli ağaç kadar yer tutar.

10.4.4 Düğüm ayırma ve ağacı bırakma

Her yeni düğüm için aynı üç iş yapılır: bellek ayrılır, ayırmanın başarılı olduğu denetlenir ve çocuk işaretçileri `NULL` yapılır. Bu işleri tek bir fonksiyonda toplamak hem kodu kısaltır hem de unutulan bir denetimin yol açacağı hataları önler. Sunumdaki `ogeal` ve onun tersi olan `agac_sil` şunlardır:

```
agac_gost ogeal(void) {          /* yeni düğüm */
    agac_gost p = malloc(sizeof *p);
    if (p == NULL) {
        fprintf(stderr, "Bellek yok!\n");
        exit(EXIT_FAILURE);
    }
    p->solcocuk = p->sagcocuk = NULL;
    return p;
}

void agac_sil(agac_gost L) {     /* kök sonda */
    if (!L) return;
    agac_sil(L->solcocuk);
    agac_sil(L->sagcocuk);
    free(L);
}
```

`ogeal`'ın ilk satırı `malloc(sizeof *p)` ile bir düğümlük yer ister. `sizeof *p` yazımı, `p`'nin gösterdiği tipin boyutunu verir; tip adı değişse bile ifade doğru kalır. `malloc` yer bulamazsa `NULL` döner. Bu durumda fonksiyon `stderr`'e bir ileti yazar ve programı `exit(EXIT_FAILURE)` ile sonlandırır. Ödev ve sınav programları için bu yeterlidir; büyük bir programda `NULL` çağırana döndürülür ve karar orada verilir. `malloc`'un verdiği bellek temizlenmemiştir, alanlar rastgele değerler taşır. Bu yüzden iki çocuk işaretçisi açıkça `NULL` yapılır. `p->solcocuk = p->sagcocuk = NULL;` ifadesi sağdan sola çalışır: önce `sagcocuk`'a `NULL` atanır, atamanın değeri de `solcocuk`'a atanır. Veri alanını çağıran doldurur.

`agac_sil` ağacın bütün düğümlerini heap'e geri verir. Sıra önemlidir: önce sol alt ağaç, sonra sağ alt ağaç silinir, kök en sonda iade edilir. Kök önce `free` edilseydi `L->solcocuk` ve `L->sagcocuk` alanlarını okumak, iade edilmiş belleğe erişmek olurdu. Bu tanımsız davranıştır. Okuma bazen "çalışır", bazen çöp değer verir; iki durumda da alt ağaçlar ya sızar ya da program çöker. Kökü sona bırakan bu sıra, kök sonda gezinmenin ta kendisidir. Fonksiyon her düğüme bir kez uğradığı için $O(n)$ zamanda çalışır. Boş ağaçla çağrılınca hiçbir şey yapmaz; `agac_sil(NULL)` güvenlidir.

Bu nottaki bütün C kodları sayaçlı bir `malloc` / `free` sarmalayıcısıyla da çalıştırıldı. Her testin sonunda ayrılan ve iade edilen düğüm sayıları eşit çıktı; hiçbir testte sızıntı kalmadı. Örneğin gezinme testlerinde 53 düğüm ayrıldı ve `agac_sil` çağrılarıyla 53 düğüm geri verildi.

> **Sık yapılan hata**
>
> Ağacı yalnız kök düğümü serbest bırakarak silmek: `free(kok);`. Bu satır tek bir düğümü iade eder; geri kalan $n - 1$ düğümün adresleri kökün içindeydi ve artık okunamaz. Bu düğümler program bitene kadar bellekte kalır. Uzun süre çalışan bir programda her ağaç silmede bu kadar bellek kaybolur. Doğru yol, `agac_sil` gibi bütün düğümlere kök sonda sırayla uğrayan bir yordamdır.

10.4.5 Dizi mi, bağlı gösterim mi?

İki gösterimin üstün ve zayıf yanları birbirini tamamlar. Tablo 10.4 karşılaştırmayı özetler.

Ölçüt	Dizi	Bağlı gösterim
Düğüm başına ek alan	Yok (işaretçi alanı yok)	İki işaretçi
Toplam bellek	$2^k - 1$ göz; tam ağaçta kayıp yok, çarpık ağaçta çok büyük	Her şekilde n düğüm
Ebeveyne erişim	$O(1)$: $\lfloor i/2 \rfloor$	Doğrudan yok (gerekirse ebeveyn alanı eklenir)
Çocuğa erişim	$O(1)$: $2i$, $2i + 1$	$O(1)$: işaretçi
Ekleme ve silme	Alt ağaçların taşınması gerekebilir	Yalnız işaretçi ataması
Uygun olduğu yer	Tam ve şekli değişmeyen ağaç (ör. heap)	Şekli değişen her ağaç

Tablo 10.4: Dizi ve bağlı gösterimin karşılaştırması.

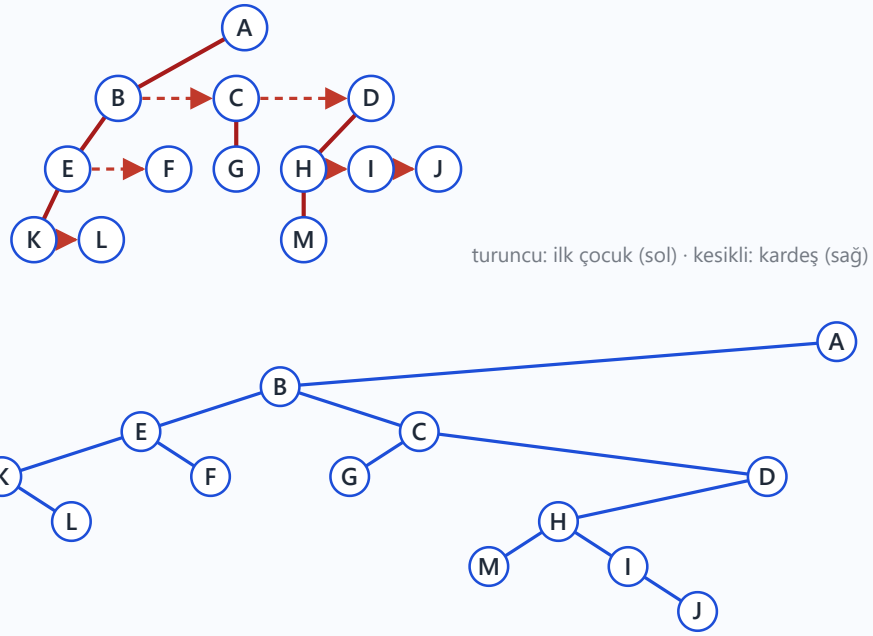
Kural kısaca şudur: ağaç tam ve sabitse dizi, şekli değişen ağaç için bağlı gösterim seçilir. Örneğin bir ağdaki bilgisayarlar sabitse ve sonradan eklenip çıkarılmayacaksa bağlantı ağacı diziyle tutulabilir. Heap, her zaman tam kalacak biçimde ekleme ve silme yaptığı için dizinin hızından tam olarak yararlanır. Arama ağaçları ise eklemeyle her yöne büyüyebilir; onlar bağlı gösterimle tutulur. Bağlı gösterimin tek zayıflığı ebeveyne erişimdir. Bir düğümden yukarı çıkmak gerekiyorsa düğüme üçüncü bir işaretçi, `ebeveyn` alanı eklenir; bunun bedeli her düğümde 8 bayt daha ve her eklemede bir atama daha demektir.

10.4.6 Genel ağacı ikili ağaca çevirme

Genel ağaçta bir düğümün çocuk sayısı değişkendir. A'nın üç çocuğu varken F'nin hiç çocuğu yoktur. Sabit alanlı bir `struct` ile her düğüme "en çok kaç çocuk olabilir" kadar işaretçi ayırmak çoğu işaretçiyi boş bırakır. Derecesi d olan n düğümlü bir ağaçta nd alanın yalnız $n - 1$'i dolu olur. Çözüm, **sol çocuk – sağ kardeş** gösterimidir: her düğüm yalnız iki bağ taşır. Birincisi düğümün ilk (en soldaki) çocuğunu, ikincisi düğümün bir sağındaki kardeşini gösterir.

```
struct gdugum {
    char veri;
    struct gdugum *ilk_cocuk;  /* = sol */
    struct gdugum *kardes;     /* = sağ */
};
```

Bu yapı biçim olarak ikili ağaç düğümüyle aynıdır: `ilk_cocuk` sol işaretçinin, `kardes` sağ işaretçinin yerine geçer. Dolayısıyla her genel ağaç bir ikili ağaç olarak saklanabilir ve ikili ağaç için yazılmış bütün yordamlar ona uygulanabilir. Şekil 10.10 dönüşümü iki adımda gösterir.



Şekil 10.10: Üstte A–M genel ağacında yalnız ilk çocuk bağları (turuncu) korunmuş, kardeşler kesikli sağ kardeş bağlarıyla zincirlenmiştir. Altta aynı bağlar sol ve sağ işaretçi olarak yeniden çizilmiştir: sonuç bir ikili ağaçtır ve kökün sağ bağı `NULL` 'dır.

Dönüşüm şöyle yapılır. Her düğüm için ilk çocuğa giden bağ tutulur, öteki çocuklara giden bağlar silinir. Sonra aynı ebeveynin çocukları soldan sağa bir zincir hâlinde birbirine bağlanır. B'nin ilk çocuğu E'dir; E'nin kardeşi F'dir. A'nın ilk çocuğu B'dir; B'nin kardeşi C, C'nin kardeşi D'dir. İkili ağaçta B, A'nın sol çocuğu olur; C, B'nin sağ çocuğu olur; D, C'nin sağ çocuğu olur. Kökün kardeşi olmadığı için kökün sağ bağı her zaman `NULL` 'dır.

Dönüşümün gezinmelerle ilgili iki sonucu vardır. İkili ağacın kök önde sırası, genel ağacın kök önde sırasıyla aynıdır: ikisinde de A B E K L F C G D H M I J çıkar. İkili ağacın kök ortada sırası ise genel ağacın kök sonda sırasına eşittir: K L E F B G C M H I J D A. İkinci sonucun nedeni şudur: ikili ağaçta bir düğümün sol alt ağacı, genel ağaçtaki bütün soyudur. Kök ortada gezinme önce sol alt ağacı, yani bütün soyu yazar; düğümün kendisini ondan sonra yazar. Sağ alt ağaç ise kardeşlerdir ve onlar düğümden sonra gelir. Bu, genel ağaçta "önce çocukları, sonra kendini" kuralıdır. İki eşitlik bir Python betiğiyle de denetlendi.

10.5 Özyinelemeli gezinmeler

Bir veri yapısının bütün öğelerine belirli bir sırayla birer kez uğramaya **gezinme** (*traversal*) denir. Dizide ve bağlı listede gezinme sırası doğaldır: baştan sona. İkili ağaçta ise her düğümden iki yöne gidilebilir ve "önce hangisi?" sorusunun birden çok cevabı vardır. Bu kesim olası sıraları sayar, bunların üçünü C ile yazar, kodu adım adım izler ve maliyetini çıkarır.

10.5.1 Altı gezinme sırası

Bir düğümde ele alınacak üç şey vardır: düğümün verisi (V), sol alt ağaç (L) ve sağ alt ağaç (R). Gezinme, bu üçünü hangi sırayla işleyeceğine karar verir ve aynı kararı her alt ağaçta yineler. Üç öğenin *3! = 6* farklı sırası vardır. Tablo 10.5 hepsini listeler.

Sıra	Ad
Sol – Veri – Sağ (L-V-R)	**kök ortada** (*inorder*)
Sol – Sağ – Veri (L-R-V)	**kök sonda** (*postorder*)
Veri – Sol – Sağ (V-L-R)	**kök önde** (*preorder*)
Sağ – Veri – Sol (R-V-L)	ters kök ortada

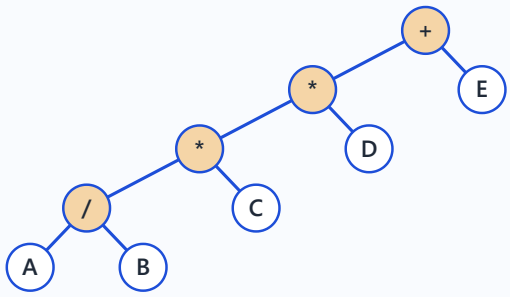
Sıra	Ad
Sağ – Sol – Veri (R-L-V)	ters kök sonda
Veri – Sağ – Sol (V-R-L)	ters kök önde

Tablo 10.5: Bir düğümde verinin ve iki alt ağacın altı olası işlenme sırası.

Solun sağdan önce gezildiği üç sıra standarttır ve adlarını kökün konumundan alır. Kök, alt ağaçlardan önce işleniyorsa kök önde, iki alt ağacın arasında işleniyorsa kök ortada, ikisinden sonra işleniyorsa kök sonda gezinmedir. C kodundaki `kok_onde_dolas` adı bu terimden gelir. Kalan üç sıra, standart sıraların sağdan sola gezilen aynalarıdır. Uygulama gerektirirse onlar da kullanılır.

10.5.2 İfade ağacı ve üç gösterim

Gezinme sıralarının adları aritmetik ifadelerden tanıdıktır. **İfade ağacı** (*expression tree*), iç düğümleri işleç, yaprakları işlenen olan ikili ağaçtır. Her işleç düğümü, iki alt ağacının değerlerine uygulanır. Şekil 10.11 *((((A / B) * C) * D) + E)* ifadesinin ağacını gösterir. En son uygulanan işleç `+` olduğu için kök odur; sol alt ağacı *((A / B) * C) * D*, sağ alt ağacı E'dir.



Şekil 10.11: ((((A/B)*C)*D)+E) ifadesinin ağacı. İç düğümler (turuncu) işleç, yapraklar işlenendir. Parantezler ağacın şeklinde saklıdır: A/B en derinde olduğu için önce hesaplanır.

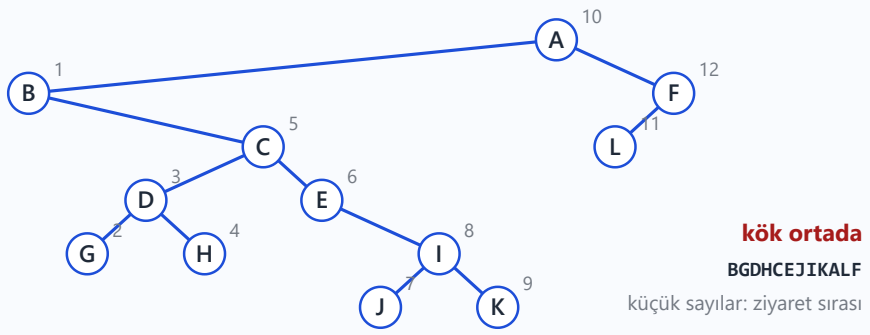
Bu ağaç üç standart sırayla gezilince ifadenin üç bilinen gösterimi çıkar:

- Kök ortada: `A / B * C * D + E`. Bu, **ara-işleç** (*infix*) gösterimidir. Ağaçta parantez yoktur; ara-işleç yazarken her alt ağacın etrafına parantez eklenerek *((((A/B)*C)*D)+E)* geri elde edilir.
- Kök önde: `+ * * / A B C D E`. Bu, **ön-işleç** (*prefix*) gösterimidir.
- Kök sonda: `A B / C * D * E +`. Bu, **son-işleç** (*postfix*) gösterimidir.

Ön-işleç ve son-işleç gösterimleri parantez gerektirmez: işleçlerin sırası hesaplama sırasını tek anlamlı belirler. Derleyiciler kaynak koddaki ifadeleri önce böyle bir ağaca çevirir. Son-işleç gösterimi bir yığınla soldan sağa tek geçişte değerlendirilir: işlenen gelince yığına itilir; işleç gelince yığından iki değer alınır, işlem yapılır ve sonuç yığına itilir. Ağacın kök sonda gezinmesi tam olarak bu sırayı üretir.

10.5.3 Aynı ağaçta üç gezinme

Gezinme çıktısını elle bulmak için kural her alt ağaçta yinelenir. On iki düğümlü A…L ağacı üzerinde üç gezinme şöyle bulunur. Ağaçta A'nın sol çocuğu B, sağ çocuğu F'dir. B'nin sol çocuğu yoktur, sağ çocuğu C'dir. C'nin çocukları D ve E'dir; D'nin çocukları G ve H'dir. E'nin yalnız sağ çocuğu I vardır; I'nın çocukları J ve K'dir. F'nin yalnız sol çocuğu L vardır. Şekil 10.12 ağacı kök ortada sırasıyla numaralanmış olarak gösterir.



Şekil 10.12: A...L ağacında kök ortada gezinme. B'nin solu boş olduğu için ilk yazılan düğüm B'dir; çıktı BGDHCEJIKALF.

- **Kök önde (V-L-R).** Önce kök A yazılır. Sonra sol alt ağacın tamamı: B, ardından B'nin sağı olan C alt ağacı (C, D, G, H, E, I, J, K). En son sağ alt ağaç: F, L. Çıktı: `ABCDGHEIJKFL`.
- **Kök ortada (L-V-R).** A'nın sol alt ağacı B'den başlar. B'nin solu boş olduğu için ilk B yazılır. Sonra B'nin sağ alt ağacı kök ortada gezilir: G, D, H, C, E, J, I, K. Ardından A yazılır, en son F'nin alt ağacı: L, F. Çıktı: `BGDHCEJIKALF`.
- **Kök sonda (L-R-V).** Kök her alt ağaçta en sondadır. G, H, D; sonra J, K, I, E; sonra C, B; sonra L, F; en son A. Çıktı: `GHDJKIECBLFA`.

Üç çıktıda yaprakların (G, H, J, K, L) birbirine göre sırası aynıdır. Gezinmeler yalnız iç düğümlerin yaprakların arasında nereye yerleştiği bakımından ayrılır. Bu gözlem, el ile bulunan bir çıktının denetlenmesinde işe yarar. Kök önde çıktının ilk harfi ve kök sonda çıktının son harfi her zaman köktür.

10.5.4 Gezinme yordamları

Gezinmenin özyinelemeli tanımı neredeyse kelimesi kelimesine C koduna dönüşür. Kök önde gezinme için "ağaç boş değilse kökü yaz, sol alt ağacı kök önde gez, sağ alt ağacı kök önde gez" denir:

```
void kok_onde_dolas(agac_gost L) {
  if (L) {
    printf("%c", L->veri);
    kok_onde_dolas(L->solcocuk);
    kok_onde_dolas(L->sagcocuk);
  }
}
```

Kök ortada ve kök sonda yordamları aynı iskelettir; yalnız `printf` satırının yeri değişir:

```
void kok_ortada_dolas(agac_gost L) {
  if (L) {
    kok_ortada_dolas(L->solcocuk);
    printf("%c", L->veri);
    kok_ortada_dolas(L->sagcocuk);
  }
}
```

```
void kok_sonda_dolas(agac_gost L) {
  if (L) {
    kok_sonda_dolas(L->solcocuk);
    kok_sonda_dolas(L->sagcocuk);
    printf("%c", L->veri);
  }
}
```

Üç yordam satır satır şöyle okunur. `if (L)` sınaması, `L` boş olmayan bir ağacı gösteriyorsa doğrudur. Bu tek sınama iki işi birden görür. Birincisi, boş ağaçla çağrılan yordam hiçbir şey yapmadan döner; `kok_ortada_dolas(NULL)` güvenlidir. İkincisi, özyinelemenin durma koşuludur: her çağrı bir alt ağaca iner, yapraktan sonraki çağrı `NULL` ile yapılır ve orada zincir durur. `printf("%c", L->veri)` düğümün ve-

risini yazar; `L->veri`, `(*L).veri` 'nin kısaltmasıdır. İki özyinelemeli çağrı sol ve sağ alt ağaçları aynı kuralla gezer. `printf` çağrılardan önceyse kök önde, arasındaysa kök ortada, sonraysa kök sonda gezinme elde edilir.

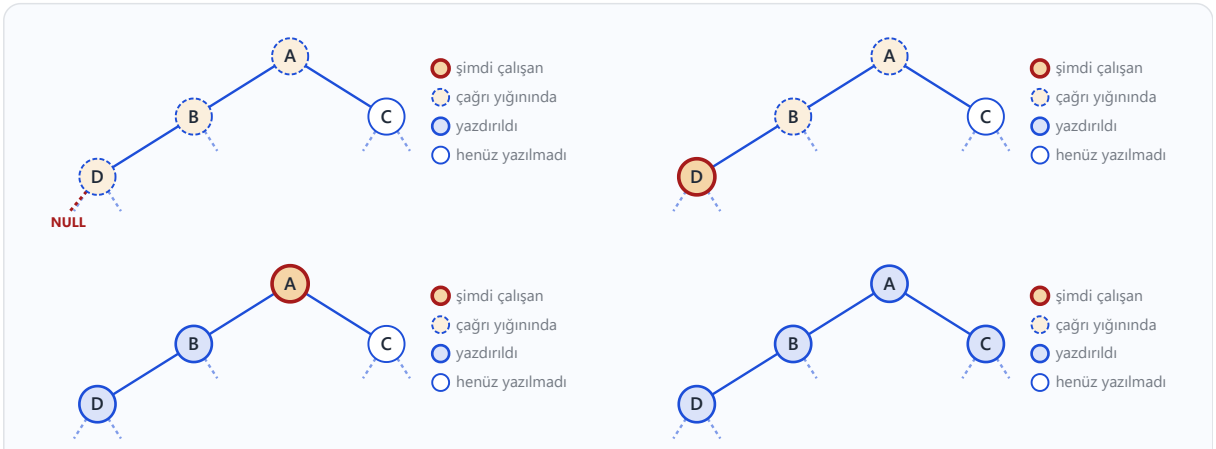
Dersin hocasının bu kodlar üzerine sık yaptığı bir uyarı vardır: ağaç işaretçisi `NULL` gelebilir ve öğrencilerin çoğu buna dikkat etmez. Gezinme yordamlarında `if (L)` sınaması bu yüzden zorunludur. Sınama unutulursa yordam ilk yaprağın boş çocuğuna inince `NULL->veri` 'ye erişir ve program çöker. Boş ağaçla çağrılan bir yordam ise daha ilk satırda çöker.

10.5.5 Kod izleme: kök ortada gezinme

Özyinelemeli bir yordamın nasıl çalıştığı, çağrı yığınındaki çerçeveler izlenerek anlaşılır. Her çağrı yığına bir çerçeve ekler; çerçeve o çağrının `L` parametresini ve dönüş adresini tutar. Çağrı bitince çerçeve yığından kalkar ve çalışma, çağıranın kaldığı yerden sürer. Aşağıdaki izleme dört düğümlü A(B(D), C) ağacı üzerindedir: A'nın sol çocuğu B, sağ çocuğu C; B'nin yalnız sol çocuğu D vardır.

Adım	Olay	Çağrı yığını	Çıktı
1	`L` = A: sol için `kok_ortada_dolas(B)` çağrılır	A	
2	`L` = B: sol için `kok_ortada_dolas(D)` çağrılır	A B	
3	`L` = D: sol için `kok_ortada_dolas(NULL)` çağrılır	A B D	
4	`L == NULL` : `if (L)` yanlış, çağrı döner	A B D NULL	
5	`L` = D: solu bitti, D yazılır	A B D	D
6–7	D'nin sağı için `NULL` çağrısı yapılır ve hemen döner; D'nin çerçevesi kalkar	A B D NULL	D
8	`L` = B: solu bitti, B yazılır	A B	DB
9–10	B'nin sağı için `NULL` çağrısı; B'nin çerçevesi kalkar	A B NULL	DB
11	`L` = A: solu bitti, A yazılır	A	DBA
12	`L` = A: sağ için `kok_ortada_dolas(C)` çağrılır	A	DBA
13–14	`L` = C: sol için `NULL` çağrısı yapılır ve döner	A C NULL	DBA
15	`L` = C: C yazılır	A C	DBAC
16–17	C'nin sağı için `NULL` çağrısı; C ve A'nın çerçeveleri sırayla kalkar	A C NULL	DBAC
18	Bütün çağrılar döndü	(boş)	DBAC

Tablo 10.6: `kok_ortada_dolas(A)` izlemesi, A(B(D), C) ağacı. Çağrı yığını alttan üste yazılmıştır.



Şekil 10.13: Kök ortada gezinmenin dört anı: D'nin sol çocuğu için yapılan `NULL` çağrısı (adım 4), D'nin yazılması (adım 5), A'nın yazılması (adım 11) ve gezinmenin sonu (adım 18). Kesikli düğümler çağrı yığınında bekleyen çerçevelerdir.

İzlemeden üç sonuç çıkar. Birincisi, ilk yazılan düğüm D'dir: kök ortada gezinme önce en soldaki düğüme kadar iner. İkincisi, çağrı yığınında aynı anda en çok dört çerçeve bulunur: A, B, D ve D'nin boş çocuğu için açılan çerçeve. Bu sayı ağacın derinliğinin (3) bir fazlasıdır. Üçüncüsü, her düğüme bir kez girilir ve bir kez çıkılır; dört düğümlü ağaçta düğüm çağrılarına ek olarak beş `NULL` çağrısı yapılır. Aynı

ağaçta kök önde gezinme `ABDC`, kök sonda gezinme `DBCA` verir. İzlemenin adımları ve çağrı sayıları aynıdır; yalnız yazdırmanın yapıldığı an değişir. Kök önde gezinmede düğüm çerçeve açılır açılmaz, kök sonda gezinmede çerçeve kapanmadan hemen önce yazılır.

10.5.6 Maliyet ve NULL çağrıları

Zaman. Gezinme yordamı her düğüm için bir kez çağrılır ve o çağrıda sabit iş yapar: bir sınama, bir `printf`, iki çağrı. Bunlara ek olarak boş alt ağaçlar için `NULL` çağrıları vardır. Bu çağrıların sayısı tam olarak bilinir.

> **Teorem 10.3 (Gezinmedeki çağrı sayısı)**
>
> n düğümlü bir ikili ağaçta özyinelemeli gezinme yordamı toplam $2n + 1$ kez çağrılır. Bu çağrıların n'i düğümlerle, $n + 1$'i `NULL` ile yapılır.

Kanıt. Yordam her düğüm için bir kez çağrılır: ilk çağrı kökle, diğerleri ebeveynin çocuk işaretçileriyle yapılır. Her düğüm çağrısı iki çağrı daha yapar; bu yüzden ilk çağrı dışındaki çağrı sayısı $2n$'dir ve ilkiyle birlikte $2n + 1$ eder. Çocuk işaretçilerinin $n - 1$'i bir düğümü gösterir, $n + 1$'i `NULL`'dır ($n_0 = n_2 + 1$'in sonucu). Dolayısıyla $n + 1$ çağrı `NULL` ile yapılır. ■

Toplam iş $2n + 1$ çağrıyla orantılıdır; zaman $\Theta(n)$'dir. Ağacın şekli zamanı değiştirmez: dengeli ağaç da çarpık ağaç da aynı sayıda çağrı yapar. A…L ağacıyla yapılan testte 12 düğüm için 25 çağrı sayıldı; bunların 13'ü `NULL` çağrısıydı.

Bellek. Çağrı yığınında aynı anda bulunan çerçeve sayısı, kökten o anki düğüme giden yolun uzunluğuna eşittir (artı en fazla bir `NULL` çerçevesi). Bu sayı en çok ağacın derinliği h kadardır. Bellek maliyeti bu yüzden $O(h)$'dir. Dengeli ağaçta $h = O(\lg n)$, çarpık ağaçta $h = n$'dir. Bir milyon düğümlü dengeli ağaç yirmi civarında çerçeveyle gezilir; aynı sayıda düğümlü çarpık ağaç bir milyon çerçeve ister. Testte, Windows'ta zig cc ile derlenen programda sola çarpık 100.000 düğümlü zincir özyinelemeyle gezilebildi; 300.000 düğümlü zincirde çağrı yığını taştı ve program çöktü. Aynı zincir yığınla yazılan yinelemeli yordamla sorunsuz gezildi. Bu sınırlar makineye, derleyiciye ve yığın boyutuna göre değişir.

`NULL` çağrıları, çağrıdan önce denetim yapılarak kaldırılabilir:

```
/* önce: n+1 NULL çağrısı */
kok_onde_dolas(L->solcocuk);
kok_onde_dolas(L->sagcocuk);
/* sonra: çağırmadan denetle */
if (L->solcocuk) kok_onde_dolas(L->solcocuk);
if (L->sagcocuk) kok_onde_dolas(L->sagcocuk);
```

İkinci biçimde yordam yalnız var olan çocuklar için çağrılır ve toplam çağrı sayısı $2n + 1$'den n'ye iner. A…L ağacında 25 çağrı 12'ye düştü. Ancak bu biçimde de yordamın en başındaki `if (L)` kalmalıdır, çünkü ağacın kendisi boş gelebilir. Kazanç bir sabit çarpandır; asimptotik zaman yine $\Theta(n)$'dir. Kod biraz uzadığı için çoğu durumda ilk, kısa biçim tercih edilir.

Dersin hocası bu noktada iki şeyin altını çizer. Birincisi, “özyinelemeli algoritmalar çok bellek yer” iddiası çoğu zaman özyinelemeli kod yazmamak için bir kaçamaktır: dengeli bir ağaçta yığın derinliği $\lg n$ ile sınırlıdır ve bu çok küçük bir sayıdır. İkincisi, algoritmayı yazmak işin yarısıdır; bilgisayar biliminde her algoritmanın analizi yapılmak zorundadır. Bellek için $O(\lg n)$ sınırının yalnız dengeli ağaçta geçerli olduğu, genel sınırın $O(h)$ olduğu da bu analizin bir parçasıdır.

> **Sık yapılan hata**
>
> Özyinelemeli gezinmenin belleğini her ağaç için $O(\lg n)$ saymak. Bu sınır yalnız derinliği $\lg n$ mertebesinde kalan dengeli ağaçlarda doğrudur. Sıralı anahtarlarla kurulan bir arama ağacı çarpık olur ve gezinmesi n çerçeve ister. Doğru ifade “bellek $O(h)$, $\lg n \le h \le n$” biçimindedir.

10.5.7 Çağrı ağacı

Özyinelemeli kod okuyarak değil, yazarak öğrenilir. Dersin hocasının benzetmesiyle, bir halterciyi yüz kez izlemek o halteri kaldırmayı öğretmez. Bir yordam anlaşılmadıysa çağrıların oluşturduğu ağaç kâğıda çizilir.

Çağrı ağacında her çağrı bir düğümdür; bir çağrının yaptığı alt çağrılar onun çocuklarıdır. Gezinme yordamının çağrı ağacı, gezilen ağacın kendisidir: her düğüm için bir çağrı vardır ve o çağrının iki çocuğu, sol ve sağ alt ağaç için yapılan çağrılardır. Tek fark, `NULL` çağrılarının yaprak olarak eklenmesidir. Böylece çağrı ağacında n iç düğüm ve $n + 1$ yaprak bulunur. Bu, her iç düğümün tam iki çocuğu olduğu bir ikili

ağaçtır ve $n_0 = n_2 + 1$ eşitliğine bir kez daha uyar. Çağrı yığınının herhangi bir andaki içeriği, çağrı ağacında kökten o anki çağrıya giden yoldur.

10.5.8 İfade ağacını hesaplamak

İfade ağacının değerini bulmak bir kök sonda gezinmedir. Bir işlecin değeri, iki alt ifadenin değerleri bilinmeden hesaplanamaz. Önce sol alt ağaç, sonra sağ alt ağaç hesaplanır, en son kökteki işleç uygulanır. Ağacı tutmak için düğüm yapısı biraz değişir: yapraklarda bir sayı, iç düğümlerde bir işleç saklanır.

```
typedef struct ifade *ifade_gost;
struct ifade {
    char      islec;   /* '+', '-', '*', '/' */
    int       deger;   /* yalnız yapraklarda */
    ifade_gost sol, sag;
};
```

`islec` alanı `'+'`, `'-'`, `'*'` ya da `'/'` karakterini tutar. `deger` alanı yalnız yapraklarda anlamlıdır. `sol` ve `sag` alt ifadeleri gösterir. Her iç düğümün tam iki çocuğu vardır, çünkü dört işleç de ikilidir; yaprakların hiç çocuğu yoktur. Tekli eksi ($-x$) gibi tek işlenenli bir işleç eklenseydi tek çocuklu düğümler de gerekirdi. Hesaplama yordamı şudur:

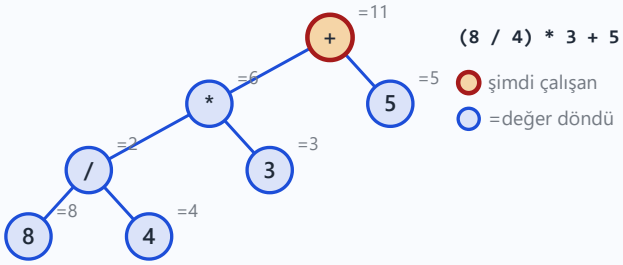
```
int hesapla(ifade_gost t) {
  if (!t->sol && !t->sag)        /* yaprak: sayı */
    return t->deger;
  int a = hesapla(t->sol);
  int b = hesapla(t->sag);
  switch (t->islec) {
    case '+': return a + b;
    case '-': return a - b;
    case '*': return a * b;
    default:  return a / b;
  }
}
```

İlk sınama düğümün yaprak olup olmadığına bakar: iki çocuk da `NULL` ise düğüm bir sayıdır ve `deger` döndürülür. Değilse sol alt ifadenin değeri `a`'ya, sağ alt ifadenin değeri `b`'ye hesaplanır. `switch`, işlece göre `a + b`, `a - b`, `a * b` ya da `a / b` döndürür. `default` dalı bölmeyi karşılar. Yordam boş ağaçla çağrılmamalıdır: `t` `NULL` ise ilk satır çöker. Bölmede sıfıra bölme denetlenmez; tam sayı bölmesi de kesirli kısmı atar. Yordam her düğüme bir kez uğradığı için zamanı $O(n)$, çağrı yığını nedeniyle belleği $O(h)$'dir.

Tablo 10.7 *(8 / 4) * 3 + 5* ifadesinin hesaplanmasını izler. Ağacın kökü `+`'dır; sol alt ağacı *(8/4) * 3*, sağ alt ağacı 5'tir.

Adım	Düğüm	Olay	Döndürülen
1	`+`	işleç: önce sol alt ifade	
2	`*`	işleç: önce sol alt ifade	
3	`/`	işleç: önce sol alt ifade	
4	8	yaprak	8
5–6	`/` → 4	`a` = 8; sağ alt ifade 4 bir yapraktır	4
7	`/`	`b` = 4; 8 / 4	2
8–9	`*` → 3	`a` = 2; sağ alt ifade 3 bir yapraktır	3
10	`*`	`b` = 3; 2 * 3	6
11–12	`+` → 5	`a` = 6; sağ alt ifade 5 bir yapraktır	5
13	`+`	`b` = 5; 6 + 5	11

Tablo 10.7: `hesapla` izlemesi, *(8 / 4) * 3 + 5*.



Şekil 10.14: `hesapla` bittiğinde her düğümün yanında döndürdüğü değer yazılıdır. Değerler aşağıdan yukarı, kök sonda sırasıyla oluşur: 8, 4, 2, 3, 6, 5, 11.

Değerlerin oluşma sırası 8, 4, 2, 3, 6, 5, 11'dir. Bu, ağacın kök sonda gezinmesidir: `8 4 / 3 * 5 +`. Sunumdaki ifade ağacı bir test programıyla kurulup `hesapla` çağrıldı ve 11 sonucu alındı. Aynı programla *(A + B) * (C − D)* ağacı A = 7, B = 3, C = 9, D = 4 değerleriyle hesaplandı: *10 · 5 = 50* bulundu.

10.6 Yığınla gezinme

Özyinelemeli gezinme kısa ve okunaklıdır, ama iki durumda yetersiz kalır. Çok derin ağaçlarda çağrı yığını taşabilir. Bazı ortamlarda da (gömülü sistemler, özyinelemeye izin vermeyen kodlama kuralları) özyineleme hiç kullanılamaz. Bu kesim aynı gezinmeleri döngü ve açık bir yığınla yazar. Ardından aynı kalıbı biraz değiştirerek sunumdaki dokuz soruyu çözer: yinelemeli kök önde ve kök sonda gezinme, uç düğümleri yazma ve çocuk sayısına göre düğüm seçme.

10.6.1 Özyinelemeden döngüye

Her özyinelemeli algoritma yinelemeli hâle getirilebilir. Bunun iki yolu vardır. Birincisi, algoritmanın davranışını doğrudan bir döngüyle ifade etmektir; örneğin faktöriyel bir `for` döngüsüyle hesaplanır. İkincisi, işletim sisteminin özyinelemeli çağrılarda yaptığını programın kendisinin yapmasıdır: dönülecek yerler bir yığına konur ve döngü bu yığından beslenir. Ağaç gezinmesinde ikinci yol izlenir.

Kök ortada gezinmeyi düşünelim. Bir düğüm yazılmadan önce onun sol alt ağacının tamamı yazılmalıdır. Kökten başlayıp sola inerken geçilen her düğüm bekletilir: önce A, sonra B, sonra D. Sola inmek bitince bekleyenlerden hangisi yazılır? En son bekletilen D. D'den sonra B, B'den sonra A gelir. Yani bekleyen düğümlere eklenme sırasının tersiyle erişilir: son eklenen önce çıkar. Bu tam olarak bir yığının (LIFO) tanımıdır. Dersin hocası bu anı “bu bir yığın yapısı!” diye vurgular; yığının buraya dışarıdan eklenmediğini, problemin doğasından kendiliğinden çıktığını gösterir.

Yığına düğümlerin kopyası değil, adresleri konur. Bir düğümün kopyası verisiyle ve iki işaretçisiyle birlikte en az 24 bayt tutar; adresi ise 8 bayttır. Daha önemlisi, kopya yığından çıktıktan sonra da asıl düğümle aynı alt ağaçları gösterir ve gereksizdir: düğüme erişmek için adresi yeterlidir. Yığında aynı anda en fazla ağacın sola doğru derinliği kadar adres bulunur.

10.6.2 Yığınla kök ortada gezinme

Yığın için iki küçük yordam yeterlidir. Yığın, `agac_gost` tipinde bir dizi ve üst indisi tutan bir tam sayıdır:

```
void yigit_ekle(int *ust, agac_gost p,
                agac_gost y[]) {
  if (*ust == MAX_SIZE - 1) {
    fprintf(stderr, "Yigit dolu\n");
    exit(EXIT_FAILURE);
  }
  y[++*ust] = p;
}
agac_gost yigit_al(int *ust,
                   agac_gost y[]) {
  return (*ust == -1) ? NULL : y[(*ust)--];
}
```

`yigit_ekle` yığın doluysa (üst indis `MAX_SIZE - 1`'e ulaşmışsa) bir ileti yazıp programı sonlandırır. Değilse `++*ust` ile üst indisi artırır ve adresi o göze yazar. `++*ust` ifadesi `++(*ust)` olarak okunur: işaretçinin kendisi değil, gösterdiği tam sayı artar. `yigit_al`, yığın boşsa `NULL` döndürür; değilse üstteki adresi döndürür ve `(*ust)--` ile üst indisi azaltır. Boş yığında `NULL` döndürmek bu uygulamada çok uygundur, çünkü yığına hiçbir zaman `NULL` konmaz; `NULL` dönüşü "yığın boş" anlamında kullanılabilir. Gezinme yordamı şudur:

```
void kok_ortada_dolas(agac_gost L) {
  agac_gost yigit[MAX_SIZE];
  int ust = -1;
  for (;;) {
    for (; L; L = L->solcocuk)  // solları at
      yigit_ekle(&ust, L, yigit);
    L = yigit_al(&ust, yigit);  // boşsa NULL
    if (!L) break;
    printf("%c", L->veri);
    L = L->sagcocuk;
  }
}
```

Kod satır satır şöyle çalışır:

1. `agac_gost yigit[MAX_SIZE]; int ust = -1;` yerel bir yığın açar. `ust` = −1 boş yığın demektir.
2. Dış döngü `for (;;)` sonsuz bir döngüdür; tek çıkışı aşağıdaki `break`'tir.
3. İç döngü `for (; L; L = L->solcocuk)`, `L`'den başlayıp sola inebildiği kadar iner ve geçtiği her düğümün adresini yığına ekler. Döngü `L` `NULL` olunca biter: en soldaki düğümün sol çocuğuna gelinmiştir.
4. `L = yigit_al(&ust, yigit);` yığının üstündeki düğümü alır. Bu düğümün sol alt ağacı ya boştur ya da tamamen yazılmıştır.
5. `if (!L) break;` yığın boşsa gezinme bitmiştir.
6. `printf("%c", L->veri);` düğümü yazar.
7. `L = L->sagcocuk;` düğümün sağ alt ağacına geçer. Dış döngünün bir sonraki turu bu alt ağaçta yine sola iner. Sağ alt ağaç boşsa iç döngü hiç dönmez ve yığından bir sonraki düğüm alınır.

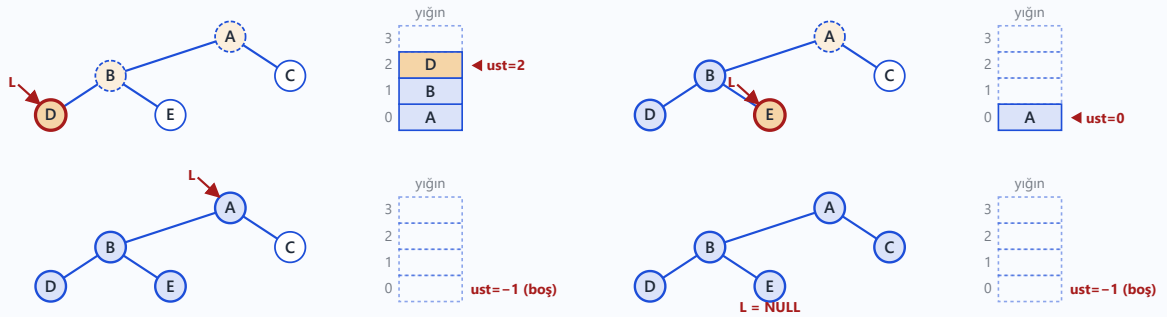
Fonksiyon boş ağaçla da doğru çalışır: `L` `NULL` ise iç döngü dönmez, yığın boş olduğu için `yigit_al` `NULL` verir ve döngü biter. Yığın dolarsa program "Yigit dolu" iletisiyle sonlanır. `MAX_SIZE` en az ağacın derinliği kadar seçilmelidir; derinlik önceden bilinmiyorsa yığın büyüyen bir diziyle tutulur.

10.6.3 Kod izleme

İzleme beş düğümlü A(B(D, E), C) ağacı üzerindedir: A'nın çocukları B ve C, B'nin çocukları D ve E'dir. Tablo 10.8 her adımda yığının içeriğini ve çıktıyı verir. Değerler, yordamın her ekleme ve almada yığını yazdıran bir sürümüyle üretildi.

Adım	İşlem	Yığın	`ust`	`L` sonra	Çıktı
1	A eklenir, sola inilir	A	0	B	
2	B eklenir, sola inilir	A B	1	D	
3	D eklenir, sola inilir	A B D	2	NULL	
4	D alınır ve yazılır; sağa geçilir	A B	1	NULL	D
5	B alınır ve yazılır; sağa geçilir	A	0	E	DB
6	E eklenir, sola inilir	A E	1	NULL	DB
7	E alınır ve yazılır; sağa geçilir	A	0	NULL	DBE
8	A alınır ve yazılır; sağa geçilir	(boş)	−1	C	DBEA
9	C eklenir, sola inilir	C	0	NULL	DBEA
10	C alınır ve yazılır; sağa geçilir	(boş)	−1	NULL	DBEAC
11	`yigit_al` `NULL` döndürür: `break`	(boş)	−1	—	DBEAC

Tablo 10.8: Yığınla kök ortada gezinme, A(B(D, E), C). Yığın alttan üste yazılmıştır.



Şekil 10.15: Yığınla kök ortada gezinmenin dört anı: A, B, D yığındayken sola inmenin bitmesi; B yazıldıktan sonra sağ çocuk E'ye geçiş; A yazılınca sağ çocuk C'ye geçiş; yığın boşalınca gezinmenin sonu. Sağdaki sütun yığın dizisidir, `ust` üst gözü gösterir.

Çıktı DBEAC'dir ve özyinelemeli `kok_ortada_dolas`'ın çıktısıyla aynıdır. Yığında aynı anda en çok üç adres bulunmuştur (adım 3): A, B ve D. Bu sayı, kökten en soldaki düğüme giden yoldaki düğüm sayısıdır. Yığının iki rolü vardır. Bir yandan özyinelemeli sürümdeki çağrı yığınının yerini tutar. Ayrıca bir düğümün "solu bitti, sırada kendisi var" anını işaretler: bir düğüm yığından alındığında sol alt ağacı tamamen yazılmıştır.

10.6.4 Maliyet ve seçici yazdırma

Zaman. Her düğüm yığına tam bir kez eklenir ve tam bir kez alınır. İç döngünün toplam tur sayısı n, dış döngünün tur sayısı $n + 1$'dir (sonuncusu `break` ile biter). Toplam iş $2n$ yığın işlemiyle orantılıdır: $O(n)$.

Bellek. Yığında aynı anda en çok derinlik kadar adres bulunur: $O(h)$. Ağacın şekli bu sayıyı belirgin biçimde değiştirir. Sola çarpık dört düğümlü zincirde dört adres birden yığına girer. Sağa çarpık zincirde ise her düğüm yığına girer girmez çıkar, çünkü hiçbir düğümün sol çocuğu yoktur; yığında en çok bir adres bulunur ve bellek $O(1)$'dir. Testte iki zincir için en büyük üst indis sırasıyla 3 ve 0 çıktı.

Yordamın güzel bir yanı, yazdırma satırının değiştirilerek başka işlere uyarlanabilmesidir. `printf` satırı bir koşula bağlanırsa yalnız seçilen düğümleri yazan bir yordam elde edilir: uç düğümler, tek çocuklular, iki çocuklular. Koşul yazılırken mantık işleçlerine dikkat etmek gerekir. `if (!(L->sagcocuk) || !(L->solcocuk))` koşulu "sağ çocuğu yok ya da sol çocuğu yok" demektir. Bu koşul yaprakları ve tek çocuklu düğümleri birlikte, yani en çok bir çocuklu düğümleri yazar. Yalnız yapraklar için "sol çocuğu yok *ve* sağ çocuğu yok" gerekir: `&&`.

Dersin amacı küçük modülleri ustaca yazabilmektir; bu yüzden küçük kodlar farklı biçimlerde tekrar tekrar yazılır. Sunumdaki dokuz soru iki kalıptan birine indirgenir: özyinelemeli ya da yığınlı gezinme ile koşullu `printf`. "Sağdan sola" istenirse `solcocuk` ile `sagcocuk` yer değiştirir.

10.6.5 Yinelemeli kök önde gezinme

Kök önde gezinmenin yinelemeli sürümü, kök ortada sürümünün neredeyse aynısıdır. Tek fark yazdırmanın yeridir: düğüm yığına *girerken* yazılır.

```
void kok_onde_dolas_d(agac_gost L) {
  agac_gost yigit[MAX_SIZE];
  int ust = -1;
  for (;;) {
    for (; L; L = L->solcocuk) {
      printf("%c", L->veri);   // girerken yaz
      yigit_ekle(&ust, L, yigit);
    }
    L = yigit_al(&ust, yigit);
    if (!L) break;
    L = L->sagcocuk;
  }
}
```

Sola inen iç döngüde her düğüm önce yazılır, sonra yığına eklenir. Böylece kök, sol alt ağacından önce yazılır. Yığından alınan düğüm artık yazılmıştır; yalnız sağ çocuğuna geçmek için kullanılır. Sonuç V-L-R sırasıdır. A(B(D, E), C) ağacında çıktı ABDEC'dir.

Aynı işin ikinci bir çözümü daha vardır. Kök yığına konur. Her adımda yığından bir düğüm alınır ve yazılır; sonra önce sağ, sonra sol çocuğu (varsa) yığına eklenir. Sol çocuk en son eklendiği için ilk o alınır; bu da sol alt ağacın sağdan önce gezilmesini sağlar.

```
if (L) yigit_ekle(&ust, L, yigit);
while ((L = yigit_al(&ust, yigit))) {
  printf("%c", L->veri);
  if (L->sagcocuk)
    yigit_ekle(&ust, L->sagcocuk, yigit);
  if (L->solcocuk)
    yigit_ekle(&ust, L->solcocuk, yigit);
}
```

Bu parça bir fonksiyonun gövdesidir; başında yığın dizisi ve `ust = -1` bildirilir. `while ((L = yigit_al(&ust, yigit)))` koşulundaki çift parantez, atamanın sonucunun bilerek koşul olarak kullanıldığını derleyiciye bildirir; tek parantezle yazılsaydı `-Wall` bir uyarı verirdi. İki çözüm de A…L ağacında ABCDGHEIJKFL verdi. İkinci çözümde yığında bekleyen düğümler sağ çocuklardır; bir yoldaki her düğüm yığında en çok bir sağ çocuk bırakır, bu yüzden bellek yine *O(h)*'dir.

10.6.6 Yinelemeli kök sonda gezinme

Kök sonda gezinmeyi yinelemeli yazmak daha zordur. Kök ortada gezinmede bir düğüm yığından alındığında sol alt ağacı bitmişti ve düğüm hemen yazılabiliyordu. Kök sonda gezinmede ise düğüm, sağ alt ağacı da bitmeden yazılamaz. Bir düğüm yığının tepesine iki kez gelir: sol alt ağaçtan dönüldüğünde ve sağ alt ağaçtan dönüldüğünde. Yordam bu iki dönüşü ayırt etmelidir. Bunun için son yazılan düğüm `son` değişkeninde tutulur. Tepedeki düğümün sağ çocuğu varsa ve son yazılan düğüm o sağ çocuk değilse sağ alt ağaç henüz gezilmemiştir. Önce sağa inilir. Değilse düğüm yazılır ve yığından çıkarılır.

```
void kok_sonda_dolas_d(agac_gost L) {
  agac_gost yigit[MAX_SIZE], son = NULL, t;
  int ust = -1;
  while (L || ust != -1) {
    if (L) {                    // sola in
      yigit_ekle(&ust, L, yigit);
      L = L->solcocuk;
    } else {
      t = yigit[ust];           // tepeye bak
      if (t->sagcocuk && t->sagcocuk != son)
        L = t->sagcocuk;        // önce sağ
      else {
        printf("%c", t->veri);
        son = yigit_al(&ust, yigit);
      }
    }
  }
}
```

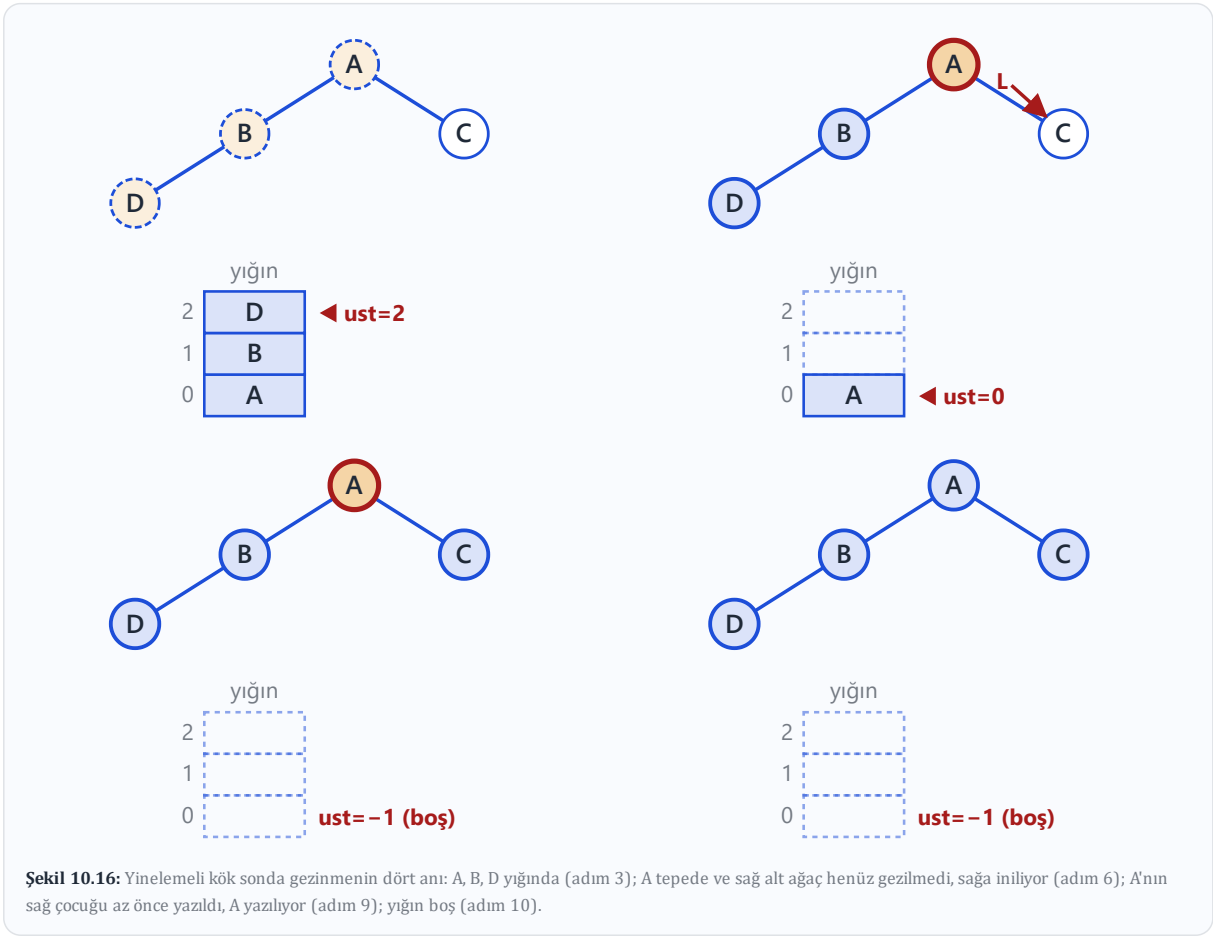
1. Döngü, gidilecek bir düğüm (`L`) ya da yığında bekleyen bir düğüm olduğu sürece döner.
2. `L` boş değilse düğüm yığına eklenir ve sola inilir. Bu dal, özyinelemeli sürümdeki “sol alt ağacı gez” çağrısına karşılık gelir.
3. `L` boşsa sola inmek bitmiştir. `t = yigit[ust]` tepedeki düğüme bakar ama onu yığından almaz.
4. `t` 'nin sağ çocuğu varsa ve `son` o çocuk değilse sağ alt ağaç henüz gezilmemiştir; `L` sağ çocuğa ayarlanır.
5. Değilse (sağ çocuk yok ya da az önce yazıldı) `t` 'nin iki alt ağacı da bitmiştir. `t` yazılır, yığından alınır ve `son` 'a kaydedilir.

`son` değişkeninin neden yalnız sağ çocukla karşılaştırıldığı şöyle açıklanır. Kök sonda sırada bir düğümden hemen önce yazılan düğüm, sağ alt ağacı varsa onun köküdür, yani sağ çocuktur. Sağ alt ağaç boşsa sol çocuktur ya da düğüm yapraktır. Bu yüzden “son yazılan, sağ çocuk mu?” sorusu sağ alt ağacın bitip bitmediğini kesin olarak söyler.

Adım	Olay	Yığın	`son`	Çıktı
1	`L` = A yığına eklenir, `L` = B	A	NULL	
2	`L` = B yığına eklenir, `L` = D	A B	NULL	
3	`L` = D yığına eklenir, `L` = NULL	A B D	NULL	
4	Tepe D: sağ çocuğu yok → D yazılır, alınır	A B	D	D
5	Tepe B: sağ çocuğu yok → B yazılır, alınır	A	B	DB

Adım	Olay	Yığın	`son`	Çıktı
6	Tepe A: sağ çocuğu C var ve `son` ≠ C → `L` = C	A	B	DB
7	`L` = C yığına eklenir, `L` = NULL	A C	B	DB
8	Tepe C: sağ çocuğu yok → C yazılır, alınır	A	C	DBC
9	Tepe A: sağ çocuğu C = `son` → A yazılır, alınır	(boş)	A	DBCA
10	`L` NULL ve yığın boş: döngü biter	(boş)	A	DBCA

Tablo 10.9: Yinelemeli kök sonda gezinme, A(B(D), C). Yığın alttan üste yazılmıştır.



Şekil 10.16: Yinelemeli kök sonda gezinmenin dört anı: A, B, D yığında (adım 3); A tepede ve sağ alt ağaç henüz gezilmedi, sağa iniliyor (adım 6); A'nın sağ çocuğu az önce yazıldı, A yazılıyor (adım 9); yığın boş (adım 10).

A tepeye iki kez gelir: adım 6'da ve adım 9'da. İlkinde `son` = B'dir ve sağ alt ağaç henüz gezilmemiştir; ikincisinde `son` = C'dir ve A yazılır. **Zaman** yine *O(n)*'dir: her düğüm bir kez eklenir, bir kez çıkarılır ve tepedeyken ona en çok iki kez bakılır. **Bellek** *O(h)*'dir. Yordam A...L ağacında GHDJKIECBLFA, A(B(D, E), C) ağacında DEBCA verdi; ikisi de özyinelemeli sürümle aynıdır.

Bir başka çözüm iki yığın kullanır: V-R-L sırası birinci yığınla üretilip ikinci yığına itilir, ikinci yığın boşaltılınca L-R-V çıkar. Bu çözüm daha kolay anlaşılır, ama *O(n)* ek bellek ister.

Sık yapılan hata

Tepedeki düğüme bakmak yerine onu yığından almak: `t = yigit_al(&ust, yigit);` . Sağ alt ağaç henüz gezilmemişse `t` 'nin yığına geri konması gerekir; konmazsa sağ alt ağaç gezildikten sonra `t` hiç yazılmaz. Kök sonda gezinmede düğüm yalnız yazılacağı anda yığından alınır; o ana kadar tepede bekler.

10.6.7 Uç düğümleri yazmak

Uç düğümleri (yaprakları) soldan sağa yazmak için herhangi bir gezinme kullanılabilir, çünkü yaprakların birbirine göre sırası üç standart gezinmede de aynıdır. Kök ortada gezinmede `printf` bir koşula bağlanır:

```
void uc_dugum_yaz(agac_gost L) {
  if (L) {
    uc_dugum_yaz(L->solcocuk);
    if (!L->solcocuk && !L->sagcocuk)
      printf("%c", L->veri);
    uc_dugum_yaz(L->sagcocuk);
  }
}
```

Koşul, düğümün iki çocuğunun da olmamasıdır. A...L ağacında çıktı GHJKL'dir. Sağdan sola yazmak için yalnız iki özyinelemeli çağrının sırası değiştirilir: önce sağ, sonra sol.

```
void uc_dugum_yaz_sag(agac_gost L) {        /* SORU 4 */
  if (L) {
    uc_dugum_yaz_sag(L->sagcocuk);
    if (!L->solcocuk && !L->sagcocuk)
      printf("%c", L->veri);
    uc_dugum_yaz_sag(L->solcocuk);
  }
}
```

Bu yordam aynı ağaçta LKJHG verir; soldan sağa çıktının tersidir. Yinelemeli sürümde de aynı yol izlenir. Yığınla kök ortada yordamında `printf` satırı şu koşula bağlanır:

```
if (!L->solcocuk && !L->sagcocuk)
  printf("%c", L->veri);
```

Sağdan sola yinelemeli yazmak için yığınlı kodda her yerde `solcocuk` ile `sagcocuk` yer değiştirir: yığına sağlar atılır, yığından alınan düğümden sonra sola gidilir.

```
void uc_dugum_yaz_sag_d(agac_gost L) {      /* SORU 6 */
  agac_gost yigit[MAX_SIZE];
  int ust = -1;
  for (;;) {
    for (; L; L = L->sagcocuk)   // sağları at
      yigit_ekle(&ust, L, yigit);
    L = yigit_al(&ust, yigit);
    if (!L) break;
    if (!L->solcocuk && !L->sagcocuk)
      printf("%c", L->veri);
    L = L->solcocuk;
  }
}
```

Bu kod bir ters kök ortada gezinmedir (R-V-L). Yapraklar ters sırada çıkar. Dört yordam da test programında aynı ağaçla çalıştırıldı: soldan sağa sürümler GHJKL, sağdan sola sürümler LKJHG verdi. Hepsi *O(n)* zamanda çalışır; özyinelemeli sürümlerin çağrı yığını ve yinelemeli sürümlerin açık yığını *O(h)* bellek ister.

10.6.8 Çocuk sayısına göre seçmek

Son üç soru, düğümleri çocuk durumlarına göre seçer. Yalnız sol çocuğu olan düğümleri yazan özyinelemeli yordam şudur:

```
void yalniz_sol_yaz(agac_gost L) {          /* SORU 7 */
  if (!L) return;
  if (L->solcocuk && !L->sagcocuk)
    printf("%c", L->veri);
  yalniz_sol_yaz(L->solcocuk);
  yalniz_sol_yaz(L->sagcocuk);
}
```

Bu yordam kök önde biçimindedir: koşulu sınayıp gerekiyorsa yazar, sonra iki alt ağaca iner. Yordamın başındaki `if (!L) return;` erken dönüşü, `if (L) { … }` bloğunun yerini tutar. `L->solcocuk && !L->sagcocuk` koşulu "sol çocuğu var ve sağ çocuğu yok" diye okunur; `!L->sagcocuk`, "sağ çocuk işaretçisi `NULL`" demektir. Öteki iki soru yalnız koşulu değiştirir. Yalnız sağ çocuğu olan düğümler için koşul `!L->solcocuk && L->sagcocuk`'tur. İki çocuklu düğümler için koşul `L->solcocuk && L->sagcocuk`'tur:

```
void iki_cocuklu_yaz(agac_gost L) {         /* SORU 9 */
  if (!L) return;
  if (L->solcocuk && L->sagcocuk)
    printf("%c", L->veri);
  iki_cocuklu_yaz(L->solcocuk);
  iki_cocuklu_yaz(L->sagcocuk);
}
```

A…L ağacında yalnız sol çocuklu tek düğüm F'dir; yalnız sağ çocuklu düğümler B ve E'dir; iki çocuklu düğümler kök önde sırayla A, C, D ve I'dır. Sayıların tutarlılığı Teorem 10.2 ile denetlenebilir: ağaçta 5 yaprak, 3 tek çocuklu ve 4 iki çocuklu düğüm vardır; *5 = 4 + 1* ve *5 + 3 + 4 = 12*. Bütün bu yordamlar "ağacı gez, koşul sağlanıyorsa yaz" kalıbının örnekleridir. Gezinmenin türü yalnız çıktının sırasını belirler; seçilen düğümlerin kümesi gezinmeden bağımsızdır.

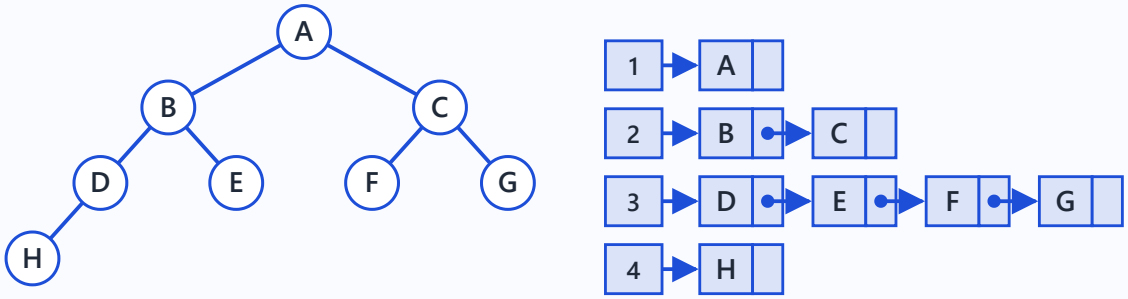
> **Sık yapılan hata**
>
> Uç düğüm koşulunda `&&` yerine `||` yazmak: `if (!L->solcocuk || !L->sagcocuk)`. Bu koşul en az bir çocuğu eksik olan bütün düğümleri yazar. A…L ağacında yapraklara ek olarak B, E ve F de yazılır ve çıktı BGHEJKLF olur. Derleyici bu hatayı fark etmez; ancak küçük bir test ağacında elle beklenen çıktıyla karşılaştırma yapılınca ortaya çıkar.

10.7 Düzey düzey gezinme

Kök önde, kök ortada ve kök sonda gezinmeler ağacın derinliklerine iner: bir dala girdiklerinde o dalı bitirmeden yan dala geçmezler. Bazı işlerde ise düğümlere kökten uzaklıklarına göre uğramak gerekir: önce kök, sonra 2. düzeydeki bütün düğümler soldan sağa, sonra 3. düzey ve böyle devam eder. Buna **düzey sırası** (*level-order*) gezinme denir. Bu kesim aynı soruyu iki yöntemle çözer ve ikinci yöntemin kuyrukla neden doğal olarak örtüştüğünü gösterir.

10.7.1 Düzey listeleriyle

İlk yöntem, gezinme sırasının bir önemi olmadığını fark etmeye dayanır. Ağaç herhangi bir gezinmeyle gezilir ve her düğüm kendi düzeyinin listesinin sonuna eklenir. Gezinme bitince listeler 1. düzeyden başlayarak sırayla yazılır. Gezinme yordamına düğümün düzeyini bir parametre olarak vermek yeterlidir: kök 1 ile çağrılır, her çocuk ebeveyninin düzeyinin bir fazlasıyla. Listeler için bir dizi bağlı liste tutulur; dizinin *i*. gözü *i*. düzeyin listesinin başını gösterir.



Şekil 10.17: Solda A...H ağacı: kök ortada HDBEAFCG, kök önde ABDHECFG, kök sonda HDEBFGCA. Sağda gezinme sırasında kurulan düzey listeleri: 1. düzey A, 2. düzey B C, 3. düzey D E F G, 4. düzey H.

Listelere ekleme her zaman sona yapıldığı için her listedeki düğümlerin sırası, gezinmenin onları bulma sırasıdır. Soldan sağa sıranın korunması için solun sağdan önce gezildiği bir gezinme (kök önde, kök ortada ya da kök sonda) seçilmelidir. Aynı düzeydeki iki düğümden soldaki, bu üç gezinmede de önce bulunur. Şekil 10.17'deki ağaçta listeler A; B, C; D, E, F, G; H olur ve art arda yazılınca ABCDEFGH çıkar.

Maliyet. Her düğüm üç kez ele alınır: gezinmede bulunur, listeye eklenir, listeden yazılır. Liste sonuna ekleme bir son işaretçisiyle $O(1)$'dir; toplam zaman $3n = O(n)$ olur. Bellek, listelerin n düğümü, düzey başları dizisinin h gözü ve gezinmenin $O(h)$ çağrı yığınıdır: $O(n + h)$. h en çok n olduğu için bu $O(n)$'dir.

10.7.2 Kuyrukla

İkinci yöntem ek listeler kurmadan aynı sırayı üretir. Düzey düzey yazmada bir düğümün çocukları, aynı düzeydeki öteki düğümlerin hepsinden sonra yazılmalıdır. Kök yazılınca çocukları B ve C sıraya girer. B yazılınca B'nin çocukları D ve E de sıraya girer, ama C'nin arkasına; çünkü C, D'den ve E'den önce yazılmalıdır. Sıraya önce giren önce çıkar: bu kuyruğun (FIFO) tanımıdır. Derinlemesine gezinmede yığın kullanmak problemin doğasından geliyordu; düzey düzey gezinmede de kuyruk kullanmak aynı biçimde zorunludur.

Algoritma şöyledir. Kök kuyruğa konur. Kuyruk boşalana kadar her adımda kuyruğun önünden bir düğüm alınır, yazılır ve çocukları (varsa önce sol, sonra sağ) kuyruğun arkasına eklenir. Ağaç boş gelebileceği için yordamın başında `if (!L) return;` sınaması vardır. Kuyruğa da düğümlerin kopyaları değil adresleri konur. Kuyruk, dairesel bir dizi olarak gerçeklenir:

```
void kuyruk_ekle(int on, int *arka,
                 agac_gost p, agac_gost k[]) {
  *arka = (*arka + 1) % MAX_SIZE;
  if (*arka == on) {            /* dolu */
    fprintf(stderr, "Kuyruk dolu\n");
    exit(EXIT_FAILURE);
  }
  k[*arka] = p;
}
agac_gost kuyruk_al(int *on, int arka,
                    agac_gost k[]) {
  if (*on == arka) return NULL; /* boş */
  *on = (*on + 1) % MAX_SIZE;
  return k[*on];
}
```

Dairesel kuyrukta iki indis tutulur. `arka` son eklenen öğenin gözünü, `on` ise ilk öğenin bir öncesindeki gözü gösterir. İkisi eşitse kuyruk boştur. `kuyruk_ekle`, `arka`'yı bir ilerletir; dizinin sonuna gelinmişse `% MAX_SIZE` ile başa döner. İlerletilen `arka` `on`'a eşit çıkarsa kuyruk doludur: bir öğe daha eklenseydi boş kuyrukla dolu kuyruk ayırt edilemezdi. Bu yüzden `MAX_SIZE` gözlü dizide en çok `MAX_SIZE - 1` öğe tutulur. `kuyruk_al`, kuyruk boşsa `NULL` döndürür; değilse `on`'u ilerletir ve o gözdeki adresi döndürür. Her yordama değiştireceği indis adresle verilir: `kuyruk_ekle` `arka`'yı, `kuyruk_al` `on`'u değiştirir. Düzey düzey gezinme yordamı şudur:

```
void duzey_dolas(agac_gost L) {
  agac_gost kuyruk[MAX_SIZE];
  int on = 0, arka = 0;
  if (!L) return;              // ağaç boş olabilir
  kuyruk_ekle(on, &arka, L, kuyruk);
  while ((L = kuyruk_al(&on, arka, kuyruk))) {
    printf("%c", L->veri);
    if (L->solcocuk)
      kuyruk_ekle(on, &arka, L->solcocuk, kuyruk);
    if (L->sagcocuk)
      kuyruk_ekle(on, &arka, L->sagcocuk, kuyruk);
  }
}
```

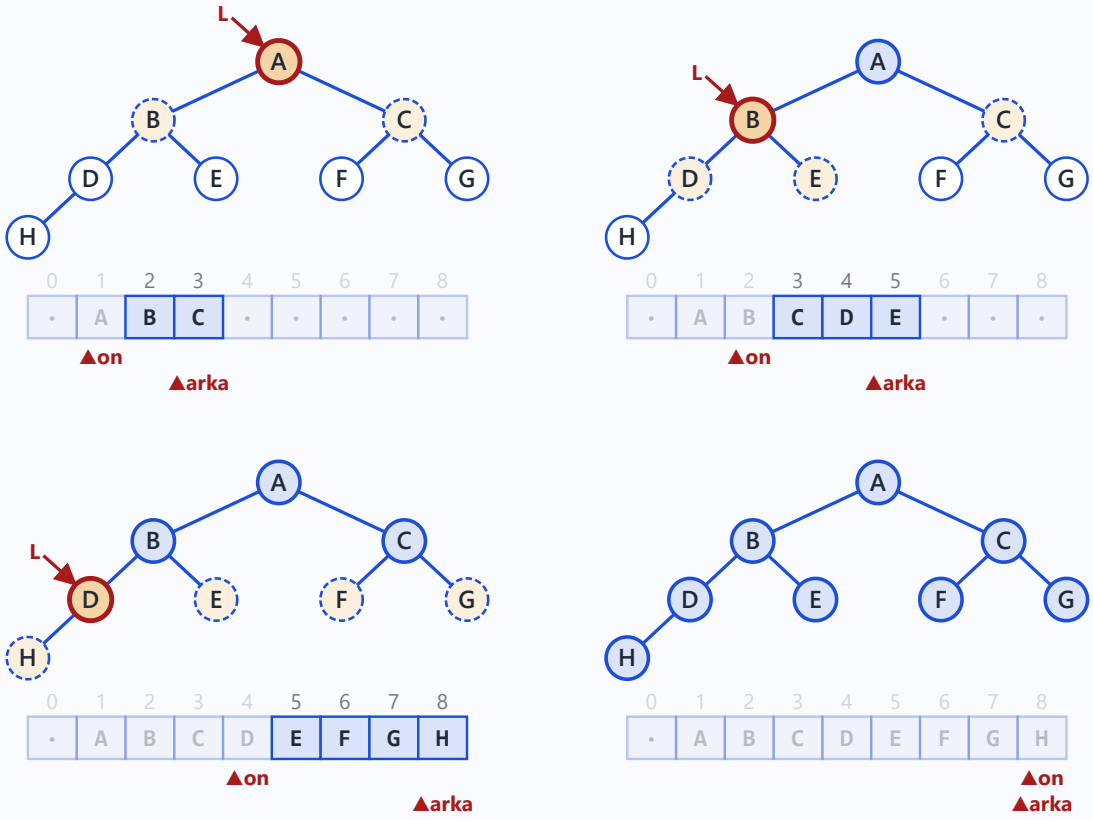
`while ((L = kuyruk_al(&on, arka, kuyruk)))` satırı hem kuyruktan düğüm alır hem de kuyruğun boşalıp boşalmadığını sınar: `kuyruk_al` `NULL` döndürünce döngü biter. Döngü gövdesi alınan düğümü yazar ve var olan çocuklarını arkaya ekler. `NULL` çocuklar kuyruğa hiç konmaz; böylece kuyruktan çıkan `NULL` yalnız “kuyruk boş” anlamına gelir.

10.7.3 Kod izleme

Tablo 10.10 A…H ağacında kuyruğun durumunu izler. Kuyruk dizisinin 0. gözü hiç kullanılmaz, çünkü ilk ekleme `arka` 'yı 0'dan 1'e ilerletir. Değerler, yordamın her almada `on` ve `arka` 'yı yazan bir sürümüyle üretildi.

Adım	Alınan (`on`)	Eklenenler	`arka`	Kuyrukta kalan	Çıktı
1	A (1)	B, C	3	B C	A
2	B (2)	D, E	5	C D E	AB
3	C (3)	F, G	7	D E F G	ABC
4	D (4)	H	8	E F G H	ABCD
5	E (5)	—	8	F G H	ABCDE
6	F (6)	—	8	G H	ABCDEF
7	G (7)	—	8	H	ABCDEFG
8	H (8)	—	8	(boş)	ABCDEFGH
9	`on == arka : NULL`	—	8	(boş)	ABCDEFGH

Tablo 10.10: `duzey_dolas` izlemesi, A…H ağacı. Başlangıçta `on` = `arka` = 0; kök eklenince `arka` = 1.



Şekil 10.18: Kuyrukla düzey düzey gezinmenin dört anı: A'nın çocukları B ve C kuyrukta; B'nin çocukları da eklendi; D'nin çocuğu H eklendi; kuyruk boşaldı. Alttaki satır kuyruk dizisidir; `on` ilk öğenin bir öncesini, `arka` son öğeyi gösterir.

Kuyruk içeriğine dikkat edilirse, herhangi bir anda kuyrukta en çok iki ardışık düzeyin düğümleri bulunur: bir düzeyin henüz yazılmamış düğümleri ve onların önündekilerin çocukları. 4. adımdan sonra kuyrukta 3. düzeyden E, F, G ve 4. düzeyden H vardır. Aynı yordam A...L ağacında ABFCLDEGHIJK verdi. Boş ağaçla çağrıldığında hiçbir şey yazmadan döndü.

10.7.4 Maliyet: yığın ile kuyruk

Zaman. Her düğüm kuyruğa bir kez girer ve bir kez çıkar: $2n = O(n)$.

Bellek. Kuyrukta aynı anda en çok bir düzeyin genişliği (w) kadar düğüm bulunur, artı sonraki düzeyden birkaç düğüm. Bu yüzden ek bellek $O(w)$'dir. Tam ağaçta son düzeyde yaklaşık $n/2$ düğüm vardır ve bellek $O(n)$ olur. Çarpık ağaçta her düzeyde tek düğüm bulunur ve kuyrukta hiçbir zaman birden çok düğüm olmaz: $O(1)$. Derinlemesine gezinmelerde durum tersidir: dengeli ağaçta yığın $O(\lg n)$, çarpık ağaçta $O(n)$ bellek ister. Tablo 10.11 iki gezinme ailesini karşılaştırır.

Gezinme	Yardımcı yapı	Ek bellek	Dengeli ağaç	Çarpık ağaç
Kök önde / ortada / sonda	yığın (LIFO)	$O(h)$	$O(\lg n)$	$O(n)$
Düzey düzey	kuyruk (FIFO)	$O(w) \le O(n)$	$O(n)$	$O(1)$

Tablo 10.11: Derinlemesine ve düzey düzey gezinmenin yardımcı yapısı ve ek belleği.

Tablodaki karşıtlık önemlidir: $O(\lg n)$ bellek sınırı kuyruklu düzey sırası gezinme için değil, dengeli ağaçta yığınlı derinlemesine gezinmeler için geçerlidir. Aynı fikirler çizgelerde de karşılığını bulur. Kuyrukla yapılan düzey düzey gezinme, çizgelerde **genişlik öncelikli arama** (*breadth-first search, BFS*) adını alır. Kuyruk yerine yığın kullanılınca algoritma **derinlik öncelikli aramaya** (*depth-first search, DFS*) döner. Çizgede bir düğüme birden çok yoldan ulaşılabildiği için orada ayrıca ziyaret işaretleri tutulur.

10.8 Diğer ağaç yordamları

Gezinme kalıbı yalnız yazdırmaya değil, ağaç üzerindeki hemen her işe uygulanır. Bu kesimdeki dört yordam aynı iskeleti kullanır: boş ağacı ele al, kökte bir iş yap, iki alt ağaç için kendini çağır. Fark, kökteki işin ne olduğunda ve özyinelemeli çağrıların sonuçlarının nasıl birleştirildiğindedir.

10.8.1 Ağacı kopyalamak

Bir ağacın kopyası, aynı şekilde ve aynı verilerle, ama tamamen yeni düğümlerden kurulmuş bir ağaçtır. Kopyada asıl ağaçla ortak tek bir düğüm bile bulunmamalıdır; yoksa kopyada yapılan bir değişiklik asıl ağacı da bozar. Yordam özyinelemeli olarak şöyle tanımlanır: boş ağacın kopyası boş ağaçtır; boş olmayan ağacın kopyası, yeni bir kökün altına iki alt ağacın kopyalarının bağlanmasıyla elde edilir.

```
agac_gost agac_kopyala(agac_gost L) {
  if (!L) return NULL;
  agac_gost T = malloc(sizeof *T);
  if (!T) { perror("malloc"); exit(1); }
  T->solcocuk = agac_kopyala(L->solcocuk);
  T->sagcocuk = agac_kopyala(L->sagcocuk);
  T->veri = L->veri;
  return T;
}
```

1. `if (!L) return NULL;` boş ağacın kopyasını döndürür. Bu, yaprakların çocukları için yapılan çağrılarda da devreye girer.
2. `agac_gost T = malloc(sizeof *T);` yeni kök için yer ayırır. Ayırma başarısızsa `perror` bir hata iletisi yazar ve program sonlanır. Burada `ogeal` yerine doğrudan `malloc` ve denetim kullanılmıştır; iki biçim de doğrudur.
3. `T->solcocuk = agac_kopyala(L->solcocuk);` sol alt ağacın kopyasını kurar ve yeni kökün sol bağına yazar.
4. `T->sagcocuk = agac_kopyala(L->sagcocuk);` aynı işi sağ alt ağaç için yapar.
5. `T->veri = L->veri;` veriyi kopyalar ve `return T;` yeni kökün adresini çağırana verir.

Veri, iki çocuk kopyalandıktan sonra yazılır; yordam ağacı kök sonda biçiminde gezer. Kopya aşağıdan yukarı kurulur: önce yapraklar, sonra onların ebeveynleri. Dersin hocası bu yordamın aşağıdan yukarı (*bottom-up*) çalıştığını, yukarıdan aşağı (*top-down*) da yazılabileceğini belirtir. Satırların yeri değiştirilerek kök önde ya da kök ortada biçimine kolayca geçilir: örneğin `T->veri = L->veri;` satırı `malloc`'un hemen arkasına alınırsa veri, çocuklardan önce kopyalanır. Sonuç değişmez. Önerilen çalışma yolu, kodu yazmadan önce tahtadaki gibi kâğıt üzerinde küçük bir ağaçla çağrıları çizmek ve yordamın özyinelemeli olarak nasıl çözüldüğünü görmektir.



Şekil 10.19: A(B, C) ağacının kopyalanması. Solda B'nin kopyası tamamlanmış ve A'nın kopyasının sol bağına yazılmak üzere; A'nın kopyasının verisi henüz yazılmamıştır (soru işareti). Sağda iş bitmiştir: kopya, asıl ağaçla aynı şekilde ama ayrı düğümlerden oluşur.

A(B, C) ağacının kopyalanması on altı adımda biter: üç düğüm için üç `malloc`, dört `NULL` çağrısı, üç veri kopyası ve dönen adreslerin bağlara yazılması. Test programında kopyanın kök önde ve kök ortada çıktıları asıl ağacınkilerle aynı oldu (ABC ve BAC); kopyanın düğüm adreslerinin asıl ağaçtakilerden farklı olduğu da denetlendi. A…L ağacının kopyası `esitmi` ile asıl ağaçla karşılaştırıldı ve eşit bulundu. **Maliyet:** zaman $O(n)$, çağrı yığını $O(h)$; kopyanın kendisi de n düğüm yer kaplar. Kopya artık bağımsız bir ağaçtır ve işi bitince `agac_sil` ile ayrıca silinmelidir.

10.8.2 Eş biçimlilik

İki ikili ağaç, aynı şekle sahipse ve karşılıklı düğümlerdeki veriler aynıysa eşittir. Yalnız şekil aranıyorsa ağaçlara **eş biçimli** (*isomorphic*) denir. Özyinelemeli tanım şöyledir. İki ağaç da boşsa eşittir. Biri boş, öteki doluysa eşit değildir. İkisi de doluysa, kökteki veriler eşit, sol alt ağaçlar eşit ve sağ alt ağaçlar eşit olmalıdır.

```c
int esitmi(agac_gost L, agac_gost T) {
  return (!L && !T) ||
         (L && T && L->veri == T->veri
          && esitmi(L->solcocuk, T->solcocuk)
          && esitmi(L->sagcocuk, T->sagcocuk));
}
```

Fonksiyon tek bir `return` ifadesinden oluşur. `||`'nin sol tarafı, iki ağacın da boş olduğu durumu yakalar ve 1 verir. Sağ taraf ise `&&` ile bağlanmış beş koşuldur: `L` dolu, `T` dolu, veriler eşit, sol alt ağaçlar eşit, sağ alt ağaçlar eşit. C'nin `&&` ve `||` işleçleri kısa devre yapar: sonucu belli olunca sağ tarafı hesaplamaz. Bu iki yarar sağlar. Birincisi, `L` ya da `T` `NULL` ise `L->veri`'ye hiç erişilmez; `L && T` sınaması bir koruyucu görevi görür. İkincisi, ilk farkta özyineleme durur ve geri kalan düğümlere bakılmaz. Yalnız biçim benzerliği isteniyorsa `L->veri == T->veri` koşulu kaldırılır.

Adım	Çağrı	Olay	Sonuç
1	(A, A)	ikisi de dolu, veriler eşit; sol alt ağaçlara geçilir	
2	(B, B)	ikisi de dolu, veriler eşit; sol alt ağaçlara geçilir	
3	(NULL, NULL)	`!L && !T` doğru; `||`'nin sağı hesaplanmaz	1
4	(B, B)	sol alt ağaçlar eşit; sağ alt ağaçlara geçilir	
5	(NULL, NULL)	ikisi de boş	1
6	(B, B)	bütün koşullar doğru	1
7	(A, A)	sol alt ağaçlar eşit; sağ alt ağaçlara geçilir	
8	(C, D)	veriler farklı; `&&` kısa devre yapar, alt ağaçlara bakılmaz	0
9	(A, A)	sağ taraf 0 döndü	0

Tablo 10.12: `esitmi` izlemesi, `L` = A(B, C) ve `T` = A(B, D).

İzlemede toplam beş çağrı yapılır: (A, A), (B, B), iki kez (NULL, NULL) ve (C, D). C ile D'nin çocuklarına hiç bakılmaz. **Maliyet.** En kötü durumda, ağaçlar eşitse, iki ağacın bütün düğümleri karşılaştırılır. Ağaçlar farklıysa yordam ilk farkta durur; iki ağacın küçüğünün düğüm sayısını aşan karşılaştırma yapılmaz. Zaman $O(min(n_1, n_2))$, çağrı yığını $O(h)$'dir.

Dersin hocası burada bir karşıtlığa dikkat çeker: ikili ağaçların eş biçimli olup olmadığını denetlemek bu kadar kolaydır, oysa iki genel çizgenin eş biçimliliğini bulmak çok zor bir problemdir. Farkın nedeni ikili ağaçtaki sıralı yapıdır. Her düğümün sol ve sağ çocuğu belli olduğu için hangi düğümün hangisiyle eşleşeceği kendiliğinden bellidir. Çizgede düğümlerin bir sırası yoktur ve olası eşleşmelerin sayısı düğüm sayısının faktöriyeli kadar büyür.

10.8.3 Erken dönüşlü yazım

Aynı yordam tek bir uzun ifade yerine, her koşulu ayrı bir satırda sınayıp sonucu belli olunca hemen dönen biçimde de yazılabilir. Bu yazıma erken dönüşlü (*guard*) yazım denir:

```c
int esitmi2(agac_gost L, agac_gost T) {
  if (L == NULL || T == NULL)
    return L == T;          /* ikisi de NULL mı? */
  if (L->veri != T->veri)
    return 0;
  if (!esitmi2(L->solcocuk, T->solcocuk))
    return 0;
  return esitmi2(L->sagcocuk, T->sagcocuk);
}
```

İlk sınama iki ağaçtan en az biri boşken çalışır. Bu durumda `return L == T;` ifadesi ikisinin de boş olup olmadığını söyler: biri `NULL` iken `L == T` ancak öteki de `NULL` ise doğrudur. İkinci sınama verileri karşılaştırır. Üçüncüsü sol alt ağaçları karşılaştırır ve eşit değilse hemen 0 döndürür. Son satır sonucu doğrudan sağ alt ağaçların karşılaştırmasına bırakır. Davranış `esitmi` ile aynıdır; testte iki yordam bütün ağaç çiftlerinde aynı sonuçları verdi.

İki yazımda da her özyinelemeli çağrının dönüş değeri kullanılır. Bu, özyinelemeli fonksiyon yazarken sık atlanan bir noktadır. `esitmi2(L->solcocuk, T->solcocuk);` satırı sonucunu bir yere aktarmadan yazılsaydı hesaplanan değer kaybolurdu ve fonksiyon yalnız köklerin verisine bakan bir sınamaya dönerdi. Hocanın ifadesiyle, "benimki güzel çalışıyor ama geri dönmüyor" diyen öğrenci özyinelemeli algoritma yazmayı henüz öğrenmemiştir: her çağrı bir değer döndürmeli ve çağıran o değeri kullanmalıdır.

> **Sık yapılan hata**
>
> Özyinelemeli çağrının sonucunu kullanmamak ya da `return` 'den sonra iş yapmaya çalışmak. `if (L->veri != T->veri) return 0; esitmi2(L->solcocuk, T->solcocuk); return 1;` biçimindeki bir yordam derlenir ve iki ağaç gerçekten eşitken doğru sonuç verir. Ama farklı alt ağaçlar için de 1 döndürür, çünkü alt çağrının sonucu atılmıştır. `return` satırından sonraki satırlar ise hiç çalışmaz.

10.8.4 Sol ve sağ çocukları değiştirmek

Bir ağacın aynadaki görüntüsü, her düğümün sol ve sağ alt ağaçlarının yer değiştirmesiyle elde edilir. Yordam verileri değil, yalnız işaretçileri takas eder:

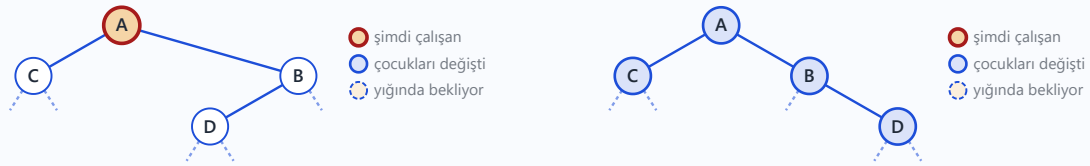
```
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
void cocukdegistir(agac_gost L) {
  agac_gost p;
  if (L) {
    SWAP(L->solcocuk, L->sagcocuk, p);
    cocukdegistir(L->solcocuk);
    cocukdegistir(L->sagcocuk);
  }
}
```

`SWAP` makrosu üç argüman alır: takas edilecek iki değer ve geçici bir değişken. `((t) = (x), (x) = (y), (y) = (t))` virgül işleciyle üç atamayı tek bir ifadede yapar: `x` 'in değeri `t` 'ye saklanır, `y` `x` 'e yazılır, saklanan değer `y` 'ye yazılır. Makro parametrelerinin parantez içinde kullanılması, `L->solcocuk` gibi karmaşık argümanlarla da doğru açılmasını sağlar. Yordamda geçici değişken `agac_gost p` 'dir. Kök takas edilir, sonra iki alt ağaç için yordam yinelenir. Takas, çağrılardan önce yapıldığı için yordam kök önde biçimindedir.

Takastan sonra `L->solcocuk` eski sağ alt ağaçtır. Bu yüzden A(B(D), C) ağacında yordam önce C'ye, sonra B'ye iner. Tablo 10.13 adımları verir.

Adım	Çağrı	Olay
1	A	sol (B) ve sağ (C) takas edilir; A'nın solu artık C
2	C	sol (NULL) ve sağ (NULL) takas edilir
3–4	NULL, NULL	C'nin iki boş çocuğu için çağrılar hemen döner
5	B	sol (D) ve sağ (NULL) takas edilir; B'nin sağı artık D
6	NULL	B'nin yeni sol çocuğu boştur
7	D	sol (NULL) ve sağ (NULL) takas edilir
8–9	NULL, NULL	D'nin iki boş çocuğu için çağrılar döner
10	—	bitti: ağaç A(C, B(—, D)) oldu

Tablo 10.13: `cocukdegistir` izlemesi, A(B(D), C).

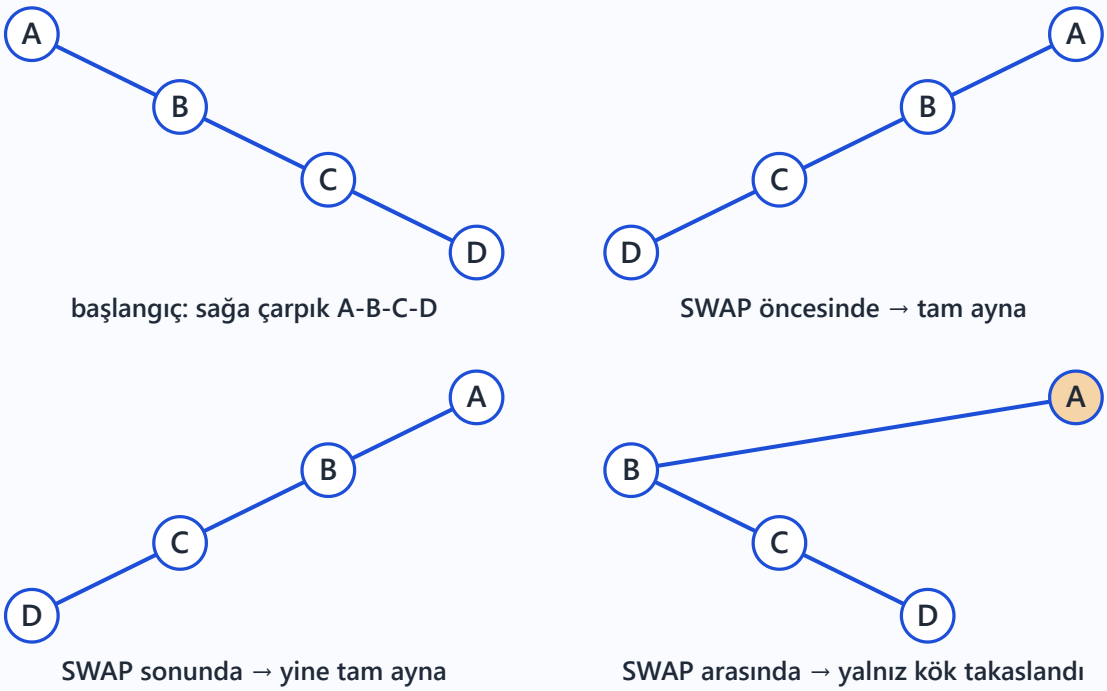


Şekil 10.20: Solda ilk adım: A'nın çocukları takas edildi, C sola geçti. Sağda son durum: ağaç aynadaki görüntüsüne döndü. Kök ortada sıra DBAC'den CABD'ye, yani tersine döndü.

Sonuçta kök ortada sıra DBAC'den CABD'ye döner. Ayna ağacın kök ortada sırası her zaman asıl ağacınkinin tersidir, çünkü ayna her düğümde L-V-R sırasını R-V-L sırasına çevirir. Yordam her düğüme bir kez uğrar ve her düğümde sabit sayıda atama yapar: zaman *O(n)*, bellek *O(h)*. Hiçbir düğüm ayrılmaz ya da silinmez; veriler yerinde kalır, yalnız işaretçiler yer değiştirir. Yordam iki kez uygulanırsa ağaç ilk hâline döner.

10.8.5 Takasın yeri

`SWAP` satırı iki özyinelemeli çağrının önüne, arasına ya da arkasına konabilir. Önde ve arkada iken sonuç aynıdır; arada iken sonuç bozulur. Bunu görmek için bütün sağ alt ağaçları tek kollu olan sağa çarpık bir zincir kullanılır: A'nın sağında B, B'nin sağında C, C'nin sağında D. Şekil 10.21 dört durumu gösterir.



Şekil 10.21: Sağa çarpık A-B-C-D zinciri (sol üst) ve takasın üç konumunun sonucu. Takas çağrılardan önce ya da sonra yapılırsa sonuç sola çarpık tam aynadır. Takas iki çağrının arasında yapılırsa yalnız kökün çocukları yer değiştirir; B alt ağacına hiç dokunulmaz.

- **Takas önde (kök önde):** A'da sol ve sağ yer değiştirir; B artık A'nın solundadır ve yordam B'ye iner. B'de de takas olur ve C sola geçer. Sonuç sola çarpık bir zincirdir: tam ayna.
- **Takas arkada (kök sonda):** Önce iki alt ağaç kendi içinde aynalanır, sonra kökte takas yapılır. Her alt ağaç aynalandıktan sonra yer değiştirdiği için sonuç yine tam aynadır.
- **Takas arada (kök ortada):** A'da önce sol alt ağaç işlenir; sol boş olduğu için hiçbir şey olmaz. Sonra takas yapılır ve B sola geçer. Ardından "sağ" alt ağaç işlenir; ama sağ artık boştur. B'nin alt ağacına hiç inilmez. Sonuçta yalnız kök takaslanmıştır.

Üç sürüm test programında çalıştırıldı. Takas önde ve arkadayken kök ortada sıra DCBA çıktı ve iki ağaç `esitmi` ile eşit bulundu. Takas aradayken kök ortada sıra BCDA çıktı. Genel ders şudur: yordamın her çağrısı, işleyeceği alt ağacı çağrı anındaki işaretçilerden okur. Takas, bu işaretçileri çağrıların arasında değiştirirse ikinci çağrı yanlış alt ağaca gider.

10.9 Gezinmelerden ağacı yeniden kurmak

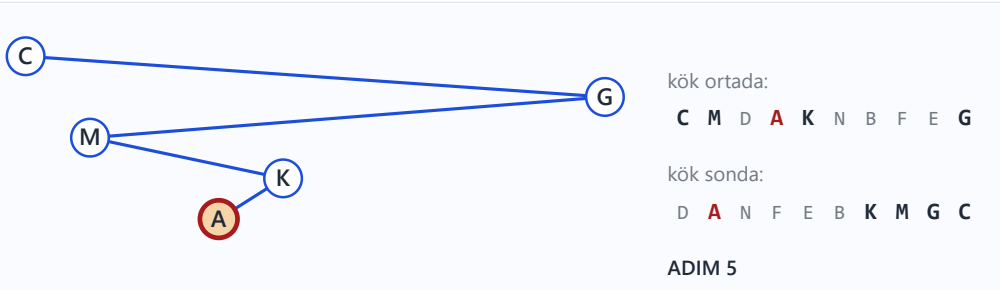
Bir ağacın tek bir gezinme çıktısı ağacı belirlemez. Örneğin kök ortada çıktısı AB olan iki ağaç vardır: A kök ve B sağ çocuk ya da B kök ve A sol çocuk. İki gezinme birlikte verilince ise çoğu durumda ağaç tek olarak bulunur. Bu, sık sorulan bir sınav sorusudur. Sunumdaki soruda kök ortada sıra `C M D A K N B F E G`, kök sonda sıra `D A N F E B K M G C`'dir. Bu sıraları veren ikili ağaç çizilecektir.

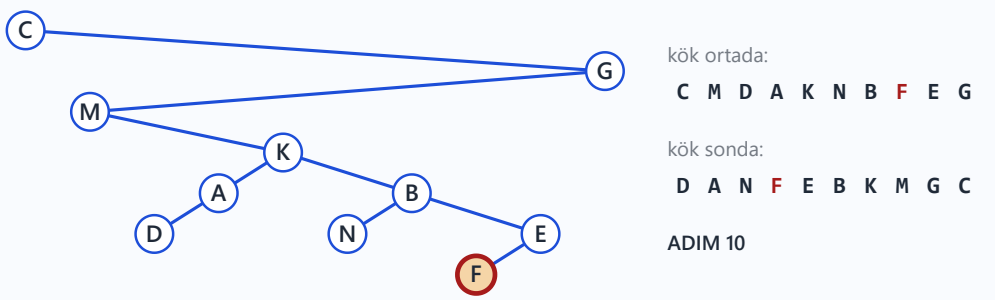
Yöntem iki gözleme dayanır. Kök sonda sırada her alt ağacın kökü o alt ağacın en son yazılan düğümüdür. Kök ortada sırada bir düğümün solundaki harfler onun sol alt ağacını, sağındaki harfler sağ alt ağacını oluşturur. Kök sonda sıra sondan başa okunur. Her harf bir alt ağacın köküdür; kök ortada sıra bu harfin solunu ve sağını ayırır. Sondan başa okumada önce sağ alt ağacın, sonra sol alt ağacın kökleri gelir, çünkü kök sonda sırada sağ alt ağaç kökten hemen öncedir.

Adım	Kök sondadan okunan	Kök ortada sıradaki konumu	Yerleşim
1	C (en son)	solunda harf yok	C köktür; sol alt ağaç boş, sağ alt ağaç MDAKNBFEG
2	G	sağında harf yok	G, C'nin sağ çocuğu; G'nin sağı boş, solu MDAKNBFE
3	M	bu alt ağaçta solunda harf yok	M, G'nin sol çocuğu; M'nin solu boş
4	K	solunda D, A; sağında N, B, F, E	K, M'nin sağ çocuğu
5	B	K'nın sağ alt ağacının kökü	B, K'nın sağ çocuğu; solunda N, sağında F, E
6	E	B'nin sağ alt ağacının kökü; solunda F	E, B'nin sağ çocuğu
7	F	E'nin solunda	F, E'nin sol çocuğu
8	N	B'nin solunda	N, B'nin sol çocuğu
9	A	K'nın sol alt ağacı D, A'nın kökü	A, K'nın sol çocuğu
10	D	A'nın solunda	D, A'nın sol çocuğu; ağaç tamam

Tablo 10.14: Kök ortada CMDAKNBFEG ve kök sonda DANFEBKMGC sıralarından ağacın kurulması.

Tablodaki sıra, kök sonda dizgisinin sondan başa okunuşudur: C, G, M, K, B, E, F, N, A, D. Sunum aynı ağacı adım adım çizerken K'nın sol alt ağacını (D ve A) sağ alt ağacından önce yerleştirir; iki sıra da aynı ağacı verir. Şekil 10.22 ara bir anı ve sonucu gösterir.





Şekil 10.22: Ağacın yeniden kurulması. Üstte sunumun sırasıyla 5. adım: C, G, M, K yerleşmiş, A K'nın soluna konmuştur. Altta sonuç: C(—, G(M(—, K(A(D, —), B(N, E(F, —)))), —)). Ağaç tektir.

Sonuç ağacı test programıyla denetlendi: kurulan ağacın kök ortada gezinmesi CMDAKNBFEG, kök sonda gezinmesi DANFEBKMGC verdi. Yöntem doğrudan özyinelemeli bir C fonksiyonuna dönüşür:

```
/* ko: kök ortada, ks: kök sonda dizgisi; ikisi de n harf */
agac_gost yeniden_kur(const char *ko, const char *ks, int n) {
  if (n == 0) return NULL;
  agac_gost T = ogeal();
  T->veri = ks[n - 1];              /* kök sonda: kök en sonda */
  int i = 0;
  while (ko[i] != T->veri) i++;     /* kök ortada kökü bul */
  T->solcocuk = yeniden_kur(ko, ks, i);
  T->sagcocuk = yeniden_kur(ko + i + 1, ks + i, n - i - 1);
  return T;
}
```

`ko` kök ortada, `ks` kök sonda dizgisidir; ikisi de aynı alt ağacın `n` harfini tutar. Kök, `ks`'nin son harfidir. Kök ortada dizgide kökün konumu `i` bulunur: solundaki `i` harf sol alt ağaçtır, sağındaki `n - i - 1` harf sağ alt ağaçtır. Kök sonda dizgide de sol alt ağaç ilk `i` harfi, sağ alt ağaç onu izleyen `n - i - 1` harfi kaplar. Sol alt ağaç `(ko, ks, i)` ile kurulur. Sağ alt ağaç `(ko + i + 1, ks + i, n - i - 1)` ile kurulur. İşaretçi aritmetiği alt dizgilerin kopyalanmasını önler: `ko + i + 1`, dizginin `i + 1`. karakterinden başlayan parçasıdır. Fonksiyon harflerin birbirinden farklı olduğunu ve iki dizginin aynı ağaca ait olduğunu varsayar. Her düğüm için kök ortada dizgide doğrusal bir arama yapılır; çarpık ağaçta toplam zaman $O(n^2)$, dengeli ağaçta $O(n \lg n)$ olur.

Kök önde ve kök ortada sıraları da ağacı tek olarak belirler; tek fark kökün kök önde sıranın *ilk* harfi olmasıdır. Kök önde ve kök sonda sıraları ise genellikle belirlemez. A kök ve B tek çocuk olan iki ağacı düşünelim: birinde B soldadır, ötekinde sağdadır. İkisinin de kök önde sırası AB, kök sonda sırası BA'dır. Bu iki gezinme kökü ve çocuğu bulur, ama tek çocuğun hangi yanda olduğunu söylemez. Kök ortada sıra ise tam bu bilgiyi taşır: sol alt ağaç kökten önce, sağ alt ağaç kökten sonra yazılır.

10.10 Karmaşıklık özeti

Tablo 10.15 bu nottaki bütün yordamların zaman ve ek bellek maliyetlerini toplar. n düğüm sayısı, h derinliktir; $\lg n \le h \le n$ (dengeli ↔ çarpık). Derinlik bu derste düzey sayısıyla ölçülür.

İşlem	Zaman	Ek bellek	Açıklama
Ebeveyn / çocuk (dizi gösterimi)	$O(1)$	$2^k - 1$ göz	tam ağaçta kayıp yok
Özyinelemeli kök önde / ortada / sonda	$O(n)$	$O(h)$	$2n+1$ çağrı, $n+1$'i `NULL`
Yığınlı kök önde / ortada / sonda	$O(n)$	$O(h)$	yığına adres konur
Düzey düzey (kuyruk)	$O(n)$	$O(w) \le O(n)$	w: en geniş düzey
Uç düğüm yazma, çocuk sayısına göre seçme	$O(n)$	$O(h)$	gezinme + koşul
`hesapla` (ifade ağacı)	$O(n)$	$O(h)$	kök sonda
`agac_kopyala`	$O(n)$	$O(h)$ + n düğüm	kök sonda
`agac_sil`	$O(n)$	$O(h)$	kök sonda zorunlu

İşlem	Zaman	Ek bellek	Açıklama
`esitmi`, `esitmi2`	$O(\min(n_1, n_2))$	$O(h)$	ilk farkta durur
`cocukdegistir`	$O(n)$	$O(h)$	yalnız işaretçiler
`yeniden_kur`	$O(n^2)$ en kötü	$O(h)$ + n düğüm	kökü aramak doğrusal

Tablo 10.15: Ağaç yordamlarının karmaşıklığı (n düğüm, h derinlik).

Bütün düğümlere dokunan her yordam $\Omega(n)$ zaman ister; buradaki yordamların hepsi bu alt sınıra ulaşır. Bellekte belirleyici olan ağacın şeklidir: dengeli ağaç birkaç düzine çerçeveyle, çarpık ağaç n çerçeveyle gezilir.

10.11 Sık yapılan hatalar

1. **`L->veri`'ye `NULL` denetimi yapmadan erişmek.** Ağaç işaretçisi `NULL` gelebilir: ağaç boş olabilir ya da özyineleme bir yaprağın boş çocuğuna inmiş olabilir. Her ağaç yordamının ilk sorusu "ağaç boş mu?" olmalıdır.
2. **Dizi gösteriminde 0'dan başlayıp 1 tabanlı formülleri kullanmak.** $2i$ ve $2i+1$ formülleri kök 1. gözdeyken doğrudur. Kök 0. gözdeyse formüller $2i+1$, $2i+2$ ve $\lfloor (i-1)/2 \rfloor$ olur.
3. **Yığına ya da kuyruğa düğümün kopyasını koymak.** Düğüme erişmek için adresi yeter. Kopya hem daha çok yer tutar hem de gereksiz bir bellek ayırma ya da yapı kopyası demektir.
4. **Özyinelemeli çağrının dönüş değerini kullanmamak ya da `return`'den sonra iş yapmaya çalışmak.** Değer döndüren her özyinelemeli çağrının sonucu ya döndürülmeli ya da bir değişkene alınmalıdır.
5. **Uç düğüm koşulunda `&&` yerine `||` yazmak.** "Çocuğu yok" iki çocuğun da olmaması demektir. `||` ile tek çocuklu düğümler de yazılır.
6. **Derece ile derinliği, düzey 1 ile derinlik 0 tanımlarını karıştırmak.** Derece genişliği, derinlik uzunluğu ölçer. Kökün düzeyi bu derste 1'dir; başka kaynaklarda 0 olabilir ve formüller buna göre bir kayar.
7. **Ağacı silerken kökü önce bırakmak.** `free(L)`'den sonra `L->solcocuk`'u okumak tanımsız davranıştır. Silme kök sonda sırayla yapılmalıdır.

10.12 Alıştırmalar ve çözümleri

Sunumun alıştırma slaytındaki altı soru aşağıda çözümleriyle verilmiştir. Kod içeren çözümler derlenip küçük test programlarıyla çalıştırıldı; ağaç çizimi isteyen çözümler bir Python betiğiyle denetlendi.

10.12.1 Yaprak sayısı eşitliği

Soru. $n_0 = n_2 + 1$ eşitliğini kanıtlamak.

Çözüm. Düğümler derecelerine göre sayılırsa $n = n_0 + n_1 + n_2$ olur. Kök dışındaki her düğüme yukarıdan bir kenar girer; kenar sayısı $n - 1$'dir. Aynı kenarlar çıktıkları düğümlere göre sayılırsa $n - 1 = n_1 + 2n_2$ bulunur. İki eşitlik taraf tarafa çıkarılınca $1 = n_0 - n_2$ kalır. Kanıtın ayrıntısı Teorem 10.2'de verilmiştir. A…L ağacında $n_0 = 5$, $n_1 = 3$, $n_2 = 4$ sayıldı; eşitlik sağlanır.

10.12.2 Bir ifade ağacının üç gezinmesi

Soru. $(A + B) * (C - D)$ ifadesinin ağacını çizip üç gezinmeyi yazmak.

Çözüm. En son uygulanan işleç `*`'dır; kök odur. Sol alt ağacın kökü `+`, çocukları A ve B'dir. Sağ alt ağacın kökü `-`, çocukları C ve D'dir. Kök önde gezinme `* + A B - C D` verir. Kök ortada gezinme `A + B * C - D`, kök sonda gezinme `A B + C D - *` verir. Kök ortada çıktıda parantezler kaybolmuştur; ifadeyi doğru okumak için her alt ağaç paranteze alınmalıdır. Ön-işleç ve son-işleç çıktıları parantezsiz de tek anlamlıdır.

10.12.3 Düğüm sayısı

Soru. Ağacın düğüm sayısını döndüren `int say(agac_gost L)` yordamını yazmak.

```
int say(agac_gost L) {           /* düğüm sayısı */
  return L ? 1 + say(L->solcocuk) + say(L->sagcocuk) : 0;
}
```

Çözüm. Boş ağaçta düğüm yoktur: 0. Boş olmayan ağaçta kök için 1, iki alt ağaçtaki düğümler için iki özyinelemeli çağrı toplanır. Koşul işleci (`? :`) iki durumu tek bir `return` ifadesinde birleştirir. Yordam kök sonda biçimindedir: kökün değeri, alt ağaçların değerleri bilindikten sonra hesaplanır. Zaman *O(n)*, bellek *O(h)*'dir. Test programında A…L ağacı için 12, A(B(D), C) için 4, boş ağaç için 0 döndü. Aynı kalıpla yaprak sayısı (`L` yaprak ise 1, değilse iki alt ağacın toplamı) ve derinlik de yazılır.

10.12.4 Kök önde ve kök ortada sıradan ağaç

Soru. Kök önde sırası `A B D E C F`, kök ortada sırası `D B E A F C` olan ağacı çizmek.

Çözüm. Kök önde sıranın ilk harfi köktür: A. Kök ortada sırada A'nın solunda D, B, E; sağında F, C vardır. Sol alt ağacın kök önde sırası B, D, E'dir; kökü B'dir. Kök ortada sırada B'nin solunda D, sağında E vardır: B(D, E). Sağ alt ağacın kök önde sırası C, F'dir; kökü C'dir. Kök ortada sırada F, C'nin solundadır: C(F, —). Ağaç A(B(D, E), C(F, —)) olur. Bu yöntem Tablo 10.14'teki yöntemin aynısıdır; tek fark kökün baştan okunmasıdır.

10.12.5 Düzeyleri sağdan sola yazmak

Soru. Ağacın düzeylerini sırayla, ama her düzeyi sağdan sola yazan kuyruklu yordamı yazmak.

```
void duzey_dolas_sag(agac_gost L) {         /* düzeyler sağdan sola */
  agac_gost kuyruk[MAX_SIZE];
  int on = 0, arka = 0;
  if (!L) return;
  kuyruk_ekle(on, &arka, L, kuyruk);
  while ((L = kuyruk_al(&on, arka, kuyruk))) {
    printf("%c", L->veri);
    if (L->sagcocuk)
      kuyruk_ekle(on, &arka, L->sagcocuk, kuyruk);
    if (L->solcocuk)
      kuyruk_ekle(on, &arka, L->solcocuk, kuyruk);
  }
}
```

Çözüm. `duzey_dolas`'tan tek farkı, çocukların kuyruğa ekleniş sırasıdır: önce sağ, sonra sol çocuk eklenir. Aynı düzeydeki düğümler kuyruğa sağdan sola girer ve aynı sırayla çıkar. A…H ağacında çıktı ACBGFEDH oldu: 1. düzey A, 2. düzey C B, 3. düzey G F E D, 4. düzey H. Maliyet `duzey_dolas` ile aynıdır.

10.12.6 İki ağaç birbirinin aynası mı?

Soru. İki ağacın birbirinin aynası olup olmadığını denetleyen yordamı yazmak.

```
int ayna_mi(agac_gost L, agac_gost T) {
  return (!L && !T) ||
         (L && T && L->veri == T->veri
          && ayna_mi(L->solcocuk, T->sagcocuk)
          && ayna_mi(L->sagcocuk, T->solcocuk));
}
```

Çözüm. `esitmi`'den tek farkı çapraz karşılaştırmadır: `L`'nin solu `T`'nin sağıyla, `L`'nin sağı `T`'nin soluyla karşılaştırılır. A(B(D), C) ağacı `cocukdegistir` ile aynalandıktan sonra asıl ağaçla karşılaştırıldı ve 1 döndü. Bir ağacın kendisiyle karşılaştırılması ise 0 verdi, çünkü ağaç simetrik değildir. Zaman $O(\min(n_1, n_2))$, bellek *O(h)*'dir.

10.13 Özet

- Ağaç, bir kökten başlayıp ayrık alt ağaçlara bölünen hiyerarşik yapıdır. Kök dışındaki her düğümün tek ebeveyni vardır; bu yüzden ağaçta döngü yoktur ve n düğümde $n - 1$ kenar bulunur.
- Temel terimler: ebeveyn, çocuk, kardeş, ata, soy, derece, yaprak, iç düğüm, düzey ve derinlik. Bu derste kök 1. düzeydedir ve derinlik en büyük düzeydir; CLRS gibi kaynaklar kökü 0 sayar.
- İkili ağaçta i. düzeyde en çok 2^{i-1}, k derinlikli ağaçta en çok $2^k - 1$ düğüm vardır. $\lfloor \lg(n+1) \rfloor \le k \le n$; alt sınıra tam ağaç, üst sınıra çarpık ağaç ulaşır.
- $n_0 = n_2 + 1$'dir. Bağlı gösterimde $2n$ işaretçinin $n + 1$'i `NULL` 'dır.
- Tam ve sabit ağaç diziyle (1 tabanlı, $\lfloor i/2 \rfloor$, $2i$, $2i + 1$), şekli değişen ağaç `agac_gost` bağlarıyla tutulur. Çarpık ağaç dizide $2^k - 1$ göz ister.
- Genel ağaç, sol çocuk – sağ kardeş gösterimiyle ikili ağaca çevrilir; kök önde sıra korunur, ikili ağacın kök ortada sırası genel ağacın kök sonda sırasına eşittir.
- Kök önde, kök ortada ve kök sonda gezinmeler aynı özyinelemeli iskelettir; $2n + 1$ çağrı yapar, $O(n)$ zaman ve $O(h)$ bellek ister. İfade ağacı kök sonda gezinmeyle hesaplanır.
- Gezinmeler açık bir yığınla yinelemeli yazılır; yığına düğüm adresleri konur. Kök sonda sürümde son yazılan düğüm tutulur.
- Düzey düzey gezinme kuyruk ister; belleği en geniş düzeyle orantılıdır.
- Kopyalama, eş biçimlilik, aynalama, sayma ve silme “gez ve işle” kalıbının örnekleridir. Silme ve kopyalama kök sonda sırayla yapılır.
- Kök ortada sıra ile kök sonda ya da kök önde sıra birlikte ağacı tek olarak belirler; kök önde ile kök sonda sıra belirlemez.

10.14 Kavram listesi

ağaç *tree* · kök *root* · alt ağaç *subtree* · çizge *graph* · ebeveyn *parent* · çocuk *child* · kardeş *sibling* · ata *ancestor* · soy *descendant* · derece *degree* · yaprak *leaf* · iç düğüm *internal node* · düzey *level* · derinlik *depth* · yükseklik *height* · ikili ağaç *binary tree* · çarpık ağaç *skewed tree* · mükemmel ikili ağaç *perfect binary tree* · tam ikili ağaç *complete binary tree* · kılavuzlu ikili ağaç *threaded binary tree* · gezinme *traversal* · kök önde *preorder* · kök ortada *inorder* · kök sonda *postorder* · düzey sırası *level-order* · ifade ağacı *expression tree* · çağrı yığını *call stack* · dairesel kuyruk *circular queue* · eş biçimli *isomorphic* · genişlik öncelikli arama *breadth-first search* · derinlik öncelikli arama *depth-first search* · dengeli ağaç *balanced tree*

10.15 Kendini Dene

1. A–M genel ağacında (A'nın çocukları B, C, D; B'nin çocukları E, F; E'nin çocukları K, L; C'nin çocuğu G; D'nin çocukları H, I, J; H'nin çocuğu M) H'nin düzeyi, D'nin derecesi, E'nin ataları ve B'nin soyu nedir?

   Cevap

   H 3. düzeydedir. D'nin derecesi 3'tür (H, I, J). E'nin ataları A ve B'dir. B'nin soyu E, F, K ve L'dir.

2. Derinliği 5 olan bir ikili ağaçta en çok kaç düğüm bulunur? Bu ağacın 4. düzeyinde en çok kaç düğüm olabilir?

   Cevap

   En çok $2^5 - 1 = 31$ düğüm. 4. düzeyde en çok $2^3 = 8$ düğüm.

3. 20 düğümlü bir tam ikili ağaç 1 tabanlı bir dizide tutuluyor. Ağacın derinliği kaçtır? 14 numaralı düğümün ebeveyni ve çocukları hangileridir? 10 numaralı düğümün çocukları var mıdır?

   Cevap

   Derinlik $\lfloor \lg 21 \rfloor = 5$'tir. 14'ün ebeveyni $\lfloor 14/2 \rfloor = 7$'dir; çocukları 28 ve 29 olurdu, ikisi de 20'yi aştığı için yoktur (14 bir yapraktır). 10'un sol çocuğu 20 vardır; sağ çocuğu 21 yoktur.

4. Bir ikili ağaçta 7 iki çocuklu ve 3 tek çocuklu düğüm vardır. Kaç yaprak vardır? Bağlı gösterimde kaç `NULL` işaretçi bulunur?

   Cevap

   $n_0 = n_2 + 1 = 8$ yaprak vardır. Toplam düğüm sayısı $8 + 3 + 7 = 18$'dir ve `NULL` işaretçi sayısı $n + 1 = 19$'dur.

5. Ağaçta A'nın çocukları B ve C, B'nin çocukları D ve E'dir; C'nin yalnız sağ çocuğu F vardır. Dört gezinmenin (kök önde, kök ortada, kök sonda, düzey sırası) çıktıları nedir?

   Cevap

   Kök önde ABDECF, kök ortada DBEACF, kök sonda DEBFCA, düzey sırası ABCDEF.

6. Son-işleç gösterimi `A B + C *` olan ifadenin ağacı nasıldır? Ön-işleç gösterimi nedir?

   Cevap

   Kök `*`'dır; sol alt ağacı `+` (A, B), sağ çocuğu C'dir. İfade $(A + B) * C$'dir. Ön-işleç gösterimi `* + A B C`'dir.

7. Yığınla kök ortada gezinme 1000 düğümlü sola çarpık bir zincire uygulanıyor. Yığında aynı anda en çok kaç adres bulunur? Zincir sağa çarpık olsaydı?

   Cevap

   Sola çarpık zincirde kökten başlayarak bütün düğümler sola inilirken yığına girer: 1000 adres (`MAX_SIZE` en az bu kadar olmalıdır). Sağa çarpık zincirde hiçbir düğümün sol çocuğu olmadığı için her düğüm girer girmez çıkar: en çok 1 adres.

8. `esitmi`, A(B, C) ile A(B, C(D, —)) ağaçlarıyla çağrılıyor. Sonuç nedir ve toplam kaç çağrı yapılır?

   Cevap

   Sonuç 0'dır. Çağrılar: (A, A), (B, B), B'nin iki boş çocuğu için iki (NULL, NULL), (C, C) ve C'lerin sol çocukları için (NULL, D). Son çağrıda iki koşul da yanlış çıkar ve 0 döner; `&&` kısa devre yaptığı için C'lerin sağ çocuklarına bakılmaz. Toplam 6 çağrı.

9. Kök önde sırası ABDGCEF, kök ortada sırası DGBAECF olan ağaç hangisidir?

   Cevap

   Kök A'dır. Sol alt ağaç {D, G, B}: kökü B, B'nin solunda D, G vardır; D kök, G onun sağ çocuğudur. Sağ alt ağaç {E, C, F}: kökü C, sol çocuğu E, sağ çocuğu F. Ağaç A(B(D(—, G), —), C(E, F)).

10. A(B(D, E), C) ağacına `cocukdegistir` uygulanıyor. Yeni ağacın kök ortada sırası nedir? Yordam ikinci kez uygulanırsa ne olur?

    Cevap

    Asıl kök ortada sıra DBEAC'dir; ayna ağacınki bunun tersidir: CAEBD. İkinci uygulama her düğümde takası geri alır ve ağaç ilk hâline döner.

10.16 İleri okuma

- E. Horowitz, S. Sahni ve S. Anderson-Freed, *Fundamentals of Data Structures in C* (2. baskı), Bölüm 5: terminoloji, ikili ağaç lemmaları, dizi ve bağlı gösterim, gezinmeler, ağaç kopyalama ve eşitlik yordamlarının bu notun izlediği kaynağıdır; kılavuzlu ikili ağaçlarla devam eder.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (3. baskı), Ek B.5 ve Bölüm 12: ağaçların matematiksel tanımlarını (yükseklik 0 tabanlı) ve ikili arama ağaçlarında gezinmeyi verir.
- R. Sedgewick, *Algorithms in C, Parts 1–4* (3. baskı), Bölüm 5: özyineleme ile ağaçları birlikte işler; gezinmelerin yığın ve kuyrukla yinelemeli sürümlerini ve ağaç özelliklerinin kanıtlarını içerir.
- D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms* (3. baskı), Kesim 2.3–2.3.2: ikili ağaç gezinme algoritmalarını yığınla verir ve genel ağaçların ikili ağaçlarla gösterimini ayrıntılı inceler.

10.17 Kaynaklar

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, Bölüm 5 (Trees).
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3. baskı, MIT Press, Ek B.5 ve Bölüm 12.
3. R. Sedgewick, *Algorithms in C, Parts 1–4*, 3. baskı, Addison-Wesley, Bölüm 5 (Trees).
4. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3. baskı, Addison-Wesley, 1997, Kesim 2.3.

KISIM III · BÖLÜM 11

Arama Ağaçları

Sıralı bir dizide ikili arama her adımda kalan aralığın yarısını eler ve aranan değeri *O(lg n)* adımda bulur. Ne var ki sıralı diziye yeni bir öğe eklemek, arkadaki bütün öğeleri bir hücre kaydırmayı gerektirir. Bağlı listede ekleme ucuzdur, ama bu kez ortadaki öğeye doğrudan ulaşılamaz ve arama doğrusal kalır. İkili arama ağacı bu iki yapının iyi yanlarını birleştirir. Her düğüm, solundaki bütün değerlerden büyük ve sağındaki bütün değerlerden küçüktür. Böylece kökten başlayan her karşılaştırma ağacın bir yanını eler. Ekleme ve silme ise yalnız birkaç işaretçiyi değiştirir. Not önce ikili arama ağacını tanımlar. Ardından ekleme, arama, derinlik ve silme yordamlarını C ile yazar ve her birini adım adım izler. Son kesimler ağacın dengesini konu alır. Sıralı gelen verinin ağacı listeye çevirdiği gösterilir; bunu önleyen AVL ağacının döndürmeleri ayrıntılı, kırmızı-siyah ağaç ile B-ağacı ise kısaca anlatılır.

Kod (C): `github.com/sercankulcu/data-structures-and-algorithms-c/Bolum11`

ÖĞRENME ÇIKTILARI

Tanımı uygulamak
İkili arama ağacının dört özelliğini verilen bir ağaçta denetlemek ve kök ortada gezinmenin değerleri neden sıralı verdiğini açıklamak.

Ekleme ve arama
`arama_agacina_ekle` ve `arama_agacinda_ara` yordamlarını yinelemeli ve özyinelemeli yazmak; boş ağaca eklemede kök işaretçisinin neden adresle gönderildiğini açıklamak.

Derinliği hesaplamak
Derinliği özyinelemeli yazmak; küresel değişkenden, gereksiz ek parametreden ve makronun çift değerlendirme tuzağından kaçınmak.

Silmek
Çocuksuz, tek çocuklu ve iki çocuklu düğüm silmeyi işaretçinin işaretçisiyle yazmak; dört silme yönteminin ağaç derinliğine etkisini karşılaştırmak.

Dengeyi korumak
Dengesizliğin maliyetini hesaplamak; AVL ağacında LL, RR, LR ve RL durumlarını tanıyıp tek ve çift döndürmeyi uygulamak.

Yapı seçmek
İkili arama ağacını, AVL ağacını, kırmızı-siyah ağacı ve B-ağacını kullanım yerine göre ayırmak.

11.1 İkili arama ağacı

Ağaç, her düğümün bir ebeveyni ve sıfır ya da daha çok çocuğu olan hiyerarşik bir yapıdır. Ebeveyni olmayan tek düğüme **kök**, çocuğu olmayan düğümlere **yaprak** denir. **İkili ağaçta** her düğümün en çok iki çocuğu vardır ve bunlar sol çocuk ile sağ çocuk olarak ayrılır. Bir düğüm ile onun altındaki bütün düğümler birlikte bir **alt ağaç** (*subtree*) oluşturur. Bu notta ağacın **derinliği**, kökten en uzak yaprağa giden yoldaki düğüm sayısıdır. Tek düğümlü ağacın derinliği 1, boş ağacınki 0'dır. Ağaç gezinmesinde üç sıra kullanılır. **Kök önde** (*preorder*) gezinmede önce kök, sonra sol ve sağ alt ağaç ziyaret edilir. **Kök ortada** (*inorder*) gezinmede sıra sol alt ağaç, kök, sağ alt ağaçtır. **Kök sonda** (*postorder*) gezinmede ise kök en sona kalır. Bu kavramlar ağaçlar konusundan bilinir; burada yalnız hatırlatma amacıyla verildi.

11.1.1 Neden yeni bir yapı?

Bir öğrenci işleri sistemi düşünülsün. Sistem her gün binlerce kez öğrenci numarasıyla kayıt arar. Yeni öğrenciler kaydolur, mezun olanlar sistemden çıkar. Kayıtlar sırasız bir dizide tutulursa arama için bütün dizi taranır ve maliyet *O(n)* olur. Kayıtlar numaraya göre sıralı tutulursa ikili arama *O(lg n)*'de sonuç verir. Fakat bu kez her ekleme ve silmede sırayı korumak için öğeler kaydırılır; bu da *O(n)* işlem

demektir. Bağlı listede ekleme ve silme birkaç işaretçi değişikliğiyle yapılır. Ama listenin ortasına doğrudan atlanamaz; ikili arama kullanılamaz ve arama yine *O(n)*'dir.

İkili arama ağacı ikili aramanın fikrini bağlı bir yapıya taşır. Sıralı dizide ikili arama önce ortadaki öğeye bakar. Aranan değer daha küçükse sol yarıda, daha büyükse sağ yarıda aramayı sürdürür. İkili arama ağacının kökü bu "ortadaki öğe" gibi davranır. Sol alt ağaç sol yarının, sağ alt ağaç sağ yarının yerini tutar. Fark şuradadır: yarılar dizideki ardışık hücreler değil, işaretçilerle bağlanmış düğümlerdir. Yeni bir öğe eklemek için hiçbir öğe kaydırılmaz; yalnız uygun yerdeki bir `NULL` işaretçi yeni düğümü gösterecek biçimde değiştirilir.

11.1.2 Tanım ve sıralı gezinme

Tanım 11.1

İkili arama ağacı (*binary search tree, BST*), boş olabilen ve boş değilse aşağıdaki dört özelliği taşıyan bir ikili ağaçtır:

1. Her öğenin değeri farklıdır; değerler tektir.
2. Boş olmayan sol alt ağaçtaki bütün değerler, kökteki değerden küçüktür.
3. Boş olmayan sağ alt ağaçtaki bütün değerler, kökteki değerden büyüktür.
4. Sol ve sağ alt ağaçlar da ikili arama ağacıdır.

Tanımın dördüncü maddesi onu özyinelemeli yapar: aynı koşul her düğümde, o düğümün kök olduğu alt ağaç için de geçerlidir. İkinci ve üçüncü maddelerdeki "bütün" sözcüğü önemlidir. Koşul yalnız düğümün çocuklarını değil, alt ağacın en derindeki öğelerini de kapsar. Kökün sol alt ağacının en altındaki, en sağdaki öğe bile kökten küçük olmalıdır.

Birinci madde ağacı bir **küme** gibi kullanır. Aynı değer ağaca ikinci kez girmez. Uygulamada değer çoğu zaman bir kaydın **anahtarıdır** (*key*): öğrenci numarası, T.C. kimlik numarası ya da bir sözcük. Kaydın öteki alanları anahtarla birlikte düğümde durur. Tekrarlı anahtarlara izin vermek gerekirse ya düğüme bir sayaç eklenir ya da eşit anahtarlar kurala göre hep aynı yana (örneğin sağa) konur. Bu notta tanıma uyulur: eşit değer gelirse ekleme yapılmaz.

Tanım ikili arama ağacını ikili ağaçların bir alt kümesi yapar. Her ikili arama ağacı bir ikili ağaçtır; ağaç üzerinde yazılmış gezinme, kopyalama ve eşitlik yordamları ona da uygulanır. Tersi doğru değildir. Sıradan bir ikili ağaçta değerler herhangi bir düzende durabilir. Tanımdaki düzenin en önemli sonucu aşağıdaki teoremdir.

Teorem 11.1 (Kök ortada gezinme sıralıdır)

Bir ikili ağacın ikili arama ağacı olması için gerek ve yeter koşul, kök ortada gezinmenin değerleri kesin artan sırada vermesidir.

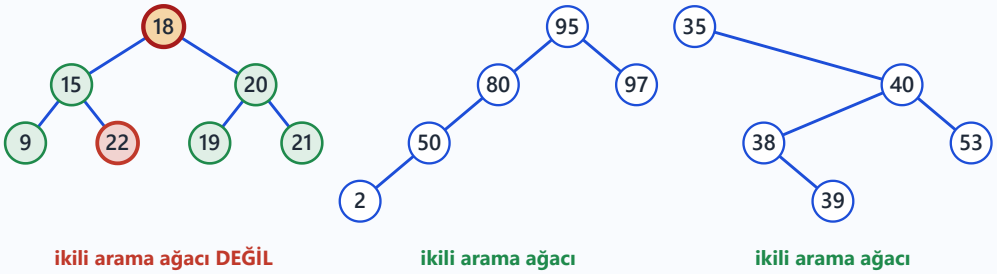
Kanıt. Önce ağacın ikili arama ağacı olduğu varsayılsın ve düğüm sayısı üzerinden tümevarım yapılsın. Boş ağaç ve tek düğümlü ağaç için iddia açıktır. Daha büyük bir ağaçta kök ortada gezinme önce sol alt ağacı, sonra kökü, sonra sağ alt ağacı yazar. Sol ve sağ alt ağaçlar da ikili arama ağacıdır; tümevarım varsayımına göre her biri kendi içinde artan sırada yazılır. Sol alt ağaçtaki bütün değerler kökten küçük, sağdakiler kökten büyüktür. Öyleyse üç parça art arda geldiğinde dizi baştan sona artandır. Ters yön için kök ortada dizinin kesin artan olduğu varsayılsın. Sol alt ağacın her değeri dizide kökten önce, sağ alt ağacın her değeri kökten sonra yazılır. Dizi artan olduğuna göre soldakiler kökten küçük, sağdakiler büyüktür. Değerler de birbirinden farklıdır. Alt ağaçların kök ortada dizileri bu dizinin ardışık parçalarıdır; onlar da artandır. Tümevarımla alt ağaçlar da ikili arama ağacıdır. ■

Teoremin pratik bir sonucu vardır. Bir değer kümesini sıralamak için değerler boş bir ikili arama ağacına tek tek eklenir, sonra ağaç kök ortada gezilir. Bu yönteme **ağaç sıralaması** (*tree sort*) denir. Ağaç dengeli kalırsa toplam maliyet *O(n lg n)* olur. Teorem ikinci bir işe de yarar: bir ağacın ikili arama ağacı olup olmadığı, kök ortada gezinip her değerin öncekinden büyük olduğuna bakılarak *O(n)* sürede denetlenir.

11.1.3 Tanımı denetlemek

Şekil 11.1 üç ikili ağacı gösterir. Soldaki ağaçta her düğüm yalnız kendi çocuklarıyla karşılaştırılırsa bir sorun görünmez. 15'in sol çocuğu 9, sağ çocuğu 22'dir ve *9 < 15 < 22*'dir. 20'nin çocukları 19 ve 21'dir; onlar da doğru yerdedir. Ancak 22, kökü 18 olan ağacın sol alt ağacındadır. Tanımın ikinci maddesi sol alt ağaçtaki *bütün* değerlerin 18'den küçük olmasını ister. 22 bu koşulu bozar; ağaç ikili arama ağacı değildir. Bu ağaçta 22 aranırsa kökte *22 > 18* karşılaştırması aramayı sağa yollar ve 22 hiçbir zaman bulunamaz.

Ortadaki ağaç dengesiz görünse de geçerlidir. 95'in solundaki her değer (80, 50, 2) 95'ten küçüktür; 80'in solu 50, 50'nin solu 2'dir ve sıralama hiçbir yerde bozulmaz. Tanım ağacın biçimi hakkında bir şey söylemez; yalnız değerlerin düzenini sınırlar. Sağdaki ağaçta kökün sağındaki her değer 35'ten büyüktür. 40'ın solunda 38 ve onun sağında 39 durur. İkisi de 40'tan küçük ve 35'ten büyüktür. Bu değerler, kökten kendilerine gelen yol üzerindeki her düğümün koşulunu sağlar; ağaç geçerlidir.



Şekil 11.1: Hangisi ikili arama ağacı? Soldaki ağaçta 22, 15'in sağında olduğu için 15'ten büyüktür; ama 18'in sol alt ağacında durduğu hâlde 18'den de büyüktür ve tanımı bozar. Ortadaki ve sağdaki ağaçlar geçerlidir: her değer, kökten kendisine gelen yoldaki bütün düğümlerin koşulunu sağlar.

Örnekler, denetimin nasıl yapılacağını da gösterir. Her düğümün değeri için bir alt ve bir üst sınır vardır. Kökün sınırı yoktur. Bir düğümden sola inildiğinde üst sınır o düğümün değeri olur, sağa inildiğinde alt sınır. Soldaki ağaçta 22'ye kökten sola, sonra sağa inilerek gelinir. Alt sınır 15, üst sınır 18'dir; 22 bu aralığa girmez. Bu fikrin C gerçeklemesi alıştırmalar kesiminde verilir.

> **Sık yapılan hata**
>
> İkili arama ağacını denetlerken her düğümü yalnız çocuklarıyla karşılaştırmak. Bu denetim Şekil 11.1'deki soldaki ağacı geçerli sayar. Doğru denetim ya kök ortada gezinmenin artan olduğuna bakar ya da her düğüme kökten gelen (alt, üst) aralığını taşır.

11.1.4 Düğümün C'deki yapısı

Ağacın her düğümü bir `struct` ile temsil edilir. Düğümde bir veri alanı ve iki işaretçi bulunur. İşaretçilerden biri sol çocuğu, öteki sağ çocuğu gösterir. Çocuk yoksa işaretçinin değeri `NULL` 'dır. Sunumdaki tanım aşağıdadır.

```
typedef enum { FALSE, TRUE } boolean;
typedef struct dugum *agac_gost;
struct dugum {
    agac_gost solcocuk;
    int       veri;
    agac_gost sagcocuk;
};
/* ogeal(): malloc + NULL denetimi,
   çocukları NULL yapar */
```

İlk satır, C'de olmayan mantıksal tipi bir sayımla (`enum`) kurar: `FALSE` 0, `TRUE` 1 değerini alır. İkinci satır `agac_gost` adını " `struct dugum` 'a işaretçi" anlamında tanımlar. Bu tanım yapının kendisinden önce gelir. C'de henüz tanımlanmamış bir yapıya işaretçi bildirmek serbesttir; derleyicinin bilmesi gereken tek şey işaretçinin boyutudur. Böylece yapının içinde kendi tipine işaretçi kullanılabilir. Bu tür yapılara **kendine başvuran yapı** (*self-referential structure*) denir.

64 bitlik bir makinede her işaretçi 8, `int` 4 bayttır. Derleyici `veri` 'den sonra 4 baytlık bir boşluk bırakır, çünkü `sagcocuk` 8'in katı olan bir adreste başlamalıdır. Bu yüzden `sizeof(struct dugum)` 24 bayttır; değer zig cc ile x86-64 üzerinde ölçülerek doğrulandı. Yani her düğüm 4 baytlık bir veri için 20 baytlık ek yük taşır. Bağlı yapıların esnekliğinin bedeli budur.

Sunum düğüm ayıran `ogeal()` yordamını yalnız açıklamasıyla verir: `malloc` ile yer ayırır, `NULL` denetimi yapar ve çocukları `NULL` yapar. Aşağıda bu tarife uyan bir gerçekleme var. Notun bütün sürücü programları bu yordamı kullandı.

```
agac_gost ogeal(void) {
  agac_gost r = malloc(sizeof *r);   /* 24 bayt */
  if (!r) {
    fprintf(stderr, "Bellek yetersiz\n");
    exit(EXIT_FAILURE);
  }
  r->solcocuk = r->sagcocuk = NULL;
  return r;
}
```

`malloc(sizeof *r)` yazımı, ayrılan boyutu tipin adı yerine işaretçinin gösterdiği nesneden alır. Yapının adı değişse de satır doğru kalır. `malloc` bellek bulamazsa `NULL` döndürür. Bu durumda `r->solcocuk`'a yazmak tanımsız davranış olur; yordam bu yüzden hatayı bildirip programı sonlandırır. Son satırdaki iki atama çok önemlidir. `malloc` belleği temizlemez; çocuk işaretçileri rastgele bir adres taşıyabilir. Ağaç yordamları bir dalın sonunu `NULL` ile tanır. Çocukları temizlenmemiş bir yaprak, yordamları belleğin rastgele bir yerine yönlendirir.

Tablo 11.1, kökü 47 olan küçük bir ağacın bellekteki görünümünü örnek adreslerle verir. Ağacın düğümleri bellekte art arda durmak zorunda değildir; her biri `malloc`'un bulduğu ayrı bir yerdedir. Ağacın biçimi yalnız işaretçi alanlarındaki adreslerle belirlenir.

Adres	`solcocuk` (8 bayt)	`veri` (4 + 4 boşluk)	`sagcocuk` (8 bayt)
`0x1A40` (kök)	`0x1B10`	47	`0x1A70`
`0x1A70`	`NULL`	55	`NULL`
`0x1B10`	`NULL`	26	`NULL`

Tablo 11.1: 47 kökünün solunda 26, sağında 55 bulunan üç düğümlü ağacın bellekteki görünümü. Adresler örnek amaçlıdır; her düğüm 24 bayttır.

Tabloda kök işaretçisi `L`'nin değeri `0x1A40`'tır. `L->veri` 47'yi, `L->solcocuk->veri` 26'yı verir. 26 bellekte 55'ten sonra durur, ama ağaçta 55'in solundadır. Düğümlerin bellek sırası ile ağaçtaki yerleri arasında bir ilişki yoktur.

11.2 Ekleme

İkili arama ağacına ekleme, bağlı listeye eklemeden daha kısıtlıdır. Listede yeni öğe başa, sona ya da araya konabilir. İkili arama ağacında ise yeni değerin yeri tektir ve ağacın içeriğiyle belirlenir. Bu kesim önce bu yerin nasıl bulunduğunu, sonra ekleme sırasının ağacın biçimini nasıl değiştirdiğini anlatır. Ardından ekleme yordamı C ile yazılır ve adım adım izlenir.

11.2.1 Yeri ağaç belirler

Sunumda hoca bu noktayı bir sınıf anısıyla anlatır. Bir öğrenciden bir değer söylemesi istenir. Öğrenci “15'in sağına ekleyin” diye yer de gösterir. Hoca yerin öğrencinin değil, ağacın işi olduğunu söyler. Gerçekten de ekleme yordamının imzası yalnız eklenecek değeri ve ağacı alır; yer bilgisi almaz. Yordam yeni değeri kökten başlayarak arar, bulamadığı yerde ekler. Değer ağaçta yoksa arama bir `NULL` işaretçide biter. O `NULL`, yeni değerin tanımı bozmadan durabileceği tek yerdir.

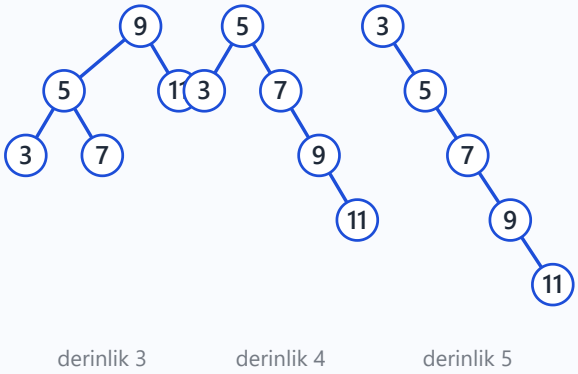
Yer tek olsa da aynı değer kümesi farklı ağaçlarla gösterilebilir. Ağacın biçimi, değerlerin hangi sırayla eklendiğine bağlıdır. Sunumdaki soru bu durumu küçük bir örnekle sorar.

Örnek 11.1

Soru. 3, 5, 7, 9, 11 değerleriyle derinliği 3, 4 ve 5 olan ikili arama ağaçları çizilsin. 2 derinliğinde bir ağaç çizilebilir mi?

Çözüm. Derinliği 5 olan ağaç en kolayıdır: değerler artan sırada eklenir. Her yeni değer öncekilerin hepsinden büyüktür ve en sağdaki düğümün sağına konur. Sonuç 3 → 5 → 7 → 9 → 11 biçiminde sağa uzanan bir zincirdir. Derinlik 4 için 5, 3, 7, 9, 11 sırası kullanılabilir: 5 köktür, 3 onun soluna, 7 sağına girer; 9 ve 11 yine sağa uzanır. Derinlik 3 için 9, 5, 3, 7, 11 sırası yeter. Kök 9'dur; 5 ve 11 onun çocuklarıdır; 3 ve 7 de 5'in çocukları olur. Bu ağaç kök önde sırasıyla 9 5 3 7 11, parantezli gösterimle 9(5(3, 7), 11)'dir. Üç ağacın kök ortada gezinmesi aynıdır: 3 5 7 9 11.

2 derinliğinde bir ağaç çizilemez. Derinliği 2 olan bir ikili ağaçta kök ve onun en çok iki çocuğu bulunur; toplam en çok 3 düğüm sığar. Beş değer bu ağaca sığmaz.



Şekil 11.2: 3, 5, 7, 9, 11 değerleriyle kurulmuş derinliği 3, 4 ve 5 olan üç ikili arama ağacı. Üçünün de kök ortada gezinmesi 3 5 7 9 11'dir; yalnız ekleme sırası farklıdır.

Örneğin son adımındaki sayım genel bir sınır verir.

Teorem 11.2 (Derinlik ve düğüm sayısı)

Derinliği k olan bir ikili ağaçta en çok $2^k - 1$ düğüm bulunur. Buna göre n düğümlü bir ikili ağacın derinliği en az $\lceil \lg(n+1) \rceil$, en çok n'dir.

Kanıt. Kök 1. düzeydedir. Her düğümün en çok iki çocuğu olduğundan i. düzeyde en çok 2^{i-1} düğüm bulunur. Derinliği k olan ağaçta düzeyler 1'den k'ye kadardır. Toplam en çok $1 + 2 + 4 + \dots + 2^{k-1} = 2^k - 1$ olur. n düğüm sığması için $2^k - 1 \ge n$, yani $k \ge \lg(n+1)$ gerekir; k tam sayı olduğundan $k \ge \lceil \lg(n+1) \rceil$'dir. Üst sınır için her düzeyde en az bir düğüm bulunduğu yeterlidir: k düzey en az k düğüm ister, öyleyse $k \le n$'dir. ■

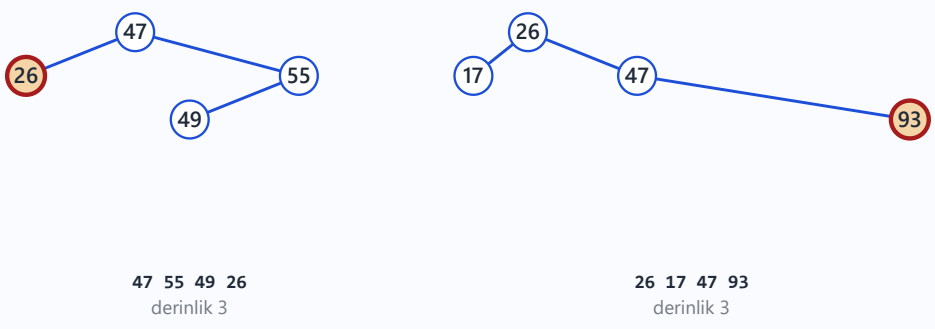
$n = 5$ için alt sınır $\lceil \lg 6 \rceil = 3$, üst sınır 5'tir. Şekil 11.2 bu iki ucu ve aradaki derinliği gösterir. Sınırların anlamı açıktır. Ağaçta arama en kötü durumda derinlik kadar düğüme bakar. Derinliği $\lceil \lg(n+1) \rceil$'e yakın bir ağaç ikili arama kadar hızlıdır. Derinliği n'ye yakın bir ağaç ise bağlı listeden farksızdır.

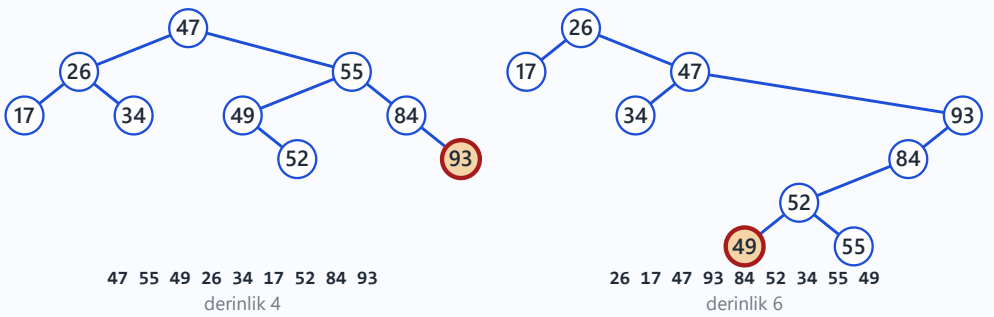
11.2.2 Ekleme sırası ağacın biçimini değiştirir

Sunum aynı dokuz değeri iki farklı sırayla iki boş ağaca ekler. Değerler 17, 26, 34, 47, 49, 52, 55, 84, 93'tür. Soldaki ağaca ekleme sırası 47, 55, 49, 26, 34, 17, 52, 84, 93'tür. Sağdaki ağaca ise 26, 17, 47, 93, 84, 52, 34, 55, 49 sırasıyla eklenir. Her adımda kural aynıdır: kökten başlanır, yeni değer düğümdekinden küçükse sola, büyükse sağa inilir ve ilk `NULL`'a konur.

Soldaki ağaçta ilk değer 47 kök olur. 55 ondan büyüktür ve sağına girer. 49, 47'den büyük olduğu için sağa iner; orada 55'ten küçük olduğu için 55'in soluna konur. 26 ve 34 kökün solunda yerleşir; 17 de 26'nın solunu alır. 52, 47 → 55 → 49 yolunu izleyip 49'un sağına; 84, 55'in sağına; 93 de 84'ün sağına girer. Sonuçta en uzun yol 47 → 55 → 49 → 52'dir ve ağacın derinliği 4'tür.

Sağdaki ağaçta kök 26'dır. 17 onun soluna, 47 sağına girer. Sonraki 93, 47'nin sağına; 84, 93'ün soluna; 52 de 84'ün soluna iner. Her yeni değer bir öncekinin altına eklendiği için ağaç aşağı doğru uzar. 34, 47'nin soluna yerleşir. 55, 26 → 47 → 93 → 84 → 52 yolundan geçip 52'nin sağına, 49 da 52'nin soluna konur. En uzun yol 26 → 47 → 93 → 84 → 52 → 55'tir ve derinlik 6'dır. Şekil 11.3 ilk dört eklemeden sonraki ve en sondaki durumu gösterir.





Şekil 11.3: Aynı dokuz değer, iki farklı sırayla iki boş ağaca ekleniyor. Üstte her iki dizinin ilk dört değeri eklenmiştir. Altta bütün değerler eklenmiştir: soldaki ağacın derinliği 4, sağdakinin 6'dır. İki ağacın kök ortada gezinmesi aynıdır: 17 26 34 47 49 52 55 84 93.

İki ağaç aynı kümeyi tutar ve ikisi de tanıma uyar. Kök ortada gezinmeleri, Teorem 11.1 gereği aynı sıralı diziyi verir. Ama arama maliyetleri farklıdır. Soldaki ağaçta en kötü arama 4 düğüme, sağdakinde 6 düğüme bakar. Ekleme sırası ağacın biçimini, biçim de bütün işlemlerin süresini belirler. Sunumdaki ifadeyle verilerin eklenme sırası önemlidir.

11.2.3 Çarpıklık ve dengeli ağaçlar

Ekleme sırasının en kötü hâli, verilerin sıralı gelmesidir. 1, 2, 3, …, n sırasıyla eklenen her değer öncekilerin hepsinden büyüktür; her biri en sağdaki düğümün sağına girer. Ağaç sağa uzanan bir zincire döner. Ters sırada gelen veri de sola uzanan bir zincir kurar. Bütün düğümleri tek bir yöne uzanan bu ağaçlara **çarpık ağaç** (*skewed tree*) denir. Çarpık ağacın derinliği n'dir ve her işlem bağlı listedeki kadar sürer. Sıralı veri gerçek hayatta sanıldığından sık görülür. Tarihe göre gelen kayıtlar, artan sırayla verilen öğrenci numaraları ya da önceden sıralanmış bir dosya bu duruma örnektir.

Derinliği $lg\ n$ düzeyinde kalan ağaca **dengeli ağaç** (*balanced tree*) denir. Dengeli bir ikili arama ağacında ekleme, silme ve arama $O(lg\ n)$ sürer. Sorun, dengenin kendiliğinden korunmamasıdır. Bu soruna iki tür çözüm geliştirilmiştir.

- **Kendini dengeleyen ağaçlar** (*self-balancing trees*), her ekleme ve silmeden sonra ağacı yerel değişikliklerle yeniden dengeler. AVL ağacı ve kırmızı-siyah ağaç bu türdendir. Derinlik her durumda $O(lg\ n)$'de kalır.
- **Sık kullanılan düğümleri köke yaklaştıran ağaçlar**, dengeyi değil erişim hızını hedefler. Bir düğüme her erişildiğinde o düğüm döndürmelerle köke taşınır. Sık aranan değerler böylece kökün yakınında toplanır. Sunumda “zig-zag” adıyla anılan bu yapının kaynaklardaki adı **splay ağacıdır** (*splay tree*). Yapıyı 1985'te D. Sleator ile R. Tarjan önermiştir. Tek bir işlem pahalı olabilir, ama uzun bir işlem dizisinde işlem başına ortalama maliyet $O(lg\ n)$'dir.

Hoca bu konuda şunu söyler: ağacın dengeli olması çok önemliyse ağacı dengeli kuran algoritmalar kullanılmalıdır. Bu algoritmalar çok zor değildir, yalnız biraz daha uzundur. Notun son büyük kesimi bunların en bilinenini, AVL ağacını ayrıntılı ele alır.

11.2.4 Ağaçtan ekleme sırasına, gezinmeden ağaca

Sunum bu noktada üç soru işaretiyle iki düşündürücü soru sorar. İlki şudur: ağacın görüntüsünden verilerin eklenme sırası elde edilebilir mi? Gelen veriye göre oluşan ağaç tek midir?

Sorunun ikinci yarısının cevabı evettir. Ekleme yordamı rastgelelik içermez; aynı değerler aynı sırayla geldiğinde her seferinde aynı ağaç oluşur. Ama ters yön tek değildir. 2, 1, 3 sırası kökü 2, solu 1, sağı 3 olan ağacı kurar. 2, 3, 1 sırası da aynı ağacı kurar; çünkü 1 ile 3 birbirinin yolunda değildir ve hangisinin önce geldiği önemsizdir. Bir ekleme sırasının belirli bir ağacı kurması için tek koşul vardır: her düğüm kendi soyundan önce eklenmelidir. Bir ata, soyundaki bir düğümden sonra gelirse o düğüm başka bir yere yerleşir. Kök önde sıra bu koşulu her zaman sağlar; çünkü bir düğümü, alt ağacındaki bütün düğümlerden önce yazar. Bu yüzden ağacın kök önde gezinmesi daima geçerli bir ekleme sırasıdır.

İkinci soru şudur: kök önde, kök ortada ve kök sonda gezinme sıralarından ağacın görüntüsü elde edilebilir mi? Genel bir ikili ağaçta tek bir gezinme sırası yetmez. Örneğin kök önde sırası A B olan iki ağaç vardır: B, A'nın sol çocuğu da olabilir, sağ çocuğu da. Kök ortada sıra ile kök önde (ya da kök sonda) sıra birlikte verilirse ağaç tek olarak belirlenir. Kök önde sıranın ilk öğesi köktür. Kök ortada sırada kökün solundakiler sol alt ağacı, sağındakiler sağ alt ağacı oluşturur. Aynı işlem alt ağaçlara özyinelemeli uygulanır.

İkili arama ağacında iş kolaylaşır. Kök ortada sıra zaten bilinir: değerlerin artan sırasıdır. Bu yüzden yalnız kök önde sıra (ya da yalnız kök sonda sıra) ağacı belirlemeye yeter. Daha da kısası, kök önde sıradaki değerler boş bir ağaca o sırayla eklenir ve ağaç yeniden kurulur.

Örnek 11.2

Soru. Bir ikili arama ağacının kök önde gezinmesi 30 20 10 25 40 35 50'dir. Ağaç nedir? Kök sonda gezinmesi nedir?

Çözüm. İlk değer 30 köktür. Kalan değerlerden 30'dan küçük olanlar (20, 10, 25) sol alt ağacı, büyük olanlar (40, 35, 50) sağ alt ağacı oluşturur. Kök önde sırada da bu sırayla gelirler. Sol alt ağacın kökü 20'dir; 10 onun soluna, 25 sağına girer. Sağ alt ağacın kökü 40'tır; 35 soluna, 50 sağına girer. Ağaç 30(20(10, 25), 40(35, 50)) biçimindedir ve derinliği 3'tür. Kök sonda gezinme önce sol alt ağacı (10 25 20), sonra sağ alt ağacı (35 50 40), en son kökü yazar: 10 25 20 35 50 40 30. Değerlerin aynı sırayla boş ağaca eklenmesi de aynı ağacı verir; sonuç sürücü programla doğrulandı.

Sunumda hoca bu konuya bir gözlem de ekler. Daha önce ikili ağaç yordamları, arka planda gerçek bir ağaç olmadığı için kodu çalıştırarak denenemiyordu. Ekleme yordamı yazıldıktan sonra bu engel kalkar. Birkaç satırla örnek ağaçlar kurulur; gezinme, kopyalama ve eşitlik yordamları bu ağaçlarla gerçekten çalıştırılır. Bu notun bütün sürücü programları da ağaçlarını bu yolla kurdu.

11.2.5 Ekleme yordamı

Ekleme algoritması dört gözleme dayanır. Birincisi, her iki çocuğu da olan bir düğüme eklenemez; yeni öğe en az bir çocuğu eksik olan bir düğümün boş yerine girer. İkincisi, gelen değer kökten başlayarak aranır. Düğümdeki değer yeni değerden büyükse sola, küçükse sağa ilerlenir. Üçüncüsü, aynı değer bulunursa tanım bozulacağı için ekleme yapılmaz. Dördüncüsü, arama işaretçisinin ilk rastladığı `NULL`'ın yerine yeni öğe konur. Ama `NULL`'a gelindiğinde arama işaretçisi artık hiçbir düğümü göstermez. Yeni düğümün bağlanacağı ebeveyni hatırlamak için bir adım arkadan gelen ikinci bir işaretçi gerekir. Kodda bu işaretçinin adı `q`'dur.

```
boolean arama_agacina_ekle(int x, agac_gost *L) {
  agac_gost q = NULL, p = *L, r; // q arka, p ön
  while (p) {                    // NULL'a dek in
    q = p;                       // arkayı günle
    if (x < p->veri) p = p->solcocuk;
    else if (x > p->veri) p = p->sagcocuk;
    else { printf("Benzer deger var.\n");
           return FALSE; }
  }
  r = ogeal();                   // çocuklar NULL
  r->veri = x;
  if (*L == NULL) *L = r;        // boş ağaç: kök
  else if (x < q->veri) q->solcocuk = r;
  else q->sagcocuk = r;
  return TRUE;
}
```

Kodun satırları şöyle çalışır.

- **2. satır.** Üç yerel işaretçi tanımlanır. `p` ağacın kökünden yola çıkar; `*L`, çağıranın kök işaretçisinin değeridir. `q`, `p`'nin bir adım gerisinde kalır ve başta `NULL`'dır. `r` yeni düğümü tutacaktır.
- **3.–9. satırlar.** Döngü `p` `NULL` olana kadar sürer. Her turda önce `q = p` ile geri işaretçi güncellenir, sonra `p` bir düzey iner. `x` düğümdeki değerden küçükse sol çocuğa, büyükse sağ çocuğa geçilir. İkisi de değilse değerler eşittir: yordam bir uyarı yazar ve `FALSE` döndürür. Bu durumda ağaçta hiçbir şey değişmez ve bellek ayrılmaz.
- **10.–11. satırlar.** Döngü bittiğinde `p` `NULL`'dır, `q` ise yeni düğümün ebeveyni olacak düğümü gösterir. Yeni düğüm `ogeal()` ile ayrılır; çocukları `NULL`'dır. Veri alanına `x` yazılır.
- **12. satır.** Ağaç boşsa döngü hiç dönmemiştir ve `q` hâlâ `NULL`'dır. Yeni düğüm kök olur: `*L = r`. Bu atama çağıranın kök değişkenini değiştirir.
- **13.–14. satırlar.** Ağaç boş değilse `x` ebeveynin değeriyle bir kez daha karşılaştırılır ve yeni düğüm ebeveynin sol ya da sağ işaretçisine bağlanır.

Döngüde `p`'nin hangi yöne gittiği bilinir, ama döngü bitince bu bilgi kaybolur. 13. satırdaki ikinci karşılaştırma bu bilgiyi yeniden üretir. Bunun yerine döngüde bir yön değişkeni tutmak da mümkündür. Kod ise kısalığı seçmiştir. Yalnız bir karşılaştırma fazladan yapılır ve maliyet değişmez.

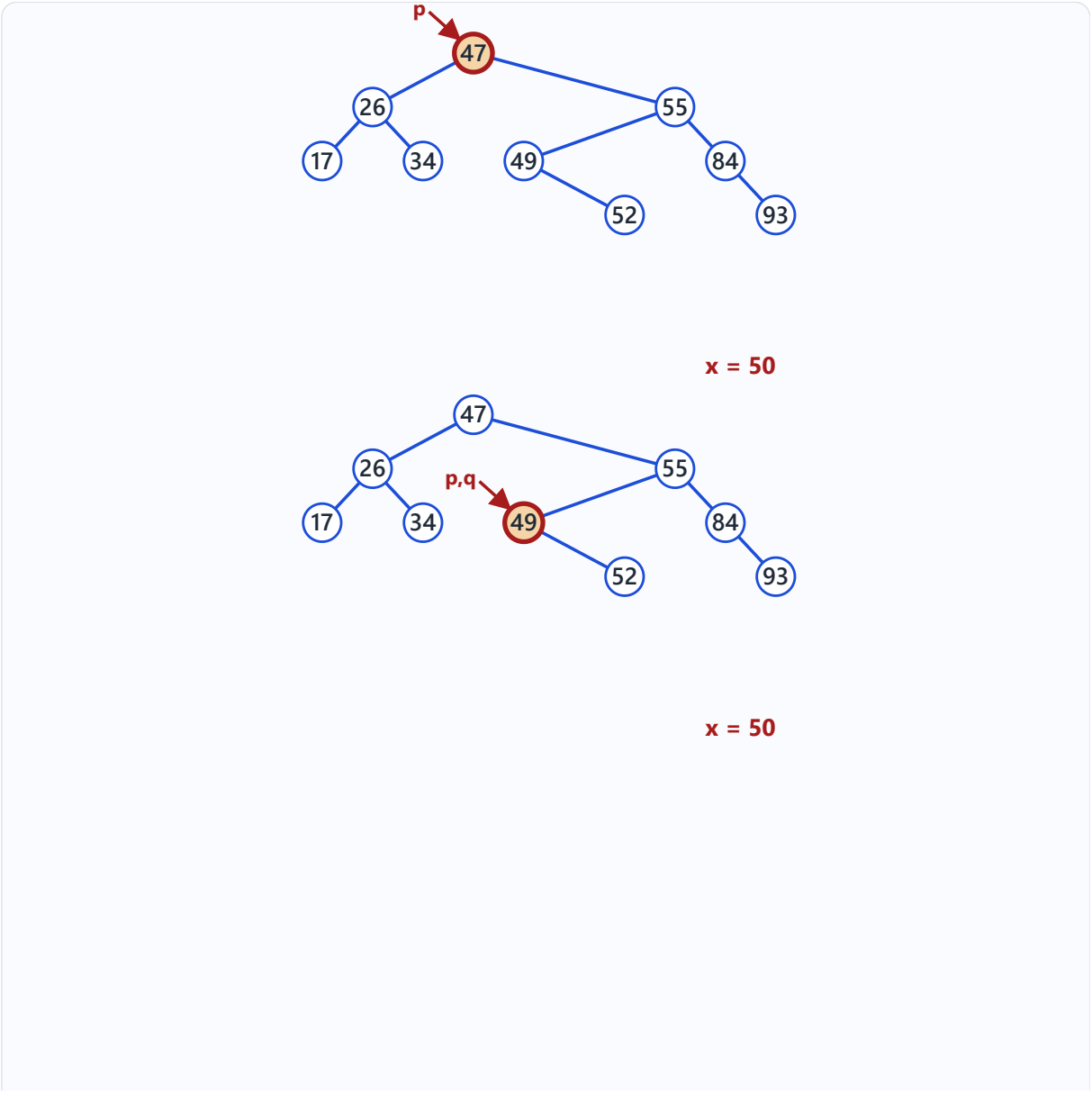
11.2.6 Kod izleme: 50'yi eklemek

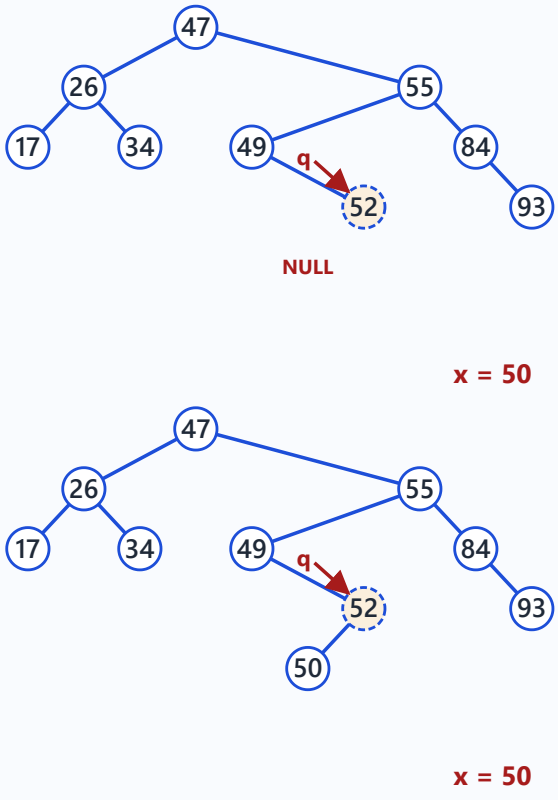
Sunum, soldaki ağaca (47, 55, 49, 26, 34, 17, 52, 84, 93) 50'nin eklenmesini izler. Tablo 11.2 her adımda değişkenlerin durumunu verir.

Adım	Satır	`q`	`p`	Açıklama
0	2	NULL	47	`p` kökte; `q` henüz bir düğüm göstermiyor.
1	4–6	47	55	*50 > 47*: yeri sağ alt ağaçta.
2	4–5	55	49	*50 < 55*: sol alt ağaca inilir.
3	4–6	49	52	*50 > 49*: sağa inilir.
4	4–5	52	NULL	*50 < 52*: 52'nin solu boştur; döngü biter.
5	10–11	52	NULL	Yeni düğüm `r` ayrılır; `r->veri = 50`.
6	13	52	NULL	Ağaç boş değil ve *50 < 52*: `q->solcocuk = r`.
7	15	—	—	`TRUE` döner.

Tablo 11.2: `arama_agacina_ekle(50, &L)` çağrısının izlemesi. Değerler düğümleri adlandırır; "düğüm(47)" yerine yalnız 47 yazıldı.

Tablodaki dört karşılaştırma, kökten 52'ye uzanan yoldaki dört düğüme karşılık gelir. Eklemeden önce ağacın derinliği 4'tür; 50, 47 → 55 → 49 → 52 yolunun altına beşinci düzeye girer ve derinlik 5 olur. Ekleme tek bir işaretçiyi değiştirdi: 52'nin sol çocuğunu. Ağacın geri kalanına dokunulmadı. Şekil 11.4 izlemenin dört karesini gösterir.





Şekil 11.4: `arama_agacina_ekle(50, &L)` izlemesinden dört kare. Soldan sağa ve yukarıdan aşağı: `p` kökte, `q` `NULL`; `p` 52'ye inmiş, `q` 49'da; `p` `NULL` olmuş, `q` 52'yi gösteriyor; yeni düğüm `r` 52'nin soluna bağlanmış.

Aynı çağrı 47 ile yapılsaydı döngünün ilk turunda eşitlik bulunurdu. Yordam "Benzer deger var." yazıp `FALSE` döndürürdü. Sürücü programda bu iki durum denendi: 50 için dönüş değeri 1, 47 için 0'dır. Eklemeden sonra ağacın kök önde gezinmesi 47 26 17 34 55 49 52 50 84 93 olur.

11.2.7 Neden `agac_gost *L`?

Ekleme yordamı ağacı `agac_gost L` olarak değil, `agac_gost *L` olarak alır. Yani kök işaretçisinin değerini değil, adresini ister. Bunun nedeni boş ağaçtır. Ağaç boşsa (`*L == NULL`) eklenen öğe kök olur ve çağıranın kök değişkeni bu yeni düğümü göstermelidir. C'de bütün parametreler değerle geçirilir. Yordam `agac_gost L` alsaydı, `L = r` ataması yalnız yordamın yerel kopyasını değiştirirdi. Çağıranın `kok` değişkeni `NULL` kalırdı, ayrılan düğüm de hiçbir yerden gösterilmediği için sızardı.

Bir değişkeni bir yordama değiştirtmek için onun adresi gönderilir. Bu yönteme **adresle çağrı** (*call by reference*) denir. Kök değişkeni zaten bir işaretçi olduğu için adresi bir **işaretçinin işaretçisidir** (*pointer to pointer*). Çağrı `arama_agacina_ekle(x, &kok)` biçiminde yazılır. Yordamın içinde `*L`, çağıranın `kok` değişkeninin kendisidir; `*L = r` o değişkene yazar.

Aynı sorunun ikinci bir çözümü daha vardır: yordam ağacın yeni kökünü dönüş değeri olarak verir. Sunum bu özyinelemeli sürümü de gösterir.

```
/* Özyinelemeli alternatif: yeni kökü döndürür */
agac_gost ekle(agac_gost L, int x) {
  if (!L) {
    L = ogeal();
    L->veri = x;
  } else if (x < L->veri)
    L->solcocuk = ekle(L->solcocuk, x);
  else if (x > L->veri)
    L->sagcocuk = ekle(L->sagcocuk, x);
  return L;        /* kok = ekle(kok, x); */
}
```

Yordam boş bir alt ağaca gelince yeni düğümü ayırır ve onu döndürür. Boş olmayan bir düğümde, değere göre sol ya da sağ alt ağaca ekleme yapar. Alt çağrının dönüşünü ilgili çocuk işaretçisine yeniden atar. Alt ağaç boş değilse alt çağrı aynı işaretçiyi geri verir ve atama hiçbir şeyi değiştirmez. Alt ağaç boşsa alt çağrı yeni düğümü verir ve atama onu ağaca bağlar. Değer eşitse iki dal da atlanır ve düğüm olduğu gibi döner. Kullanım `kok = ekle(kok, x);` biçimindedir. Çağıran dönüş değerini atamayı unutursa boş ağaca yapılan ilk ekleme kaybolur. Bu sürümde hata bildirimi de yoktur: eşit değer sessizce reddedilir. Sürücü programda dokuz değer iki yordamla ayrı ayrı eklendi; iki ağacın kök önde gezinmesi aynı çıktı.

Ekleme yordamının maliyeti, yeni değerin indiği yolun uzunluğudur. Her düzeyde bir düğüme bakılır ve sabit sayıda işlem yapılır. Ağacın derinliği h ise ekleme $O(h)$ sürer. Dengeli ağaçta $h = O(\lg n)$, çarpık ağaçta $h = n$'dir. Sunum bu aralığı $\Omega(\lg n) – O(n)$ diye yazar. Alt sınır, hiçbir ağacın derinliğinin $\lfloor \lg(n+1) \rfloor$'den küçük olamamasından gelir. Örneğin bütün düzeyleri dolu bir ağaçta boş işaretçilerin hepsi son düzeyin altındadır; her ekleme $\lg(n+1)$ düğüme bakar. Yinelemeli sürüm yalnız üç işaretçilik ek bellek kullanır, yani $O(1)$. Özyinelemeli sürüm ise her düzey için bir çağrı çerçevesi açar ve $O(h)$ ek bellek ister. Sıralı 2000 değerle kurulan çarpık ağaçta yinelemeli yordam sorunsuz çalıştı. Özyinelemeli yordam bu derinlikte 2000 iç içe çağrı yapar. Windows'un varsayılan 1 MB'lık çağrı yığını 30.000 düğümlük bir zinciri de kaldırdı. Ama çağrı derinliği n ile büyür; yeterince uzun bir zincirde yığın taşar.

Sunumda “C99 düzeltmesi” diye anılan bir ayrıntı daha vardır. Kitaptaki özgün kodda `q`'ya ilk değer verilmez. Ağaç boşken döngü hiç dönmez ve `q` tanımsız bir değerle kalır. Kod `*L == NULL` denetimiyle bu değeri kullanmaktan kaçınır. Yine de derleyiciler “ilk değer verilmeden kullanılabilir” uyarısı verir. Sunumdaki kod `q = NULL` ile başlar; böylece uyarı da kalkar, kod da güvenli olur.

11.3 Arama

Arama, ikili arama ağacının var oluş nedenidir. Ekleme ve silme de aramayla başlar: ekleme yeni değerin yerini, silme silinecek düğümü arar. Bu kesim aramayı önce bir döngüyle, sonra özyinelemeyle yazar. Ardından özyinelemeli aramadaki ince bir noktayı ve sıralı olmayan genel bir ikili ağaçta aramayı ele alır.

11.3.1 Yinelemeli arama

Aranan `x` değeri önce kökteki değerle karşılaştırılır. Eşitse arama biter. `x` daha küçükse aranan değer, varsa, sol alt ağaçtadır; tanım gereği sağ alt ağaçtaki her değer kökten, dolayısıyla `x`'ten büyüktür. `x` daha büyükse aynı akıl yürütme sağ alt ağacı gösterir. Seçilen alt ağaçta aynı işlem tekrarlanır. Bir `NULL` işaretçiye gelinirse değer ağaçta yoktur.

```
agac_gost arama_agacinda_ara(int x, agac_gost L) {
  while (L) {           // uygun alt ağaçta in
    if (x == L->veri) return L;
    if (x < L->veri) L = L->solcocuk;
    else             L = L->sagcocuk;
  }
  return NULL;          // bulunamadı
}
```

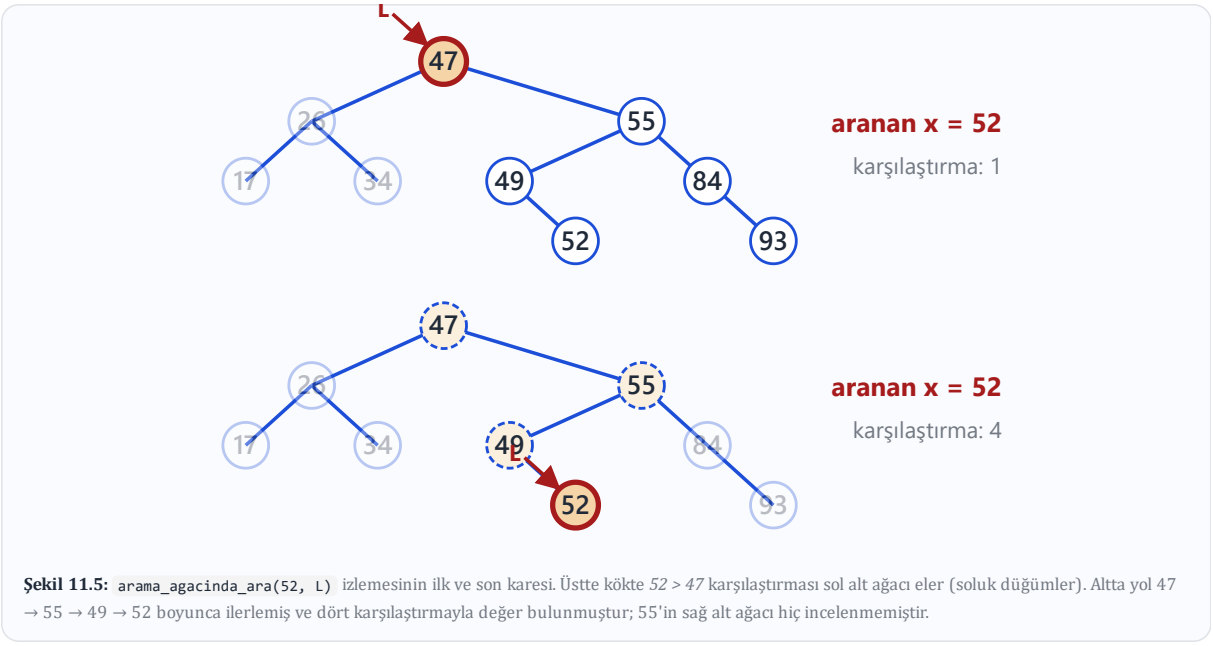
Yordam ağacı değerle aldığı `L` işaretçisini bir gezgin gibi kullanır. 2. satırdaki döngü `L` `NULL` olmadıkça sürer. 3. satır eşitliği denetler ve bulunan düğümün adresini döndürür. 4. ve 5. satırlar `L`'yi uygun çocuğa indirir. Döngüden çıkılırsa (7. satır) değer yoktur ve `NULL` döner. Burada `L`'nin değiştirilmesi çağıranı etkilemez; çağıranın kök işaretçisi yerinde kalır. Ekleme yordamının aksine aramaya adres göndermek gerekmez, çünkü arama ağacı değiştirmez.

Dönüş değeri olarak düğümün adresi seçilmiştir. Böylece çağıran hem değerin var olup olmadığını öğrenir, hem de düğümdeki öteki alanlara ulaşır. Anahtarla bir kayıt arandığında asıl istenen çoğu zaman bu öteki alanlardır.

Örnek 11.3

Soru. 50 eklenmeden önceki ağaçta (47, 55, 49, 26, 34, 17, 52, 84, 93) `arama_agacinda_ara(52, L)` hangi düğümlere bakar?

Çözüm. Kökte $52 > 47$'dir ve `L` sağa, 55'e iner. Sol alt ağacın tamamı (26, 17, 34) elenir. 55'te $52 < 55$'tir ve `L` sola, 49'a iner. 84 ile 93 hiç incelenmez. 49'da $52 > 49$'dur ve `L` sağa, 52'ye iner. 52'de eşitlik bulunur ve düğümün adresi döner. Dört düğüme bakılmıştır. C kodunda her düğümde önce eşitlik, sonra küçüklük denetlenir. İlk üç düğümde iki, sonuncusunda bir denetim yapılır; toplam yedi karşılaştırma işlemi eder. Analizde her düğüm ziyareti tek bir üç yollu karşılaştırma sayılır: küçük, eşit ya da büyük.



Şekil 11.5: `arama_agacinda_ara(52, L)` izlemesinin ilk ve son karesi. Üstte kökte *52 > 47* karşılaştırması sol alt ağacı eler (soluk düğümler). Altta yol 47 → 55 → 49 → 52 boyunca ilerlemiş ve dört karşılaştırmayla değer bulunmuştur; 55'in sağ alt ağacı hiç incelenmemiştir.

Şekil 11.5 aramanın gücünü görsel olarak gösterir: her karşılaştırma bir alt ağacın tamamını eler. Ağaç dengeliyse her eleme kalan düğümlerin yaklaşık yarısını atar. Bu, sıralı dizide ikili aramanın davranışıdır. Ağaçta olmayan bir değer arandığında da aynı yol izlenir. Örneğin 60 aranırsa 47 → 55 → 84 yoluna bakılır. 84'ün sol çocuğu `NULL` olduğu için arama üç düğümden sonra `NULL` döndürür.

Aramanın maliyeti, bakılan yolun uzunluğudur ve ağacın derinliği *h* ile sınırlıdır: *O(h)*. Dengeli ağaçta bu *O(lg n)*, çarpık ağaçta *O(n)* eder. Yinelemeli yordam ek bellek olarak yalnız `L`'yi kullanır: *O(1)*.

Sunumda bu yordamın yanında hocanın bir benzetmesi yer alır. Müdür memura görevi tanımlarken çok açık konuşmalıdır; memur sonradan kapıyı çalıp soru sormak zorunda kalmamalıdır. Yordamın tanımı da böyledir. `arama_agacinda_ara`'nın sözleşmesi açıktır: değer varsa düğümün adresini, yoksa `NULL` döndürür. Boş ağaçta da doğru çalışır, ağacı değiştirmez ve küresel bir değişkene dokunmaz. Yordamı kullanan, onun içine bakmadan doğru kullanabilir.

11.3.2 Özyinelemeli arama

Tanım özyinelemeli olduğu için arama da doğal olarak özyinelemeli yazılır. Boş ağaçta değer yoktur. Kökte değer bulunursa iş biter. Yoksa aynı soru uygun alt ağaca sorulur ve onun cevabı aynen yukarı iletilir.

```
agac_gost ara(int x, agac_gost L) {
  if (!L) return NULL;          // yok
  if (x == L->veri) return L;   // bulundu
  if (x < L->veri)
    return ara(x, L->solcocuk);
  return ara(x, L->sagcocuk);
}
```

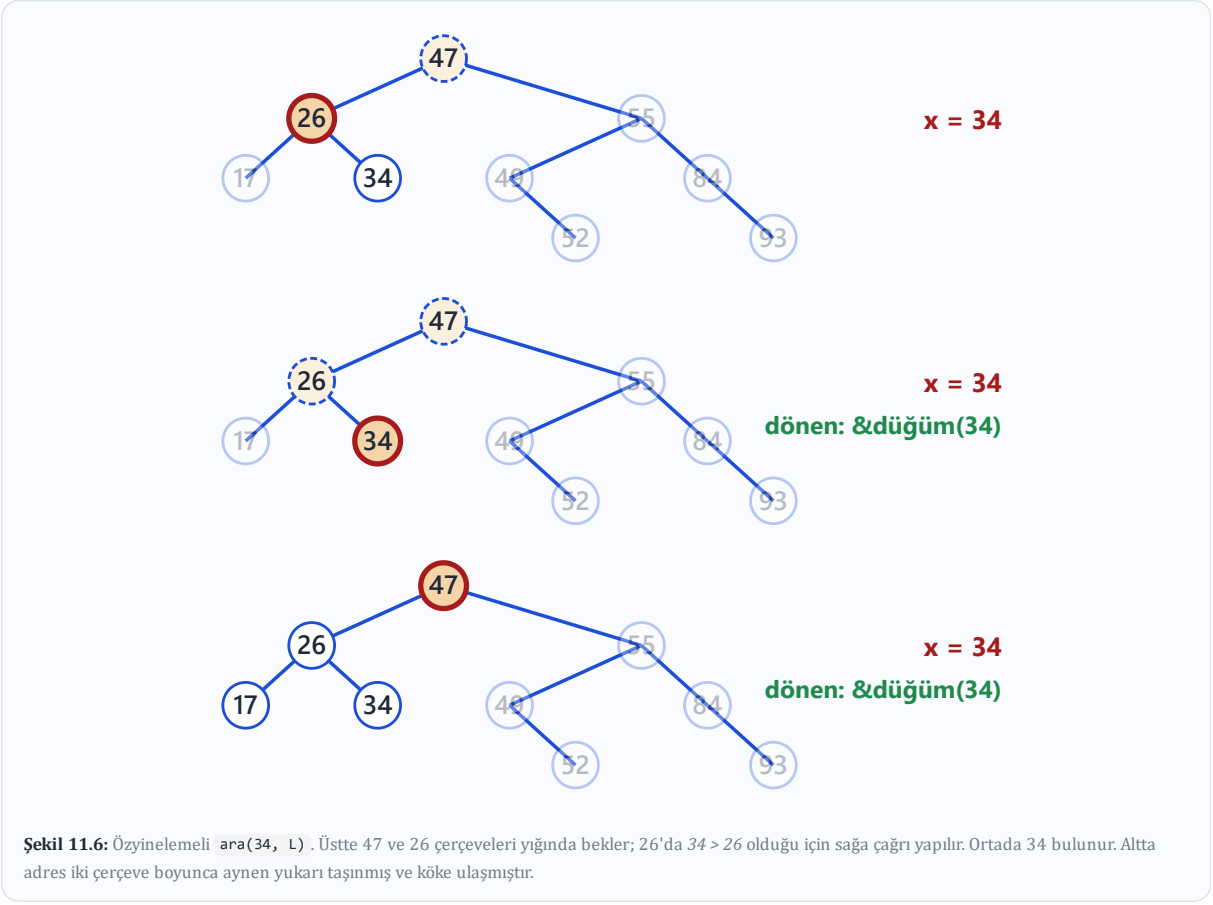
İlk iki satır temel durumlardır. Ağaç boşsa (2. satır) `NULL`, kökte değer varsa (3. satır) kökün adresi döner. 4.–6. satırlar özyinelemeli adımdır. Her iki çağrının önünde `return` bulunur. Alt çağrı ne döndürürse bu çerçeve de onu döndürür ve başka bir iş yapmaz. Bir yordamın son işi kendi kendini çağırmaksa bu çağrıya **kuyruk özyinelemesi** (*tail recursion*) denir. Böyle bir yordam kolayca döngüye çevrilir: parametre yeni değeriyle güncellenir ve başa dönülür. Yinelemeli arama, bu yordamın tam olarak bu dönüşümle elde edilmiş hâlidir. İyi eniyileme yapan derleyiciler bu dönüşümü kendileri de yapabilir.

Çerçeve	`L`	Karşılaştırma	Yaptığı	Döndürdüğü
1	47	*34 < 47*	`return ara(x, L->solcocuk)`	&düğüm(34)
2	26	*34 > 26*	`return ara(x, L->sagcocuk)`	&düğüm(34)

Çerçeve	`L`	Karşılaştırma	Yaptığı	Döndürdüğü
3	34	*34 = 34*	`return L`	&düğüm(34)

Tablo 11.3: `ara(34, L)` çağrısının çerçeve izlemesi. Her satır bir çağrı çerçevesidir; son sütun çerçevenin döndürdüğü değerdir.

Tablo 11.3 çağrıların inişini ve dönüşünü özetler. Üç çerçeve iner, en alttaki çerçeve düğümü bulur ve adresini döndürür. Dönüş yolunda çerçeveler hiçbir hesap yapmaz; aldıkları adresi aynen bir üste verirler. Şekil 11.6 üç anı gösterir: inişin ortasını, bulma anını ve sonucun köke ulaştığı anı.



Şekil 11.6: Özyinelemeli `ara(34, L)`. Üstte 47 ve 26 çerçeveleri yığında bekler; 26'da *34 > 26* olduğu için sağa çağrı yapılır. Ortada 34 bulunur. Altta adres iki çerçeve boyunca aynen yukarı taşınmış ve köke ulaşmıştır.

Zaman maliyeti yinelemeli sürümle aynıdır: *O(h)*. Bellek maliyeti farklıdır. Her çağrı bir çerçeve açar ve en derin noktada *h* çerçeve aynı anda yığında durur. Ek bellek *O(h)*'dir. Dengeli ağaçta bu birkaç düzine çerçevedir ve önemsizdir. Çarpık ağaçta ise çağrı derinliği *n*'ye ulaşır.

11.3.3 Özyinelemeli aramada bir tuzak

Sunum bir öğrencinin yazdığı aşağıdaki gövdeyi gösterir ve "Bu örnek çalışır mı?" diye sorar.

```
if (L) {
  if (x == L->veri) return L;
  if (x <  L->veri) return arama(x, L->solcocuk);
  return arama(x, L->sagcocuk);
}
return NULL;   /* L == NULL: boş (alt) ağaç */
```

Kodun sorduğu asıl soru 3. satırdadır. `x` kökten küçükse 3. satırdaki `if` bloğuna girilir ve sol alt ağaçta arama yapılır. Alt çağrı ne döndürürse yordam onu döndürür. 4. satırdaki sağ alt ağaç araması bu durumda **asla** yapılmaz. Sol alt ağaçta değer bulunamazsa `NULL` döner ve yordam sağa hiç bakmadan biter.

İkili arama ağacında bu davranış tam olarak istenendir. `x` kökten küçükse sağ alt ağaçta olamaz; oraya bakmak boşa iştir. Kod doğrudur ve *O(h)*'dir. Sıralı olmayan genel bir ikili ağaçta ise durum farklıdır. `x`, kökten küçük olduğu hâlde sağ alt ağaçta bulunabilir; bu kod onu kaçırır. Sürücü programda kökü 5, sol çocuğu 9, sağ çocuğu 2 olan bir ağaçta 2 arandı. *2 < 5* olduğu için arama sola gitti, 9'un solunda `NULL`'a ulaştı ve "yok" cevabı verdi.

Son satırdaki `return NULL` da dikkat ister. Sunumun özgün sürümünde bu satırın yanında "Buraya hiç girmez, ama olsun, biz yazalım da..." yorumu vardı. Yorum yanlıştı ve sunumda düzeltildi. `L` `NULL` olduğunda `if` bloğu atlanır ve yordam tam bu satıra gelir. Bu da her başarısız aramanın son çağrısında olur. Satır silinirse yordam boş ağaçta bir değer döndürmeden biter. Bir değer döndürmesi gereken yordamın sonuna ulaşıp o değeri kullanmak C'de tanımsız davranıştır. Derleyici bu durumda uyarı verir; çağıran ise rastgele bir adres alabilir.

Hocanın bu örnekle ilgili bir notu daha vardır: özyinelemeli kod yazılırken bazen örnekteki gibi `return` satırından sonra da işlem yapılır. Bu, özyinelemenin iki ayrı biçimini hatırlatır. Aramada alt çağrının sonucu olduğu gibi iletilir ve çağrıdan sonra iş kalmaz. Derinlik ya da düğüm sayısı hesaplanırken ise alt çağrıların sonuçları birleştirilir ve dönüşte hesap yapılır. Hangi biçimin gerektiği, problemin her iki alt ağaca da ihtiyaç duyup duymadığına bağlıdır.

> **Sık yapılan hata**
>
> Özyinelemeli aramada alt çağrının önüne `return` yazmamak: `if (x < L->veri) ara(x, L->solcocuk);`. Alt çağrı düğümü bulur ve adresini döndürür, ama çağıran çerçeve bu değeri kullanmadan atar. Yordam sonra bir sonraki satıra geçer ve yanlış bir değer döndürür. Kod çoğu zaman derlenir; derleyici yalnız "her yol bir değer döndürmüyor" uyarısı verir. Uyarılar açık derlenmeli ve ciddiye alınmalıdır.

11.3.4 Her ikili ağaçta arama

Değerleri sıralı olmayan bir ikili ağaçta arama, hangi yöne gidileceğini bilemez. İki alt ağaca da bakmak gerekir. Sunumdaki örnek çözüm önce sol alt ağaca bakar; orada bulamazsa sağa geçer.

```
agac_gost ara_ozyineli(int veri, agac_gost L) {
  if (!L) return NULL;              /* ağaç boş mu   */
  if (L->veri == veri) return L;    /* aranan bulundu */
  agac_gost T = ara_ozyineli(veri, L->solcocuk);
  return T ? T : ara_ozyineli(veri, L->sagcocuk);
}
```

Yordamın ilk iki satırı özyinelemeli aramayla aynıdır. 4. satır sol alt ağaçta arar ve sonucu `T`'de saklar. 5. satırdaki koşullu ifade, `T` `NULL` değilse onu döndürür. Değer solda bulunduysa sağa hiç bakılmaz. Solda bulunamadıysa sağ alt ağacın aramasının sonucu döner. Sağda da yoksa bu sonuç `NULL`'dır.

Yordam ağacın sıralama bilgisini kullanmaz. Değer ağaçta yoksa ya da en son ziyaret edilen düğümdeyse bütün düğümlere bakılır; en kötü durum *O(n)*'dir. Bu yordam bir ikili arama ağacında da doğru çalışır, ama gereksiz yere yavaştır. Sürücü programda kökü 5, sol çocuğu 9, 9'un sağ çocuğu 1, kökün sağ çocuğu 2 olan genel ağaçta 2, 1 ve 7 arandı. Yordam 2'yi ve 1'i buldu, 7 için `NULL` döndürdü. Aynı ağaçta ikili arama ağacı yordamları 2'yi bulamadı.

Bu yordamın bellek maliyeti çağrı derinliğidir: *O(h)*. Yordam, ağırlık bulma sınav sorusunun çözümünde de kullanılır. Hoca bu yordamlar için iddialı bir şey söyler: bunları sınıftaki herkesin gözü kapalı yazabilmesi gerekir. Sonra yumuşatır: "Hadi bir gözü açık olsun." Söylenmek istenen şudur: arama, ekleme ve derinlik yordamları ağaç konusunun alfabesidir. Daha büyük her problem bunların birleşimiyle çözülür.

11.4 Derinlik

Ağacın derinliği bütün işlemlerin maliyetini belirler. Onu hesaplayan yordam küçük olsa da öğretici bir örnektir. Problem her iki alt ağacın cevabına ihtiyaç duyar ve özyinelemeli düşünmenin en temiz örneklerinden birini verir. Bu kesim önce yinelemeli bir çözümü, sonra özyinelemeli çözümü ve onun iki yanlış kullanımını ele alır.

11.4.1 Yinelemeli derinlik ve küresel değişken

Sunum bu konuyu bir soruyla açar: derinlik yinelemeli olarak yazılsın. Hoca sınıfta görülen tipik çözümlere sert bir tepki verir. O çözümler ağacı bir yığınla gezer ve o ana kadar görülen en büyük düzeyi bir **küresel değişkende** (*global variable*) tutar. Hocaya göre bu kod çalışsa bile puan almaz. Bir öğrenci "Doğru çalışıyorsa bir puan da mı vermezsiniz?" diye sorar. Cevap kesindir: çeyrek puan bile verilmez. Neden bu kadar sert olunduğu birazdan anlatılacak. Önce küresel değişken kullanmayan yinelemeli bir çözüm görülsün.

```
/* global değişkensiz yinelemeli çözüm:
   düzey düzey gezin, düzeyleri say */
int derinlik_d(agac_gost L) {
  agac_gost k[MAX_SIZE];
  int on = 0, arka = 0, d = 0;
  if (L) k[arka++] = L;
  while (on < arka) {          /* bir düzey */
    int son = arka;
    d++;
    for (; on < son; on++) {
      if (k[on]->solcocuk) k[arka++] = k[on]->solcocuk;
      if (k[on]->sagcocuk) k[arka++] = k[on]->sagcocuk;
    }
  }
  return d;       /* MAX_SIZE >= n olmalı */
}
```

Çözüm ağacı **düzey sırasıyla** (*level-order*) gezer. `k` dizisi bir kuyruk gibi kullanılır: `on` sıradaki öğenin, `arka` ilk boş hücrenin indisidir. Kök kuyruğa konur. Dış döngünün her turu bir düzeyi işler. Tur başında `son = arka` ile o düzeyin sonu işaretlenir ve `d` bir artırılır. İç döngü, kuyruktaki o düzeyin düğümlerini sırayla alır ve çocuklarını kuyruğun arkasına ekler. Bu çocuklar bir sonraki düzeydir. Kuyruk boşaldığında bütün düzeyler sayılmıştır ve `d` derinliği verir. Boş ağaçta kuyruğa hiçbir şey konmaz ve `d = 0` döner.

Kuyruktan alınan öğe silinmez; `on` yalnız ilerler. Bu yüzden dizinin, ağacın bütün düğümlerini alacak kadar büyük olması gerekir: `MAX_SIZE ≥ n`. Kodun son satırındaki yorum bu koşulu hatırlatır. Her düğüm kuyruğa bir kez girip bir kez çıktığı için süre *O(n)*'dir. Bellek ise dizinin boyu kadardır ve yığın bölgesinde (*stack*) durur. Sürücü programda `MAX_SIZE` 60.000 seçildi; 8 baytlık işaretçilerle bu, yaklaşık 470 KB eder. Windows'ta varsayılan çağrı yığını 1 MB olduğundan birkaç yüz bin düğümlük bir dizi yerel olarak tanımlanamaz. Gerçek bir programda bu dizi `malloc` ile ayrılmalıdır. Yordam, sunumdaki örnek ağaçlarda ve 50.000 rastgele değerle kurulmuş ağaçta özyinelemeli sürümle aynı sonucu verdi.

Hocanın küresel değişkene tepkisinin nedeni yazılımın ölçeğidir. Büyük firmalarda büyük işler birlikte yapılır. Birinin yazdığı yordamı, öbür uçtaki biri kullanır. Hoca bunu bir benzetmeyle anlatır: kullanan kişi yordamı yazanı tanımaz, belki o başka bir ülkededir. Telefonu açıp "Senin bir `max` değişkenin var, bu ne oluyor?" diye soramaz. Küresel değişken kullanan bir yordamın üç sorunu vardır.

- **Gizli durum.** Yordamın sonucu yalnız parametrelerine değil, küresel değişkenin o anki değerine de bağlıdır. Değişken bir önceki çağrıdan kalma bir değer taşıyorsa ikinci çağrı yanlış sonuç verir. Kullanan, her çağrıdan önce değişkeni sıfırlamak gerektiğini bilmek zorunda kalır.
- **Ad çakışması.** İki ayrı dosyada aynı adla küresel değişken tanımlanırsa bağlayıcı hata verir ya da, daha kötüsü, iki değişkeni sessizce birleştirir.
- **Yeniden girilemezlik.** Yordam aynı anda iki ağaç için çalıştırılamaz. İki iş parçacığı aynı küresel değişkeni bozar; yordam kendini başka bir ağaç için özyinelemeli olarak da çağıramaz.

Sunum bu tartışmadan üç ilke çıkarır. Birincisi, yordam kendi içinde tam olmalıdır: girdisini parametreden alır, çıktısını dönüş değeriyle verir ve gizli bir durum tutmaz. İkincisi, bir yordamın içinde başka bir yordamın **yerel prototipi** bildirilebilir. Bu yalnız adı o yordamın içinde görünür kılar; tanım dışarıda kalır. Standart C iç içe yordam tanımına izin vermez. Bu teknik ağırlık bulma sınav sorusunun çözümünde kullanılır. Üçüncüsü, özyinelemeli çözümde her çağrının kendi yerel sol ve sağ değişkeni vardır; küresel bir sayaca hiç gerek kalmaz.

11.4.2 Özyinelemeli derinlik

Derinliğin özyinelemeli tanımı tek cümledir. Boş ağacın derinliği 0'dır. Boş olmayan bir ağacın derinliği, sol ve sağ alt ağaçlarından derin olanının derinliğinden 1 fazladır: *derinlik(L) = MAX(derinlik(sol), derinlik(sağ)) + 1*. Eklenen 1 kökün kendisidir. Kod bu tanımı birebir izler.

```
int derinlik(agac_gost L) {
  int sol, sag;
  if (L) {                  // işaretçi NULL mı
    sol = derinlik(L->solcocuk);
    sag = derinlik(L->sagcocuk);
    return MAX(sol, sag) + 1;
  }
  else return 0;            // uç yaprağa ulaşıldı
}
```

2. satırdaki iki yerel değişken, alt ağaçların derinliklerini saklar. 3. satır problemin yeterince küçülüp küçülmediğini sorar. Hoca bu soruyu özellikle vurgular: “Problem yeterince küçüldü mü? Başta da `NULL` gelebilir!” `NULL` denetimi yalnız yaprakların altındaki boş çocuklar için değildir; ilk çağrıda ağacın kendisi de boş olabilir. 4. ve 5. satırlar iki alt ağacın derinliğini hesaplar. 6. satır büyüğünü alır ve 1 ekler. 8. satır boş ağaç için 0 döndürür.

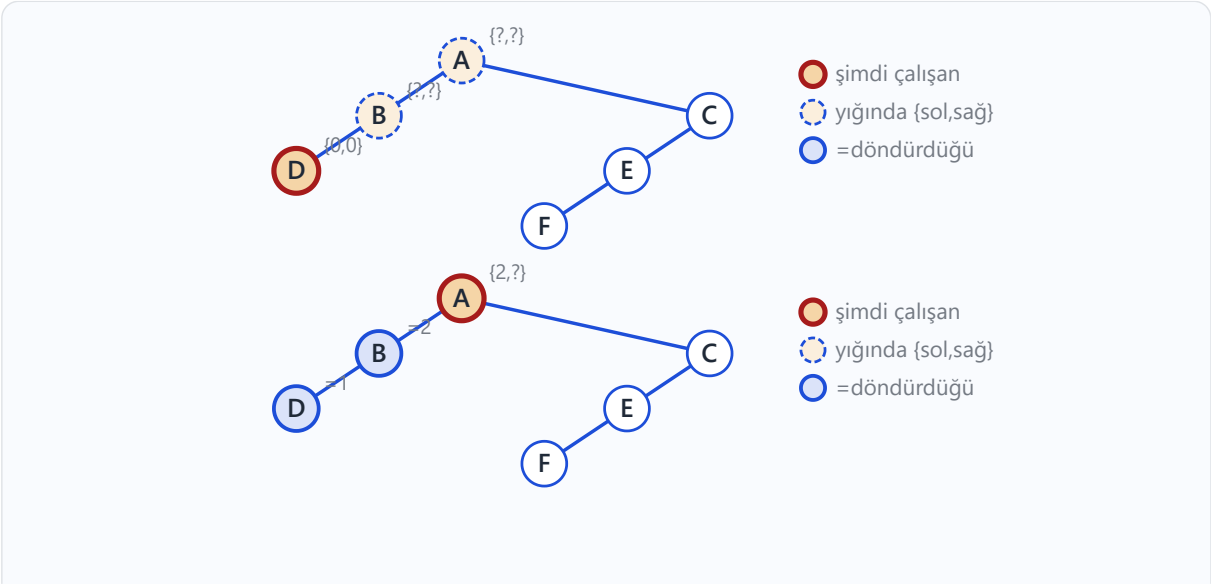
Sunumdaki `MAX` bir makrodur: `#define MAX(a,b) ((a) > (b) ? (a) : (b))`. Buradaki argümanlar yalnız birer yerel değişkendir; makronun bir argümanı iki kez değerlendirmesi zarar vermez. Aynı makronun tek satırlık sürümde neden tehlikeli olduğu bir sonraki kesimin konusudur.

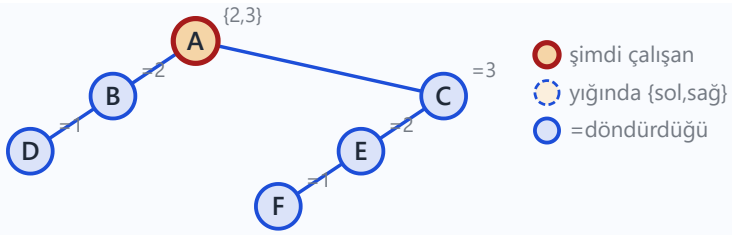
Sunum yordamı altı düğümlü bir ağaçta izler. A köktür; solunda B, sağında C vardır. B'nin solunda D, C'nin solunda E, E'nin solunda F bulunur. Tablo 11.4 her çerçevenin `{sol, sağ}` değerlerini ve dönüşünü verir.

Bitiş sırası	`L`	`sol`	`sag`	Döndürdüğü
1	D	0	0	1
2	B	1	0	2
3	F	0	0	1
4	E	1	0	2
5	C	2	0	3
6	A	2	3	4

Tablo 11.4: `derinlik(A)` çağrısının izlemesi. Satırlar çerçevelerin bitiş sırasıyla verildi; `NULL` ile yapılan çağrılar 0 döndürür ve tabloda ayrıca gösterilmedi.

Çağrılar önce sol kolu izler. `derinlik(A)`, `derinlik(B)`'yi; o da `derinlik(D)`'yi çağırır. D'nin iki çocuğu da `NULL`'dır; iki alt çağrı 0 döndürür ve D 1 döndürür. B'nin sağ çocuğu yoktur; B *MAX(1, 0) + 1 = 2* döndürür. A'nın `sol` değişkeni 2 olur ve şimdi sağ alt ağaç için `derinlik(C)` çağrılır. C'den E'ye, E'den F'ye inilir. F 1, E 2, C 3 döndürür. Son olarak A, *MAX(2, 3) + 1 = 4* döndürür. Sürücü program aynı ağaçta 4 sonucunu verdi.





Şekil 11.7: Özyinelemeli derinlik izlemesinden üç kare. Üstte D çalışıyor; A ve B çerçeveleri yığında `{?, ?}` değerleriyle bekliyor. Ortada B 2 döndürmüş, A'nın `sol` değişkeni 2 olmuş ve sıra C'ye gelmiş. Altta bütün çerçeveler bitmiş, A 4 döndürüyor.

Şekil 11.7 çerçevelerin bir özelliğini gösterir. Her düğümün kendi iki yerel değişkeni vardır: sol ve sağ derinlik. Bir çerçeve bitince yerel değişkenleri de yok olur. `NULL` ile yapılan çağrılar yerel değişkenlerini hiç kullanmadan siler. Yığında aynı anda yalnız kökten o anki düğüme giden yoldaki çerçeveler bulunur; en çok $h + 1$ çerçeve.

Maliyet analizi şöyledir. Her düğüm için bir çağrı yapılır. Ayrıca her `NULL` çocuk için de bir çağrı yapılır. n düğümlü bir ikili ağaçta $n + 1$ boş çocuk işaretçisi bulunur: toplam $2n$ işaretçinin $n - 1$'i bir düğümü gösterir, kalanı `NULL`'dır. Çağrı sayısı tam olarak $2n + 1$'dir. Sürücü programda bu sayı 1'den 24'e kadar her zincir uzunluğunda ve 262.143 düğüme kadar mükemmel ağaçlarda sayaçla doğrulandı. Her çağrı sabit iş yaptığı için zaman $\Theta(n)$'dir. Derinliği bulmak için her düğüme bakmak zorunludur; hiçbir yordam bundan hızlı olamaz. Bellek, çağrı derinliği kadardır: $O(h)$.

11.4.3 Tek satırlık sürüm ve makro tuzağı

Aynı yordam yerel değişkenler kullanılmadan tek bir ifadeyle de yazılabilir. Sunum bu sürümü "profesyonelce" bir yazım olarak sunar, ardından içindeki tuzağı gösterir.

```
int derinlik(agac_gost L) {    /* tek satır */
  return !L ? 0 : MAX(derinlik(L->solcocuk),
                      derinlik(L->sagcocuk)) + 1;
}
```

Kod okunaklıdır ve doğru sonuç verir. Ama `MAX` bir fonksiyon değil, bir **makrodur** (*macro*). Önişlemci derlemeden önce `MAX(a, b)`'yi metin olarak `((a) > (b) ? (a) : (b))` ile değiştirir. Burada `a` ve `b` birer yordam çağrısıdır. Açılmış ifade önce iki çağrıyı da karşılaştırma için işletir. Sonra büyük olanı döndürmek için o çağrıyı *bir kez daha* işletir. Her düğümde alt ağaçlardan biri iki kez hesaplanır.

Çarpık bir ağaçta bunun bedeli üsteldir. Sağa uzanan bir zincirde her düğümün sol çocuğu `NULL`'dır. Sağ alt ağacın derinliği daha büyük olduğu için sağ çağrı iki kez yapılır. n düğümlü zincirde çağrı sayısı $T(n)$ ise $T(n) = 2T(n - 1) + 2$ olur. Buradaki 2, düğümün kendi çağrısı ile `NULL` sol çocuğun çağrısıdır. Çözüm $T(n) = 3 \cdot 2^n - 2$'dir. Sola uzanan zincirde de durum aynıdır. Orada sağ çocuk `NULL`'dır, sol çağrı büyük çıkar ve iki kez yapılır. Dolu ve dengeli bir ağaçta iki alt ağacın derinliği eşittir. Karşılaştırma yanlış çıkar ve sağ alt ağaç yeniden hesaplanır. Bu kez $T(n) \approx 3T(n/2)$ olur ve çözüm $\Theta(n^{\lg 3}) \approx \Theta(n^{1{,}585})$'tir. Yani dengeli ağaçta bile süre doğrusal değildir.

Ağaç	n	Yerel değişkenli	Tek satırlık (makro)
zincir	4	9	46
zincir	12	25	12.286
zincir	20	41	3.145.726
zincir	24	49	50.331.646
dolu	1.023	2.047	88.573
dolu	65.535	131.071	64.570.081

Tablo 11.5: İki derinlik yordamının çağrı sayıları (sayaçla ölçüldü). Yerel değişkenli sürüm her ağaçta $2n + 1$ çağrı yapar.

Tablo 11.5 farkı sayılarla gösterir. 24 düğümlük bir zincirde iki yordam aynı sonucu (24) verir. Biri 49, öteki elli milyonu aşkın çağrı yapar. Her yeni düğüm makrolu sürümün çağrı sayısını ikiye katlar. 40 düğümlük bir zincirde sayı $3 \cdot 2^{40} - 2$'dir, yani üç trilyonu aşar. Ölçülen bütün değerler $3 \cdot 2^n - 2$ formülüne tam uyar. Mükemmel ağaçlarda makrolu sürüm $(3^{k+1} - 1)/2$ çağrı yapar; burada k düzey sayısıdır.

Çözüm, makro yerine gerçek bir fonksiyon kullanmaktır. Fonksiyon argümanlarını çağrıdan önce bir kez değerlendirir ve değerleri kopyalar. Ağırlık bulma sınav sorusunun çözümü bu yolu izler:

```
int max(int a, int b) { return a > b ? a : b; }
```

Bu fonksiyonla yazılan tek satırlık derinlik yordamı, sürücü programda her ağaçta tam *2n + 1* çağrı yaptı. İkinci çözüm yerel değişkenli sürümdür. Değerler önce değişkenlere alınır; makro artık yalnız değişkenleri karşılaştırır.

> **Sık yapılan hata**
>
> Yan etkisi ya da maliyeti olan bir ifadeyi makroya argüman olarak vermek: `MAX(derinlik(...), derinlik(...))`, `MAX(i++, j)`, `SWAP(a[i++], b, t)`. Makro argümanını metin olarak kopyaladığı için ifade birden çok kez işletilir. Özyinelemeli çağrılarda sonuç üstel süredir, artırma işleçlerinde ise yanlış değerdir. Makroya yalnız basit değişkenler verilmeli, aksi durumda fonksiyon kullanılmalıdır.

Sunumda bu kesimin yanında hocanın bir gözlemi daha yer alır. Hocaya göre özyinelemeli düşünmeyi ve yineleme bağıntılarını çözmeyi mühendislere kabul ettirmek zordur. Döngüyle düşünmeye alışmış biri, “Ne güzel kod yazıyorum” der ve yeni bir düşünce tarzına direnir. Derinlik örneği bu direncin neden yersiz olduğunu gösterir. Yinelemeli çözüm bir kuyruk, bir boyut sınırı ve iki iç içe döngü ister. Özyinelemeli çözüm ise tanımın kendisidir ve dört satıra sığar.

11.4.4 Ek parametre koymak

Sunum bu kesimi A. J. Perlis'in bilinen bir sözüyle açar. Söz kabaca şöyle der: on parametreli bir yordamınız varsa, büyük olasılıkla birkaçını unutmuşsunuzdur. Şaka, parametre listesinin şişmesinin bir tasarım hatası belirtisi olduğunu anlatır.

Bir öğrenci şunu sorar: `int derinlik(agac_gost L, int a)` biçiminde fazladan bir parametre eklense ve yordam `derinlik(L, 0)` diye çağrılsa olur mu? Öğrencinin aklındaki çözüm muhtemelen şudur. `a` o ana kadar inilen düzeyi taşır, her alt çağrıya `a + 1` geçirilir ve `NULL`'a gelindiğinde `a` döndürülür. Kod çalışabilir. Ama hoca bu tasarımı kabul etmez. Fazladan parametre neden tanımlanıyor ve ilk çağrıda neden sıfır geçiriliyor? Yordamı kullanana ne denecek: “`a`'ya sıfır geçirmeyi sakın ha unutma!” mı? Hoca bunu bir örnekle somutlaştırır: 0 geçirilince sıralayan, 1 geçirilince sıralayamayan bir sıralama yordamı olur mu?

Sorun, parametrenin bir *girdi* değil, yordamın iç işleyişine ait bir başlangıç değeri olmasıdır. Kullanıcıya bir yükümlülük yükler ve bu yükümlülük unutulduğunda yordam sessizce yanlış sonuç verir. Gerekirse böyle bir ek parametre, dışarıya açılmayan bir yardımcı yordamda tutulur. Kullanıcıya açılan yordam yalnız `L`'yi alır ve yardımcıyı doğru başlangıç değeriyle kendisi çağırır.

Her ek parametre de kötü değildir. Sıralama yordamına küçükten büyüğe mi büyükten küçüğe mi sıralanacağını bildiren bir parametre (−1/+1 ya da `'<'` / `'>'`) verilebilir. Verinin türünü bildiren bir parametre de verilebilir. Bu parametreler yordamın *davranışını seçer*. Her değerleri anlamlıdır ve kullanıcı bilerek seçim yapar. İlke şudur: parametre davranışı seçmeli, gizli bir başlangıç değeri taşımamalıdır.

11.5 Silme

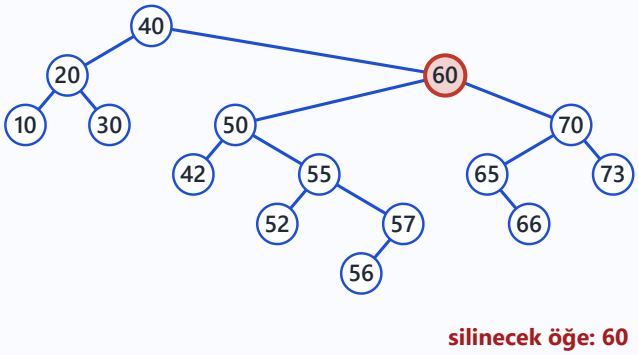
Silme, ikili arama ağacının en ince işlemidir. Ekleme her zaman bir `NULL` işaretçinin yerine yapılır ve yalnız o işaretçiyi değiştirir. Silinecek düğüm ise ağacın herhangi bir yerinde, örneğin kökte olabilir. Düğüm çıkarıldıktan sonra kalan düğümler yeniden bağlanmalı ve ağaç yine tanıma uymalıdır.

11.5.1 Üç durum

Silinecek düğümün çocuk sayısına göre üç durum vardır.

- **Çocuksuz düğüm (yaprak).** Ebeveynin bu düğümü gösteren işaretçisi `NULL` yapılır ve düğüm `free` edilir. Ağacın geri kalanı etkilenmez.
- **Tek çocuklu düğüm.** Ebeveynin işaretçisi, silinen düğümün tek çocuğunu gösterecek biçimde değiştirilir. Çocuğun alt ağacı bütünüyle bir düzey yukarı çıkar. Bu, sıralamayı bozmaz. Silinen düğüm ebeveyninin solundaysa, onun alt ağacındaki her değer ebeveynden küçüktür. Alt ağaç yukarı çıktığında yine ebeveynin solunda kalır.
- **İki çocuklu düğüm.** Ebeveynin tek işaretçisi iki alt ağacı birden gösteremez. Silinen düğümün yerine, iki alt ağacın arasına sığan bir değer konmalıdır. Bunun için farklı yöntemler vardır.

Sunum iki çocuklu durumu bir örnek ağaç üzerinde inceler. Ağaca 40, 20, 60, 10, 30, 50, 70, 42, 55, 65, 73, 52, 57, 66, 56 değerleri bu sırayla eklenmiştir. Ağacın derinliği 6'dır; en uzun yol 40 → 60 → 50 → 55 → 57 → 56'dır. Silinecek düğüm 60'tır. 60'ın iki çocuğu vardır: 50 ve 70. Her ikisinin de altında kalabalık birer alt ağaç bulunur.



Şekil 11.8: Silme örneklerinde kullanılan ağaç. 40, 20, 60, 10, 30, 50, 70, 42, 55, 65, 73, 52, 57, 66, 56 sırasıyla eklenmiştir; derinliği 6'dır. Silinecek düğüm iki çocuklu 60'tır.

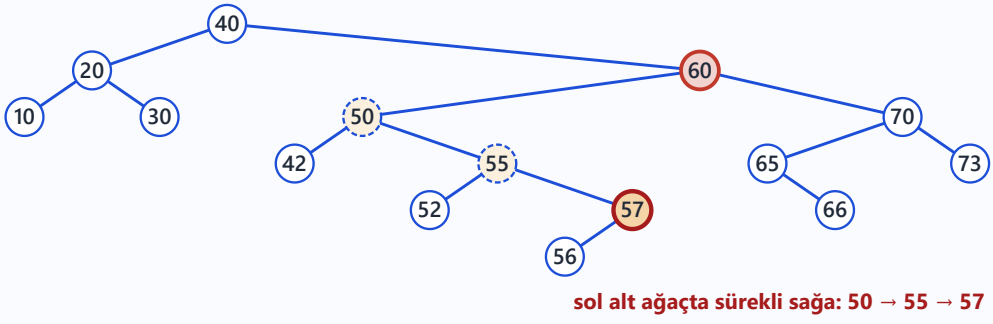
11.5.2 İki çocuklu düğüm: sol alt ağacın en büyüğü

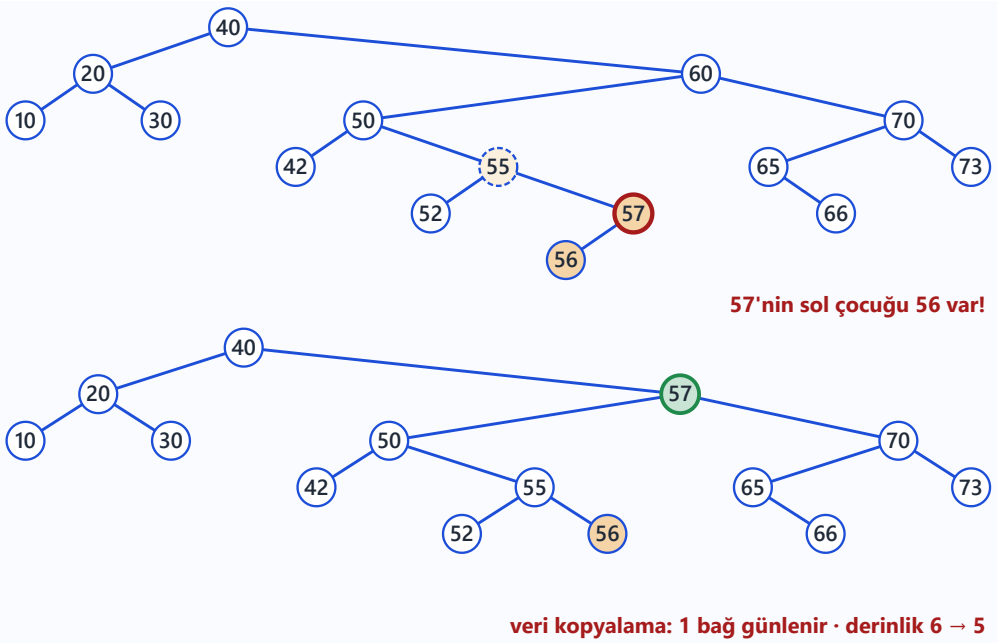
60'ın yerine hangi değer gelebilir? Yeni değer, sol alt ağaçtaki her değerden büyük, sağ alt ağaçtaki her değerden küçük olmalıdır. İki aday vardır. Birincisi sol alt ağacın en büyük değeridir. Bu değer, kök ortada sırada 60'tan hemen önce gelir; ona 60'ın **öncülü** (*predecessor*) denir. İkincisi sağ alt ağacın en küçük değeridir. Bu değer kök ortada sırada 60'tan hemen sonra gelir ve 60'ın **ardılıdır** (*successor*). İkisi de 60'ın yerine konduğunda sıralama korunur. Sunumun birinci yöntemi öncülü kullanır.

Sol alt ağaçtaki en büyük değer nasıl bulunur? Bir alt ağaçta her sağ çocuk ebeveyninden büyüktür. Bu yüzden en büyük değer, alt ağacın kökünden başlayıp sağ işaretçiler üzerinden ilerleyerek bulunur. Sağ çocuğu olmayan ilk düğüm en büyüktür. Örnekte 60'ın sol alt ağacının kökü 50'dir. 50'den sağa 55'e, 55'ten sağa 57'ye gidilir. 57'nin sağ çocuğu yoktur; sol alt ağacın en büyüğü 57'dir.

En büyük öğenin sağ çocuğu yoktur, ama bir sol alt ağacı olabilir. Örnekte 57'nin solunda 56 durur. 57 yerinden alınırken bu sol çocuk unutulmamalıdır. 56, 57'nin ebeveyni olan 55'in sağ bağına aktarılır. Bu güvenlidir: 56, 55'in sağ alt ağacındaydı ve 55'ten büyüktür.

Taşıma iki yolla yapılabilir. Birinci yolda 57'nin *düğümü* fiziksel olarak 60'ın yerine taşınır. Bunun için dört bağ değişir: 40'ın sağ bağı 57'yi, 55'in sağ bağı 56'yı, 57'nin sol bağı 50'yi, 57'nin sağ bağı 70'i gösterir. İkinci yolda 60'ın düğümü `free` edilmez. Yalnız veri alanına 57 yazılır. Sonra eski 57 düğümü, tek çocuklu bir düğüm gibi silinir. 55'in sağ bağı 56'ya güncellenir ve bu tek bağ değişikliği yeter. İkinci yol daha kısadır ve sunumdaki kod bu yolu izler. Düğümde veriden başka büyük alanlar varsa veri kopyalamak pahalı olabilir. O durumda birinci yol tercih edilir.





Şekil 11.9: Birinci yöntem. Üstte sol alt ağaçta sürekli sağa gidilerek en büyük değer bulunur: 50 → 55 → 57. Ortada 57'nin sol çocuğu 56 görülür; 56, 55'in sağ bağına aktarılmalıdır. Altta veri kopyalama yolu uygulanmıştır: 60'ın yerine 57 yazılmış, yalnız 55'in sağ bağı güncellenmiştir. Ağacın derinliği 6'dan 5'e inmiştir.

11.5.3 Silme yordamı ve işaretçinin işaretçisi

Silme yordamı da ağacı `agac_gost *L` olarak alır. Silinen düğüm kökse çağıranın kök işaretçisi değişmelidir. Ama yordamın asıl inceliği, yalnız kök için değil, ağacın her bağı için aynı fikri kullanmasıdır.

```
void sil(int x, agac_gost *L) {
  agac_gost p = *L, *bag = L;  // bag: p'ye giden
  while (p && p->veri != x) {  // x'i ara
    bag = (x < p->veri) ? &p->solcocuk
                        : &p->sagcocuk;
    p = *bag;
  }
  if (!p) return;              // x yok
  if (p->solcocuk && p->sagcocuk) {  // 2 çocuk
    agac_gost *b = &p->solcocuk;     // sol alt
    while ((*b)->sagcocuk)           // ağacın en
      b = &(*b)->sagcocuk;           // büyüğü
    p->veri = (*b)->veri;      // veriyi taşı
    bag = b;  p = *b;          // onu sileceğiz
  }
  *bag = p->solcocuk ? p->solcocuk : p->sagcocuk;
  free(p);
}
```

Yordamın merkezinde `bag` değişkeni durur. Tipi `agac_gost *`, yani işaretçiye işaretçidir. `bag`, o an incelenen `p` düğümünü gösteren işaretçinin *adresini* tutar. Bu işaretçi ya çağıranın kök değişkenidir ya da bir ebeveynin `solcocuk` veya `sagcocuk` alanıdır. `*bag`'e yazmak, `p`'ye giden bağı değiştirir. Böylece ebeveynin kim olduğunu ve `p`'nin onun hangi çocuğu olduğunu ayrıca tutmaya gerek kalmaz. Ekleme yordamındaki `q` işaretçisi ve son satırdaki "sol mu, sağ mı?" sorusu burada tek bir değişkende birleşmiştir.

- **2. satır.** `p` kökü, `bag` ise kök işaretçisinin adresini (`L`) tutar. Başta `*bag == p`'dir.
- **3.–7. satırlar.** `x` aranır. Her adımda önce `bag`, `p`'nin sol ya da sağ alanının adresine çevrilir. Sonra `p = *bag` ile bir düzey inilir. Döngü bitince ya `p` `NULL`'dır ya da `p->veri == x`'tir. Her iki durumda da `*bag == p` eşitliği korunur.
- **8. satır.** `x` ağaçta yoksa yordam hiçbir şey yapmadan döner.
- **9.–15. satırlar.** İki çocuk varsa sol alt ağacın en büyüğü aranır. `b` da bir işaretçinin adresidir. Başta 60'ın sol alanını gösterir; `(*b)->sagcocuk` `NULL` olana kadar sağ alanlara ilerler. Döngü bitince `*b` en büyük düğümdür ve `b`, bu düğümü gösteren bağın adresidir. 13. satır en büyük değeri silinecek düğümün veri alanına kopyalar. 14. satırla silinecek düğüm değişir: artık eski en büyük düğüm

silinecektir ve ona giden bağ `b`'dir. Bu düğümün sağ çocuğu olmadığından en çok bir çocuğu vardır.

- **16. satır.** Bu noktada `p`'nin en çok bir çocuğu vardır. Bağ, varsa sol çocuğa, yoksa sağ çocuğa çevrilir. İki çocuk da yoksa sağ çocuk `NULL`'dır ve bağ `NULL` olur. Yaprak, tek çocuk ve iki çocuğun ikinci aşaması aynı satırla çözülür.
- **17. satır.** Ağaçtan ayrılmış düğüm `free` edilir.

Son iki satırın sırası önemlidir. Önce `*bag` güncellenir, sonra `p` serbest bırakılır. Sıra ters olsaydı 16. satır serbest bırakılmış bir düğümün alanlarını okurdu. Bir başka ayrıntı da 11. satırdaki `(*b)->sagcocuk` yazımıdır. Parantez gereklidir, çünkü `->` işleci `*`'dan önce uygulanır. `*b->sagcocuk` yazılsaydı derleyici `b`'yi bir yapı işaretçisi sayardı ve kod derlenmezdi.

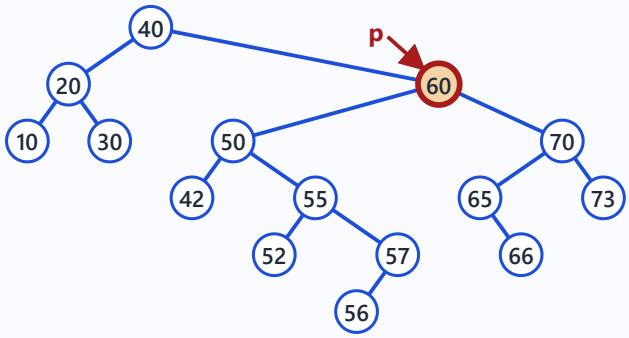
Yordam silinen değeri bulamazsa sessizce döner. Silme sonrasında düğüm sayısı bir azalır. Sürücü programda bu da denetlendi: her silmede `free` sayacı bir arttı, `malloc` sayacı değişmedi. Örnek ağaçtaki 15 değerin hepsi tek tek silindiğinde kök işaretçisi `NULL` oldu. 2000 rastgele değerli bir ağaçtan bütün değerler rastgele sırayla silindi. Her silmeden sonra ağacın hâlâ ikili arama ağacı olduğu denetlendi ve sonunda `malloc` ile `free` sayıları eşit çıktı.

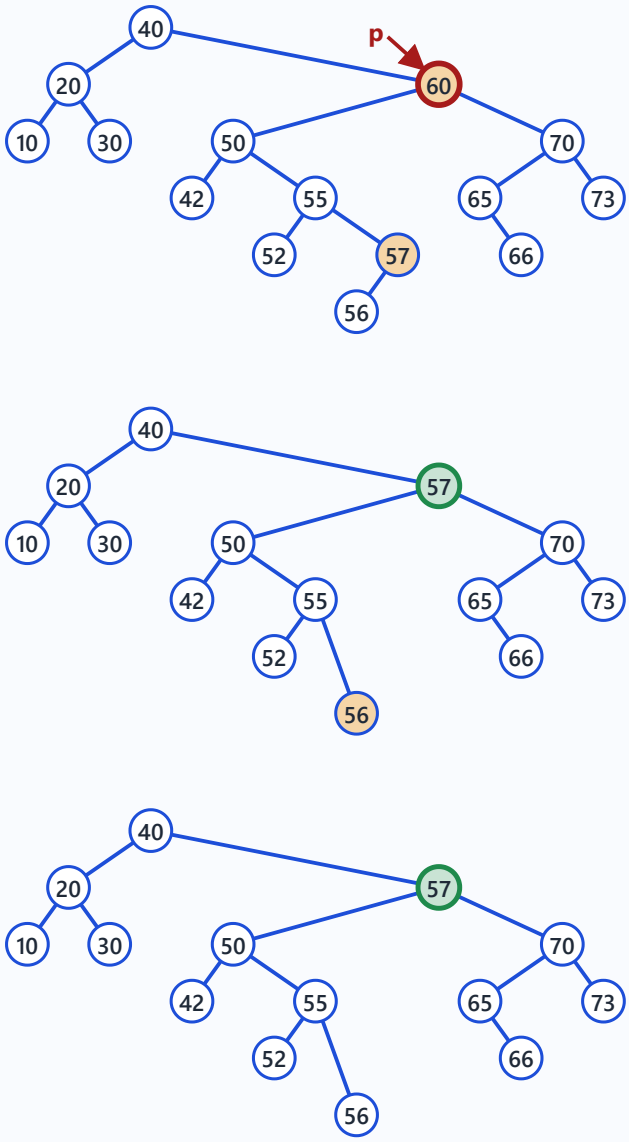
11.5.4 Kod izleme: 60'ı silmek

Tablo 11.6 `sil(60, &L)` çağrısını adım adım izler. `bag` ve `b` adresleri "&40->sagcocuk" biçiminde, yani gösterdikleri alanla yazıldı.

Satır	`p`	`bag`	`b` / `*b`	Açıklama
2	40	`L`	—	Başlangıç: `bag` kök işaretçisinin adresi.
4–6	60	&40->sagcocuk	—	*60 > 40*: sağa inilir; döngü 60'ta biter.
9–10	60	&40->sagcocuk	&60->solcocuk / 50	İki çocuk var: sol alt ağaca geçilir.
11–12	60	&40->sagcocuk	&50->sagcocuk / 55	50'nin sağı var.
11–12	60	&40->sagcocuk	&55->sagcocuk / 57	55'in sağı var.
11	60	&40->sagcocuk	&55->sagcocuk / 57	57'nin sağı yok: en büyük 57.
13	60 (veri 57)	&40->sagcocuk	aynı	60'ın veri alanına 57 yazılır.
14	eski 57	&55->sagcocuk	aynı	Silinecek düğüm artık eski 57.
16	eski 57	&55->sagcocuk	—	`*bag = 56`: 55'in sağı 56 olur.
17	—	—	—	Eski 57 düğümü `free` edilir.

Tablo 11.6: `sil(60, &L)` çağrısının izlemesi.





Şekil 11.10: `sil(60, &L)` izlemesinden dört kare. Soldan sağa ve yukarıdan aşağı: `bag` 40'ın sağ alanını gösteriyor ve `p` 60'ta; `b` sol alt ağacın en büyüğüne (57) ulaşmış; 55'in sağ bağı 56'ya çevrilmiş; eski düğüm serbest bırakılmış, derinlik 5'e inmiş.

Silmeden sonra ağaç 40(20(10, 30), 57(50(42, 55(52, 56)), 70(65(−, 66), 73))) biçimindedir. Kök ortada gezinme 10 20 30 40 42 50 52 55 56 57 65 66 70 73'tür ve hâlâ sıralıdır. Sürücü program bu ağacı ve derinliğin 5 olduğunu doğruladı.

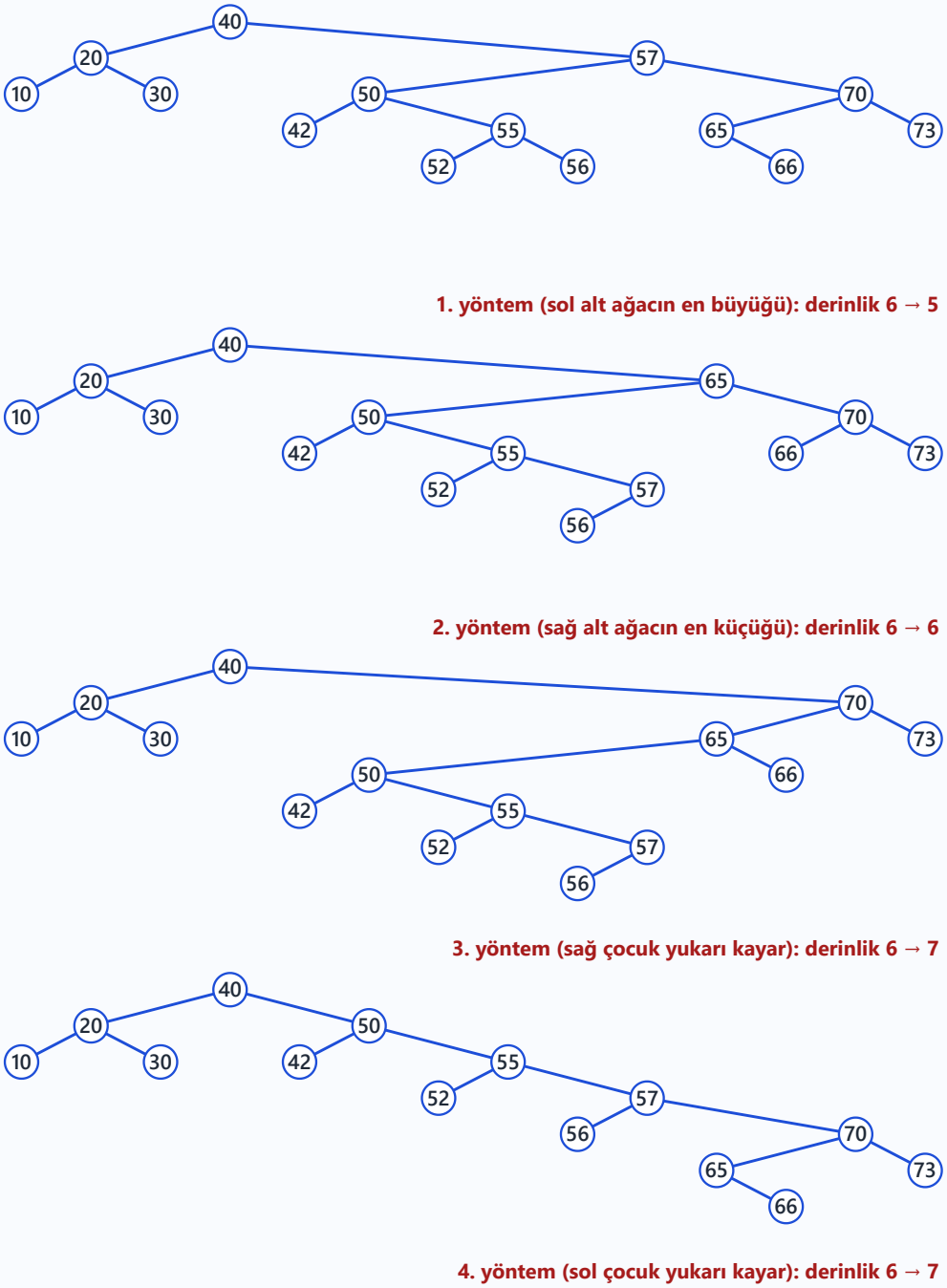
Silme yordamının maliyeti iki yolun toplamıdır: silinecek düğüme inen yol ve oradan en büyük öğeye inen yol. Bu ikisi birlikte kökten bir yaprağa giden tek bir yoldan uzun olamaz. Toplam $O(h)$'dir: dengeli ağaçta $O(\lg n)$, çarpık ağaçta $O(n)$. Yordam yinelemelidir ve ek bellek olarak birkaç işaretçi kullanır: $O(1)$.

> **Sık yapılan hata**
>
> İki çocuklu silmede taşınan öğenin sol çocuğunu unutmak. Örnekte 57 yukarı alınırken 55'in sağ bağı `NULL` yapılırsa 56 ağaçtan kopar. Düğüm hâlâ bellektedir, ama hiçbir işaretçi onu göstermez: hem değer kaybolur hem de bellek sızar. Sunumdaki kod bunu 16. satırla önler; bağı her zaman silinen düğümün varsa çocuğuna çevirir.

11.5.5 Dört yöntem ve derinliğe etkisi

Sunum iki çocuklu silme için dört yöntemi karşılaştırır. İlk ikisi silinen değerin yerine bir değer koyar, son ikisi bir alt ağacı yukarı taşır.

1. **Sol alt ağacın en büyüğü.** Önceki kesimde anlatılan yöntemdir. 57 yukarı çıkar, 56 55'in sağına bağlanır. Derinlik 5'e iner.
2. **Sağ alt ağacın en küçüğü.** Ayna yöntemidir. 60'ın sağ alt ağacında sürekli sola gidilir: 70 → 65. 65'in solu yoktur; en küçük 65'tir. 60'ın yerine 65 yazılır. 65'in sağ çocuğu 66, 70'in soluna bağlanır. 50'nin altındaki uzun kol yerinde kaldığı için derinlik 6 olarak kalır.
3. **Sağ çocuk yukarı.** Silinen düğümün yerine sağ çocuğu (70) kayar. Sol alt ağacın (50'nin alt ağacı) asılacak bir yere ihtiyacı vardır. 50'deki her değer 70'in alt ağacındaki her değerden küçüktür. Bu yüzden 50 alt ağacı, 70 alt ağacının en küçüğünün (65) soluna asılır. Derinlik 7'ye çıkar.
4. **Sol çocuk yukarı.** Üçüncünün aynasıdır. Yerine sol çocuk (50) kayar. 70 alt ağacı, 50 alt ağacının en büyüğünün (57) sağına asılır. 40 → 50 → 55 → 57 → 70 → 65 → 66 yolu yedi düğüm uzunluğundadır; derinlik 7'dir.



Şekil 11.11: 60'ı silmenin dört yöntemi, yukarıdan aşağı. Birinci yöntemde 57 yukarı çıkar ve derinlik 5'e iner. İkinci yöntemde 65 taşınır, 66 70'in soluna bağlanır ve derinlik 6 kalır. Üçüncü yöntemde 70 yukarı kayar ve 50 alt ağacı 65'in soluna asılır. Dördüncü yöntemde 50 yukarı kayar ve 70 alt ağacı 57'nin sağına asılır. Son ikisinde derinlik 7'ye çıkar.

Dört sonucun hepsi geçerli ikili arama ağacıdır. Sürücü programda dördü de ayrı ayrı uygulandı. Her birinde kök ortada gezinme aynı sıralı diziyi verdi. Derinlikler sırasıyla 5, 6, 7 ve 7 çıktı. Aralarındaki fark, ağacın biçiminde ve dolayısıyla sonraki aramaların maliyetindedir. Son iki yöntem bir alt ağacı başka bir alt ağacın en altına astığı için derinliği büyütür. İlk iki yöntem yalnız tek bir düğümü yukarı alır ve derinliği artırmaz. Hoca bu yüzden şu ilkeyi önerir: silinen öğenin kök olduğu ağaç olabildiğince dengeli kalacak biçimde silinmelidir. Pratikte ilk iki yöntem kullanılır. Bazı gerçeklemeler, sürekli aynı yanı kullanmanın ağacı zamanla bir yana yatırmaması için ikisi arasında sırayla geçiş yapar.

İkinci yöntemin kodu birinci yöntemin aynasıdır: sol yerine sağ alt ağaca geçilir ve sağ yerine sol işaretçiler izlenir.

```
void sil_sag(int x, agac_gost *L) {    /* 2. yöntem */
  agac_gost p = *L, *bag = L;
  while (p && p->veri != x) {
    bag = (x < p->veri) ? &p->solcocuk
                        : &p->sagcocuk;
    p = *bag;
  }
  if (!p) return;
  if (p->solcocuk && p->sagcocuk) {    // 2 çocuk
    agac_gost *b = &p->sagcocuk;       // sağ alt
    while ((*b)->solcocuk)             // ağacın en
      b = &(*b)->solcocuk;             // küçüğü
    p->veri = (*b)->veri;
    bag = b;  p = *b;
  }
  *bag = p->solcocuk ? p->solcocuk : p->sagcocuk;
  free(p);
}
```

Son satırdaki ifade her iki yöntemde de aynıdır. Sağ alt ağacın en küçüğünün sol çocuğu yoktur. Bu yüzden ifade bağı onun sağ çocuğuna çevirir. Örnekte 70'in sol bağı 66'yı gösterir.

11.6 Dengeli ağaçlar

Önceki kesimlerdeki her işlemin maliyeti *O(h)* idi. Bu kesim *h*'nin ne kadar büyüyebileceğini sayılarla gösterir. Ardından yüksekliği her durumda *O(lg n)* tutan AVL ağacını ayrıntılı anlatır. Kırmızı-siyah ağaç ile B-ağacına da kısaca değinir.

11.6.1 Dengesizliğin maliyeti

Tablo 11.7, *n* değerli bir ağacın en iyi ve en kötü durumdaki derinliğini karşılaştırır. Dengeli ağacın derinliği *⌈lg(n + 1)⌉*, çarpık ağacınki *n*'dir.

n	**Dengeli: *⌈lg(n + 1)⌉***	**Çarpık: *n***
1.000	10	1.000
1.000.000	20	1.000.000
2.000.000	21	2.000.000
10^9	30	10^9

Tablo 11.7: Dengeli ve çarpık ağaçta en kötü arama uzunluğu (bakılan düğüm sayısı).

Tablodaki fark çarpıcıdır. Hoca bunu tek cümleyle özetler: *O(n)* ile *O(lg n)* arasındaki fark, 2.000.000 ile 21 arasındaki fark gibidir. Bir milyar kayıtlık bir dizinde dengeli ağaç bir kaydı 30 adımda bulur. Çarpık ağaç ise en kötü durumda bir milyar adım atar.

Rastgele sırada gelen veride durum o kadar kötü değildir. Değerlerin her sırası eşit olasılıklı olduğunda *n* değerli bir ağacın ortalama derinliği *Θ(lg n)*'dir. Başarılı bir aramanın ortalama karşılaştırma sayısı yaklaşık $2 \ln n \approx 1{,}39 \lg n$'dir. Sürücü programda bu iddia ölçüldü. 1000, 10.000, 100.000 ve 1.000.000 rastgele değer boş ağaçlara eklendi ve düğümlerin ortalama düzeyi hesaplandı. Sonuçlar 12,1; 16,7; 21,0 ve 25,2 çıktı. Kesin formül $2(1 + 1/n)H_n - 3$ aynı boyutlar için 12,0; 16,6; 21,2 ve 25,8 verir. Buradaki H_n, $1 + 1/2 + \ldots + 1/n$ toplamıdır. Ölçümler formüle çok yakındır. $1{,}39 \lg n$ değeri ise bu formülün baş terimidir ve her boyutta ölçülenden yaklaşık iki fazladır. Ağacın en derin yolu daha uzundu: bir milyon değerde ortalama 47,7 düzey. Bu da $\lg n \approx 20$'nin iki katından biraz fazladır.

Ortalama durumun iyi olması güvence vermez. En kötü durum sıralı ya da neredeyse sıralı veridir ve gerçekte sık görülür. Bu yüzden her ekleme ve silmeden sonra yerel döndürmelerle derinliği *O(lg n)*'de tutan ağaçlar geliştirilmiştir. AVL ağacı, kırmızı-siyah ağaç ve B-ağacı bunların en bilinenleridir.

11.6.2 AVL ağacı

AVL ağacı, kendini dengeleyen ilk ikili arama ağacıdır. Adı, 1962'de yapıyı yayımlayan G. M. Adelson-Velski ile E. M. Landis'in baş harflerinden gelir. Bu kesimde bir düğümün **yüksekliği**, o düğümün kök olduğu alt ağacın derinliğidir. Yaprağın yüksekliği 1, boş ağacınki 0'dır. Kodda bu değer `yuk` adıyla geçer.

Tanım 11.2

Bir düğümün **denge faktörü** (*balance factor*), sol alt ağacının yüksekliğinden sağ alt ağacının yüksekliğinin çıkarılmasıyla bulunur: *yük(sol) − yük(sağ)*. **AVL ağacı** (*AVL tree*), her düğümünün denge faktörü −1, 0 ya da +1 olan ikili arama ağacıdır.

Koşul her düğümde, yalnız kökte değil, sağlanmalıdır. Denge faktörü +1 olan düğüm sola, −1 olan düğüm sağa biraz ağırdır; 0 olan düğüm tam dengelidir. Koşulun ne kadar güçlü olduğunu aşağıdaki teorem gösterir.

Teorem 11.3 (AVL ağacının yüksekliği)

n düğümlü bir AVL ağacının yüksekliği yaklaşık *1,44 lg(n + 2)*'yi geçmez. Bu yüzden AVL ağacında arama, ekleme ve silme *O(lg n)* sürer.

Kanıt taslağı. Yüksekliği *h* olan bir AVL ağacının en az kaç düğümü olabileceğine bakılsın; bu sayı *N(h)* olsun. *N(1) = 1* ve *N(2) = 2*'dir. Yüksekliği *h* olan en seyrek ağacın bir alt ağacının yüksekliği *h − 1* olmalıdır. Öteki alt ağacın yüksekliği, denge koşulu yüzünden en az *h − 2*'dir. İki alt ağaç da en seyrek biçimde seçilirse *N(h) = N(h − 1) + N(h − 2) + 1* bulunur. Bu bağıntı Fibonacci sayılarınınkine benzer. *N(h) + 1* değerleri Fibonacci dizisini izler: 2, 3, 5, 8, 13, … Fibonacci sayıları φ^h hızıyla büyür; burada $\varphi = (1 + \sqrt{5})/2 \approx 1{,}618$'dir. Öyleyse $n \geq N(h) \approx \varphi^h$, yani $h \leq \log_\varphi n = \lg n / \lg \varphi \approx 1{,}44 \lg n$ çıkar. Kesin hesap küçük düzeltmelerle *1,44 lg(n + 2)* sınırını verir. ■

Sürücü programda *N(h)* hesaplandı: *N(5) = 12*, *N(10) = 143*, *N(20) = 17.710*, *N(30) = 2.178.308*. Yani yüksekliği 30 olan bir AVL ağacında iki milyondan fazla düğüm bulunur. İki milyon düğümlü bir AVL ağacı en kötü durumda 29 düzeylidir, oysa çarpık bir ağaç iki milyon düzeyli olabilir.

Sunumdaki AVL düğümü, sıradan düğüme bir yükseklik alanı ekler. İki yardımcı yordam yükseklik ve denge faktörü hesabını kısaltır.

```
typedef struct avl *avl_gost;
struct avl {
    int      veri;
    int      yuk;        /* alt ağaç yüksekliği */
    avl_gost sol, sag;
};
int yuk(avl_gost t) { return t ? t->yuk : 0; }
int denge(avl_gost t) {
    return yuk(t->sol) - yuk(t->sag);
}
```

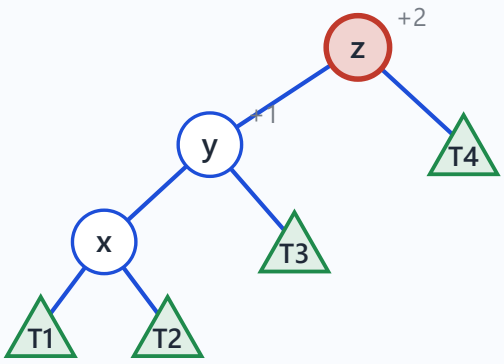
`yuk` yordamı boş ağaç için 0 döndürür. Bu sayede çağıran, alt ağacın `NULL` olup olmadığını ayrıca denetlemez. `denge` yordamı tanımdaki farkı hesaplar; `t`'nin boş olmadığını varsayar. Yükseklik her düğümde saklandığı için denge faktörü *O(1)*'de bulunur. Yükseklik saklanmasaydı her denetimde alt ağaçların derinliği yeniden hesaplanır ve her işlem *O(n)* olurdu. Düğüm boyutu değişmez: `int` veri ve `int` yükseklik birlikte 8 bayt tutar, iki işaretçiyle toplam 24 bayt eder. Sıradan düğümdeki 4 baytlık hizalama boşluğu burada yükseklik alanına gider.

Eklemenin genel planı şöyledir. Ekleme önce sıradan ikili arama ağacı eklemesiyle yapılır. Sonra eklenen düğümden köke doğru yol üzerindeki yükseklikler ve denge faktörleri güncellenir. Faktörü ±2 olan ilk ata bulunursa ağaç o atada yerel bir döndürmeyle düzeltilir. Yeni düğümün o atanın hangi torununda olduğuna göre dört durum vardır: LL, RR, LR ve RL. LL ve RR tek döndürmeyle, LR ve RL çift döndürmeyle düzelir.

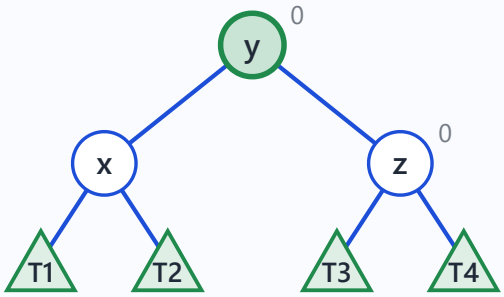
11.6.3 Tek döndürme: LL ve RR

Döndürme (*rotation*), ağacın sıralamasını koruyarak yerel yapısını değiştiren bir işlemdir. Sunum şekillerinde `z` dengesi bozulan ilk atadır. `y` onun çocuğu, `x` ise torunudur. T1–T4 üçgenleri, yükseklikleri belli birer alt ağaçtır.

LL durumu. Yeni düğüm `z`'nin sol çocuğunun (`y`) sol alt ağacına girmiştir. `z`'nin faktörü +2'ye çıkar. Düzeltmek için `z` sağa döndürülür. `y` yukarı çıkar ve `z` onun sağ çocuğu olur. `y`'nin sağ alt ağacı T3, `z`'nin soluna geçer. Bu geçiş güvenlidir: T3'teki değerler `y`'den büyük ve `z`'den küçüktür. Kök ortada sıra döndürmeden önce ve sonra aynıdır: T1 x T2 y T3 z T4.



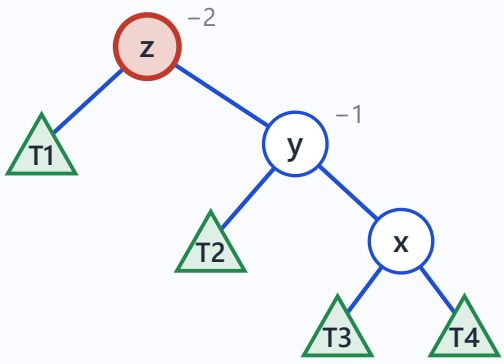
LL: z'nin solunun solu ağır



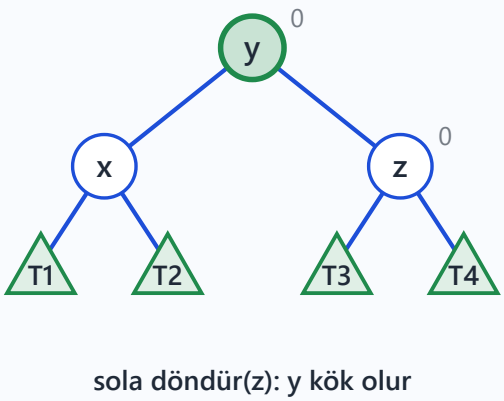
sağa döndür(z): y kök olur

Şekil 11.12: LL durumu ve sağa döndürme. Solda `z`'nin faktörü +2'dir; ağırlık solun solundadır. Sağda `z` sağa döndürülmüş, `y` alt ağacın kökü olmuş ve T3 `z`'nin soluna geçmiştir.

RR durumu LL'nin aynasıdır. Yeni düğüm `z`'nin sağ çocuğunun sağ alt ağacına girmiştir ve `z`'nin faktörü −2'dir. `z` sola döndürülür; `y`'nin sol alt ağacı T2, `z`'nin sağına geçer.



RR: z'nin sağının sağı ağır

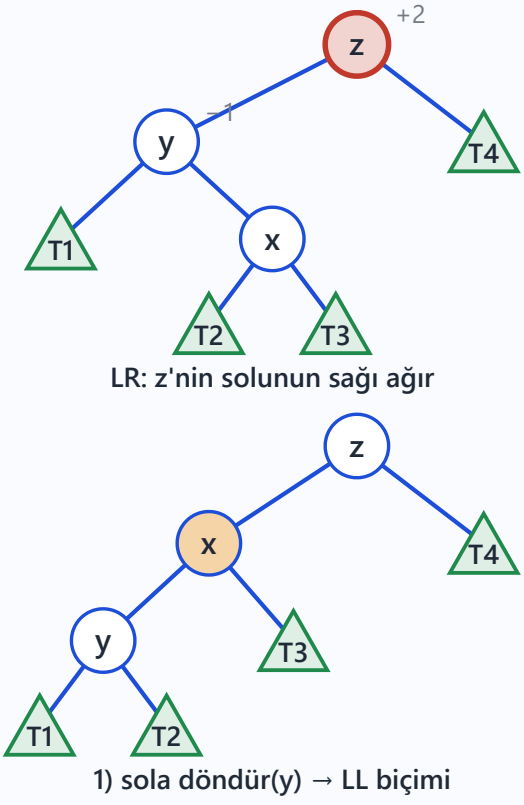


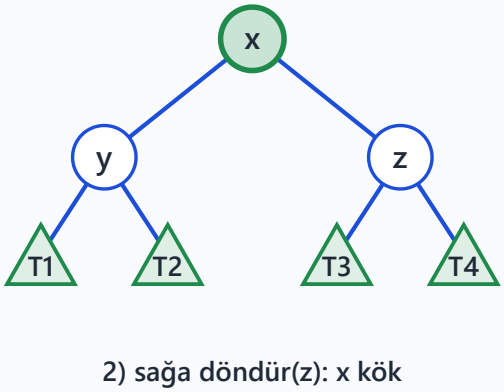
Şekil 11.13: RR durumu ve sola döndürme. Solda `z` 'nin faktörü −2'dir; ağırlık sağın sağındadır. Sağda `z` sola döndürülmüş, `y` kök olmuş ve T2 `z` 'nin sağına geçmiştir.

Her iki durumda da döndürme sonrasında alt ağacın yüksekliği eklemeden önceki değerine döner. LL için bu şöyle gösterilir. Eklemeden önce `x` 'in alt ağacının yüksekliği *k* olsun. `z` dengeli olduğuna ve ekleme onu +2'ye çıkardığına göre, eklemeden önce T3'ün yüksekliği *k*, `y` 'ninki *k + 1*, T4'ünki *k*'dir. Alt ağacın yüksekliği *k + 2*'dir. Ekleme `x` 'in alt ağacını *k + 1*'e, `y` 'yi *k + 2*'ye çıkarır. Döndürmeden sonra `z` 'nin iki alt ağacı T3 ve T4'tür. İkisinin yüksekliği de *k* olduğu için `z` 'nin yüksekliği *k + 1* olur. `y` 'nin altında yüksekliği *k + 1* olan iki alt ağaç vardır: `x` 'in alt ağacı ve `z` . Böylece `y` 'nin yüksekliği *k + 2* olur. Alt ağacın yüksekliği eklemeden önceki gibidir. Bu yüzden yukarıdaki ataların faktörleri değişmez ve eklemede tek bir döndürme yeter.

11.6.4 Çift döndürme: LR ve RL

LR durumu. Yeni düğüm `z` 'nin sol çocuğunun (`y`) *sağ* alt ağacına, yani `x` 'in altına girmiştir. Burada tek bir sağa döndürme işe yaramaz. `z` sağa döndürülseydi, ağır olan `x` alt ağacı `z` 'nin soluna geçerdi ve dengesizlik bu kez öteki yana kayardı. Çözüm iki adımdır. Önce `y` sola döndürülür. `x` , `y` 'nin yerine çıkar ve durum LL biçimine dönüşür. Sonra `z` sağa döndürülür. Torun `x` , iki atasının üstüne çıkar. `y` onun solunda, `z` sağında yer alır. `x` 'in eski alt ağaçlarından T2 `y` 'nin sağına, T3 `z` 'nin soluna geçer. İki tek döndürmeden oluşan bu işleme **çift döndürme** (*double rotation*) denir.





Şekil 11.14: LR durumu ve çift döndürme. Üstte `z`'nin faktörü +2, `y`'ninki −1'dir; ağırlık solun sağındadır. Ortada `y` sola döndürülmüş ve durum LL biçimine gelmiştir. Altta `z` sağa döndürülmüş, `x` kök olmuştur.

RL durumu LR'nin aynasıdır. Yeni düğüm `z`'nin sağ çocuğunun sol alt ağacındadır. Önce sağ çocuk sağa, sonra `z` sola döndürülür. LR ile RL'nin sonucunda `x` üç düğümün ortancasıdır; bu yüzden köke o çıkar. Dört durumun hepsinde sonuç aynı biçimdedir: üç düğümden ortanca değerli olan kök, küçüğü sol, büyüğü sağ çocuk olur; dört alt ağaç sırayla altlarına dağılır.

11.6.5 Döndürme kodu ve izlemesi

Sağa döndürme kodu yalnız iki işaretçi değiştirir ve iki yüksekliği yeniden hesaplar. Sunumdaki adlandırma, Weiss'in kitabındaki gibi `k1` ve `k2`'dir: `k2` dengesi bozulan düğüm, `k1` onun sol çocuğudur.

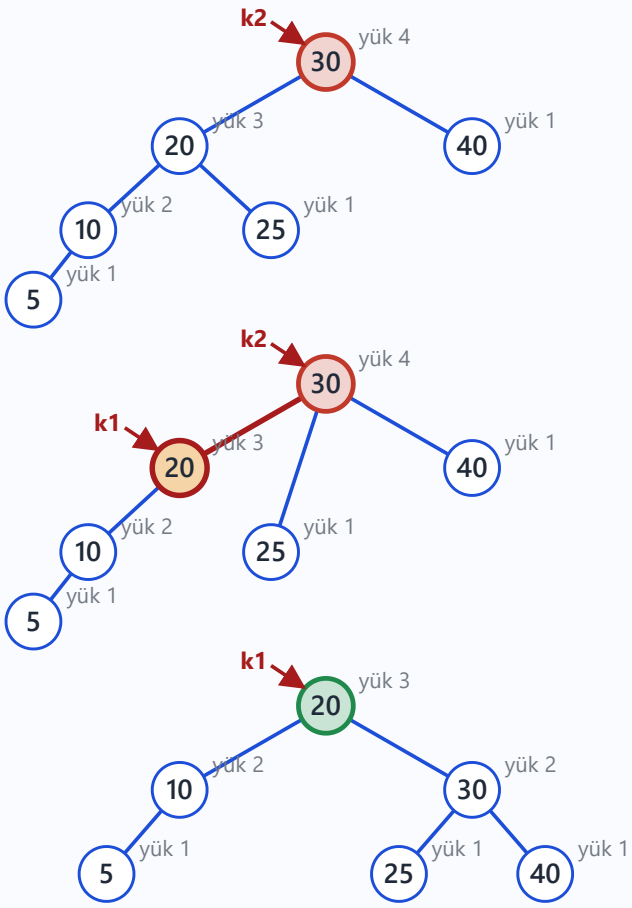
```
avl_gost sag_dondur(avl_gost k2) {  // LL
  avl_gost k1 = k2->sol;
  k2->sol = k1->sag;
  k1->sag = k2;
  k2->yuk = 1 + MAX(yuk(k2->sol), yuk(k2->sag));
  k1->yuk = 1 + MAX(yuk(k1->sol), k2->yuk);
  return k1;                      // yeni kök
}
```

- **2. satır.** `k1`, `k2`'nin sol çocuğudur ve yeni kök olacaktır.
- **3. satır.** `k1`'in sağ alt ağacı `k2`'nin soluna geçer. Bu, şekildeki T3'ün geçişidir.
- **4. satır.** `k2`, `k1`'in sağ çocuğu olur.
- **5.–6. satırlar.** Yükseklikler aşağıdan yukarı güncellenir. Önce `k2`'nin, sonra `k1`'in yüksekliği hesaplanır. `k1`'in yeni sağ çocuğu `k2`'dir ve onun yüksekliği bir önceki satırda bulunmuştur. Sıra tersine çevrilirse `k1` eski, yanlış bir değeri kullanır.
- **7. satır.** Yeni kök döner. Çağıran, ebeveynin işaretçisini bu düğüme bağlar.

Sunum yordamı küçük bir örnekle izler. Ağaçta 30 köktür; solunda 20, sağında 40 vardır. 20'nin çocukları 10 ve 25'tir. 5 eklenince 10'un soluna girer. 30'un faktörü *yük(20) − yük(40) = 3 − 1 = +2* olur. Yeni değer solun solundadır; durum LL'dir ve `sag_dondur(30)` çağrılır.

Satır	İşlem	Sonuç
2	`k1 = k2->sol`	`k1` = 20
3	`k2->sol = k1->sag`	25, 30'un soluna geçer (*20 < 25 < 30*); 25'i şimdilik iki işaretçi gösterir.
4	`k1->sag = k2`	30, 20'nin sağ çocuğu olur.
5	`k2->yuk`	*1 + MAX(yük(25) = 1, yük(40) = 1) = 2*
6	`k1->yuk`	*1 + MAX(yük(10) = 2, 2) = 3*
7	`return k1`	Yeni kök 20; ağaç 20(10(5, –), 30(25, 40)).

Tablo 11.8: `sag_dondur(30)` çağrısının izlemesi (5 eklendikten sonra).



Şekil 11.15: `sag_dondur` izlemesinden üç kare. Üstte 5 eklenmiş ve 30'un faktörü +2 olmuştur. Ortada 25 30'un soluna geçirilmiş ve 30 20'nin sağına bağlanmıştır; yükseklikler henüz güncellenmemiştir. Altta yükseklikler düzeltilmiş ve 20 kök olmuştur.

Döndürmeden sonra alt ağacın yüksekliği 3'tür; 5 eklenmeden önce de 3 idi. Yalnız iki işaretçi değişti. Döndürme sabit sayıda işlem yapar ve maliyeti *O(1)*'dir. Sürücü programda aynı ekleme dizisi (30, 20, 40, 10, 25, 5) uygulandı. Tek döndürme `sag_dondur(30)` oldu ve sonuç ağacı tablodaki gibi çıktı.

Sunum sola döndürmenin kodunu vermez, ama AVL ekleme kodu onu kullanır. Sola döndürme sağa döndürmenin aynasıdır: `sol` ile `sag` yer değiştirir. Sürücü programlarda kullanılan sürüm ve yeni AVL düğümü ayıran yordam aşağıdadır.

```
avl_gost sol_dondur(avl_gost k1) {  // RR
  avl_gost k2 = k1->sag;
  k1->sag = k2->sol;
  k2->sol = k1;
  k1->yuk = 1 + MAX(yuk(k1->sol), yuk(k1->sag));
  k2->yuk = 1 + MAX(k1->yuk, yuk(k2->sag));
  return k2;                       // yeni kök
}
avl_gost yeni_avl(int x) {
  avl_gost t = malloc(sizeof *t);
  if (!t) { fprintf(stderr, "Bellek yetersiz\n"); exit(EXIT_FAILURE); }
  t->veri = x;  t->yuk = 1;
  t->sol = t->sag = NULL;
  return t;
}
```

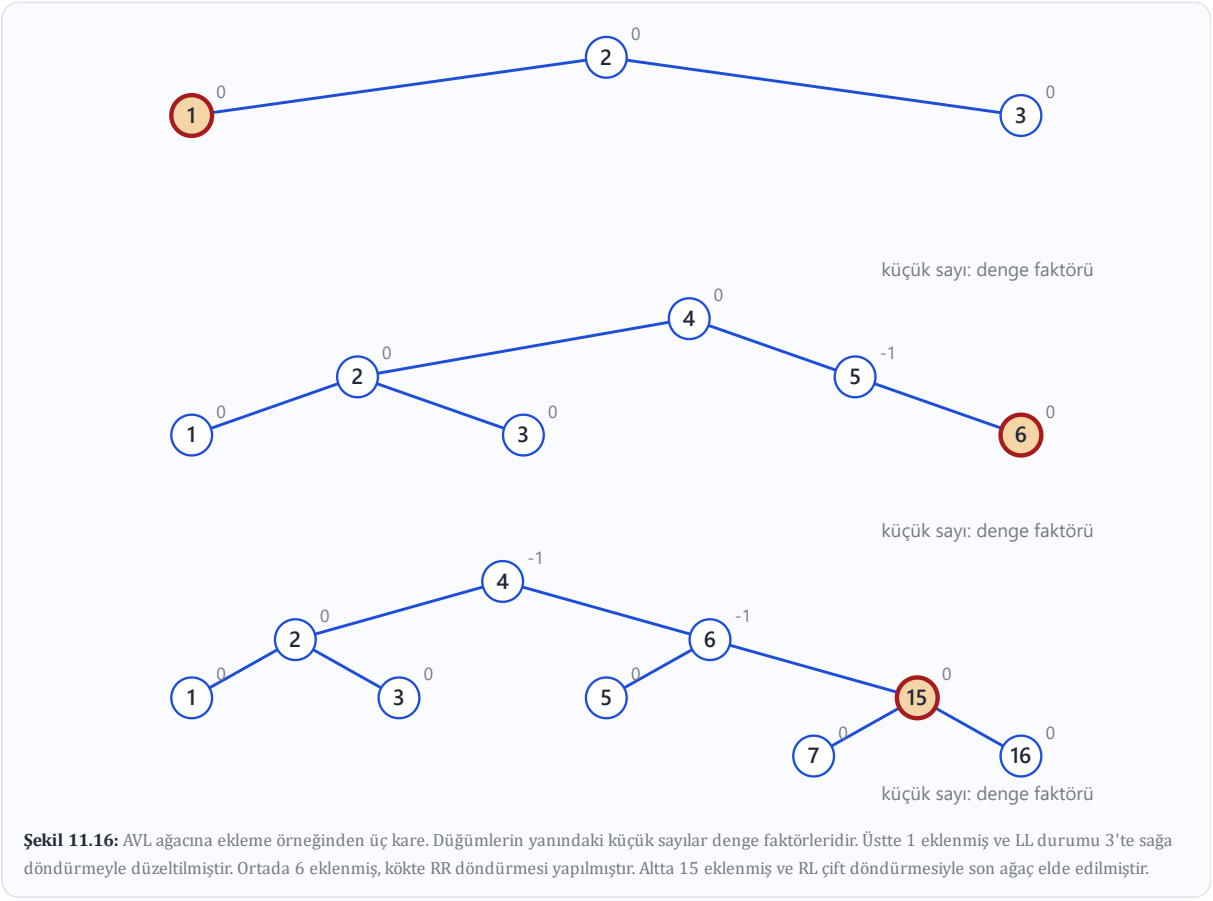
11.6.6 AVL ağacına ekleme

Sunum, Weiss'in kitabındaki örneği izler. Boş bir AVL ağacına sırasıyla 3, 2, 1, 4, 5, 6, 7, 16, 15 eklenir. Tablo 11.9 her eklemeden sonraki durumu verir.

Eklenen	Durum	Döndürme	Sonuç
3	—	yok	3
2	—	yok	3(2, –)
1	LL, 3'te +2	3'te sağa	2(1, 3)
4	—	yok	2(1, 3(–, 4))
5	RR, 3'te –2	3'te sola	2(1, 4(3, 5))
6	RR, 2'de –2	2'de sola	4(2(1, 3), 5(–, 6))
7	RR, 5'te –2	5'te sola	4(2(1, 3), 6(5, 7))
16	—	yok	4(2(1, 3), 6(5, 7(–, 16)))
15	RL, 7'de –2	16'da sağa, 7'de sola	4(2(1, 3), 6(5, 15(7, 16)))

Tablo 11.9: Boş AVL ağacına 3, 2, 1, 4, 5, 6, 7, 16, 15 ekleme. Ağaçlar kök(sol, sağ) biçiminde yazıldı; "–" boş alt ağaçtır.

Tablodaki üç adım ayrıca açıklanmaya değer. 6 eklendiğinde dengesizlik yeni düğümün ebeveyninde değil, kökte ortaya çıkar. 5'in faktörü −1, 4'ünki −1'dir; ama kök 2'nin sol alt ağacının yüksekliği 1, sağınınki 3'tür. Döndürme kökte yapılır ve 4 yeni kök olur. 7 eklendiğinde ise dengesizlik 5'tedir ve kök etkilenmez. 15 eklendiğinde yeni düğüm 7'nin sağ çocuğunun (16) soluna girer. Bu RL durumudur. Önce 16 sağa döndürülür, sonra 7 sola döndürülür ve 15, 7 ile 16'nın ebeveyni olur.



Şekil 11.16: AVL ağacına ekleme örneğinden üç kare. Düğümlerin yanındaki küçük sayılar denge faktörleridir. Üstte 1 eklenmiş ve LL durumu 3'te sağa döndürmeyle düzeltilmiştir. Ortada 6 eklenmiş, kökte RR döndürmesi yapılmıştır. Altta 15 eklenmiş ve RL çift döndürmesiyle son ağaç elde edilmiştir.

Son ağacın yüksekliği 4'tür. Aynı sıra sıradan bir ikili arama ağacına eklenseydi 3 → 4 → 5 → 6 → 7 → 16 → 15 zinciri oluşurdu ve derinlik 7 olurdu. Sürücü program her eklemeden sonra ağacı, faktörleri ve yapılan döndürmeleri yazdı. Sonuçlar tablonun her satırıyla aynı çıktı. Ayrıca her adımda saklanan yüksekliklerin gerçek yüksekliklere eşit olduğu ve bütün faktörlerin −1..+1 aralığında kaldığı denetlendi.

Sunumdaki ekleme kodu, özyinelemeli ikili arama ağacı eklemesinin dönüş yoluna denge denetimini ekler.

```
avl_gost avl_ekle(avl_gost t, int x) {
  if (!t) return yeni_avl(x);              /* malloc, yuk = 1        */
  if (x < t->veri)      t->sol = avl_ekle(t->sol, x);
  else if (x > t->veri) t->sag = avl_ekle(t->sag, x);
  else return t;                           /* tekrar eden değer yok  */
  t->yuk = 1 + MAX(yuk(t->sol), yuk(t->sag));
  int d = denge(t);
  if (d >  1 && x < t->sol->veri) return sag_dondur(t);           /* LL */
  if (d < -1 && x > t->sag->veri) return sol_dondur(t);           /* RR */
  if (d >  1) { t->sol = sol_dondur(t->sol); return sag_dondur(t); } /* LR */
  if (d < -1) { t->sag = sag_dondur(t->sag); return sol_dondur(t); } /* RL */
  return t;
}
```

- **2.–5. satırlar.** Sıradan özyinelemeli eklemedir. Boş alt ağaca gelinince yeni düğüm ayrılır ve yüksekliği 1 yapılır. Eşit değer gelirse ağaç değiştirilmeden döner.
- **6.–7. satırlar.** Özyinelemeden dönerken bu düğümün yüksekliği güncellenir ve denge faktörü hesaplanır. Bu satırlar yeni düğümden köke giden yoldaki her atada sırayla çalışır.
- **8. satır (LL).** Faktör +1'den büyükse ağırlık soldadır. `x` sol çocuğun değerinden küçükse yeni düğüm solun solundadır; tek sağa döndürme yeter.
- **9. satır (RR).** Aynası: faktör −1'den küçük ve `x` sağ çocuktan büyükse tek sola döndürme yapılır.
- **10. satır (LR).** Faktör +2 ama yeni düğüm solun sağındadır. Önce sol çocuk sola döndürülür, sonra düğüm sağa döndürülür.
- **11. satır (RL).** Aynası.
- **12. satır.** Denge bozulmamışsa düğüm olduğu gibi döner.

Kod durumu yeni değerle çocuğun değerini karşılaştırarak ayırt eder. Yeni değer sol çocuktan küçükse solun soluna, büyükse solun sağına girmiştir. Değerler tekrarsız olduğu için eşitlik olmaz. Kullanım `kok = avl_ekle(kok, x);` biçimindedir. Yeni kök dönüş değeriyle verildiği için `*L` parametresi gerekmez.

Eklemede en çok bir döndürme (tek ya da çift) yeterlidir. Döndürme, alt ağacın yüksekliğini eklemeden önceki değerine indirir; üstteki atalar bir değişiklik görmez. Kod yine de yolun bütün atalarında yükseklik hesabı ve denge denetimi yapar, ama bunlar sabit işlemlerdir. Toplam maliyet yolun uzunluğu, yani *O(lg n)*'dir. Özyineleme yığında *O(lg n)* çerçeve tutar. Silme de benzer biçimde yapılır, ama bir döndürme alt ağacın yüksekliğini düşürebilir. Bu düşüş bir üst atanın dengesini bozabilir. Silmede bu yüzden köke kadar *O(lg n)* döndürme gerekebilir.

Sürücü programda AVL ağacının büyük girdilerdeki davranışı da ölçüldü. 1, 2, …, *n* sıralı değerleri eklendiğinde yükseklik tam $\lfloor \lg(n+1) \rfloor$ çıktı: 1000 için 10, bir milyon için 20, iki milyon için 21. Sıralı veride hep sağa ağırlık bindiği için yalnız sola döndürme yapıldı: bir milyon eklemede 999.980 döndürme. Bir milyon rastgele değer eklendiğinde yükseklik 24 çıktı. Teoremdeki üst sınır bu boyut için 28'dir. Sıradan ikili arama ağacına aynı sıralı veri eklenseydi derinlik bir milyon olurdu.

11.6.7 Kırmızı-siyah ağaç ve B-ağacı

AVL ağacı dengeyi sıkı tutar. Bu sıkılığın bedeli, silmede çok sayıda döndürme gerekebilmesidir. Uygulamada iki başka dengeli ağaç da çok kullanılır.

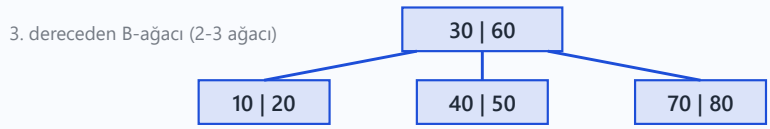
Kırmızı-siyah ağaç (*red-black tree*) her düğüme kırmızı ya da siyah bir renk verir ve renklere kurallar koyar:

- Her düğüm kırmızı ya da siyahtır; kök ve `NULL` yapraklar siyahtır.
- Kırmızı bir düğümün çocukları siyahtır; art arda iki kırmızı düğüm bulunmaz.
- Bir düğümden altındaki `NULL`'lara giden her yolda aynı sayıda siyah düğüm bulunur.

Bu kurallar en uzun yolun en kısa yolun iki katını geçmemesini sağlar. Sonuçta yükseklik en çok $2\lg(n+1)$'dir. Denge AVL'dekinden gevşektir. Buna karşılık ekleme ve silmedeki döndürme sayısı sınırlıdır. Eklemede en çok iki, silmede en çok üç döndürme ve bir miktar renk değişimi yeter. Ağacın yapısı daha az değiştiği için kırmızı-siyah ağaç çok güncellenen yapılarda tercih edilir. C++ dilindeki

`std::map`, Java'daki `TreeMap` ve Linux çekirdeğindeki birçok yapı kırmızı-siyah ağaçtır.

B-ağacı (*B-tree*) 1972'de R. Bayer ile E. McCreight'ın önerdiği *m* yollu dengeli bir arama ağacıdır. Bir düğümde birden çok anahtar ve en çok *m* çocuk bulunur. Kök dışındaki her düğüm *⌈m/2⌉ − 1* ile *m − 1* arasında anahtar tutar. Bütün yapraklar aynı düzeydedir. Bir düğüm dolduğunda ortadaki anahtarı ebeveynine çıkararak ikiye bölünür. Kök bölündüğünde yeni bir kök oluşur. Bu yüzden B-ağacı yapraklardan değil, kökten büyür.



Şekil 11.17: Küçük bir B-ağacı. Her düğüm birden çok sıralı anahtar tutar; anahtarların arasındaki boşluklardan çocuklara inilir. Bütün yapraklar aynı düzeydedir.

B-ağacının asıl kullanım yeri disk ve benzeri yavaş belleklerdir. Diskten veri bloklar hâlinde okunur ve bir bloğu okumak, bellekte binlerce karşılaştırma yapmaktan uzun sürer. B-ağacında bir düğüm bir disk bloğuna denk getirilir. *m* yüzlerce olabilir. Böylece ağacın yüksekliği $\log_m n$ düzeyine iner. Bir milyar kayıtlık bir dizin üç dört blok okumayla taranır. Veritabanı dizinleri ve dosya sistemleri, verinin yalnız yapraklarda tutulduğu türevi olan **B+ ağacını** kullanır.

11.6.8 Hangi arama ağacı?

Tablo 11.10 bu kesimde anılan arama ağaçlarını karşılaştırır.

Yapı	Arama / ekleme / silme (en kötü)	Denge koşulu	Nerede?
ikili arama ağacı	*O(n)* (ortalama *O(lg n)*)	yok	küçük, rastgele veri; öğretim
AVL ağacı	*O(lg n)*	*|yük(sol) − yük(sağ)| ≤ 1*	arama ağırlıklı işler
kırmızı-siyah ağaç	*O(lg n)*	siyah yükseklik eşit	kütüphane eşlemeleri (`map`)
B-ağacı / B+ ağacı	$O(\log_m n)$ disk erişimi	yapraklar aynı düzeyde	veritabanı, dosya sistemi
splay ağacı	*O(lg n)* amortize	yok; erişilen düğüm köke çıkar	sık erişilen öğeler

Tablo 11.10: Arama ağaçlarının karşılaştırması.

Seçim, işin niteliğine bağlıdır. Veri küçük ve rastgele sıradaysa sıradan ikili arama ağacı yeter ve en sade koddur. Arama sayısı güncelleme sayısından çok büyükse AVL ağacının daha sıkı dengesi aramaları kısaltır. Ekleme ve silme sıksa kırmızı-siyah ağacın az döndürmesi öne çıkar. Veri belleğe sığmıyorsa B-ağacı neredeyse tek seçenektir. Erişimlerin küçük bir kısmı birkaç değere yoğunlaşıyorsa splay ağacı bu değerleri köke yakın tutar. Tablodaki splay ağacı satırında geçen “amortize” sözcüğü, tek bir işlemin değil, uzun bir işlem dizisinin ortalama maliyetini anlatır.

11.7 Sınav soruları

Sunumun son bölümü, önceki yıllarda sorulmuş üç sınav sorusunu çözer. Üçü de aynı beceriyi ölçer: küçük, iyi tanımlı yordamları birleştirerek doğru ve verimli kod yazmak ve verilen bir kodun ne kadar iş yaptığını hesaplamak.

11.7.1 Bir öğenin ağırlığı

Soru şudur. Değerleri tekrarsız bir ikili ağaçta (arama ağacı olması gerekmez) `x` değerini içeren öğenin **ağırlığı**, o öğeden uç düğümlere giden yolların en uzunudur. `x` ağaçta yoksa ya da ağaç boşsa ağırlık 0'dır. `int AgirlikBul(agac_gost L, int x)` yordamı küresel değişken kullanılmadan yazılmalıdır.

Tanım dikkatle okunursa ağırlığın, `x`'i içeren düğümün kök olduğu alt ağacın derinliği olduğu görülür. Problem iki iyi bilinen işin bileşimidir: önce düğümü bulmak, sonra onun derinliğini hesaplamak. Ağaç sıralı olmadığı için arama, her ikili ağaçta çalışan yordamla yapılır.

```
int AgirlikBul(agac_gost L, int x) {
  /* agacta_ara: x'i içeren öğe ya da NULL */
  agac_gost agacta_ara(agac_gost L, int x);
  int derinlik(agac_gost L);
  return derinlik(agacta_ara(L, x));
}
agac_gost agacta_ara(agac_gost L, int x) {
  if (!L) return NULL;
  if (L->veri == x) return L;
  agac_gost T = agacta_ara(L->solcocuk, x);
  return T ? T : agacta_ara(L->sagcocuk, x);
}
int derinlik(agac_gost L) {
  int max(int a, int b);  /* a,b'den küçük olmayan */
  return L ? max(derinlik(L->solcocuk),
                 derinlik(L->sagcocuk)) + 1 : 0;
}
```

Çözümün güzelliği üçüncü satırdadır: `derinlik(agacta_ara(L, x))`. Değer bulunamazsa `agacta_ara` `NULL` döndürür. `derinlik(NULL)` da zaten 0'dır. Böylece "bulunamadı" durumu ayrıca ele alınmaz; iki yordamın tanımları onu kendiliğinden çözer. `L == NULL` durumu da aynı yolla 0 verir.

Kodun ilk iki satırı, yordam içinde bildirilmiş **yerel prototiplerdir**. Bu bildirimler `agacta_ara` ve `derinlik` adlarını yalnız `AgirlikBul`'un içinde görünür kılar. Yordamlar dosyada `AgirlikBul`'dan sonra tanımlandığı için bu bildirimler gereklidir. `derinlik` de aynı yolla `max` fonksiyonunu bildirir. Burada makro değil fonksiyon kullanılması bilinçlidir. Makro, derinlik yordamının çağrı sayısını üstel yapardı.

Maliyet iki parçanın toplamıdır. Arama en kötü durumda bütün düğümlere bakar: $O(n)$. Derinlik yalnız bulunan alt ağacı gezer; o da en çok n düğümdür. Toplam $O(n)$'dir. Sürücü programda kökü 2, çocukları 10 ve 20 olan ağaçta ağırlıklar 2, 1, 1 ve `x = 30` için 0 çıktı. Silme örneklerindeki ağaçta 40'ın ağırlığı 6, 60'ınki 5, 50'ninki 4, 56'nınki 1 bulundu.

11.7.2 Yordam kaç kez çağrılır?

İkinci soru, aynı problemi çözmeye çalışan kötü bir kodu inceler. Kodda `agirlik(L)`, `L`'nin kök olduğu alt ağacın derinliğini veren bir yordamdır.

```
int AgirlikBul(oge_gost L, int x) {
  if (!L) return 0;
  else if (L->veri == x) return agirlik(L);
  if (AgirlikBul(L->solcocuk, x) >
      AgirlikBul(L->sagcocuk, x))
    AgirlikBul(L->solcocuk, x);
  if (AgirlikBul(L->sagcocuk, x) >
      AgirlikBul(L->solcocuk, x))
    AgirlikBul(L->sagcocuk, x);
  return 0;
}
```

Soru iki bölümlüdür. Ağaç kökü 2, sol çocuğu 10, sağ çocuğu 20 olan üç düğümlü ağaçtır. Birinci bölümde `x = 20` ve `x = 30` için yordamın kaç kez çağrıldığı sorulur. İkinci bölümde en kötü durum için yineleme bağıntısı yazılıp çözülmelidir.

Önce bir yaprağın, ör. 10'un, `x`'i içermediği durumda ne yaptığına bakılsın. Yaprak `NULL` değildir ve değeri `x`'e eşit değildir. Birinci `if` iki alt çağrı yapar, ikinci `if` iki alt çağrı daha yapar. Dördü de `NULL` ile yapılır ve 0 döndürür. $0 > 0$ yanlış olduğu için koşulların içindeki çağrılar yapılmaz. Bir yaprak çağrısı kendisiyle birlikte $1 + 4 = 5$ çağrı eder.

`x = 30` ağaçta yoktur. Kök çağrısı 10'u iki kez, 20'yi iki kez çağırır. Her biri 0 döndürür ve koşullu çağrılar yapılmaz. Toplam $1 + 4 \cdot 5 = 21$ çağrıdır. `x = 20`'de durum farklıdır. 20'ye yapılan çağrı ikinci satırda eşitliği bulur ve tek çağrıda ağırlığı (1) döndürür. Kökte birinci `if`, 10 için 5, 20 için 1 çağrı yapar; $0 > 1$ yanlıştır. İkinci `if` yine 1 + 5 çağrı yapar ve $1 > 0$ doğrudur. Koşulun içindeki `AgirlikBul(L->sagcocuk, x)` bir çağrı daha yapar. Toplam $1 + 6 + 6 + 1 = 14$'tür. Bütün bu emeğe karşın yordam her iki durumda da 0 döndürür. Koşulların içindeki çağrıların sonucu kullanılmaz ve son satır her zaman 0'dır. Kod hem yavaş hem yanlıştır: yalnız `x` kökteyse doğru cevap verir.

En kötü durumda `x` ağaçta yoktur ve bütün ağaç gezilir. Her çağrı her alt ağacını iki kez çağırır. Çarpık bir ağaçta bir alt ağaç $n - 1$ düğümlü, öteki boştur. Bağıntı $T(n) = 2T(n - 1) + c$ olur. Açılırsa $T(n) = 2^n T(0) + c(2^n - 1)$ bulunur, yani $\Theta(2^n)$. Dengeli bir ağaçta iki alt ağacın her biri yaklaşık $n/2$ düğümlüdür ve dördü de çağrılır: $T(n) = 4T(n/2) + c$. Ana teorem ya da açma yöntemiyle çözüm $\Theta(n^2)$'dir. Her düzeyde çağrı sayısı dört katına çıkarken problem yarıya iner ve $4^{\lg n} = n^2$'dir.

Ağaç	n	Çağrı sayısı	Oran
zincir	4	61	çağrı / 2^n = 3,81
zincir	12	16.381	3,999
zincir	20	4.194.301	4,000
dolu	255	87.381	çağrı / n^2 = 1,344
dolu	4.095	22.369.621	1,334

Tablo 11.11: Kötü `AgirlikBul` yordamının, `x` ağaçta yokken yaptığı çağrı sayısı (sayaçla ölçüldü).

Ölçümler bağıntıları doğrular. Zincirde çağrı sayısı tam $2^{n+2} - 3$'tür ve oran 4'e yaklaşır. Mükemmel ağaçta oran $4/3$'e yaklaşır. Doğru tasarım ise önceki kesimdeki çözümdür ve $O(n)$ sürer. 20 düğümlük bir zincirde kötü yordam dört milyondan fazla çağrı yapar. Doğru yordam aynı ağaçta 20 düğüme bakmak için birkaç düzine çağrıyla yetinir.

Sorunun dersi, bir özyinelemeli çağrının sonucunu iki kez hesaplamanın bedelidir. Her düzeyde iş ikiye katlanır ve katlar birbiriyle çarpılır. Bir alt çağrının sonucu birden çok kez gerekiyorsa bir yerel değişkende saklanmalıdır. Derinlik yordamındaki `sol` ve `sag` değişkenleri tam bu işi görür.

11.7.3 İki çağrının arasında SWAP

Üçüncü soru bir yordamın ağaç üzerindeki etkisini sorar. `SWAP(x, y, t)` makrosu `t` yardımıyla `x` ile `y`'nin değerlerini değiştirir: `((t) = (x), (x) = (y), (y) = (t))`.

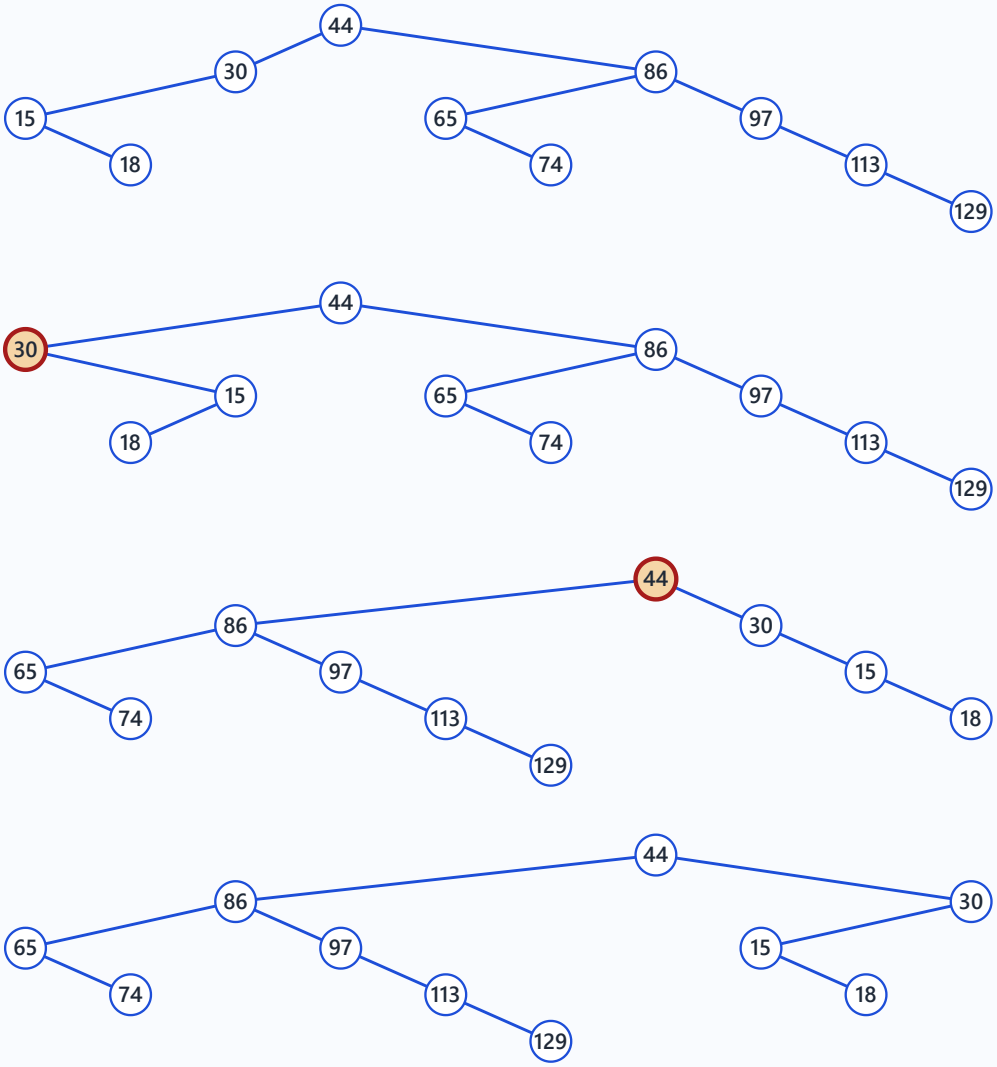
```
void yordam(agac_gost L) {
  agac_gost p;
  if (L) {
    yordam(L->solcocuk);
    SWAP(L->solcocuk, L->sagcocuk, p);
    yordam(L->sagcocuk);
  }
}
```

Soru şudur: {44, 30, 15, 18, 86, 97, 65, 113, 74, 129} değerleriyle kurulan ikili arama ağacı için yordam işletilsin. SWAP satırından her geçişte ağacın görünümü değişmişse ağaç yeniden çizilsin ve son hâli yorumlansın.

Başlangıç ağacında kök 44'tür. Solunda 30 vardır; 30'un solunda 15, 15'in sağında 18 durur. Kökün sağında 86 vardır. 86'nın solunda 65 ve onun sağında 74; sağında 97, 113 ve 129'dan oluşan sağa uzanan bir zincir bulunur.

Yordam bir düğümde önce sol alt ağacı işler, sonra iki çocuğu değiştirir, sonra *yeni* sağ çocuğu işler. Ama yeni sağ çocuk, az önce işlenmiş olan eski sol çocuktur. Eski sağ çocuk ise sola geçmiştir ve hiç işlenmez. Adımlar şöyledir.

1. `yordam(44)` → `yordam(30)` → `yordam(15)`. 15'in solu boştur. **SWAP 15**: 18 sola geçer.
2. 15'in yeni sağı boştur. 30'a dönülür. **SWAP 30**: 15 alt ağacı sağa geçer.
3. 30'un yeni sağı olan 15 yeniden işlenir. Önce solu (18) işlenir; 18'de SWAP yapılır ama iki çocuk da boş olduğu için görünüm değişmez. **SWAP 15**: 18 yeniden sağa geçer. Sağdaki 18 bir kez daha işlenir; görünüm yine değişmez.
4. 44'e dönülür. **SWAP 44**: 86 alt ağacı sola, 30 alt ağacı sağa geçer.
5. 44'ün yeni sağı olan 30 yeniden işlenir. Solu boştur. **SWAP 30**: 15 alt ağacı sola döner. Yeni sağ boştur ve yordam biter.



Şekil 11.18: SWAP iki özyinelemeli çağrının arasında. Yukarıdan aşağı: başlangıç ağacı; 30'daki ilk SWAP'tan sonra; 44'teki SWAP'tan sonra; son hâl. Kökün iki alt ağacı yer değiştirmiştir, ama alt ağaçların içi başlangıçtaki gibidir.

Görünümün değiştiği SWAP'lar sırasıyla 15, 30, 15, 44 ve 30'dadır. 18'deki iki SWAP görünümü değiştirmez. Son ağaç 44(86(65(−, 74), 97(−, 113(−, 129))), 30(15(−, 18), −)) biçimindedir. Yalnız kökün iki alt ağacı yer değiştirmiştir. 30 alt ağacı iki kez işlenmiş, her değişiklik ikinci işlemde geri alınmıştır. 86 alt ağacı ise hiç işlenmemiştir. Sonuç ne ağacın kendisi ne de aynasıdır, artık bir ikili arama ağacı da değildir. Sürücü program SWAP satırına bir izleme çağrısı ekleyerek her adımı yazdırdı; sıra ve son ağaç yukarıdaki gibidir.

SWAP iki özyinelemeli çağrının önüne ya da arkasına yazılsaydı ağacın tam aynası elde edilirdi. Önce yazıldığında çocuklar değiştirilir, sonra iki alt ağaç ayrı ayrı aynalanır. Sona yazıldığında önce iki alt ağaç aynalanır, sonra yerleri değiştirilir. Her iki durumda da her alt ağaç tam bir kez işlenir. Aynalanmış ikili arama ağacında kök ortada gezinme değerleri azalan sırada verir.

11.8 Karmaşıklık özeti

Tablo 11.12 bu notta yazılan bütün yordamların maliyetini bir arada verir. *h* ağacın derinliğidir; dengeli ağaçta *O(lg n)*, çarpık ağaçta *n* olur.

İşlem	En iyi / dengeli	En kötü (çarpık)	Ek bellek
`arama_agacinda_ara` (yinelemeli)	*O(lg n)*	*O(n)*	*O(1)*
`ara` (özyinelemeli)	*O(lg n)*	*O(n)*	*O(h)* yığın

İşlem	En iyi / dengeli	En kötü (çarpık)	Ek bellek
`arama_agacina_ekle`	$\Omega(\lg n)$	$O(n)$	$O(1)$
`sil`	$O(\lg n)$	$O(n)$	$O(1)$
`derinlik`	$\Theta(n)$	$\Theta(n)$	$O(h)$
genel ikili ağaçta ara	$O(n)$	$O(n)$	$O(h)$
AVL arama / ekleme / silme	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$ özyineleme
tek döndürme	$O(1)$	$O(1)$	—

Tablo 11.12: Arama ağacı işlemlerinin zaman ve ek bellek maliyeti.

Tablonun özü şudur: ikili arama ağacında arama, ekleme ve silme yolu izler ve $O(h)$ sürer. Ağacın her düğümüne bakması gereken işlemler (derinlik, genel ağaçta arama) $O(n)$'dir. Dengeli ağaçlar h'yi $O(\lg n)$'de tutarak yol izleyen işlemlerin en kötü durumunu da logaritmik yapar. Ekleme satırındaki $\Omega(\lg n)$, eklemenin hiçbir ağaçta $\lg n$'den hızlı olamayacağını anlatır.

11.9 Sık yapılan hatalar

1. **Kök işaretçisini güncelleyememek.** Ekleme yordamı `agac_gost L` alırsa boş ağaca yapılan ilk ekleme çağıranın kökünü değiştirmez. Ağaç boş kalır, düğüm sızar. Ya `agac_gost *L` ile adres gönderilmeli ya da yeni kök döndürülüp atanmalıdır.
2. **Arka işaretçiyi tutmamak.** Yinelemeli eklemede `p` `NULL` 'a ulaştığında yeni düğümün bağlanacağı ebeveyn artık bilinmez. Bir adım geriden gelen `q` ya da işaretçinin işaretçisi kullanılmalıdır.
3. **Alt çağrının sonucunu döndürmemek.** Özyinelemeli aramada `return ara(...)` yerine yalnız `ara(...)` yazılırsa bulunan adres kaybolur.
4. **Taşınan öğenin çocuğunu unutmak.** İki çocuklu silmede sol alt ağacın en büyüğünün sol çocuğu (örnekte 56) ebeveynine bağlanmazsa ağaçtan kopar.
5. **Küresel değişken ve gereksiz ek parametre.** Derinlik için küresel bir sayaç ya da "ilk çağrıda 0 geçirilmeli" türünden bir parametre, yordamı kırılgan ve yeniden kullanılamaz yapar.
6. **Makroya özyinelemeli çağrı vermek.** `MAX(derinlik(...), derinlik(...))` çağrılardan birini iki kez işletir ve süreyi üstele çıkarır.
7. **Serbest bıraktıktan sonra kullanmak.** Silmede bağ güncellenmeden önce düğüm `free` edilirse, sonraki satır serbest bırakılmış belleği okur.
8. **Denetimi yalnız çocuklarla yapmak.** Bir ağacın ikili arama ağacı olup olmadığı yalnız ebeveyn–çocuk karşılaştırmasıyla anlaşılmaz; her düğüme kökten gelen aralık uygulanmalıdır.

11.10 Alıştırmalar ve çözümleri

Bu kesim sunumdaki altı alıştırmayı çözer. Kodlu çözümler sürücü programlarla derlenip çalıştırıldı.

1. Ekleme ve kök önde sıra. Boş bir ağaca sırasıyla 50, 30, 70, 20, 40, 60, 80, 35 eklensin. 50 kök olur. 30 ve 70 onun çocuklarıdır. 20 ile 40, 30'un; 60 ile 80, 70'in çocuklarıdır. 35, 50 → 30 → 40 yolunu izleyip 40'ın soluna girer. Ağaç 50(30(20, 40(35, –)), 70(60, 80)) biçimindedir ve derinliği 4'tür. Kök önde sıra 50 30 20 40 35 70 60 80'dir.

2. Aynı ağaçtan 30'u iki yöntemle silmek. 30'un iki çocuğu vardır. Birinci yöntemde sol alt ağacın en büyüğü aranır. 30'un sol alt ağacı tek düğümdür (20); en büyük 20'dir. 30'un yerine 20 yazılır ve eski 20 düğümü silinir. Ağaç 50(20(–, 40(35, –)), 70(60, 80)) olur. 50 → 20 → 40 → 35 yolu kaldığı için derinlik 4'tür. İkinci yöntemde sağ alt ağacın en küçüğü aranır: 40'tan sola, 35'e gidilir. 30'un yerine 35 yazılır ve 40'ın solu `NULL` olur. Ağaç 50(35(20, 40), 70(60, 80)) biçimindedir ve derinliği 3'e iner. Bu örnekte ikinci yöntem daha dengeli bir ağaç verir.

3. En küçük ve en büyük değer. En küçük değer, kökten sürekli sola gidilerek bulunur; sol çocuğu olmayan ilk düğümdedir. En büyük değer de sürekli sağa gidilerek bulunur.

```
agac_gost en_kucuk(agac_gost L) {
  if (L)
    while (L->solcocuk) L = L->solcocuk;
  return L;                /* boş ağaçta NULL */
}
agac_gost en_buyuk(agac_gost L) {
  if (L)
    while (L->sagcocuk) L = L->sagcocuk;
  return L;
}
```

İki yordam da yalnız bir yol izler ve *O(h)* sürer. Boş ağaçta `NULL` döndürür. Silme örneklerindeki ağaçta en küçük değer 10, en büyük değer 73 çıktı.

4. İkili arama ağacı denetimi. Her düğüme izin verilen (alt, üst) aralığı parametre olarak geçirilir. Sola inerken üst sınır, sağa inerken alt sınır düğümün değeri olur.

```
#include <limits.h>
/* L'deki her değer alt < değer < ust olmalı */
boolean aralikta(agac_gost L, long long alt, long long ust) {
  if (!L) return TRUE;
  if (L->veri <= alt || L->veri >= ust) return FALSE;
  return aralikta(L->solcocuk, alt, L->veri) &&
         aralikta(L->sagcocuk, L->veri, ust);
}
boolean bst_mi(agac_gost L) {
  return aralikta(L, (long long)INT_MIN - 1, (long long)INT_MAX + 1);
}
```

Sınırlar `long long` tipindedir. Böylece `INT_MIN` ve `INT_MAX` değerleri de ağaçta bulunabilir; başlangıç sınırları bunların bir dışındadır. `long` kullanılsaydı, `long` 'un 32 bit olduğu Windows'ta `INT_MAX + 1` taşardı. Her düğüme bir kez bakıldığı için süre *O(n)*, özyineleme derinliği *O(h)*'dir. Yordam silme örneklerindeki ağacı ve onun dört silme sonucunu geçerli buldu. Kökü 15, çocukları 10 ve 18 olan bir ağaçta 18'in soluna 12 konduğunda ise `FALSE` döndürdü. 12, 18'den küçüktür; yalnız çocuklarla karşılaştırma bu ağacı geçerli sayardı. Ama 12, 15'in sağ alt ağacındadır ve 15'ten küçük olamaz.

5. AVL ağacına 10, 20, 30, 25, 28. 10 ve 20 eklenince 10'un faktörü −1 olur. 30 eklenince 10'un faktörü −2'dir ve yeni değer sağın sağındadır (RR). 10 sola döndürülür ve 20 kök olur: 20(10, 30). 25, 30'un soluna girer; faktörler −1..+1 aralığındadır. 28, 25'in sağına girer. 30'un faktörü +2 olur ve yeni değer solun sağındadır (LR). Önce 25 sola döndürülür: 28, 25'in yerine çıkar. Sonra 30 sağa döndürülür: 28, 25 ile 30'un ebeveyni olur. Son ağaç 20(10, 28(25, 30))'dur. Sürücü program aynı iki döndürmeyi raporladı: 30'da `sol_dondur(10)`, 28'de `sol_dondur(25)` ve `sag_dondur(30)`.

6. `x`'ten küçük anahtar sayısı. En basit çözüm ağacı kök ortada gezmek ve `x`'e ulaşana kadar saymaktır; süre *O(n)*'dir. İkili arama ağacının düzeni kullanılarak bazı alt ağaçlar atlanabilir. Kök `x`'ten küçük değilse sağ alt ağaç tümüyle büyüktür ve sayım yalnız solda sürer. Kök `x`'ten küçükse kökün kendisi ve bütün sol alt ağacı küçüktür. Bunlar sayılır ve sayım sağda sürer.

```
int dugum_say(agac_gost L) {
  return L ? 1 + dugum_say(L->solcocuk) + dugum_say(L->sagcocuk) : 0;
}
/* x'ten küçük değerlerin sayısı */
int kucuk_say(agac_gost L, int x) {
  if (!L) return 0;
  if (L->veri >= x)                /* kök ve sağı büyük */
    return kucuk_say(L->solcocuk, x);
  return 1 + dugum_say(L->solcocuk) /* kök ve solu küçük */
           + kucuk_say(L->sagcocuk, x);
}
```

Bu sürümde `dugum_say` hâlâ sol alt ağaçları baştan sona gezer. Bu yüzden en kötü durum yine *O(n)*'dir. Her düğümde alt ağacının boyutu bir alanda saklanırsa `dugum_say` *O(1)*'e iner ve sayım tek bir yol boyunca, yani *O(h)*'de yapılır. Bu alan her ekleme ve silmede yol üzerindeki düğümlerde güncellenmelidir. Örnek ağaçta 55'ten küçük 7, 41'den küçük 4 değer bulundu.

11.11 Özet

- **İkili arama ağacında** değerler tektir; her düğümün sol alt ağacındaki bütün değerler ondan küçük, sağ alt ağacındakiler büyüktür ve alt ağaçlar da ikili arama ağacıdır. Kök ortada gezinme değerleri artan sırada verir.
- **Ekleme**, değeri kökten başlayarak arar ve ilk `NULL`'ın yerine koyar. Yinelemeli sürüm bir adım geriden gelen `q` işaretçisini, boş ağaç için `agac_gost *L` parametresini kullanır. Ekleme sırası ağacın biçimini belirler; sıralı veri çarpık ağaç kurar.
- **Arama** her karşılaştırmada bir alt ağacı eler ve $O(h)$ sürer. Özyinelemeli sürüm kuyruk özyinelemesidir ve alt çağrının sonucunu aynen döndürür. Sıralı olmayan ağaçta iki yana da bakmak gerekir ve arama $O(n)$ olur.
- **Derinlik** $MAX(sol, sağ) + 1$ ile özyinelemeli hesaplanır; $2n + 1$ çağrı yapar ve $\Theta(n)$'dir. Küresel değişken ve gizli başlangıç değeri taşıyan parametre kullanılmaz. Makroya özyinelemeli çağrı verilirse süre üstel olur.
- **Silmede** yaprak ve tek çocuklu düğüm kolaydır. İki çocuklu düğümde öncülün (ya da ardılın) değeri kopyalanır ve onun düğümü silinir; bağ işaretçinin işaretçisiyle tek satırda güncellenir. Silme yöntemi ağacın derinliğini değiştirebilir.
- **Denge.** Rastgele veride ortalama derinlik $\Theta(\lg n)$'dir, ama sıralı veride n'ye çıkar. **AVL ağacı** her düğümde denge faktörünü $-1..+1$ aralığında tutar; yüksekliği en çok yaklaşık $1{,}44 \lg(n + 2)$'dir.
- AVL eklemesinde dengesizlik LL ve RR durumlarında tek, LR ve RL durumlarında çift döndürmeyle düzeltilir. Bir döndürme yalnız iki işaretçiyi değiştirir ve $O(1)$'dir; eklemede en çok bir döndürme yeter.
- **Kırmızı-siyah ağaç** daha gevşek bir dengeyle az döndürme yapar ve kütüphanelerin eşleme yapılarında kullanılır. **B-ağacı** çok anahtarlı düğümleriyle disk erişimini azaltır ve veritabanlarında kullanılır.

11.12 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

ikili arama ağacı *binary search tree (BST)* · anahtar *key* · alt ağaç *subtree* · yaprak *leaf* · derinlik *depth* · düzey *level* · kök önde *preorder* · kök ortada *inorder* · kök sonda *postorder* · düzey sırası *level-order* · çarpık ağaç *skewed tree* · dengeli ağaç *balanced tree* · kendini dengeleyen ağaç *self-balancing tree* · adresle çağrı *call by reference* · işaretçiye işaretçi *pointer to pointer* · küresel değişken *global variable* · makro *macro* · öncül *predecessor* · ardıl *successor* · bellek sızıntısı *memory leak* · AVL ağacı *AVL tree* · denge faktörü *balance factor* · döndürme *rotation* · çift döndürme *double rotation* · kırmızı-siyah ağaç *red-black tree* · B-ağacı *B-tree* · splay ağacı *splay tree* · yineleme bağıntısı *recurrence relation* · amortize analiz *amortized analysis*

11.13 Kendini Dene

1. Boş bir ikili arama ağacına sırasıyla 8, 3, 10, 1, 6, 14, 4, 7, 13 ekleniyor. Ağacın derinliği nedir? Kök önde ve kök ortada gezinme sıraları nelerdir?

   Cevap

   Ağaç 8(3(1, 6(4, 7)), 10(–, 14(13, –))) biçimindedir. En uzun yollar 8 → 3 → 6 → 4 ile 8 → 10 → 14 → 13'tür; derinlik 4'tür. Kök önde sıra 8 3 1 6 4 7 10 14 13, kök ortada sıra 1 3 4 6 7 8 10 13 14'tür.

2. Aynı ağaçta `arama_agacinda_ara` ile 7 ve 5 aranıyor. Her aramada hangi düğümlere bakılır ve sonuç ne olur?

   Cevap

   7 için 8, 3, 6, 7 düğümlerine bakılır ve 7'nin adresi döner. 5 için 8, 3, 6, 4 düğümlerine bakılır; 4'ün sağ çocuğu `NULL` olduğu için arama `NULL` döndürür.

3. Aynı ağaca sunumdaki `sil` yordamı uygulanıyor: önce yalnız 3 siliniyor (ayrı bir denemede de yalnız kök 8). Her durumda ağaç nasıl

değişir?

Cevap

3'ün iki çocuğu vardır. Sol alt ağacının en büyüğü 1'dir (1'in sağı yok). 3'ün yerine 1 yazılır, eski 1 düğümü silinir: 8(1(–, 6(4, 7)), 10(–, 14(13, –))). Kök 8 silinirse sol alt ağacın en büyüğü 3 → 6 → 7 yoluyla 7 bulunur. Kökün veri alanına 7 yazılır, 6'nın sağı `NULL` olur: 7(3(1, 6(4, –)), 10(–, 14(13, –))).

4. Bir ikili arama ağacının kök önde gezinmesi 30 20 10 25 40 35 50'dir. Kök sonda gezinmesi nedir?

Cevap

Ağaç 30(20(10, 25), 40(35, 50))'dir. Kök sonda gezinme 10 25 20 35 50 40 30'dur.

5. Boş bir AVL ağacına 1, 2, 3, 4, 5, 6, 7 sırayla ekleniyor. Hangi eklemelerde döndürme gerekir ve son ağaç nedir?

Cevap

Bütün dengesizlikler RR durumudur ve tek sola döndürmeyle düzelir. 3 eklenince 1'de, 5 eklenince 3'te, 6 eklenince kök 2'de, 7 eklenince 5'te sola döndürme yapılır; toplam dört döndürme. Son ağaç tam dengelidir: 4(2(1, 3), 6(5, 7)) ve yüksekliği 3'tür.

6. Yüksekliği 5 olan bir AVL ağacında en az kaç düğüm bulunur?

Cevap

$N(h) = N(h-1) + N(h-2) + 1$, $N(1) = 1$, $N(2) = 2$ bağıntısından $N(3) = 4$, $N(4) = 7$, $N(5) = 12$ bulunur. En az 12 düğüm gerekir.

7. Tek satırlık `derinlik` yordamı `MAX` makrosuyla yazılmıştır. Üç düğümlük sağa uzanan bir zincirde yordam toplam kaç kez çağrılır? Yerel değişkenli sürüm kaç kez çağrılır?

Cevap

Makrolu sürümde $T(0) = 1$ ve $T(n) = 2T(n-1) + 2$'dir: $T(1) = 4$, $T(2) = 10$, $T(3) = 22$. Yerel değişkenli sürüm $2n + 1 = 7$ çağrı yapar. İki değer de sayaçla ölçüldü.

8. `arama_agacina_ekle` yordamının parametresi `agac_gost *L` yerine `agac_gost L` yapılır ve son satırlar buna göre düzeltilirse, boş ağaca 5 ve 7 eklemenin sonucu ne olur?

Cevap

İlk eklemede yeni düğüm yalnız yordamın yerel `L` kopyasına atanır. Çağıranın kökü `NULL` kalır ve düğüm hiçbir yerden gösterilmediği için sızar. İkinci ekleme de ağacı boş bulur ve aynı şey olur. Ağaç boş kalır, iki düğüm sızar.

9. SWAP satırı `yordam`'da iki özyinelemeli çağrının önüne alınırsa, kökü 2, sol çocuğu 1, sağ çocuğu 3, 1'in sol çocuğu 0 olan ağaç nasıl değişir?

Cevap

Önce kökte çocuklar değişir: 2(3, 1(0, –)). Sonra sol alt ağaç (3) ve sağ alt ağaç (1) ayrı ayrı işlenir; 1'de 0 sağa geçer. Son ağaç 2(3, 1(–, 0))'dır ve ilk ağacın tam aynasıdır. Kök ortada gezinme 3 2 1 0, yani azalan sıradır.

11.14 İleri okuma

- E. Horowitz, S. Sahni ve S. Anderson-Freed, *Fundamentals of Data Structures in C* (2. baskı), Kesim 5.7 ve 10.2: bu notun izlediği ikili arama ağacı yordamlarının ve AVL ağacının kaynağıdır.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (3. baskı), Bölüm 12, 13 ve 18: ikili arama ağacını, rastgele kurulan ağacın ortalama derinliğini, kırmızı-siyah ağacı ve B-ağacını kanıtlarıyla verir.
- M. A. Weiss, *Data Structures and Algorithm Analysis in C* (2. baskı), Kesim 4.4: notta kullanılan AVL ekleme örneğini ve döndürme kodlarını içerir; splay ağacını da tanıtır.

- R. Sedgewick, *Algorithms in C*, Kısım 1–4, Bölüm 12–13: ikili arama ağacının ortalama durum analizini ve dengeli ağaçların farklı türlerini karşılaştırır.

11.15 Kaynaklar

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, Kesim 5.7 (ikili arama ağacı) ve 10.2 (AVL ağacı).
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3. baskı, MIT Press, Bölüm 12 (ikili arama ağaçları), 13 (kırmızı-siyah ağaçlar), 18 (B-ağaçları).
3. M. A. Weiss, *Data Structures and Algorithm Analysis in C*, Kesim 4.4 (AVL ağaçları).
4. R. Sedgewick, *Algorithms in C*, Kısım 1–4, Bölüm 12–13.

KISIM III · BÖLÜM 12

Kılavuzlu İkili Ağaçlar

Bağlı gösterimle kurulan bir ikili ağaçta işaretçi alanlarının yarıdan fazlası boş kalır. Kılavuzlu ikili ağaç bu boş alanları değerlendirir. Boş bir sol işaretçi, düğümden kök ortada sırada hemen önce gelen düğümü gösterir; boş bir sağ işaretçi de hemen sonra geleni. Böylece ağaç, yığın ya da özyineleme kullanmadan, tek bir işaretçiyle baştan sona gezilebilir. Bu not önce boş işaretçilerin sayısını kanıtlar, sonra kılavuzların nasıl kurulduğunu, gerçek bağlardan nasıl ayırt edildiğini ve başlık düğümünün rolünü anlatır. Ardından ardıl bulma, yığınsız kök ortada ve kök önde gezinme, bayrak alanı gerektirmeyen Morris gezinmesi ve kılavuzları bozmadan düğüm ekleyen `saga_ekle` ile `sola_ekle` yordamları gelir. Her yordamın C kodu, satır satır anlatımı, kod izlemesi ve karmaşıklık analizi verilir.

Kod (C): `github.com/sercankulcu/data-structures-and-algorithms-c/Bolum12`

ÖĞRENME ÇIKTILARI

Boş işaretçileri saymak
n düğümlü bir ikili ağaçta n + 1 boş işaretçi bulunduğunu kanıtlamak.

Kılavuz kurmak
Boş sol işaretçiyi kök ortada öncüle, boş sağ işaretçiyi kök ortada ardıla bağlama kuralını bir ağaca elle uygulamak.

Yapıyı eksiksiz tanımlamak
`solKilavuz` ve `sagKilavuz` bayraklarıyla, başlık düğümüyle birlikte kılavuzlu ağacın C yapısını kurmak.

Yığınsız gezinmek
`kokortada_ardil_bul` ile O(1) ek bellekle kök ortada ve kök önde gezinmek; Morris gezinmesini adım adım izlemek.

Ekleme yapmak
`saga_ekle` yordamının iki durumunu tek yordamda birleştirmek ve `sola_ekle`'yi ondan türetmek.

İyi kod yazmak
Açık tanımlı yordam, niyeti söyleyen açıklama satırı ve NULL denetiminin doğru yeri üzerine karar vermek.

12.1 Boş işaretçiler ve kılavuz fikri

Bağlı gösterimde bir ikili ağacın her düğümü üç alandan oluşur: sol alt ağacı gösteren işaretçi, veri ve sağ alt ağacı gösteren işaretçi. C'de bu düğüm kendine başvuran bir yapıyla (*self-referential struct*) tanımlanır:

```c
typedef struct agac *agac_gost;
struct agac {
    agac_gost solcocuk;   /* sol alt ağaç */
    char      veri;
    agac_gost sagcocuk;   /* sağ alt ağaç */
};
```

Çocuğu olmayan bir yöndeki işaretçiye `NULL` yazılır. `NULL`, hiçbir nesneyi göstermeyen **boş işaretçidir** (*null pointer*). Bir yaprağın iki işaretçisi de boştur. Tek çocuklu bir düğümün işaretçilerinden biri boştur. Bu bölümün sorusu basittir: bir ağaçta kaç işaretçi boş kalır ve bu alanlar bir işe yarayabilir mi?

12.1.1 n düğümde n + 1 boş işaretçi

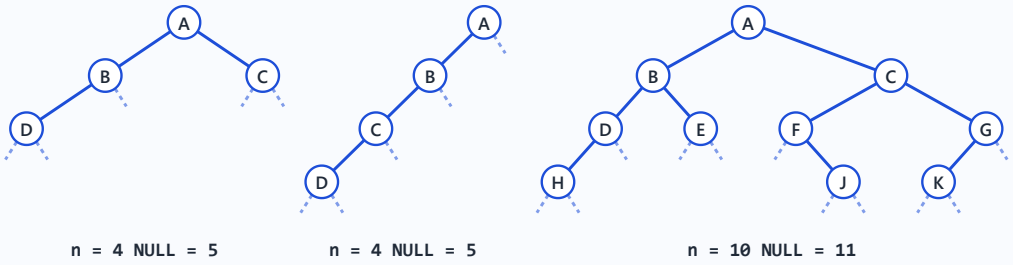
Ağacın biçimi ne olursa olsun boş işaretçilerin sayısı yalnız düğüm sayısına bağlıdır. Bunu bir sayma kanıtı gösterir.

Teorem 12.1 (Boş işaretçi sayısı)

Bağlı gösterimle saklanan, $n \geq 1$ düğümlü her ikili ağaçta tam olarak $n + 1$ işaretçi alanı `NULL` 'dır.

İspat. Her düğümde iki işaretçi alanı vardır; ağaçta toplam $2n$ alan bulunur. Kök dışındaki her düğüme ebeveyninden tam olarak bir işaretçi gelir. Bir düğüme iki farklı alandan işaretçi gelemez, çünkü her düğümün tek bir ebeveyni vardır. Köke ise ağacın içinden hiçbir işaretçi gelmez; kök, ağacın dışındaki bir değişkenle tutulur. Öyleyse dolu alanların sayısı $n - 1$'dir. Geri kalan alanların sayısı $2n - (n - 1) = n + 1$ olur. ∎

Kanıt, ağacın dengeli ya da çarpık olmasına hiç bakmaz. Şekil 12.1 bunu üç örnekle gösterir. Soldaki iki ağaç 4 düğümlüdür ve biçimleri çok farklıdır; ikisinde de 5 işaretçi boştur. Sağdaki 10 düğümlü ağaçta 11 boş işaretçi vardır.



Şekil 12.1: Üç ikili ağaç ve boş işaretçileri. Kesikli kısa çizgiler `NULL` alanlardır. 4 düğümlü iki ağaçta 5, 10 düğümlü ağaçta 11 boş işaretçi vardır.

Oran dikkat çekicidir. $2n$ alanın $n + 1$'i, yani yarıdan fazlası hiçbir bilgi taşımaz. 64 bitlik bir makinede bir işaretçi 8 bayttır; bir milyon düğümlü bir ağaçta yaklaşık 8 MB bellek boş işaretçilere gider. Bu alanları atmak mümkün değildir, çünkü hangi düğümün hangi yönde çocuğu olacağı önceden bilinmez. Ama bu alanlara anlamlı bir değer yazılabilir.

Örnek 12.1

Bu notun ana örneği dokuz düğümlü bir ağaçtır. A köktür; B ve C onun çocuklarıdır. D ve E, B'nin; F ve G, C'nin; H ve I, D'nin çocuklarıdır. Ağaçta $2 \cdot 9 = 18$ işaretçi alanı vardır. Kök dışındaki sekiz düğüme sekiz işaretçi gelir. Boş alanlar $18 - 8 = 10$ tanedir: H, I, E, F ve G yapraklarının ikişer alanı. Bu sayı $n + 1 = 10$ ile örtüşür.

Sık yapılan hata

Kökü gösteren işaretçiyi de "dolu işaretçi" sayıp n dolu alan bulmak. Kökü gösteren değişken (ör. `agac_gost kok`) ağacın düğümlerinden birinin alanı değildir. Düğümlerin içindeki dolu alanlar yalnız $n - 1$ tanedir.

12.1.2 Perlis ve Thornton'un fikri

A. J. Perlis ve C. Thornton 1960'ta boş işaretçilerin ağaçtaki başka düğümleri göstermek için kullanılabileceğini önerdiler. Bu işaretçilere **kılavuz** (*thread*) adını verdiler. Kılavuz, düğümün gerçek bir çocuğu değildir; gezinmede hangi düğüme geçileceğini gösteren bir yol işaretidir.

Fikrin değeri gezinmenin ek belleğinde görülür. Bir ikili ağacı kök ortada (*inorder*) gezmenin iki bilinen yolu vardır. Özyinelemeli yordam her düğüm için bir çağrı açar. Bir yol boyunca açık kalan çağrıların sayısı ağacın yüksekliği h kadardır; çağrı yığını (*call stack*) bu yüzden $O(h)$ bellek kullanır. Özyinelemesiz (yinelemeli) yordam ise aynı işi açık bir yığınla yapar: sola inerken geçtiği düğümleri yığına koyar, sonra onlara geri döner. Bu yığın da $O(h)$ yer tutar.

Dengeli bir ağaçta h yaklaşık $\log_2 n$'dir ve bu bedel küçüktür. Ama **çarpık ağaçta** (*skewed tree*) bütün düğümler tek bir yöne dizilir ve $h = n$ olur. Bir milyon düğümlü sola çarpık bir ağaçta özyinelemeli gezinme bir milyon çağrıyı aynı anda açık tutar. Varsayılan çağrı yığını çoğu sistemde bu kadar derinliği kaldırmaz ve program çöker.

Kılavuzlu ağaçta durum farklıdır. Her düğümden kök ortada sırada sonraki düğüme, yani **ardıla**, ya doğrudan bir kılavuzla ya da birkaç gerçek bağ izlenerek gidilir. Nereye dönüleceği ağacın içinde yazılı olduğu için hiçbir şeyi hatırlamak gerekmez. Gezinme tek bir işaretçi değişkeniyle yapılır ve ek bellek $O(1)$ olur. Bunun bedeli de vardır: her işaretçi alanı için onun gerçek çocuk mu yoksa kılavuz mu olduğu-

nu söyleyen birer bitlik bilgi tutmak gerekir.

Yöntem	Zaman	Ek bellek	Ne hatırlanır
Özyinelemeli	$O(n)$	$O(h)$	Açık çağrılar (çağrı yığını)
Özyinelemesiz, yığınlı	$O(n)$	$O(h)$	Sola inerken geçilen düğümler
Kılavuzlu	$O(n)$	$O(1)$	Hiçbir şey; dönüş yolu ağaçta yazılı

Tablo 12.1: Kök ortada gezinmenin üç yolu. h ağacın yüksekliğidir; çarpık ağaçta $h = n$.

12.2 Kılavuzları kurmak

Bir boş işaretçinin hangi düğümü göstereceği keyfî değildir. Kural, kök ortada gezinme sırasına göre konur. Bu yüzden önce o sıradaki "önceki" ve "sonraki" düğüm kavramları kesinleştirilmelidir.

12.2.1 Kök ortada öncül ve ardıl

Kök ortada gezinme bir düğümde önce sol alt ağacı, sonra düğümün kendisini, en son sağ alt ağacı ziyaret eder. Ana örnek ağaçta bu sıra `H D I B E A F C G`'dir. Bu dizide her düğümün bir solundaki ve bir sağındaki komşusu bellidir.

Tanım 12.1

Bir p düğümünün **kök ortada öncülü** (*inorder predecessor*), kök ortada gezinmede p'den hemen önce ziyaret edilen düğümdür. p'nin **kök ortada ardılı** (*inorder successor*) ise p'den hemen sonra ziyaret edilen düğümdür. Sıradaki ilk düğümün öncülü, son düğümün ardılı yoktur.

Örnek sırada E'nin öncülü B, ardılı A'dır. H ilk düğüm olduğu için öncülsüzdür; G son düğüm olduğu için ardılsızdır. Kılavuzların kuralı bu iki kavramla yazılır.

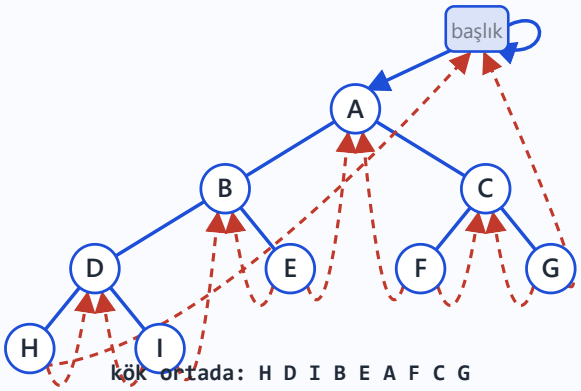
Tanım 12.2

Kılavuzlu ikili ağaç (*threaded binary tree*), boş işaretçileri şu iki kurala göre doldurulmuş ikili ağaçtır:

1. `p->solCocuk == NULL` ise `p->solCocuk`, p'nin kök ortada öncülünü gösterir.
2. `p->sagCocuk == NULL` ise `p->sagCocuk`, p'nin kök ortada ardılını gösterir.

Bu yolla doldurulan işaretçilere **kılavuz** denir.

Şekil 12.2 kuralların ana örneğe uygulanmış hâlidir. Düz mavi çizgiler gerçek çocuk bağlarıdır. Kesikli kırmızı oklar kılavuzlardır. Kılavuzlar ağacın kök ortada sırasını ağacın içine örer: her yapraktan sağa çıkan kırmızı ok, sırada bir sonraki düğüme gider.



Şekil 12.2: Ana örneğin kılavuzlu hâli. Kesikli kırmızı oklar kılavuzlardır: sol kılavuz öncüle, sağ kılavuz ardıla gider. H'nin sol ve G'nin sağ kılavuzunun gösterecek düğümü yoktur.

Kılavuzların bir ortak özelliği vardır: hepsi ağaçta yukarı doğru bir düğümü gösterir. Bunun nedeni şudur. Sol alt ağacı olmayan bir düğümün öncülü, düğümün içinde bulunduğu sağ alt ağacın sahibi olan en yakın atadır. Sağ alt ağacı olmayan bir düğümün ardılı da düğümün içinde bulunduğu sol alt ağacın sahibi olan en yakın atadır. Örneğin I, B'nin sol alt ağacındadır ve I'nın sağ alt ağacı yoktur; I'nın ardılı B'dir. Aşağı doğru yolculuk gerçek bağlarla yapılır, yukarı doğru dönüşler kılavuzlarla. Yığının hatırlamak zorunda olduğu bilgi tam da bu dönüş yollarıdır.

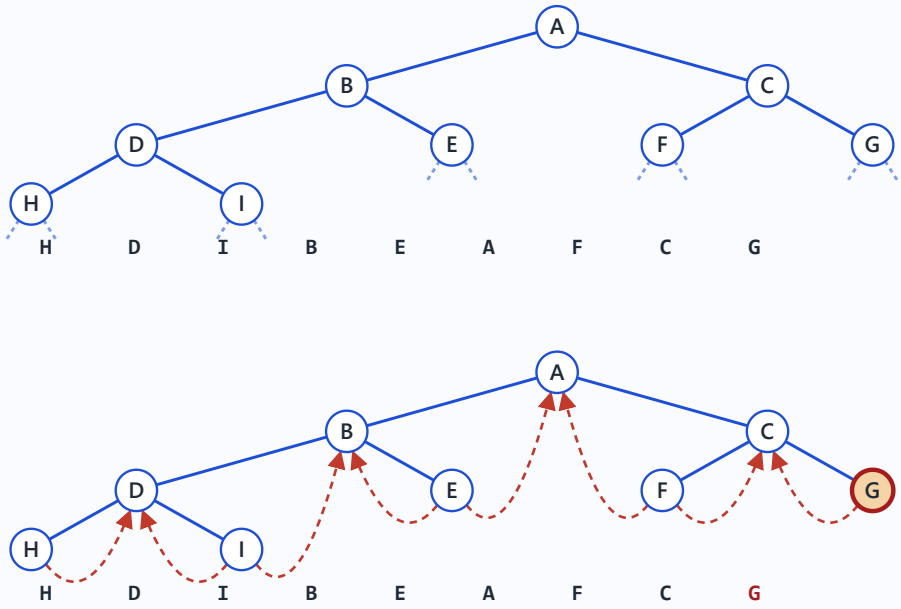
12.2.2 Kılavuzları tek tek kurmak

Kılavuzlar elle kurulurken önce kök ortada sıra yazılır. Sonra her boş işaretçi için o sırada düğümün bir solundaki ya da bir sağındaki düğüm okunur. Ana örnekte on boş alan vardır (Örnek 12.1). Tablo 12.2 her birinin nereyi göstereceğini verir.

Adım	Boş alan	Kural	Gösterdiği düğüm
1	H'nin solu	öncül	yok (başlık)
2	H'nin sağı	ardıl	D
3	I'nın solu	öncül	D
4	I'nın sağı	ardıl	B
5	E'nin solu	öncül	B
6	E'nin sağı	ardıl	A
7	F'nin solu	öncül	A
8	F'nin sağı	ardıl	C
9	G'nin solu	öncül	C
10	G'nin sağı	ardıl	yok (başlık)

Tablo 12.2: Kök ortada sıra `H D I B E A F C G` üzerinden on kılavuzun kurulması. “Başlık” satırları için gösterilecek düğüm yoktur; bu iki kılavuz başlık düğümüne bağlanacaktır.

Tablodaki her satır, sıradaki bir komşuluğa karşılık gelir. Kök ortada sırada sekiz komşu çift vardır (H–D, D–I, …, C–G). Her komşu çift tam olarak bir kılavuzla bağlanır; D ile H gibi gerçek bir bağla da bağlı olan çiftlerde bile (H'nin sağ kılavuzu D'yi gösterir). Kalan iki kılavuz sıranın iki ucundadır. Şekil 12.3 kurmanın başını ve sonunu gösterir. Üstte sıradan ağaç ve on boş işaretçisi, altta on kılavuzun hepsi yerindedir. Alttaki şeklin altına kök ortada sıra da yazılmıştır; her kırmızı okun bu sırada bir komşuyu gösterdiği oradan okunabilir.



Şekil 12.3: Kılavuz kurmanın ilk ve son durumu. Üstte 9 düğümlü sıradan ağaç ve 10 boş işaretçi (kesikli kısa çizgiler). Altta bütün boş işaretçiler öncül ya da ardıla bağlanmıştır; H'nin sol ve G'nin sağ kılavuzu şimdilik havadadır.

Örnek 12.2

A(B(D, E), C(–, F)) ağacının, yani A'nın çocukları B ve C, B'nin çocukları D ve E, C'nin yalnız sağ çocuğu F olan ağacın kılavuzları kurulsun. Kök ortada sıra `D B E A C F` 'dir. Düğüm sayısı 6, boş alan sayısı *6 + 1 = 7*'dir.

- D'nin solu: öncül yok, başlığa. D'nin sağı: ardıl B.
- E'nin solu: öncül B. E'nin sağı: ardıl A.
- C'nin solu: öncül A. C'nin sağ alt ağacı (F) olduğu için sağ işaretçisi gerçek bağdır.
- F'nin solu: öncül C. F'nin sağı: ardıl yok, başlığa.

Toplam 7 kılavuz vardır; bunların 2'si başlık düğümüne gider. C'nin sol kılavuzu dikkat ister: C'nin solu boş olduğu için C'nin öncülü A'dır. A, C'nin ebeveynidir; kılavuz bu kez ebeveyni gösterir.

12.2.3 Kılavuz bayrakları

Kâğıt üzerinde kılavuzlar kırmızı, gerçek bağlar siyah çizilir. Derleyici için ise iki işaretçi arasında hiçbir fark yoktur: ikisi de bir düğümün adresidir. Bir gezinme yordamı `p->solCocuk` 'u izlediğinde gerçek bir sol çocuğa mı inecek, yoksa yukarıdaki bir öncüle mi dönecek, bunu adresin kendisinden anlayamaz. Bu yüzden her işaretçinin yanında onun türünü söyleyen bir bilgi saklanır. Her düğüme iki ek alan eklenir:

```
typedef struct kilavuz_agac *kilavuz_gost;
struct kilavuz_agac {
    short int    solKilavuz; /* TRUE: solCocuk kılavuz */
    kilavuz_gost solCocuk;
    char         veri;
    kilavuz_gost sagCocuk;
    short int    sagKilavuz; /* TRUE: sagCocuk kılavuz */
};
```

Yapı, sıradan ikili ağaç düğümüne iki alan ekler. `solKilavuz` doğru (`TRUE`) ise `solCocuk` bir kılavuzdur, yani öncül gösterir; yanlış (`FALSE`) ise gerçek bir sol çocuğu gösterir. `sagKilavuz` aynı bilgiyi sağ işaretçi için taşır. `TRUE` ve `FALSE` sabitleri bir sayım tipiyle tanımlanır:

```
enum { FALSE, TRUE };      /* FALSE = 0, TRUE = 1 */
```

Sayım tipinde ilk ada 0, ikinciye 1 verilir. Böylece `FALSE` 0, `TRUE` 1 olur ve `if (p->solKilavuz)` gibi koşullar doğrudan çalışır. İlk satırdaki `typedef`, `struct kilavuz_agac *` tipine `kilavuz_gost` kısa adını verir. Bu satır yapının tanımından önce gelir; yapının alanları kendi tipinden işaretçi tuttuğu için önce adın bilinmesi gerekir. Yapı tanımı kapanış parantezinden sonra noktalı virgülle biter (8. satır). Bu noktalı virgül unutulursa derleyici hatayı çoğu zaman bir sonraki satırda bildirir ve hatanın yeri karışır.

Bayraklar belleği artırmaz mı? Artırır, ama zaman açısından kazanç çok daha büyüktür. Ders anlatımında bu karar açıkça söylenir: zaman karmaşıklığı alan karmaşıklığından daha önemlidir ve iki bayrak gerçekte iki bitle tutulabilir. Ne kadar yer harcandığını ölçmek öğreticidir. Yapının boyutu yalnız alanların toplamı değildir. Derleyici her alanı kendi hizalama sınırına yerleştirir ve araya dolgu baytları (*padding*) koyar. Tablo 12.3 x86-64 üzerinde `sizeof` ve `offsetof` ile ölçülen değerleri verir.

Alan	Konum (bayt)	Boyut	Arkasındaki dolgu
`solKilavuz`	0	2	6
`solCocuk`	8	8	0
`veri`	16	1	7
`sagCocuk`	24	8	0
`sagKilavuz`	32	2	6

Tablo 12.3: `struct kilavuz_agac`'ın bellekteki yerleşimi (x86-64, işaretçi 8 bayt, `short` 2 bayt). Ölçülen `sizeof` 40 bayttır.

40 baytın 19'u dolgudur. İşaretçiler 8'in katı olan adreslere yerleşmek zorundadır; araya giren küçük alanlar her seferinde 8 baytlık bir yuvayı tek başına kaplar. Alanlar büyükten küçüğe sıralanırsa (önce iki işaretçi, sonra iki `short`, en son `char`) aynı yapı 24 bayta iner. Bayraksız sıradan düğüm de 24 bayttır. Yani alan sırası iyi seçilirse bu makinede bayraklar hiç ek bellek getirmez; iki işaretçinin arkasında zaten boş kalan dolgu baytlarına yerleşirler. Bayraklar bit alanı (`unsigned solKilavuz : 1`) olarak tanımlansa da sonuç değişmez. Sunumdaki alan sırası, düğümün kâğıttaki çizimini izler: sol bayrak, sol işaretçi, veri, sağ işaretçi, sağ bayrak. Bu sıra okumayı kolaylaştırır ama bellekte en savurgan yerleşimdir.

Sık yapılan hata

Bayrağa bakmadan `solCocuk` ya da `sagCocuk`'u gerçek çocuk sanmak. Kılavuzlu ağaçta hiçbir işaretçi `NULL` değildir. Sıradan ağaç için yazılmış `while (p->solCocuk != NULL) p = p->solCocuk;` döngüsü kılavuzlu ağaçta durmaz. En soldaki düğümün sol kılavuzu başlığa, başlığın solu yeniden köke gider; döngü sonsuza dek döner. Kılavuzlu ağaçta her iniş bayrakla denetlenir.

12.2.4 Başlık düğümü

Tablo 12.2'de iki kılavuz havada kalmıştı. H'nin sol işaretçisi öncül göstermeli, ama H sıradaki ilk düğümdür. G'nin sağ işaretçisi ardıl göstermeli, ama G son düğümdür. Bu iki alana `NULL` yazmak, "hiçbir işaretçi boş değildir" düzenini bozar ve gezinme yordamlarına özel durum ekler. Çözüm, bağlı listelerden bilinen başlık düğümüdür.

Tanım 12.3

Kılavuzlu ağacın **başlık düğümü** (*header node*), veri taşımayan ve ağacın dışında duran ek bir düğümdür. Gerçek ağacın köküne `L->solCocuk` ile erişilir. Başlığın sağ işaretçisi her zaman kendisini gösterir ve `sagKilavuz` alanı `FALSE`'tur. Gösterecek düğümü olmayan iki kılavuz, yani ilk düğümün sol ve son düğümün sağ kılavuzu, başlığı gösterir.

Boş ağaçta yalnız başlık vardır. Bu durumda başlığın sol işaretçisi de kendisini gösterir ve `solKilavuz` `TRUE`'dur: gösterilen şey gerçek bir kök değil, bir kılavuzdur. Ağaca ilk düğüm eklendiğinde sol işaretçi bu düğüme çevrilir ve `solKilavuz` `FALSE` yapılır.

`solKilavuz`	`solCocuk`	`veri`	`sagCocuk`	`sagKilavuz`
`TRUE`	→ başlık	–	→ başlık	`FALSE`

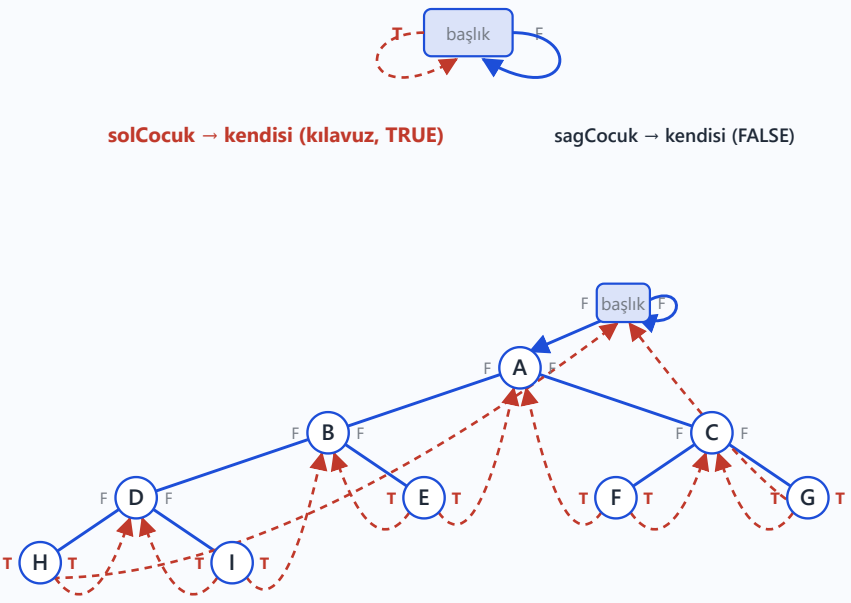
Tablo 12.4: Boş kılavuzlu ağacın tek düğümü olan başlığın alanları.

Bu düzen bir C yordamıyla kurulur. Sunumda yer almayan iki küçük yardımcı yazılır. `kilavuz_dugum`, yeni bir düğüm için bellek ayırır. `bos_kilavuzlu_agac` ise başlığı Tablo 12.4'teki gibi doldurur:

```
kilavuz_gost kilavuz_dugum(char veri) {
  kilavuz_gost d = malloc(sizeof *d);
  if (d == NULL) {              // bellek kalmadı
    fprintf(stderr, "bellek yok\n");
    exit(EXIT_FAILURE);
  }
  d->veri = veri;
  d->solCocuk = d->sagCocuk = NULL;   // ekleme yordamı dolduracak
  d->solKilavuz = d->sagKilavuz = TRUE;
  return d;
}
kilavuz_gost bos_kilavuzlu_agac(void) {
  kilavuz_gost L = kilavuz_dugum('-');  // başlık: verisi kullanılmaz
  L->solCocuk = L;  L->solKilavuz = TRUE;   // ağaç boş
  L->sagCocuk = L;  L->sagKilavuz = FALSE;  // sağı hep kendisi
  return L;
}
```

`malloc`, istenen kadar bellek bulamazsa `NULL` döndürür. Dönen değer denetlenmeden `d->veri`'ye yazmak, boş işaretçi üzerinden yazmak demektir ve tanımsız davranıştır. Yordam bu yüzden `NULL` gelince bir ileti basıp programı bitirir. `sizeof *d` ifadesi, `d`'nin gösterdiği tipin boyutunu verir. Tip adı tekrar yazılmadığı için yapı değişse bile ifade doğru kalır. Yeni düğümün işaretçileri `NULL` yapılır; düğüm ağaca eklenirken `saga_ekle` ya da `sola_ekle` bu değerlerin üzerine yazar. Bayrakların ilk değeri de aynı nedenle önemsizdir. `bos_kilavuzlu_agac`'ta başlığın verisine `'-'` yazılır; bu değer hiçbir yerde okunmaz.

Şekil 12.4'ün üst yarısı boş ağacı, alt yarısı ana örneği başlıkla birlikte gösterir. Şekildeki T ve F harfleri her düğümün iki yanındaki kılavuz bayraklarıdır. Başlığın solu köke (A) gider ve sol bayrağı F'dir. H'nin sol ve G'nin sağ kılavuzu artık başlığa bağlıdır. Başlığın sağ işaretçisi kendi üstüne kıvrılır.



Şekil 12.4: Başlık düğümü. Üstte boş ağaç: başlığın iki işaretçisi de kendini gösterir. Altta dolu ağaç: başlığın solu köke gider; havada kalan iki kılavuz başlığa bağlanmıştır. T ve F, düğümün o yandaki kılavuz bayrağıdır.

Başlıkla birlikte ağaç bir halka gibi kapanır. Kök ortada sırada son düğümden sonra başlığa, başlıktan sonra yeniden ilk düğüme gelinir. Gezinme yordamları bu halkayı kullanır: başlıktan başlar, başlığa dönünce durur. Böylece “ağaç boş mu”, “ilk düğüm hangisi”, “son düğümün ardılı yok” gibi özel durumların hiçbiri ayrı bir `if` gerektirmez.

12.2.5 Neden kök ortada sıra

Kılavuzlar kök ortada sıraya göre kurulur. Aynı ağacın üç gezinme sırası Tablo 12.5'te yan yanadır.

Gezinme	Kural	Sıra
kök ortada (*in-order*)	sol, kök, sağ	`H D I B E A F C G`
kök önde (*pre-order*)	kök, sol, sağ	`A B D H I E C F G`
kök sonda (*post-order*)	sol, sağ, kök	`H I D E B F G C A`

Tablo 12.5: Ana örneğin üç gezinme sırası.

Kök ortada sıranın ayrıcalığı kılavuzların tutarlılığından gelir. Bu sırada sol alt ağacı olmayan bir düğümün öncülü her zaman bir atasıdır; sağ alt ağacı olmayan bir düğümün ardılı da öyle. Yani boş bir sol alan tam da öncülünü, boş bir sağ alan tam da ardılını göstermeye uygundur. Dolu alanlar ise aşağıya iner ve öncül ya da ardıl, o alt ağacın en sağında ya da en solunda bulunur. İki kural birlikte her düğüm için ardılı ve öncülü kısa bir yoldan bulmayı sağlar. Kök önde sırada bu simetri yoktur. Örneğin D'nin kök önde ardılı H'dir ve D'den H'ye zaten gerçek bir sol bağ gider; bir kılavuzun eklediği bir bilgi kalmaz. Ders anlatımında bu yüzden ağacın kök ortada gezinmeye göre kurulduğu vurgulanır. Kök önde ve kök sonda gezinme de aynı kılavuzlarla yapılabilir, ama daha dolambaçlı bir yoldan.

12.3 Kök ortada gezinme

Kılavuzlu ağacın bütün gezinme yordamları tek bir soruya dayanır: verilen bir düğümün kök ortada ardılı hangisidir? Bu soru hızlıca yanıtlanabilirse gezinme, başlıktan başlayıp ardıldan ardıla geçmekten ibarettir.

12.3.1 Ardılı bulmak

Bir *p* düğümünün ardılı için iki durum vardır.

- `p->sagKilavuz == TRUE` ise `p->sagCocuk` bir kılavuzdur. Kılavuzun tanımı gereği bu işaretçi zaten *p*'nin ardılını gösterir. Arama gerekmez.
- `p->sagKilavuz == FALSE` ise *p*'nin gerçek bir sağ alt ağacı vardır. Kök ortada gezinme *p*'yi yazdıktan sonra bu alt ağaca girer ve orada ilk olarak en soldaki düğümü yazar. Ardıl, sağ alt ağacın en soldaki düğümüdür. Ona varmak için sağ çocuktan başlanır ve `solKilavuz` `TRUE` olana dek sol bağlar izlenir.

Bu iki kural sunumdaki yordamda birkaç satıra sığar:

```
kilavuz_gost kokortada_ardil_bul(kilavuz_gost L) {
  // L ile işaret edilen (NULL olmayan) öğenin ardılı
  kilavuz_gost p = L->sagCocuk;
  if (!L->sagKilavuz)        // sağ alt ağaç var mı
    while (!p->solKilavuz)   // en soluna dek in
      p = p->solCocuk;
  return p;
}
```

Yordam satır satır şöyle çalışır.

- **Satır 1–2.** Yordam bir düğüm işaretçisi alır ve ardılın adresini döndürür. Açıklama satırı ön koşulu söyler: `L` `NULL` olmamalıdır. Kılavuzlu ağaçta hiçbir işaretçi `NULL` olmadığı için yordamın içinde böyle bir denetim gerekmez; ön koşulu sağlamak çağıranın işidir.
- **Satır 3.** `p` , `L` 'nin sağ işaretçisini alır. Bu değer hem kılavuz durumunda doğrudan sonuçtur hem de sağ alt ağaç durumunda aramanın başlangıcıdır. İki durumun ortak ilk adımı tek atamada birleşir.
- **Satır 4.** `!L->sagKilavuz` doğruysa sağ işaretçi gerçek bir bağdır, yani sağ alt ağaç vardır. Açıklama satırı bayrağı değil, sorunun anlamını yazar: "sağ alt ağaç var mı".
- **Satır 5–6.** Sağ alt ağaçta sola inilir. Döngü koşulu `p->solKilavuz` 'a bakar, `p->solCocuk` 'a değil. En soldaki düğümün sol işaretçisi de dolu bir alandır (bir kılavuz), ama bayrağı `TRUE` 'dur.
- **Satır 7.** İki durumda da `p` ardılı gösterir ve döndürülür.

Kuralların doğruluğu kök ortada gezinmenin tanımından çıkar.

Teorem 12.2 (Ardıl kuralı)

Kılavuzlu bir ikili ağaçta p'nin sağ alt ağacı varsa ardılı bu alt ağacın en soldaki düğümüdür. Sağ alt ağacı yoksa ardılı `p->sagCocuk` kılavuzunun gösterdiği düğümdür. `kokortada_ardil_bul` her iki durumda da bu düğümü döndürür.

İspat. Kök ortada gezinme p'yi yazdıktan hemen sonra p'nin sağ alt ağacını gezmeye başlar. Bir alt ağacın gezinmesi her zaman en soldaki düğümüyle açılır: kökten sola inilir, sol çocuğu olmayan ilk düğüm yazılır. Öyleyse sağ alt ağaç varsa p'den sonra yazılan ilk düğüm o alt ağacın en soludur. Yordamın döngüsü sol bağları, sol çocuğu olmayan, yani `solKilavuz`'u `TRUE` olan ilk düğüme dek izler; bu düğüm en soldaki düğümdür. Sağ alt ağaç yoksa `p->sagCocuk` bir kılavuzdur ve Tanım 12.2'nin ikinci kuralı gereği ardılı gösterir. ■

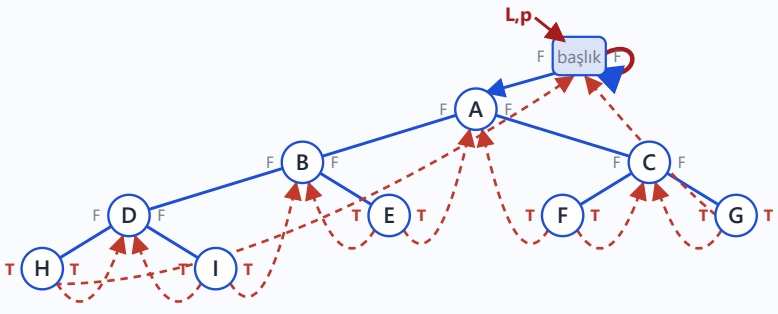
12.3.2 Kod izleme: başlık, A ve I

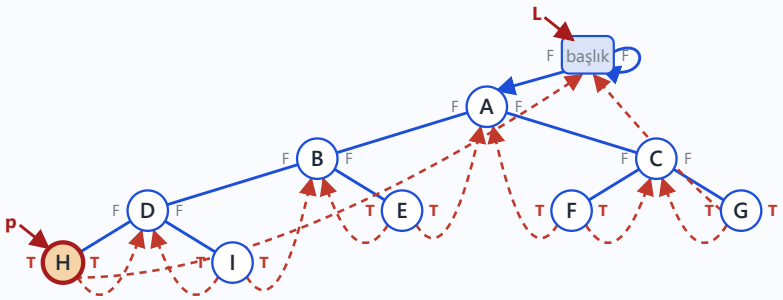
Yordam üç farklı düğümle çağrılsın. Tablo 12.6 her çağrıda `p`'nin aldığı değerleri ve karar veren bayrakları sırayla verir. Satır numaraları yukarıdaki koda aittir.

Çağrı	Satır	Olay	p
`L` = başlık	3	`p = başlık->sagCocuk`; başlığın sağı kendisidir	başlık
	4	`sagKilavuz` `FALSE`: başlığın gerçek bir sağ alt ağacı var sayılır	başlık
	5–6	başlık-> `solKilavuz` `FALSE`, sola	A
	5–6	A-> `solKilavuz` `FALSE`, sola	B
	5–6	B-> `solKilavuz` `FALSE`, sola	D
	5–6	D-> `solKilavuz` `FALSE`, sola	H
	7	H-> `solKilavuz` `TRUE`: döngü biter, H döner	H
`L` = A	3	`p = A->sagCocuk`	C
	4	`sagKilavuz` `FALSE`: A'nın sağ alt ağacı var	C
	5–6	C-> `solKilavuz` `FALSE`, sola	F
	7	F-> `solKilavuz` `TRUE`: F döner	F
`L` = I	3	`p = I->sagCocuk` (bir kılavuz)	B
	4	`sagKilavuz` `TRUE`: döngüye girilmez	B
	7	B döner	B

Tablo 12.6: `kokortada_ardil_bul` izlemesi. Ağaç, ana örneğin başlıklı hâlidir (Şekil 12.4, alt).

İlk çağrı özel bir sonuç verir: başlığın ardılı kök ortada sıradaki ilk düğümdür (H). Başlığın sağ işaretçisinin kendini göstermesi ve `sagKilavuz`'unun `FALSE` olması bunun için seçilmiştir. Yordam, başlığın sağ alt ağacının "kendisi" olduğunu düşünür, oradan sola inmeye başlar ve başlığın solundaki gerçek ağacın en soluna varır. Böylece gezinmeye başlamak için ayrı bir "en soldaki düğümü bul" yordamı gerekmez. Şekil 12.5 bu çağrının ilk ve son anını gösterir.





Şekil 12.5: `kokortada_ardil_bul(başlık)` . Üstte başlangıç: `L` ve `p` başlığı gösterir. Altta `p` , sol bağlar izlenerek H'ye inmiştir; H'nin sol bayrağı T olduğu için arama durur.

A için yapılan çağrıda sağ alt ağaç C'de başlar ve tek bir sola inişle F bulunur. I için yapılan çağrıda hiç arama yoktur; sağ işaretçi bir kılavuzdur ve doğrudan B'yi gösterir. Bir yaprağın ardılı her zaman böyle tek adımda bulunur.

> **Sık yapılan hata**
>
> Döngüyü `while (!p->solKilavuz)` yerine `while (!L->solKilavuz)` yazmak. Koşul her turda aynı düğüme baktığı için ya hiç girilmez ya da hiç çıkılmaz. Döngü değişkeni `p` 'dir ve koşul da onun bayrağını sınamalıdır.

12.3.3 Yığınsız gezinme

Ardıl yordamı hazır olunca kök ortada gezinme kısa bir döngüdür. Başlıktan başlanır; her turda bir sonraki düğüme geçilir ve yazılır. Başlığa geri dönülünce bütün ağaç gezilmiş olur.

```
void kilavuz_kokortada_dolas(kilavuz_gost L) {
  kilavuz_gost p = L;        // L: başlık düğümü
  for (;;) {
    p = kokortada_ardil_bul(p);
    if (p == L) break;       // başlığa dönüldü mü
    printf("%c ", p->veri);
  }
}
```

- **Satır 2.** `p` başlığı gösterir. Gezinmenin bütün durumu bu tek değişkendedir.
- **Satır 3.** `for (;;)` koşulsuz bir döngüdür; çıkış, gövdedeki `break` ile yapılır. Çıkış testi ardıl bulunduktan sonra yapılmak zorunda olduğu için koşul döngünün başına konamaz.
- **Satır 4.** `p` kendi ardılına ilerler.
- **Satır 5.** Ardıl başlıksa halka tamamlanmıştır. Son düğümün sağ kılavuzu başlığı gösterdiği için bu durum her zaman gerçekleşir.
- **Satır 6.** Düğüm yazılır.

Boş ağaçta ilk çağrı başlığın kendisini döndürür. `p = başlık->sagCocuk` başlıktır. `sagKilavuz` `FALSE` olduğu için döngüye girilir, ama başlığın `solKilavuz` 'u `TRUE` 'dur ve hemen çıkılır. `p == L` doğrudur ve hiçbir şey yazılmadan gezinme biter. Boş ağaç için ayrı bir denetim gerekmemesi başlık düğümünün kazancıdır.

Ana örnekte döngü on kez döner. Tablo 12.7, her turda ardılın hangi kuralla bulunduğunu ve kaç işaretçi izlendiğini verir. “İzlenen” sütunu, satır 3'teki okuma ile döngüdeki her sola inişi birer adım sayar.

Tur	`p` (önce)	Nasıl bulundu	`p` (sonra)	İzlenen	Çıktı
1	başlık	sağ alt ağacın en solu (A, B, D, H)	H	5	`H`
2	H	kılavuzla doğrudan	D	1	`D`
3	D	sağ alt ağacın en solu (I)	I	1	`I`
4	I	kılavuzla doğrudan	B	1	`B`
5	B	sağ alt ağacın en solu (E)	E	1	`E`

Tur	`p` (önce)	Nasıl bulundu	`p` (sonra)	İzlenen	Çıktı
6	E	kılavuzla doğrudan	A	1	`A`
7	A	sağ alt ağacın en solu (C, F)	F	2	`F`
8	F	kılavuzla doğrudan	C	1	`C`
9	C	sağ alt ağacın en solu (G)	G	1	`G`
10	G	kılavuzla doğrudan	başlık	1	(`break`)

Tablo 12.7: `kilavuz_kokortada_dolas` izlemesi. Çıktı `H D I B E A F C G`'dir; toplam 15 işaretçi izlenir.

Tablodaki iki kural sırayla birbirini izler. Bir yapraktan sonra kılavuzla yukarı çıkılır; iç düğümden sonra sağ alt ağaca girilip en sola inilir. Hiçbir turda bir düğüm iki kez yazılmaz ve hiçbir yerde bir yığına ihtiyaç duyulmaz.

12.3.4 Gezinmenin maliyeti

Tek bir `kokortada_ardil_bul` çağrısı uzun sürebilir. Başlıkla yapılan ilk çağrı kökten en soldaki düğüme kadar iner; bu yol ağacın yüksekliği kadar uzundur. Tek çağrının en kötü maliyeti bu yüzden $O(h)$'dir. Sola çarpık 1000 düğümlü bir ağaçta bu ilk çağrı 1001 işaretçi izler. Ama bütün gezinmenin toplam maliyeti, çağrı sayısı ile en kötü maliyetin çarpımından çok daha küçüktür.

Teorem 12.3 (Kılavuzlu gezinmenin maliyeti)

n düğümlü kılavuzlu bir ağacın `kilavuz_kokortada_dolas` ile gezinmesi en çok $2n + 1$ işaretçi izler. Zaman $\Theta(n)$, ek bellek $O(1)$'dir.

İspat. Döngü $n + 1$ kez `kokortada_ardil_bul` çağırır: n düğüm için birer kez ve başlığa dönüş için bir kez. Her çağrı satır 3'te tam bir işaretçi okur; bu, toplam $n + 1$ adım eder. Döngüdeki sola inişler yalnız gerçek sol bağlar üzerinden yapılır. Bir sol bağ, ancak bağın sahibi olan düğümün içinde bulunduğu alt ağaca soldan girilirken izlenir ve gezinme her alt ağaca bir kez girer. Öyleyse her gerçek sol bağ en çok bir kez izlenir. Gerçek sol bağların sayısı, başlıktan köke giden bağ da katılınca, en çok n'dir. Toplam $(n + 1) + n = 2n + 1$ olur. Her adım sabit süredir; zaman $O(n)$'dir. Her düğüm yazıldığı için zaman $\Omega(n)$'dir de. Yordam yalnız `p` değişkenini kullanır; ek bellek $O(1)$'dir. ■

Ana örnekte sol çocuğu olan düğümler A, B, C ve D'dir. Başlıktan köke giden bağla birlikte beş sola iniş yapılır. Toplam $10 + 5 = 15$ adım, Tablo 12.7'nin son sütununun toplamıyla aynıdır. Uç durumlar da sayılarak doğrulandı. Sola çarpık 1000 düğümlü ağaçta her düğümün sol çocuğu vardır ve sınır tam olarak erişilir: 2001 adım. Sağa çarpık 1000 düğümlü ağaçta yalnız başlıktan köke bir sola iniş yapılır: $1001 + 1 = 1002$ adım.

Toplam $2n + 1$ adım $n + 1$ çağrıya bölününce çağrı başına ikiden az adım düşer. Tek tek bakıldığında pahalı çağrılar olsa da bir gezinmede bunlar azdır; pahalı bir iniş, sonraki çağrıları ucuzlatır. Bu tür değerlendirmeye **amortize analiz** (*amortized analysis*) denir. Ardıl bulmanın amortize maliyeti gezinme içinde $O(1)$'dir.

Kılavuzların bir kazancı daha vardır. Herhangi bir düğümün adresi elde olduğunda ardılına, gezinmeye baştan başlamadan geçilebilir. Sıradan ağaçta bir düğümün ardılını bulmak için ya ebeveyn işaretçisi gerekir ya da kökten yeniden inmek gerekir. Kılavuzlu ağaç bu soruyu düğümün kendisinden başlayarak yanıtlar.

Böyle bir iyileştirmenin değeri küçümsenmemelidir. Ders anlatımında bir örnek verilir: bir problemin $O(n^2)$ sürede çözüldüğü bilinir ve birisi çalışıp $O(n^{1{,}89})$'a indiren bir algoritma yazar. Üsteki bu küçük fark büyük bir iştir ve literatüre o algoritma geçer. Kılavuzlu gezinme zamanı değil belleği iyileştirir: $O(h)$'yi $O(1)$'e indirir. Çarpık ağaçlarda bu, $O(n)$'den $O(1)$'e inmek demektir.

12.3.5 Kök önde gezinme

Kılavuzlar kök ortada sıraya göre kurulmuştur, ama aynı kılavuzlarla kök önde (*preorder*) gezinme de yapılabilir. Bunun için kök önde ardılı bulan bir yordam yazılır.

```
kilavuz_gost kokonde_ardil_bul(kilavuz_gost p) {
  if (!p->solKilavuz)        // sol çocuk varsa o
    return p->solCocuk;
  while (p->sagKilavuz)      // sağ kılavuzlarla çık
    p = p->sagCocuk;
  return p->sagCocuk;        // ilk gerçek sağ çocuk
}
```

Kök önde gezinmede bir düğüm yazıldıktan sonra varsa sol alt ağacına girilir; sol alt ağacın ilk düğümü onun köküdür. Satır 2–3 bu durumu karşılar: sol bayrak `FALSE` ise ardıl sol çocuktur. Sol çocuk yoksa ardıl, *p*'den başlayarak yukarıda bulunan ve henüz gezilmemiş ilk sağ alt ağacın köküdür. Satır 4–5 bu düğümü arar. *p*'nin sağ işaretçisi kılavuz olduğu sürece kılavuz izlenir. Sağ kılavuz, kök ortada ardıla, yani *p*'yi sol alt ağacında taşıyan en yakın ataya gider. O atanın sol alt ağacı bitmiştir ve sırada atanın sağ alt ağacı vardır. Ata da sağ alt ağaçsızsa (bayrağı yine `TRUE`) bir kılavuz daha izlenir. Gerçek bir sağ bağa sahip ilk düğümde durulur ve satır 6 onun sağ çocuğunu döndürür.

Bu yordamla gezinme döngüsü, kök ortada gezinmeyle aynı biçimdedir:

```
void kilavuz_kokonde_dolas(kilavuz_gost L) {
  kilavuz_gost p = L;        // L: başlık düğümü
  for (;;) {
    p = kokonde_ardil_bul(p);
    if (p == L) break;       // başlığa dönüldü mü
    printf("%c ", p->veri);
  }
}
```

Başlıktan başlanınca başlığın sol bayrağı `FALSE` olduğu için ardıl köktür (A). Son düğümden sonra kılavuzlar başlığa çıkar; başlığın sağ bayrağı `FALSE`, sağ işaretçisi kendisidir. Yordam başlığı döndürür ve gezinme biter. Tablo 12.8 bütün adımları verir.

`p`	Sol bayrak	İzlenen sağ kılavuzlar	Ardıl
başlık	F	–	A (sol çocuk)
A	F	–	B (sol çocuk)
B	F	–	D (sol çocuk)
D	F	–	H (sol çocuk)
H	T	H → D	I (D'nin sağ çocuğu)
I	T	I → B	E (B'nin sağ çocuğu)
E	T	E → A	C (A'nın sağ çocuğu)
C	F	–	F (sol çocuk)
F	T	F → C	G (C'nin sağ çocuğu)
G	T	G → başlık	başlık (başlığın sağı kendisi): biter

Tablo 12.8: `kokonde_ardil_bul` ile gezinme. Çıktı `A B D H I E C F G`'dir.

Bir yaprağın kök önde ardılını bulmak birden çok kılavuz izlemeyi gerektirebilir. Örneğin I'dan sonra tek kılavuzla B'ye çıkılır ve B'nin sağ çocuğu E bulunur. Daha derin bir ağaçta bir yapraktan başlayarak birkaç ata üst üste sağ alt ağaçsız olabilir; bu durumda kılavuz zinciri uzar. Yine de her sağ kılavuz bütün gezinmede en çok bir kez izlenir ve toplam zaman *O(n)*, ek bellek *O(1)* kalır.

Kök sonda (*postorder*) gezinme aynı kolaylıkla yazılamaz. Kök sonda sırada bir düğümden sonra genellikle ebeveyni gelir. Düğüm ebeveyninin sağ çocuğuysa ardılı ebeveyndir; sol çocuğuysa ardılı ebeveynin sağ alt ağacındaki ilk kök sonda düğümdür. İki durumda da ebeveyni bilmek gerekir. Kılavuzlar ise ebeveyni değil, kök ortada öncül ve ardılı gösterir; bu ikisi ebeveynle her zaman örtüşmez. H'nin sağ kılavuzu ebeveyni D'yi gösterir, ama I'nın sağ kılavuzu ebeveyni D'yi değil, B'yi gösterir. Kök sonda gezinmeyi kılavuzlarla yapmak mümkündür ama dolambaçlıdır. Her düğüme bir ebeveyn işaretçisi eklemek işi kolaylaştırır.

12.3.6 Morris gezinmesi

Kılavuzlu ağaç iki bedel öder: her düğümde bayrak alanları ve ekleme yordamlarının ek karmaşıklığı. J. M. Morris 1979'da bayrak alanı gerektirmeyen bir yöntem önerdi. **Morris gezinmesi** (*Morris traversal*), sıradan bir ikili ağaçta kılavuzları gezinme sırasında geçici olarak kurar, işi bitince söker. Ağaç, gezinmenin sonunda başlangıçtaki hâline döner. Yordam, bölümün başında verilen sıradan `struct agac` düğümüyle çalışır:

```
void morris_kokortada(agac_gost L) {
  while (L) {
    if (!L->solcocuk) {             /* sol yok: yaz, sağa */
      printf("%c", L->veri);  L = L->sagcocuk;
    } else {
      agac_gost p = L->solcocuk;    /* kök ortada öncülü */
      while (p->sagcocuk && p->sagcocuk != L)
        p = p->sagcocuk;
      if (!p->sagcocuk) {           /* geçici kılavuz kur */
        p->sagcocuk = L;  L = L->solcocuk;
      } else {                      /* kılavuzu sök */
        p->sagcocuk = NULL;
        printf("%c", L->veri);  L = L->sagcocuk;
      }
    }
  }
}
```

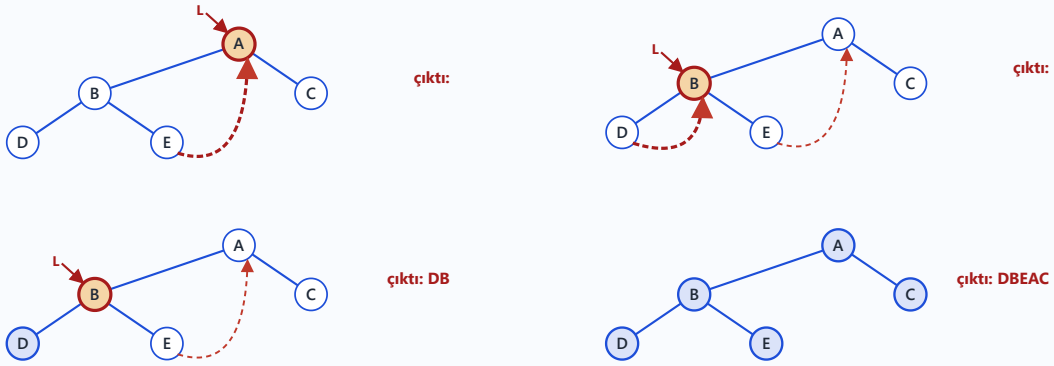
Yordamın fikri şudur. Sol alt ağacı olan bir *L* düğümüne gelindiğinde, sola inmeden önce dönüş yolu hazırlanır. *L*'nin kök ortada öncülü, sol alt ağacının en sağındaki düğümdür. Bu düğümün sağ işaretçisi boştur, çünkü en sağdaki düğümün sağ çocuğu olamaz. Bu boş alana *L*'nin adresi yazılır: geçici bir kılavuz. Sol alt ağaç gezilip öncüle varıldığında bu kılavuz izlenerek *L*'ye dönülür. *L*'ye ikinci gelişte öncül yeniden aranır; bu kez öncülün sağ işaretçisinin *L*'yi gösterdiği görülür. Bu, sol alt ağacın bittiğinin işaretidir. Kılavuz sökülür, *L* yazılır ve sağa geçilir.

- **Satır 2.** `L` `NULL` olana dek döner. Son düğümün sağ işaretçisi gerçekten `NULL` 'dır; döngü orada biter.
- **Satır 3–4.** Sol alt ağaç yoksa düğüm yazılır ve sağa geçilir. Sağ işaretçi gerçek bir çocuk da olabilir, daha önce kurulmuş geçici bir kılavuz da. Yordam ikisini ayırt etmez; ikisinde de doğru düğüme gidilir.
- **Satır 6–8.** Öncül aranır: sol çocuktan başlanır, sağa gidilebildiği sürece gidilir. Koşulun ikinci yarısı (`p->sagcocuk != L`) önemlidir. Kılavuz daha önce kurulmuşsa en sağdaki düğümün sağ işaretçisi `L` 'yi gösterir; bu sınama olmasaydı arama kılavuzu izleyip `L` 'ye, oradan yeniden sola dönerdi.
- **Satır 9–10.** Öncülün sağı boşsa ilk geliştir: kılavuz kurulur ve sola inilir.
- **Satır 11–13.** Öncülün sağı `L` 'yi gösteriyorsa ikinci geliştir: kılavuz sökülür, `L` yazılır, sağa geçilir.

Yordam A(B(D, E), C) ağacında, yani A'nın çocukları B ve C, B'nin çocukları D ve E olan beş düğümlü ağaçta izlensin. Tablo 12.9 her turu, Şekil 12.6 dört anı gösterir.

Tur	`L`	Öncül `p`	Olay	Geçici kılavuzlar	Çıktı
1	A	E (B'den sağa)	E'nin sağı boş: kılavuz kur, sola	E→A	
2	B	D	D'nin sağı boş: kılavuz kur, sola	E→A, D→B	
3	D	–	solu yok: yaz, sağa (kılavuzla B'ye)	E→A, D→B	`D`
4	B	D	D'nin sağı B: kılavuzu sök, yaz, sağa	E→A	`DB`
5	E	–	solu yok: yaz, sağa (kılavuzla A'ya)	E→A	`DBE`
6	A	E (B'den sağa)	E'nin sağı A: kılavuzu sök, yaz, sağa	–	`DBEA`
7	C	–	solu yok: yaz, sağa (`NULL`)	–	`DBEAC`

Tablo 12.9: `morris_kokortada` izlemesi, ağaç A(B(D, E), C). Çıktı `DBEAC` 'dir.



Şekil 12.6: Morris gezinmesinin dört anı. Birinci anda E → A kılavuzu kurulmuş, sola inilmiştir. İkincide D → B kılavuzu da kurulmuştur. Üçüncüde D'den B'ye dönülmüş, D → B sökülmüş, B yazılmıştır. Dördüncüde gezinme bitmiştir; ağaç eski hâlindedir.

Tablo, her geçici kılavuzun tam iki kez görüldüğünü gösterir: kurulurken ve sökülürken. Sol çocuğu olan her düğüm bir kılavuz doğurur; bu ağaçta iki tane (A ve B için). Gezinme bittiğinde D'nin ve E'nin sağ işaretçileri yeniden `NULL`'dır. Sonuç, aynı ağaçta özyinelemeli kök ortada gezinmenin çıktısıyla karşılaştırılarak doğrulandı:

```
void kokortada(agac_gost p) {
  if (p) {
    kokortada(p->solcocuk);
    printf("%c", p->veri);
    kokortada(p->sagcocuk);
  }
}
```

Morris gezinmesinin zamanı da $O(n)$'dir. Sol çocuğu olan her düğüm için öncül iki kez aranır. Her arama, sol çocuktan başlayıp sağa giden bir yolu yürür. Farklı düğümlerin bu yolları ortak kenar içermez; bir kenar en çok bir düğümün öncül yolunda bulunur. Yolların toplam uzunluğu bu yüzden en çok $n - 1$'dir ve iki kez yürünmeleri toplamı $2(n - 1)$ ile sınırlar. Buna her düğümde yapılan sabit iş eklenir. Ölçüm bunu doğrular: beş düğümlü örnekte toplam 13, yedi düğümlü tam ağaçta 18 işaretçi hareketi yapılır.

Yöntemin bir sınırı vardır. Gezinme sürerken ağaç geçici olarak değişir: bazı `NULL` alanlar başka düğümleri gösterir. Bu sırada başka bir iş parçacığı ağacı okursa döngülerle karşılaşır ve yanlış sonuç alır. Gezinme bir hata ya da erken çıkışla yarıda kesilirse ağaç bozuk kalır. Morris gezinmesi bu yüzden ağaca tek başına erişen ve gezinmeyi sonuna dek götüren yordamlar için uygundur.

12.4 Düğüm ekleme

Kılavuzlu ağaçta gezinme kolaydır; bedeli eklemede ödenir. İkili arama ağacına düğüm eklemek tek bir işaretçi atamasıdır: yeni düğüm, boş bir çocuk alanına yazılır. Kılavuzlu ağaçta ise boş alan yoktur. Yeni düğüm eklendiğinde hem onun kendi kılavuzları doğru yerleri göstermeli hem de çevresindeki kılavuzlar yeni sıraya uymalıdır. Ekleme bu yüzden tek başına kolay bir yordam değildir; biraz girifttir.

12.4.1 Ekleme neden girift

Yeni bir düğüm bir ebeveynin ya sağına ya da soluna eklenir. Sol ve sağ eklemenin işaretçi atamaları birbirinin aynası olsa da birebir aynı değildir; bu yüzden iki ayrı yordam yazılır: `saga_ekle` ve `sola_ekle`. Önce `saga_ekle` ele alınır. Sağa eklemede iki durum vardır:

1. Ebeveynin sağ alt ağacı yoktur. Yeni düğüm bir yaprak olarak ebeveynin sağına yerleşir.
2. Ebeveynin sağ alt ağacı vardır. Yeni düğüm ebeveyn ile bu alt ağacın arasına girer; eski alt ağaç yeni düğümün sağ alt ağacı olur.

İkinci durum sıradan ikili arama ağacına eklemede hiç ortaya çıkmaz; orada yeni düğüm hep bir yaprak olur. Kılavuzlu ağaçta ise iki durum da işe yarar: ikincisi, bir düğümü ağacın ortasına sokmanın yoludur.

12.4.2 Durum 1: sağ alt ağacı olmayan ebeveyne ekleme

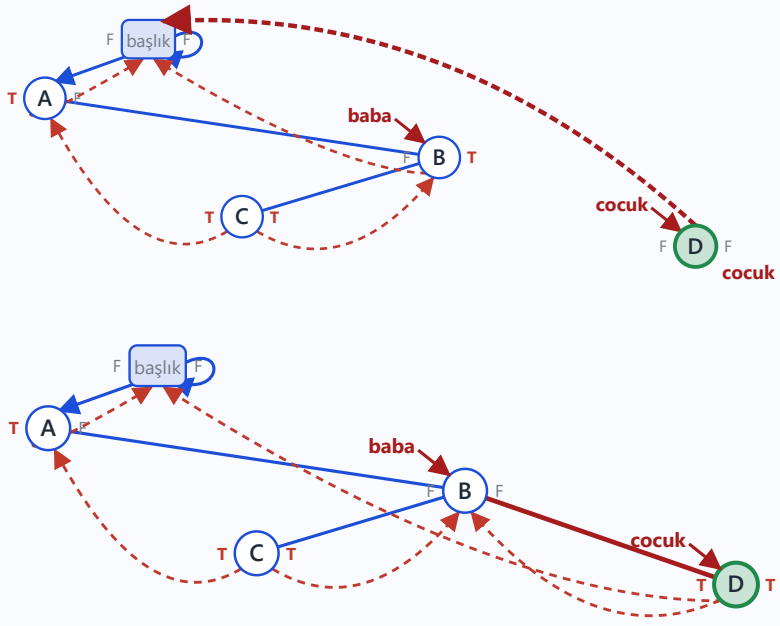
Ağaç A(–, B(C, –)) olsun: başlığın solunda A; A'nın sol çocuğu yok, sağ çocuğu B; B'nin sol çocuğu C, sağ çocuğu yok. Kök ortada sıra `A C B`'dir. Yeni düğüm D, B'nin sağına eklenecektir. B'nin sağ işaretçisi bir kılavuzdur ve B sıranın son düğümü olduğu için başlığı gösterir. Ekleme altı atamayla yapılır:

```
cocuk->sagCocuk  = baba->sagCocuk;
cocuk->sagKilavuz = TRUE;  // ebeveynin sağı yoktu
cocuk->solCocuk  = baba;
cocuk->solKilavuz = TRUE;
baba->sagCocuk   = cocuk;
baba->sagKilavuz = FALSE;  // TRUE idi
```

Her atamanın gerekçesi kök ortada sıradan okunur. Ekleme sonrası sıra `A C B D` olacaktır. D, B'den hemen sonra gelir; D'nin öncülü B, ardılı B'nin eski ardılıdır. Tablo 12.10 atamaları sırayla izler, Şekil 12.7 ilk ve son durumu gösterir.

Satır	Atama	Sonuç	Gerekçe
1	`cocuk->sagCocuk = baba->sagCocuk`	D → başlık	Ebeveynin ardılı artık çocuğun ardılıdır. Bu ardılın başlık olup olmaması önemsizdir.
2	`cocuk->sagKilavuz = TRUE`	kırmızı ok	Ebeveynin sağ alt ağacı yoktu; yaprak olan çocuğun da yoktur.
3	`cocuk->solCocuk = baba`	D → B	Çocuğun öncülü ebeveyndir.
4	`cocuk->solKilavuz = TRUE`	kırmızı ok	Bu da bir kılavuzdur.
5	`baba->sagCocuk = cocuk`	B → D	Ebeveynden çocuğa gerçek bağ.
6	`baba->sagKilavuz = FALSE`	siyah bağ	Ebeveynin sağı artık kılavuz değil, gerçek çocuktur.

Tablo 12.10: Durum 1 izlemesi: `baba` = B, `cocuk` = D.



Şekil 12.7: Durum 1. Üstte ilk atamadan sonra: D henüz ağaca bağlı değildir; sağ işaretçisi B'nin ardılı olan başlığı gösterir. Altta altı atamadan sonra: D, B'nin gerçek sağ çocuğudur; D'nin sol kılavuzu B'yi, sağ kılavuzu başlığı gösterir.

Atamaların sırası rastgele değildir. 1. satır ebeveynin sağ işaretçisini okur, 5. satır onun üzerine yazar. Bu iki satırın yeri değişirse `cocuk->sagCocuk`'a ebeveynin eski ardılı yerine çocuğun kendi adresi yazılır ve ardıl bilgisi kaybolur. Genel kural şudur: bir alanın eski değeri gerekiyorsa önce o değer kullanılır, sonra alan değiştirilir. Atamaların hepsi sabit sürelidir; Durum 1'in maliyeti *O(1)*'dir.

12.4.3 Durum 2: ebeveynin sağ alt ağacı var

Şimdi ağaç A(–, B(C, D(E, F))) olsun. B'nin sağ çocuğu D, D'nin çocukları E ve F'dir. Kök ortada sıra `A C B E D F`'dir. Yeni düğüm X, B'nin sağ çocuğu yapılacaktır. İstenen şudur: D alt ağacı kaybolmamalı ve X'in sağ alt ağacı olmalıdır. Kırmızı işaretçiler de yine doğru yerleri göstermelidir. Ekleme sonrası sıra `A C B X E D F` olacaktır.

Bu sıradan iki gereksinim okunur. X'in öncülü B'dir; bu Durum 1'deki gibidir. Ama X'ten sonra gelen düğüm E'dir, yani D alt ağacının en soldaki düğümü. E'nin sol kılavuzu eskiden B'yi gösteriyordu, çünkü E'nin eski öncülü B idi. Ekleme sonrası E'nin öncülü X olur. Öyleyse çocuk, ebeveynin eski ardılının öncülü yapılmalıdır.

Durum 1'in altı ataması bu ağaca olduğu gibi uygulanırsa ne olur? Deneme sonucu şudur: gezinme `A C B X D F` yazar; E kaybolur. Nedeni 2. satırdır. `cocuk->sagKilavuz = TRUE`, X'in sağ işaretçisini (D'yi gösteriyor) bir kılavuz sayar. Gezinme X'ten sonra kılavuz kuralıyla doğrudan D'ye atlar ve D'nin sol alt ağacına hiç inmez. E'nin sol kılavuzu da hâlâ B'yi gösterir. Ağaç bozulmuştur ve hiçbir derleyici uyarısı bunu haber vermez.

Düzeltme iki parçalıdır:

- 2. satır `cocuk->sagKilavuz = baba->sagKilavuz` olarak yazılır. Durum 1'de ebeveynin bayrağı `TRUE`'dur ve çocuğa kırmızılık geçer; Durum 2'de `FALSE`'tur ve siyahlık korunur. Ebeveynin alt ağacı artık çocuğun alt ağacıdır.
- Durum 2'de ebeveynin eski ardılının, yani yeni düzende çocuğun ardılının, sol kılavuzu çocuğa çevrilir.

Bu iki değişiklikle her iki durumun da üstesinden gelen tek bir yordam yazılabilir.

12.4.4 Tek yordam: `saga_ekle`

```
void saga_ekle(kilavuz_gost baba,
               kilavuz_gost cocuk) {
  // cocuk'u baba'nın sağ çocuğu yapar (kılavuzlu)
  kilavuz_gost p;
  cocuk->sagCocuk   = baba->sagCocuk;
  cocuk->sagKilavuz = baba->sagKilavuz;
  cocuk->solCocuk   = baba;
  cocuk->solKilavuz = TRUE;
  baba->sagCocuk    = cocuk;
  baba->sagKilavuz  = FALSE;
  if (!cocuk->sagKilavuz) { // ebeveynin alt ağacı var mı
    p = kokortada_ardil_bul(cocuk);
    p->solCocuk = cocuk;     // ardılın öncülü çocuk
  }
}
```

Yordam satır satır şöyle okunur. Satır numaraları yukarıdaki koda aittir.

- **Satır 1–2, açık tanım.** Yordamın adı ne yaptığını söyler; giden parametreler (`baba`, `cocuk`) bellidir; dönüş değeri yoktur (`void`). Açık tanımlı bir yordam, gövdesi okunmadan kullanılabilir.
- **Satır 5–6.** Çocuğun sağ işaretçisi ve sağ bayrağı ebeveynden alınır. Durum 1'de kırmızılık (`TRUE`), Durum 2'de siyahlık (`FALSE`) korunur. Ortak iş, ortak biçimde yazılmıştır.
- **Satır 7–8.** Çocuğun öncülü ebeveyndir: sol işaretçi ebeveyne gider, sol bayrak `TRUE` olur.
- **Satır 9.** Ebeveynden çocuğa gerçek bağ.
- **Satır 10.** Ebeveynin sağ bayrağı `FALSE` yapılır. Durum 2'de bayrak zaten `FALSE`'tur ve bu atama gereksiz ama zararsızdır; Durum 1'de gereklidir. Bir `if` ile ayırmak yerine iki durumda da yazmak kodu sadeleştirir.
- **Satır 11–13.** Çocuğun sağ bayrağı `FALSE` ise çocuğun gerçek bir sağ alt ağacı vardır; bu Durum 2'dir. Çocuğun ardılı bulunur ve onun sol kılavuzu çocuğa çevrilir. Durum 1'de yalnız bir sınama yapılır ve yordamdan çıkılır.

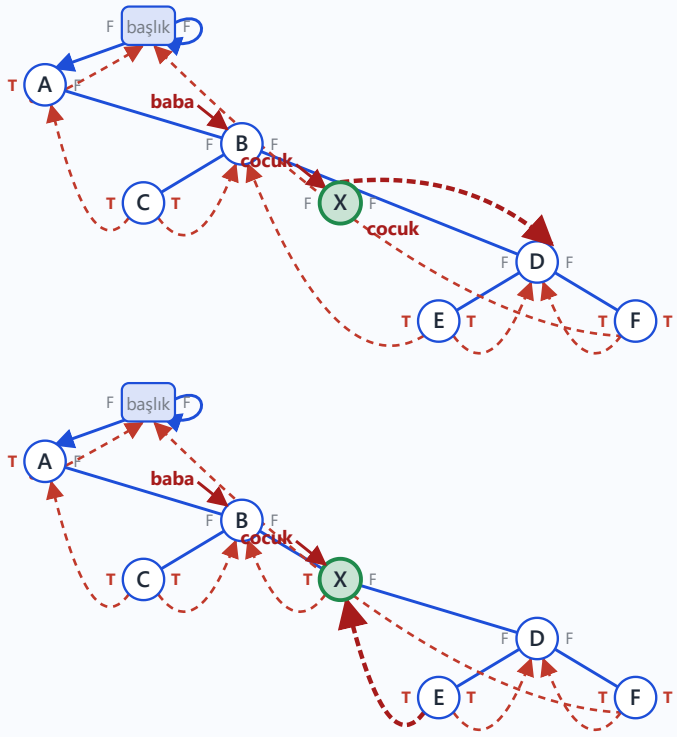
11. satırdaki koşul bayrağı sınar ama sorduğu soru şudur: ebeveynin sağ alt ağacı var mıydı? 5–6. satırlardan sonra ebeveynin eski sağ bilgisi çocuğa taşınmış olduğu için bu soru çocuğun bayrağına bakılarak yanıtlanır. Açıklama satırı da bayrağı değil, bu soruyu yazar.

Tablo 12.11, `saga_ekle(B, X)` çağrısını Durum 2 ağacında satır satır izler.

Satır	Olay	Değişen alan
5	X'in sağına B'nin eski sağ alt ağacı gelir	X-> `sagCocuk` = D

Satır	Olay	Değişen alan
6	B'nin bayrağı `FALSE` idi; siyahlık korunur	X-> `sagKilavuz` = `FALSE`
7	X'in öncülü B	X-> `solCocuk` = B
8	kırmızı ok	X-> `solKilavuz` = `TRUE`
9	B'den X'e gerçek bağ	B-> `sagCocuk` = X
10	zaten `FALSE` idi; zararsız	B-> `sagKilavuz` = `FALSE`
11	`!X->sagKilavuz` doğru: X'in sağ alt ağacı var	–
12	X'in ardılı: D'den sola inilir, E'de durulur	`p` = E
13	E'nin sol kılavuzu B yerine X'i gösterir	E-> `solCocuk` = X

Tablo 12.11: `saga_ekle(B, X)` izlemesi, ağaç A(–, B(C, D(E, F))). Sonuç sıra `A C B X E D F`'dir.



Şekil 12.8: `saga_ekle(B, X)`, Durum 2. Üstte ilk atamadan sonra: X'in sağı D'yi gösterir, X henüz ağaca bağlı değildir ve E'nin sol kılavuzu B'yi gösterir. Altta yordam bittikten sonra: X, B ile D'nin arasındadır; E'nin sol kılavuzu X'i gösterir.

10. satırdan sonra ağaç neredeyse doğrudur; X yerindedir ve gezinme bile `A C B X E D F` yazar, çünkü gezinme sol kılavuzları kullanmaz. Ama E'nin sol kılavuzu hâlâ B'yi gösterir. Kök ortada tersten gezinen ya da öncül arayan bir yordam bu hatayı hemen ortaya çıkarır. Ders anlatımı bu noktada uyarır: kodun doğru çalıştığına hemen karar verilmemelidir; daha işletilecek bir `if` bloğu vardır.

Aynı yordam Durum 1 ağacında da denendi. `saga_ekle(B, D)`, A(–, B(C, –)) ağacında altı atamalı kodla aynı sonucu verir: sıra `A C B D`, D'nin sol kılavuzu B'yi, sağ kılavuzu başlığı gösterir. Ebeveynin bayrağı `TRUE` olduğu için 11. satırdaki koşul yanlıştır ve `if` bloğu çalışmaz.

Maliyet. 5–10. satırlar sabit süredir. 12. satırdaki ardıl araması, çocuğun sağ alt ağacında sola iner; bu yol en çok ağacın yüksekliği kadardır. `saga_ekle`'nin zamanı en kötü durumda *O(h)*, Durum 1'de *O(1)*'dir. Ek bellek *O(1)*'dir.

12.4.5 Sola ekleme

`sola_ekle`, çocukla gösterilen düğümü ebeveynin sol çocuğu yapar. Yordam `saga_ekle`'nin aynasıdır: sol ile sağ ve öncül ile ardıl yer değiştirir. Ebeveynin eski öncülü, çocuğun öncülü olur. Ebeveynin sol alt ağacı varsa o alt ağacın en sağındaki düğümün, yani çocuğun yeni öncülünün, sağ kılavuzu çocuğa çevrilir. Bunun için kök ortada öncülü bulan bir yordam gerekir. Öncül kuralı ardıl kuralının aynasıdır: sol alt ağaç varsa onun en sağındaki düğüm, yoksa sol kılavuzun gösterdiği düğüm.

```
kilavuz_gost kokortada_oncel_bul(kilavuz_gost L) {
  kilavuz_gost p = L->solCocuk;
  if (!L->solKilavuz)
    while (!p->sagKilavuz) p = p->sagCocuk;
  return p;
}
void sola_ekle(kilavuz_gost baba, kilavuz_gost cocuk) {
  cocuk->solCocuk   = baba->solCocuk;
  cocuk->solKilavuz = baba->solKilavuz;
  cocuk->sagCocuk   = baba;
  cocuk->sagKilavuz = TRUE;      // ardılı ebeveyn
  baba->solCocuk    = cocuk;
  baba->solKilavuz  = FALSE;
  if (!cocuk->solKilavuz) {      // baba'nın sol alt ağacı vardı
    kilavuz_gost p = kokortada_oncel_bul(cocuk);
    p->sagCocuk = cocuk;         // öncülün ardılı çocuk
  }
}
```

`saga_ekle`	`sola_ekle`
`cocuk->sagCocuk = baba->sagCocuk`	`cocuk->solCocuk = baba->solCocuk`
`cocuk->sagKilavuz = baba->sagKilavuz`	`cocuk->solKilavuz = baba->solKilavuz`
`cocuk->solCocuk = baba` (öncülü ebeveyn)	`cocuk->sagCocuk = baba` (ardılı ebeveyn)
`baba->sagKilavuz = FALSE`	`baba->solKilavuz = FALSE`
ardılın sol kılavuzu çocuğa	öncülün sağ kılavuzu çocuğa
`kokortada_ardil_bul`: sağdan sonra sola sola	`kokortada_oncel_bul`: soldan sonra sağa sağa

Tablo 12.12: `saga_ekle` ile `sola_ekle` arasındaki ayna ilişkisi.

Örnek 12.3

Ağaç A(B(D, E), C) olsun; kök ortada sıra `D B E A C`'dir. `sola_ekle(A, X)` çağrılsın. A'nın sol alt ağacı (B) vardır; bu ikinci durumdur. Kodun ikinci yordamının satırları (8–16) sırayla şunları yapar:

1. X'in solu A'nın eski solunu alır: X-> `solCocuk` = B; bayrak `FALSE` olarak taşınır.
2. X'in sağı A'yı gösterir ve `TRUE`'dur: X'in ardılı A.
3. A-> `solCocuk` = X, A-> `solKilavuz` = `FALSE`.
4. X'in sol bayrağı `FALSE`; X'in öncülü aranır. `p` = X-> `solCocuk` = B. B'nin sağ bayrağı `FALSE`, sağa: `p` = E. E'nin sağ bayrağı `TRUE`, durulur.
5. E-> `sagCocuk` = X: E'nin sağ kılavuzu A yerine X'i gösterir.

Sonuç sıra `D B E X A C`'dir. X, B alt ağacı ile A'nın arasına girmiştir.

`kokortada_oncel_bul` yordamı başka bir iş için de kullanılabilir: başlıktan başlayıp başlığa dönene dek öncülden öncüle geçmek, ağacı kök ortada tersten gezer. Bu, alıştırmalarda ele alınır. Başlığın öncülü, kök ortada sıradaki son düğümdür: başlığın sol bayrağı `FALSE`, sol işaretçisi köktür ve kökten sağa sağa inilince en sağdaki düğüme varılır.

12.4.6 Ağacı geri vermek

Bir C programında `malloc` ile alınan her blok işi bitince `free` ile geri verilmelidir. Verilmeyen bloklar program sonuna dek bellekte kalır. Buna **bellek sızıntısı** denir. Uzun çalışan bir programda sızıntı, belleğin yavaş yavaş tükenmesine yol açar. Kılavuzlu ağacı geri vermek de gezinme ile yapılır:

```
void kilavuzlu_agac_sil(kilavuz_gost L) {
  kilavuz_gost p = kokortada_ardil_bul(L), sonraki;
  while (p != L) {
    sonraki = kokortada_ardil_bul(p);  // önce ardıl bulunur
    free(p);                           // sonra düğüm geri verilir
    p = sonraki;
  }
  free(L);                             // en son başlık
}
```

Sıra önemlidir. Bir düğüm `free` ile geri verildikten sonra onun alanlarını okumak, serbest bırakılmış belleği okumaktır ve tanımsız davranıştır. Bu yüzden önce ardıl bulunur ve `sonraki`'ne yazılır, sonra düğüm geri verilir. Ardıl yordamı yalnız `p`'nin ve sağ alt ağacındaki düğümlerin alanlarını okur. Bu düğümler henüz geri verilmemiştir, çünkü kök ortada sırada `p`'den sonra gelirler. Başlık en son geri verilir; gezinmenin durma koşulu onun adresine bakar.

Bu notun bütün kodları, `malloc` ve `free` çağrılarını sayan bir sarmalayıcıyla sınandı. Sunumdaki kodlar ve notun ek yordamları tek bir sürücüde çalıştırıldı: 32 `malloc` ve 32 `free` sayıldı; ayrıntılı sınamada bu sayılar 2116'şar oldu. Sızıntı yoktur.

12.4.7 Kod kalitesi

Kılavuzlu ağaç yordamları kısadır ama hataya açıktır. Bu yordamlar, iyi kod yazmanın birkaç ilkesini somut olarak gösterir.

NULL denetiminin yeri. `saga_ekle`, `baba`'nın ya da `cocuk`'un `NULL` olup olmadığını denetlemez. Bu bir unutkanlık değil, bir tasarım kararıdır. Ders anlatımındaki gerekçe şudur: `saga_ekle` tek başına kullanılmak için değil, başka bir yordamın içinde kullanılmak için yazılmıştır; `NULL` denetimi onu kullanan yordamda yapılmalıdır. Örneğin kılavuzlu bir ikili arama ağacına ekleme yapan yordam önce yeni değerin sağa mı sola mı ekleneceğini bulur. Sıradan ağaçtaki tek atamanın yerine `saga_ekle` ya da `sola_ekle`'yi çağırır. Bu noktada `baba` zaten gerçek bir düğümdür. Ama yordam bir kütüphanenin parçası olacak ve herkes doğrudan çağırabilecekse denetim yordama konmalıdır. Denetim yapılmıyorsa bu, açıklamada yazılmalıdır: "`baba` ve `cocuk` `NULL` olmamalı". Genel amaçlı bir yordam için savunmacı bir sarmalayıcı şöyle yazılabilir:

```
#include <assert.h>
void saga_ekle_denetimli(kilavuz_gost baba, kilavuz_gost cocuk) {
  // baba ve cocuk NULL olmamalı; cocuk henüz ağaçta olmamalı
  assert(baba != NULL && cocuk != NULL);
  saga_ekle(baba, cocuk);
}
```

`assert`, koşulu yanlışsa programı dosya adı ve satır numarasıyla durdurur. Programcı hatalarını erken yakalar ve sınama sırasında çok değerlidir. `NDEBUG` tanımlanarak derlenen sürümde kaldırılır; bu yüzden kullanıcı girdisinden gelebilecek hatalar için değil, "bu asla olmamalı" denen durumlar için kullanılır. Kullanıcı hatasına karşı bir hata kodu döndürmek daha uygundur.

Açıklama satırı. Açıklama satırı kodun ne yapmak istediğini söylemeli, kodun kendisini tekrarlamamalıdır.

```
/* KÖTÜ: kodu tekrarlıyor */
if (!cocuk->sagKilavuz)  // sağ kılavuz FALSE mu
/* İYİ: niyeti söylüyor */
if (!cocuk->sagKilavuz)  // ebeveynin alt ağacı var mı
```

"Sağ kılavuz `FALSE` mu" açıklaması okura yeni bir şey söylemez; aynı bilgi kodda yazılıdır. "Ebeveynin alt ağacı var mı" ise niyeti söyler. Niyet ile kod yan yana durunca okur ikisini karşılaştırabilir: "yapmak istediği doğru, ama kodu yanlış yazmış" diyebilir. Hata ayıklama böyle kolaylaşır. Ders anlatımında bu konu açıkça vurgulanır: sonradan gelen okur "ben onu zaten biliyorum" dememeli; açıklama satırları kodda söylenenden farklı bir şey söylemelidir.

Tek yordam. Her durum için ayrı yordam yazmak mümkündü: `cocukYok_sagaEkle`, `cocukVar_sagaEkle`, `cocukYok_solaEkle`, `cocukVar_solaEkle`. Bu, kodu bölük pörçük eder; kullanan, hangi yordamı çağıracağına her seferinde karar vermek zorunda kalır. Karmaşıklığı artırmadan dört yordamı ikiye indirmek iyidir. Ama ters yöndeki aşırılıktan da kaçınılmalıdır. İç içe `if`'lerle bütün durumları tek bir yordama sıkıştırmak okunmaz bir kod doğurur. Bu durumda ayrı ayrı yordamlar daha iyidir. `saga_ekle`'nin başarısı, iki durumun ortak işini ortak biçimde tanımlamasından gelir: kılavuz bayrağını ebeveynden kopyalamak iki durumu tek atamada karşılar. Yöntem şöyle özetlenebilir: önce üç beş farklı durum analiz edilir; sonra yordamın adı, dönüş değeri ve parametreleri belirlenir; ortak işler ortak biçimde yazılır. Sonuç sade, okunabilir ve bakımı kolay bir koddur. Ders anlatımı bu ilkeleri yazılım mühendisliğinin temeli sayar ve öğrenilmelerinin yıllar aldığını hatırlatır.

Teyakkuz. Kılavuzlu ağaç kodu yazarken her satırda "bunu yapıyorum, ama doğru mu yapıyorum?" diye sormak gerekir. Durum 1'in kodu Durum 1 için kusursuzdur ve Durum 2'de sessizce ağacı bozar. `saga_ekle`'nin ilk altı atamasından sonra ağaç doğru görünür ama bir kılavuz hâlâ yanlıştır. Bu tür hatalar ancak bütün durumların ayrı ayrı denenmesiyle ve hem ileri hem geri gezinmeyle yakalanır.

12.5 Kazanç, bedel ve karmaşıklık özeti

Kılavuzlu ağacın getirdiği ve götürdüğü şeyler iki listede toplanabilir. Kazançlar: gezinme yığınsız ve *O(1)* ek bellekle yapılır; her düğümden ardıla ve öncüle en çok *O(h)*'de, gezinme içinde amortize *O(1)*'de geçilir; boş alanlar boşa gitmez. Bedeller: her düğümde iki bayrak tutulur; ekleme ve silme daha girift olur; ağaca dokunan her yordam kılavuzları gözetmek zorundadır. Tablo 12.13 bölümün yordamlarının maliyetlerini bir arada verir.

İşlem	Zaman	Ek bellek	Açıklama
`kokortada_ardil_bul`	O(h) en kötü	O(1)	sağ kılavuz varsa O(1)
`kokortada_oncel_bul`	O(h) en kötü	O(1)	sol kılavuz varsa O(1)
`kilavuz_kokortada_dolas`	Θ(n)	O(1)	en çok 2n + 1 işaretçi; yığın ya da özyineleme yok
kök önde kılavuzlu gezinme	Θ(n)	O(1)	aynı kılavuzlarla
`saga_ekle` / `sola_ekle`	O(h)	O(1)	ardıl ya da öncül araması; 1. durumda O(1)
`kilavuzlu_agac_sil`	Θ(n)	O(1)	gezinme ile geri verme
Morris gezinmesi (bayraksız)	Θ(n)	O(1)	geçici kılavuzlar; ağaç gezinme sırasında değişir
özyinelemeli kök ortada (karşılaştırma)	Θ(n)	O(h)	çağrı yığını

Tablo 12.13: Karmaşıklık özeti. *n* düğüm sayısı, *h* ağacın yüksekliğidir.

12.6 Sık yapılan hatalar

1. **Bayrağa bakmadan işaretçiyi gerçek çocuk sanmak.** Kılavuzlu ağaçta hiçbir işaretçi `NULL` değildir. `NULL`'a dek inen bir döngü durmaz; kılavuzlar ve başlık üzerinden halkada döner. Her iniş `solKilavuz` ya da `sagKilavuz` ile denetlenir.
2. **Başlık düğümünü unutmak.** Başlıksız bir ağaçta ilk düğümün sol ve son düğümün sağ kılavuzu havada kalır. Bunlara `NULL` yazılırsa gezinme yordamları özel durum ister; yazılmazsa çöp değerler izlenir.
3. **Sağa eklemenin ikinci durumunda ardılın sol kılavuzunu güncellememek.** İleri gezinme yine doğru sonuç verir ve hata gizli kalır; öncül arayan ya da tersten gezinen bir yordam yanlış düğüme gider.
4. `cocuk->sagKilavuz`**'u sabit** `TRUE` **yazmak.** İkinci durumda ebeveynin sağ alt ağacı bir kılavuz sayılır ve o alt ağacın sol kısmı gezinmeden düşer. Örnekte E kaybolur.
5. **Boş ağaçta başlığın** `solKilavuz` **'unu unutmak.** Boş ağaçta başlığın solu kendisidir ve bayrağı `TRUE`'dur. Bu bayrağa bakmadan `L->solCocuk`'u kök sanan bir arama başlığın verisini gerçek bir değer gibi okur.
6. **Büyük ve küçük harfi karıştırmak.** C büyük ve küçük harfe duyarlıdır: `baba->SagCocuk` ya da `baba->SagKilavuz` derlenmez, çünkü yapının alanı `sagCocuk`'tur. Kılavuzlu yapıda `solCocuk`, sıradan yapıda `solcocuk` yazıldığına da dikkat edilmelidir.
7. **Atama sırasını bozmak.** `baba->sagCocuk = cocuk` satırı `cocuk->sagCocuk = baba->sagCocuk`'tan önce yazılırsa ebeveynin eski ardılı kaybolur ve çocuk kendini gösterir.
8. **Ağacı geri verirken önce** `free` **etmek.** Düğüm geri verildikten sonra ardılı aranırsa serbest bırakılmış bellek okunur. Önce ardıl bulunur, sonra düğüm geri verilir.

12.7 Alıştırmalar ve çözümleri

12.7.1 Bir ağacın kılavuzlarını çizmek

Soru. A(B(D, E), C(–, F)) ağacının kılavuzlarını çiziniz; kaç kılavuz vardır?

Çözüm. Kök ortada sıra `D B E A C F`'dir. Ağaçta 6 düğüm vardır; Teorem 12.1'e göre *6 + 1 = 7* boş alan, yani 7 kılavuz olur. Kılavuzlar Örnek 12.2'de tek tek verildi: D → başlık, D → B, E → B, E → A, C → A, F → C, F → başlık. 2'si başlığa gider. Çözüm, ağacı kurup bayrakları sayan bir programla da doğrulandı.

12.7.2 Öncül bulmak

Soru. Kök ortada öncülü bulan `kokortada_oncel_bul` yordamını yazınız.

Çözüm. Yordam ardıl bulmanın aynasıdır ve sola ekleme kesiminde verildi. Sol bayrak `TRUE` ise sol işaretçi doğrudan öncülü gösterir. Değilse sol çocuktan başlanır ve sağ bayrak `TRUE` olana dek sağa gidilir. Ana örnekte A'nın öncülü şöyle bulunur: `p` = B; B'nin sağ bayrağı `FALSE`, `p` = E; E'nin sağ bayrağı `TRUE`; sonuç E.

12.7.3 Tersten gezinme

Soru. Kılavuzlu ağacı kök ortada tersten (`G C F A …`) gezen yordamı yazınız.

Çözüm. Başlıktan başlanır ve başlığa dönene dek öncülden öncüle ilerlenir:

```
void kilavuz_ters_dolas(kilavuz_gost L) {
  kilavuz_gost p = L;
  for (;;) {
    p = kokortada_oncel_bul(p);
    if (p == L) break;
    printf("%c ", p->veri);
  }
}
```

Başlığın öncülü son düğümdür (G); ilk düğümün (H) öncülü de başlıktır. Ana örnekte çıktı `G C F A E B I D H`'dir. Zaman $\Theta(n)$, ek bellek $O(1)$'dir; gerekçe Teorem 12.3'ün aynasıdır.

12.7.4 Boş ağaca ilk düğüm

Soru. Boş ağaca (yalnız başlık) ilk düğümü `sola_ekle(baslik, x)` ile eklemek doğru mudur?

Çözüm. Doğrudur. Başlığın sol işaretçisi kendisini gösterir ve sol bayrağı `TRUE`'dur. `sola_ekle` bu ikisini çocuğa taşır: çocuğun solu başlığı gösterir ve bir kılavuzdur. Çocuğun sağı da ebeveyni, yani başlığı gösterir ve `TRUE`'dur. Başlığın solu çocuğa çevrilir ve sol bayrağı `FALSE` olur. Çocuğun sol bayrağı `TRUE` olduğu için `if` bloğu çalışmaz. Sonuçta tek düğümün iki kılavuzu da başlığı gösterir; bu, Tanım 12.3'teki düzenin tam kendisidir.

12.7.5 Kök sonda gezinmenin zorluğu

Soru. Kök sonda gezinme kılavuzlarla neden zordur? Hangi ek bilgi işi kolaylaştırır?

Çözüm. Bir düğümün kök sonda ardılı ebeveynine bağlıdır. Düğüm ebeveyninin sağ çocuğuysa ya da ebeveynin sağ alt ağacı yoksa ardıl ebeveyndir; değilse ardıl, ebeveynin sağ alt ağacında kök sonda sırada ilk düğümdür. İki durumda da ebeveynin kim olduğu ve düğümün onun hangi çocuğu olduğu bilinmelidir. Kılavuzlar kök ortada öncül ve ardılı gösterir; bunlar ebeveynle her zaman örtüşmez. Her düğüme bir ebeveyn işaretçisi eklemek bu bilgiyi doğrudan verir ve kök sonda ardılı sabit sayıda adım ve bir iniş ile bulunur.

12.7.6 Kılavuzlu ikili arama ağacına ekleme

Soru. Kılavuzlu bir ikili arama ağacına ekleme yordamını `saga_ekle` ve `sola_ekle` kullanarak yazınız.

Çözüm. Arama sıradan ikili arama ağacındaki gibidir; tek fark, inilecek yönün bayrağı `TRUE` olduğunda durulmasıdır. O noktada aranan değer düğümden küçükse `sola_ekle`, büyükse `saga_ekle` çağrılır. Yeni düğüm her zaman yaprak olur; bu yüzden iki yordam da birinci durumda çalışır ve sabit sürede biter.

```
int kilavuzlu_bst_ekle(kilavuz_gost L, char x) {
  // x yoksa ekler ve 1 döndürür; varsa 0 döndürür
  kilavuz_gost p;
  if (L->solKilavuz) {                  // ağaç boş
    sola_ekle(L, kilavuz_dugum(x));
    return 1;
  }
  p = L->solCocuk;                      // kökten başla
  for (;;) {
    if (x == p->veri) return 0;
    if (x < p->veri) {
      if (p->solKilavuz) break;         // sol çocuk yok
      p = p->solCocuk;
    } else {
      if (p->sagKilavuz) break;         // sağ çocuk yok
      p = p->sagCocuk;
    }
  }
  if (x < p->veri) sola_ekle(p, kilavuz_dugum(x));
  else             saga_ekle(p, kilavuz_dugum(x));
  return 1;
}
```

Ağaç boşsa (başlığın sol bayrağı `TRUE`) ilk düğüm başlığın soluna eklenir; önceki alıştırma bunun doğru olduğunu gösterdi. Yordam `M F T B H P X K` sırasıyla çağrıldı; sonra `H` bir kez daha eklenmek istendi ve 0 döndü. Kök ortada gezinme `B F H K M P T X`, ters gezinme `X T P M K H F B` yazdı. Arama *O(h)*, ekleme *O(1)* sürer; toplam *O(h)*'dir.

12.8 Özet

- *n* düğümlü ikili ağaçta *2n* işaretçinin *n − 1*'i doludur; kalan *n + 1* alan boştur.
- Kılavuzlu ikili ağaçta boş sol işaretçi kök ortada öncüle, boş sağ işaretçi kök ortada ardıla bağlanır. Bu işaretçilere kılavuz denir.
- `solKilavuz` ve `sagKilavuz` bayrakları gerçek bağı kılavuzdan ayırır. Alan sırası iyi seçilirse bayraklar 64 bitlik makinede dolgu baytlarına sığar.
- Başlık düğümü havada kalan iki kılavuzu toplar; kök `L->solCocuk`'tadır, başlığın sağı kendisidir. Boş ağaçta başlığın sol bayrağı `TRUE`'dur.
- Ardıl: sağ bayrak `TRUE` ise doğrudan sağ işaretçi; değilse sağ alt ağacın en solu. Öncül bunun aynasıdır.
- Kılavuzlu kök ortada gezinme en çok *2n + 1* işaretçi izler: *Θ(n)* zaman, *O(1)* ek bellek. Kök önde gezinme de aynı kılavuzlarla yapılır; kök sonda için ebeveyn bilgisi gerekir.
- Morris gezinmesi bayrak olmadan, kılavuzları geçici kurup sökerek *O(n)* zaman ve *O(1)* bellekle gezinir; gezinme sırasında ağaç değiştiği için eşzamanlı okumaya uygun değildir.
- `saga_ekle` kılavuz bayrağını ebeveynden kopyalayarak iki durumu tek yordamda birleştirir; ikinci durumda çocuk, ebeveynin eski ardılının öncülü yapılır. `sola_ekle` onun aynasıdır.
- İyi kod: açık tanımlı yordam, niyeti söyleyen açıklama, NULL denetiminin doğru yerde yapılması ve ortak işlerin ortak biçimde yazıldığı tek yordam.

12.9 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

boş işaretçi *null pointer* · kılavuz *thread* · kılavuzlu ikili ağaç *threaded binary tree* · öncül *predecessor* · ardıl *successor* · başlık düğümü *header node* · kök ortada *inorder* · kök önde *preorder* · kök sonda *postorder* · gezinme *traversal* · Morris gezinmesi *Morris traversal* · çağrı yığını *call stack* · çarpık ağaç *skewed tree* · yükseklik *height* · amortize analiz *amortized analysis*

kendine başvuran yapı *self-referential struct* | ikili arama ağacı *binary search tree*

12.10 Kendini Dene

1. 15 düğümlü bir ikili ağaç kılavuzlu hâle getiriliyor. Kaç kılavuz kurulur ve kaçı başlığa gider?

Cevap

Boş alan sayısı *15 + 1 = 16*'dır; 16 kılavuz kurulur. Ağacın biçimi ne olursa olsun yalnız ikisi başlığa gider: ilk düğümün sol ve son düğümün sağ kılavuzu.

2. Ana örnekte `kokortada_ardil_bul(B)` hangi düğümü döndürür? Döngü kaç kez döner?

Cevap

E'yi döndürür. B'nin sağ bayrağı `FALSE`, `p` = E olur. E'nin sol bayrağı `TRUE` olduğu için döngüye hiç girilmez: 0 tur.

3. Ana örnekte `kokonde_ardil_bul(I)` hangi düğümü döndürür?

Cevap

E. I'nın sol bayrağı `TRUE` olduğu için sağ kılavuzlar izlenir: I → B. B'nin sağ bayrağı `FALSE` 'tur; döngü biter ve B'nin sağ çocuğu E döner.

4. Ana örneğe `saga_ekle(E, Y)` uygulanıyor. Hangi durumdur? Y'nin kılavuzları nereyi gösterir ve yeni kök ortada sıra nedir?

Cevap

E bir yapraktır; sağ alt ağacı yoktur, birinci durumdur. Y'nin sol kılavuzu E'yi, sağ kılavuzu E'nin eski ardılı A'yı gösterir. Sıra `H D I B E Y A F C G` olur.

5. Ana örneğe `saga_ekle(A, Y)` uygulanıyor. Hangi düğümün kılavuzu değişir?

Cevap

İkinci durumdur; A'nın sağ alt ağacı (C) Y'nin sağ alt ağacı olur. Y'nin ardılı C'nin en solundaki F'dir. F'nin sol kılavuzu A yerine Y'yi gösterir. Sıra `H D I B E A Y F C G` olur.

6. Sola çarpık 1000 düğümlü kılavuzlu bir ağaçta `kokortada_ardil_bul(başlık)` kaç işaretçi izler? Bütün gezinme kaç izler?

Cevap

İlk çağrı başlıktan köke, sonra 999 sol bağla en alttaki düğüme iner: satır 3'teki okumayla birlikte 1001 işaretçi. Bütün gezinme *2n + 1 = 2001* işaretçi izler; bu, Teorem 12.3'ün sınırına tam olarak erişen durumdur.

7. Morris gezinmesi A(B(D, E), C) ağacında kaç geçici kılavuz kurar? Genel olarak bu sayı neye eşittir?

Cevap

İki: E → A ve D → B. Sol çocuğu olan her düğüm için bir kılavuz kurulur ve aynı kılavuz daha sonra sökülür.

8. `saga_ekle` 'de 10. satır (`baba->sagKilavuz = FALSE`) silinirse hangi durumda ağaç bozulur?

Cevap

Birinci durumda, ama sinsi biçimde. Ebeveynin bayrağı `TRUE` kalır. Kök ortada gezinme yine doğru sonuç verir, çünkü "kılavuz" sanılan sağ işaretçi zaten çocuğu, yani ebeveynin ardılını gösterir. Öteki yordamlar ise çocuğu atlar. A(B, –) ağacına C sağdan eklenince kök ortada gezinme `B A C` yazar; kök önde gezinme `A B`, tersten gezinme `A B` yazar. C kaybolmuştur. İkinci durumda bayrak zaten `FALSE` olduğu için zarar olmaz.

12.11 İleri okuma

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, Kesim 5.5. Bu notun yapısı, `insert_right` yordamı ve başlık düğümü düzeni oradaki biçimiyle izlenmiştir.
- D. E. Knuth, *The Art of Computer Programming*, Cilt 1, Kesim 2.3.1. Kılavuzlu ağaçlar, ardıl bulma ve kılavuzlu ağaca ekleme algoritmaları kanıtlarıyla ve alıştırmalarla anlatılır.
- J. M. Morris, "Traversing binary trees simply and cheaply", *Information Processing Letters* 9(5), 1979. Geçici kılavuzlarla bayraksız gezinmenin özgün kısa makalesidir.

12.12 Kaynaklar

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. bs., Silicon Press, 2008. Kesim 5.5, Threaded Binary Trees.
2. A. J. Perlis, C. Thornton, "Symbol manipulation by threaded lists", *Communications of the ACM* 3(4), 1960.
3. J. M. Morris, "Traversing binary trees simply and cheaply", *Information Processing Letters* 9(5), 1979.
4. D. E. Knuth, *The Art of Computer Programming*, Cilt 1: Fundamental Algorithms, 3. bs., Addison-Wesley, 1997. Kesim 2.3.1.

KISIM III · BÖLÜM 13

Öncelikli Kuyruk ve Heap

Sıradan bir kuyrukta öğeler geliş sırasıyla çıkar. Pek çok gerçek sistemde ise ilk ele alınması gereken öğe, ilk gelen değil en acil olandır. Yakıtı en az kalan uçak, en kısa sürecek iş ve en yakın düğüm buna örnektir. Bu not, böyle bir kuyruğun işlemlerini tanımlar ve onu heap adlı yapıyla gerçekler. Heap, bir diziye yerleştirilmiş tam ikili ağaçtır. İşaretçi kullanmaz, ama ağaç gibi davranır. Not önce bu dizinin indis kurallarını kurar. Ardından ekleme ve silme yordamlarını C ile yazar, satır satır izler ve maliyetlerini gerekçesiyle verir. Heap'in dizi ve bağlı liste gibi basit yapılardan neden üstün olduğu bir maliyet tablosuyla ve gerçek ölçümlerle gösterilir. Son kesimler bir diziyi doğrusal zamanda heap'e çevirmeyi, d-ary heap'i, heap sort'u ve öncelikli kuyruğun kullanıldığı yerleri anlatır.

Kod (C): `github.com/sercankulcu/data-structures-and-algorithms-c/Bolum13`

ÖĞRENME ÇIKTILARI

Tanımlamak
Öncelikli kuyruğu ve maksimum/minimum heap'i tanımlamak; heap'i ikili arama ağacından ayırmak.

Gösterimi kurmak
Tam ikili ağacı 1 tabanlı bir dizide saklamak: ebeveyn `i/2` , çocuklar `2i` ve `2i+1` .

Ekleme ve silme
`maksimum_yigin_ekle` ve `maksimum_yigin_al` yordamlarını adım adım izlemek.

Analiz etmek
Ekleme ve silmenin neden $O(\lg n)$ olduğunu, en iyi durumun $\Omega(1)$ olduğunu göstermek.

Karşılaştırmak
Sırasız/sıralı dizi ve liste ile heap'i maliyet tablosu üzerinden karşılaştırmak.

Genişletmek
Aşağıdan yukarı heap kurmanın $O(n)$ olduğunu açıklamak; d-ary heap'i tanımak.

13.1 Öncelikli kuyruk

Kuyruk, öğelerin bir uçtan girip öbür uçtan çıktığı yapıdır. Bu düzen adildir: önce gelen önce hizmet alır. Ama her iş için doğru düzen bu değildir. Hastanenin acil servisinde hastalar geliş sırasına göre değil, durumlarının ağırlığına göre muayene edilir. Bu kesim, böyle bir düzeni tanımlayan veri yapısını ve ona neden gereksinim duyulduğunu anlatır.

13.1.1 Kuyruktan farkı

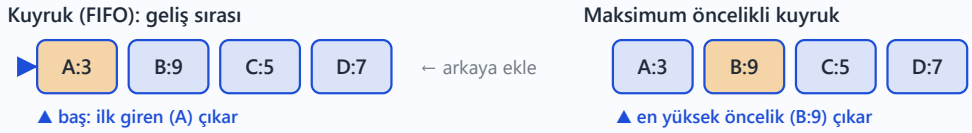
Sıradan kuyrukta (*queue*) iki işlem vardır. Ekleme yeni öğeyi kuyruğun arkasına koyar; silme kuyruğun başındaki öğeyi alır. Bu kurala **ilk giren ilk çıkar** (*first in, first out*, FIFO) denir. Öncelikli kuyrukta ise her öğenin bir **önceliği** vardır ve silme işlemi bu önceliğe bakar.

> **Tanım 13.1**
>
> **Öncelikli kuyruk** (*priority queue*), her öğesi bir öncelik değeri taşıyan ve ilk alınacak (silinecek) öğenin geliş sırasına göre değil, öncelik değerine göre belirlendiği kuyruktur. **Maksimum öncelikli kuyrukta** (*max priority queue*) önce en yüksek değerli öğe, **minimum öncelikli kuyrukta** (*min priority queue*) önce en düşük değerli öğe ele alınır.

Öncelik değerine çoğu kaynakta **anahtar** (*key*) denir. Bu notta iki sözcük aynı anlamda kullanılır. Hangi değerin "öncelikli" sayılacağı probleme bağlıdır. Bir yarışta en kısa süre en iyisidir; bir müzayedede en yüksek teklif. İki tür arasındaki fark yalnız karşılaştırmanın yönündedir. Bu yüzden not çoğunlukla maksimum öncelikli kuyruğu anlatır. Minimum sürüm, karşılaştırma işaretleri ters çevrilerek elde

edilir.

Şekil 13.1 farkı dört öğe üzerinde gösterir. Öğelerin adı bir harf, önceliği bir sayıdır: *A:3*, *B:9*, *C:5*, *D:7*. Dört öğe bu sırayla gelmiştir. Sıradan kuyruk ilk geleni, yani *A*'yı verir. Maksimum öncelikli kuyruk ise önceliği en yüksek olan *B*'yi verir; *B* ikinci gelmiş olsa da sırası beklenmez. İkinci silmede sıradan kuyruk *B*'yi, öncelikli kuyruk *D*'yi verir. Öncelikli kuyruğun bütün çıkış sırası *B, D, C, A*'dır; öğeler önceliklerinin azalan sırasıyla çıkar.



Şekil 13.1: Aynı dört öğe iki yapıda. Solda sıradan kuyruk: ekleme arkadan yapılır, ilk giren *A* çıkar. Sağda maksimum öncelikli kuyruk: en yüksek öncelikli *B:9* çıkar.

Tanım, öğelerin yapıda nasıl dizildiğinden söz etmez. Yalnız hangi öğenin ilk çıkacağını söyler. Bu önemli bir serbestliktir. Kuyruğu gerçekleyen yapı, öğeleri istediği gibi düzenleyebilir; yeter ki istendiğinde en öncelikli öğeyi bulup verebilsin. Notun büyük bölümü bu serbestliği en iyi kullanan düzeni, heap'i anlatır.

Sık yapılan hata

Öncelikli kuyruğun öğeleri her zaman sıralı tuttuğunu sanmak. Tanım yalnız ilk çıkacak öğeyi belirler. Öğeleri sıralı tutmak bu koşulu sağlamanın bir yoludur, ama en ucuz yolu değildir. Heap, öğeleri tam olarak sıralamadan yalnız en öncelikli olanı hazır tutar; bu ucuzluğun nedeni de budur.

13.1.2 Neden gerekir?

Sunum öncelikli kuyruğa duyulan gereksinimi iki örnekle açıklar. İkisinde de geliş sırası doğru kararı vermez.

Fotokopi makinesi. Bir fotokopi makinesini kullanan herkes sabit bir ücret öder. Üç sayfa çeken de otuz sayfa çeken de aynı parayı verir. Makinenin sahibi kazancını en büyük yapmak ister, yani belli bir sürede olabildiğince çok kullanıcıya hizmet vermek ister. Bunun yolu, kullanım süresi en kısa olan kullanıcıya öncelik vermektir. Üç sayfalık iş önce yapılırsa o kullanıcı hemen ayrılır ve ücretini öder; otuz sayfalık iş beklese de toplam kazanç artar. Burada ilk ele alınacak öğe en küçük değerli öğedir: bu bir **minimum** öncelikli kuyruktur.

Havaalanı. Bir havaalanına inmek için sıra bekleyen uçakları düşünelim. Kuleye önce gelen uçağın önce indirilmesi ilk bakışta adil görünür. Ama yakıtı bitmek üzere olan bir uçağa "kuyruğun arkasındasın, sıranı bekle" denemez. İniş izni, kalan yakıtı en az olan uçağa verilir. Bu da bir minimum öncelikli kuyruktur: anahtar kalan yakıt miktarıdır.

Gerçek sistemler de böyle çalışır. Bir işletim sistemi, çalışmaya hazır işlemleri (*process*) öncelikli bir kuyrukta tutar. Bir kesme geldiğinde ya da bir işlemin süresi dolduğunda, işlemciye kuyruktaki en öncelikli işlem alınır. Bu tür sistemlerde kuyruk sürekli değişir: yeni işler gelir, biten işler çıkar. Bu yüzden yapının hem eklemeyi hem en öncelikli öğeyi almayı hızlı yapması gerekir. Asıl zorluk, bu iki işlemi birlikte ucuz tutmaktır.

13.1.3 Soyut veri tipi

İşlemleri ve davranışları tanımlanan ama gerçeklemesi gizlenen tipe **soyut veri tipi** (*abstract data type*, ADT) denir. Öncelikli kuyruk da bir soyut veri tipidir. Sunum maksimum öncelikli kuyruğun arayüzünü şöyle verir:

```
/* maksimum öncelikli kuyruk arayüzü */
bool bos_mu(void);
bool dolu_mu(void);
void ekle(eleman item);     /* O(lg n) */
bool maks_al(eleman *cikan); /* sil + döndür */
bool maks_bak(eleman *ust);  /* silmeden bak */
```

Arayüzde beş işlem vardır. `bos_mu` ve `dolu_mu` kuyruğun durumunu sorar. Sabit kapasiteli bir gerçeklemede dolu kuyruğa ekleme yapılamaz; bu yüzden `dolu_mu` da arayüzdedir. `ekle` bir öğeyi önceliğiyle birlikte yapıya koyar. `maks_al` en öncelikli öğeyi hem döndürür hem yapıdan çıkarır; bu işleme sunumda *al* ya da *sil* denir. `maks_bak` ise en öncelikli öğeyi yapıdan çıkarmadan okur.

Alma ve bakma işlemleri öğeyi dönüş değeri olarak değil, bir işaretçi parametresi aracılığıyla verir. Dönüş değeri işlemin başarılı olup olmadığını söyleyen bir `bool` 'dur. Boş kuyrukta alınacak öğe yoktur. Böyle bir durumda öğe döndüren bir fonksiyon "anlamsız" bir öğe uydurmak zorunda kalırdı. `false` dönüşü ise çağırana açıkça "öğe yok" der. Öğeyi bir yapı olarak döndüren bir fonksiyonda "boş" durumunu bildirecek uygun bir değer de yoktur.

Arayüz, kuyruğun nasıl gerçekleneceğini söylemez. Öncelikli kuyruk sırasız bir diziyle, sıralı bir diziyle, bağlı bir listeyle ya da bir heap'le gerçeklenebilir. Hepsi aynı arayüzü sunar ve kullanan program farkı görmez. Fark yalnız maliyettedir. Basit yapılar işlemlerden birini çok ucuz, ötekini çok pahalı yapar. Heap ise iki işlemi de *O(lg n)* sürede yapar. Bu dengesi, heap'i öncelikli kuyruğun en yaygın gerçeklemesi yapar.

13.2 Heap yapısı

Heap iki kuralın birleşimidir. Birinci kural yapının *şeklini* belirler: heap bir tam ikili ağaçtır. İkinci kural değerlerin *düzenini* belirler: her düğüm çocuklarından büyük ya da onlara eşittir. Bu kesim önce şekil kuralını ve onun dizide nasıl saklandığını, sonra düzen kuralını anlatır.

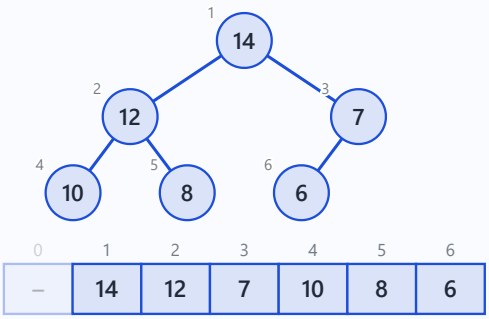
Bir terim notu: sunumdaki C kodlarında heap'i tutan dizinin adı `yigin` 'dir ve yordamlar `maksimum_yigin_ekle` gibi adlar taşır. Bu adlar ders kitabının Türkçe çevirisinden gelir. Bu derste *yığın* sözcüğü ise *stack* için kullanılır. Karışıklığı önlemek için not metinde yapının adını hep *heap* diye yazar; kodlardaki adlar olduğu gibi kalır. Heap sözcüğü `malloc` 'un bellek ayırdığı *heap bellek* ile de karıştırılmamalıdır. İkisi arasında, adın dışında bir bağ yoktur.

13.2.1 Dizide tam ikili ağaç

İkili ağaçta her düğümün en çok iki çocuğu vardır. Bir düğümün hemen üstündeki düğüme **ebeveyn** (*parent*) denir. Kökten aynı uzaklıktaki düğümler aynı **düzeydedir** (*level*). Bu notta kök 0. düzeydedir.

> **Tanım 13.2**
>
> **Tam ikili ağaç** (*complete binary tree*), son düzey dışındaki bütün düzeyleri dolu olan ve son düzeyindeki düğümleri soldan sağa doğru boşluksuz dizilmiş ikili ağaçtır.

Tam ikili ağacın şekli yalnız düğüm sayısına bağlıdır. 6 düğümlü bir tam ikili ağaç tek bir şekilde çizilebilir: kök, onun iki çocuğu ve son düzeyde soldan üç düğüm. Bu katılık bir kazanç sağlar. Düğümler düzey düzey, her düzeyde soldan sağa numaralanırsa her düğümün numarası ağacın şeklinden kesin olarak bellidir. Kök 1 numarayı, onun sol çocuğu 2'yi, sağ çocuğu 3'ü alır; sonraki düzeyin düğümleri 4, 5, 6, 7 diye devam eder. Bu numaralar doğrudan dizi indisi olarak kullanılabilir.



Şekil 13.2: Maksimum heap *14, 12, 7, 10, 8, 6*. Üstte ağaç görünümü; düğümlerin yanındaki küçük sayılar indislerdir. Altta aynı heap'in dizideki yerleşimi; 0. hücre kullanılmaz.

Şekil 13.2 altı düğümlü bir tam ikili ağacı ve onun dizideki karşılığını gösterir. Ağaçtaki her düğüm, numarasıyla aynı indisteki dizi hücresinde durur. Kökteki 14 `yigin[1]` 'de, kökün çocukları 12 ve 7 `yigin[2]` ile `yigin[3]` 'te, son düzeydeki 10, 8 ve 6 sırasıyla `yigin[4]` , `yigin[5]` ve `yigin[6]` 'dadır. Dizinin 0. hücresi kullanılmaz. Bu küçük israf, indis formüllerini sadeleştirmek içindir.

Numaralamanın en önemli sonucu, ebeveyn ile çocuklar arasındaki ilişkinin basit aritmetikle bulunmasıdır. *i*. düğümün sol çocuğu *2i*, sağ çocuğu *2i + 1*, ebeveyni ise *⌊i/2⌋* indisindedir. C'de iki tam sayının bölümü küçük tarafa yuvarlandığı için `i / 2` ifadesi bu tabanı doğrudan verir. Tablo 13.1 bu kuralları toplar.

Düğüm	İndis (1 tabanlı)	İndis (0 tabanlı)
kök	`1`	`0`
i'nin ebeveyni	`i / 2`	`(i - 1) / 2`
sol çocuk	`2i`	`2i + 1`
sağ çocuk	`2i + 1`	`2i + 2`
son öğe	`n`	`n - 1`

Tablo 13.1: 1 tabanlı dizide tam ikili ağacın indis kuralları (*n* öğe sayısıdır).

Kuralların neden doğru olduğu küçük bir sayımla görülür. *d*. düzeyde 2^d düğüm vardır ve bu düzeyin ilk düğümünün numarası 2^d'dir: 1, 2, 4, 8, ... *i* numaralı düğüm kendi düzeyinde soldan *k*. sırada olsun (0'dan sayarak); yani $i = 2^d + k$. Bu düğümün solundaki *k* düğümün her birinin iki çocuğu vardır. Bu yüzden onun sol çocuğu, bir alt düzeyde soldan *2k*. sıradadır. Alt düzeyin ilk numarası 2^{d+1} olduğundan sol çocuğun numarası $2^{d+1} + 2k = 2(2^d + k) = 2i$'dir. Sağ çocuk onun hemen sağındadır: $2i + 1$. Ebeveynin formülü bunların tersidir. $2i$ de $2i + 1$ de 2'ye bölününce taban olarak *i* verir.

Bazı kaynaklar diziyi 0. hücreden başlatır. O zaman bütün numaralar bir azalır ve formüller değişir: ebeveyn `(i - 1) / 2`, çocuklar `2i + 1` ve `2i + 2` olur. Her iki düzen de doğrudur; ama bir programda ikisi karıştırılmamalıdır. Sunum ve bu not, formülleri sade tutan 1 tabanlı düzeni kullanır.

Örnek 13.1

1 tabanlı bir heap'te 13. düğümün ebeveyni $\lfloor 13/2 \rfloor = 6$. düğümdür. Sol çocuğu $2 \cdot 13 = 26$., sağ çocuğu *27*. düğümdür. Bu çocuklar ancak heap'te yeterince öğe varsa bulunur: sol çocuk $n \geq 26$, sağ çocuk $n \geq 27$ ise vardır. 13. düğüm 3. düzeydedir, çünkü $8 \leq 13 < 16$'dır; kendi düzeyinde soldan 6. düğümdür ($13 = 8 + 5$, 0'dan sayınca 5). Aynı ağaç 0 tabanlı bir dizide tutulsaydı bu düğüm `a[12]` 'de olurdu; ebeveyni $(12 - 1)/2 = 5$, çocukları 25 ve 26 olurdu. İki düzende de aynı düğümler bulunur, yalnız numaraları birer kaymıştır.

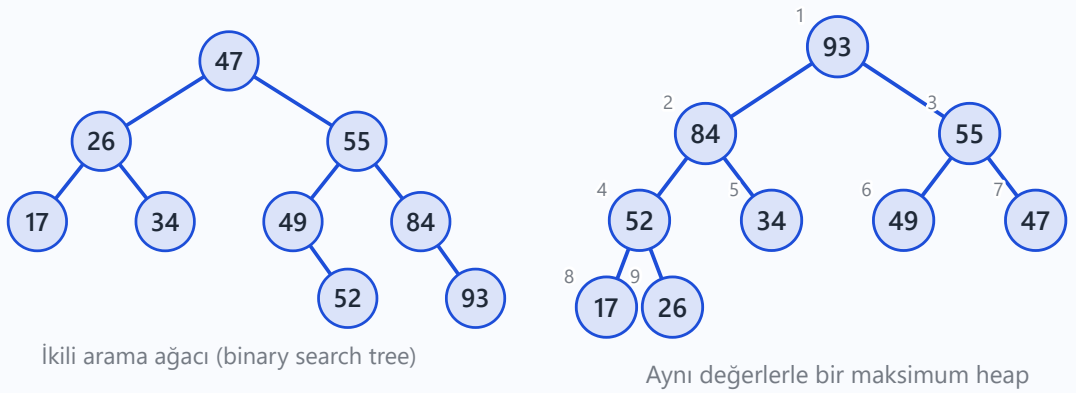
Bu gösterimin bedeli yoktur. Bağlı bir ağaçta her düğüm iki çocuk işaretçisi taşır; bir `int` anahtar için düğüm başına 8 ya da 16 bayt fazladan bellek harcanır. Dizide bu işaretçilere gerek kalmaz, çünkü tam ikili ağaçta düğümler arasında boşluk yoktur ve çocukların yeri hesapla bulunur. Dizinin hücreleri bellekte yan yana durduğu için işlemci önbelleği (*cache*) de bu yapıyla iyi çalışır. Kazanç tam ikili ağaca özgüdür. Seyrek, dengesiz bir ağaç bu yolla saklansaydı dizinin büyük kısmı boş kalırdı. Heap'te şekil kuralı tam da bu yüzden vardır.

Sık yapılan hata

1 tabanlı ve 0 tabanlı formülleri karıştırmak. Dizi 0. hücreden başlatılıp ebeveyni `i / 2` ile bulunursa kökün çocuğu sanılan 1. hücrenin ebeveyni yine 0. hücre çıkar, ama 2. hücrenin ebeveyni de 1. hücre çıkar. Oysa 0 tabanlı düzende 2. hücre kökün sağ çocuğudur. Sonuçta yanlış düğümler karşılaştırılır ve heap sessizce bozulur. Program çökmez; yalnız yanlış öğeyi verir. Böyle bir hatayı bulmak zordur.

13.2.2 Heap ikili arama ağacı değildir

Heap denince akla bir ikili arama ağacı değil, tam ikili ağaç gelmelidir. İki yapı da ikili ağaçtır ve ikisinde de değerler belli bir düzene uyar; ama düzenleri çok farklıdır. **İkili arama ağacında** (*binary search tree*) her düğümün sol alt ağacındaki değerler ondan küçük, sağ alt ağacındakiler ondan büyüktür: *sol < kök < sağ*. Heap'te ise yalnız *ebeveyn ≥ çocuk* kuralı vardır. Sol çocuk ile sağ çocuk arasında hiçbir sıra aranmaz.



Şekil 13.3: Aynı dokuz değer iki yapıda. Solda ikili arama ağacı: her düğümün solu küçük, sağı büyük; ağaç tam değil. Sağda maksimum heap: tam ikili ağaç ve her ebeveyn çocuklarından büyük.

Şekil 13.3 aynı dokuz değerin, *47, 26, 55, 17, 34, 49, 84, 52, 93*'ün iki yapıdaki yerleşimini gösterir. İkili arama ağacında kök 47'dir; 26 solda, 55 sağdadır. Ağacın şekli değerlerin geliş sırasına bağlıdır. Örnekte 49 ile 84'ün altında birer düğüm varken 17'nin hiç çocuğu yoktur; ağaç tam değildir. Heap'te kök en büyük değer olan 93'tür. Şekil tamdır: ilk üç düzey dolu, son düzeyde soldan iki düğüm vardır. Kökün sol çocuğu 84, sağ çocuğu 55'tir; bu iki değer arasında bir sıra kuralı yoktur.

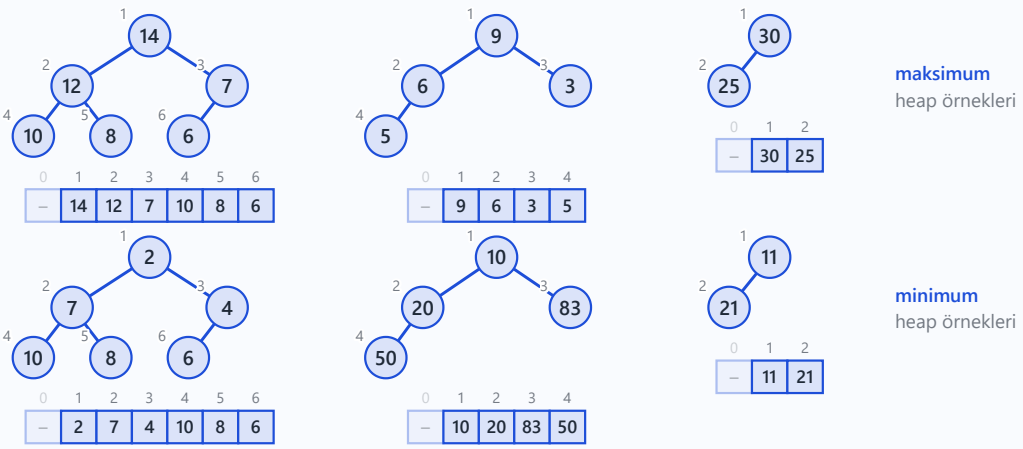
İki yapı farklı soruları cevaplar. İkili arama ağacında bir değeri aramak kolaydır: her düğümde sola ya da sağa gidilir. Heap'te bir değeri aramak için böyle bir yön yoktur; 34'ü bulmak için kökün iki alt ağacına da bakmak gerekebilir. Buna karşılık heap'in cevapladığı tek soru çok hızlı cevaplanır: en büyük değer her zaman köktedir. Heap bir arama yapısı değil, öncelikli kuyruk yapısıdır.

13.2.3 Maksimum ve minimum heap

Tanım 13.3

Maksimum heap (*max heap*), her düğümünün değeri çocuklarının değerinden küçük olmayan tam ikili ağaçtır. **Minimum heap** (*min heap*), her düğümünün değeri çocuklarının değerinden büyük olmayan tam ikili ağaçtır. Değerlere konan bu koşula **heap özelliği** denir.

Tanımdaki "küçük olmayan" ifadesi bilinçli seçilmiştir. Ebeveyn çocuğuna eşit olabilir; aynı önceliği taşıyan iki öğe heap'te ebeveyn ve çocuk olarak durabilir. Tanımın iki parçası ayrı ayrı yeterli değildir. Heap özelliğini sağlayan ama tam olmayan bir ağaç heap değildir; tam olup heap özelliğini sağlamayan ağaç da değildir. Bu tanımlar maksimum ve minimum öncelikli kuyruğu gerçeklemek için verilmiştir. Maksimum heap maksimum öncelikli kuyruğu, minimum heap minimum öncelikli kuyruğu gerçekler.



Şekil 13.4: Üstte üç maksimum heap, altta üç minimum heap; her birinin altında dizi gösterimi. İki öğelik heap'ler de geçerlidir.

Şekil 13.4 her türden üç örnek verir. Üstteki ilk maksimum heap notun ana örneğidir: *14, 12, 7, 10, 8, 6*. Her ebeveyn çocuklarından büyüktür: *14 ≥ 12*, *14 ≥ 7*, *12 ≥ 10*, *12 ≥ 8*, *7 ≥ 6*. Kökün sağ çocuğu 7, sol alt ağaçtaki 10 ve 8'den küçüktür. Bu bir sorun değildir; 7 ile 10 arasında ebeveyn–çocuk ilişkisi yoktur. Alttaki minimum heap'lerde kural terstir. İlk örnek *2, 7, 4, 10, 8, 6*'da her ebeveyn çocuklarından küçüktür ve kökte en küçük değer, 2 durur.

Heap özelliği yalnız komşu düzeyleri bağlar. Ama zincirleme olarak çok daha güçlü bir sonuç verir: kökten bir yaprağa giden her yolda değerler azalmayan değil, artmayan bir dizi oluşturur. Bundan da kökün bütün ağacın en büyük değeri olduğu çıkar.

Teorem 13.1 (Kökteki öğe)

Boş olmayan bir maksimum heap'te her düğümün değeri, atalarının hepsinin değerinden büyük değildir. Özellikle kökteki değer, heap'teki en büyük değerdir.

Kanıt. *i* numaralı bir düğüm alınsın. Ebeveyni *⌊i/2⌋*, onun ebeveyni *⌊i/4⌋*, … diye gidilirse sonlu adımda köke, 1 numaraya varılır, çünkü her adımda numara en az yarıya iner. Bu zincirdeki her ardışık çift bir ebeveyn–çocuk çiftidir ve heap özelliği gereği ebeveynin değeri çocuğunkinden küçük değildir. Eşitsizlikler zincirlenince *yigin[1] ≥ … ≥ yigin[⌊i/2⌋] ≥ yigin[i]* elde edilir. *i* herhangi bir düğüm olduğundan kök bütün değerlerden küçük değildir. ■

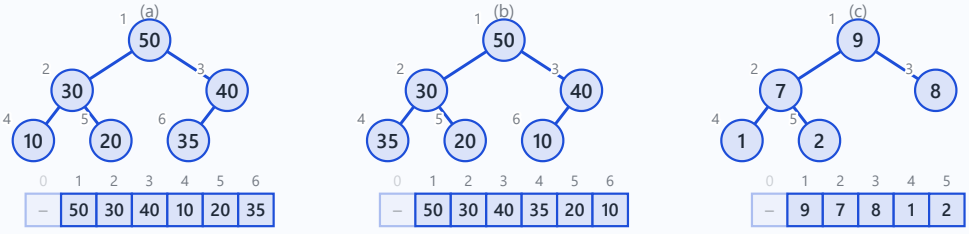
Teorem öncelikli kuyruğun en sık sorusunu, “en büyük öğe hangisi?” sorusunu *O(1)*'de cevaplar: `yigin[1]`. Dizinin geri kalanı ise sıralı değildir. Örnek heap'te ikinci büyük değer 12, 2. indistedir; ama üçüncü büyük değer 10, 3. indiste değil 4. indistedir. Garanti yalnız 1. indis içindir. İkinci büyük değer her zaman kökün çocuklarından biridir. Üçüncü büyük ise kökün öbür çocuğu da olabilir, büyük çocuğun bir çocuğu da.

13.2.4 Heap mi, değil mi?

Bir dizinin maksimum heap olup olmadığını anlamak için bütün ebeveyn–çocuk çiftlerine bakmak yeter. Dizi gösteriminde şekil kuralı kendiliğinden sağlanır, çünkü *1..n* indisleri boşluksuz doludur. Geriye yalnız düzen kuralı kalır: her *k ≥ 2* için `yigin[k/2] ≥ yigin[k]` olmalıdır. Kardeşler ya da kuzenler arasında bir sıra aranmaz. Aşağıdaki yardımcı fonksiyon bu sınamayı yapar:

```
bool heap_mi(eleman a[], int n) {
  for (int k = 2; k <= n; k++)
    if (a[k / 2].x < a[k].x) return false;  /* ebeveyn < çocuk */
  return true;
}
```

Döngü 2'den *n*'e kadar her düğümü bir kez ebeveyniyle karşılaştırır. Kök dışındaki her düğümün tek bir ebeveyni olduğu için bütün ebeveyn–çocuk çiftleri tam bir kez denetlenir. İlk ihlalde `false` döner. Maliyet en çok *n − 1* karşılaştırmadır, yani *O(n)*. Bu fonksiyon heap'in parçası değildir; ama yazılan yordamları sınamak için çok kullanışlıdır. Bu notun bütün sayıları, her işlemden sonra bu sınamayı yapan test programlarıyla doğrulanmıştır.



Şekil 13.5: Üç aday dizi. Hangisi maksimum heap'tir?

Örnek 13.2

Şekil 13.5 üç diziyi gösterir. (a) *50, 30, 40, 10, 20, 35* bir maksimum heap'tir. Ebeveyn–çocuk çiftleri *50 ≥ 30*, *50 ≥ 40*, *30 ≥ 10*, *30 ≥ 20* ve *40 ≥ 35*'tir; hepsi sağlanır. 35'in 30'dan büyük olması önemli değildir; 35'in ebeveyni 40'tır.

(b) *50, 30, 40, 35, 20, 10* maksimum heap değildir. 4. indisteki 35'in ebeveyni *⌊4/2⌋ = 2*. indisteki 30'dur ve *30 < 35*'tir. `heap_mi` bu dizide *k = 4*'te durup `false` döner.

(c) *9, 7, 8, 1, 2* bir maksimum heap'tir. Ağaç tamdır ve her ebeveyn çocuklarından büyüktür: $9 \geq 7$, $9 \geq 8$, $7 \geq 1$, $7 \geq 2$. 1 ile 2'nin 8'den küçük olması, yani sağ alt ağaçtaki bir değerin sol alt ağaçtakilerden büyük olması önemli değildir. `heap_mi` üç dizi için sırasıyla `true`, `false`, `true` döndürmüştür.

13.2.5 C tanımları

Sunumdaki ilk gerçekleme, heap'i bir global dizi ve bir öğe sayacıyla tutar. Tanımlar şöyledir:

```
#define ELEMAN_SAYISI 200   /* maksimum öğe + 1 */
#define YIGIN_DOLUMU(n) ((n) == ELEMAN_SAYISI - 1)
#define YIGIN_BOSMU(n)  (!(n))

typedef struct {
  int x;          /* öncelik (anahtar) */
  /* diğer alanlar */
} eleman;

eleman yigin[ELEMAN_SAYISI];  /* [0] kullanılmaz */
int n = 0;                    /* öğe sayısı */
```

`ELEMAN_SAYISI` dizinin boyutudur. 0. hücre kullanılmadığı için heap en çok *ELEMAN_SAYISI − 1 = 199* öğe alır; açıklamadaki "maksimum öğe + 1" bu yüzdendir. Kullanılan indisler 1..199'dur. `YIGIN_DOLUMU(n)` makrosu öğe sayısının bu sınıra ulaşıp ulaşmadığını, `YIGIN_BOSMU(n)` ise öğe sayısının sıfır olup olmadığını sınar. `!(n)` ifadesi *n* sıfırsa 1, değilse 0 verir.

Öğe tipi `eleman` bir yapıdır. Tek alanı `x`, önceliği tutar. Gerçek bir uygulamada yapıda başka alanlar da bulunur: bir işlemin numarası, bir uçağın kuyruk kodu gibi. Heap yordamları yalnız `x`'e bakar; öteki alanlar öğeyle birlikte taşınır. Bu yüzden yordamlar öğeleri tam sayı olarak değil, bütün yapı olarak kopyalar. `yigin[i] = yigin[j]` ataması yapının bütün alanlarını bir kerede kopyalar. Bu notun örneklerinde yapı tek bir `int` taşıdığı için 4 bayttır.

Makrolarda parametrenin parantez içinde yazılması bir alışkanlık değil, zorunluluktur. Makro bir fonksiyon değildir; önişlemci metni olduğu gibi yerine koyar. `#define YIGIN_BOSMU(n) (!n)` yazılmış olsun. `YIGIN_BOSMU(a + b)` çağrısı `(!a + b)`'ye açılır. `!` işleci `+`'dan önce uygulandığı için bu ifade *(!a) + b* olarak hesaplanır; `a = 0`, `b = 5` için sonuç 6'dır ve "boş" anlamına gelir. Doğru tanımla ifade `(!(a + b))` olur ve 0 verir. Dış parantez de aynı amaçla konur: makronun sonucu daha büyük bir ifadenin içinde kullanıldığında bütün olarak kalmalıdır.

Sık yapılan hata

Diziyi `eleman yigin[ELEMAN_SAYISI - 1]` diye, yani 199 öğelik (indis 0..198) tanımlamak. `YIGIN_DOLUMU` heap'i ancak *n = 199* olunca dolu sayar. 199. eklemede *n* 199'a çıkar ve öğe `yigin[199]`'a yazılır; bu hücre dizinin dışındadır. Bu hata bir test programıyla denenmiştir: 199. ekleme dizinin hemen arkasındaki belleğe yazmış ve program hiçbir uyarı vermeden devam etmiştir. Tanımsız davranış denetleyicisi (`-fsanitize=undefined`) yalnız aynı hücre okunurken "index 199 out of bounds for type 'eleman[199]'" iletisiyle durmuştur. Dizinin boyutu `ELEMAN_SAYISI` olmalıdır.

Global bir dizi ve global bir sayaç basit bir başlangıçtır, ama sınırları vardır. Programda yalnız bir heap olabilir. Kapasite derleme sırasında sabitlenir. Yordamlar hangi diziyle çalıştıklarını parametre olarak almaz. Bu sakıncalar bir yapı kullanılarak giderilebilir. Ama yordamların mantığı en sade biçimde global dizi üzerinde görülür; ekleme ve silme önce bu biçimde yazılır.

13.3 Ekleme

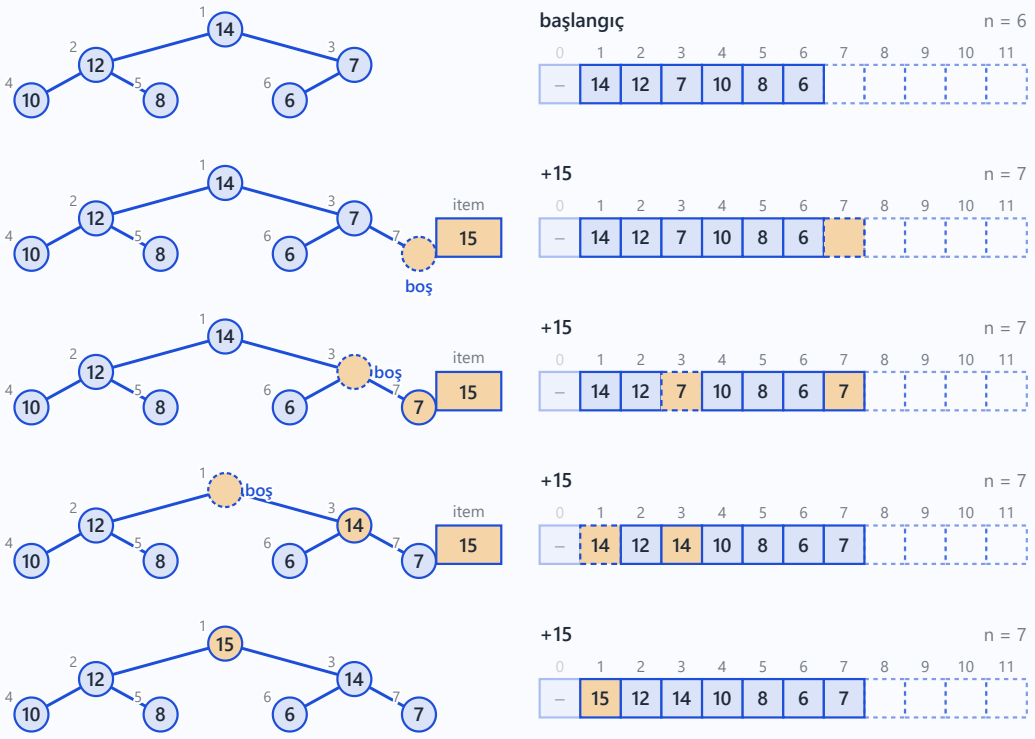
Heap'e yeni bir öğe eklenince iki kural birden korunmalıdır. Ağaç tam kalmalı ve her ebeveyn çocuklarından küçük olmamalıdır. Ekleme bu iki kuralı sırayla ele alır: önce şekli, sonra düzeni düzeltir.

13.3.1 Önce yer açılır, sonra kaydırılır

Tam ikili ağaca bir düğüm eklenecekse, yeni düğümün nereye geleceği bellidir: son düzeydeki ilk boş konum. Son düzey doluysa yeni bir düzey açılır ve en soldaki konum kullanılır. Dizi gösteriminde bu konumun indisi *n + 1*'dir. Yeni öğenin açılacağı konum bu yüzden kesin olarak bellidir. Önce bu konum açılır; ağaç yine tamdır.

Ama yeni öğe ebeveyninden büyük olabilir. O zaman heap özelliği bu ebeveyn–çocuk çiftinde bozulmuştur. Düzeltmek için yeni öğe yukarı doğru, uygun konumuna kaydırılır. Ebeveyni yeni öğeden küçükse ebeveyn bir düzey aşağı iner, yeni öğenin aday konumu bir düzey yukarı çıkar. Bu, yeni öğeden küçük olmayan bir ebeveyne ya da köke ulaşılana kadar sürer. Yukarı doğru bu harekete **yukarı kaydırma** (*up-heap*, yukarı süzme) denir.

Şekil 13.6 15'in *14, 12, 7, 10, 8, 6* heap'ine eklenişini beş anda gösterir. Başlangıçta heap'te altı öğe vardır. İlk boş konum 7'dir; bu konum 3. düğümün, yani 7 değerinin sağ çocuğudur. Yeni öğe 15, 7'den büyüktür; bu yüzden 7 bir düzey aşağı, 7. konuma kaydırılır ve boşluk 3. konuma çıkar. 3. konumun ebeveyni kökteki 14'tür. 15 bundan da büyüktür; 14 aşağı, 3. konuma iner ve boşluk 1. konuma, köke çıkar. Kökün ebeveyni yoktur. 15 köke yazılır ve heap özelliği yeniden sağlanır. Son durum *15, 12, 14, 10, 8, 6, 7*'dir.



Şekil 13.6: 15'in eklenmesi. Yukarıdan aşağıya: başlangıç; ilk boş konumun (7) açılması; 7'nin aşağı kaydırılması; 14'ün aşağı kaydırılması; 15'in köke yazılması. Turuncu daire boşluğu, sağdaki kutu yerleşmeyi bekleyen `item`'ı gösterir.

Hareketin yalnız tek bir yol üzerinde olduğuna dikkat edilmelidir. Yeni konumdan köke giden yol 7 → 3 → 1'dir. Heap'in geri kalanına, örneğin 12'nin alt ağacına hiç dokunulmaz. Bu, eklemenin ucuzluğunun kaynağıdır. Yolun uzunluğu ağacın yüksekliği kadardır ve tam ikili ağacın yüksekliği öğe sayısının logaritmasıyla büyür.

Yukarı kaydırma heap özelliğini neden korur? Yol üzerinde aşağı inen her ebeveyn, yeni öğeden küçüktür. İndiği konumun çocukları, eski konumunda zaten onun torunlarıydı; heap özelliği gereği onlardan küçük değildir. Yani aşağı inen ebeveyn yeni yerinde de kurala uyar. Yeni öğe ise durduğu yerde ebeveyninden büyük değildir. Çocuklarından da küçük değildir. Çocuklarından biri, yerinden aşağı kaydırdığı ebeveyndir; öbürü o ebeveynin eski çocuğudur. İkisi de yeni öğeden küçüktür. Böylece ekleme sonunda bütün ebeveyn–çocuk çiftleri yeniden kurala uyar.

13.3.2 `maksimum_yigin_ekle`: takas değil, kaydırma

Sunumdaki ekleme yordamı şöyledir:

```
void maksimum_yigin_ekle(eleman item) {
  if (YIGIN_DOLUMU(n)) {
    yigin_dolu();             /* hata iletisi */
    return;
  }
  int i = ++n;
  while (i != 1) {            /* i == 1: kökteyiz */
    if (item.x <= yigin[i / 2].x) break;
    yigin[i] = yigin[i / 2];  /* ebeveyni aşağı kaydır */
    i /= 2;                   /* bir üst ebeveyne bak */
  }
  yigin[i] = item;
}
```

Yordam önce heap'in dolu olup olmadığına bakar. Doluysa `yigin_dolu()` çağrılır ve ekleme yapılmadan dönülür. `yigin_dolu` bir hata iletisi yazan basit bir yordamdır; sunumda gövdesi verilmez. Test programlarında standart hata akışına "heap dolu" yazan tek satırlık bir fonksiyon kullanılmıştır.

`int i = ++n;` satırı iki iş yapar. `++n` önce öğe sayısını bir artırır, sonra artmış değeri verir; `i` bu yüzden ilk boş konumun, yani yeni son konumun indisini alır. Bu satırdan sonra `yigin[i]`'de henüz anlamlı bir değer yoktur. Yeni öğe hemen oraya yazılmaz, çünkü belki de yeri orası değildir. `i`, boşluğun o anki konumunu gösterir.

Döngü koşulu `i != 1`'dir. `i` 1 olduğunda boşluk köke ulaşmıştır ve kökün ebeveyni yoktur; daha yukarı çıkılamaz. Döngünün ilk satırı yeni öğeyi boşluğun ebeveyniyle karşılaştırır. `item.x <= yigin[i / 2].x` ise ebeveyn yeterince büyüktür, boşluk yeni öğenin yeridir ve döngü `break` ile biter. Değilse ebeveyn boşluğa, yani bir düzey aşağı kopyalanır: `yigin[i] = yigin[i / 2]`. Ardından `i /= 2` ile boşluk ebeveynin eski konumuna çıkar. Döngü bittiğinde `i` yeni öğenin yerini gösterir ve `yigin[i] = item` öğeyi oraya yazar.

Yordamın adındaki fikir dikkat ister: takas yoktur. Birçok kaynak yukarı kaydırmayı, yeni öğeyi önce sona yazıp sonra ebeveyniyle yer değiştirerek anlatır. Her yer değiştirme bir geçici değişken ve üç atama ister. Buradaki **boşluk (delik) yöntemi** ise yeni öğeyi yol boyunca taşımaz. Yalnız büyük atalar birer düzey aşağı kaydırılır; her kaydırma tek bir atamadır. Yeni öğe en sonda bir kez yazılır. *k* düzey çıkan bir eklemede takasla *3k* atama yapılırken bu yöntemle *k + 1* atama yapılır. Öğe tipi büyük bir yapı olduğunda bu fark daha da önemlidir.

> **Sık yapılan hata**
>
> Döngü koşulunu `i > 0` ya da `i >= 1` yazmak. `i` 1 iken döngüye girilirse `yigin[i / 2]`, yani kullanılmayan `yigin[0]` okunur. 0. hücrede anlamsız bir değer vardır. Bu değer yeni öğeden küçükse `yigin[1] = yigin[0]` ataması o anlamsız değeri köke kopyalar, `i` 0 olur ve yeni öğe `yigin[0]`'a, yani heap'in dışına yazılır. Kökte anlamsız bir değer kalır, yeni öğe ise kaybolur. Kökte durmak için koşul `i != 1` (ya da `i > 1`) olmalıdır.

13.3.3 Kod izleme: `ekle(15)`

Tablo 13.2, Şekil 13.6'daki eklemeyi kodun satırlarıyla izler. Satır numaraları sunumdaki izleme slaytına göredir: 2. satır dolu denetimi, 6. satır `int i = ++n;`, 7. satır döngü koşulu, 8. satır karşılaştırma, 9. satır kaydırma, 10. satır `i /= 2`, 12. satır son yazmadır. Başlangıçta heap *14, 12, 7, 10, 8, 6*, *n = 6*'dır.

Satır	`n`	`i`	`yigin[i/2]`	Koşul / işlem	`yigin[1..7]`
2	6	—	—	`YIGIN_DOLUMU(6)` yanlış: yer var	14, 12, 7, 10, 8, 6, —
6	7	7	7	`i = ++n`: ilk boş konum	14, 12, 7, 10, 8, 6, _
7	7	7	7	*7 ≠ 1*: döngüye girilir	14, 12, 7, 10, 8, 6, _
8	7	7	7	*15 ≤ 7*? Hayır	14, 12, 7, 10, 8, 6, _
9	7	7	7	`yigin[7] = yigin[3]`	14, 12, _, 10, 8, 6, 7
10	7	3	14	`i /= 2`	14, 12, _, 10, 8, 6, 7
7, 8	7	3	14	*3 ≠ 1*; *15 ≤ 14*? Hayır	14, 12, _, 10, 8, 6, 7
9	7	3	14	`yigin[3] = yigin[1]`	_, 12, 14, 10, 8, 6, 7
10	7	1	—	`i /= 2`	_, 12, 14, 10, 8, 6, 7

Satır	`n`	`i`	`yigin[i/2]`	Koşul / işlem	`yigin[1..7]`
7	7	1	—	*i = 1*: köke ulaşıldı, döngü biter	_, 12, 14, 10, 8, 6, 7
12	7	1	—	`yigin[1] = item`	15, 12, 14, 10, 8, 6, 7

Tablo 13.2: `maksimum_yigin_ekle(15)` izlemesi. Dizi sütunu `yigin[1..7]` 'yi gösterir; `_` boşluğu belirtir.

Tablodaki "boşluk" bir gösterim kolaylığıdır. Bellekte o hücre boş değildir; içinde eski değerin bir kopyası durur. `yigin[7] = yigin[3]` satırından sonra 7 değeri hem 3. hem 7. hücrededir. 3. hücredeki kopya artık anlamsızdır, çünkü bir sonraki adımda üzerine yazılacaktır. İzleme bu yüzden o hücreyi boşluk olarak gösterir. Aynı durum 14 için de olur: `yigin[3] = yigin[1]`'den sonra 14 iki hücrededir ve kökteki kopya en son satırda 15 ile ezilir.

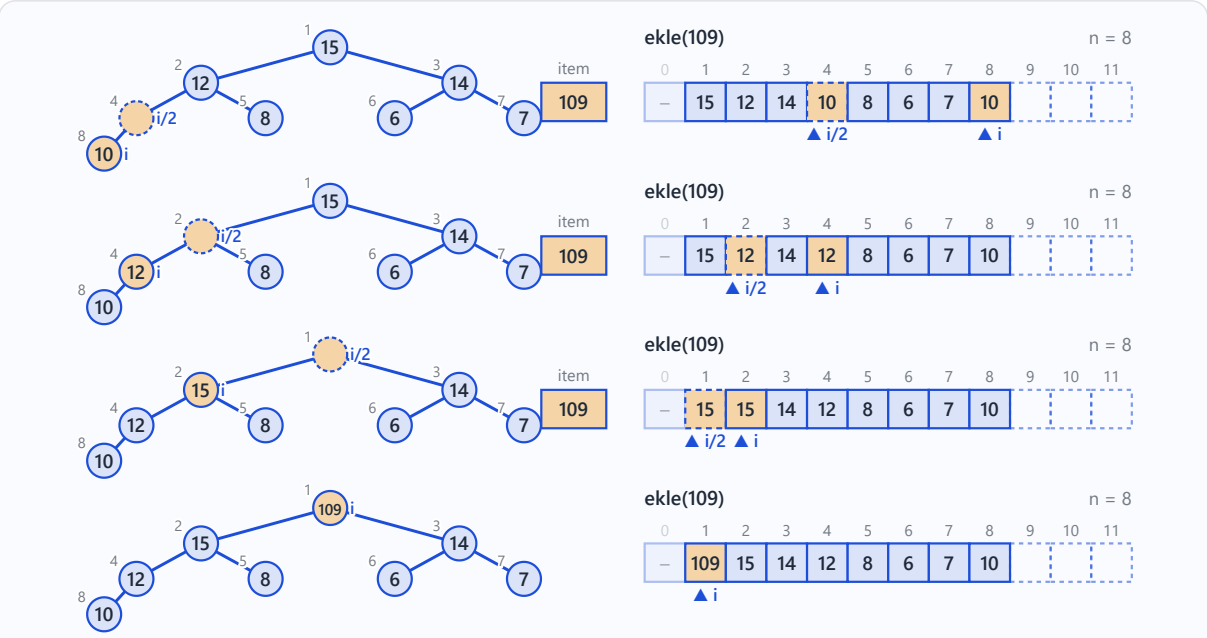
Döngü iki kez tam döner ve üçüncü denetimde `i == 1` olduğu için biter. Toplam iki karşılaştırma, iki kaydırma ve bir son yazma yapılır. Yeni öğe yol üzerinde iki düzey yukarı çıkmıştır. Kökte durması bir rastlantı değildir: 15 heap'teki her değerden büyüktür ve Teorem 13.1 gereği en büyük değer köke çıkmak zorundadır.

13.3.4 Kod izleme: `ekle(109)`

İkinci eklemede heap *15, 12, 14, 10, 8, 6, 7, n = 7*'dir ve 109 eklenir. 109 bütün öğelerden büyüktür; bu yüzden bu ekleme en kötü durumdur ve öğe köke kadar çıkacaktır. Son düzey (4, 5, 6, 7) doludur. Bu yüzden yeni konum 8, yeni bir düzeyin ilk konumudur ve 4. düğümün, yani 10'un sol çocuğudur.

Tur	`i`	`i/2`	Karşılaştırma	İşlem	`yigin[1..8]` (tur sonunda)
başlangıç	8	4	—	`i = ++n = 8`	15, 12, 14, 10, 8, 6, 7, _
1	8	4	*109 ≤ 10*? Hayır	`yigin[8] = yigin[4]`, `i = 4`	15, 12, 14, _, 8, 6, 7, 10
2	4	2	*109 ≤ 12*? Hayır	`yigin[4] = yigin[2]`, `i = 2`	15, _, 14, 12, 8, 6, 7, 10
3	2	1	*109 ≤ 15*? Hayır	`yigin[2] = yigin[1]`, `i = 1`	_, 15, 14, 12, 8, 6, 7, 10
son	1	—	*i = 1*: döngü biter	`yigin[1] = item`	109, 15, 14, 12, 8, 6, 7, 10

Tablo 13.3: `maksimum_yigin_ekle(109)` izlemesinin özeti. Her satır döngünün bir turudur.



Şekil 13.7: `ekle(109)` sırasında dört an. Yukarıdan aşağıya: 10'un, 12'nin ve 15'in birer düzey aşağı kaydırılması; 109'un köke yazılması. *i* boşluğun, *i/2* karşılaştırılan ebeveynin yerini gösterir.

Döngü üç kez tam döner: yeni konumdan köke kadar yol 8 → 4 → 2 → 1'dir ve bu yolda üç kenar vardır. Her turda bir karşılaştırma ve bir kaydırma yapılır. Dördüncü denetimde `i == 1` olur ve döngü karşılaştırma yapmadan biter. Yolun üzerindeki üç öğe, 10, 12 ve 15, birer düzey aşağı inmiştir. Yolun dışındaki öğeler, 14, 8, 6 ve 7, yerinden kıpırdamamıştır. Bu ekleme 8 öğeli bir heap'te yapılabilecek en büyük işi yapmıştır. 8 öğeli tam ikili ağacın dört düzeyi vardır ve yeni öğe en alt düzeyden köke çıkmıştır.

13.3.5 Art arda ekleme

Sunumun ekleme örneği beş değerden oluşur: *14, 12, 7, 10, 8, 6* heap'ine sırasıyla 15, 109, 107, 3 ve 15 eklenir. İlk ikisi yukarıda izlendi. Kalan üç ekleme farklı davranışlar gösterir; Tablo 13.4 her eklemenin yolunu ve sonucunu verir.

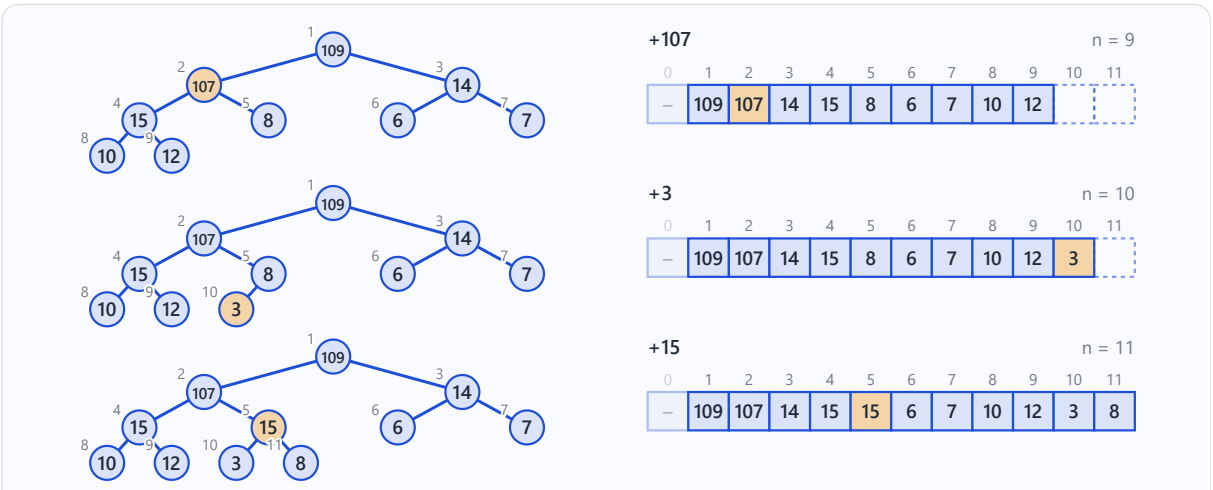
Ekleme	Yeni konum	Yol	Kaydırma	Heap (sonra)
+15	7	7 → 3 → 1	2	15, 12, 14, 10, 8, 6, 7
+109	8	8 → 4 → 2 → 1	3	109, 15, 14, 12, 8, 6, 7, 10
+107	9	9 → 4 → 2	2	109, 107, 14, 15, 8, 6, 7, 10, 12
+3	10	10	0	109, 107, 14, 15, 8, 6, 7, 10, 12, 3
+15	11	11 → 5	1	109, 107, 14, 15, 15, 6, 7, 10, 12, 3, 8

Tablo 13.4: Art arda beş ekleme. “Yol” yeni konumdan durulan konuma kadar boşluğun geçtiği indislerdir.

107'nin eklenmesinde ilk boş konum 9'dur; ebeveyni 4. konumdaki 12'dir. *107 > 12* olduğu için 12 aşağı, 9. konuma kayar ve boşluk 4'e çıkar. 4'ün ebeveyni 2. konumdaki 15'tir; *107 > 15* olduğu için 15 de aşağı kayar ve boşluk 2'ye çıkar. 2'nin ebeveyni kökteki 109'dur ve *107 ≤ 109*'dur. Ebeveyn yeterince büyüktür, döngü durur ve 107 2. konuma yazılır. Bu ekleme köke kadar çıkmadan yolun ortasında durmuştur.

3'ün eklenmesinde yeni konum 10'dur ve ebeveyni 5. konumdaki 8'dir. *3 ≤ 8* olduğu için döngü daha ilk karşılaştırmada `break` ile biter. Hiç kaydırma yapılmaz ve 3 doğrudan 10. konuma yazılır. Bu, eklemenin en iyi durumudur: bir karşılaştırma ve bir yazma, yani heap ne kadar büyük olursa olsun sabit iş.

Son olarak ikinci 15 eklenir. Yeni konum 11'dir; ebeveyni yine 5. konumdaki 8'dir. *15 > 8* olduğu için 8 aşağı, 11. konuma kayar ve boşluk 5'e çıkar. 5'in ebeveyni 2. konumdaki 107'dir ve *15 ≤ 107*'dir. 15, 5. konuma yazılır. Heap'te artık iki tane 15 vardır: biri 4., biri 5. konumda. Bu iki öğe kardeştir ve aralarında sıra kuralı yoktur. Tanımdaki “küçük olmayan” ifadesi eşit öncelikli öğelere de izin verir.

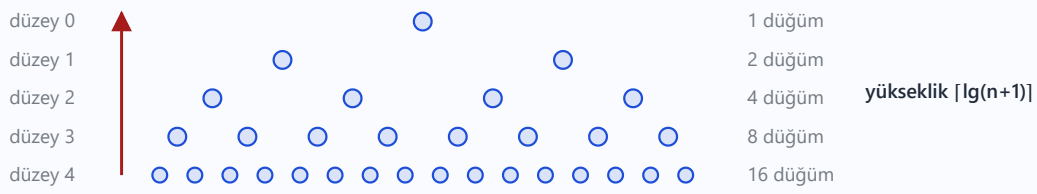


Şekil 13.8: 107, 3 ve 15 eklendikten sonraki üç durum. Turuncu düğüm, son eklenen öğenin yerleştiği konumdur.

Şekil 13.8 bu üç eklemenin sonuçlarını gösterir. Beş eklemedeki kaydırma sayıları 2, 3, 2, 0 ve 1'dir. 3'ün eklenmesi hiç kaydırma yapmamıştır; bu en iyi durumdur ve sabit sürer. 109 ise köke kadar çıkmıştır; bu en kötü durumdur ve yükseklikle orantılı sürer. Aynı yordamın maliyeti, eklenen değerin büyüklüğüne bağlı olarak bu iki uç arasında değişir.

13.3.6 Eklemenin maliyeti

Eklemenin maliyeti döngünün tur sayısıyla belirlenir. Her tur sabit iş yapar: bir karşılaştırma, bir kaydırma ve bir bölme. Tur sayısı ise boşluğun kaç düzey yukarı çıktığıdır. Boşluk en alttan köke kadar çıkabilir; o hâlde tur sayısı en çok ağacın yüksekliği kadardır. Geriye tam ikili ağacın yüksekliğini bulmak kalır.



Şekil 13.9: Tam ikili ağacın düzeyleri. Her düzeyde bir öncekinin iki katı düğüm vardır: 1, 2, 4, 8, 16. Ok, yükseklik boyunca en uzun yolu gösterir.

> **Teorem 13.2 (Tam ikili ağacın yüksekliği)**
>
> $n \geq 1$ düğümlü bir tam ikili ağacın düzey sayısı $\lfloor \lg n \rfloor + 1 = \lceil \lg(n+1) \rceil$'dir. Bir yapraktan köke giden yolda en çok $\lfloor \lg n \rfloor$ kenar vardır.

Kanıt. Ağacın düzeyleri 0'dan h'ye kadar numaralansın. 0'dan $h-1$'e kadarki düzeyler doludur ve $1 + 2 + \dots + 2^{h-1} = 2^h - 1$ düğüm taşır. Son düzeyde en az 1, en çok 2^h düğüm vardır. Bu yüzden $2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1}$'dir. Logaritma alınınca $h \leq \lg n < h+1$, yani $h = \lfloor \lg n \rfloor$ bulunur. Düzey sayısı $h+1$'dir. İkinci eşitlik için $2^h < n+1 \leq 2^{h+1}$ eşitsizliğinden $\lceil \lg(n+1) \rceil = h+1$ çıkar. Kökten en alt düzeydeki bir düğüme giden yolda h kenar vardır. ■

Dizi gösteriminde bu sonucun basit bir karşılığı vardır. i indisli düğümün düzeyi $\lfloor \lg i \rfloor$'dir ve her `i /= 2` düzeyi bir azaltır. n indisinden 1'e inmek için tam $\lfloor \lg n \rfloor$ kez ikiye bölmek gerekir. Örneğin $n = 8$ için $8 \to 4 \to 2 \to 1$ üç bölmedir ve $\lfloor \lg 8 \rfloor = 3$'tür. 109'un eklenmesi tam bu kadar kaydırma yapmıştı.

Durum	Örnek	Kaydırma	Maliyet
En iyi	3'ün eklenmesi	0	$\Omega(1)$
En kötü	109'un eklenmesi	yükseklik kadar: $\lfloor \lg n \rfloor$	$O(\lg n)$
Genel	—	$0 \dots \lfloor \lg n \rfloor$	$O(\lg n)$

Tablo 13.5: Eklemenin maliyeti.

Tablo 13.5 sonucu toplar. Eklemenin zaman karmaşıklığı $O(\lg n)$'dir; en iyi durumu ise $\Omega(1)$'dir. Burada O üst sınırı, Ω alt sınırı anlatır. Bir işlemin en iyi ve en kötü durumu aynı büyüklükteyse maliyet Θ ile, yani sıkı sınırla yazılır. Eklemede iki uç farklı olduğu için tek bir Θ yazılamaz; en kötü durum ise $\Theta(\lg n)$'dir. Ek bellek $O(1)$'dir: yalnız `i` değişkeni ve `item` parametresi kullanılır.

Bu sayıların ölçekte anlamı büyüktür. Bir milyon öğeli bir heap'te $\lfloor \lg 10^6 \rfloor = 19$'dur; en kötü eklemede bile 19 kaydırma yapılır. Bir milyar öğede bu sayı 29'dur. Öğe sayısı bin kat artarken iş yalnız 10 kaydırma artar.

Rastgele sırada gelen değerlerde ekleme çoğu zaman en kötü duruma hiç yaklaşmaz. Bir deneme, rastgele bir milyon değeri boş bir heap'e tek tek eklemiş ve toplam 1.279.942 kaydırma saymıştır; ekleme başına ortalama 1,3 kaydırma. Aynı sayılar artan sırada, yani her yeni değer öncekilerin hepsinden büyük olacak biçimde eklenince toplam 17.951.445 kaydırma olmuştur. Bu, her eklemenin köke kadar çıktığı en kötü durumdur ve toplam $\lfloor \lg 1 \rfloor + \lfloor \lg 2 \rfloor + \dots + \lfloor \lg n \rfloor$'ye eşittir. Rastgele bir değerin büyük olasılıkla yaprak düzeyinde kalmasının nedeni basittir: tam ikili ağaçta düğümlerin yaklaşık yarısı yapraktır, dörtte biri yaprakların bir üstündedir.

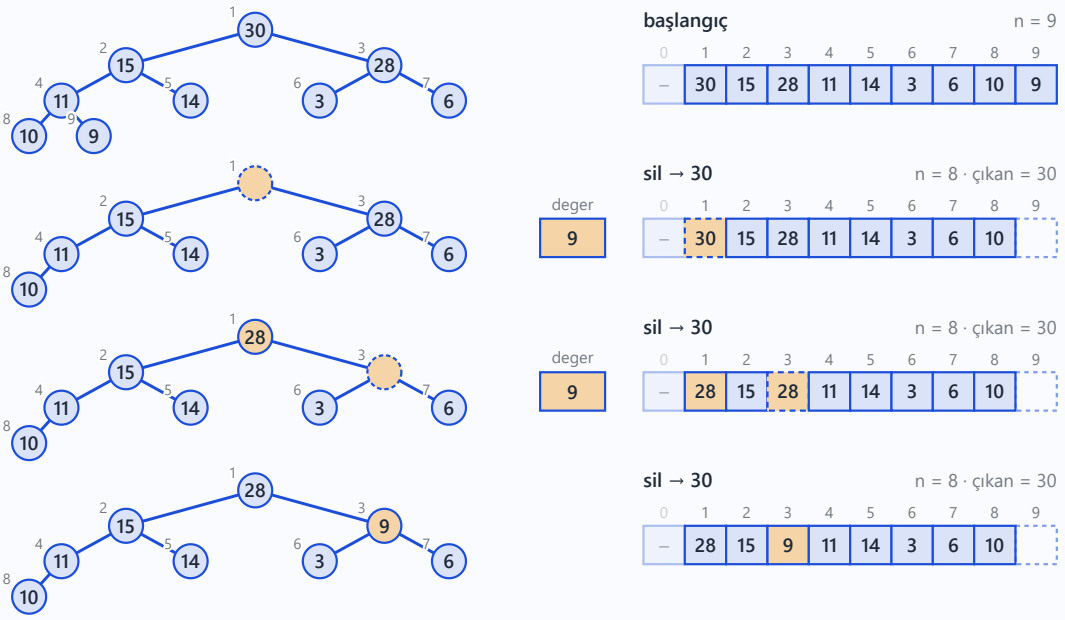
13.4 Silme

Maksimum heap'ten bir öğe almak, öncelikli kuyruğun asıl işidir. Alınacak öğe bellidir: kökteki en büyük öğe. Onu okumak kolaydır. Zor olan, öğeyi çıkardıktan sonra kalan yapıyı yeniden heap yapmaktır. Silme de eklemede olduğu gibi önce şekli, sonra düzeni düzeltir; ama düzeltme bu kez yukarıdan aşağı doğru yürür.

13.4.1 Kök alınır, sondaki öğe yerleştirilir

Kök çıkarılınca ağacın tepesinde bir boşluk oluşur. Ağaç tam kalacaksa öğe sayısı bir azaldığında boşalması gereken konum kök değil, son konumdur, yani *n*. Bu yüzden son konumdaki öğe yerinden alınır ve ona bir yer aranır. Yer arama kökteki boşluktan başlar.

Sondaki öğe büyük olasılıkla küçük bir değerdir, çünkü bir yapraktır. Kökün iki çocuğundan büyük değilse oraya konamaz. Her adımda üç değere bakılır: boşluğun iki çocuğu ve yerleştirilecek öğe. Üçünün en büyüğü boşluğa gelir. En büyük yerleştirilecek öğeyse yeri bulunmuştur. Değilse büyük çocuk bir düzey yukarı, boşluğa taşınır ve boşluk o çocuğun eski yerine iner. Bu aşağı doğru harekete **aşağı kaydırma** (*down-heap*, aşağı süzme) denir. Boşluğun çocuğu kalmadığında da yer bulunmuştur: boşluk bir yapraktır.



Şekil 13.10: Kökün alınması. Yukarıdan aşağıya: başlangıç; 30'un alınıp sondaki 9'un çıkarılması; büyük çocuk 28'in yukarı taşınması; 9'un 3. konuma yerleşmesi.

Şekil 13.10 bunu *30, 15, 28, 11, 14, 3, 6, 10, 9* heap'i üzerinde gösterir. İlk alınacak öğe kökteki 30'dur. Sondaki öğe 9, 9. konumdan çıkarılır; öğe sayısı 8'e iner. Kökün çocukları 15 ve 28'dir; büyüğü 28'dir. *28 > 9* olduğundan 28 köke çıkar ve boşluk 3. konuma iner. 3. konumun çocukları 6. konumdaki 3 ve 7. konumdaki 6'dır; büyüğü 6'dır. *9 ≥ 6* olduğundan 9 boşluğa, 3. konuma yerleşir. Yeni heap *28, 15, 9, 11, 14, 3, 6, 10*'dur.

Büyük çocuğun seçilmesi keyfi değildir. Boşluğa gelen öğe, öbür çocuğun ebeveyni olacaktır. Küçük çocuk yukarı çıksaydı büyük çocuk onun çocuğu olurdu ve heap özelliği hemen bozulurdu. Büyük çocuk yukarı çıktığında ise hem küçük kardeşinden hem kendi eski çocuklarından küçük değildir; heap özelliği korunur.

13.4.2 `maksimum_yigin_al`

Sunum, kitaptaki silme yordamının düzeltilmiş bir sürümünü verir:

```
bool maksimum_yigin_al(eleman *cikan) {
  if (YIGIN_BOSMU(n)) return false;
  *cikan = yigin[1];          /* kök döndürülecek */
  eleman deger = yigin[n--];  /* yeri aranacak */
  int i = 1, j = 2;
  while (j <= n) {
    if (j < n && yigin[j].x < yigin[j + 1].x) j++;
    if (deger.x >= yigin[j].x) break;
    yigin[i] = yigin[j];      /* çocuğu yukarı al */
    i = j; j *= 2;
  }
  yigin[i] = deger;
  return true;
}
```

Yordam önce heap'in boş olup olmadığına bakar. Boşsa `false` döndürür ve hiçbir şeye dokunmaz. Değilse kökteki öğe `*cikan`'a kopyalanır. `cikan`, çağıranın değişkenini gösteren bir işaretçidir; `*cikan = yigin[1]` ataması çağıranın değişkenine yazar. Tanım gereği ilk alınacak öğe dizinin ilk öğesidir.

`eleman deger = yigin[n--];` satırı son konumdaki öğeyi `deger`'e kopyalar ve öğe sayısını bir azaltır. `n--` sonek işlecidir: ifade önce *n*'nin eski değerini verir, sonra *n* azalır. Yani `yigin[9]` okunur ve *n* 8 olur. Bu andan sonra `yigin[9]` heap'e ait değildir. Tam ağaç bozulmasın diye sondaki öğe çıkarılmıştır ve şimdi ona kökten başlayarak yer aranacaktır.

`i` `deger`'in aday konumunu, yani boşluğu gösterir; başta köktür. `j` `i`'nin çocuklarını gösterir; başta sol çocuk olan 2'dir. Döngü `j <= n` olduğu sürece, yani `i`'nin en az bir çocuğu olduğu sürece döner. Döngünün ilk satırı büyük çocuğu seçer: sağ çocuk varsa ve soldakinden büyükse `j` bir artırılır. İkinci satır `deger`'i büyük çocukla karşılaştırır. `deger` ondan küçük değilse yer bulunmuştur ve döngü `break` ile biter. Değilse büyük çocuk boşluğa taşınır: `yigin[i] = yigin[j]`. Son satır boşluğu taşınan çocuğun eski yerine indirir: `i = j`, ve `j`'yi onun sol çocuğuna götürür: `j *= 2`. Döngüden çıkılınca `deger` boşluğa yazılır ve `true` döner.

`j < n` sınamasının işi önemlidir. `j` sol çocuğu gösterir; sağ çocuk `j + 1`'dedir. `j == n` ise sol çocuk heap'in son öğesidir ve sağ çocuk yoktur. Bu durumda `yigin[j + 1]` heap'e ait değildir. `j < n` koşulu, `yigin[j+1]` okunmadan önce sağ çocuğun var olduğunu garanti eder. `&&` işleci kısa devreyle değerlendirildiği için koşul yanlışsa ikinci karşılaştırma hiç yapılmaz.

Sunum bu sürümü "düzeltilmiş" diye adlandırır. Kitaptaki sürüm öğeyi dönüş değeri olarak verir ve boş heap'te `return 0;` yazar. Dönüş tipi `eleman` gibi bir yapıysa bu satır derlenmez. Bir tam sayı bir yapıya dönüştürülemez. Bu satır bir test dosyasında denenmiş ve derleyici "returning 'int' from a function with incompatible result type 'eleman'" hatasıyla durmuştur. Önerilen çözüm, arayüzde olduğu gibi, dönüş değerini başarı ya da başarısızlık bildiren bir `bool` yapmak ve alınan öğeyi işaretçi parametreyle döndürmektir. Böylece boş heap için anlamsız bir öğe uydurmak gerekmez.

Yordam şöyle çağrılır:

```
eleman e;
if (maksimum_yigin_al(&e))
  printf("alınan: %d\n", e.x);
else
  printf("heap boş\n");
```

`&e`, `e` değişkeninin adresidir. Yordamın içindeki `*cikan = yigin[1]` ataması bu adrese, yani `e`'ye yazar. C'de fonksiyon parametreleri değer olarak aktarılır. Bir fonksiyonun çağıranın değişkenini değiştirebilmesi için o değişkenin adresini alması gerekir. İşaretçi parametre burada bunu sağlar.

13.4.3 Kod izleme: birinci alma

Tablo 13.6 *30, 15, 28, 11, 14, 3, 6, 10, 9* heap'inde ilk `maksimum_yigin_al` çağrısını satır satır izler. Satır numaraları sunumdaki izleme slaytına göredir: 2. satır boşluk denetimi, 3. satır kökün kopyalanması, 4. satır son öğenin çıkarılması, 6. satır döngü koşulu, 7. satır büyük çocuğun seçimi, 8. satır karşılaştırma, 9. satır taşıma, 10. satır ilerleme, 12. satır son yazmadır.

Satır	`n`	`i`	`j`	Koşul / işlem	`yigin[1..9]`
2	9	—	—	`YIGIN_BOSMU(9)` yanlış	30, 15, 28, 11, 14, 3, 6, 10, 9
3	9	—	—	`*cikan = 30`	_, 15, 28, 11, 14, 3, 6, 10, 9
4	8	—	—	`deger = 9`, *n = 8*	_, 15, 28, 11, 14, 3, 6, 10, (9)
5	8	1	2	boşluk kökte, `j` sol çocukta	_, 15, 28, 11, 14, 3, 6, 10, (9)
6	8	1	2	*2 ≤ 8*: döngüye girilir	_, 15, 28, 11, 14, 3, 6, 10, (9)
7	8	1	3	*15 < 28*: `j++`	_, 15, 28, 11, 14, 3, 6, 10, (9)
8	8	1	3	*9 ≥ 28*? Hayır	_, 15, 28, 11, 14, 3, 6, 10, (9)
9	8	1	3	`yigin[1] = yigin[3]`	28, 15, _, 11, 14, 3, 6, 10, (9)
10	8	3	6	`i = j; j *= 2`	28, 15, _, 11, 14, 3, 6, 10, (9)
6, 7	8	3	7	*6 ≤ 8*; *3 < 6*: `j++`	28, 15, _, 11, 14, 3, 6, 10, (9)
8	8	3	7	*9 ≥ 6*? Evet: `break`	28, 15, _, 11, 14, 3, 6, 10, (9)

Satır	`n`	`i`	`j`	Koşul / işlem	`yigin[1..9]`
12	8	3	7	`yigin[3] = deger`	28, 15, 9, 11, 14, 3, 6, 10, (9)
13	8	3	7	`return true` ; çağıran 30'u alır	28, 15, 9, 11, 14, 3, 6, 10

Tablo 13.6: Birinci `maksimum_yigin_al` izlemesi. Dizi sütunu `yigin[1..9]` 'u, `_` boşluğu gösterir; *n*'nin dışında kalan hücre parantez içindedir.

Döngü iki kez girilmiş, bir kez taşıma yapmış ve ikinci turda `break` ile bitmiştir. 9 en alta inmeden, 3. konumda yerleşmiştir. Bu bir "araya girme" örneğidir: sondaki öğe, kökten yaprağa giden yolun ortasında yerini bulmuştur. Toplam dört karşılaştırma yapılmıştır: her turda biri çocukları karşılaştırmak, biri `deger` 'i büyük çocukla karşılaştırmak için. Tur başına iki karşılaştırma, aşağı kaydırmanın ekleme döngüsünden biraz daha pahalı olmasının nedenidir.

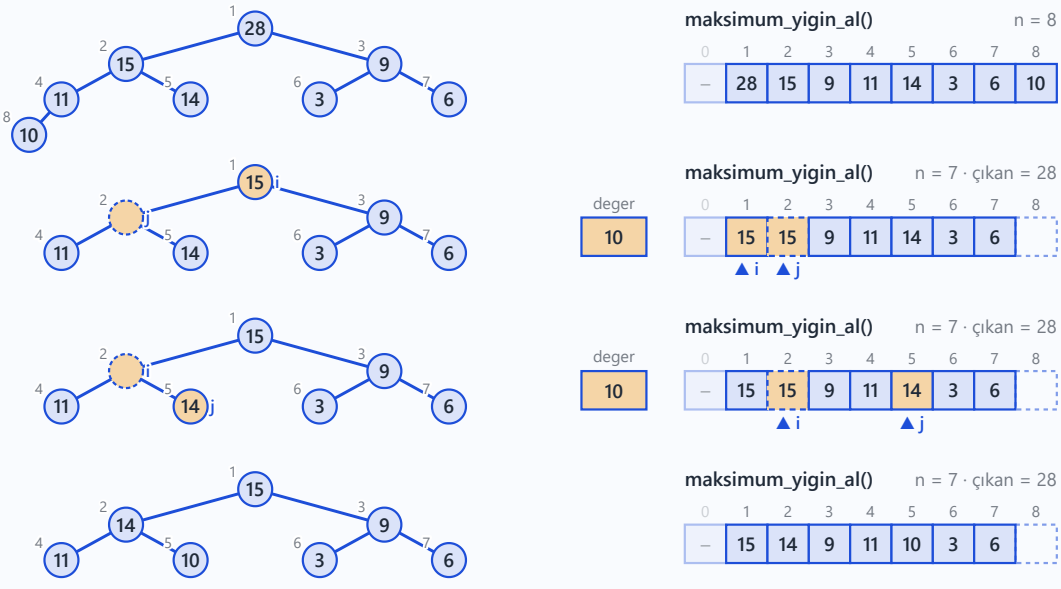
Tablonun son sütunundaki parantezli 9 bir inceliği gösterir. `n--` 'den sonra 9. hücre silinmez; içinde 9'un bir kopyası durur. Ama *n = 8* olduğu için yordamların hiçbiri o hücreye heap'in parçası olarak bakmaz. Bir sonraki eklemede o hücreye yeni öğe yazılacaktır. C dizisinde "silme" bir hücreyi boşaltmak değil, sınırı çekmektir.

13.4.4 Kod izleme: ikinci alma

İkinci çağrıda heap *28, 15, 9, 11, 14, 3, 6, 10*, *n = 8*'dir. Bu kez sondaki öğe, 10, yaprağa kadar inecektir. Tablo 13.7 döngünün turlarını özetler.

Tur	`i`	Çocuklar	Büyük çocuk (`j`)	Karşılaştırma	`yigin[1..7]` (tur sonunda)
1	1	15, 9	15 (2)	*10 ≥ 15*? Hayır: taşı	15, _, 9, 11, 14, 3, 6
2	2	11, 14	14 (5)	*10 ≥ 14*? Hayır: taşı	15, 14, 9, 11, _, 3, 6
3	5	yok (*j = 10 > 7*)	—	döngüye girilmez	15, 14, 9, 11, _, 3, 6
son	5	—	—	`yigin[5] = deger`	15, 14, 9, 11, 10, 3, 6

Tablo 13.7: İkinci `maksimum_yigin_al` izlemesi (`*cikan = 28` , `deger = 10` , *n = 7*).



Şekil 13.11: İkinci alma sırasında dört an. Yukarıdan aşağıya: başlangıç; 15'in köke taşınması; 14'ün 2. konuma taşınması; 10'un 5. konuma, bir yaprağa yazılması.

İlk turda kökün çocukları 15 ve 9'dur. Sol çocuk sağdakinden küçük olmadığı için `j` 2'de kalır. *10 < 15* olduğundan 15 köke çıkar ve boşluk 2'ye iner. İkinci turda 2'nin çocukları 11 ve 14'tür; sağdaki büyüktür ve `j` 5 olur. *10 < 14* olduğundan 14 yukarı çıkar ve boşluk 5'e iner. 5'in sol çocuğu 10. konumda olurdu; ama *10 > n = 7*'dir, yani 5. konum bir yapraktır. Döngü koşulu yanlış olur ve 10 boşluğa yazılır.

Çağıran 28'i alır; yeni heap *15, 14, 9, 11, 10, 3, 6*'dır.

Bu çağrı silmenin en kötü durumudur: sondaki öğe kökten yaprağa kadar inmiştir. 8 öğeli heap'te yaprak düzeyi 2. ya da 3. düzeydir. 10, 7 öğeli heap'in 2. düzeyinde bir yaprakta durmuştur ve iki düzey inmiştir. İki alma birlikte silmenin iki davranışını gösterir: sondaki öğe yolun ortasında araya girebilir ya da en alta kadar inebilir.

13.4.5 Heap boşalana dek almak

Heap'ten öğe almaya heap boşalana kadar devam edilirse ne olur? Tablo 13.8 kalan yedi almayı gösterir. Her satırda alınan öğe, o anda kökte olan en büyük öğedir; sondaki öğe aşağı kaydırma ile yeni yerine yerleşir.

Alınan	Sondaki öğe	Yerleştiği konum	Heap (sonra)
15	6	4	14, 11, 9, 6, 10, 3
14	3	5	11, 10, 9, 6, 3
11	3	4	10, 6, 9, 3
10	3	3	9, 6, 3
9	3	2	6, 3
6	3	1	3
3	3	— (son öğeydi)	boş

Tablo 13.8: *15, 14, 9, 11, 10, 3, 6* heap'inden art arda alma. Çıkanlar 30 ve 28 ile başlar.

Çıkış sırası *30, 28, 15, 14, 11, 10, 9, 6, 3*'tür: öğeler azalan sırada çıkmıştır. Bu bir rastlantı değildir. Her alma o anda kalan öğelerin en büyüğünü verir; en büyüklerin art arda alınması da azalan bir sıra üretir. Tablonun bir ayrıntısı da dikkat çeker. 3 değeri art arda beş kez "sondaki öğe" olmuştur. Her seferinde kökten başlayan bir yolculukla yeniden yerleştirilmiştir. Altıncı almada 6 alınınca heap'te yalnız 3 kalır. Yedinci almada 3 hem kök hem son öğedir; `deger = yigin[n--]` onu alır, *n* 0 olur, döngüye girilmez ve 3 kullanılmayan 1. hücreye yazılır. Heap boşalmıştır ve sekizinci bir çağrı `false` döndürür.

Bu gözlem önemli bir algoritmanın fikridir. *n* öğe bir heap'e konur, sonra *n* kez kök alınırsa öğeler sıralı çıkar. Bu, **heap sort**'un fikridir. Her alma *O(lg n)* sürdüğü için sıralama *O(n lg n)* sürer. Heap sort'un ek dizi kullanmayan bir sürümü de bu notta yer alır.

13.4.6 Silmenin maliyeti

Silmenin maliyeti de döngünün tur sayısıyla belirlenir. Her tur boşluğu bir düzey aşağı indirir ve en çok iki karşılaştırma ile bir taşıma yapar. Boşluk kökten başlar ve en alt düzeyde durmak zorundadır. Teorem 13.2 gereği en çok $\lfloor \lg n \rfloor$ tur yapılır. Silmenin en kötü durumu bu yüzden *O(lg n)*'dir; karşılaştırma sayısı en çok $2\lfloor \lg n \rfloor$'dir.

Silmede en iyi durum da *Ω(1)*'dir. Sondaki öğe kökün iki çocuğundan küçük değilse ilk turda yerleşir. Bu, örneğin heap'te iki öğe kalmışken olur: kök alınınca *n* 1'e iner, $j = 2 > n$ olduğu için döngüye hiç girilmez ve sondaki öğe doğrudan köke yazılır. Aynı öncelikli öğelerin çok olduğu bir heap'te de sondaki öğe çoğu kez hemen yerleşir. Ama farklı değerli büyük bir heap'te sondaki öğe yapraktan geldiği için genellikle küçüktür ve aşağıya, yaprakların yakınına kadar iner. Bu yüzden silme, rastgele verilerde eklemeden farklı olarak ortalamada da en kötü duruma yakın çalışır. Rastgele bir milyon değerle kurulan bir heap'te ilk bin silme ortalama 18,93 tur yapmıştır; $\lfloor \lg 10^6 \rfloor = 19$'dur. Ek bellek yine *O(1)*'dir: `deger`, `i` ve `j`.

> **Sık yapılan hata**
>
> Silmede sondaki öğe yerine kökün büyük çocuğunu köke çıkarıp boşluğu aşağıya taşımak, sonunda da boşluğu son öğeyle doldurmamak. Bu yöntem boşluğu yaprak düzeyinde rastgele bir konuma bırakır. Boşluk son konumda değilse dizide bir delik oluşur ve ağaç tam olmaktan çıkar. Silmede önce şekil korunur: çıkarılan konum her zaman *n*'dir ve yeri aranan öğe oradan gelen öğedir.

13.4.7 İki hatalı sürüm

Silme yordamı küçük görünür, ama iki satırındaki ayrıntılar doğruluğu belirler. Bu ayrıntıların neden gerektiği, onları bozan iki sürümle görülebilir.

Büyük çocuk yerine ilk çocuk. Döngüden `if (j < n && yigin[j].x < yigin[j + 1].x) j++;` satırı çıkarılırsa her turda sol çocuk seçilir. *30, 15, 28, 11, 14, 3, 6* heap'inde bu sürümle 30 alınmıştır. Sondaki öğe 6'dır. Sol çocuk 15, *6 < 15* olduğu için köke çıkar; boşluk 2'ye iner, orada sol çocuk 11 yukarı çıkar ve 6, 4. konuma yazılır. Sonuç *15, 11, 28, 6, 14, 3*'tür. Kökteki 15, sağ çocuğu 28'den küçüktür; heap özelliği bozulmuştur. Bir sonraki alma en büyük öğe olarak 28'i değil 15'i verir. Program çökmez, yalnız yanlış öğeyi verir.

`j < n` *sınaması olmadan.* Bu sınama atlanırsa, `j == n` olduğunda `yigin[n + 1]` okunur. Bu hücre heap'in dışındadır. `maksimum_yigin_al`'da ilginç bir durum vardır: `n--`'den sonra `yigin[n + 1]`'de, yeri aranan `deger`'in kendi kopyası durur. Bu kopya seçilirse karşılaştırma `deger.x >= deger.x` olur ve döngü doğru yerde durur. Bu yüzden bu yordamda hata sonucu değiştirmez; karışık ekleme ve almalardan oluşan yarım milyon rastgele denemede bile yanlış bir sonuç görülmemiştir. Ama bu bir rastlantıdır ve kodun doğru olduğunu göstermez. Aynı döngü aşağıdan yukarı heap kurmada kullanıldığında `a[n + 1]` dizinin dışında kalabilir. 6 öğelik bir dizide bu sınama kaldırılınca dizinin hemen arkasındaki 99 değeri heap'e girmiş, dizideki 3 kaybolmuştur. `j < n` sınaması bu yüzden her aşağı kaydırma döngüsünde yazılmalıdır.

13.5 Gerçeklemeyi geliştirmek

Ekleme ve silme yordamları doğru çalışır, ama global bir diziye bağlıdır. Bu kesim yordamları üç yönde geliştirir: heap'i bir yapıya taşımak, minimum heap'i yazmak ve önceliği zamanla değişen öğeleri ele almak.

13.5.1 Global dizi yerine yapı

Global dizi ile programda tek bir heap olabilir. Bir işletim sistemi ise birçok kuyruk tutar: her işlemci için ayrı bir hazır kuyruğu, zamanlayıcılar için ayrı bir kuyruk. Kapasite de derleme sırasında sabittir. Sunum bu sakıncaları gidermek için dizi ile öğe sayısını bir yapıda toplar ve diziyi çalışma sırasında ayırır:

```
typedef struct { eleman *a; int n, kapasite; } Yigin;

Yigin *yigin_olustur(int kapasite) {
  Yigin *h = malloc(sizeof *h);
  if (h == NULL) return NULL;
  h->a = malloc((kapasite + 1) * sizeof *h->a);
  if (h->a == NULL) { free(h); return NULL; }
  h->n = 0; h->kapasite = kapasite;
  return h;
}
void yigin_yok_et(Yigin *h) { if (h) { free(h->a); free(h); } }
```

`Yigin` yapısının üç alanı vardır. `a`, heap bellekte ayrılan dizinin adresidir. `n` öğe sayısını, `kapasite` en çok kaç öğe alınabileceğini tutar. Her heap ayrı bir `Yigin` olduğu için programda istenen sayıda heap bulunabilir. Yordamlar hangi heap üzerinde çalışacaklarını parametre olarak alır.

`yigin_olustur` iki ayrı blok ayırır. İlki yapının kendisidir: `malloc(sizeof *h)`. Burada `sizeof *h`, `h`'nin gösterdiği tipin, yani `Yigin`'in boyutudur. Tip adını tekrar yazmamak, tip sonradan değişirse satırı doğru tutar. İkinci blok dizidir. 0. hücre kullanılmadığı için `kapasite + 1` hücre ayrılır. İki ayırmanın sonucu da denetlenir. İkinci ayırma başarısız olursa yalnız `NULL` döndürmek yetmez; önce ayrılmış olan yapı da `free(h)` ile geri verilir. Aksi hâlde o blok kimsenin gösteremediği bir bellek parçası olarak kalır, yani sızar.

`yigin_yok_et` blokları ayırmanın tersi sırayla bırakır. Önce dizi, sonra yapı bırakılır, çünkü dizinin adresi yapının içindedir; yapı önce bırakılsaydı `h->a` okunamazdı. `if (h)` denetimi `NULL` bir işaretçiyle çağrıyı zararsız yapar. Bu, `free(NULL)`'ın hiçbir şey yapmamasıyla tutarlı bir davranıştır.

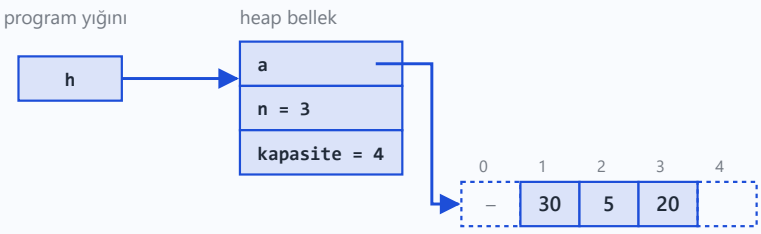
Yapı sürümünün ekleme ve alma yordamları sunumda verilmez; öneri, başarının `TRUE` / `FALSE` ile bildirilmesi ve öğenin işaretçiyle aktarılmasıdır. Aşağıdaki iki yordam bu öneriyi uygular. Gövdeleri global sürümlerle aynıdır; yalnız `yigin` yerine `h->a`, `n` yerine `h->n` yazılır:

```
bool yigin_ekle(Yigin *h, eleman item) {
  if (h->n == h->kapasite) return false;   /* dolu */
  int i = ++h->n;
  while (i != 1 && item.x > h->a[i / 2].x) {
    h->a[i] = h->a[i / 2];
    i /= 2;
  }
  h->a[i] = item;
  return true;
}

bool yigin_al(Yigin *h, eleman *cikan) {
  if (h->n == 0) return false;             /* boş */
  *cikan = h->a[1];
  eleman deger = h->a[h->n--];
  int i = 1, j = 2;
  while (j <= h->n) {
    if (j < h->n && h->a[j].x < h->a[j + 1].x) j++;
    if (deger.x >= h->a[j].x) break;
    h->a[i] = h->a[j];
    i = j; j *= 2;
  }
  h->a[i] = deger;
  return true;
}
```

Ekleme yordamında dolu heap artık bir hata iletisiyle değil, `false` dönüşüyle bildirilir; ne yapılacağına çağıran karar verir. Döngü koşulu da biraz kısalmıştır: `i != 1` ve "ebeveyni daha küçük" koşulları tek bir `while`'da birleşir. `&&` kısa devreyle değerlendirildiği için `i == 1` olduğunda `h->a[0]` hiç okunmaz. İki yazım aynı işi yapar.

Şekil 13.12 bu yapının bellekteki görünümünü çizer. Program yığınında (*stack*) yalnız `h` işaretçisi durur. Yapı ve dizi heap bellektedir; yapının `a` alanı dizinin başlangıç adresini tutar.



Şekil 13.12: `yigin_olustur(4)` ile kurulup 20, 5 ve 30 eklenmiş heap'in bellek görünümü. Dizi *kapasite + 1 = 5* hücredir; 0. hücre kullanılmaz, 4. hücre henüz boştur.

Bu sürüm sayaçlı bir `malloc` / `free` sarmalayıcısıyla sınanmıştır. Sarmalayıcı, her başarılı ayırmada ve her bırakmada bir sayaç artıran küçük iki fonksiyondur; makrolarla `malloc` ve `free`'nin yerine geçirilir. Deneme `yigin_olustur(4)` ile başlamış ve 20, 5, 30, 10, 40 değerlerini eklemeye çalışmıştır. İlk dört ekleme `TRUE`, beşincisi `FALSE` döndürmüştür, çünkü kapasite 4'tür. Ardından heap boşalana dek alınmış ve *30, 20, 10, 5* çıkmıştır. `yigin_yok_et`'ten sonra sayaçlar 2 ayırma ve 2 bırakma göstermiştir; sızıntı yoktur. `yigin_yok_et(NULL)` çağrısı da sorunsuz geçmiştir. Bu derleyici yapılandırmasında `sizeof(Yigin)` 16 bayttır: 8 baytlık işaretçi ve iki 4 baytlık `int`.

Dizinin kapasitesi, dolunca `realloc` ile iki katına çıkarılarak sınırsız da yapılabilir. O zaman ekleme hiç `false` döndürmez; büyütmeye denk gelen tek bir ekleme $\Theta(n)$ sürer, ama amortize maliyet değişmez. İki katına büyütmenin toplam kopyalama maliyeti n eklemede $2n$'yi aşmaz.

13.5.2 Minimum heap

Fotokopi ve havaalanı örnekleri minimum öncelikli kuyruk isterdi. Minimum heap'in yordamları maksimum heap'inkilerden yalnız iki noktada ayrılır: yordamın adı ve karşılaştırmaların yönü. Sunum ekleme yordamını verir:

```
void minimum_yigin_ekle(eleman item) {
  /* ... dolu denetimi aynı ... */
  int i = ++n;
  while (i != 1) {
    if (item.x >= yigin[i / 2].x) break; /* >= ! */
    yigin[i] = yigin[i / 2];
    i /= 2;
  }
  yigin[i] = item;
}
```

Tek değişiklik `break` koşulundadır: `item.x <= …` yerine `item.x >= …`. Yeni öğe ebeveyninden küçük olduğu sürece ebeveyn aşağı iner; yeni öğe ebeveyninden küçük değilse yeri bulunmuştur. Kodun ikinci satırındaki yorum, dolu denetiminin maksimum sürümdekiyle aynı olduğunu söyler; kısalık için yazılmamıştır. Test programında bu denetim eklenerek derlenmiştir.

Alma yordamında iki yer değişir: büyük çocuk yerine küçük çocuk seçilir ve durma koşulu tersine döner.

```
bool minimum_yigin_al(eleman *cikan) {
  if (YIGIN_BOSMU(n)) return false;
  *cikan = yigin[1];
  eleman deger = yigin[n--];
  int i = 1, j = 2;
  while (j <= n) {
    if (j < n && yigin[j].x > yigin[j + 1].x) j++;  /* küçük çocuk */
    if (deger.x <= yigin[j].x) break;               /* <= ! */
    yigin[i] = yigin[j];
    i = j; j *= 2;
  }
  yigin[i] = deger;
  return true;
}
```

Yer	Maksimum	Minimum
ekle: durma koşulu	`item.x <= ebeveyn`	`item.x >= ebeveyn`
al: çocuk seçimi	büyüğü: `[j] < [j+1]` ise `j++`	küçüğü: `[j] > [j+1]` ise `j++`
al: durma koşulu	`deger.x >= [j]`	`deger.x <= [j]`

Tablo 13.9: Maksimum ve minimum heap yordamlarındaki farklar.

Yordamın adı değiştirilip küçük ve büyük işaretleri de tersine çevrilirse, minimum heap başarıyla yazılmış olur. Tablo 13.9 değişen üç satırı toplar. Bu iki yordam 20, 5, 30, 10, 40 değerleriyle denenmiştir. Eklemelerden sonra dizi sırasıyla *20*; *5, 20*; *5, 20, 30*; *5, 10, 30, 20*; *5, 10, 30, 20, 40* olmuştur. Alma çağrıları *5, 10, 20, 30, 40*'ı, yani artan sırayı vermiştir.

Karşılaştırmaları ters çevirmenin bir başka yolu, maksimum heap'i hiç değiştirmeden öncelikleri eksiyle çarpmaktır. $-x$'in en büyüğü x'in en küçüğüdür. Bu hile pratikte kullanılır, ama anahtarın sayı olduğu durumlarla sınırlıdır ve `INT_MIN` gibi sınır değerlerde taşma riski taşır. Karşılaştırmayı ayrı bir fonksiyona taşımak daha genel bir çözümdür: yordamlar o fonksiyonu çağırır ve hangi tür heap olduğu bu fonksiyonla belirlenir. Standart kütüphanedeki `qsort` da sıralama yönünü böyle bir karşılaştırma fonksiyonuyla alır.

13.5.3 Önceliği değişen öğeler

Havaalanı örneğine geri dönelim. Uçakların yakıtı, kuleye ilk başvurdukları anda bir kez ölçülüp unutulmaz. Her uçağın saniyede harcadığı yakıt farklıdır. Bir uçak bir saniyede 10 galon (37,85 litre) yakıt harcarken, bir başkası 10 galonla belki bir saat uçabilir. Bu yüzden öncelikler zamanla değişir. Sistemde her saniye bir kesme (*interrupt*) gelir ve yakıtlar güncellenir. Problemi iyi yakalamak, bu değişimi de modele katmayı gerektirir.

Bir öğenin önceliği değişince heap özelliği yalnız o öğenin çevresinde bozulabilir. Maksimum heap'te önceliği artan öğe ebeveyninden büyük olabilir; yukarı kaydırılır. Önceliği azalan öğe çocuklarından küçük olabilir; aşağı kaydırılır. İki işlem de bir kök–yaprak yolu boyunca çalışır ve $O(\lg n)$ sürer. Sunum yukarı kaydırma sürümünü verir:

```c
/* k konumundaki öğenin önceliği arttı: yukarı kaydır */
void oncelik_artir(int k, int yeni) {
  eleman e = yigin[k];
  e.x = yeni;
  while (k != 1 && e.x > yigin[k / 2].x) {
    yigin[k] = yigin[k / 2];
    k /= 2;
  }
  yigin[k] = e;
}
```

Yordam *k*. konumdaki öğeyi `e`'ye kopyalar ve önceliğini yeni değere çeker. Ardından ekleme yordamındaki döngünün aynısı çalışır, yalnız başlangıç konumu *n + 1* değil *k*'dır. `k != 1` koşulu kökte durur; `e.x > yigin[k / 2].x` koşulu ebeveyni daha küçük olduğu sürece devam eder. Döngünün sonunda öğe yeni yerine yazılır. Yordam yeni değerin eskisinden büyük olduğunu varsayar. Küçük bir değerle çağrılırsa döngü hiç dönmez ve öğe olduğu yerde küçülür; bu, heap özelliğini aşağı doğru bozabilir.

Öncelik azaldığında aşağı kaydırma gerekir. Bu sürüm sunumda yoktur; silme yordamının döngüsü *k*. konumdan başlatılarak elde edilir:

```c
/* k konumundaki öğenin önceliği azaldı: aşağı kaydır */
void oncelik_azalt(int k, int yeni) {
  eleman e = yigin[k];
  e.x = yeni;
  int j = 2 * k;
  while (j <= n) {
    if (j < n && yigin[j].x < yigin[j + 1].x) j++;
    if (e.x >= yigin[j].x) break;
    yigin[k] = yigin[j];
    k = j; j *= 2;
  }
  yigin[k] = e;
}
```

Örnek 13.3

Heap *30, 15, 28, 11, 14, 3, 6, 10, 9* olsun. 9. konumdaki 9'un önceliği 31'e çıkarılır: `oncelik_artir(9, 31)`. Ebeveyni 4. konumdaki 11'dir ve *31 > 11*; 11 aşağı iner. Sonra 2. konumdaki 15 ve kökteki 30 da aşağı iner. 31 köke yazılır. Sonuç *31, 30, 28, 15, 14, 3, 6, 10, 11*'dir; yol 9 → 4 → 2 → 1, üç kaydırmadır.

Aynı başlangıç heap'inde kökün önceliği 5'e indirilir: `oncelik_azalt(1, 5)`. Kökün çocukları 15 ve 28'dir; 28 yukarı çıkar. 3. konumun çocukları 3 ve 6'dır; 6 yukarı çıkar. 7. konum yapraktır; 5 oraya yazılır. Sonuç *28, 15, 6, 11, 14, 3, 5, 10, 9*'dur. İki sonuç da `heap_mi` ile sınanmış ve geçerli bulunmuştur.

Bu yordamların zor yanı kodları değil, *k*'yı bilmektir. Kule, yakıtı değişen uçağın adını bilir; heap'teki konumunu bilmez. Heap'te bir öğeyi aramak *O(n)* sürer, çünkü heap bir arama yapısı değildir. Her güncelleme için arama yapılırsa *O(lg n)*'lik kaydırmanın kazancı kaybolur. Çözüm, her öğenin heap'teki konumunu ayrı bir **indis tablosunda** tutmaktır: örneğin `konum[ucak_no]`. Heap yordamları bir öğeyi her taşıdığında bu tabloyu da günceller. Öğe `yigin[i] = yigin[j]` ile taşınıyorsa ardından `konum[yigin[i].no] = i` yazılır. Böylece konum *O(1)*'de bulunur ve öncelik değiştirmek gerçekten *O(lg n)* sürer. Çizge algoritmalarındaki **anahtar azaltma** (*decrease key*) işlemi minimum heap'te aynı yolla yapılır.

13.6 Diğer yapılarla karşılaştırma

Heap'in iki işlemi de *O(lg n)* sürer. Bu iyi bir sonuç mudur? Daha basit bir yapı, bir dizi ya da bir liste, aynı işi daha kolay yapamaz mı? Bu kesim beş gerçeklemeyi önce kuramsal maliyetleriyle, sonra sayılarla ve son olarak gerçek bir ölçümle karşılaştırır.

13.6.1 Beş gerçeklemenin maliyeti

#	Yapı	Ekle	Sil (en büyük)	Neden
1	Sırasız dizi	*Θ(1)*	*Θ(n)*	Sona eklenir; en büyüğü bulmak için tamamına bakılır.

#	Yapı	Ekle	Sil (en büyük)	Neden
2	Sırasız liste	$\Theta(1)$	$\Theta(n)$	Başa eklenir; yine hepsinde gezinilir.
3	Sıralı dizi (küçükten büyüğe)	$O(n)$	$\Theta(1)$	`binsearch` $O(\lg n) + n$ kaydırma; en büyük sonda.
4	Sıralı bağlı liste (büyükten küçüğe)	$O(n)$	$\Theta(1)$	Yeri bulunur; baştan silinir, baş güncellenir.
5	Maksimum heap	$O(\lg n)$	$O(\lg n)$	Yalnızca bir kök–yaprak yolu.

Tablo 13.10: Maksimum öncelikli kuyruğun beş gerçeklemesi.

Tablo 13.10 sunumdaki karşılaştırmayı verir. Satırlar tek tek ele alınabilir.

Sırasız dizi. Yeni öğe dizinin sonuna yazılır: bir atama, $\Theta(1)$. En büyük öğeyi almak için önce onu bulmak gerekir. Dizi sırasız olduğu için bütün öğelere bakılır; bu her zaman $n - 1$ karşılaştırmadır ve $\Theta(n)$'dir. Bulunan öğe çıkarılınca boşluğa sondaki öğe taşınır; kaydırma gerekmez. Alma işlemi bu yüzden tam olarak doğrusal sürer.

Sırasız liste. Bağlı listede yeni öğe başa eklenir: bir düğüm ayrılır ve iki işaretçi güncellenir, $\Theta(1)$. En büyüğü bulmak yine bütün düğümleri gezmeyi gerektirir: $\Theta(n)$.

Sıralı dizi. Dizi küçükten büyüğe sıralı tutulursa en büyük öğe sondadır ve almak $\Theta(1)$'dir: n bir azaltılır. Ekleme ise pahalıdır. Yeni öğenin yeri ikili aramayla (`binsearch`) $O(\lg n)$'de bulunur, ama ondan büyük bütün öğeler bir sağa kaydırılmalıdır. Yeni öğe en küçükse n kaydırma yapılır. Yeni öğe en büyükse hiç kaydırma yapılmaz; bu yüzden ekleme $O(n)$'dir, $\Theta(n)$ değil. İkili arama kullanılmayıp yer baştan doğrusal aramayla bulunursa durum değişir. Yeri bulmak için k öğeye bakılır, ardından $n - k$ öğe kaydırılır; toplam her zaman yaklaşık n'dir. O zaman ekleme $O(n)$ değil $\Theta(n)$ olur.

Sıralı bağlı liste. Liste büyükten küçüğe sıralı tutulursa en büyük öğe baştadır. Almak, baş düğümü çıkarıp başı güncellemektir: $\Theta(1)$. Eklemede yeni öğenin yeri baştan gezilerek bulunur; kaydırma gerekmez, ama bağlı listede ikili arama yapılamaz. Yeni öğe en büyükse başa hemen eklenir, en küçükse bütün liste gezilir. Ekleme bu yüzden $O(n)$'dir.

Maksimum heap. İki işlem de yalnız bir kök–yaprak yolu üzerinde çalışır: $O(\lg n)$.

Tablonun kalıbı açıktır. Basit yapıların her biri işlemlerden birini çok ucuz yapar, ama bunun bedelini öteki işlemde öder. Sırasız yapılar eklemeyi ucuz tutar, çünkü hiçbir düzen korumaz; bedeli, en büyüğü her seferinde yeniden aramaktır. Sıralı yapılar almayı ucuz tutar, çünkü tam bir düzen korur; bedeli, her eklemede bu düzeni yeniden kurmaktır. Heap ikisinin ortasında durur. Tam sıralama yapmaz, yalnız "her ebeveyn çocuklarından büyük" diye zayıf bir düzen korur. Bu düzen en büyüğü kökte tutmaya yeter ve onarılması yalnız bir yol boyunca iş ister.

Akla gelebilecek bir başka fikir, en büyük değeri ayrı bir `max` değişkeninde tutmaktır. Sırasız diziye ek olarak bu değişken her eklemede güncellenir ve en büyüğe bakmak $O(1)$ olur. Ama en büyük öğe alındığında, yani o uçak indiğinde ne yapılacaktır? Yeni en büyüğü bulmak için yine bütün diziye bakmak gerekir. Bunu önlemek için ikinci en büyüğü, onun için üçüncüyü de tutmak gerekirdi; bu da sıralı bir dizi kullanmaktan farksızdır.

13.6.2 Sayılarla: $n = 2.000.000$

Maliyet tablosu bir sayı üzerinde somutlaşır. $n = 2.000.000$ öğe olsun ve her temel işlem bir saniye sürsün. Bu varsayım abartılıdır, ama oranları görmeyi kolaylaştırır. $\lg 2.000.000 \approx 20{,}9$'dur; yuvarlanarak 21 alınır. 2 milyonun seçilmesinin nedeni de budur: logaritması kolay hesaplanır, çünkü $2^{21} = 2.097.152$'dir.

Yapı	Ekle	Sil
1 · sırasız dizi	1	2.000.000
2 · sırasız liste	1	2.000.000
3 · sıralı dizi	2.000.000	1
4 · sıralı liste	2.000.000	1
5 · maksimum heap	≈ 21	≈ 21

Tablo 13.11: $n = 2.000.000$ için en kötü durumda işlem başına adım sayısı (bir adım = bir saniye varsayımı).

Tablo 13.11 beş yaklaşımı hiç kod yazmadan, yalnız analizle karşılaştırır. İlk dört yapıda işlemlerden biri 2 milyon saniye, yani yaklaşık 23 gün sürer. Heap'te iki işlem de yaklaşık 21 saniyedir. Bir kuyrukta ekleme ve alma aşağı yukarı aynı sayıda yapıldığından, bir ekleme ile bir almanın toplamına bakmak doğru bir ölçüdür. Basit yapılarda bu toplam yaklaşık 2 milyon adım, heap'te 42 adımdır. Heap yaklaşık 48.000 kat hızlıdır. Bir benzetmeyle: heap ile gerçeklenen program kırk bin kez çalıştırıldığında, öteki yapılardan biriyle bir kez çalıştırmaktan daha kısa sürer.

Gerçek bir komut saniye değil, bir mikrosaniyenin küçük bir kesri kadar sürer. Bu yüzden şu itiraz yapılabilir: "Havaalanına yarım saatte bir uçak geliyor; bin uçaklık bir kuyrukta bile en kötü yapı göz açıp kapayıncaya kadar çalışır. Kim anlayacak algoritmanın kötü olduğunu?" Küçük bir problem için bu doğru olabilir. Ama aynı kod bir işletim sisteminin parçası olduğunda, saniyede binlerce kez çağrılır ve aradaki fark sistemin bütününe yayılır. Uzmanlık isteyen işlerde, iş yükü büyüyünce çöken bir çözümün akademik ve bilimsel bir değeri yoktur.

13.6.3 Ölçüm

Kuramsal karşılaştırma bir ölçümle de sınanmıştır. Üç gerçekleme C ile yazılmış ve `-O2` ile derlenmiştir: sırasız dizi, doğrusal kaydırmalı sıralı dizi ve maksimum heap. Her birine n rastgele değer eklenmiş, ardından n kez en büyük alınmıştır. Üç yapının verdiği değerlerin toplamları her çalıştırmada eşit çıkmıştır; yani üçü de aynı öğeleri vermiştir. Tablo 13.12 üç çalıştırmanın değer aralıklarını verir. Değerler makineye göre değişir; önemli olan büyüme biçimidir.

n	Sırasız dizi	Sıralı dizi	Maksimum heap
10.000	0,12–0,16	0,011–0,015	≈ 0,001
50.000	3,3–3,6	0,57–0,77	0,008–0,019
100.000	16,0–17,1	2,3–2,9	0,017–0,027

Tablo 13.12: n ekleme + n alma süresi (saniye; üç çalıştırma; makineye göre değişir).

n on kat artınca sırasız ve sıralı dizinin süresi yaklaşık yüz kat artmıştır. Bu karesel büyümedir: n işlemin her biri $\Theta(n)$ sürdüğünde toplam $\Theta(n^2)$ olur. Heap'in süresi ise kabaca $n \lg n$ ile büyür. Yüz bin öğede heap, sırasız diziden yaklaşık 700 kat, sıralı diziden yaklaşık 100 kat hızlıdır. Sıralı dizinin sırasız diziden hızlı çıkması da öğreticidir. İkisi de $\Theta(n^2)$ toplam iş yapar. Ama kaydırma ardışık bellek hücreleri üzerinde çalışan çok basit bir döngüdür; aynı büyüme sınıfındaki iki yöntem arasında sabit çarpan bu kadar farklı olabilir. Asimptotik sınıf ise sabit çarpandan güçlüdür: n büyüdükçe heap'le aradaki fark açılmaya devam eder.

13.6.4 İleri düzey heap'ler

Bu notta anlatılan yapı **ikili heap**'tir (*binary heap*). Daha ileri düzeyde başka heap türleri de vardır ve bazı işlemleri daha da hızlı yapar. Araştırmacılar $\lg n$'nin yettiğini düşünmemiştir. Olimpiyat sloganındaki gibi: *Citius! Altius! Fortius!* (Daha hızlı! Daha yükseğe! Daha güçlü!) Tablo 13.13 üç yapıyı minimum heap işlemleri üzerinden karşılaştırır.

İşlem	İkili heap	Binomial heap	Fibonacci heap*
ekle	$O(\lg n)$	$O(\lg n)$	$O(1)$
en küçüğü bul	$O(1)$	$O(\lg n)$	$O(1)$
en küçüğü sil	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
anahtar azalt	$O(\lg n)$	$O(\lg n)$	$O(1)$
iki heap'i birleştir	$O(n)$	$O(\lg n)$	$O(1)$

Tablo 13.13: İkili, binomial ve Fibonacci heap'lerinin işlem maliyetleri. * Fibonacci heap süreleri amortizedir.

Binomial heap bir ağaç değil, belli boyutlarda ağaçlardan oluşan bir ormandır. Asıl kazancı birleştirmedir: iki binomial heap $O(\lg n)$'de birleştirilebilir. İkili heap'te birleştirme için iki dizi birleştirilip yeniden heap kurulur; bu $O(n)$ sürer. **Fibonacci heap** işi daha da erteler. Ekleme ve anahtar azaltma sırasında yapıyı düzenlemez; düzenlemeyi en küçüğün silindiği ana bırakır. Sürelerinin amortize olması bu yüzdendir: tek bir işlem pahalı olabilir, ama bir işlem dizisinin toplam maliyeti işlem sayısına bölündüğünde tablodaki değerler çıkar. Dijkstra ve Prim gibi çok sayıda anahtar azaltma yapan çizge algoritmalarında Fibonacci heap kuramsal bir kazanç sağlar. Uygulamada ise ikili heap'in sadeliği ve diziyle önbellek dostu yerleşimi çoğu zaman daha hızlı sonuç verir.

13.7 Aşağıdan yukarı heap kurma

Bazen öğeler tek tek gelmez; hepsi baştan bir dizide hazırdır. Örneğin heap sort sıralanacak diziyi bir kerede alır. Böyle bir diziyi heap'e çevirmenin ilk akla gelen yolu, öğeleri boş bir heap'e tek tek eklemektir. Her ekleme *O(lg n)* sürdüğü için bu yol *O(n lg n)* sürer. Daha iyi bir yol vardır: diziyi yerinde, aşağıdan yukarı düzeltmek. Bu yönteme **aşağıdan yukarı heap kurma** (*bottom-up heap construction*, heapify) denir ve *O(n)* sürer.

13.7.1 `yigin_kur`

Fikrin çıkış noktası yapraklardır. Tek bir düğümden oluşan ağaç zaten bir heap'tir; karşılaştırılacak çocuğu yoktur. *n* öğeli bir dizide *k > ⌊n/2⌋* olan bütün düğümler yapraktır, çünkü sol çocukları *2k > n* konumunda olurdu. Demek ki dizinin ikinci yarısı, daha hiçbir şey yapılmadan, tek öğelik heap'lerden oluşur.

Şimdi *k = ⌊n/2⌋*'den köke doğru geriye gidilir. Her *k* için düğümün iki alt ağacı zaten heap'tir, çünkü daha büyük indisli bütün düğümler önceden düzeltilmiştir. Bu durumda yalnız *k*'daki öğe yerinde olmayabilir. Bu tam olarak silmedeki durumdur: kökte yeri aranan bir öğe ve iki heap. *k*'daki öğe aşağı kaydırılır ve *k* köklü alt ağaç bir heap olur. Döngü köke ulaştığında bütün dizi heap'tir.

```
void yigin_kur(eleman a[], int n) {
  for (int k = n / 2; k >= 1; k--) {
    eleman deger = a[k];
    int i = k, j = 2 * k;
    while (j <= n) {
      if (j < n && a[j].x < a[j + 1].x) j++;
      if (deger.x >= a[j].x) break;
      a[i] = a[j];
      i = j; j *= 2;
    }
    a[i] = deger;
  }
}
```

Dış döngü `k = n / 2`'den 1'e kadar geriye sayar. İç kısım `maksimum_yigin_al`'daki döngünün aynısıdır; tek fark başlangıç konumunun 1 değil *k* olmasıdır. Horowitz'in kitabında bu iç döngü `adjust` adlı ayrı bir yordamdır. `deger` *k*'daki öğeyi tutar; `i` boşluğu, `j` çocukları gösterir. Büyük çocuk `deger`'den büyük olduğu sürece yukarı taşınır ve boşluk aşağı iner. Sonunda `deger` boşluğa yazılır.

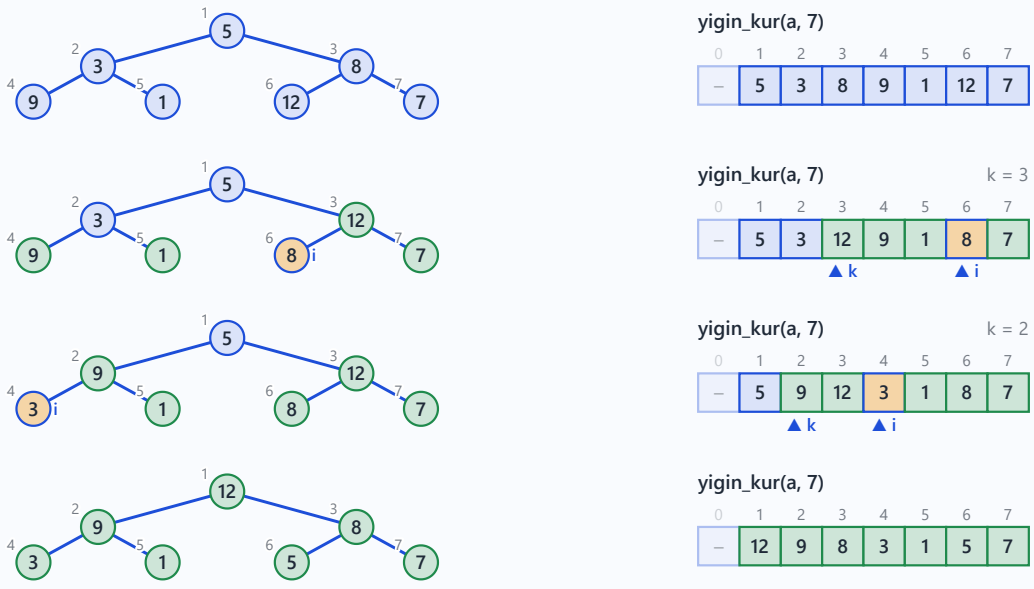
Yordam ek dizi kullanmaz; heap girdi dizisinin içinde kurulur. Bu yüzden ek belleği *O(1)*'dir. Böyle algoritmalara **yerinde** (*in-place*) algoritma denir. Dizi 1 tabanlıdır: öğeler `a[1]`'den `a[n]`'ye kadardır ve `a[0]` kullanılmaz. Çağıranın en az *n + 1* hücrelik bir dizi vermesi gerekir.

13.7.2 Kod izleme: `yigin_kur(a, 7)`

Girdi *5, 3, 8, 9, 1, 12, 7*'dir ve henüz heap değildir; kökteki 5, çocuklarından küçüktür. *n = 7*, *⌊n/2⌋ = 3*'tür. 4, 5, 6 ve 7. konumlar yapraktır. Dış döngü 3, 2 ve 1 için çalışır. Tablo 13.14 her *k* için iç döngüyü özetler.

`k`	`deger`	Adımlar	Yerleştiği konum	`a[1..7]` (sonra)
—	—	girdi	—	5, 3, 8, 9, 1, 12, 7
3	8	çocuklar 12 ve 7, büyüğü `a[6] = 12`; *8 < 12*: 12 yukarı; *j = 12 > n*	6	5, 3, 12, 9, 1, 8, 7
2	3	çocuklar 9 ve 1, büyüğü `a[4] = 9`; *3 < 9*: 9 yukarı; *j = 8 > n*	4	5, 9, 12, 3, 1, 8, 7
1	5	çocuklar 9 ve 12, büyüğü `a[3] = 12`; *5 < 12*: 12 yukarı. Çocuklar 8 ve 7, büyüğü `a[6] = 8`; *5 < 8*: 8 yukarı; *j = 12 > n*	6	12, 9, 8, 3, 1, 5, 7

Tablo 13.14: `yigin_kur(a, 7)` izlemesi.



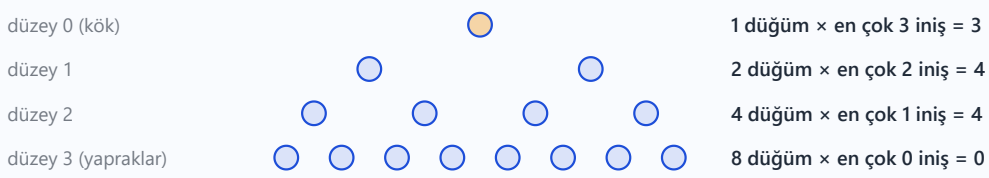
Şekil 13.13: `yigin_kur`'un dört anı. Yukarıdan aşağıya: girdi; *k = 3*, *k = 2* ve *k = 1* adımlarının sonu. Yeşil düğümler, kendi alt ağaçları zaten heap olan köklerdir.

k = 3'te 8, 12 ile yer değiştirmiş gibi görünür; aslında 12 yukarı taşınmış ve 8 boşluğa yazılmıştır. 3 köklü alt ağaç artık *12, 8, 7*'dir. *k = 2*'de benzer biçimde 9 yukarı çıkar ve 3 yaprağa iner. İki alt ağaç da heap olduktan sonra *k = 1*'de kök düzeltilir. 5'in yolu bu kez iki düzeydir: önce 12, sonra 8 yukarı taşınır ve 5, 6. konuma yerleşir. Dış döngü *k = 0*'da biter. Sonuç *12, 9, 8, 3, 1, 5, 7* geçerli bir maksimum heap'tir. Yordam bütünüyle 4 taşıma ve her *k* için bir son yazma, yani 3 son yazma yapmıştır.

Aynı yedi değer boş bir heap'e `maksimum_yigin_ekle` ile tek tek eklenseydi sonuç *12, 8, 9, 3, 1, 5, 7* olurdu. Bu da geçerli bir maksimum heap'tir, ama `yigin_kur`'un verdiğinden farklıdır. Aynı değer kümesi için birden çok heap vardır; iki yöntem farklı birine ulaşır. Hangisinin elde edildiği önemli değildir. İkisinde de kök en büyük öğedir ve sonraki almalar aynı sırayı verir.

13.7.3 Neden *O(n)*?

İlk bakışta `yigin_kur`'un maliyeti şöyle hesaplanabilir: dış döngü *n/2* kez döner, her seferinde iç döngü *O(lg n)* sürer; toplam *O(n lg n)*. Bu sınır doğrudur ama gevşektir. Asıl gözlem, düğümlerin büyük çoğunluğunun ağacın alt kısmında olması ve oradaki düğümlerin çok az inebilmesidir.



Şekil 13.14: 15 düğümlü tam ağaçta düzeylere göre en çok iniş sayıları. Kök 3 düzey inebilir; yapraklar hiç inmez.

Bir düğüm en çok, altındaki düzey sayısı kadar inebilir. Bu sayıya düğümün **yüksekliği** denir; yaprakların yüksekliği 0'dır. Düğümlerin yaklaşık yarısı yapraktır ve hiç inmez. Dörtte biri yaprakların hemen üstündedir ve en çok 1 düzey iner. Sekizde biri en çok 2 düzey iner. Yalnız tek bir düğüm, kök, *⌊lg n⌋* düzey inebilir. Şekil 13.14 15 düğümlü bir tam ağaç için bu hesabı yapar: $0 \cdot 8 + 1 \cdot 4 + 2 \cdot 2 + 3 \cdot 1 = 11$. Toplam iniş en çok 11'dir ve bu sayı *n = 15*'ten küçüktür.

> **Teorem 13.3 (Aşağıdan yukarı kurmanın maliyeti)**
>
> `yigin_kur`, *n* öğeli bir diziyi *O(n)* sürede maksimum heap'e çevirir. Toplam iniş sayısı *n*'den azdır.

Kanıt. n öğeli bir heap'te yüksekliği h olan düğüm sayısı en çok $\lceil n / 2^{h+1} \rceil$'dir. Yüksekliği h olan bir düğümün aşağı kaydırılması en çok h tur sürer ve her tur sabit iş yapar. Toplam iş şöyle sınırlanır:

$$\sum_{h \ge 0} \lceil n / 2^{h+1} \rceil \cdot O(h) = O(n \cdot \sum_{h \ge 0} h / 2^h) = O(2n) = O(n)$$

Burada $\sum h / 2^h = 1/2 + 2/4 + 3/8 + \dots = 2$'dir. Bu toplamın değeri şöyle görülür. $S = \sum_{h \ge 1} h / 2^h$ olsun. $S / 2 = \sum_{h \ge 1} h / 2^{h+1} = \sum_{h \ge 2} (h - 1) / 2^h$'dir. Çıkarılınca $S - S/2 = 1/2 + \sum_{h \ge 2} 1/2^h = 1/2 + 1/2 = 1$, yani $S = 2$ bulunur. Tam bir ağaçta daha kesin bir sayım da yapılabilir: $2^m - 1$ düğümlü mükemmel ağaçta toplam iniş en çok $n - m$'dir, yani n'den azdır. 15 düğümlü örnekte bu $15 - 4 = 11$'dir. ■

Art arda ekleme neden aynı ucuzluğu sağlamaz? Eklemede hareket yukarı doğrudur ve bir düğümün çıkabileceği düzey sayısı onun *derinliğidir*, yani kökten uzaklığı. Düğümlerin yarısı en alt düzeydedir ve her biri köke kadar $\lfloor \lg n \rfloor$ düzey çıkabilir. Kalabalık olan alt düzeyler en uzun yolu yürüyebilir; toplam $\Theta(n \lg n)$'ye ulaşır. Aşağıdan yukarı kurmada durum tersinedir: uzun yolu yalnız az sayıdaki üst düğümler yürür.

İki yöntem bir ölçümle karşılaştırılmıştır. Rastgele dizilerde `yigin_kur`'un karşılaştırma ve taşıma sayıları sayılmış, aynı değerler bir de tek tek eklenmiştir. Tablo 13.15 sonuçları verir.

n	`yigin_kur` karşılaştırma	`yigin_kur` taşıma	Ekleme (rastgele) kaydırma	Ekleme (artan) kaydırma
1.000	1.893	762	1.337	7.987
10.000	18.841	7.460	12.898	113.631
100.000	188.069	74.406	128.122	1.468.946
1.000.000	1.881.022	743.338	1.279.942	17.951.445
10.000.000	18.815.235	7.441.057	12.817.582	213.222.809

Tablo 13.15: Aşağıdan yukarı kurma ile art arda eklemenin sayımları. Son sütun artan sırada (en kötü durum) eklemedeki kaydırma sayısıdır.

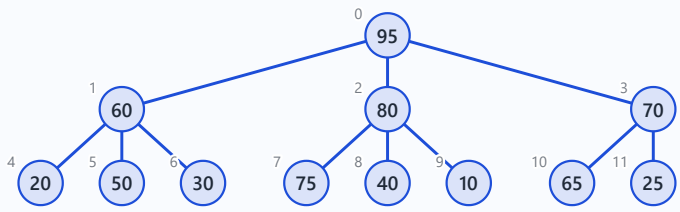
`yigin_kur`'un sayıları n ile tam orantılı büyür: karşılaştırma yaklaşık $1{,}88n$, taşıma yaklaşık $0{,}74n$'dir. Artan sırada eklemede kaydırma sayısı ise n on kat artınca on kattan fazla artar; bin öğede öğe başına 8, on milyonda öğe başına 21 kaydırma düşer. Bu $n \lg n$ büyümesidir. Rastgele sırada eklemenin ortalaması öğe başına yaklaşık 1,3 kaydırmada kalır; ama bu ortalama bir garanti değildir. `yigin_kur` ise her girdide n'den az taşıma yapar. On milyon öğede iki yöntemin süresi de bu makinede 0,1 saniye dolayında ölçülmüştür (`yigin_kur` 0,09–0,12 s, rastgele ekleme 0,13–0,15 s).

13.8 d-ary heap

İkili heap'te her düğümün en çok iki çocuğu vardır. Bu sayı 2 olmak zorunda değildir. Her düğümün en çok d çocuğu olan tam ağaçla da heap kurulabilir.

Tanım 13.4

d-ary heap, her düğümünün en çok d çocuğu olan, son düzey dışında dolu ve son düzeyi soldan sağa dolan bir ağaçtır; her düğümün değeri çocuklarınınkinden küçük değildir (maksimum sürüm). $d = 2$ ikili heap'i verir.



Şekil 13.15: 12 öğeli bir 3-ary maksimum heap. İndisler 0'dan başlar; i. düğümün çocukları $3i + 1$, $3i + 2$, $3i + 3$'tür.

d-ary heap de dizide saklanır. 0 tabanlı dizide i. düğümün çocukları $d \cdot i + 1$'den $d \cdot i + d$'ye kadardır, ebeveyni $\lfloor (i - 1)/d \rfloor$'dir. $d = 2$ için bu formüller 0 tabanlı ikili heap'inkilerle aynıdır. Şekil 13.15 bir 3-ary heap'i gösterir. Kök 95'in çocukları 1, 2 ve 3. indislerdeki 60, 80 ve 70'tir. 2. indisteki 80'in çocukları $3 \cdot 2 + 1 = 7$'den 9'a kadardır: 75, 40 ve 10. 3. indisteki 70'in yalnız iki çocuğu vardır, 10 ve 11. indislerde: 65 ve 25. Ağaç tamdır; son düzey soldan sağa dolmuştur.

	İkili ($d = 2$)	d-ary
ebeveyn	`(i - 1) / 2`	`(i - 1) / d`
çocuklar	`2i + 1`, `2i + 2`	`d·i + 1` … `d·i + d`
yükseklik	$\approx \lg n$	$\approx \log_d n$
ekle / anahtarı köke doğru değiştir	$O(\lg n)$	$O(\log_d n)$: daha hızlı
kökü sil	$O(\lg n)$	$O(d \cdot \log_d n)$: d çocuk karşılaştırılır

Tablo 13.16: İkili heap ile d-ary heap (0 tabanlı dizi).

Çocuk sayısı artınca ağaç sığlaşır: yükseklik $\lg n$'den $\log_d n = \lg n / \lg d$'ye iner. Örneğin bir milyon öğede ikili heap'in yüksekliği 19, 4-ary heap'inki yaklaşık 10'dur. Yukarı kaydırma her düzeyde yalnız bir karşılaştırma yaptığı için ekleme hızlanır. Aşağı kaydırma ise her düzeyde d çocuğun en büyüğünü bulmak zorundadır; bu $d - 1$ karşılaştırma ister. Kökü silmek bu yüzden $O(d \cdot \log_d n)$ sürer ve d büyüdükçe yavaşlar.

Sunumun tablosu maksimum heap'ten söz ettiği için yukarı kaydırma gerektiren işlemi “anahtar artırma” diye adlandırır. Minimum heap'te aynı işlem “anahtar azaltma”dır. Dijkstra'nın en kısa yol algoritması bir minimum heap kullanır ve çok sayıda anahtar azaltma, görece az sayıda silme yapar. Kenar sayısının düğüm sayısına göre çok olduğu yoğun çizgelerde bu oran belirgindir. Böyle durumlarda $d > 2$ seçmek kazançlıdır. Sığ ağacın bir yararı da önbellek dostu olmasıdır: bir düğümün d çocuğu dizide yan yana durur ve çoğu kez aynı önbellek satırına düşer.

13.9 Uygulamalar ve heap sort

Öncelikli kuyruk, “sıradaki en önemli iş hangisi?” sorusunun tekrar tekrar sorulduğu her yerde kullanılır. Sunum altı alan sayar.

- **İşletim sistemi.** İşlemlerin ve kesmelerin zamanlanması, zamanlayıcı olayları. En yakın zamanda dolacak zamanlayıcı bir minimum heap'in kökündedir.
- **Çizge algoritmaları.** Dijkstra'nın en kısa yol algoritması ve Prim'in minimum kapsayan ağaç algoritması, her adımda henüz işlenmemiş en yakın düğümü seçer. Bu seçim bir minimum heap'le yapılır.
- **Heap sort.** Heap kurulur, n kez kök alınır: $O(n \lg n)$, yerinde sıralama.
- **Olay simülasyonu.** Olaylar zaman damgalarına göre bir minimum heap'ten çekilir; her olay gelecekteki yeni olaylar ekleyebilir.
- **En büyük ya da en küçük k öğe.** Akan bir veride en büyük k öğe, boyutu k olan bir minimum heap'le izlenir. Yeni gelen değer kökten büyükse kök atılır, yeni değer eklenir. Her adım $O(\lg k)$, bütün akış $O(n \lg k)$ sürer. En küçük k öğe için aynı iş bir maksimum heap'le yapılır.
- **Huffman kodlaması.** Metin sıkıştırmada en küçük frekanslı iki ağaç art arda birleştirilir. Her adımda en küçük ikisini bulmak bir minimum heap işidir.

Bu uygulamalardan en kısası heap sort'tur ve notta incelenen bütün parçaları bir araya getirir. Art arda almanın azalan sırayı verdiği görülmüştü. Bu fikri ek dizi kullanmadan uygulamak için bir gözlem yeter: kök alındığında heap bir öğe küçülür ve dizinin sonunda bir hücre boşalır. Alınan en büyük öğe tam o hücreye konabilir. Böylece dizinin sonu, sıralanmış öğelerin biriktiği bölge olur. Aşağıdaki yordam bu fikri uygular:

```c
/* a[1..n] küçükten büyüğe sıralanır; ek dizi kullanılmaz */
void heap_sort(eleman a[], int n) {
  yigin_kur(a, n);                /* O(n) */
  for (int m = n; m > 1; m--) {
    eleman deger = a[m];          /* sondaki öğe */
    a[m] = a[1];                  /* en büyük sona */
    int i = 1, j = 2;             /* a[1..m-1] düzeltilir */
    while (j <= m - 1) {
      if (j < m - 1 && a[j].x < a[j + 1].x) j++;
      if (deger.x >= a[j].x) break;
      a[i] = a[j];
      i = j; j *= 2;
    }
    a[i] = deger;
  }
}
```

Yordam önce diziyi `yigin_kur` ile heap'e çevirir. Ardından `m`, heap'in o anki boyutu olarak n'den 2'ye iner. Her turda sondaki öğe `deger`'e alınır, kökteki en büyük öğe `a[m]`'ye yazılır ve `deger`'e `a[1..m-1]` içinde kökten başlayarak yer aranır. Bu son kısım `maksimum_yigin_al`'ın döngüsüdür; yalnız heap'in sonu `m - 1`'dir. Döngü bittiğinde `a[2..n]` sıralı ve en küçük öğe `a[1]`'dedir.

5, 3, 8, 9, 1, 12, 7 dizisi bu yordamla *1, 3, 5, 7, 8, 9, 12*'ye sıralanmıştır. Yordam ayrıca 0–149 öğelik, yinelenen değerler içeren 3000 rastgele diziyle sınanmış ve her seferinde artan sıra vermiştir. Maliyet, `yigin_kur` için $O(n)$ ile $n - 1$ aşağı kaydırma için $O(n \lg n)$'nin toplamıdır: $O(n \lg n)$. Bu süre en kötü durumda da geçerlidir. Ek bellek yalnız birkaç değişkendir: $O(1)$.

13.10 Karmaşıklık özeti

İşlem	Zaman	Ek bellek	Açıklama
en büyüğe bak	$O(1)$	$O(1)$	`yigin[1]`
ekle	$\Omega(1) \ldots O(\lg n)$	$O(1)$	yukarı kaydırma
en büyüğü al	$O(\lg n)$	$O(1)$	aşağı kaydırma
önceliği değiştir	$O(\lg n)$	$O(1)$	konum biliniyorsa
aşağıdan yukarı kurma	$O(n)$	$O(1)$	yerinde
n kez ekleyerek kurma	$O(n \lg n)$	$O(1)$	en kötü durum artan sıra
heap sort	$O(n \lg n)$	$O(1)$	yerinde
keyfi öğeyi ara	$O(n)$	$O(1)$	heap arama yapısı değildir

Tablo 13.17: n öğeli ikili heap'te işlemlerin maliyeti.

Tablo 13.17 notta incelenen işlemleri toplar. Tablonun ortak bir gerekçesi vardır: heap'te iş bir kök–yaprak yolunun uzunluğuyla sınırlıdır ve bu uzunluk Teorem 13.2 gereği $\lfloor \lg n \rfloor$'dir. En büyüğe bakmak yol yürümeyi gerektirmez. Aşağıdan yukarı kurma, yolların çoğu kısa olduğu için doğrusaldır. Arama ise heap özelliğinden yararlanamaz: aranan değer kökten küçükse herhangi bir alt ağaçta olabilir. Ek bellek bütün satırlarda sabittir, çünkü heap dizinin kendisidir ve yordamlar yalnız birkaç indis değişkeni kullanır.

13.11 Sık yapılan hatalar

1. **1 tabanlı ve 0 tabanlı formülleri karıştırmak.** 1 tabanlı dizide ebeveyn `i/2`, çocuklar `2i` ve `2i+1`'dir; 0 tabanlı dizide ebeveyn `(i-1)/2`, çocuklar `2i+1` ve `2i+2`'dir. Karıştırılınca yanlış düğümler karşılaştırılır ve heap sessizce bozulur.
2. **Dizi boyutunu kapasiteden bir eksik tanımlamak.** `eleman yigin[ELEMAN_SAYISI - 1]` ile 199. öğe `yigin[199]`'a, dizinin dışına yazılır.
3. `eleman` **döndüren yordamda** `return 0;` **yazmak.** Bir tam sayı bir yapıya dönüşmez; derleme hatasıdır. Başarı `bool` ile, öğe işaretçi parametreyle döndürülür.
4. `j < n` **sınamasını unutup** `yigin[j+1]` **'i okumak.** Okunan hücre heap'in dışındadır; `yigin_kur`'da dizinin de dışında olabilir.

5. **Silmede büyük çocuk yerine ilk çocuğu seçmek.** Küçük çocuk yukarı çıkınca büyük kardeşinin ebeveyni olur ve heap özelliği bozulur.
6. **Heap'i sıralı dizi ya da ikili arama ağacı sanmak.** Yalnız `yigin[1]` 'in en büyük olduğu garantidir; kardeşler arasında sıra yoktur ve heap'te arama *O(n)* sürer.
7. **Ekleme döngüsünü kökte durdurmamak.** Koşul `i != 1` yerine `i >= 1` yazılırsa `yigin[0]` okunur ve yeni öğe heap'in dışına yazılabilir.
8. **Öncelik değişince öğeyi yanlış yöne kaydırmak.** Maksimum heap'te önceliği artan öğe yukarı, azalan öğe aşağı kaydırılır. `oncelik_artir` 'ı küçük bir değerle çağırmak heap'i bozabilir.

13.12 Sınav tarzı soru

> **Örnek 13.4**
>
> Başlangıçta `yigin[1..6] = 14, 12, 7, 10, 8, 6` olan maksimum heap'e sırasıyla 15, 109, 107, 3, 15 ekleniyor; ardından `maksimum_yigin_al` iki kez çağrılıyor. (i) Eklemelerden sonra dizinin içeriği nedir? (ii) Alınan öğeler ve son dizi nedir? (iii) Hangi ekleme en az, hangisi en çok kaydırma yaptı?

(i) Beş ekleme Tablo 13.4'te adım adım izlendi. Son dizi *109, 107, 14, 15, 15, 6, 7, 10, 12, 3, 8*, *n = 11*'dir. Sınavda her eklemenin yolunu yazmak yeterlidir: yeni konum *n + 1*, sonra ebeveyniyle karşılaştırma ve gerekirse ebeveyne çıkış.

(ii) Birinci alma kökteki 109'u verir. Sondaki öğe 11. konumdaki 8'dir; *n* 10'a iner. Kökün çocukları 107 ve 14'tür; 107 köke çıkar ve boşluk 2'ye iner. 2'nin çocukları 4. ve 5. konumdaki iki 15'tir. Sol çocuk sağdakinden küçük olmadığı için `j` 4'te kalır; *8 < 15* olduğundan 4. konumdaki 15 yukarı çıkar ve boşluk 4'e iner. 4'ün çocukları 10 ve 12'dir; 12 büyüktür ve *8 < 12*'dir, 12 yukarı çıkar ve boşluk 9'a iner. 9'un sol çocuğu 18. konumda olurdu; *18 > 10*'dur. 8, 9. konuma yazılır. Dizi *107, 15, 14, 12, 15, 6, 7, 10, 8, 3* olur.

İkinci alma 107'yi verir. Sondaki öğe 10. konumdaki 3'tür; *n* 9'a iner. Kökün çocukları 15 ve 14'tür; 15 köke çıkar, boşluk 2'ye iner. 2'nin çocukları 12 ve 15'tir; sağdaki büyüktür, 15 yukarı çıkar ve boşluk 5'e iner. 5'in sol çocuğu 10. konumda olurdu ve *10 > 9*'dur; 3, 5. konuma yazılır. Alınanlar 109 ve 107'dir; son dizi *15, 15, 14, 12, 3, 6, 7, 10, 8*, *n = 9*'dur.

(iii) 3'ün eklenmesi hiç kaydırma yapmadı: ebeveyni 8'den küçüktü. Bu en iyi durumdur, *Ω(1)*. 109 ise köke kadar çıktı ve 3 kaydırma yaptı; 8 öğeli heap'te bu, yüksekliğe eşit olan en kötü durumdur, *O(lg n)*. Bu çözümdeki bütün diziler, sunumdaki kodları birebir kullanan bir test programının çıktısıyla aynıdır.

13.13 Alıştırmalar ve çözümleri

Sunumun altı alıştırması burada çözümleriyle verilir. Çözümlerdeki diziler test programlarıyla doğrulanmıştır.

13.13.1 Boş heap'e beş ekleme

Soru. Boş bir maksimum heap'e 20, 5, 30, 10, 40 eklenirse her eklemeden sonra dizi ne olur?

Çözüm. 20 köke yazılır: *20*. 5, 2. konuma gelir ve ebeveyni 20'den küçüktür: *20, 5*. 30, 3. konuma gelir; ebeveyni 20'den büyük olduğu için 20 aşağı iner ve 30 köke çıkar: *30, 5, 20*. 10, 4. konuma gelir; ebeveyni 2. konumdaki 5'ten büyüktür, 5 aşağı iner; yeni ebeveyni 30'dan küçüktür ve 10 2. konumda durur: *30, 10, 20, 5*. 40, 5. konuma gelir; ebeveyni 10'dan, sonra 30'dan büyüktür ve köke çıkar: *40, 30, 20, 5, 10*.

13.13.2 Minimum heap için alma

Soru. `maksimum_yigin_al` minimum heap için yazılırsa hangi satırlar değişir?

Çözüm. Yordamın adı `minimum_yigin_al` olur. Çocuk seçiminde küçük çocuk aranır: `if (j < n && yigin[j].x > yigin[j + 1].x) j++;` . Durma koşulu `if (deger.x <= yigin[j].x) break;` olur. Geri kalan her şey aynıdır. Tam kod 13.5.2'de verilmiştir.

13.13.3 Bin öğeli heap'te ekleme

Soru. n = 1000 öğeli bir heap'te ekleme en çok kaç kaydırma yapar?

Çözüm. 1000 öğeli tam ikili ağacın düzey sayısı $\lfloor \lg 1001 \rfloor = 10$'dur: $2^9 = 512 \le 1000 < 1024$. Bir yaprak ile kök arasında en çok 9 kenar vardır. Yaprak düzeyinden köke çıkan bir öğe her kenar için bir kaydırma yapar; en çok 9 kaydırma, yani $\lfloor \lg 1000 \rfloor = 9$. Yeni öğe 1001. konuma gelecekse yol 1001 → 500 → 250 → 125 → 62 → 31 → 15 → 7 → 3 → 1'dir ve gerçekten 9 kenardır.

13.13.4 13. düğümün akrabaları

Soru. 1 tabanlı bir heap'te 13. düğümün ebeveyni ve çocukları hangi indislerdedir?

Çözüm. Ebeveyn $\lfloor 13/2 \rfloor = 6$'dır. Çocuklar $2 \cdot 13 = 26$ ve 27'dir; bunlar ancak heap'te yeterince öğe varsa bulunur (sol çocuk için $n \ge 26$, sağ çocuk için $n \ge 27$). Ayrıntılı hesap Örnek 13.1'dedir.

13.13.5 Kurma ile art arda ekleme

Soru. $5, 3, 8, 9, 1, 12, 7$ dizisine `yigin_kur` yerine art arda `ekle` uygulanırsa sonuç aynı olur mu?

Çözüm. Hayır. `yigin_kur` $12, 9, 8, 3, 1, 5, 7$ verir. Art arda ekleme ise şöyle ilerler: 5; $5, 3$; $8, 3, 5$; $9, 8, 5, 3$; $9, 8, 5, 3, 1$; $12, 8, 9, 3, 1, 5$; $12, 8, 9, 3, 1, 5, 7$. İki sonuç da geçerli maksimum heap'tir, ama farklıdır. Aynı değer kümesinin birden çok heap düzeni vardır.

13.13.6 En küçük öğe nerede?

Soru. Bir maksimum heap'te en küçük öğe nerelerde olabilir? Onu bulmanın maliyeti nedir?

Çözüm. Değerler farklıysa en küçük öğe yalnız bir yaprakta olabilir. Çocuğu olan bir düğüm, heap özelliği gereği en az bir değerden, çocuğundan büyüktür; en küçük olamaz. Yapraklar $\lfloor n/2 \rfloor + 1$'den n'ye kadarki indislerdir. Yinelenen değerler varsa en küçük değer bir iç düğümde de bulunabilir, ama o değerin bir kopyası mutlaka bir yaprakta da vardır. Her iki durumda da yaprakları taramak yeter. Yaklaşık $n/2$ yaprak vardır ve hepsine bakmak gerekir, çünkü yapraklar arasında hiçbir sıra yoktur. Maliyet $\Theta(n)$'dir. Heap en büyüğü bulmayı $O(1)$ yapar, ama en küçüğü bulmayı hızlandırmaz.

13.14 Özet

- **Öncelikli kuyruk**, ilk alınacak öğenin geliş sırasına göre değil önceliğe göre belirlendiği kuyruktur: maksimum sürümde en yüksek, minimum sürümde en düşük değerli öğe önce alınır.
- Bir soyut veri tipi olarak ekleme, alma ve bakma işlemleri vardır; sırasız ya da sıralı dizi, liste ya da heap ile gerçeklenebilir.
- **Heap** tam ikili ağaç ile "ebeveyn ≥ çocuk" kuralının birleşimidir. 1 tabanlı dizide ebeveyn `i/2`, çocuklar `2i` ve `2i+1`'dir; işaretçiye gerek yoktur.
- Heap ikili arama ağacı değildir; kardeşler arasında sıra yoktur. Yalnız kökün en büyük olduğu garantidir.
- **Ekleme**: ilk boş konum açılır, büyük atalar birer düzey aşağı kaydırılır, öğe en sonda bir kez yazılır. En iyi durum $\Omega(1)$, en kötü durum $O(\lg n)$.
- **Silme**: kök alınır, sondaki öğeye kökten başlayarak yer aranır, büyük çocuklar yukarı alınır: $O(\lg n)$. `j < n` sınaması sağ çocuğun varlığını denetler.
- Tam ikili ağacın yüksekliği $\lfloor \lg n \rfloor$'dir; heap işlemlerinin logaritmik olmasının gerekçesi budur.
- Global dizi yerine yapı kullanmak birden çok heap'e ve çalışma sırasında seçilen kapasiteye izin verir. Minimum heap için karşılaştırmalar ters çevrilir. Önceliği değişen öğe, konumu biliniyorsa $O(\lg n)$'de yukarı ya da aşağı kaydırılır.
- Basit yapılar işlemlerden birini $\Theta(n)$'e iter; heap ikisini de logaritmik tutar. Ölçümde yüz bin öğede heap, sırasız diziden yaklaşık 700 kat hızlı çıkmıştır.
- **Aşağıdan yukarı kurma** bir diziyi yerinde ve $O(n)$ sürede heap'e çevirir; art arda ekleme en kötü durumda $O(n \lg n)$ sürer.
- **d-ary heap** ağacı sığlaştırır ve eklemeyi hızlandırır; binomial ve Fibonacci heap'leri birleştirme ve anahtar azaltmayı hızlandırır. Heap sort, öncelikli kuyruğun sıralamaya uygulanmasıdır: $O(n \lg n)$, yerinde.

13.15 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

öncelikli kuyruk *priority queue* · soyut veri tipi *abstract data type (ADT)* · anahtar *key* · tam ikili ağaç *complete binary tree* · ebeveyn *parent* · düzey *level* · yükseklik *height* · heap *heap* · max-heap *max-heap* · min-heap *min-heap* · ikili arama ağacı *binary search tree*

up-heap (yukarı süzme) *up-heap* · down-heap (aşağı süzme) *down-heap* · aşağıdan yukarıya heap oluşturma *bottom-up heap construction (heapify)* · anahtar azaltma *decrease key* · Fibonacci heap *Fibonacci heap* · heap sort *heap sort* · yerinde sıralama *in-place sort* · amortize analiz *amortized analysis* · önbellek *cache* · heap bellek *heap memory*

13.16 Kendini Dene

1. Boş bir maksimum heap'e sırasıyla 4, 9, 2, 7, 11 ekleniyor. Son dizi nedir? Kaç kaydırma yapılır?

Cevap

Diziler: *4*; *9, 4*; *9, 4, 2*; *9, 7, 2, 4*; *11, 9, 2, 4, 7*. Kaydırmalar: 9 için 1 (4 aşağı), 2 için 0, 7 için 1 (4 aşağı), 11 için 2 (7 ve 9 aşağı); toplam 4.

2. *50, 30, 40, 10, 20, 35* heap'inden `maksimum_yigin_al` bir kez çağrılıyor. Alınan öğe ve yeni dizi nedir?

Cevap

50 alınır. Sondaki 35 çıkarılır, $n = 5$. Kökün çocukları 30 ve 40'tır; 40 köke çıkar ve boşluk 3'e iner. 3'ün sol çocuğu 6. konumda olurdu, ama $6 > 5$'tir. 35, 3. konuma yazılır: *40, 30, 35, 10, 20*.

3. 100 öğeli 1 tabanlı bir heap'te 37. düğümün ebeveyni, çocukları ve düzeyi nedir?

Cevap

Ebeveyn $\lfloor 37/2 \rfloor = 18$, çocuklar 74 ve 75'tir (ikisi de ≤ 100, yani vardır). Düzey $\lfloor \lg 37 \rfloor = 5$'tir, çünkü $32 \le 37 < 64$'tür.

4. `yigin_kur` *1, 2, 3, 4, 5, 6, 7* dizisini neye çevirir? Kaç taşıma yapılır?

Cevap

$k = 3$: 3'ün çocukları 6 ve 7; 7 yukarı, 3 aşağı. $k = 2$: 2'nin çocukları 4 ve 5; 5 yukarı, 2 aşağı. $k = 1$: kökün çocukları 5 ve 7; 7 yukarı; ardından 6 ve 3 arasından 6 yukarı; 1, 6. konuma iner. Sonuç *7, 5, 6, 4, 2, 1, 3*; 4 taşıma.

5. Minimum heap *3, 6, 7, 14, 8, 12, 15*'e 2 ekleniyor. Yol ve son dizi nedir?

Cevap

Yeni konum 8'dir. Atalar sırasıyla 14 (4. konum), 6 (2. konum) ve 3 (kök); 2 hepsinden küçük olduğu için üçü de aşağı iner. Yol 8 → 4 → 2 → 1, 3 kaydırma. Son dizi *2, 3, 7, 6, 8, 12, 15, 14*.

6. Değerleri farklı bir maksimum heap'te ikinci ve üçüncü büyük öğeler hangi indislerde olabilir?

Cevap

İkinci büyük öğenin ebeveyni yalnız kök olabilir, çünkü ebeveyni ondan büyük olmalıdır ve ondan büyük tek öğe köktür: indis 2 ya da 3. Üçüncü büyük öğenin ebeveyni kök ya da ikinci büyük öğedir: indis 2, 3 ya da ikinci büyüğün çocukları, yani 4, 5, 6, 7'den biri.

7. 31 öğeli bir maksimum heap'e yeni bir öğe ekleniyor. En çok kaç kaydırma yapılabilir? En az kaç?

Cevap

Yeni öğe 32. konuma gelir; bu yeni bir düzeyin ilk konumudur. Yol 32 → 16 → 8 → 4 → 2 → 1, yani en çok $\lfloor \lg 32 \rfloor = 5$ kaydırma. Yeni öğe 16. konumdaki ebeveyninden küçük ya da ona eşitse hiç kaydırma yapılmaz: en az 0.

8. 0 tabanlı bir 3-ary heap'te 5. düğümün ebeveyni ve çocukları hangi indislerdedir?

Cevap

Ebeveyn $\lfloor (5 - 1)/3 \rfloor = 1$'dir. Çocuklar $3 \cdot 5 + 1 = 16$, 17 ve 18'dir.

9. Bir öncelikli kuyruğa önce bir milyon öğe eklenecek, sonra yalnız en büyük on öğe alınacaktır. Sırasız dizi mi, heap mi daha uygundur?

Cevap

Sırasız dizide bir milyon ekleme ucuzdur (her biri $\Theta(1)$), ama on alma her biri yaklaşık bir milyon karşılaştırma ister: toplam yaklaşık on milyon. Heap'te öğeler önce diziye konup `yigin_kur` ile n'den az taşımayla heap yapılabilir; on alma yaklaşık $10 \cdot 2 \cdot 19 = 380$ karşılaştırma ister. Sırasız dizi toplam yaklaşık 11 milyon, heap yaklaşık 2 milyon adım yapar (kurmada yaklaşık $1{,}9n$ karşılaştırma). Heap yine üstündür; ama fark beş kat kadardır, çünkü işin çoğu girişte yapılır. Alma sayısı arttıkça heap'in üstünlüğü büyür.

13.17 İleri okuma

- E. Horowitz, S. Sahni ve S. Anderson-Freed, *Fundamentals of Data Structures in C*: heap'lerin, `adjust` yordamının ve heap sort'un bu notta izlenen kaynağıdır.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms*, Bölüm 6: heap kurmanın $O(n)$ analizini, heap sort'u ve öncelikli kuyruk işlemlerini kanıtlarıyla verir; Fibonacci heap'lerine ayrılan bölüm amortize sürelerin kaynağıdır.
- R. Sedgewick, *Algorithms in C*, Kısım 1–4, Bölüm 9: öncelikli kuyruğun dizi, liste ve heap gerçeklemelerini ve öğeleri indisle izleyen dolaylı heap'i anlatır.
- J. W. J. Williams, "Algorithm 232: Heapsort", *Communications of the ACM* 7(6), 1964 ve R. W. Floyd, "Algorithm 245: Treesort 3", *Communications of the ACM* 7(12), 1964: ikili heap'i ve heap sort'u tanıtan, aşağıdan yukarı kurmayı ekleyen iki kısa özgün yayındır.

13.18 Kaynaklar

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, Bölüm 5.6 (Heaps), Bölüm 7 (Heap Sort).
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (CLRS), MIT Press, Bölüm 6 (Heapsort, öncelikli kuyruklar) ve Fibonacci heap'leri bölümü.
3. R. Sedgewick, *Algorithms in C*, Kısım 1–4, Bölüm 9 (Priority Queues and Heapsort).
4. J. W. J. Williams, "Algorithm 232: Heapsort", *Communications of the ACM*, 7(6), 347–348, 1964.
5. R. W. Floyd, "Algorithm 245: Treesort 3", *Communications of the ACM*, 7(12), 701, 1964.

KISIM IV

Hash, Çizgeler ve Kümeler

Bölüm 14–18

KISIM IV · BÖLÜM 14

Hash Tabloları

Sıralı dizide ikili arama ve dengeli arama ağacında arama, anahtarları birbiriyle karşılaştırarak ilerler. Bu yüzden n öğeli bir yapıda $lg\ n$ mertebesinde adımın altına inemez. Hash tablosu başka bir yol izler: anahtarı karşılaştırmaz, anahtardan öğenin dizideki konumunu hesaplar. Hesap birkaç aritmetik işlem sürdüğü için arama, ekleme ve silme ortalamada sabit zamanda biter. Not önce bu fikrin en yalın biçimini, doğrudan adreslemeyi kurar. Ardından anahtarı küçük bir indis aralığına indiren hash fonksiyonlarını tek tek hesaplar: bölme, çarpma, katlama, orta-kare ve dizgiler için polinom hash'i. İki farklı anahtar aynı konuma düşebilir; bu çakışmayı çözmenin iki ana yolu vardır. Açık adreslemede öğe tablonun başka bir boş yuvasına yerleşir; doğrusal yoklama, karesel yoklama ve çift hash bu yuvayı farklı sıralarla arar. Zincirlemede ise her kova bir bağlı listenin başıdır. Her yöntem C ile yazılır ve kodu adım adım izlenir. Son kesimler açık adreslemede silmenin tuzağını, yük faktörünün maliyete etkisini ve tablonun büyütülmesini ele alır.

Kod (C): `github.com/sercankulcu/data-structures-and-algorithms-c/Bolum14`

ÖĞRENME ÇIKTILARI

Fikri açıklamak
Anahtardan konumu hesaplayarak ortalama $O(1)$ arama yapan hash tablosunu tanımlamak; doğrudan adreslemeden farkını ve bedelini söylemek.

Fonksiyon seçmek
Bölme, çarpma, katlama, orta-kare ve dizgi hash'ini elle hesaplamak; iyi bir hash fonksiyonunun özelliklerini saymak; tablo boyutunun neden asal seçildiğini göstermek.

Çakışmayı çözmek
Doğrusal yoklama, karesel yoklama, çift hash ve zincirlemeyi küçük bir tablo üzerinde adım adım uygulamak; birincil ve ikincil kümelenmeyi ayırmak.

C ile yazmak
Ekleme, arama ve silme yordamlarını `malloc` / `free`, `NULL` denetimi ve dolu tablo durumuyla birlikte yazmak; bellek sızıntısı bırakmamak.

Silmeyi doğru yapmak
Açık adreslemede silinen yuvanın neden boş bırakılamayacağını ve silindi işaretinin aramayı nasıl koruduğunu göstermek.

Analiz etmek
Yük faktörünün beklenen yoklama sayısına etkisini formüllerle ve ölçümle açıklamak; tabloyu iki katına büyütmenin amortize maliyetini hesaplamak.

14.1 Hash fikri

Bir öğrenci işleri programı düşünelim. Program, öğrenci numarası verilince o öğrencinin kaydını bulmalıdır. Kayıtlar her gün binlerce kez sorgulanır, arada yeni öğrenciler eklenir, mezunlar silinir. Bu iş için hangi veri yapısı seçilmelidir? Kesim bu soruya bilinen yapıların maliyetiyle başlar ve hash tablosunun neden farklı olduğunu gösterir.

14.1.1 Arama maliyetini sabite indirmek

Kayıtlar sırasız bir dizide tutulursa ekleme çok ucuzdur: yeni kayıt dizinin sonuna yazılır. Arama ise pahalıdır. Aranan numara dizinin herhangi bir yerinde olabilir ya da hiç olmayabilir. Bu yüzden en kötü durumda n kaydın hepsine bakılır. Silme de aramayla başladığı için $O(n)$'dir. Dizi numaraya göre sıralı tutulursa arama ikili aramayla $O(lg\ n)$'e iner. Bu kez ekleme ve silme pahalanır: sıra bozulmasın diye yeni kaydın yerinden sonraki bütün kayıtlar bir göz kaydırılır. Dengeli bir arama ağacı üç işlemi de $O(lg\ n)$'de yapar ve kayıtları sıralı gezmeyi de sağlar. Tablo 14.1 dört yapıyı karşılaştırır.

Yapı	Ara	Ekle	Sil	Açıklama
Sırasız dizi	*O(n)*	*O(1)*	*O(n)*	sona eklenir, baştan taranır
Sıralı dizi	*O(lg n)*	*O(n)*	*O(n)*	ikili arama ve kaydırma
Dengeli arama ağacı	*O(lg n)*	*O(lg n)*	*O(lg n)*	sıralı gezinme de verir
Hash tablosu	*O(1)**	*O(1)**	*O(1)**	* ortalama; en kötü *O(n)*

Tablo 14.1: Dört yapıda arama, ekleme ve silme maliyeti. Hash tablosundaki yıldız ortalama durumu gösterir; en kötü durum *O(n)*'dir.

İlk üç yapının ortak bir sınırı vardır: hepsi anahtarları karşılaştırarak arar. Her karşılaştırma iki sonuç verir, “küçük” ya da “büyük”. *n* olası konum arasından birini seçmek için en az *lg n* tane iki sonuçlu karar gerekir. Karşılaştırmaya dayalı bir yapı bu yüzden *O(lg n)*'in altına inemez. Hash tablosu bu sınırı karşılaştırmayı bırakarak aşar. Anahtardan konum doğrudan hesaplanır: konum = *h*(anahtar). Hesap birkaç aritmetik işlemdir ve *n*'ye bağlı değildir.

Kazancın bir bedeli vardır. Hash tablosu öğeleri anahtar sırasıyla tutmaz; konumlar hesabın sonucuna göre dağılır. Bu yüzden bazı sorgular hash tablosunda ucuz değildir. “Bütün kayıtları numara sırasıyla yaz”, “en küçük numarayı bul” ya da “numarası 1000 ile 2000 arasındaki kayıtları listele” bunlardandır. Bu sorgular için bütün tabloyu taramak, gerekirse ayrıca sıralamak gerekir. İkinci bedel en kötü durumdur. Bütün anahtarlar aynı konuma düşerse tablo uzun bir listeye döner ve arama *O(n)* olur. Ortalama durumun sabit kalması, iyi bir hash fonksiyonuna ve tablonun yeterince boş tutulmasına bağlıdır. Notun geri kalanı bu iki koşulu inceler.

14.1.2 Doğrudan adresleme

Hash fikrinin en yalın biçimi, anahtarın kendisini dizi indisi olarak kullanmaktır. Anahtarlar 0 ile 9 arasında birer tam sayıysa on gözlü bir dizi yeterlidir: *k* anahtarlı kayıt `T[k]` gözüne yazılır. Arama için `T[k]` 'ye bakmak yeter; döngü yoktur, karşılaştırma yoktur.

> **Tanım 14.1**
>
> Anahtarların alabileceği bütün değerlerin kümesine **anahtar evreni** denir ve *U* ile gösterilir. *U = {0, 1, ..., |U| − 1}* olduğunda *|U|* gözlü bir `T` dizisi kurup *k* anahtarlı öğeyi `T[k]` gözünde tutmaya **doğrudan adresleme** (*direct addressing*) denir. Dizideki her göz ya bir öğe tutar ya da boştur.

Şekil 14.1 bu düzeni gösterir. Evren on anahtardan oluşur, kullanılan anahtarlar ise yalnız 2, 3, 5 ve 8'dir. Her kayıt kendi numarasının gözündedir; öteki altı göz boştur.



Şekil 14.1: Doğrudan adresleme tablosu. Evren *U = {0, ..., 9}*, kullanılan anahtarlar *K = {2, 3, 5, 8}*. `T[k]` doğrudan *k* anahtarlı kaydı tutar; kesik çizgili gözler boştur.

Sunumdaki C kodu her gözü küçük bir yapıyla gösterir. `var` alanı gözün dolu olup olmadığını, `ad` alanı kaydın verisini tutar.

```c
typedef struct { int var; char ad[16]; } kayit;
kayit T[10];              /* anahtar = indis */

kayit *da_ara(int k) {
  return T[k].var ? &T[k] : NULL;
}
```

Kod kısadır ama her satırı bir karar taşır. İlk satır `kayit` adlı yapı tipini tanımlar. Yapıda iki alan vardır: `int` tipinde `var` ve 16 karakterlik `ad` dizisi. `typedef` sayesinde tip adı her seferinde `struct` yazmadan kullanılır. İkinci satır on gözlü `T` dizisini global olarak tanımlar. C'de global değişkenler program başlarken sıfırla doldurulur. Bu yüzden başlangıçta bütün gözlerin `var` alanı 0'dır, yani tablo boştur.

Ayrıca bir başlatma döngüsü yazmak gerekmez.

`da_ara` fonksiyonu `k` anahtarını alır ve `T[k].var` alanına bakar. Alan sıfırdan farklıysa göz doludur ve fonksiyon o gözün adresini, `&T[k]` 'yi döndürür. Alan sıfırsa kayıt yoktur ve `NULL` döner. Koşul ifadesi (`? :`) bu iki durumu tek satırda yazar. Fonksiyonun kaydın kopyasını değil adresini döndürmesi bilinçli bir seçimdir. Çağıran taraf kaydı bu adres üzerinden okuyabilir ya da değiştirebilir. Bulunamama durumu da `NULL` ile açıkça bildirilir. Ekleme ve silme de aynı biçimde birer satırdır: ekleme `T[k].var = 1` yapıp veriyi yazar, silme `T[k].var = 0` yapar. Üç işlem de *O(1)*'dir.

Test programında 2, 3, 5 ve 8 numaralı gözler Ali, Can, Ece ve Naz adlarıyla doldurduldu. `da_ara` bu dört anahtar için ilgili adları, öteki altı anahtar için `NULL` döndürdü. Aynı makinede `sizeof(kayit)` 20 bayt çıktı (4 baytlık `int` ve 16 baytlık dizi), on gözlü dizi ise 200 bayt yer kapladı.

Kodda bir varsayım gizlidir: `k` her zaman 0 ile 9 arasındadır. Bu aralığın dışındaki bir `k`, dizinin sınırı dışındaki belleği okur. C bu hatayı yakalamaz; sonuç tanımsız davranıştır. Gerçek bir programda fonksiyonun başına `if (k < 0 || k >= 10) return NULL;` gibi bir denetim eklenmelidir.

Doğrudan adreslemenin sorunu evrenin büyüklüğüdür. Öğrenci numarası dokuz haneliyse $|U| = 10^9$'dur. 20 baytlık kayıtlarla bu tablo 20 gigabayt tutar. Üniversitede ise belki 20.000 öğrenci vardır: tablonun yalnız yüz binde ikisi dolu olur, gerisi boş durur. Anahtar bir dizgi olduğunda, örneğin bir değişken adı ya da bir e-posta adresi olduğunda, evren pratikte sonsuzdur. Bu durumda doğrudan adresleme hiç kurulamaz.

Çözüm, anahtarı küçük bir aralığa indiren bir fonksiyondur. Tablonun boyutu kullanılan anahtar sayısına göre seçilir, örneğin $m = 40.000$ göz. Her anahtar bu fonksiyonla 0 ile $m - 1$ arasındaki bir indise çevrilir. Bu fonksiyona hash fonksiyonu denir.

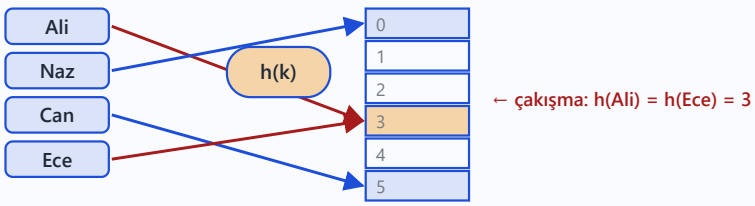
14.1.3 Hash fonksiyonu ve temel terimler

Tanım 14.2

Hash fonksiyonu (*hash function*), anahtar evreninden tablonun indislerine giden bir $h: U \to \{0, 1, \ldots, m-1\}$ fonksiyonudur. m gözlü bir dizi ile bir hash fonksiyonundan oluşan ve k anahtarlı öğeyi $h(k)$ indisli gözde arayan yapıya **hash tablosu** (*hash table*) denir. Tablonun her gözüne **kova** (*bucket*) ya da **yuva** (*slot*) denir. $h(k)$ değeri, k anahtarının **ev adresidir** (*home address*).

"Hash" sözcüğü İngilizcede kıymak, doğramak anlamına gelir. Fonksiyon anahtarı parçalara ayırıp karıştırır ve küçük bir sayı üretir. Türkçe kaynaklarda "karma fonksiyonu" ve "özetleme" karşılıkları da görülür. Bu ders, sözlükteki gibi "hash" sözcüğünü kullanır.

Hash fonksiyonu $|U|$ değeri m indise eşler ve m, $|U|$'dan çok küçüktür. Bu yüzden bazı farklı anahtarlar aynı indise düşmek zorundadır. Buna güvercin yuvası ilkesi denir: on bir güvercin on yuvaya girerse en az bir yuvada iki güvercin bulunur. Şekil 14.2 dört isim anahtarını altı kovaya eşleyen bir fonksiyonu gösterir. Ali ile Ece'nin ev adresi aynıdır, 3.



Şekil 14.2: Anahtarların hash fonksiyonuyla kovalara eşlenmesi. Naz 0'a, Can 5'e gider; Ali ile Ece'nin ikisi de 3'e düşer. Bu duruma çakışma denir.

Tanım 14.3

$k_1 \neq k_2$ iken $h(k_1) = h(k_2)$ olmasına **çakışma** (*collision*) denir. Aynı ev adresine düşen anahtarlara **eş anlamlılar** (*synonyms*) denir. Tabloda n öğe ve m kova varsa $\alpha = n / m$ oranına tablonun **yük faktörü** (*load factor*) denir.

Tablo 14.2 terimleri bir arada verir. "Eş anlamlı" adı dilbilimden gelir: iki sözcük farklı yazılıp aynı anlamı taşıdığı gibi, iki anahtar da farklı olup aynı adrese çıkar. Yük faktörü tablonun ne kadar dolu olduğunu ölçer. $\alpha = 0{,}5$ tablonun yarısının dolu olduğunu söyler. Bazı çakışma çözümlerinde bir kovada birden çok öğe tutulabilir ve α 1'i aşabilir.

Terim	Anlamı
kova / yuva (*bucket, slot*)	tablonun bir konumu, $0 \ldots m-1$
ev adresi	$h(k)$: anahtarın ilk bakılan yeri
çakışma	$k_1 \neq k_2$ ama $h(k_1) = h(k_2)$
eş anlamlılar	aynı ev adresine düşen anahtarlar
yük faktörü α	n / m, tablonun dolu oranı

Tablo 14.2: Hash tablosunun temel terimleri.

$m < |U|$ olduğu sürece çakışma kaçınılmazdır. Bu durumda hash tablosu tasarımı iki soruya iner. Birincisi, iyi bir h nasıl seçilir: anahtarlar kovalara olabildiğince eşit dağılmalı, çakışma az olmalıdır. İkincisi, çakışma olduğunda ne yapılır: aynı adrese düşen ikinci öğe nereye konur ve aranırken nasıl bulunur? Hash fonksiyonları ilk sorunun, açık adresleme ve zincirleme ikinci sorunun cevabıdır.

14.1.4 Sözlük soyut veri tipi

Hash tablosu bir gerçeklemedir. Gerçeklediği soyut veri tipi ise sözlüktür. Soyut veri tipi, bir yapının değerlerini ve işlemlerini, bu işlemlerin nasıl yazıldığından bağımsız olarak tanımlar. Sözlük, gündelik sözlüğe benzer: bir sözcük (anahtar) verilir, karşılığı (değer) bulunur.

> **Tanım 14.4**
>
> **Sözlük** (*dictionary*), anahtar–değer çiftlerinden oluşan ve her anahtarın en çok bir kez bulunduğu bir topluluktur. Temel işlemleri ekleme, anahtarla arama ve anahtarla silmedir.

```c
typedef struct { int anahtar; /* + değer */ } oge;

bool hash_ekle(oge x);      /* varsa günceller */
oge *hash_ara(int anahtar); /* yoksa NULL      */
bool hash_sil(int anahtar); /* yoksa false     */
```

Sunumdaki arayüz üç işlemi C bildirimleriyle verir. `oge` yapısı anahtarı tutar; yorumdaki "+ değer" ifadesi, gerçek bir uygulamada yapıya başka alanların da ekleneceğini söyler. `hash_ekle` bir öğeyi alır ve işlemin başarılı olup olmadığını `bool` olarak döndürür. Anahtar zaten varsa yeni öğe eklenmez; var olanın değeri güncellenir. Böylece "her anahtar en çok bir kez" kuralı korunur. `hash_ara` öğenin adresini döndürür; anahtar yoksa `NULL` verir. `hash_sil` anahtarı bulup silerse `true`, bulamazsa `false` döndürür. `bool`, `true` ve `false` C99 ile gelen `<stdbool.h>` başlığındadır; bu başlık eklenmeden kod derlenmez.

Sözlük çok yerde karşımıza çıkar. Derleyici, kaynak koddaki her değişken adını bir **sembol tablosunda** tutar: ad anahtardır, tipi ve bellekteki yeri değerdir. Horowitz'in ders kitabı hash tablolarını bu örnekle açar. Derleyici her satırda birçok ad arar; aramanın sabit zamanda bitmesi derlemeyi hızlandırır. Web tarayıcılarının önbellekleri adresi anahtar, sayfayı değer olarak tutar. Bir listede tekrar eden öğeleri bulmak için her öğe bir kümeye eklenir; eklenmeden önce kümede aranır. Küme, değeri olmayan bir sözlüktür.

Python'daki `dict`, Java'daki `HashMap` ve C++'taki `unordered_map` birer hash tablosudur. C'nin standart kitaplığında ise hazır bir hash tablosu yoktur. POSIX standardı `hsearch` adlı sınırlı bir yordam verir: aynı anda tek bir tablo kurulabilir ve tablodan öğe silinemez. Bu yüzden C programcısı hash tablosunu kendisi yazmayı bilmelidir. Notun kalan kesimleri tam olarak bunu yapar.

14.2 Hash fonksiyonları

Hash tablosunun hızı, anahtarların kovalara ne kadar eşit dağıldığına bağlıdır. Bütün anahtarları aynı kovaya atan bir fonksiyon da geçerli bir fonksiyondur, ama tabloyu tek bir uzun listeye çevirir. Bu kesim önce iyi bir fonksiyondan beklenenleri sıralar, sonra tam sayı ve dizgi anahtarlar için yaygın yöntemleri hesaplar.

14.2.1 İyi bir hash fonksiyonunun özellikleri

İyi bir hash fonksiyonunda dört özellik aranır.

- **Hızlı olmalıdır.** Fonksiyon her aramada, eklemede ve silmede çalışır. Hesap birkaç aritmetik işlemle bitmelidir. Anahtar bir dizgiyse her karakter en çok bir kez işlenmeli, yani maliyet anahtar uzunluğuyla doğrusal olmalıdır.
- **Düzgün dağıtmalıdır.** Anahtarlar m kovaya olabildiğince eşit sayıda düşmelidir. Kuramsal analizler bunu bir varsayım olarak kabul eder: her anahtar, öteki anahtarlardan bağımsız olarak m kovanın her birine $1/m$ olasılıkla düşer. Bu varsayıma **basit düzgün hash** (*simple uniform hashing*) varsayımı denir.
- **Anahtarın bütün bitlerini kullanmalıdır.** Gerçek veriler genellikle örüntülüdür. Fiyatlar 5 ya da 0 ile biter, bellek adresleri 8'in katıdır, öğrenci numaraları aynı yıl kodlarıyla başlar. Yalnız son basamağa ya da son birkaç bite bakan bir fonksiyon, bu örüntüleri kovalara taşır ve anahtarları birkaç kovada kümeler.
- **Deterministik olmalıdır.** Aynı anahtar her çağrıda aynı kovaya gitmelidir. Aksi hâlde bir kez eklenen öğe sonra bulunamaz. Rastgelelik ancak fonksiyon seçilirken kullanılabilir: program başlarken bir fonksiyon ailesinden rastgele biri seçilir ve program boyunca o kullanılır.

Düzgün dağılım, anahtar kümesi bilinmeden garanti edilemez. Her sabit hash fonksiyonu için, hepsi aynı kovaya düşen bir anahtar kümesi bulunabilir. Yöntemlerin amacı, gerçek hayatta görülen örüntülü verilerde iyi davranmaktır.

14.2.2 Bölme yöntemi

En basit ve en yaygın yöntem, anahtarı tablo boyutuna bölüp kalanı almaktır.

> **Tanım 14.5**
>
> **Bölme yöntemi** (*division method*): $h(k) = k \bmod m$. Kalan her zaman 0 ile $m - 1$ arasında olduğu için sonuç doğrudan bir indistir.

```
unsigned h_bolme(unsigned k, unsigned m) { return k % m; }
```

Fonksiyon tek bir `%` işlemidir. Parametrelerin `unsigned` seçilmesi önemlidir; nedeni bu alt kesimin sonunda verilir. Yöntemin bütün inceliği m'nin seçimindedir. Tablo 14.3 sekiz anahtarı iki farklı tablo boyutuyla dağıtır.

k	10	20	30	40	50	60	17	27
$k \bmod 10$	0	0	0	0	0	0	7	7
$k \bmod 11$	10	9	8	7	6	5	6	5

Tablo 14.3: Aynı anahtarların $m = 10$ ve $m = 11$ ile ev adresleri.

$m = 10$ iken $k \bmod 10$, k'nin son ondalık basamağıdır. Fonksiyon öteki basamakları hiç görmez. 10, 20, ..., 60 anahtarlarının hepsi 0 ile biter ve hepsi 0. kovaya düşer; 17 ile 27 de 7. kovada çakışır. Sekiz anahtar yalnız iki kovaya dağılmıştır. $m = 11$ iken aynı sekiz anahtar altı farklı kovaya gider. Hâlâ iki çakışma vardır (50 ile 17 kova 6'da, 60 ile 27 kova 5'te), ama 10'un katları artık art arda kovalara yayılır. Bunun nedeni, 11'in 10 ile ortak böleninin olmamasıdır: $10j \bmod 11$, $j = 1, \ldots, 10$ için on farklı değer alır.

Aynı sorun 2'nin kuvvetlerinde daha da belirgindir. $m = 2^p$ ise $k \bmod 2^p$, k'nin ikili gösterimindeki en düşük p bittir. Bilgisayardaki veriler ikili sayılarla tutulduğu için son bitlerdeki örüntüler çok yaygındır. 8 baytlık yapıların adresleri her zaman 8'in katıdır, yani son üç biti 000'dır. Bu adresler $m = 16$'lık bir tabloda yalnız 0 ve 8 numaralı kovalara düşer. Bu yüzden şu kural önerilir (CLRS): m olarak, 2'nin bir kuvvetine yakın olmayan bir asal sayı seçilmelidir. Örneğin yaklaşık 2000 anahtar için 701 iyi bir seçimdir: asaldır ve 512 ile 1024'ten uzaktır. Bu tabloda kova başına ortalama üç anahtar düşer.

Bölme yönteminin C'de bir tuzağı daha vardır. C'de `%` işlecinin sonucu, bölünenle aynı işareti taşır: `-7 % 11` ifadesi 4 değil −7'dir. Anahtar `int` tipindeyse ve negatif olabiliyorsa `k % m` negatif bir indis verir. Dizinin başından önceki bellek okunur ya da üzerine yazılır. Çare iki yoldan biridir. Ya anahtar baştan `unsigned` tutulur (sunumdaki `h_bolme` bunu yapar) ya da kalan pozitif yapılır:

```
unsigned h_guvenli(int k, int m) {
  int r = k % m;            /* -m < r < m */
  if (r < 0) r += m;        /* ((k % m) + m) % m ile aynı */
  return (unsigned)r;
}
```

Test programında `-7 % 11` gerçekten −7 verdi; `h_guvenli(-7, 11)` ise 4, `h_guvenli(-22, 11)` 0 döndürdü. Kalan −*m* ile *m* arasında olduğu için bir kez *m* eklemek onu 0 ile *m* − *1* arasına taşır. Sunumdaki `((k % m) + m) % m` ifadesi aynı işi dallanmadan yapar.

> **Sık yapılan hata**
>
> Matematikteki "mod" işlemiyle C'deki `%` işlecini aynı sanmak. Matematikte *−7 mod 11 = 4*'tür; C'de `-7 % 11 == -7` 'dir. Bu hata derleyici uyarısı vermez, yalnız negatif anahtar geldiğinde ortaya çıkar ve dizinin dışına yazar.

Maliyet. Bölme yöntemi tek bir tam sayı bölmesidir: *O(1)*. Bölme, toplama ve çarpmaya göre işlemcide daha yavaş bir işlemdir; yine de anahtar başına bir kez yapıldığı için tablonun maliyetini değiştirmez.

14.2.3 Çarpma yöntemi

Bölme yöntemi iyi bir *m* ister. Çarpma yöntemi bu bağımlılığı azaltır: karıştırma işini *m* değil, sabit bir çarpan yapar.

> **Tanım 14.6**
>
> **Çarpma yöntemi** (*multiplication method*): $0 < A < 1$ bir sabit olmak üzere $h(k) = \lfloor m \cdot (k \cdot A \bmod 1) \rfloor$. Burada $x \bmod 1$, x'in kesir kısmıdır: $x - \lfloor x \rfloor$.

Hesap iki adımdır. Önce *k*, *A* ile çarpılır ve sonucun tam kısmı atılır. Geriye 0 ile 1 arasında bir kesir kalır. Sonra bu kesir *m* ile çarpılır ve aşağı yuvarlanır; sonuç 0 ile *m* − *1* arasında bir tam sayıdır. Kesir kısmı, *k*'nin bütün basamaklarından etkilenir. Bu yüzden *m*'nin seçimi kritik değildir ve $m = 2^p$ rahatlıkla seçilebilir. Knuth, *A* için altın oranın kesir kısmını önerir: $A \approx (\sqrt{5} - 1)/2 = 0{,}6180339887\ldots$ Bu değer, ardışık anahtarları kesir aralığına çok düzgün yayar.

Örnek (CLRS). $k = 123456$, $p = 14$, yani $m = 2^{14} = 16384$ alalım. Kayan noktalı hesap şöyledir:

> $k \cdot A = 123456 \cdot 0{,}6180339887 = 76300{,}0041151\ldots$
> $k \cdot A \bmod 1 = 0{,}0041151\ldots$
> $m \cdot 0{,}0041151\ldots = 16384 \cdot 0{,}0041151\ldots = 67{,}42\ldots$
> $h(k) = \lfloor 67{,}42\ldots \rfloor = 67$

Bilgisayarda kayan noktalı sayılar yuvarlama hatası taşır ve yavaştır. Bu yüzden yöntem genellikle tam sayılarla yazılır. Sözcük uzunluğu $w = 32$ bit olsun ve $s = \lfloor A \cdot 2^{32} \rfloor = 2654435769$ alınsın. $k \cdot s$ çarpımının alt 32 biti, $k \cdot A$'nın kesir kısmının 2^{32} katıdır. Bu 32 bitin en üstteki *p* biti, kesir kısmının 2^p katının tam kısmıdır. Tam olarak aranan değer budur.

```c
/* w = 32 bit, m = 2^p; s = A * 2^32 */
uint32_t h_carpma(uint32_t k, int p) {
  const uint32_t s = 2654435769u;
  return (uint32_t)(k * s) >> (32 - p);
}
```

Kod üç satırdır. `uint32_t`, `<stdint.h>` başlığındaki tam 32 bitlik işaretsiz tam sayı tipidir. `k * s` çarpımı iki `uint32_t` arasında yapılır ve sonuç 32 bite sığmazsa taşar. İşaretsiz tam sayılarda taşma C'de tanımlıdır: sonuç 2^{32}'ye göre kalandır, yani üstteki bitler atılır. Atılan kısım çarpımın tam kısmına, kalan 32 bit kesir kısmına karşılık gelir. Son olarak `>> (32 - p)` kaydırması alt $32 - p$ biti atar ve üstteki *p* biti bırakır.

Örnek üzerinde: $123456 \cdot 2654435769 = 327.706.022.297.664$'tür. Bu sayının 2^{32}'ye göre kalanı 17.612.864'tür. 18 bit sağa kaydırmak $2^{18} = 262.144$'e bölüp tam kısmı almaktır: $17.612.864 / 262.144 = 67{,}19\ldots$, sonuç 67. Test programı hem tam sayı sürümünden hem kayan noktalı sürümden 67 aldı. İki sürümün kesirleri küçük farklar gösterir (67,42 ve 67,19), çünkü *s*, $A \cdot 2^{32}$'nin yuvarlanmış hâlidir; tam kısımları ise aynıdır.

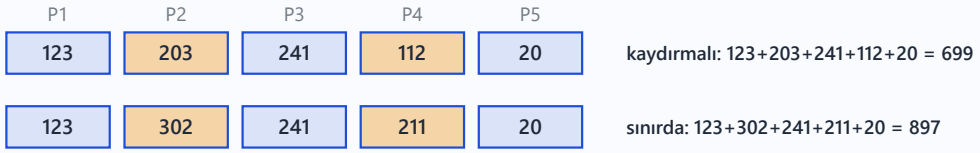
Kodun bir sınırı vardır: `p` 1 ile 32 arasında olmalıdır. `p = 0` verilirse kaydırma miktarı 32 olur ve 32 bitlik bir sayıyı 32 bit kaydırmak C'de tanımsız davranıştır. *m = 1*'lik bir tablo zaten anlamsız olduğu için bu durum pratikte sorun çıkarmaz, ama fonksiyonun sözleşmesine yazılmalıdır. **Maliyet:** bir çarpma ve bir kaydırma, *O(1)*. Çarpma bölmeden hızlı olduğu için yöntem hız açısından da avantajlıdır.

14.2.4 Katlama ve orta-kare

Anahtar bir işlemci sözcüğüne sığmayacak kadar uzunsa, örneğin 14 haneli bir kimlik numarasıysa, önce kısaltılmalıdır. Katlama yöntemi bunu yapar.

> **Tanım 14.7**
>
> **Katlama** (*folding*): anahtar soldan sağa eşit uzunlukta parçalara bölünür ve parçalar toplanır. Parçalar olduğu gibi toplanırsa **kaydırmalı katlama** (*shift folding*), çift sıradaki parçalar ters çevrilip toplanırsa **sınırda katlama** (*folding at the boundaries*) adını alır.

Horowitz'in örneğinde anahtar $x = 12320324111220$'dir. Üç haneli parçalar $P_1 = 123$, $P_2 = 203$, $P_3 = 241$, $P_4 = 112$ ve $P_5 = 20$'dir; son parça iki hanelidir. Şekil 14.3 iki toplamı gösterir.



Şekil 14.3: 12320324111220 anahtarının katlanması. Üst satırda kaydırmalı katlama: parçalar olduğu gibi toplanır, 699. Alt satırda sınırda katlama: ikinci ve dördüncü parça ters çevrilir (203 → 302, 112 → 211), 897.

Kaydırmalı katlamada toplam $123 + 203 + 241 + 112 + 20 = 699$'dur. Sınırda katlamada anahtar bir kâğıt şerit gibi parça sınırlarından zikzak katlanır. Katlanınca her ikinci parça ters yüzüyle üste gelir ve rakamları tersten okunur. Toplam $123 + 302 + 241 + 211 + 20 = 897$'dir. Sonuç tablo boyutundan büyükse ayrıca *mod m* alınır; örneğin $m = 11$ için iki değer de 6 verir. Aşağıdaki fonksiyon iki türü bir bayrakla hesaplar. Anahtar bir rakam dizgisi olarak verilir, çünkü 14 haneli bir sayı 32 bitlik bir tam sayıya sığmaz.

```c
/* x: rakam dizgisi; parçalar r haneli, soldan sağa */
unsigned katlama(const char *x, int r, bool sinirda) {
  unsigned toplam = 0;
  int n = (int)strlen(x);
  for (int bas = 0, sira = 0; bas < n; bas += r, sira++) {
    int son = bas + r < n ? bas + r : n;   /* parça: [bas, son) */
    unsigned parca = 0;
    if (sinirda && sira % 2 == 1)          /* çift sıradaki parça ters */
      for (int j = son - 1; j >= bas; j--) parca = parca * 10 + (unsigned)(x[j] - '0');
    else
      for (int j = bas; j < son; j++) parca = parca * 10 + (unsigned)(x[j] - '0');
    toplam += parca;
  }
  return toplam;
}
```

Dış döngü `bas` değişkenini her turda `r` artırır ve parçalarda sırayla gezinir; `sira` parçanın 0 tabanlı sırasıdır. `son`, parçanın bittiği yerin bir sonrasıdır; son parça kısa kalabileceği için `bas + r` ile `n`'nin küçüğü alınır. İç döngüler parçanın rakamlarını sayıya çevirir: her rakam için `parca = parca * 10 + rakam`. `x[j] - '0'` ifadesi bir rakam karakterini sayı değerine çevirir; `'7' - '0'` 7'dir. Sınırda katlamada `sira` değeri tek olan parçalar (ikinci, dördüncü, ...) sağdan sola okunur. Test programında fonksiyon 699 ve 897 değerlerini verdi. Anahtarın her rakamı bir kez işlendiği için maliyet anahtar uzunluğuyla doğrusaldır.

Orta-kare (*mid-square*) yöntemi anahtarın karesini alır ve sonucun ortasındaki r biti indis olarak kullanır; tablo boyutu $m = 2^r$ seçilir. Neden orta? Bir sayının karesinin en düşük bitleri yalnız sayının en düşük bitlerine bağlıdır; en yüksek bitleri de büyük ölçüde sayının büyüklüğüne bağlıdır. Ortadaki bitler ise çarpımda k'nin hemen bütün bitlerinin katkısıyla oluşur. Ondalık bir örnekle fikir kolay görülür: $k = 1234$ için $k^2 = 1522756$'dır. Yedi basamağın ortadaki üçü 227'dir; $m = 1000$'lik bir tabloda ev adresi 227 olur. 1234'ün herhangi bir basamağı değişse bu üç basamak da değişir.

14.2.5 Dizgi anahtarlar: polinom hash'i

Pratikte anahtarların çoğu dizgidir: değişken adları, kullanıcı adları, sözcükler. Dizgiyi bir tam sayıya çevirmenin akla gelen ilk yolu, karakter kodlarını toplamaktır.

```
unsigned toplama_hash(const char *s, unsigned m) {  /* kötü örnek */
  unsigned h = 0;
  for (; *s != '\0'; s++)
    h += (unsigned char)*s;
  return h % m;
}
```

Bu fonksiyon kötüdür. Toplama sırayı unutur: “abc” ve “cba” aynı harflerden oluşur, toplamları *97 + 98 + 99 = 294* olur ve ikisi de aynı kovaya düşer. Bir dizginin harflerinin yer değiştirmesiyle elde edilen dizgilere anagram denir; toplama hash'i bütün anagramları çakıştırır. İkinci sorun aralığın darlığıdır. 8 harfli küçük harf dizgilerinin toplamı en çok *8 · 122 = 976*'dır. Tablo 10.000 kovalı olsa bile anahtarlar ilk bin kovaya sıkışır.

Çözüm, her karakterin konumunu da hesaba katmaktır. *n* karakterli $c_0\, c_1 \ldots c_{n-1}$ dizgisi, 31 tabanında bir sayı gibi okunur:

> $$h = c_0 \cdot 31^{n-1} + c_1 \cdot 31^{n-2} + \ldots + c_{n-2} \cdot 31 + c_{n-1}$$

Bu ifade bir polinomdur. Doğrudan hesaplansaydı her terim için ayrı bir üs alınırdı. Horner kuralı polinomu iç içe yazar: $h = (\ldots((c_0 \cdot 31 + c_1) \cdot 31 + c_2) \cdot 31 + \ldots) + c_{n-1}$. Böylece her karakterde bir çarpma ve bir toplama yeter. Sunumdaki iki fonksiyon bu kuralı kullanır.

```
unsigned dizgi_hash(const char *s, unsigned m) {
  unsigned h = 0;
  for (; *s != '\0'; s++)
    h = (h * 31 + (unsigned char)*s) % m;
  return h;
}
```

`dizgi_hash`, `h`'yi 0'dan başlatır. Döngü, `s` işaretçisini dizginin karakterleri üzerinde ilerletir ve `'\0'` sonlandırıcısında durur. Her adımda `h`, 31 ile çarpılıp yeni karakterin kodu eklenir ve sonucun *mod m*'si alınır. `(unsigned char)*s` dönüşümü önemlidir: `char` birçok derleyicide işaretlidir ve “ç”, “ğ” gibi ASCII dışı baytlar negatif değer taşır. Dönüşüm, karakteri 0–255 arasında bir sayı olarak okur.

Her adımda mod almanın iki yararı vardır. Birincisi, `h` her zaman *m*'den küçük kalır ve çarpım taşmaz (yeter ki *31 · m + 255* bir `unsigned`'a sığsın). İkincisi, sonuç değişmez: modüler aritmetikte *(a · 31 + c) mod m*, *((a mod m) · 31 + c) mod m*'ye eşittir. Ara adımlarda mod almak, en sonda bir kez almakla aynı sonucu verir.

```
unsigned long djb2(const char *s) {  /* Bernstein */
  unsigned long h = 5381;
  int c;
  while ((c = (unsigned char)*s++) != 0)
    h = h * 33 + c;
  return h;       /* sonra: h % m */
}
```

İkinci fonksiyon, Dan Bernstein'ın yaygın kullanılan `djb2` hash'idir. 5381 başlangıç değeriyle başlar ve her karakterde *h · 33 + c* hesaplar. Ara adımlarda mod almaz; `unsigned long` taşınca sonuç kendiliğinden 2^w'ye göre kalan olur (*w*, tipin bit sayısıdır). İşaretsiz taşma C'de tanımlı olduğu için bu güvenlidir. Tablo indisi için çağıran taraf sonucun *mod m*'sini alır. Döngü koşulundaki `(c = (unsigned char)*s++) != 0` ifadesi üç iş yapar: karakteri okur, işaretçiyi ilerletir ve karakterin sıfır olup olmadığına bakar. Windows'ta `unsigned long` 32 bit, Linux'ta 64 bittir; bu yüzden `djb2` aynı dizgi için iki sistemde farklı sayılar verebilir. Test makinesinde (Windows) `djb2("kedi")` 2090432258 çıktı.

Tablo 14.4 üç fonksiyonu anagramlar üzerinde karşılaştırır. Polinom hash'i “abc” ile “cba”yı ayırır, çünkü “a” birinde 31^2 ile, ötekinde 1 ile çarpılır.

Dizgi	Toplama	`toplama_hash`	`dizgi_hash`
"abc"	294	8	5
"cba"	294	8	0
"ab"	195	8	3
"ba"	195	8	0

Tablo 14.4: Anagramlar için toplama ve polinom hash'i ($m = 11$). Değerler test programından alındı.

31 sayısının seçimi rastgele değildir. Taban tek sayı olmalıdır: çift bir taban, taşma sırasında en düşük bitlere sıfır doldurur ve ilk karakterlerin etkisi kaybolur. Taban ayrıca alfabedeki harf sayısından büyük olmalıdır ki farklı dizgiler aynı "sayıyı" vermesin. Java'nın `String.hashCode` yöntemi de 31 kullanır. 31 ile çarpmanın ayrıca ucuz bir karşılığı vardır: $31 \cdot h = 32 \cdot h - h$, yani bir kaydırma ve bir çıkarma.

Maliyet. Üç dizgi fonksiyonu da her karakteri bir kez işler. L karakterli bir anahtar için zaman $O(L)$, bellek $O(1)$'dir. Hash tablosunun "$O(1)$" maliyeti, dizgi anahtarlarda aslında "$O(L)$" demektir. Anahtar uzunluğu tablo boyutundan bağımsız olduğu için bu sabit sayılır.

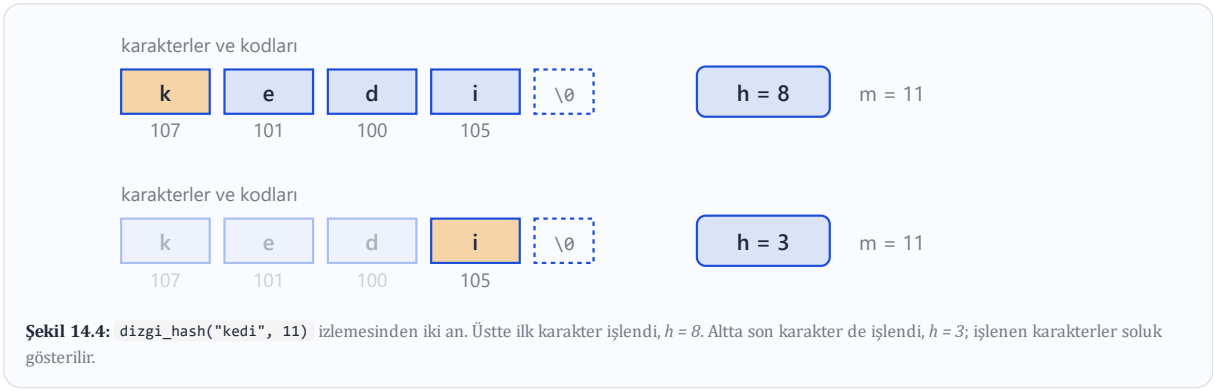
14.2.6 Kod izleme: `dizgi_hash("kedi", 11)`

"kedi" dizgisinin karakter kodları ASCII tablosunda k = 107, e = 101, d = 100 ve i = 105'tir. Tablo 11 kovalıdır. Tablo 14.5 döngünün her turunu gösterir.

Tur	`*s`	Kod	Önceki `h`	`h * 31 + kod`	Yeni `h`
1	`'k'`	107	0	$0 \cdot 31 + 107 = 107$	$107 \bmod 11 = 8$
2	`'e'`	101	8	$8 \cdot 31 + 101 = 349$	$349 \bmod 11 = 8$
3	`'d'`	100	8	$8 \cdot 31 + 100 = 348$	$348 \bmod 11 = 7$
4	`'i'`	105	7	$7 \cdot 31 + 105 = 322$	$322 \bmod 11 = 3$
5	`'\0'`	—	3	döngü koşulu yanlış; `return 3`	

Tablo 14.5: `dizgi_hash("kedi", 11)` izlemesi. Her turda `h = (h * 31 + *s) % 11`.

İlk turda `h` sıfır olduğu için çarpımın etkisi yoktur ve `h`, 107'nin 11'e göre kalanı olur: $107 = 9 \cdot 11 + 8$. İkinci turda eski değer 31 ile çarpılır, "e"nin kodu eklenir: $349 = 31 \cdot 11 + 8$. Üçüncü ve dördüncü turlar 7 ve 3 verir. Beşinci denetimde `*s` sonlandırıcı karakterdir, döngü biter ve fonksiyon 3 döndürür. "kedi" anahtarının ev adresi 3'tür.



Şekil 14.4: `dizgi_hash("kedi", 11)` izlemesinden iki an. Üstte ilk karakter işlendi, $h = 8$. Altta son karakter de işlendi, $h = 3$; işlenen karakterler soluk gösterilir.

Sonucu bir de ara adımlarda mod almadan denetleyelim. Tam polinom $107 \cdot 31^3 + 101 \cdot 31^2 + 100 \cdot 31 + 105 = 3.187.637 + 97.061 + 3.100 + 105 = 3.287.903$'tür ve $3.287.903 \bmod 11 = 3$'tür. İki hesap aynı sonucu verir; ara adımlarda mod almanın sonucu değiştirmediği burada sayılarla da görülür. Dört karakterli dizgi için döngü dört kez döner ve beşinci denetimde biter: maliyet dizgi uzunluğuyla doğrusaldır.

14.3 Açık adresleme

İyi bir hash fonksiyonu çakışmayı azaltır ama ortadan kaldıramaz. Bu kesim önce çakışmanın ne kadar erken başladığını gösterir, sonra çakışan öğeyi tablonun içinde başka bir yuvaya yerleştiren yöntemleri inceler.

14.3.1 Çakışma kaçınılmazdır

Tablo büyük ve anahtar sayısı azsa çakışmanın nadir olacağı düşünülebilir. Olasılık hesabı bu sezgiyi yanlışlar. Ünlü **doğum günü paradoksu** şunu söyler: 23 kişilik bir sınıfta iki kişinin doğum gününün aynı gün olma olasılığı yaklaşık %50,7'dir. Hesap tümleyenden yapılır. 23 kişinin hepsinin farklı günlerde doğmuş olma olasılığı

> $(365/365) \cdot (364/365) \cdot (363/365) \cdot \ldots \cdot (343/365) \approx 0{,}4927$

olur. Birinci kişi herhangi bir günde doğabilir; ikinci kişinin ondan farklı günde doğma olasılığı 364/365'tir; üçüncü kişi ilk ikisinden farklı olmalıdır, ve bu böyle sürer. En az bir çakışma olasılığı $1 - 0{,}4927 \approx 0{,}507$'dir.

Bunu hash tablosuna çevirelim: $m = 365$ yuvalı bir tabloya, her biri rastgele bir yuvaya düşen 23 anahtar eklenir. Tablonun yalnız %6'sı doludur, ama en az bir çakışma olasılığı yarıdan fazladır. Anahtar sayısı 57'ye çıkınca bu olasılık %99'u geçer. Genel kural şudur: m yuvalı bir tabloda ilk çakışma, yaklaşık $\sqrt{m}$ anahtardan sonra beklenir. Bir milyon yuvalı tabloda bu sayı bin kadardır. Çakışma bu yüzden istisnai bir durum olarak değil, olağan bir durum olarak ele alınmalıdır.

Çakışmanın iki temel çözüm yolu vardır. Şekil 14.5 ikisini aynı anahtarlar üzerinde gösterir.



Şekil 14.5: Çakışmanın iki çözümü. Solda açık adresleme: 11'in ev adresi 0 doludur; 1 ve 2 de dolu olduğu için 11, ilk boş yuva olan 3'e yazılır. Sağda zincirleme: her kova bir listenin başıdır; 11 ile 22, 0. kovanın listesinde birlikte durur.

Tanım 14.8

Açık adresleme (*open addressing*): bütün öğeler tablonun yuvalarında tutulur, her yuvada en çok bir öğe bulunur. Ev adresi doluysa belirli bir kurala göre başka yuvalara bakılır. Bir yuvanın boş olup olmadığına bakmaya **yoklama** (*probe*), bakılan yuvaların sırasına **yoklama dizisi** (*probe sequence*) denir. Yoklama dizisi $h(k, i)$ ile gösterilir: i. denemede bakılan yuva ($i = 0, 1, \ldots, m - 1$).

Açık adreslemede her öğe bir yuva kapladığı için $n \le m$, yani $\alpha \le 1$'dir. Tablo dolunca yeni öğe eklenemez. **Zincirlemede** (*chaining*) ise her kova bir bağlı listenin başıdır; aynı kovaya düşen öğeler o listeye eklenir. Liste istendiği kadar uzayabildiği için $\alpha > 1$ olabilir. Zincirleme, kendi kesiminde ayrıntılı ele alınır. Bu kesimin kalanı açık adreslemenin üç yoklama kuralını inceler: doğrusal yoklama, karesel yoklama ve çift hash. Örneklerin hepsi aynı tabloyu kullanır: $m = 11$, $h(k) = k \bmod 11$ ve sırayla eklenen 22, 1, 13, 11, 24, 33 anahtarları. Bu anahtarların ev adresleri 0, 1, 2, 0, 2, 0'dır: 11 ve 33, 22 ile; 24 de 13 ile çakışır.

14.3.2 Doğrusal yoklama

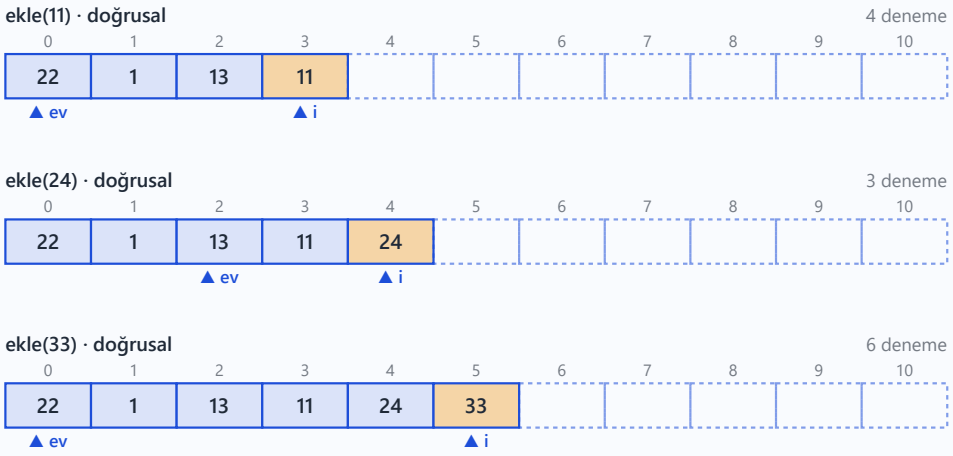
Tanım 14.9

Doğrusal yoklama (*linear probing*): $h(k, i) = (h(k) + i) \bmod m$. Ev adresi doluysa bir sonraki yuvaya, o da doluysa onun sonrakine bakılır. Tablonun sonuna gelinince başa dönülür.

Kural, sinemada yer arayan birine benzer: numaralı koltuk doluysa yanındakine oturulur. Altı anahtarın eklenişi şöyle ilerler.

1. 22'nin ev adresi $22 \bmod 11 = 0$'dır. Yuva 0 boştur; 22 oraya yazılır. Bir yoklama.
2. 1'in ev adresi 1, 13'ün ev adresi 2'dir. İkisi de boş yuvaya düşer; birer yoklama.

3. 11'in ev adresi 0'dır ve 22 ile doludur. Sıradaki yuva 1 (dolu, 1), sonra yuva 2 (dolu, 13), sonra yuva 3'e bakılır. Yuva 3 boştur. Yoklama dizisi 0, 1, 2, 3; dört yoklama.
4. 24'ün ev adresi 2'dir ve 13 ile doludur. Yuva 3'te artık 11 vardır. Yuva 4 boştur. Yoklama dizisi 2, 3, 4; üç yoklama.
5. 33'ün ev adresi 0'dır. Yuva 0'dan 4'e kadar hepsi doludur; yuva 5 boştur. Yoklama dizisi 0, 1, 2, 3, 4, 5; altı yoklama.



Şekil 14.6: Doğrusal yoklamayla ekleme. Üstte 11, ev adresinden üç yuva ötede, 3'e yerleşti. Ortada 24, ev adresi 2'den iki yuva ötede, 4'e yerleşti. Altta 33, ev adresi 0'dan beş yuva ötede, 5'e yerleşti. Yuva 0–5 kesintisiz dolu bir blok oluşturur.

Son durumda tablonun ilk altı yuvası dolu bir blok oluşturur (Şekil 14.6). Ev adresi bu bloğun içindeki herhangi bir yuvaya düşen yeni anahtar, bloğun sonuna kadar yürür ve oraya yerleşir; blok bir yuva daha büyür. Blok büyüdükçe ona düşme olasılığı da artar: 6 yuvalık bir blok, 11 olası ev adresinin altısını kapsar. Büyük bloklar daha da büyür.

Tanım 14.10

Doğrusal yoklamada dolu yuvaların art arda uzun bloklar oluşturmasına ve bu blokların kendilerini büyütme eğilimine **birincil kümelenme** (*primary clustering*) denir.

Birincil kümelenme yalnız eş anlamlıları etkilemez. Örnekte 24'ün ev adresi 2'dir ve 22 ya da 11 ile hiçbir ilgisi yoktur. Yine de 11 yüzünden bir yuva fazla yürümüştür. Farklı ev adresli anahtarlar aynı bloğu paylaşır ve birbirini yavaşlatır. Tablo dolmaya yaklaştıkça bloklar birleşir ve ortalama yoklama sayısı hızla büyür. Bu artışın sayısal değeri yük faktörü kesiminde verilir.

14.3.3 C gerçeklemesi: yuva durumu ve ekleme

Açık adreslemede bir yuvanın yalnız “dolu” ya da “boş” olması yetmez. Silme işleminin doğru çalışması için üçüncü bir durum gerekir. Bu durumun neden gerektiği silme kesiminde ayrıntılı gösterilir; burada yalnız tanımı verilir.

```
#define M 11
typedef enum { BOS, DOLU, SILINDI } durum_t;
typedef struct {
  int anahtar;          /* anahtar >= 0 varsayılır */
  durum_t durum;
} yuva;
yuva tablo[M];          /* global: hepsi 0 = BOS */
```

`#define M 11` tablo boyutunu bir makro sabit olarak tanımlar. Önişlemci, kodun her yerindeki `M`'yi derlemeden önce 11 ile değiştirir. `enum` üç sabit tanımlar: `BOS` = 0, `DOLU` = 1 ve `SILINDI` = 2. Değerler belirtilmezse C, ilk sabite 0 verir ve her birini bir artırır. `durum_t` bu sayıların tip adıdır; kodda `0` yerine `BOS` yazmak okunurluğu artırır.

`yuva` yapısının iki alanı vardır: anahtar ve durum. Yorumdaki varsayım (anahtar ≥ 0) bölme yönteminin negatif kalan sorununu bu kod için ortadan kaldırır. `tablo` dizisi globaldir ve program başlarken sıfırla dolar. `BOS` sıfıra eşit olduğu için bütün yuvalar başlangıçta boştur. `enum` sabitlerinin sırası bu yüzden önemlidir: `BOS` ilk sırada yazılmasaydı tablo başlangıçta boş sayılmazdı.

Bellekte `tablo`, 11 yapının art arda dizildiği tek bir bloktur. Test makinesinde her yuva 8 bayttır (4 baytlık `int` ve 4 baytlık `enum`), tablo 88 bayt yer kaplar. İşaretçi yoktur, `malloc` yoktur. Bu, açık adreslemenin önemli bir üstünlüğüdür ve tasarım karşılaştırmasında tekrar ele alınır.

```
int ekle(int k) {
  int i = k % M;                /* ev adresi */
  for (int d = 0; d < M; d++, i = (i + 1) % M) {
    if (tablo[i].durum != DOLU) {  /* BOS/SILINDI */
      tablo[i].anahtar = k;
      tablo[i].durum = DOLU;
      return i;
    }
  }
  return -1;                    /* tablo dolu */
}
```

`ekle` fonksiyonu satır satır şöyle çalışır.

- **Satır 2:** `i`, anahtarın ev adresidir. İlk yoklama burada yapılacaktır.
- **Satır 3:** `for` döngüsü iki değişken yönetir. `d` deneme sayacıdır ve 0'dan $M-1$'e kadar gider. `i` bakılan yuvadır ve her turun sonunda bir artırılıp *mod M* alınır; 10'dan sonra 0'a döner. Virgül işleci iki güncellemeyi tek ifadede yapar.
- **Satır 4–8:** Yuva `DOLU` değilse, yani `BOS` ya da `SILINDI` ise, anahtar oraya yazılır, durum `DOLU` yapılır ve yuvanın indisi döndürülür.
- **Satır 10:** Döngü M denemenin hepsinde dolu yuva bulduysa tablo doludur; fonksiyon −1 döndürür.

Deneme sayısının M ile sınırlanması önemlidir. Doğrusal yoklamanın dizisi M adımda bütün yuvaları birer kez gezer. Sınır olmasaydı dolu bir tabloda döngü sonsuza kadar dönerdi. Test programında 0, 11, 22, …, 110 anahtarlarıyla doldurulan tabloya 5 eklenmek istendi; `ekle` 11 denemeden sonra −1 döndürdü.

Fonksiyonun bilinen bir eksiği vardır: anahtarın tabloda zaten olup olmadığına bakmaz. Aynı anahtar ikinci kez eklenirse yoklama dizisindeki ilk boş yuvaya ikinci bir kopya yazılır. Sözlüğün “her anahtar bir kez” kuralı bu kodda çağıranın sorumluluğundadır. `SILINDI` yuvalar işin içine girince sorun daha da incelir; çözümü silme kesimindeki `ekle_tekil` fonksiyonudur.

Maliyet. Ekleme, yoklama sayısı kadar sürer. En iyi durumda ev adresi boştur: $\Theta(1)$. En kötü durumda bütün tabloda gezinilir: $O(m)$. Ortalama durum yük faktörüne bağlıdır; α sabit tutulduğu sürece $O(1)$'dir. Ek bellek kullanılmaz.

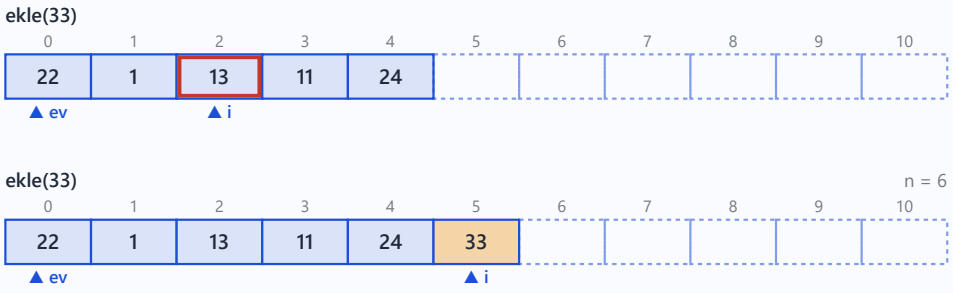
14.3.4 Kod izleme: `ekle(33)`

Tabloda 22, 1, 13, 11 ve 24 vardır; yuva 0–4 dolu bir bloktur. Tablo 14.6 `ekle(33)` çağrısını izler.

`d`	`i`	`tablo[i]`	**Karar**
—	0	—	ev adresi $33 \bmod 11 = 0$
0	0	22, `DOLU`	sonraki: $i = (0+1) \bmod 11 = 1$
1	1	1, `DOLU`	sonraki: $i = 2$
2	2	13, `DOLU`	sonraki: $i = 3$
3	3	11, `DOLU`	sonraki: $i = 4$
4	4	24, `DOLU`	sonraki: $i = 5$
5	5	`BOS`	`tablo[5] = {33, DOLU}; return 5`

Tablo 14.6: `ekle(33)` izlemesi, doğrusal yoklama, $M = 11$.

Döngü altı tur döner; her turda yapılan iş aynıdır: durum alanına bakmak ve `i`'yi ilerletmek. Altıncı yoklamada boş yuva bulunur. 33, ev adresinden beş yuva uzağa düşmüştür. Bunun nedeni 33'ün kendi eş anlamlıları değil, bloğun tamamıdır. 0–4 yuvalarındaki beş anahtardan yalnız ikisinin (22 ve 11) ev adresi 33'ünkiyle aynıdır.



Şekil 14.7: `ekle(33)` izlemesinden iki an. Üstte 2. deneme: yuva 2 dolu, `i` ilerliyor; "ev" işareti ev adresini, "i" bakılan yuvayı gösterir. Altta 33, yuva 5'e yazılmış.

14.3.5 Arama ve kod izleme

Arama, eklemenin izlediği yoklama dizisini aynen izler. Anahtar tabloda varsa bu dizinin bir yerindedir. Peki aramanın ne zaman "yok" diyeceği nasıl bilinir? Ekleme, yoklama dizisinde rastladığı ilk boş yuvaya yazar. Öyleyse aranan anahtar tabloda olsaydı, dizideki ilk `BOS` yuvadan önce bulunurdu. `BOS` bir yuvaya rastlayan arama durabilir.

```
int ara(int k) {
  int i = k % M;
  for (int d = 0; d < M; d++, i = (i + 1) % M) {
    if (tablo[i].durum == BOS) return -1;
    if (tablo[i].durum == DOLU &&
        tablo[i].anahtar == k) return i;
  }
  return -1;
}
bool sil(int k) {
  int i = ara(k);
  if (i < 0) return false;
  tablo[i].durum = SILINDI;   /* silindi işareti */
  return true;
}
```

Bu kod parçasında iki fonksiyon vardır; burada `ara` anlatılır, `sil` silme kesiminin konusudur. `ara`, `ekle` ile aynı döngü iskeletini kullanır. Her yuvada iki soru sorar. Yuva `BOS` ise arama biter ve −1 döner. Yuva `DOLU` ve anahtarı `k` ise aranan bulunmuştur; indis döner. Öteki durumlarda, yani yuva başka bir anahtarla dolu ya da `SILINDI` ise, arama bir sonraki yuvaya geçer. İkinci `if`'teki `tablo[i].durum == DOLU` koşulu gereklidir. `SILINDI` bir yuvanın `anahtar` alanında eski anahtar hâlâ durur; bu koşul olmasaydı silinmiş bir anahtar bulunmuş sayılırdı. Döngü M denemede bitip hiç `BOS` yuvaya rastlamazsa tablo doludur ve anahtar yoktur; fonksiyon yine −1 döner.

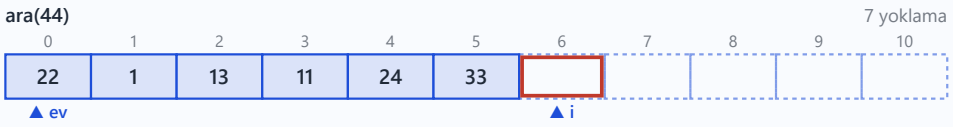
Tablo 14.7 iki aramayı izler. Tabloda 22, 1, 13, 11, 24 ve 33 vardır (yuva 0–5).

Çağrı	Bakılan yuvalar	Sonuç	Yoklama
`ara(24)`	2 (13) → 3 (11) → 4 (24)	bulundu, `return 4`	3
`ara(44)`	0 (22) → 1 (1) → 2 (13) → 3 (11) → 4 (24) → 5 (33) → 6 (`BOS`)	yok, `return -1`	7

Tablo 14.7: `ara(24)` ve `ara(44)` izlemesi. Tablo: yuva 0–5 = 22, 1, 13, 11, 24, 33; yuva 6–10 boş.

`ara(24)` ev adresi 2'den başlar. Yuva 2'de 13 vardır; durum `DOLU` ama anahtar farklıdır, devam edilir. Yuva 3'te 11 vardır, yine devam. Yuva 4'te 24 bulunur ve 4 döner. Üç yoklama, 24 eklenirken yapılan üç yoklamayla aynıdır. Bu bir rastlantı değildir: başarılı bir arama, anahtarın eklenirken izlediği yolu yeniden yürür.

`ara(44)` daha uzundur. 44'ün ev adresi 0'dır ve 0–5 bloğunun hepsi başka anahtarlarla doludur. Yuva 6 `BOS`'tur: 44 tabloda olsaydı en geç buraya yazılırdı, demek ki yoktur. Başarısız arama yedi yoklama sürmüştür. Başarısız aramalar genellikle başarılı aramalardan uzundur, çünkü her zaman bir boş yuvaya kadar yürür. Tabloda olmayan ve ev adresi 6–10 arasına düşen bir anahtarın araması ise tek yoklamada biter. Test programında `ara(5)` böyle bir aramaydı ve hemen −1 döndü.



Şekil 14.8: `ara(44)` izlemesinin sonu. Ev adresi 0'dan başlayan arama dolu bloğu yürüdü ve yuva 6'da boş yuvaya rastladı: 44 tabloda yok.

Maliyet. Başarılı arama, anahtarın eklenirken yaptığı yoklama sayısı kadar sürer. Başarısız arama, ev adresinden ilk boş yuvaya kadar olan uzaklık kadar sürer. İkisi de en kötü durumda $O(m)$, α sabitken ortalama $O(1)$'dir.

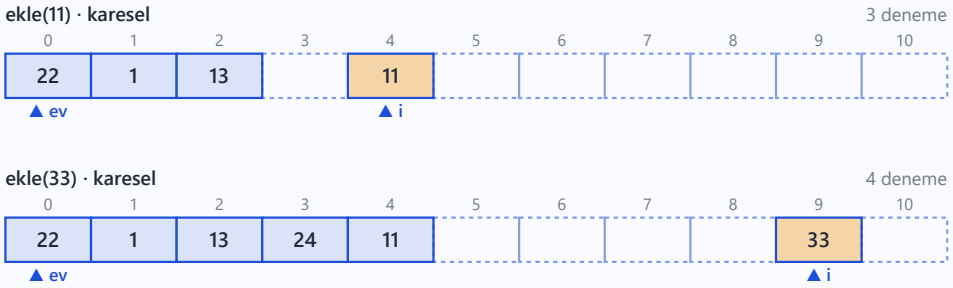
14.3.6 Karesel yoklama

Birincil kümelenmenin nedeni, yoklama dizisinin bitişik yuvalardan geçmesidir. Karesel yoklama, ev adresinden giderek büyüyen adımlarla uzaklaşır.

Tanım 14.11

Karesel yoklama (*quadratic probing*): $h(k, i) = (h(k) + i^2) \bmod m$. Denemeler ev adresinden 0, 1, 4, 9, 16, … yuva uzağa bakar.

Aynı altı anahtar karesel yoklamayla şöyle yerleşir. 22, 1 ve 13 yine ev adreslerine düşer. 11'in yoklama dizisi $0, 0 + 1 = 1, 0 + 4 = 4$'tür: yuva 0 ve 1 doludur, yuva 4 boştur. 24'ün dizisi $2, 2 + 1 = 3$'tür ve yuva 3 boş olduğu için iki yoklamada yerleşir. 33'ün dizisi $0, 1, 4, 9$'dur: yuva 0, 1 ve 4 doludur, 33 yuva 9'a yazılır. 33 için doğrusal yoklamada altı yoklama gerekmişti; karesel yoklamada dört yeter.



Şekil 14.9: Karesel yoklamayla ekleme. Üstte 11, yoklama dizisi 0, 1, 4 ile yuva 4'e yerleşti. Altta 33, yoklama dizisi 0, 1, 4, 9 ile yuva 9'a yerleşti. Dolu yuvalar artık tek bir blok oluşturmaz.

Son tablo 22, 1, 13, 24, 11 (yuva 0–4) ve 33'tür (yuva 9). Karesel yoklama birincil kümelenmeyi giderir: 24 artık 11'in arkasında beklemez, kendi dizisinde boş bulduğu ilk yere yerleşir. Ama ev adresi aynı olan anahtarlar hâlâ aynı diziyi izler. 22, 11 ve 33'ün üçü de 0, 1, 4, 9, … dizisinde ilerler ve birbirinin önünü keser. Bu daha hafif kümelenmeye **ikincil kümelenme** (*secondary clustering*) denir.

Karesel yoklamanın bir sakıncası, yoklama dizisinin bütün yuvaları gezmemesidir. $m = 11$ ve ev adresi 0 için $i = 0, 1, \ldots, 10$ denemeleri sırasıyla 0, 1, 4, 9, 5, 3, 3, 5, 9, 4, 1 yuvalarına bakar. Dizide yalnız altı farklı yuva vardır: 0, 1, 3, 4, 5, 9. Yuva 2, 6, 7, 8 ve 10 hiç yoklanmaz. Tabloda boş yer olduğu hâlde ekleme başarısız olabilir. Şu sonuç bu sorunun sınırını çizer.

Teorem 14.1 (Karesel yoklamada boş yuva)

m asal ise karesel yoklamanın ilk $\lceil m/2 \rceil$ denemesi birbirinden farklı yuvalara bakar. Bu yüzden tablo en çok yarıya kadar doluysa ($\alpha \le 1/2$), yeni anahtar için her zaman boş bir yuva bulunur.

Kanıt. $0 \le i < j < \lceil m/2 \rceil$ olan iki denemenin aynı yuvaya baktığını varsayalım. O zaman $h + i^2 \equiv h + j^2 \pmod m$, yani $m, j^2 - i^2 = (j - i)(j + i)$ çarpımını böler. m asal olduğu için çarpanlardan birini bölmelidir. Ama $0 < j - i < m$ ve $0 < j + i < m$'dir (ikisi de $m/2$'den küçük iki sayının toplamıdır). Bu yüzden iki çarpan da m'nin katı olamaz. Çelişki: ilk $\lceil m/2 \rceil$ deneme farklı yuvalara bakar. Tabloda $m/2$'den az dolu yuva varsa bu farklı yuvalardan en az biri boştur. ■

$m = 11$ için $\lceil 11/2 \rceil = 6$'dır ve gerçekten ilk altı deneme 0, 1, 4, 9, 5, 3 gibi farklı yuvalara bakar. Teoremin pratik sonucu basittir: karesel yoklama kullanan bir tablo asal boyutlu seçilir ve yarıdan fazla doldurulmaz.

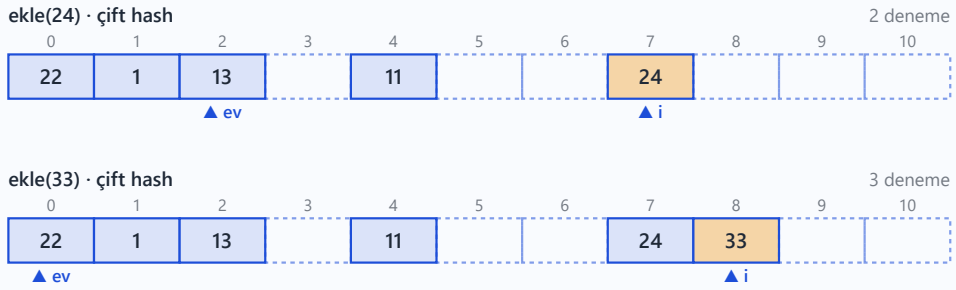
14.3.7 Çift hash

İkincil kümelenmenin nedeni, yoklama dizisinin yalnız ev adresine bağlı olmasıdır. Çift hash, adım boyunu da anahtardan hesaplar.

Tanım 14.12

Çift hash (*double hashing*): iki hash fonksiyonu h_1 ve h_2 ile $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$. h_1 ev adresini, h_2 adım boyunu verir.

Örnekte $h_1(k) = k \bmod 11$ ve $h_2(k) = 1 + (k \bmod 10)$ seçilmiştir. Adım boyları 11 için 2, 24 için 5, 33 için 4'tür. 22, 1 ve 13 ev adreslerine düşer. 11'in dizisi *0, 2, 4*'tür: yuva 0 ve 2 doludur, 11 yuva 4'e yazılır. 24'ün dizisi *2, 7*'dir; yuva 7 boştur. 33'ün dizisi *0, 4, 8*'dir: yuva 0'da 22, yuva 4'te 11 vardır; 33 yuva 8'e yazılır. 33 için üç yoklama yetmiştir.



Şekil 14.10: Çift hash ile ekleme. Üstte 24, adım boyu 5 ile yuva 2'den yuva 7'ye sıçradı. Altta 33, adım boyu 4 ile 0, 4, 8 dizisini izledi ve yuva 8'e yerleşti.

Burada 22, 11 ve 33'ün ev adresi aynıdır ama adım boyları farklıdır (sırasıyla 3, 2 ve 4). Üçü farklı yoklama dizileri izler ve birbirinin yolunu kesme olasılıkları azalır. Doğrusal ve karesel yoklamada yoklama dizisi yalnız ev adresine bağlıdır; m farklı ev adresi m farklı dizi verir. Çift hashte dizi, ev adresi ile adım boyunun çiftine bağlıdır. Bu çiftlerin sayısı m'nin karesi mertebesindedir.

h_2 iki koşulu sağlamalıdır. Birincisi, hiçbir zaman 0 olmamalıdır. $h_2(k) = 0$ olsaydı bütün denemeler ev adresine bakardı ve yoklama hiç ilerlemezdi. h_2'nin başındaki "1 +" bu yüzdendir. İkincisi, $h_2(k)$ ile m aralarında asal olmalıdır. Örneğin $m = 10$ ve adım boyu 6 olsaydı, yuva 5'ten başlayan dizi 5, 1, 7, 3, 9, 5, … diye dönerdi; on yuvanın yalnız beşi yoklanırdı. m asal seçilirse 1 ile $m - 1$ arasındaki her adım boyu m ile aralarında asaldır ve koşul kendiliğinden sağlanır. Örnekteki $m = 11$ asaldır; h_2 1 ile 10 arasında değer aldığı için her dizi 11 yuvanın hepsini gezer.

14.3.8 Üç yoklamanın karşılaştırması

Üç yöntem tek bir fonksiyonla yazılabilir. Aşağıdaki kod, *i*. denemenin yuvasını yönteme göre hesaplayan `yokla` fonksiyonunu ve onu kullanan genel bir ekleme fonksiyonunu verir. `tablo`, `yuva` ve `M` tanımları açık adreslemedeki tanımlarla aynıdır.

```c
typedef enum { DOGRUSAL, KARESEL, CIFT } yontem_t;

int yokla(int k, int i, yontem_t y) {   /* i. denemenin yuvası */
  int h1 = k % M;
  switch (y) {
    case DOGRUSAL: return (h1 + i) % M;
    case KARESEL:  return (h1 + i * i) % M;
    default:       return (h1 + i * (1 + k % 10)) % M;  /* h2 */
  }
}

int ekle_y(int k, yontem_t y) {
  for (int d = 0; d < M; d++) {
    int i = yokla(k, d, y);
    if (tablo[i].durum != DOLU) {
      tablo[i].anahtar = k;
      tablo[i].durum = DOLU;
      return i;
    }
  }
  return -1;
}
```

`yontem_t` üç yöntemi adlandırır. `switch` deyimi yönteme göre formüllerden birini seçer. Çift hash dalında adım boyu `1 + k % 10`'dur. Bu kod m'yi değil 10'u kullandığı için yalnız örnekteki tabloya uygundur. Genel bir tabloda h_2, örneğin $1 + (k \bmod (m-2))$ gibi m'ye bağlı seçilir. `ekle_y`, `ekle`'nin aynısıdır; tek farkı yuvayı `i = (i + 1) % M` yerine `yokla` ile bulmasıdır. Karesel yoklamada `i * i` çarpımı, $i < M$ olduğu sürece taşmaz. Test programı altı anahtarı üç yöntemle ayrı ayrı ekledi; sonuçlar Tablo 14.8 ile birebir aynıdır.

Yöntem	i. deneme	33'ün yoklama dizisi	Yoklama	Son tablo (yuva: anahtar)
Doğrusal	$h + i$	0, 1, 2, 3, 4, 5	6	0: 22, 1: 1, 2: 13, 3: 11, 4: 24, 5: 33
Karesel	$h + i^2$	0, 1, 4, 9	4	0: 22, 1: 1, 2: 13, 3: 24, 4: 11, 9: 33
Çift hash	$h_1 + i \cdot h_2$	0, 4, 8	3	0: 22, 1: 1, 2: 13, 4: 11, 7: 24, 8: 33

Tablo 14.8: Aynı tablo, üç yoklama: 33 nereye düşer? (22, 1, 13, 11, 24 ondan önce eklendi.)

Yöntem	Kümelenme	Farklı yoklama dizisi
Doğrusal	birincil (bloklar)	m
Karesel	ikincil (aynı ev adresi)	m
Çift hash	en az	$\approx m^2$

Tablo 14.9: Üç yoklamanın kümelenme türü ve farklı yoklama dizisi sayısı.

Bu tablolara bakınca çift hashin her zaman seçilmesi gerektiği düşünülebilir. Gerçek bilgisayarlarda durum daha karışıktır. İşlemci belleği tek tek bayt olarak değil, 64 baytlık bloklar hâlinde önbelleğe getirir. Doğrusal yoklamada ardışık yuvalar aynı bloktadır: ilk yoklama bloğu getirince sonraki birkaç yoklama neredeyse bedavaya yapılır. Çift hashin sıçramaları ise her yoklamada başka bir bloğa düşer. Bu yüzden doğrusal yoklama önbellek dostudur ve α küçükken (örneğin 0,5'in altında) çok hızlıdır. α büyüdükçe kümelenme yüzünden yoklama sayısı hızla arttığı için çift hash öne geçer. Seçim, tablonun ne kadar dolu tutulacağına bağlıdır.

14.4 Zincirleme

Açık adreslemede çakışan öğe başka bir yuvaya gider ve başka anahtarların yerini işgal eder. Zincirleme bunun tersini yapar: çakışan öğe kendi kovasında kalır. Bir kovada birden çok öğe tutulabilmesi için her kova bir bağlı listenin başı olur.

14.4.1 Kova başına bir bağlı liste

Tanım 14.13

Ayrı zincirleme (*separate chaining*), kısaca zincirleme: tablonun her kovası bir bağlı listenin baş işaretçisidir. Ev adresi *j* olan bütün anahtarlar *j*. kovanın listesinde, yani zincirinde tutulur.

```
#define MZ 5
typedef struct dugum {
  int anahtar;
  struct dugum *sonraki;
} dugum;
dugum *kova[MZ];          /* hepsi NULL */
```

Tanımlar bağlı listenin alışılmış düğümüdür. `struct dugum`'ün iki alanı vardır: anahtar ve bir sonraki düğümün adresi. Yapı kendi tipinde bir işaretçi taşır ve bu işaretçi `typedef` bitmeden yazılır. Bu yüzden yapıya hem etiket adı (`struct dugum`) hem tip adı (`dugum`) verilmiştir. `kova` dizisi beş işaretçiden oluşur. Global olduğu için program başlarken bütün işaretçiler `NULL`'dır: beş kova da boş listedir.

Bellekteki görünüm açık adreslemeden farklıdır. `kova` dizisinin kendisi küçük ve bitişiktir: beş işaretçi, 64 bitlik bir makinede 40 bayt. Düğümler ise `malloc` ile tek tek alınır ve heap'in rastgele yerlerine dağılır. Test makinesinde `sizeof(dugum)` 16 bayt çıktı: 4 baytlık anahtar, işaretçinin 8 baytlık sınıra hizalanması için 4 baytlık boşluk ve 8 baytlık işaretçi. Her öğe için tutulan 4 baytlık anahtara 12 baytlık ek yük eklenir. `malloc`'un her blok için ayrıca tuttuğu yönetim bilgisi de bu hesaba dahil değildir.

```
bool zincir_ekle(int k) {
  int i = k % MZ;
  dugum *yeni = malloc(sizeof *yeni);
  if (yeni == NULL) return false;  /* bellek yok */
  yeni->anahtar = k;
  yeni->sonraki = kova[i];         /* başa ekle */
  kova[i] = yeni;
  return true;
}
```

- **Satır 2:** `i`, anahtarın kovasıdır.
- **Satır 3:** Yeni düğüm için bellek alınır. `sizeof *yeni`, `yeni`'nin gösterdiği tipin boyutudur, yani `sizeof(dugum)`. Bu yazım, tip adı değişse bile doğru kalır. `malloc`, `<stdlib.h>` başlığındadır.
- **Satır 4:** `malloc` bellek bulamazsa `NULL` döndürür. Bu denetim atlanırsa sonraki satır `NULL` adresine yazmaya çalışır ve program çöker. Fonksiyon `false` döndürerek hatayı çağırana bildirir.
- **Satır 5–7:** Düğüm listenin başına eklenir. Önce yeni düğümün `sonraki` alanı eski başı gösterir, sonra kova yeni düğümü gösterir. Sıra önemlidir: önce `kova[i] = yeni` yapılsaydı eski liste kaybolurdu.

Başa ekleme, liste ne kadar uzun olursa olsun iki işaretçi ataması sürer: *O(1)*. Listenin sonuna eklemek için bütün liste yürünmesi gerekirdi. Zincirdeki sıranın bir anlamı olmadığı için başa eklemek en ucuz seçimdir. Yalnız bu fonksiyon da aynı anahtarın ikinci kez eklenmesini engellemez. Tekrar istenmiyorsa eklemeden önce `zincir_ara` çağrılmalıdır; o zaman ekleme maliyeti arama maliyetine eşit olur.

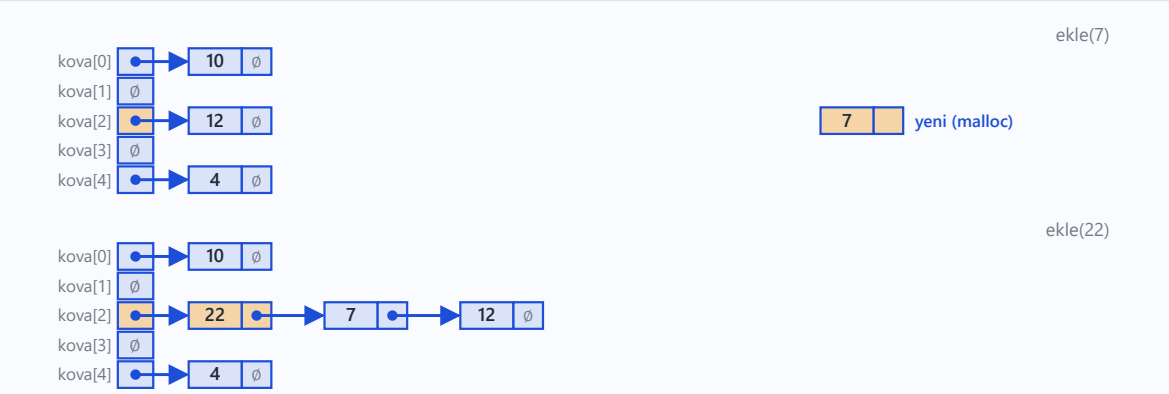
14.4.2 Kod izleme: `zincir_ekle`

Tablo beş kovalıdır, *h(k) = k mod 5*'tir. Başlangıçta kova 0'da 10, kova 4'te 4 vardır. Sırayla 12, 7 ve 22 eklenir; üçünün de kovası 2'dir. Tablo 14.10 her çağrıyı izler.

Çağrı	`i`	`yeni->sonraki`	`kova[2]` sonrası	Zincir 2
`zincir_ekle(12)`	2	`NULL` (eski baş)	düğüm(12)	12
`zincir_ekle(7)`	2	düğüm(12)	düğüm(7)	7 → 12
`zincir_ekle(22)`	2	düğüm(7)	düğüm(22)	22 → 7 → 12

Tablo 14.10: `zincir_ekle` ile 12, 7 ve 22'nin eklenmesi (*MZ = 5*).

İlk çağrıda kova 2 boştur. Yeni düğümün `sonraki` alanına eski baş, yani `NULL` yazılır ve kova yeni düğümü gösterir. Boş bir listeye ekleme ayrı bir durum gerektirmez; aynı iki atama doğru sonucu verir. İkinci ve üçüncü çağrılarda yeni düğüm her seferinde eski başın önüne girer. Üç ekleme sonunda zincir 22 → 7 → 12 olur: son eklenen başta durur. Her çağrı bir `malloc` yapmıştır.



Şekil 14.11: `zincir_ekle` izlemesinden iki an. Üstte `zincir_ekle(7)` : yeni düğümün `sonraki` alanı eski başı (12) gösteriyor, kova henüz güncellenmedi. Altta üç ekleme bitti: kova 2'nin zinciri 22 → 7 → 12. Ø işareti `NULL` 'dır.

14.4.3 Kod izleme: `zincir_ara`

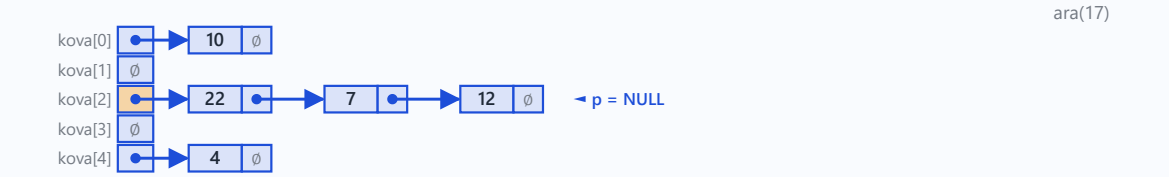
```
dugum *zincir_ara(int k) {
  dugum *p = kova[k % MZ];
  while (p != NULL && p->anahtar != k)
    p = p->sonraki;
  return p;                 /* NULL: yok */
}
```

Arama, kovanın listesini baştan yürür. `p` listenin başından başlar. `while` koşulu iki şey sorar: liste bitti mi, aranan bulundu mu? İkisi de değilse `p` bir sonraki düğüme geçer. Koşulun sırası önemlidir. `&&` işleci kısa devre yapar: `p != NULL` yanlışsa ikinci koşul hiç hesaplanmaz. Sıra ters yazılsaydı liste sonunda `NULL->anahtar` okunur ve program çökerdi. Döngü bitince `p` ya bulunan düğümü ya da `NULL` 'ı gösterir; fonksiyon ikisini de olduğu gibi döndürür.

Çağrı	`p` 'nin gösterdiği düğümler	Sonuç	Gezilen düğüm
`zincir_ara(12)`	22 (≠ 12) → 7 (≠ 12) → 12 (eşit)	düğüm(12) adresi	3
`zincir_ara(17)`	22 (≠ 17) → 7 (≠ 17) → 12 (≠ 17) → `NULL`	`NULL`	3

Tablo 14.11: `zincir_ara(12)` ve `zincir_ara(17)` izlemesi. Kova 2'nin zinciri 22 → 7 → 12.

12, zincirin sonundadır; çünkü ilk eklenen odur. Arama üç düğüme bakar. 17'nin kovası da 2'dir (*17 mod 5 = 2*). Zincirin üç düğümü de 17'den farklıdır ve `p` sonunda `NULL` olur. Başarısız arama, zincirin tamamını yürümek zorundadır. Kova 1 ya da 3'e düşen bir anahtarın başarısız araması ise hiç düğüme bakmadan biter: liste boştur.



Şekil 14.12: `zincir_ara(17)` izlemesinin sonu: `p` zincirin üç düğümünü geçti ve `NULL` oldu; 17 tabloda yok.

Maliyet. Arama, ev adresinin hesabı ile zincirde gezilen düğüm sayısının toplamı kadar sürer. Basit düzgün hash varsayımı altında her zincirin beklenen uzunluğu $\alpha = n/m$'dir. Başarısız arama bütün zinciri yürüdüğü için ortalama α düğüme bakar; hash hesabıyla birlikte $\Theta(1 + \alpha)$. Başarılı arama, aranan düğüme kadar yürür. Aranan öğe eklendikten sonra aynı kovaya eklenenler onun önündedir ve bunların

beklenen sayısı yaklaşık $\alpha/2$'dir. Bu yüzden başarılı arama ortalama $1 + \alpha/2$ düğüme bakar. İki durumda da α sabit tutulursa maliyet $O(1)$'dir. En kötü durumda bütün anahtarlar aynı kovaya düşer ve arama $O(n)$ olur.

14.4.4 Kod izleme: `zincir_sil`

Tek bağlı listeden düğüm silmek için silinecek düğümün öncülü bilinmelidir. Öncülün `sonraki` alanı, silinen düğümü atlayıp ondan sonrakini gösterecek biçimde değiştirilir. Bu yüzden silme yordamı iki işaretçiyle yürür.

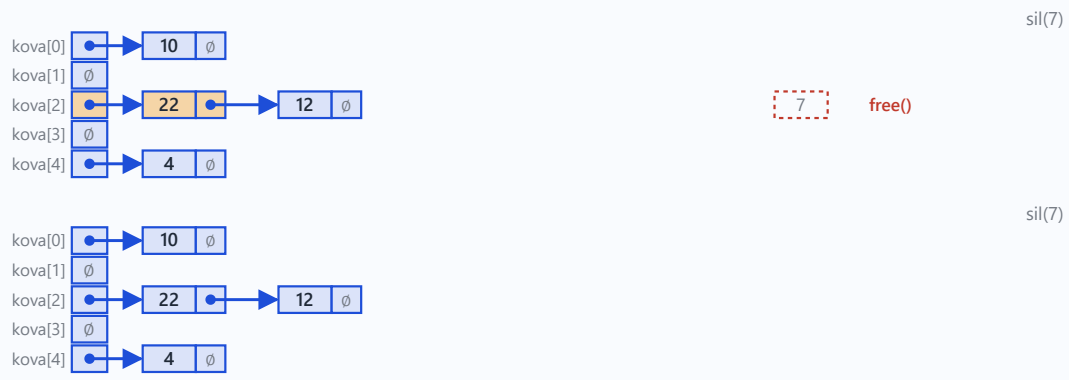
```
bool zincir_sil(int k) {
  int i = k % MZ;
  dugum *p = kova[i], *onceki = NULL;
  while (p != NULL && p->anahtar != k) {
    onceki = p;
    p = p->sonraki;
  }
  if (p == NULL) return false;          /* yok */
  if (onceki == NULL) kova[i] = p->sonraki;
  else onceki->sonraki = p->sonraki;
  free(p);
  return true;
}
```

- **Satır 3:** `p` bakılan düğümdür, `onceki` onun öncülüdür. Listenin başındaki düğümün öncülü yoktur; `onceki` bu yüzden `NULL` ile başlar.
- **Satır 4–7:** Arama döngüsü `zincir_ara`'nınkiyle aynıdır. Tek farkı, `p` ilerlemeden önce eski değerinin `onceki`'ye yazılmasıdır.
- **Satır 8:** `p` `NULL` ise anahtar yoktur ve `false` döner.
- **Satır 9–10:** Düğüm zincirden çıkarılır. `onceki` `NULL` ise silinen düğüm listenin başıdır; kova bir sonraki düğümü gösterecek biçimde güncellenir. Değilse öncülün `sonraki` alanı `p`'nin ardılını gösterir.
- **Satır 11:** Düğümün belleği `free` ile geri verilir.

Tablo 14.12 `zincir_sil(7)` çağrısını izler; 7 zincirin ortasındadır.

Adım	`onceki`	`p`	Olay
1	`NULL`	düğüm(22)	$i = 7 \bmod 5 = 2$; `p` listenin başında
2	düğüm(22)	düğüm(7)	$22 \neq 7$: `onceki` ve `p` bir adım ilerler
3	düğüm(22)	düğüm(7)	`p->anahtar == 7` : döngü biter
4	düğüm(22)	düğüm(7)	`onceki->sonraki = p->sonraki` : 22 artık 12'yi gösterir
5	düğüm(22)	(serbest)	`free(p)` ; `return true` ; zincir 22 → 12

Tablo 14.12: `zincir_sil(7)` izlemesi. Kova 2'nin zinciri 22 → 7 → 12.



Şekil 14.13: `zincir_sil(7)`. Üstte 7'nin düğümü zincirden çıkarılmış ve `free` edilmek üzere; 22 doğrudan 12'yi gösteriyor. Altta işlem bitmiş: kova 2'nin zinciri 22 → 12.

4. adımdan sonra 7'nin düğümüne zincirden hiçbir işaretçi ulaşmaz. Bu anda `free` çağrılmasaydı düğüm bellekte kalır ama bir daha hiç kullanılamazdı: bellek sızıntısı. `free(p)` 'den sonra ise `p` hâlâ eski adresi tutar. Bu adres artık programa ait değildir; `p` bir **sarkan işaretçidir**. `p->anahtar` okumak ya da `p->sonraki` 'yi izlemek tanımsız davranıştır. Fonksiyon `free` 'den hemen sonra döndüğü için bu tehlike burada doğmaz. Test programında silinen 7 için ikinci bir `zincir_sil(7)` çağrısı `false` döndürdü. Baştaki 22'nin silinmesi de denendi: `onceki` `NULL` kaldı, `kova[2]` 12'nin düğümünü gösterdi.

Zincirlemede silme, açık adreslemedeki gibi bir işaret gerektirmez. Düğüm listeden gerçekten çıkarılır ve başka hiçbir anahtarın arama yolu değişmez. Her anahtar yalnız kendi kovasının listesinde aranır; silinen düğüm başka bir kovanın aramasını kesemez. **Maliyet:** silme, aramayla aynı yolu yürür ve bulduktan sonra sabit iş yapar. Ortalama $O(1 + \alpha)$, en kötü $O(n)$.

> **Sık yapılan hata**
>
> Baştaki düğümün silinmesini unutmak. `onceki->sonraki = p->sonraki` satırı, `onceki` `NULL` iken çalışırsa `NULL` adresine yazar ve program çöker. Silinecek düğüm listenin başındaysa güncellenmesi gereken işaretçi bir düğümün alanı değil, `kova[i]` 'nin kendisidir.

14.4.5 Tablonun bütün belleğini geri vermek

Zincirli tablonun düğümleri `malloc` ile alındığı için program bitmeden ya da tablo artık gerekmediğinde hepsi geri verilmelidir. Sunumda bu yordam yoktur; aşağıdaki kod onu tamamlar.

```
void zincir_bosalt(void) {
  for (int i = 0; i < MZ; i++) {
    dugum *p = kova[i];
    while (p != NULL) {
      dugum *sonra = p->sonraki;  /* free'den önce sakla */
      free(p);
      p = sonra;
    }
    kova[i] = NULL;
  }
}
```

Dış döngü kovalarda, iç döngü her kovanın zincirinde gezinir. Kodun tek inceliği `sonra` değişkenidir. `free(p)` 'den sonra `p->sonraki` okunamaz, çünkü düğümün belleği artık programa ait değildir. Bu yüzden sonraki düğümün adresi `free` 'den önce saklanır. Kova sonunda `NULL` yapılır ve tablo yeniden kullanılabilir hâle gelir. Maliyet $O(n + m)$'dir: her düğüm bir kez, her kova bir kez ele alınır.

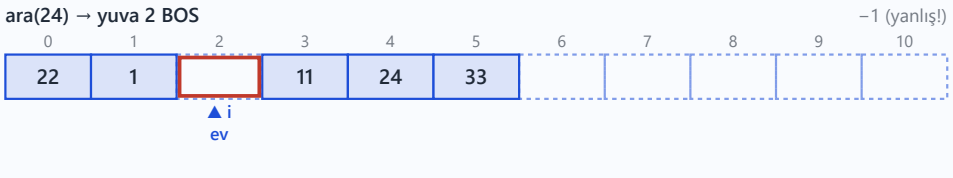
Bellek sızıntısı, Windows'ta AddressSanitizer kullanılamadığı için sayaçlı bir `malloc` / `free` sarmalayıcısıyla denetlendi. Sarmalayıcı her başarılı ayırmada bir sayacı, her `free` 'de ötekini artırır. Test programı 10, 4, 12, 7 ve 22'yi ekledi, 7 ve 22'yi sildi, sonra `zincir_bosalt` çağırdı. Sayaçlar 5 ayırma ve 5 bırakma gösterdi: sızıntı yoktur.

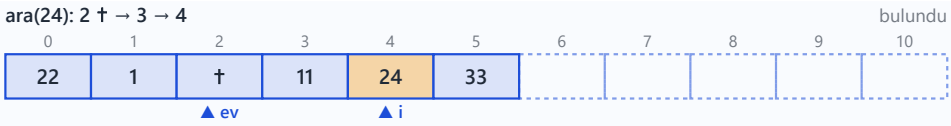
14.5 Silme, yük faktörü ve yeniden boyutlandırma

Açık adreslemede ekleme ve arama kurulmuştu; silme ise ertelenmişti. Silmenin neden ayrı bir kesim gerektirdiği ilk örnekte görülür. Kesimin ikinci yarısı tablonun ne kadar dolu tutulması gerektiğini ve eşik aşılınca tablonun nasıl büyütüleceğini ele alır.

14.5.1 Açık adreslemede silme tuzağı

Doğrusal yoklamalı tabloda 22, 1, 13, 11, 24 ve 33 vardır. 13 silinecektir. Akla gelen ilk çözüm, 13'ün yuvasını, yani yuva 2'yi `BOS` yapmaktır. Ardından `ara(24)` çağrılsın. 24'ün ev adresi 2'dir; 24, eklendiği sırada yuva 2 ve 3 dolu olduğu için yuva 4'e yerleşmişti. Arama ev adresine bakar, yuva 2'yi `BOS` bulur ve hemen −1 döndürür. Oysa 24 tablodadır.





Şekil 14.14: Açık adreslemede silme. Üstte yanlış yol: yuva 2 `BOS` yapıldı ve `ara(24)` ev adresinde boş yuva görüp durdu; 24 bulunamadı. Altta doğru yol: yuva 2 silindi işaretiyle (†) işaretlendi; arama bu yuvayı geçip 3'e ve 4'e baktı ve 24'ü buldu.

Sorunun kaynağı, aramanın durma kuralıdır. Arama, "anahtar olsaydı ilk `BOS` yuvadan önce bulunurdu" varsayımıyla durur. Bu varsayım, yuvalar yalnız `BOS` 'tan `DOLU` 'ya geçtiği sürece doğrudur. Bir `DOLU` yuva yeniden `BOS` olunca, o yuvadan geçerek yerleşmiş bütün anahtarların yolu kesilir. Silinen yuvanın gerisinde kalan anahtarlar tablodadır ama bulunamaz.

Tanım 14.14

Silindi işareti (*tombstone*): açık adreslemede silinen yuvaya konan özel durumdur (koddaki `SILINDI`). Arama bu yuvayı boş saymaz, geçip devam eder. Ekleme ise bu yuvayı boş sayar ve yeniden kullanabilir.

İngilizce adı "mezar taşı"dır: yuvada artık bir öğe yaşamaz ama orada bir öğenin bulunduğu kaydedilmiştir. Silindi işareti iki kurala da uyar. Aramanın yolu kesilmez, çünkü işaretli yuva arama için doludur. Bellek boşa gitmez, çünkü işaretli yuva ekleme için boştur.

14.5.2 Kod izleme: `sil(13)` , ardından `ara(24)`

Silme yordamı, aramada verilen kod parçasının ikinci fonksiyonudur:

```
int ara(int k) {
  int i = k % M;
  for (int d = 0; d < M; d++, i = (i + 1) % M) {
    if (tablo[i].durum == BOS) return -1;
    if (tablo[i].durum == DOLU &&
        tablo[i].anahtar == k) return i;
  }
  return -1;
}
bool sil(int k) {
  int i = ara(k);
  if (i < 0) return false;
  tablo[i].durum = SILINDI;   /* silindi işareti */
  return true;
}
```

`sil` önce `ara` ile anahtarın yuvasını bulur. Anahtar yoksa `false` döner. Varsa yuvanın durumunu `SILINDI` yapar. Anahtar alanına dokunulmaz; eski değer yuvada kalır. Bu bir sorun yaratmaz, çünkü `ara` bir anahtarı ancak durumu `DOLU` ise kabul eder. Tablo 14.13 silmeyi ve ardından yapılan aramayı izler.

Çağrı	Adım	Olay
`sil(13)`	1	`ara(13)` : ev adresi *13 mod 11 = 2*; yuva 2 `DOLU` ve anahtar 13 → 2 döner (1 yoklama)
	2	`i = 2 ≥ 0` : silinecek öğe var; `tablo[2].durum = SILINDI`
	3	`return true`
`ara(24)`	1	ev adresi *24 mod 11 = 2*; yuva 2 `SILINDI` : ne `BOS` ne `DOLU` , devam
	2	yuva 3: `DOLU` , anahtar 11 ≠ 24, devam
	3	yuva 4: `DOLU` , anahtar 24 → `return 4` (3 yoklama)

Tablo 14.13: `sil(13)` ve ardından `ara(24)` izlemesi. Başlangıç: yuva 0–5 = 22, 1, 13, 11, 24, 33.

`ara(24)` yuva 2'deki silindi işaretinde durmaz; iki `if` koşulu da yanlış çıkar ve döngü ilerler. Arama, silmeden önceki gibi üç yoklamada 24'ü bulur. Test programı aynı sırayı çalıştırdı: `sil(13)` 1 (`true`), sonraki `ara(24)` 4, `ara(13)` −1 ve ikinci bir `sil(13)` 0 (`false`) döndürdü. Yuva 2 elle `BOS` yapıldığında ise `ara(24)` −1 döndürdü; tuzak gerçekten oluştu.

Ekleme ve tekrar eden anahtar. `ekle`, `SILINDI` yuvayı yeniden kullanır. Bu, sunumun alıştırmasında sorulan bir tehlike doğurur. Tabloda 13 silinmiş, 24 yuva 4'te dururken `ekle(24)` çağrılsın. `ekle` ev adresi 2'ye bakar, yuvayı `DOLU` değil diye kabul eder ve 24'ü oraya yazar. Tabloda artık iki tane 24 vardır: yuva 2'de ve yuva 4'te. Test programında bu çağrı gerçekten 2 döndürdü ve tablo 22, 1, 24, 11, 24, 33 oldu. Daha sonra `sil(24)` yalnız ilk kopyayı siler; ikinci kopya tabloda kalır ve `ara(24)` silinmiş bir anahtarı bulmaya devam eder. Çözüm, eklemeden önce aramaktır:

```
int ekle_tekil(int k) {        /* anahtar zaten varsa yeniden eklemez */
  int i = ara(k);
  if (i >= 0) return i;        /* var: yerini döndür */
  return ekle(k);              /* ilk BOS ya da SILINDI yuvaya yaz */
}
```

`ekle_tekil`, anahtarın yoklama dizisinin tamamını `ara` ile tarar. Anahtar varsa onun yerini döndürür ve yeni kopya yazmaz. Yoksa `ekle`, dizideki ilk `BOS` ya da `SILINDI` yuvaya yazar. Test programında 13 silindikten sonra `ekle_tekil(24)` 4 döndürdü ve tablo değişmedi; `ekle_tekil(46)` ise 46'yı silindi işaretli yuva 2'ye yazdı. Bu sürüm, silindi işaretlerinin yerini geri kazanır ve tekrar eden anahtara izin vermez. Bedeli, her eklemede bir aramadır.

Silindi işaretinin de bir maliyeti vardır. İşaretli yuvalar arama için dolu sayıldığı için aramaları uzatır. Çok sayıda ekleme ve silme yapılan bir tabloda, gerçek öğe sayısı küçük olsa bile yuvaların büyük kısmı zamanla `SILINDI` olabilir. Bu durumda başarısız aramalar neredeyse bütün tabloyu tarar. Çare, tabloyu belirli aralıklarla yeniden kurmaktır: bütün `DOLU` öğeler yeni ve temiz bir tabloya yeniden eklenir, işaretler atılır. Bu işlem yeniden boyutlandırmayla aynıdır. Zincirlemede bu sorun hiç doğmaz.

14.5.3 Yük faktörü ve beklenen yoklama sayısı

Bütün çakışma çözümlerinde maliyet, tablonun doluluğuyla artar. Yük faktörü $\alpha = n/m$ bu doluluğun ölçüsüdür. Hash tablolarının kuramsal analizi, basit düzgün hash varsayımı altında beklenen yoklama sayısını α'nın fonksiyonu olarak verir. Tablo 14.14 dört α değeri için sonuçları özetler.

α	Zincir ✓	Zincir ✗	Doğrusal ✓	Doğrusal ✗	Düzgün ✓	Düzgün ✗
0,25	1,13	1,25	1,2	1,4	1,15	1,3
0,5	1,25	1,50	1,5	2,5	1,39	2,0
0,75	1,38	1,75	2,5	8,5	1,85	4,0
0,9	1,45	1,90	5,5	50,5	2,56	10,0

Tablo 14.14: Beklenen yoklama sayısı. ✓ başarılı, ✗ başarısız arama. "Düzgün" sütunları, her anahtarın yoklama dizisinin rastgele bir sıralama olduğu ideal açık adreslemedir.

Tablodaki sayılar şu formüllerden gelir:

- **Zincirleme:** başarılı arama $1 + \alpha/2$, başarısız arama $1 + \alpha$ (ilk 1, kovaya erişimdir). İkisi de $\Theta(1 + \alpha)$.
- **Doğrusal yoklama** (Knuth): başarılı arama $\tfrac{1}{2}\,(1 + 1/(1 - \alpha))$, başarısız arama $\tfrac{1}{2}\,(1 + 1/(1 - \alpha)^2)$.
- **Düzgün açık adresleme** (CLRS): başarılı arama $(1/\alpha) \cdot \ln(1/(1 - \alpha))$, başarısız arama $1/(1 - \alpha)$. Çift hash bu ideale çok yakın davranır.

Başarısız aramanın $1/(1 - \alpha)$ formülünün arkasında basit bir düşünce vardır. Her yoklama, rastgele bir yuvaya bakar ve bu yuva α olasılıkla doludur. İlk yoklama her zaman yapılır. İkinci yoklamaya ilk yuva doluysa geçilir: olasılık α. Üçüncüye ilk iki yuva doluysa geçilir: olasılık yaklaşık α^2. Beklenen toplam $1 + \alpha + \alpha^2 + \dots = 1/(1 - \alpha)$ geometrik toplamıdır. $\alpha = 0{,}9$ iken bu 10 eder: on yuvadan biri boştur ve boş yuvaya rastlamak ortalama on deneme sürer.

Formüllerin ikinci dersi, maliyetin α ile doğrusal değil, $1/(1 - \alpha)$ ile büyümesidir. α 0,5'ten 0,75'e çıkınca doğrusal yoklamada başarısız arama 2,5'ten 8,5'e, 0,9'da 50,5'e çıkar. Zincirlemede ise artış yumuşaktır: $\alpha = 0{,}9$'da başarısız arama 1,9 yoklamadır ve α 1'i aştığında da tablo çalışmaya devam eder.

Formüller bir varsayıma dayandığı için ölçümle denetlendi. 100.003 yuvalı (asal) bir tabloya, sabit tohumla üretilen sözde rastgele anahtarlar eklendi. Her α için başarılı aramada ortalama yoklama sayısı eklenen bütün anahtarlar üzerinden, başarısız aramada ise tabloda olmayan 20.000 anahtar üzerinden hesaplandı. Çift hashte $h_2(k) = 1 + (k \bmod (m - 2))$ kullanıldı. Tablo 14.15 sonuçları formüllerle yan yana verir. Anahtarlar sabit tohumla üretildiği için sayılar makineden bağımsızdır; yalnız süre ölçülmemiştir.

α	Doğrusal ✓	Doğrusal ✗	Çift hash ✓	Çift hash ✗	Zincir ✓	Zincir ✗ (düğüm)
0,25	1,170 (1,167)	1,384 (1,389)	1,154 (1,151)	1,331 (1,333)	1,128 (1,125)	0,244 (0,25)
0,5	1,500 (1,500)	2,499 (2,500)	1,389 (1,386)	2,005 (2,000)	1,248 (1,250)	0,496 (0,50)
0,75	2,521 (2,500)	8,826 (8,500)	1,855 (1,848)	3,991 (4,000)	1,374 (1,375)	0,738 (0,75)
0,9	5,636 (5,500)	51,118 (50,500)	2,551 (2,558)	10,084 (10,000)	1,450 (1,450)	0,907 (0,90)

Tablo 14.15: Ölçülen ortalama yoklama sayısı (parantez içinde formül), $m = 100.003$. Zincirlemede başarısız arama için gezilen düğüm sayısı verildi; formüldeki $1 + \alpha$'nın 1'i kovaya erişimdir.

Ölçümler formüllerle uyumludur; sapmalar %4'ün altındadır. En büyük sapmalar doğrusal yoklamada, $\alpha = 0{,}75$ ve $0{,}9$'da görülür (%2–4). Yüksek dolulukta birkaç uzun blok ortalamayı belirgin biçimde etkiler. Çift hashin değerleri, düzgün açık adresleme formülünü üç basamağa kadar tutturur.

Teorem 14.2 (Sabit yük faktörüyle sabit maliyet)

Basit düzgün hash varsayımı altında, yük faktörü 1'den küçük bir α_0 sabitiyle sınırlı tutulursa ($\alpha \le \alpha_0 < 1$), açık adreslemede ve zincirlemede arama, ekleme ve silmenin beklenen maliyeti $O(1)$'dir. Zincirlemede α'nın herhangi bir sabitle sınırlı olması yeter.

Kanıt. Zincirlemede beklenen maliyet $\Theta(1 + \alpha)$'dır; α sabitle sınırlıysa bu da sabittir. Düzgün açık adreslemede başarısız arama ve ekleme en çok $1/(1 - \alpha) \le 1/(1 - \alpha_0)$, başarılı arama bundan da azdır. Sağ taraf n'ye bağlı değildir. Silme, bir arama ve sabit iş olduğu için aynı sınırı taşır. ■

Pratikte kullanılan eşikler şöyledir: açık adreslemede $\alpha \le 0{,}5$–$0{,}7$, zincirlemede $\alpha \le 1$ civarı. Karesel yoklamada teorem gereği eşik en çok 0,5 olmalıdır. Eşik aşılınca tablo büyütülür.

14.5.4 Yeniden boyutlandırma

Tablo boyutu sabit bir makroyla verilirse yük faktörü öğe sayısıyla birlikte büyür. Öğe sayısı önceden bilinmiyorsa tablonun dizisi dinamik olmalı ve eşik aşılınca büyütülmelidir.

```
typedef struct {
  yuva *a;      /* m yuvalık dinamik dizi */
  int m, n;     /* boyut, dolu yuva sayısı */
} Tablo;
/* ekledikten sonra: */
if (2 * t->n > t->m)            /* α > 1/2 */
  yeniden_boyutla(t, sonraki_asal(2 * t->m));
```

`Tablo` yapısı üç alan taşır: yuva dizisinin adresi, yuva sayısı `m` ve dolu yuva sayısı `n`. Dizi artık `malloc` ya da `calloc` ile alınır ve boyutu çalışma sırasında değişebilir. Kodun ikinci kısmı, her eklemeden sonra çalışacak denetimdir. `2 * t->n > t->m` koşulu $\alpha > 1/2$ demektir; kesirli sayı kullanmamak için iki taraf $2m$ ile çarpılmıştır. Eşik aşılınca tablo, eski boyutun iki katından küçük olmayan ilk asal boyuta büyütülür. Asal seçim bölme yöntemi ve karesel yoklama için gereklidir.

Sunumdaki parça, iki yardımcı fonksiyonun var olduğunu varsayar. Test programı için yazılan sürümleri şunlardır:

```
int sonraki_asal(int x) {        /* x'ten küçük olmayan ilk asal */
  for (;; x++) {
    int asal = x >= 2;
    for (int d = 2; d * d <= x && asal; d++)
      if (x % d == 0) asal = 0;
    if (asal) return x;
  }
}

int tablo_ekle(Tablo *t, int k) {  /* doğrusal yoklama, m = t->m */
  int i = k % t->m;
  for (int d = 0; d < t->m; d++, i = (i + 1) % t->m) {
    if (t->a[i].durum != DOLU) {
      t->a[i].anahtar = k;
      t->a[i].durum = DOLU;
      t->n++;
      return i;
    }
  }
  return -1;
}
```

`sonraki_asal` , `x` 'ten başlayarak ilk asalı bulur. Her adayı $\sqrt{x}$'e kadar olan bölenlerle dener; `d * d <= x` koşulu karekök hesaplamadan bu sınırı uygular. `tablo_ekle` , açık adreslemedeki `ekle` 'nin dinamik tablo sürümüdür. Tek farkları, boyutu `M` makrosundan değil `t->m` alanından almak ve dolu yuva sayacını artırmaktır. 5'ten başlayan ve her seferinde iki katın ilk asalını alan boyut dizisi 5, 11, 23, 47, 97, 197, 397, 797, 1597, 3203, … biçiminde ilerler.

Neden yeniden hash? Büyütmenin akla gelen ilk yolu, eski diziyi yeni dizinin başına kopyalamaktır. Bu yanlıştır. Bir öğenin yuvası *k mod m*'ye bağlıdır ve *m* değişince ev adresi de değişir. 12, 5 yuvalı tabloda *12 mod 5 = 2*'dedir; 11 yuvalı tabloda ev adresi *12 mod 11 = 1*'dir. Yuva 2'ye kopyalanan 12, yeni tabloda `ara(12)` ile bulunamaz: arama yuva 1'e bakar, boş bulur ve durur. Her öğe yeni tabloya baştan eklenmelidir. Bu işleme **yeniden hash'leme** (*rehashing*) denir.

```
bool yeniden_boyutla(Tablo *t, int yeni_m) {
  yuva *eski = t->a;
  int eski_m = t->m;
  yuva *a = calloc(yeni_m, sizeof *a);  /* BOS */
  if (a == NULL) return false;
  t->a = a; t->m = yeni_m; t->n = 0;
  for (int i = 0; i < eski_m; i++)
    if (eski[i].durum == DOLU)
      tablo_ekle(t, eski[i].anahtar);
  free(eski);
  return true;
}
```

- **Satır 2–3:** Eski dizinin adresi ve boyutu yerel değişkenlerde saklanır. Tablo yapısı birazdan yeni diziye bağlanacağı için eski dizi ancak bu kopyalar üzerinden okunabilir.
- **Satır 4–5:** Yeni dizi `calloc` ile alınır. `calloc` , `malloc` 'tan farklı olarak belleği sıfırlar; `BOS` = 0 olduğu için bütün yuvalar boş başlar. Bellek alınamazsa fonksiyon `false` döndürür. Bu durumda tablo hiç değişmemiştir: eski dizi yerindedir, sadece büyütülememiştir.
- **Satır 6:** Tablo yeni diziye bağlanır ve `n` sıfırlanır; `tablo_ekle` her eklemede `n` 'yi yeniden artıracaktır.
- **Satır 7–9:** Eski dizinin her yuvasına bakılır. Yalnız `DOLU` yuvalar yeni tabloya eklenir. `BOS` ve `SILINDI` yuvalar atlanır; böylece silindi işaretleri de temizlenir.
- **Satır 10:** Eski dizi `free` edilir. Bu satır unutulursa her büyütme eski diziyi sızdırır.

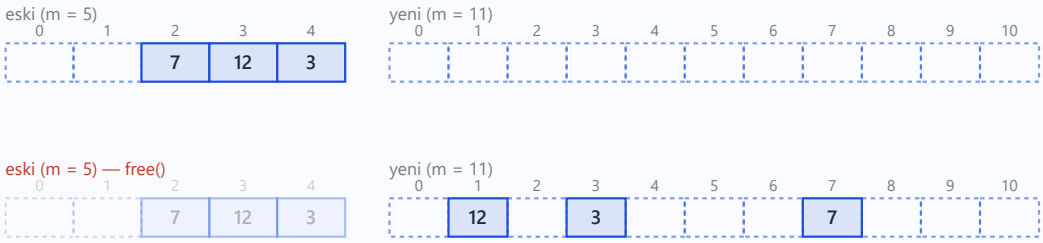
Fonksiyonun bir inceliği `tablo_ekle` ile ilişkisidir. `tablo_ekle` büyütme denetimini kendisi yapsaydı, yeniden hash'leme sırasında tekrar büyütme tetiklenebilirdi. Yeni boyut eski boyutun iki katından büyük ve `n` sıfırdan başladığı için bu kodda eşik aşılmaz. Yine de denetimin `tablo_ekle` 'nin dışında, çağıran tarafta durması daha güvenlidir. Sunumdaki düzen de budur.

14.5.5 Kod izleme: `yeniden_boyutla`

5 yuvalı tabloya 7, 12 ve 3 eklenmiştir. Doğrusal yoklamayla 7 yuva 2'ye, 12 yuva 3'e (ev adresi 2 dolu), 3 yuva 4'e (ev adresi 3 dolu) yerleşmiştir. *n = 3*, *m = 5*, *α = 0,6 > 0,5*: tablo büyütülecektir. Yeni boyut `sonraki_asal(10)` = 11'dir. Tablo 14.16 izlemeyi verir.

Adım	i	Olay	t->n
1	—	`eski` = eski dizi, `eski_m` = 5	3
2	—	`calloc(11, …)` : 11 yuva, hepsi `BOS` ; `t->a` yeni dizi, `t->m = 11`	0
3	0, 1	`eski[0]` , `eski[1]` `BOS` : atlanır	0
4	2	7 `DOLU` : *7 mod 11 = 7* → yuva 7 (eski ev adresi *7 mod 5 = 2*)	1
5	3	12 `DOLU` : *12 mod 11 = 1* → yuva 1 (eski ev adresi 2)	2
6	4	3 `DOLU` : *3 mod 11 = 3* → yuva 3 (eski ev adresi 3)	3
7	—	`free(eski)` ; `return true` ; yeni $\alpha = 3/11 \approx 0{,}27$	3

Tablo 14.16: `yeniden_boyutla(t, 11)` izlemesi. Eski tablo: yuva 2, 3, 4 = 7, 12, 3.



Şekil 14.15: `yeniden_boyutla(t, 11)` . Üstte `calloc` 11 boş yuva verdi; eski tabloda 7, 12 ve 3 duruyor. Altta üç öğe yeni ev adreslerine yerleşti (12 → 1, 3 → 3, 7 → 7) ve eski dizi `free` edildi.

Eski tabloda üç öğe bitişik bir blok oluşturuyordu ve 12 ile 3 ev adreslerinden kaymıştı. Yeni tabloda üçü de kendi ev adresindedir; çakışma yoktur. Büyütme, yük faktörünü düşürmenin yanında kümeleri de dağıtmıştır. Test programı aynı sonucu iki yoldan aldı: `yeniden_boyutla` doğrudan çağrılınca ve 7, 12, 3 eşik denetimli eklemeyle sırayla eklenince. İkinci yolda büyütme tam üçüncü ekleme-den sonra tetiklendi. Sayaçlı sarmalayıcı dört ayırma (iki başlangıç dizisi ve iki yeni dizi) ve dört bırakma gösterdi.

Tek büyütmenin maliyeti. `calloc` yeni diziyi sıfırlarken m' yuvaya dokunur, döngü eski m yuvada gezinir ve n öğeyi yeniden ekler. Toplam $O(n + m)$'dir. Tek bir ekleme bu yüzden bazen çok pahalı olur.

Teorem 14.3 (Büyütmenin amortize maliyeti)

Tablo, eşik aşıldığında boyutu iki katına (ya da daha fazlasına) çıkarılarak büyütülürse, boş bir tabloya yapılan n eklemede yeniden eklenen öğelerin toplam sayısı $2n$'den azdır. Ekleme başına amortize maliyet $O(1)$'dir.

Kanıt. Son büyütme, öğe sayısı $n_s \le n$ iken yapılmış olsun ve bu büyütmede n_s öğe yeniden eklensin. Bir önceki büyütme, en geç tablo boyutu bunun yarısıyken tetiklenmiştir; o anda en çok $n_s/2$ öğe vardı. Ondan öncekinde en çok $n_s/4$, ve bu böyle sürer. Toplam yeniden ekleme sayısı en çok $n_s + n_s/2 + n_s/4 + \ldots < 2n_s \le 2n$'dir. Her büyütmenin `calloc` maliyeti de tablo boyutuyla orantılıdır ve boyutlar da her seferinde en az iki katına çıktığı için aynı geometrik toplamla sınırlıdır. n eklemenin toplam maliyeti $O(n)$, ekleme başına $O(1)$ olur. ■

Bu türden bir analize **amortize analiz** (*amortized analysis*) denir: tek tek işlemlerin bazıları pahalı olsa da bir işlem dizisinin toplam maliyeti işlem sayısına bölünür. Teorem bir ölçümle denetlendi. 5 yuvadan başlayan tabloya bir milyon öğe eklendi ve $\alpha > 1/2$ olduğunda tablo bir sonraki asala büyütüldü. 19 büyütme oldu ve son boyut 3.292.489'a çıktı. Toplam yeniden ekleme sayısı 1.646.169'du, yani öğe başına 1,65: teoremdeki 2'nin altında. Büyütmelerde alınan yuvaların toplamı öğe başına 6,6 çıktı.

Sık yapılan hata

Tabloyu her seferinde sabit bir miktar, örneğin 100 yuva büyütmek. Bu durumda n eklemede yaklaşık $n/50$ büyütme olur ve her biri o anki n kadar öğe taşır. Toplam iş n^2 mertebesine çıkar ve ekleme başına amortize maliyet $O(n)$ olur. Büyütme çarpımsal olmalıdır.

14.6 Zincirleme mi, açık adresleme mi?

İki çakışma çözümü aynı sözlük işlemlerini sunar ve ikisi de α sabitken ortalama *O(1)*'dir. Aralarındaki seçim, verinin ve uygulamanın özelliklerine bağlıdır. Tablo 14.17 farkları yan yana koyar.

Ölçüt	Zincirleme	Açık adresleme
Yük faktörü	$\alpha > 1$ olabilir; tablo dolmaz, yavaşlar	$\alpha \le 1$; $\alpha \to 1$ iken maliyet patlar
Silme	basit; düğüm listeden çıkarılır	silindi işareti gerekir; işaretler birikir
Kötü hash fonksiyonu	daha dayanıklı; yalnız ilgili zincir uzar	kümeler komşu anahtarları da etkiler
Bellek	öğe başına işaretçi ve `malloc` yükü	ek işaretçi yok; bütün tablo bitişik
Önbellek	düğümler heap'te dağınık; önbellek dostu değil	bitişik dizi; özellikle doğrusal yoklama önbellek dostu
Yoklama seçimi	gerekmez	kümelenmeye karşı iyi bir yoklama (çift hash) seçilmeli

Tablo 14.17: Zincirleme ile açık adreslemenin karşılaştırması.

Zincirlemenin en güçlü yanı esnekliğidir. Öğe sayısı önceden bilinmese bile tablo çalışmaya devam eder; yalnız zincirler uzar. Silme, bağlı listeden düğüm çıkarmak kadar basittir ve tablonun geri kalanını etkilemez. Kötü bir hash fonksiyonu bazı zincirleri uzatır, ama zarar o kovalarla sınırlı kalır. Bedeli bellek ve önbellektir. Test makinesinde 4 baytlık bir anahtar için 16 baytlık düğüm alındı. Düğümler heap'in farklı yerlerinde olduğu için zincirde her adım işlemcinin yeni bir bellek bloğu getirmesini gerektirebilir.

Açık adresleme bu iki noktada üstündür. Bütün öğeler tek bir dizide durur, işaretçi yoktur, `malloc` yalnız tablo büyütülürken çağrılır. Doğrusal yoklamanın ardışık yoklamaları aynı önbellek bloğuna düşer. Bedeli, yük faktörüne duyarlılık ve silmenin karmaşıklığıdır. α 0,5 civarında tutulursa tablonun yarısı boş kalır; zincirlemenin düğüm yüküyle karşılaştırıldığında bu çoğu zaman yine de ucuzdur.

Pratik bir kural şöyle özetlenebilir. Öğe sayısı öngörülemiyorsa, çok silme yapılıyorsa ya da hash fonksiyonunun kalitesinden emin olunamıyorsa zincirleme seçilir. Bellek ve hız kritikse, silme seyrekse ve yük faktörü düşük tutulabiliyorsa açık adresleme seçilir. Gerçek kitaplıklar iki yolu da kullanır. Java'nın `HashMap`'i zincirlemeye dayanır; Python'un `dict`'i açık adreslemeye dayanır.

14.7 Karmaşıklık özeti

Tablo 14.18 hash tablosu işlemlerinin maliyetini toplar. Ortalama sütunu, yük faktörünün sabit tutulduğunu ve hash fonksiyonunun anahtarları düzgün dağıttığını varsayar.

İşlem	Ortalama (α sabit)	En kötü	Açıklama
ara	*O(1)*	*O(n)*	bütün anahtarlar aynı kovaya düşerse
ekle	*O(1)*	*O(n)*	zincirde başa ekleme her zaman *O(1)*
sil	*O(1)*	*O(n)*	açık adreslemede silindi işareti
yeniden boyutlandır	*O(n + m)*	*O(n + m)*	ekleme başına amortize *O(1)*
sıralı gezinme	*O(m + n lg n)*	*O(m + n lg n)*	hash sıra tutmaz; öğeler toplanıp sıralanır
en küçük / en büyük	*O(n + m)*	*O(n + m)*	bütün tablo taranır

Tablo 14.18: Hash tablosu işlemlerinin karmaşıklığı (*n* öğe, *m* kova).

Son iki satır, hash tablosunun dengeli arama ağacına göre zayıf kaldığı yerdir. Arama ağacı en küçük öğeyi *O(lg n)*'de bulur ve öğeleri *O(n)*'de sıralı gezer. Hash tablosunda öğeler kovalara dağınık durduğu için en küçük öğe için bütün kovalara bakılır. Sıralı gezinme için önce bütün öğeler bir diziye toplanır, sonra sıralanır.

En kötü durum yalnız şanssızlıkla gelmez; kasıtlı olarak da üretilebilir. Hash fonksiyonu herkesçe biliniyorsa bir saldırgan, hepsi aynı kovaya düşen binlerce anahtar hazırlayabilir. Örneğin bir web sunucusu istek parametrelerini hash tablosunda tutuyorsa, aynı kovaya düşen parametre adlarıyla dolu tek bir istek, sunucuyu $O(n^2)$ işe sokar. Bu saldırıya **hash taşkını** (*hash flooding*) denir. Önlem, hash fonksiyonunu program başlarken rastgele seçmektir. Fonksiyon, önceden tanımlanmış bir aileden rastgele seçilir. Saldırgan hangi fonksiyonun seçil-

diğini bilmediği için çakışan anahtar hazırlayamaz. Ailenin herhangi iki farklı anahtarı çakıştırma olasılığı en çok *1/m* ise aileye **evrensel** denir. Bu yönteme **evrensel hash** (*universal hashing*) adı verilir (CLRS, Kesim 11.3.3). Python ve Rust gibi dillerin dizgi hash'leri bu nedenle her çalıştırmada rastgele bir anahtarla başlar.

14.8 Sık yapılan hatalar

Hash tablosu kodlarında en sık görülen altı hata şunlardır. Her biri bu notta bir örnekle karşılaşılan bir durumdur.

1. **Negatif anahtarda `k % m`'nin negatif çıkması.** C'de `-7 % 11` −7'dir. Negatif indis dizinin dışına yazar. Anahtar `unsigned` tutulmalı ya da kalan `h_guvenli`'deki gibi düzeltilmelidir.
2. ***m*'yi 10 ya da 2'nin kuvveti seçmek.** Bölme yöntemiyle bu seçim yalnız son basamaklara ya da son bitlere bakar. 10'un katları *m = 10*'da tek kovaya toplanır. *m*, 2'nin kuvvetlerinden uzak bir asal olmalıdır.
3. **Açık adreslemede silinen yuvayı `BOS` yapmak.** Silinen yuvadan geçerek yerleşmiş anahtarların aramaları yarıda kalır. Yuva `SILINDI` ile işaretlenmelidir.
4. **Tablo doluyken sonsuz döngüye girmek.** Yoklama döngüsü boş yuva bulana kadar dönerse dolu tabloda hiç durmaz. Deneme sayısı *m* ile sınırlanmalıdır. Karesel yoklamada *m* denemede bile bütün yuvalar yoklanmaz; tablo yarıdan fazla doldurulmamalıdır.
5. **`malloc` sonucunu denetlememek; `free` sonrası düğümü kullanmak.** `NULL` dönen `malloc`'un sonucuna yazmak programı çökertir. Silinen düğümün `sonraki` alanını `free`'den sonra okumak tanımsız davranıştır; adres önce saklanmalıdır.
6. **Yeniden boyutlandırmada öğeleri eski indisleriyle kopyalamak.** Ev adresi *m*'ye bağlıdır. Eski konumuna kopyalanan öğe yeni tabloda bulunamaz. Her öğe yeni tabloya yeniden eklenmelidir.

14.9 Alıştırmalar ve çözümleri

Sunumdaki altı alıştırmanın çözümleri aşağıda ara adımlarıyla verilmiştir. Bütün sonuçlar bir doğrulama betiğiyle yeniden hesaplandı.

14.9.1 Doğrusal yoklamayla küçük tablo

Soru. *m = 7*, *h(k) = k mod 7*, doğrusal yoklama: 10, 17, 24, 3 sırayla eklenince tablo ne olur?

Çözüm. Dört anahtarın ev adresleri *10 mod 7 = 3*, *17 mod 7 = 3*, *24 mod 7 = 3* ve *3 mod 7 = 3*'tür: hepsi eş anlamlıdır. 10, yuva 3'e yazılır. 17 için yuva 3 doludur, yuva 4'e yazılır. 24 için yuva 3 ve 4 doludur, yuva 5'e yazılır. 3 için yuva 3, 4 ve 5 doludur, yuva 6'ya yazılır. Son tablo: yuva 3–6 = 10, 17, 24, 3. Yoklama sayıları 1, 2, 3 ve 4'tür; toplam 10.

14.9.2 Aynı anahtarlar zincirlemeyle

Soru. Aynı anahtarlar zincirlemeyle (başa ekleme) yerleştirilirse en uzun zincir kaç düğümdür?

Çözüm. Dört anahtarın hepsi kova 3'e düşer. Başa ekleme son ekleneni başa koyduğu için zincir 3 → 24 → 17 → 10 olur: dört düğüm. Öteki altı kova boştur. Hash fonksiyonu bu dört anahtarı hiç dağıtamamıştır; 7'nin katlarından 3 fazla olan sayılar için *k mod 7* kötü bir seçimdir. Zincirleme bu kötü dağılımı tek bir kovaya hapseder, ama o kovadaki arama yine dört düğüm sürer.

14.9.3 Karesel yoklama dizisi

Soru. *m = 11* ve karesel yoklamada 0, 1, 4, 9, 16, 25 (mod 11) yoklama dizisini yazmak.

Çözüm. İlk dört değer 11'den küçüktür ve olduğu gibi kalır: 0, 1, 4, 9. *16 mod 11 = 5* ve *25 mod 11 = 3*'tür. Dizi 0, 1, 4, 9, 5, 3'tür. Altı değerin hepsi farklıdır; bu, karesel yoklama teoremindeki *⌈11/2⌉ = 6* sınırıyla uyumludur. Yedinci deneme *36 mod 11 = 3*'tür ve tekrar başlar.

14.9.4 Dolu tabloda başarısız arama

Soru. *α = 0,9* olan doğrusal yoklamalı tabloda başarısız arama kaç yoklama bekler?

Çözüm. Knuth'un formülüyle *½ (1 + 1/(1 − 0,9)²) = ½ (1 + 1/0,01) = ½ · 101 = 50,5* yoklama. Ölçümde 51,1 bulundu. Aynı doluluktaki çift hash için beklenen değer 10, zincirleme için 1,9'dur. *α → 1* iken doğrusal yoklamanın maliyeti patlar.

14.9.5 Toplama ve polinom hash'i

Soru. “ab” ve “ba” için toplama hash'ini ve 31 tabanlı polinom hash'ini (mod 11) hesaplamak.

Çözüm. “a” = 97, “b” = 98'dir. Toplama: *97 + 98 = 195* ve *195 mod 11 = 8*; iki dizgi de 8. kovaya düşer. Polinom: “ab” için *97 · 31 + 98 = 3105* ve *3105 mod 11 = 3*; “ba” için *98 · 31 + 97 = 3135* ve *3135 mod 11 = 0*. Polinom hash'i iki anagramı ayırır. `dizgi_hash` aynı değerleri ara adımlarda mod alarak verir: “ab” için önce *97 mod 11 = 9*, sonra *(9 · 31 + 98) mod 11 = 377 mod 11 = 3*.

14.9.6 Eklemeden önce aramak

Soru. Açık adreslemede `ekle`, `SILINDI` yuvaya yazmadan önce neden `ara` çağırmalıdır?

Çözüm. Aynı anahtar, silindi işaretinden sonraki bir yuvada zaten bulunabilir. Silme kesimindeki örnekte 13 silindikten sonra 24, yuva 4'te durur; 24'ün yoklama dizisindeki ilk yuva ise silindi işaretli yuva 2'dir. Denetimsiz `ekle(24)`, 24'ü yuva 2'ye ikinci kez yazar. Önce `ara` çağrılırsa 24 yuva 4'te bulunur ve ekleme yapılmaz. `ekle_tekil` fonksiyonu bunu yapar.

14.10 Özet

- Hash tablosu anahtarı karşılaştırmaz, anahtardan konumu hesaplar. Arama, ekleme ve silme ortalama *O(1)*, en kötü *O(n)*'dir. Bedeli sıra bilgisinin kaybıdır: sıralı gezinme, en küçük/en büyük ve aralık sorguları ucuz değildir.
- Doğrudan adresleme anahtarı indis olarak kullanır; evren büyükse kurulamaz. Hash fonksiyonu *h: U → {0, …, m − 1}* anahtarı küçük bir aralığa indirir. *m < |U|* olduğundan çakışma kaçınılmazdır; doğum günü paradoksu çakışmanın yaklaşık $\sqrt{m}$ anahtardan sonra beklendiğini gösterir.
- İyi bir hash fonksiyonu hızlı, düzgün dağıtan, bütün bitleri kullanan ve deterministiktir. Bölme yönteminde *m*, 2'nin kuvvetlerinden uzak bir asal seçilir. Çarpma yönteminde *A ≈ 0,618* ve *m = 2^p* kullanılabilir. Uzun anahtarlar katlanır; dizgiler 31 tabanlı polinom hash'iyle Horner kuralına göre hesaplanır. C'de negatif `%` sonucuna dikkat edilmelidir.
- Açık adreslemede bütün öğeler tablodadır ve *α ≤ 1*'dir. Doğrusal yoklama birincil kümelenme, karesel yoklama ikincil kümelenme yaratır; çift hash anahtara özgü adım boyuyla kümelenmeyi en aza indirir. Karesel yoklamada *m* asal ve *α ≤ 1/2* olmalı, çift hashte h_2 sıfır olmamalı ve *m* ile aralarında asal olmalıdır.
- Açık adreslemede silinen yuva `BOS` yapılmaz, silindi işaretiyle işaretlenir. Arama işaretten geçer, ekleme işaretli yuvayı yeniden kullanır; tekrar eden anahtarı önlemek için eklemeden önce aranır.
- Zincirlemede her kova bir bağlı listenin başıdır. Başa ekleme *O(1)*, arama *Θ(1 + α)*'dır. Silme bağlı listeden düğüm çıkarmaktır; öncül işaretçisi ve baştaki düğüm durumu dikkat ister. Bütün düğümler sonunda `free` edilmelidir.
- Beklenen yoklama sayısı *1/(1 − α)* mertebesinde büyür. Eşik aşılınca tablo iki katının ilk asalına büyütülür ve bütün öğeler yeniden hash'lenir; amortize maliyet ekleme başına *O(1)*'dir.
- En kötü durum kötü niyetli girdilerle tetiklenebilir; evrensel hash, fonksiyonu çalışma anında rastgele seçerek buna karşı korur.

14.11 Kavram listesi

hash tablosu *hash table* · hash fonksiyonu *hash function* · doğrudan adresleme *direct addressing* · kova *bucket* · çakışma *collision* · yük faktörü *load factor* · sözlük *dictionary* · soyut veri tipi *abstract data type* · bölme yöntemi *division method* · çarpma yöntemi *multiplication method* · katlama *folding* · açık adresleme *open addressing* · yoklama *probe* · doğrusal yoklama *linear probing* · karesel yoklama *quadratic probing* · çift hash *double hashing* · kümelenme *primary clustering* · ayrı zincirleme *separate chaining* · silindi işareti *tombstone* · yeniden boyutlandırma *rehashing* · amortize analiz *amortized analysis* · önbellek *cache* · sarkan işaretçi *dangling pointer* · bellek sızıntısı *memory leak*

14.12 Kendini Dene

1. *m = 11*, *h(k) = k mod 11* ve doğrusal yoklamayla 20, 31, 42, 9, 53 sırayla ekleniyor. Son tablo nedir? 53 kaç yoklamada yerleşir?

> **Cevap**
>
> Beş anahtarın hepsinin ev adresi 9'dur. 20 → 9, 31 → 10, 42 → 9, 10 dolu, başa dönülür → 0, 9 → 9, 10, 0 dolu → 1, 53 → 9, 10, 0, 1 dolu → 2. Tablo: yuva 0 = 42, 1 = 9, 2 = 53, 9 = 20, 10 = 31. 53 beş yoklamada yerleşir.

2. Aynı anahtarlar *m = 11* kovalı zincirli tabloya başa eklemeyle konuyor. Kova 9'un zinciri nasıldır? `zincir_ara(20)` kaç düğüme bakar?

> **Cevap**
>
> Zincir 53 → 9 → 42 → 31 → 20'dir; son eklenen baştadır. 20 zincirin sonunda olduğu için arama beş düğüme bakar.

3. 16, 32, 48 ve 64 anahtarları *m = 16* ve *m = 17* ile bölme yöntemine sokuluyor. Ev adresleri nedir? Sonuç neyi gösterir?

> **Cevap**
>
> *m = 16* ile dördü de 0'a düşer, çünkü *k mod 16* yalnız son dört biti alır ve bu anahtarların son dört biti sıfırdır. *m = 17* ile ev adresleri 16, 15, 14 ve 13'tür; dördü de farklıdır. 2'nin kuvveti olan tablo boyutu örüntülü veride kümelenme yaratır, asal boyut yaratmaz.

4. `dizgi_hash("ali", 7)` kaç döndürür? (a = 97, l = 108, i = 105)

> **Cevap**
>
> 1. tur: *97 mod 7 = 6*. 2. tur: *6 · 31 + 108 = 294*, *294 mod 7 = 0*. 3. tur: *0 · 31 + 105 = 105*, *105 mod 7 = 0*. Sonuç 0.

5. Doğrusal yoklamalı tabloda yuva 0–5 = 22, 1, 13, 11, 24, 33'tür. `sil(22)` çağrılıyor. Ardından `ara(33)` kaç yoklama yapar? Yuva 0 `BOS` yapılsaydı ne olurdu?

> **Cevap**
>
> Yuva 0 `SILINDI` olur. `ara(33)` yuva 0'daki işaretten geçer, 1, 2, 3 ve 4'te farklı anahtarlar bulur ve yuva 5'te 33'ü bulur: 6 yoklama. Yuva 0 `BOS` yapılsaydı arama ilk yoklamada durur ve −1 döndürürdü; 33 tabloda olduğu hâlde bulunamazdı.

6. Zincirli bir tabloda *n = 2000*, *m = 1000*; açık adresli bir tabloda ise *α = 0,8* ve doğrusal yoklama kullanılıyor. Her birinde başarılı ve başarısız arama ortalama kaç yoklama sürer?

> **Cevap**
>
> Zincirlemede *α = 2*: başarılı *1 + α/2 = 2*, başarısız *1 + α = 3*. Doğrusal yoklamada başarılı *½ (1 + 1/0,2) = 3*, başarısız *½ (1 + 1/0,04) = 13*. Zincirli tablo iki kat dolu olduğu hâlde daha hızlıdır.

7. 11 yuvalı boş bir tabloya sırayla anahtar ekleniyor; her eklemeden sonra `2 * n > m` ise tablo `sonraki_asal(2 * m)` boyutuna büyütülüyor. İlk iki büyütme kaçıncı eklemeden sonra olur ve yeni boyutlar nedir?

> **Cevap**
>
> İlk büyütme 6. eklemeden sonra olur (*12 > 11*); yeni boyut `sonraki_asal(22)` = 23. İkincisi 12. eklemeden sonra olur (*24 > 23*); yeni boyut `sonraki_asal(46)` = 47.

8. Çift hash *m = 10* ile kullanılıyor ve bir anahtar için $h_1 = 5$, $h_2 = 6$. Yoklama dizisi nedir ve neden sorunludur?

> **Cevap**
>
> Dizi *(5 + 6i) mod 10*: 5, 1, 7, 3, 9, 5, 1, … Yalnız beş yuva yoklanır, çünkü 6 ile 10'un ortak böleni 2'dir. Tabloda boş yer olsa bile bu anahtar onu bulamayabilir. *m* asal seçilirse sorun ortadan kalkar.

9. Kova 2'nin zinciri 22 → 7 → 12 iken `zincir_sil(22)` çağrılıyor. Döngü kaç tur döner, hangi işaretçi güncellenir?

   Cevap

   Döngü hiç dönmez: `p` ilk düğümdür ve anahtarı 22'dir. `onceki` `NULL` kalır; bu yüzden `kova[2] = p->sonraki` yapılır ve kova artık 7'nin düğümünü gösterir. Ardından 22'nin düğümü `free` edilir; zincir 7 → 12 olur.

10. Anahtar `int k = -15` ve $m = 11$. C'de `k % m` kaçtır? Doğru ev adresi nedir?

    Cevap

    C'de `-15 % 11` −4'tür; bu indis dizinin dışındadır. Doğru ev adresi $-4 + 11 = 7$'dir (`h_guvenli(-15, 11)`).

14.13 İleri okuma

- E. Horowitz, S. Sahni ve S. Anderson-Freed, *Fundamentals of Data Structures in C* (2. baskı), Bölüm 8: statik ve dinamik hash, bölme, orta-kare ve katlama yöntemleri, doğrusal ve karesel yoklama ile zincirlemenin bu notun izlediği kaynağıdır.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (3. baskı), Bölüm 11: doğrudan adresleme, zincirleme analizi, bölme ve çarpma yöntemleri, evrensel hash ve açık adreslemenin beklenen yoklama sayısının kanıtlarını verir.
- R. Sedgewick, *Algorithms in C, Parts 1–4* (3. baskı), Bölüm 14: hash fonksiyonları, zincirleme, doğrusal yoklama ve çift hashi C kodlarıyla ve deneysel karşılaştırmalarla işler.
- D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching* (2. baskı), Kesim 6.4: doğrusal yoklamanın $\tfrac{1}{2}\left(1 + 1/(1-\alpha)\right)$ ve $\tfrac{1}{2}\left(1 + 1/(1-\alpha)^2\right)$ formüllerinin özgün çözümlemesini ve çarpma yöntemindeki altın oran seçiminin gerekçesini içerir.

14.14 Kaynaklar

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, Bölüm 8 (Hashing).
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3. baskı, MIT Press, Bölüm 11 (Hash Tables).
3. R. Sedgewick, *Algorithms in C, Parts 1–4*, 3. baskı, Addison-Wesley, Bölüm 14 (Hashing).
4. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2. baskı, Addison-Wesley, 1998, Kesim 6.4.

KISIM IV · BÖLÜM 15

Çizgeler

Çizge, nesneleri ve aralarındaki bağlantıları bir arada tutan en genel veri yapısıdır. Bağlı liste ve ağaç onun özel durumlarıdır. Bu not Euler'in Königsberg köprüleri problemiyle başlar. Ardından düğüm, kenar, yol, derece ve bağlantılılık gibi temel kavramları tanımlar ve derece toplamı teoremini kanıtlar. Notun ağırlık merkezi çizgenin bellekte nasıl tutulacağıdır. Komşuluk matrisi, komşuluk listesi ve tersi, tek diziyle sıralı gösterim, ortogonal liste ve çok bağlı liste C koduyla kurulur; bellek ve işlem maliyetleri karşılaştırılır. Sonra çizge üzerindeki temel işlem olan gezinme gelir. Derinlik öncelikli arama özyinelemeyle, genişlik öncelikli arama dinamik bir kuyrukla yazılır ve adım adım izlenir. Son kesimde bu iki gezinme bağlantılı bileşenleri bulmak, döngü aramak ve çizgenin iki parçalı olup olmadığını sınamak için kullanılır. Her yordamın satır satır anlatımı, izleme tablosu ve karmaşıklık analizi verilir.

Kod (C): `github.com/sercankulcu/data-structures-and-algorithms-c/Bolum15`

ÖĞRENME ÇIKTILARI

Tanımlamak
G = (V, E), yönlü ve yönsüz çizge, komşuluk, alt çizge, yol, derece ve bağlantılılık kavramlarını doğru kullanmak.

Kanıtlamak
Derecelerin toplamının 2e olduğunu ve bağlantılı bir çizgede en az n − 1 kenar bulunduğunu gerekçelendirmek.

Gerçeklemek
Komşuluk matrisini, komşuluk listesini, sıralı diziyi, ortogonal listeyi ve çok bağlı listeyi C ile kurmak; bellekteki görünümlerini çizmek.

Gezinmek
Derinlik öncelikli (DFS) ve genişlik öncelikli (BFS) aramayı kodlamak, elle izlemek ve O(n + e) analizini yapmak.

Uygulamak
Bağlantılı bileşenleri, döngüleri ve iki parçalılığı DFS ya da BFS ile bulmak.

Seçmek
Problemin işlemlerine bakarak uygun gösterimi seçmek: önce veri yapısı, sonra kod.

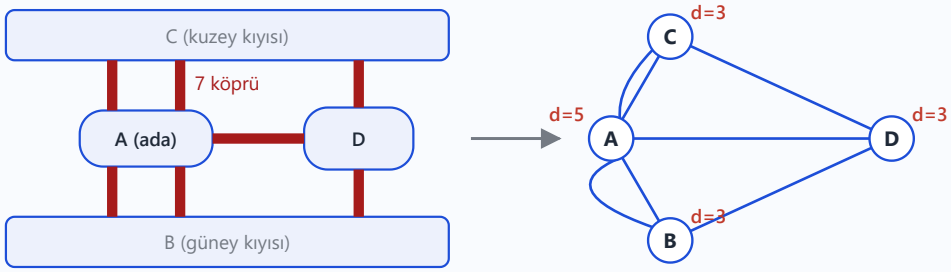
15.1 Giriş: köprüler, haritalar ve ağlar

Şimdiye kadarki veri yapılarında öğeler arasındaki ilişkinin bir biçimi vardı. Dizide ve bağlı listede her öğenin en çok bir öncesi ve bir sonrası bulunur. Ağaçta her düğümün tek bir ebeveyni vardır; çocuklar yukarıya dönmez. Gerçek dünyadaki pek çok ilişki bu kalıplara uymaz. Bir şehirden birçok yol çıkar ve bu yollar birbirine karışarak başka şehirlerde birleşir. Bir ağdaki bilgisayar birden çok bilgisayara bağlıdır; bağlantılar halkalar oluşturabilir. Böyle ilişkileri hiçbir kısıt koymadan tutan yapıya **çizge** (*graph*) denir.

15.1.1 Königsberg'in yedi köprüsü

Çizge kuramının başlangıcı olarak 1736 yılı kabul edilir. Doğu Prusya'nın Königsberg şehri Pregel nehrinin iki yakasına kurulmuştu. Nehrin ortasında iki ada vardı. Şekil 15.1'in solundaki krokide adalar A ve D, kıyılar B (güney) ve C (kuzey) harfleriyle gösterilir. Bu dört kara parçası yedi köprüyle birbirine bağlanıyordu: A'dan C'ye iki, A'dan B'ye iki, A'dan D'ye bir, D'den C'ye bir ve D'den B'ye bir köprü.

Şehir halkının sorusu şuydu: Herhangi bir kara parçasından yola çıkıp her köprüden *tam olarak bir kez* geçerek başlanan yere dönülebilir mi? Deneme yanılmayla bir cevap bulunamıyordu. Leonhard Euler (“Oyler” diye okunur) soruyu çözmek için haritanın ayrıntılarını bir kenara bıraktı. Her kara parçasını bir nokta, her köprüyü iki noktayı birleştiren bir çizgi olarak çizdi. Noktalara bugün **düğüm** (*vertex, node*), çizgilere **kenar** (*edge*) denir. Ortaya çıkan çizim, şeklin sağındaki çizgedir. A ile C arasında iki ayrı kenar vardır, çünkü iki ayrı köprü vardır.



Şekil 15.1: Königsberg köprüleri (solda) ve onlardan elde edilen çizge (sağda). Kırmızı şeritler köprülerdir. Her düğümün yanında derecesi yazar: A'nın derecesi 5, öteki üç düğümünki 3'tür.

Bu soyutlamanın önemi şudur: Soruyu belirleyen, kara parçalarının büyüklüğü, köprülerin uzunluğu ya da nehrin kıvrımları değildir. Yalnız hangi parçanın hangisine kaç köprüyle bağlı olduğu önemlidir. Çizgede düğümlerin kâğıt üzerindeki yeri de önemsizdir; A düğümü sola ya da sağa çizilebilir. Aynı çizge birçok farklı biçimde çizilebilir. Çizim bir gösterimdir, çizgenin kendisi bağlantılardan ibarettir.

Euler soruyu bir sayıya bağladı. Bir düğüme değen kenarların sayısına o düğümün **derecesi** (*degree*) denir. Yürüyüş sırasında bir ara düğüme her girişte o düğümden bir de çıkılır. Giriş ve çıkış farklı köprülerden yapılmak zorundadır. Öyleyse her uğrayışta düğümün iki kenarı harcanır. Başlangıç düğümü de aynı durumdadır: yürüyüş ondan çıkar ve sonunda ona girer. Bütün köprüleri birer kez kullanan ve başa dönen bir yürüyüşte her düğümün kenarları ikişer ikişer tükenmelidir. Bu nedenle böyle bir yürüyüş ancak *her düğümün derecesi çiftse* bulunabilir. Königsberg'de dereceler 5, 3, 3 ve 3'tür; dördü de tek sayıdır. Aranan yürüyüş yoktur.

Euler'in vardığı sonuç bugün onun adıyla anılır. Bağlantılı bir çizgede her kenardan bir kez geçip başa dönen yürüyüşe **Euler devresi** denir. Böyle bir devre, ancak ve ancak bütün dereceler çiftse vardır. Koşulun gerekli olduğu yukarıdaki sayma kanıtından çıkar. Yeterli olduğunun kanıtı daha uzundur ve çizge kuramı derslerinin konusudur. Bu notta önemli olan, bir şehir haritası sorusunun yalnızca sayılar ve bağlantılar üzerinden cevaplanabilmesidir.

15.1.2 Çizgeler nerelerde kullanılır

Königsberg'den bu yana çizgeler hemen her mühendislik alanına girdi. Kullanımları o kadar geniştir ki çizgenin kullanılmadığı bir alan bulmak, kullanıldığı alanları saymaktan zordur. Tablo 15.1 başlıca örnekleri toplar.

Alan	Düğüm	Kenar	Tipik soru
Elektrik devreleri	Bağlantı noktası	Devre elemanı	Akım ve gerilimler nedir?
Proje planlama (aktivite ağları)	Olay ya da iş	Öncelik ilişkisi	Proje en erken ne zaman biter?
Kimya	Atom	Bağ	İki molekül aynı yapıda mı?
Yol ve bilgisayar ağları	Şehir, yönlendirici	Yol, hat	En kısa yol hangisi?
Taşıma ağları	Depo, pompa	Boru (kapasiteli)	Kaynaktan hedefe en çok ne kadar akış geçer?
Altyapı kurma	Köy	Olası kablo hattı	Bütün köyler en az kabloyla nasıl bağlanır?
Biçimsel diller	Durum	Geçiş	Sonlu özdevinir bu sözcüğü kabul eder mi?
Ağ güvenilirliği	Bilgisayar	Bağlantı	Ağda kopukluk var mı, kaç parçaya ayrılmış?

Tablo 15.1: Çizgelerin kullanıldığı başlıca alanlar.

Tablodaki soruların bir kısmı adlarıyla bilinen klasik problemlerdir. En kısa yol problemi bir haritada iki şehir arasındaki en ucuz güzergâhı arar. En büyük akış (*max flow*) problemi bir boru ağından birim zamanda en çok ne kadar petrol ya da su taşınabileceğini sorar. **Minimum kapsayan ağaç** (*minimum spanning tree*) problemi bütün köyleri birbirine en az toplam kablo uzunluğuyla bağlamak ister. Ağaçların kendileri de birer çizgedir; topolojik sıralama ve sonlu özdevinirler (*finite automata*) da çizge üzerinde tanımlanır. Bu problemlerin hepsi aynı temel üzerine kurulur: çizgenin bellekte nasıl tutulacağı ve üzerinde nasıl gezinileceği. Bu not tam olarak bu temeli kurar.

15.2 Tanımlar ve terimler

Çizgeyle ilgili her yordam birkaç temel kavrama dayanır. Bu kesim onları kesin biçimde tanımlar ve notun geri kalanında kullanılacak örnek çizgeleri tanıtır.

15.2.1 Çizgenin tanımı

Tanım 15.1

Bir **çizge** *G = (V, E)* iki sonlu kümeden oluşur. *V(G)*, boş olmayan **düğüm** kümesidir. *E(G)*, düğüm çiftlerinden oluşan ve boş olabilen **kenar** kümesidir. Düğüm sayısı *n* = |*V*|, kenar sayısı *e* = |*E*| ile gösterilir.

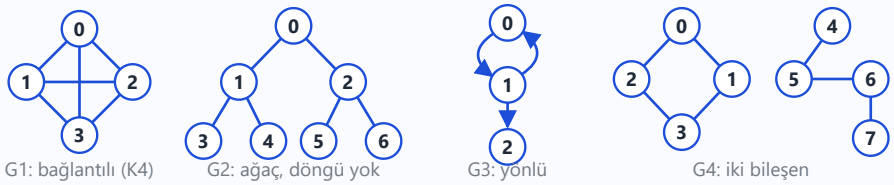
Düğüm kümesinin boş olmaması bir gerekliliktir: düğümsüz bir çizge üzerinde hiçbir işlem anlamlı değildir. Kenar kümesi ise boş olabilir. Tek bir düğümden oluşan ya da birbirine hiç bağlı olmayan düğümlerden oluşan çizgeler de geçerlidir. Aşağıdaki seyrek çizge örneğinde 11 düğümün 7'si hiçbir kenara değmez.

Kenarın iki ucunun sırası önemliyse çizge **yönlüdür** (*directed graph, digraph*); önemsizse **yönsüzdür** (*undirected graph*). Gösterimde bu fark parantez biçimiyle verilir:

- Yönsüz çizgede kenar sıradan bağımsız bir çifttir ve normal parantezle yazılır: *(v0, v1)* ile *(v1, v0)* aynı kenardır.
- Yönlü çizgede kenar sıralı bir çifttir ve açılı parantezle yazılır: *<v0, v1> ≠ <v1, v0>*. Okun çıktığı ilk düğüme **kuyruk** (*tail*), okun saplandığı ikinci düğüme **baş** (*head*) denir: *<kuyruk, baş>*.

Kenar kümesi, düğüm kümesinin kendisiyle kartezyen çarpımı *V* × *V*'nin bir alt kümesidir. Bu çarpım bütün sıralı ikilileri içerir; çizge bunlardan yalnız bazılarını seçer. Seçilen ikililer, hangi düğümün hangisine bağlı olduğunu söyler.

Not boyunca dört küçük örnek çizge kullanılır. Şekil 15.2 onları çizer, Tablo 15.2 de düğüm ve kenar kümelerini yazar.



Şekil 15.2: Örnek çizgeler. G1 dört düğümlü tam çizgedir. G2 yedi düğümlü bir ağaçtır. G3 üç düğümlü yönlü bir çizgedir; 0 ile 1 arasında iki yönde birer kenar vardır. G4 iki parçadan oluşur: {0, 1, 2, 3} ve {4, 5, 6, 7}.

Çizge	V(G)	E(G)	n	e
G1	{0, 1, 2, 3}	{(0,1), (0,2), (0,3), (1,2), (1,3), (2,3)}	4	6
G2	{0, 1, ..., 6}	{(0,1), (0,2), (1,3), (1,4), (2,5), (2,6)}	7	6
G3	{0, 1, 2}	{<0,1>, <1,0>, <1,2>}	3	3
G4	{0, 1, ..., 7}	{(0,1), (0,2), (1,3), (2,3), (4,5), (5,6), (6,7)}	8	7

Tablo 15.2: Örnek çizgelerin düğüm ve kenar kümeleri.

G3'ün kenar kümesinde *<0,1>* ile *<1,0>* ayrı ayrı yazılır. Yönlü çizgede bunlar iki farklı kenardır; birinin varlığı ötekini gerektirmez. Yönsüz bir çizgede ise *(0,1)* yazmak yeter; *(1,0)* aynı kenarın öteki yazılışıdır.

Bu notta aksi söylenmedikçe çizgeler **basittir**: iki düğüm arasında en çok bir kenar bulunur ve hiçbir kenar bir düğümü kendisine bağlamaz. Düğümü kendisine bağlayan kenara **öz döngü** (*self-loop*) denir. Königsberg çizgesi basit değildir; A ile C arasında iki kenar vardır. Böyle çizgelere çoklu çizge (*multigraph*) denir.

Sık yapılan hata

Yönlü bir çizgede *<u, v>* verilince *<v, u>*'yu da var saymak. G3'te *<1,2>* vardır ama *<2,1>* yoktur: 2'den hiçbir yere gidilemez. Yönsüz çizgede her kenar iki yönde de kullanılabilir; yönlüde yalnız oku izleyen yönde.

15.2.2 Komşu düğüm ve alt çizge

Tanım 15.2

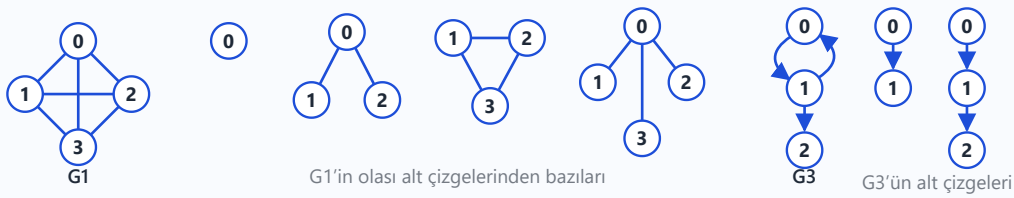
(v0, v1) yönsüz bir çizgenin kenarıysa *v0* ile *v1* **komşudur** (*adjacent*); kenar bu iki düğüme **bitişiktir** (*incident*). Yönlü çizgede *<v0, v1>* varsa *v0*, *v1*'e komşudur; *v1* de *v0*'dan komşudur.

G1'de her düğüm öteki üçüyle komşudur. G2'de 0'ın komşuları 1 ile 2'dir; 3 ile 4 birbirine komşu değildir, çünkü aralarında kenar yoktur. İkisi de 1'e komşudur, ama bu onları birbirine komşu yapmaz.

Tanım 15.3

G′ çizgesi $V(G') \subseteq V(G)$ ve $E(G') \subseteq E(G)$ koşullarını sağlıyorsa *G′*, *G*'nin bir **alt çizgesidir** (*subgraph*).

Tanımın gizli bir koşulu vardır: *G′* kendi başına bir çizge olmalıdır. Yani *E(G′)*'deki her kenarın iki ucu da *V(G′)*'de bulunmalıdır. Uçlarından biri seçilmemiş bir kenar alt çizgeye giremez. Kenar kümesi boş olabildiği için tek bir düğüm de geçerli bir alt çizgedir.



Şekil 15.3: Solda G1 ve onun dört alt çizgesi: tek düğüm 0; 0, 1, 2 düğümleri ve iki kenar; 1, 2, 3 üçgeni; 0'a bağlı üç kenar. Sağda G3 ve iki alt çizgesi: yalnız *<0,1>*; *<0,1>* ve *<1,2>*.

Bir çizgenin kaç alt çizgesi olduğu hızla büyür. G1'in yalnız kenar seçimleri için $2^6 = 64$ olasılık vardır, üstelik düğüm seçimleri de eklenir. Şekil bunlardan yalnız birkaçını gösterir.

15.2.3 Yol, bağlantılılık ve derece

Tanım 15.4

v_p düğümünden v_q düğümüne bir **yol** (*path*), (v_p, v_{i1}), (v_{i1}, v_{i2}), ..., (v_{in}, v_q) çiftlerinin her biri *E(G)*'de bulunacak biçimde dizilmiş $v_p, v_{i1}, \ldots, v_{in}, v_q$ düğüm dizisidir. Yolun **uzunluğu**, içerdiği kenar sayısıdır. Yönlü çizgede çiftler *<·,·>* biçiminde, yani okların yönünde olmalıdır.

G2'de 3, 1, 0, 2, 5 dizisi 3'ten 5'e uzunluğu 4 olan bir yoldur. Bütün düğümleri birbirinden farklı olan yola **basit yol** denir. İlk ve son düğümü aynı olan, başka bir düğümü yinelenmeyen ve en az bir kenar içeren yola **döngü** (*cycle*) denir. G1'de 0, 1, 2, 0 bir döngüdür. G2 bir ağaç olduğu için hiç döngü içermez. Yönsüz basit çizgede bir döngü en az üç kenar içerir; 0, 1, 0 dizisi aynı kenardan gidip geri döner ve döngü sayılmaz.

Tanım 15.5

Yönsüz bir çizgede *v0*'dan *v1*'e bir yol varsa bu iki düğüm **bağlantılıdır** (*connected*). Birbirinden farklı her (v_i, v_j) düğüm çifti bağlantılıysa çizgenin kendisi bağlantılıdır.

G1 ve G2 bağlantılıdır. G4 bağlantılı değildir: 0'dan 5'e hiçbir yol yoktur. G4 iki ayrı parçadan oluşur. Bu parçalara bağlantılı bileşen denir; bileşenler bu notun son kesiminde ayrıntılı olarak ele alınır.

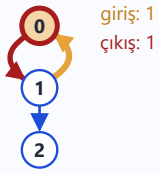
Tanım 15.6

Bir düğümün **derecesi** *d(v)*, o düğümü uç kabul eden kenarların sayısıdır. Yönlü çizgede düğüme gelen kenarların sayısı **giriş derecesi** (*in-degree*), düğümden çıkan kenarların sayısı **çıkış derecesi** (*out-degree*) adını alır. Düğümün derecesi bu ikisinin toplamıdır.

Yönsüz örneklerde G2'de *d(0) = 2*, G1'de *d(1) = 3*, G2'de *d(4) = 1*'dir. Yönlü G3'te her düğümün iki sayısı ayrı ayrı hesaplanır (Tablo 15.3).

Düğüm	Gelen kenarlar	Giriş	Çıkan kenarlar	Çıkış	Derece
0	<1,0>	1	<0,1>	1	1 + 1 = 2
1	<0,1>	1	<1,0>, <1,2>	2	1 + 2 = 3
2	<1,2>	1	—	0	1 + 0 = 1

Tablo 15.3: G3'te giriş ve çıkış dereceleri.



Şekil 15.4: G3'te 0 düğümü. Açık renkli ok 0'a gelen kenardır (giriş 1), koyu kırmızı ok 0'dan çıkan kenardır (çıkış 1). Düğüm 0'ın derecesi *1 + 1 = 2*'dir.

Derece her zaman belirli bir düğüm için söylenir. "G3'ün derecesi 2'dir" gibi bir cümlenin anlamı yoktur; doğru ifade "G3'te 0 düğümünün derecesi 2'dir" biçimindedir. Bir çizge için söylenebilecek olanlar en büyük derece ya da derecelerin toplamıdır.

15.2.4 Derecelerin toplamı

Derece ile kenar sayısı arasında basit ama çok kullanılan bir ilişki vardır.

Teorem 15.1 (Derece toplamı)

n düğümlü ve *e* kenarlı bir *G* çizgesinde, d_i ile *i*. düğümün derecesi gösterilirse

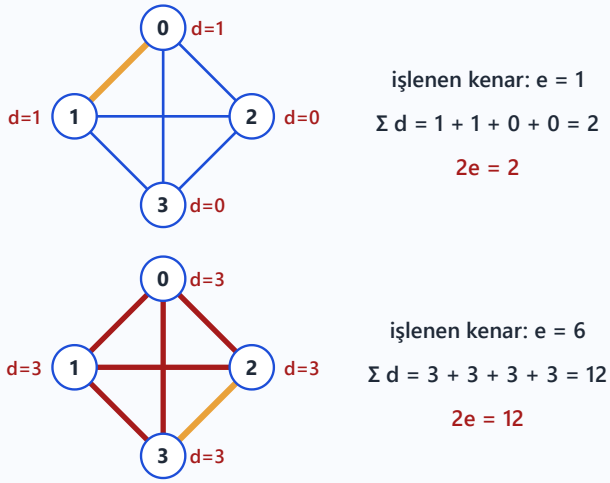
$$d_0 + d_1 + \dots + d_{n-1} = 2e$$

olur.

İspat. Derecelerin toplamı, her düğümün kendisine değen kenarları sayıp bu sayıların toplanmasıyla bulunur. Bu toplamda her kenar kaç kez sayılır? Bir *(u, v)* kenarı *u*'nun derecesinde bir kez, *v*'nin derecesinde bir kez sayılır. Başka hiçbir düğümün derecesine katılmaz. Öyleyse her kenar toplama tam olarak 2 katkı yapar ve toplam *2e*'dir. Yönlü çizgede de aynı akıl yürütme geçerlidir: *<u, v>*, *u*'nun çıkış derecesine ve *v*'nin giriş derecesine birer katkı yapar. ∎

Kanıt kenarların ayrı ayrı sayılmasına dayanır; iki düğüm arasında birden çok kenar olması bir şey değiştirmez. Teorem bu yüzden çoklu çizgelerde de geçerlidir. Königsberg çizgesinde *5 + 3 + 3 + 3 = 14 = 2 · 7*'dir. Öz döngüye izin verilirse böyle bir kenar düğümün derecesine 2 katkı yapacak biçimde sayılır ve teorem yine bozulmaz.

Şekil 15.5 teoremi G1 üzerinde adım adım gösterir. Kenarlar boş bir çizgeye birer birer eklenir. Her eklemede iki ucun derecesi birer artar ve toplam 2 büyür. Altı kenar eklendiğinde toplam 12'ye varır.



Şekil 15.5: Derece toplamı teoremi. Üstte ilk kenar (0,1) eklenmiştir: $\Sigma d = 2 = 2e$. Altta G1'in altı kenarının hepsi eklenmiştir: her düğümün derecesi 3, toplam $12 = 2 \cdot 6$.

Teoremin hemen çıkan bir sonucu vardır: Bir çizgede derecesi tek olan düğümlerin sayısı çifttir. Toplam *2e* çift olduğuna göre tek terimlerin sayısı tek olamaz. Bu sonuç "şu derecelere sahip bir çizge var mı?" sorusuna hızlı bir ön cevap verir.

Örnek 15.1

Derece dizisi (3, 3, 2, 2, 1) olan basit, yönsüz bir çizge var mıdır? Dereceler toplanır: *3 + 3 + 2 + 2 + 1 = 11*. Toplam tektir, ama teoreme göre *2e*'ye, yani bir çift sayıya eşit olmalıdır. Böyle bir çizge yoktur; kenar sayısı sorusu da anlamını yitirir. Dizi (3, 3, 2, 2, 2) olsaydı toplam 12, kenar sayısı 6 olurdu. Bu durumda teorem yalnız bir gerekli koşul verir; çizgenin gerçekten var olup olmadığını ayrıca kurarak göstermek gerekir. Örneğin 0–1–2–3–4–0 döngüsüne bir de (0,2) kenarı eklenirse dereceler (3, 2, 3, 2, 2) olur; düğümlerin adları değiştirilince dizi elde edilir.

Teoremin veri yapısı açısından önemi de büyüktür. Yönsüz bir çizgenin komşuluk listesinde her kenar iki kez, bir ucunun listesinde ve öteki ucunun listesinde, görünür. Listelerdeki toplam öğe sayısı derecelerin toplamıdır, yani *2e*'dir. Bütün listeleri tarayan bir yordamın maliyeti bu nedenle *O(n + e)* olur. Bu cümle aşağıdaki analizlerin neredeyse hepsinde yeniden karşılaşılan bir akıl yürütmedir.

15.3 Çizgeyi bellekte gerçeklemek

Kavramlar kâğıt üzerinde çizilerek anlaşılır, ama program çizimi göremez. Bir program için çizge, bellekteki sayılardan ve işaretçilerden ibarettir. Bu kesimin konusu ayrık matematik değildir; konusu, "çizgeyi bellekte nasıl tutarız?" sorusudur. Aynı çizge için birçok gösterim vardır ve aralarındaki seçim, çizge üzerinde yapılacak işlemlerin hızını belirler. Burada beş gösterim ele alınır:

1. komşuluk matrisi,
2. komşuluk listesi ve onun tersi,
3. tek dizi ile sıralı gösterim,
4. yönlü çizgeler için ortogonal liste,
5. çok bağlı liste.

Bunlara kenarlara maliyet ekleyen ağırlıklı gösterim eklenir. Her gösterim için üç soru sorulur: Ne kadar bellek ister? "*u* ile *v* arasında kenar var mı?" sorusu ne kadar sürede cevaplanır? Bir düğümün bütün komşuları ne kadar sürede bulunur? Gezinme gibi algoritmaların hızı en çok bu son soruya bağlıdır.

15.3.1 Komşuluk matrisi

Tanım 15.7

n düğümlü bir *G* çizgesinin **komşuluk matrisi** (*adjacency matrix*), *n* × *n* boyutlu iki boyutlu bir *a* dizisidir. Yönsüz çizgede *(i, j) ∈ E(G)*, yönlü çizgede *<i, j> ∈ E(G)* ise *a[i][j] = 1*, değilse *a[i][j] = 0*'dır.

C'de matris, en çok düğüm sayısına göre boyutlanmış bir tam sayı dizisiyle tutulur: `int a[MAX_DUGUM][MAX_DUGUM];`. Tablo 15.4 G1 ile G3'ün matrislerini yan yana verir.

	0	**1**	**2**	**3**		**G3**	**0**	**1**	**2**
0	0	1	1	1		**0**	0	1	0
1	1	0	1	1		**1**	1	0	1
2	1	1	0	1		**2**	0	0	0
3	1	1	1	0					

Tablo 15.4: Solda G1'in, sağda G3'ün komşuluk matrisi. Satır, kenarın kuyruğunu; sütun, başını gösterir.

G1'de *(0,2)* bir kenar olduğu için hem *a[0][2]* hem *a[2][0]* 1'dir. Yönsüz çizgenin her kenarı matrise iki kez, köşegene göre simetrik iki hücreye yazılır. Bu yüzden yönsüz çizgenin matrisi **simetriktir**: her *i, j* için *a[i][j] = a[j][i]*. Basit çizgede öz döngü olmadığından köşegen tamamen sıfırdır. G3'ün matrisi ise simetrik değildir: *a[1][2] = 1* ama *a[2][1] = 0*.

Simetri yer kazanmak için kullanılabilir. Matrisin yalnız alt ya da üst üçgeni saklanırsa hücre sayısı yaklaşık yarıya iner. Ama bellek yine de *n(n − 1)/2* hücreyle orantılıdır; asimptotik olarak $O(n^2)$ değişmez. Matrisin belleği kenar sayısına hiç bağlı değildir: hiç kenarı olmayan bir çizge de, tam çizge de aynı yeri kaplar. 50 düğümlük bir `int` matrisi, 4 baytlık `int` ile 10.000 bayt tutar.

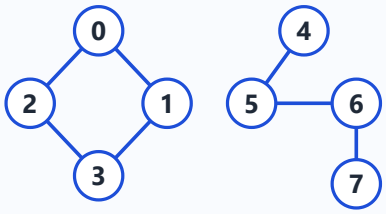
Matrisle iki işlem çok kolaydır. *(u, v)* kenarının varlığı tek bir hücreye bakılarak *O(1)*'de öğrenilir. Kenar eklemek ya da silmek de bir hücreye 1 ya da 0 yazmaktır, yine *O(1)*. Öteki işlemler satırları taramayı gerektirir:

- **Derece.** Yönsüz çizgede *i*'nin derecesi *i*. satırdaki 1'lerin sayısıdır (ya da aynı anlama gelen *i*. sütundakilerin). Satırda *n* hücre olduğu için maliyet *O(n)*'dir. Yönlü çizgede satır toplamı çıkış derecesini, sütun toplamı giriş derecesini verir; ikisi birlikte *2n* hücre okur ve maliyet yine *O(n)* olur. G3'te 2'nin çıkış derecesi 0 (satır 2 boş), giriş derecesi 1'dir (sütun 2'de bir tane 1).
- **Kenar sayısı.** Bütün 1'ler sayılıp ikiye bölünür (Teorem 15.1). G1'de 12 tane 1 vardır; kenar sayısı 6'dır. Simetri yüzünden yalnız üst üçgene bakmak yeter, ama bu *n(n − 1)/2* hücre demektir. Maliyet $O(n^2)$'den aşağı inmez.
- **Bir düğümün komşuları.** Satırın *n* hücresi taranır: *O(n)*. Düğümün tek komşusu olsa bile bütün satır okunur.

Son madde gezinme algoritmalarının maliyetini belirler ve matrisin asıl zayıflığıdır. Seyrek bir çizgede satırların çoğu sıfırdır, ama sıfırları atlamanın bir yolu yoktur.

G4'ün matrisi ve simetri denetimi

Matris elle yazılırken en sık hata, bir kenarı yalnız tek yönde işlemektir. Şekil 15.6 G4'ü, Tablo 15.5 doğru matrisini verir.



Şekil 15.6: G4 çizgesi. İki bileşenden oluşur: 0, 1, 2, 3 düğümlerinin oluşturduğu dörtgen ve 4–5–6–7 zinciri.

	0	1	2	3	4	5	6	7
0	0	1	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0
2	1	0	0	1	0	0	0	0
3	0	1	1	0	0	0	0	0
4	0	0	0	0	0	1	0	0
5	0	0	0	0	1	0	1	0
6	0	0	0	0	0	1	0	1
7	0	0	0	0	0	0	1	0

Tablo 15.5: G4'ün komşuluk matrisi. Matris simetriktir; iki bileşen köşegen üzerinde iki ayrı blok oluşturur.

Sık yapılan hata

G4'ün 1. satırını `1 0 1 0 0 0 0 0` , 7. satırını `… 0 0 1` biçiminde yazmak. Birinci satırdaki fazladan 1, var olmayan (1,2) kenarını ekler; yedinci satırdaki 1 ise 7'yi kendine bağlar. İkisi de simetriyi bozar: *a[1][2] = 1* iken *a[2][1] = 0* kalır. Doğrusu 1. satır için `1 0 0 1 0 0 0 0` (komşular 0 ve 3), 7. satır için `0 0 0 0 0 0 1 0` 'dır (tek komşu 6). Yazılan matrisin simetrisini *a[i][j] == a[j][i]* karşılaştırmasıyla sınamak bu hatayı hemen yakalar.

Matris bağlantılı bileşenleri bulmak için de kullanılabilir. Bunun için bir düğümden başlayıp ulaşılabilen bütün düğümler işaretlenir; işaretsiz kalan ilk düğümden yeniden başlanır. Ama her düğümün komşularını bulmak bir satır taraması, yani *O(n)* olduğundan toplam maliyet $O(n^2)$'dir. Aynı iş komşuluk listesiyle çok daha ucuzdur.

Matriste derece: kod ve izleme

Yönlü bir çizgede bir düğümün derecesini hesaplayan yordam, satırı ve sütunu ayrı ayrı toplar:

```
int derece(int a[][MAX_DUGUM], int n, int v)
{
    int j, cikis = 0, giris = 0;
    for (j = 0; j < n; j++)
        cikis += a[v][j];      /* satir  */
    for (j = 0; j < n; j++)
        giris += a[j][v];      /* sutun */
    return cikis + giris;
}
```

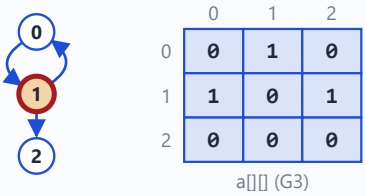
- **Satır 1.** İki boyutlu dizi parametre olarak geçirilirken ikinci boyut yazılmak zorundadır: `int a[][MAX_DUGUM]` . Derleyici *a[v][j]*'nin adresini *v · MAX_DUGUM + j* ile hesaplar ve bunun için satır uzunluğunu bilmelidir. Birinci boyut boş bırakılabilir. `n` gerçek düğüm sayısıdır; dizinin yalnız sol üst *n × n* bölümü kullanılır.
- **Satır 3.** İki sayaç sıfırlanır. Başlangıç değeri unutulursa yerel değişkenler belirsiz değerle başlar ve sonuç rastgele olur.
- **Satır 4–5.** `v` . satır taranır: `a[v][j]` , *<v, j>* kenarı varsa 1'dir. Toplam çıkış derecesidir.
- **Satır 6–7.** `v` . sütun taranır: `a[j][v]` , *<j, v>* kenarı varsa 1'dir. Toplam giriş derecesidir.
- **Satır 8.** İkisinin toplamı döndürülür.

Yordam yönsüz bir çizgeye uygulanırsa satır ve sütun aynı olduğundan her kenar iki kez sayılır ve derecenin iki katı döner. Yönsüz çizgede yalnız ilk döngü yeter. Tablo 15.6 `derece(a, 3, 1)` çağrısını G3 matrisi üzerinde izler.

Adım	Satır	`j`	Okunan hücre	`cikis`	`giris`
1	3	—	—	0	0
2	4–5	0	a[1][0] = 1	1	0
3	4–5	1	a[1][1] = 0	1	0

Adım	Satır	j	Okunan hücre	cikis	giris
4	4–5	2	a[1][2] = 1	2	0
5	6–7	0	a[0][1] = 1	2	1
6	6–7	1	a[1][1] = 0	2	1
7	6–7	2	a[2][1] = 0	2	1
8	8	—	return 2 + 1	sonuç 3	

Tablo 15.6: `derece(a, 3, 1)` izlemesi (G3). Satır numaraları yukarıdaki koda aittir.



Şekil 15.7: İzlemenin sonunda G3 ve matrisi. Düğüm 1 vurgulanmıştır; 1. satırdaki iki 1 çıkış derecesini, 1. sütundaki tek 1 giriş derecesini verir.

Her döngü tam *n* kez döner; toplam *2n* hücre okunur. Maliyet *O(2n) = O(n)*'dir ve düğümün gerçek derecesinden bağımsızdır. Aynı yordam G3'ün öteki düğümleri için 2 (düğüm 0) ve 1 (düğüm 2) döndürür; bu değerler Tablo 15.3 ile örtüşür.

15.3.2 Komşuluk listesi

Matrisin her satırı çoğunlukla sıfırlarla doludur. Komşuluk listesi bu sıfırları hiç saklamaz. Matrisin her satırı yerine bir bağlı liste tutulur ve listede yalnız 1 olan sütunların numaraları bulunur.

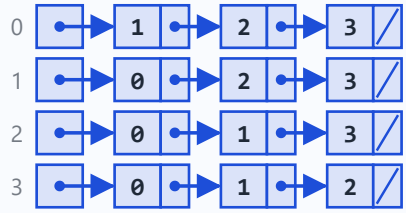
Tanım 15.8

Komşuluk listesi (*adjacency list*) gösteriminde her *i* düğümü için bir bağlı liste vardır; listenin öğeleri *i*'nin komşularının numaralarıdır. Listelerin baş işaretçileri *n* öğeli bir `baslik` dizisinde tutulur: `baslik[i]`, *i*'nin komşu listesinin ilk öğesini gösterir.

C bildirimi kendine başvuran bir yapıyla yapılır:

```
#define MAX_DUGUM 50   /* en fazla dugum sayisi */
typedef struct oge *oge_gost;
typedef struct oge {
    int dugum;         /* komsu dugumun numarasi */
    oge_gost bag;      /* listedeki sonraki komsu */
} oge;                 /* ';' ve typedef adi unutulmamali */
oge_gost baslik[MAX_DUGUM];   /* her dugume bir liste */
```

- **Satır 1.** `MAX_DUGUM`, programın tutabileceği en çok düğüm sayısıdır. Dizi boyutları bu sabitle belirlenir.
- **Satır 2.** `oge_gost`, henüz tanımı yazılmamış `struct oge`'yi gösteren işaretçi tipidir. C'de bir yapının işaretçisi, yapının tanımından önce bildirilebilir; yapının boyutunun bilinmesi gerekmez. Bu sıra, yapının içinde kendi tipini gösteren bir alan kullanmayı kolaylaştırır.
- **Satır 3–6.** Her liste öğesi iki alan taşır: komşunun numarası `dugum` ve listedeki sonraki öğeyi gösteren `bag`. Tanım `}` ve tip adı `oge`'den sonra `;` ile biter. Noktalı virgül unutulursa derleyici hatayı çoğunlukla sonraki satırda bildirir ve hatanın yeri şaşırtıcı görünür.
- **Satır 7.** `baslik` dizisi her düğüme bir liste başı ayırır. Küresel bir dizi olduğu için bütün öğeleri program başlarken `NULL`'dır; yani başta her liste boştur.



Şekil 15.8: G1'in komşuluk listesi. Soldaki kutular `baslik` dizisinin hücreleridir. Her öğe iki bölmelidir: komşu numarası ve sonraki öğenin bağı. Eğik çizgi `NULL` bağı gösterir.

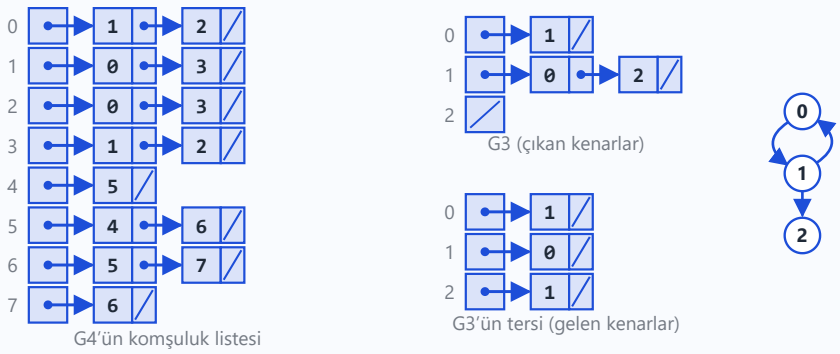
Şekil 15.8'de her düğümün üç komşusu vardır ve her liste üç öğelidir. Toplam 12 öğe bulunur; G1'in 6 kenarı vardır. Yönsüz çizgede her kenar iki ucunun listesinde birer kez görünür. Bu, gösterimin bir dezavantajıdır: aynı bilgi iki yerde saklanır ve kenar silinirken iki listenin de güncellenmesi gerekir.

Listelerin bellek ve zaman maliyetleri şöyledir:

- **Bellek.** n başlık ve yönsüz çizgede $2e$ öğe: $O(n + e)$. x86-64 üzerinde bir `oge` 16 bayttır (4 baytlık `int`, 4 bayt dolgu, 8 baytlık işaretçi); bir başlık 8 bayttır.
- **Derece.** Yönsüz çizgede i'nin listesindeki öğeler sayılır: $O(d_i)$. Matristeki $O(n)$'e göre derece küçükse büyük kazançtır.
- **Kenar sayısı.** Bütün listelerdeki öğeler sayılır ve ikiye bölünür: n başlığa bakmak ve $2e$ öğeyi saymak gerekir, toplam $O(n + e)$. Matristeki $O(n^2)$'den çok daha azdır, çünkü e çoğu çizgede n^2'den çok küçüktür.
- **(u, v) var mı?** u'nun listesinde v aranır: $O(d_u)$. Matriste bu soru $O(1)$'di; listenin kaybettiği tek yer burasıdır.

Çizge algoritmalarında karmaşıklık bu yüzden çoğunlukla hem n'ye hem e'ye bağlı yazılır. "$O(n + e)$" ifadesi, her düğüme ve her kenara sabit sayıda kez dokunan bir algoritmayı anlatır.

Yönsüz ve yönlü örnekler



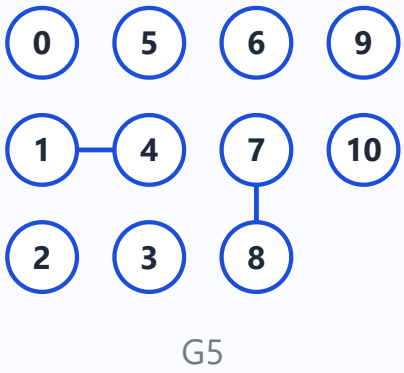
Şekil 15.9: Solda G4'ün komşuluk listesi; sağda G3'ün listesi (çıkan kenarlar) ve onun tersi (gelen kenarlar). Listeler küçükten büyüğe sıralıdır.

Yönsüz G4'ün listeleri sıralı tutulursa kenarları saymak biraz kısalır. Her (i, j) kenarı hem i'nin hem j'nin listesinde bulunur. Yalnız $j > i$ olan öğeler sayılırsa her kenar bir kez sayılır ve ikiye bölmeye gerek kalmaz. Sıralı listede $j < i$ olan öğeler başta toplanır, örneğin 3'ün listesindeki 1 ve 2. Bu öğeler hızla geçilir, kalanlar sayılır. Taranan öğe sayısı azalır, ama asimptotik maliyet yine $O(n + e)$'dir.

Yönlü G3'te durum farklıdır. $\langle 0,1 \rangle$ ile $\langle 1,0 \rangle$ ayrı kenarlar olduğundan her öğe bir kenardır ve hepsi sayılmalıdır. Liste yalnız çıkan kenarları tutar: `baslik[i]` 'deki öğe sayısı i'nin çıkış derecesidir ve $O(d)$'de bulunur. Giriş derecesi ise listede doğrudan yazmaz. i'ye gelen kenarları bulmak için bütün listeleri gezip i'yi içeren öğeleri saymak gerekir: $O(n + e)$. Giriş derecesi sık soruluyorsa bu maliyet kabul edilemez. Çözüm, listenin tersini, yani gelen kenarların listesini de tutmaktır. Bu ters liste (*inverse adjacency list*) bir sonraki alt kesimde kurulur.

Seyrek bir çizge: G5

Matris ile liste arasındaki farkı en iyi seyrek bir çizge gösterir. G5'in 11 düğümü ve yalnız 2 kenarı vardır: $(1, 4)$ ve $(7, 8)$.



Şekil 15.10: G5: 11 düğüm, 2 kenar. Kenara değmeyen yedi düğüm tek başına birer bileşendir; çizgenin toplam 9 bileşeni vardır.

Komşuluk listesinde dört öğe bulunur: `baslik[1]` → 4, `baslik[4]` → 1, `baslik[7]` → 8 ve `baslik[8]` → 7. Kalan yedi başlık `NULL`'dır. Toplam 11 başlık ve $2e = 4$ öğe tutulur. Matris ise $11 \times 11 = 121$ hücre ister; bunların yalnız 4'ü 1'dir ve 117 hücre boşa gider.

Buradan genel bir kural çıkar: seyrek çizgede (e, n^2'ye göre küçükse) liste, yoğun çizgede (e, n^2'ye yakınsa) matris düşünülür. Yoğun bir çizgede listenin işaretçileri matrisin bitlerinden çok daha fazla yer tutar. Ama karar yalnız yoğunluğa göre verilmez. Asıl ölçüt, çizge üzerinde yapılacak işlemlerdir. Program sürekli "bu iki düğüm komşu mu?" diye soruyorsa seyrek çizgede bile matris daha hızlı olabilir. Program her düğümün komşularında geziniyorsa yoğun bir çizgede bile liste daha uygun olabilir.

15.3.3 Listeyi kurmak: kenar ekleme

Örneklerdeki listeleri bir kişi çizgeye bakarak çizdi. Program ise çizgeye bakamaz. Kenarlar programa bir dosyadan ya da kullanıcıdan birer birer gelir ve her biri geldiği anda yapıya eklenebilmelidir. Bu işi yapan yordam şudur:

```
void kenar_ekle(int u, int v)   /* u -> v */
{
    oge_gost p = malloc(sizeof(oge));
    if (p == NULL) exit(1);       /* bellek yok */
    p->dugum = v;
    p->bag = baslik[u];   /* basa ekle: O(1) */
    baslik[u] = p;
}
```

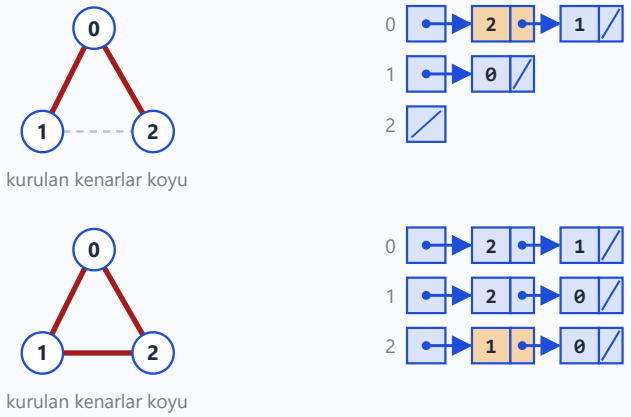
- **Satır 1.** Yordam $u \to v$ yönünde tek bir kenar ekler: u'nun listesine v'yi koyar. Yönsüz bir (u, v) kenarı için yordam iki kez, `kenar_ekle(u, v)` ve `kenar_ekle(v, u)` biçiminde çağrılır.
- **Satır 3.** Yeni öğe için heap bellekten yer alınır. `malloc`'un döndürdüğü `void *`, C'de dönüşüm yazmadan `oge_gost`'a atanabilir. `sizeof(oge)` yerine `sizeof(oge_gost)` yazmak sık bir hatadır: işaretçinin boyutu kadar (8 bayt) yer alınır, oysa öğe 16 bayttır ve sonraki atamalar ayrılmamış belleğe yazar.
- **Satır 4.** `malloc` bellek bulamazsa `NULL` döndürür. Denetim yapılmazsa bir sonraki satır `NULL` işaretçi üzerinden yazar ve program çöker. Burada en basit tepki seçilmiştir: program bir hata koduyla sonlanır.
- **Satır 5.** Komşu numarası öğeye yazılır.
- **Satır 6.** Yeni öğenin bağı, listenin şimdiki ilk öğesini gösterir. Liste boşsa bu değer `NULL`'dır ve yeni öğe listenin son öğesi de olur.
- **Satır 7.** Liste başı yeni öğeyi gösterir. Satır 6 ile 7'nin sırası önemlidir: önce 7 yazılsaydı eski liste başı kaybolur ve listenin geri kalanına ulaşılamazdı.

Yordam listeyi taramaz; her kenar sabit sayıda adımda, $O(1)$'de yerleşir. e kenarlı yönsüz bir çizge $2e$ çağrıyla kurulur; başlıkların sıfırlanması da eklenince kurulum $O(n + e)$ sürer.

Tablo 15.7, kenarlar (0,1), (0,2), (1,2) sırasıyla geldiğinde listelerin nasıl oluştuğunu izler. Tabloda `&oge(k)`, komşu numarası k olan öğenin adresini gösterir.

Çağrı	`p->dugum`	`p->bag`	`baslik[0]`	`baslik[1]`	`baslik[2]`
`kenar_ekle(0, 1)`	1	NULL	1	—	—
`kenar_ekle(1, 0)`	0	NULL	1	0	—
`kenar_ekle(0, 2)`	2	&oge(1)	2 → 1	0	—
`kenar_ekle(2, 0)`	0	NULL	2 → 1	0	0
`kenar_ekle(1, 2)`	2	&oge(0)	2 → 1	2 → 0	0
`kenar_ekle(2, 1)`	1	&oge(0)	2 → 1	2 → 0	1 → 0

Tablo 15.7: Üç kenarlı bir üçgenin komşuluk listesinin kurulması. Her yönsüz kenar iki çağrıyla eklenir.



Şekil 15.11: Listenin kurulması. Üstte ilk üç çağrıdan sonra: `baslik[0]` 'da 2 ve 1, `baslik[1]` 'de 0. Altta altı çağrının sonunda. Her yeni öğe listenin başına girdiği için listeler gelen sıranın tersidir.

Sonuçta 0'ın listesi 2 → 1'dir; kenarlar 1, 2 sırasıyla gelmiştir. Başa ekleme, listeyi gelen sıranın tersine çevirir. Bu durum bir hata değildir, ama gezinme algoritmalarının çıktısını etkiler; çünkü gezinme komşuları listedeki sırayla ziyaret eder. Liste sırası değişince gezinme sırasının da değiştiği, gezinme kesimindeki örneklerde açıkça görülür.

Başa, sona ya da sıralı ekleme

Bir öğe listeye üç yerden girebilir. Seçim, istenen sıraya ve maliyete bağlıdır (Tablo 15.8).

Yöntem	Bir ekleme	Liste sırası	Not
Başa ekleme	O(1)	Gelen sıranın tersi	Kurulum O(n + e)
Sona ekleme (yürüyerek)	O(d)	Gelen sıra	Her seferinde `NULL` 'a kadar yürünür
Sona ekleme (son işaretçisiyle)	O(1)	Gelen sıra	Her liste için ayrıca bir `son[i]` tutulur
Sıralı ekleme	O(d)	Küçükten büyüğe	Gezinme sırası öngörülebilir olur

Tablo 15.8: Komşuluk listesine ekleme seçenekleri. *d*, listenin o anki uzunluğudur.

Sıralı listeler çizgenin hangi sırayla kurulduğundan bağımsızdır; gezinme çıktısı yalnız çizgeye bağlı olur. Notun izleme örneklerinde listeler bu yüzden sıralıdır. Sıralı ekleme, bağlı listelerdeki işaretçinin işaretçisi tekniğiyle kısa yazılır:

```
void sirali_ekle(int u, int v)   /* u listesine v, artan sirada */
{
    oge_gost *pp = &baslik[u];   /* isaretcinin isaretcisi */
    oge_gost p;
    while (*pp && (*pp)->dugum < v)
        pp = &(*pp)->bag;
    p = malloc(sizeof(oge));
    if (p == NULL) exit(1);
    p->dugum = v;
    p->bag = *pp;
    *pp = p;
}
```

`pp`, değiştirilecek işaretçi alanının adresini tutar. Başta bu alan `baslik[u]`'dur. Döngü, v'den küçük öğeleri geçerken `pp`'yi her öğenin `bag` alanına ilerletir. Döngü bittiğinde `*pp`, yeni öğenin önüne gireceği öğeyi (ya da `NULL`'u) gösterir. Yeni öğe `*pp`'nin yerine yazılır. Listenin başına, ortasına ve sonuna ekleme aynı iki satırla yapılır; "liste boş mu, başa mı ekleniyor" gibi özel durumlar gerekmez.

Belleği geri vermek

Her `malloc` çağrısının karşılığında bir `free` çağrısı bulunmalıdır. Program sonunda ya da çizge artık gerekmediğinde her `baslik[i]` listesi gezilerek öğeler serbest bırakılır:

```
void listeleri_birak(oge_gost bas[], int n)
{
    int i;
    oge_gost p;
    for (i = 0; i < n; i++) {
        while (bas[i]) {
            p = bas[i];
            bas[i] = p->bag;   /* once bag saklanir */
            free(p);
        }
    }
}
```

Döngünün içindeki sıra bağlı listelerden bilinen kuraldır: öğe serbest bırakılmadan önce bağı saklanır. `free(p)` sonrasında `p->bag`'i okumak serbest bırakılmış belleğe erişmektir ve tanımsız davranıştır. Yordam her öğeye bir kez dokunur: $O(n + e)$. Notun bütün C kodları sayaçlı bir `malloc` / `free` sarmalayıcısıyla çalıştırıldı; listeler bu yordamla bırakıldığında ayrılan ve geri verilen blok sayıları eşitti.

> **Sık yapılan hata**
>
> Listeyi kurarken `malloc` sonucunu denetlememek ve program sonunda listeleri serbest bırakmamak. Küçük bir deneme programında bu hatalar görünmez. Çizgeyi defalarca kurup yıkan uzun süreli bir programda ise bellek sızıntısı yavaş yavaş birikir ve sonunda `malloc` gerçekten `NULL` döndürür.

15.3.4 Ters komşuluk listesi

Yönlü çizgede komşuluk listesi çıkan kenarları tutar. Ters liste ise gelen kenarları tutar: `ters[j]`, j'ye kenar gönderen düğümlerin listesidir. Bu, çizgenin bütün oklarının ters çevrildiği **transpoz** çizgenin komşuluk listesidir. Ters liste, mevcut listeleri bir kez gezerek kurulur:

```
void devrik_kur(oge_gost bas[], oge_gost ters[],
                int n)
{
    int i;
    oge_gost p;
    for (i = 0; i < n; i++) ters[i] = NULL;
    for (i = 0; i < n; i++)
        for (p = bas[i]; p; p = p->bag)
            ekle(ters, p->dugum, i);  /* <j,i> */
}
```

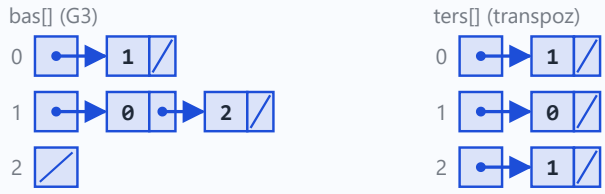
Yordamın kullandığı `ekle`, `kenar_ekle`'nin herhangi bir başlık dizisi üzerinde çalışan biçimidir:

```
void ekle(oge_gost bas[], int u, int v)   /* bas[u] listesine v */
{
    oge_gost p = malloc(sizeof(oge));
    if (p == NULL) exit(1);
    p->dugum = v;
    p->bag = bas[u];
    bas[u] = p;
}
```

- **Satır 6.** Ters listenin bütün başlıkları `NULL` yapılır.
- **Satır 7–8.** Her *i* düğümünün listesi baştan sona gezilir. Listede *j* numaralı bir öğe *<i, j>* kenarı demektir.
- **Satır 9.** Bu kenar ters çizgede *<j, i>* olur: `ters[j]` listesine *i* eklenir. Açıklama satırındaki `<j,i>` bunu söyler.

Satır	`i`	`p->dugum`	Kenar	Eklenen
6	—	—	—	`ters[0..2] = NULL`
8–9	0	1	<0,1>	`ters[1]`'e 0
8–9	1	0	<1,0>	`ters[0]`'a 1
8–9	1	2	<1,2>	`ters[2]`'ye 1
7–8	2	—	—	`bas[2]` boş, ekleme yok
7	3	—	—	döngü biter

Tablo 15.9: `devrik_kur(bas, ters, 3)` izlemesi (G3). `bas[0]` = 1, `bas[1]` = 0 → 2, `bas[2]` = NULL.



Şekil 15.12: G3'ün komşuluk listesi (solda) ve `devrik_kur` ile kurulan tersi (sağda). `ters[2]`'deki 1, 2'ye 1'den gelindiğini söyler.

Dış döngü *n* kez döner; iç döngüler toplamda her öğeye bir kez bakar. Her `ekle` çağrısı *O(1)*'dir. Ters liste bu yüzden *O(n + e)*'de kurulur. Kurulduktan sonra giriş derecesi de `ters[i]`'deki öğeleri saymakla, *O(d)*'de bulunur. Bedeli bellektir: her kenar iki kez, bir kez çıkış listesinde, bir kez giriş listesinde saklanır.

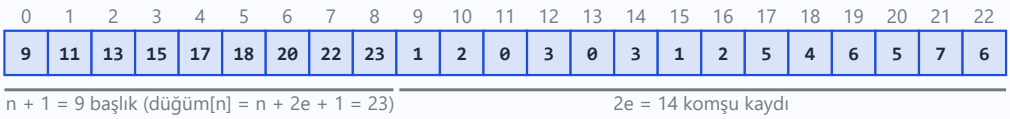
15.3.5 Dizi ile sıralı gösterim

Komşuluk listesinin her öğesi bir işaretçi taşır. 64 bitlik bir makinede bu işaretçi, 4 baytlık komşu numarasının iki katı yer tutar. Çizge hiç değişmeyecekse işaretçilerden tümüyle kurtulmak mümkündür: bütün listeler tek bir diziye art arda yazılır.

Tanım 15.9

n düğümlü ve *e* kenarlı yönsüz bir çizgenin **sıralı gösterimi** (*sequential representation*), *n + 2e + 1* öğeli tek bir `int dugum[]` dizisidir. İlk *n + 1* hücre başlangıç indislerini tutar. *i*'nin komşuları `dugum[dugum[i]]`, …, `dugum[dugum[i+1] − 1]` hücrelerindedir.

Dizinin iki bölümü vardır. Baştaki *n + 1* hücre bir içindekiler tablosu gibidir: `dugum[i]`, *i*'nin komşularının dizide nereden başladığını söyler. Arkadaki *2e* hücre komşu numaralarının kendisidir; düğümlerin listeleri sırayla, aralarında boşluk bırakılmadan yazılır. *i*'nin komşuları bir sonraki düğümün listesinin başladığı yerde biter. Son düğümün de bir bitiş sınırı olması için fazladan bir hücre, `dugum[n]`, eklenir. Bu yüzden hücre sayısı *n + 1 + 2e*'dir.



Şekil 15.13: G4'ün sıralı gösterimi. 0–8 indisli hücreler başlangıç indisleridir; `dugum[8] = 23` son düğümün bitiş sınırıdır. 9–22 indisli 14 hücre komşu numaralarını tutar.

G4 için *n = 8* ve *e = 7*'dir; dizi *8 + 14 + 1 = 23* hücrelidir. Komşu kayıtları 9. indisten başlar. 0'ın iki komşusu 9 ve 10. hücrelere, 1'in iki komşusu 11 ve 12. hücrelere yazılır; bu böyle sürer. Tablo 15.10 diziyi okumayı örneklerle gösterir.

Düğüm *i*	`dugum[i]`	`dugum[i+1]`	Okunan indisler	Komşular	Derece
0	9	11	9, 10	1, 2	2
1	11	13	11, 12	0, 3	2
4	17	18	17	5	1
5	18	20	18, 19	4, 6	2
7	22	23	22	6	1

Tablo 15.10: G4'ün sıralı gösteriminden komşu okuma. Başlangıç dahil, bitiş hariçtir.

Bir düğümün komşularını yazan yordam iki sınır arasında yürür:

```c
void komsulari_yaz(const int dugum[], int i)
{
    int k;
    for (k = dugum[i]; k < dugum[i + 1]; k++)
        printf("%d ", dugum[k]);
    printf("\n");
}
```

Dizi `const int dugum[]` olarak alınır; yordam diziyi yalnız okur ve derleyici yanlışlıkla yazılmasını engeller. Döngü değişkeni `k`, `dugum[i]`'den başlar ve `dugum[i + 1]`'e varmadan durur. Tablo 15.11 `komsulari_yaz(dugum, 5)` çağrısını izler.

Satır	`k`	Koşul `k < dugum[6]`	`dugum[k]`	Çıktı
4–5	18	18 < 20 doğru	4	`4`
4–5	19	19 < 20 doğru	6	`4 6`
4, 6	20	20 < 20 yanlış	—	`4 6` ve satır sonu

Tablo 15.11: `komsulari_yaz(dugum, 5)` izlemesi (G4).

Döngü düğümün derecesi kadar döner: *O(d)*. Derecenin kendisi döngüye bile gerek kalmadan `dugum[i+1] - dugum[i]` farkıyla, *O(1)*'de bulunur. Bellek *n + 2e + 1* tam sayıdır ve hiç işaretçi yoktur. G4 için 23 tam sayı, yani 92 bayt yeter. Aynı çizgenin komşuluk listesi 8 başlık (64 bayt) ve 14 öğe (224 bayt) ister.

Dizi var olan listelerden tek geçişte kurulabilir:

```
int sirali_kur(int dugum[], int n)   /* listelerden tek dizi */
{
    int i, k = n + 1;                /* komsular n+1'den baslar */
    oge_gost p;
    for (i = 0; i < n; i++) {
        dugum[i] = k;                /* i'nin ilk komsusunun yeri */
        for (p = baslik[i]; p; p = p->bag)
            dugum[k++] = p->dugum;
    }
    dugum[n] = k;                    /* son dugumun bitis siniri */
    return k;                        /* kullanilan hucre: n + 2e + 1 */
}
```

Yordam her düğüm için önce başlangıç indisini yazar, sonra listesindeki komşuları sıradaki boş hücrelere kopyalar. Döngü bitince `k`, kullanılan hücre sayısına, yani *n + 2e + 1*'e eşittir. Bu değer `dugum[n]`'e yazılarak son düğümün sınırı konur. G4'ün sıralı listelerine uygulandığında Şekil 15.13'teki dizinin aynısını üretir.

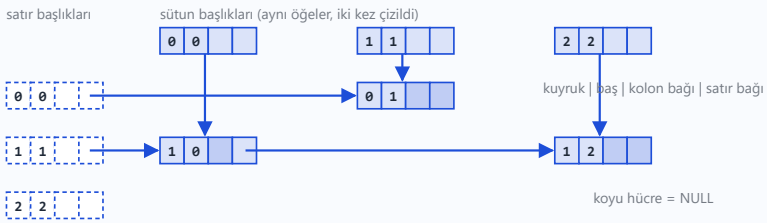
Gösterimin bedeli değişikliktir. Yeni bir kenar eklemek, iki listeye birer hücre açmak demektir. Bu da arkadaki bütün hücreleri kaydırmayı ve kaydırılan düğümlerin başlangıç indislerini güncellemeyi gerektirir: *O(n + e)*. Çok sayıda ekleme ve silme yapılan bir çizge için bu yapı verimli değildir. Depolamak ayrı bir konudur, saklanan veri üzerinde çalışacak algoritmaların etkinliği ayrı bir konudur. Örneğin bir binadaki bilgisayar ağı neredeyse hiç değişmez. Problem yalnız bu ağı tutup üzerinde sorgu yapmaksa sıralı gösterim hem az yer kaplar hem de bellekte bitişik durduğu için hızlı okunur. Seyrek matrislerdeki sıkıştırılmış satır gösterimi (CSR) aynı fikrin matrise uygulanmış biçimidir.

15.3.6 Yönlü çizgelerin ortogonal gösterimi

Yönlü çizgede hem çıkış hem giriş derecesi sık soruluyorsa komşuluk listesi ile tersini birlikte tutmak gerekir. Bu durumda her kenar iki öğeyle, toplam *2e* öğeyle saklanır. Ortogonal gösterim her kenarı tek bir öğede tutar ve bu öğeyi aynı anda iki listeye bağlar.

Tanım 15.10

Ortogonal liste (*orthogonal list*) gösteriminde yönlü çizgenin her *<kuyruk, baş>* kenarı için tek bir öğe vardır. Öğe dört alan taşır: kuyruk, baş, *satır bağı* (aynı kuyruktan çıkan sonraki kenar) ve *kolon bağı* (aynı başa gelen sonraki kenar). Her düğümün bir satır başlığı (çıkış listesi) ve bir sütun başlığı (giriş listesi) vardır.



Şekil 15.14: G3'ün ortogonal gösterimi. Solda satır başlıkları, üstte sütun başlıkları vardır. Her kenar öğesi bir satır (çıkış) ve bir sütun (giriş) listesindedir. Koyu hücreler `NULL` bağlardır.

Gösterimin adı yapının görünüşünden gelir: öğeler bir matrisin dolu hücreleri gibi satır ve sütunlara dizilir. Bir düğümün satırında gezinmek o düğümden *nereye çıkıldığını*, sütununda gezinmek o düğüme *nereden gelindiğini* verir. Aynı kenar için tek öğe yeter. G3'te *<1,0>* öğesi hem 1'in satırında hem 0'ın sütunundadır.

Bu yapı seyrek matrislerin bağlı gösterimine çok benzer. Seyrek matriste de her sıfır olmayan değer bir öğedir ve kendi satır ve sütun listelerine bağlanır. Yapı ihtiyaca göre döngülü, çift yönlü bağlı ya da başlık düğümlü kurulabilir. Çıkış ve giriş listeleri *O(d)*'de gezilir; başa ekleme *O(1)*'dir.

Bu gösterimin ortaya çıkışı öğretici bir düşünme biçimini gösterir. Liste ile tersini birlikte tutan çözüm doğrudur ama *2e* öğe kullanır. Bu israf rahatsız edicidir ve "aynı işi daha az yerle yapabilir miyim?" sorusunu doğurur. Akla gelen ilk çözüme ve onun ilk koduna hiçbir zaman tam güvenilmemelidir. Arka plandaki veri yapısı başka bir şey olsaydı problem daha verimli çözülebilir miydi? Bu soruyu sormak, iyi bir veri yapısı tasarımının ilk adımıdır.

15.3.7 Çok bağlı liste

Yönsüz çizgenin komşuluk listesinde de her kenar iki öğeyle saklanır. Kenarlara bir bilgi eklenecekse (örneğin "bu kenar işlendi" işareti) bu bilginin iki kopyasını tutarlı tutmak gerekir. Çok bağlı liste bu tekrarı kaldırır.

> **Tanım 15.11**
>
> **Çok bağlı liste** (*adjacency multilist*) gösteriminde yönsüz çizgenin her (v_i, v_j) kenarı için tek bir öğe vardır. Öğe hem v_i'nin hem v_j'nin kenar listesinde yer alır. Temel öğe düğüm değil, kenardır.

```
typedef struct kenar *kenar_gost;
typedef struct kenar {
    short isaret;        /* islendi mi */
    int dugum1, dugum2;  /* uclar */
    kenar_gost list1;    /* dugum1 sonraki */
    kenar_gost list2;    /* dugum2 sonraki */
} kenar;
kenar_gost baslik[MAX_DUGUM];
```

- **Satır 3.** `isaret`, kenarın işlenip işlenmediğini tutar. Kenar iki listede göründüğü için bir gezinme aynı kenara iki kez rastlar; işaret, ikinci rastlayışta kenarın yeniden ele alınmasını önler.
- **Satır 4.** `dugum1` ve `dugum2` kenarın iki ucudur.
- **Satır 5–6.** `list1`, `dugum1`'in listesindeki sonraki kenarı; `list2`, `dugum2`'nin listesindeki sonraki kenarı gösterir. Bir öğe iki ayrı listenin zincirinde aynı anda bulunur.
- **Satır 8.** `baslik[v]`, v'nin kenar listesindeki ilk öğeyi gösterir. Hiç kenarı olmayan düğümün başlığı `NULL`'dır.

Düğüme veri taşımak gerekirse (örneğin şehir adı) yapıya ayrı bir düğüm dizisi ya da öğeye `char veri` gibi bir alan eklenir. x86-64 üzerinde bu `kenar` yapısı dolgu baytlarıyla 32 bayttır. Komşuluk listesinde aynı kenar iki öğeyle $2 \cdot 16 = 32$ bayt tutar. Kazanç bellekte değil, bilginin tek yerde durmasındadır.

Tablo 15.12 G1'in çok bağlı listesini verir. Altı kenar N0–N5 öğeleridir.

Öğe	`dugum1`	`dugum2`	`list1`	`list2`	Hangi listelerde
N0	0	1	N1	N3	0 ve 1
N1	0	2	N2	N3	0 ve 2
N2	0	3	NULL	N4	0 ve 3
N3	1	2	N4	N5	1 ve 2
N4	1	3	NULL	N5	1 ve 3
N5	2	3	NULL	NULL	2 ve 3

Tablo 15.12: G1'in çok bağlı listesi. Başlıklar: `baslik[0]` = N0, `baslik[1]` = N0, `baslik[2]` = N1, `baslik[3]` = N2.

Tabloyu okumak için bir düğümün listesi izlenir. 1'in listesi `baslik[1]` = N0'dan başlar. N0'da 1, `dugum2`'dir; sonraki kenar `list2` = N3'tür. N3'te 1, `dugum1`'dir; sonraki kenar `list1` = N4'tür. N4'te 1 yine `dugum1`'dir ve `list1` = NULL ile liste biter. 1'in kenarları N0, N3, N4; komşuları 0, 2 ve 3'tür.

Çok bağlı listede gezinme

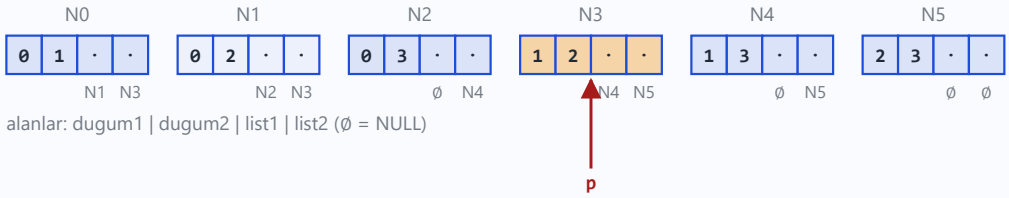
Gezinmenin kuralı kısadır: v, öğenin `dugum1` alanındaysa `list1`, `dugum2` alanındaysa `list2` izlenir. `NULL` görülünce liste biter.

```c
void komsulari_yaz(int v)
{
    kenar_gost p = baslik[v];
    while (p) {
        if (p->dugum1 == v) {       /* v: dugum1 */
            printf("%d ", p->dugum2);  p = p->list1;
        } else {                    /* v: dugum2 */
            printf("%d ", p->dugum1);  p = p->list2;
        }
    }
}
```

Her öğede iki iş yapılır: öteki uç yazılır ve doğru bağ seçilir. *v* hangi uçta ise öteki uç komşudur ve izlenecek bağ *v*'nin tarafındaki bağdır. Tablo 15.13 `komsulari_yaz(2)` çağrısını izler.

Satır	`p`	Öğe	2 hangi uçta	Yazılan	Sonraki `p`
3	N1	(0,2)	—	—	—
5, 8	N1	(0,2)	`dugum2`	0	`list2` = N3
5, 8	N3	(1,2)	`dugum2`	1	`list2` = N5
5–6	N5	(2,3)	`dugum1`	3	`list1` = NULL
4	NULL	—	—	döngü biter; çıktı `0 1 3`	

Tablo 15.13: Çok bağlı listede `komsulari_yaz(2)` izlemesi (G1).



Şekil 15.15: G1'in çok bağlı listesi bellekte. Her kutu dört alanlıdır: `dugum1`, `dugum2`, `list1`, `list2`; altlarındaki adlar bağların gösterdiği öğelerdir. `p`, izlemede 2'nin ikinci kenarı N3'e gelmiştir.

Döngü *v*'nin her kenarı için bir kez döner: $O(d_v)$. Her kenar bellekte yalnız bir kez saklanmıştır, ama iki ucunun listesinden de ulaşılabilir.

Çok bağlı listeye kenar eklemek

Yeni bir kenar iki listeye birden girer. Listelerin sonuna eklemek mümkündür, ama her iki listede `NULL`'a kadar yürümeyi gerektirir. Başa eklemek bu maliyeti ortadan kaldırır:

```c
void ml_ekle(int u, int v)
{
    kenar_gost p = malloc(sizeof(kenar));
    if (p == NULL) exit(1);
    p->isaret = 0;
    p->dugum1 = u;  p->dugum2 = v;
    p->list1 = baslik[u];  /* u listesine */
    p->list2 = baslik[v];  /* v listesine */
    baslik[u] = baslik[v] = p;  /* ikisine de basa */
}
```

- **Satır 3–4.** Öğe için bellek alınır ve denetlenir.
- **Satır 5–6.** Kenar henüz işlenmemiştir; uçlar yazılır.
- **Satır 7.** `list1`, *u*'nun eski ilk kenarını gösterir. Bu eski kenarda *u*'nun hangi uçta olduğu önemli değildir; gezinme yordamı her öğede bunu kendisi belirler.

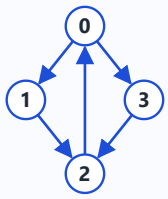
- **Satır 8.** `list2`, *v*'nin eski ilk kenarını gösterir.
- **Satır 9.** İki başlık da yeni öğeyi gösterir. Satır 7 ile 8, bu atamadan önce yapılmalıdır; yoksa eski listeler kaybolur.

Ekleme *O(1)*'dir. Yönsüz çizgede tek bir başlık dizisi yeter. Yönlü bir çizgede ise ortogonal gösterimdeki gibi bir baş dizisiyle birlikte bir kuyruk dizisi tutulur. G1'in kenarları (0,1), (0,2), ..., (2,3) sırasıyla `ml_ekle` ile eklenirse listeler gelen sıranın tersi olur; `komsulari_yaz(2)` bu kez `3 1 0` yazar. Tablo 15.12 ise kenarların listelerin sonuna eklendiği düzeni gösterir; o düzende çıktı `0 1 3`'tür. İki yapı da doğrudur, ama hangi kodun hangi çizimi ürettiği tutarlı olmalıdır. Kenarlar bu sırayla gelir ve "her biri *O(1)*'de yerleşsin" denirse liste başına eklenir. Bu durumda listeyi sona ekleniyormuş gibi çizmek, kodun yaptığıyla çelişen bir çizimdir.

15.3.8 Uygulama: aynı çizgenin dört gösterimi

Örnek 15.2

Kenarları *<0,1>, <0,3>, <1,2>, <2,0>, <3,2>* olan dört düğümlü yönlü çizge için komşuluk matrisi, komşuluk listesi, ters listesi ve ortogonal (tek öğeli) listesi kurulsun.



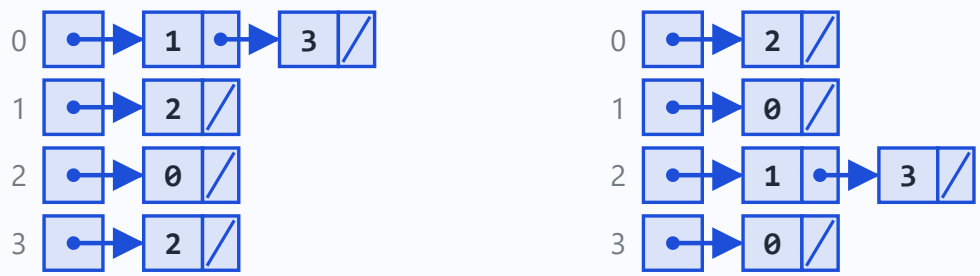
Şekil 15.16: Uygulamanın yönlü çizgesi. 0 → 1 → 2 → 0 bir yönlü döngüdür.

Matris. *<i, j>* varsa *a[i][j] = 1* yazılır; satır kuyruğu, sütun başı gösterir (Tablo 15.14). Satır toplamları çıkış derecelerini (2, 1, 1, 1), sütun toplamları giriş derecelerini (1, 1, 2, 1) verir. İki toplamın her biri 5'tir; yönlü çizgede her kenar bir çıkış ve bir giriş derecesine katkı yapar.

	0	**1**	**2**	**3**	**çıkış**
0	0	1	0	1	2
1	0	0	1	0	1
2	1	0	0	0	1
3	0	0	1	0	1
giriş	1	1	2	1	5

Tablo 15.14: Uygulama çizgesinin komşuluk matrisi.

Liste ve tersi. Çıkış listeleri: 0 → 1, 3; 1 → 2; 2 → 0; 3 → 2. Giriş listeleri (transpoz): 0 ← 2; 1 ← 0; 2 ← 1, 3; 3 ← 0. İkisinde toplam *2e = 10* öğe vardır.



Şekil 15.17: Uygulama çizgesinin komşuluk listesi (çıkan kenarlar, solda) ve tersi (gelen kenarlar, sağda).

Tek öğeli gösterim. Her kenar tek bir öğedir: N1 = <0,1>, N2 = <0,3>, N3 = <1,2>, N4 = <2,0>, N5 = <3,2>. Her öğe kuyruğunun çıkış listesinde ve başının giriş listesinde yer alır. Kuyruk bağı, aynı kuyruktan çıkan sonraki kenarı; baş bağı, aynı başa giren sonraki kenarı gösterir. Yalnız $e = 5$ öğe gerekir.

Öğe	kuyruk	baş	kuyruk bağı	baş bağı		Düğüm	çıkış	giriş
N1	0	1	N2	NULL		0	N1	N4
N2	0	3	NULL	NULL		1	N3	N1
N3	1	2	NULL	N5		2	N4	N3
N4	2	0	NULL	NULL		3	N5	N2
N5	3	2	NULL	NULL				

Tablo 15.15: Uygulama çizgesinin tek öğeli gösterimi. Solda kenar öğeleri, sağda düğüm başlıkları.

Sağlama için 2'nin giriş listesi izlenir: başlık N3'tür (<1,2>), N3'ün baş bağı N5'tir (<3,2>), N5'in baş bağı `NULL` 'dır. 2'ye 1'den ve 3'ten gelinir; giriş derecesi 2'dir ve matrisin 2. sütunuyla örtüşür. 0'ın çıkış listesi N1'den başlar, kuyruk bağıyla N2'ye geçer ve biter: 0'dan 1'e ve 3'e gidilir.

15.3.9 Gösterimlerin karşılaştırılması

Tablo 15.16 beş gösterimi aynı ölçütlerle yan yana koyar. d_u, u'nun derecesidir.

Gösterim	Bellek	(u, v) var mı?	u'nun komşuları	Ekleme / silme
Komşuluk matrisi	$O(n^2)$	$O(1)$	$O(n)$	$O(1)$
Komşuluk listesi	$O(n + 2e)$	$O(d_u)$	$O(d_u)$	başa $O(1)$; silme için arama $O(d)$
Sıralı dizi	$n + 2e + 1$ tam sayı	$O(d_u)$	$O(d_u)$	pahalı: kaydırma $O(n + e)$
Ortogonal liste (yönlü)	$O(n + e)$	$O(d)$	giriş ve çıkış $O(d)$	başa $O(1)$
Çok bağlı liste	$O(n + e)$	$O(d_u)$	$O(d_u)$	başa $O(1)$

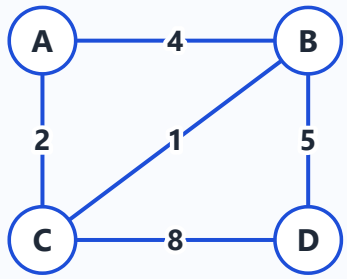
Tablo 15.16: Çizge gösterimlerinin bellek ve işlem maliyetleri.

Tablonun hiçbir satırı her sütunda en iyi değildir. Matris kenar sorgusunda rakipsizdir ama komşu taramasında ve bellekte yavaştır. Listeler gezinmede hızlıdır ama kenar sorgusunda yavaştır. Sıralı dizi en az yeri tutar ama değişikliğe gelmez. Seçim, problemde hangi işlemin kaç kez yapılacağına bağlıdır.

Bu kesimin çıkarımı bir yöntemdir: önce verinin nasıl depolanacağı düşünülür, sonra kod yazılır. Akla gelen ilk çözüm iyi bir yöntem olmayabilir. Veri yapısı iyi tasarlanırsa n^3 adımlı bir algoritmanın n adıma inmesi bile mümkün olabilir. Bunun ilk örneği hemen aşağıda görülür: aynı gezinme algoritması matrisle $O(n^2)$, listeyle $O(n + e)$ sürer. Veriyi depolamak göründüğü kadar kolay bir iş değildir.

15.3.10 Ağırlıklı çizgeler

Pek çok problemde kenarın yalnız var olması yetmez; bir de değeri vardır. Yol ağında bu değer mesafe, bilgisayar ağında gecikme, proje planında süre, kablo döşemede maliyettir. Kenarlarına sayı atanmış çizgeye **ağırlıklı çizge** (*weighted graph*), bu sayılara da **ağırlık** denir.



Şekil 15.18: Dört düğümlü ağırlıklı bir çizge. Kenarların üzerindeki sayılar ağırlıklardır.

Her iki temel gösterim kolayca genişler. Matriste 1 yerine kenarın ağırlığı yazılır; böyle bir matrise **maliyet matrisi** denir. Kenar yoksa hücreye "sonsuz" anlamına gelen bir değer yazılır. Köşegen 0'dır: bir düğümden kendisine gitmenin maliyeti yoktur. Listede ise her öğeye bir `agirlik` alanı eklenir.

	A	B	C	D
A	0	4	2	∞
B	4	0	1	5
C	2	1	0	8
D	∞	5	8	0

Tablo 15.17: Şekildeki çizgenin maliyet matrisi.
Kenar olmayan çift ∞ ile gösterilir.

```
#define SONSUZ 1000000   /* INT_MAX degil: toplamda tasar */
int maliyet[MAX_DUGUM][MAX_DUGUM];

typedef struct aoge *aoge_gost;
typedef struct aoge {
    int dugum;
    int agirlik;        /* kenarin maliyeti */
    aoge_gost bag;
} aoge;
aoge_gost abaslik[MAX_DUGUM];
```

C'de sonsuz diye bir tam sayı yoktur; büyük bir sabit seçilir. Akla ilk gelen `INT_MAX`'tır, ama bu tehlikeli bir seçimdir. En kısa yol algoritmaları sürekli $d + w$ biçiminde toplamlar hesaplar. d = `INT_MAX` iken herhangi bir pozitif w eklenirse işaretli tam sayı taşar. C'de işaretli taşma tanımsız davranıştır; pratikte sonuç çoğunlukla büyük bir negatif sayıdır ve algoritma "sonsuz uzaklıktaki" düğümü en yakın düğüm sanır. `SONSUZ` bu yüzden özenle seçilmiştir. Gerçek ağırlıkların hepsinden büyüktür, ama iki katı bile `int`'e sığar. $2 \cdot 1.000.000$, 32 bitlik `int`'in sınırının (yaklaşık 2,1 milyar) çok altındadır.

Liste öğesi `aoge`, `oge`'ye bir `int agirlik` eklenmiş hâlidir; x86-64 üzerinde iki `int` 8 bayt, işaretçi 8 bayt eder ve dolgu gerekmez: 16 bayt. Minimum kapsayan ağaç ve en kısa yol algoritmalarının hepsi bu iki gösterimden birini kullanır.

15.4 Gezinmeler

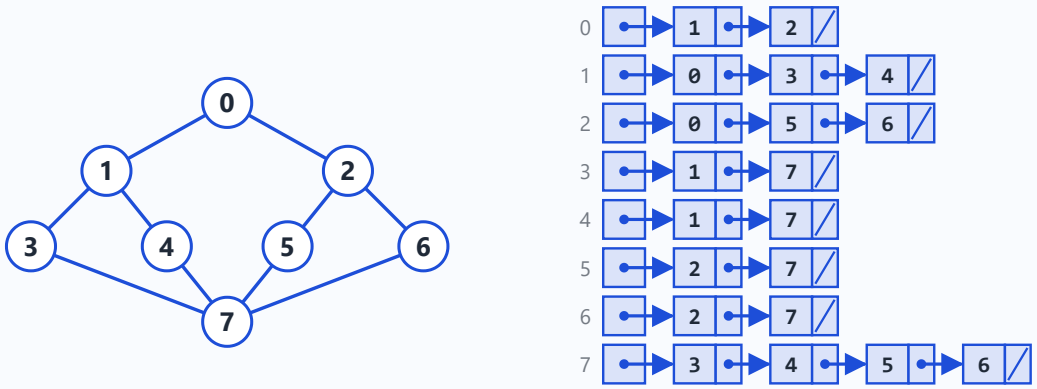
Çizge üzerindeki en temel işlem gezinmedir. Yönsüz bir $G = (V, E)$ çizgesi ve bir v düğümü verilsin. v'den ulaşılabilen bütün düğümlere birer kez uğramak istenir. "Uğramak" ya da ziyaret etmek, düğümle ilgili işi yapmak demektir; bu notta düğümün numarasını yazdırmaktır. Bileşen bulma, döngü arama, en kısa yol ve kapsayan ağaç gibi pek çok algoritma bir gezinmenin üzerine kurulur.

Ağaçlarda gezinme kolaydır, çünkü ağaçta döngü yoktur ve her düğüme kökten tek bir yolla varılır. Çizgede ise bir düğüme birçok yoldan varılabilir. Bu yüzden çizge gezinmesinin ilk kuralı şudur: ziyaret edilen düğüm işaretlenir ve işaretli bir düğüme yeniden girilmez. İşaretler olmazsa gezinme bir döngü üzerinde sonsuza dek döner.

İki temel gezinme yöntemi vardır:

- **Derinlik öncelikli arama** (*depth first search*, DFS). Bir komşuya gidilir ve oradan olabildiğince derine inilir. Ağaçtaki kök önde gezinmeye benzer.
- **Genişlik öncelikli arama** (*breadth first search*, BFS). Önce başlangıcın bütün komşularına, sonra onların komşularına uğranır. Ağaçtaki düzey düzey gezinmeye benzer.

İki yöntemde de çizge komşuluk listesiyle gerçeklenir ve listeler sıralıdır. Örnek olarak sekiz düğümlü bir çizge kullanılır (Şekil 15.19).



Şekil 15.19: Gezinmelerin örnek çizgesi (solda) ve sıralı komşuluk listesi (sağda). Çizgenin 8 düğümü ve 10 kenarı vardır; listelerde *2e = 20* öğe bulunur.

15.4.1 Derinlik öncelikli arama

DFS'nin fikri bir labirentte ip çekerek ilerlemeye benzer. *v* ziyaret edilir. Sonra *v*'nin listesinden ziyaret edilmemiş bir *w* seçilir ve arama *w* üzerinden sürer. *w*'nin de ziyaret edilmemiş bir komşusu varsa ona geçilir. Ziyaret edilmemiş komşusu kalmayan bir *u* düğümüne gelindiğinde bir adım geri dönülür ve *u*'ya hangi düğümden gelindiyse onun listesinde kalınan yerden devam edilir.

Bu geri dönüşler için her düğümün listesinde nerede kalındığının hatırlanması gerekir. Son giren ilk çıktığı için bu bilgi bir yığında (*stack*) tutulur: derine inerken düğümler yığına konur, gidecek yer kalmayınca yığından bir düğüm alınıp devam edilir. Açık bir yığınla yazılmış DFS karmaşık görünür. Özyinelemeli yordam ise aynı işi çok kısa yapar, çünkü yığını derleyicinin çağrı yığını tutar. Her çağrının yerel `w` değişkeni, o düğümün listesinde nerede kalındığını saklar.

Önce ortak bildirimler:

```
#include <stdio.h>
#include <stdbool.h>
#define TRUE  true
#define FALSE false
bool ugranildi[MAX_DUGUM];   /* ';' unutulmamali */
```

`stdbool.h`, C99'dan beri `bool` tipini ve `true` / `false` değerlerini verir. `TRUE` ve `FALSE` makroları ders kitabındaki yazımı korumak içindir. `ugranildi` küresel bir dizidir; program başlarken bütün öğeleri `false`'tur. Gezinme yeniden başlatılacaksa dizinin elle sıfırlanması gerekir.

```
void dfs(int v)   /* once derinligine arama */
{
    oge_gost w;
    ugranildi[v] = TRUE;    /* ugradim */
    printf("%d ", v);
    for (w = baslik[v]; w; w = w->bag)
        if (!ugranildi[w->dugum])
            dfs(w->dugum);
}
```

- **Satır 3.** `w`, *v*'nin listesinde ilerleyen işaretçidir. Her özyinelemeli çağrının kendi `w`'si olur.

- **Satır 4.** *v* işaretlenir. İşaret, düğümün listesi taranmadan önce konur. Böylece *v*'nin komşularından birinin listesinde *v* görüldüğünde ona geri girilmez.
- **Satır 5.** Ziyaret: düğüm numarası yazılır.
- **Satır 6.** *v*'nin listesi baştan sona gezilir. `for` başlığındaki `w` koşulu `w != NULL` anlamına gelir.
- **Satır 7–8.** Komşu işaretli değilse arama onun üzerinden sürer. Özyinelemeli çağrı döndüğünde döngü, bu çağrının `w`'sinin gösterdiği yerden, yani kalınan yerden devam eder.

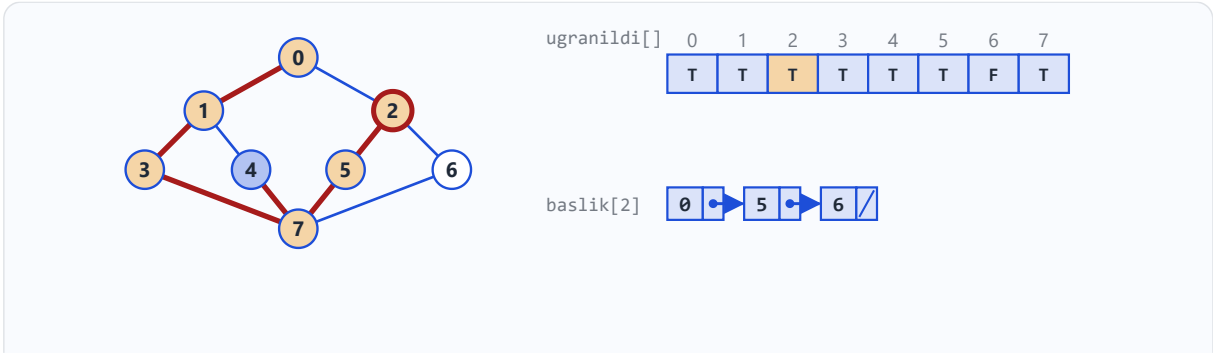
15.4.2 DFS izlemesi

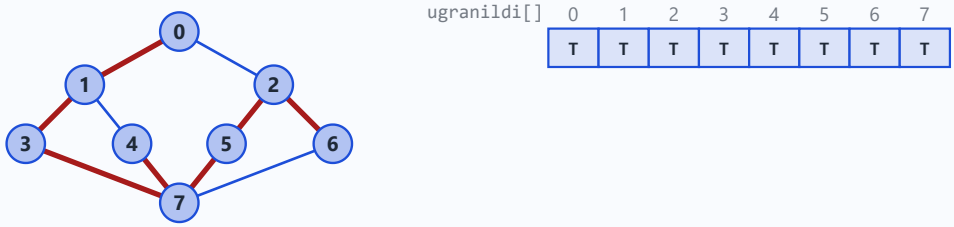
`dfs(0)` çağrısı örnek çizgede izlenir. Tablo 15.18'de çağrı yığını en alttaki `dfs(0)`'dan başlayıp en üstteki etkin çağrıya kadar yazılır. "Atlanır" satırlarında komşu zaten işaretlidir.

Adım	Olay	Çağrı yığını	Çıktı
1	`dfs(0)`: 0 işaretlenir, yazılır	0	0
2	0 → 1 işaretsiz: `dfs(1)`	0 1	0 1
3	1'in listesinde `w` → 0: işaretli, atlanır	0 1	0 1
4	1 → 3 işaretsiz: `dfs(3)`	0 1 3	0 1 3
5	3'ün listesinde `w` → 1: atlanır	0 1 3	0 1 3
6	3 → 7 işaretsiz: `dfs(7)`	0 1 3 7	0 1 3 7
7	7'nin listesinde `w` → 3: atlanır	0 1 3 7	0 1 3 7
8	7 → 4 işaretsiz: `dfs(4)`	0 1 3 7 4	0 1 3 7 4
9–10	4'ün listesi: 1 ve 7 işaretli; `dfs(4)` biter	0 1 3 7	0 1 3 7 4
11	7 → 5 işaretsiz: `dfs(5)`	0 1 3 7 5	0 1 3 7 4 5
12	5 → 2 işaretsiz: `dfs(2)`	0 1 3 7 5 2	0 1 3 7 4 5 2
13–14	2'nin listesinde 0 ve 5: atlanır	0 1 3 7 5 2	0 1 3 7 4 5 2
15	2 → 6 işaretsiz: `dfs(6)`	0 1 3 7 5 2 6	0 1 3 7 4 5 2 6
16–17	6'nın listesi: 2 ve 7 işaretli; `dfs(6)`, `dfs(2)` biter	0 1 3 7 5	aynı
18	5'te kalınan yer: `w` → 7 atlanır; `dfs(5)` biter	0 1 3 7	aynı
19	7'de kalınan yer: `w` → 6 atlanır; `dfs(7)`, `dfs(3)` biter	0 1	aynı
20	1'de kalınan yer: `w` → 4 atlanır; `dfs(1)` biter	0	aynı
21	0'da kalınan yer: `w` → 2 atlanır; `dfs(0)` biter	—	0 1 3 7 4 5 2 6

Tablo 15.18: Örnek çizgede `dfs(0)` izlemesi.

İzlemenin en önemli anları dönüşlerdir. 12. adımda arama 5'ten 2'ye geçer. 0 ile 2 arasında doğrudan bir kenar vardır. Yine de 2'ye 0 → 1 → 3 → 7 → 5 → 2 gibi uzun bir yoldan varılmıştır. DFS yakını değil, derini önce dener. 18. adımda `dfs(5)`, 2'ye inmeden önceki yerinden, listesindeki ikinci öğe 7'den devam eder. Bunu sağlayan, `dfs(5)` çerçevesinde saklı duran `w`'dir. Yığının en derin anı 15. adımdır: yedi çağrı aynı anda açıktır.





Şekil 15.20: DFS'nin iki anı. Üstte `dfs(2)` yeni çağrılmıştır: kırmızı kalın kenarlar şimdiye dek izlenen ağaç kenarlarıdır, sağda `ugranildi` dizisi ve 2'nin listesi görünür. Altta gezinme bitmiştir; ağaç kenarları derinlik öncelikli kapsayan ağacı oluşturur.

Gezinme sırasında bir düğüme ilk kez girilen kenarlar bir ağaç oluşturur: 0–1, 1–3, 3–7, 7–4, 7–5, 5–2, 2–6. Yedi kenar sekiz düğümün hepsini döngüsüz bağlar. Bu ağaca **derinlik öncelikli kapsayan ağaç** (*depth first spanning tree*) denir. Ağaca kullanılmayan kenarlardan biri, örneğin 0–2, eklenirse bir döngü oluşur ve yapı ağaç olmaktan çıkar.

15.4.3 DFS'nin sonucu neye bağlıdır

DFS'nin ürettiği sıra çizgenin kendisine ek olarak iki şeye bağlıdır.

Başlangıç düğümü. Başlangıcın 0 olması gerekmez. Aynı çizgede ve aynı listelerle `dfs(6)` `6 2 0 1 3 7 4 5`, `dfs(4)` ise `4 1 0 2 5 7 3 6` yazar. Ziyaret edilen düğümler aynıdır, dizilişleri farklıdır. Bağlantılı bir çizgede her başlangıç bütün düğümlere ulaşır.

Listelerin sırası. 0'ın listesi {1, 2} yerine {2, 1} olsaydı `dfs(0)` önce 2'ye inerdi ve sıra `0 2 5 7 3 1 4 6` olurdu. Listenin başa mı sona mı eklenerek kurulduğu ya da sıralı tutulup tutulmadığı gezinmeyi doğrudan değiştirir. Bu nedenle bir DFS çıktısı ancak listelerin tam sırası bilindiğinde kesin olarak söylenebilir.

> **Teorem 15.2 (DFS'nin maliyeti)**
>
> Komşuluk listesiyle gerçeklenmiş n düğümlü, e kenarlı bir çizgede `dfs`'nin toplam çalışma süresi $O(n + e)$'dir. Komşuluk matrisiyle gerçeklenirse $O(n^2)$'dir.

İspat. `dfs(v)` yalnız `ugranildi[v]` yanlışken çağrılır ve ilk iş olarak bu değeri doğru yapar. Bu yüzden her düğüm için en çok bir çağrı yapılır; çağrıların toplam sayısı en çok n'dir. Her çağrı kendi düğümünün listesini bir kez tarar; v'nin listesinde d_v öğe vardır. Bütün listelerin toplam uzunluğu Teorem 15.1'e göre $2e$'dir. Toplam iş $O(n + 2e) = O(n + e)$ olur. Matriste bir düğümün komşularını bulmak bütün satırı, yani n hücreyi okumayı gerektirir; n çağrının her biri $O(n)$ sürer ve toplam $O(n^2)$ olur. ■

Örnek çizgede sayaçlı bir `dfs` 8 çağrı yaptı ve listelerde tam 20 öğeye baktı; bu değerler $n = 8$ ve $2e = 20$ ile örtüşür. Kenar sayısı en çok $n(n - 1)/2$ olabilir. Tam çizgede e, n^2 mertebesine çıkar ve iki gösterimin maliyeti eşitlenir. Seyrek çizgelerde ise liste, gezinmeyi doğrusal zamanda yapar.

Ek bellek de hesaba katılmalıdır. `ugranildi` dizisi n öğelidir. Çağrı yığınının derinliği en kötü durumda n'dir: düğümleri bir zincir gibi dizilmiş bir çizgede her çağrı bir sonrakini açar. Çok büyük çizgelerde bu derinlik çağrı yığınını taşırabilir; o durumda DFS açık bir yığınla yazılır.

15.4.4 Genişlik öncelikli arama

BFS, suya atılan taşın oluşturduğu halkalar gibi yayılır. v'den başlanır ve v işaretlenir. Sonra v'nin listesindeki düğümlerin hepsi ziyaret edilir. Ardından v'nin listesindeki ilk düğümün ziyaret edilmemiş komşuları, sonra ikinci düğümünkiler ziyaret edilir. Bu böyle sürer. Ziyaret edilen her düğüm bir kuyruğa (*queue*) konur. Bir düğümün listesi bitince kuyruğun önündeki düğüm alınır ve onun listesine geçilir. Kuyruk boşalınca arama biter.

Yöntemin doğası gereği kuyruk zorunludur, tıpkı DFS'de yığının zorunlu olması gibi. Önce ziyaret edilen düğümün komşuları önce ziyaret edilmelidir; “ilk giren ilk çıkar” kuralı tam olarak budur.

```
void bfs(int v)   /* once genisligine arama */
{
    oge_gost w;
    kuyruk_gost on = NULL, arka = NULL;
    printf("%d ", v);
    ugranildi[v] = TRUE;
    kuyruk_ekle(&on, &arka, v);
    while (on) {
        v = kuyruk_al(&on, &arka);
        for (w = baslik[v]; w; w = w->bag)
            if (!ugranildi[w->dugum]) {
                printf("%d ", w->dugum);
                kuyruk_ekle(&on, &arka, w->dugum);
                ugranildi[w->dugum] = TRUE;
            }
    }
}
```

- **Satır 4.** Kuyruk bağlı listeyle gerçeklenir; `on` ve `arka` başta `NULL`'dır, yani kuyruk boştur.
- **Satır 5–7.** Başlangıç düğümü yazılır, işaretlenir ve kuyruğa konur.
- **Satır 8.** Kuyruk boş değilken döner. Boş kuyrukta `on` `NULL`'dır.
- **Satır 9.** Kuyruğun önündeki düğüm alınır. Parametre `v` burada yeniden kullanılır; yordamın başındaki değeri artık gerekmez.
- **Satır 10–15.** `v`'nin listesi taranır. İşaretsiz her komşu yazılır, kuyruğun arkasına eklenir ve işaretlenir.

İşaretin zamanı önemlidir: düğüm kuyruğa eklenirken işaretlenir, kuyruktan çıkarken değil. Böylece bir düğüm kuyrukta beklerken başka bir düğümün listesinde yeniden görülse bile ikinci kez kuyruğa girmez. Bu ayrıntının değeri, hatalı biçimin sonucuna bakılınca anlaşılır.

Dinamik kuyruk

BFS'nin kullandığı kuyruk, bağlı listeyle gerçeklenmiş dinamik bir kuyruktur. Önden alınır, arkaya eklenir; iki uç için iki işaretçi tutulur.

```
typedef struct kuyruk *kuyruk_gost;
typedef struct kuyruk {
    int dugum;
    kuyruk_gost bag;
} kuyruk;
void kuyruk_ekle(kuyruk_gost *on,
                 kuyruk_gost *arka, int d)
{
    kuyruk_gost p = malloc(sizeof(kuyruk));
    if (p == NULL) exit(1);
    p->dugum = d;  p->bag = NULL;
    if (*on) (*arka)->bag = p;  else *on = p;
    *arka = p;
}
```

`kuyruk_ekle`, iki işaretçiyi de değiştirebilmek için onların adreslerini alır (`kuyruk_gost *on`). Bu, C'de bir işaretçiyi fonksiyon içinden değiştirmenin tek yoludur. Yeni öğenin bağı `NULL`'dır, çünkü kuyruğun sonuna girer. Kuyruk boş değilse eski son öğenin bağı yeni öğeyi gösterir; boşsa yeni öğe aynı zamanda ön öğedir. İki durumda da arka yeni öğeyi gösterir. İşlem *O(1)*'dir.

```
int kuyruk_al(kuyruk_gost *on,
              kuyruk_gost *arka)
{
    kuyruk_gost p = *on;   /* on != NULL */
    int d = p->dugum;
    *on = p->bag;
    if (*on == NULL) *arka = NULL;
    free(p);
    return d;
}
```

`kuyruk_al` boş olmayan bir kuyruk bekler; açıklama satırı bu ön koşulu söyler ve BFS yordamı onu `while (on)` ile sağlar. Önündeki öğenin değeri saklanır, ön işaretçi bir sonrakine geçer ve öğe serbest bırakılır. Kuyruk boşalınca arka da `NULL` yapılmalıdır. Yapılmazsa `arka`, az önce serbest bırakılmış öğeyi göstermeye devam eder. Sonraki `kuyruk_ekle` `*on`'u boş görür ve doğru davranır, ama aradaki sürede `arka` sarkan bir işaretçidir (*dangling pointer*). Bu yüzden `int kuyruk_al(kuyruk_gost *)` gibi yalnız ön işaretçiyi alan bir prototip yetmez; `arka`'nın da adresi geçirilir. İşlem *O(1)*'dir.

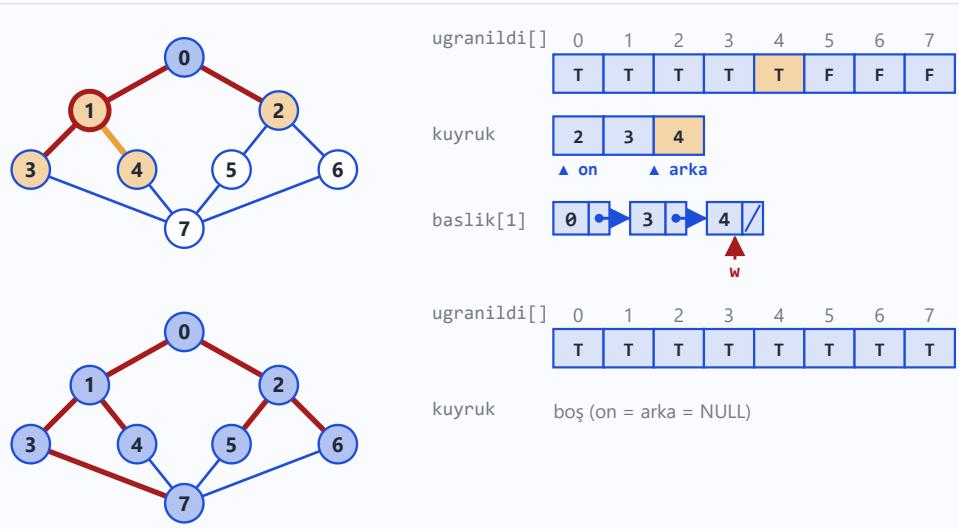
15.4.5 BFS izlemesi

Tablo 15.19 `bfs(0)` çağrısını örnek çizgede izler. Kuyruk sütununda sol uç önü, sağ uç arkayı gösterir.

Adım	v	Olay	Kuyruk (ön ... arka)	Çıktı
1	0	0 yazılır, işaretlenir, kuyruğa girer	0	0
2	0	0 alınır; listesi {1, 2}	—	0
3–4	0	1 ve 2 işaretsiz: yazılır, eklenir, işaretlenir	1 2	0 1 2
5	1	1 alınır; listesi {0, 3, 4}; 0 işaretli	2	0 1 2
6–7	1	3 ve 4 eklenir	2 3 4	0 1 2 3 4
8	2	2 alınır; listesi {0, 5, 6}; 0 işaretli	3 4	0 1 2 3 4
9–10	2	5 ve 6 eklenir	3 4 5 6	0 1 2 3 4 5 6
11	3	3 alınır; listesi {1, 7}; 1 işaretli	4 5 6	0 1 2 3 4 5 6
12	3	7 eklenir	4 5 6 7	0 1 2 3 4 5 6 7
13	4	4 alınır; listesi {1, 7}: ikisi de işaretli	5 6 7	aynı
14	5	5 alınır; listesi {2, 7}: işaretli	6 7	aynı
15	6	6 alınır; listesi {2, 7}: işaretli	7	aynı
16	7	7 alınır; listesi {3, 4, 5, 6}: işaretli	—	aynı
17	—	kuyruk boş: arama biter	—	0 1 2 3 4 5 6 7

Tablo 15.19: Örnek çizgede `bfs(0)` izlemesi.

12. adımda 7 kuyruğa girer ve işaretlenir. 13. adımda 4'ün listesinde 7 yeniden görülür, ama işaretli olduğu için ikinci kez eklenmez. 5 ve 6'nın listelerinde de aynı şey olur. 7'ye dört farklı düğümden varılabilir, ama kuyruğa bir kez girer.



Şekil 15.21: BFS'nin iki anı. Üstte 1'in listesi taranıyor; 3 ve 4 kuyruğa eklenmiştir. Altta arama bitmiştir: kırmızı kenarlar genişlik öncelikli kapsayan ağaçtır.

Çıktı `0 1 2 3 4 5 6 7` 'dir: düğümler başlangıca olan kenar sayısına göre düzey düzey ziyaret edilmiştir. 0 düzey 0'da; 1 ve 2 düzey 1'de; 3, 4, 5, 6 düzey 2'de; 7 düzey 3'tedir. Ağaç kenarları 0–1, 0–2, 1–3, 1–4, 2–5, 2–6 ve 3–7'dir. Bu **genişlik öncelikli kapsayan ağaçta** kökten her düğüme giden yol, çizgedeki en az kenarlı yoldur.

Sık yapılan hata

BFS'de düğümü kuyruğa eklerken değil, kuyruktan çıkarırken işaretlemek. Örnek çizgede `ugranildi[w->dugum] = TRUE` satırı silinsin. Yerine `v = kuyruk_al(…)` 'dan sonra `ugranildi[v] = TRUE` yazılsın. Bu durumda çıktı `0 1 2 3 4 5 6 7 7 7 7` olur. 7, kuyruktan çıkmadan önce 3, 4, 5 ve 6'nın listelerinde birer kez görülür ve dört kez kuyruğa girer. Kuyruğa 8 yerine 11 ekleme yapılır; listelerde 20 yerine 32 öğe taranır. Yinelenen her kopya kuyruktan çıktığında kendi listesini yeniden tarar. Daha yoğun çizgelerde iş $O(n + e)$ sınırını açıkça aşar.

15.4.6 BFS'nin analizi ve DFS ile karşılaştırılması

Teorem 15.3 (BFS'nin maliyeti)

Komşuluk listesiyle gerçeklenmiş bir çizgede `bfs` $O(n + e)$ sürede çalışır; komşuluk matrisiyle $O(n^2)$.

İspat. Bir düğüm kuyruğa yalnız işaretsizken eklenir ve eklendiği anda işaretlenir. Bu yüzden her düğüm kuyruğa en çok bir kez girer ve `while` döngüsü en çok n kez döner. Her turda kuyruktan alınan düğümün listesi bir kez taranır. Bütün turlarda taranan öğelerin toplamı $d_0 + \dots + d_{n-1} = 2e$'dir. `kuyruk_ekle` ve `kuyruk_al` $O(1)$'dir. Toplam $O(n + e)$ olur. Matriste her tur bir satırın n hücresini okur; n turda $O(n^2)$ eder. ∎

BFS'nin ek belleği `ugranildi` dizisi ve kuyruktur; kuyrukta aynı anda en çok n düğüm bulunur. BFS'nin verdiği sıra, başlangıca olan kenar sayısının sırasıdır. Ağırlıksız bir çizgede bu, en kısa yolların sırası demektir; BFS bu yüzden en az kenarlı yolu bulan algoritmadır.

Derinlik öncelikli ve genişlik öncelikli arama kavramları veri yapılarının dışında da karşılaşılan iki genel arama stratejisidir. Yapay zekâda bir problemin bütün durumları bir arama uzayı oluşturur ve bu uzay bir ağaç gibi düşünülerek gezilir. Min-max araması, B* araması ve dal-sınır (*branch and bound*) yöntemleri bu gezinmelerin üzerine kurulur. Bir oyun probleminde "olası hamleler şunlardır" denip her hamle ayrı ayrı açılıyorsa arama genişliğine ilerliyor demektir.

Özellik	DFS	BFS
Yardımcı yapı	Yığın (özyinelemede çağrı yığını)	Kuyruk
İlerleyiş	Bir yoldan olabildiğince derine	Katman katman, yakından uzağa
Ağaçtaki karşılığı	Kök önde gezinme	Düzey düzey gezinme
Örnek çizgede `(0)`	0 1 3 7 4 5 2 6	0 1 2 3 4 5 6 7
Örnek çizgede `(7)`	7 3 1 0 2 5 6 4	7 3 4 5 6 1 2 0
Maliyet (liste / matris)	$O(n + e)$ / $O(n^2)$	$O(n + e)$ / $O(n^2)$
Tipik kullanım	Bileşenler, döngü tespiti, topolojik sıralama	En az kenarlı yol, iki parçalılık testi

Tablo 15.20: DFS ile BFS'nin karşılaştırılması.

15.5 Bileşenler, döngüler ve iki parçalılık

Gezinme yordamları tek başına yalnız düğüm numaralarını yazar. Asıl değerleri, üzerlerine kurulan algoritmalardadır. Bu kesim DFS ve BFS'yi üç soruyu cevaplamak için kullanır: Çizge kaç parçadan oluşur? Çizgede döngü var mı? Çizgenin düğümleri, her kenar iki grup arasında kalacak biçimde ikiye ayrılabilir mi?

15.5.1 Bağlantılı bileşenler

Tanım 15.12

Yönsüz bir çizgenin **bağlantılı bileşeni** (*connected component*), çizgenin en büyük bağlantılı alt çizgelerinden biridir. "En büyük" sözü, bileşene çizgeden başka bir düğüm eklenirse alt çizgenin artık bağlantılı olmayacağı anlamına gelir.

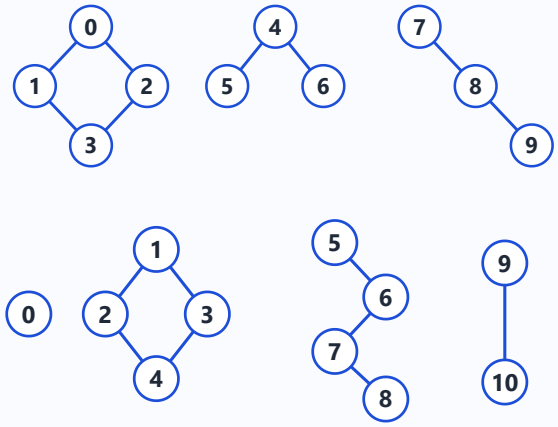
Her düğüm tam olarak bir bileşene aittir; bileşenler düğüm kümesini ayrık parçalara böler. G4'ün iki, G5'in dokuz bileşeni vardır. Bağlantılı bir çizgenin tek bileşeni kendisidir.

Bileşenleri bulmak için gezinmenin bir özelliği kullanılır: `dfs(v)`, v'den ulaşılabilen bütün düğümleri, yani v'nin bileşenini tümüyle işaretler. Gezinme bittiğinde işaretsiz kalan bir düğüm varsa o başka bir bileşendedir ve oradan yeni bir gezinme başlatılır.

```
void bitisik_bilesenler(int n)
{
    int i;
    for (i = 0; i < n; i++)
        if (!ugranildi[i]) {   /* yeni */
            dfs(i);
            printf("\n");
        }
}
```

- **Satır 4.** Bütün düğümler sırayla denenir.
- **Satır 5.** İşaretsiz bir düğüm, daha önceki hiçbir gezinmenin ulaşamadığı yeni bir bileşenin ilk düğümüdür.
- **Satır 6.** `dfs(i)` bu bileşenin bütün düğümlerini yazar ve işaretler.
- **Satır 7.** Her bileşen ayrı satıra yazılır. Bu satıra bir sayaç artırımı (`sayi++`) eklenirse yordam bileşen sayısını da verir; sayı 1 ise çizge bağlantılıdır.

Yordam DFS yerine BFS ile de aynı biçimde yazılabilir. Yordamın çalışması için `ugranildi` dizisinin başta tamamen `FALSE` olması gerekir.



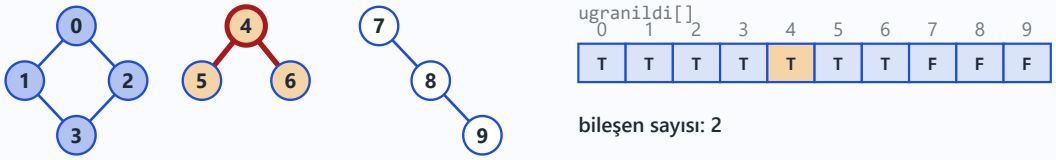
Şekil 15.22: İki bileşen örneği. Üstte 10 düğümlü, üç bileşenli çizge; çıktı `0 1 3 2 / 4 5 6 / 7 8 9`. Altta 11 düğümlü, dört bileşenli çizge; çıktı `0 / 1 2 4 3 / 5 6 7 8 / 9 10`.

Alttaki örnekte ikinci satırın çıktısı `1 2 4 3`'tür. 1'in listesi {2, 3} olduğu için DFS önce 2'ye, 2'nin listesindeki 4'e, oradan 3'e gider. 5 bu satırda görünmez; 5, 6, 7, 8 başka bir bileşendir. Yalnız 0'dan oluşan bileşen tek düğümlüdür; `dfs(0)` listesi boş olduğu için yalnız 0'ı yazar. Tablo 15.21 üstteki örneğin izlemesidir.

`i`	`ugranildi[i]`	Olay	Yazılan satır	Bileşen
0	F	yeni bileşen: `dfs(0)`	`0 1 3 2`	1
1, 2, 3	T	`if`'e girilmez, her biri $O(1)$	—	1

`i`	`ugranildi[i]`	Olay	Yazılan satır	Bileşen
4	F	yeni bileşen: `dfs(4)`	`4 5 6`	2
5, 6	T	`if`'e girilmez	—	2
7	F	yeni bileşen: `dfs(7)`	`7 8 9`	3
8, 9	T	`if`'e girilmez	—	3
10	—	döngü biter	3 bileşen	

Tablo 15.21: `bitisik_bilesenler(10)` izlemesi (üstteki çizge).



Şekil 15.23: İzlemenin `i` = 4 anı. İlk bileşenin dört düğümü işaretlenmiştir (T); `dfs(4)` ikinci bileşeni gezmiştir.

15.5.2 Analiz: çarpım değil, toplam

Yordamın maliyeti ilk bakışta yanıltıcıdır. Dıştaki `for` n kez döner. İçteki `dfs` çağrısının maliyeti $O(n + e)$'dir. Bu iki sayı çarpılıp karmaşıklık $O(n(n + e))$ denebilir mi? Hayır. "Döngüleri bulur, maliyetlerini çarparım" kuralı burada yanlış sonuç verir, çünkü iç işin maliyeti her turda aynı değildir.

Teorem 15.4 (Bileşen bulmanın maliyeti)

`bitisik_bilesenler` komşuluk listesiyle $O(n + e)$, komşuluk matrisiyle $O(n^2)$ sürede çalışır.

İspat. Bütün `dfs` çağrıları aynı `ugranildi` dizisini paylaşır. Bir düğüm bir kez işaretlenince hiçbir çağrı ona yeniden girmez. Öyleyse `for` boyunca yapılan bütün `dfs` çağrıları birlikte her düğümü bir kez ziyaret eder ve her listeyi bir kez tarar: toplam $O(n + e)$. Bu toplamın bir kısmı birinci bileşene, bir kısmı ikinciye düşer; bileşenler ayrık olduğu için hiçbir iş iki kez sayılmaz. Bunlara `for`'un kendi n turu eklenir; işaretli düğümlerde tur $O(1)$'dir. Toplam $O(n) + O(n + e) = O(n + e)$'dir. Matriste her ziyaret bir satır taraması gerektirdiğinden ziyaretlerin toplamı $O(n^2)$ olur. ∎

Uç durum bunu açıkça gösterir. Çizge bağlantılıysa ilk turda `dfs(0)` bütün çizgeyi $O(n + e)$'de gezer ve `ugranildi` dizisinin tamamı `TRUE` olur. `i`'nin geri kalan $n - 1$ değerinde `if`'e hiç girilmez. Çarpım ancak her turda `dfs`'nin bütün çizgeyi yeniden gezdiği durumda doğru olurdu; işaretler bunu engeller. Aynı türden bir akıl yürütmeyle, döngülerin iç içe olduğu ama iç döngünün toplam işinin sınırlı olduğu pek çok algoritmada karşılaşılır.

15.5.3 Bağlantılılık ve kenar sayısı

Bir çizgenin bağlantılı olabilmesi için kaç kenara ihtiyaç vardır?

Teorem 15.5 (Bağlantılılık için kenar sayısı)

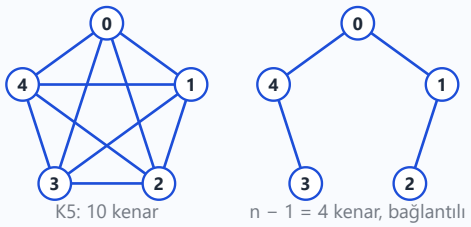
n düğümlü bağlantılı bir çizgede en az $n - 1$ kenar bulunur. Basit bir çizgede kenar sayısı en çok $n(n - 1)/2$'dir. Bağlantılı basit bir çizgede bu yüzden

$$n - 1 \le e \le n(n - 1)/2$$

olur.

İspat. Alt sınır için kenarsız bir çizgeden başlanıp kenarlar birer birer eklendiği düşünülür. Başta her düğüm kendi başına bir bileşendir; n bileşen vardır. Bir kenar eklendiğinde iki ucu ya aynı bileşendedir ve bileşen sayısı değişmez, ya da farklı bileşenlerdedir ve iki bileşen birleşir. Bir kenar bileşen sayısını en çok 1 azaltır. n bileşenden tek bileşene inmek için en az $n - 1$ kenar gerekir. Üst sınır için basit çizge-

de her düğüm çiftinin arasında en çok bir kenar olduğu kullanılır. 0 düğümü öteki $n - 1$ düğüme bağlanabilir. 1 düğümü kendinden sonraki $n - 2$ düğüme bağlanabilir. Böyle devam edilirse $(n - 1) + (n - 2) + \ldots + 1 = n(n - 1)/2$ bulunur. ■



Şekil 15.24: İki uç durum, 5 düğümle. Solda tam çizge K5: $5 \cdot 4 / 2 = 10$ kenar. Sağda 4 kenarlı bağlantılı bir çizge: $n - 1$ kenar yeterli olabilir.

Teorem tek yönlüdür. $n - 1$ kenar bağlantılılık için *gereklidir* ama *yeterli değildir*. 5 düğümlü bir çizgede 4 kenar bir zincir kurabilir; o zaman çizge bağlantılıdır. Aynı 4 kenar dört düğüm arasında bir döngü kurabilir; o zaman beşinci düğüm yalnız kalır ve çizge bağlantılı değildir. Kenar sayısı $n - 1$'den azsa çizge kesinlikle bağlantılı değildir; G5'te $e = 2 < 10$'dur.

Üst sınır, karmaşıklık ifadeleri açısından önemlidir. e, n^2 mertebesinde bir sayı olabilir. $O(n + e)$ ifadesi bu yüzden tam çizgede $O(n^2)$'ye eşittir; liste ancak seyrek çizgelerde matristen asimptotik olarak üstün olur. Bağlantılı ve tam $n - 1$ kenarlı bir çizge döngü içermez; böyle bir çizge bir ağaçtır. Gezinmelerin ürettiği kapsayan ağaçlarda da $n - 1$ kenar bulunur.

Oyun problemleri bu sınırların pratikteki önemini gösterir. Tic tac toe gibi bir oyunda her durum bir düğüm, her hamle bir kenardır. “Rakibin hamlesini nasıl bozarım?” sorusu bu çizgede bir arama yapmaktır. Durum sayısı hızla büyür ve aramanın maliyeti doğrudan düğüm ve kenar sayısına bağlıdır.

15.5.4 Yönsüz çizgede döngü tespiti

DFS'nin bir yan ürünü, çizgede döngü olup olmadığıdır. DFS bir düğümün listesini tararken işaretli bir komşuya rastlayabilir. Yönsüz çizgede bunun iki nedeni olabilir. Birincisi, komşu düğüme hemen önceki adımda ondan gelinmiştir; bu, gelinen kenarın öteki ucudur ve döngü belirtisi değildir. İkincisi, komşu daha önce başka bir yoldan işaretlenmiştir. Bu durumda düğümden o komşuya hem ağaç kenarlarıyla hem de bu kenarla gidilebilir: bir döngü vardır. Böyle kenarlara **geri kenar** (*back edge*) denir.

```
bool dongu_var(int v, int ata)
{
    oge_gost w;
    ugranildi[v] = TRUE;
    for (w = baslik[v]; w; w = w->bag)
        if (!ugranildi[w->dugum]) {
            if (dongu_var(w->dugum, v)) return TRUE;
        } else if (w->dugum != ata)
            return TRUE;          /* geri kenar */
    return FALSE;
}
```

- **Satır 1.** Yordam düğümün yanında `ata`'yı, yani bu düğüme hangi düğümden gelindiğini alır. İlk çağrıda ebeveyn yoktur; `-1` hiçbir düğümün numarası olmadığı için bu anlamda kullanılır.
- **Satır 4.** Düğüm işaretlenir.
- **Satır 6–7.** İşaretsiz komşuya inilir; ebeveyni şimdiki düğümdür. Alt çağrı döngü bulursa `TRUE`, çağrı zinciri boyunca hemen yukarı taşınır.
- **Satır 8–9.** Komşu işaretliyse ve ebeveyn değilse geri kenar bulunmuştur. `else if`, iç `if`'in değil dış `if`'in `else`'idir; süslü parantezler bunu açıkça belirler. Parantezler kaldırılırsa `else` en yakın `if`'e bağlanır ve yordamın anlamı değişir.
- **Satır 10.** Bütün komşular geçildi ve döngü bulunmadı.

Kenarları 0–1, 1–2, 2–3 ve 3–1 olan dört düğümlü çizgede sıralı listeler 0: {1}, 1: {0, 2, 3}, 2: {1, 3}, 3: {1, 2}'dir. Tablo 15.22 `dongu_var(0, -1)` çağrısını izler.

Çağrı	`w`	Durum	Sonuç
`dongu_var(0, -1)`	1	işaretsiz: in	—

Çağrı	`w`	Durum	Sonuç
`dongu_var(1, 0)`	0	işaretli ama ebeveyn	döngü sayılmaz
`dongu_var(1, 0)`	2	işaretsiz: in	—
`dongu_var(2, 1)`	1	işaretli ama ebeveyn	döngü sayılmaz
`dongu_var(2, 1)`	3	işaretsiz: in	—
`dongu_var(3, 2)`	1	işaretli ve ebeveyn (2) değil	geri kenar (3,1): `TRUE`

Tablo 15.22: `dongu_var(0, -1)` izlemesi.



Şekil 15.25: Döngü tespitinin son anı. Kırmızı kenarlar DFS ağacıdır; kesikli kenar (3,1) geri kenardır ve 1–2–3–1 döngüsünü kapatır.

`TRUE` değeri 3'ten 2'ye, 2'den 1'e, 1'den 0'a taşınır ve yordam erken biter. Döngü yoksa yordam DFS'nin kendisi kadar, $O(n + e)$ sürer. Çizgede birden çok bileşen varsa yordam, bileşen bulmadaki gibi her işaretsiz düğümden yeniden başlatılır. Ağaç olduğu bilinen G2 üzerinde yordam `FALSE` döndürür.

15.5.5 Yönlü çizgede döngü

Yönlü çizgede ebeveyn denetimi yetmez. Daha önce işaretlenmiş bir düğüme giden bir kenar döngü kapatmayabilir: o düğüm başka bir dalda çoktan bitirilmiş olabilir. Örneğin 0 → 1, 0 → 2, 1 → 3, 2 → 3 kenarlarında DFS 0, 1, 3 sırasıyla iner, sonra 2'den 3'e giden kenarı görür. 3 işaretlidir, ama döngü yoktur. Ayırt edici soru şudur: işaretli düğüm şu anda yığında mı? Yığındaki bir düğüme giden kenar, o düğümden şimdiki düğüme inen yolu kapatır ve bir döngüdür.

Bu ayrım üç renkle yapılır: **beyaz** henüz girilmemiş, **gri** girilmiş ama bitmemiş (yığında), **siyah** bitmiş düğümdür.

```
enum { BEYAZ, GRI, SIYAH };
int durum[MAX_DUGUM];   /* hepsi basta BEYAZ */

bool yonlu_dongu(int v)
{
    oge_gost w;
    durum[v] = GRI;                 /* yigina girdi */
    for (w = baslik[v]; w; w = w->bag) {
        if (durum[w->dugum] == GRI)
            return TRUE;            /* yigindaki dugume kenar */
        if (durum[w->dugum] == BEYAZ && yonlu_dongu(w->dugum))
            return TRUE;
    }
    durum[v] = SIYAH;               /* bitti */
    return FALSE;
}
```

Düğüme girilince gri yapılır. Gri bir komşu görülürse döngü vardır. Beyaz komşuya inilir. Bütün komşular bitince düğüm siyah olur ve artık hiçbir döngünün parçası olarak görülmez. Yordam G3'te (0 → 1 → 0) ve uygulama çizgesinde (0 → 1 → 2 → 0) `TRUE`, yukarıdaki dört düğümlü örnekte `FALSE` döndürdü. Maliyet yine $O(n + e)$'dir; matrisle aynı DFS $O(n^2)$ olur. Topolojik sıralama yordamı da döngüyü kendiliğinden yakalar.

Döngü tespitinin pratik önemi büyüktür. İşler arasında öncelik ilişkisi tanımlanan bir projede yönlü döngü, "A, B'den önce; B, A'dan önce bitmeli" gibi bir çelişki demektir ve proje hiç bitmez. Minimum kapsayan ağaç kurulurken döngü kapatan bir kenar ise gereksiz bir kenardır ve ağaca alınmaz.

15.5.6 İki parçalı çizge testi

Tanım 15.13

Düğüm kümesi V, ayrık iki A ve B kümesine ayrılabiliyorsa ve her kenarın bir ucu A'da, öteki ucu B'de bulunuyorsa çizge **iki parçalıdır** (*bipartite*).

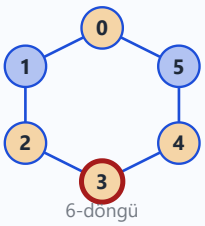
Öğrenciler ve dersler iki parçalı bir çizge oluşturur: kenarlar yalnız bir öğrenciyi bir derse bağlar. $K_{3,3}$, üçer düğümlü iki grubun her düğümünün öteki gruptaki her düğüme bağlı olduğu iki parçalı çizgedir.

Test bir boyama problemidir. Başlangıç düğümü 0 rengine boyanır. BFS her işaretsiz komşuyu, kendisinin ters rengine boyar. Bir kenarın iki ucu aynı renge boyanmış çıkarsa çizge iki parçalı değildir. Hiç çelişki çıkmazsa renkler iki grubu verir.

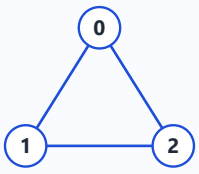
```
int renk[MAX_DUGUM];   /* -1: boyanmadi, 0 ya da 1 */

bool iki_parcali_mi(int n)
{
    int i, v;
    oge_gost w;
    kuyruk_gost on = NULL, arka = NULL;
    for (i = 0; i < n; i++) renk[i] = -1;
    for (i = 0; i < n; i++) {
        if (renk[i] != -1) continue;      /* yeni bilesen */
        renk[i] = 0;
        kuyruk_ekle(&on, &arka, i);
        while (on) {
            v = kuyruk_al(&on, &arka);
            for (w = baslik[v]; w; w = w->bag)
                if (renk[w->dugum] == -1) {
                    renk[w->dugum] = 1 - renk[v];
                    kuyruk_ekle(&on, &arka, w->dugum);
                } else if (renk[w->dugum] == renk[v]) {
                    while (on) kuyruk_al(&on, &arka);   /* kuyrugu bosalt */
                    return FALSE;
                }
        }
    }
    return TRUE;
}
```

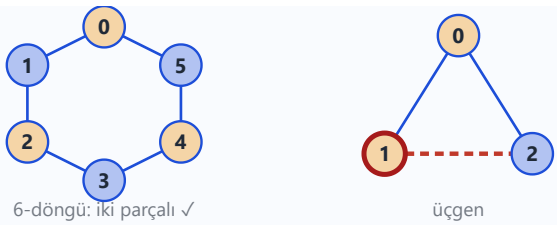
- **Satır 1.** `renk` dizisi hem işaret hem renk tutar: `-1` henüz boyanmamış düğümdür. Ayrı bir `ugranildi` dizisi gerekmez.
- **Satır 8–12.** Bütün renkler silinir; her bileşen için BFS ayrı başlatılır; bir bileşenin rengi ötekini etkilemez.
- **Satır 16–18.** Boyanmamış komşu ters renge boyanır ve kuyruğa girer.
- **Satır 19–21.** Komşu boyanmış ve rengi aynıysa test başarısızdır. Yordamdan çıkmadan önce kuyruk boşaltılır; yoksa kuyrukta kalan öğeler hiç serbest bırakılmaz.



6-döngü



üçgen



Şekil 15.26: İki renklendirme. Üstte 6 düğümlü döngü BFS ile boyanmıştır: 0, 2, 4 bir renkte, 1, 3, 5 öteki renktedir; çizge iki parçalıdır. Altta üçgen: 0 ve 1 farklı renge boyandıktan sonra 2, ikisine birden komşudur ve iki renk yetmez.

6 düğümlü döngüde BFS düğümleri 0, 1, 5, 2, 4, 3 sırasıyla boyar; renkler 0, 1, 1, 0, 0, 1'dir. Hiçbir kenarın iki ucu aynı renkte değildir. Üçgende ise 0'a renk 0, onun komşuları 1 ve 2'ye renk 1 verilir. 1'in listesi tarandığında 2 görülür; ikisi de 1 rengindedir ve test başarısız olur.

> **Teorem 15.6 (İki parçalılık)**
>
> Bir çizge, ancak ve ancak tek uzunlukta bir döngü içermiyorsa iki parçalıdır.

İspat. Bir yönü kolaydır. Çizge iki parçalıysa her kenar renk değiştirir. Bir döngü başladığı düğüme, yani başladığı renge döner; bu yüzden döngüdeki renk değişimi sayısı, yani kenar sayısı çifttir. Tek uzunlukta döngü olamaz. Öteki yön, BFS boyamasının çelişki üretmesinden çıkar. Aynı renkli iki komşu u ve v bulunursa, BFS ağacında ikisinin ortak bir atası vardır. Aynı renkte oldukları için başlangıca olan uzaklıklarının farkı çifttir; BFS'de komşu düğümlerin düzeyleri en çok 1 farklı olabildiği için düzeyleri eşittir. Ortak atadan u'ya ve v'ye inen iki yol eşit uzunluktadır; bunlara (u, v) kenarı eklenince tek uzunlukta bir döngü oluşur. Öyleyse tek döngü yoksa boyama çelişkisiz biter. ■

Testin maliyeti BFS'nin maliyetidir: $O(n + e)$. Yordam 6 düğümlü döngü ve $K_{3,3}$ için `TRUE`, üçgen için `FALSE` döndürdü; örnek gezinme çizgesi de iki parçalıdır, çünkü bütün döngüleri 4 ya da 6 kenarlıdır.

15.6 Karmaşıklık özeti

Tablo 15.23 bu notta incelenen işlemlerin maliyetlerini iki temel gösterim için bir arada verir. n düğüm sayısı, e kenar sayısı, d ilgili düğümün derecesidir. Bağlantılı basit bir çizgede $n - 1 \le e \le n(n-1)/2$'dir.

İşlem	Matris	Liste	Gerekçe
Bellek	$O(n^2)$	$O(n + e)$	Yönsüz listede $2e$ öğe
(u, v) var mı?	$O(1)$	$O(d_u)$	Tek hücre / liste taraması
Derece (yönsüz)	$O(n)$	$O(d)$	Satır / liste uzunluğu; yönlüde giriş için ters liste
Kenar sayısı	$O(n^2)$	$O(n + e)$	Derece toplamı / 2
DFS, BFS	$O(n^2)$	$O(n + e)$	Her düğüm bir kez, her liste bir kez
Bağlantılı bileşenler	$O(n^2)$	$O(n + e)$	Gezinmelerin toplamı; çarpım değil
Döngü, iki parçalılık	$O(n^2)$	$O(n + e)$	DFS ya da BFS üzerine sabit iş
Transpoz listeyi kurma	—	$O(n + e)$	Listeler bir kez gezilir

Tablo 15.23: Temel çizge işlemlerinin komşuluk matrisi ve komşuluk listesiyle maliyetleri.

Tablonun özü tek bir cümledir: komşuları bulmak matriste $O(n)$, listede $O(d)$'dir. Bütün gezinme tabanlı algoritmalar bu farkı n düğüm üzerinden toplar; matriste $n \cdot n$, listede $\Sigma d = 2e$ çıkar.

15.7 Sık yapılan hatalar

1. **Yönlü çizgeyi yönsüz sanmak.** $\langle u, v \rangle$ varken $\langle v, u \rangle$'yu da var saymak ya da yönlü çizgenin matrisini simetrik sanmak. G3'te $a[1][2] = 1$, $a[2][1] = 0$'dır.

2. **BFS'de işareti geç koymak.** Düğümü kuyruğa eklerken değil, çıkarırken işaretlemek. Aynı düğüm kuyruğa birden çok kez girer; örnek çizgede 7 dört kez yazılır.
3. **İşaret dizisini sıfırlamamak.** `ugranildi` küreseldir. İkinci bir gezinmeden önce `FALSE` yapılmazsa ikinci gezinme hiçbir düğüme giremez ve yalnız başlangıcı yazar.
4. **Bileşen analizinde maliyetleri çarpmak.** Dış `for` ile DFS'yi çarpıp *O(n(n + e))* demek. İşaretler sayesinde bütün DFS çağrılarının toplamı *O(n + e)*'dir.
5. **Belleği özensiz yönetmek.** `malloc` sonucunu denetlememek; `sizeof(oge_gost)` ile işaretçi boyutunda yer almak; program sonunda listeleri `free` etmemek; `free` ettikten sonra öğenin bağını okumak.
6. **Gereksiz yere sona eklemek.** Kenarı listenin sonuna ekleyip her seferinde listeyi baştan yürümek. Sıra önemli değilse *O(1)*'lik başa ekleme yeter; çizim de bu durumda kodun ürettiği ters sırayı göstermelidir.
7. **Matris parametresinde boyutu unutmak.** `int a[][]` yazmak derlenmez; ikinci boyut verilmelidir: `int a[][MAX_DUGUM]`.
8. **Sonsuz için `INT_MAX` seçmek.** Maliyet toplamlarında işaretli taşma olur; güvenli büyük bir sabit kullanılır.

15.8 Alıştırmalar ve çözümleri

Bu kesimdeki alıştırmalar sunumdaki sırayla verilir; her birinin ardından çözüm gelir.

15.8.1 Derece dizisi

Soru. Derece dizisi (3, 3, 2, 2, 1) olan basit, yönsüz bir çizge var mıdır? Kaç kenarı vardır?

Çözüm. Derecelerin toplamı 11'dir. Teorem 15.1'e göre toplam *2e*, yani çift olmalıdır. Böyle bir çizge yoktur. Aynı sonuç, tek dereceli düğümlerin sayısından da çıkar: 3, 3 ve 1 olmak üzere üç tek dereceli düğüm vardır, oysa bu sayı çift olmalıdır.

15.8.2 Sıralı gösterimde komşular

Soru. G4'ün dizi ile sıralı gösteriminde 5 numaralı düğümün komşularını hangi indisler tutar?

Çözüm. `dugum[5] = 18` ve `dugum[6] = 20`'dir. Komşular 18 ve 19 indisli hücrelerdedir: `dugum[18] = 4`, `dugum[19] = 6`. 5'in komşuları 4 ve 6, derecesi *20 − 18 = 2*'dir.

15.8.3 7'den başlayan gezinmeler

Soru. Sekiz düğümlü örnek çizgede DFS ve BFS 7'den başlatılırsa (listeler sıralı) çıktılar ne olur?

Çözüm. DFS: 7'nin listesi {3, 4, 5, 6}'dır; ilk komşu 3'e inilir. 3'ün işaretsiz ilk komşusu 1, 1'inki 0, 0'ınki 2, 2'ninki 5'tir. 5'in komşuları 2 ve 7 işaretlidir; 2'ye dönülür ve 6'ya inilir. Geri dönüşler 1'e kadar sürer ve 1'in listesindeki 4 bulunur. Çıktı `7 3 1 0 2 5 6 4`'tür. BFS: 7'nin komşuları 3, 4, 5, 6 sırayla yazılır ve kuyruğa girer. 3 alınınca 1, 4 alınınca hiçbir şey, 5 alınınca 2 eklenir. 6'nın komşuları işaretlidir. 1 alınınca 0 eklenir. Çıktı `7 3 4 5 6 1 2 0`'dır.

15.8.4 İşaret satırını taşımak

Soru. BFS kodunda `ugranildi[w->dugum] = TRUE` satırı `kuyruk_al`'dan sonraya taşınırsa ne olur?

Çözüm. Bir düğüm kuyrukta beklerken işaretsiz kalır. Başka bir düğümün listesinde görülünce yeniden yazılır ve yeniden kuyruğa eklenir. Örnek çizgede 7, kuyruktan çıkmadan önce 3, 4, 5 ve 6 üzerinden dört kez eklenir; `bfs(0)` `0 1 2 3 4 5 6 7 7 7 7` yazar. 7'den başlanırsa 1, 2 ve 0 da yinelenir: `7 3 4 5 6 1 1 2 2 0 0 0 0`. Çıktı yanlıştır ve iş *O(n + e)*'yi aşar.

15.8.5 Matrisle bileşen sayısı

Soru. Komşuluk matrisiyle verilen bir çizgenin bileşen sayısını bulan yordamı yazınız; karmaşıklığı nedir?

Çözüm. DFS'nin matris biçimi, bir düğümün komşularını satırını tarayarak bulur:

```c
void mdfs(int a[][MAX_DUGUM], int n, int v)
{
    int j;
    ugranildi[v] = TRUE;
    printf("%d ", v);
    for (j = 0; j < n; j++)            /* satirin tamami: O(n) */
        if (a[v][j] && !ugranildi[j])
            mdfs(a, n, j);
}

int matris_bilesen_sayisi(int a[][MAX_DUGUM], int n)
{
    int i, sayi = 0;
    for (i = 0; i < n; i++) ugranildi[i] = FALSE;
    for (i = 0; i < n; i++)
        if (!ugranildi[i]) {
            sayi++;
            mdfs(a, n, i);
            printf("\n");
        }
    return sayi;
}
```

`mdfs`, v'nin satırındaki her 1 için işaretsiz komşuya iner. `matris_bilesen_sayisi` işaretleri sıfırlar ve her işaretsiz düğümden yeni bir gezinme başlatıp sayacı artırır. G4'ün matrisine uygulandığında `0 1 3 2` ve `4 5 6 7` satırlarını yazar ve 2 döndürür. Her düğüm bir kez ziyaret edilir ve her ziyarette n hücrelik bir satır taranır: $O(n^2)$.

15.8.6 Giriş derecesini O(1)'de vermek

Soru. Yönlü bir çizgede giriş derecesini $O(1)$'de verebilmek için hangi yapı tutulur ve nasıl güncellenir?

Çözüm. Her düğüm için bir `giris[]` sayacı tutulur. $\langle u, v\rangle$ kenarı eklenirken `giris[v]++`, silinirken `giris[v]--` yapılır. Sorgu tek bir dizi okumasıdır. Gelen kenarların kendileri de gerekiyorsa ters liste ya da ortogonal liste tutulur; ters listenin uzunluğu giriş derecesidir, ama onu saymak $O(d)$'dir.

15.9 Özet

- Çizge $G = (V, E)$, boş olmayan bir düğüm kümesi ile boş olabilen bir kenar kümesidir. Yönsüzde $(u, v) = (v, u)$; yönlüde $\langle u, v\rangle \neq \langle v, u\rangle$.
- Komşu, alt çizge, yol, döngü, bağlantılılık ve derece temel kavramlardır. Derece her zaman belirli bir düğüm için söylenir; yönlüde giriş ve çıkış derecesinin toplamıdır.
- Derecelerin toplamı $2e$'dir. Bağlantılı bir çizgede $n - 1 \le e \le n(n-1)/2$; alt sınır gerekli ama yeterli değildir.
- Komşuluk matrisi $O(n^2)$ bellek ister; kenar sorgusu $O(1)$, komşu taraması $O(n)$'dir. Komşuluk listesi $O(n + e)$ bellek ister; komşu taraması $O(d)$'dir.
- Başa ekleme her kenarı $O(1)$'de yerleştirir ve listeyi gelen sıranın tersine çevirir. Yönlü çizgede giriş derecesi için ters liste $O(n + e)$'de kurulur.
- Sıralı dizi işaretçisiz, $n + 2e + 1$ tam sayılık bir gösterimdir ama değişikliğe uygun değildir. Ortogonal liste ve çok bağlı liste her kenarı tek öğede tutar.
- Ağırlıklı çizgede matrise ağırlık, listeye `agirlik` alanı eklenir; sonsuz için taşmayan büyük bir sabit seçilir.
- DFS özyineleme ya da yığınla derine iner; BFS kuyrukla katman katman ilerler. İkisi de listeyle $O(n + e)$, matrisle $O(n^2)$'dir. Sonuç sırası başlangıca ve liste sırasına bağlıdır.
- Bağlantılı bileşenler, her işaretsiz düğümden bir gezinme başlatılarak $O(n + e)$'de bulunur; dış döngü ile gezinmenin maliyetleri çarpılmaz.
- Yönsüz çizgede ebeveyn olmayan işaretli komşu, yönlü çizgede gri düğüme giden kenar döngü demektir. BFS ile iki renge boyama iki parçalılığı sınar; iki parçalı çizgede tek uzunlukta döngü yoktur.
- Yöntem: önce problem ve işlemler, sonra verinin nasıl depolanacağı, en son kod. İlk akla gelen gösterime güvenilmez.

15.10 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

çizge *graph* · düğüm *vertex* · kenar *edge* · yönlü çizge *directed graph* · komşu *adjacent* · alt çizge *subgraph* · yol *path* · döngü *cycle* · bağlantılı *connected* · derece *degree* · giriş derecesi *in-degree* · çıkış derecesi *out-degree* · tam çizge *complete graph* · komşuluk matrisi *adjacency matrix* · komşuluk listesi *adjacency list* · ortogonal liste *orthogonal list* · çok bağlı liste *adjacency multilist* · derinlik öncelikli arama *depth-first search* · genişlik öncelikli arama *breadth-first search* · kapsayan ağaç *spanning tree* · minimum kapsayan ağaç *minimum spanning tree* · bağlantılı bileşen *connected component* · döngü tespiti *cycle detection* · iki parçalı çizge *bipartite graph* · kuyruk *queue* · çağrı yığını *call stack* · en kısa yol *shortest path*

15.11 Kendini Dene

1. 9 düğümlü, bağlantılı ve basit bir çizgede en az ve en çok kaç kenar bulunabilir?

   Cevap

   En az $n - 1 = 8$, en çok $9 \cdot 8 / 2 = 36$. 8 kenarlı bağlantılı çizge bir ağaçtır; 36 kenarlı çizge tam çizge K_9'dur.

2. Yönsüz bir çizgenin komşuluk listelerinde toplam 30 öğe vardır. Çizgenin kaç kenarı vardır? Aynı çizge komşuluk matrisiyle tutulsaydı matriste kaç tane 1 bulunurdu?

   Cevap

   Her kenar iki listede göründüğü için $e = 30 / 2 = 15$. Matriste de her kenar simetrik iki hücreye yazılır: 30 tane 1.

3. G2'nin listeleri sıralı tutuluyor. `dfs(0)` ve `bfs(0)` ne yazar?

   Cevap

   DFS: `0 1 3 4 2 5 6` ; ağaçta kök önde gezinmenin aynısıdır. BFS: `0 1 2 3 4 5 6` ; düzey düzey gezinmenin aynısıdır.

4. Sekiz düğümlü örnek çizgede `bfs(6)` ne yazar?

   Cevap

   `6 2 7 0 5 3 4 1` . 6'nın komşuları 2 ve 7'dir. 2 alınınca 0 ve 5, 7 alınınca 3 ve 4 eklenir. 0 alınınca 1 eklenir; kalanların komşuları işaretlidir.

5. Sekiz düğümlü örnek çizge ($e = 10$) dizi ile sıralı gösterimle tutuluyor. Dizi kaç hücrelidir ve 7'nin komşuları hangi indislerdedir?

   Cevap

   $n + 2e + 1 = 8 + 20 + 1 = 29$ hücre. Başlangıç indisleri 9, 11, 14, 17, 19, 21, 23, 25, 29'dur. `dugum[7] = 25` , `dugum[8] = 29` : 7'nin komşuları 25–28 indislerinde, 3, 4, 5 ve 6'dır.

6. G1'in kenarları (0,1), (0,2), (0,3), (1,2), (1,3), (2,3) sırasıyla `ml_ekle` ile çok bağlı listeye ekleniyor. `komsulari_yaz(0)` ne yazar?

   Cevap

   `3 2 1` . Her kenar listelerin başına girer; 0'ın listesinde en son eklenen (0,3) öndedir.

7. G5'e (11 düğüm; kenarlar (1,4) ve (7,8)) `bitisik_bilesenler(11)` uygulanırsa kaç satır yazılır? `if` 'e kaç kez girilir?

   Cevap

   9 satır: `0` , `1 4` , `2` , `3` , `5` , `6` , `7 8` , `9` , `10` . `if` 'e 9 kez girilir; `i` = 4 ve `i` = 8'de düğümler önceki gezinmelerde işaretlenmiştir.

8. `dongu_var`'da `else if (w->dugum != ata)` yerine yalnız `else` yazılırsa ağaç olan G2 için yordam ne döndürür? Neden?

Cevap

`TRUE` döndürür, oysa G2'de döngü yoktur. `dongu_var(1, 0)`, 1'in listesindeki ilk öğe olan 0'ı işaretli görür ve onu geri kenar sayar. Yönsüz çizgede gelinen kenar, öteki ucun listesinde de bulunur; ebeveyn denetimi bu kenarı ayırmak içindir.

15.12 İleri okuma

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, Bölüm 6. Bu notun gösterimleri, `dfs`, `bfs` ve bağlantılı bileşen yordamları oradaki biçimiyle izlenmiştir; çok bağlı liste ve sıralı gösterimin ayrıntıları da oradadır.
- T. H. Cormen ve ark., *Introduction to Algorithms*, 4. baskı, Bölüm 20. DFS'nin beyaz/gri/siyah renklendirmesi, kenar sınıflandırması ve BFS'nin en kısa yol özelliğinin kanıtları verilir.
- R. Sedgewick, *Algorithms in C, Part 5: Graph Algorithms*, 3. baskı. Çizge gösterimleri ve gezinmeler C koduyla, büyük çizgeler üzerindeki deneylerle birlikte ele alınır.

15.13 Kaynaklar

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. bs., Silicon Press, 2008. Bölüm 6, Graphs.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. bs., MIT Press, 2022. Bölüm 20, Elementary Graph Algorithms.
3. R. Sedgewick, *Algorithms in C, Part 5: Graph Algorithms*, 3. bs., Addison-Wesley, 2002.

KISIM IV · BÖLÜM 16

Kümeler ve Union-Find

Birçok problem aynı iki soruya indirgenir: iki öğe aynı gruba mı ait, değilse iki grup nasıl tek grupta toplanır? Ağ bağlantısı, çizgede bileşen sayma, eşdeğerlik bildirimleri ve minimum kapsayan ağaç kurma bu soruları milyonlarca kez sorar. Ortak öğesi olmayan kümeleri tutan ve yalnız bu iki işlemi destekleyen yapıya union-find denir. Bu not, kümeleri çocuktan ebeveyne bağlı ağaçlarla gösterir ve bütün ormanı tek bir tam sayı dizisinde saklar. İlk sürümün hangi işlem dizisinde zincire dönüştüğü ve maliyetin nasıl $O(n^2)$'ye çıktığı adım adım gösterilir. Ardından iki iyileştirme gelir: ağaç derinliğini $\lfloor \lg m \rfloor + 1$ ile sınırlayan ağırlıklı kural ve bulma yolunu köke bağlayan azaltma kuralı. İkisi birlikte işlem başına maliyeti, ters Ackermann fonksiyonu yüzünden uygulamada 4'ü aşmayan bir çarpana indirir. Her yordamın C kodu, satır satır anlatımı, kod izlemesi, ölçümü ve karmaşıklık analizi verilir.

Kod (C): `github.com/sercankulcu/data-structures-and-algorithms-c/Bolum16`

ÖĞRENME ÇIKTILARI

Modellemek
İkili ayrık kümeleri, kökü küme adı olan ve bağları çocuktan ebeveyne giden ağaçlarla göstermek.

Gerçeklemek
Tek bir ebeveyn dizisiyle `Kume_Bul` ve `Birlesim` yordamlarını yazmak; diziyi `malloc` ile kurup `free` ile geri vermek.

Kötü durumu bulmak
Hangi işlem dizisinin ağacı zincire çevirdiğini ve toplam maliyetin neden $O(n^2)$ olduğunu göstermek.

İyileştirmek
Ağırlıklı kuralı ve azaltma kuralını uygulamak; derinlik lemmasını kanıtıyla yorumlamak.

Analiz etmek
Tarjan–van Leeuwen sonucunu ve ters Ackermann fonksiyonunun neden "neredeyse sabit" anlamına geldiğini açıklamak.

Kullanmak
Bağlantılı bileşen sayma, eşdeğerlik sınıfları ve Kruskal algoritmasında union-find'ı uygulamak.

16.1 Ayrık kümeler

Bir sosyal ağda kim kimin arkadaşı, bir şehir ağında hangi kavşaklar birbirine yolla bağlı, bir devrede hangi uçlar aynı iletkene lehimli? Bu soruların hepsinde öğeler gruplara ayrılır ve her öğe tek bir gruptadır. Gruplar zamanla birleşir: iki kavşak arasına yol yapılınca iki bölge tek bölge olur. Grupların bölündüğü görülmez. Bu tür problemler için gereken yapı çok sadedir. Yalnız iki işlemi destekler, ama bu iki işlemi son derece hızlı yapar. Az sayıda satırla yazılan bu yapı, çizge algoritmalarının birçoğunda bir alt yapı olarak karşımıza çıkar. Önce iyi öğrenilmelidir ki gerektiğinde akla gelsin ve kullanılabilsin.

Programlamaya geçmeden önce problemi programlama bilmeyen biri gibi, matematiğin diliyle doğru ifade etmek gerekir. Doğru kurulmuş bir tanım, hangi işlemlerin gerektiğini ve hangilerinin gereksiz olduğunu baştan gösterir.

16.1.1 Ayrık küme ve orman gösterimi

Öğeler $0, 1, \ldots, n-1$ sayılarıyla adlandırılır. Bu sayılar bir öğrencinin numarası, bir bilgisayarın ağdaki sırası ya da bir kavşağın kodu olabilir. Öğelerin asıl verisi başka bir dizide tutulur; union-find yalnız numaralarla çalışır.

Tanım 16.1

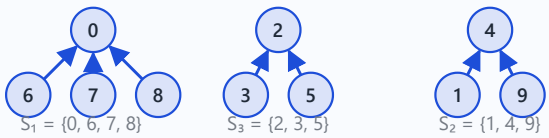
Evren $U = \{0, 1, \ldots, n-1\}$ olsun. $S_1, S_2, \ldots, S_k$ kümeleri U'nun alt kümeleriyse ve her $i \neq j$ için $S_i \cap S_j = \emptyset$ ise bu kümelere **ikili ayrık kümeler** (*pairwise disjoint sets*) denir. Kısaca **ayrık kümeler** (*disjoint sets*) diye anılır. Ayrık kümeler üzerinde birleşim ve küme bulma işlemlerini destekleyen veri yapısına **union-find** denir.

Tanımdaki koşul, hiçbir öğenin iki kümede birden bulunmadığını söyler. Her öğenin kümesi tektir; bu yüzden "öğe hangi kümede?" sorusunun her zaman tek bir cevabı vardır.

Örnek 16.1

$n = 10$ için $S_1 = \{0, 6, 7, 8\}$, $S_2 = \{1, 4, 9\}$ ve $S_3 = \{2, 3, 5\}$ kümeleri ikili ayrıktır. $S_1 \cap S_2$, $S_2 \cap S_3$ ve $S_1 \cap S_3$ kesişimlerinin üçü de boştur. Üç kümenin birleşimi evrenin tamamını, yani on öğeyi verir. Bu notun ilk örneklerinin hepsi bu üç kümeyle çalışır.

Kümeleri bilgisayarda nasıl göstermeli? Seçilen yol, her kümeyi bir ağaçla göstermektir. Kümenin öğelerinden biri kök olur, ötekiler onun altına yerleşir. Sıradan ağaçlardan önemli bir farkı vardır: bağlar **çocuktan ebeveyne** doğrudur. Her düğüm yalnız kendi ebeveynini bilir; çocuklarını bilmez. Şekil 16.1'deki oklar bu yüzden yukarı bakar. Birbirinden ayrık ağaçların oluşturduğu topluluğa **orman** (*forest*) denir. Ayrık kümelerin gösterimi de bir ormandır.



Şekil 16.1: Örnek 16.1'deki üç küme ve ağaçları. Oklar çocuktan ebeveyne gider. Köklerin (0, 2 ve 4) ebeveyni yoktur.

Ağaçlardaki yerleşimin bir anlamı yoktur. 6, 7 ve 8'in 0'ın altında durması, 0'ın bu kümede "özel" olduğunu göstermez. Aynı küme, 7'nin kök olduğu ve ötekilerin ona bağlandığı bir ağaçla da gösterilebilirdi. Ağacın görevi yalnız "bu öğeler bir aradadır" bilgisini taşımaktır. Bu serbestlik daha sonra çok işe yarar: ağacın biçimini maliyeti düşürecek yönde istediğimiz gibi seçebiliriz.

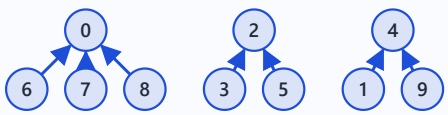
Bağların neden yukarı doğru olduğu da işlemlerden anlaşılır. Union-find hiçbir zaman "bu kümenin öğelerini listele" diye sormaz. Sorulan hep "bu öğe hangi kümede?" sorusudur. Bu soruya bir öğeden başlayıp köke kadar tırmanarak cevap verilir. Tırmanmak için her düğümün ebeveynini bilmesi yeter. Çocuk listesi tutmak hem bellek harcar hem de hiçbir işlemde kullanılmaz.

16.1.2 Birleşim ve küme bulma

Yapının desteklediği iki işlem şunlardır:

1. **Ayrık küme birleşimi** (*union*). S_i ve S_j ayrık iki kümeyse birleşimleri $S_i \cup S_j = \{x : x \in S_i \vee x \in S_j\}$ kümesidir. Birleşimden sonra S_i ile S_j artık ayrı ayrı bulunmaz; yerlerini tek bir küme alır. Örneğin $S_1 \cup S_2 = \{0, 6, 7, 8, 1, 4, 9\}$ olur.
2. **Küme bulma** (*find*). `Küme_bul(i)`, i öğesini içeren kümeyi bulur. Kümenin adı olarak ağacın kökü döndürülür.

Birleşim ağaçlarla çok kolay yapılır. İki ağacın köklerinden biri, ötekinin çocuğu yapılır. Bunun için tek bir bağ yazmak yeter: köklerden birinin ebeveyn bilgisi öteki kökü gösterecek biçimde günlenir. Şekil 16.2 aynı birleşimin iki biçimini gösterir. Altta soldaki şekilde 4'ün ebeveyni 0 yapılmış, S_2 ağacı 0'ın alt ağacı olmuştur. Altta sağdaki şekilde ise 0'ın ebeveyni 4 yapılmıştır. İki sonuç da aynı kümeyi gösterir ve ikisi de doğrudur. Ama ağaçların biçimi farklıdır. Hangisinin seçileceği, sonraki küme bulma işlemlerinin maliyetini belirler. Bu notun büyük kısmı bu seçimin nasıl yapılacağı üzerinedir.





Şekil 16.2: Üstte başlangıç: kökler 0, 2 ve 4. Altta solda $S_1 \cup S_2$: 4'ün ebeveyni 0 yapılır. Altta sağda $S_2 \cup S_1$: bu kez 0'ın ebeveyni 4 yapılır. Kırmızı ok yeni yazılan bağdır.

Sağdaki biçimde 6, 7 ve 8 öğelerinin köke uzaklığı 1'den 2'ye çıkmıştır. Soldaki biçimde ise yalnız 1, 4 ve 9'un uzaklığı birer artmıştır. Bir öğenin kümesini bulmak, o öğeden köke kadar yürümek demektir. Köke uzaklık arttıkça bulma pahalılaşır. Ağırlıklı kural bu küçük gözleme dayanır.

16.1.3 Kümenin adı: kök

Kümelerin S_1, S_2 gibi adları insanlar içindir. Algoritmalar her küme adını ağacın kökü ile ilişkilendirir. "S_1 kümesi" yerine "0 nolu küme" denir. Böylece `Küme_bul` yordamının döndürdüğü kök numarası kümenin adı olur.

Küme adı	Bağ (kök)
S_1	→ 0
S_2	→ 4
S_3	→ 2

Tablo 16.1: Küme adlarından köklere giden bağ tablosu. Adlar metin olarak gerekiyorsa böyle ayrı bir tablo tutulur.

Kökün ad olarak kullanılması önemli bir sonuç doğurur. İki öğenin aynı kümede olup olmadığı, köklerinin karşılaştırılmasıyla anlaşılır. `Küme_bul(8) == Küme_bul(6)` sorusu "8 ile 6 aynı kümede mi?" demektir. İki çağrı da 0 döndürdüğü için cevap evettir. `Küme_bul(8) == Küme_bul(9)` ise 0 ile 4'ü karşılaştırır; cevap hayırdır.

Kümenin adı birleşimle değişebilir. Şekil 16.2'nin sağındaki birleşimden sonra 8'in kümesi artık 0 değil, 4'tür. Bu yüzden bir öğenin kümesi bir değişkende saklanıp sonra kullanılmamalıdır. Her soru, o anki ağaçlar üzerinde yeniden `Küme_bul` çağrılarak cevaplanır.

> **Sık yapılan hata**
>
> Kök numarasını öğenin değişmez kimliği sanmak. `int k = Küme_bul(8);` yazıp birkaç birleşimden sonra `k` 'yı kullanmak yanlış sonuç verebilir. Birleşimde kökler yer değiştirdiği için 0 artık bir kök olmayabilir. Kök yalnız sorulduğu andaki küme adıdır.

16.2 Dizi ile gerçekleme

Model hazır: her küme bir ağaç, her ağacın kökü kümenin adı, bağlar çocuktan ebeveyne. Şimdi bu modelin C'de nasıl saklanacağına karar vermek gerekir.

16.2.1 Gerçekleme seçenekleri

Akla ilk gelen yol, her kümeyi bir bağlı liste olarak tutmaktır. İki listeyi uç uca eklemek, son düğümün işaretçisini değiştirmekle biter; birleşim *O(1)*'dir. Ama bir öğenin hangi listede olduğunu bulmak için listeleri baştan sona taramak gerekir. Küme bulma *O(n)* olur. Kümeleri sıralı dizilerde tutmak aramayı hızlandırır, ama her birleşimde iki sıralı diziyi birleştirmek ya da yeniden sıralamak gerekir. Bunun maliyeti *O(n lg n)*'ye kadar çıkar.

Yöntem	Birleşim	Küme bul	Sorun
Bağlı liste	O(1)	O(n)	Arama yavaş
Sıralı tutmak	—	—	Sıralamak O(n lg n)

Yöntem	Birleşim	Küme bul	Sorun
Ebeveyn dizisi (ağaç)	O(1)	Ağaç derinliği	Derinliği küçük tutmalı

Tablo 16.2: Ayrık kümeleri gerçeklemenin üç yolu.

Üçüncü satır en umut vericisidir. Birleşim yine tek bağ yazmaktır. Küme bulmanın maliyeti ise ağacın derinliği kadardır. Derinlik küçük tutulabilirse küme bulma da hızlanır.

Ağaçlar genellikle düğüm yapıları ve işaretçilerle kurulur. Burada buna gerek yoktur. Öğeler zaten *0*'dan *n − 1*'e numaralıdır; numara doğrudan dizi indisi olarak kullanılabilir. Her düğüm yalnız tek bir bilgiye gereksinim duyar: ebeveyninin numarası. Öyleyse bütün orman, *n* elemanlı tek bir tam sayı dizisine sığar. En güvenilir, en sağlam ve en hızlı veri yapısı olan diziyle gerçekleme yapmak mümkündür. Dizide bir öğeye erişim *O(1)*'dir, işaretçi kovalamak gerekmez, `malloc` çağrısı tek seferdir ve bellek ardışık olduğu için önbellek de iyi kullanılır.

16.2.2 Ebeveyn dizisi

Tanım 16.2

Ebeveyn dizisi (*parent array*), *n* elemanlı bir `int` dizisidir. `ata[i]`, *i* düğümünün ağaçtaki ebeveyninin, yani bir üstündeki düğümün numarasıdır. *i* bir kökse `ata[i]` negatif bir değer taşır. İlk sürümde bu değer −1'dir.

Bir düğümün hemen üstündeki düğüme ebeveyn denir; koddaki `ata` dizisi bu ebeveyn bağlarını tutar. Kökün bir ebeveyni olmadığı için onun hücresine geçerli bir indis yazılamaz. Düğüm numaraları 0 ya da pozitif olduğundan negatif bir değer hiçbir düğümle karışmaz ve "burası kök" işareti olarak güvenle kullanılır. Şekil 16.3 örnek ormanın ebeveyn dizisini gösterir.



Şekil 16.3: Örnek 16.1'in ebeveyn dizisi. `ata[i]`, *i*'nin ebeveynidir; köklerde −1 durur.

Diziyi okumak kolaydır. 1 ve 9'un ebeveyni 4'tür; `ata[1]` ve `ata[9]` hücrelerinde 4 yazar. 3 ve 5'in ebeveyni 2'dir. 6, 7 ve 8'in ebeveyni 0'dır. 0, 2 ve 4 kümelerin kökleridir: `ata[0] = ata[2] = ata[4] = -1`. C'de bu dizi şöyle tanımlanır:

```
#define N 10
int ata[N] = { -1, 4, -1, 2, -1, 2, 0, 0, 0, 4 };
```

Bellekte bu dizi 10 ardışık `int` hücresidir. 64 bitlik bir Windows ya da Linux makinesinde `int` 4 bayttır ve dizi 40 bayt yer tutar. Dizi 0x1000 adresinde başlıyorsa `ata[i]` hücresi *0x1000 + 4i* adresindedir. Tablo 16.3 bu görünümü verir. İşaretçili bir ağaçta her düğüm ayrı bir `malloc` bloğu olur ve en az bir işaretçi (8 bayt) taşır. Burada ise düğüm başına 4 bayt yeter ve bütün ağaçlar tek bir blokta durur.

Adres	0x1000	0x1004	0x1008	0x100C	0x1010	0x1014	0x1018	0x101C	0x1020	0x1024
İndis	0	1	2	3	4	5	6	7	8	9
Değer	−1	4	−1	2	−1	2	0	0	0	4

Tablo 16.3: `ata` dizisinin bellekteki görünümü (dizinin 0x1000'de başladığı varsayılmıştır; adres makineye göre değişir).

Ebeveyn dizisi bir ağacı tek başına tam olarak tanımlar. Dizi verilince ağaç çizilebilir: her *i* için *i*'den `ata[i]`'ye bir ok çizmek yeter. Ters yön de doğrudur: her ağaç tek bir ebeveyn dizisi verir. Ancak dizi bir düğümün çocuklarını doğrudan söylemez. 0'ın çocuklarını bulmak için bütün diziyi tarayıp değeri 0 olan hücreleri toplamak gerekir. Union-find bu soruyu hiç sormadığı için bu eksiklik sorun olmaz.

16.2.3 Ebeveyn dizisini dinamik bellekte kurmak

Sunumdaki kodlar `ata` dizisini `#define N 10` boyutlu küresel bir dizi olarak tanımlar. Bu, yordamları kısa tutar ve örnekler için yeterlidir. Gerçek bir programda öğe sayısı ise çoğunlukla çalışma anında belli olur: bir dosyadan okunan çizgenin düğüm sayısı derleme anında bilinmez. Bu durumda dizi `malloc` ile ayrılır. Aşağıdaki kod öğe sayısını ve diziyi tek bir yapıda toplar:

```
typedef struct {
  int  n;      /* öğe sayısı */
  int *ata;    /* ata[i]: i'nin ebeveyni; kökte -(düğüm sayısı) */
} kumeler;

kumeler *kumeler_olustur(int n) {
  kumeler *k = malloc(sizeof *k);
  if (k == NULL) return NULL;            /* bellek yok */
  k->ata = malloc(n * sizeof *k->ata);
  if (k->ata == NULL) {                  /* dizi alınamadı */
    free(k);                             /* yarım yapıyı geri ver */
    return NULL;
  }
  k->n = n;
  for (int i = 0; i < n; i++)
    k->ata[i] = -1;                      /* her öğe tek öğeli küme */
  return k;
}

void kumeler_sil(kumeler **kp) {
  if (*kp == NULL) return;
  free((*kp)->ata);                      /* önce iç dizi */
  free(*kp);                             /* sonra yapının kendisi */
  *kp = NULL;                            /* çağıranın işaretçisi boşalır */
}
```

`kumeler_olustur` iki ayrı blok ayırır. Birincisi yapının kendisidir: bir `int` ve bir işaretçi. 64 bitlik makinede `sizeof(kumeler)` 16 bayttır: 4 bayt `n`, 4 bayt hizalama boşluğu ve 8 bayt işaretçi. İkinci blok *n* tam sayılık ebeveyn dizisidir. `malloc(n * sizeof *k->ata)` ifadesindeki `sizeof *k->ata`, dizinin öğe tipinin boyutunu verir. Tip sonradan `long` 'a çevrilse bile bu satırın değişmesi gerekmez.

Her `malloc` çağrısından sonra `NULL` denetimi yapılır. İlk blok alınamazsa yapılacak bir şey yoktur; yordam `NULL` döndürür. İkinci blok alınamazsa durum daha incedir. Bu noktada yapı için ayrılmış bir blok vardır. Yordam doğrudan `NULL` döndürse o blok kimsenin elinde kalmaz ve bir daha geri verilemez. Bu bir **bellek sızıntısıdır** (*memory leak*). Kod bu yüzden önce `free(k)` ile yarım kurulmuş yapıyı geri verir, sonra `NULL` döndürür. Son döngü her öğeyi tek öğeli bir küme yapar: başlangıçta bütün hücreler −1'dir.

`kumeler_sil` blokları ayrılma sırasının tersiyle geri verir. Önce iç dizi, sonra yapı serbest bırakılır. Sıra ters olsaydı, yani önce `free(*kp)` çağrılsaydı, `(*kp)->ata` ifadesi geri verilmiş bir bloğu okurdu. Bu tanımsız davranıştır. Yordam parametre olarak **işaretçiye işaretçi** (*pointer to pointer*) alır: `kumeler **kp`. Böylece son satırda çağıranın kendi değişkenine `NULL` yazabilir. Çağıran `kumeler_sil(&k)` dedikten sonra `k` artık `NULL` 'dır. Yanlışlıkla ikinci kez `kumeler_sil(&k)` çağrılırsa ilk satırdaki denetim yüzünden hiçbir şey olmaz. Parametre yalnız `kumeler *k` olsaydı yordam kendi kopyasını `NULL` yapabilirdi, çağıranın `k` 'sı ise geri verilmiş bloğu göstermeye devam ederdi. Böyle bir işaretçiye **sarkan işaretçi** (*dangling pointer*) denir.

Bu iki yordam sayaçlı bir `malloc` / `free` sarmalayıcısıyla denendi: oluşturma ve silme sonunda 2 ayırma ve 2 geri verme sayıldı, sızıntı görülmedi. Notun geri kalanında, sunumla aynı kalmak için yordamlar küresel `ata` dizisini kullanır. Dinamik yapıya geçmek için her yordama `kumeler *k` parametresi eklenir ve `ata[i]` yerine `k->ata[i]` yazılır; mantık hiç değişmez.

16.2.4 İlk sürüm: `Kume_Bul_1` ve `Birlesim_1`

İlk sürüm, modelin en doğrudan C karşılığıdır:

```
int Kume_Bul_1(int i) {
  for (; ata[i] >= 0; i = ata[i])
    ;                /* negatife dek ilerle */
  return i;
}
void Birlesim_1(int i, int j) {
  ata[i] = j;        /* i kümesini j'ye bağla */
}
```

`Kume_Bul_1` tek bir `for` döngüsüdür. Döngünün başlatma kısmı boştur, çünkü başlangıç düğümü zaten `i` parametresidir. Koşul `ata[i] >= 0`, "*i* bir kök değil" demektir. Koşul doğru oldukça `i = ata[i]` ile bir üst düğüme çıkılır. Döngünün gövdesi boştur; tek başına duran noktalı virgül bunu gösterir. Bütün iş koşul ve güncelleme kısımlarında yapılır. Döngü ebeveyn bilgisi negatif olan ilk düğümde, yani kökte durur. Yordam bu kökü döndürür.

Boş gövdeli döngü C'de yaygın bir kalıptır, ama okuması kolay değildir. Noktalı virgülün ayrı bir satıra yazılması ve yanına açıklama konması bilinçli bir seçimdir. Noktalı virgül `for` satırının sonuna bitişik yazılsaydı, okuyan biri onu bir yazım hatası sanabilirdi.

`Birlesim_1` tek bir atamadır: `ata[i] = j`. *i* kökünün ebeveyni artık *j*'dir; *i*'nin ağacı bütünüyle *j*'nin altına girer. Yordamın bir **ön koşulu** vardır: *i* ve *j* farklı kümelerin kökleri olmalıdır. Kök olmayan bir öğe verilirse önce `Kume_Bul_1` ile kökü bulunur. Koşul bozulursa ya bir ağaç ikiye bölünür ya da kök kendini gösterir ve bulma yordamı sonsuz döngüye girer.

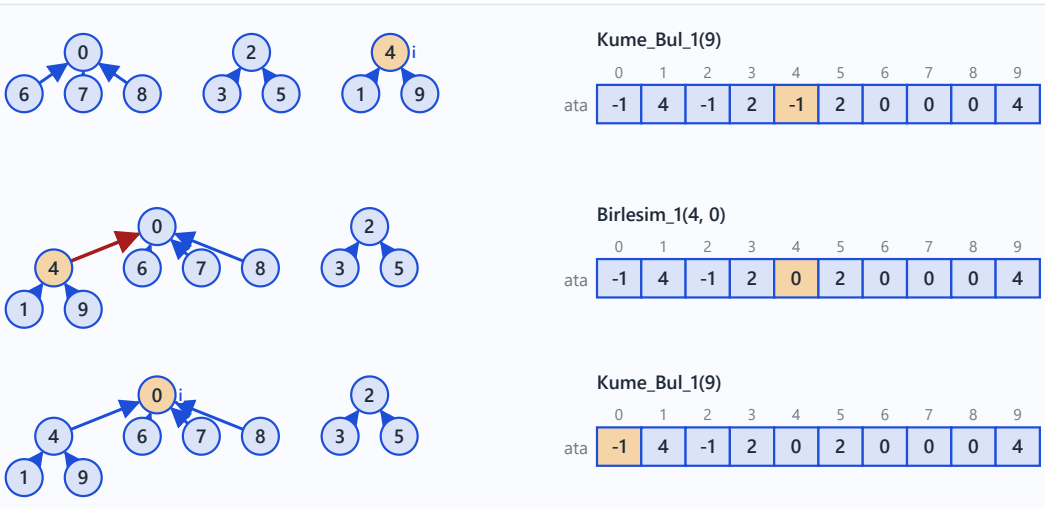
Maliyetler açıktır. `Birlesim_1` tek atama yapar: $\Theta(1)$. `Kume_Bul_1(i)`, *i*'den köke giden yoldaki her kenar için döngüyü bir kez döner. *i*'nin köke uzaklığı *d* ise *d* ilerleme ve *d* + 1 karşılaştırma yapar: $\Theta(d + 1)$. Bu notta "ilerleme" sözcüğü `i = ata[i]` atamasının bir kez yapılması anlamında kullanılır. En kötü durumda *d*, ağacın en derin düğümünün köke uzaklığıdır. Ek bellek her iki yordamda da $O(1)$'dir: yalnız `i` değişkeni kullanılır.

16.2.5 Kod izleme

Örnek ormanda sırasıyla `Kume_Bul_1(9)`, `Birlesim_1(4, 0)` ve yeniden `Kume_Bul_1(9)` çağrılsın. Tablo 16.4 her adımda `i`'nin ve dizinin durumunu verir. Şekil 16.4'te üç önemli an çizilmiştir.

Adım	İşlem	`i`	Denetim	Sonuç
1	`Kume_Bul_1(9)`	9	`ata[9] = 4 ≥ 0`	`i = 4`
2		4	`ata[4] = -1 < 0`	Kök; `return 4`
3	`Birlesim_1(4, 0)`	—	—	`ata[4] = 0`
4	`Kume_Bul_1(9)`	9	`ata[9] = 4 ≥ 0`	`i = 4`
5		4	`ata[4] = 0 ≥ 0`	`i = 0`
6		0	`ata[0] = -1 < 0`	Kök; `return 0`

Tablo 16.4: `Kume_Bul_1(9)`, `Birlesim_1(4, 0)`, `Kume_Bul_1(9)` izlemesi. Başlangıçta `ata = [-1, 4, -1, 2, -1, 2, 0, 0, 0, 4]`.



Şekil 16.4: Kod izlemesinin üç anı. Üstte ilk `Kume_Bul_1(9)` 4'te durur. Ortada `Birlesim_1(4, 0)` tek hücreyi değiştirir: `ata[4] = 0`. Altta ikinci `Kume_Bul_1(9)` bir adım daha yürüyüp 0'da durur.

İlk çağrı tek ilerlemeyle 4'e ulaşır: 9, 4 nolu kümededir. Birleşim yalnız bir hücreyi değiştirir. Ama bu tek atama, 4'ün altındaki bütün düğümlerin köke uzaklığını bir artırır. İkinci bulma çağrısı bu yüzden iki ilerleme yapar ve 0'ı döndürür: artık 9, 0 nolu kümededir. Bulmanın maliyeti, aranan düğümün derinliğine eşittir. Birleşim ucuzdur, ama bedeli sonraki bulma işlemlerine yansır.

Tablodaki değerler sunumdaki kodla derlenen bir sürücüyle doğrulandı: ilk çağrı 4'ü, ikinci çağrı 0'ı döndürdü. Aynı sürücüde `Kume_Bul_1(8) == Kume_Bul_1(6)` karşılaştırması da 1 (doğru) verdi.

16.3 İlk sürümün analizi

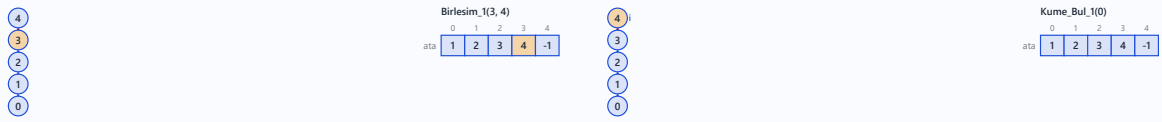
Bir algoritmanın doğru olması yetmez. Etkili olmalı ve yazılım mühendisliğinin kurallarına göre yazılmalıdır. Bunlar da yetmez: hangi verinin algoritmayı en çok zorlayacağı sorulmalıdır. Arka planda ne olursa olsun kod en fazla ne kadar çalışır? Bu soruyu sormak ve cevaplamak, bir programlama dilini bilen biriyle bilgisayar mühendisini ayıran şeydir. İşveren "hepiniz C biliyorsunuz, aynı maaşı alın" derse mühendisin cevabı hazırdır: bir problem verilsin, nasıl çözüldüğüne ve çözümün en kötü durumda ne kadar sürdüğüne bakılsın.

16.3.1 Kötü işlem dizisi

Tek öğeli n kümeyle başlayalım: $S_i = \{i\}$, her i için `ata[i] = -1`. Ardından şu işlem dizisi uygulansın:

```
/* n tek öğeli kümeyle başla */
Birlesim_1(0, 1);
Birlesim_1(1, 2);
...
Birlesim_1(n-2, n-1);
Kume_Bul_1(0);
Kume_Bul_1(1);
...
Kume_Bul_1(n-1);
```

Her birleşim, bir önceki adımda oluşan ağacın kökünü yeni bir tek öğeli kümenin altına bağlar. `Birlesim_1(0, 1)` 0'ı 1'in altına koyar. `Birlesim_1(1, 2)` 1'i, dolayısıyla 0 ile 1'den oluşan ağacı, 2'nin altına koyar. Her birleşimin iki parametresi de gerçekten köktür; ön koşul bozulmaz. Ama ağaç her adımda bir düzey uzar. Şekil 16.5 $n = 5$ için sonucu gösterir.



Şekil 16.5: $n = 5$ için dört `Birlesim_1` çağrısından sonra ağaç bir zincirdir (solda): `ata = [1, 2, 3, 4, -1]`. Sağda `Kume_Bul_1(0)` dördüncü ilerlemesini yapmış, köke ulaşmıştır.

Birleşimlerin her biri sabit sürer. Sorun bulma işlemlerinde ortaya çıkar. Tablo 16.5, zincir üzerinde `Kume_Bul_1(0)`'dan `Kume_Bul_1(4)`'e kadar her çağrının kaç ilerleme yaptığını verir.

Çağrı	Düğümün düzeyi	İzlenen yol	İlerleme
`Kume_Bul_1(0)`	5	0 → 1 → 2 → 3 → 4	4
`Kume_Bul_1(1)`	4	1 → 2 → 3 → 4	3
`Kume_Bul_1(2)`	3	2 → 3 → 4	2
`Kume_Bul_1(3)`	2	3 → 4	1
`Kume_Bul_1(4)`	1	4	0
Toplam			10 = 5 · 4 / 2

Tablo 16.5: $n = 5$ zincirinde bulma çağrıları. Düzey, kökün 1. düzey sayıldığı konumdur.

16.3.2 Toplam maliyet

Tablodaki örüntü her n için aynıdır. Zincirde i düğümü $(n - i)$. düzeydedir. Kökten $n - 1 - i$ kenar uzaktadır.

Teorem 16.1 (İlk sürümün kötü durumu)

n tek öğeli kümeye `Birlesim_1(i, i+1)` çağrıları *i = 0, 1, …, n − 2* sırasıyla uygulansın. Ardından her öğe için bir kez `Kume_Bul_1` çağrılsın. Toplam ilerleme sayısı *n(n − 1)/2*'dir. Bu işlem dizisinin maliyeti $\Theta(n^2)$'dir.

İspat. *n − 1* birleşimin her biri tek atama yapar; toplamları $\Theta(n)$'dir. Birleşimlerden sonra `ata[i] = i + 1` (*i < n − 1*) ve `ata[n-1] = -1` olur. `Kume_Bul_1(i)` bu yüzden *i, i + 1, …, n − 1* düğümlerinden geçer ve *n − 1 − i* ilerleme yapar. Toplam $\sum_{i=0}^{n-1} (n-1-i) = (n-1) + (n-2) + \dots + 1 + 0 = n(n-1)/2$'dir. Toplam maliyet $\Theta(n) + \Theta(n^2) = \Theta(n^2)$ olur. ∎

Tek bir `Kume_Bul_1` en kötü durumda $O(n)$ sürer; *n* tanesi toplamda $O(n^2)$'ye çıkar. Bu, union-find'ın kullanıldığı milyonlarca öğeli problemlerde kabul edilemez. Tablo 16.6'daki ölçüm büyümeyi somutlaştırır. *n* iki katına çıkınca ilerleme sayısı ve süre yaklaşık dört katına çıkar; bu karesel büyümenin belirtisidir.

n	İlerleme (ilk sürüm)	*n(n − 1)/2*	Süre	İlerleme (ağırlıklı)
1.000	499.500	499.500	≈ 0,001 s	999
10.000	49.995.000	49.995.000	≈ 0,13 s	9.999
20.000	199.990.000	199.990.000	≈ 0,5–0,6 s	19.999
40.000	799.980.000	799.980.000	≈ 2,0–2,7 s	39.999

Tablo 16.6: Zincir üzerinde *n* bulma çağrısının ölçümü (`zig cc -O2`, x86-64; süreler makineye göre değişir). Son sütun aynı birleşim isteklerinin ağırlıklı kuralla yapıldığı durumdur.

Bu işlem dizisi ağacın yapısını bozar. Ama suçlu birleşim işleminin kendisi değildir. Asıl sorun, birleşimde hangi ağacın hangisinin altına bağlandığıdır. Her adımda büyük ağaç (zincir) tek düğümlü küçük ağacın altına bağlanmıştır. Sıra ters olsaydı, yani her yeni düğüm büyük ağacın köküne bağlansaydı, ağaç hep iki düzeyli kalırdı. Tablonun son sütunu bunu gösterir: aynı birleşim istekleri ağırlıklı kuralla yapılınca ağaç iki düzeyde kalır ve *n* bulmanın toplamı yalnız *n − 1* ilerlemedir. Ağırlıklı kural tam olarak bu fikre dayanır.

16.3.3 En kötü durumu test etmek

En kötü durum analizinde kod, çakılacağı ya da işletiminin en uzun süreceği verilerle sınanmalıdır. Bu verileri saptamak zordur. Zincir üreten birleşim sırası `Kume_Bul_1` için tam da böyle bir girdidir. Rastgele testler onu kolayca kaçırır. Rastgele seçilen iki öğenin kökleri birleştirildiğinde ağaçlar zincir gibi uzamaz. Ölçümde *n = 100.000* öğe üzerinde 200.000 rastgele işlem yapıldı; her işlemde iki bulma, işlemlerin yarısında da birleşim vardı. İlk sürüm bulma başına ortalama 1.108 ilerleme yaptı ve en derin düğüm 10.130. düzeydeydi. Bu kötü bir sonuçtur, ama zincirin *n* düzeyinden yine de çok uzaktır. Programı yalnız rastgele verilerle deneyen biri gerçek en kötü durumu hiç görmeyebilir.

Yazılım dünyasında “beta sürümü” denen şey bu zorluğun açık bir kabulüdür. Üretici kullanıcıya şunu söyler: gözden kaçırdığımız hatalar olabilir, bulursanız bildirin ki düzeltelim. Yazılım testi (*software testing*) başlı başına bir alandır ve üzerine çok sayıda makale yazılmıştır. Bu bölümün dersi daha basittir: bir yordam yazılınca onu en çok zorlayacak girdi düşünülmeli ve o girdiyle de denenmelidir.

16.4 Ağırlıklı kural

16.4.1 Kural

Birleşimde bir ağacın kökü ötekinin altına girer. Altına giren ağacın bütün düğümlerinin köke uzaklığı 1 artar. Öteki ağacın düğümleri ise yerinde kalır. Maliyeti düşük tutmak için uzaklığı artan düğüm sayısı az olmalıdır. Öyleyse az düğümlü ağaç, çok düğümlü ağacın altına bağlanmalıdır.

Tanım 16.3

Ağırlıklı kural (*weighting rule*): `Birlesim(i, j)` yapılırken *i* ağacındaki düğüm sayısı *j* ağacındakinden azsa *j*, *i*'nin ebeveyni yapılır; aksi hâlde *i*, *j*'nin ebeveyni yapılır. Kısaca düğüm sayısı çok olan ebeveyn olur. Bu kurala boyuta göre birleştirme (*union by size*) da denir.

Şekil 16.6 kuralın etkisini gösterir. 0 kökünde beş düğümlü bir ağaç ve 5 kökünde iki düğümlü bir ağaç birleştiriliyor. Soldaki birleşimde küçük ağaç büyüğün altına girer; yalnız 5 ile 6'nın köke uzaklığı artar ve ağaç 3 düzeyli kalır. Sağdaki birleşimde büyük ağaç küçüğün altına girer; beş düğümün uzaklığı birden artar ve ağaç 4 düzeye çıkar.



Şekil 16.6: Aynı iki ağacın iki birleşimi. Solda küçük ağaç büyüğe eklenmiştir: derinlik 3. Sağda büyük ağaç küçüğe eklenmiştir: derinlik 4.

Kuralı uygulamak için her ağacın düğüm sayısını bilmek gerekir. Bunun için ayrı bir dizi mi tutulmalı? Gerek yoktur. Kök dışındaki bütün düğümler için ebeveyn dizisinde 0 ya da pozitif değerler durur. Kökler ise yalnız "ben köküm" demek için negatif bir değer taşır ve bu değerin büyüklüğü şimdiye kadar hiç kullanılmamıştır. Öyleyse kökün düğüm sayısı, ebeveyn dizisinde negatif işaretle saklanabilir. Kökte −1 yerine −(düğüm sayısı) yazılır. İşaret "kök" bilgisini, mutlak değer "boyut" bilgisini taşır.

16.4.2 `Birlesim_2` : kökte eksi düğüm sayısı

```
void Birlesim_2(int i, int j) {
  /* ağırlıklı kural; i ve j kök olmalı */
  int p = ata[i] + ata[j];  /* -(toplam düğüm) */
  if (ata[i] > ata[j]) {    /* i'de daha az düğüm */
    ata[i] = j;             /* i'yi j'ye bağla */
    ata[j] = p;             /* j'nin sayısını günle */
  } else {                  /* j'de az ya da eşit */
    ata[j] = i;
    ata[i] = p;
  }
}
```

Yordam satır satır şöyle çalışır:

- **3. satır.** `p = ata[i] + ata[j]` . İki kökteki değerler negatif düğüm sayılarıdır. Toplamları, birleşik ağacın düğüm sayısının negatifidir. 3 düğümlü ve 5 düğümlü ağaç için $p = -3 + (-5) = -8$ olur.
- **4. satır.** `ata[i] > ata[j]` karşılaştırması ilk bakışta ters görünür. Değerler negatif olduğu için büyük olan değer küçük boyutu gösterir: $-2 > -4$, 2 düğümün 4 düğümden az olduğunu söyler. Koşul doğruysa *i*'nin ağacında daha az düğüm vardır.
- **5–6. satırlar.** *i* küçük olduğu için *j*'nin altına bağlanır: `ata[i] = j` . Yeni kök *j*'dir ve hücresine birleşik düğüm sayısı yazılır: `ata[j] = p` .
- **7–10. satırlar.** Aksi durumda, yani *j*'de daha az ya da eşit düğüm varsa, *j* *i*'nin altına girer ve *i*'nin hücresi *p* olur. Eşitlikte *j*'nin bağlanması keyfî bir seçimdir; iki seçim de kuralı sağlar.

Atamaların sırası önemlidir. 3. satırda *p* hesaplanmadan önce iki hücreden biri değiştirilseydi, düğüm sayısı bilgisi kaybolurdu. Örneğin önce `ata[i] = j` yazılsa, `ata[i]` artık *i*'nin düğüm sayısını değil, *j*'nin numarasını taşırdı. Yordam bu yüzden toplamı en başta bir yerel değişkene alır.

Maliyet hâlâ $\Theta(1)$'dir: bir toplama, bir karşılaştırma ve iki atama. Ek dizi gerekmez; düğüm sayıları zaten kullanılmayan bilgi alanında saklanır. Başlangıçta bütün hücrelerin −1 olması yeni anlama da uyar: tek öğeli bir kümede bir düğüm vardır.

`Kume_Bul_1` değişmeden kalır. Onun tek ilgilendiği, değerin negatif olup olmadığıdır; −1 ile −8 arasında bir fark gözetmez. Bu, gösterim seçiminin güzel bir sonucudur: kökte saklanan ek bilgi, bulma yordamını hiç etkilemez.

16.4.3 Kod izleme

Sekiz tek öğeli kümeyle başlayalım. Önce komşu çiftler birleştirilsin: `Birlesim_2(0, 1)`, `(2, 3)`, `(4, 5)`, `(6, 7)`. Sonra iki düğümlü ağaçlar ikişer ikişer: `(0, 2)`, `(4, 6)`. En son dört düğümlü iki ağaç: `(0, 4)`. Her birleşimde iki ağaç eşit büyüklüktedir. Tablo 16.7 her adımdaki değerleri verir.

Çağrı	`ata[i]`, `ata[j]`	`p`	`ata` sonrası	Düzey
`(0, 1)`	−1, −1	−2	`[-2, 0, -1, -1, -1, -1, -1, -1]`	2
`(2, 3)`	−1, −1	−2	`[-2, 0, -2, 2, -1, -1, -1, -1]`	2
`(4, 5)`	−1, −1	−2	`[-2, 0, -2, 2, -2, 4, -1, -1]`	2
`(6, 7)`	−1, −1	−2	`[-2, 0, -2, 2, -2, 4, -2, 6]`	2
`(0, 2)`	−2, −2	−4	`[-4, 0, 0, 2, -2, 4, -2, 6]`	3
`(4, 6)`	−2, −2	−4	`[-4, 0, 0, 2, -4, 4, 4, 6]`	3
`(0, 4)`	−4, −4	−8	`[-8, 0, 0, 2, 0, 4, 4, 6]`	4

Tablo 16.7: Yedi `Birlesim_2` çağrısının izlemesi. Her satırda koşul yanlıştır (`else` dalı): önce `ata[j] = i`, sonra `ata[i] = p` yazılır.



Şekil 16.7: İzlemenin üç anı. Üstte dört iki düğümlü ağaç. Ortada `Birlesim_2(4, 6)` sonrası iki dört düğümlü ağaç. Altta `Birlesim_2(0, 4)` sonrası tek ağaç: 8 düğüm, 4 düzey, kökte −8.

Her satırda koşul yanlıştır, çünkü iki kökteki değerler eşittir: $-1 > -1$, $-2 > -2$ ve $-4 > -4$ ifadelerinin hiçbiri doğru değildir. Bu yüzden her seferinde `else` dalı çalışır ve j, i'nin altına girer. Düzey sayısı yalnız eşit büyüklükteki iki ağaç birleşince artar: 2, 3 ve 4. Sekiz düğümlü ağacın 4 düzeyi vardır. Bu, Teorem 16.2'nin verdiği sınırın tam değeridir: $\lfloor \lg 8 \rfloor + 1 = 4$.

Son durumda en derin düğüm 7'dir: 7 → 6 → 4 → 0 yolu üç ilerleme ister. Tablodaki bütün diziler sunumdaki `Birlesim_2` koduyla derlenen sürücünün çıktısıyla aynıdır.

16.4.4 Derinlik lemması

Ağırlıklı kural ağacı ne kadar bozabilir? İzlemedeki işlem dizisi bu kural için en kötü durumdur: her seferinde eşit büyüklükte iki ağaç birleşir. Her birleşimde küçük ağacın bütün düğümlerinin derinliği 1 artar, ama ağaç en az iki katına çıkar. İkinci gözlem birincisini sınırlar.

Teorem 16.2 (Ağırlıklı birleşimde derinlik)

T, tek düğümlü ağaçlardan başlayıp yalnız `Birlesim_2` ile elde edilmiş *m* düğümlü bir ağaç olsun. *T*'nin düzey sayısı $\lfloor \lg m \rfloor + 1$'den büyük olamaz.

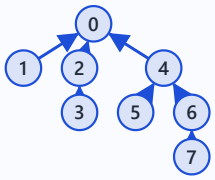
İspat. Daha güçlü bir önermeyi kanıtlayalım: *d* düzeyli her ağaçta en az 2^{d-1} düğüm vardır. Kanıt, yapılan birleşim sayısı üzerinden tümevarımla verilir.

Hiç birleşim yapılmamışsa ağaç tek düğümlüdür: $d = 1$ ve $1 \geq 2^0$. Şimdi düğüm sayıları $m_a \geq m_b$ ve düzey sayıları d_a, d_b olan iki ağacın önermeyi sağladığını varsayalım. `Birlesim_2`, *b* ağacını *a*'nın köküne bağlar (eşitlikte de kökü değişen ağaçlardan biri bağlanır; adlandırma buna göre seçilir). Yeni ağacın düzey sayısı $d = \max(d_a, d_b + 1)$, düğüm sayısı $m = m_a + m_b$'dir. İki durum vardır:

- $d = d_a$ ise $m \geq m_a \geq 2^{d_a - 1} = 2^{d-1}$.
- $d = d_b + 1$ ise $m = m_a + m_b \geq 2m_b \geq 2 \cdot 2^{d_b - 1} = 2^{d-1}$.

İki durumda da önerme korunur. Öyleyse $m \geq 2^{d-1}$, yani $d - 1 \leq \lg m$'dir. $d - 1$ bir tam sayı olduğu için $d - 1 \leq \lfloor \lg m \rfloor$ ve $d \leq \lfloor \lg m \rfloor + 1$ elde edilir. ■

Kanıtın özü şudur. Bir düğümün derinliği yalnız onun ağacı daha küçük ya da eşit bir ağaca bağlanınca 1 artar. O anda düğümün içinde bulunduğu ağaç en az iki katına çıkar. *m* düğüme en çok $\lg m$ kez iki katına çıkılabilir. Öyleyse hiçbir düğümün derinliği $\lg m$'den fazla artamaz.



m = 8 → düzey sayısı 4 = ⌊lg 8⌋ + 1

Şekil 16.8: Eşit ikililerle kurulan 8 düğümlü ağaç sınıra tam olarak ulaşır: 4 düzey = $\lfloor \lg 8 \rfloor + 1$.

Sınır sıkıdır, yani daha iyisi söylenemez. Eşit ikililerle kurulan ağaç onu her $m = 2^k$ için tam olarak yakalar. Ölçümde 16 düğümde 5 düzey, $2^{20} = 1.048.576$ düğümde 21 düzey elde edildi; ikisi de $\lfloor \lg m \rfloor + 1$'e eşittir. Rastgele birleşimlerde ağaçlar sınırın çok altında kalır: 10^6 öğede sınır 20 iken en derin düğüm 9. düzeydeydi.

Sonuç büyük bir kazançtır. Tek bir bulmanın en kötü maliyeti $O(n)$'den $O(\lg n)$'ye düştü. Bir milyon öğede bu, bir milyon ilerleme yerine en çok 19 ilerleme demektir. $n - 1$ birleşim ve *m* bulmadan oluşan bir dizinin toplam maliyeti $O(n + m \lg n)$ olur. Matematik bölümünde soru burada "kanıtlayınız" diye devam ederdi. Mühendis için asıl önemli olan, karmaşıklığın *n*'den $\lfloor \lg n \rfloor + 1$'e inmesidir. Algoritma geliştirmek budur. Ama burada durulmaz; bir adım daha atılabilir.

Sık yapılan hata

Negatif değerlerin karşılaştırmasını ters yorumlamak. `if (ata[i] < ata[j])` yazılırsa koşul "*i*'de daha çok düğüm var" anlamına gelir ve büyük ağaç küçüğün altına bağlanır. Kod derlenir, kümeler doğru kalır, sonuçlar doğrudur; yalnız ağaçlar zincirleşir ve maliyet sessizce $O(n)$'ye döner. Bu tür hatalar yalnız en kötü durum girdisiyle ya da ağaç derinliği ölçülerek fark edilir.

16.4.5 Herhangi iki öğeyi birleştirmek

`Birlesim_1` ve `Birlesim_2` kök ister. Uygulamada ise "3 ile 8'i aynı kümeye koy" gibi istekler gelir ve 3 ile 8 kök olmayabilir. Bu boşluğu dolduran küçük bir sarmalayıcı yordam yazılabilir:

```
int birlestir(int x, int y) {   /* x ve y herhangi iki öğe */
  int a = Kume_Bul_2(x);         /* x'in kökü */
  int b = Kume_Bul_2(y);         /* y'nin kökü */
  if (a == b) return 0;          /* zaten aynı kümede */
  Birlesim_2(a, b);              /* iki kökü birleştir */
  return 1;
}
```

Yordam önce iki öğenin köklerini bulur. Kökler aynıysa öğeler zaten aynı kümededir ve hiçbir şey yapılmaz; dönüş değeri 0'dır. Farklıysa iki kök `Birlesim_2`'ye verilir ve 1 döndürülür. Dönüş değeri, çağıranın "bu istek yeni bir birleşim yaptı mı?" sorusunu cevaplar. Bu bilgi bileşen saymada ve Kruskal algoritmasında doğrudan kullanılır. Yordamda kullanılan `Kume_Bul_2`, azaltma kuralını uygulayan bulma sürümüdür (Tanım 16.4); onun yerine `Kume_Bul_1` da konabilir. Maliyet iki bulmanın maliyetine eşittir, birleşimin katkısı sabittir.

"Kökler aynı mı?" denetimi bir güvenlik önlemidir. Aynı kökle `Birlesim_1(r, r)` çağrılırsa `ata[r] = r` olur: kök kendini gösterir. Ardından gelen her `Kume_Bul_1` bu düğümde sonsuza dek döner, çünkü `ata[r] ≥ 0` koşulu hiç yanlış olmaz. Denemede döngü bir milyon adımda kesildi. `Birlesim_2(r, r)` ise başka türlü bozulur: $p = 2 \cdot ata[r]$ hesaplanır, koşul yanlış çıkar, önce `ata[r] = r` sonra `ata[r] = p` yazılır. Sonuçta kök kök olarak kalır, ama düğüm sayısı iki katına çıkmış görünür. Üç öğeli bir denemede `Birlesim_2(1, 1)` sonrası dizi `[-1, -2, -1]` oldu: tek öğeli küme 2 öğeli sanılıyor. Bu yanlış sayı sonraki birleşimlerde ağırlıklı kuralı yanıltır.

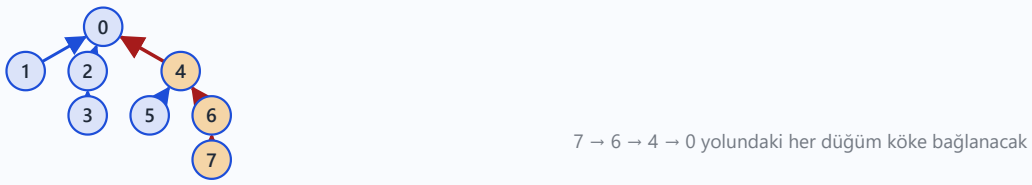
16.5 Azaltma kuralı

16.5.1 Kural

Ağırlıklı kuralla kurulan sekiz düğümlü ağaçta (Şekil 16.7'nin alt kısmı) `Kume_bul(7)` köke üç ilerlemede ulaşır. Program `Kume_bul(7)`'yi tekrar tekrar çağırırsa her çağrı aynı üç adımı yeniden yürür ve maliyet katlanır. Düğüm sayısı az olanı çok olana eklemek derinliği $lg\ n$ ile sınırlar, ama derinliğin artmasını tamamen engellemez. Derinliği birleşim anında daha fazla denetleyemediğimize göre başka bir fırsat aranır: bulma işlemi zaten yolu baştan sona yürür. Yürürken yolu kısaltabilir.

Tanım 16.4

Azaltma kuralı (*collapsing rule*): `Kume_bul(i)` sırasında, i kök değilse, i'den köke giden yol üzerindeki her j düğümü doğrudan köke bağlanır. Bu fikrin genel adı **yol kısaltmadır** (*path compression*).



Şekil 16.9: `Kume_bul(7)` sırasında 7 → 6 → 4 → 0 yolundaki her düğüm doğrudan 0'a bağlanacaktır.

Kural, ağaçların "anlamsız" yerleşiminden yararlanır. Bir öğenin ağaçtaki yeri hiçbir bilgi taşımaz; önemli olan yalnız hangi köke ulaştığıdır. 7'nin 6'nın altında ya da doğrudan 0'ın altında durması kümeleri değiştirmez. Değişen yalnız sonraki bulmaların maliyetidir. Çizge kuramında aynı düşünce "kapama" (*closure*) adıyla geçer: dolaylı bir bağlantı, doğrudan bir kenarla kısaltılır. Problemler farklı olsa da alttaki yapı aynıdır ve bu benzerlik çizge problemlerinde sık sık işe yarar.

16.5.2 `Kume_Bul_2`

```
int Kume_Bul_2(int i) {
  int r, s;
  for (r = i; ata[r] >= 0; r = ata[r])
    ;                   /* kökü bul */
  while (i != r) {      /* yolu azalt */
    s = ata[i];         /* i'nin ebeveynini sakla */
    ata[i] = r;         /* i'yi doğrudan köke bağla */
    i = s;              /* ebeveynine geç */
  }
  return r;
}
```

Yordam yolu iki kez geçer.

- **Birinci geçiş (2–4. satırlar).** `for` döngüsü `Kume_Bul_1`'in yaptığını yapar, ama `i`'yi değiştirmeden ayrı bir `r` değişkeniyle yürür. Döngü bitince `r` köktür. `i` hâlâ başlangıç düğümünü gösterir; ikinci geçiş oradan başlayacaktır.
- **İkinci geçiş (5–9. satırlar).** `while` döngüsü `i`'den köke kadar yeniden yürür. Her turda üç iş yapılır. Önce `s = ata[i]` ile `i`'nin eski ebeveyni saklanır. Sonra `ata[i] = r` ile `i` doğrudan köke bağlanır. En son `i = s` ile saklanan ebeveyne geçilir. Döngü `i` köke ulaşınca biter.
- **10. satır.** Kök döndürülür.

`s` değişkeni yordamın can alıcı parçasıdır. `ata[i] = r` ataması `i`'nin eski ebeveynini siler. Eski ebeveyn önceden saklanmazsa yolun devamı kaybolur; döngü bir sonraki düğüme geçemez. Bu tam olarak bağlı listeyi ters çevirirken bir sonraki düğümü saklama zorunluluğuna benzer: bir bağı değiştirmeden önce, o bağın gösterdiği yer bir değişkende tutulmalıdır.

İkinci döngünün küçük bir kusuru vardır. Köke zaten doğrudan bağlı bir düğüm için de bir kez çalışır ve `ata[i] = r` atamasıyla hücreye aynı değeri yeniden yazar. Bu gereksizdir ama zararsızdır. Yolun son düğümü her zaman köke bağlı olduğu için, kök dışında her çağrıda tam olarak bir "fazladan tur" yapılır.

Maliyet analizi: başlangıç düğümünün köke uzaklığı d ise birinci döngü d, ikinci döngü de d tur döner. Tek çağrının maliyeti $\Theta(d + 1)$'dir; yani tek başına `Kume_Bul_1`'in yaklaşık iki katıdır. Ağırlıklı kuralla birlikte $d \le \lfloor \lg n \rfloor$ olduğundan tek bir çağrı en kötü durumda $O(\lg n)$'dir. Kazanç tek çağrıda değil, çağrı dizisinde görülür: yol üzerindeki bütün düğümler artık köke bir adım uzaklıktadır. Ek bellek $O(1)$'dir: `r` ve `s`.

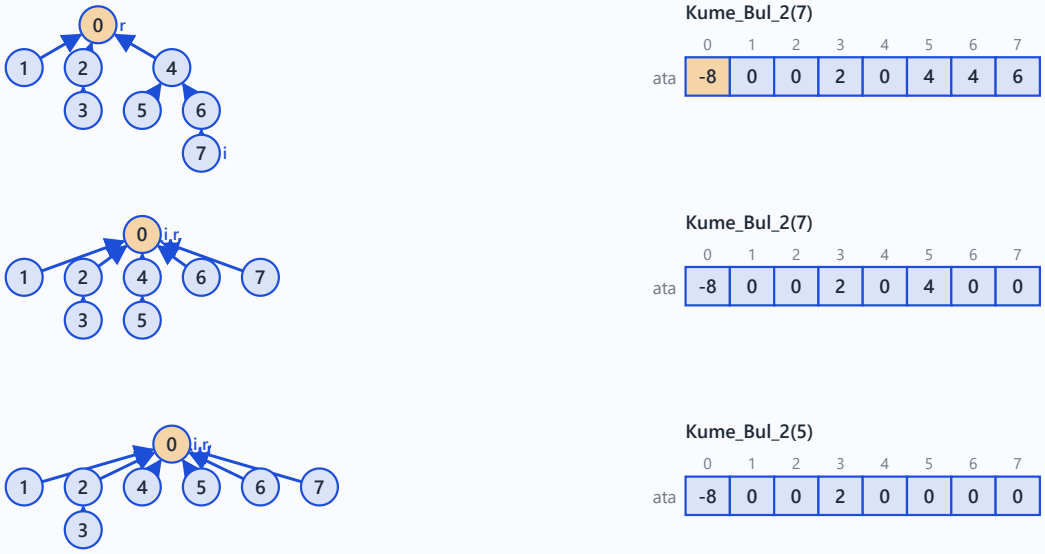
Kökteki negatif sayı yol kısaltmada bozulmaz. Yalnız kök dışındaki düğümlerin hücreleri değişir ve kökün hücresine hiç yazılmaz. Ağaçtaki düğüm sayısı da değişmez, çünkü hiçbir düğüm ağaçtan çıkmaz. Ağırlıklı kural bu yüzden yol kısaltmayla birlikte sorunsuz çalışır.

16.5.3 Kod izleme

Ağırlıklı izlemenin sonundaki ağaçla başlayalım: `ata = [-8, 0, 0, 2, 0, 4, 4, 6]`. Sırayla `Kume_Bul_2(7)`, yine `Kume_Bul_2(7)` ve `Kume_Bul_2(5)` çağrılsın. Tablo 16.8 ilk çağrıyı adım adım verir.

Adım	Döngü	Durum	İşlem	Değişen hücre
1	`for`	`r = 7`	`ata[7] = 6 ≥ 0` → `r = 6`	—
2	`for`	`r = 6`	`ata[6] = 4 ≥ 0` → `r = 4`	—
3	`for`	`r = 4`	`ata[4] = 0 ≥ 0` → `r = 0`	—
4	`for`	`r = 0`	`ata[0] = -8 < 0`: kök	—
5	`while`	`i = 7`	`s = 6`, `ata[7] = 0`, `i = 6`	`ata[7]`: 6 → 0
6	`while`	`i = 6`	`s = 4`, `ata[6] = 0`, `i = 4`	`ata[6]`: 4 → 0
7	`while`	`i = 4`	`s = 0`, `ata[4] = 0`, `i = 0`	(zaten 0: fazladan tur)
8	—	`i == r`	`return 0`	`ata = [-8, 0, 0, 2, 0, 4, 0, 0]`

Tablo 16.8: `Kume_Bul_2(7)` izlemesi. Başlangıçta `ata = [-8, 0, 0, 2, 0, 4, 4, 6]`.



Şekil 16.10: Üstte birinci geçişin sonu: `r = 0` bulundu, ağaç henüz değişmedi. Ortada `Kume_Bul_2(7)` bitti: 7 ve 6 doğrudan köke bağlı. Altta `Kume_Bul_2(5)` sonrası: 5 de köke bağlandı; ağaçta yalnız 3, 2'nin altında kaldı.

İkinci `Kume_Bul_2(7)` çağrısı kısadır. `ata[7] = 0` olduğu için birinci döngü tek ilerlemede kökü bulur. İkinci döngü de tek tur döner; bu tur yine "fazladan" turdur, çünkü 7 zaten köke bağlıdır. Dizi değişmez.

`Kume_Bul_2(5)` ise yeni bir yol açar: 5 → 4 → 0. Birinci döngü iki ilerleme yapar. İkinci döngü 5'i köke bağlar (`ata[5]` : 4 → 0), sonra 4 için fazladan bir tur döner. Son durum `ata = [-8, 0, 0, 2, 0, 0, 0, 0]` 'dır. Sekiz düğümün altısı artık köke doğrudan bağlıdır; yalnız 3, iki adım uzaktadır. Bütün diziler, sunumdaki kodla derlenen sürücünün çıktısıyla karşılaştırıldı.

> **Sık yapılan hata**
>
> `while` döngüsünde ebeveyni saklamadan bağlamak: `ata[i] = r; i = ata[i];` . İkinci atama artık `r` 'yi okur; `i` doğrudan köke atlar ve yolun geri kalanı kısaltılmadan bırakılır. Yordam doğru kökü döndürdüğü için hata çıktıda görünmez; yalnız beklenen hızlanma gelmez. Doğrusu, eski ebeveyni önce `s` 'ye almaktır.

16.5.4 Bu nasıl iyileştirme?

İlk bakışta itiraz yerindedir: bir `for` döngüsü vardı, bir de `while` döngüsü eklendi. Bu nasıl iyileştirme olur? Cevap tek çağrıya değil, çağrı dizisine bakınca görülür. `Kume_Bul_2(7)` ikinci kez çağrılınca ve sonraki her çağrıda kökü bir çırpıda bulur. Tablo 16.9 iki sürümün maliyetini aynı ağaç üzerinde karşılaştırır.

Çağrı	`Kume_Bul_1` (ilerleme)	`Kume_Bul_2` (ilerleme + bağlama)
1. `Küme_bul(7)`	3	3 + 3
2. `Küme_bul(7)`	3	1 + 1
10 kez `Küme_bul(7)`	30	6 + 9 · 2 = 24
`Küme_bul(5)` (sonra)	2	2 + 2, ardından 1 + 1

Tablo 16.9: Aynı ağaçta (`ata = [-8, 0, 0, 2, 0, 4, 4, 6]`) iki bulma sürümünün maliyeti. `Kume_Bul_2` için "ilerleme + bağlama" yazılmıştır.

İlk çağrı yaklaşık iki kat pahalıdır: 3 yerine 6 adım. İkinci çağrı ise 3 yerine 2 adım sürer. On çağrıda toplam 30'a karşı 24 olur ve fark çağrı sayısıyla büyür. Bin çağrıda `Kume_Bul_1` 3.000 ilerleme yapar, `Kume_Bul_2` ise $6 + 999 \cdot 2 = 2.004$ adımda kalır. Yolu kısaltmanın bedeli bir kez ödenir, getirisi her sonraki çağrıda toplanır.

Bu tür bir hesaba **amortize analiz** (*amortized analysis*) denir. Tek bir işlemin en kötü maliyeti yerine, uzun bir işlem dizisinin toplam maliyeti işlem sayısına bölünür. Pahalı işlemler arada bir olur ve arkalarından gelen işlemleri ucuzlatır. Maliyet birçok işlem üzerine yayılınca işlem başına neredeyse sabite iner.

Yol kısaltmanın etkisi en açık biçimde ağırlıklı kuralın en kötü ağacında görülür. Eşit ikililerle kurulan 2^{20} düğümlü ağaçta (21 düzey) 2^{20} rastgele bulma yapıldı. Yalnız ağırlıklı kuralla bulma başına ortalama 10,00 ilerleme ölçüldü; ağaç değişmediği için bu sayı hiç azalmaz. Yol kısaltmayla ortalama 1,78 ilerlemeye düştü ve ağacın düzey sayısı 21'den 6'ya indi. Süre de 0,18 saniyeden 0,03 saniyeye indi (makineye göre değişir).

16.5.5 Özyinelemeli yol kısaltma

Yol kısaltma özyinelemeyle tek satırda da yazılabilir:

```
int bul(int i) { return ata[i] < 0 ? i : (ata[i] = bul(ata[i])); }
```

Okuması şöyledir. *i* kökse (`ata[i] < 0`) kendisi döndürülür. Değilse önce ebeveyninin kökü özyinelemeyle bulunur: `bul(ata[i])`. Dönen kök `ata[i]`'ye yazılır ve aynı değer döndürülür. Atama ifadesinin değeri, atanan değerdir; `(ata[i] = bul(ata[i]))` bu yüzden hem bağı günler hem de kökü verir.

Özyineleme yolu kökten geriye doğru kısaltır. Çağrılar köke kadar iner, dönüşte her çerçeve kendi düğümünü köke bağlar. `bul(7)` sekiz düğümlü örnek ağaçta `bul(6)`'yı, o `bul(4)`'ü, o da `bul(0)`'ı çağırır. `bul(0)` 0 döndürür. Dönüşte sırasıyla `ata[4] = 0`, `ata[6] = 0` ve `ata[7] = 0` yazılır. Sonuç `Kume_Bul_2(7)` ile aynıdır: `[-8, 0, 0, 2, 0, 4, 0, 0]`. `bul(5)` ardından `[-8, 0, 0, 2, 0, 0, 0, 0]` verir; bu da aynıdır.

İki sürümün farkı belleğe ilişkindir. Döngülü `Kume_Bul_2` *O(1)* ek bellek kullanır. Özyinelemeli `bul` ise yol uzunluğu kadar çağrı çerçevesi açar: *O(d)*. Ağırlıklı kuralla $d \le \lg n$ olduğundan bu bir sorun değildir; bir milyar öğede bile en çok 30 çerçeve açılır. Ama ağırlıksız birleşimle kurulmuş bir zincirde d, n'ye ulaşabilir ve çağrı yığını taşabilir.

16.6 Tarjan–van Leeuwen sonucu ve Ackermann fonksiyonu

16.6.1 Neredeyse sabit maliyet

İki kural birlikte kullanılınca maliyet ne olur? Ağırlıklı kural tek bir bulmayı *O(lg n)* ile sınırlar. Azaltma kuralı ise yolları kısalttığı için tekrarlanan bulmaları ucuzlatır. Bu ikisinin birleşik etkisini kesin olarak hesaplamak kolay değildir. Soruyu R. E. Tarjan ile J. van Leeuwen 1984'te kapattı.

> **Teorem 16.3 (Tarjan ve van Leeuwen)**
>
> Her biri tek düğümlü n kümeyle başlansın. f küme bulma ve u birleştirme, ağırlıklı kural ve azaltma kuralıyla, herhangi bir karışık sırada uygulansın. Gereken en çok zaman $T(f, u)$ olsun. $u \ge n/2$ için öyle k_1 ve k_2 pozitif sabitleri vardır ki
>
> $$k_1 \cdot (n + f \cdot \alpha(f + n, n)) \le T(f, u) \le k_2 \cdot (n + f \cdot \alpha(f + n, n))$$
>
> olur. Burada α ters Ackermann fonksiyonudur.

Teorem iki şey söyler. Üst sınır, işlemlerin toplam maliyetinin $n + f \cdot \alpha$ ile orantılı olduğunu gösterir. Alt sınır ise bundan daha iyisinin bu iki kuralla mümkün olmadığını söyler; analiz sıkıdır. Teoremin kanıtı bu notun kapsamını aşar. Ama sonucu yorumlamak için yalnız α'nın ne kadar yavaş büyüdüğünü bilmek yeter. α o kadar yavaş büyür ki evrendeki atom sayısı kadar öğe için bile değeri 4'ü geçmez. Toplam maliyet işlem sayısına bölününce işlem başına maliyet neredeyse *O(1)* çıkar.

Bir milyon öğe düşünelim. Bir program bir öğenin kümesini neredeyse sabit sürede buluyor, öteki hâlâ listeleri tek tek tarıyor. İkisi de doğru sonuç verir, ama biri saniyenin küçük bir kesrinde biter, öteki dakikalarca çalışır. Aynı işe başvuran matematikçi ile mühendisin farkı da budur: mühendis doğru sonucun yanında maliyeti de sorgular.

16.6.2 Ackermann fonksiyonu

α'yı anlamak için önce tersi alınan fonksiyona bakmak gerekir. Ackermann fonksiyonu iki doğal sayı argümanlı, özyinelemeyle tanımlı ve son derece hızlı büyüyen bir fonksiyondur.

Tanım 16.5

Ackermann fonksiyonu (*Ackermann function*) $m, n \geq 0$ için şöyle tanımlanır:

```
/* Ackermann: çok hızlı büyür */
A(m, n) = n + 1                 m = 0
        = A(m-1, 1)             m > 0, n = 0
        = A(m-1, A(m, n-1))     m > 0, n > 0
```

Tanım üç durumdan oluşur. $m = 0$ ise sonuç $n + 1$'dir; özyineleme burada biter. $n = 0$ ise birinci argüman bir azalır, ikinci argüman 1 olur. Üçüncü durum en ilginç olanıdır: dış çağrının ikinci argümanı bir iç çağrının sonucudur. İç çağrı $A(m, n-1)$, dış çağrı $A(m-1, \ldots)$'dir. Her adımda ya m küçülür ya da m aynı kalıp n küçülür. Bu yüzden özyineleme her zaman biter; ama ne zaman bittiği başka bir sorudur.

	n = 0	**n = 1**	**n = 2**	**n = 3**	**n = 4**
m = 0	1	2	3	4	5
m = 1	2	3	4	5	6
m = 2	3	5	7	9	11
m = 3	5	13	29	61	125
m = 4	13	65533	$2^{65536} - 3$	…	…

Tablo 16.10: $A(m, n)$ değerleri. İlk dört satır özyinelemeli C koduyla hesaplanıp doğrulandı.

Satırların her biri bir öncekinin yinelenmesidir ve bu yüzden kapalı biçimleri vardır. $A(1, n) = n + 2$, $A(2, n) = 2n + 3$ ve $A(3, n) = 2^{n+3} - 3$'tür. $m = 3$ satırında değerler üstel büyür. $m = 4$ satırında ise üs kulesi başlar: $A(4, 1) = 2^{16} - 3 = 65533$, $A(4, 2) = 2^{65536} - 3$. İkincisi yaklaşık 19.729 basamaklı bir sayıdır. $A(5, 1)$ ise 65536 tane 2'den oluşan bir üs kulesinin 3 eksiğidir; akıl almaz büyüklükte bir sayıdır.

Ackermann fonksiyonunun birden çok tanımı vardır. Yukarıdaki tanımla $A(3, 4) = 125$'tir. Horowitz'in kitabında kullanılan ve Tarjan'ın çalışmasından gelen tanımda ise $A(1, j) = 2^j$ ile başlanır ve $A(3, 1) = 16$ çıkar. Hangi tanım seçilirse seçilsin varılan sonuç değişmez: hepsi aynı ölçüde hızlı büyür ve uygulamada $\alpha \leq 4$'tür.

16.6.3 Ters Ackermann fonksiyonu

Tanım 16.6

Ters Ackermann fonksiyonu (*inverse Ackermann function*) $m \geq n \geq 1$ için

$$\alpha(m, n) = \min\{\, i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \lg n \,\}$$

biçiminde tanımlanır. Yani α, Ackermann fonksiyonunun $\lg n$'yi aşması için gereken en küçük birinci argümandır.

A çok hızlı büyüdüğü için onu belli bir değerin üstüne çıkarmaya yetecek i çok küçüktür. Bu yüzden α çok yavaş büyür. Hesabı yapalım. $m \geq n$ olduğundan $\lfloor m/n \rfloor \geq 1$'dir ve $A(i, \lfloor m/n \rfloor) \geq A(i, 1)$ olur. Tablodan $A(1, 1) = 3$, $A(2, 1) = 5$, $A(3, 1) = 13$ ve $A(4, 1) = 65533$'tür. Öyleyse:

- $n \leq 2^3 = 8$ için $\lg n \leq 3$ ve $\alpha \leq 1$'dir.
- $n \leq 2^5 = 32$ için $\alpha \leq 2$'dir.
- $n \leq 2^{13} = 8192$ için $\alpha \leq 3$'tür.
- $n \leq 2^{65533}$ için $\alpha \leq 4$'tür.

Gözlenebilir evrendeki atom sayısının yaklaşık 10^{80}, yani kabaca 2^{266} olduğu tahmin edilir. 2^{65533} bundan akıl almaz ölçüde büyüktür. Bilgisayarlarda karşılaşılabilecek her n için $\alpha \leq 4$'tür. Bir Python betiğiyle tanımdan hesaplanan değerler bu eşikleri doğruladı: $n = 8$'de 1, $n = 9$ ve 32'de 2, $n = 33$ ve 8192'de 3, $n = 8193$, 10^6 ve 2^{64}'te 4.

Kuramsal olarak α sınırsız büyür; bu yüzden $O(\alpha)$ sabit değildir ve "union-find işlemleri $O(1)$'dir" demek kesin olarak doğru değildir. Uygulamada ise α en çok 4 olan bir çarpandır. Doğru ifade "neredeyse sabit" ya da "uygulamada doğrusal toplam maliyet"tir.

16.6.4 Özyinelemeli Ackermann ve çağrı yığını

Ackermann fonksiyonu, özyinelemenin gücünü ve çağrı yığınının sınırlarını göstermek için sevilen bir sınav sorusudur. Sorulan şudur: fonksiyonu özyinelemeli olarak yazınız; ardından `A(1, 2)` çağrısının çağrı yığınını adım adım gösteriniz. C karşılığı tanımın doğrudan çevirisidir:

```c
unsigned long ackermann(unsigned long m,
                        unsigned long n) {
  if (m == 0) return n + 1;
  if (n == 0) return ackermann(m - 1, 1);
  return ackermann(m - 1, ackermann(m, n - 1));
}
```

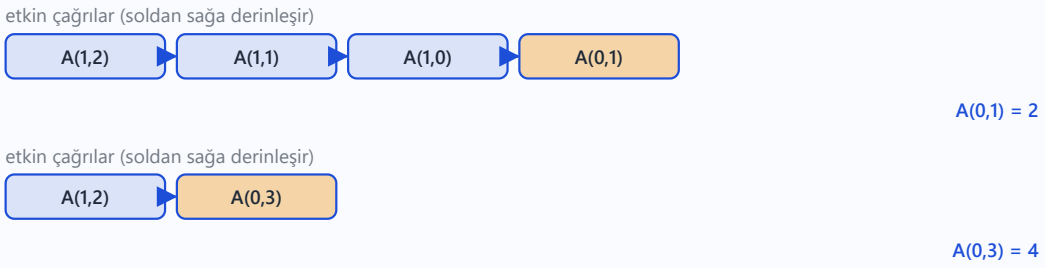
Tanımın üç durumu üç satıra karşılık gelir. Sonuçlar hızla büyüdüğü için tip olarak `unsigned long` seçilmiştir; negatif değer hiç oluşmaz. Windows'ta `unsigned long` 32 bittir, Linux'ta 64 bit. *A(4, 1) = 65533* iki tipe de sığar; *A(4, 2)* ise hiçbir tam sayı tipine sığmaz.

`A(1, 2)` 'nin açılımı şöyledir:

A(1,2) = A(0, A(1,1)) = A(0, A(0, A(1,0))) = A(0, A(0, A(0,1))) = A(0, A(0, 2)) = A(0, 3) = 4

Adım	Etkin çağrılar	Olay
1	A(1,2)	Üçüncü durum: önce iç çağrı A(1,1) gerekir.
2	A(1,2) A(1,1)	Yine üçüncü durum: önce A(1,0).
3	A(1,2) A(1,1) A(1,0)	*n = 0*: A(0,1) çağrılır.
4	A(1,2) A(1,1) A(1,0) A(0,1)	*m = 0*: 2 döner. A(1,0) = 2.
5	A(1,2) A(1,1) A(0,2)	İç çağrı 2 verdi, dış çağrı A(0,2): 3 döner. A(1,1) = 3.
6	A(1,2) A(0,3)	İç çağrı 3 verdi, dış çağrı A(0,3): 4 döner.
7	—	A(1,2) = 4.

Tablo 16.11: `ackermann(1, 2)` izlemesi. "Etkin çağrılar" sütunu çağrı yığınını soldan sağa (en dıştan en içe) gösterir.



Şekil 16.11: `ackermann(1, 2)` izlemesinden iki an. Üstte yığın en derin hâlinde: dört etkin çağrı, en içteki A(0,1) 2 döndürüyor. Altta son dış çağrı A(0,3) 4 döndürüyor.

Sayaçlı bir sürümle yapılan ölçüm, `A(1, 2)` 'nin toplam 6 çağrı yaptığını ve yığının en çok 4 çerçeve derinleştiğini gösterdi. Bu, tablodaki altı farklı çağrıyla (A(1,2), A(1,1), A(1,0), A(0,1), A(0,2), A(0,3)) uyumludur. Sayılar hızla büyür: `A(2, 2)` 27 çağrı ve 8 derinlik, `A(3, 3)` 2.432 çağrı ve 63 derinlik, `A(4, 0)` 107 çağrı ve 16 derinlik ister. Aynı sayma kuralıyla Python'da yapılan hesaba göre `A(4, 1)` 2.862.984.010 çağrı ister ve yığın 65.536 çerçeveye kadar derinleşir. Böyle bir derinlik varsayılan çağrı yığınını kolayca aşabilir. `A(4, 2)` ise hiçbir bilgisayarda bitmez; sonucun basamakları bile belleğe sığmaz.

16.6.5 Ölçümle karşılaştırma

Üç sürüm aynı rastgele işlem dizisiyle karşılaştırıldı. Her işlemde iki rastgele öğenin kökü bulundu; işlemlerin yarısında, kökler farklıysa, iki küme birleştirildi. Tablo 16.12 sonuçları verir.

Sürüm	*n*	Bulma başına ilerleme	En derin düzey	Süre
`Birlesim_1` + `Kume_Bul_1`	100.000	1.108,25	10.130	≈ 2,7–3,1 s
`Birlesim_2` + `Kume_Bul_1`	100.000	0,90	8	≈ 0,007 s
`Birlesim_2` + `Kume_Bul_2`	100.000	0,61	4	≈ 0,006 s
`Birlesim_2` + `Kume_Bul_1`	1.000.000	0,91	10	≈ 0,13 s
`Birlesim_2` + `Kume_Bul_2`	1.000.000	0,61	5	≈ 0,11 s

Tablo 16.12: *n* öğe, *2n* rastgele işlem (*4n* bulma). “İlerleme” yalnız köke doğru yapılan adımları sayar. Süreler makineye göre değişir.

Rastgele işlemlerde ilk sürüm bile zincirin *n* düzeyine ulaşmaz, ama bulma başına bin adımı aşar. Ağırlıklı kural bunu bin kattan fazla azaltır. Rastgele girdide ağırlıklı ağaçlar zaten sığ olduğu için yol kısaltmanın katkısı burada küçüktür: ilerleme 0,90'dan 0,61'e iner. Bağlama turları da sayılınca toplam iş biraz artar (100.000 öğede yaklaşık 360 bine karşı 492 bin adım), süre ise hemen hemen aynı kalır. Yol kısaltmanın asıl değeri, aynı derin yolların tekrar tekrar yüründüğü kötü durumlarda görülür. Eşit ikililerle kurulan ağaçtaki ölçüm bunu göstermişti: bulma başına 10,00 ilerleme 1,78'e indi.

16.6.6 Bu sonuçlar nereden geliyor?

Tarjan, çizge algoritmaları üzerine birçok temel sonucun sahibi ünlü bir bilgisayar bilimcisidir. Bu teorem onun J. van Leeuwen ile ortak çalışmasından çıkmıştır (1984). Bu tür sonuçlar araştırmanın ürünüdür. Lisans eğitiminde dört yıl okunur ve bir meslek kazanılır. Doktorada ise çoğu zaman yalnız bir problem üzerine dört yıl çalışılır ve dünyada henüz çözülmemiş bir soruya cevap aranır. Böyle bir çalışma tek başına değil, bir grup içinde yürütülür; grupta çalışan kişi sürekli öğrenir.

Bu konuların bir haftada bitmesi beklenmemelidir; hiç çalışılmazsa hiç bitmez. Her gün düzenli olarak iki üç saat çalışmak ve haftanın konusunu aynı hafta tekrar etmek, konuların birikmesini önler. Union-find gibi kısa ama derin bir yapı, bu tekrar sırasında küçük örnekleri elle izleyerek en iyi öğrenilir.

16.7 Uygulamalar

16.7.1 Bağlantılı bileşen sayma

Yönsüz bir çizgede birbirine bir yolla ulaşılabilen düğümlerin oluşturduğu en büyük gruplara **bağlantılı bileşen** (*connected component*) denir. Bileşenler düğüm kümesini ayrık parçalara böler; bu yüzden union-find'ın doğal bir uygulamasıdır. Başlangıçta her düğüm ayrı bir bileşendir. Her kenar, iki ucunun bulunduğu bileşenleri birleştirir. Bütün kenarlar işlenince kalan küme sayısı bileşen sayısıdır.

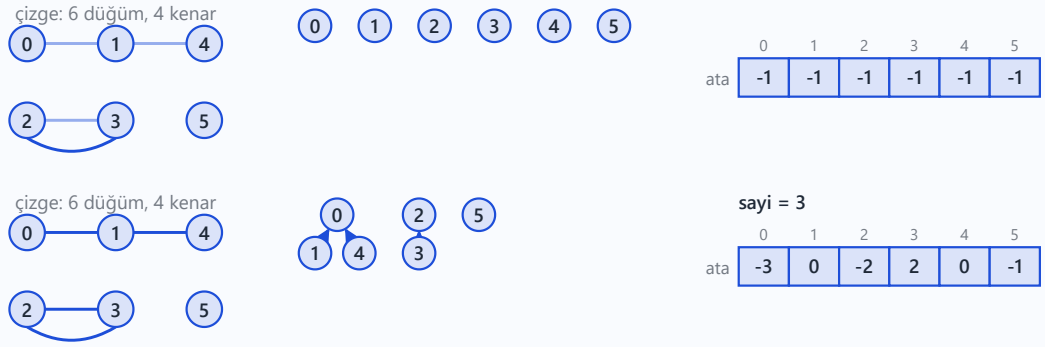
```
int bilesen_say(int n, int m, int u[], int v[]) {
  for (int i = 0; i < n; i++) ata[i] = -1;
  int sayi = n;               /* her düğüm ayrı */
  for (int e = 0; e < m; e++) {
    int a = Kume_Bul_2(u[e]), b = Kume_Bul_2(v[e]);
    if (a != b) {             /* farklı bileşen */
      Birlesim_2(a, b);
      sayi--;
    }
  }
  return sayi;
}
```

Yordam *n* düğümlü ve *m* kenarlı bir çizgeyi iki dizi olarak alır: *e*. kenar `u[e]` ile `v[e]` düğümlerini bağlar. İlk satır ebeveyn dizisini sıfırdan kurar; her düğüm tek öğeli bir kümedir. `sayi` değişkeni bileşen sayısını tutar ve *n* ile başlar. Döngü her kenar için iki ucun kökünü bulur. Kökler farklıysa kenar iki ayrı bileşeni birleştirir: kökler `Birlesim_2` ile birleştirilir ve bileşen sayısı bir azalır. Kökler aynıysa kenarın iki ucu zaten aynı bileşendedir; kenar bir döngü kapatır ve sayı değişmez.

Örnek çizgede 6 düğüm ve 4 kenar vardır: (0, 1), (2, 3), (1, 4) ve (3, 2). Son kenar, 2 ile 3 arasındaki ikinci bir kenardır. Tablo 16.13 izlemeyi verir.

Kenar	`a`	`b`	Karar	`ata` sonrası	`sayi`
(0, 1)	0	1	Farklı: `Birlesim_2(0, 1)`	`[-2, 0, -1, -1, -1, -1]`	5
(2, 3)	2	3	Farklı: `Birlesim_2(2, 3)`	`[-2, 0, -2, 2, -1, -1]`	4
(1, 4)	0	4	Farklı: `Birlesim_2(0, 4)`	`[-3, 0, -2, 2, 0, -1]`	3
(3, 2)	2	2	Aynı: döngü kapatır	değişmez	3

Tablo 16.13: `bilesen_say(6, 4, u, v)` izlemesi. Başlangıçta `ata` tümüyle −1, `sayi = 6`.



Şekil 16.12: Bileşen sayma. Solda çizge, ortada orman, sağda ebeveyn dizisi. Üstte başlangıç: altı tek öğeli küme. Altta sonuç: üç bileşen, {0, 1, 4}, {2, 3} ve {5}.

Üçüncü kenarda ilginç bir an vardır. Kenar (1, 4)'tür, ama 1 kök değildir. `Kume_Bul_2(1)`, 1'in kökü olan 0'ı döndürür ve birleşim 0 ile 4 arasında yapılır. 0'ın ağacında iki, 4'ünkünde bir düğüm vardır; *−2 > −1* yanlış olduğu için 4, 0'ın altına girer. Dördüncü kenarın iki ucu da 2 köküne ulaşır ve kenar atlanır. Sonuç 3'tür: `{0, 1, 4}`, `{2, 3}`, `{5}`. Yordam sunumdaki kodla derlenip çalıştırıldı; 3 döndürdü ve son dizi `[-3, 0, -2, 2, 0, -1]` oldu.

Maliyet: ilk döngü $\Theta(n)$'dir. Kenar döngüsü m kez döner; her turda iki bulma ve en çok bir birleşim yapılır. Teorem 16.3'e göre toplam $O(n + m \cdot \alpha(m + n, n))$ olur; uygulamada bu doğrusaldır. Bellek n tam sayılık ebeveyn dizisidir. Çizgenin komşuluk listesi kurulmaz; kenarlar okunurken işlenebilir. Bu özellik, kenarları bir dosyadan ya da ağdan sırayla gelen çok büyük çizgelerde union-find'ı derinlik öncelikli aramaya göre çekici kılar.

16.7.2 Eşdeğerlik sınıfları

Bir bağıntı yansımalı ($x \equiv x$), simetrik ($x \equiv y$ ise $y \equiv x$) ve geçişli ($x \equiv y$ ve $y \equiv z$ ise $x \equiv z$) ise eşdeğerlik bağıntısıdır. Eşdeğerlik bağıntısı, öğeleri **eşdeğerlik sınıfları** (*equivalence classes*) denen ayrık kümelere böler. Horowitz'in kitabındaki klasik problem şudur: $i \equiv j$ biçiminde bir dizi bildirim verilmiştir; sınıfları bulunuz. Örneğin derleyicide hangi değişkenlerin aynı bellek alanını paylaştığını belirleyen `EQUIVALENCE` bildirimleri böyle işlenir.

Geçişlilik yüzünden bildirimler dolaylı sonuçlar doğurur: $7 \equiv 4$ ve $0 \equiv 4$ ise $7 \equiv 0$'dır, bu ayrıca bildirilmese bile. Union-find bu dolaylı sonuçları kendiliğinden toplar. Her bildirim iki öğenin kümelerini birleştirir:

```
void esdegerlik(int n, int m, int x[], int y[]) {
  for (int i = 0; i < n; i++) ata[i] = -1;
  for (int k = 0; k < m; k++)
    birlestir(x[k], y[k]);          /* x[k] ≡ y[k] bildirimi */
  for (int r = 0; r < n; r++) {     /* her kök bir sınıf */
    if (ata[r] >= 0) continue;
    printf("{");
    for (int i = 0; i < n; i++)
      if (Kume_Bul_2(i) == r) printf(" %d", i);
    printf(" }\n");
  }
}
```

12 öğe ve dokuz bildirimle deneyelim: *0 ≡ 4, 3 ≡ 1, 6 ≡ 10, 8 ≡ 9, 7 ≡ 4, 6 ≡ 8, 3 ≡ 5, 2 ≡ 11, 11 ≡ 0*. Tablo 16.14 her bildirimde bulunan kökleri ve bildirimden sonraki sınıfları verir.

Bildirim	Kökler	Sonraki sınıflar (öğesi birden çok olanlar)
0 ≡ 4	0, 4	{0, 4}
3 ≡ 1	3, 1	{0, 4}, {1, 3}
6 ≡ 10	6, 10	{0, 4}, {1, 3}, {6, 10}
8 ≡ 9	8, 9	{0, 4}, {1, 3}, {6, 10}, {8, 9}
7 ≡ 4	7, 0	{0, 4, 7}, {1, 3}, {6, 10}, {8, 9}
6 ≡ 8	6, 8	{0, 4, 7}, {1, 3}, {6, 8, 9, 10}
3 ≡ 5	3, 5	{0, 4, 7}, {1, 3, 5}, {6, 8, 9, 10}
2 ≡ 11	2, 11	{0, 4, 7}, {1, 3, 5}, {6, 8, 9, 10}, {2, 11}
11 ≡ 0	2, 0	{0, 2, 4, 7, 11}, {1, 3, 5}, {6, 8, 9, 10}

Tablo 16.14: Eşdeğerlik bildirimlerinin işlenmesi. “Kökler” sütunu `birlestir` içindeki iki `Kume_Bul_2` çağrısının sonucudur.

Beşinci bildirimde 4 artık kök değildir; kökü 0'dır. Son bildirimde ise 11'in kökü 2'dir. 2'nin ağacında iki, 0'ın ağacında üç düğüm olduğu için *−2 > −3* doğrudur ve 2, 0'ın altına girer. Program üç sınıf yazdırır: `{ 0 2 4 7 11 }`, `{ 1 3 5 }` ve `{ 6 8 9 10 }`. Bu çıktı C sürücüsüyle alındı.

Sınıfları yazdıran son döngü her kök için bütün öğeleri tarar ve her öğe için bir `Kume_Bul_2` çağırır. Sınıf sayısı k ise bu kısım $O(k \cdot n)$ bulma yapar; bütün öğeler ayrı sınıftaysa $O(n^2)$'ye çıkar. Daha iyi bir yol, öğeleri bir kez tarayıp her birini kökünün listesine eklemektir: $O(n \cdot \alpha)$. Örnek kısa olsun diye basit yol seçilmiştir. Bildirimlerin işlenmesi ise $O(n + m \cdot \alpha)$'dır.

16.7.3 Kruskal algoritması

Bağlantılı ve ağırlıklı bir çizgenin bütün düğümlerini kapsayan ağaçlar arasında kenar ağırlıklarının toplamı en küçük olana **minimum kapsayan ağaç** (*minimum spanning tree*) denir. Kruskal algoritması bu ağacı açgözlü bir yolla kurar. Kenarlar ağırlığa göre küçükten büyüğe sıralanır ve sırayla incelenir. Bir kenarın uçları için `Kume_bul` farklı kökler veriyorsa kenar ağaca alınır ve iki küme birleştirilir. Aynı kökü veriyorsa kenar bir döngü kapatır ve reddedilir.

Reddetme kuralının gerekçesi şudur. İki uç zaten aynı kümedeyse, aralarında daha önce seçilmiş kenarlardan oluşan bir yol vardır. Yeni kenar bu yolu bir döngüye kapatır. Ağaçta döngü olmayacağı için kenar alınamaz. Union-find “bu iki uç arasında seçilmiş kenarlardan bir yol var mı?” sorusunu neredeyse sabit sürede cevaplar. Çizgede yol aramak gerekmez.

```
typedef struct { int u, v, w; } kenar;   /* uçlar ve ağırlık */

static int kenar_karsilastir(const void *a, const void *b) {
  const kenar *x = a, *y = b;
  return (x->w > y->w) - (x->w < y->w);  /* küçük ağırlık önce */
}

int kruskal(int n, int m, kenar e[], kenar secilen[]) {
  qsort(e, m, sizeof e[0], kenar_karsilastir);
  for (int i = 0; i < n; i++) ata[i] = -1;
  int k = 0;                             /* seçilen kenar sayısı */
  for (int t = 0; t < m && k < n - 1; t++) {
    int a = Kume_Bul_2(e[t].u), b = Kume_Bul_2(e[t].v);
    if (a != b) {                        /* döngü kapatmıyor */
      Birlesim_2(a, b);
      secilen[k++] = e[t];
    }                                    /* a == b: kenar reddedilir */
  }
  return k;                              /* n-1 ise çizge bağlantılı */
}
```

Kenar bir yapı olarak tutulur: iki uç ve ağırlık. `qsort` standart kütüphanenin sıralama yordamıdır; karşılaştırma fonksiyonu iki kenarın ağırlığına bakar. `(x->w > y->w) - (x->w < y->w)` ifadesi −1, 0 ya da 1 verir. `x->w - y->w` yazmak daha kısa olurdu, ama çok büyük ağırlıklarda tam sayı taşmasına yol açabilir. Döngü, $n - 1$ kenar seçilince erken biter; çünkü n düğümlü bir ağaçta tam $n - 1$ kenar vardır. Yordam seçilen kenar sayısını döndürür. Bu sayı $n - 1$'den küçükse çizge bağlantılı değildir ve sonuç bir kapsayan ağaç değil, bir kapsayan ormandır.

5 düğümlü bir örnek çizgenin kenarları (uç, uç, ağırlık) biçiminde şöyle olsun: (0, 1, 4), (0, 2, 1), (1, 2, 2), (1, 3, 5), (2, 3, 8), (3, 4, 3), (2, 4, 9). Sıralanınca 1, 2, 3, 4, 5, 8, 9 ağırlıkları gelir.

Kenar	Kökler	Karar	`ata` sonrası
(0, 2, 1)	0, 2	Alınır	`[-2, -1, 0, -1, -1]`
(1, 2, 2)	1, 0	Alınır	`[-3, 0, 0, -1, -1]`
(3, 4, 3)	3, 4	Alınır	`[-3, 0, 0, -2, 3]`
(0, 1, 4)	0, 0	Reddedilir: 0–2–1 yoluyla döngü	değişmez
(1, 3, 5)	0, 3	Alınır	`[-5, 0, 0, 0, 3]`

Tablo 16.15: Kruskal izlemesi. Dört kenar seçilince ($n - 1 = 4$) döngü biter; (2, 3, 8) ve (2, 4, 9) hiç incelenmez.

Seçilen kenarlar (0, 2), (1, 2), (3, 4) ve (1, 3)'tür; toplam ağırlık $1 + 2 + 3 + 5 = 11$. (0, 1, 4) kenarı reddedilir, çünkü 0 ile 1 arasında 0–2–1 yolu zaten seçilmiştir. Program aynı dört kenarı ve 11 toplamını verdi.

Maliyet iki parçadan oluşur. Sıralama $O(m \lg m)$'dir. Union-find kısmı en çok $2m$ bulma ve $n - 1$ birleşimdir: $O(m \cdot \alpha)$. Toplamı sıralama belirler: $O(m \lg m)$. Union-find'ın neredeyse sabit maliyeti olmasaydı, örneğin ilk sürüm kullanılsaydı, döngü kısmı $O(m \cdot n)$'ye çıkar ve sıralamayı geçerdi.

16.7.4 Öteki kullanım alanları

- **Ağ bağlantısı.** “Bu iki bilgisayar bağlı mı?” sorularını, bağlantılar eklendikçe çevrim içi cevaplamak. Sedgewick'in kitabı union-find'ı bu örnekle açar.
- **Görüntü işleme.** Aynı renkteki komşu pikselleri bölgelere ayırmak. Bu işleme bağlantılı bileşen etiketleme denir; her piksel bir öğe, her komşuluk bir birleşimdir.
- **Süzülme** (*percolation*). Rastgele açılan hücrelerden oluşan bir ızgarada üstten alta açık bir yol oluşup oluşmadığını sınamak. Monte Carlo deneylerinin çekirdeği, her yeni açık hücreden sonra sorulan bu sorudur.
- **Derleyiciler.** Tip çıkarımında iki tip ifadesini eşitleyen birleştirme (*unification*) ve hangi işaretçilerin aynı belleği gösterebileceğini bulan takma ad (*alias*) analizi.
- **Çevrim dışı en yakın ortak ata.** Tarjan'ın algoritması, bir ağaçta önceden bilinen düğüm çiftlerinin en yakın ortak atasını (*lowest common ancestor, LCA*) tek bir gezinmede union-find ile bulur.

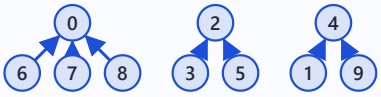
Bu uygulamaların hepsinde problem farklıdır, ama alttaki yapı aynıdır: gruplar yalnız büyür, hiç bölünmez ve sık sık “aynı grupta mı?” diye sorulur. Gruplardan öğe çıkarmak ya da bir grubu ikiye bölmek gerekiyorsa union-find uygun değildir; bu işlemleri desteklemez.

16.8 Sınav sorusu: ebeveyn dizisinin içeriği

Bu tür sorular, kodu ezberlemenin değil, elle izleyebilmenin ölçüsüdür. Soru şöyledir:

Örnek 16.2

`Birlesim_2` ve `Kume_Bul_2` verilmiştir. Kümeler $S_1 = \{0, 6, 7, 8\}$, $S_2 = \{1, 4, 9\}$ ve $S_3 = \{2, 3, 5\}$'tir (Şekil 16.13). (i) İlk başta ebeveyn dizisinin içeriği nedir? (ii) `Birlesim_2(4, 2); Kume_Bul_2(5); Birlesim_2(0, 4); Kume_Bul_2(7); Kume_Bul_2(3);` çağrılarından sonra ebeveyn dizisi ne olur? (i) hatalıysa (ii) değerlendirilmez.



Şekil 16.13: Sınav sorusundaki üç küme. Kökler 0, 2 ve 4.

(i) `Birlesim_2` kullanıldığı için köklerde −1 değil, eksi düğüm sayısı durur. Bu, sorunun tuzağıdır: (i)'de köklere −1 yazan öğrencinin (ii) cevabı da yanlış çıkar, çünkü ağırlık karşılaştırmaları yanlış sayılarla yapılır. S_1'de 4, S_2'de 3, S_3'te 3 öğe vardır:

ata = [−4, 4, −3, 2, −3, 2, 0, 0, 0, 4]

(ii) Çağrılar sırasıyla izlenir.

Çağrı	Gerekçe	`ata` sonrası
`Birlesim_2(4, 2)`	*−3 > −3* yanlış (eşit) → 2, 4'e bağlanır; `ata[4] = -6`	`[-4, 4, 4, 2, -6, 2, 0, 0, 0, 4]`
`Kume_Bul_2(5)`	Yol 5 → 2 → 4; 5 doğrudan 4'e bağlanır (2 zaten bağlı)	`[-4, 4, 4, 2, -6, 4, 0, 0, 0, 4]`
`Birlesim_2(0, 4)`	*−4 > −6* doğru → 0, 4'e bağlanır; `ata[4] = -10`	`[4, 4, 4, 2, -10, 4, 0, 0, 0, 4]`
`Kume_Bul_2(7)`	Yol 7 → 0 → 4; 7 köke bağlanır	`[4, 4, 4, 2, -10, 4, 0, 4, 0, 4]`
`Kume_Bul_2(3)`	Yol 3 → 2 → 4; 3 köke bağlanır	`[4, 4, 4, 4, -10, 4, 0, 4, 0, 4]`

Tablo 16.16: Sınav sorusunun (ii) kısmı adım adım.

Sonuç `[4, 4, 4, 4, -10, 4, 0, 4, 0, 4]` 'tür. On öğe tek kümede, 4 kökünde toplanmıştır. Yalnız 6 ve 8 hâlâ 0 üzerinden köke ulaşır; onlar için henüz bulma çağrılmamıştır. Bütün ara diziler sunumdaki kodla derlenen sürücüde aynen elde edildi.

İkinci adımdaki "2 zaten bağlı" notu, `Kume_Bul_2` 'nin fazladan turudur. Yol 5 → 2 → 4'tür; `while` döngüsü önce 5'i, sonra 2'yi köke bağlar. 2 zaten 4'e bağlı olduğu için ikinci atama değeri değiştirmez.

16.9 Karmaşıklık özeti

Sürüm	Birleşim	Küme bul (en kötü)	*n* − 1 birleşim + *m* bulma
`Birlesim_1` + `Kume_Bul_1`	$O(1)$	$O(n)$	$O(n + m \cdot n)$
`Birlesim_2` (ağırlıklı)	$O(1)$	$O(\lg n)$	$O(n + m \lg n)$
`Birlesim_2` + `Kume_Bul_2`	$O(1)$	$O(\lg n)$ tek işlem	$O(n + m \cdot \alpha(m + n, n))$

Tablo 16.17: Union-find sürümlerinin karmaşıklığı. *n* öğe sayısı, *m* bulma sayısıdır.

Bellek her sürümde aynıdır: yalnız *n* tam sayılık ebeveyn dizisi. Ağırlıklı kural düğüm sayısını kökteki negatif değerde sakladığı için ek dizi gerektirmez. $\alpha \le 4$ olduğundan son satır uygulamada doğrusaldır.

Tablonun özü tek cümleyle söylenebilir: birleşim tek başına hep *O(1)*'dir; maliyeti belirleyen ağacın derinliği, yani küme bulmadır. Ağırlıklı kural derinliği sınırlar, azaltma kuralı yolları kısaltır. Birleşimin sabit olması yanıltıcıdır, çünkü `Birlesim` kök ister ve kökü bulmak bir `Kume_bul` çağrısıdır. Uygulamada "iki öğeyi birleştir" isteği bu yüzden iki bulma artı bir birleşime mal olur.

16.10 Sık yapılan hatalar

1. **Birleşime kök yerine rastgele bir düğüm vermek.** `Birlesim_1(i, j)` 'de *i* kök değilse `ata[i] = j` ataması *i*'nin eski ebeveyniyle bağını koparır. *i* ve alt ağacı *j*'nin kümesine geçer; eski ağacın geri kalanı ise eski kümede kalır. İki küme birleşmemiş, biri bölünmüştür. Örnek ormanda `Birlesim_1(9, 2)` yalnız 9'u taşır; 1 ve 4 eski kümede kalır. Çözüm, önce `Kume_Bul` ile kökleri bulmaktır.
2. **Aynı kümedeki iki kökü, yani bir kökü kendisiyle birleştirmek** (*i == j*). `Birlesim_1(r, r)` kökü kendisine bağlar ve sonraki bulmalar sonsuz döngüye girer. `Birlesim_2(r, r)` düğüm sayısını iki katına çıkarır. Birleştirmeden önce kökler karşılaştırılmalıdır.
3. **Ağırlıklı kuralda negatif değerlerin karşılaştırmasını ters yorumlamak.** *−2 > −4*, "daha az düğüm" demektir. Ters karşılaştırma kümeleri bozmaz, ama ağaçları zincirleştirir ve hata yalnız ölçümle fark edilir.

4. **Birleşimde kökteki düğüm sayısını (`p`) günlemeyi unutmak.** Yeni kökte eski sayı kalır ve sonraki karşılaştırmalar yanlış ağırlıklarla yapılır. `p`'yi hücrelerden biri değiştirilmeden önce hesaplamak da aynı ölçüde önemlidir.
5. `Kume_Bul_2`**'de ebeveyni** `s`**'ye saklamadan bağlamak.** Yolun devamı kaybolur; yol kısaltılmadan kalır.
6. **Başlangıçta ebeveyn dizisini −1 ile doldurmamak.** Küresel diziler sıfırla başlar; `malloc` ile alınan dizinin içeriği ise belirsizdir. Sıfırla dolu bir dizide her düğümün ebeveyni 0'dır ve 0'ın ebeveyni de 0'dır: `Kume_Bul` 0'da sonsuz döngüye girer.
7. `malloc` **dönüşünü denetlememek ve** `free`**'yi unutmak.** Dinamik sürümde `ata` dizisi geri verilmezse program her çağrıda *4n* bayt sızdırır. Yapıyı geri vermeden önce iç diziyi geri vermek gerekir.

16.11 Alıştırmalar ve çözümleri

Alıştırma 1. *n = 6*, `ata` tümüyle −1. `Birlesim_2(0, 1); Birlesim_2(2, 3); Birlesim_2(0, 2); Birlesim_2(4, 5);` çağrılarından sonra `ata` nedir?

Çözüm. İlk iki çağrı eşit ağaçları birleştirir: `ata[1] = 0`, `ata[0] = -2`; `ata[3] = 2`, `ata[2] = -2`. Üçüncü çağrıda iki kökte −2 vardır; eşitlikte 2, 0'ın altına girer ve `ata[0] = -4` olur. Dördüncü çağrı `ata[5] = 4`, `ata[4] = -2` yazar. Sonuç `[-4, 0, 0, 2, -2, 4]`'tür: 0 kökünde 4, 4 kökünde 2 düğüm. Sürücüyle doğrulandı.

Alıştırma 2. Ağırlıklı kural kullanılırken 16 düğümlü bir ağacın derinliği en çok kaçtır?

Çözüm. Teorem 16.2'ye göre *⌊lg 16⌋ + 1 = 5* düzey. Sınıra ulaşan ağaç, eşit ikililerle kurulur: 8 ikili, 4 dörtlü, 2 sekizli, 1 on altılı. Ölçümde de 5 düzey çıktı.

Alıştırma 3. Ağırlık yerine yükseklik tutan bir birleşim kuralı yazınız. Ne değişir?

Çözüm. Kökte −(düzey sayısı) tutulur. Alçak ağaç yüksek olanın altına bağlanır; yükseklikler eşitse biri ötekinin altına girer ve düzey sayısı 1 artar. Bu yönteme **rütbeye göre birleştirme** (*union by rank*) denir:

```
void Birlesim_3(int i, int j) {
  /* yükseklik kuralı; i ve j kök; kökte -(düzey sayısı) */
  if (ata[i] > ata[j]) {        /* i'nin ağacı daha alçak */
    ata[i] = j;
  } else if (ata[i] < ata[j]) { /* j'nin ağacı daha alçak */
    ata[j] = i;
  } else {                      /* eşit yükseklik */
    ata[j] = i;
    ata[i]--;                   /* düzey sayısı 1 artar */
  }
}
```

Karşılaştırma yine negatif değerler arasındadır: büyük değer alçak ağacı gösterir. Farklı yükseklikte iki ağaç birleşince yükseklik değişmez, çünkü alçak ağaç yüksek olanın kökünün altına girer ve en çok onun yüksekliğine ulaşır. Alıştırma 1'in işlem dizisiyle sonuç `[-3, 0, 0, 2, -2, 4]`'tür: aynı ağaçlar, ama 0 kökünde düğüm sayısı 4 yerine düzey sayısı 3 durur. 16 düğümlük eşit ikililerde yine 5 düzey çıkar. Derinlik yine *O(lg n)* ile sınırlıdır; kanıtı ağırlıklı kuralınkiyle aynıdır (*d* düzeyli ağaçta en az 2^{d-1} düğüm). Azaltma kuralıyla birlikte kullanıldığında yol kısaltma yüksekliği düşürür ama kök bunu bilmez; saklanan değer artık gerçek yükseklik değil, onun bir üst sınırıdır. Bu yüzden ona yükseklik yerine “rütbe” denir.

Alıştırma 4. `Kume_Bul_2`**'yi özyinelemeli yazınız.**

Çözüm. Tek satırlık yol kısaltma: `int bul(int i) { return ata[i] < 0 ? i : (ata[i] = bul(ata[i])); }`. Ayrıntılı anlatımı ve bellek karşılaştırması 16.5.5'tedir.

Alıştırma 5. Kruskal algoritmasında bir kenarın reddedilmesi neden döngü demektir?

Çözüm. Kenar, uçları aynı kümedeyse reddedilir. Aynı kümede olmak, iki uç arasında daha önce seçilmiş kenarlardan oluşan bir yol bulunması demektir; çünkü kümeler yalnız seçilen kenarlarla birleşir. Yeni kenar bu yolun iki ucunu doğrudan bağlar ve yolu bir döngüye kapatır.

Alıştırma 6. *n* düğümlü, *m* kenarlı bir çizgede bileşen sayısını bulmanın maliyeti nedir?

Çözüm. Ebeveyn dizisini kurmak *O(n)*, kenar döngüsü *2m* bulma ve en çok *n − 1* birleşimdir. İki kural birlikte kullanılınca toplam *O(n + m · α(m + n, n))*'dir; uygulamada doğrusaldır.

16.12 Özet

- **Model.** Ayrık kümeler, kökü küme adı olan ve bağları çocuktan ebeveyne giden ağaçlarla gösterilir. İki işlem vardır: birleşim ve küme bulma. İki öğe, kökleri aynıysa aynı kümededir.
- **Gösterim.** Bütün orman tek bir `int` dizisinde durur: `ata[i]`, *i*'nin ebeveynidir; kökte negatif bir değer bulunur. Dizi `malloc` ile kurulursa her ayırmadan sonra `NULL` denetlenir ve bloklar ters sırayla geri verilir.
- **İlk sürüm.** `Birlesim_1` tek atamadır; `Kume_Bul_1` köke kadar yürür ve düğümün derinliği kadar sürer.
- **Kötü durum.** Kuralsız (keyfî) bağlama zincir üretebilir: `Birlesim_1(i, i+1)` dizisinden sonra *n* bulma tam $n(n-1)/2$ ilerleme, yani $\Theta(n^2)$ sürer. Rastgele testler bu girdiyi kolayca kaçırır.
- **Ağırlıklı kural.** Küçük ağaç büyüğün altına bağlanır. Düğüm sayısı kökte eksi işaretle saklanır; ek dizi gerekmez. Derinlik $\lfloor \lg m \rfloor + 1$'i aşmaz ve sınır eşit ikililerle tam olarak yakalanır.
- **Azaltma kuralı.** Bulma sırasında yol üzerindeki her düğüm doğrudan köke bağlanır. İlk çağrı yaklaşık iki kat pahalıdır, ama sonraki aramalar bir çırpıda biter. Eski ebeveyn bağlamadan önce saklanmalıdır.
- **Sonuç.** İki kural birlikte işlem başına α ile orantılı maliyet verir (Tarjan–van Leeuwen). Ters Ackermann fonksiyonu $n \le 2^{65533}$ için 4'ü aşmaz; maliyet uygulamada neredeyse $O(1)$'dir.
- **Uygulamalar.** Bileşen sayma, eşdeğerlik sınıfları ve Kruskal algoritması aynı kalıbı kullanır: iki ucun kökünü bul, farklıysa birleştir, aynıysa döngü var.

16.13 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

ayrık kümeler *disjoint sets* · union-find *union-find* · birleşim *union* · küme bulma *find* · orman *forest* · kök *root* · ebeveyn *parent* · ebeveyn dizisi *parent array* · ağırlıklı kural *weighting rule* · rütbeye göre birleştirme *union by rank* · azaltma kuralı *collapsing rule* · yol kısaltma *path compression* · en kötü durum *worst case* · amortize analiz *amortized analysis* · Ackermann fonksiyonu *Ackermann function* · ters Ackermann fonksiyonu *inverse Ackermann function* · çağrı yığını *call stack* · yığın taşması *stack overflow* · bağlantılı bileşen *connected component* · minimum kapsayan ağaç *minimum spanning tree* · bellek sızıntısı *memory leak* · sarkan işaretçi *dangling pointer* · işaretçiye işaretçi *pointer to pointer*

16.14 Kendini Dene

1. `ata = [-1, 0, 0, 1, 1, -1, 5]` dizisinde kaç küme vardır? `Kume_Bul_1(4)` neyi döndürür ve kaç ilerleme yapar?

   Cevap

   Negatif değerli iki hücre (0 ve 5) vardır; iki küme: {0, 1, 2, 3, 4} ve {5, 6}. `Kume_Bul_1(4)` 4 → 1 → 0 yolunu izler, 2 ilerlemeyle 0'ı döndürür.

2. $n = 8$ tek öğeli kümeye `Birlesim_1(i, i+1)` çağrıları $i = 0, \ldots, 6$ için uygulanıyor. Ardından her öğe için bir `Kume_Bul_1` çağrılıyor. Toplam kaç ilerleme yapılır?

   Cevap

   $n(n-1)/2 = 8 \cdot 7 / 2 = 28$. `Kume_Bul_1(0)` 7, `Kume_Bul_1(1)` 6 … `Kume_Bul_1(7)` 0 ilerleme yapar.

3. `ata = [-3, 0, 0, -5, 3, 3, 3, 3]` iken `Birlesim_2(0, 3)` çağrılıyor. Dizi ne olur?

   Cevap

   $p = -8$. $-3 > -5$ doğrudur: 0'ın ağacı daha küçüktür ve 3'ün altına girer. Sonuç `[3, 0, 0, -8, 3, 3, 3, 3]`.

4. `ata = [-8, 0, 0, 2, 0, 4, 4, 6]` iken `Kume_Bul_2(3)` çağrılıyor. Döngüler kaçar tur döner, dizi ne olur?

   Cevap

   Yol 3 → 2 → 0'dır. `for` döngüsü 2 ilerleme yapar, `while` döngüsü 2 tur döner: 3 köke bağlanır, 2 için fazladan tur yapılır. Sonuç `[-8, 0, 0, 0, 0, 4, 4, 6]`, dönüş değeri 0.

5. Ağırlıklı kuralla kurulmuş 1024 düğümlü bir ağaçta tek bir `Kume_Bul_1` en çok kaç ilerleme yapar?

   Cevap

   Düzey sayısı en çok *⌊lg 1024⌋ + 1 = 11*'dir. En derin düğümden köke 10 kenar vardır: en çok 10 ilerleme.

6. *n = 1000* için *α(n, n)* kaçtır? (Tanımdaki *A* ile.)

   Cevap

   ⌊n/n⌋ = 1, *lg 1000 ≈ 9,97*. *A(1, 1) = 3* ve *A(2, 1) = 5* yetmez; *A(3, 1) = 13 ≥ 9,97*. Cevap 3.

7. `ackermann(2, 1)` neyi döndürür ve toplam kaç çağrı yapılır?

   Cevap

   A(2, 1) = 2 · 1 + 3 = 5. Sayaçlı çalıştırmada 14 çağrı yapıldı.

8. 5 düğümlü bir çizgenin kenarları (0, 1), (1, 2), (2, 0), (3, 4)'tür. `bilesen_say` neyi döndürür? Hangi kenar sayıyı değiştirmez?

   Cevap

   2 döndürür: {0, 1, 2} ve {3, 4}. (2, 0) kenarının iki ucu da 0 köküne ulaşır; kenar 0–1–2 döngüsünü kapatır ve sayı değişmez.

9. `kumeler_sil` parametresini `kumeler **kp` yerine `kumeler *k` olarak alsaydı ne kaybedilirdi?

   Cevap

   Yordam blokları yine geri verebilirdi, ama çağıranın işaretçisini `NULL` yapamazdı. Çağıranın değişkeni geri verilmiş belleği göstermeye devam eder; yani sarkan işaretçi olur. İkinci bir silme çağrısı aynı bloğu iki kez geri vermeye çalışır ve tanımsız davranış doğar.

16.15 İleri okuma

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, Kesim 5.10 (Set Representation). Ebeveyn dizisi, ağırlıklı kural ve azaltma kuralı bu notla aynı sırada, lemmalarıyla anlatılır.
- T. H. Cormen ve diğerleri, *Introduction to Algorithms*, “Data Structures for Disjoint Sets” bölümü. Rütbeye göre birleştirme ve yol kısaltmanın *α*'lı sınırının ayrıntılı kanıtını verir.
- R. Sedgewick, *Algorithms in C*, Bölüm 1. Union-find'ı ağ bağlantısı örneğiyle adım adım geliştirir ve sürümleri ölçümlerle karşılaştırır.
- R. E. Tarjan, J. van Leeuwen, “Worst-case analysis of set union algorithms”, *Journal of the ACM* 31(2), 1984. Teorem 16.3'ün kaynağıdır; birçok yol kısaltma çeşidini birlikte inceler.

16.16 Kaynaklar

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, Kesim 5.10 (Set Representation).
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (CLRS), “Data Structures for Disjoint Sets” bölümü.
3. R. Sedgewick, *Algorithms in C*, Bölüm 1 (union-find ile bağlantılılık).
4. R. E. Tarjan, J. van Leeuwen, “Worst-case analysis of set union algorithms”, *J. ACM* 31(2), 1984.

KISIM IV · BÖLÜM 17

Minimum Kapsayan Ağaçlar

Bir çizgenin bütün düğümlerini döngü kurmadan birbirine bağlayan alt çizgeye kapsayan ağaç denir. Kenarların bir maliyeti varsa bu ağaçlar arasından toplam maliyeti en küçük olanı aranır. Köyleri en az kabloyla elektrik şebekesine bağlamak ya da bilgisayarları en ucuz hatlarla ağa katmak bu problemin örnekleridir. Not, kapsayan ağacın tanımı, sayısı ve özellikleriyle başlar. Ardından iki klasik açgözlü algoritma gelir. Kruskal algoritması kenarları ucuzdan pahalıya dener ve döngüyü kümelerle (union-find) yakalar. Prim algoritması tek bir ağacı en ucuz kenarla büyütür ve dizilerle ya da bir heap ile kodlanır. Her iki algoritmanın C kodu satır satır anlatılır, aynı örnek çizge üzerinde adım adım izlenir ve maliyeti gerekçesiyle hesaplanır. Notun asıl dersi şudur: aynı algoritma, seçilen veri yapısına göre bambaşka bir hızla çalışır.

Kod (C): `github.com/sercankulcu/data-structures-and-algorithms-c/Bolum17`

ÖĞRENME ÇIKTILARI

Tanımlamak
Kapsayan ağacı, ağaç kenarı ile ağaç dışı kenarı ve derinlik ya da genişlik öncelikli aramanın ürettiği kapsayan ağaçları açıklamak.

Saymak
Bir çizgenin kapsayan ağaçlarını üretme yollarını, eş biçimlilik kavramını ve ağaç sayısının neden hızla büyüdüğünü tartışmak.

Kruskal
Kenarları artan maliyetle seçen ve döngüyü union-find yapısıyla yakalayan Kruskal algoritmasını C ile kodlamak ve elle izlemek.

Prim
Tek bir ağacı büyüten Prim algoritmasını $O(n^2)$ süreli dizilerle ve heap ile gerçeklemek.

Analiz
$O(e \log e)$, $O(n^2)$ ve $O(e \log n)$ sınırlarını karşılaştırmak; yoğun ve seyrek çizgeye göre algoritma seçmek.

Yaklaşım
Min-heap ve ayrık kümelerin bir algoritmayı nasıl hızlandırdığını görmek: önce veri yapısını keşfetmek, sonra kodlamak.

17.1 Kapsayan ağaçlar

Bir çizge $G = (V, E)$, düğümlerden (V) ve düğüm çiftlerini birleştiren kenarlardan (E) oluşur. Notta düğüm sayısı $n = |V|$, kenar sayısı $e = |E|$ ile gösterilir ve aksi söylenmedikçe çizge yönsüzdür. Her düğüm çiftinin arasında bir yol varsa çizge **bağlantılıdır** (*connected*). Başladığı düğüme geri dönen yola **döngü** (*cycle*) denir. Bağlantılı ve döngüsüz bir çizge **ağaçtır**; ağaçta kök seçmek zorunlu değildir.

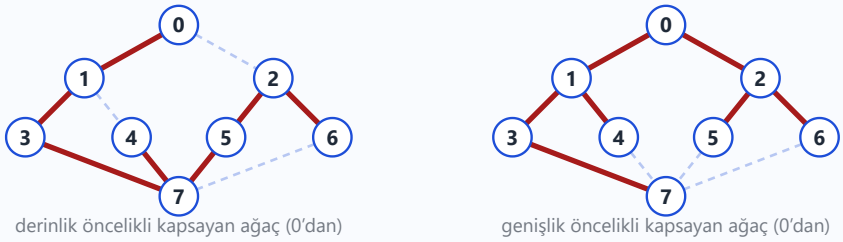
Bu notun bütün sorusu şu cümleye sığar: bir çizgenin düğümlerini birbirine bağlamak için kenarlarının hangileri yeter ve bunların en ucuzu hangisidir? Önce “yeter” sözcüğünü kesinleştirelim.

17.1.1 Gezinme kenarları ikiye ayırır

Bir çizgede gezinmenin iki temel yolu vardır. **Derinlik öncelikli arama** (*depth-first search*, DFS) bir komşuya geçer ve olabildiğince derine iner; gidecek yer kalmayınca geri döner. Özyinelemeyle ya da bir yığınla yazılır. **Genişlik öncelikli arama** (*breadth-first search*, BFS) başlangıç düğümünün bütün komşularını, sonra onların komşularını ziyaret eder; bir kuyrukla yazılır. İkisi de her düğümü bir kez işaretler ve komşuluk listesiyle $O(n + e)$ sürede biter.

Bağlantılı bir çizgede gezinme bütün düğümleri ziyaret edince kenarlar iki gruba ayrılmış olur:

- **T: ağaç kenarları** (*tree edges*). Gezinmenin işlediği, yani daha önce görülmemiş bir düğüme götüren kenarlardır. Başlangıç dışındaki her düğüme tam bir kez böyle bir kenarla varılır. Bu yüzden ağaç kenarlarının sayısı tam olarak $n - 1$'dir.
- **N: ağaç dışı kenarlar** (*non-tree edges*). Gezinmenin baktığı ama işlemediği kenarlardır; öteki ucu zaten işaretlidir. DFS'de bunlara geri kenar (*back edge*), BFS'de çapraz kenar (*cross edge*) da denir.



Şekil 17.1: Sekiz düğümlü örnek çizgenin 0'dan başlayan DFS (solda) ve BFS (sağda) kapsayan ağaçları. Kalın kırmızı kenarlar ağaç kenarlarıdır, kesikli olanlar ağaç dışı kenarlardır. DFS: N = {(0,2), (1,4), (6,7)}; BFS: N = {(4,7), (5,7), (6,7)}.

Şekil 17.1'deki çizgenin 10 kenarı vardır: (0,1), (0,2), (1,3), (1,4), (2,5), (2,6), (3,7), (4,7), (5,7), (6,7). Komşuluk listeleri küçükten büyüğe sıralıdır. Tablo 17.1 DFS'nin hangi kenarı neden işlediğini ya da atladığını sırasıyla gösterir.

Sıra	Bulunulan düğüm	Bakılan kenar	Öteki uç	Sonuç
1	0	(0,1)	1 yeni	ağaç
2	1	(1,0)	0 işaretli	geldiği kenar
3	1	(1,3)	3 yeni	ağaç
4	3	(3,7)	7 yeni	ağaç
5	7	(7,4)	4 yeni	ağaç
6	4	(4,1)	1 işaretli	dışı: (1,4)
7	7	(7,5)	5 yeni	ağaç
8	5	(5,2)	2 yeni	ağaç
9	2	(2,0)	0 işaretli	dışı: (0,2)
10	2	(2,6)	6 yeni	ağaç
11	6	(6,7)	7 işaretli	dışı: (6,7)

Tablo 17.1: Şekil 17.1'deki çizgede 0'dan başlayan DFS. Her satır bir kenara bakıştır; "ağaç" işlenen, "dışı" işaretli bir düğüme çıktığı için atlanan kenardır.

Yedi ağaç kenarı (0,1), (1,3), (3,7), (7,4), (7,5), (5,2), (2,6)'dır; $n - 1 = 7$ beklentisiyle örtüşür. Geri dönüşlerde bakılan öteki kenarlar (ör. 3'ün listesindeki 1) yeni bilgi vermediği için tabloya yazılmadı. BFS aynı çizgede 0'dan başlayınca önce 1 ve 2'yi, sonra 3, 4, 5, 6'yı, en son 7'yi işaretler. Ağaç kenarları (0,1), (0,2), (1,3), (1,4), (2,5), (2,6), (3,7) olur. 7'ye ilk ulaşan kenar (3,7)'dir; (4,7), (5,7), (6,7) ağaç dışında kalır.

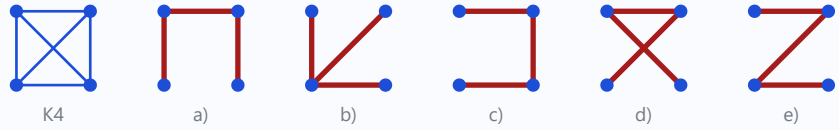
İki gezinme aynı çizgeden iki farklı ağaç çıkardı. İkisi de bütün düğümleri kapsar, ikisinde de döngü yoktur. Demek ki "bütün düğümleri bağlamaya yeten kenarlar" tek bir küme değildir. Başlangıç düğümü ya da komşuların sırası değişince yine başka ağaçlar çıkar.

17.1.2 Kapsayan ağacın tanımı

Tanım 17.1

$G = (V, E)$ bağlantılı bir çizge olsun. G'nin bütün düğümlerini içeren ve kenarları E'den seçilmiş bir ağaca G'nin **kapsayan ağacı** (*spanning tree*) denir. Kapsayan ağaçta n düğüm ve $n - 1$ kenar vardır; döngü yoktur.

Tanımdaki iki koşul birbirini tamamlar. “Bütün düğümleri içerir” koşulu ağacın kapsayıcı olmasını sağlar: hiçbir düğüm dışarıda kalmaz. “Kenarları E'den seçilir” koşulu yeni bağlantı uydurmayı yasaklar: yalnız çizgede var olan kenarlar kullanılabilir. Bağlantılı olmayan bir çizgenin kapsayan ağacı yoktur. Böyle bir çizgede her bileşenin ayrı ayrı kapsayan ağacı bulunur ve bunların topluluğuna **kapsayan orman** (*spanning forest*) denir.



Şekil 17.2: Dört düğümlü tam çizge K_4 ve kapsayan ağaçlarından beşi. a, c, d, e birer yoldur; b bir yıldızdır.

Şekil 17.2'nin solundaki K_4 **tam çizgedir** (*complete graph*): her düğüm çifti bir kenarla bağlıdır, toplam $4 \cdot 3 / 2 = 6$ kenar vardır. Yanındaki beş çizimin her biri bu altı kenardan üçünü seçer. Hepsinde dört köşe de kullanılır ve hiçbirinde döngü yoktur; beşi de kapsayan ağaçtır.

Bu beş ağaçtan a, c, d ve e aynı biçimdedir: dört düğüm tek bir çizgi üzerinde sıralanmıştır. Yalnız hangi köşenin yolun neresinde durduğu değişir. b ise farklıdır: bir köşe öteki üçüne doğrudan bağlanmıştır. Birinci biçime **yol**, ikincisine **yıldız** denir.

Tanım 17.2

İki çizgenin düğümleri birebir eşlenebiliyor ve bu eşleme altında birinin kenarları tam olarak ötekinin kenarlarına karşılık geliyorsa iki çizge **eş biçimlidir** (*isomorphic*). Eş biçimlilik “izomorfizm” adıyla da anılır ve $\equiv$ ile gösterilir.

Şekildeki ağaçlar için $a \equiv c \equiv d \equiv e$ yazılır; b bunların hiçbiriyle eş biçimli değildir. Bunu görmenin kısa yolu derecelere bakmaktır. Yolda iki düğümün derecesi 1, ikisininki 2'dir. Yıldızda bir düğümün derecesi 3, üçününki 1'dir. Eş biçimli iki çizgede derece dizileri aynı olmak zorundadır; burada farklı olduğu için a ile b eş biçimli olamaz. Tersi doğru değildir: derece dizileri aynı olan iki çizge eş biçimli olmayabilir. Bu yüzden derece karşılaştırması eş biçimliliği ancak çürütebilir, kanıtlayamaz.

K_4'ün kaç kapsayan ağacı olduğunu saymak kolaydır. Altı kenardan üçü $C(6, 3) = 20$ biçimde seçilir. Bu seçimlerin dördü üçgendir: dört köşeden üçünü seçip aralarındaki üç kenarı almak döngü kurar ve dördüncü köşeyi dışarıda bırakır. Geriye kalan 16 seçimin hepsi ağaçtır. Yıldızların sayısı merkez seçimine eşittir: 4. Yolların sayısı dört düğümün sıralamalarından bulunur. $4! = 24$ sıralama vardır, ama her yol iki kez (baştan ve sondan) okunur. Yol sayısı $24 / 2 = 12$'dir. Toplam $12 + 4 = 16$ eder.

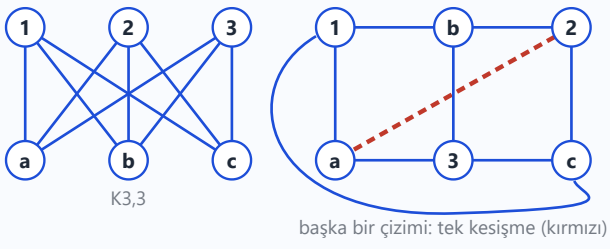
Teorem 17.1 (Cayley formülü)

n düğümlü tam çizge K_n'nin kapsayan ağaçlarının sayısı n^{n-2}'dir.

Formül $n = 4$ için $4^2 = 16$ verir ve yukarıdaki sayımla örtüşür. Sayı çok hızlı büyür: K_5'in 125, K_{10}'un yüz milyon kapsayan ağacı vardır. Kanıtı bu notun kapsamı dışındadır. Burada önemli olan sonuçtur: kapsayan ağaçları tek tek üretip en iyisini seçmek, küçük çizgelerde bile umutsuz bir iştir. Kenarlara ağırlık verildiğinde “kaç tane?” sorusu “hangisi en ucuz?” sorusuna dönüşür ve bu sayının büyüklüğü yeniden karşımıza çıkar.

17.1.3 Eş biçimlilik, çizim ve düzlemsellik

Bir çizge, kâğıda çizilmiş hâlinden ibaret değildir. Çizgeyi düğümleri ve kenarları belirler; düğümlerin kâğıttaki yeri bir tercihtir. Aynı çizge birbirine hiç benzemeyen biçimlerde çizilebilir. Şekil 17.3 bunun çarpıcı bir örneğidir.



Şekil 17.3: Aynı çizgenin ($K_{3,3}$) iki çizimi. Solda dokuz kenar birçok noktada kesişir. Sağdaki çizimde yalnız kırmızı kesikli (a,2) kenarı (b,3) kenarını keser.

$K_{3,3}$, iki gruba ayrılmış altı düğümden oluşur: {1, 2, 3} ve {a, b, c}. Her grubun her düğümü öteki grubun üç düğümüne bağlıdır, kendi grubundakilere bağlı değildir. Toplam $3 \cdot 3 = 9$ kenar vardır. Soldaki çizim bu tanımı doğrudan uygular ve kenarlar birbirine karışır. Sağdaki çizimde düğümler başka yerlere konmuş, (1,c) kenarı dışarıdan dolaştırılmıştır. Dokuz kenarın hepsi hâlâ oradadır, ama artık yalnız bir kesişme kalmıştır.

İki çizimin aynı çizge olduğunu göstermek, eş biçimliliği göstermektir: soldaki her düğüme sağda bir düğüm karşılık gelir ve kenarlar karşılıklı eşlenir. Buradaki eşleme kolaydır, çünkü etiketler korunmuştur. Etiketsiz iki çizim verildiğinde ise problem zorlaşır. Derece dizisi gibi basit ölçütler yetmez; DFS ya da BFS ile düğümleri katmanlara ayırmak eşlemeyi daraltmaya yardım eder, ama genel çözüm vermez. Çizge eş biçimliliği, verimli bir algoritması bilinip bilinmediği uzun süre tartışılmış ünlü bir problemdir. Hoca bu problemi güzel bir dördüncü sınıf projesi olarak önerir: üç boyutlu bir gösterimle bir çizimi ötekine dönüştüren, hangi düğümlerin yer değiştirmesi gerektiğini bulan bir program. Ardından bu tür soruların gerçekten zor olduğunu, ciddi biçimde ancak yüksek lisans ya da doktorada uğraşılabileceğini ekler. Vurgulanan, problemin çözülemez olduğu değil, emek istediğidir.

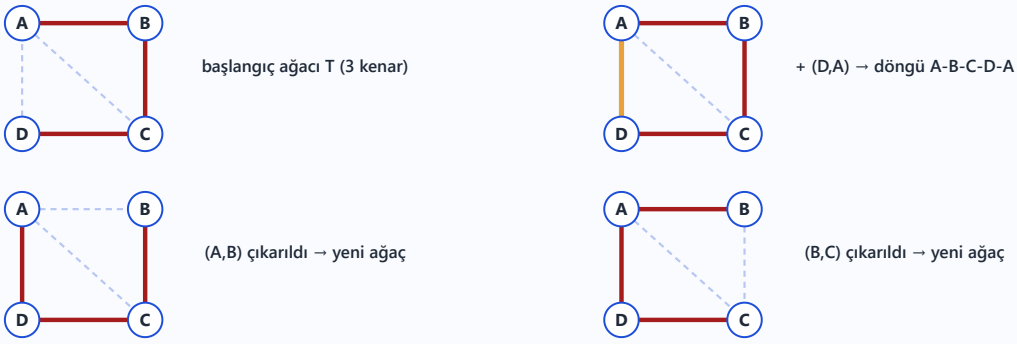
Kesişme sorusunun da bir adı vardır. Kenarları hiç kesişmeden çizilebilen çizgeye **düzlemsel** (*planar*) denir. $K_{3,3}$ düzlemsel değildir: düğümler nasıl yerleştirilirse yerleştirilsin, en az bir kesişme kalır. Sağdaki çizim bu alt sınıra ulaşmıştır. Sorunun mühendislikte somut bir karşılığı vardır. Bir baskılı devre kartında iletken yollar aynı katmanda kesişemez. $K_{3,3}$ biçiminde bağlanması gereken bir devre tek katmana sığmaz, en az iki katman ister.

17.1.4 Kaç farklı kapsayan ağaç vardır?

Kapsayan ağaçları üretmenin en kaba yolu bütün olası kenar kümelerini denemektir. Kapsayan ağaçta tam $n - 1$ kenar bulunduğundan e kenar arasından $n - 1$ tanesi seçilir. Bu, $C(e, n - 1)$ biçimde yapılabilir. Her seçim için döngü olup olmadığı denetlenir; döngüsüz olanlar ağaçtır. Eş biçimli ağaçlar ayrıca sayılmak istenmiyorsa bir de eş biçimlileri ayıklamak gerekir.

Kare biçimli dört düğümlü bir çizge düşünelim: A, B, C, D köşeleri sırayla bağlı, ayrıca A ile C arasında bir köşegen var. Beş kenardan üçü $C(5, 3) = 10$ biçimde seçilir. Bunlardan ikisi üçgendir (ABC ve ACD) ve döngü kurar; geriye 8 kapsayan ağaç kalır.

Daha akıllıca bir yöntem, bir kapsayan ağaçtan başlayıp kenar değiştirerek öteki ağaçlara geçmektir. Şekil 17.4 yöntemi adım adım gösterir.



Şekil 17.4: Döngü değiştirerek kapsayan ağaç üretmek. Sol üstte DFS ile bulunmuş ağaç T. Sağ üstte ağaç dışı (D,A) kenarı eklenmiş ve A-B-C-D-A döngüsü oluşmuştur. Alt sırada döngüden (A,B) ve (B,C) çıkarılarak iki yeni ağaç elde edilmiştir.

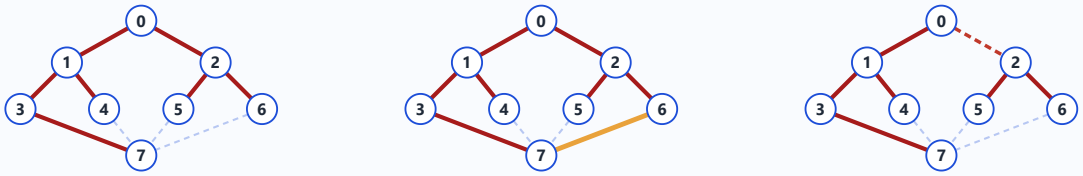
1. Önce bir kapsayan ağaç bulunur, örneğin DFS ile. Şekilde T = {(A,B), (B,C), (C,D)}'dir.
2. Ağaç dışı bir kenar eklenir. Ağaçta her düğüm çifti arasında tek yol olduğu için yeni kenar tam olarak bir döngü kurar. (D,A) eklenince döngü A-B-C-D-A olur.
3. Döngüdeki kenarlardan biri çıkarılır. Döngü kırılır, düğümler yine bağlı kalır; sonuç yeni bir kapsayan ağaçtır. (A,B) çıkarılınca {(B,C), (C,D), (D,A)} elde edilir. (A,B) geri konup (B,C) çıkarılınca bir başkası, (C,D) çıkarılınca bir üçüncüsü bulunur.
4. İşlem bütün ağaç dışı kenarlar için ve bulunan her yeni ağaç için tekrarlanır. Bu çizgede sonunda 8 kapsayan ağacın hepsine ulaşılır.

Yöntem çalışır ama uzundur. Aynı ağaca farklı yollardan defalarca varılır ve her yeni ağacın daha önce bulunup bulunmadığı ayrıca denetlenmelidir. Bu yüzden ağaç saymak ya da en ucuz ağacı bulmak için tercih edilmez. Kesin sayı gerektiğinde Kirchhoff'un matris-ağaç teoremi kullanılır. Bu teorem, çizgenin derece ve komşuluk bilgisinden kurulan bir matrisin determinantını alarak sayıyı tek hamlede

verir. Teoreme göre Şekil 17.1'deki sekiz düğümlü çizgenin 64, notun ana örneği olan yedi köy çizgesinin 46 kapsayan ağacı vardır. Teoremin ayrıntısı ileri bir konudur.

17.1.5 Kapsayan ağacın özellikleri

Kapsayan ağacın algoritmalarda kullanılan dört özelliği vardır. Şekil 17.5 bunları BFS kapsayan ağacı üzerinde gösterir.



Şekil 17.5: Kapsayan ağacın özellikleri. Solda BFS kapsayan ağacı: 8 düğüm, 7 kenar. Ortada ağaç dışı (6,7) kenarı eklenmiş ve 6-2-0-1-3-7-6 döngüsü oluşmuştur. Sağda ağaç kenarı (0,2) silinmiş, {2, 5, 6} ağaçtan kopmuştur.

Teorem 17.2 (Ağacın özellikleri)

n düğümlü bir *T* çizgesi için şu üç özellikten herhangi ikisi üçüncüsünü gerektirir: (i) *T* bağlantılıdır; (ii) *T*'de döngü yoktur; (iii) *T*'nin tam *n − 1* kenarı vardır. Bu durumda *T* bir ağaçtır ve şunlar da geçerlidir:

1. Her düğüm çifti arasında tek bir yol vardır.
2. Ağaca bir kenar eklemek tam olarak bir döngü oluşturur.
3. Ağaçtan herhangi bir kenarı silmek ağacı iki parçaya böler.

İspat. (i) ve (ii)'den (iii)'e: Bağlantılı ve döngüsüz bir çizgede en az bir yaprak (derecesi 1 olan düğüm) vardır. En uzun yolun bir ucu yaprak olmak zorundadır; öyle olmasaydı yol uzatılabilir ya da bir döngü kapanırdı. Yaprağı ve ona değen tek kenarı silmek bağlantıyı bozmaz ve döngü yaratmaz. Bu silme *n − 1* kez tekrarlanınca tek düğüm ve sıfır kenar kalır; demek ki başta *n − 1* kenar vardı. Tek yol özelliği de buradan çıkar: iki düğüm arasında iki farklı yol olsaydı bu yollar birleşip bir döngü kurardı. Bir kenar *(u, v)* eklenince *u* ile *v* arasındaki tek ağaç yolu yeni kenarla kapanır ve tam olarak bir döngü oluşur. Bir kenar silinince iki ucu arasındaki tek yol kopar ve uçlar ayrı parçalarda kalır. Öteki iki yön benzer sayımlarla gösterilir. ■

Şekilde bu özellikler tek tek görülür. Sol çizimdeki BFS ağacı 8 düğümü 7 kenarla bağlar. Ortadaki çizimde (6,7) eklenince 6'dan 7'ye ağaç üzerindeki tek yol 6-2-0-1-3-7 ile yeni kenar birleşir ve 6-2-0-1-3-7-6 döngüsü kapanır. Bu çizge artık ağaç değildir. Sağdaki çizimde (0,2) silinince 2, 5 ve 6 düğümleri 0'a bağlanamaz hâle gelir. Kapsayan ağaç, düğümleri bağlı tutan alt çizgeler arasında en az kenarlı olanıdır. Bir kenar fazlası döngü, bir kenar eksiği kopukluk demektir.

İkinci ve üçüncü özellik birlikte kullanılınca kapsayan ağaç üretme yöntemi ortaya çıkar: ağaç dışı bir kenar eklenir, oluşan tek döngüden bir kenar çıkarılır. Minimum kapsayan ağaç algoritmalarının doğruluk kanıtları da aynı iki özelliğe dayanır.

Kapsayan ağaçların uygulamaları da bu özelliklerden gelir:

- **Gereksiz bağlantıları atmak.** Bir bilgisayar ağında döngüler, yayınlanan bir paketin sonsuza kadar dolaşmasına yol açabilir. Ağın bir kapsayan ağacı seçilir ve yalnız onun hatları kullanılır.
- **Yayın ağacı.** Bir kaynaktan bütün düğümlere ileti göndermek (*broadcast*) için kapsayan ağacın kenarları izlenir; her düğüm iletiyi bir kez alır.
- **Elektrik devreleri.** Kirchhoff, devre denklemlerinin bağımsız bir kümesini bulmak için kapsayan ağaçları kullanmıştır. Her ağaç dışı kenar tek bir döngü kurar; bu döngülerin her biri bağımsız bir gerilim denklemi verir.

17.2 Minimum maliyetli kapsayan ağaç

17.2.1 Ağırlıklı çizge ve problem

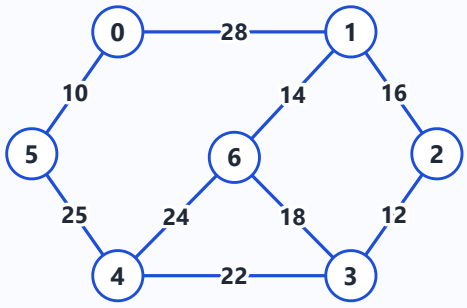
Gerçek problemlerde kenarlar eşit değildir. İki şehir arasındaki yolun bir uzunluğu, iki bilgisayar arasındaki hattın bir kira bedeli, iki köy arasına çekilecek kablonun bir maliyeti vardır. Kenarlarına böyle bir sayı atanmış çizgeye **ağırlıklı çizge** (*weighted graph*) denir ve *G(V, E)* yerine *G(V, E, w)* yazılır. Burada *w(u, v)*, *(u, v)* kenarının ağırlığıdır. Bu notta ağırlığa maliyet de denir.

Tanım 17.3

Bağlantılı ve ağırlıklı bir çizgenin kapsayan ağacının **maliyeti**, ağaçtaki kenarların ağırlıkları toplamıdır. Maliyeti en küçük olan kapsayan ağaca **minimum kapsayan ağaç** (*minimum spanning tree*, MST) denir.

Tanım "en küçük olan" der, "en küçük olanlar" demez; yine de birden çok kapsayan ağaç aynı en küçük maliyeti verebilir. Kenar ağırlıklarının hepsi farklıysa minimum kapsayan ağaç tektir. Eşit ağırlıklar varsa birden çok ağaç olabilir, ama hepsinin maliyeti aynıdır.

İki örnek problemi somutlaştırır. Birincisinde düğümler bir kurumun bilgisayarlarıdır, kenar ağırlıkları iki bilgisayar arasına hat çekmenin maliyetidir. Bütün bilgisayarları birbirine ulaştıran en ucuz hat kümesi aranır. İkincisinde yedi köye elektrik ya da fiber optik kablo götürülecektir. Fizibilite çalışması hangi köyden hangisine kaç kilometre kablo gerektiğini çıkarmıştır. Arada dağ ya da göl olduğu için doğrudan bağlanamayan köy çiftlerinin maliyeti sonsuzdur; bu kenarlar çizgeye hiç yazılmaz.



Şekil 17.6: Notun ana örneği: yedi köy (0–6) ve aralarına çekilebilecek kabloların kilometre cinsinden uzunlukları. Dokuz kenar vardır.

Şekil 17.6'daki çizgenin kenarları ağırlıklarıyla şunlardır: (0,1) = 28, (0,5) = 10, (1,2) = 16, (1,6) = 14, (2,3) = 12, (3,4) = 22, (3,6) = 18, (4,5) = 25, (4,6) = 24. Notun bütün Kruskal ve Prim izlemeleri bu çizge üzerindedir. Çizgenin 46 kapsayan ağacı vardır. Dokuz kenardan altısı *C(9, 6) = 84* biçimde seçilir; bunların 38'i döngü kurar. 46 ağacın maliyetleri 99 ile 129 arasında değişir. En ucuz ağacın maliyeti 99'dur ve tektir; ondan sonraki en ucuz iki ağacın maliyeti 101'dir. Bu sayılar bütün seçimlerin bilgisayarla tek tek denenmesiyle bulundu. Amaç, bu denemeyi yapmadan 99'luk ağaca doğrudan ulaşmaktır.

Hoca burada önemli bir noktanın altını çizer: minimum kapsayan ağaç ararken döngüden kaçınılır. Bir köye iki yoldan elektrik gelmesi, örneğin "Aşağı Tepecik'ten gelmezse Yukarı Tepecik'ten gelsin" diye yedek hat çekilmesi bu problemin konusu değildir. Döngü, yedek bağlantı demektir ve yedek bağlantı fazladan maliyettir. Güvenilirlik isteyen ağlar başka problemlerle tasarlanır.

17.2.2 Açgözlü yaklaşım

Minimum kapsayan ağaç için birçok algoritma bilinir: Kruskal, Prim ve Sollin (Borůvka) algoritmaları en tanınmışlarıdır; ters silme (*reverse-delete*) adlı bir yöntem de vardır. Bu notun ağırlığı ilk ikisindedir, öteki ikisi kısaca anlatılır.

Algoritma tasarımında birkaç genel yöntem vardır. **Geri izleme** (*backtracking*) kısmi bir çözümü adım adım genişletir, çıkmaza girince son kararı geri alır. Dinamik programlama küçük alt problemlerin çözümlerini saklayıp büyüklerini onlardan kurar. **Açgözlü yöntem** (*greedy method*) ise her adımda o an en iyi görünen seçimi yapar ve bu seçimi bir daha değiştirmez. Hoca bu tutuma "ısrarcı" der: açgözlü algoritma geri dönmez, bir kez aldığı kenarı bırakmaz.

Minimum kapsayan ağaç algoritmaları açgözlüdür ve üç kurala uyar:

1. Döngü oluşturan kenar alınmaz.
2. Seçilebilecek kenarlar arasından en az maliyetli olan tercih edilir.
3. *n* düğümlü çizgede tam *n − 1* kenar seçilir.

Birinci kural sonucun ağaç olmasını, üçüncüsü bütün düğümleri kapsamasını sağlar; ikisi birlikte Teorem 17.2'nin (ii) ve (iii) koşullarıdır. İkinci kural maliyeti küçük tutar. Algoritmalar arasındaki fark, "seçilebilecek kenarlar" kümesini nasıl tanımladıklarındadır. Kruskal'da bu küme henüz denenmemiş bütün kenarlardır. Prim'de yalnız büyüyen ağaca dokunan kenarlardır.

Bölümün sunumu Michelangelo'ya atfedilen bir sözle açılır: "Güzellik, fazlalıktan arınmışlıktır." Minimum kapsayan ağaç tam olarak budur: bağlantıyı koruyan, fazlası atılmış en yalın ağ.

17.2.3 Açgözlü seçim neden doğru sonuç verir?

Açgözlü algoritmalar çoğu problemde en iyi sonucu bulamaz. Bozuk para üstü vermek için her seferinde en büyük madeni parayı seçmek bazı para sistemlerinde fazla sayıda para kullandırır. O hâlde minimum kapsayan ağaçta açgözlü seçimin neden pişman olmadığı gösterilmelidir. Bunun anahtarı kesit özelliğidir.

Tanım 17.4

Bir çizgenin düğümlerini boş olmayan iki gruba, S ve $V - S$'ye ayırmaya **kesit** (*cut*) denir. Bir ucu S'de, öteki ucu $V - S$'de olan kenar **kesiti geçer**.

Teorem 17.3 (Kesit özelliği)

Bağlantılı ve ağırlıklı bir çizgede herhangi bir kesit alınsın. Kesiti geçen kenarlar arasında ağırlığı en küçük olan kenar (u, v) ise (u, v)'yi içeren bir minimum kapsayan ağaç vardır. Ağırlıklar farklıysa (u, v) her minimum kapsayan ağaçtadır.

İspat. T bir minimum kapsayan ağaç olsun ve (u, v)'yi içermesin. (u, v)'yi T'ye ekleyelim. Teorem 17.2'ye göre tam olarak bir döngü oluşur. Döngü u'dan (S tarafı) çıkıp v'ye (öteki taraf) varır ve yeniden u'ya döner. Öteki taraftan S'ye dönebilmek için kesiti en az bir kez daha geçmelidir; bu geçiş bir ağaç kenarıyla, diyelim (x, y) ile olur. (u, v) kesitin en hafif kenarı olduğundan $w(u, v) \le w(x, y)$'dir. (x, y)'yi silelim: döngü kırılır ve yine bir kapsayan ağaç elde edilir. Yeni ağacın maliyeti T'ninkinden büyük değildir. T minimum olduğundan maliyet eşittir; yeni ağaç da minimumdur ve (u, v)'yi içerir. Ağırlıklar farklıysa $w(u, v) < w(x, y)$ olur ve yeni ağaç T'den ucuz çıkar. Bu bir çelişkidir; demek ki T zaten (u, v)'yi içermelidir. ■

Kruskal ve Prim her adımda bir kesitin en hafif kenarını seçer. Prim'de kesit apaçıktır: S o ana kadar kurulan ağacın düğümleridir ve seçilen kenar, ağaçtan dışarı çıkan en ucuz kenardır. Kruskal'da seçilen kenarın uçları iki ayrı öbekte durur. Bir ucun öbeği S alınırsa, kenar bu kesiti geçen kenarların en ucuzudur. Daha ucuz bir geçiş kenarı olsaydı Kruskal ona daha önce bakmış ve iki öbek zaten birleşmiş olurdu. Böylece iki algoritma da her adımda bir minimum kapsayan ağacın kenarını seçer ve hiçbir seçim sonradan geri alınmak zorunda kalmaz.

Kesit özelliğinin bir ikizi daha vardır: **döngü özelliği**. Bir döngüdeki en ağır kenar (ağırlıklar farklıysa) hiçbir minimum kapsayan ağaçta bulunmaz. Ters silme yöntemi bu özelliğe dayanır.

Sık yapılan hata

"Her düğüm kendisine değen en ucuz kenarı seçsin, olur biter" diye düşünmek. Bu kural kesit özelliğinin özel bir durumudur (tek düğüm ile geri kalanı arasındaki kesit) ve seçilen kenarlar gerçekten MST'ye aittir. Ama yeterli değildir. Yedi köy örneğinde 0 ve 5 için (0,5), 1 ve 6 için (1,6), 2 ve 3 için (2,3), 4 için (3,4) seçilir. Ortaya üç ayrı öbek çıkar, ağaç tamamlanmaz. Sollin algoritması bu kuralı öbekler üzerinde turlar hâlinde tekrarlayarak işi bitirir.

17.3 Kruskal algoritması

17.3.1 Algoritma

Joseph B. Kruskal algoritmayı 1956'da yayımladı. Fikri bir cümledir: kenarları ucuzdan pahalıya sırayla dene; döngü kurmayanı al, kuranı bırak. Algoritmanın doğruluğunun kanıtı vardır ve kesit özelliğine dayanır. Kanıtı okumak algoritmanın neden böyle kurulduğunu daha iyi anlatır. Sözde kodu şöyledir:

```
T = {};                    /* seçilen kenarlar */
while (|T| < n-1 && E ≠ ∅) {
    E'den en az maliyetli (v,w) kenarını seç;
    E = E - {(v,w)};
    if ((v,w) T içinde döngü oluşturmuyorsa)
        T = T ∪ {(v,w)};
    else
        (v,w)'yi bırak;
}
if (|T| < n-1)
    printf("minimum kapsayan ağaç yok\n");
```

Kodda T seçilen kenarların, E henüz denenmemiş kenarların kümesidir; $|T|$ seçilen kenar sayısını gösterir.

- **Satır 1.** Başta hiçbir kenar seçilmemiştir. Her düğüm, tek düğümlü kendi öbeğidir.
- **Satır 2.** Döngü iki koşuldan biri bozulunca durur. $n - 1$ kenar seçildiyse ağaç tamamlanmıştır; daha fazla kenara bakmaya gerek yoktur. Seçilecek kenar kalmadıysa da durulur.
- **Satır 3–4.** Kalan kenarlardan en ucuzu alınır ve kümeden çıkarılır. Bir kenar en çok bir kez denenir.
- **Satır 5–8.** Kenar, o ana kadar seçilmiş kenarlarla bir döngü kurmuyorsa ağaca eklenir; kuruyorsa bırakılır. Bırakılan kenar bir daha ele alınmaz. Seçilenlerle döngü kuran bir kenar, sonra ne seçilirse seçilsin yine döngü kurar.
- **Satır 10–11.** Döngü bittiğinde $n - 1$'den az kenar seçilmişse kenarlar tükenmiş ama ağaç tamamlanamamıştır. Bu, çizgenin bağlantılı olmadığını gösterir ve minimum kapsayan ağaç yoktur.

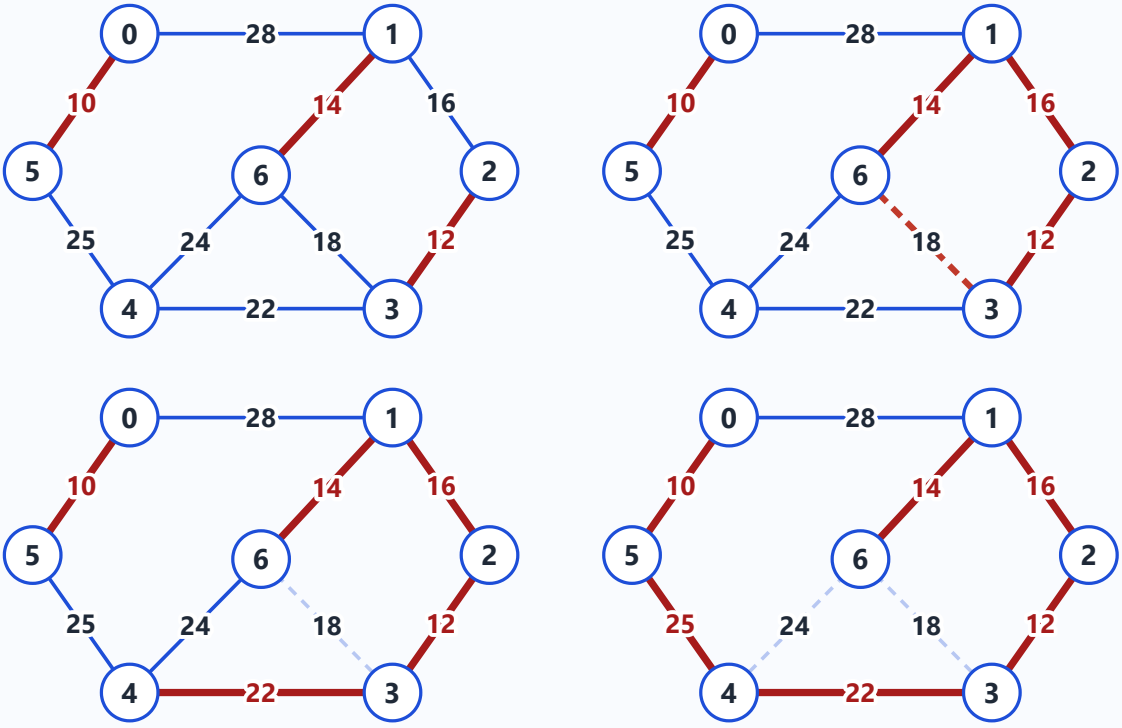
Kruskal'ın ara durumları ağaç değil ormandır. Seçilen kenarlar birbirinden ayrık birkaç **öbek** (bağlantılı bileşen) oluşturabilir. Öbekler birleşe birleşe büyür ve en sonda tek öbek kalır. Bir kenarın döngü kurması, iki ucunun zaten aynı öbekte olması demektir.

17.3.2 Kruskal adım adım

Yedi köy çizgesinin dokuz kenarı maliyete göre sıralanınca şu dizi çıkar: (0,5) 10, (2,3) 12, (1,6) 14, (1,2) 16, (3,6) 18, (3,4) 22, (4,6) 24, (4,5) 25, (0,1) 28. Tablo 17.2 her kenarın kararını, Şekil 17.7 de seçilmiş dört anı gösterir.

Adım	Kenar	w	Uçların öbekleri	Karar	Öbekler	Toplam
1	(0,5)	10	{0} ve {5}	ağaca ekle	{0,5}	10
2	(2,3)	12	{2} ve {3}	ağaca ekle	{0,5} {2,3}	22
3	(1,6)	14	{1} ve {6}	ağaca ekle	{0,5} {2,3} {1,6}	36
4	(1,2)	16	{1,6} ve {2,3}	ağaca ekle	{0,5} {1,2,3,6}	52
5	(3,6)	18	ikisi de {1,2,3,6}	döngü, bırak	{0,5} {1,2,3,6}	52
6	(3,4)	22	{1,2,3,6} ve {4}	ağaca ekle	{0,5} {1,2,3,4,6}	74
7	(4,6)	24	ikisi de {1,2,3,4,6}	döngü, bırak	{0,5} {1,2,3,4,6}	74
8	(4,5)	25	{1,2,3,4,6} ve {0,5}	ağaca ekle	{0,1,2,3,4,5,6}	99
—	(0,1)	28	hiç bakılmaz: 6 kenar seçilmiştir			

Tablo 17.2: Yedi köy çizgesinde Kruskal. “Öbekler” sütunu karardan sonraki durumdur; tek düğümlü öbekler yazılmamıştır.



Şekil 17.7: Kruskal'ın dört anı. Sol üstte üçüncü adımdan sonra üç ayrı öbek var: {0,5}, {2,3}, {1,6}. Sağ üstte (3,6) döngü kurduğu için kırmızı kesikli çizilmiş ve bırakılmıştır. Sol altta (3,4) eklenmiş, toplam 74'e çıkmıştır. Sağ altta ağaç tamamdır: altı kenar, toplam 99.

Tablonun dört satırı dikkat ister. Dördüncü adımda (1,2) iki ayrı öbeği, {1,6} ile {2,3}'ü birleştirir. Bu adımdan önce öbek sayısı (tek düğümlüler dahil) 4'tü, sonra 3'e düşer. Beşinci adımda (3,6)'nın iki ucu da aynı öbektedir. 3'ten 6'ya ağaç üzerinde 3-2-1-6 yolu zaten vardır; (3,6) bu yolu kapatıp döngü kurar. Yedinci adımda (4,6) için de aynı durum geçerlidir: 4-3-2-1-6 yolu hazırdır. Sekizinci adımda altıncı kenar seçilir ve algoritma durur. En pahalı kenar (0,1)'e hiç bakılmaz.

Bulunan ağaç {(0,5), (2,3), (1,6), (1,2), (3,4), (4,5)} ve maliyeti *10 + 12 + 14 + 16 + 22 + 25 = 99*'dur. Bu, 46 kapsayan ağacın en ucuzudur.

17.3.3 Kruskal nasıl kodlanmaz?

Sözde kodun iki satırı belirsiz kalır: “en az maliyetli kenarı seç” ve “döngü oluşturmuyorsa”. Bu iki işin nasıl yapılacağı kodun hızını belirler. Hoca derste önce bilerek yanlış bir yoldan gider ve sonra doğrusunu gösterir. Yanlış yolu görmek doğrusunu anlamayı kolaylaştırır.

Tablodaki öbek sayılarına bakan biri bir gözlem yapar: öbekler önce ikiye, sonra üçe çıktı, sonra yeniden ikiye düştü. “Yakaladım!” der, “seçtiğim kenarların öbeklerini birer listede tutarım.” Seçilmemiş kenarlar da bir dizidedir. Her seferinde dizide en ucuz kenarı aramak, onu silmek ve kalanları kaydırmak kötü bir fikirdir. Kenarları bir kez maliyete göre sıralayıp (uç1, uç2, maliyet) üçlüleri hâlinde tutmak daha iyidir.

Bu planla izleme şöyle gider. (0,5) gelir: listelerde aranır; ne 0 vardır ne 5, yeni bir liste açılır. (2,3) ve (1,6) için de yeni listeler açılır. (1,2) gelince 1 bir listede, 2 başka bir listede bulunur. Farklı listelerde oldukları için döngü yoktur; kenar alınır ve iki liste birleştirilir. (3,6) gelince ikisi de aynı listededir: döngü!

Bu kod doğru çalışır. Ama her kenar için iki ucu bütün listelerde aranır ve bu arama *O(n)* sürer. Toplam maliyet *O(e · n)* olur. Hoca bu noktada açık konuşur: döngü yakalanmıştır, sonuç da doğrudur, ama böyle kodlanmaz ve sınavda böyle yazan kod puan almaz. Algoritmanın kendisi değil, arkasındaki veri yapısı yanlıştır. Hoca ayrıca Kruskal'ın kodlanmasını ödev olarak verebileceğini ve öğrencilerin gereken veri yapılarını kendi başlarına keşfedip edemeyeceğini merak ettiğini söyler.

> **Sık yapılan hata**
>
> Döngüyü “iki uç da daha önce seçilmiş mi?” sorusuyla denetlemek. Yedi köy örneğinin dördüncü adımında (1,2)'nin iki ucu da daha önce seçilmiş kenarlardadır: 1, (1,6)'da; 2, (2,3)'te. Bu kurala göre kenar reddedilir. Oysa uçlar farklı öbeklerdedir ve (1,2)

MST'nin bir kenarıdır. Doğru soru "iki uç *aynı öbekte* mi?" sorusudur.

17.3.4 İki soru, iki veri yapısı

Kruskal'ın her adımında iki soru sorulur ve her birinin karşılığı olan bir veri yapısı vardır.

1) En az maliyetli kenar hangisi? Bu soru, en küçük öğeyi tekrar tekrar isteyen bir yapıya işaret eder: **min-heap** ile gerçeklenen **öncelikli kuyruk** (*priority queue*). Min-heap'te her düğüm çocuklarından küçük ya da onlara eşittir; en küçük öğe köktedir. Kökü almak (*deleteMin*) *O(log e)* sürer. Heap iki yolla kurulur. Kenarlar tek tek eklenirse *O(e log e)* gerekir. Kenar dizisi hazırsa **aşağıdan yukarıya heap oluşturma** (*bottom-up heap construction*) ile *O(e)* yeter. Bu yöntemde son iç düğümden köke doğru her düğüm aşağı süzülür ve dizi heap'e dönüşür. Üçüncü yol kenarları bir kez sıralamaktır; C'de bunun için standart kütüphanenin `qsort` fonksiyonu kullanılır.

2) Bu kenar döngü kurar mı? Kenarın iki ucu farklı öbeklerdeyse döngü kurmaz. Öbekler birbirinden **ayrık kümelerdir** (*disjoint sets*): her düğüm tam olarak bir kümededir. Ayrık kümeler üzerinde iki işlem gerekir. **Küme bulma** (*find*) bir düğümün hangi kümede olduğunu kümenin kökünü döndürerek söyler. **Birleşim** (*union*) iki kümeyi tek kümede toplar. Bu ikiliye kısaca **union-find** denir. Döngü sorusu böylece bir karşılaştırmaya iner: `kume_bul(3)` ile `kume_bul(2)` aynı kökü mü veriyor?

Hoca bu noktada şaşkınlığı paylaşır: arka planda heap ve kümeler çoktan anlatılmış, ama öğrenci bunların Kruskal'da işe yarayacağını fark etmeden gidip liste tutmaya kalkmıştır. İlkokulda öğretilen kümeler burada bir çizge algoritmasını hızlandırır.

Sunum maliyeti ayrıntılı bir formülle verir. Heap kenarlar tek tek eklenerek kurulursa yaklaşık $|E| \cdot \lg |E|$ iş gerekir. Ardından $|V|$ kez en küçük kenar alınır; her alma $\lg |E|$, her küme denetimi neredeyse sabit sürer. Toplam $|E| \cdot \lg |E| + |V| \cdot (\lg |E| + \Theta(1))$ olur. Heap aşağıdan yukarıya kurulursa ilk terim $2|E|$'ye iner. Buradaki "neredeyse sabit" süre, union-find'ın ters Ackermann fonksiyonu α ile sınırlanan maliyetidir. Formül bir noktada iyimserdir: döngü kuran kenarlar da heap'ten alınır. En kötü durumda $n - 1$ değil e kez en küçük alınabilir. Güvenli üst sınır *O(e log e)*'dir.

17.3.5 Union-find ve karşılaştırma fonksiyonu

Kruskal'ın C gerçeklemesi üç küçük parçaya dayanır: bir kenar yapısı, union-find işlemleri ve `qsort` için bir karşılaştırma fonksiyonu.

```c
typedef struct { int u, v, w; } kenar;

int kume_bul(int ata[], int i)   /* kok: ata < 0 */
{
    while (ata[i] >= 0) i = ata[i];
    return i;
}
void birlestir(int ata[], int a, int b)   /* a, b kok */
{
    if (ata[a] <= ata[b]) {   /* a daha buyuk kume */
        ata[a] += ata[b];  ata[b] = a;
    } else {
        ata[b] += ata[a];  ata[a] = b;
    }
}
int kars(const void *x, const void *y)
{
    return ((const kenar *)x)->w - ((const kenar *)y)->w;
}
```

- **Satır 1.** Her kenar üç tam sayıdır: uçlar `u`, `v` ve ağırlık `w`. `typedef` sayesinde tip adı `struct` yazmadan `kenar` olarak kullanılır. x86-64 üzerinde `sizeof(kenar)` 12 bayttır; dolgu yoktur, çünkü üç alan da 4 baytlık `int`'tir.
- **Satır 3–7.** `kume_bul`, bir düğümün kümesinin kökünü bulur. Kümeler bir dizide ağaç olarak tutulur: `ata[i]`, `i`'nin ağaçtaki ebeveynidir. Kökün ebeveyni yoktur; kökte `ata` negatif bir sayıdır. Döngü, negatif bir değere varana kadar ebeveynlere tırmanır. Döndürülen kök, kümenin adıdır.
- **Satır 8–15.** `birlestir`, iki kökü alıp kümelerini birleştirir. Parametreler kök olmak zorundadır; kök olmayan düğümle çağrılırsa ağaç bozulur. Kökte saklanan negatif sayının mutlak değeri küme boyutudur: `ata[a] = -4`, `a`'nın kümesinde 4 öğe olduğunu söyler. `ata[a] <= ata[b]` koşulu, `a`'nın kümesinin daha büyük ya da eşit olduğu anlamına gelir ($-4 < -2$). Büyük küme kök olarak kalır: boyutlar toplanır (`ata[a] += ata[b]`) ve küçük kümenin kökü büyüğün altına bağlanır (`ata[b] = a`). `else` kolu aynı işi ters yönde yapar.

- **Satır 16–19.** `kars`, `qsort`'un istediği imzadadır: iki `const void *` alır, birincisi küçükse negatif, büyükse pozitif, eşitse sıfır döndürür. `void *` işaretçiler `const kenar *` tipine dönüştürülür ve ağırlıklar çıkarılır.

Kümelerin bellekteki görünümü tek bir tam sayı dizisidir. Başta her düğüm kendi kümesinin köküdür ve boyutu 1'dir; dizi `{-1, -1, -1, -1, -1, -1, -1}` olur. Yedi köy örneğinde ilk dört kenar eklenince dizi `{-2, -4, 1, 2, -1, 0, 1}` hâline gelir. Bu dizi üç ağaç tanımlar. 0 köktür (boyut 2) ve 5'in ebeveynidir. 1 köktür (boyut 4); 2 ve 6 doğrudan 1'e, 3 ise 2'ye bağlıdır. 4 tek başınadır. `kume_bul(ata, 3)` çağrısı 3 → 2 → 1 yolunu izler ve 1'i döndürür.

> **Teorem 17.4 (Ağırlıklı kural)**
>
> Birleşimde her zaman küçük küme büyüğün altına bağlanırsa (**ağırlıklı kural**, *weighting rule*), m öğeli bir kümenin ağacının yüksekliği en çok $\lfloor \lg m \rfloor + 1$'dir. Bu yüzden `kume_bul` $O(\log n)$ sürer.

İspat. Bir düğümün derinliği yalnız kendi kümesi daha büyük ya da eşit bir kümenin altına bağlandığında bir artar. O anda düğümün içinde bulunduğu kümenin boyutu en az iki katına çıkar. Boyut 1'den başlar ve m'yi aşamaz; iki katına çıkma en çok $\lfloor \lg m \rfloor$ kez olabilir. Kökün derinliği 1 sayılırsa yükseklik en çok $\lfloor \lg m \rfloor + 1$'dir. ∎

Kural olmasaydı en kötü durumda ağaçlar zincire dönerdi. `birlestir(0, 1)`, `birlestir(0, 2)`, … biçiminde her seferinde büyük küme küçüğün altına bağlansa `kume_bul` $O(n)$ adım sürerdi.

Bir iyileştirme daha mümkündür: **yol kısaltma** (*path compression*). Horowitz, Sahni ve Anderson-Freed'in kitabında aynı fikir azaltma kuralı (*collapsing rule*) adıyla geçer. `kume_bul` kökü bulduktan sonra, tırmandığı yol üzerindeki her düğümü doğrudan köke bağlar. Sonraki aramalar aynı yolu bir adımda geçer:

```
int kume_bul(int ata[], int i)       /* yol kisaltmali */
{
    int kok = i, sonraki;
    while (ata[kok] >= 0) kok = ata[kok];     /* 1. tur: koku bul */
    while (i != kok) {                        /* 2. tur: yoldakileri koke bagla */
        sonraki = ata[i];
        ata[i] = kok;
        i = sonraki;
    }
    return kok;
}
```

Birinci döngü kökü bulur, ikinci döngü yolu bir kez daha yürür. Her düğümün eski ebeveyni `sonraki`'de saklanır, sonra düğüm köke bağlanır. Saklama yapılmazsa yolun devamı kaybolur. Ağırlıklı kural ile yol kısaltma birlikte kullanılınca bir dizi işlemde işlem başına amortize maliyet $O(\alpha(n))$ olur. α, ters Ackermann fonksiyonudur ve evrendeki atom sayısı kadar büyük n'ler için bile 4'ü aşmaz; pratikte sabittir. Rastgele 2000 düğümlü bir çizgede (8000 kenar) Kruskal'ın `kume_bul` çağrılarının tırmandığı bağ sayısı yol kısaltma olmadan 21.464, yol kısaltmayla 12.348 olarak ölçüldü.

> **Sık yapılan hata**
>
> `birlestir(ata, k[i].u, k[i].v)` yazmak, yani kenarın uçlarını doğrudan birleştirmek. `birlestir` iki kök bekler. Uç kök değilse onun `ata` değeri bir ebeveyni gösterir; fonksiyon bu değeri küme boyutu sanar ve dizi bozulur. Önce `kume_bul` ile kökler bulunur, birleşim kökler arasında yapılır.

`kars` fonksiyonundaki çıkarma da bir tuzak taşır. İki büyük ağırlığın farkı `int` sınırını aşabilir; örneğin $2.000.000.000 - (-2.000.000.000)$ taşar ve işaret yanlış çıkar. Burada ağırlıklar küçük, negatif olmayan tam sayılardır ve taşma olmaz. Genel amaçlı bir karşılaştırmada `(a > b) - (a < b)` biçimi güvenlidir.

17.3.6 `kruskal()` ana döngüsü ve izlemesi

Parçalar hazır olunca ana döngü on bir satıra sığar. Kodda `k`, `e` öğeli kenar dizisi; `T`, seçilen kenarların yazılacağı $n - 1$ öğeli dizi; `ata`, `n` öğeli küme dizisidir. `i` ve `t` tam sayıdır.

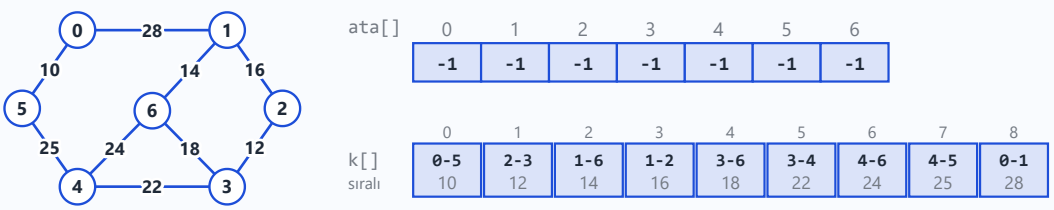
```
for (i = 0; i < n; i++) ata[i] = -1;
qsort(k, e, sizeof(kenar), kars);
for (i = 0, t = 0; i < e && t < n - 1; i++) {
    int a = kume_bul(ata, k[i].u);
    int b = kume_bul(ata, k[i].v);
    if (a != b) {                /* dongu yok */
        T[t++] = k[i];
        birlestir(ata, a, b);
    }                            /* yoksa birak */
}
if (t < n - 1) printf("agac yok\n");
```

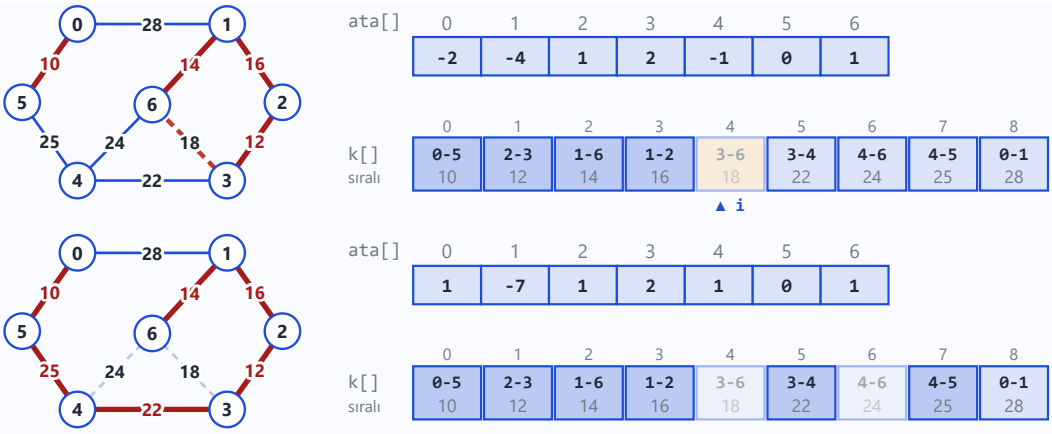
- **Satır 1.** Her düğüm kendi kümesinin kökü yapılır; boyutu 1 olduğu için `ata` değeri −1'dir. Maliyet $O(n)$.
- **Satır 2.** Kenarlar maliyete göre artan sırada dizilir. `qsort`'a dizinin başı, öğe sayısı, bir öğenin bayt boyutu ve karşılaştırma fonksiyonu verilir. `qsort` öğelerin tipini bilmez; `sizeof(kenar)` ile her öğenin kaç bayt olduğunu öğrenir ve öğeleri bayt bayt yer değiştirir. Maliyet $O(e \log e)$'dir.
- **Satır 3.** Dizinin başından itibaren kenarlar denenir. İki sayaç birlikte başlatılır: `i` denenen kenarın indisi, `t` seçilen kenar sayısıdır. Döngü kenarlar bitince ya da $n - 1$ kenar seçilince durur. İkinci koşul unutulursa algoritma yine doğru sonuç verir, ama ağaç tamamlandıktan sonra kalan bütün kenarlar boşuna denenir.
- **Satır 4–5.** Kenarın iki ucunun kökleri bulunur. `a` ve `b` döngü gövdesinde bildirilmiştir; C99'dan beri blok içinde değişken bildirmek serbesttir.
- **Satır 6–9.** Kökler farklıysa uçlar ayrı kümelerdedir ve kenar döngü kurmaz. Kenar `T`'ye kopyalanır (yapı ataması üç alanı birden kopyalar), `t` bir artar ve iki küme birleştirilir. Kökler aynıysa hiçbir şey yapılmaz; kenar bırakılır.
- **Satır 11.** Döngü $t < n - 1$ iken bittiyse kenarlar tükenmiştir; çizge bağlantılı değildir.

Kod, sunumdaki biçimiyle derlenip bir test sürücüsüyle çalıştırıldı. Tablo 17.3 programın kendi çıktısıdır: her kenar için bulunan kökler ve birleşimden sonraki `ata` dizisi.

`i`	`k[i]`	`a`	`b`	**Karar**	`t`	`ata[]`
—	—	—	—	başlangıç	0	`-1 -1 -1 -1 -1 -1 -1`
0	(0,5,10)	0	5	ekle	1	`-2 -1 -1 -1 -1 0 -1`
1	(2,3,12)	2	3	ekle	2	`-2 -1 -2 2 -1 0 -1`
2	(1,6,14)	1	6	ekle	3	`-2 -2 -2 2 -1 0 1`
3	(1,2,16)	1	2	ekle	4	`-2 -4 1 2 -1 0 1`
4	(3,6,18)	1	1	bırak	4	değişmez
5	(3,4,22)	1	4	ekle	5	`-2 -5 1 2 1 0 1`
6	(4,6,24)	1	1	bırak	5	değişmez
7	(4,5,25)	1	0	ekle	6	`1 -7 1 2 1 0 1`
8	—	koşul `t < n - 1` yanlış: döngü biter; maliyet 99				

Tablo 17.3: `kruskal()` ana döngüsünün yedi köy çizgesindeki izlemesi. `a`, `b`: satır 4–5'te bulunan kökler. Son sütun kararından sonraki `ata[0..6]`'dır.





Şekil 17.8: `kruskal()` izlemesinden üç kare. Üstte başlangıç: her `ata` −1, kenarlar sıralı. Ortada `i` = 4: (3,6)'nın iki ucunun kökü de 1'dir, kenar döngü kurar ve bırakılır. Altta son durum: 1 kökü yedi düğümlük tek kümeyi tutar (`ata[1]` = −7); soluk kutular bırakılan kenarlardır.

İzlemedeki birkaç ayrıntı ağırlıklı kuralın işleyişini gösterir. `i` = 0'da iki küme de 1 öğelidir; `ata[0] <= ata[5]` eşitlikle doğru olduğundan 0 kök kalır. `i` = 3'te {1,6} ile {2,3} yine eşit boyludur; 1 kök kalır, 2 onun altına girer, boyut −4 olur. Bu andan sonra 3'ün kökünü bulmak iki adım sürer: 3 → 2 → 1. `i` = 7'de 1'in kümesi beş, 0'ınki iki öğelidir; küçük küme (0) büyüğün (1) altına bağlanır ve `ata[0]` 1 olur. Son dizide en derin düğümler 3 ve 5'tir; ikisi de kökten iki bağ uzaktadır. Yedi öğeli bir küme için Teorem 17.4'ün verdiği sınır *⌊lg 7⌋ + 1 = 3* düzeydir; ağaç bu sınırın içindedir.

Aynı sürücü iki durumu daha denedi. Ağırlıkları farklı bir sınav çizgesinde maliyet 16 çıktı. Kopuk bir çizgede (iki bileşen, beş düğüm) yalnız üç kenar seçildi ve program “agac yok” yazdı. Bütün denemelerde `malloc` ve `free` çağrılarını sayan bir sarmalayıcı 12 ayırma ve 12 serbest bırakma saydı; sızıntı yoktur.

17.3.7 Kruskal'ın karmaşıklığı

Kruskal'ın maliyeti, kodun her satırının kaç kez çalıştığından ve her çalışmanın ne sürdüğünden toplanır. Tablo 17.4 bu toplamı satır gruplarına göre verir.

Adım	Maliyet	Açıklama
Sıralama ya da heap kurma	*O(e log e)* ya da *O(e)*	`qsort` ya da aşağıdan yukarıya heap oluşturma
En küçük kenarı almak	*O(log e)*, en çok *e* kez	min-heap'te kökü silmek; sıralı dizide *O(1)*
`kume_bul` (kenar başına 2 kez)	*O(log n)* ağırlıklı kuralla; ≈ *O(α(n))* yol kısaltmayla	en çok *2e* çağrı
`birlestir`	*O(1)*	tam *n − 1* kez
Toplam	*O(e log e) = O(e log n)*	$e \le n^2$ olduğundan *log e ≤ 2 log n*

Tablo 17.4: Union-find ile Kruskal'ın maliyeti.

Toplamda baskın terim sıralamadır. `kume_bul` çağrıları en çok *2e · O(log n)* tutar ve bu da *O(e log n)*'dir. Son satırdaki eşitlik basit bir gözleme dayanır. Basit bir çizgede $e \le n(n-1)/2 < n^2$'dir. Logaritma alınınca *log e < 2 log n* çıkar; sabit çarpan O gösteriminde düşer ve *O(e log e) = O(e log n)* olur. Bağlantılı çizgede *e ≥ n − 1* olduğundan ters yönde de *log n = O(log e)* geçerlidir. Bellek maliyeti kenar dizisi, `T` ve `ata` için *O(n + e)*'dir.

En iyi durumda ilk *n − 1* kenarın hiçbiri döngü kurmaz ve döngü *n − 1* turda biter. Ama sıralama yine bütün kenarları işler ve karşılaştırmaya dayalı bir sıralama genel olarak *O(e log e)* sürer; ağacın erken tamamlanması bu maliyeti azaltmaz. Heap kullanılırsa tablo değişir. Heap *O(e)*'de kurulur, sonra yalnız gereken kadar en küçük alınır. Ağaç erken tamamlanırsa toplam *O(e + k log e)* olur; *k* denenen kenar sayısıdır. Ödevde heap'in istenmesinin nedeni budur. Rastgele ağırlıklı yoğun bir çizgede (2000 düğüm, yaklaşık iki milyon kenar) ağaç yalnız 9685 kenar denendikten sonra tamamlandı; `qsort` ise iki milyon kenarın hepsini sıraladı.

Liste tutan “kötü” kodlama ile karşılaştırma notun ana fikrini özetler. Kötü kodlamada her kenar için uçlar bütün listelerde aranır: kenar başına *O(n)*, toplam *O(e · n)*. Yedi köy için fark önemsizdir. 5000 düğümlü ve altı milyon kenarlı bir çizgede ise *e · n* otuz milyarı aşar, *e log e* ise yaklaşık yüz kırk milyondur. Algoritma aynıdır; değişen yalnız arkasındaki veri yapısıdır.

17.4 Prim algoritması

17.4.1 Algoritma

Prim algoritmasının tarihi ilginçtir. Çek matematikçi Vojtěch Jarník algoritmayı 1930'da yayımladı. Robert C. Prim 1957'de, Edsger W. Dijkstra 1959'da aynı algoritmayı birbirinden habersiz yeniden buldu. Bu yüzden kaynaklarda DJP algoritması ya da Prim-Jarník algoritması adıyla da geçer.

Kruskal'da seçilen kenarlar birbirinden ayrık öbekler oluşturabiliyordu. Prim ise tek bir ağacı büyütür. Bir başlangıç düğümüyle başlar ve her adımda ağaçtaki bir düğümü ağaç dışındaki bir düğüme bağlayan en ucuz kenarı ekler. Sözde kodu şöyledir:

```
T  = {};
TV = {0};           /* ağaç düğümleri: 0'dan başla */
while (|T| < n-1) {
    u ∈ TV ve v ∉ TV olan en ucuz (u,v) kenarını seç;
    if (böyle kenar yok) break;
    TV = TV ∪ {v};
    T  = T  ∪ {(u,v)};
}
if (|T| < n-1)
    printf("minimum kapsayan ağaç yok\n");
```

- **Satır 1–2.** *T* seçilen kenarların, *TV* ağaca katılmış düğümlerin kümesidir. Başlangıç düğümü 0'dır; herhangi bir düğüm seçilebilir.
- **Satır 3–4.** Her turda bir ucu *TV*'de, öteki ucu *TV* dışında olan en ucuz kenar aranır. Seçilebilecek kenarlar yalnız bunlardır.
- **Satır 5.** Böyle bir kenar yoksa ağaç geri kalan düğümlere ulaşamaz; çizge bağlantılı değildir.
- **Satır 6–7.** Kenarın dıştaki ucu *TV*'ye, kenarın kendisi *T*'ye eklenir.
- **Satır 9–10.** Döngü erken bittiyse minimum kapsayan ağaç yoktur.

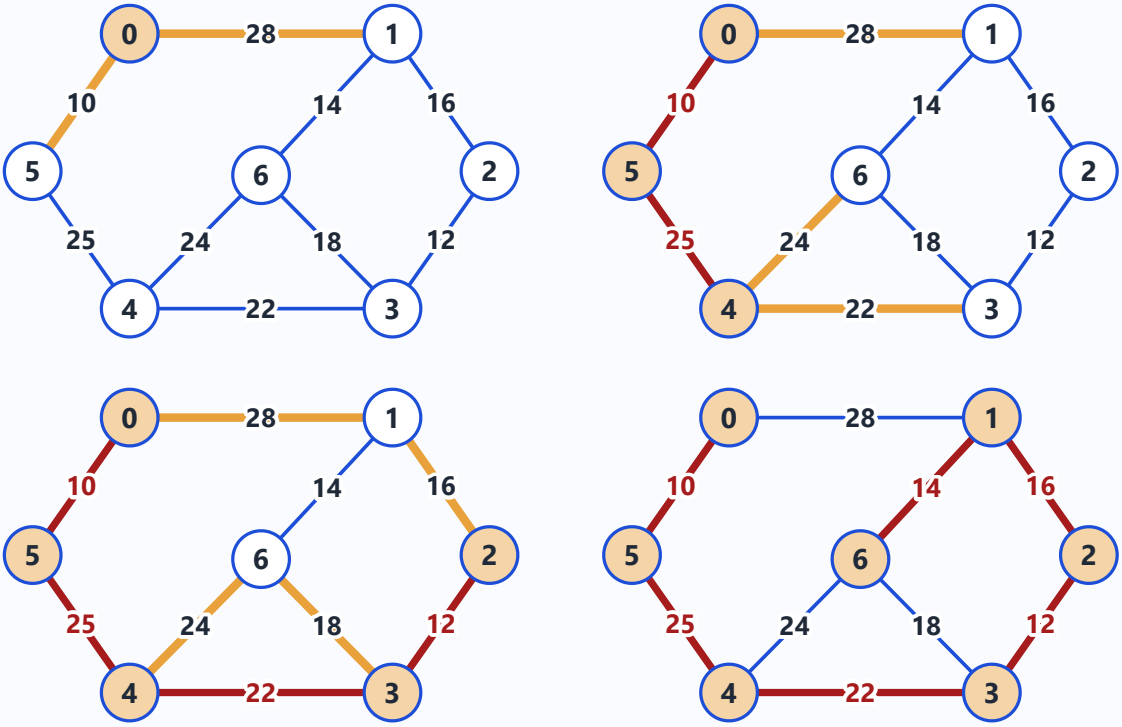
Prim'de döngü denetimi gerekmez. Seçilen kenarın bir ucu ağaçta, öteki ucu ağaç dışındadır; $v \notin TV$ koşulu kenarın yeni bir düğüme gittiğini zaten garanti eder. Yeni düğüme giden bir kenar döngü kuramaz. Seçilen kenar, $S = TV$ kesitini geçen en ucuz kenardır; kesit özelliği her seçimin doğru olduğunu söyler.

17.4.2 Prim adım adım

Yedi köy çizgesinde Prim 0'dan başlatılsın. Tablo 17.5 her turda *TV*'yi, seçilebilecek kenarları ve seçimi verir.

Tur	TV (tur başında)	Seçilebilecek kenarlar	Seçilen	Toplam
1	{0}	(0,1) 28, (0,5) 10	(0,5)	10
2	{0, 5}	(0,1) 28, (5,4) 25	(5,4)	35
3	{0, 5, 4}	(0,1) 28, (4,3) 22, (4,6) 24	(4,3)	57
4	{0, 5, 4, 3}	(0,1) 28, (3,2) 12, (3,6) 18, (4,6) 24	(3,2)	69
5	{0, 5, 4, 3, 2}	(0,1) 28, (2,1) 16, (3,6) 18, (4,6) 24	(2,1)	85
6	{0, 5, 4, 3, 2, 1}	(1,6) 14, (3,6) 18, (4,6) 24	(1,6)	99

Tablo 17.5: Yedi köy çizgesinde 0'dan başlayan Prim. “Seçilebilecek” sütununda bir ucu TV'de, öteki ucu dışarıda olan kenarlar vardır.



Şekil 17.9: Prim'in dört anı. Dolgulu düğümler TV'dedir, sarı kalın kenarlar o turun aday kenarlarıdır, kalın kırmızı kenarlar ağaçtadır. Sol üst: 1. tur. Sağ üst: 3. tur. Sol alt: 5. tur. Sağ alt: ağaç tamam, toplam 99.

Tablonun "seçilebilecek" sütunu her turda değişir. Bir düğüm ağaca girince ona değen kenarlar adaylara katılır. Bir kenarın iki ucu da ağaca girince o kenar adaylıktan düşer. Örneğin (0,1), beş tur boyunca adaydır ama hep daha ucuz bir aday bulunur. Altıncı turun başında 1 de ağaçtadır; (0,1)'in iki ucu *TV*'dedir ve artık aday değildir. Sunum bu noktaya ayrıca dikkat çeker: (0,1) çizimde hâlâ "seçilebilecek" gibi görünür, ama seçilirse döngü kurar.

Prim'in bulduğu ağaç {(0,5), (5,4), (4,3), (3,2), (2,1), (1,6)}'dır ve Kruskal'ınkiyle aynıdır; yalnız kenarların seçilme sırası farklıdır. Bu bir rastlantı değildir. Yedi köy çizgesinde bütün ağırlıklar farklıdır ve minimum kapsayan ağaç tektir. Başlangıç düğümü değiştirilse de sonuç değişmez: 3'ten başlayan Prim kenarları (3,2), (2,1), (1,6), (3,4), (4,5), (5,0) sırasıyla seçer ve yine 99'a varır.

17.4.3 Prim'i kodlamanın asıl zorluğu

Sözde kodun dördüncü satırı tek cümledir ama kodlaması kolay değildir: "bir ucu ağaçta, öteki ucu dışarıda olan en ucuz kenarı seç". Bu satır her turda çalışır ve şu gereksinimleri taşır:

- En az maliyetli aday kenar tek hamlede alınabilmelidir.
- Bir kenar seçilince adaylara yeni kenarlar eklenir ve iki ucu da ağaca düşen kenarlar çıkar. Aday kümesi her turda değişir.
- 0 seçildiğinde yalnız (0,1) ve (0,5)'i gören bir yapı gerekir; çizgenin geri kalan kenarlarına bakmamalıdır bile.

Saf bir kodlama her turda bütün kenarları tarar ve ucu ağaçta olanlar arasında en ucuzunu arar. Bu, tur başına $O(e)$, toplam $O(n \cdot e)$ eder. Kruskal'daki liste hatasının Prim'deki karşılığıdır.

Hoca iki gözlemle konunun önemini vurgular. Birincisi: bu satırın verimli bir çözümünü bulabilen öğrenci, veri yapıları konusunda belli bir düzeye gelmiş demektir. "Prim zaten yazmış, ben de kodlarım" diyerek sözde kodu doğrudan koda çeviren ise karmaşıklığın birden büyüdüğünü görür. İkincisi: "algoritma" kelimesini hiç duymamış bir çizge kuramcısına bu problem verilse o da Prim'in yaptığını yapar; en ucuz kenarı seçer. Bilgisayar mühendisinin farkı, bu seçimin arkasında hangi veri yapısının durması gerektiğini keşfetmesidir.

Çözüm şu gözleme dayanır: aday kenarların hepsini tutmak gerekmez. Ağaç dışındaki her v düğümü için yalnız bir sayı yeter: v'yi ağaca bağlayan en ucuz kenarın maliyeti. Bu sayı `enaz[v]` dizisinde, o kenarın ağaçtaki ucu da `yakin[v]` dizisinde tutulur. En ucuz aday kenar, `enaz` değeri en küçük olan ağaç dışı düğümün kenarıdır. Bir u düğümü ağaca girince yalnız u'nun komşularının `enaz` değerleri değişebilir. Bu fikir dizilerle $O(n^2)$, bir min-heap ile $O(e \log n)$ süreli iki gerçekleme verir.

17.4.4 Dizilerle $O(n^2)$ gerçekleme

Çizge bir maliyet matrisiyle tutulur: `w[u][v]`, *(u, v)* kenarının ağırlığıdır. Kenar yoksa `w[u][v]` büyük bir `SONSUZ` sabitidir; köşegen 0'dır. `secildi[v]`, *v*'nin ağaçta olup olmadığını söyler ($v \in TV$). Prim'in sunumdaki C kodu şudur:

```
secildi[0] = 1;                   /* TV = {0} */
for (v = 0; v < n; v++)
    enaz[v] = w[0][v], yakin[v] = 0;
for (k = 1; k < n; k++) {
    u = en_yakin(enaz, secildi, n);  /* O(n) */
    if (u < 0) break;                /* agac yok */
    secildi[u] = 1;  toplam += enaz[u];
    for (v = 0; v < n; v++)
        if (!secildi[v] && w[u][v] < enaz[v])
            enaz[v] = w[u][v], yakin[v] = u;
}
```

- **Satır 1.** 0 ağaca alınır. Öteki `secildi` öğelerinin ve `toplam`'ın önceden 0 yapılmış olması gerekir; kod parçası bunu varsayar.
- **Satır 2–3.** Başta ağaçta yalnız 0 vardır. Bu yüzden her *v*'nin ağaca en ucuz bağlantısı doğrudan *(0, v)* kenarıdır: `enaz[v] = w[0][v]`, `yakin[v] = 0`. Kenarı olmayan düğümlerde `enaz` `SONSUZ` kalır. Virgül işleci iki atamayı tek deyimde yapar.
- **Satır 4.** Ağaca $n - 1$ düğüm daha eklenecektir; döngü $n - 1$ tur döner.
- **Satır 5–6.** `en_yakin`, ağaç dışındaki düğümler arasında `enaz` değeri en küçük olanı döndürür. Bütün `enaz` değerleri `SONSUZ` ise ağaca bağlanabilecek düğüm kalmamıştır; fonksiyon −1 döndürür ve döngü kırılır.
- **Satır 7.** `u` ağaca alınır, kenarı *(yakin[u], u)*'dur ve maliyeti toplama eklenir.
- **Satır 8–10.** Ağaca yeni giren `u`, dışarıdaki düğümlere daha ucuz bir bağlantı sunabilir. Her ağaç dışı *v* için `w[u][v]`, kayıtlı `enaz[v]`'den küçükse kayıt güncellenir: artık *v*'nin en ucuz bağlantısı `u` üzerindendir.

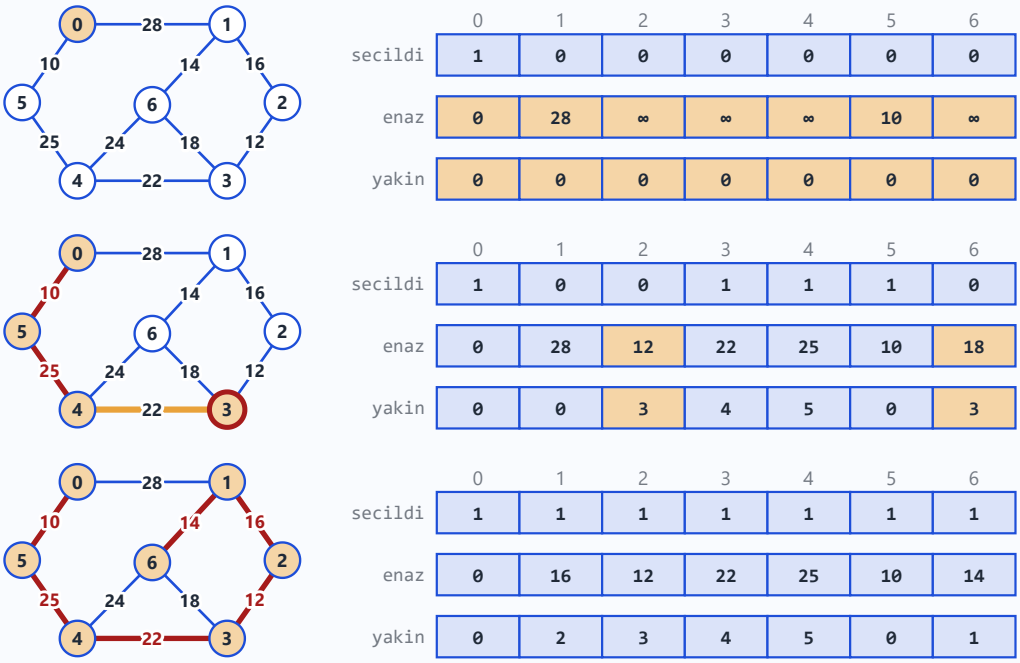
`en_yakin` fonksiyonu sunumda verilmez. Gerçeklemesi basit bir doğrusal aramadır:

```
int en_yakin(const int enaz[], const int secildi[], int n)
{
    int v, u = -1;
    for (v = 0; v < n; v++)
        if (!secildi[v] && enaz[v] < SONSUZ &&
            (u < 0 || enaz[v] < enaz[u]))
            u = v;
    return u;            /* -1: TV'ye bagli dugum kalmadi */
}
```

Arama yalnız ağaç dışındaki ve ağaca bağlanabilen (`enaz` değeri `SONSUZ` olmayan) düğümlere bakar. `u < 0` koşulu ilk uygun düğümü aday yapar; sonraki her düğüm daha küçük bir `enaz` değeriyle adayın yerini alır.

k	u	Ağaca giren kenar	toplam	Güncellenen (`enaz`, `yakin`)
—	—	başlangıç	0	`enaz` = 0 28 ∞ ∞ ∞ 10 ∞; `yakin` hep 0
1	5	(0,5) 10	10	enaz[4] = 25 (5)
2	4	(5,4) 25	35	enaz[3] = 22 (4), enaz[6] = 24 (4)
3	3	(4,3) 22	57	enaz[2] = 12 (3), enaz[6] = 18 (3)
4	2	(3,2) 12	69	enaz[1] = 16 (2)
5	1	(2,1) 16	85	enaz[6] = 14 (1)
6	6	(1,6) 14	99	yok
7	—	döngü biter; `enaz` = 0 16 12 22 25 10 14, `yakin` = 0 2 3 4 5 0 1		

Tablo 17.6: `prim()` izlemesi (yedi köy, 0'dan). "Güncellenen" sütunu satır 8–10'un değiştirdiği kayıtlardır. Son satırdaki diziler döngü bittiğindeki durumdur.



Şekil 17.10: `prim()` izlemesinden üç kare. Üstte başlangıç: yalnız 1 (28) ve 5 (10) 0'a bağlıdır. Ortada 3 ağaca girdikten sonra: enaz[2] = 12, enaz[6] = 18 güncellenmiştir. Altta son durum: `yakin` dizisi ağacın kenarlarını, `enaz` dizisi maliyetlerini taşır.

Son durumdaki iki dizi, ağacın kendisidir. Her $v \neq 0$ için *(yakin[v], v)* bir ağaç kenarı, `enaz[v]` onun maliyetidir: (2,1) 16, (3,2) 12, (4,3) 22, (5,4) 25, (0,5) 10, (1,6) 14. `enaz` 'ın ilk öğesi dışındakilerin toplamı 99'dur. 6'nın kaydı üç kez değişmiştir: önce 24 (4 üzerinden), sonra 18 (3 üzerinden), en son 14 (1 üzerinden). Her güncelleme, ağaca yeni giren bir düğümün 6'ya daha ucuz bir yol açmasıdır.

Maliyet. Dış döngü $n - 1$ tur döner. Her turda `en_yakin` n düğüme bakar, güncelleme döngüsü de n sütunu tarar. Tur başına $O(n)$, toplam $O(n^2)$ eder. Bu süre kenar sayısından bağımsızdır; en iyi ve en kötü durum aynıdır: $\Theta(n^2)$. Bellek matris için $O(n^2)$, diziler için $O(n)$'dir. Yoğun bir çizgede, yani $e \approx n^2$ iken, kenarların hepsine en az bir kez bakmak zaten $\Omega(n^2)$ gerektirir. Dolayısıyla bu sürüm yoğun çizgeler için idealdir.

Kod sunumdaki biçimiyle derlenip çalıştırıldı; Tablo 17.6'daki değerler programın çıktısıdır. Kopuk bir çizgede `en_yakin` üçüncü turda −1 döndürdü, döngü kırıldı ve yalnız iki kenar seçildi.

Sık yapılan hata

Sonsuz için `INT_MAX` kullanıp toplamda taşma yaşamak. `w[u][v] < enaz[v]` karşılaştırması `INT_MAX` ile güvenlidir. Ama ağaca bağlanamayan bir düğüm yanlışlıkla seçilirse `toplam += enaz[u]` taşar. Aynı kod en kısa yol gibi problemlere uyarlanıp `d[u] + w[u][v]` hesaplanırsa toplama doğrudan taşar; işaretli tam sayı taşması C'de tanımsız davranıştır. Güvenli yol, olası bütün toplamlardan büyük ama iki katı `int` 'e sığan bir sabit seçmektir; burada `SONSUZ` = 1.000.000 kullanıldı. Ayrıca `en_yakin` içindeki `enaz[v] < SONSUZ` koşulu bağlanamayan düğümün seçilmesini önler.

Sık yapılan hata

Güncelleme döngüsünde `!secildi[v]` koşulunu unutmak. Ağaçtaki bir düğümün `enaz` ve `yakin` değerleri, ağaca girdiği kenarı kaydeder ve bir daha değişmemelidir. Koşul olmazsa ilk bozulma hemen gelir. `u` ağaca girdiği turda `v = u` için `w[u][u] = 0` kayıtlı değerden küçüktür. Böylece `enaz[u]` 0, `yakin[u]` de `u` olur. Yedi köy çizgesinde bu sürüm denendi. `toplam` , kayıt bozulmadan önce eklendiği için yine 99 çıktı; ama döngü bittiğinde `yakin` dizisi 0 1 2 3 4 5 6 idi. Her düğüm kendine bağlı görünüyordu, ağacın kenarlarından hiçbiri kalmamıştı. Köşegen 0 değil de `SONSUZ` olsaydı bu kez ağaçtaki komşular birbirinin kaydını bozardı. Aday kümesinden iki ucu da ağaçta olan kenarları çıkarmamak bu hatanın genel adıdır.

17.4.5 Heap ile hızlandırmak

Dizili sürüm her turda n düğümü tarar. Seyrek bir çizgede, örneğin $e \approx 4n$ iken, bu taramaların çoğu boşa gider: ağaç dışındaki düğümlerin çoğunun `enaz` değeri hâlâ sonsuzdur. İki iyileştirme birlikte yapılır. Çizge komşuluk listesiyle tutulur; böylece bir düğüm ağaca girince yalnız gerçek komşulara bakılır. En küçük `enaz` değeri de doğrusal aramayla değil, bir min-heap'ten alınır. Sunumdaki kod şöyledir:

```
/* komsuluk listesi + min-heap (anahtar = enaz) */
heap_ekle(H, 0, 0);
while (!heap_bos(H)) {
    u = heap_min_al(H);          /* O(log n) */
    if (secildi[u]) continue;    /* bayat kayit */
    secildi[u] = 1;
    for (p = baslik[u]; p; p = p->bag)
        if (!secildi[p->dugum] &&
            p->agirlik < enaz[p->dugum]) {
            enaz[p->dugum] = p->agirlik;
            yakin[p->dugum] = u;
            heap_ekle(H, p->dugum, p->agirlik);
        }
}
```

Komşuluk listesinde her düğümün bir listesi vardır; `baslik[u]`, `u`'nun listesinin ilk öğesini gösterir. Her öğe üç alanlıdır: komşunun numarası `dugum`, kenarın ağırlığı `agirlik` ve listedeki sonraki öğeyi gösteren `bag`. Heap'in her kaydı bir (düğüm, anahtar) çiftidir ve anahtar, kayıt eklendiği andaki `enaz` değeridir. Kod çalışmadan önce bütün `enaz` değerleri `SONSUZ`, `enaz[0]` 0 ve bütün `secildi` değerleri 0 yapılmış olmalıdır.

- **Satır 2.** Başlangıç düğümü 0, anahtar 0 ile heap'e konur.
- **Satır 3–4.** Heap boşalana kadar anahtarı en küçük kayıt alınır. `heap_min_al` kökü döndürür, son kaydı köke taşıyıp aşağı süzer: $O(\log m)$, m heap'teki kayıt sayısıdır.
- **Satır 5.** Alınan düğüm zaten ağaçtaysa kayıt **bayattır** ve atlanır. Bayat kayıt şöyle oluşur: bir düğümün `enaz` değeri küçülünce eski kayıt heap'ten silinmez, yeni anahtarla ikinci bir kayıt eklenir. Küçük anahtarlı kayıt önce çıkar ve düğüm ağaca girer. Eski kayıt daha sonra çıktığında düğüm işaretlidir.
- **Satır 6.** Düğüm ağaca alınır. Ağaca girdiği kenar *(yakin[u], u)*, maliyeti `enaz[u]`'dur.
- **Satır 7–13.** `u`'nun listesindeki her komşu için dizili sürümdeki güncelleme yapılır. Komşu ağaç dışındaysa ve *(u, komşu)* kenarı kayıtlı bağlantıdan ucuzsa `enaz` ve `yakin` güncellenir ve yeni anahtarla bir heap kaydı eklenir. Liste yürüyüşü, bağlı listelerde alışılmış `for (p = baslik[u]; p; p = p->bag)` kalıbıdır; `p`, `NULL` olunca döngü biter.

Sunum heap işlemlerinin içini vermez. Dizi üzerinde kurulan ikili bir min-heap ile şöyle yazılabilir:

```c
typedef struct { int dugum, anahtar; } kayit;
typedef struct {
    kayit *a;          /* a[1..boyut]: kok a[1] */
    int boyut, kapasite;
} heap;

int heap_bos(const heap *H) { return H->boyut == 0; }

void heap_ekle(heap *H, int dugum, int anahtar)
{
    int i = ++H->boyut;                    /* kapasite >= 2e + 1 */
    while (i > 1 && H->a[i / 2].anahtar > anahtar) {
        H->a[i] = H->a[i / 2];             /* ebeveyni asagi indir */
        i /= 2;
    }
    H->a[i].dugum = dugum;  H->a[i].anahtar = anahtar;
}

int heap_min_al(heap *H)
{
    int kok = H->a[1].dugum;
    kayit son = H->a[H->boyut--];
    int i = 1, c = 2;
    while (c <= H->boyut) {
        if (c < H->boyut && H->a[c + 1].anahtar < H->a[c].anahtar) c++;
        if (son.anahtar <= H->a[c].anahtar) break;
        H->a[i] = H->a[c];                 /* kucuk cocugu yukari al */
        i = c;  c *= 2;
    }
    H->a[i] = son;
    return kok;
}
```

Kayıtlar `a[1..boyut]` aralığındadır; `a[i]`'nin ebeveyni `a[i/2]`, çocukları `a[2i]` ile `a[2i+1]`'dir. `heap_ekle` yeni kaydı dizinin sonuna koyar ve anahtarı ebeveyninkinden küçük olduğu sürece yukarı taşır; yer değiştirmek yerine ebeveyni aşağı indirip boşluğu yukarı kaydırır. `heap_min_al` kökü saklar, son kaydı geçici bir değişkene alır ve onu kökten aşağı doğru, küçük çocuktan büyük kalmayana kadar indirir. Heap dizisi `malloc` ile *2e + 2* kayıtlık ayrılır. Her komşuluk öğesi en çok bir ekleme yapabilir; listelerde toplam *2e* öğe vardır ve başlangıç kaydıyla *2e + 1* kayıt sığmalıdır. `a[0]` kullanılmaz. Dizi iş bitince `free` ile bırakılır; komşuluk listesinin *2e* öğesi de tek tek serbest bırakılmalıdır.

Alınan	Durum	Eklenen kayıtlar	Heap
—	başlangıç	0:0	0:0
0:0	ağaca girer	5:10, 1:28	5:10, 1:28
5:10	ağaca girer	4:25	4:25, 1:28
4:25	ağaca girer	3:22, 6:24	3:22, 6:24, 1:28
3:22	ağaca girer	2:12, 6:18	2:12, 6:18, 6:24, 1:28
2:12	ağaca girer	1:16	1:16, 6:18, 6:24, 1:28
1:16	ağaca girer	6:14	6:14, 6:18, 6:24, 1:28
6:14	ağaca girer	—	6:18, 6:24, 1:28
6:18	bayat, atlanır	—	6:24, 1:28
6:24	bayat, atlanır	—	1:28
1:28	bayat, atlanır	—	boş

Tablo 17.7: Heap'li Prim'in yedi köy çizgesindeki izlemesi. Heap sütunu, satır 7–13 bittikten sonraki kayıtları anahtar sırasıyla “düğüm:anahtar” biçiminde gösterir.

Düğümlerin ağaca giriş sırası dizili sürümle aynıdır: 0, 5, 4, 3, 2, 1, 6. Toplam 10 kayıt eklendi ve 10 kayıt alındı; üçü bayattı. 6 için üç kayıt oluştu (24, 18, 14), çünkü 6'nın `enaz` değeri üç kez küçüldü. Program bu sayıları ve 99 toplamını aynen verdi. Sayaçlı `malloc` / `free` sarmalayıcısı 19 ayırma (18 liste öğesi ve heap dizisi) ile 19 serbest bırakma saydı.

Maliyet. Her komşuluk öğesi en çok bir ekleme yapar; heap'e en çok *2e + 1* kayıt girer ve o kadar kayıt çıkar. Heap'te en çok *2e + 1* kayıt bulunduğundan her işlem *O(log e)* sürer, bu da *O(log n)*'dir. Komşuluk listelerinin yürünmesi toplam *O(n + e)* tutar. Toplam *O((n + e) log n)*, bağlantılı çizgede *O(e log n)* olur. Bellek *O(n + e)*'dir.

Bayat kayıt bırakmak yerine heap'teki kaydın anahtarı yerinde küçültülebilir. Bu işleme **anahtar azaltma** (*decrease-key*) denir. Bunun için her düğümün heap dizisinin neresinde durduğu ayrı bir dizide tutulmalıdır. Heap en çok *n* kayıtlı kalır, ama asimptotik sınır değişmez. Fibonacci heap ise anahtar azaltmayı amortize *O(1)*'de yapar ve sınırı *O(e + n log n)*'ye indirir. Bu sınır kuramsal açıdan önemlidir; pratikte yapının sabitleri büyüktür ve ikili heap çoğu zaman daha hızlıdır.

Gerçekleme	Zaman	Uygun olduğu yer
Prim, diziler (`enaz`, `yakin`)	$O(n^2)$	yoğun çizge ($e \approx n^2$)
Prim, ikili heap + komşuluk listesi	$O(e \log n)$	seyrek çizge
Prim, Fibonacci heap	$O(e + n \log n)$	kuramsal
Kruskal	$O(e \log e)$	seyrek çizge, kenar listesi hazır

Tablo 17.8: Prim gerçeklemeleri ve Kruskal.

Tablodaki sınırların pratikteki karşılığını görmek için üç gerçekleme rastgele ağırlıklı çizgelerde çalıştırıldı. Tablo 17.9 süreleri verir. Süreler makineye, derleyiciye ve rastgele çizgeye göre değişir; burada önemli olan oranlardır.

n	*e*	Çizge	Kruskal (qsort)	Prim, diziler	Prim, heap
2000	8000	seyrek	0,001	0,017	0,002
2000	1.999.000	yoğun	0,42	0,019	0,040
5000	20.000	seyrek	0,004	0,106	0,003
5000	6.248.750	yoğun	1,47	0,114	0,109

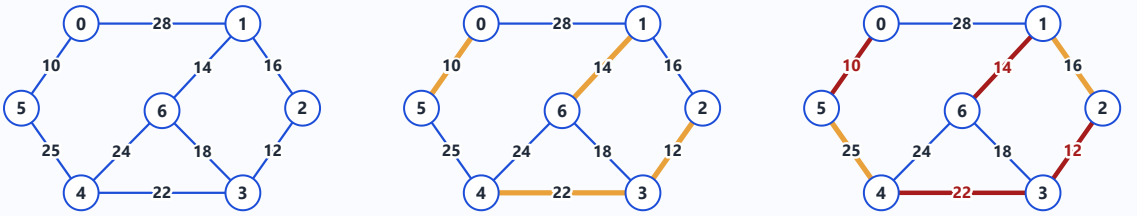
Tablo 17.9: Rastgele bağlantılı çizgelerde ölçülen süreler (saniye; tek ölçüm, zig cc -O2, makineye göre değişir). Dört satırda da üç algoritmanın bulduğu maliyet aynıdır.

Seyrek çizgelerde Kruskal ile heap'li Prim birkaç milisaniyede biter; dizili Prim onlardan 8 ila 35 kat yavaştır. 2000'den 5000'e çıkınca dizili Prim'in süresi yaklaşık altı kat arttı; $(5000/2000)^2 \approx 6{,}3$ beklentisine yakındır. Yoğun çizgelerde tablo tersine döner. Dizili Prim kenar sayısından etkilenmez ve en hızlılar arasındadır. Kruskal'ın süresinin neredeyse tamamı milyonlarca kenarı sıralamaya gider, oysa ağaç ilk on bin kenar içinde tamamlanır. Yoğun çizgede heap'li Prim ancak dizili sürüme yetişir. Bir heap işlemi, matristeki bir hücreye bakmaktan çok daha pahalıdır; yoğun çizgede ise kenar sayısı hücre sayısına yakındır.

17.5 Öteki algoritmalar ve karşılaştırmalar

17.5.1 Sollin (Borůvka) ve ters silme

Çek mühendis Otakar Borůvka 1926'da Moravya'nın elektrik şebekesini tasarlarken minimum kapsayan ağaç problemini çözdü. Bu, problemin bilinen ilk algoritmasıdır; Sollin aynı yöntemi 1960'larda yeniden önerdiği için iki adla da anılır. Algoritma turlar hâlinde çalışır. Başta her düğüm ayrı bir bileşendir. Her turda *her bileşen* kendisinden dışarı çıkan en ucuz kenarı seçer; seçilen kenarların hepsi aynı anda eklenir ve bileşenler birleşir.



Şekil 17.11: Yedi köy çizgesinde Sollin (Borůvka). Solda başlangıç: yedi bileşen. Ortada 1. tur: (0,5), (1,6), (2,3), (3,4) seçilir, üç bileşen kalır. Sağda 2. tur: (4,5) ve (1,2) seçilir, tek bileşen kalır. Toplam 99.

Birinci turda 0 ile 5'in en ucuz kenarı ortaktır: (0,5) = 10. 1 ile 6 için (1,6) = 14, 2 ile 3 için (2,3) = 12'dir. 4'ün kenarları 22, 24 ve 25'tir; en ucuzu (3,4)'tür. Aynı kenarı iki bileşen seçerse kenar bir kez eklenir. Tur sonunda {0,5}, {1,6} ve {2,3,4} kalır. İkinci turda {0,5}'ten çıkan kenarlar (0,1) 28 ve (4,5) 25'tir; en ucuzu (4,5). {1,6}'nın en ucuzu (1,2) 16'dır. {2,3,4}'ün en ucuzu da (1,2)'dir. Seçilen iki kenar eklenince tek bileşen kalır.

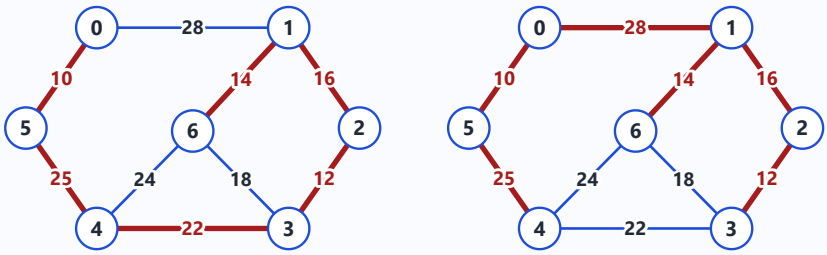
Her turda her bileşen başka bir bileşenle birleşir; bileşen sayısı en az yarıya iner. n bileşenden bire en çok $\lg n$ turda varılır. Her tur bütün kenarlara bir kez bakar: $O(e)$. Toplam $O(e \log n)$'dir. Bir turdaki seçimler birbirinden bağımsız olduğu için iş birçok işlemciye bölünebilir; algoritma paralelleştirmeye uygundur. Ağırlıklarda eşitlik varsa aynı ağırlıklı kenarlar arasında tutarlı bir seçim kuralı (ör. küçük numaralı uç) gerekir; yoksa bir turda döngü kurulabilir.

Ters silme (*reverse-delete*) Kruskal'ın tersidir. Bütün kenarlarla başlar ve kenarları azalan maliyetle dener. Bir kenarı silmek çizgeyi koparmıyorsa kenar silinir; koparıyorsa tutulur. Yedi köy çizgesinde (0,1) 28 silinir; (4,5) 25 tutulur, çünkü silinirse {0,5} kopar. (4,6) 24 ve (3,6) 18 silinir; geri kalanların her biri bir köprüdür ve tutulur. Kalan altı kenar yine 99'luk ağaçtır. Yöntem döngü özelliğine dayanır: silinen her kenar, bir döngünün en ağır kenarıdır.

Dört algoritma da aynı kesit ve döngü özelliklerine dayanır. Ağırlıklar farklıysa minimum kapsayan ağaç tek olduğundan hepsi aynı ağacı bulur.

17.5.2 Minimum kapsayan ağaç ile en kısa yol ağacı

Minimum kapsayan ağaç, sık karıştırılan bir başka yapıya benzer: **en kısa yol** (*shortest path*) ağacı. Bir kaynak düğümden her düğüme giden en kısa yollar birleştirilince bir ağaç çıkar. İkisi de kapsayan ağaçtır ama farklı şeyleri küçültür.



Şekil 17.12: Yedi köy çizgesinin minimum kapsayan ağacı (solda, toplam 99) ve 0'dan en kısa yol ağacı (sağda, toplam 105).

- **Minimum kapsayan ağaç** toplam kenar maliyetini en aza indirir. Kökü yoktur; hangi düğümden bakıldığı önemsizdir. Ağaç üzerinde 0'dan 1'e gitmek 0-5-4-3-2-1 yolunu izler ve *10 + 25 + 22 + 12 + 16 = 85* tutar. Oysa doğrudan (0,1) kenarı 28'dir.
- **En kısa yol ağacı** her düğümün kaynağa uzaklığını tek tek en aza indirir. 0'dan başlayınca 1'e uzaklık 28, 2'ye 44, 3'e 56, 4'e 35, 5'e 10, 6'ya 42'dir. Ağacın kenarları (0,1), (0,5), (1,2), (1,6), (2,3), (5,4)'tür ve toplam maliyeti 105'tir; 99'dan büyüktür.

Hangi ağacın gerektiği soruya bağlıdır. Köylere kablo çekmenin toplam maliyeti soruluyorsa minimum kapsayan ağaç; merkezden her köye en çabuk ulaşım soruluyorsa en kısa yol ağacı aranır. En kısa yol ağacını bulan Dijkstra algoritması Prim'e çok benzer: o da tek bir ağacı büyütür ve her turda en küçük anahtarlı düğümü seçer. Tek fark anahtarın tanımıdır. Prim'de anahtar, düğümü ağaca bağlayan tek bir kenarın maliyetidir; Dijkstra'da kaynaktan o düğüme kadar olan yolun toplam uzunluğudur.

17.5.3 Kruskal mı, Prim mi?

	Kruskal	Prim
Büyüyen yapı	orman (ayrık öbekler birleşir)	tek ağaç (TV)
Seçim ölçütü	bütün kenarlar arasında en ucuzu	TV'den çıkan en ucuz kenar
Döngü denetimi	union-find gerekir	gerekmez ($v \notin TV$)
Veri yapısı	sıralı dizi ya da min-heap + kümeler	`enaz[]` dizisi ya da min-heap
Zaman	$O(e \log e)$	$O(n^2)$ ya da $O(e \log n)$
İyi olduğu yer	seyrek çizge, kenar listesi	yoğun çizge, maliyet matrisi
Bağlantılı olmayan çizge	minimum kapsayan orman verir	yalnız başlangıç düğümünün bileşenini kapsar

Tablo 17.10: Kruskal ile Prim'in karşılaştırması.

Seçimi çizgenin yoğunluğu ve elde hazır olan gösterim belirler. Kenarlar bir dosyadan liste hâlinde geliyorsa ve çizge seyrekse Kruskal en doğal seçimdir: liste olduğu gibi sıralanır. Çizge bir maliyet matrisiyle verilmişse ve yoğunsa dizili Prim hem kısa hem hızlıdır. Seyrek ve komşuluk listesiyle tutulan bir çizgede heap'li Prim ile Kruskal yakın sonuç verir. Son satırdaki fark da önemlidir. Kruskal kopuk bir çizgede durmaz; bütün kenarları dener ve her bileşenin minimum kapsayan ağacını, yani bir ormanı bulur. Prim ise başlangıç düğümünün bileşenini bitirince durur; öteki bileşenler için ayrıca başlatılması gerekir.

17.5.4 Problem çözmede yaklaşım

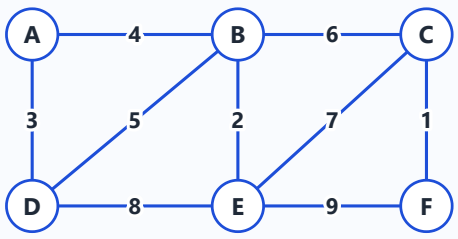
Kruskal ve Prim'in öyküsü bir yöntemi de öğretir. Yeni bir problemle karşılaşan mühendis önce şu soruları sorar: Bu problem daha önce nasıl çözülmüş? Literatürde hangi veri yapıları önerilmiş? Dünya bunu nasıl çözüyor, ben nasıl çözeyim? Hoca, “algoritmaya çok iyi hâkimiz” rahatlığına da karşı çıkar. Bu algoritmalar üzerinde bugün de çalışılıyor. Bir problem çözülürken hangi yapının nerede kullanılacağını bilmek, veri yapıları dersinin asıl kazancıdır.

Sunum bu tutumu ustalık üzerine bir sözle anlatır. Söz, yolu izlemenin aşamalarını sıralar: ustaya bakmak, ustayı izlemek, ustayla yürümek, usta gibi görmek ve sonunda usta olmak. Yani önce literatürün çözümü öğrenilir ve uygulanır; özgün çözüm ondan sonra gelir. Kruskal'ı liste tutarak kodlayan öğrenci ilk aşamayı atlamıştır. Kümelerin ve heap'in bu probleme nasıl uyduğunu gören öğrenci ise sonraki aşamalara geçmiştir.

17.6 Sınav tarzı soru

Kruskal'ın kodlanması ödev olarak verilir; aşağıdaki soru ise sınavlarda sorulabilecek türdendir. Şekil 17.13'teki altı düğümlü çizge için üç şey istenir:

1. Kruskal ile minimum kapsayan ağacı bulmak ve her kenar için “ekle” ya da “döngü” kararını yazmak.
2. Prim'i A'dan başlatıp seçilen kenarların sırasını yazmak.
3. Ağırlıklı birleşimle çalışan union-find'ın `ata[]` dizisinin son hâlini vermek.



Şekil 17.13: Sınav sorusunun çizgesi. Kenarlar: (A,B) 4, (B,C) 6, (A,D) 3, (B,D) 5, (B,E) 2, (C,E) 7, (C,F) 1, (D,E) 8, (E,F) 9.

Kruskal. Kenarlar sıralanır: (C,F) 1, (B,E) 2, (A,D) 3, (A,B) 4, (B,D) 5, (B,C) 6, (C,E) 7, (D,E) 8, (E,F) 9. Düğümler A..F, 0..5 diye numaralanır ve her kenar için `birlestir(ata, kume_bul(ata, u), kume_bul(ata, v))` çağrılır. Tablo 17.11 kararları ve `ata` dizisini verir.

Kenar	w	Kökler	Karar	`ata[]` (A B C D E F)
—	—	—	başlangıç	`-1 -1 -1 -1 -1 -1`
(C,F)	1	C, F	ekle	`-1 -1 -2 -1 -1 2`
(B,E)	2	B, E	ekle	`-1 -2 -2 -1 1 2`
(A,D)	3	A, D	ekle	`-2 -2 -2 0 1 2`
(A,B)	4	A, B	ekle	`-4 0 -2 0 1 2`
(B,D)	5	A, A	döngü	değişmez
(B,C)	6	A, C	ekle	`-6 0 0 0 1 2`

Tablo 17.11: Sınav sorusunda Kruskal ve `ata[A..F]`.

Beşinci kenar seçildiğinde $n - 1 = 5$ olur ve algoritma durur; (C,E), (D,E), (E,F)'ye bakılmaz. Ağaç {(C,F), (B,E), (A,D), (A,B), (B,C)}, maliyeti $1 + 2 + 3 + 4 + 6 = 16$'dır. (B,D) döngü kurar, çünkü B ile D, A üzerinden zaten bağlıdır. Eşit boylu kümelerde ilk parametre kök kalır: (A,B)'de A ile B'nin ikisi de 2 öğelidir, A kök kalır. (B,C)'de A'nın kümesi 4, C'ninki 2 öğelidir; C, A'nın altına girer. Son dizi `[-6, 0, 0, 0, 1, 2]`'dir: A altı öğeli kümenin köküdür; E'nin ebeveyni B, F'nin ebeveyni C'dir.

Prim (A'dan). A'nın kenarları (A,B) 4 ve (A,D) 3'tür: (A,D) seçilir. TV = {A, D}; adaylar (A,B) 4, (D,B) 5, (D,E) 8: (A,B) seçilir. TV = {A, B, D}; adaylar (B,E) 2, (B,C) 6, (D,E) 8: (B,E) seçilir. TV = {A, B, D, E}; adaylar (B,C) 6, (E,C) 7, (E,F) 9: (B,C) seçilir. Son turda (C,F) 1 seçilir. Sıra (A,D) 3 → (A,B) 4 → (B,E) 2 → (B,C) 6 → (C,F) 1, toplam 16'dır. Ağaç Kruskal'ınkiyle aynıdır; ağırlıklar farklı olduğu için başka türlü olamaz. Dikkat çeken nokta, Prim'in en ucuz kenar (C,F)'yi en sona bırakmasıdır: C ağaca girene kadar bu kenar aday değildir.

Bütün değerler bir test sürücüsüyle doğrulandı. Çizgenin 55 kapsayan ağacı vardır ve maliyeti 16 olan tek ağaç budur.

17.7 Ödev: Kruskal'ı veri yapısıyla kodlamak

Hoca bir sonraki hafta için herkesin Kruskal algoritmasını derste anlatıldığı biçimde kodlamasını ister. Beklenen program bir **min-heap** ve **union-find** kullanır. Girdisi bir kenar listesidir; çıktısı ağacın kenarları ve toplam maliyettir. Ödevin gereksinimleri şunlardır:

- Kenar yapısı `kenar {u, v, w}` biçimindedir. Heap, kenar dizisinden aşağıdan yukarıya kurulur ($O(e)$).
- `kume_bul` yol kısaltmalı, `birlestir` ağırlıklı kurallı olmalıdır.
- Çizge bağlantılı değilse ($|T| < n - 1$) program “minimum kapsayan ağaç yok” yazmalıdır.
- Yedi köy örneğinde 99'u bulduğunu ve (3,6) ile (4,6)'nın reddedildiğini gösteren bir çıktı üretmelidir.

Teslimden önce bakılacak bir denetim listesi de verilir. Heap'ten e kez değil, $t = n - 1$ olunca durulmalıdır. Her `malloc` çağrısından sonra `NULL` denetimi yapılmalı, program sonunda ayrılan her blok `free` ile bırakılmalıdır. Eşit ağırlıklı kenarlar varsa minimum kapsayan ağaç tek olmayabilir; farklı öğrencilerin programları farklı ağaçlar bulabilir, ama toplam maliyet hepsinde aynıdır.

Bu notta verilen parçalar ödevin bütün yapı taşlarını içerir: `kenar` yapısı ve `birlestir`, yol kısaltmalı `kume_bul`, aşağı süzme mantığı (heap'li Prim'deki `heap_min_al`) ve ana döngü. Ödevin asıl işi, bunları anahtar olarak kenar ağırlığını kullanan bir heap etrafında birleştirmektir.

17.8 Karmaşıklık özeti

Yöntem	Yapı	Zaman	Açıklama
Bütün alt kümeleri denemek	—	$C(e, n-1) \times O(n)$	üstel; pratik değil
Kruskal, kötü kodlama	listeler + arama	$O(e \cdot n)$	doğru ama böyle kodlanmaz
Kruskal	sıralama ya da min-heap + union-find	$O(e \log e)$	orman büyür, öbekler birleşir
Prim	diziler (`enaz`, `yakin`)	$O(n^2)$	yoğun çizge
Prim	ikili heap + komşuluk listesi	$O(e \log n)$	seyrek çizge
Sollin (Borůvka)	bileşen etiketleri	$O(e \log n)$	en çok $\lg n$ tur; paralel

Yöntem	Yapı	Zaman	Açıklama
DFS/BFS kapsayan ağacı	yığın / kuyruk	$O(n + e)$	ağırlıksız; herhangi bir kapsayan ağaç

Tablo 17.12: Kapsayan ağaç ve minimum kapsayan ağaç yöntemlerinin maliyetleri.

İlk satırdaki $O(n)$ çarpanı, her seçimin döngü denetimidir. Yedi köy gibi küçük bir çizgede bile 84 seçim denenir; 30 düğüm ve 60 kenarlı bir çizgede $C(60, 29)$ 10^{17}'yi aşar. Son satırdaki DFS ve BFS ağırlıklara bakmadığı için herhangi bir kapsayan ağaç bulur. Bütün kenarların ağırlığı eşitse her kapsayan ağaç minimumdur ve doğrusal sürede bulunur.

17.9 Sık yapılan hatalar

1. **Kruskal'da döngüyü "iki uç da daha önce seçilmiş mi?" diye denetlemek.** İki uç farklı öbeklerde olabilir. Yedi köy örneğinde (1,2) bu yanlış kurala takılır, oysa MST'nin kenarıdır. Doğru denetim `kume_bul(u) != kume_bul(v)` 'dir.
2. **Prim'de aday kümesinden iki ucu da TV'de olan kenarları çıkarmamak.** (0,1) beş tur boyunca adaydır, altıncı turda iki ucu da ağaçtadır. Bu kenar seçilirse döngü kurulur. Dizili sürümde bu hata güncelleme döngüsündeki `!secildi[v]` koşulunun unutulmasıdır, heap'li sürümde bayat kayıtları atlamamaktır.
3. **Minimum kapsayan ağaç ile en kısa yol ağacını karıştırmak.** MST toplam maliyeti, en kısa yol ağacı tek tek uzaklıkları küçültür. Yedi köy örneğinde MST'de 0'dan 1'e uzaklık 85, en kısa yol ağacında 28'dir; toplamlar ise 99 ve 105'tir.
4. **Bağlantılı olmayan çizgede $|T| < n - 1$ denetimini unutmak.** Kruskal kenarlar bitince sessizce durur ve elinde bir orman kalır. Prim'de `en_yakin` 'in −1 döndürmesi denetlenmezse `secildi[-1]` yazılır; bu, dizinin dışına yazmaktır.
5. **Union-find'da kök olmayan düğümleri birleştirmek.** `birlestir` iki kök bekler. Önce `kume_bul` çağrılır, sonra bulunan kökler birleştirilir.
6. **Sonsuz için `INT_MAX` kullanıp toplamda taşma yaşamak.** Karşılaştırma güvenli olsa da toplama taşar ve tanımsız davranışa yol açar. Olası en büyük toplamdan büyük, ama taşmayacak bir sabit seçilir.

17.10 Alıştırmalar

1. Yedi köy örneğinde (3,6) kenarının maliyeti 18 yerine 11 olsaydı minimum kapsayan ağaç ve maliyeti ne olurdu?

   Çözüm

   Sıralı kenarlar: (0,5) 10, (3,6) 11, (2,3) 12, (1,6) 14, (1,2) 16, (3,4) 22, (4,6) 24, (4,5) 25, (0,1) 28. Kruskal: (0,5) ekle, (3,6) ekle, (2,3) ekle, (1,6) ekle, (1,2) döngü (1 ile 2, 6 ve 3 üzerinden bağlıdır), (3,4) ekle, (4,6) döngü, (4,5) ekle. Ağaç {(0,5), (3,6), (2,3), (1,6), (3,4), (4,5)}, maliyet $10 + 11 + 12 + 14 + 22 + 25 = 94$. Ucuzlayan (3,6) ağaca girmiş, onunla aynı döngüdeki en pahalı kenar (1,2) çıkmıştır. Kod çıktısında son `ata` dizisi `[3, 3, 3, -7, 3, 0, 3]` 'tür.

2. Kruskal ve Prim aynı çizgede hangi koşulda farklı ağaçlar verebilir?

   Çözüm

   Yalnız eşit ağırlıklı kenarlar varsa. Ağırlıklar farklıysa minimum kapsayan ağaç tektir ve her doğru algoritma onu bulur. Eşitlik varsa iki algoritma eşit kenarlar arasında farklı seçim yapabilir. Örneğin bütün kenarları 1 olan bir üçgende Kruskal sıralamadaki ilk iki kenarı alır. Prim ise başlangıç düğümünün iki kenarını alır ve bunlar farklı olabilir. Toplam maliyet yine aynıdır.

3. Sekiz düğümlü örnek çizgede (Şekil 17.1) 7'den başlayan DFS ve BFS kapsayan ağaçlarının kenarlarını yazın.

   Çözüm

   Komşuluk listeleri sıralıyken DFS(7): (7,3), (3,1), (1,0), (0,2), (2,5), (2,6), (1,4). 5'in komşuları 2 ve 7 işaretli olduğundan 2'ye, oradan 6'ya geçilir; en sonda 1'e dönülüp 4 alınır. BFS(7): (7,3), (7,4), (7,5), (7,6), (3,1), (5,2), (1,0). İki ağaçta da yedi kenar vardır.

4. K_4'ün kaç kapsayan ağacı vardır? Kaçı yol, kaçı yıldız biçimlidir?

> **Çözüm**
>
> 16 (4^{4-2}). Yıldızlar merkezin seçimiyle belirlenir: 4. Yollar dört düğümün sıralamalarıdır, ters okuma aynı yolu verir: *4!/2 = 12*. *12 + 4 = 16*.

5. Kenar ağırlıkları 1..10 arası tam sayıysa Kruskal'ın sıralama adımı nasıl *O(e)*'ye iner?

> **Çözüm**
>
> Karşılaştırmaya dayalı sıralama yerine **saymalı sıralama** (*counting sort*) ya da kova sıralaması kullanılır. Her ağırlık için bir kova açılır, kenarlar ağırlıklarına göre kovalara dağıtılır ve kovalar sırayla okunur. Maliyet *O(e + 10) = O(e)*'dir. Kruskal'ın toplam süresi bu durumda union-find işlemlerine iner.

6. Prim'in $O(n^2)$ sürümü neden seyrek çizgede kötüdür?

> **Çözüm**
>
> Her turda kenar sayısından bağımsız olarak *n* düğümü tarar. $e \approx n$ olan bir çizgede heap'li sürüm *O(e log n) = O(n log n)* sürer; $O(n^2)$ bundan çok büyüktür. 5000 düğüm ve 20.000 kenarlı rastgele bir çizgede ölçülen süreler 0,106 ve 0,003 saniyedir.

17.11 Özet

- Kapsayan ağaç, bağlantılı bir çizgenin bütün düğümlerini $n - 1$ kenarla, döngüsüz bağlayan alt çizgesidir. DFS ve BFS birer kapsayan ağaç üretir; kenarları ağaç ve ağaç dışı olarak ikiye ayırır.
- Bağlantılılık, döngüsüzlük ve $n - 1$ kenar özelliklerinden ikisi üçüncüsünü gerektirir. Ağaca kenar eklemek tek bir döngü kurar, ağaçtan kenar silmek ağacı ikiye böler.
- Kapsayan ağaçların sayısı hızla büyür: K_n'de n^{n-2}. Hepsini denemek pratik değildir; eş biçimliliği sınamak ayrıca zor bir problemdir.
- Minimum kapsayan ağaç, toplam ağırlığı en küçük kapsayan ağaçtır. Açgözlü seçim doğru sonuç verir; gerekçesi kesit özelliğidir. Ağırlıklar farklıysa ağaç tektir.
- Kruskal kenarları artan maliyetle dener, döngüyü union-find ile yakalar ve $n - 1$ kenar seçince durur: *O(e log e)*. Ara durumlar ormandır.
- Union-find bir `ata` dizisidir; kökte küme boyutunun eksi işaretlisi durur. Ağırlıklı kural yüksekliği *O(log n)*'de tutar, yol kısaltma işlem başına maliyeti pratikte sabite indirir.
- Prim tek bir ağacı en ucuz kenarla büyütür ve döngü denetimine ihtiyaç duymaz. `enaz` / `yakin` dizileriyle $O(n^2)$, komşuluk listesi ve heap ile *O(e log n)* sürer.
- Aynı fikir, farklı yapı: liste aramasıyla *O(e · n)* süren Kruskal kümelerle *O(e log e)*'ye iner.
- Seyrek çizgede Kruskal ya da heap'li Prim, yoğun çizgede dizili Prim seçilir. Sollin (Borůvka) turlar hâlinde çalışır ve paralelleştirmeye uygundur.
- Minimum kapsayan ağaç toplam maliyeti, en kısa yol ağacı kaynaktan uzaklıkları küçültür; ikisi farklı ağaçlardır.

17.12 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

kapsayan ağaç *spanning tree* · minimum kapsayan ağaç *minimum spanning tree* · orman *forest* · eş biçimli *isomorphic* · tam çizge *complete graph* · ağırlıklı çizge *weighted graph* · bağlantılı bileşen *connected component* · döngü *cycle* · açgözlü yöntem *greedy method* · Kruskal algoritması *Kruskal's algorithm* · Prim algoritması *Prim's algorithm* · Sollin (Borůvka) algoritması *Sollin's algorithm* · ayrık kümeler *disjoint sets* · küme bulma *find* · birleşim *union* · ağırlıklı kural *weighting rule* · yol kısaltma *path compression* · ters Ackermann fonksiyonu *inverse Ackermann function* · min-heap *min-heap* · öncelikli kuyruk *priority queue* · aşağıdan yukarıya heap oluşturma *bottom-up heap construction* · komşuluk listesi *adjacency list* · derinlik öncelikli arama *depth-first search* · genişlik öncelikli arama *breadth-first search* · en kısa yol *shortest path* · saymalı sıralama *counting sort*

17.13 Kendini Dene

1. 10 düğümlü, 15 kenarlı bağlantılı bir çizgenin her kapsayan ağacında kaç kenar vardır? DFS bu çizgede kaç kenarı ağaç dışında bırakır?

> **Cevap**
>
> Kapsayan ağaçta $n - 1 = 9$ kenar vardır. Kalan $15 - 9 = 6$ kenar ağaç dışıdır; bu sayı gezinmenin türünden ve başlangıç düğümünden bağımsızdır.

2. K_5'in kaç kapsayan ağacı vardır? Bunlardan kaçı yıldız biçimlidir?

> **Cevap**
>
> Cayley formülüyle $5^3 = 125$. Yıldızın merkezi beş biçimde seçilir: 5 yıldız.

3. Yedi köy çizgesinde Kruskal hangi kenarları döngü kurdukları için bırakır? Hangi kenara hiç bakmaz?

> **Cevap**
>
> (3,6) 18 ve (4,6) 24 döngü kurar: ikisinin de uçları o anda aynı öbektedir. (0,1) 28'e hiç bakılmaz, çünkü (4,5) 25 seçildiğinde altı kenar tamamlanmıştır.

4. `ata = [-3, 0, 0, -2, 3]` veriliyor. `kume_bul(ata, 2)` ne döndürür? Ardından `birlestir(ata, 0, 3)` çağrılırsa dizi ne olur?

> **Cevap**
>
> `ata[2] = 0`, `ata[0] = -3 < 0`: kök 0'dır. Birleşimde `ata[0] = -3 <= ata[3] = -2` olduğundan 0 kök kalır: `ata[0] = -5`, `ata[3] = 0`. Dizi `[-5, 0, 0, 0, 3]` olur; 4'ün ebeveyni hâlâ 3'tür.

5. Yedi köy çizgesinde Prim 3'ten başlatılırsa kenarlar hangi sırayla seçilir? Maliyet değişir mi?

> **Cevap**
>
> (3,2) 12, (2,1) 16, (1,6) 14, (3,4) 22, (4,5) 25, (5,0) 10. Maliyet yine 99'dur; ağırlıklar farklı olduğundan ağaç tektir, yalnız seçilme sırası değişir.

6. Heap'li Prim'de `if (secildi[u]) continue;` satırı silinirse yedi köy çizgesinde ne olur?

> **Cevap**
>
> Bayat kayıtlar da işlenir. 6:18, 6:24 ve 1:28 kayıtları çıktığında 6 ve 1 yeniden "ağaca alınır" ve komşuları yeniden taranır. Komşuların hepsi işaretli olduğu için `enaz` ve `yakin` bozulmaz. Ama ağaca giren düğüm sayısı ya da toplam bu satırda sayılıyorsa düğümler birden çok kez sayılır; satır 5, her düğümün tam bir kez işlenmesini güvenceye alır.

7. 1000 düğümlü ve 3000 kenarlı bir yol ağı ile 1000 düğümlü ve yaklaşık 500.000 kenarlı bir tam çizge veriliyor. Her biri için hangi Prim gerçeklemesi seçilmelidir?

> **Cevap**
>
> Yol ağı seyrektir: heap'li Prim yaklaşık $e \log n \approx 3000 \cdot 10 = 30.000$, dizili Prim $n^2 = 10^6$ adım tutar; heap seçilir. Tam çizge yoğundur: heap'li sürüm $5 \cdot 10^5 \cdot 10 = 5 \cdot 10^6$ civarında heap işi yapar, dizili sürüm 10^6 ile kalır; dizili Prim seçilir.

8. Bütün kenar ağırlıkları farklı bir çizgede en ağır kenar minimum kapsayan ağaçta olabilir mi?

> **Cevap**
>
> Olabilir, ama yalnız o kenar bir döngüde değilse, yani silinince çizge kopuyorsa. Örneğin bir düğüme tek bir kenarla bağlanılıyorsa o kenar ne kadar pahalı olursa olsun her kapsayan ağaçta bulunur. Kenar bir döngüdeyse döngü özelliğine göre ağaçta bulunamaz.

17.14 İleri okuma

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, Kesim 6.3. Bu notun Kruskal ve Prim sözde kodları ile yedi köy örneği oradaki biçimiyle izlenmiştir; Sollin algoritması da aynı kesimde anlatılır.
- T. H. Cormen ve ark., *Introduction to Algorithms*, 4. baskı, Bölüm 19 ve 21. Ayrık küme ormanlarının amortize analizi, ters Ackermann sınırının kanıtı ve kesit özelliğinin “güvenli kenar” adıyla genel kanıtı buradadır.
- R. Sedgewick, *Algorithms in C, Part 5: Graph Algorithms*, 3. baskı, Bölüm 20. Prim, Kruskal ve Borůvka'nın C gerçeklemeleri, büyük rastgele çizgelerde yapılmış deneysel karşılaştırmalarla birlikte verilir.

17.15 Kaynaklar

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. bs., Silicon Press, 2008. Kesim 6.3, Minimum Cost Spanning Trees.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. bs., MIT Press, 2022. Bölüm 19, Data Structures for Disjoint Sets; Bölüm 21, Minimum Spanning Trees.
3. R. Sedgewick, *Algorithms in C, Part 5: Graph Algorithms*, 3. bs., Addison-Wesley, 2002. Bölüm 20, Minimum Spanning Trees.

KISIM IV · BÖLÜM 18

En Kısa Yol ve Aktivite Ağları

Bu not ağırlıklı yönlü çizgelerde iki büyük soruyu ele alır: bir yerden ötekine en kısa yol hangisidir ve birbirine bağlı işlerden oluşan bir proje en erken ne zaman biter? İlk soru kaba kuvvetle başlar ve bu çözümün neden kullanılamayacağını sayarak gösterir. Ardından Dijkstra'nın açgözlü algoritması C koduyla kurulur, izlenir ve çözümlenir; yolun kendisi tek bir öncüller dizisinden okunur. Negatif kenarlar için Bellman-Ford, bütün düğüm çiftleri için Floyd-Warshall algoritması verilir. İkinci yarı aktivite ağlarına ayrılmıştır. Düğümde aktivite ağlarında topolojik sıra, sayaçlı başlıklar ve gömülü bir yığınla doğrusal zamanda bulunur. Kenarda aktivite ağlarında aynı yordam en erken zamanları, ters çizgede çalışan eşi de en geç zamanları hesaplar. Bu iki dizi her aktivitenin bolluğunu ve projenin kritik yolunu verir. Her yordamın satır satır anlatımı, izleme tablosu ve karmaşıklık analizi verilir.

Kod (C): `github.com/sercankulcu/data-structures-and-algorithms-c/Bolum18`

ÖĞRENME ÇIKTILARI

Modellemek
Şehir ve demiryolu, ağ gecikmesi ya da proje planı gibi problemleri ağırlıklı yönlü çizge olarak kurmak.

Uygulamak
Dijkstra algoritmasını $O(n^2)$ C koduyla izlemek, doğruluğunu gerekçelendirmek ve yolu öncüller dizisiyle yazdırmak.

Genişletmek
Negatif kenarlı çizgede Bellman-Ford'u, tüm çiftler için Floyd-Warshall'ı uygulamak; hangi problemde hangi algoritmanın seçileceğine karar vermek.

Sıralamak
Düğümde aktivite ağında topolojik sırayı sayaçlı başlıklar ve gömülü yığınla $O(n + e)$'de bulmak, döngüyü yakalamak.

Hesaplamak
Kenarda aktivite ağında en erken ve en geç başlama zamanlarını ileri ve geri çözümlemeyle adım adım hesaplamak.

Yorumlamak
Bollukları bulup kritik aktiviteleri ve kritik yolu belirlemek; bir sürenin değişmesinin projeye etkisini açıklamak.

18.1 En kısa yol problemi

Bir haritada şehirler ve aralarındaki demiryolu hatları düşünülsün. Her hattın bir uzunluğu vardır. Yolcunun sorusu basittir: bir şehirden ötekine en kısa yol hangisidir? Aynı soru bir bilgisayar ağında da sorulur. Orada düğümler yönlendiriciler (*router*), kenarlar iki yönlendirici arasındaki bağlantılardır. Kenarın değeri bu kez uzunluk değil, paketin o bağlantıda beklediği süredir; trafik sıkıştıkça bu süre artar. Bir yönlendiricinin paketi hangi komşusuna göndereceğine karar vermesi, en kısa yol probleminin her saniye yeniden çözülmesi demektir.

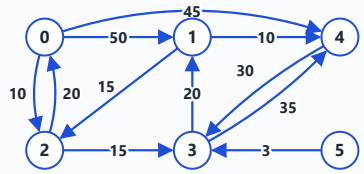
İki örneğin ortak modeli **ağırlıklı yönlü çizgedir** (*weighted directed graph*). Düğümler şehirleri ya da yönlendiricileri, kenarlar bağlantıları gösterir. Her kenarın bir maliyeti (ağırlığı) vardır. Kenarın yönlü olması gerçekçidir: tek yönlü bir yol ya da yalnız bir yönde yoğun olan bir bağlantı, iki yönde farklı maliyetler doğurur. Bir yolun uzunluğu, üzerindeki kenarların maliyetlerinin toplamıdır. Kenar sayısı burada önemli değildir; üç kenarlı bir yol, tek kenarlı bir yoldan kısa olabilir.

> **Tanım 18.1**
>
> $G = (V, E)$ ağırlıklı yönlü bir çizge, $maliyet(<u, w>)$ de kenarların ağırlığı olsun. Bir $v_0, v_1, \ldots, v_k$ yolunun **uzunluğu** $maliyet(<v_0, v_1>) + \ldots + maliyet(<v_{k-1}, v_k>)$ toplamıdır. x'ten y'ye giden yollar içinde uzunluğu en küçük olana **en kısa yol** (*shortest path*) denir.

Problemin içinde iki ayrı soru vardır. Birincisi, *x*'ten *y*'ye herhangi bir yol var mıdır? Bu bir erişilebilirlik sorusudur ve çizgede derinlik öncelikli ya da genişlik öncelikli bir gezinmeyle cevaplanır. İkincisi, birden çok yol varsa en kısası hangisidir? Bu notun konusu ikinci sorudur. Problem birkaç biçimde sorulabilir. **Tek kaynaktan tek hedefe** sürümünde bir başlangıç ve bir varış düğümü verilir. **Tek kaynaktan bütün hedeflere** sürümünde bir kaynak düğümden öteki bütün düğümlere giden en kısa yollar istenir. **Tüm çiftler arası** sürümde ise her düğüm çifti için ayrı bir cevap aranır.

En kısa yol problemi, adı benzeyen başka bir problemle karıştırılmamalıdır. Bütün düğümlere birer kez uğrayıp başlangıca dönen en kısa turu bulmak gezgin satıcı problemidir. Gezgin satıcıda bütün düğümlerin sırası seçilir; en kısa yolda ise yalnız iki düğüm arasındaki güzergâh aranır. İlki için bilinen bütün kesin algoritmalar üstel zaman ister, ikincisi ise çok verimli çözülür.

Not boyunca Şekil 18.1'deki altı düğümlü çizge kullanılır. Bu, Horowitz ve Sahni'nin ders kitabındaki örnektir. 0'dan çıkan kenarlar 1'e (50), 2'ye (10) ve 4'e (45) gider. 2'den 3'e 15 maliyetli, 3'ten 1'e 20 maliyetli kenarlar vardır. 5 numaralı düğümden yalnız çıkan bir kenar vardır (5 → 3); hiçbir düğümden 5'e gelinemez.



Şekil 18.1: Notun ana örneği: altı düğümlü ağırlıklı yönlü çizge. Her ok bir kenardır; okun üzerindeki sayı kenarın maliyetidir.

0 kaynak alınırsa bütün en kısa yolların uzunlukları Tablo 18.1'de görülür. Sonuçların bazıları çizgeden bir bakışta okunmaz. 0'dan 1'e doğrudan 50 maliyetli bir kenar vardır, ama en kısa yol bu kenar değildir: 0 → 2 → 3 → 1 yolunun uzunluğu 10 + 15 + 20 = 45'tir. Üç kenar, tek kenardan daha ucuzdur. 0'dan 4'e ise doğrudan kenar (45) en iyisidir. 0 → 2 → 3 → 4 yolu 10 + 15 + 35 = 60 tutar. 5'e giden kenar olmadığı için 0'dan 5'e yol yoktur.

Hedef	En kısa yol	Uzunluk
2	0 → 2	10
3	0 → 2 → 3	25
1	0 → 2 → 3 → 1	45
4	0 → 4	45
5	—	yol yok

Tablo 18.1: Örnek çizgede 0'dan bütün düğümlere en kısa yollar.

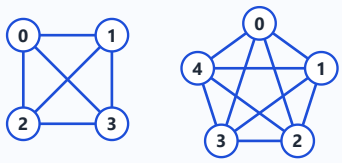
Tablonun satırları uzunluğa göre sıralanmıştır. Bu sıralama rastlantı değildir; en verimli algoritma hedefleri tam da bu sırayla bulur. Satırlar arasında bir bağ da göze çarpar. 1'e giden en kısa yol 3'ün en kısa yolundan geçer, 3'ünki de 2'ninkinden. Her en kısa yol, kendinden kısa bir en kısa yolun sonuna tek kenar eklenerek elde edilir. Bu iki gözlem notun ilk yarısının temelidir.

18.2 Kaba kuvvet çözümü

Bir problemin akla gelen ilk çözümü çoğu zaman **kaba kuvvettir** (*brute force*): bütün adaylar tek tek denenir, en iyisi seçilir. En kısa yol için adaylar, kaynaktan hedefe giden bütün basit yollardır. Basit yol, aynı düğümden iki kez geçmeyen yoldur. Negatif ağırlık yokken en kısa yolun basit olduğu kolayca görülür: bir yol aynı düğüme iki kez uğruyorsa aradaki döngü atılır, uzunluk artmaz. Bu yüzden yalnız basit yolları denemek yeterlidir.

Algoritma tek kaynaklı, tek hedefli sürüm için şöyle kurulur. Kaynaktan başlanır, ziyaret edilmemiş her komşuya geçilir ve hedefe varılınca o yolun uzunluğu en iyi değerle karşılaştırılır. Bu, geri izleme ile yapılan bir derinlik öncelikli aramadır. Mantık doğrudur, sonuç da doğrudur. Sorun, denenecek yol sayısındadır.

En kötü durum, her düğümün öteki bütün düğümlere bağlı olduğu **tam çizgedir** (*complete graph*). n düğümlü tam çizge K_n diye yazılır. Şekil 18.2 K_4 ve K_5'i gösterir. K_4'te 0'dan 3'e beş yol vardır: 0-3, 0-1-3, 0-2-3, 0-1-2-3 ve 0-2-1-3. K_5'te 0'dan 4'e giden yollar ara düğüm sayısına göre sayılır. Ara düğümsüz yol 1 tanedir (0-4). Tek ara düğümlü yollar 3 tanedir; ara düğüm 1, 2 ya da 3 olabilir. İki ara düğümlü yollarda ilk ara düğüm 3, ikincisi 2 biçimde seçilir: 6 yol. Üç ara düğümlü yollarda 1, 2 ve 3'ün bütün sıralamaları denenir: $3! = 6$ yol. Toplam $1 + 3 + 6 + 6 = 16$ yoldur.



Şekil 18.2: Tam çizgeler K_4 ve K_5. K_4 kenarları kesişmeden düzleme çizilebilir; K_5 çizilemez. Yol sayısı için düzlemsellik önemli değildir; önemli olan her düğüm çiftinin bağlı olmasıdır.

Aynı sayma genel n için de yapılır. Kaynak ve hedef dışında $n - 2$ düğüm vardır. k ara düğümlü bir yol, bu düğümlerden k tanesinin sıralı seçimidir. Sıralı seçimlerin sayısı $(n-2)(n-3)\cdots(n-1-k)$'dir. Bütün k değerleri toplanınca yol sayısı bulunur.

Teorem 18.1 (Tam çizgede yol sayısı)

K_n'de iki farklı düğüm arasındaki basit yolların sayısı

$$Y(n) = 1 + (n-2) + (n-2)(n-3) + \cdots + (n-2)!$$

toplamıdır. Toplamın son terimi en büyüğüdür ve $Y(n) < e \cdot (n-2)!$ olur ($e \approx 2{,}718$).

İspat. İlk eşitlik yukarıdaki saymadır. Üst sınır için toplam sondan başa yazılır: $(n-2)! \cdot (1 + 1 + 1/2! + 1/3! + \cdots)$. Parantezdeki seri $1/0! + 1/1! + 1/2! + \ldots$ serisinin bir parçasıdır ve bu serinin toplamı e'dir. ■

Formül bir programla da sınandı. Program, tam çizgede 0'dan $n-1$'e giden yolları geri izlemeyle tek tek sayar. Sonuçlar Tablo 18.2'de formülle yan yana verilmiştir; her satırda iki değer aynıdır. Son sütun, aramanın kaç kez bir kenar boyunca ilerlediğini gösterir. Bu sayı yol sayısının yaklaşık iki katıdır, yani iş miktarı yol sayısıyla aynı hızda büyür.

n	Yol sayısı	Formül	Kenar ilerlemesi
4	5	5	9
5	16	16	31
6	65	65	129
8	1.957	1.957	3.913
10	109.601	109.601	219.201
11	986.410	986.410	1.972.819

Tablo 18.2: Tam çizgede iki düğüm arasındaki basit yol sayısı (programla sayılan ve formülden hesaplanan).

Büyüme faktöriyel hızdadır. 11 düğümde bir milyona yakın yol vardır; 20 düğümde bu sayı $18! \approx 6 \cdot 10^{15}$ mertebesine çıkar. Saniyede bir milyar yol deneyen bir bilgisayar bile bu kadar yolu yaklaşık iki ayda bitirir. Üstelik bu, tek bir hedef içindir.

Sunumdaki $O(n^n)$ gösterimi bu büyümenin kaba ama doğru bir üst sınırıdır. $(n-2)! \le n^n$ olduğu için $Y(n) = O(n^n)$ yazılabilir. Daha sıkı sınır $O((n-2)!)$'dir; ikisi de üsteldir ve pratikte aynı sonucu söyler: bu yöntem ancak birkaç düğümlük çizgelerde kullanılabilir. Kaba kuvvetin mantığı doğrudur. Ama tek bir hedef için bile bu kadar pahalı olan bir yöntemle problem çözülmez. Daha iyisi için yolların yapısındaki düzen kullanılmalıdır.

18.3 Dijkstra algoritması

18.3.1 Açgözlü fikir

Kaba kuvvet her yolu baştan kurar. Oysa Tablo 18.1'deki gözlemler daha akıllı bir yol gösterir: en kısa yollar kısadan uzuna doğru bulunabilir ve her yeni yol, daha önce bulunmuş bir yolun bir kenar uzatılmasıdır. Edsger W. Dijkstra 1959'da bu fikirden çok verimli bir algoritma çıkardı. Algoritma **açgözlüdür** (*greedy*): her adımda o an en iyi görünen seçimi yapar ve bu seçimi bir daha değiştirmez.

Algoritma bir S kümesi tutar. S, en kısa yolu kesinleşmiş düğümlerden oluşur; başta yalnız kaynak, yani 0 vardır. S dışındaki her w düğümü için bir de *uzaklik[w]* değeri tutulur.

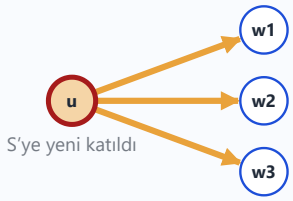
Tanım 18.2

$w \notin S$ için *uzaklik[w]*, kaynaktan başlayan, w'de biten ve aradaki bütün düğümleri S'de olan yolların en kısasının uzunluğudur. Böyle bir yol yoksa *uzaklik[w]* = ∞ alınır.

Tanımdaki kısıt önemlidir: *uzaklik[w]* bütün yolların değil, yalnız S'nin içinden geçen yolların en kısasıdır. Başta $S = \{0\}$ olduğu için bu yollar tek kenarlıdır ve *uzaklik[w]*, 0'dan w'ye giden kenarın maliyetine eşittir. Algoritma üç gözleme dayanır.

1. Bir sonraki kesinleşecek u düğümünün en kısa yolu, u dışında yalnız S'deki düğümlerden geçer. Yolun ortasında S dışından bir x bulunsaydı, kaynaktan x'e giden kısım daha kısa olurdu ve x daha önce seçilirdi.
2. Bu yüzden bir sonraki düğüm, S dışında *uzaklik* değeri en küçük olan düğümdür. Eşitlik varsa herhangi biri seçilebilir.
3. u seçilip S'ye katılınca, u'dan geçen yeni yollar açılır. S dışındaki bir w için *uzaklik[u] + maliyet(<u, w>) < uzaklik[w]* ise *uzaklik[w]* bu küçük değerle güncellenir. Yeni yolda u ile w arasında ara düğüm yoktur; u'dan sonra doğrudan w'ye geçilir.

Üçüncü gözlemdeki işleme **gevşetme** (*relaxation*) denir. Ad, *uzaklik[w]*'nin bir üst sınır olmasından gelir: değer gerçek en kısa uzaklıktan hiçbir zaman küçük değildir. Gevşetme bu sınırı aşağı çeker, yani "gerilmiş" bir tahmini gevşetir. Şekil 18.3 bir gevşetme adımını gösterir: yeni katılan u'nun komşuları w_1, w_2, w_3 için u üzerinden yeni yollar denenir.



Şekil 18.3: S'ye yeni katılan u, komşularına kendi üzerinden geçen yeni yollar açar. Her w_i için *uzaklik[u] + maliyet(<u, w_i>)* eski değerle karşılaştırılır.

Birinci gözlemin gerekçesi bir koşula dayanır: kenar ağırlıkları negatif olmamalıdır. Bu koşul olmadan algoritma yanılır. Doğruluğu kesin biçimde ifade etmek için bir teorem yazılabilir.

Teorem 18.2 (Dijkstra'nın doğruluğu)

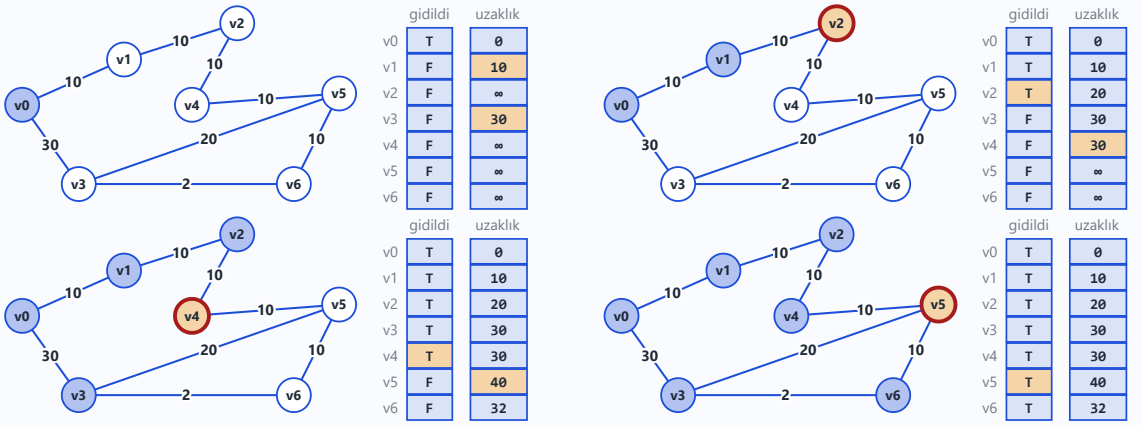
Bütün kenar ağırlıkları negatif değilse, S dışında *uzaklik* değeri en küçük olan u düğümü için *uzaklik[u]*, kaynaktan u'ya giden en kısa yolun uzunluğudur.

İspat. Kaynaktan u'ya herhangi bir P yolu alınsın. P kaynakta, yani S'nin içinde başlar ve S'nin dışındaki u'da biter. Öyleyse P üzerinde S dışındaki ilk bir y düğümü vardır. P'nin kaynaktan y'ye kadarki kısmı yalnız S'nin içinden geçer; tanım gereği bu kısmın uzunluğu en az *uzaklik[y]*'dir. u en küçük değerli düğüm olarak seçildiği için *uzaklik[y]* ≥ *uzaklik[u]* olur. P'nin y'den u'ya kalan kısmının uzunluğu ağırlıklar negatif olmadığı için 0 ya da daha büyüktür. Toplamda P'nin uzunluğu *uzaklik[u]*'dan küçük olamaz. *uzaklik[u]* de gerçek bir yolun uzunluğu olduğuna göre en kısa yolun uzunluğudur. ∎

Kanıtın tek zayıf halkası "kalan kısım 0 ya da daha büyüktür" cümlesidir. Negatif bir kenar bu cümleyi bozar: y'den u'ya negatif bir kenarla inilirse P, *uzaklik[u]*'dan kısa olabilir. Bu durumun somut bir örneği ve çaresi negatif kenarlar kesiminde verilir.

18.3.2 Adım adım bir örnek

Algoritmanın işleyişini görmek için yedi düğümlü küçük bir çizge alınsın. Kenarlar şunlardır: *v0* → *v1* (10), *v1* → *v2* (10), *v2* → *v4* (10), *v4* → *v5* (10), *v0* → *v3* (30), *v3* → *v5* (20), *v3* → *v6* (2) ve *v6* → *v5* (10). Kaynak *v0*'dır. Şekil 18.4 yedi adımın dördünü, Tablo 18.3 hepsini verir. Şekillerde koyu renkli düğümler *S*'dedir; sağdaki tablo `gidildi` işaretini (T) ve `uzaklik` değerini gösterir.



Şekil 18.4: Yedi düğümlü çizgede Dijkstra'nın dört anı. Sol üstte başlangıç: yalnız *v0* kesindir, komşuları doğrudan kenar maliyetlerini alır. Sağ üstte *v2* katılmış, *v4* 30'a inmiştir. Sol altta *v4* katılmış ve *v5* 50'den 40'a inmiştir. Sağ altta bütün düğümler kesindir.

Adım	Katılan	v1	v2	v3	v4	v5	v6	Güncellenen
0	v0	10	∞	30	∞	∞	∞	v1, v3 (doğrudan kenar)
1	v1 (10)	**10**	20	30	∞	∞	∞	v2 = 20
2	v2 (20)	**10**	**20**	30	30	∞	∞	v4 = 30
3	v3 (30)	**10**	**20**	**30**	30	50	32	v5 = 50, v6 = 32
4	v4 (30)	**10**	**20**	**30**	**30**	40	32	v5 = 40
5	v6 (32)	**10**	**20**	**30**	**30**	40	**32**	yok (32 + 10 = 42 > 40)
6	v5 (40)	**10**	**20**	**30**	**30**	**40**	**32**	yok

Tablo 18.3: Yedi düğümlü çizgede *S*'ye katılma sırası ve uzaklıkların değişimi (∞: henüz yol yok).

Tablonun iki satırı özellikle öğreticidir. Dördüncü adımda *v5*'in değeri 50'den 40'a iner. *v5*'e ilk bulunan yol *v0* → *v3* → *v5*'tir (30 + 20). *v4* katılınca dört kenarlı *v0* → *v1* → *v2* → *v4* → *v5* yolu (10 + 10 + 10 + 10) bulunur ve daha kısadır. Bir düğümün değeri, düğüm *S*'ye girene kadar birkaç kez değişebilir; kesinleşmiş sayılmaz.

Beşinci adımda ise *v6*, *v5*'ten önce seçilir. *v6*'nın numarası daha büyüktür, ama uzaklığı (32) *v5*'inkinden (40) küçüktür. Seçim sırası düğüm numarasına göre değil, uzaklığa göre belirlenir. Üçüncü ve dördüncü adımlarda ise *v3* ile *v4*'ün uzaklıkları eşittir (30). Kuramsal olarak hangisinin önce seçildiği fark etmez. C kodu eşitlikte küçük numaralı düğümü seçtiği için önce *v3* katılır.

18.3.3 C gerçeklemesi: veri ve `minimumBul`

Algoritmanın C sürümü çizgeyi bir **maliyet matrisiyle** tutar. `maliyet[i][j]` , *<i, j>* kenarının maliyetidir. Kenar yoksa ∞ yerine büyük bir sabit yazılır; köşegendeki değerler 0'dır. Kesinleşmiş düğümler `gidildi` dizisinde işaretlenir; *S* kümesi budur. `uzaklik` dizisi tanımdaki değerleri taşır. Aşağıdaki kod örnek çizgenin matrisini ve en küçük uzaklıklı işaretsiz düğümü bulan yordamı verir.

```c
#include <limits.h>
#define MAX_DUGUM 6
#define SONSUZ 1000      /* baglanti yok */
int maliyet[][MAX_DUGUM] = {
    {   0,   50,  10, 1000,   45, 1000},
    {1000,    0,  15, 1000,   10, 1000},
    {  20, 1000,   0,   15, 1000, 1000},
    {1000,   20, 1000,   0,   35, 1000},
    {1000, 1000, 1000,  30,    0, 1000},   /* eksik satir */
    {1000, 1000, 1000,   3, 1000,    0}};  /* 100 -> 1000 */
int minimumBul(int uzaklik[], int n, short gidildi[])
{
    int i, min = INT_MAX, pos = -1;
    for (i = 0; i < n; i++)
        if (uzaklik[i] < min && !gidildi[i]) {
            min = uzaklik[i];  pos = i;
        }
    return pos;
}
```

- **Satır 1.** `INT_MAX` sabiti `limits.h` başlık dosyasındadır; `int` türünün alabileceği en büyük değerdir. Başlık eklenmezse derleyici `INT_MAX`'ı tanımaz.
- **Satır 2–3.** Düğüm sayısı ve sonsuzu temsil eden sabit. `SONSUZ` için 1000 seçilmiştir; örnekteki hiçbir gerçek yol bu kadar uzun değildir.
- **Satır 4–10.** Maliyet matrisi. Satır kenarın çıktığı düğümü, sütun vardığı düğümü gösterir. 0. satırdaki `50, 10, 1000, 45` değerleri 0 → 1, 0 → 2, 0 → 3 (kenar yok) ve 0 → 4 kenarlarıdır. İlk boyut boş bırakılmıştır; derleyici satır sayısını ilk değer listesinden çıkarır. İkinci boyut ise yazılmak zorundadır, çünkü derleyici bir satırın kaç `int` tuttuğunu bilmelidir.
- **Satır 11–19.** `minimumBul`. `min` en büyük `int` ile, `pos` da geçersiz bir indisle (−1) başlar. Döngü işaretsiz düğümler arasında `uzaklik` değeri en küçük olanı arar. Karşılaştırma katı küçüktür (`<`); eşit değerler arasında ilk görülen, yani küçük numaralı düğüm kalır.

Matrisin iki satırında açıklama vardır. Ders kitabındaki matriste 4. satır eksiktir; bu satır çizgeden okunarak tamamlanmıştır (4'ten tek kenar 4 → 3, maliyet 30). 5. satırda da kitap sonsuz için 100 yazar; öteki satırlarla tutarlı olsun diye 1000'e çevrilmiştir. Ders kitaplarındaki küçük yazım hataları, kodu gerçekten çalıştırınca hemen ortaya çıkar. Kâğıt üzerinde izlenen kodla çalışan kodun aynı olması bu yüzden önemlidir.

`SONSUZ` değerinin seçimi göründüğünden inceliklidir. İlk akla gelen `INT_MAX`'tır, ama bu değer gevşetmede taşmaya yol açar: `uzaklik[u] + maliyet[u][w]` toplamında iki terimden biri `INT_MAX` ise sonuç `int`'e sığmaz. C'de işaretli tam sayı taşması tanımsız davranıştır; pratikte değer çoğu zaman negatif bir sayıya döner ve "daha kısa yol bulundu" sanılır. 1000 gibi bir sabitte bu tehlike yoktur: `SONSUZ + SONSUZ` 2000'dir ve rahatça sığar. Bedeli de vardır: sabit, çizgedeki en uzun gerçek yoldan büyük seçilmelidir. Aksi hâlde gerçek bir yol "yol yok" ile karışır.

Yordamın −1 döndürdüğü tek durum, işaretsiz düğüm kalmamasıdır. Erişilemeyen bir düğüm kalmışsa yordam yine onu seçebilir, çünkü değeri 1000'dir ve `INT_MAX`'tan küçüktür. Bu zararsızdır. Erişilemeyen düğümün değeri 1000'dir; üzerinden yapılan gevşetmeler 1000'den küçük bir değer üretmez, çünkü ağırlıklar negatif değildir. Yine de çağıran yordam −1 dönüşünü denetlemelidir; denetimsiz bir `gidildi[-1]` erişimi dizinin dışına yazar.

> **Sık yapılan hata**
>
> Koşulu `!gidildi(w)` biçiminde yazmak. Yuvarlak ayraç diziyi bir fonksiyon gibi çağırır ve derleyici "çağrılabilir bir nesne değil" hatası verir. Dizi öğesine köşeli ayraçla erişilir: `!gidildi[w]`. Benzer bir başka hata `FALSE` ve `TRUE`'yu tanımlamadan kullanmaktır; bunlar C'nin anahtar sözcükleri değildir, `#define` ile tanımlanmaları gerekir.

18.3.4 `EnKisaYolBul`: ana yordam

Ana yordam kaynak düğümü, maliyet matrisini ve üç diziyi alır. `uzaklik` ve `gidildi` yukarıdaki anlamlarıyla kullanılır. `onceki` dizisi ise yolun kendisini bulmak için tutulur; ayrıntısı bir sonraki kesimdedir.

```
void EnKisaYolBul(int v, int maliyet[][MAX_DUGUM],
                  int uzaklik[], int onceki[],
                  int n, short gidildi[])
{
    int i, u, w;
    for (i = 0; i < n; i++) {                  /* A1 */
        gidildi[i] = FALSE;
        uzaklik[i] = maliyet[v][i];
        onceki[i] = v;             /* yol icin tutulur */
    }
    gidildi[v] = TRUE;  uzaklik[v] = 0;
    for (i = 2; i < n; i++) {        /* n-2 tur */
        u = minimumBul(uzaklik, n, gidildi);
        if (u < 0) break;            /* kalan yok */
        gidildi[u] = TRUE;
        for (w = 0; w < n; w++)
            if (!gidildi[w] &&
                uzaklik[u] + maliyet[u][w] < uzaklik[w]) {
                uzaklik[w] = uzaklik[u] + maliyet[u][w];
                onceki[w] = u;
            }
    }
}
```

Yordamın dört bölümü vardır ve analizde bu bölümlerin adları kullanılır:

- **A1 (satır 6–10).** Başlangıç değerleri. Her düğüm işaretsizdir. `uzaklik[i]`, kaynaktan `i`'ye doğrudan kenarın maliyetidir; bu, $S = \{v\}$ için tanımın verdiği değerdir. `onceki[i] = v` atamasıyla her düğümün öncülü başta kaynak sayılır.
- **Satır 11.** Kaynak işaretlenir, uzaklığı 0 yapılır. Matrisin köşegeni zaten 0'dır; atama yine de açık yazılmıştır, çünkü kod köşegene güvenmemelidir.
- **A2 (satır 12).** Dış döngü $i = 2$'den $n - 1$'e, yani $n - 2$ tur döner. Kaynak zaten kesindir. Her tur bir düğümü daha kesinleştirir. $n - 2$ tur sonunda tek bir işaretsiz düğüm kalır. O düğüm için bütün öteki düğümler S'dedir ve tanımdaki değer en kısa yolun uzunluğudur. Son bir tur yalnız onu işaretlerdi; güncellenecek işaretsiz düğüm kalmadığı için hiçbir şey değişmezdi.
- **A3 (satır 13–15).** En küçük uzaklıklı işaretsiz düğüm `u` bulunur ve işaretlenir. −1 dönerse işaretsiz düğüm kalmamıştır, döngüden çıkılır.
- **A4 (satır 16–21).** Gevşetme. İşaretsiz her `w` için `u` üzerinden geçen yol denenir. Daha kısaysa hem uzaklık hem öncül güncellenir.

Satır 17–18'deki koşul iki parçalıdır ve sıra önemlidir. `&&` işleci soldan sağa, kısa devreyle değerlendirilir: `gidildi[w]` doğruysa toplama hiç yapılmaz. Bu, kesinleşmiş düğümlerin değerlerinin bozulmasını önler. Kesinleşmiş bir düğüme daha kısa bir yol zaten bulunamaz (Teorem 18.2), ama kodun bunu açıkça denetlemesi hem okunurluğu hem güvenliği artırır.

Maliyet matrisi parametresi `int maliyet[][MAX_DUGUM]` biçiminde yazılmıştır. C bir diziyi fonksiyona kopyalamaz; ilk öğenin adresini geçirir. İki boyutlu bir dizide `maliyet[u][w]`'nin adresini hesaplamak için satır uzunluğu gerekir: adres, başlangıç + $(u \cdot MAX_DUGUM + w) \cdot sizeof(int)$'tir. Bu yüzden ikinci boyut parametrede yazılmak zorundadır. Aynı nedenle `uzaklik`, `onceki` ve `gidildi` dizilerine yordamın içinde yapılan atamalar çağıranın dizilerini değiştirir; sonuçlar bu yolla geri döner.

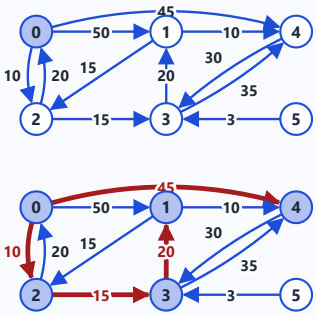
18.3.5 Kod izlemesi

`EnKisaYolBul(0, maliyet, …)` çağrısı örnek çizgede izlenir. Tablo 18.4 her turda seçilen düğümü, yapılan gevşetmeleri ve tur sonundaki `uzaklik` dizisini verir. Değerler programın her tur sonunda yazdırdığı çıktıdan alınmıştır.

Tur (`i`)	Seçilen `u`	Gevşetmeler	`uzaklik[0..5]`
A1	—	0'ın satırı kopyalanır	0 50 10 1000 45 1000
2	2 (10)	`w` = 3: 10 + 15 = 25 < 1000 → `onceki[3] = 2`	0 50 10 25 45 1000
3	3 (25)	`w` = 1: 25 + 20 = 45 < 50 → `onceki[1] = 3`	0 45 10 25 45 1000
4	1 (45)	yok (1 → 4: 45 + 10 = 55 > 45)	0 45 10 25 45 1000
5	4 (45)	yok (4'ün tek kenarı 3'e gider, 3 işaretli)	0 45 10 25 45 1000

Tablo 18.4: Örnek çizgede `EnKisaYolBul(0, …)` izlemesi (1000: sonsuz).

Üçüncü turda 3 üzerinden 4'e de bir yol denenir: 25 + 35 = 60. Bu, 45'ten büyük olduğu için değer değişmez. Dördüncü turda 1 ile 4'ün uzaklıkları eşittir (45). `minimumBul` katı küçük karşılaştırması yaptığı için ilk gördüğü 1'i seçer. Döngü *n − 2 = 4* tur döner ve biter. 5 hiç seçilmemiştir; değeri 1000 kalır ve "yol yok" diye okunur. Şekil 18.5 izlemenin ilk ve son anını gösterir.



	0	1	2	3	4	5
uzaklik	0	50	10	∞	45	∞
gidildi	T	F	F	F	F	F
onceki	0	0	0	0	0	0

	0	1	2	3	4	5
uzaklik	0	45	10	25	45	∞
gidildi	T	T	T	T	T	F
onceki	0	3	0	2	0	0

Şekil 18.5: `EnKisaYolBul(0, …)` izlemesinin ilk ve son anı. Üstte A1 sonrası: `uzaklik` dizisi 0'ın satırının kopyasıdır, `onceki` dizisinin bütün öğeleri 0'dır. Altta dört tur bitmiştir: kalın kenarlar en kısa yol ağacını oluşturur.

Sonuç Tablo 18.1 ile birebir aynıdır: 2 → 10, 3 → 25, 1 → 45, 4 → 45, 5'e yol yok. Seçilme sırası da uzunluk sırasıdır. Kaynak 2 alınırsa uzaklıklar 0'a 20, 3'e 15, 1'e 35 (3 üzerinden), 4'e 45 (3 ve 1 üzerinden) olur; 5'e yine yol yoktur. Bu değerler de programla bulunmuştur.

18.3.6 Analiz ve daha iyi veri yapıları

Maliyet her bölüm için ayrı hesaplanır. A1 bir kez n tur döner: $O(n)$. A2'nin dış döngüsü $n - 2$ tur döner. Her turda A3, yani `minimumBul`, bütün diziyi tarar: $O(n)$. A4 de her `w`'yi bir kez dener: $O(n)$. Toplam

$$n + (n - 2) \cdot (n + n) \approx n + 2n^2 = O(n^2)$$

olur. Bu sınır en iyi durumda da geçerlidir: kenar sayısı ne olursa olsun her tur iki tam tarama yapar. Yani algoritmanın süresi $\Theta(n^2)$'dir. Ek bellek üç dizidir: $O(n)$. Kaba kuvvetin faktöriyel büyümesiyle karşılaştırıldığında fark çarpıcıdır. 20 düğümde kaba kuvvet aylar sürerken bu yordam birkaç yüz işlemle biter. Matrisin en büyük öğesini bulmak bile n^2 öğeye bakmayı gerektirir; Dijkstra bütün en kısa yolları aynı mertebede bir sürede bulur.

$O(n^2)$ yoğun çizgeler için iyidir. Seyrek çizgelerde ise iki darboğaz vardır. Birincisi A3'tür: en küçük öğeyi bulmak için her turda bütün dizi taranır. Oysa en küçük öğeyi hızlı vermek, **min-heap** ile gerçeklenen bir **öncelikli kuyruğun** (*priority queue*) işidir. Uzaklıklar min-heap'te tutulursa en küçük öğe $O(\log n)$'de alınır. İkincisi A4'tür: matris kullanıldığı için her turda n sütun taranır, oysa gerçekte yalnız `u`'nun komşuları ilgilenilmesi gereken düğümlerdir. Komşuluk listesi kullanılırsa A4, `u`'nun çıkış derecesi kadar sürer.

İki iyileştirme birlikte uygulanınca toplam maliyet kolayca yanlış hesaplanır. Sık yapılan hata sonucu $O(n \log n)$ sanmaktır. Doğru hesap şöyledir. Her düğüm heap'ten bir kez alınır: n kez $O(\log n)$. Her kenar bir kez gevşetilir ve başarılı her gevşetme heap'teki bir değeri küçültür; bu da $O(\log n)$ sürer. Kenar sayısı e olduğundan toplam

$$O(n \log n + e \log n) = O((n + e) \log n)$$

olur. Fibonacci heap adlı daha karmaşık bir yapı değer küçültmeyi ortalama $O(1)$'e indirir ve sınırı $O(e + n \log n)$'ye çeker. Tablo 18.5 üç yaklaşımı yan yana koyar.

Yöntem	Veri yapısı	Süre	Uygun olduğu çizge
Kaba kuvvet	bütün yollar	$O(n^n)$	birkaç düğüm
Dijkstra	matris + diziler	$O(n^2)$	yoğun ($e \approx n^2$)
Dijkstra	min-heap + komşuluk listesi	$O((n + e) \log n)$	seyrek ($e \approx n$)

Tablo 18.5: Tek kaynaklı en kısa yol için üç yaklaşım.

Hangi sürümün daha iyi olduğu çizgeye bağlıdır. Seyrek bir çizgede, örneğin bir yol haritasında, her düğümün birkaç komşusu vardır ve e, n mertebesindedir. Heap sürümü burada $O(n \log n)$'e iner ve dizi sürümünü açık farkla geçer. Yoğun bir çizgede ise $e \approx n^2$ olur ve heap sürümü $O(n^2 \log n)$'ye çıkar; basit dizi sürümü daha hızlıdır. Öncelikli kuyruk öğrenildikten sonra dizide en küçük öğeyi aramak hantal görünebilir. Ama doğru soru "hangi yapı daha zarif?" değil, "bu çizgede hangi işlem kaç kez yapılıyor?" sorusudur.

18.4 Yolu bulmak: öncüller dizisi

18.4.1 Tek dizi neden yeter?

`EnKisaYolBul` uzunlukları bulur, ama yolcu yalnız uzunlukla yetinmez; hangi şehirlerden geçeceğini de bilmek ister. Maliyet bulunduktan sonra sıra yoldadır. İlk akla gelen, her düğüm için yolun bütün düğümlerini ayrı bir listede tutmaktır. Bu hem bellek israfıdır hem de her güncellemede listeyi kopyalamayı gerektirir. Yolu kaynaktan ileri doğru kurmak da mümkün değildir: kaynaktan çıkarken hangi kenardan gidileceği, algoritma bitene kadar belli olmaz.

Çözüm tek bir dizidir. Her düğümün en kısa yolu, **öncülünün** (*predecessor*) en kısa yolunun sonuna tek bir kenar eklenerek elde edilir. Bu yüzden her düğüm için yalnız yoldaki bir önceki düğümü saklamak yeterlidir.

> **Tanım 18.3**
>
> Kaynaktan h'ye bulunan en kısa yolda h'den hemen önce gelen düğüme h'nin **öncülü** denir ve `onceki[h]` 'de saklanır. Bütün düğümlerin öncüllerini tutan diziye **öncüller dizisi** denir.

Tek dizinin yetmesinin nedeni en kısa yolların bir özelliğidir. Kaynaktan h'ye giden bir en kısa yol u'dan geçiyorsa, bu yolun kaynaktan u'ya kadarki parçası da kaynaktan u'ya bir en kısa yoldur. Parça daha kısa bir yolla değiştirilebilseydi bütün yol da kısalırdı. Bu özelliğe **en iyi alt yapı** (*optimal substructure*) denir. Sonuç olarak bütün en kısa yollar bir ağaç oluşturur. Kökü kaynaktır ve her düğümün ebeveyni öncülüdür. Bu ağaca **en kısa yol ağacı** denir. Öncüller dizisi, kümeler için kullanılan ebeveyn dizisi gibi, bu ağacın her düğümünden ebeveynine giden bağları tutar.

Dizi gevşetme sırasında güncellenir. `EnKisaYolBul` 'un 20. satırı, gevşetme başarılı olduğunda `onceki[w] = u` atamasını yapar: `w` 'ye giden en iyi yol artık `u` 'dan geçer. Maliyet değiştikçe öncül de değişir. Örnekte 1'in değeri 50'den 45'e inerken öncülü de 0'dan 3'e geçmiştir. Algoritma bittiğinde örnek çizge için öncüller dizisi `onceki = {0, 3, 0, 2, 0, 0}` olur. 1'in öncülü 3, 3'ünki 2, 2'ninki 0'dır.

Dizi ileriye değil geriye doğru okunur. Hedeften başlanıp öncüllere yürünerek kaynağa varılır: $1 \leftarrow 3 \leftarrow 2 \leftarrow 0$. Bu sıra yolun tersidir. Doğru sırada yazmak için ya özyineleme ya da bir yığın kullanılır. İkisi de aynı ilkeye dayanır: son giren ilk çıkar.

18.4.2 Özyinelemeli yol yazdırma

Özyinelemeli çözüm kısa ve zariftir:

```
void yol_yaz(int onceki[], int s, int h)
{
    if (h == s) {
        printf("%d", s);
        return;
    }
    yol_yaz(onceki, s, onceki[h]);   /* once oncul */
    printf(" -> %d", h);
}
```

- **Satır 3–6.** Temel durum: hedef kaynağın kendisiyse yol tek düğümdür; kaynak yazılır ve dönülür.
- **Satır 7.** Önce öncüle kadar olan yol yazdırılır. Özyinelemeli çağrı, `h` 'yi öncülüyle değiştirerek kaynağa bir adım yaklaşır.
- **Satır 8.** Öncüle kadarki yol yazıldıktan sonra `h` yazılır. Yazma işi çağrıdan *sonra* yapıldığı için düğümler geri dönüş sırasında, yani kaynaktan hedefe doğru çıkar.

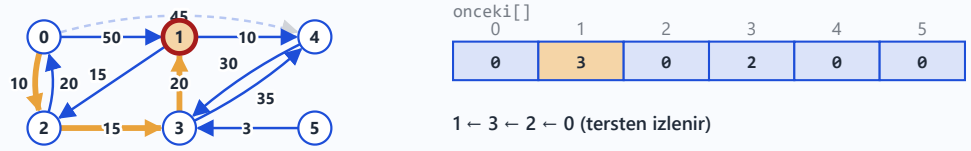
Yordamın doğru çalışması, öncül zincirinin sonunda kaynağa varmasına bağlıdır. Erişilebilen her düğüm için bu böyledir, çünkü öncüller en kısa yol ağacının ebeveyn bağlarıdır. Erişilemeyen bir düğümde ise `onceki` değeri anlamsızdır. Örnekte `onceki[5] = 0` 'dır, ama 0'dan 5'e yol yoktur; bu değer A1'deki başlangıç atamasından kalmıştır. `yol_yaz(onceki, 0, 5)` çağrılırsa "0 -> 5" gibi gerçekte var olmayan bir yol yazılır. Bu yüzden çağıran yordam önce `uzaklik[h] >= SONSUZ` koşulunu denetlemeli, öyleyse "yol yok" yazmalıdır.

`yol_yaz(onceki, 0, 1)` çağrısının izlemesi Tablo 18.6'dadır. Çağrı yığını sütununda en soldaki çerçeve en eski, en sağdaki en yeni çağrıdır.

Adım	Olay	Çağrı yığını	Çıktı
1	`h` = 1 ≠ 0; `onceki[1]` = 3 için çağrı	yy(1)	
2	`h` = 3 ≠ 0; `onceki[3]` = 2 için çağrı	yy(1) yy(3)	
3	`h` = 2 ≠ 0; `onceki[2]` = 0 için çağrı	yy(1) yy(3) yy(2)	
4	`h` = `s` = 0: kaynak yazılır, dönülür	yy(1) yy(3) yy(2) yy(0)	`0`
5	`yol_yaz(2)` 'ye dönüldü: " -> 2"	yy(1) yy(3) yy(2)	`0 -> 2`
6	`yol_yaz(3)` 'e dönüldü: " -> 3"	yy(1) yy(3)	`0 -> 2 -> 3`
7	`yol_yaz(1)` 'e dönüldü: " -> 1"	yy(1)	`0 -> 2 -> 3 -> 1`

Tablo 18.6: `yol_yaz(onceki, 0, 1)` izlemesi (`onceki = {0, 3, 0, 2, 0, 0}`).

İzleme yolun hedeften geriye yürünerek bulunduğunu, yazarken ise çağrı yığınının sırayı düzelttiğini gösterir. Yolun maliyeti 10 + 15 + 20 = 45'tir ve `uzaklik[1]` ile tutarlıdır. Şekil 18.6 öncül bağlarını çizge üzerinde gösterir.



Şekil 18.6: Öncüller dizisiyle 1'in yolu: 1 ← 3 ← 2 ← 0. Vurgulu kenarlar öncül bağlarıdır; zincir tersten izlenir, yol ise kaynaktan hedefe yazılır.

Yordamın maliyeti yolun uzunluğuyla orantılıdır. Yolda k kenar varsa $k + 1$ çağrı yapılır ve her çağrı sabit iş yapar: $O(k)$. k en çok $n - 1$ olabilir. Çağrı yığınının derinliği de $k + 1$'dir; bu yüzden ek bellek de $O(k)$'dir.

18.4.3 Yığınla yol yazdırma

Aynı iş özyineleme kullanmadan, açık bir yığınla da yapılır. Önce hedeften kaynağa yürünür ve geçilen düğümler yığına konur. Sonra kaynak yazılır ve yığındaki düğümler çıkış sırasıyla yazılır. Yığın, öncül zincirinin sırasını tersine çevirir.

```
void yol_yaz_yigin(int onceki[], int s, int h)
{
    int yigin[MAX_DUGUM], ust = -1;
    while (h != s) {                  /* hedeften kaynaga yuru */
        yigin[++ust] = h;
        h = onceki[h];
    }
    printf("%d", s);
    while (ust >= 0)                  /* ters sirada yaz */
        printf(" -> %d", yigin[ust--]);
}
```

Yığın burada bir dizi ile bir `ust` indisinden oluşur. `yigin[++ust] = h` ifadesi önce `ust`'ü artırır, sonra o hücreye yazar; bu bir ekleme (*push*) işlemidir. `yigin[ust--]` ise önce hücreyi okur, sonra `ust`'ü azaltır; bu bir çıkarma (*pop*) işlemidir. Yolda en çok $n - 1$ kenar olduğu için `MAX_DUGUM` boyutlu dizi yeter. Program, örnek çizgedeki dört erişilebilir hedef için iki yordamın aynı çıktıyı verdiğini doğruladı. Özyinelemeli sürüm daha kısadır, yığınlı sürüm ise çağrı yığınının derinliğine bağlı değildir. Çok uzun yolların olabileceği büyük çizgelerde ikincisi daha güvenlidir.

18.4.4 Küçük yordamlar ve yazılım kalitesi

Bu bölümdeki kodlar bilinçli olarak küçük yordamlara bölünmüştür: `minimumBul`, `EnKisaYolBul`, `yol_yaz`. Her biri tek bir iş yapar ve ayrı ayrı sınanabilir. Bölmenin değeri en çok hatalı örneklerde anlaşılır. Bir sınavda en kısa yol sorusunu tek bir büyük özyinelemeli yordamla çözmeye çalışan öğrencilerin kodları çoğu zaman ne okunabildi ne de çalıştı. Bazı çözümlerde de yordam `AgirlikBul(…);` biçiminde çağrılmıştı. Yordam bir `int` döndürüyordu, ama dönen değer hiçbir değişkene atanmamıştı; değer hesaplanıp kayboluyordu. Derleyici böyle bir satırı hata saymaz. Hatayı ancak kodu okuyan biri görür.

Yazılım mühendisliği, bir çözümün yalnız doğru değil, aynı zamanda modüler, basit, okunabilir, yeniden kullanılabilir ve bakımı kolay olmasını ister. Bu nitelikler ders kitaplarında İngilizce adlarıyla listelenir. Tablo 18.7 en sık geçenleri Türkçe karşılıklarıyla verir.

Nitelik	İngilizcesi	Nitelik	İngilizcesi
doğruluk	*correctness*	güvenilirlik	*reliability*
kullanılabilirlik	*usability*	verimlilik	*efficiency*
bakım kolaylığı	*maintainability*	değiştirilebilirlik	*modifiability*
taşınabilirlik	*portability*	birlikte çalışabilirlik	*interoperability*
bütünlük	*integrity*	esneklik	*flexibility*
sınanabilirlik	*testability*	izlenebilirlik	*traceability*
anlaşılabilirlik	*understandability*	modülerlik	*modularity*
genişletilebilirlik	*expandability*		

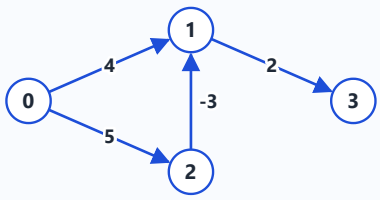
Tablo 18.7: Yazılım kalitesinin bazı nitelikleri.

Tablodaki niteliklerin çoğu küçük yordamlarla doğrudan ilgilidir. `minimumBul` ayrı bir yordam olduğu için heap sürümüne geçerken yalnız onun değiştirilmesi yeter: değiştirilebilirlik. `yol_yaz` ayrı olduğu için kendi başına küçük bir öncül dizisiyle sınanabilir: sınanabilirlik. Fonksiyonların dönüş değerlerinin kullanılması, `NULL` denetimleri ve açıklama satırları da güvenilirliğin ve anlaşılabilirliğin parçasıdır.

18.5 Negatif kenarlar ve tüm çiftler

18.5.1 Dijkstra negatif kenarda neden yanılır?

Gerçek problemlerde kenar ağırlığı her zaman bir uzunluk değildir. Bir kenar kazancı, bir enerji geri kazanımını ya da bir indirimi gösterebilir; bu durumda ağırlık negatif olur. Dijkstra'nın doğruluk kanıtı negatif olmayan ağırlıklara dayanıyordu. Şekil 18.7'deki dört düğümlü çizge bu varsayım bozulunca ne olduğunu gösterir. Kenarlar 0 → 1 (4), 0 → 2 (5), 1 → 3 (2) ve 2 → 1 (−3)'tür.



Şekil 18.7: Negatif kenarlı küçük çizge. 2 → 1 kenarının ağırlığı −3'tür; 0 → 2 → 1 yolu (5 − 3 = 2) doğrudan 0 → 1 kenarından (4) kısadır.

Dijkstra'nın izlemesi programla yapıldı. Başta uzaklıklar 1 için 4, 2 için 5, 3 için ∞'dur. İlk turda en küçük değerli 1 seçilir (4) ve kesinleşir; 1 üzerinden 3'ün değeri 6 olur. İkinci turda 2 seçilir (5). 2 → 1 kenarı 5 − 3 = 2 değerinde bir yol açar, ama 1 artık işaretlidir; gevşetme yapılmaz. Sonuç Tablo 18.8'dedir: 1 ve 3 yanlıştır.

Düğüm	Dijkstra	Doğru	Doğru yol
0	0	0	—
1	4	2	0 → 2 → 1

Düğüm	Dijkstra	Doğru	Doğru yol
2	5	5	0 → 2
3	6	4	0 → 2 → 1 → 3

Tablo 18.8: Negatif kenarlı çizgede Dijkstra'nın sonucu ve doğru uzaklıklar.

Hatanın kaynağı açgözlü seçimin varsayımıdır. Dijkstra, seçtiği düğüme daha sonra daha ucuz bir yoldan gelinemeyeceğini varsayar. Ağırlıklar negatif değilse bu doğrudur: henüz seçilmemiş düğümler daha uzaktadır ve onlardan geçen yol ancak uzayabilir. Negatif bir kenar ise uzaktaki bir düğümden geri dönüp yolu kısaltabilir. 1'in yanlış değeri 3'e de geçer, çünkü 3'ün yolu 1'den geçer. Ağırlıkların hepsine aynı sabit eklenerek negatiflikten kurtulmak da çözüm değildir. Bu dönüşüm çok kenarlı yolları tek kenarlılardan daha fazla cezalandırır ve en kısa yolun kendisini değiştirir.

Negatif kenarlarda bir tehlike daha vardır: **negatif döngü** (*negative cycle*). Ağırlıklarının toplamı negatif olan bir döngü kaynaktan erişilebilir durumdaysa, döngüde her dönüş yolu biraz daha kısaltır. Bu durumda en kısa yol tanımsızdır; "en küçük" uzunluk yoktur, değer sınırsız azalır. İyi bir algoritma böyle bir döngüyü bulup bildirebilmelidir.

18.5.2 Bellman-Ford algoritması

Richard Bellman ve Lester Ford'un adıyla anılan algoritma açgözlü değildir; hiçbir düğümü erken kesinleştirmez. Bunun yerine bütün kenarları defalarca gevşetir. Algoritma kenarları bir **kenar listesinde** tutar. Her öğe kenarın iki ucunu ve ağırlığını taşır:

```
typedef struct {
    int u, v, w;        /* kenar <u,v>, agirlik w */
} kenar;
int BellmanFord(kenar K[], int n, int e, int s, int d[])
{
    int i, j, t, u, v;
    for (i = 0; i < n; i++) d[i] = SONSUZ;
    d[s] = 0;
    for (t = 1; t <= n - 1; t++)          /* n-1 tur */
        for (j = 0; j < e; j++) {
            u = K[j].u;  v = K[j].v;
            if (d[u] != SONSUZ && d[u] + K[j].w < d[v])
                d[v] = d[u] + K[j].w;
        }
    for (j = 0; j < e; j++)               /* n. tur */
        if (d[K[j].u] != SONSUZ &&
            d[K[j].u] + K[j].w < d[K[j].v]) return -1;
    return 0;              /* -1: negatif dongu var */
}
```

Kodun gövdesi sunumdaki parçanın aynısıdır; çevresine yalnız yordam başlığı ve yerel değişkenler eklenmiştir. `K` kenar listesi, `n` düğüm sayısı, `e` kenar sayısı, `s` kaynak, `d` de sonuç dizisidir.

- **Satır 7–8.** Bütün uzaklıklar sonsuz, kaynağınki 0 yapılır. Başlangıçta yalnız kaynağa ulaşılabildiği bilinir.
- **Satır 9–14.** Ana döngü $n - 1$ tur döner. Her turda bütün kenarlar sırayla gevşetilir. Koşulun ilk parçası `d[u] != SONSUZ`, henüz ulaşılmamış bir düğümden yapılan gevşetmeyi engeller. Bu denetim negatif kenarlar yüzünden zorunludur: `SONSUZ + (-3)` sonsuzdan küçüktür ve denetim olmasaydı ulaşılmamış bir düğüm "ulaşılmış" görünürdü.
- **Satır 15–17.** n. tur: kenarlar bir kez daha denenir, ama bu kez değer değiştirilmez. Bir kenar hâlâ gevşeyebiliyorsa negatif bir döngü vardır ve yordam −1 döndürür.
- **Satır 18.** Döngü yoksa 0 döner; `d` dizisi en kısa uzaklıkları taşır.

Neden tam $n - 1$ tur? Negatif döngü yoksa her en kısa yol basit seçilebilir ve basit bir yolda en çok $n - 1$ kenar vardır. Gevşetmenin yaptığı iş bir yolu kenar kenar ilerletmektir.

Teorem 18.3 (Bellman-Ford'un tur sayısı)

Kaynaktan erişilebilen negatif döngü yoksa, t tur sonunda en çok t kenarlı bir en kısa yolu olan her düğümün `d` değeri doğrudur. Bu yüzden $n - 1$ tur bütün düğümler için yeterlidir.

İspat. Tümevarım t üzerinden yapılır. $t = 0$ için tek düğüm kaynaktır ve $d[s] = 0$ doğrudur. h'nin en kısa yolu t kenarlı ve son kenarı $\langle u, h\rangle$ olsun. Yolun u'ya kadarki parçası $t - 1$ kenarlı bir en kısa yoldur. Tümevarım varsayımına göre $t - 1$ tur sonunda $d[u]$ doğrudur. t. turda $\langle u, h\rangle$ kenarı gevşetilince $d[h] \le d[u] + w$ olur, bu da en kısa uzunluktur. $d[h]$ her zaman gerçek bir yolun uzunluğu olduğu için bundan küçük olamaz. ∎

Teoremin tersi döngü testini açıklar. Negatif döngü yoksa $n - 1$ turdan sonra bütün değerler kesindir ve hiçbir kenar gevşeyemez. n. turda bir kenar hâlâ gevşiyorsa varsayım bozulmuştur: erişilebilir bir negatif döngü vardır. Program bu durumu $1 \to 2 \to 1$ döngüsünün toplamı -1 olan bir çizgede denedi; yordam -1 döndürdü.

Örnek çizgedeki izleme Tablo 18.9'dadır. Kenar listesinin sırası $0 \to 1$, $0 \to 2$, $1 \to 3$, $2 \to 1$'dir; sonuç bu sıraya bağlı değildir, ama ara değerler bağlıdır.

Tur	Kenar	Hesap	`d[0..3]`
—	—	başlangıç	0 ∞ ∞ ∞
1	$0 \to 1$ (4)	$0 + 4 = 4 < \infty$	0 4 ∞ ∞
1	$0 \to 2$ (5)	$0 + 5 = 5 < \infty$	0 4 5 ∞
1	$1 \to 3$ (2)	$4 + 2 = 6 < \infty$	0 4 5 6
1	$2 \to 1$ (−3)	$5 + (-3) = 2 < 4$	0 **2** 5 6
2	$0 \to 1$, $0 \to 2$	$4 \ge 2$, $5 \ge 5$: değişmez	0 2 5 6
2	$1 \to 3$ (2)	$2 + 2 = 4 < 6$	0 2 5 **4**
2	$2 \to 1$ (−3)	$5 + (-3) = 2 \ge 2$: değişmez	0 2 5 4
3	hepsi	hiçbir kenar gevşemez	0 2 5 4
n.	hepsi	gevşeme yok → `return 0`	0 2 5 4

Tablo 18.9: Bellman-Ford izlemesi (kaynak 0, $n = 4$, $e = 4$).

İkinci turdaki gevşeme önemlidir. 3'ün değeri ilk turda 1'in eski değeri (4) üzerinden 6 olmuştu. 1'in değeri aynı turda 2'ye inince, 3'ün doğru değeri ancak bir sonraki turda görülebildi. Kenarların sırası buna yol açar: $1 \to 3$ kenarı listede $2 \to 1$'den önce gelir. Sıra farklı olsaydı doğru değerler daha erken bulunabilirdi. Algoritmanın garantisi, sıra ne olursa olsun $n - 1$ turun yettiğidir. Şekil 18.8 izlemenin son durumunu gösterir.



Şekil 18.8: Bellman-Ford bittiğinde: `d = [0, 2, 5, 4]`. Sağda kenar listesi gevşetme sırasıyla yazılmıştır.

Maliyet açıktır: $n - 1$ turun her birinde e kenar denenir, son tur da e kenar dener. Toplam $O(n \cdot e)$'dir. Ek bellek `d` dizisidir: $O(n)$. Yoğun bir çizgede $e \approx n^2$ olduğu için süre $O(n^3)$'e çıkar. Bellman-Ford bu yüzden Dijkstra'dan yavaştır; yalnız negatif ağırlıklar varken ya da negatif döngü aranırken tercih edilir. Pratikte bir turda hiçbir değer değişmezse döngüden erken çıkmak da sık kullanılan bir iyileştirmedir. Örnekte üçüncü tur hiçbir şey değiştirmemiştir; bu iyileştirmeyle algoritma orada durabilirdi.

18.5.3 Tüm çiftler: Floyd-Warshall algoritması

Bazen tek bir kaynak yetmez; bir yol haritasındaki uzaklık tablosu gibi her düğüm çifti için en kısa uzunluk istenir. Bu, **tüm çiftler arası en kısa yollar** (*all-pairs shortest paths*) problemidir. Dijkstra'yı her düğümden bir kez çalıştırmak bir çözümdür, ama negatif kenar varsa işe yaramaz. Robert Floyd ve Stephen Warshall'ın adıyla anılan algoritma problemi çok kısa bir kodla çözer.

Algoritmanın fikri, yollarda kullanılabilecek ara düğümleri birer birer serbest bırakmaktır.

Tanım 18.4

$A^k[i][j]$, i'den j'ye giden ve ara düğümlerinin hepsi $0, 1, \ldots, k$ arasında olan yolların en kısasının uzunluğudur. A^{-1} hiç ara düğüm kullanılamayan durumdur; maliyet matrisinin kendisidir.

A^k ile A^{k-1} arasındaki bağ iki durumla kurulur. i'den j'ye en kısa yol k'den geçmiyorsa ara düğümleri $k-1$'e kadardır ve uzunluğu $A^{k-1}[i][j]$'dir. k'den geçiyorsa yol iki parçaya ayrılır: i'den k'ye ve k'den j'ye. Negatif döngü olmadığı için iki parça da k'yi ara düğüm olarak kullanmaz ve uzunlukları A^{k-1}'de bulunur. Böylece

$$A^k[i][j] = \min \{ A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j] \}$$

bağıntısı elde edilir. Bu bağıntıyı $k = 0, 1, \ldots, n-1$ için sırayla uygulamak bir **dinamik programlama** çözümüdür: büyük problem, sonuçları bir tabloda saklanan küçük alt problemlerden kurulur. A^{n-1} bütün düğümlerin ara düğüm olabildiği durumdur ve sonucun kendisidir.

```
void Floyd(int maliyet[][MAX_DUGUM], int A[][MAX_DUGUM], int n)
{
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            A[i][j] = maliyet[i][j];
    for (k = 0; k < n; k++)              /* ara dugum k */
        for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                if (A[i][k] + A[k][j] < A[i][j])
                    A[i][j] = A[i][k] + A[k][j];
}
```

- **Satır 4–6.** `A` matrisine maliyet matrisi kopyalanır; bu A^{-1}'dir.
- **Satır 7.** Dış döngü ara düğümü seçer. Döngülerin sırası önemlidir: `k` en dışta olmalıdır. Her `k` turu, bir önceki turun bütün matrisini kullanır.
- **Satır 8–11.** Her `(i, j)` çifti için `k` üzerinden geçen yol denenir; kısaysa yazılır.

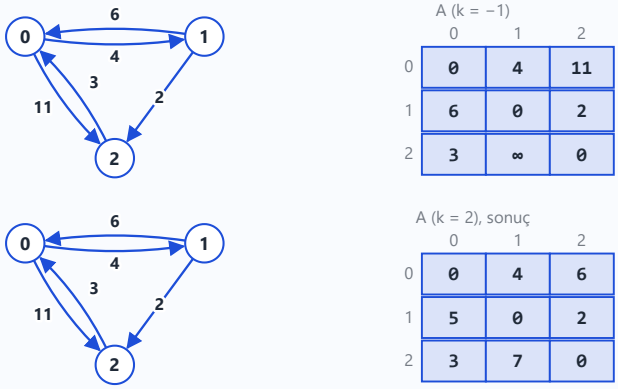
Bağıntı A^{k-1} ve A^k diye iki ayrı matrisle yazılmıştır, ama kod tek bir `A` matrisini yerinde günceller. Bu güvenlidir. k. turda `A[i][k]` ve `A[k][j]` değişmez, çünkü `A[i][k] + A[k][k]` ifadesinde `A[k][k] = 0`'dır ve toplam `A[i][k]`'dan küçük olamaz. Tur boyunca okunan değerler bu yüzden A^{k-1}'deki değerlerle aynıdır.

Algoritma Horowitz'in üç düğümlü örneğinde izlenir. Maliyet matrisinde $0 \to 1 = 4$, $0 \to 2 = 11$, $1 \to 0 = 6$, $1 \to 2 = 2$, $2 \to 0 = 3$'tür ve 2'den 1'e doğrudan kenar yoktur. Program her `k` turunda değişen hücreleri yazdırdı; her turda yalnız bir hücre değişti (Tablo 18.10).

Matris	Değişen hücre	Satır 0	Satır 1	Satır 2
A^{-1}	—	0 4 11	6 0 2	3 ∞ 0
A^0	`A[2][1]` : 3 + 4 = 7 < ∞	0 4 11	6 0 2	3 **7** 0
A^1	`A[0][2]` : 4 + 2 = 6 < 11	0 4 **6**	6 0 2	3 7 0
A^2	`A[1][0]` : 2 + 3 = 5 < 6	0 4 6	**5** 0 2	3 7 0

Tablo 18.10: Floyd-Warshall izlemesi (üç düğüm). Her satır bir `k` turunun sonundaki matrisi verir.

Her değişiklik bir yolun keşfidir. $k = 0$'da 2'den 1'e 0 üzerinden bir yol bulunur: $2 \to 0 \to 1$. $k = 1$'de 0'dan 2'ye 1 üzerinden geçmek doğrudan kenardan ucuzdur. $k = 2$'de 1'den 0'a 2 üzerinden giden yol (2 + 3 = 5) doğrudan kenardan (6) kısadır. Bu değişikliğin ancak $k = 2$ turunda olabilmesinin nedeni basittir: yol 2'yi ara düğüm olarak kullanır ve 2'nin kullanılması bu turda serbest kalır. Şekil 18.9 başlangıç ve bitiş matrislerini gösterir.



Şekil 18.9: Floyd-Warshall'ın başı ve sonu. Üstte A^{-1}, yani maliyet matrisi; altta tüm çiftler arası en kısa uzunluklar.

Analiz kısadır: üç iç içe döngünün her biri n tur döner. Süre $\Theta(n^3)$'tür ve çizgenin kenar sayısına bağlı değildir. Bellek $n \times n$ matristir. Aynı yordam altı düğümlü ana örneğe de uygulandı. Matrisin 0. satırı `0 45 10 25 45 1000` çıktı; Dijkstra'nın sonucuyla aynıdır. 5. satır `58 23 38 3 33 0` 'dır: 5'ten çıkan tek kenar 3'e gittiği için 5'in bütün yolları 3'ten geçer. 5. sütundaki 1000'ler ise hiçbir düğümden 5'e gidilemediğini gösterir.

Floyd-Warshall negatif kenarlara izin verir, negatif döngüye vermez. Negatif döngü varsa çalıştırma sonunda köşegende negatif bir değer belirir: bir düğümden kendine negatif uzunlukta bir yol bulunmuştur. Bu, döngünün varlığını sınamanın basit bir yoludur.

> **Sık yapılan hata**
>
> Negatif kenarlı bir çizgede `SONSUZ` 'u sıradan bir sayı gibi toplamak. Kodda `A[i][k] + A[k][j]` toplamı denetimsiz yapılır. `A[i][k]` sonsuz (1000), `A[k][j]` ise −5 olsun. Toplam 995'tir ve 1000'den küçüktür; olmayan bir yol "bulunur". Program bu durumu üç düğümlü bir çizgede sınadı: 0'dan 2'ye hiç yol yokken `A[0][2] = 995` çıktı. Negatif kenar varsa koşul `A[i][k] != SONSUZ && A[k][j] != SONSUZ && …` biçiminde yazılmalıdır. Bellman-Ford kodu bu denetimi zaten yapar.

18.5.4 Hangi algoritma, ne zaman?

En kısa yol için tek bir "en iyi" algoritma yoktur. Seçim problemin biçimine ve çizgenin yapısına bağlıdır. Tablo 18.11 karar için bir özet verir.

Problem	Algoritma	Süre	Koşul
Ağırlıksız, tek kaynak	genişlik öncelikli arama	$O(n + e)$	bütün kenarlar eşit
Tek kaynak, $w \geq 0$	Dijkstra (matris)	$O(n^2)$	yoğun çizge
Tek kaynak, $w \geq 0$	Dijkstra (heap + liste)	$O((n + e) \log n)$	seyrek çizge
Tek kaynak, negatif kenar	Bellman-Ford	$O(n \cdot e)$	negatif döngüyü yakalar
Yönlü döngüsüz çizge, tek kaynak	topolojik sırada gevşetme	$O(n + e)$	döngü yok
Tüm çiftler	Floyd-Warshall	$O(n^3)$	negatif döngü yok
Tüm çiftler, seyrek	n kez Dijkstra	$O(n\,(n + e) \log n)$	$w \geq 0$

Tablo 18.11: En kısa yol problemlerinin türleri ve uygun algoritmalar.

İlk satır bir hatırlatmadır: bütün kenarların ağırlığı eşitse en kısa yol, en az kenarlı yoldur. Genişlik öncelikli arama düğümleri kaynaktan kenar sayısı sırasıyla ziyaret ettiği için bu yolu doğrusal zamanda bulur. Beşinci satır ise notun ikinci yarısının konusudur. Çizgede döngü yoksa düğümler, her kenarın başlangıcı bitişinden önce gelecek biçimde sıralanabilir. Bu sırada her düğümün kenarları bir kez gevşetilirse, bir düğüme gelindiğinde onun bütün öncülleri zaten kesinleşmiştir. Kenar ağırlıkları negatif olsa bile tek geçiş yeter. Aktivite ağlarındaki zaman hesapları tam olarak bu fikre dayanır.

18.6 Aktivite ağları ve topolojik sıra

18.6.1 Proje ve aktiviteler

Büyük bir iş, daha küçük işlerin toplamıdır. Bir binanın inşaatı temel atmakla başlar; temel donduktan belli bir süre sonra ilk kat, ardından ikinci kat dökülür. Elektrik tesisatı duvarlar örülmeden çekilemez, ama boya ile bahçe düzenlemesi aynı anda yürüyebilir. Bir üniversite programı da böyledir: mezuniyet, derslerin her birinin geçilmesiyle gelir ve bazı derslere ancak başka dersler alındıktan sonra girilebilir. Bu küçük işlere **aktivite** (*activity*), aktivitelerin bütününe **proje** denir. Proje, aktivitelerinin hepsi bitince biter.

Proje yöneticisinin soruları bellidir. İşler hangi sırayla yapılabilir? Proje en erken ne zaman biter? Hangi işler kritiktir, yani bir gün gecikirse bütün proje bir gün gecikir? Hangi işlerde bolluk vardır, yani biraz gecikse de bir şey olmaz? Bu soruları cevaplamak için aktiviteler ile aralarındaki önce-sonra ilişkileri bir çizgeyle gösterilir. Bu çizgeye **aktivite ağı** (*activity network*) denir.

Tablo 18.12 bir bilgisayar bilimi programının on beş dersini ve ön koşullarını verir. Örnek, Horowitz ve Sahni'nin kitabından alınmıştır; ders adları da orada olduğu gibi İngilizce bırakılmıştır.

Kod	Ders	Ön koşul	Kod	Ders	Ön koşul
C1	*Programming I*	—	C9	*Operating Systems*	C7, C8
C2	*Discrete Mathematics*	—	C10	*Programming Languages*	C7
C3	*Data Structures*	C1, C2	C11	*Compiler Design*	C10
C4	*Calculus I*	—	C12	*Artificial Intelligence*	C7
C5	*Calculus II*	C4	C13	*Computational Theory*	C7
C6	*Linear Algebra*	C5	C14	*Parallel Algorithms*	C13
C7	*Analysis of Algorithms*	C3, C6	C15	*Numerical Analysis*	—
C8	*Assembly Language*	C3			

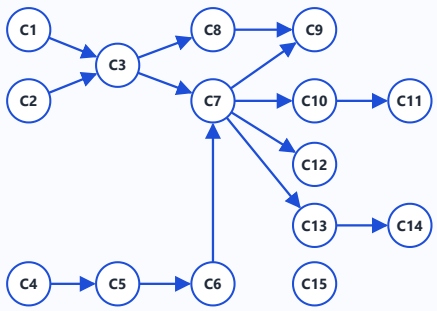
Tablo 18.12: Bir bilgisayar bilimi programının dersleri ve ön koşulları.

18.6.2 Düğümde aktivite ağları ve öncül ilişkisi

Tablodaki bilgiyi çizgeye çevirmenin doğal yolu, her dersi bir düğüm yapmaktır. *i* dersi *j*'nin ön koşuluysa *i*'den *j*'ye yönlü bir kenar çizilir. On beş düğümlü ve on dört kenarlı çizge Şekil 18.10'dadır.

Tanım 18.5

Düğümleri aktiviteleri, yönlü kenarları aktiviteler arasındaki **öncül ilişkisini** (*precedence relation*) gösteren çizgeye **düğümde aktivite ağı** (*activity on vertex network*, kısaca AoV ağı) denir. *i* aktivitesi *j* başlamadan bitmek zorundaysa ağda $\langle i, j \rangle$ kenarı bulunur.



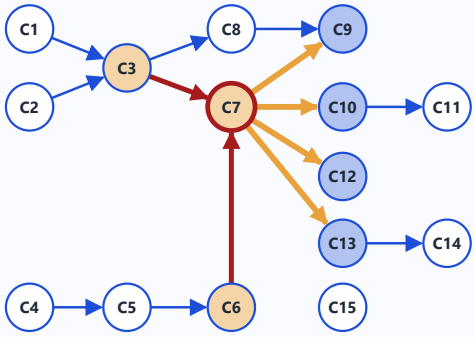
Şekil 18.10: Derslerin düğümde aktivite ağı. Her ok bir ön koşulu gösterir: C1 ve C2, C3'ün; C3 ve C6, C7'nin ön koşuludur. C15'in ne ön koşulu ne de ardılı vardır.

Ağdaki ilişkileri adlandırmak için birkaç terim gerekir. Bunlar aile ilişkilerine benzer, ama yön kenarlarla belirlenir.

Tanım 18.6

AoV ağında *i*'den *j*'ye yönlü bir yol varsa *i*, *j*'nin **öncülü** (*predecessor*), *j* de *i*'nin **ardılıdır** (*successor*). Yol tek kenardan oluşuyorsa, yani <*i*, *j*> bir kenarsa, *i* *j*'nin **yakın öncülü** (*immediate predecessor*), *j* de *i*'nin **yakın ardılıdır** (*immediate successor*).

Ders ağında C3 ve C6, C7'nin yakın öncülleridir. C9, C10, C12 ve C13 C7'nin yakın ardıllarıdır. C14 ise C3'ün ardılıdır, ama yakın ardılı değildir: C3'ten C14'e C3 → C7 → C13 → C14 yoluyla varılır. Şekil 18.11 C7'nin yakın öncül ve ardıllarını vurgular. Öncül ilişkisinin iki temel özelliği vardır.



Şekil 18.11: C7 ve çevresi. Yakın öncüllerden (C3, C6) gelen kenarlar kalın kırmızı, yakın ardıllara (C9, C10, C12, C13) giden kenarlar turuncu çizilmiştir.

- **Geçişlilik.** *i*, *j*'nin; *j* de *k*'nin öncülüyse *i*, *k*'nin öncülüdür. C1 C3'ün, C3 C7'nin ön koşuluysa C1 de dolaylı olarak C7'nin ön koşuludur.
- **Döngüsüzlük.** Hiçbir aktivite kendisinin öncülü olamaz. A, B'nin öncülüyse B, A'nın öncülü olamaz: A bitmeden B başlayamaz, B bitmeden A başlayamaz ve proje hiç bitmez. Bu, çizgede yönlü döngü bulunmaması demektir.

İkinci özellik yalnız kuramsal bir koşul değildir; gerçek projelerde planlama hatası olarak sık görülür. Yönlü döngü içermeyen yönlü çizgeye **yönlü döngüsüz çizge** (*directed acyclic graph*, DAG) denir. Geçerli bir AoV ağı her zaman bir DAG'dir. Bu yüzden ağ kurulurken döngü bulunup bulunmadığı denetlenmelidir.

Ders örneğinin sayısal analiz ve doğrusal cebir gibi derslerle örülmüş olması bir rastlantı değildir. Proje planlaması, kaynak dağıtımı ve en kısa yol gibi konuların kökü yöneylem araştırmasına uzanır. Yöneylem ve sayısal analiz derslerinin müfredatlardan çıkarılması, öğrencilerin bu tür problemlerin matematiğinde zayıf kalmasına yol açar.

18.6.3 Topolojik sıra

Ders ağı bir öğrencinin önüne pratik bir soru koyar: dersler hangi sırayla alınabilir? Sıra, her dersin ön koşullarının kendisinden önce geldiği bir sıra olmalıdır.

Tanım 18.7

Bir AoV ağının düğümlerinin, *i* *j*'nin öncülü olduğunda *i*'nin sırada *j*'den önce geldiği doğrusal bir sıralamasına **topolojik sıra** (*topological order*) denir.

Tanımın üç sonucu vardır. Birincisi, topolojik sıra çoğu zaman tek değildir. Aralarında öncül ilişkisi olmayan iki aktivite herhangi bir sırada yazılabilir. İkincisi, çizgede yönlü döngü varsa topolojik sıra yoktur. Döngüdeki düğümlerden hangisi önce yazılırsa yazılsın, döngüdeki bir kenar ters yöne bakar. Üçüncüsü, döngü yoksa en az bir topolojik sıra vardır; bunun kanıtı aşağıdaki algoritmadır.

Ders ağı için iki geçerli sıra şunlardır:

- C1, C2, C4, C5, C3, C6, C8, C7, C10, C13, C12, C14, C15, C11, C9
- C4, C5, C2, C1, C6, C3, C8, C15, C7, C9, C10, C11, C12, C13, C14

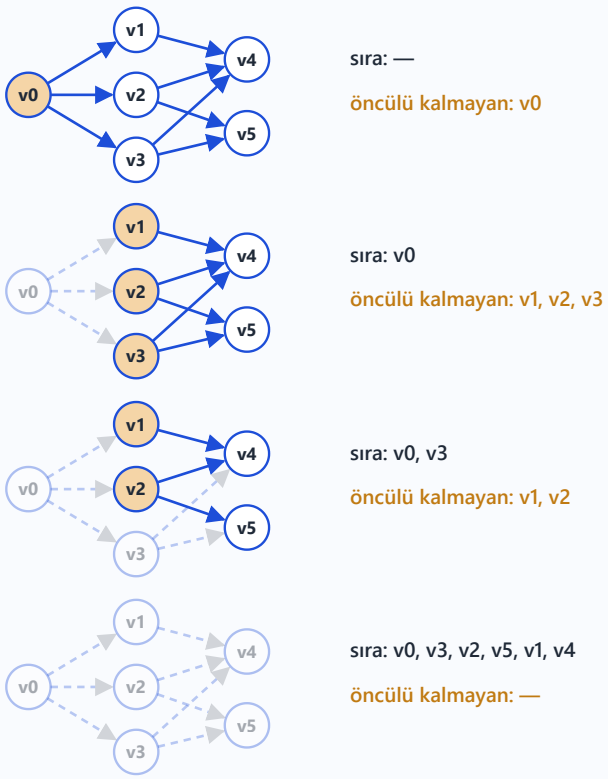
Bir sıranın topolojik olup olmadığı kolayca denetlenir. Her düğümün sıradaki yeri bulunur. Sonra her <*i*, *j*> kenarı için *i*'nin yerinin *j*'ninkinden küçük olduğu denetlenir. Ders ağında 14 kenar vardır ve iki sıra da 14 kenarın hepsini sağlar; bu denetim bir programla yapıldı. Aynı ağa aşağıda yazılacak `TopolojikSira` yordamı uygulanınca üçüncü bir geçerli sıra çıktı: C15, C4, C5, C6, C2, C1, C3, C8, C7, C13, C14, C12, C10, C11, C9. Üç sıranın üçü de geçerlidir; hangisinin çıkacağı yalnız algoritmanın düğümleri seçme biçimine bağlıdır.

Topolojik sıranın kullanım alanı geniştir. Ders planlaması bunlardan yalnız biridir. Bir yazılım projesinde kaynak dosyaların derlenme sırası (`make` aracının yaptığı iş), bir işletim sisteminde görevlerin zamanlanması ve bir hesap tablosunda hücrelerin yeniden hesaplanma sırası da topolojik sıradır. Hesap tablosunda bir hücrenin formülü başka hücrelere başvuruyorsa o hücreler önce hesaplanmalıdır; iki hücre birbirine başvuruyorsa program "döngüsel başvuru" hatası verir. Bu, AoV ağındaki döngünün bir başka yüzüdür. Aktivite ağlarındaki en erken zaman hesabı da topolojik sırada yapılır.

18.6.4 Öncülsüz düğümü silme yöntemi

Topolojik sırayı bulmanın yolu basit bir gözleme dayanır: sıranın ilk düğümünün hiç öncülü olamaz. Öyleyse öncülü olmayan bir düğüm seçilip yazılır. Sonra bu düğüm ve ondan çıkan kenarlar çizgeden silinir. Kalan çizge de bir AoV ağıdır ve aynı işlem onda tekrarlanır. Bütün düğümler yazılınca sıra tamamlanmış olur.

Yöntem altı düğümlü bir ağda izlenir (Şekil 18.12). Kenarlar *<0,1>, <0,2>, <0,3>, <1,4>, <2,4>, <2,5>, <3,4>, <3,5>*'tir.



Şekil 18.12: Öncülsüz düğümü silme yöntemi, yukarıdan aşağıya: (a) yalnız v0 öncülsüzdür; (b) v0 yazılıp kenarları silinince v1, v2, v3 öncülsüz kalır; (c) v3 seçilmiş, kenarları silinmiştir; (g) bütün düğümler yazılmıştır. Turuncu düğümler o an seçilebilecek öncülsüz düğümlerdir; sağdaki satırlar o ana kadarki sırayı ve öncülü kalmayan düğümleri yazar.

Başta yalnız v0'ın öncülü yoktur ve ilk o yazılır. v0'ın üç kenarı silinince v1, v2 ve v3'ün öncülü kalmaz. Üçünden herhangi biri seçilebilir; örnekte v3 seçilmiştir. v3'ün kenarları *<3,4>* ve *<3,5>* silinir. Sonra v2, ardından v5, v1 ve en son v4 yazılır. Sonuç v0, v3, v2, v5, v1, v4'tür. Programla sayıldığında bu küçük ağın 14 farklı topolojik sırası olduğu görüldü; seçim her adımda farklı yapılarak hepsine ulaşılabilir.

Yöntem döngüyü de bulur. Bir adımda henüz yazılmamış düğümler varken hiçbiri öncülsüz değilse, kalan çizgede her düğümün en az bir öncülü vardır. Öncüllerden geriye doğru yüründüğünde sonlu sayıda düğüm olduğu için bir düğüme ikinci kez gelinir: bu bir yönlü döngüdür. Tersi de doğrudur: döngü yoksa her adımda en az bir öncülsüz düğüm bulunur ve yöntem bütün düğümleri yazar. Böylece yöntem hem sırayı bulur hem de ağın geçerli olup olmadığını sınar.

Yöntemin verimli bir koda dönüşmesi için iki işlemin hızlı yapılması gerekir: öncülsüz düğümü bulmak ve bir düğümün çıkan kenarlarını silmek. Her adımda bütün düğümler taranıp öncülsüz olan aranırsa maliyet $O(n^2)$'ye çıkar. Daha iyisi, her düğümün kaç öncülü kaldığını bir sayaçta tutmak ve sayacı sıfıra düşen düğümleri ayrı bir yerde biriktirmektir.

18.6.5 Veri yapısı: sayaçlı başlıklar

Çizge komşuluk listesiyle tutulur. Her düğümün bir **başlık** öğesi vardır. Başlıkta listenin ilk öğesini gösteren bağın yanında bir de `sayac` alanı bulunur. Bu alan düğümün kaç yakın öncülü olduğunu, yani **giriş derecesini** (*in-degree*) tutar.

```
typedef struct oge *oge_gost;
typedef struct oge {
    int dugum;
    oge_gost bag;
} oge;
typedef struct {
    int sayac;      /* <i,j> icin: basi j olan kenar sayisi */
    oge_gost bag;
} basoge;
basoge baslik[MAX_DUGUM];
```

- **Satır 1.** `oge_gost`, bir liste öğesini gösteren işaretçi türüdür. Tür adı, yapı tanımlanmadan önce bildirilebilir; C buna eksik tür (*incomplete type*) üzerinden izin verir. Böylece yapının içinde kendi türünü gösteren bir alan yazılabilir.
- **Satır 2–5.** Liste öğesi: kenarın vardığı düğüm (`dugum`) ve listedeki bir sonraki öğe (`bag`).
- **Satır 6–9.** Başlık öğesi: öncül sayacı ve listenin başı. Başlıklar bir dizide durur; liste öğeleri ise `malloc` ile heap bellekten alınır.
- **Satır 10.** Bütün düğümlerin başlıkları. Küresel bir dizi olduğu için program başlarken bütün alanları sıfırdır: sayaçlar 0, bağlar `NULL` 'dur.

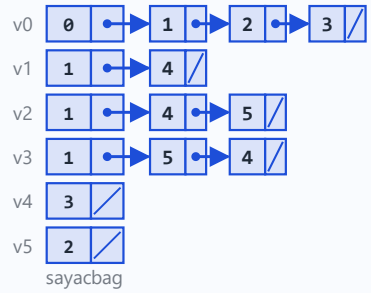
Kenarlar okunurken her *<i, j>* için *i*'nin listesine *j* eklenir ve `baslik[j].sayac` bir artırılır. Ders kitabı bu adımı ayrıca yazmaz; aşağıdaki yordam onu tamamlar. Öğeyi listenin sonuna ekler, böylece listeler kenarların veriliş sırasını korur.

```
int kenar_ekle(basoge baslik[], int i, int j)
{
    oge_gost *son = &baslik[i].bag;   /* bag alaninin adresi */
    oge_gost yeni = malloc(sizeof(oge));
    if (yeni == NULL)
        return 0;                     /* bellek yetmedi */
    yeni->dugum = j;
    yeni->bag = NULL;
    while (*son != NULL)              /* listenin sonunu bul */
        son = &(*son)->bag;
    *son = yeni;
    baslik[j].sayac++;                /* j'nin bir onculu daha var */
    return 1;
}
```

Yordamın ilginç yanı `son` değişkenidir. `son` bir öğeyi değil, bir *bağ alanını* gösterir; türü işaretçinin işaretçisidir (`oge_gost *`). Başta `baslik[i].bag` alanının adresini tutar. Liste boşsa bu alan `NULL` 'dur ve yeni öğe doğrudan oraya yazılır. Liste boş değilse döngü her adımda bir sonraki öğenin `bag` alanına geçer ve `NULL` olan son bağda durur. `*son = yeni` ataması, boş liste ile dolu listeyi aynı satırla karşılar. İşaretçinin işaretçisi kullanılmasaydı boş liste için ayrı bir `if` yazmak gerekirdi.

`malloc` 'un dönüşü her zaman denetlenmelidir. Bellek yetmezse `malloc` `NULL` döndürür ve `yeni->dugum` ataması tanımsız davranışa yol açar. Yordam bu durumda 0 döndürerek çağırana haber verir. Ekleme listenin sonuna yapıldığı için bir düğümün çıkış derecesi *d* ise maliyet *O(d)*'dir. Başa ekleme *O(1)* sürerdi, ama listeler kenarların tersi sırasında olurdu; bu da topolojik sıranın çıktısını değiştirirdi.

Altı düğümlü ağın kenarları *<0,1>, <0,2>, <0,3>, <1,4>, <2,4>, <2,5>, <3,5>, <3,4>* sırasıyla eklenince Şekil 18.13'teki yapı oluşur. v3'ün listesinde v5'in v4'ten önce gelmesi, kenarların bu sırayla verilmesindendir. Sayaçlar 0, 1, 1, 1, 3, 2'dir: v4'ün üç öncülü (v1, v2, v3), v5'in iki öncülü (v2, v3) vardır.



Şekil 18.13: Altı düğümlü ağın sayaçlı komşuluk listesi. Solda başlık dizisi (`sayac` ve `bag`), sağda her düğümün yakın ardılları.

Bellekte her liste öğesi bir `int` ve bir işaretçiden oluşur. 64 bitlik bir sistemde `int` 4 bayt, işaretçi 8 bayttır. İşaretçi 8'in katı bir adreste durmak zorunda olduğu için `int` 'ten sonra 4 baytlık bir dolgu eklenir ve `sizeof(oge)` 16 olur. Başlık öğesi de aynı nedenle 16 bayttır. Bu değerler zig cc derleyicisiyle x86-64'te ölçüldü. Altı düğümlü ve sekiz kenarlı ağ başlıklar için 96, liste öğeleri için 128 bayt kullanır. Toplam bellek $O(n + e)$'dir.

Listeler artık gerekmediğinde bellek geri verilmelidir. Aşağıdaki yordam her listeyi baştan sona gezer ve öğeleri tek tek bırakır:

```
void listeleri_birak(basoge baslik[], int n)
{
    int i;
    oge_gost p, sonraki;
    for (i = 0; i < n; i++) {
        for (p = baslik[i].bag; p; p = sonraki) {
            sonraki = p->bag;         /* free'den once sakla */
            free(p);
        }
        baslik[i].bag = NULL;
        baslik[i].sayac = 0;
    }
}
```

Döngüde `sonraki` değişkeni zorunludur. `free(p)` 'den sonra `p->bag` 'e erişmek, geri verilmiş belleği okumak demektir ve tanımsız davranıştır. Bu yüzden bir sonraki öğenin adresi `free` 'den önce saklanır. Sızıntı denetimi için testlerde `malloc` ve `free` , çağrıları sayan küçük sarmalayıcılarla değiştirildi. Notun topolojik sıra testlerinde 24 `malloc` ve 24 `free` sayıldı; bellek sızıntısı yoktur. Windows'ta zig derleyicisi adres denetleyicisini (`-fsanitize=address`) bağlayamadığı için bu sayaç yöntemi kullanıldı.

18.6.6 `TopolojikSira` yordamı

Yordam öncülsüz düğümleri bir yığında biriktirir. Yığın için ayrı bir dizi ayrılmaz; sayacı sıfıra düşmüş bir düğümün `sayac` alanı artık boştadır ve yığının bağı olarak kullanılır.

```
void TopolojikSira(basoge baslik[], int n)
{
    int i, j, k, ust = -1;
    oge_gost p;
    for (i = 0; i < n; i++)      /* sayac 0: yigina */
        if (!baslik[i].sayac) {
            baslik[i].sayac = ust;  ust = i; }
    for (i = 0; i < n; i++) {
        if (ust == -1) { printf("dongu"); break; }
        j = ust;  ust = baslik[ust].sayac;
        printf("%d ", j);
        for (p = baslik[j].bag; p; p = p->bag) {
            k = p->dugum;
            if (--baslik[k].sayac == 0) {
                baslik[k].sayac = ust;  ust = k; }
        }
    }
}
```

- **Satır 3.** `ust` yığının tepesindeki düğümdür. −1 yığının boş olduğunu, yani dibini gösterir.
- **Satır 5–7.** İlk döngü sayacı 0 olan bütün düğümleri yığına koyar. `baslik[i].sayac = ust` ataması, düğümün altında kalan düğümü onun `sayac` alanına yazar. Sonra `ust = i` ile yeni düğüm tepeye çıkar. Bu, bağlı listeyle gerçeklenmiş bir yığının başa ekleme işleminin aynısıdır; bağ alanı olarak `sayac` kullanılır.
- **Satır 8.** İkinci döngü *n* tur döner; her turda bir düğüm yazılır.
- **Satır 9.** Düğüm yazılacakken yığın boşsa öncülsüz düğüm kalmamıştır. Henüz *n* düğüm yazılmadığı için kalan düğümler bir döngü oluşturur. Yordam "dongu" yazar ve durur.
- **Satır 10–11.** Yığından çıkarma: tepedeki düğüm `j` 'ye alınır, altındaki düğüm (`baslik[ust].sayac`) yeni tepe olur. `j` yazılır.
- **Satır 12–16.** `j` 'nin listesi gezilir. Her yakın ardıl `k` 'nin öncül sayısı bir azaltılır; `j` 'nin silinmesi, `j` 'den çıkan kenarların silinmesi demektir. Sayaç 0'a düşerse `k` 'nin öncülü kalmamıştır ve yığına konur.

Gömülü yığın tekniği ilk bakışta şaşırtıcıdır. Kodda "yığın" diye bir değişken yoktur; yalnız açıklama satırı ve `ust` değişkeni ne olduğunu ele verir. Açıklama satırları tam da bu yüzden önemlidir: kodu okuyan biri yığının nerede ve nasıl kullanıldığını onlardan anlar. Teknik, kümelerde kullanılan ebeveyn dizisine benzer. Orada her öğe ebeveyninin numarasını tutar ve bağlar bir dizinin içine gömülüdür. Burada da yığının bağları başlık dizisinin içine gömülüdür. Bir alanın iki ayrı amaçla kullanılabilmesinin nedeni, iki amacın zamanda çakışmamasıdır. Sayaç 0'a düşene kadar alan öncül sayısını tutar; düştükten sonra bir daha öncül sayısına gerek kalmaz.

`--baslik[k].sayac == 0` ifadesi iki işi birleştirir: önce sayacı azaltır, sonra yeni değeri 0 ile karşılaştırır. Ders kitabındaki `sayac--; if (!sayac)` ikilisiyle aynı anlamdadır. Önek azaltma işleci burada önemlidir. Sonek kullanılsaydı (`baslik[k].sayac-- == 0`) karşılaştırma eski değerle yapılırdı. Son öncül silinirken eski değer 1 olduğu için hiçbir ardıl yığına girmez; örnekte çıktı `0 dongu` ile kesilirdi.

> **Sık yapılan hata**
>
> `TopolojikSira` 'yı aynı başlık dizisi üzerinde ikinci kez çağırmak. Yordam bittiğinde `sayac` alanları öncül sayılarını değil, yığın bağlarının kalıntılarını tutar. Örnekte yordamdan sonra sayaçlar −1, −1, 1, 2, −1, 1 olarak kaldı. Yeniden çalıştırmak için sayaçlar listelerden baştan hesaplanmalı ya da listeler yeniden kurulmalıdır.

18.6.7 Kod izlemesi

`TopolojikSira(baslik, 6)` çağrısının izlemesi Tablo 18.13'tedir. Yığın sütununda tepe soldadır. Program aynı ağda `0 3 2 5 1 4` yazdı.

Olay	`sayac[]`	Yığın	Çıktı
İlk döngü: v0'ın sayacı 0 → yığına	−1 1 1 1 3 2	0	
Yığından al: v0	−1 1 1 1 3 2	—	0
<0,1>: v1 0'a düştü → yığına	−1 **−1** 1 1 3 2	1	0
<0,2>: v2 0'a düştü → yığına	−1 −1 **1** 1 3 2	2 1	0
<0,3>: v3 0'a düştü → yığına	−1 −1 1 **2** 3 2	3 2 1	0
Yığından al: v3	−1 −1 1 2 3 2	2 1	0 3
<3,5>: v5'in sayacı 1; <3,4>: v4'ün sayacı 2	−1 −1 1 2 2 1	2 1	0 3
Yığından al: v2	−1 −1 1 2 2 1	1	0 3 2
<2,4>: v4'ün sayacı 1; <2,5>: v5 0'a düştü → yığına	−1 −1 1 2 1 **1**	5 1	0 3 2
Yığından al: v5 (listesi boş)	−1 −1 1 2 1 1	1	0 3 2 5
Yığından al: v1	−1 −1 1 2 1 1	—	0 3 2 5 1
<1,4>: v4 0'a düştü → yığına	−1 −1 1 2 **−1** 1	4	0 3 2 5 1
Yığından al: v4	−1 −1 1 2 −1 1	—	0 3 2 5 1 4

Tablo 18.13: `TopolojikSira(baslik, 6)` izlemesi. Sayaçlar v0…v5 sırasıyladır; yığına girmiş düğümün sayacı, altındaki düğümü gösterir.

Tablonun iki ayrıntısına dikkat edilmelidir. Yığına giren düğümün sayacına `ust` 'ün o anki değeri yazılır: v2 yığına girerken tepe v1 olduğu için `sayac[2] = 1` olur, v3 girerken tepe v2 olduğu için `sayac[3] = 2` olur. Ayrıca v0 yığından alındıktan sonra yığın boşalmış, ama hemen v0'ın ardılları ile dolmuştur. Döngü testi yalnız bir turun *başında* yapılır; tur içindeki geçici boşluk döngü sayılmaz.

Çıktı sırası şekildeki elle bulunan sırayla aynıdır. Elle yapılan seçimde v3'ün seçilmesi keyfiydi; kodda ise yığının son giren ilk çıkar kuralından gelir. v1, v2, v3 bu sırayla yığına girdiği için ilk çıkan v3 olur. Şekil 18.14 izlemenin iki anını gösterir.



Şekil 18.14: `TopolojikSira` izlemesinden iki an. Üstte v0'ın listesi bitmiştir: v1, v2, v3 yığındadır ve sayaç alanları yığının bağlarını tutar. Altta bütün düğümler yazılmış, bütün kenarlar silinmiştir.

18.6.8 Döngü tespiti, analiz ve kuyruklu sürüm

Döngü, iki aktivitenin karşılıklı olarak birbirinin öncülü olmasıdır ve bir planlama hatasıdır: döngü varsa proje hiç bitmez. Yordamın döngüyü nasıl yakaladığı küçük bir ağla sınandı. Kenarlar *<0,1>, <1,2>, <2,1>, <2,3>*'tür; 1 ile 2 birbirinin öncülüdür. Yordam önce 0'ı yazar. 1'in sayacı 2'den 1'e iner ve sıfıra düşmez, çünkü 2'den gelen kenar hâlâ vardır. İkinci turun başında yığın boştur, ama yalnız bir düğüm yazılmıştır. Çıktı `0 dongu` olur.

Yordamın maliyeti iki döngüden gelir. İlk döngü n başlığa bir kez bakar: $O(n)$. İkinci döngü en çok n tur döner. Her turda bir düğüm yığından alınır ve onun listesi gezilir. j'nin listesinde d_j öğe vardır, yani iç `for` çıkış derecesi kadar döner. Her düğüm yığından en çok bir kez alındığı için iç döngünün toplam tur sayısı bütün çıkış derecelerinin toplamıdır. Yönlü çizgede bu toplam kenar sayısına eşittir. Sonuç

$$O(n + d_0 + d_1 + \cdots + d_{n-1}) = O(n + e)$$

olur. Kenar sayısı çizgenin bağlanışına göre değişir. Seyrek bir ağda e n mertebesindedir ve süre doğrusaldır. Çok yoğun bir ağda e n^2 mertebesine çıkabilir. Kenarların her biri en az bir kez okunmak zorunda olduğu için hiçbir algoritma $O(n + e)$'den hızlı olamaz; bu yordam en iyisidir.

Öncülsüz düğümleri biriktirmek için yığın şart değildir; herhangi bir kap işe yarar. Kuyruk kullanılırsa algoritmaya Kahn'ın adı verilir (Arthur Kahn, 1962). Aşağıdaki sürüm kuyruğu basit bir diziyle gerçekler:

```
void KuyrukluSira(basoge baslik[], int n)
{
    int kuyruk[MAX_DUGUM], on = 0, arka = 0, i, j, k;
    oge_gost p;
    for (i = 0; i < n; i++)
        if (!baslik[i].sayac)
            kuyruk[arka++] = i;       /* kuyruga ekle */
    while (on < arka) {
        j = kuyruk[on++];             /* kuyruktan al */
        printf("%d ", j);
        for (p = baslik[j].bag; p; p = p->bag) {
            k = p->dugum;
            if (--baslik[k].sayac == 0)
                kuyruk[arka++] = k;
        }
    }
    if (arka < n)
        printf("dongu");
}
```

Bu sürümde döngü testi sonda yapılır: kuyruğa giren düğüm sayısı n'den azsa bazı düğümlerin sayacı hiç sıfıra düşmemiştir. Aynı altı düğümlü ağda kuyruklu sürüm `0 1 2 3 5 4` yazdı. v1, v2, v3 bu kez girdikleri sırayla çıkar. v3 işlenirken önce v5'in, sonra v4'ün sayacı sıfıra düşer, çünkü v3'ün listesinde v5 öndedir. İki çıktı farklıdır, ama ikisi de geçerli topolojik sıradır. Kabın türü yalnız eşit derecede geçerli sıralar arasında seçim yapar. Kuyruklu sürüm döngülü ağda da `0 dongu` yazdı ve iki sürümün bütün testlerinde bellek sızıntısı görülmedi.

18.7 Kenarda aktivite ağları ve kritik yol

18.7.1 Kenarda aktivite ağı

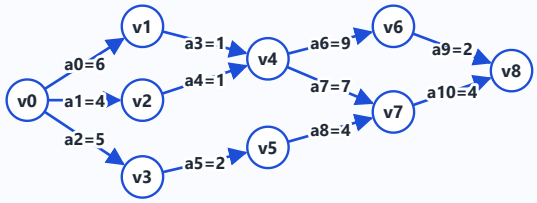
AoV ağı işlerin sırasını gösterir, ama süreleri göstermez. Proje süresini hesaplamak için ikinci bir model kullanılır. Bu modelde aktiviteler kenarlara, aktivitelerin bitişini belirleyen anlar da düğümlere konur.

Tanım 18.8

Yönlü kenarları aktiviteleri, kenar ağırlıkları aktivitelerin sürelerini, düğümleri de **olayları** (*event*) gösteren yönlü döngüsüz çizgeye **kenarda aktivite ağı** (*activity on edge network*, kısaca AoE ağı) denir. Bir olay, kendisine giren bütün aktivitelerin bittiği andır; olaydan çıkan aktiviteler ancak bu andan sonra başlayabilir.

Olay, projenin tamamlanması için ulaşılması gereken bir ara aşamadır. Bitmesi gereken iş bir düğümle ifade edilir: "temel bitti", "kaba inşaat bitti" gibi. Ağın tek bir başlangıç olayı (giren kenarı olmayan düğüm) ve tek bir bitiş olayı (çıkan kenarı olmayan düğüm) vardır. Başlangıç projenin başladığı, bitiş projenin bittiği andır.

Notun geri kalanında Şekil 18.15'teki ağ kullanılır. Ağda dokuz olay (v0–v8) ve on bir aktivite (a0–a10) vardır. v0 başlangıç, v8 bitiş olayıdır.



Şekil 18.15: Dokuz olaylı, on bir aktiviteli AoE ağı. Kenar etiketi aktivitenin adını ve süresini (gün) verir: a0 = 6, v0'dan v1'e giden 6 günlük aktivitedir.

Ağ paralelliği doğrudan gösterir. a0, a1 ve a2 aynı anda, projenin ilk günü başlayabilir; üçünün de tek koşulu başlangıç olayıdır. a6 ve a7 v4 olayından çıkar. v4'e de a3 ve a4 girer. Öyleyse a6 ve a7, a3 ile a4'ün ikisi de bitmeden başlayamaz. Benzer biçimde v7'ye a7 ve a8 girer; a10 ancak bu ikisi bitince başlar. Aktivite listesi Tablo 18.14'tedir.

Aktivite	a0	a1	a2	a3	a4	a5	a6	a7	a8	a9	a10
Kenar	0→1	0→2	0→3	1→4	2→4	3→5	4→6	4→7	5→7	6→8	7→8
Süre	6	4	5	1	1	2	9	7	4	2	4

Tablo 18.14: AoE ağının aktiviteleri.

18.7.2 AoV'den AoE'ye: yapay aktivite

Bir AoV ağı AoE ağına çevrilebilir, ama çeviri her zaman kenarları birebir eşlemekle olmaz. Şekil 18.16'daki AoV ağında 4 numaralı aktivite hem 2'yi hem 3'ü bekler. AoE ağında 2 ve 3 birer kenardır; 4'ün başlayabilmesi için iki kenarın da aynı olaya varması gerekir. İki kenar farklı olaylarda bitiyorsa bu olaylar, süresi 0 olan bir kenarla birleştirilir.

Tanım 18.9

AoE ağında yalnız bir öncül ilişkisini göstermek için eklenen, süresi 0 olan kenara **yapay aktivite** (*dummy activity*) denir.



Şekil 18.16: Solda AoV ağı: 4, hem 2'yi hem 3'ü bekler. Sağda aynı ilişki AoE ağında: kesikli kırmızı kenar süresi 0 olan yapay aktivitedir.

Yapay aktiviteye ana örnekte de gerek doğabilir. a6 ve a7'nin a3 ve a4'ün yanında a5 bittikten sonra başlaması istenseydi, a5'in bittiği v5 olayından a6 ile a7'nin başladığı v4 olayına bir kenar gerekirdi. Bu kenar bir iş değildir, yalnız bir bekleme kuralıdır: *<v5, v4>* yapay aktivitesi, süre 0.

Bir ağı en az sayıda yapay aktiviteyle çizmek kolay görünen ama zor bir problemdir. Bu problemin NP-zor olduğu gösterilmiştir; yani bilinen hiçbir algoritma onu büyük ağlarda verimli çözemez. Çizgelerin okunaklı çizimiyle uğraşan alan (*graph drawing*) için tez düzeyinde çalışmaların yapıldığı bir araştırma konusudur.

AoE ağları proje yönetiminin klasik yöntemlerinin temelidir. PERT (*Program Evaluation and Review Technique*) 1958'de ABD Donanması'nın Polaris füze programı için geliştirildi. CPM (*Critical Path Method*, kritik yol yöntemi) aynı yıllarda sanayide, bakım ve inşaat projelerinin planlanması için ortaya çıktı. RAMPS (*Resource Allocation and Multi-Project Scheduling*) ise kaynakların birden çok proje arasında paylaştırılmasını ele alır. Bugünkü proje yönetimi yazılımlarının çoğu bu yöntemlerin hesaplarını yapar.

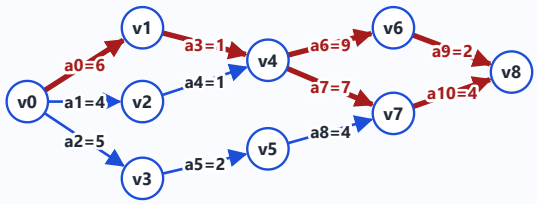
18.7.3 Kritik yol

Aktiviteler paralel yürüyebildiği için proje süresi bütün sürelerin toplamı değildir. Toplam 6 + 4 + 5 + 1 + 1 + 2 + 9 + 7 + 4 + 2 + 4 = 45 gündür, ama proje bundan çok daha kısa sürede biter. Bitiş olayı, kendisine giden her yolun aktiviteleri bitince gerçekleşir. Öyleyse proje süresi, başlangıçtan bitişe giden yolların *en uzununun* uzunluğudur.

Tanım 18.10

AoE ağında başlangıç olayından bitiş olayına giden en uzun yola **kritik yol** (*critical path*) denir. Kritik yolun uzunluğu, projenin bitebileceği en kısa süredir. Bir ağda birden çok kritik yol bulunabilir.

Örnek ağda iki kritik yol vardır: v0-v1-v4-v6-v8 (6 + 1 + 9 + 2 = 18) ve v0-v1-v4-v7-v8 (6 + 1 + 7 + 4 = 18). Proje en erken 18 günde biter. Başka yollar daha kısadır; örneğin v0-v3-v5-v7-v8 yolu 5 + 2 + 4 + 4 = 15 gündür. Şekil 18.17 kritik yolları kalın çizgiyle gösterir.



Şekil 18.17: Kritik yollar: v0-v1-v4-v6-v8 ve v0-v1-v4-v7-v8, uzunluk 18. İki yol v4'e kadar ortaktır.

"En kısa süre" ile "en uzun yol"un aynı şey olması ilk anda çelişkili görünür. Çelişki yoktur: proje, en uzun bekleme zincirinden daha kısa sürede bitemez, ama o zincir bitince başka bekleyen iş de kalmaz. Genel bir çizgede en uzun basit yolu bulmak NP-zordur. AoE ağı ise döngüsüzdür ve döngüsüz çizgede en uzun yol, topolojik sırada tek geçişle bulunur. Aşağıdaki hesaplar bunu yapar.

Kritik yol kavramı yöneticiye iki şey söyler. Kritik yoldaki bir aktivite bir gün gecikirse proje de bir gün gecikir. Bu işlerin ustalarına "sakın 4 günü 5'e sarkıtmayın; 3 günde biterse memnun olurum" denir. Kritik yolda olmayan aktivitelerde ise bir miktar gecikme projeyi etkilemez. Bu miktarın ne olduğunu bulmak için her aktivitenin en erken ve en geç başlama zamanları hesaplanır. Problem bir yöneylem araştırması problemidir.

18.8 En erken ve en geç zamanlar

Bu hesaplar kâğıt üzerinde basit görünür. Ama elle işletilmesi dikkat ister: hangi değerin hangi anda kesinleştiği, hangi değerin henüz değişebileceği karıştırılırsa sonuç bozulur. Bu yüzden hesaplar aşağıda tablo hâlinde, adım adım verilmiştir.

18.8.1 En erken başlama zamanı

Bir aktivite, kuyruğundaki olay gerçekleşir gerçekleşmez başlayabilir. a_i = <k, m> aktivitesinin en erken başlama zamanı bu yüzden k olayının en erken zamanına eşittir. Olayların en erken zamanları `EnErkenBas` dizisinde tutulur.



Şekil 18.18: a_i aktivitesi k olayından m olayına gider.

Örneğin a1'in kuyruğu v0'dır ve en erken 0. günde başlar. a5'in kuyruğu v3'tür; v3'e yalnız a2 girer, o da 5 gün sürer. a5 en erken 5. günde başlar. a8'in kuyruğu v5'tir; v5'e 5. günde başlayan 2 günlük a5 girer, a8 en erken 5 + 2 = 7. günde başlar.

Bir olaya birden çok aktivite giriyorsa durum değişir. v4'e a3 ve a4 girer. a3, 6. günde başlar ve 6 + 1 = 7. günde biter. a4, 4. günde başlar ve 4 + 1 = 5. günde biter. v4 olayı ikisinin de bitmesini bekler; bu yüzden en geç biteni, yani 7. günü bekler. Kural genel olarak

> EnErkenBas[m] = max { EnErkenBas[k] + süre(<k, m>) : k ∈ P(m) }

biçiminde yazılır. *P(m)*, *m*'nin yakın öncüllerinin kümesidir. Başlangıç olayının değeri 0'dır. 0, projenin başladığı günü gösterir; proje 8 Mayıs'ta başlıyorsa 0. gün 8 Mayıs'tır. Başlangıca 10 verilseydi bütün değerler 10 gün kayardı.

Formülün bir koşulu vardır: *EnErkenBas[m]* hesaplanırken öncüllerin değerleri kesinleşmiş olmalıdır. Bu, hesabın **topolojik sırada** yapılması demektir. Dolayısıyla `TopolojikSira` yordamı küçük bir eklemeyle bu hesabı da yapar. Liste öğesine aktivitenin süresini tutan bir `sure` alanı eklenir:

```
typedef struct oge *oge_gost;
typedef struct oge {
    int dugum;
    int sure;           /* aktivitenin suresi */
    oge_gost bag;
} oge;
typedef struct {
    int sayac;
    oge_gost bag;
} basoge;
int EnErkenBas[MAX_DUGUM], EnGecBas[MAX_DUGUM];
```

Bu yapıyla öğe boyutu değişmez: `dugum` ile `sure` 4'er bayt yer kaplar ve önceki dolgunun yerini doldurur; `sizeof(oge)` yine 16'dır. Kenar ekleme yordamı da yalnız süreyi yazacak biçimde genişletilir:

```
int kenar_ekle(basoge baslik[], int i, int j, int sure)
{
    oge_gost *son = &baslik[i].bag;
    oge_gost yeni = malloc(sizeof(oge));
    if (yeni == NULL)
        return 0;
    yeni->dugum = j;
    yeni->sure = sure;
    yeni->bag = NULL;
    while (*son != NULL)
        son = &(*son)->bag;
    *son = yeni;
    baslik[j].sayac++;
    return 1;
}
```

`TopolojikSira`'nın iç döngüsünde, sayaç azaltılmadan önce şu iki satır çalıştırılır:

```
/* TopolojikSira icinde sayac azaltilmadan once: */
if (EnErkenBas[k] < EnErkenBas[j] + p->sure)
    EnErkenBas[k] = EnErkenBas[j] + p->sure;
```

`j` o an yığından alınmış ve yazılmış düğümdür, `k` de onun bir yakın ardılıdır. `j` üzerinden `k`'ye gelen yol daha uzun bir bekleme gerektiriyorsa `k`'nin değeri büyütülür. Bu, Dijkstra'daki gevşetmenin aynısıdır; tek fark `<` karşılaştırmasının yönü ve aranan değerin en büyük olmasıdır. `EnErkenBas` başta 0'larla doldurulur. Sürelerin hepsi negatif olmadığı için 0, her olay için geçerli bir alt sınırdır. Eklemelerle oluşan tam yordam şöyledir:

```
void EnErkenHesapla(basoge baslik[], int n)
{
    int i, j, k, ust = -1;
    oge_gost p;
    for (i = 0; i < n; i++)
        EnErkenBas[i] = 0;
    for (i = 0; i < n; i++)
        if (!baslik[i].sayac) {
            baslik[i].sayac = ust;  ust = i; }
    for (i = 0; i < n; i++) {
        if (ust == -1) { printf("dongu"); break; }
        j = ust;  ust = baslik[ust].sayac;   /* yigindan al */
        printf("%d ", j);
        for (p = baslik[j].bag; p; p = p->bag) {
            k = p->dugum;
            if (EnErkenBas[k] < EnErkenBas[j] + p->sure)
                EnErkenBas[k] = EnErkenBas[j] + p->sure;
            if (--baslik[k].sayac == 0) {
                baslik[k].sayac = ust;  ust = k;  /* yigina */
            }
        }
    }
}
```

- **Satır 5–9.** Diziye başlangıç değeri verilir ve öncülsüz düğümler yığına konur. Bu kısım `TopolojikSira`'nın ilk döngüsüyle aynıdır.
- **Satır 12–13.** Yığından bir olay alınır. Bu andan sonra `EnErkenBas[j]` bir daha değişmez, çünkü `j`'nin bütün öncülleri daha önce işlenmiştir.
- **Satır 16–17.** `j`'den çıkan her aktivite için ardılın değeri gerekirse büyütülür.
- **Satır 18–20.** Ardılın son öncülü de işlendiyse ardıl yığına girer. Satırların sırası önemlidir: değer güncellemesi sayaç azaltmadan önce yapılmalıdır. Aksi hâlde düğüm yığına girip başka bir turda yazıldığında, son kenardan gelen katkı henüz eklenmemiş olabilir.

18.8.2 İleri çözümlemenin izlenmesi

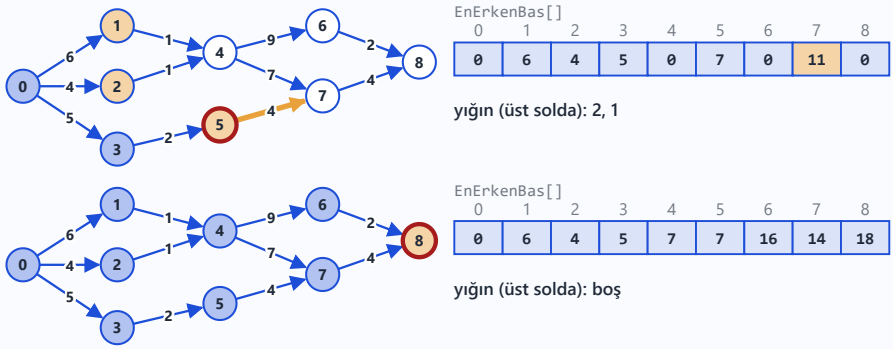
Hesap başlangıçtan bitişe doğru ilerlediği için **ileri çözümleme** (*forward stage*) diye adlandırılır. Ağ listeleri Tablo 18.14'teki sırayla kurulmuştur: v0'ın listesi v1, v2, v3; v4'ün listesi v6, v7'dir. Tablo 18.15 her olay yazıldıktan sonraki durumu verir. Satırlar programın çıktısıyla karşılaştırılmıştır.

Durum	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Yığın
Başlangıç	0	0	0	0	0	0	0	0	0	0
Çıktı v0	0	6	4	5	0	0	0	0	0	3, 2, 1
Çıktı v3	0	6	4	5	0	7	0	0	0	5, 2, 1
Çıktı v5	0	6	4	5	0	7	0	11	0	2, 1
Çıktı v2	0	6	4	5	5	7	0	11	0	1
Çıktı v1	0	6	4	5	7	7	0	11	0	4
Çıktı v4	0	6	4	5	7	7	16	14	0	7, 6
Çıktı v7	0	6	4	5	7	7	16	14	18	6
Çıktı v6	0	6	4	5	7	7	16	14	18	8
Çıktı v8	0	6	4	5	7	7	16	14	18	—

Tablo 18.15: `EnErkenBas` hesap tablosu. Her satır, adı geçen olay yazılıp listesi işlendikten sonraki durumdur; yığında tepe soldadır.

Tablonun en önemli dersi, hangi değerlerin kalıcı olduğudur. Ancak çıktıya yazılmış olayların en erken zamanları kesinleşmiştir; o andan sonra bir daha değişmez. Yazılmamış bir olayın değeri yalnız bir alt sınırdır. v5 işlenince `EnErkenBas[7]` 11 yapıldı. Ama v7'nin bir öncülü daha vardı: v4. v4 işlenince 7 + 7 = 14 bulundu ve değer 14'e çıktı. Benzer biçimde v4'ün değeri önce v2 üzerinden 5, sonra v1 üzerinden 7 oldu. Değişen hücreler tabloda vurgulanmıştır.

Son satırda `EnErkenBas[8] = 18`'dir: proje en erken 18 günde biter. Bu, kritik yolun uzunluğudur. v8'e iki aktivite girer: a10 (14 + 4 = 18) ve a9 (16 + 2 = 18). İkisi de 18 verir; bu yüzden iki kritik yol vardır. Şekil 18.19 ileri çözümlemenin iki anını gösterir.



Şekil 18.19: İleri çözümlemeden iki an. Üstte v5'in listesi işleniyor: `EnErkenBas[7]` 11 olur, ama bu değer kesin değildir. Altta bütün olaylar yazılmıştır; dizinin son hâli 0 6 4 5 7 7 16 14 18'dir.

18.8.3 En geç başlama zamanı

En erken zamanlar "en çabuk ne zaman başlanabilir?" sorusunu cevaplar. Yöneticinin ikinci sorusu "projeyi geciktirmeden en geç ne zaman başlanmalı?" sorusudur. Bu kez hesap bitişten başlar ve geriye doğru yürür. Bitiş olayının en geç zamanı en erken zamanına eşit alınır: proje 18 günde bitebiliyorsa 18. günde bitmelidir. Öyleyse *EnGecBas[8] = EnErkenBas[8] = 18*'dir.

a_i = *<k, m>* aktivitesi en geç *EnGecBas[m] − süre* anında başlamalıdır; yoksa *m* olayına geç kalınır. a10 (v7 → v8, 4 gün) en geç 18 − 4 = 14. günde başlamalıdır. a9 (v6 → v8, 2 gün) en geç 18 − 2 = 16. günde başlamalıdır. Bu yüzden *EnGecBas[7] = 14*, *EnGecBas[6] = 16*'dır.

Bir olaydan birden çok aktivite çıkıyorsa olay, en erken başlaması gereken aktiviteye yetişmelidir. Bu yüzden en küçük değer alınır:

> EnGecBas[k] = min { EnGecBas[m] − süre(<k, m>) : m ∈ S(k) }

S(k), *k*'nin yakın ardıllarıdır. v0 bunun iyi bir örneğidir. v0'dan üç aktivite çıkar. a0 için 6 − 6 = 0, a1 için 6 − 4 = 2, a2 için 8 − 5 = 3 bulunur. En küçüğü 0'dır ve *EnGecBas[0] = 0* olur. Büyüğü, yani 3 seçilseydi a0 üç gün geç başlar, v1 9. güne, v4 10. güne, v6 19. güne kayar ve proje 21 güne uzardı. İleri çözümlemede en büyük, geri çözümlemede en küçük değer alınır; ikisini karıştırmak bu tür hesapların en sık hatasıdır.

Hesap için her olayın *ardıllarına* ihtiyaç vardır ve ardılların değerleri önce kesinleşmelidir. En kolay yol bütün okları ters çevirmektir. Ters çizgede ardıllar öncüle dönüşür ve aynı topolojik sıra yordamı kullanılabilir. Ters çizge, komşuluk listesinin tersi kurularak elde edilir:

```
int ters_kur(basoge baslik[], basoge ters[], int n)
{
    int i;
    oge_gost p;
    for (i = 0; i < n; i++) {
        ters[i].sayac = 0;
        ters[i].bag = NULL;
    }
    for (i = 0; i < n; i++)
        for (p = baslik[i].bag; p; p = p->bag)      /* <i, p->dugum> */
            if (!kenar_ekle(ters, p->dugum, i, p->sure))
                return 0;
    return 1;
}
```

Yordam her *<i, j>* kenarı için *j*'nin ters listesine *i*'yi, aynı süreyle ekler. `kenar_ekle` ters listedeki `sayac` alanını da artırır; bu sayaç ters çizgedeki giriş derecesi, yani asıl çizgedeki çıkış derecesidir. Yordam her kenara bir kez dokunur ve *O(n + e)*'de biter; liste sonuna eklemenin ek maliyeti de toplamda kenar sayısıyla sınırlıdır. Ters listeler, ileri çözümleme `sayac` alanlarını bozmadan önce ya da ondan bağımsız olarak, yalnız `bag` ve `sure` alanlarından kurulur.

Geri çözümlemenin döngüsü ileri çözümlemenin aynısıdır; yalnız gevşetme satırı değişir:

```
if (EnGecBas[k] > EnGecBas[j] - p->sure)
    EnGecBas[k] = EnGecBas[j] - p->sure;
```

Burada `j` ters listede yazılan olay, `k` de onun asıl çizgedeki bir öncülüdür. `EnGecBas` başta proje süresiyle, yani 18 ile doldurulur ve değerler yalnız küçülür. Tam yordam şöyledir:

```
void EnGecHesapla(basoge baslik[], int n, int proje)
{
    int i, j, k, ust = -1;
    oge_gost p;
    for (i = 0; i < n; i++)
        EnGecBas[i] = proje;
    for (i = 0; i < n; i++)
        if (!baslik[i].sayac) {
            baslik[i].sayac = ust;  ust = i; }
    for (i = 0; i < n; i++) {
        if (ust == -1) { printf("dongu"); break; }
        j = ust;  ust = baslik[ust].sayac;   /* ters liste */
        printf("%d ", j);
        for (p = baslik[j].bag; p; p = p->bag) {
            k = p->dugum;                       /* k: oncul */
            if (EnGecBas[k] > EnGecBas[j] - p->sure)
                EnGecBas[k] = EnGecBas[j] - p->sure;
            if (--baslik[k].sayac == 0) {
                baslik[k].sayac = ust;  ust = k;
            }
        }
    }
}
```

18.8.4 Geri çözümlemenin izlenmesi

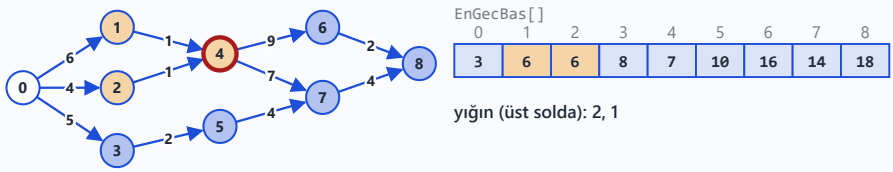
Ters çizgede öncülsüz tek düğüm v8'dir; hesap ondan başlar. Ters listeler asıl kenarlar *i* sırasıyla taranarak kurulduğu için v8'in ters listesi v6, v7; v7'ninki v4, v5; v4'ünki v1, v2 sırasındadır. Tablo 18.16 izlemeyi verir; değerler programla doğrulandı.

Durum	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Yığın
Başlangıç	18	18	18	18	18	18	18	18	18	8
Çıktı v8	18	18	18	18	18	18	16	14	18	7, 6
Çıktı v7	18	18	18	18	7	10	16	14	18	5, 6
Çıktı v5	18	18	18	8	7	10	16	14	18	3, 6
Çıktı v3	3	18	18	8	7	10	16	14	18	6
Çıktı v6	3	18	18	8	7	10	16	14	18	4
Çıktı v4	3	6	6	8	7	10	16	14	18	2, 1
Çıktı v2	2	6	6	8	7	10	16	14	18	1
Çıktı v1	0	6	6	8	7	10	16	14	18	0
Çıktı v0	0	6	6	8	7	10	16	14	18	—

Tablo 18.16: `EnGecBas` hesap tablosu (ters çizgede). Her satır, adı geçen olay yazılıp ters listesi işlendikten sonraki durumdur.

`EnGecBas[0]` üç kez değişmiştir: v3'ten 8 − 5 = 3, v2'den 6 − 4 = 2, v1'den 6 − 6 = 0. Her güncelleme daha sıkı bir zorunluluk getirir; sonuncusu, v0'ın gecikme payı olmadığını söyler. v6 işlenirken v4 için 16 − 9 = 7 bulunur; bu, v7 üzerinden bulunan değerle (14 − 7 = 7) aynıdır ve dizi değişmez. v4'ün iki ardılı da kritiktir.

Geri çözümlemenin sırası (8 7 5 3 6 4 2 1 0) ters çizgenin bir topolojik sırasıdır. İleri çözümlemenin sırasını (0 3 5 2 1 4 7 6 8) tersten okumak da aynı işi görür: bir topolojik sıranın tersi, ters çizgenin topolojik sırasıdır. Bir betik bu ikinci sırayla da aynı `EnGecBas` değerlerini buldu. Önemli olan, bir olay işlenirken bütün ardıllarının kesinleşmiş olmasıdır; rastgele bir sırada bu garanti yoktur.



Şekil 18.20: Geri çözümlemede v4 işlendikten sonra: v1 ve v2'nin en geç zamanları 7 − 1 = 6 olmuştur ve ikisi de yığındadır.

18.8.5 Bolluk ve kritik aktiviteler

Artık her aktivite için iki zaman bilinir. a_i = <*k, m*> için en erken başlama zamanı *EnErkenBas[k]*, en geç başlama zamanı *EnGecBas[m] − süre*'dir. İkisi arasındaki fark, aktivitenin projeyi geciktirmeden kaydırılabileceği gün sayısıdır.

> **Tanım 18.11**
>
> a_i = <*k, m*> aktivitesinin **bolluğu** (*slack*), en geç ve en erken başlama zamanlarının farkıdır: *EnGecBas[m] − süre*(a_i) *− EnErkenBas[k]*. Bolluğu 0 olan aktiviteye **kritik aktivite** denir.

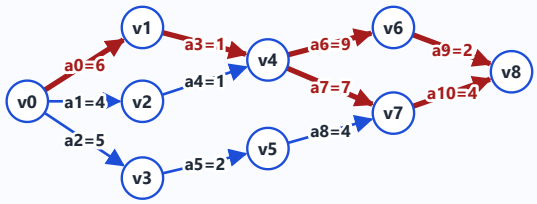
Kritik bir aktivite en erken zamanında başlamak zorundadır; bir günlük gecikmesi bütün projeyi bir gün geciktirir. Örnek ağın bütün aktiviteleri Tablo 18.17'dedir. Tablo programın çıktısıyla aynıdır.

Aktivite	Kenar	Süre	En erken	En geç	Bolluk
a0	(0,1)	6	0	0	**0: kritik**
a1	(0,2)	4	0	2	2

Aktivite	Kenar	Süre	En erken	En geç	Bolluk
a2	(0,3)	5	0	3	3
a3	(1,4)	1	6	6	**0: kritik**
a4	(2,4)	1	4	6	2
a5	(3,5)	2	5	8	3
a6	(4,6)	9	7	7	**0: kritik**
a7	(4,7)	7	7	7	**0: kritik**
a8	(5,7)	4	7	10	3
a9	(6,8)	2	16	16	**0: kritik**
a10	(7,8)	4	14	14	**0: kritik**

Tablo 18.17: Aktivitelerin en erken ve en geç başlama zamanları ve bollukları.

Bir örnekle: a4'ün en erken başlama zamanı 4, en geç başlama zamanı 7 − 1 = 6'dır. a4 iki gün gecikebilir ve proje yine 18 günde biter. Kritik aktiviteler a0, a3, a6, a7, a9 ve a10'dur. Bunlar v0'dan v8'e iki yol oluşturur ve bu yollar kritik yollardır (Şekil 18.21). Proje yönetimi yazılımları bolluk dağılımını çubuk grafiklerle gösterir; yönetici kaynakları bolluğu olan işlerden kritik işlere aktarabilir.



Şekil 18.21: Bolluğu 0 olan kritik aktiviteler kalın kırmızı kenarlarla çizilmiştir; etiketleri de vurgulanmıştır.

Tablodaki bolluk **toplam bolluktur** (*total float*). Toplam bolluk aynı yoldaki aktiviteler arasında paylaşılır. a2, a5 ve a8 aynı yol üzerindedir ve üçünün de bolluğu 3'tür. Ama bu, üçünün de üçer gün gecikebileceği anlamına gelmez. a2 üç gün gecikirse a5 ve a8'in payı tükenir; ikisi de kritik olur. Bolluk yol boyunca bir kez harcanabilir.

Bunun için ikinci bir ölçü kullanılır. **Serbest bolluk** (*free float*), bir aktivitenin ardıllarının en erken zamanını geciktirmeden kullanabileceği bolluktur: *EnErkenBas[m] − EnErkenBas[k] − süre*. a2 için 5 − 0 − 5 = 0, a5 için 7 − 5 − 2 = 0, a8 için 14 − 7 − 4 = 3'tür. Yani a2 ve a5'in bollukları kendilerine ait değildir, yoldaki a8'den gelir. a8 ise üç gününü kimseyi etkilemeden kullanabilir. a4'ün serbest bolluğu da 7 − 4 − 1 = 2'dir.

18.8.6 Kritik yolu programın bulması

İki dizi hesaplandıktan sonra her aktivitenin bolluğunu yazdırmak tek bir çift döngü ister. Asıl çizgenin listeleri gezilir; her öğe bir aktivitedir.

```
void Bolluklar(basoge baslik[], int n)
{
    int i;
    oge_gost p;
    /* EnErkenBas[] ve EnGecBas[] hesaplandiktan sonra */
    for (i = 0; i < n; i++)
        for (p = baslik[i].bag; p; p = p->bag) {
            int erken = EnErkenBas[i];
            int gec   = EnGecBas[p->dugum] - p->sure;
            printf("<%d,%d> erken=%d gec=%d bolluk=%d%s\n",
                   i, p->dugum, erken, gec, gec - erken,
                   gec == erken ? " KRITIK" : "");
        }
}
```

Çıktının ilk dört satırı şöyledir:

```
<0,1> erken=0 gec=0 bolluk=0 KRITIK
<0,2> erken=0 gec=2 bolluk=2
<0,3> erken=0 gec=3 bolluk=3
<1,4> erken=6 gec=6 bolluk=0 KRITIK
```

Kalan satırlar Tablo 18.17 ile birebir aynıdır. Yordam yalnız bolluğu 0 olan aktiviteleri tutarak kritik yolu da verir: kritik aktiviteler v0'dan v8'e uzanan bir ya da birkaç yol oluşturur. `printf`'teki `?:` işleci koşullu bir dizgi seçer: bolluk 0 ise `" KRITIK"`, değilse boş dizgi yazılır.

Bütün hesabın maliyeti parçaların toplamıdır. İleri çözümleme *O(n + e)*, ters listenin kurulması *O(n + e)*, geri çözümleme *O(n + e)*, bolluk taraması da *O(n + e)*'dir. Toplam *O(n + e)*'dir; proje ne kadar büyük olursa olsun hesap doğrusal sürede biter. Notun testlerinde ileri ve ters listeler için 22 `malloc` ve 22 `free` sayıldı; sızıntı yoktur.

Sık yapılan hata

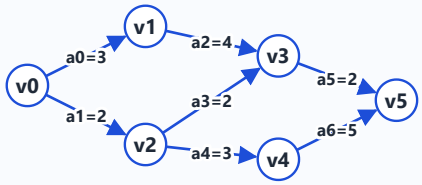
En erken zamanda en küçük, en geç zamanda en büyük değeri almak. Doğrusu tersidir. Olay, kendisine giren bütün işlerin bitmesini beklediği için en erken zaman en büyük değerdir. Olay, kendisinden çıkan bütün işlere yetişmek zorunda olduğu için en geç zaman en küçük değerdir. Kodda bu, karşılaştırma işaretlerinin yönüdür: ileri çözümlemede `<` ile büyütme, geri çözümlemede `>` ile küçültme.

18.8.7 Uygulama: bir sınav sorusu

Bu hesaplar sınavlarda sık sorulur ve kâğıt üzerinde adım adım işletilmeleri beklenir. Anlaşıldığı sanılan ama elle işletilirken çoğu zaman yanlış yapılan bir konudur. Aşağıdaki örnek bütün adımlarıyla çözülür.

Örnek 18.1

Şekil 18.22'deki AoE ağı için topolojik sıra ve her adımdaki yığın yazılsın; `EnErkenBas` ve `EnGecBas` tabloları doldurulsun; her aktivitenin bolluğu bulunup kritik yol belirtilsin. Listeler aktivite numarası sırasıyla kurulmuştur.



Şekil 18.22: Sınav sorusunun ağı: altı olay, yedi aktivite. a0 = <0,1> (3), a1 = <0,2> (2), a2 = <1,3> (4), a3 = <2,3> (2), a4 = <2,4> (3), a5 = <3,5> (2), a6 = <4,5> (5).

İleri çözümleme. Sayaçlar v0…v5 için 0, 1, 1, 2, 1, 2'dir. Yalnız v0 yığındadır. v0 yazılınca v1 ve v2'nin değerleri 3 ve 2 olur; ikisi de yığına girer ve tepe v2'dir. v2 yazılınca v3 için 2 + 2 = 4, v4 için 2 + 3 = 5 bulunur. v3'ün sayacı 1'e iner; v4 yığına girer. v4 yazılınca v5 için 5 + 5 = 10 bulunur. Sonra v1 yazılır ve v3'ün değeri 3 + 4 = 7'ye çıkar; v3 yığına girer. v3 yazılınca v5 için 7 + 2 = 9 bulunur, ama 9 < 10 olduğu için değer değişmez. En son v5 yazılır. Topolojik sıra 0 2 4 1 3 5, sonuç `EnErkenBas = [0, 3, 2, 7, 5, 10]`'dur. Proje en erken 10 günde biter.

Geri çözümleme. `EnGecBas` 10'larla başlar. Ters çizgede v5'ten başlanır: v3 için 10 − 2 = 8, v4 için 10 − 5 = 5 bulunur. v4 işlenince v2 için 5 − 3 = 2 bulunur. v3 işlenince v1 için 8 − 4 = 4 bulunur; v2 için 8 − 2 = 6, ama 2 daha küçük olduğu için değer 2 kalır. v2 işlenince v0 için 2 − 2 = 0, v1 işlenince 4 − 3 = 1 bulunur ve 0 korunur. Sıra 5 4 3 2 1 0, sonuç `EnGecBas = [0, 4, 2, 8, 5, 10]`'dur.

Aktivite	a0	a1	a2	a3	a4	a5	a6
En erken	0	0	3	2	2	7	5
En geç	1	0	4	6	2	8	5

Aktivite	a0	a1	a2	a3	a4	a5	a6
Bolluk	1	**0**	1	4	**0**	1	**0**

Tablo 18.18: Sınav sorusunun bollukları.

Kritik aktiviteler a1, a4 ve a6'dır; kritik yol v0-v2-v4-v5'tir ve uzunluğu 2 + 3 + 5 = 10'dur. Bütün değerler aynı yordamlarla programda da hesaplandı ve elle bulunanlarla örtüştü. a3'ün bolluğu (4) en büyüktür: v2 2. günde gerçekleşir, v3 ise 8. güne kadar beklenebilir; 2 günlük a3'ün önünde 4 günlük bir boşluk vardır.

18.9 Karmaşıklık özeti

Tablo 18.19 notta ele alınan bütün algoritmaları, kullandıkları veri yapılarını ve sürelerini bir arada verir. n düğüm, e kenar sayısıdır.

Algoritma	Veri yapısı	Zaman	Ek bellek
Kaba kuvvet (tek hedef)	bütün yollar	$O(n^n)$	$O(n)$
Dijkstra (temel)	maliyet matrisi + diziler	$O(n^2)$	$O(n)$
Dijkstra (iyileştirilmiş)	min-heap + komşuluk listesi	$O((n + e) \log n)$	$O(n)$
Yolu yazdırma	öncüller dizisi	$O(\text{yol uzunluğu})$	$O(\text{yol uzunluğu})$
Bellman-Ford	kenar listesi	$O(n \cdot e)$	$O(n)$
Floyd-Warshall	$n \times n$ matris	$O(n^3)$	$O(n^2)$
Topolojik sıra	sayaçlı liste + gömülü yığın	$O(n + e)$	$O(1)$
En erken / en geç zaman, bolluk	topolojik sıra + ters liste	$O(n + e)$	$O(n + e)$

Tablo 18.19: Bölümün algoritmalarının karmaşıklığı.

Ek bellek sütunu, girdinin kendisi dışında ayrılan yeri gösterir. Topolojik sıranın ek belleği sabittir, çünkü yığın başlık dizisinin içine gömülüdür. Zaman hesaplarında ise ters liste ve iki dizi gerekir. Tablodaki bütün sınırlar en kötü durum içindir. Temel Dijkstra ve Floyd-Warshall için sınır aynı zamanda en iyi durumdur: bu yordamlar girdiye bakmadan her zaman aynı sayıda tur döner.

18.10 Sık yapılan hatalar

1. **Negatif kenarlı çizgede Dijkstra kullanmak.** Algoritma hata vermez, sessizce yanlış sonuç üretir. Negatif ağırlık varsa Bellman-Ford, döngüsüz çizgede ise topolojik sırada gevşetme kullanılmalıdır.
2. **Sonsuz için `INT_MAX` seçmek.** `uzaklik[u] + maliyet[u][w]` toplamı taşar; işaretli tam sayı taşması tanımsız davranıştır ve sonuç çoğu zaman negatif bir sayıdır. Gerçek yollardan büyük ama iki katı `int`'e sığan bir sabit seçilmelidir.
3. **En erken zamanda en küçüğü, en geç zamanda en büyüğü almak.** Doğrusu tersidir: en erken zaman giren aktivitelerin en büyüğünü bekler, en geç zaman çıkan aktivitelerin en küçüğüne yetişir.
4. **En geç zamanı ardıllar kesinleşmeden hesaplamak.** Geri çözümleme ters çizgenin topolojik sırasında ya da ileri sıranın tersinde yapılmalıdır. Rastgele bir sırada, ardılı henüz işlenmemiş bir olayın değeri kesin sanılır.
5. **Topolojik sıraya rastgele bir düğümden başlamak.** Öncülleri işlenmemiş bir düğümün en erken zamanı yalnız bir alt sınırdır. Bir değer, ancak düğüm yığından alınıp yazıldığında kesinleşir.
6. **Yolu ileri doğru kurmaya çalışmak.** Kaynaktan çıkarken hangi kenarın seçileceği belli değildir. Öncüller dizisi hedeften geriye okunur, özyineleme ya da yığınla ters çevrilerek yazılır.
7. **Erişilemeyen düğüm için yol yazdırmak.** `uzaklik[h] >= SONSUZ` ise `onceki[h]` anlamsızdır; önce bu koşul denetlenmelidir.
8. **`TopolojikSira`'yı aynı yapı üzerinde ikinci kez çağırmak.** Sayaç alanları yığın bağlarına dönüşmüştür; sayaçlar yeniden hesaplanmalıdır.

18.11 Alıştırmalar ve çözümleri

18.11.1 Başka bir kaynak

Soru. Altı düğümlü örnek çizgede kaynak 2 olsaydı `EnKisaYolBul` hangi uzaklıkları bulurdu?

Çözüm. Başlangıçta 2'nin satırı kopyalanır: 0 için 20, 3 için 15, ötekiler 1000. İlk turda 3 seçilir (15); 3 üzerinden 1'e 15 + 20 = 35, 4'e 15 + 35 = 50 bulunur. İkinci turda 0 seçilir (20); 0 üzerinden hiçbir değer kısalmaz (1'e 70, 4'e 65). Üçüncü turda 1 seçilir (35); 1 üzerinden 4'e 35 + 10 = 45 bulunur ve 50'nin yerini alır. Dördüncü turda 4 seçilir. Sonuç: *d(2→0) = 20*, *d(2→3) = 15*, *d(2→1) = 35* (3 üzerinden), *d(2→4) = 45* (3 ve 1 üzerinden); 5'e yol yoktur.

18.11.2 Dış döngünün tur sayısı

Soru. `EnKisaYolBul`'un dış döngüsü neden *n − 2* tur döner? *n − 1* tur dönse ne olurdu?

Çözüm. Kaynak başta kesindir; *n − 2* tur sonunda *n − 1* düğüm işaretlenmiş olur. Kalan tek düğüm için öteki bütün düğümler *S*'dedir ve `uzaklik` değeri tanım gereği en kısa uzunluktur. Bir tur daha dönülseydi bu düğüm seçilip işaretlenirdi, ama gevşetilecek işaretsiz düğüm kalmadığı için hiçbir değer değişmezdi. Fazladan tur zararsızdır, ama gereksizdir.

18.11.3 Kenar ağırlığı değişirse

Soru. Yedi düğümlü adım adım örnekte *v3 → v6* kenarı 2 yerine 15 olsaydı *S*'ye katılma sırası ne olurdu?

Çözüm. İlk dört seçim değişmez: v1 (10), v2 (20), sonra v3 ve v4 (ikisi de 30; eşitlikte küçük indis önce: v3, v4). v3 katılınca v6'nın değeri bu kez 30 + 15 = 45 olur. v4 katılınca v5 40'a iner. 40 < 45 olduğu için v5, v6'dan önce seçilir. v5'ten v6'ya kenar yoktur; v6 45'te kalır. Sıra v1, v2, v3, v4, v5, v6 olur. Program da aynı sırayı ve `uzaklik = 0 10 20 30 30 40 45` sonucunu verdi.

18.11.4 Yığın yerine kuyruk

Soru. Topolojik sıra yordamında yığın yerine kuyruk kullanılsaydı v0…v5 örneğinde çıktı ne olurdu?

Çözüm. `0 1 2 3 5 4`. v0 alınınca v1, v2, v3 bu sırayla kuyruğa girer ve aynı sırayla çıkar. v1 alınınca v4'ün sayacı 2'ye, v2 alınınca 1'e iner; v5'in sayacı 1'e iner. v3 alınınca önce v5'in, sonra v4'ün sayacı 0'a düşer, çünkü v3'ün listesinde v5 v4'ten öncedir. Kuyruklu sürüm programda da bu çıktıyı verdi.

18.11.5 Bir aktivitenin süresi uzarsa

Soru. AoE örneğinde a8'in süresi 4 yerine 8 olsaydı proje süresi ve kritik yol ne olurdu?

Çözüm. v7'ye v5 üzerinden 7 + 8 = 15'te varılır; bu, v4 üzerinden gelen 14'ten büyüktür. `EnErkenBas[7] = 15`, v8 için 15 + 4 = 19 olur ve proje 19 güne uzar. Kritik yol artık v0-v3-v5-v7-v8'dir (5 + 2 + 8 + 4 = 19). Eski kritik yollar 18 gün sürdüğü için kritik olmaktan çıkar; program a0, a3, a6, a7 ve a9 için 1 günlük bolluk buldu. a8'in bolluğu 3 gündü; bolluğundan fazla (4 gün) uzayan kritik olmayan bir aktivite kritik yolu değiştirir. Bu yüzden hesap her değişiklikte yeniden yapılmalıdır.

18.11.6 Floyd-Warshall'da geç gelen değişiklik

Soru. Floyd-Warshall örneğinde `A[1][0]` neden *k = 2* turunda değişti?

Çözüm. 1'den 0'a en kısa yol 1 → 2 → 0'dır: 2 + 3 = 5, doğrudan kenar ise 6'dır. Bu yol 2'yi ara düğüm olarak kullanır. A^k yalnız *0…k* arasındaki ara düğümlere izin verdiği için 2'nin kullanılabilmesi ancak *k = 2* turunda mümkündür.

18.12 Özet

- En kısa yol problemi ağırlıklı yönlü çizgede kurulur; yolun uzunluğu kenar maliyetlerinin toplamıdır. Tek kaynaktan tek hedefe, tek kaynaktan bütün hedeflere ve tüm çiftler arası sürümleri vardır.
- Kaba kuvvet bütün basit yolları dener. Tam çizgede yol sayısı *1 + (n − 2) + ⋯ + (n − 2)!*'dir; tek bir hedef için bile $O(n^n)$ mertebesinde iş gerekir.

- Dijkstra açgözlüdür: S dışında en küçük uzaklıklı düğümü kesinleştirir, sonra onun üzerinden gevşetme yapar. $w \geq 0$ ister. Matrisle $O(n^2)$, min-heap ve komşuluk listesiyle $O((n + e) \log n)$'dir.
- Sonsuz için taşma yaratmayan bir sabit seçilir. `minimumBul` −1'i yalnız işaretsiz düğüm kalmadığında döndürür.
- Öncüller dizisi tek başına yeter: en kısa yollar bir ağaç oluşturur. Yol hedeften geriye okunur, özyineleme ya da yığınla doğru sırada yazılır.
- Negatif kenarda Dijkstra yanılır. Bellman-Ford bütün kenarları $n - 1$ tur gevşetir, n. turda negatif döngüyü yakalar: $O(n \cdot e)$. Floyd-Warshall ara düğümleri birer birer serbest bırakarak tüm çiftleri $O(n^3)$'te bulur.
- Düğümde aktivite ağı öncül ilişkisini gösterir ve döngüsüz olmalıdır. Topolojik sıra öncülsüz düğümleri birer birer silerek bulunur. Sayaçlı başlıklar ve gömülü yığınla $O(n + e)$'dir; aynı yordam döngüyü de yakalar.
- Kenarda aktivite ağında kenarlar aktiviteler, düğümler olaylardır. Proje süresi başlangıçtan bitişe en uzun yoldur: kritik yol.
- En erken zamanlar topolojik sırada en büyük değerle (ileri çözümleme), en geç zamanlar ters çizgede en küçük değerle (geri çözümleme) bulunur. Bolluk, en geç ile en erken başlama zamanının farkıdır. Bolluğu 0 olan aktiviteler kritik yolu oluşturur.

18.13 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

ağırlıklı çizge *weighted graph* · en kısa yol *shortest path* · kaba kuvvet *brute force* · tam çizge *complete graph* · açgözlü yöntem *greedy method* · Dijkstra algoritması *Dijkstra's algorithm* · gevşetme *relaxation* · öncül *predecessor* · ardıl *successor* · öncelikli kuyruk *priority queue* · heap *heap* · Fibonacci heap *Fibonacci heap* · negatif döngü *negative cycle* · Bellman-Ford algoritması *Bellman-Ford algorithm* · tüm çiftler arası en kısa yollar *all-pairs shortest paths* · Floyd-Warshall algoritması *Floyd-Warshall algorithm* · aktivite ağı *activity network* · düğümde aktivite ağı (AoV) *activity on vertex network* · kenarda aktivite ağı (AoE) *activity on edge network* · yönlü döngüsüz çizge *directed acyclic graph* · topolojik sıra *topological order* · giriş derecesi *in-degree* · çıkış derecesi *out-degree* · yapay aktivite *dummy activity* · kritik yol *critical path* · bolluk *slack*

18.14 Kendini Dene

1. Altı düğümlü örnek çizgede kaynak 5 alınırsa Dijkstra düğümleri hangi sırayla kesinleştirir ve uzaklıklar ne olur?

   Cevap

   5'ten tek kenar 3'e gider (3). Sıra 3 (3), 1 (23), 4 (33), 2 (38)'dir; 0 son kalan düğümdür ve değeri 2 üzerinden 58'dir. Uzaklıklar 0'dan 4'e sırasıyla 58, 23, 38, 3, 33'tür. Floyd-Warshall'ın 5. satırı da aynı değerleri verir.

2. `EnKisaYolBul` bittiğinde `onceki = {0, 3, 0, 2, 0, 0}` 'dır. `yol_yaz(onceki, 0, 3)` kaç özyinelemeli çağrı yapar ve ne yazar?

   Cevap

   İlk çağrı dahil üç çağrı yapılır: `h` = 3, 2, 0. Çıktı `0 -> 2 -> 3` 'tür; maliyet 10 + 15 = 25.

3. Sonsuz olarak `INT_MAX` kullanan bir Dijkstra kodunda, erişilemeyen bir `u` seçilip `uzaklik[u] + maliyet[u][w]` hesaplanırsa ne olabilir?

   Cevap

   Toplam `int` 'e sığmaz; işaretli taşma tanımsız davranıştır. Pratikte sonuç çoğu zaman negatif bir sayıya döner, `uzaklik[w]` 'den küçük görünür ve var olmayan bir "kısa yol" yazılır.

4. Bellman-Ford dört düğümlü negatif örnekte kaç tur döner ve toplam kaç gevşetme denemesi yapar? Hangi turdan sonra hiçbir değer

değişmez?

Cevap

n − 1 = 3 tur ve her turda 4 kenar: 12 deneme; *n*. turda da 4 kenar denetlenir. Son değişiklik ikinci turda olur (`d[3]` 6'dan 4'e); üçüncü tur hiçbir şey değiştirmez.

5. Floyd-Warshall kodunda döngülerin sırası `i` , `j` , `k` (en içte `k`) olarak değiştirilirse ne olur?

Cevap

Algoritma doğruluğunu yitirir. `A[i][j]` hesaplanırken kullanılan `A[i][k]` ve `A[k][j]` değerleri henüz bütün ara düğümler hesaba katılmadan alınmış olabilir. A^{k}'nın tanımı, `k` en dışta olduğunda sağlanır.

6. v0…v5 topolojik sıra örneğinde v3'ün listesi v5, v4 yerine v4, v5 sırasında olsaydı `TopolojikSira` ne yazardı?

Cevap

`0 3 2 5 1 4` ; çıktı değişmez. v3 işlenirken v4'ün sayacı 2'ye, v5'inki 1'e iner; ikisi de sıfıra düşmediği için yığına giriş sırası etkilenmez. Liste sırası ancak sayaçları aynı anda sıfıra düşen düğümlerin sırasını değiştirir.

7. AoE örneğinde a6'nın süresi 9 yerine 10 olsaydı proje süresi ve kritik yol ne olurdu?

Cevap

v6 en erken 7 + 10 = 17'de, v8 17 + 2 = 19'da gerçekleşir; proje 19 gün sürer. Kritik yol yalnız v0-v1-v4-v6-v8'dir. a7 ve a10'un bolluğu 1 olur, çünkü v7 üzerinden giden yol 18 gündür.

8. AoE örneğinde a4'ün toplam bolluğu ve serbest bolluğu nedir? İkisinin eşit olması ne anlama gelir?

Cevap

Toplam bolluk 6 − 4 = 2, serbest bolluk *EnErkenBas[4] − EnErkenBas[2] − 1 = 7 − 4 − 1 = 2*'dir. Eşit olmaları, a4'ün iki günlük payını hiçbir ardılın en erken zamanını etkilemeden kullanabileceğini gösterir; pay başka bir aktiviteyle paylaşılmaz.

18.15 İleri okuma

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, kesim 6.4 ve 6.5. Bu notun örnek çizgeleri, Dijkstra'nın dizi sürümü, gömülü yığınlı topolojik sıra ve en erken/en geç zaman hesapları oradaki biçimiyle izlenmiştir.
- T. H. Cormen ve ark., *Introduction to Algorithms*, 4. baskı, bölüm 20.4, 22 ve 23. Topolojik sıranın DFS ile bulunması, Dijkstra ve Bellman-Ford'un doğruluk kanıtları ve Floyd-Warshall'ın dinamik programlama çözümlemesi ayrıntılı verilir.
- R. Sedgewick, *Algorithms in C, Part 5: Graph Algorithms*, 3. baskı, bölüm 21. En kısa yol algoritmaları C koduyla ve büyük çizgeler üzerindeki deneylerle birlikte ele alınır.

18.16 Kaynaklar

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. bs., Silicon Press, 2008. §6.4 Shortest Paths and Transitive Closure, §6.5 Activity Networks.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. bs., MIT Press, 2022. Bölüm 20.4 (Topological Sort), 22 (Single-Source Shortest Paths), 23 (All-Pairs Shortest Paths).
3. R. Sedgewick, *Algorithms in C, Part 5: Graph Algorithms*, 3. bs., Addison-Wesley, 2002. Bölüm 21 (Shortest Paths).

KISIM V

Sıralama

Bölüm 19–20

Temel Sıralama Algoritmaları

Sıralama, bilgisayarın en sık yaptığı işlerden biridir. Bu not önce sıralamanın dilini kurar: kayıt, anahtar, iç ve dış sıralama, kararlılık ve yerinde sıralama. Ardından üç temel algoritmayı, seçimli, eklemeli ve kabarcık sıralamasını C ile yazar ve küçük dizilerde adım adım izler. Her biri için karşılaştırma ve taşıma sayıları en iyi, en kötü ve ortalama durumda hesaplanır. Eklemeli sıralamanın ortalama durumu bütün dizilişler üzerinden beklenen değer olarak çıkarılır. Ters çift kavramı bu üç algoritmayı tek bir ölçüye bağlar. Karşılaştırma yapmayan saymalı ve kova sıralaması, verinin aralığı ya da dağılımı biliniyorsa doğrusal zamana iner. Shell sıralaması aralıklı eklemeli sıralamayla ters çiftleri hızla yok eder. Not, karar ağacıyla kanıtlanan bir teoremle biter: karşılaştırmaya dayanan hiçbir sıralama algoritması en kötü durumda $\lceil \lg n! \rceil$ karşılaştırmanın altına inemez.

Kod (C): `github.com/sercankulcu/data-structures-and-algorithms-c/Bolum19`

ÖĞRENME ÇIKTILARI

Kavramlar
Kayıt, anahtar, iç ve dış sıralama, kararlılık ve yerinde sıralamayı ayırt etmek.

Üç temel algoritma
Seçimli, eklemeli ve kabarcık sıralamasını C ile yazmak ve bir diziyi adım adım izlemek.

Analiz
Karşılaştırma ve taşıma sayılarını en iyi, en kötü ve ortalama durum için hesaplamak; ortalamanın neden en iyi ile en kötünün ortalaması olmadığını açıklamak.

Karşılaştırmasız sıralama
Saymalı ve kova sıralamasının hangi koşulda $O(n)$ olduğunu açıklamak; saymalı sıralamanın neden kararlı olduğunu göstermek.

Shell sıralaması
Aralıklı eklemeli sıralamanın eklemeli sıralamadan neden hızlı olduğunu ters çiftlerle göstermek.

Alt sınır
Karar ağacıyla karşılaştırmalı sıralamanın $\Omega(n \lg n)$ alt sınırını kanıtlamak.

19.1 Sıralama kavramları

Telefon rehberindeki adlar, bir sınavın sonuç listesi, bir arama motorunun döndürdüğü sayfalar, bir mağazanın fiyata göre dizilmiş ürünleri: hepsi sıralanmış verilerdir. Sıralı veri iki büyük kolaylık sağlar. Birincisi arama hızlanır. Sıralı bir dizide ikili arama *O(lg n)* adımda sonuca ulaşır, sırasız dizide ise bütün öğelere bakmak gerekir. İkincisi pek çok soru sıralı veride kendiliğinden cevaplanır. En küçük ve en büyük değer dizinin iki ucundadır. Aynı değerler yan yana düşer, bu yüzden tekrarları bulmak tek bir geçiş ister. İki listenin ortak öğeleri, iki sıralı liste aynı anda gezilerek bulunur.

Sıralama algoritmaları algoritma analizinin de en iyi çalışma alanıdır. Problem kolay anlaşılır, çözümler kısadır, ama çözümlerin maliyetleri birbirinden çok farklıdır. Aynı bin sayıyı bir algoritma yarım milyon karşılaştırmayla, bir başkası on beş bin karşılaştırmayla sıralar. Bu notta önce ortak dil kurulur. Sonra üç temel algoritma ayrıntısıyla incelenir, ardından karşılaştırma yapmayan iki algoritma ve Shell sıralaması gelir. Not, karşılaştırmalı her sıralamanın uyması gereken alt sınırla biter.

19.1.1 Kayıt, alan ve anahtar

Sıralanan şey çoğu zaman tek tek sayılar değildir. Bir öğrenci işleri programı öğrencinin numarasını, adını, soyadını ve not ortalamasını birlikte tutar. Bu bilgilerin her birine **alan** (*field*), bir öğrenciye ait alanların bütününe **kayıt** (*record*) denir. C'de bir kayıt doğal olarak bir `struct` ile gösterilir:

```c
typedef struct {
    int    no;          /* anahtar */
    char   ad[20];
    char   soyad[20];
    double ort;
} ogrenci;
```

Kayıtlar bellekte tutuluyorsa topluluğa **liste**, bellek dışında, örneğin diskte tutuluyorsa **kütük** (*file*) denir. Sıralama her zaman alanlardan birine ya da birkaçına göre yapılır.

Tanım 19.1

Bir kaydı öteki kayıtlardan ayırt eden, yani biricik (*unique*) olarak tanıtan alana ya da alanlar topluluğuna **anahtar** (*key*) denir. **Sıralama** (*sorting*), *n* kayıttan oluşan bir listeyi anahtarları küçükten büyüğe (ya da büyükten küçüğe) dizilecek biçimde yeniden düzenlemektir.

Şekil 19.1 üç öğrenci kaydını gösterir. Öğrenci numarası anahtardır: iki öğrencinin adı ve soyadı aynı olabilir, numarası aynı olamaz. Ad, soyad ve ortalama veri alanlarıdır. Sıralama yalnız anahtarları karşılaştırır, ama kaydın bütün alanları anahtarla birlikte yer değiştirir. Bu ayrım iki maliyet doğurur: anahtarları karşılaştırmanın maliyeti ve kayıtları taşımanın maliyeti. Kayıt büyükse ikincisi baskın çıkabilir.

kayıt (record) = alanlar (fields)

no (anahtar)	ad	soyad	ort.
1042	Ayşe	Yılmaz	3.1
0977	Can	Demir	2.8
1310	Ece	Kaya	3.6

bellekte: liste · bellek dışında: kütük (file)

Şekil 19.1: Üç öğrenci kaydı. Turuncu başlıklı "no" alanı anahtardır; ad, soyad ve ortalama veri alanlarıdır.

Gerçek programlarda anahtar her zaman biricik olmak zorunda değildir. Öğrenciler not ortalamasına göre de sıralanabilir ve iki öğrencinin ortalaması aynı çıkabilir. Bu durumda "anahtar" sözcüğü sıralamanın dayandığı alan anlamında kullanılır. Eşit anahtarlı kayıtların sıralamadan sonra hangi sırada kalacağı ise kararlılık konusudur.

Sıralamalar verinin nerede durduğuna göre de ikiye ayrılır. **İç sıralamada** (*internal sorting*) verinin tamamı ana bellektedir ve herhangi bir öğeye sabit sürede erişilir. **Dış sıralamada** (*external sorting*) veri belleğe sığmaz; disk ya da teyp gibi yardımcı bellek kullanılır. Diskten bir blok okumak, bellekte binlerce karşılaştırma yapmaktan uzun sürer. Bu yüzden dış sıralamada maliyeti belirleyen, karşılaştırma sayısı değil disk erişim sayısıdır. Bu notun konusu iç sıralamadır. Yine de buradaki fikirler dış sıralamanın da temelidir: dış sıralama algoritmaları belleğe sığan parçaları iç sıralamayla sıralar, sonra sıralı parçaları birleştirir.

19.1.2 Maliyeti neyle ölçeriz?

Bir sıralama algoritmasının süresini doğrudan saniyeyle ölçmek yanıltıcıdır, çünkü süre makineye, derleyiciye ve girdiye göre değişir. Bunun yerine algoritmanın yaptığı temel işlemler sayılır. Sıralamada iki temel işlem vardır:

- **Karşılaştırma.** İki anahtarın sırası sorulur: `a[j] < a[min]` gibi. Karşılaştırmaya dayanan algoritmalar girdi hakkında bilgiyi yalnız bu sorulardan edinir.
- **Taşıma.** Bir öğe dizide başka bir yere yazılır. Takas (*swap*) üç atamadan oluşur; eklemeli sıralamanın kaydırması ise tek bir atamadır. Notta "taşıma" sözcüğü ikisini birden anlatır; gerektiğinde hangisinin sayıldığı ayrıca belirtilir.

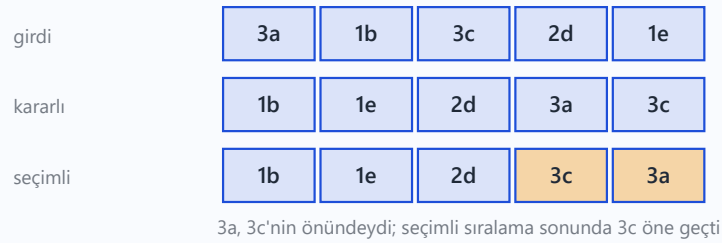
Bir algoritmanın çalışma süresi bu iki sayının toplamıyla orantılıdır. Hangisinin daha önemli olduğu veriye bağlıdır. Anahtarlar uzun karakter dizileriyse karşılaştırma pahalıdır. Kayıtlar büyükse taşıma pahalıdır. Flash bellekte her yazma hücreyi biraz yıprattığı için yazma sayısını azaltmak ayrıca değerlidir. Sayılar üç durum için verilir. **En iyi durum** algoritmanın en az iş yaptığı girdidir, **en kötü durum** en çok iş yaptığı girdidir. **Ortalama durum**, bütün olası girdilerin eşit olasılıkla geldiği varsayımıyla beklenen iştir. Ortalama durumun bu tanımı önemlidir ve eklemeli sıralamada ayrıntısıyla hesaplanır.

19.1.3 Kararlılık

Tanım 19.2

Eşit anahtarlı kayıtların sıralamadan önceki göreli sırası sıralamadan sonra da korunuyorsa sıralama algoritması **kararlıdır** (*stable*). Kesin söylemek gerekirse: girdide *i* < *j* ve *a[i]* ile *a[j]*'nin anahtarları eşitse çıktıda da *a[i]*, *a[j]*'den önce gelir.

Şekil 19.2 tanımı küçük bir örnekle gösterir. Girdide beş kayıt vardır; sayı anahtardır, harf yalnız kayıtları birbirinden ayırmak için eklenmiştir. 3a ile 3c'nin anahtarı aynıdır ve girdide 3a önce gelir. Kararlı bir algoritma çıktıda da 3a'yı 3c'nin önüne koyar: *1b 1e 2d 3a 3c*. Seçimli sıralama ise *1b 1e 2d 3c 3a* üretir. Çıktı anahtarlara göre doğru sıralanmıştır, ama 3c öne geçmiştir. Seçimli sıralamanın bu sonucu nasıl ürettiği, algoritma tanıtılınca adım adım gösterilir.



Şekil 19.2: Kararlılık. Girdide 3a, 3c'nin önündedir. Kararlı bir sıralama bu sırayı korur; seçimli sıralama 3c'yi öne geçirir.

Sayılar tek başına sıralanıyorsa kararlılığın bir önemi yoktur: iki 3'ü birbirinden ayırmanın yolu yoktur. Kararlılık, kayıtların anahtar dışında başka alanları olduğunda önem kazanır. En bilinen kullanım, birden çok alana göre sıralamadır.

Örnek 19.1

Bir sınıf listesi soyada göre, soyadı aynı olanlar da ada göre sıralanmak isteniyor. Liste (ad, soyad) çiftlerinden oluşur: (Mehmet, Kaya), (Ali, Yılmaz), (Zeynep, Kaya), (Ayşe, Yılmaz), (Can, Demir).

Önce bütün liste ada göre sıralanır: (Ali, Yılmaz), (Ayşe, Yılmaz), (Can, Demir), (Mehmet, Kaya), (Zeynep, Kaya). Sonra aynı liste soyada göre *kararlı* bir algoritmayla sıralanır. Demir tektir ve başa geçer. İki Kaya'nın göreli sırası korunur: ilk adımdan sonra Mehmet, Zeynep'ten önceydi, öyle kalır. İki Yılmaz için de Ali, Ayşe'den önce kalır. Sonuç: (Can, Demir), (Mehmet, Kaya), (Zeynep, Kaya), (Ali, Yılmaz), (Ayşe, Yılmaz). Soyadları sıralı, soyadı aynı olanlar ad sırasındadır.

İkinci sıralama kararsız olsaydı ilk sıralamanın emeği boşa gidebilirdi: Zeynep Kaya, Mehmet Kaya'nın önüne geçebilirdi.

Örnekteki sıra dikkat ister: önce *ikincil* alana (ad), sonra *birincil* alana (soyad) göre sıralanır. Son yapılan sıralama baskın gelir, öncekiler ise eşitlikleri bozmak için kalır. Basamak sıralaması (*radix sort*) aynı fikri sayıların basamaklarına uygular. Sayılar önce birler basamağına, sonra onlar basamağına, en son da en yüksek basamağa göre sıralanır; her geçişte kararlı bir algoritma kullanılır. Her geçişte kullanılan algoritmanın kararlı olması bu yüzden şarttır.

19.1.4 Yerinde sıralama

Tanım 19.3

Girdinin tutulduğu dizinin dışında yalnız sabit miktarda, yani *O(1)* ek bellek kullanan sıralama algoritmasına **yerinde** (*in-place*) algoritma denir. Bazı kaynaklar özyineleme yığınına izin vermek için *O(lg n)* ek belleği de yerinde sayar.

Seçimli, eklemeli ve kabarcık sıralaması yerindedir: birkaç döngü değişkeni ve takas için tek bir geçici değişken dışında bellek istemez. Saymalı sıralama yerinde değildir; çıktıyı ayrı bir `b` dizisine yazar ve sayaçlar için `c` dizisi kullanır. Kova sıralaması da kovalar ve düğümler için *O(n)* ek bellek ister. Bellek kıt olduğunda, örneğin gömülü bir denetleyicide, yerinde bir algoritma bu yüzden tek seçenek olabilir.

19.1.5 Ortak altyapı: `kucuk` ve `SWAP` makroları

Bu nottaki karşılaştırmalı algoritmalar aynı iki yapı taşını kullanır: iki öğeyi karşılaştıran `kucuk` ve iki öğeyi takas eden `SWAP`. İkisi de C önişlemcisinin makrolarıdır. Makro (*macro*), derlemeden önce metin olarak yerine konan bir kısaltmadır; fonksiyon çağrısının maliyeti yoktur.

```
#define N 15
#define kucuk(A, B) ((A) < (B))
#define SWAP(A, B) do { int t_ = (A); (A) = (B); (B) = t_; } while (0)
```

- **Satır 1.** `N`, deneme programının sıralayacağı öğe sayısıdır.
- **Satır 2.** `kucuk(A, B)`, $A < B$ ise 1, değilse 0 üretir. Parametrelerin ve bütün ifadenin parantez içine alınması bilinçlidir. Makro metin olarak açılır. Parantezsiz bir tanımda `kucuk(x & 1, y)` çağrısı `x & 1 < y` metnine dönüşürdü. C'de `<` işleci `&`'den önce uygulandığı için bu ifade `x & (1 < y)` olarak hesaplanırdı. Algoritmalar karşılaştırmayı bu makro üzerinden yaptığında, karşılaştırma ölçütünü değiştirmek tek satırlık bir iştir. Örneğin büyükten küçüğe sıralamak için tanımda `<` yerine `>` yazmak yeter.
- **Satır 3.** `SWAP(A, B)`, `A`'nın değerini geçici `t_` değişkenine alır, `A`'ya `B`'yi, `B`'ye eski `A`'yı yazar. Üç atama yapılır. Bu sürüm iki argümanlıdır ve öğe türü olarak `int` kullanır; başka türden öğeler için tanımdaki `int` değiştirilir.

Makronun gövdesinin `do { … } while (0)` içine alınması ilk bakışta tuhaf görünür. Döngü bir kez çalışır ve hiçbir şey yinelenmez. Amacı, makroyu sözdizimi bakımından tek bir deyim yapmaktır. Gövde yalnız süslü parantezlerle yazılsaydı ne olacağına bakalım:

```
#define SWAP(A, B) { int t = A; A = B; B = t; }

if (a[0] > a[1])
    SWAP(a[0], a[1]);
else
    puts("sirali");
```

Önişlemci `SWAP(a[0], a[1]);` satırını `{ int t = a[0]; a[0] = a[1]; a[1] = t; };` metnine çevirir. Bloktan sonraki noktalı virgül, boş bir deyimdir ve `if` deyimini sonlandırır. Ardından gelen `else` sahipsiz kalır ve derleyici hata verir. `do { … } while (0)` biçiminde ise sondaki noktalı virgül `while (0)`'ı kapatır; `if`–`else` yapısı bozulmaz. Bu kalıp, birden çok deyimden oluşan makrolar için C'de yerleşik bir alışkanlıktır.

Algoritmalarda iki makro çoğu zaman birlikte görünür: `if (kucuk(A, B)) SWAP(A, B);`. Bu satır `A < B` ise iki öğeyi takas eder. Kalıba **karşılaştır-takas** (*compare-exchange*) denir. Yönüne dikkat etmek gerekir: `A` dizide `B`'nin *sağındaysa* bu satır küçük öğeyi sola taşır. Kabarcık sıralamasındaki `if (kucuk(a[j], a[j - 1])) SWAP(a[j], a[j - 1]);` satırı tam böyledir: sağdaki `a[j]` soldakinden küçükse yer değiştirirler.

> **Sık yapılan hata**
>
> Makroya yan etkili bir argüman vermek. Makro fonksiyon değildir; argümanlar değerlendirilmeden metin olarak kopyalanır. `SWAP(a[i++], a[j])` çağrısında `a[i++]` metni gövdede iki kez geçer, bu yüzden `i` iki kez artar ve takas yanlış öğelerle yapılır. `a = {10, 20, 30, 40}`, `i = 0` ve `j = 3` iken bu çağrıdan sonra `i` 2 olur, dizi de `{10, 40, 30, 10}`'a döner: 20 kaybolmuş, 10 iki kez yazılmıştır. Argümanı gövdede iki kez geçen bir makroya `f(x)` verilirse `f` de iki kez çağrılır. Makro argümanları her zaman yan etkisiz, basit ifadeler olmalıdır.

Algoritmaları sınamak için kullanılan ana program da kısadır:

```
int main(void) {
    int *a = malloc(N * sizeof *a);
    if (a == NULL) return 1;
    for (int i = 0; i < N; i++)
        a[i] = rand() % 50;
    selection(a, 0, N - 1);   /* sıralama yordamına çağrı */
    for (int i = 0; i < N; i++) printf("%d ", a[i]);
    free(a);
    return 0;
}
```

- **Satır 2–3.** `N` tam sayılık bir dizi heap bellekten alınır. `sizeof *a`, `a`'nın gösterdiği öğenin boyudur, yani `sizeof(int)`. Bu yazım, dizinin türü değişirse kendiliğinden doğru kalır. `malloc` bellek bulamazsa `NULL` döndürür; program bu durumda 1 koduyla biter. Denetim yapılmasaydı `a[i]` yazımı `NULL` adresine yazmaya çalışırdı ve program çökerdi.
- **Satır 4–5.** Dizi 0 ile 49 arasında rastgele sayılarla doldurulur. `rand()` her çalıştırmada aynı diziyi üretir, çünkü tohum (`srand`) verilmemiştir. Bu durum sınamada bir kolaylıktır: hata bulunduğunda aynı girdi yeniden elde edilir.
- **Satır 6.** Sıralama yordamı çağrılır. Bütün yordamlar dizinin `a[L..R]` parçasını sıralar; bütün diziyi sıralamak için `L = 0`, `R = N - 1` verilir. Bu arayüz, bir dizinin yalnız bir parçasını sıralamayı da mümkün kılar. Hızlı sıralama gibi algoritmalar küçük parçaları eklemeli sıralamaya bu arayüzle devreder.
- **Satır 7–8.** Sonuç yazılır ve bellek geri verilir. `free` unutulursa küçük bir programda bir şey olmaz. Ama aynı kod uzun çalışan bir programda tekrar tekrar çağrılırsa bellek sızıntısı birikir.

Program `stdio.h` ve `stdlib.h` başlıklarıyla derlenir. `rand()`'ın ürettiği sayılar C kütüphanesine göre değişir. Linux'ta (glibc) çıktı `9 12 13 15 21 27 27 33 35 36 36 40 42 43 49` olur; başka bir kütüphanede, örneğin Windows'ta, sayılar farklıdır.

19.2 Seçimli sıralama

19.2.1 Fikir ve kod

Seçimli sıralama (*selection sort*), sıralamanın belki de en doğal yoludur. Masadaki bir deste sınav kâğıdını numara sırasına koyan biri önce en küçük numaralı kâğıdı arar ve en üste koyar. Sonra kalanlar arasında en küçüğü arar ve ikinci sıraya koyar. Bu iş kâğıt bitene kadar sürer. Algoritma üç cümleyle anlatılır:

1. Listedeki en küçük değerli öğe bulunur.
2. Bu öğe ilk konumdaki öğeyle yer değiştirir.
3. Aynı adımlar ikinci öğeden başlayarak kalan alt liste için yinelenir.

C ile gerçeklemesi:

```
void selection(int a[], int L, int R) {
    for (int i = L; i < R; i++) {
        int min = i;
        for (int j = i + 1; j <= R; j++)
            if (kucuk(a[j], a[min]))
                min = j;
        SWAP(a[i], a[min]);
    }
}
```

- **Satır 2.** Dış döngü, sıradaki en küçük öğenin yerleşeceği konumu `i` ile gezer. `i`, `R`'ye kadar değil `R - 1`'e kadar gider. $n - 1$ öğe yerine oturunca sonuncusu ister istemez doğru yerdedir: kalan tek öğe en büyük öğedir.
- **Satır 3.** `min`, şimdiye kadar görülen en küçük öğenin *indisini* tutar. Başlangıçta aday, kalan alt listenin ilk öğesi `a[i]`'dir.
- **Satır 4–6.** İç döngü kalan alt listenin geri kalanını, `a[i+1..R]`'yi tarar. Daha küçük bir öğe görülünce `min` onun indisine güncellenir. Karşılaştırma `kucuk` makrosuyla, yani katı "küçüktür" ile yapılır: eşit bir öğe `min`'i değiştirmez.
- **Satır 7.** En küçük öğe `a[i]` ile takas edilir. `min == i` ise takas öğeyi kendisiyle değiştirir; dizi değişmez, ama üç atama yine yapılır.

Algoritmanın doğruluğu bir döngü değişmeziyle (*invariant*) gösterilir. Değişmez, döngünün her turunun sonunda doğru kalan bir özelliktir.

> Her dış tur sonunda $a[L..i]$ sıralıdır ve dizinin en küçük $i - L + 1$ öğesini içerir.

İlk tur sonunda $a[L]$'de bütün dizinin en küçüğü vardır; tek öğeli bir parça sıralıdır. Bir tur başlarken değişmez $i - 1$ için doğruysa, tur kalan öğelerin en küçüğünü bulur ve $a[i]$'ye koyar. Bu öğe $a[L..i-1]$'deki her öğeden küçük değildir (onlar daha önce seçildi) ve kalanların hepsinden büyük değildir. Değişmez i için de doğru olur. Döngü $i = R - 1$ ile bittiğinde $a[L..R-1]$ sıralıdır ve en küçük $n - 1$ öğeyi taşır; kalan $a[R]$ en büyüktür. Dizi sıralanmıştır.

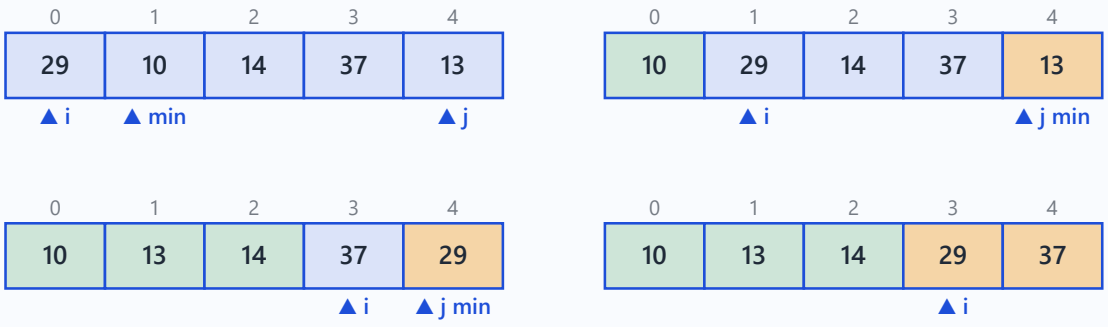
Seçimli sıralamanın belirgin bir özelliği, girdinin durumuna hiç bakmamasıdır. En küçüğü seçmek için kalan alt liste her turda sonuna kadar taranmalıdır. Dizi baştan sıralı olsa bile algoritma bunu fark etmez, çünkü "daha küçük bir öğe kalmadı" bilgisini ancak taramayı bitirince edinir.

19.2.2 Adım adım izleme

`selection(a, 0, 4)` çağrısı *a = [29, 10, 14, 37, 13]* dizisi üzerinde izlenir. Tablo 19.1 her karşılaştırmayı ayrı bir satırda verir; Şekil 19.3 her turun sonundaki takası gösterir.

`i`	`j`	Karşılaştırma	`min`	Dizi
0	1	10 < 29 doğru	1	[29, 10, 14, 37, 13]
0	2	14 < 10 yanlış	1	
0	3	37 < 10 yanlış	1	
0	4	13 < 10 yanlış	1	
0	—	`SWAP(a[0], a[1])`	1	[**10**, 29, 14, 37, 13]
1	2	14 < 29 doğru	2	
1	3	37 < 14 yanlış	2	
1	4	13 < 14 doğru	4	
1	—	`SWAP(a[1], a[4])`	4	[10, **13**, 14, 37, 29]
2	3	37 < 14 yanlış	2	
2	4	29 < 14 yanlış	2	
2	—	`SWAP(a[2], a[2])` (kendisiyle)	2	[10, 13, **14**, 37, 29]
3	4	29 < 37 doğru	4	
3	—	`SWAP(a[3], a[4])`	4	[10, 13, 14, **29**, **37**]

Tablo 19.1: `selection(a, 0, 4)` izlemesi, *a = [29, 10, 14, 37, 13]*. Her satır bir karşılaştırma ya da tur sonundaki takastır.



Şekil 19.3: Seçimli sıralamada turların sonu, soldan sağa ve yukarıdan aşağıya. (1) tur 0'ın taraması bitti, en küçük 10 (`min = 1`). (2) tur 1'in sonunda en küçük 13 (`min = 4`). (3) tur 3'te 29 seçildi. (4) son takastan sonra sıralı dizi. Yeşil kutular yerine oturmuş öğeleri, turuncu kutular o anki en küçük adayı ya da az önce takaslanan öğeleri gösterir; oklar `i` , `j` ve `min` indislerini işaret eder.

İzlemede üç ayrıntı göze çarpar. Birincisi, tur 0'da 13 de 10'dan büyük olduğu için `min` yalnız bir kez değişti. Tur 1'de ise iki kez değişti: önce 14'e, sonra 13'e. Kaç güncelleme yapılacağı girdiye bağlıdır, ama karşılaştırma sayısı bağlı değildir. İkincisi, tur 2'de 14 zaten yerindeydi ve takas kendisiyle yapıldı. Üçüncüsü, dizi tur 3'ten önce henüz sıralı değildi; son takas iki öğeyi birden yerine oturttu. Toplam 4 + 3 + 2 + 1 = 10 karşılaştırma ve 4 takas çağrısı yapıldı.

19.2.3 Analiz

Tur *i*'de iç döngü *j = i + 1, ..., R* için çalışır. *n* öğeli bir dizide (*L = 0*, *R = n − 1*) ilk turda *n − 1*, ikinci turda *n − 2*, son turda 1 karşılaştırma yapılır:

$$C(n) = (n-1) + (n-2) + \ldots + 2 + 1 = n(n-1)/2 \approx n^2/2$$

Bu toplam, ilk *n − 1* doğal sayının toplamıdır. Toplamı bulmanın bilinen yolu, terimleri bir de ters sırayla yazıp alt alta toplamaktır. Her sütunun toplamı *n*'dir ve *n − 1* sütun vardır. İki kat toplam *n(n − 1)* eder, toplamın kendisi yarısıdır.

Karşılaştırma sayısı girdiden bağımsızdır: dizi sıralı da olsa, ters sıralı da olsa, rastgele de olsa iç döngü aynı sayıda döner. Seçimli sıralama için en iyi, ortalama ve en kötü durum aynıdır ve $\Theta(n^2)$'dir. Takas sayısı ise tam olarak *n − 1*'dir: her dış turda bir takas yapılır ve son öğe için tur yoktur. Bunların bazıları öğenin kendisiyle takasıdır; kodu `if (min != i)` ile korumak gereksiz takasları önler, ama her tura bir karşılaştırma ekler. Ek bellek yalnız `i`, `j`, `min` ve takasın geçici değişkenidir: *O(1)*. Algoritma yerindedir.

n	Karşılaştırma *n(n − 1)/2*	Takas *n − 1*
5	10	4
10	45	9
100	4.950	99
1.000	499.500	999

Tablo 19.2: Seçimli sıralamada karşılaştırma ve takas sayıları. Değerler her girdi için aynıdır.

Tablo 19.2 iki sayının ne kadar farklı büyüdüğünü gösterir. *n* on katına çıktığında karşılaştırma sayısı yaklaşık yüz katına, takas sayısı ise yalnız on katına çıkar. Takas pahalıysa bu fark seçimli sıralamayı değerli kılar. Kayıtlar büyükse, örneğin her biri birkaç yüz bayt ise, her takas üç büyük kopyalama demektir. Flash bellekte her yazma hücreyi yıpratır. Bu tür durumlarda seçimli sıralamanın yaptığı *n − 1* takas, öteki temel algoritmaların n^2 mertebesindeki taşımasından çok daha ucuzdur.

Seçimli sıralama kararlı değildir. Bunun nedeni uzak takastır. `a[i]` ile `a[min]` arasında birçok öğe olabilir. `a[i]` bu öğelerin üzerinden atlar ve eşit anahtarlı bir öğenin arkasına düşebilir. Şekil 19.2'deki girdide (3a, 1b, 3c, 2d, 1e) bu adım adım görülür:

- **Tur 0.** En küçük 1b'dir (`min = 1`). 3a ile 1b takaslanır: (1b, 3a, 3c, 2d, 1e).
- **Tur 1.** Kalanların en küçüğü 1e'dir (`min = 4`). 3a ile 1e takaslanır: (1b, 1e, 3c, 2d, 3a). 3a, 3c'nin ve 2d'nin üzerinden atlayarak en sona gitti. Kararlılık bu adımda bozuldu.
- **Tur 2.** En küçük 2d'dir. 3c ile 2d takaslanır: (1b, 1e, 2d, 3c, 3a).
- **Tur 3.** `kucuk(3a, 3c)` yanlıştır; `min` değişmez ve takas kendisiyledir. Sonuç (1b, 1e, 2d, 3c, 3a).

Bu girdide 1b ile 1e'nin sırası korunmuştur, ama bu bir rastlantıdır. Kararlılık bütün girdiler için geçerli olması gereken bir özelliktir; tek bir karşı örnek algoritmayı kararsız yapar. En küçük karşı örnek üç öğelidir ve alıştırmalarda ele alınır.

19.3 Eklemeli sıralama

19.3.1 Fikir ve kod

Eklemeli sıralama (*insertion sort*), iskambil oynayan birinin elindeki kâğıtları dizme biçimidir. Kâğıtlar masadan tek tek alınır. Eldeki kâğıtlar her an sıralıdır; yeni kâğıt bunların arasında uygun yere sokulur. Bunun için yeni kâğıttan büyük olanlar bir sağa itilir ve açılan boşluğa yeni kâğıt konur.

Dizide de aynısı yapılır. Dizinin solunda sıralı bir **ön ek** büyür. Başlangıçta ön ek yalnız ilk öğedir ve tek öğe her zaman sıralıdır. Her turda ön ekin hemen sağındaki öğe alınır ve ön ek içinde yerine eklenir. Tur sonunda ön ek bir öğe uzar. İki sürümden ilki şudur:

```
void insertion(int a[], int L, int R) {
    for (int i = L + 1; i <= R; i++) {
        int v = a[i];      // yerleştirilecek
        int j = i - 1;     // a[L..i-1] sıralı
        while (j >= L && a[j] > v) {
            a[j + 1] = a[j]; // sağa kaydır
            j--;
        }
        a[j + 1] = v;
    }
}
```

- **Satır 2.** `i`, yerleştirilecek öğenin indisidir ve `L + 1`'den başlar. Bu anda `a[L..i-1]` sıralıdır.
- **Satır 3.** Yerleştirilecek değer `v`'ye alınır. Bu adım zorunludur: ilk kaydırma `a[i]`'nin üzerine yazar ve değer kaybolur.
- **Satır 4.** `j`, ön ekin sağ ucundan başlar ve sola doğru ilerler.
- **Satır 5.** Döngü iki koşul birlikte doğru olduğu sürece döner: `j` alt dizinin içindedir (`j >= L`) ve `a[j]`, `v`'den büyüktür. C'nin `&&` işleci kısa devreyle çalışır: ilk koşul yanlışsa ikincisi hiç hesaplanmaz. `j` $L-1$'e indiğinde `a[j]` okunmaz; okunsaydı dizinin dışına çıkılırdı.
- **Satır 6–7.** `v`'den büyük öğe bir sağa kaydırılır ve `j` bir azalır. Kaydırma tek bir atamadır, takas değildir.
- **Satır 9.** Döngü bittiğinde ya `a[j]`, `v`'den büyük değildir ya da `j` alt dizinin solundadır. Her iki durumda da `v`'nin yeri `j + 1`'dir.

Döngü değişmezi bu kez şöyledir: dış döngünün her turu başlarken *a[L..i−1]*, girdide bu konumlarda bulunan öğelerin sıralı hâlidir. Seçimli sıralamanın değişmeziyle aradaki fark önemlidir. Eklemeli sıralamada ön ek sıralıdır ama dizinin en küçük öğelerini içermek zorunda değildir: sonradan gelen küçük bir öğe ön ekin en başına girebilir. Algoritma son tura kadar hiçbir öğenin son yerini kesin bilmez.

> **Sık yapılan hata**
>
> Döngü koşulunu `j >= 0` yazmak. Bütün dizi sıralanırken $L = 0$ olduğu için hata görünmez. Yordam bir alt dizi için, örneğin `insertion(a, 2, 5)` ile çağrılınca `j`, `L`'nin soluna geçip `a[0]` ve `a[1]`'e kadar iner. Bu öğeler sıralanan parçaya ait değildir; kaydırma onları bozar. Doğru sınır `j >= L`'dir. *a = [9, 8, 7, 1, 6, 5, 0]* için `insertion(a, 2, 5)` doğru kodla *[9, 8, 1, 5, 6, 7, 0]* verir: yalnız *a[2..5]* sıralanır.

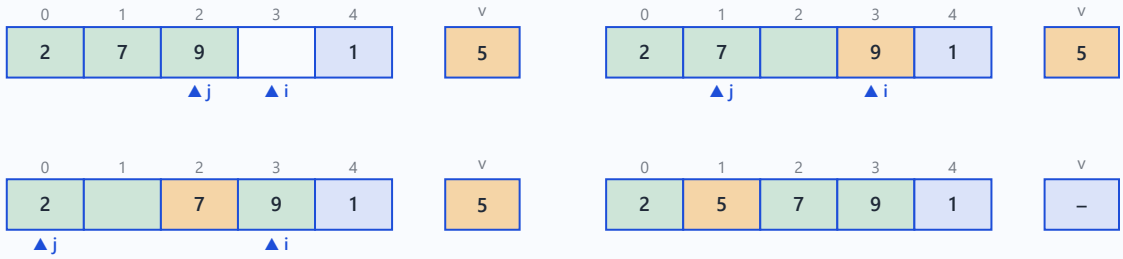
19.3.2 Adım adım izleme

`insertion(a, 0, 4)` çağrısı *a = [7, 2, 9, 5, 1]* üzerinde izlenir. Tablo 19.3 her turun karşılaştırmalarını, kaydırmalarını ve tur sonundaki diziyi verir.

`i`	`v`	Sınamalar	Kaydırma	Karş.	Tur sonu
1	2	7 > 2 doğru; `j = -1`, döngü biter	1	1	[**2**, 7, 9, 5, 1]
2	9	7 > 9 yanlış	0	1	[2, 7, **9**, 5, 1]
3	5	9 > 5 doğru, 7 > 5 doğru, 2 > 5 yanlış	2	3	[2, **5**, 7, 9, 1]
4	1	9, 7, 5, 2 > 1 doğru; `j = -1`, döngü biter	4	4	[**1**, 2, 5, 7, 9]
Toplam			**7**	**9**	

Tablo 19.3: `insertion(a, 0, 4)` izlemesi, *a = [7, 2, 9, 5, 1]*. "Karş." sütunu `a[j] > v` sınamalarının sayısıdır.

En öğretici tur *i = 3*'tür. Şekil 19.4 bu turun ara durumlarını gösterir. `v = 5` bir kenara alınmıştır. 9, 5'ten büyüktür ve bir sağa kayar; dizide geçici olarak iki 9 görünür. Sonra 7 kayar ve iki 7 görünür. 2 ise 5'ten küçüktür; döngü durur ve 5, `a[1]`'e yazılır. Ara durumlarda dizide bir değerin iki kopyası olması normaldir. Kaydırma bir öğeyi sağa kopyalar; eski yer bir sonraki adımda ya başka bir öğeyle ya da `v` ile doldurulur.



Şekil 19.4: Eklemeli sıralamada *i = 3* turu, soldan sağa ve yukarıdan aşağıya. (1) `v = 5` kenara alındı. (2) 9 bir sağa kaydı. (3) 7 bir sağa kaydı. (4) *2 ≤ 5*, döngü durdu ve 5, `a[1]` 'e yazıldı. Yeşil kutular sıralı ön eki gösterir.

İzlemede 7 kaydırma ve 9 karşılaştırma sayıldı. Kaydırma sayısı rastlantı değildir: 7, dizideki ters sıralı çiftlerin sayısıdır. Bu bağlantı ters çift kavramıyla birlikte ayrıca kanıtlanır. Karşılaştırma sayısı kaydırma sayısından biraz fazladır. Bir tur ya `a[j] > v` yanlış çıkınca biter ve bu son sınama kaydırmasız bir karşılaştırmadır. Ya da `j` dizinin başından çıkınca biter; o zaman fazladan karşılaştırma yoktur. Burada *i = 2* ve *i = 3* turları ilk yolla bitti, *i = 1* ve *i = 4* turları ikinci yolla. Karşılaştırma sayısı *7 + 2 = 9* oldu.

19.3.3 Bekçili sürüm: `insertion1`

İç döngünün her adımı iki sınama yapar: `j >= L` ve `a[j] > v`. Birincisi yalnız `v` dizinin başına kadar gidecekse işe yarar. Bu yalnız `v` o ana kadar görülen öğelerin en küçüğü olduğunda olur. Dizinin en küçük öğesi baştan `a[L]` 'ye yerleştirilirse hiçbir `v` onun soluna geçemez; `a[L]` her döngüyü kendiliğinden durdurur. Böyle bir öğeye **bekçi** (*sentinel*) denir. İkinci sürüm bu fikri kullanır:

```
void insertion1(int a[], int L, int R) {
    for (int i = R; i > L; i--) // bekçi
        if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
    for (int i = L + 2; i <= R; i++) {
        int v = a[i], j = i;
        while (kucuk(v, a[j - 1])) {
            a[j] = a[j - 1];
            j--;
        }
        a[j] = v;
    }
}
```

- **Satır 2–3.** Hazırlık geçişi: dizinin sağından soluna komşular karşılaştırılır ve sağdaki küçükse takaslanır. Bu tek geçiş en küçük öğeyi kabarcık gibi `a[L]` 'ye taşır. *n − 1* karşılaştırma ve en çok *n − 1* takas yapılır.
- **Satır 4.** Asıl döngü `L + 2` 'den başlar. Hazırlık geçişinden sonra `a[L]` en küçüktür, bu yüzden `a[L..L+1]` zaten sıralıdır.
- **Satır 5–10.** İç döngüde yalnız `kucuk(v, a[j - 1])` sınanır. `j`, `L + 1` 'e indiğinde `a[L]`, `v` 'den büyük olamayacağı için koşul yanlış olur ve döngü durur. `j` 'nin `L` 'nin soluna geçmesi imkânsızdır.

Bu sürüm iki küçük farkla daha hızlıdır. İç döngü adım başına bir sınama eksik yapar; döngü binlerce kez döndüğünde bu fark ölçülebilir. Ayrıca koşul tek bir karşılaştırma olduğu için derleyici döngüyü daha sıkı derler. Bedeli, hazırlık geçişinin fazladan *n − 1* karşılaştırmasıdır. Tablo 19.4, `insertion1(a, 0, 3)` çağrısını *a = [4, 3, 1, 2]* üzerinde izler.

Aşama	Sınama	İşlem	Dizi
Bekçi, `i = 3`	2 < 1 yanlış	takas yok	[4, 3, 1, 2]
Bekçi, `i = 2`	1 < 3 doğru	takas	[4, 1, 3, 2]
Bekçi, `i = 1`	1 < 4 doğru	takas; bekçi hazır	[**1**, 4, 3, 2]
`i = 2`, `v = 3`	3 < 4 doğru	4 sağa kayar, `j = 1`	[1, 4, 4, 2]
	3 < 1 yanlış	`a[1] = 3`	[1, **3**, 4, 2]
`i = 3`, `v = 2`	2 < 4 doğru	4 sağa kayar, `j = 2`	[1, 3, 4, 4]
	2 < 3 doğru	3 sağa kayar, `j = 1`	[1, 3, 3, 4]

Aşama	Sınama	İşlem	Dizi
	2 < 1 yanlış (bekçi)	`a[1] = 2`	[1, **2**, 3, 4]

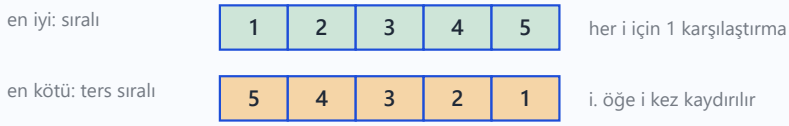
Tablo 19.4: `insertion1(a, 0, 3)` izlemesi, *a = [4, 3, 1, 2]*.

Son turda `v = 2`, iki öğeyi kaydırdıktan sonra `a[0] = 1` ile karşılaştı ve durdu. Sınır denetimi olmadan güvenle durabildi, çünkü bekçi her `v`'den küçük ya da ona eşittir. Hazırlık geçişinin bir yan etkisi de vardır: yalnız en küçük öğe değil, yol üzerindeki başka küçük öğeler de biraz sola kayar. Burada 1, 3'ün ve 4'ün önüne geçerken 3 de bir sağa itildi. Bu yüzden sonraki turlarda kaydırılacak iş biraz azalabilir.

Bekçi fikri yalnız sıralamaya özgü değildir. Bir dizide doğrusal arama yaparken aranan değer dizinin sonuna geçici olarak yazılırsa döngü her adımda "dizi bitti mi?" diye sormak zorunda kalmaz. Bağlı listelerde başlık düğümü de aynı işi görür: liste boş olsa bile ilk düğümün öncülü vardır ve özel durum ortadan kalkar.

19.3.4 En iyi ve en kötü durum

Eklemeli sıralamanın işi, seçimli sıralamanınkinin tersine, girdiye çok bağlıdır. Şekil 19.5 iki uç durumu yan yana koyar.



Şekil 19.5: Eklemeli sıralamanın en iyi durumu sıralı girdi (her öğe için tek karşılaştırma), en kötü durumu ters sıralı girdidir (her öğe ön ekin tamamını geçer).

En iyi durum: sıralı girdi. Her turda `v`, ön ekin en sağındaki öğeyle karşılaştırılır. O öğe `v`'den küçük ya da eşittir; döngü hiç dönmez ve `v` yerinde kalır. *n − 1* turun her birinde bir karşılaştırma yapılır, kaydırma yapılmaz. Toplam *n − 1* karşılaştırma eder; çalışma süresi *Θ(n)*'dir. Hiçbir algoritma bundan iyisini yapamaz: dizinin sıralı olduğundan emin olmak için her öğeye en az bir kez bakmak gerekir.

En kötü durum: ters sıralı girdi. Her yeni öğe o ana kadar görülenlerin hepsinden küçüktür. *i.* turda `v`, ön ekteki *i* öğenin hepsiyle karşılaştırılır, hepsi bir sağa kayar ve `v` en başa yazılır. Toplam

$$1 + 2 + \ldots + (n - 1) = n(n - 1)/2$$

karşılaştırma ve aynı sayıda kaydırma eder. Çalışma süresi $\Theta(n^2)$'dir. *n = 1.000* için bu 499.500 karşılaştırma ve 499.500 kaydırmadır.

Eklemeli sıralama kararlıdır. İç döngünün koşulu katı eşitsizliktir: `a[j] > v`. Ön ekte `v`'ye eşit bir öğeyle karşılaşılınca koşul yanlış olur ve `v` o öğenin *sağına* yazılır. Girdide sonra gelen öğe, çıktıda da sonra kalır. Koşul `a[j] >= v` yazılsaydı `v` eşitlerin üzerinden atlayıp onların soluna geçerdi; dizi yine sıralanırdı ama kararlılık kaybolurdu ve gereksiz kaydırmalar yapılırdı. Algoritma yerindedir: `i`, `j` ve `v` dışında bellek kullanmaz.

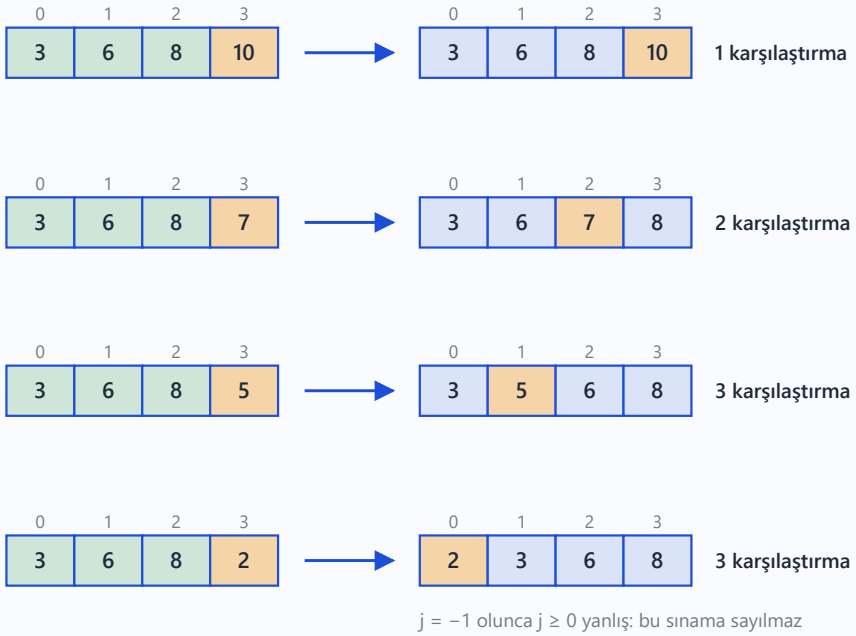
Eklemeli sıralamanın asıl gücü "neredeyse sıralı" veridir. Her öğe son yerine birkaç konum uzaksa her tur birkaç adımda biter ve toplam iş *n*'ye yakın kalır. Sıralı bir listeye arada bir yeni kayıt eklenen uygulamalar, büyük ölçüde sıralanmış bir diziyi son kez düzelten algoritmalar ve küçük parçalar bu durumdadır. Hızlı sıralamanın pratik gerçeklemeleri, parçalar 10–20 öğenin altına inince eklemeli sıralamaya geçer.

19.3.5 Ortalama durum: bir örnek

En iyi durum *n − 1*, en kötü durum *n(n − 1)/2* karşılaştırmadır. Rastgele bir girdi için kaç karşılaştırma beklenir? Akla ilk gelen, iki ucun ortalamasını almaktır. Bu yanlıştır. Ortalama durum analizinde örnekler uzayındaki bütün durumlar gözlenmeli ve her biri olasılığıyla tartılmalıdır. Uçların ortalaması ise yalnız iki durumu hesaba katar.

Önce tek bir turun ortalaması bulunur. Tur *j*'de (bu kesimde tur indisi için *j* kullanılır) *a[0..j−1]* sıralıdır ve *a[j]* yerleştirilecektir. Girdi rastgeleyse *a[j]*'nin ön ekteki *j* öğe arasındaki sırası da rastgeledir. Öğe *j + 1* konumdan herhangi birine eşit olasılıkla yerleşir: ön ekin en sağına, bir solundakine, … , en başa.

Şekil 19.6, *j = 3* için bu dört durumu sıralı ön ek *[3, 6, 8]* üzerinde gösterir.



Şekil 19.6: *j = 3* için dört durum. Yukarıdan aşağıya 10, 7, 5 ve 2 değeri *[3, 6, 8]* ön ekine eklenir; gereken karşılaştırma sayısı sırasıyla 1, 2, 3 ve 3'tür.

- **Durum 1,** `v = 10`**.** *8 > 10* yanlıştır. 1 karşılaştırma.
- **Durum 2,** `v = 7`**.** *8 > 7* doğru, *6 > 7* yanlıştır. 2 karşılaştırma.
- **Durum 3,** `v = 5`**.** *8 > 5* ve *6 > 5* doğru, *3 > 5* yanlıştır. 3 karşılaştırma.
- **Durum 4,** `v = 2`**.** *8 > 2*, *6 > 2*, *3 > 2* doğrudur. Sonra `j = -1` olur, `j >= 0` yanlış çıkar ve kısa devre yüzünden `a[j] > v` hiç sınanmaz. 3 karşılaştırma.

Son iki durum aynı sayıda karşılaştırma ister. Öğe en başa gidecekse son karşılaştırma "dur" cevabı vermez; döngüyü sınır sınaması bitirir. Bu yüzden 4 karşılaştırmalık bir durum yoktur. Dört durum eşit olasılıklıdır ve beklenen karşılaştırma sayısı

$$(1 + 2 + 3 + 3) / 4 = 9/4 = 2{,}25$$

olur. Bu sayı en iyi tur (1) ile en kötü turun (3) ortalaması olan 2'den farklıdır. Burada fark küçüktür, ama yöntem yanlıştır ve başka algoritmalarda büyük hatalara yol açar. Erken duran kabarcık sıralamasında uçların ortalaması $\approx n^2/4$ karşılaştırma verir; bütün dizilişler sayılınca gerçek ortalamanın $\approx n^2/2$ olduğu görülür.

19.3.6 Ortalama durum: genel hesap

Hesabı genelleştirmek için olasılığın bir kavramı gerekir. Değeri rastlantıya bağlı bir sayıya **rastlantı değişkeni** denir. C_j, *j*. öğeyi yerleştirirken yapılan karşılaştırma sayısı olsun. Değeri girdiye bağlıdır, bu yüzden bir rastlantı değişkenidir. Bir rastlantı değişkeninin **beklenen değeri** $E(C_j)$, alabileceği her değerin olasılığıyla çarpılıp toplanmasıyla bulunur. Bütün durumlar eşit olasılıklıysa beklenen değer basit aritmetik ortalamadır.

Tur *j*'de *j + 1* durum vardır. Öğe sağdan *k*. konuma yerleşecekse (*k = 1, …, j*) *k* karşılaştırma yapılır. En başa yerleşecekse, yani *j + 1*. durumda, *j* karşılaştırma yapılır. Böylece

$$E(C_j) = (1 / (j + 1)) \cdot [1 + 2 + \ldots + j + j] = (1 / (j + 1)) \cdot [2j + \Sigma_{k=1}^{j-1} k]$$

Köşeli parantezin içi *j(j + 1)/2 + j*'dir. *j + 1*'e bölünce

$$E(C_j) = j/2 + j/(j + 1) = j/2 + 1 - 1/(j + 1) = (j + 2)/2 - 1/(j + 1)$$

bulunur. *j = 3* için *5/2 − 1/4 = 2,25* eder; bu, örnekte elle bulunan değerdir. *j = 4* için *3 − 1/5 = 2,8*'dir: beş durumun karşılaştırma sayıları 1, 2, 3, 4 ve 4'tür, ortalamaları *14/5 = 2,8*.

Bütün dizinin toplam karşılaştırma sayısı $Y = C_1 + C_2 + \dots + C_{N-1}$'dir. Beklenen değer toplamın üzerine dağılır: toplamın beklenen değeri, beklenen değerlerin toplamıdır. Bu yüzden turlar arasındaki bağıntıyla uğraşmadan

$$E(Y) = \Sigma_{j=1}^{N-1} E(C_j) = \Sigma_{j=1}^{N-1} [(j + 2)/2 - 1/(j + 1)]$$

yazılır. İlk parça $(1/2) \cdot [N(N - 1)/2 + 2(N - 1)] = (N^2 + 3N)/4 - 1$'dir. İkinci parça $1/2 + 1/3 + \dots + 1/N = H_N - 1$'dir; burada $H_N = 1 + 1/2 + \dots + 1/N$, **harmonik sayıdır**. Harmonik sayı yavaş büyür: $H_N = \ln N + O(1)$. İki parça birleşince -1'ler birbirini götürür:

Teorem 19.1 (Eklemeli sıralamanın ortalaması)

Birbirinden farklı N anahtarın bütün $N!$ dizilişi eşit olasılıklıysa `insertion` yordamının beklenen karşılaştırma sayısı

$$E(Y) = (N^2 + 3N)/4 - H_N \approx N^2/4$$

olur. Beklenen kaydırma sayısı $N(N - 1)/4$'tür.

İspat. Karşılaştırma sayısı yukarıdaki hesaptır. Kaydırma sayısı için aynı yol izlenir: öğe sağdan k. konuma yerleşirse $k - 1$ kaydırma yapılır, $k = 1, \dots, j + 1$. Ortalaması $j/2$'dir. $j = 1, \dots, N - 1$ üzerinden toplam $N(N - 1)/4$ eder.

Formülün doğruluğu küçük N değerleri için bütün dizilişler tek tek sayılarak denetlendi. Tablo 19.5'te "bütün dizilişler" sütunu, $N!$ permütasyonun her biri sayaçlı bir `insertion` ile sıralanıp karşılaştırma sayılarının ortalaması alınarak bulundu. Sonuç formülle birebir aynıdır.

N	$N!$	Bütün dizilişler	Formül	$N^2/4$	Ortalama kaydırma
3	6	2,667	2,667	2,25	1,5
4	24	4,917	4,917	4,00	3,0
5	120	7,717	7,717	6,25	5,0
6	720	11,050	11,050	9,00	7,5
7	5.040	14,907	14,907	12,25	10,5
8	40.320	19,282	19,282	16,00	14,0

Tablo 19.5: Eklemeli sıralamanın ortalama karşılaştırma sayısı. Bütün $N!$ diziliş C programıyla sayıldı.

Küçük N'lerde formül $N^2/4$'ten belirgin biçimde büyüktür, çünkü $3N/4$ terimi henüz önemlidir. N büyüdükçe $N^2/4$ baskın gelir. $N = 1.000$ için formül $250.750 - 7,49 \approx 250.743$ verir; bu, en kötü durumun (499.500) yaklaşık yarısıdır. Ortalama durum en kötü durumdan iki kat iyidir, ama yine $\Theta(n^2)$'dir. Kaydırmalar da ortalamada $n^2/4$ kadardır. Kaydırmaya "yarı değişim" de denir: takasta iki öğe yer değiştirir, kaydırmada yalnız biri.

Ortalama durum analizinin dayandığı varsayıma dikkat etmek gerekir: bütün dizilişler eşit olasılıklıdır. Gerçek veri bu varsayıma uymayabilir. Bir uygulamanın verisi çoğunlukla neredeyse sıralıysa eklemeli sıralamanın gerçek ortalaması $n^2/4$'ten çok küçük olur. Analiz bir model üzerinde yapılır; model veriye uymuyorsa sonuç da uymaz.

19.4 Kabarcık sıralaması

19.4.1 Fikir ve kod

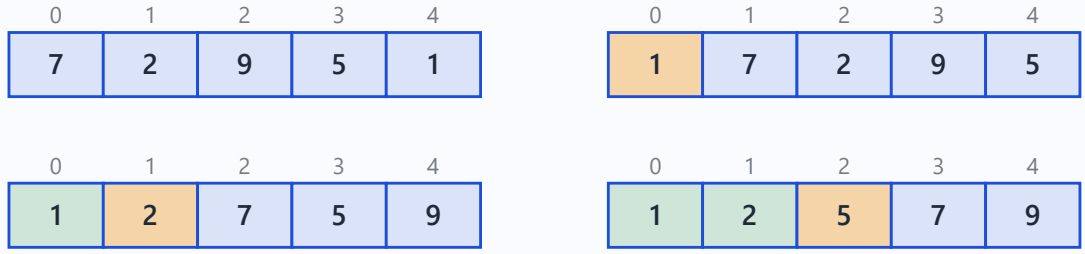
Kabarcık sıralaması (*bubble sort*) yalnız komşu öğeleri karşılaştırır ve yanlış sıradaysa takaslar. Dizinin sağ ucundan sola doğru ilerleyen bir geçişte küçük bir öğe, her komşu takasında bir konum sola kayar. Geçiş bitince kalan öğelerin en küçüğü en sola varmış olur, tıpkı suyun dibinden yüzeye çıkan bir hava kabarcığı gibi. Algoritmanın adı bu benzetmeden gelir.

```
void bubble(int a[], int L, int R) {
    int i, j;
    for (i = L; i < R; i++)
        for (j = R; j > i; j--)
            if (a[j] < a[j-1]) SWAP(a[j], a[j-1]);
}
```

- **Satır 3.** Dış döngü *n − 1* tur döner. Tur `i` 'nin sonunda `a[i]` 'de kalanların en küçüğü bulunur.
- **Satır 4.** İç döngü `j` 'yi dizinin sağ ucundan `i + 1` 'e kadar indirir. `a[L..i-1]` önceki turlarda yerine oturmuştur; o bölgeye girilmez.
- **Satır 5.** Karşılaştır-takas: sağdaki `a[j]` soldaki `a[j - 1]` 'den küçükse yer değiştirirler. Karşılaştırma katıdır; eşit komşular takaslanmaz.

Seçimli sıralama da her turda kalanların en küçüğünü `a[i]` 'ye koyar. Aradaki fark yöntemdedir. Seçimli sıralama en küçüğü bulup tek bir uzak takasla taşır. Kabarcık sıralaması ise küçük öğeyi komşu takaslarıyla adım adım sola götürür. Bu yolculuk sırasında başka öğeler de yer değiştirir. Örneğin büyük öğeler her turda biraz sağa kayar. Kabarcık sıralaması bu yüzden seçimli sıralamadan çok daha fazla taşıma yapar, ama karşılığında önemli bir özellik kazanır: yalnız komşular takaslandığı için kararlıdır.

Şekil 19.7, `bubble(a, 0, 4)` çağrısının *a = [7, 2, 9, 5, 1]* üzerindeki turlarını gösterir; Tablo 19.6 her karşılaştırmayı ayrı ayrı verir.



Şekil 19.7: Kabarcık sıralamasında turların sonu, soldan sağa ve yukarıdan aşağıya. (1) başlangıç. (2) tur 0'dan sonra 1 en sola yükseldi. (3) tur 1'den sonra 2 yerinde. (4) tur 2'den sonra dizi sıralı; tur 3 hiçbir şeyi değiştirmez. Turuncu kutu, turun yerine oturttuğu öğedir.

Tur `i`	`j`	Sınama `a[j] < a[j-1]`	Dizi
0	4	1 < 5 doğru, takas	[7, 2, 9, 1, 5]
0	3	1 < 9 doğru, takas	[7, 2, 1, 9, 5]
0	2	1 < 2 doğru, takas	[7, 1, 2, 9, 5]
0	1	1 < 7 doğru, takas	[**1**, 7, 2, 9, 5]
1	4	5 < 9 doğru, takas	[1, 7, 2, 5, 9]
1	3	5 < 2 yanlış	[1, 7, 2, 5, 9]
1	2	2 < 7 doğru, takas	[1, **2**, 7, 5, 9]
2	4	9 < 5 yanlış	[1, 2, 7, 5, 9]
2	3	5 < 7 doğru, takas	[1, 2, **5**, 7, 9]
3	4	9 < 7 yanlış	[1, 2, 5, **7**, 9]

Tablo 19.6: `bubble(a, 0, 4)` izlemesi, *a = [7, 2, 9, 5, 1]*.

Tur 0'da 1, dört takasla en sola çıktı. Tur 1'de iki şey birden oldu: 5 ile 9 yer değiştirdi ve 9 en sağa ulaştı, ardından 2 yerine yükseldi. Tur 2'de tek takas diziyi sıraladı. Tur 3 ise yalnız bir karşılaştırma yaptı ve hiçbir şey değiştirmedi; 7 zaten `a[3]` 'teydi. Toplam 10 karşılaştırma ve 7 takas yapıldı. Takas sayısı yine 7'dir, tıpkı eklemeli sıralamanın aynı dizideki kaydırma sayısı gibi.

19.4.2 Erken duran sürüm: `bubble2`

İzlemedeki son tur boşunaydı: dizi tur 2'den sonra sıralıydı, ama algoritma bunu bilmiyordu. Oysa bilgi elindeydi. Bir tur boyunca hiç takas yapılmadıysa her komşu çifti doğru sıradadır, yani dizi sıralıdır. Sonraki turlara girmeye gerek yoktur. Kod bu gözlemle biraz daha dikkatli yazılabilir:

```
void bubble2(int a[], int L, int R) {
    for (int i = L; i < R; i++) {
        int takas = 0;      // bu turda takas oldu mu?
        for (int j = R; j > i; j--)
            if (kucuk(a[j], a[j - 1])) {
                SWAP(a[j], a[j - 1]);
                takas = 1;
            }
        if (!takas) return; // sıralı: erken çık
    }
}
```

- **Satır 3.** Her turun başında `takas` bayrağı sıfırlanır.
- **Satır 5–8.** Takas yapılınca bayrak 1 olur. Karşılaştırma bu kez `kucuk` makrosuyla yazılmıştır; anlamı `bubble`'daki satırla aynıdır.
- **Satır 9.** Tur takassız bittiyse yordam hemen döner.

Tablo 19.7, `bubble2(a, 0, 4)` çağrısını *a = [2, 1, 3, 5, 4]* üzerinde izler. Bu dizide yalnız iki çift yanlış sıradadır.

Tur `i`	`j`	Sınama	`takas`	Dizi
0	—	tur başı	0	[2, 1, 3, 5, 4]
0	4	4 < 5 doğru, takas	1	[2, 1, 3, 4, 5]
0	3	4 < 3 yanlış	1	
0	2	3 < 1 yanlış	1	
0	1	1 < 2 doğru, takas	1	[1, 2, 3, 4, 5]
1	—	tur başı	0	
1	4	5 < 4 yanlış	0	
1	3	4 < 3 yanlış	0	
1	2	3 < 2 yanlış	0	
1	—	`!takas` doğru, `return`	0	[1, 2, 3, 4, 5]

Tablo 19.7: `bubble2(a, 0, 4)` izlemesi, *a = [2, 1, 3, 5, 4]*.

İlk tur iki takasla diziyi sıraladı. İkinci tur hiç takas yapmadı ve yordam döndü. Toplam 4 + 3 = 7 karşılaştırma yapıldı; erken durma olmasaydı iki tur daha (2 + 1 = 3 karşılaştırma) boşuna dönecekti. Sıralı bir girdide `bubble2` tek turda, $n - 1$ karşılaştırmayla biter; en iyi durumu $\Theta(n)$'dir.

Erken durmanın kazancı girdinin biçimine bağlıdır. Sağdan sola ilerleyen bu sürümde küçük bir öğe tek turda en sola gidebilir, ama büyük bir öğe her turda yalnız bir konum sağa ilerler. *[2, 3, 4, 5, 1]* dizisinde 1 ilk turda başa çıkar ve ikinci turda yordam durur: 2 tur. *[5, 1, 2, 3, 4]* dizisinde ise 5'in en sağa varması için dört tur gerekir; erken durma hiçbir şey kazandırmaz. İki dizide de yalnız dört çift yanlış sıradadır.

19.4.3 Analiz

`bubble` her durumda $(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2$ karşılaştırma yapar; iç döngünün sınırları girdiye bağlı değildir. Takas sayısı ise girdiye bağlıdır. Sıralı girdide hiç takas yapılmaz. Ters sıralı girdide her karşılaştırma bir takasla sonuçlanır ve $n(n - 1)/2$ takas yapılır. Rastgele girdide ortalama takas sayısı $n(n - 1)/4$'tür; bunun nedeni hemen aşağıda ters çiftlerle açıklanır.

n	Karşılaştırma	Takas (en kötü)	Takas (ortalama)
4	6	6	3

n	Karşılaştırma	Takas (en kötü)	Takas (ortalama)
5	10	10	5
6	15	15	7,5
7	21	21	10,5

Tablo 19.8: Kabarcık sıralamasında (`bubble`) karşılaştırma ve takas sayıları. Ortalama sütunu *n!* dizilişin hepsi sayılarak bulundu.

Bazı kaynaklar, örneğin Sedgewick'in kitabı, kabarcık sıralamasının ortalamada da yaklaşık $n^2/2$ değiştirme yaptığını söyler. Tablo 19.8'deki sayım ise ortalama takas sayısının $n(n-1)/4$, yani en kötü durumun tam yarısı olduğunu gösterir. Karşılaştırma sayısı için iki ifade de uyuşur: $n^2/2$.

Erken duran `bubble2` sıralı girdide $O(n)$'dir, ama ortalamada pek bir şey kazanmaz. Bütün dizilişler üzerinden sayıldığında $n = 8$ için ortalama tur sayısı 5,79'dur; en çok 7 tur dönebilirdi. Ortalama karşılaştırma sayısı 26,0'dır, erken durmasız sürümün 28 karşılaştırmasına çok yakındır. Bu yüzden karmaşıklık özetinde kabarcık sıralamasının ortalaması $\approx n^2/2$ karşılaştırma olarak verilir. Eklemeli sıralama aynı ortalama girdide bunun yarısıyla yetinir.

Kabarcık sıralaması kararlıdır: yalnız komşular takaslanır ve eşit komşular takaslanmaz. Eşit iki öğe birbirinin üzerinden hiç atlayamaz, dolayısıyla göreli sıraları değişmez. Algoritma yerindedir; ek bellek $O(1)$'dir. Toplam karmaşıklık $O(n^2)$'dir.

Uygulamada kabarcık sıralaması nadiren kullanılır. Eklemeli sıralama aynı işi daha az karşılaştırma ve daha ucuz kaydırmalarla yapar. Kabarcık sıralamasının iki değeri vardır: öğretmek kolaydır ve erken duran sürümü bir dizinin sıralı olup olmadığını tek geçişte söyler.

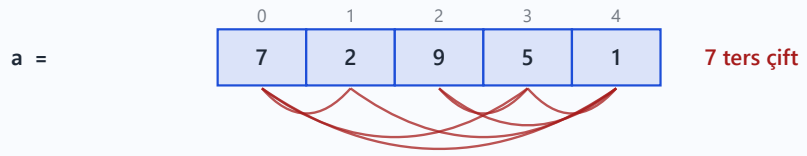
19.5 Ters çiftler: ortak ölçü

Eklemeli sıralama *[7, 2, 9, 5, 1]*'i 7 kaydırmayla, kabarcık sıralaması 7 takasla sıraladı. İki farklı algoritmanın aynı sayıda taşıma yapması bir rastlantı değildir. İkisi de aynı şeyi sayar: dizideki "ters duran" çiftleri.

Tanım 19.4

Bir *a* dizisinde $i < j$ iken $a[i] > a[j]$ olan (i, j) indis çiftine **ters çift** (*inversion*) denir. Dizinin ters çift sayısı, bu çiftlerin sayısıdır.

[7, 2, 9, 5, 1] dizisinin ters çiftleri, değerleriyle yazılırsa (7, 2), (7, 5), (7, 1), (2, 1), (9, 5), (9, 1) ve (5, 1)'dir. Şekil 19.8 bu yedi çifti yaylarla gösterir. Sıralı bir dizinin ters çift sayısı 0'dır. Ters sıralı bir dizide her çift ters çifttir ve *n* öğeden $n(n-1)/2$ çift seçilebilir. Ters çift sayısı bu yüzden bir dizinin "ne kadar sırasız" olduğunun ölçüsüdür: 0 ile $n(n-1)/2$ arasında değişir.



Şekil 19.8: *[7, 2, 9, 5, 1]* dizisindeki 7 ters çift. Her yay, solundaki öğenin sağındakinden büyük olduğu bir çifti birleştirir.

Teorem 19.2 (Komşu takası)

Bir dizide yanlış sırada duran iki komşu öğenin takası, ters çift sayısını tam olarak 1 azaltır. Eklemeli sıralamadaki her kaydırma da ters çift sayısını tam olarak 1 azaltır. Bu yüzden kabarcık sıralamasının takas sayısı ve eklemeli sıralamanın kaydırma sayısı, girdinin ters çift sayısına eşittir.

İspat. Komşu $a[k] > a[k+1]$ takaslansın. Bu iki öğenin oluşturduğu çift ters çiftti, takastan sonra değildir: 1 azalma. Başka bir çifte bakalım. İki öğesi de *k* ve $k+1$ dışındaysa çift etkilenmez. Bir öğesi dışarıda, örneğin $m < k$ konumunda olsun. $a[m]$ takastan önce de sonra da iki öğenin solundadır. Bu iki öğeyle oluşturduğu ters çiftlerin sayısı değişmez; yalnız hangi konumla eşleştiği değişir. $m > k+1$ için de aynısı geçerlidir. Toplam değişim tam olarak −1'dir.

Eklemeli sıralamada *v*'den büyük *a[j]* bir sağa kaydırılınca, sonuçta *v* o öğenin soluna geçer. Bu, *(a[j], v)* ters çiftini giderir ve başka hiçbir çiftin sırasını değiştirmez; öteki öğelerin birbirine göre sırası korunur. Kaydırma başına ters çift sayısı 1 azalır. Algoritmalar dizi sıralanınca, yani ters çift sayısı 0 olunca durduğu için taşıma sayıları ters çift sayısına eşittir.

Teoremin ortalama duruma uygulanması, rastgele bir dizinin ortalama ters çift sayısını bulmakla tamamlanır. Her *(i, j)* çifti için iki öğeden biri büyüktür; rastgele bir dizilişte büyüğün solda olma olasılığı *1/2*'dir. Bunu görmek için her dizilişi, ters çevrilmiş hâliyle eşleştirmek yeter: bir çift bu ikisinden tam birinde ters çifttir. *n(n − 1)/2* çiftin her biri ortalamada *1/2* ters çift katkısı yapar. Ortalama ters çift sayısı

$$n(n-1)/2 \cdot 1/2 = n(n-1)/4$$

olur. Kabarcık sıralamasının ortalama takas sayısı ve eklemeli sıralamanın ortalama kaydırma sayısı bu yüzden *n(n − 1)/4*'tür. Tablolardaki 3, 5, 7,5 ve 10,5 değerleri tam olarak bu formülden gelir.

Bu sonucun güçlü bir yan sonucu vardır. Yalnız komşu öğeleri takaslayan ya da öğeleri teker teker kaydıran her algoritma, her taşımada en çok bir ters çift giderir. Ortalama girdide *n(n − 1)/4* ters çift vardır. Öyleyse bu tür her algoritma ortalamada $\Omega(n^2)$ iş yapar; daha akıllı bir döngü düzeni bu duvarı aşamaz. Duvarı aşmanın tek yolu, bir taşımada birden çok ters çift gidermektir. Bu da birbirinden uzak öğeleri takaslamakla olur. Shell sıralaması ve hızlı sıralama bu yolu seçer.

Seçimli sıralama bu kuralın dışındadır; zaten uzak takas yapar. Tek bir uzak takas birçok ters çifti birden giderebilir. Örneğin *[5, 2, 3, 4, 1]* dizisinde 5 ile 1'in takası 7 ters çifti birden yok eder ve diziyi sıralar. Ancak seçimli sıralamanın sorunu taşıma değil, karşılaştırmadır: girdiden bağımsız olarak $n^2/2$ karşılaştırma yapar.

19.6 Saymalı sıralama

19.6.1 Karşılaştırmadan sıralamak

Şimdiye kadarki algoritmalar girdi hakkında bilgiyi yalnız iki öğeyi karşılaştırarak edindi. Saymalı sıralama (*counting sort*) bambaşka bir yol izler: öğeleri birbirleriyle hiç karşılaştırmaz. Bunun yerine değerleri dizi indisi olarak kullanır. Algoritma, Harold H. Seward'ın 1954'teki çalışmasına dayanır. Tek bir koşulu vardır: sıralanacak değerler tam sayıdır ve aralıkları bilinir, örneğin *0 … k*.

Fikir şöyledir. Her *x* değeri için girdide *x*'ten küçük ya da ona eşit kaç öğe olduğu bilinirse *x*'in çıktıdaki yeri de bilinir. Girdide 3'ten küçük ya da 3'e eşit üç öğe varsa son 3, çıktının üçüncü konumuna, yani 2 indisine gider. Bu sayıları bulmak için önce her değerin kaç kez geçtiği sayılır, sonra sayılar soldan sağa toplanır.

```
int counting_sort(const int a[], int b[],
                  int r, int k) {
    int *c = malloc((k + 1) * sizeof *c);
    if (c == NULL) return -1;
    for (int i = 0; i <= k; i++) c[i] = 0;  // L1
    for (int j = 0; j <= r; j++) c[a[j]]++; // L2
    for (int i = 1; i <= k; i++)            // L3
        c[i] += c[i - 1];
    for (int j = r; j >= 0; j--) {          // L4
        b[c[a[j]] - 1] = a[j];
        c[a[j]]--;
    }
    free(c);
    return 0;
}
```

Ad	Anlamı
`a[]`	girdi dizisi; `const` olduğu için yordam onu değiştirmez
`b[]`	sıralı çıktı dizisi; çağıran ayırır, en az *r + 1* öğelik olmalıdır
`c[]`	yardımcı sayaç dizisi, *k + 1* öğe; yordamın içinde ayrılır ve geri verilir
`r`	son indis: `a[0..r]` sıralanır, yani *r + 1* öğe
`k`	değerler *0 … k* aralığındadır

Tablo 19.9: `counting_sort` yordamının parametreleri ve yerel dizisi.

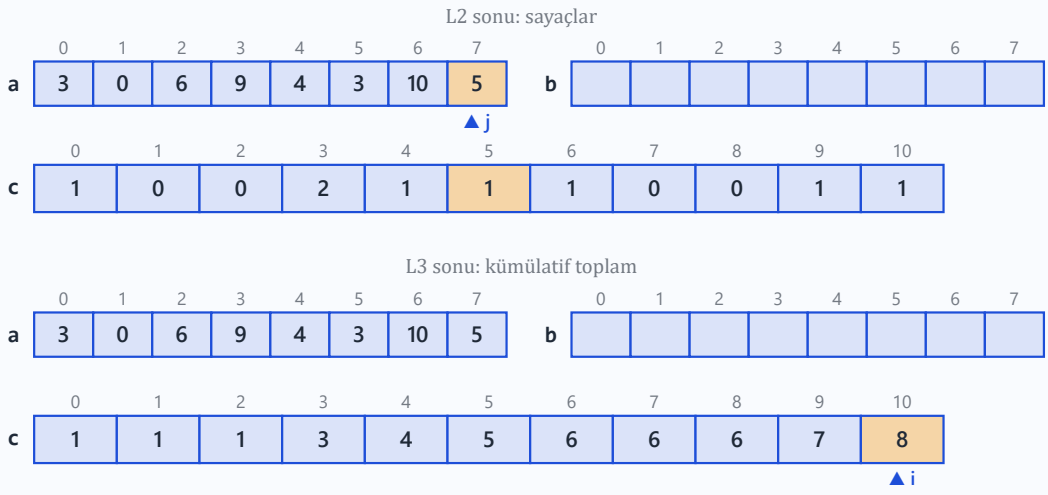
- **Satır 3–4.** Sayaç dizisi heap bellekten alınır. Bunun nedeni, `k`'nın çalışma zamanında belli olmasıdır. Kitaplardaki `int c[K]` yazımı derleme zamanında sabit bir boyut ister. Değişken uzunluklu dizi (`int c[k + 1]`) C99'da vardır, ama yığında yer kaplar ve `k` büyükse yığını taşırır. `malloc` başarısız olursa yordam *−1* döndürür ve çağıran durumu anlar.
- **Satır 5 (L1).** Sayaçlar sıfırlanır. `malloc` belleği temizlemez; bu satır olmadan sayaçlar rastgele değerlerle başlar. Aynı işi `calloc` da yapabilirdi.
- **Satır 6 (L2).** Her öğe kendi değerinin sayacını bir artırır. `c[a[j]]++` ifadesinde değer, indis olarak kullanılır. Karşılaştırma yoktur.
- **Satır 7–8 (L3).** Sayaçlar soldan sağa toplanır. Döngü bitince `c[x]`, girdide *x*'ten küçük ya da eşit öğe sayısıdır. Bu tür toplamaya **kümülatif toplam** (önek toplamı) denir.
- **Satır 9–12 (L4).** Girdi sağdan sola gezilir. `a[j]`'nin yeri `c[a[j]] − 1`'dir (indisler 0'dan başladığı için 1 çıkarılır). Öğe yazılınca sayaç bir azaltılır. Aynı değerli bir sonraki öğe bir sola yerleşir.
- **Satır 13–14.** Sayaç dizisi geri verilir ve 0 döndürülür.

19.6.2 Sayma ve kümülatif toplam

Yordam *a = [3, 0, 6, 9, 4, 3, 10, 5]*, *r = 7*, *k = 10* ile çağrılsın. L2 her öğeyi sayar: `c[3]` önce 1, ikinci 3 gelince 2 olur; 0, 4, 5, 6, 9 ve 10'un sayaçları 1'dir; 1, 2, 7 ve 8 girdide yoktur ve sayaçları 0 kalır. L3 sayaçları biriktirir. Tablo 19.10 iki aşamanın sonunu, Şekil 19.9 bellekteki üç diziyi gösterir.

İndis *x*	0	1	2	3	4	5	6	7	8	9	10
L2 sonu (sayı)	1	0	0	2	1	1	1	0	0	1	1
L3 sonu (≤ x olan)	1	1	1	3	4	5	6	6	6	7	8

Tablo 19.10: `c` dizisi L2 ve L3 sonunda, *a = [3, 0, 6, 9, 4, 3, 10, 5]*, *k = 10*.



Şekil 19.9: `counting_sort`'un sayma aşaması. Üstte L2 bitmiştir: `c[3] = 2`, girdide iki 3 vardır. Altta L3 bitmiştir: `c[x]`, *x*'ten küçük ya da eşit öğe sayısıdır. `b` henüz boştur.

L3 sonundaki satır doğrudan okunabilir. `c[9] = 7`: girdide 9'dan küçük ya da eşit 7 öğe vardır, bu yüzden 9 çıktının 7. konumuna, yani `b[6]`'ya gider. `c[3] = 3`: iki 3'ün sonuncusu `b[2]`'ye, ilki `b[1]`'e gider. Girdide olmayan değerlerin sayaçları (1, 2, 7, 8) bir öncekini tekrarlar; bu değerler çıktıya hiçbir şey yazmaz.

19.6.3 Yerleştirme ve kararlılık

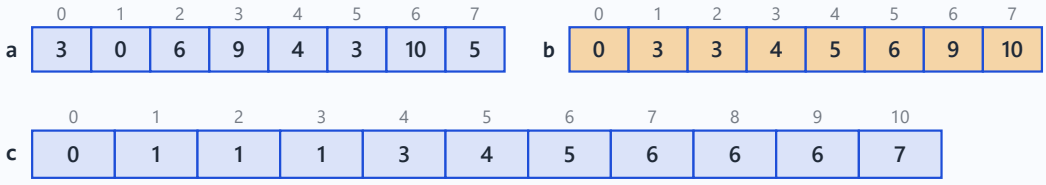
L4, girdiyi sağdan sola gezer. Tablo 19.11 her adımı verir.

`j`	`a[j]`	`c[a[j]]` **önce**	**Yazılan**	`c[a[j]]` **sonra**
7	5	5	`b[4] = 5`	4
6	10	8	`b[7] = 10`	7

`j`	`a[j]`	`c[a[j]]` **önce**	**Yazılan**	`c[a[j]]` **sonra**
5	3	3	`b[2] = 3`	2
4	4	4	`b[3] = 4`	3
3	9	7	`b[6] = 9`	6
2	6	6	`b[5] = 6`	5
1	0	1	`b[0] = 0`	0
0	3	2	`b[1] = 3`	1

Tablo 19.11: L4 izlemesi. `b[c[a[j]] - 1] = a[j]`, ardından `c[a[j]]--`.

Sonuç *b = [0, 3, 3, 4, 5, 6, 9, 10]*'dur. Şekil 19.10 L4'ün bitişini gösterir. L4 bitince `c[x]`, her değerin çıktıdaki *ilk* konumunu tutar: `c[3] = 1`, ilk 3'ün `b[1]`'de olduğunu söyler.



Şekil 19.10: L4 sonunda `b` sıralıdır. Her yazmadan sonra azaltılan sayaçlar artık her değerin ilk konumunu gösterir.

İki 3'e dikkat edelim. `a[5]`'teki 3 önce işlendi ve `b[2]`'ye yazıldı. `a[0]`'daki 3 sonra işlendi ve sayaç bir azaldığı için `b[1]`'e, yani bir sola yazıldı. Girdide önce gelen 3 (`a[0]`) çıktıda da önce gelir. Algoritma kararlıdır ve bunun nedeni L4'ün sağdan sola gezmesidir: aynı değerli öğelerden en sağdaki önce yerleşir ve en sağdaki yeri alır.

L4 soldan sağa yazılsaydı çıktı yine sıralı olurdu, ama eşitlerin sırası tersine dönerdi. Bu durumda `a[0]`'daki 3 önce işlenir ve `b[2]`'ye, `a[5]`'teki 3 ise `b[1]`'e yazılır. Sayılar tek başına sıralanıyorsa fark görünmez. Saymalı sıralama kayıtları ya da basamak sıralamasının bir geçişini sıralıyorsa fark sonucu bozar.

> **Sık yapılan hata**
>
> Değer aralığını denetlememek. Girdide negatif bir değer ya da `k`'dan büyük bir değer varsa `c[a[j]]++`, `c` dizisinin dışına yazar. C bunu yakalamaz: program ya sessizce başka bir değişkeni bozar ya da çöker. Yordamın sözleşmesi "bütün değerler *0* … *k* aralığındadır" biçimindedir. Veri bu sözleşmeye uymayabiliyorsa önce en küçük ve en büyük değer bulunmalı, değerler en küçük değer kadar kaydırılmalı ve `k` buna göre seçilmelidir.

19.6.4 Analiz ve sınırlar

Yordamın dört döngüsü vardır ve hiçbiri iç içe değildir. Maliyetleri Tablo 19.12'de toplanmıştır.

Döngü	İş	Maliyet
L1	`c`'yi sıfırla	*O(k)*
L2	say	*O(n)*
L3	kümülatif toplam	*O(k)*
L4	yerleştir	*O(n)*

Tablo 19.12: Saymalı sıralamanın döngüleri; *n = r + 1*.

Toplam *O(n + k)*'dir. *k = O(n)* ise, yani değer aralığı öğe sayısından çok büyük değilse, saymalı sıralama doğrusal zamanda çalışır. Ek bellek `c` için *k + 1* ve `b` için *n* öğedir, toplam *O(n + k)*. Algoritma yerinde değildir.

Doğrusal zaman bir bedelle gelir: veri aralığı bilinmeli ve küçük olmalıdır. Aralık büyürse k terimi baskın çıkar. 10 sayı sıralanacak, ama değerler 0 ile 1.000 arasında olabiliyorsa L1 ve L3 binlik döngülerdir. $n = 10$ için $k = 1.000 = n^3$'tür; "doğrusal" algoritma bu girdide n^3'lük bir algoritmaya dönüşür. Aynı 10 sayıyı eklemeli sıralama en çok 45 karşılaştırmayla sıralar. 32 bitlik rastgele tam sayılar için ise $k \approx 4 \cdot 10^9$'dur ve sayaç dizisi 16 GB tutar. Saymalı sıralama bu yüzden yaş, not, gün, harf kodu gibi dar aralıklı veriler içindir. Geniş aralıklı tam sayılar basamak sıralamasıyla sıralanır: sayı basamaklara bölünür ve her basamak dar aralıklı bir saymalı sıralamayla işlenir.

Algoritmanın iki geçişli yapısı başka yerlerde de karşımıza çıkar: ilk geçişte frekanslar sayılır, ikinci geçişte her öğe hesaplanan yerine doğrudan yazılır. Seyrek matrislerin hızlı devrik alma yöntemi (*fast transpose*) tam olarak budur. Önce her sütunda kaç sıfırdan farklı terim olduğu sayılır. Sayıların kümülatif toplamı, her sütunun devrik matristeki başlangıç konumunu verir. İkinci geçişte her terim doğrudan yerine yazılır ve bütün iş tek bir çıktı dizisiyle *O(sütun sayısı + terim sayısı)* zamanda biter.

19.7 Shell sıralaması

19.7.1 Fikir ve kod

Eklemeli sıralamanın zayıflığı, bir öğeyi her adımda yalnız bir konum ilerletmesidir. Dizinin sonundaki küçük bir öğe başa varmak için $n - 1$ kaydırma ister. Ters çiftler açısından bakılınca her kaydırma tek bir ters çift giderir ve ters sıralı dizide $n(n - 1)/2$ ters çift vardır. Donald Shell 1959'da basit bir çare önerdi: önce birbirinden uzak öğeleri sırala, sonra aralığı küçült.

Bir h sayısı seçilsin. Dizi, aralarında h konum bulunan öğelerden oluşan h ayrı zincire bölünür: *a[0], a[h], a[2h], ...* birinci zincir, *a[1], a[h+1], ...* ikinci zincirdir. Her zincir eklemeli sıralamayla ayrı ayrı sıralanır. Sonuçta dizi ***h*-sıralı** olur: her öğe kendinden h sonraki öğeden büyük değildir. Bu adım büyük bir h ile başlar ve h her adımda küçültülür. Son adımda $h = 1$'dir; bu, sıradan eklemeli sıralamadır.

```c
void shell_sort(int a[], int n) {
    for (int h = n / 2; h > 0; h /= 2) // aralık
        for (int i = h; i < n; i++) {
            int v = a[i], j = i;
            while (j >= h && a[j - h] > v) {
                a[j] = a[j - h];    // h geri kaydır
                j -= h;
            }
            a[j] = v;
        }
}
```

- **Satır 2.** Aralık $n/2$'den başlar ve her adımda yarıya iner: $n = 8$ için 4, 2, 1; $n = 1.000$ için 500, 250, 125, 62, 31, 15, 7, 3, 1. Bu, Shell'in kendi önerdiği aralık dizisidir. Tam sayı bölmesi sayesinde dizi her zaman 1 ile biter.
- **Satır 3–9.** İç kısım, eklemeli sıralamanın 1 yerine h adımla yazılmış hâlidir. `v = a[i]` kenara alınır; `j`, h'şer adımla geri gider; `v`'den büyük öğeler h konum ileri kaydırılır. Koşuldaki `j >= h`, `a[j - h]`'nin dizinin içinde olmasını sağlar.

Kodda zincirler ayrı ayrı sıralanmaz. `i` döngüsü h'den $n - 1$'e kadar sırayla gider ve her öğeyi kendi zincirinde yerine ekler. Zincirler böylece iç içe, öğe öğe ilerler. Sonuç, zincirleri tek tek sıralamakla aynıdır, çünkü her zincirin eklemeli sıralaması öğelerini soldan sağa sırayla işler.

Algoritmanın doğruluğu son adımdan gelir: $h = 1$ turu eksiksiz bir eklemeli sıralamadır ve her diziyi sıralar. Önceki turlar yalnız bu son turun işini azaltır. Hangi aralık dizisi seçilirse seçilsin, 1 ile bittiği sürece sonuç doğrudur. Aralık dizisi yalnız hızı belirler.

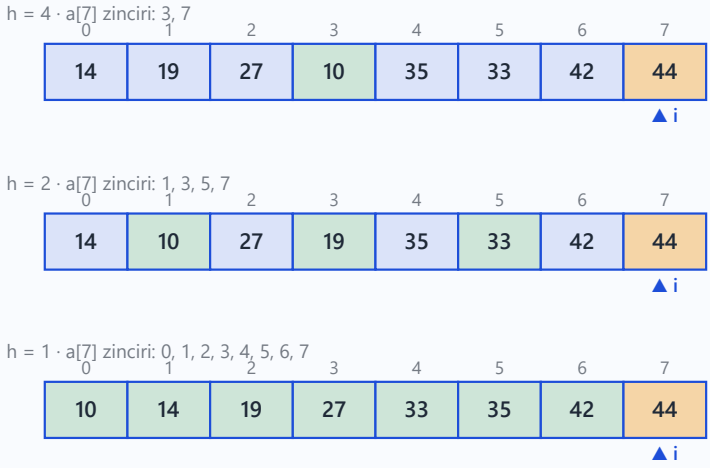
19.7.2 Adım adım izleme

`shell_sort(a, 8)` çağrısı *a = [35, 33, 42, 10, 14, 19, 27, 44]* üzerinde izlenir. Tablo 19.13 her aralığın sonundaki diziyi ve yapılan işi verir.

h	Zincirler (indisler)	Olaylar	Karş.	Kayd.	Aralık sonu
4	{0,4} {1,5} {2,6} {3,7}	14, 19, 27 birer kez 4 geri kaydı; 44 yerinde	4	3	[14, 19, 27, 10, 35, 33, 42, 44]
2	{0,2,4,6} {1,3,5,7}	10 bir kez 2 geri kaydı; ötekiler yerinde	6	1	[14, 10, 27, 19, 35, 33, 42, 44]
1	tek zincir	10, 19 ve 33 birer kez 1 geri kaydı	9	3	[10, 14, 19, 27, 33, 35, 42, 44]

h	Zincirler (indisler)	Olaylar	Karş.	Kayd.	Aralık sonu
Toplam			**19**	**7**	

Tablo 19.13: `shell_sort(a, 8)` izlemesi, *a = [35, 33, 42, 10, 14, 19, 27, 44]*. Kaydırma bir öğenin *h* konum ileri taşınmasıdır.



Şekil 19.11: Shell sıralamasında üç aralığın sonu. Üstte *h = 4*: dört zincirin her biri sıralı (son işlenen zincir 3, 7). Ortada *h = 2*: iki zincir sıralı. Altta *h = 1*: dizi sıralı.

h = 4 turunda *i = 4* için `v = 14` alınır. `a[0] = 35 > 14` olduğundan 35, `a[4]`'e kayar ve `j = 0` olur. `j >= 4` yanlıştır; 14, `a[0]`'a yazılır. Tek bir kaydırmayla 14 dört konum sola gitti. Aynı kaydırma 35'i de dört konum sağa götürdü. Eklemeli sıralamada bu iş dört komşu adım isterdi. *i = 5* ve *i = 6*'da 19 ve 27 aynı biçimde 33 ve 42'nin önüne geçer. *i = 7*'de `a[3] = 10 ≤ 44`'tür ve 44 yerinde kalır.

h = 2 turunda dizi iki zincire ayrılır: çift indisler (14, 27, 35, 42) zaten sıralıdır. Tek indislerde (19, 10, 33, 44) yalnız 10, 19'un önüne geçer. *h = 1* turu başladığında dizi *[14, 10, 27, 19, 35, 33, 42, 44]*'tür. Her öğe son yerinden en çok bir konum uzaktadır; eklemeli sıralama üç kısa kaydırmayla işi bitirir.

Aynı diziyi doğrudan eklemeli sıralama 18 karşılaştırma ve 13 kaydırmayla sıralar; dizide 13 ters çift vardır. Shell sıralaması 19 karşılaştırma ve 7 kaydırma yaptı. Bu kadar küçük bir dizide karşılaştırmada kazanç yoktur, ama kaydırma sayısı yarıya inmiştir. 7 kaydırma 13 ters çifti giderdi: uzak kaydırmalar tek hamlede birden çok ters çift yok eder. Örneğin *h = 4*'teki ilk kaydırma 35'i 33, 42 ve 10'un üzerinden atlattı. Dizi büyüdükçe bu etki belirginleşir.

19.7.3 Aralık dizisi ve karmaşıklık

Shell sıralamasının çalışma süresi aralık dizisine bağlıdır ve tam analizi zordur. Bilinen sonuçların bir kısmı şöyledir:

- Shell'in *n/2, n/4, ..., 1* dizisinde en kötü durum $\Theta(n^2)$'dir. Sorun, *n* 2'nin kuvveti olduğunda bütün aralıkların son tura kadar çift olmasıdır. Tek ve çift konumlar 1'e kadar hiç karşılaşmaz ve kötü bir girdide son tur yine n^2 mertebesinde iş yapar.
- Knuth'un *1, 4, 13, 40, 121, ...* dizisinde ($h \leftarrow 3h + 1$) en kötü durum $O(n^{3/2})$'dir. Ardışık aralıklar ortak bölen paylaşmadığı için zincirler birbirine karışır.
- Daha karmaşık dizilerle (örneğin Pratt'ın $2^p 3^q$ biçimindeki aralıkları) en kötü durum $O(n \lg^2 n)$'ye iner. Ortalama durum için kapalı bir formül bilinmez.

Shell sıralamasının işe yaramasının bir nedeni şu özelliktir: *h*-sıralı bir dizi *k*-sıralandığında *h*-sıralı kalır. Yani sonraki turlar önceki turların kazandırdığı düzeni bozmaz, üzerine ekler. Bu özelliğin kanıtı bu notun kapsamı dışındadır; Knuth'un kitabında verilir.

Tablo 19.14, iki aralık dizisini *n = 1.000* öğede karşılaştırır. Değerler sayaçlı bir C programıyla ölçüldü.

Aralık dizisi	Sıralı	Ters sıralı	Rastgele
Shell: 500, 250, ..., 1	8.006 / 0	11.716 / 4.700	15.218 / 7.700
Knuth: 364, 121, 40, 13, 4, 1	5.457 / 0	8.550 / 3.920	13.693 / 8.646

Aralık dizisi	Sıralı	Ters sıralı	Rastgele
Eklemeli sıralama (karşılaştırma için)	999 / 0	499.500 / 499.500	241.305 / 240.312

Tablo 19.14: *n = 1.000* için Shell sıralaması: karşılaştırma / kaydırma. Rastgele girdi bir karıştırma ile üretildi; başka bir rastgele girdide sayılar biraz değişir.

Rastgele girdide eklemeli sıralama 240 binden fazla kaydırma yapar; her iki Shell sürümü 10 binin altında kalır. Sıralı girdide ise eklemeli sıralama rakipsizdir, çünkü Shell her aralık için bütün diziyi bir kez gezer: *n = 1.000* ve Shell aralıkları için *500 + 750 + 875 + 938 + 969 + 985 + 993 + 997 + 999 = 8.006* karşılaştırma. Bu, aralık sayısı kadar *n* mertebesinde iştir, yani *O(n lg n)*.

Shell sıralaması yerindedir: eklemeli sıralama kadar bellek kullanır. Kararlı değildir. Uzak kaydırmalar eşit öğeleri birbirinin üzerinden atlatabilir. En küçük örnek dört öğelidir: *[2a, 5, 2b, 2c]*. *h = 2* turunda *i = 3* için `v = 2c` alınır; `a[1] = 5 > 2` olduğundan 5 sağa kayar ve 2c, `a[1]` 'e yazılır. Dizi *[2a, 2c, 2b, 5]* olur. *h = 1* turu eşit öğeleri yerinden oynatmaz. 2c, 2b'nin önüne geçmiştir.

Kod kısadır, ek bellek istemez ve orta boy dizilerde n^2'li algoritmalardan çok hızlıdır. Bu üç özellik Shell sıralamasını gömülü sistemlerde ve küçük kütüphanelerde sevilen bir seçenek yapar. Verinin özellikleri biliniyorsa Shell sıralaması kova sıralamasıyla birlikte de kullanılabilir. Kovalara dağıtılan öğeler her kovada ayrıca sıralanır; bu iş için kısa ve yerinde bir algoritma uygundur.

19.8 Kova sıralaması

19.8.1 Fikir

Kova sıralaması (*bucket sort*), saymalı sıralamanın fikrini gerçel sayılara taşır. Girdinin *[0, 1)* aralığında *n* gerçel sayı olduğu ve sayıların bu aralığa düzgün dağıldığı varsayılır. Aralık *n* eşit parçaya, yani *n* **kovaya** bölünür. Her sayı kendi kovasına atılır. Düzgün dağılımda her kovaya ortalama bir sayı düşer. Kovalar küçük olduğu için kendi içlerinde eklemeli sıralamayla ucuza sıralanır. Son olarak kovalar sırayla birleştirilir.

Şekil 19.12, bu yöntemi 10 sayılık bir girdi üzerinde gösterir. 10 kova vardır ve her kova 0,1 genişliğindedir. *x* değeri *⌊10 · x⌋* numaralı kovaya gider: 0,78 → 7, 0,17 → 1, 0,39 → 3.



Şekil 19.12: Kova sıralaması. Üstte girdi on kovaya dağıtılmıştır; kovaların içi henüz geliş sırasındadır (1. kovada önce .17, sonra .12). Altta kovalar kendi içinde sıralanmış ve soldan sağa okunarak çıktı oluşturulmuştur.

19.8.2 C ile gerçekleme

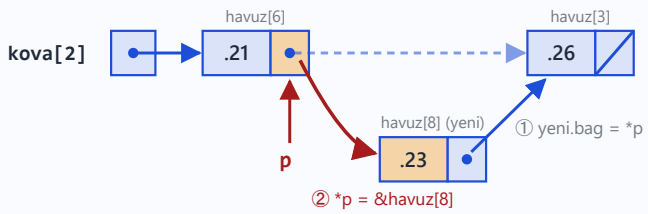
Her kova bir bağlı listedir. Kova dizisi bu listelerin baş işaretçilerini tutar. Aşağıdaki gerçekleme dağıtma ile kova içi sıralamayı birleştirir: her sayı kovasına atılırken listede sıralı yerine eklenir. Bu, kova içi eklemeli sıralamanın ta kendisidir.

```c
typedef struct kdugum { double x; struct kdugum *bag; } kdugum;

int bucket_sort(double a[], int n) {        /* 0 <= a[i] < 1 */
    kdugum **kova = calloc(n, sizeof *kova);
    kdugum *havuz = malloc(n * sizeof *havuz);
    if (!kova || !havuz) { free(kova); free(havuz); return -1; }
    for (int i = 0; i < n; i++) {           /* dağıt + sıralı ekle */
        kdugum **p = &kova[(int)(a[i] * n)];
        while (*p && (*p)->x <= a[i]) p = &(*p)->bag;
        havuz[i].x = a[i]; havuz[i].bag = *p; *p = &havuz[i];
    }
    int k = 0;                              /* kovaları birleştir */
    for (int b = 0; b < n; b++)
        for (kdugum *q = kova[b]; q; q = q->bag) a[k++] = q->x;
    free(kova); free(havuz);
    return 0;
}
```

- **Satır 1.** Liste düğümü: değer `x` ve sonraki düğüme bağ `bag`. Yapı kendine gönderme yapan bir `struct`'tır.
- **Satır 4.** `kova`, *n* tane `kdugum *` işaretçisinden oluşan bir dizidir. `calloc` belleği sıfırlar; bütün kovalar `NULL`, yani boş liste olarak başlar. `malloc` kullanılsaydı işaretçiler rastgele adresler taşırdı ve ilk `(*p)->x` okuması tanımsız davranış olurdu.
- **Satır 5.** Düğümler tek tek `malloc` ile değil, tek bir `havuz` dizisinden alınır. `havuz[i]`, `a[i]`'nin düğümüdür. *n* ayrı ayırma yerine iki ayırma yapılır ve bellek tek bir `free` çiftiyle geri verilir. Ayırma maliyeti düşer, sızıntı riski azalır.
- **Satır 6.** İki ayırmadan biri başarısızsa ikisi de bırakılır. `free(NULL)` güvenlidir; hangisinin başarısız olduğunu ayrıca sınamaya gerek yoktur.
- **Satır 8.** `p`, bir *işaretçinin işaretçisidir* (`kdugum **`). Başlangıçta kova dizisindeki ilgili hücreyi, yani listenin baş işaretçisini gösterir. Kova numarası `(int)(a[i] * n)`'dir. *0 ≤ a[i] < 1* olduğundan bu sayı 0 ile *n − 1* arasındadır.
- **Satır 9.** `p`, listede ilerler: gösterdiği işaretçi boş değilse ve o düğümün değeri `a[i]`'den küçük ya da eşitse `p`, o düğümün `bag` alanını göstermeye başlar. Döngü bittiğinde `*p`, yeni düğümün önüne geçeceği düğümü (ya da `NULL`'ı) gösterir; `p` ise değiştirilecek bağın adresidir.
- **Satır 10.** Yeni düğüm doldurulur, bağı `*p`'nin gösterdiği düğüme yöneltilir ve `*p` yeni düğümü gösterir. Ekleme tamamlanır.
- **Satır 12–14.** Kovalar 0'dan *n − 1*'e sırayla gezilir; her listenin değerleri sırayla `a`'ya geri yazılır.
- **Satır 15.** İki ayırma geri verilir. Düğümler ayrı ayrı ayrılmadığı için ayrı ayrı bırakılmaz.

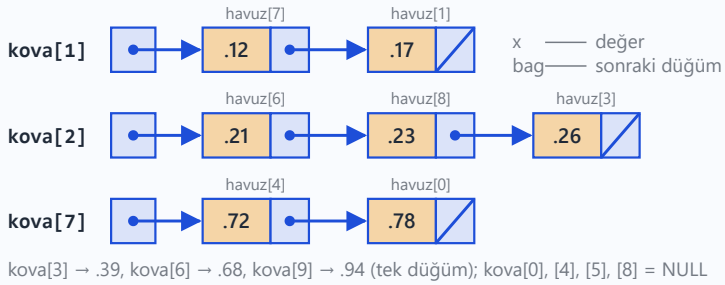
İşaretçinin işaretçisi bu kodun en ince noktasıdır. Tek yıldızlı bir işaretçiyle listede gezilseydi, listenin başına ekleme özel bir durum olurdu. Başa eklenen düğüm kova dizisindeki baş işaretçiyi değiştirir. Ortaya eklenen düğüm ise önceki düğümün `bag` alanını değiştirir. `p` "değiştirilecek işaretçinin adresi"ni tuttuğu için iki durum aynı satırla yapılır. Şekil 19.13 bunu *0,23*'ün eklenişinde gösterir.



while: .21 ≤ .23 → p = &(.21 düğümü)->bag; .26 ≤ .23 yanlış → dur
sonuç: kova[2] → .21 → .23 → .26 → NULL (bağlar değişti, düğümler yerinde kaldı)

Şekil 19.13: 0,23'ün (`havuz[8]`) 2 numaralı kovaya eklenmesi. Kovada 0,21 ve 0,26 vardır. `p` önce `&kova[2]`'dir; *0,21 ≤ 0,23* olduğu için 0,21 düğümünün `bag` alanına ilerler ve *0,26 > 0,23*'te durur. Yeni düğümün bağı 0,26'yı gösterir (①), sonra `*p` yeni düğümü gösterir (②).

Yordamın sonunda bellekteki durum Şekil 19.14'te görülür. Düğümler `havuz` dizisinde girdi sırasıyla durur: 0,12'nin düğümü `havuz[7]`'dir, çünkü 0,12 girdinin 7 indisindedir. Bağlar ise her kovanın içini sıralı bir zincire dizer. Bağlı listenin gücü buradadır: öğeler bellekte yer değiştirmez, yalnız bağlar değişir.



Şekil 19.14: `bucket_sort` birleştirme döngüsünden önce bellek. Kova dizisinin her hücresi bir listenin başıdır. Düğümlerin üstündeki etiketler, düğümün `havuz` dizisindeki yeridir; çapraz çizgi `NULL` bağdır.

Karşılaştırma `<=` ile yapıldığı için eşit bir değer, listedeki eşitlerinin *sonuna* eklenir. Girdide önce gelen önce kalır; kova sıralaması bu gerçeklemede kararlıdır. Karşılaştırma `<` olsaydı yeni öğe eşitlerin önüne girerdi ve kararlılık kaybolurdu.

19.8.3 Analiz

Dağıtma ve birleştirme $O(n)$'dir: her öğe bir kez kovaya atılır ve bir kez geri yazılır. Asıl soru, kovaların içini sıralamanın maliyetidir. Kova i'ye n_i öğe düşerse eklemeli sıralama o kovada $O(n_i^2)$ iş yapar. Toplam maliyet $O(n + \Sigma\, n_i^2)$'dir.

Girdi düzgün dağılmışsa her öğe her kovaya $1/n$ olasılıkla düşer. Bu durumda bir kovanın beklenen öğe sayısı 1'dir ve n_i^2'nin beklenen değeri $2 - 1/n$'dir (Cormen ve arkadaşlarının kitabında kanıtlanır). n kova üzerinden toplam $n \cdot (2 - 1/n) = 2n - 1$ eder. Ortalama çalışma süresi $O(n)$'dir.

Düzgün dağılım varsayımı bozulursa sonuç da bozulur. En kötü durumda bütün öğeler aynı kovaya düşer. Örneğin 5 sayı 0,90, 0,91, 0,92, 0,93 ve 0,94 ise $\lfloor 5x \rfloor$ hepsi için 4'tür. Tek kovada 5 öğe eklemeli sıralanır ve maliyet $O(n^2)$'ye çıkar. Ek bellek kova dizisi ve düğümler için $O(n)$'dir; algoritma yerinde değildir.

Kova sıralaması da karşılaştırma kullanır, ama yalnız kovaların içinde. Öğenin hangi kovaya gideceği bir hesapla, $\lfloor n \cdot x \rfloor$ ile bulunur. Bu hesap, öğeyi bütün öteki öğelerle tek seferde karşılaştırmak gibidir. Algoritmanın karşılaştırmalı sıralamanın alt sınırına takılmamasının nedeni budur.

19.9 Karşılaştırmalı sıralamanın alt sınırı

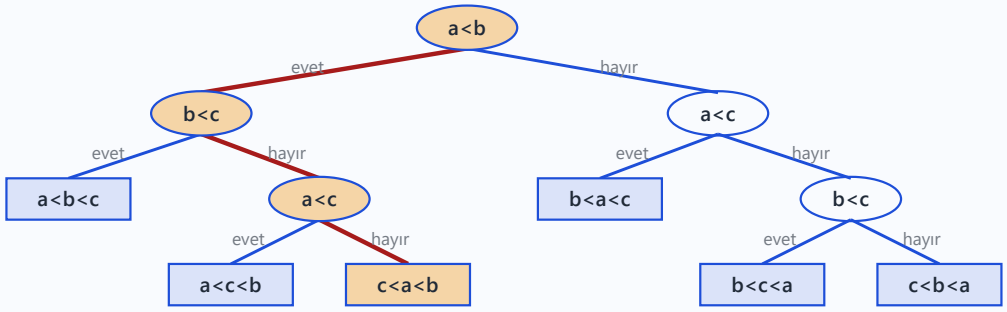
19.9.1 Karar ağacı

Seçimli, eklemeli, kabarcık ve Shell sıralaması n^2 ile $n^{3/2}$ arasında karşılaştırma yaptı. Daha iyisi mümkün müdür ve ne kadar iyisi? Bu soruyu bir algoritma için değil, karşılaştırmaya dayanan *bütün* algoritmalar için cevaplamak gerekir. Bunun aracı karar ağacıdır.

Karşılaştırmalı bir sıralama algoritması girdi hakkında bilgiyi yalnız `kucuk` çağrılarından edinir. Her çağrının iki sonucu vardır: evet ya da hayır. Algoritmanın bir sonraki adımı bu cevaba bağlıdır. Algoritmanın bütün olası çalışmaları bir ikili ağaç biçiminde çizilebilir. Bu ağaca **karar ağacı** (*decision tree*) denir.

- Her iç düğüm bir karşılaştırmadır (çizimde elips): "$a < b$?".
- Düğümün iki çocuğu vardır: soldaki "evet" cevabının, sağdaki "hayır" cevabının ardından gelen adımdır.
- Her yaprak bir karardır (çizimde dikdörtgen): girdinin hangi dizilişte olduğu, yani öğelerin hangi sırayla yazılacağı.

Belli bir girdi için algoritma kökten başlar, her karşılaştırmanın cevabına göre sola ya da sağa iner ve bir yaprakta durur. Kökten yaprağa giden yolun uzunluğu, o girdide yapılan karşılaştırma sayısıdır. Şekil 19.15, üç öğeyi (a, b, c) sıralayan bir karar ağacını gösterir. $n = 3$ için $3! = 6$ diziliş ve dolayısıyla 6 yaprak vardır.



Şekil 19.15: Üç öğe için karar ağacı. Elipsler karşılaştırma, dikdörtgenler kararıdır. Kalın yol *a = 5, b = 8, c = 2* girdisinin izlediği yoldur: 3 karşılaştırmadan sonra *c < a < b* kararına varılır.

Girdi *a = 5, b = 8, c = 2* olsun. Kökteki soru "*a < b*?"'dir: *5 < 8*, cevap evet. Sol çocukta "*b < c*?" sorulur: *8 < 2*, cevap hayır, yani *c*, *b*'den önce gelmelidir. Bu noktada *a < b* ve *c < b* bilinir, ama *a* ile *c*'nin sırası bilinmez. Üçüncü soru "*a < c*?"'dir: *5 < 2*, cevap hayır. Yaprak *c < a < b*'dir ve çıktı 2, 5, 8 olur. Bu girdi için 3 karşılaştırma yapıldı.

Ağacın bazı yolları daha kısadır. *a < b* ve *b < c* ise geçişlilik yoluyla *a < c* de bilinir; algoritma iki karşılaştırmayla *a < b < c* kararını verir. En kötü durumdaki karşılaştırma sayısı en uzun yolun uzunluğudur, yani ağacın **yüksekliğidir**. Bu ağacın yüksekliği 3'tür.

Her karşılaştırmalı sıralama algoritması, her *n* için böyle bir ağaca karşılık gelir. Eklemeli sıralamanın ağacı bir biçimdedir, seçimli sıralamanınki başka bir biçimde; ama her ağaç aynı koşulu sağlamak zorundadır. Bu koşul, alt sınırın kaynağıdır.

19.9.2 Alt sınır teoremi

Teorem 19.3 (Karşılaştırmalı sıralamanın alt sınırı)

n farklı öğeyi sıralayan her karar ağacının derinliği (düzey sayısı) en az *⌈lg(n!)⌉ + 1*'dir. Yani karşılaştırmaya dayanan her sıralama algoritması en kötü durumda en az *⌈lg(n!)⌉* karşılaştırma yapar. Bu sayı *Θ(n lg n)*'dir.

İspat. **Yaprak sayısı.** Girdinin *n!* dizilişinin her biri farklı bir çıktı ister: iki farklı diziliş aynı yaprağa varsaydı algoritma ikisinden birini yanlış sıralardı. Bu yüzden ağacın en az *n!* yaprağı vardır.

Yükseklik ile yaprak ilişkisi. Yüksekliği *h* olan bir ikili ağacın en çok 2^h yaprağı olabilir. Kök tek düğümdür; her düzeyde düğüm sayısı en çok iki katına çıkar ve *h* karşılaştırmadan sonra en çok 2^h farklı yere varılır.

Sonuç. İki gözlem birleşince $2^h \geq n!$, yani $h \geq \lg(n!)$ olur. *h* bir tam sayı olduğu için $h \geq \lceil \lg(n!) \rceil$'dir. Düzey sayısı yükseklikten bir fazladır: $\lceil \lg(n!) \rceil + 1$.

Büyüme hızı. *n!*'in son *n/2* çarpanının her biri en az *n/2*'dir; bu yüzden $n! \geq (n/2)^{n/2}$ ve $\lg(n!) \geq (n/2) \cdot \lg(n/2)$, yani $\Omega(n \lg n)$'dir. Ayrıca $n! \leq n^n$ olduğundan $\lg(n!) \leq n \lg n$'dir. İkisi birlikte $\lg(n!) = \Theta(n \lg n)$ verir.

Daha keskin bir yaklaşım Stirling formülünden gelir:

$$\lg(n!) = n \lg n - n \lg e + \Theta(\lg n) \approx n \lg n - 1{,}4427\, n$$

Burada $\lg e = 1 / \ln 2 \approx 1{,}4427$'dir. Tablo 19.15 alt sınırı, Stirling yaklaşımını ve temel algoritmaların en kötü durumunu yan yana koyar. Değerler C programında *lg(n!) = lg 2 + lg 3 + … + lg n* toplamıyla hesaplandı.

n	*⌈lg n!⌉*	*n lg n − 1,44n*	*n(n − 1)/2*
3	3	0,4	3
4	5	2,2	6
5	7	4,4	10
10	22	18,8	45
20	62	57,6	190

n	*⌈lg n!⌉*	*n lg n − 1,44n*	*n(n − 1)/2*
100	525	520,1	4.950

Tablo 19.15: Karşılaştırmalı sıralamanın alt sınırı ve n^2'li algoritmaların en kötü durumu.

Küçük *n*'lerde Stirling yaklaşımı alt sınırın epey altında kalır, çünkü atılan *Θ(lg n)* terimi henüz önemlidir. *n = 100*'de fark 5'e iner. Tablonun son iki sütunu arasındaki uçurum da dikkat çekicidir: 100 öğe için alt sınır 525, seçimli sıralamanın yaptığı ise 4.950 karşılaştırmadır. Aradaki boşluk, daha iyi algoritmalar için yer olduğunu gösterir. Birleştirme sıralaması ve heap sort bu boşluğu kapatır: en kötü durumda *O(n lg n)* karşılaştırma yaparlar ve alt sınıra sabit bir çarpan kadar yakındırlar.

Teorem bundan sonraki bütün sıralama algoritmalarını karşılaştırmak için bir ölçüt, bir mihenk taşı verir. Karşılaştırmalı bir algoritma *O(n lg n)*'ye ulaşmışsa, mertebe bakımından daha iyisi yoktur; yalnız sabit çarpanlar ve pratik özellikler (bellek, kararlılık, önbellek dostluğu) iyileştirilebilir. Aynı sınır ortalama durum için de geçerlidir: bir karar ağacında yaprakların ortalama derinliği de en az *lg(n!)*'dir.

Teoremin bir varsayımı vardır: algoritma girdi hakkında bilgiyi yalnız karşılaştırmayla edinir. Saymalı, kova ve basamak sıralaması bu varsayımı çiğner. `c[a[j]]++` satırı `a[j]`'yi başka bir öğeyle karşılaştırmaz; değerin kendisini indis olarak kullanır ve bu tek işlemle *k + 1* yönlü bir karar verir. Karar ağacı ikili olmaktan çıkar ve alt sınır geçerliliğini yitirir. Doğrusal zamanın bedeli de buradan gelir: değerin indis olarak kullanılabilmesi için aralığın bilinmesi gerekir.

19.10 Algoritmaları karşılaştırmak

19.10.1 Aynı girdide sayım

Formüller algoritmaların büyüme hızını verir, ama sabit çarpanları ve girdinin biçimini gizler. Dört karşılaştırmalı algoritmanın aynı 1.000 öğelik girdilerde yaptığı iş Tablo 19.16'da yan yana durur. Her hücrede önce karşılaştırma, sonra taşıma sayısı yazar. Taşıma seçimli sıralamada takas, eklemeli ve Shell sıralamasında kaydırma, kabarcık sıralamasında takastır.

Algoritma	Sıralı	Ters sıralı	Rastgele
Seçimli	499.500 / 999	499.500 / 999	499.500 / 999
Eklemeli	999 / 0	499.500 / 499.500	247.876 / 246.884
Kabarcık (erken duran)	999 / 0	499.500 / 499.500	499.035 / 246.884
Shell (*n/2, n/4, ...*)	8.006 / 0	11.716 / 4.700	14.760 / 7.254

Tablo 19.16: *n = 1.000* için karşılaştırma / taşıma sayıları. Rastgele sütun Python benzetimiyle sayıldı (tohum 19).

Sıralı ve ters sıralı sütunlar formüllerden doğrudan çıkar ve bu notun C programıyla da birebir aynı bulundu. Rastgele sütunun kendisi seçilen karıştırmaya bağlıdır. Bu notun C programı başka bir karıştırmayla şu değerleri ölçtü: eklemeli 241.305 / 240.312, erken duran kabarcık 499.247 / 240.312 (977 tur), Shell 15.218 / 7.700. Sayılar biraz farklıdır, ama ilişkiler aynıdır. İki sayımda da eklemeli sıralamanın kaydırma sayısı ile kabarcık sıralamasının takas sayısı eşittir, çünkü ikisi de girdinin ters çift sayısıdır. Teoremin öngördüğü *n(n − 1)/4 = 249.750* değeri her iki ölçüme de yakındır.

- **Seçimli** girdiye kördür: her durumda 499.500 karşılaştırma ve 999 takas yapar.
- **Eklemeli** sıralı girdide 999 karşılaştırmayla biter; rastgele girdide $\approx n^2/4$, ters girdide $n^2/2$ karşılaştırma yapar.
- **Kabarcık** erken dursa bile rastgele girdide neredeyse en kötü durum kadar karşılaştırma yapar: 1.000 öğede yalnız son 20–30 tur atlanır.
- **Shell** rastgele girdide eklemeli sıralamaya göre karşılaştırmaları yaklaşık on altı kat, kaydırmaları otuz kattan fazla azaltır. Uzak kaydırmalar ters çiftleri hızla yok eder.

Sayımlar bir de süre ölçümüyle desteklenebilir. Tablo 19.17 aynı rastgele girdileri sıralamanın milisaniye cinsinden süresini verir. Süreler makineye, derleyiciye ve derleme seçeneklerine göre değişir; burada önemli olan mutlak değerler değil, *n* iki katına çıktığında sürenin nasıl değiştiğidir.

n	Seçimli	Eklemeli	Kabarcık (erken duran)	Shell
5.000	39	23	48	1
10.000	159	106	174	2
20.000	719	474	1.018	4
40.000	2.941	1.578	4.712	13

Tablo 19.17: Rastgele girdide süre (ms). Ölçüm: zig cc `-O2`, Windows, tek çalıştırma. Değerler makineye göre değişir.

n her ikiye katlandığında üç temel algoritmanın süresi yaklaşık dört katına çıkar; bu, n^2'li büyümenin işaretidir. Shell sıralaması ise 40.000 öğeyi 13 milisaniyede sıralar. Eklemeli sıralama, karşılaştırma sayısı seçimlininkinin yarısı olduğu için yaklaşık iki kat hızlıdır. Kabarcık sıralaması en yavaşıdır: seçimli kadar karşılaştırma yapar, üstüne eklemeli kadar taşıma yapar ve her taşıması üç atamalı bir takastır.

19.10.2 Karmaşıklık özeti

Algoritma	En iyi	Ortalama	En kötü	Ek bellek	Kararlı	Yerinde
Seçimli	$n^2/2$	$n^2/2$	$n^2/2$	$O(1)$	hayır	evet
Eklemeli	$O(n)$	$\approx n^2/4$	$n^2/2$	$O(1)$	evet	evet
Kabarcık (erken duran)	$O(n)$	$\approx n^2/2$	$n^2/2$	$O(1)$	evet	evet
Shell (Knuth aralıkları)	$O(n \lg n)$	—	$O(n^{3/2})$	$O(1)$	hayır	evet
Saymalı	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(n + k)$	evet	hayır
Kova (düzgün dağılım)	$O(n)$	$O(n)$	$O(n^2)$	$O(n)$	evet*	hayır

Tablo 19.18: Bu notun algoritmalarının özeti. Seçimli, eklemeli ve kabarcık satırlarındaki sayılar karşılaştırma sayısıdır.

Tablodaki yıldız, kova sıralamasının kararlılığının kova içinde kullanılan sıralamaya bağlı olduğunu anlatır; buradaki gerçekleme `<=` ile eklediği için kararlıdır. Shell sıralamasının ortalaması aralık dizisine bağlıdır ve kapalı bir formülle bilinmez. Karşılaştırmalı her algoritma için alt sınır $\lceil \lg n! \rceil \approx n \lg n$'dir; tablodaki ilk dört satırın hiçbiri en kötü durumda bu sınıra ulaşmaz.

19.10.3 Hangi durumda hangisi?

"En iyi sıralama algoritması" diye bir şey yoktur; her algoritma bir duruma uygundur. Seçim, verinin büyüklüğüne, biçimine ve pahalı olan işleme bağlıdır.

- **Takas ya da yazma pahalıysa seçimli sıralama.** Tam $n - 1$ takas yapar. Büyük kayıtlar ve flash bellek gibi yazmanın pahalı olduğu ortamlar için uygundur.
- **Veri küçük ya da neredeyse sıralıysa eklemeli sıralama.** Kararlıdır, yerindedir ve sıralıya yakın veride $O(n)$'e iner. Hızlı sıralama ve birleştirme sıralamasının pratik sürümleri küçük parçaları eklemeli sıralamaya bırakır.
- **Öğretim ve sıralılık testi için kabarcık sıralaması.** Erken duran sürüm bir dizinin sıralı olduğunu tek geçişte anlar. Üretim kodunda nadiren yer alır.
- **Değerler küçük bir tam sayı aralığındaysa saymalı sıralama.** $O(n + k)$ ve kararlıdır. k, n'den çok büyükse bellek ve zaman boşa gider.
- **Orta boy veri ve ek bellek yoksa Shell sıralaması.** Birkaç satırlık kodla n^2'li algoritmalardan belirgin biçimde hızlıdır.
- **Veri $[0, 1)$'de düzgün dağılmışsa kova sıralaması.** Ortalama $O(n)$'dir, ama dağılım bozuksa n^2'ye kadar yavaşlar.

19.11 Sık yapılan hatalar

1. **Seçimli sıralamada en küçüğün değerini tutmak.** `int min = a[i];` biçiminde yalnız değer tutulursa tur sonunda en küçük öğenin yeri bilinmez ve takas yapılamaz. Takas için indis (`min = j`) gerekir.
2. **Eklemeli sıralamada `v = a[i]`'yi kaydırmadan önce saklamamak.** İlk kaydırma `a[i]`'nin üzerine yazar ve yerleştirilecek değer kaybolur.

3. **Sınırı `j >= 0` yazmak.** Alt dizi `a[L..R]` sıralanıyorsa doğru sınır `j >= L`'dir. `j >= 0`, *L > 0* olduğunda parçanın dışındaki öğeleri bozar.
4. `if (kucuk(A, B)) SWAP(A, B);` **kalıbında yönü karıştırmak.** Bu satır *A < B* ise takas eder. Küçük öğe sola gidecekse `A` sağdaki öğe olmalıdır: `kucuk(a[j], a[j - 1])`.
5. **Saymalı sıralamada aralığı unutmak.** Negatif ya da `k`'dan büyük bir değer `c[]`'nin dışına yazar; C bunu yakalamaz.
6. **Saymalı sıralamada L4'ü soldan sağa yazmak.** Dizi yine sıralanır, ama eşit öğelerin sırası tersine döner ve kararlılık kaybolur.
7. **Çok deyimli makroyu yalnız süslü parantezle yazmak.** `if (…) SWAP(…); else …` yapısında derleme hatası verir; gövde `do { … } while (0)` içine alınmalıdır.
8. **Ortalama durumu en iyi ile en kötünün ortalaması sanmak.** Erken duran kabarcık sıralamasında uçlar *n − 1* ve *n(n − 1)/2* karşılaştırmadır; ortalamaları ≈ $n^2/4$ eder. Gerçek ortalama ise ≈ $n^2/2$'dir: *n = 8* için uçların ortalaması 17,5, bütün dizilişlerin ortalaması 26,0'dır. Ortalama, bütün dizilişler üzerinden alınan beklenen değerdir.

19.12 Alıştırmalar ve çözümleri

19.12.1 Seçimli sıralamada turlar

Soru. *[5, 1, 4, 2, 8]* için seçimli sıralamada her tur sonundaki diziyi yazınız. Kaç takas yapıldı?

Çözüm. Tur 0'da en küçük 1'dir (`min = 1`); 5 ile takaslanır: *[1, 5, 4, 2, 8]*. Tur 1'de kalanların en küçüğü 2'dir (`min = 3`); 5 ile takaslanır: *[1, 2, 4, 5, 8]*. Tur 2'de en küçük 4 zaten `a[2]`'dedir; takas kendisiyledir. Tur 3'te 5 ile 8 karşılaştırılır; 5 yerindedir. Dizi dört turun sonunda *[1, 5, 4, 2, 8]* → *[1, 2, 4, 5, 8]* → *[1, 2, 4, 5, 8]* → *[1, 2, 4, 5, 8]* biçiminde ilerler. `SWAP` dört kez çağrılır, ama yalnız ilk ikisi gerçek takastır. Karşılaştırma sayısı her zamanki gibi 10'dur.

19.12.2 Eklemeli sıralamanın iki ucu

Soru. Eklemeli sıralama *[1, 2, 3, …, n]* ve *[n, …, 2, 1]* için kaç karşılaştırma yapar?

Çözüm. Sıralı girdide her tur tek karşılaştırmayla biter: *n − 1*. Ters sıralı girdide *i*. tur *i* karşılaştırma yapar: *1 + 2 + … + (n − 1) = n(n − 1)/2*. *n = 1.000* için 999 ve 499.500.

19.12.3 Erken duran kabarcık

Soru. `bubble2`, *[1, 2, 3, 5, 4]* girdisinde kaç tur çalışır?

Çözüm. 2 tur. İlk turda *j = 4*'te *4 < 5* doğrudur ve 5 ile 4 takaslanır; turun geri kalanında takas olmaz. Dizi sıralanmıştır, ama algoritma bunu ancak ikinci turda öğrenir: ikinci turda hiç takas olmaz ve yordam döner. Toplam *4 + 3 = 7* karşılaştırma ve 1 takas yapılır.

19.12.4 Sayaç dizisi

Soru. *a = [2, 5, 3, 0, 2, 3, 0, 3]* ve *k = 5* için saymalı sıralamanın L3 sonrası `c` dizisini yazınız.

Çözüm. L2 sonunda sayaçlar *c = [2, 0, 2, 3, 0, 1]*'dir: iki 0, hiç 1 yok, iki 2, üç 3, hiç 4 yok, bir 5. L3 bunları biriktirir: *c = [2, 2, 4, 7, 7, 8]*. Okuma örneği: `c[3] = 7`, 3'ten küçük ya da eşit yedi öğe vardır; son 3, `b[6]`'ya yazılır. Sıralı çıktı *[0, 0, 2, 2, 3, 3, 3, 5]*'tir.

19.12.5 Karar ağacının yüksekliği

Soru. 3 öğe için karar ağacının en az yüksekliği nedir? 4 öğe için?

Çözüm. *⌈lg 3!⌉ = ⌈lg 6⌉ = 3* ve *⌈lg 4!⌉ = ⌈lg 24⌉ = 5*. 3 öğe için Şekil 19.15'teki ağaç bu sınıra ulaşır. 4 öğe için 4 karşılaştırma yetmez: 4 karşılaştırma en çok $2^4 = 16$ farklı sonuca ayırabilir, oysa 24 diziliş vardır. 5 karşılaştırmayla 4 öğeyi sıralayan algoritmalar vardır.

19.12.6 Seçimli sıralamanın kararsızlığı

Soru. Seçimli sıralamanın kararsız olduğunu gösteren en küçük örneği bulunuz.

Çözüm. İki öğe yetmez: iki öğeli bir dizide eşit anahtarlar takaslanmaz, farklı anahtarlar için kararlılık sorusu yoktur. Üç öğeli *[2a, 2b, 1]* yeter. İlk turda en küçük 1'dir ve `a[0]`'daki 2a ile takaslanır: *[1, 2b, 2a]*. İkinci turda `kucuk(2a, 2b)` yanlıştır ve takas kendisiyledir. Sonuç *[1, 2b, 2a]*'dır: 2b, 2a'nın önüne geçmiştir. Uzak takas 2a'yı 2b'nin üzerinden atlatmıştır.

19.13 Özet

- Sıralama, kayıtları anahtarlarına göre düzenler. İç sıralamada veri bellektedir; dış sıralamada maliyeti disk erişimleri belirler.
- Kararlı bir algoritma eşit anahtarlı kayıtların göreli sırasını korur; birden çok alana göre sıralama ve basamak sıralaması buna dayanır. Yerinde bir algoritma $O(1)$ ek bellek kullanır.
- `kucuk` ve `SWAP` makroları algoritmaları tür ve ölçütten bağımsız yazmayı sağlar; çok deyimli makro `do { … } while (0)` içinde yazılır.
- Seçimli sıralama girdiye kördür: her durumda $n(n-1)/2$ karşılaştırma ve $n-1$ takas. Uzak takas yüzünden kararsızdır.
- Eklemeli sıralama sıralı girdide $n-1$, ters sıralı girdide $n(n-1)/2$ karşılaştırma yapar. Ortalaması bütün $n!$ diziliş üzerinden $(n^2+3n)/4 - H_n \approx n^2/4$'tür; en iyi ile en kötünün ortalaması değildir. Kararlı ve yerindedir. Bekçili sürüm iç döngüdeki sınır sınamasını kaldırır.
- Kabarcık sıralaması komşuları takaslar; kararlıdır. Erken duran sürüm sıralı girdide $O(n)$'dir, ama ortalamada yine $\approx n^2/2$ karşılaştırma yapar.
- Ters çift, yanlış sırada duran çifttir. Komşu takası ya da tek konumluk kaydırma tam olarak bir ters çift giderir; rastgele dizide ortalama $n(n-1)/4$ ters çift olduğu için bu tür her algoritma ortalamada $\Omega(n^2)$'dir.
- Saymalı sıralama karşılaştırma yapmaz; $O(n+k)$ zamanda ve $O(n+k)$ ek bellekle çalışır. Yerleştirme sağdan sola yapıldığı için kararlıdır. Aralık büyükse yararsızdır.
- Shell sıralaması azalan aralıklarla eklemeli sıralama yapar; son tur $h=1$ doğruluğu garanti eder. Uzak kaydırmalar ters çiftleri hızla giderir. Hız aralık dizisine bağlıdır; yerindedir, kararsızdır.
- Kova sıralaması $[0, 1)$'de düzgün dağılmış veride ortalama $O(n)$'dir; bütün öğeler tek kovaya düşerse $O(n^2)$'ye çıkar. C gerçeklemesi işaretçinin işaretçisiyle sıralı ekleme yapar ve düğümleri tek bir havuzdan alır.
- Karşılaştırmalı her sıralama bir karar ağacıdır. Ağacın en az $n!$ yaprağı olduğundan yüksekliği en az $\lceil \lg n! \rceil = \Theta(n \lg n)$'dir. Bu sınır ileri sıralama algoritmalarının ölçütüdür; karşılaştırma yapmayan algoritmalar ona takılmaz.

19.14 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

sıralama *sorting* · kayıt *record* · anahtar *key* · kararlı *stable (sort)* · yerinde sıralama *in-place sort* · makro *macro* · seçimli sıralama *selection sort* · eklemeli sıralama *insertion sort* · değişmez *invariant* · bekçi *sentinel* · permütasyon *permutation* · kabarcık sıralaması *bubble sort* · ters çift *inversion* · saymalı sıralama *counting sort* · basamak sıralaması *radix sort* · Shell sıralaması *Shell sort* · kova *bucket* · kova sıralaması *bucket sort* · karar ağacı *decision tree* · alt sınır *lower bound* · bellek sızıntısı *memory leak*

19.15 Kendini Dene

1. `selection` yordamı $[3, 1, 2]$ dizisinde kaç karşılaştırma ve kaç gerçek takas yapar? Her turun sonundaki dizi nedir?

   Cevap

   3 karşılaştırma ve 2 takas. Tur 0'da en küçük 1'dir: $[1, 3, 2]$. Tur 1'de $2 < 3$'tür ve takas yapılır: $[1, 2, 3]$. İki takas da gerçektir.

2. $[4, 1, 3, 2]$ dizisinin ters çift sayısı kaçtır? `insertion` bu diziyi kaç kaydırma ve kaç karşılaştırmayla sıralar?

   Cevap

   4 ters çift: (4, 1), (4, 3), (4, 2), (3, 2). Kaydırma sayısı ters çift sayısına eşittir: 4. Karşılaştırma 6'dır: $i = 1$'de 1 (1 başa gider), $i = 2$'de 2, $i = 3$'te 3.

3. `bubble2`, *[5, 1, 2, 3, 4]* ve *[2, 3, 4, 5, 1]* dizilerinde kaçar tur döner? İki dizinin ters çift sayısı aynı olduğu hâlde neden fark vardır?

> **Cevap**
>
> Birincide 4 tur, ikincide 2 tur. İç döngü sağdan sola ilerlediği için küçük bir öğe tek turda en sola gidebilir, ama büyük bir öğe her turda yalnız bir konum sağa ilerler. *[5, 1, 2, 3, 4]*'te 5'in sona varması dört tur ister.

4. `counting_sort`, *a = [1, 0, 1, 2]* ve *k = 2* ile çağrılıyor. L3 sonunda `c` nedir? `a[0]` 'daki 1 çıktıda hangi indise yazılır?

> **Cevap**
>
> *c = [1, 3, 4]*. L4 sağdan sola gezer: `a[3] = 2 → b[3]`, `a[2] = 1 → b[2]`, `a[1] = 0 → b[0]`, `a[0] = 1 → b[1]`. Girdide önce gelen 1 çıktıda da öndedir; algoritma kararlıdır.

5. Eklemeli sıralamada beşinci öğe (*j = 4*) yerleştirilirken beklenen karşılaştırma sayısı kaçtır?

> **Cevap**
>
> $E(C_4) = (4 + 2)/2 - 1/5 = 2{,}8$. Beş durumun karşılaştırma sayıları 1, 2, 3, 4 ve 4'tür; ortalamaları *14/5 = 2,8*.

6. 5 öğeyi sıralayan karşılaştırmalı bir algoritma en kötü durumda en az kaç karşılaştırma yapmak zorundadır? Seçimli sıralama kaç yapar?

> **Cevap**
>
> $\lceil \lg 5! \rceil = \lceil \lg 120 \rceil = \lceil 6{,}91 \rceil = 7$. Seçimli sıralama her girdide 10 karşılaştırma yapar; alt sınırın üç fazlası.

7. `bucket_sort`, *n = 5* ve girdi 0,94, 0,93, 0,92, 0,91, 0,90 ile çağrılıyor. Kovalara nasıl dağılır ve bu girdi neden kötüdür?

> **Cevap**
>
> *⌊5x⌋* beş sayının hepsi için 4'tür; hepsi 4 numaralı kovaya düşer. Sayılar azalan sırada geldiği için her yeni sayı listenin başına girer ve bu yüzden `while` döngüsü her seferinde yalnız bir karşılaştırma yapar. Girdi artan sırada gelseydi her ekleme bütün listeyi gezerdi. Tek kovaya düşen *n* öğe için en kötü durum $O(n^2)$'dir; düzgün dağılım varsayımı bozulmuştur.

8. `shell_sort` 10 öğelik bir dizide hangi aralıkları kullanır? Son aralık neden her zaman 1'dir ve bu neden önemlidir?

> **Cevap**
>
> 5, 2, 1. Aralık her adımda tam sayı bölmesiyle yarıya iner ve 1'e varmadan 0 olamaz: *h = 1* iken bir sonraki değer 0'dır ve döngü biter. *h = 1* turu tam bir eklemeli sıralamadır; önceki turlar ne yapmış olursa olsun diziyi sıralar. Algoritmanın doğruluğu bu son turdan gelir.

19.16 İleri okuma

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, Bölüm 7. Sıralama kavramları, eklemeli sıralamanın analizi ve karar ağacı bu kaynağın düzenini izler.
- R. Sedgewick, *Algorithms in C, Parts 1–4*, 3. baskı, Bölüm 6. `kucuk` ve `SWAP` makrolarıyla yazılmış temel sıralama algoritmaları, bekçili eklemeli sıralama ve Shell sıralamasının aralık dizileri ayrıntılı deneylerle tartışılır.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, Bölüm 2 ve 8. Eklemeli sıralamanın döngü değişmeziyle doğruluk kanıtı, saymalı ve kova sıralaması, kova sıralamasının ortalama durum analizi ve alt sınır teoremi.
- D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2. baskı, Kesim 5.2. Ters çiftlerin, Shell sıralamasının ve *h*-sıralı dizilerin derinlemesine incelemesi; bu konuların başvuru kaynağı.

19.17 Kaynaklar

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. bs., Silicon Press, 2008. Bölüm 7, Sorting.

2. R. Sedgewick, *Algorithms in C, Parts 1–4: Fundamentals, Data Structures, Sorting, Searching*, 3. bs., Addison-Wesley, 1998. Bölüm 6, Elementary Sorting Methods.

3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. bs., MIT Press, 2022. Bölüm 2 (Getting Started) ve Bölüm 8 (Sorting in Linear Time).

4. D. L. Shell, "A High-Speed Sorting Procedure", *Communications of the ACM*, 2(7), 30–32, 1959.

5. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2. bs., Addison-Wesley, 1998.

KISIM V · BÖLÜM 20

İleri Sıralama Algoritmaları

Temel sıralamalar öğe sayısı on binleri geçince n^2 ile orantılı süreleri yüzünden kullanılamaz olur. Bu not, aynı işi $n \lg n$ ile orantılı sürede yapan dört algoritmayı C koduyla ve adım adım inceler. Bunlar hızlı sıralama, birleştirme sıralaması, heap sort ve basamak sıralamasıdır. C kitaplığının `qsort` yordamı da karşılaştırma fonksiyonuyla birlikte ele alınır. Her algoritmanın zaman ve bellek maliyeti gerekçesiyle verilir; sayılar çalıştırılan programlarla doğrulanmıştır.

Kod (C): `github.com/sercankulcu/data-structures-and-algorithms-c/Bolum20`

ÖĞRENME ÇIKTILARI

Hızlı sıralama
Bölümleme değişmezini açıklamak; `bol()` ve `quicksort1()`'i bir dizi üzerinde adım adım izlemek.

Analiz
En iyi durumda $n \lg n$, en kötü durumda n^2 ve ortalamada $\approx 1{,}39\, n \lg n$ karşılaştırma sonuçlarını yineleme bağıntılarından elde etmek.

İyileştirmeler
Özyinelemesiz sürümün yığınını $O(\lg n)$'de tutmak; üçün ortancası ve küçük parça eşiğinin etkisini göstermek; `qsort`'u doğru bir karşılaştırma fonksiyonuyla kullanmak.

Birleştirme sıralaması
`merge` işlemini, çerçeve ağacını ve bağlı (indisli) sürümü izlemek; $O(n)$ ek belleği `malloc` ile ayırıp geri vermek.

Heap sort
Diziyi aşağıdan yukarıya $O(n)$'de heap'e çevirmek ve yerinde $O(n \lg n)$'de sıralamak.

Basamak sıralaması
En az anlamlı basamaktan başlayan basamak sıralamasını kovalı bağlı listelerle uygulamak ve karmaşıklığını yorumlamak.

20.1 Sıralamayı hızlandırmak: böl ve fethet

20.1.1 Temel sıralamaların sınırı

Seçimli sıralama her turda kalan öğelerin en küçüğünü bulur ve onu yerine koyar. Kabarcık sıralaması komşu öğeleri karşılaştırarak en büyüğü sona taşır. İkisi de her turda tek bir öğeyi kesin yerine yerleştirir. Ne var ki bir sonraki tur yine $n - 1$ öğelik bir problemle başlar. Problem her turda yalnız bir öğe küçülür ve toplam iş $n + (n - 1) + \dots + 1$, yani n^2 mertebesinde olur. Eklemeli sıralama neredeyse sıralı veride doğrusal sürer, ama genel girdide o da n^2'ye çıkar.

Bu sınırı aşmak için öğeleri yerleştirmek yetmez; problemi her adımda hızla küçültmek gerekir. Karşılaştırmaya dayanan her sıralama algoritması için bilinen bir alt sınır da vardır. n öğenin $n!$ dizilişinden doğrusunu bulmak için en kötü durumda en az $\lceil \lg n! \rceil \approx n \lg n - 1{,}44\, n$ karşılaştırma gerekir. Burada $\lg$, 2 tabanlı logaritmadır. Bu notun algoritmaları bu alt sınıra katsayısı küçük bir çarpanla yaklaşır.

20.1.2 Böl ve fethet

> **Tanım 20.1**
>
> **Böl ve fethet** (*divide and conquer*), bir problemi doğrudan çözülecek kadar küçülene dek özyinelemeli olarak daha küçük alt problemlere bölen, alt problemleri çözen ve çözümleri birleştiren tasarım yöntemidir. Üç adımı vardır: *böl*, *çöz* ve *birleştir*.

Hızlı sıralama ve birleştirme sıralaması bu yöntemin iki farklı yorumudur. Hızlı sıralama işin ağır kısmını bölerken yapar: diziyi öyle iki parçaya ayırır ki soldakilerin hepsi sağdakilerden küçük ya da onlara eşit olur. Bu durumda birleştirmeye gerek kalmaz; iki parça ayrı ayrı sıralanınca bütün dizi sıralanmış olur. Birleştirme sıralaması ise tam tersini yapar. Diziyi düşünmeden ortadan ikiye böler, işin ağır kısmını iki sıralı yarıyı birleştirirken yapar.

Her iki algoritmanın da maliyeti bir **yineleme bağıntısı** (*recurrence relation*) ile yazılır. n öğeli problemin maliyeti, daha küçük problemlerin maliyetleri ve bölme ya da birleştirme işinin maliyeti cinsinden ifade edilir. Örneğin her adımda iki yarıya bölen ve bölme işi n süren bir algoritmanın maliyeti $C_n = 2C_{n/2} + n$ bağıntısına uyar.

Notta sıralanan diziler hep `int` dizisidir. Karşılaştırma ve takas için temel sıralamalarda kullanılan iki makro aynen korunur:

```
#define kucuk(A, B) ((A) < (B))
#define SWAP(A, B) do { int t_ = (A); (A) = (B); (B) = t_; } while (0)
```

`kucuk(A, B)`, $A < B$ ise doğrudur. Karşılaştırmayı tek bir yerde toplamak, sayaç eklemeyi ya da öğe türünü değiştirmeyi kolaylaştırır. Bu notun ölçümlerinde karşılaştırmalar bu makroya bir sayaç eklenerek sayılmıştır. `SWAP` iki argümanlıdır ve geçici değişkeni kendi içinde tanımlar. Gövdenin `do { … } while (0)` içine alınması, makronun `if (…) SWAP(x, y); else …` gibi bir yerde tek bir deyim gibi davranmasını sağlar. Gövde yalnız süslü ayraçla yazılsaydı, sondaki noktalı virgül `else`'i boşa düşürür ve derleme hatası verirdi.

20.2 Hızlı sıralama

Hızlı sıralama (*quick sort*), C. A. R. (Tony) Hoare'un 1962'de yayımladığı algoritmadır. Türkçede "çabuk sıralama" adıyla da anılır. Ortalama durumda $n \lg n$ ile orantılı sürer; en kötü durumda n^2'ye çıkar. Yardımcı olarak yalnız küçük bir yığın kullanır. **Kararlı** (*stable*) değildir, yani eşit anahtarlı öğelerin girişteki göreli sırasını korumaz. Bu kusuruna karşın bilinen en hızlı genel amaçlı sıralama algoritmalarından biridir. Standart C kitaplığındaki `qsort` adı da bu algoritmadan gelir.

Aşağıdaki kod R. Sedgewick'in *Algorithms in C* kitabından alınmıştır. Sedgewick, Donald Knuth'un doktora öğrencisidir ve hızlı sıralama üzerine kapsamlı çalışmalar yapmıştır. Kod kısa ve dikkatle yazılmıştır; her satırın bir görevi vardır.

20.2.1 `quicksort1`

```
int bol(int a[], int L, int R);

void quicksort1(int a[], int L, int R) {
    if (R <= L) return;
    int i = bol(a, L, R);
    quicksort1(a, L, i - 1);
    quicksort1(a, i + 1, R);
}
```

Yordam `a[L..R]` aralığını sıralar. Bütün diziyi sıralamak için `quicksort1(a, 0, n - 1)` çağrılır. İlk satırdaki prototip, `bol`'un tanımı daha aşağıda olsa bile derleyicinin çağrıyı denetleyebilmesi içindir.

- `if (R <= L) return;` özyinelemenin taban durumudur. $R < L$ ise aralık boştur, $R = L$ ise tek öğe vardır. İki durumda da yapılacak iş yoktur.
- `int i = bol(a, L, R);` aralığı bölümler. `bol` bir öğeyi, **pivotu** (*pivot*), sıralı dizide duracağı konuma koyar ve bu konumu döndürür. Dönüşte `a[L..i-1]`'deki her öğe pivottan küçük ya da ona eşit, `a[i+1..R]`'deki her öğe pivottan büyük ya da ona eşittir.
- İki özyinelemeli çağrı pivotun solunu ve sağını ayrı ayrı sıralar. Pivot yerindedir; bir daha hiçbir çağrı ona dokunmaz.

Yordamın doğruluğu bu üç satırdan kolayca çıkar. Sol parça sıralandığında, her öğesi pivottan büyük olmadığı için hepsi pivotun solunda doğru sırada durur. Sağ parça için de aynı şey geçerlidir. Bütün iş `bol`'dadır.

20.2.2 Bölümleme: `bol()`

Tanım 20.2

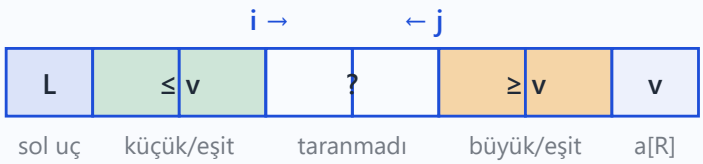
`a[L..R]` aralığının **bölümlenmesi** (*partition*), öğelerin yerlerini öyle değiştirmektir ki bir *i* konumu için *a[L..i−1] ≤ a[i] ≤ a[i+1..R]* olur. *a[i]*'deki öğeye pivot denir. Bölümleme sürerken her an doğru kalan önermeye **bölümleme değişmezi** (*invariant*) denir.

```
int bol(int a[], int L, int R) {
    int i = L - 1, j = R, v = a[R]; // pivot a[R]
    for (;;) {
        while (kucuk(a[++i], v))
            ;
        while (kucuk(v, a[--j]))
            if (j == L) break;
        if (i >= j) break;
        SWAP(a[i], a[j]);
    }
    SWAP(a[i], a[R]);
    return i;
}
```

`bol` pivot olarak en sağdaki öğeyi, `v = a[R]` 'yi seçer. İki indis kullanır: `i` soldan sağa, `j` sağdan sola tarar. Satır satır:

- **Satır 2.** `i = L - 1` ve `j = R` ile başlanır. Taramalar `++i` ve `--j` ile önce ilerleyip sonra baktığı için ilk bakılan öğeler `a[L]` ve `a[R-1]` 'dir. Pivotun kendisi taramaya girmez.
- **Satır 4–5.** `i` , *v*'den büyük ya da ona eşit ilk öğede durur. Bu döngüde sınır denetimi yoktur ve gerek de yoktur. En kötü durumda `i` , `R` 'ye ulaşır ve `a[R] = v` , `kucuk(v, v)` yanlış olduğu için taramayı durdurur. Pivot burada doğal bir **bekçi** (*sentinel*) görevi görür.
- **Satır 6–7.** `j` , *v*'den küçük ya da ona eşit ilk öğede durur. Bu taramanın solunda bekçi yoktur: pivot aralığın en küçük öğesiyse `j` dizinin başından taşabilirdi. `if (j == L) break;` satırı bunu önler.
- **Satır 8.** `i >= j` ise indisler buluşmuş ya da çaprazlanmıştır. Bütün öğeler taranmıştır ve döngü biter.
- **Satır 9.** Aksi hâlde `a[i]` sağ tarafa, `a[j]` sol tarafa aittir; ikisi de yanlış taraftadır. Takas edilince ikisi de doğru tarafa geçer ve taramalar kaldıkları yerden sürer.
- **Satır 11–12.** Döngüden çıkıldığında `a[i]` , sağ tarafın ilk öğesidir. Pivot onunla takas edilir ve `i` döndürülür.

Bölümleme değişmezi şekildeki dört bölgeyle anlatılır. Döngünün her adımında `i` 'nin solundaki öğeler (`a[L..i-1]`) *v*'den büyük değildir, `j` 'nin sağındakiler (`a[j+1..R-1]`) *v*'den küçük değildir. Arada henüz taranmamış bir bölge kalır. Her takas taranmamış bölgeyi iki öğe daraltır. İndisler çaprazlandığında bu bölge boşalır.



Şekil 20.1: Bölümleme değişmezi. Solda *v*'den küçük ya da ona eşit öğeler, sağda *v*'den büyük ya da ona eşit öğeler, ortada henüz taranmamış bölge, en sağda pivot *v = a[R]*.

Eşit anahtarlarda iki taramanın da durması bilinçli bir tercihtir. Bütün öğeleri eşit bir dizi düşünelim. Taramalar eşitte durmasaydı `i` hep sona kadar giderdi. Bölme hep *n − 1* ve 0 öğelik parçalar üretirdi ve süre n^2'ye çıkardı. Eşitte durulunca gereksiz görünen takaslar yapılır, ama indisler ortada buluşur.

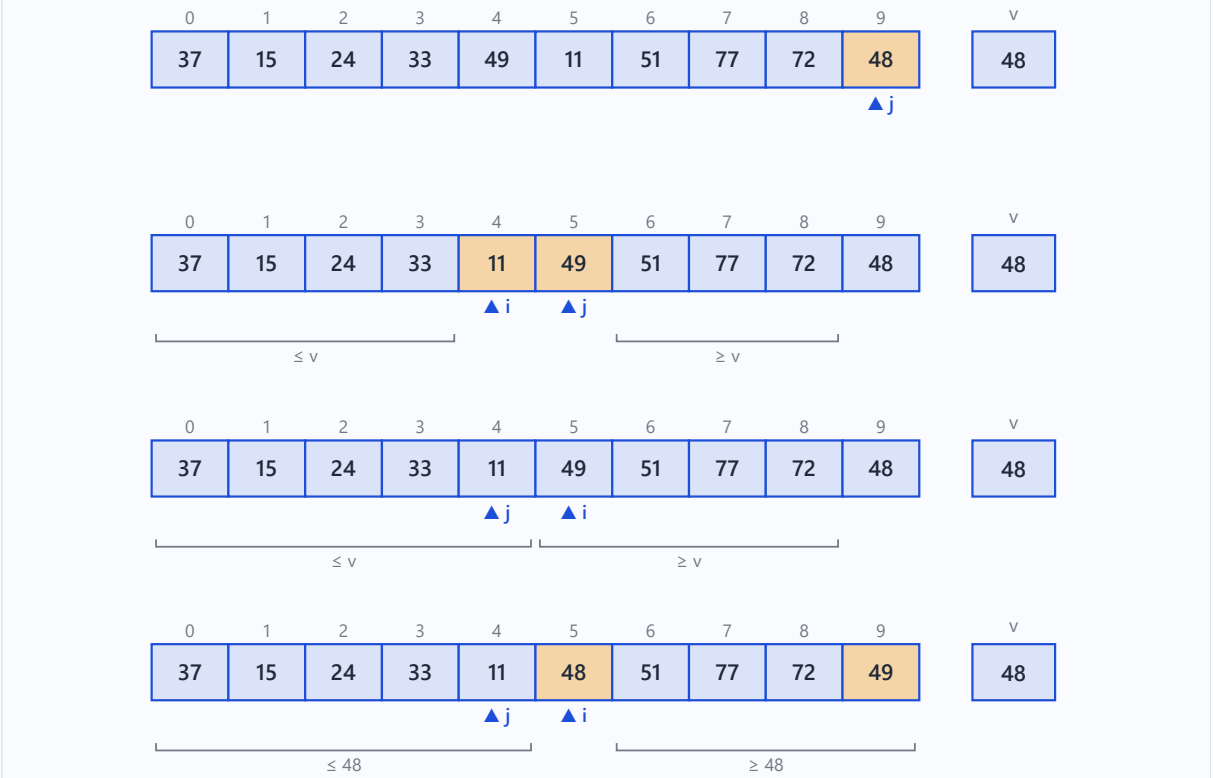
`bol` 'un maliyeti *n = R − L + 1* öğeli aralık için *Θ(n)*'dir. `i` yalnız sağa, `j` yalnız sola gider. İkisi buluşana kadar her öğeye bir kez bakılır; çaprazlanmada en çok birkaç öğeye ikinci kez bakılır. Karşılaştırma sayısı *n* ya da *n + 1*'dir. Ek bellek birkaç değişkendir: *O(1)*.

20.2.3 Kod izleme: `bol(a, 0, 9)`

Dizi *[37, 15, 24, 33, 49, 11, 51, 77, 72, 48]* olsun. Pivot *v = a[9] = 48*'dir. Tablo her adımda indislerin nerede durduğunu ve dizinin hâlini verir; şekil önemli dört anı gösterir.

Adım	Satır	`i`	`j`	Olay	Dizi
0	2	−1	9	başlangıç	`37 15 24 33 49 11 51 77 72 48`
1–4	4	0…3	9	37, 15, 24, 33 < 48: `i` ilerler	aynı
5	4	4	9	*a[4] = 49 ≥ 48*: `i` durur	aynı
6–8	6	4	8…6	72, 77, 51 > 48: `j` sola ilerler	aynı
9	6	4	5	*a[5] = 11 ≤ 48*: `j` durur	aynı
10	9	4	5	*i < j*: takas	`37 15 24 33 11 49 51 77 72 48`
11	4	5	5	*a[5] = 49 ≥ 48*: `i` durur	aynı
12	6	5	4	*a[4] = 11 ≤ 48*: `j` durur	aynı
13	8	5	4	*i ≥ j*: döngüden çık	aynı
14	11–12	5	4	pivot `a[5]`'e; 5 döner	`37 15 24 33 11 48 51 77 72 49`

Tablo 20.1: `bol(a, 0, 9)` izlemesi. Pivot *v = 48*.



Şekil 20.2: `bol(a, 0, 9)` izlemesinin dört anı, yukarıdan aşağıya: başlangıç (*i = −1, j = 9*); 49 ile 11'in takası; indislerin çaprazlanması (*i = 5, j = 4*); pivotun 5. konuma taşınması. Turuncu hücreler o adımda yer değiştiren öğelerdir; altlardaki ayraçlar *≤ v* ve *≥ v* bölgelerini gösterir.

Son durumda 48'in solunda 37, 15, 24, 33, 11 vardır ve hepsi 48'den küçüktür. Sağında 51, 77, 72, 49 vardır ve hepsi büyüktür. İki tarafın kendi içinde sıralı olmadığına dikkat edilmelidir. Bölümleme yalnız pivotun yerini kesinleştirir: veriler sıralansaydı 48 tam 5. konumda olurdu. Bu izlemede *n + 1 = 11* karşılaştırma yapılmıştır: `i` taraması 6, `j` taraması 5 kez bakmıştır. 4 ve 5. konumlara iki tarama da bakmıştır; fazladan karşılaştırma çaprazlanmadan gelir.

Sık yapılan hata

`j` döngüsündeki `if (j == L) break;` satırını "`i` döngüsünde de sınır denetimi yok, burada da gerekmez" diye silmek. İki tarama simetrik değildir. `i` 'yi sağda pivot durdurur, ama `j` 'yi solda durduracak bir öğe yoktur. Pivot aralığın en küçüğüyse `j` , `a[L-1]` 'e, hatta dizinin başından önceki belleğe taşar. Bu tanımsız davranıştır; program bazen doğru çalışıyor gibi görünür, bazen çöker.

20.2.4 Kod izleme: `quicksort1(a, 0, 9)`

Aynı dizi bu kez `quicksort1` ile baştan sona sıralansın. Tablo her `bol` çağrısını, döndürdüğü konumu ve sonraki dizi durumunu verir. Özyineleme önce sol parçayı tamamen bitirir, sonra sağ parçaya geçer. Bu yüzden çağrı yığını sol parçalarda derinleşir.

Çağrı	Pivot	Konum	Sonraki çağrılar	Dizi (sonra)
`bol(0, 9)`	48	5	(0, 4), (6, 9)	`37 15 24 33 11 48 51 77 72 49`
`bol(0, 4)`	11	0	(0, −1), (1, 4)	`11 15 24 33 37 48 51 77 72 49`
`bol(1, 4)`	37	4	(1, 3), (5, 4)	aynı
`bol(1, 3)`	33	3	(1, 2), (4, 3)	aynı
`bol(1, 2)`	24	2	(1, 1), (3, 2)	aynı
`bol(6, 9)`	49	6	(6, 5), (7, 9)	`11 15 24 33 37 48 49 77 72 51`
`bol(7, 9)`	51	7	(7, 6), (8, 9)	`11 15 24 33 37 48 49 51 72 77`
`bol(8, 9)`	77	9	(8, 8), (10, 9)	aynı

Tablo 20.2: `quicksort1(a, 0, 9)` sırasında yapılan `bol` çağrıları. Boş ya da tek öğeli aralıkla yapılan çağrılar hemen döner ve tabloda gösterilmez.

`bol(0, 4)` 'te pivot 11, aralığın en küçüğüdür. `i` hemen `a[0] = 37` 'de durur. `j` ise sola doğru 33, 24, 15'i geçer ve `j == L` denetimiyle 0'da durur. Çaprazlanma olur, pivot 0. konuma gelir ve 37 sona geçer. Sol parça boştur. Bu, `j` döngüsündeki sınır denetiminin iş başında görüldüğü andır.

Toplam 17 `quicksort1` çağrısı yapılır. Sekizi `bol` çağırır ve her biri bir pivotu kesin yerine koyar. Kalan dokuzu boş ya da tek öğeli aralıkla çağrılır ve hemen döner: *(0, −1)*, *(1, 1)*, *(3, 2)*, *(4, 3)*, *(5, 4)*, *(6, 5)*, *(7, 6)*, *(8, 8)*, *(10, 9)*. Her `bol` çağrısı iki alt çağrı ürettiği için yaprak sayısı iç düğüm sayısından bir fazladır: *8 + 9 = 17*. En derin anda yığında `main` 'in üstünde altı `quicksort1` çerçevesi vardır: (0, 9), (0, 4), (1, 4), (1, 3), (1, 2) ve (1, 1).



Şekil 20.3: `quicksort1(a, 0, 9)` 'un üç anı, yukarıdan aşağıya: ilk bölümlemeden sonra 48 yerinde; sol parça bitip `bol(6, 9)` çalıştıktan sonra; son boş çağrı (10, 9). Yeşil hücreler kesin yerine oturmuş öğelerdir.

20.2.5 Neden hızlı?

Kabarcık ya da seçimli sıralamada da her adımda bir öğe için doğru konum bulunur. Fark şuradadır: onlar bir öğeyi yerine koyar ama problemi küçültmeye katkı yapamaz. Sonraki adımın alt problemi $n - 1$ öğeliktir. Hızlı sıralama pivotu yerine koyarken problemi de iki küçük parçaya ayırır. İyi bir bölmede iki parçanın her biri yaklaşık $n/2$ öğelidir. Bir sonraki düzeyde dört parça $n/4$ öğelidir ve bu böyle sürer. $\lg n$ düzeyden sonra parçalar tek öğeye iner. Her düzeyde bölümleme işinin toplamı n civarındadır; bu yüzden toplam iş $n \lg n$ mertebesindedir.

Bu nottaki hızlı sıralama sürümlerinin hepsi aynı `bol`'u kullanır. Bu kadar kısa kodlar bir kez okumakla kavranmaz; farklı dizilerle birkaç kez izlenmeli ve mantığından yeniden yazılmalıdır.

20.2.6 Algoritma ve bilgisayar

Öğrencilerin sık sorduğu bir soru şudur: Hoare bu algoritmayı 1960'larda nerede denedi, nasıl test etti? “Algoritma” kelimesi nereden geliyor; 9. yüzyılda el-Harezmî'nin dizüstü bilgisayarı mı vardı?

Cevap algoritma kavramının kendisindedir. Algoritma, konuyu hiç bilmeyen birinin adım adım işleteceği ve doğru sonuca ulaşacağı kesin bir yöntemdir; tanımında bilgisayar geçmez. 1960'larda bilgisayarlar zaten vardı, ama bir algoritmanın doğruluğu ve maliyeti kâğıt üzerinde kanıtlanır; kanıtın bir makalede yayımlanması yeterlidir. Bu ders de bu yüzden bilgisayardan bağımsızdır.

Kelimenin kökü de bunu doğrular. El-Harezmî (780–850), Bağdat'taki Beytü'l-Hikme'de çalışmış bir matematikçi ve gökbilimcidir. Birinci ve ikinci dereceden denklemlerin sistematik çözümünü veren *el-Kitâbü'l-Muhtasar fî Hisâbi'l-Cebr ve'l-Mukâbele* kitabını yazmıştır. “Cebir” ve Batı dillerindeki “algebra” bu kitaptaki *el-cebr* yönteminin adından gelir. “Algoritma” ise el-Harezmî'nin adının Latince biçimi *Algoritmi*'den türemiştir. El-Harezmî onluk konumsal sayı sistemini Batı'ya tanıtan eserleriyle de bilinir.

20.3 Hızlı sıralamanın analizi

Hızlı sıralamanın süresini belirleyen tek şey, `bol`'un aralığı nasıl böldüğüdür. Her `bol` çağrısı kendi aralığının boyu kadar iş yapar. Bölmeler dengeliyse özyineleme ağacı sığ kalır, dengesizse ağaç bir zincire döner. Bu kesimde maliyet karşılaştırma sayısıyla ölçülür, çünkü sıralamanın asıl işi karşılaştırmadır ve takas sayısı onunla orantılıdır. C_n, n öğeli bir diziyi sıralarken yapılan karşılaştırma sayısını gösterir.

20.3.1 En kötü durum: sıralı dizi

Dizi zaten sıralı olsun. Pivot `a[R]` her seferinde aralığın en büyük öğesidir. `i` taraması bütün aralığı geçip `R`'de durur; `j` ilk baktığı öğede durur. Pivot yerinde kalır ve sol parça $n - 1$ öğeli, sağ parça boştur. Sonraki çağrı yine en büyük öğeyi pivot alır ve yine bir eksik boyla devam eder. Toplam iş

$$n + (n-1) + \dots + 2 + 1 = n(n+1)/2 = O(n^2)$$

olur. Kesin sayı da hesaplanabilir. Boyu s olan aralıkta `bol` tam $s + 1$ karşılaştırma yapar: `i` taraması s, `j` taraması 1. Boyları $n, n-1, \dots, 2$ olan $n - 1$ aralık bölümlenir. $n = 10$ için toplam $11 + 10 + \dots + 3 = 63$ karşılaştırmadır.

Ayrıca özyineleme derinliği $n - 1$'e çıkar; büyük sıralı diziler, yığın boyu küçük sistemlerde **yığın taşmasına** (*stack overflow*) yol açabilir. 100.000 öğelik sıralı dizi `-O2` ile 0,95 saniye, aynı boyda rastgele dizi 0,008 saniye sürmüştür (süreler makineye göre değişir). Ters sıralı dizi de aynı ölçüde kötüdür.

20.3.2 En iyi durum: tam ortadan bölme

En iyi durumda pivot her seferinde aralığın tam ortancasıdır. `i` ve `j` toplam n adım ilerler ve geriye $n/2$'şer öğeli iki alt problem kalır:

$$C_n = 2C_{n/2} + n,\quad C_1 = 0$$

Bağıntı kendi içine yerleştirilerek açılır. $C_{n/2} = 2C_{n/4} + n/2$ olduğundan $C_n = 2(2C_{n/4} + n/2) + n = 4C_{n/4} + 2n$ bulunur. Bir adım daha açılınca $8C_{n/8} + 3n$ elde edilir. k adımdan sonra

$$C_n = 2^k\, C_{n/2^k} + k \cdot n$$

olur. $n = 2^k$, yani $k = \lg n$ alınırsa $C_{n/2^k} = C_1 = 0$ olur ve $C_n = n \lg n$ bulunur.

Karşılaştırmalı sıralamanın alt sınırı *⌈lg n!⌉ ≈ n lg n − 1,44 n* idi. Baş terimin katsayısı yine 1'dir; aradaki fark yalnız *1,44 n* kadardır. Asıl soru rastgele bir dizideki davranıştır: “Bunun neresi hızlı?” sorusunun cevabı ortalama durumdadır.

20.3.3 Sayım: sıralı ve rastgele girdi

İki uç durum arasındaki farkı somutlaştırmak için `bol` 'daki karşılaştırmalar sayılmıştır. Tablo sıralı ve rastgele girdide karşılaştırma sayısını ve özyineleme derinliğini verir. Derinlik, iç içe bekleyen ve `bol` çağıran `quicksort1` çerçevelerinin en büyük sayısıdır.

n	Sıralı: karşılaştırma	Derinlik	Rastgele: karşılaştırma	Derinlik	*n lg n*	Rastgele ortalama
10	63	9	36	5	33	35,4
100	5.148	99	721	9	664	765,6
1000	501.498	999	12.831	25	9.966	12.198,3

Tablo 20.3: `quicksort1` 'de karşılaştırma sayısı ve özyineleme derinliği. Rastgele sütunlar sunumdaki tek bir denemedir (Python simülasyonu, tohum 20); son sütun çok sayıda rastgele dizinin ortalamasıdır.

Sıralı girdide karşılaştırma sayısı *(n + 1)(n + 2)/2 − 3*, yani yaklaşık *n²/2*'dir; derinlik *n − 1*'dir. *n = 1000*'de yarım milyon karşılaştırma yapılır. Rastgele girdide sayı *n lg n* mertebesinde kalır ve derinlik yalnız birkaç *lg n* kadardır. Aynı boydaki iki girdide iş kırk kat farklıdır. Sıralı ya da neredeyse sıralı veri gerçek hayatta seyrek değildir.

20.3.4 Ortalama durum: bağıntının kurulması

Ortalama durum analizi sezgiye değil matematiğe dayanır. Bir para bir milyon kez atılınca tura oranı ½'ye yaklaşır; sonuçların nerede yoğunlaştığını ortalama durum analizi böyle saptar. “Bana göre olasılıklar eşittir” demek yetmez; eşitlik kanıtlanmalıdır.

Girdinin *n!* dizilişinin hepsi eşit olasılıklı olsun ve anahtarlar birbirinden farklı olsun. Pivot `a[R]` 'dir. Rastgele bir dizilişte en sağdaki öğenin sıralı dizideki sırası *1, 2, …, n* değerlerinden her birini eşit olasılıkla, yani *1/n* olasılıkla alır. Pivotun sırası *k* ise `bol` onu *k*. konuma koyar; geriye *k − 1* ve *n − k* öğeli iki alt dizi kalır. Bir şey daha gerekir: alt diziler de rastgele olmalıdır. `bol` öğeleri yalnız pivotla karşılaştırır, birbirleriyle karşılaştırmaz. Bu yüzden her parçanın kendi içindeki dizilişleri yine eşit olasılıklıdır. Parçalar da aynı analize uyar.

`bol` pivotla *n + 1* karşılaştırma yapar. Bu sayıyı model olarak alırsak ortalama karşılaştırma sayısı şu bağıntıya uyar:

$$C_n = (n + 1) + (1/n)\,\Sigma_{k=1}^{n}\,(C_{k-1} + C_{n-k}),\; C_0 = C_1 = 0$$

İlk terim bölümleme işidir. Toplam, *k*'nın her değeri için iki alt problemin maliyetidir; *1/n* çarpanı bu değerlerin eşit olasılıklı olmasından gelir. İki toplam aynı terimleri içerir. *k = 1, …, n* için C_{k-1} terimleri $C_0, C_1, \ldots, C_{n-1}$'dir. C_{n-k} terimleri de aynı değerlerdir, yalnız ters sırada gelir. Öyleyse

$$\Sigma_{k=1}^{n}\,(C_{k-1} + C_{n-k}) = 2\,\Sigma_{k=1}^{n}\,C_{k-1}$$

yazılabilir. Bağıntının iki yanı *n* ile çarpılınca kesirden kurtulunur:

$$n \cdot C_n = n(n + 1) + 2\,\Sigma_{k=1}^{n}\,C_{k-1} \quad (1)$$

Gerçek `bol` , karşılaştırmalı bir taramada *n* ya da *n + 1* karşılaştırma yapar; indisler aynı konumda buluşursa bir eksik sayar. 20.000 rastgele dizide yapılan denemede çağrıların yaklaşık %90'ı *n + 1*, kalanı *n* karşılaştırma yapmıştır. Bu yüzden ölçülen ortalamalar C_n'den biraz küçük çıkar (tablonun son sütunu). Bağıntı küçük bir üst modeldir; asimptotik sonucu değişmez.

20.3.5 Çıkarma hilesi ve çözüm

(1) bağıntısındaki toplam her *n* için bütün önceki değerleri içerir. Bu tür bağıntılar genellikle yerine koyma yöntemiyle çözülür. Horowitz'in kitabı da böyle yapar. Daha kısa bir yol vardır: toplamı ortadan kaldırmak. (1)'de *n* yerine *n − 1* yazılır:

$$(n - 1) \cdot C_{n-1} = (n - 1)n + 2\,\Sigma_{k=1}^{n-1}\,C_{k-1} \quad (2)$$

(1)'den (2) çıkarılınca iki toplamın ortak terimleri gider; yalnız $2C_{n-1}$ kalır. Sağdaki ilk terimlerin farkı *n(n + 1) − n(n − 1) = 2n*'dir:

$$n \cdot C_n - (n-1) \cdot C_{n-1} = 2n + 2C_{n-1} \Rightarrow n \cdot C_n = (n+1) \cdot C_{n-1} + 2n$$

Artık bağıntı yalnız bir önceki terime bağlıdır. İki yan $n(n+1)$'e bölünür:

$$C_n/(n+1) = C_{n-1}/n + 2/(n+1)$$

Bu biçimin güzelliği, sol yanla sağdaki ilk terimin aynı kalıpta olmasıdır. $D_n = C_n/(n+1)$ denirse $D_n = D_{n-1} + 2/(n+1)$ olur. Bu, iterasyonla hemen açılır: $D_n = D_{n-1} + 2/(n+1) = D_{n-2} + 2/n + 2/(n+1) = \ldots$. Çıkarma adımı (2)'nin $n - 1 \geq 2$ için geçerli olmasını ister. Bu yüzden açılım $C_2 = 3$'te durdurulur; $D_2 = 1$'dir:

$$C_n/(n+1) = 1 + 2(1/4 + 1/5 + \ldots + 1/(n+1)) \ (n \geq 2)$$

Parantezdeki toplam, harmonik sayı $H_{n+1} = 1 + 1/2 + \ldots + 1/(n+1)$'den ilk üç terimin çıkarılmış hâlidir. Baskın kısım $2(n+1)(1/4 + \ldots + 1/(n+1))$'dir ve bu da $2(n+1)\ln n$'den küçüktür. $H_n \approx \ln n + 0{,}577$ olduğundan

$$C_n \approx 2n \ln n = 2 \ln 2 \cdot n \lg n \approx 1{,}39\, n \lg n$$

bulunur. Son adımda $\ln n = \ln 2 \cdot \lg n$ eşitliği kullanılmıştır; $2 \ln 2 \approx 1{,}386$'dır.

Teorem 20.1 (Hızlı sıralamanın ortalama durumu)

Farklı anahtarlı n öğenin her dizilişi eşit olasılıklıysa ve `bol` $n+1$ karşılaştırma yapıyorsa, `quicksort1` 'in ortalama karşılaştırma sayısı $C_n = 2(n+1)H_{n+1} - 8(n+1)/3$'tür ($n \geq 2$). Buna göre $C_n \approx 2n \ln n \approx 1{,}39\, n \lg n$ ve $C_n = \Theta(n \lg n)$ olur.

İspat. Kapalı biçim, yukarıdaki açılımdan çıkar: $C_n = (n+1)(1 + 2(H_{n+1} - 1 - 1/2 - 1/3)) = 2(n+1)H_{n+1} - (n+1)(2 \cdot 11/6 - 1)$ ve $2 \cdot 11/6 - 1 = 8/3$'tür. $n = 2$ için $3 \cdot (2 \cdot 11/6) - 8 = 3$ bulunur ve bağıntıdaki $C_2 = 3 + (2/2)(0+0) = 3$ ile örtüşür. $H_{n+1} = \ln n + O(1)$ olduğundan baş terim $2n \ln n$'dir.

Bağıntı Python ile tam olarak hesaplanıp formülle karşılaştırılmıştır. Tablo sonucu üç boy için verir. Oranın 1'e çok yavaş yaklaştığı görülür. Bunun nedeni, kapalı biçimdeki $-cn$ türünden alt dereceli terimlerdir. Bu terimler $n \lg n$'nin yanında yavaş yavaş önemini yitirir.

n	C_n (bağıntıdan)	$1{,}39\, n \lg n$	Oran
10	37,1	46,2	0,80
100	780,5	923,5	0,85
1000	12.318,6	13.852,4	0,89

Tablo 20.4: Ortalama karşılaştırma sayısı. C_n bağıntıdan tam olarak hesaplanmıştır.

Rastgele bir girdide hızlı sıralama en iyi durumundan yalnız %39 kadar fazla karşılaştırma yapar; n^2 durumunun olasılığı çok küçüktür. İç döngüsü de çok kısadır: bir artırma, bir karşılaştırma ve bir dallanma; öğeler bellekte sırayla okunur.

20.4 Hızlı sıralamayı iyileştirmek

`quicksort1` 'in iki zayıf yanı vardır. Birincisi, özyineleme derinliği en kötü durumda n'ye çıkar ve yığını tüketebilir. İkincisi, sıralı ya da ters sıralı girdi onu n^2'ye sürükler. Ayrıca küçük parçalarda özyinelemenin maliyeti, yapılan asıl işten fazladır. Bu kesim üç iyileştirmeyi sırayla inceler: açık yığınla özyinelemesiz sürüm, pivotun üç öğenin ortancasından seçilmesi ve küçük parçaların eklemeli sıralamaya bırakılması.

20.4.1 Özyinelemesiz sürüm ve açık yığın

Özyinelemeli bir yordam bekleyen işlerini çağrı yığınında tutar; her çağrı parametreleri, yerel değişkenleri ve dönüş adresini taşıyan bir **çerçeve** (*stack frame*) açar. `quicksort1` 'in bekleyen her işi bir (L, R) ikilisidir. Bu ikililer diziyle gerçeklenen bir yığına konursa özyineleme gerekmez; çağrı maliyeti kalkar ve hangi işin önce yapılacağı denetim altına girer.

```
#define YMAX 64     /* > 2·lg n */
static int yigit[YMAX], ust;
void yigit_baslat(void) { ust = 0; }
int yigit_bosmu(void) { return ust == 0; }
void yigit_ekle(int x) { yigit[ust++] = x; }
int yigit_al(void) { return yigit[--ust]; }
#define yigita_at(A, B) \
  do { yigit_ekle(B); yigit_ekle(A); } while (0)
```

Yığın, boyu `YMAX` olan `static` bir `int` dizisidir. `ust`, dizide ilk boş hücrenin indisidir; `ust == 0` yığının boş olduğunu gösterir. `static` anahtar sözcüğü, dizinin ve `ust`'ün yalnız bu kaynak dosyasından görülmesini sağlar ve ikisini program boyunca yaşatır. `yigit_ekle` değeri `yigit[ust]`'e yazar ve `ust`'ü artırır. `yigit_al` önce `ust`'ü azaltır, sonra o hücreyi döndürür. İki işlem de *O(1)*'dir.

`yigita_at(A, B)` makrosu bir ikiliyi iter. Önce `B`'yi, sonra `A`'yı iter. Yığından ilk alınan son itilen olduğu için ilk alınan `A` olur. `quicksort2`'de `yigita_at(L, R)` yazıldığında alma sırası `L`, ardından `R`'dir. Makronun gövdesi yine `do { … } while (0)` içindedir; böylece `if – else` dallarında tek deyim gibi kullanılabilir. Satır sonundaki ters bölü (`\`), makro tanımının alt satırda sürdüğünü önişlemciye bildirir.

Yığın taşma denetimi yapmaz; boyu hiçbir girdinin aşamayacağı biçimde seçilmiştir. Teorem 20.2'ye göre yığında en çok *⌊lg n⌋ + 1* ikili, yani *2(⌊lg n⌋ + 1)* `int` bulunur. $n < 2^{31}$ için bu 62'yi aşmaz.

```
void quicksort2(int a[], int L, int R) {
  yigit_baslat(); yigita_at(L, R);
  while (!yigit_bosmu()) {
    L = yigit_al(); R = yigit_al();
    if (R <= L) continue;
    int i = bol(a, L, R);
    if (i - L > R - i) {       // büyük parça önce
      yigita_at(L, i - 1); yigita_at(i + 1, R);
    } else {
      yigita_at(i + 1, R); yigita_at(L, i - 1);
    }
  }
}
```

- **Satır 2.** Yığın boşaltılır ve ilk problem, bütün aralık itilir.
- **Satır 3–5.** Yığın boşalana kadar bir ikili alınır. Aralık boş ya da tek öğeliyse `continue` ile sıradaki ikiliye geçilir. Bu satırlar `quicksort1`'in taban durumunun karşılığıdır.
- **Satır 6.** Aralık bölümlenir. Sol parça `a[L..i-1]`, sağ parça `a[i+1..R]`'dir; boyları *i − L* ve *R − i*'dir.
- **Satır 7–11.** Büyük parça önce, küçük parça sonra itilir. Yığın son gireni ilk çıkardığı için bir sonraki turda hep küçük parça işlenir. Boylar eşitse `else` dalı çalışır ve sol parça üstte kalır.

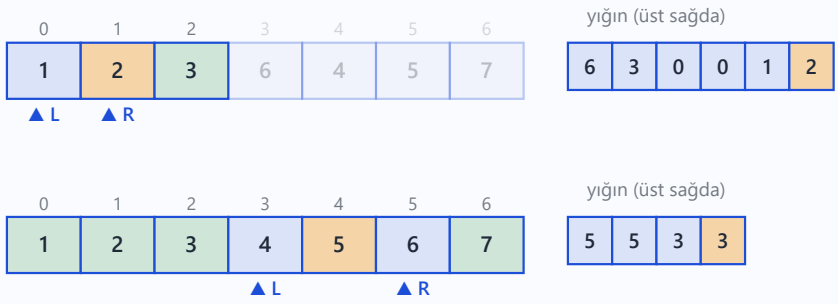
20.4.2 Kod izleme: `quicksort2(a, 0, 6)`

Dizi *[4, 7, 2, 6, 1, 5, 3]* olsun. Tablo, yığının her turdaki içeriğini alttan üste doğru verir. Yığında ikililer *R, L* sırasıyla durur; üstteki iki sayı sıradaki işin *L* ve *R*'sidir.

Tur	Alınan	Olay	Dizi (sonra)	Yığın (sonra)
0	—	ilk problem itilir	`4 7 2 6 1 5 3`	`[6, 0]`
1	(0, 6)	pivot 3 → 2; sağ (3, 6) büyük, önce o; sonra sol (0, 1)	`1 2 3 6 4 5 7`	`[6, 3, 1, 0]`
2	(0, 1)	pivot 2 → 1; sol (0, 0) büyük, önce o; sonra sağ (2, 1)	aynı	`[6, 3, 0, 0, 1, 2]`
3	(2, 1)	boş aralık: `continue`	aynı	`[6, 3, 0, 0]`
4	(0, 0)	tek öğe: `continue`	aynı	`[6, 3]`
5	(3, 6)	pivot 7 → 6; sol (3, 5) büyük, önce o; sonra sağ (7, 6)	aynı	`[5, 3, 6, 7]`
6	(7, 6)	boş aralık: `continue`	aynı	`[5, 3]`
7	(3, 5)	pivot 5 → 4; parçalar eşit, sağ (5, 5) önce, sonra sol (3, 3)	`1 2 3 4 5 6 7`	`[5, 5, 3, 3]`

Tur	Alınan	Olay	Dizi (sonra)	Yığın (sonra)
8	(3, 3)	tek öğe: `continue`	aynı	`[5, 5]`
9	(5, 5)	tek öğe: `continue`	aynı	`[]`
10	—	yığın boş: döngü biter	sıralı	`[]`

Tablo 20.5: `quicksort2(a, 0, 6)` izlemesi. Yığın alttan üste yazılmıştır.



Şekil 20.4: `quicksort2` izlemesinin iki anı. Üstte 2. turun sonu: yığında en çok üç ikili bekler. Altta 7. turun sonu: dizi sıralanmıştır, yığında yalnız iki tek öğeli iş kalmıştır. Yığının üstü sağdadır.

Yığında en çok 3 ikili, yani 6 `int` beklemiştir. Tek öğeli ve boş aralıklar da itilip alındıktan sonra atılır; bunları önceden denetlemek birkaç işlem kurtarır ama kodu uzatır.

20.4.3 Yığın neden küçük kalır?

Teorem 20.2 (Açık yığının boyu)

`quicksort2` n öğeli bir diziyi sıralarken yığında aynı anda en çok $\lfloor \lg n \rfloor + 1$ ikili bulunur. Ek bellek $O(\lg n)$'dir.

İspat. Bir aralık bölümlendiğinde büyük ve küçük iki parça itilir. Küçük parça en çok $\lfloor (s-1)/2 \rfloor$ öğelidir, çünkü iki parçanın toplamı $s-1$'dir. Küçük parça hemen alınır. Bu andan büyük parça alınana kadar yığına itilen her ikili, küçük parçanın içinden gelir. Öyleyse herhangi bir anda yığında bekleyen büyük parçalar, iç içe geçmiş aralıkların bölümlenmesinden kalmıştır: $S_1 \supset S_2 \supset \dots \supset S_k$. Her S_{i+1}, S_i'nin küçük parçasının içindedir ve $|S_{i+1}| < |S_i|/2$ olur. $|S_1| \le n$ olduğundan $|S_k| < n/2^{k-1}$'dir. Bölümlenen bir aralıkta en az iki öğe bulunur: $2 \le |S_k| < n/2^{k-1}$, yani $2^k < n$ ve $k \le \lfloor \lg n \rfloor$'dir. Yığında bu k büyük parçaya ek olarak en çok bir küçük parça bekler.

Aynı sonuç bellek bağıntısından da çıkar: küçük parça en çok yarı boydadır ve her düzey yığına iki `int` ekler; $T_n = T_{n/2} + 2$, ikili aramadaki gibi $T_n = 2 \lg n$ verir. Tablo sınırı ölçümle karşılaştırır. Sıralı girdide yığın neredeyse boş kalır, çünkü küçük parça boştur.

n	$\lfloor \lg n \rfloor + 1$	Rastgele girdi	Sıralı girdi
7	3	3	2
100	7	6	2
1000	10	9	2
10.000	14	10	2
1.048.576	21	15	—

Tablo 20.6: `quicksort2`'de yığında aynı anda bekleyen en çok ikili sayısı. Rastgele sütunu $n \le 1000$ için 2000, daha büyük n için 5 dizinin en büyüğüdür.

Sıralı girdide yığın küçük kalsa da süre yine n^2'dir; `quicksort2` 100.000 sıralı öğeyi 0,94 saniyede sıralamıştır. Özyinelemesiz sürüm bellek sorununu çözer, süre sorununu çözmez. Süre sorunu pivot seçimiyle çözülür.

> **Sık yapılan hata**
>
> Parçaları boyuna bakmadan, örneğin hep önce sol sonra sağ parçayı itmek. Kod yine doğru sıralar, ama yığının *O(lg n)* sınırı kaybolur. Kötü bir girdide yığında *n* mertebesinde ikili birikir; `YMAX` 64 olduğu için dizi taşar ve program kendi değişkenlerinin üzerine yazar.

20.4.4 Üçün ortancası ve küçük parça eşiği

Hızlı sıralama en kötü performansını sıralı veride gösterir. Eklemeli sıralama ise en iyi performansını tam da sıralı veride gösterir. Geliştirilmiş sürüm bu iki gözlemi birleştirir:

```
#define M 10    /* bu boydan küçük parça kalsın */
void quicksort3(int a[], int L, int R) {
    if (R - L <= M) return;
    SWAP(a[(L + R) / 2], a[R - 1]);
    if (a[R-1] < a[L]) SWAP(a[R-1], a[L]); // a[L] <= a[R-1]
    if (a[R] < a[L]) SWAP(a[R], a[L]);     // a[L] <= a[R]
    if (a[R] < a[R-1]) SWAP(a[R], a[R-1]); // a[R-1] <= a[R]
    int i = bol(a, L + 1, R - 1);
    quicksort3(a, L, i - 1);
    quicksort3(a, i + 1, R);
}
void sirala(int a[], int L, int R) {
    quicksort3(a, L, R);
    insertion(a, L, R);   /* kalan küçük parçalar */
}
```

Üçün ortancası (*median-of-three*), kısaca medyan-3, pivotu tek bir öğe yerine üç öğenin ortancası olarak seçer. Bu üç öğe aralığın ilk öğesi, ortadaki öğe ve son öğedir. Satır satır:

- **Satır 3.** Aralık en çok $M + 1$ öğeliyse ($R - L \le M$) yordam hiçbir şey yapmadan döner. Bu parçalar sıralanmadan bırakılır.
- **Satır 4.** Ortadaki öğe `a[R-1]` 'e taşınır. Artık üç aday `a[L]`, `a[R-1]` ve `a[R]` 'dedir.
- **Satır 5–7.** Üç "karşılaştır, gerekirse takas et" adımı bu üç öğeyi sıraya koyar. Sonunda $a[L] \le a[R-1] \le a[R]$ olur ve ortanca `a[R-1]` 'dedir.
- **Satır 8.** Bölümleme yalnız `a[L+1..R-1]` üzerinde yapılır. `bol` bu aralığın son öğesini, yani ortancayı pivot alır.
- **Satır 9–10.** Pivotun iki yanı özyinelemeli olarak işlenir.

Üç takas satırında argümanların sırası önemlidir. `if (kucuk(A, B)) SWAP(A, B)` kalıbı "$A < B$ ise takas et" demektir. Takastan sonra `B` 'deki öğe ikisinin küçüğüdür, `A` 'daki büyüğüdür. Koddaki ilk satır `if (a[R-1] < a[L]) SWAP(a[R-1], a[L]);` biçimindedir ve küçüğü `a[L]` 'ye koyar. Argümanlar `(a[L], a[R-1])` sırasıyla verilseydi büyük öğe `a[L]` 'ye giderdi ve bölümleme bozulurdu. Bu yanlış sürüm 1000 rastgele diziyle denenmiş, dizilerin hepsinde en az bir bölümlemede pivotun solunda ondan büyük bir öğe kalmıştır.

Medyan-3'ün iki kazancı vardır. Sıralı ve ters sıralı girdide pivot aralığın ortasından gelir ve n^2 durumu kalkar. 100.000 öğelik sıralı dizi `quicksort1` ile 0,95 saniye, `sirala` ile 0,001 saniye sürmüştür. Ayrıca uç öğeler **doğal bekçiler** olur: $a[L] \le pivot \le a[R]$ olduğu için `j` taraması en geç `a[L]` 'de durur.

İkinci iyileştirme küçük parça eşiği M'dir. Birkaç öğelik bir aralıkta çağrının ve hazırlığın sabit maliyeti asıl işten büyüktür; `quicksort3` $R - L \le M$ olan parçalara dokunmaz. Dizi "neredeyse sıralı" kalır: her öğe kendi küçük parçasının içinde yanlış yerdedir, parçalar birbirine göre doğru sıradadır. `sirala` sonunda tek bir eklemeli sıralama geçişi uygular:

```
void insertion(int a[], int L, int R) {
    for (int i = L + 1; i <= R; i++) {
        int v = a[i];      // yerleştirilecek
        int j = i - 1;     // a[L..i-1] sıralı
        while (j >= L && a[j] > v) {
            a[j + 1] = a[j]; // sağa kaydır
            j--;
        }
        a[j + 1] = v;
    }
}
```

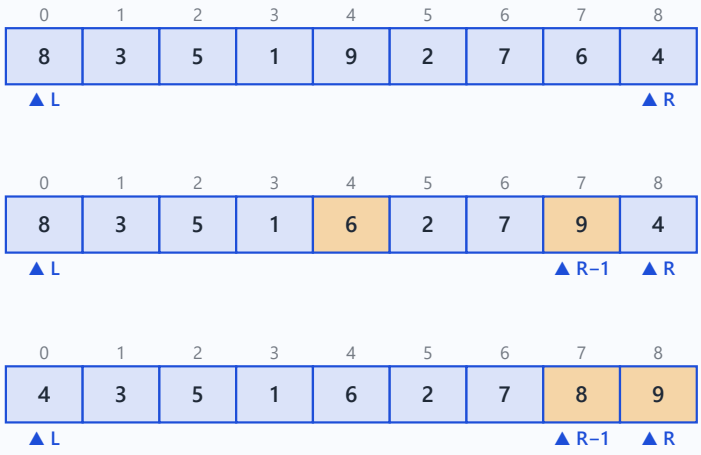
Eklemeli sıralamanın maliyeti, öğelerin kaç konum kaydığına bağlıdır. Burada hiçbir öğe kendi parçasının dışına kaymaz. Bu yüzden son geçiş $O(n \cdot M)$, sabit M için doğrusal sürer.

20.4.5 Kod izleme: `quicksort3(a, 0, 8)`

İzlemeyi kısa tutmak için $M = 3$ alınır. Dizi *[8, 3, 5, 1, 9, 2, 7, 6, 4]*'tür. Tablo hazırlık adımlarını ve özyinelemeyi verir.

Satır	Olay	Dizi (sonra)
3	$R - L = 8 > 3$: parça büyük, bölümlenecek	`8 3 5 1 9 2 7 6 4`
4	orta öğe `a[4] = 9` ile `a[7] = 6` takas	`8 3 5 1 6 2 7 9 4`
5	$a[7] = 9 \geq a[0] = 8$: takas yok	aynı
6	$a[8] = 4 < a[0] = 8$: takas	`4 3 5 1 6 2 7 9 8`
7	$a[8] = 8 < a[7] = 9$: takas	`4 3 5 1 6 2 7 8 9`
8	`bol(a, 1, 7)` : pivot 8, konum 7	aynı
9	sol `a[0..6]` : $R - L = 6 > 3$; medyan-3 ile `a[0..6] = 1 3 5 2 6 4 7`	`1 3 5 2 6 4 7 8 9`
9 → 8	`bol(a, 1, 5)` : pivot 4, konum 3	`1 3 2 4 6 5 7 8 9`
9 → 9, 10	`a[0..2]` ve `a[4..6]` : $R - L = 2 \leq 3$, dönerler; ardından satır 10'da sağ `a[8..8]` ($R - L = 0 \leq 3$) döner	aynı
`sirala`	son eklemeli sıralama geçişi	`1 2 3 4 5 6 7 8 9`

Tablo 20.7: `quicksort3(a, 0, 8)` izlemesi, $M = 3$.



Şekil 20.5: Medyan-3 hazırlığı, yukarıdan aşağıya: başlangıç; orta öğe `a[R-1]` 'e taşındıktan sonra; üç karşılaştırmadan sonra $a[L] = 4 \leq a[R-1] = 8 \leq a[R] = 9$.

Bu izlemede ilk pivot şanssızdır. Üç aday 8, 9 ve 4'tür; ortancaları 8, dizinin ikinci en büyük öğesidir. Bölme 7'ye 1 olur. Medyan-3 kötü bölmeyi tümüyle önlemez, yalnız olasılığını azaltır. Sol parçada aday öğeler 4, 1 ve 7'dir. Ortanca 4 bu kez parçanın tam ortasına düşer. `quicksort3` bittiğinde dizi *[1, 3, 2, 4, 6, 5, 7, 8, 9]*'dur: parçalar birbirine göre sıralıdır, ama 3 ile 2 ve 6 ile 5 kendi parçalarında yer değiştirmiş durumdadır. Son eklemeli geçiş bu iki çifti iki kaydırmayla düzeltir.

$M = 10$ ile bu dokuz öğelik dizide `quicksort3` ilk satırda döner ve bütün işi eklemeli sıralama yapar. İki sürüm de 3000 rastgele, tekrarlı diziyle sınanmıştır.

20.4.6 Eşiğin etkisi ve sınırları

Kaynaklarda $5 \leq M \leq 25$ aralığındaki eşiklerin süreyi yaklaşık %10 azalttığı belirtilir. Kazanç makineye bağlıdır ve küçüktür; ölçmek için çok tekrar gerekir. Tablo, bir milyon rastgele öğenin `sirala` ile sıralanma süresini farklı M değerleri için verir. $M = 1$, eşiğin pratikte kapalı olduğu durumdur: medyan-3 için en az üç öğe gerektiğinden en küçük anlamlı değer budur.

M	1	5	10	15	25	50
Süre (s)	0,105	0,092	0,089	0,101	0,089	0,093

Tablo 20.8: *n = 1.000.000* rastgele öğede `sirala` süresi (`zig cc -O2` , x86-64; üç ölçümün ortancası, her ölçüm beş sıralamanın ortalaması). Süreler makineye göre değişir ve tek tek ölçümler ± %15 oynar.

Eşik süreyi %10–15 kısaltmış, 5 ile 50 arasındaki değerler arasında belirgin bir fark çıkmamıştır. Farklar gürültüye yakın olduğu için ölçüm tekrarlanıp ortanca alınmıştır.

Medyan-3'ün bir sınırı vardır. Sıralı ve ters sıralı girdiyi kurtarır, ama her girdiyi kurtarmaz. Ortanca adaylarının hep küçük çıkacağı biçimde özel olarak kurulmuş girdiler yine n^2 sürer. Rastgele veride bu olasılık ihmal edilebilir; ama girdiyi bir saldırganın seçebildiği sistemlerde, örneğin bir ağ sunucusunda, bu bir zayıflıktır. Kesin çözüm, en kötü durum güvencesi veren hibrit bir yöntemdir: özyinelemesi fazla derinleşen parçalar heap sort ile sıralanır.

Böyle bir ölçümün bugün birkaç dakika sürmesi de yeni bir durumdur. Eskiden program delikli kartlara aktarılır, bir kişiye günde bir, en çok iki kez derleme sırası gelirdi. Zorunlu stajların bu kıtlık döneminde, uygulama eksikliğini kapatmak için konduğu düşünülebilir. Bugün günde onlarca kez derlemek mümkündür. Dersin amacı da hazır kalıpları uygulayan değil, bir algoritmayı kanıtlayan, ölçen ve gerekirse yenisini kuran, bilim insanı gibi düşünen mühendisler yetiştirmektir.

20.5 C kitaplığı ve hibrit sıralamalar

20.5.1 `qsort`

Standart C kitaplığı, `<stdlib.h>` başlığında genel amaçlı bir sıralama yordamı sunar:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Parametreler dizinin adresi, öğe sayısı, bir öğenin bayt cinsinden boyu ve karşılaştırma fonksiyonunun adresidir. `qsort` öğe türünü bilmez; diziyi `size` baytlık bloklar olarak görür. İlk parametre bu yüzden bir **void işaretçi**dir (*void pointer*): her nesnenin adresini taşıyabilen ama tür bilgisi taşımayan işaretçi. `void *` doğrudan çözülemez; önce gerçek türe dönüştürülür.

Dördüncü parametre bir **fonksiyon işaretçisi**dir (*function pointer*): iki `const void *` alıp `int` döndüren bir fonksiyonun adresi. C'de ayraçsız yazılan fonksiyon adı fonksiyonun adresini verir. `qsort` iki öğeyi karşılaştırırken bu fonksiyonu öğelerin adresleriyle çağırır; karşılaştırıcı değiştirilerek her tür sıralanabilir.

```
int artan(const void *p, const void *q) {
    int x = *(const int *)p;
    int y = *(const int *)q;
    return (x > y) - (x < y);  // -1, 0, +1
}
int main(void) {
    int a[] = {37, 15, 24, 33, 49};
    size_t n = sizeof a / sizeof a[0];
    qsort(a, n, sizeof a[0], artan);
    for (size_t i = 0; i < n; i++)
        printf("%d ", a[i]);
    return 0;
}
```

Program `<stdio.h>` ve `<stdlib.h>` başlıklarıyla derlenir ve `15 24 33 37 49` yazar.

- `artan` . Gelen `void` işaretçiler `const int *` 'a dönüştürülür ve çözülür. `const` , karşılaştırma fonksiyonunun öğeleri değiştirmeyeceğini belirtir. Dönüş değeri `(x > y) - (x < y)` 'dir. C'de bir karşılaştırma işleci doğruysa 1, yanlışsa 0 verir. *x > y* ise sonuç *1 − 0 = 1*, *x < y* ise *0 − 1 = −1*, eşitse 0'dır.

- `sizeof a / sizeof a[0]`. Dizinin toplam boyu bir öğenin boyuna bölünür. `int` 4 bayt olduğunda *20 / 4 = 5* bulunur. Bu ifade yalnız dizinin tanımlandığı yerde doğru sonuç verir. Dizi bir fonksiyona parametre olarak geçirildiğinde işaretçiye dönüşür ve `sizeof` işaretçinin boyunu verir.
- `qsort(a, n, sizeof a[0], artan)`. `artan` ayraçsız yazılmıştır; çağrılmaz, adresi geçirilir. Öğe boyu `sizeof(int)` yerine `sizeof a[0]` ile yazılmıştır. Dizinin türü değişirse bu ifade kendiliğinden doğru kalır.

20.5.2 Kod izleme: `qsort` ve karşılaştırma fonksiyonu

Adım	Olay	Değerler
1	dizi hazır; `n = 20 / 4 = 5`	`a = [37, 15, 24, 33, 49]`
2	`qsort(a, 5, 4, artan)` çağrılır	yığında `main`, `qsort`
3	`artan(&37, &15)`	*x = 37, y = 15*, dönüş 1: ilk öğe sonra gelmeli
4	`artan(&15, &24)`	dönüş −1: ilk öğe önce gelmeli
5	`artan(&24, &24)`	dönüş 0: eşit, göreli sıra güvence altında değil
6	`qsort` döner	`a = [15, 24, 33, 37, 49]`
7–11	döngü öğeleri yazar	çıktı: `15 24 33 37 49`

Tablo 20.9: Örnek programın izlemesi. `artan` satırları örnek çağrılardır; `qsort`'un hangi çiftleri hangi sırayla karşılaştıracağını standart belirtmez.



Şekil 20.6: `qsort` çağrısından önce (solda) ve sonra (sağda) dizi.

Karşılaştırma fonksiyonu bir sözleşmeye uymalıdır. Birinci öğe ikinciden önce gelmeliyse negatif, sonra gelmeliyse pozitif, ikisi eşitse sıfır döndürür. Dönüş değerinin büyüklüğü önemsizdir, yalnız işareti kullanılır. Fonksiyon tutarlı da olmalıdır: $a < b$ ve $b < c$ diyorsa $a < c$ demelidir. Aynı iki öğe için her çağrıda aynı cevabı vermeli, öğeleri değiştirmemelidir. Bu kurallardan biri bozulursa `qsort`'un davranışı tanımsızdır: dizi yanlış sıralanabilir, hatta bazı gerçeklemelerde dizinin dışına çıkılabilir.

20.5.3 Kayıt sıralamak

Gerçek uygulamalarda sıralanan şey genellikle bir sayı değil, birden çok alanı olan bir **kayıttır** (*record*). Kayıtlar C'de `struct` ile gösterilir. Sıralamanın hangi alana göre yapılacağını karşılaştırma fonksiyonu belirler:

```c
typedef struct { char ad[20]; int not_; } ogrenci;

int ada_gore(const void *p, const void *q) {
    const ogrenci *x = p, *y = q;
    return strcmp(x->ad, y->ad);        // alfabetik
}
int nota_gore_azalan(const void *p, const void *q) {
    const ogrenci *x = p, *y = q;
    return (y->not_ > x->not_) - (y->not_ < x->not_);
}
/* qsort(s, n, sizeof s[0], ada_gore); */
```

`ogrenci` yapısı 20 karakterlik bir ad ve bir not taşır; boyu 24 bayttır. `not_`'taki alt çizgi, `<iso646.h>`'teki `not` makrosuyla ve C++'taki `not` işleciyle çakışmayı önler.

C'de `void *` açık dönüştürme yazılmadan başka bir nesne işaretçisine atanabilir; `const ogrenci *x = p` bu yüzden geçerlidir. `strcmp` zaten `qsort`'un sözleşmesine uyan bir değer döndürür. `nota_gore_azalan`'da `x` ile `y`'nin yeri değiştirilerek yüksek not öne alınmıştır.

Bu iki fonksiyon *{Zeynep 85, Ali 70, Mehmet 85, Ayse 92, Can 70}* dizisiyle denenmiştir. `ada_gore` ile sıralama *Ali, Ayse, Can, Mehmet, Zeynep* verir. Bu sıralı dizi ardından `nota_gore_azalan` ile sıralandığında sonuç *Ayse 92, Mehmet 85, Zeynep 85, Can 70, Ali 70* olmuştur. Notu 70 olan iki öğrencinin alfabetik sırası bozulmuştur. Bu bir hata değildir. Standart, `qsort` 'un kararlı olmasını ya da belirli bir algoritmayı kullanmasını istemez. Eşit anahtarlarda sıra önemliyse karşılaştırma fonksiyonuna ikincil bir anahtar eklenir:

```
int nota_sonra_ada(const void *p, const void *q) {
    const ogrenci *x = p, *y = q;
    int k = (y->not_ > x->not_) - (y->not_ < x->not_);
    if (k != 0) return k;              /* notlar farklı: yüksek not önce */
    return strcmp(x->ad, y->ad);       /* notlar eşit: alfabetik */
}
```

Bu fonksiyonla sonuç *Ayse 92, Mehmet 85, Zeynep 85, Ali 70, Can 70* olur ve girdinin sırasından bağımsızdır.

> **Sık yapılan hata**
>
> Karşılaştırma fonksiyonunu `return x - y;` diye kısa yoldan yazmak. Küçük değerlerde doğru çalışır, ama fark `int` 'e sığmadığında **işaretli tam sayı taşması** olur ve bu C'de tanımsız davranıştır. *x = INT_MIN, y = 1* için gerçek fark *−2.147.483.649*'dur ve `int` 'in alt sınırının altındadır. `-fsanitize=undefined` ile derlenen program bu satırda "signed integer overflow" iletisiyle durmuştur. Taşmanın sarmalandığı bir derlemede (`-fwrapv`) sonuç *+2.147.483.647* çıkmış, yani `INT_MIN` 1'den büyük sayılmıştır. `(x > y) - (x < y)` hiçbir değerde taşmaz.

Kitaplık yordamının bedeli, her karşılaştırmanın bir fonksiyon çağrısı olmasıdır. Bir milyon rastgele `int` `qsort` ile 0,21 saniyede, `sirala` ile 0,10 saniyede sıralanmıştır (`-O2` ; süreler makineye göre değişir). Yine de `qsort` sınanmış ve her türle çalışan hazır bir yordamdır; çoğu programda doğru seçim odur. Hangi algoritmayı kullandığı gerçeklemeden gerçeklemeye değişir.

20.5.4 Hibrit sıralamalar

Tek bir algoritma her durumda en iyi değildir. Hızlı sıralama ortalamada hızlıdır ama en kötü durumda n^2'ye çıkar. Heap sort en kötü durumda da $n \lg n$'dir ama sabiti büyüktür. Eklemeli sıralama küçük ve neredeyse sıralı dizilerde rakipsizdir. **Hibrit sıralama** (*hybrid sort*) bu güçlü yanları birleştirir:

- **Hızlı + eklemeli.** Küçük parçalarda özyinelemenin maliyeti baskındır. M'den küçük parçalar eklemeli sıralamayla bitirilir. `quicksort3` ile `sirala` bunun örneğidir.
- **Introsort** (Musser, 1997). Hızlı sıralamayla başlar ve özyineleme derinliğini izler. Derinlik yaklaşık $2\lfloor \lg n \rfloor$'i aşarsa, bu, bölmelerin kötü gittiğinin işaretidir; o parça heap sort ile sıralanır. Böylece en kötü durum $O(n \lg n)$'ye iner. Küçük parçalar yine eklemeliyle bitirilir. C++ kitaplığındaki `std::sort` gerçeklemelerinde yaygındır.
- **Timsort** (Peters, 2002). Birleştirme sıralaması ile eklemeli sıralamayı birleştirir. Verideki hazır sıralı bölümleri, "koşuları" (*run*) bulur ve onları birleştirir. Gerçek verideki düzeni kullandığı için neredeyse sıralı girdilerde doğrusal süreye yaklaşır. Kararlıdır. Python'un `sort` 'u ve Java'nın nesne dizileri için `Arrays.sort` 'u Timsort'tur.

Ortak fikir üç katmanlıdır: büyük ölçekte iyi asimptotik davranış ($n \lg n$), küçük ölçekte düşük sabit (eklemeli) ve en kötü durum için bir sigorta. Gerçek kitaplıklardaki sıralama yordamları ders kitaplarındaki saf algoritmalardan bu yüzden farklıdır.

20.6 Birleştirme sıralaması

20.6.1 Fikir

Birleştirme sıralaması (*merge sort*) John von Neumann'ın 1945'te tanımladığı bir böl ve fethet algoritmasıdır. Dört cümleyle anlatılır:

1. Liste 0 ya da 1 öğeliyse zaten sıralıdır.
2. Değilse liste olabildiğince eşit iki parçaya ayrılır.
3. Her parça birleştirme sıralaması özyinelemeli uygulanarak sıralanır.
4. Kendi içinde sıralı iki alt liste tek bir sıralı listede birleştirilir.

Algoritma iki gözleme dayanır. Birincisi, küçük bir listeyi sıralamak daha az adım ister. İkincisi, sıralı iki listeden tek bir sıralı liste elde etmek, sırasız iki listeden elde etmekten çok daha ucuzdur. Sıralı iki listenin en küçük öğesi her zaman baştadır; birleştirilmiş listenin sıradaki öğesi bu iki baştan küçük olanıdır. Bu yüzden birleştirme, iki listenin uzunlukları toplamı kadar adımda biter.

Hızlı sıralamada her bölümleme pivotu kesin yerine koyar. Birleştirme sıralamasında bölme sırasında hiçbir öğe yerine oturmaz; algoritma sıralı iki dizinin "sıralı olma" avantajını kullanır. Girdiyi ortadan ikiye bölmek hızlı sıralamada şansa bağlıydı; burada her zaman böyledir.

20.6.2 `merge`: iki uçtan birleştirme

Kod Sedgewick'in kitabındaki sürümdür. Birleştirmeyi yerinde yapmak zordur; bu yüzden `a` ile aynı boyda yardımcı bir `b` dizisi kullanılır.

```
void merge(int a[], int b[], int L, int m, int R) {
    int i, j, k;
    for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
    for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
    for (k = L; k <= R; k++)
        if (kucuk(b[j], b[i]))
            a[k] = b[j--];
        else
            a[k] = b[i++];
}
```

`merge(a, b, L, m, R)`, kendi içinde sıralı olan `a[L..m]` ve `a[m+1..R]` yarılarını `a[L..R]`'de tek bir sıralı dizi olarak birleştirir. Satır satır:

- **Satır 3.** Sol yarı olduğu gibi `b[L..m]`'ye kopyalanır. Döngü `i = m + 1`'den başlar, `b[i-1] = a[i-1]` ataması yapar ve `i = L` olunca biter. Döngüden sonra `i`, sol yarının ilk öğesini gösterir.
- **Satır 4.** Sağ yarı `b[m+1..R]`'ye *ters sırada* kopyalanır. `j = m` için `b[R] = a[m+1]`, `j = R - 1` için `b[m+1] = a[R]` yazılır. Döngü `j = R` ile biter; `j`, sağ yarının en küçük öğesini gösterir.
- **Satır 5–9.** `k`, `a`'da yazılacak konumdur. Her adımda `b[i]` ile `b[j]` karşılaştırılır ve küçük olan `a[k]`'ye yazılır. Sağdan alınırsa `j` azalır, soldan alınırsa `i` artar. Eşitlikte `kucuk(b[j], b[i])` yanlıştır ve soldaki öğe alınır.

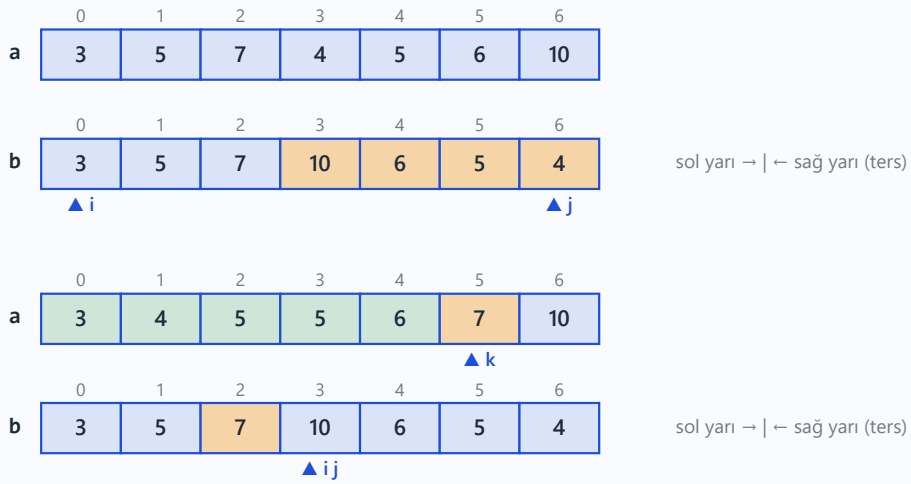
Ters kopyalama kodun en ince noktasıdır. Kopyadan sonra `b[L..R]` önce artan, sonra azalan, yani **bitonik** bir dizidir. Sıradan birleştirmede her adımda "bu yarı bitti mi?" denetimi gerekir. Burada sol yarı bitince `i` sağ yarının büyük öğelerine girer ve bunlar `b[j]`'den küçük değildir; döngü artık hep `j`'den alır. Sağ yarı için de durum simetriktir: her yarı ötekinin bekçisidir.

20.6.3 Kod izleme: `merge(a, b, 0, 2, 6)`

`a = [3, 5, 7, 4, 5, 6, 10]` olsun. Sol yarı `a[0..2] = 3 5 7`, sağ yarı `a[3..6] = 4 5 6 10`'dur ve ikisi de sıralıdır. İki kopyalama döngüsünden sonra `b = [3, 5, 7, 10, 6, 5, 4]` olur; *i = 0, j = 6*'dır.

`k`	`b[i]`	`b[j]`	Karar	`i`, `j` (sonra)	`a` (sonra)
0	3	4	*4 ≥ 3*: `a[0] = b[0]`	1, 6	`3 5 7 4 5 6 10`
1	5	4	*4 < 5*: `a[1] = b[6]`	1, 5	`3 4 7 4 5 6 10`
2	5	5	*5 ≥ 5*: `a[2] = b[1]`	2, 5	`3 4 5 4 5 6 10`
3	7	5	*5 < 7*: `a[3] = b[5]`	2, 4	`3 4 5 5 5 6 10`
4	7	6	*6 < 7*: `a[4] = b[4]`	2, 3	`3 4 5 5 6 6 10`
5	7	10	*10 ≥ 7*: `a[5] = b[2]`	3, 3	`3 4 5 5 6 7 10`
6	10	10	*10 ≥ 10*: `a[6] = b[3]`	4, 3	`3 4 5 5 6 7 10`

Tablo 20.10: `merge(a, b, 0, 2, 6)`'nın üçüncü döngüsü. `b = [3, 5, 7, 10, 6, 5, 4]`.



Şekil 20.7: `merge` izlemesinin iki anı. Üstte kopyalamadan sonra: `b`'nin sol yarısı artan, sağ yarısı ters sırada; uçlardan ortaya doğru iki taraf da artar. Altta *k = 5* adımı: sol yarının son öğesi 7 alınmış, `i` ile `j` aynı hücrede buluşmuştur.

k = 5'te sol yarının son öğesi 7 alınır ve `i`, 3'e, yani sağ yarının içine girer. Bu noktada `b[i] = b[j] = 10`'dur; aynı hücreyi iki indis birlikte gösterir. Son adımda bu öğeyi `i` alır. Sonunda *i = 4*, *j = 3*'tür; iki indis ortada buluşmuştur. Hiçbir sınır denetimi yapılmamış, buna gerek de olmamıştır. Eşit 5'lerde soldaki, yani `b[i]` önce alınmıştır.

Birleştirme $R - L + 1$ öğeyi kopyalar ve her yerleştirmede tam bir karşılaştırma yapar. Zaman maliyeti $\Theta(R - L + 1)$'dir; iki yarının boyu ne olursa olsun bu değişmez.

20.6.4 `mergesort` ve yardımcı dizi

```
void mergesort(int a[], int b[], int L, int R) {
    if (R <= L) return;
    int m = (R + L) / 2;
    mergesort(a, b, L, m);
    mergesort(a, b, m + 1, R);
    merge(a, b, L, m, R);
}
```

Yordam aralık boş ya da tek öğeliyse döner. Değilse ortayı `m = (R + L) / 2` ile bulur, iki yarıyı sıralar ve birleştirir. Aralık tek sayıda öğeliyse sol yarı bir fazla öğe alır: 17 öğe 9 + 8 diye bölünür.

Yardımcı dizi `b` parametre olarak geçirilir. Her `merge` çağrısında yeniden ayrılsaydı her çağrı bir `malloc` ve bir `free` yapardı; bu, $n - 1$ kez bellek isteğinin yükü demektir. Doğrusu, diziyi bir kez ayırıp bütün çağrılara aynı diziyi vermektir. Bunu bir sarmalayıcı fonksiyon yapabilir:

```
int birlestirme_sirala(int a[], int n) {
    int *b = malloc(n * sizeof *b);   /* a ile aynı boyda */
    if (b == NULL) return 0;          /* bellek yok: sıralanmadı */
    mergesort(a, b, 0, n - 1);
    free(b);
    return 1;
}
```

`malloc`'un döndürdüğü işaretçi `NULL` ise bellek ayrılamamıştır. Bu durumda dizi sıralanmadan 0 döndürülür; çağıran durumu bu dönüş değerinden anlar. `n * sizeof *b` ifadesi, `b`'nin gösterdiği türün boyunu kullanır. `b`'nin türü değişse bile ifade doğru kalır. İş bitince blok `free` ile geri verilir. `free` unutulsaydı her çağrıda $4n$ bayt kaybolurdu; bu bir **bellek sızıntısıdır**. Sarmalayıcı, 10 öğelik bir diziyle ve sayaçlı bir `malloc` / `free` çiftiyle sınanmıştır: bir ayırma, bir geri verme.

Sık yapılan hata

`b[]` 'yi hiç ayırmamak, yani `int *b;` bildirip doğrudan `mergesort(a, b, 0, n - 1)` çağırmak. `b` başlangıç değeri verilmemiş bir işaretçidir ve rastgele bir adresi gösterir. `merge` bu adrese yazar; program ya çöker ya da başka değişkenlerin üzerine yazar. Tersi de bir hatadır: `b` 'yi `merge` 'ün içinde her çağrıda `malloc` ile ayırmak. Program doğru çalışır, ama gereksiz yere yavaşlar.

20.6.5 Kod izleme: dokuz `merge` çağrısı

Dizi *[89, 67, 59, 62, 23, 93, 24, 20, 28, 60]* olsun. `mergesort(a, b, 0, 9)` önce sol yarıyı tamamen sıralar. İlk birleştirme ancak en derindeki *(0, 1)* çağrısında olur. Tablo on öğe için yapılan dokuz birleştirmeyi sırasıyla verir.

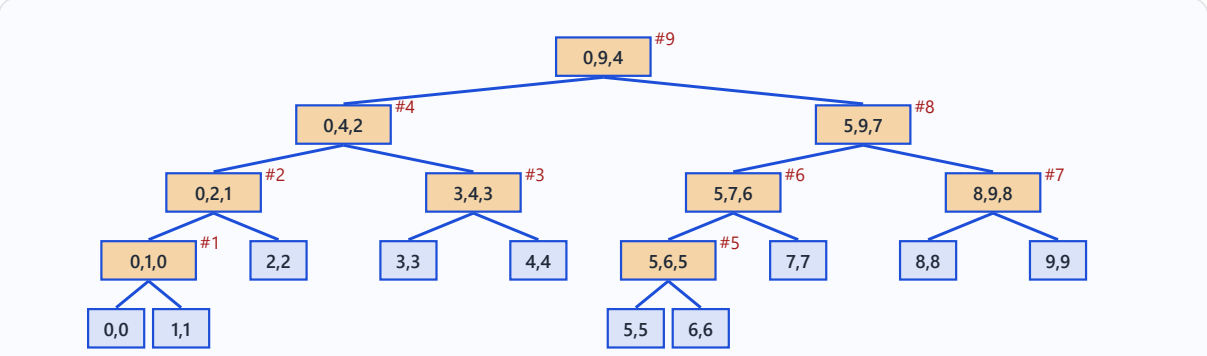
#	`(L, m, R)`	Birleşen yarılar	`a` (sonra)
1	(0, 0, 1)	`a[0..0]` + `a[1..1]`	**67 89** 59 62 23 93 24 20 28 60
2	(0, 1, 2)	`a[0..1]` + `a[2..2]`	**59 67 89** 62 23 93 24 20 28 60
3	(3, 3, 4)	`a[3..3]` + `a[4..4]`	59 67 89 **23 62** 93 24 20 28 60
4	(0, 2, 4)	`a[0..2]` + `a[3..4]`	**23 59 62 67 89** 93 24 20 28 60
5	(5, 5, 6)	`a[5..5]` + `a[6..6]`	23 59 62 67 89 **24 93** 20 28 60
6	(5, 6, 7)	`a[5..6]` + `a[7..7]`	23 59 62 67 89 **20 24 93** 28 60
7	(8, 8, 9)	`a[8..8]` + `a[9..9]`	23 59 62 67 89 20 24 93 **28 60**
8	(5, 7, 9)	`a[5..7]` + `a[8..9]`	23 59 62 67 89 **20 24 28 60 93**
9	(0, 4, 9)	`a[0..4]` + `a[5..9]`	**20 23 24 28 59 60 62 67 89 93**

Tablo 20.11: `mergesort(a, b, 0, 9)` izlemesi. Her satır bir `merge(L, m, R)` çağrısıdır; kalın kısım o anda sıralı hâle gelen aralıktır.

7. birleştirmede dizi değişmez; 28 ve 60 zaten sıralıdır. `merge` bunu bilmez ve iki öğeyi yine kopyalayıp geri yazar. Birleştirme sıralaması girdinin düzeninden yararlanmaz; analizinin ana sonucu da budur.

20.6.6 Çerçeve ağacı

Özyinelemeli çağrıların düzeni bir ağaçla gösterilebilir. Her düğüm bir `mergesort` çağrısının çerçevesidir ve içinde *L*, *R* ve *m* değerleri durur. Şekil bu ağacı verir. Algoritma soldan derine iner, dönüşte birleştirir. Bir düğümdeki `merge` ancak iki çocuğunun işi bittikten sonra yapılır; bu, ağacın **kök sonda** gezinmesidir.



Şekil 20.8: `mergesort(a, b, 0, 9)` 'un çerçeve ağacı. Kutularda *L, R, m* yazar. Turuncu kutular `merge` çağıran çerçevelerdir; yanlarındaki *#k*, kaçıncı birleştirme olduğunu gösterir. Yapraklar tek öğeli aralıklardır ve hemen döner.

Ağaçta 19 düğüm vardır: 9 iç düğüm `merge` yapar, 10 yaprak tek öğeli aralıklardır. Ağacın yüksekliği *⌈lg 10⌉ = 4*'tür; çağrı yığınında aynı anda en çok beş `mergesort` çerçevesi bulunur. Sıralı iki kaynaktan tek bir sıralı çıktı üretme mantığı, veri yapılarında "merge" adıyla sık sık karşımıza çıkar.

20.6.7 Özyinelemeli satırların sırası

Yordamdaki iki `mergesort` satırının yerleri değiştirilsin. Önce sağ yarı, sonra sol yarı sıralanır. Ne değişir? Sonuç değişmez, çünkü iki yarı birbirinden bağımsızdır ve `merge` ikisi de bittikten sonra çağrılır. Değişen, birleştirmelerin sırası ve ara tablolardır. Tablo aynı dizi için yeni sırayı verir.

#	Aralık	Dizi (sonra)
1	(8, 9)	`89 67 59 62 23 93 24 20 28 60`
2	(5, 6)	`89 67 59 62 23 24 93 20 28 60`
3	(5, 7)	`89 67 59 62 23 20 24 93 28 60`
4	(5, 9)	`89 67 59 62 23 20 24 28 60 93`
5	(3, 4)	`89 67 59 23 62 20 24 28 60 93`
6	(0, 1)	`67 89 59 23 62 20 24 28 60 93`
7	(0, 2)	`59 67 89 23 62 20 24 28 60 93`
8	(0, 4)	`23 59 62 67 89 20 24 28 60 93`
9	(0, 9)	`20 23 24 28 59 60 62 67 89 93`

Tablo 20.12: İki özyinelemeli satırın yeri değiştiğinde birleştirmeler. Önce sağ yarılar birleşir.

Çerçeve ağacı aynıdır; yalnız her düğümde önce sağ çocuk gezilir.

20.6.8 Analiz ve kararlılık

n öğeli bir dizi $\lceil n/2 \rceil$ ve $\lfloor n/2 \rfloor$ öğeli iki yarıya bölünür; birleştirme n karşılaştırma yapar. Karşılaştırma sayısı a_n şu bağıntıya uyar:

$$a_n = a_{\lceil n/2 \rceil} + a_{\lfloor n/2 \rfloor} + n,\ a_1 = 0$$

Örneğin 16 öğe 8 + 8, 17 öğe 9 + 8 diye bölünür. $n = 2^k$ ise iki yarı eşittir ve bağıntı $a_n = 2a_{n/2} + n$ olur. Bu, hızlı sıralamanın en iyi durum bağıntısıdır ve $a_n = n \lg n$ verir.

Teorem 20.3 (Birleştirme sıralamasının maliyeti)

`mergesort` n öğeli her dizide $\Theta(n \lg n)$ karşılaştırma yapar; $n = 2^k$ ise sayı tam $n \lg n$'dir. $2n - 1$ çağrı ve $n - 1$ birleştirme yapılır. Ek bellek n öğelik yardımcı dizi ile $O(\lg n)$ derinlikli çağrı yığınıdır.

İspat. Her `merge`, aralığındaki öğe sayısı kadar karşılaştırma yapar ve bu sayı girdinin düzenine bağlı değildir. Bu yüzden karşılaştırma sayısı yalnız n'ye bağlıdır ve a_n'ye eşittir. Çerçeve ağacının her düzeyinde aralıklar diziyi örter ve düzeyin toplam işi en çok n'dir; düzey sayısı $\lceil \lg n \rceil$'dir. Böylece $a_n \le n\lceil \lg n \rceil$ olur. Alt sınır için: $d < \lfloor \lg n \rfloor$ derinliğindeki her aralığın boyu en az $\lfloor n/2^d \rfloor \ge 2$'dir; bu ilk $\lfloor \lg n \rfloor$ düzeyin her biri tam n karşılaştırma yapar ve $a_n \ge n\lfloor \lg n \rfloor$ olur. Çağrı sayısı için $T_n = T_{\lceil n/2 \rceil} + T_{\lfloor n/2 \rfloor} + 1$, $T_1 = 1$ bağıntısı $T_n = 2n - 1$ verir: $n \ge 2$ için $(2\lceil n/2 \rceil - 1) + (2\lfloor n/2 \rfloor - 1) + 1 = 2n - 1$. Ağacın n yaprağı vardır; geri kalan $n - 1$ düğüm birleştirme yapar.

Sayaçlı çalıştırma bu değerleri doğrulamıştır. 10 öğe için 19 çağrı, 9 birleştirme ve 34 karşılaştırma; 16 öğe için 31 çağrı, 15 birleştirme ve tam $16 \cdot 4 = 64$ karşılaştırma; 17 öğe için 33 çağrı ve 70 karşılaştırma yapılmıştır. Bağıntı elle de açılabilir: $a_{10} = 2a_5 + 10$, $a_5 = a_3 + a_2 + 5 = 5 + 2 + 5 = 12$ ve $a_{10} = 34$.

Süre dizinin durumundan bağımsızdır: en iyi, ortalama ve en kötü durumun üçü de $n \lg n$'dir; kötü bir girdi yoktur. Bedeli $O(n)$ ek bellektir. Bunu kaldırmanın yolları maliyeti artırır; ek bellek istenmiyorsa heap sort öne çıkar.

Birleştirme, sıralı parçaları sırayla okuyup sırayla yazar. Bu özellik onu, belleğe sığmayan dosyaların diskte sıralanmasının, yani **dış sıralamanın** temeli yapar: belleğe sığan parçalar sıralanıp diske yazılır, sonra birleştirilir.

Kararlılık konusu bir ayrım gerektirir. Klasik birleştirme, eşitlikte sol yarıdaki öğeyi alır ve kalan öğeleri ayrıca kopyalar; bu birleştirme **kararlıdır**. Hızlı sıralama ve heap sort kararlı değildir; birleştirme sıralaması bu yönüyle onlardan ayrılır. Ancak buradaki ters kopyalı, bitonik `merge` kararlı değildir. Sol yarı bittiğinde `i` sağ yarının ters sıradaki öğelerine girer ve eşit öğeleri ters sırada alır. Dört eşit anahtar

[2a, 2b, 2c, 2d] ile sıralama *[2a, 2b, 2d, 2c]* vermiştir: 2c ile 2d yer değiştirmiştir. Değerleri 0–3 arasında, 2–10 öğelik 1000 rastgele diziyle yapılan denemede 194 dizide eşitlerin sırası bozulmuştur. Kararlılık gerekiyorsa sağ yarı ters kopyalanmaz ve yarılardan birinin bitip bitmediği denetlenir.

20.7 Bağlı birleştirme sıralaması

20.7.1 Büyük kayıtlar ve `bag` alanı

Gerçek uygulamalarda öğe, örneğin bir öğrencinin adını ve adresini taşıyan yüzlerce baytlık bir kayıt olabilir. Böyle kayıtları her birleştirmede iki dizi arasında kopyalamak zaman alır. Çözüm, kayıtları hiç taşımamaktır: her kayda bir `bag` alanı eklenir ve dizinin içinde indislerle bir bağlı liste kurulur. Sıralama yalnız indisleri değiştirir.

```
typedef struct {
    int anahtar;   /* + büyük veri alanları */
    int bag;       /* sonraki öğenin indisi, 0 = son */
} eleman;

int mergesort(eleman a[], int L, int R) {
    if (R <= L) return L;          /* tek öğeli liste */
    int m = (R + L) / 2;
    int sag = mergesort(a, m + 1, R);
    int sol = mergesort(a, L, m);
    return merge(a, sol, sag);     /* baş indisi */
}
```

`eleman` yapısı bir anahtar ve bir bağ taşır. Açıklamadaki "+ büyük veri alanları" ifadesi, gerçek uygulamada yapının çok daha büyük olabileceğini hatırlatır. `bag`, listede bu öğeden sonra gelen öğenin dizideki indisidir; 0 listenin sonunu gösterir. Bu yüzden veriler `a[1..n]`'de durur ve `a[0]` öğe olarak kullanılmaz. `a[0]`, birleştirme sırasında geçici bir liste başı olarak iş görür. Başlangıçta bütün `bag` alanları 0'dır: her öğe tek başına bir listedir.

`mergesort` bu kez bir değer döndürür: `a[L..R]` aralığındaki öğelerden kurulan sıralı listenin baş indisini. Tek öğeli aralıkta liste o öğenin kendisidir ve `L` döndürülür. Daha büyük aralıkta iki yarı ayrı ayrı sıralanır, döndükleri listeler `merge` ile birleştirilir ve yeni listenin başı döndürülür.

Kodda sağ yarı önce sıralanır ve bu sıra bilinçli olarak iki ayrı satırda yazılmıştır. İki çağrı `merge`'ün argümanları olarak yazılsaydı hangisinin önce çalışacağı belli olmazdı: C'de argümanların değerlendirilme sırası **belirtilmemiştir**. Sonuç aynı çıkar, ama ara durumlar ve elle hazırlanan izleme tablosu derleyiciye göre değişir.

20.7.2 Bağlı `merge`

```
int merge(eleman a[], int birinci, int ikinci) {
    int sonuc = 0;              // a[0]: geçici baş
    int i1 = birinci, i2 = ikinci;
    while (i1 && i2)
        if (a[i1].anahtar <= a[i2].anahtar) {
            a[sonuc].bag = i1; sonuc = i1;
            i1 = a[i1].bag;
        } else {
            a[sonuc].bag = i2; sonuc = i2;
            i2 = a[i2].bag;
        }
    a[sonuc].bag = i1 ? i1 : i2;
    return a[0].bag;
}
```

`merge(a, birinci, ikinci)`, başları `birinci` ve `ikinci` olan iki sıralı listeyi tek sıralı listede birleştirir ve yeni listenin baş indisini döndürür. Satır satır:

- **Satır 2.** `sonuc`, birleşmiş listenin o ana kadarki son öğesidir. Başta liste boştur ve `sonuc = 0`, yani geçici baş `a[0]`'dır. Bu küçük hile, "ilk öğe mi?" diye ayrı bir durum yazma zorunluluğunu kaldırır. İlk eklenen öğe de `a[sonuc].bag` üzerinden, yani `a[0].bag`'e yazılır.

- **Satır 3.** `i1` ve `i2`, iki listenin henüz alınmamış ilk öğeleridir.
- **Satır 4–11.** İki listede de öğe kaldıkça döngü sürer. Küçük anahtarlı öğe birleşmiş listenin sonuna bağlanır (`a[sonuc].bag = i1`), `sonuc` ona ilerler ve alınan listenin indisi bir sonraki öğeye geçer. Eşitlikte `<=` sayesinde birinci listedeki öğe alınır. Birinci liste sol yarıdan geldiği sürece eşit anahtarlar girişteki sıralarını korur; bu sürüm kararlıdır.
- **Satır 12.** Listelerden biri bitince ötekinin kalanı tek bağla eklenir. Dizi sürümündeki gibi kalan öğeleri tek tek kopyalamaya gerek yoktur; kalan kısım zaten sıralı bir listedir.
- **Satır 13.** Birleşmiş listenin başı `a[0].bag`'dedir ve döndürülür.

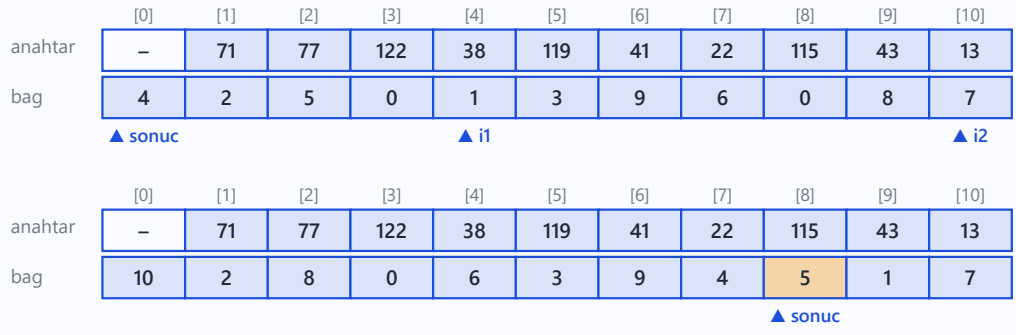
Bir öğe taşınmaz; yalnız bağlar yeniden yazılır. Bir birleştirmenin maliyeti, döngünün tur sayısı kadardır ve bu sayı iki listenin uzunlukları toplamından azdır. Karşılaştırma sayısı dizi sürümünden biraz küçük çıkar, çünkü listelerden biri bitince karşılaştırma da biter.

20.7.3 Kod izleme: son birleştirme

Anahtarlar `a[1..10] = 71, 77, 122, 38, 119, 41, 22, 115, 43, 13` olsun. `mergesort(a, 1, 10)`'un son, dokuzuncu birleştirmesi iki beşer öğelik listeyi birleştirir. Birinci liste (baş 4) *38 → 71 → 77 → 119 → 122*, ikinci liste (baş 10) *13 → 22 → 41 → 43 → 115*'tir.

Tur	Karşılaştırma	Yazılan bağ	`sonuc`	`i1`	`i2`
0	—	—	0	4	10
1	*38 > 13*	`a[0].bag = 10`	10	4	7
2	*38 > 22*	`a[10].bag = 7`	7	4	6
3	*38 ≤ 41*	`a[7].bag = 4`	4	1	6
4	*71 > 41*	`a[4].bag = 6`	6	1	9
5	*71 > 43*	`a[6].bag = 9`	9	1	8
6	*71 ≤ 115*	`a[9].bag = 1`	1	2	8
7	*77 ≤ 115*	`a[1].bag = 2`	2	5	8
8	*119 > 115*	`a[2].bag = 8`	8	5	0
son	`i2 = 0`: ikinci liste bitti	`a[8].bag = 5`	8	5	0

Tablo 20.13: Bağlı `merge`'ün son çağrısı, `merge(a, 4, 10)`.



Şekil 20.9: Son birleştirmenin başı (üstte) ve sonu (altta). Üst satır anahtarları, alt satır `bag` alanlarını gösterir. Öğeler yerinden hiç oynamaz; değişen yalnız bağlardır.

Fonksiyon `a[0].bag = 10`'u döndürür. Bağlar izlenince sıralı liste okunur: *13 → 22 → 38 → 41 → 43 → 71 → 77 → 115 → 119 → 122*. Sekiz turda sekiz karşılaştırma yapılmış, kalan iki öğe (119 ve 122) tek bağla eklenmiştir.

20.7.4 `bag` alanlarının değişimi

Tablo bütün sıralama boyunca dokuz birleştirmeyi verir. Kodda sağ yarı önce sıralandığı için ilk birleştirme en sağdaki *(9, 10)* aralığındadır.

#	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	Dönüş	*L, R*
anahtar	71	77	122	38	119	41	22	115	43	13		
1	0	0	0	0	0	0	0	0	0	**9**	10	9, 10
2	0	0	0	0	0	0	**6**	0	0	9	7	6, 7
3	0	0	0	0	0	**8**	6	0	0	9	7	6, 8
4	0	0	0	0	0	**9**	6	0	**8**	**7**	10	6, 10
5	0	0	0	**5**	0	9	6	0	8	7	4	4, 5
6	**2**	0	0	5	0	9	6	0	8	7	1	1, 2
7	2	**3**	0	5	0	9	6	0	8	7	1	1, 3
8	2	**5**	0	**1**	**3**	9	6	0	8	7	4	1, 5
9	2	**8**	0	**6**	3	9	**4**	**5**	**1**	7	10	1, 10

Tablo 20.14: Bağlı birleştirme sıralamasında her `merge`'den sonra `bag` alanları. Kalın: o birleştirmede değişen bağ.

Her satır, dönüş değerinden başlayıp bağlar izlenerek okunur. 4. satırda dönüş 10'dur: *13 → 22 → 41 → 43 → 115*; sağ yarı *a[6..10]* sıralanmıştır. 8. satırda dönüş 4'tür: *38 → 71 → 77 → 119 → 122*.

Sonunda öğeler dizide hâlâ eski yerlerindedir; sıralı çıktı listeyi gezerek yazılır (`for (p = bas; p; p = a[p].bag)`). Ek bellek öğe başına bir `int`'tir; karşılaştırma sayısı *O(n lg n)*, kayıt taşıma sayısı sıfırdır.

> **Sık yapılan hata**
>
> Bağlı sürümü yazarken üç hata sık görülür. Birincisi, `mergesort`'u alışkanlıkla `void` bildirip sonunda değer döndürmeye çalışmaktır; derleyici bunu hata olarak bildirir. İkincisi, `else` dalında `i2 = a[i2].bag` yerine `i1 = a[i1].bag` yazmaktır. Bu durumda ikinci liste hiç ilerlemez, birinci liste ise yanlış yerde ilerler ve listeler bozulur. Üçüncüsü, dönüşte `a[0].bag` yerine `sonuc`'u döndürmektir; `sonuc` listenin başını değil sonunu gösterir.

20.7.5 Sınav sorusu

Aşağıdaki soru bağlı sürümün bir başka yazımını kullanır. Soruda öğeler `a[1..9]`'dur ve anahtarlarına göre sıralanır. İstenenler şunlardır: (i) özyinelemeli çağrıların yığın çerçevelerini çizmek ve `merge` çağrılmışsa dönüş değerini yazmak; (ii) işletim tablosunu `merge` numaraları, dönüş değerleri ve her adımdaki `bag` alanlarıyla doldurmak.

```
int mergesort(eleman a[], int L, int R) {
    int m;
    if (R <= L) return L;
    m = (R + L) / 2;
    return merge(a, mergesort(a, m + 1, R),
                    mergesort(a, L, m));
}
```

Anahtarlar `[1] ?, [2] ?, [3] 18, [4] -5, [5] 27, [6] 11, [7] ?, [8] 83, [9] 30`'dur. Soru kâğıdındaki soru işaretleri yerine burada 42, 7 ve 55 alınmıştır. `merge`, bağlı birleştirme sıralamasındakinin aynısıdır. Sorunun basılı sürümünde `else` dalında `i1` yazılmıştı; bu bir yazım hatasıdır ve doğrusu `i2`'dir.

Sorunun ilk tuzağı, özyinelemeli çağrıların `merge`'ün argümanları olmasıdır; hangi yarının önce sıralanacağı derleyiciye kalır. Çözümde argümanların soldan sağa değerlendirildiği, yani önce sağ yarının sıralandığı varsayılır. Çağrı sırası şöyledir: *(1, 9) → (6, 9) → (8, 9) → (9, 9) → (8, 8) → (6, 7) → (7, 7) → (6, 6) → (1, 5) → ….* İkinci tuzak, `merge`'e birinci liste olarak sağ yarının geçirilmesidir. Eşitlikte birinci liste önce alındığı için bu yazım kararlı değildir; bu sorudaki anahtarlar farklı olduğundan sonucu etkilemez.

#	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	Dönüş	*L, R, m*
1	0	0	0	0	0	0	0	0	**8**	9	8, 9, 8

#	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	Dönüş	*L, R, m*
2	0	0	0	0	0	**7**	0	0	8	6	6, 7, 6
3	0	0	0	0	0	**9**	**8**	0	**7**	6	6, 9, 7
4	0	0	0	**5**	0	9	8	0	7	4	4, 5, 4
5	0	**1**	0	5	0	9	8	0	7	2	1, 2, 1
6	0	**3**	**1**	5	0	9	8	0	7	2	1, 3, 2
7	0	3	**5**	**2**	**1**	9	8	0	7	4	1, 5, 3
8	**7**	**6**	5	2	**9**	**3**	8	0	**1**	4	1, 9, 5

Tablo 20.15: Sınav sorusunun çözümü (argümanlar soldan sağa). Anahtarlar: 42, 7, 18, −5, 27, 11, 55, 83, 30.

Sonuç listesinin başı 4'tür: *−5 → 7 → 11 → 18 → 27 → 30 → 42 → 55 → 83*. Tablo, değerlendirme sırası açıkça yazılmış bir C programıyla üretilmiştir. Argümanlar sağdan sola değerlendirilseydi ilk birleştirme *(1, 2)*'de olur, ara satırlar değişir; son satır ve sonuç aynı kalır.

20.8 Heap sort

Birleştirme sıralaması her durumda *O(n lg n)*'dir ama *O(n)* ek bellek ister. Hızlı sıralama yerinde çalışır ama en kötü durumda n^2'ye çıkar. **Heap sort** iki iyi özelliği birleştirir: en kötü durumda da *O(n lg n)* sürer ve fazladan bellek kullanmaz. Bunu, öncelikli kuyrukları gerçeklemekte kullanılan heap yapısını sıralanacak dizinin içinde kurarak yapar.

20.8.1 Dizi bir ağaçtır

Dizi `a[1..n]` olarak düşünülür; `a[0]` kullanılmaz. Bu dizi bir **tam ikili ağaç** (*complete binary tree*) gibi okunur: kök `a[1]`'dir, `a[i]`'nin çocukları `a[2i]` ve `a[2i+1]`'dir, ebeveyni `a[i/2]`'dir. Ağacın düğümleri düzey düzey, soldan sağa diziye yerleşir. Bu yüzden hiçbir işaretçi gerekmez; bağlar indis aritmetiğinden çıkar. 1 tabanlı indis bu formülleri sadeleştirir. 0 tabanlı dizide çocuklar *2i + 1* ve *2i + 2* olurdu.

> **Tanım 20.3**
>
> Her düğümün değeri çocuklarının değerlerinden küçük olmayan tam ikili ağaca **maksimum heap** (*max-heap*) denir. Dizi gösteriminde bu koşul her $2 \le i \le n$ için $a[i/2] \ge a[i]$ demektir. Maksimum heap'te en büyük öğe köktedir.

Heap sort iki aşamada çalışır. Önce dizi bir maksimum heap'e çevrilir. Sonra kökteki en büyük öğe dizinin sonuna alınır, heap bir öğe küçülür ve kalan heap düzeltilir. Bu adım *n − 1* kez tekrarlanınca dizi küçükten büyüğe sıralanmış olur.

```c
void heapify(int a[], int kok, int n) {
    int t = a[kok], j;
    for (j = 2 * kok; j <= n; j *= 2) {
        if (j < n && a[j] < a[j + 1]) j++; // büyük
        if (t >= a[j]) break;   // t buraya sığar
        a[j / 2] = a[j];        // çocuğu yukarı al
    }
    a[j / 2] = t;
}
void heapsort(int a[], int n) {     /* a[1..n] */
    int i;
    for (i = n / 2; i >= 1; i--)     /* heap kur */
        heapify(a, i, n);
    for (i = n - 1; i >= 1; i--) {   /* sırala */
        SWAP(a[1], a[i + 1]);    /* en büyüğü sona */
        heapify(a, 1, i);            /* heap'i düzelt */
    }
}
```

20.8.2 `heapify`

`heapify(a, kok, n)`, `a[1..n]` heap'inde `kok` düğümünün sol ve sağ alt ağaçları zaten maksimum heap iken kökteki değere uygun yeri arar. Bu, öncelikli kuyruklarda en büyüğü alırken yapılan **aşağı kaydırma** işleminin aynısıdır; Horowitz'in kitabında `adjust` adıyla geçer. Satır satır:

- **Satır 2.** Kökteki değer `t`'ye alınır. Kökte artık bir "boşluk" vardır. `t` en sonda bu boşluğun indiği yere yazılacaktır.
- **Satır 3.** `j`, boşluğun sol çocuğudur: `2 * kok`. Döngü her turda bir düzey aşağı iner (`j *= 2`) ve `j` heap'in dışına çıkınca biter.
- **Satır 4.** Sağ çocuk varsa (`j < n`) ve sol çocuktan büyükse `j` sağ çocuğa geçer. `j` artık büyük çocuğu gösterir.
- **Satır 5.** `t` büyük çocuktan küçük değilse boşluk tam yerindedir; döngüden çıkılır.
- **Satır 6.** Aksi hâlde büyük çocuk boşluğa, yani ebeveynine `a[j/2]`'ye çıkarılır. Boşluk `j`'ye iner.
- **Satır 8.** Döngü nerede biterse bitsin boşluk `a[j/2]`'dedir ve `t` oraya yazılır. `break` ile çıkıldığında `j/2` boşluğun kendisidir. Döngü `j > n` ile bittiğinde de `j` son boşluğun çocuğu olurdu; `j/2` yine boşluğu verir.

Takas yerine kaydırma yapılır: düzey başına bir atama, sonda `t` için bir atama. `heapify`'ın maliyeti boşluğun indiği düzey sayısıyla orantılıdır ve *lg n*'yi aşamaz.

20.8.3 Heap kurma

Diziyi heap'e çevirmenin ilk akla gelen yolu, öğeleri boş bir heap'e tek tek eklemektir. Kruskal algoritması gibi uygulamalarda öğeler zaten tek tek geldiği için heap böyle kurulur. Her ekleme *O(lg n)* sürdüğünden toplam *O(n lg n)*'dir. Öğeler baştan bir dizide hazırsa daha iyi bir yol vardır: **aşağıdan yukarıya heap oluşturma** (*bottom-up heap construction*).

Yapraklar zaten tek öğelik heap'lerdir. $i > \lfloor n/2 \rfloor$ olan düğümlerin çocuğu yoktur, çünkü sol çocukları $2i > n$ konumuna düşerdi. Bu yüzden döngü $\lfloor n/2 \rfloor$'den başlar ve köke doğru geriye gider: $\lfloor n/2 \rfloor, \lfloor n/2 \rfloor - 1, \ldots, 1$. Her i için düğümün alt ağaçları daha önce düzeltilmiştir; `heapify`'ın ön koşulu sağlanır. Döngü köke ulaştığında bütün dizi bir maksimum heap'tir.

Horowitz'in örneği *[26, 5, 77, 1, 61, 11, 59, 15, 48, 19]* dizisidir; $n = 10$, $\lfloor n/2 \rfloor = 5$'tir.

Çağrı	`t`	Olay	Kaydırma	`a[1..10]` (sonra)
`heapify(a, 5, 10)`	61	tek çocuk 19; *61 ≥ 19*	0	`26 5 77 1 61 11 59 15 48 19`
`heapify(a, 4, 10)`	1	çocuklar 15, 48; 48 yukarı, 1 `a[9]`'a	1	`26 5 77 48 61 11 59 15 1 19`
`heapify(a, 3, 10)`	77	çocuklar 11, 59; *77 ≥ 59*	0	aynı
`heapify(a, 2, 10)`	5	61 yukarı; çocuk 19 yukarı; 5 `a[10]`'a	2	`26 61 77 48 19 11 59 15 1 5`
`heapify(a, 1, 10)`	26	77 yukarı; 59 yukarı; 26 `a[7]`'ye	2	`77 61 59 48 19 11 26 15 1 5`

Tablo 20.16: Heap kurma aşaması. Kaydırma, bir çocuğun ebeveynine çıkarılmasıdır (`a[j / 2] = a[j]`).

Kurma aşaması toplam 5 kaydırma yapmıştır. Sonuç geçerli bir maksimum heap'tir: kök 77'dir ve her düğüm çocuklarından büyüktür. Örneğin `a[2] = 61`'in çocukları `a[4] = 48` ve `a[5] = 19`'dur.

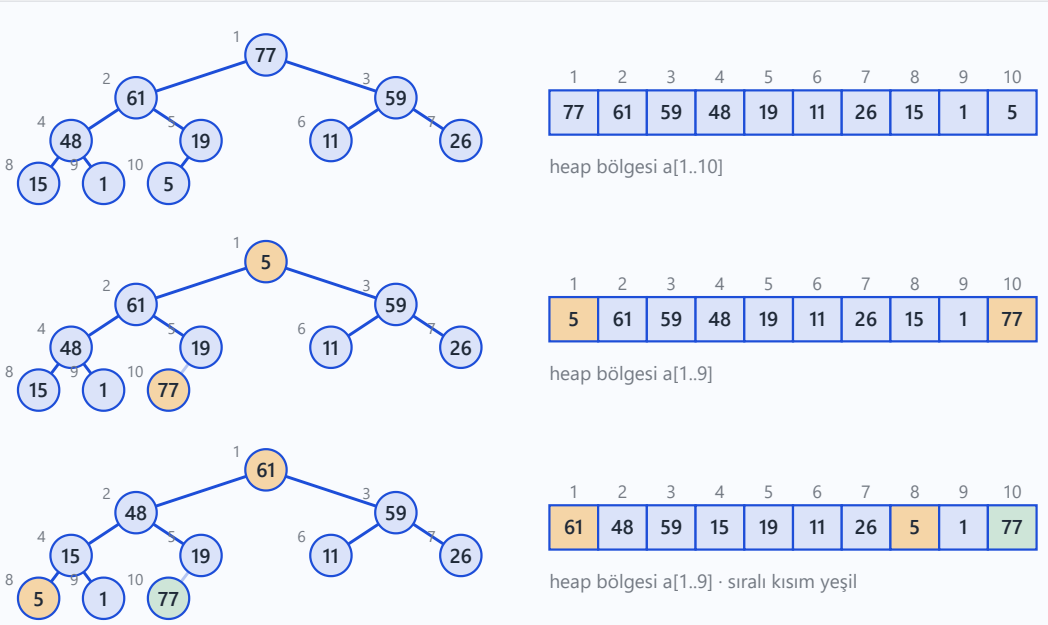
20.8.4 Sıralama aşaması

Dizi heap olduktan sonra en büyük öğe köktedir. Bu öğe dizinin son hücresiyle takas edilir: `SWAP(a[1], a[i + 1])`. En büyük öğe artık kesin yerindedir. Heap bir öğe küçülür ve `a[1..i]` olur. Köke gelen öğe heap koşulunu büyük olasılıkla bozar, ama iki alt ağaç hâlâ heap'tir; `heapify(a, 1, i)` bunu düzeltir. Dizinin sonunda sıralı öğelerin biriktiği bölge her turda bir hücre büyür.

`i`	Sona giden	Köke gelen, indiği yer	`a[1..10]` (sonra)
9	77	5 → `a[8]`	`61 48 59 15 19 11 26 5 1 | 77`
8	61	1 → `a[7]`	`59 48 26 15 19 11 1 5 | 61 77`
7	59	5 → `a[5]`	`48 19 26 15 5 11 1 | 59 61 77`
6	48	1 → `a[6]`	`26 19 11 15 5 1 | 48 59 61 77`
5	26	1 → `a[4]`	`19 15 11 1 5 | 26 48 59 61 77`
4	19	5 → `a[2]`	`15 5 11 1 | 19 26 48 59 61 77`

`i`	Sona giden	Köke gelen, indiği yer	`a[1..10]` (sonra)
3	15	1 → `a[3]`	`11 5 1 | 15 19 26 48 59 61 77`
2	11	1 → `a[2]`	`5 1 | 11 15 19 26 48 59 61 77`
1	5	1 yerinde kalır	`1 | 5 11 15 19 26 48 59 61 77`

Tablo 20.17: Sıralama aşaması. Her satır bir `SWAP(a[1], a[i + 1])` ve ardından `heapify(a, 1, i)` 'dir; dikey çizginin sağı sıralı bölgedir.



Şekil 20.10: Sıralama aşamasının ilk turu, yukarıdan aşağıya: kurulmuş heap; `SWAP(a[1], a[10])` ile 77 sona gitmiş, kökte 5 heap'i bozuyor; `heapify(a, 1, 9)` sonrası yeni kök 61. Her şekilde solda ağaç, sağda aynı dizi görülür. Turuncu öğeler o adımda yer değiştirenler, yeşil öğeler sıralı bölgedir.

Son turda $i = 1$'dir. 5 sona gider ve kökte tek öğe, 1 kalır. Tek öğelik bir ağaç zaten heap'tir. `heapify(a, 1, 1)` 'de $j = 2 > 1$ olduğu için döngüye hiç girilmez ve 1 yerinde kalır. Dizi artık sıralıdır: *[1, 5, 11, 15, 19, 26, 48, 59, 61, 77]*. Kod ayrıca 1–200 öğelik, tekrarlı anahtarlar içeren 2000 rastgele diziyle sınanmış ve hepsini doğru sıralamıştır.

"Heap'i anladık, ama dizi ne zaman sıralanacak?" sorusunun cevabı, heap'in bir ara düzen olmasıdır: yalnız en büyük öğenin yerini bilir. Sıralama bu bilgiyi $n - 1$ kez kullanır.

Sık yapılan hata

1 tabanlı indisleri 0 tabanlı bir diziyle karıştırmak. Çocukların $2i$ ve $2i + 1$ olduğu formüller yalnız `a[1..n]` 'de doğrudur. C dizisi `int a[10]` olarak bildirilip `heapsort(a, 10)` çağrılırsa yordam `a[10]` 'a, yani dizinin dışına erişir ve `a[0]` 'daki öğe hiç sıralanmaz. Doğrusu diziyi `n + 1` öğelik bildirmek ya da formülleri 0 tabanlı biçimiyle yeniden yazmaktır.

20.8.5 Analiz

Kaba analizle `heapify` en çok $\lg n$ kaydırma yapar ve $n/2$ kez çağrılır: $O(n \lg n)$. Bu sınır gevşektir, çünkü çağrıların çoğu küçük alt ağaçlarda yapılır.

İnce analiz için ağaç düzey düzey incelenir. k, ağacın düzey sayısı olsun; $k \approx \lg(n + 1)$'dir. i. düzeyde en çok 2^{i-1} düğüm vardır. Bu düzeydeki bir düğüm en çok $k - i$ düzey aşağı kayabilir. En kötü durumda toplam kaydırma sayısı

$$\sum_{i=1}^{k} 2^{i-1}(k - i) = k \sum_{i=1}^{k} 2^{i-1} - \sum_{i=1}^{k} i \cdot 2^{i-1}$$

olur. İlk toplam geometrik seridir: $2^k - 1$. İkinci toplam, geometrik serinin türevinden bulunur. $S(y) = 1 + y + \dots + y^k = (y^{k+1} - 1)/(y - 1)$ olsun. Terim terim türev alınınca $S'(y) = \Sigma_{i=1}^{k} i \cdot y^{i-1}$ elde edilir; aranan toplam $S'(2)$'dir. Kesrin türevi alınıp $y = 2$ konursa $S'(2) = (k + 1)2^k - (2^{k+1} - 1) = (k - 1)2^k + 1$ bulunur. Böylece

$$k(2^k - 1) - (k - 1)2^k - 1 = 2^k - k - 1$$

Teorem 20.4 (Heap sort'un maliyeti)

k düzeyli bir ağaçta aşağıdan yukarıya heap kurma en çok $2^k - k - 1 \le 2n$ kaydırma yapar; heap kurma $O(n)$'dir. Sıralama aşaması $n - 1$ `heapify` çağrısıyla $O(n \lg n)$ sürer. Heap sort en kötü durumda da $O(n \lg n)$'dir, ek belleği $O(1)$'dir ve kararlı değildir.

İspat. Kurma sınırı yukarıdaki hesaptır. k düzeyli bir tam ikili ağaçta en az 2^{k-1} düğüm vardır; $n \ge 2^{k-1}$ ve $2^k \le 2n$ olur. Sıralama aşamasında `heapify(a, 1, i)` en çok $\lfloor \lg i \rfloor$ kaydırma yapar; $\Sigma_{i=1}^{n-1} \lfloor \lg i \rfloor \le n \lg n$'dir. Yordam yalnız birkaç değişken kullanır ve özyinelemesizdir. Kararsızlık için bir örnek yeter: $[2a, 2b, 1]$ dizisi $[1, 2b, 2a]$ olarak sıralanır.

Sınırın sıkı olduğu, en kötü durumu üreten girdiyle gösterilebilir. Artan sıralı bir dizide her iç düğümün değeri bütün alt ağacından küçüktür ve her `heapify` boşluğu yaprağa kadar indirir. $n = 2^k - 1$ öğelik artan dizilerde sayılan kaydırma sayısı formülle tam olarak örtüşmüştür (Tablo).

k	$\Sigma\, 2^{i-1}(k - i)$	$2^k - k - 1$ (ölçülen)	$n = 2^k - 1$
2	1	1	3
3	4	4	7
4	11	11	15
6	57	57	63
10	1013	1013	1023
20	1.048.555	1.048.555	1.048.575

Tablo 20.18: Heap kurmanın en kötü durumu. Ortadaki iki sütun aynıdır; son sütun düğüm sayısıdır. Kaydırma sayısı hep n'den küçüktür.

"$\le 2n$" sınırı $O(n)$ demektir, $o(n)$ değil: kaydırma sayısı n ile aynı hızda büyür.

Karşılaştırma sayısı da ölçülmüştür. Her kaydırma düzeyinde iki karşılaştırma vardır: iki çocuk arasında ve `t` ile büyük çocuk arasında. Rastgele dizilerde toplam karşılaştırma 1000 öğe için 16.899, 50.000 öğe için 1,41 milyon, bir milyon öğe için 36,8 milyon çıkmıştır. Bunlar $2n \lg n$'nin yaklaşık %85–92'sidir. Kurma aşamasının payı küçüktür: bir milyon öğede 1,88 milyon karşılaştırma, yani $2n$'den az.

20.8.6 İleri heap sort ve veriyi tanımak

Yaklaşık $2n \lg n$ karşılaştırma, hızlı sıralamanın $1{,}39\, n \lg n$'sinden fazladır ve heap sort belleğe düzensiz erişir. Elli bin civarına kadar veri için makul bir seçimdir; daha büyük veride karşılaştırma sayısını azaltan sürümler vardır.

Bunların en bilineni **aşağıdan yukarı heap sort**'tur (*bottom-up heapsort*). Fikri şudur: kökten çıkarılan öğenin yerine gelen `t` neredeyse her zaman yaprağa kadar iner, çünkü dizinin sonundan gelmiştir ve küçüktür. Öyleyse her düzeyde `t` ile büyük çocuğu karşılaştırmak çoğu zaman boşunadır. Bottom-up sürümde boşluk, `t` ile karşılaştırılmadan büyük çocuklar boyunca yaprağa kadar indirilir. Sonra `t`, yaprakta başlayıp yukarı doğru yerine yerleştirilir. Bu yukarı yürüyüş genellikle bir iki adımda biter. Sonuçta düzey başına iki karşılaştırma yerine çoğunlukla bir karşılaştırma yapılır. Kuramsal olarak bundan da iyi sürümler bilinir.

Sıralanacak verinin bazı özellikleri biliniyorsa algoritma bu özelliklere göre seçilir. Saymalı sıralamada anahtarların küçük bir aralıkta olduğu bilinir ve hiç karşılaştırma yapılmadan doğrusal sürede sıralanır. Shell sıralaması ile kova sıralamasının birlikte kullanıldığı uygulamalar da vardır. Bu yöntemlerin temel birkaç tanesi bilinmeli ve verinin yapısı tanınmalıdır. Basamak sıralaması bu düşüncenin bir başka örneğidir.

20.9 Basamak sıralaması

20.9.1 Çok anahtarlı sıralama

Bazı sıralamalar birden çok anahtara göre yapılır. Bir kişi listesi önce soyadına, soyadı aynı olanlar ada göre sıralanabilir. Anahtarlar K_0, K_1, ..., K_{r-1} ile gösterilsin. K_0 **en anlamlı** anahtardır (*most significant digit*, MSD), K_{r-1} **en az anlamlı** anahtardır (*least significant digit*, LSD). Kişi listesi örneğinde soyadı en anlamlı anahtardır. Üç basamaklı bir sayı da aynı biçimde düşünülebilir: yüzler basamağı K_0, onlar K_1, birler K_2'dir.

MSD yolunda önce en anlamlı anahtara göre gruplara ayrılır ve her grup kendi içinde sıralanır; bu yol özyinelemeli ve karmaşıktır. LSD yolu ilk bakışta tersine görünür ama çok daha sadedir.

> **Tanım 20.4**
>
> **Basamak sıralaması** (*radix sort*), LSD biçiminde, öğeleri önce en az anlamlı basamağa göre, sonra sırasıyla daha anlamlı basamaklara göre, en son da en anlamlı basamağa göre sıralar. Her geçişin **kararlı** olması gerekir: eşit basamaklı öğeler, bir önceki geçişin bıraktığı sırayı korumalıdır.

Kararlılık şartının nedeni şudur. Yüzler geçişinde 208 ile 271'in yüzler basamağı aynıdır, 2'dir. Bu geçiş ikisini ayırt edemez. Doğru sırayı önceki geçişler belirlemiştir: onlar geçişi 208'i (onlar basamağı 0) 271'in (onlar basamağı 7) önüne koymuştur. Yüzler geçişi bu düzeni bozmazsa sonuç doğru olur. Her geçiş bir öncekinin işini korur ve üstüne daha anlamlı bir basamağı ekler.

Basamak sıralaması sayıları birbiriyle hiç karşılaştırmaz. Bu yüzden **karşılaştırmasız** (*non-comparison*) bir sıralamadır ve karşılaştırmalı sıralamaların $n \lg n$ alt sınırına bağlı değildir. Bunun bedeli, anahtarların yapısının, yani basamakların bilinmesi gerekliliğidir.

20.9.2 Veri yapısı ve liste kurma

```c
#define MAX_DIGIT 3
#define RADIX 10
typedef struct oge *oge_gost;
struct oge {
    int deger[MAX_DIGIT];  /* 179 → {1, 7, 9} */
    oge_gost bag;
};
```

Her öğe bir bağlı liste düğümüdür. `deger` dizisi sayının basamaklarını tutar: 179 için *{1, 7, 9}*. `deger[0]` yüzler (MSD), `deger[1]` onlar, `deger[2]` birler (LSD) basamağıdır. Sayılar *[0, 999]* aralığındadır. `bag`, listede sonraki düğümün adresidir.

`typedef struct oge *oge_gost;` yapı tanımından önce gelir ve `struct oge`'yi gösteren işaretçi türüne ad verir. İşaretçinin boyu yapının içeriğinden bağımsız olduğu için yapı henüz tanımlanmamış olabilir; böylece yapının içinde `oge_gost bag;` yazılabilir. `typedef struct oge {…};` gibi ad verilmeyen bir `typedef` ise anlamsızdır. x86-64'te `sizeof(struct oge)` 24 bayttır: 12 baytlık dizi, 4 bayt hizalama boşluğu ve 8 baytlık işaretçi.

Sıralanacak listeyi kurmak da bellek yönetimi ister. Aşağıdaki iki yordam bu nota eklenmiştir. `liste_kur`, bir `int` dizisinden aynı sırada bir liste kurar:

```
/* 3 basamaklı sayılardan bağlı liste kurar; bellek yetmezse kurulanı geri verir */
oge_gost liste_kur(const int x[], int n) {
    oge_gost bas = NULL, son = NULL;
    for (int k = 0; k < n; k++) {
        oge_gost d = malloc(sizeof *d);
        if (d == NULL) {              /* yarıda kaldı: sızıntı olmasın */
            liste_sil(&bas);
            return NULL;
        }
        d->deger[0] = x[k] / 100;      /* yüzler */
        d->deger[1] = x[k] / 10 % 10;  /* onlar */
        d->deger[2] = x[k] % 10;       /* birler */
        d->bag = NULL;
        if (son) son->bag = d; else bas = d;
        son = d;
    }
    return bas;
}
```

```
/* listenin bütün düğümlerini geri verir ve çağıranın işaretçisini NULL yapar */
void liste_sil(oge_gost *pp) {
    while (*pp) {
        oge_gost sonraki = (*pp)->bag;
        free(*pp);
        *pp = sonraki;
    }
}
```

`liste_kur` her sayı için bir düğüm ayırır. `malloc` `NULL` döndürürse bellek bitmiştir. Bu durumda yarıda kalan liste geri verilmezse o ana kadar ayrılmış düğümler sızar. Yordam bu yüzden önce kurduğu kısmı `liste_sil` ile siler, sonra `NULL` döndürür. Basamaklar tam sayı bölmesi ve kalanla ayrılır: *179 / 100 = 1, 179 / 10 % 10 = 7, 179 % 10 = 9*. `son` işaretçisi listenin son düğümünü tutar; yeni düğüm onun arkasına bağlanır ve liste girdi sırasını korur.

`liste_sil`, parametresini `oge_gost *pp`, yani **işaretçinin işaretçisi** olarak alır. Bu sayede düğümleri geri vermekle kalmaz, çağıranın baş işaretçisini de `NULL` yapar. Döngü her turda sıradaki düğümün adresini önce `sonraki`'ye alır, sonra düğümü geri verir. Sıra ters olsaydı, yani önce `free(*pp)` sonra `(*pp)->bag` okunsaydı, geri verilmiş belleğe erişilirdi. Bu bir tanımsız davranıştır. İki yordam sayaçlı bir `malloc` / `free` çiftiyle sınanmıştır: 500 rastgele liste kurulup sıralanıp silinmiş, 74.115 ayırmaya 74.115 geri verme karşılık gelmiştir. Bellek dördüncü düğümde bitirilen bir denemede ilk üç düğüm geri verilmiş ve `NULL` dönmüştür.

20.9.3 `radixsort`

Sunumdaki döngü gövdesi, kova dizilerinin bildirimiyle birlikte bir fonksiyon olarak yazılırsa şu biçimi alır:

```
oge_gost radixsort(oge_gost p) {
    oge_gost on[RADIX], arka[RADIX];  /* kova başları ve sonları */
    for (int i = MAX_DIGIT - 1; i >= 0; i--) {
        for (int j = 0; j < RADIX; j++)
            on[j] = arka[j] = NULL;
        for (; p; p = p->bag) {     // dağıt
            int b = p->deger[i];
            if (!on[b]) on[b] = p;
            else arka[b]->bag = p;
            arka[b] = p;
        }
        for (int j = RADIX - 1; j >= 0; j--) // topla
            if (on[j]) { arka[j]->bag = p; p = on[j]; }
    }
    return p;
}
```

Her basamak değeri için bağlı liste biçiminde bir **kova** (*bucket*) vardır. `on[j]` kovanın ilk, `arka[j]` son düğümünü gösterir; kuyruktaki gibi, son düğümü tutmak arkaya sabit sürede eklemeyi sağlar.

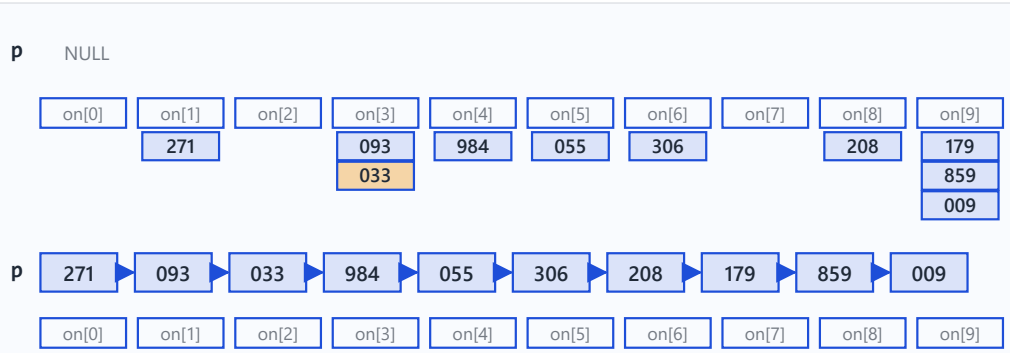
- **Dış döngü.** `i`, `MAX_DIGIT - 1`'den 0'a iner: önce birler (*i = 2*), sonra onlar, en son yüzler basamağı işlenir. Bu LSD sırasıdır.
- **Kovaları boşaltma.** Her geçişin başında on kovanın ön ve arka işaretçileri `NULL` yapılır.
- **Dağıtma.** Liste baştan sona gezilir. Her düğümün *i*. basamağı `b`'ye alınır. Kova boşsa düğüm kovanın ilk düğümü olur. Değilse kovanın son düğümünün `bag`'i bu düğüme bağlanır. Her iki durumda da düğüm kovanın yeni sonudur. Düğümler kovaya geliş sırasıyla eklendiği için dağıtma kararlıdır. Döngü `p = p->bag` ile ilerler. Bu okumada düğümün eski `bag`'i kullanılır; düğümün kendi bağı daha değiştirilmemiştir, yalnız kovadaki önceki düğümün bağı değişmiştir. Döngü bittiğinde `p` `NULL`'dır.
- **Toplama.** Kovalar 9'dan 0'a gezilir. Dolu her kovanın son düğümü, o ana kadar kurulmuş listenin başına bağlanır (`arka[j]->bag = p`) ve listenin başı kovanın ilk düğümü olur (`p = on[j]`). Liste sondan başa doğru kurulur; en son eklenen 0. kova en önde durur. Böylece küçük basamaklı kova öne gelir. İlk eklenen kovanın son düğümünün `bag`'i, o anda `NULL` olan `p`'ye bağlanır; listenin sonu doğru biçimde kapanır.
- **Dönüş.** Üç geçişten sonra liste sıralıdır ve baş işaretçisi döndürülür.

20.9.4 Kod izleme

Liste *179, 208, 306, 093, 859, 984, 055, 009, 271, 033* olsun. Sayılar üç basamaklı yazılmıştır, çünkü yapı her sayının üç basamağını tutar. İlk geçiş birler basamağına göredir. Tablo dağıtımı adım adım verir.

Öğe	`deger[2]`	Kova durumu (sonra)
179	9	9: 179
208	8	8: 208 · 9: 179
306	6	6: 306 · 8: 208 · 9: 179
093	3	3: 093 · 6: 306 · 8: 208 · 9: 179
859	9	… · 9: 179 859
984	4	… · 4: 984 · …
055	5	… · 5: 055 · …
009	9	… · 9: 179 859 009
271	1	1: 271 · …
033	3	1: 271 · 3: 093 033 · 4: 984 · 5: 055 · 6: 306 · 8: 208 · 9: 179 859 009

Tablo 20.19: Birler geçişinde (*i = 2*) dağıtım. Her öğe kendi kovasının arkasına eklenir.



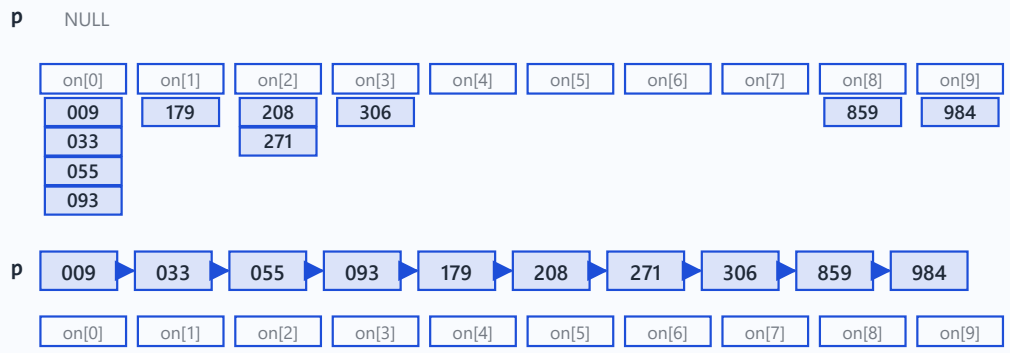
Şekil 20.11: Birler geçişi. Üstte dağıtım bitmiştir: her kova, öğeleri geliş sırasıyla taşır. Altta toplama sonrası liste: *271 093 033 984 055 306 208 179 859 009*.

9. kovada 179, 859 ve 009 bu sırayla durur; üçü de girdide bu sırada gelmiştir. Toplama kovaları 9'dan 0'a gezer ve her kovayı listenin önüne bağlar. Sonuçta küçük kova öndedir: *271 093 033 984 055 306 208 179 859 009*. Liste artık birler basamağına göre sıralıdır.

İkinci ve üçüncü geçişler aynı biçimde onlar ve yüzler basamaklarına göre yapılır. Tablo her geçişten sonraki listeyi verir.

Geçiş	Liste
başlangıç	`179 208 306 093 859 984 055 009 271 033`
i = 2 (birler)	`271 093 033 984 055 306 208 179 859 009`
i = 1 (onlar)	`306 208 009 033 055 859 271 179 984 093`
i = 0 (yüzler)	`009 033 055 093 179 208 271 306 859 984`

Tablo 20.20: Her geçişten sonra liste.



Şekil 20.12: Yüzler geçişi. Üstte dağıtım: 0. kovada dört öğe (009, 033, 055, 093), onlar geçişinin bıraktığı sırayla durur. Altta toplama sonrası sıralı liste.

Onlar geçişinden sonra 0. kovada 306, 208 ve 009 vardır; onlar basamaklarının hepsi 0'dır ve birler geçişinin bıraktığı sırayla dizilmişlerdir. Yüzler geçişinde 0. kovaya 009, 033, 055 ve 093 düşer. Kararlılık sayesinde bu dört öğe onlar geçişindeki sırasını korur ve kova kendiliğinden sıralı çıkar. Üç geçişin sonunda liste sıralıdır. Yordam ayrıca 1–300 öğelik 500 rastgele listeyle sınanmıştır.

20.9.5 Analiz

Her geçiş üç iş yapar: `RADIX` kovayı boşaltır, *n* düğümü dağıtır ve `RADIX` kovayı toplar. Geçiş sayısı `MAX_DIGIT` 'tir. Toplam süre

$$O(\text{MAX_DIGIT} \cdot (\text{RADIX} + n + \text{RADIX})) = O(\text{MAX_DIGIT} \cdot (\text{RADIX} + n))$$

olur. Basamak sayısı *d*, taban *r* ile gösterilirse $O(d(n + r))$ yazılır. Örnekte bu $O(3 \cdot (10 + n))$'dir ve doğrusal görünür. Ancak bu, saymalı sıralamadaki gibi özel bir durumdur: basamak sayısı ve taban sabit kabul edilmiştir. *n* farklı sayıyı ayırt etmek için en az $\log_r n$ basamak gerekir. Anahtarların aralığı *n* ile birlikte büyürse *d* de büyür ve süre yine $n \lg n$ mertebesine yaklaşır.

Algoritmanın kullanılabilmesi için veriler (a) negatif olmayan, (b) tam sayı ve (c) küçük bir aralıkta olmalıdır. Aralık büyürse algoritma yine sıralar, ama geçiş sayısı ve karmaşıklık hızla artar. Yalnız tam sayıları sıralayabilmesi gerçek bir dezavantajdır. Kayan noktalı sayılar için uyarlanabilir, ama o uyarlama yapılana kadar öteki algoritmalar işi çoktan bitirmiş olur.

Ek bellek $2 \cdot RADIX$ işaretçidir: kovaların ön ve arka işaretçileri. Liste yerinde yeniden bağlanır. Her düğümdeki `bag` işaretçisi de bir bellek maliyetidir, ama bu maliyet bağlı liste gösteriminden gelir, algoritmadan değil. Basamak sıralaması kararlıdır ve karşılaştırma yapmaz.

20.9.6 Sınav sorusu: değiştirilmiş radix

Bir öğrenci, yordamın önce en anlamlı basamağı ele alması gerektiğini düşünür. Dış döngüyü `for (i = 0; i <= MAX_DIGIT-1; i++)`, toplama döngüsünü `for (j = 0; j <= RADIX-1; j++)` biçimine getirir. Soru, *346, 180, 008, 930, 023, 451, 670, 098, 203, 007* listesi için yordamı işletmeyi, her adımda listeyi yazmayı ve sonucu yorumlamayı ister.

Geçiş	Liste
başlangıç	`346 180 008 930 023 451 670 098 203 007`

Geçiş	Liste
i = 0 (yüzler)	`930 670 451 346 203 180 008 023 098 007`
i = 1 (onlar)	`098 180 670 451 346 930 023 203 008 007`
i = 2 (birler)	`098 008 007 346 023 203 451 180 670 930`

Tablo 20.21: Değiştirilmiş yordamın geçişleri.

İki değişikliğin etkisi ayrı ayrı yorumlanır. Birinci değişiklik geçişlerin sırasını tersine çevirir. Son geçiş birler basamağına göre yapıldığı için birler basamağı en anlamlı basamak gibi davranır. Sayılar tersten okunursa sıralıdır: 098, 008, 007 tersten 890, 800, 700'dür. İkinci değişiklik toplama döngüsünü tersine çevirir. Kovalar 0'dan 9'a gezilip her biri listenin önüne bağlandığı için en son eklenen 9. kova öne geçer. Sıra küçükten büyüğe değil, büyükten küçüğe olur. Sonuç olarak liste, tersten okunan sayılara göre azalan sıradadır: 890, 800, 700, 643, 320, 302, 154, 081, 076, 039. Tablo, değiştirilmiş döngülerle yazılmış bir C programıyla üretilmiştir.

MSD ile sıralamak için en anlamlı basamağa göre ayrılan her grubun ayrıca sıralanması gerekir; grupları bir arada tutup sonraki basamağa geçmek önceki geçişin işini bozar.

20.10 Karşılaştırma ve seçim

Tablo temel ve ileri sıralama algoritmalarını karşılaştırır; zaman sütunları adım sayısının mertebesidir.

Algoritma	En iyi	Ortalama	En kötü	Ek bellek	Kararlı	Yerinde
Seçimli	n^2	n^2	n^2	*O(1)*	hayır	evet
Eklemeli	n	n^2	n^2	*O(1)*	evet	evet
Kabarcık (erken duran)	n	n^2	n^2	*O(1)*	evet	evet
Shell (Knuth aralıkları)	$n \lg n$	—	$n^{3/2}$	*O(1)*	hayır	evet
Hızlı	$n \lg n$	$1{,}39\, n \lg n$	n^2	*O(lg n)* yığın*	hayır	evet
Birleştirme	$n \lg n$	$n \lg n$	$n \lg n$	*O(n)*	evet**	hayır
Heap sort	$n \lg n$	$n \lg n$	$n \lg n$	*O(1)*	hayır	evet
Saymalı	$n + k$	$n + k$	$n + k$	*O(n + k)*	evet	hayır
Basamak (*d* basamak, taban *r*)	$d(n + r)$	$d(n + r)$	$d(n + r)$	*O(r)****	evet	—

Tablo 20.22: Sıralama algoritmalarının karşılaştırması. * Küçük parça önce işlenirse; aksi hâlde en kötü *O(n)*. ** Klasik birleştirmeyle; bu nottaki bitonik `merge` kararlı değildir. *** Bağlı liste üzerinde.

Karşılaştırmalı sıralamada alt sınır $\lceil \lg n! \rceil \approx n \lg n - 1{,}44\, n$'dir. Tablodaki karşılaştırmalı algoritmalardan hızlı sıralamanın ortalaması, birleştirme sıralaması ve heap sort bu sınıra sabit bir çarpan içinde yaklaşır. Saymalı ve basamak sıralaması karşılaştırma yapmadığı için bu sınıra bağlı değildir; buna karşılık anahtarlar hakkında ek bilgiye ihtiyaç duyar.

Kuramsal mertebeler aynı olsa da gerçek süreler farklıdır. Tablo bu nottaki yordamların rastgele `int` dizilerindeki sürelerini verir.

n	`quicksort1`	`sirala`	`quicksort2`	`qsort`	`mergesort`	`heapsort`
100.000	0,008	0,008	0,008	0,017	0,006	0,010
1.000.000	0,130	0,101	0,130	0,211	0,076	0,180

Tablo 20.23: Rastgele `int` dizilerinde süre, saniye (`zig cc -O2`, x86-64; üç ölçümün ortalaması). Süreler makineye göre değişir; yalnız oranlar anlamlıdır.

Bu ölçümde birleştirme sıralaması en hızlısı çıkmıştır. Bitonik `merge`'ün iç döngüsünde sınır denetimi yoktur ve bellek hep sırayla okunup yazılır; bu düzen modern işlemcilerin önbelleğine çok uygundur. Bedeli, bir milyon öğe için 4 MB'lık ek dizidir. Heap sort en yavaşlardandır, çünkü kökten yaprağa inen yollar bellekte büyük sıçramalar yapar. `qsort`'un yavaşlığı her karşılaştırmadaki fonksiyon çağrısından gelir. Medyan-3 ve eşikli `sirala`, `quicksort1`'den %20 kadar hızlıdır. Bu sonuçlar derleyiciye, işlemciye ve veriye göre değişebilir.

Hangi durumda hangi algoritmanın seçileceği şöyle özetlenebilir:

- **Genel amaçlı, bellekte sıralama.** Medyan-3 ve küçük parçada eklemeli kullanan hızlı sıralama ya da kitaplıktaki `qsort`.
- **En kötü durum güvencesi.** Yerinde çalışan heap sort ya da introsort.
- **Kararlılık, bağlı liste ya da disk.** Birleştirme sıralaması: kararlıdır, sıralı parçaları birleştirir ve dış sıralamanın temelidir.
- **Büyük kayıtlar.** Kayıtların kendileri değil, indisleri ya da bağları sıralanır; bağlı birleştirme sıralaması bunun örneğidir.
- **Sabit uzunluklu tam sayı anahtarlar.** Basamak sıralaması: d geçişin her biri $O(n + r)$.
- **Küçük aralıktaki anahtarlar.** Saymalı sıralama.

20.11 Sık yapılan hatalar

1. `bol`'daki `j` döngüsünde `j == L` sınamasını unutmak: pivot en küçükse dizinin başından taşılır.
2. Sıralı girdide pivotu hep `a[R]` seçmek: derinlik n, süre n^2 olur. Çare medyan-3 ya da rastgele pivottur.
3. Özyinelemesiz sürümde parçaları boyuna bakmadan itmek: yığın $O(n)$'e büyür ve sabit boylu dizi taşar.
4. Medyan-3'te `SWAP` argümanlarını ters vermek: büyük öğe `a[L]`'ye gider ve bölümleme bozulur.
5. `b[]`'yi her çağrıda `malloc`'lamak (yavaş) ya da hiç ayırmamak (tanımsız davranış); `NULL` denetimini ve `free`'yi unutmak.
6. Bağlı `merge`'ün `else` dalında `i2` yerine `i1` yazmak; argümanların değerlendirilme sırasına güvenmek.
7. `heapify`'da 1 tabanlı formülleri 0 tabanlı diziyle kullanmak.
8. `qsort` karşılaştırıcısında `return x - y;` yazmak ya da tutarsız bir karşılaştırıcı kullanmak.
9. Basamak sıralamasında düğümü kovanın önüne eklemek: dağıtım kararsızlaşır ve sonuç yanlış çıkar.

20.12 Alıştırmalar ve çözümleri

Cevaplar sunumdakilerle aynıdır; burada çözüm adımları da verilmiş ve kodlar çalıştırılarak doğrulanmıştır.

1. *[5, 9, 1, 7, 3]* için `bol(a, 0, 4)` sonucunda dizi ve dönüş değeri nedir?

Çözüm. Pivot $v = a[4] = 3$'tür. `i`, `a[0] = 5`'te durur ($5 \geq 3$). `j`, 7'yi geçer ve `a[2] = 1`'de durur ($1 \leq 3$). $i = 0 < j = 2$: takas, dizi *[1, 9, 5, 7, 3]*. Sonra `i`, `a[1] = 9`'da; `j`, 9'u geçip `a[0] = 1`'de durur. $i = 1 \geq j = 0$: çaprazlandı. Pivot `a[1]` ile takas edilir. Dizi *[1, 3, 5, 7, 9]*, dönüş değeri 1'dir.

2. Hızlı sıralama (`quicksort1`) 8 öğelik sıralı bir dizide kaç `bol` çağrısı yapar? Özyineleme derinliği nedir?

Çözüm. Pivot her seferinde aralığın en büyüğüdür ve yerinde kalır. Boyları 8, 7, …, 2 olan aralıklar bölümlenir: 7 `bol` çağrısı. Her bölümleme bir öncekinin içinde yapıldığı için iç içe bekleyen ve `bol` çağıran çerçeve sayısı $n - 1 = 7$'dir. Karşılaştırma sayısı $9 + 8 + \ldots + 3 = 42$'dir.

3. Birleştirme sıralaması 16 öğe için kaç `merge` çağrısı yapar? En çok kaç karşılaştırma yapılır?

Çözüm. Çerçeve ağacının 16 yaprağı ve $16 - 1 = 15$ iç düğümü vardır: 15 `merge` çağrısı. Bu nottaki `merge` her yerleştirmede tam bir karşılaştırma yapar. Ağacın $\lg 16 = 4$ düzeyinin her birinde toplam 16 öğe birleştirilir: $4 \times 16 = 64$ karşılaştırma. Sayı girdiden bağımsızdır; sayaçlı çalıştırma da 64 vermiştir.

4. *[4, 10, 3, 5, 1]* (`a[1..5]`) için heap kurma sonrası dizi nedir?

Çözüm. $[5/2] = 2$'den başlanır. `heapify(a, 2, 5)`: $t = 10$, çocuklar 5 ve 1; $10 \geq 5$, değişiklik yok. `heapify(a, 1, 5)`: $t = 4$, çocuklar 10 ve 3; 10 yukarı çıkar. Boşluk 2'ye iner; çocuklar 5 ve 1, $4 < 5$: 5 yukarı çıkar. Boşluk 4'e iner; çocuğu yoktur ve 4 oraya yazılır. Sonuç *[10, 5, 3, 4, 1]*'dir.

5. Basamak sıralaması *170, 45, 75, 90, 802, 24, 2, 66* listesinde birler geçişinden sonra hangi sırayı verir?

Çözüm. Birler basamakları sırasıyla 0, 5, 5, 0, 2, 4, 2, 6'dır. Kovalar: 0: 170, 090; 2: 802, 002; 4: 024; 5: 045, 075; 6: 066. Kararlı dağıtım ve toplama sonucu *170 090 802 002 024 045 075 066* verir. Sonraki iki geçiş *802 002 024 045 066 170 075 090* ve *002 024 045 066 075 090 170 802* listelerini üretir.

6. Heap sort neden kararlı değildir? Küçük bir örnek verin.

Çözüm. *[2a, 2b, 1]* dizisi zaten bir maksimum heap'tir. İlk turda kökteki 2a sona taşınır: *[1, 2b, 2a]*. `heapify` 2b'yi köke çıkarır, ikinci turda 2b ikinci konuma gider. Sonuç *[1, 2b, 2a]*'dır ve eşit anahtarların sırası değişmiştir. Heap sort öğeleri uzak konumlar arasında takas eder; bu uzak takaslar eşitlerin göreli sırasını korumaz.

20.13 Özet

- **Hızlı sıralama.** `bol` pivotu kesin yerine koyar ve aralığı ikiye ayırır. En iyi $n \lg n$, ortalama $\approx 1{,}39\, n \lg n$, en kötü n^2 karşılaştırma; yerinde, kararsız. Pivot `i`'nin bekçisidir, `j` için `j == L` gerekir.
- **Ortalama durum.** Yineleme bağıntısı çıkarma hilesiyle $C_n/(n+1) = C_{n-1}/n + 2/(n+1)$ biçimine getirilir ve harmonik sayılarla çözülür.
- **İyileştirmeler.** Büyük parça önce itilirse açık yığında en çok $\lfloor \lg n \rfloor + 1$ ikili bekler. Medyan-3 sıralı girdideki n^2'yi kaldırır; küçük parçalar eklemeliye bırakılır. Introsort en kötü durumu heap sort'la $O(n \lg n)$'ye indirir.
- `qsort`, `void *` ve fonksiyon işaretçisiyle her türü sıralar; karşılaştırıcı tutarlı olmalı ve taşmamalıdır. Kararlılık güvencesi yoktur.
- **Birleştirme sıralaması.** Her durumda $\Theta(n \lg n)$; $2n - 1$ çağrı, $n - 1$ birleştirme, $O(n)$ ek bellek. Bağlı sürüm kayıtları taşımaz, yalnız `bag` indislerini değiştirir.
- **Heap sort.** Heap kurma en çok $2^k - k - 1 \le 2n$ kaydırmayla $O(n)$, sıralama $O(n \lg n)$; ek bellek $O(1)$, kararsız.
- **Basamak sıralaması.** Kararlı kovalarla LSD'den MSD'ye $O(d(n + r))$; negatif olmayan tam sayılar ve küçük aralık için, karşılaştırmasız.

20.14 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

böl ve fethet *divide and conquer* · hızlı sıralama *quick sort* · bölümleme *partition* · pivot *pivot* · değişmez *invariant* · bekçi *sentinel* · yineleme bağıntısı *recurrence relation* · alt sınır *lower bound* · yığın çerçevesi *stack frame* · yığın taşması *stack overflow* · üçün ortancası *median-of-three* · eklemeli sıralama *insertion sort* · hibrit sıralama *hybrid sort* · introsort *introsort* · void işaretçi *void pointer* · fonksiyon işaretçisi *function pointer* · kayıt *record* · kararlı *stable (sort)* · yerinde sıralama *in-place sort* · birleştirme sıralaması *merge sort* · heap sort *heap sort* · max-heap *max-heap* · aşağıdan yukarıya heap oluşturma *bottom-up heap construction* · tam ikili ağaç *complete binary tree* · basamak sıralaması *radix sort* · kova *bucket* · saymalı sıralama *counting sort* · anahtar *key*

20.15 Kendini Dene

1. *[2, 8, 7, 1, 3, 5, 6, 4]* dizisinde `bol(a, 0, 7)` çağrılıyor. Dizi ne olur, hangi değer döner?

   Cevap

   Pivot 4. `i` 8'de, `j` 3'te durur: takas, *[2, 3, 7, 1, 8, 5, 6, 4]*. Sonra `i` 7'de, `j` 1'de durur: takas, *[2, 3, 1, 7, 8, 5, 6, 4]*. Ardından `i` 7'de (konum 3), `j` 1'de (konum 2) durur ve çaprazlanır. Pivot 3. konuma gelir: *[2, 3, 1, 4, 8, 5, 6, 7]*, dönüş 3.

2. `quicksort1` ters sıralı *[6, 5, 4, 3, 2, 1]* dizisinde kaç `bol` çağrısı yapar? Bu girdi neden de kötüdür?

   Cevap

   5 çağrı. Her pivot aralığın bir ucuna düşer ve her bölme yalnız bir öğe ayırır; derinlik $n - 1$, iş n^2 mertebesindedir.

3. Ortalama durum bağıntısından C_2 ve C_3'ü hesaplayın.

   Cevap

   $C_2 = 3 + (2/2)(C_0 + C_1) = 3$. $C_3 = 4 + (2/3)(C_0 + C_1 + C_2) = 4 + 2 = 6$. Kapalı biçim de $n = 3$ için $2 \cdot 4 \cdot H_4 - 32/3 = 50/3 - 32/3 = 6$ verir.

4. `quicksort2` 1000 öğelik bir diziyi sıralarken yığında aynı anda en çok kaç *(L, R)* ikilisi bulunabilir? `YMAX = 64` neden yeterlidir?

   Cevap

   $\lfloor \lg 1000 \rfloor + 1 = 10$ ikili, yani 20 `int`. Sınır $n < 2^{31}$ için bile $2 \cdot 31 = 62$ `int`'tir ve 64'ü aşmaz. Bunun koşulu büyük parçanın önce itilmesidir.

5. *a = [2, 9, 4, 5]* iken `merge(a, b, 0, 1, 3)` çağrılıyor. Kopyalamadan sonra `b` nedir? Sonuçta `a` ne olur?

> **Cevap**
>
> Sol yarı olduğu gibi, sağ yarı ters kopyalanır: *b = [2, 9, 5, 4]*. Birleştirme 2, 4, 5, 9 sırasını verir: *a = [2, 4, 5, 9]*.

6. Birleştirme sıralaması 8 öğe için kaç `mergesort` çağrısı, kaç `merge` ve kaç karşılaştırma yapar?

> **Cevap**
>
> *2n − 1 = 15* çağrı, *n − 1 = 7* birleştirme ve *8 · lg 8 = 24* karşılaştırma.

7. `a[1..8] = [3, 1, 4, 1, 5, 9, 2, 6]` dizisine aşağıdan yukarıya heap kurma uygulanıyor. Sonuç nedir?

> **Cevap**
>
> *[9, 6, 4, 1, 5, 3, 2, 1]*. `heapify` 4, 3, 2 ve 1 için çağrılır ve toplam 5 kaydırma yapılır.

8. Basamak sıralaması *512, 037, 208, 099* listesine uygulanıyor. Onlar geçişinden sonra liste nedir?

> **Cevap**
>
> Birler geçişi *512 037 208 099* verir (2, 7, 8, 9). Onlar geçişi 1, 3, 0, 9 basamaklarına göre *208 512 037 099* verir. Yüzler geçişinden sonra liste *037 099 208 512* olur.

9. `int` dizisini `qsort` ile büyükten küçüğe sıralayan, taşmayan bir karşılaştırma fonksiyonu nasıl yazılır?

> **Cevap**
>
> `int azalan(const void *p, const void *q) { int x = *(const int *)p, y = *(const int *)q; return (y > x) - (y < x); }`. `artan`'daki `x` ile `y`'nin yeri değiştirilmiştir. `return y - x;` yazılsaydı büyük farklarda taşma olurdu.

20.16 İleri okuma

- R. Sedgewick, *Algorithms in C*, Kısım 1–4, Bölüm 7–10. Bu nottaki `bol`, `quicksort2`, medyan-3 ve bitonik `merge` kodlarının kaynağıdır; hızlı sıralamanın sürümlerini ölçümlerle karşılaştırır.
- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, Bölüm 7. Bağlı birleştirme sıralamasını, `adjust` ile heap sort'u ve kovalı basamak sıralamasını bu nottaki örneklerle anlatır.
- T. H. Cormen ve diğerleri, *Introduction to Algorithms*, Bölüm 6–8. Heap sort, hızlı sıralamanın olasılıklı analizi ve doğrusal zamanlı sıralamalar için ayrıntılı kanıtlar verir.
- D. Musser, "Introspective Sorting and Selection Algorithms", 1997. Introsort'un tanıtıldığı makaledir; medyan-3'ü n^2'ye zorlayan girdileri de inceler.

20.17 Kaynaklar

1. R. Sedgewick, *Algorithms in C*, Kısım 1–4, Bölüm 7–10 (quicksort, mergesort, priority queues ve heapsort, radix sorting).
2. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, Bölüm 7.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, Bölüm 6–8.
4. D. Musser, "Introspective Sorting and Selection Algorithms", 1997.
5. ISO/IEC 9899 (C standardı), `qsort` tanımı (§7.22.5.2).

EK A

Terim Sözlüğü

Kitapta geçen İngilizce terimler ve her biri için kullanılan tek Türkçe karşılık. Sağdaki sayı, terimin ayrıntılı anlatıldığı bölümdür. Sözlüğün aranabilir hâli ders sayfasındadır.

A

abstract data type (ADT) → **soyut veri tipi** 1
Değerleri ve işlemleri, nasıl gerçeklendiklerinden bağımsız olarak tanımlayan tip.

Ackermann function → **Ackermann fonksiyonu** 16
Çok hızlı büyüyen özyinelemeli fonksiyon; tersi α(n) union-find maliyetini sınırlar.

activity network → **aktivite ağı** 18
Bir projenin işlerini ve aralarındaki öncelik ilişkilerini gösteren yönlü çizge.

activity on edge (AOE) network → **kenarda aktivite ağı (AOE)** 18
İşlerin kenar, olayların düğüm olduğu ağ.

activity on vertex (AOV) network → **düğümde aktivite ağı (AOV)** 18
İşlerin düğüm, öncelik ilişkilerinin kenar olduğu ağ.

address-of operator (&) → **adres işleci** 4
Bir nesnenin adresini veren tekli işleç.

adjacency list → **komşuluk listesi** 15
Her düğüm için komşularını tutan bağlı liste.

adjacency matrix → **komşuluk matrisi** 15
a[i][j] = 1 ise i ile j arasında kenar bulunan n×n matris.

adjacency multilist → **çok bağlı liste** 15
Yönsüz çizgede her kenarın tek düğümle iki uca bağlandığı gösterim.

adjacent → **komşu** 15
Aralarında bir kenar bulunan iki düğüm.

algorithm → **algoritma** 1
Bir problemi sonlu sayıda kesin ve etkin adımda çözen yöntem.

alignment → **hizalama** 2
Derleyicinin alanları belirli adres katlarına yerleştirmesi; yapıların boyunu etkiler.

all-pairs shortest paths → **tüm çiftler arası en kısa yollar** 18
Her düğüm çifti arasındaki en kısa yolu bulma problemi.

amortized analysis → **amortize analiz** 1
Bir işlem dizisinin toplam maliyetini işlem sayısına bölerek işlem başına maliyeti bulma.

ancestor → **ata** 10
Kökten bir düğüme giden yol üzerindeki düğümler.

array → **dizi** 2
Aynı tipte öğelerin bellekte art arda durduğu, indisle erişilen yapı.

arrow operator (->) → **ok işleci** 1
İşaretçi üzerinden yapı alanına erişim: p->alan, (*p).alan ile aynıdır.

associativity → **birleşme yönü** 8
Aynı öncelikteki işleçlerin soldan mı sağdan mı uygulanacağı.

asymptotic notation → **asimptotik gösterim** 1
Fonksiyonların büyük n için büyüme hızını sabitleri atarak karşılaştıran gösterim.

available space list → **kullanılabilir düğüm listesi** 4
Serbest bırakılan düğümleri yeniden kullanmak için tutan liste (*avail*).

average case → **ortalama durum** 1
Girdilerin belirli bir dağılımına göre beklenen süre.

AVL tree → **AVL ağacı** 11
Her düğümde iki alt ağacın yükseklik farkını en çok 1 tutan arama ağacı.

B

backtracking → **geri izleme** 3
Kısmi çözümü adım adım genişletip çıkmaza girince son seçimi geri alan arama yöntemi.

balance factor → **denge faktörü** 11
Sol alt ağaç yüksekliği ile sağ alt ağaç yüksekliğinin farkı.

balanced parentheses → **parantez denetimi** 6
Açılan her parantezin doğru türde ve sırada kapanıp kapanmadığını yığınla denetleme.

balanced tree → **dengeli ağaç** 10
Yüksekliği O(*lg n*) tutulan arama ağacı.

band matrix → **kuşak matris** 2
Sıfırdan farklı öğeleri ana köşegen çevresindeki bir kuşakta toplanan matris.

base case → **taban durum** 1
Özyinelemenin kendini çağırmadan doğrudan sonuç verdiği durum.

Bellman-Ford algorithm → **Bellman-Ford algoritması** 18
Bütün kenarları n − 1 tur gevşeten, negatif ağırlıklara izin veren en kısa yol algoritması.

best case → **en iyi durum** 1
Aynı boyuttaki girdiler arasında en az adım gerektiren girdi için süre.

Big O → **büyük O** 1
Üst sınır: $n \geq n_0$ için $f(n) \leq c \cdot g(n)$ ise $f(n) = O(g(n))$.

Big Omega → **büyük Omega** 1
Alt sınır: $n \geq n_0$ için $f(n) \geq c \cdot g(n)$ ise $f(n) = \Omega(g(n))$.

Big Theta → **büyük Teta** 1
Sıkı sınır: f(n) hem O(g(n)) hem Ω(g(n)) ise $f(n) = \Theta(g(n))$.

binary search → **ikili arama** 3
Sıralı dizide aralığı her adımda yarıya indirerek arama; O(*lg n*).

binary search tree (BST) → **ikili arama ağacı** 11
Her düğümde sol alt ağaçtaki anahtarların küçük, sağdakilerin büyük olduğu ikili ağaç.

binary tree → **ikili ağaç** 9
Her düğümün en çok iki çocuğu (sol ve sağ) olan ağaç.

bipartite graph → **iki parçalı çizge** 15
Düğümleri, her kenar iki grubu birbirine bağlayacak biçimde iki gruba ayrılabilen çizge.

bottom-up → **aşağıdan yukarı** 1
Önce küçük parçaları kurup onları birleştirerek bütüne ulaşan yaklaşım.

bottom-up heap construction (heapify) → **aşağıdan yukarıya heap oluşturma** 13
Diziyi son iç düğümden köke doğru süzerek O(n) sürede heap'e çevirme.

breadth-first search (BFS) → **genişlik öncelikli arama** 7
Düğümlere başlangıçtan uzaklık sırasıyla, kuyruk yardımıyla uğrayan gezinme.

brute force → **kaba kuvvet** 1
Bütün olası çözümleri tek tek deneyen yaklaşım.

B-tree → **B-ağacı** 11
Düğüm başına birçok anahtar tutan, bütün yaprakları aynı düzeyde olan dengeli m yollu arama ağacı.

bubble sort → **kabarcık sıralaması** 19
Komşu öğeleri karşılaştırıp yanlış sıradakileri takas ederek büyükleri sona taşıyan sıralama.

bucket → **kova** 14
Hash tablosunda bir indisin karşılık geldiği bölme.

bucket sort → **kova sıralaması** 19
Öğeleri değer aralıklarına göre kovalara dağıtıp kovaları ayrı ayrı sıralayan yöntem.

buffer → **tampon** 7
Üreten ve tüketen iki süreç arasında veriyi geçici olarak tutan bellek.

C

cache → **önbellek** 6
İşlemcinin sık kullanılan veriyi tuttuğu küçük ve hızlı bellek; dizilerin bağlı yapılardan hızlı olmasının nedeni.

call by reference → **adresle çağrı** 11
Değişkenin adresini geçirerek fonksiyonun onu değiştirmesine izin verme.

call by value → **değerle çağrı** 6
Argümanın kopyasının geçirilmesi; C'de bütün argümanlar böyle geçer.

call stack → **çağrı yığını** 1
Etkin fonksiyon çağrılarının çerçevelerini tutan yığın.

calloc → **calloc** 2
n öğelik bloğu ayırıp sıfırlayan fonksiyon.

capacity → **kapasite** 1
Dinamik dizinin yeniden ayırma gerekmeden tutabileceği öğe sayısı.

child → **çocuk** 3
Bir düğümün hemen altındaki düğüm.

circular linked list → **dairesel bağlı liste** 4
Son düğümün ilk düğümü gösterdiği bağlı liste.

circular queue → **dairesel kuyruk** 7
Dizinin sonuna gelen indisin başa döndüğü kuyruk gerçeklemesi.

collapsing rule → **azaltma kuralı** 16
Küme bulma sırasında yol üzerindeki düğümleri doğrudan köke bağlama.

collision → **çakışma** 14
İki farklı anahtarın aynı kovaya düşmesi.

column-major order → **sütun öncelikli düzen** 2
Çok boyutlu dizinin sütun sütun yerleştirilmesi.

complete binary tree → **tam ikili ağaç** 10
Son düzey dışında dolu, son düzeyi soldan sağa dolan ikili ağaç.

complete graph → **tam çizge** 15
Her düğüm çiftinin bir kenarla bağlı olduğu çizge; n düğümde n(n − 1)/2 kenar.

compressed sparse row (CSR) → **sıkıştırılmış satır gösterimi** 5
Seyrek matrisi değer, sütun ve satır başlangıcı dizileriyle tutan gösterim.

connected → **bağlantılı** 15
Her düğüm çiftinin arasında bir yol bulunan.

connected component → **bağlantılı bileşen** 15
Çizgenin en büyük bağlantılı alt çizgelerinden biri.

const qualifier → **const niteleyicisi** 1
Değişkenin ya da işaretçinin gösterdiği nesnenin değiştirilmeyeceğini bildiren anahtar sözcük.

counting sort → **saymalı sıralama** 5
Her anahtar değerinden kaç tane olduğunu sayıp öğeleri doğrudan yerine koyan sıralama.

critical path → **kritik yol** 18
Başlangıçtan bitişe en uzun yol; projenin en kısa süresini belirler.

cycle → **döngü** 4
Başladığı düğüme geri dönen yol; bağlı listede de çizgede de görülür.

cycle detection → **döngü tespiti** 4
Bağlı bir yapıda kendine dönen bir yol olup olmadığını bulma.

D

dangling pointer → **sarkan işaretçi** 2
free edilmiş ya da ömrü bitmiş bir nesneyi göstermeye devam eden işaretçi.

decision tree → **karar ağacı** 19
Bir karşılaştırma algoritmasının olası bütün karşılaştırma sonuçlarını gösteren ikili ağaç.

decrease key → **anahtar azaltma** 13
Heap'teki bir öğenin anahtarını küçültüp yerini düzeltme; Dijkstra ve Prim'de kullanılır.

degree → **derece** 10
Ağaçta bir düğümün çocuk sayısı; çizgede bir düğüme değen kenar sayısı.

depth → **derinlik** 10
Ağaçtaki en büyük düzey; ders kitabı ağacın yüksekliğini de bu adla anar.

depth-first search (DFS) → **derinlik öncelikli arama** 6
Bir komşuya gidip olabildiğince derine inen, yığın ya da özyinelemeyle yapılan gezinme.

deque (double-ended queue) → **çift uçlu kuyruk** 7
İki ucundan da ekleme ve çıkarma yapılabilen kuyruk.

dequeue → **dequeue (önden çıkarma)** 7
Kuyruğun önündeki öğeyi çıkarma.

dereference (*) → **içerik işleci** 4
İşaretçinin gösterdiği nesneye ulaşan tekli işleç.

descendant → **soy** 10
Bir düğümün altındaki bütün düğümler.

dictionary → **sözlük** 14
Anahtar–değer çiftlerini tutan ve anahtarla arama sunan ADT.

Dijkstra's algorithm → **Dijkstra algoritması** 18
Negatif olmayan ağırlıklarda en yakın düğümü kesinleştirerek en kısa yolları bulan açgözlü algoritma.

direct addressing → **doğrudan adresleme** 14
Anahtarın kendisini dizi indisi olarak kullanma.

directed acyclic graph (DAG) → **yönlü döngüsüz çizge** 18
Hiç yönlü döngü içermeyen yönlü çizge.

directed graph (digraph) → **yönlü çizge** 15
Kenarlarının bir başı ve bir sonu olan çizge.

disjoint sets → **ayrık kümeler** 16
Ortak öğesi olmayan kümeler; her öğe tek bir kümededir.

divide and conquer → **böl ve fethet** 20
Problemi küçük alt problemlere bölüp çözümleri birleştiren yaklaşım.

division method → **bölme yöntemi** 14
h(k) = k mod m; m genellikle asal seçilir.

double hashing → **çift hash** 14
Yoklama adımını ikinci bir hash fonksiyonuyla belirleme.

double rotation → **çift döndürme** 11
Zıt yönlerde art arda yapılan iki tek döndürme.

doubling → **ikiye katlama** 2
Dizi dolduğunda kapasiteyi iki katına çıkarma; sona eklemenin amortize maliyeti O(1) olur.

doubly linked list → **çift yönlü bağlı liste** 4
Her düğümün hem öncekini hem sonrakini gösterdiği bağlı liste.

down-heap → **down-heap (aşağı süzme)** 13
Kökteki öğenin büyük (ya da küçük) çocuğuyla yer değiştirerek aşağı inmesi.

dummy activity → **yapay aktivite** 18
Süresi sıfır olan, yalnız bir öncelik ilişkisini göstermek için eklenen iş.

dynamic array → **dinamik dizi** 1
Dolduğunda realloc ile büyütülen dizi.

dynamic memory allocation → **dinamik bellek ayırma** 2
Çalışma sırasında malloc, calloc ya da realloc ile bellek isteme.

E

edge → **kenar** 15
İki düğümü bağlayan bağlantı.

enqueue → **enqueue (arkaya ekleme)** 7
Kuyruğun arkasına öğe ekleme.

enumeration (enum) → **sayım tipi** 2
Adlandırılmış tam sayı sabitlerinden oluşan C tipi.

exponential → **üstel** 1
2^n gibi, girdiye her öğe eklendiğinde katlanan büyüme.

expression evaluation → **ifade değerlendirme** 6
Aritmetik bir ifadenin değerini işleç önceliğine uyarak hesaplama.

expression tree → **ifade ağacı** 10
İç düğümleri işleç, yaprakları işlenen olan ikili ağaç.

F

fast transpose → **hızlı transpoz** 5
Sütun başına öğe sayılarını önceden sayarak seyrek matrisi O(sütun + terim) sürede transpoz eden yöntem.

Fibonacci heap → **Fibonacci heap** 13
Anahtar azaltmayı amortize O(1) sürede yapan ileri düzey heap.

FIFO (first in, first out) → **ilk giren ilk çıkar** 7
İlk eklenen öğenin ilk çıkarıldığı sıra; kuyruğun kuralı.

find → **küme bulma** 16
Bir öğenin hangi kümede olduğunu, kümenin kökünü döndürerek bulma.

Floyd-Warshall algorithm → **Floyd-Warshall algoritması** 18
Ara düğümleri birer birer ekleyerek tüm çiftler arası en kısa yolları $O(n^3)$ sürede bulan algoritma.

folding → **katlama** 14
Anahtarı parçalara bölüp parçaları toplayarak hash değeri üretme.

forest → **orman** 17
Birbirinden ayrık ağaçlar topluluğu.

free → **free** 1
malloc ile ayrılan bloğu sisteme geri veren fonksiyon.

front → **ön** 7
Kuyrukta bir sonraki çıkarılacak öğenin bulunduğu uç.

function pointer → **fonksiyon işaretçisi** 20
Bir fonksiyonun adresini tutan işaretçi; qsort'a karşılaştırma fonksiyonu böyle verilir.

G

garbage collection → **çöp toplama** 9
Artık erişilemeyen düğümleri otomatik olarak bulup geri kazanma.

generalized list → **genelleştirilmiş liste** 9
Öğeleri atom ya da başka bir liste olabilen liste.

global variable → **küresel değişken** 6
Bütün fonksiyonlardan erişilebilen, dosya düzeyinde tanımlı değişken.

graph → **çizge** 10
Düğümler ve onları bağlayan kenarlardan oluşan yapı: G = (V, E).

greedy method → **açgözlü yöntem** 17
Her adımda o an en iyi görünen seçimi yapıp geri dönmeyen yaklaşım.

H

hash function → **hash fonksiyonu** 14
Anahtarı tablo boyutundan küçük bir indise dönüştüren fonksiyon.

hash table → **hash tablosu** 14
Anahtarı bir hash fonksiyonuyla doğrudan bir kovaya eşleyen tablo; ortalama O(1) erişim.

head → **baş** 4
Bağlı listenin ilk düğümünü gösteren işaretçi.

header file → **başlık dosyası** 1
Tip ve fonksiyon bildirimlerini taşıyan, #include ile eklenen .h dosyası.

header node → **başlık düğümü** 4
Veri taşımayan, listenin başında duran ve boş liste özel durumunu ortadan kaldıran düğüm.

heap → **heap** 13
Her düğümün anahtarı çocuklarınınkinden büyük (ya da küçük) olan tam ikili ağaç.

heap memory → **heap bellek** 2
malloc ile ayrılan ve free edilene kadar yaşayan bellek bölgesi.

heap sort → **heap sort** 13
Diziyi heap'e çevirip kökü art arda sona taşıyarak sıralayan yerinde algoritma.

height → **yükseklik** 10
Bir düğümden en uzak yaprağa kadar olan düzey sayısı; AVL ağacında denge buna göre ölçülür.

heuristic → **sezgisel** 3
En iyi çözümü garanti etmeden, akıllıca bir seçim kuralıyla çözüme hızla yaklaşan yöntem; ör. at gezisinde Warnsdorff kuralı.

Horner's method → **Horner yöntemi** 2
Polinomu iç içe çarpımlarla hesaplama: $a_0 + x(a_1 + x(a_2 + \ldots))$.

hybrid sort → **hibrit sıralama** 20
Birden çok sıralama algoritmasını, örneğin hızlı ve eklemeli sıralamayı, birleştiren yöntem.

I

implementation → **gerçekleme** 1
Bir ADT'nin belirli bir veri yapısı ve kodla somut olarak yazılması.

in place → **yerinde** 2
Girdinin dışında yalnız sabit ek bellek kullanarak çalışma.

include guard → **çoklu ekleme koruması** 1
#ifndef … #define … #endif kalıbı; başlığın aynı dosyaya iki kez eklenmesini önler.

incoming precedence (icp) → **gelen öncelik** 8
Bir işlecin girdiden okunduğu andaki önceliği.

incomplete type → **eksik tip** 1
Boyutu henüz bilinmeyen tip; yalnız işaretçisi tanımlanabilir.

in-degree → **giriş derecesi** 15
Yönlü çizgede bir düğüme gelen kenar sayısı.

index → **indis** 2
Dizide bir öğenin konumunu belirten tam sayı.

infix → **ara ek** 8
İşlecin iki işlenenin arasına yazıldığı gösterim: a + b.

inorder → **kök ortada** 3
Sol alt ağaç, kök, sağ alt ağaç sırasıyla gezinme.

in-place sort → **yerinde sıralama** 13
Yalnız sabit ek bellek kullanan sıralama.

insertion sort → **eklemeli sıralama** 19
Her öğeyi önündeki sıralı bölümde doğru yere kaydırarak yerleştiren sıralama.

in-stack precedence (isp) → **yığın içi öncelik** 8
Bir işlecin yığında beklerken sahip olduğu öncelik.

interface → **arayüz** 1
Bir modülün dışarıya açtığı tipler ve fonksiyon bildirimleri; C'de başlık dosyasında durur.

internal node → **iç düğüm** 10
En az bir çocuğu olan düğüm.

introsort → **introsort** 20
Özyineleme derinliği aşılınca heap sort'a geçen hızlı sıralama.

invariant → **değişmez** 20
Bir döngünün ya da yapının her adımda doğru kalan özelliği.

inverse Ackermann function α(n) → **ters Ackermann fonksiyonu** 16
Pratikte 4'ü aşmayan, son derece yavaş büyüyen fonksiyon.

inversion → **ters çift** 19
Dizide i < j iken a[i] > a[j] olan öğe çifti.

isomorphic → **eş biçimli** 10
Düğümleri birebir eşlenerek aynı yapıya getirilebilen.

K

key → **anahtar** 5
Öğeleri aramak, sıralamak ya da ayırt etmek için kullanılan değer.

knight's tour → **at gezisi** 3
Satranç atıyla tahtanın her karesine bir kez uğrama problemi.

Kruskal's algorithm → **Kruskal algoritması** 16
Kenarları artan maliyetle deneyip döngü oluşturmayanları alan MST algoritması.

L

leaf → **yaprak** 3
Çocuğu olmayan düğüm.

level → **düzey** 10
Bir düğümün kökten uzaklığı; kök 1. düzeydedir.

level-order → **düzey sırası** 7
Düğümlere kuyruk yardımıyla düzey düzey, soldan sağa uğrama.

LIFO (last in, first out) → **son giren ilk çıkar** 6
Son eklenen öğenin ilk çıkarıldığı sıra; yığının kuralı.

linear probing → **doğrusal yoklama** 14
Çakışmada sırayla bir sonraki kovaya bakma.

linear (sequential) search → **doğrusal arama** 1
Öğeleri baştan sona tek tek karşılaştırarak arama; O(n).

link → **bağ** 4
Bir düğümü başka bir düğüme bağlayan işaretçi alanı.

linked list → **bağlı liste** 1
Her düğümün sonrakinin adresini tuttuğu, bellekte dağınık duran öğe dizisi.

list ADT → **liste ADT** 9
Konuma göre ekleme, silme ve erişim sunan sıralı öğe dizisi tipi.

load factor → **yük faktörü** 14
Tablodaki öğe sayısının kova sayısına oranı: α = n / m.

look-up table → **arama tablosu** 8
Koşullarla tek tek hesaplanacak değerlerin önceden konduğu dizi; ör. labirentte sekiz yönün satır ve sütun artışları.

lower bound → **alt sınır** 19
Bir problemi çözen her algoritmanın en az harcaması gereken süre; karşılaştırmalı sıralama için Ω(*n lg n*).

lower triangular matrix → **alt üçgen matris** 2
Ana köşegenin üstündeki bütün öğeleri sıfır olan kare matris.

M

macro → **makro** 1
#define ile tanımlanan ve derlemeden önce metin olarak yerine konan ad.

magic square → **sihirli kare** 1
Her satır, sütun ve ana köşegen toplamı eşit olan n×n sayı tablosu.

malloc → **malloc** 1
İstenen bayt sayısında bellek bloğu ayıran, başarısızlıkta NULL döndüren fonksiyon.

mapping → **eşleme** 2
Her indise bir değer karşılık getiren ilişki; soyut bakışta dizi budur.

max-heap → **max-heap** 13
Her düğümün çocuklarından büyük ya da eşit olduğu heap; en büyük öğe köktedir.

maze → **labirent** 6
Duvar ve yollardan oluşan ızgara; giriş-çıkış yolu yığınla geri izleyerek bulunur.

median-of-three → **üçün ortancası** 20
Pivotu ilk, orta ve son öğenin ortancası olarak seçme; sunumlarda "medyan-3" denir.

memory leak → **bellek sızıntısı** 4
Artık erişilemeyen ama free edilmemiş bellek.

merge → **birleştirme** 5
İki sıralı diziyi ya da listeyi tek sıralı dizide birleştirme.

merge sort → **birleştirme sıralaması** 20
Diziyi ikiye bölüp parçaları sıralayan ve sıralı parçaları birleştiren algoritma; O(*n lg n*).

min-heap → **min-heap** 13
Her düğümün çocuklarından küçük ya da eşit olduğu heap; en küçük öğe köktedir.

minimum spanning tree (MST) → **minimum kapsayan ağaç** 15
Kenar maliyetleri toplamı en küçük olan kapsayan ağaç.

modular arithmetic → **modüler aritmetik** 7
Kalanlarla hesap; dairesel kuyruk ve hash fonksiyonları buna dayanır.

Morris traversal → **Morris gezinmesi** 12
Kılavuzları geçici olarak kurup kaldırarak yığınsız ve O(1) ek bellekle kök ortada gezinme.

multidimensional array → **çok boyutlu dizi** 2
Öğelerine birden çok indisle erişilen dizi.

multiplication method → **çarpma yöntemi** 14
k·A'nın kesir kısmını m ile çarparak indis bulma.

multivariate polynomial → **çok değişkenli polinom** 9
Birden çok değişken içeren polinom; genelleştirilmiş listeyle gösterilir.

N

negative cycle → **negatif döngü** 18
Kenar ağırlıkları toplamı negatif olan döngü; en kısa yolu tanımsız kılar.

node → **düğüm** 3
Veri ve bir ya da daha çok bağ alanı taşıyan yapı.

n-queens problem → **n-vezir problemi** 3
n veziri n×n tahtaya birbirini tehdit etmeyecek biçimde yerleştirme problemi.

null character ('\0') → **dizgi sonu karakteri** 2
Dizginin bittiğini gösteren sıfır değerli karakter.

null pointer (NULL) → **boş işaretçi** 1
Hiçbir nesneyi göstermeyen işaretçi; boş listenin ve boş alt ağacın gösterimi.

O

opaque type → **opak tip** 1
Yalnız adı bildirilen, alanları kullanıcıdan gizlenen yapı tipi; C'de ADT'nin karşılığı.

open addressing → **açık adresleme** 14
Çakışmada tablonun içinde başka bir boş yer arama.

operator precedence → **işleç önceliği** 8
Bir ifadede hangi işlecin önce uygulanacağını belirleyen kural.

orthogonal list → **ortogonal liste** 15
Her kenar düğümünün hem satır hem sütun listesinde yer aldığı gösterim.

out-degree → **çıkış derecesi** 15
Yönlü çizgede bir düğümden çıkan kenar sayısı.

overflow → **taşma** 6
Dolu bir yapıya ekleme yapılmak istenmesi.

P

parent → **ebeveyn** 10
Bir düğümün hemen üstündeki düğüm.

parent array → **ebeveyn dizisi** 16
Her öğenin ağaçtaki ebeveynini tutan dizi; kökte negatif değer durur.

partition → **bölümleme** 20
Diziyi pivottan küçükler ve büyükler olarak iki parçaya ayırma.

path → **yol** 15
Art arda komşu düğümlerden oluşan dizi.

path compression → **yol kısaltma** 16
Azaltma kuralının genel adı; sonraki bulma işlemlerini kısaltır.

perfect binary tree → **mükemmel ikili ağaç** 10
k derinliğinde $2^k - 1$ düğümü olan, bütün düzeyleri dolu ikili ağaç; ders kitabı (Horowitz) bu ağaca *full binary tree* der.

performance analysis → **performans analizi** 1
Bir programın zaman ve bellek gereksiniminin makineden bağımsız olarak kestirilmesi.

permutation → **permütasyon** 1
Bir kümenin öğelerinin olası dizilişlerinden biri; n öğenin n! permütasyonu vardır.

pivot → **pivot** 20
Hızlı sıralamada diziyi ikiye ayırmak için seçilen öğe.

pointer → **işaretçi** 2
Bir nesnenin bellek adresini tutan değişken.

pointer arithmetic → **işaretçi aritmetiği** 2
İşaretçiye tam sayı ekleyip çıkarma; birim bayt değil, gösterilen tipin bir öğesidir.

pointer to pointer → **işaretçiye işaretçi** 9
Bir işaretçinin adresini tutan işaretçi; çağıranın işaretçisini değiştirmek için kullanılır.

polynomial → **polinom** 5
Katsayı ve üs çiftlerinden oluşan terimlerin toplamı.

pop → **pop (üstten çıkarma)** 6
Yığının üstündeki öğeyi çıkarma.

postfix (reverse Polish notation) → **sonek** 8
İşlecin işlenenlerden sonra yazıldığı, parantez gerektirmeyen gösterim: a b +.

postorder → **kök sonda** 3
Sol alt ağaç, sağ alt ağaç, kök sırasıyla gezinme.

predecessor → **öncül** 12
Kök ortada sırada bir düğümden hemen önce gelen düğüm.

prefix (Polish notation) → **önek** 8
İşlecin işlenenlerden önce yazıldığı gösterim: + a b.

preorder → **kök önde** 9
Kök, sol alt ağaç, sağ alt ağaç sırasıyla gezinme.

preprocessor → **önişlemci** 1
#include, #define gibi yönergeleri derlemeden önce işleyen aşama.

primary clustering → **kümelenme** 14
Doğrusal yoklamada dolu kovaların uzun ardışık bloklar oluşturması; sunumlarda "birincil kümelenme" biçimiyle geçer.

Prim's algorithm → **Prim algoritması** 17
Tek düğümden başlayıp ağaca en ucuz kenarla bağlanan düğümü ekleyen MST algoritması.

priority queue → **öncelikli kuyruk** 5
Her zaman en yüksek (ya da en düşük) öncelikli öğenin çıkarıldığı kuyruk.

probe → **yoklama** 14
Açık adreslemede bir kovanın boş olup olmadığına bakma.

program step → **program adımı** 1
Süresi girdi boyutundan bağımsız, anlamlı bir kod parçası; adım saymanın birimi.

push → **push (üste ekleme)** 4
Yığının üstüne öğe ekleme.

Q

quadratic probing → **karesel yoklama** 14
Çakışmada i^2 adım uzaktaki kovalara bakma.

queue → **kuyruk** 7
Eklemenin arkadan, çıkarmanın önden yapıldığı doğrusal yapı.

quick sort → **hızlı sıralama** 19
Diziyi bir pivot öğeye göre bölümleyip iki parçayı özyinelemeyle sıralayan algoritma.

R

radix sort → **basamak sıralaması** 20
Sayıları en az anlamlı basamaktan başlayarak kararlı bir sıralamayla basamak basamak sıralama.

realloc → **realloc** 1
Ayrılmış bir bloğu büyüten ya da küçülten, gerekirse taşıyan fonksiyon.

rear → **arka** 7
Kuyrukta yeni öğenin eklendiği uç.

record → **kayıt** 19
Bir ya da daha çok alandan oluşan, sıralamada bütün olarak taşınan veri birimi.

recurrence relation → **yineleme bağıntısı** 3
Bir fonksiyonu kendi küçük değerleri cinsinden tanımlayan eşitlik; T(n) = 2T(n/2) + n gibi.

recursion → **özyineleme** 1
Bir fonksiyonun kendini daha küçük bir girdiyle çağırması.

recursive → **özyinelemeli** 1
Kendini çağıran; özyineleme kullanan.

red-black tree → **kırmızı-siyah ağaç** 11
Düğüm renklerine konan kurallarla yüksekliği O(*lg n*) tutan arama ağacı.

reference count → **referans sayısı** 9
Bir düğümü gösteren işaretçi sayısı; sıfıra inince düğüm geri verilir.

rehashing (resizing) → **yeniden boyutlandırma** 14
Yük faktörü büyüyünce tabloyu büyütüp bütün anahtarları yeniden hash'leme.

relaxation → **gevşetme** 18
Bir kenar üzerinden daha kısa bir yol bulunursa uzaklık değerini güncelleme.

return address → **dönüş adresi** 3
Çağrı bitince yürütmenin süreceği komutun adresi.

reversing a list → **listeyi ters çevirme** 4
Bağların yönünü değiştirerek listenin sırasını tersine çevirme; yerinde ve O(n).

root → **kök** 3
Ağaçta ebeveyni olmayan en üst düğüm.

rotation → **döndürme** 11
Arama sırasını koruyarak ağacın yerel yapısını değiştiren işlem.

row-major order → **satır öncelikli düzen** 2
Çok boyutlu dizinin satır satır art arda yerleştirilmesi; C'nin düzeni.

S

satisfiability (SAT) → **sağlanabilirlik** 3
Bir mantık formülünü doğru yapan değişken ataması olup olmadığı problemi.

selection sort → **seçimli sıralama** 1
Her adımda kalan öğelerin en küçüğünü bulup sıradaki konuma koyan sıralama.

self-balancing tree → **kendini dengeleyen ağaç** 11
Ekleme ve silmeden sonra dengesini döndürmelerle kendiliğinden düzelten ağaç.

self-referential struct → **kendine başvuran yapı** 2
Kendi tipinden bir işaretçi alanı taşıyan yapı; bağlı düğümlerin temeli.

sentinel → **bekçi** 1
Döngüden ayrı bir sınır denetimini gereksiz kılmak için sona konan özel değer ya da düğüm.

separate chaining → **ayrı zincirleme** 14
Her kovada o kovaya düşen anahtarları bir bağlı listede tutma; sunumlarda kısaca "zincirleme" denir.

Shell sort → **Shell sıralaması** 19
Eklemeli sıralamayı azalan aralıklarla uzak öğelere uygulayan sıralama.

short-circuit evaluation → **kısa devre değerlendirme** 2
&& ve || işleçlerinde sonuç belli olunca kalan işlenenin değerlendirilmemesi.

shortest path → **en kısa yol** 8
İki düğüm arasındaki kenar maliyetleri toplamı en küçük olan yol.

sibling → **kardeş** 3
Aynı ebeveynin çocukları.

singly linked list → **tek yönlü bağlı liste** 4
Her düğümün yalnız sonraki düğümü gösterdiği bağlı liste.

sizeof → **sizeof işleci** 1
Bir tipin ya da nesnenin bayt cinsinden boyunu derleme sırasında veren işleç.

skewed tree → **çarpık ağaç** 10
Bütün düğümleri tek bir yöne uzanan, liste gibi davranan ağaç.

slack → **bolluk** 18
Bir işin en geç ve en erken zamanı arasındaki fark; kritik işlerde sıfırdır.

Sollin's (Borůvka's) algorithm → **Sollin (Borůvka) algoritması** 17
Her turda her bileşenin en ucuz dış kenarını seçtiği MST algoritması.

sorting → **sıralama** 1
Öğeleri anahtarlarına göre artan ya da azalan düzene koyma.

space complexity → **alan karmaşıklığı** 1
Bir programın girdi boyutuna bağlı olarak gereksinim duyduğu bellek miktarı.

spanning tree → **kapsayan ağaç** 15
Bağlantılı bir çizgenin bütün düğümlerini döngüsüz bağlayan alt çizgesi; n − 1 kenarı vardır.

sparse matrix → **seyrek matris** 5
Öğelerinin çoğu sıfır olan matris; yalnız sıfır olmayanlar saklanır.

splay tree → **splay ağacı** 11
Erişilen düğümü döndürmelerle köke taşıyan arama ağacı.

stable (sort) → **kararlı** 19
Eşit anahtarlı öğelerin göreli sırasını koruyan sıralama.

stack → **yığın** 6
Ekleme ve çıkarmanın yalnız üst uçtan yapıldığı doğrusal yapı.

stack frame (activation record) → **yığın çerçevesi** 1
Bir çağrının parametrelerini, yerel değişkenlerini ve dönüş adresini tutan kayıt; metinde kısaca "çerçeve" diye de geçer.

stack overflow → **yığın taşması** 3
Çağrı yığınının ayrılan belleği aşması; çoğunlukla çok derin ya da sonsuz özyinelemede olur.

string → **dizgi** 2
C'de '\0' ile biten char dizisi.

struct → **struct (yapı)** 1
Farklı tipteki alanları tek bir kayıtta toplayan C tipi.

subgraph → **alt çizge** 15
Düğümleri ve kenarları bir çizgenin düğüm ve kenarlarının alt kümesi olan çizge.

subset → **alt küme** 3
Bir kümenin öğelerinden bazılarının oluşturduğu küme; n öğenin 2^n alt kümesi vardır.

subtree → **alt ağaç** 9
Bir düğüm ve onun bütün soyundan oluşan ağaç.

successor → **ardıl** 12
Kök ortada sırada bir düğümden hemen sonra gelen düğüm; silmede ve kılavuzlu ağaçlarda kullanılır.

swap → **takas** 1
İki değişkenin değerlerini yer değiştirme.

system life cycle → **sistem yaşam döngüsü** 1
Gereksinim, çözümleme, tasarım, kodlama ve doğrulama aşamalarından oluşan geliştirme süreci.

T

tagged union → **etiketli union** 2
Hangi alanın geçerli olduğunu bir enum etiketiyle tutan yapı.

tail (of a generalized list) → **kuyruk** 9
Genelleştirilmiş listede ilk öğe çıkarıldıktan sonra kalan liste.

tail recursion → **kuyruk özyinelemesi** 3
Özyinelemeli çağrının fonksiyonun son işlemi olduğu özyineleme; döngüye kolayca çevrilir.

thread → **kılavuz** 10
Kılavuzlu ağaçta boş çocuk yerine öncül ya da ardılı gösteren işaretçi.

threaded binary tree → **kılavuzlu ikili ağaç** 10
Boş işaretçileri kök ortada öncül ve ardılı gösteren ikili ağaç.

time complexity → **zaman karmaşıklığı** 1
Bir programın girdi boyutuna bağlı olarak çalıştırdığı adım sayısı.

token → **belirteç** 8
İfade ayrıştırılırken tek parça olarak işlenen öğe: işlenen, işleç ya da parantez.

tombstone → **silindi işareti** 14
Açık adreslemede silinen kovaya konan, aramanın orada durmamasını sağlayan işaret.

top → **üst** 6
Yığında son eklenen öğenin bulunduğu uç.

top-down → **yukarıdan aşağı** 1
Problemi önce bütün olarak ele alıp giderek küçük parçalara bölen tasarım yaklaşımı.

topological order → **topolojik sıra** 15
Her kenarın başının sonundan önce geldiği düğüm sırası.

tortoise and hare → **tavşan ve kaplumbağa** 4
Biri bir, öteki iki adım ilerleyen iki işaretçiyle döngü bulan Floyd yöntemi.

Towers of Hanoi → **Hanoi kuleleri** 3
Diskleri üç çubuk arasında kurallara uyarak taşıma problemi; $2^n - 1$ hamle gerekir.

transpose → **transpoz** 2
Matrisin satırlarıyla sütunlarının yer değiştirmiş hâli.

traversal → **gezinme** 2
Bir yapının bütün öğelerine belirli bir sırayla birer kez uğrama.

tree → **ağaç** 1
Bir kökten başlayıp alt ağaçlara ayrılan, döngü içermeyen hiyerarşik yapı.

triple → **üçlü** 5
Seyrek matriste sıfır olmayan bir öğeyi tutan (satır, sütun, değer) kaydı.

typedef → **typedef (tip adı)** 1
Var olan bir tipe yeni bir ad veren C bildirimi.

U

underflow → **alt taşma** 6
Boş bir yapıdan öğe çıkarılmak istenmesi.

undirected graph → **yönsüz çizge** 15
Kenarlarının yönü olmayan çizge.

undo → **geri al** 6
Yapılan son işlemleri ters sırayla geri alma; yığınla gerçeklenir.

union → **birleşim** 16
İki kümeyi tek kümede toplama işlemi.

union by rank → **rütbeye göre birleştirme** 16
Birleşimde ağaç ağırlığı yerine yükseklik sınırını (rütbe) kullanma.

union (C) → **union (birlik)** 2
Bütün alanları aynı bellek bölgesini paylaşan C tipi; aynı anda tek alan geçerlidir.

union-find → **union-find** 16
Birleşim ve küme bulma işlemlerini destekleyen ayrık küme yapısı.

up-heap → **up-heap (yukarı süzme)** 13
Eklenen öğenin ebeveyniyle yer değiştirerek uygun yere çıkması.

V

vertex → **düğüm** 15
Çizgenin bir öğesi; ağaçtaki node ile aynı karşılık kullanılır.

*void pointer (void *)* → **void işaretçi** 20
Tipi belirtilmemiş işaretçi; qsort gibi genel yordamlarda her tipe dönüştürülür.

W

weighted graph (network) → **ağırlıklı çizge** 15
Kenarlarına maliyet ya da uzunluk atanmış çizge.

weighting rule → **ağırlıklı kural** 16
Birleşimde az düğümlü ağacı çok düğümlü ağacın altına bağlama; boyuta göre birleştirme (*union by size*) de denir.

worst case → **en kötü durum** 1
Aynı boyuttaki girdiler arasında en çok adım gerektiren girdi için süre.

EK B

C ile Veri Yapıları İçin Ön Bilgiler

Bu derste her yapı C ile, bellek elle yönetilerek kurulur: bir düğüm `struct` ile tanımlanır, `malloc` ile ayrılır, işaretçilerle bağlanır ve işi bitince `free` ile geri verilir. Bu yüzden C'nin tipleri, işaretçileri, dizileri ve dinamik belleği derse girmeden önce rahatça okunabilmelidir. Maliyetleri karşılaştırmak için de logaritma, toplam formülleri ve tümevarım gibi küçük bir matematik birikimi gerekir.

Her kartta konunun kısa bir özeti, küçük bir C örneği ve bir kontrol sorusu yer alır; cevap sorunun altındaki açılır kutudadır. Bağlantılar konunun derste kullanıldığı slaytlara gider.

Önerilen hazırlık sırası

Tipler ve fonksiyonlar · İşaretçiler · Diziler ve dizgiler · struct · Dinamik bellek · Özyineleme · Logaritma ve toplamlar · Büyük O

C temelleri

TB

Temel Tipler ve sizeof

int, char, double · tam sayı bölmesi · bayt sayısı

C'de her değişkenin bir tipi ve bellekte kapladığı belirli bir bayt sayısı vardır; `sizeof` bu sayıyı derleme sırasında verir. İki tam sayı arasındaki bölme tam sayı bölmesidir: kesirli kısım atılır. Dizi indisleri, heap'te ebeveyn indisi (`i / 2`) ve ikili aramadaki orta nokta hep bu kurala dayanır.

```
int a = 7, b = 2;
printf("%d\n", a / b);           /* 3   */
printf("%d\n", a % b);           /* 1   */
printf("%.1f\n", (double)a / b); /* 3.5 */
printf("%zu\n", sizeof(int));    /* çoğu sistemde 4 */
```

Kontrol sorusu: `double m = 7 / 2;` satırından sonra `m` değişkeninin değeri nedir?

▶ Cevap

DERSTE Bölüm 2 · C'de Tek Boyutlu Dizi · Bölüm 13 · Dizide tam ikili ağaç

FD

Fonksiyonlar ve Değerle Çağrı

parametre · dönüş değeri · kopya

C'de bütün argümanlar değerle geçirilir (*call by value*): fonksiyon, çağıranın değişkeninin bir kopyasını alır. Bu yüzden iki değişkeni takas etmek isteyen bir fonksiyon, değişkenlerin kendilerini değil adreslerini almalıdır. Derste `swap`, yığının `top` sayacı ve arama ağacına ekleme hep bu nedenle adres alır.

```
void takas_yanlis(int x, int y) { int t = x; x = y; y = t; }
void takas(int *x, int *y)      { int t = *x; *x = *y; *y = t; }

int a = 1, b = 2;
takas_yanlis(a, b);   /* a == 1, b == 2: kopyalar değişti */
takas(&a, &b);        /* a == 2, b == 1 */
```

Kontrol sorusu: `takas_yanlis` içindeki üç atama çalıştığı hâlde `main`'deki `a` neden değişmez?

▶ **Cevap**

DERSTE **Bölüm 1 · SWAP: Makro ve Fonksiyon** **Bölüm 1 · swap() Adım Adım** **Bölüm 6 · ++*top mu, *top++ mı?**

DZ

Diziler ve Dizgiler

ardışık bellek · '\0' ile biten char dizisi

Dizi, aynı tipte öğelerin bellekte art arda durduğu bir bloktur; `a[i]` erişimi O(1) sürer ve C sınır denetimi yapmaz. Bir diziyi fonksiyona geçirmek, ilk öğesinin adresini geçirmektir; boyut ayrıca verilmelidir. Dizgi (*string*) ise sonunda `'\0'` bulunan bir `char` dizisidir: `"dog"` dört bayt kaplar.

```
int topla(const int a[], int n) {
    int s = 0;
    for (int i = 0; i < n; i++) s += a[i];
    return s;
}

int uzunluk(const char *s) {
    int n = 0;
    while (s[n] != '\0') n++;
    return n;
}
/* topla((int[]){4, 5, 6}, 3) == 15, uzunluk("dog") == 3 */
```

Kontrol sorusu: `char s[] = "veri";` tanımında `sizeof s` ile `uzunluk(s)` aynı değeri mi verir?

▶ **Cevap**

DERSTE **Bölüm 2 · C'de Tek Boyutlu Dizi** **Bölüm 2 · Dizgi = char Dizisi** **Bölüm 2 · uzunluk() Adım Adım**

ST

struct ve typedef

kayıt · nokta ve ok işleci · kendine başvuran yapı

`struct`, farklı tipteki alanları tek bir kayıtta toplar; `typedef` bu tipe kısa bir ad verir. Alana değişken üzerinden `.`, işaretçi üzerinden `->` ile erişilir; `p->deger` yazımı `(*p).deger` ile aynıdır. Bağlı listenin düğümü, kendi tipinden bir işaretçi taşıyan kendine başvuran yapıdır (*self-referential struct*).

```
typedef struct dugum *dugum_gost;
typedef struct dugum {
    int deger;
    dugum_gost bag;      /* sonraki düğüm ya da NULL */
} dugum;

dugum d = { 3, NULL };
dugum_gost p = &d;
p->deger = 8;            /* d.deger artık 8 */
```

Kontrol sorusu: Bir yapı, kendi tipinden bir alanı (`struct dugum sonraki;`) doğrudan içerebilir mi?

▶ **Cevap**

DERSTE Bölüm 2 · struct · Bölüm 2 · İç İçe ve Kendine Başvuran · Bölüm 4 · Düğüm Yapısı

MB

Makrolar ve Başlık Dosyaları

#define · #include · opak tip

Önişlemci, `#define` ile tanımlanan makroları derlemeden önce metin olarak yerine koyar; bu yüzden makro parametreleri parantez içinde yazılır. Başlık dosyası (`.h`) bir modülün arayüzünü, `.c` dosyası ise gerçeklemesini taşır. Başlıkta yalnız `typedef struct kesir Kesir;` bildirilirse yapının içi kullanıcıdan gizlenir: bu bir opak tiptir ve C'de soyut veri tipinin karşılığıdır.

```
/* kesir.h */
#ifndef KESIR_H
#define KESIR_H
typedef struct kesir Kesir;     /* opak: alanlar görünmez */
Kesir *kesir_olustur(int pay, int payda);
void   kesir_sil(Kesir *k);
#endif

#define KARE(x) ((x) * (x))     /* KARE(1 + 2) == 9 */
```

Kontrol sorusu: Makro `#define KARE(x) x * x` biçiminde yazılsaydı `KARE(1 + 2)` kaç olurdu?

▶ **Cevap**

DERSTE Bölüm 1 · C'de Soyutlama: Başlık Dosyası · Bölüm 1 · Arayüzü Kullanmak · Bölüm 19 · Ortak Altyapı: Makrolar

EN

bool ve enum

<stdbool.h> · true/false · adlandırılmış sabitler

C99'dan beri `<stdbool.h>` başlığı `bool` tipini ve `true`, `false` sabitlerini verir; derste ekleme ve silme fonksiyonları işin başarılı olup olmadığını `bool` ile bildirir. Koşullarda C yine tam sayıya bakar: 0 yanlış, sıfırdan farklı her değer doğrudur. `enum` bir grup tam sayı sabitine ad verir; ilk ad 0 değerini alır, sonrakiler birer artar. İşleç türleri ve düğüm etiketleri böylece sayı yerine okunaklı adlarla yazılır.

```
#include <stdbool.h>

typedef enum { beyaz, gri, siyah } renk;   /* 0, 1, 2 */

bool bos_mu(int n) { return n == 0; }

renk r = gri;
printf("%d %d\n", r, bos_mu(0));           /* 1 1 */
```

Kontrol sorusu: `enum { lparen, rparen, plus, minus, times };` tanımında `times` hangi değeri alır?

▶ **Cevap**

DERSTE **Bölüm 8 · token_al** **Bölüm 8 · isp ve icp** **Bölüm 9 · Genelleştirilmiş Liste**

ÖY

Özyineleme ve Çağrı Yığını

taban durum · yığın çerçevesi · yığın taşması

Kendini çağıran bir fonksiyon, her çağrıda problemi küçültmeli ve bir taban duruma ulaşmalıdır. Her çağrı için çağrı yığınına yerel değişkenleri ve dönüş adresini tutan bir çerçeve eklenir; derinlik n ise ek bellek O(n) olur. Ağaç gezinmeleri, ikili arama, Hanoi kuleleri ve hızlı sıralama bu düzenle yazılır.

```
long fakt(int n) {
    if (n <= 1) return 1;        /* taban durum */
    return n * fakt(n - 1);      /* küçülen problem */
}
/* fakt(4) -> 4 * fakt(3) -> 4 * 3 * fakt(2)
           -> 4 * 3 * 2 * fakt(1) = 24 */
```

Kontrol sorusu: `fakt(4)` çağrıldığında çağrı yığınında aynı anda en çok kaç `fakt` çerçevesi bulunur?

▶ **Cevap**

DERSTE **Bölüm 3 · Özyinelemeli Algoritmalar** **Bölüm 3 · Yığın Çerçevesi** **Bölüm 3 · fakt(3) Adım Adım**

İşaretçiler ve bellek

İŞ

İşaretçiler

& ve * · NULL · takma ad

İşaretçi (*pointer*), bir nesnenin adresini tutan değişkendir. `&x` x'in adresini verir, `*p` p'nin gösterdiği nesneye ulaşır. `NULL` hiçbir yeri göstermeyen işaretçidir ve derste boş listenin, boş alt ağacın gösterimidir. İki işaretçi aynı nesneyi gösterebilir; biri üzerinden yapılan değişiklik ötekinden de görülür.

```
int x = 5;
int *p = &x;     /* p, x'in adresini tutar */
int *q = p;      /* q de x'i gösterir      */
*q = 7;          /* x == 7, *p == 7        */
p = NULL;        /* p artık hiçbir yeri göstermez; x hâlâ 7 */
```

Kontrol sorusu: Son satırdan sonra `*q` ifadesinin değeri nedir?

▶ **Cevap**

DERSTE Bölüm 4 · İşaretçi Hatırlatması · Bölüm 4 · Bellekte Düğümler · Bölüm 6 · İşaretçiler ve Bellek

İA

Dizi ve İşaretçi Aritmetiği

a[i] == *(a + i) · öğe sayar, bayt değil

Bir ifade içinde dizinin adı ilk öğesinin adresine dönüşür. İşaretçiye 1 eklemek onu bir bayt değil, bir sonraki öğeye ilerletir: `int` 4 bayt ise adres 4 artar. `a[i]` yazımı tanım gereği `*(a + i)` demektir. Çok boyutlu dizilerin satır öncelikli adres hesabı da bu kurala dayanır.

```
int a[] = {10, 20, 30, 40};
int *p = a;               /* p == &a[0] */
printf("%d\n", *(p + 2)); /* 30 */
p++;                      /* p == &a[1] */
printf("%d\n", p[1]);     /* 30: p[1] == a[2] */
printf("%td\n", &a[3] - p);  /* 2 öğe */
```

Kontrol sorusu: `int` 4 bayt ve `a` dizisi 1000 adresinden başlıyorsa `a + 3` hangi adresi gösterir?

▶ **Cevap**

DERSTE Bölüm 2 · Dizi ve İşaretçi · Bölüm 2 · yazdir() Adım Adım · Bölüm 2 · Satır ve Sütun Öncelikli

DB

Dinamik Bellek

malloc · calloc · realloc · free

Yerel değişkenler fonksiyon bitince yok olur; `malloc` ile ayrılan blok ise `free` çağrılana kadar heap bellekte yaşar. `malloc` bellek kalmazsa `NULL` döndürür, bu yüzden dönüş değeri her seferinde denetlenir. Geri verilmeyen blok bellek sızıntısına (*memory leak*), geri verildikten sonra kullanılan işaretçi ise sarkan işaretçiye (*dangling pointer*) yol açar.

```
int n = 5;
int *a = malloc(n * sizeof *a);
if (a == NULL) return 1;          /* bellek yok */
for (int i = 0; i < n; i++) a[i] = i * i;
int *b = realloc(a, 2 * n * sizeof *a);  /* büyüt */
if (b == NULL) { free(a); return 1; }
a = b;
free(a);
a = NULL;                         /* sarkan işaretçi kalmasın */
```

Kontrol sorusu: `a = realloc(a, yeni_boy);` yazımı, `realloc` başarısız olursa neden sızıntıya yol açar?

▶ **Cevap**

DERSTE Bölüm 2 · malloc ile Dizi · Bölüm 4 · malloc ve free · Bölüm 4 · Serbest Bırakma Adım Adım

Çİ

İşaretçiye İşaretçi

listenin başını değiştirmek · &kok

Bir fonksiyon, çağıranın işaretçisini (örneğin listenin başını ya da ağacın kökünü) değiştirecekse o işaretçinin adresini almalıdır. Tip bu durumda `dugum **` olur. Değerle çağrı kuralının işaretçilere uygulanmasıdır: işaretçi de bir değişkendir ve kopyalanarak geçirilir.

```
void basa_ekle(dugum_gost *bas, dugum_gost yeni) {
    yeni->bag = *bas;   /* yeni düğüm eski başı gösterir */
    *bas = yeni;        /* çağıranın baş işaretçisi değişir */
}

dugum_gost liste = NULL;
basa_ekle(&liste, p);   /* liste artık p'nin düğümünü gösterir */
```

Kontrol sorusu: Fonksiyon `void basa_ekle(dugum_gost bas, dugum_gost yeni)` biçiminde yazılıp sonda `bas = yeni;` yapılsaydı ve liste boşsa, çağrıdan sonra `liste` ne olurdu?

▶ **Cevap**

DERSTE Bölüm 4 · Başa Ekleme · Bölüm 11 · Neden agac_gost *L

Fİ

Fonksiyon İşaretçileri ve qsort

void * · karşılaştırma fonksiyonu

Fonksiyonun adı da bir adrestir ve başka bir fonksiyona argüman olarak geçirilebilir. Standart kitaplıktaki `qsort`, öğe tipini bilmeden çalışır: diziyi `void *` olarak alır ve iki öğenin sırasını, kullanıcının verdiği karşılaştırma fonksiyonuna sorar. Fonksiyon negatif, sıfır ya da pozitif döndürür.

```
int artan(const void *x, const void *y) {
    int a = *(const int *)x, b = *(const int *)y;
    return (a > b) - (a < b);   /* -1, 0 ya da 1 */
}

int d[] = {37, 15, 24};
qsort(d, 3, sizeof d[0], artan);   /* d: 15 24 37 */
```

Kontrol sorusu: Karşılaştırma için kısaca `return a - b;` yazmak hangi durumda yanlış sonuç verir?

▶ **Cevap**

DERSTE Bölüm 20 · qsort Kullanımı · Bölüm 20 · qsort ile Kayıt Sıralama

Matematik

LG

Üsler ve Logaritmalar

lg n = $\log_2 n$ · art arda ikiye bölme

Derste `lg n`, 2 tabanında logaritmadır: n'yi 1'e indirmek için kaç kez ikiye bölmek gerektiğini söyler. $2^{10} = 1024 \approx 10^3$ olduğundan bir milyon öğede lg n ≈ 20'dir. İkili arama, dengeli ağacın yüksekliği ve heap işlemleri bu yüzden O(lg n) sürer. Logaritmanın tabanı yalnız sabit bir çarpanı değiştirir; büyük O içinde taban yazılmaz.

```
int ikiye_bolme_sayisi(int n) {
    int k = 0;
    while (n > 1) { n /= 2; k++; }
    return k;                    /* ⌊lg n⌋ */
}
/* 1 → 0, 8 → 3, 1000 → 9, 1024 → 10 */
```

Kontrol sorusu: Sıralı bir dizide 1.000.000 öğe varsa ikili arama en çok yaklaşık kaç karşılaştırma yapar?

▶ **Cevap**

DERSTE Bölüm 3 · İkili Aramanın Karmaşıklığı · Bölüm 10 · Düğüm Sayısı İlişkileri · Bölüm 1 · Alışılmış Karmaşıklıklar

TF

Toplam Formülleri

aritmetik ve geometrik seriler

İç içe döngülerin maliyeti çoğu zaman bir toplamdır. İki formül derste sürekli kullanılır: $1 + 2 + \ldots + n = n(n + 1)/2$ ve $1 + 2 + 4 + \ldots + 2^k = 2^{k+1} - 1$. İlki seçimli ve eklemeli sıralamanın $O(n^2)$ olduğunu, ikincisi k derinlikli ikili ağacın en çok $2^k - 1$ düğüm taşıdığını ve kapasitesi ikiye katlanan dizide toplam kopya sayısının $O(n)$ kaldığını gösterir.

```
int sayac = 0;
for (int i = 0; i < n - 1; i++)       /* seçimli sıralama */
    for (int j = i + 1; j < n; j++)
        sayac++;                      /* bir karşılaştırma */
/* sayac == (n-1) + (n-2) + ... + 1 == n(n-1)/2
   n = 5 için 10 */
```

Kontrol sorusu: Kapasitesi 1'den başlayıp her dolduğunda ikiye katlanan bir diziye 16 öğe eklenirse büyütmeler sırasında toplam kaç öğe kopyalanır?

▶ **Cevap**

DERSTE Bölüm 19 · Seçimli: Analiz · Bölüm 2 · Amortize Maliyet · Bölüm 13 · Neden O(n)?

TV

Tümevarım

taban adım · tümevarım adımı

Bir önerme n = 1 için doğruysa (taban adım) ve n − 1 için doğruluğu n için doğruluğunu gerektiriyorsa (tümevarım adımı), bütün $n \geq 1$ için doğrudur. Özyinelemeli bir fonksiyonun doğruluğu da aynı biçimde kanıtlanır: taban durum doğru sonuç verir, küçük çağrılar doğruysa büyük çağrı da doğrudur. Ağaçlardaki düğüm sayısı lemmaları bu yöntemle kanıtlanır.

```
/* Önerme: toplam(n) == n * (n + 1) / 2 */
int toplam(int n) {
    if (n == 0) return 0;            /* taban: 0 == 0·1/2 */
    return toplam(n - 1) + n;        /* (n-1)n/2 + n == n(n+1)/2 */
}
```

Kontrol sorusu: İkili ağacın i. düzeyinde en çok 2^{i-1} düğüm bulunduğunu kanıtlarken tümevarım adımında hangi gözlem kullanılır?

▶ **Cevap**

DERSTE Bölüm 10 · Lemma · Bölüm 3 · Hanoi: Kaç Hamle? · Bölüm 15 · Teorem 1: Σd = 2e

MA

Modüler Aritmetik

% işleci · dairesel indis · hash

`x % m`, x'in m'ye bölümünden kalandır ve negatif olmayan x için 0 ile m − 1 arasında kalır. Dairesel kuyrukta dizinin sonuna gelen indis `(arka + 1) % MAX` ile başa döner; hash tablosunda `k % m` anahtarı bir kovaya eşler. C'de negatif bir sayının kalanı negatif çıkabilir; hash hesaplarında bu yüzden `unsigned` tipler kullanılır.

```
#define MAX 5
int arka = 4;
arka = (arka + 1) % MAX;     /* 0: başa dönüldü */

unsigned h(unsigned k, unsigned m) { return k % m; }
/* m = 13: h(27, 13) == 1, h(40, 13) == 1 → çakışma */
```

Kontrol sorusu: m = 13 için 18 ve 44 anahtarları hangi kovalara düşer?

▶ **Cevap**

DERSTE **Bölüm 7 · Modüler Aritmetik** **Bölüm 14 · Bölme yöntemi**

KO

Sayma: Faktöriyel ve Alt Kümeler

n! · 2^n · kaba kuvvetin sınırı

n öğenin n! farklı dizilişi ve 2^n farklı alt kümesi vardır. Kaba kuvvet algoritmaları bu sayıların tümünü dener; permütasyon üretimi O(n · n!), alt küme üretimi ve SAT denemesi $O(2^n)$ sürer. 2^n ve n! o kadar hızlı büyür ki n = 25 civarında bile en hızlı makine yetmez; geri izleme ve iyi bir algoritma bu yüzden önemlidir.

```
/* n öğenin bütün alt kümeleri: her bit bir öğe */
for (unsigned m = 0; m < (1u << n); m++) {
    for (int i = 0; i < n; i++)
        if (m & (1u << i)) printf("%d ", a[i]);
    printf("\n");
}   /* n = 3 için 8 satır, ilki boş küme */
```

Kontrol sorusu: "abcd" dizgisinin kaç farklı permütasyonu vardır ve her birinde kaç harf yazılır?

▶ **Cevap**

DERSTE **Bölüm 3 · Permütasyon Problemi** **Bölüm 3 · Alt Küme Üretimi Adım Adım** **Bölüm 1 · Daha Hızlı Makine Yetmez**

MX

Matrisler ve İki Boyutlu Diziler

satır öncelikli saklama · devrik · çarpım

m × n bir matris C'de `int a[M][N]` ile tutulur ve satır öncelikli saklanır: `a[i][j]`, dizinin başından i·N + j öğe ötededir. Devrik (transpoz) satırları sütun yapar: $A^T[j][i] = A[i][j]$. m × n bir matrisle n × p bir matrisin çarpımı m × p'dir; her öğe bir satırla bir sütunun iç çarpımıdır ve üç iç içe döngüyle toplam m·n·p çarpma yapılır. Seyrek matrisler, çizgelerin komşuluk matrisi ve tüm çiftler için en kısa yol bu işlemlerle yazılır.

```
/* c = a × b : a M×N, b N×P, c M×P */
for (int i = 0; i < M; i++)
    for (int j = 0; j < P; j++) {
        c[i][j] = 0;
        for (int k = 0; k < N; k++)
            c[i][j] += a[i][k] * b[k][j];
    }
```

Kontrol sorusu: A 2 × 3, B ise 3 × 4 boyutlu. AB kaç öğelidir ve hesabı kaç çarpma gerektirir?

▶ **Cevap**

DERSTE Bölüm 5 · Transpoz Nedir? Bölüm 5 · Seyrek Çarpım Fikri Bölüm 15 · Komşuluk Matrisi

E[X]

Temel Olasılık ve Beklenen Değer

düzgün dağılım · ortalama durum · yük faktörü

Ortalama durum analizi, girdinin ya da hash değerlerinin rastgele dağıldığı varsayımıyla yapılır. Bir niceliğin beklenen değeri, alabileceği her değerin olasılığıyla çarpılıp toplanmasıdır; bir toplamın beklenen değeri de terimlerin beklenen değerlerinin toplamıdır. Sıralı aramada aranan öğe her konumda eşit olasılıkla bulunuyorsa ortalama (n + 1)/2 karşılaştırma yapılır. n anahtar m kovaya düzgün dağılırsa bir kovaya düşen beklenen anahtar sayısı n/m'dir; bu oran hash tablosunun yük faktörü α'dır.

```
/* öğe k. konumdaysa k karşılaştırma, her konumun olasılığı 1/n */
double ort = 0;
for (int k = 1; k <= n; k++)
    ort += k * (1.0 / n);
/* ort == (n + 1) / 2.0 ; n = 9 için 5 */
```

Kontrol sorusu: 10 kovalı, zincirlemeli bir hash tablosuna 25 anahtar düzgün dağılmıştır. Yük faktörü kaçtır ve başarısız bir arama ortalama kaç düğüm yürür?

▶ **Cevap**

DERSTE Bölüm 14 · Yük faktörü Bölüm 14 · Zincirleme mi, açık adresleme mi? Bölüm 20 · Ortalama Durum (1)

≡

Kümeler ve Bağıntılar

birleşim · ayrık kümeler · eşdeğerlik sınıfları

Küme, yinelenmeyen öğelerin topluluğudur; birleşim (A ∪ B) iki kümenin bütün öğelerini, kesişim (A ∩ B) ortak öğeleri verir. Kesişimi boş olan kümelere ayrık denir. Bir bağıntı yansımalı (x ≡ x), simetrik (x ≡ y ise y ≡ x) ve geçişli (x ≡ y ve y ≡ z ise x ≡ z) ise eşdeğerlik bağıntısıdır; öğeleri birbirinden ayrık eşdeğerlik sınıflarına böler. Union-find yapısı bu sınıfları tutar; Kruskal algoritması da iki düğümün aynı bileşende olup olmadığını aynı yapıyla sorar.

```
/* 0..4 öğeleri; bildirimler: 0 ≡ 2, 3 ≡ 4, 2 ≡ 1 */
/* sınıflar: {0, 1, 2} ve {3, 4} */
int temsilci[5] = {0, 0, 0, 3, 3};

bool ayni_sinif(int x, int y) {
    return temsilci[x] == temsilci[y];
}   /* ayni_sinif(1, 0) == true, ayni_sinif(1, 4) == false */
```

Kontrol sorusu: "Aynı yıl doğmuş olmak" bir eşdeğerlik bağıntısı mıdır? "x, y'den küçüktür" bağıntısı?

▶ **Cevap**

DERSTE **Bölüm 16 · Ayrık kümeler** **Bölüm 16 · İki işlem** **Bölüm 16 · Diğer uygulamalar**

Algoritma analizine giriş

AS

Adım Sayma

program adımı · döngü sayısı

Bir algoritmanın zamanı, girdi boyutu n'nin fonksiyonu olarak kaç temel adım çalıştığıyla ölçülür. Her atama, karşılaştırma ve dönüş bir adım sayılır; döngünün koşulu gövdeden bir kez fazla denetlenir. Sabitler ayrıntıdır; önemli olan adım sayısının n ile nasıl büyüdüğüdür.

```
float sum(float list[], int n) {
    float t = 0;                 /* 1 adım      */
    for (int i = 0; i < n; i++)  /* n + 1 adım  */
        t += list[i];            /* n adım      */
    return t;                    /* 1 adım      */
}                                /* toplam 2n + 3 */
```

Kontrol sorusu: İç içe iki döngüyle m × n'lik iki matrisi toplayan bir fonksiyonun adım sayısı hangi terimle büyür?

▶ **Cevap**

DERSTE **Bölüm 1 · Zaman Karmaşıklığı** **Bölüm 1 · Adım Sayma: sum()** **Bölüm 1 · Adım Sayma: Matris Toplama**

BO

Büyük O, Ω ve Θ

üst sınır · alt sınır · sıkı sınır

f(n) = O(g(n)), yeterince büyük her n için f(n) ≤ c · g(n) olacak biçimde pozitif bir c sabitinin bulunduğunu söyler: g, f için bir üst sınırdır. Ω alt sınırı, Θ ise ikisinin birlikte geçerli olduğunu gösterir. Alışılmış sıra $1 < \lg n < n < n \lg n < n^2 < n^3 < 2^n < n!$ biçimindedir.

```
/* T(n) = 3n² + 6n + 7 için:
   n ≥ 1 iken 6n ≤ 6n² ve 7 ≤ 7n² olduğundan
   T(n) ≤ 3n² + 6n² + 7n² = 16n²  →  T(n) = O(n²)
   T(n) ≥ 3n²                     →  T(n) = Ω(n²)
   ikisi birlikte                 →  T(n) = Θ(n²) */
```

Kontrol sorusu: $2n + 3 = O(n^2)$ yazmak yanlış mıdır?

▶ **Cevap**

DERSTE **Bölüm 1 · Büyük O Gösterimi** **Bölüm 1 · Örnek: $3n^3 + 6n^2 + 7n$** **Bölüm 1 · Ω ve Θ Gösterimleri**

ED

En İyi, Ortalama, En Kötü Durum

girdiye bağlı süre · amortize maliyet

Aynı boyuttaki iki girdi farklı sürelerde işlenebilir. Doğrusal aramada aranan ilk öğedeyse tek karşılaştırma yeter (en iyi durum), hiç yoksa n karşılaştırma gerekir (en kötü durum). Ortalama durum, girdilerin dağılımına dair bir varsayım ister. Amortize analiz ise tek tek işlemlere değil bir işlem dizisinin toplam maliyetine bakar.

```
int dogrusal_ara(const int a[], int n, int x) {
    for (int i = 0; i < n; i++)
        if (a[i] == x) return i;   /* bulundu */
    return -1;                     /* yok: n karşılaştırma */
}
```

Kontrol sorusu: Aranan öğe dizide varsa ve her konumda olma olasılığı eşitse doğrusal arama ortalama kaç karşılaştırma yapar?

▶ **Cevap**

DERSTE **Bölüm 1 · En İyi, Ortalama, En Kötü** **Bölüm 1 · Amortize Analiz** **Bölüm 1 · Uygulama 2: Doğrusal Arama**

YB

Yineleme Bağıntıları

özyinelemeli algoritmaların süresi

Özyinelemeli bir algoritmanın süresi kendisi cinsinden yazılır ve açılarak çözülür. Her adımda problemi yarıya indirip sabit iş yapan algoritma T(n) = T(n/2) + c ile O(lg n) sürer (ikili arama). İki yarıyı çözüp doğrusal işle birleştiren algoritma T(n) = 2T(n/2) + cn ile O(n lg n) sürer (birleştirme sıralaması). Hanoi kulelerinde h(n) = 2h(n − 1) + 1 bağıntısı $2^n - 1$ verir.

```
int ikili_ara(const int a[], int sol, int sag, int x) {
    if (sol > sag) return -1;
    int orta = sol + (sag - sol) / 2;
    if (a[orta] == x) return orta;
    if (a[orta] < x) return ikili_ara(a, orta + 1, sag, x);
    return ikili_ara(a, sol, orta - 1, x);
}   /* T(n) = T(n/2) + c  →  O(lg n) */
```

Kontrol sorusu: T(n) = T(n − 1) + n bağıntısı (T(1) = 1) hangi büyüme sınıfındadır?

▶ **Cevap**

DERSTE Bölüm 3 · Özyinelemeli İkili Arama · Bölüm 3 · Hanoi: Kaç Hamle? · Bölüm 20 · Merge Sort: Analiz

AÖ

Alan Karmaşıklığı ve Ölçüm

ek bellek · clock() ile süre ölçme

Alan karmaşıklığı, girdinin dışında gereken ek belleği ölçer: döngüyle toplama O(1), aynı işi özyinelemeyle yapmak çağrı yığınında O(n) yer tutar. Kuramsal analiz gerçek ölçümle de sınanır: `clock()` programın harcadığı işlemci zamanını verir. Kısa işler birçok kez tekrarlanarak ölçülür, `printf` gibi yavaş giriş çıkış işlemleri ölçülen bölgenin dışında tutulur.

```
#include <time.h>
clock_t bas = clock();
for (int t = 0; t < 1000; t++)
    dogrusal_ara(a, n, -1);        /* en kötü durum */
double sn = (double)(clock() - bas) / CLOCKS_PER_SEC;
printf("arama başına %g sn\n", sn / 1000);
```

Kontrol sorusu: n büyüdükçe ölçülen süre n ikiye katlandığında yaklaşık dört katına çıkıyorsa algoritmanın büyümesi hakkında ne söylenebilir?

▶ **Cevap**

DERSTE Bölüm 1 · Alan Karmaşıklığı · Bölüm 1 · İki Ölçüm Yöntemi · Bölüm 1 · Ölçüm Sonuçları

Kod örnekleri C99'a göre yazılmıştır; `int` tipinin 4 bayt olduğu varsayılmıştır. Örneklerdeki `dugum` ve `dugum_gost` tipleri “struct ve typedef” kartında tanımlanır.

Sercan KÜLCÜ, Tüm hakları saklıdır.

EK C

C ile Veri Yapılarında Önemli İsimler

Bu dersteki yapı ve algoritmaların çoğu, onları bulan ya da yaygınlaştıran kişilerin adını taşır: AVL ağacı, Kruskal ve Prim algoritmaları, Dijkstra'nın en kısa yolu, Shell sıralaması, Floyd'un döngü tespiti. Dersin dayandığı kitaplar da C dilinin ve veri yapıları öğretiminin öncülerine aittir.

Her kartta kişinin yaşadığı yıllar, çalıştığı kurum, derste karşılığı olan katkısı ve bu katkının geçtiği slaytlara bağlantılar yer alır. Kartlar konuya göre süzülebilir, aranabilir, soyadına ya da katkı yılına göre sıralanabilir.

LE

Leonhard Euler

1707–1783 · St. Petersburg ve Berlin bilimler akademileri

Königsberg köprüleri · 1736

Königsberg'in yedi köprüsünün her birinden yalnız bir kez geçen bir yürüyüş olup olmadığını sordu. Kara parçalarını düğüm, köprüleri kenar sayarak sorunun şekle değil yalnız bağlantılara bağlı olduğunu gösterdi ve böyle bir yürüyüşün olmadığını kanıtladı. 1736 tarihli bu çalışma çizge kuramının başlangıcı kabul edilir; her düğümün derecesi çift olmadıkça kapalı bir Euler yürüyüşü kurulamaz.

DERSTE Bölüm 15 · Königsberg'in 7 Köprüsü · Bölüm 15 · Yol, Bağlantılılık, Derece

çizge kuramı · derece · Euler yürüyüşü

WH

William George Horner

1786–1837 · Kingswood School; Bath'ta kendi okulu

Horner yöntemi · 1819

Polinom köklerini yaklaşık hesaplayan bir yöntemle tanındı; yöntem ondan önce Ruffini'de de görülür. Bugün adı daha çok polinomu iç içe çarpımlarla hesaplama biçimine verilir: $a_0 + x(a_1 + x(a_2 + \dots))$ yazımıyla n dereceli bir polinom n çarpmayla hesaplanır. Derste aynı düzen iki yerde kullanılır: n boyutlu bir dizide öğenin bağıl konumunu bulurken ve bir dizgiyi `h = h * 31 + c` döngüsüyle hash'lerken.

DERSTE Bölüm 2 · n Boyutlu Genelleme · Bölüm 2 · ofset() Adım Adım · Bölüm 14 · Dizgi hash'i

polinom · satır öncelikli adres · dizgi hash'i

AC

Arthur Cayley

1821–1895 · Cambridge Üniversitesi

Etiketli ağaç sayısı · 1889

1889'da yayımladığı kısa bir notta n etiketli düğüm üzerinde n^{n-2} farklı ağaç kurulabileceğini ileri sürdü. Sonuç ilk kez Borchardt'ın 1860 tarihli çalışmasında görülür, ama bugün Cayley formülü adıyla bilinir. Aynı sayı, n düğümlü tam çizgenin kapsayan ağaç (*spanning tree*) sayısıdır: K_4 için $4^2 = 16$.

DERSTE Bölüm 17 · Tanım: Kapsayan Ağaç · Bölüm 17 · Kaç Farklı Kapsayan Ağaç?

kapsayan ağaç · tam çizge · sayma

WA

Wilhelm Ackermann

1896–1962 · Göttingen Üniversitesi (doktora); Burgsteinfurt ve Lüdenscheid liseleri

Ackermann fonksiyonu · 1928

Hilbert'in öğrencisi olan bu mantıkçı, hesaplanabilir olduğu hâlde ilkel özyinelemeli (*primitive recursive*) olmayan bir fonksiyon tanımladı. Bugün ders kitaplarında kullanılan iki değişkenli biçimi Rózsa Péter ile Raphael Robinson'un sadeleştirmesidir. Fonksiyon şaşırtıcı hızla büyür; tersi α(n) ise o kadar yavaş büyür ki ağırlıklı kural ve azaltma kuralıyla çalışan ayrık küme işlemlerinin maliyeti pratikte sabit sayılır.

DERSTE **Bölüm 16 · Ackermann ve α** **Bölüm 16 · Sınav Sorusu** **Bölüm 16 · İzleme: ackermann(1, 2)**

özyineleme · ters Ackermann · union-find

VJ

Vojtěch Jarník

1897–1970 · Prag Charles Üniversitesi

Minimum kapsayan ağaç için tek öbekli büyütme · 1930

Asıl alanı sayılar kuramı olan Çek matematikçi, Borůvka'nın sonucunu sadeleştiren bir algoritmayı 1930'da Çekçe yayımladı: ağaç tek bir düğümden başlar ve her adımda ağaçla ağaç dışını birleştiren en ucuz kenar eklenir. Algoritma Prim (1957) ve Dijkstra (1959) tarafından yeniden bulundu; bu yüzden DJP ya da Prim–Jarník algoritması adıyla da anılır.

DERSTE **Bölüm 17 · Prim: Algoritma**

minimum kapsayan ağaç · açgözlü · Prim–Jarník

OB

Otakar Borůvka

1899–1995 · Brno (Masaryk Üniversitesi)

İlk minimum kapsayan ağaç algoritması · 1926

Batı Moravya'da bir elektrik şirketinin şebeke sorusu üzerine, bilinen ilk minimum kapsayan ağaç algoritmasını 1926'da yayımladı. Her turda her bileşen kendinden çıkan en ucuz kenarı seçer; bileşen sayısı her turda en az yarıya indiği için en çok log n tur yeter. Çekçe yazılan çalışma uzun süre fark edilmedi; Sollin 1965'te yöntemi yeniden bulunca algoritma Sollin adıyla da yaygınlaştı. Turlar birbirinden bağımsız olduğu için paralel hesaplamaya uygundur.

DERSTE **Bölüm 17 · Açgözlü Yaklaşım** **Bölüm 17 · Sollin (Borůvka) ve Diğerleri**

minimum kapsayan ağaç · bileşen · paralel

JN

John von Neumann

1903–1957 · Princeton İleri Araştırmalar Enstitüsü (IAS)

Birleştirme sıralaması · 1945

1945'te EDVAC bilgisayarı için iki sıralı diziyi tek sıralı dizide birleştiren bir yordam yazdı; bu yordam bugün birleştirme sıralaması (*merge sort*) denen algoritmanın parçasıydı. El yazması, von Neumann'ın kodlarından günümüze ulaşan en eski örnektir. Algoritma böl ve fethet yaklaşımının ve en kötü durumda da O(n lg n) süren kararlı sıralamanın standart örneğidir.

DERSTE **Bölüm 20 · Birleştirme Sıralaması** **Bölüm 20 · Merge Sort: Analiz**

birleştirme · böl ve fethet · EDVAC

HC

H. S. M. Coxeter

1907–2003 · Toronto Üniversitesi

Sihirli kare kuralı · Coxeter grupları

Çok boyutlu çokyüzlüler ve yansıma grupları üzerine çalışan, yüzyılın önde gelen geometricilerinden biridir. W. W. Rouse Ball'un *Mathematical Recreations and Essays* kitabının 1939'dan sonraki baskılarını o hazırladı; ders kitabı tek n için sihirli kare kuralını ona dayandırır. Kural n^2 adımda biter; kaba kuvvetle 25! dizilişi denemek ise milyonlarca yüzyıl sürer. Derste bu fark, iyi bir algoritmanın daha hızlı bir makineden önemli olduğunu göstermek için kullanılır.

DERSTE **Bölüm 1 · Sihirli Kare Problemi** **Bölüm 1 · Daha Hızlı Makine Yetmez**

sihirli kare · kaba kuvvet · $O(n^2)$

BF

Richard Bellman ve Lester R. Ford Jr.

1920–1984 · 1927–2017 · RAND Corporation

Bellman–Ford algoritması · 1956, 1958

Ford 1956'da, Bellman 1958'de negatif ağırlıklı kenarlara izin veren tek kaynaklı en kısa yol yöntemini yayımladı; aynı fikir Shimbel'de (1955) ve Moore'da da görülür. Bütün kenarlar n − 1 tur gevşetilir; n. turda hâlâ gevşeyen bir kenar varsa çizgede negatif döngü vardır. Bellman dinamik programlamanın, Ford ise Fulkerson'la birlikte en büyük akış kuramının kurucularındandır.

DERSTE **Bölüm 18 · Negatif Kenar Sorunu** **Bölüm 18 · Bellman-Ford Adım Adım**

en kısa yol · gevşetme · negatif döngü

RP

Robert C. Prim

1921–2021 · Bell Laboratuvarları

Prim algoritması · 1957

Bell Laboratuvarları'nda Kruskal'la aynı dönemde çalışırken minimum kapsayan ağaç için tek öbeği büyüten algoritmayı 1957'de yayımladı; Jarník'in 1930 tarihli yöntemini bağımsız olarak yeniden buldu. Ağaçtaki ve ağaç dışındaki düğümleri birleştiren en ucuz kenar seçildiği için döngü denetimi gerekmez. Dizi ile $O(n^2)$ süren algoritma, heap ile $O(e \log n)$'ye iner ve seyrek çizgelerde daha hızlıdır.

DERSTE **Bölüm 17 · Prim: Algoritma** **Bölüm 17 · Prim Adım Adım** **Bölüm 17 · Prim: Heap ile (ek)**

minimum kapsayan ağaç · açgözlü · heap

AL

Georgy Adelson-Velsky ve Evgenii Landis

1922–2014 · 1921–1997 · Moskova (ITEP, Moskova Devlet Üniversitesi)

AVL ağacı · 1962

“Bilginin düzenlenmesi için bir algoritma” başlıklı 1962 makalelerinde, her düğümde iki alt ağacın yükseklik farkını en çok 1 tutan ilk kendini dengeleyen ikili arama ağacını tanımladılar; yapı adını ikisinin soyadlarının baş harflerinden alır. Denge bozulduğunda tek ya da çift döndürme yapılır ve yükseklik ≈ 1,44 lg n'yi aşmaz. Adelson-Velsky ayrıca Kaissa satranç programının yazarlarındandır; Landis'in asıl alanı kısmi türevli denklemlerdir.

DERSTE **Bölüm 11 · AVL Tanımı** **Bölüm 11 · Tek Döndürme** **Bölüm 11 · Çift Döndürme**

AVL ağacı · denge faktörü · döndürme

AP

Alan J. Perlis

1922–1990 · Carnegie Tech, Yale Üniversitesi

Kılavuzlu ağaçlar · 1960

ALGOL'ün tasarımcılarından biridir ve 1966'da ilk Turing Ödülü'nü aldı. C. Thornton'la birlikte 1960'ta, bir ikili ağaçtaki boş işaretçilerin kök ortada sıradaki öncülü ve ardılı göstermek için kullanılabileceğini önerdi; bu işaretçilere kılavuz (*thread*) denir. Kılavuzlu ağaçta kök ortada gezinme yığın ya da özyineleme olmadan yapılır. Derste sık alıntılanan kısa vecizeleri de 1982'de yayımlanan "Epigrams on Programming" yazısından gelir.

DERSTE **Bölüm 12 · Perlis ve Thornton** **Bölüm 12 · Boş İşaretçiler** **Bölüm 11 · Ek Parametre Mevzusu**

kılavuz · kök ortada gezinme · ALGOL

DS

Donald L. Shell

1924–2015 · General Electric; Cincinnati Üniversitesi (doktora)

Shell sıralaması · 1959

"A High-Speed Sorting Procedure" başlıklı makalesini 1959'da Communications of the ACM'de yayımladı. Eklemeli sıralama birbirinden h uzaklıktaki öğelere uygulanır ve h küçültülerek 1'e indirilir; uzak takaslar ters çiftleri hızla azalttığı için komşu takaslı algoritmaların $\Omega(n^2)$ sınırı aşılır. Süre aralık dizisine bağlıdır ve uzak takaslar yüzünden sıralama kararlı değildir.

DERSTE **Bölüm 19 · Shell Sıralama** **Bölüm 19 · shell_sort() Adım Adım** **Bölüm 19 · Ters Çift Sayısı**

aralık dizisi · eklemeli sıralama · ters çift

JK

Joseph B. Kruskal

1928–2010 · Princeton (doktora); Bell Laboratuvarları

Kruskal algoritması · 1956

Minimum kapsayan ağacı bulmak için kenarları maliyete göre sıralayıp döngü oluşturmayan her kenarı ağaca ekleyen yöntemi 1956'da yayımladı. Seçilen kenarlar önce bir orman oluşturur, sonunda tek ağaçta birleşir. Döngü denetimi ayrık kümelerle yapılır: kenarın iki ucu farklı kümelerdeyse kenar alınır ve kümeler birleştirilir; toplam süre O(e log e) olur. İstatistikte çok boyutlu ölçekleme çalışmalarıyla da tanınır.

DERSTE **Bölüm 17 · Kruskal: Algoritma** **Bölüm 17 · Kruskal Nasıl Kodlanır?** **Bölüm 16 · Diğer uygulamalar**

minimum kapsayan ağaç · union-find · açgözlü

ED

Edsger W. Dijkstra

1930–2002 · Mathematisch Centrum Amsterdam; Eindhoven; Texas Üniversitesi, Austin

En kısa yol algoritması · 1959

1959'da yayımladığı iki sayfalık notta hem tek kaynaklı en kısa yol algoritmasını hem de minimum kapsayan ağaç için Jarník–Prim yöntemini verdi. En kısa yol algoritması açgözlüdür: uzaklığı en küçük olan düğüm kesinleşir ve komşularının uzaklıkları onun üzerinden güncellenir. Bu seçim yalnız negatif olmayan kenar ağırlıklarında doğrudur. Yapısal programlamanın öncülerinden olan Dijkstra 1972'de Turing Ödülü'nü aldı.

DERSTE **Bölüm 18 · Açgözlü Fikir** **Bölüm 18 · Dijkstra Adım Adım** **Bölüm 17 · Prim: Algoritma**

en kısa yol · açgözlü · öncelikli kuyruk

TH

C. A. R. (Tony) Hoare

1934–2026 · Elliott Brothers; Queen's University Belfast; Oxford Üniversitesi; Microsoft Research

Hızlı sıralama · 1961–1962

Hızlı sıralamayı (*quicksort*) 1959'da Moskova'da bir sözlük çevirisi üzerinde çalışırken tasarladı; algoritmayı 1961'de Communications of the ACM'de, ayrıntılı çözümlemesiyle 1962'de The Computer Journal'da yayımladı. Dayanak öğeye göre bölümleme ve iki parçanın özyinelemeli sıralanması, ortalamada O(n lg n) süren yerinde bir sıralama verir. Program doğruluğu için Hoare mantığını da geliştirdi ve 1980'de Turing Ödülü'nü aldı. 5 Mart 2026'da Cambridge'de öldü.

DERSTE **Bölüm 20 · Hızlı Sıralama** **Bölüm 20 · Bölümleme Değişmezi** **Bölüm 20 · En Kötü ve En İyi Durum**

quicksort · bölümleme · böl ve fethet

SW

Stephen Warshall

1935–2006 · Technical Operations; Massachusetts Computer Associates

Geçişli kapanış · 1962

1962'de bir Boole matrisinin geçişli kapanışını $O(n^3)$ sürede bulan algoritmayı yayımladı: her k için “i'den k'ye ve k'den j'ye yol varsa i'den j'ye de yol vardır” kuralı bütün çiftlere uygulanır. Floyd aynı yıl bu düzeni yol uzunluklarına taşıdı; tüm çiftler arasındaki en kısa yolları bulan algoritma bu yüzden Floyd–Warshall adıyla anılır.

DERSTE **Bölüm 18 · Floyd-Warshall Adım Adım** **Bölüm 18 · Hangi Algoritma?**

geçişli kapanış · tüm çiftler · $O(n^3)$

RF

Robert W. Floyd

1936–2001 · Carnegie Tech; Stanford Üniversitesi

Floyd–Warshall · döngü tespiti

1962'de tüm düğüm çiftleri arasındaki en kısa yolları bulan Floyd–Warshall algoritmasını yayımladı. Bağlı listede iki işaretçiyle (biri bir, öteki iki adım ilerleyen) döngü bulan “tavşan ve kaplumbağa” yöntemi de Knuth'un aktarımıyla ona atfedilir; yöntem O(1) ek bellekle çalışır. 1964'te heap'i aşağıdan yukarı O(n) sürede kuran yöntemi verdi. Program doğrulamasına katkılarıyla 1978'de Turing Ödülü'nü aldı.

DERSTE **Bölüm 4 · Döngü Tespiti: Floyd** **Bölüm 4 · Floyd Adım Adım** **Bölüm 18 · Floyd-Warshall Adım Adım** **Bölüm 13 · Aşağıdan yukarı heap kurma**

tavşan ve kaplumbağa · tüm çiftler · heap kurma

DK

Donald E. Knuth

d. 1938 · Stanford Üniversitesi

The Art of Computer Programming · 1968'den bu yana

İlk cildi 1968'de çıkan *The Art of Computer Programming* dizisiyle algoritmaların matematiksel çözümlemesini bir disiplin hâline getirdi; 1974'te Turing Ödülü'nü aldı. Derste birçok sonuç ona dayanır: doğrusal yoklamada beklenen yoklama sayıları, çarpma yönteminde önerilen $A \approx (\sqrt{5}-1)/2$ sabiti, Shell sıralaması için 1, 4, 13, 40, … aralık dizisi ve Floyd'un döngü tespitinin aktarımı. Knuth–Morris–Pratt dizgi arama algoritmasının da ortak yazarıdır. Sedgewick onun doktora öğrencisidir.

DERSTE **Bölüm 14 · Yük faktörü** **Bölüm 14 · Çarpma yöntemi** **Bölüm 19 · Shell Sıralama** **Bölüm 4 · Döngü Tespiti: Floyd**

TAOCP · algoritma analizi · hash

BM

Rudolf Bayer ve Edward M. McCreight

d. 1939 (Bayer) · Boeing Bilimsel Araştırma Laboratuvarları; TU München (Bayer); Xerox PARC (McCreight)

B-ağacı · 1970–1972

Boeing'de birlikte çalışırken büyük sıralı dizinleri diskte tutmak için B-ağacını tasarladılar; makale 1970'te rapor olarak dağıtıldı, 1972'de Acta Informatica'da yayımlandı. m yollu bu dengeli ağaçta her düğüm birçok anahtar tutar ve bütün yapraklar aynı düzeydedir; dolan düğüm ortadaki anahtarı ebeveynine göndererek ikiye bölünür. Bayer'in 1972'de tanımladığı simetrik ikili B-ağaçları, kırmızı-siyah ağaçların öncüsüdür.

DERSTE **Bölüm 11 · Kırmızı-Siyah ve B-Ağacı** **Bölüm 11 · Arama Ağaçları Karşılaştırması**

B-ağacı · disk · veri tabanı dizini

RT

Robert E. Tarjan

d. 1948 · Stanford (doktora); Princeton Üniversitesi

Ayrık kümelerin çözümlemesi · 1975, 1984

1975'te birleştirme kuralıyla yol sıkıştırmanın birlikte kullanıldığı ayrık küme işlemlerinin ters Ackermann fonksiyonu α ile sınırlandığını kanıtladı. Jan van Leeuwen'le (d. 1946, Utrecht Üniversitesi) 1984'te yayımladığı çalışma, bu türden birleştirme ve bulma yöntemlerinin en kötü durum çözümlemesini bir arada verdi. Sleator'la birlikte splay ağaçlarını, Fredman'la birlikte Fibonacci heap'lerini tanımladı; 1986'da Turing Ödülü'nü aldı.

DERSTE **Bölüm 16 · Tarjan–van Leeuwen** **Bölüm 11 · Dengeli Ağaçlar** **Bölüm 13 · İleri düzey heap'ler**

union-find · splay ağacı · Fibonacci heap

DR

Dennis M. Ritchie

1941–2011 · Bell Laboratuvarları

C dili · Unix

C dilini 1972 dolayında Bell Laboratuvarları'nda Unix işletim sistemini yazmak için geliştirdi; Unix'i Ken Thompson'la birlikte tasarladı ve 1983'te onunla Turing Ödülü'nü paylaştı. Dersteki her yapı C'nin sunduğu araçlarla kurulur: `struct` ile kayıt, işaretçiyle bağ, `malloc` ve `free` ile çalışma anında bellek. Kernighan'la yazdığı *The C Programming Language* (1978) dersin başucu kitabıdır.

DERSTE **Bölüm 1 · Ders Kitapları** **Bölüm 2 · Dizi ve İşaretçi** **Bölüm 2 · struct**

C dili · Unix · K&R

BK

Brian W. Kernighan

d. 1942 · Bell Laboratuvarları; Princeton Üniversitesi

The C Programming Language · 1978

Ritchie'yle yazdığı *The C Programming Language* (1978; ANSI C'ye göre ikinci baskı 1988), yazarlarının baş harfleriyle "K&R" diye anılır ve C'nin fiilî tanımı oldu. Kitap kısa, okunaklı örnekleriyle bir programlama dili kitabının nasıl yazılacağına da örnek oldu. AWK dilinin ortak yazarıdır (adındaki K onundur); 2000'den bu yana Princeton'da ders vermektedir.

DERSTE **Bölüm 1 · Ders Kitapları** **Bölüm 2 · Dizgi = char Dizisi**

K&R · C dili · AWK

EH

Ellis Horowitz

d. 1944 · University of Southern California (USC)

Fundamentals of Data Structures · 1976

Sahni'yle birlikte yazdığı *Fundamentals of Data Structures* (1976) ve *Fundamentals of Computer Algorithms* (1978) uzun yıllar dünyada en çok okutulan ders kitapları arasında yer aldı. Kitabın C sürümü *Fundamentals of Data Structures in C*, Anderson-Freed'in katılımıyla 1993'te çıktı ve bu dersin ana kitabıdır; slaytlardaki "Horowitz 2.2" gibi göndermeler bu kitabın bölümlerini gösterir. USC'de bilgisayar bilimleri bölüm başkanlığını 1990–1999 arasında yürüttü.

DERSTE Bölüm 1 · Ders Kitapları · Bölüm 1 · Soyut Veri Tipi (ADT) · Bölüm 4 · Kullanılabilir Düğüm Listesi

ders kitabı · ADT · C

SS

Sartaj Sahni

d. 1949 · Cornell (doktora); Florida Üniversitesi

Horowitz–Sahni kitapları

Horowitz'le birlikte veri yapıları ve algoritmalar kitaplarını yazdı; ders kitabının sistem yaşam döngüsü, algoritmanın tanımı ve ADT anlatımı bu ortak çalışmadan gelir. Florida Üniversitesi'nde ayırt edici profesör (*distinguished professor*) olarak paralel hesaplama, NP-tamlık ve yaklaşım algoritmaları üzerine çalıştı. 1997'de IEEE Taylor L. Booth Eğitim Ödülü'nü aldı.

DERSTE Bölüm 1 · Ders Kitapları · Bölüm 1 · Sistem Yaşam Döngüsü · Bölüm 1 · Algoritma Nedir?

ders kitabı · algoritma · paralel hesaplama

SA

Susan Anderson-Freed

ö. 2012 · Illinois Wesleyan University

Fundamentals of Data Structures in C · 1993

Horowitz ve Sahni'nin kitabını C diline uyarlayan üçüncü yazardır; 1993'te çıkan *Fundamentals of Data Structures in C* beş dile çevrildi. Illinois Wesleyan University'de sosyoloji öğretim üyesi olarak başlayıp programlama dersleri verdi ve 1986'da üniversitenin bilgisayar bilimleri bölümünü kurarak ilk başkanı oldu. 4 Kasım 2012'de öldü.

DERSTE Bölüm 1 · Ders Kitapları

ders kitabı · C · eğitim

RS

Robert Sedgewick

d. 1946 · Stanford (doktora); Brown; Princeton Üniversitesi

Hızlı sıralamanın çözümlemesi · Algorithms in C

Knuth'un danışmanlığında 1975'te hızlı sıralama üzerine doktora tezini yazdı; medyan-3 ve küçük parçaları eklemeli sıralamayla birtirme gibi iyileştirmelerin çözümlemesi bu çalışmalardan gelir. *Algorithms in C* dersin ek kaynaklarındandır; hızlı sıralama bölümündeki kod bu kitaptan alınmıştır. Guibas'la birlikte kırmızı-siyah ağaçları tanımladı ve Princeton'da bilgisayar bilimleri bölümünün kurucu başkanı oldu.

DERSTE Bölüm 1 · Ders Kitapları · Bölüm 20 · Hızlı Sıralama · Bölüm 20 · Medyan-3 ve Eşik M

quicksort · Algorithms in C · kırmızı-siyah ağaç

HW

H. C. von Warnsdorff

Des Rösselsprunges einfachste und allgemeinste Lösung (1823)

At gezisi için Warnsdorff kuralı · 1823

1823'te at gezisi için bilinen en eski sezgisel kuralı önerdi: at her adımda, oradan sonra gidilebilecek boş kare sayısı en az olan kareye gider. Önce en sıkışık kareleri bitiren bu sıra, 8×8 tahtada çoğunlukla hiç geri dönmeden tam bir gezi bulur. Aynı problemin düz geri izlemeyle çözümü ise çok daha uzun sürer. Kural, geri izlemede adayları iyi sıralamanın ne kadar iş kazandırdığını gösteren klasik örnektir.

DERSTE **Bölüm 3 · Satranç Tahtasında At Gezisi** **Bölüm 3 · Warnsdorff Kuralı**

at gezisi · geri izleme · sezgisel kural

JŁ

Jan Łukasiewicz

1878–1956 · Lviv ve Varşova üniversiteleri

Polonya (önek) gösterimi · 1924

Polonyalı mantıkçıdır. Mantık formüllerini parantezsiz yazmak için işleci işlenenlerinden önce koyan önek gösterimini 1924'te buldu ve 1929'dan başlayarak yayımlarında kullandı. Gösterim bu yüzden Polonya gösterimi diye anılır; işlecin sona yazıldığı sonek (*postfix*) biçimine de ters Polonya gösterimi denir. Sonek ifade, parantez ve öncelik kuralı gerektirmeden tek bir yığınla soldan sağa hesaplanır.

DERSTE **Bölüm 8 · Üç Gösterim** **Bölüm 8 · İfade Değerlendirme** **Bölüm 8 · Prefix Hesaplama**

önek gösterim · sonek gösterim · ifade değerlendirme

HS

Harold H. Seward

MIT Dijital Bilgisayar Laboratuvarı (yüksek lisans tezi, 1954)

Saymalı sıralama · 1954

1954'te MIT'deki yüksek lisans tezinde saymalı sıralamayı (*counting sort*) tanımladı ve onu taban sıralamasının (*radix sort*) her geçişinde kullanmayı önerdi. Algoritma öğeleri birbiriyle karşılaştırmaz: önce her değerin kaç kez geçtiğini sayar, sonra bu sayıların önek toplamlarıyla her öğenin çıktıdaki yerini bulur. Aralığı 0…k olan n tam sayıyı O(n + k) sürede ve kararlı biçimde sıralar.

DERSTE **Bölüm 19 · counting_sort(): Sayma** **Bölüm 20 · radixsort(): Birler Geçişi**

saymalı sıralama · taban sıralaması · karşılaştırmasız sıralama

JM

John McCarthy

1927–2011 · MIT, Stanford Üniversitesi

LISP · 1958

LISP dilini tasarladı; dil 1958'de ortaya çıktı ve 1960'ta *Communications of the ACM*'de yayımlandı. LISP'te veriler, öğeleri atom ya da başka bir liste olabilen genelleştirilmiş listelerle gösterilir; listenin başı `car`, kuyruğu `cdr` işlemiyle alınır. Artık erişilemeyen liste düğümlerinin belleğini geri kazanan çöp toplamayı da LISP için buldu. Yapay zekâ terimini ortaya atan kişi olarak bilinir ve 1971'de Turing Ödülü'nü aldı.

DERSTE **Bölüm 9 · Genelleştirilmiş Liste**

LISP · genelleştirilmiş liste · çöp toplama

AK

Arthur B. Kahn

Communications of the ACM 5(11), 1962

Topolojik sıralama · 1962

"Topological sorting of large networks" (1962) makalesinde, giriş derecesi sıfır olan düğümleri art arda çıkararak yönlü ve döngüsüz bir çizgenin topolojik sırasını bulan algoritmayı verdi. Çıkarılan her düğümün ardıllarının giriş derecesi bir azaltılır; derecesi sıfıra inen düğüm bekleyenlere katılır. Bekleyenler kuyrukta tutulduğunda yönteme Kahn algoritması denir. Algoritma O(n + e) sürede çalışır; bütün düğümler çıkarılamıyorsa çizgede döngü vardır.

DERSTE Bölüm 18 · Topolojik Sıra · Bölüm 18 · Topolojik Sıra Bulmak · Bölüm 18 · Topolojik Sıra: Analiz

topolojik sıra · giriş derecesi · AOV ağı

JW

J. W. J. Williams

1930–2012 · Elliott Brothers (Londra); Bell-Northern Research (Ottawa)

İkili heap ve heap sort · 1964

İkili heap'i ve heap sort'u 1963'te Elliott Brothers'ta çalışırken buldu; algoritmayı 1964'te "Algorithm 232: Heapsort" adıyla *Communications of the ACM*'de yayımladı. Heap, tam ikili ağacı bir dizide boşluksuz saklar; ebeveyn ve çocuklar indis aritmetiğiyle bulunur. Heap sort, n öğeyi ek dizi kullanmadan en kötü durumda O(n lg n) sürede sıralar. Floyd aynı yıl heap'i aşağıdan yukarı O(n) sürede kuran yöntemi ekledi.

DERSTE Bölüm 13 · Maksimum ve minimum heap · Bölüm 20 · Heap Sort · Bölüm 20 · Heap Sort: Analiz

ikili heap · heap sort · öncelikli kuyruk

JM

Joseph M. Morris

Information Processing Letters 9, 1979

Morris gezinmesi · 1979

"Traversing binary trees simply and cheaply" (1979) makalesinde ikili ağacı yığın, özyineleme ya da bayrak alanı kullanmadan kök ortada gezmenin yolunu verdi. Bir düğümün kök ortadaki öncülünün boş sağ işaretçisine geçici bir kılavuz yazılır; gezinme bu kılavuzla geri döner ve kılavuzu siler. Ağaç gezinme sırasında geçici olarak değişir, sonunda eski hâline döner. Ek bellek O(1), süre O(n)'dir.

DERSTE Bölüm 12 · Morris Gezinmesi · Bölüm 12 · Morris Adım Adım

Morris gezinmesi · geçici kılavuz · O(1) ek bellek

CLRS

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest ve Clifford Stein

Dartmouth College, MIT, Columbia Üniversitesi

Introduction to Algorithms · 1990

Baş harflerinden dolayı CLRS diye anılan *Introduction to Algorithms* kitabının yazarlarıdır. İlk baskı 1990'da Cormen, Leiserson ve Rivest imzasıyla çıktı; Stein 2001'deki ikinci baskıda yazarlara katıldı. Kitap dersin ek kaynaklarındandır: asimptotik gösterim, yineleme bağıntıları, heap, hash tabloları, çizge algoritmaları ve karşılaştırmalı sıralamanın alt sınırı burada kanıtlarıyla verilir. Rivest, RSA şifreleme sistemi için 2002 Turing Ödülü'nü Shamir ve Adleman ile paylaştı.

DERSTE Bölüm 1 · Ders Kitapları

CLRS · algoritma analizi · ders kitabı

DM

David R. Musser

Rensselaer Polytechnic Institute

Introsort · 1997

"Introspective Sorting and Selection Algorithms" (1997) makalesinde introsort'u tanıttı. Algoritma hızlı sıralamayla başlar ve özyineleme derinliğini izler; derinlik yaklaşık 2⌊lg n⌋'yi aşınca o parçayı heap sort'la bitirir. Böylece hızlı sıralamanın ortalama hızı korunur, en kötü durum da O(n lg n)'de kalır. Yöntem C++ standart kitaplığı için geliştirildi ve std::sort gerçeklemelerinde yaygındır. Musser, Alexander Stepanov ile birlikte genel programlama ve STL üzerinde çalıştı.

DERSTE **Bölüm 20 · Hibrit Sıralamalar**

introsort · hızlı sıralama · heap sort

TP

Tim Peters

Python çekirdek geliştiricisi

Timsort · 2002

2002'de Python için Timsort'u yazdı. Algoritma veride hazır bulunan sıralı bölümleri (koşu, *run*) bulur, kısa koşuları eklemeli sıralamayla uzatır ve koşuları birleştirme sıralamasıyla birleştirir. Kararlıdır; en kötü durumda O(n lg n), zaten sıralı veride O(n) sürer. Python 2.3'ten 3.11'e kadar dilin standart sıralaması oldu; Java SE 7 (nesne dizileri için), Android ve V8 de Timsort'u benimsedi.

DERSTE **Bölüm 20 · Hibrit Sıralamalar**

Timsort · kararlı sıralama · koşu

Yıllar ve katkılar özgün yayınlara ve ansiklopedik kaynaklara dayanır. "d." doğum yılını gösterir ve yaşayan kişiler için kullanılır; "ö." yalnız ölüm yılı bilinen kişiyi gösterir. Edward McCreight'ın doğum yılı yayımlanmış kaynaklarda bulunamadığı için verilmemiştir. Yaşam yılları güvenilir kaynaklarda bulunamayan ya da ortak kartta toplanan kişilerde yalnız katkının yılı ve yayımlandığı yer verilmiştir.

Sercan KÜLCÜ, Tüm hakları saklıdır.