DERS KİTABI

Veri Yapıları

Sercan Külcü

Giresun Üniversitesi · Bilgisayar Mühendisliği Bölümü

Veri Yapıları

Sercan Külcü
Giresun Üniversitesi, Bilgisayar Mühendisliği Bölümü
Giresun, Türkiye

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

Sunumlar, çalışma soruları ve simülatörler o sayfadadır. Bölümlerin kodu `github.com/sercankulcu/data-structures-java` deposundadır.

© Sercan Külcü. Tüm hakları saklıdır.

Notların 14 Kasım 2026 tarihli hâlinden üretilmiştir.

Önsöz

Veri yapıları, verinin bellekte nasıl düzenleneceğini ve ona nasıl erişileceğini belirler; doğru yapıyı seçmek arama, ekleme, silme ve gezinme gibi işlemlerin hızını doğrudan etkiler. Bu kitap, Giresun Üniversitesi Bilgisayar Mühendisliği Bölümü'nde verilen Veri Yapıları dersinin notlarından oluşur ve bölümleri dersteki sırayla izler. Örnekler Java ile yazılmıştır.

Kitap dört kısımdır. Kısımlar sırasıyla şunlardır: **Temeller** (Bölüm 1–4), **Doğrusal Yapılar** (Bölüm 5–8), **Ağaçlar** (Bölüm 9–11) ve **Eşlemeler, Kümeler ve Çizgeler** (Bölüm 12–14).

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 işlemin yapıyı adım adım nasıl değiştirdiğini 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.

Bölümlerin kodu derse eşlik eden `github.com/sercankulcu/data-structures-java` deposunda, her bölüm için ayrı bir klasördedir; klasörün adı bölümün başında yazılıdır. Sunumlar, çalışma soruları ve yapıları adım adım çalıştıran simülatörler ders sayfasındadır.

Sercan Külcü
Giresun

İçindekiler

Dersin Haritası 1

KISIM I · TEMELLER

1 Giriş 4

1.1 Dijital veri 5
1.2 Veri yapıları ve algoritmalar 10
1.3 Veri tipleri 13
1.4 Java bellek modeli 17
1.5 Jenerikler 23
1.6 İlkel olmayan veri yapıları 27
1.7 Soyut veri tipleri ve veri yapıları 30
1.8 Verilerin düzenlenmesi ve yapı seçimi 37
1.9 Temel veri yapılarına genel bakış 41
1.10 Sık yapılan hatalar 42
1.11 Özet 43
1.12 Kavram listesi 44
1.13 Kendini Dene 45
1.14 İleri okuma 47
1.15 Kaynaklar 47

2 Algoritma Analizi 48

2.1 Algoritmalar ve ölçüm 48
2.2 Asimptotik gösterim 58
2.3 Koddan karmaşıklığa 68
2.4 Alan, doğruluk ve sağlamlık 85
2.5 Sık yapılan hatalar 91
2.6 Özet 92
2.7 Kavram listesi 93
2.8 Kendini Dene 94
2.9 İleri okuma 96
2.10 Kaynaklar 96

3 Diziler 98

3.1 Dizi kavramı 99
3.2 Java'da diziler 104
3.3 Dizi algoritmaları 118
3.4 Çok boyutlu diziler 131
3.5 Dizi işlemlerinin karmaşıklığı 146
3.6 Sık yapılan hatalar 160
3.7 Özet 161
3.8 Kavram listesi 162

3.9 Kendini Dene 163
3.10 İleri okuma 165
3.11 Kaynaklar 165

4 Özyineleme 166

4.1 Özyineleme nedir? 167
4.2 Çağrı yığını 173
4.3 Doğrusal özyineleme 179
4.4 İkili özyineleme 187
4.5 Hanoi kuleleri ve geri izleme 196
4.6 Yineleme bağıntıları 206
4.7 Kuyruk özyinelemesi 211
4.8 Bağlı listede özyineleme 216
4.9 Sık yapılan hatalar 220
4.10 Özet 221
4.11 Kavram listesi 222
4.12 Kendini Dene 222
4.13 İleri okuma 224
4.14 Kaynaklar 225

KISIM II · DOĞRUSAL YAPILAR

5 Bağlı Listeler 227

5.1 Temeller 228
5.2 Tek yönlü liste: gezinme ve ekleme 237
5.3 Tek yönlü liste: silme ve diğer işlemler 248
5.4 Çift yönlü bağlı liste 264
5.5 Dairesel bağlı liste 280
5.6 Dizi mi, bağlı liste mi? 290
5.7 Sık yapılan hatalar 292
5.8 Özet 293
5.9 Kavram listesi 294
5.10 Kendini Dene 294
5.11 İleri okuma 296
5.12 Kaynaklar 297

6 Liste ADT ve Java Koleksiyonları 298

6.1 Koleksiyon arayüzleri 299
6.2 Liste ADT'si 306
6.3 List metotları 312
6.4 Gezinme 324
6.5 ArrayList 335
6.6 LinkedList ve konumsal liste 344
6.7 Öteki liste sınıfları 353
6.8 Özet 360
6.9 Kavram listesi 361

6.10 Kendini Dene 362
6.11 İleri okuma 364
6.12 Kaynaklar 364

7 Yığınlar 365

7.1 Yığın ADT'si 366
7.2 Gerçekleme 372
7.3 Yığın uygulamaları 386
7.4 Hanoi kuleleri ve stock span 401
7.5 İfade gösterimleri 407
7.6 Sık yapılan hatalar ve ortak desen 420
7.7 Özet 421
7.8 Kavram listesi 422
7.9 Kendini Dene 422
7.10 İleri okuma 425
7.11 Kaynaklar 425

8 Kuyruklar ve Deque 426

8.1 Kuyruk ADT'si 427
8.2 Dizi ile kuyruk 434
8.3 Bağlı liste ile kuyruk 450
8.4 Kuyruk türleri ve çift uçlu kuyruk 457
8.5 Kuyruk uygulamaları 468
8.6 Sık yapılan hatalar 478
8.7 Özet 479
8.8 Kavram listesi 480
8.9 Kendini Dene 480
8.10 İleri okuma 482
8.11 Kaynaklar 483

KISIM III · AĞAÇLAR

9 Ağaçlar 485

9.1 Ağaç kavramı 486
9.2 İkili ağaçlar 503
9.3 Gösterimler 511
9.4 Ağaç gezinmeleri 520
9.5 Gezinmelerin kodu 528
9.6 Gezinmelerin uygulamaları 542
9.7 Sık yapılan hatalar 553
9.8 Özet 554
9.9 Kavram listesi 555
9.10 Kendini Dene 555
9.11 İleri okuma 558
9.12 Kaynaklar 558

10 Arama Ağaçları 559

10.1 Arama ağaçları 560
10.2 İkili arama ağacı 563
10.3 BST işlemleri 571
10.4 Denge ve AVL ağaçları 595
10.5 Kırmızı-siyah ağaçlar 612
10.6 Splay ağaçları 622
10.7 Çok yollu arama ağaçları 627
10.8 Sık yapılan hatalar 634
10.9 Özet 634
10.10 Kavram listesi 635
10.11 Kendini Dene 636
10.12 İleri okuma 638
10.13 Kaynaklar 638

11 Öncelikli Kuyruklar ve Heap 640

11.1 Öncelikli kuyruk ADT'si 641
11.2 Liste ile gerçekleme 646
11.3 İkili heap 655
11.4 Ekleme ve silme 664
11.5 Heap oluşturma ve heap sort 679
11.6 Java ve uygulamalar 688
11.7 Sık yapılan hatalar 693
11.8 Özet 694
11.9 Kavram listesi 695
11.10 Kendini Dene 696
11.11 İleri okuma 698
11.12 Kaynaklar 698

KISIM IV · EŞLEMELER, KÜMELER VE ÇİZGELER

12 Eşlemeler ve Hash Tabloları 700

12.1 Eşleme ve hash fikri 701
12.2 Hash fonksiyonları 708
12.3 Ayrı zincirleme 719
12.4 Açık adresleme 732
12.5 Performans ve uygulamalar 744
12.6 Sık yapılan hatalar 756
12.7 Özet 757
12.8 Kavram listesi 758
12.9 Kendini Dene 759
12.10 İleri okuma 761
12.11 Kaynaklar 761

13 Kümeler ve Ayrık Kümeler 762

13.1 Küme ADT'si 763
13.2 Küme gerçeklemeleri 769

13.3 Çoklu küme ve çoklu eşleme 783
13.4 Ayrık kümeler ve Union-Find 789
13.5 Union-Find'ı hızlandırmak 798
13.6 Sık yapılan hatalar 814
13.7 Özet 815
13.8 Kavram listesi 816
13.9 Kendini Dene 817
13.10 İleri okuma 819
13.11 Kaynaklar 819

14 Çizgeler 821

14.1 Temel kavramlar 822
14.2 Çizge gösterimleri 833
14.3 Gezinme: BFS ve DFS 845
14.4 En kısa yol 868
14.5 Minimum kapsayan ağaç 881
14.6 Özet 894
14.7 Kavram listesi 895
14.8 Kendini Dene 896
14.9 İleri okuma 898
14.10 Kaynaklar 898

EKLER

Ek A Terim Sözlüğü 899

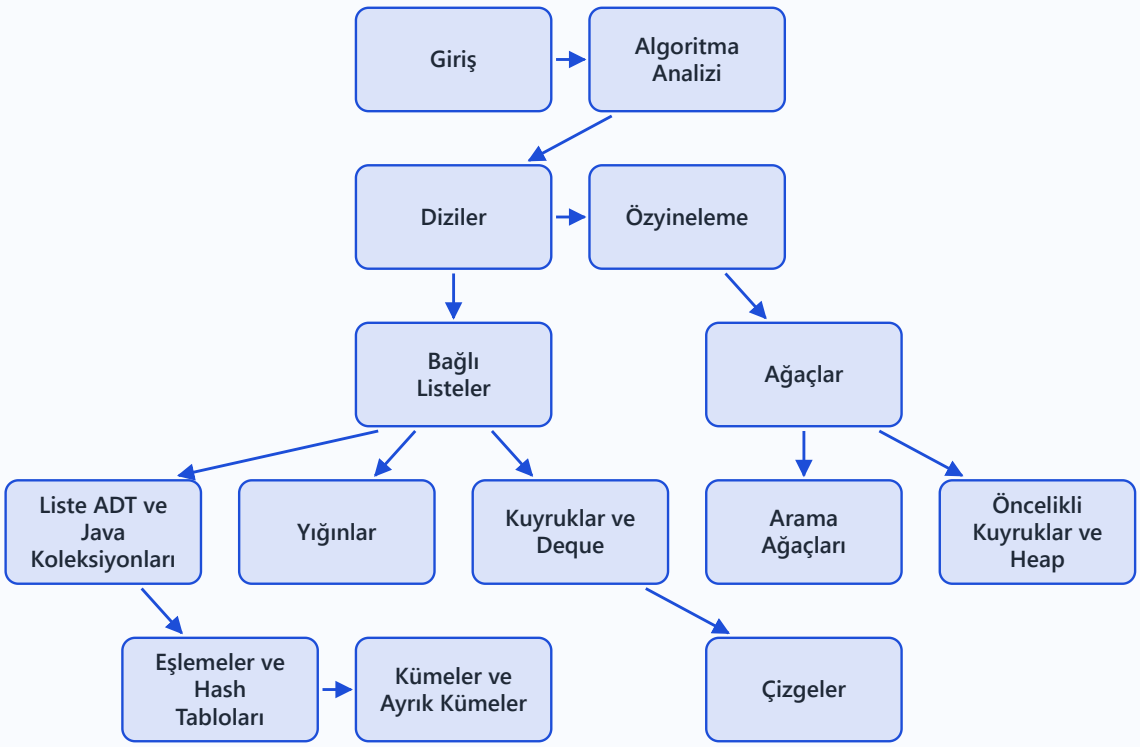
Ek B Veri Yapıları İçin Ön Bilgiler 904

Ek C Veri Yapılarında Önemli İsimler 910

VERİ YAPILARI

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. Diziden başlayan yol bağlı yapılara, ağaçlara ve çizgelere ayrılır.

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

Konu (bölüm)	Dayandığı konu	Bağ
Giriş (1)	—	Verinin bellekte temsili, ilkel ve referans tipler, soyut veri tipi; bütün yapılar bu dille anlatılır.
Algoritma Analizi (2)	Giriş	Bir yapıyı seçmek için işlemlerin maliyetini ölçmek gerekir: Büyük O, Omega ve Teta.

Konu (bölüm)	Dayandığı konu	Bağ
Diziler (3)	Algoritma Analizi	İlk yapı: ardışık bellek ve indisle O(1) erişim; ekleme ve silmenin bedeli analizle görülür.
Özyineleme (4)	Algoritma Analizi, Diziler	Problemi daha küçük örneklerine indirmek; çağrı yığını ve yineleme bağıntısıyla analiz.
Bağlı Listeler (5)	Diziler	Dizinin sabit boyu ve kaydırma maliyeti düğümlerle aşılır: başa ekleme O(1), rastgele erişim yok.
Liste ADT ve Java Koleksiyonları (6)	Diziler, Bağlı Listeler	Dizi ve bağlı liste aynı Liste ADT'sinin iki gerçeklemesidir; Java koleksiyonları ve Iterator.
Yığınlar (7)	Diziler, Bağlı Listeler	Yalnızca üstten erişilen liste: push ve pop dizi ya da bağlı listeyle O(1).
Kuyruklar ve Deque (8)	Diziler, Bağlı Listeler, Yığınlar	İki uçlu erişim: dairesel dizi ya da bağlı liste; iki yığınla kuyruk.
Ağaçlar (9)	Özyineleme, Bağlı Listeler	Düğümler artık birden çok çocuğa bağlanır; gezinmeler özyinelemeyle yazılır.
Arama Ağaçları (10)	Ağaçlar	Ağaca sıralama koşulu eklenir: arama, ekleme ve silme O(h); denge yüksekliği O(log n) tutar.
Öncelikli Kuyruklar ve Heap (11)	Kuyruklar ve Deque, Ağaçlar	Kuyruk ADT'sinin öncelikli biçimi; tam ikili ağacın dizide saklanmasıyla heap.
Eşlemeler ve Hash Tabloları (12)	Diziler, Liste ADT ve Java Koleksiyonları	Eşleme ADT'si; hash tablosu dizi kovaları ve zincirleme listeleriyle ortalama O(1).
Kümeler ve Ayrık Kümeler (13)	Arama Ağaçları, Eşlemeler ve Hash Tabloları	Küme, değeri olmayan eşlemedir: hash ve ağaç tabanlı gerçeklemeler; ayrık kümeler ağaçla.
Çizgeler (14)	Kuyruklar ve Deque, Ağaçlar, Öncelikli Kuyruklar ve Heap, Kümeler ve Ayrık Kümeler	Ağacın genellemesi: BFS kuyruk, Dijkstra ve Prim öncelikli kuyruk, Kruskal ayrık küme kullanır.

Tablo 1: Haritadaki bağlar.

Haritanın ana omurgası şöyle okunabilir: **analiz → dizi → bağlı yapı → ağaç → çizge**. İlk dört bölüm bütün dersin aracıdır: her yapı bir soyut veri tipi olarak tanımlanır ve işlemleri Büyük O ile ölçülür. Dizi ile bağlı liste, sonraki bütün yapıların iki temel gerçekleme biçimidir; yığın, kuyruk ve liste ADT'si bu ikisinin üzerine kurulur. Özyineleme ağaçlara götürür; arama ağaçları ile hash tabloları eşleme ve kümelerin iki gerçeklemesidir. Çizgeler son bölümde öncekilerin hepsini kullanır: gezinme kuyruğu ve yığını, öncelikli kuyruk ve ayrık kümeler.

KISIM I

Temeller

Bölüm 1–4

KISIM I · BÖLÜM 1

Giriş

Bir program ne kadar akıllıca yazılmış olursa olsun, sonunda bellekteki verilerle çalışır. Bu verilerin nasıl temsil edildiği, nasıl düzenlendiği ve hangi işlemlerle kullanıldığı, programın hem doğruluğunu hem de hızını belirler. Not, en alttaki düzeyden başlar: metin, sayı, görüntü ve sesin 0 ve 1'lerle nasıl temsil edildiğini anlatır. Ardından Java'nın ilkel tiplerine, tip güvenliğine, çağrı yığını ile heap belleğe ve değer ile referans kopyalama arasındaki farka geçer. Jenerikler, tek bir sınıfın her referans tipiyle nasıl çalışabildiğini gösterir. Notun merkezinde soyut veri tipi ile veri yapısı arasındaki ayrım vardır: arayüz ne yapılacağını, sınıf nasıl yapılacağını söyler. Bu ayrım bir yığın örneğiyle, satır satır izlenen bir Java koduyla somutlaştırılır. Son kesimler, verilerin düzenlenme biçimlerini, veri yapısı seçerken sorulacak soruları ve dersin temel veri yapılarını bir arada tanıtır.

Bu bölümün kodu: `github.com/sercankulcu/data-structures-java/Bolum01/src`

ÖĞRENME ÇIKTILARI

Temel kavramlar
Veri tipi, soyut veri tipi ve veri yapısı arasındaki farkı açıklamak.

Yapı seçimi
Bir problem için uygun veri yapısını gereksinimlere göre seçmek.

Java ile ADT
Bir ADT'yi arayüz (*interface*), bir veri yapısını sınıf olarak yazmak; referans ve jenerikleri doğru kullanmak.

Temel yapılar
Temel veri yapılarını düzenleme biçimleri ve tipik kullanımlarıyla tanımak.

1.1 Dijital veri

Veri yapıları dersinin konusu, verinin bellekte düzenlenmesidir. Bir düzenden söz edebilmek için önce düzenlenecek şeyin ne olduğunu bilmek gerekir. Bilgisayarın belleğinde ne harf vardır ne fotoğraf ne de şarkı; yalnızca 0 ve 1 değerli küçük hücreler vardır. Gündelik hayattaki bilgiler bu hücrelere nasıl sığdırılır? Bu kesimin sorusu budur.

1.1.1 Dijital veri nedir?

Telefonla bir ses kaydı yapıldığını düşünelim. Konuşan kişinin sesi havada bir basınç dalgası olarak yayılır. Bu dalga süreklidir: her an bir değeri vardır ve bu değer iki ölçüm arasında da sürekli değişir. Böyle sürekli büyüklüklere **analog** denir. Telefonun mikrofonu bu dalgayı bir elektrik gerilimine çevirir; gerilim de hâlâ analogdur. Telefonun işlemcisi ise sürekli bir büyüklükle çalışamaz. Ona sayılar, daha doğrusu 0 ve 1'lerden oluşan diziler gerekir.

Analog sinyali sayılara çeviren devreye **analogdan dijitale dönüştürücü** (*analog-to-digital converter*, kısaca ADC) denir. ADC, sinyalin değerini belirli aralıklarla ölçer ve her ölçümü bir sayı olarak kaydeder. Kayıt dinlenirken tersi yapılır: **dijitalden analoğa dönüştürücü** (*digital-to-analog converter*, kısaca DAC) sayıları yeniden bir gerilime, hoparlör de bu gerilimi ses dalgasına çevirir. Aradaki her iş bilgisayarın sayılar üzerinde yaptığı bir işlemdir: kaydetmek, kopyalamak, göndermek, sesi açıp kısmak, gürültüyü azaltmak. Şekil 1.1 bu zinciri gösterir.



Şekil 1.1: Analog ses → ADC → 0 ve 1'ler → bilgisayar → DAC → analog ses. Mikrofondan gelen sürekli sinyal ADC'de sayılara çevrilir; bilgisayar yalnız bu sayılarla çalışır; DAC sayıları yeniden kulaklığa giden sinyale dönüştürür.

Tanım 1.1

Dijital veri (*digital data*), sayı, metin, görüntü ve ses gibi bilgilerin elektronik ortamda 0 ve 1'lerden oluşan diziler hâlinde temsil edilmiş biçimidir. Metin belgeleri, resim dosyaları, veritabanları ve ses kayıtları dijital veri örnekleridir.

Tanımdaki önemli sözcük *temsil*dir. Bilgisayardaki 0 ve 1'ler, kendi başlarına hiçbir şey anlatmaz. Aynı sekiz bitlik dizi bir yerde bir harfi, başka bir yerde bir sayıyı, bir başka yerde bir pikselin kırmızılık düzeyini gösterebilir. Bitlere anlam kazandıran, onların nasıl yorumlanacağını söyleyen kuraldır. Bu

kurala **kodlama** denir. Dosya uzantısı (.txt, .png, .mp3) da bir ipucudur: programa, dosyadaki bitleri hangi kodlamayla okuyacağını söyler.

Sık yapılan hata

"Dijital" sözcüğünü "bilgisayarda olan" anlamında kullanıp analog ile dijital arasındaki asıl farkı gözden kaçırmak. Fark, değerlerin sürekli mi yoksa sayılabilir adımlarla mı ifade edildiğidir. Analog bir saat ibresi her ara konumu alabilir. Dijital bir saat ise yalnızca belirli değerleri, örneğin 14.07 ve 14.08'i gösterir; aradaki anları göstermez. Dijitale çevirirken bilginin bir kısmı kaçınılmaz olarak kaybolur. Mühendisliğin işi, bu kaybı kullanıcının fark etmeyeceği kadar küçük tutmaktır.

1.1.2 Bit, bayt ve ikili sayılar

Belleğin en küçük birimine **bit** (*binary digit*: ikili basamak) denir. Bir bit yalnız iki değer alabilir: 0 ya da 1. Donanımda bu iki değer, örneğin bir devredeki gerilimin düşük ya da yüksek olmasıyla, bir disk yüzeyindeki küçük bir bölgenin iki ayrı yönde mıknatıslanmasıyla gösterilir. İki durumu ayırt etmek, on durumu ayırt etmekten çok daha kolay ve güvenilirdir. Bilgisayarların ikili sistemi kullanmasının temel nedeni budur.

Tek bir bit az bilgi taşır, bu yüzden bitler gruplar hâlinde kullanılır. Sekiz bitlik gruba **bayt** (*byte*) denir. Bit sayısı arttıkça temsil edilebilecek değer sayısı hızla büyür. Bir bit 2 değer verir. İki bit 00, 01, 10, 11 olmak üzere 4 değer verir. Her yeni bit, var olan her diziyi bir 0 ve bir 1 ile sürdürerek değer sayısını iki katına çıkarır. Dolayısıyla n bit 2^n farklı değer temsil edebilir: 8 bit 256, 16 bit 65.536, 32 bit 4.294.967.296 değer.

Bir bit dizisini sayı olarak okumanın kuralı **ikili sayı sistemidir**. Onluk sistemde her basamak, sağdan sola 1, 10, 100, 1000 gibi onun kuvvetleriyle çarpılır: 365 = 3·100 + 6·10 + 5·1. İkili sistemde aynı fikir ikinin kuvvetleriyle uygulanır. Sekiz bitlik bir dizide basamak değerleri soldan sağa 128, 64, 32, 16, 8, 4, 2, 1'dir.

Örnek 1.1

13 sayısını sekiz bitle yazalım, sonra 01000001 dizisinin hangi sayıyı gösterdiğini bulalım.

1. **13'ü ayrıştırma:** 13'ten küçük ya da ona eşit en büyük ikinin kuvveti 8'dir. 13 − 8 = 5 kalır. 5 içinde 4 vardır, 5 − 4 = 1 kalır. 1 içinde 1 vardır, 0 kalır. Demek ki 13 = 8 + 4 + 1.
2. **Basamakları doldurma:** 128, 64, 32, 16 basamaklarına 0; 8 basamağına 1; 4 basamağına 1; 2 basamağına 0; 1 basamağına 1 yazılır. Sonuç `00001101` 'dir.
3. **Ters yön:** `01000001` dizisinde 1 olan basamaklar 64 ve 1'dir. Değer 64 + 1 = 65'tir.
4. **Denetim:** `00001011` dizisinde 1 olan basamaklar 8, 2 ve 1'dir; bu 11 eder, 13 etmez. Ayrıştırmada tek bir basamağı kaydırmak bambaşka bir sayı verir.

Bayttan büyük birimler de günlük kullanımda karşımıza çıkar. Bir kilobayt (KB) yaklaşık bin bayt, bir megabayt (MB) yaklaşık bir milyon bayt, bir gigabayt (GB) yaklaşık bir milyar bayttır. "Yaklaşık" denmesinin nedeni, bazı yazılımların bu birimleri 1000 yerine 1024'ün kuvvetleriyle hesaplamasıdır; $1024 = 2^{10}$, ikili sistemde yuvarlak bir sayıdır.

> **Sık yapılan hata**
>
> Bit ile baytı karıştırmak. İnternet hızları genellikle saniyede bit (Mbit/s), dosya boyutları ise bayt (MB) ile verilir. 100 Mbit/s'lik bir bağlantı, saniyede 100 MB değil, en iyi durumda 100 / 8 = 12,5 MB aktarır.

1.1.3 Dijital verinin özellikleri

Dijital verinin yaygınlaşmasının ardında üç özellik vardır.

- **Sayısal olması.** Veriler 0 ve 1 olarak, yani ikili sayılarla temsil edilir; örneğin `1011 0010`. Her şey sayı olduğu için her tür bilgi aynı donanımla saklanabilir, aynı devrelerle taşınabilir. Bir sabit disk, içinde fotoğraf mı yoksa roman mı olduğunu bilmez.
- **Kolay saklanması.** Dijital veri elektronik ortamda saklanabilir ve kolayca paylaşılabilir: disk, bulut ve ağ bunun örnekleridir. Daha önemlisi, kopyalama kayıpsızdır. Bir plak ya da kaset her kopyalamada biraz bozulur; dijital bir dosyanın milyonuncu kopyası ilkiyle bit bit aynıdır.
- **İşlemeye uygun olması.** Bilgisayarlar dijital veriyi çok hızlı işler. Günümüzün sıradan bir işlemcisi saniyede yaklaşık 10^9, yani bir milyar düzeyinde temel işlem yapar.

Üçüncü madde, veri yapıları dersinin neden var olduğunu da açıklar. Saniyede bir milyar işlem, ilk bakışta her problemi anında çözmeye yetecek gibi görünür. Oysa veri büyüdükçe kötü düzenlenmiş bir programın yaptığı iş, verinin boyutundan çok daha hızlı büyüyebilir. Bir milyon kayıt olsun ve program her kaydı öteki bütün kayıtlarla karşılaştırsın. Bu, $10^6 \cdot 10^6 = 10^{12}$ karşılaştırma eder. Saniyede 10^9 işlemle iş yaklaşık 1000 saniye, yani 16 dakikadan fazla sürer. Aynı iş, verileri uygun bir yapıda tutan bir programla saniyenin çok küçük bir kesrinde bitebilir. Bilgisayar yalnızca 0 ve 1'leri tanır. Veri yapıları, bu bitleri anlamlı ve verimli bir düzende tutmanın yollarıdır.

1.1.4 Metin, sayı, görüntü ve sesin temsili

Her bilgi türü için ayrı bir kodlama kuralı vardır. Tablo 1.1 dört temel türü ve her birinin kısa bir örneğini verir; ardından her biri ayrı ayrı açıklanır.

Bilgi türü	Nasıl temsil edilir?	Örnek
Metin	ASCII, Unicode gibi karakter kodlamaları	`'A'` → 65 → `01000001`
Sayısal	tam sayılar, kayan nokta sayıları	13 → `00001101` · 3,14 → IEEE 754
Görüntü	piksellerin renk değerleri	kırmızı piksel → (R, G, B) = (255, 0, 0)

Bilgi türü	Nasıl temsil edilir?	Örnek
Ses	ses dalgalarının dijital örneklemi	CD kalitesi → saniyede 44.100 örnek

Tablo 1.1: Dijital verinin temsil biçimleri.

Metin. Bir metni saklamak için her karaktere bir sayı karşılık getirilir. Bu eşlemeyi veren tabloya **karakter kodlaması** denir. En eski ve en yaygın kodlamalardan biri **ASCII**'dir (*American Standard Code for Information Interchange*). ASCII, 128 karakteri 0–127 arasındaki sayılarla gösterir: büyük harfler 65'ten (`'A'`), küçük harfler 97'den (`'a'`), rakamlar 48'den (`'0'`) başlar. 128 değer 7 bite sığar; pratikte her karakter bir baytta saklanır. Örneğin `'A'` karakteri 65 sayısıyla, o da `01000001` bit dizisiyle temsil edilir.

ASCII yalnız İngilizce harfleri kapsar; "ş", "ğ", "ı" gibi harfler onda yoktur. Dünyadaki bütün yazı sistemlerini tek bir tabloda toplayan standart **Unicode**'dur. Unicode her karaktere bir *kod noktası* verir; örneğin "ş" harfinin kod noktası 351'dir. Bu sayının bayt olarak nasıl yazılacağı ayrı bir kuraldır. Web'de en yaygın biçim olan UTF-8, ASCII karakterlerini yine tek baytla, "ş" gibi karakterleri iki ya da daha çok baytla yazar. Bu yüzden Türkçe harf içeren bir sözcük, harf sayısından fazla bayt tutabilir. "Veri" 4 harf ve 4 bayttır. "Şeker" ise 5 harflidir ama UTF-8'de 6 bayt tutar, çünkü "Ş" iki bayt kaplar. Java'nın `char` tipi ise 16 bitlik bir UTF-16 kod birimidir; "ş" Java'da tek bir `char`'a sığar.

Örnek 1.2

"Veri" sözcüğünün ASCII ile bellekte nasıl durduğunu bulalım.

1. **Kodlar:** `'V'` = 86, `'e'` = 101, `'r'` = 114, `'i'` = 105.
2. **İkili yazım:** 86 = 64 + 16 + 4 + 2 → `01010110`. 101 = 64 + 32 + 4 + 1 → `01100101`. 114 = 64 + 32 + 16 + 2 → `01110010`. 105 = 64 + 32 + 8 + 1 → `01101001`.
3. **Sonuç:** Bellekte ardışık dört bayt durur: `01010110 01100101 01110010 01101001`. Bu 32 bit kendi başına bir sözcük olduğunu söylemez. Aynı 32 bit bir `int` olarak okunursa çok büyük bir tam sayı verir. Anlamı belirleyen, bitleri okuyan programın hangi kodlamayı kullandığıdır.

Sayılar. Tam sayılar ikili sistemde doğrudan yazılır. Negatif sayılar için bilgisayarlar *ikiye tümleyen* denen bir kural kullanır; bu kuralın ayrıntısı burada gerekmez, sonucu yeterlidir: n bitlik bir işaretli tam sayı -2^{n-1} ile $2^{n-1} - 1$ arasındaki değerleri alır. 8 bit için bu aralık −128 … 127'dir. Ondalıklı sayılar ise **kayan nokta** (*floating point*) biçiminde saklanır. Bu biçimi tanımlayan standart IEEE 754'tür. Kayan nokta sayısı, bilimsel gösterimdeki gibi bir işaret, bir anlamlı basamaklar kısmı ve bir üs taşır: 3,14 gibi bir sayı yaklaşık olarak $1{,}57 \cdot 2^1$ biçiminde saklanır. Anlamlı basamak sayısı sınırlı olduğu için pek çok ondalıklı sayı ancak yaklaşık olarak temsil edilebilir.

Görüntü. Bir dijital görüntü, **piksel** denen küçük kareler ızgarasıdır. Her pikselin rengi, kırmızı (R), yeşil (G) ve mavi (B) ışığın ne kadar karıştırıldığını gösteren üç sayıyla verilir. Her bileşen genellikle bir bayttır, yani 0 ile 255 arasında bir değer alır. Tam kırmızı bir piksel (255, 0, 0), beyaz (255, 255,

255), siyah (0, 0, 0) olarak yazılır. Görüntü, satır satır dizilmiş bu üçlülerin uzun bir dizisi olarak saklanır.

Ses. Ses, ADC'nin sinyali belirli aralıklarla ölçmesiyle sayıya dönüşür. Her ölçüme **örnek**, saniyedeki ölçüm sayısına **örnekleme hızı** denir. Müzik CD'leri saniyede 44.100 örnek alır; her örnek 16 bitlik bir tam sayıdır ve stereo kayıtta sol ve sağ kanal için ayrı ayrı örnek tutulur.

Örnek 1.3

Sıkıştırılmamış bir dakikalık CD sesi ile sıkıştırılmamış 1920 × 1080 piksellik bir fotoğraf kaç bayt tutar?

1. **Ses, bir saniye:** 44.100 örnek × 2 bayt (16 bit) × 2 kanal = 176.400 bayt.
2. **Ses, bir dakika:** 176.400 × 60 = 10.584.000 bayt, yani yaklaşık 10,6 MB.
3. **Fotoğraf:** 1920 × 1080 = 2.073.600 piksel. Her piksel 3 bayt (R, G, B) tutar: 2.073.600 × 3 = 6.220.800 bayt, yaklaşık 6,2 MB.
4. **Yorum:** Telefonlardaki MP3 ya da JPG dosyaları bu değerlerin çok altındadır. Bu biçimler, insan kulağının ve gözünün fark etmeyeceği ayrıntıları atarak veriyi sıkıştırır. Sıkıştırma da bir veri düzenleme işidir: aynı bilgiyi daha az bitle temsil etmenin yollarını arar.

Sık yapılan hata

Rakam karakteri ile sayının kendisini aynı sanmak. `'7'` karakterinin ASCII kodu 55'tir, değeri 7 değildir. Bir metin dosyasından okunan “7” bilgisi önce sayıya çevrilmeden hesaba katılırsa yanlış sonuç alınır. Java'da `'7' - '0'` ifadesi 55 − 48 = 7 verir; rakam karakterini sayıya çevirmenin kısa yolu budur.

1.1.5 Dijital veri türleri

Dijital veriler, nasıl düzenlendiklerine göre de sınıflandırılır. Tablo 1.2 yaygın bir sınıflandırmayı verir.

Tür	Açıklama	Örnek
Yapısal	belirli bir şemaya göre düzenlenmiş veri	ilişkisel veritabanı tabloları, CSV
Yarı yapısal	etiketlerle kısmen düzenlenmiş veri	XML, JSON
Metin	insanların okuyabileceği metinler	TXT, e-posta
Resim ve grafik	görsel veriler	JPG, PNG
Ses ve müzik	ses kayıtları	MP3, WAV

Tablo 1.2: Dijital veri türleri.

Yapısal veride her kaydın hangi alanlardan oluştuğu önceden bellidir. Bir öğrenci tablosunun her satırı numara, ad, bölüm ve not ortalaması alanlarını içerir; bu alanların listesine *şema* denir. CSV (*comma-separated values*: virgülle ayrılmış değerler) dosyası da yapısaldır: her satır bir kayıt, virgüllerle ayrılan her sütun bir alandır. Yapısal veride arama ve sıralama kolaydır, çünkü her değerin yeri bellidir.

Yarı yapısal veride bir şema yoktur ya da gevşektir, ama veri etiketlerle kendi yapısını kendisi anlatır. Aşağıdaki JSON (*JavaScript Object Notation*) parçası bir öğrenciyi anlatır:

```
{ "no": 101, "ad": "Ayşe", "dersler": ["Veri Yapıları", "Fizik"] }
```

Her değerin önünde onun ne olduğunu söyleyen bir etiket (`"no"` , `"ad"`) vardır. Bir başka öğrencinin kaydında `"dersler"` alanı hiç olmayabilir ya da fazladan bir `"telefon"` alanı bulunabilir. XML (*eXtensible Markup Language*) aynı işi `<ad>Ayşe</ad>` gibi açılış ve kapanış etiketleriyle yapar. Yarı yapısal veri iç içe yapılar kurmaya elverişlidir: bir öğrencinin içinde bir ders listesi, her dersin içinde bir not listesi bulunabilir. Bu iç içelik, ağaç biçiminde bir düzendir.

Metin, görüntü ve ses gibi türler ise bir şemaya göre değil, içeriğe göre düzenlenir. Bir e-postanın gövdesi ya da bir fotoğrafın pikselleri, bir veritabanı tablosu gibi alanlara bölünmez. Bu tür veriler çoğu zaman *yapısal olmayan* veri olarak da anılır.

Bu sınıflandırmadan veri yapıları için şu sonuç çıkar: verinin dışarıdaki biçimi, program içinde seçilecek yapıyı etkiler. CSV satırları doğal olarak bir dizi ya da liste içinde tutulur. JSON ve XML'in iç içe yapısı, program içinde bir ağaçla temsil edilir. Bir e-posta adresinden kişiye ulaşmak için ise anahtar–değer eşlemesi kullanılır.

1.2 Veri yapıları ve algoritmalar

Bir program iki şeyden oluşur: üzerinde çalıştığı veriler ve bu verilere uyguladığı adımlar. Bu iki parça birbirinden bağımsız değildir. Verilerin nasıl tutulduğu, hangi adımların kolay, hangilerinin zahmetli olacağını belirler. Adımların neler olacağı da verilerin nasıl tutulması gerektiğini belirler.

1.2.1 Algoritma ve veri yapısı

Tanım 1.2

Algoritma (*algorithm*), belirli bir görevi yerine getirmek için adım adım hazırlanmış bir plan, yani sonlu sayıda açık talimatın sıralı bütünüdür. Algoritma bilgisayar biliminin temel yapı taşlarından biridir.

Yemek tarifi, algoritmanın gündelik hayattaki en tanıdık örneğidir: malzemeler bellidir, adımlar sıralıdır, her adım ne yapılacağını açıkça söyler ve tarif sonunda biter. Bilgisayardaki bir algoritmanın da aynı özellikleri taşıması gerekir. Adımlar bilgisayarın yapabileceği kadar açık olmalı, belirsiz bir talimat ("biraz tuz ekle") bulunmamalı ve algoritma sonlu sayıda adımda durmalıdır.

Tanım 1.3

Veri yapısı (*data structure*), verileri bilgisayarda düzenlemenin ve saklamanın bir yoludur. Bir veri yapısı, verilerin bellekte hangi biçimde duracağını ve bu verilere hangi işlemlerle erişileceğini tanımlar.

Veri yapısı, verileri yönetmek ve işlemek için gereken altyapıdır. Algoritma bu altyapının üzerinde çalışır. İkisi birlikte iyi seçilirse yazılım verimli çalışır ve problem sistemli biçimde çözülür. İsviçreli bilgisayar bilimci Niklaus Wirth, bu ilişkiyi 1976'da yayımladığı kitabının adıyla özetlemiştir: *Algoritmalar + Veri Yapıları = Programlar* (*Algorithms + Data Structures = Programs*).

İlişkiyi bir örnekle görelim. Bin kişilik bir telefon rehberinde bir ismi aramak gerekiyor. İsimler rastgele sırada duruyorsa tek yol baştan başlayıp her ismi tek tek kontrol etmektir. Aranan isim en sonda ya da hiç yoksa bin karşılaştırma gerekir. İsimler alfabetik sıradaysa bambaşka bir algoritma kullanılabilir: ortadaki isme bakılır; aranan isim alfabede ondan önceyse sağ yarı, sonraysa sol yarı tümüyle elenir. Her karşılaştırma aday sayısını yarıya indirir: 1000, 500, 250, 125, 63, 32, 16, 8, 4, 2, 1. En çok 10 karşılaştırmada sonuca ulaşılır. Aynı veriler, farklı bir düzende tutulunca yüz kat hızlı bir algoritmaya izin vermiştir. Bunun bedeli de vardır: sıralı düzeni korumak için her yeni isim doğru yere eklenmelidir.

Sık yapılan hata

Veri yapısını yalnız "verilerin durduğu yer", algoritmayı ise ayrı ve bağımsız bir konu sanmak. Yukarıdaki ikiye bölme yöntemi, sıralı olmayan bir listede çalışmaz; yanlış sonuç verir. Bir algoritmanın doğru ve hızlı olup olmadığı, üzerinde çalıştığı veri yapısının sağladığı güvencelere bağlıdır.

1.2.2 Düzenin bedeli ve kazancı

Veri yapılarının temel amacı verilere hızlı ve etkili erişimdir. Sunumdaki şekil bunu bir mutfak çekmecesiyle anlatır. Aynı eşyalar bir çekmecede karmakarışık, ötekinde bölmelere ayrılmış olarak durur. Karışık çekmecede bir kaşık aramak, bütün eşyaları tek tek elden geçirmeyi gerektirebilir. Düzenli çekmecede kaşıkların bölmesi bellidir; el doğrudan oraya uzanır.

Benzetmenin öğrettiği ikinci şey daha az fark edilir: düzen bedava değildir. Düzenli çekmeceye bir eşya koyarken onun bölmesini bulmak gerekir. Karışık çekmeceye ise eşya gelişigüzel atılır. Demek ki düzen, ekleme işini biraz zorlaştırıp arama işini çok kolaylaştırır. Hangi düzenin seçileceği, hangi işin

daha sık yapılacağına bağlıdır. Çekmeceye yılda bir eşya giriyor ama her gün bir şey aranıyorsa düzen kesinlikle kazançlıdır. Bir şey her gün atılıp hiç aranmıyorsa (çöp kutusunu düşünelim) düzenle uğraşmak gereksizdir.

Veri yapısı seçmenin özü bu ödünleşimdir (*trade-off*). Her yapı bazı işlemleri ucuzlatır, bazılarını pahalılaştırır. Programın hızını ve bellek kullanımını doğrudan bu seçim belirler.

1.2.3 Temel yapı taşları: nesneler, diziler, bağlantılar

Ne kadar karmaşık görünürse görünsün, veri yapılarının neredeyse hepsi üç temel parçadan kurulur.

Nesneler. Bir nesne, birbiriyle ilgili nitelikleri adlandırılmış alanlar hâlinde tek bir birimde toplar. Bir öğrencinin adı ve numarası iki ayrı değişkende de tutulabilir, ama bu durumda hangi adın hangi numaraya ait olduğunu programcının aklında tutması gerekir. Java'da bu iki niteliği tek bir birimde toplayan sınıf şöyle yazılır:

```
class Ogrenci {
  String ad;   // öğrencinin adı
  int no;      // öğrenci numarası
  Ogrenci(String ad, int no) { this.ad = ad; this.no = no; }
}
```

Bir `Ogrenci` nesnesi oluşturulduğunda `ad` ve `no` alanları bellekte birlikte durur ve birlikte taşınır. Nesneler, veri yapılarındaki *düğümlerin* de temelidir: bir düğüm, bir değer ile o değerin komşularını gösteren alanları bir arada tutan küçük bir nesnedir.

Diziler. Bir dizi, aynı türden öğeleri bellekte ardışık yuvalarda toplu olarak saklar. Ardışıklık dizinin en büyük gücüdür. Dizinin bellekte nerede başladığı ve her öğenin kaç bayt tuttuğu biliniyorsa i. öğenin adresi tek bir çarpma ve toplamayla hesaplanır: başlangıç adresi + i × öğe boyutu. Öğe 4 baytlık bir `int` ise ve dizi 1000 numaralı adreste başlıyorsa 5 numaralı öğe 1000 + 5 × 4 = 1020 adresindedir. Bu hesap dizinin boyundan bağımsızdır; on öğeli bir dizide de on milyon öğeli bir dizide de aynı sürede yapılır.

Bağlantılar. Bir bağlantı, bir varlıktan başka bir varlığa yönelen bir işarettir. Java'da bu işarete **referans** denir. Bağlantılar varlıklar arasında isteğe bağlı, yönlü ilişkiler kurar: bir öğrenci danışmanını, bir şehir komşu şehirleri, bir müzik listesindeki şarkı sıradaki şarkıyı gösterebilir. Bağlantılı yapıların öğeleri bellekte dağınık durabilir; düzeni ardışıklık değil, oklar sağlar. Sunumdaki şekilde son figürün okunun ucunda bir soru işareti vardır. Zincirin bir sonu olmalıdır ve hiçbir şeyi göstermeyen bu son bağlantı Java'da `null` ile ifade edilir.

Dizi ile bağlantılar arasındaki seçim, veri yapıları dersinin ana temalarından biridir. Dizi indisle anında erişim sağlar ama boyutu baştan bellidir ve araya ekleme yapmak için öğeleri kaydırmak gerekir. Bağlantılı yapı istediği kadar büyür ve araya ekleme yalnızca birkaç okun yönünü değiştirmekle yapılır; ama i. öğeye ulaşmak için baştan başlayıp okları tek tek izlemek gerekir.

1.3 Veri tipleri

Bir programlama dili, bitleri nasıl yorumlayacağını programcıdan öğrenir. Programcı her değişkenin tipini belirterek “bu bellek alanındaki bitleri tam sayı olarak oku” ya da “bunları bir karakter olarak oku” demiş olur. Bu kesim Java'nın hazır tiplerini ve tiplerin programı hatalardan nasıl koruduğunu anlatır.

1.3.1 İlkel veri tipleri

Java'da dilin kendisine gömülü sekiz temel tip vardır. Bunlara **ilkel tip** (*primitive type*) denir. İlkel tipler tam sayıları, ondalıklı sayıları, mantıksal değerleri ve karakterleri temsil eder. Tablo 1.3 bu tipleri, bellekte kapladıkları yeri ve alabilecekleri değerleri gösterir.

Java tipi	Boyut	Değerler
`byte`	8 bit	−128 … 127
`short`	16 bit	−32.768 … 32.767
`int`	32 bit	yaklaşık ±2,1 milyar (−2.147.483.648 … 2.147.483.647)
`long`	64 bit	yaklaşık $\pm 9{,}2 \cdot 10^{18}$
`float`	32 bit	ondalıklı (≈7 anlamlı basamak)
`double`	64 bit	ondalıklı (≈15 anlamlı basamak)
`boolean`	—	`true` / `false`
`char`	16 bit	UTF-16 kod birimi

Tablo 1.3: Java'nın ilkel veri tipleri.

Tablodaki aralıklar rastgele değildir; 1.1.4'teki kuraldan doğrudan çıkar. n bitlik işaretli bir tam sayı -2^{n-1} ile $2^{n-1} - 1$ arasında değer alır. `byte` için $n = 8$: $2^7 = 128$, aralık −128 … 127. `short` için $2^{15} = 32.768$. `int` için $2^{31} = 2.147.483.648$; “yaklaşık ±2,1 milyar” buradan gelir. `long` için $2^{63} \approx 9{,}22 \cdot 10^{18}$'dir. Pozitif tarafın bir eksik olmasının nedeni, 0'ın da pozitif yarıdaki değerlerden birini kullanmasıdır.

İlkel tiplerin dört ortak özelliği vardır. Basit ve temel tiplerdir; başka tiplerden kurulmazlar. Bellekte sabit bir boyut kaplarlar: bir `int` içinde 3 de olsa 2 milyar da olsa 32 bit tutar. İşlemci onları doğrudan işler; iki `int`'i toplamak işlemcinin tek bir komutudur. Son olarak, bir ilkel değişken değerini doğrudan kendi içinde tutar. Bu son özelliğin önemi 1.4'te ortaya çıkacak.

Sabit boyutun bir sonucu, her tipin bir sınırı olmasıdır. Sınır aşıldığında Java hata vermez; değer, aralığın öbür ucuna *sarar*. Buna **taşma** (*overflow*) denir.

```
int buyuk = Integer.MAX_VALUE;          // 2147483647
System.out.println(buyuk + 1);          // -2147483648
long genis = (long) buyuk + 1;
System.out.println(genis);              // 2147483648
System.out.println(0.1 + 0.2);          // 0.30000000000000004
```

Örnek 1.4

Yukarıdaki kodun ilk iki satırında ne olduğunu adım adım izleyelim.

1. **İlk değer:** `buyuk`, `int`'in en büyük değeri olan 2.147.483.647'yi tutar. İkili gösterimde bu sayı, başta bir 0 ve ardından 31 tane 1'dir.
2. **Bir ekleme:** 31 tane 1'e 1 eklenince elde zinciri bütün basamaklardan geçer: başta bir 1, ardından 31 tane 0 kalır.
3. **Yorum:** İşaretli tam sayıda en soldaki bitin 1 olması sayının negatif olduğunu gösterir. Bu bit dizisi `int`'in en küçük değeridir: −2.147.483.648. Program `buyuk + 1` için −2.147.483.648 yazdırır.
4. **Önlem:** Sonucun `int` aralığını aşabileceği hesaplarda `long` kullanılır. Java'da `Math.addExact` gibi metotlar taşma olduğunda sessizce sarmak yerine istisna fırlatır.

Ondalıklı tiplerin sınırı farklıdır: çok büyük ya da çok küçük sayılar temsil edilebilir, ama her sayı yalnız yaklaşık olarak. `float` yaklaşık 7, `double` yaklaşık 15 anlamlı basamak tutar. 0,1 sayısı ikili sistemde sonsuz bir kesirdir (onluk sistemde 1/3'ün 0,333… olması gibi); bu yüzden bellekte ancak yaklaşığı durur. Kodun son satırı bunun sonucunu gösterir: `0.1 + 0.2` ifadesi `0.30000000000000004` yazdırır.

Sık yapılan hata

Ondalıklı sayıları `==` ile karşılaştırmak. `0.1 + 0.2 == 0.3` ifadesi `false` verir. İki `double` değerin eşitliği, farklarının mutlak değeri çok küçük bir sınırdan (örneğin 10^{-9}) küçük mü diye bakılarak sınanır. Para gibi kuruşu kuruşuna doğru olması gereken değerler için de `double` yerine kuruş cinsinden `long` ya da `BigDecimal` kullanılır.

1.3.2 Veri tipi: değer kümesi ve işlemler

Bir tipi yalnızca “alabileceği değerler” olarak düşünmek eksiktir. `int` ile `String` arasındaki fark, yalnız birinin sayı ötekinin metin tutması değildir; ikisine uygulanabilen işlemler de farklıdır.

Tanım 1.4

Veri tipi (*data type*), bir değer kümesini ve bu değerler üzerindeki davranışları birlikte tanımlar. **Değer kümesi**, tipin alabileceği geçerli değerlerdir. **Davranışlar**, bu değerler üzerinde yapılabilecek işlemlerdir.

Tam sayılar için toplama, çıkarma, çarpma ve bölme gibi işlemler tanımlıdır. Dizgiler (`String`) için ise karşılaştırma, birleştirme ve alt dizgi alma gibi işlemler vardır: `"veri" + "yapısı"` ifadesi `"veriyapısı"` dizgisini, `"yapısı".substring(0, 4)` ise `"yapı"` dizgisini verir. Aynı `+` işareti iki tipte farklı iş görür: sayılarda toplar, dizgilerde uç uca ekler. Davranışın tipe ait olduğunun en açık göstergesi budur.

Tipin davranışı belirlediğini en iyi gösteren örnek bölmedir. Java'da iki `int` 'in bölümü yine bir `int` 'tir; ondalık kısım yuvarlanmaz, doğrudan atılır.

```
System.out.println(7 / 2);     // 3
System.out.println(7 % 2);     // 1
System.out.println(-7 / 2);    // -3
System.out.println(-7 % 2);    // -1
System.out.println(7.0 / 2);   // 3.5
System.out.println(7 / 2.0);   // 3.5
int n1 = 7, n2 = 8;
double ort1 = (n1 + n2) / 2;     // 7.0
double ort2 = (n1 + n2) / 2.0;   // 7.5
```

Kodu satır satır okuyalım. `7 / 2` iki tam sayının bölümüdür: 3,5'in ondalık kısmı atılır, sonuç 3 olur. `7 % 2` kalan işlemidir ve 1 verir; bölüm ile kalan birlikte $7 = 2 \cdot 3 + 1$ eşitliğini sağlar. `-7 / 2` sonucu −4 değil −3'tür, çünkü Java ondalık kısmı sıfıra doğru atar; buna uygun olarak kalan da −1 olur: $-7 = 2 \cdot (-3) + (-1)$. İşlenenlerden biri `double` olduğunda, örneğin `7.0 / 2` ya da `7 / 2.0` , bölme ondalıklı yapılır ve sonuç 3.5'tir.

Örnek 1.5

Kodun son iki satırı 7 ve 8'in ortalamasını iki yoldan hesaplıyor. Sonuçlar neden farklı?

1. **Birinci satır:** `(n1 + n2)` bir `int` 'tir ve değeri 15'tir. `15 / 2` iki `int` 'in bölümüdür: sonuç 7. Bu 7, ancak bölme bittikten sonra `double` 'a çevrilir ve `ort1` 'e 7.0 yazılır. Kayıp, atamadan önce olmuştur.
2. **İkinci satır:** `2.0` bir `double` 'dır. Java, işlenenlerden biri `double` olunca ötekini de `double` 'a yükseltir: `15.0 / 2.0 = 7.5` .
3. **Ders:** Sonucun saklandığı değişkenin tipi, işlemin nasıl yapılacağını belirlemez. İşlemi belirleyen, işlenenlerin tipidir.

Tipler birbirine karıştığında Java belirli kurallar uygular. `char` aslında 16 bitlik bir sayıdır; aritmetik bir işlemde `int`'e yükseltilir. Bu yüzden `'A' + 1` ifadesi 66 verir. `+` işleminin bir tarafı `String` olduğunda ise öbür taraf metne çevrilip eklenir: `"A" + 1` ifadesi `"A1"` dizgisini verir. İşlemler soldan sağa yapıldığı için `1 + 2 + "x"` önce 3'ü hesaplar ve `"3x"` verir; `"x" + 1 + 2` ise `"x12"` verir.

> **Sık yapılan hata**
>
> Ortalama, yüzde ya da oran hesabında tam sayı bölmesini unutmak. 30 öğrenciden 7'si başarılıysa `7 / 30 * 100` ifadesi 0 verir, çünkü `7 / 30` tam sayı bölmesinde 0'dır. Doğru sonuç için iki yol vardır. Önce çarpma yapılabilir: `7 * 100 / 30` 23 verir. Ya da işlenenlerden biri ondalıklı yazılır: `7.0 / 30 * 100` yaklaşık 23,33 verir.

1.3.3 Tip güvenliği

Her işlem her tipte geçerli değildir. Bir metni bir sayıyla çarpmanın Java'da bir anlamı yoktur: `"abc" * 2` yazıldığında derleyici programı hiç çalıştırmadan hatayı bildirir. Tiplerin bu koruyucu rolüne **tip güvenliği** denir. Veri tipleri hataları en aza indirmek için kullanılır; tipin getirdiği kısıtlamalar, programcıya neyin mümkün olup neyin olmadığını söyler.

Hatanın ne zaman yakalandığı çok önemlidir. Java gibi *statik tipli* dillerde değişkenlerin tipi kaynak kodda yazılıdır ve derleyici her işlemin tiplere uygun olup olmadığını program çalışmadan önce denetler. Bu denetime **derleme zamanı** denetimi denir. Bir hata derleme zamanında yakalanırsa programcı onu daha programı kimseye vermeden görür ve düzeltir. Aynı hata ancak **çalışma zamanında** ortaya çıkabilseydi, program belki aylarca sorunsuz çalışır, sonra o satıra ilk kez gelindiğinde bir kullanıcının elinde çökerdi.

Tip güvenliği, veri yapılarında özellikle değerlidir. Bir veri yapısı binlerce öğe tutar ve bu öğeler programın çok farklı yerlerinde eklenip çıkarılır. Yanlış tipte tek bir öğenin yapıya girmesi, hatanın kaynağından çok uzakta bir yerde sorun çıkarır. Bu yüzden Java'da veri yapıları, içlerine hangi tipte öğe gireceğini derleyiciye bildiren jeneriklerle yazılır (1.5).

> **Sık yapılan hata**
>
> Derlenen her programın doğru olduğunu sanmak. Tip güvenliği yalnız tiplere aykırı işlemleri yakalar. `7 / 30 * 100` ifadesinin tipleri uygundur ve program derlenir; yanlış olan, programcının niyetidir. Derleyici tip hatalarını yakalar, mantık hatalarını yakalamaz.

1.4 Java bellek modeli

Veri yapılarının neredeyse hepsi, bir değişkenin başka bir yerdeki veriyi "göstermesi" fikrine dayanır. Bu fikri doğru kavramadan bağlı liste, ağaç ya da çizge kodu okunamaz. Bu kesim, Java programı çalışırken değişkenlerin ve nesnelerin bellekte nerede durduğunu anlatır.

1.4.1 Çağrı yığını ve heap

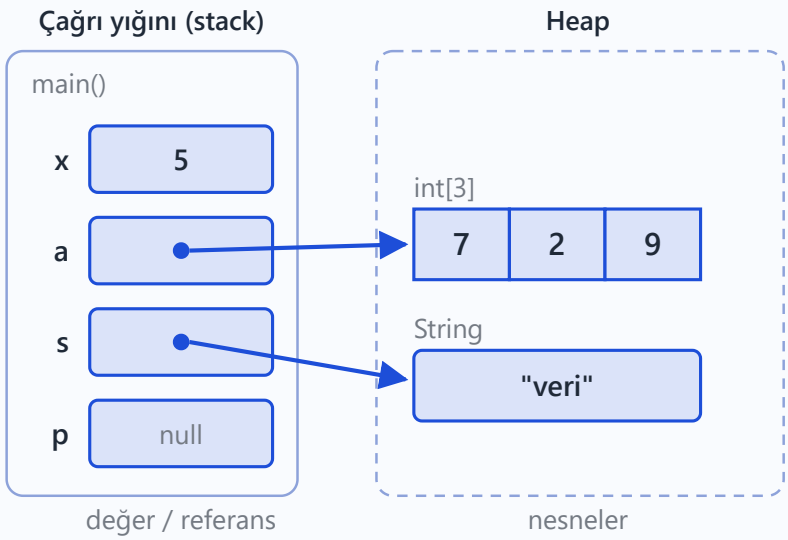
Çalışan bir Java programı belleği iki ana bölgede kullanır. Birinci bölge **çağrı yığınıdır** (*call stack*). Bir metot her çağrıldığında çağrı yığınında o çağrıya ait bir **çerçeve** (*stack frame*) açılır. Metodun parametreleri ve yerel değişkenleri bu çerçevede durur. Metot bittiğinde çerçeve silinir ve içindeki bütün yerel değişkenler onunla birlikte yok olur. Çerçeveler üst üste konur ve en son açılan ilk kapanır; bölgenin "yığın" adını alması bu yüzdendir.

İkinci bölge **heap bellektir** (*heap memory*). `new` ile oluşturulan her nesne, her dizi ve her `String` heap'te yer alır. Heap'teki nesneler hiçbir metodun çerçevesine bağlı değildir; onları oluşturan metot bittikten sonra da yaşamaya devam edebilir.

> **Tanım 1.5**
>
> **İlkel değişken**, değerini doğrudan kendi içinde tutar. **Referans değişken** ise heap'teki bir nesnenin değerini değil, o nesnenin yerini, yani adresini tutar. Bir değişkene atanan bu adrese **referans** (*reference*) denir. Hiçbir nesneyi göstermeyen referansın değeri `null`'dır.

Java'da sekiz ilkel tipin dışındaki her tip bir referans tipidir: diziler, `String`, kendi yazdığımız sınıflar ve kütüphanedeki bütün sınıflar. Aşağıdaki dört satır iki türü yan yana gösterir:

```
int x = 5;              // ilkel: değer
int[] a = {7, 2, 9};    // referans → dizi
String s = "veri";      // referans → nesne
int[] p = null;         // hiçbir şeyi göstermez
```



Şekil 1.2: Dört satırdan sonra bellek. `main()` çerçevesinde `x` 5 değerini doğrudan tutar. `a` ve `s` yalnızca birer ok, yani referanstır: biri heap'teki üç öğeli `int` dizisini, öteki `"veri"` içeren `String` nesnesini gösterir. `p`'nin değeri `null`'dır; hiçbir şeyi göstermez.

Şekil 1.2 bu satırların bellekte bıraktığı izi gösterir. Çerçevedeki dört kutunun dördü de küçüktür ve aynı boydadır. Dizi ister 3 ister 3 milyon öğeli olsun, `a` değişkeni yalnız bir adres tutar. Asıl veri heap'tedir.

Bir nesneyi artık hiçbir referans göstermiyorsa programın ona ulaşmasının bir yolu kalmaz. Böyle nesneler boşuna yer kaplar. Java'da bu nesneleri **çöp toplayıcı** (*garbage collector*) zaman zaman bulup temizler; programcının belleği elle geri vermesi gerekmez. C gibi dillerde bu iş programcıya kalır ve unutulması bellek sızıntısına yol açar.

`null` bir referansın gösterdiği nesneye erişmeye çalışmak çalışma zamanında hata verir. `p` hiçbir diziyi göstermediği için `p[0]` ifadesi bir öğe bulamaz ve Java `NullPointerException` fırlatır. Bağlı yapılarla çalışırken en sık karşılaşılan hata budur.

Sık yapılan hata

Bellekteki "yığın" bölgesi ile yığın veri yapısını karıştırmak. Çağrı yığını, metot çağrılarını yönetmek için Java'nın kendi kullandığı bir yığındır; son giren ilk çıkar kuralıyla çalıştığı için bu adı alır. Bu notun ilerleyen kesimlerinde yazılan `Yigin` arayüzü ise programcının kendi verileri için kullandığı bir veri yapısıdır. "Heap" sözcüğü de iki anlamlıdır: bellek bölgesi olarak heap ile öncelikli kuyrukta kullanılan heap veri yapısı birbirinden bağımsızdır. Sözlükte ilki *heap bellek*, ikincisi *heap* olarak geçer.

1.4.2 Kod izleme: değer mi, referans mı?

Atama işlemi `=`, sağdaki değeri soldaki değişkene kopyalar. Kopyalanan şeyin ne olduğu ise değişkenin türüne bağlıdır. Aşağıdaki sekiz satırlık kod bu farkı adım adım gösterir:

```
int x = 5;
int y = x;
y = 7;
int[] a = {1, 2, 3};
int[] b = a;
b[0] = 9;
b = new int[2];
System.out.println(x + " " + a[0]);
```

Kodu satır satır izleyelim. Tablo 1.4 her satırdan sonra değişkenlerin ve heap'teki nesnelerin durumunu verir; nesneler oluşturulma sırasına göre #1, #2 diye numaralandırılmıştır.

Satır	`x`	`y`	`a`	`b`	Heap
1 `int x = 5;`	5	—	—	—	boş
2 `int y = x;`	5	5	—	—	boş
3 `y = 7;`	5	7	—	—	boş
4 `int[] a = {1, 2, 3};`	5	7	→#1	—	#1 = [1, 2, 3]
5 `int[] b = a;`	5	7	→#1	→#1	#1 = [1, 2, 3]
6 `b[0] = 9;`	5	7	→#1	→#1	#1 = [9, 2, 3]
7 `b = new int[2];`	5	7	→#1	→#2	#1 = [9, 2, 3], #2 = [0, 0]
8 `println`	5	7	→#1	→#2	çıktı: `5 9`

Tablo 1.4: Değer ve referans kopyalama kodunun izlemesi. "→#1" değişkenin 1 numaralı nesneyi gösterdiğini belirtir.

Satır 1–3. `x` ilkel bir değişkendir ve 5 değeri doğrudan çerçevededir. `int y = x;` satırı `x`'in değerinin bir kopyasını `y`'ye yazar. Artık iki ayrı kutuda iki ayrı 5 vardır ve bu kutular birbirinden bağımsızdır. `y = 7;` yalnız `y`'nin kutusunu değiştirir; `x` hâlâ 5'tir.

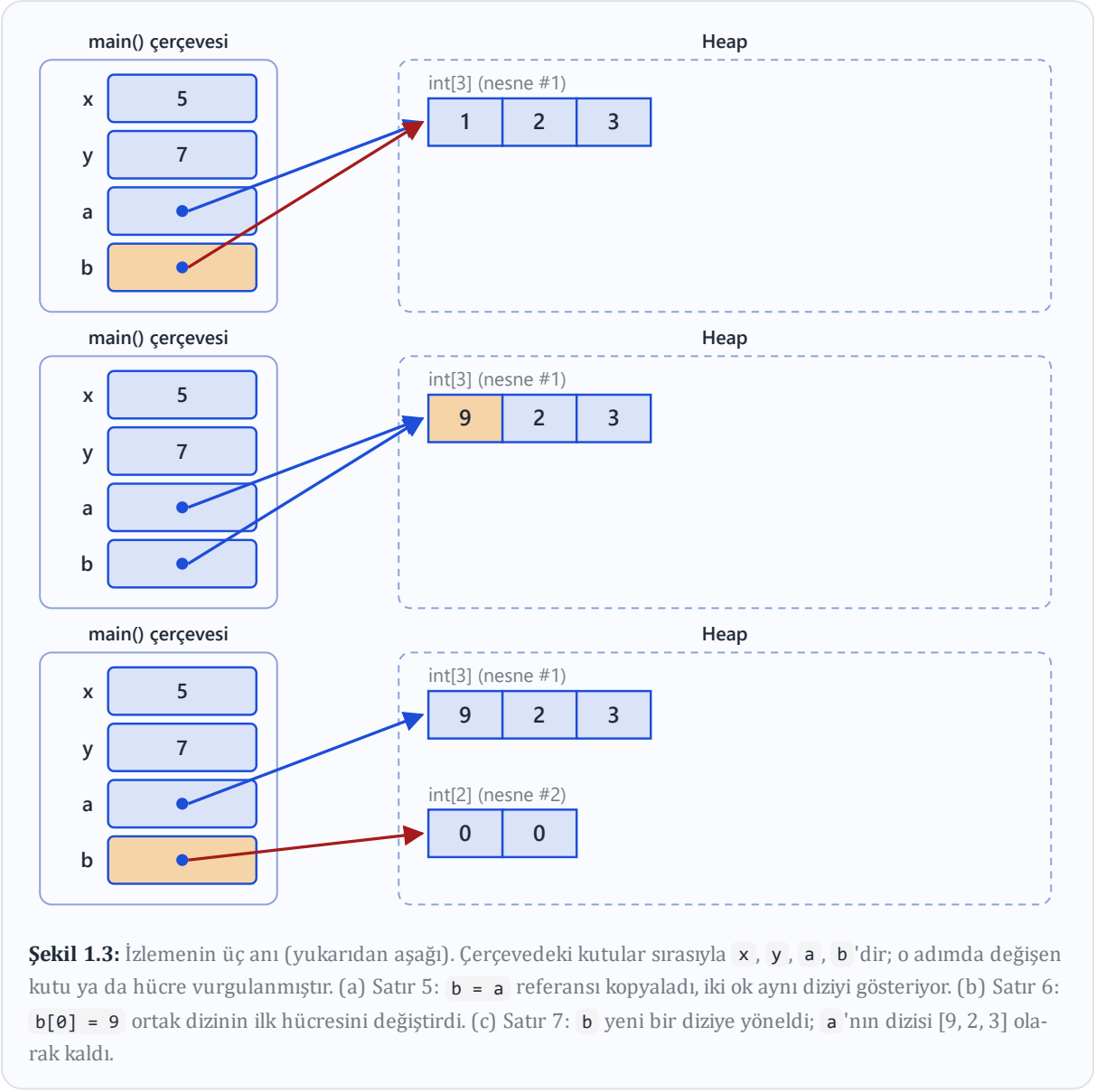
Satır 4. `{1, 2, 3}` ifadesi heap'te üç öğeli yeni bir dizi oluşturur. `a` dizinin kendisini değil, ona bir referansı tutar; şekillerde bu referans bir okla çizilir.

Satır 5. `int[] b = a;` satırı da bir atamadır ve yine kopyalama yapar. Ama kopyalanan şey `a`'nın içindeki değerdir, o değer de bir adrestir. Dizi kopyalanmaz; heap'te hâlâ tek bir dizi vardır. `a` ile `b` şimdi aynı nesneyi gösteren iki oktur.

Satır 6. `b[0] = 9;` satırı `b`'nin gösterdiği nesneye gider ve onun 0. hücresini değiştirir. Aynı nesneyi `a` da gösterdiği için `a[0]` da artık 9'dur. Bir nesneye birden çok referansla ulaşılabilmesine **takma ad** (*aliasing*) denir: `a` ve `b` aynı dizinin iki ayrı adıdır.

Satır 7. `b = new int[2];` heap'te iki öğeli yeni bir dizi oluşturur ve `b`'nin okunu ona çevirir. Java yeni bir `int` dizisinin hücrelerini 0 ile doldurur. Bu satır `a`'yı etkilemez: değişen, `b`'nin içindeki adrestir, 1 numaralı nesne değildir.

Satır 8. `x + " " + a[0]` ifadesi 5, bir boşluk ve 9'u yan yana yazar. Çıktı `5 9`'dur.



Şekil 1.3: İzlemenin üç anı (yukarıdan aşağı). Çerçevedeki kutular sırasıyla `x`, `y`, `a`, `b`'dir; o adımda değişen kutu ya da hücre vurgulanmıştır. (a) Satır 5: `b = a` referansı kopyaladı, iki ok aynı diziyi gösteriyor. (b) Satır 6: `b[0] = 9` ortak dizinin ilk hücresini değiştirdi. (c) Satır 7: `b` yeni bir diziye yöneldi; `a`'nın dizisi [9, 2, 3] olarak kaldı.

Bütün izlemenin özeti tek bir kuraldır: `=` işareti ilkel tipte *değeri*, referans tipte *adresi* kopyalar. Referans tipte bir değişkene yapılan atama okun yönünü değiştirir. Referansın gösterdiği nesneye `b[0] = …` ya da `nesne.alan = …` biçiminde yapılan atama ise nesnenin kendisini değiştirir. Aynı kural kendi yazdığımız sınıflar için de geçerlidir:

```
Ogrenci o1 = new Ogrenci("Ayşe", 101);
Ogrenci o2 = o1;          // ikinci referans, yeni nesne yok
o2.no = 202;
System.out.println(o1.no);   // 202
```

`o2 = o1` yeni bir öğrenci oluşturmaz; ikinci bir ok çizer. `o2.no = 202` ortak nesnenin alanını değiştirdiği için `o1.no` da 202 olur.

Sık yapılan hata

`int[] b = a;` satırının dizinin bir kopyasını oluşturduğunu sanmak. Gerçek bir kopya için yeni bir dizi oluşturulup öğeler tek tek aktarılmalıdır; Java bunu `int[] b = a.clone();` ya da `Arrays.copyOf(a, a.length)` ile kısaca yapar. Ancak bu kopya da yalnız bir düzey derindir: dizinin öğeleri nesne ise kopyalanan yine onların referanslarıdır. Bu tür kopyaya *sığ kopya* denir.

1.4.3 Metotlara değer geçirme

Bir metot çağrıldığında, çağıran tarafın verdiği her argüman metodun parametresine atanır. Bu atama da sıradan bir `=` gibi çalışır. Java her zaman **değerle geçirir**: parametre, argümanın bir kopyasını alır. İlkel tipte kopyalanan değer sayının kendisi, referans tipte ise adrestir. Sonuç iki takas metodunda açıkça görülür:

```
static void takas(int a, int b) {
  int g = a; a = b; b = g;
}
static void takas(int[] d) {
  int g = d[0]; d[0] = d[1]; d[1] = g;
}
```

```
int p = 1, q = 2;
takas(p, q);
System.out.println(p + " " + q);             // 1 2
int[] dd = {1, 2};
takas(dd);
System.out.println(Arrays.toString(dd));     // [2, 1]
```

Örnek 1.6

İki çağrıdan sonra `p`, `q` ve `dd` nedir?

1. `takas(p, q)` **çağrısı:** Yeni bir çerçeve açılır. `a`'ya 1'in, `b`'ye 2'nin kopyası yazılır. Metot `a` ile `b`'yi değiştirir; çerçevede `a = 2`, `b = 1` olur. Metot bitince çerçeve silinir. `p` ve `q`'ya hiç dokunulmamıştır: 1 ve 2.
2. `takas(dd)` **çağrısı:** Yeni çerçevede `d`, `dd`'nin içindeki adresin kopyasını alır. İki ok aynı diziyi gösterir. Metot `d[0]` ile `d[1]`'i değiştirdiğinde ortak dizinin hücreleri yer değiştirir.
3. **Sonuç:** Çıktı `1 2` ve `[2, 1]`'dir. Metot, çağıranın ilkel değişkenlerini değiştiremez; ama çağıranın gösterdiği bir nesneyi referans üzerinden değiştirebilir.

Bu kuralın sınırını da bilmek gerekir. Metot içinde `d = new int[2];` gibi bir atama yapılsaydı yalnız yerel `d` okunun yönü değişirdi; çağıranın `dd` değişkeni eski diziyi göstermeye devam ederdi. Metot bitince yeni diziyi gösteren bir referans kalmaz ve çöp toplayıcı onu temizler. Kısacası metot, nesnenin içini değiştirebilir ama çağıranın değişkenini başka bir nesneye yöneltemez.

Sık yapılan hata

"Java ilkel tipleri değerle, nesneleri referansla geçirir" demek. Bu söz, sonuçları çoğu zaman doğru tahmin ettirir ama mekanizmayı yanlış anlatır. Java her şeyi değerle geçirir; referans tipte kopyalanan değer bir adrestir. Fark, metot içinde parametreye yeni bir nesne atandığında ortaya çıkar: gerçek referansla geçirmede çağıranın değişkeni de değişirdi, Java'da değişmez.

1.4.4 String nesneleri ve karşılaştırma

`String` bir referans tipidir, ama dizilerden önemli bir farkı vardır: bir `String` nesnesi oluşturulduktan sonra içeriği değiştirilemez. Bu özelliğe **değiştirilemezlik** (*immutability*) denir. `toUpperCase()`, `substring()` ya da `+` gibi işlemler var olan nesneyi değiştirmez; sonucu yeni bir nesne olarak döndürür.

```
String s1 = "ders";
String s2 = s1;
s2 = s2.toUpperCase();
System.out.println(s1 + " " + s2);      // ders DERS
String u1 = new String("veri");
String u2 = new String("veri");
System.out.println(u1 == u2);           // false
System.out.println(u1.equals(u2));      // true
```

İlk iki satırdan sonra `s1` ve `s2` aynı `"ders"` nesnesini gösterir. Üçüncü satırda `s2.toUpperCase()` yeni bir `"DERS"` nesnesi oluşturur ve atama `s2`'nin okunu bu yeni nesneye çevirir. Eski nesne değişmemiştir; `s1` hâlâ onu gösterir ve `ders` yazdırır. Dizide ortak nesneyi değiştirmek mümkündü

(`b[0] = 9`), `String`'de böyle bir işlem yoktur. Bu yüzden aynı `String`'i paylaşan değişkenler birbirini etkileyemez.

Kodun son iki satırı ikinci önemli noktayı gösterir. `new String("veri")` her çağrıldığında heap'te ayrı bir nesne oluşturur. `u1 == u2` ifadesi iki referansı karşılaştırır, yani "iki ok aynı nesneyi mi gösteriyor?" diye sorar; cevap `false`'tur. `u1.equals(u2)` ise nesnelerin içeriğini karşılaştırır ve `true` verir.

> **Sık yapılan hata**
>
> İki dizgiyi `==` ile karşılaştırmak. `==` referans tiplerde içeriği değil adresi karşılaştırır. Java, kaynak koda doğrudan yazılmış aynı dizgileri çoğu zaman tek bir nesnede toplar. Bu yüzden `==` bazen "doğru çalışıyor gibi" görünür. Klavyeden okunan ya da hesaplanan bir dizgide ise yanlış sonuç verir. İçerik karşılaştırması her zaman `equals` ile yapılır.

1.5 Jenerikler

Bir yığın, bir liste ya da bir kuyruk, içinde ne tuttuğundan bağımsız olarak aynı biçimde çalışır. Tam sayıları tutan bir yığın ile öğrenci nesnelerini tutan bir yığının `push` ve `pop` işlemleri arasında hiçbir fark yoktur. Her öğe tipi için ayrı bir yığın sınıfı yazmak emek kaybıdır. Üstelik bir hata, sınıfın bütün kopyalarında tekrarlanır. Jenerikler bu sorunu tip güvenliğinden ödün vermeden çözer.

1.5.1 Jenerik olmayan kutunun sorunu

Tek bir öğe saklayan en basit "veri yapısını", bir kutuyu düşünelim. Her tipte öğe alabilmesi için Java'daki bütün sınıfların ortak atası olan `Object` tipini kullanabiliriz:

```
public class EskiKutu {
  private Object icerik;
  public void koy(Object x) { icerik = x; }
  public Object al() { return icerik; }
}
```

```
EskiKutu e = new EskiKutu();
e.koy(42);                    // derlenir: 42 de bir Object
String z = (String) e.al();   // derlenir, ama çalışırken ClassCastException
```

Bu kutu her şeyi kabul eder; sorun da buradadır. `koy(42)` çağrısı derlenir. `al()` ise `Object` döndürdüğü için sonucu bir `String` değişkenine yazmak isteyen programcı bir tür dönüşümü (`(String)`) yazmak zorundadır. Derleyici bu dönüşümün doğru olup olmadığını bilemez ve programı derler. Hata

ancak program çalışırken, kutudan çıkan 42'nin bir `String` olmadığı anlaşıldığında ortaya çıkar: `ClassCastException`. 1.3.3'teki ölçüye göre bu en kötü durumdur: hata derleme zamanında değil, çalışma zamanında yakalanmıştır.

1.5.2 Kutu<T>: her tür için tek yapı

> **Tanım 1.6**
>
> **Jenerik** (*generic*) sınıf, bir ya da daha çok **tür parametresi** alan sınıftır. Tür parametresi, sınıf yazılırken bilinmeyen bir tipin yerini tutan addır (genellikle `T`, `E`, `K`, `V` gibi tek bir büyük harf). Sınıf kullanılırken bu parametreye somut bir referans tipi verilir. Dilin bu olanağına **jenerikler** (*generics*) denir.

Aynı kutunun jenerik hâli şöyledir:

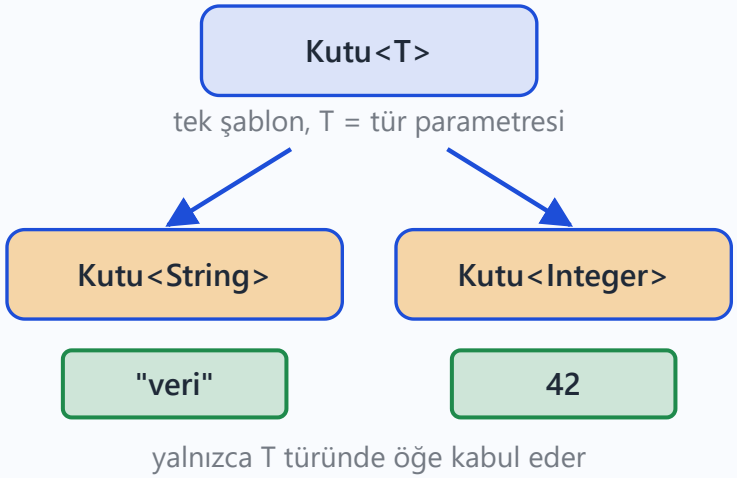
```
public class Kutu<T> {
  private T icerik;
  public void koy(T x) { icerik = x; }
  public T al() { return icerik; }
}

Kutu<String> k = new Kutu<>();
k.koy("veri");
String s = k.al();      // dönüşüm gerekmez
Kutu<Integer> m = new Kutu<>();
m.koy(42);              // int → Integer
// m.koy("x");          // DERLEME HATASI
```

Sınıf adının yanındaki `<T>`, `T`'nin bir tür parametresi olduğunu bildirir. Sınıfın içinde `T`, sanki gerçek bir tipmiş gibi kullanılır: `icerik` alanı `T` tipindedir, `koy` bir `T` alır, `al` bir `T` döndürür.

İstemci kod `Kutu<String> k` yazdığında, derleyici bu değişken için `T`'nin yerine `String` koyar. `k.koy` artık yalnız `String` kabul eder, `k.al()` bir `String` döndürür. Bu yüzden `String s = k.al();` satırında tür dönüşümüne gerek yoktur. `new Kutu<>()` yazımındaki boş açılı ayraçlara *elmas* işleci denir; derleyici tür argümanını soldaki bildirimden çıkarır.

`Kutu<Integer> m` ise aynı sınıftan, bu kez tam sayılar için üretilmiş bir kutudur. Açıklama satırı yapılmış `m.koy("x")` çağrısı açılırsa program derlenmez: `Kutu<Integer>`'ın `koy` metodu yalnız `Integer` kabul eder. 1.5.1'deki hata artık derleme zamanında yakalanır.



Şekil 1.4: Tek bir `Kutu<T>` şablonundan `Kutu<String>` ve `Kutu<Integer>` türleri üretilir. Her biri yalnız kendi türündeki öğeyi saklar: biri `"veri"`, öteki 42.

Şekil 1.4 bu ilişkiyi özetler: kod bir kez yazılır, her tür argümanı için ayrı ve tip güvenli bir sınıf gibi kullanılır. Jeneriklerin kazancı üç maddeyle söylenebilir. Tür denetimi derleme zamanında yapılır; yanlış tür program çalışmadan yakalanır. Kutudan çıkan değer için tür dönüşümü yazılmaz. Kod tekrarı ortadan kalkar.

Sık yapılan hata

Jenerik bir tipi tür argümanı vermeden kullanmak: `Kutu k = new Kutu();`. Java eski kodlarla uyum için buna izin verir ve yalnız bir uyarı yazar. Ama bu durumda kutu `Object` kabul eden eski kutuya döner ve tip güvenliği kaybolur. Jenerik bir sınıf her zaman açılı ayraçlar içinde bir tür argümanıyla kullanılır.

1.5.3 Sarmalayıcı sınıflar ve otomatik kutulama

Tür parametresinin önemli bir kısıtı vardır: `T` yalnız bir referans tipi olabilir. `Kutu<int>` yazılamaz. Bunun nedeni, jenerik sınıfın içinde `T` tipindeki değerlerin hep referans gibi işlenmesidir; ilkel bir değer ise referans değildir, değeri doğrudan tutar.

Çözüm, her ilkel tip için Java'da hazır bulunan **sarmalayıcı sınıflardır**. Bir sarmalayıcı nesne, içinde tek bir ilkel değer taşıyan küçük bir heap nesnesidir: `int` için `Integer`, `double` için `Double`, `char` için `Character`, `boolean` için `Boolean` ve diğerleri. Tam sayı tutan kutu bu yüzden `Kutu<Integer>` olarak yazılır.

Programcının her seferinde ilkel değeri elle sarmalayıcıya koyması zahmetli olurdu. Java bu dönüşümü kendiliğinden yapar. Bir `int`, `Integer` beklenen bir yere verildiğinde otomatik olarak bir `Integer` nesnesine konur; buna **otomatik kutulama** (*autoboxing*) denir. `m.koy(42)` satırındaki

açıklama ("int → Integer") bunu anlatır. Tersi de olur: bir `Integer` aritmetik bir işlemde ya da bir `int` değişkene atanırken içindeki değer çıkarılır; buna **kutudan çıkarma** (*unboxing*) denir.

Örnek 1.7

`Kutu<Integer> m` kutusuna 42 konduktan sonra `int toplam = m.al() + 8;` satırı çalıştırılıyor. Hangi dönüşümler olur?

1. `m.koy(42)` : 42 bir `int` 'tir; `koy` ise `Integer` bekler. Java heap'te 42 değerli bir `Integer` nesnesi hazırlar ve `icerik` alanına onun referansını yazar (otomatik kutulama).
2. `m.al()` : `Integer` nesnesine bir referans döndürür.
3. `+ 8` : Toplama ilkel değerlerle yapılır. Java nesnenin içindeki 42'yi çıkarır (kutudan çıkarma) ve 42 + 8 = 50 hesaplanır.
4. **Atama:** 50 bir `int` 'tir ve doğrudan `toplam` 'a yazılır.

Otomatik kutulama kodu kısaltır ama iki tuzak getirir. Birincisi `null` 'dır. İçine hiçbir şey konmamış bir `Kutu<Integer>` 'ın `al()` metodu `null` döndürür. Bu değer bir `int` 'e atanmak istendiğinde Java içinden değer çıkarmaya çalışır ve `NullPointerException` fırlatır. İkincisi karşılaştırmadır:

```
Integer i1 = 127, i2 = 127;
Integer i3 = 128, i4 = 128;
System.out.println(i1 == i2);         // true
System.out.println(i3 == i4);         // false
System.out.println(i3.equals(i4));    // true
```

`Integer` bir referans tipi olduğu için `==` iki nesnenin adresini karşılaştırır. Java, −128 ile 127 arasındaki tam sayılar için hazır nesneler tutar ve kutulamada bunları yeniden kullanır. Bu yüzden 127 için iki değişken aynı nesneyi gösterir ve `==` `true` verir. 128 için iki ayrı nesne oluşur ve `==` `false` verir. `equals` ise içerikleri karşılaştırdığı için her iki durumda da doğru sonucu verir.

Sık yapılan hata

Jenerik bir yapıdan çıkan iki `Integer` 'ı `==` ile karşılaştırmak. Küçük sayılarla yapılan denemelerde kod doğru çalışır; gerçek veride 127'yi aşan bir değer geldiğinde sessizce yanlış sonuç verir. Sarmalayıcı nesneler `equals` ile ya da önce `int` 'e çevrilerek karşılaştırılır.

1.5.4 Java koleksiyonları

Java'nın standart kütüphanesindeki veri yapılarının hepsi jeneriktir. Bu yapılara topluca **koleksiyonlar** denir. En sık kullanılanlardan ikisi `ArrayList<E>` ve `Map<K,V>` 'dir. `ArrayList<E>` , `E` tipinde öğeleri sıralı tutan ve gerektikçe büyüyen bir listedir. `Map<K,V>` ise `K` tipindeki anahtarları `V` tipin-

deki değerlere eşler; iki tür parametresi alır. Aşağıdaki kod üç koleksiyonu birlikte kullanır:

```
ArrayList<String> liste = new ArrayList<>();
liste.add("Ali"); liste.add("Can"); liste.add("Ali");
HashSet<String> kume = new HashSet<>(liste);
HashMap<Integer, String> ogr = new HashMap<>();
ogr.put(101, "Ayşe");
ogr.put(102, "Mehmet");
ogr.put(101, "Zeynep");
System.out.println(liste.size() + " " + kume.size());   // 3 2
System.out.println(ogr.size() + " " + ogr.get(101));    // 2 Zeynep
System.out.println(ogr.get(103));                       // null
```

Liste aynı öğeyi birden çok kez tutabilir; üç `add` çağrısından sonra boyutu 3'tür. `HashSet` bir kümedir ve her öğeyi yalnız bir kez tutar; listeden oluşturulduğunda ikinci "Ali" elenir ve boyut 2 olur. `HashMap`'te anahtarlar benzersizdir: aynı anahtarla ikinci `put` yeni bir çift eklemez, eski değerin üzerine yazar. Bu yüzden 101 numarası artık "Zeynep"i gösterir ve eşlemede 2 çift vardır. Olmayan bir anahtar (103) için `get` `null` döndürür.

Tür parametreleri bu kodda da koruma sağlar. `ogr.put("101", "Ayşe")` yazılsaydı derleyici anahtarın `Integer` değil `String` olduğunu görüp programı derlemezdi. Bu notta yazılan kendi veri yapılarımız da aynı biçimde, jenerik sınıflar olarak yazılır.

1.6 İlkel olmayan veri yapıları

İlkel tipler tek bir sayı, tek bir karakter ya da tek bir mantıksal değer tutar. Gerçek programlar ise yüzlerce öğrenciyi, binlerce şehri, milyonlarca kelimeyi bir arada işler. Bu kesim, çok sayıda değeri belirli bir düzende bir arada tutan yapıları tanıtır.

1.6.1 Karmaşık yapılar

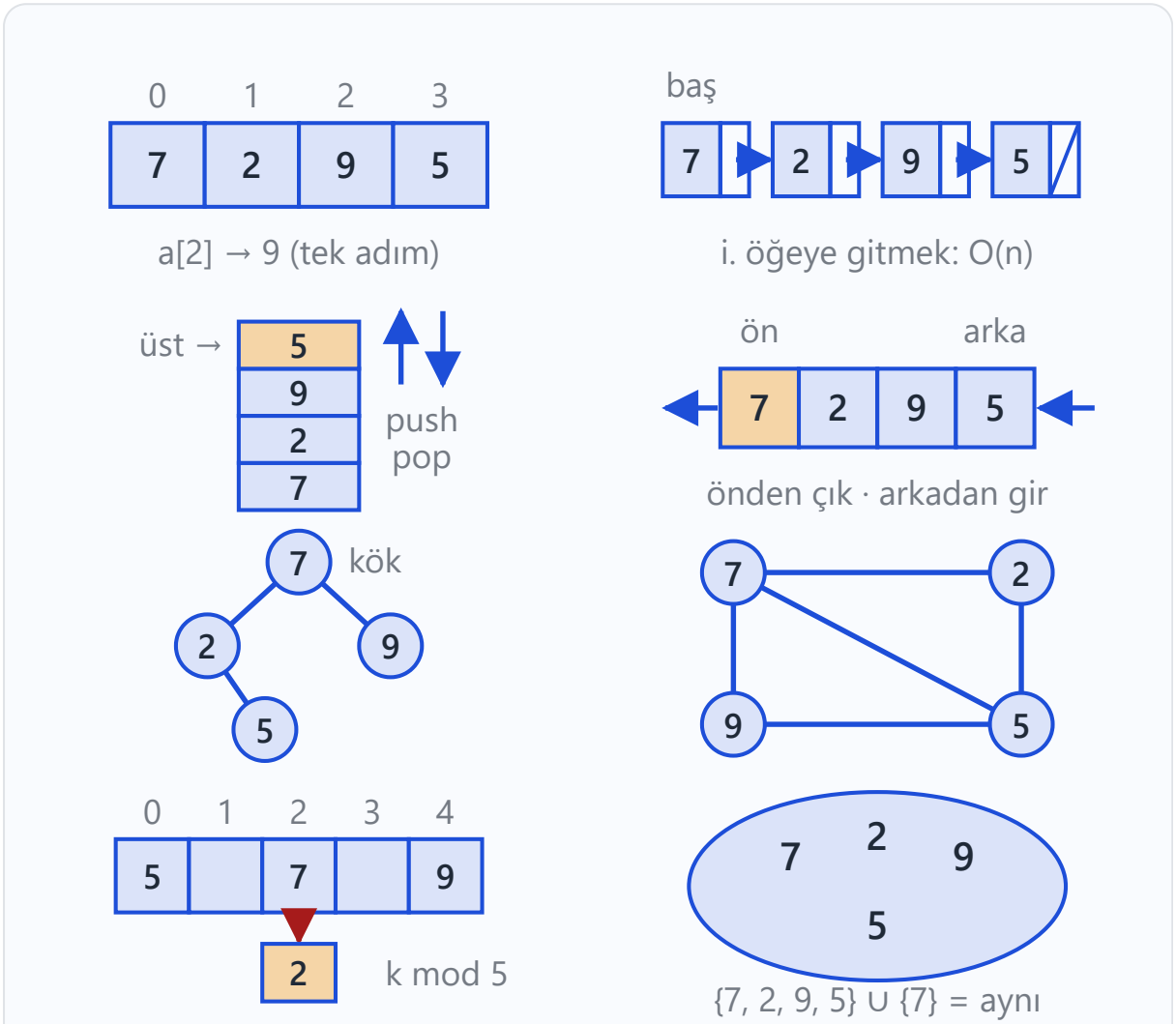
İlkel tiplerden ve başka yapılardan kurulan yapılara **ilkel olmayan** (*non-primitive*) veri yapıları denir. Bu yapılar, öğeleri arasında kurdukları ilişkiye göre kabaca şöyle gruplanır:

- **Sıralı:** diziler ve listeler. Öğeler bir sıra içindedir; her öğenin bir öncesi ve bir sonrası vardır.
- **Bağlantılı:** bağlı listeler. Sıra, öğeleri birbirine bağlayan referanslarla kurulur.
- **Hiyerarşik:** ağaçlar. Her öğenin bir üstü ve birden çok altı olabilir.
- **Ağ:** çizgeler. Her öğe istediği kadar başka öğeyle ilişkili olabilir.
- **İndislenmiş:** hash tabloları. Öğenin yeri, kendi anahtarından hesaplanır.
- Bunların yanında kümeler, eşlemeler ve daha birçok yapı.

İlkel olmayan yapıların üç ortak özelliği vardır. Birincisi, karmaşıktırlar ve birden fazla ilkel değeri, hatta farklı tiplerde değerleri bir arada içerebilirler. İkincisi, çoğu dinamik boyutludur: program çalışırken öğe eklendikçe büyür, çıkarıldıkça küçülür. Diziler bu kuralın istisnasıdır; Java'da bir dizinin uzunluğu oluşturulduğu anda belirlenir ve sonradan değişmez. Üçüncüsü, her yapı belirli işlemler için özelleştirilmiştir: biri en son ekleneni hızla verir, öteki bir anahtarı hızla bulur, bir başkası iki nokta arasındaki yolu bulmaya elverişlidir.

1.6.2 Aynı dört değer, sekiz yapı

Yapıların farkını görmenin en iyi yolu, aynı verinin her birinde nasıl durduğuna bakmaktır. Şekil 1.5, aynı 7, 2, 9, 5 değerlerini sekiz farklı düzende gösterir. Her birinin güçlü ve zayıf yönü farklıdır.



Şekil 1.5: Aynı dört değer (7, 2, 9, 5) sekiz farklı düzende; soldan sağa, yukarıdan aşağı: (a) dizi, (b) bağlı liste, (c) yığın, (d) kuyruk, (e) ağaç, (f) çizge, (g) hash tablosu, (h) küme.

(a) Dizi. Değerler ardışık hücrelerde, 0'dan 3'e kadar indislerle durur. `a[2]` tek adımda 9'u verir; 1.2.3'teki adres hesabı sayesinde indisle erişim *O(1)*'dir. Boyut ise sabittir: beşinci bir değer için yer yoktur.

(b) Bağlı liste. Her değer ayrı bir düğümdedir ve her düğüm bir sonrakini gösterir: 7 → 2 → 9 → 5 → `null`. Listenin başını gösteren tek bir referans yeterlidir. Yeni bir düğüm eklemek için yalnız birkaç okun yönü değişir, boyut esnektir. Bedeli şudur: *i*. öğeye gitmek için baştan başlayıp *i* ok izlemek gerekir, bu da *O(n)*'dir.

(c) Yığın. Değerler üst üste konmuştur; 7 en alttadır, en son eklenen 5 en üsttedir. Ekleme (`push`) ve çıkarma (`pop`) yalnız üstten yapılır. Bu düzene **LIFO** (*Last-In, First-Out*: son giren ilk çıkar) denir. Bir sonraki `pop` 5'i verir.

(d) Kuyruk. Değerler geliş sırasıyla dizilmiştir: önde 7, arkada 5. Yeni öğe arkadan girer, çıkış önden yapılır. Bu düzene **FIFO** (*First-In, First-Out*: ilk giren ilk çıkar) denir. Bir sonraki çıkarma 7'yi verir. Yığın ile kuyruk aynı dört değeri tam ters sırada bırakır.

(e) Ağaç. Değerler bir hiyerarşi içindedir. 7 köktür; 2 ve 9 onun çocuklarıdır; 5, 2'nin çocuğudur. Çocuğu olmayan 9 ve 5 birer yapraktır. Kökten herhangi bir düğüme tek bir yol vardır.

(f) Çizge. Değerler düğümlerdir ve aralarındaki çizgiler kenarlardır. Şekildeki çizgede 7, 2, 9 ve 5'in her biri iki ya da üç düğüme bağlıdır. Ağaçtan farkı, ilişkinin hiyerarşik olmak zorunda olmamasıdır: bir düğüme birden çok yoldan ulaşılabilir, kenarlar döngü oluşturabilir. Bu tür ilişkiye birçoğa-birçok ilişki denir.

(g) Hash tablosu. Değerler 5 hücreli bir diziye, yerleri bir **hash fonksiyonu** *h(k)* ile hesaplanarak konmuştur. Şekildeki fonksiyon *h(k) = k mod 5*'tir, yani değerin 5'e bölümünden kalan. 5 mod 5 = 0, 7 mod 5 = 2, 9 mod 5 = 4'tür. 2 mod 5 de 2'dir, ama 2 numaralı hücrede 7 zaten vardır. İki değerin aynı hücreye düşmesine **çakışma** (*collision*) denir ve hash tabloları çakışmaları çözmek için ek kurallar kullanır. Çakışmalar az olduğunda bir değeri aramak, hücresini hesaplayıp oraya bakmaktan ibarettir; bu yüzden hash tablosunda arama ortalama *O(1)*'dir.

(h) Küme. Bir küme yalnız tekil öğeler tutar ve öğelerin sırası önemli değildir. {7, 2, 9, 5} kümesine 7'yi yeniden eklemek kümeyi değiştirmez: {7, 2, 9, 5} ∪ {7} yine aynı kümedir. Kümenin yakın akrabası olan **eşleme** ise her tekil anahtara bir değer bağlar (anahtar → değer).

Örnek 1.8

Aynı 5 hücreli tabloya ve *h(k) = k mod 5* fonksiyonuna 7, 2, 9, 5 değerlerinin ardından 13 ve 20 eklenmek isteniyor. Hangi hücrelere düşerler?

1. **13:** 13 = 5 · 2 + 3, kalan 3. Hücre 3 boştur; 13 oraya yerleşir.
2. **20:** 20 = 5 · 4 + 0, kalan 0. Hücre 0'da 5 vardır; bu bir çakışmadır.
3. **Gözlem:** 5 hücreli tabloda 6 değer vardır ve 2 çakışma oluşmuştur (2 ile 7, 20 ile 5). Tablo doldukça çakışmalar artar. Hash tabloları bu yüzden doluluk belirli bir oranı aşınca daha büyük bir tabloya taşınır.

Sık yapılan hata

Bir yapının her işlemde en iyisi olduğunu sanmak. Dizi indisle erişimde rakipsizdir ama araya eklemede yavaştır; bağlı liste tam tersidir. Hash tablosu anahtarla aramada çok hızlıdır ama öğeleri sıralı vermez. Sekiz yapının her biri, bazı işlemleri ucuzlatmak için bazılarından vazgeçer.

1.7 Soyut veri tipleri ve veri yapıları

"Yığın" denince iki farklı şey kastedilebilir. Birincisi bir davranıştır: öğeler üstten eklenir, üstten çıkar, son giren ilk çıkar. İkincisi bu davranışın bellekte kurulmuş hâlidir: belirli bir dizi, belirli bir indis değişkeni, belirli bir kod. Bu iki düzeyi birbirinden ayırmak, veri yapılarını anlamanın ve tasarlamanın anahtarıdır.

1.7.1 Ne ile nasıl ayrımı

Tanım 1.7

Soyut veri tipi (*abstract data type*, kısaca ADT), bir veri türü üzerinde hangi işlemlerin yapılabileceğini ve bu işlemlerin nasıl davranacağını belirler; işlemlerin nasıl yapılacağıyla ilgilenmez. **Veri yapısı** ise bir soyut veri tipini uygulayan somut yapıdır: verilerin bellekte nasıl duracağını ve her işlemin hangi adımlarla yürüyeceğini belirler.

Listeler, yığınlar, kuyruklar, kümeler, eşlemeler ve çizgeler birer soyut veri tipidir. Hepsi bir davranış tarifidir. Bağlı listeler, diziler ve komşuluk matrisleri ise birer veri yapısıdır; soyut veri tiplerini bellekte kurmanın yollarıdır. ADT davranışa, veri yapısı bellek ve performans ayrıntılarına odaklanır.

Bir benzetme ayrımı netleştirir. Bir arabanın sürücüye sunduğu şey direksiyon, gaz ve fren pedalıdır; sürücü bu "arayüzü" bilerek her arabayı kullanabilir. Motorun benzinli mi elektrikli mi olduğu, frenin hangi mekanizmayla çalıştığı sürücüyü ilgilendirmez. Pedallar ve direksiyon ADT'ye, motor ve fren mekanizması veri yapısına karşılık gelir. Aynı arayüz bambaşka mekanizmalarla sağlanabilir; sürücü bunu fark etmeden araba değiştirebilir.



Şekil 1.6: Liste ADT'si (ekle, sil, al(i), boyut işlemleri) iki farklı veri yapısıyla uygulanabilir: dizi ve bağlı liste.

Şekil 1.6 bu ilişkiyi liste üzerinde gösterir. Liste ADT'si dört işlem sunar: ekle, sil, *i*. öğeyi al ve boyutu söyle. Bu dört işlem bir diziyle de, bir bağlı listeyle de sağlanabilir. Kullanıcı açısından iki gerçekleme aynı şeyi yapar. Farkları maliyettedir: dizide `al(i)` anında biter ama başa ekleme bütün öğeleri kaydırır; bağlı listede başa ekleme anında biter ama `al(i)` listeyi baştan gezer. Tablo 1.5 iki kavramı karşılaştırır.

	Soyut veri tipi (ADT)	**Veri yapısı**
Cevapladığı soru	ne yapılır?	nasıl yapılır?
Odak	işlemler ve davranış	bellek düzeni ve performans
Örnekler	liste, yığın, kuyruk, küme, eşleme, çizge	dizi, bağlı liste, komşuluk matrisi
Java'daki karşılığı	arayüz (`interface`)	sınıf (`class`)
Sayısı	bir ADT	aynı ADT için birden çok gerçekleme

Tablo 1.5: Soyut veri tipi ile veri yapısının karşılaştırması.

Sık yapılan hata

"Yığın bir veri yapısıdır, dizi de bir veri yapısıdır" deyip ikisini aynı düzeye koymak. Yığın bir ADT'dir: yalnız davranışı tarif eder. Dizi ise bir bellek düzenidir ve yığını kurmanın yollarından biridir. Günlük konuşmada iki düzey için de "veri yapısı" denir; ama bir tasarım kararı verirken önce hangi ADT'nin gerektiği, sonra onun hangi veri yapısıyla kurulacağı ayrı ayrı sorulur.

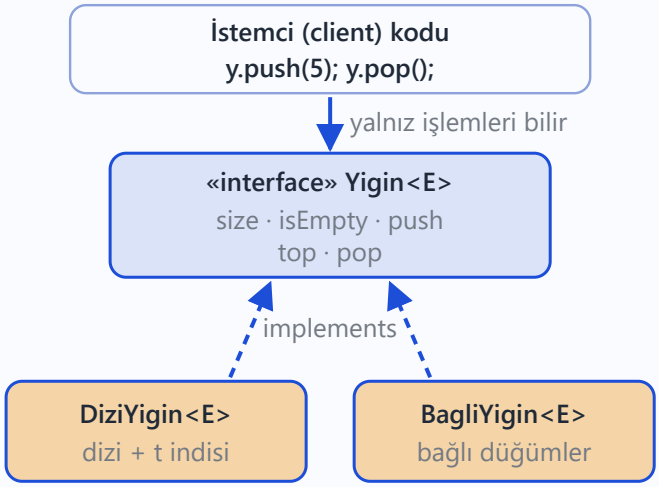
1.7.2 ADT'den Java'ya: arayüz

Java, "ne" ile "nasıl" ayrımı için iki ayrı dil öğesi sunar. **Arayüz** (*interface*), bir ADT'nin sözleşmesini yazmanın aracıdır: hangi metotların bulunacağını, her metodun ne aldığını ve ne döndürdüğünü söyler, ama metotların gövdesini içermez. Yığın ADT'sinin arayüzü şöyledir:

```
public interface Yigin<E> {
  int size();         // öğe sayısı
  boolean isEmpty();  // boş mu?
  void push(E e);     // üste ekle
  E top();            // üsttekini döndür
  E pop();            // üsttekini çıkar
}
```

Arayüz beş işlemi tanımlar. `size()` yığındaki öğe sayısını, `isEmpty()` yığının boş olup olmadığını döndürür. `push(e)`, `e` öğesini üste ekler. `top()` üstteki öğeyi yığından çıkarmadan döndürür; `pop()` ise üstteki öğeyi çıkarır ve döndürür. Arayüz jeneriktir: `E`, yığında tutulacak öğelerin tipidir ve arayüz bütün öğe tipleri için yalnız bir kez yazılır.

Arayüzde hiçbir alan ve hiçbir gövde yoktur. Öğelerin bir dizide mi, bağlı düğümlerde mi tutulacağı burada söylenmez. Bu bilgiyi bir **sınıf** verir: `implements Yigin<E>` diyen bir sınıf, arayüzdeki beş metodun hepsini gövdeleriyle yazmak zorundadır. Kısacası `interface` ADT'nin sözleşmesidir ve ne yapılacağını söyler; `class` ise veri yapısıdır ve nasıl yapılacağını söyler. Aynı ADT'nin birden çok gerçeklemesi (*implementation*) olabilir.



Şekil 1.7: İstemci kodu yalnız `Yigin<E>` arayüzünü kullanır ve yalnız işlemleri bilir. Arayüzü iki sınıf uygular: bir dizi ve `t` indisi kullanan `DiziYigin<E>` ile bağlı düğümler kullanan `BagliYigin<E>`.

Yığını kullanan koda **istemci kodu** denir. Şekil 1.7 istemcinin yalnız arayüze bağlı olduğunu gösterir. İstemci, yığını şöyle oluşturur:

```
Yigin<Integer> y = new DiziYigin<>(10);
```

Değişkenin tipi arayüzdür (`Yigin<Integer>`), oluşturulan nesne ise belirli bir sınıftandır (`DiziYigin`). Bu satırdan sonraki bütün kod `y.push(…)`, `y.pop()` gibi yalnız arayüzdeki işlemleri kullanabilir. Gerçeklemeyi değiştirmek, örneğin dizi yerine bağlı düğümler kullanan bir yığına geçmek için yalnız bu satır değişir: `new DiziYigin<>(10)` yerine `new BagliYigin<>()` yazılır. İstemci kodunun geri kalanı aynı kalır.

Sık yapılan hata

Değişkeni sınıf tipinde tanımlamak: `DiziYigin<Integer> y = …`. Program çalışır, ama istemci artık sınıfa bağlanmıştır. Sınıfa özgü bir metot (örneğin bir hata ayıklama metodu) kullanıldıysa gerçeklemeyi değiştirmek bütün istemci kodunu etkiler. Arayüz tipinde tanımlanan değişken, istemciyi yalnız sözleşmeye bağlar.

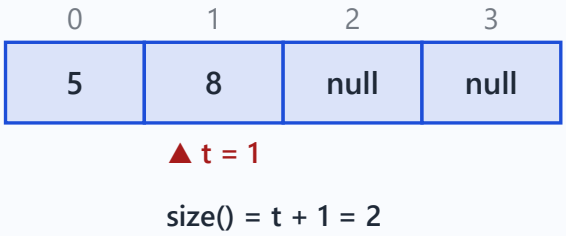
1.7.3 Bir gerçekleme: DiziYigin

Yığın ADT'sini bir diziyle kurmanın fikri basittir. Öğeler dizinin başından itibaren sırayla yerleştirilir: ilk eklenen 0. hücreye, ikincisi 1. hücreye. Yığının üstü, dolu olan son hücredir. Bu hücrenin indisini tutan bir değişken yeterlidir; ona `t` (*top*) diyelim.

```
public class DiziYigin<E> implements Yigin<E> {
  private E[] veri;         // öğeler
  private int t = -1;       // üst öğenin indisi
  public DiziYigin(int kapasite) {
    veri = (E[]) new Object[kapasite];
  }
  public int size() { return t + 1; }
  public boolean isEmpty() { return t == -1; }
  public E top() {
    if (isEmpty()) return null;
    return veri[t];
  }
  // push ve pop
}
```

Sınıfı satır satır okuyalım.

- `implements Yigin<E>` : sınıf yığın sözleşmesini kabul eder. Sınıfın tür parametresi `E` , arayüzün `E` 'sine aktarılır.
- `private E[] veri` : öğelerin tutulduğu dizi. `private` anahtar sözcüğü bu alana sınıf dışından erişilmesini engeller. İstemci diziyi doğrudan değiştiremez; yalnız arayüzdeki işlemleri kullanabilir. Böylece yığının kuralı (yalnız üstten erişim) korunur.
- `private int t = -1` : üst öğenin indisi. Boş yığında üst öğe yoktur; bu durumu −1 gösterir, çünkü −1 geçerli bir dizi indisi değildir.
- Yapıcı metot, istenen kapasitede bir dizi oluşturur. Java'da `new E[kapasite]` yazılamaz: tür parametresi program çalışırken bilinmez ve Java bilinmeyen tipte dizi oluşturamaz. Bu yüzden bir `Object` dizisi oluşturulup `(E[])` dönüşümüyle `veri` 'ye atanır. Derleyici bu satır için "denetlenmemiş dönüşüm" uyarısı verir; dizi yalnız sınıfın içinde kullanıldığı ve içine yalnız `E` tipinde öğe konduğu için bu güvenlidir.
- `size()` : öğeler 0'dan `t` 'ye kadar olan hücrelerdedir, yani `t + 1` tanedir. Boş yığında −1 + 1 = 0 verir.
- `isEmpty()` : `t == -1` ise yığın boştur.
- `top()` : yığın boşsa döndürülecek öğe yoktur ve `null` döner. Değilse üstteki öğe, `veri[t]` döner. Öğe yığından çıkarılmaz; `t` değişmez.



Şekil 1.8: Kapasitesi 4 olan bir `DiziYigin`: hücrelerde 5, 8, `null`, `null` var ve `t = 1`. Üst öğe 8'dir ve `size() = t + 1 = 2`'dir.

Şekil 1.8 bir ara durumu gösterir. Önce 5, sonra 8 eklenmiştir. `t` 1'dir ve 8'in bulunduğu hücreyi gösterir. Dizinin 2 ve 3 numaralı hücreleri boştur ve yeni öğeleri bekler.

Sık yapılan hata

`t`'yi "öğe sayısı" sanmak. `t` üst öğenin indisidir; öğe sayısı `t + 1`'dir. Bu karışıklık hem `size()`'da hem de dolu denetiminde bir eksik ya da bir fazla hatasına yol açar. Bazı kitaplar indis yerine öğe sayısını tutan bir değişken kullanır; o zaman boş yığın 0 ile gösterilir. İki yol da doğrudur, ama aynı sınıfta karıştırılmamalıdır.

1.7.4 push ve pop

Sınıfın eksik kalan iki metodu yığını değiştiren işlemlerdir. Sunum, bu iki metodu onları kullanan kısa bir istemci koduyla birlikte verir:

```
public void push(E e) {
  if (size() == veri.length)
    throw new IllegalStateException("dolu");
  veri[++t] = e;
}
public E pop() {
  if (isEmpty()) return null;
  E e = veri[t];
  veri[t--] = null;     // GC için boşalt
  return e;
}
y.push(5); y.push(8);
System.out.println(y.pop());
y.push(3);
```

`push(e)`. Önce dizinin dolu olup olmadığı denetlenir: öğe sayısı dizinin uzunluğuna eşitse yeni öğeye yer yoktur. Bu durumda metot `IllegalStateException` fırlatır; kapasitesi sabit bir yığına dolu iken eklemeye **taşma** denir. Yer varsa `veri[++t] = e` satırı çalışır. `++t` önce `t`'yi bir artırır, sonra artı-

rılmış değeri indis olarak kullanır. Yani önce artırılır, sonra yazılır. `veri[t++] = e` yazılsaydı öğe eski üstün üzerine yazılırdı.

`pop()`. Boş yığından çıkarılacak öğe yoktur; metot bu durumda `null` döndürür. Yığın boş değilse üstteki öğe `e`'ye alınır. `veri[t--] = null` satırı iki iş yapar: önce `t`'nin eski değeriyle gösterilen hücreye `null` yazar, sonra `t`'yi bir azaltır. Hücreyi boşaltmanın nedeni çöp toplayıcıdır. Hücre eski öğeyi göstermeye devam etseydi, istemci o öğeyle işini bitirmiş olsa bile nesne bellekten temizlenemezdi. Son olarak saklanan öğe döndürülür.

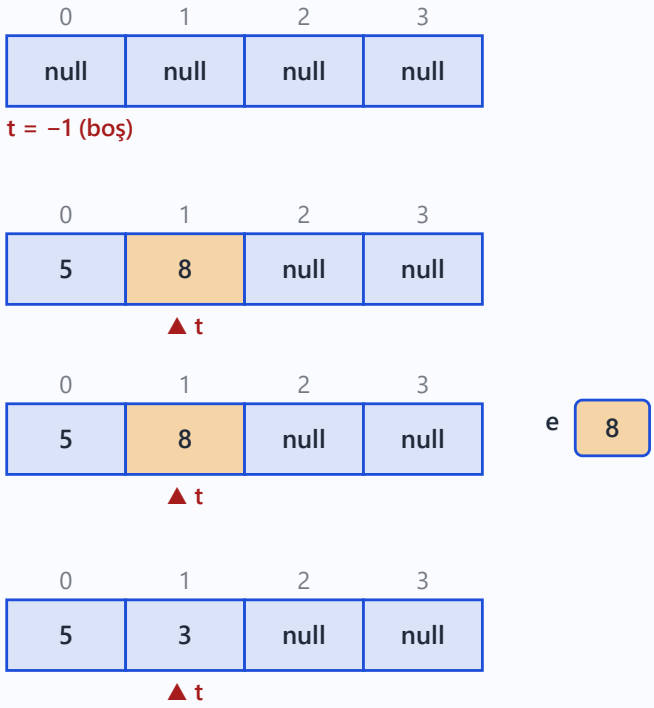
> **Sık yapılan hata**
>
> `pop`'ta önce hücreyi boşaltıp sonra öğeyi okumak: `veri[t--] = null; return veri[t];`. İlk satır hem öğeyi siler hem de `t`'yi azaltır; ikinci satır çıkarılması gereken öğeyi değil, onun altındakini döndürür. Doğru sıra şudur: önce öğeyi bir değişkene al, sonra hücreyi boşalt ve `t`'yi azalt, en son değişkeni döndür.

1.7.5 Kod izleme: DiziYigin

İstemci kodu kapasitesi 4 olan boş bir `DiziYigin` üzerinde çalışır: `push(5)`, `push(8)`, `pop()`'un sonucunu yazdırma ve `push(3)`. Tablo 1.6 her adımda çağrı yığınını, `t`'yi ve diziyi verir. Satır numaraları sunumdaki kod bloğuna göredir: `push` 1–5, `pop` 6–11, istemci 12–14.

Adım	Satır	Çağrı yığını	`t`	`veri`	Açıklama
1	12	main	−1	[null, null, null, null]	`push(5)` çağrılır
2	2	main | push(5)	−1	[null, null, null, null]	`size()` = 0 ≠ 4: dolu değil
3	4	main | push(5)	0	[5, null, null, null]	`++t`, sonra `veri[0] = 5`
4	12	main	0	[5, null, null, null]	`push(8)` çağrılır
5	2	main | push(8)	0	[5, null, null, null]	`size()` = 1 ≠ 4
6	4	main | push(8)	1	[5, 8, null, null]	`veri[1] = 8`
7	13	main	1	[5, 8, null, null]	`pop()` çağrılır
8	7	main | pop()	1	[5, 8, null, null]	`t` = 1: boş değil
9	8	main | pop()	1	[5, 8, null, null]	`e = veri[1]` = 8
10	9	main | pop()	0	[5, null, null, null]	hücre boşaltılır, `t--`
11	10	main | pop()	0	[5, null, null, null]	`return 8`
12	13	main	0	[5, null, null, null]	ekrana `8` yazılır
13	14	main	0	[5, null, null, null]	`push(3)` çağrılır
14	2	main | push(3)	0	[5, null, null, null]	`size()` = 1 ≠ 4
15	4	main | push(3)	1	[5, 3, null, null]	`veri[1] = 3`
16	14	main	1	[5, 3, null, null]	son durum

Tablo 1.6: `DiziYigin` kod izlemesi (kapasite 4).



Şekil 1.9: İzlemenin dört anı (soldan sağa, yukarıdan aşağı). (a) Başlangıç: dört hücre `null`, `t = -1`. (b) İki `push`'tan sonra [5, 8], `t = 1`. (c) `pop`'ta `e = veri[1]` = 8 alındı. (d) `push(3)`'ten sonra son durum: [5, 3], `t = 1`.

İzlemede üç noktaya dikkat etmek gerekir. Birincisi, her metot çağrısı çağrı yığınında yeni bir çerçeve açar ("main | push(5)") ve metot bitince bu çerçeve kapanır; 1.4.1'deki bellek modeli burada iş başındadır. İkincisi, `pop` en son eklenen 8'i döndürür; 5 eklenen ilk öğedir ve yerinde kalır (LIFO). Üçüncüsü, `push(3)` az önce boşaltılan 1 numaralı hücreyi yeniden kullanır. Dizi tabanlı yığında öğeler hiçbir zaman kaydırılmaz; yalnız `t` ileri geri hareket eder.

Örnek 1.9

Kapasitesi 2 olan boş bir `DiziYigin<String>`'e `push("a")`, `push("b")`, `push("c")` uygulanıyor. Ne olur? Ayrıca kapasitesi 3 olan boş bir yığında `pop()` ve `top()` ne döndürür?

1. `push("a")`: `size()` = 0 ≠ 2; `t = 0`, `veri = ["a", null]`.
2. `push("b")`: `size()` = 1 ≠ 2; `t = 1`, `veri = ["a", "b"]`. Dizi doludur.
3. `push("c")`: `size()` = 2 = `veri.length`. Metot `"dolu"` iletisiyle `IllegalStateException` fırlatır; yığın değişmez.
4. **Boş yığın:** `t = -1` olduğu için `isEmpty()` `true` verir. `pop()` ve `top()`'un ikisi de `null` döndürür.

Örneğin son adımı bir tasarım kararını gösterir: bu sınıf boş yığından çıkarmayı (**alt taşma**, *underflow*) bir hata olarak değil, `null` döndürerek bildirir. Bazı kütüphaneler bu durumda istisna fırlatır. İki yol da kullanılır, ama `null` döndüren bir yığında istemci, dönen değeri kullanmadan önce `null` olup olmadığını denetlemelidir. Ayrıca yığına öğe olarak `null` konmuşsa, `pop`'un döndürdüğü `null`'ın "yığın boştu" mu yoksa "üstte `null` vardı" mı anlamına geldiği ayırt edilemez.

1.7.6 İşlemlerin maliyeti

Bir veri yapısının işlemlerini karşılaştırmak için her işlemin ne kadar iş yaptığına bakılır. Bunun için yaygın olarak **büyük O** (*Big O*) gösterimi kullanılır. Bu gösterimin ayrıntısı burada gerekmez; şu iki anlamı bilmek yeterlidir. *O(1)*, işlemin yaptığı adım sayısının yapıdaki öğe sayısından bağımsız, sabit olduğunu söyler. *O(n)* ise adım sayısının öğe sayısı *n* ile orantılı büyüdüğünü söyler: öğe sayısı iki katına çıkınca iş de yaklaşık iki katına çıkar.

İşlem	Süre
`size`, `isEmpty`, `top`	O(1)
`push`, `pop`	O(1)
Bellek	O(N) (kapasite)

Tablo 1.7: `DiziYigin` işlemlerinin maliyeti (*N*: kapasite).

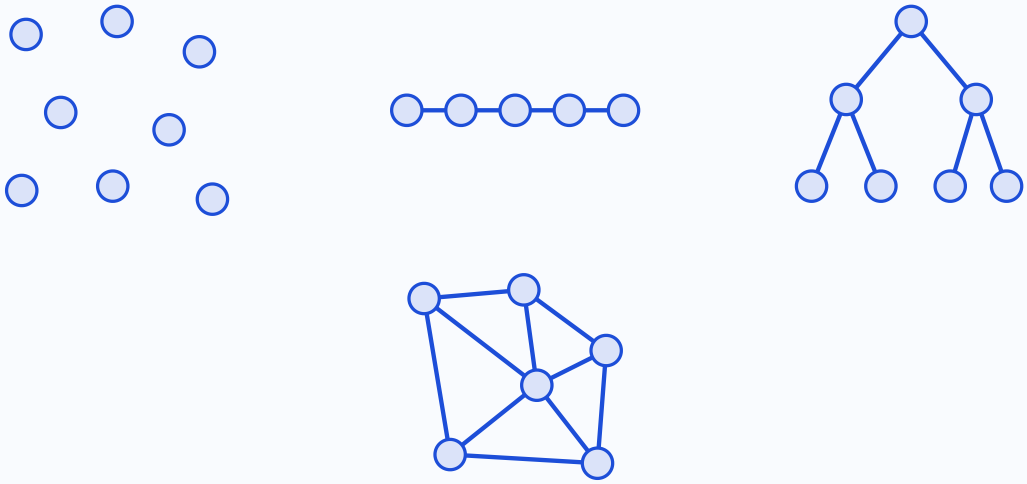
Tablo 1.7 bütün işlemlerin *O(1)* olduğunu söyler. Gerekçesi kodun kendisindedir. `size` bir toplama, `isEmpty` bir karşılaştırma, `top` bir karşılaştırma ve bir dizi erişimi yapar. `push` bir karşılaştırma, bir artırma ve bir yazma yapar. `pop` bir karşılaştırma, bir okuma, bir yazma ve bir azaltma yapar. Hiçbir metotta döngü yoktur ve dizi erişimi 1.2.3'teki adres hesabıyla tek adımda yapılır. Yığında 3 öğe de olsa 3 milyon öğe de olsa her işlem aynı sayıda adımda biter.

Bellek ise kapasiteyle orantılıdır: kapasitesi *N* olan bir yığın, içinde kaç öğe olursa olsun *N* hücrelik bir dizi ayırır. Bu, dizi tabanlı gerçeklemenin bedelidir. Kapasite sabittir ve dolunca `push` hata fırlatır. Kapasite gereğinden büyük seçilirse bellek boşa harcanır, küçük seçilirse yığın erken dolar. Bu sorunu çözmenin iki yolu vardır: dizi dolunca daha büyük bir diziye taşınmak ya da diziden vazgeçip bağlı düğümler kullanmak. İki yolda da yığın ADT'si değişmez; yalnız onu gerçekleyen sınıf değişir. 1.7.1'deki ayrımın pratikteki kazancı budur.

1.8 Verilerin düzenlenmesi ve yapı seçimi

Bir veri yapısı seçmeden önce verilerin birbiriyle nasıl bir ilişki içinde olduğunu görmek gerekir. Öğeler birbirinden bağımsız mı, bir sıra mı oluşturuyor, bir hiyerarşi mi kuruyor, yoksa karmaşık bir ağ mı? Bu sorunun cevabı, uygun yapıların listesini büyük ölçüde daraltır.

1.8.1 Dört düzen biçimi



Şekil 1.10: Verilerin dört düzenlenme biçimi (soldan sağa): (a) konumsal olmayan, birbirine bağlı olmayan dağınık öğeler; (b) sıralı, düz bir zincir; (c) ağaç, kökten dallanan bir hiyerarşi; (d) çizge, birden çok bağlantıyla birbirine bağlı öğeler.

Şekil 1.10 dört temel düzeni gösterir. Tablo 1.8 her birinin öğeler arasında kurduğu ilişkiyi ve gündelik hayattan bir örneğini verir.

Düzen	İlişki	Gündelik hayattan örnek
Konumsal olmayan	ilişkisiz	sınıftaki öğrenciler
Sıralı	bire-bir	sıra bekleyen öğrenciler
Ağaç	bire-birçok	aile ağacı
Çizge	birçoğa-birçok	yol haritası

Tablo 1.8: Verilerin düzenlenme biçimleri.

Konumsal olmayan düzende öğeler arasında bir ilişki yoktur. Sınıftaki öğrenciler bir topluluk oluşturur, ama “Ayşe'den sonraki öğrenci” diye bir kavram yoktur. Bu düzende sorulan sorular üyelikle ilgilidir: Mehmet bu sınıfta mı? Kaç öğrenci var? Küme ve eşleme bu düzene uyan yapılardır.

Sıralı düzende her öğenin en çok bir öncülü ve en çok bir ardılı vardır; ilişki bire-birdir. Kantinde sıra bekleyen öğrencilerde herkesin önünde ve arkasında birer kişi vardır, yalnız ilk ve son kişi bu kuralın istisnasıdır. Dizi, liste, bağlı liste, yığın ve kuyruk sıralı yapılardır. Aralarındaki fark, sıranın hangi uçlarından ekleme ve çıkarma yapılabildiğidir.

Ağaç düzeninde her öğenin en çok bir üstü (ebeveyni) vardır, ama birden çok altı (çocuğu) olabilir; ilişki bire-birçoktur. Aile ağacında, bir şirketin örgüt şemasında ya da bilgisayardaki klasör yapısında bu düzen görülür. Her klasörün tek bir üst klasörü vardır, ama içinde birçok alt klasör bulunabilir.

Çizge düzeninde her öğe istediği kadar öğeyle ilişkili olabilir; ilişki birçoğa-birçoktur. Yol haritasında bir şehirden birçok şehre yol gider ve her şehre birçok şehirden ulaşılır. Sosyal ağlardaki arkadaşlık ilişkileri ve bilgisayar ağları da çizgedir. Çizge en genel düzendir: sıralı düzen ve ağaç, çizgenin kısıtlanmış özel durumlarıdır.

Örnek 1.10

Aşağıdaki verilerin her biri hangi düzene uyar?

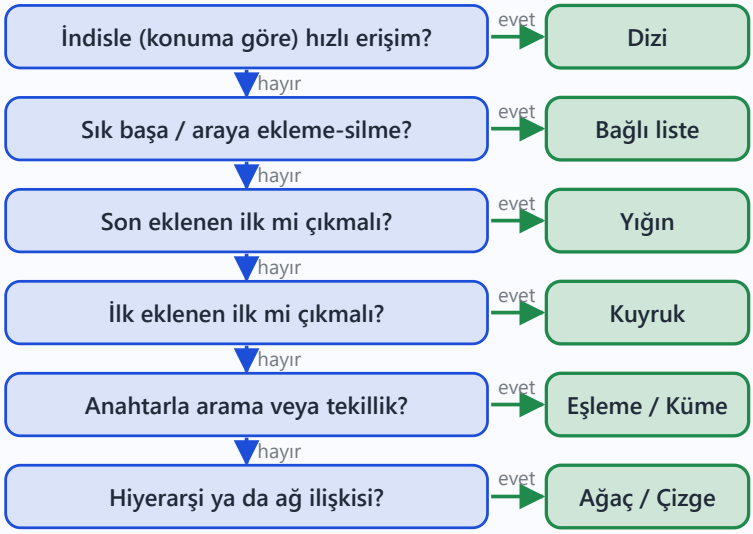
1. **Bir kitabın bölümleri, kesimleri ve alt kesimleri:** Her alt kesim tek bir kesime, her kesim tek bir bölüme aittir; bir bölümün ise birçok kesimi vardır. İlişki bire-birçoktur: ağaç. Bu notun içindekiler listesi de böyle bir ağaçtır.
2. **Bir otobüs hattının durakları:** Her durağın bir önceki ve bir sonraki durağı vardır: sıralı.
3. **Bir üniversitedeki derslerin ön koşul ilişkileri:** Bir dersin birden çok ön koşulu olabilir; bir ders de birden çok dersin ön koşulu olabilir. İlişki birçoğa-birçoktur: çizge.
4. **Bir kütüphanenin üye listesi:** Üyeler arasında bir sıra ya da hiyerarşi yoktur; önemli olan bir kişinin üye olup olmadığıdır: konumsal olmayan.

1.8.2 Veri yapısı seçimi

Doğru yapıyı seçmek için önce verilerin nasıl kullanılacağı anlaşılmalıdır. Seçimde şu sorular belirleyicidir.

- **Veri yapısı başlangıçta tamamen doldurulabilir mi?** Bütün veriler baştan biliniyorsa (örneğin bir yılın on iki ayının adları) yapı bir kez kurulur ve sonra yalnız okunur. Veriler zamanla geliyorsa yapının büyüyebilmesi gerekir.
- **Veri yapısından öğeler silinebilir mi?** Silme gerekmiyorsa yapı çok daha basit tutulabilir. Sık silme varsa, silinen öğenin bıraktığı boşluğu ucuza kapatabilen bir yapı gerekir.
- **Ekleme sırasının önemi var mı?** Öğeler geliş sırasıyla işlenecekse kuyruk, en son gelen önce işlenecekse yığın uygundur. Sıra önemsizse küme ya da eşleme düşünülebilir.
- **Veri yapısındaki öğeler sıralanabilir mi?** Öğeler büyüklüğe ya da alfabeye göre sıralanabiliyorsa, 1.2.1'deki ikiye bölerek arama gibi hızlı yöntemler kullanılabilir.
- **Öğeler belirli bir sırayla mı işlenir, yoksa rastgele erişim de gerekli mi?** “5. öğeyi ver” gibi konuma göre erişim sık gerekiyorsa dizi; yalnız baştan sona gezinmek yetiyorsa bağlı liste de iş görür.

Bu soruların ortak bir sonucu vardır: durum değişmeyecekse **statik** (*static*) yapılar, sık ekleme ve silme varsa **dinamik** yapılar tercih edilir. Statik bir yapı, örneğin sabit boyutlu bir dizi, bir kez kurulur ve en hızlı erişimi sağlar. Dinamik bir yapı, örneğin bağlı liste, değişime uyum sağlar ama erişimde bir miktar yavaştır.



Şekil 1.11: Veri yapısı seçimi için bir soru akışı. Sorular yukarıdan aşağı sorulur ve ilk "evet" cevabı bir yapıyı önerir: indisle hızlı erişim → dizi; sık başa ya da araya ekleme-silme → bağlı liste; son eklenen ilk çıkmalı → yığın; ilk eklenen ilk çıkmalı → kuyruk; anahtarla arama ya da tekillik → eşleme ya da küme; hiyerarşi ya da ağ ilişkisi → ağaç ya da çizge.

Şekil 1.11 bu soruları bir akış hâlinde toplar. Akış kesin bir kural değil, bir başlangıç noktasıdır. Gerçek problemlerde birden çok soruya "evet" denebilir ve bu durumda birden çok yapı birlikte kullanılır.

Örnek 1.11

Bir sözlük uygulaması tasarlanıyor. Kullanıcı bir kelime yazınca anlamı hemen gösterilmeli; ayrıca son aranan 20 kelime, en son aranan en üstte olacak biçimde listelenmeli. Hangi yapılar uygundur?

1. **Kelime → anlam:** İndisle erişim gerekmez (kelimenin numarası yoktur). Sık araya ekleme de asıl ihtiyaç değildir. "Anahtarla arama" sorusuna cevap evettir: anahtar kelime, değer anlamdır. Uygun yapı eşlemedir; hash tablosuyla ortalama *O(1)*'de arama yapılır.
2. **Son aranan kelimeler:** En son aranan en üstte görünmelidir; "son eklenen ilk mi çıkmalı?" sorusuna cevap evettir. Uygun ADT yığındır. 20 sınırı, kapasitesi sabit bir yapı fikrini akla getirir.
3. **İnce ayrıntı:** Sınır aşıldığında en *eski* kelimenin atılması gerekir; saf bir yığın ise yalnız üstten çıkarır. Dolayısıyla her iki uçtan da işlem yapılabilen bir yapı daha uygundur. Akıştaki ilk "evet" bir başlangıçtır; gereksinimin tamamı her zaman yeniden denetlenir.

Sık yapılan hata

Her problem için en tanıdık yapıyı, genellikle diziyi ya da `ArrayList` 'i seçmek. Bir listede "bu kelime daha önce görüldü mü?" sorusunu cevaplamak her seferinde listeyi baştan sona taramayı gerektirir. Bir milyon kelimelik bir metinde bu, 1.1.3'teki 10^{12} düzeyinde karşılaştırmaya kadar varabilir. Aynı soruyu bir küme ortalama sabit sürede cevaplar.

1.9 Temel veri yapılarına genel bakış

Önceki kesimlerde tek tek değinilen temel veri yapıları Tablo 1.9'da bir arada verilmiştir. Tablo her yapının temel fikrini ve tipik kullanımlarını özetler.

Yapı	Temel fikir	Tipik kullanım
Dizi	ardışık hücreler; indisle O(1) erişim, sabit boyut	tablo ve matris, diğer yapıların temeli
Bağlı liste	referanslarla bağlı düğümler; esnek boyut	sık ekleme ve silme
Yığın	LIFO: yalnız üstten ekle / çıkar	geri al (undo), çağrı yığını, parantez kontrolü
Kuyruk	FIFO: arkadan ekle, önden çıkar	yazdırma kuyruğu, görev sırası, BFS
Ağaç	hiyerarşi: kök, çocuk, yaprak	dosya sistemi, HTML belgesi (DOM), indisler
Öncelikli kuyruk	en öncelikli öğe hep önde (heap)	görev çizelgeleme, acil servis triyajı
Eşleme / hash tablosu	anahtar → değer; h(k) ile kovaya, ortalama O(1)	öğrenci no → kayıt, önbellek, kelime sayımı
Küme	tekil öğeler; hızlı üyelik testi	tekrarları ayıklama, birleşim ve kesişim
Çizge	düğüm + kenar; birçoğa-birçok ilişki	yol haritası, sosyal ağ, en kısa yol

Tablo 1.9: Temel veri yapıları: genel bakış.

Tablodaki kullanımların her birinin ardında yapının temel özelliği yatar. Birkaçını açalım.

Dizi öteki yapıların temelidir. Yığın, kuyruk, hash tablosu ve öncelikli kuyruk çoğu zaman içeride bir diziyle kurulur. Bir resim, bir oyun tahtası ya da bir tablo, iki boyutlu dizi (matris) olarak saklanır.

Bağlı liste, öğe sayısı sürekli değişen ve sık sık başa ya da araya ekleme yapılan durumlara uyar. Bir müzik çalarda çalma listesine şarkı eklemek ya da çıkarmak, öteki şarkıları yerinden oynatmadan yapılır.

Yığın, "en son yapılanı ilk geri al" mantığının yapısıdır. Bir metin düzenleyicide her işlem yığına konur; geri al komutu üstteki işlemi çıkarıp tersini uygular. 1.4.1'deki çağrı yığını da metot çağrılarını aynı kuralla yönetir. Parantez kontrolünde her açılan parantez yığına konur ve her kapanan parantez üstteki açılışla eşleşmelidir.

Kuyruk, adil sıranın yapısıdır. Yazıcıya gönderilen belgeler, sunucuya gelen istekler ve bir işletim sisteminde çalışmayı bekleyen görevler geliş sırasıyla işlenir. Genişlik öncelikli arama (*breadth-first search*, kısaca BFS) da sıradaki düğümü bir kuyruktan alır.

Ağaç, hiyerarşinin yapısıdır. Bilgisayardaki klasörler, bir web sayfasının HTML öğeleri (tarayıcılarda bu ağaca DOM, *Document Object Model* denir) ve veritabanlarının hızlı arama için kullandığı indisler ağaçtır.

Öncelikli kuyruk, geliş sırasına değil önceliğe göre çıkarma yapar. Acil serviste hastalar geliş sırasıyla değil, durumlarının ağırlığına göre muayene edilir. Bu yapı genellikle heap denen özel bir ağaçla kurulur.

Eşleme ve hash tablosu, bir anahtardan değere hızla ulaşmanın yoludur. Öğrenci numarasından öğrenci kaydına, bir kelimeden metinde kaç kez geçtiğine ulaşmak tipik kullanımlardır. Bir web tarayıcısının önbelleği de adres → sayfa içeriği eşlemesidir.

Küme, tekrarları ayıklamanın ve "bu öğe var mı?" sorusunu hızla cevaplamanın yapısıdır. İki kümenin birleşimi ve kesişimi gibi matematiksel işlemler de doğrudan desteklenir.

Çizge, en genel ilişkileri modeller. Bir harita uygulaması şehirleri ya da kavşakları düğüm, yolları kenar olarak tutar ve iki nokta arasındaki en kısa yolu bu çizge üzerinde arar. Kenarlara mesafe gibi sayılar yazıldığında çizge ağırlıklı çizge adını alır.

Örnek 1.12

Çok oyunculu bir çevrim içi oyun için aşağıdaki gereksinimlere uygun yapıları seçelim.

1. **Oyuncu adından oyuncu bilgisine ulaşmak:** anahtarla arama → eşleme (hash tablosu).
2. **Oyuna girmek için bekleyen oyuncular, geliş sırasıyla:** FIFO → kuyruk.
3. **Haritadaki bölgeler ve bölgeler arası geçitler:** her bölgeden birçok bölgeye geçilebilir → çizge.
4. **Bir oyuncunun envanterindeki eşya türleri, tekrarsız:** tekillik → küme.
5. **Lonca yapısı (lider, subaylar, üyeler):** hiyerarşi → ağaç.

Tek bir uygulamada beş farklı yapı kullanılmıştır. Bu durum istisna değil, kuraldır: gerçek yazılımlar her gereksinim için ayrı bir yapı seçer ve bunları birlikte kullanır.

1.10 Sık yapılan hatalar

Bu notta tek tek ele alınan hatalar, Tablo 1.10'da bir arada verilmiştir. Çoğunun kaynağı aynıdır: bir değişkenin değeri mi yoksa bir nesneye referans mı tuttuğunu gözden kaçırmak ya da bir işlemin sonucunu belirleyen tipi yanlış yerde aramak.

Hata	Sonuç	Doğrusu
`7 / 30 * 100` ile oran hesaplamak	tam sayı bölmesi: 0	önce çarp ya da `7.0` yaz
`int` aralığını aşan toplam	sessiz taşma, negatif sonuç	`long` ya da `Math.addExact`
`double`'ları `==` ile karşılaştırmak	`0.1 + 0.2 == 0.3` yanlış	farkın küçüklüğüne bak
`b = a` ile dizi kopyalamak	iki ad, tek dizi (takma ad)	`a.clone()` ya da `Arrays.copyOf`
Metotta parametreye yeni nesne atayıp çağıranı değiştirmeyi beklemek	çağıranın değişkeni değişmez	sonucu döndür ya da nesnenin içini değiştir
`String` ya da `Integer`'ı `==` ile karşılaştırmak	adres karşılaştırılır	`equals`
`Kutu<int>` yazmak	derleme hatası	`Kutu<Integer>`
`null` bir `Integer`'ı `int`'e atamak	`NullPointerException`	önce `null` denetimi
Değişkeni sınıf tipinde tanımlamak	istemci gerçeklemeye bağlanır	arayüz tipi: `Yigin<E>`
`t`'yi öğe sayısı sanmak	bir eksik / bir fazla hatası	`size() = t + 1`
`veri[t++] = e` yazmak	üstteki öğenin üzerine yazılır	`veri[++t] = e`
`pop`'ta önce boşaltıp sonra okumak	yanlış öğe döner	önce oku, sonra boşalt

Tablo 1.10: Bölümdeki sık yapılan hatalar ve doğrusu.

1.11 Özet

- **Dijital veri**, metin, sayı, görüntü ve sesin 0 ve 1'lerle temsil edilmiş biçimidir. Analog sinyal ADC ile sayılara, sayılar DAC ile yeniden sinyale çevrilir. n bit 2^n değer temsil eder. Metin karakter kodlamalarıyla (ASCII, Unicode), ondalıklı sayılar IEEE 754 ile, görüntü piksellerin RGB değerleriyle, ses örneklemeyle temsil edilir.
- **Algoritma** bir görevi yerine getiren adımların planıdır; **veri yapısı** verileri düzenleme ve saklama yoludur. Wirth'in deyişiyle: Algoritmalar + Veri Yapıları = Programlar. Her yapı bazı işlemleri ucuzlatıp bazılarını pahalılaştırır. Yapıların temel parçaları nesneler, diziler ve bağlantılardır.
- **Veri tipi**, değer kümesi ile işlemleri birlikte tanımlar. Java'nın sekiz **ilkel tipi** sabit boyutludur; sınır aşılınca değer sarar. İki `int`'in bölümü `int`'tir. **Tip güvenliği**, tiplere aykırı işlemleri derleme zamanında yakalar.
- Java'da yerel değişkenler **çağrı yığınındaki** çerçevelerde, nesneler **heap bellekte** durur. İlkel değişken değeri, **referans** değişken adresi tutar. `=` ilkel tipte değeri, referans tipte adresi kopyalar; iki referans aynı nesneyi gösterebilir (takma ad). Java parametreleri değerle geçirir. Erişilemeyen nesneleri çöp toplayıcı temizler.

- **Jenerikler** (`Kutu<T>`) her referans tipi için tek bir sınıf yazmayı ve tür hatalarını derleme zamanında yakalamayı sağlar. Tür parametresi ilkel olamaz: `int` yerine `Integer` kullanılır ve dönüşümü otomatik kutulama yapar. Java koleksiyonları jeneriktir.
- **Soyut veri tipi** ne yapılacağını (davranışı), **veri yapısı** nasıl yapılacağını (bellek düzenini) belirler. Java'da ADT bir **arayüzle**, veri yapısı onu gerçekleyen bir **sınıfla** yazılır. İstemci arayüz tipinde değişken kullanırsa gerçeklemeyi değiştirmek tek satırlık iştir.
- `DiziYigin`, yığını bir dizi ve üst indis `t` ile kurar: boş yığında `t = -1`, `size() = t + 1`. `push` önce artırıp sonra yazar, `pop` önce okuyup sonra hücreyi boşaltır. Bütün işlemler *O(1)*'dir; bellek kapasiteyle orantılıdır ve kapasite dolunca `push` hata fırlatır.
- Veriler konumsal olmayan, sıralı, ağaç ve çizge biçiminde düzenlenir. Yapı seçimini erişim biçimi, ekleme ve silme sıklığı, sıra ihtiyacı ve ilişki türü belirler; durağan veride statik, sık değişen veride dinamik yapılar seçilir.

Konu	Hatırlanacaklar
Dijital veri	metin, sayı, görüntü ve ses 0 ve 1'lerle, farklı kodlamalarla temsil edilir
Veri tipi	değer kümesi + işlemler; tip güvenliği hataları derleme zamanında yakalar
İlkel / referans	ilkel değişken değeri tutar; referans heap'teki nesneyi gösterir, atama yalnız referansı kopyalar
Jenerikler	`Kutu<T>`: her referans tipi için tek sınıf; `int` yerine `Integer`
ADT / veri yapısı	ADT ne yapılacağını (arayüz), veri yapısı nasıl yapılacağını (sınıf) belirler
Yapı seçimi	erişim biçimi, ekleme / silme sıklığı, sıra ihtiyacı ve ilişki türü belirler

Tablo 1.11: Bölümün özeti.

1.12 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

dijital veri *digital data* · algoritma *algorithm* · veri yapısı *data structure* · veri tipi *data type* · ilkel tip *primitive type* · ilkel olmayan *non-primitive* · referans *reference* · null *null* · heap bellek *heap memory* · çağrı yığını *call stack* · yığın çerçevesi *stack frame* · sığ kopya *shallow copy* · değiştirilemez *immutable* · jenerik *generic* · jenerikler *generics* · otomatik kutulama *autoboxing* · kutudan çıkarma *unboxing* · koleksiyon *collection* · soyut veri tipi *abstract data type (ADT)* · arayüz *interface* · dizi *array* · indis *index* · kapasite *capacity* · bağlı liste *linked list* · düğüm *node* · yığın *stack* · üst *top* · taşma *overflow* · alt taşma *underflow* · kuyruk *queue* · ilk giren ilk çıkar *FIFO* · ağaç *tree* · öncelikli kuyruk *priority queue* · eşleme *map* · hash tablosu *hash table* · hash fonksiyonu *hash function* · çakışma *collision* · küme *set* · çizge *graph* · statik dizi *static array* · büyük O *Big O*

1.13 Kendini Dene

1. 1000 farklı değeri temsil etmek için en az kaç bit gerekir? 200 sayısı sekiz bitle nasıl yazılır ve Java'da `(byte) 200` ifadesi hangi değeri verir?

   Cevap

   9 bit $2^9 = 512$ değer verir, yetmez; 10 bit $2^{10} = 1024$ değer verir. En az 10 bit gerekir. 200 = 128 + 64 + 8 olduğu için `11001000` yazılır. `byte` işaretlidir ve en çok 127 tutar; en soldaki bit 1 olduğu için bu dizi negatif okunur: 200 − 256 = −56.

2. `int a = 9;` için `a / 4 * 4` ve `a * 4 / 4` ifadeleri ne verir? Neden farklıdır?

   Cevap

   İlki 8, ikincisi 9 verir. İşlemler soldan sağa yapılır. İlkinde `9 / 4` tam sayı bölmesiyle 2 olur, ondalık kısım kaybolur; 2 · 4 = 8. İkincisinde önce 9 · 4 = 36 hesaplanır, sonra 36 / 4 = 9 tam bölünür ve kayıp olmaz.

3. Aşağıdaki satırlardan sonra `r` ve `r2` hangi dizileri gösterir?

```
int[] r = {3, 4, 5};
int[] r2 = r;
r2[2] = r2[0] + r2[1];
r2 = new int[]{0, 0, 0};
r2[0] = 1;
```

   Cevap

   `r`: `[3, 4, 7]`, `r2`: `[1, 0, 0]`. İkinci satır yalnız referansı kopyalar; üçüncü satır ortak dizinin 2. hücresine 3 + 4 = 7 yazar, bu yüzden `r` da değişir. Dördüncü satır `r2`'yi yeni bir diziye yöneltir; beşinci satır yalnız bu yeni diziyi değiştirir.

4. `StringBuilder sb = new StringBuilder("ab"); StringBuilder sb2 = sb; sb2.append("c");` satırlarından sonra `sb` neyi içerir? `StringBuilder` yerine `String` ve `append` yerine `sb2 = sb2 + "c"` kullanılsaydı ne değişirdi?

   Cevap

   `sb` `"abc"` içerir. `StringBuilder` değiştirilebilir bir nesnedir; `sb` ve `sb2` aynı nesneyi gösterir ve `append` o nesnenin içeriğini değiştirir. `String` ise değiştirilemez: `sb2 + "c"` yeni bir nesne oluşturur ve atama yalnız `sb2`'nin okunu ona çevirir. Bu durumda `sb` `"ab"` olarak kalırdı.

5. `Kutu<Double> d = new Kutu<>(); d.koy(2.5);` satırlarından sonra `double x = d.al() * 2;` hangi değeri verir? `d.koy(3);` satırı derlenir mi?

   Cevap

   `x` 5.0 olur: 2.5 otomatik olarak bir `Double`'a konmuş, çarpmada kutudan çıkarılmıştır. `d.koy(3)` derlenmez. 3 bir `int`'tir ve Java onu yalnız `Integer`'a kutulayabilir; `Integer` ise `Double` değildir. Doğrusu `d.koy(3.0)`'dır.

6. Kapasitesi 3 olan boş bir `DiziYigin<Integer>` üzerinde sırayla `push(4)`, `push(6)`, `pop()`, `push(1)`, `top()`, `pop()`, `push(2)`, `push(8)` çağrılıyor. Değer döndüren çağrılar ne döndürür? Sonda `t` ve `veri` nedir? Ardından `push(5)` çağrılırsa ne olur?

   Cevap

   Dönen değerler sırasıyla 6, 1, 1'dir: ilk `pop` 6'yı, `top` 1'i çıkarmadan, ikinci `pop` 1'i döndürür. Sonra 2 ve 8 eklenir. Son durum `t = 2`, `veri = [4, 2, 8]`'dir; dizi doludur. `push(5)`'te `size()` = 3 = `veri.length` olduğu için `IllegalStateException` fırlatılır.

7. Kuyruk, dizi, eşleme, bağlı liste, küme ve komşuluk matrisi kavramlarından hangileri soyut veri tipi, hangileri veri yapısıdır?

   Cevap

   Kuyruk, eşleme ve küme soyut veri tipleridir: hangi işlemlerin yapılacağını tarif ederler. Dizi, bağlı liste ve komşuluk matrisi veri yapılarıdır: verilerin bellekte nasıl tutulacağını belirlerler. Örneğin kuyruk ADT'si bir diziyle de bir bağlı listeyle de kurulabilir; çizge ADT'si bir komşuluk matrisiyle saklanabilir.

8. Aşağıdaki gereksinimler için hangi yapı uygundur? (a) Bir çağrı merkezinde bekleyen arayanlar, (b) bir şirketin örgüt şeması, (c) bir web sitesini ziyaret eden farklı kullanıcıların sayısı, (d) derslerin ön koşul ilişkileri.

   Cevap

   (a) Kuyruk: arayanlar geliş sırasıyla bağlanır (FIFO). (b) Ağaç: her çalışanın tek bir yöneticisi, her yöneticinin birçok çalışanı vardır (bire-birçok). (c) Küme: aynı kullanıcı birden çok kez ziyaret etse de bir kez sayılmalıdır; kümenin boyutu cevaptır. (d) Çizge: bir dersin birden çok ön koşulu, bir dersin de birden çok sonrakisi olabilir (birçoğa-birçok).

9. 10, 24, 3 ve 17 değerleri 7 hücreli bir tabloya *h(k) = k mod 7* ile yerleştiriliyor. Hangi hücrelere dü-

şerler? Aynı değerler 5 hücreli tabloya *k mod 5* ile yerleştirilseydi ne olurdu?

Cevap

Mod 7'de dördü de 3 numaralı hücreye düşer: 10 = 7 + 3, 24 = 21 + 3, 3 = 0 + 3, 17 = 14 + 3. İlkinden sonraki her ekleme bir çakışmadır ve arama artık sabit sürede yapılamaz. Mod 5'te hücreler 0, 4, 3, 2 olur ve hiç çakışma yoktur. Hash fonksiyonunun değerleri hücrelere dengeli dağıtması, tablonun hızı için belirleyicidir.

10. Bir istemci `Yigin<String> y = new DiziYigin<>(100);` satırıyla başlayan 500 satırlık bir program yazmıştır. Yığının kapasite sınırından kurtulmak için bağlı düğümler kullanan `BagliYigin` sınıfına geçmek istiyor. Kaç satır değişir? Değişken `DiziYigin<String>` tipinde tanımlanmış olsaydı ne olurdu?

Cevap

Yalnız ilk satır değişir: `new DiziYigin<>(100)` yerine `new BagliYigin<>()` yazılır. Program yalnız arayüzdeki beş işlemi kullandığı için geri kalan satırlar olduğu gibi çalışır. Değişken sınıf tipinde tanımlansaydı en azından tip bildiriminin de değişmesi gerekirdi. Program `DiziYigin`'e özgü bir metot kullanmışsa o satırlar da yeniden yazılmak zorunda kalırdı.

1.14 İleri okuma

- M. T. Goodrich, R. Tamassia ve M. H. Goldwasser, *Data Structures and Algorithms in Java* (6. baskı, 2014), Bölüm 1 ve 2: Java'nın temel tiplerini, referanslarını, sınıf ve arayüzlerini ve jenerikleri bu nottaki yığın örneğine yakın kodlarla anlatır.
- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 1.2 ve 1.3: veri soyutlamasını, API ile gerçekleme ayrımını ve jenerik yığın, kuyruk sınıflarını kısa, açık örneklerle işler.
- N. Wirth, *Algorithms + Data Structures = Programs* (1976): algoritma ile veri yapısının birlikte tasarlanması gerektiği fikrinin klasik kaynağıdır; ilk bölümü temel veri tiplerini ve yapılarını tarihsel bir bakışla anlatır.

1.15 Kaynaklar

1. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6. baskı, Wiley, 2014.
2. Oracle, *Java SE 17 API Specification*: `java.lang.Integer`, `java.lang.String`, `java.util.ArrayList`, `java.util.HashMap` ve `java.util.HashSet` belgeleri.
3. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011.
4. N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.

KISIM I · BÖLÜM 2

Algoritma Analizi

Bir problemi çözen tek bir program yazmak çoğu zaman yetmez. Aynı işi yapan iki program arasında seçim yapmak, yazdığımız programın büyük veriyle ne kadar süreceğini önceden kestirmek ve gerekirse daha iyisini aramak gerekir. Algoritma analizi bu soruların cevabını verir. Not, çalışma süresini saat ile ölçerek başlar ve bu yolun sınırlarını gösterir. Ardından ilkel işlemleri saymaya, oradan da büyük O, büyük Omega ve büyük Teta gösterimlerine geçer. Notun ana gövdesi şu konulardan oluşur: en iyi, ortalama ve en kötü durum ayrımı; döngülerin yapısından karmaşıklık çıkarma; doğrusal ve ikili arama; önek ortalamaları problemine iki farklı çözüm. Son kesimler bellek kullanımını, doğruluğu ve sağlamlığı ele alır.

Bu bölümün kodu: `github.com/sercankulcu/data-structures-java/Bolum02/src`

ÖĞRENME ÇIKTILARI

Ölçme ve sayma
Bir algoritmayı deneysel olarak ölçmek ve ilkel işlemlerini sayarak çözümlemek.

Asimptotik gösterim
Zaman ve alan karmaşıklığını O, Ω ve Θ gösterimleriyle ifade etmek.

Koddan karmaşıklığa
Döngü kalıplarından en iyi, ortalama ve en kötü durum maliyetini çıkarmak.

Karşılaştırma
Aynı problemi çözen algoritmaları maliyet, doğruluk ve sağlamlık açısından karşılaştırmak.

2.1 Algoritmalar ve ölçüm

Veri yapıları dersinin her konusunda aynı soru tekrar tekrar sorulur: bu işlem ne kadar sürer? Bir dizide arama, bir listeye ekleme, bir ağaçta silme… Her veri yapısı bazı işlemleri hızlı, bazılarını yavaş yapar. Hangi yapının hangi işte iyi olduğunu söyleyebilmek için önce “hızlı” ve “yavaş” sözcüklerine kesin bir anlam vermek gerekir. **Algoritma analizi** (*algorithm analysis*) bu işi yapar. Bu kesim, algoritma kavramıyla başlar ve bir algoritmanın maliyetini ölçmenin iki yolunu tanıtır: saatle ölçmek ve işlem saymak.

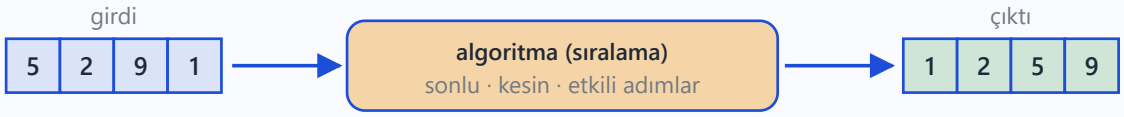
2.1.1 Algoritma nedir?

Yemek tarifini düşünelim. Tarif, malzemeleri (girdiyi) alır, sıralı adımlarla işler ve sonunda bir yemek (çıktı) verir. Adımlar açık yazılmıştır: "suyu kaynat", "pirinci ekle", "on beş dakika pişir". Tarifi izleyen herkes aynı yemeği elde eder. Tarif bir yerde biter; sonsuza kadar karıştırmaya devam etmez.

Bilgisayar bilimindeki **algoritma** (*algorithm*) aynı yapıdadır. Algoritma bir problemi çözmek için izlenen adım adım bir yöntemdir: girdi alınır, adımlar uygulanır, çıktı üretilir.

Tanım 2.1

Algoritma, bir problemin her geçerli girdisi için doğru çıktıyı üreten, sonlu sayıda kesin ve etkili adımdan oluşan yöntemdir. Adımların *sonlu* olması algoritmanın bir noktada durmasını, *kesin* olması her adımın tek bir anlamı olmasını, *etkili* olması da her adımın gerçekten yapılabilir olmasını ister.



Şekil 2.1: Algoritma girdiyi alır ve adımlarını uygulayarak çıktıyı üretir. Örnekte bir sıralama algoritması 5, 2, 9, 1 girdisini 1, 2, 5, 9 çıktısına dönüştürür.

Şekil 2.1 bu tanımı bir sıralama algoritması üzerinde gösterir. Girdi sırasız dört sayıdır; çıktı aynı sayıların küçükten büyüğe dizilmiş hâlidir. Algoritmanın içinde ne olduğu, yani sayıları hangi sırayla karşılaştırıp yer değiştirdiği, algoritmadan algoritmaya değişir. Aynı girdiyi aynı çıktıya götüren pek çok yol vardır.

Analizin çıkış noktası da budur. Aynı problemi çözen farklı algoritmalar bulunabilir ve bunlardan bazıları ötekilerden çok daha verimli çalışır. Doğru algoritmayı seçmek önemlidir, çünkü iyi bir algoritma aynı işi daha az zaman ve daha az bellekle yapar. Küçük girdilerde fark hissedilmeyebilir. Girdi büyüdükçe ise bir algoritma saniyede biterken öteki günlerce sürebilir; bu farkın sayısal örnekleri Tablo 2.7'de verilir.

Algoritmalar çözdükleri problem türüne göre gruplanır. Tablo 2.1 ders boyunca sık karşılaşılacak üç grubu özetler.

Grup	Ne yapar?	Örnekler
Sıralama	Verileri belirli bir sıraya göre düzenler.	kabarcık sıralaması, birleştirme sıralaması
Arama	Belirli bir öğeyi veri kümesinde bulur.	doğrusal arama, ikili arama
Çizge	Çizge kuramı problemlerini çözer.	en kısa yol, ağ akışı

Tablo 2.1: Bazı yaygın algoritma grupları.

Arama algoritmalarından ikisi, doğrusal ve ikili arama, bu notta ayrıntılı olarak incelenir (2.3.5–2.3.7). İkisi de aynı soruyu cevaplar: "Aranan değer dizide var mı, varsa hangi indiste?" Ama biri dizinin bütün öğelerine bakarken öteki yalnız birkaçına bakar. Bu fark, algoritma seçiminin neden önemli olduğunu en açık biçimde gösteren örneklerden biridir.

> **Sık yapılan hata**
>
> Algoritmayı programla, yani belirli bir dilde yazılmış kodla aynı şey saymak. Algoritma bir yöntemdir; Java'da, Python'da ya da kâğıt üzerinde sözde kodla ifade edilebilir. Aynı algoritmanın iki farklı dilde yazılmış hâli aynı analiz sonucunu verir. Analiz, kodun yazıldığı dile değil, yöntemin kaç adım attığına bakar.

2.1.2 İyi bir algoritmanın ölçütleri

İyi bir algoritma, problemi etkili biçimde çözen algoritmadır. "Etkili" sözcüğünü dört ölçüte ayırarak somutlaştırabiliriz. İlk ikisi maliyetle, son ikisi güvenilirlikle ilgilidir.

- **Zaman karmaşıklığı** (*time complexity*): Algoritmanın çalışma süresi ya da yaptığı işlem sayısı. Girdi büyüdükçe bu sürenin nasıl arttığı zaman karmaşıklığını belirler.
- **Alan karmaşıklığı** (*space complexity*): Algoritmanın kullandığı bellek miktarı. Bunun da girdi boyutuna göre nasıl arttığına bakılır (2.4.1).
- **Doğruluk** (*correctness*): Algoritma her geçerli girdi için doğru sonucu üretmelidir. Yanlış sonuç veren bir algoritma, ne kadar hızlı olursa olsun işe yaramaz (2.4.2).
- **Sağlamlık** (*robustness*): Algoritma beklenmeyen girdiler ve durumlarla da başa çıkabilmeli, çökmemeli ya da sessizce yanlış sonuç vermemelidir (2.4.4).

Bu ölçütler arasında bir sıra vardır. Doğruluk pazarlık konusu değildir: yanlış sonuç veren hızlı bir algoritma, doğru sonuç veren yavaş bir algoritmadan her zaman kötüdür. Doğru algoritmalar arasında seçim yapılırken zaman ve alan maliyeti devreye girer. İkisi bazen çatışır: daha fazla bellek kullanarak daha az zaman harcayan algoritmalar vardır. Önek ortalamaları örneğinde (2.3.12) ise böyle bir çatışma yoktur: hızlı çözüm aynı belleği kullanır, yalnız gereksiz işi atar.

Notun büyük kısmı zaman karmaşıklığına ayrılmıştır, çünkü veri yapısı seçiminde en sık belirleyici olan budur. Bellek için kullanılan gösterim ve yöntem aynıdır; zamanı ölçmeyi öğrenen, belleği de aynı araçlarla ölçebilir.

2.1.3 Deneysel analiz: System.nanoTime()

Bir programın ne kadar sürdüğünü öğrenmenin en doğal yolu onu çalıştırıp süreyi ölçmektir. Bu yaklaşıma **deneysel analiz** (*empirical analysis*) denir. Java'da süre ölçmek için `System.nanoTime()` kullanılır. Bu metot bir sayı döndürür. İki çağrının döndürdüğü sayıların farkı, aradan geçen süreyi nanosaniye (ns, saniyenin milyarda biri) cinsinden verir.

Aşağıdaki kod, bir dizide tekrar eden öğe olup olmadığını denetleyen `tekrarVar` metodunun süresini farklı dizi boyutları için ölçer. `tekrarVar`, dizideki bütün öğe çiftlerini karşılaştıran bir metottur; kodu ve analizi 2.3.3'te verilir. Şimdilik yalnız şunu bilmek yeterlidir: dizide tekrar yoksa metot bütün çiftlere bakar ve ancak ondan sonra `false` döndürür.

```
for (int n = 1000; n <= 64000; n *= 2) {
  int[] a = new int[n];
  for (int i = 0; i < n; i++) a[i] = i; // tekrarsız
  long bas = System.nanoTime();
  tekrarVar(a);                         // ölçülen iş
  long sure = System.nanoTime() - bas;  // ns
  System.out.println(n + " " + sure / 1e6 + " ms");
}
```

Kod satır satır şöyle çalışır:

- Dış döngü `n`'yi 1000'den başlatır ve her turda ikiyle çarpar: 1000, 2000, 4000, …, 64000. Böylece yedi farklı boyut denenir.
- `new int[n]` ile `n` hücrelik bir dizi oluşturulur. İç döngü `a[i] = i` ile diziyi 0, 1, 2, … değerleriyle doldurur. Bütün değerler farklı olduğu için dizide tekrar yoktur.
- `bas = System.nanoTime()` ölçümden hemen önceki anı kaydeder. Ardından ölçülen iş, yani `tekrarVar(a)` çağrısı yapılır.
- `System.nanoTime() - bas` geçen süreyi nanosaniye olarak verir. Bu değer 10^6'ya bölünerek milisaniyeye (ms) çevrilir ve `n` ile birlikte yazdırılır.

`System.nanoTime()`'ın döndürdüğü sayının kendi başına bir anlamı yoktur. Bu sayı günün saatini göstermez; yalnız iki okuma arasındaki fark anlamlıdır. Saatin "şu an 14.05" demesine benzeyen değer `System.currentTimeMillis()`'tir. Kısa süreleri ölçmek için `nanoTime()` daha uygundur, çünkü hem daha ince çözünürlüklüdür hem de bilgisayarın saati ayarlanınca kaymaz.

Ölçüm yaparken üç ilkeye uyulur:

1. **Ölçmek.** Ölçülecek işin hemen öncesinde ve hemen sonrasında `System.nanoTime()` okunur. Dizinin oluşturulması ve doldurulması ölçüme katılmaz; yalnız `tekrarVar` çağrısı ölçülür.
2. **Tekrarlamak.** Tek bir ölçüm güvenilir değildir. Java sanal makinesi (*Java Virtual Machine*, JVM) kodu ilk çalıştırmalarda yorumlar, sık çalışan kısımları ise sonradan makine koduna derler. Bu derlemeye anında derleme (*just-in-time compilation*, JIT) denir ve ilk ölçümleri yavaşlatır. Çöp toplayıcı ve bilgisayarda aynı anda çalışan öteki programlar da ölçüme gürültü katar. Bu yüzden her ölçüm birkaç kez yapılır ve **ortanca** (*median*) alınır. Ortanca, ölçümler küçükten büyüğe sıralandığında ortaya düşen değerdir; tek bir aşırı değerden etkilenmez.
3. **Çizmek.** Süre `n`'ye karşı bir grafiğe çizilir. Asıl bakılacak şey tek tek sürelerin büyüklüğü değil, `n` iki katına çıkınca sürenin kaç katına çıktığıdır. Bu oran bir sonraki altkesimin konusudur.

Yukarıdaki kodun bir dizüstü bilgisayarda bir kez çalıştırılmasıyla elde edilen değerler Tablo 2.2'de verilmiştir. Değerler makineden makineye, hatta aynı makinede çalıştırmadan çalıştırmaya değişir; burada yalnız ölçümün nasıl yorumlanacağını göstermek için kullanılıyor.

`n`	Süre (ms)	Oran
1.000	8,01	—
2.000	0,86	0,11
4.000	2,71	3,1
8.000	10,64	3,9
16.000	37,83	3,6
32.000	146,11	3,9
64.000	727,33	5,0

Tablo 2.2: `tekrarVar` için örnek bir ölçüm (tek çalıştırma). Son sütun, bir önceki satıra göre sürenin kaç katına çıktığını gösterir.

Tablonun ilk iki satırı tuhaftır: 1000 öğeli dizi, 2000 öğeli diziden yaklaşık dokuz kat uzun sürmüştür. Bunun nedeni algoritma değil, JIT'tir. İlk çağrıda `tekrarVar` henüz yorumlanarak çalışır; ikinci çağrıya gelindiğinde derlenmiş ve hızlanmıştır. İlk ölçüm bu yüzden "ısınma" ölçümüdür ve yorumda kullanılmaz. Sonraki satırlarda oran 3,1 ile 5,0 arasında dolaşır ve çoğunlukla 4'e yakındır. Son satırdaki 5,0, büyük dizinin işlemci önbelleğine sığmaması ya da o sırada başka bir programın çalışması gibi nedenlerle oluşan gürültüdür. Ölçümü tekrarlayıp ortancayı almak bu tür sapmaları büyük ölçüde giderir.

> **Sık yapılan hata**
>
> En kötü durumu ölçmek isterken diziyi doldurmayı unutmak. `new int[n]` ile oluşturulan Java dizisi sıfırlarla dolar. Bütün öğeleri 0 olan dizide `tekrarVar` daha ilk karşılaştırmada `a[0] == a[1]` eşitliğini bulur ve `true` döndürür. Böyle bir ölçüm, 64.000 öğeli dizi için bile mikrosaniyeler düzeyinde süre verir ve metodun çok hızlı olduğu izlenimini yaratır. En kötü durumu ölçmek için dizi tekrarsız değerlerle doldurulmalıdır; koddaki `a[i] = i` satırı bu işi yapar.

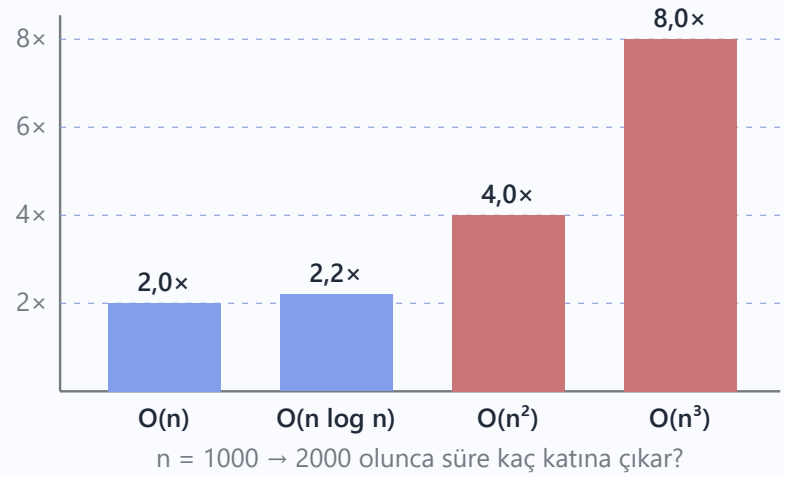
2.1.4 Katlama deneyi

Ölçülen süreler tek başına az şey söyler. "64.000 öğeli dizide 727 ms" bilgisi başka bir bilgisayarda geçerli değildir. Daha kalıcı bir bilgi, sürenin *girdiyle nasıl büyüdüğüdür*. Bunu bulmanın pratik yolu **katlama deneyidir** (*doubling experiment*): girdi boyutu her adımda iki katına çıkarılır ve ardışık iki sürenin oranına bakılır.

Çalışma süresini *T(n)* ile gösterelim. Sürenin yaklaşık olarak $T(n) \approx c \cdot n^k$ biçiminde olduğunu varsayalım. Burada *c* makineye bağlı bir sabit, *k* ise algoritmaya bağlı bir üstür. Girdi iki katına çıkınca:

$$T(2n) / T(n) \approx c \cdot (2n)^k / (c \cdot n^k) = 2^k$$

Sabit *c* sadeleşir ve geriye yalnız 2^k kalır. Oran makineden bağımsızdır. Hızlı bir bilgisayarda *c* küçülür, ama oran değişmez. Oran 2 ise *k = 1*'dir: süre doğrusal büyür. Oran 4 ise *k = 2*'dir: süre karesel büyür. Oran 8 ise *k = 3*'tür: süre kübik büyür.



Şekil 2.2: *n* 1000'den 2000'e çıkınca beklenen süre oranları. *O(n)* için 2, *O(n log n)* için yaklaşık 2,2, $O(n^2)$ için 4, $O(n^3)$ için 8 kat.

Şekil 2.2 dört büyüme biçimi için beklenen oranları gösterir. *O(...)* gösterimi 2.2.1'de tanımlanır; burada "süre yaklaşık bu fonksiyon gibi büyür" diye okunabilir. Dikkat çeken sütun *n log n*'dir. Logaritma çarpanı *n* iki katına çıkınca biraz büyür; bu yüzden oran 2'den biraz fazladır:

$$(2000 \cdot \log_2 2000) / (1000 \cdot \log_2 1000) = 2 \cdot 10{,}966 / 9{,}966 \approx 2{,}2$$

Bu küçük fark, katlama deneyinde *n* ile *n log n*'yi ayırmanın zor olduğunu da gösterir. Ölçüm gürültüsü 2,0 ile 2,2'yi kolayca karıştırabilir.

Örnek 2.1

Tablo 2.2'deki ölçümleri katlama deneyi gözüyle yorumlayalım ve 128.000 öğeli dizi için süreyi tahmin edelim.

1. **Isınma ölçümü.** İlk satır (1000 öğe) JIT etkisi taşıdığı için kullanılmaz.
2. **Oranlar.** Kalan oranlar 3,1; 3,9; 3,6; 3,9 ve 5,0'dır. Hepsi 2'den çok uzak, 8'den de uzaktır; 4'ün çevresinde toplanır.

3. **Üs.** $2^k \approx 4$ olduğundan $k = 2$'dir. Süre yaklaşık $c \cdot n^2$ biçimindedir; büyüme karesel.
4. **Tahmin.** 128.000, 64.000'in iki katıdır. Karesel büyümede süre 4 katına çıkar. Son ölçüm 727 ms olduğuna göre tahmin $4 \cdot 727 \approx 2909$ ms, yani yaklaşık 3 saniyedir. Gürültüden arınmış bir tahmin için oranı kesin 4 almak yeterlidir; tek tek oranların ortalamasını kullanmak gürültüyü tahmine taşır.

Sonuç, `tekrarVar`'ın kodundan çıkarılacak $n(n - 1)/2$ karşılaştırma sayısıyla uyumludur (2.3.3): karşılaştırma sayısı n^2 gibi büyür.

Sık yapılan hata

Oranı sürelerin farkıyla karıştırmak. 8.000'den 16.000'e geçişte süre 10,64 ms'den 37,83 ms'ye çıkar. Fark 27 ms'dir ve kendi başına bir şey anlatmaz; bilgi veren, oranın yaklaşık 3,6 olmasıdır. Katlama deneyinde her zaman bölme yapılır, çıkarma değil.

2.1.5 Deneysel analizin sınırları

Deneysel analiz somuttur ve gerçek bir programın gerçek bir makinedeki davranışını gösterir. Yine de üç önemli sınırı vardır (Goodrich vd., Kesim 4.1):

- **Sonuçlar ortama bağlıdır.** Süre; işlemciye, belleğe, JVM sürümüne ve o anda çalışan öteki programlara bağlıdır. İki algoritmayı karşılaştırmak için ikisi aynı makinede, aynı koşullarda ölçülmelidir. Başka birinin makinesinde ölçülmüş bir süreyle kendi ölçümümüzü karşılaştırmak anlamsızdır.
- **Yalnız denenen girdiler hakkında bilgi verir.** Sınırlı sayıda girdi denenebilir. Algoritmayı yavaşlatan önemli bir girdi türü denemelerin dışında kalabilir. `tekrarVar`'ı yalnız sıfırlarla dolu dizilerle ölçen biri, metodun en kötü durumunu hiç görmez.
- **Algoritmanın tamamen kodlanmasını gerektirir.** Ölçmek için önce çalışan bir program gerekir. Oysa çoğu zaman karar, kod yazılmadan önce verilmelidir: on farklı yöntemi kodlayıp ölçtükten sonra birini seçmek pahalıdır.

Bu sınırları aşmanın yolu, süreyi ölçmek yerine algoritmanın attığı adımları *saymaktır*. Sayma yöntemi donanımdan bağımsızdır, bütün girdiler için geçerli bir sonuç verir ve kodu çalıştırmadan, hatta yalnız sözde koddan yapılabilir. Deneysel analiz bu yöntemin yerini tutmaz; ama onun sonucunu sınamak için iyi bir araçtır. Örnek 2.1, ölçümle saymanın aynı sonuca vardığı bir durumdu.

2.1.6 İlkel işlemleri sayma: maksimum()

Saymaya başlamak için neyin sayılacağına karar vermek gerekir. Bir Java satırı makinede birkaç komuta dönüşebilir ve bu komutların süreleri işlemciden işlemciye değişir. Bu ayrıntılarla uğraşmamak için bir basitleştirme yapılır: bazı temel işlemlerin her birinin sabit sürede çalıştığı kabul edilir ve yalnız kaç kez çalıştıkları sayılır.

Tanım 2.2

İlkel işlem (*primitive operation*), süresi girdi boyutuna bağlı olmayan ve bir sabitle sınırlanabilen temel işlemdir. Bir değişkene değer atama, iki değeri karşılaştırma, bir aritmetik işlem (toplama, çarpma …), dizinin bir hücresine indisle erişme, bir metodu çağırma ve metottan dönme ilkel işlemlerdir.

Bir algoritmanın ilkel işlem sayısı, onun çalışma süresiyle orantılıdır. Her ilkel işlem en çok bir sabit süre alıyorsa, toplam süre de en çok bu sabitle işlem sayısının çarpımıdır. Sabit makineden makineye değişir; işlem sayısı ise değişmez. Böylece makineden bağımsız bir ölçü elde edilir.

İlk örnek, bir dizinin en büyük öğesini bulan `maksimum` metodudur:

```
int maksimum(int[] a) {
  int enBuyuk = a[0];
  for (int i = 1; i < a.length; i++) {
    if (a[i] > enBuyuk)
      enBuyuk = a[i];
  }
  return enBuyuk;
}
```

Metot, dizinin ilk öğesini geçici en büyük olarak alır. Sonra diziyi 1. indisten sona kadar tarar. Her öğeyi o ana kadarki en büyükle karşılaştırır; öğe daha büyükse `enBuyuk`'u günceller. Döngü bitince `enBuyuk` bütün dizinin en büyüğüdür.

Şimdi her ifadenin kaç kez çalıştığını sayalım. Dizinin uzunluğu n olsun. Satır numaraları, metot başlığı 1. satır olacak biçimde verilmiştir.

Satır	İfade	Kaç kez?
2	`enBuyuk = a[0]`	1
3	`i = 1`	1
3	`i < a.length`	n
4	`a[i] > enBuyuk`	n − 1
5	`enBuyuk = a[i]`	0 … n − 1
3	`i++`	n − 1
7	`return`	1

Tablo 2.3: `maksimum` metodunda ilkel işlemlerin çalışma sayıları (n = `a.length`).

Sayıların gerekçesi şöyledir:

- Döngüden önceki atama (`enBuyuk = a[0]`) ve döngü değişkeninin ilk değeri (`i = 1`) birer kez çalışır.
- Döngü gövdesi `i` = 1, 2, …, *n − 1* için çalışır; bu *n − 1* turdur. Gövdedeki karşılaştırma ve tur sonundaki `i++` bu yüzden *n − 1*'er kez çalışır.
- Döngü koşulu `i < a.length` , gövdeden bir kez fazla sınanır. *n − 1* kez doğru çıkar ve gövde çalışır; son bir kez, `i = n` iken yanlış çıkar ve döngü biter. Toplam *n* sınama.
- Güncelleme satırı (`enBuyuk = a[i]`) yalnız karşılaştırma doğru çıkınca çalışır. Kaç kez çalışacağı dizinin içeriğine bağlıdır: hiç çalışmayabilir, en çok *n − 1* kez çalışır.

Güncelleme sayısını *g* ile gösterirsek toplam işlem sayısı şudur:

$$1 + 1 + n + (n - 1) + g + (n - 1) + 1 = 3n + 1 + g, \quad 0 \le g \le n - 1$$

Buradan iki uç durum çıkar. **En iyi durumda** `a[0]` dizinin en büyüğüdür ve hiçbir güncelleme yapılmaz: *g = 0*, toplam *3n + 1*. **En kötü durumda** dizi artan sıralıdır; her yeni öğe öncekilerin hepsinden büyük olduğu için her turda güncelleme yapılır: *g = n − 1*, toplam *3n + 1 + n − 1 = 4n*.

Analizin vardığı sonuç, sayıların kendisinden daha önemlidir. En iyi durumda da en kötü durumda da işlem sayısı *n* ile doğru orantılı büyür. *n* iki katına çıkınca işlem sayısı da yaklaşık iki katına çıkar. 3 ya da 4 katsayısı ve sondaki +1 gibi sabitler ise makineye göre zaten değişen ayrıntılardır. Bu yüzden atılırlar ve iki durum için de aynı sonuç yazılır: `maksimum` 'un çalışma süresi *Θ(n)*'dir. *Θ* gösteriminin kesin anlamı 2.2.2'de verilir.

> **Sık yapılan hata**
>
> Döngü koşulunun gövdeden bir kez fazla çalıştığını unutmak. `for (int i = 1; i < n; i++)` döngüsünde gövde *n − 1* kez çalışır, ama koşul *n* kez sınanır. Son sınama döngüyü bitiren sınamadır. Bu fark sonucun büyüme hızını değiştirmez, ama tam sayım isteyen sorularda yanlış sayıya götürür.

Bir noktaya dikkat etmek gerekir: hangi işlemlerin ayrı sayılacağı bir uzlaşmadır. Örneğin `enBuyuk = a[i]` satırı aslında iki iş yapar: dizi hücresini okur ve değeri atar. Bunu iki işlem saymak toplamı *3n + 1* yerine başka bir doğrusal ifadeye çevirir, ama büyüme hızı yine doğrusal kalır. Sabitlerin atılmasının bir nedeni de budur: sayım kuralları değişince değişen şey yalnız sabitlerdir.

2.1.7 Kod izleme: maksimum(a)

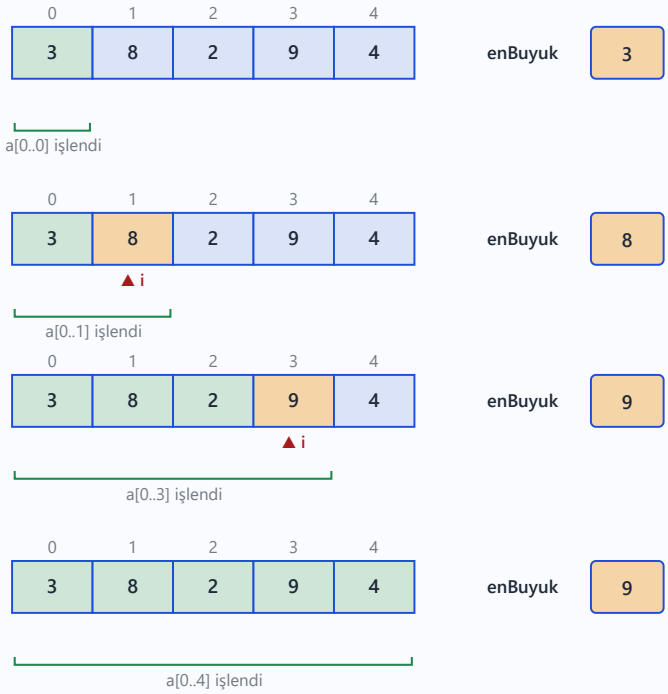
Sayımı somut bir dizi üzerinde izleyelim. Girdi `a = {3, 8, 2, 9, 4}` , yani *n = 5*. Tablo 2.4 her adımda değişkenlerin durumunu verir.

Adım	`i`	Koşul `i < 5`	`a[i] > enBuyuk`	`enBuyuk`
başlangıç	—	—	—	3

Adım	`i`	Koşul `i < 5`	`a[i] > enBuyuk`	`enBuyuk`
1. tur	1	doğru	8 > 3 doğru	8
2. tur	2	doğru	2 > 8 yanlış	8
3. tur	3	doğru	9 > 8 doğru	9
4. tur	4	doğru	4 > 9 yanlış	9
çıkış	5	yanlış	—	9 döner

Tablo 2.4: `maksimum({3, 8, 2, 9, 4})` izlemesi.

İzlemenin ayrıntıları şunlardır. Başlangıçta `enBuyuk = a[0] = 3`'tür; ilk öğe geçici en büyüktür. 1. turda `a[1] = 8`, 3'ten büyüktür ve `enBuyuk` 8 olur. 2. turda `a[2] = 2`, 8'den küçüktür; değişiklik olmaz. 3. turda `a[3] = 9 > 8` olduğu için yeni en büyük 9'dur. 4. turda `a[4] = 4`, 9'dan küçüktür. `i` 5 olunca koşul yanlış çıkar, döngü biter ve metot 9 döndürür.



Şekil 2.3: `maksimum` izlemesinin dört anı (soldan sağa, yukarıdan aşağı). (a) `enBuyuk = a[0] = 3`. (b) 1. turda 8 bulunur. (c) 3. turda 9 bulunur. (d) Döngü bitmiştir; `a[0..4]` işlenmiş ve sonuç 9'dur. Alttaki yeşil çizgi, o ana kadar işlenen kısmı gösterir.

Sayımı tabloyla doğrulayalım. Koşul 5 kez sınanmıştır (4 doğru, 1 yanlış), karşılaştırma 4 kez yapılmış, `i++` 4 kez çalışmıştır. Güncelleme iki kez olmuştur (8 ve 9 için), yani $g = 2$. Toplam işlem sayısı $3 \cdot 5 + 1 + 2 = 18$'dir. Bu değer en iyi durum sınırı 16 ile en kötü durum sınırı 20 arasındadır.

Şekil 2.3'ün alt kısmındaki yeşil çizgi, algoritmanın doğruluğu hakkında da bir şey söyler. Her turun başında `enBuyuk`, o ana kadar işlenen kısmın en büyüğüdür. Döngü bitince işlenen kısım bütün dizi olduğu için sonuç doğrudur. Bu gözlem, 2.4.3'te döngü değişmezi adıyla bir ispata dönüşür.

2.2 Asimptotik gösterim

`maksimum` için *3n + 1* ile *4n* arasında bir işlem sayısı bulduk ve sonunda yalnız "*n* ile orantılı" sonucunu tuttuk. Bu ayıklama işi her analizde tekrarlanır. Onu düzenli ve kesin biçimde yapmanın aracı **asimptotik** (*asymptotic*) gösterimdir. "Asimptotik" sözcüğü, girdi boyutu çok büyüdüğünde ne olduğuna bakıldığını anlatır. Küçük girdilerdeki ayrıntılar bir yana bırakılır; büyük girdilerdeki büyüme hızı karşılaştırılır.

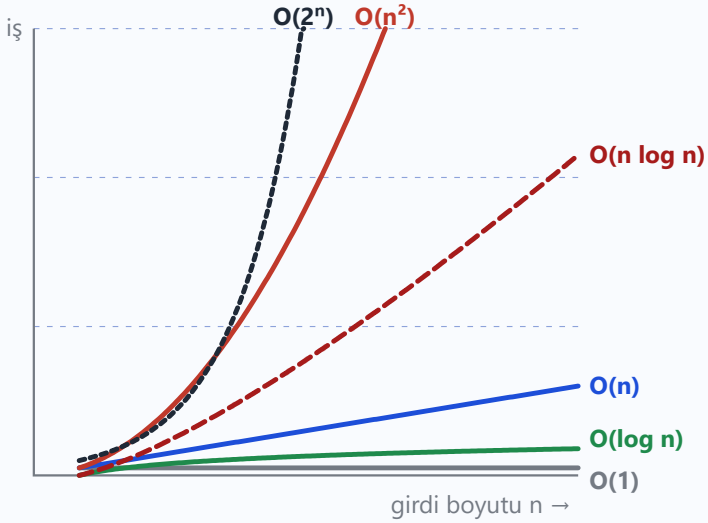
2.2.1 Zaman karmaşıklığı ve büyük O

Zaman karmaşıklığı, bir algoritmanın ne kadar sürede çalıştığını girdi boyutu *n*'ye bağlı bir fonksiyonla ifade eder. Girdi boyutu probleme göre seçilir: dizide öğe sayısı, dizgide karakter sayısı, çizgede düğüm ve kenar sayısı. Bu fonksiyonun tam biçimini bilmek gerekmez; nasıl büyüdüğünü bilmek yeterlidir.

Büyük O (*Big O*) gösterimi, çalışma süresi için bir *üst sınır* verir. "Bu algoritma *O(n)*'dir" demek, çalışma süresinin büyük *n* için *n*'nin bir sabit katını aşmadığı anlamına gelir. Büyük O çoğunlukla en kötü durum için kullanılır, çünkü en kötü durumun üst sınırı bütün girdiler için bir garanti verir. Ama bu bir alışkanlıktır, tanımın parçası değildir (2.2.5).

İki sık görülen sınıfın anlamı şudur:

- ***O(1)***: İşlem sayısı girdi boyutuyla büyümez. Dizinin ilk öğesine erişmek, dizi bir milyon öğeli de olsa üç öğeli de olsa aynı sayıda adım sürer. *O(1)*, farklı girdilerde sürenin birebir aynı olduğunu söylemez; yalnız *n*'den bağımsız bir sabitle sınırlandığını söyler.
- ***O(n)***: Girdi büyüdükçe süre doğrusal artar. *n* iki katına çıkınca süre de en çok yaklaşık iki katına çıkar. Diziyi baştan sona bir kez taramak bu sınıftadır.



Şekil 2.4: Girdi boyutu *n* arttıkça *1*, *log n*, *n*, *n log n*, n^2 ve 2^n fonksiyonlarının büyümesi (*1 ≤ n ≤ 12*, dikey eksen 60'ta kesilmiştir). 2^n ve n^2 hızla yükselirken *log n* ve *1* neredeyse düz kalır.

Şekil 2.4 altı temel büyüme biçimini aynı eksen üzerinde gösterir. *n* yalnız 12'ye kadar çizildiği hâlde farklar açıktır. 2^n eğrisi *n = 6* civarında grafiğin üstünden çıkar; n^2 eğrisi *n ≈ 7,7*'de çıkar. *n log n*, *n* doğrusundan belirgin biçimde yukarıdadır. *log n* ise 12'de bile 3,6 civarındadır. Büyük girdilerde bu farkların ne kadar açıldığı Tablo 2.7'de sayılarla görülür.

Büyük O ile yazarken yalnız *en hızlı büyüyen terim* tutulur. Sabit çarpanlar ve daha yavaş büyüyen terimler atılır:

$$3n^2 + 5n + 2 \to O(n^2) \;\cdot\; 4n + 100 \to O(n) \;\cdot\; 7 \to O(1)$$

Neden atılabildiklerini ilk örnek üzerinde görelim. *n = 10* için $3n^2 = 300$, $5n = 50$, sabit terim 2'dir; karesel terim toplamın %85'ini oluşturur. *n = 1000* için $3n^2 = 3.000.000$, $5n = 5000$'dir; karesel terim toplamın %99,8'inden fazlasıdır. *n* büyüdükçe küçük terimlerin payı sıfıra yaklaşır. Sabit çarpan 3 ise makineye bağlıdır ve büyüme hızını değiştirmez: *n* iki katına çıkınca $3n^2$ de n^2 de dört katına çıkar.

Sık yapılan hata

O(1)'in her zaman hızlı olduğunu sanmak. Sabit süre, büyük bir sabit olabilir. Her çağrıda 1000 adım atan bir *O(1)* işlem, *n = 50* için 50 adım atan bir *O(n)* işlemden yavaştır. Asimptotik gösterim büyük *n* için karşılaştırma yapar; küçük girdilerde sabitler belirleyici olabilir.

2.2.2 O, Ω ve Θ'nın kesin tanımı

"Büyük *n* için bir sabit katını aşmaz" sözünü kesin bir tanıma çevirelim. Burada *f(n)* incelenen algoritmanın işlem sayısı, *g(n)* ise karşılaştırma için seçilen basit bir fonksiyondur (*n*, n^2, *log n* gibi).

Tanım 2.3

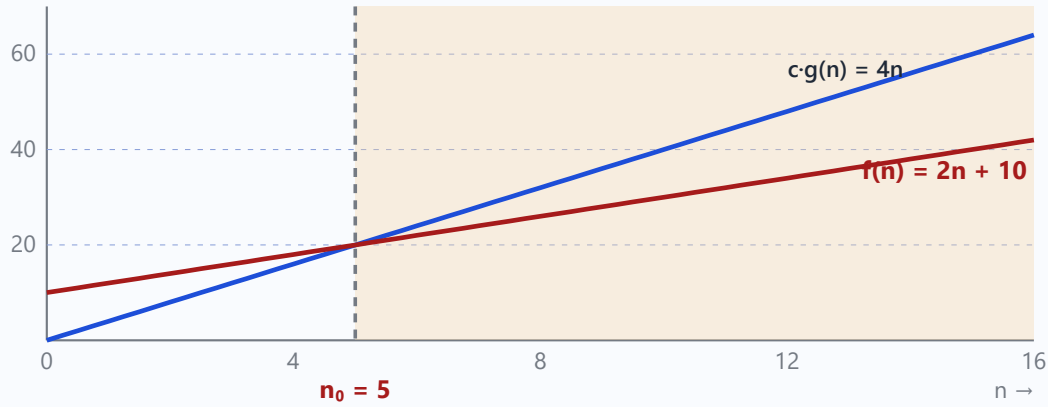
$f(n) = O(g(n))$ olması için, öyle bir $c > 0$ sabiti ve $n_0 \geq 1$ tam sayısı bulunmalıdır ki her $n \geq n_0$ için $f(n) \leq c \cdot g(n)$ olsun. Bu durumda $g(n)$, $f(n)$ için bir **üst sınırdır**: yeterince büyük n için f, g'den daha hızlı büyümez.

$f(n) = \Omega(g(n))$ (büyük Omega) olması için aynı koşulda eşitsizlik ters yöndedir: her $n \geq n_0$ için $f(n) \geq c \cdot g(n)$. Bu durumda $g(n)$ bir **alt sınırdır**: f, g'den daha yavaş büyümez.

$f(n) = \Theta(g(n))$ (büyük Teta) olması için ikisi birden sağlanmalıdır: $f(n) = O(g(n))$ ve $f(n) = \Omega(g(n))$. Başka bir deyişle, öyle $c_1, c_2 > 0$ ve n_0 vardır ki her $n \geq n_0$ için $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ olur. Bu bir **sıkı sınırdır**: f ile g aynı hızda büyür.

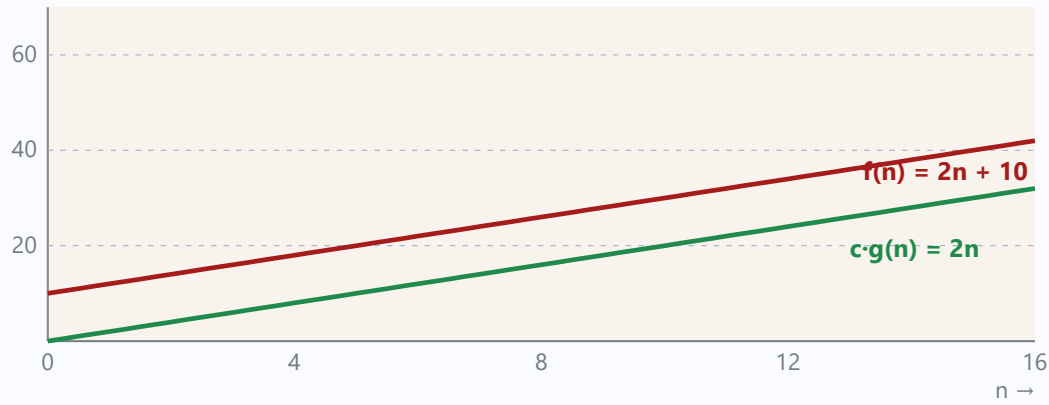
Tanımın üç parçası da gereklidir. c sabiti, sabit çarpanların neden önemsiz olduğunu kesinleştirir: $3n$ ile n arasındaki fark, $c = 3$ seçilerek kapatılır. n_0 eşiği, küçük girdilerin neden önemsiz olduğunu kesinleştirir: eşitsizliğin yalnız bir noktadan sonra sağlanması yeter. "Her $n \geq n_0$ için" ifadesi de sınırın tek tük noktalarda değil, eşikten sonra *her yerde* geçerli olmasını ister.

Üç tanımı tek bir fonksiyon üzerinde görelim: $f(n) = 2n + 10$.



Şekil 2.5: Büyük O. $n \geq n_0 = 5$ için $f(n) = 2n + 10$ doğrusu $4n$ doğrusunun altında kalır; bu yüzden $f(n) = O(n)$'dir ($c = 4$). Taralı bölge, eşitsizliğin istendiği $n \geq n_0$ aralığıdır.

Büyük O. Şekil 2.5 iki doğruyu gösterir. $n = 0$'da f 10'dan, $4n$ ise 0'dan başlar; $4n$ daha dik olduğu için bir noktada f'yi yakalar. Kesişme noktasını bulmak için $2n + 10 = 4n$ denklemini çözeriz: $n = 5$. $n \geq 5$ için $2n + 10 \leq 4n$ sağlanır. Böylece $c = 4$, $n_0 = 5$ ile $f(n) = O(n)$ gösterilmiş olur. $n < 5$ için f daha büyüktür; örneğin $n = 2$'de $f = 14$, $4n = 8$'dir. Tanım bu küçük değerlerle ilgilenmez.



Şekil 2.6: Büyük Omega. Her $n \geq 1$ için $2n + 10 \geq 2n$; bu yüzden $f(n) = \Omega(n)$'dir. f doğrusu her yerde $2n$ doğrusunun üstündedir.

Büyük Omega. Alt sınır için $c = 2$ seçmek yeterlidir. $2n + 10 \geq 2n$ eşitsizliği, 10 pozitif olduğu için her n'de doğrudur (Şekil 2.6). Burada $n_0 = 1$ alınabilir; eşik gerekmez. $f(n) = \Omega(n)$ sonucu, f'nin n'den daha yavaş büyümediğini söyler.



Şekil 2.7: Büyük Teta. $n \geq 5$ için $2n \leq f(n) \leq 4n$; f iki doğru arasına sıkışır ve $f(n) = \Theta(n)$ olur.

Büyük Teta. İki sonucu birleştirince $n \geq 5$ için $2n \leq 2n + 10 \leq 4n$ elde edilir (Şekil 2.7). f, eğimleri 2 ve 4 olan iki doğru arasına sıkışmıştır. Bu durumda $f(n) = \Theta(n)$ yazılır: f'nin büyüme hızı tam olarak n'dir.

Üç gösterim arasındaki fark, bir sınav notu benzetmesiyle kolayca anlaşılır. “Notum en çok 90” bir üst sınırdır (O); “notum en az 70” bir alt sınırdır (Ω). İkisi birlikte notu daraltır. Asimptotik dünyada “en çok” ve “en az” sabit çarpanlara kadar söylenir. Alt ve üst sınır aynı fonksiyonla verilebiliyorsa büyüme hızı kesin olarak bilinir (Θ).

Bir algoritma için hangisinin yazılacağı, ne bilindiğine bağlıdır. O en sık görülendir, ama tek başına gevşek olabilir. $2n + 10 = O(n^2)$ ifadesi de doğrudur, çünkü $n \geq 5$ için $2n + 10 \leq n^2$'dir. Yine de yararsızdır; gerçek büyüme hızını gizler. Bu yüzden analizde mümkün olan en sıkı sınır verilir. Sıkı sınır bilini-

yorsa Θ yazmak en açık ifadedir.

Sık yapılan hata

$f(n) = O(g(n))$ yazısındaki "=" işaretini gerçek eşitlik gibi okumak. Bu ifade "f, $O(g)$ sınıfındadır" demektir. $n = O(n^2)$ doğrudur, ama buradan $O(n^2) = n$ sonucu çıkmaz. Eşitlik simetriktir, bu gösterim değildir.

2.2.3 Tanımı kullanmak: c ve n_0 bulma

Bir ifadenin büyük O sınıfını göstermek için tanımdaki c ve n_0 değerlerini bulmak gerekir. Bunlar tek değildir; koşulu sağlayan herhangi bir çift yeter. İki yaygın yol vardır: her terimi en büyük terimle sınırlamak ya da eşitsizliği doğrudan çözmek.

Örnek 2.2

$f(n) = 3n^2 + 5n + 2$ fonksiyonunun $\Theta(n^2)$ olduğunu gösterelim.

1. **Üst sınır, birinci yol (terimleri büyütmek).** $n \geq 1$ için $n \leq n^2$ ve $1 \leq n^2$'dir. Bu yüzden $5n \leq 5n^2$ ve $2 \leq 2n^2$ olur. Toplarsak $f(n) \leq 3n^2 + 5n^2 + 2n^2 = 10n^2$. Böylece $c = 10$, $n_0 = 1$ ile $f(n) = O(n^2)$.
2. **Üst sınır, ikinci yol (daha küçük c).** $c = 4$ deneyelim: $3n^2 + 5n + 2 \leq 4n^2$ ancak $n^2 - 5n - 2 \geq 0$ ise doğrudur. Bu ikinci derece ifadenin pozitif kökü $(5 + \sqrt{33})/2 \approx 5{,}37$'dir. Demek ki $n \geq 6$ için eşitsizlik sağlanır. Denetim: $n = 6$'da $f(6) = 108 + 30 + 2 = 140 \leq 4 \cdot 36 = 144$; $n = 5$'te ise $f(5) = 102 > 100$. Böylece $c = 4$, $n_0 = 6$ de geçerli bir çifttir.
3. **Alt sınır.** $5n + 2$ pozitif olduğundan her $n \geq 1$ için $f(n) \geq 3n^2$'dir. $c = 3$, $n_0 = 1$ ile $f(n) = \Omega(n^2)$.
4. **Sonuç.** $n \geq 1$ için $3n^2 \leq f(n) \leq 10n^2$ olduğundan $f(n) = \Theta(n^2)$'dir.

İki yol arasındaki fark ödünleşimdir: c küçüldükçe n_0 büyür. Hangisinin seçildiği sonucu değiştirmez; tanım yalnız *bir* çiftin varlığını ister.

Örnekteki birinci yol bütün polinomlara uygulanabilir. Katsayıları negatif olmayan d. dereceden bir polinomun her terimi $n \geq 1$ için n^d'nin bir katıyla sınırlanır; katsayıların toplamı c olarak seçilir. Alt sınır için de en yüksek dereceli terimin katsayısı yeter. Bu gözlem aşağıdaki teoremde toplanır.

Teorem 2.1 (Polinomların büyüme hızı)

$f(n) = a_d n^d + a_{d-1} n^{d-1} + \ldots + a_1 n + a_0$ bir polinom, $a_d > 0$ ve öteki katsayılar negatif olmayan sayılar olsun. Bu durumda $f(n) = \Theta(n^d)$'dir. Gerekçe: $n \geq 1$ için $a_d n^d \leq f(n) \leq (a_d + a_{d-1} + \ldots + a_0) \cdot n^d$.

Teorem negatif katsayılı polinomlar için de geçerlidir (yalnız $a_d > 0$ olsun), ama o durumda n_0 daha dikkatli seçilir. Örneğin $n(n-1)/2 = n^2/2 - n/2$ ifadesi $\Theta(n^2)$'dir: $n \geq 2$ için $n/2 \leq n^2/4$ olduğundan $n^2/4 \leq n(n-1)/2 \leq n^2/2$ sağlanır. Bu ifade, `tekrarVar` analizinde yeniden karşımıza çıkar.

> **Sık yapılan hata**
>
> c'yi n'ye bağlı seçmek. "$n^2 \leq c \cdot n$ için $c = n$ seçelim, o hâlde $n^2 = O(n)$" akıl yürütmesi yanlıştır. Tanımdaki c bir sabittir; n büyüdükçe değişemez. Hangi sabit seçilirse seçilsin $n > c$ olduğu anda $n^2 > c \cdot n$ olur. Bu yüzden n^2, $O(n)$ değildir.

2.2.4 Hesap kuralları

Her analizde tanıma dönüp c ve n_0 aramak yorucu olur. Tanımdan çıkan birkaç kural, işin çoğunu kısaltır:

- **Sabit çarpanlar atılır.** $k > 0$ bir sabitse $k \cdot f(n)$, $f(n)$ ile aynı sınıftadır. $5n$ de $n/2$ de $\Theta(n)$'dir.
- **Toplamda en büyük terim kalır.** $f_1 = O(g_1)$ ve $f_2 = O(g_2)$ ise $f_1 + f_2 = O(\max(g_1, g_2))$. Ardışık iki kod bloğunun maliyeti bu kuralla birleşir: $O(n) + O(n^2) = O(n^2)$.
- **Çarpımda sınırlar çarpılır.** $f_1 = O(g_1)$ ve $f_2 = O(g_2)$ ise $f_1 \cdot f_2 = O(g_1 \cdot g_2)$. İç içe döngülerin maliyeti bu kuralla bulunur: dış döngü n tur, iç döngü her turda $\log n$ tur dönüyorsa toplam $O(n \log n)$'dir.
- **Logaritmanın tabanı önemsizdir.** $\log_a n = \log_b n \,/\, \log_b a$ olduğundan farklı tabanlı iki logaritma arasında yalnız sabit bir çarpan vardır. Örneğin $\log_2 n = \log_{10} n \,/\, \log_{10} 2 \approx 3{,}32 \cdot \log_{10} n$. Bu yüzden taban yazılmaz ve yalnız $O(\log n)$ denir.
- **Geçişlilik.** $f = O(g)$ ve $g = O(h)$ ise $f = O(h)$'dir. Aynısı Ω ve Θ için de geçerlidir.

Logaritma kuralı ilk bakışta şaşırtıcı gelebilir. $n = 1.000.000$ için $\log_2 n \approx 19{,}9$, $\log_{10} n = 6$'dır; sayılar farklıdır. Ama oran her n için aynıdır (yaklaşık 3,32). n bin katına çıktığında ikisi de bin kat artmaz; her biri n'den bağımsız sabit bir *miktar* artar: $\log_2$ yaklaşık 10, $\log_{10}$ tam 3. Sabit çarpan farkı, büyüme sınıfını değiştirmez.

Üstel fonksiyonlarda durum farklıdır. Tabanı değiştirmek bir sabit çarpan değil, büyüme hızında gerçek bir fark yaratır: $3^n \,/\, 2^n = 1{,}5^n$ oranı sınırsız büyür. Bu yüzden 3^n, $O(2^n)$ değildir. Üste eklenen bir sabit ise yalnız sabit çarpan getirir: $2^{n+3} = 8 \cdot 2^n = \Theta(2^n)$.

> **Sık yapılan hata**
>
> $\log_2 n$ ile $\log_{10} n$'yi farklı karmaşıklık sınıfları saymak. Aralarındaki fark yaklaşık 3,32'lik sabit bir çarpandır; ikisi de $O(\log n)$'dir. Bu durum logaritmalara özgüdür. Üslerde taban fark yaratır: 2^n ile 3^n farklı sınıflardadır.

2.2.5 En iyi, ortalama ve en kötü durum

`maksimum` örneğinde aynı n için farklı girdilerin farklı sayıda işlem gerektirdiğini gördük: azalan bir dizide $3n + 1$, artan bir dizide $4n$. Bu farklılık çoğu algoritmada çok daha büyüktür. Bu yüzden "algoritmanın çalışma süresi" derken hangi girdiden söz ettiğimizi de söylememiz gerekir. Üç bakış açısı vardır.

Tanım 2.4

Girdi boyutu n sabit tutulduğunda:

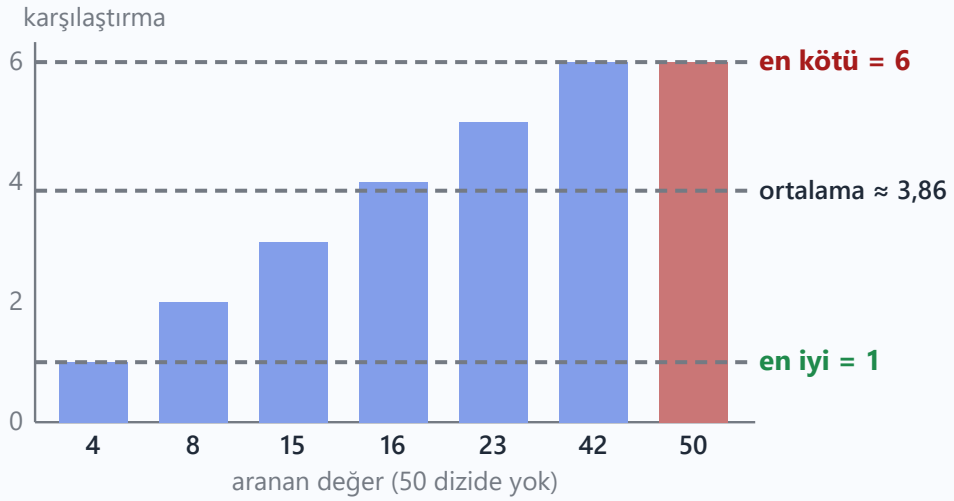
- **En iyi durum** (*best case*), boyutu n olan girdiler arasında en az işlem gerektiren girdinin maliyetidir.
- **En kötü durum** (*worst case*), boyutu n olan girdiler arasında en çok işlem gerektiren girdinin maliyetidir.
- **Ortalama durum** (*average case*), girdilerin belirli bir olasılık dağılımına göre seçildiği varsayımı altında beklenen maliyettir.

Tablo 2.5, üç asimptotik sınırı ve üç durumu bir arada özetler. Tablonun üst kısmı "ne tür bir sınır?", alt kısmı "hangi girdi için?" sorusunu cevaplar. İki soru birbirinden bağımsızdır.

Kavram	Ne söyler / ne sorar?	Doğrusal arama örneği
$O(g(n))$: üst sınır	Çalışma miktarı, yeterince büyük n için $g(n)$'den hızlı büyümez.	—
$\Omega(g(n))$: alt sınır	Çalışma miktarı, yeterince büyük n için $g(n)$'den yavaş büyümez.	—
$\Theta(g(n))$: sıkı sınır	Üst ve alt sınır aynıdır; büyüme hızı $g(n)$ ile aynı sınıftadır.	—
En iyi durum	En elverişli geçerli girdide kaç işlem gerekir?	İlk öğe hedefse $\Theta(1)$
Ortalama durum	Bir girdi dağılımı varsayımı altında beklenen maliyet nedir?	Hedefin konumu eşit olasılıklıysa $\Theta(n)$
En kötü durum	Her geçerli girdi için garanti edilen üst maliyet nedir?	Hedef sondaysa ya da yoksa $\Theta(n)$

Tablo 2.5: Asimptotik sınırlar ve durum analizleri; sağ sütunda doğrusal arama örneği.

Doğrusal arama, dizinin öğelerine baştan başlayarak tek tek bakar ve aranan değeri bulunca durur (kodu 2.3.5'te). Küçük bir örnekle üç durumu hesaplayalım. Dizi `[4, 8, 15, 16, 23, 42]`, yani $n = 6$.



Şekil 2.8: `[4, 8, 15, 16, 23, 42]` dizisinde doğrusal aramanın her olası hedef için yaptığı karşılaştırma sayısı. 50 dizide yoktur. En iyi durum 1, en kötü durum 6 karşılaştırmadır; yedi hedefin eşit olasılıklı olduğu varsayımıyla ortalama yaklaşık 3,86'dır.

Örnek 2.3

1. **Her hedefin maliyeti.** 4 aranırsa ilk karşılaştırmada bulunur: 1. 8 için 2, 15 için 3, 16 için 4, 23 için 5, 42 için 6 karşılaştırma yapılır. Dizide olmayan 50 için altı öğenin hepsine bakılır ve hiçbiri tutmaz: 6 karşılaştırma.
2. **En iyi durum.** En az karşılaştırma, hedef ilk öğe olduğunda yapılır: 1. Bu sayı n'den bağımsızdır; en iyi durum $\Theta(1)$'dir.
3. **En kötü durum.** En çok karşılaştırma, hedef son öğe olduğunda ya da dizide hiç olmadığında yapılır: 6, yani n. En kötü durum $\Theta(n)$'dir.
4. **Ortalama durum.** Yedi olası hedefin (altı öğe ve dizide olmayan bir değer) eşit olasılıkla arandığını varsayalım. Ortalama, karşılaştırma sayılarının toplamının 7'ye bölümüdür: $(1 + 2 + 3 + 4 + 5 + 6 + 6) / 7 = 27 / 7 \approx 3{,}86$.

Varsayım değişirse ortalama da değişir. Hedefin her zaman dizide olduğu ve altı konumun eşit olasılıklı olduğu varsayılırsa ortalama $(1 + 2 + \ldots + 6)/6 = 21/6 = 3{,}5$ olur. Genel olarak, n konumun eşit olasılıklı olduğu durumda ortalama $(1 + 2 + \ldots + n)/n = (n + 1)/2$'dir. Bu da $\Theta(n)$'dir: ortalamada dizinin yaklaşık yarısına bakılır, ama yarısı da n ile doğru orantılıdır.

Üç durumdan hangisinin raporlanacağı amaca bağlıdır:

- **En kötü durum** bir garanti verir. “Bu işlem hiçbir girdide $4n$ adımı geçmez” demek, sistemi tasarlayan kişiye güvenilir bir bilgi verir. Hesaplaması da genellikle kolaydır. Bu yüzden en sık raporlanan budur.
- **Ortalama durum** pratikte ne beklendiğini anlatır, ama bir olasılık dağılımı varsayar. Varsayım gerçeğe uymazsa sonuç yanıltıcı olur. Hesaplaması da çoğu zaman en kötü durumdan zordur.

- **En iyi durum** tek başına yanıltıcıdır. Hemen her arama algoritması, hedefin ilk bakılan yerde olduğu şanslı bir girdide hızlıdır. Bu bilgi algoritmanın genel davranışı hakkında hemen hiçbir şey söylemez.

> **Sık yapılan hata**
>
> "O en kötü durumdur, Ω en iyi durumdur" diye düşünmek. O, Ω ve Θ bir fonksiyonun büyümesi hakkında sınır gösterimleridir. En iyi, ortalama ve en kötü durum ise hangi girdinin maliyetinin incelendiğini belirler. İki kavram birbirinden bağımsızdır ve her durum için üç gösterim de kullanılabilir. Doğrusal aramanın en iyi durumu $\Theta(1)$'dir; en kötü durumu $\Theta(n)$'dir; algoritmanın genel çalışma süresi için $O(n)$ ve $\Omega(1)$ yazılabilir.

2.2.6 Amortize analiz

Bazı veri yapılarında işlemlerin çoğu ucuzdur, ama arada bir pahalı bir işlem gelir. Bu durumda en kötü durum analizi gereğinden kötümser bir tablo çizer. **Amortize analiz** (*amortized analysis*), tek bir işlemin değil, uzun bir işlem dizisinin toplam maliyetine bakar ve bu toplamı işlem sayısına böler.

Standart örnek, dolunca büyüyen dizidir. Java'daki `ArrayList` öğelerini bir dizide tutar. Sona ekleme çoğu zaman tek adımdır: boş hücreye yazılır. Dizi dolunca ise iki kat büyük yeni bir dizi ayrılır ve bütün öğeler oraya kopyalanır. Bu tek ekleme $O(n)$ sürer.

> **Örnek 2.4**
>
> Kapasitesi 1 olan boş bir büyüyen diziye 16 öğe ekleyelim. Dizi dolunca kapasite iki katına çıkar.
>
> 1. **Büyütmeler.** Kapasite 1, 2, 4, 8 iken dolar. Büyütmeler 2., 3., 5. ve 9. eklemelerde yapılır ve sırasıyla 1, 2, 4 ve 8 öğe kopyalanır. Son kapasite 16'dır; 16. ekleme diziyi tam doldurur ama yeni büyütme gerektirmez.
> 2. **Toplam iş.** 16 yazma ve $1 + 2 + 4 + 8 = 15$ kopyalama: toplam 31 adım.
> 3. **Ekleme başına.** $31 / 16 \approx 1{,}94$ adım. Tek tek eklemelerden biri (9. ekleme) 8 kopyalama yapmıştır, ama ortalama 2'nin altında kalır.
>
> Genel durumda da hesap aynıdır. n eklemede kopyalanan öğe sayısı $1 + 2 + 4 + \ldots$ toplamıdır. Bu toplamın son terimi n'den küçüktür; bu yüzden toplam $2n$'den azdır. Yazmalarla birlikte toplam iş $3n$'den azdır. Ekleme başına maliyet sabittir: sona ekleme **amortize $O(1)$**'dir.

Amortize analiz ortalama durum analizinden farklıdır. Ortalama durum bir olasılık varsayımı yapar ve beklenen maliyeti hesaplar. Amortize analizde olasılık yoktur: her işlem dizisi için, en kötüsü dahil, toplam maliyet $3n$'den azdır. Bu yine bir garantidir; yalnız tek işlem için değil, işlem dizisi için verilir.

Sık yapılan hata

Amortize *O(1)*'i "her işlem *O(1)*" diye okumak. Büyütme yapan tek bir ekleme yine *O(n)* sürer. Gerçek zamanlı bir sistemde, örneğin her işlemin belirli bir süre içinde bitmesi gereken bir denetleyicide, bu ara sıra gelen gecikme sorun olabilir. Amortize sınır yalnız uzun işlem dizilerinin toplamı hakkında bir şey söyler.

2.2.7 Büyük O sınıfları

Algoritma analizinde sonuçlar çoğunlukla birkaç standart sınıftan birine düşer. Tablo 2.6 bu sınıfları yavaştan hızlıya doğru büyüme sırasıyla verir.

Gösterim	Adı	Örnek	Neden?
O(1)	sabit	dizinin ilk öğesine erişme	dizinin boyutu ne olursa olsun erişim süresi sabit
O(log n)	logaritmik	sıralı dizide ikili arama	her adımda arama alanı yarıya iner
O(n)	doğrusal	diziyi baştan sona tarama	her öğe bir kez işlenir
O(n log n)	doğrusal logaritmik	birleştirme sıralaması	böl ve fethet: *log n* düzey, her düzeyde *n* iş
O(n^2)	karesel	bütün öğe çiftlerini karşılaştırma	iç içe iki döngü
O(2^n)	üstel	bütün alt kümeleri bulma	*n* öğeli kümenin 2^n alt kümesi var

Tablo 2.6: Sık görülen büyük O sınıfları.

Her sınıfın arkasında bir sezgi yatar. *O(log n)* algoritmalar problemi her adımda sabit bir oranda küçültür; *n* milyon olsa bile 20 civarında adımda biter. *O(n)* algoritmalar her öğeye sabit sayıda bakar; girdiyi okumak bile *n* adım sürdüğü için pek çok problemde bundan iyisi mümkün değildir. *O(n log n)*, iyi sıralama algoritmalarının sınıfıdır. *O(n^2)*, "her öğeyi her öğeyle karşılaştır" yaklaşımının doğal sonucudur. *O(2^n)* ise bütün olasılıkları tek tek denemenin bedelidir; *n*'ye bir eklemek süreyi ikiye katlar.

Birleştirme sıralamasındaki "*log n* düzey" ifadesi biraz açıklama ister. Bu algoritma diziyi ikiye böler, iki yarıyı ayrı ayrı sıralar ve sonra birleştirir. Yarılar da aynı biçimde bölünür. *n* öğeli dizi, tek öğeli parçalara inene kadar $\log_2 n$ kez ikiye bölünebilir. Her düzeydeki birleştirmeler toplam *n* öğeye dokunur. *log n* düzey çarpı *n* iş, *n log n* eder.

2.2.8 n büyüdükçe ne olur?

Sınıflar arasındaki farkın ne kadar büyük olduğunu en iyi sayılar anlatır. Tablo 2.7 her fonksiyonun üç farklı *n* değerindeki değerini verir.

Karmaşıklık	*n = 10*	*n = 100*	*n = 1.000*
$\log_2 n$	3,3	6,6	10

Karmaşıklık	$n = 10$	$n = 100$	$n = 1.000$
n	10	100	1.000
$n \log_2 n$	33	664	9.966
n^2	100	10.000	1.000.000
2^n	1.024	$\approx 1{,}3 \cdot 10^{30}$	$\approx 1{,}1 \cdot 10^{301}$

Tablo 2.7: Büyüme fonksiyonlarının değerleri (yuvarlanmış).

Tablo sütun sütun okunduğunda iki şey dikkat çeker. Birincisi, $\log_2 n$ neredeyse hiç büyümez: n yüz kat artınca yalnız 3,3'ten 10'a çıkar. İkincisi, 2^n $n = 10$'da henüz makul görünür (1.024), ama $n = 100$'de 31 basamaklı bir sayıdır.

Örnek 2.5

Saniyede 10^9 işlem yapan bir bilgisayar, $n = 100$ için 2^n adımlık bir algoritmayı ne kadar sürede bitirir?

1. **Adım sayısı.** $2^{100} \approx 1{,}27 \cdot 10^{30}$.
2. **Saniye.** $1{,}27 \cdot 10^{30} / 10^9 = 1{,}27 \cdot 10^{21}$ saniye.
3. **Yıl.** Bir yıl yaklaşık $365{,}25 \cdot 24 \cdot 3600 \approx 3{,}16 \cdot 10^7$ saniyedir. $1{,}27 \cdot 10^{21} / 3{,}16 \cdot 10^7 \approx 4 \cdot 10^{13}$ yıl.

Evrenin yaşı yaklaşık $1{,}4 \cdot 10^{10}$ yıldır. Bu süre, evrenin yaşının yaklaşık üç bin katıdır. Aynı bilgisayar $n^2 = 10.000$ adımı 10 mikrosaniyede bitirir.

Bu örnek, daha hızlı donanımın neden üstel bir algoritmayı kurtaramayacağını da gösterir. Bilgisayar bin kat hızlansa bile süre ancak $4 \cdot 10^{10}$ yıla iner. Hızlanmanın kazandırdığı şey çok küçüktür: bin kat hız, 2^n algoritmada aynı sürede çözülebilecek n'yi yalnız 10 artırır ($2^{10} \approx 1000$). Karesel bir algoritmada ise aynı hızlanma, çözülebilecek n'yi yaklaşık 32 katına çıkarır ($\sqrt{1000} \approx 31{,}6$). Büyük girdilerde algoritma seçimi, donanım seçiminden çok daha etkilidir.

2.3 Koddan karmaşıklığa

Asimptotik gösterim elimizdeyken asıl işe geçebiliriz: bir kod parçasına bakıp karmaşıklığını söylemek. Çoğu zaman her ilkel işlemi tek tek saymak gerekmez. Kodun maliyetini belirleyen, döngülerin kaç kez döndüğüdür. Döngüsüz bir kod parçası sabit sayıda işlem yapar ve $O(1)$'dir. Döngülerin yapısı ise birkaç kalıba ayrılır. Bu kesim önce bu kalıpları tanıtır, sonra onları gerçek metotlar üzerinde uygular: toplam, doğrusal ve ikili arama, tekrar denetimi ve önek ortalamaları.

2.3.1 Döngü kalıpları

Bir döngünün maliyeti, gövdenin kaç kez çalıştığıyla gövdenin bir kez çalışmasının maliyetinin çarpımıdır. Gövde sabit sayıda işlem yapıyorsa döngünün maliyeti tur sayısıyla orantılıdır. Tablo 2.8 en sık görülen kalıpları, $n = 16$ için tur sayısını ve sınıfı verir.

Kalıp	Gövde kaç kez?	$n = 16$	Sınıf
`for (i = 0; i < n; i++)`	n	16	$O(n)$
`for (i = 0; i < n; i += 2)`	$n / 2$	8	$O(n)$
`for (i = 1; i < n; i *= 2)`	$\lceil \log_2 n \rceil$	4	$O(\log n)$
iç içe: `i < n`, `j < n`	n^2	256	$O(n^2)$
iç içe: `i < n`, `j < i`	$n(n-1)/2$	120	$O(n^2)$
ardışık iki `i < n` döngüsü	$2n$	32	$O(n)$

Tablo 2.8: Döngü kalıpları ve gövdenin çalışma sayısı.

Kalıpları tek tek açıklayalım.

Basit döngü. `i` 0'dan başlar, her turda 1 artar ve `n`'ye ulaşınca durur. `i` 0, 1, …, $n - 1$ değerlerini alır; bu n turdur.

Adımı 2 olan döngü. `i` 0, 2, 4, … değerlerini alır. n çiftse tur sayısı tam $n/2$'dir; tekse $\lceil n/2 \rceil$'dir. Adım 2 olsa da 100 olsa da tur sayısı n'nin sabit bir kesridir ve döngü $O(n)$'dir. Adımı büyütmek döngüyü hızlandırır, ama sınıfını değiştirmez.

Katlanan döngü. `i` her turda iki katına çıkar: 1, 2, 4, 8, … Gövde, `i` n'ye ulaşmadığı sürece çalışır. k tur sonra $i = 2^k$ olur. Döngü $2^k \geq n$ olduğu ilk anda durur, yani $k = \lceil \log_2 n \rceil$'de. $n = 16$ için `i` 1, 2, 4, 8 değerlerini alır; 16'da durur: 4 tur. Bu kalıbın ayrıntılı izlemesi 2.3.2'de verilir. Aynı sonuç, `i`'nin n'den başlayıp her turda yarıya indiği döngü için de geçerlidir.

İç içe döngüler. Dış döngünün her turunda iç döngü baştan sona çalışır. İç döngü dıştan bağımsız olarak n tur dönüyorsa toplam $n \cdot n = n^2$ turdur. İç döngünün sınırı dış değişkene bağlıysa (`j < i`) çarpım yerine toplam yazılır: `i = 0`'da iç döngü 0 kez, `i = 1`'de 1 kez, …, `i = n - 1`'de $n - 1$ kez döner. Toplam:

$$0 + 1 + 2 + \ldots + (n - 1) = n(n - 1)/2$$

Bu toplam formülü sık kullanılır ve kolay hatırlanan bir gerekçesi vardır. Toplamı bir kez baştan, bir kez sondan yazıp alt alta toplayalım: $0 + (n - 1)$, $1 + (n - 2)$, … Her çift $n - 1$ eder ve n çift vardır. İki toplamın toplamı $n(n - 1)$ olduğu için tek toplam bunun yarısıdır. $n = 16$ için $16 \cdot 15 / 2 = 120$. İç döngü ortalamada dış döngünün yarısı kadar dönse de sonuç yine $O(n^2)$'dir; 1/2 katsayısı atılır.

Ardışık döngüler. Biri bittikten sonra öteki başlayan iki döngünün maliyetleri toplanır: $n + n = 2n$, yani $O(n)$. Ardışık döngüler iç içe döngülerden çok daha ucuzdur.

Bu gözlemlerden iki genel kural çıkar:

- **Ardışık bloklarda maliyetler toplanır ve en büyüğü kalır:** $O(n) + O(n^2) = O(n^2)$. Bir metotta önce doğrusal bir tarama, sonra iç içe bir döngü varsa metodun maliyetini iç içe döngü belirler.
- **İç içe döngülerde maliyetler çarpılır:** dış döngünün tur sayısı × iç döngünün tur sayısı. İç döngünün tur sayısı dış değişkene bağlıysa çarpım yerine bütün turların toplamı alınır.

Örnek 2.6

Aşağıdaki kod parçası bittiğinde `s` kaçtır ve karmaşıklığı nedir? ($n = 16$)

```
for (i = 0; i < n; i++) s++;
for (i = 0; i < n; i++) for (j = 1; j < n; j *= 2) s++;
```

1. **Birinci blok.** Basit döngü: 16 tur, `s` 16 olur. $O(n)$.
2. **İkinci blok, iç döngü.** `j` 1, 2, 4, 8 değerlerini alır: $\lfloor \log_2 16 \rfloor = 4$ tur. İç döngünün sınırı dış değişkene bağlı değildir.
3. **İkinci blok, toplam.** Dış döngü 16 tur; her turda iç döngü 4 tur döner. Çarpım: $16 \cdot 4 = 64$. $O(n \log n)$.
4. **Bütün parça.** `s = 16 + 64 = 80` . Ardışık bloklar toplanır: $O(n) + O(n \log n) = O(n \log n)$.

Sık yapılan hata

"İç içe iki döngü varsa karmaşıklık n^2'dir" diye ezberlemek. Bu kural yalnız iki döngü de n ile orantılı tur döndüğünde geçerlidir. İç döngü `j *= 2` ile ilerliyorsa toplam $n \log n$'dir; iç döngü sabit sayıda (örneğin 10 kez) dönüyorsa toplam $O(n)$'dir. Her döngünün tur sayısı ayrı ayrı bulunur ve sonra çarpılır.

2.3.2 Kod izleme: adimSay(16)

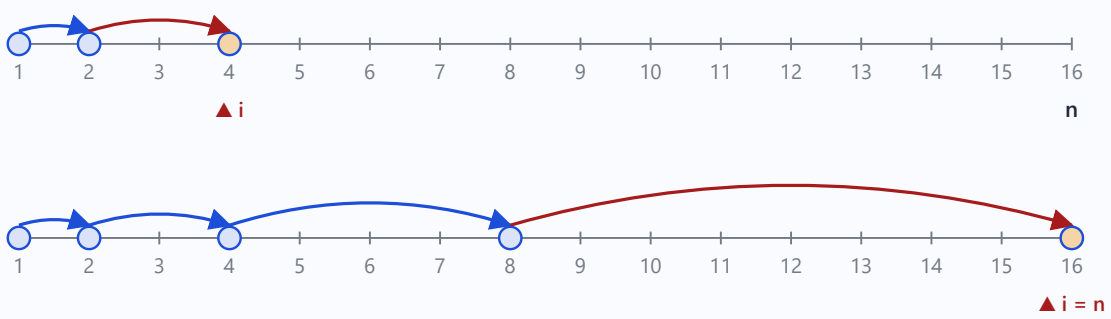
Katlanan döngüyü bir metot içinde izleyelim. `adimSay` , 1'den başlayıp ikiye katlayarak n'ye kaç adımda ulaşıldığını sayar:

```
int adimSay(int n) {
  int k = 0;
  for (int i = 1; i < n; i *= 2)
    k++;
  return k;
}
```

Metot bir sayaç `k`'yi 0'la başlatır. Döngü değişkeni `i` 1'den başlar ve her turda `i *= 2` ile iki katına çıkar. Her tur sayacı bir artırır. Döngü bitince sayaç, döngünün kaç tur döndüğünü verir.

Tur	Turun başında `i`	`i < 16` ?	`k` (turdan sonra)
1	1	doğru	1
2	2	doğru	2
3	4	doğru	3
4	8	doğru	4
—	16	yanlış, döngü biter	4 döner

Tablo 2.9: `adimSay(16)` izlemesi.



Şekil 2.9: `adimSay(16)` sayı doğrusu üzerinde. Üstte üçüncü tur: `i` 1'den 2'ye, 2'den 4'e atlamıştır. Altta son durum: `i` 8'den 16'ya atlar ve `i = n` olduğu için döngü biter. Her atlama bir öncekinin iki katı uzunluktadır.

Şekil 2.9 katlanan döngünün neden bu kadar az tur döndüğünü görsel olarak anlatır. Her atlama bir öncekinin iki katı uzunluktadır: 1, 2, 4, 8. Dört atlamada 1'den 16'ya varılır. Döngü n sayısının bütün değerlerine tek tek uğramaz; aradaki sayıların çoğunu atlar.

Metodun döndürdüğü 4, $log_2\ 16$'ya eşittir. Genel olarak `adimSay(n)`, $\lfloor log_2\ n \rfloor$ döndürür. Bu fonksiyonun ne kadar yavaş büyüdüğünü birkaç değerle görelim: `adimSay(17) = 5`, `adimSay(1000) = 10`, `adimSay(16000) = 14`. n'yi 1000 kat büyütmek tur sayısını yalnız 10 civarında artırır, çünkü $2^{10} = 1024 \approx 1000$'dir. Metot $O(\log n)$'dir.

Sık yapılan hata

Katlanan döngüde `i`'yi 0'dan başlatmak. `for (int i = 0; i < n; i *= 2)` döngüsünde `i` hep 0 kalır, çünkü $0 \cdot 2 = 0$'dır. Koşul her zaman doğrudur ve döngü sonsuza kadar döner. Katlanan döngüler 1'den, yarıya inen döngüler ise 0'a ulaşmadan durdurulacak biçimde yazılır.

2.3.3 Üç metot, üç sınıf

Kalıpları üç kısa metot üzerinde uygulayalım. Üçü de bir `int` dizisi alır; aralarındaki fark yalnız döngü yapısıdır.

***O(1)*: ilk öğe.**

```
int ilk(int[] a) {
    return a[0];
}
```

Metotta döngü yoktur. Bir dizi erişimi ve bir dönüş yapılır. Dizi indisle erişimde bellekteki adres doğrudan hesaplanır: başlangıç adresine indis çarpı öğe boyutu eklenir. Dizinin uzunluğu bu hesabı etkilemez. İşlem sayısı sabittir; metot *O(1)*'dir.

***O(n)*: toplam.**

```
int toplam(int[] a) {
    int s = 0;
    for (int x : a)  // n kez
        s += x;
    return s;
}
```

`for (int x : a)` biçimi, genişletilmiş `for` döngüsüdür (*for-each*): `x` sırasıyla dizinin her öğesini alır. Gövde her öğe için bir kez, toplam n kez çalışır. Gövdede yalnız bir toplama ve bir atama vardır. Döngü dışındaki atama ve dönüş sabit işlemlerdir. Toplam işlem sayısı n ile doğru orantılıdır; metot *O(n)*'dir.

***O(n^2)*: tekrar denetimi.**

```
boolean tekrarVar(int[] a) {
    for (int i = 0; i < a.length; i++)
        for (int j = i + 1; j < a.length; j++)
            if (a[i] == a[j])
                return true;
    return false;
}
// dış döngü n, iç döngü ≈ n/2
// → n(n−1)/2 karşılaştırma
```

Metot dizide birbirine eşit iki öğe olup olmadığını denetler. Dış döngü her `i` indisi için iç döngüyü çalıştırır. İç döngü `j`'yi `i + 1`'den başlatır ve `a[i]`'yi sağındaki her öğeyle karşılaştırır. `j`'nin `i + 1`'den başlaması iki şeyi sağlar: bir öğe kendisiyle karşılaştırılmaz ve her çift yalnız bir kez karşılaştı-

rılır (`a[0]` ile `a[3]` karşılaştırıldıysa `a[3]` ile `a[0]` ayrıca karşılaştırılmaz). Eşit bir çift bulunduğu anda metot `true` döndürür. Döngüler hiçbir eşitlik bulmadan biterse sonuç `false`'tur.

Karşılaştırma sayısını en kötü durum için, yani dizide hiç tekrar olmadığında sayalım. `i = 0` iken iç döngü *n − 1* kez, `i = 1` iken *n − 2* kez, …, `i = n − 2` iken 1 kez, `i = n − 1` iken hiç dönmez:

> $$(n-1) + (n-2) + \ldots + 1 + 0 = n(n-1)/2 = n^2/2 - n/2$$

Sabitler ve küçük terim atılınca sonuç $O(n^2)$'dir. İç döngü ortalamada yaklaşık $n/2$ tur döner; ama dış döngünün n turuyla çarpılınca yine karesel bir toplam çıkar. Bu sayı, aynı zamanda n öğeden seçilebilecek farklı çiftlerin sayısıdır. Bütün çiftlere bakan her algoritma en az bu kadar karşılaştırma yapar.

En iyi durum ise ilk karşılaştırmada eşitlik bulunan girdidir: `a[0] == a[1]` ise metot tek karşılaştırmayla döner, $\Theta(1)$. 2.1.3'teki sıfırlarla dolu dizi tam olarak bu durumdu.

2.3.4 Kod izleme: toplam()

`toplam` metodunu, onu çağıran kodla birlikte izleyelim. İzleme, metot çağrısının nasıl yürüdüğünü de gösterir.

```
int toplam(int[] a) {
    int s = 0;
    for (int x : a) {
        s += x;
    }
    return s;
}

// main içinde:
System.out.println(toplam(a));
```

`main` içinde `a = {7, 2, 9}` olsun. `toplam(a)` çağrıldığında yeni bir **yığın çerçevesi** (*stack frame*) açılır. Çerçeve, metodun parametrelerini ve yerel değişkenlerini (`s`, `x`) tutar. Parametre `a`, `main`'deki diziyle *aynı diziyi* gösterir; dizi kopyalanmaz. Java'da dizi değişkeni bir **referanstır** (*reference*): dizinin kendisini değil, bellekteki yerini tutar. Metoda geçirilen de bu yerdir. Bu yüzden çağrının maliyeti dizinin boyutuna bağlı değildir.

Adım	`x`	`s`	Açıklama
çağrı	—	—	yeni çerçeve açılır; `a` aynı diziyi gösterir
`s = 0`	—	0	toplam değişkeni başlatılır
1. tur	7	7	*0 + 7*
2. tur	2	9	*7 + 2*
3. tur	9	18	*9 + 9*

Adım	`x`	`s`	Açıklama
döngü sonu	—	18	dizide öğe kalmadı
`return`	—	—	çerçeve kapanır; 18 çağırana döner, `println` 18 yazar

Tablo 2.10: `toplam({7, 2, 9})` izlemesi.

Döngü gövdesi $n = 3$ kez çalışmıştır. Gövdenin dışında kalan işler, yani çerçeveyi açma, `s` 'yi başlatma, dönüş ve yazdırma, dizinin boyutundan bağımsızdır. Dizi 3 değil 3 milyon öğeli olsaydı yalnız gövdenin tur sayısı değişirdi. Toplam süre $c_1 \cdot n + c_2$ biçimindedir: $O(n)$.

> **Sık yapılan hata**
>
> Diziyi metoda geçirmenin diziyi kopyaladığını, dolayısıyla $O(n)$ sürdüğünü sanmak. Java'da metoda dizinin referansı geçer; çağrı $O(1)$'dir. Bunun bir sonucu daha vardır: metot içinde `a[0] = 100` gibi bir atama yapılırsa `main` 'deki dizi de değişir, çünkü ikisi aynı dizidir.

2.3.5 Doğrusal arama

Doğrusal arama (*linear search*), bir değeri dizide bulmanın en basit yoludur: dizinin başından başlanır ve öğelere sırayla bakılır.

```
int dogrusalArama(int[] a, int hedef) {
    for (int i = 0; i < a.length; i++)
        if (a[i] == hedef) return i;
    return -1;
}
```

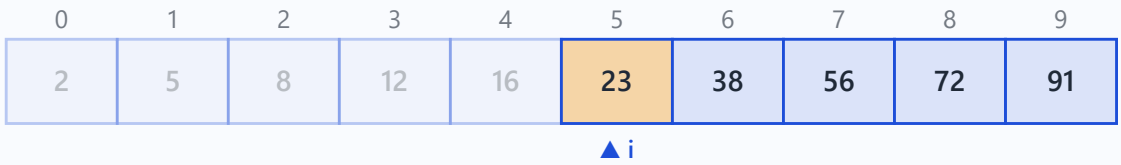
Döngü `i` 'yi 0'dan dizinin sonuna kadar ilerletir. Her turda `a[i]` hedefle karşılaştırılır. Eşitse metot o anda `i` 'yi döndürür ve döngünün geri kalanı çalışmaz. Döngü bir eşitlik bulmadan biterse hedef dizide yoktur ve metot `-1` döndürür. `-1` geçerli bir indis olmadığı için "bulunamadı" anlamında kullanılır.

Metodu sıralı bir dizi üzerinde izleyelim: `a = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}` ($n = 10$), hedef 23.

`i`	`a[i]`	`a[i] == 23 ?`	Sonuç
0	2	hayır	`i` ilerler
1	5	hayır	`i` ilerler
2	8	hayır	`i` ilerler
3	12	hayır	`i` ilerler
4	16	hayır	`i` ilerler

`i`	`a[i]`	`a[i] == 23` ?	**Sonuç**
5	23	**evet**	`return 5`

Tablo 2.11: `dogrusalArama(a, 23)` izlemesi.



Şekil 2.10: Doğrusal aramanın son adımı. Soluk hücreler (0–4) tek tek elenmiştir; 5. indisteki 23 bulunur. Her adımda yalnız bir öğe elenir.

Hedef 6 karşılaştırmada bulunmuştur. Dizinin sıralı olduğuna dikkat edelim: doğrusal arama bu bilgiyi hiç kullanmaz. 2'nin 23'ten küçük olduğunu görmek, ötekiler hakkında bir şey öğretmez; her adımda yalnız bir öğe elenir (Şekil 2.10). Hedef son öğe olsaydı ya da dizide hiç olmasaydı (örneğin 40 aransaydı) on öğenin hepsine bakılırdı: 10 karşılaştırma.

Analiz Örnek 2.3'teki sonuçların aynısını verir. En iyi durum, hedefin ilk öğe olmasıdır: 1 karşılaştırma, $\Theta(1)$. En kötü durum, hedefin sonda olması ya da hiç olmamasıdır: n karşılaştırma, $\Theta(n)$. Ortalama durum da konumlar eşit olasılıklıysa $(n + 1)/2$ karşılaştırmadır, yine $\Theta(n)$. Metot yalnız birkaç değişken kullanır; ek alanı $O(1)$'dir.

Doğrusal aramanın iyi bir yanı da vardır: dizinin sıralı olmasını gerektirmez. Sırasız bir dizide bir değeri bulmanın daha hızlı bir yolu yoktur. Hiçbir öğeye bakmadan onun hedef olmadığını bilemeyiz; en kötü durumda her öğeye bakmak zorunludur.

2.3.6 İkili arama

Dizi sıralıysa çok daha iyisi yapılabilir. Telefon rehberinde bir isim ararken ilk sayfadan başlayıp tek tek bakmayız. Rehberi ortadan açarız; aradığımız isim alfabede daha önceyse sağ yarıyı, daha sonraysa sol yarıyı bir kenara bırakırız. Kalan yarıda aynı işi tekrarlarız. **İkili arama** (*binary search*) bu yöntemin algoritma hâlidir.

```java
int ikiliArama(int[] a, int hedef) {
    int lo = 0, hi = a.length - 1;
    while (lo <= hi) {
        int mid = (lo + hi) / 2;
        if (a[mid] == hedef) return mid;
        if (a[mid] < hedef) lo = mid + 1;
        else hi = mid - 1;
    }
    return -1;
}
```

Kod satır satır şöyle çalışır:

- `lo` ve `hi`, hedefin bulunabileceği aralığın sol ve sağ uçlarıdır. Başlangıçta aralık bütün dizidir: `lo = 0`, `hi = a.length - 1`.
- `while (lo <= hi)`: aralıkta en az bir öğe kaldığı sürece arama sürer. `lo > hi` olunca aralık boşalmıştır.
- `mid = (lo + hi) / 2` aralığın ortasındaki indistir. Java'da iki `int`'in bölümü tam sayıdır; kesirli kısım atılır. Örneğin *(0 + 9)/2 = 4*.
- `a[mid] == hedef` ise hedef bulunmuştur ve `mid` döndürülür.
- `a[mid] < hedef` ise hedef ortadakinden büyüktür. Dizi sıralı olduğu için hedef ancak `mid`'in sağında olabilir: `lo = mid + 1`. Sol yarı, ortadaki öğeyle birlikte elenir.
- Aksi hâlde `a[mid] > hedef`'tir ve hedef ancak solda olabilir: `hi = mid - 1`.
- Döngü hedefi bulmadan biterse `-1` döndürülür.

Algoritmanın gücü, tek bir karşılaştırmanın aralığın yarısını elemesinden gelir. Doğrusal aramada `a[i] ≠ hedef` bilgisi yalnız bir öğeyi eler. İkili aramada ise `a[mid] < hedef` bilgisi, dizinin sıralı olması sayesinde `mid`'in solundaki her öğenin de hedeften küçük olduğunu söyler.

Aynı diziyi ve aynı hedefi kullanalım: `a = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}`, hedef 23.

Tur	`lo`	`hi`	`mid`	`a[mid]`	Karar
1	0	9	4	16	16 < 23 → `lo = 5`
2	5	9	7	56	56 > 23 → `hi = 6`
3	5	6	5	23	eşit → `return 5`

Tablo 2.12: `ikiliArama(a, 23)` izlemesi.





Şekil 2.11: `ikiliArama(a, 23)` izlemesinin üç turu (yukarıdan aşağı). 1. tur: aralık 0–9, `mid = 4`. 2. tur: sol yarı elenmiş, aralık 5–9, `mid = 7`. 3. tur: sağ taraf da elenmiş, aralık 5–6, `mid = 5` ve 23 bulunur. Soluk hücreler elenmiş kısımdır.

İzlemenin ayrıntıları şunlardır. 1. turda aralık bütün dizidir ve `mid = (0 + 9)/2 = 4` olur. `a[4] = 16`, 23'ten küçüktür; hedef sağ yarıdadır ve `lo = 5` yapılır. Beş öğe tek karşılaştırmayla elenmiştir. 2. turda `mid = (5 + 9)/2 = 7`, `a[7] = 56`'dır. 56 hedeften büyük olduğu için `hi = 6` yapılır. 3. turda `mid = (5 + 6)/2 = 5` ve `a[5] = 23`: hedef bulunmuştur.

İkili arama 3 turda bitmiştir; doğrusal arama aynı hedef için 6 karşılaştırma yapmıştı. Her turda aralık yarıya iner: 10 öğe, 5 öğe, 2 öğe. Fark bu küçük dizide iki kattır; büyük dizilerde çarpıcı hâle gelir.

> **Sık yapılan hata**
>
> İkili aramayı sırasız bir dizide kullanmak. Algoritma `a[mid] < hedef` bilgisinden "soldaki her öğe de küçüktür" sonucunu çıkarır. Bu çıkarım yalnız sıralı dizide geçerlidir. Sırasız bir dizide metot hata vermez, ama dizide bulunan bir değer için `-1` döndürebilir. Örneğin `{5, 1, 3}` dizisinde 1 aranırsa `mid = 1`'de hemen bulunur; ama 5 aranırsa `a[1] = 1 < 5` olduğu için sol yarı elenir ve 5 hiç bulunamaz.

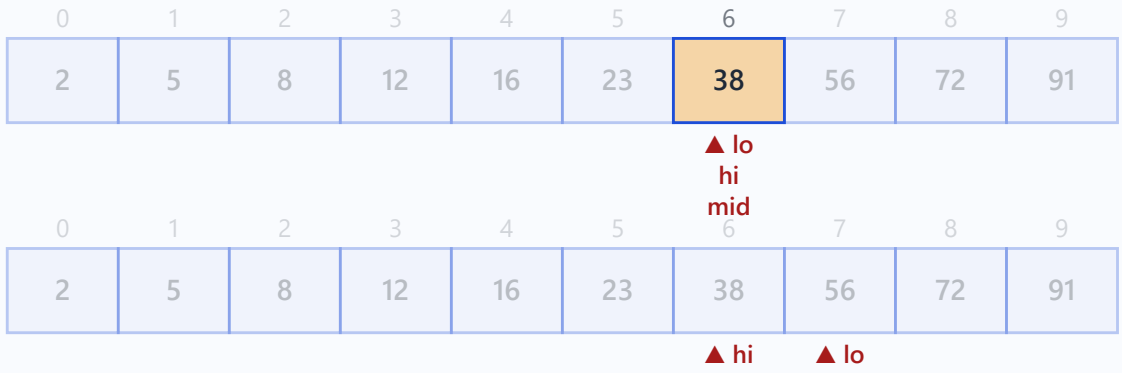
2.3.7 İkili aramada en kötü durum

İkili aramanın en kötü durumu, hedefin dizide olmadığı ve aralığın tek öğeye kadar daraldığı durumdur. Aynı dizide 40'ı arayalım. 40, 38 ile 56 arasında kalan ve dizide bulunmayan bir değerdir.

Tur	`lo`	`hi`	`mid`	`a[mid]`	Karar
1	0	9	4	16	16 < 40 → `lo = 5`
2	5	9	7	56	56 > 40 → `hi = 6`
3	5	6	5	23	23 < 40 → `lo = 6`
4	6	6	6	38	38 < 40 → `lo = 7`

Tur	`lo`	`hi`	`mid`	`a[mid]`	Karar
—	7	6	—	—	`7 ≤ 6` yanlış → `return -1`

Tablo 2.13: `ikiliArama(a, 40)` izlemesi; hedef dizide yoktur.



Şekil 2.12: `ikiliArama(a, 40)` izlemesinin sonu. Üstte 4. tur: aralık tek öğeye (6. indis, 38) inmiştir. Altta döngü sonrası: `lo = 7`, `hi = 6`; `lo`, `hi`'nin sağına geçmiştir ve aralık boştur.

İlk iki tur, 23 aramasıyla aynıdır. 3. turda `a[5] = 23` hedeften küçük çıkar ve `lo = 6` olur. Aralık artık tek öğedir: 6. indisteki 38. 4. turda `mid = 6`, 38 de hedeften küçüktür ve `lo = 7` yapılır. Bu noktada `lo = 7 > hi = 6`'dır (Şekil 2.12). Döngü koşulu yanlış çıkar ve metot `-1` döndürür.

Son durum ilginç bir bilgi de taşır. Döngü bittiğinde `lo`, hedefin dizide olsaydı yerleşeceği indistir: 40, 38'in (6. indis) ve 56'nın (7. indis) arasına, yani 7. indise girerdi. Sıralı diziye ekleme yapan bazı algoritmalar ikili aramanın bu özelliğini kullanır.

En kötü durumda bile yalnız 4 tur yapılmıştır; doğrusal arama aynı hedef için 10 karşılaştırma yapardı. Bu sayının nereden geldiğini aşağıdaki teorem açıklar.

Teorem 2.2 (İkili aramanın tur sayısı)

$n \geq 1$ öğeli sıralı bir dizide `ikiliArama` en çok *$\lfloor \log_2 n \rfloor + 1$* tur döner. Bu yüzden ikili arama *$O(\log n)$* zamanda çalışır.

Gerekçe şöyledir. Aralıktaki öğe sayısına m diyelim. Bir tur sonunda hedef bulunmamışsa yeni aralık, `mid`'in solundaki ya da sağındaki kısımdır. Ortadaki öğe de elendiği için yeni aralıkta en çok $\lfloor m/2 \rfloor$ öğe kalır. Başlangıçta $m = n$'dir. Her tur öğe sayısını en az yarıya indirdiği için k turdan sonra en çok $\lfloor n/2^k \rfloor$ öğe kalır. Bu sayı $2^k > n$ olduğunda 0'a iner, yani $k = \lfloor \log_2 n \rfloor + 1$ turdan sonra aralık kesinlikle boştur.

Örnekte $n = 10$ için $\lfloor \log_2 10 \rfloor + 1 = 3 + 1 = 4$'tür. Öğe sayıları 10 → 5 → 2 → 1 → 0 diye azalmıştır. Tablo 2.14 farklı dizi boyutları için en kötü durum tur sayısını, doğrusal aramanın en kötü durumuyla karşılaştırır.

n	Doğrusal arama	İkili arama ($\lfloor \log_2 n \rfloor + 1$)
10	10	4
1.000	1.000	10
1.024	1.024	11
1.000.000	1.000.000	20

Tablo 2.14: En kötü durumda doğrusal arama karşılaştırma sayısı ve ikili arama tur sayısı.

Bir milyon öğeli sıralı dizide ikili arama en çok 20 tur döner. Doğrusal arama aynı dizide bir milyon karşılaştırma yapabilir. Fark elli bin kattır ve n büyüdükçe daha da açılır. Bedeli, dizinin sıralı tutulmasıdır. Dizi bir kez sıralanıp çok kez aranacaksa bu bedel kolayca karşılanır.

Sık yapılan hata

Orta indisi `(lo + hi) / 2` ile hesaplamak, çok büyük dizilerde taşmaya yol açar. Java'da `int` en çok $2^{31} - 1 \approx 2{,}1 \cdot 10^9$ değerini tutar. `lo = 1.500.000.000` ve `hi = 2.000.000.000` ise `lo + hi = 3,5 · 10^9` bu sınırı aşar. Java hata vermez; sonuç sessizce negatife sarar ve `(lo + hi) / 2` değeri `-397483648` olur. Bu negatif indis `a[mid]`'de `ArrayIndexOutOfBoundsException`'a yol açar. Güvenli yazım `lo + (hi - lo) / 2`'dir: `hi - lo` farkı hiçbir zaman sınırı aşmaz ve sonuç doğru olarak `1750000000` çıkar. Bu hata, Java kütüphanesindeki ikili aramada da yıllarca fark edilmeden kalmıştır (Bloch, 2006).

2.3.8 Kod izleme: tekrarVar()

2.3.3'teki `tekrarVar` metodunu küçük bir dizide izleyelim: `a = {4, 7, 1, 7}`. Dizide 7 iki kez geçer.

```
boolean tekrarVar(int[] a) {
    for (int i = 0; i < a.length; i++)
        for (int j = i + 1; j < a.length; j++)
            if (a[i] == a[j])
                return true;
    return false;
}
```

Karşılaştırma	`i`	`j`	`a[i]` ile `a[j]`	Sonuç
1	0	1	4 ile 7	farklı

Karşılaştırma	`i`	`j`	`a[i]` ile `a[j]`	Sonuç
2	0	2	4 ile 1	farklı
3	0	3	4 ile 7	farklı
4	1	2	7 ile 1	farklı
5	1	3	7 ile 7	**eşit** → `return true`

Tablo 2.15: `tekrarVar({4, 7, 1, 7})` izlemesi.



Şekil 2.13: `tekrarVar` izlemesinin ilk ve son karşılaştırması. Solda `i = 0`, `j = 1`: 4 ile 7 karşılaştırılır. Sağda `i = 1`, `j = 3`: iki 7 eşittir ve metot `true` döndürür.

Dış döngü `i = 0` iken iç döngü `a[0] = 4`'ü sağındaki üç öğeyle karşılaştırır; hiçbiri 4 değildir. `i = 1`'de iç döngü `j = 2`'den başlar. `a[1] = 7` önce 1 ile, sonra `a[3] = 7` ile karşılaştırılır. Eşitlik bulunur ve metot beşinci karşılaştırmada `true` döndürür. `i = 2` turu hiç çalışmaz.

Dizide tekrar olmasaydı, örneğin `{4, 7, 1, 9}` verilseydi, bütün çiftler karşılaştırılırdı: $n(n-1)/2 = 4 \cdot 3 / 2 = 6$ karşılaştırma. Bu en kötü durumdur ve $O(n^2)$'dir. Tablo 2.2'deki ölçümde sürenin her katlamada yaklaşık 4 katına çıkması bu karesel büyümenin sonucudur.

Aynı problem daha hızlı çözülebilir mi? Evet. Dizi önce sıralanırsa eşit öğeler yan yana gelir ve yalnız komşu öğeleri karşılaştırmak yeterli olur. Sıralama $O(n \log n)$, komşu taraması $O(n)$ sürer; toplam $O(n \log n)$'dir. Bir sonraki örnek, gereksiz işi atarak karesel bir algoritmayı doğrusala indirmenin başka bir yolunu gösterir.

2.3.9 Önek ortalamaları problemi

Aynı problemi çözen iki algoritmanın maliyetleri ne kadar farklı olabilir? Bu soruyu somut bir problem üzerinde yanıtlayalım (Goodrich vd., Kesim 4.3).

Tanım 2.5

Önek ortalamaları (*prefix averages*) problemi: n sayılık bir `x[0..n-1]` dizisi verildiğinde, her j için ilk $j+1$ öğenin ortalamasını içeren `a` dizisini hesaplamak:

$$a[j] = (x[0] + x[1] + \ldots + x[j]) / (j + 1), \quad j = 0, 1, \ldots, n - 1$$

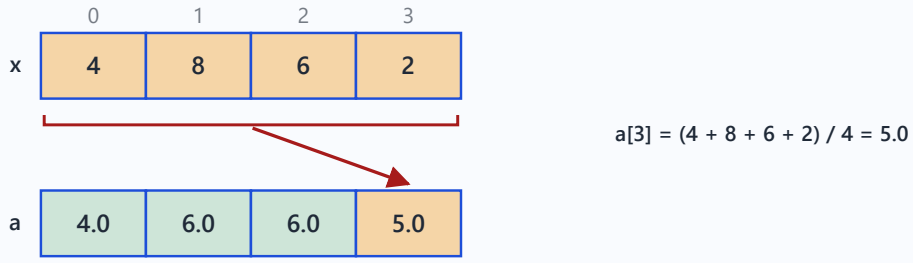
Bir dizinin "öneki", baştan başlayan bir parçasıdır: `x[0..j]`. Problem her önekin ortalamasını ister. Bu hesap gerçek hayatta sık kullanılır. Bir hisse senedinin yılbaşından bu yana günlük ortalama fiyatı ya da bir öğrencinin dönem başından bu yana not ortalaması birer önek ortalamasıdır. Hareketli ortalama ve finansal eğilim analizi de benzer hesaplara dayanır.

Örnek 2.7

`x = [4, 8, 6, 2]` için önek ortalamalarını hesaplayalım.

1. *a[0] = 4 / 1 = 4,0*
2. *a[1] = (4 + 8) / 2 = 12 / 2 = 6,0*
3. *a[2] = (4 + 8 + 6) / 3 = 18 / 3 = 6,0*
4. *a[3] = (4 + 8 + 6 + 2) / 4 = 20 / 4 = 5,0*

Sonuç: `a = [4.0, 6.0, 6.0, 5.0]`.



Şekil 2.14: Önek ortalamalarının son adımı: `a[3]`, `x[0..3]` öğelerinin ortalamasıdır, *(4 + 8 + 6 + 2)/4 = 5,0*. Yeşil hücreler önceki adımlarda hesaplanmış ortalamalardır.

Hesaba dikkatle bakınca bir tekrar göze çarpar: `a[2]` 'nin payı olan 18, `a[1]` 'in payı 12'ye `x[2] = 6` eklenerek bulunabilirdi. Problemin iki çözümü bu gözlemden ayrılır. Birinci çözüm her *j* için toplamı baştan hesaplar ve $O(n^2)$ sürer. İkinci çözüm toplamı turdan tura taşır ve $O(n)$ sürer.

2.3.10 onekOrt1: her j için baştan toplama

```
double[] onekOrt1(double[] x) {
  int n = x.length;
  double[] a = new double[n];
  for (int j = 0; j < n; j++) {
    double top = 0;
    for (int i = 0; i <= j; i++)
      top += x[i];
    a[j] = top / (j + 1);
  }
  return a;
}
```

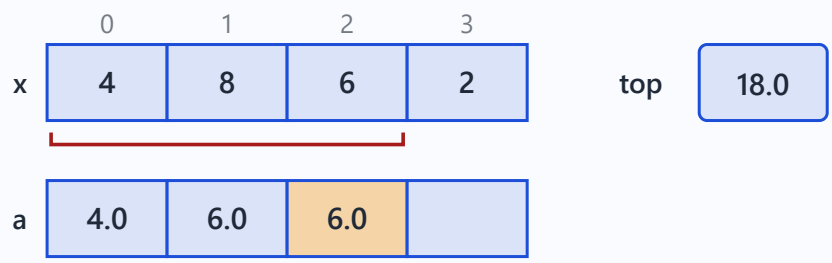
Kod satır satır şöyle çalışır:

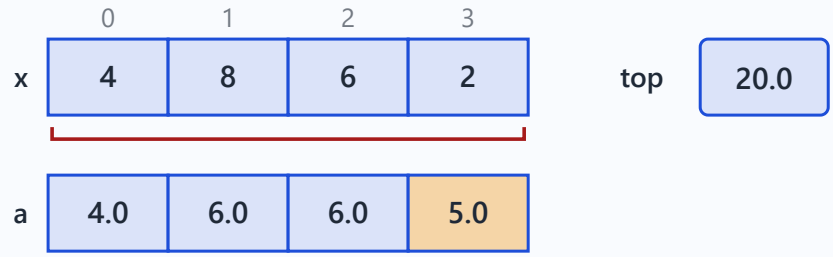
- `n` girdinin uzunluğudur. `a = new double[n]` sonuç dizisini ayırır; Java bu diziyi 0.0 değerleriyle doldurur.
- Dış döngü her `j` için bir önek ortalaması hesaplar.
- `double top = 0` satırı dış döngünün içindedir. Toplam her `j` için sıfırdan başlar.
- İç döngü `i`'yi 0'dan `j`'ye kadar (dahil) ilerletir ve `x[i]`'leri `top`'a ekler. Koşul `i <= j`'dir; `<` yazılsaydı `x[j]` toplama girmezdi.
- `a[j] = top / (j + 1)` ortalamayı yazar. `top` bir `double` olduğu için bölme kesirli yapılır.

Tablo 2.16, `x = [4, 8, 6, 2]` için iç döngünün her turunu gösterir.

`j`	Toplanan öğeler	`top` (ara değerler)	Toplama sayısı	`a[j]`
0	x[0]	0 → 4.0	1	4.0 / 1 = 4.0
1	x[0], x[1]	0 → 4.0 → 12.0	2	12.0 / 2 = 6.0
2	x[0] … x[2]	0 → 4.0 → 12.0 → 18.0	3	18.0 / 3 = 6.0
3	x[0] … x[3]	0 → 4.0 → 12.0 → 18.0 → 20.0	4	20.0 / 4 = 5.0

Tablo 2.16: `onekOrt1([4, 8, 6, 2])` izlemesi.





Şekil 2.15: `onekOrt1` izlemesinden iki an. Üstte `j = 2` bitmiştir: `top = 18.0`, `a[2] = 6.0`. Altta `j = 3` bitmiştir: `top` sıfırdan başlayıp dört öğeyi yeniden toplamış ve 20.0 olmuştur; `a[3] = 5.0`.

Tablonun üçüncü sütunu algoritmanın zayıf noktasını açıkça gösterir. `x[0] = 4` dört kez, `x[1] = 8` üç kez, `x[2] = 6` iki kez toplanmıştır. Her `j` turu, bir önceki turun bulduğu toplamı unutur ve aynı toplamaları yeniden yapar.

Toplama sayısını genel olarak sayalım. `j` turunda iç döngü $j + 1$ kez döner. Bütün turların toplamı:

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

$n = 4$ için $4 \cdot 5 / 2 = 10$ toplama yapılır; tablodaki 1 + 2 + 3 + 4 ile aynıdır. Bu ifade $n^2/2 + n/2$'dir ve Teorem 2.1'e göre $\Theta(n^2)$'dir. Algoritmanın ek alanı, sonuç dizisi `a` için $O(n)$'dir.

2.3.11 onekOrt2: birikimli toplam

Gereksiz işi atmak için toplamı turdan tura taşımak yeterlidir. `j`. önekin toplamı, `j - 1`. önekin toplamına `x[j]` eklenerek bulunur:

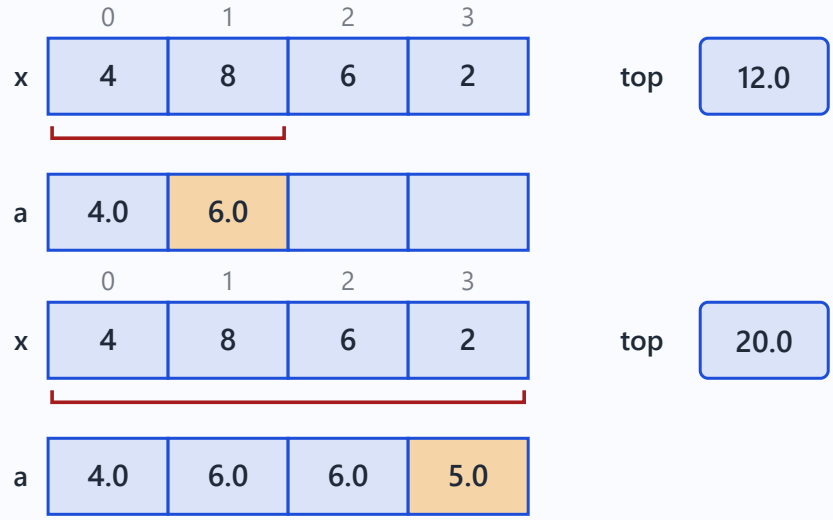
```
double[] onekOrt2(double[] x) {
  int n = x.length;
  double[] a = new double[n];
  double top = 0;
  for (int j = 0; j < n; j++) {
    top += x[j];
    a[j] = top / (j + 1);
  }
  return a;
}
```

İki kod arasındaki fark küçüktür ama belirleyicidir. `double top = 0` satırı döngünün dışına alınmıştır; toplam bir kez sıfırlanır ve turlar boyunca korunur. İç döngü kaldırılmış, yerine tek bir `top += x[j]` satırı gelmiştir. Her turun başında `top`, `x[0..j-1]`'in toplamını tutar; `x[j]` eklenince `x[0..j]`'nin toplamı olur.

`j`	`top += x[j]`	`top`	`a[j]`
başlangıç	—	0.0	—

`j`	`top += x[j]`	`top`	`a[j]`
0	+ 4	4.0	4.0 / 1 = 4.0
1	+ 8	12.0	12.0 / 2 = 6.0
2	+ 6	18.0	18.0 / 3 = 6.0
3	+ 2	20.0	20.0 / 4 = 5.0

Tablo 2.17: `onekOrt2([4, 8, 6, 2])` izlemesi.



Şekil 2.16: `onekOrt2` izlemesinden iki an. Üstte `j = 1` : önceki toplam 4.0'a yalnız `x[1] = 8` eklenmiş, `top = 12.0` , `a[1] = 6.0` . Altta son tur: `top = 20.0` , `a[3] = 5.0` .

Sonuç `[4.0, 6.0, 6.0, 5.0]` 'dır; `onekOrt1` ile aynıdır. Toplama sayısı ise 10 yerine 4'tür. Genel olarak döngü n tur döner ve her turda sabit sayıda işlem yapar: bir toplama, bir bölme, bir atama. `onekOrt2` , $\Theta(n)$ zamanda çalışır. Ek alanı yine sonuç dizisi için $O(n)$'dir.

Sık yapılan hata

`onekOrt2` 'yi yazarken `double top = 0` satırını döngünün içine koymak. Bu durumda `top` her turda sıfırlanır ve `a[j] = x[j] / (j + 1)` hesaplanır: `x = [4, 8, 6, 2]` için sonuç `[4.0, 4.0, 2.0, 0.5]` olur. Birikimli toplam yönteminin bütün fikri, toplamın turlar arasında korunmasıdır.

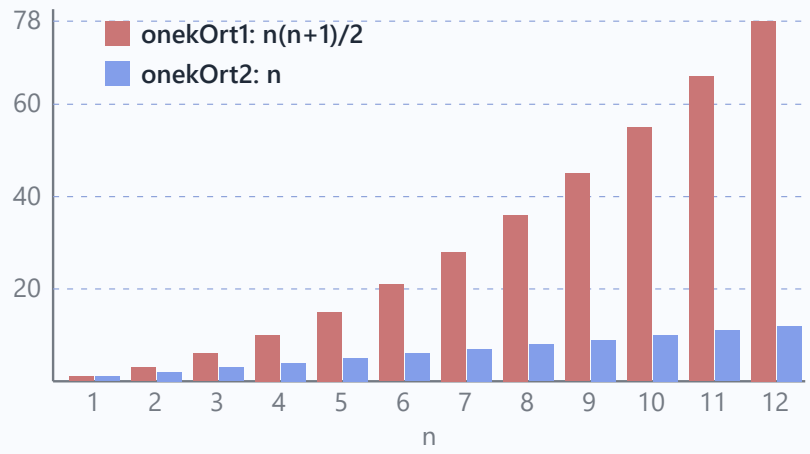
2.3.12 İki çözümün karşılaştırması

İki algoritmanın toplama sayılarını farklı n değerleri için yan yana koyalım.

n	`onekOrt1` : $n(n+1)/2$	`onekOrt2` : n
10	55	10

n	`onekOrt1` : *n(n + 1)/2*	`onekOrt2` : *n*
100	5.050	100
1.000	500.500	1.000
1.000.000	500.000.500.000	1.000.000

Tablo 2.18: Önek ortalamaları için toplama sayıları.



Şekil 2.17: *n = 1 … 12* için toplama sayıları. `onekOrt1` 'in çubukları *n(n + 1)/2* ile hızla yükselir; `onekOrt2` 'ninkiler *n* ile doğrusal artar.

n = 10'da fark 5,5 kattır ve önemsiz görünebilir. *n = 10^6*'da ise `onekOrt1` yaklaşık *$5 \cdot 10^{11}$* toplama yapar, `onekOrt2` yalnız *10^6*. Saniyede *10^9* toplama yapan bir makinede birincisi yaklaşık 500 saniye (8 dakikadan fazla), ikincisi 1 milisaniye sürer. Oran, *(n + 1)/2* olduğu için *n* ile birlikte büyür.

Farkın kaynağı tekrarlanan işin ortadan kaldırılmasıdır. `onekOrt2` önceki toplamı hatırlar; `onekOrt1` unutur ve yeniden hesaplar. Bu fikir yalnız bu probleme özgü değildir. Bir hesabın sonucunu saklayıp sonraki hesaplarda kullanmak, pek çok verimli algoritmanın temelidir. Bir algoritmayı hızlandırmanın ilk adımı çoğu zaman şu soruyu sormaktır: "Aynı işi birden fazla kez yapıyor muyum?"

İki algoritmanın alan maliyeti aynıdır: ikisi de sonuç için *n* hücrelik bir dizi ayırır ve birkaç değişken kullanır. Hızlı çözüm, ek bellek harcamadan kazanılmıştır.

2.4 Alan, doğruluk ve sağlamlık

Şimdiye kadar yalnız zamanı ölçtük. İyi bir algoritmanın dört ölçütünden üçü daha vardır (2.1.2): bellek kullanımı, doğruluk ve sağlamlık. Bellek, zamanla aynı araçlarla ölçülür. Doğruluk ve sağlamlık ise ölçülmez; gösterilir ve sınanır.

2.4.1 Alan karmaşıklığı

Alan karmaşıklığı, bir algoritmanın kullandığı bellek miktarının girdi boyutuna göre nasıl arttığını gösterir. Zaman karmaşıklığında olduğu gibi büyük O gösterimiyle ifade edilir; sabitler ve küçük terimler yine atılır.

Girdinin kendisi çoğu zaman sayılmaz. `maksimum` metodu n öğeli bir dizi alır, ama bu dizi metot çağrılmadan önce zaten bellektedir. Metodun kendi harcadığı bellek yalnız `enBuyuk` ve `i` değişkenleridir. Algoritmanın girdi dışında kullandığı bu belleğe **ek alan** denir ve analizde genellikle bu ölçülür. Bir analizde girdinin sayılıp sayılmadığı açıkça belirtilmelidir.



Şekil 2.18: Üç alan sınıfı (soldan sağa). *O(1)*: tek bir değişken. *O(n)*: n hücreli bir dizi. *O(n^2)*: $n \times n$ boyutunda bir matris.

Şekil 2.18 üç temel sınıfı gösterir:

- ***O(1)*, sabit alan.** Birkaç değişken oluşturmak. Bellek kullanımı girdi boyutuyla değişmez. `maksimum`, `toplam`, `dogrusalArama`, `ikiliArama` ve `tekrarVar` yalnız birkaç değişken (en büyük değer, toplam, indisler, `lo`, `hi`, `mid`) kullanır; ek alanları *O(1)*'dir.
- ***O(n)*, doğrusal alan.** Girdinin boyutunda yeni bir dizi oluşturmak. `onekOrt1` ve `onekOrt2`, sonucu tutan `a` dizisi için n hücre ayırır; ek alanları *O(n)*'dir.
- ***O(n^2)*, karesel alan.** $n \times n$ boyutunda bir matris saklamak. Örneğin n şehir arasındaki uzaklıkları tutan bir tablo n^2 hücre gerektirir. $n = 10.000$ için bu yüz milyon hücredir; her biri 4 baytlık `int` ise yaklaşık 400 MB eder.

Bellek verimliliği, belleği sınırlı sistemlerde önemlidir: gömülü cihazlar, telefonlar, aynı anda binlerce isteğe hizmet eden sunucular. Ama sınırsız bellekli bir sistem yoktur. Karesel bellek gerektiren bir algoritma, n yeterince büyüyünce en güçlü makinede de belleği tüketir.

Alan hesabında gözden kaçan bir kalem de **çağrı yığınıdır** (*call stack*). Her metot çağrısı bir çerçeve açar (2.3.4) ve çerçeve, metot dönene kadar bellekte kalır. Döngüyle yazılmış bir metot tek çerçeve kullanır. Kendini çağıran, yani özyinelemeli bir metotta ise aynı anda açık olan çerçeve sayısı çağrı derinliği kadardır. n derinliğe inen bir özyineleme, hiç dizi ayırmasa bile *O(n)* ek alan kullanır.

Örnek 2.8

Bir diziyi tersine çeviren iki metot düşünelim.

1. **Yeni dizi ile.** `b = new int[n]` ayrılır ve `b[i] = a[n - 1 - i]` yazılır. Zaman *O(n)*, ek alan *O(n)*.

2. **Yerinde.** İki indis kullanılır: `i` baştan, `j` sondan başlar. `a[i]` ile `a[j]` yer değiştirir, `i` artar, `j` azalır; `i < j` olduğu sürece devam edilir. Yer değiştirme için tek bir geçici değişken yeter. Zaman *O(n)*, ek alan *O(1)*.

İki metodun zaman maliyeti aynıdır; bellek maliyetleri farklıdır. Yerinde çalışan sürümün bir bedeli vardır: girdi dizisini değiştirir. Özgün dizi sonradan gerekecekse yeni dizi ayırmak kaçınılmazdır.

Sık yapılan hata

Ek alanı hesaplarken metodun döndürdüğü diziyi unutmak ya da girdiyi ek alana katmak. `onekOrt2` yalnız birkaç değişkenle çalışıyor gibi görünür, ama sonucu yazdığı `double[n]` dizisi yenidir; ek alanı *O(n)*'dir. Öte yandan `maksimum` 'un aldığı *n* öğeli dizi onun ek alanına girmez; ek alanı *O(1)*'dir.

2.4.2 Doğruluk

Hızlı çalışan ama yanlış sonuç veren bir algoritmanın değeri yoktur. Bu yüzden doğruluk, maliyetten önce gelir.

Tanım 2.6

Bir algoritma, problemin her geçerli girdisi için sonlu sürede beklenen çıktıyı üretiyorsa **doğrudur**. Tanımın iki parçası vardır: algoritma *durmalı* ve durduğunda verdiği sonuç *doğru* olmalıdır.

"Her geçerli girdi" ifadesi önemlidir. Sınırsız sayıda girdi vardır ve hepsini denemek mümkün değildir. Doğruluğu göstermek için dört araç kullanılır:

- **Test senaryoları.** Algoritma, sonucu önceden bilinen girdilerle çalıştırılır ve çıktılar beklenen sonuçlarla karşılaştırılır. Örneğin `maksimum({3, 8, 2, 9, 4})` 9 döndürmelidir. Testler hataları bulmakta çok etkilidir, ama doğruluğu kanıtlamaz: denenmeyen bir girdide hata olabilir.
- **Uç durumlar.** Hataların büyük kısmı sınırlarda saklanır. Boş girdi, tek öğeli girdi, en büyük ve en küçük değerler, bütün öğeleri eşit dizi, zaten sıralı ya da ters sıralı dizi mutlaka denenir. `maksimum` için tek öğeli dizi, en büyüğün başta, sonda ve ortada olduğu diziler ve negatif sayılardan oluşan dizi iyi uç durumlardır.
- **Döngü değişmezi.** Her turda doğru kalan bir özellik bulunur ve algoritmanın doğruluğu bu özellikten ispatlanır. Bu, testlerden farklı olarak bütün girdileri kapsar (2.4.3).
- **Karşı örnek.** Bir algoritmanın yanlış olduğunu göstermek için tek bir girdi yeter. Bu girdiye karşı örnek denir. Doğruluğu göstermek için bütün girdiler hakkında konuşmak gerekir; yanlışlığı göstermek için bir tanesi yeterlidir.

Örnek 2.9

`maksimum` 'un şu sürümü doğru mudur?

```
int enBuyuk = 0;
for (int i = 0; i < a.length; i++) if (a[i] > enBuyuk) enBuyuk = a[i];
return enBuyuk;
```

1. **Test.** `{3, 8, 2, 9, 4}` için 9 döner. `{1, 2, 3}` için 3 döner. Pozitif sayılarla yapılan bütün testler geçer.
2. **Şüphe.** Başlangıç değeri 0, dizinin bir öğesi değildir. Dizideki bütün öğeler 0'dan küçükse hiçbir öğe `enBuyuk` 'u geçemez.
3. **Karşı örnek.** `{-3, -1, -7}` için metot 0 döndürür. Doğru cevap −1'dir ve 0 dizide hiç yoktur.

Tek karşı örnek, metodun yanlış olduğunu kanıtlar. Düzeltme, başlangıç değerini dizinin bir öğesinden almaktır: `enBuyuk = a[0]` . 2.1.6'daki `maksimum` tam olarak böyle yazılmıştır.



Şekil 2.19: `maksimum` için döngü değişmezi. `i = 3` turunun başında işlenen kısım `a[0..2]` 'dir ve `enBuyuk = 8` bu kısmın en büyüğüdür. Döngü bitince `i = n` olur ve işlenen kısım bütün dizidir.

2.4.3 Döngü değişmeziyle ispat

Testler “bu girdilerde çalışıyor” der. İspat ise “her girdide çalışır” der. Döngü içeren algoritmalar için en kullanışlı ispat aracı döngü değişmezidir.

Tanım 2.7

Döngü değişmezi (*loop invariant*), bir döngünün her turunun başında doğru olan bir önermedir. Değişmezle ispat üç adımdan oluşur:

- **Başlangıç:** Önerme, ilk turdan önce doğrudur.
- **Koruma:** Önerme bir turun başında doğruysa, o tur bitince (yani sonraki turun başında) da doğrudur.
- **Sonlanma:** Döngü bittiğinde önerme, algoritmanın doğru olduğunu gösteren bir sonuç verir.

Yöntem, matematikteki tümevarıma benzer. Başlangıç adımı ilk basamağı, koruma adımı bir basamaktan ötekine geçişi sağlar; ikisi birlikte önermenin her turda doğru olduğunu gösterir. Sonlanma adımı da bu önermeyi istenen sonuca bağlar.

Teorem 2.3 (maksimum'un doğruluğu)

En az bir öğeli her `a` dizisi için `maksimum(a)`, dizinin en büyük öğesini döndürür.

İspat. Değişmez şudur: *her turun başında* `enBuyuk`, `a[0..i-1]` *öğelerinin en büyüğüdür.*

- **Başlangıç.** İlk turdan önce `i = 1` ve `enBuyuk = a[0]`'dır. `a[0..0]` tek öğelidir ve en büyüğü `a[0]`'dır. Değişmez doğrudur.
- **Koruma.** Turun başında `enBuyuk`, `a[0..i-1]`'in en büyüğü olsun. Tur `a[i]`'yi `enBuyuk` ile karşılaştırır. `a[i]` daha büyükse `enBuyuk = a[i]` yapılır; `a[i]` hem eski en büyükten hem de dolayısıyla öncekilerin hepsinden büyük olduğu için `a[0..i]`'nin en büyüğüdür. `a[i]` daha büyük değilse `enBuyuk` değişmez; eski en büyük `a[i]`'den küçük değildir ve `a[0..i]`'nin en büyüğü olarak kalır. Her iki durumda tur sonunda `enBuyuk`, `a[0..i]`'nin en büyüğüdür. `i++` sonrası bu, "`a[0..i-1]`'in en büyüğü" demektir. Değişmez korunur.
- **Sonlanma.** `i` her turda 1 artar ve döngü `i = n` olunca durur; yani döngü sonlu sürede biter. Durduğu anda değişmez, `enBuyuk`'un `a[0..n-1]`'in, yani bütün dizinin en büyüğü olduğunu söyler. Metot bu değeri döndürür. ■

Şekil 2.19 değişmezi `{3, 8, 2, 9, 4}` dizisinde `i = 3` anında gösterir. Yeşil çizgiyle işaretli `a[0..2] = {3, 8, 2}` işlenmiştir ve `enBuyuk = 8` bu kısmın en büyüğüdür. Sıradaki `a[3] = 9` daha büyük olduğu için tur sonunda `enBuyuk = 9` olur ve değişmez `a[0..3]` için de doğru kalır.

İspatın başında "en az bir öğeli" koşulu vardır. Boş dizide `a[0]` yoktur ve başlangıç adımı çalışmaz. Bu koşul ispatın bir eksiği değil, metodun ön koşuludur. Ön koşulun sağlanmadığı durumda ne olacağı ise sağlamlık sorusudur.

Sık yapılan hata

Değişmezi turun ortasında doğru olan bir şey sanmak. Değişmez, turun *başında* doğru olmalıdır; tur içinde kısa bir süre bozulabilir. Örneğin `onekOrt2` 'de "`top` , `x[0..j-1]` 'in toplamıdır" değişmezi, `top += x[j]` satırından sonra geçici olarak bozulur. `j++` ile yeniden kurulur. Koruma adımı tam da bunu gösterir: tur bitince değişmez yeniden doğrudur.

2.4.4 Sağlamlık

Doğruluk, geçerli girdiler hakkındadır. Gerçek programlar ise geçersiz girdilerle de karşılaşır: boş bir dosya, eksik bir satır, beklenmedik biçimde bir sayı. **Sağlamlık**, algoritmanın bu tür beklenmeyen durumlarla başa çıkabilmesidir. Sağlam bir algoritma, normal aralığın dışındaki girdilerde de çökmez ya da sessizce yanlış sonuç üretmez. Ya girdiyi uygun biçimde işler ya da sorunu açık bir hata mesajıyla bildirir.

Sağlamlık dört açıdan sınanır:

Durum	Ne sınanır?	Örnek
Geçersiz girdiler	Uygun olmayan ya da beklenmeyen girdilerle başa çıkma	`null` , negatif boyut
Hatalı veriler	Bozuk ya da eksik verilerle başa çıkma	eksik satır, yanlış biçim
Büyük veri kümeleri	Çok büyük girdileri işleyebilme	10^8 kayıt
Aşırı yüklenme	Ağır yük altındaki davranış	aynı anda binlerce istek

Tablo 2.19: Sağlamlığın sınandığı durumlar.

Bu durumların her biri gerçek bir hata kaynağına karşılık gelir. `maksimum` 'a boş dizi verilirse `a[0]` satırı `ArrayIndexOutOfBoundsException` fırlatır; `null` verilirse `NullPointerException` fırlatır. Bu istisnalar programı durdurur, ama hata mesajı sorunun nedenini açıkça söylemez. İkili aramadaki taşma hatası (2.3.7) ise büyük veri kümelerinde ortaya çıkan bir sağlamlık sorunudur: algoritma küçük dizilerde kusursuz çalışır, ancak bir milyardan büyük dizilerde bozulur.

Sağlam bir sürüm, ön koşulu açıkça denetler ve ihlali anlaşılır bir mesajla bildirir. Aşağıdaki metot, `maksimum` 'un sağlamlaştırılmış bir hâlidir:

```
int maksimumSaglam(int[] a) {
  if (a == null || a.length == 0)
    throw new IllegalArgumentException("Dizi boş olamaz");
  int enBuyuk = a[0];
  for (int i = 1; i < a.length; i++)
    if (a[i] > enBuyuk)
      enBuyuk = a[i];
  return enBuyuk;
}
```

İlk satır iki geçersiz durumu birlikte yakalar. `a == null` denetimi önce yapılır; `||` işleci soldaki koşul doğruysa sağdakini hiç değerlendirmez. Bu sayede `null` bir dizinin `length` alanına erişilmeye çalışılmaz. Geçersiz girdide `IllegalArgumentException` fırlatılır ve mesaj sorunun ne olduğunu söyler. Metodun geri kalanı `maksimum` ile aynıdır.

Denetim en çok iki karşılaştırmadır ve $O(1)$ sürer. Metodun karmaşıklığı $\Theta(n)$ olarak kalır. Sağlamlığın maliyeti çoğu zaman bu kadar küçüktür. Bu yüzden özellikle başkalarının çağıracağı metotlarda ön koşulların denetlenmesi iyi bir alışkanlıktır.

Boş dizide ne yapılacağı bir tasarım kararıdır. İstisna fırlatmak bir seçenektir. Başka bir seçenek, özel bir değer döndürmektir; Java kütüphanesindeki `OptionalInt` bu iş için tasarlanmıştır. Yanlış olan tek seçenek, sessizce anlamsız bir değer döndürmektir. Örnek 2.9'daki hatalı sürüm boş dizide 0 döndürür. Bu değer, çağıran kodun "dizinin en büyüğü 0'dır" sonucuna varmasına yol açar.

> **Sık yapılan hata**
>
> Sağlamlığı yalnız "programın çökmemesi" olarak anlamak. Hatayı yutup yanlış bir değerle devam eden program çökmez, ama sağlam da değildir. Sağlam bir program geçersiz durumu fark eder ve açıkça bildirir. Sessiz bir yanlış sonuç, gürültülü bir çökmeden daha tehlikelidir; çünkü fark edilmeden başka hesaplara karışır.

2.5 Sık yapılan hatalar

Not boyunca her konunun yanında verilen hataların en önemlileri Tablo 2.20'de bir arada toplanmıştır.

Yanılgı ya da hata	Doğrusu
"O = en kötü durum"	O bir üst sınır gösterimidir; en iyi ve en kötü durum ayrı kavramlardır. Her durum için O, Ω, Θ yazılabilir.
"$O(1)$ her zaman hızlıdır"	Sabit büyük olabilir: her çağrıda 1000 adım atan $O(1)$ işlem, $n < 1000$ iken n adımlık $O(n)$ işlemden yavaştır.
"İç içe döngü = n^2"	İç döngü `j *= 2` ile ilerliyorsa $n \log n$; sabit sayıda dönüyorsa $O(n)$.
"Logaritmanın tabanı önemlidir"	$\log_2 n$ ile $\log_{10} n$ arasında yalnız sabit bir çarpan vardır; ikisi de $O(\log n)$.
`(lo + hi) / 2` **taşması**	Çok büyük dizilerde `int` taşar; `lo + (hi - lo) / 2` güvenlidir.
`a == b` **ile dizi karşılaştırmak**	`==` referansları karşılaştırır; içerik için `Arrays.equals(a, b)` kullanılır.
Ölçümde ısınmayı ve gürültüyü yok saymak	İlk ölçüm JIT yüzünden yavaştır; ölçümler tekrarlanır ve ortanca alınır.
En kötü durumu yanlış girdiyle ölçmek	Sıfırlarla dolu dizi `tekrarVar`'ın en iyi durumudur; en kötü durum için tekrarsız veri gerekir.

Tablo 2.20: Algoritma analizinde sık görülen yanılgılar ve hatalar.

Dizi karşılaştırma hatası, `tekrarVar` gibi metotları sınarken sık yapılır. `int[] a = {1, 2}; int[] b = {1, 2};` yazıldığında `a == b` `false`'tur, çünkü `a` ve `b` bellekte iki ayrı diziyi gösterir. `==` "aynı dizi mi?" sorusunu sorar, "aynı içerikte mi?" sorusunu değil. `Arrays.equals(a, b)` ise öğeleri tek tek karşılaştırır ve `true` döndürür. Bu karşılaştırma *O(n)* sürer; `==` ise *O(1)*'dir, çünkü yalnız iki referansı karşılaştırır.

"*O(1)* her zaman hızlıdır" satırının ardındaki ders, öteki sınıflar için de geçerlidir. Örneğin *1000n* adımlık doğrusal bir algoritma ile n^2 adımlık karesel bir algoritmayı karşılaştıralım. $1000n < n^2$ eşitsizliği ancak $n > 1000$ için doğrudur. Daha küçük girdilerde karesel algoritma daha hızlıdır. Asimptotik analiz, girdi büyüdükçe hangisinin kazanacağını söyler; küçük girdilerde hangisinin hızlı olduğunu söylemek için sabitlere de bakmak gerekir.

2.6 Özet

- **Algoritma**, her geçerli girdi için doğru çıktıyı üreten sonlu, kesin ve etkili adımlardan oluşan yöntemdir. İyi bir algoritma dört ölçütle değerlendirilir: zaman karmaşıklığı, alan karmaşıklığı, doğruluk ve sağlamlık.
- **Deneysel analiz** süreyi `System.nanoTime()` ile ölçer. Ölçümler tekrarlanır, ortanca alınır ve ilk (ısınma) ölçümü ayrılır. **Katlama deneyinde** $T(2n)/T(n) \approx 2^k$ ise $T(n) \approx c \cdot n^k$'dir. Deneysel analiz donanıma bağlıdır, yalnız denenen girdileri kapsar ve tam bir kod gerektirir.
- **İlkel işlemleri saymak** donanımdan bağımsızdır ve bütün girdiler için geçerlidir. `maksimum` en iyi durumda $3n + 1$, en kötü durumda $4n$ işlem yapar; ikisi de $\Theta(n)$'dir.
- $f(n) = O(g(n))$: bir $c > 0$ ve n_0 için her $n \geq n_0$'da $f(n) \leq c \cdot g(n)$ (üst sınır). Ω alt sınır, Θ ikisi birden (sıkı sınır). Sabit çarpanlar ve küçük terimler atılır; d. dereceden polinom $\Theta(n^d)$'dir; logaritmanın tabanı önemsizdir.
- **En iyi, ortalama ve en kötü durum**, hangi girdinin maliyetine bakıldığını belirler ve sınır gösterimlerinden bağımsızdır. En kötü durum garanti verir; ortalama durum bir olasılık dağılımı varsayar; en iyi durum tek başına yanıltıcıdır. **Amortize analiz**, uzun bir işlem dizisinin toplam maliyetini işlem sayısına böler; büyüyen dizide sona ekleme amortize *O(1)*'dir.
- Sık görülen sınıflar, yavaş büyüyenden hızlı büyüyene: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$. Saniyede 10^9 işlemle $n = 100$ için 2^n adım yaklaşık $4 \cdot 10^{13}$ yıl sürer.
- **Döngü kalıpları**: tek döngü $O(n)$; katlanan ya da yarıya inen döngü $O(\log n)$; iç içe iki tam döngü $O(n^2)$; dış n, iç $\log n$ tur dönen döngü $O(n \log n)$. Ardışık bloklarda maliyetler toplanır, iç içe döngülerde çarpılır.
- **Doğrusal arama** $O(n)$'dir ve sıralı dizi gerektirmez. **İkili arama** sıralı dizide her turda aralığı yarıya indirir ve en çok $\lfloor \log_2 n \rfloor + 1$ tur döner: $O(\log n)$. Orta indis `lo + (hi - lo) / 2` ile hesaplanır.
- `tekrarVar` bütün çiftleri karşılaştırır: en kötü durumda $n(n-1)/2$ karşılaştırma, $O(n^2)$. **Önek ortalamalarında** her öneki baştan toplayan `onekOrt1` $n(n+1)/2$ toplama ile $O(n^2)$, toplamı taşıyan `onekOrt2` n toplama ile $O(n)$'dir.

- **Alan karmaşıklığı** genellikle girdi dışındaki ek belleği ölçer: birkaç değişken $O(1)$, yeni dizi $O(n)$, $n \times n$ matris $O(n^2)$. Özyinelemede çağrı yığını da sayılır.
- **Doğruluk** testlerle, uç durumlarla ve döngü değişmeziyle (başlangıç, koruma, sonlanma) gösterilir; tek bir karşı örnek yanlışlığı kanıtlar. **Sağlam** bir algoritma geçersiz girdiyi fark eder ve açıkça bildirir.

Konu	Akılda kalacaklar
Deneysel analiz	süre donanıma ve girdiye bağlıdır; katlama deneyi: $T(2n)/T(n) \approx 2^k \to T(n) \approx c \cdot n^k$
İlkel işlem sayma	atama, karşılaştırma, aritmetik, dizi erişimi; donanımdan bağımsız, bütün girdiler için
O, Ω, Θ	üst, alt ve sıkı sınır; sabit çarpanlar ve küçük terimler atılır
Durumlar	en iyi, ortalama, en kötü; garanti için en kötü durum, uzun işlem dizisi için amortize
Döngü kalıpları	tek döngü $O(n)$ · iç içe $O(n^2)$ · yarıya inen $O(\log n)$ · dış n, iç $\log n \to O(n \log n)$
Algoritma seçimi	önek ortalamaları $O(n^2) \to O(n)$; sıralı dizide ikili arama $O(\log n)$
Alan, doğruluk	ek bellek de büyük O ile ölçülür; doğru ve sağlam olmayan hızlı algoritmanın değeri yoktur

Tablo 2.21: Bölümün özeti.

Bu bölümde geliştirilen araçlar, sonraki bütün veri yapılarının işlemlerini karşılaştırmakta kullanılır. Tablo 2.22, temel veri yapılarında sık kullanılan işlemlerin maliyetlerini bir arada verir. Tablodaki yapılar dersin ilerleyen konularıdır; buradaki amaç, aynı işlemin farklı yapılarda ne kadar farklı maliyetli olabileceğini görmektir.

Veri yapısı	*i*. öğeye erişim	Arama	Ekleme	Silme
Dizi	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Bağlı liste	$O(n)$	$O(n)$	$O(1)$†	$O(1)$†
Yığın	yalnız üst	$O(n)$	$O(1)$ push	$O(1)$ pop
Kuyruk	yalnız ön	$O(n)$	$O(1)$ enqueue	$O(1)$ dequeue
Eşleme / küme (hash)	—	$O(1)$*	$O(1)$*	$O(1)$*

Tablo 2.22: Temel veri yapılarında işlem maliyetleri. † Konum (önceki düğüm) biliniyorsa. * Ortalama durum; en kötü durumda $O(n)$.

Tablonun ilk satırı bu bölümün sonuçlarıyla doğrudan ilişkilidir. Dizide *i*. öğeye erişim `ilk` metodundaki gibi adres hesabıyla yapılır: $O(1)$. Sırasız dizide arama doğrusal aramadır: $O(n)$. Dizi sıralı tutulursa arama ikili aramayla $O(\log n)$'e iner. Ortaya ekleme ve silme ise sonraki öğelerin kaydırılmasını gerektirdiği için $O(n)$'dir.

2.7 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

algoritma *algorithm* · algoritma analizi *algorithm analysis* · karmaşıklık *complexity*

zaman karmaşıklığı *time complexity* · alan karmaşıklığı *space complexity* · deneysel analiz *empirical analysis* · katlama deneyi *doubling experiment* · ilkel işlem *primitive operation* · asimptotik *asymptotic* · büyük O *Big O* · büyük Omega *Big Omega* · büyük Teta *Big Theta* · en iyi durum *best case* · ortalama durum *average case* · en kötü durum *worst case* · amortize analiz *amortized analysis* · döngü *loop* · doğrusal arama *linear search* · ikili arama *binary search* · önek ortalamaları *prefix averages* · yığın çerçevesi *stack frame* · referans *reference* · doğruluk *correctness* · değişmez *invariant* · sağlamlık *robustness*

2.8 Kendini Dene

1. `maksimum({5, 1, 7, 7, 2})` çağrısında `enBuyuk` kaç kez güncellenir? Tablo 2.3'teki sayım kurallarıyla toplam kaç ilkel işlem yapılır?

   Cevap

   Karşılaştırmalar: 1 > 5 yanlış, 7 > 5 doğru (güncelleme), 7 > 7 yanlış, 2 > 7 yanlış. Yalnız bir güncelleme yapılır: $g = 1$. İkinci 7, eşitlik olduğu için güncelleme yaptırmaz. Toplam $3n + 1 + g = 3 \cdot 5 + 1 + 1 = 17$ işlem.

2. $f(n) = 4n + 7$ için $f(n) = O(n)$ olduğunu gösteren bir c ve n_0 çifti nedir? $f(n) = \Theta(n)$ midir?

   Cevap

   $c = 5$ seçilirse $4n + 7 \le 5n$ eşitsizliği $n \ge 7$ için sağlanır: $n_0 = 7$. ($c = 11$, $n_0 = 1$ de geçerlidir.) Alt sınır için her $n \ge 1$'de $4n + 7 \ge 4n$'dir, yani $f(n) = \Omega(n)$. İkisi birlikte $f(n) = \Theta(n)$ verir.

3. Aşağıdakilerden hangileri doğrudur? (a) $n^2 + 100n = O(n^2)$ (b) $2^{n+1} = O(2^n)$ (c) $n \log n = O(n)$ (d) $4^n = O(2^n)$ (e) $\log_{10} n = \Theta(\log_2 n)$

   Cevap

   (a), (b) ve (e) doğrudur. (a): $n \ge 1$ için $n^2 + 100n \le 101n^2$. (b): $2^{n+1} = 2 \cdot 2^n$; sabit çarpan. (e): iki logaritma arasında sabit çarpan vardır. (c) yanlıştır: $n \log n / n = \log n$ oranı sınırsız büyür. (d) yanlıştır: $4^n / 2^n = 2^n$ oranı sınırsız büyür.

4. `for (int i = 1; i <= n; i *= 3) s++;` döngüsü $n = 81$ için kaç tur döner? Karmaşıklığı nedir?

Cevap

`i` 1, 3, 9, 27, 81 değerlerini alır; koşul `<=` olduğu için 81'de de gövde çalışır. Sonra `i = 243` olur ve döngü biter: 5 tur, yani $\log_3 81 + 1$. Karmaşıklık $O(\log n)$'dir; tabanın 3 olması sınıfı değiştirmez.

5. Aşağıdaki kod bittiğinde $n = 10$ için `s` kaçtır? Karmaşıklık nedir?

```
for (i = 0; i < n; i++) for (j = 0; j < 5; j++) s++;
for (k = 0; k < n; k++) for (m = 0; m < k; m++) s++;
```

Cevap

Birinci blok: iç döngü sabit 5 tur, dış döngü 10 tur: 50; $O(n)$. İkinci blok: $0 + 1 + \ldots + 9 = 10 \cdot 9 / 2 = 45$; $O(n^2)$. Toplam `s = 95`. Ardışık bloklarda en büyük kalır: $O(n^2)$.

6. `a = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}` dizisinde `ikiliArama(a, 91)` ve `ikiliArama(a, 2)` kaçar tur döner? Her turda `lo`, `hi` ve `mid` ne olur?

Cevap

91 için: (0, 9, `mid` 4, 16 < 91) → `lo = 5`; (5, 9, 7, 56) → `lo = 8`; (8, 9, 8, 72) → `lo = 9`; (9, 9, 9, 91) bulundu, 9 döner. 4 tur. 2 için: (0, 9, 4, 16 > 2) → `hi = 3`; (0, 3, 1, 5 > 2) → `hi = 0`; (0, 0, 0, 2) bulundu, 0 döner. 3 tur. İkisi de $\lfloor \log_2 10 \rfloor + 1 = 4$ sınırını aşmaz.

7. 9 öğeli bir dizide doğrusal arama yapılıyor. Hedefin her zaman dizide olduğu ve dokuz konumdan her birinde eşit olasılıkla bulunduğu varsayılırsa ortalama kaç karşılaştırma yapılır? Hedef dizide yoksa kaç karşılaştırma yapılır?

Cevap

Hedef k. konumdaysa (1'den sayarak) k karşılaştırma yapılır. Ortalama $(1 + 2 + \ldots + 9)/9 = 45/9 = 5$; bu $(n + 1)/2$ formülüyle aynıdır. Hedef yoksa dokuz öğenin hepsine bakılır: 9 karşılaştırma.

8. Bir katlama deneyinde n = 1000, 2000, 4000, 8000 için süreler 40, 81, 160 ve 322 ms ölçülmüştür. Algoritma hangi büyüme sınıfındadır? $n = 16.000$ için süre yaklaşık kaçtır?

Cevap

Oranlar $81/40 \approx 2{,}03$, $160/81 \approx 1{,}98$, $322/160 \approx 2{,}01$'dir. $2^k \approx 2$ olduğundan $k = 1$: büyüme doğrusaldır. $n = 16.000$ için süre yaklaşık $2 \cdot 322 = 644$ ms olur.

9. Bir öğrenci `maksimum`'u `enBuyuk = 0` ile başlatıp döngüyü `i = 0`'dan çalıştırıyor. Bu sürümün yanlış olduğunu gösteren bir karşı örnek nedir? Hangi değişmez bu sürümde başlangıç adımında bozulur?

   Cevap

   `{-4, -2, -9}` için metot 0 döndürür; doğru cevap −2'dir. Değişmez "her turun başında `enBuyuk`, `a[0..i-1]`'in en büyüğüdür" önermesidir. İlk turdan önce `i = 0`'dır ve `a[0..-1]` boştur; boş kümenin en büyüğü yoktur ve 0 değeri dizinin bir öğesi değildir. Başlangıç adımı sağlanmadığı için ispat çöker ve negatif sayılı dizilerde sonuç yanlış çıkar.

10. Aşağıdaki metotların ek alan karmaşıklığı nedir? (Girdi sayılmaz.) (a) `toplam` (b) `onekOrt1` (c) n öğeli bir diziden, her `(i, j)` çifti için `a[i] + a[j]` değerini tutan $n \times n$ bir matris oluşturan metot (d) `adimSay`

    Cevap

    (a) $O(1)$: yalnız `s` ve `x`. (b) $O(n)$: sonuç dizisi `a`. (c) $O(n^2)$: n^2 hücreli matris. (d) $O(1)$: yalnız `k` ve `i`.

2.9 İleri okuma

- M. T. Goodrich, R. Tamassia ve M. H. Goldwasser, *Data Structures and Algorithms in Java* (6. baskı, 2014), Bölüm 4: deneysel analizi, ilkel işlem saymayı, büyük O, Omega ve Teta gösterimlerini ve önek ortalamaları örneğini bu nottakine yakın bir sırayla ve Java koduyla anlatır.
- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 1.4: katlama deneyini ve gözleme dayalı çalışma süresi tahminini ayrıntılı örneklerle işler; deneysel ve matematiksel analizin birlikte nasıl kullanılacağını gösterir.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 2 ve 3: döngü değişmeziyle doğruluk ispatını ve asimptotik gösterimin biçimsel tanımlarını daha matematiksel bir dille verir.
- J. Bloch, "Extra, Extra — Read All About It: Nearly All Binary Searches and Mergesorts are Broken" (Google Research Blog, 2006): ikili aramadaki orta indis taşmasının gerçek bir yazılımda nasıl ortaya çıktığını anlatan kısa bir yazı.

2.10 Kaynaklar

1. J. Bloch, "Extra, Extra — Read All About It: Nearly All Binary Searches and Mergesorts are Broken", Google Research Blog, 2 Haziran 2006.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022.

3. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6. baskı, Wiley, 2014.
4. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011.

KISIM I · BÖLÜM 3

Diziler

Dizi, programlamada karşılaşılan ilk ve en temel veri yapısıdır. Aynı türden öğeleri bellekte yan yana tutar ve her öğeye numarasıyla, tek adımda ulaşmayı sağlar. Öteki veri yapılarının çoğu ya bir dizinin üzerine kurulur ya da dizinin zayıf kaldığı bir noktayı gidermek için ortaya çıkar. Bu yüzden dizinin bellekte nasıl durduğunu, hangi işlemde hızlı, hangisinde yavaş olduğunu iyi anlamak gerekir. Not, dizi kavramıyla ve dizinin bellekteki yerleşimiyle başlar. Java'da dizilerin tanımlanmasını, kopyalanmasını, stack ve heap bellekteki yerini, sık görülen hataları anlatır. Ardından ortalama, ortanca, mod, tersine çevirme ve Sezar şifresi gibi dizi algoritmalarını kod izlemeleriyle işler. Çok boyutlu diziler, matris işlemleri ve küçük bir Tic Tac Toe oyunu bunu izler. Son kesim dizi işlemlerinin zaman ve alan maliyetini, arama, ekleme, silme ve iki basit sıralama algoritması üzerinden inceler.

Bu bölümün kodu: `github.com/sercankulcu/data-structures-java/Bolum03/src`

ÖĞRENME ÇIKTILARI

Dizi modeli
Dizilerin bellekteki yerleşimini ve indisleme mantığını açıklamak; bir öğenin adresinin neden tek adımda hesaplandığını göstermek.

Java kullanımı
Tek ve çok boyutlu dizileri doğru biçimde tanımlayıp kullanmak; referans ile kopya, sığ kopya ile derin kopya arasındaki farkı kodda gözetmek.

Algoritmalar
Arama, ekleme, silme, tersine çevirme ve sıralama işlemlerini diziler üzerinde uygulamak ve elle izlemek.

Maliyet analizi
Dizi işlemlerinin zaman ve alan maliyetlerini gerekçesiyle vermek ve karşılaştırmak.

3.1 Dizi kavramı

Bir sınıftaki 40 öğrencinin sınav notlarını saklayan bir program düşünelim. Her not için ayrı bir değişken açmak (`not1`, `not2`, …, `not40`) hem yorucudur hem de işe yaramaz. Notların ortalamasını almak için kırk değişkeni tek tek yazmak gerekir. Öğrenci sayısı 41'e çıkınca program da değişmek zorunda kalır. Asıl ihtiyaç, kırk notu tek bir ad altında toplamak ve her birine bir sıra numarasıyla ulaşmaktır. Dizi tam olarak bu ihtiyacı karşılar.

3.1.1 Dizi nedir?

Dizi (*array*), aynı türden birden çok veriyi tek bir ad altında, düzenli biçimde saklayan yapıdır. Dizideki her veriye **öğe** (*element*) denir. Öğeler bir sıra içinde durur ve her birinin bir sıra numarası vardır. Bu numaraya **indis** (*index*) denir. Java'da, C'de ve pek çok dilde indis 0'dan başlar: ilk öğenin indisi 0, ikincininki 1'dir.

> **Tanım 3.1**
>
> **Dizi**, aynı türden sabit sayıda öğeyi bellekte **ardışık** (*contiguous*), yani yan yana yuvalarda tutan veri yapısıdır. *n* öğeli bir dizinin öğelerine 0, 1, …, *n* − *1* indisleriyle erişilir. Öğe sayısı *n*'ye dizinin **boyutu** ya da uzunluğu denir.

Tanımdaki her ifadenin bir sonucu vardır:

- **Aynı tür.** Bir `int` dizisinde yalnız tam sayılar, bir `String` dizisinde yalnız metinler durur. Bütün öğeler aynı türden olduğu için bellekte aynı sayıda bayt kaplar. Bu eşitlik, bir öğenin yerini hesaplamayı kolaylaştırır.
- **Sabit boyut.** Dizinin boyutu dizi oluşturulurken belirlenir ve sonradan değiştirilemez. Beş öğelik bir diziye altıncı bir hücre eklenemez. Daha çok yer gerekirse daha büyük yeni bir dizi oluşturulur ve eski öğeler ona kopyalanır.
- **Ardışık bellek.** Öğeler bellekte arada boşluk olmadan, art arda dizilir. 0. öğenin hemen arkasında 1. öğe, onun arkasında 2. öğe gelir.

Boyutun sabit olması, dizideki değerlerin de sabit olduğu anlamına gelmez. Öğelerin değerleri her zaman güncellenebilir: 2. hücredeki 4, istenince 40 yapılabilir. Sabit olan, hücre sayısıdır. Bu yüzden dizide “ekleme” ve “silme”nin anlamı biraz farklıdır. Araya bir öğe eklemek, sağdaki öğeleri birer hücre kaydırıp yer açmak demektir. Bir öğeyi silmek de sağdaki öğeleri sola kaydırıp boşluğu kapatmaktır. Bu kaydırmaların maliyeti 3.5.5 ve 3.5.6'da ayrıntılı incelenir.



Şekil 3.1: Beş öğeli bir dizi. Üst sırada indisler (0'dan 4'e), alt sırada öğeler (13, −1, 4, 2, 7) vardır. Dizinin uzunluğu 5'tir; ilk indis 0, son indis 4'tür.

Şekil 3.1, diziyi düşünmenin en kullanışlı yolunu gösterir: yan yana dizilmiş, numaralı kutular. Kutuların içinde değerler, üstlerinde indisler durur. Değer ile indisi karıştırmamak önemlidir. Şekildeki dizide 2 indisli öğe 4'tür; değeri 2 olan öğe ise 3 indislidir.

Diziler, aynı türden çok sayıda verinin bir arada işlendiği her yerde kullanılır. Birkaç tanıdık örnek:

- **Matris.** Satır ve sütunlardan oluşan sayı tabloları, iki boyutlu dizilerle tutulur (3.4).
- **Resim.** Dijital bir resim, piksellerden oluşan bir ızgaradır. Her piksel bir renk değeri taşır ve ızgara iki boyutlu bir dizide saklanır.
- **Ses.** Kaydedilmiş bir ses, saniyede binlerce kez ölçülen genlik değerlerinden, yani örneklerden oluşur. Bu örnekler zaman sırasıyla tek boyutlu bir dizide durur.

3.1.2 İndisle erişim ve adres hesabı

Dizinin bir öğesine ulaşmak için dizinin adı ve köşeli parantez içinde indis yazılır: `diziAdi[indis]`. Aşağıdaki kod beş öğeli bir dizi oluşturur, ilk ve son öğeyi okur:

```
int[] dizi = {7, 2, 9, 5, 1};
int ilkOge = dizi[0];      // 7
int sonOge = dizi[dizi.length - 1];      // 1
```

İlk öğenin indisi 0'dır, bu yüzden `dizi[0]` 7'yi verir. Son öğenin indisi ise uzunluğun bir eksiğidir. Java'da her dizi kendi uzunluğunu `length` alanında taşır; burada `dizi.length` 5'tir ve son öğe `dizi[dizi.length - 1]`, yani `dizi[4]` = 1'dir. Son öğeye `dizi[4]` diye sabit bir sayıyla değil de `length` üzerinden ulaşmak, dizinin boyu değişse de doğru çalışan kod yazmayı sağlar.

İndisin neden 0'dan başladığı, öğelerin bellekte nasıl bulunduğuyla ilgilidir. Bellek, her birinin bir numarası (adresi) olan baytlardan oluşur. Dizi oluşturulduğunda, bütün öğeleri sığacak kadar ardışık bir bellek bölgesi ayrılır. Bu bölgenin ilk baytının adresine dizinin **başlangıç adresi** denir. Bir `int` 4 bayt kapladığı için 0. öğe başlangıç adresinde, 1. öğe 4 bayt ötede, 2. öğe 8 bayt ötede durur. Genel olarak *i*. öğenin adresi şöyle hesaplanır:

> *adres(i) = başlangıç + i × öğe boyutu*

İndis 0'dan başladığı için ilk öğenin adresi doğrudan başlangıç adresidir; formülde ek bir "−1" gerekmez. İndis, aslında öğenin başlangıçtan kaç öğe uzakta olduğunu söyler.

Örnek 3.1

Bir `int` dizisi bellekte 1000 adresinden başlasın. `int` 4 bayt olduğuna göre öğelerin adresleri şöyledir:

1. `dizi[0]` : 1000 + 0 × 4 = 1000.
2. `dizi[1]` : 1000 + 1 × 4 = 1004.
3. `dizi[2]` : 1000 + 2 × 4 = 1008.
4. `dizi[4]` : 1000 + 4 × 4 = 1016.

Hangi indis istenirse istensin, adres bir çarpma ve bir toplamayla bulunur. 2. öğeye ulaşmak için 0. ve 1. öğelere uğramak gerekmez.

Bu hesap, dizinin en büyük üstünlüğünün kaynağıdır. Bir öğeye erişmenin süresi, dizinin kaç öğeli olduğuna ve istenen öğenin nerede durduğuna bağlı değildir. Beş öğeli bir dizide de beş milyon öğeli bir dizide de `dizi[i]` aynı sürede okunur. Bu özelliğe **rastgele erişim** (*random access*) denir; buradaki "rastgele", "istenen herhangi bir öğeye doğrudan" anlamındadır. Erişimin maliyeti *O(1)*'dir (3.5.2).

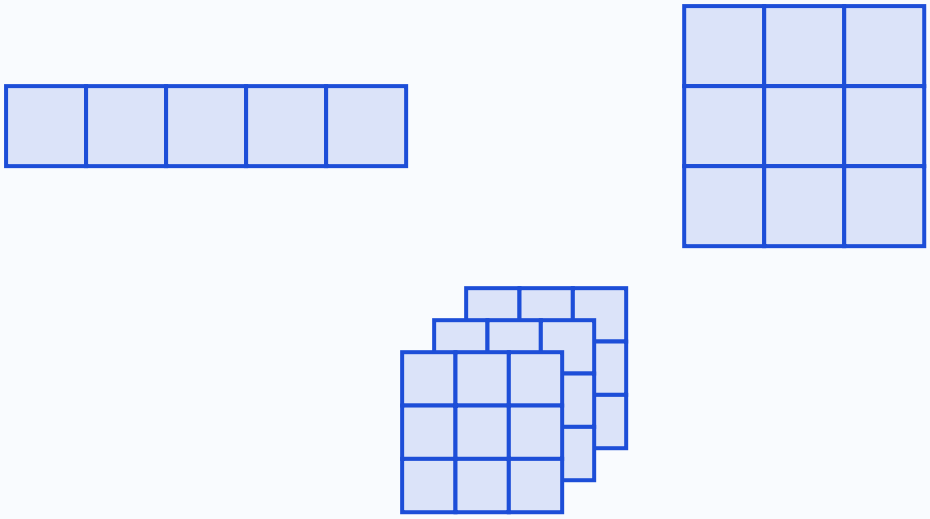
Sık yapılan hata

Son öğeye `dizi[dizi.length]` ile ulaşmaya çalışmak. 5 öğeli dizide geçerli indisler 0, 1, 2, 3, 4'tür; 5 indisli bir hücre yoktur. İnsanlar saymaya 1'den başladığı için "beşinci öğe"nin indisini 5 sanmak kolaydır. Beşinci öğenin indisi 4'tür. Java bu hatayı çalışma sırasında yakalar ve bir istisna fırlatır (3.2.12).

3.1.3 Dizi türleri

Diziler, öğelerin kaç yönde dizildiğine göre adlandırılır. Şekil 3.2 üç türü yan yana gösterir.

- **Tek boyutlu dizi.** Öğeler tek bir satırda saklanır ve her öğeye tek bir indisle ulaşılır. Şimdiye kadarki örneklerin hepsi tek boyutludur. Java'da tanımı: `int[] a = new int[5];`
- **İki boyutlu dizi** (*two-dimensional array*). Öğeler satırlar ve sütunlar hâlinde, bir tablo gibi saklanır. Böyle bir tabloya **matris** (*matrix*) de denir. Her öğeye iki indisle, önce satır sonra sütun numarasıyla ulaşılır. Tanımı: `int[][] m = new int[3][3];`
- **Çok boyutlu dizi** (*multidimensional array*). İkiden çok indis gerektiren düzenlerdir. Örneğin bir videonun kareleri zaman × satır × sütun biçiminde üç indisle tutulabilir. Tanımı: `int[][][] k = new int[3][3][3];`



Şekil 3.2: Dizi türleri. Solda tek boyutlu dizi: öğeler tek satırdadır. Ortada iki boyutlu dizi: 3 satır ve 3 sütun. Sağda üç boyutlu dizi: üst üste duran üç tane 3 × 3'lük katman.

Her yeni boyut, Java tanımında bir çift köşeli parantez ekler. Öğe sayısı da boyutların çarpımıdır: `new int[3][3]` 9, `new int[3][3][3]` 27 öğe içerir. Çok boyutlu dizilerin Java'da bellekte nasıl tutulduğu ilk bakışta sanıldığı gibi değildir; bu konu 3.4.2'de işlenir.

3.1.4 Avantajlar ve dezavantajlar

Dizi her işe uygun bir yapı değildir. Bir problem için veri yapısı seçerken, dizinin nerede güçlü, nerede zayıf olduğunu bilmek gerekir. Tablo 3.1 iki tarafı karşılaştırır.

Avantajları	Dezavantajları
İndisle doğrudan, hızlı erişim: *O(1)*.	Sabit boyutludur; dolunca büyütülemez, yenisi oluşturulup öğeler kopyalanır.
Verileri gruplar ve eklenme sırasını korur.	Araya ekleme ve silme öğe kaydırmayı gerektirir: *O(n)*.
Hem ilkel tipleri (`int`, `char` ...) hem nesneleri tutabilir.	Kullanılmayan kapasite boşa gider.
Ardışık bellek: ek referans yükü yoktur, önbellek dostudur.	Dizinin sıralama, arama gibi metotları yoktur; bunlar için `java.util.Arrays` kullanılır.
—	Java dizileri kovaryanttır: bu hem esneklik hem de bir tür güvenliği açığı getirir.

Tablo 3.1: Dizilerin avantajları ve dezavantajları.

Avantajların çoğu ardışık bellekten gelir. Erişimin hızlı olmasının nedeni 3.1.2'deki adres hesabıdır. “Ek referans yükü yok” ifadesi, bağlı yapılarla karşılaştırınca daha iyi anlaşılır. Bağlı bir listede her öğe, bir sonraki öğenin yerini gösteren bir referansı da yanında taşır; bu referanslar ek bellek kaplar. Dizide ise sıradaki öğenin yeri zaten bellidir: hemen yanıdır. Böylece bellek yalnız veriye harcanır.

Önbellek (*cache*) dostluğu da ardışıklıktan doğar. İşlemci ana bellekten veriyi tek tek bayt olarak değil, bloklar hâlinde okur ve bu blokları işlemcinin yanındaki küçük ve çok hızlı önbellekte tutar. Bir dizinin 0. öğesi okunduğunda, aynı bloktaki 1., 2., 3. öğeler de önbelleğe gelmiş olur. Dizi baştan sona gezilirken sonraki öğelerin çoğu önbellekte hazır bekler. Bu yüzden aynı sayıda adım atan iki algoritmadan diziyle çalışanı pratikte çoğu zaman daha hızlıdır.

Dezavantajların çoğu da aynı kaynaktan, sabit boyut ve ardışıklıktan gelir. Dizinin ortasına bir öğe eklemek için yer açmak gerekir, çünkü öğeler arasında boşluk yoktur. Dizi dolunca sonuna bir hücre eklenemez, çünkü dizinin hemen arkasındaki bellek başka bir veriye ait olabilir. Öğe sayısı önceden bilinmiyorsa geniş bir dizi ayırmak gerekir ve kullanılmayan hücreler boşa yer kaplar.

Tablonun son satırındaki **kovaryans** kavramı biraz daha açıklama ister.

3.1.5 Dizi kovaryansı ve ArrayStoreException

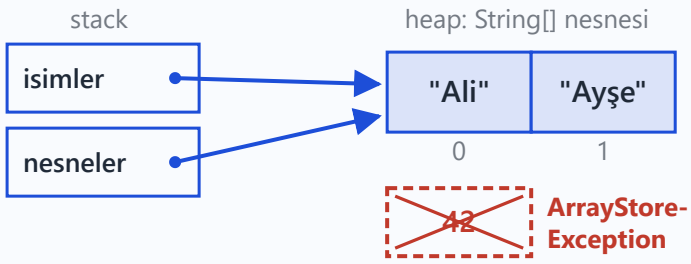
Java'da sınıflar arasında bir alt tür ilişkisi vardır. Örneğin `String`, `Object`'in alt türüdür: her `String` aynı zamanda bir `Object`'tir. Bu yüzden `Object o = "Ali";` ataması geçerlidir. Soru şudur: `String`'ler dizisi de `Object`'ler dizisinin alt türü sayılır mı? Java'nın cevabı "evet"tir.

> **Tanım 3.2**
>
> `S`, `T`'nin alt türü olduğunda `S[]` de `T[]`'nin alt türü sayılıyorsa dizilere **kovaryant** (*covariant*) denir. Bu özelliğe **dizi kovaryansı** (*array covariance*) denir. Java'da referans türlerinden oluşan diziler kovaryanttır: `String[]`, `Object[]`'in alt türüdür.

Kovaryansın kazancı esnekliktir. `static void yazdir(Object[] a)` biçiminde yazılmış tek bir metot, `String[]`, `Integer[]` ya da başka herhangi bir referans dizisiyle çağrılabilir. Her tür için ayrı bir metot yazmak gerekmez. Ama bu esnekliğin bir bedeli vardır. Aşağıdaki kodu inceleyelim:

```
String[] isimler = {"Ali", "Ayşe"};
Object[] nesneler = isimler; // derlenir
nesneler[0] = 42;   // derlenir, ama
                    // çalışırken istisna!
```

Birinci satır iki öğeli bir `String` dizisi oluşturur. İkinci satırda `nesneler` değişkenine aynı dizinin referansı atanır; kovaryans sayesinde bu atama derlenir. Artık iki değişken heap bellekteki tek bir diziyi gösterir. Üçüncü satırda `nesneler[0] = 42` yazılır. Derleyicinin gözünde `nesneler` bir `Object[]`'dir ve 42 (bir `Integer` olarak) bir `Object`'tir; bu yüzden derleyici itiraz etmez. Oysa nesnenin kendisi gerçekte bir `String[]`'dir ve içine tam sayı konamaz. Java sanal makinesi (*Java Virtual Machine*, JVM) bunu çalışma sırasında, atamanın yapıldığı anda fark eder ve `ArrayStoreException: java.lang.Integer` fırlatır.



Şekil 3.3: `isimler` ve `nesneler` değişkenleri stack bellekte durur ve heap bellekteki aynı `String[]` nesnesini gösterir. `nesneler[0] = 42` ataması çalışma zamanında reddedilir ve `ArrayStoreException` fırlatılır.

Hatanın bir an önce, kod derlenirken yakalanması her zaman daha iyidir. Derleme hatası programcının masasında görülür; çalışma zamanı hatası ise belki aylar sonra, programı kullanan birinin karşısına çıkar. Dizi kovaryansı, bir tür hatasını derleme zamanından çalışma zamanına kaydırır. Tür güvenliğindeki "açık" budur.

Java'nın sonradan eklenen **jenerik** (*generic*) yapıları bu açığı kapatacak biçimde tasarlanmıştır. Jenerik listeler kovaryant değildir: `List<String>`, `List<Object>`'in alt türü sayılmaz. Bu yüzden `List<Object> l = new ArrayList<String>();` satırı derlenmez. Aynı hata, dizideki gibi çalışırken değil, derleme sırasında yakalanır.

> **Sık yapılan hata**
>
> "Kod derlendiyse türler doğrudur" diye düşünmek. Dizilerde bu her zaman geçerli değildir. Bir `Object[]` değişkeni gerçekte daha dar türden bir diziyi gösterebilir ve yanlış türde bir atama ancak çalışırken fark edilir. Bir diziyi `Object[]` olarak alan bir metot, o diziye yeni öğe yazıyorsa bu riski taşır.

3.2 Java'da diziler

Dizi kavramı bütün programlama dillerinde aynıdır, ama her dil diziyi kendine göre tanımlar ve bellekte kendine göre yerleştirir. Java'da dizinin bir nesne olması pek çok davranışı belirler: dizinin nerede durduğunu, atamanın ne kopyaladığını, bir metoda dizi verildiğinde neyin aktarıldığını. Bu kesim Java'daki dizileri sözdiziminden başlayarak bellek düzenine kadar anlatır.

3.2.1 Dizi tanımlama ve varsayılan değerler

Java'da bir dizi şu kalıpla tanımlanır ve oluşturulur:

```
VeriTipi[] diziAdi = new VeriTipi[diziBoyutu];
```

Kalıbın üç parçası vardır:

- `VeriTipi`: Dizide saklanacak verilerin türüdür; `int`, `double`, `String` gibi. Türün yanındaki `[]`, değişkenin tek bir değeri değil, bir diziyi gösterdiğini belirtir.
- `diziAdi`: Diziyi gösteren referans değişkenin adıdır; örneğin `sayilar`.
- `diziBoyutu`: Öğe sayısıdır. Sabit bir sayı olmak zorunda değildir; `5`, `n` ya da `2 * n` gibi, program çalışırken hesaplanan bir ifade de olabilir.

Eşittir işaretinin iki yanı ayrı işler yapar. Sol taraf, `diziAdi` adında bir **referans** (*reference*) değişkeni tanımlar. Referans, bir nesnenin bellekteki yerini tutan değerdir; nesnenin kendisi değildir. Sağ taraftaki `new VeriTipi[diziBoyutu]` ise bellekte o kadar öğelik yeni bir dizi nesnesi oluşturur. Atama, yeni dizinin referansını değişkene yazar. Bu ayrım ilk bakışta önemsiz görünür, ama 3.2.6'daki kopyalama konusunun temelidir.

```
int[] sayilar = new int[5];
// → [0, 0, 0, 0, 0]
```

Bu satır beş öğeli bir `int` dizisi oluşturur. Öğelere henüz değer atanmamıştır, ama boş da değildirler: `new` ile oluşturulan dizinin her öğesi, türünün **varsayılan değerini** alır. `int` için bu değer 0'dır; dizi `[0, 0, 0, 0, 0]` durumundadır. Tablo 3.2 türlere göre varsayılan değerleri verir.

Tür	Varsayılan değer
`int`, `long`, `short`, `byte`	`0`
`double`, `float`	`0.0`
`boolean`	`false`
`char`	`'\u0000'` (kodu 0 olan karakter)
nesne (`String` ...)	`null`

Tablo 3.2: `new` ile oluşturulan dizide öğelerin varsayılan değerleri.

Varsayılan değer kuralı, yerel değişkenlerle diziler arasındaki önemli bir farkı gösterir. Bir metodun içinde `int x;` yazıp `x`'e değer vermeden okumak derleme hatasıdır. Dizi öğeleri ise hiçbir zaman değersiz kalmaz; JVM diziyi oluştururken bütün hücreleri sıfırlar. Bu sıfırlama da bir iştir: *n* öğeli bir dizinin oluşturulması *O(n)* sürer.

`char` dizisinin varsayılanı olan `'\u0000'`, ekranda görünmeyen ve boşluk karakterinden farklı olan özel bir karakterdir. `String` dizisinin varsayılanı ise boş metin (`""`) değil, `null`'dır. `null`, "hiçbir nesneyi göstermiyor" anlamına gelen referans değeridir. Bu öğelerden biri üzerinden bir metot çağrılırsa (örneğin `isimler[0].length()`) `NullPointerException` fırlatılır.

Sık yapılan hata

`String[] isimler = new String[3];` satırından sonra dizinin üç boş metin içerdiğini sanmak. Dizide üç `null` vardır, metin nesnesi yoktur. Önce her hücreye bir metin atanmalıdır; ancak ondan sonra `isimler[i].length()` gibi çağrılar güvenlidir.

3.2.2 Erişim, ilklendirme ve length

Öğelerin değerleri baştan biliniyorsa diziyi oluşturup hücreleri tek tek doldurmak gerekmez. Değerler süslü parantez içinde, virgülle ayrılarak yazılabilir. Bu listeye **ilklendirici** (*initializer*) listesi denir. Aşağıdaki kod ilklendirmeyi, okumayı, yazmayı ve `length`'i bir arada gösterir:

```
int[] sayilar = {10, 20, 30, 40, 50};   // 5 ilklendirici → boyut 5
int ilkOge    = sayilar[0];             // 10
int ikinciOge = sayilar[1];             // 20
sayilar[4] = 55;                          // son öğe güncellendi
int diziBoyutu   = sayilar.length;      // 5

sayilar = new int[] {1, 2, 3};          // sonradan atamada new int[] gerekir
int[] hatali = new int[3] {1, 2, 3};    // derleme hatası: boyut + liste
```

Kodu satır satır inceleyelim:

- **1. satır:** İlklendirici listesinde beş değer vardır. Listede boyut yazılmaz; öğe sayısı boyutu belirler. Java beş öğeli bir `int` dizisi oluşturur, değerleri sırayla hücrelere yazar ve referansı `sayilar`'a atar.
- **2. ve 3. satır:** Okuma. `sayilar[0]` ilk öğedir (10), `sayilar[1]` ikinci öğedir (20).
- **4. satır:** Yazma. Okumayla aynı sözdizimi kullanılır; tek fark, ifadenin atamanın solunda durmasıdır. Son hücrenin (indis 4) değeri 50'den 55'e değişir.
- **5. satır:** `sayilar.length` dizinin öğe sayısını, yani 5'i verir. `length` bir metot değil, dizinin bir **alanıdır**; bu yüzden ardından parantez gelmez.
- **7. satır:** Değişkene sonradan yeni bir dizi atanıyor. İlklendirici listesi yalnız tanımlama satırında tek başına yazılabilir. Başka bir yerde kullanılacaksa önüne `new int[]` eklenir.
- **8. satır:** Derleme hatası. Hem boyut (`3`) hem liste verilemez, çünkü boyutu zaten liste belirler. Doğru yazım `new int[] {1, 2, 3}`'tür.

Sık yapılan hata

Dizide `length()`, metinde `length` yazmak. İkisi ters yönde karışır: dizinin uzunluğu bir alandır (`a.length`), `String`'in uzunluğu ise bir metottur (`s.length()`). Yanlış yazım derleme hatası verir.

3.2.3 Kod izleme: dizide okuma ve yazma

Bir önceki kodun bellekte ne yaptığını adım adım izleyelim. İzlenen satırlar şunlardır:

```
int[] sayilar = {10, 20, 30, 40, 50};
int ilkOge    = sayilar[0];
int ikinciOge = sayilar[1];
sayilar[4] = 55;
int diziBoyutu   = sayilar.length;
sayilar = new int[] {1, 2, 3};
```

Satır	Yapılan iş	Değişkenler	`sayilar`
1	heap bellekte 5 öğeli dizi oluşur	—	[10, 20, 30, 40, 50]
2	`sayilar[0]` okunur	`ilkOge` = 10	[10, 20, 30, 40, 50]
3	`sayilar[1]` okunur	`ikinciOge` = 20	[10, 20, 30, 40, 50]
4	`sayilar[4]` 'e 55 yazılır	—	[10, 20, 30, 40, **55**]
5	`length` okunur	`diziBoyutu` = 5	[10, 20, 30, 40, 55]
6	yeni 3 öğeli dizi oluşur, `sayilar` onu gösterir	`diziBoyutu` hâlâ 5	[1, 2, 3]

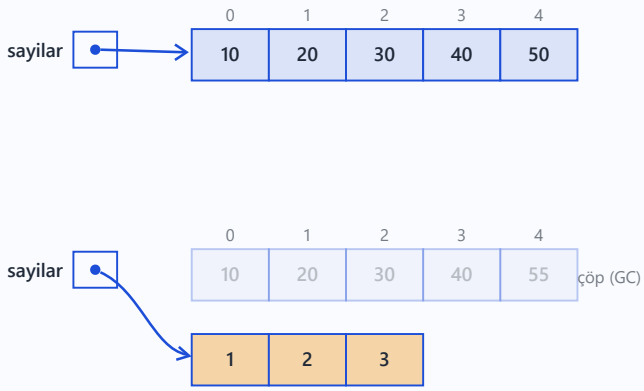
Tablo 3.3: Okuma ve yazma kodunun izlemesi. Son sütun, `sayilar`'ın gösterdiği dizinin durumudur.

İlk satırda iki ayrı şey oluşur. Heap bellekte beş öğeli bir `int[]` nesnesi oluşur. `sayilar` değişkeni ise bu nesneyi değil, yalnız onun referansını tutar (Şekil 3.4, solda). Değişkenden diziye uzanan ok bu referansı gösterir.

2. satırdaki okuma, 3.1.2'deki adres hesabıyla yapılır: başlangıç + 0 × 4. Okuma tek adımda biter ve dizinin boyutu ne olursa olsun süresi değişmez. 3. satır aynı işi 1 indisi için yapar. 4. satırda aynı sözdizimi atamanın solunda yer alır ve bu kez okuma değil yazma yapılır: son hücrenin değeri 55 olur. 5. satır dizinin uzunluğunu okur. Bu değer dizi oluşturulurken sabitlenmiştir ve dizinin ömrü boyunca değişmez.

Son satır en öğretici olanıdır. `new int[] {1, 2, 3}` heap bellekte üç öğeli *yeni* bir dizi oluşturur ve `sayilar` artık onu gösterir. Eski dizi değişmemiştir; hâlâ [10, 20, 30, 40, 55] değerlerini taşır. Ama artık onu gösteren hiçbir referans kalmamıştır. Program o diziye bir daha hiçbir yoldan ulaşamaz. Java'nın **çöp toplayıcısı** (*garbage collector*, GC) böyle erişilemeyen nesneleri bulup belleklerini geri alır.

`diziBoyutu`'nun hâlâ 5 olması da dikkat çekicidir. 5. satırda `diziBoyutu`'na `sayilar.length`'in o anki *değeri* kopyalanmıştı. `diziBoyutu` diziyle bağlantılı bir değişken değildir; dizi değişince kendiliğinden güncellenmez.



Şekil 3.4: Okuma/yazma izlemesinin iki anı. Solda 1. satırdan sonra: `sayilar` referansı heap bellekteki [10, 20, 30, 40, 50] dizisini gösterir. Sağda son satırdan sonra: `sayilar` yeni [1, 2, 3] dizisini gösterir; soluk çizilen eski dizi [10, 20, 30, 40, 55] artık erişilemez; çöp toplayıcı onu toplayabilir.

3.2.4 Örnek kod: tanımla, doldur, yazdır

Aşağıdaki tam program bir dizi oluşturur, bir döngüyle doldurur ve iki ayrı biçimde yazdırır:

```
public class DiziOrnegi {
    public static void main(String[] args) {
        int[] sayilar = new int[5];            // 5 öğeli dizi
        for (int i = 0; i < sayilar.length; i++)
            sayilar[i] = (i + 1) * 10;             // 10, 20, 30, 40, 50

        System.out.println("Dizinin birinci öğesi: " + sayilar[0]);
        System.out.println("Dizinin üçüncü öğesi: " + sayilar[2]);

        System.out.print("Dizi Öğeleri: ");
        for (int x : sayilar)                      // for-each: indis gerekmiyorsa
            System.out.print(x + " ");
    }
}
```

Programın çıktısı:

```
Dizinin birinci öğesi: 10
Dizinin üçüncü öğesi: 30
Dizi Öğeleri: 10 20 30 40 50
```

Programın yapısı, dizilerle çalışan çoğu programın iskeletidir:

- **Oluşturma.** `new int[5]` beş öğeli, sıfırlarla dolu bir dizi oluşturur.

- **Doldurma.** `for` döngüsü `i`'yi 0'dan başlatır ve `i < sayilar.length` olduğu sürece artırır. Her turda `sayilar[i]`'ye `(i + 1) * 10` yazılır; sonuçta dizi 10, 20, 30, 40, 50 olur.
- **Tek öğe yazdırma.** `sayilar[0]` ve `sayilar[2]` okunur. Üçüncü öğenin indisi 2'dir.
- **Bütün öğeleri yazdırma.** İkinci döngü bir **for-each döngüsüdür**: `for (int x : sayilar)`, "`sayilar`'daki her `x` için" diye okunur. Döngü değişkeni `x`, sırayla her öğenin değerini alır.

Beş ayrı `sayilar[k] = …` ataması yazmak yerine döngü kullanmanın asıl kazancı, kodun dizinin boyutuna bağlı olmamasıdır. Dizi 5 yerine 500 öğeli olsa da döngü değişmeden çalışır. Bunun koşulu, döngü sınırının sabit bir sayı değil `sayilar.length` olmasıdır. Sınırı `i < 5` yazmak bugün çalışır, ama boyut değiştiğinde ya diziyi yarım doldurur ya da sınır dışına taşar.

Programda iki ayrı yazdırma metodu kullanılır. `System.out.println` yazdıktan sonra satır sonu ekler; sonraki çıktı yeni satıra geçer. `System.out.print` ise satır sonu eklemez; ardından gelen çıktı aynı satırda devam eder. Bu yüzden son döngüdeki öğeler tek satırda, aralarında birer boşlukla yazılır.

for-each döngüsü, indise ihtiyaç olmadığında daha okunaklı bir seçenektir. Ama iki sınırı vardır. Birincisi, öğenin indisi bilinmez; "hangi indiste bulundu" sorusunu for-each ile cevaplamak için ayrı bir sayaç gerekir. İkincisi, `x` öğenin kendisi değil, değerinin bir kopyasıdır. Döngü içinde `x = 0` yazmak dizideki değeri değiştirmez. Diziye yazmak gereken her durumda indisli `for` döngüsü kullanılır.

3.2.5 Kod izleme: doldur ve yazdır

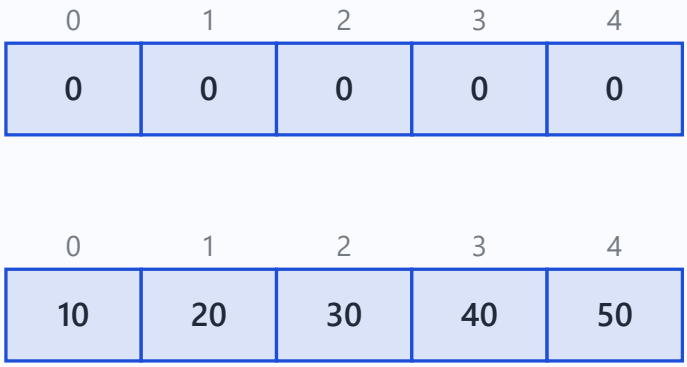
Programın gövdesini adım adım izleyelim. Tablo 3.4 doldurma döngüsünü, ardından gelen yazdırmaları gösterir.

Adım	Koşul / işlem	`sayilar`	Ekrana
`new int[5]`	5 hücre ayrılır	[0, 0, 0, 0, 0]	—
`i = 0`	0 < 5 → `sayilar[0]` = 1 × 10	[10, 0, 0, 0, 0]	—
`i = 1`	1 < 5 → `sayilar[1]` = 2 × 10	[10, 20, 0, 0, 0]	—
`i = 2`	2 < 5 → `sayilar[2]` = 3 × 10	[10, 20, 30, 0, 0]	—
`i = 3`	3 < 5 → `sayilar[3]` = 4 × 10	[10, 20, 30, 40, 0]	—
`i = 4`	4 < 5 → `sayilar[4]` = 5 × 10	[10, 20, 30, 40, 50]	—
`i = 5`	5 < 5 yanlış → döngü biter	[10, 20, 30, 40, 50]	—
1. `println`	`sayilar[0]` okunur	değişmez	Dizinin birinci öğesi: 10
2. `println`	`sayilar[2]` okunur	değişmez	Dizinin üçüncü öğesi: 30
`print`	satır sonu yok	değişmez	Dizi Öğeleri:
for-each	`x` = 10, 20, 30, 40, 50	değişmez	10 20 30 40 50

Tablo 3.4: `DiziOrnegi`'nin izlemesi.

Döngünün son denetimi kolayca gözden kaçar. `i` 5 olduğunda `5 < 5` yanlıştır ve gövde çalışmaz; `sayilar[5]`'e hiç dokunulmaz. Koşulda `<` yerine `<=` yazılsaydı, son turda var olmayan 5 indisli hücreye yazılmaya çalışılır ve program bir istisnayla dururdu.

Gövde 5 kez çalışmıştır ve her turda bir çarpma ile bir atama yapılmıştır. Öğe sayısı *n* olsaydı gövde *n* kez çalışırdı. Dizinin bütün öğelerini bir kez gezen her döngünün süresi *n* ile doğru orantılı büyür. Böyle bir döngünün maliyeti *O(n)*'dir (3.5).



Şekil 3.5: Doldurma döngüsünün başı ve sonu. Solda `new int[5]`'ten sonra bütün hücreler 0'dır. Sağda döngü bittiğinde dizi 10, 20, 30, 40, 50 değerlerini taşır.

3.2.6 Dizi kopyalama: referans mı, kopya mı?

Java'da dizi bir **nesnedir**. Bütün nesneler gibi heap bellekte oluşur ve bir değişkene atandığında değişkene nesnenin kendisi değil, referansı yazılır. Bu kuralın en çok şaşırtan sonucu, atama işlecinin (`=`) diziyi kopyalamamasıdır. Aşağıdaki kod bu farkı gösterir:

```
int[] dizi = {1, 2, 3, 4, 5};
int[] referans = dizi;   // aynı nesne
int[] kopya =
    Arrays.copyOf(dizi, dizi.length);
referans[0] = 99;
```

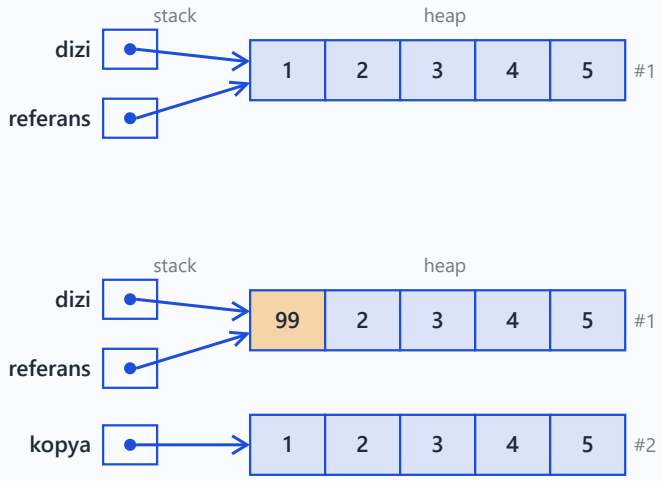
Kodu dört adımda izleyelim. Heap bellekteki dizileri ayırt etmek için onlara #1 ve #2 diye numara verelim.

Örnek 3.2

1. `int[] dizi = {1, 2, 3, 4, 5};` Heap bellekte #1 dizisi oluşur. Stack bellekteki `dizi` değişkeni yalnız #1'in adresini tutar.
2. `int[] referans = dizi;` `=` yalnız adresi kopyalar. `referans` da #1'i gösterir. Yeni bir dizi oluşmamıştır; aynı diziye iki ayrı ad verilmiştir.

3. `int[] kopya = Arrays.copyOf(dizi, dizi.length);` `Arrays.copyOf` heap bellekte yeni bir dizi (#2) oluşturur ve #1'in öğelerini tek tek ona kopyalar. Öğe sayısı *n* ise bu iş *O(n)* sürer. `kopya` #2'yi gösterir.
4. `referans[0] = 99;` `referans` #1'i gösterdiği için #1'in ilk hücresi 99 olur. `dizi` de #1'i gösterdiğinden `dizi[0]` da artık 99'dur. `kopya[0]` ise ayrı nesnede durur ve 1 olarak kalır.

Sonuç: `dizi` = [99, 2, 3, 4, 5], `kopya` = [1, 2, 3, 4, 5].



Şekil 3.6: Referans ile kopya farkı. Solda `referans = dizi` atamasından sonra iki değişken de heap bellekteki aynı #1 dizisini gösterir. Sağda `Arrays.copyOf` ayrı bir #2 dizisi oluşturmuştur; `referans[0] = 99` yalnız #1'i değiştirir ve değişiklik `dizi` üzerinden de görünür.

Gerçek bir kopya elde etmenin üç yaygın yolu vardır. Üçü de yeni bir dizi oluşturur ve öğeleri tek tek kopyalar; üçünün maliyeti de *O(n)*'dir:

- `Arrays.copyOf(dizi, yeniBoyut)` : İstenen boyutta yeni bir dizi döndürür. Yeni boyut eskisinden büyükse fazla hücreler varsayılan değeri alır; küçükse dizi kırpılır. Bu yüzden dolan bir diziyi büyütmek için de kullanılır.
- `dizi.clone()` : Aynı boyutta bir kopya döndürür.
- `System.arraycopy(kaynak, kaynakBas, hedef, hedefBas, adet)` : Var olan bir hedef diziye, istenen konumdan başlayarak istenen sayıda öğe kopyalar.

Aynı kural metot çağrılarında da geçerlidir. Bir metoda dizi verildiğinde metoda dizinin referansı aktarılır. Metot dizinin içindeki bir değeri değiştirirse, değişiklik çağıran yerde de görünür; çünkü ikisi aynı nesneye bakar. 3.3.4'teki `ortanca` metodunun diziyi sıralamadan önce kopyalaması bu yüzdendir.

Sık yapılan hata

`int[] b = a;` yazıp `b`'nin `a`'nın kopyası olduğunu sanmak. `b` ile yapılan her değişiklik `a`'yı da değiştirir. Bağımsız bir kopya için `int[] b = a.clone();` ya da `Arrays.copyOf(a, a.length)` kullanılır.

3.2.7 Hazır dizi işlemleri: java.util.Arrays

Java dizilerinin kendi üzerinde çağrılabilen sıralama, arama ya da yazdırma metotları yoktur. Bu işler `java.util` paketindeki `Arrays` sınıfının statik metotlarıyla yapılır. Statik metot, bir nesne üzerinden değil sınıfın adıyla çağrılır: `a.sort()` değil, `Arrays.sort(a)`. Sınıfı kullanmak için dosyanın başına `import java.util.Arrays;` yazılır. Tablo 3.5 en sık kullanılan metotları ve maliyetlerini verir.

Metot	Ne yapar?	Maliyet
`Arrays.toString(a)`	`"[1, 2, 3]"` biçiminde metin üretir	*O(n)*
`Arrays.equals(a, b)`	Uzunluklar ve öğeler tek tek eşit mi?	*O(n)*
`Arrays.fill(a, v)`	Bütün hücrelere `v` yazar	*O(n)*
`Arrays.copyOf(a, n)`	Yeni dizi; `n > a.length` ise fazlası varsayılan değer	*O(n)*
`Arrays.sort(a)`	Artan sıraya dizer (yerinde)	*O(n log n)*
`Arrays.binarySearch(a, x)`	Sıralı dizide ikili arama; yoksa negatif sonuç	*O(log n)*

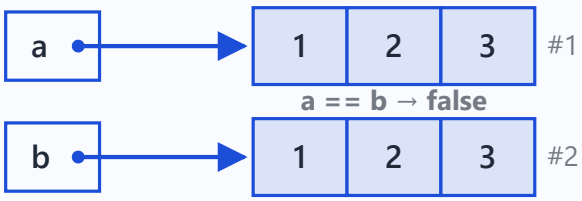
Tablo 3.5: `java.util.Arrays` sınıfının sık kullanılan metotları.

Tablodaki maliyetlerin çoğu *O(n)*'dir, çünkü bu metotlar her öğeye bir kez dokunur. `sort` daha pahalıdır; iyi bir sıralama algoritmasının maliyeti *O(n log n)*'dir (3.5.7). `binarySearch` ise daha ucuzdur, ama yalnız sıralı dizide doğru çalışır. **İkili arama** (*binary search*) aranan değeri dizinin ortasındaki öğeyle karşılaştırır. Değer daha küçükse sol yarıda, büyükse sağ yarıda aramayı sürdürür. Her adımda aralık yarıya indiği için n öğeli dizide yaklaşık $\log_2 n$ adım yeter: bir milyon öğede 20 adım.

Bu sınıfın neden gerektiğini, dizilerle en sık yapılan iki hata gösterir:

```
int[] a = {1, 2, 3};
int[] b = {1, 2, 3};
a == b              // false: ayrı nesne
Arrays.equals(a, b) // true: aynı içerik
println(a)          // [I@1b6d3586
```

`a` ve `b` aynı içerikli iki ayrı dizidir. `==` işleci referansları karşılaştırır, yani "iki değişken aynı nesneyi mi gösteriyor?" sorusunu sorar. İki ayrı nesne olduğu için sonuç `false`'tur. İçeriğin eşitliğini sınamak için `Arrays.equals(a, b)` kullanılır; o, önce uzunlukları, sonra öğeleri tek tek karşılaştırır ve `true` döndürür.



Şekil 3.7: `a` ve `b` içerikleri aynı (1, 2, 3) olan iki ayrı dizi nesnesini (#1 ve #2) gösterir. Bu yüzden `a == b` yanlıştır, `Arrays.equals(a, b)` doğrudur.

İkinci hata, diziyi doğrudan yazdırmaktır. `System.out.println(a)` dizinin öğelerini değil, `[I@1b6d3586` gibi bir metin basar. Bu metindeki `[I` "`int` dizisi" anlamına gelir; `@` işaretinden sonraki kısım ise nesneye özgü bir sayıdır ve her çalıştırmada değişebilir. Öğeleri görmek için `System.out.println(Arrays.toString(a))` yazılır; çıktı `[1, 2, 3]` olur.

> **Sık yapılan hata**
>
> Sırasız dizide `Arrays.binarySearch` kullanmak. Metot hata vermez, ama yanlış sonuç döndürebilir. Örneğin sırasız `{8, 3, 5, 1}` dizisinde 8 aranırsa metot ortadaki öğelere bakıp sağa yönelir ve −5 döndürür; bu, "bulunamadı" demektir. Oysa 8 dizinin ilk öğesidir. İkili aramadan önce dizi `Arrays.sort` ile sıralanmalıdır.

3.2.8 Değişken sayıda argüman

Bazı metotların kaç argüman alacağı önceden belli değildir. Sayıların ortalamasını hesaplayan bir metot iki, üç ya da on sayıyla çağrılabilir. Java bunun için **değişken sayıda argüman** (*varargs*) yazımını sağlar: parametre türünden sonra üç nokta yazılır.

```
static double ortalama(double... dizi) {
    if (dizi.length == 0)
        return Double.NaN;  // tanımsız
    var toplam = 0.0;
    for (var oge : dizi)
        toplam += oge;
    return toplam / dizi.length;
}
```

`double... dizi` parametresi, metodun istediği sayıda `double` argümanla çağrılabileceğini söyler. Derleyici, çağrıda verilen argümanları bir `double[]` dizisine koyar ve metoda o diziyi verir. Metodun içinde `dizi` sıradan bir dizidir: `length`'i vardır, indisle ya da for-each ile gezilebilir. Kuralları şunlardır:

- Değişken sayıda argüman alan parametre, metodun son parametresi olmalıdır.

- Bir metotta bu türden yalnız bir parametre bulunabilir.
- Hazır bir dizi de doğrudan geçirilebilir: `ortalama(new double[] {1, 2})`.

Metodun gövdesinde iki küçük ayrıntı daha vardır. Birincisi, boş çağrı (`ortalama()`) için ayrı bir denetim yapılır ve `Double.NaN` döndürülür. `NaN` (*Not a Number*: sayı değil), tanımsız bir sonucu gösteren özel bir `double` değeridir; hiç sayı yokken ortalama tanımsızdır. İkincisi, `var` anahtar sözcüğüyle tanımlanan yerel değişkenlerin türü, sağdaki değerden çıkarılır: `0.0` bir `double` olduğu için `toplam` da `double`'dır. Tablo 3.6 farklı çağrıların sonucunu gösterir.

Çağrı	`dizi.length`	Sonuç
`ortalama(4, 8, 15)`	3	`9.0`
`ortalama(7.5)`	1	`7.5`
`ortalama()`	0	`NaN`
`ortalama(new double[] {1, 2})`	2	`1.5`

Tablo 3.6: `ortalama` metodunun farklı çağrıları.

Bu yazım aslında yabancı değildir. Her Java programının başındaki `main(String[] args)` metodu, komut satırından verilen argümanları bir dizi olarak alır. `String.format` ve `System.out.printf` gibi kütüphane metotları da değişken sayıda argüman alır.

3.2.9 Kod izleme: ortalama(4, 8, 15)

`ortalama(4, 8, 15)` çağrısını izleyelim. Argümanlar tam sayıdır, ama parametre `double` olduğu için her biri `double`'a genişletilir: dizi [4.0, 8.0, 15.0] olur.

Adım	İşlem	`oge`	`toplam`
çağrı	derleyici `{4.0, 8.0, 15.0}` dizisini oluşturur	—	—
`if`	`dizi.length` = 3 ≠ 0 → devam	—	—
`var toplam`	türü `double` çıkarılır	—	`0.0`
1. tur	`toplam += oge`	`4.0`	`4.0`
2. tur	`toplam += oge`	`8.0`	`12.0`
3. tur	`toplam += oge`	`15.0`	`27.0`
`return`	`27.0 / 3`	—	sonuç `9.0`

Tablo 3.7: `ortalama(4, 8, 15)` çağrısının izlemesi.

Döngü her öğeyi bir kez toplar; n argümanlı bir çağrıda n toplama yapılır ve süre $O(n)$'dir. Metot yalnız `toplam` ve `oge` değişkenlerini kullanır; argüman dizisi dışında ek bellek gerekmez. `ortalama()` çağrılsaydı `length` 0 olur, `if` koşulu doğru çıkar ve 3. satırdaki `return Double.NaN` çalışırdı. Bu denetim olmasaydı sonuç yine `NaN` çıkardı, çünkü `0.0 / 0` kayan noktalı bir bölmedir ve `NaN` verir. Açık denetim, bu durumun bilerek ele alındığını gösterir.



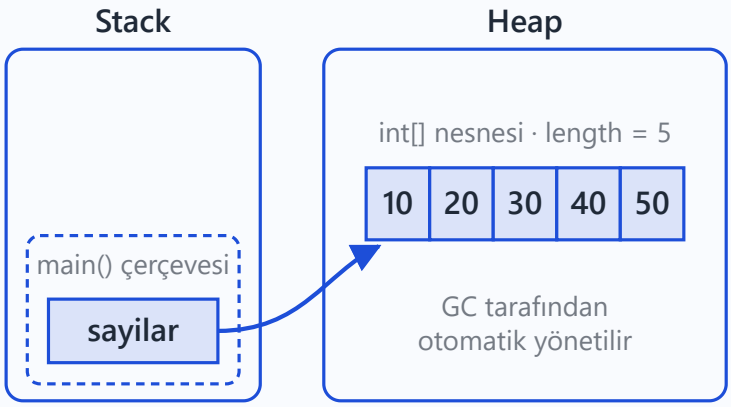
Şekil 3.8: `ortalama(4, 8, 15)` çağrısında derleyicinin oluşturduğu dizi: [4.0, 8.0, 15.0]. Metodun içinde `dizi` bu nesneyi gösterir.

3.2.10 Bellekte diziler: stack ve heap

Önceki örneklerde "stack bellekteki değişken" ve "heap bellekteki dizi" ifadeleri sık geçti. Java programı çalışırken belleği iki ana bölgeye ayırır. Bu bölgeler farklı işler görür ve farklı biçimde yönetilir.

- **Stack bellek** (yığın belleği): Metot çağrıları ve yerel değişkenler burada durur. Her metot çağrısı için stack bellekte bir **yığın çerçevesi** (*stack frame*) açılır. Çerçeve, metodun parametrelerini ve yerel değişkenlerini tutar. Metot bitince çerçevesi kendiliğinden silinir.
- **Heap bellek** (*heap memory*): `new` ile oluşturulan bütün nesneler burada durur. Java'da her dizi heap bellektedir; stack bellekte yalnız dizinin referansı bulunur.

Heap belleği 3.2.3'te adı geçen çöp toplayıcı yönetir. Çöp toplayıcı ara ara çalışır, hiçbir referansın göstermediği nesneleri bulur ve belleklerini serbest bırakır. Programcının bir nesneyi elle silmesi gerekmez, zaten Java'da buna izin veren bir komut da yoktur.



Şekil 3.9: Stack bellekteki `main()` çerçevesinde `sayilar` referansı durur. Referans, heap bellekteki 5 öğeli `int[]` nesnesini (10, 20, 30, 40, 50; `length` = 5) gösterir. Heap belleği çöp toplayıcı yönetir.

	Stack	Heap
Ne tutulur?	yerel değişkenler, referanslar, çağrı çerçeveleri	nesneler, diziler
Ömür	metot bitene kadar	erişilebilir olduğu sürece
Yönetim	otomatik (çağrı / dönüş)	otomatik (çöp toplayıcı)

Tablo 3.8: Stack ve heap belleğin karşılaştırması.

Bu iki bölgenin ömürleri farklıdır ve bu fark önemli bir sonuç doğurur. Bir metodun içinde oluşturulan dizi, metot bitince yok olmaz. Metodun çerçevesi ve dizinin referansını tutan yerel değişken silinir, ama dizi heap bellekte durmaya devam eder. Metot dizinin referansını döndürmüşse, çağıran yer o referansla diziyi kullanmayı sürdürür. 3.3.1'deki `diziOku` metodu tam olarak böyle çalışır.

Dizinin ömrü, ona en az bir referans olduğu sürece devam eder. Son referans da kaybolduğunda (değişkene başka bir dizi atandığında, değişken `null` yapıldığında ya da değişkenin metodu bittiğinde) dizi erişilemez olur ve çöp toplayıcı onu toplayabilir.

3.2.11 "Statik" ve "dinamik" diziler: C ve Java

Dizilerle ilgili kaynaklarda **statik dizi** (*static array*) ve **dinamik dizi** (*dynamic array*) terimleri sık geçer. Bu terimler en çok C ve C++ için anlamlıdır, çünkü bu dillerde dizi iki ayrı yerde oluşturulabilir. Tablo 3.9 iki dili karşılaştırır.

	C / C++	Java
Stack bellekte dizi	`int a[10];` — boyut derleme zamanında sabit, fonksiyon bitince yok olur	Yok: bütün diziler heap bellekte
Heap bellekte dizi	`malloc` / `new int[n]`	`new int[n]` (ilklendirici listesi de heap bellekte)
Boyut	heap bellekte çalışma zamanında belirlenir	çalışma zamanında belirlenir, sonra sabit
Serbest bırakma	programcı: `free` / `delete[]`	çöp toplayıcı otomatik; `dizi = null` yalnız referansı koparır

Tablo 3.9: C/C++ ve Java'da dizilerin yeri ve ömrü.

C'de `int a[10];` yazılınca dizi stack bellekte, fonksiyonun çerçevesinin içinde oluşur. Boyutu derleme sırasında bilinir ve fonksiyon bitince dizi de çerçeveyle birlikte yok olur. Bu tür dizilere "statik" denir. Boyut ancak çalışma sırasında belli oluyorsa dizi heap bellekte `malloc` (C) ya da `new` (C++) ile ayrılır ve buna "dinamik" denir. C'de heap bellekteki diziyi işi bitince `free` ile serbest bırakmak programcının görevidir. Unutulursa bellek boşa harcanır; aynı bellek iki kez serbest bırakılırsa programın davranışı tanımsız olur ve çoğu zaman program çöker.

Java'da bu ayrım yoktur. Bütün diziler heap bellekte oluşur. Boyut çalışma sırasında hesaplanabilir, ama dizi oluştuktan sonra değişmez. Aşağıdaki kod Java'daki tipik durumu gösterir:

```
int[] dinamikDizi = new int[n];   // heap'te n öğelik yer ayrılır
dinamikDizi = null;               // başka referans yoksa GC toplayabilir
```

İlk satır, `n`'nin o anki değeri kadar öğelik bir diziyi heap bellekte oluşturur. İkinci satır diziyi silmez; yalnız `dinamikDizi` değişkeninin referansını koparır. Diziyi gösteren başka bir referans yoksa çöp toplayıcı onu uygun bir anda toplar. `System.gc()` çağrısı da toplamayı garanti etmez; yalnız JVM'e bir istekte bulunur.

"Dinamik dizi" adının ikinci bir anlamı daha vardır. Dolunca kendiliğinden büyüyen diziler de bu adla anılır. Java'daki `ArrayList` böyle bir yapıdır: içinde sıradan bir dizi tutar ve dizi dolunca daha büyük bir dizi oluşturup öğeleri ona kopyalar. Bu anlamdaki dinamik dizi, sabit boyutlu dizinin üzerine kurulmuş bir yapıdır; dizinin kendisinin boyutu yine değişmez.

> **Sık yapılan hata**
>
> `dizi = null;` yazmanın diziyi hemen sildiğini sanmak. Bu atama yalnız bir referansı koparır. Aynı diziyi gösteren başka bir değişken varsa dizi yaşamaya devam eder; yoksa bile ne zaman toplanacağına çöp toplayıcı karar verir.

3.2.12 İndis sınır dışı hatası

n öğeli bir dizide geçerli indisler 0'dan *n* − *1*'e kadardır. Bu aralığın dışındaki bir indise erişilirse, yani indis negatifse ya da `length`'e eşit veya ondan büyükse, Java `ArrayIndexOutOfBoundsException` fırlatır. Bu istisna, daha genel `IndexOutOfBoundsException`'ın alt sınıfıdır.

```
int[] dizi = new int[5];
dizi[5] = 1;  // Index 5 out of
              // bounds for length 5
dizi[-1] = 1; // aynı hata
```

Beş öğeli dizide 5 indisli hücre yoktur. `dizi[5] = 1` satırı derlenir, çünkü derleyici indisin değerini genellikle bilemez. Ama çalışma sırasında JVM her erişimden önce indisi denetler ve istisna fırlatır. İstisnanın iletisi sorunu açıkça söyler: `Index 5 out of bounds for length 5` (5 indisi, 5 uzunluğun sınırları dışında). `dizi[-1]` de aynı hatayı verir. İstisna yakalanmazsa program o noktada sonlanır.



Şekil 3.10: 5 öğeli dizide geçerli indisler 0 … `length` − 1, yani 0–4'tür. Kesikli çizilen 5 indisli hücre dizinin dışındadır; `dizi[5]` erişimi hata verir.

Bu denetim, Java'yı C'den ayıran önemli bir güvenlik özelliğidir. C dizi erişimlerini denetlemez. `a[5]` yazıldığında adres formülü yine hesaplanır ve dizinin hemen arkasındaki bellek okunur ya da üzerine yazılır. Orada başka bir değişken varsa onun değeri sessizce bozulur. Böyle bir hata program başka bir

yerde tuhaf davranana kadar fark edilmeyebilir. Bu tür taşmalar tarih boyunca pek çok güvenlik açığının da nedeni olmuştur. Java'da ise hata, yapıldığı satırda ve açık bir iletiyle ortaya çıkar. Denetimin bedeli her erişimde yapılan küçük bir karşılaştırmadır.

Kullanıcıdan gelen ya da hesaplanan bir indisle erişim yapılacaksa indis önceden denetlenebilir:

```
if (indis >= 0
        && indis < dizi.length) {
    int deger = dizi[indis];
} else {
    System.out.println("Geçersiz");
}
```

Koşul iki parçalıdır: indis 0'dan küçük olmamalı ve `length`'ten küçük olmalıdır. İkisi `&&` ile bağlanır; ikisi de doğruysa erişim güvenlidir.

Sık yapılan hata

Döngü koşulunda `i <= a.length` yazmak. Bu, bir eksik ya da bir fazla saymaktan doğan **sınır hatasıdır** (*off-by-one error*). Döngü son turda `a[a.length]`'e erişir ve istisna fırlatır. Diziyi baştan sona gezen döngünün doğru biçimi `for (int i = 0; i < a.length; i++)`'dır.

3.3 Dizi algoritmaları

Bu kesimdeki algoritmalar küçük ama öğreticidir. Her biri diziyi farklı bir biçimde kullanır: kimi baştan sona bir kez gezer, kimi önce sıralar, kimi iki uçtan ortaya doğru ilerler. Algoritmaları anlamanın en güvenilir yolu, küçük bir girdi üzerinde her adımı kâğıda yazarak izlemektir. Aşağıdaki izleme tabloları bu yöntemin örnekleridir.

3.3.1 Diziyi kullanıcıdan okuma

Ortalama, ortanca ve mod programlarının üçü de aynı biçimde başlar: kullanıcıdan önce kaç sayı gireceğini, ardından sayıları tek tek ister. Bu ortak işi her programda yeniden yazmak yerine bir yardımcı metotta toplamak daha doğrudur:

```
import java.util.Scanner;

static double[] diziOku(Scanner scanner) {
    System.out.print("Dizi boyutunu girin: ");
    int boyut = scanner.nextInt();
    double[] dizi = new double[boyut];  // boyut çalışma zamanında
    for (int i = 0; i < boyut; i++) {
        System.out.print((i + 1) + ". öğeyi girin: "); // 1'den say
        dizi[i] = scanner.nextDouble();
    }
    return dizi;
}
```

Metot, klavyeden okuma yapan bir `Scanner` nesnesini parametre olarak alır. `Scanner`'ı metodun içinde değil dışarıda oluşturup vermenin nedeni, programın tek bir `Scanner` kullanmasıdır. Aynı girdi kaynağına bağlı birden çok `Scanner` açmak, okunan verilerin karışmasına yol açabilir.

Metodun adımları şunlardır:

1. Kullanıcıya boyut sorulur. `print` satır sonu eklemediği için imleç sorunun hemen yanında bekler.
2. `nextInt()` kullanıcının yazdığı tam sayıyı okur ve `boyut`'a atar.
3. `new double[boyut]` diziyi oluşturur. Boyut ancak bu satırda, program çalışırken belli olmuştur; Java'da bu bir sorun değildir. Hücrelerin hepsi `0.0` ile başlar.
4. Döngü her öğeyi sırayla okur. Kullanıcıya gösterilen sıra numarası `i + 1`'dir: dizi 0'dan sayar, ama insanlar 1'den sayar. “0. öğeyi girin” demek kullanıcıyı şaşırtır.
5. Dizinin referansı döndürülür.

Son adım, 3.2.10'daki bellek düzeninin pratikteki sonucudur. `dizi` değişkeni metodun yerel değişkenidir ve metot bitince silinir. Dizi ise heap bellektedir ve silinmez. Döndürülen referans çağıran yerdeki bir değişkene atanır ve dizi orada kullanılmaya devam eder.

Bu kesimde işlenecek üç istatistik Tablo 3.10'da özetlenmiştir. Hepsi aynı diziyi kullanır, ama maliyetleri farklıdır.

Ölçü	Tanım	Maliyet
Ortalama	Toplam / öğe sayısı	*O(n)*
Ortanca	Sıralı dizinin ortası	*O(n log n)*
Mod	En sık tekrar eden değer	*O(n log n)*

Tablo 3.10: Dizi üzerinde hesaplanan üç istatistik.

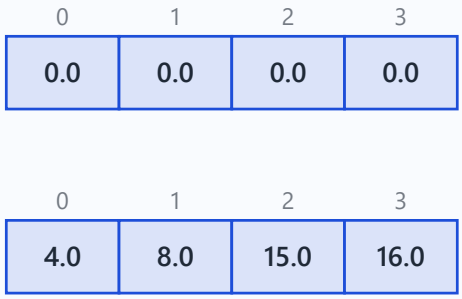
3.3.2 Kod izleme: diziOku(sc)

Kullanıcının boyut olarak 4, ardından sırayla 4, 8, 15 ve 16 girdiğini varsayalım. Tablo 3.11 metodun her adımını gösterir.

Adım	Ekranda	Okunan	`dizi`
boyut sorulur	Dizi boyutunu girin:	`boyut` = 4	henüz yok
`new double[4]`	—	—	[0.0, 0.0, 0.0, 0.0]
`i = 0`	1. öğeyi girin:	4	[4.0, 0.0, 0.0, 0.0]
`i = 1`	2. öğeyi girin:	8	[4.0, 8.0, 0.0, 0.0]
`i = 2`	3. öğeyi girin:	15	[4.0, 8.0, 15.0, 0.0]
`i = 3`	4. öğeyi girin:	16	[4.0, 8.0, 15.0, 16.0]
`i = 4`	4 < 4 yanlış → döngü biter	—	[4.0, 8.0, 15.0, 16.0]
`return`	—	—	referans döner

Tablo 3.11: `diziOku(sc)` izlemesi. Kullanıcı 4, 4, 8, 15, 16 giriyor.

İzlemede iki ayrıntıya dikkat edelim. Boyut okunmadan önce dizi yoktur; dizi ancak kullanıcı 4 yazdıktan sonra oluşur (Şekil 3.11). Ayrıca `nextDouble()` kullanıcının yazdığı 4'ü `4.0` olarak okur, çünkü dizi `double` türündedir.



Şekil 3.11: `diziOku` sırasında dizi. Solda `new double[4]` 'ten hemen sonra dört hücre `0.0` 'dır. Sağda döngü bittiğinde dizi 4.0, 8.0, 15.0, 16.0 değerlerini taşır.

Metot *n* öğe için *n* kez okuma yapar; süresi *O(n)*'dir. Gerçekte bu sürenin çoğu kullanıcının yazmasını beklemekle geçer, ama algoritmanın yaptığı iş öğe sayısıyla doğru orantılıdır.

Sık yapılan hata

Kullanıcıya gösterilen sıra numarasıyla dizinin indisini aynı yapmaya çalışmak. Döngüyü `i = 1` 'den başlatıp `i <= boyut` 'a kadar götürmek kullanıcıya doğru numaraları gösterir. Ama bu döngü `dizi[0]` 'ı boş bırakır ve son turda `dizi[boyut]` 'a, yani sınır dışına erişir. Doğrusu indisi 0'dan saymak, yalnız ekranda `i + 1` göstermektir.

3.3.3 Ortalama hesaplama

Ortalama, bütün değerlerin toplamının değer sayısına bölümüdür. Aşağıdaki `main` metodu diziyi `diziOku` ile okur, ortalamayı hesaplar ve sonucu yazdırır:

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    double[] dizi = diziOku(sc);
    double toplam = 0;
    for (int i = 0; i < dizi.length; i++)
        toplam += dizi[i];
    double ortalama = toplam / dizi.length;
    System.out.println("Dizi: "
        + Arrays.toString(dizi));
    System.out.println("Ortalama: "
        + ortalama);
    sc.close();
}
```

Hesabın merkezinde bir **toplayıcı değişken** vardır. `toplam` 0 ile başlar ve döngünün her turunda bir öğe eklenir. Döngü bitince `toplam` bütün öğelerin toplamını tutar. Bu kalıp, dizi üzerinde biriktirme yapan bütün algoritmalarda görülür: en büyüğü bulmak, belirli bir koşula uyanları saymak, çarpım almak gibi.

`toplam`'ın türü `double`'dır ve başlangıç değeri olarak `0` yazılmıştır. Java bu tam sayıyı kendiliğinden `0.0`'a genişletir. Bölme de `double` ile yapıldığı için sonuç ondalıklı çıkar.

Sonuç yazdırılırken dizi `Arrays.toString(dizi)` ile metne çevrilir. `println(dizi)` yazılsaydı 3.2.7'de görüldüğü gibi `[D@…` biçiminde bir tür ve sayı basılırdı. Programın sonundaki `sc.close()` çağrısı `Scanner`'ı kapatır.

Örnek 3.3

Kullanıcı bu kez 5 değer girsin: `diziOku` {4, 8, 15, 16, 23} dizisini döndürür.

1. `toplam` = 0.0.
2. `i = 0` : 0.0 + 4.0 = 4.0.
3. `i = 1` : 4.0 + 8.0 = 12.0.
4. `i = 2` : 12.0 + 15.0 = 27.0.
5. `i = 3` : 27.0 + 16.0 = 43.0.
6. `i = 4` : 43.0 + 23.0 = 66.0.
7. `i = 5` : 5 < 5 yanlış, döngü biter.
8. `ortalama` = 66.0 / 5 = 13.2.

Program `Dizi: [4.0, 8.0, 15.0, 16.0, 23.0]` ve `Ortalama: 13.2` satırlarını yazar.



Şekil 3.12: Örnek 3.3'teki dizi: 4.0, 8.0, 15.0, 16.0, 23.0. Toplam 66,0, ortalama 66,0 / 5 = 13,2'dir.

Her öğe bir kez toplanır; *n* öğe için *n* toplama yapılır. Süre *O(n)*'dir. Algoritma dizinin dışında yalnız `toplam`, `i` ve `ortalama` değişkenlerini kullanır. Bu değişkenlerin sayısı *n* büyüse de değişmez; bu yüzden ek bellek *O(1)*'dir.

Sınır durumu da düşünülmelidir. Kullanıcı boyut olarak 0 girerse `new double[0]` geçerli ama boş bir dizi oluşturur. Döngü hiç dönmez, `toplam` 0.0 kalır ve `0.0 / 0` hesaplanır. Bu bir kayan noktalı bölmedir ve hata vermez; sonuç `NaN` olur. Program "Ortalama: NaN" yazar. Bu teknik olarak doğrudur, ama kullanıcıya yardımcı olmaz. Boş dizi için "hiç sayı girilmedi" gibi bir uyarı vermek daha iyidir.

Sık yapılan hata

Toplamı ve öğe sayısını tam sayı tutup bölmek. `int` ile `int` bölünürse sonuç tam sayıya kırpılır: `66 / 5` 13 verir, 13,2 değil. Boş dizide de `0 / 0` tam sayı bölmesidir ve `ArithmeticException` fırlatır. En az bir işlenen `double` olmalıdır.

3.3.4 Ortanca

Ortalama, uç değerlere karşı çok duyarlıdır. Beş kişinin aylık geliri 3, 5, 7, 9 ve 1000 birim olsun. Ortalama (3 + 5 + 7 + 9 + 1000) / 5 = 204,8'dir; bu sayı beş kişiden hiçbirinin durumunu yansıtmaz. Tek bir uç değer ortalamayı yukarı çekmiştir. Böyle durumlarda "ortadaki kişinin" geliri daha anlamlı bir ölçüdür.

Tanım 3.3

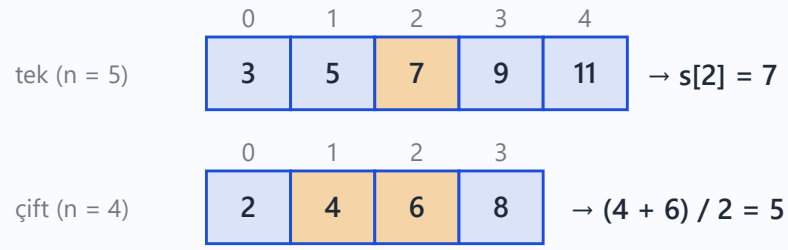
Bir sayı dizisinin **ortancası** (*median*), dizi sıralandığında ortaya düşen değerdir. Öğe sayısı *n* tekse ortanca, sıralı dizinin *n / 2* indisli öğesidir (tam sayı bölmesiyle). *n* çiftse ortada iki öğe vardır, *n/2 − 1* ve *n/2* indisli öğeler; ortanca bu ikisinin ortalamasıdır.

Yukarıdaki gelir örneğinde sıralı dizi 3, 5, 7, 9, 1000'dir ve ortanca 7'dir. 1000 yerine 10 ya da bir milyon yazılsa da ortanca değişmez. Ortanca, uç değerlerden etkilenmez.

```java
static double ortanca(double[] dizi) {
    double[] s = dizi.clone(); // kopya
    Arrays.sort(s);            // O(n log n)
    int boyut = s.length;
    double ortanca;
    if (boyut % 2 == 0) {      // çift
        int o1 = boyut/2 - 1, o2 = boyut/2;
        ortanca = (s[o1] + s[o2]) / 2;
    } else {                   // tek
        ortanca = s[boyut / 2];
    }
    return ortanca;
}
```

Metot dört adımda çalışır:

1. **Kopyalama.** `dizi.clone()` ile dizinin bir kopyası alınır. Bu adım gereklidir, çünkü sonraki adım diziyi sıralar. Sıralama doğrudan `dizi` üzerinde yapılsaydı, çağıranın elindeki dizinin sırası da bozulurdu (3.2.6). Ortancayı soran birinin, verisinin yeniden düzenlenmesini beklemesi için bir neden yoktur.
2. **Sıralama.** `Arrays.sort(s)` kopyayı artan sıraya dizer. Bu adımın maliyeti *O(n log n)*'dir.
3. **Tek mi, çift mi?** `boyut % 2`, boyutun 2'ye bölümünden kalandır. Kalan 0 ise boyut çifttir.
4. **Ortadaki değer.** Çift boyutta `o1 = boyut/2 - 1` ve `o2 = boyut/2` indislerindeki iki değerin ortalaması alınır. Tek boyutta `s[boyut / 2]` döndürülür; tam sayı bölmesi tam ortadaki indisi verir.



Şekil 3.13: Ortancanın iki durumu. Üstte tek boyut ($n = 5$): sıralı 3, 5, 7, 9, 11 dizisinde tam ortadaki `s[2]` = 7. Altta çift boyut ($n = 4$): sıralı 2, 4, 6, 8 dizisinde ortadaki iki değerin ortalaması (4 + 6) / 2 = 5.

3.3.5 Kod izleme: ortanca()

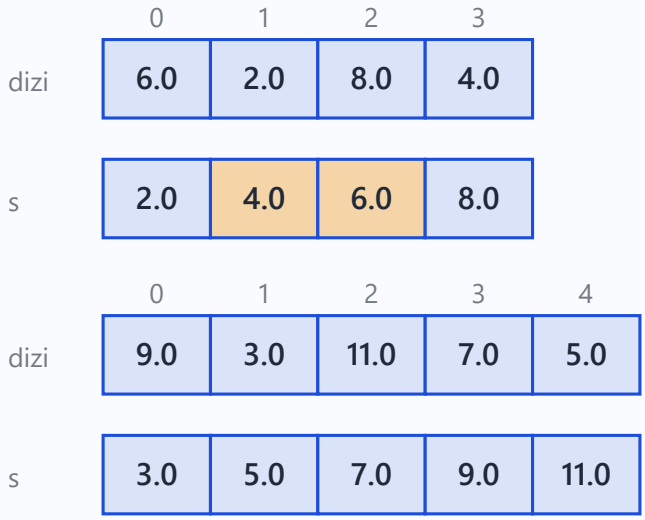
Metodu biri çift, biri tek boyutlu iki diziyle çağıralım: önce {6, 2, 8, 4}, sonra {9, 3, 11, 7, 5}.

Adım	1. çağrı: {6, 2, 8, 4}	2. çağrı: {9, 3, 11, 7, 5}
`s = dizi.clone()`	[6.0, 2.0, 8.0, 4.0]	[9.0, 3.0, 11.0, 7.0, 5.0]
`Arrays.sort(s)`	[2.0, 4.0, 6.0, 8.0]	[3.0, 5.0, 7.0, 9.0, 11.0]
`boyut`	4	5

Adım	1. çağrı: {6, 2, 8, 4}	2. çağrı: {9, 3, 11, 7, 5}
`boyut % 2`	0 → çift	1 → tek
ortadaki indis(ler)	`o1` = 1, `o2` = 2	5 / 2 = 2
hesap	(4.0 + 6.0) / 2	`s[2]`
`return`	`5.0`	`7.0`
çağıranın dizisi	değişmedi: [6.0, 2.0, 8.0, 4.0]	değişmedi: [9.0, 3.0, 11.0, 7.0, 5.0]

Tablo 3.12: `ortanca` metodunun iki çağrısının izlemesi.

Tablonun son satırı, kopyalama adımının işe yaradığını gösterir: iki çağrıdan sonra da çağıranın dizileri ilk sıralarını korur. Sıralanan, yalnız metodun içindeki `s` kopyasıdır.



Şekil 3.14: İki çağrının son durumu. Solda 1. çağrı: üstte çağıranın dizisi (6, 2, 8, 4), altta sıralı kopya `s`; ortadaki 4.0 ve 6.0 vurgulu. Sağda 2. çağrı: dizi 9, 3, 11, 7, 5 ve sıralı kopya 3, 5, 7, 9, 11.

Metodun maliyetini sıralama belirler. Kopyalama *O(n)*, sıralama *O(n log n)*, ortadaki değeri okumak *O(1)*'dir. Toplam maliyet en büyük terimle belirlenir: *O(n log n)*. Kopya için *n* öğelik ek bellek gerekir; ek bellek *O(n)*'dir. Ortancayı sıralamadan, ortalama *O(n)* sürede bulan daha ileri algoritmalar da vardır, ama sıralama yolu en kolay anlaşılanıdır.

Sık yapılan hata

Ortancayı sıralanmamış dizinin ortasındaki öğe sanmak. {6, 2, 8, 4} dizisinin ortasında 2 ve 8 vardır; ortalamaları 5'tir ve bu örnekte tesadüfen doğru çıkar. {9, 3, 11, 7, 5} dizisinin ortasındaki öğe ise 11'dir, oysa ortanca 7'dir. Ortanca, yalnız sıralı dizide ortadaki değerdir.

3.3.6 Mod

Tanım 3.4

Bir dizinin **modu** (*mode*), dizide en sık tekrar eden değerdir. Birden çok değer aynı en yüksek sayıda tekrar ediyorsa dizinin birden çok modu vardır; bir algoritma bunlardan birini seçmek zorundadır.

Mod bulmanın en düz yolu, her değer için dizide kaç kez geçtiğini saymaktır. Ama her öğe için bütün diziyi yeniden gezmek $n \times n$ karşılaştırma demektir. Daha iyi bir yol, diziyi önce sıralamaktır. Sıralı dizide eşit değerler yan yana gelir. Böylece her değerin tekrar sayısı, diziyi tek bir kez soldan sağa gezerek bulunur.

```
static int mod(int[] dizi) { // boş değil
    int[] s = dizi.clone();
    Arrays.sort(s);  // eşitler yan yana
    int enCok = s[0], enCokSayi = 1;
    int mevcutSayi = 1;
    for (int i = 1; i < s.length; i++) {
        if (s[i] == s[i - 1]) mevcutSayi++;
        else mevcutSayi = 1;   // yeni değer
        if (mevcutSayi > enCokSayi) {
            enCokSayi = mevcutSayi;
            enCok = s[i];
        }
    }
    return enCok;
}
```

Metodun değişkenleri şunlardır:

- `s` : dizinin sıralı kopyası. Kopyalamanın nedeni ortanca metodundakiyle aynıdır.
- `mevcutSayi` : o anda okunan değerin, şimdiye kadar art arda kaç kez görüldüğü.
- `enCok` ve `enCokSayi` : şimdiye kadarki rekor; en uzun tekrar dizisinin değeri ve uzunluğu.

Başlangıçta ilk değer (`s[0]`) bir kez görülmüştür ve rekor odur. Döngü 1 indisinden başlar ve her öğeyi solundakiyle karşılaştırır. Değer solundakiyle aynıysa aynı tekrar dizisi sürüyor demektir ve `mevcutSayi` bir artar. Farklıysa yeni bir değer başlamıştır ve sayaç 1'e döner. Her turun sonunda sayaç rekorla karşılaştırılır. Sayaç rekoru geçmişse rekor güncellenir.



Şekil 3.15: Sıralı dizi 1, 1, 2, 4, 4, 4. Eşit değerler yan yana durur: 1 iki kez, 2 bir kez, 4 üç kez geçer. Mod 4'tür.

Karşılaştırmanın `>` ile yapılması bir seçimdir. İki değer aynı sayıda tekrar ederse sonra gelen değer rekoru *geçemez*, yalnız *eşitler*. Bu yüzden rekor değişmez ve önce görülen değer kalır. Dizi artan sırada olduğu için önce görülen değer, eşitler arasında en küçük olandır. Koşul `>=` olsaydı eşitlikte sonraki, yani en büyük değer seçilirdi.

Metodun başındaki yorum ("boş değil") bir ön koşuldur. Boş dizide `s[0]` olmadığı için metot ilk satırlarda `ArrayIndexOutOfBoundsException` fırlatır. Metodu çağıran, diziyi boş vermemekle yükümlüdür.

3.3.7 Kod izleme: mod()

`mod({4, 1, 4, 2, 1, 4})` çağrısını izleyelim. Girdide eşit değerler dağınıktır. Sıralanınca kopya `s` = {1, 1, 2, 4, 4, 4} olur. Başlangıçta `enCok` = 1, `enCokSayi` = 1, `mevcutSayi` = 1'dir.

`i`	`s[i]` ile `s[i-1]`	`mevcutSayi`	Rekor geçildi mi?	`enCok` / `enCokSayi`
1	1 == 1 → aynı	2	2 > 1 evet	1 / 2
2	2 ≠ 1 → yeni değer	1	1 > 2 hayır	1 / 2
3	4 ≠ 2 → yeni değer	1	1 > 2 hayır	1 / 2
4	4 == 4 → aynı	2	2 > 2 hayır	1 / 2
5	4 == 4 → aynı	3	3 > 2 evet	4 / 3

Tablo 3.13: `mod({4, 1, 4, 2, 1, 4})` izlemesi; `s` = {1, 1, 2, 4, 4, 4}.

Metot 4 döndürür: 4 değeri üç kez geçer. `i = 4` satırı eşitlik kuralını gösterir. O anda 4 iki kez görülmüştür ve rekor da 2'dir. `2 > 2` yanlış olduğu için rekor 1'de kalır. Dizi o noktada bitseydi 1 ile 4 ikişer kez geçmiş olurdu ve metot önce görülen 1'i döndürürdü. Bu dizide 4 bir kez daha geçtiği için `i = 5`'te rekor 4'e geçer.

Maliyet iki parçalıdır. Sıralama *O(n log n)*, ardından gelen tarama *O(n)*'dir; toplam *O(n log n)*'dir. Kopya için *O(n)* ek bellek kullanılır.

Sık yapılan hata

Sıralama satırını gereksiz sanıp silmek. Algoritma yalnız *komşu* öğeleri karşılaştırır. Sırasız {4, 1, 4, 2, 1, 4} dizisinde hiçbir komşu çift eşit değildir; `mevcutSayi` hep 1 kalır, rekor hiç değişmez ve metot ilk öğeyi, 4'ü döndürür. Bu örnekte sonuç tesadüfen doğrudur, ama ilk öğe mod olmasaydı yanlış çıkardı. Sıralama, eşit değerleri yan yana getirerek algoritmanın varsayımını sağlar.

3.3.8 Diziyi yerinde tersine çevirme

Bir diziyi tersine çevirmek, ilk öğeyi sona, son öğeyi başa, ikinci öğeyi sondan ikinciye götürmek demektir. {3, 8, 1, 9, 4, 6} dizisi {6, 4, 9, 1, 8, 3} olur. Bunun en basit yolu, aynı boyutta yeni bir dizi açıp öğeleri sondan başa kopyalamaktır: `yeni[i] = dizi[n - 1 - i]`. Bu yöntem doğrudur, ama *n* öğelik ikinci bir dizi ister. Aynı iş ek dizi kullanmadan, dizinin kendi içinde de yapılabilir.

Ek dizi kullanmadan, dizinin kendi üzerinde çalışan algoritmalara **yerinde** (*in-place*) algoritma denir. Yerinde tersine çevirmenin fikri basittir. Önce ilk ve son öğenin yeri değiştirilir, sonra ikinci ve sondan ikinci öğeninki. Bu, ortaya varılana kadar sürer. İki öğenin yerini değiştirmeye **takas** (*swap*) denir.

```
static void tersineCevir(int[] dizi) {
    int baslangic = 0;
    int bitis = dizi.length - 1;
    while (baslangic < bitis) {
        // uçtaki iki öğeyi takas et
        int gecici = dizi[baslangic];
        dizi[baslangic] = dizi[bitis];
        dizi[bitis] = gecici;
        baslangic++;   // içe doğru ilerle
        bitis--;
    }
}
```

Metot iki indis tutar. `baslangic` soldan, `bitis` sağdan başlar. Bu indislere, dizinin içinde bir konumu "gösterdikleri" için **işaretçi** de denir. Döngünün her turunda iki işaretçinin gösterdiği öğeler takas edilir, sonra soldaki bir sağa, sağdaki bir sola ilerler. İşaretçiler ortada karşılaşınca, yani `baslangic < bitis` yanlış olunca döngü biter.

Takas üç atamayla yapılır ve geçici bir değişken gerektirir. İlk atama soldaki değeri `gecici`'de saklar. İkinci atama sağdaki değeri sola yazar; bu anda soldaki eski değer dizide artık yoktur ve yalnız `gecici`'de durur. Üçüncü atama `gecici`'deki değeri sağa yazar. `gecici` olmadan, `dizi[baslangic] = dizi[bitis]; dizi[bitis] = dizi[baslangic];` yazılsaydı, ilk atamadan sonra iki hücre de aynı değeri taşırdı ve soldaki değer kaybolurdu.

Tur	`baslangic`, `bitis`	Koşul	`gecici`	Takastan sonra `dizi`
1	0, 5	0 < 5	3	[6, 8, 1, 9, 4, 3]
2	1, 4	1 < 4	8	[6, 4, 1, 9, 8, 3]
3	2, 3	2 < 3	1	[6, 4, 9, 1, 8, 3]
—	3, 2	3 < 2 yanlış → biter	—	[6, 4, 9, 1, 8, 3]

Tablo 3.14: `tersineCevir({3, 8, 1, 9, 4, 6})` izlemesi.



Şekil 3.16: Tersine çevirmenin üç anı. (a) Başlangıç: `bas` 0 indisinde, dizi 3, 8, 1, 9, 4, 6. Şekildeki `bas` ve `bit`, `baslangic` ile `bitis`'in kısaltmalarıdır. (b) İlk takastan ve işaretçiler içe ilerledikten sonra: dizi 6, 8, 1, 9, 4, 3; `bas` = 1, `bit` = 4; `gecici`'de son saklanan değer 3. (c) Son: işaretçiler ortada birbirini geçmiştir (`bit` = 2, `bas` = 3); dizi 6, 4, 9, 1, 8, 3.

Altı öğeli dizide üç takas yapıldı. Öğe sayısı tek olsaydı ortadaki öğe kendi yerinde kalırdı: {1, 2, 3, 4, 5} dizisinde 1 ile 5, 2 ile 4 takas edilir; 3'e dokunulmaz ve sonuç {5, 4, 3, 2, 1} olur.

Teorem 3.1 (Tersine çevirmenin maliyeti)

n öğeli bir diziyi `tersineCevir` metodu *⌊n/2⌋* takasla tersine çevirir. Süre *O(n)*, ek bellek *O(1)*'dir.

İspat. Her takas bir soldaki ve bir sağdaki öğeyi yerine koyar ve işaretçiler arasındaki uzaklığı 2 azaltır. Başlangıçta `bitis - baslangic = n - 1`'dir. Döngü, bu fark pozitif olduğu sürece döner. *n* çiftse *n/2* turdan sonra fark −1 olur; *n* tekse *(n − 1)/2* turdan sonra fark 0 olur ve ortadaki öğe kendi yerinde kalır. İki durumda da tur sayısı *⌊n/2⌋*'dir. Her tur sabit sayıda (üç atama, iki artırma/azaltma, bir karşılaştırma) iş yapar; toplam süre *O(n)*'dir. Metot yalnız `baslangic`, `bitis` ve `gecici` değişkenlerini kullanır; ek bellek *O(1)*'dir. □

Yeni diziye kopyalama yöntemi de *O(n)* sürer: *n* kopyalama yapar. İki yöntemin farkı bellektedir. Kopyalama yöntemi *O(n)*, yerinde yöntem *O(1)* ek bellek ister. Bellek kısıtlıysa ya da dizi çok büyükse yerinde yöntem tercih edilir. Buna karşılık kopyalama yöntemi orijinal diziyi bozmaz; orijinal dizi başka bir yerde de kullanılıyorsa bu bir üstünlüktür.

> **Sık yapılan hata**
>
> Döngüyü dizinin sonuna kadar sürdürmek: `for (int i = 0; i < n; i++)` ile `dizi[i]` ile `dizi[n-1-i]`'yi takas etmek. İlk yarıda dizi tersine döner, ikinci yarıda ise aynı çiftler yeniden takas edilir ve dizi ilk hâline geri döner. Döngü ortada, *n/2* turda durmalıdır.

3.3.9 Uygulama: Sezar şifresi

Sezar şifresi, adını Roma devlet adamı Julius Caesar'dan alan çok eski bir şifreleme yöntemidir. Metindeki her harf, alfabede sabit bir *k* adım ilerideki harfle değiştirilir. *k* = 3 ise A → D, B → E, C → F olur. Alfabenin sonuna gelindiğinde başa dönülür: X → A, Y → B, Z → C. Yöntem bugün hiçbir güvenlik sağlamaz, çünkü yalnız 25 farklı anahtar vardır ve hepsi kolayca denenebilir. Ama karakter dizileriyle ve kalan işlemiyle çalışmayı göstermek için iyi bir örnektir.

```
static String sifrele(String mesaj, int k) {
    char[] harf = mesaj.toCharArray();
    for (int i = 0; i < harf.length; i++) {
        int sira = harf[i] - 'A';        // 0..25 (yalnız A-Z)
        harf[i] = (char) ('A' + (sira + k) % 26);
    }
    return new String(harf);
}
```

Metodun her satırı bir Java ayrıntısına dayanır:

- `toCharArray()`. Java'da `String` nesneleri değiştirilemez; bir metnin 2. harfini başka bir harfle değiştirmenin yolu yoktur. Bu yüzden metnin harfleri önce yeni bir `char[]` dizisine kopyalanır. Dizinin öğeleri ise yerinde değiştirilebilir.
- `harf[i] - 'A'`. Java'da `char` değerleri aslında tam sayıdır; her karakterin bir kodu vardır ve büyük harflerin kodları alfabe sırasıyla ardışıktır. Bir harften `'A'`'yı çıkarmak, harfin alfabedeki sırasını verir: `'A'` için 0, `'E'` için 4, `'Z'` için 25.

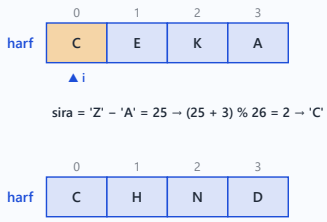
- `(sira + k) % 26`. Sıra *k* kadar ilerletilir. Sonuç 25'i aşarsa % 26 (26'ya bölümden kalan) onu başa sarar: *(25 + 3) % 26 = 2*.
- `(char) ('A' + …)`. Yeni sıra yeniden harfe çevrilir: `'A'`'nın koduna sıra eklenir ve sonuç `char`'a dönüştürülür.
- `new String(harf)`. Değiştirilmiş dizi yeniden metne çevrilip döndürülür.

`sifrele("ZEKA", 3)` çağrısını izleyelim:

`i`	`harf[i]`	`sira`	`(sira + 3) % 26`	**Yeni harf**	`harf`
0	Z	25	28 % 26 = 2	C	[C, E, K, A]
1	E	4	7	H	[C, H, K, A]
2	K	10	13	N	[C, H, N, A]
3	A	0	3	D	[C, H, N, D]

Tablo 3.15: `sifrele("ZEKA", 3)` izlemesi.

Metot `"CHND"` döndürür. İlk satır, başa sarmanın nasıl çalıştığını gösterir: Z'nin sırası 25'tir, 3 ilerletilince 28 olur ve 28 % 26 = 2 sonucu C harfine karşılık gelir.



Şekil 3.17: Sezar şifresinin iki anı. Solda `i = 0` : `sira` = 'Z' − 'A' = 25, (25 + 3) % 26 = 2 → 'C'; ilk hücre C olmuştur. Sağda döngü bittiğinde `harf` dizisi C, H, N, D'dir.

Şifreyi çözmek için ayrı bir metot gerekmez. Her harf 3 ileri kaydırıldıysa, 23 ileri daha kaydırılınca toplam 26 adım atılmış olur ve alfabe bir tam tur döner. Bu yüzden çözme, aynı metodu *k = 26 − 3 = 23* ile çağırmaktır: `sifrele("CHND", 23)` yeniden `"ZEKA"` verir.

Her harf bir kez işlenir; *n* harfli metin için süre *O(n)*'dir. `toCharArray` ve `new String` de birer *O(n)* kopyalama yapar. Bu sabit sayıda geçiş, toplam maliyeti değiştirmez.

Sık yapılan hata

Metodun her türlü metinle çalıştığını sanmak. Kod, metnin yalnız A–Z arasındaki büyük harflerden oluştuğunu varsayar. Boşluk, rakam ya da küçük harf içeren bir metinde `harf[i] - 'A'` 0–25 dışında bir değer verir ve sonuç anlamsız karakterler olur. Ç, Ğ, İ, Ö, Ş, Ü gibi Türkçe

harflerin kodları da A–Z aralığının dışındadır. Gerçek bir uygulamada bu karakterler ya olduğu gibi bırakılmalı ya da ayrı bir alfabe dizisiyle ele alınmalıdır.

3.4 Çok boyutlu diziler

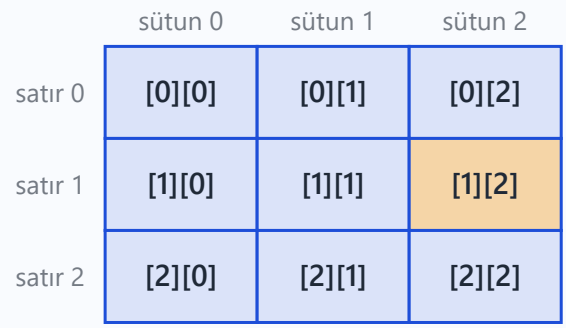
Bazı veriler doğal olarak tek bir sıra hâlinde değil, bir tablo hâlinde durur: bir sınıfın haftalık ders programı, satranç tahtası, bir resmin pikselleri, bir şehrin semtler arası uzaklık çizelgesi. Bu verileri tek boyutlu bir diziye sıkıştırmak mümkündür, ama her erişimde satır ve sütun numarasından tek bir indis hesaplamak gerekir. İki boyutlu dizi bu hesabı dilin kendisine bırakır.

3.4.1 İki boyutlu diziler

İki boyutlu dizi, öğeleri satırlar ve sütunlar hâlinde düzenlenmiş, tablo benzeri bir yapıdır. Matematikteki **matrisin** programlamadaki karşılığıdır. Bir öğeye iki indisle erişilir: `matris[satır][sütun]`. İlk indis satırı, ikinci indis sütunu seçer; ikisi de 0'dan başlar.

```
int[][] matris = new int[3][3];
matris[1][2] = 7;   // 2. satır, 3. sütun
```

İlk satır 3 satır ve 3 sütunlu, bütün hücreleri 0 olan bir matris oluşturur. İkinci satır, 1 indisli satırın 2 indisli sütununa 7 yazar. İndisler 0'dan başladığı için bu hücre, insanların saymasıyla 2. satırın 3. sütunudur.



Şekil 3.18: 3 satır ve 3 sütunlu matris. Her hücre `[satır][sütun]` ile adreslenir; vurgulu hücre `[1][2]` 'dir, yani 1. satırın (ikinci satır) 2. sütunu (üçüncü sütun).

İki boyutlu diziler, satranç tahtası (8 × 8), resim (piksel ızgarası) ve hesap tablosu gibi verilerde kullanışlıdır. Bir resimde `resim[y][x]`, yukarıdan `y`. satırdaki, soldan `x`. pikselin rengidir.

Sık yapılan hata

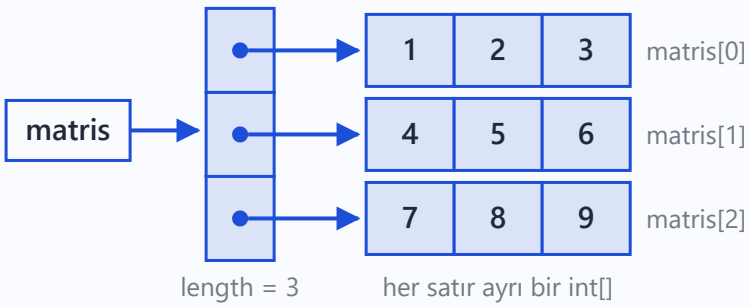
İndislerin sırasını karıştırmak. `matris[1][2]` ile `matris[2][1]` farklı hücrelerdir. Java'da ilk indis her zaman satırı seçer. Koordinat düzleminde önce *x* (yatay), sonra *y* (dikey) yazmaya alışkın olanlar, resim dizisinde `resim[x][y]` yazıp satır ile sütunu yer değiştirebilir.

3.4.2 Java'da matris: dizilerin dizisi

Bazı dillerde iki boyutlu dizi, bellekte satır satır art arda dizilmiş tek bir blok hâlindedir. Java'da durum farklıdır. Java'da gerçek bir iki boyutlu dizi türü yoktur; matris, **dizilerin dizisidir**. `int[][]`, öğeleri `int[]` olan bir dizidir.

```
int[][] matris = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
matris.length      // 3 (satır sayısı)
matris[0].length   // 3 (sütun sayısı)
```

Bu kod bellekte dört dizi oluşturur. Birincisi, üç öğeli bir **dış dizidir**; öğeleri sayı değil, referanstır. Diğer üçü, her biri bir satırı tutan üç öğeli `int` dizileridir. Dış dizinin her hücresi bir satır dizisini gösterir. `matris` değişkeni de dış diziyi gösterir.



Şekil 3.19: `matris` değişkeni 3 referanslık bir dış diziyi gösterir (`length` = 3). Dış dizinin her hücresi ayrı bir `int[]` satır dizisini gösterir: `matris[0]` = 1 2 3, `matris[1]` = 4 5 6, `matris[2]` = 7 8 9.

Bu yapının birkaç doğrudan sonucu vardır:

- `matris.length` dış dizinin uzunluğudur, yani **satır sayısıdır** (3).
- `matris[0]` bir satır dizisidir; `matris[0].length` o satırın uzunluğu, yani **sütun sayısıdır** (3).
- `matris[1][2]` iki adımda okunur: önce `matris[1]` ile 1. satır dizisine ulaşılır, sonra o dizinin 2 indisli öğesi okunur. Sonuç 6'dır.

- İlklendiricide boyut yazılmaz. Satır sayısı dış listedeki iç liste sayısından, sütun sayısı iç listelerdeki öğe sayısından çıkar.

En önemli sonuç şudur: her satır ayrı bir dizi olduğu için satırların uzunlukları farklı olabilir. Böyle bir diziye **düzensiz dizi** (*jagged array*) denir.

3.4.3 Düzensiz dizi: Pascal üçgeni

> **Tanım 3.5**
>
> **Düzensiz dizi**, satırlarının uzunlukları birbirinden farklı olabilen iki (ya da daha çok) boyutlu dizidir. Java'da dış dizi önce yalnız satır sayısı verilerek oluşturulur: `new int[n][]`. Her satır daha sonra, kendi uzunluğuyla ayrıca oluşturulur: `dizi[i] = new int[k];`

Düzensiz dizinin doğal bir örneği **Pascal üçgenidir**. Üçgenin her satırı bir öncekinden bir öğe uzundur. Her satırın iki ucu 1'dir. Aradaki her sayı, hemen üstündeki iki sayının toplamıdır:

> *p[i][j] = p[i−1][j−1] + p[i−1][j]*

İlk dört satır 1; 1 1; 1 2 1; 1 3 3 1'dir. Kare bir matris kullanılsaydı her satır en uzun satır kadar yer kaplardı ve hücrelerin yarısına yakını boş kalırdı. Düzensiz dizi her satıra yalnız gerektiği kadar yer ayırır. Aşağıdaki kod ilk 4 satırı oluşturur:

```
int n = 4;
int[][] p = new int[n][];          // satırlar yok
for (int i = 0; i < n; i++) {
    p[i] = new int[i + 1];         // i + 1 uzunluk
    p[i][0] = p[i][i] = 1;
    for (int j = 1; j < i; j++)
        p[i][j] = p[i-1][j-1] + p[i-1][j];
}
```

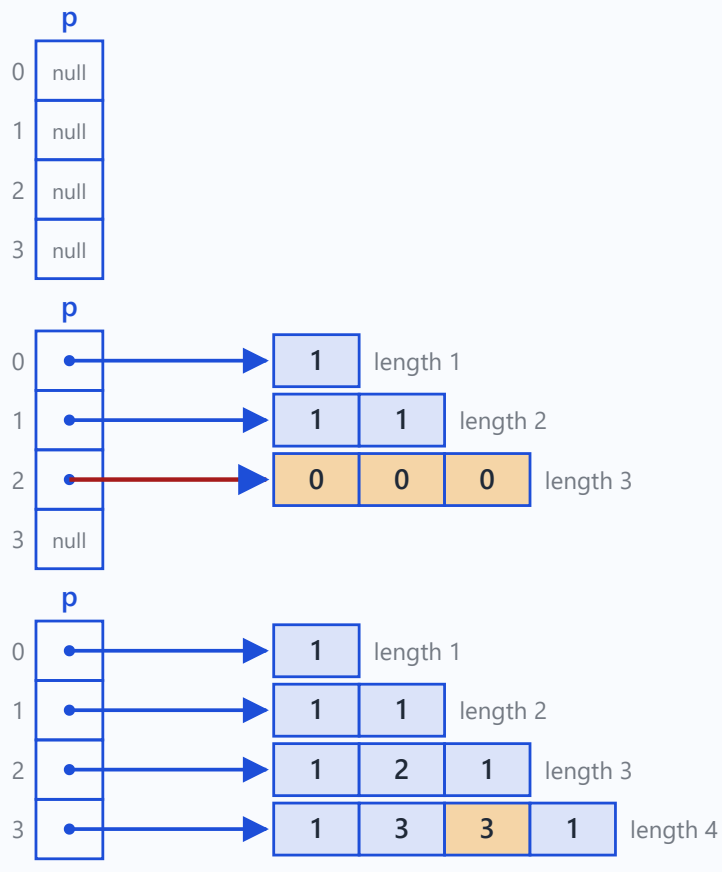
Kodun çalışması şöyledir. `new int[n][]` yalnız 4 referanslık dış diziyi oluşturur; ikinci köşeli parantezin boş bırakılması, satır dizilerinin henüz oluşturulmadığını söyler. Dış dizinin dört hücresi de `null`'dır. Döngünün her turunda önce `i`. satır `i + 1` uzunlukta oluşturulur. Sonra satırın iki ucuna 1 yazılır. `p[i][0] = p[i][i] = 1` bir **zincirleme atamadır** ve sağdan sola çalışır: önce `p[i][i] = 1` yapılır, bu atamanın değeri (1) de `p[i][0]`'a yazılır. Son olarak iç döngü, uçlar dışındaki hücreleri üst satırdan hesaplar.

`i`	Satırın oluşturulması	Uçlar	İç hücreler (`j` = 1 … `i` − 1)	`p[i]`
—	`new int[4][]`	—	—	dört satır da `null`

`i`	Satırın oluşturulması	Uçlar	İç hücreler (`j` = 1 … `i` − 1)	`p[i]`
0	`new int[1]`	`p[0][0]` = 1	yok (`j` < 0)	[1]
1	`new int[2]`	`p[1][0]` = `p[1][1]` = 1	yok (`j` < 1)	[1, 1]
2	`new int[3]`	`p[2][0]` = `p[2][2]` = 1	`p[2][1]` = 1 + 1 = 2	[1, 2, 1]
3	`new int[4]`	`p[3][0]` = `p[3][3]` = 1	`p[3][1]` = 1 + 2 = 3; `p[3][2]` = 2 + 1 = 3	[1, 3, 3, 1]

Tablo 3.16: Pascal üçgeninin oluşturulması (`n` = 4).

0. satırda `p[0][0]` hem baş hem son hücredir; zincirleme atama aynı hücreye iki kez 1 yazar. 0. ve 1. satırlarda iç döngü hiç dönmez, çünkü uçlar dışında hücre yoktur. 3. satırın iç hücreleri 2. satırdan hesaplanır: `p[3][1] = p[2][0] + p[2][1] = 1 + 2 = 3` ve `p[3][2] = p[2][1] + p[2][2] = 2 + 1 = 3`.



Şekil 3.20: Pascal üçgeninin oluşturulması. (a) `new int[4][]` 'den sonra: dış dizinin dört hücresi de `null` 'dır. (b) `p[2] = new int[3]` 'ten hemen sonra: ilk iki satır dolu, yeni satır sıfırlarla başlar. (c) Son durum: uzunlukları 1, 2, 3, 4 olan dört ayrı satır; son hesaplanan `p[3][2]` = 3 vurgulu.

Üçgen toplam 1 + 2 + 3 + 4 = 10 hücre kullanır. Aynı veriyi 4 × 4'lük bir matriste tutmak 16 hücre gerektirirdi. *n* satırda fark daha da büyür: düzensiz dizi *n(n + 1)/2*, kare matris n^2 hücre kullanır. Kazanç, yarıya yakın bellektir.

Sık yapılan hata

Satırları oluşturmadan erişmek. `int[][] m = new int[3][];` satırından sonra `m[0][0] = 5` yazılırsa `NullPointerException` fırlatılır: `m[0]` henüz `null` 'dır ve `null` 'ın 0. öğesi yoktur. Her satıra önce `m[i] = new int[k];` ile bir dizi atanmalıdır.

3.4.4 İki boyutlu dizide sığ ve derin kopya

Matrisin dizilerin dizisi olması, kopyalamada beklenmedik bir sonuç doğurur. 3.2.6'da `clone()` 'un dizinin bağımsız bir kopyasını oluşturduğu söylenmişti. Bu, öğeleri sayı olan dizi için doğrudur. Matriste ise dış dizinin öğeleri referanstır ve `clone()` referansları kopyalar, gösterdikleri satırları değil.

Tanım 3.6

Öğeleri referans olan bir dizinin **sığ kopyası** (*shallow copy*), yalnız dizinin kendisi kopyalanarak elde edilir; kopyanın öğeleri, orijinalin gösterdiği nesnelerin aynısını gösterir. **Derin kopya** (*deep copy*) ise gösterilen nesneleri de kopyalar; kopya ile orijinal hiçbir nesneyi paylaşmaz.

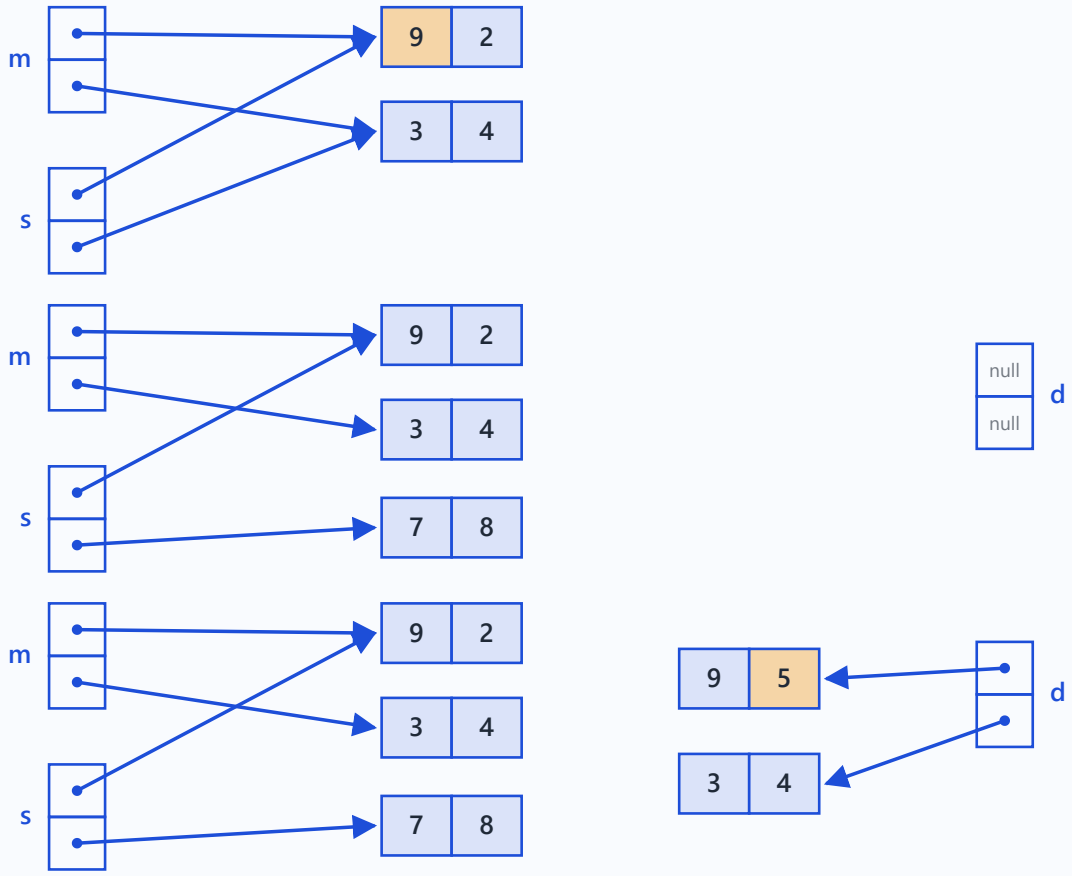
Aşağıdaki kod iki kopya türünü yan yana gösterir:

```
int[][] m = { {1, 2}, {3, 4} };
int[][] s = m.clone();          // sığ kopya
s[0][0] = 9;                    // m[0][0] de 9!
s[1] = new int[] {7, 8};        // yalnız s değişir
int[][] d = new int[2][];       // derin kopya:
for (int i = 0; i < 2; i++)
    d[i] = m[i].clone();        // satırlar da
d[0][1] = 5;                    // m etkilenmez
```

Örnek 3.4

1. `int[][] m = { {1, 2}, {3, 4} };` Dış dizi `m` iki satır dizisini gösterir: {1, 2} ve {3, 4}.
2. `int[][] s = m.clone();` Yalnız dış dizi kopyalanır. `s` yeni bir dış dizidir, ama içindeki iki referans `m` 'dekilerle aynıdır: `s[0]` ve `m[0]` aynı satırı gösterir. Bu bir sığ kopyadır.
3. `s[0][0] = 9;` `s[0]` ortak satırdır. Değişiklik o satırda yapılır ve `m[0][0]` de 9 olur. `m` = {{9, 2}, {3, 4}}.

4. `s[1] = new int[] {7, 8};` Bu kez bir satırın içi değil, `s` 'nin kendi dış dizisindeki referans değiştirilir. `s[1]` artık yeni bir satırı gösterir; `m[1]` hâlâ {3, 4}'tür. `s` = {{9, 2}, {7, 8}}.
5. `int[][] d = new int[2][];` Derin kopya için önce iki referanslık yeni bir dış dizi oluşturulur. Satırları henüz yoktur.
6. **Döngü:** `d[0] = m[0].clone()` ve `d[1] = m[1].clone()` . Her satır ayrı ayrı kopyalanır. O anda `m[0]` {9, 2} olduğu için `d[0]` = {9, 2}, `d[1]` = {3, 4} olur.
7. `d[0][1] = 5;` Yalnız `d` 'nin kendi satırı değişir: `d` = {{9, 5}, {3, 4}}. `m[0][1]` hâlâ 2'dir.



Şekil 3.21: Sığ ve derin kopya. (a) `s[0][0] = 9` 'dan sonra: `m` ve `s` dış dizileri aynı iki satırı gösterir; ortak satırdaki değişiklik iki yandan da görünür. (b) `s[1] = new int[] {7, 8}` ve `d = new int[2][]` 'den sonra: `s[1]` artık ayrı bir satırı gösterir; `d` 'nin iki hücresi de `null` 'dır. (c) Son durum: `d` 'nin satırları (9, 5 ve 3, 4) `m` 'nin satırlarından bağımsızdır.

Örnekteki 3. ve 4. adımlar, sığ kopyanın davranışını anlamanın anahtarıdır. `s[0][0] = …` ortak bir satırın *içini* değiştirir; bu değişiklik `m` 'den de görünür. `s[1] = …` ise `s` 'nin dış dizisindeki bir *referansı* değiştirir; `s` 'nin dış dizisi `m` 'ninkinden ayrı olduğu için `m` etkilenmez. Kural kısaca şudur: sığ kopyada iki dış dizi ayrıdır, satırlar ortaktır.

Derin kopyanın maliyeti, bütün hücrelerin kopyalanmasıdır: r satır ve c sütunlu matriste $O(r \cdot c)$ zaman ve $O(r \cdot c)$ ek bellek. Sığ kopya yalnız dış diziyi kopyaladığı için $O(r)$ sürer. Sığ kopya ucuzdur, ama ancak satırlar hiç değiştirilmeyecekse güvenlidir.

> **Sık yapılan hata**
>
> Bir matrisi `m.clone()` ya da `Arrays.copyOf(m, m.length)` ile kopyalayıp kopya üzerinde hücre değiştirmek. İki yöntem de sığ kopya verir ve hücre değişiklikleri orijinal matrise yansır. Bağımsız bir matris için her satır ayrıca kopyalanmalıdır: `d[i] = m[i].clone();`

3.4.5 Matrisi doldurma ve yazdırma

Matrisin bütün hücrelerini gezmenin standart yolu iç içe iki döngüdür: dış döngü satırları, iç döngü sütunları gezer. Aşağıdaki program 3 × 3'lük bir matrisi 1'den 9'a kadar sayılarla doldurur ve tablo biçiminde yazdırır:

```
public class CokBoyutluDiziOrnegi {
  public static void main(String[] a) {
    int[][] matris = new int[3][3];
    // ilk değer atama: 1…9
    for (int i = 0; i < 3; i++)      // satır
      for (int j = 0; j < 3; j++)    // sütun
        matris[i][j] = i * 3 + j + 1;

    System.out.println("Matrisin İçeriği:");
    for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 3; j++)
        System.out.print(matris[i][j] + " ");
      System.out.println();   // yeni satır
    }
  }
}
```

Çıktı:

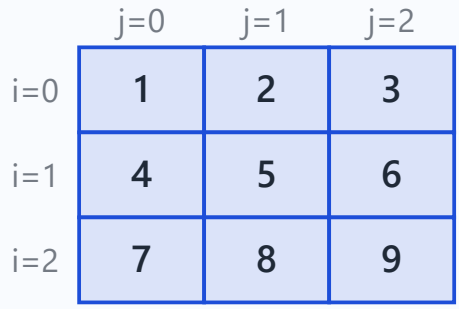
```
Matrisin İçeriği:
1 2 3
4 5 6
7 8 9
```

Doldurma döngüsünde her hücreye `i * 3 + j + 1` yazılır. Bu ifade satır satır numaralandırmanın formülüdür. `i`. satırdan önce `i` tam satır, yani `i × 3` hücre vardır. Satırın içinde `j` hücre daha geçilir. Sonuca 1 eklenir, çünkü numaralar 0'dan değil 1'den başlıyor. Tablo 3.17 dokuz atamayı sırasıyla gösterir.

`i`	`j`	`i * 3 + j + 1`	**Atama**
0	0	0 + 0 + 1	`matris[0][0]` = 1
0	1	0 + 1 + 1	`matris[0][1]` = 2
0	2	0 + 2 + 1	`matris[0][2]` = 3
1	0	3 + 0 + 1	`matris[1][0]` = 4
1	1	3 + 1 + 1	`matris[1][1]` = 5
1	2	3 + 2 + 1	`matris[1][2]` = 6
2	0	6 + 0 + 1	`matris[2][0]` = 7
2	1	6 + 1 + 1	`matris[2][1]` = 8
2	2	6 + 2 + 1	`matris[2][2]` = 9

Tablo 3.17: Doldurma döngüsünün izlemesi. Her satır, iç döngü gövdesinin bir çalışmasıdır.

Tablodaki sıra, iç içe döngülerin çalışma düzenini gösterir. `i` sabit kalırken `j` 0, 1, 2 değerlerini alır; iç döngü bitince `i` bir artar ve `j` yeniden 0'dan başlar. Dış döngünün her turu, iç döngünün bütün turlarını içerir. Bu yüzden gövde 3 × 3 = 9 kez çalışır. Genel olarak $r \times c$'lik bir matriste gövde $r \cdot c$ kez çalışır ve doldurmanın maliyeti $O(r \cdot c)$'dir.



Şekil 3.22: Doldurma bittiğinde matris. Satırlar `i` = 0, 1, 2; sütunlar `j` = 0, 1, 2. Hücreler satır satır 1'den 9'a numaralanmıştır.

Yazdırma döngüsü aynı düzeni izler, ama bir farkla. İç döngü bir satırın öğelerini `print` ile aynı satıra yazar. İç döngü bittiğinde dış döngünün gövdesindeki `System.out.println()` çalışır ve imleç alt satıra geçer. `println()` iç döngünün içine konsaydı her sayı ayrı satıra yazılırdı; hiç konmasaydı dokuz sayı tek satıra yazılırdı.

Programda sınırlar sabit 3 olarak yazılmıştır. Aynı kod `i < matris.length` ve `j < matris[i].length` ile yazılırsa her boyuttaki matriste, düzensiz dizilerde bile doğru çalışır. İç döngünün sınırının `matris[0].length` değil `matris[i].length` olması, düzensiz dizide her satırın kendi uzunluğunun kullanılmasını sağlar.

3.4.6 Üç boyutlu diziler

Java'da üç ya da daha çok boyutlu diziler de tanımlanabilir. Üç boyutlu bir dizi, matrislerden oluşan bir dizidir. Zaman içinde değişen bir tablo (zaman × satır × sütun) ya da renkli bir resmin kırmızı, yeşil ve mavi katmanları böyle tutulabilir.

```
int[][][] ucBoyutluDizi =
        new int[3][3][3];
for (int x = 0; x < 3; x++)
  for (int y = 0; y < 3; y++)
    for (int z = 0; z < 3; z++)
      ucBoyutluDizi[x][y][z] =
          x * 9 + y * 3 + z + 1;

// yazdırma: aynı üç döngü;
// z döngüsünden sonra println (y),
// y döngüsünden sonra println (x)
```

Her boyut bir iç içe döngü demektir. 3 × 3 × 3'lük dizide en içteki atama 27 kez çalışır. Hücrelere yazılan `x * 9 + y * 3 + z + 1` değeri, matris örneğindeki formülün bir boyut genişletilmiş hâlidir: her katman 9 hücre, her satır 3 hücre içerir. Örneğin `ucBoyutluDizi[1][2][0]` = 1 · 9 + 2 · 3 + 0 + 1 = 16'dır.



Şekil 3.23: 3 × 3 × 3'lük dizi, üç katman hâlinde. `x` = 0 katmanı 1–9, `x` = 1 katmanı 10–18, `x` = 2 katmanı 19–27 değerlerini taşır. Her katman bir 3 × 3 matristir; satır `y`, sütun `z` ile seçilir.

Yazdırma için de aynı üç döngü kullanılır. `z` döngüsü bittiğinde bir satır tamamlanmıştır ve `println` ile alt satıra geçilir. `y` döngüsü bittiğinde bir katman tamamlanmıştır ve katmanları ayırmak için bir boş satır daha yazılır.

Boyut sayısı arttıkça öğe sayısı hızla büyür: her boyutu 100 olan üç boyutlu bir `int` dizisi bir milyon öğe, yani yaklaşık 4 MB veri tutar. Dört boyutta bu 400 MB olur. Çok boyutlu diziler kullanılırken bellek ihtiyacı önceden hesaplanmalıdır.

Değişken adında Türkçe karakter kullanmak (örneğin `üçBoyutluDizi`) Java'da geçerlidir. Yine de kaynak dosyalar farklı kodlamalarla açılabildiği ve bazı araçlar bu karakterleri bozabildiği için ASCII adlar (`ucBoyutluDizi`) daha güvenlidir.

3.4.7 Matris transpozu

> **Tanım 3.7**
>
> $m \times n$'lik bir M matrisinin **transpozu** (*transpose*), satırları M'nin sütunlarından oluşan $n \times m$'lik T matrisidir: her i, j için $T[j][i] = M[i][j]$.

Transpozda satırlar sütuna, sütunlar satıra dönüşür. 2 satır ve 3 sütunlu bir matrisin transpozu 3 satır ve 2 sütunludur. Matrisin 0. satırı, transpozun 0. sütunu olur.

```
static int[][] transpoz(int[][] m) {
    int satir = m.length;
    int sutun = m[0].length;
    int[][] t = new int[sutun][satir];
    for (int i = 0; i < satir; i++)
        for (int j = 0; j < sutun; j++)
            t[j][i] = m[i][j];
    return t;
}
static void yazdirMatris(int[][] m) {
    for (int[] s : m) {
        for (int x : s)
            System.out.print(x + " ");
        System.out.println();
    }
}
```

`transpoz` metodu önce boyutları bulur. `m.length` satır sayısıdır; `m[0].length` ilk satırın uzunluğu, yani sütun sayısıdır. Sonra boyutları yer değiştirmiş yeni bir matris oluşturur: `new int[sutun][satir]`. İç içe döngüler `m`'nin her hücresini bir kez gezer ve değerini `t`'deki yansımasına yazar: `t[j][i] = m[i][j]`.

`yazdirMatris` metodu iki for-each döngüsüyle çalışır. Dış döngüde `s` sırayla her satır dizisini, iç döngüde `x` o satırın her öğesini alır. Bu yazım indis kullanmadığı için düzensiz dizilerde de doğru çalışır.

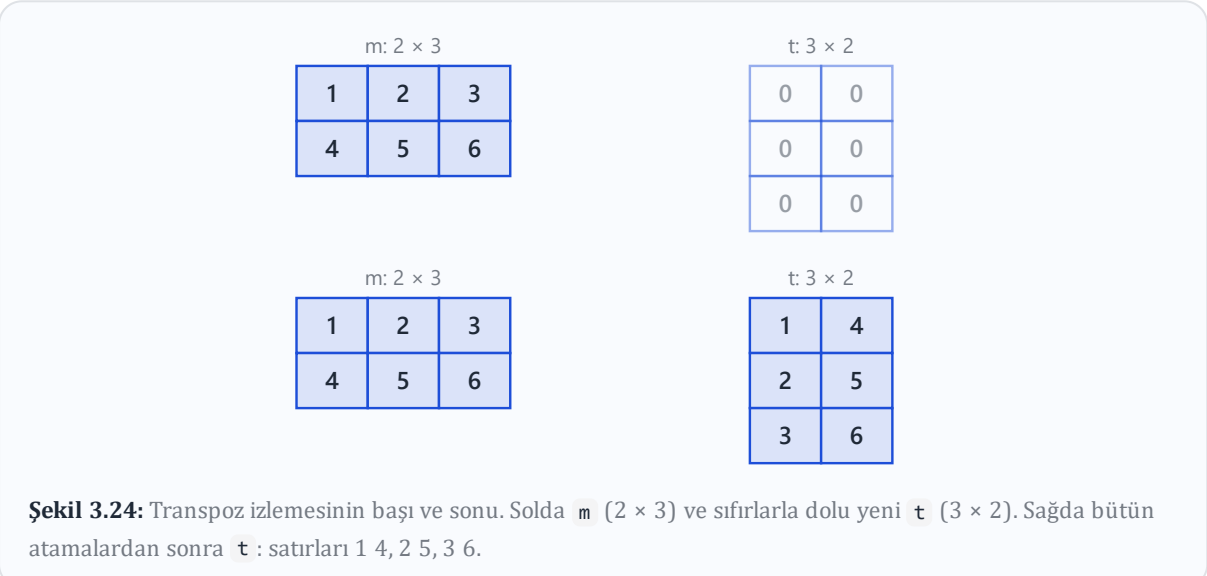
Metotlar `static` tanımlanmıştır. Bunun nedeni, onları çağıracak olan `main`'in de `static` olmasıdır. Statik bir metot, bir nesne oluşturmadan yalnız sınıfın öteki statik metotlarını doğrudan çağırabilir. `static` yazılmazsa `main`'den çağrı derleme hatası verir.

`m` = {{1, 2, 3}, {4, 5, 6}} için `transpoz(m)` çağrısını izleyelim. `satir` = 2, `sutun` = 3'tür ve `t` 3 × 2'lik, sıfırlarla dolu bir matris olarak başlar.

`i`	`j`	Atama	`t` (satır satır)
—	—	`new int[3][2]`	[0, 0] [0, 0] [0, 0]
0	0	`t[0][0]` = `m[0][0]` = 1	[1, 0] [0, 0] [0, 0]
0	1	`t[1][0]` = `m[0][1]` = 2	[1, 0] [2, 0] [0, 0]
0	2	`t[2][0]` = `m[0][2]` = 3	[1, 0] [2, 0] [3, 0]
1	0	`t[0][1]` = `m[1][0]` = 4	[1, 4] [2, 0] [3, 0]
1	1	`t[1][1]` = `m[1][1]` = 5	[1, 4] [2, 5] [3, 0]
1	2	`t[2][1]` = `m[1][2]` = 6	[1, 4] [2, 5] [3, 6]

Tablo 3.18: `transpoz({{1, 2, 3}, {4, 5, 6}})` izlemesi.

İlk üç atamadan sonra `t`'nin 0. sütunu 1, 2, 3 olmuştur; bu, `m`'nin 0. satırıdır. Son üç atama `m`'nin 1. satırını `t`'nin 1. sütununa yazar. `yazdirMatris(t)` çıktısı "1 4", "2 5", "3 6" satırlarıdır.



Şekil 3.24: Transpoz izlemesinin başı ve sonu. Solda `m` (2 × 3) ve sıfırlarla dolu yeni `t` (3 × 2). Sağda bütün atamalardan sonra `t`: satırları 1 4, 2 5, 3 6.

Her öğe bir kez kopyalanır: *m* × *n*'lik matriste *m* · *n* atama yapılır ve süre *O(m · n)*'dir. Sonuç yeni bir matriste tutulduğu için ek bellek de *O(m · n)*'dir. Kare matrislerin transpozu yerinde de alınabilir: köşegenin üstündeki her hücre, köşegenin altındaki yansımasıyla takas edilir. Kare olmayan matriste boyutlar değiştiği için bu yapılamaz.

Sık yapılan hata

Transpoz matrisini `new int[satir][sutun]` ile, yani orijinalle aynı boyutlarda oluşturmak. Kare matriste bu fark edilmez. 2 × 3'lük bir matriste ise `t` 2 × 3 olur ve `t[2][0]` ataması `ArrayIndexOutOfBoundsException` verir. Bir başka hata da kare matriste yerinde transpoz

alırken bütün hücreleri gezmektir: her çift iki kez takas edilir ve matris ilk hâline döner. Yerinde transpozda yalnız köşegenin bir yanı gezilmelidir.

3.4.8 Uygulama: Tic Tac Toe

İki boyutlu dizilerin tipik bir uygulaması, tahta üzerinde oynanan oyunlardır. Tic Tac Toe (X-O oyunu) bunların en basitidir:

- Oyun 3 × 3'lük bir tahtada oynanır; tahta bir `char[][]` dizisidir.
- İki oyuncu, X ve O, sırayla boş bir hücreye kendi işaretini koyar.
- Kendi işaretini aynı satırda, aynı sütunda ya da bir çaprazda üçleyen oyuncu kazanır.
- Tahtada boş hücre kalmaz ve kimse kazanmamışsa oyun berabere biter.

Oyunun her hamlesi tek bir dizi atamasıdır: `tahta[satir][sutun] = oyuncu`. Programın geri kalanı, bu atamanın etrafındaki denetimlerdir. Ana döngü şöyledir:

```
public class TicTacToe {   // import java.util.Scanner;
    static Scanner scanner = new Scanner(System.in); // tek Scanner

    public static void main(String[] args) {
        char[][] tahta = new char[3][3];   // 3x3'lük oyun tahtası
        char oyuncu = 'X';                 // ilk oyuncu X
        boolean oyunDevamEdiyor = true;
        tahtaDoldur(tahta);                // tüm hücreler ' '
        while (oyunDevamEdiyor) {
            tahtayiGoster(tahta);
            hamleYap(tahta, oyuncu);
            oyunDevamEdiyor = oyunDevamEdiyorMu(tahta, oyuncu);
            oyuncu = (oyuncu == 'X') ? 'O' : 'X';   // sıra değişir
        }
        tahtayiGoster(tahta);
    }
```

`main` metodu tahtayı oluşturur, ilk oyuncuyu X yapar ve oyun bitene kadar dört adımı yineler: tahtayı göster, hamle al, oyunun bitip bitmediğini denetle, sırayı öbür oyuncuya ver. Sıra değişimi **koşul işleciyle** yazılmıştır: `(oyuncu == 'X') ? 'O' : 'X'`, oyuncu X ise O'yu, değilse X'i verir.

`new char[3][3]` ile oluşturulan tahtanın hücreleri `'\u0000'` ile başlar (3.2.1). Bu karakter ekranda görünmez ve boşluk karakterinden farklıdır. Boş hücreyi `' '` (boşluk) ile göstermek için tahta oyun başlamadan `tahtaDoldur` ile doldurulur. Sınıfın başında tanımlanan `scanner` alanı da `static`'tir, çünkü statik metotlardan kullanılır; programın tamamı tek bir `Scanner` ile çalışır.

```
static void tahtaDoldur(char[][] t) {
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      t[i][j] = ' ';
}
static void tahtayiGoster(char[][] t) {
  System.out.println("-------------");
  for (int i = 0; i < 3; i++) {
    System.out.print("| ");
    for (int j = 0; j < 3; j++)
      System.out.print(t[i][j]+" | ");
    System.out.println("\n-------------");
  }
}
```

`tahtaDoldur` iç içe döngüyle dokuz hücrenin hepsine boşluk yazar. `tahtayiGoster` tahtayı çizgilerle çevrili bir tablo olarak yazar. Her satırdan sonra yazılan `"\n-------------"` metnindeki `\n`, satır sonu karakteridir; önce satır biter, sonra ayırıcı çizgi yazılır.

```
static void hamleYap(char[][] t,
                     char oyuncu) {
  int satir, sutun;
  do {
    System.out.print("Sıra " + oyuncu
      + ": satır ve sütun (1-3): ");
    satir = scanner.nextInt() - 1;
    sutun = scanner.nextInt() - 1;
  } while (satir < 0 || satir > 2
        || sutun < 0 || sutun > 2
        || t[satir][sutun] != ' ');
  t[satir][sutun] = oyuncu;
}
// kullanıcı 1-3 girer, dizi 0-2;
// geçersiz/dolu hücre → tekrar sor
```

`hamleYap` metodu, geçerli bir hamle gelene kadar kullanıcıya sorar. Bunun için bir **do-while döngüsü** kullanır. Bu döngünün gövdesi koşul denetlenmeden önce bir kez çalışır; ilk soruyu sormadan önce denetlenecek bir şey olmadığı için bu yapı burada doğaldır. Kullanıcı satır ve sütunu 1–3 aralığında girer, dizi ise 0–2 kullanır; bu yüzden okunan sayılardan 1 çıkarılır. Döngü üç durumda soruyu yineler: satır geçersizse, sütun geçersizse ya da hücre dolu ise. Koşuldaki sıra önemlidir: önce indislerin geçerliliği denetlenir. `||` işleci soldan sağa değerlendirilir ve ilk doğru koşulda durur. Böylece geçersiz indisle `t[satir][sutun]`'a hiç erişilmez.

3.4.9 Kazanma ve beraberlik denetimi

3 × 3'lük tahtada kazanmanın 8 yolu vardır: 3 satır, 3 sütun ve 2 çapraz. Bu sekiz çizgiyi tek tek yazmak yerine, satır ve sütunlar bir döngüyle, çaprazlar tek bir `return` ifadesiyle denetlenir:

```
static boolean kazandi(char[][] t, char o) {
    for (int i = 0; i < 3; i++)
        if ((t[i][0] == o && t[i][1] == o && t[i][2] == o)     // satır i
         || (t[0][i] == o && t[1][i] == o && t[2][i] == o))    // sütun i
            return true;
    return (t[0][0] == o && t[1][1] == o && t[2][2] == o)      // ana çapraz
        || (t[0][2] == o && t[1][1] == o && t[2][0] == o);     // ters çapraz
}
static boolean oyunDevamEdiyorMu(char[][] t, char o) {
    if (kazandi(t, o)) {
        System.out.println("Tebrikler, " + o + " oyuncusu kazandı!");
        return false;
    }
    for (char[] satir : t)
        for (char c : satir)
            if (c == ' ') return true;          // boş hücre var → devam
    System.out.println("Oyun berabere bitti.");
    return false;
}
```

`kazandi` metodunun döngüsü `i` = 0, 1, 2 için iki çizgiyi birden sınar. `t[i][0]`, `t[i][1]`, `t[i][2]` `i`. satırın; `t[0][i]`, `t[1][i]`, `t[2][i]` `i`. sütunun hücreleridir. Satır ile sütun arasındaki tek fark indislerin yer değiştirmesidir; transpozdaki ilişkinin aynısı. Döngüde kazanan bir çizgi bulunmazsa iki çapraz denetlenir: ana çapraz `[0][0]`, `[1][1]`, `[2][2]`; ters çapraz `[0][2]`, `[1][1]`, `[2][0]`.

`oyunDevamEdiyorMu` önce son hamleyi yapan oyuncunun kazanıp kazanmadığına bakar. Kazanmışsa tebrik iletisini yazar ve `false` döndürür. Kazanmamışsa tahtada boş hücre arar. Tek bir boş hücre bulmak oyunun sürmesi için yeter; metot ilk boş hücrede `true` döndürür. Hiç boş hücre yoksa oyun berabere biter. Yalnız son hamleyi yapan oyuncunun denetlenmesi yeterlidir, çünkü bir hamle yalnız onu yapanı kazandırabilir.

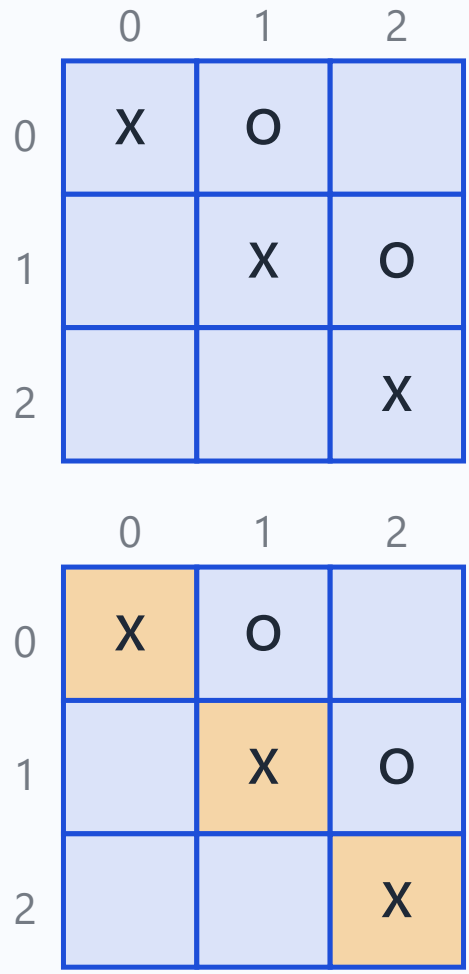
Kazanma denetimini şu tahta üzerinde izleyelim: son hamleyi X yapmıştır ve tahta satır satır `X O _` / `_ X O` / `_ _ X` durumundadır (`_` boş hücre).

Denetim	İlk yanlış hücre	Bakılmayan	Sonuç
`i = 0`, satır 0	`t[0][1]` = 'O'	1	yanlış
`i = 0`, sütun 0	`t[1][0]` = boş	1	yanlış
`i = 1`, satır 1	`t[1][0]` = boş	2	yanlış
`i = 1`, sütun 1	`t[0][1]` = 'O'	2	yanlış
`i = 2`, satır 2	`t[2][0]` = boş	2	yanlış
`i = 2`, sütun 2	`t[0][2]` = boş	2	yanlış

Denetim	İlk yanlış hücre	Bakılmayan	Sonuç
ana çapraz	yok: üç hücre de 'X'	—	**doğru** → `return true`

Tablo 3.19: `kazandi(t, 'X')` izlemesi. "Bakılmayan" sütunu, `&&`'nin kısa devresi yüzünden hiç okunmayan hücre sayısıdır.

Tablo, `&&` ve `||` işleçlerinin **kısa devre** davranışını gösterir. `&&` ile bağlı üç koşuldan biri yanlış çıkınca ifadenin sonucu kesin olarak yanlıştır ve kalan koşullara bakılmaz. Satır 0'da `t[0][0]` 'X'tir, ama `t[0][1]` 'O' olduğu için `t[0][2]`'ye hiç bakılmaz. Satır 1'de ilk hücre `t[1][0]` zaten boştur ve kalan iki hücre atlanır. Son satırda `||`'nin sol tarafı, yani ana çapraz doğrudur; ifadenin sonucu kesinleşmiştir ve ters çapraza hiç bakılmaz. Metot `true` döndürür.



Şekil 3.25: `kazandi(t, 'X')` izlemesinin başı ve sonu. Solda tahta: 0. satır X O boş, 1. satır boş X O, 2. satır boş boş X. Sağda ana çapraz (`[0][0]`, `[1][1]`, `[2][2]`) vurgulu: üç hücre de X'tir.

Tahta hep 3 × 3 olduğu için `kazandi` en çok 8 çizgiyi, yani 24 hücreyi denetler; bu sabit bir iştir. Tahta $n \times n$ olsaydı ve n işaretin aynı çizgide dizilmesi gerekseydi, $2n + 2$ çizginin her biri n hücre içerirdi ve denetim $O(n^2)$ sürerdi. Daha akıllı bir denetim yalnız son hamlenin satırına, sütununa ve (varsa) çaprazlarına bakar ve $O(n)$'de biter.

Sık yapılan hata

Beraberliği kazanma denetiminden önce yapmak. Son hamle tahtadaki son boş hücreyi doldurup aynı zamanda bir çizgiyi tamamlamış olabilir. Önce "boş hücre kaldı mı?" diye bakılırsa oyun "berabere" ilan edilir, oysa son hamleyi yapan kazanmıştır. `oyunDevamEdiyorMu` bu yüzden önce `kazandi` 'yı çağırır.

3.5 Dizi işlemlerinin karmaşıklığı

Önceki kesimlerde her algoritmanın sonunda bir maliyet verildi: *O(1)*, *O(n)*, *O(n log n)*. Bu kesim bu gösterimi kısaca açıklar ve dizi üzerindeki temel işlemlerin maliyetlerini tek tek gerekçelendirir. Amaç, "bu işlem hızlı mı?" sorusunu ezbere değil, işlemin dizide ne yaptığına bakarak cevaplayabilmektir.

3.5.1 Dizi işlemlerinin maliyeti

Algoritma karmaşıklığı, bir algoritmanın çalışma süresinin ve bellek kullanımının girdi boyutu n büyüdükçe nasıl büyüdüğünü anlatır. Süre saniye olarak ölçülmez, çünkü saniye bilgisayardan bilgisayara değişir. Bunun yerine algoritmanın attığı temel adımlar (karşılaştırma, atama, toplama gibi) sayılır ve bu sayının n'ye göre büyüme hızına bakılır.

Tanım 3.8

Bir algoritmanın adım sayısı, yeterince büyük her n için $c \cdot f(n)$'yi aşmıyorsa (c sabit bir sayı) algoritmanın süresi *O(f(n))*'dir. Bu gösterime **büyük O gösterimi** (*big-O notation*) denir. Sabit çarpanlar ve küçük terimler gösterime yazılmaz: *3n + 5* adım atan algoritma *O(n)*'dir.

Sık karşılaşılan büyüme hızlarının anlamı şudur:

- *O(1)*, **sabit**: adım sayısı n'ye bağlı değildir.
- *O(log n)*, **logaritmik**: n iki katına çıkınca yalnız bir adım artar.
- *O(n)*, **doğrusal**: n iki katına çıkınca adım sayısı da iki katına çıkar.
- *O(n log n)*: doğrusaldan biraz hızlı büyür; iyi sıralama algoritmalarının maliyetidir.
- $O(n^2)$, **karesel**: n iki katına çıkınca adım sayısı dört katına çıkar.

Farkın ne kadar büyük olduğunu bir örnek gösterir. Bir milyon öğeli dizide $\log_2 n$ yaklaşık 20, n bir milyon, n^2 ise bir trilyondur. Saniyede bir milyar adım atan bir bilgisayarda doğrusal bir algoritma binde bir saniyede biter; karesel bir algoritma ise yaklaşık 17 dakika sürer.

Aynı algoritmanın süresi girdinin kendisine de bağlı olabilir. Doğrusal aramada aranan değer dizinin başındaysa iş hemen biter, hiç yoksa bütün dizi gezilir. Bu yüzden maliyetler çoğu zaman **en iyi durum**, **ortalama durum** ve **en kötü durum** için ayrı verilir. Aksi söylenmedikçe verilen maliyet en kötü durumun maliyetidir; algoritmanın hiçbir girdide bundan yavaş çalışmayacağını garanti eder. Tablo 3.20 dizi işlemlerini özetler.

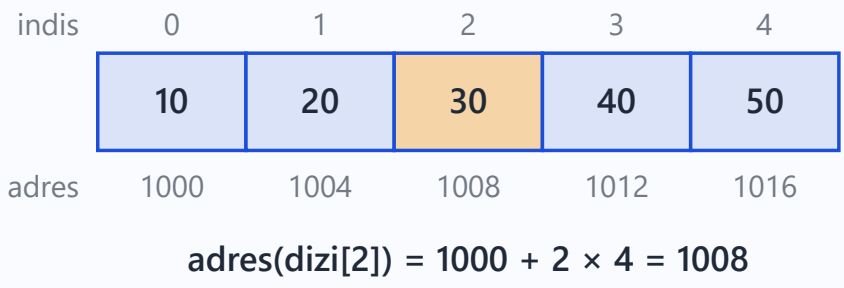
İşlem	Zaman	Neden?
İndisle erişim / güncelleme `a[i]`	*O(1)*	adres doğrudan hesaplanır
Değer arama (sırasız dizi)	*O(n)*	her öğeye bakmak gerekebilir
Değer arama (sıralı dizi, ikili arama)	*O(log n)*	her adımda aralık yarıya iner
Sona ekleme / sondan silme (yer varsa)	*O(1)*	kaydırma yok
Araya ekleme / aradan silme	*O(n)*	sonraki öğeler kaydırılır
Sıralama (iyi bir algoritma)	*O(n log n)*	ör. birleştirme sıralaması
Kopyalama / büyütme	*O(n)*	bütün öğeler yeni diziye taşınır

Tablo 3.20: Dizi işlemlerinin zaman maliyeti.

3.5.2 Erişim: O(1)

```
int[] dizi = {10, 20, 30, 40, 50};
int oge = dizi[2];      // 30 — tek adımda, O(1)
```

`dizi[2]` okunurken dizinin başından başlanıp öğeler sayılmaz. 3.1.2'deki formül uygulanır: başlangıç adresine indis ile öğe boyutunun çarpımı eklenir. Bu, dizinin boyutundan bağımsız olarak bir çarpma, bir toplama ve bir bellek okumasıdır. Beş öğeli dizide de beş milyon öğeli dizide de aynı sürer. Erişim bu yüzden *O(1)*'dir. Aynı hesap yazma için de geçerlidir: `dizi[2] = 35` da *O(1)*'dir.



Şekil 3.26: 1000 adresinden başlayan `int` dizisi. Öğeler 4'er bayt arayla 1000, 1004, 1008, 1012, 1016 adreslerindedir. `dizi[2]` 'nin adresi 1000 + 2 × 4 = 1008 olarak tek adımda hesaplanır.

Bu üstünlük her veri yapısında yoktur. Bağlı listede öğeler bellekte dağınık durur ve her öğe yalnız bir sonrakinin yerini bilir. *k*. öğeye ulaşmak için baştan başlayıp *k* adım yürümek gerekir; erişim *O(n)*'dir. İndisle erişimin sık yapıldığı her problemde dizi bu yüzden doğal seçimdir.

3.5.3 Doğrusal arama: O(n)

Erişim, indisi bilinen bir öğeyi okumaktır. Arama ise ters soruyu sorar: belli bir değer dizide var mı, varsa nerede? Dizi sırasızsa değerin nerede olabileceği hakkında hiçbir ipucu yoktur. Tek yol, öğelere baştan başlayarak tek tek bakmaktır. Bu yönteme **doğrusal arama** (*linear search*) denir.

```
static boolean dogrusalArama(int[] dizi,
                             int hedef) {
    for (int oge : dizi)
        if (oge == hedef)
            return true; // bulundu
    return false;
}
```

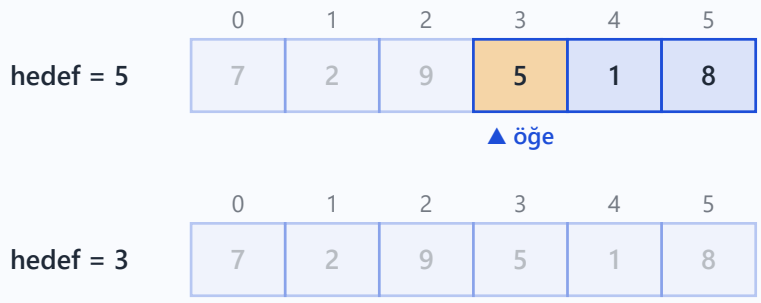
Metot dizinin her öğesini sırayla hedefle karşılaştırır. Eşit bir öğe bulduğu anda `true` döndürür ve döngüden çıkar; kalan öğelere bakmaz. Döngü bütün diziyi gezip bitirdiyse hedef dizide yoktur ve metot `false` döndürür. `return false`'un döngünün *dışında* durması önemlidir.

Metodu {7, 2, 9, 5, 1, 8} dizisinde iki kez çağıralım: önce 5'i, sonra 3'ü arayalım.

Çağrı	Karşılaştırmalar	Sayı	Sonuç
`dogrusalArama(dizi, 5)`	7 ≠ 5, 2 ≠ 5, 9 ≠ 5, 5 == 5	4	`true`
`dogrusalArama(dizi, 3)`	7, 2, 9, 5, 1, 8 hepsi ≠ 3	6	`false`

Tablo 3.21: {7, 2, 9, 5, 1, 8} dizisinde doğrusal arama.

İlk çağrıda 5, 3 indisinde bulunur ve 4 karşılaştırma yapılır; 1 ve 8'e hiç bakılmaz. İkinci çağrıda 3 dizide yoktur. Bunu kesin olarak söyleyebilmek için bütün öğelere bakmak gerekir: *n* = 6 karşılaştırma.



Şekil 3.27: İki arama. Solda hedef 5: 7, 2, 9'a bakılmış (soluk), 5 indis 3'te bulunmuştur; 1 ve 8'e bakılmaz. Sağda hedef 3: altı öğenin hepsine bakılmış ve hedef bulunamamıştır.

Üç durumu ayrı ayrı düşünelim:

- **En iyi durum:** Hedef ilk öğedir. 1 karşılaştırma yeter: *O(1)*.
- **En kötü durum:** Hedef son öğedir ya da dizide yoktur. *n* karşılaştırma gerekir: *O(n)*.
- **Ortalama durum:** Hedef dizide varsa ve her konumda olma olasılığı eşitse, ortalama *(n + 1)/2* karşılaştırma yapılır. Bu da *n* ile doğru orantılıdır: *O(n)*.

Dizi sıralıysa çok daha hızlı bir yol vardır. 3.2.7'deki ikili arama her adımda aralığı yarıya indirir ve *O(log n)*'de biter. Ama diziyi sıralamak *O(n log n)* sürer. Tek bir arama için sıralamak, doğrusal aramadan pahalıdır. Aynı dizide çok sayıda arama yapılacaksa bir kez sıralayıp her aramayı ikili aramayla yapmak kazançlıdır.

> **Sık yapılan hata**
>
> `return false`'u döngünün içine, bir `else` dalına koymak: `if (oge == hedef) return true; else return false;`. Bu durumda metot yalnız ilk öğeye bakar; ilk öğe hedef değilse hemen `false` döndürür. "Bulunamadı" kararı ancak bütün öğelere bakıldıktan sonra verilebilir.

3.5.4 İç içe döngü: iki dizide ortak öğe

İki dizide ortak olan öğeleri bulmak istensin. En düz yol, birinci dizinin her öğesini ikinci dizinin her öğesiyle karşılaştırmaktır:

```
static void eslestir(int[] dizi1, int[] dizi2) {
    for (int oge1 : dizi1)          // n kez
        for (int oge2 : dizi2)      // m kez
            if (oge1 == oge2)
                System.out.println(
                    "Eşleşen öğe: " + oge1);
}
```

Dış döngü `dizi1`'in her öğesi için bir kez döner: *n* tur. Her turda iç döngü `dizi2`'nin bütün öğelerini gezer: *m* tur. Toplam karşılaştırma sayısı $n \cdot m$'dir ve bu sayı girdiye göre değişmez. Eşleşme bulunsa da iç döngü sürer, çünkü kodda erken çıkış yoktur.

`eslestir({3, 7, 5}, {5, 1, 3, 8})` çağrısını izleyelim:

`oge1`	İç döngünün karşılaştırmaları	Eşleşme	Toplam karşılaştırma
3	3 ≠ 5, 3 ≠ 1, 3 == 3, 3 ≠ 8	3	4
7	7 ≠ 5, 7 ≠ 1, 7 ≠ 3, 7 ≠ 8	—	8
5	5 == 5, 5 ≠ 1, 5 ≠ 3, 5 ≠ 8	5	12

Tablo 3.22: `eslestir({3, 7, 5}, {5, 1, 3, 8})` izlemesi.

Metot “Eşleşen öğe: 3” ve “Eşleşen öğe: 5” satırlarını yazar ve 3 × 4 = 12 karşılaştırma yapar. Şekil 3.28, bu karşılaştırmaları bir tablo olarak gösterir: her satır `dizi1`'in bir öğesi, her sütun `dizi2`'nin bir öğesidir ve tablonun her hücresi bir karşılaştırmadır.



Şekil 3.28: İç içe döngünün karşılaştırma tablosu. Satırlar `dizi1` (3, 7, 5), sütunlar `dizi2` (5, 1, 3, 8). 3 × 4 = 12 hücrenin her biri bir karşılaştırmadır; iki hücrede (3 ile 3, 5 ile 5) eşleşme vardır.

İki dizi de n öğeliyse karşılaştırma sayısı n^2'dir: $O(n^2)$. Küçük dizilerde bu sorun değildir. Ama iki dizi de 100.000 öğeli olsa 10 milyar karşılaştırma gerekir. Daha iyi iki yol vardır:

- İkinci diziyi sıralayıp birinci dizinin her öğesini ikili aramayla aramak. Sıralama $O(n \log n)$, n ikili arama da $O(n \log n)$'dir; toplam $O(n \log n)$.
- İkinci dizinin öğelerini bir hash kümesine koyup birinci dizinin her öğesini kümede aramak. Kümede arama ortalama $O(1)$ sürdüğü için toplam ortalama $O(n)$'dir.

Sık yapılan hata

İç içe döngünün maliyetini döngülerin toplamı sanmak. Bir döngü n, öteki m kez dönüyorsa ve biri ötekinin *içindeyse* gövde $n + m$ değil, $n \cdot m$ kez çalışır. Döngüler art arda yazılmışsa ise toplam $n + m$'dir.

3.5.5 Araya ekleme: sağa kaydırma

Dizinin boyutu sabit olduğu için “ekleme” ancak dizide kullanılmayan yer varsa yapılabilir. Uygulamada dizi çoğu zaman kapasitesinden az öğe tutar ve kaç hücrenin dolu olduğu ayrı bir değişkende (`boyut`) izlenir. Dolu hücreler her zaman dizinin başında, arada boşluk olmadan durur. 3.2.11'de anılan dinamik dizi de bu düzenle çalışır.

Dizinin ortasına bir öğe eklemek için önce yer açmak gerekir. Ekleme konumundan sonraki bütün öğeler birer hücre sağa kaydırılır, sonra açılan hücreye yeni değer yazılır:

```
static int ekle(int[] a, int boyut,
                int indis, int deger) {
  // ön koşul: boyut < a.length
  for (int k = boyut; k > indis; k--)
    a[k] = a[k - 1];   // sağa kaydır
  a[indis] = deger;
  return boyut + 1;   // yeni boyut
}
```

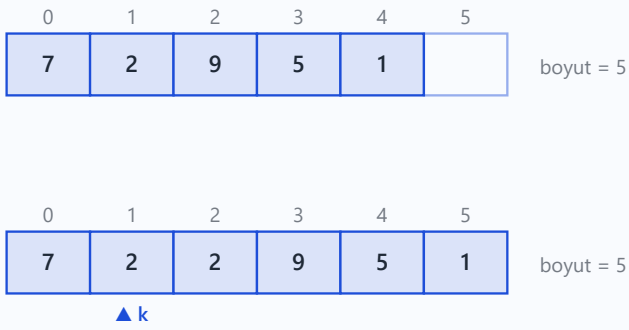
Metot dört parametre alır: dizi `a`, dolu hücre sayısı `boyut`, ekleme konumu `indis` ve eklenecek `deger`. Yorumdaki **ön koşul**, metodun yalnız dizide boş yer varken çağrılabileceğini söyler. Döngü `k`'yi `boyut`'tan başlatır ve `indis`'e kadar geriye doğru götürür. Her turda `k - 1`'deki öğe `k`'ye kopyalanır. Döngü bitince `indis`'teki hücreye yeni değer yazılır ve yeni boyut döndürülür.

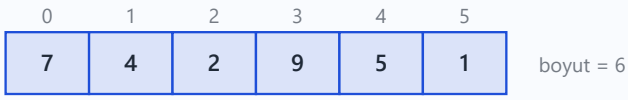
Kapasitesi 6 olan `a` = {7, 2, 9, 5, 1, _} dizisinde (5 hücre dolu) 4'ü 1 indisine ekleyelim: `ekle(a, 5, 1, 4)`.

`k`	**Koşul** `k > 1`	**İşlem**	`a`
—	—	başlangıç	[7, 2, 9, 5, 1, _]
5	doğru	`a[5] = a[4]`	[7, 2, 9, 5, 1, 1]
4	doğru	`a[4] = a[3]`	[7, 2, 9, 5, 5, 1]
3	doğru	`a[3] = a[2]`	[7, 2, 9, 9, 5, 1]
2	doğru	`a[2] = a[1]`	[7, 2, 2, 9, 5, 1]
1	yanlış → biter	`a[1] = 4`	[7, 4, 2, 9, 5, 1]

Tablo 3.23: `ekle(a, 5, 1, 4)` izlemesi. `_` kullanılmayan hücredir (değeri 0).

Metot 6 döndürür. Dört kaydırma yapılmıştır: 1, 5, 9 ve 2 birer sağa geçmiştir. Döngü bittiğinde `a[1]` ile `a[2]` geçici olarak aynı değeri (2) taşır; `a[1] = 4` ataması bu durumu düzeltir.





Şekil 3.29: Araya ekleme. (a) Başlangıç: kapasite 6, dolu 5 hücre (7, 2, 9, 5, 1), `boyut` = 5. (b) Dört kaydırmadan sonra: `k` = 1'de döngü durmuştur; `a[1]` ve `a[2]` ikisi de 2'dir. (c) `a[1] = 4` 'ten sonra: dizi 7, 4, 2, 9, 5, 1 ve `boyut` = 6.

Kaydırmanın neden sağdan başladığı önemlidir. Soldan başlansaydı ilk adımda `a[2] = a[1]` yazılır ve 9 kaybolurdu: 9'un üzerine 2 yazılmıştır ve 9 hiçbir yerde saklanmamıştır. Sonraki adımlarda da 2 sağa doğru çoğalırdı. Sağdan başlamak, her öğenin üzerine yazılmadan önce sağa taşınmasını sağlar. İlk kopyalama boş hücreye yapılır, sonraki her kopyalama da bir önceki adımda taşınmış öğenin eski yerine.

Kaydırma sayısı `boyut - indis` 'tir. Maliyet ekleme konumuna bağlıdır:

- **Başa ekleme** (`indis` = 0): *n* kaydırma, *O(n)*. En kötü durum budur.
- **Ortaya ekleme**: yaklaşık *n/2* kaydırma; bu da *O(n)*'dir.
- **Sona ekleme** (`indis` = `boyut`): döngü hiç dönmez, yalnız bir atama yapılır: *O(1)*.

Dizi doluysa (`boyut == a.length`) ekleme yapılamaz. Önce daha büyük bir dizi oluşturulup öğeler ona kopyalanmalıdır; bu da *O(n)* sürer. Kapasite her seferinde iki katına çıkarılırsa bu kopyalamalar seyrek olur ve sona eklemenin ortalama maliyeti sabit kalır.

3.5.6 Aradan silme: sola kaydırma

Silme, eklemenin tersidir. Bir öğe silindiğinde dizide boşluk kalmaması gerekir; silinen konumdan sonraki bütün öğeler birer hücre sola kaydırılır:

```
static int sil(int[] a, int boyut, int indis) {
  for (int k = indis; k < boyut - 1; k++)
    a[k] = a[k + 1];   // sola kaydır
  a[boyut - 1] = 0;    // eski son hücreyi temizle
  return boyut - 1;    // yeni boyut
}
```

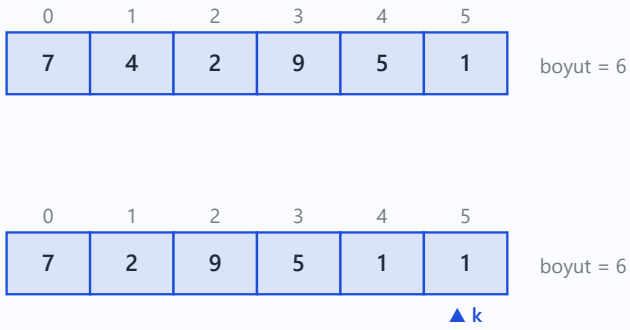
Döngü `k` 'yi silinen konumdan başlatır ve her turda sağdaki öğeyi sola kopyalar. Bu kez kaydırma **soldan** başlar. İlk kopyalama silinecek değerin üzerine yazar; silinen değerin kaybolması zaten istenen şeydir. Sonraki her kopyalama, bir önceki adımda sola taşınmış öğenin eski yerine yazar. Sağdan başlansaydı bu kez öğeler kaybolurdu.

Önceki eklemenin sonucunda `a` = {7, 4, 2, 9, 5, 1}'dir. 1 indisindeki 4'ü silelim: `sil(a, 6, 1)` .

k	Koşul `k < 5`	İşlem	a
—	—	başlangıç	[7, 4, 2, 9, 5, 1]
1	doğru	`a[1] = a[2]`	[7, 2, 2, 9, 5, 1]
2	doğru	`a[2] = a[3]`	[7, 2, 9, 9, 5, 1]
3	doğru	`a[3] = a[4]`	[7, 2, 9, 5, 5, 1]
4	doğru	`a[4] = a[5]`	[7, 2, 9, 5, 1, 1]
5	yanlış → biter	`a[5] = 0`	[7, 2, 9, 5, 1, 0]

Tablo 3.24: `sil(a, 6, 1)` izlemesi.

Metot 5 döndürür ve dizi eklemeden önceki hâline döner. Dört kaydırma yapılmıştır. Döngü bittiğinde son hücrede, sola taşınmış olan 1'in eski bir kopyası kalmıştır. Bu hücre artık kullanılmayan alana düştüğü için `a[5] = 0` ile temizlenir. `int` dizisinde bu temizlik zorunlu değildir; `boyut` zaten 5'e indiği için o hücre okunmaz. Ama nesne dizisinde bu hücreye `null` yazmak gerekir. Yazılmazsa hücre, silinmiş bir nesnenin referansını tutmaya devam eder ve çöp toplayıcı o nesneyi toplayamaz. Programın artık kullanmadığı nesnelerin böyle bellekte kalmasına **bellek sızıntısı** denir.



Şekil 3.30: Aradan silme. Solda başlangıç: 7, 4, 2, 9, 5, 1 ve `boyut` = 6; 1 indisindeki 4 silinecek. Sağda dört kaydırmadan sonra: 2, 9, 5, 1 birer sola geçmiştir; son hücrede 1'in eski kopyası durur. Bu hücre ardından 0 ile temizlenir ve `boyut` 5'e iner.

Kaydırma sayısı `boyut - indis - 1`'dir. Baştan silmede $n - 1$ kaydırma yapılır: $O(n)$. Sondan silmede döngü hiç dönmez: $O(1)$. Ekleme ve silme maliyetlerinin ikisi de konuma bağlıdır ve dizinin başında en yüksektir. Sık sık baştan ekleme ya da silme yapılan bir problemde dizi iyi bir seçim değildir.

Sık yapılan hata

Kaydırma yönünü karıştırmak. Eklemede sağdan sola (`k--`), silmede soldan sağa (`k++`) gidilir. Kural şudur: kaydırmaya, üzerine yazılması güvenli olan hücreden başlanır. Eklemede bu hücre en sağdaki boş hücre, silmede silinen öğenin hücresidir.

3.5.7 Sıralama algoritmalarının maliyeti

Sıralı bir dizi pek çok işi kolaylaştırır: ikili arama, ortanca, mod, tekrar eden öğeleri bulma. Ama sıralamanın kendisi de bir maliyettir ve bu maliyet kullanılan algoritmaya bağlıdır. Tablo 3.25 sık kullanılan beş algoritmayı karşılaştırır.

Algoritma	En iyi	Ortalama	En kötü	Ek bellek
Kabarcık sıralaması (*bubble sort*)	$O(n)$*	$O(n^2)$	$O(n^2)$	$O(1)$
Seçimli sıralama (*selection sort*)	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Eklemeli sıralama (*insertion sort*)	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Hızlı sıralama (*quick sort*)	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$**
Birleştirme sıralaması (*merge sort*)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

Tablo 3.25: Sıralama algoritmalarının zaman ve bellek maliyeti.

* Kabarcık sıralamasının en iyi durumu, hiç takas yapılmayan bir turda duran sürüm için geçerlidir. Zaten sıralı bir dizide bu sürüm tek bir turda durur.

** Hızlı sıralamanın ek belleği özyinelemenin derinliğidir: ortalamada $O(\log n)$, en kötü durumda $O(n)$. Önce küçük parçayı sıralayan gerçeklemeler en kötü durumda da $O(\log n)$'de kalır.

Tablodan üç gözlem çıkar. Birincisi, ilk üç algoritma basittir ve ek bellek istemez, ama ortalama durumda $O(n^2)$'dir; ancak küçük dizilerde kullanışlıdır. İkincisi, hızlı sıralama ortalamada çok hızlıdır, ama en kötü durumu karesel olabilir. Bu durum kötü pivot seçiminde ortaya çıkar. Pivot, diziyi ikiye bölmek için seçilen öğedir. Örneğin zaten sıralı bir dizide her seferinde ilk öğe pivot seçilirse bölmeler hep dengesiz olur. Üçüncüsü, birleştirme sıralaması her durumda $O(n \log n)$'dir, ama $O(n)$ ek bellek ister.

Java'nın `Arrays.sort` metodu türe göre farklı algoritma kullanır. İlkel tipli dizilerde (`int[]`, `double[]` ...) çift pivotlu hızlı sıralama kullanılır. Nesne dizilerinde ise **kararlı** (*stable*) bir algoritma olan TimSort kullanılır. Kararlı bir sıralama, eşit anahtarlı öğelerin baştaki göreli sırasını korur. Öğrencileri önce ada, sonra nota göre sıralayan bir program, ikinci sıralamada aynı notu alanların ad sırasının bozulmamasını ister. İlkel değerlerde eşit öğeler birbirinden ayırt edilemediği için kararlılık gerekmez.

3.5.8 Seçimli sıralama

Seçimli sıralama diziyi iki kısma ayrılmış olarak düşünür: solda sıralanmış kısım, sağda sıralanmamış kısım. Başta sıralanmış kısım boştur. Her turda sıralanmamış kısmın en küçük öğesi seçilir ve o kısmın başındaki öğeyle takas edilir. Böylece sıralanmış kısım bir öğe büyür. $n - 1$ turdan sonra dizi sıralıdır; son öğe kendiliğinden yerine oturur.

```
int[] dizi = {64, 25, 12, 22, 11};
int n = dizi.length;
for (int i = 0; i < n - 1; i++) {
    int minIndis = i;
    for (int j = i + 1; j < n; j++)
        if (dizi[j] < dizi[minIndis])
            minIndis = j;
    // takas: indislerle, yerinde
    int gecici = dizi[minIndis];
    dizi[minIndis] = dizi[i];
    dizi[i] = gecici;
}
```

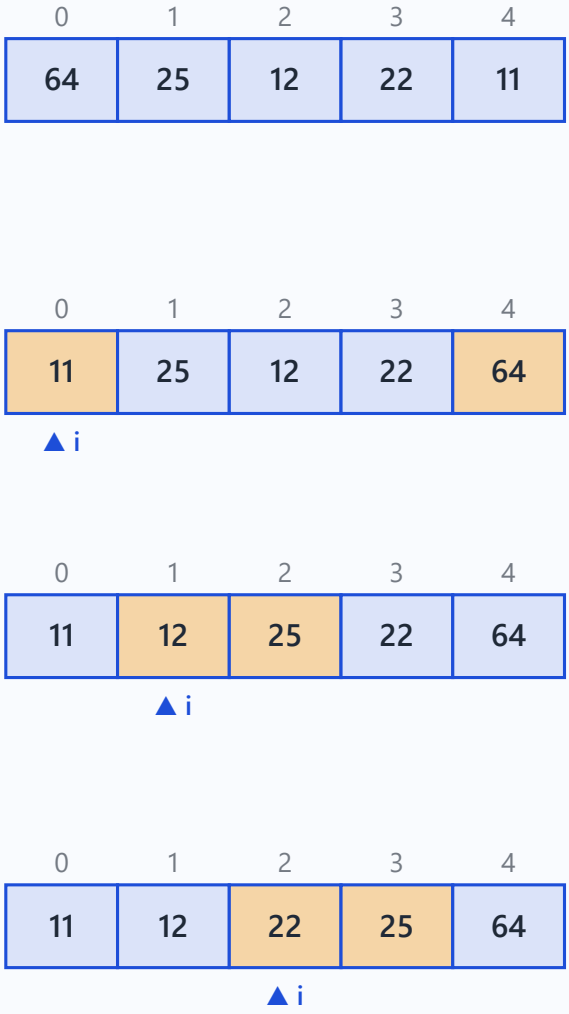
Dış döngünün `i` değişkeni, sıralanmamış kısmın başıdır. `minIndis` şimdilik `i`'dir; yani "şu ana kadar görülen en küçük öğe, sıralanmamış kısmın ilk öğesi"dir. İç döngü `i + 1`'den sonuna kadar gider ve daha küçük bir öğe gördüğünde `minIndis`'i günceller. İç döngü bitince en küçük öğe `i` konumundaki öğeyle takas edilir. Takas, 3.3.8'deki gibi geçici bir değişkenle ve indisler üzerinden yapılır.

{64, 25, 12, 22, 11} dizisini sıralayalım:

Tur (`i`)	İç döngünün karşılaştırmaları	`minIndis`	Takas	Turdan sonra dizi
1 (0)	25 < 64 ✓, 12 < 25 ✓, 22 < 12 ✗, 11 < 12 ✓	4	64 ↔ 11	[11, 25, 12, 22, 64]
2 (1)	12 < 25 ✓, 22 < 12 ✗, 64 < 12 ✗	2	25 ↔ 12	[11, 12, 25, 22, 64]
3 (2)	22 < 25 ✓, 64 < 22 ✗	3	25 ↔ 22	[11, 12, 22, 25, 64]
4 (3)	64 < 25 ✗	3	25 kendisiyle	[11, 12, 22, 25, 64]

Tablo 3.26: Seçimli sıralamanın izlemesi; dizi {64, 25, 12, 22, 11}.

1. turda en küçük öğe 11'dir ve dizinin sonundadır; 64 ile takas edilir ve 0 indisi kesin olarak yerini bulur. 2. turda sıralanmamış kısım 1 indisinden başlar ve en küçüğü 12'dir. 4. turda en küçük öğe zaten başta durur (`minIndis` = `i` = 3); takas kendisiyle yapılır ve dizi değişmez. Son öğe olan 64 kendiliğinden yerindedir.



Şekil 3.31: Seçimli sıralamanın dört anı. (a) Başlangıç: 64, 25, 12, 22, 11. (b) 1. tur takası (64 ↔ 11): 11 başa geçmiştir. (c) 2. tur takası (25 ↔ 12). (d) 3. tur takası (25 ↔ 22): dizi 11, 12, 22, 25, 64 olmuştur. Vurgulu hücreler, o turda yer değiştiren öğelerdir.

Teorem 3.2 (Seçimli sıralamanın karşılaştırma sayısı)

n öğeli bir dizide seçimli sıralama, girdi nasıl olursa olsun tam olarak $n(n-1)/2$ karşılaştırma ve $n-1$ takas yapar. Süresi her durumda $O(n^2)$, ek belleği $O(1)$'dir.

İspat. `i`. turda iç döngü `j` = `i + 1`'den `n - 1`'e kadar döner ve $n-1-i$ karşılaştırma yapar. `i` 0'dan $n-2$'ye kadar değiştiğinde karşılaştırma sayıları $n-1, n-2, \ldots, 1$ olur. Bu sayıların toplamı $n(n-1)/2$'dir. Döngülerin sınırları dizideki değerlere bağlı olmadığı için bu sayı her girdide aynıdır. Her tur tam olarak bir takas yapar (kendisiyle takas dahil): $n-1$ takas. Algoritma yalnız `i`, `j`, `minIndis` ve `gecici` değişkenlerini kullanır. □

Örnekte $n = 5$'tir ve $4 + 3 + 2 + 1 = 10 = 5 \cdot 4 / 2$ karşılaştırma yapılmıştır. Seçimli sıralama zaten sıralı bir dizide de 10 karşılaştırma yapar; girdinin sıralı olduğunu fark etmez. Buna karşılık takas sayısı en çok $n - 1$'dir. Takasın pahalı olduğu durumlarda (örneğin öğeler büyük kayıtlar olduğunda) bu bir üstünlüktür.

> **Sık yapılan hata**
>
> Takası değerler üzerinden yapan bir yardımcı metot yazmak: `yerDegistir(dizi[minIndis], dizi[i])`. Java'da parametreler değerle aktarılır; metot iki `int`'in kopyasını alır ve yalnız kopyaları değiştirir. Dizi değişmez. Takas indislerle, ya doğrudan döngünün içinde ya da diziyi ve iki indisi alan bir metotla yapılmalıdır: `yerDegistir(dizi, i, minIndis)`. Bu metot dizinin referansını aldığı için dizinin kendisini değiştirebilir.

3.5.9 Eklemeli sıralama

Eklemeli sıralama, elindeki iskambil kâğıtlarını sıraya koyan bir oyuncunun yöntemine benzer. Oyuncu kâğıtları birer birer alır ve her yeni kâğıdı, elindeki sıralı kâğıtların arasında doğru yere sokar. Dizide de sol kısım her zaman sıralıdır. Başta bu kısım yalnız `a[0]`'dır; tek öğeli bir dizi her zaman sıralıdır. Her turda sıralı kısmın hemen sağındaki öğe alınır ve sıralı kısımda doğru yerine yerleştirilir.

```
static void eklemeliSirala(int[] a) {
    for (int k = 1; k < a.length; k++) {
        int anahtar = a[k];
        int j = k;
        while (j > 0 && a[j - 1] > anahtar) {
            a[j] = a[j - 1];   // sağa kaydır
            j--;
        }
        a[j] = anahtar;        // yerleş
    }
}
```

Her turda sıradaki öğe `anahtar` değişkenine alınır. Bu andan sonra anahtarın hücresi bir "boşluk" sayılabilir; değeri `anahtar`'da güvendedir. `while` döngüsü boşluğun solundaki öğeye bakar. O öğe anahtardan büyükse bir sağa, boşluğa kaydırılır ve boşluk bir sola geçer. Soldaki öğe anahtardan büyük değilse ya da sıralı kısmın başına gelinmişse (`j == 0`) döngü durur ve anahtar boşluğa yerleştirilir. Bu kaydırma, 3.5.5'teki araya eklemenin aynısıdır; tek fark, ekleme konumunun kaydırma sırasında bulunmasıdır.

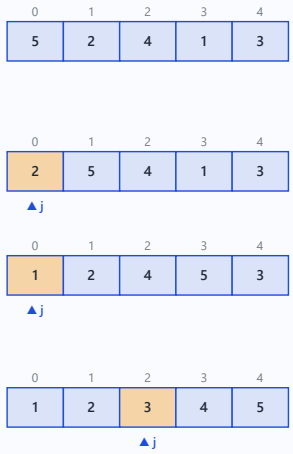
{5, 2, 4, 1, 3} dizisini sıralayalım:

Tur `k`	`anahtar`	Kaydırılanlar	Duruş nedeni	Turdan sonra dizi	Kaydırma
1	2	5	`j == 0`	[2, 5, 4, 1, 3]	1

Tur `k`	`anahtar`	Kaydırılanlar	Duruş nedeni	Turdan sonra dizi	Kaydırma
2	4	5	2 ≤ 4	[2, 4, 5, 1, 3]	1
3	1	5, 4, 2	`j == 0`	[1, 2, 4, 5, 3]	3
4	3	5, 4	2 ≤ 3	[1, 2, 3, 4, 5]	2

Tablo 3.27: Eklemeli sıralamanın izlemesi; dizi {5, 2, 4, 1, 3}.

3. turu ayrıntılı izleyelim. Anahtar 1'dir ve sıralı kısım {2, 4, 5}'tir. `a[2]` = 5 > 1 olduğu için 5 sağa kayar, boşluk 2 indisine geçer. `a[1]` = 4 > 1 olduğu için 4 sağa kayar, boşluk 1 indisine geçer. `a[0]` = 2 > 1 olduğu için 2 de kayar ve boşluk 0 indisine gelir. `j` artık 0'dır ve döngü durur; `a[0] = 1` yazılır. Anahtar, sıralı kısmın hepsinden küçük olduğu için bütün kısım kaydırılmıştır.



Şekil 3.32: Eklemeli sıralamanın dört anı. (a) Başlangıç: 5, 2, 4, 1, 3. (b) 1. tur sonu: 2, `j` = 0'a yerleşmiştir. (c) 3. tur sonu: 1 başa yerleşmiştir; ilk dört öğe 1, 2, 4, 5'tir. (d) 4. tur sonu: 3, 2 indisine yerleşmiş ve dizi sıralanmıştır.

Toplam 1 + 1 + 3 + 2 = 7 kaydırma yapılmıştır. Bu sayı rastgele değildir; dizinin ne kadar “bozuk” olduğunu ölçen bir büyüklüğe eşittir.

Tanım 3.9

Bir dizide *i* < *j* iken *a[i]* > *a[j]* olan her *(i, j)* indis çiftine **ters sıralı çift** (*inversion*) denir. Sıralı dizide ters sıralı çift yoktur. Tersten sıralı *n* öğeli dizide her çift ters sıralıdır ve sayıları *n(n − 1)/2*'dir.

{5, 2, 4, 1, 3} dizisinin ters sıralı çiftleri (değerleriyle) şunlardır: (5, 2), (5, 4), (5, 1), (5, 3), (2, 1), (4, 1), (4, 3). Toplam 7 çift vardır; kaydırma sayısı da 7'dir.

Teorem 3.3 (Eklemeli sıralamada kaydırma sayısı)

Eklemeli sıralamanın yaptığı toplam kaydırma sayısı, dizideki ters sıralı çiftlerin sayısına eşittir. Bu yüzden algoritma zaten sıralı dizide $O(n)$, tersten sıralı dizide $O(n^2)$ sürer.

İspat. Her kaydırma, sıralı kısımda anahtardan büyük bir öğeyi anahtarın sağına geçirir. Kaydırılan öğe dizide anahtarın solundaydı ve anahtardan büyüktü; yani ikisi bir ters sıralı çift oluşturuyordu. Kaydırmadan sonra bu çift düzelmiştir ve bir daha bozulmaz, çünkü algoritma iki öğenin göreli sırasını bir daha değiştirmez. Ayrıca anahtardan büyük her soldaki öğe kaydırılır; anahtarın dahil olduğu, solunda kalan hiçbir ters sıralı çift atlanmaz. Böylece her kaydırma tam olarak bir ters sıralı çifti giderir ve her çift tam olarak bir kaydırmayla giderilir. Karşılaştırma sayısı her turda kaydırma sayısından en çok bir fazladır; toplam süre $O(n + \text{ters sıralı çift sayısı})$'dır. Sıralı dizide bu $O(n)$, tersten sıralı dizide $O(n + n(n-1)/2) = O(n^2)$'dir. □

Teorem, eklemeli sıralamanın neden "neredeyse sıralı" dizilerde çok hızlı olduğunu açıklar. Az sayıda öğesi yerinden oynamış bir dizide ters sıralı çift az olur ve algoritma doğrusala yakın sürede biter. Seçimli sıralama ise sıralı dizide bile $n(n-1)/2$ karşılaştırma yapar. TimSort gibi gerçek kütüphane algoritmaları bu yüzden küçük parçaları eklemeli sıralamayla sıralar. Eklemeli sıralama ayrıca kararlıdır: `a[j - 1] > anahtar` koşulu eşit öğede durur ve eşit öğelerin sırası korunur.

Sık yapılan hata

`while` koşulundaki iki parçanın sırasını değiştirmek: `a[j - 1] > anahtar && j > 0`. `j` 0 olduğunda ilk parça `a[-1]`'e erişmeye çalışır ve istisna fırlatılır. `&&` soldan sağa değerlendirildiği için sınır denetimi (`j > 0`) önce yazılmalıdır; o yanlışsa dizi erişimine hiç geçilmez.

3.5.10 Alan karmaşıklığı ve bellek kullanımı

Alan karmaşıklığı (*space complexity*), bir algoritmanın ya da veri yapısının bellek ihtiyacının n ile nasıl büyüdüğünü anlatır. Zaman karmaşıklığıyla aynı gösterim kullanılır. n öğeli bir dizi $O(n)$ alan kaplar. Bayt olarak miktar ise öğe tipine bağlıdır.

Dizi tipi	Öğe başına	100 öğe
`int[]`	4 bayt	400 bayt
`char[]` (UTF-16)	2 bayt	200 bayt
`double[]`, `long[]`	8 bayt	800 bayt
`Object[]` (referans)	4 / 8 bayt	400 / 800 bayt + nesneler

Tablo 3.28: Dizi tiplerine göre bellek kullanımı (yalnız veri kısmı).

Tablonun iki ayrıntısı açıklama ister. Java'da `char`, UTF-16 kodlamasında bir kod birimidir ve 2 bayt kaplar; ASCII karakterler için de böyledir. Referansın 4 mü 8 bayt mı olduğu JVM'e ve ayarlarına bağlıdır. 64 bitlik JVM'ler çoğu zaman referansları sıkıştırıp 4 baytta tutar.

Veri kısmına ek olarak iki maliyet daha vardır:

- **Ek yük.** Her dizi nesnesinin ayrıca yaklaşık 16 baytlık bir başlığı vardır. Başlıkta nesnenin türü ve dizinin `length` değeri gibi bilgiler tutulur. Büyük dizilerde bu yük önemsizdir; çok sayıda küçük dizide ise toplamı fark edilir.
- **Referans dizileri.** `String[]` gibi nesne dizilerinde yalnız referanslar ardışıktır. Gösterilen nesneler heap bellekte ayrı ayrı yer kaplar ve kendi başlıklarını taşır. 100 öğeli bir `String[]`'in 400 baytı yalnız referanslardır; metinlerin kendisi bunun üstüne eklenir.

Örnek 3.5

1000 × 1000'lik bir `int` matrisinin bellek ihtiyacını kabaca hesaplayalım (4 baytlık referans ve 16 baytlık başlık varsayımıyla).

1. **Veri:** 1000 · 1000 = 1.000.000 `int`, her biri 4 bayt: 4.000.000 bayt, yaklaşık 4 MB.
2. **Satır başlıkları:** Her satır ayrı bir dizidir: 1000 · 16 = 16.000 bayt.
3. **Dış dizi:** 1000 referans · 4 bayt + 16 bayt başlık = 4016 bayt.

Toplam yaklaşık 4.020.016 bayttır. Ek yükler toplamın binde 5'inden azdır; matrisin bellek ihtiyacını belirleyen, veri kısmıdır.

Algoritmaların alan maliyeti de aynı biçimde düşünülür. Girdinin kendisi sayılmaz; algoritmanın ayrıca ayırdığı bellek hesaplanır. Bu bölümdeki algoritmalardan doğrusal arama, ortalama, yerinde tersine çevirme, ekleme, silme, seçimli ve eklemeli sıralama $O(1)$ ek bellek kullanır. Ortanca ve mod, girdiyi kopyaladıkları için $O(n)$; transpoz yeni bir matris oluşturduğu için $O(m \cdot n)$ ek bellek kullanır. Zaman ile bellek arasında çoğu zaman bir değiş tokuş vardır: daha çok bellek kullanmak bir algoritmayı hızlandırabilir, belleği kısmak yavaşlatabilir.

3.6 Sık yapılan hatalar

Dizilerle ilgili hataların çoğu birkaç kaynaktan çıkar: indisin 0'dan başlaması, dizinin bir nesne olması ve `=` ile `==`'nin referanslar üzerinde çalışması. Tablo 3.29 notun farklı yerlerinde ele alınan hataları bir arada verir.

Hata	Ne olur?	Doğrusu
Sınır hatası	`<=` son turda `a[a.length]`'e erişir → istisna.	`for (i = 0; i < a.length; i++)`

Hata	Ne olur?	Doğrusu
`length` / `length()`	Dizide alan, `String`'de metot; karıştırılırsa derleme hatası.	`a.length` · `s.length()`
`=` **kopya değildir**	`b = a` aynı diziyi gösterir; `b`'deki değişiklik `a`'yı da değiştirir.	`int[] b = a.clone();`
`==` **ile içerik**	`==` referansları karşılaştırır; aynı içerikli iki dizi için `false`.	`Arrays.equals(a, b)`
Diziyi doğrudan yazdırmak	`println(a)` tür ve sayı basar (`[I@…`).	`Arrays.toString(a)`
Düzensiz dizide `null` **satır**	`new int[3][]` sonrası `m[0][0]` → `NullPointerException`.	`m[i] = new int[k];`
Sırasız dizide ikili arama	`binarySearch` hata vermez ama yanlış sonuç verebilir.	önce `Arrays.sort(a)`
2B dizide sütun sayısı	Her satır için `m[0].length` kullanmak düzensiz dizide yanlıştır.	`m[i].length`
Matrisin sığ kopyası	`m.clone()` satırları paylaşır; hücre değişikliği orijinale yansır.	`d[i] = m[i].clone();`
Değerle takas	`yerDegistir(a[i], a[j])` yalnız kopyaları değiştirir.	`yerDegistir(a, i, j)`

Tablo 3.29: Dizilerle sık yapılan hatalar ve doğru yazımları.

Tablodaki hataların ortak bir dersi vardır: bir dizi değişkeni, dizinin kendisi değil, ona giden bir oktur. `=` oku kopyalar, `==` okları karşılaştırır, `println` okun kendisini yazar. Dizinin içeriğiyle çalışmak isteyen her işlem ya indislerle hücrelere tek tek erişir ya da bunu yapan bir `Arrays` metodu kullanır.

3.7 Özet

- **Dizi**, aynı türden sabit sayıda öğeyi bellekte **ardışık** yuvalarda tutar. İndisler 0'dan `length - 1`'e kadardır. *i*. öğenin adresi *başlangıç + i × öğe boyutu* ile tek adımda hesaplanır; bu yüzden indisle erişim *O(1)*'dir.
- Dizinin üstünlükleri hızlı erişim, ek referans yükünün olmaması ve önbellek dostu olmasıdır. Zayıflıkları sabit boyut, araya ekleme ve silmede kaydırma (*O(n)*) ve boşa giden kapasitedir. Java dizileri **kovaryanttır**; yanlış türde atama ancak çalışırken `ArrayStoreException` ile yakalanır.
- Java'da dizi bir **nesnedir** ve **heap bellekte** durur; stack bellekteki değişken yalnız referansını tutar. `new` ile oluşturulan dizinin öğeleri varsayılan değerleri alır (0, 0.0, `false`, `'\u0000'`, `null`). `length` bir alandır.
- `=` referansı kopyalar; gerçek kopya için `clone()`, `Arrays.copyOf` ya da `System.arraycopy` kullanılır (*O(n)*). `==` referansları karşılaştırır; içerik için `Arrays.equals`, yazdırmak için `Arrays.toString` gerekir.
- Erişilemez hâle gelen dizileri **çöp toplayıcı** toplar. Sınır dışı erişim `ArrayIndexOutOfBoundsException` fırlatır; Java her erişimi denetler, C denetlemez.

- **Ortalama** tek geçişte *O(n)* zaman ve *O(1)* ek bellekle hesaplanır. **Ortanca** ve **mod**, dizinin sıralı bir kopyası üzerinde bulunur; maliyetleri sıralamanın getirdiği *O(n log n)*'dir. Kopya, çağıranın dizisini korur.
- Diziyi **yerinde tersine çevirme** iki uçtan ortaya ilerleyen işaretçilerle *⌊n/2⌋* takas yapar: *O(n)* zaman, *O(1)* ek bellek. **Sezar şifresi**, `char` değerlerinin tam sayı olmasından ve `% 26` ile başa sarmadan yararlanır.
- Java'da matris **dizilerin dizisidir**: `m.length` satır, `m[i].length` o satırın sütun sayısıdır. Satırlar farklı uzunlukta olabilir (**düzensiz dizi**). `m.clone()` **sığ kopya** verir; **derin kopya** için her satır ayrıca kopyalanır.
- *m × n*'lik bir matrisi gezmek, doldurmak ya da **transpozunu** almak *O(m · n)*'dir. İç içe döngülerde gövde, döngü sayılarının çarpımı kadar çalışır.
- **Doğrusal arama** en kötü durumda *O(n)*, ikili arama sıralı dizide *O(log n)*'dir. İki dizide ortak öğeyi iç içe döngüyle bulmak *O(n · m)*'dir.
- **Araya ekleme** sağdan sola, **aradan silme** soldan sağa kaydırır; ikisi de en kötü durumda *O(n)*, sonda ise *O(1)*'dir.
- **Seçimli sıralama** her girdide *n(n − 1)/2* karşılaştırma yapar. **Eklemeli sıralama**, ters sıralı çift sayısı kadar kaydırma yapar: sıralı dizide *O(n)*, en kötü durumda *O(n^2)*. İkisi de *O(1)* ek bellek kullanır.

İşlem	Zaman	Ek bellek
Erişim / güncelleme `a[i]`	*O(1)*	*O(1)*
Kopyalama (`clone` **,** `copyOf` **)**	*O(n)*	*O(n)*
Ortalama	*O(n)*	*O(1)*
Ortanca, mod	*O(n log n)*	*O(n)*
Yerinde tersine çevirme	*O(n)*	*O(1)*
Doğrusal arama	*O(n)*	*O(1)*
İkili arama (sıralı dizi)	*O(log n)*	*O(1)*
Araya ekleme / aradan silme	*O(n)*	*O(1)*
Seçimli sıralama	*O(n^2)*	*O(1)*
Eklemeli sıralama	*O(n)* … *O(n^2)*	*O(1)*
Transpoz (*m × n*)	*O(m · n)*	*O(m · n)*

Tablo 3.30: Bölümdeki işlemlerin maliyetleri.

3.8 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

dizi *array* · öğe *element* · indis *index* · ilklendirici *initializer* · kapasite *capacity*

statik dizi *static array* · dinamik dizi *dynamic array* · referans *reference* · kovaryans *covariance* · heap bellek *heap memory* · yığın çerçevesi *stack frame* · iki boyutlu dizi *two-dimensional array* · çok boyutlu dizi *multidimensional array* · matris *matrix* · düzensiz dizi *jagged array* · sığ kopya *shallow copy* · derin kopya *deep copy* · transpoz *transpose* · ortanca *median* · mod *mode* · yerinde *in-place* · takas *swap* · doğrusal arama *linear search* · ikili arama *binary search* · kabarcık sıralaması *bubble sort* · seçimli sıralama *selection sort* · eklemeli sıralama *insertion sort* · hızlı sıralama *quick sort* · birleştirme sıralaması *merge sort* · kararlı *stable (sort)* · zaman karmaşıklığı *time complexity* · alan karmaşıklığı *space complexity*

3.9 Kendini Dene

1. `int[] a = {4, 7, 1}; int[] b = a; a = new int[] {0}; b[1] = 9;` satırlarından sonra `a` ve `b` hangi dizileri gösterir?

   Cevap

   `a` = [0], `b` = [4, 9, 1]. `b = a` ile iki değişken ilk diziyi gösterir. Sonra `a`'ya yeni bir dizi atanır; bu yalnız `a`'nın referansını değiştirir, `b` hâlâ ilk diziyi gösterir. `b[1] = 9` o diziyi değiştirir. Yeni dizi bu değişiklikten etkilenmez.

2. `Arrays.toString(new double[3])`, `Arrays.toString(new boolean[2])` ve `Arrays.toString(new String[2])` ne yazar?

   Cevap

   `[0.0, 0.0, 0.0]`, `[false, false]` ve `[null, null]`. `new` ile oluşturulan dizinin öğeleri türün varsayılan değerini alır; `String`'in varsayılanı boş metin değil, `null`'dır.

3. {10, 2, 7, 3, 100, 1} dizisinin ortalaması ve ortancası nedir? Hangisi verinin “ortasını” daha iyi anlatır?

   Cevap

   Toplam 123, ortalama 123 / 6 = 20,5'tir. Sıralı dizi {1, 2, 3, 7, 10, 100}'dür; boyut çift olduğu için 2 ve 3 indislerindeki 3 ile 7'nin ortalaması alınır: ortanca 5,0'tır. Altı değerden beşi 10 ya da altındadır; 100 değeri ortalamayı yukarı çekmiştir. Ortanca uç değerden etkilenmediği için ortayı daha iyi anlatır.

4. `mod({8, 3, 8, 5, 3, 8})` ne döndürür? Döngü sırasında rekor kaç kez güncellenir?

Cevap

Sıralı kopya {3, 3, 5, 8, 8, 8}'dir. `i = 1`'de 3 ikinci kez görülür ve rekor 3 (2 kez) olur. `i = 2` ve `i = 3`'te yeni değerler başlar, sayaç 1'dir. `i = 4`'te 8 iki kez görülmüştür, ama `2 > 2` yanlıştır. `i = 5`'te 8 üç kez görülür ve rekor 8 (3 kez) olur. Metot 8 döndürür; rekor döngüde iki kez değişir.

5. `sifrele("JAVA", 5)` ne döndürür? Şifreyi çözmek için aynı metot hangi *k* ile çağrılmalıdır?

Cevap

J: 9 + 5 = 14 → O. A: 0 + 5 = 5 → F. V: 21 + 5 = 26, 26 % 26 = 0 → A. A → F. Sonuç `"OFAF"`'tır. Çözmek için *k = 26 − 5 = 21* kullanılır: `sifrele("OFAF", 21)` → `"JAVA"`.

6. Pascal üçgeni kodu `n = 6` ile çalıştırılırsa `p[5]` satırı nedir? Üçgen toplam kaç hücre kullanır?

Cevap

Satırlar 1; 1 1; 1 2 1; 1 3 3 1; 1 4 6 4 1; 1 5 10 10 5 1'dir. `p[5]` = [1, 5, 10, 10, 5, 1]. Örneğin `p[5][2] = p[4][1] + p[4][2] = 4 + 6 = 10`'dur. Hücre sayısı 1 + 2 + … + 6 = 21'dir; 6 × 6'lık kare matris 36 hücre kullanırdı.

7. 3 × 3 × 3'lük diziye `x * 9 + y * 3 + z + 1` yazılmıştır. 20 değeri hangi hücrededir?

Cevap

20 − 1 = 19 = 2 · 9 + 0 · 3 + 1 olduğu için `x` = 2, `y` = 0, `z` = 1'dir: `ucBoyutluDizi[2][0][1]`. Yani 20, üçüncü katmanın ilk satırının ikinci hücresindedir.

8. `a` = {5, 6, 8, 9, 0} ve `boyut` = 4 iken `ekle(a, 4, 0, 7)` çağrılıyor. Kaç kaydırma yapılır ve dizi ne olur? Ardından `sil(a, 5, 4)` çağrılırsa ne olur?

Cevap

Başa ekleme dört öğenin hepsini kaydırır: `k` = 4, 3, 2, 1 için 4 kaydırma. Dizi {7, 5, 6, 8, 9} olur ve metot 5 döndürür. `sil(a, 5, 4)` son öğeyi siler: döngü koşulu `4 < 4` baştan yanlıştır, kaydırma yapılmaz. `a[4] = 0` yazılır; dizi {7, 5, 6, 8, 0} olur ve metot 4 döndürür. Başa ekleme *O(n)*, sondan silme *O(1)*'dir.

9. {4, 3, 2, 1} dizisi önce seçimli, sonra (aynı ilk diziyle) eklemeli sıralamayla sıralanıyor. Seçimli sıra-

lama kaç karşılaştırma, eklemeli sıralama kaç kaydırma yapar? {1, 2, 3, 4} için cevaplar ne olur?

Cevap

Seçimli sıralama her girdide $n(n-1)/2 = 6$ karşılaştırma yapar; iki dizide de 6'dır. Eklemeli sıralamada kaydırma sayısı ters sıralı çift sayısıdır. {4, 3, 2, 1}'de bütün çiftler ters sıralıdır: 6 kaydırma (1 + 2 + 3). {1, 2, 3, 4}'te ters sıralı çift yoktur: 0 kaydırma. Bu durumda her tur tek bir karşılaştırmayla biter ve toplam 3 karşılaştırma yapılır.

10. `new int[1000][1000]` ile oluşturulan matrisin veri kısmı kaç bayttır? Bu matriste bir değeri bütün hücreleri tek tek gezerek aramak en kötü durumda kaç karşılaştırma gerektirir?

Cevap

Veri kısmı 1.000.000 × 4 = 4.000.000 bayttır (yaklaşık 4 MB); bunun üstüne 1000 satır dizisinin ve dış dizinin başlıkları ile dış dizideki 1000 referans eklenir. Matrisin bütün hücrelerini gezen bir arama, değer yoksa 1.000.000 karşılaştırma yapar: $r \times c$'lik matriste $O(r \cdot c)$.

3.10 İleri okuma

- M. T. Goodrich, R. Tamassia ve M. H. Goldwasser, *Data Structures and Algorithms in Java* (6. baskı, 2014), Kesim 3.1: dizilerde sıralama, `java.util.Arrays`, karakter dizileriyle basit şifreleme ve iki boyutlu dizilerle tahta oyunları gibi bu nottaki konuları Java koduyla işler.
- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 1.1 ve 2.1: Java'da dizilerin kullanımını, referans ile kopya ayrımını ve seçimli ile eklemeli sıralamanın karşılaştırmalı analizini açık örneklerle verir.
- J. Gosling ve öteki yazarlar, *The Java Language Specification, Java SE 17 Edition*, Bölüm 10: dizilerin tanımlanmasını, ilklendirilmesini, varsayılan değerleri, `clone`'un sığ kopya davranışını ve `ArrayStoreException`'ı dilin resmî tanımıyla anlatır.

3.11 Kaynaklar

1. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6. baskı, Wiley, 2014.
2. J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, G. Bierman, *The Java Language Specification, Java SE 17 Edition*, Oracle, 2021.
3. R. Lafore, *Data Structures and Algorithms in Java*, 2. baskı, Sams, 2002.
4. Oracle, *Java SE 17 API Specification*: `java.util.Arrays` ve `java.lang.System` belgeleri.
5. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011.

KISIM I · BÖLÜM 4

Özyineleme

Özyineleme, bir problemi aynı problemin daha küçük bir örneğine indirgeyerek çözme yöntemidir. Java'da bunun karşılığı, kendini daha küçük bir girdiyle çağıran metottur. Yöntem ilk karşılaşmada tuhaf gelir: henüz yazılmakta olan bir metot nasıl kendini kullanabilir? Cevap iki parçalıdır. En küçük girdide metot kimseye sormadan cevabı verir. Daha büyük girdilerde ise küçük problemin cevabını alıp tek bir adımla büyüğünkini kurar. Not bu fikirle başlar; çağrıların bellekte çağrı yığınıyla nasıl yürütüldüğünü adım adım izler. Sonra özyinelemenin türleri gelir. Doğrusal özyineleme dizi toplamı, tersine çevirme ve hızlı üs almayla; ikili özyineleme Fibonacci ve ikili aramayla işlenir. Hanoi kuleleri ile geri izlemeyle alt küme ve permütasyon üretme bu türleri tamamlar. Son kesimler maliyet üzerinedir: yineleme bağıntıları, çağrı ağacı, alan maliyeti, kuyruk özyinelemesi ve özyinelemenin döngüye çevrilmesi. Bağlı listede özyineleme, konuyu sonraki veri yapılarına bağlar.

Bu bölümün kodu: `github.com/sercankulcu/data-structures-java/Bolum04/src`

ÖĞRENME ÇIKTILARI

Özyinelemeli tasarım
Bir problemi taban durum ve özyinelemeli adımla tanımlamak ve bu tanımı Java'da doğru bir metoda dökmek.

Çağrı yığını
Özyinelemeli bir metodu çerçeve çerçeve izleyip çıktısını ve dönüş değerini bulmak; yığının en çok kaç çerçeveye ulaştığını söylemek.

Özyineleme türleri
Doğrusal, ikili ve çoklu özyinelemeyi ve geri izlemeyi örnekleriyle ayırt etmek.

Maliyet
Yineleme bağıntısını yazıp çağrı ağacıyla zaman ve alan maliyetini bulmak; gerektiğinde özyinelemeli metodu döngüye çevirmek.

4.1 Özyineleme nedir?

Bir metot başka metotları çağırabilir; bunu programlamaya başlarken öğreniriz. `main`, `System.out.println`'i çağırır; bir sıralama metodu, iki öğenin yerini değiştiren yardımcı bir metodu çağırır. Özyinelemede çağrılan metot, çağıranın kendisidir. Böyle bir metot neden sonsuza kadar dönüp durmaz, nasıl tasarlanır ve satır satır nasıl çalışır? Kesimin konusu bu üç sorudur.

4.1.1 Günlük hayattan bir örnek

Bir sinema gişesinin önünde uzun bir kuyruk var. Kuyrukta kaçıncı sırada olduğumuzu öğrenmek istiyoruz. Yerimizi bırakıp en öne yürüyemiyoruz; yürürsek sıramızı kaybederiz. Görebildiğimiz tek kişi hemen önümüzdeki kişidir.

Bu durumda şöyle bir yol izlenebilir. Önümüzdekine "Sen kaçıncısın?" diye sorarız. O da kendi sırasını bilmiyorsa aynı soruyu kendi önündekine sorar. Soru böylece kişiden kişiye öne doğru ilerler. En öndeki kişinin önünde kimse yoktur; o hiç düşünmeden "1" der. Cevap bu kez geriye doğru yol alır. En öndekinin arkasındaki kişi "önümdeki 1. ise ben 2.'yim" der ve bunu arkasına iletir. Herkes duyduğu sayıya 1 ekleyip arkadakine söyler. Cevap bize ulaştığında sıramızı öğrenmiş oluruz.



Şekil 4.1: Kuyrukta sırayı öğrenmek. Mavi oklar soruyu öne taşır: her kişi önündekine "Sen kaçıncısın?" diye sorar. En öndeki kişi (turuncu) kimseye sormadan "1" der. Yeşil oklar cevabı geri taşır; her kişi duyduğu sayıya 1 ekler: 1, 2, 3, 4, 5. Soruyu ilk soran kişi ("ben") 5. sıradadır.

Şekil 4.1'deki süreçte üç ayrı parça vardır:

- **Aynı soru, daha kısa kuyruk.** Her kişi önündekine aynı soruyu sorar. Ama soru her seferinde bir kişi daha kısa bir kuyruk için sorulur. Kimse farklı bir iş yapmaz; herkes aynı kuralı uygular.
- **Durma noktası.** En öndeki kişi soruyu başkasına aktarmadan cevaplar. Sorular burada durur. Böyle bir kişi olmasaydı soru sonsuza kadar öne aktarılırdı.
- **Geri dönüş.** Cevap geri gelirken her kişi kendi payını, yani 1'i ekler. Asıl hesap bu dönüş yolunda yapılır.

Özyinelemeli bir metot da tam böyle çalışır. Kendini daha küçük bir girdiyle çağırır. En küçük girdide cevabı doğrudan verir. Dönüşte küçük problemin cevabını kullanarak kendi cevabını kurar. Kuyruktaki her kişi, metodun bir çağrısına karşılık gelir. Her kişinin kendi aklında tuttuğu bir bilgi

vardır (“arkamdakine cevap vereceğim”); metot çağrılarında da her çağrının kendi değişkenleri vardır.

4.1.2 Taban durum ve özyinelemeli adım

Özyinelemenin en bilinen örneği faktöriyeldir. *n!*, 1'den *n*'ye kadar tam sayıların çarpımıdır: $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$. Bu tanımda bir düzen göze çarpar. $4!$'ün içinde $3 \cdot 2 \cdot 1 = 3!$ zaten vardır. Dolayısıyla $4! = 4 \cdot 3!$ yazılabilir. Aynı gözlem her $n > 1$ için geçerlidir. Faktöriyel bu yüzden kendisi cinsinden tanımlanabilir:

> $n! = 1$ ($n \le 1$ ise) · $n! = n \cdot (n-1)!$ ($n > 1$ ise)

Tanımın iki satırı iki farklı iş görür. İlk satır, en küçük girdiler için cevabı başka hiçbir şeye başvurmadan verir. İkinci satır, büyük bir girdinin cevabını bir küçüğünün cevabından kurar. $4!$'ü bulmak için ikinci satır üç kez uygulanır, sonra birinci satır devreye girer: $4! = 4 \cdot 3! = 4 \cdot 3 \cdot 2! = 4 \cdot 3 \cdot 2 \cdot 1! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$.

Tanım 4.1

Özyineleme (*recursion*), bir metodun kendini daha küçük bir girdiyle çağırmasıdır. Özyinelemeli bir metot iki parçadan oluşur:

- **Taban durum** (*base case*): kendini çağırmadan, doğrudan çözülen girdiler. Özyineleme burada durur.
- **Özyinelemeli adım** (*recursive step*): problemi küçültüp taban duruma yaklaştıran ve küçük problemin cevabından büyüğünkini kuran kısım.

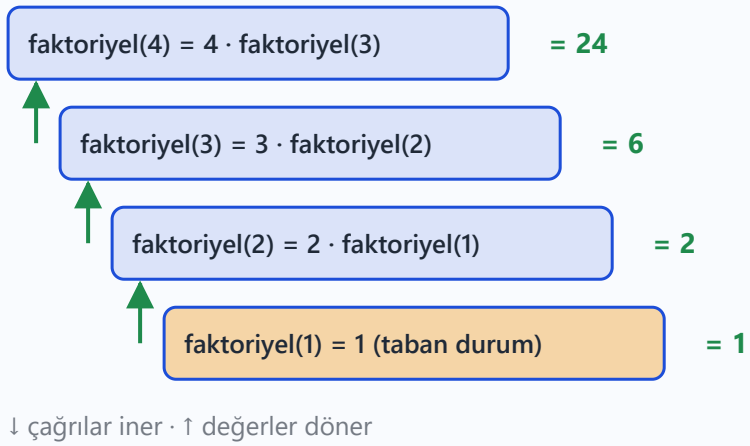
Faktöriyel tanımı Java'ya neredeyse kelimesi kelimesine çevrilir:

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

// main içinde:
System.out.println(faktoriyel(4));
```

İkinci satırdaki `if (n <= 1)` koşulu taban durumu yakalar ve 1 döndürür. Son `return` satırı özyinelemeli adımdır: `faktoriyel(n - 1)` çağrısı daha küçük bir girdiyle aynı metodu çalıştırır, dönen değer `n` ile çarpılır. `main` içindeki satır `faktoriyel(4)`'ü çağırır ve 24 yazdırır.

Taban durum neden bu kadar önemlidir? Metot her çağrıda kendini yeniden çağırıyorsa bir yerde bu zincirin kopması gerekir. Taban durum o kopma noktasıdır. Taban durum olmazsa çağrılar hiç bitmez. Her çağrı bellekte biraz yer tuttuğu için bellek bir noktada dolar ve Java programı `StackOverflowError` hatasıyla durdurur. Bu hatanın nereden geldiği 4.2.3'te ayrıntılı anlatılır.



Şekil 4.2: `faktoriyel(4)` 'ün açılımı. Çağrılar yukarıdan aşağı iner: her kutu bir küçük girdiyle çağrı yapar. En alttaki `faktoriyel(1)` taban durumdur ve 1 döndürür (turuncu). Değerler yeşil oklarla yukarı döner: 1, 2, 6, 24.

Şekil 4.2 özyinelemenin iki yönlü hareketini gösterir. İniş sırasında hiçbir çarpma yapılmaz; her çağrı yalnızca kendi `n` 'sini aklında tutup bir alttakini bekler. Çarpmalar çıkış sırasında yapılır: önce *2 · 1 = 2*, sonra *3 · 2 = 6*, en son *4 · 6 = 24*.

Bir ayrıntıyı atlamamak gerekir. Aynı anda dört ayrı `faktoriyel` çağrısı bellekte durur ve her birinin kendi `n` değişkeni vardır: 4, 3, 2 ve 1. Bu dört `n` birbirine karışmaz. `faktoriyel(1)` çalışırken `n = 1` 'dir; ama `faktoriyel(4)` 'ün `n` 'si hâlâ 4'tür ve sırası gelince 4 ile çarpma yapılır. Her çağrı bellekte kendine ait bir alan açar. Bu alana **çerçeve** denir; çerçeveler 4.2'nin konusudur.

Sık yapılan hata

Özyinelemeli metodu, aynı değişkenleri paylaşan bir döngü gibi düşünmek. "`faktoriyel(1)` çağrılınca `n` 1 oldu, artık `4 * ...` satırında da `n` 1'dir" gibi bir akıl yürütme yanlıştır. Parametreler ve yerel değişkenler her çağrıda yeniden oluşturulur. Bir çağrının değişkenini değiştirmek, öteki çağrıların değişkenlerini etkilemez.

4.1.3 Özyinelemeli düşünmek

Özyinelemeli bir metot yazarken bütün çağrı zincirini kafada canlandırmaya çalışmak yorucudur ve çoğu zaman gereksizdir. Bunun yerine üç soru sorulur:

1. **Taban durum:** En küçük girdi hangisidir ve cevabı nedir? Özyineleme burada durur.

2. **Küçültme:** Problem, aynı türden daha küçük bir probleme nasıl indirgenir? Küçültme çoğunlukla *n − 1*, *n / 2* ya da bir indisi *i + 1* yapmak biçimindedir.
3. **Birleştirme:** Küçük problemin cevabından büyük problemin cevabı nasıl kurulur?

Üçüncü soru cevaplanırken işimizi kolaylaştıran bir kabul vardır: küçük problemin doğru çözüldüğü *kabul edilir*. `faktoriyel(n)`'i yazarken `faktoriyel(n - 1)`'in nasıl çalıştığı düşünülmez; onun *(n − 1)!* değerini doğru döndürdüğü varsayılır ve yalnızca son adım düşünülür: "*(n − 1)!* elimdeyse *n!*'yi nasıl bulurum? *n* ile çarparım."

Bu varsayım bir hile değildir; matematikteki **tümevarım** ilkesinin aynısıdır. Tümevarımla bir önermenin her doğal sayı için doğru olduğunu göstermek için iki şey yeterlidir. Önerme en küçük sayı için doğrudur. Önerme *n − 1* için doğruysa *n* için de doğrudur. Bu ikisi gösterilince önerme 1 için, dolayısıyla 2 için, dolayısıyla 3 için doğru olur ve zincir hiç kopmaz. Özyinelemeli metotta da aynıdır: taban durum doğru cevabı veriyorsa ve her adım, küçük cevap doğruysa büyük cevabı doğru kuruyorsa, metot her girdi için doğru çalışır.

Bir koşul daha vardır: her özyinelemeli çağrı girdiyi taban duruma *yaklaştırmalıdır*. Tümevarım zinciri ancak her adım bir öncekine bağlanıyorsa işler. Girdi küçülmüyorsa ya da taban durumu atlayıp öteye geçiyorsa zincirin başı hiç gelmez.

Bu üç soruyu basit bir örnekle cevaplayalım. `geriSay(n)` metodu, *n*'den 1'e kadar sayıları birer satıra yazsın, en sonda da "Bitti" yazsın.

Örnek 4.1

1. **Taban durum.** Yazılacak sayı kalmadıysa, yani *n = 0* ise, yalnızca "Bitti" yazılır ve metot biter.
2. **Küçültme.** *n*'den 1'e geri saymak, önce *n*'yi yazıp sonra *n − 1*'den 1'e geri saymaktır. Küçük problem `geriSay(n - 1)`'dir.
3. **Birleştirme.** Burada bir değer döndürülmez; iş, yazdırma sırasını doğru kurmaktır. Önce *n* yazılır, sonra küçük problem çözülür.

Her çağrıda *n* 1 azalır. Pozitif bir *n* ile başlanırsa *n* sonlu adımda 0'a ulaşır ve özyineleme durur.

Bu üç cevap doğrudan koda dönüşür:

```
void geriSay(int n) {
  if (n == 0) {     // taban durum
    System.out.println("Bitti");
    return;
  }
  System.out.println(n);
  geriSay(n - 1);   // özyinelemeli adım
}
geriSay(3);         // main içinde
```

Kodun yapısı, tasarımdaki üç parçayı tek tek yansıtır. 2–5. satırlar taban durumdur: `n == 0` ise "Bitti" yazılır ve `return` ile metottan çıkılır. `return` burada önemlidir; olmasaydı akış 6. satıra geçer, 0 yazdırılır ve `geriSay(-1)` çağrılırdı. 6. satır *n*'yi yazar. 7. satır özyinelemeli adımdır ve daha küçük problemi çözer.

Sık yapılan hata

Taban durumu yalnızca "normal" girdiler için düşünmek. `geriSay`'e negatif bir sayı verilirse *n* −1, −2, −3 diye azalır ve 0'a hiç ulaşmaz. Metodun yalnızca negatif olmayan girdilerle çağrılacağı biliniyorsa bu kabul edilebilir; bilinmiyorsa taban durumu `n <= 0` diye yazmak daha güvenlidir.

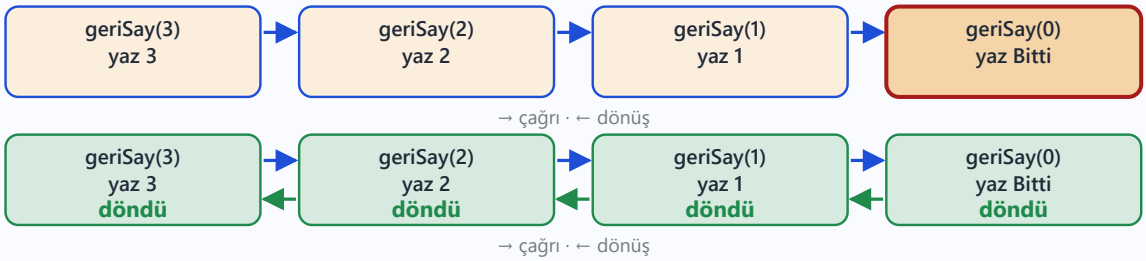
4.1.4 Kod izleme: geriSay(3)

Özyinelemeyi anlamanın en sağlam yolu, küçük bir girdiyle kodu satır satır izlemektir. `main`, `geriSay(3)`'ü çağırıyor. İzlemede her adımda hangi çağrının çalıştığı, hangilerinin beklediği ve ekrana ne yazıldığı not edilir.

Adım	Olay	Çağrılar	Ekran
1	`main`, `geriSay(3)`'ü çağırır	main	
2	*n* = 3 ≠ 0: 3 yazılır, `geriSay(2)` çağrılır	main, gS(3)	3
3	*n* = 2 ≠ 0: 2 yazılır, `geriSay(1)` çağrılır	main, gS(3), gS(2)	3 2
4	*n* = 1 ≠ 0: 1 yazılır, `geriSay(0)` çağrılır	main, gS(3), gS(2), gS(1)	3 2 1
5	*n* = 0: taban durum, "Bitti" yazılır	main, gS(3), gS(2), gS(1), gS(0)	3 2 1 Bitti
6	`geriSay(0)` `return` ile biter	main, gS(3), gS(2), gS(1)	3 2 1 Bitti
7	`geriSay(1)` 7. satırdaki çağrıdan döner; kalan iş yok, biter	main, gS(3), gS(2)	3 2 1 Bitti
8	`geriSay(2)` döner ve biter	main, gS(3)	3 2 1 Bitti
9	`geriSay(3)` döner ve biter; akış `main`'e geçer	main	3 2 1 Bitti

Tablo 4.1: `geriSay(3)` izlemesi. Çağrılar sütunu, o anda bellekte bulunan çağrıları dıştan içe sıralar; en sağdaki çalışan çağrıdır.

Tabloda `gS`, `geriSay`'in kısaltmasıdır. Çağrılar sütunu, özyinelemenin bellekteki izini gösterir. Sütun 2–5. adımlarda uzar, 6–9. adımlarda kısalır. En uzun hâlinde `main` ile birlikte 5 çağrı aynı anda bellektedir. Toplam 4 `geriSay` çağrısı yapılmıştır: *n = 3, 2, 1, 0*.



Şekil 4.3: `geriSay(3)` çağrı zinciri. Üstte en derin an (Tablo 4.1, 5. adım): üç çağrı bekliyor, kırmızı çerçeveli `geriSay(0)` taban durumda "Bitti" yazıyor. Altta son durum: dört çağrı da döndü (yeşil). Mavi oklar çağrıları, yeşil oklar dönüşleri gösterir.

Dönüş yolunda (6–9. adımlar) ekrana hiçbir şey yazılmaz. Her çağrı 7. satırdaki özyinelemeli çağrıdan döndüğünde metodun sonuna gelmiştir; yapacak işi kalmamıştır. Bütün yazdırmalar iniş yolunda yapılır. Bu durum, yazdırma satırının çağrıdan *önce* gelmesinin sonucudur. Satırların yeri değişirse çıktı da değişir.

4.1.5 İş çağrıdan önce mi, sonra mı?

Özyinelemeli bir metotta bir işin özyinelemeli çağrıdan önce mi, sonra mı yapıldığı sonucu doğrudan belirler. Bunu iki kısa metotla görelim. İkisi de taban durumda hiçbir şey yapmaz; tek fark yazdırma satırının yeridir.

```
void geriSay2(int n) {
  if (n == 0) return;
  System.out.print(n + " ");
  geriSay2(n - 1);     // sonra çağrı
}
```

```
void ileriSay(int n) {
  if (n == 0) return;
  ileriSay(n - 1);     // önce çağrı
  System.out.print(n + " ");
}
```

`geriSay2(3)` çıktısı `3 2 1`'dir. Yazdırma çağrıdan önce olduğu için her çağrı önce kendi sayısını yazar, sonra daha küçük problemi çözmeye geçer. Yazdırmalar çağrılar *inerken* yapılır.

`ileriSay(3)` çıktısı ise `1 2 3`'tür. Burada her çağrı önce özyinelemeli çağrıyı yapar ve onun bitmesini bekler. `ileriSay(3)`, 3'ü yazmadan önce `ileriSay(2)`'nin bitmesini bekler. `ileriSay(2)` de 2'yi yazmadan önce `ileriSay(1)`'i bekler. `ileriSay(1)`, `ileriSay(0)`'ı çağırır; o taban durumdur ve hemen döner. İlk yazdırmayı bu yüzden `ileriSay(1)` yapar: 1. Sonra `ileriSay(2)` kaldığı yerden devam eder ve 2'yi yazar. En son `ileriSay(3)` 3'ü yazar. Yazdırmalar çağrılar *dönerken* yapılır.

Örnek 4.2

`ileriSay(3)` izlemesi:

1. `ileriSay(3)` : $n \neq 0$, `ileriSay(2)` çağrılır. 3'ün yazdırılması bekletilir.
2. `ileriSay(2)` : `ileriSay(1)` çağrılır. 2'nin yazdırılması bekletilir.
3. `ileriSay(1)` : `ileriSay(0)` çağrılır. 1'in yazdırılması bekletilir.
4. `ileriSay(0)` : taban durum, hemen döner.
5. `ileriSay(1)` kaldığı yerden sürer: `1` yazar ve biter.
6. `ileriSay(2)` sürer: `2` yazar ve biter.
7. `ileriSay(3)` sürer: `3` yazar ve biter.

Çıktı: `1 2 3`. En derin anda bellekte `main` ve dört `ileriSay` çağrısı vardır.

İlk biten çağrı, en son başlayan çağrıdır. Bu yüzden çağrıdan sonra yapılan işler, çağrıların başlama sırasının tersiyle yapılır. Çağrıdan sonra kalan iş kaybolmaz: çağrının bilgileri bellekte bekler ve sırası gelince bu iş yapılır. Bu gözlem bölüm boyunca birçok kez işe yarar. Tek yönlü bir bağlı listeyi sondan başa yazdırmak (4.8.3) ve Hanoi kulelerinde ikinci yarıyı taşımak (4.5.3) bu sayede mümkün olur.

Sık yapılan hata

`ileriSay(3)`'ün çıktısını `3 2 1` sanmak. Çağrılar 3, 2, 1 sırasıyla başlar, ama yazdırma satırı çağrıdan sonra olduğu için hiçbiri çağrı dönmeden yazamaz. Bir özyinelemeli metodun çıktısı sorulduğunda önce yazdırmanın çağrıdan önce mi, sonra mı yapıldığına bakılır.

4.2 Çağrı yığını

Önceki kesimde aynı metodun birden çok çağrısının bellekte aynı anda durduğunu ve her birinin kendi değişkenlerini taşıdığını gördük. Java bu çağrıları nerede ve hangi düzenle tutar? Bu düzen anlaşılınca özyinelemeli bir metodun hem çıktısı hem de bellek maliyeti kolayca bulunur.

4.2.1 Çerçeveler ve çağrı yığını

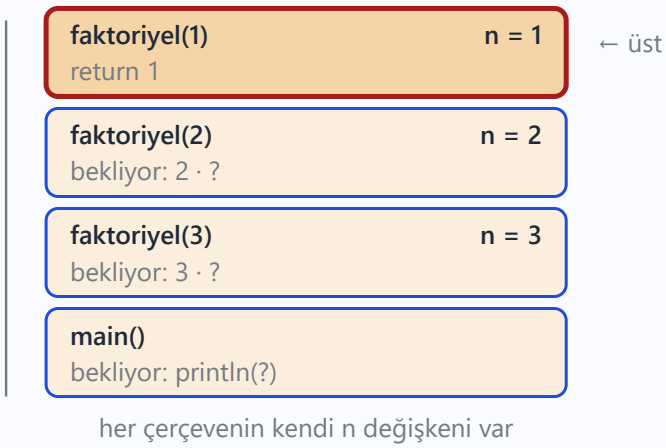
Bir metot çağrıldığında Java, o çağrı için bellekte küçük bir alan ayırır. Bu alanda üç bilgi bulunur: metodun parametreleri, metodun yerel değişkenleri ve metot bittiğinde programın nereden devam edeceği. Son bilgiye **dönüş adresi** denir. Örneğin `faktoriyel(3)`, `faktoriyel(2)`'yi çağırdığında dönüş adresi "`n * …` çarpmasının yapılacağı yer"dir.

> **Tanım 4.2**
>
> **Yığın çerçevesi** (*stack frame*) ya da kısaca çerçeve, çağrı yığınında bir metot çağrısının kaydıdır: parametreleri, yerel değişkenleri ve dönüşte devam edilecek yeri tutar. Çerçeveler **çağrı yığını** (*call stack*) denen bellek alanında saklanır. Bir çağrı yapılınca yeni çerçeve yığının üstüne eklenir; metot bitince çerçevesi üstten çıkarılır.

Çağrı yığını adını, yığın veri yapısının kuralına uymasından alır: son eklenen ilk çıkar. Bu kurala **LIFO** (*Last-In, First-Out*: son giren ilk çıkar) denir. Metot çağrılarının doğası da böyledir. `main` bir `f` metodunu, `f` de `g`'yi çağırırsa önce `g` biter, sonra `f`, en son `main`. En son başlayan çağrı ilk biter.

Çağrı yığınının üç özelliği özyinelemeyi anlamak için yeterlidir:

- **Yalnız en üstteki çerçeve çalışır.** Alttaki çerçeveler askıdadır; her biri, hemen üstündeki çağrının sonucunu bekler.
- **Aynı metodun birden çok çerçevesi aynı anda yığında bulunabilir.** Özyinelemede durum tam budur. Her çerçevenin `n`'si ayrıdır.
- **Bir çağrı bitince çerçevesi silinir.** Dönüş değeri, artık en üstte olan çağıran çerçeveye verilir ve o çerçeve dönüş adresinden devam eder.



Şekil 4.4: `faktoriyel(3)` en derindeyken çağrı yığını. Altta `main`, üstünde sırayla `faktoriyel(3)`, `faktoriyel(2)` ve `faktoriyel(1)` çerçeveleri vardır. Yalnız en üstteki (kırmızı çerçeveli) çalışır ve 1 döndürmek üzeredir. Alttakiler bekler: `faktoriyel(2)` "2 · ?" çarpmasını, `faktoriyel(3)` "3 · ?" çarpmasını, `main` ise yazdırmayı tamamlamak için. Her çerçevenin kendi `n` değişkeni vardır.

Şekil 4.4, özyinelemenin bellek maliyetini de açıklar. Yığında aynı anda dört çerçeve vardır; `faktoriyel(n)` için bu sayı *n + 1* olur (`main` dahil). Her çerçeve sabit büyüklükte yer kapladığından bellek kullanımı *n* ile doğru orantılı büyür: *O(n)*.

Bu gözlem genel bir kurala dönüşür. Özyinelemeli bir metodun bellek maliyeti, yapılan toplam çağrı sayısıyla değil, yığında *aynı anda* bulunan en çok çerçeve sayısıyla belirlenir. Bu sayı, en derin çağrı zincirinin uzunluğudur. Toplam çağrı sayısı ile en derin zincir, ikili özyinelemede birbirinden çok farklıdır; bu fark 4.6.5'te ayrıntılı işlenir.

> **Sık yapılan hata**
>
> Çağrı yığınının derinliğini sayarken `main` 'i unutmak ya da taban durum çağrısını saymamak. `faktoriyel(4)` çalışırken yığın en derin anda `main` ve `faktoriyel(4)` , `faktoriyel(3)` , `faktoriyel(2)` , `faktoriyel(1)` çerçevelerini içerir: 4 + 1 = 5 çerçeve. Soru “ `main` dahil” mi yoksa yalnız özyinelemeli çağrıları mı soruyor, buna dikkat etmek gerekir.

4.2.2 Kod izleme: faktoriyel(4)

4.1.2'deki `faktoriyel` metodunu `faktoriyel(4)` çağrısıyla, çağrı yığınının her adımdaki durumuna bakarak izleyelim. Kod şöyledir:

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

// main içinde:
System.out.println(faktoriyel(4));
```

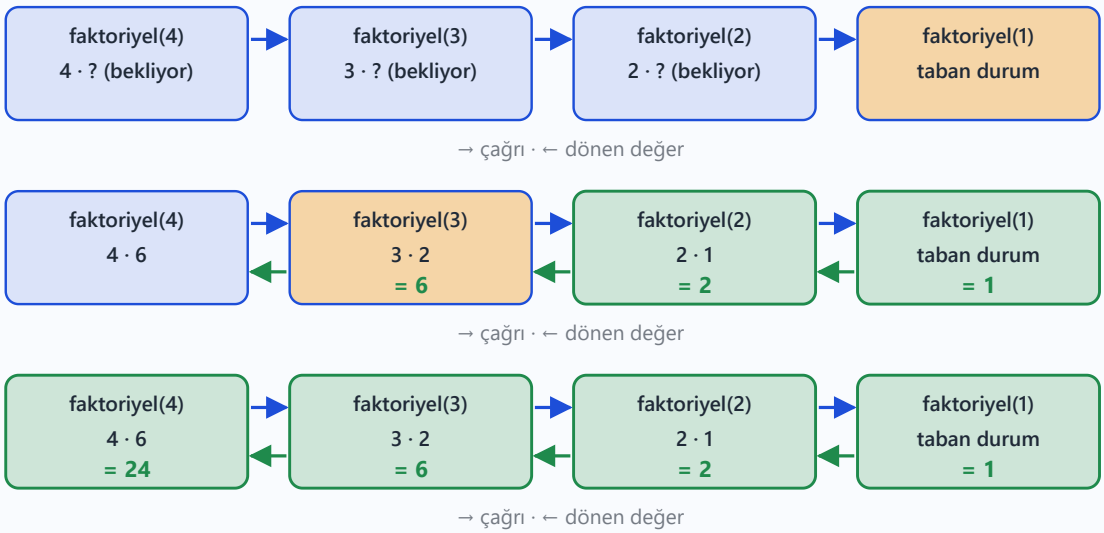
Her çağrıda yalnızca iki satır önemlidir. 2. satırdaki karşılaştırma taban durumu ayırır. 4. satır ise önce `faktoriyel(n - 1)` 'i çağırır, dönen değeri `n` ile çarpar ve sonucu döndürür. Çarpma, çağrı dönmeden yapılamaz. Çağrı yapılırken çerçeve bu çarpmayı “yarım kalmış iş” olarak saklar.

Adım	Olay	Çağrı yığını (alttan üste)	Dönen
1	`main` , `faktoriyel(4)` 'ü çağırır	main	
2	Yeni çerçeve: *n = 4*. *4 ≤ 1* mi? Hayır	main, f(4)	
3	`4 * faktoriyel(3)` : önce `faktoriyel(3)` bitmeli; bu çerçeve bekler	main, f(4)	
4	Yeni çerçeve: *n = 3*. *3 ≤ 1* mi? Hayır	main, f(4), f(3)	
5	`3 * faktoriyel(2)` : bu çerçeve bekler	main, f(4), f(3)	
6	Yeni çerçeve: *n = 2*. *2 ≤ 1* mi? Hayır	main, f(4), f(3), f(2)	
7	`2 * faktoriyel(1)` : bu çerçeve bekler	main, f(4), f(3), f(2)	

Adım	Olay	Çağrı yığını (alttan üste)	Dönen
8	Yeni çerçeve: $n = 1$. $1 \le 1$ mi? Evet	main, f(4), f(3), f(2), f(1)	
9	Taban durum: `return 1`	main, f(4), f(3), f(2), f(1)	1
10	`faktoriyel(1)` çıktı; $2 \cdot 1 = 2$, `return 2`	main, f(4), f(3), f(2)	2
11	`faktoriyel(2)` çıktı; $3 \cdot 2 = 6$, `return 6`	main, f(4), f(3)	6
12	`faktoriyel(3)` çıktı; $4 \cdot 6 = 24$, `return 24`	main, f(4)	24
13	`faktoriyel(4)` çıktı; `main` 24 yazdırır	main	

Tablo 4.2: `faktoriyel(4)` izlemesi. Çağrı yığını alttan üste yazılmıştır; `f`, `faktoriyel`'in kısaltmasıdır.

Tablodaki 2–8. adımlar **iniş** evresidir. Her adımda yığına bir çerçeve eklenir; hiçbir çarpma yapılmaz. 9. adımda taban duruma ulaşılır ve ilk somut değer, 1, üretilir. 10–13. adımlar **dönüş** evresidir. Her adımda bir çerçeve yığından çıkar ve hemen altındaki çerçeve yarım kalan çarpmasını tamamlar. Yığının bu biçimde geriye doğru boşalmasına **çözülme** (*unwinding*) de denir.



Şekil 4.5: `faktoriyel(4)` izlemesinden üç an. Üstte en derin an (Tablo 4.2, 8. adım): üç çağrı bekliyor (mavi), `faktoriyel(1)` taban durumda (turuncu). Ortada 11. adım: `faktoriyel(1)` ve `faktoriyel(2)` değerlerini döndürdü (yeşil), `faktoriyel(3)` $3 \cdot 2 = 6$ hesaplıyor. Altta son durum: `faktoriyel(4)` $4 \cdot 6 = 24$ döndürdü.

Maliyeti bu izlemeden okumak kolaydır. Toplam 4 çağrı yapılmıştır. Her çağrı sabit miktarda iş yapar: bir karşılaştırma, en çok bir çarpma ve bir dönüş. Zaman bu yüzden $O(n)$'dir. Yığın en derin anda 4 `faktoriyel` çerçevesi ve `main` içerir; bellek de $O(n)$'dir. Aynı çarpımı bir döngü $O(1)$ bellekle yapar. Özyinelemeli sürüm bu ek belleği, tanıma birebir uyan kısa bir kod için öder.

Sık yapılan hata

`int` türünün sınırını unutmak. `faktoriyel` `int` döndürür ve `int` en çok $2^{31} - 1 \approx 2{,}1$ milyar tutar. *12! = 479.001.600* sığar, ama *13! = 6.227.020.800* sığmaz ve sonuç sessizce yanlış çıkar. Daha büyük değerler için `long` (en çok *20!*) ya da `BigInteger` kullanılır. Bu, özyinelemeden bağımsız bir taşmadır; çağrı yığınının taşmasıyla karıştırılmamalıdır.

4.2.3 Yığın taşması

Çağrı yığını sınırsız değildir. JVM (*Java Virtual Machine*, Java sanal makinesi) her iş parçacığı için yığına sabit büyüklükte bir bellek ayırır. Bu büyüklük JVM'e ve işletim sistemine göre değişir ve JVM'in `-Xss` seçeneğiyle ayarlanabilir. Her çerçeve bu alandan biraz yer kaplar. Varsayılan ayarlarla özyineleme derinliği genellikle birkaç bin ile birkaç on bin çağrıyla sınırlıdır.

Aşağıdaki metot 1'den *n*'ye kadar sayıları toplar:

```
long topla(int n) {        // 1 + 2 + … + n
  if (n == 0) return 0;
  return n + topla(n - 1);
}
// main içinde:
System.out.println(topla(10_000));   // 50005000
System.out.println(topla(1_000_000)); // hata
```

`topla(10_000)` sorunsuz çalışır ve 50005000 yazar. Yığında aynı anda 10.001 `topla` çerçevesi bulunur; bu, varsayılan yığına sığar. `topla(1_000_000)` ise taban duruma ulaşmadan yığını doldurur. Bir milyon çerçeve için yer yoktur. Java bu durumda programı şu hatayla durdurur:

```
Exception in thread "main"
  java.lang.StackOverflowError
    at Ornek.topla(Ornek.java:3)
    at Ornek.topla(Ornek.java:3)
    …
```

Hata mesajının altındaki satırlar, hata anındaki çağrı yığınının dökümüdür. Aynı satırın (`Ornek.topla(Ornek.java:3)`) binlerce kez tekrarlanması, sorunun özyinelemeden geldiğini hemen gösterir. Bu döküme **yığın izi** (*stack trace*) denir.

Bir ölçüm fikir verir. Hiçbir iş yapmadan yalnız kendini çağıran bir metot, varsayılan ayarlarla yaklaşık 21.800 derinliğe inebildi. Çerçevesi biraz daha büyük olan `topla` için sınır daha düşüktü. Kesin değerin önemi yoktur ve makineden makineye değişir. Önemli olan sınırın *var olması* ve büyüklük derecesidir: on binler mertebesinde bir derinlik çoğu zaman sorun çıkarmaz, milyonlar mertebesinde bir derinlik ise neredeyse her zaman taşar.

Aynı toplam, özyinelemeye hiç gerek kalmadan hesaplanabilir. Bir döngü *O(n)* zaman ve *O(1)* bellek kullanır. Daha da iyisi, *1 + 2 + … + n = n(n + 1)/2* formülü sonucu *O(1)* zamanda verir. *n = 1.000.000* için bu 500.000.500.000'dir; sonuç `int`'e sığmadığı için metodun `long` döndürmesi doğru bir seçimdir.

> **Sık yapılan hata**
>
> `StackOverflowError`'ı `try – catch` ile yakalayıp programı sürdürmeye çalışmak. Bu hata `Exception` değil, `Error`'dır; JVM'in ciddi bir kaynak sorununu bildirir. Doğru çözüm, özyinelemenin derinliğini azaltmak (örneğin problemi her adımda yarıya bölmek) ya da metodu döngüye çevirmektir.

4.2.4 Taban durumu unutmak

Yığın taşmasının en sık nedeni çok büyük bir girdi değil, hatalı yazılmış bir taban durumdur. Üç tipik hata vardır. Üçünün de sonucu aynıdır, ama nedenleri farklıdır.

1. Taban durum yok.

```
int f(int n) {
  return n * f(n - 1);
}
```

Metot her çağrıda kendini yeniden çağırır. `f(3)` için girdiler 3, 2, 1, 0, −1, −2, … diye sürer. Hiçbir çağrı geri dönmez; yığın dolar.

2. Taban duruma ulaşılamıyor.

```
if (n == 1) return 1;
return n * f(n - 1);
```

Burada taban durum vardır, ama yalnız *n = 1*'i yakalar. Metot `f(0)` ile çağrılırsa girdiler 0, −1, −2, … diye azalır ve *n* hiçbir zaman 1 olmaz. Koşulu `n <= 1` yapmak bu girdiyi de yakalar: `f(0)` hemen 1 döndürür. Benzer bir durum, girdinin her çağrıda 2 azaldığı metotlarda da görülür. `n * f(n - 2)` biçimindeki bir metot `n == 1` taban durumuyla tek sayılarda durur, çift sayılarda 2, 0, −2, … diye sürer ve taşar.

3. Problem küçülmüyor.

```
if (n <= 1) return 1;
return n * f(n);
```

Bu kez taban durum doğrudur, ama özyinelemeli adım girdiyi küçültmez: `f(n)`, yine `f(n)`'yi çağırır. `f(1)` doğru cevap verir; `f(2)` ise sonsuza kadar `f(2)`'yi çağırır. Taban durum vardır ama ona hiç yaklaşılmaz.

Örnek 4.3

Aşağıdaki metot *n*'ye kadar çift sayıların toplamını bulmak için yazılmıştır: `int g(int n) { if (n == 0) return 0; return n + g(n - 2); }`. `g(6)` ve `g(5)` ne yapar?

1. `g(6)` : *6 ≠ 0*, `6 + g(4)`.
2. `g(4)` : `4 + g(2)` ; `g(2)` : `2 + g(0)`.
3. `g(0)` : taban durum, 0 döner. Dönüşte *2 + 0 = 2*, *4 + 2 = 6*, *6 + 6 = 12*. `g(6) = 12` ; 4 çağrı yapılmıştır.
4. `g(5)` : `5 + g(3)` ; `g(3)` : `3 + g(1)` ; `g(1)` : `1 + g(-1)` ; … Girdi hep tek kalır ve 0'a hiç eşit olmaz. Sonuç `StackOverflowError`.

Taban durum `n <= 0` yapılırsa `g(5)`, `g(-1)`'de durur ve *5 + 3 + 1 + 0 = 9* döndürür. Metot artık her girdide durur; ama tek bir *n* için tek sayıları topladığı da görülür. Metodun amacına göre ya girdinin çift olması şart koşulur ya da tek girdide önce *n − 1*'e geçilir.

Bu üç hatadan çıkan kural şudur: her olası girdi, 0 ve negatif değerler dahil, sonlu sayıda adımda bir taban duruma ulaşmalıdır. Bir özyinelemeli metot yazıldıktan sonra iki soru sorulur. Hangi girdiler doğrudan taban durumda biter? Öteki her girdi, her çağrıda bunlardan birine yaklaşıyor mu? İki sorunun cevabı da "evet" değilse metot bazı girdilerde taşar.

Sık yapılan hata

Taban durumu yalnızca örnek girdiyle denemek. `f(5)` doğru çalıştı diye metodun her girdide çalıştığı söylenemez. 0, 1, negatif bir sayı ve tek/çift ayrımı önemliyse her iki türden birer girdi ayrıca denenmelidir.

4.3 Doğrusal özyineleme

Şimdiye kadarki örneklerin hepsinde her çağrı en çok bir özyinelemeli çağrı yaptı. Bu, özyinelemenin en sade türüdür. Adını çağrıların oluşturduğu biçimden alır: çağrılar dallanmayan, düz bir zincir oluşturur. Kesimin konusu aynı türden üç problemdir: dizinin toplamı, dizinin tersine çevrilmesi ve üs alma. Son örnek, zincirin uzunluğunu kısaltmanın maliyeti nasıl değiştirdiğini gösterir.

4.3.1 Tanım ve dizinin toplamı

> **Tanım 4.3**
>
> **Doğrusal özyineleme** (*linear recursion*), her çağrının en fazla bir özyinelemeli çağrı yaptığı özyinelemedir. *n* boyutlu problem, *n − 1* (ya da *n / 2*) boyutlu tek bir probleme indirgenir; çağrılar dallanmayan bir zincir oluşturur.

`faktoriyel`, `geriSay` ve `topla` doğrusal özyinelemenin örnekleridir. Dizinin toplamı, diziyi tersine çevirme, üs alma ve bağlı listenin uzunluğu da bu türdendir.

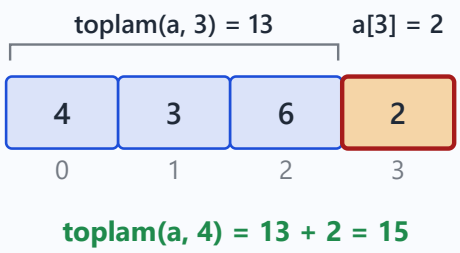
Bir tam sayı dizisindeki öğelerin toplamını özyinelemeyle bulalım. Özyinelemeli düşünmenin üç sorusu (4.1.3) burada da yol gösterir. Ama önce problemi küçültülebilir bir biçimde yazmak gerekir. "Dizinin toplamı" yerine "dizinin ilk *n* öğesinin toplamı" diye düşünülürse *n* küçültülebilecek bir parametre olur.

- **Taban durum:** *n = 0* ise parça boştur; boş parçanın toplamı 0'dır.
- **Küçültme:** İlk *n* öğenin toplamı, ilk *n − 1* öğenin toplamına dayandırılır.
- **Birleştirme:** İlk *n − 1* öğenin toplamına son öğe `a[n - 1]` eklenir.

Dizi indisleri 0'dan başladığı için ilk *n* öğe `a[0]`, …, `a[n - 1]`'dir; sondaki öğe `a[n]` değil `a[n - 1]`'dir. Kod şöyledir:

```
int toplam(int[] a, int n) {   // ilk n öğenin toplamı
  if (n == 0)
    return 0;                  // taban durum
  return toplam(a, n - 1) + a[n - 1];
}
```

Metodun ikinci parametresi, dizinin hangi parçasıyla ilgilenildiğini söyler. Dizinin kendisi hiçbir çağrıda değişmez ve kopyalanmaz. Java'da diziler referansla geçirilir: her çerçevede `a`, aynı dizinin adresini tutar. Böylece her çağrı yalnız bir referans ve bir tam sayı taşır. Bütün dizinin toplamı için ilk çağrı `toplam(a, a.length)` biçiminde yapılır.



Şekil 4.6: `a = {4, 3, 6, 2}` için tek adım. İlk üç öğenin toplamı `toplam(a, 3) = 13`'tür; son öğe `a[3] = 2`

eklenince `toplam(a, 4) = 13 + 2 = 15` bulunur.

Şekil 4.6 özyinelemeli düşünmenin özünü gösterir. `toplam(a, 4)`'ü yazan kişinin `toplam(a, 3)`'ün 13'ü nasıl bulduğunu bilmesi gerekmez. 13'ün doğru olduğunu kabul eder ve yalnızca son öğeyi ekler.

4.3.2 Kod izleme: toplam(a, 4)

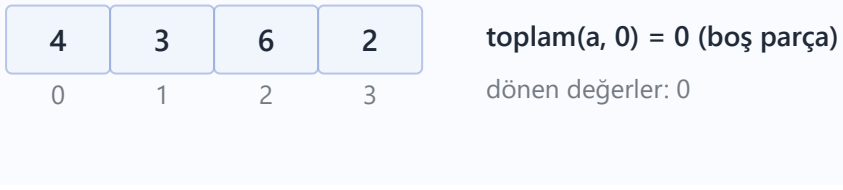
```
int toplam(int[] a, int n) {   // ilk n öğenin toplamı
  if (n == 0)
    return 0;                  // taban durum
  return toplam(a, n - 1) + a[n - 1];
}
// main içinde:
int[] a = {4, 3, 6, 2};
System.out.println(toplam(a, 4));
```

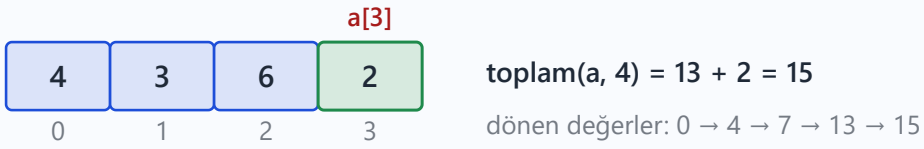
`main`, `a = {4, 3, 6, 2}` dizisi için `toplam(a, 4)`'ü çağırıyor. Her çağrı önce bir küçük parçanın toplamını ister; kendi öğesini toplamaya eklemeyi dönüşe bırakır.

Adım	Çağrı	Olay	Dönen
1	`toplam(a, 4)`	*n ≠ 0*: önce `toplam(a, 3)` ; `a[3] = 2` bekler	
2	`toplam(a, 3)`	önce `toplam(a, 2)` ; `a[2] = 6` bekler	
3	`toplam(a, 2)`	önce `toplam(a, 1)` ; `a[1] = 3` bekler	
4	`toplam(a, 1)`	önce `toplam(a, 0)` ; `a[0] = 4` bekler	
5	`toplam(a, 0)`	taban durum: boş parça	0
6	`toplam(a, 1)`	*0 + a[0] = 0 + 4*	4
7	`toplam(a, 2)`	*4 + a[1] = 4 + 3*	7
8	`toplam(a, 3)`	*7 + a[2] = 7 + 6*	13
9	`toplam(a, 4)`	*13 + a[3] = 13 + 2*	15

Tablo 4.3: `toplam(a, 4)` izlemesi, `a = {4, 3, 6, 2}`.

5. adımda yığın en derindedir: `main` ve 5 `toplam` çerçevesi. Toplamalar dönüş yolunda, dizinin başından sonuna doğru yapılır: önce `a[0]`, en son `a[3]`. Bunun nedeni, toplama işleminin özyinelemeli çağrıdan sonra yapılmasıdır; 4.1.5'teki `ileriSay` ile aynı durumdur.





Şekil 4.7: `toplam(a, 4)` izlemesinden iki an. Üstte taban durum (Tablo 4.3, 5. adım): boş parça soluk gösterilmiştir ve `toplam(a, 0)` 0 döndürür. Altta son durum: dönen değerler sırasıyla 0, 4, 7, 13, 15'tir; en son `a[3] = 2` eklenmiştir.

Maliyet: *n* öğeli parça için *n + 1* çağrı yapılır (*n, n − 1, …, 0*). Her çağrı bir karşılaştırma, bir toplama ve bir dizi erişimi yapar; zaman *O(n)*'dir. Yığın en derin anda *n + 1* çerçeve (ve `main`) içerir; bellek de *O(n)*'dir.

> **Sık yapılan hata**
>
> Birleştirme adımında `a[n]` yazmak. İlk *n* öğenin sonuncusu `a[n - 1]`'dir. `a[n]` yazılırsa ilk çağrıda (`n = a.length`) dizinin dışına çıkılır ve `ArrayIndexOutOfBoundsException` fırlatılır. Taban durum `n == 0` olduğu için `a[0]` da hiç toplanmaz.

Aynı kalıp dizinin başka özelliklerine de uygulanır. Örneğin dizinin en büyük öğesi de ilk *n* öğe üzerinden düşünülebilir. Taban durum *n = 1*'dir: tek öğeli parçanın en büyüğü `a[0]`'dır (boş parçanın en büyüğü tanımsızdır). Özyinelemeli adım, ilk *n − 1* öğenin en büyüğünü son öğeyle karşılaştırır: `Math.max(enBuyuk(a, n - 1), a[n - 1])`. Maliyet yine *O(n)* zaman ve *O(n)* yığındır.

4.3.3 Diziyi tersine çevirme

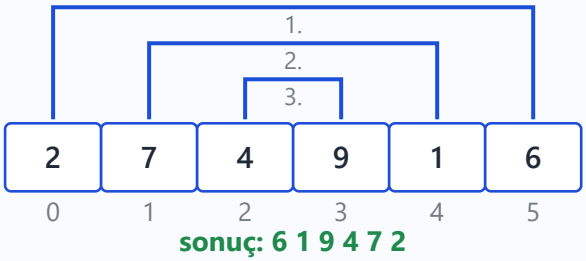
Bir diziyi yerinde, yani ikinci bir dizi kullanmadan, tersine çevirmek isteyelim. Fikir şudur: dizinin iki ucundaki öğelerin yeri değiştirilir, sonra aradaki parça aynı yöntemle tersine çevrilir. Aradaki parça, iki öğe daha kısa bir dizidir; problem böylece küçülür.

Bu kez parça, başlangıç ve bitiş indisleriyle verilir: `a[i..j]`, `i`'den `j`'ye kadar (ikisi dahil) öğeleri gösterir. İlk çağrı bütün dizi için yapılır: `ters(a, 0, n - 1)`.

- **Taban durum:** `i >= j` ise parçada 0 ya da 1 öğe kalmıştır. Tek öğeli ya da boş bir parça zaten kendi tersidir; yapılacak iş yoktur.
- **Küçültme ve birleştirme:** `a[i]` ile `a[j]` yer değiştirir; ardından iç parça `a[i+1..j-1]` tersine çevrilir.

```
void ters(int[] a, int i, int j) {
  if (i >= j)
    return;                    // taban: 0 ya da 1 öğe
  int t = a[i]; a[i] = a[j]; a[j] = t;  // uçlar
  ters(a, i + 1, j - 1);       // içteki parça
}
// main içinde:
int[] a = {2, 7, 4, 9, 1, 6};
ters(a, 0, a.length - 1);
```

Bu metodun öncekilerden iki farkı vardır. Birincisi, bir değer döndürmez (`void`). İş, dizinin kendisi değiştirilerek yapılır. Bütün çağrılar aynı diziyi gördüğü için bir çağrının yaptığı değiştirme ötekilere de yansır. İkincisi, iş özyinelemeli çağrıdan *önce* yapılır; dönüş yolunda yapılacak bir şey kalmaz.



Şekil 4.8: `{2, 7, 4, 9, 1, 6}` dizisini tersine çevirmek. Önce 0. ile 5. öğe, sonra 1. ile 4. öğe, en son 2. ile 3. öğe yer değiştirir. Sonuç: `6 1 9 4 7 2` .

Kaç değiştirme ve kaç çağrı yapılır? Her çağrı parçayı 2 öğe kısaltır. *n* öğeli dizide *⌊n/2⌋* değiştirme yapılır; buradaki *⌊ ⌋* işareti aşağı yuvarlamayı gösterir. Değiştirme yapan her çağrıya ek olarak, en sonda taban duruma düşen bir çağrı daha vardır. Toplam çağrı sayısı *⌊n/2⌋ + 1*'dir. 6 öğeli dizi için 3 değiştirme ve 4 çağrı; 7 öğeli dizi için de 3 değiştirme ve 4 çağrı (ortadaki öğe yerinde kalır). Zaman *O(n)*, yığın belleği *O(n)*'dir.

4.3.4 Kod izleme: ters(a, 0, 5)

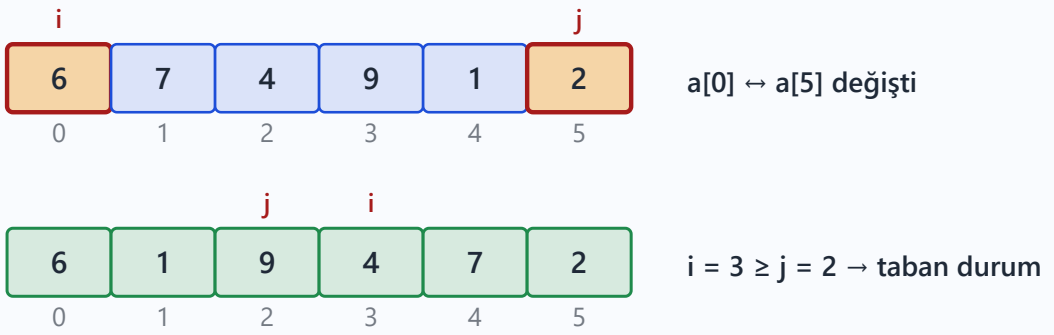
`a = {2, 7, 4, 9, 1, 6}` için `ters(a, 0, 5)` çağrısını izleyelim. Tabloda her çağrının `i` ve `j` değerleri ve çağrının sonunda dizinin durumu verilmiştir.

Çağrı	`i` , `j`	Olay	Dizi
—		başlangıç	`2 7 4 9 1 6`
1	0, 5	*0 < 5*: `a[0]` ↔ `a[5]` , sonra `ters(a, 1, 4)`	`6 7 4 9 1 2`
2	1, 4	*1 < 4*: `a[1]` ↔ `a[4]` , sonra `ters(a, 2, 3)`	`6 1 4 9 7 2`
3	2, 3	*2 < 3*: `a[2]` ↔ `a[3]` , sonra `ters(a, 3, 2)`	`6 1 9 4 7 2`

Çağrı	`i , j`	Olay	Dizi
4	3, 2	*3 ≥ 2*: taban durum, döner	`6 1 9 4 7 2`

Tablo 4.4: `ters(a, 0, 5)` izlemesi.

4. çağrıda yığında `main` ve 4 `ters` çerçevesi vardır. Bu çağrı döner; ardından 3., 2. ve 1. çağrılar sırayla döner. Dönüşte hiçbirinin yapacak işi yoktur, çünkü `ters(a, i + 1, j - 1)` satırı metodun son satırıdır. Sonuç: `6 1 9 4 7 2`; 3 değiştirme, 4 çağrı.



Şekil 4.9: `ters(a, 0, 5)` izlemesinden iki an. Üstte ilk değiştirme: `a[0]` ile `a[5]` yer değiştirmiş, dizi `6 7 4 9 1 2` olmuştur. Altta taban durum: *i = 3, j = 2*; indisler birbirini geçmiş, bütün öğeler yerine oturmuştur (yeşil).

Sık yapılan hata

Taban durumu `i == j` diye yazmak. Tek sayıda öğeli dizide indisler ortada buluşur ve `i == j` olur; ama çift sayıda öğeli dizide hiç eşit olmazlar. 6 öğeli dizide 3. çağrıdan sonra `i = 3, j = 2` olur; `i == j` yanlıştır, değiştirme yeniden yapılır ve dizi bozulmaya başlar. Çağrılar `(4, 1)`, `(5, 0)` ve `(6, -1)` ile sürer; sonuncusunda `a[6]`'ya erişilir ve `ArrayIndexOutOfBoundsException` fırlatılır. Doğru koşul `i >= j`'dir.

4.3.5 Üs alma: yavaş ve hızlı

x^n, x'in kendisiyle n kez çarpımıdır. Bu tanım hemen bir özyineleme verir: $x^n = x \cdot x^{n-1}$ ve $x^0 = 1$.

```
long us(long x, int n) {
  if (n == 0) return 1;
  return x * us(x, n - 1);
}
```

Bu metot doğrudur, ama yavaştır. n her çağrıda yalnızca 1 azalır; `us(x, n)` $n + 1$ çağrı yapar ($n, n - 1, \ldots, 0$). Zaman $O(n)$, yığın belleği $O(n)$'dir. $n = 1.000.000$ için bir milyon çerçeve gerekir ve yığın taşar.

Daha hızlı bir yol için üslerin bir özelliğinden yararlanılır: $x^{10} = x^5 \cdot x^5 = (x^5)^2$. x^5 bir kez hesaplanıp karesi alınırsa x^{10} bulunur. Böylece üs her adımda yarıya iner. Üs tek sayıysa yarıya tam bölünmez; o zaman bir x dışarıda bırakılır: $x^5 = x \cdot x^4 = x \cdot (x^2)^2$. Genel kural şudur:

> $x^n = (x^{n/2})^2$ (n çift) · $x^n = x \cdot (x^{(n-1)/2})^2$ (n tek)

Java'da tam sayı bölmesi aşağı yuvarladığı için tek n için `n / 2`, $(n - 1)/2$'ye eşittir. İki durumu da `n / 2` ile yazmak mümkündür. Örneğin 2^{10} şöyle açılır:

> $2^{10} = (2^5)^2,\ \ 2^5 = 2 \cdot (2^2)^2,\ \ 2^2 = (2^1)^2,\ \ 2^1 = 2 \cdot (2^0)^2$

```
long hizliUs(long x, int n) {
  if (n == 0) return 1;           // taban durum
  long y = hizliUs(x, n / 2);     // x^(n/2)
  if (n % 2 == 0) return y * y;   // n çift
  return x * y * y;               // n tek
}
// main içinde:
System.out.println(hizliUs(2, 10));
```

3. satırdaki `y` değişkeni bu metodun püf noktasıdır. $x^{n/2}$ bir kez hesaplanır ve `y`'de saklanır; kare almak için `y * y` yazılır. `hizliUs(x, n / 2) * hizliUs(x, n / 2)` yazılsaydı aynı değer iki kez hesaplanır ve kazanç kaybolurdu (bu tuzak 4.4.3'te yeniden ele alınır). 4. satır çift üs için kareyi döndürür. 5. satıra yalnız tek üste gelinir; burada kareye bir x çarpanı daha eklenir.

Hızlı sürüm kaç çağrı yapar? Her çağrıda n yarıya iner. n'yi 1'e indirmek için gereken yarılama sayısı $\lfloor \log_2 n \rfloor$'dir. Buna 1'in kendi çağrısı ve $n = 0$ taban durum çağrısı eklenir: toplam $\lfloor \log_2 n \rfloor + 2$ çağrı ($n \geq 1$ için). Zaman ve yığın belleği $O(\log n)$'dir.

n	`us` : çağrı	`hizliUs` : çağrı
10	11	5
100	101	8
1.000	1.001	11
10^6	1.000.001	21

Tablo 4.5: `us` ve `hizliUs` çağrı sayıları.

Tablo 4.5 iki büyüme hızı arasındaki farkı somutlaştırır. n bin kat büyüyünce `us`'un çağrı sayısı da bin kat büyür; `hizliUs`'unki yalnız yaklaşık 10 artar. Bir milyonluk üs için `us` yığını taşırır, `hizliUs` ise 21 çerçeveyle işini bitirir. Bu fark, problemi her adımda sabit miktarda değil, sabit oranda küçültmenin sonucudur.

Sık yapılan hata

Hızlı üs almanın sonucunun taşmayacağını sanmak. Çağrı sayısı azdır, ama sonuç yine çok büyüktür. `long` en çok yaklaşık $9{,}2 \cdot 10^{18}$ tutar; 2^{63} ve daha büyük kuvvetler sessizce yanlış çıkar. Hızlı üs alma pratikte çoğunlukla bir modüle göre (örneğin her çarpmadan sonra `% m` alınarak) kullanılır.

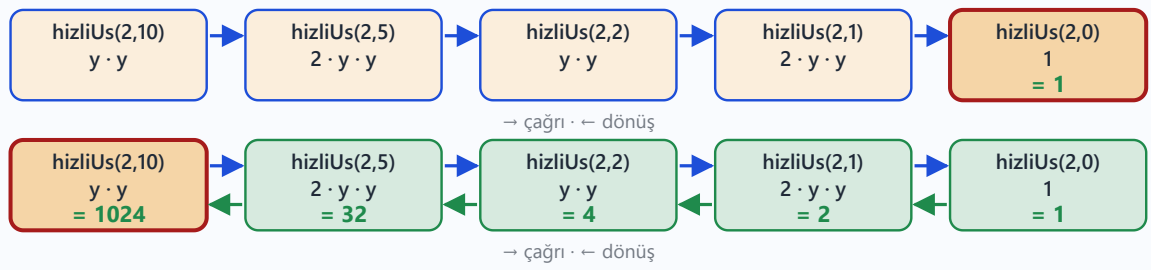
4.3.6 Kod izleme: hizliUs(2, 10)

`main`, `hizliUs(2, 10)`'u çağırıyor. İniş yolunda yalnız üs yarıya indirilir; çarpmaların hepsi dönüş yolunda yapılır.

Adım	Çağrı	Olay	Dönen
1	`hizliUs(2, 10)`	$n \neq 0$: önce `y = hizliUs(2, 5)` ($10 / 2 = 5$)	
2	`hizliUs(2, 5)`	önce `y = hizliUs(2, 2)` ($5 / 2 = 2$)	
3	`hizliUs(2, 2)`	önce `y = hizliUs(2, 1)` ($2 / 2 = 1$)	
4	`hizliUs(2, 1)`	önce `y = hizliUs(2, 0)` ($1 / 2 = 0$)	
5	`hizliUs(2, 0)`	taban durum: $2^0 = 1$	1
6	`hizliUs(2, 1)`	$y = 1$; $n = 1$ tek: $2 \cdot 1 \cdot 1$	2
7	`hizliUs(2, 2)`	$y = 2$; $n = 2$ çift: $2 \cdot 2$	4
8	`hizliUs(2, 5)`	$y = 4$; $n = 5$ tek: $2 \cdot 4 \cdot 4$	32
9	`hizliUs(2, 10)`	$y = 32$; $n = 10$ çift: $32 \cdot 32$	1024

Tablo 4.6: `hizliUs(2, 10)` izlemesi. İlk dört satır iniş, 5. satır taban durum, son dört satır dönüştür.

Sonuç 1024'tür ve yalnız 5 çağrıyla bulunmuştur; yavaş `us(2, 10)` 11 çağrı yapardı. Dönen değerler dikkatle incelenirse 1, 2, 4, 32, 1024 dizisi görülür. Bunlar sırasıyla 2^0, 2^1, 2^2, 2^5, 2^{10}'dur. Üsler 0, 1, 2, 5, 10'dur; bu sayılar 10'un ikili gösterimi olan `1010`'un soldan büyüyen önekleridir (`1`, `10`, `101`, `1010`). Tek üslü adımlar, ikili gösterimdeki 1 basamaklarına karşılık gelir.



Şekil 4.10: `hizliUs(2, 10)` çağrı zinciri. Üstte en derin an (Tablo 4.6, 5. adım): dört çağrı bekliyor, her kutuda dönüşte yapılacak işlem yazılı (çift üste `y · y`, tek üste `2 · y · y`). Altta son an: değerler 1, 2, 4, 32 diye geri dönmüş, ilk çağrı $32 \cdot 32 = 1024$'ü hesaplıyor.

Örnek 4.4

`hizliUs(3, 6)` kaç çağrı yapar ve ne döndürür?

1. İniş: n değerleri 6, 3, 1, 0'dır ($6 / 2 = 3$, $3 / 2 = 1$, $1 / 2 = 0$). 4 çağrı yapılır; formül de $\lfloor \log_2 6 \rfloor + 2 = 2 + 2 = 4$ verir.
2. `hizliUs(3, 0)` 1 döndürür.
3. `hizliUs(3, 1)` : n tek, $3 \cdot 1 \cdot 1 = 3$.
4. `hizliUs(3, 3)` : n tek, $3 \cdot 3 \cdot 3 = 27$.
5. `hizliUs(3, 6)` : n çift, $27 \cdot 27 = 729$.

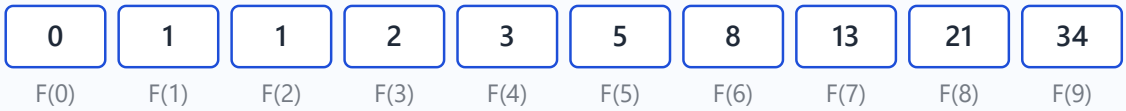
Sonuç $3^6 = 729$. Yavaş sürüm 7 çağrı yapardı.

4.4 İkili özyineleme

Bazı problemlerde bir çağrı, kendini iki kez çağırır. Çağrılar artık düz bir zincir oluşturmaz; her çağrıdan iki dal çıkar ve bir ağaç oluşur. Böyle yapıların en bilinen örneği Fibonacci sayılarıdır. Fibonacci ağacının maliyeti ve bu maliyetten kurtulma yolları burada işlenir. Kesimin sonundaki ikili aramanın kodunda da iki çağrı yazılıdır, ama çağrılar ağaç değil zincir oluşturur.

4.4.1 Fibonacci sayıları

Fibonacci dizisi 0 ve 1 ile başlar. Sonraki her sayı, kendinden önceki iki sayının toplamıdır: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, … Dizi doğada (ayçiçeğindeki tohum dizilişi, bazı bitkilerde yaprakların dizilişi) ve bilgisayar biliminde (bazı veri yapılarının analizi) sık karşımıza çıkar.



Şekil 4.11: Fibonacci dizisinin ilk on terimi, $F(0)$'dan $F(9)$'a: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34. Her terim, önceki iki terimin toplamıdır; örneğin $F(7) = F(6) + F(5) = 8 + 5 = 13$.

Dizinin tanımı kendiliğinden özyinelemelidir:

> $F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2) \ (n \geq 2)$

Bu tanımda iki taban durum vardır: $n = 0$ ve $n = 1$. İki taban durum gereklidir, çünkü özyinelemeli adım iki önceki terime başvurur. Yalnız $F(0)$ verilseydi $F(1) = F(0) + F(-1)$ tanımsız kalırdı. Tanım koda doğrudan dökülür:

```
int fib(int n) {
  if (n <= 1) return n;    // taban durum
  return fib(n - 1) + fib(n - 2);
}
```

2. satır iki taban durumu tek koşulla yakalar: $n \le 1$ iken $F(n) = n$'dir ($F(0) = 0$, $F(1) = 1$). 3. satır iki özyinelemeli çağrı yapar ve sonuçlarını toplar.

Tanım 4.4

İkili özyineleme (*binary recursion*), bir çağrının iki özyinelemeli çağrı yaptığı özyinelemedir. Bir çağrı ikiden çok özyinelemeli çağrı yapabiliyorsa **çoklu özyineleme** (*multiple recursion*) denir. Bu türlerde çağrılar bir **çağrı ağacı** oluşturur: her düğüm bir çağrıdır; düğümün çocukları, o çağrının yaptığı çağrılardır.

Çağrı ağacı, ikili özyinelemeyi çözümlemenin temel aracıdır. Ağaçtaki düğüm sayısı toplam çağrı sayısını, ağacın derinliği ise yığında aynı anda bulunan en çok çerçeve sayısını verir. Fibonacci ağacının dikkat çekici bir özelliği, aynı değerin ağaçta birden çok kez hesaplanmasıdır. `fib`'in maliyetini bu tekrar belirler.

4.4.2 Kod izleme: fib(4) çağrı ağacı

```
int fib(int n) {
  if (n <= 1) return n;    // taban durum
  return fib(n - 1) + fib(n - 2);
}
// main içinde:
System.out.println(fib(4));
```

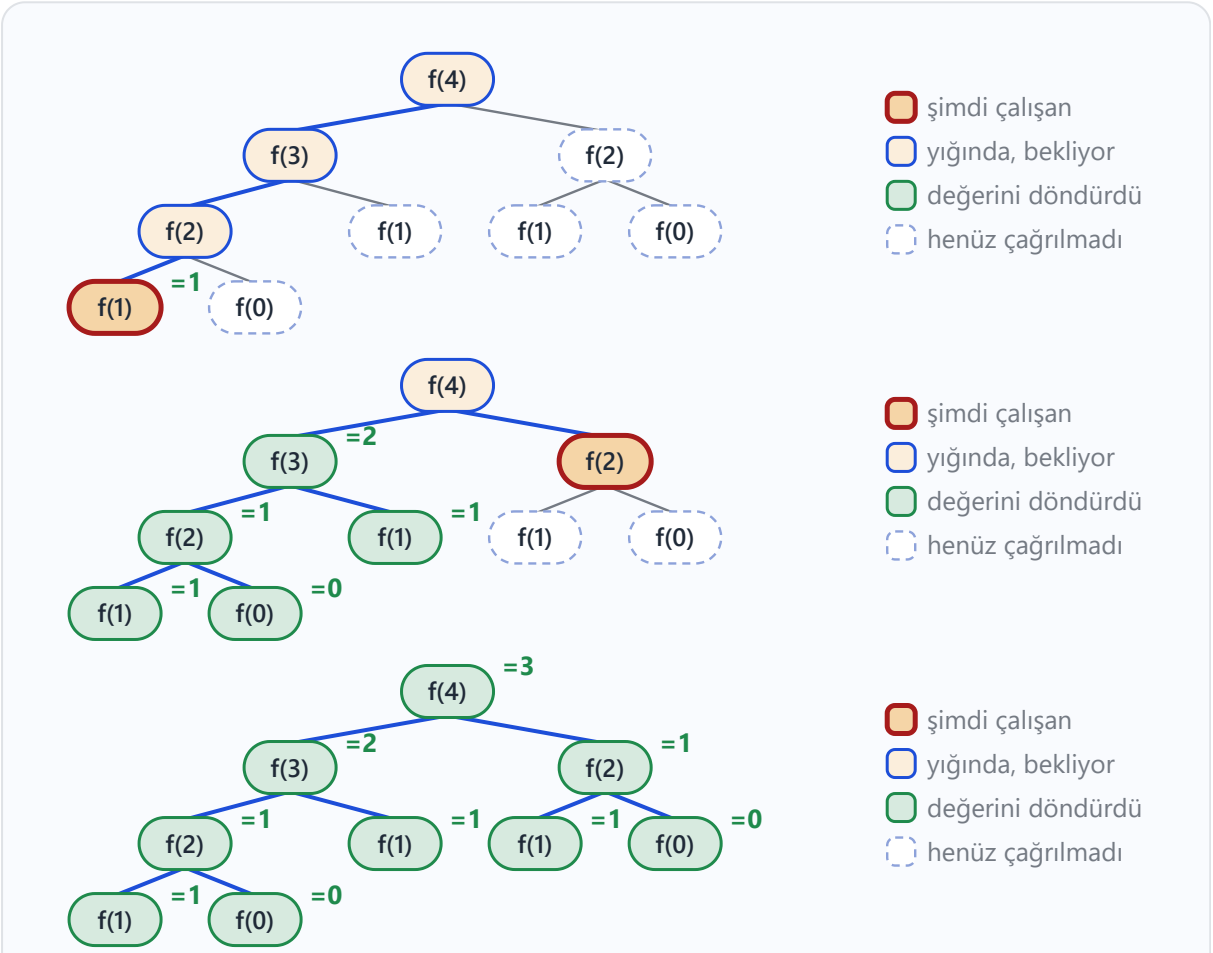
`fib(4)`'ü izlerken önemli kural şudur: Java bir ifadeyi soldan sağa değerlendirir. `fib(n - 1) + fib(n - 2)` ifadesinde önce `fib(n - 1)` çağrılır ve *tamamen* biter. Ancak ondan sonra `fib(n - 2)` çağrılır. Sol dal bütünüyle bitmeden sağ dala geçilmez.

Çağrı	Kim çağırdı	Olay	Dönen
1: `fib(4)`	`main`	*n > 1*: önce `fib(3)`; `fib(2)` sırada bekler	
2: `fib(3)`	1, sol	önce `fib(2)`; `fib(1)` sırada bekler	
3: `fib(2)`	2, sol	önce `fib(1)`; `fib(0)` sırada bekler	
4: `fib(1)`	3, sol	taban durum	1
5: `fib(0)`	3, sağ	taban durum	0
3: `fib(2)`		*1 + 0*	1

Çağrı	Kim çağırdı	Olay	Dönen
6: `fib(1)`	2, sağ	taban durum	1
2: `fib(3)`		*1 + 1*	2
7: `fib(2)`	1, sağ	ikinci kez hesaplanıyor: önce `fib(1)`	
8: `fib(1)`	7, sol	taban durum	1
9: `fib(0)`	7, sağ	taban durum	0
7: `fib(2)`		*1 + 0*	1
1: `fib(4)`		*2 + 1*	3

Tablo 4.7: `fib(4)` izlemesi. Çağrılar başlama sırasıyla numaralanmıştır.

Sonuç *F(4) = 3*'tür ve 9 çağrıyla bulunmuştur. Bu 9 çağrının dağılımı şöyledir: `fib(4)` ve `fib(3)` birer kez, `fib(2)` iki kez, `fib(1)` üç kez, `fib(0)` iki kez. `fib(2)`'nin ikinci hesabı (7. çağrı) tamamen gereksizdir: aynı değer 3. çağrıda zaten bulunmuştu, ama hiçbir yerde saklanmadığı için yeniden hesaplanır.



Şekil 4.12: `fib(4)` çağrı ağacının üç anı. Üstte ilk taban durum: `fib(1)` çalışıyor (kırmızı çerçeveli), `fib(2)`, `fib(3)`, `fib(4)` yığında bekliyor; kesikli düğümler henüz çağrılmadı. Ortada `fib(3) = 2` döndükten sonra `fib(4)`, sağ dal için `fib(2)`'yi ikinci kez hesaplamaya başlamış. Altta bütün ağaç: 9 çağrı da değerini döndürmüş (yeşil), `fib(4) = 3`.

Şekil 4.12'deki ağaçtan bellek maliyeti de okunur. Yığında aynı anda bulunan çerçeveler, kökten o an çalışan düğüme giden yoldur. En uzun yol `fib(4)` → `fib(3)` → `fib(2)` → `fib(1)`'dir: 4 çerçeve, `main` ile 5. Toplam çağrı 9 olduğu hâlde yığın hiçbir zaman 4 `fib` çerçevesini aşmaz.

> **Sık yapılan hata**
>
> İki özyinelemeli çağrının aynı anda çalıştığını sanmak. Java'da bir metot çağrısı bitmeden çağıran satır devam etmez. `fib(n - 1) + fib(n - 2)` ifadesinde ikinci çağrı, birincinin bütün alt ağacı bitince başlar. Çağrı ağacı bu yüzden soldan sağa, derinlemesine gezilir.

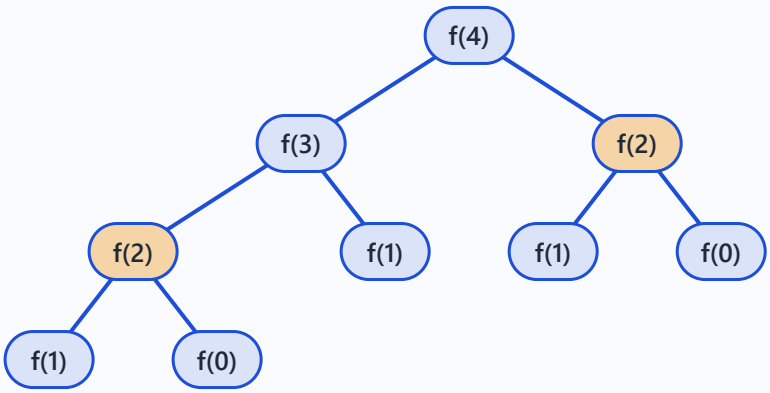
4.4.3 Özyinelemenin maliyeti

Özyinelemeli bir metodun zaman maliyeti, yaptığı toplam iştir. Her çağrı sabit miktarda iş yapıyorsa (bir karşılaştırma, bir toplama, bir dönüş), toplam iş çağrı sayısıyla orantılıdır. Maliyeti bulmak, çağrı sayısını bulmaya indirgenir.

`faktoriyel` için bu kolaydır. `faktoriyel(n)` bir kez `faktoriyel(n - 1)`'i çağırır. $T(n)$, `faktoriyel(n)`'in yaptığı toplam iş olsun ve c bir çağrının kendi içindeki sabit işi göstersin. O zaman $T(n) = T(n - 1) + c$ yazılır. Her çağrı bir sonrakini yapar ve zincir n çağrı uzunluğundadır. Zaman $O(n)$, yığın belleği $O(n)$'dir. Bu tür denklemlere **yineleme bağıntısı** denir; 4.6 bu konuya ayrılmıştır.

`fib` için durum çok farklıdır. Her çağrı iki çağrı yapar ve aynı alt problemler tekrar tekrar çözülür.

```
int fib(int n) {
  if (n <= 1) return n;
  return fib(n - 1) + fib(n - 2);
}
```



Şekil 4.13: `fib(4)` çağrı ağacı; 9 düğüm. Turuncu `f(2)` düğümleri aynı alt problemin iki ayrı hesabıdır. Her `f(2)` kendi altında `f(1)` ve `f(0)`'ı yeniden çağırır.

Çağrı sayısı n büyüdükçe çok hızlı artar. `fib(n)`'in toplam çağrı sayısına $C(n)$ diyelim. $n \le 1$ için tek çağrı yapılır: $C(0) = C(1) = 1$. Daha büyük n için çağrının kendisi, sol alt ağaç ve sağ alt ağaç toplanır: $C(n) = C(n-1) + C(n-2) + 1$. Bu bağıntı Fibonacci bağıntısına çok benzer ve çözümü de Fibonacci sayılarıyla yazılır.

Teorem 4.1 (Fibonacci çağrı sayısı)

Özyinelemeli `fib(n)` toplam $C(n) = 2 \cdot F(n+1) - 1$ çağrı yapar.

İspat. Tümevarımla gösterelim. $n = 0$ için $2 \cdot F(1) - 1 = 2 \cdot 1 - 1 = 1$; $n = 1$ için $2 \cdot F(2) - 1 = 2 \cdot 1 - 1 = 1$. İkisi de doğrudur. $n \ge 2$ için formülün $n-1$ ve $n-2$'de doğru olduğunu kabul edelim. Bağıntıya koyunca $C(n) = (2F(n) - 1) + (2F(n-1) - 1) + 1 = 2(F(n) + F(n-1)) - 1 = 2F(n+1) - 1$ bulunur. □

Teorem, `fib(4)` için $2 \cdot F(5) - 1 = 2 \cdot 5 - 1 = 9$ verir; izlemeyle aynı. Fibonacci sayıları her adımda yaklaşık 1,6 katına çıktığı için (oran, altın oran denen $\varphi \approx 1{,}618$ sayısına yaklaşır) çağrı sayısı da yaklaşık $1{,}6^n$ gibi büyür. Bu **üstel** bir büyümedir: n her 1 arttığında iş yaklaşık 1,6 katına çıkar.

Bellek ise hâlâ $O(n)$'dir. Ağaç ne kadar büyük olursa olsun, yığında aynı anda yalnız kökten bir düğüme giden tek bir yol bulunur ve en uzun yol n düğümlüdür. `fib`'in sorunu bellek değil, zamandır.

Sık yapılan hata

Aynı özyinelemeli çağrıyı bir ifadede iki kez yazmak. `hizliUs`'ta `y * y` yerine `hizliUs(x, n / 2) * hizliUs(x, n / 2)` yazılırsa doğrusal bir zincir ikili bir ağaca dönüşür. Ağacın derinliği $\log_2 n$ olduğu için düğüm sayısı yaklaşık $2^{\log n} = n$ olur: $O(\log n)$'lik metot $O(n)$'e düşer. Aynı değer birden çok kez gerekiyorsa bir kez hesaplanıp bir değişkende saklanır.

4.4.4 Tekrarlanan hesaptan kaçınmak

Üstel büyümenin ne demek olduğu sayılara bakınca daha iyi anlaşılır. Tablo 4.8, `fib(n)`'in değerini ve yaptığı çağrı sayısını birkaç n için verir.

n	`fib(n)`	çağrı sayısı
5	5	15
10	55	177
20	6.765	21.891
30	832.040	2.692.537
40	102.334.155	331.160.281

Tablo 4.8: Özyinelemeli `fib`'in çağrı sayısı.

`fib(40)` 331 milyondan fazla çağrı yapar. Oysa farklı alt problem sayısı yalnızca 41'dir: `fib(0)`'dan `fib(40)`'a kadar. Çağrıların neredeyse hepsi daha önce çözülmüş bir problemi yeniden çözer. Bu israfı önlemenin iki yolu vardır.

1. Bellekleme. Bir değer ilk kez hesaplandığında bir dizide saklanır. Aynı değer yeniden istendiğinde hesaplanmaz, diziden okunur. Bu tekniğe **bellekleme** (*memoization*) denir.

```
long[] bellek = new long[91];   // 0: hesaplanmadı
long fibBellek(int n) {
  if (n <= 1) return n;
  if (bellek[n] != 0) return bellek[n];
  bellek[n] = fibBellek(n - 1) + fibBellek(n - 2);
  return bellek[n];
}
```

`bellek` dizisi 91 öğelidir ve *F(0)–F(90)* değerlerini tutar. Bu değerlerin hepsi `long`'a sığar; `long` ancak *F(92)*'ye kadar yeter. Dizinin her hücresi başlangıçta 0'dır ve 0 "henüz hesaplanmadı" anlamına gelir. Bu seçim güvenlidir, çünkü $n \geq 2$ için *F(n)* hiçbir zaman 0 değildir. Metot önce taban duruma bakar. Ardından değerin daha önce hesaplanıp hesaplanmadığını denetler; hesaplanmışsa hemen döndürür. Hesaplanmamışsa iki çağrıyı yapar, sonucu `bellek[n]`'e yazar ve döndürür.

`fibBellek(40)` 79 çağrı yapar. Bunun nedeni şudur. `fibBellek(n)`'in sol çağrısı `fibBellek(n - 1)`, $n - 1$'e kadar her değeri hesaplar ve saklar. Sağ çağrı `fibBellek(n - 2)` geldiğinde değer zaten bellektedir; çağrı hemen döner. Her $k \geq 2$ için bir "hesaplayan" çağrı ve onun sağ dalında değeri hazır bulan (ya da `fibBellek(0)` gibi taban duruma düşen) bir çağrı vardır; buna en alttaki `fibBellek(1)` eklenir. Toplam $2n - 1$ çağrı yapılır: $n = 40$ için 79. Zaman *O(n)*'dir; yığın derinliği n ve `bellek` dizisi de *O(n)* yer kaplar.

2. Döngüyle hesaplamak. Fibonacci dizisi baştan sona doğru hesaplanırsa her adımda yalnız son iki terim gerekir.

```
long fibDongu(int n) {
  long a = 0, b = 1;              // F(0), F(1)
  for (int i = 0; i < n; i++) {
    long t = a + b;  a = b;  b = t;
  }
  return a;
}
```

`a` ve `b` her an ardışık iki terimi tutar. Döngünün başında `a = F(0)`, `b = F(1)`'dir. Her turda `t = a + b` bir sonraki terimi hesaplar; sonra pencere bir adım kayar. n tur sonunda `a = F(n)` olur.

`i`	`t = a + b`	`a`	`b`
başlangıç		0	1

`i`	`t = a + b`	`a`	`b`
0	1	1	1
1	2	1	2
2	3	2	3
3	5	3	5
4	8	5	8
5	13	8	13

Tablo 4.9: `fibDongu(6)` izlemesi.

Altı turdan sonra `a = 8 = F(6)` döndürülür. Döngülü sürüm *O(n)* zaman ve *O(1)* bellek kullanır; çerçeve açmaz, yığın taşması riski yoktur. Fibonacci için en iyi seçim budur.

Sık yapılan hata

Sonuç saklama dizisini metodun içinde tanımlamak. `long[] bellek = new long[91];` satırı `fibBellek`'in içine yazılırsa her çağrı kendi boş dizisini oluşturur; hiçbir değer çağrılar arasında paylaşılmaz ve metot yine üstel sürede çalışır. Dizi, bütün çağrıların gördüğü bir yerde (bir alan ya da parametre olarak) tutulmalıdır.

4.4.5 Özyinelemeli ikili arama

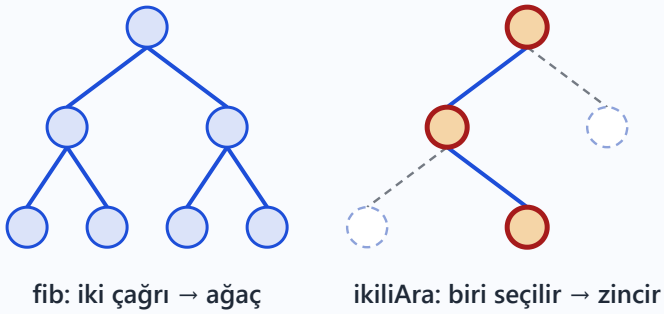
Sıralı bir dizide bir *x* değerini aramak için ortadaki öğeye bakılır. *x* ortadaki öğeye eşitse aranan bulunmuştur. *x* daha küçükse, dizi sıralı olduğu için ancak sol yarıda olabilir; daha büyükse ancak sağ yarıda. Her iki durumda da aynı arama, yarı boyutlu bir parça üzerinde tekrarlanır. Bu, doğası gereği özyinelemeli bir algoritmadır.

Aranan parça `lo` ve `hi` indisleriyle verilir: `a[lo..hi]`. İki taban durum vardır:

- **Aralık boş** (`lo > hi`): aranacak öğe kalmamıştır; *x* dizide yoktur. Metot −1 döndürür.
- **Ortadaki öğe *x***: aranan bulunmuştur; metot ortanın indisini döndürür.

```
int ikiliAra(int[] a, int x, int lo, int hi) {
  if (lo > hi) return -1;       // taban: boş aralık
  int mid = (lo + hi) / 2;
  if (a[mid] == x) return mid;  // taban: bulundu
  if (x < a[mid])
    return ikiliAra(a, x, lo, mid - 1); // sol yarı
  return ikiliAra(a, x, mid + 1, hi);   // sağ yarı
}
System.out.println(ikiliAra(a, 23, 0, 9));
```

Kodda iki özyinelemeli çağrı yazılıdır (6. ve 7. satırlar). Ama bir çağrı bunlardan yalnız birini yapar: `if` koşulu doğruysa sol yarıya, değilse sağ yarıya gidilir. Fibonacci'deki gibi bir ağaç oluşmaz; çağrılar yine bir zincirdir. İkili arama bu yüzden ikili değil, *doğrusal* özyinelemedir. Örnek, ağaç ile zincir arasındaki farkı göstermek için bu kesimde ele alınır.



Şekil 4.14: Solda `fib` : her çağrı iki çağrı yapar, ağaç oluşur. Sağda `ikiliAra` : iki yarıdan yalnız birine girilir (kırmızı yol); öteki dala hiç gidilmez (kesikli). Sonuç bir zincirdir.

Her çağrıda aralık yarıya iner. Yineleme bağıntısı $T(n) = T(n/2) + c$'dir; çözümü $O(\log n)$ zaman ve $O(\log n)$ yığın belleğidir (4.6.3). $n = 10$ öğeli dizide en çok 5, $n = 10^6$ öğeli dizide en çok 21 çağrı yapılır; bu sayılar boş aralıkla biten son çağrıyı da içerir.

Sık yapılan hata

Özyinelemeli çağrının önündeki `return` 'ü unutmak. 6. satır `ikiliAra(a, x, lo, mid - 1);` diye yazılırsa sol yarının sonucu atılır ve akış 7. satıra, sağ yarı çağrısına düşer. Derleyici bu hatayı her zaman yakalamaz (metodun sonunda bir `return` olduğu için kod derlenir), ama cevap yanlış çıkar. Bir değer döndüren özyinelemeli metotta, dönen değerin kullanıldığından ya da iletildiğinden emin olunmalıdır.

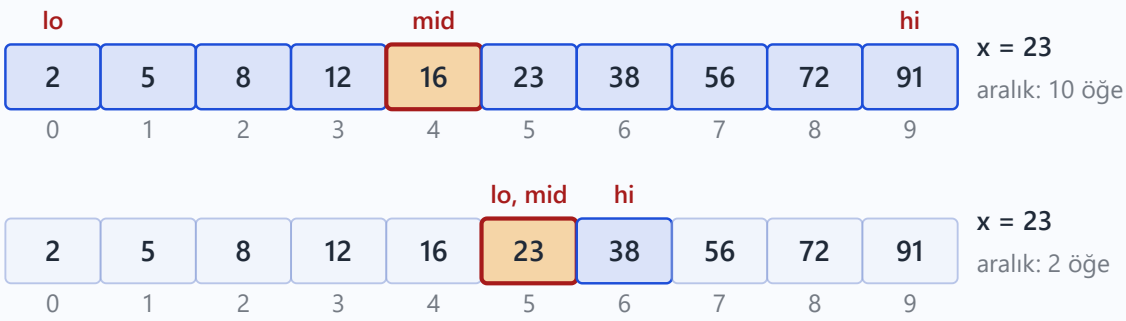
4.4.6 Kod izleme: ikiliAra(a, 23, 0, 9)

Dizi `a = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}` , aranan $x = 23$'tür. `main` , `ikiliAra(a, 23, 0, 9)` 'u çağırır.

Çağrı	`lo , hi`	`mid`	`a[mid]`	Karar
1	0, 9	$(0 + 9)/2 = 4$	16	$23 > 16$: sağ yarı, `ikiliAra(a, 23, 5, 9)`
2	5, 9	$(5 + 9)/2 = 7$	56	$23 < 56$: sol yarı, `ikiliAra(a, 23, 5, 6)`
3	5, 6	$(5 + 6)/2 = 5$	23	bulundu: 5 döner

Tablo 4.10: `ikiliAra(a, 23, 0, 9)` izlemesi.

3. çağrı taban durumda 5 döndürür. 2. çağrı aldığı 5'i `return` ile olduğu gibi iletir; 1. çağrı da aynısını yapar. Dönüş yolunda yeni bir hesap yapılmaz. Sonuç 5'tir ve 3 çağrıyla bulunmuştur; aralık 10 → 5 → 2 öğeye inmiştir.



Şekil 4.15: `ikiliAra(a, 23, 0, 9)` izlemesinden iki an. Üstte 1. çağrı: aralık 10 öğe, `mid = 4` ve `a[4] = 16`. Altta 3. çağrı: aralık `lo = 5`, `hi = 6` ile 2 öğeye inmiş, `a[5] = 23` bulunmuş. Aralık dışındaki öğeler soluk gösterilmiştir.

Örnek 4.5

Aynı dizide *x = 50* aranırsa ne olur?

1. `(0, 9)` : `mid = 4`, `a[4] = 16`; *50 > 16*, sağ yarı.
2. `(5, 9)` : `mid = 7`, `a[7] = 56`; *50 < 56*, sol yarı.
3. `(5, 6)` : `mid = 5`, `a[5] = 23`; *50 > 23*, sağ yarı.
4. `(6, 6)` : `mid = 6`, `a[6] = 38`; *50 > 38*, sağ yarı.
5. `(7, 6)` : `lo > hi`, aralık boş: −1 döner.

5 çağrı yapılır; *n = 10* için en kötü durum budur. Son çağrıdaki `lo = 7`, 50'nin sıralamayı bozmadan ekleneceği indistir; ama metot bu bilgiyi döndürmez, yalnız −1 döndürür.

Sık yapılan hata

Yeni aralığı `mid` ile sınırlamak: `ikiliAra(a, x, lo, mid)` ya da `ikiliAra(a, x, mid, hi)`. `a[mid]` zaten denetlenmiştir; aralıkta bırakılırsa iki öğeli bir aralıkta `mid = lo` çıkar ve aynı aralıkla tekrar tekrar çağrı yapılır. Problem küçülmez ve yığın taşar. Doğrusu `mid - 1` ve `mid + 1`'dir.

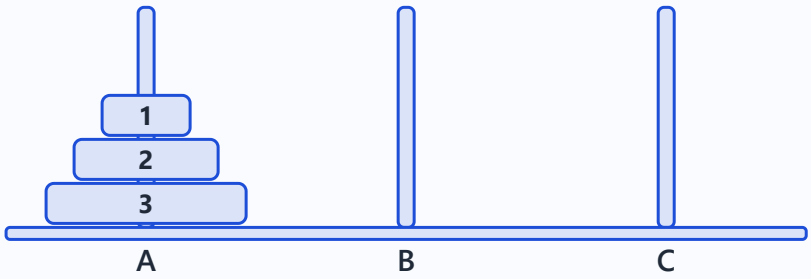
4.5 Hanoi kuleleri ve geri izleme

Önceki kesimdeki Fibonacci ağacı, özyinelemenin kötü kullanımına bir örnekti: aynı iş tekrar tekrar yapılıyordu. Bu kesimdeki problemlerde de çağrı ağacı üstel büyür, ama bu kez ağacın her düğümü *farklı* ve gerekli bir iş yapar. Hanoi kulelerinde her düğüm bir disk hamlesidir. Geri izlemede her yaprak, üretilmesi istenen bir alt küme ya da permütasyondur. Bu problemlerde üstel maliyetten kaçış yoktur, çünkü çıktının kendisi üstel büyüklüktedir. Özyineleme ise bu çıktıyı üretmenin en açık yoludur.

4.5.1 Hanoi kuleleri

Hanoi kuleleri (*Towers of Hanoi*), bilgisayar biliminde ve matematikte klasik bir problemdir. Üç çubuk vardır: A, B ve C. A çubuğunda, büyükten küçüğe doğru üst üste dizilmiş n disk bulunur; en büyük disk en alttadır. Amaç, bütün diskleri A'dan C'ye taşımaktır. B çubuğu yardımcı olarak kullanılabilir. Taşıma iki kurala uyar:

- Her hamlede tek disk taşınır; bu, bir çubuğun en üstündeki disktir.
- Disk yalnız kendinden büyük bir diskin üstüne ya da boş çubuğa konabilir.



Şekil 4.16: Üç diskli Hanoi kuleleri. Başlangıçta üç disk A çubuğunda, büyükten küçüğe dizilidir (3 en büyük, 1 en küçük). Hedef, hepsini C'ye taşımaktır.

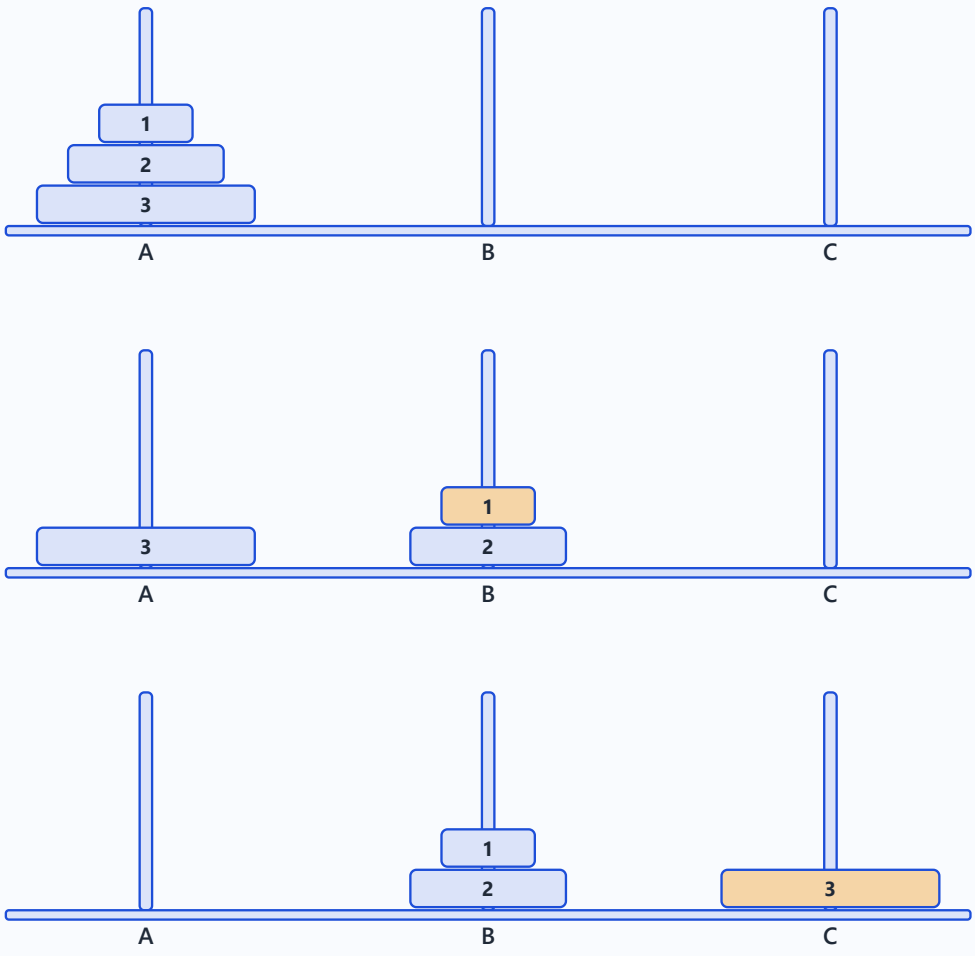
Problemi doğrudan, hamle hamle düşünmek zordur. Özyinelemeli düşünmek ise çözümü neredeyse kendiliğinden verir. En büyük diske odaklanalım. Bu disk bir noktada A'dan C'ye taşınmak zorundadır. O hamle yapılırken A'da başka disk olamaz (en büyük disk en üstte olmalıdır) ve C boş olmalıdır (en büyük disk hiçbir diskin üstüne konamaz). Öyleyse geri kalan $n - 1$ diskin hepsi o anda B'de olmalıdır. Bu gözlem üç adımlı bir plan verir:

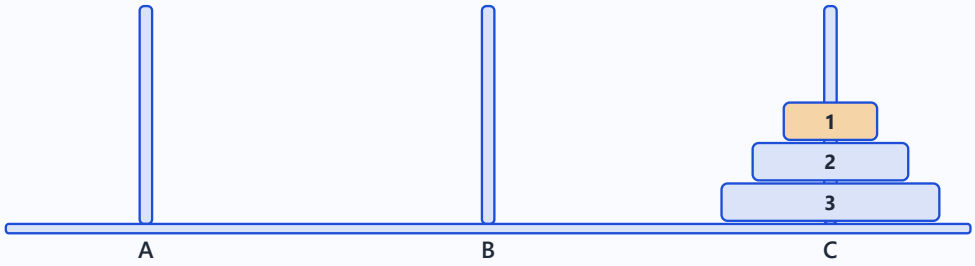
1. Üstteki $n - 1$ diski A'dan B'ye taşı (C'yi yardımcı kullanarak).
2. En büyük diski A'dan C'ye taşı.
3. $n - 1$ diski B'den C'ye taşı (A'yı yardımcı kullanarak).

1. ve 3. adımlar, *n − 1* diskli aynı problemdir; yalnız çubukların rolleri değişmiştir. Bu iki adım sırasında en büyük disk hiç rahatsız edilmez ve her disk ondan küçük olduğu için onun üstüne serbestçe konabilir. En büyük disk, küçük problem için yok sayılabilir. Taban durum tek disktir: tek disk doğrudan kaynaktan hedefe taşınır.

4.5.2 Üç diskle yedi hamle

Üç diskli problemde plan şöyle işler. Üstteki 2 disk A'dan B'ye taşınır; bu, 3 hamle tutar. Disk 3, A'dan C'ye gider. Sonra 2 disk B'den C'ye taşınır; bu da 3 hamledir. Toplam 7 hamle eder.





Şekil 4.17: Üç diskle yedi hamlenin dört anı, yukarıdan aşağı: başlangıç; 3. hamleden sonra (küçük iki disk B'de toplanmış); 4. hamleden sonra (disk 3 C'de); 7. hamleden sonra (bütün diskler C'de). Hamleler sırasıyla şunlardır: disk 1, A → C; disk 2, A → B; disk 1, C → B; disk 3, A → C; disk 1, B → A; disk 2, B → C; disk 1, A → C. Her durumda son hamlede taşınan disk turuncu gösterilmiştir.

Şekil 4.17, planın üç adımını açıkça gösterir. İlk üç hamle küçük iki diski B'de toplar (hamle 3'ten sonraki durum). Hamle 4 en büyük diski taşır. Son üç hamle küçük iki diski C'de en büyük diskin üstüne yerleştirir. İlk üç hamle ile son üç hamle aynı "2 diski taşı" işinin iki örneğidir; yalnız çubuklar farklıdır.

Hamle sayısı bu plandan hesaplanır. $T(n)$, n diski taşımak için gereken hamle sayısı olsun. Tek disk için $T(1) = 1$'dir. Daha fazla disk için planın üç adımı toplanır: $T(n) = T(n-1) + 1 + T(n-1) = 2T(n-1) + 1$. Birkaç değer hesaplayalım: $T(2) = 3$, $T(3) = 7$, $T(4) = 15$. Her değer bir ikinin kuvvetinin bir eksiğidir.

Teorem 4.2 (Hanoi hamle sayısı)

n diskli Hanoi kulelerini çözmek için $2^n - 1$ hamle yeter ve gerekir. Özyinelemeli plan tam $2^n - 1$ hamle yapar; daha az hamleli bir çözüm yoktur.

İspat. *Yeterlik.* Tümevarımla: $T(1) = 1 = 2^1 - 1$. $T(n-1) = 2^{n-1} - 1$ ise $T(n) = 2(2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1$.

Gereklik. Herhangi bir çözümde en büyük disk en az bir kez taşınır. İlk taşındığı anda, yukarıdaki akıl yürütmeye göre öteki $n - 1$ disk tek bir çubukta toplanmış olmalıdır; bunun için en az $T(n-1)$ hamle gerekir. En büyük diskin C'ye son taşınışından sonra da $n - 1$ disk bir çubuktan C'ye taşınmalıdır: yine en az $T(n-1)$ hamle. Arada en büyük disk için en az bir hamle vardır. Toplam en az $2T(n-1) + 1$ hamledir. □

$2^n - 1$ çok hızlı büyür. 20 disk için 1.048.575 hamle gerekir. Bir efsaneye göre tapınaktaki rahipler 64 diskli kuleyi taşıyor; iş bitince dünyanın sonu gelecektir. $2^{64} - 1$, yaklaşık $1{,}8 \cdot 10^{19}$ hamledir; saniyede bir hamleyle yaklaşık 585 milyar yıl sürer.

4.5.3 Kod izleme: hanoi(3, 'A', 'B', 'C')

Plan Java'ya doğrudan çevrilir. Metot dört parametre alır: disk sayısı `n`, kaynak çubuk `k`, yardımcı çubuk `y` ve hedef çubuk `h`.

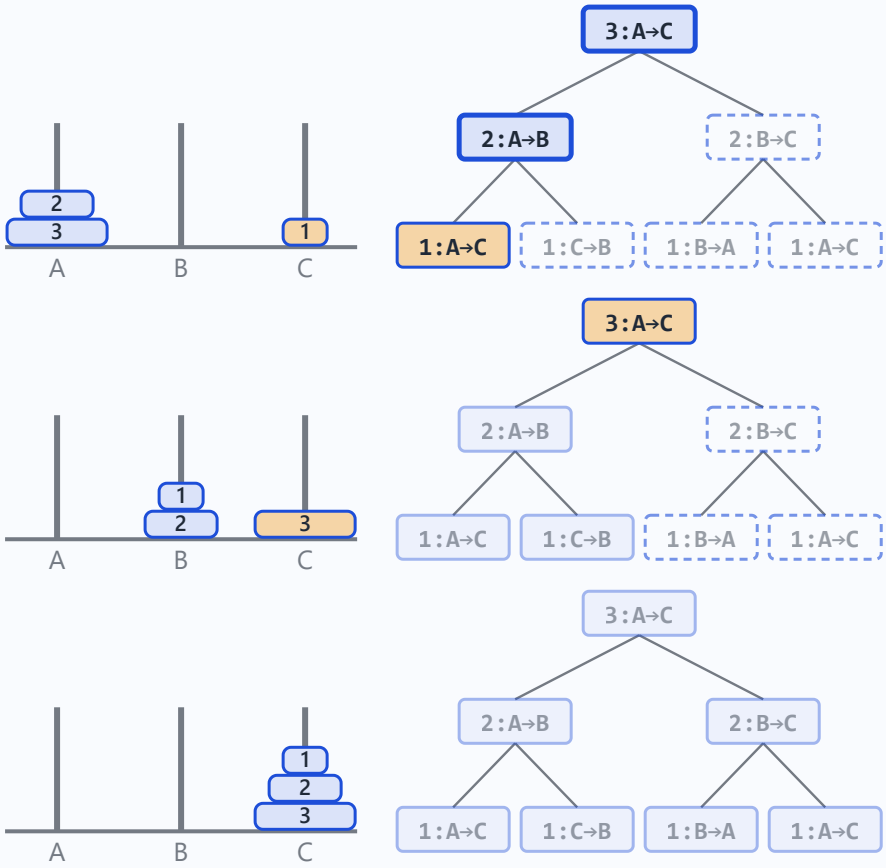
```
void hanoi(int n, char k, char y, char h) {
  if (n == 1) {
    System.out.println("1: " + k + "→" + h);
    return;
  }
  hanoi(n - 1, k, h, y);
  System.out.println(n + ": " + k + "→" + h);
  hanoi(n - 1, y, k, h);
}
// main içinde:
hanoi(3, 'A', 'B', 'C');
```

2–5. satırlar taban durumdur: tek disk kaynaktan hedefe taşınır ve hamle "`1: A→C`" biçiminde yazdırılır. Baştaki sayı, taşınan diskin numarasıdır. 6. satır planın 1. adımıdır: *n − 1* disk kaynaktan yardımcıya taşınır. Bu çağrıda hedefin (`h`) yardımcı, yardımcının (`y`) hedef rolüne geçtiğine dikkat edilmelidir: `hanoi(n - 1, k, h, y)`. 7. satır en büyük diski, yani `n` numaralı diski taşır. 8. satır 3. adımdır: *n − 1* disk yardımcıdan hedefe taşınır; bu kez kaynak yardımcı kullanılır: `hanoi(n - 1, y, k, h)`.

Adım	Çalışan çağrı	Olay	Yazılan
1	`H(3,A,B,C)`	*n ≠ 1*; satır 6: önce 2 disk A'dan B'ye	
2	`H(2,A,C,B)`	*n ≠ 1*; satır 6: önce 1 disk A'dan C'ye	
3	`H(1,A,B,C)`	taban durum	`1: A→C`
4	`H(2,A,C,B)`	satır 7: disk 2	`2: A→B`
5	`H(1,C,A,B)`	satır 8'den çağrıldı; taban durum	`1: C→B`
6	`H(3,A,B,C)`	`H(2,A,C,B)` bitti; satır 7: disk 3	`3: A→C`
7	`H(2,B,A,C)`	satır 8'den çağrıldı; satır 6: önce 1 disk B'den A'ya	
8	`H(1,B,C,A)`	taban durum	`1: B→A`
9	`H(2,B,A,C)`	satır 7: disk 2	`2: B→C`
10	`H(1,A,B,C)`	satır 8'den çağrıldı; taban durum	`1: A→C`
11	`H(3,A,B,C)`	alt çağrılar bitti; `main`'e döner	

Tablo 4.11: `hanoi(3, 'A', 'B', 'C')` izlemesi. `H(n,k,y,h)`, `hanoi` çağrısının kısaltmasıdır.

Yazılan yedi satır, Şekil 4.17'deki yedi hamlenin aynısıdır. Çağrı ağacında 7 düğüm vardır: bir `H(3,…)`, iki `H(2,…)` ve dört `H(1,…)`. Her düğüm tam olarak bir satır yazar; çağrı sayısı ile hamle sayısı eşittir: $2^3 - 1 = 7$.



Şekil 4.18: `hanoi(3, 'A', 'B', 'C')` çağrı ağacı ve çubukların durumu. Ağaçtaki her kutu bir çağrıdır ve `n:kaynak→hedef` biçiminde, o çağrının yazdırdığı hamleyi gösterir. Üstte ilk hamle (`1:A→C`) yapılmış; kalın çerçeveli kutular yığındaki çağrılardır. Ortada sol alt ağaç bitmiş, küçük iki disk B'de toplanmış ve disk 3 A'dan C'ye taşınmıştır. Altta bütün çağrılar bitmiş, üç disk C'dedir.

Maliyet: zaman $O(2^n)$'dir, çünkü $2^n - 1$ çağrının her biri sabit iş yapar. Bellek ise $O(n)$'dir. Yığında aynı anda en çok n `hanoi` çerçevesi bulunur (n, $n-1$, ..., 1 disklik çağrılar) ve `main` ile $n + 1$ çerçeve olur.

Sık yapılan hata

Özyinelemeli çağrılarda çubukların sırasını karıştırmak. 6. satırda `hanoi(n - 1, k, y, h)` yazılırsa küçük diskler hedefe taşınır ve en büyük disk hedefte kendinden küçük disklerin üstüne konmaya çalışılır; kurallar bozulur. Parametreleri doğru yazmanın bir yolu, her çağrı için "nereden, nereye, hangisi boş kalıyor?" sorusunu cevaplamaktır: 1. adımda $n - 1$ disk `k` 'dan `y` 'ye gider, `h` boştur.

Çıktıda bir düzen daha göze çarpar. Disk 1 her ikinci hamlede, yani tek numaralı hamlelerde oynar; n diskte toplam 2^{n-1} kez taşınır. Disk k ise 2^{n-k} kez taşınır; en büyük disk yalnız bir kez. Bu, ağaçta her düzeydeki düğüm sayısıyla aynıdır: en alt düzeyde (tek disklik çağrılar) 2^{n-1} düğüm vardır. Hanoi ku-

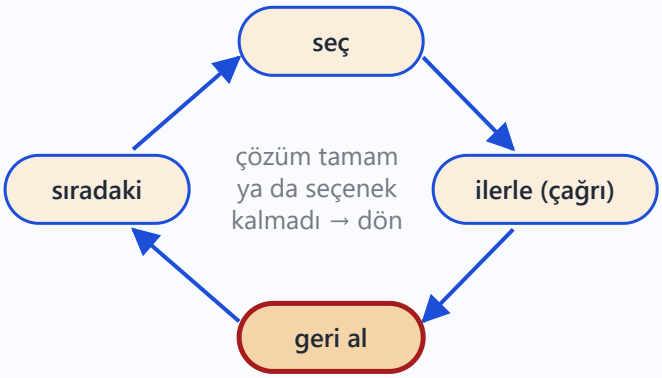
leleri simülatöründe bulmaca 1–6 diskle elle çözülebilir ya da en kısa çözüm adım adım izlenebilir. Simülatör yapılan hamle sayısını en az hamle sayısıyla ($2^n - 1$; 6 disk için 63) karşılaştırır. Her kule bir yığın gibi davranır: yalnız en üstteki disk alınabilir.

4.5.4 Geri izleme

Bazı problemlerde cevap tek bir değer değil, bir dizi seçimin sonucudur. Labirentte her kavşakta hangi yöne gidileceği, sudoku tablosunda her boş kareye hangi rakamın yazılacağı, kümeden hangi öğelerin seçileceği böyledir. Bu problemlerin genel çözüm yöntemine **geri izleme** denir.

Tanım 4.5

Geri izleme (*backtracking*), çözümü adım adım kuran bir yöntemdir. Her adımda olası seçeneklerden biri denenir ve özyinelemeli bir çağrıyla bir sonraki adıma geçilir. Çağrı döndüğünde yapılan seçim **geri alınır** ve sıradaki seçenek denenir. Bir seçimin çözüme götüremeyeceği anlaşılırsa o dal bırakılır, daha derine inilmez.



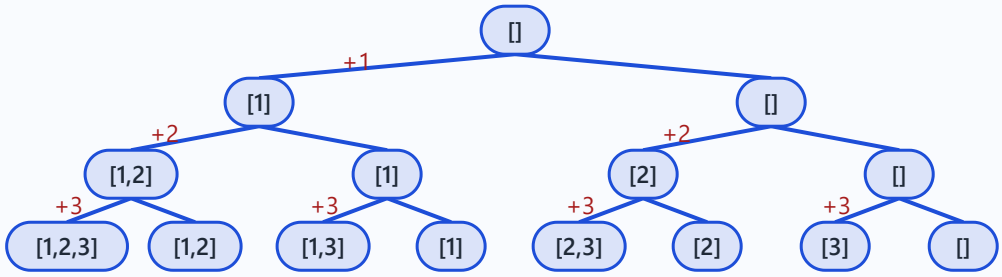
Şekil 4.19: Geri izleme döngüsü. Bir seçenek seçilir, özyinelemeli çağrıyla ilerlenir, çağrı dönünce seçim geri alınır ve sıradaki seçeneğe geçilir. Çözüm tamamlandığında ya da seçenek kalmadığında çağrı döner.

Seçimi geri almak neden gereklidir? Geri izlemede kurulmakta olan çözüm (örneğin bir liste) genellikle bütün çağrılar arasında paylaşılan tek bir nesnedir. Bir dalda yapılan değişiklik geri alınmazsa sonraki dal bu değişikliği hazır bulur ve yanlış bir durumdan başlar. Seçim geri alınınca her dal, aynı başlangıç durumunu görür.

Hanoi'de olduğu gibi geri izlemede de her çağrı birden çok özyinelemeli çağrı yapar. Çağrı ağacı, denenen bütün seçimlerin ağacıdır; buna **karar ağacı** da denir. Kökten bir yaprağa giden her yol, bir seçimler dizisidir. Geri izlemenin bilinen örnekleri şunlardır: alt kümeleri ya da permütasyonları listelemek, labirentte yol bulmak, sudoku çözmek ve n-vezir problemi. Son problemde bir satranç tahtasına birbirini tehdit etmeyen n vezir yerleştirilir.

4.5.5 Alt kümeleri listeleme

Bir kümenin bütün alt kümelerini yazdırmak isteyelim. *{1, 2, 3}* kümesinin 8 alt kümesi vardır: *{1, 2, 3}*, *{1, 2}*, *{1, 3}*, *{1}*, *{2, 3}*, *{2}*, *{3}* ve boş küme. Bunları sistemli üretmenin yolu, her öğe için tek tek karar vermektir: bu öğe alt kümeye alınacak mı, alınmayacak mı? Her öğe için iki seçenek vardır ve n öğe için kararlar birbirinden bağımsızdır. Toplam $2 \cdot 2 \cdot \ldots \cdot 2 = 2^n$ farklı karar dizisi, yani 2^n alt küme vardır.



Şekil 4.20: *{1, 2, 3}* için karar ağacı. Her düzeyde bir öğe için karar verilir: sol dal öğeyi alır (`+1` , `+2` , `+3`), sağ dal almaz. Düğümlerdeki listeler o ana kadar seçilen öğelerdir. 8 yaprak, 8 alt kümedir.

Karar ağacı doğrudan bir özyinelemeli metoda dönüşür. Metot üç parametre alır: dizi `a` , karar verilecek öğenin indisi `i` ve o ana kadar seçilen öğeleri tutan liste `s` .

```
void altKumeler(int[] a, int i, List<Integer> s) {
  if (i == a.length) {              // taban durum
    System.out.print(s + " "); return;
  }
  s.add(a[i]);                      // a[i]'yi al
  altKumeler(a, i + 1, s);
  s.remove(s.size() - 1);           // geri al
  altKumeler(a, i + 1, s);          // a[i]'siz dene
}
// main içinde:
altKumeler(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

Metodun her parçası karar ağacındaki bir öğeye karşılık gelir:

- **Taban durum** (2–4. satırlar): `i == a.length` ise bütün öğeler için karar verilmiştir; ağacın bir yaprağındayız. `s` bir alt kümedir ve yazdırılır.
- **Sol dal** (5–6. satırlar): `a[i]` listeye eklenir ve bir sonraki öğe için karar verilir.
- **Geri alma** (7. satır): sol daldan dönülünce `a[i]` listeden çıkarılır. Liste, bu çağrının başındaki hâline döner.
- **Sağ dal** (8. satır): `a[i]` alınmadan bir sonraki öğeye geçilir.

`s.remove(s.size() - 1)` listenin *son* öğesini indisle siler. Son öğe her zaman bu çağrının eklediği `a[i]`'dir, çünkü alt çağrılar ekledikleri her şeyi kendileri geri almıştır. `List<Integer>` için `s.remove(a[i])` yazmak tehlikelidir: `a[i]` bir `int` olduğundan Java bunu indis olarak yorumlar ve yanlış öğeyi siler.

Çağrı sayısı: karar ağacı tam bir ikili ağaçtır. Düzey k'de 2^k düğüm vardır ($k = 0, 1, \ldots, n$). Toplam $1 + 2 + \ldots + 2^n = 2^{n+1} - 1$ çağrı yapılır. $n = 3$ için 8 alt küme ve 15 çağrı. Her yaprakta alt küme yazdırılırken $O(n)$ iş yapıldığı için toplam zaman $O(n \cdot 2^n)$'dir. Yığın derinliği $n + 1$ çerçevedir; bellek $O(n)$'dir.

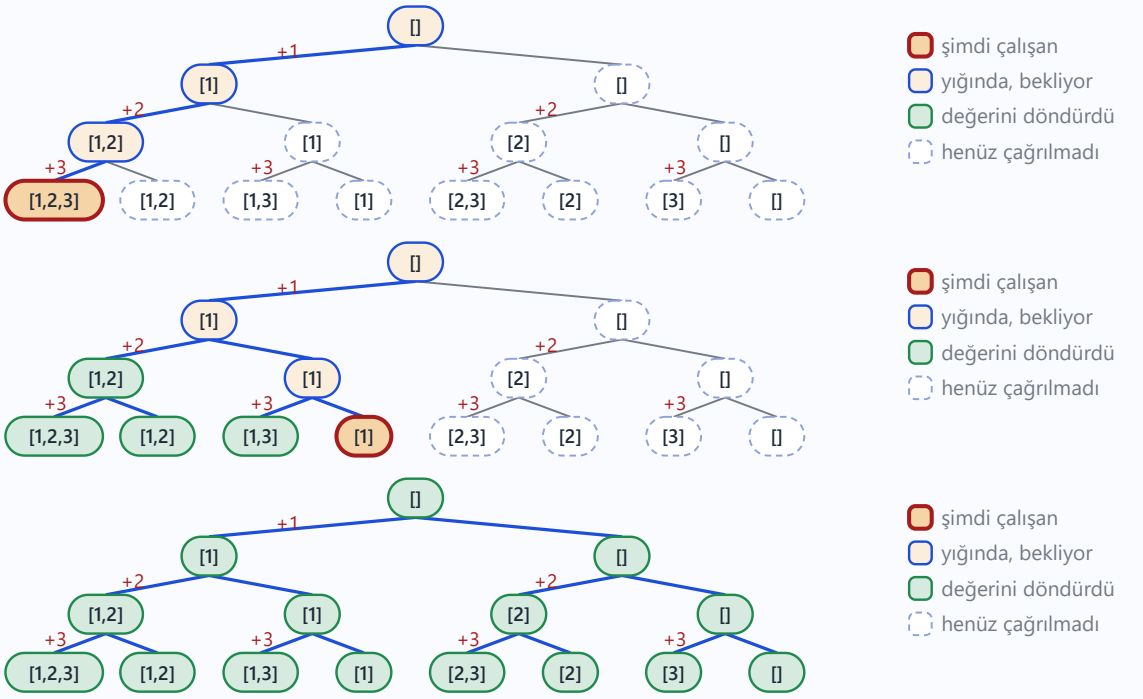
4.5.6 Kod izleme: altKumeler ile {1, 2, 3}

`main`, `altKumeler(new int[]{1, 2, 3}, 0, new ArrayList<>())` çağrısını yapıyor. Tablo 4.12, 15 çağrıyı başlama sırasıyla verir. Her satırda çağrının `i` değeri, çağrı başladığı andaki `s` ve çağrının ilk yaptığı iş yazılıdır.

Çağrı	`i`	girişte `s`	Olay
1	0	[]	1 alınır, sol dala inilir
2	1	[1]	2 alınır, sol dala inilir
3	2	[1, 2]	3 alınır, sol dala inilir
4	3	[1, 2, 3]	taban durum: **[1, 2, 3]** yazılır
5	3	[1, 2]	(3. çağrı 3'ü geri aldı) taban durum: **[1, 2]**
6	2	[1]	(2. çağrı 2'yi geri aldı) 3 alınır
7	3	[1, 3]	taban durum: **[1, 3]**
8	3	[1]	(3 geri alındı) taban durum: **[1]**
9	1	[]	(1. çağrı 1'i geri aldı) 2 alınır
10	2	[2]	3 alınır
11	3	[2, 3]	taban durum: **[2, 3]**
12	3	[2]	(3 geri alındı) taban durum: **[2]**
13	2	[]	(2 geri alındı) 3 alınır
14	3	[3]	taban durum: **[3]**
15	3	[]	(3 geri alındı) taban durum: **[]**

Tablo 4.12: `altKumeler` izlemesi, `a = {1, 2, 3}`.

Çıktı sırası şöyledir: `[1, 2, 3] [1, 2] [1, 3] [1] [2, 3] [2] [3] []`. Sıra, karar ağacının yapraklarının soldan sağa sırasıdır; çünkü her çağrı önce sol dalı (“al”), sonra sağ dalı (“alma”) bütünüyle gezer. “Al” ve “alma” dallarının sırası değiştirilirse yapraklar ters sırayla gelir; alt kümeler aynı kalır.



Şekil 4.21: `altKumeler` karar ağacında üç an. Üstte ilk yaprak (Tablo 4.12, 4. çağrı): `[1, 2, 3]` yazılıyor; kökten bu yaprağa giden yol yığındadır. Ortada 8. çağrı: `[1]` yazılıyor; sol alt ağacın bütün yaprakları bitmiştir. Altta son durum: 15 çağrının hepsi döndü.

Yığında aynı anda en çok `main` ve 4 `altKumeler` çerçevesi vardır (kökten bir yaprağa giden yol: *i = 0, 1, 2, 3*).

Sık yapılan hata

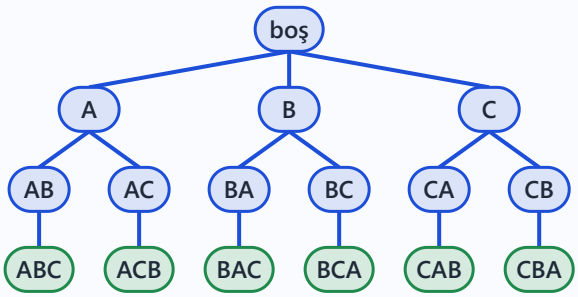
Geri almayı unutmak. 7. satır silinirse sol dalda eklenen öğe listede kalır ve sağ dala taşar. *{1, 2}* için çıktı `[1, 2] [1, 2] [1, 2, 2] [1, 2, 2]` olur: hiçbir seçim geri alınmaz, 2 ikinci kez eklenir ve "1'siz" alt kümeler hiç görünmez. Paylaşılan bir nesneyi değiştiren her seçim, çağrı dönünce geri alınmalıdır.

4.5.7 Permütasyonları listeleme

Bir dizi harfin bütün sıralanışlarını, yani **permütasyonlarını** yazdırmak isteyelim. A, B, C harflerinin 6 permütasyonu vardır: ABC, ACB, BAC, BCA, CAB, CBA. Geri izlemeyle çözüm şöyle kurulur: kelimenin ilk harfi için 3 seçenek vardır; ilk harf seçildikten sonra ikinci harf için kalan 2 harften biri seçilir; üçüncü harf için tek seçenek kalır. Toplam $3 \cdot 2 \cdot 1 = 3! = 6$ permütasyon çıkar.

```
void perm(char[] h, boolean[] kul, StringBuilder s) {
  if (s.length() == h.length) {  // taban durum
    System.out.print(s + " "); return;
  }
  for (int k = 0; k < h.length; k++) {
    if (kul[k]) continue;        // harf kullanıldı
    kul[k] = true; s.append(h[k]);   // seç
    perm(h, kul, s);
    kul[k] = false;              // geri al
    s.deleteCharAt(s.length() - 1);
  }
}
```

Metodun üç parametresi vardır. `h` harfleri tutar. `kul` dizisi hangi harflerin o ana kadar kullanıldığını gösterir: `kul[k]` doğruysa `h[k]` kelimede vardır. `s`, kurulmakta olan kelimedir. Kelime harf sayısı uzunluğuna ulaşınca (2. satır) bir permütasyon tamamlanmıştır ve yazdırılır. Öteki durumda her kullanılmamış harf sırayla denenir: harf kullanıldı olarak işaretlenir ve kelimenin sonuna eklenir (7. satır); kalan harfler için özyinelemeli çağrı yapılır (8. satır); çağrı dönünce işaret kaldırılır ve harf kelimeden silinir (9–10. satırlar).



Şekil 4.22: A, B, C için permütasyon ağacı. Kökte kelime boştur; birinci düzeyde 3, ikincide 6, üçüncüde 6 düğüm vardır. Yapraklar (yeşil) soldan sağa ABC, ACB, BAC, BCA, CAB, CBA'dır. Toplam *1 + 3 + 6 + 6 = 16* çağrı.

Bu metotta bir çağrı, kalan harf sayısı kadar özyinelemeli çağrı yapar: kökteki çağrı 3, ikinci düzeydekiler 2'şer, üçüncü düzeydekiler 1'er. Çağrı sayısı sabit değil, değişkendir; bu tür özyinelemeye **çoklu özyineleme** denir. n harf için yaprak sayısı $n!$'dir. Ağaçtaki toplam düğüm sayısı $n!$'in en çok yaklaşık 2,72 katıdır ($n = 3$ için 16 düğüm, 6 yaprak). Her düğüm $O(n)$ adımlık bir `for` döngüsü çalıştırır ve her yaprakta n harflik bir kelime yazdırılır. Zaman bu yüzden $O(n \cdot n!)$'dir. Yığın derinliği $n + 1$'dir; bellek $O(n)$.

Örnek 4.6

A, B, C için ilk iki permütasyonun nasıl üretildiğini izleyelim.

1. Kök: `s = ""`. `k = 0`: A kullanılmamış; işaretlenir, `s = "A"`, çağrı.

2. `s = "A"` : `k = 0` atlanır (A kullanıldı). `k = 1` : B seçilir, `s = "AB"` , çağrı.
3. `s = "AB"` : yalnız C kullanılmamış; `s = "ABC"` , çağrı.
4. `s = "ABC"` : uzunluk 3, **ABC** yazılır, dönülür.
5. `s = "AB"` çağrısında C geri alınır (`s = "AB"`); başka harf yoktur, dönülür.
6. `s = "A"` çağrısında B geri alınır (`s = "A"`). `k = 2` : C seçilir, `s = "AC"` ; ardından B eklenir ve **ACB** yazılır.

Sonra kökte A geri alınır ve ilk harf olarak B denenir; BAC ve BCA üretilir. En son C ile CAB ve CBA gelir.

Sık yapılan hata

`kul[k] = false;` satırını unutmak. İşaret kaldırılmazsa bir harf, bir dalda kullanıldıktan sonra öteki dallarda da kullanılmış sayılır. A, B, C için yalnız ABC yazılır; sonraki denemelerde kullanılmamış harf kalmadığı için hiçbir kelime tamamlanmaz. `kul` ve `s` bütün çağrılarca paylaşıldığı için her denemeden sonra iki değişiklik de geri alınmalıdır.

4.6 Yineleme bağıntıları

Önceki kesimlerde her özyinelemeli metodun maliyetini çağrıları tek tek sayarak bulduk. Bu yöntem küçük girdilerde işe yarar, ama genel bir kural vermez. Daha genel bir yol, maliyeti bir denklemle yazıp bu denklemi çağrı ağacıyla çözmektir. Bölümde geçen bütün metotlar beş denklem tipinden birine uyar (Tablo 4.13).

4.6.1 Yineleme bağıntısı nedir?

T(n), bir metodun *n* boyutlu girdide yaptığı toplam iş olsun. Özyinelemeli bir metotta bu iş iki parçadan oluşur: metodun kendi içinde yaptığı iş ve özyinelemeli çağrıların yaptığı iş. İkinci parça, aynı metodun daha küçük girdilerdeki maliyetidir. Böylece *T(n)*, *T*'nin daha küçük değerleri cinsinden yazılır.

Tanım 4.6

Yineleme bağıntısı (*recurrence relation*), bir fonksiyonun değerini aynı fonksiyonun daha küçük girdilerdeki değerleriyle veren denklemdir. Özyinelemeli bir metodun maliyeti *T(n)*, bir taban değer (örneğin *T(0) = c*) ve bir yineleme bağıntısıyla tanımlanır.

Bağıntıyı yazmak için metoda bakıp üç soru cevaplanır:

1. Taban durumun maliyeti nedir? Çoğunlukla bir sabittir; *c* ile gösterilir.

2. Kaç özyinelemeli çağrı yapılıyor ve girdileri ne kadar küçük?
3. Çağrılar dışında metodun kendisi ne kadar iş yapıyor?

Örneğin `hizliUs` bir çağrı yapar, girdisi *n / 2*'dir ve çağrı dışında bir karşılaştırma ile en çok iki çarpma yapar (sabit iş). Bağıntısı *T(n) = T(n/2) + c*'dir. Bölümdeki metotların bağıntıları Tablo 4.13'te toplanmıştır.

Metot	Çağrılar	Yineleme bağıntısı	Çözüm
`faktoriyel`, `toplam`, `geriSay`	bir çağrı, *n − 1*	*T(n) = T(n − 1) + c*	*O(n)*
`ters`	bir çağrı, *n − 2*	*T(n) = T(n − 2) + c*	*O(n)*
`ikiliAra`, `hizliUs`	bir çağrı, *n / 2*	*T(n) = T(n/2) + c*	*O(log n)*
`hanoi`	iki çağrı, *n − 1*	*T(n) = 2T(n − 1) + c*	$O(2^n)$
`fib`	iki çağrı, *n − 1* ve *n − 2*	*T(n) = T(n − 1) + T(n − 2) + c*	üstel

Tablo 4.13: Bölümdeki metotların yineleme bağıntıları.

Bağıntıyı çözmenin en kolay yolu çağrı ağacını çizmektir. Her düğüm bir çağrıdır ve kendi içinde *c* kadar iş yapar. Toplam iş, düğüm sayısı ile *c*'nin çarpımıdır: *T(n) ≈ c ·* (çağrı sayısı). Bağıntıyı çözmek, çağrı ağacındaki düğümleri saymaya indirgenir.

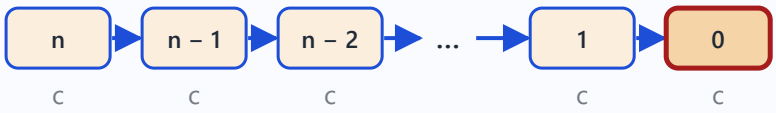
> **Sık yapılan hata**
>
> Bağıntıyı koddaki çağrı yazılışlarını sayarak kurmak. `ikiliAra`'nın kodunda iki özyinelemeli çağrı yazılıdır, ama her çağrı bunlardan yalnız birini yapar. Bağıntı *2T(n/2) + c* değil, *T(n/2) + c*'dir. Bağıntı, bir çağrıda *gerçekten yapılan* çağrıları sayar.

4.6.2 T(n) = T(n − 1) + c

Bu bağıntı, girdiyi her adımda 1 azaltan doğrusal özyinelemeye aittir. Çözmek için bağıntı kendi içine tekrar tekrar yerleştirilir; buna bağıntıyı **açmak** denir:

```
T(n) = T(n − 1) + c
     = T(n − 2) + 2c
     = T(n − 3) + 3c
     …
     = T(0) + n · c
     = c + n · c   →  O(n)
```

Her açılışta bir *c* daha eklenir ve *n* 1 azalır. *k* açılıştan sonra *T(n) = T(n − k) + k · c* olur. *k = n* olunca taban duruma inilir: *T(n) = T(0) + n · c = c + n · c*. Bu *O(n)*'dir.



Şekil 4.23: *T(n) = T(n − 1) + c* çağrı ağacı: dallanmayan bir zincir. *n, n − 1, ..., 1, 0* girdili *n + 1* çağrı vardır ve her biri *c* iş yapar.

Çağrı ağacı burada düz bir zincirdir (Şekil 4.23). Zincirde *n + 1* çağrı vardır ve zincirin tamamı aynı anda yığındadır: ilk çağrı, son çağrı dönmeden bitemez. Bellek de bu yüzden *O(n)*'dir. `faktoriyel`, `toplam`, `geriSay`, yavaş üs alma ve bağlı listenin uzunluğu bu bağıntıya uyar.

`ters` için girdi, yani parçanın uzunluğu, her adımda 2 azalır: *T(n) = T(n − 2) + c*. Açılınca *T(n) = T(n − 2k) + k · c* çıkar ve taban duruma yaklaşık *k = n/2* adımda inilir. Çağrı sayısı yarıya iner, ama *n/2* de *n* ile doğru orantılı olduğundan sonuç yine *O(n)*'dir. Girdiyi sabit bir *miktar* azaltmak, bu miktar ne olursa olsun, doğrusal bir zincir verir.

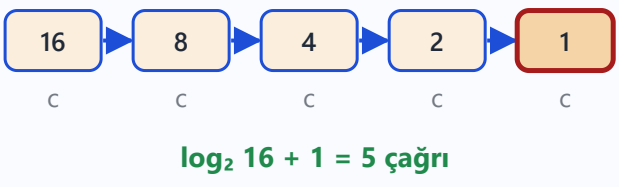
4.6.3 T(n) = T(n/2) + c

Bu bağıntı, girdiyi her adımda yarıya indiren özyinelemeye aittir. Açalım:

```
T(n) = T(n/2) + c
     = T(n/4) + 2c
     = T(n/8) + 3c
     …
     = T(n/2^k) + k · c
```

k açılıştan sonra girdi $n/2^k$'dir. Taban duruma girdi 1 olunca ulaşılır: $n/2^k = 1$, yani $2^k = n$, yani $k = \log_2 n$. Bu değer yerine konunca $T(n) = T(1) + c \cdot \log_2 n$ bulunur. Bu *O(log n)*'dir.

Logaritma burada tek bir soruya cevaptır: *n*'yi kaç kez yarıya bölersek 1'e ulaşırız? 16 → 8 → 4 → 2 → 1 dizisinde 4 bölme vardır ve $\log_2 16 = 4$'tür. *n* ikinin kuvveti değilse tam sayı bölmesi aşağı yuvarlar ve bölme sayısı $\lfloor \log_2 n \rfloor$ olur.



Şekil 4.24: *n = 16* için yarılama zinciri: 16, 8, 4, 2, 1. Her çağrı *c* iş yapar; $\log_2 16 + 1 = 5$ çağrı.

n	16	1.024	1.000.000
çağrı	5	11	20

Tablo 4.14: Yarılama zincirinin uzunluğu (*n*'den 1'e, iki uç dahil).

Girdi bin kat büyüyünce (1.024'ten yaklaşık bir milyona) zincir yalnızca yaklaşık 10 çağrı uzar. Logaritmik büyüme bu kadar yavaştır. Zincirin derinliği de *log n* olduğu için bellek *O(log n)*'dir. İkili arama ve hızlı üs alma bu bağıntıya uyar. (Gerçek metotlarda taban durum 0 ya da boş aralık olabilir; o zaman zincire bir çağrı daha eklenir. `hizliUs` 'un $\lfloor \log_2 n \rfloor + 2$ çağrısı bu yüzdendir.)

Örnek 4.7

T(n) = T(n/2) + c, *T(1) = c* bağıntısını *n = 100* için çağrı sayarak çözelim. Java'nın tam sayı bölmesi kullanılsın.

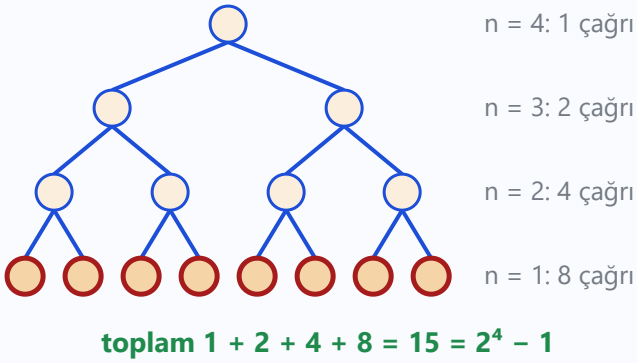
1. Zincir: 100, 50, 25, 12, 6, 3, 1. *25 / 2 = 12* ve *3 / 2 = 1*, aşağı yuvarlamanın sonucudur.
2. Çağrı sayısı 7'dir. $\lfloor \log_2 100 \rfloor = 6$ ($2^6 = 64 \le 100 < 128 = 2^7$), buna 1'in çağrısı eklenir: *6 + 1 = 7*.
3. *T(100) = 7c*.

4.6.4 T(n) = 2T(n − 1) + c

Bu bağıntıda her çağrı, *n − 1* boyutlu *iki* çağrı yapar. Çağrı ağacının her düzeyinde çağrı sayısı ikiye katlanır. Kökte (düzey 0) 1 çağrı, düzey 1'de 2 çağrı, düzey 2'de 4 çağrı vardır; düzey *k*'de 2^k çağrı. Taban durum *n = 1* ise düzeyler 0'dan *n − 1*'e kadar gider.

> $1 + 2 + 4 + \ldots + 2^{n-1} = 2^n - 1$ çağrı → $O(2^n)$

Bu toplamın değeri, ikinin kuvvetlerinin bilinen bir özelliğinden gelir: ikinin ardışık kuvvetlerinin toplamı, bir sonraki kuvvetin bir eksiğidir. Örneğin *1 + 2 + 4 + 8 = 15 = 16 − 1*. İkili sayılarla düşünülünce bu açıktır: `1111` ikili sayısı 15'tir ve 1 eklenince `10000`, yani 16 olur.



Şekil 4.25: *$T(n) = 2T(n-1) + c$ çağrı ağacı, $n = 4$.* Düzeylerde 1, 2, 4 ve 8 çağrı vardır; toplam *$1 + 2 + 4 + 8 = 15 = 2^4 - 1$.* En alttaki 8 düğüm (kırmızı) taban durumdur.

Hanoi kuleleri bu bağıntıya tam uyar: $2^n - 1$ hamle. $n = 20$ için 1.048.575, $n = 64$ için yaklaşık $1{,}8 \cdot 10^{19}$ hamle.

`fib`'in bağıntısı $T(n) = T(n-1) + T(n-2) + c$'dir. $T(n-2) \le T(n-1)$ olduğu için $T(n) \le 2T(n-1) + c$ yazılabilir. Buradan `fib` için üst sınır olarak $O(2^n)$ çıkar. Gerçek büyüme bundan biraz yavaştır: 4.4.3'te gösterildiği gibi yaklaşık $1{,}6^n$. Ama 1,6 tabanı da üsteldir; $n = 40$ için 331 milyon çağrı bunun sonucudur.

Bu ağaç ne kadar büyük olursa olsun derinliği yalnız n'dir. Yığında aynı anda en çok n çerçeve bulunur. Zaman üstel, bellek doğrusaldır.

Sık yapılan hata

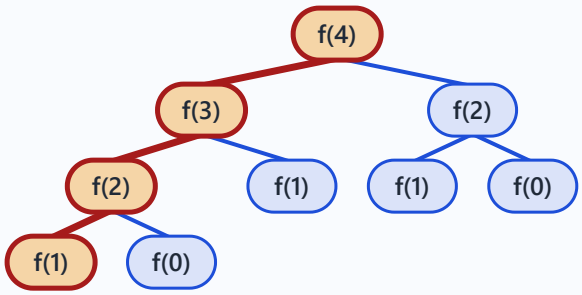
$2T(n-1)$ ile $2T(n/2)$'yi karıştırmak. İkisi de iki çağrı yapar, ama sonuçları çok farklıdır. $T(n) = 2T(n/2) + c$'de ağacın derinliği $\log_2 n$'dir ve son düzeyde $2^{\log n} = n$ düğüm vardır; toplam yaklaşık $2n$ çağrı, yani $O(n)$. $T(n) = 2T(n-1) + c$'de ise derinlik n'dir ve toplam $O(2^n)$. Çağrı sayısı kadar, girdinin *ne kadar* küçüldüğü de önemlidir.

4.6.5 Alan maliyeti: en derin çağrı zinciri

Zaman maliyeti çağrı ağacındaki *bütün* düğümlerden gelir. Alan maliyeti ise başka bir ölçüye bağlıdır. Bir çağrı, kardeşi başlamadan önce tamamen biter. Örneğin `fib(4)`'te `fib(3)`'ün bütün alt ağacı bitmeden `fib(2)` başlamaz. Bu yüzden yığında aynı anda yalnız kökten bir düğüme giden tek bir yol bulunur. Bitmiş çağrıların çerçeveleri silinmiştir, henüz başlamamış çağrıların çerçeveleri ise yoktur.

> Bellek = en derin çağrı zinciri × çerçeve boyutu

Toplam çağrı sayısı değil, ağacın derinliği belirleyicidir. `fib(4)` 9 çağrı yapar, ama yığında aynı anda en çok 4 `fib` çerçevesi (ve `main`) bulunur.



Şekil 4.26: `fib(4)` çağrı ağacında en derin zincir (kırmızı): `fib(4)` → `fib(3)` → `fib(2)` → `fib(1)`. Yığın hiçbir anda bu yoldan daha uzun olmaz.

Çağrı	Toplam çağrı	En derin zincir	Zaman	Alan
`faktoriyel(4)`	4	4	$O(n)$	$O(n)$
`fib(40)`	331.160.281	40	üstel	$O(n)$
`hanoi(3, …)`	7	3	$O(2^n)$	$O(n)$
`altKumeler`, $n = 3$	15	4	$O(n \cdot 2^n)$	$O(n)$

Tablo 4.15: Toplam çağrı sayısı ile en derin zincirin karşılaştırılması.

Tablo 4.15 çarpıcı bir farkı gösterir. `fib(40)` üç yüz milyondan fazla çağrı yapar ve sonucu beklemek fark edilir bir süre alır, ama belleği hiç zorlamaz: yığın en çok 40 çerçeve derinliğe iner. Tersine, `topla(1_000_000)` yalnız bir milyon çağrı yapar ve bu zaman açısından hiç sorun değildir; ama bir milyon çerçevelik zincir yığını taşırır (4.2.3). Özyinelemede zaman sorunu ile bellek sorunu birbirinden bağımsızdır ve ayrı ayrı denetlenmelidir.

`altKumeler` satırındaki zaman, yalnız çağrı sayısından gelmez. Çağrı sayısı $2^{n+1} - 1$'dir, ama 2^n yaprağın her birinde n'ye kadar öğeli bir liste yazdırılır. Yazdırmanın maliyeti eklenince toplam $O(n \cdot 2^n)$ olur.

4.7 Kuyruk özyinelemesi

Özyinelemeli metotların büyük bir kısmı dönüş yolunda iş yapar: `faktoriyel` çarpmayı, `toplam` toplamayı, `ileriSay` yazdırmayı dönüşte yapar. Bu yüzden çerçeveler, alt çağrılar bitene kadar bellekte beklemek zorundadır. Bazı metotlarda ise dönüşte yapılacak hiçbir iş yoktur. Böyle metotlar özeldir: döngüye kolayca çevrilir.

4.7.1 Kuyruk özyinelemesi nedir?

Tanım 4.7

Özyinelemeli çağrı metodun son işiyse, yani çağrı döndükten sonra yapılacak bir iş kalmıyorsa, metoda **kuyruk özyinelemeli** denir (*tail recursion*). Böyle bir çağrıya **kuyruk çağrısı** denir: metot, çağrının döndürdüğü değeri hiç değiştirmeden döndürür.

`faktoriyel` kuyruk özyinelemeli değildir:

```
int faktoriyel(int n) {
  if (n <= 1) return 1;
  return n * faktoriyel(n - 1);
  // çarpma, çağrı dönünce yapılır
}
```

`return n * faktoriyel(n - 1);` satırında çağrı son iş gibi görünür, ama değildir. Çağrı döndükten sonra sonucun `n` ile çarpılması gerekir. Son iş çarpmadır.

Faktöriyeli kuyruk özyinelemeli yazmak için çarpmayı çağrıdan *önceye* almak gerekir. Bunun yolu, ara sonucu bir parametreyle aşağı taşımaktır. Bu parametreye **biriktirici** (*accumulator*) denir ve geleneksel olarak `acc` diye adlandırılır.

```
int faktKuyruk(int n, int acc) {
  if (n <= 1) return acc;
  return faktKuyruk(n - 1, n * acc);
  // son iş: çağrı (çarpma önce)
}
```

`acc`, o ana kadar yapılmış çarpmaların sonucunu tutar. Her çağrı kendi `n`'sini `acc` ile çarpar ve sonucu bir sonraki çağrıya verir. Taban durumda bütün çarpmalar yapılmıştır; `acc` sonucun kendisidir ve döndürülür. İlk çağrı `faktKuyruk(n, 1)` biçiminde yapılır, çünkü henüz hiçbir çarpma yapılmamıştır ve 1 çarpmanın etkisiz öğesidir.

Bölümdeki metotlar bu açıdan ikiye ayrılır:

- **Kuyruk özyinelemeli:** `geriSay` (yazdırma çağrıdan önce), `ters` (değiştirme çağrıdan önce), `ikiliAra` (dönen değer olduğu gibi iletilir), `faktKuyruk`.
- **Kuyruk özyinelemeli değil:** `faktoriyel` ve `toplam` (çağrıdan sonra çarpma ya da toplama), `fib` (iki çağrının sonucu toplanır), `ileriSay` (çağrıdan sonra yazdırma).

Sık yapılan hata

Çağrı `return` satırında diye metodu kuyruk özyinelemeli saymak. Ölçüt, çağrının satırdaki yeri değil, çağrı döndükten sonra iş kalıp kalmadığıdır. `return f(n - 1) + 1;` satırında çağrı döndükten sonra 1 eklenir; metot kuyruk özyinelemeli değildir. `return g(n - 1, acc + n);` ise kuyruk çağrısıdır: toplama çağrıdan *önce*, parametre hesaplanırken yapılır.

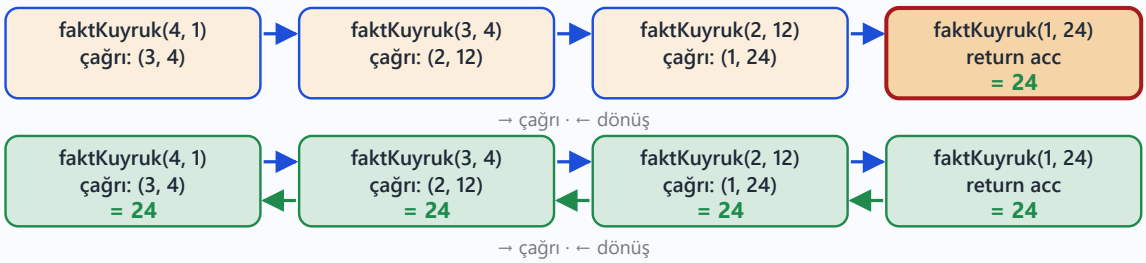
4.7.2 Kod izleme: faktKuyruk(4, 1)

```
int faktKuyruk(int n, int acc) {
  if (n <= 1) return acc;   // taban durum
  return faktKuyruk(n - 1, n * acc);  // son iş
}
// main içinde:
System.out.println(faktKuyruk(4, 1));
```

Adım	Çağrı	Olay	Dönen
1	`faktKuyruk(4, 1)`	*n = 4 > 1*: önce çarpma *4 · 1 = 4*, sonra `faktKuyruk(3, 4)`	
2	`faktKuyruk(3, 4)`	*3 · 4 = 12*, sonra `faktKuyruk(2, 12)`	
3	`faktKuyruk(2, 12)`	*2 · 12 = 24*, sonra `faktKuyruk(1, 24)`	
4	`faktKuyruk(1, 24)`	taban durum: `acc` döner; sonuç burada hazırdır	24
5	`faktKuyruk(2, 12)`	24'ü değiştirmeden iletir	24
6	`faktKuyruk(3, 4)`	24'ü değiştirmeden iletir	24
7	`faktKuyruk(4, 1)`	24'ü değiştirmeden iletir	24

Tablo 4.16: `faktKuyruk(4, 1)` izlemesi.

Tablo 4.2'deki `faktoriyel(4)` izlemesiyle karşılaştırınca fark açıkça görülür. Orada çarpmalar dönüş yolunda yapılıyordu: *2 · 1*, *3 · 2*, *4 · 6*. Burada çarpmalar iniş yolunda yapılır: *4 · 1*, *3 · 4*, *2 · 12*. Çarpma sırası farklıdır ama çarpma değişmeli olduğu için sonuç aynıdır. Asıl fark dönüş yolundadır. 5–7. adımlarda hiçbir çerçeve yeni bir hesap yapmaz; hepsi aldığı 24'ü olduğu gibi iletir.



Şekil 4.27: `faktKuyruk(4, 1)` çağrı zinciri. Üstte taban durum (Tablo 4.16, 4. adım): her kutuda o çağrının yaptığı çağrının parametreleri yazılıdır; `faktKuyruk(1, 24)` sonucu, 24'ü, hazır döndürüyor. Altta bütün çerçe-

veler aynı 24'ü iletmiştir.

Dönüşte yapılacak iş yoksa bekleyen çerçevelere de gerek yoktur. Bir derleyici bunu fark edip kuyruk çağrısında yeni çerçeve açmak yerine mevcut çerçeveyi yeniden kullanabilir. Bu eniyilemeye **kuyruk çağrısı eniyilemesi** denir. Ama Java bunu yapmaz: `faktKuyruk` için de her çağrıda yeni bir çerçeve açılır. Yukarıdaki izlemede 4 çerçeve açılmıştır; dönüşte bu çerçevelere ihtiyaç yoktu, ama yine de yığında yer tuttular.

4.7.3 Döngüye çevirme

Kuyruk özyinelemeli bir metot her zaman bir döngüye çevrilebilir ve bu çevirme tamamen mekaniktir. Bir kuyruk çağrısı, parametrelere yeni değerler verip metodun başına dönmek demektir. Döngü de tam olarak bunu yapar.

```
int faktDongu(int n) {
  int acc = 1;
  while (n > 1) {   // taban durum değilken
    acc = n * acc;  // çağrının yeni
    n = n - 1;      //   parametreleri
  }
  return acc;       // taban: return acc
}
```

Çevirmenin kuralları şunlardır:

- **Parametreler değişken olur.** `n` zaten bir parametredir; `acc`, ilk çağrıdaki değeri olan 1 ile başlatılan yerel bir değişkene dönüşür.
- **Taban durumun tersi döngü koşulu olur.** Özyineleme `n <= 1` iken durur; döngü `n > 1` iken sürer.
- **Kuyruk çağrısı, parametrelerin güncellenmesi olur.** `faktKuyruk(n - 1, n * acc)` çağrısı, `acc = n * acc;` ve `n = n - 1;` atamalarına dönüşür. Atamaların sırası önemlidir: `acc` eski `n` ile hesaplanmalıdır. Önce `n` azaltılsaydı yanlış sayıyla çarpılırdı.
- **Taban durumdaki dönüş, döngüden sonraki dönüş olur:** `return acc;`.

Tur	`n`	`acc`	Karşılığı
başlangıç	4	1	`faktKuyruk(4, 1)`
1	3	4	`faktKuyruk(3, 4)`
2	2	12	`faktKuyruk(2, 12)`
3	1	24	`faktKuyruk(1, 24)` : koşul yanlış, 24 döner

Tablo 4.17: `faktDongu(4)` izlemesi. Her satır, özyinelemeli sürümdeki bir çağrının parametrelerine karşılık gelir.

Döngü sabit bellekle çalışır: yalnız iki değişken vardır, çerçeve açılmaz. Bellek *O(1)*, zaman *O(n)*'dir. Aynı çevirme `ikiliAra`'ya da uygulanabilir: `lo` ve `hi` değişken olur, döngü `lo <= hi` iken sürer, her turda `lo` ya da `hi` güncellenir. Bilinen döngülü ikili arama, özyinelemeli sürümün tam olarak bu biçimde çevrilmiş hâlidir.

Dil desteği konusunda bir not: Java derleyicisi ve JVM kuyruk çağrısını eniyilemez; çok derin bir kuyruk özyinelemesi de `StackOverflowError` verir. Scheme gibi bazı diller bu eniyilemeyi dilin kuralı olarak garanti eder. Scala ve Kotlin derleyicileri, bir metodun kendini çağıran kuyruk özyinelemesini döngüye çevirebilir. Java'da derinliği büyük olabilecek kuyruk özyinelemesi, bu yüzden elle döngüye çevrilir.

Örnek 4.8

Aşağıdaki kuyruk özyinelemeli metot bir sayının basamak sayısını bulur: `int bas(int n, int acc) { if (n < 10) return acc + 1; return bas(n / 10, acc + 1); }`. İlk çağrı `bas(n, 0)`'dır. Döngüye çevirelim ve `bas(4096, 0)`'ı izleyelim.

1. Parametreler değişken olur: `acc = 0` ile başlanır.
2. Taban durum `n < 10`'dur; döngü `n >= 10` iken sürer.
3. Kuyruk çağrısı `bas(n / 10, acc + 1)`, `acc = acc + 1; n = n / 10;` olur (burada sıra fark etmez, çünkü iki ifade birbirinin eski değerini kullanmaz).
4. Döngüden sonra taban durumdaki ifade döndürülür: `return acc + 1;`.
5. İzleme: *(n, acc)* = (4096, 0) → (409, 1) → (40, 2) → (4, 3). *4 < 10*, döngü biter, *3 + 1 = 4* döner.

4.7.4 Özyineleme mi, döngü mü?

Her döngü özyinelemeyle, her özyineleme de döngüyle yazılabilir. Seçim, problemin yapısına ve girdinin büyüklüğüne bağlıdır. Tablo 4.18 iki yaklaşımı dört ölçüte göre karşılaştırır.

Ölçüt	Özyineleme	Döngü
Okunabilirlik	tanıma yakın, kısa	durum değişkenleri elle yönetilir
Bellek	*O*(derinlik) çağrı yığını	çoğunlukla *O(1)*
Hız	her çağrının ek maliyeti var	genellikle daha hızlı
Derinlik sınırı	`StackOverflowError` riski	yok

Tablo 4.18: Özyineleme ile döngünün karşılaştırılması.

Şu durumlarda **özyineleme doğal seçimdir**: veri iç içe bir yapıdaysa (ağaçlar, iç içe klasörler, iç içe ifadeler); problem geri izleme gerektiriyorsa (alt kümeler, permütasyonlar); problem her adımda yarıya bölünüyorsa; Hanoi gibi tanımı özyinelemeli olan problemlerde. Bu durumların ortak noktası, derinliğin küçük olmasıdır: *log n* kadar ya da *n*'nin kendisi küçük.

Şu durumlarda **döngü daha iyidir**: basit sayma ve toplama işlerinde; derinliği *n*'ye ulaşan doğrusal özyinelemede, özellikle *n* büyükse (uzun bir bağlı liste, büyük bir sayı); tekrarlanan alt problemli ikili özyinelemede (`fib`).

Daha genel bir sonuç da vardır: kuyruk özyinelemeli olmayanlar dahil, her özyinelemeli metot döngüye çevrilebilir. Bunun yolu, çağrı yığınının yaptığı işi açık bir yığın veri yapısıyla yapmaktır. Bekleyen işler (yarım kalmış çarpmalar, henüz çözülmemiş alt problemler) bir yığına konur ve döngü yığın boşalana kadar sürer. Hanoi kuleleri böyle bir "görev yığınıyla" özyinelemesiz çözülebilir. Bu çevirme kuyruk özyinelemesindeki kadar kısa değildir, ama yığın taşması riskini ortadan kaldırır: açık yığın, çağrı yığınının aksine, programın genel belleğinde büyür.

> **Sık yapılan hata**
>
> Özyinelemenin her zaman yavaş ve kötü olduğunu düşünmek. `fib` 'in yavaşlığı özyinelemeden değil, aynı alt problemin tekrar tekrar çözülmesinden gelir. `ikiliAra` ya da `hizliUs` gibi *O(log n)* derinlikli özyinelemeler hem hızlıdır hem de yığını zorlamaz. Sorun, derinliğin ve tekrarın büyüklüğündedir; özyinelemenin kendisinde değil.

4.8 Bağlı listede özyineleme

Özyineleme yalnızca sayılar ve diziler üzerinde değil, veri yapılarının kendisi üzerinde de doğal bir araçtır. En basit bağlantılı veri yapısı tek yönlü bağlı listedir ve özyinelemeli bir gözle bakılınca işlemleri çok kısa yazılır. Aynı bakış ağaçlar gibi daha karmaşık yapılarda da işe yarar.

4.8.1 Liste özyinelemeli bir yapıdır

Bağlı liste (*linked list*), öğeleri **düğümlerde** (*node*) tutan bir veri yapısıdır. Her düğüm bir veri ve bir sonraki düğümün referansını taşır. Son düğümün "sonraki" referansı `null` 'dır. Listenin ilk düğümünü gösteren referansa genellikle `bas` denir. Liste boşsa `bas` da `null` 'dır.

Bu yapıya özyinelemeli bir tanım verilebilir: bağlı liste ya boştur (`null`) ya da bir düğüm ile onu izleyen *daha kısa bir listedir*. `bas → 7 → 3 → 9 → null` listesi, 7'yi taşıyan bir düğüm ile `3 → 9 → null` listesinden oluşur. O liste de 3'ü taşıyan düğüm ile `9 → null` listesinden oluşur. En sonda boş liste vardır.



Şekil 4.28: `bas → 7 → 3 → 9 → null` listesi. İlk düğümün `sonraki` referansından başlayan kısım (`d.sonraki`), kendisi de bir bağlı listedir: "kalan liste".

Tanım özyinelemeli olduğu için liste üzerindeki işlemler de doğal olarak özyinelemeli yazılır. Kalıp hep aynıdır: düğüm işlenir, kalan liste (`d.sonraki`) aynı metoda verilir. Taban durum boş listedir: `d == null`. Listenin uzunluğu bu kalıbın en sade örneğidir:

```
class Dugum { int veri; Dugum sonraki; }

int uzunluk(Dugum d) {
  if (d == null) return 0;        // boş liste
  return 1 + uzunluk(d.sonraki);  // 1 + kalan
}
```

Boş listenin uzunluğu 0'dır. Boş olmayan listenin uzunluğu, ilk düğüm için 1 artı kalan listenin uzunluğudur. Özyinelemeli düşünmenin üç sorusu burada şöyle cevaplanır: taban durum `d == null`, küçültme `d.sonraki`'ye geçmek, birleştirme 1 eklemek. Listeyi baştan sona yazdıran metot da aynı kalıba uyar:

```
void yazdir(Dugum d) {
  if (d == null) return;
  System.out.print(d.veri + " ");
  yazdir(d.sonraki);
}
```

Sık yapılan hata

Taban durumu `d.sonraki == null` diye yazmak. Bu koşul tek düğümlü listeyi taban durum sayar, ama boş listeyi (`d == null`) yakalamaz: `uzunluk(null)` çağrılınca `d.sonraki`'ye erişilir ve `NullPointerException` fırlatılır. Taban durum, yapının en küçük hâli olan boş listedir.

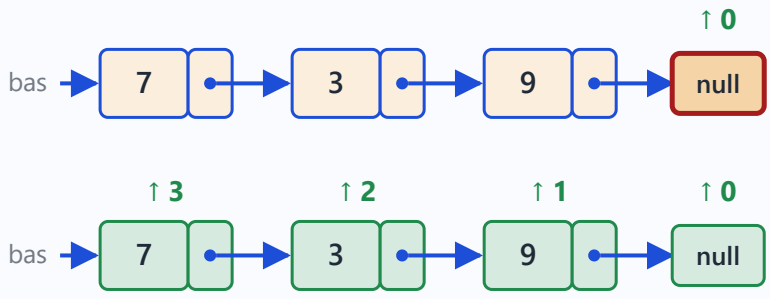
4.8.2 Kod izleme: uzunluk(bas)

```
int uzunluk(Dugum d) {
  if (d == null) return 0;        // boş liste
  return 1 + uzunluk(d.sonraki);  // 1 + kalan
}
// main içinde (bas → 7 → 3 → 9 → null):
System.out.println(uzunluk(bas));
```

Adım	`d`	Olay	Dönen
1	7 düğümü	$d \neq$ `null` : `1 + uzunluk(d.sonraki)` ; kalan liste 3 ile başlar	
2	3 düğümü	$d \neq$ `null` : kalan liste 9 ile başlar	
3	9 düğümü	$d \neq$ `null` : `d.sonraki` `null` , boş liste için çağrı	
4	`null`	taban durum: boş liste	0
5	9 düğümü	*1 + 0*	1
6	3 düğümü	*1 + 1*	2
7	7 düğümü	*1 + 2*	3

Tablo 4.19: `uzunluk(bas)` izlemesi, liste `7 → 3 → 9 → null` .

Sonuç 3'tür. 4 çağrı yapılmıştır: 3 düğüm ve sondaki `null` . *n* düğümlü listede *n + 1* çağrı yapılır ve yığın en derin anda *n + 1* çerçeve içerir. Zaman *O(n)*, yığın belleği *O(n)*'dir.



Şekil 4.29: `uzunluk(bas)` izlemesinden iki an. Üstte taban durum (Tablo 4.19, 4. adım): `d` , `null` 'ı gösteriyor ve 0 dönüyor. Altta son durum: düğümlerin üstündeki sayılar, o düğümden başlayan listenin uzunluğudur (3, 2, 1, 0).

Bu bellek maliyeti, listeler için ciddi bir sorundur. Bağlı listeler milyonlarca düğüm içerebilir ve `uzunluk` , bir milyon düğümlü listede bir milyon çerçeve açmaya çalışır. 4.2.3'te görüldüğü gibi bu kadar çerçeve varsayılan yığına sığmaz. Döngülü sürüm aynı işi sabit bellekle yapar: `int s = 0; for (Dugum d = bas; d != null; d = d.sonraki) s++;` .

4.8.3 Listeyi tersten yazdırma

Tek yönlü bağlı listede her düğüm yalnız bir sonrakini bilir; bir öncekine giden referans yoktur. Listeyi sondan başa yazdırmak bu yüzden döngüyle kolay değildir. Önce son düğüme gidilir, ama ondan sonra bir öncekine dönülemez. Özyineleme bu sorunu kendiliğinden çözer.

```
void tersYazdir(Dugum d) {
  if (d == null) return;
  tersYazdir(d.sonraki);  // önce kalan
  System.out.print(d.veri + " ");
}
```

4.8.1'deki `yazdir` ile bu metot arasındaki tek fark, yazdırma satırının özyinelemeli çağrıdan önce mi sonra mı geldiğidir. Bu, 4.1.5'teki `geriSay2` ile `ileriSay` farkının aynısıdır. `yazdir(bas)` çıktısı `7 3 9` 'dur: yazdırma çağrıdan önce yapılır ve düğümler gidiş sırasıyla yazılır. `tersYazdir(bas)` çıktısı ise `9 3 7` 'dir.



Şekil 4.30: `bas → 7 → 3 → 9 → null` listesi. `yazdir` düğümleri gidiş yönünde (7 3 9), `tersYazdir` çağrı yığını çözülürken dönüş yönünde (9 3 7) yazar.

Örnek 4.9

`tersYazdir(bas)` izlemesi:

1. `tersYazdir(7)` : $d \neq$ `null` ; önce `tersYazdir(3)` çağrılır, 7'nin yazdırılması bekler.
2. `tersYazdir(3)` : önce `tersYazdir(9)` ; 3 bekler.
3. `tersYazdir(9)` : önce `tersYazdir(null)` ; 9 bekler.
4. `tersYazdir(null)` : taban durum, hemen döner.
5. Yığın çözülür: `tersYazdir(9)` `9` yazar, sonra `tersYazdir(3)` `3` yazar, en son `tersYazdir(7)` `7` yazar.

Çıktı `9 3 7` 'dir. Düğümler, yığın çözülürken sondan başa yazdırılmıştır.

Tek yönlü listede geriye giden bağlantı yoktur; geri dönüşü çağrı yığını sağlar. Her çerçeve kendi düğümünü "aklında tutar" ve sırası gelince yazar. Bunun bedeli $O(n)$ yığın belleğidir. Çok uzun listelerde aynı iş bir döngü ve açık bir yığın veri yapısıyla daha güvenli yapılır. Döngü düğümleri baştan sona gezer ve her birini yığına ekler. Sonra yığın boşaltılır; düğümler ters sırayla çıkar ve yazdırılır.

Sık yapılan hata

Listeyi tersten yazdırmak için listeyi tersine çevirmek. Liste yerinde tersine çevrilirse yapı değişir; başka bir yerde aynı listeye bakan kod bozulur. `tersYazdir` listeye hiç dokunmaz, yalnız okur. Yapıyı değiştirmeden ters sırada gezmek gerekiyorsa özyineleme ya da açık bir yığın kullanılır.

4.9 Sık yapılan hatalar

Özyinelemedeki hataların çoğu birkaç tipe ayrılır. Not boyunca her konunun yanında verilen hatalar burada bir arada toplanmıştır; özyinelemeli bir metodu gözden geçirirken kontrol listesi olarak kullanılabilir.

Hata	Ne olur?	Doğrusu
Taban durum eksik ya da ulaşılamaz	0 ya da negatif girdi taban durumu atlar; çağrılar bitmez, `StackOverflowError`.	`n == 1` yerine `n <= 1` gibi her girdiyi yakalayan bir koşul.
Problem küçülmüyor	`f(n)` içinden yine `f(n)` ya da `f(n + 1)` çağrılır; taban duruma yaklaşılmaz.	Her çağrıda girdi taban duruma doğru değişmeli.
Dönüş değerini kaybetmek	`ikiliAra`'da sol yarı çağrısının önündeki `return` unutulursa sonuç atılır, akış sağ yarıya düşer; derleyici uyarmaz.	Özyinelemeli çağrının sonucu kullanılmalı ya da döndürülmeli.
Geri almayı unutmak	Geri izlemede bir dalın seçimleri sonraki dallara taşar; alt kümeler ya da permütasyonlar yanlış çıkar.	Paylaşılan nesneye yapılan her değişiklik, çağrı dönünce geri alınmalı.
Tekrarlanan hesap	`fib` gibi ikili özyinelemede aynı alt problem üstel sayıda çözülür.	Bellekleme ya da döngü: *O(n)*.
Derin doğrusal özyineleme	Milyon düğümlü listede `uzunluk` yığını taşırır; Java kuyruk çağrısını eniyilemez.	Döngü.

Tablo 4.20: Özyinelemeli kodda sık görülen hatalar.

Bu hataların hepsi aynı iki soruya bağlanır. Birincisi doğrulukla ilgilidir: *taban durum her girdiyi yakalıyor mu ve her adım ona yaklaşıyor mu?* İlk iki hata bu sorunun cevabının "hayır" olduğu durumlardır. Üçüncü ve dördüncü hata da doğrulukla ilgilidir, ama tümevarım varsayımının bozulmasıyla: küçük problemin cevabı ya kaybolur ya da küçük problem yanlış bir durumdan başlar. İkinci soru maliyetle ilgilidir: *çağrı ağacı kaç düğümlü ve ne kadar derin?* Son iki hata bu sorunun cevabı göz ardı edildiğinde ortaya çıkar. Düğüm sayısı üstelse zaman, derinlik çok büyükse bellek sorun olur.

4.10 Özet

- **Özyineleme**, bir metodun kendini daha küçük bir girdiyle çağırmasıdır. Her özyinelemeli metot bir **taban durum** (doğrudan çözülen girdiler) ve taban duruma yaklaşan bir **özyinelemeli adım** içerir. Özyinelemeli düşünmede küçük problemin doğru çözüldüğü kabul edilir ve yalnız son adım düşünülür; bu, tümevarımın mantığıdır.
- Bir iş özyinelemeli çağrıdan **önce** yapılırsa çağrılar inerken, **sonra** yapılırsa çağrılar dönerken ve ters sırayla yapılır (`geriSay2` → `3 2 1` , `ileriSay` → `1 2 3`).
- Her çağrı **çağrı yığınında** bir **çerçeve** açar: parametreler, yerel değişkenler, dönüş adresi. Çerçeveler LIFO düzeniyle eklenip çıkarılır; yalnız en üstteki çalışır. Yığın sınırlıdır: çok derin ya da sonsuz özyineleme `StackOverflowError` verir. Her girdi, 0 ve negatifler dahil, sonlu adımda bir taban duruma ulaşmalıdır.
- **Doğrusal özyinelemede** her çağrı en çok bir çağrı yapar ve çağrılar bir zincir oluşturur: `faktoriyel` , `toplam` , `ters` , `uzunluk` . Girdi her adımda yarıya inerse zincir $O(\log n)$ uzunluğundadır: `hizliUs` $\lfloor \log_2 n \rfloor + 2$ çağrı yapar, `ikiliAra` $O(\log n)$'dir.
- **İkili özyinelemede** bir çağrı iki çağrı yapar ve bir **çağrı ağacı** oluşur. `fib(n)` $2F(n+1) - 1$ çağrı yapar ve yaklaşık $1{,}6^n$ hızında büyür, çünkü aynı alt problemler tekrar tekrar çözülür. Bellekleme ya da döngü $O(n)$ verir; döngülü sürüm $O(1)$ bellek kullanır.
- **Hanoi kuleleri** üç adımla çözülür: $n - 1$ disk yardımcıya, en büyük disk hedefe, $n - 1$ disk hedefe. Hamle sayısı $2^n - 1$'dir ve bu en azdır.
- **Geri izleme**, çözümü adım adım kurar: bir seçenek seçilir, özyinelemeli çağrıyla ilerlenir, dönüşte seçim **geri alınır**. `altKumeler` 2^n alt küme ve $2^{n+1} - 1$ çağrı, `perm` $n!$ permütasyon üretir.
- **Yineleme bağıntısı**, maliyeti küçük girdilerin maliyetiyle yazar ve çağrı ağacıyla çözülür: $T(n-1) + c \to O(n)$, $T(n/2) + c \to O(\log n)$, $2T(n-1) + c \to O(2^n)$. Zaman çağrı sayısından, alan **en derin çağrı zincirinden** gelir.
- **Kuyruk özyinelemesinde** özyinelemeli çağrı metodun son işidir; ara sonuç bir **biriktiriciyle** taşınır. Böyle bir metot mekanik olarak döngüye çevrilir: parametreler değişken, taban durumun tersi döngü koşulu olur. Java kuyruk çağrısını eniyilemez; `faktKuyruk` da $O(n)$ yığın kullanır.
- **Bağlı liste** özyinelemeli bir yapıdır: ya boştur ya da bir düğüm ve daha kısa bir listedir. Taban durum `d == null` 'dır. `tersYazdir` , çağrı yığınını kullanarak tek yönlü listeyi sondan başa yazar; bedeli $O(n)$ yığın belleğidir.

Örnek	Özyineleme türü	Zaman	Alan (yığın)
`faktoriyel` , `toplam` , `ters`	doğrusal	$O(n)$	$O(n)$
`hizliUs` , `ikiliAra`	doğrusal, yarılayan	$O(\log n)$	$O(\log n)$
`fib`	ikili	üstel ($\approx 1{,}6^n$)	$O(n)$
`hanoi`	ikili	$O(2^n)$	$O(n)$
`altKumeler`	geri izleme	$O(n \cdot 2^n)$	$O(n)$
`perm`	çoklu, geri izleme	$O(n \cdot n!)$	$O(n)$

Örnek	Özyineleme türü	Zaman	Alan (yığın)
`faktKuyruk`	kuyruk özyinelemesi	$O(n)$	$O(n)$ (Java'da)
`uzunluk`, `tersYazdir`	doğrusal (bağlı liste)	$O(n)$	$O(n)$

Tablo 4.21: Bölümdeki özyinelemeli metotların özeti.

Bölümün tamamı tek bir cümleye sığar: özyineleme, taban durum ile taban duruma yaklaşan özyinelemeli adımdan oluşur; zaman maliyeti çağrı sayısından, alan maliyeti en derin çağrı zincirinden gelir.

4.11 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

özyineleme *recursion* · taban durum *base case* · özyinelemeli adım *recursive step* · yığın çerçevesi *stack frame* · çağrı yığını *call stack* · LIFO *last-in, first-out* · doğrusal özyineleme *linear recursion* · ikili özyineleme *binary recursion* · çoklu özyineleme *multiple recursion* · ikili arama *binary search* · Hanoi kuleleri *Towers of Hanoi* · geri izleme *backtracking* · yineleme bağıntısı *recurrence relation* · kuyruk özyinelemesi *tail recursion* · biriktirici *accumulator* · bağlı liste *linked list* · düğüm *node* · zaman karmaşıklığı *time complexity* · alan karmaşıklığı *space complexity*

4.12 Kendini Dene

1. `void yaz(int n) { if (n == 0) return; yaz(n / 10); System.out.print(n % 10 + " "); }` için `yaz(472)` ne yazar? Kaç çağrı yapılır ve yığında (`main` dahil) en çok kaç çerçeve bulunur?

   Cevap

   Çağrılar 472, 47, 4, 0 girdileriyle yapılır: 4 çağrı. `yaz(0)` taban durumdur ve hiçbir şey yazmaz. Yazdırma çağrıdan sonra olduğu için dönüşte, ters sırayla yapılır: önce `yaz(4)` 4'ü, sonra `yaz(47)` 7'yi, en son `yaz(472)` 2'yi yazar. Çıktı `4 7 2 `'dir; basamaklar soldan sağa yazılmıştır. En derin anda 4 `yaz` çerçevesi ve `main` vardır: 5 çerçeve.

2. `hizliUs(3, 9)` kaç çağrı yapar? Her çağrı ne döndürür?

   Cevap

   n değerleri 9, 4, 2, 1, 0'dır: 5 çağrı ($\lfloor \log_2 9 \rfloor + 2 = 3 + 2$). Dönüşte: `hizliUs(3, 0) = 1`; $n = 1$ tek, $3 \cdot 1 \cdot 1 = 3$; $n = 2$ çift, $3 \cdot 3 = 9$; $n = 4$ çift, $9 \cdot 9 = 81$; $n = 9$ tek, $3 \cdot 81 \cdot 81 = 19.683$. Sonuç $3^9 = 19.683$.

3. Özyinelemeli `fib(7)` kaç çağrı yapar? Yığında aynı anda en çok kaç `fib` çerçevesi bulunur?

> **Cevap**
>
> $F(8) = 21$ olduğundan çağrı sayısı $2 \cdot 21 - 1 = 41$'dir; `fib(7) = 13`. En derin zincir `fib(7)` → `fib(6)` → … → `fib(1)` 'dir: 7 çerçeve (`main` ile 8). Zaman üstel, bellek $O(n)$.

4. `a = {1, 4, 7, 10, 13, 16, 19}` dizisinde `ikiliAra(a, 19, 0, 6)` ve `ikiliAra(a, 5, 0, 6)` kaçar çağrı yapar, ne döndürür?

> **Cevap**
>
> 19 için: `(0, 6)` `mid = 3` , $19 > 10$; `(4, 6)` `mid = 5` , $19 > 16$; `(6, 6)` `mid = 6` , bulundu. 3 çağrı, 6 döner. 5 için: `(0, 6)` $5 < 10$; `(0, 2)` `mid = 1` , $5 > 4$; `(2, 2)` `mid = 2` , $5 < 7$; `(2, 1)` aralık boş. 4 çağrı, −1 döner.

5. 5 diskli Hanoi kulelerinde toplam kaç hamle yapılır? 2 numaralı disk kaç kez taşınır?

> **Cevap**
>
> $2^5 - 1 = 31$ hamle. Disk k, 2^{n-k} kez taşınır: disk 2 için $2^3 = 8$. Denetim: disk 1'den 5'e taşınma sayıları 16, 8, 4, 2, 1'dir ve toplamı 31'dir.

6. `altKumeler(new int[]{5, 6}, 0, new ArrayList<>())` ne yazar ve kaç çağrı yapar?

> **Cevap**
>
> Çıktı `[5, 6] [5] [6] []` 'dir. Karar ağacında 1 + 2 + 4 = 7 düğüm vardır: $2^{2+1} - 1 = 7$ çağrı. Yığın en çok 3 `altKumeler` çerçevesi içerir.

7. Bir metot `n <= 0` iken sabit iş yapıp duruyor, öteki durumda sabit iş yapıp kendini $n - 3$ ile bir kez çağırıyor. Yineleme bağıntısı nedir? $n = 12$ için kaç çağrı yapılır?

> **Cevap**
>
> $T(n) = T(n - 3) + c$. Zincir 12, 9, 6, 3, 0'dır: 5 çağrı. Genel olarak yaklaşık $n/3 + 1$ çağrı yapılır; bu $O(n)$'dir, çünkü girdi her adımda sabit bir miktar azalır.

8. `int ik(int n, int acc) { if (n == 0) return acc; return ik(n - 1, acc * 2); }` kuyruk

özyinelemeli midir? `ik(5, 1)` hangi çağrı zinciriyle ne döndürür? Döngülü karşılığı nedir?

Cevap

Evet: çarpma parametre hesaplanırken, çağrıdan önce yapılır; dönen değer değiştirilmeden iletilir. Zincir *(5, 1) → (4, 2) → (3, 4) → (2, 8) → (1, 16) → (0, 32)*; 6 çağrı, 32 döner (2^5). Döngülü karşılığı: `int acc = 1; while (n != 0) { acc = acc * 2; n = n - 1; } return acc;` . Java'da özyinelemeli sürüm *O(n)*, döngü *O(1)* bellek kullanır.

9. `int say(Dugum d, int x) { if (d == null) return 0; return (d.veri == x ? 1 : 0) + say(d.sonraki, x); }` . Liste `2 → 7 → 2 → 2 → null` ise `say(bas, 2)` ne döndürür, kaç çağrı yapar? Metot kuyruk özyinelemeli midir?

Cevap

3 döndürür (2 değerli üç düğüm). Çağrı sayısı düğüm sayısı artı 1'dir: 5 (dört düğüm ve `null`). Kuyruk özyinelemeli değildir: çağrı döndükten sonra sonuca 0 ya da 1 eklenir. Biriktiricili bir sürüm (`say(d.sonraki, x, acc + …)`) kuyruk özyinelemeli olur.

10. `int t(int n) { if (n <= 1) return 1; return t(n - 1) + t(n - 1) + t(n - 1); }` için `t(3)` ne döndürür, kaç çağrı yapar? Metot nasıl doğrusal yapılır?

Cevap

Çağrı ağacında düzeyler 1, 3, 9 düğümlüdür: *1 + 3 + 9 = 13* çağrı. 9 yaprağın her biri 1 döndürür: `t(3) = 9 = 3²` . Genel olarak $t(n) = 3^{n-1}$ ve çağrı sayısı $(3^n - 1)/2$'dir: $O(3^n)$. Üç çağrı aynı değeri hesaplar; `return 3 * t(n - 1);` yazılırsa tek çağrı kalır ve maliyet *O(n)* olur.

4.13 İleri okuma

- M. T. Goodrich, R. Tamassia ve M. H. Goldwasser, *Data Structures and Algorithms in Java* (6. baskı, 2014), Bölüm 5: özyinelemeyi doğrusal, ikili ve çoklu özyineleme diye ayırarak anlatır ve bu nottakine yakın Java örnekleri (ikili arama, diziyi tersine çevirme, üs alma) verir.
- R. Sedgewick ve K. Wayne, *Computer Science: An Interdisciplinary Approach* (2016), Kesim 2.3: Hanoi kuleleri, Fibonacci'deki tekrarlanan hesap ve bellekleme konularını kısa programlar ve deneylerle işler.
- E. Roberts, *Thinking Recursively with Java* (2006): kitabın tamamı özyinelemeli düşünmeye ayrılmıştır; tümevarımla bağlantıyı ve geri izleme örneklerini başlangıç düzeyindeki okura göre açıklar.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 4: yineleme bağıntılarını özyineleme ağacı ve yerine koyma yöntemleriyle daha biçimsel bir dille çözer.

4.14 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022.
2. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6. baskı, Wiley, 2014.
3. E. Roberts, *Thinking Recursively with Java*, Wiley, 2006.
4. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016.

KISIM II

Doğrusal Yapılar

Bölüm 5–8

KISIM II · BÖLÜM 5

Bağlı Listeler

Dizi, öğeleri bellekte yan yana tutar. Bu düzen indisle erişimi çok hızlı yapar, ama başa ya da araya ekleme gerektiğinde öğelerin kaydırılmasını zorunlu kılar. Bağlı liste aynı öğeleri başka bir yolla saklar: her öğe kendi küçük kutusunda, yani bir düğümde durur ve kendisinden sonra gelen düğümün yerini bilir. Sırayı bellekteki konum değil, bu bağlar belirler. Not, düğüm kavramıyla başlar; tek yönlü bağlı listede gezinme, arama, ekleme ve silme işlemlerini kodlarıyla ve adım adım izlemeleriyle anlatır. Ardından ters çevirme, orta öğeyi bulma, kopyalama, tekrarları kaldırma, sıralı listeleri birleştirme ve sıralama gibi klasik problemleri işler. Sonraki kesimler çift yönlü listeyi, bekçi düğümleri ve dairesel listeyi ele alır. Son kesim, dizi ile bağlı liste arasındaki seçimi ve işlem maliyetlerini bir arada verir.

Bu bölümün kodu: `github.com/sercankulcu/data-structures-java/Bolum05/src`

ÖĞRENME ÇIKTILARI

Düğüm yapısı
Düğüm, bağlantı ve referans kavramlarını açıklamak; bir dizi atamadan sonra bağlı yapının nasıl göründüğünü çizmek.

Temel işlemler
Tek yönlü, çift yönlü ve dairesel listelerde ekleme, silme, arama ve gezinme işlemlerini doğru sırada bağ kurarak uygulamak.

Liste türleri
Tek yönlü, çift yönlü ve dairesel listeleri karşılaştırmak; bekçi düğümlerin ve son işaretçisinin neyi kolaylaştırdığını açıklamak.

Maliyet analizi
Bağlı liste işlemlerinin zaman ve alan maliyetini gerekçesiyle vermek; aynı işlemin dizideki maliyetiyle karşılaştırmak.

5.1 Temeller

Bir programın sakladığı öğelerin sayısı çoğu zaman önceden bilinmez. Bir kuyruğa kaç müşteri gireceği, bir metin düzenleyicide kaç satır yazılacağı ya da bir oyunda kaç nesnenin ekranda olacağı çalışma sırasında belli olur. Dizi bu durumda zorlanır: kapasitesi baştan belirlenir, dolunca daha büyük bir diziye kopyalanması gerekir. Başa ya da araya eklenen her öğe için de arkadaki öğelerin hepsi bir hücre kaydırılır. Bağlı liste bu iki sorunu ortadan kaldırmak için tasarlanmıştır. Bu kolaylığın da bir bedeli vardır; kesimin sonundaki üstünlükler ve zayıflıklar listesi bu bedeli ayrıntısıyla verir.

5.1.1 Bağlı liste ve türleri

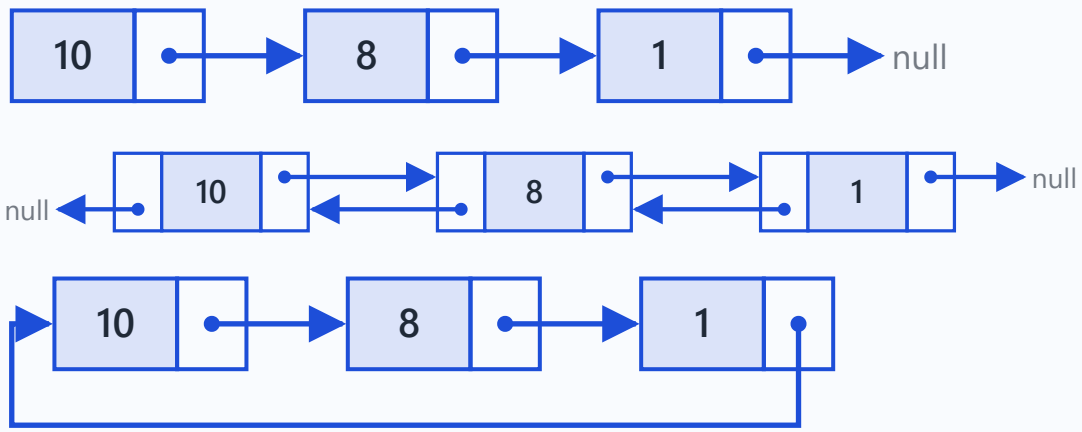
Bir hazine avı oyununu düşünelim. Oyuncu ilk ipucunu alır. İpucunun yazılı olduğu kâğıtta ikinci ipucunun nerede saklandığı yazar. İkinci kâğıt üçüncünün yerini söyler, böyle sürüp gider. Kâğıtlar evin dört bir yanına dağılmıştır; çekmecede, kitabın arasında, kapının arkasında durabilirler. Yine de sıra hiç karışmaz, çünkü her kâğıt bir sonrakini gösterir. Oyunu başlatmak için yalnız ilk kâğıdın yerini bilmek yeter.

Bağlı liste (*linked list*) aynı fikre dayanır. Öğeler, dizideki gibi bellekte yan yana hücrelerde durmaz. Her öğe, **düğüm** (*node*) denen ayrı bir nesnede saklanır. Düğüm, öğenin kendisinin yanında bir sonraki düğümün nerede olduğunu da tutar. Hazine avındaki kâğıt düğüme, kâğıttaki adres de bağa karşılık gelir.

> **Tanım 5.1**
>
> **Bağlı liste**, öğelerini düğümlerde saklayan ve düğümleri birbirine referanslarla bağlayan doğrusal veri yapısıdır. Öğelerin sırasını bellekteki konumları değil, düğümlerdeki bağlar belirler. Listenin ilk düğümüne **baş** (*head*), son düğümüne **son** (*tail*) denir.

Düğümlerin birbirine nasıl bağlandığına göre üç temel bağlı liste türü vardır. Şekil 5.1 üçünü yan yana gösterir.

- **Tek yönlü bağlı liste** (*singly linked list*): Her düğüm yalnız kendinden sonrakini gösterir. Son düğümün bağı boştur; Java'da bu boş bağ `null` değeriyle gösterilir. Liste yalnız baştan sona doğru gezilebilir.
- **Çift yönlü bağlı liste** (*doubly linked list*): Her düğüm hem öncekini hem sonrakini gösterir. Liste iki yönde de gezilebilir. İlk düğümün önceki bağı ve son düğümün sonraki bağı `null`'dır.
- **Dairesel bağlı liste** (*circularly linked list*): Son düğüm `null` yerine ilk düğümü gösterir. Liste bir halka olur; hiçbir bağ boş değildir.



Şekil 5.1: Üç bağlı liste türü, aynı 10, 8, 1 öğeleriyle. Üstte tek yönlü liste: oklar yalnız ileri gider, son düğüm `null` 'ı gösterir. Ortada çift yönlü liste: her komşu çifti arasında biri ileri, biri geri iki ok vardır; iki uçta `null` bulunur. Altta dairesel liste: son düğümden çıkan bağ ilk düğüme döner.

Üç türün ortak noktası şudur: öğeler birbirini gösteren düğümlerde saklanır. Farklar yalnız düğümün kaç bağ tuttuğunda ve son bağın nereyi gösterdiğindedir. Bu küçük farklar, bazı işlemlerin maliyetini büyük ölçüde değiştirir. Örneğin son düğümü silmek tek yönlü listede bütün listeyi gezmeyi gerektirir, çift yönlü listede ise sabit sürede yapılır.

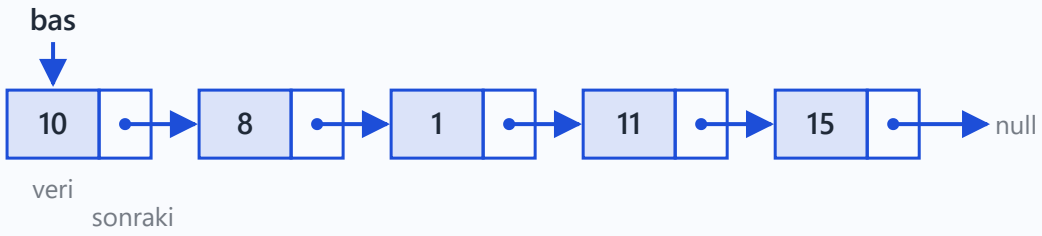
5.1.2 Tek yönlü bağlı liste

En yalın tür tek yönlü listedir. Öğeler tek yönlü bağlarla birbirine bağlıdır: her öğe kendinden sonrakine bir referans tutar. Burada **referans** (*reference*), bir nesnenin bellekteki yerini gösteren değerdir. Java'da sınıf türünden her değişken aslında bir referanstır; nesnenin kendisini değil, nesneye giden yolu tutar.

Listenin tamamına tek bir değişkenle erişilir. Bu değişken ilk düğümü gösterir ve **baş** adını alır; kodda `bas` diye yazılır. Baştan başlayıp bağları izleyen bir program bütün düğümlere ulaşır. Liste boşsa `bas` `null` 'dır.

Her düğüm iki parçadan oluşur:

- **Veri**: Düğümün taşıdığı değerdir. Bir tam sayı gibi ilkel tipte bir değer olabileceği gibi bir öğrenci kaydı gibi bir nesne de olabilir.
- **İşaretçi** (*pointer*): Sonraki düğümün referansıdır. Son düğümde bu alan `null` 'dır. Java'da "işaretçi" ile "referans" aynı şeyi anlatır; C gibi dillerdeki işaretçi aritmetiği Java'da yoktur.



Şekil 5.2: Tek yönlü bağlı liste. Her kutunun sol bölmesi düğümün verisini (`veri`), sağ bölmesi sonraki düğüme giden bağı (`sonraki`) tutar. `bas` ilk düğümü (10) gösterir; zincir 10 → 8 → 1 → 11 → 15 diye ilerler ve son düğümün bağı `null` 'da biter.

Şekil 5.2'de dikkat edilecek iki nokta vardır. Birincisi, `bas` bir düğüm değildir; yalnız ilk düğümü gösteren bir değişkendir. İkincisi, listenin sonunu gösteren ayrı bir işaret yoktur. Liste, sonraki alanı `null` olan düğümde biter. Bir program "son düğüme geldim mi?" sorusunu bu alana bakarak cevaplar.

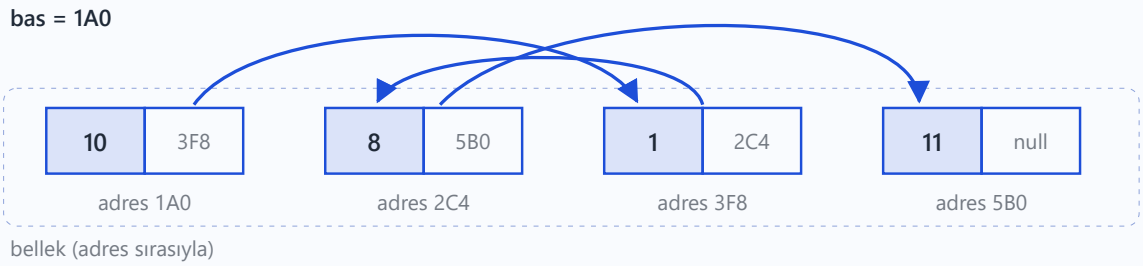
Sık yapılan hata

`bas` 'ı listenin ilk öğesiyle karıştırmak. `bas` düğümü tutmaz, düğüme giden yolu tutar. `bas` başka bir düğüme yöneltilirse eski ilk düğüme giden tek yol kaybolabilir. O düğümü ve ardından gelenleri gösteren başka bir referans yoksa liste parçası programdan tamamen kopar.

5.1.3 Düğümler bellekte nasıl durur?

Dizide öğeler ardışık bellek konumlarında tutulur. Bu yüzden *i*. öğenin adresi bir çarpma ve bir toplamayla hesaplanır. Bağlı listede böyle bir hesap yapılamaz. Her düğüm `new` ile ayrı ayrı oluşturulur ve Java'nın bellek yöneticisi onu o anda uygun bulduğu bir yere koyar. Ardışık oluşturulan iki düğüm bellekte yan yana düşebilir de, birbirinden çok uzakta da durabilir.

Buna rağmen bağlı liste **doğrusal** bir veri yapısıdır. Doğrusal olmak, her öğenin (sonuncusu hariç) tek bir ardılı olması demektir. Ağaçta bir düğümün birden çok çocuğu olabilir; bağlı listede ise her düğümün en çok bir sonrakisi vardır. Sırayı bellekteki konum değil, `sonraki` alanlarındaki referanslar belirler.



Şekil 5.3: Düğümlerin bellekteki yeri. Kutular adres sırasıyla dizilmiştir: 1A0'da 10, 2C4'te 8, 3F8'de 1, 5B0'da 11. Her düğümün sağ bölmesi sonraki düğümün adresini tutar. `bas = 1A0` 'dan başlayıp okları izleyen bir okuyucu 10 → 1 → 8 → 11 sırasını bulur; bu sıra adres sırası olan 10, 8, 1, 11'den farklıdır.

Şekil 5.3 bu farkı somut olarak gösterir. Adresler on altılık tabanda yazılmış uydurma değerlerdir; Java programları gerçek adresleri görmez. Asıl dikkat edilecek nokta, okların bellek sırasına uymamasıdır. 10'un düğümü 3F8'i, yani 1'in düğümünü gösterir. 1'in düğümü geri dönüp 2C4'teki 8'i gösterir. 8 de 5B0'daki 11'i gösterir. 11'in bağı `null` 'dır. Bellekte 8, 1'den önce gelir, ama listede 1'den sonra gelir.

Bu düzenin iki sonucu vardır. Birincisi iyidir: araya bir düğüm eklemek için hiçbir öğe kaydırılmaz, yalnız birkaç bağ değiştirilir. İkincisi kötüdür: *i*. öğeye doğrudan atlanamaz. Adres hesaplanamadığı için baştan başlanır ve bağlar tek tek izlenir. Bağlı listedeki işlemlerin maliyetlerini bu iki sonuç belirler.

5.1.4 Düğüm sınıfı

Java'da düğüm küçük bir sınıfla tanımlanır. Tam sayı tutan bir düğüm şöyledir:

```
// Düğüm sınıfı
class Dugum {
    int veri;
    Dugum sonraki;
    Dugum(int veri) {
        this.veri = veri;
        this.sonraki = null;
    }
}
```

Sınıfın iki alanı vardır. `veri`, düğümün taşıdığı tam sayıdır. `sonraki` ise türü yine `Dugum` olan bir alandır. Bir sınıfın kendi türünden bir alan içermesine **öz-referanslı yapı** (*self-referential structure*) denir. İlk bakışta bu tanım kendi içine dönen, sonsuz bir tanım gibi görünebilir: düğüm bir düğüm içeriyor, o düğüm de bir düğüm içeriyor ve böyle sürüyor. Ama `sonraki` alanı bir düğümün kendisini değil, bir düğüme giden referansı tutar. Referansın boyutu sabittir; bu yüzden her `Dugum` nesnesi bellekte, nesne başlığı dışında, yalnız bir tam sayı ve bir referans kadar yer kaplar.

Yapıcı, verilen değeri `veri` alanına yazar ve `sonraki`'yi `null` yapar. Java'da referans alanları zaten varsayılan olarak `null` başlar; satır yalnız niyeti açıkça göstermek için yazılmıştır. Yeni oluşturulan düğüm hiçbir yere bağlı değildir. Listeye ancak birisi onun referansını bir başka düğümün `sonraki` alanına ya da `bas`'a yazınca katılır.

Aynı sınıfın her tür veriyi saklayan **jenerik** (*generic*) sürümü de yazılabilir:

```
// Düğüm sınıfı
class Dugum<T> {
    T veri;
    Dugum<T> sonraki;
    Dugum(T veri) {
        this.veri = veri;
        this.sonraki = null;
    }
}
```

`T` bir tür parametresidir. `Dugum<String> d = new Dugum<>("Ali");` yazılınca `T` yerine `String` geçer ve düğüm bir metin saklar. Not boyunca örnekleri sade tutmak için tam sayı sürümü kullanılır, ama bütün algoritmalar jenerik sürümde de aynen çalışır. Değişen tek şey karşılaştırmalardır: tam sayılar `==` ve `<` ile karşılaştırılırken nesneler `equals` ve `compareTo` ile karşılaştırılır.

> **Sık yapılan hata**
>
> Jenerik düğümde değerleri `==` ile karşılaştırmak. `==` iki referansın aynı nesneyi gösterip göstermediğine bakar, değerlerin eşitliğine bakmaz. İki ayrı `String` nesnesi aynı metni taşısa bile `==` `false` verebilir. Jenerik listede aramada `gecici.veri.equals(aranan)` yazılır.

5.1.5 Elle liste oluşturma

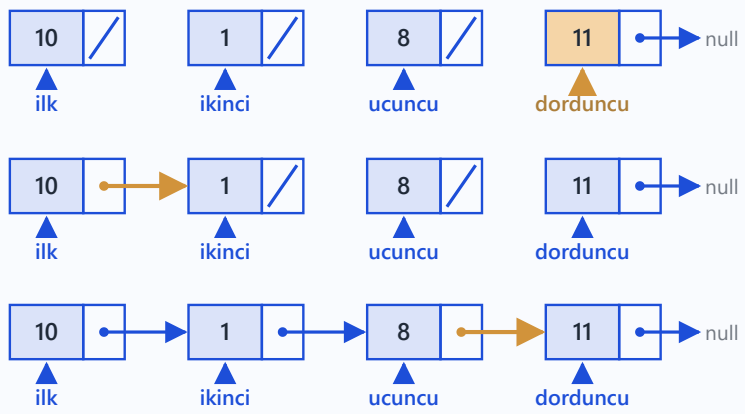
Bir listenin nasıl kurulduğunu görmenin en açık yolu, düğümleri tek tek oluşturup elle bağlamaktır. Aşağıdaki kod dört düğüm oluşturur ve onları 10 → 1 → 8 → 11 sırasıyla bağlar:

```
Dugum ilk = new Dugum(10);
Dugum ikinci = new Dugum(1);
Dugum ucuncu = new Dugum(8);
Dugum dorduncu = new Dugum(11);
ilk.sonraki = ikinci;
ikinci.sonraki = ucuncu;
ucuncu.sonraki = dorduncu;
```

İlk dört satır dört bağımsız düğüm oluşturur. Bu anda düğümlerin hepsinin `sonraki` alanı `null`'dır; aralarında hiçbir bağ yoktur. Her düğüm yalnız kendi değişkeni (`ilk`, `ikinci`, `ucuncu`, `dorduncu`) üzerinden erişilebilir. Son üç satır bağları kurar. `ilk.sonraki = ikinci` satırı, `ikinci` değişkenindeki referansı 10'un düğümünün `sonraki` alanına kopyalar. Artık 10'un düğümü 1'in düğümünü gösterir.

Satır	10.sonraki	1.sonraki	8.sonraki	11.sonraki	`ilk`'ten zincir
`ilk = new Dugum(10)`	null	—	—	—	10
`ikinci = new Dugum(1)`	null	null	—	—	10
`ucuncu = new Dugum(8)`	null	null	null	—	10
`dorduncu = new Dugum(11)`	null	null	null	null	10
`ilk.sonraki = ikinci`	düğüm(1)	null	null	null	10 → 1
`ikinci.sonraki = ucuncu`	düğüm(1)	düğüm(8)	null	null	10 → 1 → 8
`ucuncu.sonraki = dorduncu`	düğüm(1)	düğüm(8)	düğüm(11)	null	10 → 1 → 8 → 11

Tablo 5.1: Liste oluşturma kodunun izlemesi. Her satırdan sonra düğümlerin `sonraki` alanları ve `ilk`'ten başlayarak ulaşılabilen zincir.



Şekil 5.4: Liste oluşturmanın üç anı (yukarıdan aşağı). (a) Dört bağımsız düğüm: her birinin `sonraki` alanı boştur (çapraz çizgi `null` demektir). (b) `ilk.sonraki = ikinci`: 10'dan 1'e ilk bağ kuruldu (turuncu ok). (c) Üç bağ da kuruldu: 10 → 1 → 8 → 11 → null.

Bu küçük örnek, bağlı listelerle çalışırken akılda tutulacak bir gerçeği gösterir. Bir atama, nesneyi kopyalamaz; yalnız referansı kopyalar. `ilk.sonraki = ikinci` satırından sonra 1'in düğümüne iki yoldan ulaşılır: `ikinci` değişkeniyle ve `ilk.sonraki` ile. Düğüm tektir, ona giden yollar iki tanedir. Son satırdan sonra yardımcı değişkenlere de gerek kalmaz. Bütün listeye `ilk` üzerinden ulaşılır: `ilk.sonraki.sonraki.veri` ifadesi 8'dir.

Gerçek programlarda düğümler böyle elle bağlanmaz; listeye ekleme yapan metotlar kullanılır. Ama bu metotların her biri, içeride tam olarak buradaki işi yapar: yeni bir düğüm oluşturur ve birkaç referans alanını değiştirir.

5.1.6 Temel işlemler ve maliyetleri

Tek yönlü bağlı listede yapılan işlemleri ve maliyetlerini önce bir tablo hâlinde görelim. Gerekçelerin ayrıntısı, her işlem kendi kesiminde ele alınırken verilecek. Tablodaki *n*, listedeki düğüm sayısıdır.

İşlem	Nasıl yapılır?	Zaman
Başa ekleme	Yeni düğüm eski başı gösterir, baş yeni düğüm olur.	O(1)
Sona ekleme	Son düğüme kadar gezinilir; son işaretçisi tutuluyorsa gezinme gerekmez.	O(n) · O(1)
Baştan silme	Baş bir sonraki düğüme kaydırılır.	O(1)
Sondan silme	Sondan bir önceki düğüm bulunmalıdır.	O(n)
Arama, yazdırma, uzunluk	Baştan sona gezinilir.	O(n)
Ters çevirme, birleştirme	Listeler tek geçişte gezinilir (birleştirmede son işaretçisiyle).	O(n)

Tablo 5.2: Tek yönlü bağlı listede temel işlemler.

Tablo 5.2'deki maliyetlerin arkasında tek bir kural vardır: *bir işlem yalnız başa yakın düğümlere dokunuyorsa sabit sürelidir, listenin içine yürümesi gerekiyorsa doğrusal sürelidir.* Baş her zaman bilinir. Ona yapılan her şey, liste ne kadar uzun olursa olsun aynı sayıda adım sürer. Sondaki düğüm ise ancak bütün bağlar izlenerek bulunur. Bu yüzden son düğümle ilgili işlemler, son işaretçisi gibi ek bir yardım yoksa *O(n)*'dir.

i. öğeye erişim de *O(n)*'dir. Dizide `a[i]` adres hesabıyla tek adımda okunur. Bağlı listede ise *i*. düğüme ulaşmak için baştan başlanır ve *i* kez bir sonraki düğüme geçilir. En kötü durumda, yani son düğüm istendiğinde, $n - 1$ adım atılır.

> **Sık yapılan hata**
>
> Bağlı listede dizi gibi indisle gezinmek: `for (int i = 0; i < n; i++) islem(liste.get(i));` . Her `get(i)` baştan başlayıp *i* adım yürüdüğü için toplam adım sayısı $0 + 1 + \dots + (n-1) = n(n-1)/2$ olur, yani $O(n^2)$. Aynı iş, bir sonraki kesimdeki gezinme deseniyle tek geçişte, *O(n)*'de yapılır.

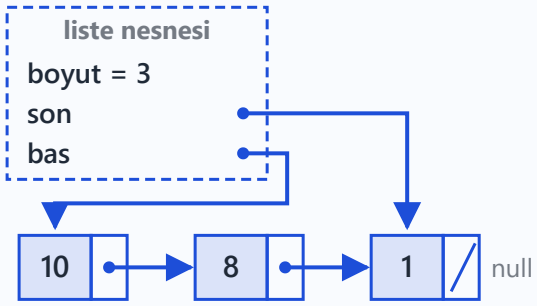
5.1.7 Liste ADT'si ve Java sınıfı

Bir veri yapısının hangi işlemleri sunduğunu, nasıl gerçeklendiğinden ayrı olarak tarif eden tanıma **soyut veri tipi** (*abstract data type*, kısaca ADT) denir. Goodrich, Tamassia ve Goldwasser'in ders kitabı tek yönlü liste için küçük bir ADT tanımlar. Bu ADT'nin işlemleri şunlardır: boyutu sorma, boş olup olmadığını sorma, ilk ve son öğeyi okuma, başa ve sona ekleme, baştan silme. Bu notta aynı işlemler Türkçe adlarla yazılır. Listeyi temsil eden sınıfın iskeleti şöyledir:

```
public class TekYonluBagliListe {
    private Dugum bas = null, son = null;
    private int boyut = 0;    // öğe sayısı
    public int boyut() { return boyut; }
    public boolean bosMu() { return boyut == 0; }
    public Integer ilk() {
        return bosMu() ? null : bas.veri; }
      // basaEkle, sonaEkle, ilkOgeyiSil …
}
```

Sınıf üç alan tutar. `bas` ilk düğümü, `son` son düğümü gösterir. `boyut` listedeki öğe sayısıdır. Üçü de `private` olarak tanımlanmıştır: sınıfın dışındaki kod bu alanları doğrudan değiştiremez, ancak sınıfın metotlarını çağırabilir. Böylece listenin tutarlılığı, yani `bas`, `son` ve `boyut`'un her an gerçek durumu yansıtması, yalnız sınıfın kendi metotlarına bağlı kalır.

`boyut()` ve `bosMu()` metotları yalnız `boyut` alanını okur; ikisi de *O(1)*'dir. `ilk()` ise liste boşsa `null`, değilse baştaki değeri döndürür. Dönüş türünün `int` değil `Integer` olmasının nedeni budur: `int` türü `null` değeri alamaz, `Integer` ise bir nesne referansı olduğu için alabilir.



Şekil 5.5: Bir `TekYonluBagliListe` nesnesi. Kesikli kutu liste nesnesidir ve üç alan tutar: `boyut = 3`, ilk düğümü (10) gösteren `bas` ve son düğümü (1) gösteren `son`. Düğümler liste nesnesinin dışında durur; nesne onlara yalnız iki referansla bağlanır.

Goodrich–Tamassia	Bu notta	Görev	Zaman
`size()`, `isEmpty()`	`boyut()`, `bosMu()`	`boyut` alanı her işlemde güncellenir	O(1)
`first()`, `last()`	`ilk()`, `bosMu() ? null : son.veri`	boş listede `null` döner	O(1)
`addFirst`, `addLast`	`basaEkle`, `sonaEkle`	son işaretçisiyle ikisi de sabit süre; boş listeye eklemede `son` da ayarlanır	O(1)
`removeFirst()`	`ilkOgeyiSil()`	silinen düğümü döndürür	O(1)

Tablo 5.3: Ders kitabındaki ADT işlemleri ve bu nottaki karşılıkları.

Tablo 5.3'teki bütün işlemler sabit sürelidir. Bunun bir koşulu vardır: listeyi değiştiren her metot `bas`, `son` ve `boyut`'u birlikte güncel tutmalıdır. Ekleme yapan metot `boyut`'u bir artırır, silen metot bir azaltır. Boş listeye ilk düğüm eklenirken bu düğüm hem baş hem son olur; tek düğümlü liste boşaltılırken `son` da `null` yapılır. Başa ekleme, son işaretçili sona ekleme ve silme metotları bu güncellemeleri yapar. Yalnız gezinerek sona ekleyen sürüm, son işaretçisi olmayan yalın listeyi anlattığı için bu alanları kullanmaz.

> **Sık yapılan hata**
>
> `boyut` alanını tutmaya başlayıp bazı metotlarda güncellemeyi unutmak. Silme metotları `boyut--` yapıp ekleme metotlarından biri `boyut++` yapmıyorsa sayaç gerçeği yansıtmaz. İki ekleme ve bir silmeden sonra `boyut()` −1 bile döndürebilir. `bosMu()` da `boyut == 0`'a baktığı için yanlış cevap verir. Bir sayaç tutulacaksa listeyi değiştiren her metot onu günceller; tutulmayacaksa uzunluk her seferinde sayılır.

5.1.8 Üstünlükler ve zayıflıklar

Tek yönlü bağlı listenin güçlü ve zayıf yanları, düğümlerin bağlarla birbirine tutunmasından doğar. Güçlü yanları şunlardır:

- **Kolay gerçeklenir.** Bir düğüm sınıfı ve bir baş referansı yeterlidir.
- **Başa ekleme ve baştan silme *O(1)*'dir.** Dizide aynı işlemler bütün öğelerin kaydırılmasını gerektirir ve *O(n)*'dir.
- **Araya ekleme kaydırma gerektirmez.** Yeni düğüm, iki bağ değiştirilerek araya girer.
- **Boyutu dinamiktir.** Önceden kapasite ayırmaz. Her ekleme yalnız bir `new` yapar; liste dolmaz, büyütme ve kopyalama yoktur.

Zayıf yanları da aynı yapıdan gelir:

- **Arama yavaştır.** Bir değerin listede olup olmadığı ancak düğümler tek tek denetlenerek anlaşılır: *O(n)*. Liste sıralı olsa bile ikili arama işe yaramaz, çünkü ortadaki düğüme atlamak mümkün değildir.
- **Rastgele erişim yoktur.** *i*. öğeye ulaşmak için baştan gezmek gerekir.
- **Ek bellek harcar.** Her düğüm veriye ek olarak bir referans taşır. Küçük veriler için bu ek yük oransal olarak büyüktür: tek bir `int` saklamak için ayrıca bir referans ve nesne başlığı harcanır.
- **Geriye doğru gezilemez.** Bir düğümden öncekine dönmenin yolu yoktur; öncekini bulmak için baştan yeniden başlanır.

Araya ekleme ve silmenin *O(1)* olduğu söylenirken bir koşul gözden kaçmamalıdır. Bu maliyet, işlemin yapılacağı konumun, daha doğrusu o konumdan önceki düğümün *zaten bilindiği* durum için geçerlidir. Konum yalnız indisle ya da değerle verilmişse önce o düğüme yürümek gerekir ve bu yürüyüş

O(n) sürer. "Bağlı listede araya ekleme *O(1)*'dir" cümlesi, bu yürüyüş hesaba katılmadan söylendiğinde yanıltıcıdır.

> **Sık yapılan hata**
>
> "Bağlı listede ortaya ekleme *O(1)*, dizide *O(n)*; öyleyse bağlı liste her zaman daha hızlıdır" diye düşünmek. Ortadaki konumu bulmak bağlı listede de *O(n)* sürer. Asıl kazanç, aynı konumda art arda çok sayıda ekleme ya da silme yapıldığında ortaya çıkar: konum bir kez bulunur, sonraki işlemlerin her biri sabit sürer. Dizide ise her ekleme yeniden kaydırma ister.

5.2 Tek yönlü liste: gezinme ve ekleme

Bağlı listedeki işlemlerin çoğu iki temel hareketten kurulur: listede baştan sona gezinmek ve birkaç bağı doğru sırayla değiştirmek. Bu kesim önce gezinmeyi ele alır. Uzunluk bulma ve arama, gezinmenin küçük birer değişkesidir. Ardından üç ekleme türü gelir: başa, sona ve belirli bir konuma ekleme. Örneklerin çoğu 10 → 1 → 8 → 11 listesi üzerinde çalışır.

5.2.1 Gezinme ve yazdırma

Bir listenin bütün düğümlerini sırayla ziyaret etmeye **gezinme** (*traversal*) denir. Yazdırma, gezinmenin en yalın örneğidir: her düğüme gelindiğinde onun verisi ekrana basılır.

```
void listeyiYazdir() {
    Dugum gecici = bas;
    System.out.print("Bağlı Liste: ");
    while (gecici != null) {
        System.out.print(gecici.veri + " -> ");
        gecici = gecici.sonraki;
    }
    System.out.println("null");
}
```

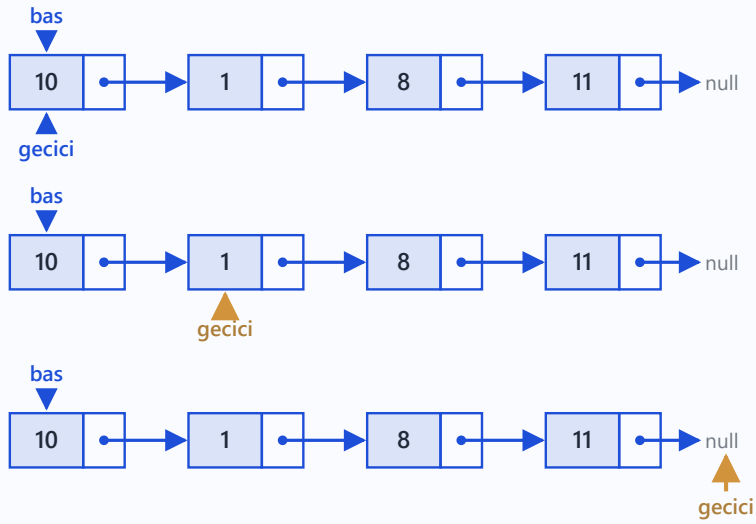
Metot yerel bir `gecici` değişkeniyle çalışır. İlk satır `gecici`'yi `bas`'ın gösterdiği düğüme yöneltir. Döngü, `gecici` bir düğümü gösterdiği sürece döner. Gövdede önce o düğümün verisi yazılır, sonra `gecici = gecici.sonraki` ile bir sonraki düğüme geçilir. Son düğümün verisi yazılıp `gecici` onun `sonraki` alanına ilerleyince `gecici` `null` olur ve döngü biter. En sonda liste sonunu göstermek için `null` yazdırılır.

Burada `bas`'ın hiç değişmediğine dikkat edelim. Gezinme boyunca yalnız yerel `gecici` ilerler. Metot bittiğinde `gecici` ortadan kalkar; liste ise olduğu gibi durur.

Adım	`gecici`	Koşul `gecici != null`	Bu adımda yazılan
`gecici = bas`	düğüm(10)	—	`Bağlı Liste:`
1. tur	düğüm(10) → düğüm(1)	doğru	`10 ->`
2. tur	düğüm(1) → düğüm(8)	doğru	`1 ->`
3. tur	düğüm(8) → düğüm(11)	doğru	`8 ->`
4. tur	düğüm(11) → null	doğru	`11 ->`
döngü sonu	null	yanlış	`null`

Tablo 5.4: 10 → 1 → 8 → 11 listesinde `listeyiYazdir()` izlemesi.

Ekrandaki son çıktı `Bağlı Liste: 10 -> 1 -> 8 -> 11 -> null` olur. Döngü dört düğüm için dört kez döner ve koşul beş kez sınanır. *n* düğümlü listede döngü *n* kez döner; her turda sabit sayıda iş yapıldığı için zaman *O(n)*'dir. Metot yalnız bir yerel değişken kullanır, ek bellek *O(1)*'dir.



Şekil 5.6: Gezinmenin üç anı. (a) `gecici = bas` : iki referans da 10'u gösterir. (b) İlk turun sonunda `gecici` 1'e ilerlemiştir; `bas` yerinde durur. (c) Son turda `gecici` , 11'in `sonraki` alanını, yani `null` 'ı alır ve döngü biter.

Sık yapılan hata

Gezinirken yerel değişken yerine `bas` 'ı ilerletmek: `while (bas != null) { …; bas = bas.sonraki; }` . Döngü bittiğinde `bas` `null` olur. Düğümlere giden tek yol kaybolduğu için liste fiilen silinmiş olur. İkinci bir yazdırma çağrısı yalnız `null` basar. Gezinme her zaman `Dugum gecici = bas;` gibi bir kopya referansla yapılır.

5.2.2 Aynı desen: uzunluk ve arama

Yazdırmanın iskeleti pek çok metotta tekrar eder. Bu iskelete **gezinme deseni** diyebiliriz: `gecici = bas` ile başla; `gecici != null` olduğu sürece o düğümle yapılacak işi yap; `gecici = gecici.sonraki` ile ilerle. Değişen tek şey, döngü gövdesindeki iştir. Listenin uzunluğunu bulan metot, gövdede yazdırmak yerine bir sayacı artırır:

```
int listeUzunlugu() {
    int uzunluk = 0;
    Dugum gecici = bas;
    while (gecici != null) {
        uzunluk++;
        gecici = gecici.sonraki;
    }
    return uzunluk;
}
```

Sayaç sıfırdan başlar ve her düğümde bir artar. Döngü bittiğinde düğüm sayısını tutar. Boş listede döngüye hiç girilmez ve sonuç 0 olur; bu durum için ayrıca bir kontrol gerekmez. Zaman *O(n)*'dir. Uzunluk sık soruluyorsa listede bir `boyut` alanı tutulur ve her ekleme ile silmede güncellenir. O zaman uzunluk sorusu *O(1)*'de cevaplanır; bedeli, her değiştirici metodun sayacı doğru tutmak zorunda olmasıdır.

Arama da aynı desendir. Gövdedeki iş, düğümün verisini aranan değerle karşılaştırmaktır:

```
int ogeyiAra(int aranan) {
    Dugum gecici = bas;
    int konum = 0;
    while (gecici != null) {
        if (gecici.veri == aranan)
            return konum;
        konum++;
        gecici = gecici.sonraki;
    }
    return -1;   // bulunamadı
}
```

Metot bir de `konum` sayacı tutar. Bu sayaç, `gecici`'nin gösterdiği düğümün listedeki sırasıdır; ilk düğümün konumu 0'dır. Aranan değer bulunduğu anda o konum döndürülür ve metot biter. Döngü hiç bulmadan biterse değer listede yoktur ve metot −1 döndürür. −1 geçerli bir indis olamayacağı için "bulunamadı" anlamında güvenle kullanılır.

Aramanın maliyeti aranan değerin yerine bağlıdır. Değer baştaysa tek karşılaştırmada bulunur: en iyi durum *O(1)*'dir. Değer sondaysa ya da hiç yoksa *n* düğümün hepsine bakılır: en kötü durum *O(n)*'dir. Değer listede rastgele bir yerdeyse ortalama yaklaşık *n/2* düğüme bakılır; bu da *O(n)*'dir.

5.2.3 Kod izleme: ogeyiAra(8)

10 → 1 → 8 → 11 listesinde 8'i arayalım. Aranan değer listenin üçüncü düğümündedir.

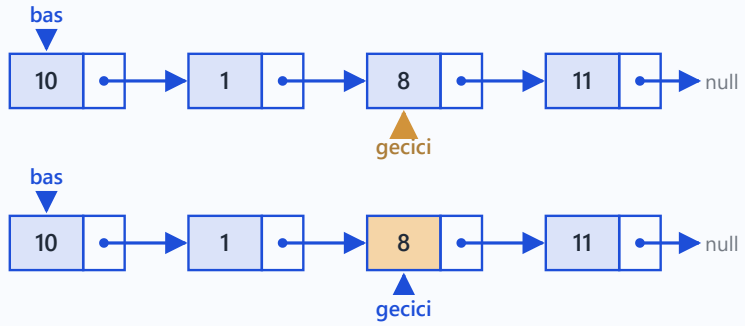
Örnek 5.1

1. **Başlangıç.** `gecici = bas`, yani düğüm(10); `konum = 0`.
2. **1. tur.** `gecici` `null` değil, döngüye girilir. `10 == 8` yanlış. `konum` 1 olur, `gecici` düğüm(1)'e ilerler.
3. **2. tur.** `1 == 8` yanlış. `konum` 2 olur, `gecici` düğüm(8)'e ilerler.
4. **3. tur.** `8 == 8` doğru. Metot `return konum` ile 2 döndürür.

Üç düğüme bakılmış, üç karşılaştırma yapılmıştır. Aynı listede `ogeyiAra(5)` çağrılsaydı dört düğümün dördüne de bakılır, `gecici` `null` olunca döngü biter ve −1 döndürülürdü.

Tur	`gecici`	`konum`	`gecici.veri == 8`	Sonuç
1	düğüm(10)	0	yanlış	ilerle
2	düğüm(1)	1	yanlış	ilerle
3	düğüm(8)	2	doğru	`return 2`

Tablo 5.5: `ogeyiAra(8)` izlemesi. Her satır, o turdaki karşılaştırmadan hemen önceki değerleri gösterir.



Şekil 5.7: `ogeyiAra(8)`'in son iki anı. (a) `konum` 2'ye çıktıktan sonra `gecici` düğüm(8)'e ilerler. (b) Karşılaştırma doğrudur; aranan değer 2. konumda bulunur.

Sık yapılan hata

Aramada `konum++` satırını `if`'ten önce yazmak. Bu durumda karşılaştırma yapılırken sayaç bir fazla olur ve 8 için 3 döndürülür. Sayaç, düğüm denetlendikten *sonra* ve bir sonraki düğüme geçerken artırılır.

5.2.4 Başa ekleme

Listenin başına yeni bir öğe eklemek, bağlı listenin en hızlı işlemlerinden biridir:

```
void basaEkle(int veri) {
    Dugum yeniDugum = new Dugum(veri);
    yeniDugum.sonraki = bas;
    bas = yeniDugum;
    if (son == null) son = bas;
    boyut++;
}
```

Metodun gövdesi beş satırdır. Önce yeni düğüm oluşturulur; `sonraki` alanı `null`'dır. İkinci satırda yeni düğümün `sonraki` alanına `bas`'taki referans yazılır. Böylece yeni düğüm, listenin o anki ilk düğümünü gösterir. Üçüncü satırda `bas` yeni düğüme yöneltilir. Liste artık yeni düğümle başlar ve eski düğümlerin hepsine onun üzerinden ulaşılır.

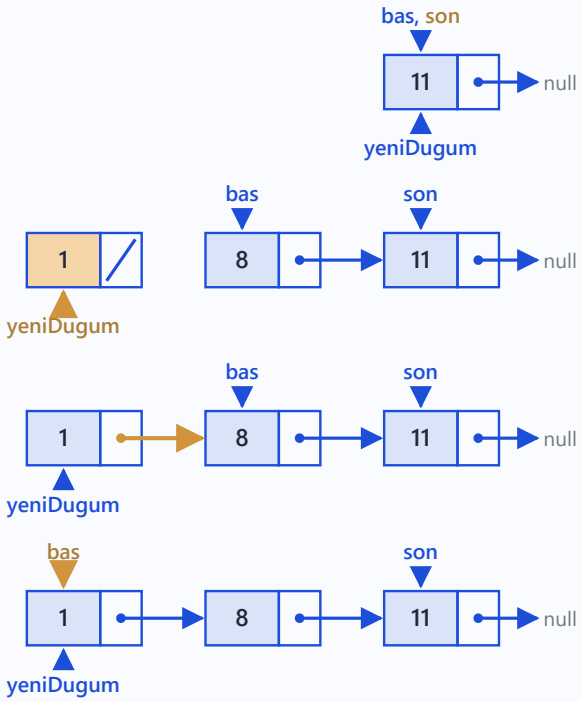
Son iki satır, listenin öteki alanlarını güncel tutar. Liste boşken eklenen düğüm aynı zamanda son düğümdür. `son` o ana kadar `null` olduğu için `if (son == null) son = bas;` satırı onu yeni düğüme yöneltir. Liste boş değilse `son` zaten bir düğümü gösterir ve başa eklemeden etkilenmez. `boyut++` ise öğe sayısını bir artırır.

Satırların sırası bu işlemin özüdür. Önce yeni düğüm eski listeye bağlanır, sonra baş değiştirilir. Boş listede de bağlama doğru çalışır: `bas` `null` olduğu için yeni düğümün `sonraki` alanına yine `null` yazılır ve tek düğümlü bir liste oluşur. Boş listenin tek farkı `son`'un da ayarlanmasıdır.

Çağrı	Satır	`yeniDugum`	`bas`	`son`	`boyut`	Liste
`basaEkle(11)`	`new Dugum(11)`	11 → null	null	null	0	boş
	`yeniDugum.sonraki = bas`	11 → null	null	null	0	boş
	`bas = yeniDugum`	11	düğüm(11)	null	0	11 → null
	`son = bas` ; `boyut++`	11	düğüm(11)	düğüm(11)	1	11 → null
`basaEkle(8)`	`new Dugum(8)`	8 → null	düğüm(11)	düğüm(11)	1	11 → null
	`yeniDugum.sonraki = bas`	8 → 11	düğüm(11)	düğüm(11)	1	11 → null
	`bas = yeniDugum`	8	düğüm(8)	düğüm(11)	1	8 → 11 → null
	`son` aynı; `boyut++`	8	düğüm(8)	düğüm(11)	2	8 → 11 → null
`basaEkle(1)`	`new Dugum(1)`	1 → null	düğüm(8)	düğüm(11)	2	8 → 11 → null
	`yeniDugum.sonraki = bas`	1 → 8	düğüm(8)	düğüm(11)	2	8 → 11 → null
	`bas = yeniDugum`	1	düğüm(1)	düğüm(11)	2	1 → 8 → 11 → null
	`son` aynı; `boyut++`	1	düğüm(1)	düğüm(11)	3	1 → 8 → 11 → null

Tablo 5.6: Boş listeye sırasıyla `basaEkle(11)`, `basaEkle(8)`, `basaEkle(1)` izlemesi.

Tablo 5.6 ilginç bir sonucu da gösterir: öğeler 11, 8, 1 sırasıyla eklenmiş, liste ise 1 → 8 → 11 olmuştur. Art arda başa ekleme, ekleme sırasını tersine çevirir. Bu özellik bir yığının davranışıyla aynıdır: en son eklenen öğe en öndedir.



Şekil 5.8: Başa eklemenin dört anı. (a) `basaEkle(11)` 'in sonu: liste boştu, 11 hem baş hem son olur. (b) `basaEkle(1)` : yeni düğüm 1 oluşturulur; henüz hiçbir yere bağlı değildir, `son` 11'dedir. (c) `yeniDugum.sonraki = bas` : 1, eski baş olan 8'i gösterir. (d) `bas = yeniDugum` : baş 1'e taşınır; liste 1 → 8 → 11 → null olur.

Liste ne kadar uzun olursa olsun başa ekleme bir nesne oluşturma, iki bağ ataması, bir karşılaştırma ve bir sayaç artırmasından ibarettir. Hiçbir düğüm gezilmez, hiçbir öğe kaydırılmaz. Zaman *O(1)*, ek bellek yeni düğüm için *O(1)*'dir. Aynı işlem dizide bütün öğelerin bir hücre sağa kaydırılmasını gerektirir ve *O(n)* sürer.

Sık yapılan hata

Satırları ters sırada yazmak: önce `bas = yeniDugum;` , sonra `yeniDugum.sonraki = bas;` . İlk satırdan sonra `bas` zaten yeni düğümü gösterir. İkinci satır yeni düğümün `sonraki` alanını kendisine yöneltir: düğüm kendini gösterir. Eski listeye giden tek referans kaybolmuştur. Bu listede gezinme hiç bitmez, çünkü her adımda aynı düğüme dönülür.

5.2.5 Sona ekleme

Sona ekleme, yalnız `bas` tutan bir listede daha pahalıdır. Yeni düğümü son düğüme bağlamak gerekir, ama son düğümün nerede olduğu bilinmez. Önce oraya yürünür:

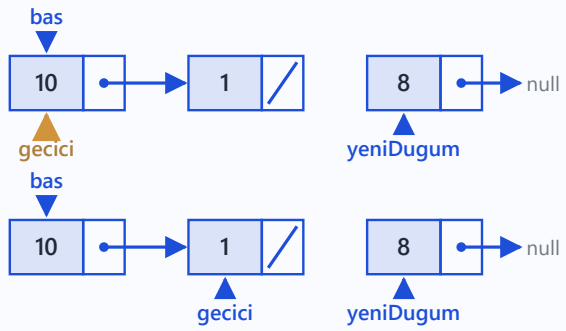
```
void sonaEkle(int veri) {
    Dugum yeniDugum = new Dugum(veri);
    if (bas == null) {
        bas = yeniDugum;
        return;
    }
    Dugum gecici = bas;
    while (gecici.sonraki != null)
        gecici = gecici.sonraki;
    gecici.sonraki = yeniDugum;
}
```

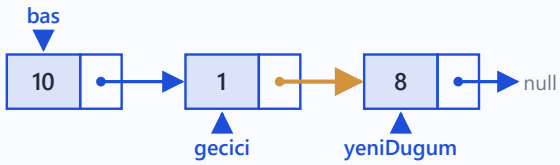
Kod iki durumu ayırır. Liste boşsa yeni düğüm doğrudan baş olur ve metot biter; yürünecek bir liste yoktur. Liste boş değilse `gecici` baştan başlar. Döngü koşulu bu kez `gecici.sonraki != null`'dır. Yani döngü, `gecici`'nin *sonrakisi* var olduğu sürece ilerler ve `gecici` son düğümde durur. Son satır, son düğümün `sonraki` alanına yeni düğümü yazar.

Gezinmedeki koşul ile buradaki koşulun farkı önemlidir. `gecici != null` koşulu, `gecici`'yi son düğümün de ötesine, `null`'a götürür. O noktada son düğüme bir bağ yazılamaz, çünkü elimizde son düğüm kalmamıştır. `gecici.sonraki != null` koşulu ise bir adım erken durur ve `gecici`'yi son düğümün üzerinde bırakır. "Son düğümde işim var" denen her yerde bu ikinci koşul kullanılır.

Çağrı	Dal	`gecici` 'nin uğradığı düğümler	Bağlanan	Liste
`sonaEkle(10)`	`bas == null`	—	`bas = düğüm(10)`	10 → null
`sonaEkle(1)`	gezinme	10 (sonrakisi null, durur)	10.sonraki = düğüm(1)	10 → 1 → null
`sonaEkle(8)`	gezinme	10 → 1 (sonrakisi null, durur)	1.sonraki = düğüm(8)	10 → 1 → 8 → null

Tablo 5.7: Boş listeye sırasıyla `sonaEkle(10)`, `sonaEkle(1)`, `sonaEkle(8)` izlemesi.





Şekil 5.9: `sonaEkle(8)` 'in üç anı. (a) `gecici = bas` : son düğümü bulmak için baştan başlanır. (b) `gecici` düğüm(1)'e gelmiştir ve 1'in `sonraki` alanı `null` 'dır: döngü biter. (c) `gecici.sonraki = yeniDugum` : liste 10 → 1 → 8 → null olur.

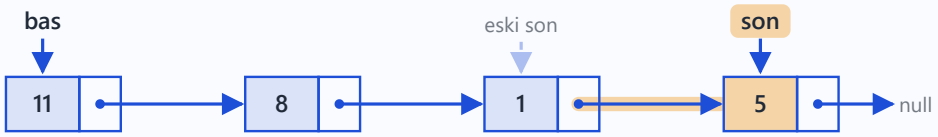
Maliyet, son düğüme kadar yürünen adım sayısıdır. *n* düğümlü listede döngü *n − 1* kez döner; zaman *O(n)*'dir. Bu maliyet tek bir çağrı için kabul edilebilir. Boş listeye art arda *n* öğe sona eklendiğinde ise durum değişir. *k*. eklemede *k − 1* adım yürünür; toplam adım sayısı *0 + 1 + … + (n − 1) = n(n − 1)/2* olur. Bu da $O(n^2)$'dir. Örneğin 1.000 öğe için yaklaşık 500.000 adım atılır. Son işaretçisi bu maliyeti ortadan kaldırır.

5.2.6 Son işaretçisi ile sona ekleme

Sona eklemeyi hızlandırmanın yolu basittir: son düğümü de bir alanda tutmak. Bu alana **son işaretçisi** (*tail pointer*) denir ve kodda `son` diye yazılır. Son düğüm her an bilindiği için ona yürümek gerekmez:

```
class TekYonluBagliListe {
    Dugum bas, son;  int boyut;
    void sonaEkle(int veri) {
        Dugum yeniDugum = new Dugum(veri);
        if (bas == null) {
            bas = son = yeniDugum;
        } else {
            son.sonraki = yeniDugum;
            son = yeniDugum;
        }
        boyut++;
    }
}
```

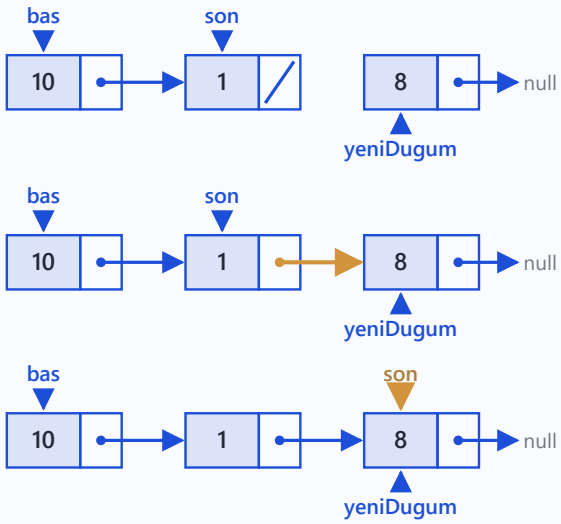
Liste boşsa yeni düğüm hem baş hem son olur: `bas = son = yeniDugum` . Java'da atama sağdan sola yapılır; önce `son` 'a, sonra aynı değer `bas` 'a yazılır. Liste boş değilse iki atama yeter. `son.sonraki = yeniDugum` yeni düğümü eski son düğüme bağlar. `son = yeniDugum` ise son işaretçisini yeni düğüme taşır. İki durumda da en sonda `boyut++` ile öğe sayısı bir artar.



Şekil 5.10: Baş ve son işaretçili liste. 11 → 8 → 1 listesinin sonuna 5 eklenmiştir: eski son (1) yeni düğümü gösterir (vurgulu bağ) ve `son` artık 5'tedir. Hiçbir düğüm gezilmemiştir.

Çağrı	Satır	`bas`	`son`	`boyut`	Liste
`sonaEkle(10)`	`bas = son = yeniDugum`	düğüm(10)	düğüm(10)	0	10 → null
	`boyut++`	düğüm(10)	düğüm(10)	1	10 → null
`sonaEkle(1)`	`son.sonraki = yeniDugum`	düğüm(10)	düğüm(10)	1	10 → 1 → null
	`son = yeniDugum`	düğüm(10)	düğüm(1)	1	10 → 1 → null
	`boyut++`	düğüm(10)	düğüm(1)	2	10 → 1 → null
`sonaEkle(8)`	`son.sonraki = yeniDugum`	düğüm(10)	düğüm(1)	2	10 → 1 → 8 → null
	`son = yeniDugum`	düğüm(10)	düğüm(8)	2	10 → 1 → 8 → null
	`boyut++`	düğüm(10)	düğüm(8)	3	10 → 1 → 8 → null

Tablo 5.8: `son` ile `sonaEkle(10)`, `sonaEkle(1)`, `sonaEkle(8)` izlemesi.



Şekil 5.11: `son` ile `sonaEkle(8)`. (a) Son düğüm zaten bilinir: düğüm(1). (b) `son.sonraki = yeniDugum`: 1 → 8 bağı kurulur. (c) `son = yeniDugum`: son işaretçisi 8'e geçer.

Tablo 5.8'deki her çağrı, listenin uzunluğundan bağımsız olarak en çok iki bağ ataması ve bir sayaç artırması yapar. Sona ekleme artık *O(1)*'dir ve art arda *n* ekleme toplam *O(n)* sürer.

Son işaretçisinin iki bedeli vardır. Birincisi bakım yüküdür: listeyi değiştiren her işlem `son`'u güncel tutmak zorundadır. Boş listeye başa ekleme yapılırken yeni düğüm aynı zamanda son olur. Son düğüm silinirken `son` bir öncekine taşınır. Liste boşalınca `son` da `null` yapılır. Bu adımlardan biri unutulursa `son` artık listede olmayan bir düğümü ya da `null`'ı gösterir ve sonraki ekleme yanlış yere yapılır ya da istisna fırlatır. İkinci bedel, son işaretçisinin her işlemi hızlandırmamasıdır. Sondan silme hâlâ *O(n)*'dir. Silmede değişmesi gereken bağ, son düğümün değil, sondan bir önceki düğümün bağıdır. Tek yönlü listede `son`'dan geriye gidilemez; öncekini bulmak için yine baştan yürünür.

> **Sık yapılan hata**
>
> Boş liste dalında yalnız `bas = yeniDugum;` yazıp `son`'u atlamak. İlk `sonaEkle(5)` sorunsuz görünür, ama `son` `null` kalır. İkinci `sonaEkle(7)` çağrısında liste boş olmadığı için `else` dalına girilir ve `son.sonraki` ifadesi `NullPointerException` fırlatır. Aynı sorun, boş listeye `son`'u güncellemeyen bir `basaEkle` ile ilk düğüm eklendiğinde de ortaya çıkar.

5.2.7 Belirli bir konuma ekleme

Bazen yeni öğenin listenin belirli bir sırasına girmesi istenir. Konumlar 0'dan başlar: `konumaEkle(veri, 0)` başa ekler, `konumaEkle(veri, 2)` yeni öğeyi üçüncü sıraya koyar.

```
void konumaEkle(int veri, int indis) {   // indis 0'dan başlar
    if (indis == 0) { basaEkle(veri); return; }
    Dugum gecici = bas;
    for (int i = 0; i < indis - 1 && gecici != null; i++)
        gecici = gecici.sonraki;
    if (gecici == null) {          // liste indis'ten kısa
        System.out.println("Belirtilen konum bulunamadı.");
        return;
    }
    Dugum yeniDugum = new Dugum(veri);
    yeniDugum.sonraki = gecici.sonraki;   // (1)
    gecici.sonraki = yeniDugum;           // (2)
    if (gecici == son) son = yeniDugum;   // sona eklendiyse
    boyut++;
}
```

Metot üç bölümden oluşur. İlk satır özel durumu ayırır: indis 0 ise ekleme başa yapılır ve bu iş `basaEkle`'ye bırakılır. Başa eklemede `bas` değişir; geri kalan kod ise `bas`'ı hiç değiştirmediği için bu durumu kendisi karşılayamaz.

İkinci bölüm konuma yürür. Yeni düğümü *indis*. sıraya koymak için ondan *önceki* düğüm, yani *indis − 1*. düğüm gerekir; değişecek bağ onun bağıdır. Döngü `gecici`'yi baştan başlatıp *indis − 1* adım ilerletir. Koşuldaki `gecici != null` parçası, liste kısa kalırsa döngünün `null`'da durmasını sağlar.

Döngüden sonra `gecici` `null` ise istenen konuma ulaşılamamıştır; metot bir uyarı yazar ve listeyi değiştirmeden döner.

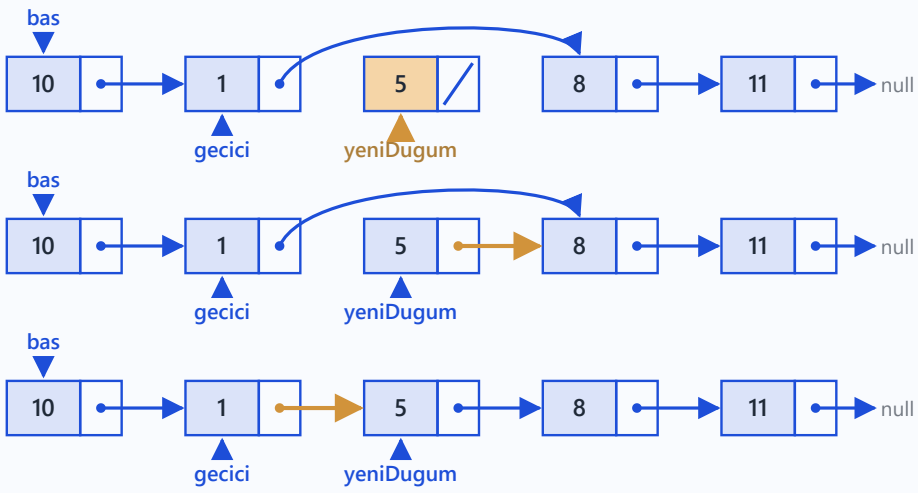
Üçüncü bölüm bağlamadır ve iki satırdır. (1) numaralı satır yeni düğümü kalan listeye bağlar: yeni düğüm, `gecici`'nin o anki sonrakisini gösterir. (2) numaralı satır `gecici`'yi yeni düğüme bağlar. Sıra yine kritiktir: önce (1), sonra (2). Son iki satır sınıfın alanlarını tutarlı tutar: ekleme listenin sonuna yapıldıysa (`gecici == son`) `son` yeni düğüme taşınır ve her durumda `boyut` bir artar.

Örnek 5.2

10 → 1 → 8 → 11 listesinde `konumaEkle(5, 2)` çağrılıyor. 5'in üçüncü sıraya, 1 ile 8'in arasına girmesi bekleniyor.

1. **Özel durum.** İndis 0 değildir; başa ekleme yapılmaz.
2. **Yürüme.** `gecici = bas` (düğüm 10). Döngü *indis − 1 = 1* kez dönecektir. `i = 0 : 0 < 1` doğru, `gecici` düğüm(1)'e ilerler. `i = 1 : 1 < 1` yanlış, döngü biter.
3. **Denetim.** `gecici` düğüm(1)'dir, `null` değildir; konum geçerlidir.
4. **Yeni düğüm.** `new Dugum(5)` . Bu anda 1 hâlâ 8'i gösterir; liste değişmemiştir.
5. **(1)** `yeniDugum.sonraki = gecici.sonraki` : 5, 8'i gösterir. Liste yine değişmemiştir, çünkü listedeki hiçbir düğüm 5'i göstermez.
6. **(2)** `gecici.sonraki = yeniDugum` : 1, 5'i gösterir. Liste 10 → 1 → 5 → 8 → 11 → null olur.

Toplam iki bağ değişmiştir. 8 ve 11 bellekte yerinden kıpırdamamıştır.



Şekil 5.12: Örnek 5.2'nin bağlama adımları. (a) Yeni düğüm 5 oluşturulmuştur (çizimde 8 ve 11 sağa kaydırılmıştır; bellekte hiçbir öğe kaymaz). (b) (1) numaralı satır: 5, 8'i gösterir. (c) (2) numaralı satır: 1, 5'i gösterir.

Sıra ters olsaydı ne olurdu? Önce (2) çalışırsa `gecici.sonraki` artık 5'tir. Ardından (1) satırı `yeniDugum.sonraki = gecici.sonraki` ile 5'in `sonraki` alanına yine 5'i yazar: düğüm kendini gösterir. 8'e giden tek referans kaybolmuştur; 8 ve 11 listeden düşer. Gezinme 10 → 1 → 5 → 5 → … diye sonsuza kadar döner.

Maliyet iki parçadır. Konuma yürümek *indis − 1* adım sürer, yani *O(indis)*'tir; en kötü durumda, sona yakın bir konumda, *O(n)*'dir. Bağlama ise iki atamadır ve *O(1)*'dir. İndis tam olarak liste uzunluğuna eşitse `gecici` son düğümde durur ve yeni düğüm sona eklenir. İndis uzunluktan büyükse `gecici` `null`'a düşer ve "Belirtilen konum bulunamadı." yazılır. Örneğin 10 → 1 → 8 listesinde `konumaEkle(99, 3)` 99'u sona ekler, `konumaEkle(99, 5)` ise listeyi değiştirmez.

> **Sık yapılan hata**
>
> Döngüyü `i < indis` diye yazmak. Bu durumda `gecici` eklenecek yerin öncesinde değil, kendisinde durur ve yeni düğüm bir sıra geriye düşer: `konumaEkle(5, 2)` 5'i 8'den sonraya koyar. Ekleme ve silmede yürünen düğüm her zaman *işlem yapılacak düğümün öncesidir*, çünkü tek yönlü listede değiştirilebilen bağ öncekinin bağıdır.

5.3 Tek yönlü liste: silme ve diğer işlemler

Silme, eklemenin tersidir: bir düğüm, ona giden bağ değiştirilerek zincirden çıkarılır. Java'da düğümü bellekten ayrıca silmek gerekmez. Bir nesneye programın hiçbir yerinden referans kalmadığında **çöp toplayıcı** (*garbage collector*, GC) o nesnenin belleğini kendiliğinden geri alır. Bu yüzden silme işlemlerinin asıl işi, silinecek düğümü listeden ulaşılamaz hâle getirmek ve bu sırada listenin geri kalanını kaybetmemektir. Kesimin ikinci yarısı, bağlı listelerle ilgili klasik problemlere ayrılmıştır: ters çevirme, orta öğe, kopyalama, tekrarları kaldırma, birleştirme ve sıralama.

5.3.1 Baştan silme

İlk düğümü silmek, başa ekleme kadar ucuzdur. Metot silinen düğümü çağırana döndürür; böylece çağıran, silinen değeri kullanabilir.

```
Dugum ilkOgeyiSil() {
    if (bas == null)
        return null;
    Dugum gecici = bas;
    bas = bas.sonraki;  if (bas == null) son = null;
    gecici.sonraki = null;  boyut--;
    return gecici;
}
```

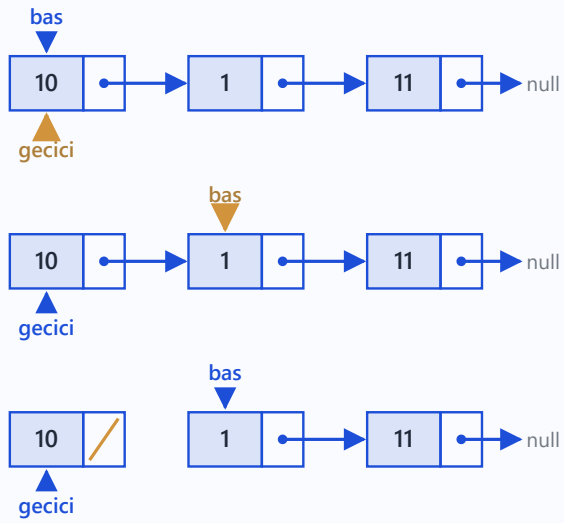
İlk denetim boş listeyi ayırır. Liste boşsa silinecek düğüm yoktur ve `null` döndürülür. Bu denetim olmasaydı sonraki satırlardaki `bas.sonraki` ifadesi `NullPointerException` fırlatırdı.

Liste boş değilse silinecek düğüm önce `gecici`'de saklanır. Ardından `bas = bas.sonraki` ile baş bir sonraki düğüme kaydırılır. Bu satırdan sonra eski ilk düğüme listeden ulaşılamaz; ona yalnız `gecici` ile ulaşılır. Liste bu silmeyle boşaldıysa, yani yeni `bas` `null` ise, `son` da `null` yapılır. Aksi hâlde `son` artık listede olmayan düğümü göstermeye devam ederdi. Son olarak silinen düğümün `sonraki` alanı `null` yapılır, `boyut` bir azaltılır ve düğüm döndürülür.

`gecici.sonraki = null` satırı listenin doğruluğu için zorunlu değildir; liste zaten silinen düğümü göstermez. Satırın amacı, çağırana dönen düğümün hâlâ listenin içini göstermesini önlemektir. Böylece silinen düğüm üzerinden listeye yanlışlıkla dokunulamaz.

Çağrı	`gecici`	Yeni `bas`	`son`	Dönen	Liste
1.	düğüm(10)	düğüm(1)	düğüm(11)	10	1 → 11 → null
2.	düğüm(1)	düğüm(11)	düğüm(11)	1	11 → null
3.	düğüm(11)	null	null	11	boş
4.	—	null	null	`null`	boş

Tablo 5.9: 10 → 1 → 11 listesinde `ilkOgeyiSil()` dört kez çağrılıyor.



Şekil 5.13: İlk çağrının adımları. (a) `gecici = bas` : silinecek düğüm (10) kaybolmasın diye saklanır. (b) `bas = bas.sonraki` : baş 1'e geçer; 10'a listeden ulaşılamaz. (c) `gecici.sonraki = null` : 10 listeden tamamen koparılır.

Her çağrı en çok bir karşılaştırma ve birkaç atama yapar. Liste uzunluğu hiçbir adımı etkilemez: baştan silme *O(1)*'dir. Metodun döndürdüğü düğümü kimse saklamazsa, o düğüme hiçbir referans kalmaz ve çöp toplayıcı belleğini bir süre sonra geri alır.

5.3.2 Sondan silme

Son düğümü silmek tek yönlü listede pahalıdır. Silmek için değişmesi gereken bağ, son düğümü gösteren bağdır; bu bağ sondan bir önceki düğümdedir. O düğüme ulaşmanın tek yolu baştan yürümektir.

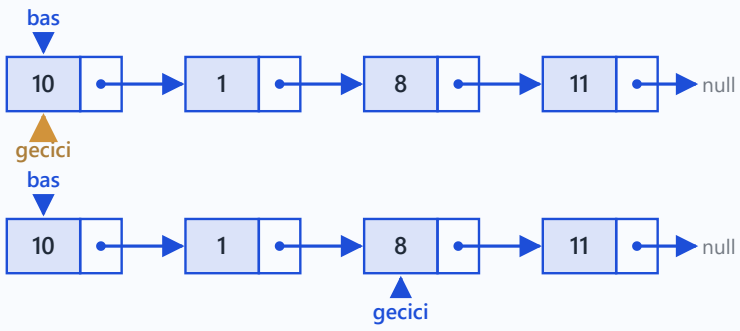
```
void sonOgeyiSil() {
    if (bas == null) return;
    if (bas.sonraki == null) {  // tek düğüm
        bas = son = null;  boyut--;
        return;
    }
    Dugum gecici = bas;
    while (gecici.sonraki.sonraki != null)
        gecici = gecici.sonraki;
    gecici.sonraki = null;  son = gecici;  boyut--;
}
```

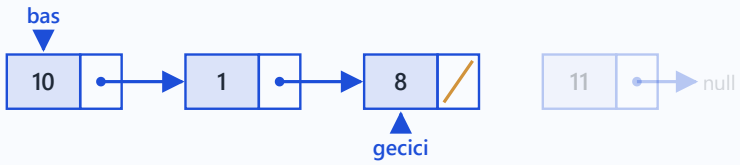
Metot önce iki özel durumu ayırır. Liste boşsa yapılacak bir şey yoktur. Liste tek düğümlüyse, yani `bas.sonraki == null` ise, “sondan bir önceki düğüm” yoktur. Bu durumda liste doğrudan boşaltılır: `bas` ve `son` birlikte `null` yapılır.

Genel durumda `gecici` baştan başlar. Döngü koşulu `gecici.sonraki.sonraki != null`'dır; yani “benden iki sonraki düğüm var mı?” diye sorar. Cevap hayır olduğunda `gecici`'nin sonrakisi son düğümdür; demek ki `gecici` sondan bir önceki düğümdedir. Döngüden sonra `gecici.sonraki = null` ile son düğüm zincirden çıkarılır ve `son` `gecici`'ye taşınır.

Adım	`gecici`	`gecici.sonraki.sonraki`	Karar
başlangıç	düğüm(10)	düğüm(8)	null değil, ilerle
1. ilerleme	düğüm(1)	düğüm(11)	null değil, ilerle
2. ilerleme	düğüm(8)	null	dur: `gecici` sondan bir önceki
bağlama	`gecici.sonraki = null`, `son = gecici`: liste 10 → 1 → 8 → null, `son` = düğüm(8)		

Tablo 5.10: 10 → 1 → 8 → 11 listesinde `sonOgeyiSil()` izlemesi.





Şekil 5.14: Sondan silme. (a) `gecici = bas` : amaç sondan bir önceki düğümü bulmaktır. (b) `gecici` düğüm(8)'e gelmiştir; 8'den iki sonraki bağ `null` 'dır, döngü biter. (c) `gecici.sonraki = null` : 11 listeden çıkar.

n düğümlü listede döngü *n − 2* kez döner; zaman *O(n)*'dir. Son işaretçisi bu maliyeti düşürmez. `son` son düğümü gösterir, ama silmede gereken düğüm ondan bir öncekidir. Tek yönlü listede bir düğümden öncekine geri gidilemediği için `son` 'un varlığı burada işe yaramaz. Bu sınır, çift yönlü listeyi doğuran temel nedendir.

Sık yapılan hata

Tek düğüm denetimini atlamak. Liste yalnız 7'den oluşuyorsa `gecici.sonraki` zaten `null` 'dır ve döngü koşulundaki `gecici.sonraki.sonraki` ifadesi `null` üzerinden alan okumaya çalışır: `NullPointerException` . `a.b.c` gibi zincirleme bir ifade yazılırken aradaki her referansın `null` olmadığından emin olmak gerekir.

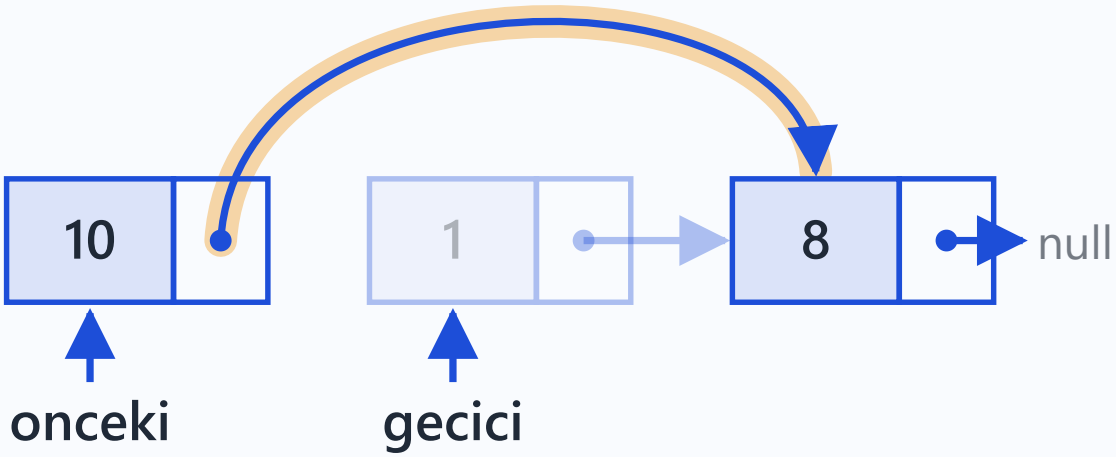
5.3.3 Belirli bir öğeyi silme

Değeri verilen bir öğeyi silmek iki işin birleşimidir: önce öğe aranır, sonra zincirden çıkarılır. Tek yönlü listede bir düğümü çıkarmak için ondan önceki düğümün bağı değiştirilir. Bu yüzden arama sırasında iki işaretçi birlikte yürür: `gecici` denetlenen düğümü, `onceki` ondan bir öncekini gösterir.

```
void ogeyiSil(int silinecek) {
    if (bas == null) return;
    if (bas.veri == silinecek) {
        bas = bas.sonraki;  boyut--;
        if (bas == null) son = null;  return;
    }
    Dugum onceki = bas;
    Dugum gecici = bas.sonraki;
    while (gecici != null) {
        if (gecici.veri == silinecek) {
            onceki.sonraki = gecici.sonraki;
            if (gecici == son) son = onceki;  boyut--;
            return;
        }
        onceki = gecici;
        gecici = gecici.sonraki;
    }
    System.out.println("Bulunamadı.");
}
```

Metot üç durumu sırayla ele alır.

1. **Liste boş:** silinecek bir şey yoktur, metot hemen döner.
2. **Silinecek öğe baştadır:** baştaki düğümün öncesi yoktur; bu yüzden ayrı ele alınır. `bas` bir sonraki düğüme kaydırılır, `boyut` azaltılır. Liste boşaldıysa `son` da `null` yapılır.
3. **Genel durum:** `onceki = bas`, `gecici = bas.sonraki` ile başlanır. Her turda `gecici`'nin verisi aranan değerle karşılaştırılır. Eşitse `onceki.sonraki = gecici.sonraki` ile silinecek düğüm atlanır. Silinen düğüm son düğümse `son` `onceki`'ye taşınır. Eşit değilse iki işaretçi birer düğüm ilerler. Döngü bulmadan biterse “Bulunamadı.” yazılır.



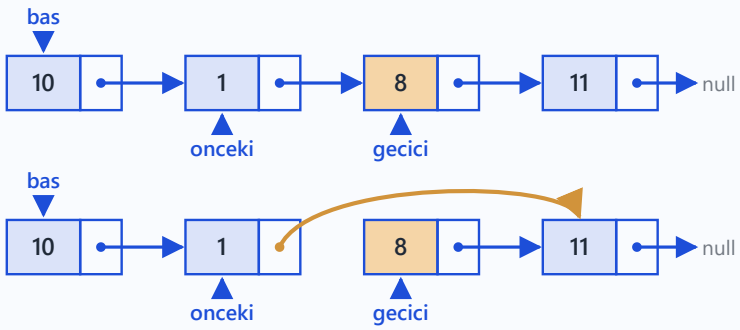
Şekil 5.15: Silme bir atlamadır. `onceki.sonraki = gecici.sonraki` ataması 10'un bağını doğrudan 8'e yöneltir (vurgulu bağ). Silinen düğüm 1 soluk çizilmiştir: hâlâ 8'i gösterir, ama listeden ona giden yol kalmamıştır.

Örnek 5.3

10 → 1 → 8 → 11 listesinde `ogeyiSil(8)` çağrılıyor.

1. **Özel durumlar.** Liste boş değildir; baştaki 10, 8 değildir. İlk iki `if` geçilir.
2. **Başlangıç.** `onceki` = düğüm(10), `gecici` = düğüm(1).
3. **1. tur.** `1 == 8` yanlış. `onceki` düğüm(1)'e, `gecici` düğüm(8)'e ilerler.
4. **2. tur.** `8 == 8` doğru. `onceki.sonraki = gecici.sonraki` : 1 artık 11'i gösterir. `gecici` son düğüm değildir, `son` değişmez. `boyut` azalır ve metot döner.

Sonuç 10 → 1 → 11 → null'dır. Silinen 8'in düğümü hâlâ 11'i gösterir, ama ona listeden ulaşan bir bağ yoktur.



Şekil 5.16: Örnek 5.3'ün son iki adımı. (a) `gecici` 8'i bulmuştur; `onceki` bir geride, 1'dedir. (b) `onceki.sonraki = gecici.sonraki` : 1, 11'i gösterir ve 8 atlanır.

Maliyet aramadan gelir. Silinecek öğeyi bulmak en kötü durumda *n* düğüme bakmayı gerektirir: *O(n)*. Bağlama ise tek atamadır: *O(1)*. Toplam *O(n)*'dir. Metot yalnız ilk eşleşen öğeyi siler. 4 → 8 → 4 → 6 listesinde `ogeyiSil(4)` yalnız baştaki 4'ü siler; ikinci 4 yerinde kalır.

> **Sık yapılan hata**
>
> Silinecek düğümü bulduktan sonra `gecici = gecici.sonraki;` yazıp silindiğini sanmak. Bu satır yalnız yerel `gecici` değişkenini değiştirir; listedeki hiçbir bağa dokunmaz. Listeden bir düğümü çıkarmanın tek yolu, o düğümü gösteren bağı, yani *öncekinin* `sonraki` alanını değiştirmektir.

5.3.4 Listeyi ters çevirme

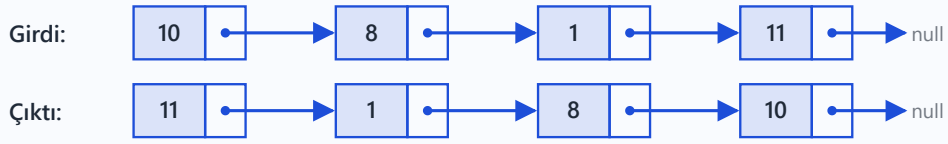
Ters çevirme, bağlı listelerle ilgili en çok sorulan problemlerden biridir. 10 → 8 → 1 → 11 listesi 11 → 1 → 8 → 10 olmalıdır. Kolay ama savurgan bir yol, değerleri bir yığına atıp geri okumak ya da yeni bir liste kurmaktır. Daha iyi yol, hiçbir yeni düğüm oluşturmadan yalnız bağların yönünü çevirmektir.

```
void listeyiTersCevir() {
    son = bas;                 // eski baş yeni son olur
    Dugum onceki = null;
    Dugum suanki = bas;
    Dugum sonraki = null;
    while (suanki != null) {
        sonraki = suanki.sonraki;  // 1) sonrakini sakla
        suanki.sonraki = onceki;   // 2) bağı ters çevir
        onceki = suanki;           // 3) onceki ilerler
        suanki = sonraki;          // 4) suanki ilerler
    }
    bas = onceki;
}
```

Fikir şudur: listeyi baştan sona bir kez gezeriz ve her düğümün `sonraki` bağını, kendinden öncekini gösterecek biçimde çevirir iz. Bunun için üç işaretçi gerekir:

- `suanki` : bağı çevrilecek düğüm.
- `onceki` : `suanki` 'nin yeni bağının göstereceği düğüm; listenin zaten ters çevrilmiş kısmının başı.
- `sonraki` : `suanki` 'nin bağı değişmeden önce saklanan sonraki düğüm. Bağ çevrilince listenin kalanına giden tek yol budur.

Döngünün her turu dört adımdır. (1) Sonraki düğüm saklanır. (2) `suanki` 'nin bağı geriye, `onceki` 'ye çevrilir. (3) `onceki` `suanki` 'ye ilerler. (4) `suanki` saklanan `sonraki` 'ye ilerler. Döngü bittiğinde `suanki` `null` , `onceki` ise eski son düğümdedir. Bu düğüm yeni baştır: `bas = onceki` . Metodun ilk satırı da eski başı yeni son yapar.

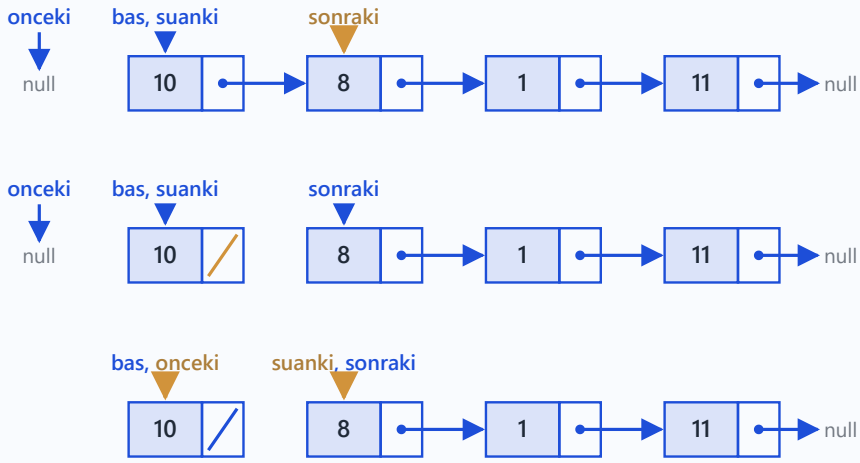


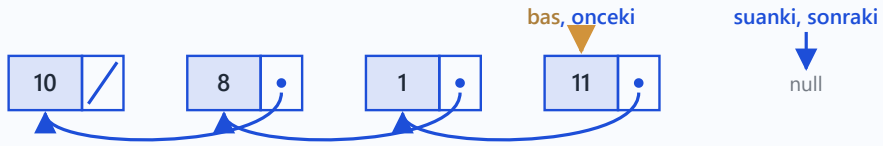
Şekil 5.17: Ters çevirmenin girdisi ve çıktısı. Üstte 10 → 8 → 1 → 11 → null; altta aynı düğümler 11 → 1 → 8 → 10 → null sırasındadır. Yeni düğüm oluşturulmamış, yalnız bağların yönü çevrilmiştir.

Tur	`sonraki`	Çevrilen bağ	`onceki`	`suanki`	Ters kısım
başlangıç	null	—	null	düğüm(10)	boş
1	düğüm(8)	10 → null	düğüm(10)	düğüm(8)	10 → null
2	düğüm(1)	8 → 10	düğüm(8)	düğüm(1)	8 → 10 → null
3	düğüm(11)	1 → 8	düğüm(1)	düğüm(11)	1 → 8 → 10 → null
4	null	11 → 1	düğüm(11)	null	11 → 1 → 8 → 10 → null

Tablo 5.11: 10 → 8 → 1 → 11 listesinde `listeyiTersCevir()` izlemesi. Değerler her turun sonundaki durumdur; "Ters kısım" `onceki`'den başlayan zinciri gösterir.

Her turun sonunda liste iki parçadan oluşur. `onceki`'den başlayan parça ters çevrilmiştir. `suanki`'den başlayan parça henüz el değmemiş hâldedir. Her tur, el değmemiş parçanın ilk düğümünü alıp ters parçanın başına ekler. Bu açıdan ters çevirme, düğümleri sırayla yeni bir listenin başına eklemeye benzer; farkı, yeni düğüm oluşturmadan aynı düğümleri kullanmasıdır.





Şekil 5.18: İlk turun üç adımı ve sonuç. (a) `sonraki = suanki.sonraki` : 8 saklanır. (b) `suanki.sonraki = onceki` : 10'un bağı `null` 'a çevrilir; 8'e yalnız `sonraki` ile ulaşılır. (c) `onceki` ve `suanki` birer düğüm ilerler. (d) Dört tur sonunda `bas = onceki` : 11 → 1 → 8 → 10 → null.

Döngü her düğüm için bir kez döner ve her turda dört atama yapar: zaman *O(n)*'dir. Yalnız üç yerel işaretçi kullanılır, yeni düğüm oluşturulmaz: ek bellek *O(1)*'dir. Boş listede döngüye hiç girilmez ve `bas` `null` kalır; tek düğümlü listede tek tur yapılır ve liste değişmez. İki uç durum da ayrı bir denetim gerektirmez.

Sık yapılan hata

(1) numaralı satırı unutmak ya da (2)'den sonraya koymak. `suanki.sonraki = onceki` çalıştığı anda listenin kalanına giden bağ silinir. Sonraki düğüm önceden saklanmamışsa ona bir daha ulaşılamaz ve ters çevirme ilk düğümden sonra durur. Ayrıca döngü bittiğinde `bas = onceki` yazılmazsa `bas` eski ilk düğümde, yani artık listenin *sonunda* kalır ve liste tek düğümlü görünür.

5.3.5 Orta öğe: yavaş ve hızlı işaretçi

Bir listenin ortadaki öğesini bulmanın akla gelen ilk yolu iki geçiştir: önce uzunluk *n* sayılır, sonra baştan *n/2* adım yürünür. Aynı iş tek geçişte de yapılabilir. Yöntemin adı **yavaş ve hızlı işaretçi**dir (*slow and fast pointers*). İki işaretçi aynı anda baştan yola çıkar. Yavaş olan her turda bir, hızlı olan iki düğüm ilerler. Hızlı işaretçi sona vardığında yavaş olan tam yolun yarısındadır.

```
int ortaOgeyiBul() {
    if (bas == null)
        throw new IllegalStateException("Liste boş");
    Dugum yavas = bas, hizli = bas;
    while (hizli != null && hizli.sonraki != null) {
        yavas = yavas.sonraki;          // 1 adım
        hizli = hizli.sonraki.sonraki;  // 2 adım
    }
    return yavas.veri;
}
```

Döngü koşulu iki parçadır: `hizli != null && hizli.sonraki != null`. Hızlı işaretçinin iki adım atabilmesi için hem kendisinin hem sonrakisinin var olması gerekir. Tek sayıda düğümlü listede hızlı işaretçi son düğümde durur (sonrakisi `null`). Çift sayıda düğümlü listede son düğümü de aşar ve

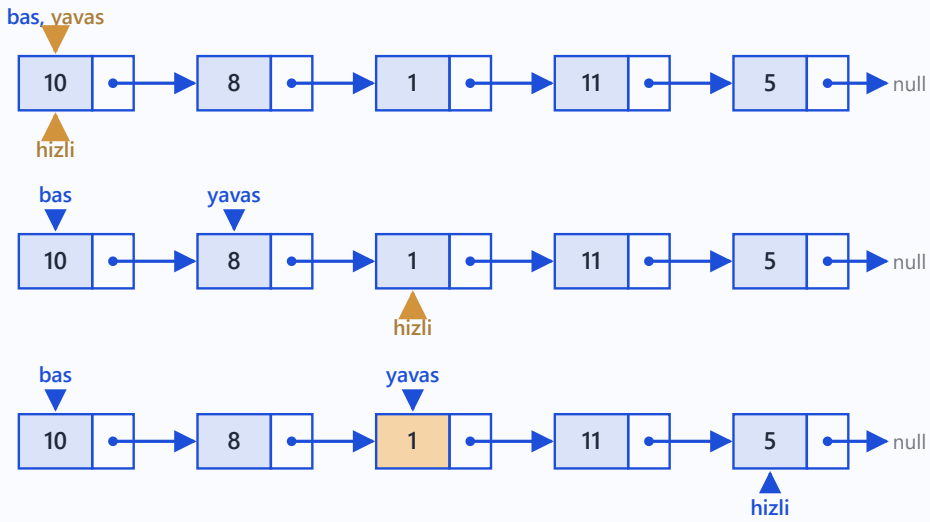
`null`'da durur. İki durumda da döngü biter ve `yavas` ortadadır. Boş listede ortada bir öğe olmadığı için metot baştan bir istisna fırlatır.

Örnek 5.4

10 → 8 → 1 → 11 → 5 listesinin (beş düğüm) ortası aranıyor.

1. **Başlangıç.** `yavas` ve `hizli` düğüm(10)'dadır.
2. **1. tur.** `hizli` (10) ve sonrakisi (8) vardır. `yavas` 8'e, `hizli` iki adımla 1'e geçer.
3. **2. tur.** `hizli` (1) ve sonrakisi (11) vardır. `yavas` 1'e, `hizli` 5'e geçer.
4. **Bitiş.** `hizli.sonraki` `null`'dır; döngü biter. Metot `yavas.veri`, yani 1 döndürür.

İki turda `yavas` 2, `hizli` 4 adım atmıştır. 1, beş düğümün üçüncüsüdür; tam ortadır.



Şekil 5.19: Yavaş ve hızlı işaretçi. (a) İkisi de baştan başlar. (b) İlk turun sonunda `yavas` 8'de, `hizli` 1'dedir. (c) `hizli` son düğüme vardığında `yavas` ortadaki 1'dedir.

Çift uzunlukta iki orta vardır. 10 → 8 → 1 → 11 listesinde 8 ve 1 ortadadır. Bu kod *ikinci* ortayı döndürür: iki turdan sonra `hizli` `null`'a düşer, `yavas` 1'dedir. İlk orta istenirse döngü koşulu `hizli.sonraki != null && hizli.sonraki.sonraki != null` diye değiştirilir.

Hızlı işaretçi en çok *n* düğüm ilerler, döngü yaklaşık *n/2* kez döner: zaman *O(n)*, ek bellek *O(1)*'dir. Toplam adım sayısı iki geçişli yöntemle hemen hemen aynıdır; kazanç, listede bir kez gezinilmesi ve uzunluğun bilinmesine gerek olmamasıdır. Aynı iki işaretçi fikri, listede döngü olup olmadığını anlamak için de kullanılır (Kesim 5.5.3).

Sık yapılan hata

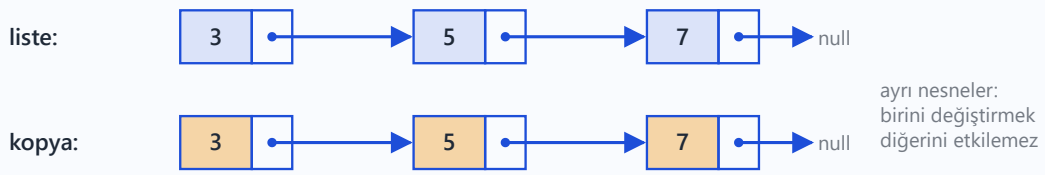
Döngü koşulunu yalnız `hizli.sonraki != null` diye yazmak. Çift uzunluklu listede hızlı işaretçi bir turda `null` 'a düşer; bir sonraki koşul sınamasında `null.sonraki` okunmaya çalışılır ve `NullPointerException` fırlatılır. Koşulun ilk parçası olan `hizli != null` bu yüzden gereklidir ve sırası önemlidir: `&&` ilk parça yanlışsa ikincisini hiç değerlendirmez.

5.3.6 Listeyi kopyalama

Bir listenin kopyasını almak, aynı değerleri taşıyan *yeni düğümlerden* oluşan ayrı bir liste kurmak demektir. Kopya üzerinde yapılan bir değişiklik asıl listeyi etkilememelidir.

```
TekYonluBagliListe kopyala() {
    TekYonluBagliListe kopya =
        new TekYonluBagliListe();
    Dugum gecici = bas;
    while (gecici != null) {
        kopya.sonaEkle(gecici.veri);
        gecici = gecici.sonraki;
    }
    return kopya;
}
```

Metot önce boş bir liste oluşturur. Sonra asıl listede gezinme deseniyle ilerler ve her düğümün değerini `sonaEkle` ile kopyanın sonuna ekler. `sonaEkle` her çağrıda `new Dugum` yaptığı için kopyadaki düğümler asıl listedekilerden tamamen ayrıdır. Bu tür kopyaya **derin kopya** (*deep copy*) denir.



Şekil 5.20: Derin kopya. Üstte asıl liste 3 → 5 → 7, altta kopyası. Değerler aynıdır, düğümler farklı nesnelerdir: birinde yapılan değişiklik ötekini etkilemez.

Kolay görünen ama yanlış olan yol, `kopya.bas = bas` yazmaktır. Bu atama hiçbir düğüm oluşturmaz; yalnız aynı düğümlere giden ikinci bir referans yaratır. Böyle bir "kopya"ya **sığ kopya** (*shallow copy*) denir. Kopyada bir düğümün değeri değiştirildiğinde asıl listede de değişir, çünkü ikisi aynı düğümdür. 10 → 1 → 8 listesinin `kopyala()` ile alınan kopyası yine 10 → 1 → 8'dir; ama iki listenin baş düğümleri farklı nesnelerdir: `kopya.bas == bas` karşılaştırması `false` verir.

Maliyet, `sonaEkle`'nin maliyetine bağlıdır. Son işaretçisi olan listede her ekleme *O(1)*'dir ve kopyalama toplam *O(n)* sürer. Yalnız `bas` tutan listede ise her ekleme kopyanın sonuna kadar yürür: *k*. eklemede *k − 1* adım. Toplam *n(n − 1)/2* adım, yani $O(n^2)$ olur. Ek bellek her iki durumda da *n* yeni düğüm için *O(n)*'dir.

> **Sık yapılan hata**
>
> Kopyalamayı `basaEkle` ile yapmak. Başa ekleme ucuzdur, ama ekleme sırasını tersine çevirir: 10 → 1 → 8 listesinden 8 → 1 → 10 kopyası çıkar. Son işaretçisi yoksa ve sıranın korunması gerekiyorsa, kopyanın son düğümü kopyalama sırasında yerel bir değişkende tutulur ve her yeni düğüm doğrudan ona bağlanır.

5.3.7 Tekrarlayan öğeleri kaldırma

Bir listede aynı değer birden çok kez geçebilir. Amaç, her değerin yalnız ilk geçişini bırakıp sonraki tekrarları silmektir: 3 → 5 → 3 → 7 → 5 listesi 3 → 5 → 7 olmalıdır.

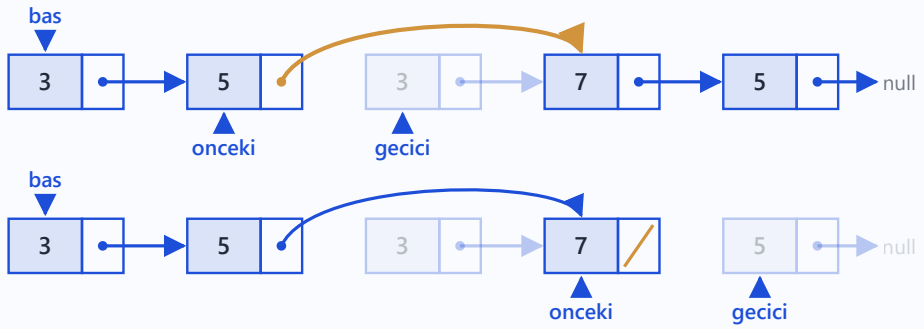
```
void tekrarlayanlariKaldir() {
    TekYonluBagliListe gorulen =
        new TekYonluBagliListe();
    Dugum onceki = null, gecici = bas;
    while (gecici != null) {
        int v = gecici.veri;
        if (gorulen.iceriyor(v)) {
            onceki.sonraki =
                gecici.sonraki;   // atla
            if (gecici == son) son = onceki;  boyut--;
        } else {
            gorulen.basaEkle(v);
            onceki = gecici;
        }
        gecici = gecici.sonraki;
    }
}
```

Görülen değerler ikinci bir listede, `gorulen`'de tutulur. Asıl liste gezilirken her düğümün değeri için şu soru sorulur: bu değer daha önce görüldü mü? Bu soruyu `iceriyor(v)` metodu cevaplar; metot `ogeyiAra(v) != -1` sonucunu döndürür. Cevap evetse düğüm tekrardır ve `onceki.sonraki = gecici.sonraki` ile atlanır. Bu durumda `onceki` yerinde kalır, çünkü yeni sonrakisi de denetlenmemiş bir düğümdür. Cevap hayırsa değer `gorulen`'e eklenir ve `onceki` bu düğüme ilerler. Her iki durumda da `gecici` bir sonraki düğüme geçer.

Kodda `onceki` `null` ile başlar ve ilk düğümde `onceki.sonraki` yazılsaydı istisna fırlatılırdı. Bu hiç olmaz: ilk düğümün değeri daha önce görülmüş olamaz, bu yüzden ilk düğümde her zaman `else` dalı çalışır ve `onceki` ilk düğüme ilerler.

Düğüm	`iceriyor(v)`	Yapılan	`onceki` (sonra)	`gorulen`	Liste
3 (1.)	false	`basaEkle(3)`, `onceki` ilerler	düğüm(3)	3	3 → 5 → 3 → 7 → 5
5 (2.)	false	`basaEkle(5)`, `onceki` ilerler	düğüm(5)	5 → 3	3 → 5 → 3 → 7 → 5
3 (3.)	true	5'in bağı 7'ye çevrilir, 3 atlanır	düğüm(5)	5 → 3	3 → 5 → 7 → 5
7 (4.)	false	`basaEkle(7)`, `onceki` ilerler	düğüm(7)	7 → 5 → 3	3 → 5 → 7 → 5
5 (5.)	true	7'nin bağı `null` olur; düğüm son olduğu için `son = onceki`	düğüm(7)	7 → 5 → 3	3 → 5 → 7

Tablo 5.12: 3 → 5 → 3 → 7 → 5 listesinde `tekrarlayanlariKaldir()` izlemesi.



Şekil 5.21: İki atlama anı. (a) Üçüncü düğümdeki 3 tekrardır: 5'in bağı doğrudan 7'ye yöneltilir; `onceki` 5'te kalır. (b) Son düğümdeki 5 tekrardır: 7'nin bağı `null`'a yöneltilir. Sonuç 3 → 5 → 7 → null'dır.

Maliyeti belirleyen, `iceriyor` çağrılarıdır. `gorulen` listesinde arama, o listenin uzunluğu kadar sürer ve bu uzunluk en çok *n*'dir. *n* düğümün her biri için bir arama yapıldığından toplam zaman $O(n^2)$'dir. Görülen değerler bir liste yerine bir **hash kümesinde** (*hash set*) tutulursa her arama ortalama *O(1)* sürer ve toplam *O(n)*'e iner. Ek bellek her iki yöntemde de farklı değerlerin sayısı kadardır, en çok *O(n)*.

`gorulen`'e eklemenin `basaEkle` ile yapılması da bilinçli bir seçimdir. Sıra burada önemli değildir; yalnız "bu değer var mı?" sorusu sorulur. Başa ekleme *O(1)* olduğu için en ucuz ekleme yolu odur.

Sık yapılan hata

Tekrar eden düğüm atlandıktan sonra `onceki`'yi de ilerletmek. Bu durumda `onceki` silinmiş düğüme geçer. Arka arkaya iki tekrar geldiğinde, örneğin 2 → 2 → 2 listesinde, ikinci atlama silinmiş düğümün bağını değiştirir ve listedeki üçüncü 2 yerinde kalır. Bir düğüm silindiğinde `onceki` yerinde durur; yalnız `gecici` ilerler.

5.3.8 İki sıralı listeyi birleştirme

Küçükten büyüğe sıralı iki liste verilsin. Amaç, ikisindeki bütün öğeleri içeren ve yine sıralı olan üçüncü bir liste kurmaktır. Örneğin 1 → 4 → 7 ile 2 → 3 → 8 birleşince 1 → 2 → 3 → 4 → 7 → 8 çıkar. Bu işlem, birleştirme sıralamasının (*merge sort*) da temel adımıdır.

```
TekYonluBagliListe birlestir(
        TekYonluBagliListe l1,
        TekYonluBagliListe l2) {
    TekYonluBagliListe sonuc =
        new TekYonluBagliListe();
    Dugum g1 = l1.bas, g2 = l2.bas;
    while (g1 != null && g2 != null) {
        if (g1.veri <= g2.veri) {
            sonuc.sonaEkle(g1.veri);
            g1 = g1.sonraki;
        } else {
            sonuc.sonaEkle(g2.veri);
            g2 = g2.sonraki;
        }
    }
```

```
    // kalan öğeleri ekle
    while (g1 != null) {
        sonuc.sonaEkle(g1.veri);
        g1 = g1.sonraki;
    }
    while (g2 != null) {
        sonuc.sonaEkle(g2.veri);
        g2 = g2.sonraki;
    }
    return sonuc;
}
```

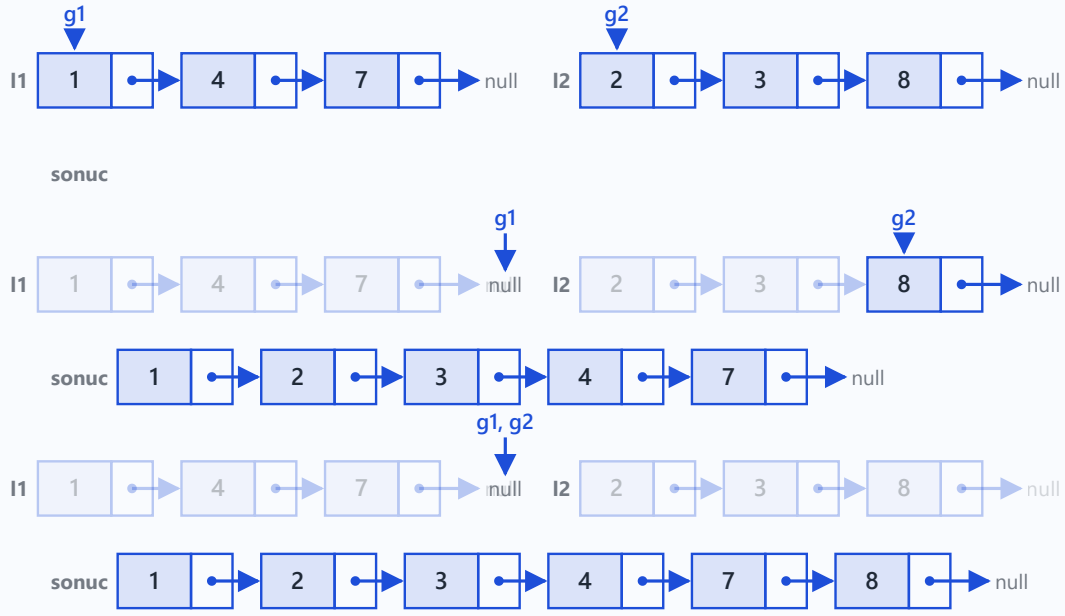
İki işaretçi, `g1` ve `g2`, iki listenin başından başlar. Ana döngü, iki listede de öğe kaldığı sürece döner. Her turda iki işaretçinin gösterdiği değerler karşılaştırılır; küçük olan sonuca eklenir ve yalnız onun işaretçisi ilerler. İki değer eşitse `<=` sayesinde önce birinci listedeki eklenir. Böylece eşit öğelerin girdideki göreli sırası korunur.

Ana döngü, listelerden biri bitince durur. Öteki listede kalan öğeler zaten sıralıdır ve hepsi sonuca eklenmiş öğelerden büyük ya da onlara eşittir. Bu yüzden karşılaştırma yapmadan sırayla sonuca eklenir. İki "kalan" döngüsünden en çok biri çalışır; bitmiş listenin döngüsü koşulu baştan yanlış bulur.

Tur	`g1`	`g2`	Karşılaştırma	Eklenen	`sonuc`
1	1	2	1 ≤ 2 doğru	1	1
2	4	2	4 ≤ 2 yanlış	2	1 → 2

Tur	`g1`	`g2`	Karşılaştırma	Eklenen	`sonuc`
3	4	3	4 ≤ 3 yanlış	3	1 → 2 → 3
4	4	8	4 ≤ 8 doğru	4	1 → 2 → 3 → 4
5	7	8	7 ≤ 8 doğru	7	1 → 2 → 3 → 4 → 7
kalan	null	8	— (ana döngü bitti)	8	1 → 2 → 3 → 4 → 7 → 8

Tablo 5.13: 1 → 4 → 7 ile 2 → 3 → 8'in birleştirilmesi.



Şekil 5.22: Birleştirme. (a) `sonuc` boştur; `g1` ve `g2` iki listenin başındadır. (b) Beş karşılaştırmadan sonra `g1` `null`'a ulaşır; ikinci listede yalnız 8 kalmıştır. (c) Kalan 8 karşılaştırmasız eklenir: 1 → 2 → 3 → 4 → 7 → 8.

Uzunlukları n ve m olan iki liste için her öğe sonuca bir kez eklenir: toplam $n + m$ ekleme. Ana döngüdeki karşılaştırma sayısı en çok $n + m - 1$'dir; örnekte 5'tir. `sonaEkle` son işaretçisiyle $O(1)$ ise toplam zaman $O(n + m)$'dir. Her ekleme sonuca baştan yürüseydi toplam $O((n + m)^2)$ olurdu. Sonuç listesi yeni düğümlerden oluştuğu için ek bellek $O(n + m)$'dir. Asıl düğümler yeniden bağlanarak birleştirme yapılırsa yeni düğüm gerekmez ve ek bellek $O(1)$'e iner; bu durumda girdi listeleri bozulur.

Sık yapılan hata

Kalan öğeleri ekleyen döngüleri unutmak. Ana döngü listelerden biri biter bitmez durur. Örnekteki 8 gibi öteki listede kalan öğeler, kalan döngüleri yazılmamışsa sonuca hiç girmez ve sonuç 1 → 2 → 3 → 4 → 7 olur.

5.3.9 Listeyi sıralama

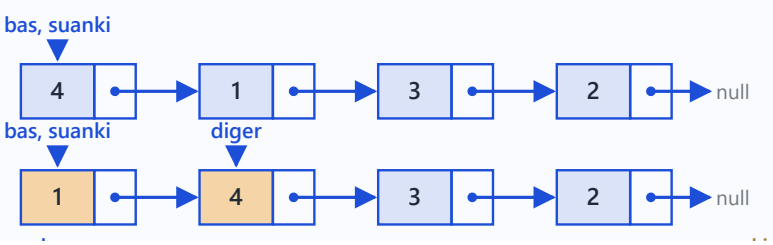
Bir bağlı liste, düğümleri yerinden oynatmadan, yalnız veri alanlarındaki değerler takas edilerek de sıralanabilir. Aşağıdaki kod basit bir değiş tokuş sıralamasıdır:

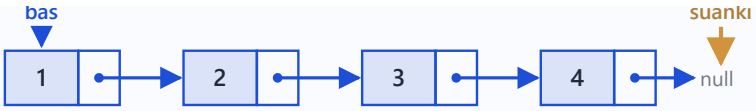
```
void sirala() {
    Dugum suanki = bas;
    while (suanki != null) {
        Dugum diger = suanki.sonraki;
        while (diger != null) {
            if (suanki.veri > diger.veri) {
                int gecici = suanki.veri;
                suanki.veri = diger.veri;
                diger.veri = gecici;
            }
            diger = diger.sonraki;
        }
        suanki = suanki.sonraki;
    }
}
```

Dış döngü `suanki`'yi baştan sona yürütür. İç döngü, `suanki`'den sonraki her düğümü `diger` ile ziyaret eder. `suanki`'deki değer `diger`'dekinden büyükse iki değer **takas** (*swap*) edilir; takas için üçüncü bir değişken, yerel `gecici`, kullanılır. İç döngü bittiğinde `suanki`'nin hücresinde, o hücre ve sonrasındaki değerlerin en küçüğü durur. Dış döngü bir hücre ilerler ve aynı iş kalan kısım için tekrarlanır.

`suanki`	`diger`	Karşılaştırma	Takas	Liste
1. düğüm	2. düğüm	4 > 1	evet	1 → 4 → 3 → 2
1. düğüm	3. düğüm	1 > 3	hayır	1 → 4 → 3 → 2
1. düğüm	4. düğüm	1 > 2	hayır	1 → 4 → 3 → 2
2. düğüm	3. düğüm	4 > 3	evet	1 → 3 → 4 → 2
2. düğüm	4. düğüm	3 > 2	evet	1 → 2 → 4 → 3
3. düğüm	4. düğüm	4 > 3	evet	1 → 2 → 3 → 4
4. düğüm	null	—	—	1 → 2 → 3 → 4

Tablo 5.14: 4 → 1 → 3 → 2 listesinde `sirala()` izlemesi. Konumlar 1'den 4'e kadar numaralanmıştır; düğümler yerinde kalır, yalnız değerler değişir.





Şekil 5.23: Değer takasıyla sıralama. (a) Başlangıç: 4 → 1 → 3 → 2. (b) İlk karşılaştırmada 4 > 1; değerler takas edilir: 1 → 4 → 3 → 2. (c) Sonuç: 1 → 2 → 3 → 4. Düğümlerin yeri ve bağları hiç değişmemiştir.

Tablo 5.14'teki karşılaştırma sayısı 3 + 2 + 1 = 6'dır. Genel olarak `suanki` *k*. düğümdeyken iç döngü *n* − *k* karşılaştırma yapar. Toplam $(n-1) + (n-2) + \dots + 1 = n(n-1)/2$ olur. Bu sayı listenin başlangıçtaki düzenine bağlı değildir; liste zaten sıralı olsa bile aynı sayıda karşılaştırma yapılır. Takas sayısı ise girdiye bağlıdır: örnekte 4'tür. Zaman $O(n^2)$, ek bellek $O(1)$'dir.

Bağlı liste için daha iyi bir yöntem birleştirme sıralamasıdır. Liste yavaş ve hızlı işaretçiyle ortadan ikiye bölünür, iki yarı ayrı ayrı sıralanır ve önceki kesimdeki gibi birleştirilir. Bu yöntem $O(n \log n)$ zamanda çalışır. Dizideki birleştirme sıralamasının aksine, düğümler yeniden bağlanarak birleştirildiğinde ek bir diziye de gerek kalmaz. Hızlı sıralama ve heap sort gibi indisle rastgele erişim isteyen yöntemler ise bağlı listeye pek uymaz.

Sık yapılan hata

Değer takası yerine düğüm takası yapmaya çalışıp yalnız iki yerel değişkeni değiştirmek: `Dugum t = suanki; suanki = diger; diger = t;` . Bu satırlar listedeki hiçbir bağı değiştirmez; yalnız işaretçilerin adlarını karıştırır ve döngüyü bozar. Düğümlerin kendisini yer değiştirmek için hem öncüllerinin hem kendilerinin bağları güncellenmelidir. Değer takası çok daha basittir ve aynı sonucu verir.

5.4 Çift yönlü bağlı liste

Tek yönlü listenin en büyük kısıtı, bir düğümden öncekine dönülememesidir. Sondan silmenin $O(n)$ sürmesi, belirli bir öğeyi silerken iki işaretçinin birlikte yürümek zorunda kalması ve listenin geriye doğru yazdırılamaması hep bu kısıttan doğar. Çift yönlü liste, her düğüme bir bağ daha ekleyerek bu sorunu çözer.

5.4.1 Tanım, düğüm ve yapı

Tanım 5.2

Çift yönlü bağlı liste, her düğümün hem kendinden önceki hem kendinden sonraki düğümün referansını tuttuğu bağlı listedir. İlk düğümün önceki bağı ve son düğümün sonraki bağı `null` 'dır. Liste genellikle hem baş (`bas`) hem son (`son`) referansıyla tutulur ve iki yönde de gezilebilir.

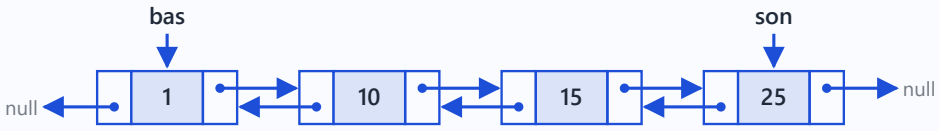


Şekil 5.24: Çift yönlü düğüm üç alandan oluşur: önceki düğümü gösteren `onceki`, değeri tutan `veri` ve sonraki düğümü gösteren `sonraki`.

Düğüm sınıfına tek bir alan eklenir:

```
class Dugum {
    int veri;
    Dugum onceki;
    Dugum sonraki;
    Dugum(int veri) {
        this.veri = veri;
        this.onceki = null;
        this.sonraki = null;
    }
}
```

Liste sınıfı iki alan tutar: `Dugum bas, son;`. Bu yapının getirdiği kazanç ve bedel açıktır. Kazanç, geriye doğru gezinme ile bilinen bir düğümün ve son düğümün *O(1)*'de silinebilmesidir. Bedel ise iki parçadır. Her düğüm ikinci bir referans taşır; tam sayı tutan bir listede bu, düğüm başına harcanan belleği gözle görülür biçimde artırır. Ayrıca her ekleme ve silmede iki yönün bağları birlikte güncellenmelidir; güncellenmesi gereken bağ sayısı aşağı yukarı iki katına çıkar.



Şekil 5.25: Çift yönlü liste 1 ⇄ 10 ⇄ 15 ⇄ 25. `bas` 1'i, `son` 25'i gösterir. Her komşu çifti arasında bir ileri ve bir geri bağ vardır; iki uçtaki boş bağlar `null`'ı gösterir.

Çift yönlü listede her işlem şu kurallara uyar:

- Öğelere hem ileri hem geri yönde erişilir.
- Düğüm eklenince işaretçiler her iki yönde güncellenir: yeni düğümün iki bağı ve komşularının ona dönük birer bağı.
- Düğüm çıkarılınca hem önceki hem sonraki düğümün işaretçileri güncellenir.

- Arama ileri ya da geri yönde gezinerek yapılabilir. Aranan öğenin sona yakın olduğu biliniyorsa aramaya sondan başlamak yürünen yolu kısaltır.

Listenin doğruluğu tek bir değişmezle özetlenebilir: *her* `d` *düğümü için* `d.sonraki` `null` *değilse* `d.sonraki.onceki == d` *'dir;* `d.onceki` `null` *değilse* `d.onceki.sonraki == d` *'dir.* Yani her ileri bağın karşısında onu doğrulayan bir geri bağ vardır. Bir işlemin doğru yazılıp yazılmadığını anlamanın kolay bir yolu, işlemden sonra bu koşulun bozulup bozulmadığına bakmaktır.

5.4.2 Başa ve sona ekleme

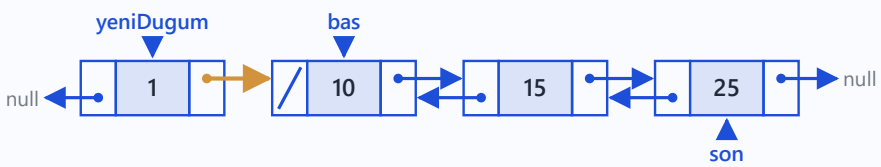
Başa ekleme, tek yönlüdeki iki atamaya bir atama ekler:

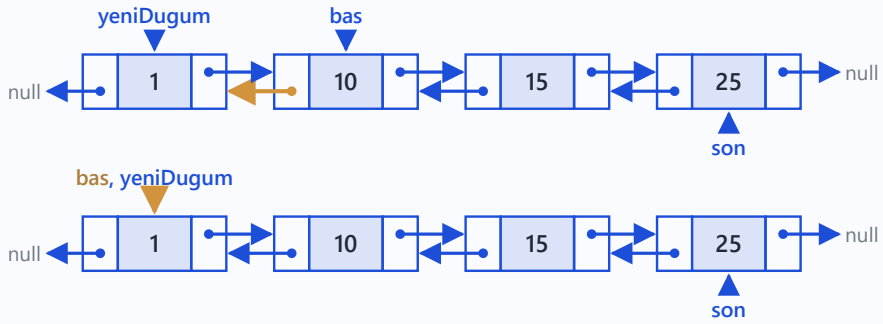
```
public void basaEkle(int veri) {
    Dugum yeniDugum = new Dugum(veri);
    if (bas == null) {
        bas = yeniDugum;
        son = yeniDugum;
    } else {
        yeniDugum.sonraki = bas;
        bas.onceki = yeniDugum;
        bas = yeniDugum;
    }
}
```

Liste boşsa yeni düğüm hem baş hem son olur. Boş değilse üç adım vardır. Yeni düğümün `sonraki` alanı eski başı gösterir (ileri bağ). Eski başın `onceki` alanı yeni düğümü gösterir (geri bağ); bu alan o ana kadar `null`'dı. Son olarak `bas` yeni düğüme taşınır. Yeni düğümün `onceki` alanı zaten `null`'dır ve öyle kalmalıdır, çünkü yeni düğüm artık ilk düğümdür.

Satır	1.onceki	1.sonraki	10.onceki	bas
`new Dugum(1)`	null	null	null	düğüm(10)
`yeniDugum.sonraki = bas`	null	düğüm(10)	null	düğüm(10)
`bas.onceki = yeniDugum`	null	düğüm(10)	düğüm(1)	düğüm(10)
`bas = yeniDugum`	null	düğüm(10)	düğüm(1)	düğüm(1)

Tablo 5.15: 10 ⇄ 15 ⇄ 25 listesinde `basaEkle(1)` izlemesi.



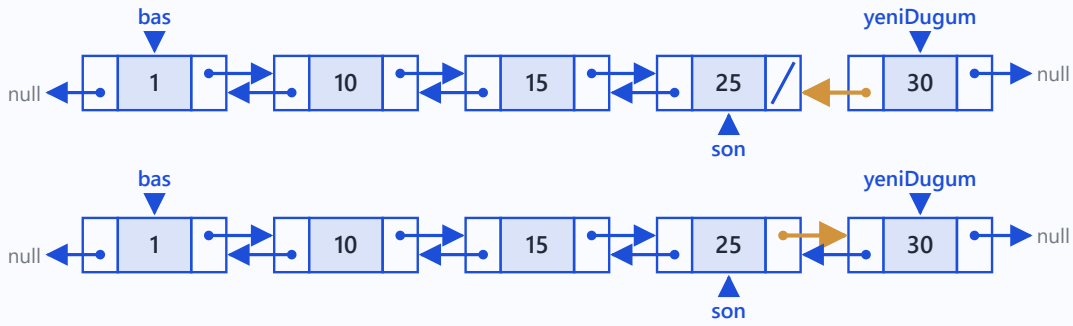


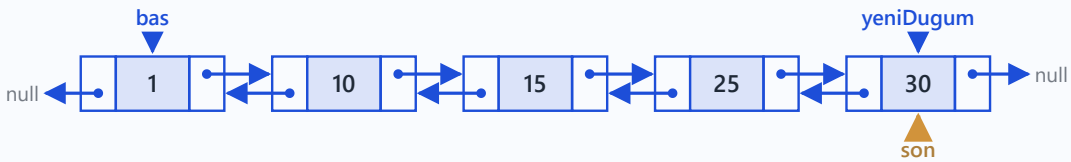
Şekil 5.26: Çift yönlü listede `basaEkle(1)`. (a) `yeniDugum.sonraki = bas`: 1 → 10 ileri bağı. (b) `bas.onceki = yeniDugum`: 10 → 1 geri bağı; 10'un `null` geri bağı gider. (c) `bas = yeniDugum`: liste 1 ⇄ 10 ⇄ 15 ⇄ 25 olur.

Sona ekleme, başa eklemenin ayna görüntüsüdür:

```
public void sonaEkle(int veri) {
    Dugum yeniDugum = new Dugum(veri);
    if (bas == null) {
        bas = yeniDugum;
        son = yeniDugum;
    } else {
        yeniDugum.onceki = son;
        son.sonraki = yeniDugum;
        son = yeniDugum;
    }
}
```

Yeni düğümün `onceki` alanı eski sonu gösterir. Eski sonun `sonraki` alanı yeni düğümü gösterir. Son olarak `son` yeni düğüme taşınır. Son düğüm `son` ile doğrudan bilindiği için hiçbir gezinme yapılmaz. 1 ⇄ 10 ⇄ 15 ⇄ 25 listesine `sonaEkle(30)` uygulanınca 1 ⇄ 10 ⇄ 15 ⇄ 25 ⇄ 30 çıkar ve geri yönde okununca 30, 25, 15, 10, 1 sırası elde edilir.





Şekil 5.27: Çift yönlü listede `sonaEkle(30)`. (a) `yeniDugum.onceki = son`: 30 → 25 geri bağı. (b) `son.sonraki = yeniDugum`: 25 → 30 ileri bağı. (c) `son = yeniDugum`: son işaretçisi 30'a geçer.

İki ekleme de liste uzunluğundan bağımsız olarak sabit sayıda atama yapar: *O(1)*. Tek yönlü listeyle karşılaştırıldığında her eklemede bir atama fazladır. Bu ek atama, geri yönde gezinmenin ve sondan silmenin ucuz olmasının bedelidir.

Sık yapılan hata

Çift yönlü listede yalnız ileri bağları kurmak. `bas.onceki = yeniDugum` satırı unutulursa ileri yazdırma doğru çıkar ve hata gizli kalır. Ama geri yazdırma 10'da durur ve 1'i hiç göstermez; 10'dan geriye doğru silme işlemleri de yanlış düğümü günceller. Her ekleme ve silmeden sonra hem ileri hem geri yazdırma ile listeyi denetlemek bu tür hataları erken yakalar.

5.4.3 Konuma ekleme

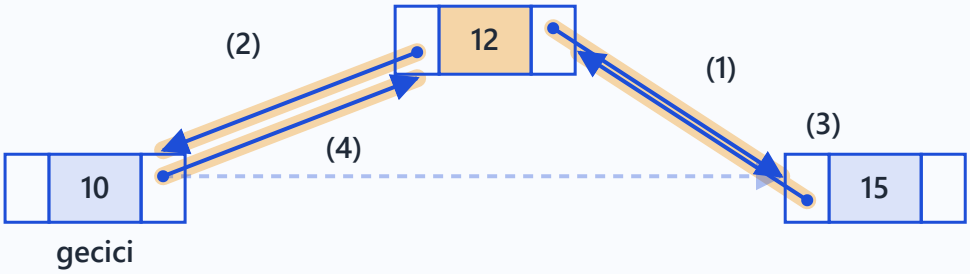
Belirli bir konuma ekleme, tek yönlüdeki gibi önce eklenecek yerin öncesine yürür. Bağlama kısmında ise iki yerine dört bağ değişir.

```
void konumaEkle(int veri, int indis) {   // indis 0'dan başlar
    if (indis == 0) { basaEkle(veri); return; }
    Dugum gecici = bas;
    for (int i = 0; i < indis - 1 && gecici != null; i++)
        gecici = gecici.sonraki;
    if (gecici == null) { System.out.println("Geçersiz konum."); return; }
    if (gecici == son) { sonaEkle(veri); return; }   // son da güncellenir
    Dugum yeni = new Dugum(veri);
    yeni.sonraki = gecici.sonraki;     // (1)
    yeni.onceki = gecici;              // (2)
    gecici.sonraki.onceki = yeni;      // (3)
    gecici.sonraki = yeni;             // (4)
}
```

İlk satırlar tek yönlü sürümle aynıdır: indis 0 başa eklemeye bırakılır, `gecici` *indis − 1*. düğüme yürür, konum geçersizse uyarı yazılır. Yeni bir denetim daha vardır: `gecici` son düğümse ekleme sona yapılır ve iş `sonaEkle`'ye bırakılır. Bu dal gereklidir, çünkü son düğümün `sonraki` alanı `null`'dır. Aşağıdaki (3) numaralı satır `gecici.sonraki.onceki` ifadesinde `null` üzerinden alan okumaya çalışırdı. Ayrıca `son` işaretçisinin güncellenmesi de `sonaEkle`'ye bırakılmış olur.

Genel durumda dört bağ şu sırayla kurulur. Yeni düğümün iki bağı önce kurulur; bu sırada liste hiç değişmez. Sonra komşuların yeni düğüme dönük bağları kurulur.

1. `yeni.sonraki = gecici.sonraki` : yeni düğüm, sağındaki düğümü gösterir.
2. `yeni.onceki = gecici` : yeni düğüm, solundaki düğümü gösterir.
3. `gecici.sonraki.onceki = yeni` : sağdaki düğümün geri bağı yeni düğüme döner.
4. `gecici.sonraki = yeni` : soldaki düğümün ileri bağı yeni düğüme döner.



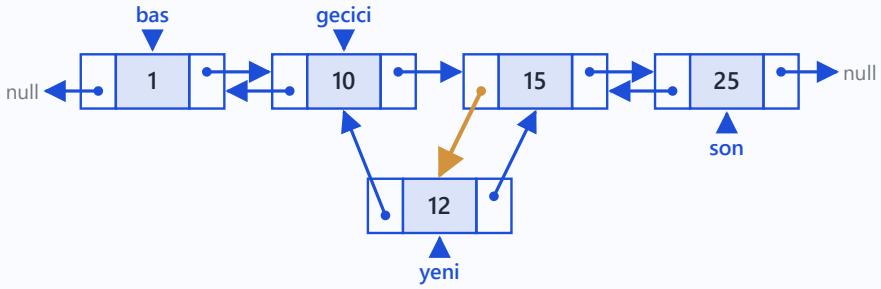
Şekil 5.28: 10 ile 15 arasına 12'nin eklenmesi. Numaralar koddaki satırlardır: (1) 12 → 15, (2) 12 → 10, (3) 15'in geri bağı 12'ye, (4) 10'un ileri bağı 12'ye. Kesikli çizgi, eklemeden önce 10 ile 15 arasındaki doğrudan bağdır.

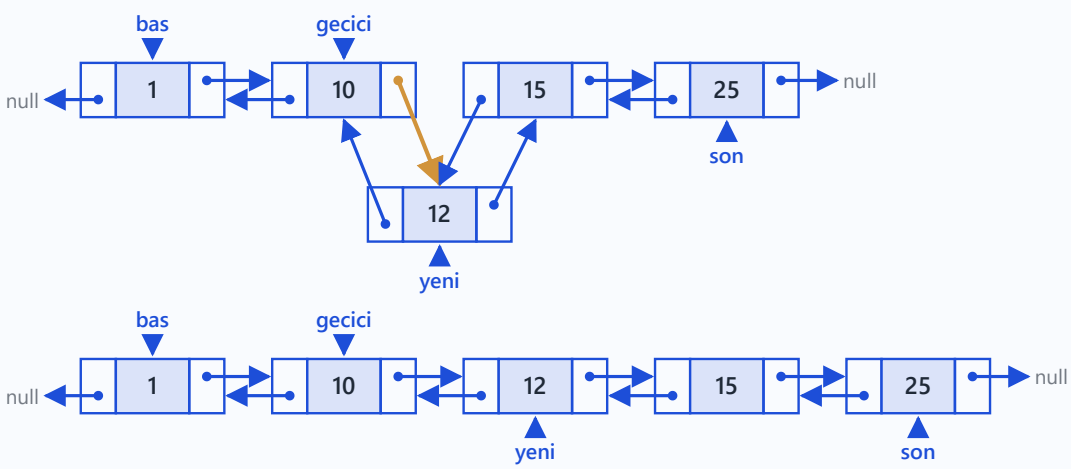
Örnek 5.5

1 ⇄ 10 ⇄ 15 ⇄ 25 listesinde `konumaEkle(12, 2)` çağrılıyor.

1. **Yürüme.** `gecici` , *indis − 1 = 1* adım ilerleyip düğüm(10)'a gelir. 10 son düğüm değildir.
2. **(1)** 12.sonraki = 15. Listede henüz hiçbir düğüm 12'yi göstermez.
3. **(2)** 12.onceki = 10.
4. **(3)** `gecici.sonraki` hâlâ 15'tir; 15.onceki = 12 olur.
5. **(4)** 10.sonraki = 12 olur.

İleri okuma 1, 10, 12, 15, 25; geri okuma 25, 15, 12, 10, 1 verir. Dört bağ değişmiştir.





Şekil 5.29: Örnek 5.5'in son adımları. (a) (3): 15'in geri bağı 12'yi gösterir. (b) (4): 10'un ileri bağı 12'yi gösterir. (c) Sonuç, düzenlenmiş çizimle: 1 ⇄ 10 ⇄ 12 ⇄ 15 ⇄ 25.

(3) ile (4)'ün sırası neden önemlidir? (3) numaralı satır 15'e `gecici.sonraki` üzerinden ulaşır. (4) önce çalışırsa `gecici.sonraki` artık 12'dir. Bu durumda (3) satırı 12.onceki = 12 yapar: 12 kendini gösterir. 15'in geri bağı ise hâlâ 10'u gösterir. İleri okuma 1, 10, 12, 15, 25 verir ve hata gizli kalır; geri okuma ise 25, 15, 10, 1 verir ve 12'yi atlar. Kural: bir düğüme başka bir bağ üzerinden ulaşılıyorsa, o bağ değiştirilmeden önce o düğümle yapılacak iş bitirilir.

Bağlama dört atamadır ve *O(1)*'dir. Konuma yürümek ise tek yönlüde olduğu gibi *O(indis)*, en kötü durumda *O(n)*'dir. İndis listenin ikinci yarısındaysa sondan geriye yürümek yolu kısaltır; bu durumda en çok *n/2* adım atılır, ama maliyet yine *O(n)* sınıfındadır.

5.4.4 Baştan ve sondan silme

Çift yönlü listede iki uçtan silme de sabit sürelidir:

```
void ilkOgeyiSil() {
    if (bas == null) return;  // boş
    if (bas == son) {  // tek düğüm
        bas = son = null;
    } else {
        bas = bas.sonraki;
        bas.onceki = null;
    }
}
```

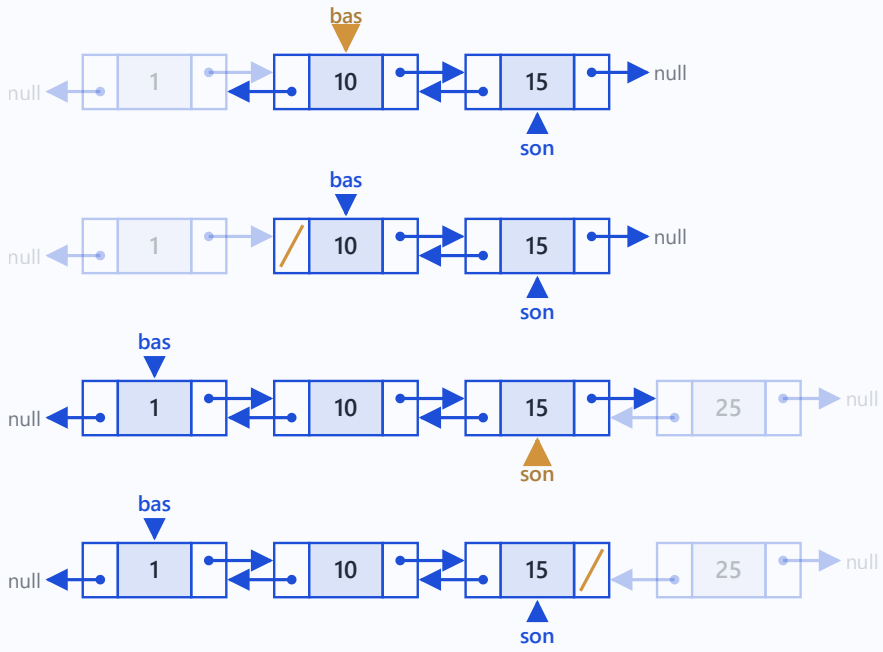
```
void sonOgeyiSil() {
    if (son == null) return;  // boş
    if (bas == son) {  // tek düğüm
        bas = son = null;
    } else {
        son = son.onceki;
        son.sonraki = null;
    }
}
```

İki metot aynı iskeleti kullanır. Liste boşsa yapılacak bir şey yoktur. Liste tek düğümlüyse (`bas == son`) `bas` ve `son` birlikte `null` yapılır ve liste boşalır. Genel durumda baştan silmede `bas` bir sonraki düğüme kayar ve yeni başın `onceki` alanı `null` yapılır. Sondan silmede `son` bir önceki düğüme kayar ve yeni sonun `sonraki` alanı `null` yapılır.

Sondan silmenin hızının kaynağı `son = son.onceki` satırıdır. Sondan bir önceki düğüme tek adımda ulaşılır. Tek yönlü listede aynı düğümü bulmak için bütün listeyi yürümek gerekiyordu.

İşlem	Satır	Etki	Liste
`ilkOgeyiSil()` , 1 ⇄ 10 ⇄ 15	`bas = bas.sonraki`	baş 10'a kayar	(10 hâlâ 1'i geri gösterir)
	`bas.onceki = null`	10'un geri bağı temizlenir	10 ⇄ 15
`sonOgeyiSil()` , 1 ⇄ 10 ⇄ 15 ⇄ 25	`son = son.onceki`	son 15'e kayar	(15 hâlâ 25'i ileri gösterir)
	`son.sonraki = null`	15'in ileri bağı temizlenir	1 ⇄ 10 ⇄ 15

Tablo 5.16: Çift yönlü listede iki uçtan silme izlemesi.



Şekil 5.30: İki uçtan silme. (a) `bas = bas.sonraki` : baş 10'a kayar. (b) `bas.onceki = null` : 10'un geri bağı da temizlenir. (c) `son = son.onceki` : sondan bir önceki düğüme tek adımda ulaşılır. (d) `son.sonraki = null` :

25 listeden ayrılır; sonuç 1 ⇄ 10 ⇄ 15.

İki metot da en çok iki karşılaştırma ve iki atama yapar: *O(1)*. Tek düğüm durumunda `bas` ve `son`'un *birlikte* `null` yapılması önemlidir. Yalnız biri temizlenirse liste bir yandan boş, öbür yandan tek düğümlü görünür ve sonraki ekleme yanlış dala girer.

Sık yapılan hata

Baştan silmede `bas.onceki = null` satırını atlamak. İleri gezinme yine doğru çalışır. Ama yeni baş hâlâ silinmiş düğümü geri gösterir. Geri yazdırma, silinmiş 1'e kadar iner ve onu da yazar. Ayrıca silinmiş düğüme bir referans kaldığı için çöp toplayıcı onu geri alamaz.

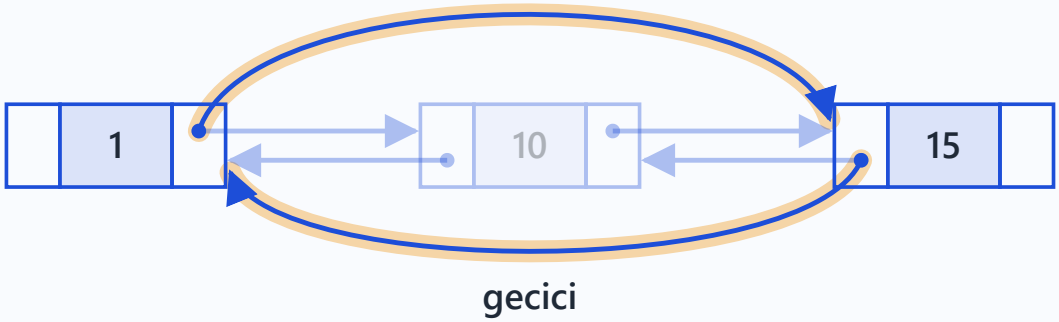
5.4.5 Belirli bir öğeyi silme

Tek yönlü listede değeri verilen öğeyi silmek için `onceki` ve `gecici` birlikte yürümek zorundaydı. Çift yönlü listede tek işaretçi yeter, çünkü bulunan düğümün öncesi `gecici.onceki` ile hemen bilinir.

```
void ogeyiSil(int veri) {
    Dugum gecici = bas;
    while (gecici != null && gecici.veri != veri)
        gecici = gecici.sonraki;
    if (gecici == null) return;  // bulunamadı
    if (gecici.onceki != null)
        gecici.onceki.sonraki = gecici.sonraki;
    else bas = gecici.sonraki;   // baş siliniyor
    if (gecici.sonraki != null)
        gecici.sonraki.onceki = gecici.onceki;
    else son = gecici.onceki;    // son siliniyor
}
```

Önce aranan değer bulunur. Döngü koşulu iki parçalıdır: `gecici` `null` olmadığı ve verisi aranan değer olmadığı sürece ilerlenir. Döngü biter bitmez `gecici` ya aranan düğümdedir ya da `null`'dır. `null` ise değer listede yoktur ve metot döner.

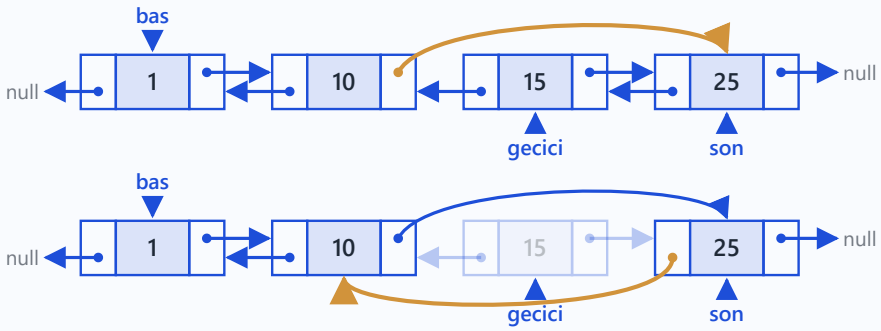
Bulunan düğüm iki bağ değiştirilerek atlanır. Soldaki komşunun ileri bağı sağdaki komşuya yöneltilir; sağdaki komşunun geri bağı soldaki komşuya yöneltilir. İki uç için özel durum vardır. Silinen düğümün solu yoksa düğüm baştır ve ileri bağ yerine `bas` güncellenir. Sağı yoksa düğüm sondur ve geri bağ yerine `son` güncellenir. Tek düğümlü listede iki özel durum birlikte gerçekleşir; `bas` ve `son` ikisi de `null` olur.



Şekil 5.31: Çift yönlü listede aradan silme. 10 silinirken 1'in ileri bağı 15'e, 15'in geri bağı 1'e yöneltilir (vurgulu iki bağ). Silinen düğüm soluk çizilmiştir.

Adım	`gecici`	Denetim	Yapılan
arama	düğüm(1)	1 ≠ 15	ilerle
arama	düğüm(10)	10 ≠ 15	ilerle
arama	düğüm(15)	15 == 15	döngü biter
sol	düğüm(15)	`gecici.onceki` = 10, null değil	10.sonraki = 25
sağ	düğüm(15)	`gecici.sonraki` = 25, null değil	25.onceki = 10

Tablo 5.17: 1 ⇄ 10 ⇄ 15 ⇄ 25 listesinde `ogeyiSil(15)` izlemesi.



Şekil 5.32: `ogeyiSil(15)` 'in bağlama adımları. (a) `gecici.onceki.sonraki = gecici.sonraki` : 10 → 25 ileri bağı; 15 ileri yönde atlanır. (b) `gecici.sonraki.onceki = gecici.onceki` : 25 → 10 geri bağı. Sonuç 1 ⇄ 10 ⇄ 25.

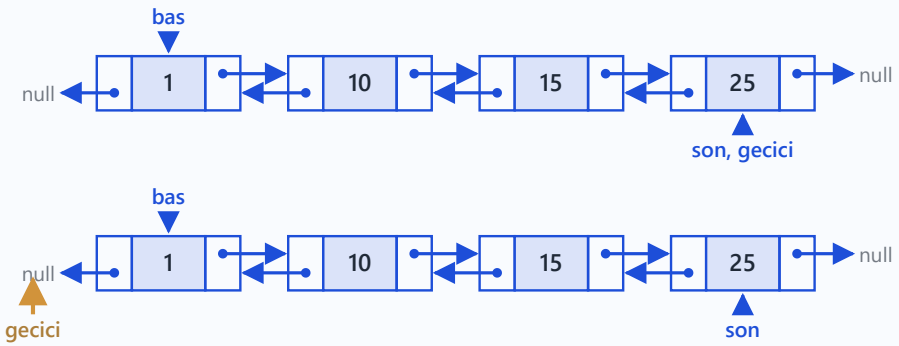
Arama *O(n)*, bağlama *O(1)*'dir; değer verildiğinde toplam *O(n)* olur. Asıl kazanç, silinecek düğümün kendisi zaten elde olduğunda ortaya çıkar. Örneğin bir program düğümlerin referanslarını ayrıca saklıyorsa, bir düğümü aramadan, yalnız onun `onceki` ve `sonraki` alanlarını kullanarak *O(1)*'de silebilir. Tek yönlü listede bu mümkün değildir: elde silinecek düğüm olsa bile öncesini bulmak için baştan yürümek gerekir.

5.4.6 Geri yönde yazdırma

Geri yazdırma, ileri yazdırmanın simetriğidir. Gezinme `son` 'dan başlar ve `onceki` bağları izlenir:

```
void listeyiGeriYazdir() {
    Dugum gecici = son;
    System.out.print("Geri Yönde: ");
    while (gecici != null) {
        System.out.print(gecici.veri + " <-> ");
        gecici = gecici.onceki;
    }
    System.out.println("null");
}
```

Son düğüm `son` ile doğrudan bilindiği için oraya yürümeye gerek yoktur. Her turda `gecici` 'nin verisi yazılır ve `gecici = gecici.onceki` ile bir öncekine geçilir. İlk düğümün `onceki` alanı `null` olduğu için döngü orada biter. 1 ⇄ 10 ⇄ 15 ⇄ 25 listesi için çıktı `Geri Yönde: 25 <-> 15 <-> 10 <-> 1 <-> null` olur.



Şekil 5.33: Geri yazdırma. (a) `gecici = son` : gezinme son düğümden başlar. (b) 1 yazdırıldıktan sonra `gecici` , 1'in `onceki` alanını, yani `null` 'ı alır ve döngü biter.

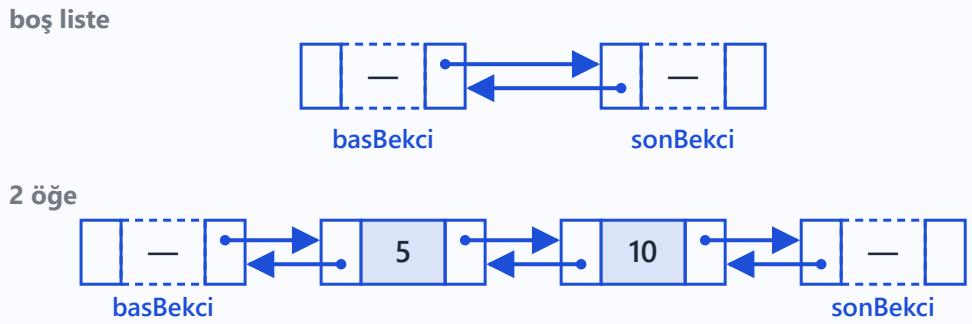
İleri ve geri yazdırma ikisi de her düğümü bir kez ziyaret eder: *O(n)*. Tek yönlü listede de geri yazdırma yapılabilir. Bunun iki yolu vardır: özyineleme ya da değerleri bir yığına atıp geri okumak. İkisi de *O(n)* ek bellek harcar; özyinelemede bu bellek çağrı yığınıdır. Çift yönlü liste aynı işi *O(1)* ek bellekle yapar.

5.4.7 Bekçi düğümler

Çift yönlü listenin kodlarına bakınca her metotta aynı özel durumların tekrar ettiği görülür: liste boş mu, düğüm baş mı, düğüm son mu? Bu denetimler hem kodu uzatır hem de hata kaynağıdır; birini unutmak `NullPointerException` 'a ya da yanlış `bas` / `son` değerine yol açar. Bu özel durumları ortadan kaldırmanın zarif bir yolu vardır.

Tanım 5.3

Bekçi düğüm (*sentinel*), listenin bir ucuna konan ve gerçek bir öğe taşımayan sabit düğümdür. Bekçili çift yönlü listede iki bekçi vardır: başlık bekçisi (`basBekci` , Goodrich–Tamassia'da *header*) ve bitiş bekçisi (`sonBekci` , *trailer*). Gerçek düğümler her zaman bu iki bekçinin arasında durur.



Şekil 5.34: Bekçili çift yönlü liste. Üstte boş liste: yalnız iki bekçi vardır ve birbirini gösterir. Altta iki öğeli liste: 5 ve 10, `basBekci` ile `sonBekci` arasındadır. Bekçilerin veri alanı "—" ile gösterilmiştir.

Bekçilerin getirdiği kolaylık şudur: her gerçek düğümün *her zaman* bir öncesi ve bir sonrası vardır. İlk gerçek düğümün öncesi başlık bekçisi, son gerçek düğümün sonrası bitiş bekçisidir. Böylece `null` denetimine ve `bas` / `son` güncellemesine gerek kalmaz. Boş liste de özel değildir: iki bekçinin birbirini gösterdiği listedir. Liste kurulurken iki bekçi oluşturulup birbirine bağlanır ve bu bağlantı bir daha hiç bozulmaz. Bütün ekleme ve silmeler iki küçük yardımcı metotla yapılabilir:

```
void arayaEkle(int v, Dugum once, Dugum sonra) {
    Dugum yeni = new Dugum(v);
    yeni.onceki = once;
    yeni.sonraki = sonra;
    once.sonraki = yeni;
    sonra.onceki = yeni;  boyut++;
}
void basaEkle(int v) {
    arayaEkle(v, basBekci, basBekci.sonraki); }
void sil(Dugum d) {
    d.onceki.sonraki = d.sonraki;
    d.sonraki.onceki = d.onceki;
    boyut--; }
```

`arayaEkle(v, once, sonra)` , birbirine komşu iki düğümün arasına yeni bir düğüm koyar. Dört atama yapar: yeni düğümün iki bağı ve komşuların ona dönük iki bağı. `basaEkle` , bu metodu başlık bekçisi ile ilk gerçek düğüm arasında çağırır. Sona ekleme de aynı biçimde son gerçek düğüm ile bitiş

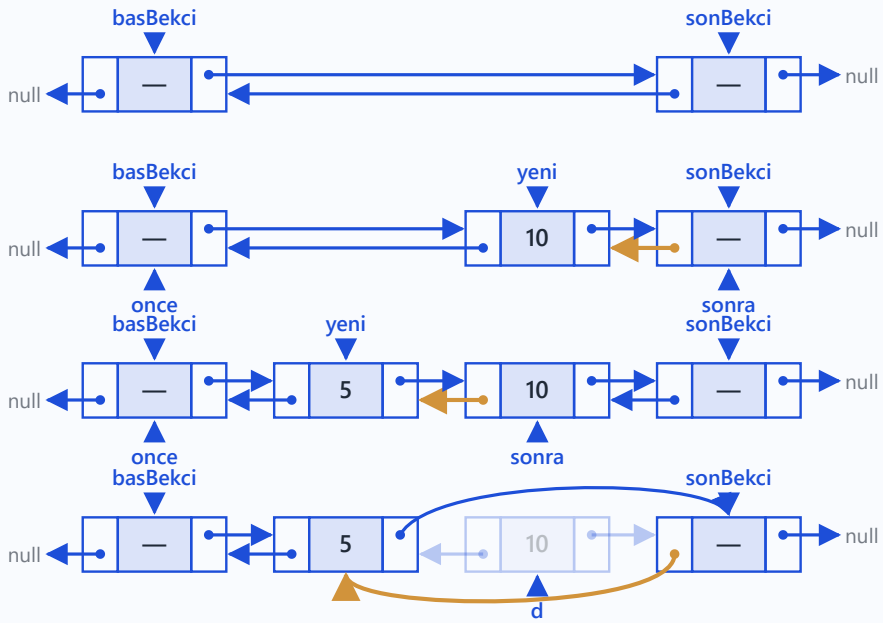
bekçisi arasında yapılır: `arayaEkle(v, sonBekci.onceki, sonBekci)`. `sil(d)` ise `d`'nin iki komşusunu birbirine bağlar. Hiçbir metotta `if` yoktur.

Örnek 5.6

Boş bekçili listeye `basaEkle(10)`, `basaEkle(5)` uygulanıyor; ardından 10'un düğümü `sil` ile siliniyor.

1. **Başlangıç.** `basBekci.sonraki = sonBekci`, `sonBekci.onceki = basBekci`, `boyut = 0`.
2. `basaEkle(10)`. `arayaEkle(10, basBekci, sonBekci)` çağrılır; liste boşken `basBekci.sonraki` bitiş bekçisidir. Dört atama: 10.onceki = başlık, 10.sonraki = bitiş, başlık.sonraki = 10, bitiş.onceki = 10. `boyut = 1`.
3. `basaEkle(5)`. `arayaEkle(5, basBekci, düğüm(10))`. Aynı dört atama: 5, başlık ile 10'un arasına girer. Liste: başlık ⇄ 5 ⇄ 10 ⇄ bitiş; `boyut = 2`.
4. `sil(d)`, **d = düğüm(10).** `d.onceki.sonraki = d.sonraki`: 5 doğrudan bitiş bekçisini gösterir. `d.sonraki.onceki = d.onceki`: bitiş bekçisi 5'i geri gösterir. `boyut = 1`.

10, listenin son gerçek düğümüdür. Bekçisiz listede bu silme `son`'un güncellenmesini gerektirirdi; burada hiçbir özel işlem yapılmamıştır.



Şekil 5.35: Örnek 5.6'nın dört anı. (a) Boş liste: iki bekçi birbirini gösterir. (b) `basaEkle(10)` sonrası. (c) `basaEkle(5)` sonrası: 5, başlık ile 10'un arasındadır. (d) `sil` ile 10 iki yönden atlanmıştır.

Bekçili listede her ekleme ve silme sabit sayıda atamadır: *O(1)*. Kodun kısalığı da ayrı bir kazançtır; daha az dal, daha az hata demektir. Bedeli iki ek düğümdür. Ayrıca gezinme `basBekci.sonraki`'den başlar ve `sonBekci`'ye gelince durur; döngü koşulu `gecici != null` değil, `gecici != sonBekci`'dir.

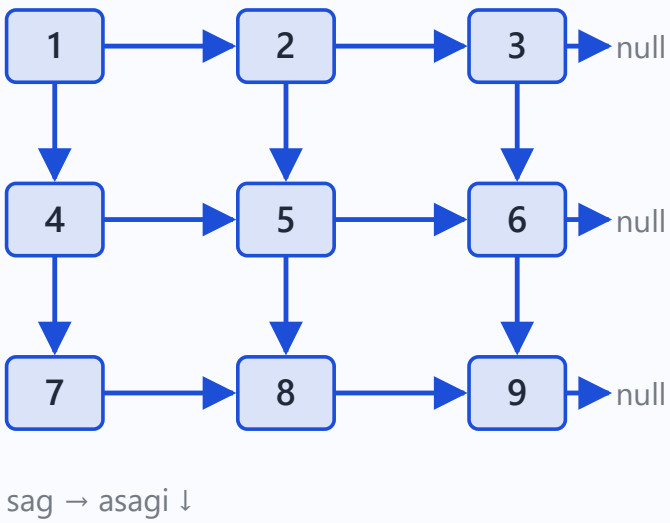
Java'nın `java.util.LinkedList` sınıfı da çift yönlü bir listedir; ders kitaplarındaki pek çok gerçekleme bekçi düğüm kullanır.

Sık yapılan hata

Bekçili listede gezinmeyi `basBekci`'den başlatmak ya da `null`'a kadar sürdürmek. Birinci durumda bekçinin anlamsız verisi de yazdırılır. İkinci durumda gezinme bitiş bekçisini de yazar; sonra bitiş bekçisinin `null` olan dış bağına ulaşıp durur. Bekçiler veri değildir: gezinme ilk gerçek düğümden başlar ve bitiş bekçisinde durur.

5.4.8 Çok yönlü bağlantı: matristen bağlı yapı

Bir düğüm yalnız iki komşu tutmak zorunda değildir. Bağlar, yapının biçimine göre istenen yönlerde kurulabilir. Bunun güzel bir örneği, iki boyutlu bir matrisin bağlı bir yapıya çevrilmesidir. Her düğüm iki bağ tutar: sağındaki hücreyi gösteren `sag` ve altındaki hücreyi gösteren `asagi`. Düğüm sınıfı `class Dugum { int veri; Dugum sag, asagi; }` biçimindedir ve yapıcısı `Dugum(int veri)`'dir.



Şekil 5.36: 3×3 matrisin bağlı yapısı. Her düğüm sağındakini ve altındakini gösterir. Son sütundaki düğümlerin `sag` alanı, son satırdakilerin `asagi` alanı `null`'dır. Bütün yapıya sol üst köşedeki düğümden ulaşılır.

Yapı, sol üst düğümden başlayan iki iç içe gezinmeyle satır satır yazdırılır:

```
void goruntule(Dugum baslangic) {
    Dugum asagi = baslangic;
    while (asagi != null) {       // satırlar
        Dugum sag = asagi;
        while (sag != null) {     // sütunlar
            System.out.print(sag.veri + " ");
            sag = sag.sag;
        }
        System.out.println();
        asagi = asagi.asagi;
    }
}
```

Dış döngü `asagi` bağlarıyla satırları, iç döngü `sag` bağlarıyla bir satırın hücrelerinde gezinir. Her satırın başı bir önceki satırın başının `asagi`'sidir. Şekil 5.36'daki yapı için çıktı üç satırdır: `1 2 3`, `4 5 6`, `7 8 9`. Her düğüm bir kez ziyaret edilir: zaman *O(satır · sütun)*'dur.

Yapıyı kurmak daha ilginçtir. Bir düğümün `sag` ve `asagi` bağları kurulurken sağdaki ve alttaki düğümlerin zaten var olması gerekir. Bu yüzden düğümler sağ alt köşeden başlanarak, satırlarda ve sütunlarda geriye doğru gezinilerek oluşturulur:

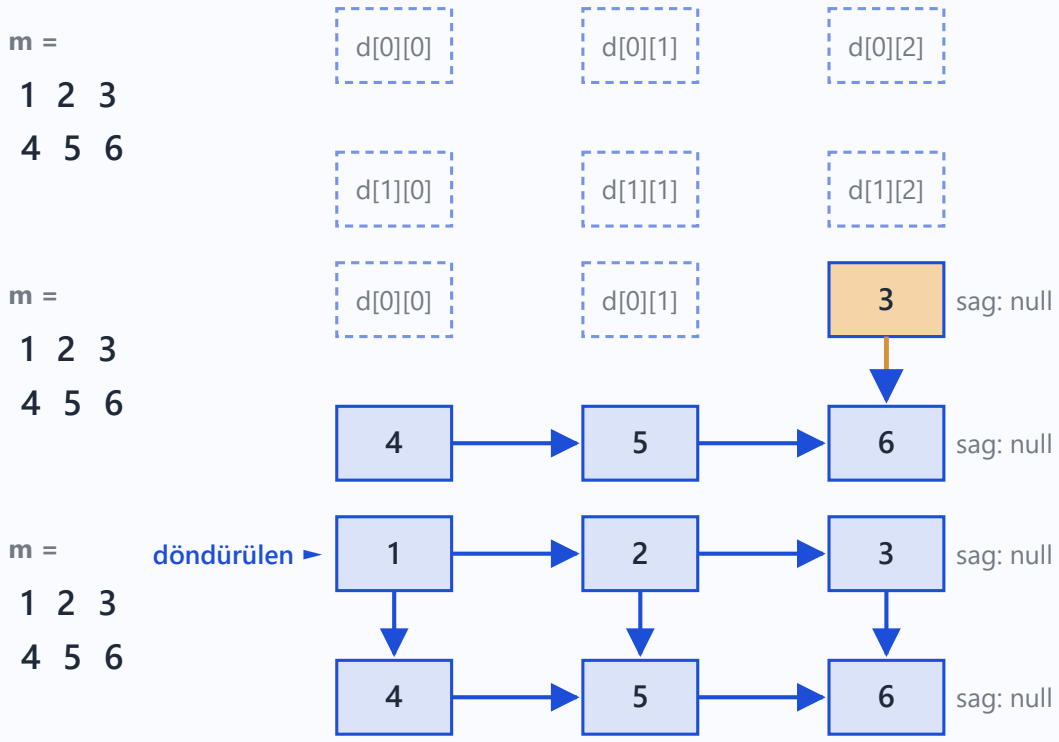
```
Dugum olustur(int[][] m) {
    int r = m.length, c = m[0].length;
    Dugum[][] d = new Dugum[r][c];
    for (int i = r - 1; i >= 0; i--)
        for (int j = c - 1; j >= 0; j--) {
            Dugum n = new Dugum(m[i][j]);
            if (j + 1 < c) n.sag = d[i][j+1];
            if (i + 1 < r) n.asagi = d[i+1][j];
            d[i][j] = n;
        }
    return d[0][0];
}
```

Oluşturulan düğümler, yardımcı bir `d` dizisinde matrisle aynı konumda saklanır. `(i, j)` hücresinin düğümü oluşturulurken sağdaki `d[i][j+1]` ve alttaki `d[i+1][j]` önceden oluşturulmuştur, çünkü döngülerde `i` ve `j` büyükten küçüğe ilerler. Son sütunda sağ komşu, son satırda alt komşu yoktur; bu hücrelerde ilgili bağ `null` kalır. En sonda sol üst köşedeki düğüm döndürülür.

(i, j)	Yeni düğüm	`sag`	`asagi`
(1, 2)	6	yok (son sütun)	yok (son satır)
(1, 1)	5	6	yok (son satır)
(1, 0)	4	5	yok (son satır)
(0, 2)	3	yok (son sütun)	6
(0, 1)	2	3	5

(i, j)	Yeni düğüm	`sag`	`asagi`
(0, 0)	1	2	4

Tablo 5.18: 2×3 matris `{{1, 2, 3}, {4, 5, 6}}` için `olustur(m)` izlemesi.



Şekil 5.37: `olustur(m)` 'nin üç anı. (a) `d` dizisi boştur. (b) Alt satırın üç düğümü ve 3 oluşturulmuştur; 3'ün `asagi` bağı 6'yı gösterir. (c) Altı hücre için altı düğüm oluşmuştur; `d[0][0]` (1) döndürülür.

Her hücre bir kez işlenir ve tek bir düğüm oluşur: zaman *O(satır · sütun)*, ek bellek `d` dizisi için yine *O(satır · sütun)*'dur. *r* satır ve *c* sütunlu bir matriste *r(c − 1)* tane `sag` bağı ve *(r − 1)c* tane `asagi` bağı kurulur. 3×3 matriste bu sayılar 6 ve 6'dır.

`d` tablosu neden gereklidir? Akla ilk gelen özyinelemeli çözüm, `(i, j)` düğümünü oluşturup `sag` için `olustur(i, j+1)`, `asagi` için `olustur(i+1, j)` çağırmaktır. Bu çözüm aynı hücreye farklı yollardan defalarca gelir ve her gelişte yeni bir düğüm oluşturur. Örneğin (1, 1) hücresine hem (0, 1)'in altından hem (1, 0)'ın sağından ulaşılır ve iki ayrı düğüm oluşur. Bir hücre için oluşan düğüm sayısı, sol üst köşeden o hücreye giden sağ-aşağı yol sayısı kadardır. 3×3 matriste bu yaklaşım 9 yerine 19 düğüm oluşturur; matris büyüdükçe fark üstel olarak artar. Üstelik sağdan ve alttan gelen yollar ortak bir düğümde buluşmadığı için ortaya çıkan yapı bir ızgara bile değildir. `d` tablosu her hücrenin düğümünü bir kez oluşturup saklayarak bu sorunu çözer.

Sık yapılan hata

Döngüleri sol üstten başlatmak: `for (int i = 0; i < r; i++)`. Bu durumda `(i, j)` düğümü oluşturulurken `d[i][j+1]` ve `d[i+1][j]` henüz `null`'dır. Bütün `sag` ve `asagi` bağları `null` kalır ve yapı birbirinden kopuk dokuz düğüme dönüşür. Bağlanacak düğüm, bağlayan düğümden önce oluşturulmalıdır.

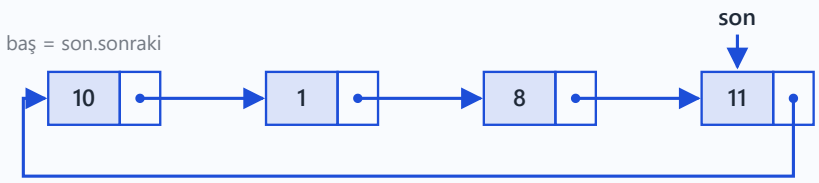
5.5 Dairesel bağlı liste

Bazı işler sırayla ve döne döne yapılır. Bir masa oyununda sıra oyuncudan oyuncuya geçer; son oyuncudan sonra yine ilk oyuncuya gelir. İşletim sistemi işlemciyi bekleyen programlara sırayla kısa süreler verir; listenin sonuna gelince başa döner. Bir müzik çalar "tekrar" kipindeyken son şarkıdan sonra ilk şarkıyı çalar. Bu tür düzenlerde listenin bir sonu yoktur; sonu başına bağlıdır.

5.5.1 Tanım ve sınıf

Tanım 5.4

Dairesel bağlı liste, son düğümün `null` yerine ilk düğümü gösterdiği bağlı listedir. Düğümler bir halka oluşturur. Her düğüm yine kendinden sonrakini gösterir ve listede hiçbir `null` bağ yoktur.



Şekil 5.38: Dairesel liste 10 → 1 → 8 → 11. Son düğüm (11) yeniden ilk düğümü (10) gösterir. Liste yalnız `son` referansıyla tutulur; baş düğüm `son.sonraki`'dir.

Dairesel listede yalnız `son`'u tutmak yeterlidir. Baş düğüm her zaman `son.sonraki`'dir. Böylece tek bir referansla hem başa hem sona *O(1)*'de ulaşılır. Ayrıca bir `bas` alanı tutulsaydı her işlemde iki referansın güncellenmesi gerekirdi; bu hem gereksiz iş hem de ek bir hata kaynağı olurdu.

```
public class DaireselBagliListe {
    class Dugum {
        int veri;
        Dugum sonraki;
        Dugum(int veri) {
            this.veri = veri;
            this.sonraki = null;
        }
    }
    private Dugum son;  // baş = son.sonraki
    private int boyut;
}
```

Sınıf, düğüm sınıfını kendi içinde tanımlar ve iki alan tutar: `son` ve `boyut`. İki özel durum vardır. Boş listede `son == null`'dır. Tek düğümlü listede düğüm kendini gösterir: `son.sonraki == son`. Bu ikinci durum ilk bakışta tuhaf görünse de tanımla uyumludur: tek düğüm hem ilk hem son düğümdür ve son düğüm ilk düğümü gösterir.

Sık yapılan hata

Dairesel listeyi gezerken tek yönlü listedeki `while (g != null)` koşulunu kullanmak. Halkada `null` yoktur; `g` hiçbir zaman `null` olmaz ve döngü sonsuza kadar döner. Dairesel listede gezinme, başlanan düğüme geri gelindiğinde ya da `son`'a ulaşıldığında durdurulur.

5.5.2 Dairesel listeyi yazdırma

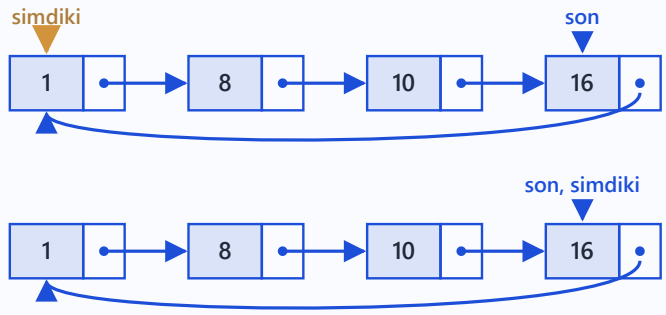
```
void yazdir() {
    if (son == null) return;
    Dugum simdiki = son.sonraki;
    while (simdiki != son) {
        System.out.print(simdiki.veri + " ");
        simdiki = simdiki.sonraki;
    }
    System.out.print(simdiki.veri + " ");
}
```

Liste boşsa yazdırılacak bir şey yoktur. Değilse gezinme baş düğümden, yani `son.sonraki`'den başlar. Döngü koşulu `simdiki != son`'dur: son düğüme gelinceye kadar her düğümün verisi yazılır ve bir sonrakine geçilir. Döngü, `simdiki` son düğüme geldiğinde durur. Son düğümün verisi henüz yazılmamıştır; bu yüzden döngüden sonra ayrıca yazdırılır.

Adım	`simdiki`	`simdiki != son`	Yazılan
başlangıç	düğüm(1)	—	—

Adım	`simdiki`	`simdiki != son`	Yazılan
1. tur	düğüm(1)	doğru	1
2. tur	düğüm(8)	doğru	8
3. tur	düğüm(10)	doğru	10
döngü sonu	düğüm(16)	yanlış	—
son satır	düğüm(16)	—	16

Tablo 5.19: 1 → 8 → 10 → 16 → (1) halkasında `yazdir()` izlemesi (`son` = düğüm 16).



Şekil 5.39: Dairesel listeyi yazdırma. (a) `simdiki = son.sonraki` : gezinme baş düğüm 1'den başlar. (b) `simdiki` son düğüme (16) gelince döngü biter; `null` 'a hiç ulaşılmaz.

Çıktı `1 8 10 16` 'dır. Her düğüm bir kez yazılır: zaman *O(n)*'dir. Tek düğümlü listede `simdiki` ilk anda `son` 'a eşittir, döngüye hiç girilmez ve yalnız son satır çalışır; tek düğüm doğru biçimde bir kez yazılır.

Aynı gezinme bir `do-while` döngüsüyle de yazılabilir. Baştan başlanır; önce düğüm yazılır, sonra ilerlenir; baş düğüme geri gelinince durulur. `do-while` gövdeyi koşuldan önce bir kez çalıştırır. Bu yüzden ayrı bir "son düğümü yazdır" satırına gerek kalmaz. İki yazım da aynı işi yapar.

5.5.3 Döngü tespiti: Floyd algoritması

Dairesel listede halka bilerek kurulur. Bazen de bir program hatası tek yönlü listede istenmeyen bir döngü kurar: bir düğümün `sonraki` alanı yanlışlıkla daha önceki bir düğümü gösterir. Böyle bir listede yazdırma ya da uzunluk bulma hiç bitmez. Bir listede döngü olup olmadığını anlamak bu yüzden pratik bir sorudur.

İlk akla gelen fikir, gezinirken baş düğüme geri dönülüp dönülmediğine bakmaktır. Bu yetmez, çünkü döngü baş düğüme değil, listenin ortasındaki bir düğüme bağlanabilir. Örneğin 1 → 2 → 3 → 4 → 5 → 6 listesinde 6 yeniden 3'ü gösteriyorsa gezinme 3, 4, 5, 6 arasında sonsuza kadar döner ama 1'e hiç dönmez. Ziyaret edilen düğümleri bir kümede saklamak işe yarar, ama *O(n)* ek bellek ister.

Floyd'un döngü tespit algoritması aynı soruyu *O(1)* ek bellekle cevaplar. Yöntem, orta öğeyi bulmada kullanılan yavaş ve hızlı işaretçi fikrinin aynısıdır. Bu yüzden "kaplumbağa ve tavşan" algoritması diye de anılır.

```
boolean donguVarMi(Dugum bas) {
    Dugum yavas = bas, hizli = bas;
    while (hizli != null &&
           hizli.sonraki != null) {
        yavas = yavas.sonraki;
        hizli = hizli.sonraki.sonraki;
        if (yavas == hizli) return true;
    }
    return false;
}
```

Yavaş işaretçi her turda bir, hızlı işaretçi iki düğüm ilerler. İki sonuç mümkündür:

- **Döngü yoksa** hızlı işaretçi listenin sonuna ulaşır: kendisi ya da sonrakisi `null` olur, döngü koşulu yanlışlanır ve metot `false` döndürür.
- **Döngü varsa** iki işaretçi de sonunda döngünün içine girer ve orada dönmeye başlar. Hızlı işaretçi her turda yavaş olana bir düğüm yaklaşır. Aralarındaki uzaklık her turda bir azaldığı için hızlı işaretçi yavaş olanı atlayıp geçemez; bir turda tam onun üzerine gelir. O anda `yavas == hizli` olur ve metot `true` döndürür.

Teorem 5.1 (Floyd algoritmasının doğruluğu)

Tek yönlü bir listede döngü varsa yavaş ve hızlı işaretçi en çok n tur sonra aynı düğümde buluşur; döngü yoksa hızlı işaretçi en çok $n/2$ tur sonra listenin sonuna ulaşır. Burada n, listedeki farklı düğüm sayısıdır.

İspat. Döngü yoksa hızlı işaretçi her turda iki düğüm ilerler ve n düğümlük yolu en çok $n/2$ turda tüketir. Döngü varsa, döngüden önceki kısmın uzunluğu μ, döngünün uzunluğu λ olsun ($\mu + \lambda = n$). Yavaş işaretçi μ tur sonra döngüye girer; hızlı işaretçi o anda zaten döngünün içindedir. Bu andan sonra hızlı işaretçinin, döngü üzerinde ileri doğru sayıldığında yavaşın kaç düğüm arkasında kaldığına d diyelim; $0 \le d < \lambda$'dır. Her turda yavaş bir, hızlı iki adım attığı için bu uzaklık tam bir azalır. d tur sonra uzaklık sıfır olur ve işaretçiler buluşur. Toplam tur sayısı $\mu + d < \mu + \lambda = n$'dir. Tek istisna $\mu = 0$ durumudur, yani döngünün baş düğüme dönmesidir. O zaman başlangıçta $d = 0$'dır, ama kod eşitliği ilk ilerlemeden önce sınamaz. İşaretçiler $\lambda = n$ tur sonra yeniden aynı düğüme gelir; sınır yine n'dir. □

Örnek 5.7

Liste 1 → 2 → 3 → 4 → 5 → 6 ve 6 yeniden 3'ü gösteriyor.

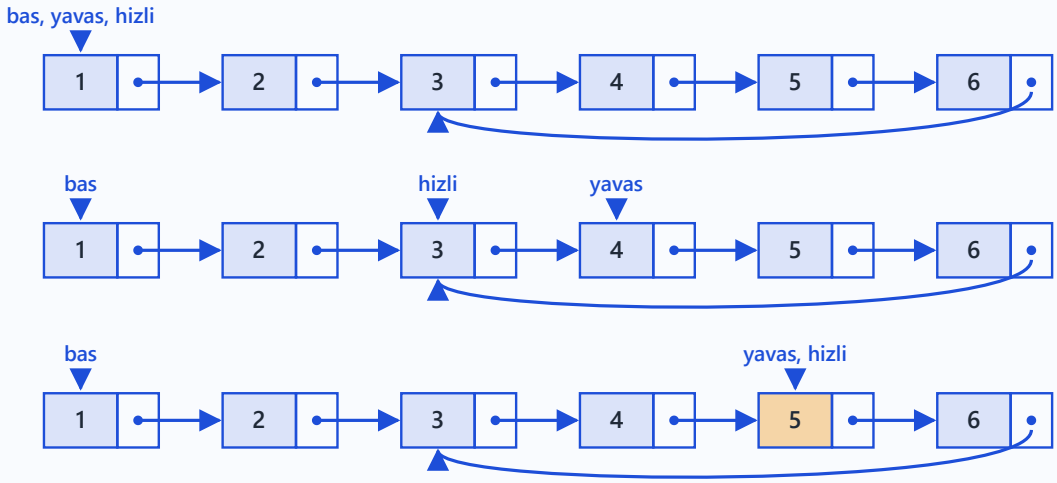
1. **1. tur.** `yavas` 2'ye, `hizli` 3'e gelir. 2 ≠ 3.

2. **2. tur.** `yavas` 3'e, `hizli` 5'e gelir. 3 ≠ 5.
3. **3. tur.** `yavas` 4'e gelir. `hizli` 5'ten 6'ya, oradan döngü bağıyla 3'e geçer. 4 ≠ 3. Hızlı işaretçi döngüyü dolaşmış, yavaşın bir düğüm arkasına düşmüştür.
4. **4. tur.** `yavas` 5'e, `hizli` 3'ten 4'e, oradan 5'e gelir. 5 == 5: metot `true` döndürür.

Teoremdeki değerlerle: $\mu = 2$ (1 ve 2), $\lambda = 4$ (3, 4, 5, 6). Yavaş işaretçi 2. turda döngüye girer; o anda hızlı işaretçi 5'tedir ve yavaşın (3) iki düğüm arkasındadır. İki tur sonra, toplam 4. turda buluşurlar.

Tur	`yavas`	`hizli`	`yavas == hizli`
başlangıç	1	1	(sınanmaz)
1	2	3	hayır
2	3	5	hayır
3	4	3	hayır
4	5	5	evet → `true`

Tablo 5.20: `donguVarMi` izlemesi (1 → … → 6, 6 → 3).



Şekil 5.40: Floyd algoritması. (a) İki işaretçi baştan başlar; 6'dan 3'e dönen bağ döngüyü oluşturur. (b) 3. turun sonunda hızlı işaretçi döngüyü dolaşıp 3'e gelmiştir; yavaş işaretçi 4'tedir. (c) 4. turda ikisi 5'te buluşur.

Döngü yoksa en çok $n/2$, varsa en çok n tur yapılır: zaman $O(n)$'dir. Yalnız iki işaretçi kullanılır: ek bellek $O(1)$'dir.

Sık yapılan hata

Karşılaştırmayı döngünün başına, işaretçiler ilerlemeden önce koymak. İki işaretçi de `bas`'tan başladığı için ilk sınamada eşittirler ve metot döngü olmayan listeler için de hemen `true` döndürür. Eşitlik, işaretçiler ilerledikten *sonra* sınanır.

5.5.4 Sona ekleme

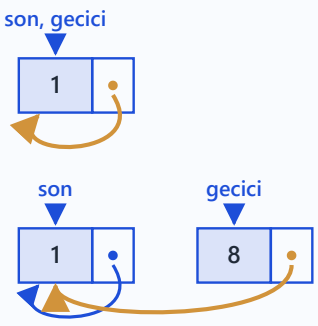
Dairesel listede sona ekleme, yeni düğümü son düğüm ile baş düğümün arasına koymaktır:

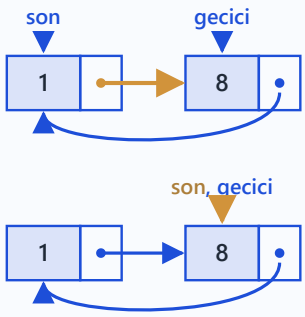
```
Dugum gecici = new Dugum(deger);
if (son == null) {
    son = gecici;
    son.sonraki = son;
} else {
    gecici.sonraki = son.sonraki;
    son.sonraki = gecici;
    son = gecici;
}
boyut++;
```

Liste boşsa yeni düğüm son olur ve kendini gösterir: `son.sonraki = son`. Halka tek düğümle hemen kurulmuş olur. Liste boş değilse üç atama yapılır. Önce yeni düğüm başı gösterir: `gecici.sonraki = son.sonraki`. Sonra eski son yeni düğümü gösterir: `son.sonraki = gecici`. Son olarak `son` yeni düğüme taşınır. Her iki durumda da `boyut` bir artar.

Çağrı	Atamalar	`son`	Halka	`boyut`
`sonaEkle(1)`	`son = gecici` ; `son.sonraki = son`	düğüm(1)	1 → (1)	1
`sonaEkle(8)`	8 → 1; 1 → 8; `son` = 8	düğüm(8)	1 → 8 → (1)	2
`sonaEkle(10)`	10 → 1; 8 → 10; `son` = 10	düğüm(10)	1 → 8 → 10 → (1)	3

Tablo 5.21: Boş dairesel listeye `sonaEkle(1)`, `sonaEkle(8)`, `sonaEkle(10)` izlemesi.





Şekil 5.41: Dairesel listeye ekleme. (a) `sonaEkle(1)` : tek düğüm kendini gösterir. (b) `sonaEkle(8)` , ilk atama: yeni düğüm başı (1) gösterir. (c) İkinci atama: eski son (1) yeni düğümü gösterir. (d) `son` 8'e taşınır.

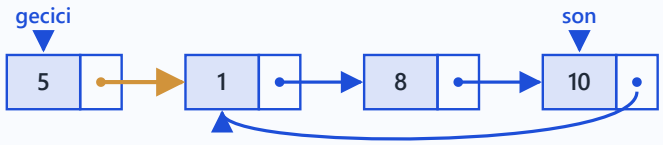
Her çağrı en çok üç atama yapar ve hiçbir düğüm gezilmez: *O(1)*. Atamaların sırası burada da önemlidir. `son.sonraki = gecici` önce yapılırsa başa giden tek bağ silinir. Ardından `gecici.sonraki = son.sonraki` yeni düğümü kendisine bağlar ve eski düğümler halkadan kopar.

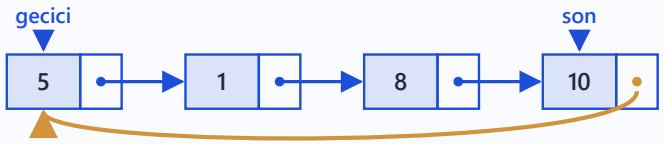
5.5.5 Başa ekleme ve baştan silme

Dairesel listede başa ekleme ile sona ekleme neredeyse aynıdır. Yeni düğüm her iki durumda da halkada aynı yere, son düğüm ile baş düğümün arasına girer. Fark yalnız, işlemden sonra kimin "son" sayıldığıdır:

```
void basaEkle(int deger) {
    Dugum gecici = new Dugum(deger);
    if (son == null) {
        son = gecici;
        son.sonraki = son;
    } else {
        gecici.sonraki = son.sonraki;
        son.sonraki = gecici;
    }   // son değişmez
    boyut++;
}
```

Kod, sona eklemenin `son = gecici` satırı çıkarılmış hâlidir. `son` yerinde kaldığı için yeni düğüm son düğümden hemen sonra, yani başta durur. 1 → 8 → 10 → (1) halkasına (`son` = 10) `basaEkle(5)` uygulanınca önce 5, eski başı (1) gösterir. Sonra 10, 5'i gösterir. Halka 5 → 1 → 8 → 10 → (5) olur ve `son` hâlâ 10'dur.



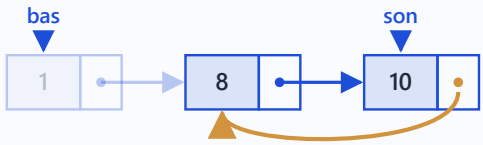


Şekil 5.42: Dairesel listede `basaEkle(5)`. (a) `gecici.sonraki = son.sonraki` : 5, eski başı (1) gösterir. (b) `son.sonraki = gecici` : 10 artık 5'i gösterir; 5 yeni baştır, `son` değişmez.

Baştan silme de sabit sürelidir:

```
void bastanSil() {
    if (son == null) return;
    Dugum bas = son.sonraki;
    if (bas == son)   // tek düğüm
        son = null;
    else
        son.sonraki = bas.sonraki;
    boyut--;
}
```

Baş düğüm `son.sonraki` ile bulunur. Tek düğüm varsa (`bas == son`) liste boşaltılır: `son = null`. Birden çok düğüm varsa son düğümün bağı, baş düğümün sonrakine yöneltilir: `son.sonraki = bas.sonraki`. Böylece baş düğüm halkadan çıkar ve yeni baş, eskisinin sonrakisi olur. 1 → 8 → 10 → (1) halkasında `bastanSil()` sonrası halka 8 → 10 → (8) olur.



Şekil 5.43: Dairesel listede baştan silme. `son.sonraki = bas.sonraki` : 10 artık 8'i gösterir; 1 halkadan çıkmıştır.

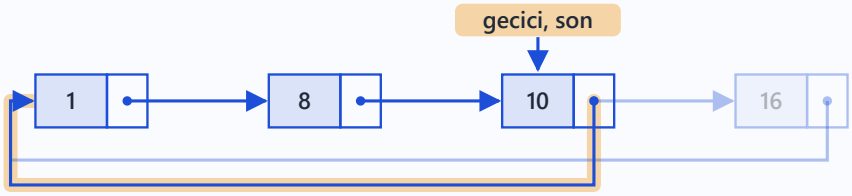
İki işlem de liste uzunluğundan bağımsız birkaç atama yapar: *O(1)*. Metotta yerel değişkenin adı `bas` 'tır; bu ad sınıfta bir alan değildir, yalnız o an baştaki düğümü gösteren geçici bir referanstır.

5.5.6 Sondan silme

Yalnız `son` tutan tek yönlü dairesel listede sondan silme yine *O(n)*'dir. Son düğümü halkadan çıkarmak için ondan önceki düğümün bağı değiştirilmelidir ve o düğüme ulaşmak için halkada gezinmek gerekir:

```
void sondanSil() {
    if (son == null) return;
    if (son.sonraki == son) son = null;   // tek düğüm
    else {
        Dugum gecici = son.sonraki;
        while (gecici.sonraki != son)     // sondan bir önceki
            gecici = gecici.sonraki;
        gecici.sonraki = son.sonraki;     // başa bağla
        son = gecici;
    }
    boyut--;
}
```

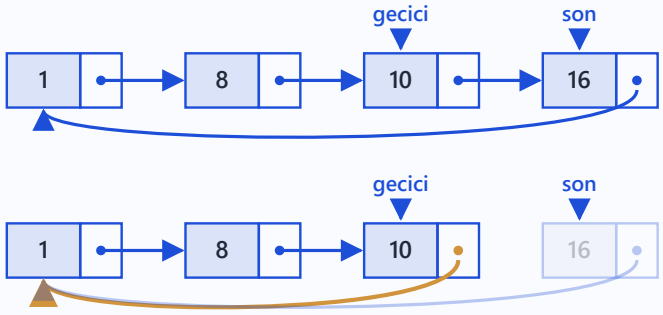
Tek düğüm varsa (`son.sonraki == son`) liste boşaltılır. Aksi hâlde `gecici` baştan başlar ve sonraki-si `son` olana kadar ilerler. Döngüden sonra `gecici` sondan bir önceki düğümdedir. `gecici.sonraki = son.sonraki` ile `gecici` baş düğüme bağlanır; böylece eski son halkadan çıkar. Son olarak `son` `gecici` 'ye taşınır.

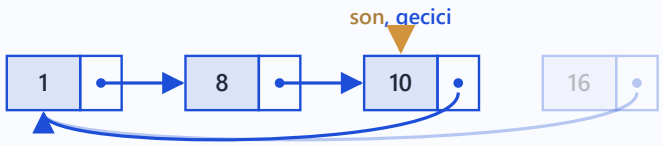


Şekil 5.44: 1 → 8 → 10 → 16 halkasında sondan silme. `gecici` ve yeni `son` 10'dadır; 10'un bağı başa (1) yö-neltilir (vurgulu bağ) ve 16 halkadan çıkar (soluk).

Adım	`gecici`	`gecici.sonraki == son`	Karar
başlangıç	düğüm(1)	hayır (8)	ilerle
1. ilerleme	düğüm(8)	hayır (10)	ilerle
2. ilerleme	düğüm(10)	evet (16)	dur
bağlama	10.sonraki = 1; `son` = 10; `boyut` 3: halka 1 → 8 → 10 → (1)		

Tablo 5.22: 1 → 8 → 10 → 16 → (1) halkasında `sondanSil()` izlemesi (`son` = düğüm 16).





Şekil 5.45: `sondanSil()` 'in son adımları. (a) 10'un sonrakisi `son` 'dur: `gecici` sondan bir önceki düğümdedir. (b) `gecici.sonraki = son.sonraki` : 10 başı gösterir, 16 halkadan çıkar. (c) `son = gecici` : son artık 10'dur.

Döngü $n - 2$ kez döner: $O(n)$. Çift yönlü dairesel listede ise sondan bir önceki düğüm `son.onceki` ile tek adımda bulunur ve sondan silme $O(1)$ olur. Java'nın `LinkedList` sınıfı gibi pek çok gerçekleme bu yüzden çift yönlü bağ kullanır.

5.5.7 Döndürme ve sırayla hizmet

Dairesel listenin en zarif işlemi döndürmedir. Baştaki öğeyi sona almak için hiçbir bağ değiştirilmez; yalnız `son` bir düğüm ilerletilir:

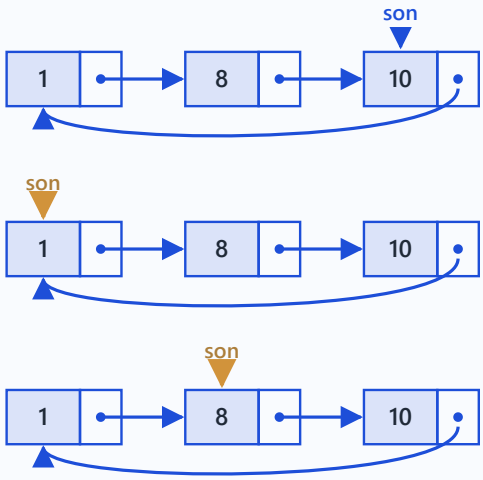
```
void dondur() {              // Goodrich: rotate()
    if (son != null)
        son = son.sonraki;   // eski baş sona geçer
}
// round-robin: her tur baştakine hizmet et
for (int tur = 1; tur <= 4; tur++) {
    System.out.print(son.sonraki.veri + " ");
    liste.dondur();
}
```

`son = son.sonraki` atamasından sonra eski baş düğüm "son" sayılır. Yeni baş ise `son.sonraki`, yani eski başın sonrakisidir. Halkanın kendisi aynı kalır; değişen, halkaya nereden bakıldığıdır. Goodrich–Tamassia bu işleme `rotate()` adını verir. İşlem tek atamadır: $O(1)$.

Döndürme, **round robin** (sırayla dönen çizelgeleme) düzeninin doğrudan modelidir. İşlemciyi paylaşan işler bir halkada durur. Her turda baştaki işe kısa bir süre verilir, ardından liste döndürülür ve sıra bir sonraki işe geçer. Kodun ikinci kısmı bunu dört tur için yapar.

Tur	Hizmet alan (`son.sonraki`)	`dondur()` sonrası `son`	Yeni baş
1	1	1	8
2	8	8	10
3	10	10	1
4	1	1	8

Tablo 5.23: 1 → 8 → 10 → (1) halkasında (başta `son` = 10) dört tur sırayla hizmet.



Şekil 5.46: Döndürme. (a) Başlangıç: `son` = 10, baş 1. (b) İlk `dondur()` : `son` 1'e ilerler, baş 8 olur. (c) İkinci `dondur()` : `son` 8'e ilerler, baş 10 olur. Hiçbir bağ değişmemiştir.

Çıktı `1 8 10 1` 'dir: işler adil biçimde dönüşümlü sıra alır. Aynı düzen tek yönlü düz bir listeyle de kurulabilirdi: baştan silip sona eklemek. Ama bu iki işlem yeni bağlar kurmayı ve son işaretçisini güncellemeyi gerektirir. Dairesel listede aynı sonuç tek bir atamayla elde edilir. Bir iş bitince listeden çıkarmak için `bastanSil()` , yeni gelen bir işi sıranın sonuna koymak için `sonaEkle` kullanılır; ikisi de *O(1)*'dir.

Sık yapılan hata

Döndürmeyi düğümleri yerinden alıp sona ekleyerek yapmak. Bu yol sonuç olarak doğrudur ama gereksiz yere bağ değiştirir, yeni düğüm oluşturur ya da `son` 'u yanlış güncelleme riskini taşır. Dairesel listede başlangıç noktası yalnız `son` 'un nereyi gösterdiğiyle belirlenir; döndürme için bu referansı ilerletmek yeter.

5.6 Dizi mi, bağlı liste mi?

Dizi ile bağlı liste aynı soruya, yani bir öğe dizisini saklama sorusuna iki farklı cevap verir. Hangisinin daha iyi olduğu, programın bu öğelerle ne yapacağına bağlıdır. Şekil 5.47 iki yapının bellekteki görünüşünü yan yana koyar.



Şekil 5.47: Solda dizi: dört öğe bellekte ardışık hücrelerde, 1000, 1004, 1008 ve 1012 adreslerindedir. Sağda

bağlı liste: aynı dört öğe bellekte dağınık düğümlerdedir ve sıra oklarla kurulur.

Dizide *i*. öğenin adresi, başlangıç adresine *i* kere öğe boyu eklenerek hesaplanır. Şekildeki tam sayı dizisinde öğe boyu 4 bayttır; indisi 2 olan öğe $1000 + 2 \cdot 4 = 1008$ adresindedir. Bu hesap tek adımdır ve dizinin uzunluğuna bağlı değildir. Bağlı listede böyle bir hesap yoktur; *i*. öğeye bağları izleyerek gidilir.

Ölçüt	Dizi	Bağlı liste
***i*. öğeye erişim**	O(1): adres hesaplanır	O(n): baştan gezinme
Başa ekleme / silme	O(n): kaydırma	O(1)
Bilinen konumdan sonra ekleme	O(n)	O(1): iki bağ
Boyut	sabit; büyütmek O(n) kopya	dinamik, her ekleme bir `new`
Ek bellek ve önbellek	boş kapasite; önbellek dostu	düğüm başına referans; dağınık erişim

Tablo 5.24: Dizi ile tek yönlü bağlı listenin karşılaştırılması.

Tablo 5.24'ün son satırı, büyük-O gösteriminin yakalamadığı bir farkı gösterir. İşlemci, belleği tek tek bayt olarak değil, ardışık bloklar hâlinde okur ve bu blokları **önbellek** (*cache*) denen hızlı bir ara bellekte tutar. Dizinin öğeleri yan yana durduğu için bir blok okunduğunda sonraki birkaç öğe de önbelleğe gelmiş olur. Bağlı listenin düğümleri bellekte dağınık olabilir; her yeni düğüm ayrı bir okuma gerektirebilir. Bu yüzden ikisi de *O(n)* olan bir gezinme, uygulamada dizide çoğu zaman belirgin biçimde daha hızlıdır.

Bellek kullanımında iki yapının da zayıf yanı vardır. Dizi kapasitesinin bir kısmı boş kalabilir; iki katına büyüyen bir dinamik dizide bu boşluk öğe sayısı kadar olabilir. Bağlı liste ise hiç boş yer tutmaz, ama her öğe için bir referans ve bir nesne başlığı harcar. Tam sayı gibi küçük öğelerde bu ek yük, verinin kendisinden büyüktür.

Seçim için kaba bir kural şöyle söylenebilir:

- **Dizi**: Sık rastgele erişim gerekiyorsa, öğe sayısı aşağı yukarı biliniyorsa ve öğeler çoğunlukla sona ekleniyorsa.
- **Bağlı liste**: Ekleme ve silmeler uçlarda ya da önceden bilinen konumlarda yoğunlaşıyorsa, öğe sayısı öngörülemiyorsa ve indisle erişim nadirse. Yığın ve kuyruk gibi yalnız uçlarda çalışan yapılar bağlı listeyle doğal biçimde gerçeklenir.

Farklı bağlı liste türleri arasındaki seçim de işlemlere bağlıdır. Tablo 5.25 dört yaygın biçimin maliyetlerini bir arada verir.

İşlem	Tek yönlü (yalnız `bas`)	Tek yönlü (`bas` + `son`)	Çift yönlü (`bas` + `son`)	Dairesel (yalnız `son`)
Başa ekleme	O(1)	O(1)	O(1)	O(1)
Sona ekleme	O(n)	O(1)	O(1)	O(1)
Baştan silme	O(1)	O(1)	O(1)	O(1)

İşlem	Tek yönlü (yalnız `bas`)	Tek yönlü (`bas` + `son`)	Çift yönlü (`bas` + `son`)	Dairesel (yalnız `son`)
Sondan silme	O(n)	O(n)	O(1)	O(n)
Arama / *i*. öğe	O(n)	O(n)	O(n)	O(n)
Bilinen düğümü silme	O(n)*	O(n)*	O(1)	O(n)*

Tablo 5.25: Bağlı liste türlerinde işlem maliyetleri.

Yıldızlı hücrelerde maliyetin kaynağı, silinecek düğümden önceki düğümü bulmak için yapılan gezinmedir. Bilinen bir düğümden *sonra* ekleme ise dört türde de *O(1)*'dir, çünkü değişecek bağ o düğümün kendi bağıdır. Tablodan çıkan sonuç şudur: tek yönlü yapılarda sondaki ya da ortadaki bir düğümü silmek pahalıdır; bu işlem sık yapılacaksa çift yönlü liste seçilir.

5.7 Sık yapılan hatalar

Bağlı liste kodundaki hataların çoğu birkaç tipe ayrılır. Not boyunca her işlemin yanında verilen hatalar burada bir arada toplanmıştır; bir bağlı liste kodunu gözden geçirirken denetim listesi olarak kullanılabilir.

Hata	Ne olur?	Doğrusu
Boş liste denetimi yok	`bas` `null` iken `bas.veri` ya da `bas.sonraki`: `NullPointerException`.	`if (bas == null) return;`
Bağları yanlış sırada kurmak	Önce `bas` ya da öncekinin bağı değişirse listenin kalanı kaybolur; düğüm kendini gösterebilir.	Önce yeni düğüm kalan listeye bağlanır: `yeni.sonraki = bas; bas = yeni;`
Gezinirken `bas`'ı ilerletmek	Yazdırmadan sonra listeye erişilemez.	`Dugum gecici = bas;` ile gezinilir.
`son`'u unutmak	Tek düğüm silinince `son` eski düğümü göstermeye devam eder; boş listeye eklemede `son` `null` kalır.	`if (bas == null) son = null;`; boş listeye eklemede `bas = son = yeni;`
Çift yönlüde tek yönü güncellemek	`onceki` güncellenmezse geri gezinme bozulur; ileri yazdırma hatayı gizler.	Her eklemede dört, her silmede iki bağ: `sonraki.onceki = yeni;`
Dairesel listede `!= null`	Halkada `null` yoktur: sonsuz döngü.	`while (g != son) …`
`boyut`'u bazı metotlarda güncellememek	`boyut()` ve `bosMu()` yanlış sonuç verir, sayaç eksiye düşebilir.	Listeyi değiştiren her metot sayacı günceller.

Tablo 5.26: Bağlı liste kodunda sık görülen hatalar.

Bu hataların hepsinin arkasında aynı iki soru vardır. Birincisi: **“Bu atamadan sonra hangi düğüme artık ulaşılamıyor?”** Bir bağın üzerine yazılan her atama, eski hedefe giden bir yolu siler. O hedefe başka yoldan ulaşılmıyorsa ve hâlâ gerekliyse, atamadan önce bir yerel değişkende saklanmalıdır. Ters çevirmedeki `sonraki` değişkeni ve baştan silmedeki `gecici` bu işi görür. İkincisi: **“Bu kod boş listede, tek düğümlü listede, ilk ve son düğümde de doğru mu?”** Bağlı liste hatalarının büyük ço-

ğunluğu bu dört uç durumda ortaya çıkar. Bir metodu yazdıktan sonra onu bu dört durum için kâğıt üzerinde izlemek hataların çoğunu yakalar. Bekçi düğümler, bu uç durumları yapının kendisiyle ortadan kaldırdığı için değerlidir.

5.8 Özet

- **Bağlı liste**, öğeleri **düğümlerde** tutan ve düğümleri referanslarla bağlayan doğrusal bir yapıdır. Sırayı bellekteki konum değil, bağlar belirler. Düğüm sınıfı kendi türünden bir alan içerir (öz-referanslı yapı).
- **Tek yönlü listede** her düğüm sonrakini gösterir; son düğümün bağı `null` 'dır. Liste `bas` ile tutulur. Başa ekleme ve baştan silme *O(1)*; arama, *i*. öğeye erişim, sona ekleme (son işaretçisi yoksa) ve sondan silme *O(n)*'dir.
- **Gezinme deseni**: `gecici = bas` ; `gecici != null` olduğu sürece işi yap; `gecici = gecici.sonraki` . Yazdırma, uzunluk ve arama bu desenin değişkeleridir. Son düğümde durmak gerekiyorsa koşul `gecici.sonraki != null` 'dır.
- Ekleme ve silmede **bağların sırası** kritiktir: önce yeni düğüm kalan listeye bağlanır, sonra önceki-nin bağı ya da `bas` değiştirilir. Tek yönlü listede ekleme ve silme için işlem yapılacak düğümün *öncesine* yürünür.
- **Son işaretçisi** sona eklemeyi *O(1)* yapar, ama sondan silmeyi hızlandırmaz. Her işlem `son` 'u ve varsa `boyut` 'u güncel tutmak zorundadır.
- **Ters çevirme** üç işaretçiyle yeni düğüm oluşturmadan *O(n)* zaman ve *O(1)* ek bellekle yapılır. **Yavaş ve hızlı işaretçi**, orta öğeyi tek geçişte bulur; çift uzunlukta ikinci orta döner.
- **Derin kopya** yeni düğümler oluşturur; `kopya.bas = bas` yalnız ikinci bir referanstır. Tekrarları kaldırma görülen değerler listesiyle $O(n^2)$, hash kümesiyle *O(n)*'dir. İki sıralı listenin birleştirilmesi son işaretçisiyle *O(n + m)*'dir. Değer takaslı sıralama $n(n - 1)/2$ karşılaştırma yapar; bağlı liste için daha iyisi *O(n log n)*'lik birleştirme sıralamasıdır.
- **Çift yönlü listede** her düğüm önceki ve sonraki düğümü gösterir. Başa ve sona ekleme, iki uçtan silme ve bilinen bir düğümü silme *O(1)*'dir; liste geri yönde de gezilir. Bedeli, düğüm başına ikinci bir referans ve her işlemde iki yönün güncellenmesidir. Aradan eklemede dört bağ değişir ve sıra önemlidir.
- **Bekçi düğümler** (başlık ve bitiş), her gerçek düğümün bir öncesi ve sonrası olmasını sağlar; `null` denetimleri ve `bas` / `son` güncellemeleri ortadan kalkar.
- **Matristen bağlı yapı**, her düğümün `sag` ve `asagi` bağı tuttuğu çok yönlü bir yapıdır. Sağ alttan başlayarak ve düğümleri bir tabloda saklayarak *O(satır · sütun)*'da kurulur.
- **Dairesel listede** son düğüm başı gösterir; yalnız `son` tutulur ve baş `son.sonraki` 'dir. Sona ekleme, başa ekleme, baştan silme ve **döndürme** *O(1)*; sondan silme *O(n)*'dir. Gezinme `null` 'a göre değil, `son` 'a göre durdurulur. Döndürme, sırayla dönen çizelgelemenin (round robin) doğrudan modelidir.

- **Floyd algoritması** yavaş ve hızlı işaretçiyle bir listede döngü olup olmadığını *O(n)* zaman ve *O(1)* ek bellekle bulur.
- **Dizi** rastgele erişimde ve önbellek kullanımında, **bağlı liste** uçlarda ekleme ve silmede ve öngörülemeyen boyutta üstündür.

5.9 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

bağlı liste *linked list* · düğüm *node* · baş *head* · son *tail* · işaretçi *pointer* · referans *reference* · null *null* · tek yönlü bağlı liste *singly linked list* · çift yönlü bağlı liste *doubly linked list* · dairesel bağlı liste *circularly linked list* · gezinme *traversal* · bekçi *sentinel* · döngü tespiti *cycle detection* · soyut veri tipi *abstract data type (ADT)* · jenerik *generic* · derin kopya *deep copy* · sığ kopya *shallow copy* · takas *swap* · birleştirme sıralaması *merge sort* · hash tablosu *hash table* · matris *matrix* · round robin *round robin* · zaman karmaşıklığı *time complexity* · alan karmaşıklığı *space complexity*

5.10 Kendini Dene

1. Boş bir `TekYonluBagliListe`'ye (`bas` ve `son` tutan) sırasıyla `sonaEkle(4)`, `basaEkle(2)`, `sonaEkle(9)`, `konumaEkle(7, 1)` ve `ilkOgeyiSil()` uygulanıyor. Son çağrı hangi değeri taşıyan düğümü döndürür ve sonda liste nedir?

   Cevap

   Liste sırasıyla şu durumlardan geçer: 4; 2 → 4; 2 → 4 → 9; 2 → 7 → 4 → 9. `konumaEkle(7, 1)`'de `gecici` hiç ilerlemez (döngü *indis − 1 = 0* kez döner) ve 7, baştaki 2'nin arkasına girer. `ilkOgeyiSil()` 2'nin düğümünü döndürür; liste 7 → 4 → 9 olur, `son` hâlâ 9'dur.

2. 6 → 3 → 6 → 1 listesinde `ogeyiAra(6)` ve `ogeyiAra(5)` ne döndürür? Her biri kaç düğümün verisini karşılaştırır?

   Cevap

   `ogeyiAra(6)` ilk düğümde eşleşir ve 0 döndürür; tek karşılaştırma yapılır, ikinci 6'ya hiç bakılmaz. `ogeyiAra(5)` dört düğümün dördüne de bakar, `gecici` `null` olunca −1 döndürür. Bu, aramanın en iyi (*O(1)*) ve en kötü (*O(n)*) durumudur.

3. 5 → 2 → 7 listesinde `listeyiTersCevir()` çalışıyor. İkinci turun sonunda `onceki`, `suanki` ve

ters çevrilmiş kısım nedir? Sonuç liste ve `son` nedir?

Cevap

1. turda 5'in bağı `null`'a çevrilir; `onceki` 5, `suanki` 2 olur. 2. turda 2'nin bağı 5'e çevrilir; `onceki` 2, `suanki` 7 olur. Ters kısım 2 → 5 → null'dır; 7'nin bağı henüz `null`'dır. 3. turdan sonra `bas = onceki` ile liste 7 → 2 → 5 olur. `son`, metodun ilk satırında eski baş olan 5'e yöneltilmiştir.

4. `ortaOgeyiBul()` 3 → 1 → 4 → 1 → 5 ve 9 → 8 → 7 → 6 listelerinde ne döndürür? İkincisinde ilk ortayı elde etmek için ne değişmelidir?

Cevap

İlk listede turlar (yavaş, hızlı): (1, 4), (4, 5); `hizli.sonraki` `null` olunca döngü biter ve 4 döner. İkinci listede turlar (8, 7), (7, null); 7, yani ikinci orta döner. İlk orta (8) için döngü koşulu `hizli.sonraki != null && hizli.sonraki.sonraki != null` yapılır; o zaman tek tur atılır ve `yavas` 8'de kalır.

5. 2 → 2 → 5 → 2 → 5 → 9 listesinde `tekrarlayanlariKaldir()` çalışıyor. Sonuç liste nedir, kaç düğüm atlanır, `iceriyor` kaç kez çağrılır ve sonda `gorulen` listesinin içeriği nedir?

Cevap

Sonuç 2 → 5 → 9'dur. İkinci, dördüncü ve beşinci düğümler (2, 2, 5) atlanır: 3 düğüm. `iceriyor` her düğüm için bir kez, toplam 6 kez çağrılır. `gorulen`'e ekleme başa yapıldığı için içeriği 9 → 5 → 2'dir. İkinci düğüm silinirken `onceki` ilk düğümde kalır, bu yüzden dördüncü düğümdeki 2 de doğru biçimde 5'ten sonra atlanır.

6. 1 → 5 → 6 ile 2 → 3 → 7 → 8 listeleri `birlestir` ile birleştiriliyor. Ana döngüde kaç karşılaştırma yapılır ve hangi öğeler karşılaştırmasız eklenir?

Cevap

Karşılaştırmalar: 1 ≤ 2 (1 eklenir), 5 ≤ 2 yanlış (2), 5 ≤ 3 yanlış (3), 5 ≤ 7 (5), 6 ≤ 7 (6). Birinci liste bitince ana döngü durur: 5 karşılaştırma. Kalan 7 ve 8 karşılaştırmasız eklenir. Sonuç 1 → 2 → 3 → 5 → 6 → 7 → 8'dir.

7. Çift yönlü 3 ⇄ 6 ⇄ 9 listesine sırasıyla `ogeyiSil(3)`, `sonOgeyiSil()`, `basaEkle(1)` uygulanıyor.

Her çağrıda `ogeyiSil` ve `sonOgeyiSil`'in hangi dalı çalışır? Sonda ileri ve geri okuma ne verir?

Cevap

`ogeyiSil(3)`: 3 baştadır, `gecici.onceki` `null`'dır; `else bas = gecici.sonraki` dalı çalışır. Sağ komşu (6) vardır; 6.onceki = null olur. Liste 6 ⇄ 9. `sonOgeyiSil()`: `bas != son`, genel dal; `son` 6'ya geçer, 6.sonraki = null. `basaEkle(1)`: liste boş değildir; 1 ⇄ 6. İleri okuma 1 6, geri okuma 6 1'dir.

8. 1 → 2 → 3 → 4 → 5 → 6 → 7 listesinde 7 yeniden 5'i gösteriyor. `donguVarMi` kaçıncı turda, hangi düğümde `true` döndürür? Döngüsüz 1 → 2 → 3 → 4 → 5 listesinde kaç tur sonra `false` döner?

Cevap

Turlara göre (yavaş, hızlı): (2, 3), (3, 5), (4, 7), (5, 6), (6, 5), (7, 7). 6. turda 7'de buluşurlar. Burada $\mu = 4, \lambda = 3$; yavaş 4. turda döngüye girdiğinde hızlı 6'dadır ve yavaşın iki düğüm arkasındadır, iki tur sonra buluşurlar. Döngüsüz listede hızlı işaretçi 1 → 3 → 5 ilerler; 2. turdan sonra `hizli.sonraki` `null`'dır ve metot `false` döndürür.

9. Dairesel 4 → 7 → 2 → (4) listesinde (`son` = 2) `sondanSil()` ve ardından `basaEkle(9)` çağrılıyor. `sondanSil`'deki döngü kaç kez ilerler? Sonda `yazdir()` ne yazar ve `son` hangi düğümdür?

Cevap

`gecici` 4'ten başlar; 4'ün sonrakisi 7, `son` değildir, bir kez ilerler. 7'nin sonrakisi `son`'dur, döngü durur. 7, 4'e bağlanır ve `son` 7 olur: 4 → 7 → (4). `basaEkle(9)`: 9, 4'ü gösterir; 7, 9'u gösterir; `son` değişmez. `yazdir()` `9 4 7` yazar; `son` 7'dir.

10. 3×3 bir matris `olustur` ile bağlı yapıya çevriliyor. Kaç düğüm oluşur, kaç `sag` ve kaç `asagi` bağı `null` değildir? `d` tablosu kullanmayan, `olustur(i, j+1)` ve `olustur(i+1, j)`'yi doğrudan çağıran özyinelemeli sürüm kaç düğüm oluşturur?

Cevap

`olustur` 9 düğüm oluşturur; $3 \cdot 2 = 6$ `sag` ve $2 \cdot 3 = 6$ `asagi` bağı doludur. Özyinelemeli sürümde bir hücre, sol üstten ona giden sağ-aşağı yol sayısı kadar oluşturulur. Satır satır bu sayılar 1, 1, 1 / 1, 2, 3 / 1, 3, 6'dır; toplam 19 düğüm oluşur.

5.11 İleri okuma

- M. T. Goodrich, R. Tamassia ve M. H. Goldwasser, *Data Structures and Algorithms in Java* (6. baskı, 2014), Bölüm 3: tek yönlü, dairesel ve bekçili çift yönlü listeleri bu nottaki ADT'ye çok yakın bir Java koduyla anlatır; `rotate()` ve listenin kopyalanması da orada işlenir.

- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 1.3: bağlı listeyi yığın, kuyruk ve torba gerçeklemelerinin temeli olarak kullanır; kısa ve düzenli kod örnekleri verir.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (3. baskı, 2009), Kesim 10.2: çift yönlü listeyi ve tek bir bekçiyle kurulan dairesel listeyi sözde kodla, daha biçimsel bir dille anlatır.
- D. E. Knuth, *The Art of Computer Programming*, Cilt 2 (3. baskı, 1997), Kesim 3.1, Alıştırma 6: yavaş ve hızlı işaretçiyle döngü bulma yöntemini Floyd'a atfeden ilk yazılı kaynaklardandır.

5.12 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3. baskı, MIT Press, 2009.
2. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6. baskı, Wiley, 2014.
3. D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3. baskı, Addison-Wesley, 1997.
4. R. Lafore, *Data Structures and Algorithms in Java*, 2. baskı, Sams, 2002.
5. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011.

KISIM II · BÖLÜM 6

Liste ADT ve Java Koleksiyonları

Liste, programlamada en sık başvurulan veri yapısıdır. Öğrenci adları, sepetteki ürünler, bir oyundaki hamleler hep bir sıra içinde tutulur ve bu sıraya konumla erişilir. Önceki bölümlerde dizi ve bağlı liste ayrı ayrı ele alındı. Bu bölüm, ikisinin üstünde duran soyut "liste" kavramını ve Java'nın bu kavramı nasıl düzenlediğini anlatır. Not, Java koleksiyon arayüzleriyle başlar: `Iterable`, `Collection` ve `List`. Ardından liste ADT'sinin işlemleri, görünümler ve değiştirilemez listeler gelir. Sonra `List` metotları ve listede gezinmenin üç aracı ele alınır: `Iterator`, `ListIterator` ve `Spliterator`. Son kesimler `ArrayList` ile `LinkedList`'in içini açar, amortize maliyeti hesaplar ve konumsal listeyi tanıtır. `Vector`, `Stack`, `CopyOnWriteArrayList` ve `Arrays.asList` ile bölüm kapanır.

Bu bölümün kodu: `github.com/sercankulcu/data-structures-java/Bolum06/src`

ÖĞRENME ÇIKTILARI

Liste ADT'si
Sıralı koleksiyonların temel işlemlerini (ekleme, araya ekleme, silme, indisleme, dilimleme, arama, uzunluk) açıklamak ve her birinin Java `List` karşılığını vermek.

Gezinme
`Iterator`, `ListIterator` ve `Spliterator`'ın ne işe yaradığını ayırt etmek; gezinirken güvenle silmeyi ve `ConcurrentModificationException`'ın nedenini açıklamak.

Gerçeklemeler
`ArrayList` ve `LinkedList`'in iç yapısını ve işlem maliyetlerini karşılaştırmak; büyüyen dizinin amortize maliyetini hesaplamak.

API seçimi
Bir işin gereksinimine göre uygun liste sınıfını ve metodu seçmek; görünüm, kopya ve değiştirilemez liste arasındaki farkı gözetmek.

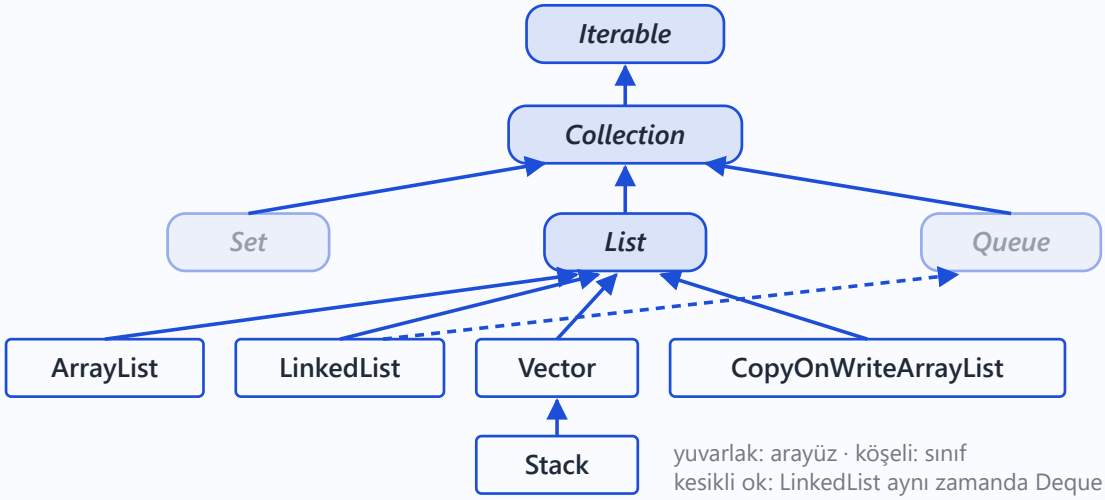
6.1 Koleksiyon arayüzleri

Bir veri yapısını iki düzeyde anlatmak mümkündür. Birinci düzey yapının *ne yaptığını* söyler: hangi işlemler vardır, her işlem neyi değiştirir, ne döndürür. İkinci düzey bunun *nasıl yapıldığını* söyler: veriler bellekte nasıl durur, her işlem hangi adımlarla yürür. Birinci düzeydeki tarife **soyut veri tipi** (*abstract data type*, kısaca ADT) denir; ikinci düzeydeki somut yapıya ise o ADT'nin **gerçeklemesi** denir.

Java bu ayrımı dilin kendisine taşır. Bir ADT'nin işlemleri bir **arayüzde** (*interface*) yazılır. Arayüz yalnız metotların adlarını, parametrelerini ve dönüş türlerini verir; gövde yazmaz. Bu işlemleri gerçekten yapan kod ise arayüzü uygulayan bir **sınıftadır**. Java'nın hazır veri yapıları bu düzen içinde, `java.util` paketindeki **Java Koleksiyon Çerçevesi**nde (*Java Collections Framework*) toplanmıştır. Bu kesim çerçevenin liste tarafındaki arayüzleri yukarıdan aşağıya tanıtır.

6.1.1 Java Koleksiyon Çerçevesi: liste tarafı

Koleksiyon (*collection*), birden çok öğeyi tek bir nesne içinde tutan yapıya verilen genel addır. Liste, küme ve kuyruk birer koleksiyondur. Çerçevedeki arayüzler bir kalıtım ağacı oluşturur. Ağacın tepesinde en az şey isteyen arayüz durur; aşağı inildikçe arayüzler yeni yetenekler ekler. Şekil 6.1 bu ağacın liste tarafını gösterir.



Şekil 6.1: Java koleksiyon hiyerarşisinin liste tarafı. Yuvarlak köşeli kutular arayüz, köşeli kutular sınıftır. `Iterable` en tepededir; `Collection` onu genişletir; `List`, `Set` ve `Queue` da `Collection`'ı genişletir. `ArrayList`, `LinkedList`, `Vector` ve `CopyOnWriteArrayList` `List`'i uygular; `Stack`, `Vector`'dan türer. Kesikli ok, `LinkedList`'in aynı zamanda bir `Deque` (dolayısıyla bir `Queue`) olduğunu gösterir.

Şekildeki ilişkiler okunurken iki ayrı fiil kullanılır. Bir arayüz başka bir arayüzü *genişletir* (`extends`): `List`, `Collection`'ın bütün metotlarını devralır ve yenilerini ekler. Bir sınıf ise bir arayüzü *uygular* (`implements`): `ArrayList`, `List`'in istediği her metodun gövdesini yazar. Kalıtım zincirinden ötürü bir `ArrayList` nesnesi aynı anda bir `List`, bir `Collection` ve bir `Iterable`'dır.

Arayüz ile sınıf arasındaki ayrım, ADT ile veri yapısı arasındaki ayrımın aynısıdır. Arayüz *ne yapılabileceğini*, sınıf *nasıl yapıldığını* belirler. `List` arayüzü "indisle öğe al" der; `ArrayList` bunu bir dizinin hücresini okuyarak, `LinkedList` ise düğümler üzerinde yürüyerek yapar. Kullanıcı yalnız arayüzü biliyorsa, iki gerçeklemeden hangisinin çalıştığını umursamadan aynı kodu yazabilir.

Bu yüzden Java'da yerleşmiş bir alışkanlık vardır: değişken arayüz türüyle tanımlanır, nesne ise somut sınıftan oluşturulur.

```
List<String> l = new ArrayList<>();
```

Bu satırın sol yanı "`l` bir listedir" der; sağ yanı "bu liste şimdilik bir `ArrayList`'tir" der. İleride listenin başına çok sık ekleme yapıldığı anlaşılırsa yalnız sağ yan `new LinkedList<>()` yapılır; `l`'yi kullanan yüzlerce satır olduğu gibi kalır. Değişken `ArrayList<String>` türüyle tanımlansaydı, kodun bir yerinde yalnız `ArrayList`'e özgü bir metot (örneğin `trimToSize()`) kullanılmış olabilirdi ve değişiklik derlenmezdi.

> **Sık yapılan hata**
>
> Arayüzden doğrudan nesne oluşturmaya çalışmak: `List<String> c = new List<>();` derlenmez. Arayüzün metot gövdeleri yoktur; ortada çalıştırılacak kod bulunmaz. `new`'den sonra her zaman arayüzü uygulayan bir sınıf yazılır. Buna karşılık `Iterable<String> d = new ArrayList<>();` derlenir, çünkü `ArrayList` zincirin tepesindeki `Iterable`'ı da uygular.

6.1.2 Iterable arayüzü ve for-each döngüsü

Hiyerarşinin en üstündeki `Iterable` (yinelenebilir) arayüzü en az şeyi ister: nesnenin öğeleri üzerinde sırayla gezinebilmek. Bir nesne `Iterable` ise ondan şu üç şey elde edilir:

- `iterator()`: öğelere sırayla erişen bir `Iterator` nesnesi döndürür. Arayüzün tek soyut metodu budur; bir sınıfın `Iterable` olması için yalnız bunu yazması yeter.
- `spliterator()`: öğeleri parçalara bölerek paralel işlemeyi destekleyen bir `Spliterator` döndürür.
- `forEach(...)`: her öğe için verilen işlemi (bir lambda ifadesini) çalıştırır.

Son ikisinin hazır (varsayılan) gövdesi vardır; ikisi de arka planda `iterator()`'ı ya da benzer bir gezinme aracını kullanır. Bu yüzden `Iterable`'ın asıl sözleşmesi şudur: "Bana sor, sana bir yineleyici vereyim."

Yineleyici (*iterator*), bir koleksiyonun öğelerini birer birer veren ayrı bir nesnedir. Koleksiyonun içinde nerede kaldığını kendisi hatırlar. İki temel sorusu vardır: "Sırada öğe var mı?" (`hasNext()`) ve "Sıradakini ver" (`next()`). Java'nın sevilen for-each döngüsü bu iki metot üzerine kuruludur.

Programcı kısa biçimi yazar; derleyici onu uzun biçime çevirir:

```
// yazdığımız
for (var x : a) {
    System.out.println(x);
}
```

```
// derleyicinin ürettiği (a Iterable ise)
Iterator<T> it = a.iterator();
while (it.hasNext()) {
    var x = it.next();
    System.out.println(x);
}
```

İlk parçadaki for-each döngüsünde yineleyici görünmez. Derleyici `a`'nın türüne bakar. `a` bir `Iterable` ise ikinci parçadaki koda dönüştürür: önce `a.iterator()` çağrılır, sonra `hasNext()` doğru döndükçe `next()` ile sıradaki öğe alınır ve döngü gövdesi o öğeyle çalışır. `a` bir dizi ise yineleyici kullanılmaz; döngü, `i = 0`'dan `a.length - 1`'e giden sıradan bir indisli döngüye çevrilir.

Buradan önemli bir kural çıkar: for-each döngüsü yalnız sağ yanındaki ifade bir dizi ya da `Iterable`'ı uygulayan bir sınıfın nesnesiyse derlenir. Kendi yazdığımız bir sınıfın nesnesini for-each ile gezmek istiyorsak o sınıfın `Iterable`'ı uygulaması gerekir.

6.1.3 Kod izleme: kendi Iterable sınıfımız

Öğeleri bellekte hiç tutulmayan bir koleksiyon düşünelim: `Aralik(2, 5)` nesnesi 2, 3 ve 4 sayılarını temsil etsin, yani *[2, 5)* yarı açık aralığını. Sayılar bir dizide saklanmaz; yineleyici onları gezinme sırasında üretir. Sınıf şöyle yazılır:

```
class Aralik implements Iterable<Integer> {
  private final int bas, son;
  Aralik(int b, int s) { bas = b; son = s; }
  public Iterator<Integer> iterator() {
    return new Iterator<Integer>() {
      int k = bas;
      public boolean hasNext() { return k < son; }
      public Integer next() {
        if (k >= son)
          throw new NoSuchElementException();
        return k++;
      }
    };
  }
}
```

Kodun satırlarını sırayla okuyalım.

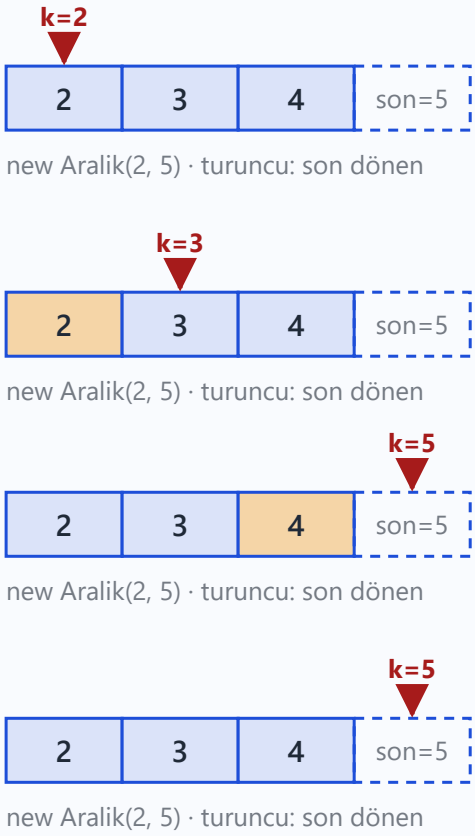
- `class Aralik implements Iterable<Integer>`: sınıf, `Integer` öğeler veren bir `Iterable` olacağını bildirir. Tür parametresi `Integer`'dır, çünkü jenerik tür parametreleri ilkel tür (`int`) olamaz.
- `bas` ve `son` alanları aralığın sınırlarıdır ve `final`'dır; nesne oluşturulduktan sonra değişmez.
- `iterator()` her çağrıldığında *yeni* bir yineleyici döndürür. Yineleyici, `new Iterator<Integer>() { … }` biçiminde yazılmış **isimsiz iç sınıfın** bir nesnesidir. Bu sınıfın kendi alanı `k`'dir ve `bas` ile başlar. İsimsiz iç sınıf, dıştaki `Aralik` nesnesinin `bas` ve `son` alanlarına doğrudan erişebilir.
- `hasNext()`, sıradaki değerin aralıkta olup olmadığını sorar: `k < son`.
- `next()` önce sözleşmeyi korur: öğe kalmamışsa (`k >= son`) `NoSuchElementException` fırlatır. Öğe varsa `return k++` çalışır. Sonek artırma, *önce* `k`'nin o anki değerini döndürür, *sonra* `k`'yi bir artırır.

Şimdi `for (int x : new Aralik(2, 5))` döngüsünün ne yaptığını adım adım izleyelim. Döngü değişkeni `int` olduğu için her `next()`'in döndürdüğü `Integer` otomatik olarak `int`'e çevrilir.

Adım	Çağrı	Denetim	Döner	`k` (sonra)	Gövde
1	`new Aralik(2, 5)`	—	nesne	—	—
2	`iterator()`	—	yineleyici	2	—
3	`hasNext()`	2 < 5	`true`	2	—
4	`next()`	2 >= 5? hayır	2	3	`x = 2` yazılır
5	`hasNext()`	3 < 5	`true`	3	—
6	`next()`	3 >= 5? hayır	3	4	`x = 3` yazılır
7	`hasNext()`	4 < 5	`true`	4	—
8	`next()`	4 >= 5? hayır	4	5	`x = 4` yazılır
9	`hasNext()`	5 < 5	`false`	5	döngü biter

Tablo 6.1: `for (int x : new Aralik(2, 5))` döngüsünün izlemesi. `k`, yineleyicinin alanıdır.

Çıktı 2, 3, 4'tür. Dikkat edilirse 5 hiçbir zaman döndürülmez: aralık sağdan açıktır. Yineleyici, kendi durumunu yalnız `k`'de tutar; `Aralik` nesnesi gezinme boyunca hiç değişmez. Aynı `Aralik` nesnesi üzerinde iki ayrı for-each döngüsü yazılırsa her biri `iterator()`'ı ayrı çağırır ve kendi `k`'siyle 2'den başlar.



Şekil 6.2: `Aralik(2, 5)` üzerinde yineleyicinin dört anı. (a) `iterator()` sonrası `k = 2`. (b) İlk `next()` 2'yi döndürmüş (turuncu), `k = 3` olmuş. (c) Üçüncü `next()` 4'ü döndürmüş, `k = 5`. (d) `hasNext()` `5 < 5` sonucunu yanlış bulur ve döngü biter.

Döngü bittikten sonra aynı yineleyiciyle bir kez daha `next()` çağrılırsa ne olur? `k = 5` ve `son = 5`'tir; `k >= son` denetimi doğru çıkar ve `NoSuchElementException` fırlatılır. Bu, `Iterator` arayüzünün sözleşmesidir: öğe kalmamışken `next()` çağrısı bu istisnayla sonuçlanmalıdır.

Sık yapılan hata

`next()` içindeki `k >= son` denetimini "nasılsa `hasNext()` önce çağrılıyor" diye atlamak. for-each döngüsü gerçekten önce `hasNext()`'i çağırır; ama yineleyiciyi elle kullanan bir kod bunu yapmak zorunda değildir. Denetim olmasaydı `next()` aralık dışındaki 5'i, 6'yı, 7'yi sessizce döndürürdü. Yanlış sonuç veren ama hata vermeyen bir yineleyici, hata veren bir yineleyiciden çok daha zor yakalanır.

6.1.4 Collection arayüzü

`Iterable`'ın bir altında `Collection` arayüzü durur. Listeler, kümeler ve kuyruklar birbirinden çok farklı davranır, ama hepsinin ortak bir çekirdeği vardır: öğe eklenir, öğe çıkarılır, kaç öğe olduğu sorulur, bir öğenin içeride olup olmadığı sorulur. `Collection` bu ortak çekirdeği tek bir arayüzde toplar.

Yetenek	Metotlar	Örnek
Öğe ekleme / çıkarma	`add(e)`, `remove(o)`, `clear()`	`c.add("Ali")` → `true`
Boyut ve boşluk sorgusu	`size()`, `isEmpty()`	`c.size()` → 1
Üyelik testi	`contains(o)`	`c.contains("Ali")` → `true`
Akışla işleme	`stream()`	`c.stream().count()`
Diziye aktarma	`toArray()`	`c.toArray(new String[0])`
Toplu işlemler	`addAll`, `removeAll`, `removeIf`	`c.removeIf(s -> s.isEmpty())`

Tablo 6.2: `Collection` arayüzünün yetenekleri. Örnek sütununda `c`, başta boş bir `Collection<String>`'dir.

Tablodaki metotların bazılarının dönüş değeri dikkat ister. `add(e)` bir `boolean` döndürür: koleksiyon bu ekleme yüzünden değiştiyse `true`. Listede bu her zaman `true`'dur, çünkü liste aynı öğeyi ikinci kez de kabul eder. Kümede ise zaten var olan bir öğeyi eklemek koleksiyonu değiştirmez ve `false` döner. `remove(o)` da benzer biçimde, verilen öğeyi bulup sildiyse `true` döndürür.

`stream()`, koleksiyonun öğelerini bir **akışa** (*stream*) dönüştürür. Akış üzerinde süzme, dönüştürme, sayma gibi işlemler zincir hâlinde yazılır. `toArray(new String[0])` öğeleri doğru türde bir diziye kopyalar; boş dizi argümanı yalnız sonuç dizisinin türünü bildirmek içindir. `removeIf` ise bir koşul (lambda) alır ve koşulu sağlayan bütün öğeleri tek seferde siler.

`Collection`'da olmayan şey de en az olanlar kadar önemlidir: **indis** yoktur. Kümede öğelerin bir sırası olmayabilir; "3. öğe" sorusu anlamsızdır. Bu yüzden `get(i)` gibi konuma dayalı işlemler `Collection`'da değil, bir alt düzeyde, `List` arayüzündedir.

> **Sık yapılan hata**
>
> Nesnenin gerçek türüne bakıp değişkenin türünü unutmak. `Collection<String> c = new ArrayList<>(List.of("x", "y"));` satırından sonra `c.get(0)` derlenmez. Nesne gerçekten bir `ArrayList`'tir ve `get` metodu vardır, ama derleyici yalnız değişkenin türüne (`Collection`) bakar ve orada `get` bulamaz. Değişkeni arayüz türüyle tanımlamak esneklik verir; karşılığında yalnız o arayüzün metotları kullanılabilir.

6.1.5 List arayüzü

`List` arayüzü `Collection`'ı genişletir ve ona sıra kavramını ekler. Java Koleksiyon Çerçevesinin üyesidir ve `java.util` paketindedir. Belgelerde *sıralı koleksiyon* (*ordered collection*) ya da *dizi* (*sequence*) diye anılır. Buradaki "sıralı" sözcüğü küçükten büyüğe dizilmiş anlamına gelmez; öğelerin bir *konum sırası* olduğu anlamına gelir.

Tanım 6.1

Liste (*list*), öğeleri bir konum sırası içinde tutan ve her öğeye 0'dan başlayan bir tam sayı **indisiyle** erişilen soyut veri tipidir. Öğenin hangi konuma ekleneceğine kullanıcı karar verir; öğeler eklendikleri konumda kalır ve yalnız araya ekleme ya da silme sonucunda kayar. Liste aynı öğenin birden çok kez bulunmasına izin verir.

`List` arayüzünün temel özellikleri şöyle sıralanabilir:

- **Sıra korunur.** "Ali", "Veli", "Can" sırasıyla eklenen öğeler gezinmede de bu sırayla gelir.
- **Konumu kullanıcı belirler.** `add(e)` sona ekler; `add(i, e)` verilen konuma ekler.
- **İndisle erişim vardır.** `get(i)`, `set(i, e)`, `remove(i)` gibi metotlar konum üzerinden çalışır. İndisler 0'dan başlar.
- **Tekrar serbesttir.** Aynı öğe birden çok kez eklenebilir. Uygulamaların çoğu `null` öğeye de izin verir.
- **Arama yapılabilir.** `indexOf(o)` öğenin ilk geçtiği konumu, `contains(o)` var olup olmadığını söyler.

Gündelik bir benzetme olarak yapılacaklar listesini düşünelim. Maddeler alt alta yazılır; ilk madde, ikinci madde diye konumla anılır. Yeni bir madde araya sıkıştırılabilir, biten bir madde silinebilir, aynı madde ("su iç") iki kez yazılabilir. Java'daki `List` bu davranışı tam olarak taklit eder.



Şekil 6.3: Yapılacaklar listesi: maddeler bir konum sırasıyla yazılır, araya madde eklenebilir, aynı madde iki kez yazılabilir.

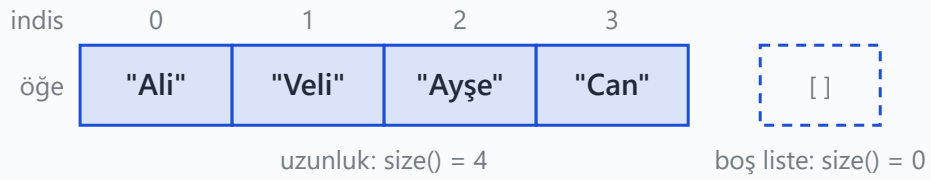
Son bir uyarı: `List` de bir arayüzdür. `new List()` yazılamaz; onu uygulayan bir sınıf seçmek gerekir. Java'nın sunduğu başlıca gerçeklemeler `ArrayList`, `LinkedList`, `Vector`, `Stack` ve `CopyOnWriteArrayList`'tir. Hepsi aynı metotları sunar, ama her biri farklı bir iç yapı kullandığı için aynı işlemin maliyeti sınıftan sınıfa değişir.

6.2 Liste ADT'si

Java'nın sınıflarına girmeden önce listeyi soyut düzeyde ele alalım. Bir liste hangi kavramlarla anlatılır, hangi işlemleri sunar? Bu sorulara verilen cevap hiçbir programlama diline bağlı değildir. Python'daki `list`, C#'taki `List<T>` ve Java'daki `List` aynı ADT'nin farklı dillerdeki yüzleridir.

6.2.1 Temel kavramlar

Liste anlatılırken dört kavram sürekli geçer. Şekil 6.4 dördünü tek bir resimde gösterir.



Şekil 6.4: Dört öğeli bir liste ve boş bir liste. Üst satırdaki 0–3 sayıları indis, kutulardaki adlar öğedir. Soldaki listenin uzunluğu 4, sağdaki boş listenin uzunluğu 0'dır.

Öğe (*element*)

Listede saklanan veridir: bir ad, bir sayı, bir nesne. Java listelerinde öğeler her zaman nesnedir; ilkel değerler sarmalayıcı sınıflara kutulanarak saklanır.

İndis (*index*)

Her öğenin konumunu gösteren tam sayıdır. İlk öğenin indisi 0, son öğeninki *uzunluk* − *1*'dir. Şekil 6.4'te "Ayşe"nin indisi 2'dir.

Boş liste (*empty list*)

Hiç öğe içermeyen listedir. Java'da `isEmpty()` bu durumda `true` döndürür. Boş liste `null` ile aynı şey değildir: boş liste vardır ve metotları çağrılabilir; `null` ise hiçbir nesneyi göstermez.

Uzunluk (*length*)

Listedeki öğe sayısıdır; Java'da `size()` ile okunur. Uzunluk **kapasiteyle** karıştırılmamalıdır. Kapasite, dizi tabanlı bir listenin iç dizisinde ayrılmış hücre sayısıdır ve genellikle uzunluktan büyüktür.

Uzunluk ile kapasite arasındaki farkı bir otobüs benzetmesi iyi anlatır: otobüsün koltuk sayısı kapasite, o anda oturan yolcu sayısı uzunluktur. Kullanıcı yalnız yolcuları görür; boş koltuklar listenin iç işidir.

6.2.2 Liste işlemleri ve Java karşılıkları

Liste ADT'si yedi temel işlem sunar. Tablo 6.3 her işlemi, ne yaptığını ve Java'daki `List` karşılığını yan yana verir.

ADT işlemi	Anlamı	Java `List` metodu
Ekleme (*append*)	sona yeni öğe ekler	`add(e)`
Araya ekleme (*insert*)	verilen konuma ekler, sonrakiler kayar	`add(i, e)`
Silme (*remove*)	konumdaki ya da verilen öğeyi çıkarır	`remove(i)`, `remove(o)`
İndisleme (*indexing*)	konumdaki öğeyi okur ya da değiştirir	`get(i)`, `set(i, e)`
Dilimleme (*slicing*)	bir aralığı seçer: *[başlangıç, bitiş)*	`subList(from, to)`
Arama (*search*)	öğenin konumunu bulur	`indexOf(o)`; yoksa −1
Uzunluk (*length*)	öğe sayısını döndürür	`size()`

Tablo 6.3: Liste ADT işlemleri ve Java `List` metotları.

Araya ekleme ve silmede "sonrakiler kayar" ifadesi listenin özünü anlatır. `[A, B, C]` listesinde 1. konuma `X` eklenirse sonuç `[A, X, B, C]` olur; B'nin indisi 1'den 2'ye, C'ninki 2'den 3'e çıkar. Silmede tersi olur: kalan öğelerin indisleri birer azalır. Listede "boşluk" kalmaz; indisler her zaman 0'dan *uzunluk − 1*'e kesintisiz gider.

Dilimleme işlemindeki *[başlangıç, bitiş)* gösterimi, Java'nın her yerinde kullanılan **yarı açık aralık** kuralıdır. Başlangıç dahildir, bitiş dahil değildir. `subList(1, 3)`, 1 ve 2 indisli öğeleri seçer; seçilen öğe sayısı *3 − 1 = 2*'dir. Bu kural iki kolaylık getirir: dilimin uzunluğu bir çıkarmayla bulunur ve `subList(0, size())` tüm listeyi verir.

`subList`'in bir özelliği öteki işlemlerden ayrılır. Döndürdüğü liste bir **kopya** değil, asıl listenin bir **görünümüdür**.

> **Tanım 6.2**
>
> **Görünüm** (*view*), kendi öğelerini saklamayan, işlemlerini başka bir koleksiyonun öğeleri üzerinde yapan nesnedir. Görünüm üzerinden yapılan değişiklik alttaki koleksiyona yansır; alttaki koleksiyondaki değişiklik de görünümden görülür.

Bu özellik kısa ve güçlü bir deyim sağlar: `l.subList(1, 3).clear()`, asıl listeden 1 ve 2 indisli öğeleri siler. Ayrı bir "aralık sil" metodu gerekmez.

6.2.3 Kod izleme: subList bir görünümdür

Beş öğeli bir `ArrayList` üzerinde bir görünüm açıp onu değiştirelim:

```
List<String> l = new ArrayList<>(
    List.of("A", "B", "C", "D", "E"));
List<String> alt = l.subList(1, 3);
System.out.println(alt);
alt.set(0, "X");
alt.add("Y");
alt.clear();
System.out.println(l);
```

`subList(1, 3)` çağrısı yeni bir dizi ayırmaz ve öğe kopyalamaz. Döndürdüğü nesne yalnız üç bilgi tutar: asıl listeye bir referans, başlangıç kayması (*offset*) ve görünümün uzunluğu. Burada kayma 1, uzunluk *3 − 1 = 2*'dir. Görünüm üzerinde `alt.get(j)` çağrıldığında asıl listede `l.get(1 + j)` okunur. Bütün işlemler bu basit çeviriyle yapılır.

Satır	Asıl liste `l`	Görünüm `alt`	Görünüm uzunluğu	Not
`new ArrayList<>(…)`	[A, B, C, D, E]	—	—	5 öğe
`l.subList(1, 3)`	[A, B, C, D, E]	[B, C]	2	kopya yok; *[1, 3)*
`println(alt)`	[A, B, C, D, E]	[B, C]	2	`l[1]`, `l[2]` okunur; çıktı `[B, C]`
`alt.set(0, "X")`	[A, X, C, D, E]	[X, C]	2	`l.set(1 + 0, "X")`
`alt.add("Y")`	[A, X, C, Y, D, E]	[X, C, Y]	3	`l.add(1 + 2, "Y")`; D, E sağa kayar
`alt.clear()`	[A, D, E]	[ ]	0	`l`'nin 1–3 indisleri silinir
`println(l)`	[A, D, E]	[ ]	0	çıktı `[A, D, E]`

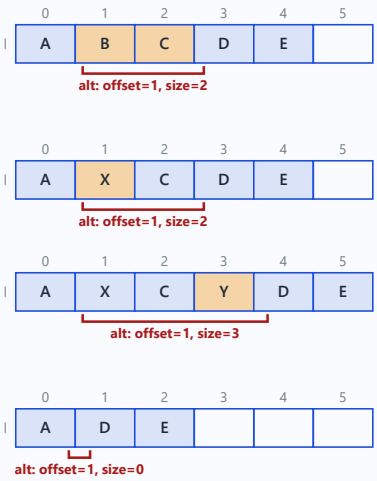
Tablo 6.4: `subList` izlemesi. Görünümün kayması (`offset`) baştan sona 1'dir.

İzlemenin üç anı üzerinde duralım.

Okuma ve yazma. `alt.set(0, "X")`, görünümün 0. öğesini değiştirmek ister. Görünüm bu isteği kaymayı ekleyerek asıl listeye iletir: `l.set(1, "X")`. Asıl listede B'nin yerine X yazılır. Görünüm ayrı bir kopya tutmadığı için onu yeniden yazdırmak `[X, C]` verir.

Ekleme. `alt.add("Y")` görünümün *sonuna* ekler. Görünümün sonu, asıl listede *kayma + uzunluk = 1 + 2 = 3* numaralı konumdur. Böylece çağrı `l.add(3, "Y")`'ye dönüşür: Y, 3. indise girer; D ile E birer sağa kayar. Görünüm de büyür; uzunluğu 3 olur.

Temizleme. `alt.clear()`, görünümdeki üç öğeyi asıl listeden siler: 1, 2 ve 3 indislerindeki X, C ve Y. Arkadaki D ile E üç konum sola kayar. Asıl liste `[A, D, E]` olur, görünüm boşalır.



Şekil 6.5: `subList` izlemesinin dört anı. Alttaki köşeli çizgi görünümün kapsadığı aralıktır. (a) `subList(1, 3)`: kayma 1, uzunluk 2. (b) `alt.set(0, "X")` asıl listenin 1. hücresine yazar. (c) `alt.add("Y")` asıl listenin 3. indisine ekler; görünüm 3 öğeye çıkar. (d) `alt.clear()` sonrası asıl liste `[A, D, E]`, görünüm boş.

Görünümün gücü bir tehlikeyi de beraberinde getirir. Görünüm, açıldığı andaki asıl listeye göre hesap yapar. Asıl liste görünüm dışından *yapısal* olarak değiştirilirse (öğe eklenir ya da silinirse) görünümün kayma ve uzunluk bilgisi artık doğru olmayabilir. Java bu durumu sezer: eski görünüm bir sonraki kullanımda `ConcurrentModificationException` fırlatır. Sezme düzeneği, yineleyicilerin kullandığı `modCount` sayacının aynısıdır (6.4.4).

Örnek 6.1

`l = [A, B, C, D]` olsun. `List<String> g = l.subList(1, 3); g.clear();` çalıştırılıyor. Ardından `l.add("E")` yapılıp `g.size()` çağrılıyor. Sonuç ne olur?

1. **Görünüm.** `g` kayması 1, uzunluğu 2 olan bir görünümdür: `[B, C]`.
2. **Temizleme.** `g.clear()` asıl listenin 1 ve 2 indislerini siler. `l = [A, D]`, `g` boş. Bu silme görünüm üzerinden yapıldığı için görünüm kendi bilgisini günceller; `g` hâlâ geçerlidir.
3. **Dışarıdan ekleme.** `l.add("E")` asıl listeyi doğrudan, görünümü aşarak değiştirir. `l = [A, D, E]`.
4. **Eski görünümü kullanma.** `g.size()` çağrısında görünüm, asıl listenin kendisinden habersiz değiştiğini fark eder ve `ConcurrentModificationException` fırlatır.

Sık yapılan hata

`subList`'in bağımsız bir liste döndürdüğünü sanmak. `List<String> ilkUc = l.subList(0, 3);` yazıp sonra `ilkUc`'u değiştiren kod, farkında olmadan `l`'yi de değiştirir. Bağımsız bir parça gerekiyorsa görünüm kopyalanır: `new ArrayList<>(l.subList(0, 3))`.

6.2.4 Değiştirilebilir ve değiştirilemez listeler

Şimdiye kadarki listeler oluşturulduktan sonra serbestçe değiştirildi. Her liste böyle değildir. Java'da iki tür liste vardır.

Değiştirilebilir (*mutable*) liste
: Oluşturulduktan sonra öğe eklenebilir, silinebilir, değiştirilebilir. Örnekler: `new ArrayList<>()`, `new LinkedList<>()`.

Değiştirilemez (*immutable*) liste
: Oluşturulduktan sonra içeriği hiçbir biçimde değişmez. Değişik bir liste gerekiyorsa yeni bir liste oluşturulur. Örnekler: `List.of(...)`, `List.copyOf(...)`.

Aşağıdaki kod iki türü yan yana koyar:

```
List<String> sabit = List.of("Ali", "Veli");
sabit.add("Can");    // UnsupportedOperationException
List<String> yeni = new ArrayList<>(sabit);   // değiştirilebilir kopya
yeni.add("Can");     // [Ali, Veli, Can] — sabit hâlâ [Ali, Veli]
```

`List.of("Ali", "Veli")` değiştirilemez bir liste üretir. Değişkenin türü yine `List<String>` olduğundan `sabit.add("Can")` satırı *derlenir*; derleyici hiçbir şey fark etmez. Hata çalışma zamanında ortaya çıkar: metot `UnsupportedOperationException` fırlatır. Yasak yalnız ekleme ve silmeyle sınırlı değildir; `sabit.set(0, "X")` da aynı istisnayı verir. Ayrıca `List.of` `null` öğe kabul etmez; `List.of("a", null)` çağrısı `NullPointerException` fırlatır.

Değiştirilebilir bir sürüm gerekiyorsa `new ArrayList<>(sabit)` yazılır. Bu yapıcı, verilen koleksiyonun öğelerini yeni bir diziye kopyalar. Artık ortada iki ayrı nesne vardır: `yeni`'ye "Can" eklemek `sabit`'i etkilemez. Kodun sonunda `yeni` `[Ali, Veli, Can]`, `sabit` ise hâlâ `[Ali, Veli]`'dir.

Değiştirilemez yapıların neden istendiği ilk bakışta açık değildir; değişiklik yapamamak bir kısıtlama gibi görünür. Ama aynı listeyi programın birçok yeri paylaşıyorsa kısıtlama bir güvenceye dönüşür. Liste değişmeyeceğine göre, onu alan hiçbir metot başkalarının verisini bozamaz. Birden çok iş parçacığı aynı listeyi kilitsiz okuyabilir, çünkü okuma sırasında listenin değişme olasılığı yoktur. Bu yüzden değiştirilemezlik, fonksiyonel programlamanın temel düzeneklerinden biridir.

Değiştirilemezlik fikrinin ileri bir biçimi **kalıcı veri yapılarıdır** (*persistent data structures*). Kalıcı bir yapıda her "güncelleme" yeni bir sürüm üretir ve eski sürüm de kullanılabilir kalır. Her güncellemede bütün yapıyı kopyalamak pahalı olurdu. Kalıcı yapılar bunun yerine değişmeyen kısımları eski sürümle paylaşır ve yalnız değişen yolu yeniden oluşturur. Örneğin bir bağlı listenin başına öğe eklemek, eski listeyi hiç değiştirmeden ona işaret eden yeni bir düğüm oluşturmaktır; eski liste ve yeni liste kuyruğu paylaşır.

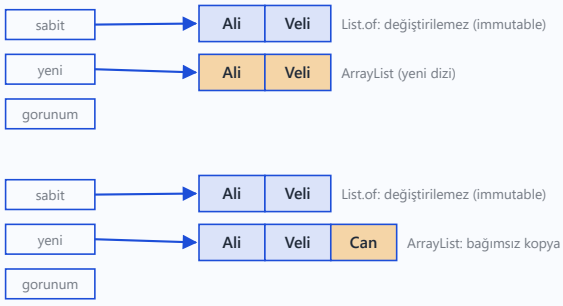
Java'da bir üçüncü seçenek daha vardır ve sık karıştırılır: `Collections.unmodifiableList(l)`. Bu çağrı değiştirilemez bir liste *üretmez*; `l`'yi saran **salt okunur bir görünüm** döndürür. Görünüm üzerinden yazmak yasaktır, ama alttaki `l` değişirse görünüm de değişir.

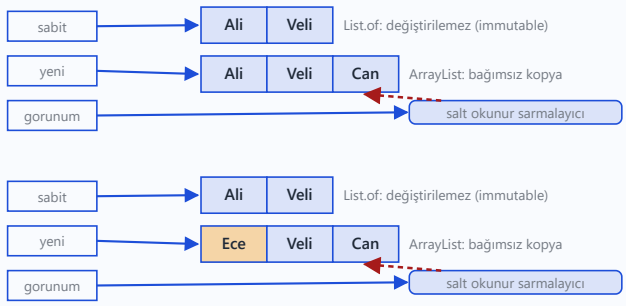
6.2.5 Kod izleme: kopya mı, görünüm mü, değiştirilemez mi?

```
List<String> sabit = List.of("Ali", "Veli");
// sabit.add("Can");      → hata
List<String> yeni = new ArrayList<>(sabit);
yeni.add("Can");
List<String> gorunum =
    Collections.unmodifiableList(yeni);
// gorunum.set(0, "Ece"); → hata
yeni.set(0, "Ece");
System.out.println(gorunum);
```

Satır	`sabit`	`yeni`	`gorunum`	Not
`List.of(...)`	[Ali, Veli]	—	—	değiştirilemez; `null` kabul etmez
`// sabit.add("Can")`	[Ali, Veli]	—	—	açılsaydı: `UnsupportedOperationException`
`new ArrayList<>(sabit)`	[Ali, Veli]	[Ali, Veli]	—	yeni bir diziye kopya
`yeni.add("Can")`	[Ali, Veli]	[Ali, Veli, Can]	—	`sabit` etkilenmez
`unmodifiableList(yeni)`	[Ali, Veli]	[Ali, Veli, Can]	[Ali, Veli, Can]	kopya yok; `yeni` 'yi sarar
`// gorunum.set(0, "Ece")`	[Ali, Veli]	[Ali, Veli, Can]	[Ali, Veli, Can]	açılsaydı: `UnsupportedOperationException`
`yeni.set(0, "Ece")`	[Ali, Veli]	[Ece, Veli, Can]	[Ece, Veli, Can]	görünüm değişikliği görür
`println(gorunum)`	çıktı: `[Ece, Veli, Can]`			salt okunur ≠ değiştirilemez

Tablo 6.5: Üç değişkenin her satırdan sonraki durumu.





Şekil 6.6: Bellekteki durum. (a) `new ArrayList<>(sabit)` öğeleri yeni bir diziye kopyalar. (b) Kopyaya "Can" eklenir; `sabit` değişmez. (c) `unmodifiableList`, `yeni`'nin dizisine bakan bir sarmalayıcı döndürür. (d) `yeni.set(0, "Ece")` sonrası değişiklik görünümden de görülür.

İzlemenin son satırı sorunun cevabını verir: `gorunum` hiç değiştirilmediği hâlde içeriği değişmiştir. Çünkü `gorunum` bir kopya değil, `yeni`'nin dizisine açılmış salt okunur bir penceredir. Pencereden içeri uzanıp bir şeyi değiştirmek yasaktır, ama oda değişirse pencereden bakan da değişikliği görür.

Gerçekten değişmeyen bir liste isteniyorsa `List.copyOf(yeni)` kullanılır. Bu çağrı öğeleri yeni ve değiştirilemez bir listeye kopyalar. Sonradan `yeni`'ye öğe eklense de kopya etkilenmez. Üç seçeneğin farkı şöyle özetlenebilir:

Oluşturma	Kendisi değiştirilebilir mi?	Kaynak değişirse ne olur?	Maliyet
`new ArrayList<>(l)`	evet	etkilenmez (kopya)	*O(n)*
`Collections.unmodifiableList(l)`	hayır	değişikliği görür (görünüm)	*O(1)*
`List.copyOf(l)`	hayır	etkilenmez (kopya)	*O(n)*

Tablo 6.6: Kopya, salt okunur görünüm ve değiştirilemez liste.

Sık yapılan hata

"Salt okunur" ile "değiştirilemez" sözcüklerini eş anlamlı sanmak. Bir sınıf, iç listesini dışarıya `Collections.unmodifiableList(liste)` olarak verirse dışarıdaki kod listeyi değiştiremez; ama sınıf kendi içinde listeyi değiştirdikçe dışarıdaki kodun elindeki liste de değişir. Dışarıya verilen listenin o anki hâlinin dondurulması isteniyorsa `List.copyOf(liste)` verilir.

6.3 List metotları

`List` arayüzü `Collection`'dan devraldıklarına ek olarak konuma dayalı birçok metot tanımlar. Bu kesim en çok kullanılanlarını, her birinin tuzaklarıyla birlikte ele alır. Örneklerin hepsi `ArrayList` üzerindedir; metotların *davranışı* öteki `List` sınıflarında da aynıdır, yalnız maliyetleri değişir.

6.3.1 Ana metotlar ve indis sınırları

Grup	Metot	Açıklama
Ekleme	`add(E e)`	sona ekler, `true` döner
	`add(int index, E e)`	verilen indise ekler; sonrakiler bir sağa kayar
Çıkarma	`remove(Object o)`	öğenin ilk geçtiği yeri siler; bulduysa `true`
	`remove(int index)`	indisteki öğeyi siler ve döndürür
Erişim	`get(int index)`	indisteki öğeyi döndürür
	`set(int index, E e)`	öğeyi değiştirir, eskisini döndürür
Uzunluk	`size()`	öğe sayısı

Tablo 6.7: `List` arayüzünün ana metotları.

Tablodaki `E`, listenin tür parametresidir; `List<String>` için `E` yerine `String` konur. Dönüş değerleri dikkatle okunmalıdır. `set`, yeni değeri değil *eski* değeri döndürür: `[A, B, C]` listesinde `set(0, "X")` "A" döndürür. `remove(int)` silinen öğeyi, `remove(Object)` ise silme olup olmadığını (`boolean`) döndürür.

İndisle çalışan her metot önce indisin geçerli olup olmadığını denetler. Geçersiz indis `IndexOutOfBoundsException` fırlatır. Geçerli aralık metoda göre değişir:

- `get`, `set` ve `remove(int)` için geçerli indisler *0 … size() − 1*'dir. Yani *index < 0* ya da *index ≥ size()* hatadır.
- `add(int, E)` için geçerli indisler *0 … size()*'dır. `add(size(), e)` sona eklemektir ve geçerlidir; yalnız *index > size()* hatadır.

Aradaki fark mantıklıdır: `get` var olan bir öğeyi ister, `add` ise iki öğenin arasına ya da en sona yer açar. *n* öğeli bir listede *n* öğe vardır ama *n + 1* "ara" vardır: ilk öğenin önü, her iki öğenin arası ve son öğenin arkası.

Örnek 6.2

`l = [A, B, C]` (uzunluk 3) olsun. Aşağıdaki çağrılardan hangileri geçerlidir?

1. `l.get(3)` : geçerli indisler 0, 1, 2'dir; 3 ≥ 3 olduğundan `IndexOutOfBoundsException`.
2. `l.add(3, "D")` : `add` için üst sınır `size()`'dır; 3 ≤ 3 olduğundan geçerlidir ve D sona eklenir. Liste `[A, B, C, D]` olur.
3. `l.set(0, "X")` : geçerlidir; liste `[X, B, C, D]` olur ve çağrı "A" döndürür.
4. `l.add(5, "E")` : *5 > size() = 4* olduğundan hata. Liste bir "delik" bırakarak büyüyemez.

6.3.2 Kod izleme: tekrar, null ve arama

Listeyi kümeden ayıran iki özellik vardır: aynı öğe tekrar eklenebilir ve öğelerin sırası anlamlıdır. Aşağıdaki program bu iki özelliği ve `null` öğeyi sınar:

```
public class ListDemo {   // import java.util.ArrayList; import java.util.List;
    public static void main(String[] args) {
        List<String> isimler = new ArrayList<>();
        isimler.add("Ali");
        isimler.add("Ali");           // tekrar: izin var
        isimler.add(null);            // null: izin var
        isimler.add("Ali");
        System.out.println(isimler);          // [Ali, Ali, null, Ali]
        System.out.println(isimler.size());   // 4
        System.out.println(isimler.indexOf("Ali"));      // 0 (ilk geçiş)
        System.out.println(isimler.lastIndexOf("Ali"));  // 3
    }
}
```

Programın sunumdaki izleme biçimi, sona bir satır daha ekler: `System.out.println(isimler.indexOf(null));`. Aranan `null` olunca arama algoritmasının nasıl değiştiğini görmek için bu satır önemlidir.

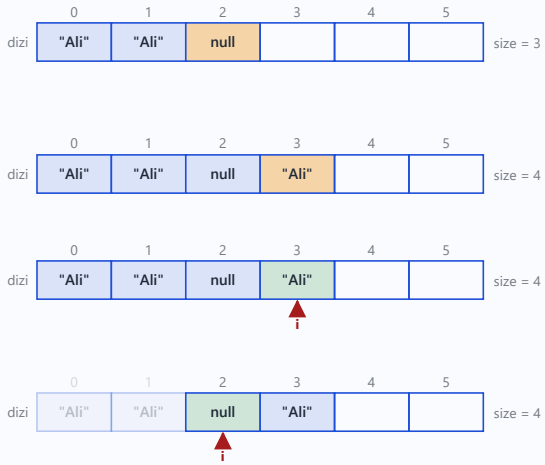
Satır	İç dizi (0–3)	`size`	Çıktı / dönüş
`new ArrayList<>()`	[ ]	0	—
`add("Ali")`	[Ali]	1	`true`
`add("Ali")`	[Ali, Ali]	2	`true` (tekrar serbest)
`add(null)`	[Ali, Ali, null]	3	`true` (`null` serbest)
`add("Ali")`	[Ali, Ali, null, Ali]	4	`true`
`println(isimler)`	—	4	`[Ali, Ali, null, Ali]`
`size()`	—	4	4
`indexOf("Ali")`	0. hücrede eşleşme	4	0
`lastIndexOf("Ali")`	sondan: 3. hücrede eşleşme	4	3
`indexOf(null)`	0: hayır, 1: hayır, 2: evet	4	2

Tablo 6.8: `ListDemo` izlemesi. `dizi` , `ArrayList` 'in iç dizisidir.

Çıktının ilk satırı `toString()` 'in işidir: öğeler sırasıyla, virgülle ayrılarak köşeli parantez içinde yazılır; `null` öğe “null” sözcüğüyle görünür. `size()` öğe sayısını, yani 4'ü verir. Burada iç dizinin kapasitesi 10'dur (boş bir `ArrayList` ilk eklemede 10 hücre ayırır), ama `size()` kapasiteyi değil öğe sayısını döndürür.

`indexOf` diziyi baştan tarar ve ilk eşleşmede durur. Karşılaştırmayı `==` ile değil `equals` ile yapar: `"Ali".equals(dizi[0])` doğru çıkar ve 0 döner. `lastIndexOf` aynı işi sondan başlayarak yapar; ilk baktığı hücre 3'tür ve eşleşme hemen bulunur. Öğe hiç yoksa ikisi de −1 döndürür. İkisi de en kötü durumda bütün listeyi tarar; maliyetleri *O(n)*'dir.

Son satır ince bir noktayı gösterir. Aranan değer `null` ise `indexOf`, `o.equals(dizi[i])` çağıramaz: `null.equals(...)` bir `NullPointerException` fırlatırdı. Bu yüzden `ArrayList`'in kodu iki ayrı döngü içerir. Aranan `null` ise hücreler `dizi[i] == null` ile, değilse `o.equals(dizi[i])` ile karşılaştırılır. İzlemede 0. ve 1. hücrelerde "Ali" vardır, 2. hücrede `null` bulunur ve 2 döner.



Şekil 6.7: İç dizinin dört anı. (a) `null` 2. hücreye eklenmiş. (b) Üçüncü "Ali" 3. hücrede; `size = 4`. (c) `lastIndexOf` sondan başlar ve 3. hücrede durur. (d) `indexOf(null)`, `==` karşılaştırmasıyla 2. hücreyi bulur.

Liste ile küme arasındaki fark bu örnekte açıkça görülür. Liste aynı "Ali"yi üç kez tutar ve her birini ayrı bir konumda saklar. Tekil öğeler gerekiyorsa, yani her değerin en fazla bir kez bulunması isteniyorsa, liste yerine küme (`Set`) kullanılır.

Sık yapılan hata

Her listenin `null` kabul ettiğini sanmak. `ArrayList` ve `LinkedList` kabul eder, ama `List.of(...)` ile oluşturulan değiştirilemez liste `null` öğe görünce `NullPointerException` fırlatır. Aynı liste üzerinde `contains(null)` ya da `indexOf(null)` çağrısı da aynı istisnayı verebilir.

6.3.3 remove(int) mi, remove(Object) mi?

`List`'te iki ayrı `remove` metodu vardır. Biri indis alır (`remove(int index)`), öteki öğe alır (`remove(Object o)`). Aynı adlı ama parametreleri farklı metotlara **aşırı yükleme** (*overload*) denir. Derleyici, çağrıdaki argümanın türüne bakarak hangisinin çalışacağına karar verir.

Liste `String` tutarken karışıklık olmaz: `remove(1)` indis, `remove("Ali")` öğe silmedir. Liste `Integer` tutunca iş değişir, çünkü bir tam sayı hem indis hem öğe olabilir.

```
List<Integer> sayilar = new ArrayList<>(List.of(10, 20, 30));
sayilar.remove(1);                    // indis 1 silinir → [10, 30]
sayilar.remove(Integer.valueOf(10));  // 10 değeri silinir → [30]
sayilar.remove(Integer.valueOf(99));  // yok → false, liste değişmez
sayilar.remove(5);                    // IndexOutOfBoundsException
```

Metot	Ne yapar?	Dönüş	Dizi tabanlı listede maliyet
`remove(int index)`	verilen konumdaki öğeyi siler	silinen öğe (`E`)	*O(n)*: sonrakiler kayar
`remove(Object o)`	öğenin ilk geçtiği yeri arar ve siler	silindi mi (`boolean`)	*O(n)*: arama + kaydırma

Tablo 6.9: İki `remove` metodunun karşılaştırması.

Sunumdaki izleme biçimi dönüş değerlerini de değişkenlere alır:

```
List<Integer> sayilar =
    new ArrayList<>(List.of(10, 20, 30));
Integer a = sayilar.remove(1);
boolean b = sayilar.remove(Integer.valueOf(10));
boolean c = sayilar.remove(Integer.valueOf(99));
sayilar.remove(5);   // IndexOutOfBoundsException
```

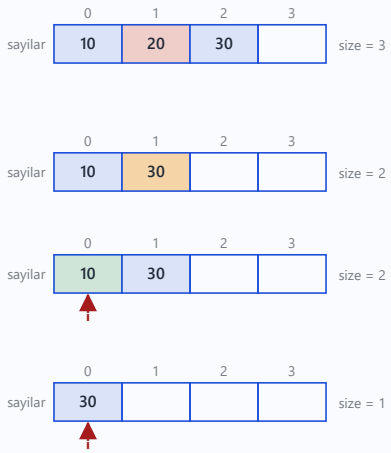
Çağrı	Seçilen metot	Liste (sonra)	Dönüş
(başlangıç)	—	[10, 20, 30]	—
`remove(1)`	`remove(int)` : indis 1	[10, 30]	`a = 20`
`remove(Integer.valueOf(10))`	`remove(Object)` : değer 10	[30]	`b = true`
`remove(Integer.valueOf(99))`	`remove(Object)` : değer 99	[30]	`c = false`
`remove(5)`	`remove(int)` : indis 5	[30]	`IndexOutOfBoundsException`

Tablo 6.10: `remove` çağrılarının izlemesi.

Derleyicinin seçimini anlamak için Java'nın aşırı yükleme kuralını bilmek gerekir. Derleyici önce kutulama ya da kutudan çıkarma gerektirmeyen bir eşleşme arar. `remove(1)`'deki argüman ilkel bir `int`'tir ve `remove(int)` ona birebir uyar; `remove(Object)`'i seçmek için 1'i `Integer`'a kutulamak gerekirdi. Kutulamasız eşleşme önce geldiği için indis silmesi seçilir. 1. indisteki 20 silinir, 30 bir sola kayar ve silinen 20 döndürülür.

`Integer.valueOf(10)` ise zaten bir nesnedir; `remove(int)`'i seçmek için kutudan çıkarmak gerekirdi. Bu kez `remove(Object)` doğrudan uyar. Metot listeyi baştan tarar, `Integer.valueOf(10).equals(10)` 0. indiste doğru çıkar, öğe silinir ve `true` döner. 99 için tarama listenin sonuna kadar gider, eşleşme bulunmaz; liste değişmez ve `false` döner.

Son çağrıda liste tek öğelidir. `remove(5)` indis silmesidir ve önce *5 < size() = 1* denetimi yapılır. Denetim başarısız olur; Java'nın mesajı tam olarak şudur: `Index 5 out of bounds for length 1`.



Şekil 6.8: (a) `remove(1)` : silinecek hücre 1. (b) 30 sola kaydı, `a = 20` . (c) `remove(Integer.valueOf(10))` değeri 0. hücrede bulur. (d) 99 aranırken liste sonuna gelindi; eşleşme yok.

Sık yapılan hata

`List<Integer>` 'ten bir *değeri* silmek isterken `l.remove(7)` yazmak. Bu çağrı 7 değerini değil, 7. indisteki öğeyi siler; liste kısaysa istisna fırlatır. Değere göre silmek için argüman nesne olmalıdır: `l.remove(Integer.valueOf(7))` ya da `l.remove((Integer) 7)` . Ters yönde de tuzak vardır: `l.remove(l.get(1))` çağrısında `get` bir `Integer` döndürdüğü için *değer* silmesi yapılır.

6.3.4 Lambda ile toplu işlem: replaceAll ve sort

Listenin her öğesine aynı işlemi uygulamak sık rastlanan bir iştir. Bunu indisli bir döngüyle yazmak mümkündür, ama Java 8'den beri `List` bu iş için hazır metotlar sunar. Metotlara yapılacak iş bir **lambda ifadesi** olarak verilir. Lambda ifadesi, adı olmayan kısa bir fonksiyondur: `isim -> isim.toUpperCase()` "bir `isim` al, büyük harfli hâlini döndür" demektir.

```
List<String> isimler =
        new ArrayList<>();
isimler.add("Ali");
isimler.add("Ahmet");
isimler.add("Mehmet");
isimler.replaceAll(
        isim -> isim.toUpperCase());
System.out.println(isimler);
// [ALI, AHMET, MEHMET]
```

`replaceAll`, listenin her indisi için lambdayı çağırır ve sonucu *aynı yere* yazar. Yeni bir liste oluşturulmaz; öğeler yerinde değişir. Sonuç `[ALI, AHMET, MEHMET]`'tir.

```
List<Integer> sayilar =
        new ArrayList<>();
sayilar.add(5);
sayilar.add(3);
sayilar.add(1);
sayilar.sort((s1, s2) ->
        Integer.compare(s1, s2));
System.out.println(sayilar);
// [1, 3, 5]
```

`sort` ise iki öğeyi karşılaştıran bir lambda alır. Bu tür bir lambdaya **karşılaştırıcı** (*comparator*) denir. Karşılaştırıcı, ilk öğe ikinciden küçükse negatif, eşitse sıfır, büyükse pozitif bir sayı döndürür. `Integer.compare(s1, s2)` tam olarak bunu yapar. Doğal sıra için daha kısa yazımlar da vardır: `sayilar.sort(null)` ya da `sayilar.sort(Comparator.naturalOrder())`. Ters sıra için `Comparator.reverseOrder()` verilir.

Sunumdaki izleme, iki işi tek programda birleştirir ve büyük harfe çevirmede bir düzeltme yapar:

```
List<String> isimler = new ArrayList<>(
    List.of("Ali", "Ahmet", "Mehmet"));
isimler.replaceAll(
    isim -> isim.toUpperCase(Locale.ROOT));
List<Integer> sayilar = new ArrayList<>(
    List.of(5, 3, 1));
sayilar.sort((s1, s2) -> Integer.compare(s1, s2));
System.out.println(isimler + " " + sayilar);
```

Buradaki `Locale.ROOT` argümanı Türkçe konuşan programcılar için özellikle önemlidir. `toUpperCase()`, argümansız çağrıldığında bilgisayarın dil ayarını (yerelini) kullanır. Türkçe yerelde küçük "i"nin büyüğü "İ"dir; bu yüzden "Ali" "ALİ" olur. Programın her bilgisayarda aynı sonucu vermesi isteniyorsa yerelden bağımsız `Locale.ROOT` verilir ve "Ali" "ALI" olur.

`sort`'un içinde ne olduğunu izlemek ilginçtir. Java nesne listelerini **TimSort** adlı bir algoritmayla sıralar. TimSort önce dizide zaten sıralı ya da ters sıralı olan parçaları (*run*) arar. `[5, 3, 1]` listesinde bu arama şöyle yürür:

Çağrı	Sonuç	Anlamı
`compare(3, 5)`	−1	3 < 5: dizinin başı azalan bir parça
`compare(1, 3)`	−1	1 < 3: azalan parça sürüyor; uzunluk 3
(ters çevirme)	—	tüm dizi tek azalan parça; ters çevrilir: `[1, 3, 5]`

Tablo 6.11: `[5, 3, 1]` üzerinde `sort`'un yaptığı karşılaştırmalar.

Yalnız iki karşılaştırmayla sıralama biter. Zaten sıralı ya da tam ters sıralı listelerde TimSort *n − 1* karşılaştırma yapar; genel durumda maliyeti *O(n log n)*'dir. Programın çıktısı `[ALI, AHMET, MEHMET] [1, 3, 5]` 'tir.



Şekil 6.9: (a) `replaceAll` 1. hücreye "AHMET" yazıyor. (b) Sıralama öncesi `[5, 3, 1]`. (c) İki karşılaştırma sonrası üç öğe tek azalan parça olarak işaretli. (d) Parça ters çevrildi: `[1, 3, 5]`.

Sık yapılan hata

Karşılaştırıcıyı çıkarmayla yazmak: `(a, b) -> a - b`. Küçük sayılarda doğru çalışır, ama `a` çok büyük, `b` çok küçük bir negatif sayıysa çıkarma `int` sınırını taşar ve işaret ters döner. Sıralama sessizce yanlış sonuç verir. `Integer.compare(a, b)` taşma yapmaz ve her zaman doğrudur.

6.3.5 İndisle erişim, jenerikler ve kutulama

Listedeki bir öğeye konumuyla erişmek için `get(i)` kullanılır. Bu basit işlem, Java'nın iki önemli dil özelliğiyle iç içedir: jenerik türler ve kutulama.

```
List<String> isimler =
        new ArrayList<>();
isimler.add("Ali");
isimler.add("Ahmet");
isimler.add("Mehmet");
String ilkIsim = isimler.get(0);
System.out.println(ilkIsim);  // Ali
isimler.get(3);  // IndexOutOfBounds
```

```
List<String> isimler =
        new ArrayList<>();
isimler.add("Ali");
isimler.add(5);   // derleme hatası

List<Integer> sayilar =
        new ArrayList<>();
sayilar.add(5);   // otomatik kutulama
int x = sayilar.get(0);  // kutudan çıkarma
```

Jenerik türler (*generics*), bir sınıfın ya da arayüzün hangi türde öğeyle çalışacağını bir tür parametresiyle bildirmesidir. `List<String>` yazıldığında derleyici bu listeye yalnız `String` girebileceğini bilir. `isimler.add(5)` satırı bu yüzden *derleme* hatasıdır; yanlış türdeki öğe program çalışmadan yakalanır. Ayrıca `get` doğrudan `String` döndürür; dönen değeri `(String)` ile dönüştürmeye (*cast*) gerek kalmaz.

Tür parametresi ilkel bir tür olamaz. `List<int>` yazılamaz; onun yerine `int`'in **sarmalayıcı sınıfı** (*wrapper class*) `Integer` kullanılır: `List<Integer>`. Her ilkel türün bir sarmalayıcısı vardır: `double` için `Double`, `char` için `Character`, `boolean` için `Boolean`.

İlkel değerle sarmalayıcı nesne arasındaki dönüşümü derleyici kendisi yapar:

- **Otomatik kutulama** (*autoboxing*): `sayilar.add(5)` satırında derleyici gizlice `sayilar.add(Integer.valueOf(5))` yazar. Listede ilkel 5 değil, 5'i tutan bir `Integer` nesnesine referans durur.
- **Kutudan çıkarma** (*unboxing*): `int x = sayilar.get(0);` satırında derleyici `sayilar.get(0).intValue()` yazar ve nesnenin içindeki ilkel değeri alır.

Sunumdaki izleme programı bu adımların hepsini sırayla gösterir:

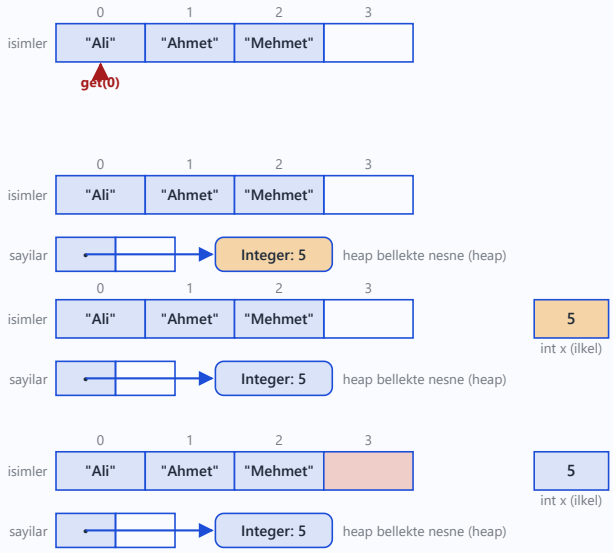
```
List<String> isimler = new ArrayList<>();
isimler.add("Ali");
isimler.add("Ahmet");
isimler.add("Mehmet");
String ilkIsim = isimler.get(0);
System.out.println(ilkIsim);
List<Integer> sayilar = new ArrayList<>();
sayilar.add(5);          // Integer.valueOf(5)
int x = sayilar.get(0);  // .intValue()
isimler.get(3);          // IndexOutOfBounds
```

Satır	Derleyicinin ürettiği	Durum	Sonuç
`new ArrayList<>()`	—	`isimler` boş	yalnız `String` girebilir
üç `add`	—	[Ali, Ahmet, Mehmet]	indis 0, 1, 2
`get(0)`	—	*0 < 3* denetimi geçer	“Ali”, *O(1)*

Satır	Derleyicinin ürettiği	Durum	Sonuç
`println(ilkIsim)`	—	—	çıktı: `Ali`
`sayilar.add(5)`	`add(Integer.valueOf(5))`	listede bir `Integer` referansı	kutulama
`int x = sayilar.get(0)`	`get(0).intValue()`	`x = 5`	kutudan çıkarma
`isimler.get(3)`	—	*3 < 3* yanlış	`IndexOutOfBoundsException`

Tablo 6.12: `get`, jenerikler ve kutulama izlemesi.

`get(0)`'ın maliyeti dikkat çekicidir. `ArrayList`, öğeleri bir dizide tuttuğu için önce indisi denetler, sonra `dizi[0]`'ı doğrudan okur. Dizinin hangi hücresinin okunacağı bir adres hesabıyla bulunur; liste ne kadar uzun olursa olsun süre değişmez. Bu yüzden `ArrayList`'te `get` *O(1)*'dir. `LinkedList`'te ise aynı işlem düğümler üzerinde yürümeyi gerektirir ve pahalıdır (6.4.2).



Şekil 6.10: (a) `get(0)` 0. hücreyi doğrudan okur. (b) `sayilar.add(5)`: listede yığın bellekteki bir `Integer` nesnesine referans durur. (c) `get(0).intValue()` ilkel 5'i `x`'e verir. (d) `get(3)`: geçerli indisler 0–2'dir; istisna.

Kutulamanın görünmeyen bir bedeli vardır. Her `Integer` ayrı bir nesnedir. İlkel bir `int` 4 bayt tutar; bir `Integer` nesnesi ise başlığıyla birlikte 16 bayt civarında yer kaplar. Listede ayrıca ona bir referans tutulur. Milyonlarca sayıyla çalışan programlarda `int[]` dizisi `List<Integer>`'den hem daha az bellek kullanır hem de daha hızlıdır.

Sık yapılan hata

Kutudan çıkarmanın her zaman güvenli olduğunu sanmak. `List<Integer>` içinde `null` bir öğe varsa `int x = l.get(0);` satırı `null.intValue()` çağırmaya çalışır ve `NullPointerException` fırlatır. Hata, kodda hiçbir metot çağrısı görünmeyen bir atama satırında ortaya çıktığı için şaşırtıcıdır.

6.3.6 Spliterator: bölünebilir yineleyici

Yineleyici öğeleri tek tek, baştan sona verir. Çok çekirdekli bir işlemcide büyük bir listeyi işlerken bu yetmez; işin parçalara bölünüp her parçanın ayrı bir **iş parçacığına** (*thread*) verilmesi istenir. `Spliterator` (bölünebilir yineleyici) bu ihtiyaçtan doğmuştur. Adı İngilizce "split" (böl) ve "iterator" sözcüklerinin birleşimidir.

```
List<String> kelimeler = new ArrayList<>();
kelimeler.add("Java");
kelimeler.add("Python");
kelimeler.add("C++");
Spliterator<String> split =
        kelimeler.spliterator();
while (split.tryAdvance(
        System.out::println)) {
    // her çağrı bir öğeyi işler
}
```

`Spliterator`'ın iki temel metodu vardır:

- `tryAdvance(eylem)`: sırada öğe varsa ona eylemi uygular ve `true` döner; öğe kalmadıysa hiçbir şey yapmadan `false` döner. `hasNext()` ile `next()`'in tek bir çağrıda birleşmiş hâlidir. Yukarıdaki kodda eylem `System.out::println`'dır; bu, "öğeyi yazdır" anlamına gelen bir metot referansıdır. Döngünün gövdesi boştur, çünkü bütün iş koşul içinde yapılır. Çıktı alt alta Java, Python, C++'tır.
- `trySplit()`: kalan öğelerin bir kısmını yeni bir `Spliterator`'a devreder ve onu döndürür. Böylece iki parça iki ayrı iş parçacığına verilebilir. Bölünecek kadar öğe kalmamışsa `null` döner.

Java'nın paralel akışları (`parallelStream()`) arka planda tam olarak bu düzeneği kullanır: listeyi `trySplit` ile parçalara böler ve her parçayı `tryAdvance` ile ayrı ayrı işler. Bölmenin nasıl yapıldığını sunumdaki izleme gösterir:

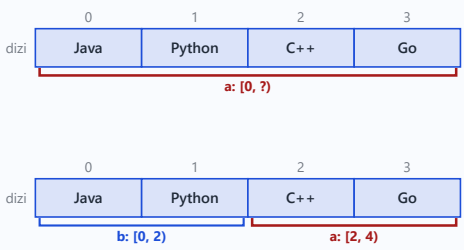
```
List<String> kelimeler = new ArrayList<>(
    List.of("Java", "Python", "C++", "Go"));
Spliterator<String> a = kelimeler.spliterator();
Spliterator<String> b = a.trySplit();
while (b.tryAdvance(System.out::println)) { }
while (a.tryAdvance(System.out::println)) { }
```

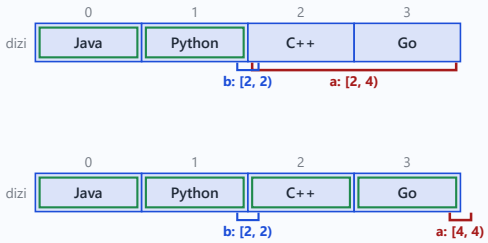
`ArrayList`'in spliterator'ı yalnız iki sayı tutar: sıradaki indis (`index`) ve bitiş sınırı (*fence*). Başta indis 0, sınır −1'dir. −1 "henüz belirlenmedi" anlamına gelir: sınır, spliterator ilk kullanıldığında `size()` okunarak saptanır. Bu tembel davranış, spliterator oluşturulduktan sonra ama kullanılmadan önce listeye eklenen öğelerin de kapsanmasını sağlar.

Çağrı	`a`	`b`	Dönüş / çıktı
`kelimeler.spliterator()`	index 0, fence −1	—	sınır henüz yok
`a.trySplit()`	*[2, 4)*	*[0, 2)*	fence = 4, orta = *(0 + 4) / 2 = 2*
`b.tryAdvance`	*[2, 4)*	*[1, 2)*	Java; `true`
`b.tryAdvance`	*[2, 4)*	*[2, 2)*	Python; `true`
`b.tryAdvance`	*[2, 4)*	*[2, 2)*	öğe yok; `false`
`a.tryAdvance`	*[3, 4)*	—	C++; `true`
`a.tryAdvance`	*[4, 4)*	—	Go; `true`
`a.tryAdvance`	*[4, 4)*	—	öğe yok; `false`

Tablo 6.13: `trySplit` ve `tryAdvance` izlemesi. Aralıklar *[index, fence)* biçiminde yazılmıştır.

`trySplit`'in kararı küçük bir ayrıntı içerir: listenin *ilk* yarısı yeni spliterator'a (`b`) verilir, eski spliterator (`a`) ikinci yarıyla kalır. Orta nokta *(index + fence) / 2* ile tam sayı bölmesiyle bulunur. Beş öğeli bir listede orta *(0 + 5) / 2 = 2* olur; yeni parça 2, eski parça 3 öğe alır. Program iki parçayı ardı ardına işlediği için çıktı yine Java, Python, C++, Go sırasındadır. Paralel bir akışta ise iki parça aynı anda, farklı çekirdeklerde işlenebilirdi.





Şekil 6.11: (a) `spliterator()` : sınır henüz belirsiz. (b) `trySplit()` : `b` *[0, 2)*, `a` *[2, 4)*. (c) `b` iki öğesini işledi ve tükendi. (d) `a` da tükendi; dört öğenin her biri bir kez işlendi.

6.4 Gezinme

Bir listenin bütün öğelerini sırayla ziyaret etmeye **gezinme** (*traversal*) denir. Toplamı bulmak, en büyüğü aramak, her öğeyi yazdırmak, koşula uyanları silmek hep gezinme gerektirir. Gezinme basit bir iş gibi görünür, ama yanlış araç seçilirse doğrusal sürmesi gereken bir döngü karesel süreye çıkabilir ya da program bir istisnayla durabilir. Bu kesim gezinmenin yollarını ve tuzaklarını anlatır.

6.4.1 Koleksiyonları gezmenin üç yolu

Java'da bir listeyi gezmenin üç yolu vardır. Üçü de aynı öğeleri aynı sırada verir; farkları, neye izin verdikleri ve ne kadar sürdükleridir.

1. İndisli for döngüsü.

```
int n = list.size();
for (int i = 0;
     i < n; i++) {
  String s =
      list.get(i);
  ...
}
```

Bu yol yalnız `List`'te çalışır, çünkü `get(i)` gerektirir. Döngü içinde indisin kendisi de gerekiyorsa (örneğin "3. öğe" diye yazdırmak ya da komşu öğeleri karşılaştırmak için) doğal seçimdir.

2. forEach ya da for-each.

```
list.forEach(
  System.out::println);

// ya da for-each:
for (String s : list) ...
```

Her `Iterable` için çalışır ve en kısa yazımdır. `list.forEach(...)` bir lambda ya da metot referansı alır; for-each döngüsü ise derleyicinin bir yineleyiciye çevirdiği tanıdık sözdizimidir.

3. Açıkça yazılmış yineleyici.

```
Iterator<String> it =
    list.iterator();
while (it.hasNext()) {
  String s = it.next();
}
```

En uzun yazımdır, ama tek bir ayrıcalığı vardır: gezinirken öğeyi güvenle silmeye izin verir. Bu ayrıcalığın gerekçesi, dışarıdan silmenin yol açtığı istisnadır (6.4.4).

Üç yolun maliyeti listenin türüne göre değişir. `ArrayList`'te `get(i)` *O(1)* olduğu için üçü de toplamda *O(n)* sürer. `LinkedList`'te ise `get(i)` düğümler üzerinde yürümek zorundadır ve *O(n)*'dir; indisli döngü bu yürüyüşü her adımda yeniden yapar. Sonuç olarak `LinkedList`'te indisli döngü $O(n^2)$, forEach ve yineleyici ise *O(n)* sürer.

6.4.2 Kod izleme: LinkedList'te indisli döngünün bedeli

`LinkedList` çift yönlü bir bağlı listedir: her düğüm hem sonrakini hem öncekini bilir, liste de ilk (`first`) ve son (`last`) düğümü tutar. `get(i)` çağrıldığında Java, iç `node(i)` metoduyla aranan düğüme yürür. Akıllı bir kestirme kullanır: $i < n/2$ ise baştan ileri, değilse sondan geri yürür. Böylece yürünen bağlantı sayısı $\min(i, n-1-i)$ olur.

Aşağıdaki program bu yürüyüşü bir sayaçla ölçer. `adim` değişkeni, döngü boyunca bütün `get` çağrılarının toplam kaç bağlantı yürüdüğünü biriktirir:

```
List<String> list = new LinkedList<>(
    List.of("A", "B", "C", "D", "E", "F"));
int n = list.size(), adim = 0;
for (int i = 0; i < n; i++) {
  String s = list.get(i);  // node(i) yürür
  adim += Math.min(i, n - 1 - i);
}
System.out.println(adim);
```

`i`	Karar	Başlangıç	Yürünen bağlantı	`adim` (toplam)
0	$0 < 3$	`first`	0	0
1	$1 < 3$	`first`	1	1
2	$2 < 3$	`first`	2	3
3	$3 \geq 3$	`last`	2	5

`i`	Karar	Başlangıç	Yürünen bağlantı	`adim` (toplam)
4	$4 \geq 3$	`last`	1	6
5	$5 \geq 3$	`last`	0	6

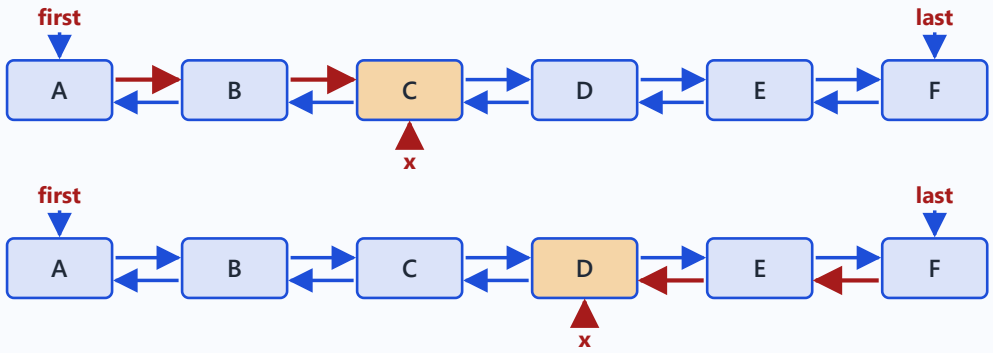
Tablo 6.14: 6 düğümlü `LinkedList`'te indisli döngünün yürüdüğü bağlantılar ($n/2 = 3$).

Program 6 yazdırır. Bu küçük sayı yanıltıcı olabilir; önemli olan n büyüdükçe toplamın nasıl büyüdüğüdür. Çift n için yürüyüşler $0, 1, \ldots, n/2 - 1$ dizisini iki kez (baştan ve sondan) verir:

> $$2 \cdot (0 + 1 + \ldots + (n/2 - 1)) = 2 \cdot (n/2)(n/2 - 1)/2 = n^2/4 - n/2$$

$n = 6$ için bu $9 - 3 = 6$'dır ve tabloyla uyuşur. $n = 1000$ için 249.500 bağlantı yürünür. Toplam $n^2/4$ gibi büyür; döngü $O(n^2)$'dir. Baştan ve sondan yürüme kestirmesi işi yalnız yarıya indirir, büyüme hızını değiştirmez.

Yineleyici ise bambaşka çalışır. Bulunduğu düğümü hatırlar; her `next()` çağrısında yalnız bir bağlantı ilerler. Altı öğenin hepsini gezmek için toplam $n - 1 = 5$ bağlantı yürünür. Maliyet $O(n)$'dir.



Şekil 6.12: `LinkedList`'te `get`'in yürüyüşü. Üstte `get(2)` : $2 < 3$ olduğundan `first`'ten başlar ve iki bağlantı ileri gider. Altta `get(3)` : $3 \geq 3$ olduğundan `last`'tan başlar ve iki bağlantı geri gider. Kırmızı oklar yürünen bağlantılardır.

Sık yapılan hata

Bir metodun parametresi `List` türündeyken içeride `for (int i = 0; i < l.size(); i++)` … `l.get(i)` … yazmak. Metot bir `ArrayList`'le sınandığında hızlıdır; aynı metot bir gün `LinkedList` alırsa karesel süreye düşer. Türü bilinmeyen bir listeyi gezmenin güvenli yolu for-each ya da yineleyicidir.

6.4.3 Iterator arayüzü ve imleç

`Iterator` (yineleyici) arayüzü üç metot tanımlar:

`hasNext()`

Gezilecek öğe kaldı mı? Sona gelindiyse `false` döner. Durumu değiştirmez; arka arkaya kaç kez çağrılırsa çağrılsın aynı cevabı verir.

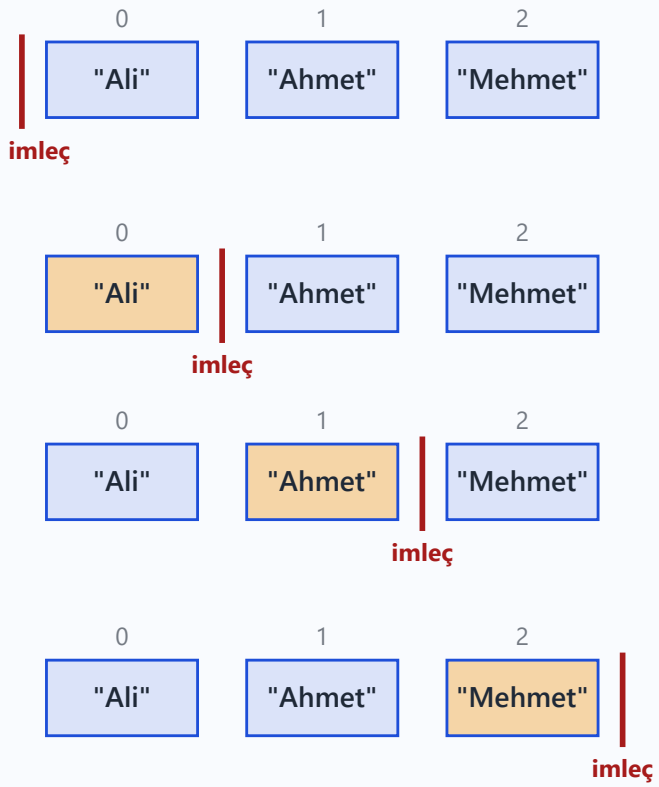
`next()`

Sıradaki öğeyi döndürür ve yineleyiciyi bir ilerletir. Öğe kalmadıysa `NoSuchElementException` fırlatır.

`remove()`

Son `next()` çağrısının döndürdüğü öğeyi koleksiyondan siler. İsteğe bağlı bir işlemdir: desteklemeyen koleksiyonlar `UnsupportedOperationException` fırlatır.

Yineleyicinin davranışını anlamanın en iyi yolu bir **imleç** (*cursor*) düşünmektir. İmleç, bir kelime işlemcideki yazı imleci gibi, öğelerin *üzerinde* değil *arasında* durur. Başta ilk öğenin önündedir. Her `next()` imleci bir öğenin üzerinden atlatır ve üzerinden atladığı öğeyi döndürür. İmleç son öğenin arkasına geldiğinde `hasNext()` yanlış olur.



Şekil 6.13: İmleç modeli. (a) `iterator()` sonrası imleç ilk öğenin önündedir; `hasNext()` doğrudur. (b) `next()` "Ali"yi döndürür, imleç onun arkasına geçer. (c) İkinci `next()` "Ahmet"i döndürür. (d) Üçüncü `next()` "Mehmet"i döndürür; imleç sondadır ve `hasNext()` yanlıştır.

Yineleyici, listenin kendisinden ayrı bir nesnedir. Aynı liste üzerinde aynı anda birden çok yineleyici açılabilir ve her biri kendi imlecini tutar. `ArrayList` 'in yineleyicisinde imleç bir tam sayıdır: `cursor`, bir sonraki `next()` 'in döndüreceği öğenin indisidir. `hasNext()` yalnız `cursor != size` denetimini

yapar.

```
Iterator<String> iterator = isimler.iterator();
while (iterator.hasNext()) {
    String isim = iterator.next();
    System.out.println(isim);
}
```

Çağrı	Denetim	Döner	`cursor` (sonra)	Çıktı
`iterator()`	—	yineleyici	0	—
`hasNext()`	0 ≠ 3	`true`	0	—
`next()`	—	"Ali"	1	Ali
`hasNext()`	1 ≠ 3	`true`	1	—
`next()`	—	"Ahmet"	2	Ahmet
`hasNext()`	2 ≠ 3	`true`	2	—
`next()`	—	"Mehmet"	3	Mehmet
`hasNext()`	3 = 3	`false`	3	döngü biter

Tablo 6.15: `[Ali, Ahmet, Mehmet]` üzerinde yineleyici izlemesi (`size = 3`).

Döngüden sonra bir kez daha `next()` çağrılırsa imlecin önünde öğe yoktur ve `NoSuchElementException` fırlatılır. Bu yüzden `next()` çağrısının önüne her zaman bir `hasNext()` denetimi konur ya da döngü for-each ile yazılır.

> **Sık yapılan hata**
>
> Aynı döngü gövdesinde `next()`'i iki kez çağırmak: `while (it.hasNext()) { if (it.next().startsWith("A")) System.out.println(it.next()); }`. Her `next()` imleci ilerletir; ikinci çağrı aynı öğeyi değil bir *sonrakini* verir, son turda da istisna fırlatabilir. Öğe bir değişkene alınır ve gövdede o değişken kullanılır: `String s = it.next();`.

6.4.4 Gezerken silme: ConcurrentModificationException

Gezinme sırasında koşula uyan öğeleri silmek çok istenen bir iştir. En doğal görünen yazım yanlıştır:

```
for (String s : isimler)
    if (s.startsWith("A"))
        isimler.remove(s);
// ConcurrentModification-
// Exception (fail-fast)
```

Doğru yazım ya yineleyicinin kendi `remove()` metodunu ya da `removeIf`'i kullanır:

```
Iterator<String> it =
        isimler.iterator();
while (it.hasNext())
    if (it.next().startsWith("A"))
        it.remove();
// ya da kısaca:
isimler.removeIf(s -> s.startsWith("A"));
```

Yanlış yazımın neden hata verdiğini anlamak için `ArrayList`'in bir iç sayacını tanımak gerekir. Her `ArrayList`, `modCount` adlı bir alan tutar. Listede her **yapısal değişiklik**, yani öğe sayısını değiştiren her ekleme ya da silme, bu sayacı bir artırır. (`set` yapısal değildir; sayacı artırmaz.) Yineleyici oluşturulduğunda o anki sayacı `expectedModCount` adıyla saklar. Her `next()` çağrısında önce iki sayıyı karşılaştırır. Farklıysa liste, yineleyicinin haberi olmadan değiştirilmiş demektir ve yineleyici `ConcurrentModificationException` fırlatır.

Tanım 6.3

Koleksiyonun gezinme sırasında dışarıdan yapısal olarak değiştirildiğini sezip hemen istisna fırlatan yineleyiciye **hızlı başarısız** (*fail-fast*) yineleyici denir. Amaç, tutarsız bir durumda sessizce yanlış sonuç üretmek yerine hatayı ilk fırsatta görünür kılmaktır.

Sunumdaki izleme, for-each döngüsünün derleyicinin ürettiği hâlini açıkça yazar:

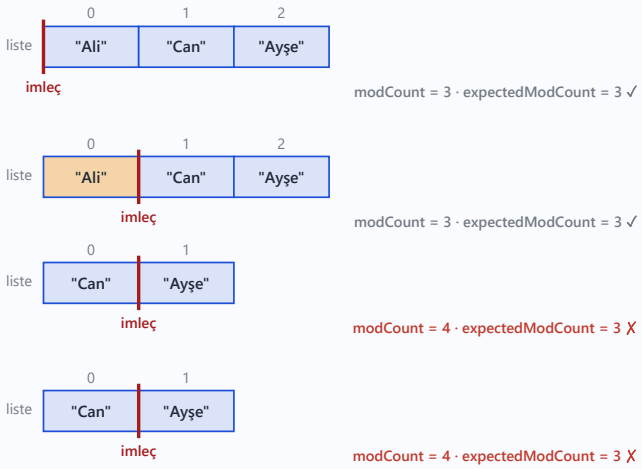
```
List<String> isimler = new ArrayList<>();
isimler.add("Ali"); isimler.add("Can");
isimler.add("Ayşe");            // modCount = 3
// for (String s : isimler) ... derleyici:
Iterator<String> it = isimler.iterator();
while (it.hasNext()) {
  String s = it.next();
  if (s.startsWith("A"))
    isimler.remove(s);          // it dışından!
}
```

Adım	Liste	`size`	`cursor`	`modCount / exp`	Not
üç `add`	[Ali, Can, Ayşe]	3	—	3 / —	her ekleme sayacı artırır
`iterator()`	[Ali, Can, Ayşe]	3	0	3 / 3	sayaç saklanır
`hasNext()`	[Ali, Can, Ayşe]	3	0	3 / 3	0 ≠ 3 → `true`
`next()`	[Ali, Can, Ayşe]	3	1	3 / 3 ✓	"Ali" döner
`isimler.remove(s)`	[Can, Ayşe]	2	1	4 / 3	yineleyici habersiz

Adım	Liste	`size`	`cursor`	`modCount / exp`	Not
`hasNext()`	[Can, Ayşe]	2	1	4 / 3	1 ≠ 2 → `true`
`next()`	[Can, Ayşe]	2	1	4 ≠ 3 ✗	`ConcurrentModificationException`

Tablo 6.16: Dışarıdan silmenin izlemesi. `exp`, yineleyicinin `expectedModCount` alanıdır.

İzlemenin beşinci satırı sorunun kökünü gösterir. “Ali” silinince “Can” ve “Ayşe” birer sola kayar; “Can” 0. indise gelir. Yineleyicinin `cursor`'ı ise hâlâ 1'dir. Denetim olmasaydı sonraki `next()` 1. indisteki “Ayşe”yi döndürür ve “Can” hiç gezilmezdi. Hızlı başarısız yineleyici bu sessiz atlamaya izin vermez: `modCount` 4, `expectedModCount` 3'tür ve istisna fırlatılır.



Şekil 6.14: (a) `iterator()`: imleç başta, iki sayaç eşit. (b) `next()` “Ali”yi döndürdü. (c) `isimler.remove(s)`: öğeler kaydı, `modCount = 4`; “Can” imlecin gerisinde kaldı. (d) `next()` sayaçların farklı olduğunu görür ve durur.

Hızlı başarısızlık bir güvencedir ama mutlak değildir. Yineleyici farkı yalnız `next()` çağrısında denetler. Silme, döngünün son turlarından birinde yapılırsa `hasNext()` yanlış dönebilir ve `next()` hiç çağrılmaz. Örneğin `[Can, Ali, Ece]` listesinde “Ali” for-each içinde dışarıdan silinirse istisna fırlatılmaz: silmeden sonra `cursor = 2`, `size = 2`'dir ve döngü sessizce biter. Liste `[Can, Ece]` olur, ama “Ece” hiç gezilmemiştir. Java belgeleri bu yüzden hızlı başarısızlığa güvenerek program yazılmamasını, onun yalnız hata ayıklamaya yardım ettiğini belirtir.

6.4.5 Kod izleme: it.remove() ile güvenli silme

```
Iterator<String> it = isimler.iterator();
while (it.hasNext()) {
  String s = it.next();
  if (s.startsWith("A"))
    it.remove();   // sayaçları eşitler
}
System.out.println(isimler);
```

Yineleyicinin `remove()` metodu iki işi birlikte yapar: öğeyi listeden siler ve kendi durumunu bu silmeye göre düzeltir. Bunun için `cursor`'ın yanında bir alan daha tutar: `lastRet`, son `next()`'in döndürdüğü öğenin indisidir. Henüz öğe döndürülmediyse ya da son döndürülen öğe zaten silindiyse `lastRet = -1`'dir.

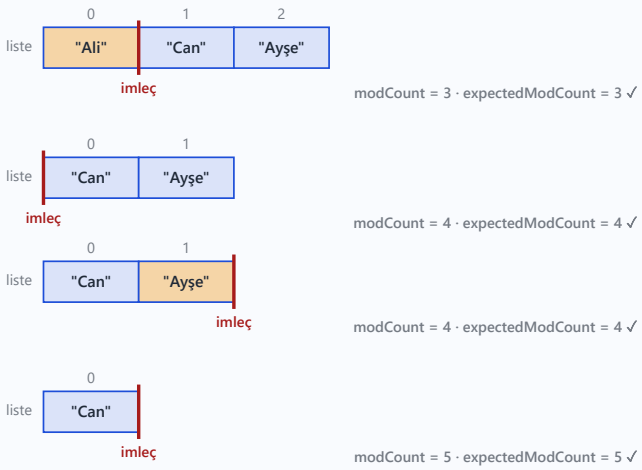
`it.remove()` çağrıldığında şunlar olur:

1. `lastRet = -1` ise silinecek öğe yoktur; `IllegalStateException` fırlatılır.
2. Liste `lastRet` indisindeki öğeyi siler; `modCount` bir artar.
3. `cursor = lastRet` yapılır. Silinen öğenin yerine kayan öğe atlanmasın diye imleç bir geri çekilir.
4. `lastRet = -1` yapılır; ikinci bir `remove()` araya `next()` girmeden yapılamaz.
5. `expectedModCount = modCount` yapılır. Değişikliği yineleyicinin kendisi yaptığı için sayaçlar eşitlenir.

Adım	Liste	`cursor`	`lastRet`	`modCount` / `exp`	Not
`iterator()`	[Ali, Can, Ayşe]	0	−1	3 / 3	—
`next()`	[Ali, Can, Ayşe]	1	0	3 / 3	“Ali”; A ile başlıyor
`it.remove()`	[Can, Ayşe]	0	−1	4 / 4	imleç geri çekildi
`next()`	[Can, Ayşe]	1	0	4 / 4	“Can”; silinmez
`next()`	[Can, Ayşe]	2	1	4 / 4	“Ayşe”; A ile başlıyor
`it.remove()`	[Can]	1	−1	5 / 5	—
`hasNext()`	[Can]	1	−1	5 / 5	1 = 1 → `false`; hata yok

Tablo 6.17: `it.remove()` ile güvenli silmenin izlemesi. Başlangıç: `[Ali, Can, Ayşe]`, `modCount = 3`.

Program `[Can]` yazdırır: A ile başlayan iki ad silinmiş, “Can” atlanmadan gezilmiştir. Üçüncü satırdaki `cursor = 0` güncellemesi kritiktir. “Ali” silinince “Can” 0. indise kaymıştır; imleç 0'a çekilmeseydi “Can” atlanırdı.



Şekil 6.15: (a) `next()` "Ali"yi döndürdü. (b) `it.remove()` : "Ali" silindi, imleç başa çekildi, sayaçlar 4'te eşit. (c) `next()` "Ayşe"yi döndürdü. (d) İkinci `it.remove()` sonrası liste `[Can]` , sayaçlar 5'te eşit.

Aynı işi tek satırda yapan `removeIf` da vardır: `isimler.removeIf(s -> s.startsWith("A"));` . `removeIf` , `ArrayList` 'te silinecek öğeleri önce işaretler, sonra kalanları tek geçişte sola sıkıştırır. Her silmede bütün kuyruğu kaydırmadığı için $O(n)$'dir; yineleyiciyle tek tek silmek ise en kötü durumda $O(n^2)$ kaydırma yapabilir.

Sık yapılan hata

`it.remove()` 'u arka arkaya iki kez çağırmak ya da hiç `next()` çağırmadan çağırmak. `remove()` "bir önceki öğeyi" silmez; yalnız son `next()` 'in döndürdüğü öğeyi siler ve bunu bir kez yapabilir. İkinci çağrıda `lastRet = -1` 'dir ve `IllegalStateException` fırlatılır.

6.4.6 ListIterator arayüzü

`Iterator` tek yönlüdür ve yalnız silmeye izin verir. Listeler için daha yetenekli bir yineleyici vardır: `ListIterator` . Yalnız `List` 'lerde bulunur ve `list.listIterator()` ile alınır. `Iterator` 'a ek olarak iki yönde gezinir ve gezinirken ekleme, silme, güncelleme yapabilir. İmleci yine öğelerin arasında durur.

Metot	Görevi
`hasNext()` / `next()`	ileride öğe var mı? / sonraki öğeyi döndür, imleci ileri al
`hasPrevious()` / `previous()`	geride öğe var mı? / önceki öğeyi döndür, imleci geri al
`nextIndex()` / `previousIndex()`	`next()` / `previous()` çağrılsa dönecek öğenin indisi (başta 0 / −1)
`add(e)`	öğeyi imlecin önüne ekler (sonraki `next()` etkilenmez)
`remove()`	son `next()` / `previous()` ile dönen öğeyi siler
`set(e)`	son `next()` / `previous()` ile dönen öğeyi değiştirir

Tablo 6.18: `ListIterator` metotları.

Tablodaki kurallar tek bir ilkeden çıkar: `remove()` ve `set(e)` "son döndürülen öğe" üzerinde çalışır, `add(e)` ise imlecin bulunduğu yere öğe koyar. `add` veya `remove`'dan hemen sonra, araya yeni bir `next` / `previous` girmeden `remove` ya da `set` çağrılırsa son döndürülen öğe artık belirsizdir ve `IllegalStateException` fırlatılır.

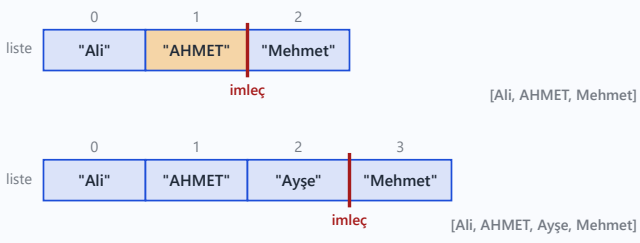
Bir dikkat noktası, `previous()`'tan sonra gelen `next()`'tir. İmleç arada durduğu için `previous()` bir öğenin üzerinden geri atlar; hemen ardından gelen `next()` *aynı* öğenin üzerinden ileri atlar ve aynı öğeyi döndürür. `listIterator(1)` ile `[A, B, C]` listesinde imleç A ile B arasında başlar; `previous()` A'yı, ardından `next()` yine A'yı döndürür.

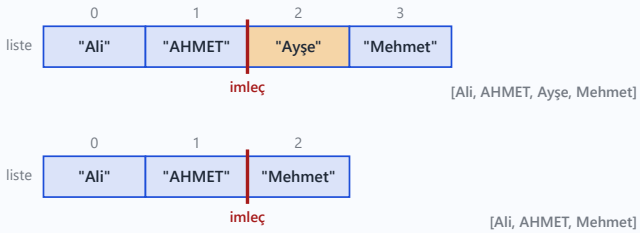
```
ListIterator<String> it = isimler.listIterator();
it.next();          // "Ali"
it.next();          // "Ahmet"
it.set("AHMET");    // [Ali, AHMET, Mehmet]
it.add("Ayşe");     // [Ali, AHMET, Ayşe, Mehmet]
it.previous();      // "Ayşe"
it.remove();        // [Ali, AHMET, Mehmet]
```

Çağrı	Liste ve imleç	Döner	`lastRet`
`listIterator()`	| Ali Ahmet Mehmet	—	−1
`next()`	Ali | Ahmet Mehmet	"Ali"	0
`next()`	Ali Ahmet | Mehmet	"Ahmet"	1
`set("AHMET")`	Ali AHMET | Mehmet	—	1
`add("Ayşe")`	Ali AHMET Ayşe | Mehmet	—	−1
`previous()`	Ali AHMET | Ayşe Mehmet	"Ayşe"	2
`remove()`	Ali AHMET | Mehmet	—	−1

Tablo 6.19: `ListIterator` izlemesi. Liste sütununda `|` imleçtir. Başlangıç `[Ali, Ahmet, Mehmet]`.

`set("AHMET")` imleci oynatmaz; yalnız son döndürülen "Ahmet"in yerine yazar. `add("Ayşe")`, imlecin önüne ekler ve imleci eklenen öğenin arkasına taşır. Bu yüzden sonraki `next()` yine "Mehmet"i döndürürdü: ekleme, ileri gezinmeyi etkilemez. `add`'den sonra `lastRet` −1 olur; o anda bir `set` çağrısı `IllegalStateException` verirdi. `previous()` "Ayşe"nin üzerinden geri atlar ve onu döndürür; `remove()` da bu son döndürülen öğeyi siler. Sonuç `[Ali, AHMET, Mehmet]`'tir ve imleç AHMET ile Mehmet arasındadır (`nextIndex() = 2`).





Şekil 6.16: Turuncu, son döndürülen öğedir. (a) `set("AHMET")` : son döndürülen öğe değişti, imleç yerinde. (b) `add("Ayşe")` : öğe imlecin önüne girdi; son döndürülen öğe yok. (c) `previous()` "Ayşe"yi döndürdü. (d) `remove()` "Ayşe"yi sildi.

`ListIterator` 'ın en değerli özelliği maliyetle ilgilidir. `LinkedList` 'te `add(i, e)` önce i. düğüme yürüdüğü için *O(n)*'dir. Oysa `ListIterator` zaten bir düğümün yanında durur; `add` , `remove` ve `set` yalnız birkaç bağlantıyı değiştirir ve *O(1)*'dir. Bir listeyi bir kez baştan sona gezip yol üstünde ekleme ve silme yapan algoritmalar `LinkedList` 'in gücünü ancak bu yolla kullanabilir.

6.4.7 Kod izleme: ileri ve geri gezinme

```
ListIterator<String> iterator =
        isimler.listIterator();
// listeyi ileri yönde gez
while (iterator.hasNext()) {
    String isim = iterator.next();
    System.out.println(isim);
}
// imleç artık sonda: ters yönde gez
while (iterator.hasPrevious()) {
    String isim = iterator.previous();
    System.out.println(isim);
}
```

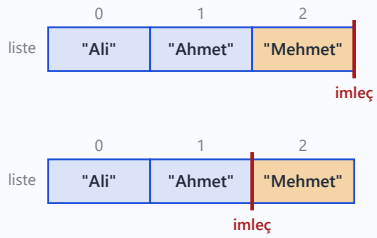
Programın çıktısı alt alta şöyledir:

```
Ali
Ahmet
Mehmet
Mehmet
Ahmet
Ali
```

İki döngü *aynı* yineleyiciyi kullanır. İlk döngü bittiğinde imleç listenin sonundadır (`cursor = 3`). İkinci döngü buradan geriye doğru `previous()` çağırır: önce "Mehmet", sonra "Ahmet", sonra "Ali" döner ve imleç başa (`cursor = 0`) gelir. `hasPrevious()` bu kez *cursor > 0* denetimini yapar ve yanlış çıkınca ikinci döngü de biter. Toplamda *3 + 3 = 6* satır yazılır.

Çağrı	Denetim	Döner / yazar	`cursor` (sonra)
`next()` ×3	0, 1, 2 < 3	Ali, Ahmet, Mehmet	1, 2, 3
`hasNext()`	3 < 3 değil	`false`	3
`previous()` ×3	3, 2, 1 > 0	Mehmet, Ahmet, Ali	2, 1, 0
`hasPrevious()`	0 > 0 değil	`false`	0

Tablo 6.20: İleri ve geri gezinmede `cursor`'ın değişimi.



Şekil 6.17: (a) İleri gezinmenin sonu: "Mehmet" döndürüldü, imleç sonda. (b) Geri gezinmenin ilk adımı: `previous()` yine "Mehmet"i döndürür ve imleç bir geri gelir.

Listeyi doğrudan sondan başlayarak gezmek gerekiyorsa ilk döngüye gerek yoktur. `isimler.listIterator(isimler.size())`, imleci baştan son öğenin arkasına koyar; ardından yalnız `previous()` döngüsü yazılır.

6.5 ArrayList

Şimdiye kadar `List` arayüzünün ne yaptığı anlatıldı. Bu kesim ile sonraki iki kesim, arayüzün arkasındaki sınıfların bunu *nasıl* yaptığını açar. İlk durak, Java programlarında en çok kullanılan liste sınıfı olan `ArrayList`'tir.

6.5.1 List arayüzünü uygulayan sınıflar

Java'nın standart kütüphanesinde `List`'i uygulayan altı yapı öne çıkar. Hepsi aynı metotları sunar; iç yapıları ve dolayısıyla güçlü ve zayıf yanları farklıdır.

`ArrayList`
: Büyüyebilen dizi. İndisle erişim hızlıdır; genel amaçlı varsayılan seçimdir.

`LinkedList`
: Çift yönlü bağlı liste. Uçlarda ekleme ve silme hızlıdır; indisle erişim yavaştır.

`Vector`
: `ArrayList`'in eski, metotları senkronize (`synchronized`) sürümü.

`Stack`

`Vector`'dan türeyen yığın sınıfı: `push`, `pop`, `peek` (son giren ilk çıkar).

`CopyOnWriteArrayList`

Her yazmada iç diziyi kopyalayan, çok iş parçacıklı okumaya uygun liste.

`Arrays.asList()`

Bir diziyi sabit boyutlu bir `List` görünümüne çeviren yardımcı metot.

6.5.2 Büyüyebilen dizi

Java dizilerinin boyu oluşturulurken belirlenir ve bir daha değişmez. Oysa bir listenin kaç öğe tutacağı çoğu zaman önceden bilinmez. `ArrayList` bu çelişkiyi şöyle çözer: öğeleri içeride sıradan bir dizide tutar, ama bu dizinin boyunu kullanıcıdan gizler. Dizinin boyuna **kapasite** (*capacity*), içindeki gerçek öğe sayısına **boyut** (`size`) denir. Her zaman *kapasite ≥ boyut*'tur; aradaki hücreler gelecekteki eklemeler için ayrılmış boş yerlerdir.

Dizi dolunca, yani boyut kapasiteye eşitken bir ekleme daha gelirse, `ArrayList` yaklaşık 1,5 kat büyük yeni bir dizi oluşturur, bütün öğeleri ona kopyalar ve eski diziyi bırakır. Bu kopyalama *O(n)* sürer. Ancak seyrek olur: büyütmeden sonra uzun bir süre yeni ekleme yapılabilir. Bu yüzden sona ekleme **amortize *O(1)***'dir.

Silme sırasında `ArrayList` kendiliğinden küçülmez. Bir milyon öğe eklenip sonra silinirse kapasite bir milyonda kalır. Bellek geri istenirse `trimToSize()` çağrılır; kapasite o anki boyuta indirilir.

Gerçek Java'da iki ayrıntı vardır. Birincisi, `new ArrayList<>()` başlangıçta hiç hücre ayırmaz; ilk eklemede 10 kapasiteli bir dizi oluşturur. İkincisi, yeni kapasite *eski + eski/2* formülüyle hesaplanır; kodda bu `eski + (eski >> 1)` diye yazılır, çünkü bir bit sağa kaydırmak 2'ye tam bölmeyle aynıdır. Sunumdaki izleme, adımların az olması için kapasite 4'ten başlar.

6.5.3 Kod izleme: add(e) ve büyüme

`ArrayList`'in sona ekleme metodunun özü şu birkaç satırdır:

```
public boolean add(E e) {
  if (boyut == dizi.length) {
    int yeni = Math.max(dizi.length + 1, dizi.length + dizi.length / 2);
    dizi = Arrays.copyOf(dizi, yeni);
  }
  dizi[boyut++] = e;
  return true;
}
```

Metot önce dizinin dolu olup olmadığını sorar: `boyut == dizi.length`. Doluysa yeni kapasiteyi hesaplar (`Math.max`, kapasite 0 ya da 1 iken de en az bir hücre eklenmesini sağlar; yoksa *1 + 1/2 = 1* olur ve dizi büyümez) ve `Arrays.copyOf` ile o kapasitede yeni bir dizi oluşturur. `Arrays.copyOf`,

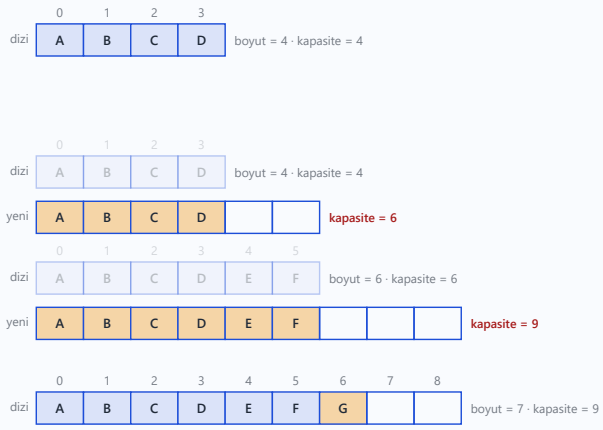
eski dizinin bütün öğelerini yeni dizinin başına kopyalar ve kalan hücreleri `null` bırakır. Sonra `dizi` referansı yeni diziye yöneltilir; eski diziyi hiçbir şey göstermediği için çöp toplayıcı onu temizler. Son olarak `dizi[boyut++] = e` satırı öğeyi ilk boş hücreye yazar ve boyutu bir artırır.

Kapasitesi 4, boyutu 3 olan `[A, B, C]` listesine sırayla D, E, F, G ekleyelim:

Çağrı	Dolu mu?	Büyütme	Kopya	Boyut	Kapasite
`add(D)`	3 == 4? hayır	—	0	4	4
`add(E)`	4 == 4? **evet**	*4 + 4/2 = 6*	4	5	6
`add(F)`	5 == 6? hayır	—	0	6	6
`add(G)`	6 == 6? **evet**	*6 + 6/2 = 9*	6	7	9

Tablo 6.21: `add(e)` ve büyüme izlemesi. Başlangıç: `[A, B, C]`, kapasite 4.

Dört eklemenin ikisi ucuzdur: yalnız bir hücreye yazar. İkisi pahalıdır: önce bütün öğeleri kopyalar. Toplam *4 + 6 = 10* kopya yapılmıştır. Büyütmenin kapasiteyi oranla artırdığına dikkat edelim: 4'ten 6'ya, 6'dan 9'a. Bir sonraki büyütme 9'dan 13'e, sonraki 13'ten 19'a olur. Kapasite büyüdükçe iki büyütme arasındaki ucuz ekleme sayısı da büyür; büyütmeler giderek seyrekleşir.



Şekil 6.18: (a) `add(E)` geldiğinde dizi dolu: boyut 4, kapasite 4. (b) 6 hücreli yeni dizi ve 4 öğenin kopyası; eski dizi soluk. (c) `add(G)` 'de ikinci büyütme: 9 hücreli dizi, 6 kopya. (d) G yazıldı: boyut 7, kapasite 9.

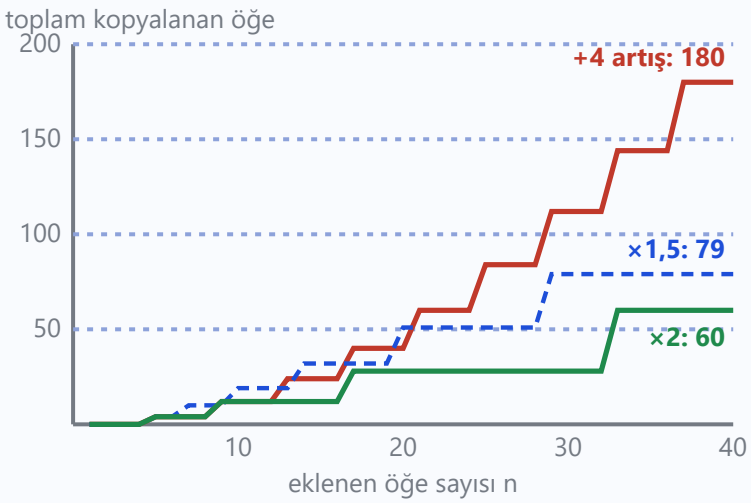
6.5.4 Amortize analiz: neden katlayarak büyütülür?

Büyütme yapan bir ekleme *O(n)* sürer. O hâlde sona ekleme neden *O(1)* sayılır? Cevap, tek bir işleme değil uzun bir işlem dizisine bakmaktır.

Tanım 6.4

Bir işlemin **amortize maliyeti**, uzun bir işlem dizisinin toplam maliyetinin işlem sayısına bölümüdür. Bu ortalama olasılığa dayanmaz; işlem dizisi nasıl seçilirse seçilsin geçerli olan bir üst sınırdır. Böyle bir sınırı bulmaya **amortize analiz** (*amortized analysis*) denir.

Kapasitenin nasıl büyütüldüğü bu analizin sonucunu tamamen belirler. Üç seçeneği karşılaştıralım: kapasiteyi her dolumda iki katına çıkarmak, 1,5 katına çıkarmak ve sabit bir miktar (örneğin 4) artırmak. Şekil 6.19, kapasite 4'ten başlayan bir dinamik dizide 40 ekleme boyunca yapılan toplam kopyaları gerçekten sayarak çizer.



Şekil 6.19: Kapasite 4'ten başlayan dinamik dizide 40 eklemede biriken kopya sayısı. İki katına büyütme 60, 1,5 katına büyütme 79, 4'er artırma 180 kopya yapar. Sabit artışın eğrisi giderek dikleşir.

Grafikteki sayılar elle de bulunabilir.

Örnek 6.3

Kapasite 4'ten başlayan boş bir dinamik diziye 40 öğe ekleniyor. Üç büyütme kuralı için toplam kopya sayısı nedir?

1. **İki katına (×2).** Kapasite 4 → 8 → 16 → 32 → 64 diye büyür. Büyütmeler dizi 4, 8, 16 ve 32 öğeyle dolunca yapılır. Kopya: $4 + 8 + 16 + 32 = 60$.
2. **1,5 katına (×1,5).** Kapasite 4 → 6 → 9 → 13 → 19 → 28 → 42 diye büyür (her adımda *eski + eski/2*, tam sayı bölmesiyle). Kopya: $4 + 6 + 9 + 13 + 19 + 28 = 79$.
3. **Sabit artış (+4).** Kapasite 4 → 8 → 12 → … → 40 diye büyür; büyütmeler 4, 8, …, 36 öğede yapılır. Kopya: $4 + 8 + \ldots + 36 = 4 \cdot (1 + 2 + \ldots + 9) = 4 \cdot 45 = 180$.

40 öğe için fark üç kattır; öğe sayısı arttıkça fark çok daha büyür. Nedenini genel bir hesap gösterir.

Teorem 6.1 (Geometrik büyütmenin amortize maliyeti)

Kapasitesi her dolumda 2 katına çıkarılan bir dinamik diziye n öğe eklemek, toplamda $2n$'den az kopya yapar. Kapasite 1,5 katına çıkarılıyorsa toplam kopya yaklaşık $3n$'yi geçmez. İki durumda da sona eklemenin amortize maliyeti $O(1)$'dir.

İspat. İki katına büyütmede kopya sayıları, her biri öncekinin iki katı olan bir dizi oluşturur: $c, 2c, 4c, \ldots, K$. Burada K son büyütmedeki kopya sayısıdır ve o anda dizide K öğe bulunduğu için $K < n$'dir. Sondan başa doğru okunduğunda terimler $K, K/2, K/4, \ldots$ diye yarılanır. Böyle bir toplam $K \cdot (1 + 1/2 + 1/4 + \ldots) < 2K < 2n$'dir.

1,5 katına büyütmede her terim bir sonrakinin yaklaşık $2/3$'üdür. Sondan başa toplam $K \cdot (1 + 2/3 + 4/9 + \ldots) = K \cdot 1/(1 - 2/3) = 3K$ civarındadır ve $K < n$ olduğundan yaklaşık $3n$'yi geçmez. Tam sayı bölmesi oranı $2/3$'ten biraz saptırır; bu sapma yalnız sabiti çok az değiştirir, doğrusal sınırı bozmaz. Örneğin Java'nın 10 kapasiteyle başlayan büyütmesinde ilk üç milyon eklemenin hiçbirinde toplam kopya $3n$'ye ulaşmaz.

Her eklemede ayrıca bir yazma yapılır, toplam n yazma. Kopyalar ve yazmalar birlikte iki katına büyütmede $3n$'den az, 1,5 katına büyütmede yaklaşık $4n$'dir. İşlem başına düşen maliyet sabittir. □

Sabit artışta tablo tersine döner. Kapasite c ile başlayıp her dolumda c artırılsın. Büyütmeler $c, 2c, 3c, \ldots$ öğede yapılır ve kopyalar

$$c + 2c + 3c + \ldots + (n/c)\cdot c \approx c \cdot (n/c)^2/2 = n^2/(2c)$$

kadar tutar. Toplam $O(n^2)$, işlem başına $O(n)$'dir. c'yi büyütmek yalnız sabiti küçültür; büyüme hızı karesel kalır. Örneğin kapasite 10'dan başlayıp 10'ar artırılan bir diziye 100 öğe eklemek $10 + 20 + \ldots + 90 = 450$ kopya yapar; aynı diziyi 1,5 katına büyütmek 202, iki katına büyütmek 150 kopya yapar.

Amortize maliyet bir benzetmeyle de açıklanabilir. Goodrich ve Tamassia bunu "bankacı yöntemi" diye anlatır. Her ucuz eklemeye gerçek maliyetinden birkaç birim fazla ödetildiğini, fazlanın bir kumbarada "jeton" olarak biriktirildiğini düşünelim. İki katına büyütmede, iki büyütme arasında dizinin yarısı kadar ucuz ekleme yapılır. Her biri 2 jeton biriktirirse, büyütme anında kumbarada dizideki bütün öğeleri kopyalamaya yetecek kadar jeton bulunur. Hiçbir işlem kumbarayı eksiye düşürmediği için her eklemenin sabit sayıda jetonla ödendiği söylenebilir.

Sık yapılan hata

"Amortize $O(1)$" ifadesini "her ekleme $O(1)$" diye okumak. Büyütme yapan ekleme yine $O(n)$ sürer. Her işlemin kısa sürmesi gereken gerçek zamanlı bir sistemde bu duraklama sorun olabilir; öğe sayısı biliniyorsa liste baştan yeterli kapasiteyle oluşturulur: `new ArrayList<>`

`(10000)`.

6.5.5 ArrayList kullanımı ve maliyetleri

```
ArrayList<String> liste =
        new ArrayList<>();
liste.add("Ali");     // [Ali]
liste.add("Veli");    // [Ali, Veli]
liste.add(0, "Can");  // [Can, Ali, Veli]
liste.get(1);         // "Ali"
liste.remove(0);      // [Ali, Veli]
liste.size();         // 2
```

İşlem	Maliyet	Gerekçe
`get(i)`, `set(i, e)`	*O(1)*	adres hesabıyla doğrudan erişim
`add(e)` (sona)	amortize *O(1)*	büyütme seyrek
`add(i, e)`, `remove(i)`	*O(n)*	sonraki öğeler kayar
`contains`, `indexOf`	*O(n)*	doğrusal arama
`size()`	*O(1)*	sayaç alanı okunur

Tablo 6.22: `ArrayList` işlemlerinin maliyeti.

Sunumdaki izleme bu kodu adım adım çalıştırır ve her işlemde kaç öğenin kaydığını sayar:

```
ArrayList<String> liste = new ArrayList<>();
liste.add("Ali");
liste.add("Veli");
liste.add(0, "Can");
String s = liste.get(1);
liste.remove(0);
int n = liste.size();
```

Satır	Liste	Kapasite	Kaydırma	Maliyet
`new ArrayList<>()`	[ ]	0	—	*O(1)*
`add("Ali")`	[Ali]	10	0	ilk dizi ayrılır
`add("Veli")`	[Ali, Veli]	10	0	amortize *O(1)*
`add(0, "Can")`	[Can, Ali, Veli]	10	2 (sağa)	*O(n)*
`get(1)`	[Can, Ali, Veli]	10	0	“Ali”, *O(1)*
`remove(0)`	[Ali, Veli]	10	2 (sola)	*O(n)*
`size()`	[Ali, Veli]	10	0	2, *O(1)*

Tablo 6.23: `ArrayList` kullanım izlemesi.

Toplam 4 kaydırmanın hepsi başı değiştiren iki işlemden gelir: başa ekleme ve baştan silme. Dizi tabanlı bir listede başı değiştiren her işlem bütün listeyi kaydırır. Sona ekleme, indisle okuma ve boyut sorgusu ise kaydırma yapmaz. `remove(0)` 'dan sonra kapasite hâlâ 10'dur: `ArrayList` kendiliğinden küçülmez.

`ArrayList` 'i sıradan bir diziyle karşılaştırmak da öğreticidir. Artıları: boyut kendiliğinden büyür; ekleme, silme, arama ve sıralama hazır metotlarla yapılır. Eksileri: yalnız nesne tutabilir, bu yüzden ilkel değerler kutulanır; her erişimde bir metot çağrısı ve sınır denetimi yapılır. Sabit boyutlu, ilkel türden büyük veri için dizi daha uygundur.

6.5.6 Kod izleme: add(i, e)

Araya ekleme, dizinin ortasında bir yer açmayı gerektirir. Kod şöyledir:

```
public void add(int i, E e) {
  Objects.checkIndex(i, boyut + 1);
  if (boyut == dizi.length) buyut();
  for (int k = boyut; k > i; k--)
    dizi[k] = dizi[k - 1];
  dizi[i] = e;
  boyut++;
}
```

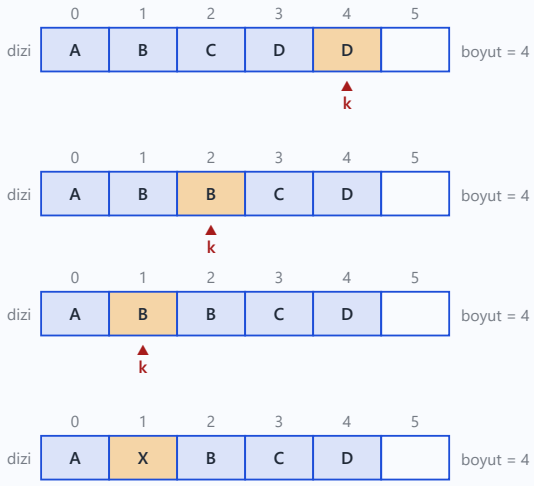
`Objects.checkIndex(i, boyut + 1)` , $0 \le i < boyut + 1$ koşulunu, yani $0 \le i \le boyut$'u denetler; değilse `IndexOutOfBoundsException` fırlatır. Sona eklemeye izin vermek için üst sınır `boyut + 1` 'dir. Dizi doluysa önce büyütülür. Ardından `for` döngüsü, `k = boyut` 'tan başlayıp `k = i + 1` 'e kadar geriye giderek her öğeyi bir sağa kopyalar. Son olarak boşalan `dizi[i]` 'ye yeni öğe yazılır ve boyut artar.

Döngünün *sağdan* başlaması zorunludur. Soldan başlansaydı `dizi[i + 1] = dizi[i]` ilk adımda `dizi[i + 1]` 'deki öğenin üzerine yazar ve o öğe kaybolurdu. Sağdan başlanınca her öğe, üzerine yazılmadan önce bir sağa taşınmış olur.

Adım	Denetim / atama	`dizi` (0–5)	`boyut`
başlangıç	—	[A, B, C, D, –, –]	4
indis denetimi	$0 \le 1 \le 4$ ✓	[A, B, C, D, –, –]	4
doluluk	$4 < 6$: büyütme yok	[A, B, C, D, –, –]	4
`k = 4`	`dizi[4] = dizi[3]`	[A, B, C, D, D, –]	4
`k = 3`	`dizi[3] = dizi[2]`	[A, B, C, C, D, –]	4
`k = 2`	`dizi[2] = dizi[1]`	[A, B, B, C, D, –]	4
`k = 1`	$1 > 1$ değil: döngü biter	[A, B, B, C, D, –]	4
`dizi[1] = e`	X yazılır	[A, X, B, C, D, –]	4
`boyut++`	—	[A, X, B, C, D, –]	5

Tablo 6.24: `[A, B, C, D]` (kapasite 6) üzerinde `add(1, "X")` izlemesi.

Döngü bittiğinde 1. hücrede B'nin eski bir kopyası durur; bu bir hata değildir, çünkü bir sonraki satır onun üzerine X yazar. Toplam $n - i = 4 - 1 = 3$ kaydırma yapılmıştır. Başa eklemede ($i = 0$) bütün n öğe kayar; sona eklemede ($i = n$) hiç kayma olmaz. En kötü durum $O(n)$'dir.



Şekil 6.20: (a) `k = 4` : D sağa kopyalandı. (b) `k = 2` : B sağa kopyalandı. (c) Döngü bitti; 1. hücrede B'nin eski kopyası. (d) X yazıldı: `[A, X, B, C, D]` .

6.5.7 Kod izleme: remove(i)

```
public E remove(int i) {
  Objects.checkIndex(i, boyut);
  E eski = dizi[i];
  for (int k = i; k < boyut - 1; k++)
    dizi[k] = dizi[k + 1];
  dizi[--boyut] = null;
  return eski;
}
```

Silme, eklemenin tersidir. İndis bu kez $0 \le i < boyut$ aralığında olmalıdır, çünkü var olan bir öğe silinir. Silinecek öğe `eski` 'de saklanır. Döngü `k = i` 'den başlayıp *soldan sağa* ilerler ve her öğeyi bir sola kopyalar. Bu kez soldan başlamak doğrudur: `dizi[i]` zaten silinecek öğedir, üzerine yazılması istenir.

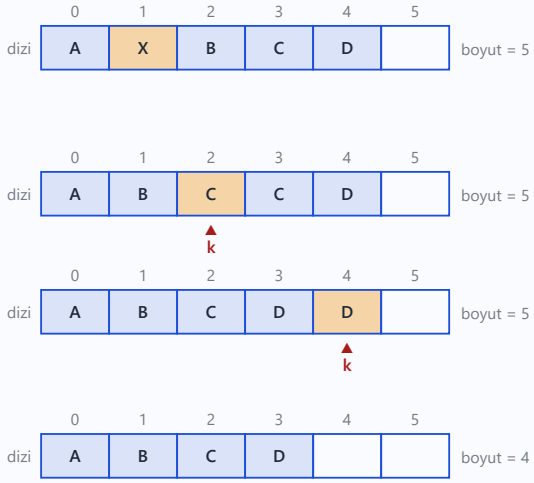
Adım	Denetim / atama	`dizi` (0–5)	`boyut`
indis denetimi	$0 \le 1 < 5$ ✓	[A, X, B, C, D, –]	5
`eski = dizi[1]`	`eski = X`	[A, X, B, C, D, –]	5
`k = 1`	`dizi[1] = dizi[2]`	[A, B, B, C, D, –]	5
`k = 2`	`dizi[2] = dizi[3]`	[A, B, C, C, D, –]	5

Adım	Denetim / atama	`dizi` (0–5)	`boyut`
`k = 3`	`dizi[3] = dizi[4]`	[A, B, C, D, D, –]	5
`k = 4`	*4 < 4* değil: döngü biter	[A, B, C, D, D, –]	5
`dizi[--boyut] = null`	boyut 4, `dizi[4] = null`	[A, B, C, D, –, –]	4
`return eski`	X döner	[A, B, C, D, –, –]	4

Tablo 6.25: `[A, X, B, C, D]` üzerinde `remove(1)` izlemesi.

Döngü bittiğinde son iki hücrede D iki kez durur. `dizi[--boyut] = null` satırı önce boyutu 4'e indirir, sonra 4. hücreye `null` yazar. Bu atama gereksiz görünebilir, çünkü 4. hücre artık listenin dışındadır. Ama orada kalan referans, çöp toplayıcının D nesnesini temizlemesini engellerdi. Liste D'yi sonradan gerçekten silse bile bu "hayalet" referans nesneyi bellekte tutmaya devam ederdi. Bu tür hatalara **bellek sızıntısı** denir; `null` ataması onu önler.

Kaydırma sayısı *n − i − 1 = 5 − 1 − 1 = 3*'tür. Baştan silmede *n − 1* öğe kayar, sondan silmede (*i = boyut − 1*) hiç kayma olmaz ve işlem *O(1)*'dir. Kapasite yine değişmez.



Şekil 6.21: (a) `eski = X`. (b) `k = 2`: C sola kaydı. (c) Döngü bitti; son iki hücrede D. (d) `dizi[4] = null`, boyut 4.

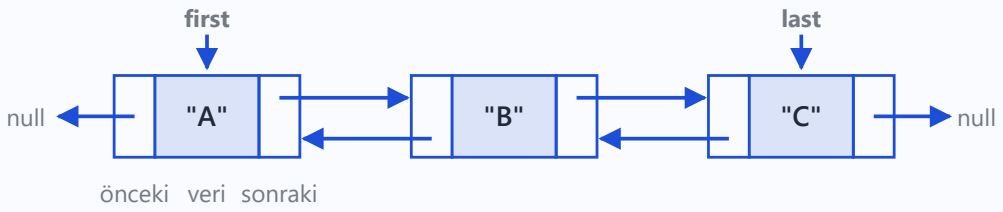
Sık yapılan hata

Araya eklemede kaydırma döngüsünü soldan, silmede sağdan başlatmak. Yön ters seçilirse her adım bir sonraki öğenin üzerine yazar ve listenin bir bölümü tek bir öğenin kopyalarıyla dolar. Kural basittir: öğeler sağa kayacaksa sağdan, sola kayacaksa soldan başlanır. Java'nın kendi kodu bu işi `System.arraycopy` ile yapar; bu metot kaynak ve hedef çakışsa bile doğru sonucu verir.

6.6 LinkedList ve konumsal liste

Dizi tabanlı listenin zayıf yanı başı değiştiren işlemlerdir: başa ekleme ya da baştan silme bütün öğeleri kaydırır. Bağlı liste bu sorunu öğeleri bellekte ardışık tutmayarak çözer. Bu kesim Java'nın `LinkedList` sınıfını, onun iki temel işleminin kodunu ve bağlı listenin gücünü tam kullanan konumsal liste ADT'sini anlatır.

6.6.1 Çift yönlü bağlı liste



Şekil 6.22: Üç öğeli bir `LinkedList`. Her düğümün üç alanı vardır: önceki düğüme referans, veri ve sonraki düğüme referans. Liste nesnesi ilk düğümü `first`, son düğümü `last` ile tutar; uçlardaki boş referanslar `null`'dır.

`LinkedList`'in yapısı Şekil 6.22'de görülüyor. Öğeler bellekte yan yana durmaz; her öğe ayrı bir **düğüm** (*node*) nesnesinin içindedir. Her düğüm veriyi ve iki referansı tutar: önceki düğüm (`onceki`) ve sonraki düğüm (`sonraki`). İki yönde bağlantı olduğu için bu yapıya **çift yönlü bağlı liste** (*doubly linked list*) denir. Liste nesnesinin kendisi yalnız ilk düğümü (`first`), son düğümü (`last`) ve öğe sayısını (`size`) bilir.

Bu yapının iki sonucu vardır. Birincisi, uçlarda iş çok ucuzdur. Başa ekleme yeni bir düğüm oluşturup `first`'ü ona yöneltmekten ibarettir; hiçbir öğe kaymaz. Sona ekleme de `last` sayesinde listeyi gezmeden yapılır. İkincisi, rastgele erişim pahalıdır. 500. öğeye ulaşmak için bir adres hesabı yoktur; düğümler tek tek, bağlantı bağlantı yürünür. Java bu yürüyüşü yakın uçtan başlatarak kısaltır, ama maliyet yine *O(n)*'dir.

`LinkedList`, `List`'in yanında `Deque` (çift uçlu kuyruk) arayüzünü de uygular. Bu yüzden `addFirst`, `addLast`, `removeFirst`, `removeLast`, `push`, `pop`, `offer`, `poll` gibi metotları da vardır ve hem yığın hem kuyruk olarak kullanılabilir.

6.6.2 LinkedList kullanımı ve maliyetleri

```
LinkedList<String> liste =
        new LinkedList<>();
liste.add("B");         // [B]
liste.addFirst("A");    // [A, B]
liste.addLast("C");     // [A, B, C]
liste.get(1);           // "B" (yürüyerek)
liste.removeFirst();    // [B, C]
liste.size();           // 2
```

İşlem	Maliyet
başa / sona ekleme, baştan / sondan silme	*O(1)*
`get(i)` , `set(i, e)`	*O(n)*
`add(i, e)` , `remove(i)`	*O(n)**
yineleyiciyle ekleme / silme	*O(1)*

Tablo 6.26: `LinkedList` işlemlerinin maliyeti.

Tablodaki yıldız önemli bir ayrıntıyı işaret eder. Bağlı listede bir düğümü araya bağlamak ya da aradan çıkarmak birkaç referans ataması gerektirir ve *O(1)*'dir. Ama `add(i, e)` ve `remove(i)` önce i. düğümü *bulmak* zorundadır ve bu yürüyüş *O(n)* sürer. "Bağlı listede ekleme hızlıdır" sözü ancak konum zaten elimizdeyse doğrudur: liste uçlarında ya da bir `ListIterator` 'ın durduğu yerde.

Sunumdaki izleme yukarıdaki kodu adım adım çalıştırır:

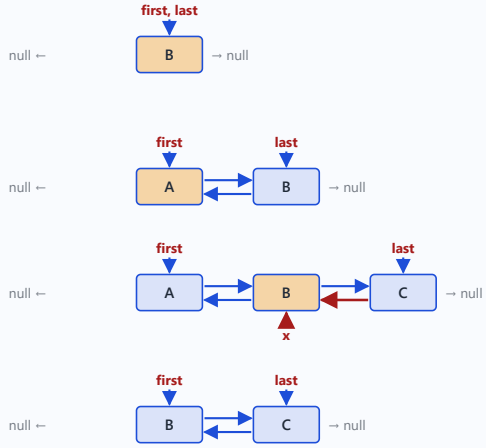
```
LinkedList<String> liste = new LinkedList<>();
liste.add("B");
liste.addFirst("A");
liste.addLast("C");
String s = liste.get(1);
liste.removeFirst();
int n = liste.size();
```

Satır	Liste	`first` / `last`	İş	Maliyet
`new LinkedList<>()`	[ ]	`null` / `null`	—	*O(1)*
`add("B")`	[B]	B / B	`add` = `addLast`	*O(1)*
`addFirst("A")`	[A, B]	A / B	`A.sonraki = B` , `B.onceki = A`	*O(1)*
`addLast("C")`	[A, B, C]	A / C	`last` sayesinde yürüme yok	*O(1)*
`get(1)`	[A, B, C]	A / C	*1 < 3/2 = 1* değil: `last` 'tan 1 geri	*O(n)*
`removeFirst()`	[B, C]	B / C	`B.onceki = null`	*O(1)*

Satır	Liste	`first / last`	İş	Maliyet
`size()`	[B, C]	B / C	sayaç alanı	*O(1)*

Tablo 6.27: `LinkedList` kullanım izlemesi.

`get(1)` satırında Java'nın karar kuralı iş başındadır. Liste 3 öğelidir ve `size >> 1`, yani *3/2*'nin tam sayı kısmı 1'dir. *1 < 1* yanlış olduğu için yürüyüş sondan başlar: `x = last` (C), ardından bir kez `x = x.onceki` ile B'ye gelinir. `removeFirst()` ise yalnız iki referansı değiştirir: `first` B'yi gösterir, B'nin `onceki` alanı `null` olur. A'yı artık hiçbir referans göstermez; çöp toplayıcı onu temizler.



Şekil 6.23: (a) `add("B")` : tek düğüm hem ilk hem son. (b) `addFirst("A")` : yeni düğüm başa bağlandı. (c) `get(1)` : `last` 'tan bir bağlantı geri yürüyüş. (d) `removeFirst()` sonrası `[B, C]` .

6.6.3 Kod izleme: addLast(e)

Sona ekleme, Java'daki `linkLast` metoduyla aynı mantıkla şöyle yazılır:

```
void addLast(E e) {
  Dugum<E> l = last;
  Dugum<E> d = new Dugum<>(l, e, null);
  last = d;
  if (l == null) first = d;
  else l.sonraki = d;
  size++;
}
```

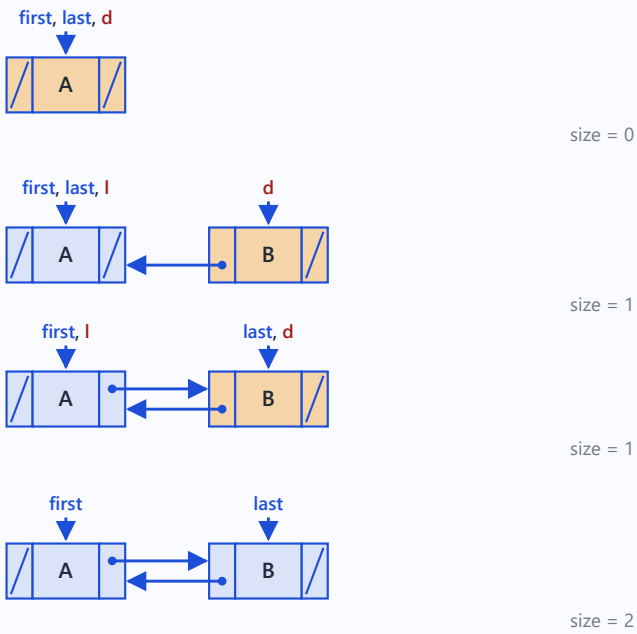
Kod beş adımdan oluşur. Önce eski son düğüm `l` 'de saklanır. Sonra yeni düğüm oluşturulur; yapıcıya verilen üç argüman sırasıyla önceki düğüm (`l`), veri (`e`) ve sonraki düğümdür (`null` , çünkü yeni düğüm en sonda olacak). Böylece yeni düğümün geri bağlantısı oluşturulurken hazır kurulur. Üçüncü

adımda `last` yeni düğüme yöneltilir. Dördüncü adım iki durumu ayırır. Liste boştuysa (`l == null`) yeni düğüm aynı zamanda ilk düğümdür; `first` da ona yöneltilir. Liste boş değilse eski son düğümün ileri bağlantısı yeni düğüme bağlanır. Son olarak `size` artar.

Satır	1. çağrı (“A”)	2. çağrı (“B”)
`l = last`	`l = null`	`l` = düğüm(A)
`d = new Dugum<>(l, e, null)`	d: önceki `null` , sonraki `null`	d: önceki düğüm(A), sonraki `null`
`last = d`	`last` = düğüm(A)	`last` = düğüm(B)
`if (l == null)`	doğru: `first` = düğüm(A)	yanlış: `A.sonraki` = düğüm(B)
`size++`	1	2

Tablo 6.28: Boş listeye `addLast("A")` ve `addLast("B")` .

İzlemenin sonunda liste `first → A ⇄ B ← last` biçimindedir. Hiçbir adımda liste gezilmemiştir; işlem sayısı listenin uzunluğundan bağımsızdır. Sona ekleme *O(1)*'dir. Bu sonuç `last` referansına bağlıdır: liste yalnız `first` 'ü tutsaydı son düğümü bulmak için baştan sona yürümek gerekir ve sona ekleme *O(n)* olurdu.



Şekil 6.24: (a) İlk çağrı: tek düğüm hem `first` hem `last` . (b) İkinci çağrıda yeni düğüm oluşturuldu; geri bağlantısı A'yı gösteriyor. (c) `l.sonraki = d` : A'dan B'ye ileri bağlantı. (d) Sonuç: iki düğümlü liste, `size = 2` .

Sık yapılan hata

Boş liste durumunu unutmak. `if (l == null) first = d;` satırı yazılmazsa ilk eklemede `l.sonraki = d` çalışır ve `null` referans üzerinden alan değiştirilmeye çalışıldığı için `NullPointerException` fırlatılır. Bağlı liste kodlarında boş liste, tek öğeli liste ve uçlardaki

düğüm her zaman ayrıca sınanmalıdır.

6.6.4 Kod izleme: remove(i)

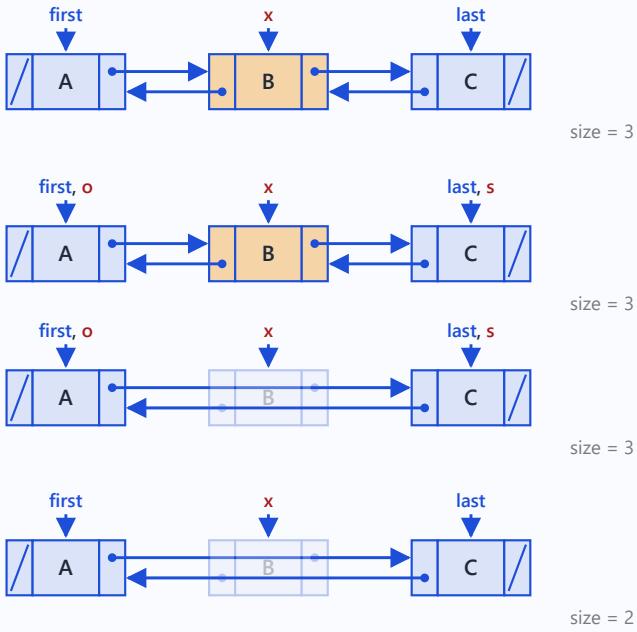
```
E remove(int i) {
  Objects.checkIndex(i, size);
  Dugum<E> x = first;
  for (int k = 0; k < i; k++) x = x.sonraki;
  Dugum<E> o = x.onceki, s = x.sonraki;
  if (o == null) first = s; else o.sonraki = s;
  if (s == null) last = o; else s.onceki = o;
  size--;
  return x.veri;
}
```

İndisle silme iki aşamalıdır. İlk aşama aranan düğümü bulur: `x` ilk düğümden başlar ve `i` kez `sonraki` bağlantısını izler. İkinci aşama düğümü listeden ayırır. Düğümün iki komşusu `o` (önceki) ve `s` (sonraki) adlarıyla saklanır. Önceki düğümün ileri bağlantısı `s`'ye, sonraki düğümün geri bağlantısı `o`'ya yöneltilir. Böylece `x` iki yönden de atlanır. Uç durumlar da kodda vardır: `x` ilk düğümse (`o == null`) `first` güncellenir, son düğümse (`s == null`) `last` güncellenir.

Satır	Durum	Not
`checkIndex(1, 3)`	$0 \le 1 < 3$ ✓	—
`x = first`	`x` = A	dizideki gibi doğrudan erişim yok
`k = 0`	`x = x.sonraki` → B	1 bağlantı yüründü
`k = 1`	$1 < 1$ değil	yürüme biter: $O(i)$
`o = x.onceki`, `s = x.sonraki`	`o` = A, `s` = C	iki komşu
`o.sonraki = s`	A → C	`o` `null` değil
`s.onceki = o`	C → A	`s` `null` değil; B koptu
`size--`	`size = 2`	—
`return x.veri`	"B" döner	liste `[A, C]`

Tablo 6.29: `[A, B, C]` üzerinde `remove(1)` izlemesi.

Bağlantıları değiştiren dört satır $O(1)$'dir, ama önlerindeki yürüyüş $O(i)$'dir. En kötü durumda toplam $O(n)$ olur. Sunumdaki kod sadeliği korumak için hep baştan yürür; Java'nın gerçek kodu $i \ge size/2$ ise `last`'tan geriye yürür ve yürüyüşü en fazla $n/2$ bağlantıya indirir. Bu da büyüme hızını değiştirmez.



Şekil 6.25: (a) `x` bir bağlantı yürüyerek B'ye geldi. (b) Komşular `o` = A, `s` = C. (c) İki bağlantı da B'yi atlıyor; B soluk. (d) `size = 2`; B döndürüldü.

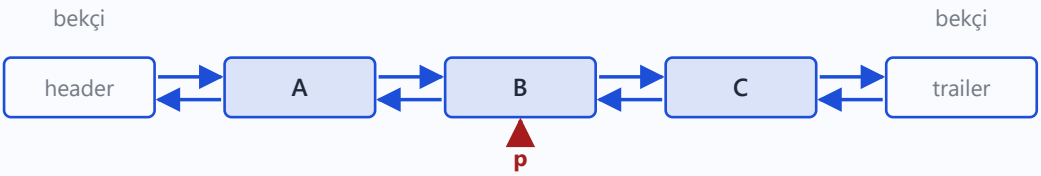
6.6.5 Konumsal liste ADT'si

İndis, dizi tabanlı listeye uygun bir adreslemedir; bağlı listeye pek uymaz. Üstelik indisler kararsızdır: listenin başına bir öğe eklendiğinde bütün öğelerin indisleri bir artar. "5. öğe" diye tutulan bir referans, bir ekleme sonrasında başka bir öğeyi gösterir hâle gelir. Metin düzenleyicideki imleci düşünelim: imlecin önüne yazı eklense de imleç bulunduğu kelimenin yanında kalır. İmleç bir sayı değil, metindeki bir *yerdir*.

Goodrich, Tamassia ve Goldwasser bu fikri bir ADT'ye dönüştürür.

Tanım 6.5

Konum (*position*), bir listede bir öğenin bulunduğu yeri temsil eden ve o öğe silinene kadar geçerli kalan nesnedir; ekleme ve silmelerle değişmez. **Konumsal liste** (*positional list*), öğelere indislerle değil konumlarla erişilen ve konum verilerek ekleme, silme ve güncelleme yapılan liste ADT'sidir.



after(p) = C'nin konumu · before(p) = A'nın konumu

Şekil 6.26: Bekçili konumsal liste. Uçlardaki `header` ve `trailer` veri tutmayan bekçi düğümleridir. `p`, B'nin konumudur; `after(p)` C'nin, `before(p)` A'nın konumunu verir.

Kitaptaki tasarımda `Position<E>` arayüzü yalnız tek bir metot sunar: `getElement()`. Bağlı gerçeklemede konum düğümün kendisidir; düğüm sınıfı `Position`'ı uygular. Kullanıcı düğümün iç alanlarını göremez, yalnız öğesini okuyabilir ve konumu listeye geri verebilir.

Gerçeklemenin ikinci önemli fikri **bekçi** (*sentinel*) düğümleridir. Listenin başına `header`, sonuna `trailer` adlı, veri tutmayan iki düğüm konur. Boş listede bile bu iki düğüm vardır ve birbirini gösterir. Bunun kazancı şudur: her gerçek düğümün hem öncesinde hem sonrasında mutlaka bir düğüm bulunur. Ekleme her zaman iki düğümün arasına yapılır; "liste boş mu, ilk düğüm mü, son düğüm mü" gibi `null` denetimlerine gerek kalmaz.

Metot	Maliyet
`first()`, `last()`, `before(p)`, `after(p)`	*O(1)*
`addFirst(e)`, `addLast(e)`	*O(1)*
`addBefore(p, e)`, `addAfter(p, e)`	*O(1)*
`set(p, e)`, `remove(p)`	*O(1)*

Tablo 6.30: Konumsal liste metotları ve maliyetleri.

Tablodaki bütün metotların sabit süreli olması tesadüf değildir. Konum düğümün kendisi olduğu için hiçbir işlem düğüm aramaz; yalnız komşu bağlantıları değiştirir. Java'nın standart kütüphanesinde konumsal listenin doğrudan bir karşılığı yoktur. En yakını `ListIterator`'dır: o da imlecin bulunduğu yerde *O(1)* ekleme ve silme yapar. Ancak imleç tek bir yerde durur. Aynı listede yapısal değişiklik yapan iki yineleyici de bir arada kullanılamaz.

6.6.6 Kod izleme: addAfter(p, e)

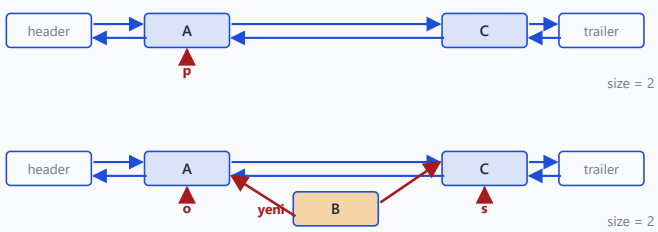
```
Position<E> addAfter(Position<E> p, E e) {
  Dugum<E> d = validate(p);
  return addBetween(e, d, d.sonraki);
}
Position<E> addBetween(E e, Dugum<E> o, Dugum<E> s) {
  Dugum<E> yeni = new Dugum<>(o, e, s);
  o.sonraki = yeni;
  s.onceki = yeni;
  size++;
  return yeni;
}
```

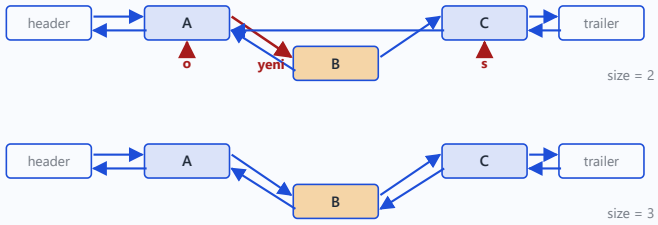
`addAfter`, önce `validate(p)` ile verilen konumun bu listenin geçerli bir düğümü olduğunu denetler ve onu iç düğüm türüne çevirir. Sonra işi `addBetween`'e devreder: yeni öğe `d` ile `d.sonraki` arasına girecektir. `addBetween` dört atama yapar. Yeni düğüm oluşturulurken iki komşusunu gösterir; ardından önceki komşunun ileri bağlantısı ve sonraki komşunun geri bağlantısı yeni düğüme yöneltilir; boyut artar. Metot yeni düğümü, yani yeni öğenin konumunu döndürür.

Satır	Durum
`validate(p)`	`d` = düğüm(A)
`addBetween(e, d, d.sonraki)`	`o` = A, `s` = C
`yeni = new Dugum<>(o, e, s)`	B'nin önceki A, sonraki C; ama henüz kimse B'yi göstermiyor
`o.sonraki = yeni`	ileri yön: A → B → C
`s.onceki = yeni`	geri yön: C → B → A; eski A ⇄ C bağlantıları kalmadı
`size++`	`size = 3`
`return yeni`	B'nin konumu döner

Tablo 6.31: `header ⇄ A ⇄ C ⇄ trailer` listesinde `addAfter(p, "B")` (`p`, A'nın konumu).

Hiçbir satır listeyi gezmez ve hiçbir öğe kaymaz; dört atama yapılır. İşlemin *O(1)* olmasının nedeni budur. Atamaların sırası da önemlidir: yeni düğüm önce komşularını gösterecek biçimde oluşturulur, komşuların bağlantıları ancak sonra değiştirilir. Ters sırada yapılsaydı, örneğin önce `o.sonraki = yeni` yazılıp sonra `yeni.sonraki = o.sonraki` denseydi, yeni düğüm kendini gösterirdi.





Şekil 6.27: (a) Başlangıç: `p`, A'nın konumu. (b) Yeni düğüm iki komşusunu gösteriyor. (c) `o.sonraki = yeni`. (d) `s.onceki = yeni` ve `size = 3`: B iki yönden de bağlı.

Konumsal listede `remove(p)` da aynı nedenle *O(1)*'dir: konum düğümün kendisidir, düğümün iki komşusu bağlantılarından okunur ve birbirine bağlanır. `LinkedList.remove(int)` ise önce indise kadar yürümek zorunda olduğu için *O(n)*'dir. Aradaki fark veri yapısında değil, işlemin aldığı argümandadır.

Sık yapılan hata

Silinmiş bir konumu yeniden kullanmak. `remove(p)` sonrasında `p` artık listede değildir. İyi bir gerçekleme silinen düğümün bağlantılarını `null` yapar ve `validate` bu durumda istisna fırlatır. Denetim olmasaydı `addAfter(p, e)`, listeden kopmuş bir düğümün komşularını değiştirerek listeyi bozabilirdi.

6.6.7 ArrayList mi, LinkedList mi?

	`ArrayList`	`LinkedList`
İç yapı	büyüyebilen dizi	çift yönlü bağlı düğümler
`get(i)`	*O(1)*	*O(n)*
Sona ekleme	amortize *O(1)*	*O(1)*
Başa ekleme / baştan silme	*O(n)*	*O(1)*
Öğe başına bellek	1 referans (+ boş kapasite)	düğüm nesnesi (3 referans)
Önbellek	ardışık, hızlı	dağınık, yavaş

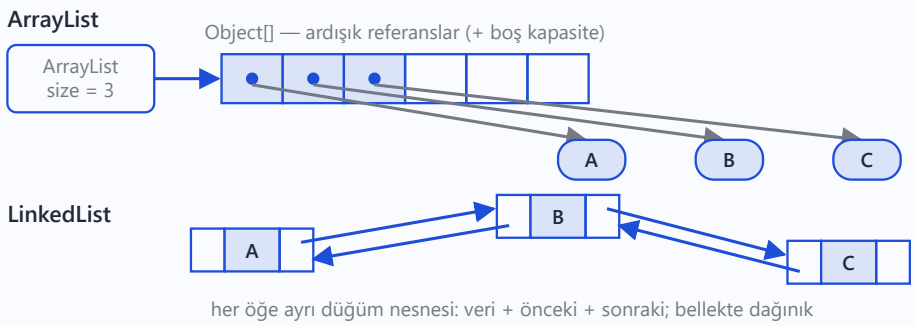
Tablo 6.32: `ArrayList` ile `LinkedList`'in karşılaştırması.

Tablo kâğıt üzerinde dengeli görünür, ama pratikte terazi `ArrayList`'ten yanadır. Programların çoğu okuma ağırlıklıdır ve sona ekler; bu iki işte `ArrayList` en az `LinkedList` kadar iyidir. Üstelik `LinkedList`'in kâğıttaki avantajları, bellek ve önbellek etkileri (6.6.8) yüzünden çoğu zaman gerçeğe yansımaz. Bu yüzden varsayılan seçim `ArrayList`'tir.

İki uçta da sık ekleme ve silme gerekiyorsa, yani yapı bir kuyruk ya da çift uçlu kuyruk gibi kullanılıyorsa `LinkedList` düşünülebilir. Bu iş için genellikle daha iyi bir seçenek de vardır: `ArrayDeque`. Dairesel bir dizi kullandığı için iki uçta da amortize *O(1)*'dir ve `LinkedList`'in bellek yükünü taşımaz.

`LinkedList`'in gerçekten öne çıktığı yer, bir `ListIterator`'la gezinirken araya çok sayıda ekleme ve silme yapılan durumlardır.

6.6.8 Bellekte iki liste



Şekil 6.28: Bellek yerleşimi. Üstte `ArrayList`: ardışık bir referans dizisi ve dizinin gösterdiği öğe nesneleri. Altta `LinkedList`: her öğe için bellekte dağınık duran ayrı bir düğüm nesnesi.

`ArrayList`'te öğe başına yaklaşık bir referans harcanır; bu, JVM ayarına göre 4 ya da 8 bayttır. Buna boş kapasitedeki hücreler eklenir. `LinkedList`'te ise her öğe için ayrı bir düğüm nesnesi oluşturulur. Düğüm nesnesinin bir başlığı ve üç referansı (önceki, veri, sonraki) vardır; toplam 24–40 bayt civarındadır. Aynı sayıda öğe için `LinkedList` birkaç kat fazla bellek kullanır.

Daha az bilinen ama daha etkili fark **önbellekle** (*cache*) ilgilidir. İşlemci bellekten veri okurken tek bir hücreyi değil, çevresindeki bir bloğu birlikte getirir ve kendi hızlı önbelleğine koyar. `ArrayList`'in referans dizisi ardışık olduğu için bir hücre okunduğunda sonraki birkaç hücre de önbelleğe gelmiş olur; gezinme önbellekten beslenir. `LinkedList`'in düğümleri ise bellekte dağınıktır; her `sonraki` bağlantısını izlemek, önbellekte bulunmayan bir adrese gitmek demek olabilir. Bu duruma **önbellek ıskası** (*cache miss*) denir ve ana bellekten okuma önbellekten okumadan onlarca kat yavaştır. İki yapı da gezinmede *O(n)*'dir, ama gerçek sürede `ArrayList` çoğu zaman belirgin biçimde öndedir.

6.7 Öteki liste sınıfları

`ArrayList` ve `LinkedList` günlük işlerin neredeyse hepsini karşılar. Java'da bunların yanında özel durumlar için tasarlanmış ya da eski sürümlerden kalmış dört yapı daha vardır. Bu yapıların çoğu, birden çok iş parçacığının aynı listeyi kullandığı durumlarla ilgilidir.

6.7.1 Vector

`Vector`, `ArrayList` gibi büyüyebilen bir dizidir: öğelere indisle erişilir, gerektiğinde büyür. İki farkı vardır. Birincisi, büyürken kapasitesini varsayılan olarak iki katına çıkarır. İkincisi, bütün metotları `synchronized`'dır.

`synchronized` anahtar sözcüğü, metodu bir **kilitle** korur. Bir iş parçacığı metodu çalıştırırken başka bir iş parçacığı aynı nesnenin senkronize bir metoduna giremez; kilit bırakılana kadar bekler. Böylece iki iş parçacığı aynı anda `add` yapıp iç diziyi bozamaz. Tek tek işlemler çok iş parçacıklı ortamda güvenlidir.

```
Vector<Integer> v = new Vector<>();
v.add(10);            // [10]
v.add(20);            // [10, 20]
v.get(0);             // 10
v.remove(0);          // [20]
v.size();             // 1
v.capacity();         // 10

// bileşik işlem: dışarıdan kilit gerekir
synchronized (v) {
    if (!v.isEmpty()) v.remove(0);
}
```

Koddaki `v.capacity()` 10 döndürür: `Vector` ilk dizisini 10 hücreyle ayırır ve silmede küçülmez. Kilidin bir bedeli vardır: tek iş parçacıklı bir programda bile her çağrı kilidi alıp bırakır. Daha önemlisi, kilitler yalnız tek tek metotları korur. "Kontrol et, sonra yap" biçimindeki bileşik işlemler, örneğin "boş değilse ilk öğeyi sil", yine de güvenli değildir. Kodun son satırları bu yüzden bileşik işlemi bir `synchronized (v) { … }` bloğuna alır.

`Vector`, Java 1.0'dan kalma eski bir sınıftır. Yeni kodda tek iş parçacığı için `ArrayList`, çok iş parçacığı için `Collections.synchronizedList(...)` ya da `CopyOnWriteArrayList` tercih edilir.

6.7.2 "Kontrol et, sonra yap" yarışı

Bileşik işlemin neden korunmasız kaldığını iki iş parçacığının olası bir sıralanışı gösterir. `v = [20]` olsun. T1 ve T2 adlı iki iş parçacığı aynı satırı çalıştıracak: `if (!v.isEmpty()) v.remove(0);`.

Adım	T1	T2	`v`
1	`isEmpty()` → `false`	—	[20]
2	—	`isEmpty()` → `false`	[20]
3	`remove(0)` → 20	—	[ ]
4	—	`remove(0)` → `ArrayIndexOutOfBoundsException`	[ ]

Tablo 6.33: İki iş parçacığının bir sıralanışı. Her metot tek başına kilitlidir, ama iki çağrının arası korunmasızdır.

Birinci adımda T1 `isEmpty()`'nin kilidini alır, `false` görür ve kilidi bırakır. Tam bu anda işletim sistemi T2'yi çalıştırabilir. T2 de `v`'yi bir öğeli görür. İki iş parçacığı da silebileceğine karar vermiştir. T1 tek öğeyi siler; T2'nin silme girişimi boş vektörde istisna fırlatır. Her metot ayrı ayrı senkronizedir, ama iki metot birlikte **atomik** değildir; yani bölünmez bir bütün gibi çalışmaz. Sonucun iş parçacıklarının sıralanışına bağlı olduğu bu tür hatalara **yarış durumu** (*race condition*) denir.

Çözüm, bileşik işlemi tek bir kilidin altına almaktır:

```
synchronized (v) { if (!v.isEmpty()) v.remove(0); }
```

`synchronized (v)` bloğu, `Vector`'ın kendi metotlarının kullandığı kilidin aynısını alır. T1 bloğa girdiğinde T2 blok bitene kadar bekler. T2 kilidi aldığında `v`'yi boş görür ve silmez. "Kontrol et, sonra yap" artık güvenlidir.



Şekil 6.29: (a) T2 araya girdi; ikisi de vektörü dolu gördü. (b) T1 sildi; T2'nin silmesi hata verdi. (c) Çözüm: kilit T1'deyken T2 bloğa giremez. (d) T2 kilidi alınca vektörü boş görür ve silmez.

Sık yapılan hata

"Sınıf iş parçacığı güvenli, öyleyse benim kodum da güvenli" diye düşünmek. `Vector`, `Collections.synchronizedList` ve benzerleri yalnız tek tek çağrıları korur. Birden çok çağrıdan oluşan her mantıksal işlem (kontrol et ve sil, yoksa ekle, gez ve değiştir) ayrıca kilitlenmelidir.

6.7.3 Stack sınıfı

`Stack`, `Vector`'ı genişletir ve **son giren ilk çıkar** (LIFO, *last-in, first-out*) ilkesiyle çalışan bir yığın sunar. İşlemler yalnız yığının üstündeki öğeyi etkiler. Geri al / yinele (*undo/redo*) gibi özellikler için kullanışlıdır: son yapılan iş, ilk geri alınacak iştir.

Metot	Görevi
`push(e)`	üste ekler
`pop()`	üsttekini çıkarır ve döndürür
`peek()`	üsttekini döndürür, çıkarmaz
`empty()`	yığın boş mu?
`search(o)`	üstten 1'den başlayan uzaklık; yoksa −1

Tablo 6.34: `Stack` metotları.

Boş yığında `pop` ya da `peek` çağrılırsa `EmptyStackException` fırlatılır. `search`'ün sayması alışılmışın dışındadır: üstteki öğe 1'dir. `a`, `b`, `c` sırasıyla push edilmiş bir yığında `search("c")` 1, `search("a")` 3 döndürür.

`Stack`'in bir tasarım kusuru vardır. `Vector`'dan türediği için onun bütün metotlarını da taşır: `get(i)`, `add(i, e)`, `remove(i)`. Bu metotlar yığının ortasına erişerek LIFO ilkesini bozar. Örneğin `add(0, "z")` yığının *dibine* öğe sokar. Ayrıca her işlem `Vector`'ın senkronizasyon yükünü taşır. Java belgeleri bu yüzden yığın için `Deque` arayüzünü önerir:

```
Deque<E> s = new ArrayDeque<>();
```

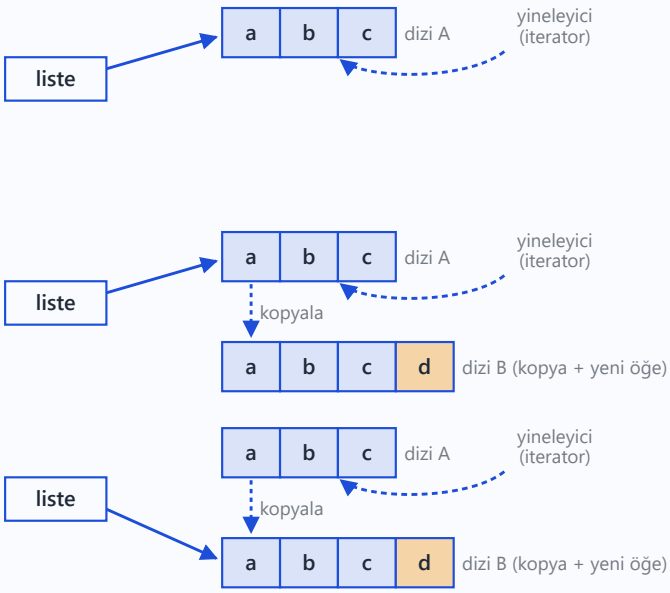
`ArrayDeque`'in `push`, `pop` ve `peek` metotları `Stack`'tekilerle aynı anlamdadır, ama kilit kullanmaz ve yığının ortasına erişim sunmaz.

6.7.4 CopyOnWriteArrayList

Bazı listeler çok sık okunur ama çok seyrek değişir. Bir grafik arayüzde bir düğmeye tıklanınca haber verilecek nesnelerin listesi, yani **olay dinleyicileri** (*listeners*), buna iyi bir örnektir. Liste her olayda baştan sona gezilir; yeni dinleyici eklemek ise nadirdir. Birden çok iş parçacığı listeyi aynı anda gezerken kilit beklemek pahalıdır. `CopyOnWriteArrayList` (yazarken kopyalayan dizi listesi) bu durum için tasarlanmıştır.

Fikir adında saklıdır. Listede bir değişiklik (`add`, `set`, `remove`) yapılacağı zaman iç dizi kopyalanır, değişiklik kopyada yapılır, sonra liste referansı yeni diziye yöneltilir. Eski dizi hiç değiştirilmez. Bunun sonuçları şunlardır:

- **Okumalar kilitsizdir ve çok hızlıdır.** Okuyucu, o anda listenin gösterdiği diziyi okur; dizi hiçbir zaman yerinde değişmediği için kilide gerek yoktur.
- **Yazmalar pahalıdır.** Her yazma bütün diziyi kopyalar ve *O(n)* sürer. Yazarlar kısa bir kilitle sıraya girer; böylece iki yazar birbirinin kopyasını ezmez.
- **Yineleyiciler anlık görüntü gezer.** Bir yineleyici, oluşturulduğu anda listenin gösterdiği diziyi gezer. Gezinme sırasında listeye yazılsa bile yineleyici eski diziyi görmeye devam eder; `ConcurrentModificationException` hiç fırlatılmaz. Karşılığında yineleyici yeni öğeleri görmez ve `it.remove()` desteklenmez.



Şekil 6.30: Yazarken kopyalama. (a) Liste ve yineleyici aynı A dizisine bakıyor. (b) `add("d")`: yazar A'yı kopyalayıp yeni öğeyi kopyaya (B) ekliyor. (c) Liste referansı B'ye döndü; eski yineleyici A'yı gezmeye devam ediyor.

Uygun olduğu yerler: sık gezilen, nadiren değişen, birçok iş parçacığının paylaştığı listeler; olay dinleyicileri, yapılandırma bilgileri. Uygun olmadığı yerler: büyük ve sık değişen listeler. Bir milyon öğeli bir listeye her eklemede bir milyon öğe kopyalanır.

6.7.5 Kod izleme: yazarken kopyalama

```
public boolean add(E e) {
  synchronized (kilit) {
    Object[] eski = dizi;
    int n = eski.length;
    Object[] yeni = Arrays.copyOf(eski, n + 1);
    yeni[n] = e;
    dizi = yeni;       // tek referans ataması
    return true;
  }
}
```

Kodun her satırı tek bir işi yapar. `synchronized (kilit)`, yazarları sıraya sokar; okuyucular bu kilide hiç dokunmaz. `eski = dizi`, o anki diziyi yerel bir değişkene alır. `Arrays.copyOf(eski, n + 1)`, bir hücre büyük yeni bir dizi oluşturup bütün öğeleri kopyalar. `yeni[n] = e`, değişikliği yalnız kopyada yapar. `dizi = yeni` ise tek bir referans atamasıdır: liste o andan sonra yeni diziyi gösterir. Java'da referans ataması bölünmez; bir okuyucu ya eski diziyi ya yeni diziyi görür, yarım kalmış bir diziyi asla görmez.

Adım	Liste gösteriyor	Yineleyici	Not
başlangıç	A = [a, b, c]	A, `cursor = 1`	"a" okunmuş
kilit alınır	A	A	okuyucular beklemez
`eski = dizi`, `n = 3`	A	A	—
`copyOf(eski, 4)`	A	A	B = [a, b, c, –]; *O(n)*
`yeni[3] = "d"`	A	A	B = [a, b, c, d]
`dizi = yeni`	B	A	tek referans ataması
`return true`	B	A	kilit bırakılır
`it.next()` ×2	B	A: "b", "c"	"d" görülmez; istisna yok
yeni `iterator()`	B	B	[a, b, c, d] gezilir

Tablo 6.35: `[a, b, c]` listesinde bir yineleyici "a"yı okumuşken `add("d")`.

Eski yineleyici işini bitirdiğinde A dizisini gösteren hiçbir referans kalmaz ve çöp toplayıcı onu temizler. Aynı mantık for-each döngüsünde de geçerlidir. `[a, b, c, d]` listesi for-each ile gezilirken "a" görüldüğünde listeye "e" eklensin. Döngü yine yalnız a, b, c, d'yi yazar ve istisna fırlatmaz. Döngü bittiğinde liste `[a, b, c, d, e]` olur.



Şekil 6.31: (a) `Arrays.copyOf` : B dizisi oluşturuldu, A'ya dokunulmadı. (b) `dizi = yeni` : liste B'yi, yineleyici hâlâ A'yı gösteriyor. (c) Yineleyici A'nın sonuna geldi. (d) Yeni bir yineleyici B'yi gezer.

6.7.6 Arrays.asList: diziden liste görünümü

Elde bir dizi varken onu bir `List` bekleyen bir metoda vermek gerekebilir. `Arrays.asList(dizi)` bunu kopyalamadan yapar: diziyi saran bir liste görünümü döndürür. Dizi ile liste aynı belleği paylaşır.

```
String[] dizi = {"a", "b", "c"};
List<String> liste = Arrays.asList(dizi);
liste.set(0, "x");     // çalışır: dizi[0] de artık "x"
dizi[2] = "z";          // liste de değişir → [x, b, z]
liste.add("d");         // UnsupportedOperationException (boyut sabit)
List<String> esnek = new ArrayList<>(Arrays.asList(dizi));   // bağımsız kopya
```

Bu listenin davranışı iki kuralla özetlenir. Birincisi, liste **sabit boyutludur**: dizinin boyu değişemeyeceği için `add` ve `remove` `UnsupportedOperationException` fırlatır. İkincisi, liste **değiştirilemez değildir**: `set` çalışır, çünkü boyutu değiştirmez; yalnız dizinin bir hücresine yazar. Değişiklik iki yöne de yansır: `liste.set(0, "x")` diziyi, `dizi[2] = "z"` listeyi değiştirir.

	`set`	`add` / `remove`	**Diziyle bağlantı**
`Arrays.asList(dizi)`	✓	✗	paylaşır
`List.of(...)`	✗	✗	yok (değiştirilemez)
`new ArrayList<>(...)`	✓	✓	yok (kopya)

Tablo 6.36: Üç yolla oluşturulan listelerin karşılaştırması.

Sunumdaki izleme, bağımsız ve büyüyebilen bir kopyanın nasıl alındığını da gösterir:

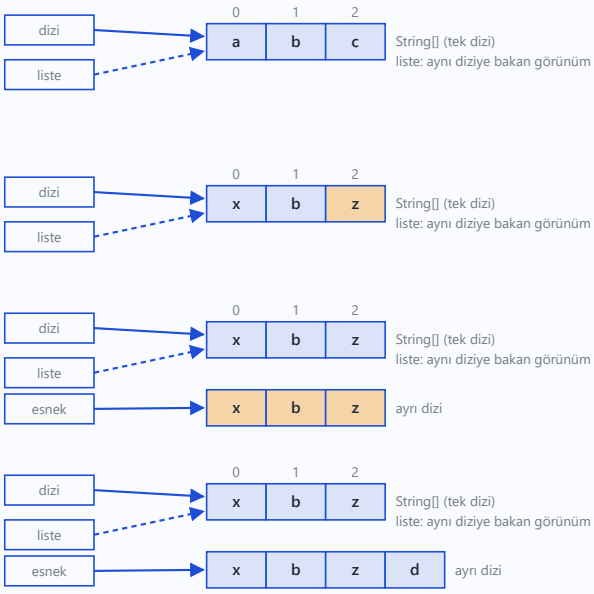
```
String[] dizi = {"a", "b", "c"};
List<String> liste = Arrays.asList(dizi);
liste.set(0, "x");
dizi[2] = "z";
// liste.add("d");  → hata (sabit boyut)
List<String> esnek = new ArrayList<>(liste);
esnek.add("d");
System.out.println(liste + " " + esnek);
```

Satır	`dizi`	`liste`	`esnek`
`String[] dizi = …`	[a, b, c]	—	—
`Arrays.asList(dizi)`	[a, b, c]	[a, b, c] (aynı dizi)	—
`liste.set(0, "x")`	[x, b, c]	[x, b, c]	—
`dizi[2] = "z"`	[x, b, z]	[x, b, z]	—
`new ArrayList<>(liste)`	[x, b, z]	[x, b, z]	[x, b, z] (ayrı dizi, kapasite 3)
`esnek.add("d")`	[x, b, z]	[x, b, z]	[x, b, z, d] (kapasite 4)

Tablo 6.37: `Arrays.asList` izlemesi.

Program `[x, b, z] [x, b, z, d]` yazdırır. `Arrays.asList` kopyalamadığı için *O(1)*'dir. `new ArrayList<>(liste)` ise öğeleri tam öğe sayısı kadar hücreli yeni bir diziye kopyalar (*O(n)*); ilk eklemede bu dizi *3 + 3/2 = 4* hücreye büyür. Kopyaya ekleme serbesttir ve ne diziyi ne de görünümü etki-

ler.



Şekil 6.32: (a) `Arrays.asList` : `liste` aynı diziye bakan bir görünüm. (b) `dizi[2] = "z"` listeden de görülür. (c) `new ArrayList<>(liste)` ayrı bir dizi ayırır. (d) Kopyaya "d" eklendi; dizi ve görünüm değişmedi.

Sık yapılan hata

İlkel türden bir diziyi `Arrays.asList`'e vermek. `int[] d = {1, 2, 3}; Arrays.asList(d)` üç öğeli bir `List<Integer>` *vermez*; tek öğeli bir `List<int[]>` verir ve `size()` 1 döndürür. Jenerik tür parametresi ilkel olamadığı için dizinin kendisi tek bir öğe sayılır. Sayı listesi gerekiyorsa dizi `Integer[]` olarak tanımlanır ya da `Arrays.stream(d).boxed().toList()` kullanılır.

6.8 Özet

- **Liste**, öğeleri bir konum sırası içinde tutan, 0'dan başlayan indislerle erişilen ve tekrara izin veren ADT'dir. Temel işlemleri ekleme, araya ekleme, silme, indisleme, dilimleme, arama ve uzunluktur. Uzunluk (`size`) ile kapasite farklı şeylerdir.
- Java'da ADT arayüzle, gerçekleme sınıfla ifade edilir. Hiyerarşi `Iterable` → `Collection` → `List` diye iner. `Iterable` for-each döngüsünü mümkün kılar; `Collection` ortak işlemleri verir ama indis içermez; `List` konuma dayalı metotları ekler. Değişken arayüz türüyle tanımlanır: `List<String> l = new ArrayList<>();` .
- `subList` bir **görünümdür**: değişiklik asıl listeye yansır; asıl liste dışarıdan yapısal değişirse görünüm geçersizleşir. `List.of` / `List.copyOf` **değiştirilemez**, `Collections.unmodifiableList` yalnız **salt okunurdur**, `new ArrayList<>(…)` bağımsız bir kopyadır.

- `remove(int)` indis siler ve öğeyi döndürür; `remove(Object)` değer siler ve `boolean` döndürür. `List<Integer>`'te ilkel argüman indis sayılır. Jenerik tür parametresi ilkel olamaz; derleyici kutulama ve kutudan çıkarmayı kendisi yapar, `null` öğe kutudan çıkarılırken `NullPointerException` verir.
- Gezinmenin üç yolu indisli döngü, for-each/forEach ve yineleyicidir. `LinkedList`'te indisli döngü $O(n^2)$, öteki ikisi $O(n)$'dir. Yineleyici öğelerin arasında duran bir imleç gibi düşünülür.
- Gezinirken listeyi dışarıdan değiştirmek, hızlı başarısız yineleyicinin `modCount` denetimine takılır ve `ConcurrentModificationException` verir. Güvenli silme `it.remove()` ya da `removeIf` ile yapılır; `it.remove()` her `next()`'ten sonra bir kez çağrılabilir. `ListIterator` iki yönde gezer ve gezinirken `add`, `set`, `remove` yapar. `Spliterator` listeyi paralel işleme için parçalara böler.
- `ArrayList` büyüyebilen bir dizidir: `get` / `set` $O(1)$, sona ekleme amortize $O(1)$, araya ekleme ve silme $O(n)$. Kapasite dolunca yaklaşık 1,5 katına çıkar (boş listede ilk ekleme 10). Geometrik büyütmede toplam kopya $O(n)$, sabit artışta $O(n^2)$'dir.
- `LinkedList` çift yönlü bağlı listedir: uçlarda ekleme ve silme $O(1)$, indisle erişim $O(n)$. Bağlantı değiştirmek ucuz, konumu bulmak pahalıdır. **Konumsal liste** konumu düğümün kendisi yaparak ve bekçi düğümler kullanarak bütün işlemleri $O(1)$'e indirir.
- Varsayılan seçim `ArrayList`'tir; bellek kullanımı ve önbellek davranışı onu pratikte öne çıkarır. Uçlarda sık işlem için `ArrayDeque` ya da `LinkedList` düşünülür.
- `Vector` ve `Stack` eski, senkronize sınıflardır; bileşik işlemleri yine korumazlar. Yığın için `ArrayDeque` önerilir. `CopyOnWriteArrayList` her yazmada kopyalar, yineleyicileri anlık görüntü gezer. `Arrays.asList` diziyi paylaşan sabit boyutlu bir görünümdür: `set` çalışır, `add` / `remove` çalışmaz.

Sınıf	`get(i)`	Sona ekleme	Başa / araya	İş parçacığı
`ArrayList`	O(1)	amortize O(1)	O(n)	güvenli değil
`LinkedList`	O(n)	O(1)	O(1) / O(n)	güvenli değil
`Vector`	O(1)	amortize O(1)	O(n)	senkronize
`Stack`	O(1)	`push` O(1)	—	senkronize
`CopyOnWriteArrayList`	O(1)	O(n)	O(n)	güvenli (kopyalayarak)
`Arrays.asList`	O(1)	✗	✗	güvenli değil

Tablo 6.38: Liste sınıflarının özeti.

6.9 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

liste *list* · koleksiyon *collection* · soyut veri tipi *abstract data type (ADT)* · arayüz *interface* · jenerikler *generics* · indis *index* · kapasite *capacity* · görünüm *view* · değiştirilebilir *mutable* · değiştirilemez *immutable* · kalıcı veri yapısı *persistent data structure* · otomatik kutulama *autoboxing*

kutudan çıkarma *unboxing* | gezinme *traversal* | yineleyici *iterator* | imleç *cursor*
amortize analiz *amortized analysis* | bağlı liste *linked list* | çift yönlü bağlı liste *doubly linked list*
düğüm *node* | konum *position* | konumsal liste *positional list* | bekçi *sentinel* | null *null*
çift uçlu kuyruk *deque*

6.10 Kendini Dene

1. `l = [A, B, C, D, E, F]` bir `ArrayList` olsun. Önce `l.subList(1, 4).clear();`, sonra `l.subList(0, 2).set(1, "Q");` çalıştırılıyor. Son durumda `l` nedir?

   Cevap

   İlk görünüm *[1, 4)* aralığını, yani B, C, D'yi kapsar; `clear()` bunları asıl listeden siler: `[A, E, F]`. İkinci görünüm *[0, 2)*, yani `[A, E]`'dir. `set(1, "Q")` görünümün 1. öğesini, yani asıl listenin *0 + 1 = 1*. indisini değiştirir. Sonuç `[A, Q, F]`'tir.

2. `List<Integer> m = new ArrayList<>(List.of(4, 0, 2, 1));` listesine sırayla `m.remove(m.get(1));` ve `m.remove(1);` uygulanıyor. Liste nedir?

   Cevap

   `m.get(1)` bir `Integer` (0) döndürür; argüman nesne olduğu için `remove(Object)` seçilir ve *0 değeri* silinir: `[4, 2, 1]`. `m.remove(1)`'in argümanı ilkel `int`'tir; `remove(int)` seçilir ve 1. indisteki 2 silinir. Sonuç `[4, 1]`'dir.

3. `new ArrayList<>()` ile oluşturulan boş listeye 25 öğe ekleniyor. Kapasite hangi eklemelerde kaça çıkar? Büyütmelerde toplam kaç öğe kopyalanır?

   Cevap

   1. eklemede 0'dan 10'a (ilk dizi ayrılır, kopya yok), 11. eklemede 10'dan 15'e (10 kopya), 16. eklemede 15'ten 22'ye (15 kopya), 23. eklemede 22'den 33'e (22 kopya). Toplam *10 + 15 + 22 = 47* kopya; sonunda boyut 25, kapasite 33'tür.

4. 7 öğeli bir `LinkedList` indisli `for` döngüsüyle `get(i)` kullanılarak gezilirse toplam kaç bağlantı yürünür? Yineleyiciyle kaç bağlantı yürünürdü?

   Cevap

   n/2'nin tam sayı kısmı 3'tür. *i = 0, 1, 2* baştan 0, 1, 2 bağlantı; *i = 3, 4, 5, 6* sondan *7 − 1 − i* = 3, 2, 1, 0 bağlantı yürür. Toplam *0 + 1 + 2 + 3 + 2 + 1 + 0 = 9*'dur. Yineleyici her `next()`'te bir bağlantı ilerler: toplam *n − 1 = 6*.

5. `r = [P, Q, R]` listesinde `ListIterator<String> it = r.listIterator(3);` alınıyor ve sırayla `it.previous()`, `it.previous()`, `it.set("X")`, `it.add("Y")`, `it.next()` çağrılıyor. Liste ne olur, son çağrı ne döndürür?

   Cevap

   İmleç sondan başlar: `P Q R |`. İlk `previous()` R'yi, ikincisi Q'yu döndürür; imleç `P | Q R`. `set("X")` son döndürülen Q'yu X yapar: `P | X R`. `add("Y")` imlecin önüne ekler: `P Y | X R`. `next()` X'i döndürür. Liste `[P, Y, X, R]` 'dir.

6. Beş öğeli bir `ArrayList` 'in spliterator'ı `a` ile alınıp `Spliterator b = a.trySplit();` çağrılıyor. `a` ve `b` hangi indisleri kapsar?

   Cevap

   Sınır 5 olarak belirlenir ve orta nokta *(0 + 5) / 2 = 2*'dir. İlk yarı yeni spliterator'a verilir: `b` *[0, 2)*, yani 0 ve 1 indislerini; `a` *[2, 5)*, yani 2, 3 ve 4 indislerini kapsar.

7. Kapasitesi 10 olan boş bir dinamik diziye 100 öğe ekleniyor. Kapasite (a) her dolumda 10 artırılırsa, (b) her dolumda *eski + eski/2* yapılırsa toplam kaç kopya yapılır?

   Cevap

   (a) Büyütmeler 10, 20, ..., 90 öğede yapılır: *10 + 20 + ... + 90 = 450* kopya. (b) Kapasite 10 → 15 → 22 → 33 → 49 → 73 → 109 diye büyür: *10 + 15 + 22 + 33 + 49 + 73 = 202* kopya. Sabit artış öğe sayısıyla karesel, oranlı büyütme doğrusal büyür.

8. `Stack<String> s` boş bir yığın olsun. `s.push("a"); s.push("b"); s.push("c"); s.add(0, "z");` sonrasında `s.pop()` ne döndürür ve ardından `s.search("z")` kaçtır? Bu örnek hangi tasarım sorununu gösterir?

   Cevap

   `add(0, "z")`, `Vector` 'dan gelen bir metottur ve "z"yi yığının dibine sokar: `[z, a, b, c]`. `pop()` üstteki "c"yi döndürür; yığın `[z, a, b]` olur. `search("z")` üstten 1'den sayar: b 1, a 2, z 3. Örnek, `Stack` 'in `Vector` 'dan türediği için LIFO ilkesini bozan metotları da taşıdığını gösterir.

9. `CopyOnWriteArrayList` `c = [a, b, c, d]` for-each ile geziliyor; gövde, öğe "a" ise `c.add("e")` yapıyor ve her öğeyi yazdırıyor. Ne yazdırılır, istisna olur mu? Aynı kod bir `ArrayList` ile çalıştı-

rılsaydı ne olurdu?

> **Cevap**
>
> Yineleyici, oluşturulduğu andaki diziyi (anlık görüntüyü) gezer. “e” eklenince liste yeni bir diziyi gösterir, ama yineleyici eski diziyle devam eder: `abcd` yazılır, istisna olmaz; döngü sonunda liste `[a, b, c, d, e]` 'dir. `ArrayList` 'te ekleme `modCount` 'u artırırdı ve bir sonraki `next()` `ConcurrentModificationException` fırlatırdı.

10. `int[] d = {1, 2, 3};` için `Arrays.asList(d).size()` kaç döndürür? Neden?

> **Cevap**
>
> 1 döndürür. Jenerik tür parametresi ilkel olamaz; `int` öğeleri `Integer` 'a kutulanmaz. Bunun yerine dizinin kendisi tek bir nesne sayılır ve sonuç tek öğeli bir `List<int[]>` olur. Üç öğeli liste için dizi `Integer[]` olarak tanımlanmalıdır.

6.11 İleri okuma

- M. T. Goodrich, R. Tamassia ve M. H. Goldwasser, *Data Structures and Algorithms in Java* (6. baskı, 2014), Bölüm 7: liste ADT'sini, dizi tabanlı listeyi ve amortize analizini, konumsal listeyi ve yineleyicileri bu nottaki `addBetween` koduna çok yakın bir Java gerçeklemesiyle anlatır.
- J. Bloch, *Effective Java* (3. baskı, 2018): nesnelere arayüz türüyle başvurma, for-each döngüsünü tercih etme, ilkel türleri kutulu türlere tercih etme ve değiştirilebilirliği en aza indirme üzerine maddeleri, bu bölümdeki tasarım önerilerinin gerekçelerini Koleksiyon Çerçevesinin tasarımcılarından birinin ağzından verir.
- Oracle, Java SE 17 API belgeleri: `List`, `ArrayList`, `LinkedList`, `ListIterator` ve `CopyOnWriteArrayList` sayfaları her metodun sözleşmesini, fırlattığı istisnaları ve hızlı başarısız yineleyicilerin sınırlarını kesin bir dille tanımlar.

6.12 Kaynaklar

1. J. Bloch, *Effective Java*, 3. baskı, Addison-Wesley, 2018.
2. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6. baskı, Wiley, 2014.
3. Oracle, *Java SE 17 API Specification*: `java.util.Collection`, `List`, `Iterator`, `ListIterator` ve `Spliterator` belgeleri.

KISIM II · BÖLÜM 7

Yığınlar

Yığın, veri yapılarının en sade ama en çok iş gören örneklerinden biridir. Öğeler yalnız bir uçtan eklenir ve yine yalnız o uçtan çıkarılır; en son giren öğe ilk çıkar. Bu tek kural, ilk bakışta bir kısıtlama gibi görünür. Oysa aynı kural yığının bütün temel işlemlerini sabit sürede yapmasını sağlar ve "en son açılanı ilk kapat" biçimindeki pek çok problemi doğal olarak çözer. Not, yığının soyut tanımıyla başlar; dizi ve bağlı liste ile nasıl gerçekleneceğini, dinamik dizinin neden iki katına büyütüldüğünü anlatır. Ardından tarayıcı geçmişi, çağrı yığını, parantez kontrolü, sonraki büyük öğe, Hanoi kuleleri ve stock span gibi uygulamalara geçer. Son konu, aritmetik ifadelerin ara ek, önek ve sonek gösterimleri arasında yığınla dönüştürülmesi ve sonek ifadelerin değerlendirilmesidir.

Bu bölümün kodu: `github.com/sercankulcu/data-structures-java/Bolum07/src`

ÖĞRENME ÇIKTILARI

LIFO modeli
Yığının son giren ilk çıkar davranışını açıklamak ve bir işlem dizisinden sonra yığının durumunu ve dönen değerleri bulmak.

Gerçekleme
Yığını dizi ve bağlı liste kullanarak gerçeklemek; taşma, alt taşma ve dinamik büyütme durumlarını kodda doğru ele almak.

Uygulamalar
Yığını parantez kontrolü, ifade dönüşümü ve değerlendirme, çağrı yönetimi, geri/ileri gezinme, sonraki büyük öğe ve stock span gibi problemlerde kullanmak.

Maliyet analizi
Temel yığın işlemlerinin zaman ve alan maliyetini gerekçesiyle vermek; iki katına büyütmenin amortize maliyetini hesaplamak.

7.1 Yığın ADT'si

Bir veri yapısını iki düzeyde anlatmak mümkündür. Birinci düzey, yapının *ne yaptığını* söyler: hangi işlemler vardır, her işlem neyi değiştirir, ne döndürür. İkinci düzey, bunun *nasıl yapıldığını* söyler: veriler bellekte hangi biçimde saklanır, her işlem hangi adımlarla yürür. Birinci düzeydeki tarife **soyut veri tipi** (*abstract data type*, kısaca ADT) denir. Bu kesimde yığını önce bu soyut düzeyde tanıyoruz; gerçekleme ayrıntıları bir sonraki kesimin konusudur.

7.1.1 Yığın nedir?

Mutfakta üst üste dizilmiş temiz tabakları düşünelim. Yıkanan tabak her zaman en üste konur. Sofraya tabak gerektiğinde de en üstteki alınır. En alttaki tabağa ulaşmak için üstündekilerin hepsini tek tek kaldırmak gerekir. Kimse yığının ortasından tabak çekmeye kalkmaz; hem zordur hem de yığın devrilir.

Bilgisayar bilimindeki **yığın** (*stack*) tam olarak bu düzeni taklit eder. Öğeler bir sıra hâlinde tutulur, ama sıranın yalnız bir ucu kullanılabilir. Bu uca yığının **üstü** (*top*) denir. Ekleme de çıkarma da üstte yapılır. Böylece yığından çıkan öğe, her zaman o anda yığında bulunanlar arasında en son eklenmiş olandır.

> **Tanım 7.1**
>
> **Yığın**, öğeleri bir sıra içinde tutan ve ekleme ile çıkarmanın yalnız bir uçtan, yığının üstünden yapıldığı soyut veri tipidir. Bu düzene **LIFO** (*Last-In, First-Out*: son giren ilk çıkar) denir. Aynı düzen, "ilk giren son çıkar" anlamında **FILO** (*First-In, Last-Out*) adıyla da anılır.

LIFO ve FILO aynı davranışın iki ayrı söylenişidir. Bir yığına 1, 2, 3 sırasıyla öğe eklenip ardından üç kez çıkarma yapılırsa 3, 2, 1 sırası elde edilir. Son giren 3 ilk çıkmıştır (LIFO); ilk giren 1 ise en son çıkmıştır (FILO).

Yığın düzeni yazılımda sık karşımıza çıkar. Üç tanıdık örnek, kavramın neden bu kadar yaygın olduğunu gösterir:

- **Geri al** (*undo*): Bir metin düzenleyicide yapılan her değişiklik bir yığına eklenir. "Geri al" komutu en son yapılan değişikliği geri çevirir; bir daha basılınca ondan önceki değişikliği çevirir. Değişiklikler yapıldıkları sıranın tersiyle geri alınır.
- **Tarayıcıdaki "geri" düğmesi**: Ziyaret edilen sayfalar bir yığında tutulur. "Geri"ye basıldığında en son ayrılan sayfaya dönülür. Bu örnek, iki yığınla birlikte 7.3.2'de ayrıntılı işlenir.
- **Fonksiyon çağrıları**: `main` bir `f` fonksiyonunu, `f` de `g`'yi çağırırsa önce `g` biter, sonra `f`, en son `main`. En son çağrılan fonksiyon ilk biter. Programlama dilleri bu yüzden çağrıları bir yığında izler (7.3.3).

Üç örneğin ortak noktası şudur: hepsinde “en son başlayan, henüz bitmemiş iş” ile ilgilenilir. Bir işi bitirince, ondan hemen önce başlamış ve bekleyen işe dönülür. Yığın bu “bekleyen işler” listesini en ucuz biçimde tutar.

Sık yapılan hata

Yığını, öğelerin istenen sırayla okunabildiği bir liste gibi düşünmek. Yığında yalnız üstteki öğe görünür. Ortadaki bir öğeye ulaşmanın tek yolu, üstündeki öğeleri tek tek çıkarmaktır. “Yığının üçüncü öğesini oku” gibi bir işlem yığın ADT'sinde yoktur.

7.1.2 Temel işlemler

Yığın ADT'si beş işlemle tanımlanır. İşlem adları İngilizce kalıplaşmıştır ve Java kütüphanesinde de aynı adlarla geçer; bu yüzden notta da aynen kullanılır. `push` “üste ekleme”, `pop` ise “üstten çıkarma” demektir.

İşlem	Açıklama
`push(x)`	Üste yeni öğe `x` ekler.
`pop()`	Üstteki öğeyi çıkarır ve döndürür.
`top()`	Üstteki öğeyi çıkarmadan döndürür (Java kütüphanesinde adı `peek()` 'tir).
`isEmpty()`	Yığın boşsa `true`, değilse `false` döndürür.
`size()`	Yığındaki öğe sayısını döndürür.

Tablo 7.1: Yığın ADT'sinin temel işlemleri.

Bu işlemlerden yalnız ikisi yığını değiştirir: `push` öğe sayısını bir artırır, `pop` bir azaltır. Öteki üçü yalnız bilgi verir. `top()` ile `pop()` arasındaki fark önemlidir: ikisi de üstteki öğeyi döndürür, ama `top()` öğeyi yerinde bırakır. Bir öğeye bakıp karar vermek, sonra gerekiyorsa çıkarmak gereken durumlarda (örneğin 7.3.9'daki sonraki büyük öğe probleminde) önce `top()`, sonra `pop()` kullanılır.

İki işlem boş yığında anlamsızdır: boş yığının üstünde öğe yoktur, dolayısıyla `pop()` da `top()` da döndürecek bir şey bulamaz. Boş yığından çıkarma denemesine **alt taşma** (*underflow*) denir ve hata sayılır. Kapasitesi sınırlı bir yığına, dolu iken ekleme denemesine ise **taşma** (*overflow*) denir. Taşma yalnız sınırlı yığınlarda görülür; bu ayrım 7.2.1'de ele alınır.

7.1.3 push ve pop adım adım

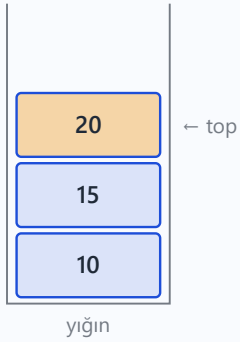
İşlemlerin birlikte nasıl çalıştığını küçük bir işlem dizisi üzerinde izleyelim. Boş bir yığına sırayla `push(10)`, `push(15)`, `push(20)`, `top()` ve dört kez `pop()` uygulanıyor.

Örnek 7.1

1. **Başlangıç.** Yığın boştur: `size = 0`, üstte öğe yoktur.

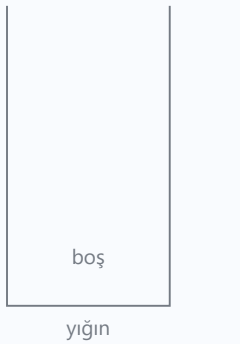
2. `push(10)` : 10 boş yığına eklenir. Tek öğe olduğu için hem en altta hem en üsttedir. `size = 1` .
3. `push(15)` : 15, 10'un üstüne konur. Artık üst 15'tir. `size = 2` .
4. `push(20)` : 20 en üste konur. Yığın alttan üste 10, 15, 20'dir. `size = 3` .
5. `top()` : Üstteki 20 okunur ve döndürülür, ama çıkarılmaz. `size` 3'te kalır.
6. `pop()` : Son giren 20 ilk çıkar ve döndürülür. Üst artık 15'tir; `size = 2` .
7. `pop()` : 15 çıkar ve döndürülür. Üst 10'dur; `size = 1` .
8. `pop()` : 10 çıkar. Yığın yeniden boştur; `size = 0` .
9. `pop()` : Boş yığından çıkarılacak öğe yoktur. Bu bir alt taşmadır ve hata olarak bildirilir.

Dönen değerler sırasıyla 20 (`top`), 20, 15, 10 ve hatadır. Çıkış sırası 20, 15, 10, giriş sırası olan 10, 15, 20'nin tam tersidir.



İşlem	Dönen değer	size
push(10)	–	1
push(15)	–	2
push(20)	–	3

alt taşma!



İşlem	Dönen değer	size
push(10)	–	1
push(15)	–	2
push(20)	–	3
top()	20	3
pop()	20	2
pop()	15	1
pop()	10	0
pop()	hata!	0

Şekil 7.1: Örnek 7.1'in iki anı. Üstte üç `push` 'tan sonraki durum: yığın alttan üste 10, 15, 20'dir ve üst (top) 20'yi gösterir. Altta işlem dizisinin sonu: üç `pop` yığını boşaltmış, dördüncü `pop` alt taşma hatası vermiştir. Sağdaki tablolar her işlemin döndürdüğü değeri ve işlemden sonraki `size` değerini gösterir.

Şekil 7.1'deki tablo, yığın sorularını çözerken kullanılabilecek pratik bir düzen önerir: her satıra bir işlem, işlemin döndürdüğü değer ve işlemden sonraki boyut yazılır. Yığının kendisi de her adımda alttan üste bir liste olarak not edilirse yanlış öğeyi çıkarma hatası pek yapılmaz.

> **Sık yapılan hata**
>
> `top()`'un yığını değiştirdiğini sanmak. `top()` yalnız okur. Yukarıdaki dizide `top()`'tan sonra ilk `pop()` yine 20'yi döndürür. `top()` çıkarma yapsaydı ilk `pop()` 15'i döndürürdü.

7.1.4 Java arayüzü: ADT ile gerçeklemeyi ayırmak

Java'da bir ADT'yi kodla ifade etmenin doğal yolu bir **arayüz** (*interface*) yazmaktır. Arayüz, bir sınıfın hangi metotları sağlaması gerektiğini söyler, ama metotların gövdesini içermez. Yığın ADT'si için arayüz şöyledir:

```
public interface Stack<E> {
  int size();
  boolean isEmpty();
  void push(E e);
  E top();    // boşsa null
  E pop();    // boşsa istisna
}
```

Arayüzün ilk satırındaki `<E>`, yığının **jenerik** (*generic*) olduğunu gösterir. `E` bir tür parametresidir: yığın tam sayı tutacaksa `Stack<Integer>`, karakter tutacaksa `Stack<Character>` yazılır ve `E` her yerde o türün yerine geçer. Böylece aynı kod her tür öğe için çalışır. Metotlar Tablo 7.1'deki işlemlerin aynısıdır. Yorumlar, boş yığındaki davranışı da sözleşmenin parçası yapar: boş yığında `top()` `null` döndürür, `pop()` ise bir **istisna** (*exception*) fırlatır.

Arayüzün yaptığı iş, "ne" sorusunu "nasıl" sorusundan ayırmaktır:

- **ADT, "ne?" sorusunu cevaplar.** Arayüz yalnız işlemleri ve anlamlarını söyler. Verinin dizide mi, bağlı listede mi saklandığından hiç söz etmez.
- **Gerçekleme, "nasıl?" sorusunu cevaplar.** `DiziYigin<E> implements Stack<E>` gibi bir sınıf, arayüzdeki her metodu belirli bir saklama biçimiyle yazar. Bağlı liste kullanan başka bir sınıf da aynı arayüzü uygulayabilir.
- **Kod arayüze göre yazılırsa gerçekleme değiştirilebilir.** Yığını kullanan bir metot parametresini `Stack<Integer>` türünde alırsa, ona dizi tabanlı ya da bağlı liste tabanlı bir yığın verilebilir. Çağıran kodda tek satır değişmez.

Bu ayrım yalnız düzen için değildir. Bir programın yığına yaptığı istekler değişmeden kalırken, ölçümler sonucunda daha hızlı bir gerçeklemeye geçmek gerekebilir. Arayüz bu değişikliği yerel tutar.

"Boş yığında ne olur?" sorusunun cevabı ise kütüphaneden kütüphaneye değişir. Aynı soyut işlem farklı sınıflarda farklı davranır:

Boş yığında	`pop()`	`top()` / `peek()`
Kendi `Stack<E>` 'imiz	istisna fırlatır	`null` döner
`java.util.Stack`	`EmptyStackException`	`EmptyStackException`
`java.util.ArrayDeque`	`NoSuchElementException`	`null` döner

Tablo 7.2: Boş yığında `pop()` ve `top()` / `peek()` davranışı.

Tablo 7.2'deki farklar, yığın kullanan kodda bir alışkanlığı gerekli kılar: `pop()` ya da `peek()` çağrılmadan önce `isEmpty()` ile yığının boş olmadığı denetlenir. Bu denetim yapılmazsa aynı kod bir kütüphaneyle istisna fırlatır, ötekiyle sessizce `null` alır ve hata çok daha ileride, anlaşılması zor bir yerde ortaya çıkar.

> **Sık yapılan hata**
>
> Kendi `Stack` arayüzümüzle `java.util.Stack` sınıfını aynı dosyada kullanmak. İkisinin adı aynıdır; dosyada `import java.util.*;` ya da `import java.util.Stack;` varsa derleyici hangi `Stack` 'in kastedildiğini karıştırabilir ya da beklenmedik sınıfı seçer. Kendi arayüzümüz kullanılıyorsa `java.util.Stack` içe aktarılmaz.

7.1.5 İşlemlerin algoritmaları

Gerçeklemeye geçmeden önce işlemleri, dili bağlamayan bir sözde kodla yazmak yararlıdır. Yığının bir dizide tutulduğunu varsayalım. Dizinin yanında bir tam sayı tutulur: `top`, üstteki öğenin *indisidir*. Dizi indisleri 0'dan başladığı için ilk öğe 0. hücreye yazılır. Boş yığında henüz geçerli bir üst yoktur; bu durum `top = -1` ile gösterilir.

push(x):

```
başla
  eğer yığın doluysa   // top = kap-1
    hata ver: taşma (overflow)
  değilse
    top değerini 1 artır
    yığın[top] ← x
son
```

Ekleme önce yer olup olmadığını sorar. Dizinin kapasitesi `kap` ise son geçerli indis `kap - 1` 'dir; `top` bu değere ulaşmışsa dizi doludur ve ekleme taşmaya yol açar. Yer varsa `top` bir artırılır ve `x` yeni üst hücreye yazılır. Sıra önemlidir: önce indis artar, sonra yazılır. Ters sırada yapılsaydı `x` mevcut üstün üzerine yazılır ve o öğe kaybolurdu.

pop():

```
başla
  eğer yığın boşsa     // top = -1
    hata ver: alt taşma (underflow)
  değilse
    x ← yığın[top]; top 1 azalt
    x değerini döndür
son
```

Çıkarma önce yığının boş olup olmadığını sorar; `top = -1` ise alt taşma vardır. Boş değilse üstteki öğe bir değişkene alınır, `top` bir azaltılır ve saklanan öğe döndürülür. Burada da sıra önemlidir: önce okunur, sonra indis azaltılır.

top():

```
başla
  eğer yığın boşsa null döndür
  değilse yığın[top] döndür
son
```

isEmpty() ve size():

```
isEmpty: top = -1 ise true, değilse false
size:    top + 1 döndür
```

`size` neden `top + 1`'dir? Öğeler 0, 1, …, `top` indislerindedir; bu aralıkta `top + 1` tane indis vardır. Örneğin üç öğeli yığında öğeler 0, 1 ve 2. hücrelerdedir, `top = 2`'dir ve `size = 3`'tür. Boş yığında `top = -1` olduğu için `size = 0` çıkar; formül boş yığında da doğrudur.

> **Sık yapılan hata**
>
> `top` ile `size`'ı karıştırmak. `top` son öğenin indisidir, öğe sayısı değildir. Bu karışıklık iki tipik hataya yol açar: dolu koşulunu `top == kap` diye yazmak (doğrusu `top == kap - 1`) ve `size()`'ı `top` döndürecek biçimde yazmak (doğrusu `top + 1`). İlk hatada dizi dolduğu hâlde bir ekleme daha denenir ve `yığın[kap]` dizinin dışına düşer.

7.1.6 Zaman ve alan maliyeti

Yığının en güçlü yanı, temel işlemlerinin hepsinin sabit sürede çalışmasıdır. Bir işlemin süresi yığındaki öğe sayısına bağlı değilse o işleme **O(1)** zamanlı denir. Tablo 7.3 her işlemin maliyetini ve gerekçesini verir.

İşlem	Zaman	Neden?
`push`	O(1)	yalnız üst değişir
`pop`	O(1)	yalnız üst değişir
`top`	O(1)	üst doğrudan bilinir
`isEmpty`, `size`	O(1)	sayaç ya da indis tutulur
bir öğeyi arama	O(n)	öğeler tek tek çıkarılmalı

Tablo 7.3: Yığın işlemlerinin zaman maliyeti (n: yığındaki öğe sayısı).

7.1.5'teki sözde kodlara bakıldığında bu sonuç açıktır. Her işlem en çok bir karşılaştırma, bir indis değişikliği ve bir dizi erişimi yapar. Dizi erişimi, indis bilindiği sürece dizinin boyutundan bağımsızdır. Hiçbir işlemde döngü yoktur; hiçbir öğe kaydırılmaz ya da taranmaz. Bir milyon öğeli yığında `push`, üç öğeli yığındakiyle aynı sayıda adım atar.

Arama ise farklıdır. Yığında belirli bir değerin olup olmadığını öğrenmek için ADT'nin sunduğu tek yol, öğeleri üstten başlayarak tek tek çıkarıp bakmaktır. En kötü durumda (aranan değer en altta ya da hiç yoksa) n öğenin hepsi çıkarılır: $O(n)$. Arama sık gerekiyorsa yığın doğru veri yapısı değildir.

Alan maliyeti de basittir. n öğeli bir yığın en az n öğeyi saklamak zorundadır, bu yüzden $O(n)$ bellek kullanır. İşlemlerin kendisi ise ek bellek gerektirmez: her işlem yalnız birkaç yerel değişken kullanır, yani $O(1)$ ek alan harcar.

Tek bir istisna vardır: kapasitesi dolunca büyüyen **dinamik dizi** (*dynamic array*) kullanan yığında, büyütme yapan `push` bütün öğeleri yeni diziye kopyalar ve $O(n)$ sürer. Büyütme seyrek yapıldığı için uzun bir işlem dizisinde `push` başına ortalama maliyet yine sabittir; buna **amortize O(1)** denir. Bu sonuç 7.2.6'da hesapla gösterilir.

Yığının hızının kaynağı kısıtlamanın kendisidir. Erişim yalnız üstle sınırlandığı için hiçbir işlem öğeleri kaydırmak ya da aramak zorunda kalmaz. Ortadan ekleme ya da istenen indisteki öğeyi okuma gibi işlemlere izin veren bir liste daha esnektir, ama bu esnekliğin bedeli bazı işlemlerin $O(n)$ sürmesidir.

7.2 Gerçekleme

Bir ADT'yi **gerçeklemek**, onu belirli bir saklama biçimiyle çalışan bir sınıf olarak yazmak demektir. Yığın için iki temel seçenek vardır: öğeleri bir dizide ya da birbirine bağlı düğümlerden oluşan bir listede tutmak. Bu kesim iki seçeneği de kod düzeyinde inceler, ardından ikisini karşılaştırır.

7.2.1 Sabit boyutlu ve dinamik boyutlu yığın

Gerçeklemeye geçmeden önce yığının kapasitesi hakkında bir karar vermek gerekir. Yığın en çok kaç öğe tutabilecek? Bu soruya iki farklı cevap, iki tür yığın doğurur.

Sabit boyutlu yığın (*fixed size stack*) kapasitesini oluşturulurken belirler; sonradan büyüyemez, küçülemez. Arka planda sabit uzunlukta bir dizi kullanır. Dolu yığına ekleme denemesi taşma, boş yığından çıkarma denemesi alt taşma hatasıdır.

Dinamik boyutlu yığın (*dynamic size stack*) ise çalışma sırasında büyüyüp küçülebilir. Dolduğunda kapasitesini kendiliğinden artırır, örneğin iki katına çıkarır. Öğe sayısı kapasiteye göre çok azalırsa belleği geri vermek için küçülebilir de. Dinamik dizi ya da bağlı liste ile gerçeklenir.

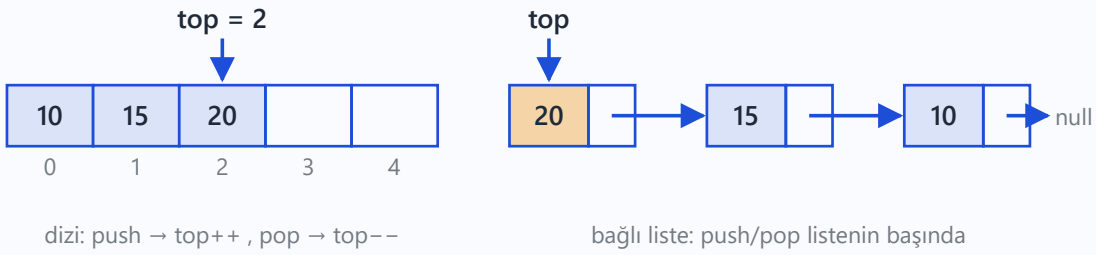


Şekil 7.2: Sabit ve dinamik boyutlu yığın farkı. Solda kapasitesi 4 olan dizi doludur (10, 20, 30, 40); `push(50)` taşma hatası verir. Sağda aynı durumda dinamik yığın iki katı kapasiteli (8 hücrelik) yeni bir dizi ayırır, eski öğeleri kopyalar ve 50'yi yeni dizinin 4. indisine yazar.

İki tür arasındaki fark yalnız taşma durumunda ortaya çıkar. Alt taşma ise her iki türde de hatadır: dinamik yığın büyüyebilir, ama boş bir yığından öğe çıkarmanın hiçbir anlamı yoktur. Kısacası taşma yalnız sabit boyutlu yığında görülür; alt taşma her yığında görülebilir.

7.2.2 İki gerçekleme yolu

Yığın ADT'si bir **dizi** (*array*) ya da bir **bağlı liste** (*linked list*) ile gerçeklenebilir. Şekil 7.3 aynı üç öğeli yığını (alttan üste 10, 15, 20) iki biçimde gösterir.



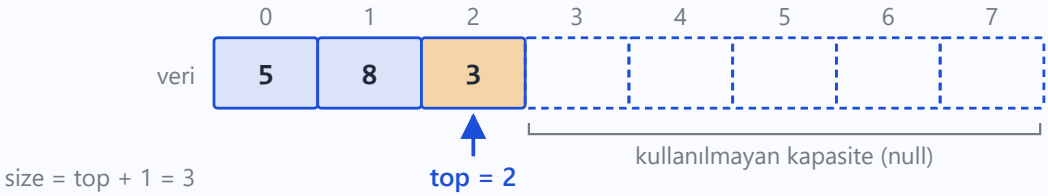
Şekil 7.3: Aynı yığının iki gerçeklemesi. Solda dizi tabanlı yığın: öğeler 0, 1, 2 indislerindedir ve `top = 2`'dir; `push` `top`'u artırır, `pop` azaltır. Sağda bağlı liste tabanlı yığın: `top` 20 düğümünü gösterir, zincir 20 → 15 → 10 → null biçimindedir; `push` ve `pop` listenin başında yapılır.

Dizi tabanlı yığında `push` işlemi `top` indisini artırır ve öğeyi `dizi[top]` hücresine yazar. `pop` işlemi `dizi[top]`'u alır, sonra `top`'u azaltır. Yığının dibi her zaman dizinin 0. hücresidir ve yığın dizinin sonuna doğru büyür.

Bağlı liste tabanlı yığında ise üst, listenin **başıdır** (*head*). `push` yeni düğümü listenin başına ekler; `pop` baştaki düğümü çıkarır ve değerini döndürür. Dikkat edilirse iki şekilde okların ve indislerin yönü farklıdır, ama ADT açısından iki yığın aynıdır: ikisinde de üstte 20 vardır, bir `pop` ikisinde de 20'yi döndürür.

7.2.3 Dizi tabanlı yığın

Kapasitesi oluşturulurken belirlenen ve sonra değişmeyen dizi tabanlı yığına **sınırlı yığın** (*bounded stack*) da denir. Şekil 7.4 kapasitesi 8 olan bir dizide üç öğenin (5, 8, 3) nasıl durduğunu gösterir.



Şekil 7.4: Kapasitesi 8 olan `veri` dizisinde 5, 8, 3 saklanıyor. `top = 2` son öğeyi gösterir; öğe sayısı `size = top + 1 = 3` 'tür. 3–7. hücreler kullanılmayan kapasitedir ve `null` içerir.

Dizi tabanlı yığının avantajları şunlardır:

- **Basittir.** Bir dizi ve bir tam sayı yeterlidir; kod birkaç satırdır.
- **Bellek tasarrufu sağlar.** Her öğe için yalnız öğenin kendisi saklanır. Bağlı listedeki gibi düğüm başına ek bir referans tutulmaz.
- **Bellek kullanımı sıkı biçimde denetlenir.** Yığın, kapasitesinden fazla bellek isteyemez. Kötü niyetli bir girdinin yığını sonsuza dek büyütüp belleği doldurmaya çalıştığı saldırılara karşı bu sınır bir güvencedir.
- **Önbellek dostudur.** Öğeler bellekte ardışık hücrelerdedir. İşlemci belleği bloklar hâlinde okuduğu için ardışık veriye erişim, dağınık düğümlere erişimden hızlıdır. Bu yüzden aynı *O(1)* işlemler dizi tabanlı yığında pratikte genellikle daha hızlı çalışır.

Dikkat edilmesi gereken noktalar da vardır:

- Eklemeden önce yığının dolu olmadığı denetlenmelidir. Denetim yapılmazsa yazma dizinin dışına düşer ve Java `ArrayIndexOutOfBoundsException` fırlatır.
- Kapasite seçimi bir uzlaşmadır. Az seçilirse taşma yaşanır, çok seçilirse kullanılmayan hücreler belleği boşa tutar.
- Nesne tutan dizide, çıkarılan öğenin hücresine `null` atanmalıdır. Atanmazsa dizi o nesneyi göstermeye devam eder ve nesne bellekte gereksiz yere kalır.

7.2.4 Kod: DiziYigin

Aşağıdaki sınıf, 7.1.4'teki `Stack<E>` arayüzünü sabit kapasiteli bir diziyle uygular.

```
public class DiziYigin<E> implements Stack<E> {
    private E[] veri;
    private int top = -1;                    // -1 → boş yığın
    public DiziYigin(int kapasite) { veri = (E[]) new Object[kapasite]; }
    public boolean isEmpty() { return top == -1; }
    public int size()        { return top + 1; }
    public E top()           { return isEmpty() ? null : veri[top]; }
    public void push(E x) {
        if (top == veri.length - 1) throw new IllegalStateException("Taşma");
        veri[++top] = x;
    }
    public E pop() {
        if (isEmpty()) throw new IllegalStateException("Alt taşma");
        E x = veri[top];
        veri[top--] = null;                  // referansı bırak
        return x;
    }
}
```

Kod satır satır şöyle çalışır:

- `veri`, öğeleri tutan dizidir; `top` üstteki öğenin indisidir ve `-1` ile başlar, çünkü yeni oluşturulan yığın boştur.
- Yapıcı, istenen kapasitede bir dizi ayırır. Java'da `new E[kapasite]` yazılamaz: tür parametresi `E` çalışma zamanında bilinmediği için jenerik dizi doğrudan oluşturulamaz. Bunun yerine bir `Object` dizisi oluşturulur ve `(E[])` ile dönüştürülür. Derleyici bu dönüşüm için "unchecked" (denetlenmemiş) uyarısı verir; dizi sınıfın dışına hiç verilmediği sürece bu dönüşüm güvenlidir.
- `isEmpty()` yalnız `top == -1` sorusunu sorar; `size()` `top + 1` döndürür (7.1.5).
- `top()` metodu yığın boşsa `null`, değilse `veri[top]` döndürür. Burada `? :` koşul işlecidir: koşul doğruysa ilk, yanlışsa ikinci değeri verir. Sınıfta hem `top` alanı hem de `top()` metodu vardır; Java ikisini parantezlerden ayırt eder.
- `push(x)` önce doluluğu sorar: `top == veri.length - 1` ise son hücre de doludur ve `IllegalStateException("Taşma")` fırlatılır. Yer varsa `veri[++top] = x;` çalışır. `++top` *ön artırma*dır: önce `top` bir artar, sonra artmış değer indis olarak kullanılır. Böylece 7.1.5'teki "önce artır, sonra yaz" sırası tek satıra sığar.
- `pop()` önce boşluğu sorar; boşsa "Alt taşma" istisnası fırlatılır. Değilse üstteki öğe `x`'e alınır. `veri[top--] = null;` satırında `top--` *son azaltma*dır: önce `top`'un eski değeri indis olarak kullanılır ve o hücreye `null` yazılır, sonra `top` bir azalır. Son olarak saklanan `x` döndürülür.

`++top` ile `top++` arasındaki fark burada belirleyicidir. `push`'ta `veri[top++] = x` yazılsaydı öğe eski `top` indisine, yani mevcut üstün üzerine yazılırdı. `pop`'ta `veri[--top] = null` yazılsaydı üstteki değil, onun altındaki hücre silinirdi.

Bütün metotlar döngüsüzdür ve sabit sayıda adım atar; her biri *O(1)* zamanlıdır. Sınıf, kapasite kadar hücrelik bir dizi kullanır: alan *O(kapasite)*'dir.

Sık yapılan hata

Çıkarılan hücreyi boşaltmamak. `pop`'ta yalnız `top--` yazılıp `null` atanmazsa metot doğru değeri döndürür, ama dizi çıkarılan nesneyi göstermeye devam eder. Java'da belleği **çöp toplayıcı** (*garbage collector*) geri alır ve bunu yalnız hiçbir referansın göstermediği nesneler için yapar. Yığının artık kullanmadığı ama dizinin hâlâ gösterdiği nesne temizlenemez. Bu tür bellek sızıntısına İngilizcede *loitering* (oyalanma) denir. `int[]` gibi ilkel tür tutan dizide referans olmadığı için bu sorun yoktur.

7.2.5 Kod izleme: DiziYigin

Sınıfın davranışını, kapasitesi 4 olan bir yığın üzerinde izleyelim. `main` metodu sırayla şu çağrıları yapıyor: `y.push(10)`, `y.push(15)`, `y.push(20)`, `y.top()`, `y.pop()`, `y.push(25)`, `y.push(30)`, `y.push(40)`. İzlemede kullanılan metot gövdeleri şunlardır:

```
public void push(E x) {
  if (top == veri.length - 1)
    throw new IllegalStateException("Taşma");
  veri[++top] = x;
}
public E pop() {
  if (isEmpty())
    throw new IllegalStateException("Alt taşma");
  E x = veri[top];
  veri[top--] = null;          // referansı bırak
  return x;
}
public E top() { return isEmpty() ? null : veri[top]; }
```

Çağrı	Doluluk/boşluk denetimi	`top`	`veri`	Döner
(başlangıç)	—	−1	[null, null, null, null]	—
`push(10)`	−1 == 3? hayır	0	[10, null, null, null]	—
`push(15)`	0 == 3? hayır	1	[10, 15, null, null]	—
`push(20)`	1 == 3? hayır	2	[10, 15, 20, null]	—
`top()`	boş mu? hayır	2	[10, 15, 20, null]	20
`pop()`	2 == −1? hayır	1	[10, 15, null, null]	20

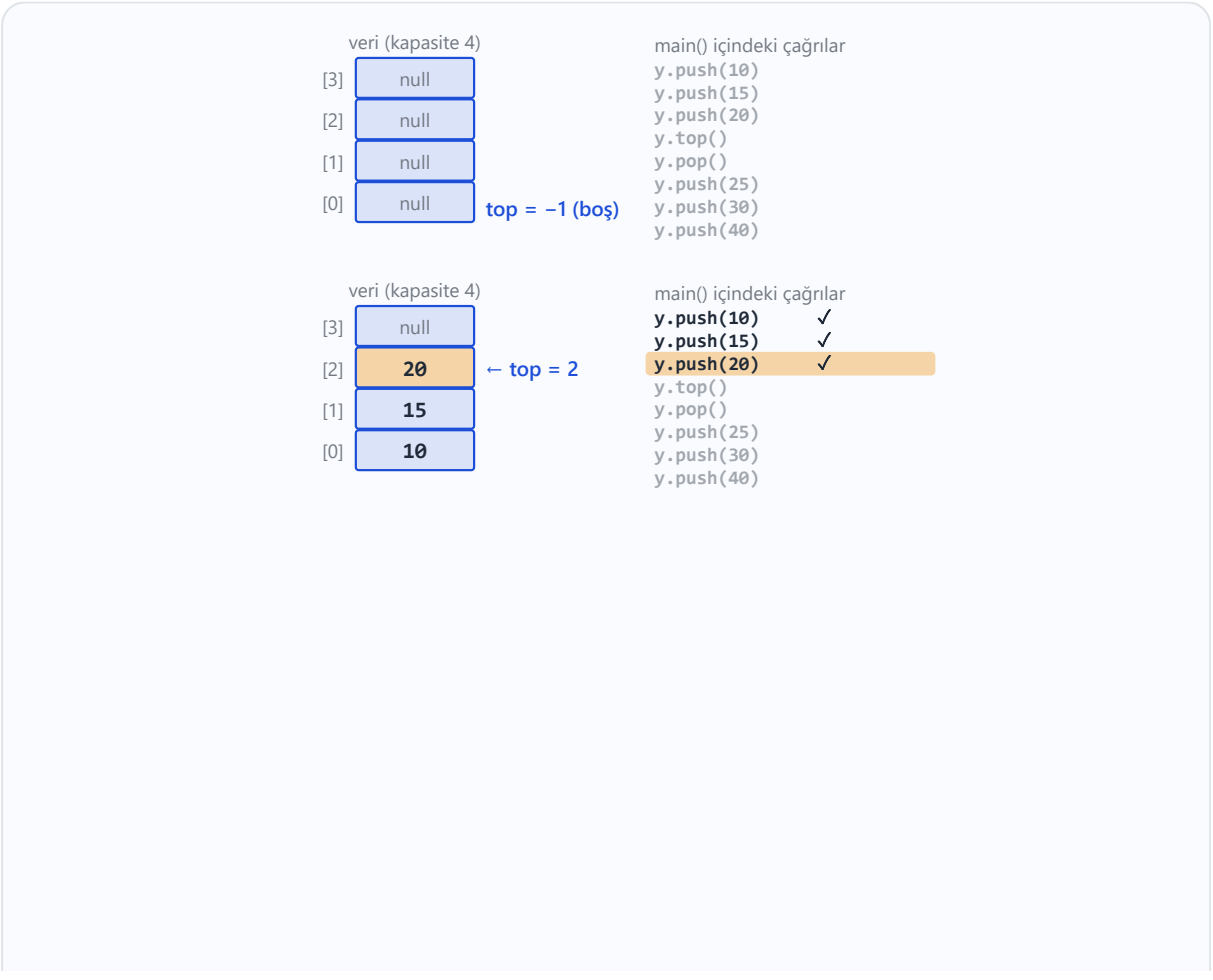
Çağrı	Doluluk/boşluk denetimi	`top`	`veri`	Döner
`push(25)`	1 == 3? hayır	2	[10, 15, 25, null]	—
`push(30)`	2 == 3? hayır	3	[10, 15, 25, 30]	—
`push(40)`	3 == 3? **evet**	3	[10, 15, 25, 30]	istisna: “Taşma”

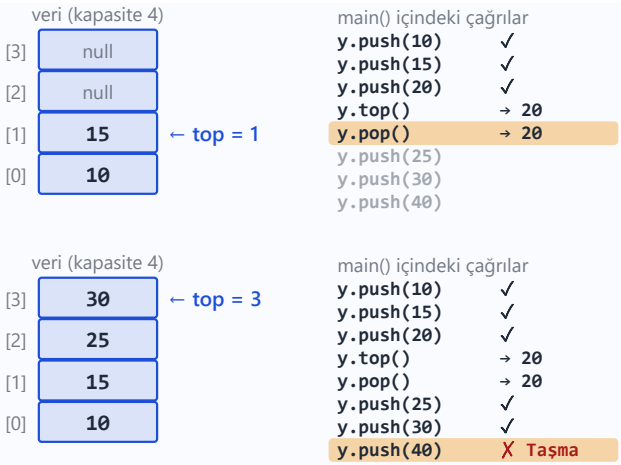
Tablo 7.4: `new DiziYigin<>(4)` üzerinde çağrı dizisinin izlemesi. `veri` 0'dan 3'e yazılmıştır.

İzlemenin önemli anları şunlardır. İlk `push(10)`'da denetim `top = -1` ile `veri.length - 1 = 3`'ü karşılaştırır; eşit olmadıkları için devam edilir. `veri[++top] = x` önce `top`'u −1'den 0'a çıkarır, sonra `veri[0] = 10` yazar. Bu tek bir atamadır; dizi ne kadar büyük olursa olsun süre değişmez.

`top()` çağrısı yığın boş olmadığı için `veri[2] = 20`'yi döndürür; dizi ve `top` değişmez. Hemen ardından gelen `pop()` önce `isEmpty()` ile `top = 2 ≠ -1` olduğunu görür. Sonra `x = veri[2] = 20` saklanır, `veri[top--] = null` ile 2. hücre boşaltılır ve `top` 1'e iner. Metot 20'yi döndürür: son giren ilk çıkmıştır.

`push(25)`, boşalan 2. hücreyi yeniden kullanır. `push(30)` ile son hücre de dolar ve `top = 3` olur. `push(40)`'ta denetim `3 == 3` sonucunu verir: dizi doludur, `IllegalStateException("Taşma")` fırlatılır. Dizi ve `top` değişmeden kalır. Sabit kapasiteli yığında taşma budur; dinamik dizi aynı noktada diziyi büyütürdü.





Şekil 7.5: İzlemenin dört anı (soldan sağa, yukarıdan aşağı). (a) Başlangıç: dört hücre `null`, `top = -1`. (b) Üç `push`'tan sonra `top = 2`. (c) `pop()` 20'yi döndürmüş, 2. hücre `null` olmuş, `top = 1`. (d) `push(25)` ve `push(30)`'dan sonra dizi dolu, `top = 3`; `push(40)` taşma verir.

Şekil 7.5'in (c) ve (d) karelerinde dikkat çekici bir nokta vardır: 2. hücre önce 20'yi tutmuş, `pop` sonrası `null` olmuş, sonra 25 ile yeniden dolmuştur. Dizi tabanlı yığın hücreleri yeniden kullanır; bir öğe çıktığında dizinin geri kalanı kaydırılmaz.

7.2.6 Dinamik dizi: dolunca iki katına büyütme

Sabit kapasite, kaç öğe geleceği önceden bilinmeyen durumlarda sıkıntı yaratır. Çözüm, dizi dolduğunda daha büyük bir dizi ayırmak ve öğeleri oraya taşımaktır. Java'da bir dizinin uzunluğu sonradan değiştirilemez; bu yüzden "büyütme" aslında yeni bir dizi oluşturup kopyalamak demektir.

```
public void push(E x) {
  if (top == veri.length - 1)
    buyut(2 * veri.length);
  veri[++top] = x;
}
private void buyut(int yeniKap) {
  E[] yeni = (E[]) new Object[yeniKap];
  for (int i = 0; i <= top; i++)
    yeni[i] = veri[i];
  veri = yeni;
}
```

Yeni `push`, taşma istisnası fırlatmak yerine `buyut(2 * veri.length)` çağırır. `buyut` metodu istenen kapasitede yeni bir dizi ayırır, döngüyle 0'dan `top`'a kadar bütün öğeleri aynı indislere kopyalar ve `veri` referansını yeni diziye yöneltir. Eski diziyi artık hiçbir referans göstermez; çöp toplayıcı onu temizler. Büyütmeden sonra `veri[++top] = x` her zamanki gibi çalışır.

Büyütme yapan bir `push`'un maliyeti büyüktür: n öğe kopyalanır, yani o tek `push` $O(n)$ sürer. Buna karşın büyütmeler seyrektir. Kapasite her seferinde iki katına çıktığı için bir büyütmeden sonra, bir sonraki büyütmeye kadar o ana kadarki öğe sayısı kadar ucuz `push` yapılabilir. Pahalı işlemlerin maliyeti, aradaki çok sayıda ucuz işleme yayılır.

Tanım 7.2

Bir işlemin **amortize maliyeti**, uzun bir işlem dizisinin toplam maliyetinin işlem sayısına bölümüdür. Tek bir işlemin en kötü durum maliyetinden farklı olarak, ara sıra görülen pahalı işlemleri ucuz işlemlerle birlikte ortalar. Bu ortalama olasılığa dayanmaz: *her* işlem dizisi için geçerli bir üst sınırdır. Böyle bir sınırı bulma yöntemine **amortize analiz** (*amortized analysis*) denir.

Örnek 7.2

Kapasitesi 4 olan boş bir dinamik dizi yığınına 17 kez `push` yapılıyor. Her `push` bir yazma yapar; büyütme yapan `push` ayrıca o anki bütün öğeleri kopyalar. Kaç kopyalama yapılır, toplam maliyet nedir?

1. **1–4. push:** Dizide yer vardır; yalnız yazma yapılır. 4. `push`'tan sonra dizi doludur.
2. **5. push:** `top == 3 == veri.length - 1`; kapasite 4'ten 8'e çıkar ve 4 öğe kopyalanır. Sonra 5. öğe yazılır.
3. **6–8. push:** Yer vardır; yalnız yazma.
4. **9. push:** Dizi 8 öğeyle doludur; kapasite 16'ya çıkar, 8 öğe kopyalanır.
5. **10–16. push:** Yalnız yazma.
6. **17. push:** 16 öğe kopyalanır, kapasite 32 olur.
7. **Toplam.** Kopya: 4 + 8 + 16 = 28. Yazma: 17. Toplam 45 temel adım; bu değer $3n = 51$ sınırının altındadır. `push` başına ortalama 45 / 17 ≈ 2,65 adımdır.

Örnekteki gözlem genelleşir. Kopya sayıları 4, 8, 16 gibi her biri öncekinin iki katı olan bir dizidir. Böyle bir dizinin toplamı son terimin iki katından küçüktür. Son büyütmede kopyalanan öğe sayısı da n'den küçük olduğu için bütün kopyaların toplamı $2n$'yi geçmez.

Teorem 7.1 (İki katına büyütmenin amortize maliyeti)

Kapasitesi dolunca iki katına çıkarılan dinamik dizi yığınında, boş yığından başlayan n `push`'un toplam maliyeti $3n$'den azdır. Dolayısıyla `push`'un amortize maliyeti $O(1)$'dir.

İspat. Başlangıç kapasitesi 1 olsun (daha büyük bir başlangıç kapasitesi kopya sayısını yalnız azaltır). Büyütmeler dizi 1, 2, 4, …, 2^k öğeyle dolduğunda yapılır; burada 2^k, ikinin n'den küçük en büyük kuvvetidir. Kopya sayısı $1 + 2 + 4 + \ldots + 2^k = 2^{k+1} - 1$'dir. $2^k < n$ olduğundan bu toplam $2n - 1$'den, dolayısıyla $2n$'den küçüktür. Her `push` ayrıca bir yazma yapar: n yazma. Toplam $< 2n + n = 3n$'dir. Bölünce `push` başına ortalama 3'ten küçük, yani sabittir. □

Teoremin söylemediği şeye de dikkat etmek gerekir: tek bir `push` yine $O(n)$ sürebilir. Örnek 7.2'deki 17. `push` 16 kopya yapmıştır. Bir oyun ya da gerçek zamanlı bir sistemde her işlemin kısa sürmesi gerekiyorsa amortize sınır yetmez; en kötü durum sınırı önemlidir.

Kapasiteyi oranla değil de sabit bir miktarla büyütmek çok daha kötü sonuç verir. Kapasite 10'dan başlayıp her dolumda 10 hücre artırılsın (10 → 20 → 30 → …). Büyütmeler 10, 20, 30, … öğede yapılır ve kopya toplamı $10 + 20 + 30 + \ldots$ olur. $n = 100$ için bu toplam $10 + 20 + \ldots + 90 = 450$'dir; genel olarak yaklaşık $n^2/20$'dir. Toplam maliyet $O(n^2)$, `push` başına ortalama $O(n)$ olur. Karşılaştırma için iki katına büyütmede 1000 `push` 1023 kopya yaparken, 10'ar büyütmede 49.500 kopya yapar.

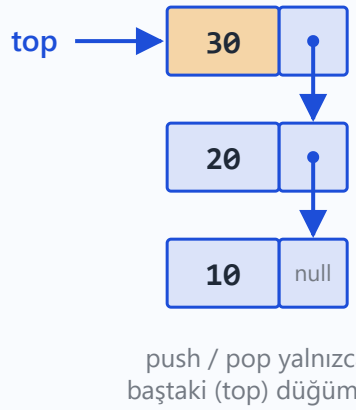
> **Sık yapılan hata**
>
> "Amortize $O(1)$" ile "her işlem $O(1)$" ifadelerini aynı sanmak. Dinamik dizide `push` amortize $O(1)$'dir, ama büyütme yapan `push` $O(n)$'dir. İkinci hata da kapasiteyi sabit miktarla büyütmektir: o zaman amortize maliyet bile $O(n)$'e çıkar. Kapasite oranla (örneğin ×2) büyütülmelidir.

7.2.7 Bağlı liste tabanlı yığın

İkinci gerçeklemede öğeler birbirine bağlı **düğümlerden** (*node*) oluşan bir zincirde tutulur. Her düğüm iki alan taşır: öğenin değeri ve zincirdeki bir sonraki düğüme referans.

```
class Dugum {
    int veri;
    Dugum sonraki;
    Dugum(int veri) { this.veri = veri; }   // sonraki = null
}
```

`Dugum` sınıfında `veri` değeri, `sonraki` ise bir alttaki düğümü gösterir. Yapıcı yalnız `veri`'yi atar; Java'da atanmayan referans alanları varsayılan olarak `null`'dır, bu yüzden yeni düğümün `sonraki` alanı başta `null`'dır.



Şekil 7.6: Bağlı liste ile yığın. `top` 30'u, 30 20'yi, 20 10'u gösterir; 10'un `sonraki` alanı `null`'dır. `push` ve `pop` yalnız baştaki (top) düğümde yapılır.

Bu gerçeklemenin özellikleri şöyle sıralanabilir:

- **Esnek boyut.** Her `push` yeni bir düğüm ayırır. Boyut çalışma zamanında ihtiyaca göre değişir; bellek bitmedikçe taşma olmaz.
- **Ek bellek.** Her düğüm, değerin yanında bir `sonraki` referansı da tutar. Küçük öğelerde (örneğin bir `int`) referans, öğenin kendisi kadar ya da ondan fazla yer kaplayabilir.
- **Rastgele erişim yok, ama üste erişim hızlı.** Bağlı listede *k*. öğeye ulaşmak için baştan *k* adım yürümek gerekir. Yığın ise yalnız üstle ilgilenir ve üst, listenin başıdır: `top` referansı onu doğrudan gösterir.
- **Ekleme ve çıkarma *O(1)*.** İkisi de listenin başında yapılır ve yalnız birkaç referansı değiştirir.

Neden üst listenin başında, sonunda değil? **Tek yönlü bağlı listede** (*singly linked list*) her düğüm yalnız sonrakini bilir. Sondaki düğümü silmek için, sondan bir önceki düğümün `sonraki` alanını `null` yapmak gerekir. Ama o düğüme ulaşmanın tek yolu baştan başlayıp listeyi yürümektir: *O(n)*. Baştaki düğümü silmek ise yalnız baş referansını bir ilerletmektir: *O(1)*. Yığının üstünü başa koymak, iki işlemi de sabit süreli yapar.

7.2.8 Kod izleme: bağlı listede push

Bağlı liste yığınının alanları ve `push` metodu şöyledir:

```
Dugum top;         // yığının üstü
int uzunluk;

public void push(int veri) {
  Dugum gecici = new Dugum(veri);
  gecici.sonraki = top;
  top = gecici;
  uzunluk++;
}
```

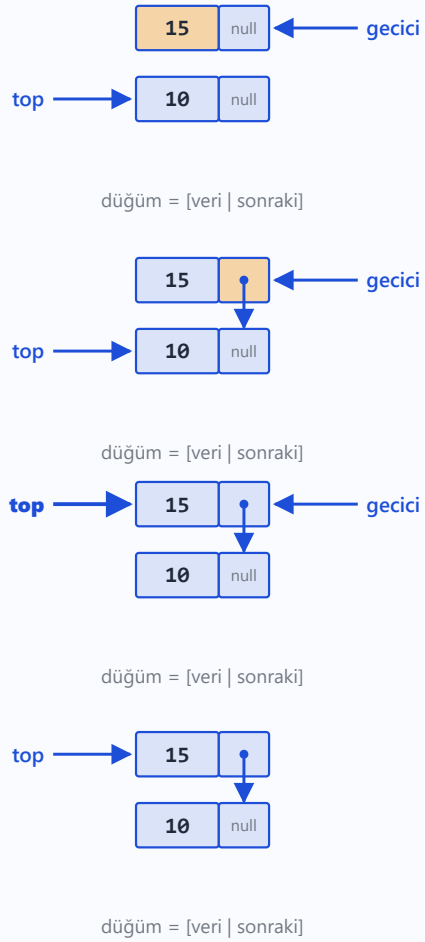
`top` yığının üstündeki düğümü, `uzunluk` öğe sayısını tutar. Boş yığında `top = null` ve `uzunluk = 0`'dır. `push` dört adımda çalışır: yeni düğüm oluşturulur ve yerel `gecici` değişkeni onu gösterir; yeni düğümün `sonraki` alanı eski üste bağlanır; `top` yeni düğüme yöneltilir; `uzunluk` bir artırılır. Boş yığına `push(10)` ve ardından `push(15)` yapıldığını izleyelim.

Satır	`gecici`	`top`	Zincir	`uzunluk`
(başlangıç)	—	null	null	0
`gecici = new Dugum(10)`	düğüm(10), sonraki = null	null	null	0
`gecici.sonraki = top`	düğüm(10) → null	null	null	0
`top = gecici`	düğüm(10)	düğüm(10)	10 → null	0
`uzunluk++`	(metot biter)	düğüm(10)	10 → null	1
`gecici = new Dugum(15)`	düğüm(15), sonraki = null	düğüm(10)	10 → null	1
`gecici.sonraki = top`	düğüm(15) → düğüm(10)	düğüm(10)	10 → null	1
`top = gecici`	düğüm(15)	düğüm(15)	15 → 10 → null	1
`uzunluk++`	(metot biter)	düğüm(15)	15 → 10 → null	2

Tablo 7.5: Boş yığına `push(10)` ve `push(15)` izlemesi. "Zincir" sütunu `top`'tan başlayarak düğümleri sıralar.

İlk `push(10)`'da `top` `null` olduğu için `gecici.sonraki = top` satırı `sonraki`'yi yine `null` bırakır: yeni düğüm tek başınadır. Metot bittiğinde yerel `gecici` değişkeni ortadan kalkar, ama düğüm kaybolmaz; artık `top` üzerinden erişilir.

İkinci `push`'ta 15 düğümü oluşturulduğu anda henüz listeye bağlı değildir; `top` hâlâ 10'u gösterir. `gecici.sonraki = top` ile 15, eski üst olan 10'a bağlanır. Ancak bundan sonra `top = gecici` ile üst değiştirilir. Hiçbir öğe kaydırılmaz; yalnız iki referans değişir. Bu yüzden `push` *O(1)*'dir.



Şekil 7.7: `push(15)` 'in adımları. (a) Yeni düğüm 15 oluşturulmuş, `gecici` onu gösteriyor; `top` hâlâ 10'da. (b) `gecici.sonraki = top` : 15, 10'a bağlandı. (c) `top = gecici` : yeni üst 15. (d) Metot bitti: yığın üstten alta 15 → 10.

Sık yapılan hata

`push` 'ta iki satırın sırasını değiştirmek: önce `top = gecici` , sonra `gecici.sonraki = top` . İlk satırdan sonra `top` zaten yeni düğümü gösterir; ikinci satır yeni düğümün `sonraki` alanını kendisine yöneltir. Eski liste hiçbir referansla gösterilmediği için kaybolur ve yığın kendine dönen tek bir düğüme dönüşür. Kural: önce yeni düğüm eski listeye bağlanır, sonra üst değiştirilir.

7.2.9 Kod izleme: bağlı listede pop

`pop` metodu ve onu üç kez çağıran döngü şöyledir:

```
public int pop() {
  if (top == null)
    throw new RuntimeException("boş");
  int sonuc = top.veri;
  top = top.sonraki;
  uzunluk--;
  return sonuc;
}
// main içinde (yığın: 15 → 10):
for (int k = 0; k < 3; k++)
  System.out.println(pop());
```

`pop` önce yığının boş olup olmadığını `top == null` ile denetler; boşsa `RuntimeException("boş")` fırlatır. Değilse üstteki değeri `sonuc`'a saklar, `top`'u bir alttaki düğüme ilerletir, `uzunluk`'u azaltır ve saklanan değeri döndürür. Değerin, düğüm zincirden koparılmadan *önce* saklanması gerekir; `top` ilerledikten sonra eski üst düğüme ulaşmanın yolu kalmaz.

Çağrı	`top == null` ?	`sonuc`	Zincir (sonra)	`uzunluk`	Yazdırılan
1. `pop()`	hayır	15	10 → null	1	15
2. `pop()`	hayır	10	null	0	10
3. `pop()`	**evet**	—	null	0	istisna: "boş"

Tablo 7.6: Yığın 15 → 10 iken döngünün üç `pop()` çağrısı.

İlk çağrıda `top = top.sonraki` satırı üstü 10 düğümüne indirir. Eski üst düğüm (15) artık hiçbir referansla gösterilmez; çöp toplayıcı onu daha sonra temizler. Dizi tabanlı yığındaki gibi elle `null` atamaya gerek yoktur, çünkü düğümü gösteren tek referans zaten `top`'tu ve o ilerlemiştir. İkinci çağrıda `top` `null`'a iner ve yığın boşalır. Üçüncü çağrı boş yığına rastlar ve istisna fırlatılır; program, istisna yakalanmadığı için burada durur.





Şekil 7.8: `pop()` 'un adımları. (a) `sonuc = top.veri` : 15 saklandı. (b) `top = top.sonraki` : üst 10'a indi; soluk çizilen 15 düğümüne artık referans yok. (c) İkinci `pop` 'ta `top` `null` 'a indi, yığın boş. (d) Üçüncü `pop` boş yığına rastlar: alt taşma istisnası.

Boşluk denetimi olmasaydı üçüncü çağrıda `top.veri` ifadesi `null` bir referans üzerinden alana erişmeye çalışırdı ve Java `NullPointerException` fırlatırdı. Her iki durumda da program durur; ama açıklayıcı bir istisna hatanın nedenini hemen söyler, `NullPointerException` ise yalnızca belirtisini gösterir.

7.2.10 Dizi mi, bağlı liste mi?

İki gerçeklemenin de temel işlemleri *O(1)*'dir. Seçim, bellek kullanımı ve kapasite davranışı gibi ikinci derece ölçütlere göre yapılır.

Ölçüt	Dizi tabanlı	Bağlı liste tabanlı
Boyut	sabit kapasite (dinamik dizide büyütülür)	esnek, çalışma zamanında değişir
`push` / `pop`	O(1) (büyütmede amortize O(1))	O(1)
Taşma	taşma mümkün	yalnız bellek biterse
Öğe başına bellek	yalnız veri	veri + `sonraki` referansı
Boş yer	kullanılmayan kapasite	yok
Java	`ArrayDeque` , `Stack` (`Vector` tabanlı)	`LinkedList` (`Deque` olarak)

Tablo 7.7: Dizi tabanlı ve bağlı liste tabanlı yığının karşılaştırması.

Tabloda bir takas görülür. Dizi tabanlı yığın öğe başına daha az bellek kullanır, ama kapasitenin boş kalan kısmı israf olabilir. Bağlı liste boş kapasite tutmaz, ama her öğe için ek bir referans taşır. Dinamik dizi en kötü durumda bir `push` 'ta *O(n)* kopyalama yapar; bağlı liste her `push` 'ta yeni bir nesne ayırır ve bu da sabit ama küçük olmayan bir maliyettir. Pratikte önbellek etkisi yüzünden dizi tabanlı yığın genellikle daha hızlıdır.

Java'da yeni kodda yığın için önerilen yol `Deque` arayüzü ile `ArrayDeque` sınıfıdır:

```
Deque<Integer> y = new ArrayDeque<>();
```

Bu nesne `push`, `pop` ve `peek` metotlarını sunar. `Deque` (çift uçlu kuyruk) her iki uçtan ekleme ve çıkarma destekler; yığın olarak kullanıldığında yalnız baş uç kullanılır. Eski `java.util.Stack` sınıfı `Vector`'dan türer ve bütün metotları senkronizedir: birden çok iş parçacığının aynı anda erişimine karşı her çağrıda kilit alır. Tek iş parçacıklı programlarda bu kilit gereksiz bir yavaşlıktır. Ayrıca `Stack`, `Vector`'dan kalıtım yoluyla istenen indise ekleme gibi yığına yabancı işlemleri de dışarı açar. Notun bundan sonraki örneklerinde bu yüzden `ArrayDeque` kullanılır.

7.3 Yığın uygulamaları

Yığının değeri, uygulamalarında ortaya çıkar. Bu kesimdeki problemler birbirinden çok farklı görünür: tarayıcı geçmişi, fonksiyon çağrıları, parantezler, sayı dizileri. Ama hepsinde aynı soru sorulur: "En son görülen ve henüz sonuçlanmamış öğe hangisi?" Bu soruyu sabit sürede cevaplayan yapı yığındır.

7.3.1 Yığının kullanıldığı yerler

Yığın, bilgisayar biliminin hemen her alanında karşımıza çıkar. Başlıca kullanım yerleri şunlardır:

- **İfadeler.** Ara ek, önek ve sonek gösterimler arasında dönüşüm, sonek ifadelerin değerlendirilmesi ve parantez kontrolü (7.5 ve 7.3.5).
- **Geri al / yinele.** Metin düzenleyicilerde *undo* ve *redo* işlemleri.
- **Tarayıcı geçmişi.** "Geri" ve "ileri" düğmeleri iki yığınla yönetilir.
- **Çağrı yığını.** Fonksiyon çağrıları, yerel değişkenler ve dönüş adresleri.
- **Geri izleme** (*backtracking*). Labirentte yol bulma, n-vezir ve Şövalye Turu gibi problemlerde, çıkmaza girilince en son karar noktasına dönmek.
- **Algoritmalar.** Hanoi kuleleri, ağaç gezinme, stock span ve histogram problemleri.
- **Çizge algoritmaları. Derinlik öncelikli arama** (*depth-first search*, DFS), topolojik sıralama, güçlü bağlantılı bileşenler.
- **Dizgi ters çevirme.** Karakterler sırayla eklenir, ters sırayla çıkar.
- **Sonraki büyük öğe.** Bir dizideki her öğe için sağındaki ilk büyük değer.

Listenin ortak yönü şudur: işler iç içe başlar ve başlama sırasının tersiyle biter. Geri izlemede en son verilen karar ilk geri alınır; parantezlerde en son açılan ilk kapanır; çağrılarda en son çağrılan ilk döner. Bu iç içelik, LIFO düzeninin doğal karşılığıdır.

7.3.2 Tarayıcıda geri ve ileri: iki yığın

Bir web tarayıcısı "geri" ve "ileri" düğmelerini iki yığınla yönetir. *Geri yığını*, şu anki sayfadan önce ziyaret edilen sayfaları; *ileri yığını*, geri gidilerek bırakılan sayfaları tutar. Şu anki sayfa iki yığının da dışında, ayrı bir değişkende durur. Üç işlem şu kurallarla çalışır:

- `ziyaret(s)`: şu anki sayfa geri yığınına `push` edilir, ileri yığını temizlenir, `s` şu anki sayfa olur.
- `geri()`: şu anki sayfa ileri yığınına `push` edilir, geri yığınından `pop` edilen sayfa şu anki sayfa olur.
- `ileri()`: şu anki sayfa geri yığınına `push` edilir, ileri yığınından `pop` edilen sayfa şu anki sayfa olur.

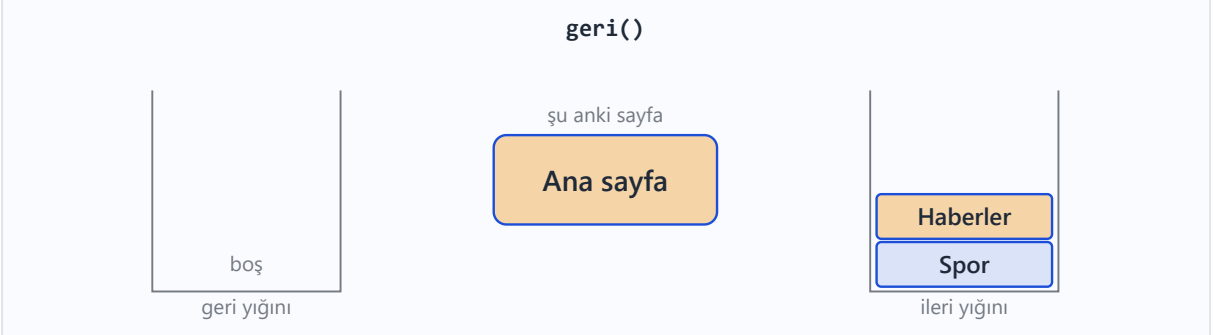
Örnek 7.3

Tarayıcı "Ana sayfa" ile açılıyor; sonra sırayla `ziyaret(Haberler)`, `ziyaret(Spor)`, `geri()`, `geri()`, `ileri()`, `ziyaret(Hava)`, `geri()` yapılıyor. Her adımdan sonraki durum tabloda verilmiştir (yığınlar alttan üste yazılmıştır).

İşlem	Geri yığını	Şu anki sayfa	İleri yığını
(açılış)	[]	Ana sayfa	[]
`ziyaret(Haberler)`	[Ana sayfa]	Haberler	[]
`ziyaret(Spor)`	[Ana sayfa, Haberler]	Spor	[]
`geri()`	[Ana sayfa]	Haberler	[Spor]
`geri()`	[]	Ana sayfa	[Spor, Haberler]
`ileri()`	[Ana sayfa]	Haberler	[Spor]
`ziyaret(Hava)`	[Ana sayfa, Haberler]	Hava	[] (Spor silindi)
`geri()`	[Ana sayfa]	Haberler	[Hava]

Tablo 7.8: İki yığınla tarayıcı geçmişi.

Kritik adım `ziyaret(Hava)`'dır. O anda ileri yığınında Spor vardır. Ama Haberler sayfasından yeni bir bağlantıya gidilmiş, yani yeni bir yola girilmiştir. Spor artık "ileri" gidilecek bir sayfa değildir; ileri yığını boşaltılır.





Şekil 7.9: Örnek 7.3'ün iki anı. Üstte iki `geri()` 'den sonra: geri yığını boş, şu anki sayfa Ana sayfa, ileri yığınında alttan üste Spor ve Haberler var. Altta `ziyaret(Hava)` 'dan sonra: geri yığınında Ana sayfa ve Haberler, ileri yığını temizlenmiş (Spor silinmiş).

Her işlem birkaç `push` ve `pop` yapar; ileri yığınını temizlemek dışında hepsi *O(1)*'dir. Temizleme, yığını yeni ve boş bir nesneyle değiştirerek de *O(1)* yapılabilir. Metin düzenleyicilerdeki geri al / yinele düzeni de aynıdır: "geri al" yığını yapılan değişiklikleri, "yinele" yığını geri alınanları tutar; yeni bir değişiklik yapılınca yinele yığını temizlenir.

> **Sık yapılan hata**
>
> Yeni bir sayfa ziyaret edildiğinde ileri yığınını temizlemeyi unutmak. Temizlenmezse yukarıdaki örnekte Hava'dan sonra "ileri" düğmesi kullanıcıyı Spor'a götürür; oysa Spor, artık terk edilmiş bir yolun sayfasıdır.

7.3.3 Çağrı yığını

Bir program çalışırken her metot çağrısı için bellekte bir kayıt tutulur. Bu kayıtta metodun parametreleri, yerel değişkenleri ve metot bitince programın nereden devam edeceği (dönüş adresi) bulunur. Bu kayda **yığın çerçevesi** (*stack frame*) denir. Çerçeveler bir yığında tutulur: **çağrı yığını** (*call stack*). Bir metot çağrıldığında çerçevesi yığına `push` edilir; metot dönünce çerçeve `pop` edilir ve kontrol, artık üstte olan çağıran metoda geçer.

Çağrı yığınını en açık gösteren örnek, kendini çağıran bir metottur. Bir metodun kendini çağırmasına **özyineleme** (*recursion*) denir. Aşağıdaki metot $n! = n \cdot (n-1) \cdot \ldots \cdot 1$ değerini hesaplar.

```
int fakt(int n) {
  if (n <= 1)
    return 1;
  return n * fakt(n - 1);
}
// main içinde:
System.out.println(fakt(3));
```

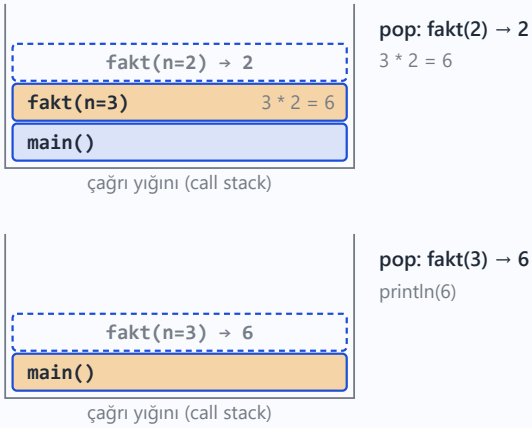
`n <= 1` durumu, metodun kendini çağırmadan doğrudan cevap verdiği **taban durumdur** (*base case*). Öteki her durumda `n * fakt(n - 1)` hesaplanır; ama çarpma, `fakt(n - 1)` bitmeden yapılamaz.

Adım	Olay	Çağrı yığını
1	`main` çalışıyor	main
2	`fakt(3)` çağrıldı; 3 <= 1 yanlış; `3 * fakt(2)` bekliyor	main, fakt(3)
3	`fakt(2)` çağrıldı; 2 <= 1 yanlış; `2 * fakt(1)` bekliyor	main, fakt(3), fakt(2)
4	`fakt(1)` çağrıldı; 1 <= 1 doğru; `return 1`	main, fakt(3), fakt(2), fakt(1)
5	`fakt(1)` pop edildi, 1 döndü; `fakt(2)` : 2 · 1 = 2	main, fakt(3), fakt(2)
6	`fakt(2)` pop edildi, 2 döndü; `fakt(3)` : 3 · 2 = 6	main, fakt(3)
7	`fakt(3)` pop edildi, 6 döndü; `main` 6 yazdırır	main

Tablo 7.9: `fakt(3)` çağrısında çağrı yığını (alttan üste).

Yığın 2–4. adımlarda büyür, 5–7. adımlarda küçülür. Taban duruma ulaşıldıktan sonra yığının geriye doğru boşalmasına **çözülme** (*unwinding*) denir. Askıda bekleyen her çerçeve, üstündeki çerçeve dönünce kaldığı yerden devam eder: `fakt(2)`, `fakt(1)`'in döndürdüğü 1 ile çarpmasını tamamlar.





Şekil 7.10: `fakt(3)` 'te çağrı yığını. (a) `fakt(3)` askıda: `3 * fakt(2)` hesaplanamıyor. (b) Taban durum: en üstte `fakt(1)`, 1 döndürecek. (c) `fakt(2)` pop edildi, 2 döndü; `fakt(3)` 3 · 2'yi hesaplıyor. (d) `fakt(3)` 6 ile döndü; yığında yalnız `main` kaldı.

Çağrı yığınının derinliği en çok *n* çerçevedir; bu yüzden `fakt(n)` *O(n)* bellek kullanır. Çağrı yığınının boyutu sınırlıdır. Çok derin özyineleme, ya da taban durumu hiç gelmeyen hatalı bir özyineleme, yığını doldurur ve Java `StackOverflowError` verir. Bu, çağrı yığınındaki *taşmanın* kendisidir.

Sık yapılan hata

Özyinelemeli metotta taban durumu unutmak ya da ona hiç ulaşılamayacak biçimde yazmak (örneğin `fakt(n + 1)` çağırmak). Her çağrı yeni bir çerçeve iter, hiçbiri dönmez; çerçeveler birikir ve program `StackOverflowError` ile durur.

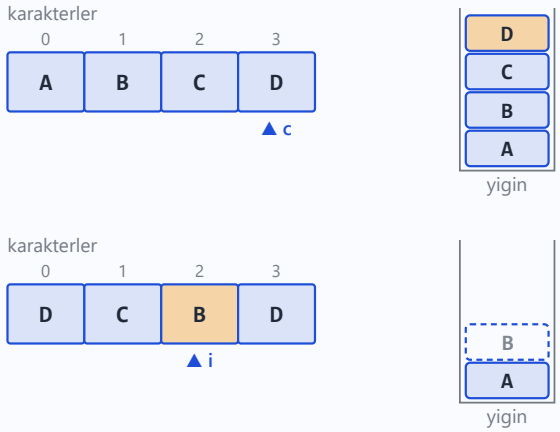
7.3.4 Dizgiyi tersine çevirme

LIFO düzeninin en doğrudan sonucu şudur: bir dizi öğe sırayla yığına eklenip sonra çıkarılırsa ters sırada çıkar. Bir dizgiyi tersine çevirmek için bu yeterlidir.

```
String tersCevir(String str) {
    Deque<Character> yigin = new ArrayDeque<>();
    char[] karakterler = str.toCharArray();
    for (char c : karakterler)
        yigin.push(c);
    for (int i = 0; i < str.length(); i++)
        karakterler[i] = yigin.pop();
    return new String(karakterler);
}
// main içinde:
System.out.println(tersCevir("ABCD"));
```

Metot önce boş bir karakter yığını oluşturur. `Deque<Character>` türü kullanılır, çünkü Java'da jenerik türler ilkel türlerle (`char`) değil, onların sarmalayıcı sınıflarıyla (`Character`) çalışır; `push(c)` sırasında `char` kendiliğinden `Character` 'a çevrilir. `toCharArray()` dizginin karakterlerini yeni bir diziye kopyalar; Java'da `String` değiştirilemez olduğu için sonuç bu dizinin üzerine yazılarak kurulur.

Birinci döngü karakterleri sırayla yığına iter. İkinci döngü `i = 0` 'dan başlayarak her hücreye yığından çıkan karakteri yazar. `"ABCD"` için birinci döngüden sonra yığın alttan üste A, B, C, D'dir. İkinci döngüde ilk `pop` D'yi verir ve `karakterler[0] = 'D'` olur; ardından C, B, A gelir. Metot `"DCBA"` döndürür.



Şekil 7.11: `tersCevir("ABCD")` . (a) Birinci döngü bitti: yığın alttan üste A, B, C, D; en üstte son karakter. (b) İkinci döngüde `i = 2` : D ve C yerleşmiş, şimdi B çıkıp 2. hücreye yazılıyor.

Her karakter bir kez `push` , bir kez `pop` edilir; iki döngü de n adım sürer. Zaman $O(n)$'dir. Yığın en çok n karakter tutar, karakter dizisi de n hücredir: ek bellek $O(n)$'dir. Aynı fikir, bir kelimenin **palindrom** (tersten de aynı okunan kelime, örneğin "kayak") olup olmadığını denetlemek için de kullanılabilir: ters çevrilmiş dizgi asıl dizgiyle karşılaştırılır.

7.3.5 Parantez kontrolü: problem ve algoritma

Bir derleyicinin ilk işlerinden biri, kaynak koddaki parantezlerin doğru eşleşip eşleşmediğini denetlemektir. Problemi sadeleştirelim: `S` dizisi yalnız `( ) { } [ ]` karakterlerinden oluşsun.

Tanım 7.3

Yalnız `( ) { } [ ]` karakterlerinden oluşan bir `S` dizisi, ancak ve ancak şu iki koşul sağlanırsa **geçerlidir**:

1. her açık parantez aynı tipte bir kapalı parantezle kapatılmıştır;
2. parantezler doğru sırada kapatılmıştır: en son açılan parantez ilk kapanır.

Bu denetime **parantez kontrolü** (*balanced parentheses*) denir.

İkinci koşul, LIFO kuralının ta kendisidir. Bu yüzden algoritma bir yığın kullanır: henüz kapanmamış, yani *eşleşmemiş* açık parantezler yığında tutulur. `S` baştan sona bir kez gezilir:

1. **Açık parantez** gelirse yığına `push` edilir. Kapanışını bekleyecektir.
2. **Kapalı parantez** gelirse: yığın boşsa `S` geçersizdir (kapatılacak açık parantez yoktur). Boş değilse `pop` yapılır; çıkan açık parantez gelen kapalıyla aynı tipte değilse `S` geçersizdir.
3. `S` **bitince** yığın boşsa geçerli, değilse geçersizdir (kapanmamış parantez kalmıştır).

Dört küçük örnek, algoritmanın yakaladığı hata türlerini gösterir:

Dizgi	Sonuç	Neden?
`{[()]}`	geçerli	her parantez ters sırada kapanıyor
`([)]`	geçersiz	sıra yanlış: `)` geldiğinde en son açılan `[`
`(()`	geçersiz	kapanmamış: sonda yığında `(` kalır
`())`	geçersiz	fazla kapanış: son `)` boş yığına rastlar

Tablo 7.10: Parantez dizileri ve sonuçları.

`([)]` örneği yalnız sayma ile çözülemeyeceğini gösterir. Bu dizide iki açık ve iki kapalı parantez vardır, her tipten de birer açık ve birer kapalı. Sayılar tutar, ama sıra yanlıştır. Yalnız sayaç tutan bir yöntem bu hatayı göremez; yığın görür.

7.3.6 Kod: gecerliMi()

```
boolean gecerliMi(String s) {
  Deque<Character> yigin = new ArrayDeque<>();
  for (char c : s.toCharArray()) {
    if (c == '(' || c == '[' || c == '{') {
      yigin.push(c);
    } else {
      if (yigin.isEmpty()) return false;
      char acik = yigin.pop();
      if (!eslesir(acik, c)) return false;
    }
  }
  return yigin.isEmpty();
}
```

Yardımcı metot, bir açık ve bir kapalı parantezin aynı tipte olup olmadığını söyler:

```
boolean eslesir(char a, char k) {
  return (a == '(' && k == ')')
      || (a == '[' && k == ']')
      || (a == '{' && k == '}');
}
```

`gecerliMi` dizginin her karakterini sırayla `c`'ye alır. Açık parantezse `yigin.push(c)` çalışır. Değilse (yani kapalı parantezse) önce `yigin.isEmpty()` sorulur; boşsa hemen `false` döner. Boş değilse en son açılan parantez `acik`'e çıkarılır ve `eslesir(acik, c)` ile tip denetlenir; eşleşmezse `false` döner. Döngü bittiğinde `return yigin.isEmpty();` kapanmamış parantez olup olmadığını cevaplar.

`eslesir` metodu üç olası doğru çifti `||` (veya) ile bağlar. `&&` (ve) her çiftte açık ve kapalının birlikte doğru olmasını ister. Örneğin `eslesir('[', ')')` çağrısında üç çiftin hiçbiri tutmaz ve sonuç `false`'tur.

Her karakter bir kez işlenir ve her işlem *O(1)*'dir; zaman *O(n)*'dir. En kötü durumda (örneğin bütün karakterler açık parantezse) yığın *n* öğe tutar: bellek *O(n)*'dir.

> **Sık yapılan hata**
>
> Kapalı parantez geldiğinde boş yığın denetimini atlamak. `if (yigin.isEmpty()) return false;` satırı olmasaydı `"())"` ya da yalnız `"]"` girdisinde boş `ArrayDeque`'ten `pop` yapılır ve `NoSuchElementException` fırlatılırdı. İkinci tipik hata, döngü bitince `true` döndürmektir: o zaman `"(()"` gibi kapanmamış parantezli dizgiler geçerli sayılır.

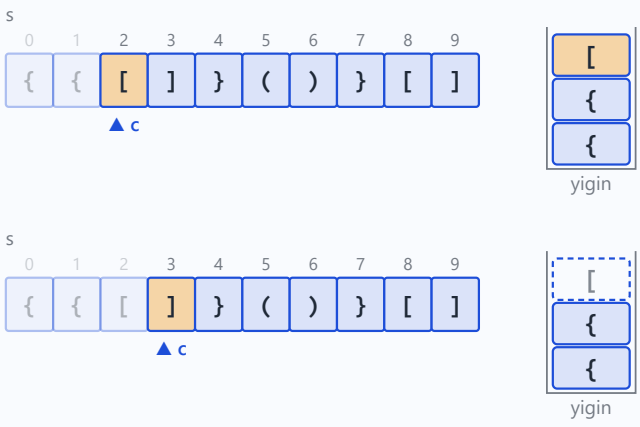
7.3.7 Kod izleme: geçerli bir dizgi

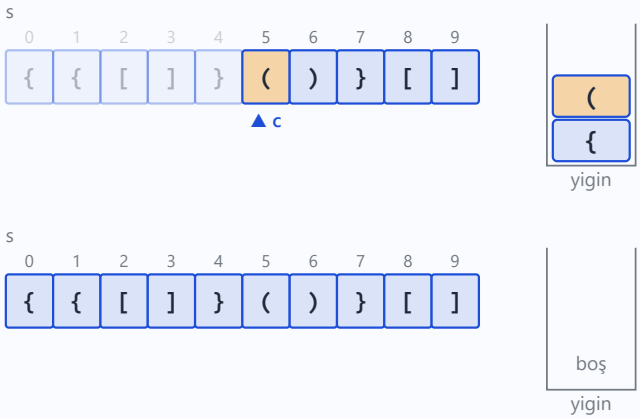
`gecerliMi("{{[]}()}[]")` çağrısını izleyelim. Dizginin on karakteri vardır; indisler 0'dan 9'a gider.

İndis	c	İşlem	Yığın (sonra)
0	`{`	push	{
1	`{`	push	{{
2	`[`	push	{{[
3	`]`	pop `[` ; `eslesir('[', ']')` doğru	{{
4	`}`	pop `{` ; `eslesir('{', '}')` doğru	{
5	`(`	push	{(
6	`)`	pop `(` ; `eslesir('(', ')')` doğru	{
7	`}`	pop `{` ; `eslesir('{', '}')` doğru	boş
8	`[`	push	[
9	`]`	pop `[` ; `eslesir('[', ']')` doğru	boş
(son)	—	`return yigin.isEmpty()`	boş → `true`

Tablo 7.11: `gecerliMi("{{[]}()}[]")` izlemesi. Yığın alttan üste yazılmıştır.

Her kapalı parantez geldiğinde yığının üstünde, ona en yakın ve henüz kapanmamış açık parantez durur. 4. indisteki `}`, 1. indisteki `{` ile eşleşir; 7. indisteki `}` ise 0. indisteki ile. Yığın bu eşleşmeyi hiçbir arama yapmadan, yalnız LIFO düzeniyle bulur. 7. indisten sonra yığın bir an boşalır; bu, o ana kadarki önekin kendi başına geçerli olduğunu gösterir. Dizgi bittiğinde yığın boştur ve metot `true` döndürür.





Şekil 7.12: `gecerliMi("{{[]}()[]")` izlemesinden dört an. (a) 2. indis: `[` push edildi, yığın { { [. (b) 3. indis: `]` geldi, en son açılan `[` çıkıyor. (c) 5. indis: `(` push edildi, yığın { (. (d) Dizgi bitti, yığın boş: `true`.

7.3.8 Geçersiz dizgiler

Algoritma üç farklı yerde `false` döndürebilir. Her biri ayrı bir hata türünü yakalar.

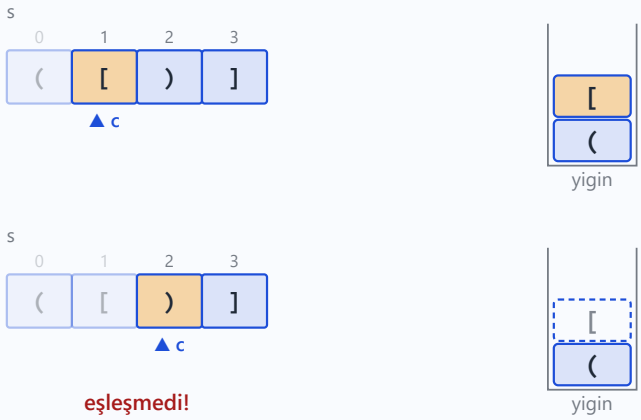
Örnek 7.4

Sıra yanlış: `"([)]"`.

1. `(` push edilir; yığın: (.
2. `[` push edilir; yığın: ([.
3. `)` gelir. Yığın boş değildir; pop en son açılan `[` 'i verir. `eslesir('[', ')')` yanlıştır: `)` kapanıyor ama en son açılan `[` 'dir. Metot burada `false` döndürür; son karakter `]` hiç okunmaz.

Kapanmamış: `"(()"`. İki `(` push edilir, `)` birini çıkarır. Döngü biter, yığında bir `(` kalır; `yigin.isEmpty()` yanlış olduğu için `false` döner.

Fazla kapanış: `"())"`. `(` push edilir, ilk `)` onu çıkarır ve yığın boşalır. İkinci `)` geldiğinde yığın boştur; boşluk denetimi `false` döndürür.



Şekil 7.13: `gecerliMi("([)]")`. (a) İki açık parantez push edildi: yığın ([. (b) `)` geldi, pop `[`'i verdi; tipler eşleşmedi ve metot `false` döndürdü.

Erken dönüş, algoritmanın bir başka yararıdır: hata bulunduğu anda sonuç bellidir ve dizginin geri kalanını okumaya gerek yoktur. En kötü durum yine *O(n)*'dir, ama geçersiz girdilerde metot çoğu zaman çok daha erken biter.

Aynı algoritma yalnız parantezlerle sınırlı değildir. Örneğin `<b><i>…</i></b>` gibi HTML etiketlerinin doğru iç içe olup olmadığı da böyle denetlenir: açılış etiketi yığına push edilir, kapanış etiketi geldiğinde pop yapılır ve etiket adları karşılaştırılır.

7.3.9 Sonraki büyük öğe

Bir sayı dizisi verilsin. Her öğe için, sağında kalan ve kendisinden büyük olan *ilk* öğe isteniyor; böyle bir öğe yoksa cevap −1'dir. Bu probleme **sonraki büyük öğe** (*next greater element*) denir.

indis	0	1	2	3	4	5
dizi	4	7	3	4	8	1
sonuc	7	8	4	8	−1	−1

Tablo 7.12: `dizi = {4, 7, 3, 4, 8, 1}` için sonraki büyük öğeler.

Tabloyu elle doğrulayalım. 0. öğe 4'ün sağındaki ilk büyük öğe 7'dir. 7'nin sağında 3 ve 4 daha küçüktür, ilk büyük 8'dir. 3'ün hemen sağındaki 4 ondan büyüktür. 8'in sağında yalnız 1 vardır; büyük öğe yoktur, cevap −1'dir. Son öğenin sağında hiç öğe yoktur: −1.

En doğrudan çözüm **kaba kuvvettir**: her öğe için sağındaki öğeler tek tek taranır ve ilk büyük olanda durulur. En kötü durumda (örneğin azalan bir dizide) her öğe için sağın tamamı taranır; toplam $(n-1) + (n-2) + \dots + 1 = n(n-1)/2$ karşılaştırma, yani $O(n^2)$ zaman gerekir.

Yığın ile *O(n)*'de çözülebilir. Fikir şudur: dizi sağdan sola gezilir ve yığında yalnız "aday" değerler tutulur. Aday, solda kalan öğeler için sonraki büyük öğe olabilecek bir değerdir. Yeni bir öğe `x` geldiğinde yığının üstündeki `x`'ten küçük ya da ona eşit değerler atılır. Neden? Bu değerler `x`'in sağındadır ve `x`'ten büyük değildir. `x`'in solundaki herhangi bir öğe için önce `x` görülür; `x` yeterince büyük değilse onlar da değildir. Yani bu değerler bir daha asla cevap olamaz. Atılmayan ilk değer (varsa) `x`'in cevabıdır. Sonra `x` yeni aday olarak yığına eklenir.

```
int[] sonrakiBuyuk(int[] dizi) {
    int[] sonuc = new int[dizi.length];
    Deque<Integer> yigin = new ArrayDeque<>();
    for (int i = dizi.length-1; i >= 0; i--) {
        while (!yigin.isEmpty() &&
               yigin.peek() <= dizi[i])
            yigin.pop();
        sonuc[i] = yigin.isEmpty()
                 ? -1 : yigin.peek();
        yigin.push(dizi[i]);
    }
    return sonuc;
}
```

Bu kuralın bir sonucu olarak yığın her an alttan üste **azalan** sırada kalır: yeni eklenen öğe, kendisinden küçük ya da eşit olanları attıktan sonra eklendiği için altındaki her değerden küçüktür. İçeriği hep sıralı kalan böyle bir yığına **monoton yığın** denir.

Teorem 7.2 (Monoton yığının doğrusal süresi)

`sonrakiBuyuk` metodu *n* öğeli bir dizide *O(n)* zamanda çalışır ve *O(n)* ek bellek kullanır.

İspat. İç içe iki döngü vardır, ama süre $O(n^2)$ değildir. İç `while` döngüsünün her turu bir `pop` yapar. Her öğe yığına tam bir kez `push` edilir (dış döngünün son satırında) ve bir kez çıkarıldıktan sonra bir daha eklenmez; dolayısıyla en çok bir kez `pop` edilir. Bütün çalışma boyunca toplam `pop` sayısı en çok *n*'dir. Dış döngü *n* tur döner ve her turda iç döngü dışında sabit iş yapar. Toplam iş $n + n = 2n$ mertebesindedir, yani *O(n)*. Yığın en çok *n* öğe tutar ve `sonuc` dizisi *n* hücredir: bellek *O(n)*. □

İspattaki düşünce biçimi, amortize analizin bir başka örneğidir: tek bir dış döngü turunda iç döngü birçok kez dönebilir, ama bütün turların toplamı sınırlıdır. İç içe döngü gördüğümüzde sürenin otomatik olarak karesel olduğunu varsaymak yanlıştır; iç döngünün toplamda kaç kez döndüğü sayılmalıdır.

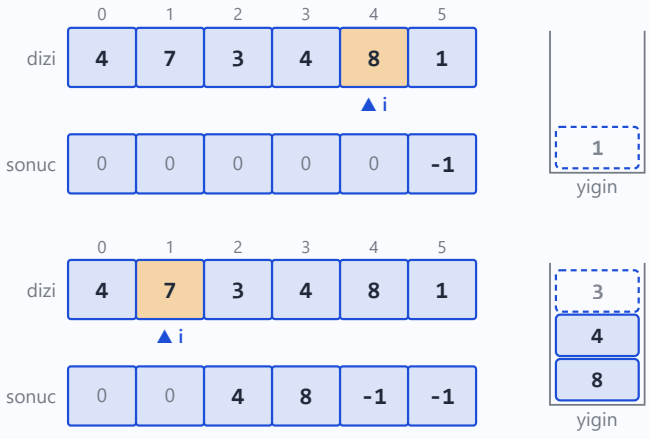
7.3.10 Kod izleme: sonrakiBuyuk()

`i`	`dizi[i]`	pop edilenler	`sonuc[i]`	Yığın (push'tan sonra)
5	1	—	−1 (yığın boş)	1
4	8	1 (1 ≤ 8)	−1 (yığın boş)	8
3	4	—	8	8 4
2	3	—	4	8 4 3
1	7	3, 4 (≤ 7)	8	8 7
0	4	—	7	8 7 4

Tablo 7.13: `sonrakiBuyuk({4, 7, 3, 4, 8, 1})` izlemesi. Yığın alttan üste yazılmıştır.

Başlangıçta `sonuc` dizisinin bütün hücreleri 0'dır (Java yeni `int` dizisini sıfırla doldurur) ve yığın boştur. `i = 5` 'te yığın boş olduğu için cevap −1'dir; 1, soldaki öğeler için yeni aday olarak eklenir. `i = 4` 'te üstteki 1, 8'den küçüktür ve atılır: 1, 8'in solundaki hiçbir öğenin cevabı olamaz, çünkü arada ondan büyük 8 vardır. Yığın boşalır, 8'in cevabı −1'dir.

`i = 1` satırı yöntemin işleyişini en iyi gösteren adımdır. Yığında alttan üste 8, 4, 3 vardır. Gelen 7, önce 3'ü sonra 4'ü atar; 8'de durur, çünkü `8 <= 7` yanlıştır. Cevap 8'dir. Atılan 3 ve 4, 7'nin solundakiler için gereksizdir. Son olarak `i = 0` 'da üstteki 7, 4'ten büyüktür ve hemen cevap olur. Metot `[7, 8, 4, 8, -1, -1]` döndürür. Toplam altı `push` ve üç `pop` yapılmıştır; Teorem 7.2'deki sınırla uyumludur.





Şekil 7.14: `sonrakiBuyuk` izlemesinden dört an. (a) `i = 4` : üstteki 1 ≤ 8, atılıyor. (b) `i = 1` : üstteki 3 ≤ 7, atılıyor. (c) `i = 1` : 4 de atıldı, üstte 8 kaldı; `sonuc[1] = 8` . (d) Sonuç dizisi tamamlandı: [7, 8, 4, 8, −1, −1].

Sık yapılan hata

İç döngüdeki karşılaştırmayı `<` diye yazmak. `yigin.peek() < dizi[i]` olsaydı eşit değerler atılmaz ve bir öğenin "sonraki büyüğü" olarak kendisine eşit bir değer bulunabilirdi. Örneğin `{5, 5}` dizisinde ilk 5'in cevabı −1 olmalıdır, ama `<` ile 5 çıkar. Eşitlik de "büyük değil" demektir; `<=` kullanılır.

7.3.11 En küçüğü O(1)'de veren yığın

Yığın ADT'sine bir işlem ekleyelim: `getMin()` , yığındaki en küçük değeri döndürsün. Bunu yığını tarayarak yapmak *O(n)* sürer. Amaç `getMin()` 'i de öteki işlemler gibi *O(1)* yapmaktır.

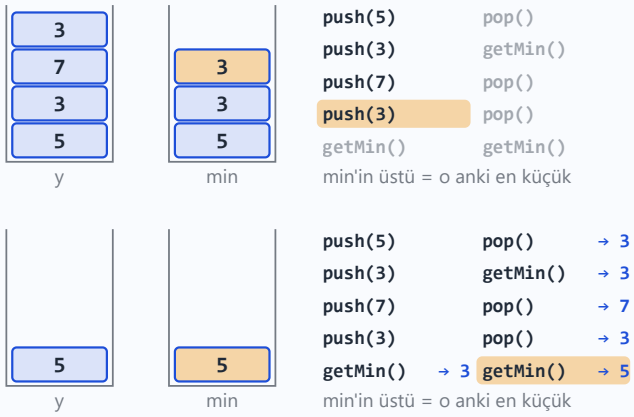
Çözüm, ikinci bir yığın kullanmaktır. `y` bütün öğeleri tutar. `min` yığınının üstü ise her an `y` 'deki en küçük değerdir. Fikir şöyledir: yeni gelen değer o ana kadarki en küçükten büyük değilse `min` 'e de eklenir. Bir değer `y` 'den çıktığında, `min` 'in üstüne eşitse oradan da çıkar; böylece bir önceki en küçük yeniden üste gelir.

```
Deque<Integer> y = new ArrayDeque<>();
Deque<Integer> min = new ArrayDeque<>();
void push(int x) {
  y.push(x);
  if (min.isEmpty() || x <= min.peek())
    min.push(x);
}
int pop() {
  int x = y.pop();
  if (x == min.peek()) min.pop();
  return x;
}
int getMin() { return min.peek(); }
```

İşlem	Koşul	`y`	`min`	Döner
`push(5)`	`min` boş → ekle	5	5	—
`push(3)`	3 <= 5 doğru → ekle	5 3	5 3	—
`push(7)`	7 <= 3 yanlış	5 3 7	5 3	—
`push(3)`	3 <= 3 doğru → ekle	5 3 7 3	5 3 3	—
`getMin()`	—	5 3 7 3	5 3 3	3
`pop()`	3 == 3 → `min`'den de çıkar	5 3 7	5 3	3; `getMin()` = 3
`pop()`	7 == 3 yanlış	5 3	5 3	7; `getMin()` = 3
`pop()`	3 == 3 → `min`'den de çıkar	5	5	3; `getMin()` = 5

Tablo 7.14: `push(5)`, `push(3)`, `push(7)`, `push(3)`, `getMin()`, ardından üç `pop()` ve her birinden sonra `getMin()`. Yığınlar alttan üste yazılmıştır.

Her işlem en çok iki `push`, iki `pop` ya da bir `peek` yapar; hepsi *O(1)*'dir. `getMin()` yalnız `min.peek()`'tir, arama yapmaz. Bedel ek bellektir: en kötü durumda (örneğin azalan sırada eklemelerde) `min` de *n* öğe tutar, ek alan *O(n)*'dir.



Şekil 7.15: Min yığın. (a) İkinci `push(3)` : 3 <= 3 olduğu için `min`'e de eklendi; `min` alttan üste 5, 3, 3. (b) Üç `pop`'tan sonra `y`'de yalnız 5 var; `getMin()` = 5.

Sık yapılan hata

`push`'taki koşulu `x < min.peek()` diye yazmak. Eşit en küçükler o zaman `min`'e bir kez eklenir. Yukarıdaki dizide ikinci 3 `min`'e girmez; ilk `pop` bu 3'ü çıkarırken `min`'deki tek 3'ü de siler. `y`'de hâlâ bir 3 olduğu hâlde `getMin()` yanlışlıkla 5 döndürür. `<=` sayesinde her eşit en küçük `min`'de ayrı bir kopya olarak durur.

`pop`'taki `x == min.peek()` karşılaştırması da bir incelik taşır. `x` bir `int`, `min.peek()` ise bir `Integer` döndürür. Java burada `Integer`'ı `int`'e açar ve değerleri karşılaştırır; bu doğrudur. İki taraf da `Integer` olsaydı `==` nesne kimliğini karşılaştırırdı ve 127'den büyük değerlerde yanlış

sonuç verebilirdi. Böyle durumlarda `equals` kullanılır.

7.4 Hanoi kuleleri ve stock span

Bu kesimdeki iki problem, yığının iki farklı kullanım biçimini gösterir. Hanoi kulelerinde yığın, özyinelemenin gizli çağrı yığınını açık bir veri yapısına dönüştürür. Stock span probleminde ise yığın, sonraki büyük öğedekine benzer bir "aday" listesi tutar ve karesel bir çözümü doğrusal hâle getirir.

7.4.1 Hanoi kuleleri: yığınla özyinelemesiz çözüm

Hanoi kuleleri bulmacasında üç çubuk (A, B, C) ve farklı büyüklükte n disk vardır. Başta bütün diskler A çubuğunda, büyükten küçüğe dizilidir. Amaç, diskleri aynı düzende C çubuğuna taşımaktır. İki kural vardır: her hamlede bir çubuğun en üstündeki tek disk taşınır ve büyük disk hiçbir zaman küçük diskin üstüne konamaz. Her çubuk kendi başına bir yığındır: yalnız en üstteki diske erişilir.

Bulmacanın klasik çözümü özyinelemelidir. n diski A'dan C'ye, B'yi yardımcı kullanarak taşımak için:

1. üstteki $n-1$ disk A'dan B'ye taşınır (C yardımcı);
2. en büyük disk A'dan C'ye taşınır;
3. $n-1$ disk B'den C'ye taşınır (A yardımcı).

Tek disk kaldığında (taban durum) disk doğrudan taşınır. Özyinelemeli çözümde bekleyen alt görevleri çağrı yığını tutar: 1. adım yapılırken 2. ve 3. adım askıdadır. Aynı işi açık bir yığınla yapmak mümkündür. Her bekleyen iş bir *görev* olarak saklanır.

```
// Gorev: n diski k(aynak)'tan h(edef)'e, y(ardımcı) ile taşı
record Gorev(int n, char k, char y, char h) {}

void hanoi(int n) {
  Deque<Gorev> yigin = new ArrayDeque<>();
  yigin.push(new Gorev(n, 'A', 'B', 'C'));
  while (!yigin.isEmpty()) {
    Gorev g = yigin.pop();
    if (g.n() == 1)
      System.out.println(g.k() + " → " + g.h());
    else {                          // ters sırayla
      yigin.push(new Gorev(g.n()-1, g.y(), g.k(), g.h()));
      yigin.push(new Gorev(1, g.k(), g.y(), g.h()));
      yigin.push(new Gorev(g.n()-1, g.k(), g.h(), g.y()));
    }
  }
}
```

`record`, Java 16 ile gelen kısa bir sınıf tanımıdır: `Gorev` dört alanlı değişmez bir nesnedir ve alanlarına `g.n()`, `g.k()` gibi metotlarla erişilir. Alan adları yorumda açıklanmıştır: `n` taşınacak disk sayısı, `k` kaynak, `y` yardımcı, `h` hedef çubuktur. Görev `(3, A, B, C)`, "3 diski A'dan C'ye, B yardımcıyla taşı" demektir.

Metot başlangıç görevini yığına koyar ve yığın boşalana kadar döner. Her turda üstteki görev çıkarılır. Görev tek diskliyse doğrudan yapılır: hamle yazdırılır. Değilse görev, yukarıdaki üç alt göreve bölünür ve bu alt görevler yığına *ters sırayla* eklenir: önce 3. adım `(n-1, y, k, h)`, sonra 2. adım `(1, k, y, h)`, en son 1. adım `(n-1, k, h, y)`. Böylece ilk yapılması gereken 1. adım en üste gelir ve bir sonraki turda ilk o çıkar.

Ters sıra, LIFO'nun doğrudan sonucudur. Alt görevler doğal sırayla eklenseydi en üstte 3. adım kalır ve ilk o yapılırdı: henüz A'dan alınmamış diskleri B'den C'ye taşımaya kalkışılırdı.

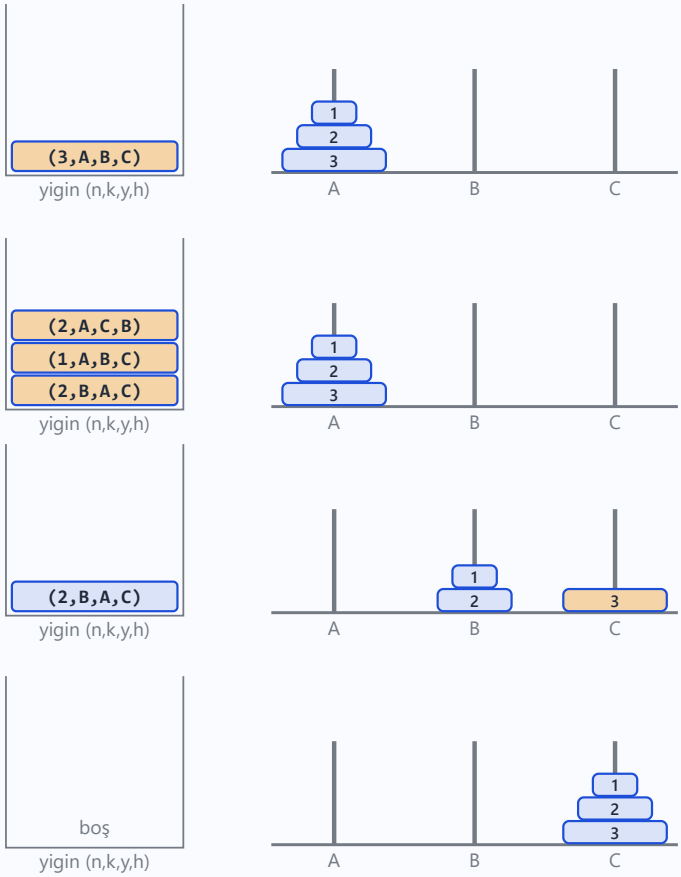
7.4.2 Kod izleme: hanoi(3)

Tablo 7.15, `hanoi(3)`'ün bütün döngü turlarını gösterir. Yığın sütununda görevler *üstten alta* yazılmıştır: ilk yazılan, bir sonraki turda çıkacak görevdir. Görevler `(n,k,y,h)` biçimindedir.

Tur	pop	Yapılan	Yığın (tur sonunda, üstten alta)
0	—	başlangıç görevi	(3,A,B,C)
1	(3,A,B,C)	3 alt görev push	(2,A,C,B) (1,A,B,C) (2,B,A,C)
2	(2,A,C,B)	3 alt görev push	(1,A,B,C) (1,A,C,B) (1,C,A,B) (1,A,B,C) (2,B,A,C)
3	(1,A,B,C)	hamle 1: A → C	(1,A,C,B) (1,C,A,B) (1,A,B,C) (2,B,A,C)
4	(1,A,C,B)	hamle 2: A → B	(1,C,A,B) (1,A,B,C) (2,B,A,C)
5	(1,C,A,B)	hamle 3: C → B	(1,A,B,C) (2,B,A,C)
6	(1,A,B,C)	hamle 4: A → C	(2,B,A,C)
7	(2,B,A,C)	3 alt görev push	(1,B,C,A) (1,B,A,C) (1,A,B,C)
8	(1,B,C,A)	hamle 5: B → A	(1,B,A,C) (1,A,B,C)
9	(1,B,A,C)	hamle 6: B → C	(1,A,B,C)
10	(1,A,B,C)	hamle 7: A → C	boş

Tablo 7.15: `hanoi(3)` izlemesi. Yığın üstten alta yazılmıştır.

1. turda `(3,A,B,C)` bölünür: `g.n()-1 = 2`, `g.y() = B`, `g.k() = A`, `g.h() = C` olduğu için önce `(2,B,A,C)`, sonra `(1,A,B,C)`, en son `(2,A,C,B)` eklenir. Üstte "2 diski A'dan B'ye taşı" görevi kalır. 2. turda bu görev de bölünür ve üç tek diskli görev üste gelir. 3–5. turlar en küçük iki diski B'ye taşır (A → C, A → B, C → B). 6. turda en büyük disk A'dan C'ye geçer; bu, ilk bölmedeki 2. adımdır. 7. tur, kalan "2 diski B'den C'ye taşı" görevini böler ve 8–10. turlar onu tamamlar.



Şekil 7.16: Görev yığını ve çubuklar. (a) Başlangıç: yığında yalnız (3,A,B,C), diskler A'da. (b) İlk bölmeden sonra yığın (üstten alta) (2,A,C,B), (1,A,B,C), (2,B,A,C). (c) Hamle 4: en büyük disk A'dan C'ye geçti; küçük iki disk B'de. (d) Yığın boş, döngü bitti; üç disk de C'de.

Hamle sayısı $7 = 2^3 - 1$'dir. Genel olarak n disk için $2^n - 1$ hamle gerekir, çünkü $T(n) = 2T(n-1) + 1$ ve $T(1) = 1$'dir. Yığından çıkan görev sayısı hamle sayısından biraz fazladır: her hamle bir tek diskli görevdir, bunlara bölünen görevler eklenir. $n = 3$'te 7 tek diskli ve 3 bölünen görev, toplam 10 tur vardır. Zaman $O(2^n)$'dir; bu, problemin kendi alt sınırıdır, çünkü her hamlenin yazılması gerekir.

Bellek açısından durum çok daha iyidir. Yığın, izlemede en çok 5 görev tutmuştur. Her bölme en üstteki görevi üç görevle değiştirir, yani yığını iki büyütür; bölmeler en çok $n - 1$ kez üst üste yapılabilir (görevdeki disk sayısı her seferinde bir azalır). Yığın en çok $1 + 2(n-1) = 2n - 1$ görev tutar: $O(n)$ bellek. $n = 3$'te 5, $n = 4$'te 7 görev.

Açık yığının özyinelemeye karşı bir üstünlüğü vardır: bekleyen işlerin listesi programın kendi denetimindedir. Çağrı yığınının boyutunu dil ortamı sınırlar ve çok derin özyineleme `StackOverflowError` verir; açık yığın ise yığın belleğinde (heap) büyür. Aynı dönüşüm, derinlik öncelikli arama gibi başka özyinelemeli algoritmalara da uygulanabilir.

Sık yapılan hata

Alt görevleri yapılacakları sırayla push etmek. Yığın son ekleneni ilk çıkardığı için, ilk yapılması gereken görev en son eklenmelidir. Özyinelemeyi açık yığına çevirirken alt çağrılar her zaman ters sırada eklenir.

7.4.3 Stock span problemi

Bir hisse senedinin günlük kapanış fiyatları bir dizide verilsin. Finansal analizde sık sorulan bir soru, bugünkü fiyatın "kaç gündür aşılmadığıdır". Bu soruya **stock span** (hisse senedi sıçrama) problemi denir.

Tanım 7.4

Bir günün **span değeri**, o gün dahil, geriye doğru fiyatı o günün fiyatından küçük ya da ona eşit olan ardışık günlerin sayısıdır. Daha yüksek fiyatlı ilk günde sayım durur.

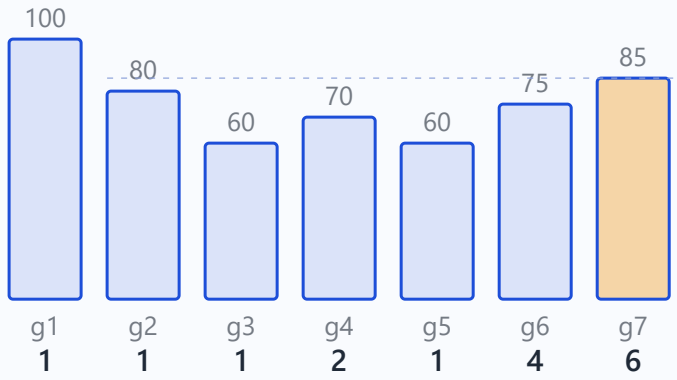
Tanımdaki üç ayrıntı önemlidir: sayım bugünü de içerdiği için her span en az 1'dir; eşit fiyatlar sayıma dahildir; ve sayım *ardışık* günler üzerinden yapılır, yani daha yüksek bir fiyata rastlanınca, onun gerisinde daha düşük fiyatlar olsa bile sayım biter.

Örnek 7.5

Fiyatlar 7 gün boyunca `[100, 80, 60, 70, 60, 75, 85]` olsun. Günler 1'den 7'ye numaralandırılmıştır.

1. **1. gün (100):** öncesinde gün yok. Span 1.
2. **2. gün (80):** önceki gün 100 > 80, sayım durur. Span 1.
3. **3. gün (60):** önceki gün 80 > 60. Span 1.
4. **4. gün (70):** 3. gün 60 ≤ 70 sayılır; 2. gün 80 > 70, durur. Span 2.
5. **5. gün (60):** 4. gün 70 > 60. Span 1.
6. **6. gün (75):** 60, 70, 60 ≤ 75 sayılır; 2. gün 80 > 75, durur. Span 4.
7. **7. gün (85):** 75, 60, 70, 60, 80 ≤ 85 sayılır; 1. gün 100 > 85, durur. Span 6.

Sonuç: `[1, 1, 1, 2, 1, 4, 6]`.



Şekil 7.17: Yedi günlük fiyat çubukları (g1–g7) ve altlarında kalın yazılmış span değerleri. 7. gün (85) için önceki beş gün (g2–g6) ve kendisi sayılır; kesikli çizgi bu günlerin üzerinden geçer. Sayım, fiyatı 85'ten yüksek olan g1'de (100) durur.

Kaba kuvvet çözümü her gün için geriye doğru yürür ve daha yüksek fiyat bulunca durur. Fiyatlar sürekli artıyorsa her gün başa kadar yürünür ve toplam $1 + 2 + \ldots + n$ adım, yani $O(n^2)$ zaman gerekir. Yığın bu tekrarlı yürüyüşü ortadan kaldırır.

7.4.4 Kod: span()

```
int[] span(int[] fiyatlar) {
    int[] sicrama = new int[fiyatlar.length];
    Deque<Integer> yigin = new ArrayDeque<>();
    yigin.push(0); sicrama[0] = 1;
    for (int i = 1; i < fiyatlar.length; i++) {
        while (!yigin.isEmpty() &&
            fiyatlar[i] >= fiyatlar[yigin.peek()])
            yigin.pop();
        sicrama[i] = yigin.isEmpty() ? i + 1
                                     : i - yigin.peek();
        yigin.push(i);
    }
    return sicrama;
}
```

Yöntemin fikri, sonraki büyük öğedekiyle aynıdır, ama bu kez dizi soldan sağa gezilir ve yığında değerler yerine *indisler* tutulur. Yığında, henüz aşılmamış günlerin indisleri vardır; bu günlerin fiyatları alttan üste azalır. Gün `i` geldiğinde, fiyatı bugünkünden küçük ya da ona eşit olan günler yığından atılır. Bu günler artık hiçbir gelecek gün için sınır olamaz: bugün onları aşmıştır ve bugünün kendisi onlardan daha yakın ve daha yüksek (ya da eşit) bir engeldir.

Atma bittikten sonra yığının üstünde, bugünden önceki ve bugünden *daha yüksek* fiyatlı ilk gün kalır. Span, bu günle bugün arasındaki gün sayısıdır: `i - yigin.peek()`. Yığın boşalmışsa bugünkü fiyat bütün önceki fiyatları aşmıştır ve span `i + 1`'dir (0'dan `i`'ye bütün günler). Son olarak bugün yığına eklenir; gelecek günler için olası bir sınırdır.

Kodda ilk gün döngüden önce ele alınır: `yigin.push(0); sicrama[0] = 1;`. İlk günün öncesinde gün olmadığı için span'i 1'dir ve 0 indisi yığının ilk öğesi olur. Döngü bu yüzden `i = 1`'den başlar. İç `while` döngüsünün koşulundaki `fiyatlar[i] >= fiyatlar[yigin.peek()]`, yığında indis tutulduğu için fiyata indis üzerinden erişir.

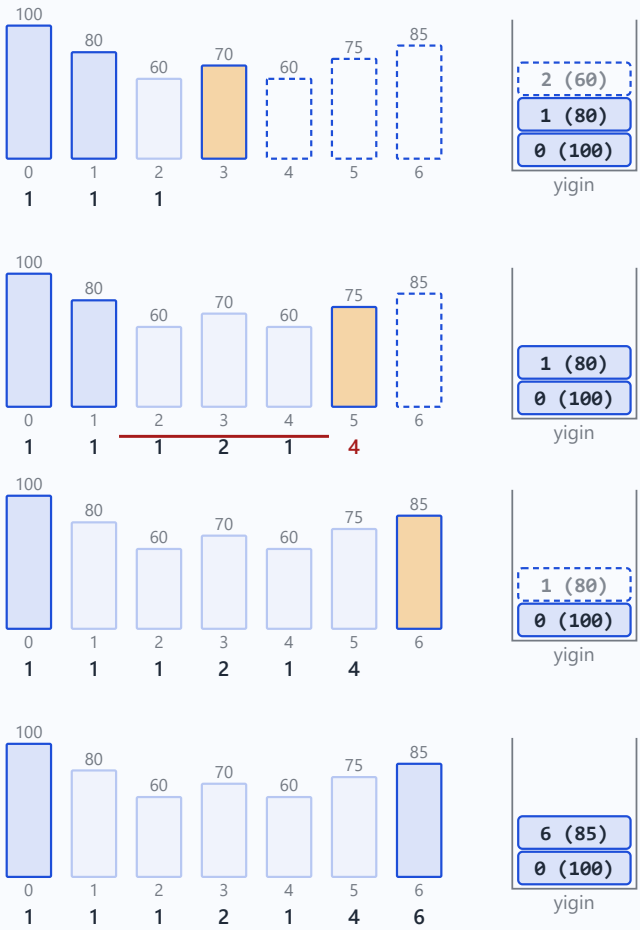
Maliyet, sonraki büyük öğedeki gerekçeyle aynıdır: her indis tam bir kez eklenir ve en çok bir kez çıkarılır. Toplam `pop` sayısı en çok *n*'dir; zaman *O(n)*, bellek *O(n)*'dir.

7.4.5 Kod izleme: span()

`i`	Fiyat	pop edilenler	Üst	`sicrama[i]`	Yığın (push'tan sonra)
0	100	—	—	1 (ilk gün)	0(100)
1	80	—	0	1 − 0 = 1	0(100) 1(80)
2	60	—	1	2 − 1 = 1	0(100) 1(80) 2(60)
3	70	2 (60 ≤ 70)	1	3 − 1 = 2	0(100) 1(80) 3(70)
4	60	—	3	4 − 3 = 1	0(100) 1(80) 3(70) 4(60)
5	75	4, 3	1	5 − 1 = 4	0(100) 1(80) 5(75)
6	85	5, 1	0	6 − 0 = 6	0(100) 6(85)

Tablo 7.16: `span({100, 80, 60, 70, 60, 75, 85})` izlemesi. Yığında indisler alttan üste yazılmıştır; parantez içinde o günün fiyatı vardır.

`i = 3`'te 70, üstteki 2. günün fiyatını (60) aşar ve 2 atılır. 2. gün artık kimsenin sınırı olamaz: 3. günden sonraki her gün için 3. gün (70) ondan daha yakın ve daha yüksektir. Üstte 1. gün (80) kalır ve span *3 − 1 = 2*'dir. `i = 5`'te 75, önce 4. günü (60) sonra 3. günü (70) atar, 1. günde (80) durur: span *5 − 1 = 4*. `i = 6`'da 85, 5. günü (75) ve 1. günü (80) atar, 0. günde (100) durur: span 6. Metot `[1, 1, 1, 2, 1, 4, 6]` döndürür; Örnek 7.5'teki elle bulunan sonuçla aynıdır. Toplam 7 `push` ve 5 `pop` yapılmıştır.



Şekil 7.18: `span` izlemesinden dört an. (a) `i = 3` : 70 ≥ 60, 2. gün atılıyor. (b) `i = 5` : 3. ve 4. günler atıldı, üstte 1. gün (80); span = 4. (c) `i = 6` : 85 ≥ 80, 1. gün de atılıyor. (d) Bitiş: yığında 0(100) ve 6(85); span dizisi [1, 1, 1, 2, 1, 4, 6].

Sık yapılan hata

Yığın boşaldığında span'i `i` olarak hesaplamak. Günler 0'dan başlayarak numaralandığı için `i` indisli güne kadar, kendisi dahil `i + 1` gün vardır. İkinci tipik hata, iç döngüde `>` kullanmaktır: o zaman eşit fiyatlı günler atılmaz ve sayıma dahil edilmez; tanım ise "küçük ya da eşit" der.

7.5 İfade gösterimleri

Okulda öğrenilen `3 + 4 * 2` gibi aritmetik ifadeler bir insan için kolay okunur, ama bir bilgisayar için iki zorluk taşır. Birincisi, işlemlerin sırası yazılış sırasıyla aynı değildir: çarpma, solda yazılmış toplamadan önce yapılır. İkincisi, parantezler bu sırayı değiştirir ve iç içe geçebilir. Bu kesim, aynı ifadeyi işlem sırasını açıkça taşıyan iki başka biçimde yazmayı ve bu biçimler arasında yığınla dönüşüm yapmayı anlatır.

7.5.1 Ara ek, önek ve sonek gösterim

Bir ifadede **işlenenler** (A, B, 5 gibi değerler) ve **işleçler** (+, −, *, /) bulunur. Üç gösterim, işlecin işlenenlere göre nereye yazıldığıyla ayrılır.

Gösterim	İşlecin yeri	A * (B + C) / D
Ara ek (*infix*)	işlenenlerin arasında	`A * ( B + C ) / D`
Önek (*prefix*) · Polonya gösterimi	işlenenlerden önce	`/ * A + B C D`
Sonek (*postfix*) · ters Polonya gösterimi	işlenenlerden sonra	`A B C + * D /`

Tablo 7.17: Aynı ifadenin üç gösterimi.

Günlük yazdığımız biçim **ara ek** gösterimdir: `B + C`'de işleç iki işlenenin arasındadır. **Önek** gösterimde işleç işlenenlerinden önce gelir: `+ B C`. Bu gösterimi Polonyalı mantıkçı Jan Łukasiewicz 1920'lerde önerdiği için *Polonya gösterimi* de denir. **Sonek** gösterimde işleç işlenenlerinden sonra gelir: `B C +`. Bunun adı *ters Polonya gösterimidir* ve İngilizce kısaltmasıyla RPN (*Reverse Polish Notation*) olarak da bilinir. Kod ve simülatör adlarında İngilizce kısa adlar kullanıldığı için not boyunca yer yer *infix*, *prefix* ve *postfix* adları da geçer.

Sonek ifadeyi okumak başta yadırgatıcıdır. `A B C + * D /` şöyle çözülür: soldan okunurken bir işlece gelince, ondan hemen önceki iki değer o işlecin işlenenleridir ve yerlerini sonuç alır. `B C +` → `(B + C)`; sonra `A (B + C) *` → `A * (B + C)`; sonra `… D /` → `(A * (B + C)) / D`.

Önek ve sonek gösterimin büyük üstünlüğü şudur: **parantez ve öncelik kuralı gerekmez**. İşlem sırası, simgelerin dizilişinden kesin olarak okunur. Sonek ifade soldan sağa, önek ifade sağdan sola tek geçişte ve bir yığınla değerlendirilir. Bu yüzden derleyiciler ve yorumlayıcılar ifadeleri çoğunlukla sonek biçime çevirir. Java derleyicisinin ürettiği bayt kodu da yığın tabanlıdır: `a + b` ifadesi "a'yı yığına koy, b'yi yığına koy, topla" komutlarına dönüşür. Bazı hesap makineleri (örneğin eski HP modelleri) doğrudan RPN ile çalışır. Sembolik cebir programları ve ifade ağaçları üzerinde çalışan algoritmalar da bu gösterimleri kullanır.

> **Sık yapılan hata**
>
> Önek ifadede işleçleri ara ekteki sırasıyla öne almak. `A * (B + C) / D` için `* A / + B C D` yazmak yanlıştır; bu ifade `A * ((B + C) / D)` demektir. Tam sayı bölmesinde bu iki ifade farklı sonuç verebilir; sonuç aynı olsa bile hesaplama ağacı farklıdır. Ara ekte `*` ve `/` soldan sağa yapılır: önce `A * (B + C)`, sonra `/ D`. Doğru önek bu yüzden `/ * A + B C D`'dir: en dıştaki işlem olan `/` en başa gelir.

7.5.2 Öncelik, birleşme ve elle dönüşüm

Ara ek ifadeyi doğru okumak için iki kural gerekir. **Öncelik**, farklı işleçlerden hangisinin önce yapılacağını söyler. **Birleşme**, aynı öncelikteki işleçler yan yana geldiğinde hangi yönde yapılacağını söyler.

İşleç	Öncelik	Birleşme
`^` (üs)	3	sağdan sola
`*` `/`	2	soldan sağa
`+` `-`	1	soldan sağa

Tablo 7.18: İşleçlerin öncelik ve birleşme yönleri.

Soldan birleşmenin anlamı şudur: `A - B + C`, `(A - B) + C` demektir, `A - (B + C)` değil. Sayılarla: `10 - 4 + 3` önce `10 - 4 = 6`, sonra `6 + 3 = 9` verir; sağdan yapılsaydı `10 - 7 = 3` çıkardı. Bu ifadenin soneki `A B - C +`'dır. Üs ise sağdan birleşir: `2 ^ 3 ^ 2`, `2 ^ (3 ^ 2) = 2`^{9} `= 512` demektir, `(2 ^ 3) ^ 2 = 64` değil.

Bir ara ek ifadeyi elle önek ya da sonek biçime çevirmenin güvenilir bir yolu **tam parantezlemedir**:

1. Öncelik ve birleşme kurallarına göre her işlem kendi parantezine alınır.
2. **Sonek** için her işleç, kendi parantezinin sağ ucuna taşınır ve parantezler silinir.
3. **Önek** için her işleç, kendi parantezinin sol ucuna taşınır ve parantezler silinir.

Örnek 7.6

(a) `A * (B + C) / D`. Önce parantez içi, sonra soldan sağa `*` ve `/`: `((A * (B + C)) / D)`.

- Sonek: `(B + C)` → `B C +`; `(A * …)` → `A B C + *`; `(… / D)` → `A B C + * D /`.
- Önek: `(B + C)` → `+ B C`; `(A * …)` → `* A + B C`; `(… / D)` → `/ * A + B C D`.

(b) `A + B * C - D / E`. `*` ve `/` önce yapılır; ardından `+` ve `-` soldan sağa: `((A + (B * C)) - (D / E))`.

- Sonek: `A B C * +` sol taraf, `D E /` sağ taraf, en sonda `-`: `A B C * + D E / -`.
- Önek: `-` başta, sonra sol taraf `+ A * B C`, sonra sağ taraf `/ D E`: `- + A * B C / D E`.

Dönüşümü denetlemenin hızlı bir yolu vardır: **işlenenlerin sırası üç gösterimde de aynı kalır**. Yukarıdaki iki örnekte de A, B, C, D (ve E) her gösterimde bu sırayla geçer; yalnız işleçlerin yeri değişir. Bir dönüşüm sonucunda işlenenlerin sırası bozulmuşsa sonuç kesinlikle yanlıştır. Ters yön her zaman doğru değildir: işlenen sırası doğru olduğu hâlde işleçler yanlış yerde olabilir (7.5.1'deki `* A / + B C D` örneği gibi).

7.5.3 Ara ekten soneke: algoritma

Tam parantezleme elle kolaydır, ama bir programın uygulaması için önce bütün ifadeyi çözümlemek gerekir. Edsger Dijkstra'nın 1961'de tanımladığı ve demiryolu makas alanına benzetilerek *shunting-yard* (manevra sahası) algoritması adını alan yöntem, aynı işi ifadeyi soldan sağa tek geçişte okuyarak yapar. İşleçler ve açık parantezler bir yığında bekletilir; işlenenler doğrudan çıktıya gider.

Okunan simge	Yapılacak işlem
İşlenen (A, B, 5 ...)	Doğrudan çıktıya yaz.
`(`	Yığına koy.
`)`	`(` görülene kadar yığından çıkarıp çıktıya yaz; `(` 'ı at.
İşleç `op`	Üstte `(` olmadığı ve üstteki işlecin önceliği ≥ `op` olduğu sürece çıkarıp çıktıya yaz; sonra `op` 'u yığına koy.
Girdi bitti	Yığında kalan bütün işleçleri çıkarıp çıktıya yaz.

Tablo 7.19: Ara ekten soneke dönüşüm kuralları.

Kuralların arkasındaki düşünce şudur. Bir işleç okunduğunda henüz sağ işleneni gelmemiştir, bu yüzden hemen çıktıya yazılamaz; yığında bekler. Yeni bir işleç `op` geldiğinde, yığındaki işleçlerden önceliği `op` 'unkinden büyük ya da eşit olanların işlenenleri tamamlanmıştır ve `op` 'tan önce yapılmaları gerekir. Bu yüzden çıktıya alınırlar. Önceliği daha düşük olanlar ise `op` 'tan sonra yapılacaktır ve beklemeye devam eder.

Eşitlikte de çıkarmanın (≥) nedeni soldan birleşmedir. `A - B + C` 'de `+` geldiğinde yığında `-` vardır; ikisinin önceliği eşittir. Soldaki `-` önce yapılmalıdır, bu yüzden `-` çıktıya alınır ve sonuç `A B - C +` olur. Sağdan birleşen `^` için kural farklıdır: yalnız önceliği daha büyük (>) olanlar çıkarılır, eşit olan `^` yığında kalır. Böylece `A ^ B ^ C` 'de ikinci `^` birincinin üstüne konur ve önce o yapılır.

Parantez, öncelik kurallarına bir “duvar” örer. `(` yığına konduktan sonra gelen işleçler onun altındakilere hiç bakmaz; `)` gelince yalnız `(` 'a kadar olan işleçler çıkarılır. Parantezler hiçbir zaman çıktıya yazılmaz: sonek gösterimde gerek yoktur.

Her simge bir kez okunur. Her işleç yığına bir kez girer ve bir kez çıkar. Toplam iş, ifadenin uzunluğu *n* ile orantılıdır: zaman *O(n)*, yığın için bellek en kötü durumda *O(n)* 'dir.

7.5.4 Kod: infixToPostfix()

```
String infixToPostfix(String ifade) {
  StringBuilder cikti = new StringBuilder();
  Deque<Character> yigin = new ArrayDeque<>();
  for (char c : ifade.toCharArray()) {
    if (Character.isLetterOrDigit(c)) cikti.append(c);
    else if (c == '(') yigin.push(c);
    else if (c == ')') {
      while (yigin.peek() != '(')
        cikti.append(yigin.pop());
      yigin.pop();                  // '(' atılır
    } else {                        // işleç
      while (!yigin.isEmpty() &&
          oncelik(yigin.peek()) >= oncelik(c))
        cikti.append(yigin.pop());
      yigin.push(c);
    }
  }
  while (!yigin.isEmpty()) cikti.append(yigin.pop());
  return cikti.toString();
}
```

Öncelikleri veren yardımcı metot:

```
int oncelik(char op) {
  switch (op) {
    case '+': case '-': return 1;
    case '*': case '/': return 2;
    default: return 0;   // '('
  }
}
```

Kod, Tablo 7.19'daki beş kuralı sırasıyla uygular. `Character.isLetterOrDigit(c)` bir harf ya da rakam için doğrudur; bunlar işlenendir ve `cikti`'ya eklenir. `StringBuilder`, Java'da bir dizgiyi parça parça kurmanın verimli yoludur: her `append` sona ekler. `(` yığına konur. `)` geldiğinde `while (yigin.peek() != '(')` döngüsü `(`'a kadar bütün işleçleri çıktıya taşır, ardından tek bir `yigin.pop()` `(`'ı atar. Öteki her karakter işleçtir: `while` döngüsü, üstteki işlecin önceliği gelen işlecinkinden küçük olmadığı sürece çıkarma yapar; sonra gelen işleç yığına konur. Girdi bitince kalan işleçler sırayla çıktıya eklenir.

Kodda "üstte `(` yoksa" koşulu açıkça yazılmamıştır. Bu koşulu `oncelik` metodu sağlar: `(` için `default` dalı çalışır ve 0 döner. Her gerçek işlecin önceliği en az 1 olduğu için `oncelik('(') >= oncelik(c)` hiçbir zaman doğru olmaz. Yani işleç geldiğinde `(` hiçbir zaman çıkarılmaz; onu yalnız `)` atar.

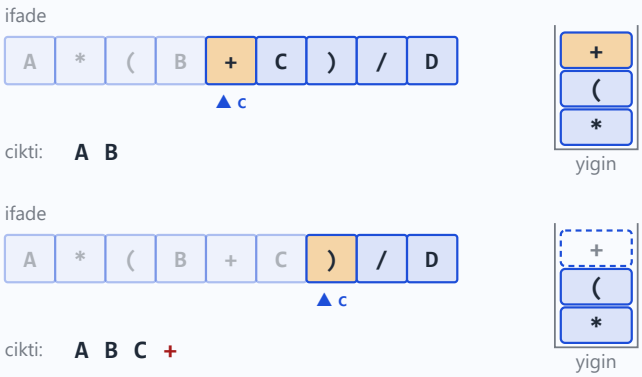
Kodun bazı varsayımları vardır: işlenenler tek karakterlidir (A, B, 5 gibi), yalnız `+ - * /` işleçleri kullanılır (`^` yoktur), ifadede boşluk yoktur ve ifade geçerlidir. Geçersiz bir ifadede, örneğin eşi olan `(` bulunmayan bir `)` varsa, `yigin.peek()` boş yığında `null` döndürür ve `null != '('` karşılaştırması `Character`'ı `char`'a açmaya çalışırken `NullPointerException` fırlatır.

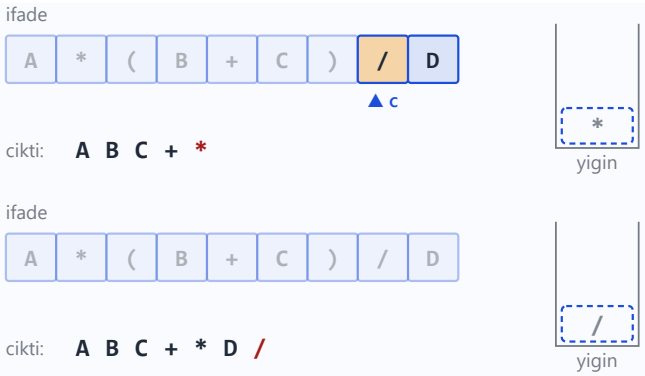
7.5.5 Kod izleme: A*(B+C)/D

`c`	İşlem	Yığın (sonra)	`cikti`
`A`	işlenen → çıktıya	boş	A
`*`	yığın boş → push	*	A
`(`	her zaman push	* (	A
`B`	işlenen → çıktıya	* (	A B
`+`	üstte `(` (öncelik 0) → push	* (+	A B
`C`	işlenen → çıktıya	* (+	A B C
`)`	`(`'a kadar pop: `+` çıktıya; `(` atılır	*	A B C +
`/`	üstteki `*` (2) ≥ `/` (2) → `*` çıktıya; yığın boş → push `/`	/	A B C + *
`D`	işlenen → çıktıya	/	A B C + * D
(son)	kalan `/` çıktıya	boş	A B C + * D /

Tablo 7.20: `infixToPostfix("A*(B+C)/D")` izlemesi. Yığın alttan üste yazılmıştır.

Belirleyici an `/` okunduğunda gelir. Yığında `*` vardır ve iki işlecin önceliği eşittir. ≥ kuralı `*`'ı çıktıya alır: ara ekte `*` soldadır ve önce yapılır. `*` yığında bırakılsaydı sonuç `A B C + D / *` olurdu; bu da `A * ((B + C) / D)` demektir. Sonuç `A B C + * D /`, Örnek 7.6'daki elle bulunan sonekle aynıdır.





Şekil 7.19: `A*(B+C)/D` dönüşümünden dört an. (a) `+` okundu: üstte `(` olduğu için doğrudan push edildi. (b) `)` okundu: `+` çıktıya alındı. (c) `/` okundu: eşit öncelikli `*` çıktıya alınıyor. (d) Girdi bitti: kalan `/` çıktıya; sonuç A B C + * D /.

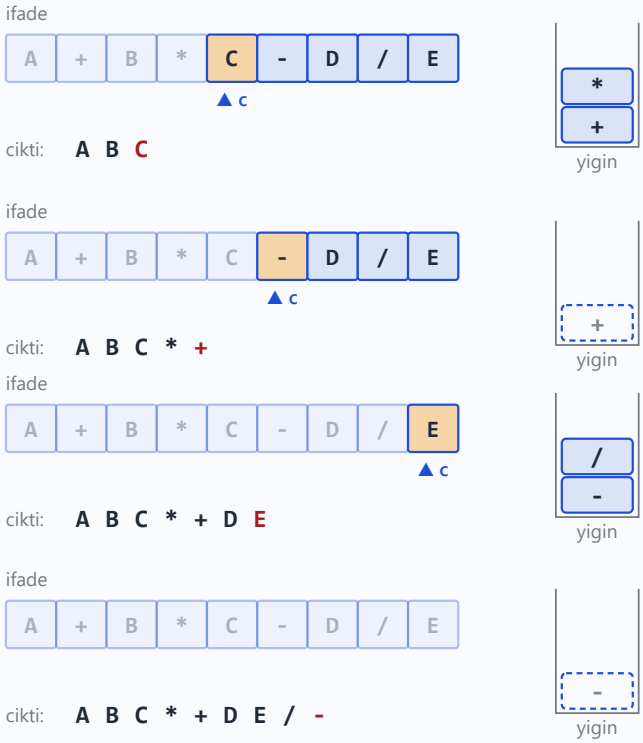
7.5.6 Kod izleme: A+B*C-D/E

Parantezsiz bu ifadede bütün iş öncelik kurallarına kalır.

`c`	**İşlem**	**Yığın (sonra)**	`cikti`
`A`	çıktıya	boş	A
`+`	yığın boş → push	+	A
`B`	çıktıya	+	A B
`*`	üstteki `+` (1) < 2 → push	+ *	A B
`C`	çıktıya	+ *	A B C
`-`	`*` (2) ≥ 1 → çıktıya; `+` (1) ≥ 1 → çıktıya; push `-`	−	A B C * +
`D`	çıktıya	−	A B C * + D
`/`	üstteki `-` (1) < 2 → push	− /	A B C * + D
`E`	çıktıya	− /	A B C * + D E
(son)	önce `/`, sonra `-` çıktıya	boş	A B C * + D E / −

Tablo 7.21: `infixToPostfix("A+B*C-D/E")` izlemesi.

`*` okunduğunda üstteki `+`'nın önceliği daha düşüktür; `+` beklemeye devam eder ve `*` onun üstüne konur. Bu, “çarpma önce yapılır” kuralının yığındaki karşılığıdır. `-` okunduğunda iki işleç birden çıkar: önce daha yüksek öncelikli `*`, sonra eşit öncelikli `+`. Girdi bittiğinde yığın üstten alta boşaltılır: önce `/`, sonra `-`. Sonuç `A B C * + D E / -`, yani `(A + B·C) − D/E`'dir.



Şekil 7.20: `A+B*C-D/E` dönüşümünden dört an. (a) `*`, daha düşük öncelikli `+`'nın üstüne konmuş; çıktı A B C. (b) `-` geldi: `*` çıktıya alındı, şimdi eşit öncelikli `+` çıkıyor. (c) `/` `-`'nin üstünde, E çıktıya yazıldı. (d) Son: yığın boşaltıldı; sonuç A B C * + D E / −.

Sık yapılan hata

İşleç geldiğinde yığından yalnız *bir* işleç çıkarmak. Kural bir `while` döngüsüdür: koşul sağlandığı sürece çıkarma sürer. `A+B*C-D/E`'de `-` geldiğinde yalnız `*` çıkarılıp `+` bırakılsaydı sonuç `A B C * D E / - +` olurdu; bu da `A + (B·C − D/E)` demektir ve soldan birleşmeyi bozar.

7.5.7 Sonek ifadeyi değerlendirme

Sonek gösterimin ödülü, değerlendirmenin son derece basit olmasıdır. İfade soldan sağa okunur:

1. Sayı gelirse yığına `push` edilir.
2. İşleç gelirse yığından iki öğe `pop` edilir: **önce sağ, sonra sol işlenen**. İşlem yapılır ve sonuç yığına `push` edilir.
3. İfade bitince yığında tek öğe kalır: sonuç.

Parantez de öncelik de gerekmez; her simge bir kez işlenir ve süre $O(n)$'dir.

Yardımcı metot, iki işlenen ve bir işleçle hesabı yapar:

```
int hesapla(int sol, char op, int sag) {
  switch (op) {
    case '+': return sol + sag;
    case '-': return sol - sag;
    case '*': return sol * sag;
    default:  return sol / sag;
  }
}
```

Değerlendirme metodu:

```
int postfixDegerlendir(String ifade) {
  Deque<Integer> yigin = new ArrayDeque<>();
  for (char c : ifade.toCharArray()) {
    if (Character.isDigit(c)) {
      yigin.push(c - '0');
    } else {
      int sag = yigin.pop();
      int sol = yigin.pop();
      yigin.push(hesapla(sol, c, sag));
    }
  }
  return yigin.pop();
}
```

`Character.isDigit(c)` rakamları tanır. `c - '0'`, bir rakam karakterini sayı değerine çevirir: Java'da karakterler sayı kodlarıyla saklanır ve `'0'` 'dan `'9'` 'a kadar kodlar ardışıktır; `'7' - '0'` bu yüzden 7'dir. İşleç geldiğinde ilk `pop` `sag` 'a, ikinci `pop` `sol` 'a atanır ve `hesapla(sol, c, sag)` çağrılır. Döngü bitince son `pop` sonucu döndürür. Kodun varsayımları şunlardır: sayılar tek basamaklıdır, boşluk yoktur ve ifade geçerlidir. `/` tam sayı bölmesidir; `7 / 2` 3 verir.

Neden önce sağ işlenen çıkar? Sol işlenen ifadede daha önce okunmuş, dolayısıyla yığına daha önce konmuştur ve altta durur. Sağ işlenen en son konmuştur ve üsttedir. Toplama ve çarpmada sıra fark etmez, ama çıkarma ve bölmede eder: `5 2 -` ifadesinde 5 önce, 2 sonra push edilir. İlk `pop` 2'yi (`sag`), ikincisi 5'i (`sol`) verir ve sonuç *5 − 2 = 3*'tür, *2 − 5 = −3* değil.

Sık yapılan hata

İşlenen sırasını karıştırmak: ilk `pop` 'u sol işlenen sanmak. Kodda `int sol = yigin.pop(); int sag = yigin.pop();` yazılsaydı her çıkarma ve bölmenin sonucu ters hesaplanırdı. Kural: ilk `pop` sağ, ikinci `pop` sol işlenendir.

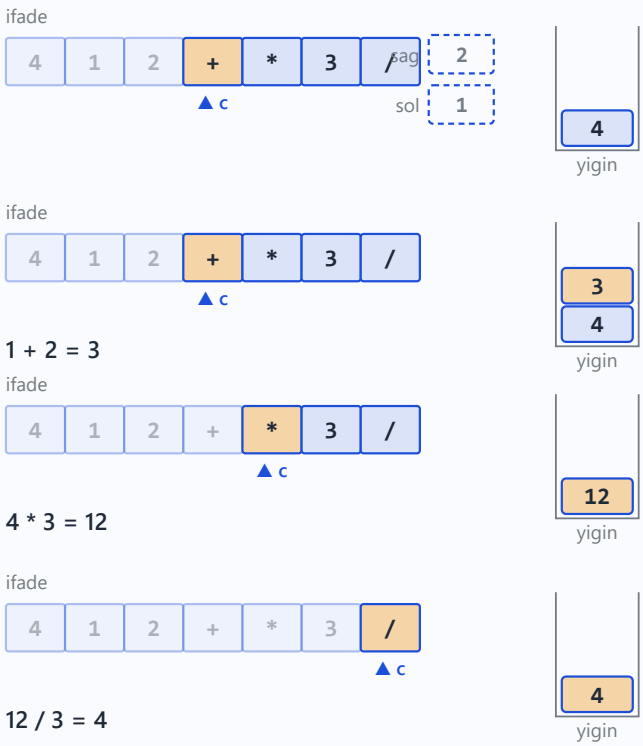
7.5.8 Kod izleme: postfixDegerlendir()

Örnek 7.6'daki `A B C + * D /` ifadesinde `A = 4`, `B = 1`, `C = 2`, `D = 3` konursa sonek dizgi `"412+*3/"` olur. Ara ek karşılığı *4 · (1 + 2) / 3 = 4*'tür.

`c`	İşlem	Yığın (sonra)
`4`	rakam → push 4	4
`1`	rakam → push 1	4 1
`2`	rakam → push 2	4 1 2
`+`	`sag` = 2, `sol` = 1; `hesapla(1, '+', 2)` = 3 → push	4 3
`*`	`sag` = 3, `sol` = 4; `hesapla(4, '*', 3)` = 12 → push	12
`3`	rakam → push 3	12 3
`/`	`sag` = 3, `sol` = 12; `hesapla(12, '/', 3)` = 4 → push	4
(son)	`return yigin.pop()`	boş → sonuç 4

Tablo 7.22: `postfixDegerlendir("412+*3/")` izlemesi. Yığın alttan üste yazılmıştır.

Yığın, değerlendirme boyunca "henüz bir işlece işlenen olmamış ara sonuçları" tutar. `+` geldiğinde 1 ve 2 bu rolü bitirir, yerlerini 3 alır. Yığındaki öğe sayısının seyri (1, 2, 3, 2, 1, 2, 1) ifadenin geçerliliği hakkında da bilgi verir: her sayı yığını bir büyütür, her ikili işleç bir küçültür. Geçerli bir sonek ifadede her işleç geldiğinde yığında en az 2 öğe bulunur ve sayı sonda tam 1'e iner.



Şekil 7.21: `"412+*3/"` değerlendirmesinden dört an. (a) `+` geldi: önce `sag` = 2, sonra `sol` = 1 çıkıyor. (b) 1 + 2 = 3 yığına kondu. (c) 4 · 3 = 12 yığına kondu. (d) 12 / 3 = 4; yığında tek değer kaldı.

7.5.9 Ara ekten öneke

Ara ek ifadeyi önek biçime çevirmek için yeni bir algoritma yazmak gerekmez. Soneke dönüşüm algoritması, küçük bir değişiklikle ve iki ters çevirme adımıyla birlikte kullanılır:

1. Ara ek ifade ters çevrilir.
2. Parantezler değiştirilir: `(` ↔ `)`.
3. Soneke dönüşüm algoritması uygulanır; ama **eşit öncelikte çıkarma yapılmaz**, yalnız önceliği daha büyük (>) olanlar çıkarılır.
4. Çıktı ters çevrilir.

Adım	Sonuç
1. Ters çevir	`D / ) C + B ( * A`
2. Parantezleri değiştir	`D / ( C + B ) * A`
3. Soneke dönüştür (eşitte çıkarmadan)	`D C B + A * /`
4. Ters çevir → önek	`/ * A + B C D`

Tablo 7.23: `A * (B + C) / D` ifadesinin öneke dönüşümü.

Yöntemin mantığı şudur: önek gösterim, ifadenin “aynadaki” sonekidir. İfade ters çevrilince işleçlerin işlenenlerine göre konumu da ters döner; sağdaki işlenen sola geçer. Ters çevrilmiş ifadenin soneki bulunup yeniden ters çevrilince işleç işlenenlerinin önüne geçer. Ters çevirme parantezlerin yönünü de bozduğu için 2. adımda düzeltilir.

3. adımdaki “eşitte çıkarma yok” kuralı birleşme yönünü korumak içindir. Ters çevirmek soldan birleşmeyi sağdan birleşmeye dönüştürür. Ters ifadede `*` geldiğinde üstte eşit öncelikli `/` vardır; `/` çıkarılmaz ve `*` onun üstüne konur. Böylece `*` önce çıkar ve önce yapılır. Ters çevirince işlem sırası önce `A * (B + C)`, sonra `/ D` olarak korunur. Eşitte çıkarma yapılsaydı 3. adımın çıktısı `D C B + / A *`, önek de `* A / + B C D` olurdu; bu, 7.5.1'de gösterilen yanlış önekin ta kendisidir.

İkinci örnek `A + B * C - D / E` için aynı yöntem `- + A * B C / D E` verir; bu da Örnek 7.6'daki elle bulunan sonuçtur.

7.5.10 Kod izleme: infixToPrefix()

```
String ters(String s) {
  return new StringBuilder(s).reverse().toString();
}
String infixToPrefix(String ifade) {
  String t = ters(ifade);          // 1. ters çevir
  t = parantezCevir(t);            // 2. ( ⇔ )
  String p = prefixIcinPostfix(t); // 3. eşitte pop yok
  return ters(p);                  // 4. ters çevir
}
// main içinde:
System.out.println(infixToPrefix("A*(B+C)/D"));
```

Metot dört adımı dört satırda yapar. `ters`, `StringBuilder`'ın hazır `reverse()` metoduyla dizgiyi ters çevirir. `parantezCevir`, her `(`'ı `)`'ya ve her `)`'yı `(`'a çevirir; sunumda gövdesi verilmeyen bu yardımcı şöyle yazılabilir:

```
String parantezCevir(String s) {
  StringBuilder sb = new StringBuilder();
  for (char c : s.toCharArray()) {
    if (c == '(') sb.append(')');
    else if (c == ')') sb.append('(');
    else sb.append(c);
  }
  return sb.toString();
}
```

`prefixIcinPostfix`, `infixToPostfix`'in kopyasıdır; tek fark işleç dalındaki karşılaştırmadır. Döngüsü şöyledir:

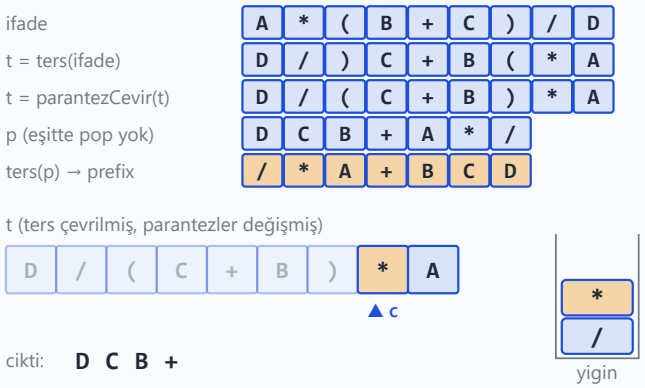
```
for (char c : ifade.toCharArray()) {
  if (Character.isLetterOrDigit(c)) cikti.append(c);
  else if (c == '(') yigin.push(c);
  else if (c == ')') {
    while (yigin.peek() != '(')
      cikti.append(yigin.pop());
    yigin.pop();                   // '(' atılır
  } else {                         // işleç
    while (!yigin.isEmpty() &&
        oncelik(yigin.peek()) > oncelik(c))
      cikti.append(yigin.pop());
    yigin.push(c);
  }
}
while (!yigin.isEmpty()) cikti.append(yigin.pop());
```

Burada `>=` yerine `>` yazılmıştır: eşit öncelikli işleç yığında kalır. Ters çevrilmiş ve parantezleri değiştirilmiş `"D/(C+B)*A"` girdisinin izlemesi şöyledir:

`c`	İşlem	Yığın (sonra)	`cikti`
`D`	çıktıya	boş	D
`/`	yığın boş → push	/	D
`(`	push	/ (	D
`C`	çıktıya	/ (	D C
`+`	üstte `(` → push	/ (+	D C
`B`	çıktıya	/ (+	D C B
`)`	`+` çıktıya, `(` atılır	/	D C B +
`*`	üstteki `/` (2) > 2? hayır → push	/ *	D C B +
`A`	çıktıya	/ *	D C B + A
(son)	önce `*`, sonra `/` çıktıya	boş	D C B + A * /

Tablo 7.24: `prefixIcinPostfix("D/(C+B)*A")` izlemesi.

`p = "DCB+A*/"` ters çevrilince `"/*A+BCD"` elde edilir ve `main` bunu yazdırır. Sonucu denetlemek için işlenenlere bakılabilir: A, B, C, D sırası korunmuştur.



Şekil 7.22: Öneke dönüşüm. (a) Dört aşama alt alta: `ifade`, ters çevrilmiş `t`, parantezleri değiştirilmiş `t`, eşitte pop yapılmadan bulunan `p` ve ters çevrilmiş sonuç `/ * A + B C D`. (b) 3. adımın kritik anı: `*` geldiğinde üstteki `/` eşit öncelikli olduğu için çıkarılmıyor; `*` onun üstüne konuyor.

Dört adımın her biri *O(n)*'dir: iki ters çevirme, bir parantez değişimi ve bir dönüşüm. Toplam süre de *O(n)*'dir.

7.5.11 Önek ifadeyi değerlendirme

Önek ifade de bir yığınla değerlendirilir. Kurallar sonekteki gibidir, iki farkla: ifade **sağdan sola** okunur ve bir işleç geldiğinde **ilk çıkan sol işlenendir**. Sağdan sola okunduğunda sağ işlenen önce görülür ve yığına önce konur; sol işlenen sonra konur ve üstte durur.

Örnek 7.7

`- * 3 4 / 8 2` önek ifadesini değerlendirelim (ara ek karşılığı *3 · 4 − 8 / 2*). Simgeler sağdan sola okunur; yığın alttan üste yazılmıştır.

1. `2` → push. Yığın: 2.
2. `8` → push. Yığın: 2 8.
3. `/` → `sol` = 8 (ilk pop), `sag` = 2; *8 / 2 = 4* → push. Yığın: 4.
4. `4` → push. Yığın: 4 4.
5. `3` → push. Yığın: 4 4 3.
6. `*` → `sol` = 3, `sag` = 4; *3 · 4 = 12* → push. Yığın: 4 12.
7. `-` → `sol` = 12, `sag` = 4; *12 − 4 = 8* → push. Yığın: 8.

Sonuç 8'dir. 7. adımda sıra karıştırılsaydı *4 − 12 = −8* bulunurdu.

Sonek ve önek değerlendirme, aynı yığın fikrinin iki yönlü uygulamasıdır. İkisinde de yığın, bir işleci bekleyen işlenenleri tutar; fark yalnız okuma yönünde ve dolayısıyla işlenenlerin yığından çıkış sırasındadır.

7.6 Sık yapılan hatalar ve ortak desen

Yığınla ilgili hataların çoğu birkaç tipe ayrılır. Not boyunca her konunun yanında verilen hatalar burada bir arada toplanmıştır; bir yığın kodunu gözden geçirirken kontrol listesi olarak kullanılabilir.

Hata	Ne olur?	Doğrusu
Boş yığında `pop` / `peek`	Kütüphaneye göre istisna ya da `null`; parantez kontrolünde `"())"` tam bu hatayı tetikler.	Önce `isEmpty()` denetlenir.
`top` ile `size` karışıklığı	Dizinin dışına yazma ya da yanlış öğe sayısı.	`top` son öğenin indisidir; `size = top + 1`; dolu koşulu `top == kapasite - 1`.
Çıkarılan hücreyi boşaltmamak	Nesne çöp toplayıcıdan kaçar (bellek sızıntısı, *loitering*).	Nesne dizisinde `veri[top--] = null`.
Sonek değerlendirmede işlenen sırası	Çıkarma ve bölme ters hesaplanır.	İlk `pop` sağ, ikinci `pop` sol işlenendir: `5 2 -` = 5 − 2 = 3.
Yineleme sırası	Aynı içerikli iki yığın ters sırada yazdırılır.	`ArrayDeque` üstten alta, eski `Stack` alttan üste gezinir; hangisinin kullanıldığı bilinmelidir.
Sonsuz özyineleme	Çerçeveler birikir, `StackOverflowError` (çağrı yığını taşar).	Her özyinelemeli metodun ulaşılabilir bir taban durumu olmalıdır.

Tablo 7.25: Yığın kodunda sık görülen hatalar.

Yineleme sırası hatası, kütüphane sınıflarının iç yapısından gelir. `java.util.Stack`, `Vector`'dan türediği için öğeleri eklenme sırasıyla bir dizide tutar; `toString()` ve `for-each` döngüsü onları alttan üste verir. `ArrayDeque` ise `push`'ta öğeyi başa ekler; gezinme baştan, yani üstten başlar. 1, 2, 3 sırasıyla push edilen iki yığından `Stack` `[1, 2, 3]`, `ArrayDeque` `[3, 2, 1]` yazdırır. İçerik aynıdır, ama çıktılar ters görünür.

Bölümdeki bütün problemlere geri dönüp bakıldığında tek bir desen görülür. Yığın, şu soru sorulduğunda doğru araçtır: **"En son görülen ve henüz sonuçlanmamış öğe hangisi?"** Parantez kontrolünde bu öğe en son açılmış ama henüz kapanmamış parantezdir. Çağrı yığınında en son çağrılmış ama henüz dönmemiş metottur. Sonek değerlendirmede henüz bir işlece işlenen olmamış en son ara sonuçtur. Sonraki büyük öğe ve stock span'de, henüz "aşılmamış" en yakın adaydır. Hanoi'de, henüz yapılmamış en son alt görevdir. Bir problemde bu soru beliriyorsa, çözümün bir yığın kullanması büyük olasılıktır.

7.7 Özet

- **Yığın**, ekleme ve çıkarmanın yalnız üstten yapıldığı bir ADT'dir ve **LIFO** (son giren ilk çıkar) düzeniyle çalışır. Temel işlemleri `push`, `pop`, `top` / `peek`, `isEmpty` ve `size`'dır; hepsi *O(1)*'dir. Bir öğeyi aramak *O(n)*'dir; *n* öğeli yığın *O(n)* bellek kullanır.
- Boş yığından çıkarma **alt taşma**, dolu sınırlı yığına ekleme **taşmadır**. Boş yığındaki davranış kütüphaneden kütüphaneye değişir; `pop` ve `peek`'ten önce `isEmpty()` denetlenir.
- **Dizi tabanlı yığında** `top` son öğenin indisidir, boş yığında −1'dir ve `size = top + 1`'dir. Nesne dizisinde çıkarılan hücreye `null` atanır. **Dinamik dizi** dolunca iki katına büyütülür; büyütme yapan `push` *O(n)*, ama *n* `push`'un toplamı *3n*'den az olduğu için `push` **amortize *O(1)***'dir. Sabit miktarla büyütme *O(n^2)* toplam maliyet verir.
- **Bağlı liste tabanlı yığında** üst, listenin başıdır; `push` başa ekler, `pop` baştan çıkarır, ikisi de *O(1)*'dir. Taşma yalnız bellek bitince olur; her düğüm ek bir referans taşır. Java'da yeni kodda `Deque<E> y = new ArrayDeque<>()` önerilir.
- Tarayıcıda **geri/ileri** iki yığınla, metot çağrıları **çağrı yığınıyla** yönetilir. Özyinelemenin derinliği kadar bellek gerekir; taban durumu olmayan özyineleme `StackOverflowError` verir.
- **Parantez kontrolü**, eşleşmemiş açık parantezleri yığında tutar ve *O(n)* zamanda üç hata türünü (yanlış sıra, kapanmamış, fazla kapanış) yakalar.
- **Sonraki büyük öğe** ve **stock span**, alttan üste azalan bir **monoton yığınla** *O(n)*'de çözülür, çünkü her öğe bir kez eklenir ve en çok bir kez çıkarılır. `getMin()`'i *O(1)* yapan yığın, ikinci bir yığında en küçükleri (`<=` ile) tutar.
- **Hanoi kuleleri**, alt görevleri ters sırayla push eden açık bir görev yığınıyla özyinelemesiz çözülür: $2^n - 1$ hamle, *O(n)* yığın.
- **Ara ek** ifade, işleç yığınıyla *O(n)*'de **soneke** çevrilir: üstteki işlecin önceliği ≥ gelenin önceliği olduğu sürece çıkarılır. **Öneke** dönüşüm: ters çevir, parantezleri değiştir, eşitte çıkarmadan soneke çevir, yeniden ters çevir. İşlenenlerin sırası üç gösterimde de aynıdır.

- **Sonek değerlendirmede** ifade soldan sağa okunur; işleçte ilk `pop` sağ, ikinci `pop` sol işlenendir. Önek ifade sağdan sola okunur ve ilk çıkan sol işlenendir.

Problem	Yığında tutulan	Zaman
`push` / `pop` / `top` / `isEmpty` / `size`	—	O(1)
Dizgi ters çevirme	karakterler	O(n)
Parantez kontrolü	eşleşmemiş açık parantezler	O(n)
Sonraki büyük öğe	sağdaki aday değerler (azalan)	O(n)
Stock span	henüz aşılmamış günlerin indisleri	O(n)
Hanoi kuleleri	alt görevler (n, kaynak, yardımcı, hedef)	$O(2^n)$
Ara ek → sonek / önek	işleçler ve `(`	O(n)
Sonek değerlendirme	ara sonuçlar	O(n)
Min yığın (`getMin`)	her an en küçük değer	O(1)
Geri / ileri gezinme	ziyaret edilen sayfalar (iki yığın)	O(1)

Tablo 7.26: Yığınla çözülen problemler.

7.8 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

yığın *stack* · üst *top* · push (üste ekleme) *push* · pop (üstten çıkarma) *pop* · LIFO *last-in, first-out* · soyut veri tipi *abstract data type (ADT)* · arayüz *interface* · jenerik *generic* · taşma *overflow* · alt taşma *underflow* · kapasite *capacity* · dinamik dizi *dynamic array* · amortize analiz *amortized analysis* · bağlı liste *linked list* · tek yönlü bağlı liste *singly linked list* · düğüm *node* · referans *reference* · geri al *undo* · çağrı yığını *call stack* · yığın çerçevesi *stack frame* · özyineleme *recursion* · taban durum *base case* · geri izleme *backtracking* · parantez kontrolü *balanced parentheses* · sonraki büyük öğe *next greater element* · stock span *stock span* · ara ek *infix* · önek *prefix* · sonek *postfix* · derinlik öncelikli arama *depth-first search (DFS)* · zaman karmaşıklığı *time complexity* · alan karmaşıklığı *space complexity*

7.9 Kendini Dene

1. Boş bir yığına sırayla `push(7)`, `push(2)`, `pop()`, `push(9)`, `top()`, `push(4)`, `pop()`, `pop()`

uygulanıyor. Değer döndüren çağrılar ne döndürür ve sonda yığında ne kalır?

Cevap

İlk `pop()` 2'yi döndürür; yığın: 7. `push(9)` 'dan sonra yığın 7 9'dur ve `top()` 9'u çıkarmadan döndürür. `push(4)` 'ten sonra 7 9 4; ardından `pop()` 4'ü, sonraki `pop()` 9'u döndürür. Dönen değerler sırasıyla 2, 9, 4, 9'dur; sonda yığında yalnız 7 kalır.

2. `gecerliMi("([]{})")` ve `gecerliMi("([]{)}")` ne döndürür? Geçersiz olanda hata hangi karakterde fark edilir?

Cevap

İlki `true` döndürür: `(` , `[` push; `]` `[` 'i çıkarır; `{` push; `}` `{` 'yi çıkarır; `)` `(` 'ı çıkarır; sonda yığın boştur. İkincisi `false` döndürür: 4. indisteki `)` geldiğinde yığın ({ durumundadır ve pop `{` 'yi verir; `eslesir('{', ')')` yanlıştır. Son karakter `}` hiç okunmaz.

3. `(A + B) * C - D / (E + F)` ifadesinin sonek ve önek karşılıkları nedir?

Cevap

Tam parantezleme: `(((A + B) * C) - (D / (E + F)))` . Sonek: `A B + C * D E F + / -` . Önek: `- * + A B C / D + E F` . Denetim: işlenenler iki gösterimde de A, B, C, D, E, F sırasındadır. `infixToPostfix` izlenirse `-` geldiğinde yığındaki `*` çıkarılır, `/` geldiğinde ise üstteki `-` daha düşük öncelikli olduğu için kalır.

4. `7 2 - 3 * 8 4 / +` sonek ifadesinin değeri nedir? Her işleçten sonra yığında ne vardır?

Cevap

`-` : sağ 2, sol 7 → 5; yığın 5. `*` : sağ 3, sol 5 → 15; yığın 15. `/` : sağ 4, sol 8 → 2; yığın 15 2. `+` : sağ 2, sol 15 → 17. Değer 17'dir; ara ek karşılığı *(7 − 2) · 3 + 8 / 4*'tür.

5. Fiyatlar `[50, 40, 45, 60, 55, 70]` için span dizisi nedir? `span` metodu çalışırken toplam kaç `pop` yapılır?

Cevap

Span: `[1, 1, 2, 4, 1, 6]` . `i = 2` 'de 45, 1. günü (40) atar ve 0. günde (50) durur: 2 − 0 = 2. `i = 3` 'te 60, 2. ve 0. günleri atar, yığın boşalır: 3 + 1 = 4. `i = 5` 'te 70, 4. ve 3. günleri atar, yığın boşalır: 6. Toplam 5 `pop` yapılır (1, 2, 0, 4, 3 indisleri); bu, eklenen 6 indisten azdır.

6. `{3, 8, 2, 5, 1, 9}` dizisi için sonraki büyük öğe dizisi nedir? `i = 1` adımında yığından hangi

değerler atılır?

Cevap

Sonuç `[8, 9, 5, 9, 9, -1]`'dir. Sağdan sola: 9 → −1; 1 → 9; 5, 1'i atar → 9; 2 → 5; `i = 1`'de 8 gelir, yığın alttan üste 9 5 2'dir ve 8, önce 2'yi sonra 5'i atar; üstte kalan 9 cevaptır. Son olarak 3 → 8.

7. Kapasitesi 2 olan ve dolunca iki katına büyüyen boş bir yığına 12 kez `push` yapılıyor. Kaç büyütme ve kaç kopyalama yapılır? Toplam maliyet *3n* sınırının altında mıdır?

Cevap

Büyütmeler 3. `push`'ta (2 → 4, 2 kopya), 5. `push`'ta (4 → 8, 4 kopya) ve 9. `push`'ta (8 → 16, 8 kopya) yapılır: 3 büyütme, 14 kopya. Yazma sayısı 12'dir; toplam 26'dır ve $3 \cdot 12 = 36$'nın altındadır. Son kapasite 16'dır.

8. Yığınla yazılmış `hanoi(2)` hangi hamleleri yazdırır? İlk turdan sonra yığında (üstten alta) hangi görevler vardır?

Cevap

İlk turda `(2,A,B,C)` bölünür ve yığın üstten alta `(1,A,C,B)`, `(1,A,B,C)`, `(1,B,A,C)` olur. Sonraki üç turda bu görevler sırayla yapılır: A → B, A → C, B → C. Toplam $2^2 - 1 = 3$ hamle.

9. Min yığında `push`'taki koşul `x <= min.peek()` yerine `x < min.peek()` yazılmış olsun. Boş yığına `push(2)`, `push(2)`, `pop()`, `getMin()` uygulanırsa ne olur?

Cevap

İlk 2 `min`'e eklenir; ikinci 2 için `2 < 2` yanlıştır ve eklenmez. `pop()` 2'yi çıkarır ve `2 == min.peek()` doğru olduğu için `min`'deki tek 2 de silinir. `y`'de hâlâ 2 vardır, ama `min` boştur. `getMin()`'de `min.peek()` `null` döndürür; bu değer `int`'e açılırken `NullPointerException` fırlatılır.

10. `/ * 4 + 1 2 3` önek ifadesinin değeri nedir?

Cevap

Sağdan sola: 3, 2, 1 push edilir (yığın 3 2 1). `+`: sol 1 (ilk pop), sağ 2 → 3; yığın 3 3. 4 push; yığın 3 3 4. `*`: sol 4, sağ 3 → 12; yığın 3 12. `/`: sol 12, sağ 3 → 4. Değer 4'tür; ara ek karşılığı $4 \cdot (1 + 2) / 3$'tür.

7.10 İleri okuma

- M. T. Goodrich, R. Tamassia ve M. H. Goldwasser, *Data Structures and Algorithms in Java* (6. baskı, 2014), Kesim 6.1: yığın ADT'sini, dizi ve bağlı liste gerçeklemelerini ve parantez eşleme ile HTML etiketi denetimini bu nottakine yakın bir Java koduyla anlatır.
- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 1.3: yığınları dinamik dizi ve bağlı listeyle gerçekler; Dijkstra'nın iki yığınla ifade değerlendirme algoritmasını kısa ve açık bir örnekle verir.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (3. baskı, 2009), Kesim 10.1 ve 17.4: yığının dizi gerçeklemesini ve dinamik tabloların amortize analizini daha biçimsel bir dille işler.

7.11 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3. baskı, MIT Press, 2009.
2. E. W. Dijkstra, “Algol 60 translation: An Algol 60 translator for the X1 and Making a translator for Algol 60”, Rapor MR 35/61, Mathematisch Centrum, Amsterdam, 1961.
3. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6. baskı, Wiley, 2014.
4. R. Lafore, *Data Structures and Algorithms in Java*, 2. baskı, Sams, 2002.
5. Oracle, *Java SE 17 API Specification*: `java.util.Deque` ve `java.util.ArrayDeque` belgeleri.
6. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011.

KISIM II · BÖLÜM 8

Kuyruklar ve Deque

Kuyruk, günlük hayattaki sıranın bilgisayardaki karşılığıdır: önce gelen önce hizmet alır. Öğeler bir uçtan girer, öbür uçtan çıkar ve hiçbiri sırayı atlayamaz. Bu sade kural, işletim sistemlerinde işlemcinin süreçler arasında paylaştırılmasından yazıcıya gönderilen belgelere, ağ paketlerinden çizgelerde genişlik öncelikli aramaya kadar pek çok yerde karşımıza çıkar. Not, kuyruğun soyut tanımıyla ve Java'daki `Queue` arayüzüyle başlar. Ardından kuyruğun diziyle nasıl gerçekleneceğini, basit dizi kuyruğunun neden yer israf ettiğini ve dairesel kuyruğun bu sorunu mod işlemiyle nasıl çözdüğünü anlatır. Bağlı liste ile kuyruk, kuyruk türleri, iki uçtan da ekleme ve çıkarma yapılabilen çift uçlu kuyruk (deque) ve onunla çözülen kayan pencerede en büyük problemi bunu izler. Son kesimde round robin çizelgeleme, Josephus problemi, ikilik sayı üretimi ve iki yığınla kuyruk gibi uygulamalar kod izlemeleriyle işlenir.

Bu bölümün kodu: `github.com/sercankulcu/data-structures-java/Bolum08/src`

ÖĞRENME ÇIKTILARI

FIFO modeli
Kuyruğun ilk giren ilk çıkar davranışını açıklamak; bir işlem dizisinden sonra kuyruğun durumunu ve dönen değerleri bulmak.

Gerçekleme
Kuyruğu dizi ve bağlı liste kullanarak gerçeklemek; dairesel dizide indisleri mod işlemiyle hesaplamak, boş ve dolu durumlarını ayırt etmek ve dolu kuyruğu sırayı bozmadan büyütmek.

Kuyruk türleri
Dairesel kuyruk ile çift uçlu kuyruğu (deque) ayırt etmek; çift uçlu kuyruğu çift yönlü bağlı listeyle gerçeklemek ve uygun yerde kullanmak.

Uygulamalar
Uygun kuyruk türünü gerçek problemlerde seçmek: round robin çizelgeleme, Josephus problemi, ikilik sayı üretimi, kayan pencerede en büyük ve iki yığınla kuyruk; her çözümün maliyetini gerekçesiyle vermek.

8.1 Kuyruk ADT'si

Bir veri yapısı iki düzeyde anlatılabilir. Birinci düzey yapının *ne yaptığını* söyler: hangi işlemler vardır, her işlem neyi değiştirir ve ne döndürür. İkinci düzey bunun *nasıl yapıldığını* söyler: veriler bellekte hangi biçimde durur, her işlem hangi adımlarla yürür. Birinci düzeydeki tarife **soyut veri tipi** (*abstract data type*, kısaca ADT) denir. Bu kesim kuyruğu soyut düzeyde, yani gerçeklemeden bağımsız olarak tanıtır.

8.1.1 Kuyruk nedir?

Market kasasındaki sırayı düşünelim. Kasaya yeni gelen müşteri sıranın sonuna geçer. Kasiyer her seferinde sıranın en önündeki müşteriye hizmet verir. Hizmeti biten müşteri sıradan ayrılır, arkasındaki öne geçer. Sıranın ortasına girmek ya da ortadan birini öne almak kurala aykırıdır. Bilet gişesi, banka veznesi ve otoyol gişesi de aynı kuralla işler: önce gelen önce hizmet alır.

Bilgisayar bilimindeki **kuyruk** (*queue*) bu düzeni taklit eder. Öğeler bir sıra hâlinde tutulur ve sıranın iki ucu ayrı işlere ayrılır. Yeni öğe her zaman **arka** (*rear*) uca eklenir. Çıkarma ise her zaman **ön** (*front*) uçtan yapılır. Böylece kuyruktan çıkan öğe, o anda kuyrukta bekleyenler arasında en uzun süredir bekleyen, yani en eski öğedir.

> **Tanım 8.1**
>
> **Kuyruk**, öğeleri bir sıra içinde tutan ve eklemenin yalnız arka uçtan, çıkarmanın yalnız ön uçtan yapıldığı soyut veri tipidir. Bu düzene **FIFO** (*First-In, First-Out*: ilk giren ilk çıkar) denir.

Kuyruğu bir önceki konudaki yığınla karşılaştırmak öğreticidir. Yığında ekleme ve çıkarma aynı uçta, üstte yapılır; en son giren ilk çıkar (LIFO). Kuyrukta ise ekleme ve çıkarma ayrı uçlardadır; en önce giren ilk çıkar. Bir yapıya 1, 2, 3 sırasıyla öğe eklenip üç kez çıkarma yapılsın. Yığından 3, 2, 1 çıkar; giriş sırası tersine döner. Kuyruktan 1, 2, 3 çıkar; giriş sırası aynen korunur. Kuyruğun asıl işi budur: gelen işleri *geliş sırasıyla* saklamak ve aynı sırayla teslim etmek.

Kuyruğun her iki ucu da açıktır, ama her uç tek yönde kullanılır. Bu yüzden kuyruk **doğrusal** bir yapıdır: her öğenin (ilk öğe dışında) tek bir öncülü, (son öğe dışında) tek bir ardılı vardır. Öğeler arasında dallanma yoktur.

> **Sık yapılan hata**
>
> Ekleme ve çıkarma uçlarını karıştırmak. Kuyruğa yeni gelen öğe öne değil, *arkaya* eklenir. Çıkan öğe ise en yeni değil, *en eski* öğedir. “Son eklenen önce çıkar” diye düşünen biri kuyruğu yığın gibi kullanmış olur.

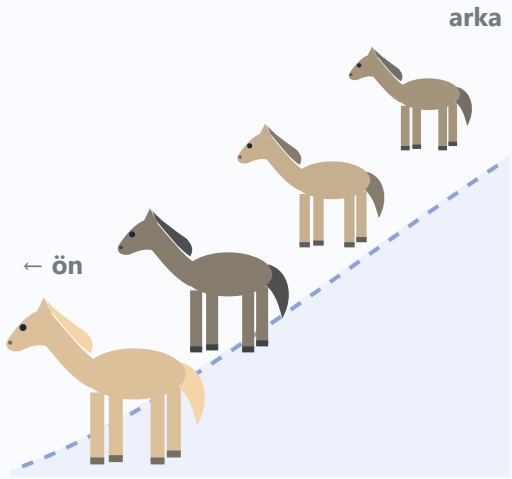
8.1.2 Temel işlemler

Kuyruğun iki temel işlemi vardır. Adları İngilizcede kalıplaşmıştır ve ders boyunca aynen kullanılır.

- **Ekleme** (`enqueue`): Kuyruğun sonuna, yani arkasına yeni bir öğe ekler. "Sıraya girmek" olarak düşünülebilir.
- **Çıkarma** (`dequeue`): Kuyruğun başındaki, yani önündeki öğeyi çıkarır ve döndürür. İlk giren öğe ilk çıkar.

İki işlem de iyi bir gerçeklemede *O(1)* zamanlıdır: kuyrukta kaç öğe olursa olsun sabit sayıda adımda biter. Bunun için gerçeklemenin iki ucu da doğrudan, aramaya gerek kalmadan bulabilmesi gerekir. Dizi ve bağlı liste gerçeklemelerinin bütün tasarımı bu tek amaca hizmet eder.

Kuyrukta ortadaki bir öğeye doğrudan erişilmez; yalnız ön ve arka uçlar kullanılır. Bu bir eksiklik değil, bilinçli bir kısıtlamadır. Kuyruğu kullanan kod sırayı bozamaz, öğe atlatamaz. Kısıtlama aynı zamanda gerçeklemeyi de kolaylaştırır: yalnız iki uç hızlı olmak zorundadır.



Şekil 8.1: Tek sıra yürüyen atlar kuyruğa iyi bir benzetmedir. Öndeki at önce geçer; yeni gelen at sıranın sonuna katılır. Hiçbir at önündekini sollayamaz.

8.1.3 Kuyruk ADT'sinin işlemleri ve arayüzü

İki temel işleme birkaç yardımcı işlem eklenince kuyruk ADT'si tamamlanır. Tablo 8.1 bu işlemleri özetler.

İşlem	Açıklama
`enqueue(e)`	`e` 'yi arkaya ekler.
`dequeue()`	Öndeki öğeyi çıkarır ve döndürür.
`first()` / `peek()`	Öndeki öğeyi döndürür, çıkarmaz.
`rear()`	Arkadaki öğeyi döndürür, çıkarmaz.

İşlem	Açıklama
`size()`	Öğe sayısını döndürür.
`isEmpty()`	Kuyruk boş mu?
`isFull()`	Kuyruk dolu mu? (Yalnız sabit kapasiteli gerçeklemede.)

Tablo 8.1: Kuyruk ADT'sinin işlemleri.

Bu işlemlerden yalnız `enqueue` ve `dequeue` kuyruğu değiştirir. Ötekiler yalnız bilgi verir. `first()` ile `dequeue()` arasındaki fark yığındaki `top()` ile `pop()` arasındaki farka benzer: ikisi de öndeki öğeyi döndürür, ama `first()` onu yerinde bırakır. Öndeki öğeye bakıp karar vermek, sonra gerekiyorsa çıkarmak gereken durumlarda önce `first()` kullanılır.

`isFull()` yalnız kapasitesi baştan belirlenmiş gerçeklemelerde anlamlıdır. Bağlı liste ile gerçeklenen ya da dolunca büyüyen bir kuyruk hiçbir zaman "dolu" olmaz; bellek yettiği sürece yeni öğe kabul eder.

ADT'yi Java'da yazmanın doğal yolu bir **arayüz** (*interface*) tanımlamaktır. Arayüz yalnız metotların imzalarını verir, gövdelerini vermez. Aşağıdaki arayüz, Goodrich, Tamassia ve Goldwasser'in ders kitabındaki kuyruk arayüzünün kısaltılmış biçimidir:

```
public interface Queue<E> {
    int size();
    boolean isEmpty();
    void enqueue(E e);
    E first();    // boşsa null
    E dequeue();  // boşsa null
}
```

`Queue<E>` içindeki `E` bir tür parametresidir: kuyruğun hangi türden öğe tutacağı, kuyruk oluşturulurken belirlenir. `Queue<Integer>` tam sayı, `Queue<String>` dizgi tutan bir kuyruktur. Böyle tür parametresi alan sınıf ve arayüzlere **jenerik** (*generic*) denir. Arayüzde `rear()` ve `isFull()` yoktur. İlki kuyruğun temel kullanımında gerekmez; ikincisi ise arayüzün sınırsız bir kuyruğu da kapsaması için bilerek dışarıda bırakılmıştır.

Yorum satırları önemli bir sözleşmeyi kaydeder: kuyruk boşken `first()` ve `dequeue()` `null` döndürür. Boş kuyruktan çıkarma denemesine **alt taşma** (*underflow*), sabit kapasiteli dolu bir kuyruğa ekleme denemesine **taşma** (*overflow*) denir. İkisi de hata durumudur. Hatanın nasıl bildirileceği ise bir tasarım kararıdır. Goodrich'in arayüzü boş kuyrukta `null` döndürür; bazı gerçeklemeler istisna (*exception*) fırlatır. Bu notta dizi tabanlı dairesel kuyruk ve bağlı liste kuyruğu istisna fırlatır, çift uçlu kuyruk ise `null` döndürür. Seçilen yol, kodu kullanan programcıyı doğrudan ilgilendirir: `null` dönüşü denetlenmezse hata sessizce ilerler, istisna ise programı hemen durdurur.

Sık yapılan hata

`dequeue()` çağırmadan önce kuyruğun boş olup olmadığını denetlememek. `null` döndüren bir gerçeklemede dönen değer hemen kullanılırsa (örneğin `int x = q.dequeue();`) `null` değeri `int`'e açılamaz ve `NullPointerException` fırlatılır. Güvenli kalıp önce `isEmpty()` sormaktır.

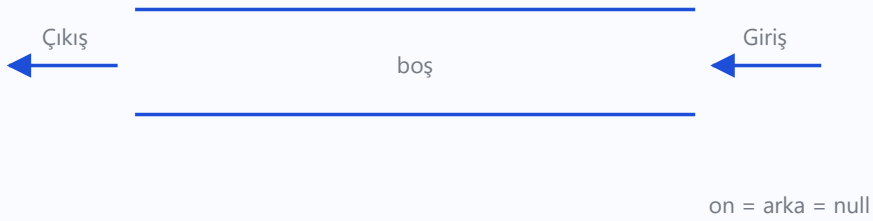
8.1.4 Kuyruk işlemleri adım adım

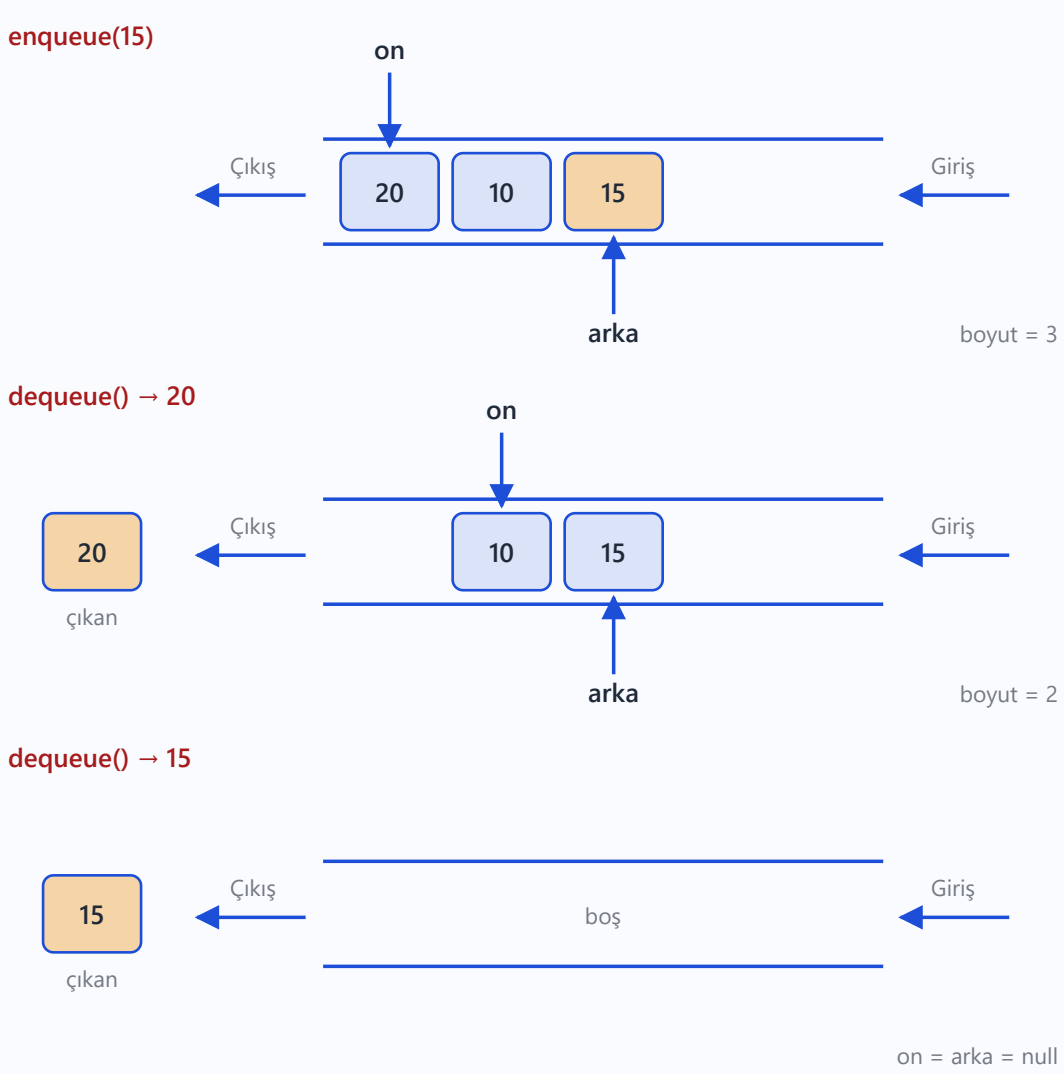
İşlemlerin birlikte nasıl çalıştığını küçük bir işlem dizisi üzerinde izleyelim. Boş bir kuyruğa sırayla `enqueue(20)`, `enqueue(10)`, `enqueue(15)` ve üç kez `dequeue()` uygulanıyor. Kuyruğun önünü gösteren referansın adı `on`, arkasını gösteren referansın adı `arka`'dır. Java'da değişken adında Türkçe karakter kullanılabilse de kodlarda alışkanlık olarak "ö" yerine "o" yazılır.

Örnek 8.1

1. **Başlangıç.** Kuyruk boştur. `on` ve `arka` hiçbir öğeyi göstermez: ikisi de `null`'dır.
2. `enqueue(20)`: 20 boş kuyruğa girer. Tek öğe olduğu için hem öndedir hem arkadadır: `on` ve `arka` aynı öğeyi gösterir. Boyut 1'dir.
3. `enqueue(10)`: 10 arkaya eklenir. `arka` 10'a ilerler, `on` 20'de kalır. Boyut 2'dir.
4. `enqueue(15)`: 15 arkaya eklenir. Kuyruk önden arkaya 20, 10, 15'tir. Boyut 3'tür.
5. `dequeue()`: Öndeki 20 çıkar ve döndürülür. `on` bir sonraki öğeye, 10'a ilerler. Boyut 2'dir.
6. `dequeue()`: 10 çıkar. Artık `on` ve `arka` aynı öğeyi, 15'i gösterir. Boyut 1'dir.
7. `dequeue()`: 15 çıkar. Kuyruk boşalır; `on` ve `arka` yeniden `null` olur.

Çıkış sırası 20, 10, 15'tir ve giriş sırasıyla aynıdır. FIFO kuralı tam olarak bunu söyler.





Şekil 8.2: Örnek 8.1'in dört anı (yukarıdan aşağı). (a) Boş kuyruk: `on = arka = null`. Öğeler sağdaki "Giriş" ucundan girer, soldaki "Çıkış" ucundan çıkar. (b) Üç `enqueue`'dan sonra önden arkaya 20, 10, 15; `on` 20'yi, `arka` 15'i gösterir. (c) İlk `dequeue()` 20'yi döndürmüş, `on` 10'a ilerlemiştir. (d) Son öğe de çıkınca kuyruk boşalır ve iki referans da `null` olur.

Örnekte iki uç durum dikkat çeker. Birincisi boş kuyruğa ilk eklemedir: bu öğe hem ön hem arka olduğu için iki referansın da güncellenmesi gerekir. İkincisi son öğenin çıkarılmasıdır: kuyruk boşalınca yalnız `on` değil, `arka` da `null` yapılmalıdır. Gerçeklemelerdeki hataların büyük çoğunluğu bu iki uç durumdan çıkar.

Kuyruk sorularını çözerken pratik bir yöntem şudur: her işlemden sonra kuyruğun içeriği *önden arkaya* tek satıra yazılır, işlemin döndürdüğü değer de yanına not edilir. Sıra hep aynı yönde yazılırsa yanlış uçtan çıkarma hatası pek yapılmaz.

8.1.5 Java'da kuyruk: java.util.Queue

Java kütüphanesi kuyruk için `java.util.Queue<E>` arayüzünü sunar. Bu arayüzde her işlemin iki biçimi vardır. Birinci biçim başarısızlıkta istisna fırlatır, ikinci biçim özel bir değer döndürür. Tablo 8.2 iki grubu yan yana gösterir.

İşlem	Hata fırlatır	Özel değer döner
Ekleme	`add(e)`	`offer(e)` → `false`
Çıkarma	`remove()`	`poll()` → `null`
İnceleme	`element()`	`peek()` → `null`

Tablo 8.2: `java.util.Queue` metotları.

Boş bir kuyrukta `remove()` ve `element()` `NoSuchElementException` fırlatır; `poll()` ve `peek()` ise `null` döndürür. Kapasitesi sınırlı bir kuyruk dolduğunda `add(e)` `IllegalStateException` fırlatır, `offer(e)` ise `false` döndürür. Sınırsız kuyruklarda (aşağıdaki iki sınıf da sınırsızdır) `offer` her zaman `true` döndürür.

`Queue` bir arayüz olduğu için doğrudan nesnesi oluşturulamaz. Onu uygulayan iki yaygın sınıf vardır:

- `ArrayDeque`: dairesel dizi ile gerçeklenir ve genellikle en hızlı seçenektir. Bu notta anlatılan dairesel kuyruğun büyüyebilen bir biçimidir.
- `LinkedList`: bağlı liste ile gerçeklenir. Liste işlemlerini de destekler, ama her öğe için ayrı bir düğüm nesnesi oluşturur.

Aşağıdaki kod `ArrayDeque` ile oluşturulmuş bir kuyruk üzerinde temel işlemleri gösterir. Değişkenin türü `Queue<Integer>`'dır; böylece kod yalnız kuyruk işlemlerini kullanabilir ve ileride uygulayan sınıf kolayca değiştirilebilir.

```
Queue<Integer> q = new ArrayDeque<>();
q.offer(20);
q.offer(10);
q.offer(15);        // [20, 10, 15]
q.peek();           // 20, çıkarmaz
q.poll();           // 20 → [10, 15]
q.poll();           // 10 → [15]
q.size();           // 1
q.poll();           // 15 → []
q.poll();           // null (boş)
```

Çağrı	Döner	Kuyruk (sonra)
`offer(20)`	`true`	[20]
`offer(10)`	`true`	[20, 10]
`offer(15)`	`true`	[20, 10, 15]
`peek()`	20	[20, 10, 15]

Çağrı	Döner	Kuyruk (sonra)
`poll()`	20	[10, 15]
`poll()`	10	[15]
`size()`	1	[15]
`poll()`	15	[]
`poll()`	`null`	[]

Tablo 8.3: `Queue` örneğinin izlemesi. Kuyruk önden arkaya yazılmıştır.

İzlemede iki nokta önemlidir. `peek()` 20'yi döndürür ama kuyruğu değiştirmez; hemen ardından gelen `poll()` yine 20'yi verir. Son `poll()` boş kuyrukta çağrılmıştır. Bu çağrı istisna fırlatmaz, `null` döndürür. Aynı noktada `remove()` çağrılsaydı program `NoSuchElementException` ile dururdu.

Adlandırma konusunda da bir not düşmek gerekir. Kendi yazdığımız sınıflarda işlem adları ADT'deki gibi `enqueue` / `dequeue` (ya da Türkçe `ekle` / `cikar`) olur. Java kütüphanesinde ise aynı işlemler `offer` / `poll` / `peek` üçlüsüyle yapılır. Ad farklıdır, davranış aynıdır.

> **Sık yapılan hata**
>
> `peek()` 'in de boş kuyrukta istisna fırlatacağını sanmak. `peek()` özel değer döndüren gruptadır ve boş kuyrukta `null` verir. İstisna fırlatan karşılığı `element()` 'tir. Kural kısaca şöyle hatırlanabilir: `add`, `remove`, `element` hata fırlatır; `offer`, `poll`, `peek` `false` ya da `null` döndürür.

8.1.6 Kuyrukların kullanıldığı yerler

Kuyruk, "işler geldiği sırayla yapılsın" istenen her yerde doğal çözümdür. Dört yaygın örnek şunlardır:

- **İşletim sistemi çizelgeleyicisi.** Çalışmaya hazır süreçler bir *hazır kuyruğunda* bekler. İşlemci (CPU) sırayla bu süreçlere dağıtılır. Bu fikrin en sade biçimi olan round robin çizelgeleme 8.5.1'de ayrıntılı işlenir.
- **Yazdırma kuyruğu.** Bir yazıcıya aynı anda birden çok belge gönderilebilir. Yazıcı belgeleri geliş sırasıyla basar; sonradan gelen belge bekler.
- **Tamponlar** (*buffer*). Klavye girdisi, ağ paketleri ve video akışı bir tamponda bekletilir. Veriyi üreten taraf ile kullanan taraf farklı hızlarda çalışabilir. Tampon bu hız farkını dengeler: üretici hızlıyken veri birikir, tüketici hızlıyken birikmiş veri eritilir.
- **Genişlik öncelikli arama** (*breadth-first search*, BFS). Bir çizgede ya da ağaçta keşfedilen düğümler kuyrukta sıraya alınır. Böylece başlangıca yakın düğümler uzak olanlardan önce ziyaret edilir ve gezinme düzey düzey ilerler. 8.5.4'teki ikilik sayı üretimi bu fikrin küçük bir örneğidir.



Şekil 8.3: Üretici–tüketici düzeni. Üretici (klavye, ağ, süreç) yeni işleri kuyruğun arkasına ekler (`enqueue`). Tüketici (işlemci, yazıcı) öndeki işi alır (`dequeue`). Aradaki tampon kuyruk iki tarafın hız farkını karşılar; İş 1 en eskidir ve ilk işlenir.

Bu örneklerin ortak noktası adalet ve sıranın korunmasıdır. Yazıcı, önce gönderilen belgeyi önce basmalıdır. Ağ paketleri, gönderildikleri sırayla işlenmezse bir video karıştırılmış kareler gösterir. Çizelgeleyici, bir süreci sürekli bekletmemelidir. Kuyruk bu gereksinimleri yapısı gereği karşılar: bir öğe kuyruğa girdikten sonra, ondan önce girmiş öğeler çıkınca sıra mutlaka ona gelir.

8.2 Dizi ile kuyruk

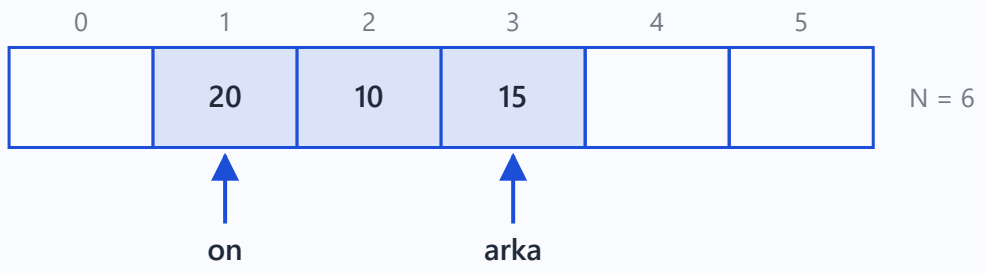
Kuyruğu gerçeklemenin ilk akla gelen yolu öğeleri bir dizide tutmaktır. Dizi bellekte ardışık hücrelerden oluşur ve her hücreye indisiyle *O(1)* zamanda erişilir. Kuyruk için gereken tek şey iki ucun yerini bilmektir. Bu kesim önce en basit fikri dener, onun neden yer israf ettiğini gösterir ve ardından sorunu dairesel kuyrukla çözer.

8.2.1 Dizi temsili

Öğeler bir dizide tutulur ve iki indis kuyruğun uçlarını gösterir:

- `on` (ön, *front*): ilk öğenin, yani bir sonraki çıkarmada alınacak öğenin indisi.
- `arka` (*rear*): son öğenin, yani en son eklenen öğenin indisi.

Eklemede `arka` bir artar ve yeni öğe `dizi[arka]` 'ya yazılır. Çıkarmada `dizi[on]` alınır ve `on` bir artar. Öğeler yerinden kaydırılmaz; yalnız iki indis ilerler. Bu yüzden iki işlem de sabit sayıda adım atar ve *O(1)*'dir. Yöntem basit ve hızlıdır, ama dizinin kapasitesi sabittir.



Şekil 8.4: Kapasitesi *N = 6* olan dizide kuyruk. Öğeler 20, 10, 15 sırasıyla 1, 2 ve 3. indislerdedir; `on = 1` , `arka = 3` 'tür. 0. indisteki öğe daha önce çıkarılmıştır. O yuva artık boştur, ama basit dizi kuyruğu onu yeniden kulla-

namaz.

Şekil 8.4 yöntemin zayıf noktasını şimdiden gösterir. Kuyruğun önü çıkarmalarla sağa kayar. Önde boşalan yuvalar geride kalır ve bir daha kullanılmaz. `arka` indisi, kuyruktaki öğe sayısına değil, şimdiye kadar yapılan ekleme sayısına bağlı olarak dizinin sonuna yaklaşır.

8.2.2 Basit dizi kuyruğunun sorunu

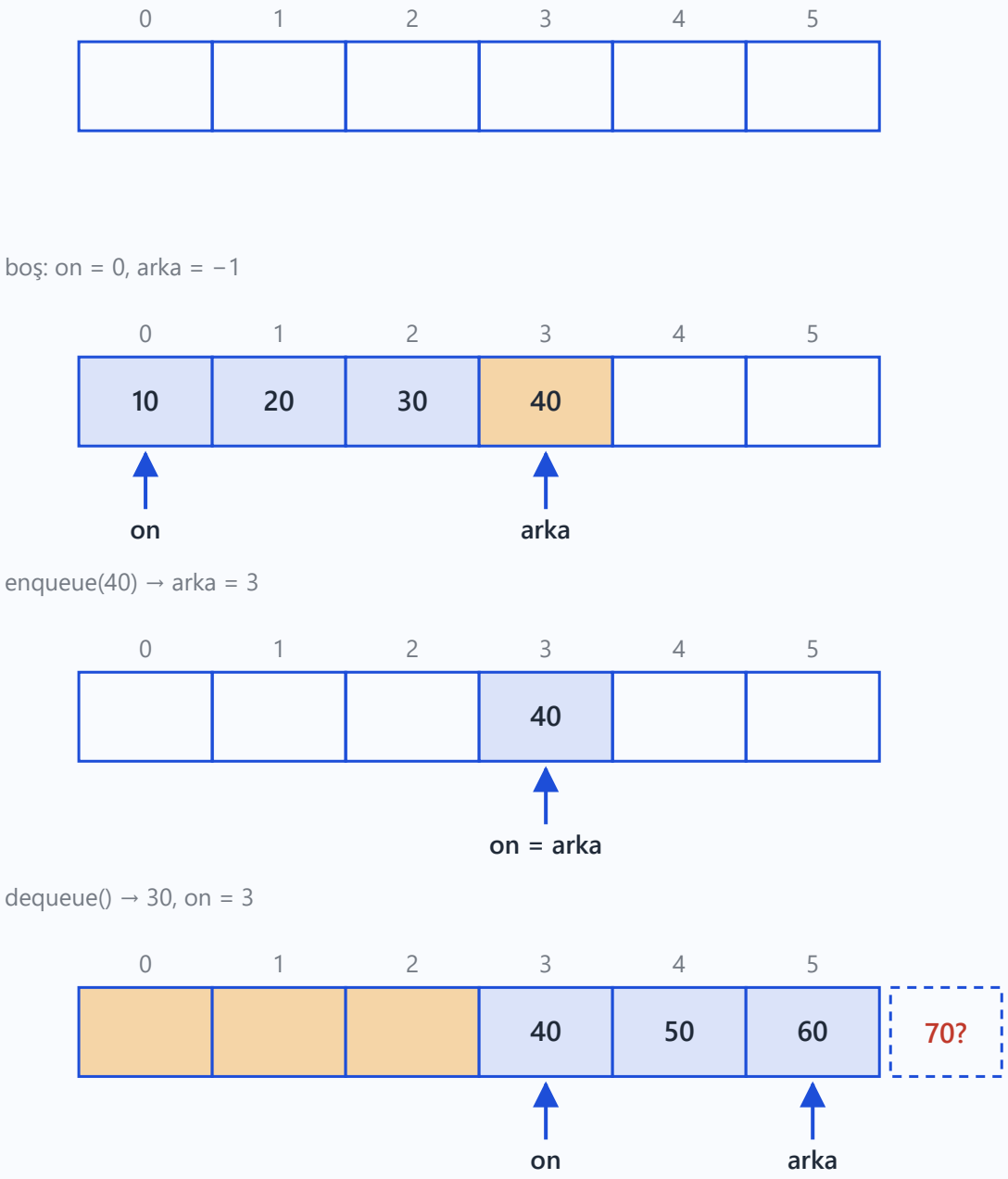
Sorunu somut bir örnekle görelim. Kapasitesi $N = 6$ olan basit dizi kuyruğu `on = 0`, `arka = -1` ile başlıyor. `arka = -1`, henüz hiç öğe eklenmediğini gösterir: ilk eklemede `arka` 0'a çıkacak ve öğe 0. indise yazılacaktır. Kuyruğa sırayla 10, 20, 30, 40 ekleniyor, üç kez çıkarma yapılıyor ve ardından 50, 60, 70 eklenmek isteniyor.

İşlem	`on`	`arka`	Dizi	Döner
(başlangıç)	0	−1	[_, _, _, _, _, _]	—
`enqueue(10)`	0	0	[10, _, _, _, _, _]	—
`enqueue(20)`	0	1	[10, 20, _, _, _, _]	—
`enqueue(30)`	0	2	[10, 20, 30, _, _, _]	—
`enqueue(40)`	0	3	[10, 20, 30, 40, _, _]	—
`dequeue()`	1	3	[_, 20, 30, 40, _, _]	10
`dequeue()`	2	3	[_, _, 30, 40, _, _]	20
`dequeue()`	3	3	[_, _, _, 40, _, _]	30
`enqueue(50)`	3	4	[_, _, _, 40, 50, _]	—
`enqueue(60)`	3	5	[_, _, _, 40, 50, 60]	—
`enqueue(70)`	3	5	[_, _, _, 40, 50, 60]	“dolu”

Tablo 8.4: Basit dizi kuyruğunun izlemesi ($N = 6$). Dizi 0'dan 5'e yazılmıştır; “_” boş yuvadır.

İlk dört eklemede `arka` 0'dan 3'e çıkar. Üç çıkarma 10, 20 ve 30'u döndürür ve `on`'u 3'e taşır. Bu noktada `on` ve `arka` aynı yuvayı, 40'ı gösterir; kuyrukta tek öğe vardır. 50 ve 60 eklenince `arka` 5'e, yani dizinin son indisine ($N - 1$) ulaşır. `enqueue(70)` için `arka + 1 = 6 = N` olur. Dizinin sonunda yer kalmamıştır ve kuyruk “dolu” der.

Oysa kuyrukta yalnız üç öğe vardır: 40, 50 ve 60. 0, 1 ve 2. indisler boştur, ama basit dizi kuyruğu boşalan yuvaları yeniden kullanamaz. Altı yuvalık dizinin yarısı boşa gitmiştir. Daha kötüsü, kuyruk hiç dolmadan, örneğin her eklemeden hemen sonra bir çıkarma yapılsa bile, altı eklemeden sonra `arka` dizinin sonuna dayanır. Basit dizi kuyruğunun ömrü, içindeki öğe sayısıyla değil toplam ekleme sayısıyla sınırlıdır.



Şekil 8.5: Tablo 8.4'ün dört anı. (a) Boş kuyruk: `on = 0`, `arka = -1`. (b) Dört eklemeden sonra `arka = 3`. (c) Üç çıkarmadan sonra `on = arka = 3`; boşalan yuvalar bir daha kullanılmaz. (d) 50 ve 60 eklendikten sonra `arka = 5 = N - 1`; `enqueue(70)` "dolu" der, oysa turuncu üç yuva boştur.

Sorunun iki çözümü vardır.

Çözüm 1: kaydırma. Her çıkarmada kalan öğeler bir yuva sola kaydırılır. Böylece kuyruğun önü hep `dizi[0]`'da kalır ve `on` indisine gerek bile kalmaz. Aşağıdaki metot bu fikri gerçekler; `boyut`, kuyruktaki öğe sayısıdır:

```
public int dequeue() {              // kaydırmalı basit dizi kuyruğu
    if (boyut == 0)
        throw new IllegalStateException("boş");
    int sonuc = dizi[0];            // ön her zaman dizi[0]
    for (int i = 1; i < boyut; i++)
        dizi[i - 1] = dizi[i];      // herkes bir yuva sola
    boyut--;
    return sonuc;
}
```

Döngü `boyut - 1` öğeyi kaydırır. Kuyrukta n öğe varken bir çıkarma $n - 1$ atama yapar; çıkarma $O(n)$ olur. 1000 öğelik bir kuyruğun tamamını boşaltmak $999 + 998 + \ldots + 1 + 0 = 499.500$ atama gerektirir. Yer israfı biter, ama hız gider.

Çözüm 2: dairesel dizi. İndisler dizinin sonuna gelince başa sarar. Öğeler hiç kaydırılmaz, boşalan yuvalar yeniden kullanılır ve her işlem $O(1)$ kalır. Bu yapıya dairesel kuyruk denir.

Sık yapılan hata

Basit dizi kuyruğunda "dolu" koşulunu öğe sayısıyla karıştırmak. `arka == N - 1` koşulu dizinin sonuna gelindiğini söyler, kuyruğun dolu olduğunu söylemez. Tablo 8.4'teki son durumda kuyrukta 3 öğe vardır ve 3 yuva boştur, ama ekleme yine de yapılamaz.

8.2.3 Dairesel kuyruk

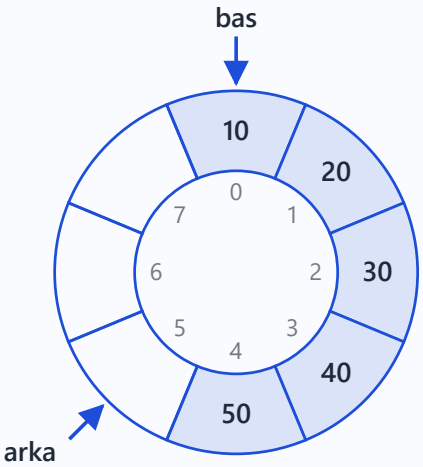
Basit dizi kuyruğunda dizinin sonu bir duvardır: `arka` son indise ulaşınca ilerleyecek yer kalmaz. Dairesel kuyruk bu duvarı kaldırır. Dizinin son yuvasından sonra ilk yuvaya dönülür. Dizinin başı ve sonu mantıksal olarak birleşir ve bir halka oluşturur. Bellekte dizi yine düz bir hücre sırasıdır; halka yalnız indislerin hesaplanma biçiminde vardır.

Tanım 8.2

Dairesel kuyruk (*circular queue*), öğeleri N yuvalı bir dizide tutan ve indislerini $(i + 1) \% N$ biçiminde mod işlemiyle ilerleten kuyruk gerçeklemesidir. Son indisten $(N - 1)$ sonra 0. indise geçilir. Aynı yapı **halka tampon** (*ring buffer*) adıyla da anılır.

Dairesel kuyruk normal bir kuyruk gibi FIFO düzeninde çalışır. Fark yalnız indislerin ilerleme biçimindedir. Basit dizide `i + 1` olan "bir sonraki indis", dairesel dizide `(i + 1) % N` olur. $i < N - 1$ iken iki ifade aynı sonucu verir. $i = N - 1$ olduğunda ise $(N - 1 + 1) \% N = N \% N = 0$ çıkar ve indis başa sarar. Boşalan yuvalar böylece yeniden kullanılır ve öğe kaydırmaya gerek kalmaz.

Bu notta dairesel kuyruk iki alanla tutulur: `bas` (ilk öğenin indisi) ve `boyut` (öğe sayısı). Arka uç ayrı bir alan olarak saklanmaz. Bir sonraki öğenin yazılacağı boş yuva gerektiğinde `(bas + boyut) % N` ile hesaplanır.



Şekil 8.6: Kapasitesi 8 olan dairesel kuyruğun halka görünümü. 0–4. indislerde 10, 20, 30, 40, 50 vardır. `bas` 0. indisi gösterir. `arka` burada bir sonraki öğenin yazılacağı boş yuvadır: *(bas + boyut) % N = (0 + 5) % 8 = 5*. 7. yuvadan sonra 0. yuvaya dönülür.

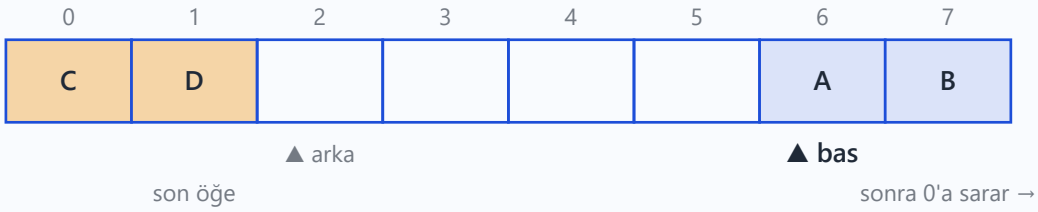
Halka görünümü dizinin "sonu" diye bir şey olmadığını açıkça gösterir. 7. yuvanın komşusu hem 6. hem 0. yuvadır. Kuyruk saat yönünde büyür ve küçülür: ekleme arkada saat yönünde ilerler, çıkarma da önü saat yönünde ilerletir. Kuyruk halkanın herhangi bir yerinde başlayıp herhangi bir yerinde bitebilir.

8.2.4 İndislerin mod ile hesaplanması

Dairesel kuyrukta bütün indisler `bas`, `boyut` ve *N*'den hesaplanır. Mod işlemi (`%`), bir sayının *N*'ye bölümünden kalanı verir. Negatif olmayan her *a* için *a % N* sonucu 0 ile *N − 1* arasındadır; yani her zaman geçerli bir dizi indisidir. Tablo 8.5 dört temel formülü *N = 8*, `bas = 6`, `boyut = 4` için hesaplar.

Ne?	Formül	Örnek
Sonraki boş yuva (ekleme yeri)	`(bas + boyut) % N`	(6 + 4) % 8 = 2
Son öğe	`(bas + boyut − 1) % N`	(6 + 3) % 8 = 1
Baştan k. öğe (k = 0, 1, ...)	`(bas + k) % N`	k = 2 → (6 + 2) % 8 = 0
Çıkarmadan sonra `bas`	`(bas + 1) % N`	(6 + 1) % 8 = 7

Tablo 8.5: Dairesel kuyrukta indis formülleri (örnek: *N = 8*, `bas = 6`, `boyut = 4`).



Şekil 8.7: Tablo 8.5'teki örneğin dizi görünümü. Önden arkaya sıra A (6), B (7), C (0), D (1)'dir. `bas = 6`, son öğe 1. indiste, sonraki boş yuva 2. indistedir. 7. indisten sonra 0'a sarılır.

Formüllerin hepsi aynı düşünceden çıkar: kuyruğun *k*. öğesi, baştan *k* adım ötededir. Düz dizide bu yer *bas + k* olurdu. Dairesel dizide *bas + k* dizinin sınırını aşabilir; `% N` taşan kısmı başa sarar. Örnekte C kuyruğun 2. öğesidir (k = 2). *6 + 2 = 8* dizinin dışındadır, ama *8 % 8 = 0* C'nin gerçek yerini verir.

Sonraki boş yuva formülü de bu kuralın özel bir durumudur: kuyrukta `boyut` öğe varsa 0'dan `boyut - 1`'e kadar indisler dolu, `boyut`. konum ise boştur. Son öğe `boyut - 1`. konumdadır. Mod işlemi dizinin sonundan başına geçişi tek bir ifadeyle çözer; `if` ile "sona geldiysem başa dön" denetimi yazmaya gerek kalmaz.

Formüllerde toplananlar negatif değildir ve *bas* < *N*, *boyut* ≤ *N* olduğu için toplam *2N*'yi geçmez. Bu yüzden `%` en fazla bir tur geri sarar. Bir indisi *geri* almak gerektiğinde ise dikkat gerekir. Örneğin önün bir öncesindeki yuva `(bas - 1) % N` ile hesaplanırsa, `bas = 0` için Java *−1 % N = −1* verir. Java'da `%` işleminin sonucu bölünenin işaretini taşır. Doğru ifade `(bas - 1 + N) % N`'dir. Çift uçlu kuyruğun diziyle gerçeklenmesinde başa ekleme tam bu ifadeyi kullanır.

> **Sık yapılan hata**
>
> Mod işlemini unutmak. Ekleme yeri `bas + boyut` diye yazılırsa, kuyruk dizinin sonuna sarmış olduğunda indis *N*'yi aşar ve `ArrayIndexOutOfBoundsException` fırlatılır. Örnekte *6 + 4 = 10*, 8 yuvalı dizide geçerli bir indis değildir.

8.2.5 Dolu mu, boş mu?

Dairesel kuyruğu yalnız iki indisle tutmak ilk bakışta yeterli görünür: `bas` ilk öğeyi, `arka` bir sonraki yazma yerini göstersin. Ancak bu temsilde boş ve dolu kuyruk ayırt edilemez. Boş kuyrukta `bas` ile `arka` aynı yuvayı gösterir. Kuyruğa *N* öğe eklendiğinde ise `arka` halkayı tam bir tur dolaşıp yine `bas`'a gelir. İki durumda da `bas == arka`'dır.



Şekil 8.8: *N = 4* için üç durum. Solda boş kuyruk: `bas == arka`. Ortada dolu kuyruk (önden arkaya B, C, D, E): yine `bas == arka`. Sağda "bir yuva boş" yöntemi: B, C, D varken 0. yuva boş bırakılır ve dolu koşulu *(arka + 1) % N == bas* olur.

Bu belirsizliği gidermenin iki yaygın yolu vardır.

1. Öğe sayacı tutmak. İndislerin yanında bir `boyut` alanı tutulur. Boş koşulu `boyut == 0`, dolu koşulu `boyut == N`'dir. Dizinin *N* yuvasının tamamı kullanılır. Bu notta (ve sunumda) kullanılan yöntem budur. Sayaç zaten tutulduğu için arka indisini ayrıca saklamaya da gerek kalmaz; `(bas + boyut) % N` ile hesaplanır.

2. Bir yuvayı boş bırakmak. Kuyruk hiçbir zaman *N* öğeye ulaşmaz; en fazla *N − 1* öğe saklanır. Boş koşulu `bas == arka`, dolu koşulu `(arka + 1) % N == bas`'tır. Dolu kuyrukta `arka`, `bas`'ın hemen bir gerisinde kalır ve aradaki tek yuva boştur. Sayaç gerekmez, ama bir yuva hep boşa gider.

Örnek 8.2

"Bir yuva boş" yöntemiyle *N = 7* yuvalı bir dairesel kuyruk `bas = arka = 3` ile başlıyor. Kuyruğa en fazla kaç öğe eklenebilir ve dolduğunda `arka` kaçtır?

1. **Başlangıç:** `bas == arka == 3`, kuyruk boştur.
2. **Eklemeler:** Her ekleme `arka`'yı bir ilerletir: 3 → 4 → 5 → 6 → 0 → 1 → 2. Dördüncü eklemede `arka` 6'dan 0'a sarar.
3. **Dolu denetimi:** Altı eklemeden sonra `arka = 2` olur ve *(2 + 1) % 7 = 3 = bas*'tır. Kuyruk doludur; yedinci ekleme reddedilir.

Kuyruk *N − 1 = 6* öğe saklar ve dolduğunda `arka = 2`'dir. Yedinci öğe kabul edilseydi `arka` 3'e, yani `bas`'a gelirdi ve dolu kuyruk boş kuyruktan ayırt edilemezdi.

İki yöntem arasındaki fark küçüktür. Sayaç yöntemi bir tam sayı alanı fazla tutar, ama dizinin bütün yuvalarını kullanır ve kodu daha okunur kılar: `boyut == 0` ve `boyut == N` ifadeleri ne denetlendiğini doğrudan söyler. Bir yuva boş yöntemi ise sayacın güncellenmesinin pahalı ya da sakıncalı olduğu yerlerde tercih edilir. Örneğin bir üretici ve bir tüketici aynı tamponu eş zamanlı kullanıyorsa, üretici yalnız `arka`'yı, tüketici yalnız `bas`'ı değiştirir. Ortak bir sayaç olmadığı için iki tarafın aynı alanı güncellemesi gerekmez.

Sık yapılan hata

Yalnız `bas` ve `arka` tutup N öğenin tamamını saklamaya çalışmak. Bu durumda `bas == arka` koşulu hem boş hem dolu kuyrukta doğrudur. Kod iki durumdan birini yanlış yorumlar: ya dolu kuyruğa yazmaya devam edip en eski öğenin üzerine yazar, ya da dolu kuyruğu boş sanıp hiç öğe vermez. Çözüm ya bir sayaç ya da bir boş yuvadır.

8.2.6 Dairesel kuyruk adım adım

Sayaç yöntemiyle çalışan dairesel kuyruğu, kapasitesi $N = 5$ olan bir dizi üzerinde izleyelim. Başlangıçta `bas = 0`, `boyut = 0`'dır. Ekleme `ekle`, çıkarma `cikar` adıyla yapılır. Her eklemede yeni öğe `(bas + boyut) % 5` yuvasına yazılır ve `boyut` bir artar. Her çıkarmada `dizi[bas]` döndürülür, yuva boşaltılır, `bas` `(bas + 1) % 5` olur ve `boyut` bir azalır.

İşlem	Yazılan / okunan yuva	bas	boyut	Dizi	Döner
(başlangıç)	—	0	0	[_, _, _, _, _]	—
`ekle(10)`	(0 + 0) % 5 = 0	0	1	[10, _, _, _, _]	—
`ekle(20)`	(0 + 1) % 5 = 1	0	2	[10, 20, _, _, _]	—
`ekle(30)`	(0 + 2) % 5 = 2	0	3	[10, 20, 30, _, _]	—
`cikar()`	0	1	2	[_, 20, 30, _, _]	10
`cikar()`	1	2	1	[_, _, 30, _, _]	20
`ekle(40)`	(2 + 1) % 5 = 3	2	2	[_, _, 30, 40, _]	—
`ekle(50)`	(2 + 2) % 5 = 4	2	3	[_, _, 30, 40, 50]	—
`ekle(60)`	(2 + 3) % 5 = 0	2	4	[60, _, 30, 40, 50]	—
`ekle(70)`	(2 + 4) % 5 = 1	2	5	[60, 70, 30, 40, 50]	—
`cikar()`	2	3	4	[60, 70, _, 40, 50]	30
`cikar()`	3	4	3	[60, 70, _, _, 50]	40
`cikar()`	4	0	2	[60, 70, _, _, _]	50
`cikar()`	0	1	1	[_, 70, _, _, _]	60
`cikar()`	1	2	0	[_, _, _, _, _]	70

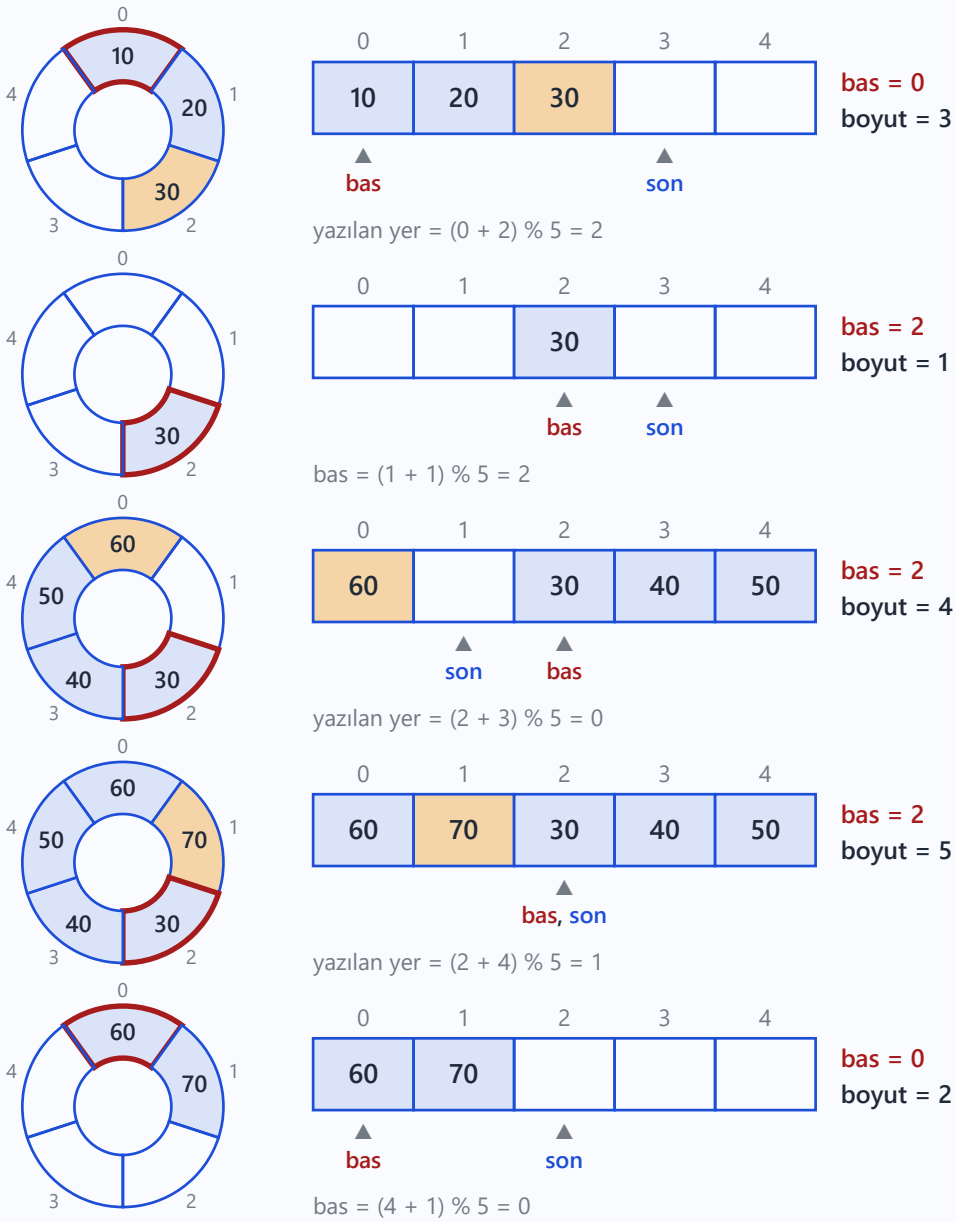
Tablo 8.6: $N = 5$ dairesel kuyruğun izlemesi. Dizi 0'dan 4'e yazılmıştır; "_" boş yuvadır.

İzlemenin ilk beş satırı basit dizi kuyruğundakinden farksızdır. Üç ekleme 0, 1 ve 2. yuvaları doldurur. İki çıkarma 10 ve 20'yi döndürür ve `bas`'ı 2'ye taşır. Öğeler yerinden oynamaz; yalnız `bas` ilerler.

Asıl fark `ekle(60)` satırında görülür. Ekleme yeri *(2 + 3) % 5 = 0*'dır: dizinin sonu aşılmış ve indis başa sarmıştır. Basit dizi kuyruğu bu noktada "dolu" derdi. Dairesel kuyruk ise boşalmış 0. yuvayı kullanır. `ekle(70)` 1. yuvaya yazılır ve `boyut` 5'e ulaşır. Kuyruk artık gerçekten doludur: beş yuvanın beşi de doludur ve hiçbiri boşa gitmemiştir.

Dolu kuyrukta bir sonraki yazma yeri *(2 + 5) % 5 = 2*'dir, yani `bas`'ın kendisi. Yalnız indislere bakılsaydı kuyruk boş sanılabilirdi. `boyut == 5` bilgisi bu karışıklığı önler.

Ardından gelen beş çıkarma önden arkaya 30, 40, 50, 60 ve 70'i döndürür. Üçüncü çıkarmada `bas` *(4 + 1) % 5 = 0* olur; bu kez ön de başa sarmıştır. Son çıkarmadan sonra `boyut = 0`'dır ve kuyruk yine boştur. Çıkış sırası 30, 40, 50, 60, 70'tir. Bu, öğelerin giriş sırasıyla aynıdır: dairesel düzen FIFO kuralını bozmaz.



Şekil 8.9: Tablo 8.6'nın beş anı; her satırda solda halka, sağda dizi görünümü vardır. Halkada kırmızı kenarlı dilim `bas`'tır; dizide ▲ `son` bir sonraki yazma yerini gösterir. (a) Üç eklemeden sonra. (b) İki çıkarmadan sonra `bas = 2`. (c) `ekle(60)` 0. yuvaya yazılmış, indis başa sarmıştır. (d) `ekle(70)` ile kuyruk dolmuş, `bas` ile `son` aynı yuvaya gelmiştir. (e) Üç çıkarmadan sonra `bas` da başa sarmıştır: *(4 + 1) % 5 = 0*.

8.2.7 Kod: DaireselKuyruk sınıfı

Aşağıdaki sınıf sayaç yöntemiyle çalışan dairesel kuyruğun alanlarını, yapıcısını ve durum sorgularını içerir. Ekleme ve çıkarma metotları 8.2.8'dedir.

```
public class DaireselKuyruk<E> {
    private E[] dizi;       // öğeleri saklar
    private int bas = 0;    // ilk öğenin indisi (front)
    private int boyut = 0;  // öğe sayısı (size)

    @SuppressWarnings("unchecked")
    public DaireselKuyruk(int kapasite) {
        dizi = (E[]) new Object[kapasite];
    }
    public int boyut()      { return boyut; }
    public boolean bosMu()  { return boyut == 0; }
    public boolean doluMu() { return boyut == dizi.length; }
    public E ilk()          { return bosMu() ? null : dizi[bas]; }
    // ekle(), cikar()
}
```

Kod satır satır şöyle çalışır:

- `dizi` öğeleri tutar. `bas` ilk öğenin indisi, `boyut` öğe sayısıdır. İkisi de 0 ile başlar, çünkü yeni kuyruk boştur.
- Yapıcı istenen kapasitede bir dizi ayırır. Java'da `new E[kapasite]` yazılamaz: tür parametresi `E` çalışma zamanında bilinmediği için jenerik dizi doğrudan oluşturulamaz. Bunun yerine bir `Object` dizisi oluşturulur ve `(E[])` ile dönüştürülür. Derleyici bu dönüşüm için denetlenmemiş dönüşüm uyarısı verir. `@SuppressWarnings("unchecked")` bu uyarıyı yalnız yapıcı için susturur. Dizi sınıfın dışına hiç verilmediği sürece dönüşüm güvenlidir.
- `boyut()` öğe sayısını döndürür. `bosMu()` yalnız `boyut == 0` sorusunu sorar. `doluMu()` ise `boyut == dizi.length` sorusunu sorar; dizinin uzunluğu kapasitedir.
- `ilk()` kuyruk boşsa `null`, değilse `dizi[bas]`'ı döndürür ve kuyruğu değiştirmez. Burada `? :` koşul işlecidir: koşul doğruysa ilk, yanlışsa ikinci değeri verir.

Arka indisi ayrıca saklanmaz; gerektiğinde `(bas + boyut) % dizi.length` ile hesaplanır. Bunun bir yararı vardır. Hem `arka` hem `boyut` saklansaydı, her işlemde ikisinin birlikte ve doğru güncellenmesi gerekirdi. Bir metotta birini güncelleyip ötekini unutmak, iki alanın birbirine uymadığı tutarsız bir duruma yol açardı. Aynı bilgiyi tek yerde tutmak bu riski ortadan kaldırır.

Bütün metotlar döngüsüzdür ve sabit sayıda adım atar; her biri *O(1)*'dir. Sınıf, kapasite kadar yuvalı bir dizi kullanır. Alan, kuyrukta kaç öğe olduğundan bağımsız olarak *O(N)*'dir.

8.2.8 Kod: ekle() ve cikar()

Sınıfın iki temel metodu şunlardır:

```
public void ekle(E oge) {           // enqueue
    if (doluMu())
        throw new IllegalStateException("dolu");
    int son = (bas + boyut) % dizi.length;
    dizi[son] = oge;
    boyut++;
}
```

```
public E cikar() {                  // dequeue
    if (bosMu())
        throw new IllegalStateException("boş");
    E oge = dizi[bas];
    dizi[bas] = null;               // referansı bırak
    bas = (bas + 1) % dizi.length;
    boyut--;
    return oge;
}
```

`ekle` önce doluluğu sorar. Kuyruk doluysa `IllegalStateException("dolu")` fırlatılır; bu, taşmanın kodda bildirilme biçimidir. Yer varsa ekleme yeri `son` değişkenine hesaplanır, öğe oraya yazılır ve `boyut` bir artar. `bas` değişmez: eklemek kuyruğun önünü etkilemez.

`cikar` önce boşluğu sorar. Kuyruk boşsa `IllegalStateException("boş")` fırlatılır; bu da alt taşmadır. Değilse öndeki öğe `oge`'ye alınır. Ardından üç güncelleme yapılır: yuvaya `null` yazılır, `bas` bir ilerler, `boyut` bir azalır. Son olarak saklanan öğe döndürülür.

`dizi[bas] = null;` satırı kuyruğun doğruluğu için gerekli değildir. Bu satır olmasa da `cikar` doğru öğeyi döndürür; o yuva bir sonraki sarmada zaten üzerine yazılır. Satırın amacı bellekle ilgilidir. Java'da belleği **çöp toplayıcı** (*garbage collector*) geri alır ve bunu yalnız hiçbir referansın göstermediği nesneler için yapar. Kuyruktan çıkmış ama dizide hâlâ duran bir nesne temizlenemez. Büyük nesneler (görüntüler, dosya tamponları) taşıyan bir kuyrukta bu fark bellekte belirgin bir sızıntıya dönüşebilir.

Satırların sırası da önemlidir. `oge = dizi[bas]` satırı, `null` atamasından *önce* gelmelidir; yoksa döndürülecek değer silinmiş olur. `bas` da `null` atamasından *sonra* ilerlemelidir; yoksa yanlış yuva boşaltılır.

Örnek 8.3

İki formülü sınır durumlarında hesaplayalım.

1. **Ekleme yeri:** $N = 5$, `bas = 3`, `boyut = 2` olsun. Öğeler 3. ve 4. yuvalardadır. `son = (3 + 2) % 5 = 5 % 5 = 0`. Yeni öğe 0. yuvaya yazılır: dizinin sonu aşılmış ve başa sarılmıştır.
2. **Önü ilerletme:** $N = 5$, `bas = 4` olsun. Öndeki öğe dizinin son yuvasındadır. Çıkarmadan sonra `bas = (4 + 1) % 5 = 0` olur; ön de başa sarar.

İki hesap da tek bir toplama ve bir mod işlemidir. Döngü ya da kaydırma yoktur; her iki metot da *O(1)*'dir.

8.2.9 Kod izleme: ekle()

`ekle` metodunu, dizinin sonundan başa sarmayı gerektiren bir durumda izleyelim. Başlangıçta `dizi.length = 5`, `bas = 2`, `boyut = 2`'dir; kuyrukta önden arkaya 30 (2. yuva) ve 40 (3. yuva) vardır. Kuyruğa sırayla 50, 60, 70 ve 80 eklenecektir. İzlemede kullanılan metot şudur:

```
public void ekle(E oge) {
  if (doluMu())
    throw new IllegalStateException("dolu");
  int son = (bas + boyut) % dizi.length;
  dizi[son] = oge;
  boyut++;
}
```

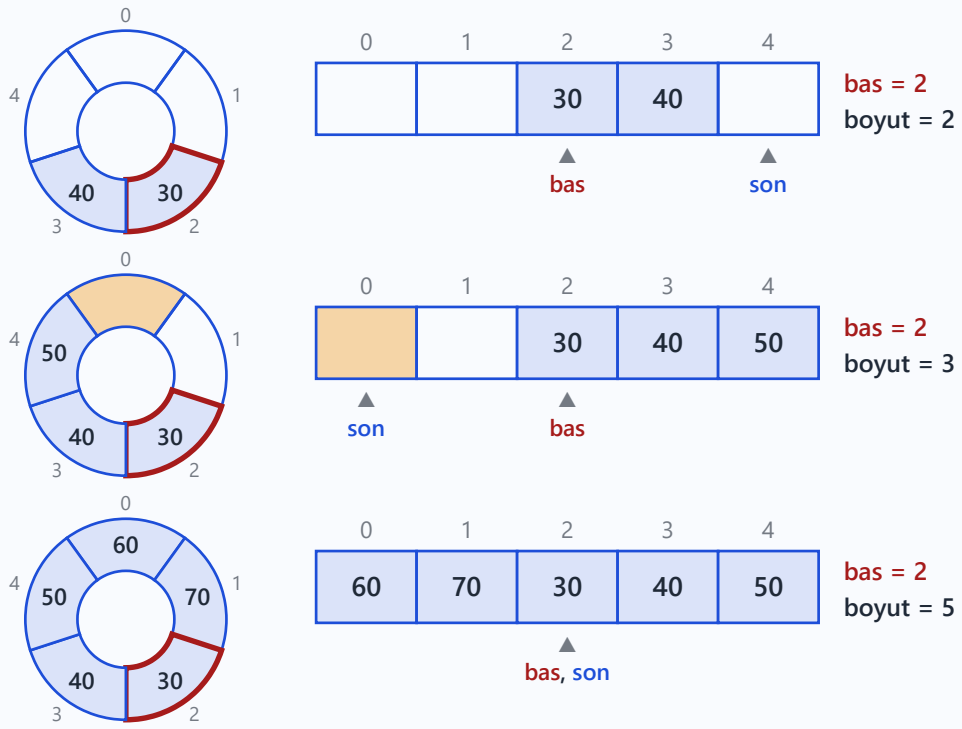
Çağrı	`doluMu()`	`son`	`boyut` (sonra)	Dizi (sonra)
(başlangıç)	—	—	2	[_, _, 30, 40, _]
`ekle(50)`	2 == 5? hayır	(2 + 2) % 5 = 4	3	[_, _, 30, 40, 50]
`ekle(60)`	3 == 5? hayır	(2 + 3) % 5 = 0	4	[60, _, 30, 40, 50]
`ekle(70)`	4 == 5? hayır	(2 + 4) % 5 = 1	5	[60, 70, 30, 40, 50]
`ekle(80)`	5 == 5? **evet**	—	5	değişmez; istisna: “dolu”

Tablo 8.7: `ekle` çağrılarının izlemesi (`dizi.length = 5`, `bas = 2` hep sabit).

`ekle(50)`'de denetim `boyut`'u (2) kapasiteyle (5) karşılaştırır; yer vardır. Ekleme yeri *(2 + 2) % 5 = 4*'tür. Bu hesapta mod işlemi bir şey değiştirmez, çünkü 4 zaten dizinin içindedir. `dizi[4] = 50` yazılır ve `boyut` 3 olur. `bas` değişmez; ön hâlâ `dizi[2]`'dir.

`ekle(60)`'ta ekleme yeri *(2 + 3) % 5 = 5 % 5 = 0*'dır. Dizinin sonu aşılmış ve indis başa sarmıştır. 60, 0. yuvaya yazılır. Mantıksal olarak 60 kuyruğun en arkasındadır, fiziksel olarak ise dizinin en başında durur. `ekle(70)` aynı biçimde *(2 + 4) % 5 = 1*'e yazılır ve `boyut` 5'e çıkar.

`ekle(80)`'de denetim `5 == 5` sonucunu verir. Kuyruk doludur ve `IllegalStateException("dolu")` fırlatılır. Dizi ve alanlar değişmeden kalır. Kapasite tam 5 öğede tükenmiştir; hiçbir yuva boşa gitmemiştir.



Şekil 8.10: `ekle` izlemesinin üç anı. (a) Başlangıç: 30 ve 40, `bas = 2`, `son = 4`. (b) `ekle(60)`: `son = (2 + 3) % 5 = 0`, yazma yeri başa sarmıştır. (c) `ekle(70)`'ten sonra kuyruk doludur; `boyut = 5` ve `son`, `bas` ile aynı yuvaya gelmiştir.

8.2.10 Kod izleme: cikar()

Önceki izlemenin sonundaki dolu kuyruktan devam edelim. `bas = 2`'dir ve kuyruk önden arkaya 30, 40, 50, 60, 70'tir. Üç kez `cikar()` çağrılacaktır.

```
public E cikar() {
  if (bosMu())
    throw new IllegalStateException("boş");
  E oge = dizi[bas];
  dizi[bas] = null;           // referansı bırak
  bas = (bas + 1) % dizi.length;
  boyut--;
  return oge;
}
```

Çağrı	`oge`	Boşaltılan yuva	`bas` (sonra)	`boyut` (sonra)	Dizi (sonra)
(başlangıç)	—	—	2	5	[60, 70, 30, 40, 50]
1. `cikar()`	30	2	(2 + 1) % 5 = 3	4	[60, 70, _, 40, 50]
2. `cikar()`	40	3	(3 + 1) % 5 = 4	3	[60, 70, _, _, 50]

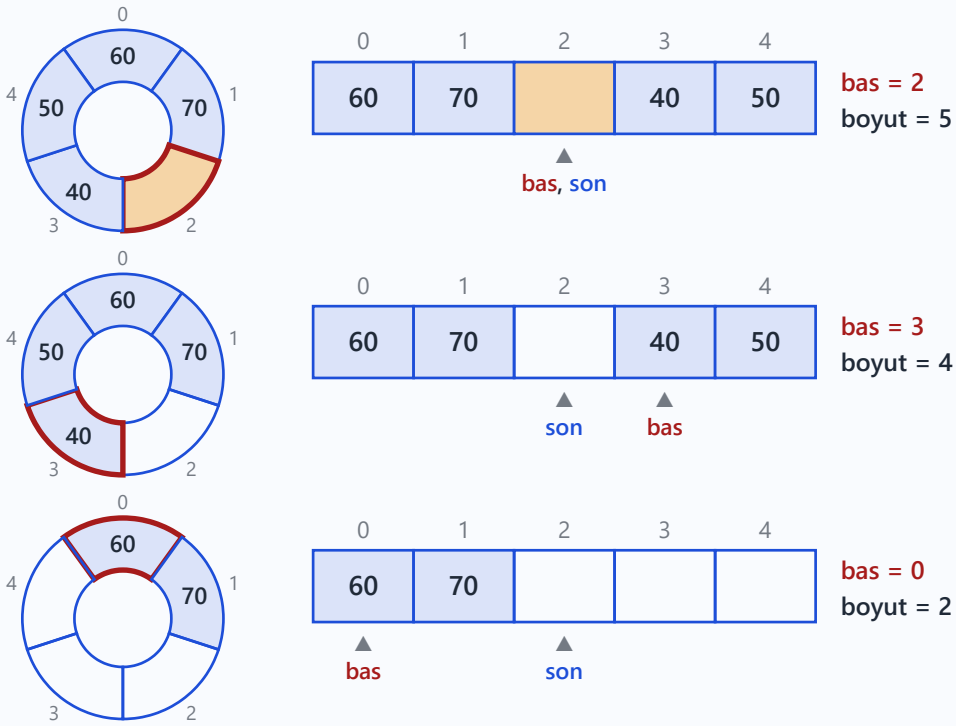
Çağrı	`oge`	Boşaltılan yuva	`bas` (sonra)	`boyut` (sonra)	Dizi (sonra)
3. `cikar()`	50	4	(4 + 1) % 5 = 0	2	[60, 70, _, _, _]

Tablo 8.8: `cikar` çağrılarının izlemesi (`dizi.length = 5`).

Her çağrı aynı beş adımı atar. Önce `bosMu()` sorulur; `boyut` sıfırdan büyük olduğu için devam edilir. Sonra `oge = dizi[bas]` ile öndeki, yani en eski öğe alınır. İlk çağrıda bu `dizi[2] = 30`'dur. Ardından `dizi[2] = null` ile yuva boşaltılır. `bas` *(2 + 1) % 5 = 3*'e ilerler, `boyut` 4'e iner ve 30 döndürülür.

Üçüncü çağrıda öndeki öğe dizinin son yuvasındadır: `dizi[4] = 50`. `bas` *(4 + 1) % 5 = 5 % 5 = 0* olur, yani ön de başa sarar. Kuyrukta 60 (0. yuva) ve 70 (1. yuva) kalır ve ön artık 60'tır. Fiziksel olarak dizinin başında duran bu iki öğe, mantıksal olarak da kuyruğun önündedir.

Çıkarmalarda hiçbir öğe yerinden oynamaz ve hiçbir döngü çalışmaz. Her çağrı sabit sayıda adım atar: *O(1)*.



Şekil 8.11: `cikar` izlemesinin üç anı. (a) İlk çağrıda `oge = 30` alınmış ve `dizi[2] = null` yapılmıştır; `bas` ve `boyut` henüz güncellenmemiştir. (b) İlk çağrının sonu: `bas = 3`, `boyut = 4`, 30 döndürülür. (c) Üç çağrıdan sonra `bas = 0`, `boyut = 2`; kuyrukta 60 ve 70 kalmıştır.

8.2.11 Dolu dairesel kuyruğu büyütme

Sabit kapasiteli kuyruk dolunca hata verir. Kapasitenin önceden bilinmediği durumlarda daha iyi bir yol vardır: dolu kuyruğa ekleme yapılırken iki katı büyüklükte yeni bir dizi ayırıp öğeleri oraya taşımak. Aşağıdaki kod `ekle`'yi bu biçimde değiştirir:

```
private void buyut(int yeniKap) {
  E[] yeni = (E[]) new Object[yeniKap];
  for (int k = 0; k < boyut; k++)
    yeni[k] = dizi[(bas + k) % dizi.length];
  dizi = yeni;
  bas = 0;
}
public void ekle(E oge) {
  if (boyut == dizi.length) buyut(2 * dizi.length);
  dizi[(bas + boyut) % dizi.length] = oge;
  boyut++;
}
```

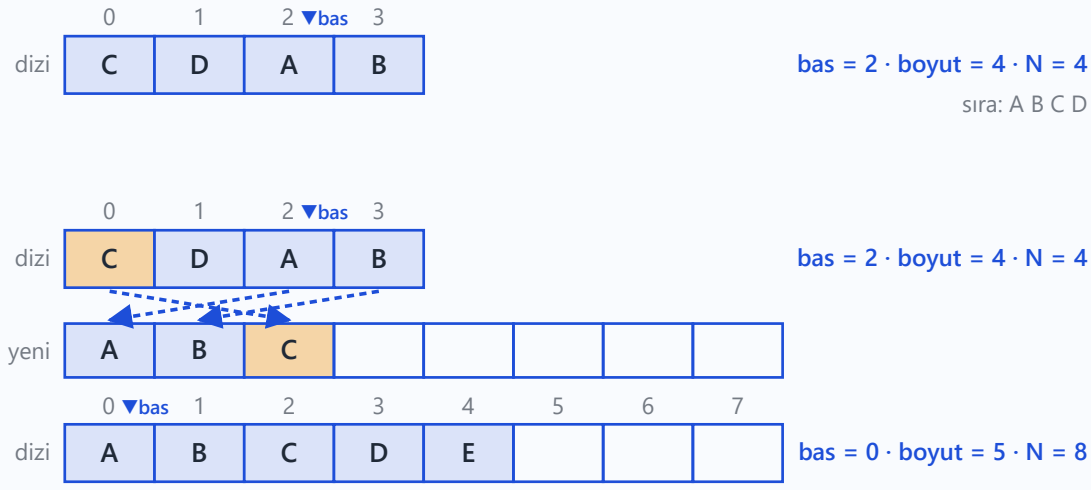
`buyut` metodu, istenen kapasitede yeni bir dizi ayırır. Ardından kuyruktaki öğeleri *mantıksal sırayla* yeni dizinin başına kopyalar: kuyruğun *k*. öğesi eski dizide `(bas + k) % dizi.length` yuvasındadır ve yeni dizinin *k*. yuvasına yazılır. Kopyalama bitince eski dizi bırakılır ve `bas` 0 yapılır, çünkü kuyruğun önü artık yeni dizinin başındadır. `ekle`, dolu kuyrukta önce kapasiteyi ikiye katlar, sonra her zamanki gibi yazar.

Örnek olarak kapasitesi 4 olan dolu bir kuyruk düşünelim. `bas = 2`'dir; kuyruk mantıksal olarak A, B, C, D, dizide ise fiziksel olarak C, D, A, B biçimindedir. Kuyruk daha önce başa sarmıştır. Şimdi `ekle(E)` çağrılıyor.

Adım	Hesap	Yeni dizi (sonra)
Doluluk	`boyut == dizi.length` → 4 == 4, dolu	—
Yeni dizi	`new Object[8]`	[␣, ␣, ␣, ␣, ␣, ␣, ␣, ␣]
`k = 0`	`yeni[0] = dizi[(2 + 0) % 4] = dizi[2]` = A	[A, ␣, ␣, ␣, ␣, ␣, ␣, ␣]
`k = 1`	`yeni[1] = dizi[(2 + 1) % 4] = dizi[3]` = B	[A, B, ␣, ␣, ␣, ␣, ␣, ␣]
`k = 2`	`yeni[2] = dizi[(2 + 2) % 4] = dizi[0]` = C	[A, B, C, ␣, ␣, ␣, ␣, ␣]
`k = 3`	`yeni[3] = dizi[(2 + 3) % 4] = dizi[1]` = D	[A, B, C, D, ␣, ␣, ␣, ␣]
`dizi = yeni; bas = 0`	ön artık 0. yuvada	[A, B, C, D, ␣, ␣, ␣, ␣]
Yazma	`(0 + 4) % 8 = 4` → `dizi[4] = E ; boyut = 5`	[A, B, C, D, E, ␣, ␣, ␣]

Tablo 8.9: `ekle(E)` sırasında `buyut(8)` 'in izlemesi. Eski dizi `[C, D, A, B]`, `bas = 2`.

`k = 2` adımında mod işlemi devreye girer: *(2 + 2) % 4 = 0*. Eski dizinin sonu aşılmış ve kopyalama başa sarmıştır. Bu sayede başa sarmış C ve D, A ile B'nin hemen arkasına, doğru sıraya yerleşir. Büyütmeden sonra kuyruk yeni dizide düzgün bir sıra hâlindedir ve `bas = 0`'dır. E, *(0 + 4) % 8 = 4*. yuvaya yazılır.



Şekil 8.12: Dolu dairesel kuyruğun büyütülmesi. (a) Dolu kuyruk: dizide C, D, A, B; `bas = 2`, mantıksal sıra A, B, C, D. (b) `k = 2` adımı: `dizi[0]`'daki C, yeni dizinin 2. yuvasına kopyalanır. (c) Büyütme ve yazma bittikten sonra yeni dizi A, B, C, D, E'dir; `bas = 0`, `boyut = 5`, $N = 8$.

Sık yapılan hata

Büyütmede `Arrays.copyOf(dizi, 2 * dizi.length)` kullanmak. Bu metot öğeleri *aynı indislere* kopyalar. Kuyruk başa sarmamışsa sonuç doğru olur. Başa sarmış bir kuyrukta ise mantıksal sıra bozulur. Yukarıdaki örnekte `copyOf` yeni diziyi `[C, D, A, B, _, _, _, _]` yapar ve `bas = 2` kalır. Kuyruk A, B'den sonra 4. ve 5. yuvalardaki boşlukları öğe sanar; başa sarmış C ve D ise kuyruktan düşer. Öğeler mutlaka `(bas + k) % N` sırasıyla kopyalanmalı ve `bas` 0 yapılmalıdır.

Büyütme bütün öğeleri kopyaladığı için tek başına $O(n)$ sürer. Buna karşın kapasite her seferinde iki katına çıktığı için büyütme seyrek yapılır. Bu durumda tek tek işlemlere değil, uzun bir işlem dizisinin toplam maliyetine bakmak daha anlamlıdır. Bu tür incelemeye **amortize analiz** (*amortized analysis*) denir.

Teorem 8.1 (İki katına büyütmenin amortize maliyeti)

Dolunca kapasitesi iki katına çıkarılan dairesel kuyrukta, boş kuyruktan başlayan n eklemenin toplam maliyeti (yazma ve kopyalama sayısı) $3n$'den azdır. Dolayısıyla `ekle`'nin amortize maliyeti $O(1)$'dir.

İspat. Başlangıç kapasitesi 1 olsun; daha büyük bir başlangıç kapasitesi kopya sayısını yalnız azaltır. Büyütmeler kuyrukta 1, 2, 4, …, 2^k öğe varken yapılır; burada 2^k, ikinin n'den küçük en büyük kuvvetidir. Toplam kopya sayısı $1 + 2 + 4 + \ldots + 2^k = 2^{k+1} - 1$'dir. $2^k < n$ olduğundan bu toplam $2n - 1$'den, dolayısıyla $2n$'den küçüktür. Her ekleme ayrıca bir yazma yapar: toplam n yazma. Hepsi birlikte $2n + n = 3n$'den azdır. Ekleme başına ortalama maliyet 3'ten küçük, yani sabittir. Araya giren çıkarmalar kuyruktaki öğe sayısını azalttığı için kopya sayısını artıramaz. □

Somut bir sayı verelim. Kapasitesi 4 olan boş kuyruğa hiç çıkarma yapmadan 17 öğe eklensin. Büyütmeler 5. eklemede (4 kopya), 9. eklemede (8 kopya) ve 17. eklemede (16 kopya) yapılır: toplam $4 + 8 + 16 = 28$ kopya. 17 yazmayla birlikte toplam maliyet 45'tir ve $3 \cdot 17 = 51$'in altındadır. Teoremin söylemediği bir şeye de dikkat etmek gerekir: tek bir ekleme yine $O(n)$ sürebilir. Örnekteki 17. ekleme tek başına 16 kopya yapmıştır. Her işlemin kısa sürmesi gereken gerçek zamanlı sistemlerde amortize sınır yetmez; en kötü durum sınırı önemlidir.

> **Sık yapılan hata**
>
> Kapasiteyi iki katına değil, sabit bir miktarla (örneğin her seferinde 10 yuva) artırmak. Bu durumda büyütmeler 10, 20, 30, … öğede yapılır ve n eklemedeki kopya toplamı yaklaşık $n^2/20$'ye çıkar. Toplam maliyet $O(n^2)$, ekleme başına ortalama $O(n)$ olur. Oranla büyütme amortize $O(1)$'in koşuludur.

8.3 Bağlı liste ile kuyruk

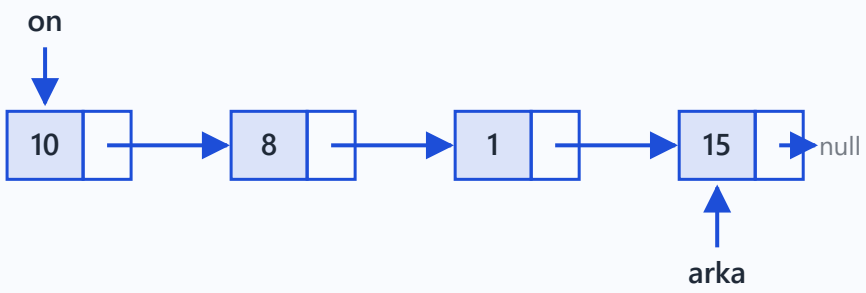
Dizi tabanlı kuyruğun kapasitesi baştan belirlenir; büyütme mümkündür ama kopyalama gerektirir. Bağlı liste ise öğe geldikçe yeni bir düğüm oluşturur ve hiç kopyalama yapmaz. Bu kesim kuyruğu tek yönlü bağlı listeyle gerçekler ve iki gerçeklemeyi karşılaştırır.

8.3.1 Bağlı liste temsili

Bağlı liste kuyruğunda her öğe bir **düğümdür** (*node*). Düğüm iki alan taşır: öğenin değeri (`veri`) ve listede kendisinden sonra gelen düğümün referansı (`sonraki`). Kuyruğun kendisi iki referans ve bir sayaç tutar: `on` listenin ilk düğümünü, `arka` son düğümünü gösterir; `uzunluk` öğe sayısıdır.

```
class Dugum {
    int veri;
    Dugum sonraki;
    Dugum(int veri) {
        this.veri = veri;
    }
}
// alanlar:
Dugum on, arka;
int uzunluk;
```

Ekleme listenin sonuna, çıkarma listenin başından yapılır. Boyut dinamiktir: her ekleme yeni bir düğüm ayırır, her çıkarma bir düğümü serbest bırakır. Bu yüzden kapasite sınırı, yani "dolu kuyruk" durumu yoktur; kuyruk bellek yettiği kadar büyür. Bunun bedeli, her düğüm için fazladan bir referans saklamak ve her eklemede yeni bir nesne oluşturmaktır.



Şekil 8.13: Bağlı liste kuyruğu. Önden arkaya 10, 8, 1, 15 öğeleri zincir hâlindedir. `on` ilk düğümü (10), `arka` son düğümü (15) gösterir; son düğümün `sonraki` referansı `null`'dır.

Neden ekleme sona, çıkarma başa yapılır da tersi yapılmaz? Cevap tek yönlü listenin yapısındadır. Listenin başından düğüm silmek kolaydır: `on = on.sonraki` tek bir atamadır ve *O(1)* sürer. Sona ekleme de `arka` referansı sayesinde kolaydır: yeni düğüm `arka.sonraki`'ye bağlanır ve `arka` ona ilerler. Oysa sondan silmek için sondan bir önceki düğümün bulunması gerekir, çünkü yeni son o düğüm olacaktır. Tek yönlü listede geriye doğru referans yoktur. O düğüme ulaşmak için listeyi baştan sona gezmek gerekir; bu da *O(n)*'dir. Uçlar ters seçilseydi `dequeue` *O(n)* olurdu.

Sık yapılan hata

Yalnız `on` referansını tutup `arka`'yı saklamamak. Bu durumda her `enqueue` son düğümü bulmak için listeyi baştan sona gezer ve *O(n)* olur. *n* öğeli bir kuyruğu doldurmak $O(n^2)$'ye çıkar. İki ucu da *O(1)* tutmanın koşulu iki referanstır.

8.3.2 enqueue ve dequeue algoritması

Gerçekleme ne olursa olsun, iki temel işlemin adımları aynıdır. Önce algoritmayı gerçeklemeden bağımsız olarak yazalım.

`enqueue(x)`

1. Kuyruk dolu mu? (Yalnız sabit kapasiteli dizide sorulur.)
2. Doluysa taşma hatası ver ve bitir.
3. Değilse `x`'i arka konuma yaz.
4. Arkayı bir sonraki konuma ilerlet.
5. Öğe sayısını artır ve başarıyla dön.

`dequeue()`

1. Kuyruk boş mu?
2. Boşsa alt taşma hatası ver ve bitir.
3. Değilse öndeki öğeyi al.
4. Önü bir sonraki öğeye ilerlet.
5. Öğe sayısını azalt ve alınan öğeyi döndür.

İki gerçekleme bu adımları farklı kodlarla yapar. Tablo 8.10 adımların karşılıklarını yan yana verir.

Adım	Dairesel dizi	Bağlı liste
Arkaya yaz	`dizi[(bas + boyut) % N] = x`	`arka.sonraki = yeni; arka = yeni`
Önü ilerlet	`bas = (bas + 1) % N`	`on = on.sonraki`
Dolu kontrolü	`boyut == N`	gerekmez (bellek yettikçe)

Tablo 8.10: Algoritma adımlarının iki gerçeklemedeki karşılığı.

Dairesel dizide “arkayı ilerletmek” ayrı bir adım değildir; `boyut` artınca bir sonraki yazma yeri kendiliğinden ilerler. Bağlı listede ise arkaya yazmak ile arkayı ilerletmek iki atamadır: önce eski son düğüm yenisine bağlanır, sonra `arka` yeni düğüme taşınır.

8.3.3 Kod izleme: enqueue

Bağlı liste kuyruğunun `enqueue` metodu şudur:

```
public void enqueue(int veri) {
    Dugum gecici = new Dugum(veri);
    if (bosMu()) {
        on = gecici;
    } else {
        arka.sonraki = gecici;
    }
    arka = gecici;
    uzunluk++;
}
```

Metot önce yeni bir düğüm oluşturur. `new Dugum(veri)` çağrısı, `veri` alanı verilen değer ve `sonraki` alanı `null` olan bir düğüm döndürür. Bu düğüm henüz listeye bağlı değildir. Ardından iki durum ayrılır. Kuyruk boşsa yeni düğüm aynı zamanda ilk düğümdür ve `on` ona yöneltilir. Boş değilse eski son düğümün `sonraki` alanı yeni düğümü gösterecek biçimde ayarlanır. İki durumda da yeni düğüm artık son düğümdür; bu yüzden `arka = gecici` satırı `if` / `else`'in dışındadır. Son olarak `uzunluk` bir artar.

Boş kuyruğa sırayla `enqueue(10)`, `enqueue(15)` ve `enqueue(20)` uygulayalım.

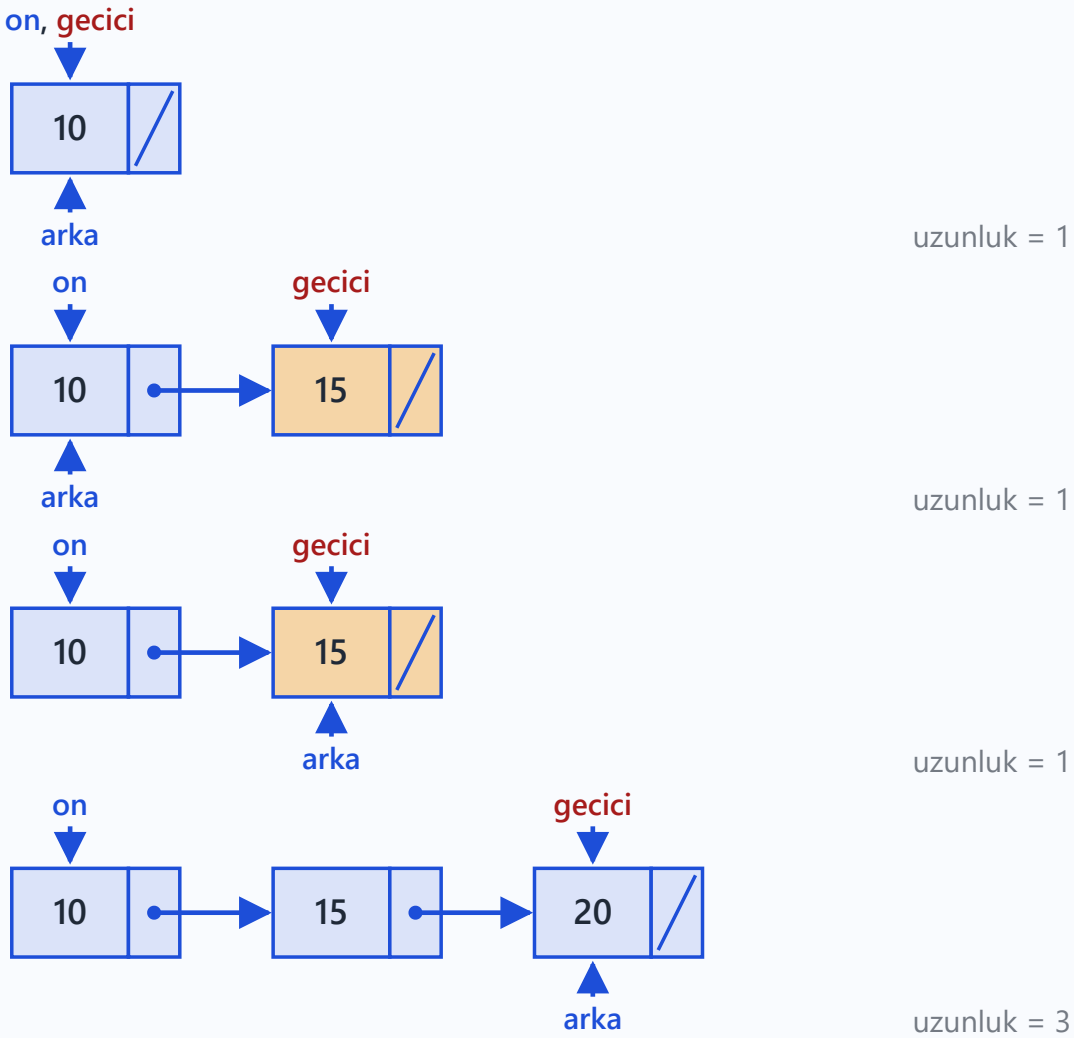
Çağrı	`bosMu()`	Çalışan dal	Liste (sonra)	`on`	`arka`	`uzunluk`
(başlangıç)	—	—	boş	`null`	`null`	0
`enqueue(10)`	`true`	`on = gecici`	10	10	10	1
`enqueue(15)`	`false`	`arka.sonraki = gecici`	10 → 15	10	15	2
`enqueue(20)`	`false`	`arka.sonraki = gecici`	10 → 15 → 20	10	20	3

Tablo 8.11: `enqueue` çağrılarının izlemesi. Liste önden arkaya yazılmıştır.

İlk çağrıda kuyruk boştur. `on = gecici` ile 10 düğümü ilk düğüm olur, `arka = gecici` ile son düğüm de olur. Tek düğüm hem öndür hem arkadır.

İkinci çağrıda `else` dalı çalışır. `arka.sonraki = gecici` satırı eski son düğümü (10) yeni düğüme (15) bağlar. Bu atama `arka` değişmeden *önce* yapılmalıdır, çünkü eski son düğüme yalnız `arka` üzerinden ulaşılır. Ardından `arka = gecici` ile arka 15'e ilerler. `on` değişmez. Üçüncü çağrı aynı adımları 20 için tekrarlar.

Hiçbir çağrıda liste gezilmemiştir. Her `enqueue` bir düğüm oluşturur, en çok iki referans atar ve bir sayaç artırır: *O(1)*.



Şekil 8.14: `enqueue` izlemesinin dört anı. (a) İlk çağrının sonu: tek düğüm (10) hem `on` hem `arka`'dır. (b) İkinci çağrıda `arka.sonraki = gecici`: 10, yeni düğüm 15'e bağlanmış ama `arka` henüz 10'dadır. (c) `arka = gecici`: arka 15'e ilerlemiştir. (d) Üç çağrıdan sonra liste 10 → 15 → 20'dir.

Sık yapılan hata

`arka = gecici` satırını yalnız `else` dalına yazmak ya da tamamen unutmak. Boş kuyruğa ilk eklemede `on` yeni düğümü gösterir, ama `arka` `null` kalır. İkinci eklemede `else` dalındaki `arka.sonraki` erişimi `null` üzerinde yapılır ve `NullPointerException` fırlatılır. Yeni düğüm her iki durumda da son düğümdür; atama `if` / `else`'ten sonra yapılmalıdır.

8.3.4 Kod izleme: dequeue

Bağlı liste kuyruğunun `dequeue` metodu şudur:

```java
public int dequeue() {
    if (bosMu())
        throw new NoSuchElementException();
    int sonuc = on.veri;
    on = on.sonraki;
    if (on == null)
        arka = null;
    uzunluk--;
    return sonuc;
}
```

Metot önce boşluğu sorar; kuyruk boşsa `NoSuchElementException` fırlatılır. Değilse öndeki değer `sonuc`'a alınır ve `on` bir sonraki düğüme ilerler. Eski ilk düğüme artık hiçbir referans bakmaz; onu çöp toplayıcı temizler. Ardından gelen `if (on == null) arka = null;` satırı uç durumu ele alır: çıkarılan düğüm listenin tek düğümüyse `on` `null` olmuştur ve kuyruk boşalmıştır. Bu durumda `arka` da `null` yapılır. Son olarak `uzunluk` bir azalır ve `sonuc` döndürülür.

Önceki izlemenin sonundaki 10 → 15 → 20 kuyruğuna üç kez `dequeue()` uygulayalım.

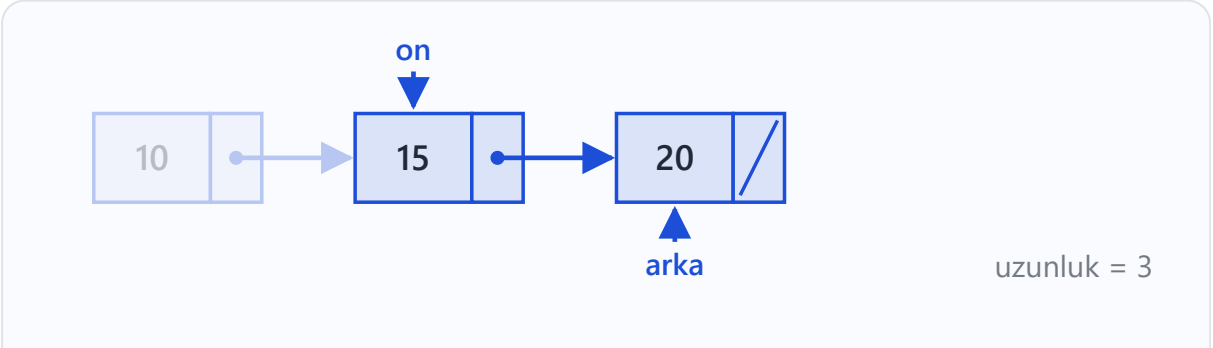
Çağrı	`sonuc`	`on` (sonra)	`on == null` ?	`arka` (sonra)	`uzunluk`	Liste (sonra)
(başlangıç)	—	10	—	20	3	10 → 15 → 20
1. `dequeue()`	10	15	`false`	20	2	15 → 20
2. `dequeue()`	15	20	`false`	20	1	20
3. `dequeue()`	20	`null`	`true`	`null`	0	boş

Tablo 8.12: `dequeue` çağrılarının izlemesi.

İlk iki çağrı aynı biçimde ilerler: öndeki değer alınır, `on` bir düğüm ilerler, `on` `null` olmadığı için `arka`'ya dokunulmaz. İkinci çağrıdan sonra `on` ve `arka` aynı düğümü (20) gösterir.

Üçüncü çağrı uç durumdur. `sonuc = 20` alınır ve `on = on.sonraki` ile `on` `null` olur. Ama `arka` hâlâ silinen 20 düğümünü gösterir. `on == null` koşulu doğrudur ve `arka = null` çalışır. Bu satır olmasaydı kuyruk boşken `arka` silinmiş düğümü göstermeye devam ederdi. Böyle bir tutarsız durumda `rear()` gibi bir sorgu kuyrukta olmayan bir değeri döndürür ve düğüm bellekten atılamaz.

Üç çağrının çıkış sırası 10, 15, 20'dir; giriş sırasıyla aynıdır. Her çağrı sabit sayıda adım atar: *O(1)*.





Şekil 8.15: `dequeue` izlemesinin üç anı. (a) İlk çağrıda `on = on.sonraki` : ön 15'e geçmiş, 10 düğümü listeden ayrılmıştır. (b) Üçüncü çağrıda `on` `null` olmuş, ama `arka` hâlâ silinen 20 düğümünü gösterir. (c) `arka = null` ile iki uç da `null` olmuş, kuyruk tutarlı biçimde boşalmıştır.

Sık yapılan hata

Son öğe çıkınca `arka` 'yı `null` yapmayı unutmak. Kuyruk boşalır ama `arka` eski düğümü gösterir. Ardından gelen bir `enqueue` , `bosMu()` doğru olduğu için `on` 'u yeni düğüme yöneltir ve `arka` 'yı da yeniler; bu yüzden hata çoğu zaman fark edilmez. Ama boş kuyrukta `rear()` çağrılırsa yanlış değer döner. Bu tür "bazen görünen" hatalar en zor bulunan hatalardır.

8.3.5 Dairesel dizi mi, bağlı liste mi?

İki gerçeklemenin de temel işlemleri *O(1)*'dir. Seçimi belirleyen, bellek kullanımı ve pratik hızdır.

Dairesel dizinin öğeleri bellekte ardışıktır. İşlemci belleği bloklar hâlinde okuduğu için ardışık veriye erişim, bellekte dağınık duran düğümlere erişimden hızlıdır; bu özelliğe *önbellek dostu* olmak denir. Öğe başına ek referans da yoktur. Buna karşın kapasite sabittir. Kapasite dolunca ya taşma hatası verilir ya da daha büyük bir diziye kopyalama yapılır; bu kopyalama tek başına *O(n)*, amortize *O(1)*'dir. Dizi ayrıca kapasite kadar yer kaplar; kuyrukta az öğe varken yuvaların çoğu boş durur.

Bağlı listenin kapasite sınırı yoktur ve gerektiği kadar büyür. Her ekleme ve çıkarma, amortize değil, *kesin O(1)*'dir; hiçbir zaman toplu kopyalama yapılmaz. Bellek öğe sayısıyla orantılıdır. Buna karşın her düğüm ek bir referans taşır ve her eklemede yeni bir nesne oluşturulur. Nesne oluşturma ve sonradan çöp toplama, dizideki bir yuvaya yazmaktan pahalıdır.

İşlem	Dairesel dizi	Bağlı liste
`enqueue`	O(1)	O(1)
`dequeue`	O(1)	O(1)
`first` / `size` / `isEmpty`	O(1)	O(1)

İşlem	Dairesel dizi	Bağlı liste
Bellek	O(N) (kapasite kadar)	O(n) (öğe sayısı kadar)

Tablo 8.13: İki kuyruk gerçeklemesinin maliyetleri (*n*: öğe sayısı, *N*: kapasite).

Pratikte çoğu durumda dairesel dizi tercih edilir; Java'nın `ArrayDeque` sınıfının bu yolu seçmesi de bu yüzdendir. Bağlı liste, her işlemin kesin olarak kısa sürmesi gereken ya da kuyruk boyunun çok geniş bir aralıkta dalgalandığı durumlarda öne çıkar.

8.4 Kuyruk türleri ve çift uçlu kuyruk

Temel kuyruk, ekleme ve çıkarmayı iki sabit uca bağlar. Bu kuralın gevşetildiği ya da değiştirildiği birkaç kuyruk türü vardır. Bu kesim türleri tanıtır ve en genel olanı, iki uçtan da ekleme ve çıkarma yapılabilen çift uçlu kuyruğu ayrıntılı işler.

8.4.1 Kuyruk türleri

Tür	Ekleme	Çıkarma	Not
Basit kuyruk	arka	ön	FIFO
Giriş sınırlı (*input restricted*)	yalnız arka	ön ve arka	kısıtlı çift uçlu kuyruk
Çıkış sınırlı (*output restricted*)	ön ve arka	yalnız ön	kısıtlı çift uçlu kuyruk
Dairesel (*circular*)	arka	ön	dizi yuvaları yeniden kullanılır
Çift uçlu (*deque*)	ön ve arka	ön ve arka	FIFO'ya bağlı değil
Öncelikli (*priority*)	herhangi	en yüksek öncelik	sıra önceliğe göre

Tablo 8.14: Kuyruk türleri.

Tablodaki türler iki gruba ayrılır. Dairesel kuyruk yeni bir davranış tanımlamaz; basit kuyruğun bir gerçekleme biçimidir. Kullanıcı açısından basit kuyruktan farkı yoktur. Öteki türler ise ADT'yi değiştirir.

Çift uçlu kuyruk iki uçta da hem ekleme hem çıkarma yapar ve en genel türdür. **Giriş sınırlı** ve **çıkış sınırlı** kuyruklar onun kısıtlanmış biçimleridir. Giriş sınırlı kuyrukta ekleme yalnız arkadan yapılır, ama çıkarma iki uçtan da yapılabilir. Böyle bir yapı hem kuyruk (önden çıkararak) hem yığın (arkadan çıkararak) gibi kullanılabilir. Çıkış sınırlı kuyrukta ise öğe iki uçtan da girebilir, ama yalnız önden çıkar. Örneğin acil bir iş sıranın önüne, sıradan bir iş sona eklenebilir.

Öncelikli kuyruk (*priority queue*) ise geliş sırasını hiç dikkate almaz. Her öğenin bir önceliği vardır ve çıkarma her zaman en yüksek öncelikli öğeyi verir. Hastanenin acil servisinde hastalar geliş sırasıyla değil, durumlarının ciddiyetine göre muayene edilir. Bu yapı genellikle heap ile gerçeklenir ve ayrı bir konu olarak incelenir. Burada akılda tutulması gereken, adında "kuyruk" geçse de FIFO olmadığıdır.

Sık yapılan hata

Deque ile `dequeue`'yu karıştırmak. Deque, İngilizce *double-ended queue* (çift uçlu kuyruk) ifadesinin kısaltmasıdır ve "dek" diye okunur. `dequeue` ise "kuyruktan çıkar" işleminin adıdır. Biri bir veri yapısı, öteki bir işlemdir.

Sık yapılan hata

Java'daki `PriorityQueue` sınıfını FIFO sanmak. `Queue<Integer> q = new PriorityQueue<>();` satırında değişkenin türü `Queue`'dur, ama nesne öncelikli kuyruktur. `q.offer(5); q.offer(1); q.offer(3);` sonrasında `poll()` önce 1'i, sonra 3'ü verir: doğal sıralamada en küçük öğe en yüksek önceliklidir. Geliş sırası korunmaz.

8.4.2 Çift uçlu kuyruk (deque)

Tanım 8.3

Çift uçlu kuyruk (*double-ended queue, deque*), öğeleri bir sıra içinde tutan ve hem ön hem arka uçtan ekleme ve çıkarma yapılabilen soyut veri tipidir.



Şekil 8.16: Çift uçlu kuyruk. Öğeler (önden arkaya 27, 26, 22, 21, 20, 23, 24, 25) hem ön (*front/head*) hem arka (*rear/tail*) uçtan eklenebilir ve çıkarılabilir.

Çift uçlu kuyruk FIFO kuralına bağlı değildir. Hangi davranışı göstereceği, onu kullanan kodun hangi uçları seçtiğine bağlıdır. Yalnız arka uçtan ekleyip yine arka uçtan çıkaran bir çift uçlu kuyruk **yığın** (LIFO) gibi davranır. Arkadan ekleyip önden çıkaran bir çift uçlu kuyruk ise **kuyruk** (FIFO) olur. Bu yüzden çift uçlu kuyruk, yığın ve kuyruğun ikisini de içine alan genel bir yapıdır. Java'nın `ArrayDeque` sınıfının hem yığın hem kuyruk olarak önerilmesinin nedeni budur.

Şekil 8.16'daki öğe sırası da bu esnekliği yansıtır. 20 ilk eklenen öğe olsun. Ardından 21 ve 22 öne, 23 ve 24 arkaya, 26 ve 27 yine öne, 25 arkaya eklenmiş olabilir. Sonuçta kuyrukta öğeler ne geliş sırasındadır ne de ters sırada; her öğenin yeri hangi uca eklendiğine bağlıdır.

Tablo 8.15 bu notta kullanılan işlem adlarını ve Java'daki `ArrayDeque` karşılıklarını verir.

İşlem	Java `ArrayDeque`	Süre
`basaEkle(e)`	`offerFirst(e)`	O(1)
`sonaEkle(e)`	`offerLast(e)`	O(1)
`bastanCikar()`	`pollFirst()`	O(1)
`sondanCikar()`	`pollLast()`	O(1)
`ilk()` / `son()`	`peekFirst()` / `peekLast()`	O(1)

Tablo 8.15: Çift uçlu kuyruk işlemleri.

`ArrayDeque` bunlara ek olarak yığın işlemlerini de sunar: `push(e)` başa ekler (`addFirst`), `pop()` baştan çıkarır (`removeFirst`). Kuyruk işlemleri `offer` ve `poll` ise sırasıyla `offerLast` ve `pollFirst`'tür. Aynı nesne üzerinde iki takımı karıştırmak kafa karıştırabilir. `push` ile eklenen öğe *öne* girer; `offer` ile eklenen öğe ise *arkaya* girer.

Çift uçlu kuyruk iki yoldan gerçeklenebilir. Dairesel dizi yolunda başa ekleme, `bas = (bas - 1 + N) % N` ile önü bir yuva geri alır ve yeni öğeyi oraya yazar. Sondan çıkarma ise `boyut`'u bir azaltmakla yetinir. `ArrayDeque` bu yolu kullanır. Bu notta ise ikinci yol, çift yönlü bağlı liste işlenir.

Örnek 8.4

Bir kelimenin palindrom olup olmadığı (tersten okunuşunun aynı olup olmadığı) çift uçlu kuyrukla denetlenebilir. Harfler sırayla sona eklenir; sonra iki uçtan birer harf çıkarılıp karşılaştırılır.

```
boolean palindromMu(String s) {
    Deque<Character> d = new ArrayDeque<>();
    for (char c : s.toCharArray())
        d.offerLast(c);
    while (d.size() > 1)
        if (!d.pollFirst().equals(d.pollLast()))
            return false;
    return true;
}
```

1. `"kayak"`: Kuyruk k, a, y, a, k olur. İlk turda baştan k, sondan k çıkar; eşittir. İkinci turda a ile a eşittir. Kuyrukta yalnız y kalır; `size() > 1` yanlıştır ve döngü biter. Sonuç `true`'dur.
2. `"kalem"`: İlk turda baştan k, sondan m çıkar. Eşit değildir ve metot hemen `false` döndürür.

Her harf bir kez eklenir ve en çok bir kez çıkarılır: *O(n)* zaman, *O(n)* ek alan. Karşılaştırmada `!=` değil `equals` kullanılır, çünkü `pollFirst()` ve `pollLast()` `Character` nesneleri döndürür; `!=` değerleri değil nesne kimliklerini karşılaştırırdı.

8.4.3 Çift uçlu kuyruk işlemleri adım adım

Çift uçlu kuyruğu **çift yönlü bağlı liste** (*doubly linked list*) ile gerçekleyeceğiz. Her düğümde iki referans vardır: `onceki` bir önceki düğümü, `sonraki` bir sonraki düğümü gösterir. Kuyruğun kendisi `bas` (ilk düğüm), `son` (son düğüm) ve `boyut` alanlarını tutar.

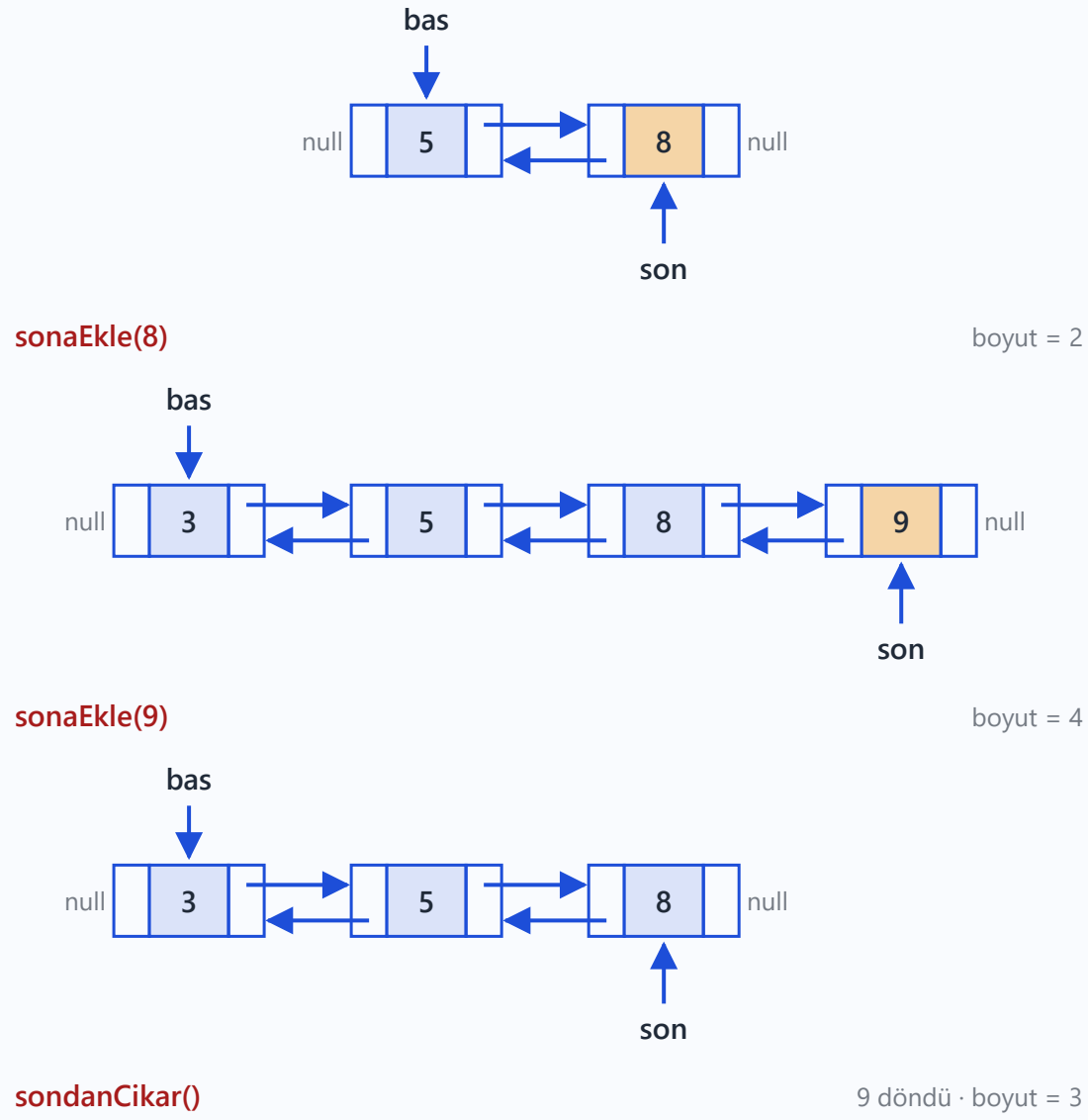
Boş bir çift uçlu kuyruğa sırayla `sonaEkle(5)`, `sonaEkle(8)`, `basaEkle(3)`, `sonaEkle(9)`, `sondanCikar()` ve `bastanCikar()` uygulayalım.

İşlem	Değişen referanslar	Liste (sonra)	`boyut`	Döner
(başlangıç)	—	boş; `bas = son = null`	0	—
`sonaEkle(5)`	`bas = son = yeni`	5	1	—
`sonaEkle(8)`	`yeni.onceki = son; son.sonraki = yeni; son = yeni`	5 ⇄ 8	2	—
`basaEkle(3)`	`yeni.sonraki = bas; bas.onceki = yeni; bas = yeni`	3 ⇄ 5 ⇄ 8	3	—
`sonaEkle(9)`	(sona ekleme)	3 ⇄ 5 ⇄ 8 ⇄ 9	4	—
`sondanCikar()`	`son = son.onceki; son.sonraki = null`	3 ⇄ 5 ⇄ 8	3	9
`bastanCikar()`	`bas = bas.sonraki; bas.onceki = null`	5 ⇄ 8	2	3

Tablo 8.16: Çift uçlu kuyruk işlemlerinin izlemesi.

İlk ekleme özel bir durumdur: liste boş olduğu için yeni düğüm hem baş hem son olur. Sonraki eklemelerde yeni düğüm, eklendiği uçtaki düğüme *iki yönde* bağlanır. Sona eklemede yeni düğümün `onceki`'si eski son düğümü, eski son düğümün `sonraki`'si yeni düğümü gösterir. Başa ekleme bunun aynadaki görüntüsüdür.

Her iki uç da referansla tutulduğu için bütün eklemeler *O(1)*'dir. Çıkarmalarda ise `onceki` referansının değeri ortaya çıkar. `sondanCikar()`, yeni son düğümü `son.onceki` ile tek adımda bulur. Tek yönlü listede sondan çıkarma *O(n)* olurdu, çünkü sondan bir önceki düğüm baştan gezilerek bulunmak zorundaydı. `onceki` referansı bu işlemi *O(1)* yapar. Bunun bedeli düğüm başına bir referans daha saklamaktır.



Şekil 8.17: Tablo 8.16'nın üç anı. (a) İki `sonaEkle`'den sonra 5 ⇄ 8; her bağ iki yönlüdür. (b) `basaEkle(3)` ve `sonaEkle(9)`'dan sonra 3 ⇄ 5 ⇄ 8 ⇄ 9; `boyut = 4`. (c) `sondanCikar()` 9'u döndürmüş, `son` 8'e geri çekilmiştir.

8.4.4 Kod: başa ve sona ekleme

Çift yönlü listenin düğüm sınıfı şudur:

```
class Dugum<E> {
    E veri;
    Dugum<E> onceki, sonraki;
    Dugum(E veri) { this.veri = veri; }
}
```

Ekleme metotları:

```
public void basaEkle(E veri) {
    Dugum<E> d = new Dugum<>(veri);
    if (bosMu()) {
        bas = son = d;
    } else {
        d.sonraki = bas;
        bas.onceki = d;
        bas = d;
    }
    boyut++;
}
```

```
public void sonaEkle(E veri) {
    Dugum<E> d = new Dugum<>(veri);
    if (bosMu()) {
        bas = son = d;
    } else {
        d.onceki = son;
        son.sonraki = d;
        son = d;
    }
    boyut++;
}
```

İki metot aynı kalıbı izler. Önce yeni düğüm oluşturulur; `onceki` ve `sonraki` alanları `null`'dır. Çift uçlu kuyruk boşsa yeni düğüm hem baş hem son olur: `bas = son = d;` atama zinciri sağdan sola çalışır, önce `son`'a sonra `bas`'a `d` atanır. Boş değilse üç atama yapılır.

`basaEkle`'de önce yeni düğüm eski başı gösterir (`d.sonraki = bas`). Sonra eski baş geri bağlantıyla yeni düğümü gösterir (`bas.onceki = d`). En son baş referansı yeni düğüme geçer (`bas = d`). Bu sıra önemlidir: `bas = d` önce yapılsaydı eski başa ulaşılamaz ve bağlar kurulamazdı.

İki metot birbirinin aynadaki görüntüsüdür. `basaEkle`'de `bas` yerine `son`, `onceki` yerine `sonraki` (ve tersi) yazılırsa `sonaEkle` elde edilir. Hiçbir metotta döngü yoktur; ikisi de *O(1)*'dir.

8.4.5 Kod izleme: basaEkle()

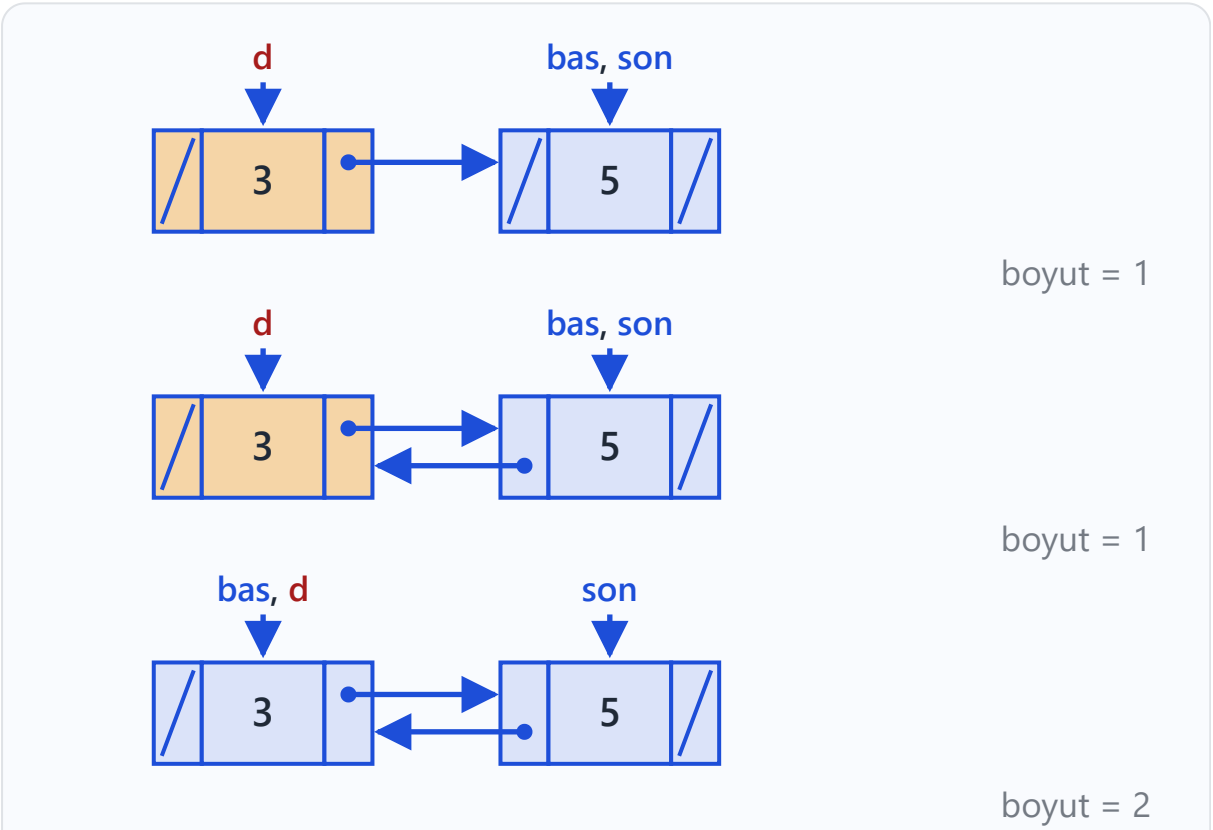
Boş bir çift uçlu kuyruğa önce `basaEkle(5)`, sonra `basaEkle(3)` uygulayalım. 3 başa geçecektir.

Satır	Etki	`bas`	`son`	`boyut`
`basaEkle(5)`				
`new Dugum<>(5)`	`onceki = sonraki = null`	`null`	`null`	0
`bosMu()`	`true`	`null`	`null`	0
`bas = son = d`	tek düğüm iki ucu birden oluşturur	5	5	0

Satır	Etki	`bas`	`son`	`boyut`
`boyut++`		5	5	1
`basaEkle(3)`				
`new Dugum<>(3)`	`onceki = sonraki = null`	5	5	1
`bosMu()`	`false`	5	5	1
`d.sonraki = bas`	yeni düğüm eski başı (5) gösterir	5	5	1
`bas.onceki = d`	geri bağlantı: 5 yeni düğümü gösterir	5	5	1
`bas = d`	baş referansı yeni düğüme geçer	3	5	1
`boyut++`		3	5	2

Tablo 8.17: `basaEkle` çağrılarının izlemesi.

İzlemenin sonunda liste 3 ⇄ 5'tir. Baştan sona gezilirse 3, 5; sondan başa gezilirse 5, 3 okunur. İki yönde de gezilebilmesi, `d.sonraki = bas` ve `bas.onceki = d` atamalarının ikisinin de yapılmasına bağlıdır. Hiç gezinme yoktur; işlem *O(1)*'dir.



Şekil 8.18: `basaEkle(3)`'ün üç anı. (a) `d.sonraki = bas`: yeni düğüm 5'i gösterir, ama 5 henüz geri bağlı değildir. (b) `bas.onceki = d`: bağ iki yönlü olmuştur; `bas` hâlâ 5'tedir. (c) `bas = d` ve `boyut++` sonrası liste 3 ⇄ 5, `boyut = 2`.

8.4.6 Kod: baştan ve sondan çıkarma

```
public E bastanCikar() {
    if (bosMu()) return null;
    E veri = bas.veri;
    bas = bas.sonraki;
    if (bas == null)
        son = null;       // boşaldı
    else
        bas.onceki = null;
    boyut--;
    return veri;
}
```

```
public E sondanCikar() {
    if (bosMu()) return null;
    E veri = son.veri;
    son = son.onceki;
    if (son == null)
        bas = null;       // boşaldı
    else
        son.sonraki = null;
    boyut--;
    return veri;
}
```

İki metot da önce boşluğu sorar. Boş çift uçlu kuyrukta, Goodrich'in arayüzündeki gibi `null` döndürülür. Değilse uçtaki değer `veri`'ye alınır ve uç referansı içeri doğru bir düğüm ilerler: `bastanCikar`'da `bas = bas.sonraki`, `sondanCikar`'da `son = son.onceki`.

Ardından iki durum ayrılır. Uç referansı `null` olduysa çıkarılan düğüm tek düğümdü ve liste boşalmıştır. Bu durumda öbür uç da `null` yapılır. Liste boşalmadıysa yeni uç düğümün dışarıyı gösteren bağı koparılır: `bas.onceki = null` ya da `son.sonraki = null`. Böylece çıkarılan düğüme listeden hiçbir bağ kalmaz. Son olarak `boyut` azalır ve değer döndürülür.

> **Sık yapılan hata**
>
> Son öğe çıkarıldığında iki ucu birden `null` yapmamak. Örneğin `sondanCikar`'da `bas = null` satırı unutulursa `son` `null` iken `bas` silinmiş düğümü göstermeye devam eder. Düğüm bellekten atılamaz ve `ilk()` kuyrukta olmayan bir değer döndürür. Bağlı liste kuyruğundaki `arka = null` hatasının aynısıdır.

8.4.7 Kod izleme: sondanCikar()

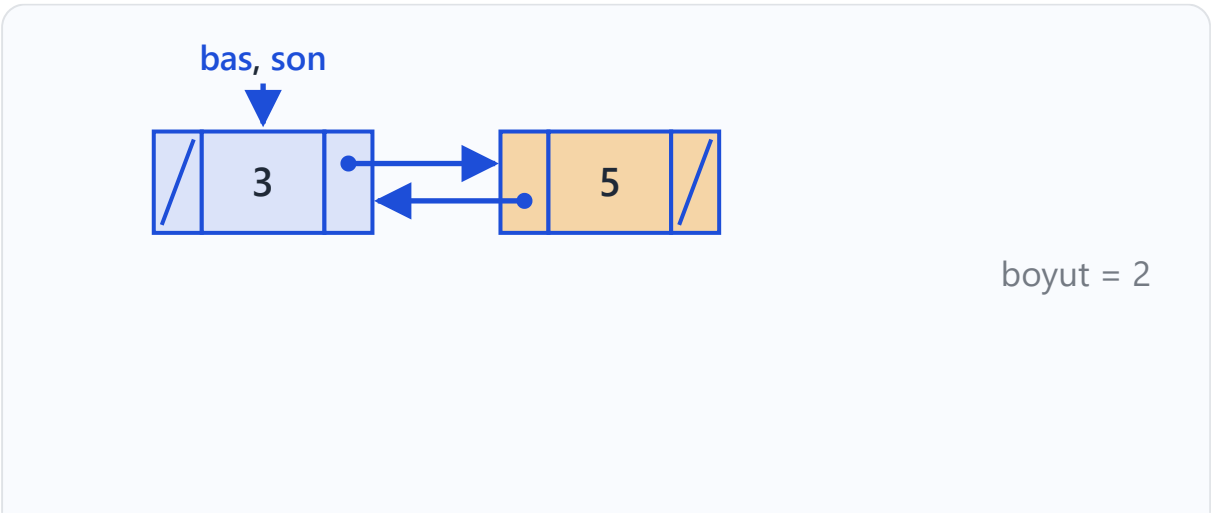
Çift uçlu kuyruk 3 ⇄ 5 olsun. İki kez `sondanCikar()` çağrılacaktır: önce 5, sonra 3 çıkacak ve kuyruk boşalacaktır.

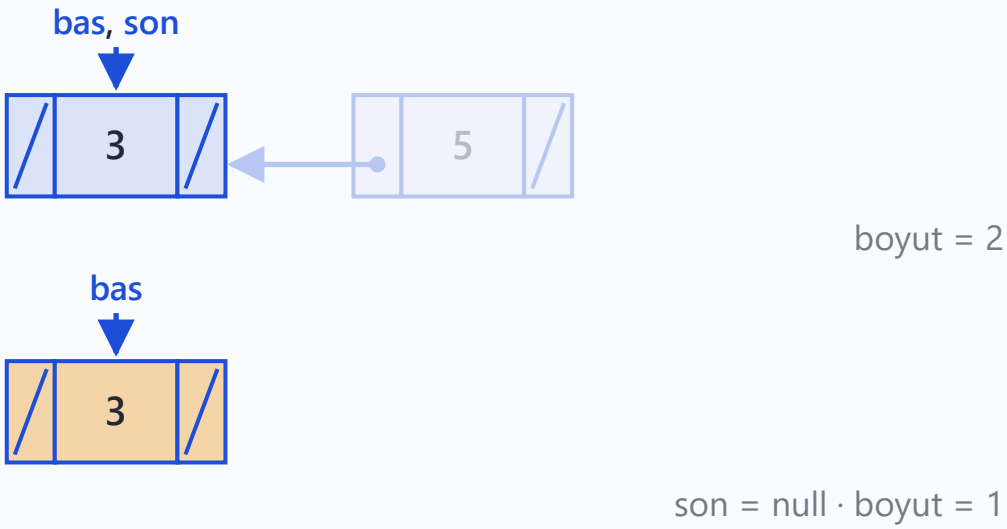
Satır	Etki	bas	son	boyut
1. `sondanCikar()`				
`bosMu()`	`false` (boyut 2)	3	5	2
`veri = son.veri`	`veri = 5`	3	5	2
`son = son.onceki`	son 3'e geri çekilir	3	3	2
`son == null` ?	`false` : kuyruk boşalmadı	3	3	2
`son.sonraki = null`	5'e giden bağ kopar	3	3	2
`boyut--` ; `return`	5 döner	3	3	1
2. `sondanCikar()`				
`bosMu()`	`false` (boyut 1)	3	3	1
`veri = son.veri`	`veri = 3`	3	3	1
`son = son.onceki`	tek düğüm vardı: `son = null`	3	`null`	1
`son == null` ?	`true` : kuyruk boşaldı	3	`null`	1
`bas = null`	iki uç da `null`	`null`	`null`	1
`boyut--` ; `return`	3 döner	`null`	`null`	0

Tablo 8.18: `sondanCikar` çağrılarının izlemesi.

İlk çağrıda `son = son.onceki` satırı, sondan bir önceki düğümü aramadan bulur. `son` `null` olmadığı için `else` dalı çalışır ve `son.sonraki = null` ile 5 düğümüne giden bağ koparılır. 5 düğümü listeden ayrılmıştır.

İkinci çağrıda tek düğüm vardır ve `son.onceki` `null`'dır. `son` `null` olur, ama `bas` hâlâ 3 düğümünü gösterir. `son == null` koşulu doğru olduğu için `bas = null` çalışır. Bu satır olmasaydı `bas` silinmiş düğümü göstermeye devam ederdi. Her iki çağrı da sabit sayıda adım atar: *O(1)*.





Şekil 8.19: `sondanCikar` izlemesinin üç anı. (a) İlk çağrıda `son = son.onceki` : `bas` ve `son` 3'tedir, 5 düğümü henüz bağlıdır. (b) `son.sonraki = null` : 5'e giden bağ kopmuştur. (c) İkinci çağrıda `son = null` olmuş, ama `bas` hâlâ 3'ü gösterir; bir sonraki satır onu da `null` yapar.

8.4.8 Kayan pencerede en büyük

Çift uçlu kuyruğun iki ucunu birlikte kullanan klasik bir problem şudur: bir `a` dizisi ve bir *k* pencere genişliği verilsin. Dizi üzerinde soldan sağa *k* uzunluğunda bir pencere kaydırılır. Her pencere konumundaki en büyük değer istenir. Bu tür probleme **kayan pencere** (*sliding window*) problemi denir. Örneğin `a = {1, 3, -1, -3, 5, 3, 6}` ve *k = 3* için beş pencere vardır: {1, 3, −1}, {3, −1, −3}, {−1, −3, 5}, {−3, 5, 3}, {5, 3, 6}. En büyükler sırasıyla 3, 3, 5, 5, 6'dır.

Kaba kuvvet çözüm her pencerede *k* öğeyi tarar. *n − k + 1* pencere olduğu için maliyet *O(n·k)*'dir. *k* büyüdükçe bu yavaşlar; *k = n/2* için $O(n^2)$'ye çıkar. Daha iyi bir çözüm, pencere kayarken önceki hesaplardan yararlanır. Fikir şudur: bir öğenin sağında ondan büyük ya da ona eşit bir öğe varsa, o öğe bundan sonraki hiçbir pencerede en büyük olamaz. Çünkü pencere sağa kaydıkça soldaki öğe pencereden daha önce çıkar, sağdaki büyük öğe ise pencerede kalmaya devam eder. Böyle öğeler aday listesinden hemen atılabilir.

Algoritma, adayların *indislerini* bir çift uçlu kuyrukta tutar. Kuyruktaki indislere karşılık gelen değerler önden arkaya azalan sıradadır; bu yüzden en büyük değer hep öndedir. Her yeni *i* indisi için üç iş yapılır:

1. **Önden atma:** Öndeki indis pencerenin dışına düştüyse (≤ *i − k*) çıkarılır.
2. **Arkadan atma:** Arkadaki indislerin değeri `a[i]` 'den küçük ya da ona eşitse bunlar artık aday değildir; arkadan çıkarılır.
3. **Ekleme:** *i* arkaya eklenir. İlk tam pencereye ulaşıldıysa (*i ≥ k − 1*) öndeki indisin değeri o pencerenin en büyüğüdür.

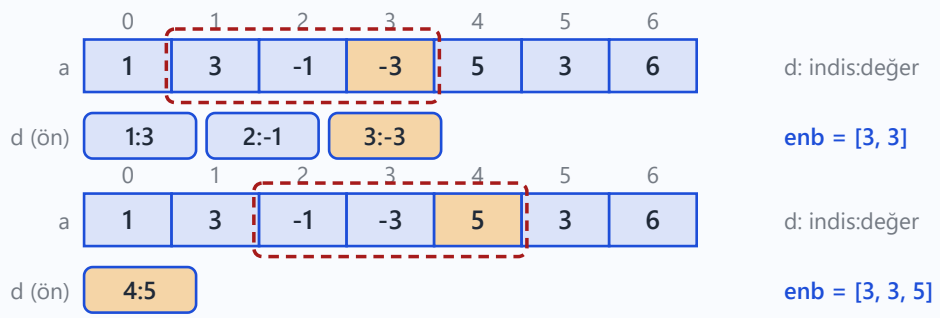
```
int[] pencereMax(int[] a, int k) {
  int[] enb = new int[a.length - k + 1];
  Deque<Integer> d = new ArrayDeque<>(); // indis
  for (int i = 0; i < a.length; i++) {
    if (!d.isEmpty() && d.peekFirst() <= i - k)
      d.pollFirst();        // pencere dışı
    while (!d.isEmpty() && a[d.peekLast()] <= a[i])
      d.pollLast();         // aday değil
    d.offerLast(i);
    if (i >= k - 1) enb[i - k + 1] = a[d.peekFirst()];
  }
  return enb;
}
```

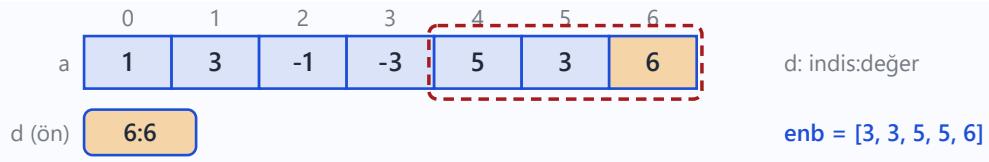
Kodda `d` indis tutar; değerlere `a[d.peekFirst()]` ve `a[d.peekLast()]` ile ulaşılır. İndis tutulur, çünkü bir öğenin pencereden çıkıp çıkmadığı konumundan anlaşılır. Yalnız değer tutulsaydı, aynı değerden birden çok olduğunda hangisinin çıktığı bilinemezdi. `enb` dizisi $n - k + 1$ sonucu tutar ve i. indisle biten pencerenin sonucu `enb[i - k + 1]`'e yazılır.

i	`a[i]`	Atılanlar	`d` (sonra)	`enb`
0	1	—	0:1	—
1	3	arkadan 0:1	1:3	—
2	−1	—	1:3, 2:−1	[3]
3	−3	—	1:3, 2:−1, 3:−3	[3, 3]
4	5	önden 1:3 (pencere dışı); arkadan 3:−3, 2:−1	4:5	[3, 3, 5]
5	3	—	4:5, 5:3	[3, 3, 5, 5]
6	6	arkadan 5:3, 4:5	6:6	[3, 3, 5, 5, 6]

Tablo 8.19: `pencereMax({1, 3, -1, -3, 5, 3, 6}, 3)` izlemesi. `d` önden arkaya "indis:değer" biçiminde yazılmıştır.

İzlemenin kritik adımı $i = 4$'tür. Pencere artık 2–4. indisleri kapsar ve öndeki 1. indis $1 \le 4 - 3 = 1$ olduğu için pencerenin dışındadır; önden atılır. Ardından gelen 5, arkadaki −3 ve −1'den büyüktür. İkisi de bundan sonra hiçbir pencerede en büyük olamaz ve arkadan atılır. Kuyrukta yalnız 4:5 kalır. $i = 3$'te ise −3 arkaya eklenmiş ama hiçbir şey atılmamıştı: −3 küçüktür, ama ileride 3 ve −1 pencereden çıktığında en büyük olma şansı vardı.





Şekil 8.20: Kayan pencere izlemesinin üç anı. Kesik çerçeve o anki penceredir. (a) *i* = *3*: `d` = 1:3, 2:−1, 3:−3; değerler önden arkaya azalır. (b) *i* = *4*: 1 önden, 3 ve 2 arkadan atılmış, `d`'de yalnız 4:5 kalmıştır. (c) *i* = *6*: 6 arkadaki iki adayı atmıştır; sonuç [3, 3, 5, 5, 6].

Teorem 8.2 (Kayan pencerenin maliyeti)

`pencereMax`, *n* öğeli bir dizide *O(n)* zamanda çalışır ve *O(k)* ek alan kullanır.

İspat. İç `while` döngüsü tek bir *i* için birçok kez dönebilir; bu yüzden döngülere tek tek bakmak yanıltıcıdır. Bunun yerine her indisin başına gelenleri sayalım. Her indis `offerLast` ile tam bir kez kuyruğa girer. Bir indis kuyruktan en çok bir kez çıkar: ya önden (`pollFirst`) ya arkadan (`pollLast`). Dolayısıyla bütün çalışma boyunca toplam ekleme sayısı *n*, toplam çıkarma sayısı en çok *n*'dir. Her kuyruk işlemi *O(1)* olduğu için toplam iş *O(n)*'dir. Kuyrukta yalnız pencere içindeki indisler bulunabildiği için ek alan en çok *k*'dır. □

Bu ispattaki sayma tekniği, yani “her öğe bir kez girer, en çok bir kez çıkar” gözlemi, kuyruk ve yığın algoritmalarında çok sık kullanılır. İki yığınla kuyruk gerçeklemesinin amortize analizi de aynı fikre dayanır.

Sık yapılan hata

Kayan pencerede *en küçüğü* bulmak için kodu değiştirirken karşılaştırmayı ters çevirmeyi unutmak. En küçük için kuyruktaki değerler önden arkaya *artan* olmalıdır. Arkadan atma koşulu `a[d.peekLast()] >= a[i]` olur: yeni gelen öğeden büyük ya da ona eşit olan adaylar atılır.

8.5 Kuyruk uygulamaları

Bu kesimdeki ilk üç problemde kuyruk, işleri adil bir sırayla döndürmek ya da bir yapıyı düzey düzey gezmek için kullanılır. Dördüncü ve son problem ise kuyruğun yığınla nasıl kurulabileceğini gösterir.

8.5.1 Round robin çizelgeleyici

Bir işletim sisteminde aynı anda çalışmak isteyen birçok süreç vardır, ama tek bir işlemci çekirdeği bir anda yalnız bir süreç çalıştırabilir. İşlemciyi süreçler arasında paylaştıran bileşene **çizelgeleyici** (*scheduler*) denir. En sade ve en adil yöntemlerden biri **round robin** (sırayla dönen çizelgeleme) yön-

temidir.

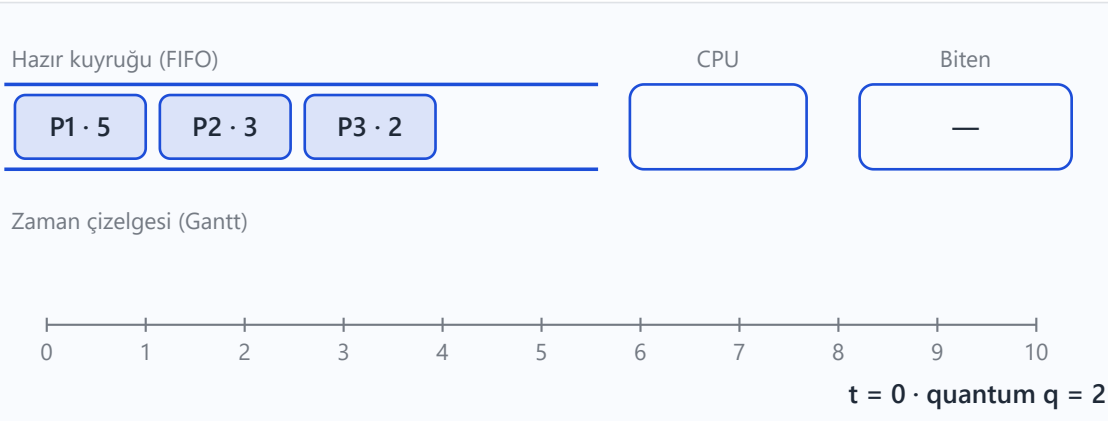
Yöntemde çalışmaya hazır süreçler bir FIFO *hazır kuyruğunda* bekler. Kuyruğun önündeki süreç alınır ve en fazla *q* birim süre çalıştırılır. Bu süreye **zaman dilimi** (*time slice, quantum*) denir. Süreç bu sürede biterse sistemden ayrılır. Bitmezse kalan süresiyle birlikte kuyruğun arkasına döner ve sırası yeniden gelene kadar bekler. Hiçbir süreç işlemciyi uzun süre tekeline alamaz; her süreç en geç bir tur sonra yeniden çalışır.

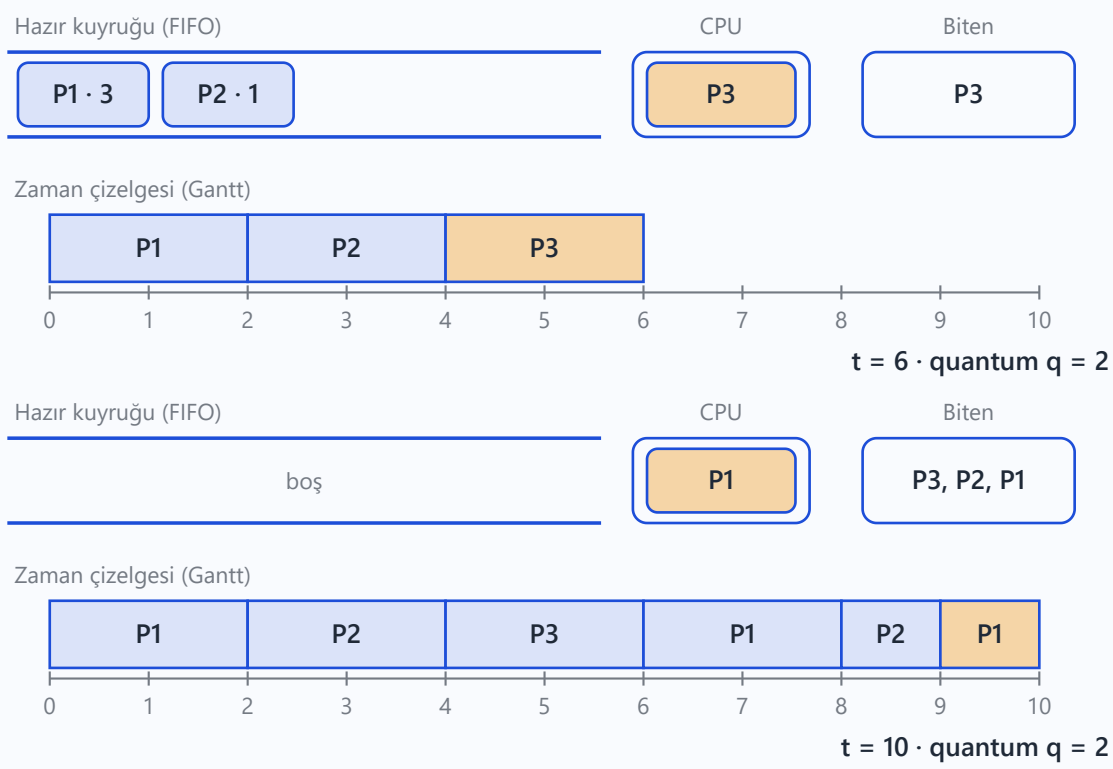
Örnek 8.5

Hazır kuyruğunda sırayla P1, P2 ve P3 süreçleri var; kalan süreleri 5, 3 ve 2 birimdir. Zaman dilimi *q = 2*'dir. Her sürecin tamamlanma anını bulalım.

1. **t = 0–2:** P1 önden alınır ve 2 birim çalışır. Kalan 3'tür; P1 arkaya eklenir. Kuyruk: P2, P3, P1.
2. **t = 2–4:** P2 2 birim çalışır; kalan 1 ile arkaya döner. Kuyruk: P3, P1, P2.
3. **t = 4–6:** P3'ün 2 birimi bir dilime sığar. P3 *t = 6*'da biter ve kuyruğa dönmez. Kuyruk: P1, P2.
4. **t = 6–8:** P1 yine 2 birim çalışır; kalan 1 ile arkaya döner. Kuyruk: P2, P1.
5. **t = 8–9:** P2'nin kalan 1 birimi biter (*t = 9*). Dilimin tamamı kullanılmaz; işlemci hemen sıradakine geçer. Kuyruk: P1.
6. **t = 9–10:** P1'in son birimi çalışır ve P1 *t = 10*'da biter. Kuyruk boşalır.

Tamamlanma zamanları P3 = 6, P2 = 9, P1 = 10'dur. En kısa süreç ilk, en uzun süreç son bitmiştir; oysa P1 kuyruğa ilk girmişti.





Şekil 8.21: Örnek 8.5'in üç anı. Üstte hazır kuyruğu, CPU ve biten süreçler; altta zaman çizelgesi (Gantt şeması). (a) *t = 0*: kuyrukta P1 · 5, P2 · 3, P3 · 2. (b) *t = 6*: P3 bitmiş, kuyrukta P1 · 3 ve P2 · 1 kalmıştır. (c) *t = 10*: bütün süreçler bitmiştir; tamamlanma sırası P3, P2, P1'dir.

Round robin'in başarımı zaman diliminin seçimine bağlıdır. q çok büyük seçilirse her süreç bir dilimde biter ve yöntem "ilk gelen ilk hizmet alır" düzenine döner; kısa süreçler uzun süreçlerin arkasında bekler. q çok küçük seçilirse süreçler arasında çok sık geçiş yapılır. Her geçişin (bağlam değiştirme) kendi maliyeti olduğu için işlemci zamanının önemli bir kısmı asıl işe değil, geçişlere harcanır. Pratikte q, süreçlerin çoğunun bir dilimde etkileşimli bir işi bitirebileceği kadar uzun seçilir.

Örnekteki süreçlerin bekleme sürelerini de hesaplayalım. Bekleme süresi, tamamlanma zamanından çalışma süresinin çıkarılmasıyla bulunur: P1 için $10 - 5 = 5$, P2 için $9 - 3 = 6$, P3 için $6 - 2 = 4$. Ortalama bekleme $15/3 = 5$ birimdir. Aynı süreçler sırayla, kesintisiz çalıştırılsaydı (P1, sonra P2, sonra P3) tamamlanma zamanları 5, 8 ve 10, bekleme süreleri 0, 5 ve 8 olurdu; ortalama $13/3 \approx 4{,}33$. Round robin ortalama beklemeyi her zaman azaltmaz. Kazandırdığı şey, hiçbir sürecin ilk kez çalışmak için uzun süre beklememesidir: üç süreç de ilk 6 birim içinde işlemciye ulaşmıştır.

8.5.2 Kod izleme: roundRobin()

Round robin çizelgeleyicinin kodu kısadır. Süreçler aşağıdaki küçük sınıfla temsil edilir:

```
class Surec {
    String ad;   // süreç adı (P1, P2, …)
    int kalan;   // kalan çalışma süresi
    Surec(String ad, int kalan) { this.ad = ad; this.kalan = kalan; }
}
```

```
void roundRobin(Queue<Surec> q, int dilim) {
  int t = 0;
  while (!q.isEmpty()) {
    Surec p = q.poll();
    int calis = Math.min(dilim, p.kalan);
    t += calis;
    p.kalan -= calis;
    if (p.kalan > 0) q.offer(p);
    else System.out.println(p.ad + " t=" + t);
  }
}
```

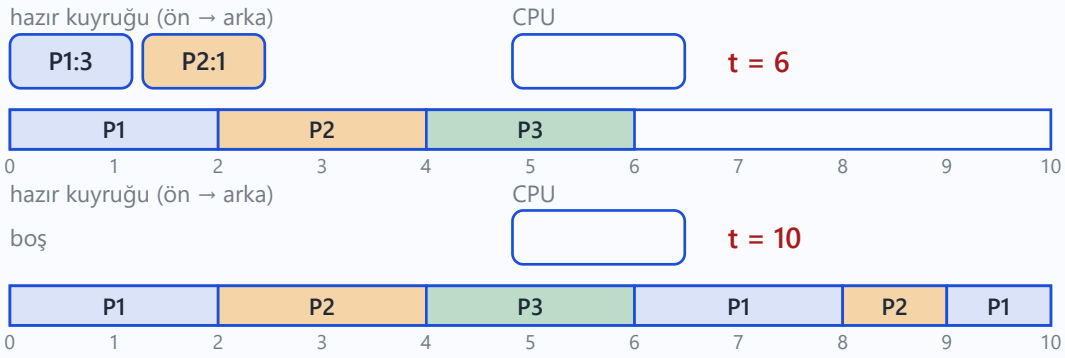
`t` o ana kadar geçen süredir. Döngü kuyruk boşalana kadar döner. Her turda öndeki süreç `poll()` ile alınır. Çalışma süresi `calis = Math.min(dilim, p.kalan)` 'dır: süreç dilimden önce bitecekse yalnız kalan süresi kadar çalışır. `t` bu kadar ilerler ve sürecin kalan süresi bu kadar azalır. Kalan süre sıfırdan büyükse süreç `offer(p)` ile kuyruğun arkasına döner; değilse tamamlanma zamanı yazdırılır.

Tur	`p`	`calis`	`t`	`p.kalan`	Sonuç	Kuyruk (sonra)
(başlangıç)	—	—	0	—	—	P1:5, P2:3, P3:2
1	P1	min(2, 5) = 2	2	3	arkaya döner	P2:3, P3:2, P1:3
2	P2	min(2, 3) = 2	4	1	arkaya döner	P3:2, P1:3, P2:1
3	P3	min(2, 2) = 2	6	0	“P3 t=6”	P1:3, P2:1
4	P1	min(2, 3) = 2	8	1	arkaya döner	P2:1, P1:1
5	P2	min(2, 1) = 1	9	0	“P2 t=9”	P1:1
6	P1	min(2, 1) = 1	10	0	“P1 t=10”	boş

Tablo 8.20: `roundRobin(q, 2)` izlemesi (P1 · 5, P2 · 3, P3 · 2). Kuyruk önden arkaya, “ad:kalan” biçiminde yazılmıştır.

Kod, örnekteki elle yapılan hesabın aynısını yürütür. 5. ve 6. turlarda `calis` 1'dir: süreçlerin kalan süresi dilimden kısadır ve dilimin tamamı kullanılmaz. Kuyruk boşalınca döngü biter. Çıktı sırası “P3 t=6”, “P2 t=9”, “P1 t=10”dur.

Her tur bir `poll` ve en çok bir `offer` yapar; bir turun maliyeti *O(1)*'dir. Toplam tur sayısı, her sürecin kaç dilime bölündüğüne bağlıdır: süresi *s* olan süreç *⌈s/q⌉* tur alır. Örnekte bu *⌈5/2⌉ + ⌈3/2⌉ + ⌈2/2⌉ = 3 + 2 + 1 = 6* turdur.



Şekil 8.22: `roundRobin` izlemesinin iki anı. (a) 3. turun sonu: P3 bitmiştir ($t = 6$), kuyrukta P1 ve P2 kalmıştır. (b) Döngünün sonu: kuyruk boştur, zaman çizelgesi P1, P2, P3, P1, P2, P1 dilimlerinden oluşur.

Sık yapılan hata

Kalan süre dilimden kısayken de `t` 'yi dilim kadar artırmak (`t += dilim`). Bu durumda her tur 2 birim sayılır: P3 yine 6'da biter, ama P2 9 yerine 10'da, P1 de 10 yerine 12'de bitmiş görünür. İşlemcinin boşta geçirdiği süre de çalışma süresine eklenmiş olur. Süreç kalan süresi kadar çalışır; artış `Math.min(dilim, p.kalan)` kadardır.

8.5.3 Josephus problemi

Josephus problemi eski bir bilmeceye dayanır. *n* kişi bir halka oluşturur. Belirli bir kişiden başlanarak sayılır ve her *k*. kişi halkadan çıkarılır. Sayma, çıkarılanın yanındaki kişiden devam eder. Halkada tek kişi kalana kadar sürer. Soru, kimin kalacağıdır.

Kuyruk, halkayı döndürmenin en doğal yoludur. Kişiler sırayla kuyruğa konur. Öndeki kişiyi alıp arkaya eklemek, sayıyı bir ilerletmek demektir. Her turda öndeki $k - 1$ kişi arkaya geçirilir; ardından öndeki, yani *k*. kişi çıkarılır ve elenir.

```
String josephus(Queue<String> q, int k) {
  while (q.size() > 1) {
    for (int i = 1; i < k; i++)
      q.offer(q.poll());      // öndekini arkaya
    String elenen = q.poll();
    System.out.println(elenen + " elendi");
  }
  return q.poll();
}
```

`for` döngüsü `i = 1` 'den `k - 1` 'e kadar döner ve her adımda `q.offer(q.poll())` çalıştırır: öndeki kişi alınır ve hemen arkaya eklenir. Döngü bitince öndeki kişi `poll()` ile çıkarılır, adı yazdırılır ve bir daha kuyruğa girmez. Dış `while` kuyrukta birden çok kişi olduğu sürece döner. En sonda kalan kişi

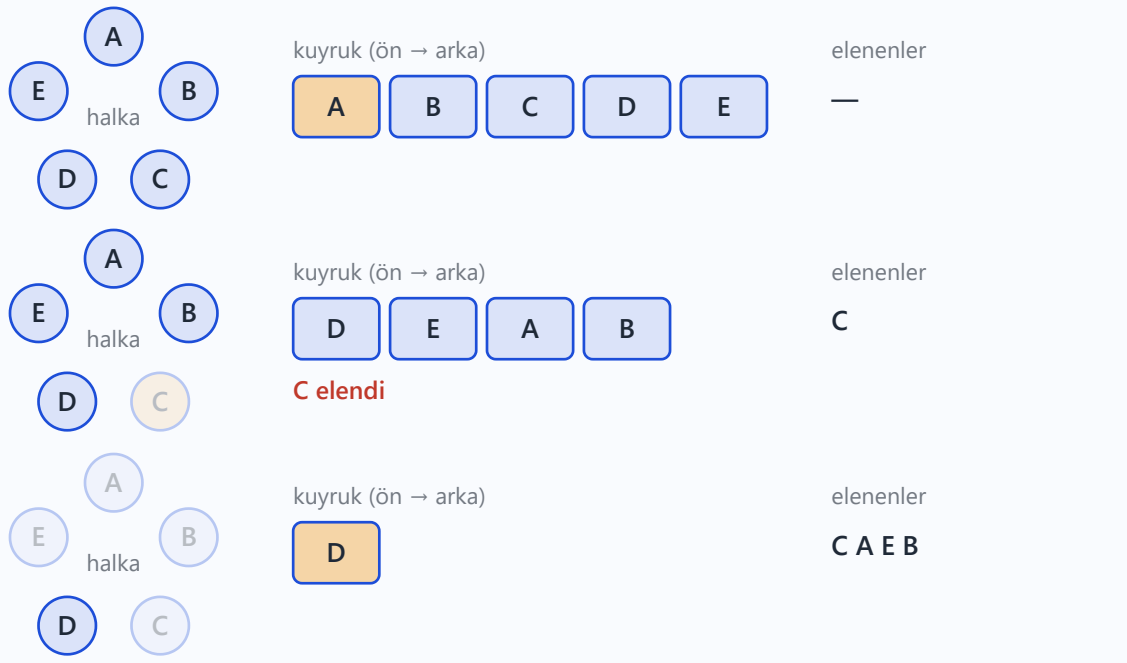
döndürülür.

n = 5 kişi (A, B, C, D, E) ve *k = 3* için izleyelim.

Tur	Arkaya geçenler	Kuyruk (eleme öncesi)	Elenen	Kuyruk (sonra)
(başlangıç)	—	—	—	A, B, C, D, E
1	A, B	C, D, E, A, B	C	D, E, A, B
2	D, E	A, B, D, E	A	B, D, E
3	B, D	E, B, D	E	B, D
4	B, D	B, D	B	D

Tablo 8.21: `josephus(q, 3)` izlemesi (*n = 5*). Kuyruk önden arkaya yazılmıştır.

Eleme sırası C, A, E, B'dir ve kazanan D'dir. Dördüncü tur dikkat çekicidir: kuyrukta iki kişi vardır ve her ikisi de arkaya geçer. B ve D bir tam tur atar, sayma yeniden B'ye gelir ve B elenir. Halka küçüldükçe sayma halkayı birden çok kez dolaşabilir; kuyruk bunu hiçbir özel durum yazmadan doğru yapar.



Şekil 8.23: Josephus izlemesinin üç anı; solda halka, ortada kuyruk, sağda elenenler. (a) Başlangıç: A, B, C, D, E. (b) İlk tur: A ve B arkaya geçmiş, C elenmiştir; kuyruk D, E, A, B. (c) Son durum: elenenler C, A, E, B; kazanan D.

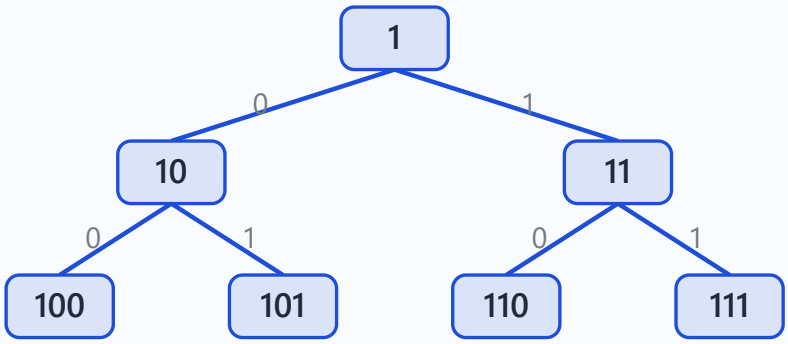
Maliyet hesabı kolaydır. *n* kişiden *n − 1*'i elenir; her eleme turu *k − 1* taşıma ve bir çıkarma, yani *k* kuyruk işlemi yapar. Toplam *(n − 1) · k* işlem, yani *O(n·k)*'dir. Örnekte *4 · 3 = 12* kuyruk işlemi yapılmıştır.

Yalnız kazananı bulmak isteniyorsa kuyruk kullanmadan da bir yineleme bağıntısı vardır. Kişiler 0'dan numaralanırsa *J(1) = 0* ve *J(n) = (J(n − 1) + k) % n*'dir. *k = 3* için *J(2) = 1, J(3) = 1, J(4) = 0, J(5) = 3* bulunur. 3 numaralı kişi dördüncü kişi, yani D'dir. Bu yöntem *O(n)*'de çalışır, ama eleme sırasını vermez. Kuyruk çözümü ise süreci adım adım canlandırır.

8.5.4 1'den n'e kadar ikilik sayılar

Problem şudur: 1'den *n*'e kadar sayıların ikilik (*binary*) gösterimlerini artan sırada üretmek. Örneğin *n = 3* için sonuç {"1", "10", "11"}, *n = 5* için {"1", "10", "11", "100", "101"}'dir.

Her sayıyı ayrı ayrı ikiliğe çevirmek mümkündür. Kuyruk ise daha zarif bir yol sunar. Bir *x* sayısının ikilik gösteriminin sonuna "0" eklemek sayıyı *2x*'e, "1" eklemek *2x + 1*'e çevirir. Örneğin "10" (2) dizgisinden "100" (4) ve "101" (5) elde edilir. Bu ilişki sayıları bir ağaç gibi düzenler: kök "1"'dir ve her düğümün iki çocuğu, sonuna "0" ve "1" eklenmiş hâlleridir.



Şekil 8.24: İkilik sayı ağacı. 1'in çocukları 10 ve 11'dir; 10'un çocukları 100 ve 101, 11'in çocukları 110 ve 111'dir. Ağaç düzey düzey, soldan sağa okunursa sayılar artan sırada çıkar: 1, 10, 11, 100, 101, 110, 111.

Ağacı düzey düzey gezmenin yolu kuyruktur. Kuyruktan bir dizgi alınır ve sonuca yazılır; ardından onun iki çocuğu kuyruğun arkasına eklenir. Önce eklenen önce çıktığı için bir düzeydeki bütün dizgiler, bir alt düzeydekilerden önce işlenir. Aynı düzeyde de soldan sağa sıra korunur. Bu gezinme biçimine **genişlik öncelikli gezinme** denir. Aynı düzeydeki ikilik sayılar aynı uzunluktadır ve soldan sağa artar; daha alt düzeydekiler daha uzundur, dolayısıyla daha büyüktür. Sonuç olarak sayılar tam olarak artan sırada üretilir.

```
String[] ikilikSayiUret(int n) {
    String[] sonuc = new String[n];
    Queue<String> q = new LinkedList<>();
    q.offer("1");
    for (int i = 0; i < n; i++) {
        sonuc[i] = q.poll();
        q.offer(sonuc[i] + "0");
        q.offer(sonuc[i] + "1");
    }
    return sonuc;
}
```

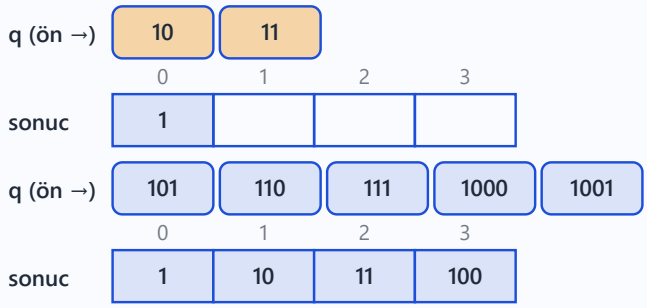
8.5.5 Kod izleme: ikilikSayiUret(4)

Metot *n* yuvalı bir sonuç dizisi ve boş bir kuyruk oluşturur. Kuyruk `LinkedList` ile gerçeklenir, ama değişkenin türü `Queue` olduğu için yalnız kuyruk işlemleri kullanılır. Kök "1" kuyruğa eklenir. Döngü *n* kez döner. Her turda öndeki dizgi çıkarılıp sonuca yazılır ve iki çocuğu arkaya eklenir.

`i`	`poll()`	Arkaya eklenenler	Kuyruk (sonra)	`sonuc`
(başlangıç)	—	"1"	1	[␣, ␣, ␣, ␣]
0	"1"	"10", "11"	10, 11	[1, ␣, ␣, ␣]
1	"10"	"100", "101"	11, 100, 101	[1, 10, ␣, ␣]
2	"11"	"110", "111"	100, 101, 110, 111	[1, 10, 11, ␣]
3	"100"	"1000", "1001"	101, 110, 111, 1000, 1001	[1, 10, 11, 100]

Tablo 8.22: `ikilikSayiUret(4)` izlemesi. Kuyruk önden arkaya yazılmıştır.

Her turda kuyruğun önündeki dizgi, kuyrukta bekleyenlerin en küçüğüdür. `i = 4`'te *4 < 4* yanlıştır ve döngü biter. Metot {"1", "10", "11", "100"} döndürür. Kuyrukta kalan beş dizgi hiç kullanılmaz. Bu küçük bir israftır: son turlarda eklenen çocuklar gerekmez. İstenirse yalnız gerekli sayıda dizgi üretmek için ekleme bir sayaçla sınırlanabilir.



Şekil 8.25: `ikilikSayiUret(4)` izlemesinin iki anı. (a) İlk turun sonu: "1" sonuca yazılmış, çocukları "10" ve "11" kuyruğa eklenmiştir. (b) Döngünün sonu: `sonuc` = {1, 10, 11, 100}; kuyrukta 101, 110, 111, 1000, 1001 kalmıştır.

Maliyet iki düzeyde verilir. Kuyruk işlemi olarak sayılırsa *n* `poll` ve *2n* `offer` yapılır: *O(n)* işlem. Ancak her işlem bir dizgi oluşturur ve *n*'ye kadarki bir sayının ikilik gösterimi yaklaşık $\log_2 n$ karakterdir. Dizgi birleştirme karakter sayısıyla orantılı sürdüğü için karakter düzeyinde toplam iş *O(n log n)*'dir. Kuyrukta en çok yaklaşık *n + 1* dizgi bulunur.

8.5.6 İki yığınla kuyruk

Son problem bir tasarım sorusudur: elimizde yalnız yığın varsa kuyruk kurulabilir mi? Tek yığın yetmez, çünkü yığın sırayı ters çevirir. Ama sırayı iki kez ters çevirmek onu eski hâline getirir. Bu gözlem iki yığınlı çözümün temelidir.

Bir yığın (`gelen`) eklemeler için, öteki (`giden`) çıkarmalar için kullanılır. `enqueue` öğeyi doğrudan `gelen`'e push eder. `dequeue` önce `giden`'e bakar. `giden` boşsa `gelen`'deki bütün öğeler tek tek pop edilip `giden`'e push edilir. Bu aktarım sırayı ters çevirir ve en eski öğe `giden`'in üstüne çıkar. Ardından `giden`'den pop yapılır.

```
Deque<Integer> gelen = new ArrayDeque<>();
Deque<Integer> giden = new ArrayDeque<>();
void enqueue(int x) {
  gelen.push(x);
}
int dequeue() {
  if (giden.isEmpty())
    while (!gelen.isEmpty())
      giden.push(gelen.pop());
  return giden.pop();
}
```

Kodda yığın olarak `ArrayDeque` kullanılmıştır; `push` ve `pop` baş uçta çalışır. Boş kuyrukta `dequeue` çağrılırsa iki yığın da boştur, aktarım hiçbir şey taşımaz ve `giden.pop()` `NoSuchElementException` fırlatır. Yani alt taşma istisnayla bildirilir.

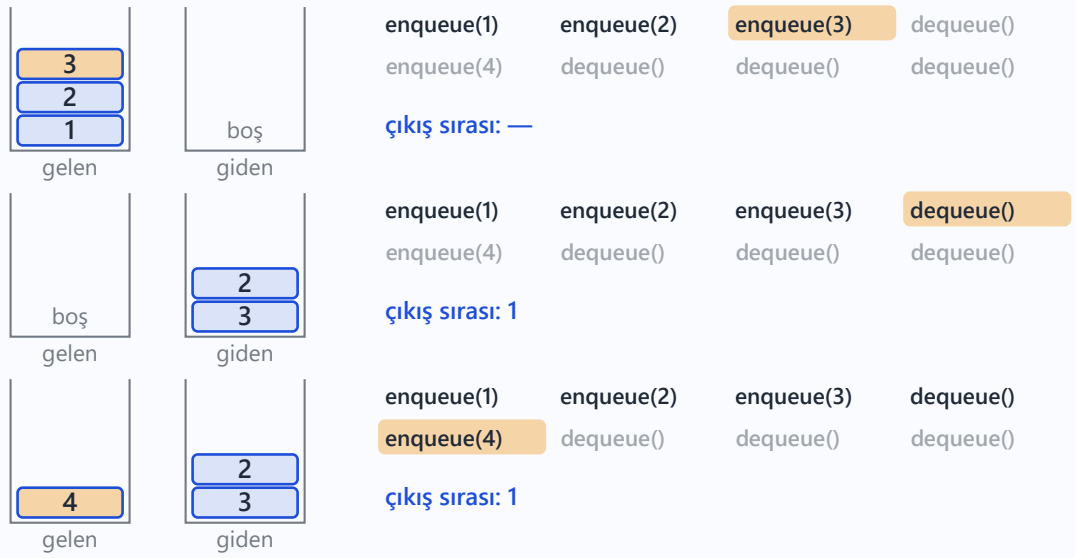
Kuyruğa sırayla `enqueue(1)`, `enqueue(2)`, `enqueue(3)`, `dequeue()`, `enqueue(4)` ve üç kez `dequeue()` uygulayalım. Yığınlar alttan üste yazılmıştır.

İşlem	Aktarım	`gelen` (sonra)	`giden` (sonra)	Döner
`enqueue(1)`	—	1	boş	—
`enqueue(2)`	—	1, 2	boş	—
`enqueue(3)`	—	1, 2, 3	boş	—
`dequeue()`	3, 2, 1 taşınır (3 öğe)	boş	3, 2	1
`enqueue(4)`	—	4	3, 2	—
`dequeue()`	yok (`giden` boş değil)	4	3	2
`dequeue()`	yok	4	boş	3
`dequeue()`	4 taşınır (1 öğe)	boş	boş	4

Tablo 8.23: İki yığınla kuyruğun izlemesi. Yığınlar alttan üste yazılmıştır.

İlk `dequeue`'da `giden` boştur. `gelen`'in üstündeki 3 önce taşınır ve `giden`'in dibine yerleşir; ardından 2, en son 1 taşınır. 1, `giden`'in üstündedir ve pop edilir. FIFO kuralı korunmuştur: ilk eklenen ilk çıkmıştır.

`enqueue(4)`, `giden`'de bekleyen 2 ve 3'e dokunmaz; 4 `gelen`'e girer. Sonraki iki `dequeue`'da `giden` boş değildir ve aktarım yapılmaz; 2 ve 3 sırayla çıkar. 4'ün `gelen`'de beklemesi doğrudur, çünkü 2 ve 3'ten sonra eklenmiştir. Son `dequeue`'da `giden` boşalmıştır ve 4 aktarılıp çıkarılır. Çıkış sırası 1, 2, 3, 4'tür; giriş sırasıyla aynıdır. Toplam 4 öğe aktarılmıştır.



Şekil 8.26: İki yığınla kuyruğun üç anı. (a) Üç `enqueue`'dan sonra `gelen` alttan üste 1, 2, 3'tür. (b) İlk `dequeue`: aktarım sırayı ters çevirmiş, 1 pop edilmiş, `giden`'de 3, 2 kalmıştır. (c) `enqueue(4)`: 4 `gelen`'e girer, `giden`'e dokunulmaz.

Tek bir `dequeue` çağrısı pahalı olabilir: `giden` boşken `gelen`'de n öğe varsa aktarım n pop ve n push yapar. En kötü durumda tek çağrı $O(n)$'dir. Ama böyle pahalı çağrılar sık olamaz.

Teorem 8.3 (İki yığınla kuyruğun amortize maliyeti)

Boş kuyruktan başlayan, m işlemden oluşan herhangi bir `enqueue` / `dequeue` dizisinin toplam maliyeti $O(m)$'dir. Dolayısıyla iki işlemin amortize maliyeti $O(1)$'dir.

İspat. Bir öğenin kuyruktaki ömrü boyunca başına gelenleri sayalım. Öğe `enqueue`'da `gelen`'e bir kez push edilir. Bir aktarımda `gelen`'den bir kez pop edilir ve `giden`'e bir kez push edilir. Bir öğe `giden`'e geçtikten sonra bir daha `gelen`'e dönmez; bu yüzden en çok bir kez aktarılır. Son olarak `dequeue`'da `giden`'den bir kez pop edilir. Toplam en çok 4 yığın işlemi eder. m işlemlik dizide en çok m öğe eklenebileceği için toplam yığın işlemi $4m$'yi geçmez. □

Tablo 8.23'teki 8 işlemlik dizi bu sınırı somut olarak gösterir. 4 push (`gelen`'e), 4 aktarım (her biri bir pop ve bir push) ve 4 pop (`giden`'den) yapılmıştır: toplam $4 + 8 + 4 = 16$ yığın işlemi. Bu, öğe başına en çok 4 işlemden gelen $4 \cdot 4 = 16$ sınırına eşittir, çünkü her öğe tam bir kez aktarılmıştır.

Sık yapılan hata

Her `dequeue`'da aktarım yapmak, yani `giden` boş değilken de `gelen`'deki öğeleri `giden`'e taşımak. Bu hem yavaştır hem de yanlıştır. Örnekte `enqueue(4)`'ten sonra 4 hemen aktarılsaydı `giden`'in üstüne çıkardı ve 2'den önce dönerdi. Aktarım yalnız `giden` boşken yapılır.

Tersi soru da sorulabilir: yalnız kuyruk kullanarak yığın kurulabilir mi? Tek bir kuyrukla mümkündür. `push(x)` önce `x`'i kuyruğa ekler, sonra ondan önceki bütün öğeleri birer birer önden alıp arkaya ekler. Böylece `x` öne gelir ve `pop` doğrudan önden çıkarabilir. Bu tasarımda `pop` *O(1)*, `push` ise *O(n)*'dir; maliyet bu kez amortize olarak da düşmez, çünkü her `push` bütün kuyruğu döndürür.

8.6 Sık yapılan hatalar

Kuyruk kodundaki hataların çoğu birkaç tipe ayrılır. Not boyunca konuların yanında verilen hatalar burada bir arada toplanmıştır. Bir kuyruk kodunu gözden geçirirken kontrol listesi olarak kullanılabilir.

Hata	Ne olur?	Doğrusu
Son öğede `arka`'yı unutmak	Bağlı liste kuyruğunda son düğüm çıkınca `on = null` olur, ama `arka` silinmiş düğümü göstermeye devam eder.	`if (on == null) arka = null;`
Dolu / boş karışıklığı	Yalnız `bas` ve `arka` tutulursa boşta da dolu kuyrukta da `bas == arka` olur.	Bir `boyut` sayacı ya da bir boş yuva.
Mod'u unutmak	Başa sarmış kuyrukta indis dizinin dışına taşar: `ArrayIndexOutOfBoundsException`.	`(bas + boyut) % N`; geri giderken `(bas - 1 + N) % N`.
Büyütmede `Arrays.copyOf`	Başa sarmış dairesel kuyrukta mantıksal sıra bozulur, öğeler kaybolur.	Öğeler `(bas + k) % N` sırasıyla kopyalanır, `bas = 0` yapılır.
`add` / `offer`, `remove` / `poll` karışıklığı	`add`, `remove`, `element` hata fırlatır; `offer`, `poll`, `peek` `false` ya da `null` döner.	Boş kuyruk beklenen bir durumsa `poll` / `peek` ve `null` denetimi.
`PriorityQueue`'yu FIFO sanmak	`Queue<Integer> q = new PriorityQueue<>()` öğeleri önceliğe göre verir, geliş sırasına göre değil.	FIFO için `ArrayDeque` ya da `LinkedList`.
`ArrayDeque` ve `null`	`ArrayDeque` `null` öğe kabul etmez (`NullPointerException`); `null`, `poll`'un "boş" sonucuyla karışırdı.	Kuyruğa `null` konmaz; gerekiyorsa ayrı bir "boş" nesnesi kullanılır.

Tablo 8.24: Kuyruk kodunda sık görülen hatalar.

Son satırdaki kısıt bir tasarım kararıdır. `poll()` boş kuyrukta `null` döndürür. Kuyrukta `null` öğeler de bulunabilseydi, `null` dönüşünün "kuyruk boş" mu yoksa "öndeki öğe `null`" mı anlamına geldiği ayırt edilemezdi. `ArrayDeque` bu belirsizliği `null` eklemeyi yasaklayarak önler. `LinkedList` ise `null` kabul eder; bu yüzden onunla çalışırken `poll` sonucunu yorumlamak daha dikkat ister.

Bölümdeki problemlere geri dönüp bakıldığında ortak bir desen görülür. Kuyruk şu soru sorulduğunda doğru araçtır: **"Bekleyenler arasında en uzun süredir bekleyen hangisi?"** Round robin'de bu, işlemciyi en uzun süredir bekleyen süreçtir. Josephus probleminde sayma sırası gelen kişidir. İkilik sayı üretiminde, henüz çocukları üretilmemiş en küçük sayıdır. Yığının sorusu ise bunun tersidir: "en son görülen ve henüz sonuçlanmamış öğe hangisi?" Bir problemde öğelerin geliş sırasıyla işlenmesi gerekiyorsa çözüm büyük olasılıkla bir kuyruk kullanır. İki uçla da çalışmak gerekiyorsa (kayan pencerede olduğu gibi) çift uçlu kuyruk gerekir.

8.7 Özet

- **Kuyruk**, eklemenin arkadan, çıkarmanın önden yapıldığı bir ADT'dir ve **FIFO** (ilk giren ilk çıkar) düzeniyle çalışır. Temel işlemleri `enqueue`, `dequeue`, `first` / `peek`, `size` ve `isEmpty`'dir; iyi bir gerçeklemede hepsi *O(1)*'dir.
- Boş kuyruktan çıkarma **alt taşma**, sabit kapasiteli dolu kuyruğa ekleme **taşmadır**. Java'daki `Queue` arayüzünde `add` / `remove` / `element` istisna fırlatır, `offer` / `poll` / `peek` `false` ya da `null` döndürür. `ArrayDeque` dairesel diziyle, `LinkedList` bağlı listeyle gerçeklenir.
- **Basit dizi kuyruğunda** öndeki boşalan yuvalar yeniden kullanılamaz; kuyruk içi boşken bile "dolu" olur. Kaydırma bu sorunu çözer ama çıkarmayı *O(n)* yapar.
- **Dairesel kuyruk** indisleri *(i + 1) % N* ile ilerletir ve yuvaları yeniden kullanır. Sayaç yönteminde ekleme yeri `(bas + boyut) % N`, boş koşulu `boyut == 0`, dolu koşulu `boyut == N`'dir. "Bir yuva boş" yönteminde en fazla *N − 1* öğe saklanır ve dolu koşulu `(arka + 1) % N == bas`'tır.
- Dolu dairesel kuyruk, öğeler `(bas + k) % N` sırasıyla yeni diziye kopyalanarak iki katına **büyütülür**. Tek büyütme *O(n)*'dir, ama *n* eklemenin toplamı *3n*'den az olduğu için ekleme amortize *O(1)*'dir. `Arrays.copyOf` başa sarmış kuyruğun sırasını bozar.
- **Bağlı liste kuyruğu** `on` ve `arka` referanslarıyla çalışır: ekleme sona, çıkarma baştandır ve ikisi de kesin *O(1)*'dir. Boş kuyruğa ilk eklemede `on`, son öğe çıkınca `arka` ayrıca güncellenmelidir.
- **Çift uçlu kuyruk** (deque) iki uçtan da ekleme ve çıkarma yapar; yalnız bir uç kullanılırsa yığın, bir uçtan eklenip öbüründen çıkarılırsa kuyruk olur. Çift yönlü bağlı listede `onceki` referansı sondan çıkarmayı *O(1)* yapar; son öğe çıkınca iki uç da `null` olmalıdır.
- **Kayan pencerede en büyük** problemi, değerleri önden arkaya azalan bir indis kuyruğuyla *O(n)*'de çözülür; her indis bir kez girer ve en çok bir kez çıkar.
- **Round robin** çizelgeleme, her süreci en fazla *q* birim çalıştırıp bitmeyeni kuyruğun arkasına döndürür. **Josephus** problemi, *k − 1* kişiyi arkaya geçirip *k*.'yi eleyerek *O(n·k)*'de çözülür. **İkilik sayılar**, "x0" ve "x1" çocuklarını kuyruğa ekleyerek artan sırada üretilir.
- **İki yığınla kuyruk**, eklemeleri bir yığına yapar ve öbür yığın boşaldığında öğeleri aktarır. Her öğe en çok bir kez aktarıldığı için işlemler amortize *O(1)*'dir; tek bir `dequeue` en kötü durumda *O(n)*'dir.

Gerçekleme	Ekleme	Çıkarma	Ön öğe	Not
Basit dizi (kaydırmalı)	O(1)	O(n)	O(1)	her çıkarmada kaydırma
Dairesel dizi	O(1)	O(1)	O(1)	sabit kapasite, % N
Bağlı liste	O(1)	O(1)	O(1)	`on` + `arka` referansı
Çift uçlu kuyruk (çift yönlü liste)	O(1) iki uç	O(1) iki uç	O(1)	`onceki` referansı
İki yığınla kuyruk	O(1)	O(1) amortize	O(1) amortize	her öğe bir kez aktarılır

Tablo 8.25: Kuyruk gerçeklemelerinin maliyetleri.

8.8 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

kuyruk *queue* · ilk giren ilk çıkar *FIFO* · ön *front* · arka *rear* · enqueue (arkaya ekleme) *enqueue* · dequeue (önden çıkarma) *dequeue* · soyut veri tipi *abstract data type (ADT)* · arayüz *interface* · jenerik *generic* · taşma *overflow* · alt taşma *underflow* · dizi *array* · kapasite *capacity* · dairesel kuyruk *circular queue* · amortize analiz *amortized analysis* · tek yönlü bağlı liste *singly linked list* · çift yönlü bağlı liste *doubly linked list* · düğüm *node* · çift uçlu kuyruk *deque (double-ended queue)* · öncelikli kuyruk *priority queue* · kayan pencere *sliding window* · çizelgeleyici *scheduler* · round robin (sırayla dönen çizelgeleme) *round robin* · zaman dilimi *time slice (quantum)* · genişlik öncelikli arama *breadth-first search (BFS)* · yığın *stack* · zaman karmaşıklığı *time complexity* · alan karmaşıklığı *space complexity*

8.9 Kendini Dene

1. Boş bir kuyruğa sırayla `enqueue(5)`, `enqueue(8)`, `dequeue()`, `enqueue(2)`, `first()`, `enqueue(6)`, `dequeue()`, `dequeue()` uygulanıyor. Değer döndüren çağrılar ne döndürür ve sonda kuyrukta ne kalır?

   Cevap

   İlk `dequeue()` 5'i döndürür; kuyruk: 8. `enqueue(2)`'den sonra kuyruk 8, 2'dir ve `first()` 8'i çıkarmadan döndürür. `enqueue(6)`'dan sonra 8, 2, 6; ardından `dequeue()` 8'i, sonraki `dequeue()` 2'yi döndürür. Dönen değerler sırasıyla 5, 8, 8, 2'dir; sonda kuyrukta yalnız 6 kalır.

2. Sayaç yöntemli bir dairesel kuyrukta $N = 8$, `bas = 5`, `boyut = 6`'dır. Öğeler hangi indislerdedir? Son öğe hangi indistedir ve bir sonraki ekleme nereye yazılır?

   Cevap

   Öğeler önden arkaya $(5 + k) \% 8$ indislerindedir ($k = 0, \ldots, 5$): 5, 6, 7, 0, 1, 2. Son öğe $(5 + 6 - 1) \% 8 = 2$. indistedir. Sonraki ekleme $(5 + 6) \% 8 = 3$. yuvaya yazılır. Kuyruk iki kez daha ekleme kabul eder (3. ve 4. yuvalar), sonra `boyut == 8` olur.

3. Kapasitesi 4 olan, sayaç yöntemli boş bir dairesel kuyruğa `ekle(1)`, `ekle(2)`, `ekle(3)`, `cikar()`, `cikar()`, `ekle(4)`, `ekle(5)`, `ekle(6)` uygulanıyor. Sonda dizi, `bas` ve `boyut` nedir?

Bir `ekle(7)` daha çağrılırsa ne olur?

Cevap

Üç ekleme 0, 1, 2. yuvaları doldurur; iki çıkarma 1 ve 2'yi döndürür ve `bas = 2`, `boyut = 1` olur. 4, *(2 + 1) % 4 = 3*'e; 5, *(2 + 2) % 4 = 0*'a; 6, *(2 + 3) % 4 = 1*'e yazılır. Sonda dizi `[5, 6, 3, 4]`, `bas = 2`, `boyut = 4` 'tür; kuyruk önden arkaya 3, 4, 5, 6'dır ve doludur. `ekle(7)` `IllegalStateException("dolu")` fırlatır.

4. Kapasitesi 3 olan dolu bir dairesel kuyrukta `dizi = [R, P, Q]` ve `bas = 1` 'dir. Büyüyen `ekle(S)` çağrısından sonra dizi, `bas` ve `boyut` ne olur?

Cevap

Mantıksal sıra P, Q, R'dir. `boyut == 3` olduğu için `buyut(6)` çağrılır: `yeni[0] = dizi[1] = P`, `yeni[1] = dizi[2] = Q`, `yeni[2] = dizi[(1 + 2) % 3] = dizi[0] = R`; `bas = 0`. S, *(0 + 3) % 6 = 3*'e yazılır. Sonuç: `[P, Q, R, S, _, _]`, `bas = 0`, `boyut = 4`.

5. Bağlı liste kuyruğunun `dequeue` metodundan `if (on == null) arka = null;` satırı silinmiş olsun. Boş kuyruğa `enqueue(7)`, `dequeue()`, `enqueue(9)` uygulanırsa ara durumlar ve son durum ne olur? Hata hangi koşulda görünür hâle gelir?

Cevap

`enqueue(7)` 'den sonra `on` ve `arka` 7 düğümünü gösterir. `dequeue()` 7'yi döndürür, `on` `null` olur, ama `arka` hâlâ 7 düğümünü gösterir ve `uzunluk = 0` 'dır. `enqueue(9)` 'da `bosMu()` doğrudur; `on` 9'u gösterir ve `arka = gecici` satırı `arka` 'yı da 9'a taşır. Son durum doğrudur. Hata yalnız kuyruk boşken `arka` 'ya bakıldığında (örneğin bir `rear()` çağrısında) görünür: kuyrukta olmayan 7 döner ve 7 düğümü bellekten atılamaz.

6. `Deque<Integer> d = new ArrayDeque<>();` üzerinde sırayla `d.offerLast(1)`, `d.offerFirst(2)`, `d.push(3)`, `d.offer(4)`, `d.pollLast()`, `d.pop()` çağrılıyor. Son iki çağrı ne döndürür ve sonda `d` ne yazdırır?

Cevap

`offerLast(1)` → [1]; `offerFirst(2)` → [2, 1]; `push(3)` başa ekler → [3, 2, 1]; `offer(4)` sona ekler → [3, 2, 1, 4]. `pollLast()` 4'ü döndürür → [3, 2, 1]. `pop()` baştan çıkarır ve 3'ü döndürür → [2, 1]. `System.out.println(d)` `[2, 1]` yazdırır.

7. `pencereMax({2, 7, 3, 1, 5, 4}, 2)` ne döndürür? Çalışma boyunca toplam kaç kez

`pollLast()` çağrılır?

Cevap

Sonuç `[7, 7, 3, 5, 5]`'tir. *i = 1*'de 7, arkadaki 0:2'yi atar (1. `pollLast`). *i = 2*'de 3 eklenir; *i = 3*'te 1:7 önden atılır (pencere dışı), 1 eklenir. *i = 4*'te önden 2:3 atılır; 5, arkadaki 3:1'i atar (2. `pollLast`). *i = 5*'te 4 eklenir. Toplam 2 `pollLast` ve 2 `pollFirst` yapılır; ikisinin toplamı 6 eklemeyi geçmez.

8. Hazır kuyruğunda sırayla P1 (3 birim), P2 (4 birim), P3 (1 birim) var; zaman dilimi $q = 2$'dir. Süreçler round robin ile hangi anlarda tamamlanır? Ortalama bekleme süresi nedir?

Cevap

0–2: P1 (kalan 1), kuyruk P2, P3, P1. 2–4: P2 (kalan 2), kuyruk P3, P1, P2. 4–5: P3 biter. 5–6: P1 biter. 6–8: P2 biter. Tamamlanma: P3 = 5, P1 = 6, P2 = 8. Bekleme süreleri $6 - 3 = 3$, $8 - 4 = 4$, $5 - 1 = 4$; ortalama $11/3 \approx 3{,}67$ birimdir.

9. 1'den 6'ya numaralı altı kişi için `josephus(q, 3)` hangi sırayla eleme yapar ve kimi döndürür? Sonuç $J(n) = (J(n-1) + k) \% n$ bağıntısıyla nasıl denetlenir?

Cevap

Elenenler sırasıyla 3, 6, 4, 2, 5'tir; kazanan 1'dir. Bağıntıyla (0'dan numaralı): $J(1) = 0$, $J(2) = 1$, $J(3) = 1$, $J(4) = 0$, $J(5) = 3$, $J(6) = (3 + 3) \% 6 = 0$. 0 numaralı konum 1. kişidir; iki yöntem aynı sonucu verir. Toplam $5 \cdot 3 = 15$ kuyruk işlemi yapılır.

10. İki yığınla kuyruğa sırayla `enqueue(a)`, `enqueue(b)`, `dequeue()`, `enqueue(c)`, `enqueue(d)`, `dequeue()`, `dequeue()`, `dequeue()` uygulanıyor. Çıkarmalar ne döndürür? Hangi `dequeue`'larda aktarım yapılır ve toplam kaç öğe aktarılır?

Cevap

1. `dequeue`: `giden` boştur; b ve a aktarılır (2 öğe), a döner. `enqueue(c)` ve `enqueue(d)` `gelen`'e girer. 2. `dequeue`: `giden`'de b vardır, aktarım yok; b döner. 3. `dequeue`: `giden` boştur; d ve c aktarılır (2 öğe), c döner. 4. `dequeue`: d döner. Çıkış sırası a, b, c, d'dir; aktarım 1. ve 3. çağrılarda yapılır, toplam 4 öğe aktarılır.

8.10 İleri okuma

- M. T. Goodrich, R. Tamassia ve M. H. Goldwasser, *Data Structures and Algorithms in Java* (6. baskı, 2014), Kesim 6.2 ve 6.3: kuyruk ADT'sini, dairesel dizi ve bağlı liste gerçeklemelerini ve çift uçlu kuyruğu bu nottakine yakın bir Java arayüzüyle anlatır.

- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 1.3: kuyruğu bağlı liste ve büyüyen diziyle gerçekler; yineleyiciler ve jeneriklerle birlikte kısa, açık kodlar verir.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (3. baskı, 2009), Kesim 10.1: kuyruğun dizi gerçeklemesini biçimsel olarak verir; alıştırmalarında iki yığınla kuyruk ve çift uçlu kuyruk yer alır.
- A. Silberschatz, P. B. Galvin ve G. Gagne, *Operating System Concepts* (10. baskı, 2018), Bölüm 5: round robin dahil işlemci çizelgeleme algoritmalarını, zaman diliminin seçimini ve bekleme süresi gibi ölçütleri ayrıntılı işler.

8.11 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3. baskı, MIT Press, 2009.
2. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6. baskı, Wiley, 2014.
3. R. Lafore, *Data Structures and Algorithms in Java*, 2. baskı, Sams, 2002.
4. Oracle, *Java SE 17 API Specification*: `java.util.Queue`, `java.util.Deque`, `java.util.ArrayDeque` ve `java.util.PriorityQueue` belgeleri.
5. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011.
6. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10. baskı, Wiley, 2018.

KISIM III

Ağaçlar

Bölüm 9–11

KISIM III · BÖLÜM 9

Ağaçlar

Diziler, bağlı listeler, yığınlar ve kuyruklar öğeleri tek bir sıra hâlinde tutar: her öğenin en çok bir öncülü ve bir ardılı vardır. Oysa bilgisayarda saklanan verinin önemli bir kısmı sıralı değil, hiyerarşiktir. Klasörlerin içinde klasörler, bir kitabın içinde kısımlar ve bölümler, bir web sayfasının içinde iç içe geçmiş öğeler bulunur. Bu tür ilişkileri doğal olarak tutan yapı ağaçtır. Not, ağacın terimleriyle başlar: kök, ebeveyn, çocuk, yaprak, derece, derinlik ve yükseklik. Ardından her düğümün en çok iki çocuğu olan ikili ağaçlara, bunların dolu, tam ve mükemmel türlerine ve düğüm sayısıyla yükseklik arasındaki bağıntılara geçer. Ağacın bellekte dizi ve bağlantılarla nasıl tutulacağı anlatıldıktan sonra notun ağırlık merkezi gezinmelerdir: kök önde, kök ortada, kök sonda ve düzey sırası. Bu gezinmeler önce özyinelemeyle, sonra açık bir yığın ve kuyrukla kodlanır; her kod adım adım izlenir. Son kesimler ifade ağaçlarını, ağaç üzerinde çalışan özyinelemeli metotları ve iki gezinme çıktısından ağacı yeniden kurmayı işler.

Bu bölümün kodu: `github.com/sercankulcu/data-structures-java/Bolum09/src`

ÖĞRENME ÇIKTILARI

Temel kavramlar
Kök, ebeveyn, çocuk, kardeş, ata, soy, yaprak, derece, derinlik ve yükseklik terimlerini açıklamak ve verilen bir ağaçta bu değerleri bulmak.

Gösterimler
Ağaçları düğüm bağlantılarıyla ve dizilerle temsil etmek; iki gösterimin bellek ve erişim maliyetlerini karşılaştırmak.

Gezinmeler
Kök önde, kök ortada, kök sonda ve düzey sırası gezinmelerini elle ve kodla uygulamak; özyinelemeli çağrıları çağrı yığını üzerinde izlemek.

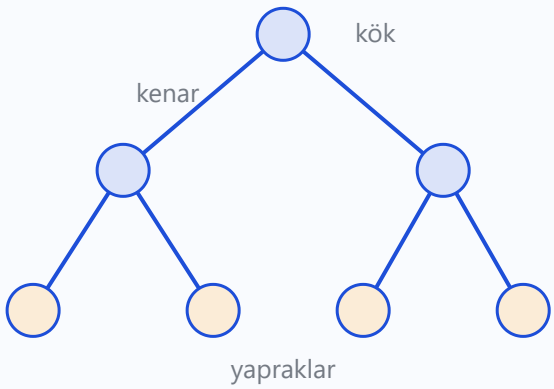
Uygulama ve maliyet
Genel ve ikili ağaçları kullanım alanları ve işlem maliyetleriyle karşılaştırmak; ağaç üzerinde çalışan özyinelemeli metotları yazmak.

9.1 Ağaç kavramı

Şimdiye kadarki veri yapılarının hepsi **doğrusaldır**: öğeler bir çizgi üzerinde dizilir, her öğeden sonra en çok bir öğe gelir. Bu düzen, sıra bilgisinin önemli olduğu işler için uygundur. Ancak bir şirketin örgüt şeması, bir bilgisayarın dosya sistemi ya da bir soy ağacı tek bir çizgiye sığmaz. Bu yapılarda bir öğenin altında birden çok öğe bulunabilir; her biri de kendi altındakilerin üstüdür. Bu kesim, böyle hiyerarşileri tutan ağaç yapısının dilini kurar.

9.1.1 Ağaç nedir?

Ağaç (*tree*), öğeleri hiyerarşik biçimde tutan, doğrusal olmayan bir veri yapısıdır. Öğelerin her birine **düğüm** (*node*), iki düğümü birbirine bağlayan çizgiye **kenar** (*edge*) denir. En üstte tek bir düğüm bulunur; buna **kök** (*root*) adı verilir. Bilgisayar biliminde ağaç ters çizilir: kök yukarıda durur, dallar aşağı doğru iner. Şekil 9.1 küçük bir ağacı bu biçimde gösterir.



Şekil 9.1: Yedi düğümlü bir ağaç. En üstteki düğüm köktür; kök iki çocuğa, onlar da kendi çocuklarına dallanır. En alttaki dört düğümün çocuğu yoktur, bunlar yapraklardır. Yedi düğümü altı kenar bağlar.

Kenarlar bir yön taşır: her kenar bir üst düğümü bir alt düğüme bağlar. Üstteki düğüm alttakinin **ebeveyni** (*parent*), alttaki de üsttekinin **çocuğu**dur (*child*). Ağacı ağaç yapan kural şudur: kök dışındaki her düğümün tam olarak bir ebeveyni vardır. Kökün ise hiç ebeveyni yoktur. Bir düğümün çocuk sayısına bu tanımda bir sınır konmaz; sıfır, bir ya da daha çok çocuk olabilir.

Tanım 9.1

Bir **ağaç**, düğümlerden ve düğümleri ebeveyn–çocuk ilişkisiyle bağlayan kenarlardan oluşan bir yapıdır. Ağaç boş değilse şu iki koşulu sağlar: (1) ebeveyni olmayan tek bir düğüm vardır; bu düğüm **kök**tür. (2) Kök dışındaki her düğümün tam olarak bir ebeveyni vardır ve her düğümden ebeveynden ebeveyne yukarı çıkılarak köke ulaşılır.

Aynı yapı özyinelemeli olarak da tanımlanabilir: ağaç ya boştur ya da bir kök ile kökün altına asılan sıfır ya da daha çok ağaçtan oluşur. Köke asılan bu ağaçlara kökün **alt ağaçları** denir.

Tanımdaki ikinci koşulun sonucu önemlidir: ağaçta **döngü** (*cycle*) bulunamaz. Bir düğümden başlayıp kenarlar üzerinden yürüyerek aynı düğüme, aynı kenarı iki kez kullanmadan dönmek mümkün değildir. Döngü olsaydı, döngüdeki düğümlerden birinin iki ayrı yoldan gelen iki ebeveyni olması ya da yukarı çıkışın köke hiç varmayıp dönüp durması gerekirdi. Her iki durum da tanıma aykırıdır.

Ağaçlarla ilgili en çok kullanılan sayısal gerçek, kenar sayısıdır. Kök dışındaki her düğüm ebeveynine bir kenarla bağlanır ve her kenar tam olarak bir düğümü ebeveynine bağlar. Kenarları, alttaki uçlarındaki düğümlerle eşlersek bire bir bir eşleme elde ederiz.

Teorem 9.1 (Kenar sayısı)

$n \geq 1$ düğümlü her ağaçta tam olarak *$n - 1$* kenar vardır.

İspat. Her kenarı alt ucundaki düğümle, yani çocukla eşleyelim. Bir düğümün tek ebeveyni olduğu için her çocuk en çok bir kenarla eşlenir. Kök hiçbir kenarın alt ucu değildir; kök dışındaki her düğüm ise tam olarak bir kenarın alt ucudur. Böylece kenarlar ile kök dışındaki *$n - 1$* düğüm arasında bire bir bir eşleme kurulur ve kenar sayısı *$n - 1$* olur. □

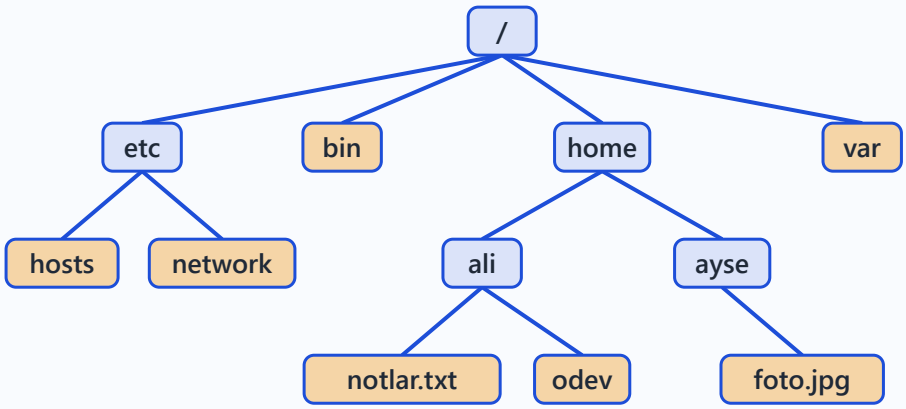
Şekil 9.1 bu kuralı doğrular: yedi düğüm, altı kenar. Kural iki yönde de iş görür. Bir ağacın düğüm sayısı biliniyorsa kenar sayısı hemen bulunur. Bağlantılı bir yapıda kenar sayısı düğüm sayısından küçük değilse, yapı ağaç olamaz; içinde mutlaka bir döngü vardır.

9.1.2 Günlük hayatta hiyerarşiler

Ağaç, bilgisayar bilimine dışarıdan gelmiş bir fikirdir; insanlar hiyerarşileri çok eskiden beri ağaç biçiminde çizer. Birkaç örnek kavramı somutlaştırır.

- **Dosya sistemi.** En üstte kök dizin bulunur (Unix benzeri sistemlerde `/`). Klasörler başka klasör ve dosyaları içerir, dosyalar ise hiçbir şey içermez. Klasörler ağacın iç düğümleri, dosyalar yapraklarıdır.
- **Kitap.** Bir kitap kısımlara, kısımlar bölümlere, bölümler alt başlıklara ayrılır. İçindekiler sayfası bu ağacın girintili bir yazımıdır.

- **Coğrafya.** Kıtalar ülkelerden, ülkeler eyalet ya da illerden, iller de şehirlerden oluşur.
- **Web sayfası.** Bir HTML belgesi iç içe öğelerden oluşur: `<html>` öğesinin içinde `<head>` ve `<body>`, onların içinde başka öğeler bulunur. Tarayıcı belgeyi okurken bu öğelerden bir ağaç kurar. Bu ağaca **belge nesne modeli** (*Document Object Model*, kısaca DOM) denir.

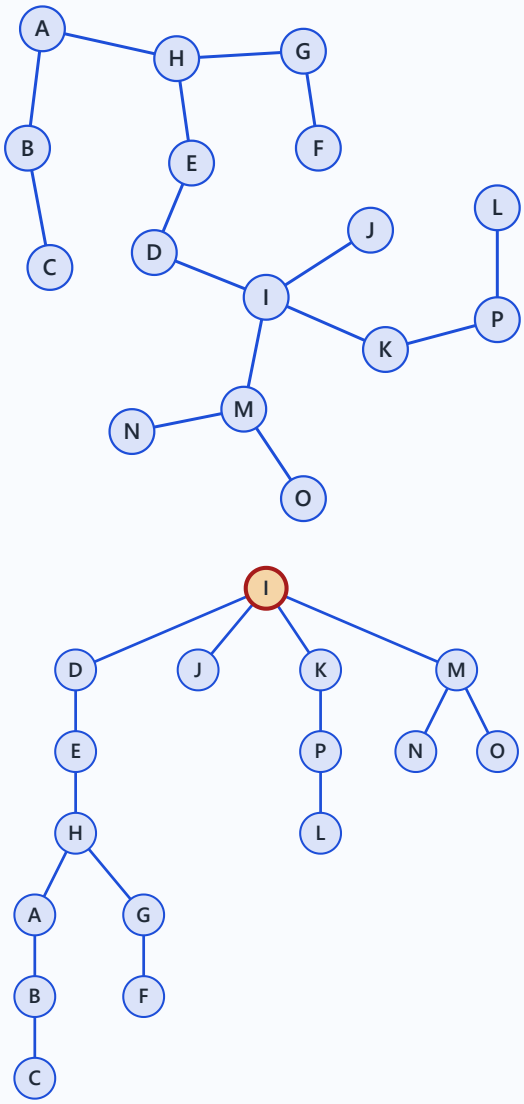


Şekil 9.2: Bir dosya sistemi ağacı. Kök dizin `/` altında `etc`, `bin`, `home` ve `var` bulunur; `home` altında `ali` ve `ayse` klasörleri vardır. Turuncu kutular dosyalardır ve ağacın yapraklarını oluşturur.

Şekil 9.2 bir dosyanın tam yolunun neden tek olduğunu da gösterir. `notlar.txt` dosyasına kökten başlayıp `/home/ali/notlar.txt` yolundan ulaşılır; başka bir yol yoktur. Her düğümün tek ebeveyni olduğundan kökten bir düğüme inen yol da tektir. Kısayol (bağ) dosyaları bu kuralı delen ek bağlantılardır; onlar eklenince yapı artık saf bir ağaç olmaz.

9.1.3 Bir yapı ne zaman ağaçtır?

Kökü işaretlenmemiş, rastgele çizilmiş bir bağlantı yapısının ağaç olup olmadığını anlamak için tanımdaki koşulları yapının kendi özelliklerine çevirmek gerekir. Kök henüz seçilmemişse ebeveyn kavramı da yoktur. Bu durumda ağaç olmak iki koşula bağlıdır. Birincisi, yapı **bağlantılıdır**: her düğümden her düğüme bir yol vardır. İkincisi, yapıda **döngü** yoktur. Bu iki koşulun sonucu olarak kenar sayısı, düğüm sayısından bir eksik çıkar. Bu koşulları sağlayan bir yapıda herhangi bir düğüm kök seçilebilir. Seçilen kökten uzaklaştıkça her düğümün ebeveyni, köke doğru olan tek komşusu olur.



Şekil 9.3: Solda A'dan P'ye 16 düğümlü, kökü belirtilmemiş bir bağlantı yapısı. Sağda aynı yapı, I düğümü kök seçilerek ağaç biçiminde yeniden çizilmiştir: I'nın çocukları D, J, K ve M olur; A, B ve C'ye D–E–H üzerinden inilir.

Örnek 9.1

Şekil 9.3'ün solundaki yapı bir ağaç mıdır?

1. **Düğüm ve kenar sayısı.** Düğümler A'dan P'ye 16 tanedir. Kenarlar sayılınca 15 bulunur: A–B, B–C, A–H, H–G, G–F, H–E, E–D, D–I, I–J, I–K, K–P, P–L, I–M, M–N, M–O.
2. **Bağlantılılık.** I'dan başlanırsa D, J, K, M'ye; D'den E'ye, E'den H'ye, H'den A ve G'ye; A'dan B'ye, B'den C'ye; G'den F'ye; K'den P'ye, P'den L'ye; M'den N ve O'ya ulaşılır. 16 düğümün hepsine varılmıştır.
3. **Sonuç.** Yapı bağlantılıdır ve 16 düğüme 15 kenar düşer; bu nedenle döngü yoktur ve yapı bir ağaçtır. Herhangi bir düğüm, örneğin I, kök seçilebilir. Sağdaki çizim bu seçimin sonucudur.

Örnekte döngünün yokluğunu ayrıca denetlemedik; sayım bunu kendiliğinden verdi. *n* düğümü bağlantılı kılmak için en az *n − 1* kenar gerekir. Bağlantılı bir yapıya bir kenar daha eklemek, zaten arasında yol bulunan iki düğümü ikinci kez bağlamak demektir. Bu da bir döngü oluşturur. Dolayısıyla "bağlantılı ve *n − 1* kenarlı" olmak, "bağlantılı ve döngüsüz" olmakla aynı şeydir.



Şekil 9.4: Ağaç olmayan üç yapı. Solda üç düğüm bir döngü oluşturur (kırmızı kenar döngüyü kapatır). Ortada alttaki düğümün iki ebeveyni vardır. Sağda yapı iki bağlantısız parçaya bölünmüştür.

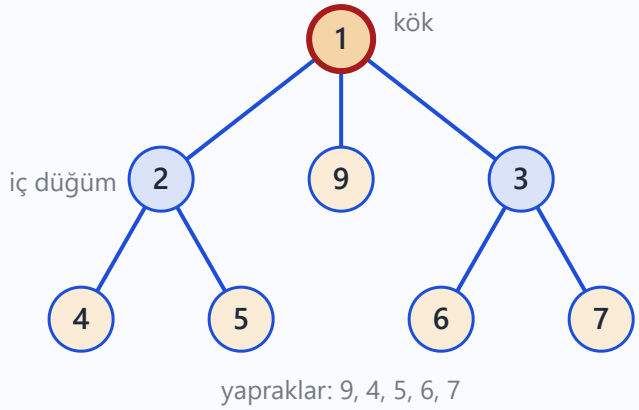
Şekil 9.4 ağaç tanımını bozan üç tipik durumu gösterir. Soldaki üçgende üç düğüme üç kenar düşer; kenar sayısı bir fazladır ve döngü oluşmuştur. Ortadaki yapıda alttaki düğüm iki üst düğüme bağlıdır; hangisi kök seçilirse seçilsin, bu düğüme kökten iki ayrı yolla ulaşılır. Sağdaki yapıda ise üç düğüme bir kenar düşer. Kenar eksik olduğu için yapı iki parçaya ayrılmıştır. Birbirinden kopuk ağaçların oluşturduğu böyle bir topluluğa **orman** (*forest*) denir.

> **Sık yapılan hata**
>
> Yalnız bağlantılılığa bakıp yapıyı ağaç saymak. Bağlantılı, 10 düğümlü ve 10 kenarlı bir yapı ağaç değildir: ağaçta kenar sayısı 9 olmalıdır ve fazladan kenar mutlaka bir döngü kurar. Tersine, kenar sayısının *n − 1* olması da tek başına yetmez; döngülü bir parça ile kopuk bir düğüm birlikte de *n − 1* kenar verebilir. İki koşulun birlikte aranması gerekir: bağlantılılık ve *n − 1* kenar (ya da bağlantılılık ve döngüsüzlük).

9.1.4 Kök, iç düğüm ve yaprak

Ağaçtaki düğümler çocuk sahibi olup olmadıklarına göre iki gruba ayrılır. Bu ayrım, ağaç üzerindeki algoritmaların neredeyse hepsinde karşımıza çıkar; çünkü özyinelemeli bir metot çoğu zaman yapraklarda durur, iç düğümlerde çocuklarına iner.

- **Kök düğüm** (*root*): Ağacın en üstündeki, ebeveyni olmayan düğümdür. Dosya sisteminde kök dizine karşılık gelir.
- **İç düğüm** (*internal node*): En az bir çocuğu olan düğümdür. Dosya sisteminde klasörlere karşılık gelir.
- **Yaprak** (*leaf*): Hiç çocuğu olmayan düğümdür. Ağacın dış kenarında durduğu için **dış düğüm** (*external node*) adıyla da anılır. Dosya sisteminde dosyalara karşılık gelir.



Şekil 9.5: Kökü 1 olan ağaç. 2 ve 3 iç düğümdür; 9, 4, 5, 6 ve 7'nin çocuğu yoktur, bunlar yapraktır. Kökün de çocuğu olduğu için kök aynı zamanda bir iç düğümdür.

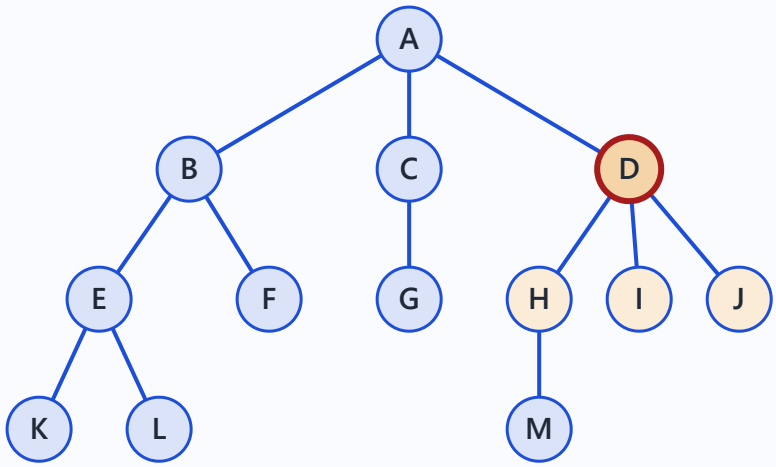
Şekil 9.5 iki ince noktaya dikkat çeker. Birincisi, kök de çocuğu varsa bir iç düğümdür; "kök" ve "iç düğüm" birbirini dışlayan sınıflar değildir. Kök, konumuyla; iç düğüm ve yaprak ise çocuk sayısıyla tanımlanır. İkincisi, yaprakların hepsi aynı düzeyde olmak zorunda değildir: 9, kökün doğrudan çocuğudur ama 4, 5, 6, 7 gibi o da bir yapraktır. Tek düğümlü bir ağaçta ise kökün çocuğu yoktur; bu durumda kök aynı zamanda yapraktır.

Sık yapılan hata

Yaprakları yalnız en alt satırda aramak. Yaprak, çizimde nerede durduğuna göre değil, çocuğu olup olmadığına göre belirlenir. Şekil 9.5'teki 9 düğümü ikinci satırdadır ama çocuğu yoktur, yani yapraktır. Aynı biçimde kökü yaprak saymamak da hatalı olabilir: tek düğümlü ağaçta kök yapraktır.

9.1.5 Düğümler arası ilişkiler

Ağaç terimlerinin çoğu bir aile ağacından alınmıştır. Bir düğümün hemen üstündeki düğüm **ebeveyn**, hemen altındakiler **çocuklar**dır. Aynı ebeveynin çocukları birbirinin **kardeşi**dir (*sibling*). Bir düğümden köke giden yol üzerindeki düğümler o düğümün **ataları**dır (*ancestor*). Bir düğümün altında kalan, yani o düğümden aşağı inerek ulaşılabilen bütün düğümler ise onun **soyu**dur (*descendant*). Tablo 9.1 bu ilişkileri Şekil 9.6'daki ağaç üzerinde örnekler.



Şekil 9.6: İlişkiler için örnek ağaç. A köktür; B, C ve D onun çocuklarıdır. D'nin çocukları H, I ve J vurgulanmıştır. H'nin altında M, B'nin altında E ve F, E'nin altında K ve L, C'nin altında G bulunur.

Terim	Anlamı	Örnek
Ebeveyn	Hemen üstteki düğüm	D, H'nin ebeveynidir.
Çocuk	Hemen alttaki düğüm	H, I ve J, D'nin çocuklarıdır.
Kardeş	Ebeveyni aynı olan düğüm	H, I ve J kardeştir.
Ata	Köke giden yoldaki düğüm	M'nin ataları H, D ve A'dır.
Soy	Alt ağaçtaki düğüm	D'nin soyu H, I, J ve M'dir.

Tablo 9.1: Ağaçtaki ilişkiler ve Şekil 9.6'daki ağaçta örnekleri.

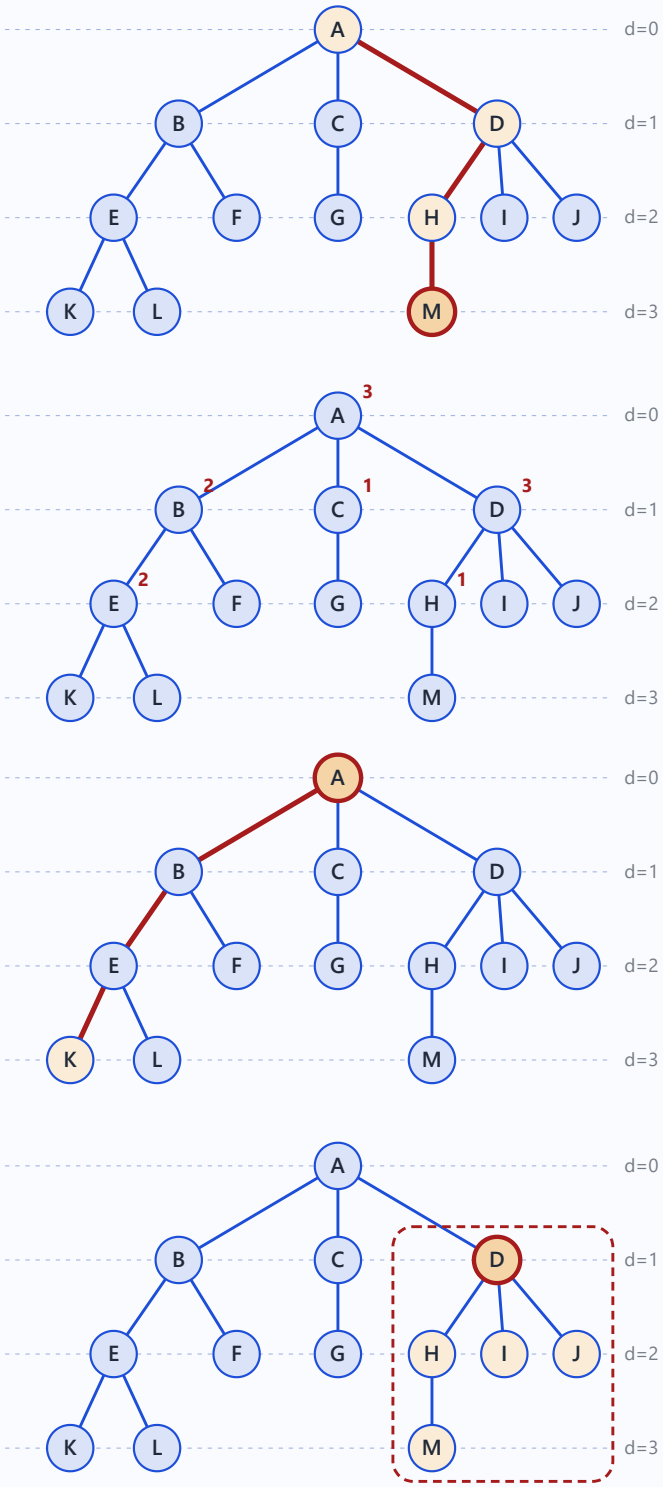
Ata ve soy birbirinin tersidir: M'nin ataları H, D ve A olduğuna göre M de H'nin, D'nin ve A'nın soyundandır. Kök, ağaçtaki bütün düğümlerin atasıdır; yapraklar ise kimsenin atası değildir. Bazı kaynaklar bir düğümü kendi atası ve kendi soyu da sayar. Bu notta, aksi söylenmedikçe, ata ve soy düğümün kendisini içermez. Bir düğümün soyu, o düğümün **alt ağacındaki** diğer düğümlerdir.

Kardeşlik çoğu zaman yanlış anlaşılır. İki düğümün aynı derinlikte, yani çizimde aynı satırda olması onları kardeş yapmaz. Şekil 9.6'daki F ve G aynı satırdadır, ama F'nin ebeveyni B, G'ninki C'dir; bu yüzden kardeş değillerdir. Akrabalık diliyle söylenirse F ile G kuzendir. Kardeşlik yalnız ortak ebeveyn ile tanımlanır.

Sık yapılan hata

Aynı düzeydeki düğümleri kardeş saymak ya da bir düğümün soyunu yalnız çocukları sanmak. Soy, alt ağacın tamamıdır: D'nin soyu yalnız H, I, J değil, H'nin çocuğu M ile birlikte H, I, J, M'dir. Benzer biçimde M'nin ataları yalnız ebeveyni H değil, köke kadar bütün yoldur: H, D, A.

Şekil 9.7 aynı ağacı, terimleri adım adım gösteren sunum çiziminin dört karesiyle verir. Ağaçta 13 düğüm vardır: A, B, C, D, E, F, G, H, I, J, K, L, M. Yapraklar çocuğu olmayan yedi düğümdür: K, L, F, G, M, I, J. İç düğümler en az bir çocuğu olan altı düğümdür: A, B, C, D, E, H. Sayılar tutar: 7 + 6 = 13. Kenar sayısı da Teorem 9.1'e göre 12'dir.



Şekil 9.7: 13 düğümlü ağaçta terimler (soldan sağa, yukarıdan aşağı). (a) M'nin ataları: köke giden yol M–H–D–A. (b) Dereceler: A 3, B 2, C 1, D 3, E 2, H 1; ağacın derecesi 3. (c) Ağacın yüksekliği: kökten en uzak yaprağa A–

B–E–K yolu, 3 kenar. (d) D'nin alt ağacı: D ve soyu H, I, J, M. Kesikli yatay çizgiler derinlikleri (d = 0, 1, 2, 3) gösterir.

9.1.6 Derece, derinlik, düzey ve yükseklik

Ağaçla ilgili hemen her maliyet hesabı birkaç sayısal ölçüye dayanır. Bunların en temeli, bir düğümün ne kadar dallandığını ve kökten ne kadar uzakta olduğunu söyleyen ölçülerdir.

Tanım 9.2

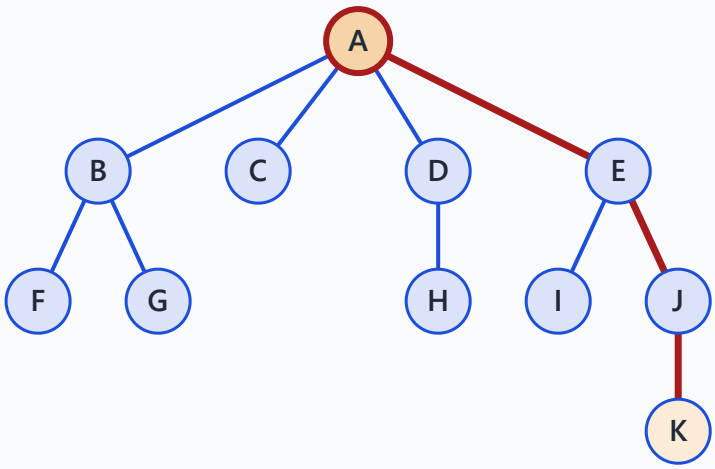
- Bir düğümün **derecesi** (*degree*), çocuklarının sayısıdır; başka bir deyişle düğümün altına asılan alt ağaçların sayısıdır. Ağacın derecesi, düğümlerin derecelerinin en büyüğüdür.
- Bir düğümün **derinliği** (*depth*), kökten o düğüme inen yoldaki kenar sayısıdır. Kökün derinliği 0'dır.
- Aynı derinlikteki düğümlerin kümesine bir **düzey** (*level*) denir. Derinliği d olan düğümler d. düzeyi oluşturur.
- Bir düğümün **yüksekliği** (*height*), o düğümden altındaki en uzak yaprağa inen yoldaki kenar sayısıdır. Yaprağın yüksekliği 0'dır.
- **Ağacın yüksekliği**, kökün yüksekliğidir. Bu değer, düğümlerin derinliklerinin en büyüğüne eşittir.

Derinlik ile yükseklik ters yönlere bakar. Derinlik yukarıya, köke doğru ölçülür: “Bu düğüm kökten ne kadar aşağıda?” Yükseklik aşağıya, yapraklara doğru ölçülür: “Bu düğümün altında ne kadar uzun bir dal var?” Kökün derinliği her zaman 0'dır; yüksekliği ise ağacın yüksekliğidir. Yaprakların yüksekliği her zaman 0'dır; derinlikleri ise yaprağın nerede durduğuna göre değişir.

Ölçü	Tanım	Örnek
Derece	Düğümün çocuk sayısı	D → 3
Derinlik	Kökten düğüme kenar sayısı; kökte 0	E → 2 (A→B→E)
Düzey	Aynı derinlikteki düğümler	Düzey 2: E, F, G, H, I, J
Düğüm yüksekliği	Düğümden en uzak yaprağa kenar sayısı; yaprakta 0	D → 2 (D→H→M)
Ağacın yüksekliği	Kökün yüksekliği = en büyük derinlik	3

Tablo 9.2: Ağaç ölçüleri ve Şekil 9.7'deki 13 düğümlü ağaçta değerleri.

Bu notta derinlik ve yükseklik **kenar sayısıyla** ölçülür. Bazı kitaplar bunun yerine yol üzerindeki **düğüm sayısını** kullanır ve düzeyleri 1'den başlatır. O zaman kökün derinliği 1, yaprağın yüksekliği 1 olur; bütün değerler bir fazla çıkar. İki tanım da tutarlıdır, ama karıştırılırsa sonuçlar bir kayar. Kaynaklar karşılaştırılırken hangi tanımın kullanıldığına bakılmalıdır.



Şekil 9.8: Derinlik ve yükseklik için örnek ağaç. A'nın çocukları B, C, D, E; B'nin F ve G; D'nin H; E'nin I ve J; J'nin K. Vurgulu yol A→E→J→K ağacın en uzun yoludur.

Örnek 9.2

Şekil 9.8'deki ağaçta her düğümün derinliğini ve yüksekliğini bulalım.

1. **Derinlikler yukarıdan aşağı.** Kök A'nın derinliği 0'dır. Çocukları B, C, D, E'nin derinliği 1'dir. Bir alt satırdaki F, G, H, I, J'nin derinliği 2'dir. En alttaki K'nin derinliği 3'tür.
2. **Yükseklikler aşağıdan yukarı.** Yapraklar C, F, G, H, I, K'nin yüksekliği 0'dır. Bir iç düğümün yüksekliği, çocuklarının yüksekliklerinin en büyüğünün bir fazlasıdır. J'nin tek çocuğu K'dir: *1 + 0 = 1*. B'nin çocukları F ve G'dir: *1 + max(0, 0) = 1*. D'nin tek çocuğu H'dir: 1. E'nin çocukları I (0) ve J'dir (1): *1 + max(0, 1) = 2*.
3. **Kök.** A'nın çocuklarının yükseklikleri B 1, C 0, D 1, E 2'dir. A'nın yüksekliği *1 + 2 = 3*'tür. Ağacın yüksekliği de 3'tür ve en derin düğüm K'nin derinliğine eşittir.

Düğüm	A	B	C	D	E	F	G	H	I	J	K
Yükseklik	3	1	0	1	2	0	0	0	0	1	0
Derinlik	0	1	1	1	1	2	2	2	2	2	3

Tablo 9.3: Şekil 9.8'deki ağaçta yükseklik ve derinlik.

Tablodaki iki satır farklı bilgiler taşır. Derinliği bulmak için yalnız düğümün atalarına bakmak yeter; yükseklik ise düğümün bütün soyuna bakmayı gerektirir. Bu fark kodda da görülür: derinlik hesabı ebeveyn zinciri boyunca yukarı çıkar, yükseklik hesabı çocuklar üzerinden aşağı iner.

Sık yapılan hata

Bir düğümün yüksekliğini ağacın yüksekliğinden kendi derinliğini çıkararak bulmaya çalışmak. Bu yalnız düğüm en uzun dal üzerindeyse doğrudur. Şekil 9.8'deki B'nin derinliği 1, ağacın yüksekliği 3'tür; ama B'nin yüksekliği $3 - 1 = 2$ değil, 1'dir. Çünkü B'nin altındaki en uzun dal yalnız bir kenar uzunluğundadır. Yükseklik her zaman düğümün kendi alt ağacına bakılarak hesaplanır.

9.1.7 Yol ve alt ağaç

Kenarlar boyunca bir düğümden diğerine yürümek bir **yol** (*path*) oluşturur. Biçimsel olarak bir yol, düğümler ve kenarlar arasında gidip gelen bir dizidir:

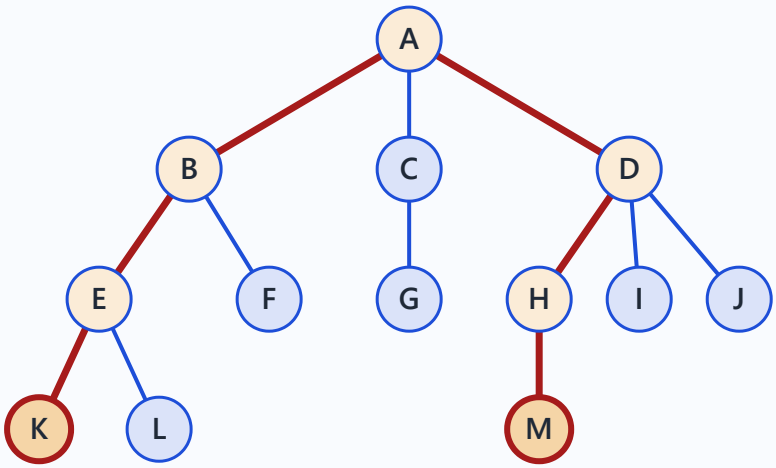
> $p = (v_0, e_1, v_1, e_2, \ldots, e_k, v_k)$

Burada her e_i kenarı v_{i-1} ile v_i düğümlerini bağlar. Yolun başı $start(p) = v_0$, sonu $end(p) = v_k$, uzunluğu da kenar sayısı olan $length(p) = k$'dir. Hiçbir düğümün tekrar etmediği yola **basit yol** denir. Derinlik, kökten düğüme inen yolun uzunluğudur; yükseklik de düğümden en uzak yaprağa inen yolun uzunluğudur.

Teorem 9.2 (Tek yol)

Bir ağaçta herhangi iki düğüm arasında tam olarak bir basit yol vardır.

İspat. Varlık: Her iki düğümden de ebeveyn zinciri boyunca köke çıkılabilir. İki zincirin ilk ortak düğümü c olsun. Birinci düğümden c'ye çıkıp oradan ikinci düğüme inmek bir basit yol verir. Teklik: İki düğüm arasında iki farklı basit yol bulunsaydı, bu yollar bir noktada ayrılıp daha sonra yeniden birleşirdi. Ayrılma noktasından birleşme noktasına bir yoldan gidip öteki yoldan dönmek bir döngü oluştururdu. Ağaçta döngü olmadığından ikinci bir yol bulunamaz. □



Şekil 9.9: K'den M'ye giden tek yol vurgulanmıştır: K → E → B → A → D → H → M. Yol iki düğümün ilk ortak atası olan A'ya kadar çıkar, sonra M'ye iner; uzunluğu 6 kenardır.

Şekil 9.9 ispattaki inşayı somut olarak gösterir. K'nin ataları E, B, A; M'nin ataları H, D, A'dır. İki zincirin ilk ortak düğümü A'dır. Yol K'den A'ya üç kenarla çıkar, A'dan M'ye üç kenarla iner; toplam uzunluk 6'dır. Bu iki düğüm arasında başka bir yol yoktur. Dosya sistemindeki "bir dosyanın tek bir tam yolu vardır" gözlemi de bu teoremin özel bir durumudur.

Alt ağaç (*subtree*), bir düğüm ile onun bütün soyundan oluşan yapıdır. Alt ağaç kendi başına bir ağaçtır: kökü seçilen düğümdür, içindeki her düğümün ebeveyni yine alt ağacın içindedir. Şekil 9.7'nin (d) karesindeki kesikli çerçeve, D'nin alt ağacını gösterir: D, H, I, J ve M. Dosya sisteminde bir klasör ile altındaki her şey bir alt ağaçtır. Bir klasörü kopyalamak ya da silmek, aslında bir alt ağacı kopyalamak ya da silmektir.

Alt ağaç kavramı, ağaç algoritmalarının özyinelemeli yazılmasının temelidir. Bir düğümün çocukları, kendi alt ağaçlarının kökleridir. Bir ağaç üzerinde yapılacak bir iş çoğu zaman şöyle tarif edilir: aynı işi her alt ağaçta yap, sonuçları birleştir, kökü de hesaba kat. Yüksekliğin tanımı bunun ilk örneğidir: bir düğümün yüksekliği, alt ağaçlarının yüksekliklerinin en büyüğünün bir fazlasıdır.

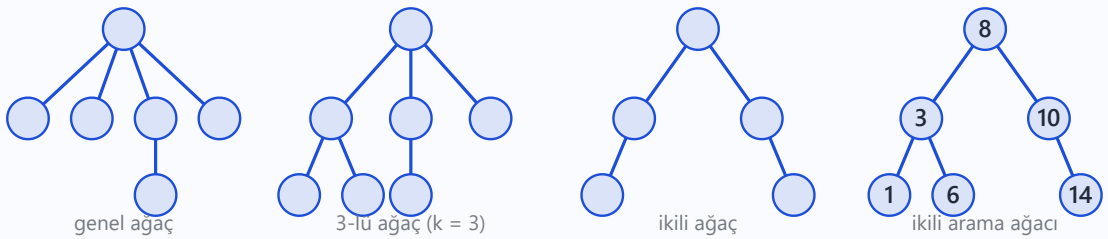
9.1.8 Temel ağaç türleri

Ağaç tanımı bir düğümün çocuk sayısına sınır koymaz. Uygulamada ise çocuk sayısını sınırlamak ya da çocuklara bir düzen getirmek çoğu zaman işleri kolaylaştırır. Tablo 9.4 bu derste karşılaşılan dört türü özetler.

Tür	Özellik
Genel ağaç	Bir düğümün çocuk sayısına sınır yoktur.
***k*-lı ağaç (*k-ary tree*)**	Her düğümün en çok k çocuğu vardır.
İkili ağaç (*binary tree*)	Her düğümün en çok iki çocuğu vardır: sol ve sağ.

Tür	Özellik
İkili arama ağacı (*binary search tree*, BST)	İkili ağaçtır; ayrıca her düğümde sol alt ağaçtaki anahtarlar düğümünkinden küçük, sağ alt ağaçtakiler büyüktür.

Tablo 9.4: Temel ağaç türleri.



Şekil 9.10: Dört ağaç türü (soldan sağa): çocuk sayısı sınırsız genel ağaç, en çok üç çocuklu ağaç ($k = 3$), en çok iki çocuklu ikili ağaç ve ikili arama ağacı. Sonuncusunda kök 8'dir; solda 3 ve onun çocukları 1, 6; sağda 10 ve onun sağ çocuğu 14 bulunur.

Türler birbirinin içine geçer. Her ikili arama ağacı bir ikili ağaçtır, her ikili ağaç $k = 2$ için bir k-lı ağaçtır, her k-lı ağaç da bir genel ağaçtır. Kısıt arttıkça yapı daha az esnek, ama üzerinde çalışmak daha kolay olur. Örneğin ikili arama ağacında bir anahtarı aramak için her düğümde yalnız bir çocuğa inmek yeter: aranan anahtar küçükse sola, büyükse sağa gidilir. Şekil 9.10'un sağındaki ağaçta 6 aranırken 8'den sola, 3'ten sağa inilir ve iki karşılaştırmada 6'ya ulaşılır.

Bu notun ağırlığı ikili ağaçlardadır. Genel ağaçlar da önemlidir, ama her genel ağaç bir ikili ağaç biçiminde saklanabilir; bu dönüşüm gösterimler kesiminde anlatılır. İkili arama ağaçlarının ekleme, silme ve dengeleme işlemleri ise kendi başına geniş bir konudur; burada yalnız tanım düzeyinde anılır.

9.1.9 Ağaç ADT'si

Ağacı kodda kullanmadan önce ona hangi işlemlerle erişileceğini belirlemek gerekir. Yığın ve kuyrukta olduğu gibi burada da işlemleri gerçeklemeden ayırmak işe yarar. Yapının ne yaptığını söyleyen tarife **soyut veri tipi** (*abstract data type*, ADT) denir. Aşağıdaki Java arayüzü, Goodrich, Tamassia ve Goldwasser'in ağaç ADT'sinin Türkçe adlarla yazılmış hâlidir.

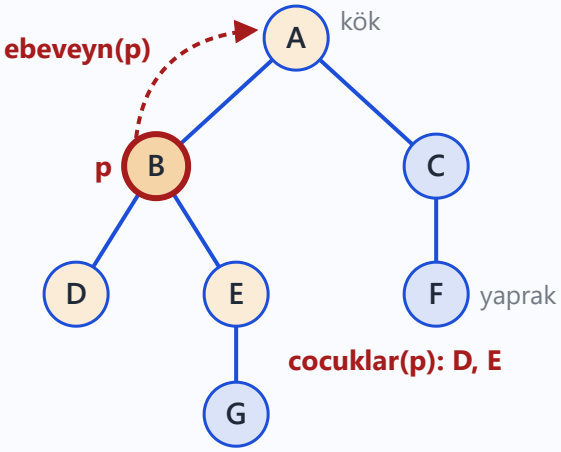
```
public interface Agac<E> extends Iterable<E> {
  Konum<E> kok();
  Konum<E> ebeveyn(Konum<E> p);
  Iterable<Konum<E>> cocuklar(Konum<E> p);
  int cocukSayisi(Konum<E> p);
  boolean icMi(Konum<E> p);      // isInternal
  boolean yaprakMi(Konum<E> p);  // isExternal
  boolean kokMu(Konum<E> p);     // isRoot
  int size();
  boolean isEmpty();
  Iterable<Konum<E>> konumlar(); // positions
}
// Konum<E> (Position): E getElement();
```

Arayüzün merkezinde **konum** (*position*) kavramı vardır. `Konum<E>`, ağaçtaki bir yeri temsil eden küçük bir nesnedir; tek bir işlemi vardır: `getElement()`, o yerdeki öğeyi döndürür. Kullanıcı ağacın içindeki düğüm nesnelerini doğrudan görmez, onlara referans tutamaz, alanlarını değiştiremez. Yalnız konumlar üzerinden ADT metotlarını çağırır. Böylece ağacın bellekte bağlantılarla mı yoksa bir diziyle mi tutulduğu kullanıcıdan gizlenir. Gerçekleme değişse bile ağacı kullanan kod değişmez.

Metotlar üç grupta toplanabilir.

- **Gezinme metotları.** `kok()` kökün konumunu döndürür (ağaç boşsa `null`). `ebeveyn(p)`, `p`'nin ebeveyninin konumunu döndürür; `p` kökse `null` döner. `cocuklar(p)` çocukların konumlarını sırayla verir; dönüş tipi `Iterable` olduğu için sonuç doğrudan bir `for-each` döngüsünde kullanılabilir. `cocukSayisi(p)` çocukların sayısını, yani `p`'nin derecesini verir.
- **Sorgu metotları.** `icMi(p)`, `yaprakMi(p)` ve `kokMu(p)` bir konumun iç düğüm, yaprak ya da kök olup olmadığını söyler. Yorumlardaki İngilizce adlar (`isInternal`, `isExternal`, `isRoot`) kaynak kitaptakilerdir.
- **Genel metotlar.** `size()` düğüm sayısını, `isEmpty()` ağacın boş olup olmadığını verir. `konumlar()` bütün konumları, `iterator()` ise (`Iterable<E>`'den gelir) bütün öğeleri bir sırayla gezer.

İkili ağaç ADT'si bu arayüzü genişletir ve üç metot ekler: `sol(p)`, `sag(p)` ve `kardes(p)`. İlk ikisi `p`'nin sol ve sağ çocuğunu (yoksa `null`), sonuncusu `p`'nin kardeşini döndürür. Kaynak kitapta bu metotlar `left`, `right` ve `sibling` adını taşır.



Şekil 9.11: ADT metotlarının bir örnek üzerinde anlamı. `p` B düğümünün konumudur. `ebeveyn(p)` kök A'yı, `cocuklar(p)` D ve E'yi verir. D bir yaprak, A köktür; E'nin tek çocuğu G, C'nin tek çocuğu F'dir.

Şekil 9.11 üzerinde birkaç çağrının sonucunu yazalım. `cocukSayisi(p)` 2 döndürür. `icMi(p)` `true`, `yaprakMi(p)` `false`, `kokMu(p)` `false` döndürür. `kok()` A'nın konumunu verir; `ebeveyn(kok())` ise `null`'dır. Ağaçta yedi düğüm olduğu için `size()` 7 döndürür.

9.1.10 Kod izleme: derinlik ve yükseklik

ADT metotlarıyla, ağacın nasıl gerçeklendiğini bilmeden derinlik ve yükseklik hesaplanabilir. İki metot da tanımlarını neredeyse kelimesi kelimesine koda çevirir.

```
int derinlik(Konum<E> p) {
    if (kokMu(p)) return 0;
    return 1 + derinlik(ebeveyn(p));
}
int yukseklik(Konum<E> p) {
    int h = 0;
    for (Konum<E> c : cocuklar(p))
        h = Math.max(h, 1 + yukseklik(c));
    return h;
}
```

`derinlik(p)` tanımı izler: kökün derinliği 0'dır; kök olmayan bir düğümün derinliği, ebeveyninin derinliğinin bir fazlasıdır. Metot `p`'den başlayıp ebeveyn zinciri boyunca köke kadar çıkar. Her çağrı bir kenar yukarı çıktığı için `p`'nin derinliği *d* ise metot *d + 1* kez çağrılır; zaman *O(d + 1)*'dir. Çağrı yığınında da en çok *d + 1* çerçeve birikir.

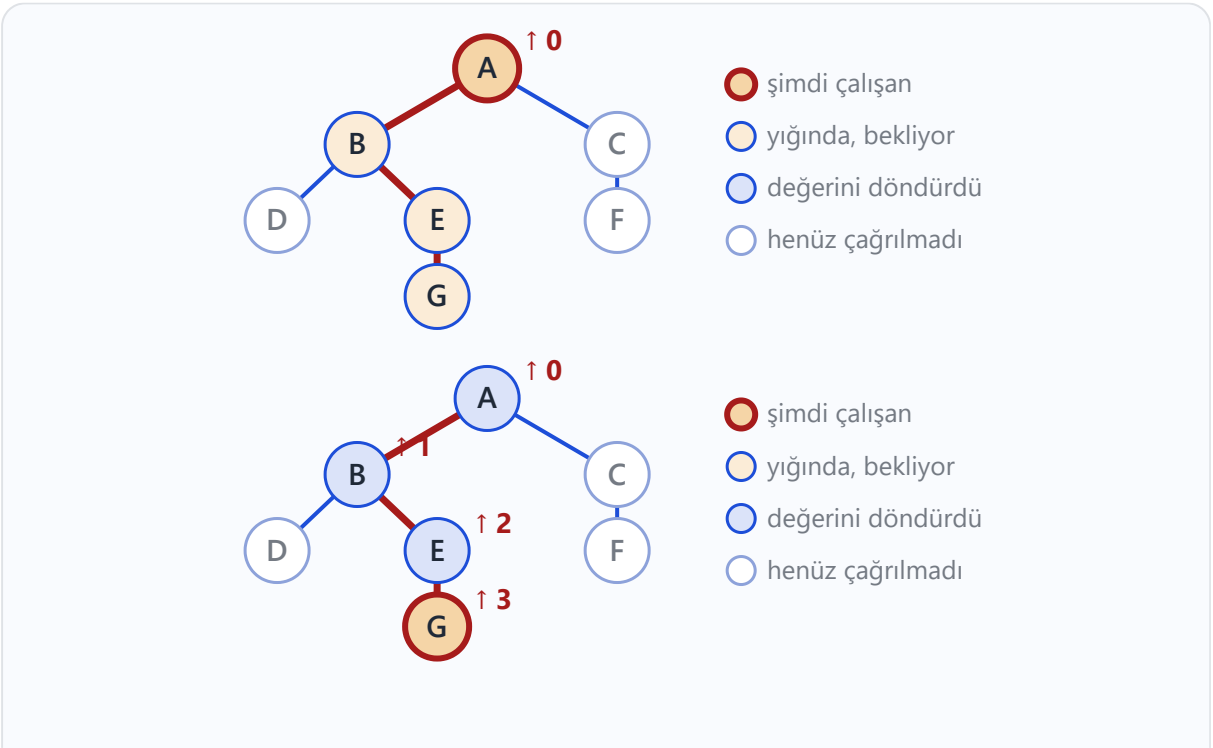
`yukseklik(p)` de tanımı izler, ama aşağı bakar. `h` 0'dan başlar. Döngü `p`'nin her çocuğu `c` için `1 + yukseklik(c)` değerini hesaplar ve `h`'yi bu değerle büyütür. Çocuğu olmayan bir düğümde döngü hiç dönmez ve metot 0 döndürür; bu, yaprağın yüksekliğidir. Metot `p`'nin alt ağacındaki her düğüm için

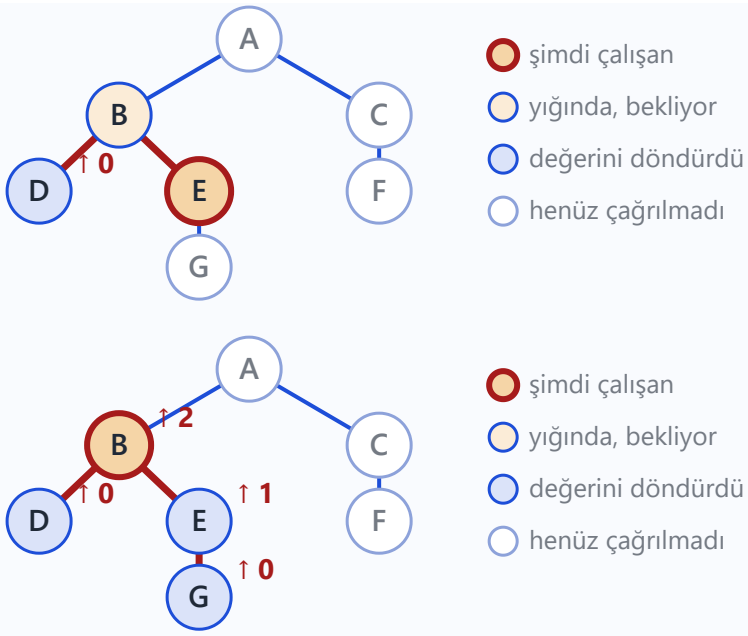
bir kez çağrılır. Bir düğümün çocuklarında gezinmenin maliyeti çocuk sayısıyla orantılıdır; bütün düğümlerin çocuk sayıları toplamı da kenar sayısı kadardır. Bu yüzden `yukseklik(p)`, alt ağaçta n_p düğüm varsa $O(n_p)$ sürer.

İzlemeyi Şekil 9.11'deki ağaç üzerinde yapalım: önce `derinlik(G)`, sonra `yukseklik(B)`. Tablo 9.5 her adımda çağrı yığınını (alttan üste) ve dönen değerleri verir.

Adım	Çalışan çağrı	Çağrı yığını	Ne olur?
1	`derinlik(G)`	main, derinlik(G)	G kök değil; ebeveyni E için `derinlik(E)` çağrılır.
2	`derinlik(E)`	…, derinlik(E)	E kök değil; `derinlik(B)` çağrılır.
3	`derinlik(B)`	…, derinlik(B)	B kök değil; `derinlik(A)` çağrılır.
4	`derinlik(A)`	main, G, E, B, A (5 çerçeve)	A kök: taban durum, 0 döner.
5	`derinlik(B)`	main, G, E, B	*1 + 0 = 1* döner.
6	`derinlik(E)`	main, G, E	*1 + 1 = 2* döner.
7	`derinlik(G)`	main, G	*1 + 2 = 3* döner. G'nin derinliği 3'tür (A–B–E–G).
8	`yukseklik(B)`	main, yukseklik(B)	`h = 0`; çocuklar D, E. İlk çocuk D için özyineleme.
9	`yukseklik(D)`	…, B, D	D'nin çocuğu yok; döngü dönmez, 0 döner.
10	`yukseklik(B)`	main, B	`h = max(0, 1 + 0) = 1`; sıradaki çocuk E.
11	`yukseklik(E)`	…, B, E	`h = 0`; tek çocuk G için özyineleme.
12	`yukseklik(G)`	…, B, E, G	G yaprak: 0 döner.
13	`yukseklik(E)`	…, B, E	`h = max(0, 1 + 0) = 1`; çocuk kalmadı, 1 döner.
14	`yukseklik(B)`	main, B	`h = max(1, 1 + 1) = 2`; çocuk kalmadı, 2 döner.

Tablo 9.5: `derinlik(G)` ve `yukseklik(B)` izlemesi. Çağrı yığını alttan üste yazılmıştır.





Şekil 9.12: İzlemenin dört anı. (a) `derinlik(A)` taban durumdadır; G, E, B çerçeveleri yığında bekler. (b) Değerler geri döner: B 1, E 2, G 3. (c) `yukseklik(B)` D'den 0 almış, E'nin alt ağacına inmiştir. (d) G'den 0, E'den 1 gelmiş; `yukseklik(B) = 2`.

İzleme iki metodun simetrisini açıkça gösterir. `derinlik` yukarı çıkarken çağrı yaparsa, değerler aşağı inerken hesaplanır: önce kök 0 der, sonra her çerçeve aldığı değere 1 ekler. `yukseklik` aşağı inerken çağrı yaparsa, değerler yukarı çıkarken hesaplanır: önce yapraklar 0 der, her iç düğüm çocuklarından gelenlerin en büyüğüne 1 ekler. `yukseklik(B)` dört kez çağrılmıştır: B, D, E ve G için. Bunlar B'nin alt ağacındaki düğümlerin tamamıdır; ağacın geri kalanına (A, C, F) hiç dokunulmamıştır.

Ağacın yüksekliği iki yolla hesaplanabilir. Birincisi `yukseklik(kok())` çağrısıdır ve $O(n)$ sürer. İkincisi, bütün düğümlerin derinliklerini `derinlik` ile hesaplayıp en büyüğünü almaktır. Bu ikinci yol her düğüm için ayrı bir yukarı tırmanış yapar. Derinliği d_p olan düğüm için maliyet $O(d_p + 1)$ olduğundan toplam maliyet $O(n + \Sigma\, d_p)$ olur. Zincir biçimli bir ağaçta derinlikler 0, 1, …, $n - 1$'dir ve toplamları $n(n - 1)/2$'ye ulaşır; maliyet $O(n^2)$'ye çıkar. Doğru yöntem birincisidir.

Sık yapılan hata

`yukseklik` metodunda `h`'yi 0 yerine 1 ile başlatmak ya da yaprakta 1 döndürmek. O zaman metot yüksekliği kenar değil düğüm sayısıyla ölçer ve bütün değerler bir fazla çıkar. Tanım kenar sayısıyla yapıldıysa yaprak 0 döndürmelidir. Benzer biçimde `derinlik`'te taban durumu unutmak (`kokMu(p)` denetimi olmadan `ebeveyn(p)` çağırmak) kökte `null` bir konumla devam edip hataya yol açar.

9.2 İkili ağaçlar

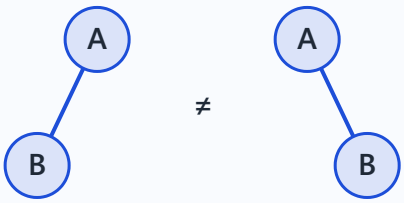
Genel ağaçta bir düğümün kaç çocuğu olacağı bilinmez. Bu esneklik, ağacı saklayan kodu da karmaşıklaştırır: her düğüm değişken uzunlukta bir çocuk listesi tutmak zorunda kalır. Çocuk sayısı en çok ikiyle sınırlanınca hem saklama hem de algoritmalar sadeleşir. İkili ağaçlar bu yüzden ağaçların en çok kullanılan biçimidir. İfade ağaçları, ikili arama ağaçları, heap yapıları ve Huffman kodlama ağaçları ikili ağaç üzerine kuruludur.

9.2.1 İkili ağacın tanımı

İkili ağaçta her düğümün en çok iki çocuğu vardır. Ama asıl ayırt edici özellik bu sayı değil, çocukların *adlandırılmasıdır*. Her çocuk ya **sol çocuk** ya da **sağ çocuktur**. Tek çocuklu bir düğümde bile o çocuğun solda mı sağda mı olduğu bilinir ve önemlidir.

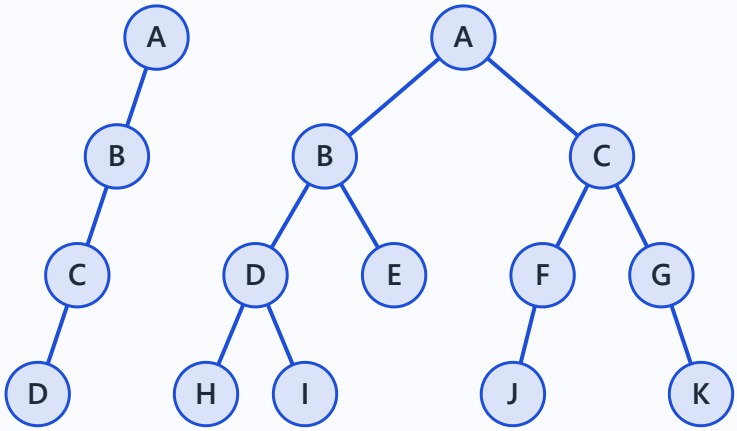
> **Tanım 9.3**
>
> Bir **ikili ağaç** (*binary tree*) ya boştur ya da bir kök düğüm ile iki ikili ağaçtan oluşur. Bu iki ikili ağaca kökün **sol alt ağacı** ve **sağ alt ağacı** denir. Sol alt ağaç boş değilse kökü, ana kökün sol çocuğudur; sağ alt ağaç için de aynısı geçerlidir.

Tanım özyinelemelidir: ikili ağacı yine ikili ağaçlar cinsinden tarif eder. Bu bir döngü değildir, çünkü her adımda alt ağaçlar daha küçüktür ve sonunda boş ağaca varılır. Boş ağaç özyinelemenin **taban durumudur** (*base case*). Tanımın bu biçimi, ikili ağaç metotlarının yazılışını da belirler. Metot boş ağaçta bir değer döndürür. Boş olmayan ağaçta kendini sol ve sağ alt ağaca uygular, sonra sonuçları birleştirir.



Şekil 9.13: İki farklı ikili ağaç. Solda B, A'nın sol çocuğudur; sağda B, A'nın sağ çocuğudur. Genel ağaç olarak ikisi aynıdır, ikili ağaç olarak farklıdır.

Şekil 9.13 bu farkı gösterir. İki çizimde de A'nın tek bir çocuğu, B vardır. Genel ağaç gözüyle bakılırsa iki yapı aynıdır: bir kök ve onun tek çocuğu. İkili ağaç gözüyle bakılırsa farklıdır: birinde sol alt ağaç dolu, sağ alt ağaç boştur; ötekinde tersi. Ayrımın pratik bir anlamı vardır. Örneğin ikili arama ağacında sol, küçük anahtarlar demektir. Kök ortada gezinme de sol ve sağı ayırır; iki ağaç için farklı çıktılar verir: soldaki için B A, sağdaki için A B.



Şekil 9.14: İki ikili ağaç. Solda A, B, C, D düğümlerinden oluşan sola çarpık bir zincir; sağda 11 düğümlü, dalları daha dengeli dağılmış bir ikili ağaç.

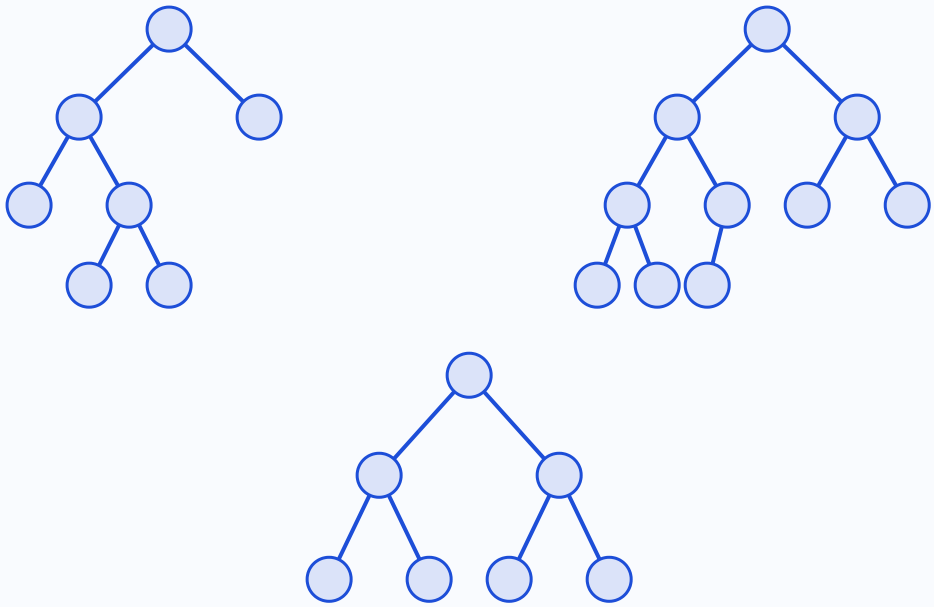
İkili ağacın düğümleri dengeli dağılmak zorunda değildir. Şekil 9.14'ün solundaki yapıda her düğümün yalnız sol çocuğu vardır ve ağaç bir zincire dönüşmüştür. Böyle ağaçlara **çarpık** (*skewed*) ağaç denir. Çarpık ağaç da geçerli bir ikili ağaçtır, ama ikili ağacın sağladığı avantajların çoğunu kaybeder. Dört düğümlü zincirin yüksekliği 3'tür; aynı dört düğüm dengeli dizilseydi yükseklik 2 olurdu. Fark küçük ağaçta önemsiz görünür, ama bir milyon düğümde yaklaşık 20 ile bir milyon arasındaki farka dönüşür.

9.2.2 Dolu, tam ve mükemmel ikili ağaç

İkili ağaçların biçimini anlatmak için üç özel ad kullanılır. Bu adlar birbirine çok yakın olduğu için en çok karıştırılan kavramlar arasındadır. Türkçe kaynaklarda “dolu” ve “tam” sözcükleri yer yer ters anlamda da kullanılır. Karışıklığı önlemenin en güvenli yolu İngilizce karşılıkları esas almaktır: *full*, *complete* ve *perfect*.

Tanım 9.4

- **Dolu ikili ağaç** (*full binary tree*): Her düğümün ya 0 ya da 2 çocuğu vardır. Tek çocuklu düğüm yoktur.
- **Tam ikili ağaç** (*complete binary tree*): Son düzey dışındaki bütün düzeyler tamamen doludur; son düzeydeki düğümler soldan sağa boşluksuz yerleşir.
- **Mükemmel ikili ağaç** (*perfect binary tree*): Bütün iç düğümlerin iki çocuğu vardır ve bütün yapraklar aynı derinliktedir.



Şekil 9.15: Soldan sağa: dolu ikili ağaç (her düğümde 0 ya da 2 çocuk), tam ikili ağaç (son düzeyden önceki düzeyler dolu, son düzey soldan sağa dolar) ve mükemmel ikili ağaç (bütün yapraklar aynı derinlikte).

Üç tanım farklı şeylere bakar. *Dolu* olmak, her düğümün kendi çocuk sayısıyla ilgili yerel bir koşuldur; ağacın genel şekline bakmaz. Bu yüzden dolu bir ağaç çok dengesiz olabilir: her adımda sağ çocuğu bir yaprak olan, sol çocuğu ise yeniden ikiye dallanan bir ağaç doludur ama bir zincire çok benzer. *Tam* olmak, ağacın düzeylerine bakan genel bir koşuldur; tek çocuklu bir düğüme izin verir, yeter ki bu düğüm son düzeyin hemen üstünde ve doğru konumda dursun. *Mükemmel* olmak ise en güçlü koşuldur: ağaç, yüksekliğinin izin verdiği bütün düğümleri içerir.

Tam ağacı anlamanın pratik bir yolu, düğümleri düzey düzey ve her düzeyde soldan sağa numaralamaktır. Kök 1, onun çocukları 2 ve 3, bir alt düzey 4, 5, 6, 7 numarasını alır. Bir ikili ağacın n düğümü 1'den n'ye kadar bütün numaraları boşluksuz dolduruyorsa ağaç tamdır. Arada bir numara boş kalıyorsa, örneğin 10 boşken 11 doluysa, ağaç tam değildir. Bu numaralama, dizi gösteriminin de temelidir.

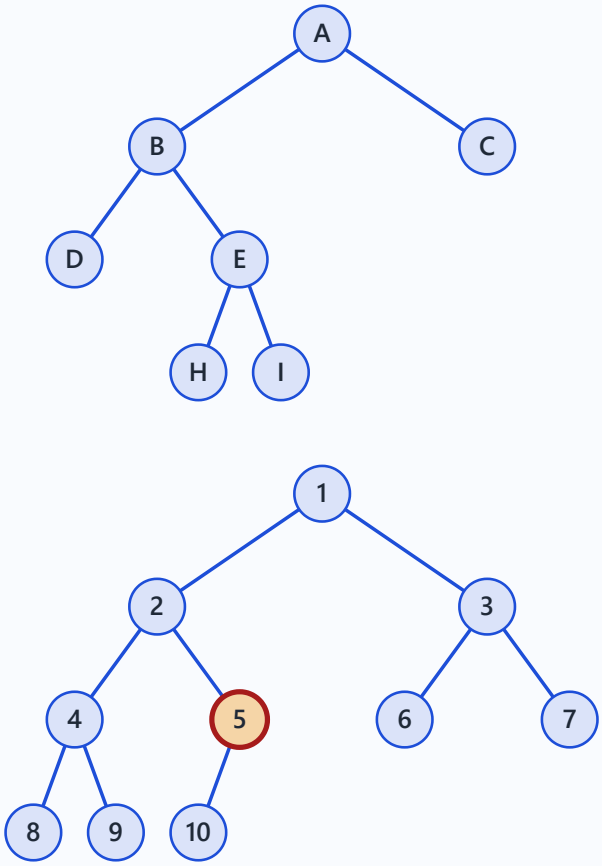
Tanımlar arasındaki ilişkiler şöyle özetlenebilir. Mükemmel bir ağaç hem dolu hem tamdır: iç düğümlerin hepsinde iki çocuk vardır ve bütün düzeyler, son düzey de dahil, doludur. Dolu ile tam ise birbirini gerektirmez. Dolu ama tam olmayan, tam ama dolu olmayan, ne dolu ne tam olan ağaçlar vardır.

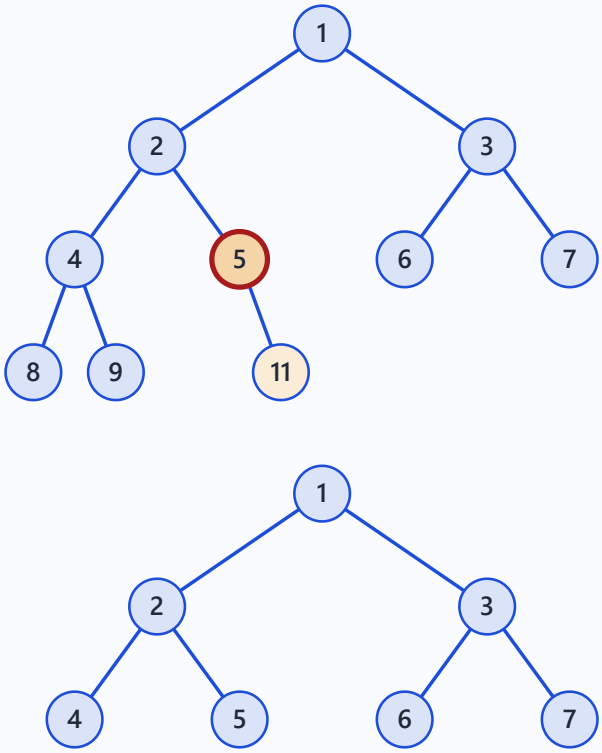
Sık yapılan hata

"Dolu" ile "tam"ı aynı sanmak ya da birinin ötekini gerektirdiğini düşünmek. Dolu, her düğümün 0 ya da 2 çocuğu olmasıdır; tam, son düzey dışındaki düzeylerin dolu ve son düzeyin soldan dolması demektir. Tam bir ağaçta tek çocuklu bir düğüm bulunabilir; dolu bir ağaçta da üst düzeylerde boşluklar olabilir. Bir ağacın iki özelliği de ayrı ayrı denetlenmelidir.

9.2.3 Dört ağaç üzerinde ayrım

Tanımları sınamanın en iyi yolu, onları somut ağaçlara uygulamaktır. Şekil 9.16 dört ağaç verir; her biri için üç özellik ayrı ayrı denetlenir.





Şekil 9.16: Dört ikili ağaç. (a) A kök; B'nin çocukları D ve E; E'nin çocukları H ve I; C yapraktır. (b) 1'den 10'a numaralı ağaç; son düzeyde 8, 9, 10 soldan sağa dizilir. (c) 1'den 9'a numaralı düğümler ve 11; 10 numaralı konum boştur. (d) 1'den 7'ye mükemmel ağaç.

Örnek 9.3

Şekil 9.16'daki her ağaç dolu mu, tam mı, mükemmel mi?

1. **(a)** Çocuk sayıları: A 2, B 2, E 2, C, D, H, I 0. Tek çocuklu düğüm yoktur; ağaç *doludur*. Düzeylere bakalım: 0. düzeyde A, 1. düzeyde B ve C, 2. düzeyde yalnız D ve E vardır. C'nin altındaki iki konum boştur. Buna karşın 3. düzeyde H ve I bulunur. Son düzeyden önceki düzey dolu olmadığı için ağaç *tam değildir*. Yapraklar farklı derinliklerde olduğu için (C 1'de, D 2'de, H ve I 3'te) *mükemmel de değildir*.
2. **(b)** Düğümler 1'den 10'a kadar boşluksuz numaralanmıştır: 0., 1. ve 2. düzeyler doludur; son düzeyde 8, 9, 10 soldan sağa dizilmiştir. Ağaç *tamdır*. Ancak 5'in yalnız bir çocuğu (10) vardır; ağaç *dolu değildir*. Dolu olmadığı için mükemmel de değildir.
3. **(c)** 5'in tek çocuğu vardır: 11 numaralı sağ çocuk. Ağaç *dolu değildir*. Numaralamada 10 boşken 11 doludur; son düzey soldan sağa boşluksuz dolmamıştır. Ağaç *tam da değildir*. Ne dolu ne tam olan bir ağaç mükemmel de olamaz.

4. **(d)** Bütün iç düğümlerin (1, 2, 3) iki çocuğu vardır ve bütün yapraklar (4, 5, 6, 7) aynı derinliktedir. Ağaç *mükemmeldir*; dolayısıyla hem dolu hem tamdır.

Ağaç	Dolu	Tam	Mükemmel
(a)	evet	hayır	hayır
(b)	hayır	evet	hayır
(c)	hayır	hayır	hayır
(d)	evet	evet	evet

Tablo 9.6: Şekil 9.16'daki dört ağacın sınıflandırması.

Tablonun ilk iki satırı, dolu ile tamın bağımsız olduğunun somut kanıtıdır: (a) dolu ama tam değil, (b) tam ama dolu değil. (b) ile (c) arasındaki fark da öğreticidir. İki ağaçta da 5 tek çocukludur. (b)'de bu çocuk sol konumdadır (10) ve numaralar boşluksuz devam eder. (c)'de çocuk sağ konumdadır (11) ve 10 boş kalır. Tek bir düğümün sol yerine sağda durması, ağacı tam olmaktan çıkarmıştır.

9.2.4 Düğüm sayısı ve yükseklik bağıntıları

İkili ağaçta her düğüm en çok iki çocuk taşıdığından, her düzeydeki düğüm sayısı bir öncekinin en çok iki katı olabilir. Bu basit gözlem, düğüm sayısı ile yükseklik arasındaki bütün bağıntıların kaynağıdır. Tablo 9.7 bu bağıntıları toplar; ardından her birinin gerekçesi verilir.

Özellik	Formül	Örnek
***d* derinliğindeki en çok düğüm**	2^d	$d = 3 \rightarrow$ en çok 8 düğüm
Yüksekliği *h* olan ağaçta en çok düğüm	$2^{h+1} - 1$	$h = 3 \rightarrow 15$ (mükemmel ağaç)
Yüksekliği *h* olan ağaçta en az düğüm	$h + 1$	çarpık (zincir) ağaç
***n* düğümlü ağacın yüksekliği**	$\lfloor \log_2 n \rfloor \le h \le n - 1$	$n = 15 \rightarrow 3 \le h \le 14$
***n* düğümlü ağaçta kenar sayısı**	$n - 1$	her düğümün tek ebeveyni
Dolu ikili ağaçta yaprak sayısı	iç düğüm sayısı + 1	7 düğüm → 3 iç, 4 yaprak

Tablo 9.7: İkili ağaç özellikleri.

Teorem 9.3 (Düğüm sayısı sınırları)

Bir ikili ağaçta (a) d derinliğinde en çok 2^d düğüm bulunur; (b) yüksekliği h olan ağaçta düğüm sayısı n için $h + 1 \le n \le 2^{h+1} - 1$ geçerlidir; (c) $n \ge 1$ düğümlü ağacın yüksekliği $\lfloor \log_2 n \rfloor \le h \le n - 1$ aralığındadır.

İspat. (a) Tümevarımla: 0. derinlikte yalnız kök vardır, $2^0 = 1$. d derinliğinde en çok 2^d düğüm varsa, her biri en çok iki çocuk taşıdığı için $d + 1$ derinliğinde en çok $2 \cdot 2^d = 2^{d+1}$ düğüm olabilir.

(b) Üst sınır: Yüksekliği h olan ağaçta derinlikler 0'dan h'ye kadardır. (a)'ya göre toplam en çok $1 + 2 + 4 + \ldots + 2^h = 2^{h+1} - 1$'dir. Alt sınır: Yükseklik h ise kökten en derin yaprağa h kenarlı bir yol vardır ve bu yol $h + 1$ düğüm içerir.

(c) (b)'deki iki eşitsizlik h için çözülür. $h + 1 \le n$'den $h \le n - 1$ çıkar. $n \le 2^{h+1} - 1$'den $n < 2^{h+1}$, yani $\log_2 n < h + 1$ çıkar. h tam sayı olduğundan $h \ge \lfloor \log_2 n \rfloor$ olur. □

Üst sınırdaki toplam, ikinin ardışık kuvvetlerinin toplamıdır ve her zaman bir sonraki kuvvetten bir eksiktir: $1 + 2 + 4 + 8 = 15 = 16 - 1$. Bu sınıra yalnız mükemmel ağaçta ulaşılır. Yüksekliği 3 olan mükemmel ağaç 15 düğüm içerir; yüksekliği 4 olan 31 düğüm içerir. Alt sınıra ise her düğümün tek çocuklu olduğu zincirde ulaşılır: yüksekliği 3 olan zincir 4 düğümlüdür.

Örnek 9.4

1000 düğümlü bir ikili ağacın yüksekliği en az ve en çok kaç olabilir?

1. **En çok.** Her düğümün tek çocuğu olduğu zincirde yükseklik $n - 1 = 999$'dur.
2. **En az.** Yüksekliği h olan bir ağaçta en çok $2^{h+1} - 1$ düğüm bulunur. $h = 8$ için bu değer 511'dir; 1000 düğüme yetmez. $h = 9$ için 1023'tür; yeter. En küçük yükseklik 9'dur. Formülle: $\lfloor \log_2 1000 \rfloor = \lfloor 9{,}97 \rfloor = 9$.
3. **Yorum.** Kökten yaprağa inen bir işlem, dengeli ağaçta yaklaşık 10, zincir biçimli ağaçta yaklaşık 1000 adım sürer.

Bu bağıntıların neden önemli olduğu son satırda görülür. Ağaç üzerindeki birçok işlem, örneğin bir anahtarı aramak ya da yeni bir düğümü uygun yere eklemek, kökten bir yaprağa doğru tek bir yol boyunca ilerler. Böyle işlemlerin maliyeti $O(h)$'dir. Dengeli bir ağaçta $h \approx \log_2 n$ olduğundan bir milyon düğümlü ağaçta yaklaşık 20 düzey vardır ($2^{20} \approx 10^6$). Çarpık ağaçta ise $h = n - 1$'dir ve aynı işlem bir bağlı listede arama kadar yavaşlar. Arama ağaçlarının dengelenmesi bu farkı kapatmak için yapılır.

Sık yapılan hata

Yüksekliği h olan ağaçta en çok düğüm sayısını 2^h ya da $2^h - 1$ yazmak. 2^h, yalnız en alt düzeydeki (derinlik h) düğüm sayısının sınırıdır. Bütün düzeyler toplanınca $2^{h+1} - 1$ bulunur. $2^h - 1$ değeri ise yüksekliği düğüm sayısıyla ölçen kaynaklardaki formüldür; iki tanım karıştırıldığında ortaya çıkar.

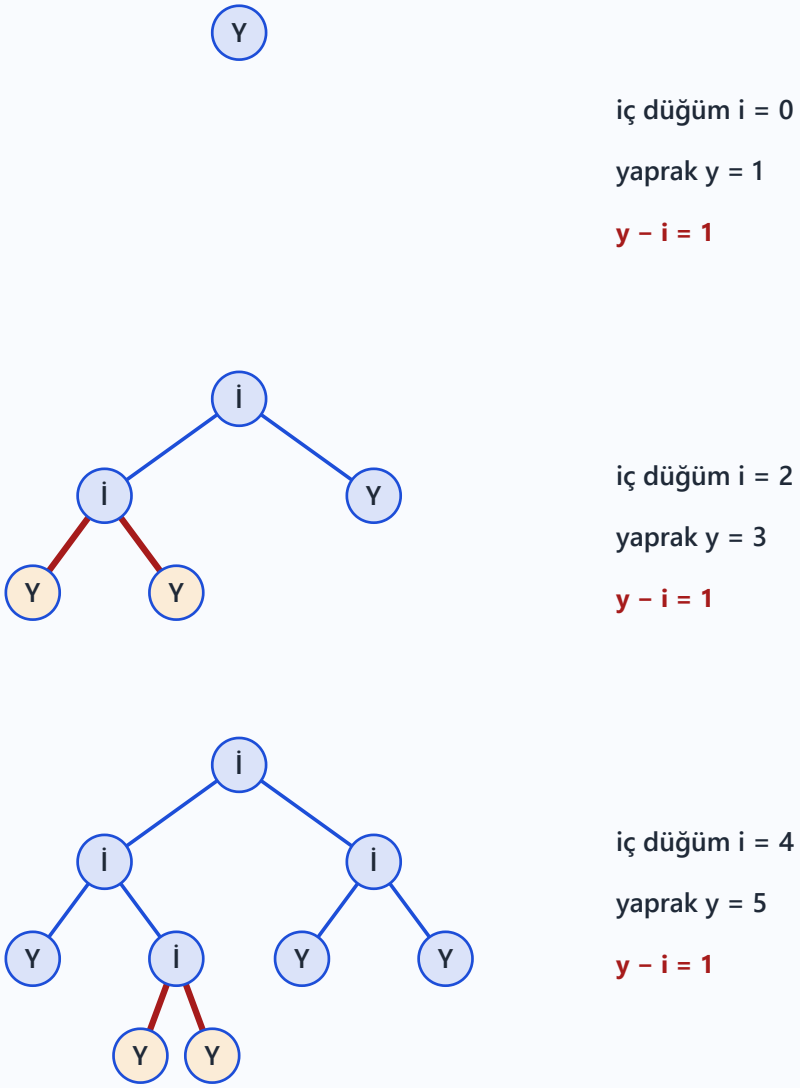
9.2.5 Dolu ikili ağaçta yaprak ve iç düğüm sayısı

Dolu ikili ağaçların ilginç bir sayma özelliği vardır: yaprak sayısı her zaman iç düğüm sayısından tam bir fazladır. Tablo 9.7'deki son satır bunu 7 düğümlü mükemmel ağaçla örnekler: 3 iç düğüm (kök ve onun iki çocuğu), 4 yaprak.

Teorem 9.4 (Dolu ikili ağaçta yaprak sayısı)

Boş olmayan dolu bir ikili ağaçta iç düğüm sayısı i, yaprak sayısı y ise $y = i + 1$'dir. Dolayısıyla düğüm sayısı $n = i + y = 2i + 1$ her zaman tektir.

İspatı sezmenin en kolay yolu, her dolu ikili ağacın tek bir düğümden başlayıp aynı işlemin tekrarıyla elde edilebildiğini görmektir. Dolu ağaçta bir yaprağa tek çocuk eklenemez; ağaç dolu kalsın diye her seferinde bir yaprağa *iki* çocuk birden eklenir. Şekil 9.17 bu büyümeyi adım adım gösterir.



Şekil 9.17: Dolu ikili ağacın büyümesi (İ: iç düğüm, Y: yaprak; turuncu: son adımda eklenen yapraklar). Üstte tek düğümlü ağaç: $i = 0, y = 1$. Ortada iki genişletmeden sonra: $i = 2, y = 3$. Altta dört genişletmeden sonra: $i = 4, y = 5$. Her durumda $y - i = 1$.

İspat. Ağaçtaki düğüm sayısı üzerinden tümevarım yapalım. *Taban:* Tek düğümlü ağaçta kök bir yapraktır: $i = 0, y = 1, y = i + 1$. *Adım:* Birden çok düğümlü dolu bir ağaçta iki çocuğu da yaprak olan bir iç düğüm vardır. En derin yaprağın ebeveyni böyledir. Dolu olma koşulu gereği bu yaprağın bir kardeşi bulunur; kardeşin çocuğu olsaydı en derin yaprak o olurdu, bu yüzden kardeş de yapraktır. Bu iki yaprak silinirse ebeveynleri yaprak olur ve yine dolu, daha küçük bir ağaç elde edilir. Bu küçük ağaçta $y' = i' + 1$'dir. İki yaprağı geri eklemek, bir yaprağı iç düğüme çevirir ($i = i' + 1$) ve yaprak sayısını $y = y' - 1 + 2 = y' + 1$ yapar. Fark değişmez: $y - i = y' - i' = 1$. □

Teoremin sonucu olan $n = 2i + 1$, dolu ikili ağaçta düğüm sayısının hiçbir zaman çift olamayacağını söyler. 6 düğümlü dolu ikili ağaç yoktur. Ayrıca düğüm sayısı biliniyorsa yaprak ve iç düğüm sayıları da bilinir: $y = (n + 1)/2$, $i = (n - 1)/2$. 15 düğümlü bir dolu ağaçta 8 yaprak ve 7 iç düğüm vardır.

Bu özellik birçok uygulamada doğrudan kullanılır. Bir aritmetik ifade ağacında her işleç iki işlenen alır; işleçler iç düğüm, sayılar yapraktır ve ağaç doludur. Dolayısıyla ikili işleçlerle yazılmış bir ifadede sayı adedi, işleç adedinden her zaman bir fazladır: $(3 + 4) \times (5 - 2)$ ifadesinde 3 işleç ve 4 sayı vardır. Huffman kodlamasında da her iç düğüm iki alt ağacı birleştirerek oluşur. k farklı karakter için kurulan ağacın k yaprağı ve $k - 1$ iç düğümü olur.

> **Sık yapılan hata**
>
> $y = i + 1$ eşitliğini her ikili ağaca uygulamak. Eşitlik yalnız dolu ağaçlarda geçerlidir. Tek çocuklu düğümler yaprak sayısını artırmadan iç düğüm sayısını artırır. Örneğin üç düğümlü bir zincirde 2 iç düğüm ve yalnız 1 yaprak vardır. Genel ikili ağaçta doğru bağıntı şudur: iki çocuklu düğümlerin sayısı i_2 ise yaprak sayısı $i_2 + 1$'dir; tek çocuklu düğümler hesaba girmez.

Kutudaki genel bağıntı da kenar sayılarak kolayca gösterilir. Tek çocuklu düğüm sayısı i_1, iki çocuklu düğüm sayısı i_2 olsun. Kenar sayısı bir yandan $n - 1 = y + i_1 + i_2 - 1$'dir. Her kenar bir çocuğa da iner; yani kenar sayısı $i_1 + 2i_2$'dir. İkisi eşitlenince $y = i_2 + 1$ kalır. Dolu ağaçta $i_1 = 0$ olduğu için $i_2 = i$ olur ve teorem yeniden elde edilir.

9.3 Gösterimler

Ağaç kâğıt üzerinde düğümler ve çizgilerden oluşur; bellekte ise ya ardışık hücrelerden oluşan bir dizi ya da birbirini gösteren nesneler vardır. Bir ağacı bu iki araçtan biriyle temsil etmenin yollarına **gösterim** denir. Hangi gösterimin seçildiği, ebeveyn ve çocuklara ne kadar hızlı ulaşılacağını, ne kadar bellek harcanacağını ve ağacın ne kadar kolay büyütüleceğini belirler.

9.3.1 Dizi gösterimi

Tam ağaçların numaralanması, ikili ağacı bir diziye yerleştirmenin doğal yolunu verir. Düğümler düzey düzey, her düzeyde soldan sağa okunur ve bu sırayla dizinin hücrelerine yazılır. Kök ilk hücreye, onun çocukları sonraki iki hücreye, bir alt düzey sonraki dört hücreye gider. Bu düzende bir düğü-

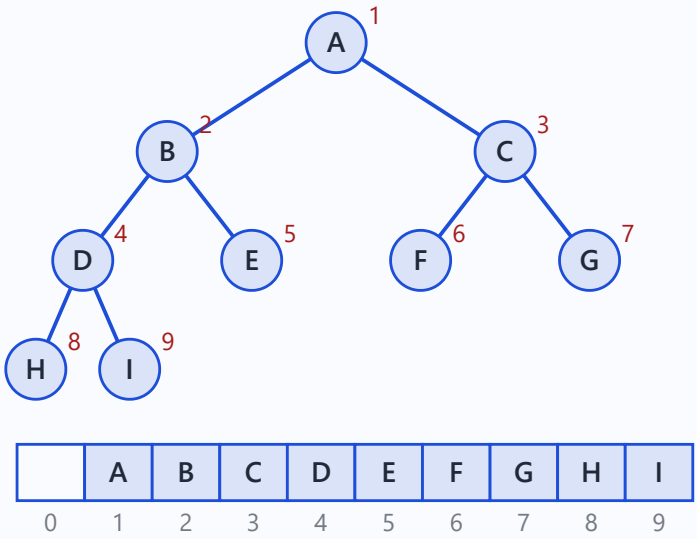
mün çocuklarının ve ebeveyninin indisi, düğümün kendi indisinden basit bir aritmetikle bulunur.

	1 tabanlı (kök `a[1]` 'de)	0 tabanlı (kök `a[0]` 'da)
Sol çocuk	$2i$	$2i + 1$
Sağ çocuk	$2i + 1$	$2i + 2$
Ebeveyn	$\lfloor i / 2 \rfloor$	$\lfloor (i - 1) / 2 \rfloor$

Tablo 9.8: Dizi gösteriminde indis *i*'deki düğüm için indis bağıntıları.

1 tabanlı düzende formüller daha sadedir; bu yüzden birçok kitap dizinin 0. hücresini boş bırakır ve kökü 1. hücreye koyar. Formüllerin nedeni şöyle görülür. d derinliğindeki ilk düğüm 2^d indisini alır, çünkü üstündeki düzeylerde toplam $2^d - 1$ düğüm vardır. Bir düğümden önce kendi düzeyinde j düğüm bulunuyorsa, bir alt düzeyde onların $2j$ çocuğu bulunur. Bu sayımlar birleşince indis i'deki düğümün sol çocuğu $2i$'ye düşer; sağ çocuk hemen yanındadır. Ebeveyn formülü bu ikisinin tersidir: hem $2i$ hem $2i + 1$, 2'ye tam sayı bölmesiyle i'yi verir.

0 tabanlı formüller, 1 tabanlılardan indisleri bir kaydırarak elde edilir. 0 tabanlı indis i, 1 tabanlı $i + 1$'e karşılık gelir. Onun sol çocuğu 1 tabanlıda $2(i + 1) = 2i + 2$'dir; 0 tabanlıya geri dönmek için bir çıkarılır ve $2i + 1$ bulunur. Diğer formüller de aynı yolla çıkar.



Şekil 9.18: A'dan I'ya dokuz düğümlü ikili ağaç ve 1 tabanlı dizideki yerleşimi. A indis 1'de, B 2'de, C 3'te, …, I 9'dadır; 0. hücre kullanılmaz. Düğümlerin yanındaki kırmızı sayılar indislerdir.

Şekil 9.18'deki ağaçta E indis 5'tedir. Sol çocuğu $2 \cdot 5 = 10$, sağ çocuğu 11 indisinde olurdu. Dizide yalnız 9 düğüm bulunduğu için 10 ve 11 dizinin dışındadır; E'nin çocuğu yoktur, E bir yapraktır. E'nin ebeveyni $\lfloor 5 / 2 \rfloor = 2$, yani B'dir. Bu gösterimde düğümler referans taşımaz. Ebeveyn ve çocuk bilgisi, konumun kendisinde saklıdır; ulaşmak için birkaç aritmetik işlem yeterlidir ve her erişim $O(1)$ sürer.

Dizi gösterimi en çok **tam** ikili ağaçlara yakışır. Tam ağaçta düğümler 1'den *n*'ye kadar bütün indisleri boşluksuz doldurur; dizi tam *n* hücre kullanır ve hiç boşluk kalmaz. İkili heap bu yüzden hemen her zaman bir dizide tutulur. Tam olmayan ağaçlarda ise boş konumlar dizide boş hücre olarak kalır; bunun bedeli aşağıda hesaplanır.

9.3.2 Kod izleme: indis hesabı ve köke giden yol

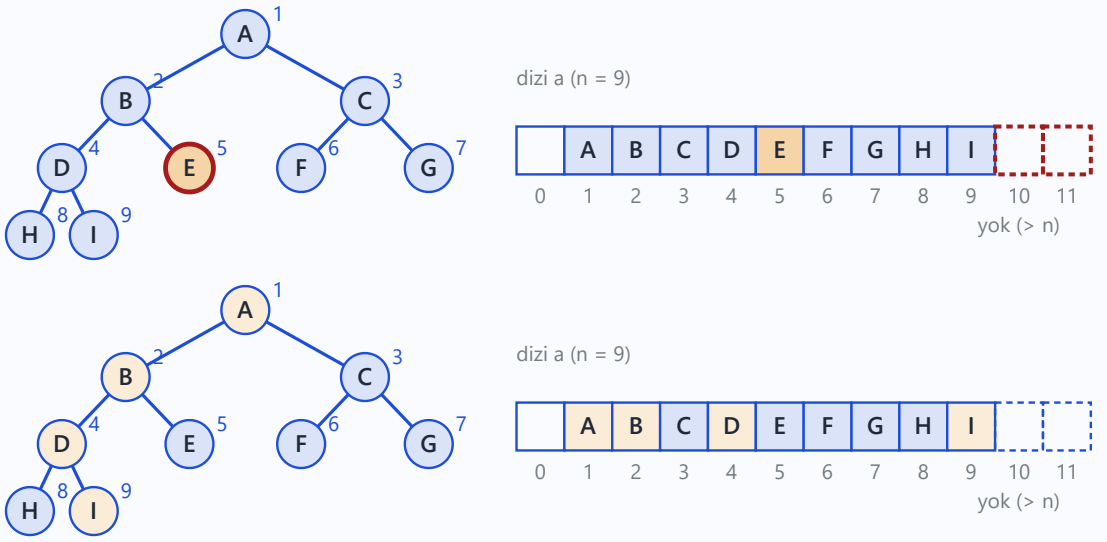
Aşağıdaki kod 1 tabanlı formülleri üç küçük metoda çevirir ve onları kullanarak bir düğümden köke kadar bütün ataları yazdırır.

```
int sol(int i)     { return 2 * i; }
int sag(int i)     { return 2 * i + 1; }
int ebeveyn(int i) { return i / 2; }
void kokeYol(char[] a, int i) {  // i'den köke
  while (i >= 1) {
    System.out.print(a[i] + " ");
    i = ebeveyn(i);
  }
}
```

`sol`, `sag` ve `ebeveyn` tek satırlık metotlardır. `ebeveyn`'de `i / 2` ifadesi Java'da tam sayı bölmesidir; sonucun ondalık kısmı atılır, yani aşağı yuvarlanır. Bu, *⌊i / 2⌋*'nin ta kendisidir. `kokeYol(a, i)`, `i` 1'den küçük olmadığı sürece döner. Her turda `a[i]`'yi yazdırır ve `i`'yi ebeveynin indisine çeker. Kökün indisi 1'dir; onun "ebeveyni" *⌊1 / 2⌋ = 0* çıkar ve döngü durur. 0. hücrenin boş bırakılması bu yüzden kullanışlıdır: 0 indisi "ebeveyn yok" anlamına gelir.

Çağrı / adım	Hesap	Sonuç
`sol(2)`	$2 \cdot 2 = 4$	`a[4] = D`, B'nin sol çocuğu
`sag(2)`	$2 \cdot 2 + 1 = 5$	`a[5] = E`, B'nin sağ çocuğu
`ebeveyn(2)`	$2 / 2 = 1$	`a[1] = A`, kök
`sol(4)`, `sag(4)`	8 ve 9	H ve I, D'nin çocukları
`sol(5)`, `sag(5)`	10 ve 11 > *n*	dizinin dışında: E yapraktır
`ebeveyn(9)`	$9 / 2 = 4$ (4,5 aşağı yuvarlanır)	D; `ebeveyn(8)` de 4'tür
`kokeYol` 1. tur	$i = 9 \geq 1$	I yazdırılır; `i = 4`
2. tur	$i = 4 \geq 1$	D yazdırılır; `i = 2`
3. tur	$i = 2 \geq 1$	B yazdırılır; `i = 1`
4. tur	$i = 1 \geq 1$	A yazdırılır; `i = 0`
Döngü sonu	$0 \geq 1$ yanlış	çıktı: `I D B A`

Tablo 9.9: Şekil 9.18'deki dizi üzerinde indis hesapları ve `kokeYol(a, 9)` izlemesi (*n* = 9).



Şekil 9.19: Üstte E (*i = 5*) için çocuk indisleri 10 ve 11 dizinin dışına düşer; dizinin sağındaki kesikli hücreler "yok" anlamındadır. Altta `kokeYol(a, 9)` bitmiştir: I, D, B ve A sırayla yazdırılmıştır.

İzlemede iki nokta öne çıkar. Birincisi, bir indisin geçerli bir çocuk olup olmadığı dizi uzunluğuyla denetlenir: *n* düğümlü bir ağaçta $2i \le n$ ise sol çocuk vardır, $2i + 1 \le n$ ise sağ çocuk vardır. Bu denetim yapılmazsa dizinin sonundan sonraki bir hücre okunmaya çalışılır ve `ArrayIndexOutOfBoundsException` alınır. Tam olmayan ağaçlarda ayrıca hücrenin boş olup olmadığına da bakmak gerekir. İkincisi, kardeş düğümler aynı ebeveyni verir: 8 ve 9'un ikisi de 4'e gider. Tam sayı bölmesi, sağ çocuğun indisindeki fazladan 1'i siler.

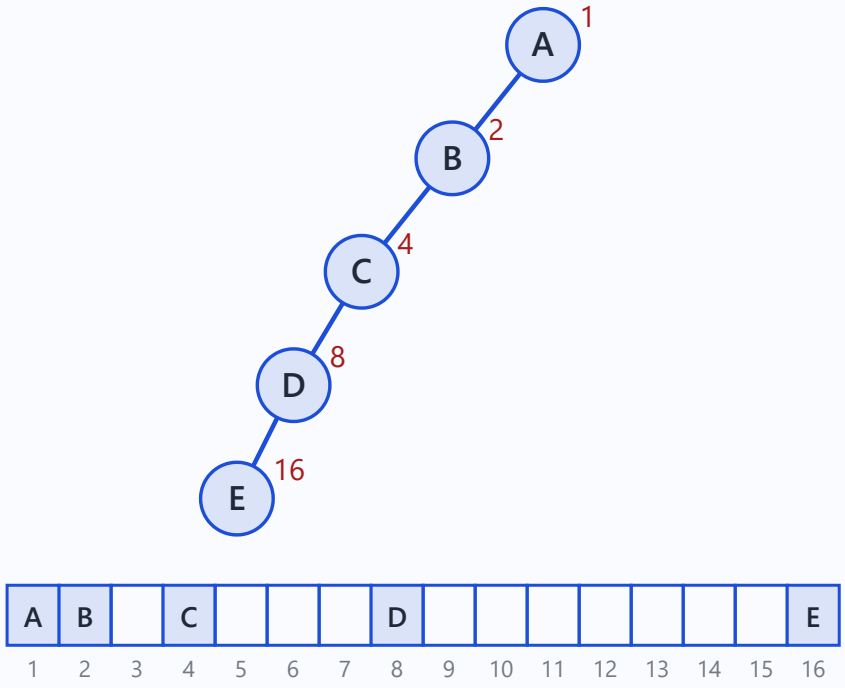
`kokeYol`'un her turu indisi yaklaşık yarıya indirir. Döngü, düğümün derinliğinin bir fazlası kadar döner. Tam ağaçta derinlik en çok $\lfloor \log_2 n \rfloor$ olduğundan maliyet $O(\log n)$'dir. Bu sayının küçüklüğü, heap işlemlerinin neden hızlı olduğunu da açıklar: bir öğeyi yukarı taşımak, bu döngüdeki gibi en çok yükseklik kadar adım sürer.

Sık yapılan hata

1 tabanlı ve 0 tabanlı formülleri karıştırmak. Kök `a[0]`'daysa sol çocuk $2i$ değil, $2i + 1$'dir; kökün "sol çocuğu" $2 \cdot 0 = 0$ çıkar ve kök kendini gösterir. 0 tabanlı dizide indis 6'daki düğümün ebeveyni $\lfloor (6 - 1) / 2 \rfloor = 2$, sol çocuğu $2 \cdot 6 + 1 = 13$'tür; 1 tabanlı formüllerle bulunan 3 ve 12 yanlıştır. Kod yazılmadan önce hangi düzenin kullanıldığı açıkça belirlenmelidir.

9.3.3 Dizide bellek israfı

Dizi gösteriminin zayıf yanı, ağacın şeklinden bağımsız olarak her konuma bir hücre ayırmasıdır. Bir düğümün indisi, düğümün ağaçtaki konumuyla belirlenir; öndeki konumların dolu olup olmadığına bakılmaz. Ağaç tam değilse aradaki boş konumlar dizide boş hücre olarak kalır.



Şekil 9.20: Sola çarpık beş düğümlü zincir ve dizi karşılığı. A indis 1'de, B 2'de, C 4'te, D 8'de, E 16'dadır. 16 hücrelik dizinin yalnız beşi doludur.

Şekil 9.20'deki zincirde her düğüm bir öncekinin sol çocuğudur, bu yüzden her adımda indis iki katına çıkar: 1, 2, 4, 8, 16. Beş düğüm için 16 hücre gerekir ve 11'i boş kalır. Zincir sağa çarpık olsaydı durum daha da kötüleşirdi. Sağ çocuk indisi $2i + 1$ olduğundan indisler 1, 3, 7, 15, 31 olur; beş düğüm için 31 hücre gerekir.

Genel olarak n düğümlü bir ikili ağaç, dizi gösteriminde $2^n - 1$ hücreye kadar yer isteyebilir. En kötü durum, yüksekliği $n - 1$ olan bir zincirdir; dizi bu yükseklikteki mükemmel ağacın bütün konumlarına, yani $2^n - 1$ hücreye yer açmak zorundadır. 30 düğümlü sağa çarpık bir zincir bir milyardan fazla hücre ister. Bu, üstel bir israftır ve dizi gösterimini şekli öngörülemeyen ağaçlar için kullanışsız yapar.

Artılar	Eksiler
Referans tutulmaz; ebeveyn ve çocuk indisleri $O(1)$'de hesaplanır.	Seyrek ya da çarpık ağaçta büyük israf: $2^n - 1$ hücreye kadar.
Tam ağaçlarda boşluk kalmaz; bellek en sıkı biçimde kullanılır.	Dizi boyutu sabittir; ağaç büyüyünce yeni ve daha büyük bir diziye kopyalama gerekir.

Tablo 9.10: Dizi gösteriminin artıları ve eksileri.

9.3.4 Bağlı gösterim

İkinci yol, her düğümü ayrı bir nesne yapmak ve çocuklarını referanslarla göstermektir. Bu, bağlı listedeki düğüm fikrinin ağaca uyarlanmış hâlidir. Bağlı listede her düğüm bir sonraki düğümü gösterir; ikili ağaçta ise iki düğümü gösterir: sol çocuğu ve sağ çocuğu.

```
public class AgacDugumu {
    int veri;
    AgacDugumu sol;   // sol çocuk (yoksa null)
    AgacDugumu sag;   // sağ çocuk (yoksa null)

    public AgacDugumu(int veri) {
        this.veri = veri;   // sol = sag = null
    }
}
```

`AgacDugumu` sınıfının üç alanı vardır. `veri` düğümde saklanan değeri, `sol` ve `sag` ise çocuk düğümleri gösterir. Bir çocuk yoksa ilgili referans `null`'dır. Yapıcı yalnız `veri`'yi atar. Java'da atanmayan referans alanları varsayılan olarak `null` olduğu için yeni oluşturulan her düğüm başlangıçta bir yapraktır.



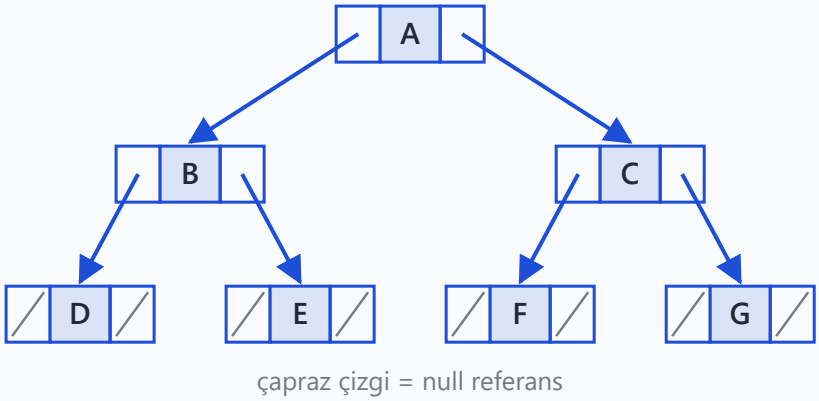
Şekil 9.21: Bağlı gösterimde bir ağaç düğümü üç alandan oluşur: sol referans, veri ve sağ referans. Sol referans sol alt ağacın kökünü, sağ referans sağ alt ağacın kökünü gösterir.

Küçük bir ağaç, düğümleri oluşturup referansları elle bağlayarak kurulabilir:

```
// 9 kökünde küçük bir ağaç kuralım
AgacDugumu kok = new AgacDugumu(9);
kok.sol = new AgacDugumu(2);
kok.sag = new AgacDugumu(3);
kok.sol.sol = new AgacDugumu(4);
```

İlk satır kökü, 9 değerli düğümü oluşturur. Sonraki iki satır kökün sol ve sağ çocuğunu, 2 ve 3'ü ekler. Son satır `kok.sol.sol` zinciriyle 2'nin sol çocuğuna 4'ü yerleştirir. Ortaya çıkan ağaç dört düğümlüdür: 9 kök; 2 ve 3 onun çocukları; 4, 2'nin sol çocuğudur. 2'nin sağ çocuğu ile 3'ün iki çocuğu yoktur.

Bağlı gösterimde bellek yalnız var olan düğümler için ayrılır. Ağaç ne kadar çarpık olursa olsun n düğüm n nesne demektir. Yeni bir düğüm eklemek, bir nesne oluşturup uygun referansı ona yönelt-mekten ibarettir; ekleme yeri biliniyorsa bu $O(1)$ sürer. Bedeli ise erişimdedir. Bir düğüme ulaşmak için kökten başlayıp referansları izlemek gerekir; indisle doğrudan atlamak mümkün değildir. Ebeveyne çıkmak da ancak düğüme ayrıca bir `ebeveyn` referansı eklenirse mümkündür.



Şekil 9.22: Bağlı gösterimde yedi düğümlü ağaç. Her düğümün sol ve sağ referans hücreleri çocuklarını gösterir; çapraz çizgiyle kapatılmış hücreler `null` referanslardır. Yaprakların (D, E, F, G) iki referansı da `null`'dır.

Şekil 9.22 referansların nasıl dağıldığını sayma fırsatı verir. Yedi düğümün her birinde iki referans alanı vardır: toplam 14. Bunların 6'sı doludur ve her biri bir kenara karşılık gelir; Teorem 9.1'e göre yedi düğümlü ağaçta 6 kenar vardır. Kalan 8 referans `null`'dır. Sayıların bu dağılımı rastlantı değildir.

Teorem 9.5 (Boş referans sayısı)

Sol ve sağ referanslı bağlı gösterimde, $n \geq 1$ düğümlü her ikili ağaçta tam olarak $n + 1$ tane `null` çocuk referansı vardır.

İspat. Her düğümde iki referans vardır; toplam $2n$. Dolu bir referans bir kenara karşılık gelir: ebeveynden çocuğa giden kenar. Kenar sayısı $n - 1$ olduğundan dolu referans sayısı da $n - 1$'dir. Geriye $2n - (n - 1) = n + 1$ boş referans kalır. □

Teoremin pratik anlamı şudur: bağlı ikili ağaçta referans alanlarının yarısından fazlası her zaman boştur. 50 düğümlü bir ağaçta 100 referansın 51'i `null`'dır. Bu boş alanları değerlendirmek için geliştirilmiş yapılar da vardır. Örneğin "kılavuzlu" (*threaded*) ikili ağaçlarda boş sol ve sağ referanslar, kök ortada sırasındaki önceki ve sonraki düğümü gösterecek biçimde kullanılır.

Sık yapılan hata

Bir düğümün alanlarına erişmeden önce `null` denetimini unutmak. `kok.sol.sol` ifadesi, `kok.sol` `null` ise `NullPointerException` fırlatır. Ağaç kodunda her referans izlenmeden önce `null` olup olmadığı düşünülmelidir. Özyinelemeli metotlarda bu denetim genellikle metodun ilk satırındaki taban durumla bir kez yapılır.

9.3.5 Dizi mi, bağlı gösterim mi?

İki gösterimin güçlü ve zayıf yanları birbirini tamamlar. Tablo 9.11 bunları yan yana koyar.

Ölçüt	Dizi gösterimi	Bağlı gösterim
Bellek	Tam ağaçta en az; seyrek ağaçta $2^n - 1$'e kadar israf	Düğüm başına 2 ek referans; israf yok
Çocuk / ebeveyn erişimi	$O(1)$, indis aritmetiğiyle	Çocuk $O(1)$; ebeveyn için ek referans gerekir
***i*. düzeydeki *k*. düğüm**	$O(1)$: indis $2^i + k - 1$ (1 tabanlı, $k \geq 1$)	Kökten gezinme gerekir
Büyüme	Dizi dolunca kopyalanarak büyütülür	Yeni düğüm eklemek $O(1)$ (konum biliniyorsa)
Uygun olduğu yer	Tam ikili ağaçlar (heap)	Genel ve değişken şekilli ağaçlar (BST, ifade ağacı)

Tablo 9.11: Dizi ve bağlı gösterimin karşılaştırılması.

Tablodaki üçüncü satır, dizi gösteriminin bir başka avantajını gösterir. i. düzeyin ilk düğümü 2^i indisindedir; o düzeydeki k. düğüm $2^i + k - 1$ indisine düşer. Şekil 9.18'deki ağaçta 2. düzeyin 3. düğümü $4 + 3 - 1 = 6$ indisindeki F'dir. Bağlı gösterimde aynı düğümü bulmak için düzey düzey gezinmek gerekir.

Seçim için kısa bir kural yeter: ağacın şekli her zaman tam kalıyorsa dizi, şekli öngörülemiyorsa bağlı gösterim tercih edilir. Heap, ekleme ve silmelerde şeklini tam tutar; bu yüzden dizide saklanır. İkili arama ağacı ise eklenen anahtarlara göre herhangi bir şekil alabilir; bağlı gösterim kullanılır.

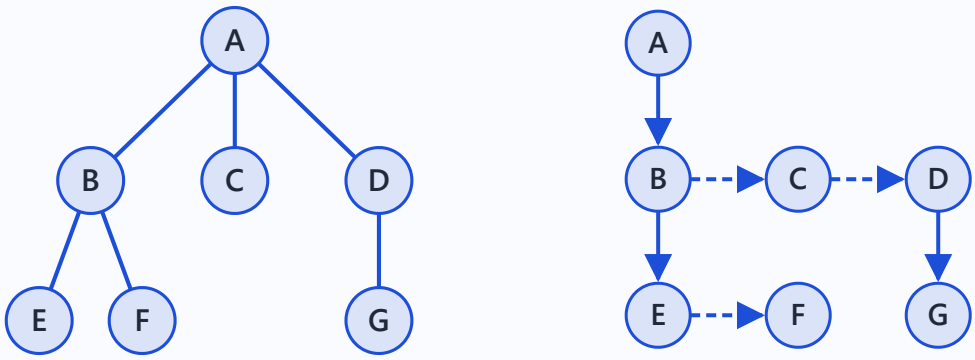
9.3.6 Genel ağaçların gösterimi

Çocuk sayısı sınırsız olduğunda düğüme sabit sayıda referans alanı koymak işe yaramaz. İki yaygın çözüm vardır.

```
// 1) Çocuk listesi
class GenelDugum {
    int veri;
    List<GenelDugum> cocuklar
        = new ArrayList<>();
}
// 2) İlk çocuk – sonraki kardeş
class KardesDugum {
    int veri;
    KardesDugum ilkCocuk;
    KardesDugum sonrakiKardes;
}
```

Birinci yolda (`GenelDugum`) her düğüm, çocuklarının referanslarını bir listede tutar. `ArrayList` gerektikçe büyüdüğü için çocuk sayısına sınır yoktur. `cocuklar` listesi üzerinde bir `for-each` döngüsü, ağaç ADT'sindeki `cocuklar(p)` metodunun doğrudan karşılığıdır. Bu gösterim sezgiseldir, ama her düğüm ayrı bir liste nesnesi taşır.

İkinci yolda (`KardesDugum`) her düğüm yalnız iki referans tutar: **ilk çocuk** (`ilkCocuk`) ve **sonraki kardeş** (`sonrakiKardes`). Bir düğümün bütün çocuklarına ulaşmak için önce ilk çocuğa inilir, sonra kardeş zinciri boyunca sağa yürünür. Çocuk listesi böylece düğümlerin kendisine dağıtılmış bir bağlı listeye dönüşür.



Şekil 9.23: Solda genel ağaç: A'nın çocukları B, C, D; B'nin çocukları E ve F; D'nin çocuğu G. Sağda aynı ağacın ilk çocuk (düz ok) – sonraki kardeş (kesikli ok) gösterimi.

Örnek 9.5

Şekil 9.23'teki ağacın ilk çocuk – sonraki kardeş gösteriminde her düğümün iki referansını yazalım.

1. **A.** İlk çocuk B; kardeşi yok (kök): `ilkCocuk = B` , `sonrakiKardes = null` .
2. **B, C, D.** Üçü A'nın çocuklarıdır ve kardeş zinciri B → C → D'dir. B'nin ilk çocuğu E, sonraki kardeşi C'dir. C'nin çocuğu yok, sonraki kardeşi D'dir. D'nin ilk çocuğu G, sonraki kardeşi yoktur.
3. **E, F, G.** E'nin çocuğu yok, sonraki kardeşi F'dir. F ve G'nin iki referansı da `null` 'dır.
4. **Sayım.** Yedi düğümde 14 referans alanı vardır. Dolu olanlar A→B, B→E, B→C, C→D, D→G, E→F'dir: 6 tane. Bu da *n − 1 = 6*'dır.

İkinci gösterimde her düğüm tam iki referans taşır. İlk çocuk referansı “sol”, sonraki kardeş referansı “sağ” sayılırsa her genel ağaç bir ikili ağaç olarak saklanabilir. Şekil 9.23'ün sağ yarısı, 45 derece döndürülerek bakıldığında bir ikili ağaçtır: A'nın sol çocuğu B; B'nin sol çocuğu E, sağ çocuğu C; C'nin sağ çocuğu D; E'nin sağ çocuğu F; D'nin sol çocuğu G. Bu dönüşüm, ikili ağaçlar için yazılmış birçok algoritmanın genel ağaçlara da uygulanabilmesini sağlar. Aynı dönüşümle bir orman da tek bir ikili ağaca çevrilebilir: ormandaki ağaçların kökleri birbirinin kardeşi sayılır.

İki gösterim arasında seçim yine kullanım biçimine bağlıdır. Bir düğümün *k*. çocuğuna sık sık doğrudan erişilecekse çocuk listesi uygundur: `cocuklar.get(k)` *O(1)* sürer. İlk çocuk – sonraki kardeş gösteriminde aynı çocuğa kardeş zinciri boyunca *k* adımda ulaşılır. Buna karşılık bu gösterim, her düğümde sabit ve küçük bir bellek kullanır ve çocuk eklemek için liste nesnesine ihtiyaç duymaz.

9.4 Ağaç gezinmeleri

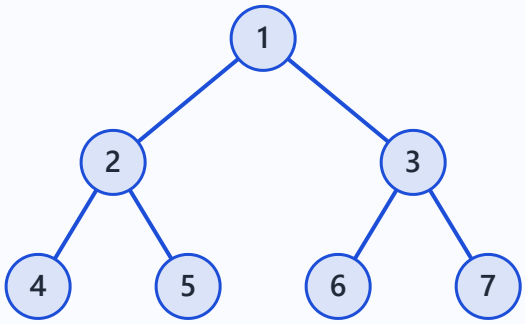
Dizide ya da bağlı listede bütün öğelerde gezinmenin tek doğal sırası vardır: baştan sona. Ağaçta ise böyle bir tek sıra yoktur. Bir düğüme gelindiğinde önce düğümün kendisi mi işlenecek, yoksa önce alt ağaçları mı? Alt ağaçlardan hangisi önce gezilecek? Yoksa düğümler düzey düzey mi taranacak? Bu soruların her yanıtı farklı bir gezinme verir ve her gezinme farklı bir problemin doğal çözümüdür.

9.4.1 Gezinme nedir?

Tanım 9.5

Bir ağacın **gezinmesi** (*traversal*), ağaçtaki her düğümü belirli bir sırayla tam olarak bir kez **ziyaret etmektir**. Ziyaret, düğüm üzerinde yapılacak işin genel adıdır: değeri yazdırmak, bir toplama eklemek, düğümü kopyalamak ya da silmek olabilir.

Gezinmeler iki aileye ayrılır. **Derinlik öncelikli** (*depth-first*) gezinmeler bir dalın sonuna kadar iner, sonra geri dönüp bir sonraki dala geçer. Bu ailede üç gezinme vardır ve aralarındaki tek fark, kökün alt ağaçlara göre *ne zaman* ziyaret edildiğidir. **Genişlik öncelikli** (*breadth-first*) gezinme ise ağacı düzey düzey tarar: önce kökü, sonra bütün çocukları, sonra bütün torunları ziyaret eder. Bu gezinmeye **düzey sırası** denir.



Şekil 9.24: Gezinmeler için örnek ağaç: 1'den 7'ye mükemmel ikili ağaç. 1 köktür; 2 ve 3 onun çocukları; 4, 5, 6, 7 yapraklardır.

Gezinme	Sıra	Çıktı	Tipik kullanım
Kök önde (*preorder*)	kök → sol → sağ	`1 2 4 5 3 6 7`	Ağacı kopyalamak, önek ifade
Kök ortada (*inorder*)	sol → kök → sağ	`4 2 5 1 6 3 7`	BST'yi sıralı gezmek
Kök sonda (*postorder*)	sol → sağ → kök	`4 5 2 6 7 3 1`	Silme, boyut ve yükseklik hesabı
Düzey sırası (*level-order*)	düzey düzey, soldan sağa	`1 2 3 4 5 6 7`	Düzeye dayalı işlemler

Tablo 9.12: Dört gezinme ve Şekil 9.24'teki ağaçta çıktıları.

Derinlik öncelikli üç gezinmenin adları, kökün sıradaki yerinden gelir: kök önde, ortada ya da sonda. Üçünde de sol alt ağaç sağ alt ağaçtan önce gezilir. Değişen tek şey kökün bu iki alt ağaca göre konumudur. Gezinmeler, ikili ağacın özyinelemeli tanımını izler: kökte ne yapılacağı söylenir, alt ağaçlar için aynı gezinme yeniden uygulanır. Bu yüzden bir gezinmenin çıktısını elle bulurken her alt ağaç, kendi başına küçük bir ağaç gibi ele alınır.

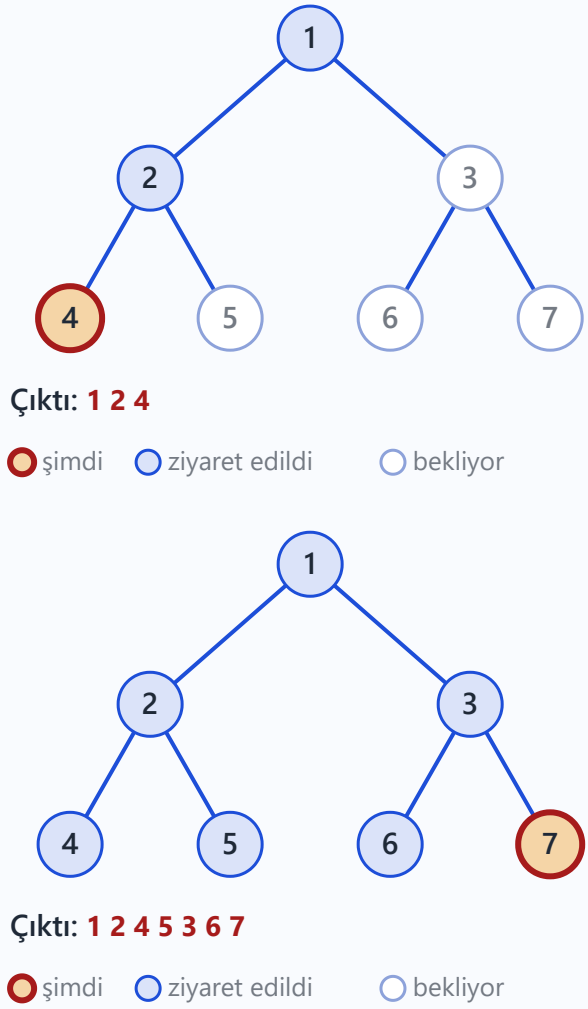
9.4.2 Kök önde gezinme

Kök önde (*preorder*) gezinme üç adımdan oluşur:

1. Kökü ziyaret et.
2. Sol alt ağacı kök önde gez.
3. Sağ alt ağacı kök önde gez.

```
ziyaret(kok);
kokBasta(kok.sol);
kokBasta(kok.sag);
```

Bir düğüm, altındaki bütün düğümlerden önce ziyaret edilir. Çıktıda her düğüm, kendi soyunun hepsinden önce gelir. Şekil 9.24'teki ağaçta adımlar şöyledir. Kök 1 ziyaret edilir. Sonra sol alt ağaca, 2 köklü ağaca geçilir ve onun da kökü, 2, hemen ziyaret edilir. 2'nin sol alt ağacı tek düğümlü 4'tür: 4 ziyaret edilir. 2'nin sağ alt ağacı 5'tir: 5 ziyaret edilir. Böylece 1'in sol alt ağacı biter ve sağ alt ağaca geçilir: 3, ardından 3'ün sol çocuğu 6 ve sağ çocuğu 7. Çıktı `1 2 4 5 3 6 7`'dir.



Şekil 9.25: Kök önde gezinmenin iki anı. Solda 1 ve 2'den sonra yaprak 4 ziyaret edilmiştir; çıktı `1 2 4`. Sağda gezinme bitmiştir; çıktı `1 2 4 5 3 6 7`.

Kök önde gezinmenin tipik kullanım alanları, düğümün çocuklarından önce işlenmesinin gerektiği durumlardır. **Ağacı kopyalamak** bunun en açık örneğidir: bir düğümün kopyası oluşturulmadan onun çocuklarının kopyaları bağlanamaz. **Ağacı bir diziye ya da dosyaya yazmak** da kök önde sırayla yapılır; dosyayı okuyan kod, her düğümü ebeveyninden sonra gördüğü için ağacı aynı sırayla yeniden kurabilir. İçindekiler sayfası da bir kök önde gezinmedir: bölüm başlığı alt başlıklarından önce yazılır. İfade ağaçlarında kök önde gezinme **önek** (*prefix*) gösterimi verir; işleç, işlenenlerinden önce yazılır.

9.4.3 Kök ortada gezinme

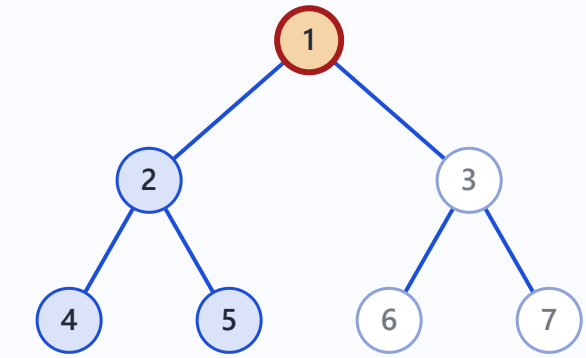
Kök ortada (*inorder*) gezinmede kök, iki alt ağacın arasında ziyaret edilir:

1. Sol alt ağacı kök ortada gez.
2. Kökü ziyaret et.

3. Sağ alt ağacı kök ortada gez.

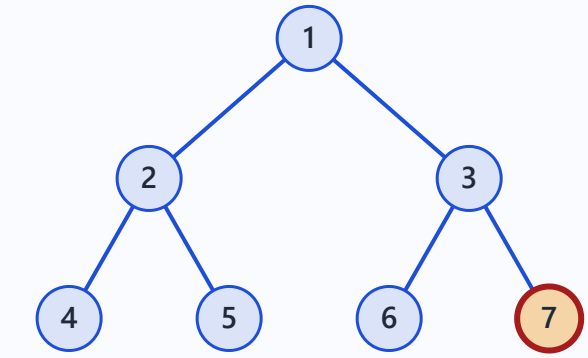
```
kokOrtada(kok.sol);
ziyaret(kok);
kokOrtada(kok.sag);
```

Bir düğüm, sol alt ağacındaki bütün düğümlerden sonra ve sağ alt ağacındakilerin hepsinden önce ziyaret edilir. Bu yüzden çıktı kökle başlamaz; kökten sola sola inilir ve ilk ziyaret edilen düğüm en soldaki düğüm olur. Şekil 9.24'teki ağaçta 1'den 2'ye, 2'den 4'e inilir. 4'ün sol alt ağacı boştur; 4 ziyaret edilir. 4'ün sağ alt ağacı da boştur ve 2'ye dönülür. 2'nin sol alt ağacı bittiği için 2 ziyaret edilir, ardından sağ alt ağacı olan 5. 1'in sol alt ağacı tamamlanmıştır; şimdi 1 ziyaret edilir. Sağ alt ağaçta yine en sola inilir: 6, sonra 3, sonra 7. Çıktı `4 2 5 1 6 3 7` 'dir.



Çıktı: **4 2 5 1**

şimdi ziyaret edildi bekliyor



Çıktı: **4 2 5 1 6 3 7**

şimdi ziyaret edildi bekliyor

Şekil 9.26: Kök ortada gezinmenin iki anı. Solda sol alt ağaç (4, 2, 5) bitmiş ve kök 1 ziyaret edilmiştir. Sağda gezinme bitmiştir; çıktı `4 2 5 1 6 3 7`.

Kök ortada çıktısının dikkat çekici bir geometrik anlamı vardır: düğümler, ağaç çiziminde soldan sağa göründükleri sırayla yazılır. Ağacın yatay eksene izdüşümü alınsa, yani her düğüm dikey olarak aşağıya "düşürülse", sıralama kök ortada çıktısını verir. Şekil 9.24'te soldan sağa sıra 4, 2, 5, 1, 6, 3, 7'dir.

Bu gezinmenin en önemli kullanımı **ikili arama ağaçlarındadır**. İkili arama ağacında her düğümün solundaki anahtarlar küçük, sağındakiler büyüktür. Kök ortada gezinme önce küçükleri, sonra düğümü, sonra büyükleri yazar. Bu her alt ağaçta tekrarlandığı için anahtarlar **küçükten büyüğe sıralı** çıkar. Tersi de doğrudur: kök ortada çıktısı sıralı olan bir ikili ağaç, (farklı anahtarlar için) bir ikili arama ağacıdır. İfade ağaçlarında kök ortada gezinme **ara ek** (*infix*) gösterimi verir; ama parantezler kaybolur.

> **Sık yapılan hata**
>
> Kök ortada gezinmeye kökten başlamak. Kök, sol alt ağacının bütün düğümleri yazıldıktan sonra yazılır; ilk yazılan düğüm, kökten sola sola inilerek ulaşılan en soldaki düğümdür. En soldaki düğümün sağ alt ağacı olabileceğini unutmak da sık görülür. Bu düğüm yazıldıktan sonra sağ alt ağacına geçilir ve orada yine en sola inilir.

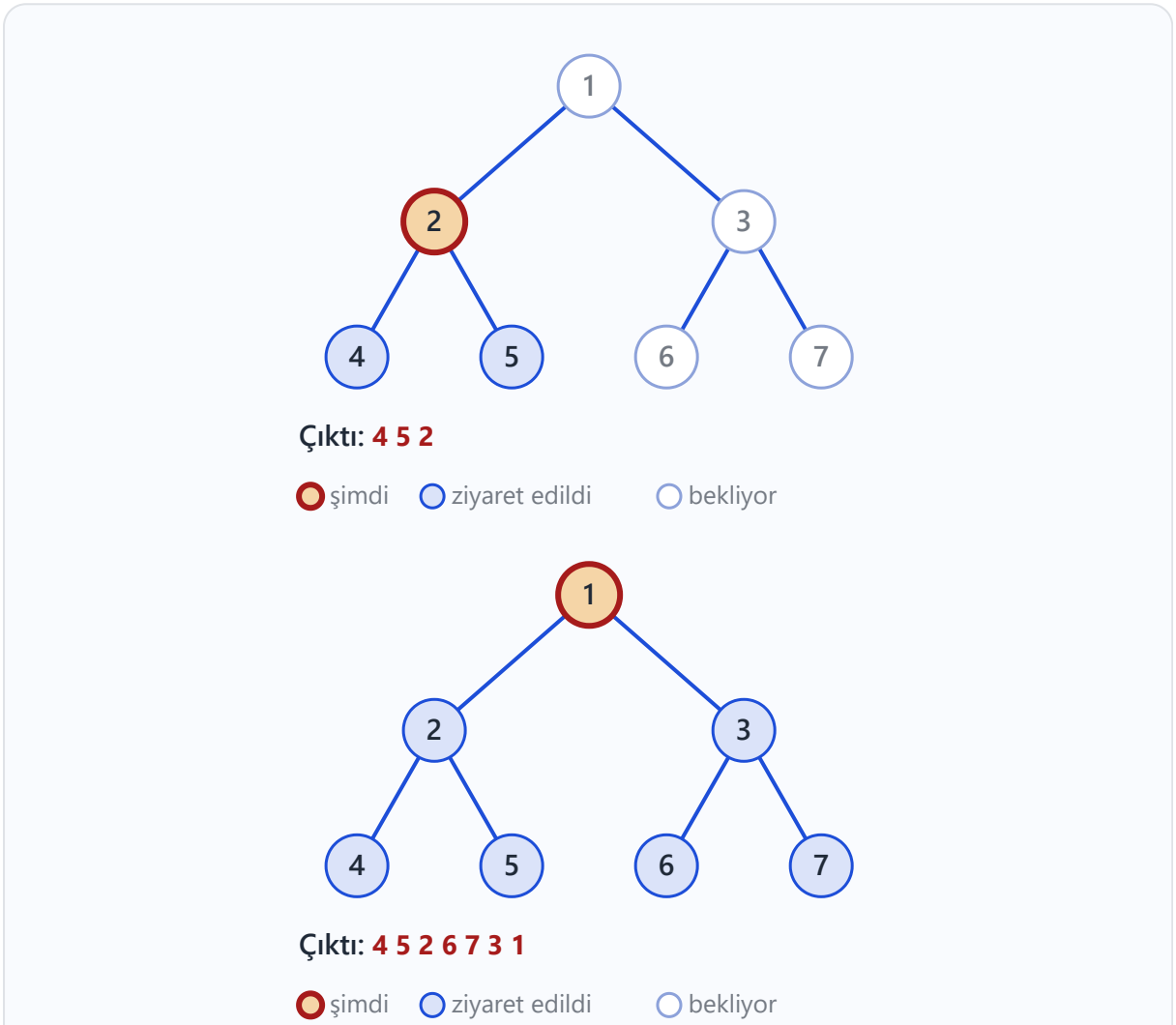
9.4.4 Kök sonda gezinme

Kök sonda (*postorder*) gezinmede kök, iki alt ağacın ikisi de bittikten sonra ziyaret edilir:

1. Sol alt ağacı kök sonda gez.
2. Sağ alt ağacı kök sonda gez.
3. Kökü ziyaret et.

```
kokSonda(kok.sol);
kokSonda(kok.sag);
ziyaret(kok);
```

Her düğüm, bütün soyundan sonra ziyaret edilir. Kök en son yazılır. Şekil 9.24'teki ağaçta önce en soldaki yaprak 4 ziyaret edilir. Sonra 2'nin sağ çocuğu 5 gelir. 2'nin iki alt ağacı da bittiği için 2 yazılır. 1'in sağ alt ağacına geçilir: önce 6, sonra 7, sonra 3. En sonda kök 1 yazılır. Çıktı `4 5 2 6 7 3 1`'dir.



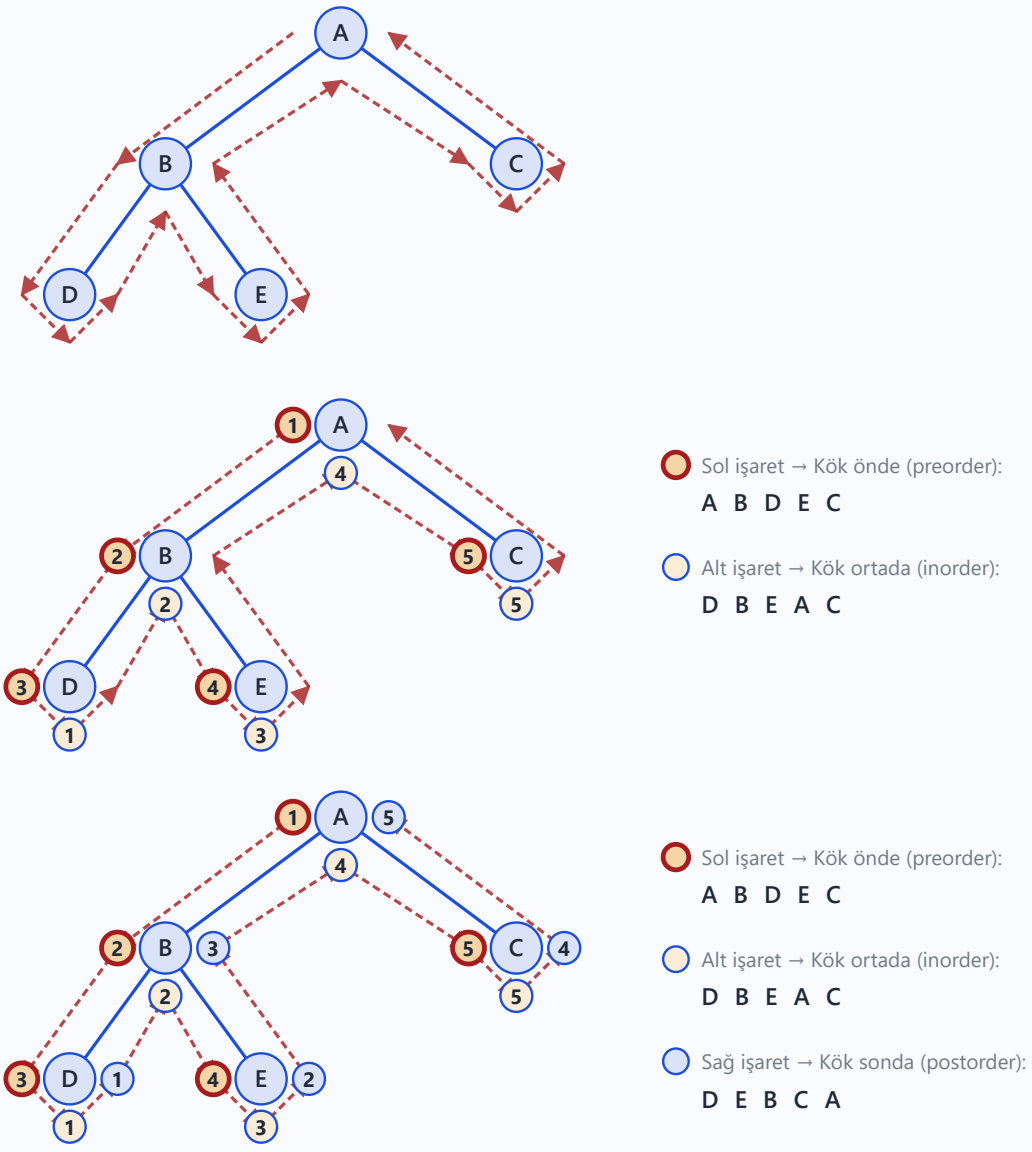
Şekil 9.27: Kök sonda gezinmenin iki anı. Solda 4 ve 5'ten sonra 2 ziyaret edilmiştir; çıktı `4 5 2`. Sağda kök en son ziyaret edilmiştir; çıktı `4 5 2 6 7 3 1`.

Kök sonda gezinme, bir düğümün işinin çocuklarının sonucuna bağlı olduğu her durumda kullanılır. **Ağacı silmek** bunun tipik örneğidir: önce çocuklar silinmelidir, yoksa ebeveyn silindiğinde çocuklara ulaşan referanslar kaybolur. Dosya sisteminde bir klasörü silmek de böyle çalışır; boş olmayan bir klasör silinemez, önce içindekiler silinir. Bir klasörün diskte kapladığı toplam alanı hesaplamak da kök sondadır: önce alt klasörlerin boyutları bulunur, sonra toplanır. Ağacın **düğüm sayısı** ve **yüksekliği** de aynı mantıkla hesaplanır. İfade ağaçlarında kök sonda gezinme **sonek** (*postfix*) gösterimi verir; ifadenin değeri de bu sırayla hesaplanır.

Üç gezinmenin çıktıları arasında kolay denetlenebilir bir ilişki vardır. Kök önde çıktısının *ilk* öğesi köktür; kök sonda çıktısının *son* öğesi köktür; kök ortada çıktısında kök, sol ve sağ alt ağaçların arasında durur. Tablo 9.12'deki çıktılarda bu kolayca görülür: 1, kök önde çıktısının başında, kök sonda çıktısının sonunda ve kök ortada çıktısının tam ortasındadır. Bu gözlem, iki gezinme çıktısından ağacı yeniden kurmanın da temelidir.

9.4.5 Üç gezinme, tek tur: Euler turu

Derinlik öncelikli üç gezinmeyi tek bir resimde birleştirmenin zarif bir yolu vardır. Ağacın çevresinde, kökün solundan başlayıp kenarlara sol elimizle dokunarak yürüdüğümüzü düşünelim. Bu yürüyüşe **Euler turu** (*Euler tour*) denir. Tur ağacın bütün kenarlarını iki kez geçer: bir kez inerken, bir kez çıkarken. Her düğüme de üç kez uğranır: önce solundan, sonra altından, sonra sağından.



Şekil 9.28: Beş düğümlü ağaçta (A kök; B'nin çocukları D ve E; C, A'nın sağ çocuğu) Euler turu. Üstte turun yolu. Ortada soldan geçişler (kök önde: A B D E C) ve alttan geçişler (kök ortada: D B E A C) numaralanmıştır. Altta sağdan geçişler (kök sonda: D E B C A) eklenmiştir.

Bir düğümü hangi uğrayışta yazdığımıza göre üç gezinmeden biri ortaya çıkar.

- Düğümü **solundan** geçerken, yani ona ilk kez varıldığında yazarsak **kök önde** sırası çıkar: `A B D E C`.

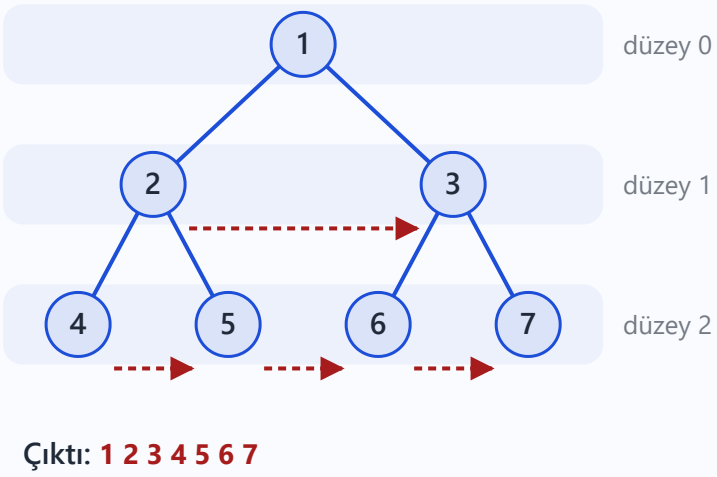
- **Altından** geçerken, yani sol alt ağacı bitmiş ve sağ alt ağaca henüz başlanmamışken yazarsak **kök ortada** sırası çıkar: `D B E A C`.
- **Sağından** geçerken, yani iki alt ağaç da bitmişken yazarsak **kök sonda** sırası çıkar: `D E B C A`.

Yaprakların üç uğrayışı art arda gelir, çünkü yaprağın alt ağaçları boştur. Bu yüzden yapraklar üç gezinmede de aynı göreli sırada çıkar: D, E, C. Bir gezinmeden ötekine yalnız iç düğümlerin yeri değişir. Euler turu, üç gezinmenin aslında *aynı turun farklı anları* olduğunu gösterir. Tur her kenarı iki kez geçtiği için *n* düğümlü ağaçta *2(n − 1)* kenar adımı sürer; maliyet *O(n)*'dir. Bu, üç gezinmenin de *O(n)* olduğunun bir başka açıklamasıdır.

Euler turu fikri gezinmeleri birleştirmekten fazlasına yarar. Bazı işler, bir düğümün hem öncesinde hem sonrasında bir şey yapmayı gerektirir. Örneğin bir ifade ağacını tam parantezli yazarken bir alt ağaca girerken "(" açılır, alttan geçerken işleç yazılır, çıkarken ")" kapanır. Bu, üç uğrayışın üçünün de kullanıldığı bir Euler turudur.

9.4.6 Düzey sırası gezinme

Düzey sırası (*level-order*) gezinme kökten başlar, her düzeyi soldan sağa tarar, sonra bir alt düzeye iner. Çizgelerdeki **genişlik öncelikli aramanın** (*breadth-first search*, BFS) ağaçtaki hâlidir.



Şekil 9.29: Düzey sırası gezinme: 0. düzeyde 1; 1. düzeyde 2, 3; 2. düzeyde 4, 5, 6, 7. Kesikli oklar tarama yönünü gösterir; çıktı `1 2 3 4 5 6 7`.

Derinlik öncelikli gezinmelerden farklı olarak düzey sırası, alt ağaçların yapısını izlemez. 2'yi ziyaret ettikten sonra onun çocuğu 4'e değil, kardeşi 3'e geçer. 2'nin çocukları ancak bütün 1. düzey bittikten sonra gelir. Bu yüzden düzey sırası doğal biçimde özyinelemeyle yazılamaz. Özyineleme, bir alt ağacı bitirmeden ötekine geçmez; düzey sırası ise iki alt ağaç arasında sürekli gidip gelir.

Çözüm bir **kuyruktur** (*queue*). Kuyruk, önce gelenin önce çıktığı FIFO (*first-in, first-out*) düzeniyle çalışır. Bir düğüm ziyaret edilirken çocukları kuyruğun arkasına eklenir. Önce eklenen düğümlerin çocukları da önce işlenir. Bir düzeyin bütün düğümleri, bir sonraki düzeyin düğümlerinden önce kuyruğa girdiği için düzeyler sırasıyla taranır.

Düzey sırasının kullanım alanları, düzeylerin anlam taşıdığı işlerdir. Ağacı düzey düzey yazdırmak, her düzeydeki değerlerin toplamını ya da en büyüğünü bulmak, köke en yakın yaprağı bulmak ve ağacın en geniş düzeyini (genişliğini) hesaplamak bunlara örnektir. Köke en yakın yaprak için düzey sırası en uygun yöntemdir. Kuyruktan çıkan ilk yaprak, köke en yakın olanlardan biridir; daha derindeki düğümlere hiç inmeden gezinme durdurulabilir.

> **Sık yapılan hata**
>
> Düzey sırasını kök önde ile karıştırmak. İki gezinme de kökle başlar ve Şekil 9.24'teki mükemmel ağaçta ilk üç öğeleri bile aynı değildir: kök önde `1 2 4`, düzey sırası `1 2 3`. Kök önde bir dala derinlemesine iner; düzey sırası önce kardeşleri bitirir. Düzey sırasını yığınla yazmaya çalışmak da aynı karışıklığın kod hâlidir: yığın kullanılırsa çıktı bir derinlik öncelikli gezinmeye döner.

9.5 Gezinmelerin kodu

Gezinmelerin tanımları özyinelemelidir ve en kısa kodları da özyinelemeli yazılır. Ama özyinelemenin arkasında görünmeyen bir yapı çalışır: **çağrı yığını** (*call stack*). Her metot çağrısı yığına bir **çerçeve** (*stack frame*) ekler; çerçeve, çağrının parametrelerini, yerel değişkenlerini ve metodun döndükten sonra nereden devam edeceğini tutar. Bu kesim önce üç özyinelemeli gezinmeyi çağrı yığını üzerinde izler. Ardından aynı gezinmeleri, çağrı yığınının yerine açık bir `Deque` yığını kullanarak özyinelemesiz yazar. Son olarak düzey sırasını bir kuyrukla kodlar.

Bu kesimdeki bütün kodlar bağlı gösterimdeki `AgacDugumu` sınıfını kullanır. Özyinelemeli izlemeler 9 kökündeki dört düğümlü ağaç üzerinde yapılır: 9'un çocukları 2 ve 3; 2'nin sol çocuğu 4. Bu ağacın yüksekliği 2'dir.

9.5.1 Özyinelemeli kök önde ve çağrı yığını

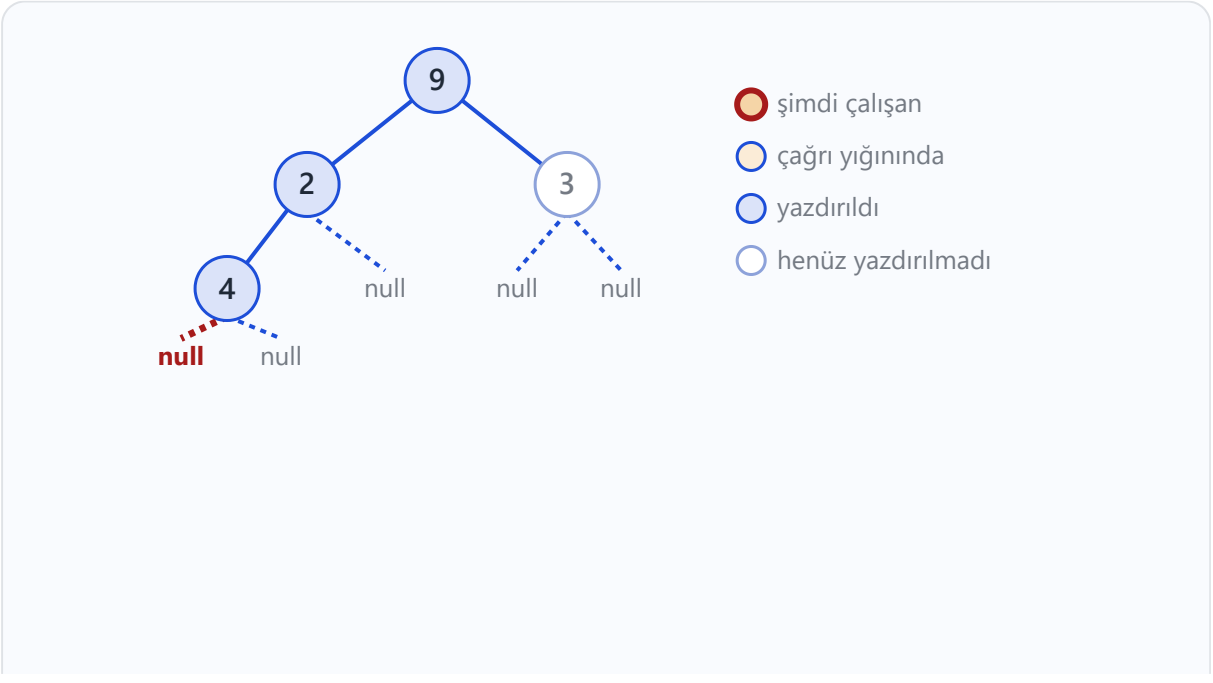
```
public void kokBasta(AgacDugumu kok) {
  if (kok == null) {
    return;
  }
  System.out.print(kok.veri + " ");
  kokBasta(kok.sol);
  kokBasta(kok.sag);
}
```

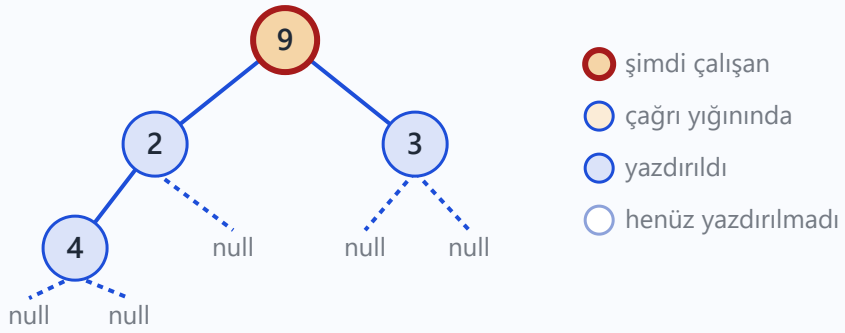
Metodun ilk iş olarak yaptığı şey, ağacın boş olup olmadığını denetlemektir. `kok == null` ise boş bir alt ağaçtayızdır ve yapılacak bir şey yoktur; metot hemen döner. Bu **taban durumdur**. Ağaç boş değilse önce kökün değeri yazdırılır (satır 5), sonra sol alt ağaç (satır 6), en son sağ alt ağaç (satır 7) için aynı metot çağrılır. Satır 8'deki kapanış parantezine gelindiğinde çağrı biter ve çerçevesi yığından çıkar.

Taban durumun `null` ile tanımlanması kodu sadeleştirir. Yaprak için özel bir durum yazılmaz: yaprağın iki çocuğu `null`'dır ve iki özyinelemeli çağrı hemen döner. Bunun bedeli, her `null` çocuk için bir çağrı daha yapılmasıdır. Teorem 9.5'e göre n düğümlü ağaçta $n + 1$ `null` referans vardır; toplam çağrı sayısı $n + (n + 1) = 2n + 1$'dir. Dört düğümlü ağaçta 9 çağrı yapılır.

Adım	Çağrı yığını	Olay	Çıktı
1	kokBasta(9)	9 yazdırılır	9
2	9 · s.6 | kokBasta(2)	2 yazdırılır	9 2
3	9 · s.6 | 2 · s.6 | kokBasta(4)	4 yazdırılır	9 2 4
4	… | 4 · s.6 | kokBasta(null)	taban durum, return	9 2 4
5	… | 4 · s.7 | kokBasta(null)	taban durum, return	9 2 4
6	9 · s.6 | 2 · s.6 | kokBasta(4)	4 biter (satır 8)	9 2 4
7	9 · s.6 | 2 · s.7 | kokBasta(null)	2'nin sağı boş	9 2 4
8	9 · s.6 | kokBasta(2)	2 biter	9 2 4
9	9 · s.7 | kokBasta(3)	3 yazdırılır	9 2 4 3
10–11	9 · s.7 | 3 · s.6 / s.7 | kokBasta(null)	3'ün iki çocuğu boş	9 2 4 3
12	9 · s.7 | kokBasta(3)	3 biter	9 2 4 3
13	kokBasta(9)	9 biter, yığın boşalır	9 2 4 3

Tablo 9.13: `kokBasta(9)` izlemesi. Çağrı yığını alttan üste; "s.6" ve "s.7", çağrının yapıldığı satırdır.





Şekil 9.30: Özyinelemeli kök önde. Üstte 4. adım: `kokBasta(4.sol)` taban durumdadır; 9, 2 ve 4 çerçeveleri yığında bekler. Altta son adım: bütün düğümler yazdırılmıştır, çıktı `9 2 4 3`.

İzlemenin en önemli gözlemi, çağrı yığınının derinliğiyle ilgilidir. 4. ve 5. adımlarda yığında dört çerçeve vardır: 9, 2, 4 ve 4'ün `null` çocuğu. Bu, izlemedeki en büyük derinliktir. Yığın derinliği, kökten en derin `null` çocuğa inen yolla belirlenir. Yükseklik *h* ise bu yolda *h + 1* düğüm ve bir `null` çağrısı vardır; yığında en çok *h + 2* çerçeve bulunur. Burada *h = 2* ve en çok *2 + 2 = 4* çerçeve görülür.

Zaman maliyeti *O(n)*'dir: her düğüm için bir çağrı ve her `null` referans için bir çağrı yapılır; her çağrı sabit iş yapar. Ek bellek, yığın derinliği kadardır: *O(h)*. Dengeli ağaçta bu *O(log n)*, çarpık ağaçta *O(n)*'dir.

9.5.2 Özyinelemeli kök ortada

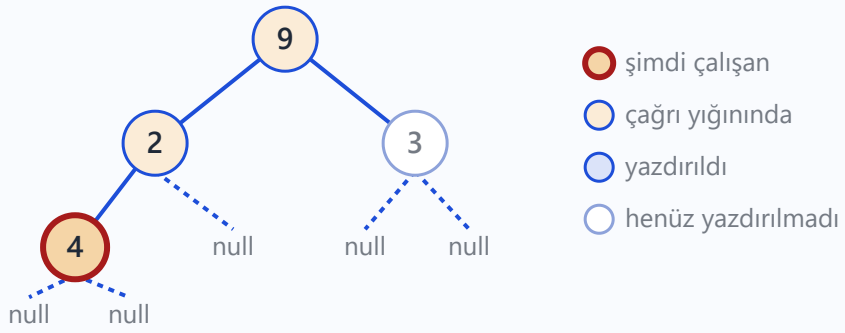
```
public void kokOrtada(AgacDugumu kok) {
  if (kok == null) {
    return;
  }
  kokOrtada(kok.sol);
  System.out.print(kok.veri + " ");
  kokOrtada(kok.sag);
}
```

Kod kök önde ile aynıdır; tek fark yazdırma satırının iki özyinelemeli çağrının arasına alınmasıdır. Bu küçük değişiklik, değerin hangi anda yazılacağını değiştirir. Bir çağrı başladığında değer hemen yazılmaz; önce sol alt ağaç bütünüyle gezilir. Değer, sol çağrı döndükten sonra, satır 6'da yazılır.

Adım	Çağrı yığını	Olay	Çıktı
1–3	9 · s.5 | 2 · s.5 | kokOrtada(4)	9, 2, 4 çağrılır; hiçbiri henüz yazılmaz	—
4	… | 4 · s.5 | kokOrtada(null)	4'ün solu boş	—
5	9 · s.5 | 2 · s.5 | kokOrtada(4)	sol bitti: 4 yazdırılır	4
6–7	… | 4 · s.7 | kokOrtada(null)	4'ün sağı boş; 4 biter	4
8	9 · s.5 | kokOrtada(2)	sol bitti: 2 yazdırılır	4 2

Adım	Çağrı yığını	Olay	Çıktı
9–10	9 · s.5 | 2 · s.7 | kokOrtada(null)	2'nin sağı boş; 2 biter	4 2
11	kokOrtada(9)	sol bitti: 9 yazdırılır	4 2 9
12–13	9 · s.7 | 3 · s.5 | kokOrtada(null)	3 çağrılır; 3'ün solu boş	4 2 9
14	9 · s.7 | kokOrtada(3)	3 yazdırılır	4 2 9 3
15–17	kokOrtada(9)	3'ün sağı boş; 3 ve 9 biter	4 2 9 3

Tablo 9.14: `kokOrtada(9)` izlemesi. Sol çağrılar satır 5'ten, sağ çağrılar satır 7'den yapılır.



Şekil 9.31: Özyinelemeli kök ortada, 5. adım: 4'ün sol alt ağacı (boş) bitmiş ve 4 yazdırılmıştır. 2 ve 9 çerçeveleri, sol çağrılarının dönmesini beklemektedir.

Çıktı `4 2 9 3`'tür. Kök 9, sol alt ağacının (4 ve 2) arkasından, sağ alt ağacının (3) önünden gelir. İzlemede 9 numaralı çerçevenin ömrüne bakmak öğreticidir. Çerçeve ilk adımda oluşur, ama 9 ancak 11. adımda yazılır. Arada çerçeve, satır 5'teki çağrının dönmesini bekler. Yığındaki bir çerçeve, "burada yarım kalmış bir iş var" bilgisini taşır.

9.5.3 Özyinelemeli kök sonda

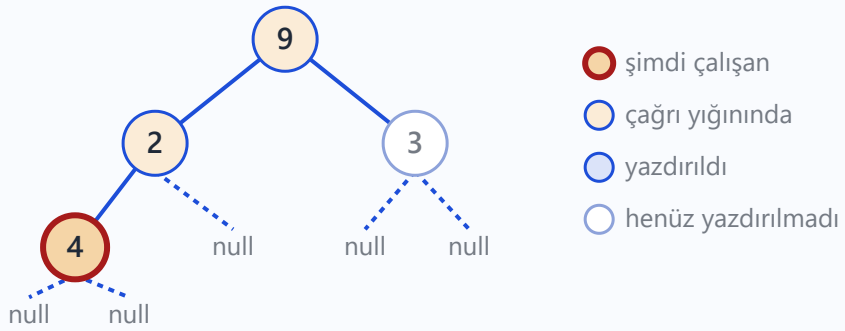
```
public void kokSonda(AgacDugumu kok) {
  if (kok == null) {
    return;
  }
  kokSonda(kok.sol);
  kokSonda(kok.sag);
  System.out.print(kok.veri + " ");
}
```

Bu kez yazdırma iki çağrının arkasına, satır 7'ye alınmıştır. Sol çağrı satır 5'ten, sağ çağrı satır 6'dan yapılır. Bir düğümün değeri, iki alt ağacının da tamamen gezilmesinden sonra yazılır.

Adım	Çağrı yığını	Olay	Çıktı
1–3	9 · s.5 | 2 · s.5 | kokSonda(4)	9, 2, 4 çağrılır	—

Adım	Çağrı yığını	Olay	Çıktı
4	… | 4 · s.5 | kokSonda(null)	4'ün solu boş	—
5	… | 4 · s.6 | kokSonda(null)	4'ün sağı boş	—
6–7	9 · s.5 | 2 · s.5 | kokSonda(4)	iki alt ağaç bitti: 4 yazdırılır; 4 biter	4
8	9 · s.5 | 2 · s.6 | kokSonda(null)	2'nin sağı boş	4
9–10	9 · s.5 | kokSonda(2)	2 yazdırılır; 2 biter	4 2
11–13	9 · s.6 | 3 · s.5 / s.6 | kokSonda(null)	3 çağrılır; iki çocuğu boş	4 2
14–15	9 · s.6 | kokSonda(3)	3 yazdırılır; 3 biter	4 2 3
16–17	kokSonda(9)	9 yazdırılır; yığın boşalır	4 2 3 9

Tablo 9.15: `kokSonda(9)` izlemesi. Sol çağrılar satır 5'ten, sağ çağrılar satır 6'dan yapılır.



Şekil 9.32: Özyinelemeli kök sonda, 6. adım: 4'ün iki alt ağacı da (ikisi de boş) gezilmiş ve 4 yazdırılmıştır. 9 ve 2 henüz yazdırılmamıştır.

Çıktı `4 2 3 9`'dur. Üç izleme yan yana konduğunda çağrı sırasının hiç değişmediği görülür. Üç metotta da çağrılar aynı sırayla yapılır ve aynı sırayla döner: 9, 2, 4, `null`, `null`, … Değişen tek şey, yazdırmanın bu sıranın hangi noktasında yapıldığıdır. Bu, Euler turu gözleminin kod düzeyindeki karşılığıdır. Çağrının başlangıcı düğümün solundan geçişe, iki çağrının arası altından geçişe, çağrının sonu sağından geçişe karşılık gelir.

Sık yapılan hata

Taban durumu unutmak. `if (kok == null) return;` satırı olmadan ilk yaprakta `kokBasta(kok.sol)` `null` ile çağrılır ve `kok.veri` okunurken `NullPointerException` fırlatılır. Taban durumu yazdırma satırından *sonraya* koymak da aynı hatayı verir. Her özyinelemeli ağaç metodu, düğümün alanlarına dokunmadan önce boş ağaç durumunu ele almalıdır.

9.5.4 Yığınla kök önde

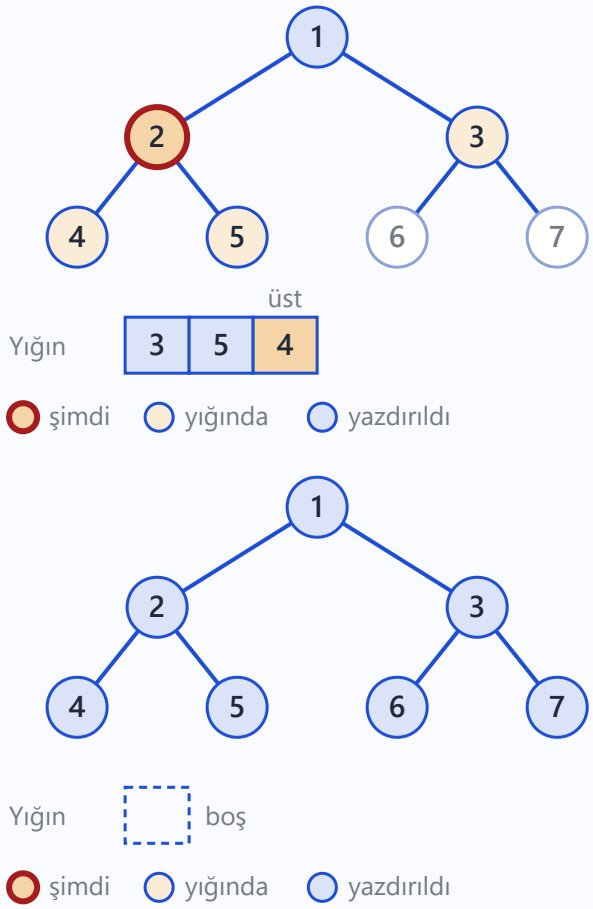
Özyineleme, çağrı yığınını sistemin kendisine bırakır. Çağrı yığınının boyutu ise sınırlıdır; Java'da varsayılan ayarlarla birkaç bin ile birkaç on bin çerçeve arasında bir derinlikte `StackOverflowError` alınır. Çok derin, örneğin yüz binlerce düğümlü bir zincir biçimindeki ağaçta özyinelemeli gezinme çöker. Çözüm, çağrı yığınının yaptığı işi `java.util.Deque` ile açık bir yığına yaptırmaktır. Bu yığın heap bellekte yaşar ve ancak bellek bitince dolar.

```
public void kokBastaYigin(AgacDugumu kok) {
  if (kok == null) return;
  Deque<AgacDugumu> yigin = new ArrayDeque<>();
  yigin.push(kok);
  while (!yigin.isEmpty()) {
    AgacDugumu gecici = yigin.pop();
    System.out.print(gecici.veri + " ");
    if (gecici.sag != null)
      yigin.push(gecici.sag);  // önce sağ
    if (gecici.sol != null)
      yigin.push(gecici.sol);  // sol üstte
  }
}
```

Kök önde, açık yığınla yazılması en kolay gezinmedir. Bir düğüm yığından çıkarıldığı anda yazdırılabilir, çünkü kök önde gezinmede düğüm çocuklarından önce gelir. Döngünün her turu üst düğümü çıkarır (`pop`), değerini yazdırır, sonra çocuklarını yığına iter (`push`). İtme sırası önemlidir: **önce sağ, sonra sol**. Yığın LIFO çalıştığı için en son itilen sol çocuk üstte kalır ve bir sonraki turda önce o çıkar. Böylece sol alt ağaç bütünüyle gezildikten sonra sağ çocuk sıraya gelir.

Tur	pop	İtilenler	Tur sonunda yığın	Çıktı
başlangıç	—	1	[1]	—
1	1	3, 2	[3, 2]	1
2	2	5, 4	[3, 5, 4]	1 2
3	4	—	[3, 5]	1 2 4
4	5	—	[3]	1 2 4 5
5	3	7, 6	[7, 6]	1 2 4 5 3
6	6	—	[7]	1 2 4 5 3 6
7	7	—	[]	1 2 4 5 3 6 7

Tablo 9.16: Şekil 9.24'teki ağaçta `kokBastaYigin` izlemesi. Yığın alttan üste yazılmıştır.



Şekil 9.33: Yığınla kök önde. Solda 2. tur bitmiştir: 1 ve 2 yazdırılmış, yığında alttan üste 3, 5, 4 vardır. Sağda yığın boşalmış ve döngü bitmiştir; çıktı `1 2 4 5 3 6 7`, özyinelemeli sürümle aynıdır.

Yığının içeriği, özyinelemedeki "yarım kalmış işler"in listesidir. 2. turun sonunda yığında 3, 5, 4 vardır. Bu, "önce 4'ün alt ağacı, sonra 5'in alt ağacı, sonra 3'ün alt ağacı gezilecek" demektir. Özyinelemeli sürümde aynı bilgi, 2 ve 9 çerçevelerinin hangi satırda beklediğinde saklıydı.

Her düğüm yığına tam bir kez itilir ve bir kez çıkarılır; döngü n tur döner, zaman $O(n)$'dir. Yığının boyutu yükseklikle sınırlıdır: her düzeyden en çok bir sağ kardeş beklerken bir de üstteki düğüm bulunur. Bu ağaçta yığın en çok 3 öğe tutmuştur (2. turun sonunda). Ek bellek $O(h)$'dir.

Sık yapılan hata

Çocukları önce sol, sonra sağ sırasıyla itmek. Bu durumda sağ çocuk üstte kalır ve önce işlenir; çıktı "kök, sağ, sol" sırasına döner. Şekil 9.24'teki ağaçta yanlış sırayla itme `1 3 7 6 2 5 4` yazdırır. Doğru kök önde sırası için sağ çocuk önce itilmeli, sol çocuk üstte kalmalıdır.

9.5.5 Yığınla kök ortada

Kök ortada gezinmede bir düğüm, yığına girdiği anda yazdırılamaz; önce sol alt ağacının bitmesi gerekir. Bu yüzden yığın burada "sonra yazdırılacak" düğümleri tutar. Algoritma iki hareketten oluşur. `gecici` bir düğümü gösterdiği sürece o düğüm yığına itilir ve sola inilir. `gecici` `null`'a ulaştığında sola gidecek yer kalmamıştır; yığının üstündeki düğümün sol alt ağacı bitmiştir. O düğüm çıkarılır, yazdırılır ve sağ alt ağacına geçilir.

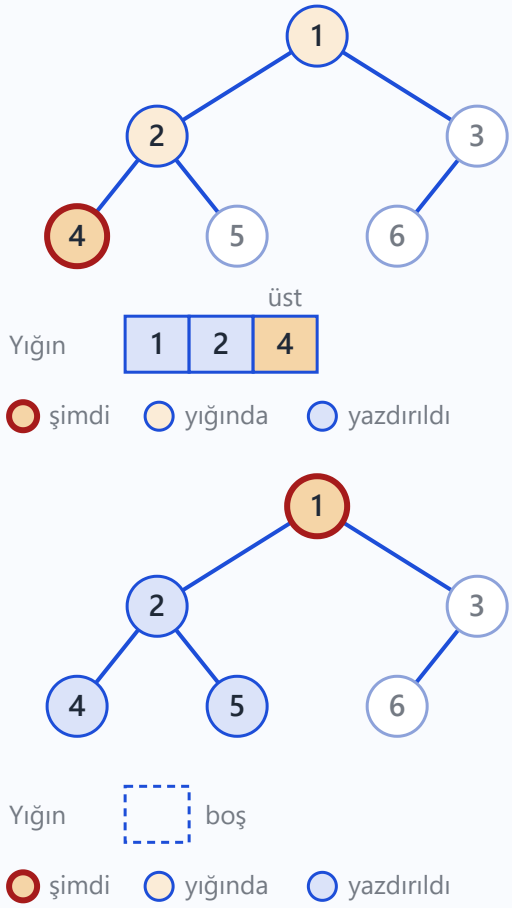
```
public void kokOrtadaYigin(AgacDugumu kok) {
  if (kok == null) return;
  Deque<AgacDugumu> yigin = new ArrayDeque<>();
  AgacDugumu gecici = kok;
  while (!yigin.isEmpty() || gecici != null) {
    if (gecici != null) {
      yigin.push(gecici);
      gecici = gecici.sol;
    } else {
      gecici = yigin.pop();
      System.out.print(gecici.veri + " ");
      gecici = gecici.sag;
    }
  }
}
```

Döngü koşulu iki parçalıdır: `!yigin.isEmpty() || gecici != null`. Yalnız yığının boşluğuna bakmak yetmez. Kök yazdırıldıktan hemen sonra yığın boş olabilir; ama `gecici` kökün sağ çocuğunu gösterir ve sağ alt ağaç henüz gezilmemiştir. İzleme bu anı içerir. Örnek olarak 1'in çocukları 2 ve 3, 2'nin çocukları 4 ve 5, 3'ün yalnız sol çocuğu 6 olan ağacı kullanalım.

Adım	Hareket	Yığın	`gecici`	Çıktı
0	başlangıç	[]	1	—
1	push 1, sola in	[1]	2	—
2	push 2, sola in	[1, 2]	4	—
3	push 4, sola in	[1, 2, 4]	null	—
4	pop 4, yaz, sağa geç	[1, 2]	null	4
5	pop 2, yaz, sağa geç	[1]	5	4 2
6	push 5, sola in	[1, 5]	null	4 2
7	pop 5, yaz, sağa geç	[1]	null	4 2 5
8	pop 1, yaz, sağa geç	[]	3	4 2 5 1
9	push 3, sola in	[3]	6	4 2 5 1
10	push 6, sola in	[3, 6]	null	4 2 5 1
11	pop 6, yaz, sağa geç	[3]	null	4 2 5 1 6

Adım	Hareket	Yığın	`gecici`	Çıktı
12	pop 3, yaz, sağa geç	[]	null	4 2 5 1 6 3

Tablo 9.17: `kokOrtadaYigin` izlemesi (ağaç: 1(2(4, 5), 3(6, —))). Yığın alttan üste.



Şekil 9.34: Yığınla kök ortada. Solda 3. adım: 1, 2, 4 yığındadır ve `gecici = null`; sola gidecek yer kalmamıştır. Sağda 8. adım: 1 yazdırılmıştır, yığın boştur ama `gecici` 3'ü gösterir; döngü sürer.

8. adım, döngü koşulunun neden iki parçalı olduğunu gösterir. Bu adımın sonunda yığın boştur; koşul yalnız `!yigin.isEmpty()` olsaydı döngü burada biter ve 3 ile 6 hiç yazdırılmazdı. `gecici != null` parçası, gezilmeyi bekleyen bir sağ alt ağaç olduğunu yakalar. Döngü ancak yığın boş *ve* `gecici` `null` olduğunda, 12. adımdan sonra biter. Çıktı `4 2 5 1 6 3`'tür.

Her düğüm bir kez itilir ve bir kez çıkarılır. Ayrıca her `null` çocuğa bir kez ulaşılır. Toplam iş $O(n)$'dir. Yığın her an kökten `gecici`'ye inen yol üzerindeki, sol alt ağacı henüz bitmemiş düğümleri tutar; boyutu en çok $h + 1$'dir. Ek bellek $O(h)$'dir.

9.5.6 Yığınla kök sonda

Kök sonda en zor yığınlı gezinmedir. Bir düğüm ancak *iki* alt ağacı da bittiğinde yazdırılabilir. Yığının üstünde bir düğüm görüldüğünde sol alt ağacının bittiği bilinir, ama sağ alt ağacın bitip bitmediği bilinmez. Kod bu yüzden yığındaki düğümü hemen çıkarmaz; önce `peek` ile bakar.

```
public void kokSondaYigin(AgacDugumu kok) {
  AgacDugumu simdiki = kok;
  Deque<AgacDugumu> yigin = new ArrayDeque<>();
  while (simdiki != null || !yigin.isEmpty()) {
    if (simdiki != null) {
      yigin.push(simdiki);
      simdiki = simdiki.sol;
    } else {
      AgacDugumu gecici = yigin.peek().sag;
      if (gecici == null) {
        gecici = yigin.pop();
        System.out.print(gecici.veri + " ");
        while (!yigin.isEmpty()
           && gecici == yigin.peek().sag) {
          gecici = yigin.pop();
          System.out.print(gecici.veri + " ");
        }
      } else {
        simdiki = gecici;
      }
    }
  }
}
```

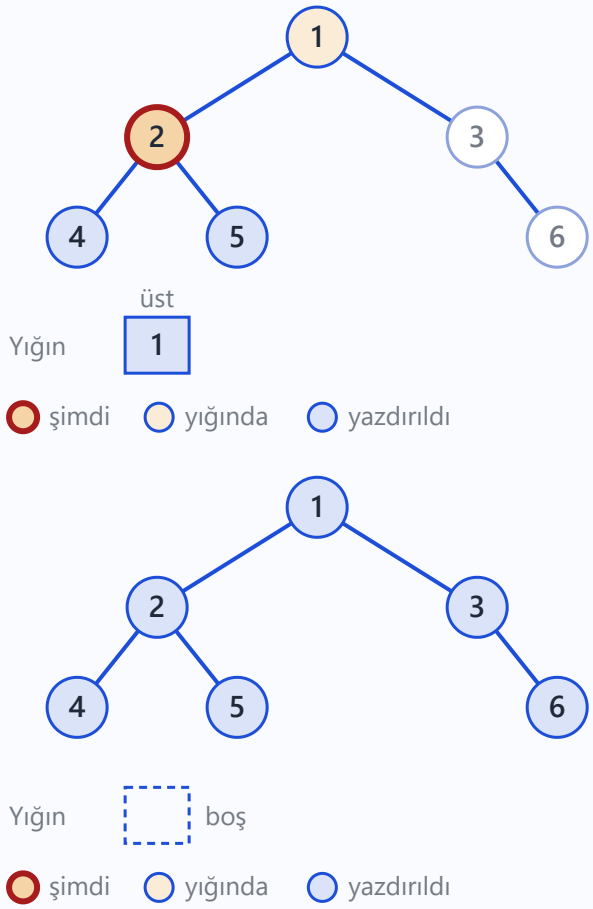
Kodun mantığı üç parçadır.

- **Sola inme.** `simdiki` `null` değilse düğüm yığına itilir ve sola inilir. Bu, kök ortadaki sola inmenin aynısıdır.
- **Sağ alt ağaca geçme.** Sola gidecek yer kalmayınca üstteki düğümün sağ çocuğuna bakılır (`yigin.peek().sag`). Sağ çocuk varsa düğüm henüz yazdırılamaz; `simdiki` sağ çocuğa atanır ve o alt ağaca girilir. Düğüm yığında kalır.
- **Yazdırma ve yukarı çıkma.** Sağ çocuk yoksa üstteki düğümün iki alt ağacı da bitmiştir; düğüm çıkarılır ve yazdırılır. Ardından iç `while` döngüsü yukarı bakar. Az önce yazdırılan düğüm, yığındaki yeni üst düğümün sağ çocuğuysa, o ebeveynin sağ alt ağacı da bitmiştir; o da çıkarılıp yazdırılır. Bu, ebeveyn zinciri boyunca sağ çocuk olmayı sürdüren düğümlerde tekrarlanır.

İç döngüdeki `gecici == yigin.peek().sag` karşılaştırması referans eşitliğidir: aynı nesne mi, diye sorar. Bu karşılaştırma, "bu ebeveynin sağ alt ağacından şimdi mi dönüyoruz?" sorusunun yanıtıdır. Yanıt evetse ebeveyn yazdırılır. Yanıt hayırsa, yani az önce yazdırılan düğüm sol çocuksa, ebeveynin sağ alt ağacına henüz girilmemiştir ve iç döngü durur. İzleme 1'in çocukları 2 ve 3, 2'nin çocukları 4 ve 5, 3'ün yalnız sağ çocuğu 6 olan ağaç üzerinde yapılır.

Adım	Olay	Yığın	Çıktı
1–3	1, 2, 4 itilir; `simdiki = 4.sol = null`	[1, 2, 4]	—
4	`peek() = 4`, sağ çocuk yok: 4 çıkarılıp yazdırılır	[1, 2]	4
5	iç döngü: 4, 2'nin sağ çocuğu değil (2.sag = 5); durur	[1, 2]	4
6	`peek() = 2`, sağ çocuk 5: `simdiki = 5`	[1, 2]	4
7	5 itilir; `simdiki = null`	[1, 2, 5]	4
8	`peek() = 5`, sağ çocuk yok: 5 yazdırılır	[1, 2]	4 5
9	iç döngü: 5, 2'nin sağ çocuğu: 2 yazdırılır	[1]	4 5 2
10	iç döngü: 2, 1'in sağ çocuğu değil (1.sag = 3); durur	[1]	4 5 2
11	`peek() = 1`, sağ çocuk 3: `simdiki = 3`	[1]	4 5 2
12	3 itilir; `simdiki = 3.sol = null`	[1, 3]	4 5 2
13	`peek() = 3`, sağ çocuk 6: `simdiki = 6`	[1, 3]	4 5 2
14	6 itilir; `simdiki = null`	[1, 3, 6]	4 5 2
15	`peek() = 6`, sağ çocuk yok: 6 yazdırılır	[1, 3]	4 5 2 6
16	iç döngü: 6, 3'ün sağ çocuğu: 3 yazdırılır	[1]	4 5 2 6 3
17	iç döngü: 3, 1'in sağ çocuğu: 1 yazdırılır	[]	4 5 2 6 3 1
18–19	yığın boş: iç döngü durur; `simdiki` `null`, dış döngü biter	[]	4 5 2 6 3 1

Tablo 9.18: `kokSondaYigin` izlemesi (ağaç: 1(2(4, 5), 3(—, 6))). Yığın alttan üste.



Şekil 9.35: Yığınla kök sonda. Solda 9. adım: 5 yazdırıldıktan sonra iç döngü, 5'in 2'nin sağ çocuğu olduğunu görmüş ve 2'yi de yazdırmıştır; yığında yalnız 1 kalmıştır. Sağda gezinme bitmiştir; çıktı `4 5 2 6 3 1`.

İzlemenin son kısmı iç döngünün gücünü gösterir. 15. adımda 6 yazdırılınca iç döngü art arda iki düğümü daha yazdırır. 6, 3'ün sağ çocuğudur; 3'ün iki alt ağacı da bitmiştir. 3 de 1'in sağ çocuğudur; 1'in de iki alt ağacı bitmiştir. Tek bir yaprak, ebeveyn zinciri boyunca bir dizi düğümün "tamamlanmasını" tetiklemiştir. 3'ün sol çocuğunun olmaması da kodu bozmaz. 12. adımda `simdiki = 3.sol` doğrudan `null` olur ve 13. adımda sağ alt ağaca geçilir.

Maliyet yine *O(n)*'dir. Her düğüm bir kez itilir, bir kez çıkarılır; her `peek` ya bir çıkarma ya da bir sağ alt ağaca geçişle sonuçlanır. Yığın kökten o anki düğüme inen yolu tuttuğu için boyutu en çok *h + 1*'dir. Aynı sonuca daha kısa bir yoldan da varılır. Yığınlı kök önde kodunda çocuklar ters sırayla (önce sol, sonra sağ) itilirse "kök, sağ, sol" sırası çıkar. Bu çıktı ikinci bir yığınla ters çevrilir. Ters çevrilmiş "kök, sağ, sol" tam olarak "sol, sağ, kök"tür. Bu yöntem kolaydır, ama *O(n)* ek bellek ister.

9.5.7 Kuyrukla düzey sırası

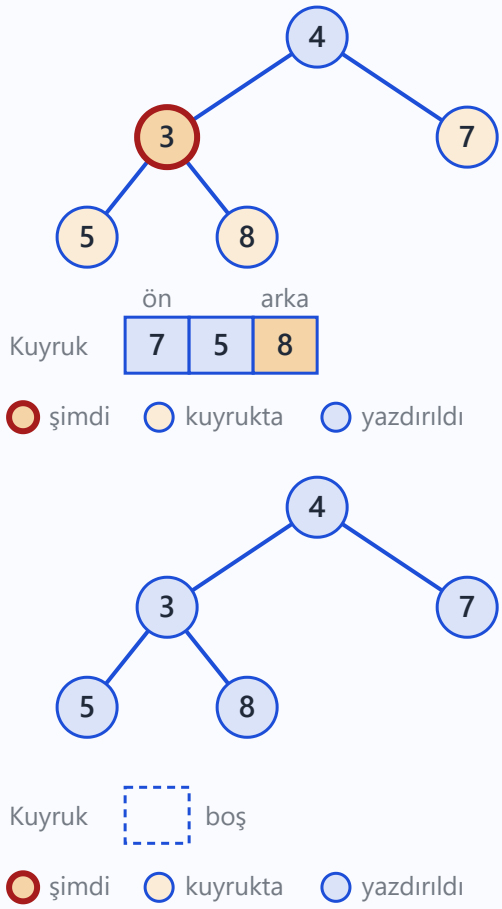
```
public void seviyeSirasi(AgacDugumu kok) {
  if (kok == null) return;
  Queue<AgacDugumu> kuyruk = new LinkedList<>();
  kuyruk.add(kok);
  while (!kuyruk.isEmpty()) {
    AgacDugumu gecici = kuyruk.poll();
    System.out.print(gecici.veri + " ");
    if (gecici.sol != null)
      kuyruk.add(gecici.sol);
    if (gecici.sag != null)
      kuyruk.add(gecici.sag);
  }
}
```

Düzey sırası kodu, yığınla kök önde koduna çok benzer. Üç fark vardır. Yığının yerinde bir `Queue` (burada `LinkedList`) bulunur. `push` ve `pop` yerine `add` (arkaya ekle) ve `poll` (önden çıkar) kullanılır. Çocuklar doğal sırayla eklenir: önce sol, sonra sağ. Kuyruk FIFO olduğu için önce eklenen sol çocuk önce çıkar; ters sırayla eklemeye gerek yoktur.

İzleme 4 kökündeki ağaç üzerinde yapılır: 4'ün çocukları 3 ve 7; 3'ün çocukları 5 ve 8; 7 yapraktır.

Tur	poll	Eklenenler	Tur sonunda kuyruk	Çıktı
başlangıç	—	4	[4]	—
1	4	3, 7	[3, 7]	4
2	3	5, 8	[7, 5, 8]	4 3
3	7	—	[5, 8]	4 3 7
4	5	—	[8]	4 3 7 5
5	8	—	[]	4 3 7 5 8

Tablo 9.19: `seviyeSirasi` izlemesi (ağaç: 4(3(5, 8), 7)). Kuyruk önden arkaya yazılmıştır.



Şekil 9.36: Kuyrukla düzey sırası. Solda 2. tur bitmiştir: 3'ün çocukları 5 ve 8, 7'nin arkasına eklenmiştir. Sağda kuyruk boşalmıştır; çıktı `4 3 7 5 8`.

2. tur düzey sırasının özünü gösterir. 3 çıkarılırken çocukları 5 ve 8 kuyruğun arkasına, 7'nin arkasına eklenir. 7, 3 ile aynı düzeydedir ve 3'ün çocuklarından önce işlenir. Kök önde olsaydı 3'ten sonra hemen 5 ve 8 gelirdi; çıktı `4 3 5 8 7` olurdu. Düzey sırası ise `4 3 7 5 8` verir.

Her düğüm kuyruğa bir kez girer ve bir kez çıkar; zaman *O(n)*'dir. Ek bellek ise yığınlı gezinmelerden farklı bir büyüklüğe bağlıdır. Kuyrukta aynı anda bulunan düğümler en çok iki ardışık düzeye yayılır. Bu yüzden kuyruğun boyutu ağacın **genişliğiyle**, yani en kalabalık düzeydeki düğüm sayısı *w* ile sınırlanır; ek bellek *O(w)*'dir. Bu izlemede kuyruk en çok 3 düğüm tutmuştur. Mükemmel ağaçta son düzeyde yaklaşık *n/2* düğüm bulunur; düzey sırası bu ağaçlarda *O(n)* bellek harcar. Yığınlı kök ortada ise aynı ağaçta yalnız *O(log n)* bellek kullanır.

Sık yapılan hata

`add()` yerine `push()` kullanmanın farkını gözden kaçırmak. `LinkedList` hem kuyruk hem yığın arayüzünü uygular: `pop()` baştan çıkarır, `push()` ise başa ekler. Kodda `add` yerine yanlışlıkla `push` yazılırsa çocuklar başa eklenir ve kuyruk yığına dönüşür; çıktı düzey sırası

olmaktan çıkar. Değişkeni `Queue<AgacDugumu>` tipiyle bildirmek, yığın metotlarının kullanılmasını derleme anında engeller.

9.6 Gezinmelerin uygulamaları

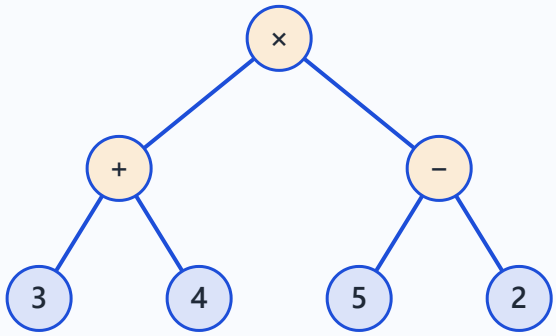
Gezinmeler yalnız düğümleri yazdırmak için değildir. Ağaç üzerinde çalışan hemen her algoritma bir gezinmedir: düğümleri belirli bir sırayla ziyaret eder ve her ziyarette bir iş yapar. Bu kesim, gezinme kalıbının nasıl gerçek problemlere uygulandığını örneklerle gösterir.

9.6.1 İfade ağacı

Bir aritmetik ifade doğal olarak bir ağaçtır. *(3 + 4) × (5 − 2)* ifadesinde en son yapılacak işlem çarpmadır; çarpmanın iki işleneni de birer alt ifadedir. Bu yapı bir ikili ağaçla doğrudan temsil edilir.

Tanım 9.6

Bir **ifade ağacı** (*expression tree*), iç düğümleri işleç, yaprakları işlenen (sayı ya da değişken) olan bir ikili ağaçtır. İkili işleçler kullanıldığında her iç düğümün tam iki çocuğu vardır; ağaç doludur. Bir iç düğümün değeri, işlecinin sol ve sağ alt ağacın değerlerine uygulanmasıyla bulunur.



Şekil 9.37: *(3 + 4) × (5 − 2)* ifadesinin ağacı. Kökte çarpma; solda 3 artı 4, sağda 5 eksi 2 alt ağaçları bulunur.

Ağaçta parantez yoktur; parantezlerin görevini ağacın yapısı üstlenir. Toplama ve çıkarma, çarpmadan daha derinde durduğu için önce hesaplanır. İşleç önceliği ve parantez bilgisi ağacın şekline gömülmüştür. Aynı ağaç, üç derinlik öncelikli gezinmeyle üç farklı gösterim verir.

Gezinme	Çıktı	Gösterim
Kök önde	`× + 3 4 − 5 2`	önek (*prefix*)
Kök ortada	`3 + 4 × 5 − 2` ; parantezle `(3 + 4) × (5 − 2)`	ara ek (*infix*)

Gezinme	Çıktı	Gösterim
Kök sonda	`3 4 + 5 2 - ×` → *7 × 3 = 21*	sonek (*postfix*)

Tablo 9.20: Şekil 9.37'deki ağacın gezinmeleri.

Önek ve sonek gösterimler parantez gerektirmez; işleçlerin ve işlenenlerin sırası ifadeyi tek anlamlı belirler. Kök ortada çıktısı ise parantezsiz yazıldığında anlam değiştirir. `3 + 4 × 5 − 2` olağan öncelik kurallarıyla *3 + 20 − 2 = 21*'e eşit görünse de bu bir rastlantıdır; okunan ifade ağaçtaki ifade değildir. Doğru ara ek gösterimi elde etmek için her iç düğümün alt ağacı parantez içine alınmalıdır. Bu da düğüme girerken "(", çıkarken ")" yazan bir Euler turudur.

> **Sık yapılan hata**
>
> Kök ortada çıktısını doğrudan ara ek ifade saymak. Parantezler kaybolduğu için ifade başka bir ağaca karşılık gelebilir. *((6 + 2) × 3) − (8 / 4)* ifadesinin ağacı kök ortada gezilirse `6 + 2 × 3 − 8 / 4` çıkar; bu, olağan önceliklerle *6 + 6 − 2 = 10* verir, oysa ağacın değeri 22'dir.

9.6.2 Kod izleme: hesapla()

İfade ağacının değeri kök sonda mantığıyla hesaplanır: bir işleç, iki işleneninin değeri bilinmeden uygulanamaz. Aşağıdaki kod, yaprakta bir sayı (`sayi`), iç düğümde bir işleç karakteri (`islec`) tutan bir `IfadeDugumu` sınıfı varsayar.

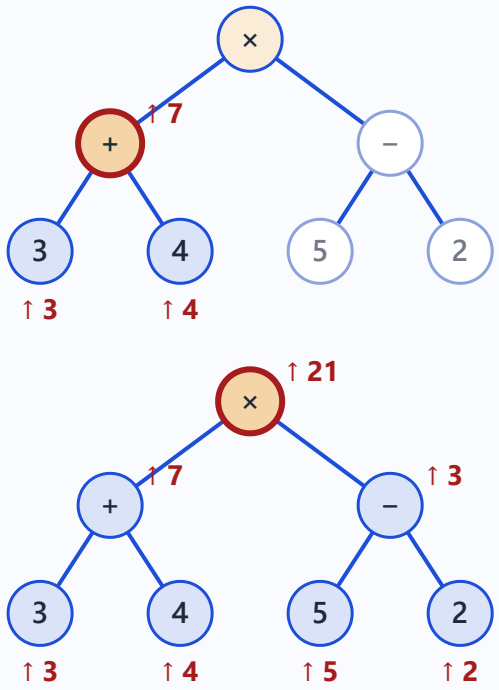
```
int hesapla(IfadeDugumu d) {
  if (d.sol == null && d.sag == null)
    return d.sayi;              // yaprak: işlenen
  int a = hesapla(d.sol);
  int b = hesapla(d.sag);
  switch (d.islec) {            // iç düğüm: işleç
    case '+': return a + b;
    case '-': return a - b;
    case '*': return a * b;
    default:  return a / b;
  }
}
```

Metot önce düğümün yaprak olup olmadığına bakar. İki çocuğu da yoksa düğüm bir işlenendir ve sayısı döndürülür; taban durum budur. Değilse sol alt ağacın değeri hesaplanıp `a`'ya (satır 4), sağ alt ağacınki `b`'ye (satır 5) atanır. En son `switch` ile işleç uygulanır. Sol ve sağın sırası çıkarma ile bölmede önemlidir: `a - b`'de `a` sol işlenendir. İfade ağacı dolu olduğu için bir iç düğümün iki çocuğu da vardır; tek çocuklu düğüm durumu ele alınmaz.

Adım	Çağrı yığını	Olay
1	hesapla(×)	iç düğüm; önce sol işlenen: `hesapla(+)`

Adım	Çağrı yığını	Olay
2	× · s.4 | hesapla(+)	iç düğüm; önce sol işlenen: `hesapla(3)`
3	× · s.4 | + · s.4 | hesapla(3)	yaprak: 3 döner
4–5	× · s.4 | + · s.5 | hesapla(4)	`a = 3`; sağ işlenen yaprak: 4 döner
6	× · s.4 | hesapla(+)	`b = 4`; *3 + 4 = 7* döner
7–8	× · s.5 | hesapla(−)	`a = 7`; sağ işlenen `hesapla(-)`
9–11	× · s.5 | − · s.4 / s.5 | hesapla(5), hesapla(2)	yapraklar 5 ve 2 döner
12	× · s.5 | hesapla(−)	*5 − 2 = 3* döner
13	hesapla(×)	`b = 3`; *7 × 3 = 21* döner

Tablo 9.21: `hesapla(×)` izlemesi. Çağrı yığını alttan üste; "s.4" sol, "s.5" sağ işlenen çağrısıdır.



Şekil 9.38: `hesapla` izlemesi. Solda 6. adım: 3 ve 4 yukarı taşınmış, + düğümü 7 döndürmüştür. Sağda son adım: − düğümü 3, kök 21 döndürmüştür. Oklu sayılar dönüş değerleridir.

Değerler ağaçta aşağıdan yukarı akar: yapraklar sayılarını, iç düğümler hesapladıkları değeri ebeveynlerine verir. İşleçlerin uygulanma sırası `+`, `−`, `×`'tir; bu tam olarak kök sonda sırasıdır ve sonek gösterimdeki `3 4 + 5 2 − ×` dizisinin okunuşuyla örtüşür. Sonek ifadeyi bir yığınla değerlendiren algoritma da aynı işi yapar; burada yığın, özyinelemenin çağrı yığınıdır. Her düğüm bir kez işlenir; zaman *O(n)*, ek bellek *O(h)*'dir.

9.6.3 En büyük değeri bulma

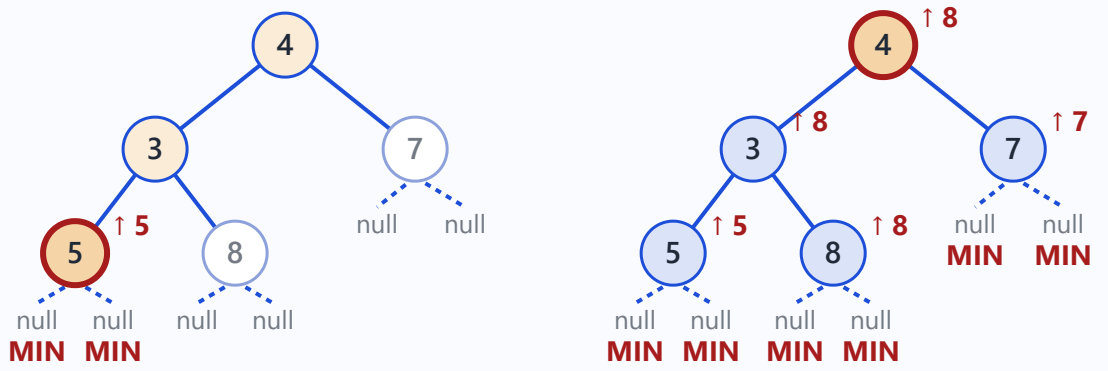
Sıralı olmayan bir ikili ağaçta en büyük değeri bulmak için her düğüme bakmak gerekir. Özyinelemeli çözüm şöyle düşünür: ağacın en büyüğü, kökün değeri, sol alt ağacın en büyüğü ve sağ alt ağacın en büyüğü arasındaki en büyüktür.

```
public int enBuyukBul(AgacDugumu kok) {
  if (kok == null) {
    return Integer.MIN_VALUE;
  }
  int sonuc = kok.veri;
  int sol = enBuyukBul(kok.sol);
  int sag = enBuyukBul(kok.sag);
  if (sol > sonuc)
    sonuc = sol;
  if (sag > sonuc)
    sonuc = sag;
  return sonuc;
}
```

Boş ağaç için hangi değerin döndürüleceği dikkat ister. Boş ağacın “en büyük değeri” yoktur; ama döndürülen değer karşılaştırmayı bozmamalıdır. `Integer.MIN_VALUE` (-2^{31}) hiçbir `int` değerden büyük olamaz; `sol > sonuc` karşılaştırmasında hiçbir zaman kazanmaz. Bu yüzden boş alt ağaç sonucu etkilemez. Metodun gövdesi kök sonda kalıbındadır: iki alt ağacın sonucu alınır, sonra kökle birleştirilir.

Çağrı	`sonuc` başta	`sol`	`sag`	Döner
`enBuyukBul(5)`	5	MIN	MIN	5
`enBuyukBul(8)`	8	MIN	MIN	8
`enBuyukBul(3)`	3	5 (> 3: `sonuc = 5`)	8 (> 5: `sonuc = 8`)	8
`enBuyukBul(7)`	7	MIN	MIN	7
`enBuyukBul(4)`	4	8 (> 4: `sonuc = 8`)	7 (≤ 8)	8

Tablo 9.22: `enBuyukBul` izlemesi (ağaç: 4(3(5, 8), 7)). MIN, `Integer.MIN_VALUE` 'dur.



Şekil 9.39: `enBuyukBul` izlemesi. Solda 5'in iki boş alt ağacı MIN döndürmüş, 5 kendi değerini döndürmüştür. Sağda bütün değerler yukarı taşınmıştır: 3'ün alt ağacı 8, 7 kendisi, kök 8 döndürür.

Tablo, çağrıların *bitiş* sırasına göre yazılmıştır: 5, 8, 3, 7, 4. Bu, kök sonda sırasıdır. Her düğüm bir kez ziyaret edilir ve her `null` çocuk için bir çağrı yapılır; beş düğümlü ağaçta toplam 11 çağrı vardır. Zaman *O(n)*'dir. Ağaç bir ikili arama ağacı olsaydı en büyük değer çok daha hızlı bulunurdu: kökten sağa sağa inmek yeterdi ve maliyet *O(h)* olurdu. Sıralı olmayan bir ağaçta ise bütün düğümlere bakmaktan kaçınılamaz.

Sık yapılan hata

Boş ağaç için 0 döndürmek. Bütün değerleri negatif olan bir ağaçta, örneğin −5 ve −3 değerli iki düğümde, 0 hiçbir düğümde bulunmadığı hâlde sonuç olarak döner. Taban durumun değeri, birleştirme işleminin *etkisiz öğesi* olmalıdır: en büyükte `Integer.MIN_VALUE`, toplamda 0, çarpımda 1.

9.6.4 Özyinelemeli ağaç fonksiyonları

`hesapla` ve `enBuyukBul` aynı kalıbı izler. Boş ağaç için bir taban değer döndürülür; boş olmayan ağaçta iki alt ağacın sonucu özyinelemeyle alınır ve kökle birleştirilir. Bu kalıp, ağacın birçok temel özelliğini birkaç satırla hesaplamaya yeter.

```
int boyut(AgacDugumu k) {            // düğüm sayısı
    if (k == null) return 0;
    return 1 + boyut(k.sol) + boyut(k.sag);
}

int yukseklik(AgacDugumu k) {        // kenar sayısı
    if (k == null) return -1;        // boş ağaç: -1
    return 1 + Math.max(yukseklik(k.sol),
                        yukseklik(k.sag));
}

int yaprakSayisi(AgacDugumu k) {
    if (k == null) return 0;
    if (k.sol == null && k.sag == null) return 1;
    return yaprakSayisi(k.sol) + yaprakSayisi(k.sag);
}
```

- `boyut(k)` düğüm sayısını verir. Boş ağaçta 0 düğüm vardır. Boş olmayan ağaçta kök için 1, sol alt ağaçtakiler için `boyut(k.sol)`, sağdakiler için `boyut(k.sag)` toplanır.
- `yukseklik(k)` yüksekliği kenar sayısıyla verir. Taban durumda **−1** döndürülür. Bu seçim ilk bakışta tuhaf gelir, ama tanımı tutarlı kılar: yaprağın iki çocuğu da boştur ve yüksekliği *1 + max(−1, −1) = 0* çıkar. Taban durumda 0 döndürülseydi yaprağın yüksekliği 1 olurdu; bu da yüksekliği düğüm sayısıyla ölçmek demektir.
- `yaprakSayisi(k)` iki taban durum kullanır. Boş ağaçta 0 yaprak vardır. İki çocuğu da `null` olan düğüm bir yapraktır ve 1 döndürür. Diğer düğümlerde yaprak sayısı, iki alt ağacın yaprak sayılarının toplamıdır; düğümün kendisi yaprak olmadığı için eklenmez.

9 kökündeki dört düğümlü ağaçta (9'un çocukları 2 ve 3; 2'nin sol çocuğu 4) bu metotlar `boyut = 4`, `yukseklik = 2` ve `yaprakSayisi = 2` verir; yapraklar 4 ve 3'tür. Üç metot da her düğümü bir kez işler: *O(n)* zaman, çağrı yığını için *O(h)* alan.

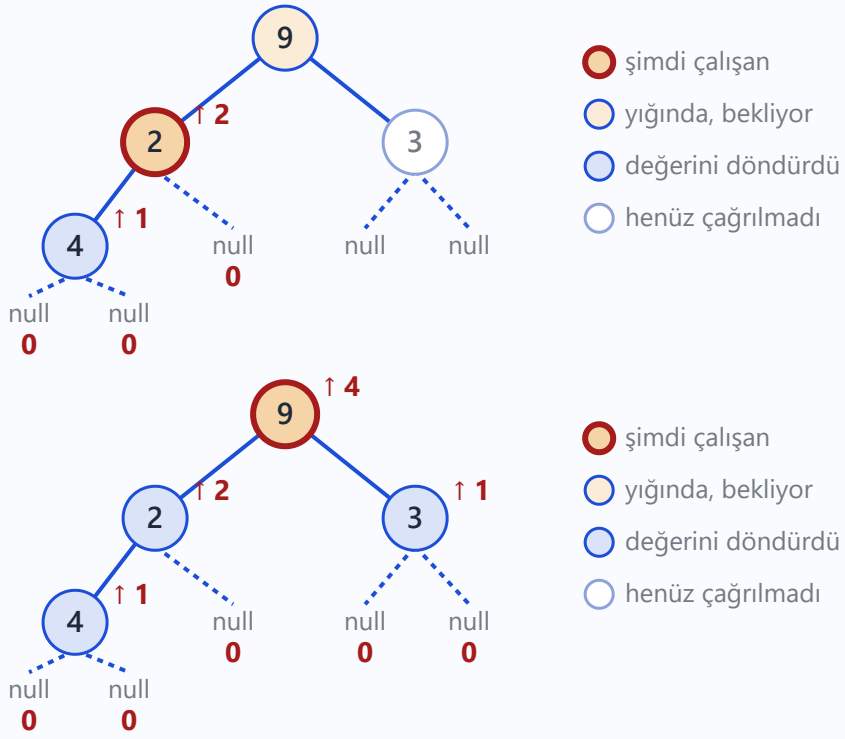
9.6.5 Kod izleme: boyut()

`boyut`'un izlemesi, değerlerin alt ağaçlardan yukarı nasıl taşındığını gösterir. Bütün özyinelemeli çağrılar tek satırdan, satır 3'ten yapılır; Java ifadeyi soldan sağa hesapladığı için önce `boyut(k.sol)`, sonra `boyut(k.sag)` çağrılır.

Adım	Çağrı	Olay	Döner
1–3	`boyut(9)`, `boyut(2)`, `boyut(4)`	her biri önce sol alt ağacı çağırır	—
4	`boyut(4.sol)`	boş ağaç	0
5–6	`boyut(4.sag)`	sol 0 geldi; sağ da boş	0
7	`boyut(4)`	*1 + 0 + 0*	1
8–9	`boyut(2.sag)`	sol 1 geldi; sağ boş	0

Adım	Çağrı	Olay	Döner
10	`boyut(2)`	*1 + 1 + 0*	2
11–15	`boyut(3)`	iki çocuğu boş; *1 + 0 + 0*	1
16–17	`boyut(9)`	*1 + 2 + 1*	4

Tablo 9.23: `boyut(9)` izlemesi (ağaç: 9(2(4, —), 3)).



Şekil 9.40: `boyut` izlemesi. Üstte 10. adım: 4'ün alt ağacı 1, 2'nin alt ağacı 2 döndürmüştür. Altta son adım: 3 de 1 döndürmüş, kök *1 + 2 + 1 = 4* vermiştir. `null` çocukların altındaki 0'lar, boş ağaçların dönüş değeridir.

İzlemede dokuz çağrı yapılır: dört düğüm ve beş `null` çocuk için. Bu, *2n + 1 = 9*'dur. Her düğümün döndürdüğü sayı, o düğümün alt ağacındaki düğüm sayısıdır: 4 için 1, 2 için 2, 3 için 1, 9 için 4. Böylece tek bir kök sonda gezinmesi bütün alt ağaçların boyutunu da hesaplamış olur. Bu bilgi düğümlerde saklanırsa, örneğin "ağaçtaki *k*. en küçük anahtar" gibi sorular hızla yanıtlanabilir.

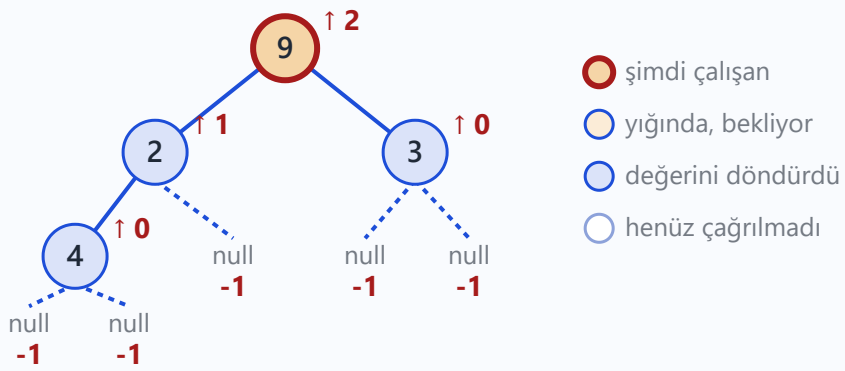
9.6.6 Kod izleme: yukseklik()

`yukseklik`'te sol çağrı satır 3'ten, sağ çağrı satır 4'ten yapılır: `Math.max`'ın iki argümanı iki ayrı satıra yazılmıştır. İzleme aynı ağaç üzerindedir.

Adım	Çağrı	Hesap	Döner
1–3	`yukseklik(9)`, `(2)`, `(4)`	her biri önce sol alt ağacı çağırır	—
4–6	`yukseklik(4.sol)`, `yukseklik(4.sag)`	boş ağaçlar	−1, −1
7	`yukseklik(4)`	*1 + max(−1, −1)*	0

Adım	Çağrı	Hesap	Döner
8–9	`yukseklik(2.sag)`	boş ağaç	−1
10	`yukseklik(2)`	*1 + max(0, −1)*	1
11–16	`yukseklik(3)`	*1 + max(−1, −1)*	0
17	`yukseklik(9)`	*1 + max(1, 0)*	2

Tablo 9.24: `yukseklik(9)` izlemesi (ağaç: 9(2(4, —), 3)).



Şekil 9.41: `yukseklik` izlemesinin son adımı. Boş ağaçlar −1, yapraklar 4 ve 3 ise 0 döndürmüştür; 2'nin yüksekliği 1, kökün yüksekliği *1 + max(1, 0) = 2*'dir.

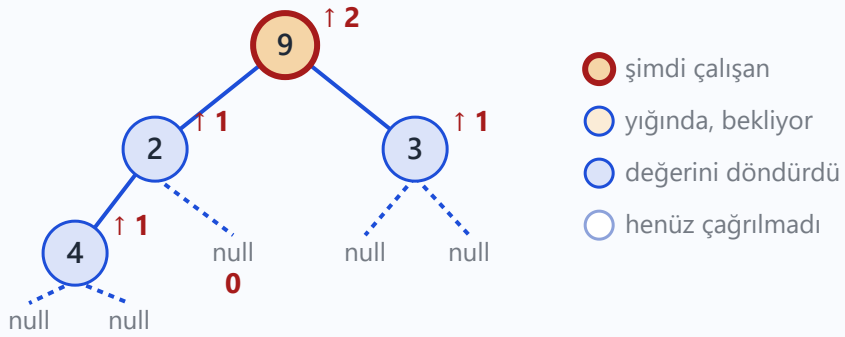
Sonuç 2'dir: kökten en derin yaprak 4'e giden 9–2–4 yolu iki kenarlıdır. Çağrı yapısı `boyut` ile aynıdır, birleştirme işlemi farklıdır: toplama yerine en büyük alınır. Kalıbın gücü buradadır. Taban değeri ve birleştirme işlemi değiştirilerek aynı iskeletle düğümlerin toplamı, en küçük değer, belirli bir değerin kaç kez geçtiği gibi birçok özellik hesaplanabilir.

9.6.7 Kod izleme: yaprakSayisi()

`yaprakSayisi` bir yaprakta alt çağrı yapmadan döner. Bu yüzden izlemesi daha kısadır: yaprakların `null` çocukları hiç çağrılmaz.

Adım	Çağrı	Olay	Döner
1	`yaprakSayisi(9)`	yaprak değil; önce sol	—
2	`yaprakSayisi(2)`	yaprak değil; önce sol	—
3	`yaprakSayisi(4)`	iki çocuğu da `null` : yaprak	1
4	`yaprakSayisi(2.sag)`	boş ağaç	0
5	`yaprakSayisi(2)`	*1 + 0*	1
6	`yaprakSayisi(3)`	yaprak	1
7	`yaprakSayisi(9)`	*1 + 1*	2

Tablo 9.25: `yaprakSayisi(9)` izlemesi (ağaç: 9(2(4, —), 3)).



Şekil 9.42: `yaprakSayisi` izlemesinin son adımı. 4 ve 3 yaprak olarak 1 döndürmüş, 2'nin boş sağ alt ağacı 0 vermiştir; kökte *1 + 1 = 2* bulunur.

Bu izlemede yalnız beş çağrı yapılır: dört düğüm ve 2'nin boş sağ çocuğu. `boyut`'taki dokuz çağrıdan azdır, çünkü yaprak denetimi yaprakların `null` çocuklarına inmeyi önler. İkinci taban durumun bir başka işlevi daha vardır. O satır olmasaydı metot boş ağaçta 0 döndürür ve hiçbir zaman 1 üretmezdi; her ağaç için sonuç 0 çıkardı. Yaprak, sayılacak şeyin kendisidir; bu yüzden ayrı bir taban durum ister.

Sık yapılan hata

`yaprakSayisi`'nde yalnız `k == null` taban durumunu yazıp yaprak denetimini unutmak ya da yaprak denetimini `null` denetiminden önce yazmak. Birincisinde sonuç her zaman 0'dır. İkincisinde `k.sol` okunurken `k` `null` olabilir ve `NullPointerException` fırlatılır. İki taban durum, kodda gösterilen sırayla yazılmalıdır.

9.6.8 Gezinmelerden ağacı yeniden kurma

Bir ağacın tek bir gezinme çıktısı, ağacı belirlemez. Örneğin kök önde çıktısı `1 2` olan iki ağaç vardır: 2, 1'in sol çocuğu da olabilir sağ çocuğu da. Ama iki uygun gezinme birlikte verilirse ve düğüm değerleri birbirinden farklıysa ağaç tek biçimde kurulabilir.

Teorem 9.6 (Gezinmelerden kurma)

Düğüm değerleri birbirinden farklı bir ikili ağaç, kök ortada çıktısı ile kök önde çıktısından (ya da kök ortada ile kök sonda çıktısından) tek biçimde kurulabilir. Kök önde ile kök sonda çıktıları ise, tek çocuklu düğüm bulunan ağaçlarda ağacı belirlemez.

Kurma yöntemi gezinmelerin tanımından çıkar. Kök önde çıktısının ilk öğesi köktür. Bu değer kök ortada çıktısında bulunur. Kök ortada çıktısında kökün solundaki öğeler sol alt ağacı, sağındakiler sağ alt ağacı oluşturur. Sol alt ağaçtaki öğe sayısı bilindiği için kök önde çıktısında da hangi öğelerin sol alt ağaca ait olduğu bulunur. Aynı işlem iki alt ağaca özyinelemeyle uygulanır. Kök sonda ile çalışırken tek fark, kökün çıktının *son* öğesi olmasıdır.

Kök önde ile kök sonda birlikte neden yetmez? İki gezinmede de kökün yeri bellidir, ama sol ve sağ alt ağaçları ayıran bir bilgi yoktur. Kök önde `A B` ve kök sonda `B A` çıktıları hem "B, A'nın sol çocuğu" hem de "B, A'nın sağ çocuğu" ağacına uyar. Ayrımı yalnız kök ortada çıktısı yapar: soldaki çocuk kökten önce, sağdaki sonra yazılır. Her düğümün 0 ya da 2 çocuğu olan dolu ağaçlarda ise belirsizlik kalkar; çünkü tek çocuklu düğüm yoktur.

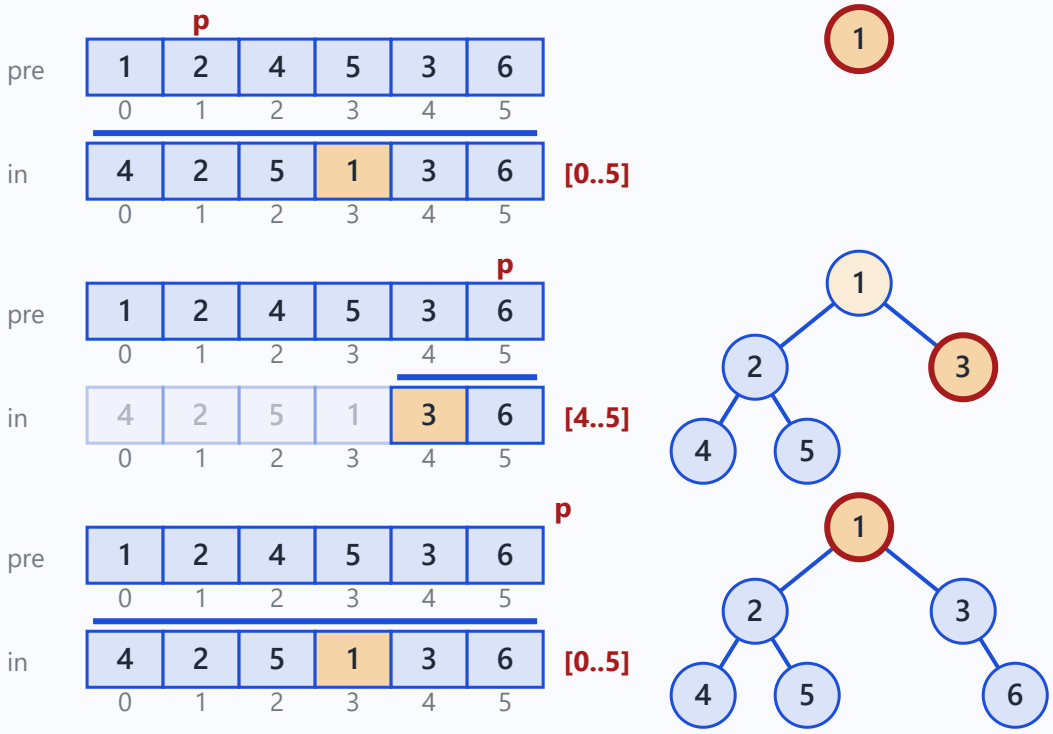
```
int p = 0; // pre[] içinde sıradaki kök
AgacDugumu kur(int[] pre, int[] in, int bas, int son) {
    if (bas > son) return null;       // boş aralık
    AgacDugumu k = new AgacDugumu(pre[p++]);
    int m = bas; while (in[m] != k.veri) m++;
    k.sol = kur(pre, in, bas, m - 1); // soldakiler
    k.sag = kur(pre, in, m + 1, son); // sağdakiler
    return k;
}
```

`kur(pre, in, bas, son)`, kök ortada dizisinin `in[bas..son]` aralığındaki öğelerden bir alt ağaç kurar ve kökünü döndürür. Aralık boşsa (`bas > son`) alt ağaç da boştur ve `null` döner. Değilse kök önde dizisinde sıradaki öğe, `pre[p]`, bu alt ağacın köküdür; düğüm oluşturulur ve `p` bir artırılır. `p` bir alan olduğu için bütün çağrılar arasında paylaşılır ve kök önde dizisinde ileri doğru tek bir kez ilerler. Sonra kökün değeri kök ortada aralığında aranır ve indisi `m`'ye yazılır. `in[bas..m-1]` sol, `in[m+1..son]` sağ alt ağaçtır. Önce sol alt ağaç kurulur. Bu sıra zorunludur, çünkü kök önde dizisinde sol alt ağacın öğeleri sağdakilerden önce gelir.

Örnek 9.6

`pre = [1, 2, 4, 5, 3, 6]` ve `in = [4, 2, 5, 1, 3, 6]` dizilerinden ağacı kuralım. İlk çağrı `kur(pre, in, 0, 5)` 'tir.

1. `pre[0] = 1`. Kök 1'dir. `in`'de 1'in indisi *m = 3*'tür. `in[0..2] = 4, 2, 5` sol alt ağaç, `in[4..5] = 3, 6` sağ alt ağaçtır.
2. `pre[1] = 2`. Sol alt ağacın kökü 2'dir; *m = 1*. Sol aralığı `[0..0]` (4), sağ aralığı `[2..2]` (5).
3. `pre[2] = 4`. 2'nin sol çocuğu 4'tür; *m = 0*. İki aralık da boştur; iki çağrı `null` döner. 4 yapraktır.
4. `pre[3] = 5`. 2'nin sağ çocuğu 5'tir; *m = 2*; o da yapraktır. 2'nin alt ağacı tamamlanır ve `1.sol = 2` yapılır.
5. `pre[4] = 3`. Sağ alt ağacın kökü 3'tür; *m = 4*. Sol aralık `[4..3]` boştur: 3'ün sol çocuğu yoktur. Sağ aralık `[5..5]` 'tir.
6. `pre[5] = 6`. 3'ün sağ çocuğu 6'dır; *m = 5*; yapraktır. `1.sag = 3` yapılır.
7. **Sonuç.** Ağaç 1(2(4, 5), 3(—, 6))'dır. Denetim: bu ağacın kök önde çıktısı `1 2 4 5 3 6`, kök ortada çıktısı `4 2 5 1 3 6` 'dır.



Şekil 9.43: `kur` izlemesinin üç anı. Üstte kök 1 oluşturulmuş ve `in` dizisi 1'in iki yanında bölünmüştür. Ortada sol alt ağaç (2, 4, 5) kurulmuş, sağ alt ağacın kökü 3 için `in[4..5]` aralığına geçilmiştir. Altta ağaç tamamlanmıştır.

Maliyeti belirleyen, `m`'yi bulan doğrusal aramadır. Her düğüm için kendi aralığında bir arama yapılır. Dengeli ağaçta aralıklar her düzeyde yarıya iner ve toplam $O(n \log n)$ olur. Çarpık ağaçta ise aralıklar birer birer küçülür ve toplam $O(n^2)$'ye çıkar. İyileştirme kolaydır: `in` dizisindeki her değerin indisi önceden bir eşlemeye (`HashMap`) yazılırsa `m` $O(1)$'de bulunur ve kurma $O(n)$'e iner.

Sık yapılan hata

Kök sonda ile kurarken sol alt ağacı önce kurmak. Kök sonda dizisi sondan başa okunduğunda önce kök, sonra *sağ* alt ağacın kökü gelir. Bu yüzden kök sonda ile kurulurken dizi sondan başa okunur ve önce sağ alt ağaç kurulmalıdır. Sıra karıştırılırsa sağ alt ağacın öğeleri sol alt ağaca yerleştirilir.

9.6.9 Gezinmelerin maliyeti

Bütün gezinmeler her düğümü bir kez ziyaret eder; ziyaret başına iş sabittir. Zaman maliyetleri bu yüzden aynıdır: $O(n)$. Gezinmeler ek bellekte ayrışır.

Yöntem	Zaman	Ek bellek	Neden?
Özyinelemeli kök önde / ortada / sonda	*O(n)*	*O(h)*	Çağrı yığınında en çok *h + 2* çerçeve (`null` çağrısı dahil)
Yığınla kök önde / ortada / sonda	*O(n)*	*O(h)*	Açık yığın, özyinelemenin yerini tutar
Kuyrukla düzey sırası	*O(n)*	*O(w)*	*w*: en geniş düzeydeki düğüm sayısı

Tablo 9.26: Gezinme yöntemlerinin maliyeti.

Derinlik öncelikli gezinmelerde belleği yükseklik belirler. Dengeli ağaçta $h \approx log_2 n$ olduğundan bellek küçüktür. Çarpık ağaçta ise $h = n - 1$'dir ve bellek *O(n)*'e çıkar. Bu durumda özyinelemeli sürüm, çağrı yığınının sınırına takılıp `StackOverflowError` verebilir; yığınlı sürüm aynı belleği heap'te kullandığı için çalışmayı sürdürür.

Düzey sırasında belleği genişlik belirler. Mükemmel ağaçta son düzeyde yaklaşık *n/2* düğüm vardır; kuyruk bu düzeyi tutarken *O(n)* bellek ister. Çarpık ağaçta ise her düzeyde bir düğüm vardır ve kuyruk hiçbir zaman bir iki öğeden fazlasını tutmaz. İki aile böylece birbirinin tersi davranır. Derin ve dar ağaçta düzey sırası, geniş ve sığ ağaçta derinlik öncelikli gezinme daha az bellek harcar. Örneğin *n* düğümlü mükemmel ağaçta düzey sırasının ek belleği *O(n)*, özyinelemeli kök ortada gezinmenin ek belleği *O(log n)*'dir.

9.7 Sık yapılan hatalar

Ağaç kodlarındaki hataların çoğu birkaç tipe ayrılır. Aşağıdaki tablo, notun içinde ayrı ayrı anlatılan hataları bir arada verir; bir ağaç kodunu gözden geçirirken kontrol listesi olarak kullanılabilir.

Hata	Ne olur?	Doğrusu
Taban durumu unutmak	`k.sol`'a erişmeden önce `k == null` denetlenmezse `NullPointerException`.	Her özyinelemeli ağaç metodu boş ağaç durumuyla başlar.
Yükseklik tanımını karıştırmak	Bütün değerler bir kayar.	Kenar sayısıyla: yaprak 0, boş ağaç −1. Düğüm sayısıyla: yaprak 1, boş ağaç 0. Bir tanım seçilip tutarlı kullanılmalıdır.
Dolu ile tamı aynı saymak	Yanlış sınıflandırma.	Dolu: her düğümde 0 ya da 2 çocuk. Tam: son düzey dışındakiler dolu, son düzey soldan dolar. Biri ötekini gerektirmez.
Tek gezinmeden ağaç kurmak	Birden çok ağaç aynı çıktıyı verir.	Kök ortada ile kök önde ya da kök sonda gerekir. Kök önde ile kök sonda, tek çocuklu düğümlerde belirsizdir.
İndis formüllerini karıştırmak	Yanlış çocuk ya da ebeveyn; kök kendini gösterebilir.	1 tabanlı: *2i*, *2i + 1*, *⌊i/2⌋*. 0 tabanlı: *2i + 1*, *2i + 2*, *⌊(i − 1)/2⌋*.
Derin ağaçta özyineleme	Çarpık ağaçta *h = n − 1*; yüz binlerce çerçeve `StackOverflowError` verir.	Yığınlı ya da kuyruklu sürüm.

Hata	Ne olur?	Doğrusu
Yığınlı kök önde itme sırası	Önce sol itilirse çıktı kök, sağ, sol olur.	Önce sağ, sonra sol itilir.
Yanlış taban değeri	Boş ağaç için en büyük değer olarak 0 döndürülürse negatif değerlerde yanlış sonuç çıkar.	Taban değer birleştirme işleminin etkisiz öğesi olmalıdır (`Integer.MIN_VALUE` , 0, 1).

Tablo 9.27: Ağaçlarla ilgili sık yapılan hatalar.

Tablodaki hataların ortak bir kökü vardır: ağacın özyinelemeli yapısını kodda tam yansıtmamak. Bir ağaç ya boştur ya da bir kök ve iki alt ağaçtan oluşur. Doğru bir ağaç metodu bu iki durumu da açıkça ele alır. Boş ağaç için anlamlı bir değer döndürür. Boş olmayan ağaçta alt ağaçların sonuçlarını, kökün nerede yer alacağına dikkat ederek birleştirir. Gezinmeler arasındaki fark da yalnız bu son noktadadır: kök, alt ağaçların önünde mi, arasında mı, arkasında mı?

9.8 Özet

- **Ağaç**, düğümler ve onları ebeveyn–çocuk ilişkisiyle bağlayan kenarlardan oluşan hiyerarşik bir yapıdır. Tek bir kök vardır; kök dışındaki her düğümün tam olarak bir ebeveyni vardır. Bu yüzden döngü yoktur, n düğümlü ağaçta $n - 1$ kenar bulunur ve iki düğüm arasında tek bir yol vardır.
- **İç düğüm** en az bir çocuğu olan, **yaprak** hiç çocuğu olmayan düğümdür. Ebeveyn, çocuk, kardeş, ata ve soy aile ağacındaki anlamlarıyla kullanılır; aynı düzeyde olmak kardeş olmak demek değildir.
- **Derece** çocuk sayısıdır. **Derinlik** kökten düğüme, **yükseklik** düğümden en uzak yaprağa kenar sayısıdır. Ağacın yüksekliği kökün yüksekliği, yani en büyük derinliktir. Düğüm sayısıyla ölçen kaynaklarda bütün değerler bir fazladır.
- **İkili ağaçta** her düğümün en çok iki çocuğu vardır ve sol ile sağ ayrılır. **Dolu**: her düğümde 0 ya da 2 çocuk. **Tam**: son düzey dışındakiler dolu, son düzey soldan dolar. **Mükemmel**: bütün yapraklar aynı derinlikte; hem dolu hem tam.
- İkili ağaçta d derinliğinde en çok 2^d, yüksekliği h olan ağaçta en çok $2^{h+1} - 1$ düğüm bulunur; $\lfloor \log_2 n \rfloor \le h \le n - 1$. Dolu ikili ağaçta yaprak sayısı iç düğüm sayısından bir fazladır.
- **Dizi gösteriminde** 1 tabanlı indis i'nin çocukları $2i$ ve $2i + 1$, ebeveyni $\lfloor i/2 \rfloor$'dir; tam ağaçlarda (heap) idealdir, çarpık ağaçlarda $2^n - 1$ hücreye kadar israf eder. **Bağlı gösterimde** her düğüm sol ve sağ referans taşır; $n + 1$ referans `null`'dır. Genel ağaçlar çocuk listesiyle ya da ilk çocuk – sonraki kardeş gösterimiyle saklanır.
- **Kök önde** K-S-S, **kök ortada** S-K-S, **kök sonda** S-S-K sırasıyla gezer; üçü de aynı Euler turunun farklı anlarıdır. **Düzey sırası** ağacı düzey düzey bir kuyrukla tarar.
- Özyinelemeli gezinmeler çağrı yığınında en çok $h + 2$ çerçeve kullanır. Açık yığınlı sürümler aynı işi heap bellekte yapar: kök önde için önce sağ, sonra sol itilir; kök ortada sola iner ve çıkarırken yazar; kök sonda `peek` ile sağ alt ağacın bitip bitmediğine bakar.

- İfade ağacında kök önde önek, kök sonda sonek, parantezli kök ortada ara ek gösterimi verir; değer kök sonda mantığıyla hesaplanır. `boyut`, `yukseklik`, `yaprakSayisi` ve `enBuyukBul` aynı kalıbı izler: boş ağaç için taban değer, sonra alt ağaç sonuçlarının birleştirilmesi.
- Kök ortada ile kök önde (ya da kök sonda) çıktıları, değerleri farklı bir ikili ağacı tek biçimde belirler. Bütün gezinmeler *O(n)* zaman ister; ek bellek derinlik öncelikli gezinmelerde *O(h)*, düzey sırasında *O(w)*'dir.

Konu	Akılda kalması gerekenler
Terimler	kök, ebeveyn/çocuk, yaprak, iç düğüm; *n* düğüm → *n* − *1* kenar
Ölçüler	derece = çocuk sayısı; derinlik = kökten kenar sayısı; yükseklik = en uzun aşağı yol
İkili ağaç	dolu: 0/2 çocuk · tam: son düzey soldan dolar · mükemmel: bütün düzeyler dolu
Gösterim	dizi: indis bağıntıları · bağlı: sol/sağ referansları
Gezinme	kök önde K-S-S · kök ortada S-K-S · kök sonda S-S-K · düzey sırası kuyrukla
Maliyet	bütün gezinmeler *O(n)*; ek bellek *O(h)* (yığın) ya da *O(w)* (kuyruk)

Tablo 9.28: Bölümün özeti.

9.9 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

ağaç *tree* · düğüm *node* · kenar *edge* · kök *root* · ebeveyn *parent* · çocuk *child* · kardeş *sibling* · ata *ancestor* · soy *descendant* · yaprak *leaf* · iç düğüm *internal node* · derece *degree* · derinlik *depth* · düzey *level* · yükseklik *height* · yol *path* · alt ağaç *subtree* · döngü *cycle* · konum *position* · soyut veri tipi *abstract data type (ADT)* · ikili ağaç *binary tree* · dolu ikili ağaç *full binary tree* · tam ikili ağaç *complete binary tree* · mükemmel ikili ağaç *perfect binary tree* · ikili arama ağacı *binary search tree (BST)* · heap *heap* · gezinme *traversal* · kök önde *preorder* · kök ortada *inorder* · kök sonda *postorder* · düzey sırası *level-order* · genişlik öncelikli arama *breadth-first search (BFS)* · derinlik öncelikli arama *depth-first search (DFS)* · ifade ağacı *expression tree* · özyineleme *recursion* · taban durum *base case* · çağrı yığını *call stack* · yığın çerçevesi *stack frame* · kuyruk *queue*

9.10 Kendini Dene

1. Bir genel ağaçta A köktür; A'nın çocukları B ve C, B'nin çocukları D, E, F, C'nin çocuğu G, G'nin çocukları H ve I'dır. Yapraklar ve iç düğümler hangileridir? Ağacın derecesi, kenar sayısı ve yüksekli-

ği kaçtır? G'nin derinliği ve yüksekliği nedir?

Cevap

Yapraklar D, E, F, H, I; iç düğümler A, B, C, G'dir (5 + 4 = 9 düğüm). En büyük derece B'nindir (3); ağacın derecesi 3'tür. Kenar sayısı $9 - 1 = 8$'dir. En uzun yol A→C→G→H'dir; ağacın yüksekliği 3'tür. G'nin derinliği 2 (A→C→G), yüksekliği 1'dir (G→H).

2. Yüksekliği 5 olan bir ikili ağaçta en az ve en çok kaç düğüm bulunabilir? 100 düğümlü bir ikili ağacın yüksekliği en az kaç olabilir?

Cevap

En az $h + 1 = 6$ (zincir), en çok $2^6 - 1 = 63$ düğüm (mükemmel ağaç). 100 düğüm için yükseklik 5 yetmez (en çok 63 düğüm), 6 yeter (en çok 127 düğüm); en az yükseklik $\lfloor \log_2 100 \rfloor = 6$'dır.

3. 21 düğümlü dolu bir ikili ağaçta kaç yaprak ve kaç iç düğüm vardır? Aynı ağaç bağlı gösterimde saklanırsa kaç `null` çocuk referansı bulunur?

Cevap

$n = 2i + 1$'den $i = 10$, $y = i + 1 = 11$'dir. `null` referans sayısı $n + 1 = 22$'dir. Denetim: her yaprağın iki `null` referansı vardır, $2 \cdot 11 = 22$; iç düğümlerin iki çocuğu da dolu olduğu için başka `null` yoktur.

4. 0 tabanlı dizi gösteriminde indis 9'daki düğümün ebeveyninin ve çocuklarının indisleri nedir? Aynı düğüm 1 tabanlı dizide hangi indiste durur ve bu indislerin karşılıkları nedir?

Cevap

0 tabanlı: ebeveyn $\lfloor (9 - 1)/2 \rfloor = 4$, sol çocuk $2 \cdot 9 + 1 = 19$, sağ çocuk 20. 1 tabanlıda aynı düğüm indis 10'dadır: ebeveyn $\lfloor 10/2 \rfloor = 5$, çocuklar 20 ve 21. Her 1 tabanlı indis, 0 tabanlı karşılığından tam bir fazladır.

5. Kök 10; 10'un çocukları 5 ve 15; 5'in çocukları 2 ve 7; 7'nin yalnız sol çocuğu 6; 15'in yalnız sağ çocuğu 20 olan ağacın kök önde, kök ortada, kök sonda ve düzey sırası çıktıları nedir? Bu ağaç bir ikili arama ağacı mıdır?

Cevap

Kök önde: `10 5 2 7 6 15 20`. Kök ortada: `2 5 6 7 10 15 20`. Kök sonda: `2 6 7 5 20 15 10`. Düzey sırası: `10 5 15 2 7 20 6`. Kök ortada çıktısı küçükten büyüğe sıralı olduğu için ağaç bir ikili arama ağacıdır.

6. Bir önceki sorudaki ağaçta `kokBastaYigin` çalıştırılıyor. Her turun sonunda yığında (alttan üste) neler vardır? Yığın en çok kaç öğe tutar?

   Cevap

   Başta `[10]`. 10 çıkar, 15 ve 5 itilir: `[15, 5]`. 5 çıkar, 7 ve 2 itilir: `[15, 7, 2]`. 2 çıkar: `[15, 7]`. 7 çıkar, 6 itilir: `[15, 6]`. 6 çıkar: `[15]`. 15 çıkar, 20 itilir: `[20]`. 20 çıkar: `[]`. Yığın en çok 3 öğe tutar; çıktı kök önde sırasıdır.

7. *((9 − 3) / (1 + 2)) × 4* ifadesinin ağacında kök hangi işleçtir? Önek ve sonek gösterimleri nedir, `hesapla` hangi değeri döndürür?

   Cevap

   En son yapılan işlem çarpma olduğu için kök `×`'tir; sol alt ağacın kökü `/`, sağ çocuğu 4'tür. Önek: `× / − 9 3 + 1 2 4`. Sonek: `9 3 − 1 2 + / 4 ×`. `hesapla` önce *9 − 3 = 6*, sonra *1 + 2 = 3*, *6 / 3 = 2*, en son *2 × 4 = 8* hesaplar ve 8 döndürür.

8. Kök ortada çıktısı `8 4 9 2 5 1 3 7`, kök sonda çıktısı `8 9 4 5 2 7 3 1` olan ağaç hangisidir? Kök önde çıktısı nedir?

   Cevap

   Kök sonda çıktısının son öğesi kök: 1. Kök ortada 1'in solu `8 4 9 2 5`, sağı `3 7`'dir. Sağ alt ağaçta kök sonda sırasının sondan bir önceki öğesi 3 köktür; 7 onun sağındadır. Sol alt ağaçta kök 2'dir; solunda `8 4 9`, sağında 5 vardır. `8 4 9`'un kökü 4'tür; 8 sol, 9 sağ çocuktur. Ağaç 1(2(4(8, 9), 5), 3(—, 7))'dir. Kök önde: `1 2 4 8 9 5 3 7`.

9. Kök önde çıktısı `A B`, kök sonda çıktısı `B A` olan kaç farklı ikili ağaç vardır? Hangi ek bilgi ağacı belirler?

   Cevap

   İki ağaç vardır: B, A'nın sol çocuğu ya da sağ çocuğu olabilir; iki ağacın kök önde ve kök sonda çıktıları aynıdır. Kök ortada çıktısı ayrımı yapar: B sol çocuksa `B A`, sağ çocuksa `A B` çıkar. Belirsizliğin nedeni A'nın tek çocuklu olmasıdır.

10. `yukseklik` metodunda taban durum `return -1` yerine `return 0` yazılmış olsun. 9 kökündeki dört düğümlü ağaçta (9'un çocukları 2 ve 3; 2'nin sol çocuğu 4) metot ne döndürür? Bir yaprak

için ne döndürür?

Cevap

Yaprak için *1 + max(0, 0) = 1* döner. 2 için *1 + max(1, 0) = 2*, 3 için 1, kök için *1 + max(2, 1) = 3* döner. Metot artık yüksekliği düğüm sayısıyla ölçer: her değer kenar tanımındakinden bir fazladır (doğru yanıt 2'dir).

9.11 İleri okuma

- M. T. Goodrich, R. Tamassia ve M. H. Goldwasser, *Data Structures and Algorithms in Java* (6. baskı, 2014), Bölüm 8: genel ve ikili ağaç ADT'lerini konum soyutlamasıyla tanımlar, bağlı ve dizi gerçeklemelerini verir; gezinmeleri, Euler turunu ve ifade ağaçlarını bu nottakine yakın bir Java koduyla anlatır.
- D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms* (3. baskı, 1997), Kesim 2.3: ağaç gezinmelerinin klasik incelemesidir; özyinelemesiz gezinme algoritmalarını, kılavuzlu ikili ağaçları ve genel ağaçların ikili ağaçlarla gösterimini ayrıntılı işler.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 10'daki köklü ağaçların gösterimi kesimi ve Ek B.5: ilk çocuk – sonraki kardeş gösterimini ve ağaçlarla ilgili tanım ve sayma sonuçlarını daha biçimsel bir dille verir.

9.12 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022.
2. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6. baskı, Wiley, 2014.
3. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3. baskı, Addison-Wesley, 1997.
4. Oracle, *Java SE 17 API Specification*: `java.util.Deque`, `java.util.ArrayDeque`, `java.util.Queue` ve `java.util.LinkedList` belgeleri.
5. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011.

KISIM III · BÖLÜM 10

Arama Ağaçları

Sıralı bir dizide ikili arama çok hızlıdır, ama araya yeni bir öğe koymak bütün öğeleri kaydırmayı gerektirir. Bağlı listeye eklemek kolaydır, ama orada arama baştan sona yürümek demektir. Arama ağaçları bu iki dünyanın iyi yanlarını birleştirir. Öğeler anahtarlarına göre bir ağaçta öyle yerleştirilir ki her düğümde "sola mı, sağa mı?" sorusunun tek bir cevabı olur. Böylece arama, ekleme ve silme kökten bir yaprağa inen tek bir yol üzerinde yürür ve maliyet ağacın yüksekliğine bağlı kalır. Not, ikili arama ağacının (BST) tanımı ve temel işlemleriyle başlar. Ardından yüksekliğin neden her şeyi belirlediğini gösterir ve yüksekliği kendiliğinden küçük tutan AVL, kırmızı-siyah ve splay ağaçlarını anlatır. Son kesim, disk üzerindeki büyük veriler için geliştirilen B-ağacını ve B+ ağacını anlatır.

Bu bölümün kodu: `github.com/sercankulcu/data-structures-java/Bolum10/src`

ÖĞRENME ÇIKTILARI

BST işlemleri
Arama, ekleme ve silme işlemlerini ikili arama ağacında uygulamak; ardıl, öncül ve aralık sorgularını yapmak.

Denge gereksinimi
Ağaç yüksekliğinin işlem maliyetlerine etkisini açıklamak; aynı anahtarların farklı sıralarla eklenince neden çok farklı ağaçlar verdiğini göstermek.

Dengeleme
AVL döndürmelerini ve kırmızı-siyah ağaç kurallarını yorumlamak; bir ekleme dizisinden sonra oluşan dengeli ağacı adım adım bulmak.

Karşılaştırma
BST, AVL, kırmızı-siyah, splay ve B-ağaçlarını kullanım amacına göre seçmek.

10.1 Arama ağaçları

Bir telefon rehberinde “Yılmaz” soyadını arayan kimse rehberi ilk sayfadan okumaya başlamaz. Rehberi ortadan açar, soyadın o sayfadan önce mi sonra mı geldiğine bakar ve aramayı yarısına indirir. Bu yöntem rehber sıralı olduğu için işler. Bilgisayarda aynı yöntemin adı ikili aramadır ve sıralı bir dizide n öğe arasında aranan öğeyi yaklaşık $\log_2 n$ karşılaştırmada bulur.

Sıralı dizinin zayıf yanı güncellemedir. Rehbere yeni bir kişi eklemek için ondan sonra gelen bütün kayıtların bir sıra kaydırılması gerekir; bu, en kötü durumda n kaydırma demektir. Bağlı listede ekleme ve silme ucuzdur, ama listenin ortasına doğrudan atlanamaz; arama baştan başlayıp öğe öğe ilerler. Arama ağaçları bu iki yapının arasında durur. Bağlı yapıları sayesinde ekleme ve silme yalnız birkaç bağı değiştirir. Sıralı düzenleri sayesinde de arama her adımda kalan adayların bir kısmını eler.

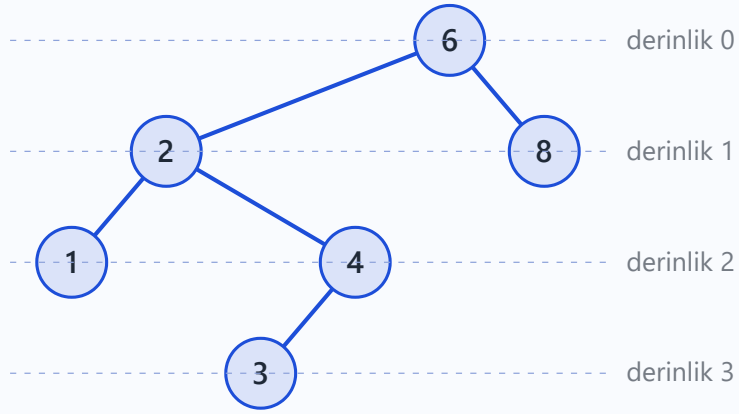
10.1.1 Arama ağacı nedir?

Ağaçla ilgili temel kavramları kısaca hatırlayalım. Bir ağaç, düğümlerden ve düğümleri birbirine bağlayan kenarlardan oluşur. En üstteki düğüme **kök** (*root*), çocuğu olmayan düğümlere **yaprak** (*leaf*) denir. Bir düğümün **derinliği** (*depth*), kökten o düğüme inen yoldaki kenar sayısıdır; kökün derinliği 0'dır. Ağacın **yüksekliği** (*height*) ise en derin düğümün derinliğidir. Yükseklik genellikle h ile gösterilir. Tek düğümlü ağacın yüksekliği 0, boş ağacınki ise -1 kabul edilir.

Arama ağacında her düğüm bir öğe taşır ve her öğenin bir **anahtarı** (*key*) vardır. Anahtar, öğeyi ötekilerden ayıran ve karşılaştırılabilen değerdir: öğrenci kaydında öğrenci numarası, sözlükte kelimenin kendisi, rehberde soyadı. Öğenin geri kalanı (ad, adres, not ortalaması) anahtara bağlı olarak saklanır. Bu notta örnekleri sade tutmak için anahtar çoğunlukla bir tam sayıdır ve düğümde yalnız o tam sayı vardır.

> **Tanım 10.1**
>
> **Arama ağacı** (*search tree*), her düğümü bir öğe taşıyan ve öğeleri anahtarlarına göre sıralı bir düzende tutan ağaçtır. Düzen öyle kurulur ki bir anahtarı ararken her düğümde yalnız bir alt ağaca inmek yeter. Bu yüzden arama kökten başlar, her düzeyde tek bir dal seçer ve en çok $h + 1$ düğüm ziyaret eder.

Tanımın son cümlesi bölümün bütün hikâyesini özetler. Arama, ekleme ve silme kökten aşağı tek bir yol izler. Yolun uzunluğu en çok ağacın yüksekliği kadardır. Öyleyse bu işlemlerin maliyeti $O(h)$'dir. Yükseklik küçük kalırsa işlemler hızlı olur, büyürse yavaşlar.



Şekil 10.1: Kökü 6 olan altı düğümlü bir arama ağacı. Kökün derinliği 0, 2 ile 8'in derinliği 1, 1 ile 4'ün derinliği 2, 3'ün derinliği 3'tür. En derin düğüm 3 olduğu için ağacın yüksekliği $h = 3$'tür.

n düğümlü bir ağacın yüksekliği ne kadar olabilir? İki uç durum vardır. En kötü durumda her düğümün tek çocuğu olur ve ağaç bir zincire dönüşür; bu durumda yükseklik $n - 1$'dir. En iyi durumda her düzey tamamen doludur. Derinliği d olan düzeyde en çok 2^d düğüm bulunabilir. Öyleyse yüksekliği h olan ikili ağaçta en çok $1 + 2 + 4 + \dots + 2^h = 2^{h+1} - 1$ düğüm vardır. $n \le 2^{h+1} - 1$ eşitsizliğinden $h \ge \log_2(n + 1) - 1$ çıkar; yani en iyi durumda yükseklik yaklaşık $\log_2 n$'dir.

Örnek 10.1

Bir milyon ($n = 10^6$) anahtar bir arama ağacında tutulsun. Ağaç tam dengeliyse yükseklik yaklaşık $\log_2 10^6 \approx 19{,}93$, yani 20 civarındadır; bir arama en çok 20 küsur karşılaştırma yapar. Ağaç bir zincire dönüşmüşse yükseklik 999.999'dur ve en kötü arama bir milyon karşılaştırma yapar. Aynı anahtarlar, aynı işlem; tek fark ağacın şeklidir. Aradaki oran yaklaşık 50.000 kattır.

Bu fark, bölümün geri kalanındaki bütün yapıların varlık nedenidir. Düz ikili arama ağacı yükseklik için hiçbir söz vermez. AVL ve kırmızı-siyah ağaçlar, her güncellemeden sonra ağacı küçük yerel değişikliklerle düzelterek $h = O(\log n)$ garantisi verir. Splay ağacı tek tek işlemler için söz vermez, ama uzun işlem dizilerinin ortalamasında aynı sınıra ulaşır. B-ağacı ise düğüm başına çok anahtar tutarak ağacı hem kısa hem geniş yapar.

Sık yapılan hata

Yüksekliği düğüm sayısıyla saymak. Yükseklik, kökten en derin yaprağa inen yoldaki *kenar* sayısıdır. Şekil 10.1'deki ağaçta 6 → 2 → 4 → 3 yolunda dört düğüm, üç kenar vardır; yükseklik 3'tür. Bazı kitaplar yüksekliği düğüm sayısıyla tanımlar; o zaman bütün formüller bir kayar. Bu notta yükseklik her yerde kenar sayısıdır ve boş ağacın yüksekliği −1'dir.

10.1.2 Temel işlemler

Bir arama ağacından beklenen işlemler, sıralı bir koleksiyondan beklenenlerle aynıdır. Tablo 10.1 bunları ve bu notta kullanılan metot adlarını verir.

İşlem	Ne yapar?	Örnek
Arama (*search*)	Verilen anahtar ağaçta varsa `true`, yoksa `false` döndürür.	`ara(kok, 5)` → `true`
Ekleme (*insertion*)	Yeni öğeyi sıralama düzenini bozmadan uygun yaprak konumuna ekler.	`ekle(kok, 7)`
Silme (*deletion*)	Verilen anahtarlı öğeyi çıkarır ve düzeni korur.	`sil(kok, 40)`
Gezinme (*traversal*)	Bütün öğeleri belirli bir sırayla ziyaret eder. Kök ortada gezinme anahtarları sıralı verir.	`inorder(kok)` → 1 2 3 4 6 8

Tablo 10.1: Arama ağacının temel işlemleri. Örnekler sunumdaki ağaçlara aittir.

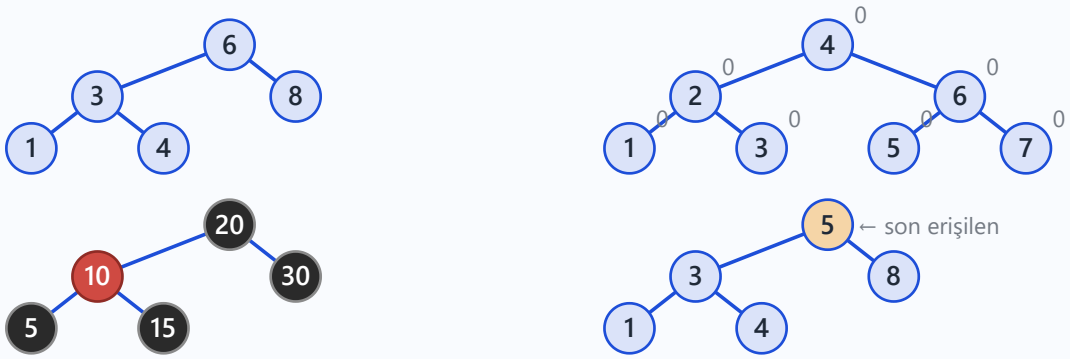
Sıralı düzen birkaç sorguyu daha mümkün kılar. En küçük ve en büyük anahtar bunların ilk ikisidir. Bir anahtarın ardılı kendisinden büyük en küçük anahtar, öncülü ise kendisinden küçük en büyük anahtardır. Son sorgu, belli bir aralığa düşen bütün anahtarları ister. Bu sorguların ortak noktası, hepsinin de kökten tek bir yol izlemesidir. Gezinme dışındaki bütün işlemler bu yüzden $O(h)$ sürer. Gezinme ise her düğümü bir kez ziyaret ettiği için $O(n)$'dir.

Hash tablosu da arama ve eklemeyi ortalama sabit sürede yapar. Arama ağacını ondan ayıran şey bu sıralı sorgulardır. Hash tablosu anahtarları dağıtarak saklar. “50'den büyük en küçük anahtar hangisi?” ya da “20 ile 45 arasındaki anahtarları sırayla ver” gibi sorular için bütün tabloyu taraması gerekir. Arama ağacı ise bu soruları da $O(h)$ ya da $O(h + s)$ sürede cevaplar; s burada döndürülen anahtar sayısıdır.

10.1.3 Arama ağacı türleri

Bölümde dört ikili arama ağacı türü ve iki çok yollu ağaç işlenir. İkili türlerin hepsinde aynı sıralama kuralı geçerlidir; farkları, yüksekliği nasıl ve ne ölçüde denetledikleridir. Şekil 10.2 dört ikili türün küçük birer örneğini gösterir.

- **İkili arama ağacı** (*binary search tree*, kısaca BST): Her düğümün anahtarı, sol alt ağacındaki anahtarların hepsinden büyük, sağ alt ağacındakilerin hepsinden küçüktür. Denge garantisi yoktur; şekli ekleme sırasına bağlıdır.
- **AVL ağacı**: Her düğümde iki alt ağacın yükseklik farkı, yani **denge faktörü** en çok 1'dir. Bu koşul bozulunca döndürmelerle düzeltilir.
- **Kırmızı-siyah ağaç** (*red-black tree*): Düğümler kırmızı ya da siyaha boyanır. Renk kuralları yüksekliği en çok $2 \log_2(n + 1)$ ile sınırlar.
- **Splay ağacı** (*splay tree*): Erişilen düğümü döndürmelerle köke taşır. Tek bir işlem pahalı olabilir, ama işlem başına **amortize** maliyet $O(\log n)$'dir.



Şekil 10.2: Dört ikili arama ağacı türü. (a) BST: 6 kökte; solunda 3, 1, 4, sağında 8. (b) AVL: yedi düğümlü, her düğümde denge faktörü 0 olan mükemmel dengeli ağaç. (c) Kırmızı-siyah: 20 siyah kök, 10 kırmızı, 30, 5 ve 15 siyah. (d) Splay: en son erişilen 5 kökte duruyor.

Çok yollu ağaçlarda (B-ağacı ve B+ ağacı) bir düğüm birden çok anahtar ve ikiden çok çocuk taşır. Bu ağaçlar özellikle verinin diskte durduğu durumlar için tasarlanmıştır ve 10.7'nin konusudur.

10.2 İkili arama ağacı

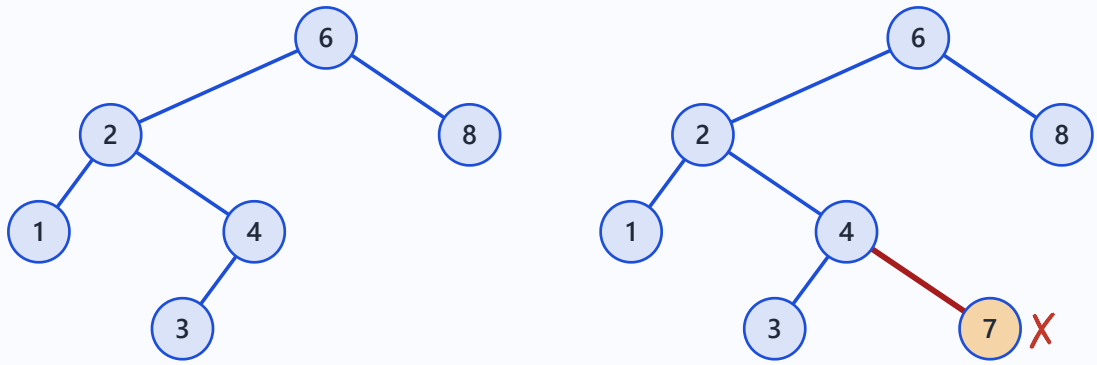
İkili ağaçta her düğümün en çok iki çocuğu vardır: sol çocuk ve sağ çocuk. İkili arama ağacı, ikili ağaca bir sıralama kuralı ekler. Bu kural sayesinde ağaç, sıralı bir dizinin "bağlı" bir karşılığı gibi davranır.

10.2.1 BST özelliği

Tanım 10.2

İkili arama ağacı (*binary search tree*, BST), her düğümü x için şu koşulu sağlayan ikili ağaçtır. x'in sol alt ağacındaki bütün anahtarlar $x.anahtar$'dan küçüktür. Sağ alt ağacındaki bütün anahtarlar ise $x.anahtar$'dan büyüktür. Bu koşula **BST özelliği** denir.

Tanımdaki "bütün" sözcüğü önemlidir. Koşul yalnız düğüm ile çocukları arasında değil, düğüm ile soyundaki bütün düğümler arasında da geçerlidir. Kökün sol alt ağacındaki en derin yaprak bile kökten küçük olmak zorundadır. Şekil 10.3 bu farkı gösterir. Soldaki ağaç geçerli bir BST'dir. Sağdakinde 4'ün sağ çocuğu olarak 7 eklenmiştir. Yalnız 4 ile 7'ye bakılırsa sorun yoktur: 7, 4'ün sağındadır ve 4'ten büyüktür. Ama 7, 6'nın sol alt ağacındadır ve 6'dan büyüktür. Bu yüzden sağdaki ağaç BST değildir.



Şekil 10.3: Solda geçerli bir ikili arama ağacı. Sağda aynı ağaçta 4'ün sağ çocuğu olarak 7 eklenmiş. 7, 4'ten büyük olduğu için ebeveyniyle sorun yok; ama 6'nın sol alt ağacında olduğu hâlde 6'dan büyük. Ağaç BST değildir.

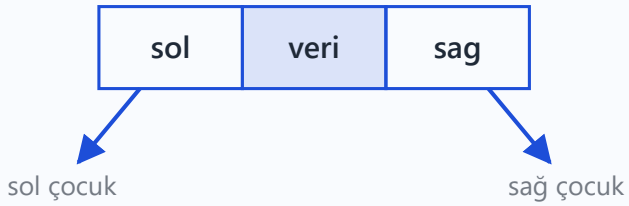
BST özelliğinin doğrudan bir sonucu şudur: bir düğümün anahtarı, ağacın kalanını iki parçaya böler. Küçük anahtarların hepsi solda, büyüklerin hepsi sağdadır. Bu, sıralı dizide ortadaki öğenin diziyi ikiye bölmesine benzer. İkili aramanın her adımda aralığı yarıya indirmesi gibi, BST'de de her düğümde aramanın gideceği taraf kesinleşir ve öbür taraf tamamen elenir.

Aynı anahtar ağaçta iki kez bulunabilir mi? Tanım katı eşitsizlik kullandığı için hayır. Uygulamada iki yol vardır. Birincisi, tekrarlayan anahtarları reddetmektir; bu notun kodları böyle davranır. İkincisi, eşit anahtarları tutarlı biçimde hep aynı tarafa, örneğin hep sağa koymaktır. Hangisi seçilirse seçilsin, arama, ekleme ve silme aynı kurala uymalıdır. Yoksa eklenen bir anahtar sonra bulunamayabilir.

> **Sık yapılan hata**
>
> BST özelliğini yalnız ebeveyn ile çocuk arasında denetlemek. “Her düğümün sol çocuğu küçük, sağ çocuğu büyük” koşulu Şekil 10.3'teki sağ ağaçta da sağlanır; ama ağaç BST değildir. Doğru denetim, her düğüm için izin verilen bir alt ve üst sınırı kökten aşağı taşır (10.3.12).

10.2.2 Düğüm yapısı

BST'nin her düğümü üç alandan oluşur: veri (anahtar), sol çocuğa bir referans ve sağ çocuğa bir referans (Şekil 10.4). Çocuğu olmayan tarafta referans `null`'dır. Yaprakta iki referans da `null`'dır. Bağlı listedeki düğümle karşılaştırırsak tek fark, bir “sonraki” yerine iki çocuk referansının bulunmasıdır.



Şekil 10.4: İkili ağaç düğümünün üç alanı: sol çocuk referansı, veri ve sağ çocuk referansı.

Sunumda düğüm iki biçimde yazılır. İlki yalnız `int` anahtar tutan sade bir sınıftır:

```
class AgacDugumu {
    int veri;
    AgacDugumu sol, sag;

    AgacDugumu(int veri) {
        this.veri = veri;
        this.sol = null;
        this.sag = null;
    }
}
```

Yapıcı, veriyi yazar ve iki çocuğu açıkça `null` yapar. Java'da referans türündeki alanlar zaten `null` ile başladığı için son iki atama zorunlu değildir; okuyana niyeti göstermek için yazılmıştır. Ardıl, aralık sorgusu ve BST doğrulama kodları bu sınıfı kullanır; alan adları kısadır: `sol` ve `sag`.

İkinci biçim jenerik (*generic*) sınıftır. Anahtar tam sayı olmak zorunda değildir; karşılaştırılabilen her tür olabilir:

```
class IkiliAgacDugum<E extends Comparable<E>> {
    E veri;                    // nesne
    IkiliAgacDugum<E> solCocuk; // sol çocuk
    IkiliAgacDugum<E> sagCocuk; // sağ çocuk

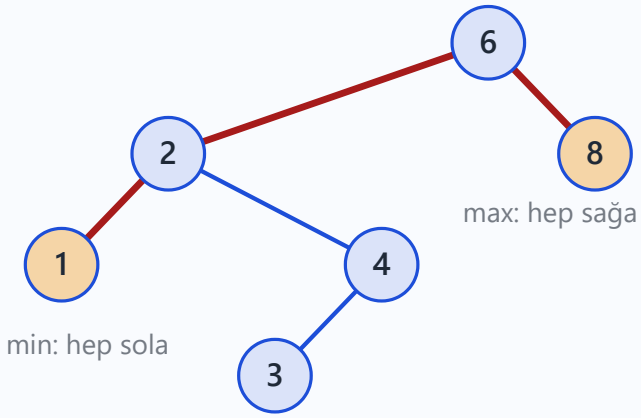
    IkiliAgacDugum(E veri) {
        this.veri = veri;
        // çocuklar başlangıçta null
    }
}
```

Tür parametresi `E extends Comparable<E>` biçiminde sınırlanmıştır. Bu, "`E` türünden iki nesne `compareTo` ile karşılaştırılabilmelidir" demektir. `Integer`, `String` ve `LocalDate` gibi sınıflar `Comparable`'ı gerçekler; bu yüzden hepsi anahtar olabilir. `a.compareTo(b)` çağrısı $a < b$ ise negatif, eşitse 0, $a > b$ ise pozitif bir sayı döndürür. Arama, ekleme ve silme kodları karar verirken yalnız bu işarete bakar. Bu sınıfta alan adları `solCocuk` ve `sagCocuk`'tur.

Bellek açısından her düğüm, verinin yanında iki referans taşır. *n* düğümlü ağaç bu yüzden *O(n)* bellek kullanır. Dengeli ağaç türleri düğüme küçük bir ek bilgi ekler: AVL ağacında yükseklik, kırmızı-siyah ağaçta bir bitlik renk. Bu ekler de düğüm başına sabittir; toplam bellek yine *O(n)* kalır.

10.2.3 En küçük ve en büyük anahtar

BST özelliğinden ilk çıkan sonuç, en küçük ve en büyük anahtarın yeridir. Bir düğümün sol alt ağacında ondan küçük anahtarlar vardır. Öyleyse sol çocuğu olan bir düğüm en küçük olamaz; daha küçüğü solundadır. Bu akıl yürütme en küçük anahtara kökten başlayıp hep sola inerek ulaşıldığını gösterir. İniş, sol çocuğu olmayan ilk düğümde durur; o düğüm en küçüktür. En büyük için aynı yol sağa doğru izlenir (Şekil 10.5).



Şekil 10.5: En küçük anahtar 1'e kökten hep sola (6 → 2 → 1), en büyük anahtar 8'e hep sağa (6 → 8) inilerek ulaşılır.

Sunumdaki kod iki metodu bir arada verir:

```
IkiliAgacDugum<E> minDeger(
        IkiliAgacDugum<E> dugum) {
    while (dugum.solCocuk != null)
        dugum = dugum.solCocuk; // sola in
    return dugum;  // en küçük
}
IkiliAgacDugum<E> maxDeger(
        IkiliAgacDugum<E> dugum) {
    while (dugum.sagCocuk != null)
        dugum = dugum.sagCocuk; // sağa in
    return dugum;  // en büyük
}
```

`minDeger`, kendisine verilen düğümden başlar. Döngü koşulu "bu düğümün sol çocuğu var mı?" sorusudur. Varsa `dugum` sol çocuğa geçer. Sol çocuk `null` olunca döngü biter ve o anki düğüm döndürülür. `maxDeger` aynı işi sağ çocuklarla yapar. İki metodun da ön koşulu ağacın boş olmamasıdır; `dugum` başlangıçta `null` olursa `dugum.solCocuk` ifadesi `NullPointerException` fırlatır.

Metotlar hiç anahtar karşılaştırması yapmaz; yalnız bağları izler. Her döngü turunda bir düzey aşağı inilir. En çok *h* tur dönülür, dolayısıyla süre *O(h)*'dir. Ek bellek olarak yalnız bir değişken kullanılır: *O(1)*.

Metotların parametre olarak kökü değil herhangi bir düğümü alması bilinçli bir seçimdir. Silme kodu, iki çocuklu bir düğümün yerine koyacağı değeri bulmak için `minDeger`'i o düğümün sağ alt ağacının kökünden başlatır. Yani aynı metot "bütün ağacın en küçüğü" için de "bir alt ağacın en küçüğü" için de kullanılır.

> **Sık yapılan hata**
>
> Döngü koşulunu `while (dugum != null)` diye yazmak. Bu döngü en soldaki düğümü de geçer ve `null`'a ulaşınca durur; metot her zaman `null` döndürür. Doğru koşul bir adım önce durur: sol çocuğu olmayan düğümde. "Bir sonraki var mı?" diye sorup ondan sonra ilerlemek gerekir.

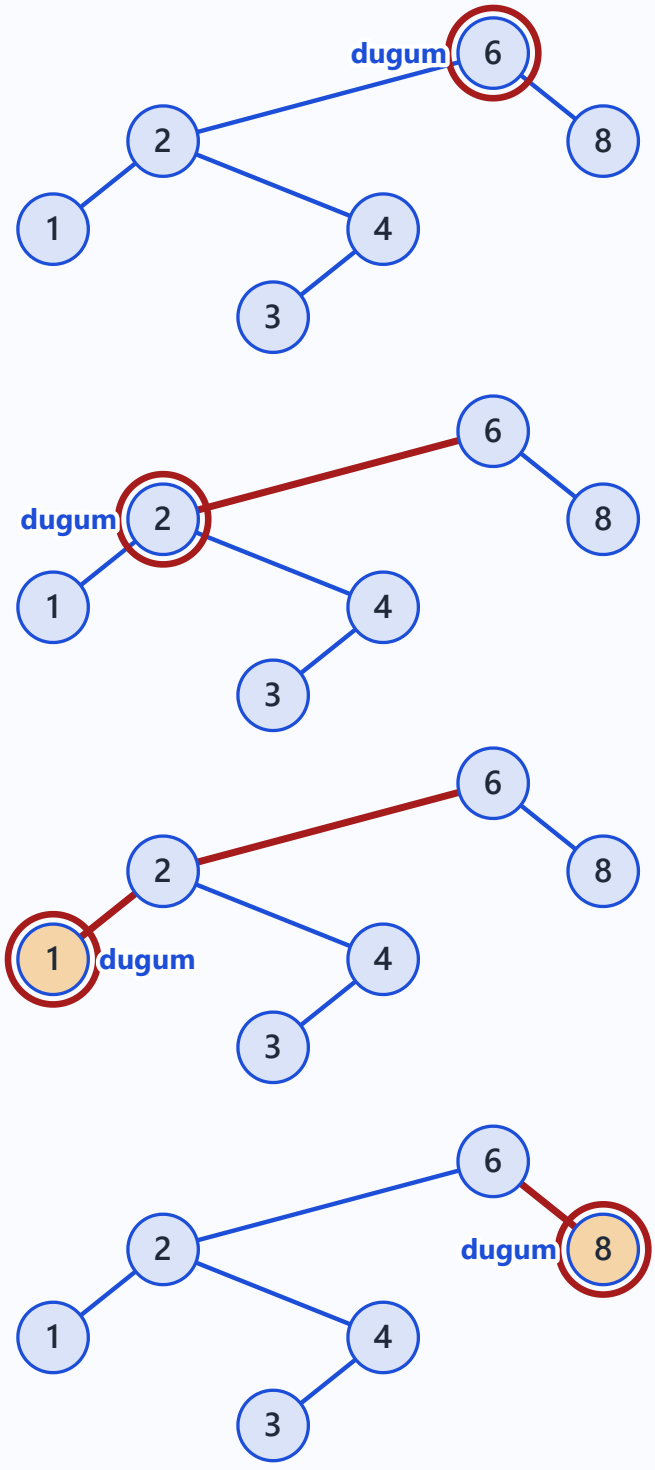
10.2.4 Kod izleme: minDeger ve maxDeger

İzleme, Şekil 10.5'teki ağaç üzerinde yapılır. Ağaç 6, 2, 8, 1, 4, 3 sırasıyla eklenerek oluşmuştur: kök 6; solunda 2, sağında 8; 2'nin çocukları 1 ve 4; 4'ün sol çocuğu 3. Tablo 10.2 her döngü turunda `dugum`'un nerede olduğunu ve koşulun nasıl değerlendirildiğini verir.

Çağrı	Adım	`dugum`	Koşul	Sonuç
`minDeger`	1	6	`solCocuk` = 2 ≠ null	sola in
	2	2	`solCocuk` = 1 ≠ null	sola in
	3	1	`solCocuk` = null	döngü biter, `return` 1
`maxDeger`	1	6	`sagCocuk` = 8 ≠ null	sağa in
	2	8	`sagCocuk` = null	döngü biter, `return` 8

Tablo 10.2: `minDeger(kok)` ve `maxDeger(kok)` izlemesi.

İzlemede dikkat çekici bir nokta vardır. Ağaçtaki en derin düğüm 3'tür, ama 3 en küçük değildir. 3, 4'ün solundadır; 4 de 2'nin sağındadır. Yani 3, 2'den büyüktür. En küçük anahtar en derin düğümde değil, *en soldaki* düğümdedir. Şekil 10.6 `minDeger`'in üç anını ve `maxDeger`'in sonucunu gösterir.



Şekil 10.6: Kod izlemesinin dört anı. (a) `minDeger(kok)` : `dugum` kökten (6) başlar. (b) `dugum = dugum.solCocuk` : 2'ye inildi. (c) 1'in sol çocuğu yok; döngü bitti, en küçük 1. (d) `maxDeger` 6'dan 8'e iner ve 8'de durur: en büyük 8.

10.2.5 Kök ortada gezinme

Gezinme, ağacın bütün düğümlerini belli bir sırayla ziyaret etmektir. İkili ağaçta üç klasik özyinelemeli sıra vardır: kök önde (*preorder*), kök ortada (*inorder*) ve kök sonda (*postorder*). Adlar, kökün iki alt ağaca göre ne zaman ziyaret edildiğini söyler. Kök ortada gezinmede önce sol alt ağaç, sonra kök, en son sağ alt ağaç gezilir.

```
void inorder(IkiliAgacDugum<E> d) {
    if (d == null) return;
    inorder(d.solCocuk);  // sol
    System.out.print(d.veri + " ");
    inorder(d.sagCocuk);  // sağ
}
```

Metodun gövdesi taban durumdan sonra üç satırdır ve her satır sıranın bir parçasıdır. Boş alt ağaçta (`d == null`) hiçbir şey yapmadan döner; bu, özyinelemenin taban durumudur. Boş değilse önce sol alt ağacı bütünüyle gezer, sonra kendi verisini yazar, en son sağ alt ağacı gezer.

BST'de bu sıra çok değerli bir sonuç verir: anahtarlar küçükten büyüğe sıralı yazılır. Nedenini bir düğüm için düşünelim. Düğüm yazılmadan önce sol alt ağacının tamamı yazılmıştır ve oradaki her anahtar düğümden küçüktür. Düğümden sonra sağ alt ağaç yazılır ve oradaki her anahtar düğümden büyüktür. Her düğüm, kendinden küçüklerin hepsinden sonra ve kendinden büyüklerin hepsinden önce yazılır. Bu da tam olarak sıralı çıktının tanımıdır.

> **Teorem 10.1 (Kök ortada gezinme sıralıdır)**
>
> Bir ikili arama ağacının kök ortada gezinmesi, ağaçtaki anahtarları artan sırayla verir.

İspat. Ağacın yüksekliği üzerinden tümevarım yapalım. Boş ağaç için çıktı boştur ve iddia doğrudur. Yüksekliği h olan bir ağacın kökü r olsun. Sol ve sağ alt ağaçların yüksekliği h'den küçüktür; tümevarım varsayımıyla ikisinin gezinmesi de kendi anahtarlarını artan sırayla verir. Çıktı "sol alt ağacın sıralı listesi, r, sağ alt ağacın sıralı listesi" biçimindedir. BST özelliğine göre sol listedeki her anahtar r'den küçük, sağ listedeki her anahtar r'den büyüktür. Üç parça art arda konunca liste bütünüyle artan sırada olur. □

Teoremin tersi de doğrudur: kök ortada gezinmesi kesin artan sıra veren ikili ağaç bir BST'dir. Bu yüzden bir ağacın BST olup olmadığını denetlemenin bir yolu, kök ortada gezinip çıktının artan olduğuna bakmaktır.

Gezinme her düğümü bir kez yazar. Her düğüm için iki özyinelemeli çağrı yapılır; bunların toplam sayısı da $O(n)$'dir (n dolu düğüm için $n + 1$ boş çağrı vardır). Süre bu yüzden $O(n)$'dir. Bellek açısından çağrı yığınında en çok $h + 2$ çerçeve (boş çağrı dahil) birikir: $O(h)$.

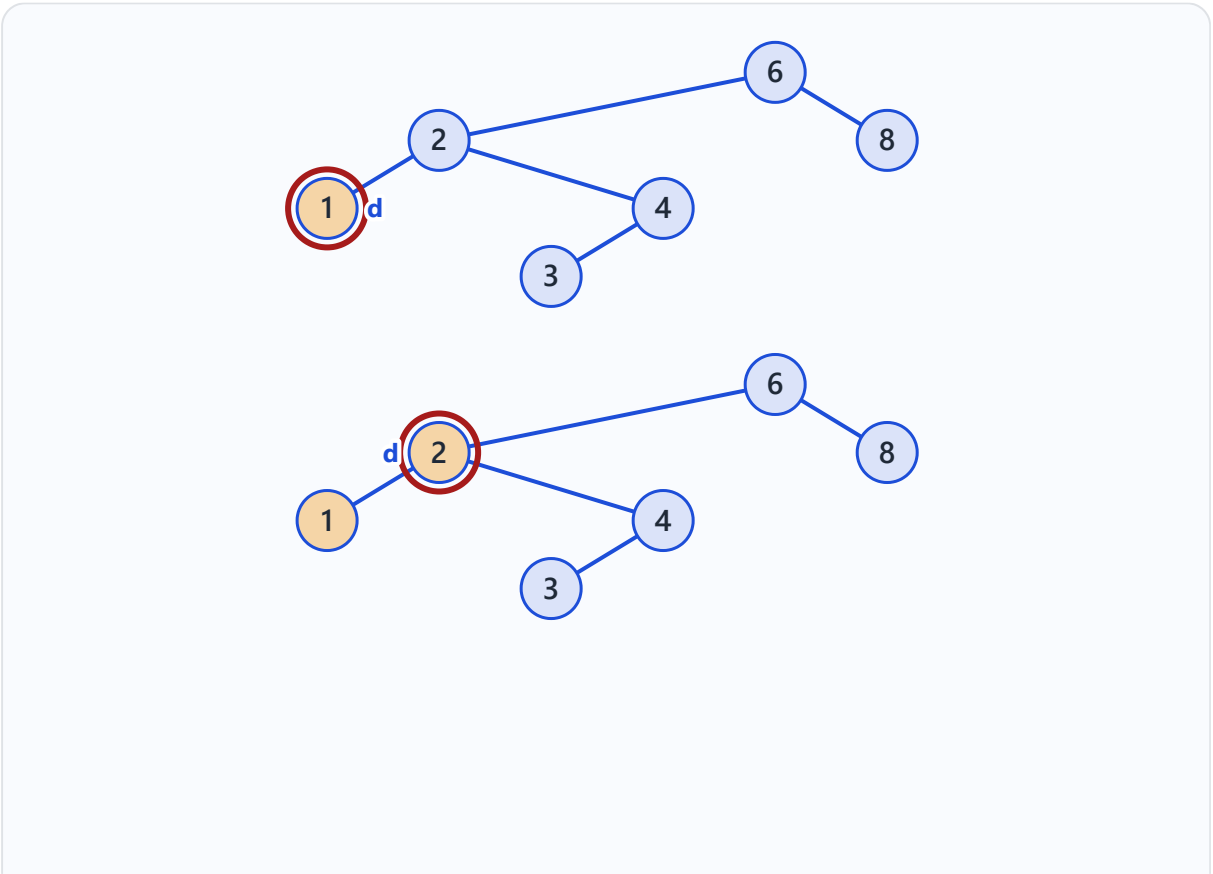
10.2.6 Kod izleme: inorder

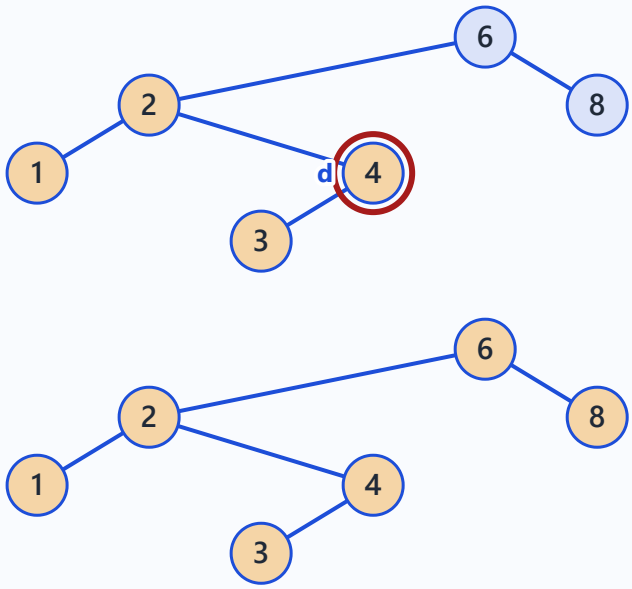
Aynı altı düğümlü ağaçta `inorder(kok)` çağrıldığında olaylar Tablo 10.3'teki sırayla gelişir. "Bekleyenler" sütunu, o anda çağrı yığınında duran ve sol alt ağaçlarının bitmesini bekleyen düğümleri alttan üste sıralar.

Adım	Olay	Bekleyenler	Çıktı
1	`inorder(6)` → önce sol: `inorder(2)`	6	
2	`inorder(2)` → önce sol: `inorder(1)`	6, 2	
3	`inorder(1)` : sol `null` , hemen döner; 1 yazılır	6, 2	1
4	1'in sağı `null` ; çerçeve kapanır; 2 yazılır	6	1 2
5	2'nin sağı: `inorder(4)` → önce sol: `inorder(3)`	6, 4	1 2
6	`inorder(3)` : solu ve sağı `null` ; 3 yazılır	6, 4	1 2 3
7	4 yazılır; 4'ün sağı `null` ; `inorder(4)` ve `inorder(2)` kapanır	6	1 2 3 4
8	6 yazılır; sağ alt ağaç: `inorder(8)`	—	1 2 3 4 6
9	`inorder(8)` : solu `null` ; 8 yazılır; sağı `null`	—	1 2 3 4 6 8

Tablo 10.3: `inorder(kok)` izlemesi (ağaç: 6(2(1, 4(3, —)), 8)).

Çıktı 1 2 3 4 6 8'dir ve sıralıdır. İzlemede her düğümün "bekleme" süresi dikkat çekicidir. 6 ilk çağrılan düğümdür, ama beşinci yazılır; çünkü kendisinden küçük dört anahtarın hepsi sol alt ağacındadır. 8 en son çağrılır ve en son yazılır. Şekil 10.7 izlemenin dört anını gösterir.





Şekil 10.7: `inorder` izlemesinden dört an; yazdırılmış düğümler turuncudur. (a) Sol taraf bitti, 1 yazıldı. (b) 2 yazıldı. (c) 4 yazıldı: 1, 2, 3, 4 sırayla çıktıda. (d) Bütün çağrılar bitti; çıktı 1 2 3 4 6 8. Her düğüm bir kez yazıldı: *O(n)*.

Sık yapılan hata

Kök ortada gezinmenin sıralı çıktı vermesini her ikili ağaçtan beklemek. Bu özellik yalnız BST'de geçerlidir. Sıralama kuralı olmayan bir ikili ağaçta kök ortada gezinme anahtarları ağacın şekline göre karışık bir sırada verir. Tersine, BST'de kök önde ya da kök sonda gezinme de sıralı çıktı vermez; sıralı çıktıyı yalnız kök ortada sıra verir.

10.3 BST işlemleri

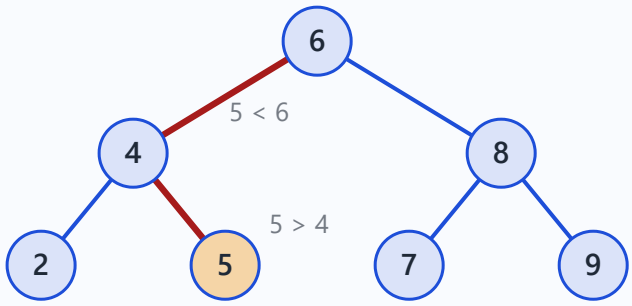
BST'nin bütün güncelleme ve sorgu işlemleri aynı iskelete dayanır: kökten başla, aranan anahtarı düğümün anahtarıyla karşılaştır, küçükse sola, büyükse sağa in. İşlemleri birbirinden ayıran, bu inişin nerede durduğu ve durduktan sonra ne yapıldığıdır. Arama, eşit anahtarı bulunca ya da boş bir bağa düşünce durur. Ekleme boş bağa düşünce oraya yeni bir düğüm bağlar. Silme, bulduğu düğümü çocuk sayısına göre üç farklı biçimde ağaçtan çıkarır.

10.3.1 Arama

Arama algoritması üç kuraldan oluşur:

1. Kökten başla. Düğümün anahtarı aranan anahtara eşitse arama başarıyla biter.
2. Aranan anahtar küçükse sol alt ağaçta, büyükse sağ alt ağaçta aynı işi tekrarla.
3. `null` bir bağa ulaşılırsa anahtar ağaçta yoktur.

Üçüncü kuralın doğruluğu BST özelliğinden gelir. Aranan anahtar ağaçta olsaydı, her adımda seçilen alt ağaçta olmak zorundaydı; çünkü öbür alt ağaçtaki anahtarların hepsi ya ondan küçük ya da ondan büyüktür. Seçilen alt ağaç boş çıktıysa anahtarın bulunabileceği başka yer kalmamıştır.



Şekil 10.8: 6, 4, 8, 2, 5, 7, 9 anahtarlı ağaçta 5 aranıyor. 5 < 6 olduğu için sola, 5 > 4 olduğu için sağa inilir ve 5 bulunur.

Sunumdaki kod jenerik düğüm sınıfıyla yazılmıştır ve özyinelemelidir:

```
boolean ara(IkiliAgacDugum<E> kok, E veri) {
    if (kok == null)
        return false;            // yok
    int k = veri.compareTo(kok.veri);
    if (k == 0)
        return true;             // bulundu
    else if (k < 0)
        return ara(kok.solCocuk, veri);
    else
        return ara(kok.sagCocuk, veri);
}
```

İlk `if` taban durumudur: boş alt ağaçta hiçbir anahtar yoktur, sonuç `false`'tur. Sonra aranan veri düğümün verisiyle bir kez karşılaştırılır ve sonuç `k`'de saklanır. `k == 0` ise anahtar bulunmuştur. `k < 0` ise aranan daha küçüktür ve arama sol alt ağaçta, `k > 0` ise sağ alt ağaçta sürer. Özyinelemeli çağrının döndürdüğü değer, hiç değiştirilmeden yukarı aktarılır.

Karşılaştırmanın bir kez yapılıp sonucun bir değişkende tutulması hem okunabilirliği hem de verimi artırır. `compareTo` bir `String` ya da karmaşık bir nesne için pahalı olabilir; iki ayrı `if`'te iki kez çağırmak gereksizdir.

Sık yapılan hata

Nesne anahtarları `==` ile karşılaştırmak. `==`, iki referansın *aynı nesneyi* gösterip göstermediğine bakar; değerlerin eşitliğine bakmaz. `Integer` anahtarlı bir ağaçta düğümdeki 1000 ile aranan 1000, ayrı ayrı kutulanmış (*boxing*) iki nesnedir. `veri == kok.veri` yanlış çıkar ve ağaçta olan anahtar bulunamaz. Küçük değerlerde (−128 ile 127 arası) Java hazır nesneleri yeniden kullandığı için hata gizli kalır ve yalnız büyük sayılarda ortaya çıkar. Sıralama için `compareTo`, eşitlik için `compareTo(...) == 0` ya da `equals` kullanılır.

Maliyet açısından her çağrı bir düzey aşağı iner ve sabit iş yapar. Başarılı arama, bulunan düğümün derinliğinin bir fazlası kadar düğüme bakar. Başarısız arama bir yaprağın boş çocuğuna kadar iner ve en çok *h + 1* düğüme bakar. İkisi de *O(h)*'dir. Özyinelemeli sürüm çağrı yığınında *O(h)* bellek kullanır. Aynı algoritma bir `while` döngüsüyle de yazılabilir; o zaman ek bellek *O(1)*'e iner. Özyinelemeli çağrı metodun son işi olduğu için bu dönüşüm mekaniktir: `kok = kok.solCocuk` ya da `kok = kok.sagCocuk` atanıp döngü başa döner.

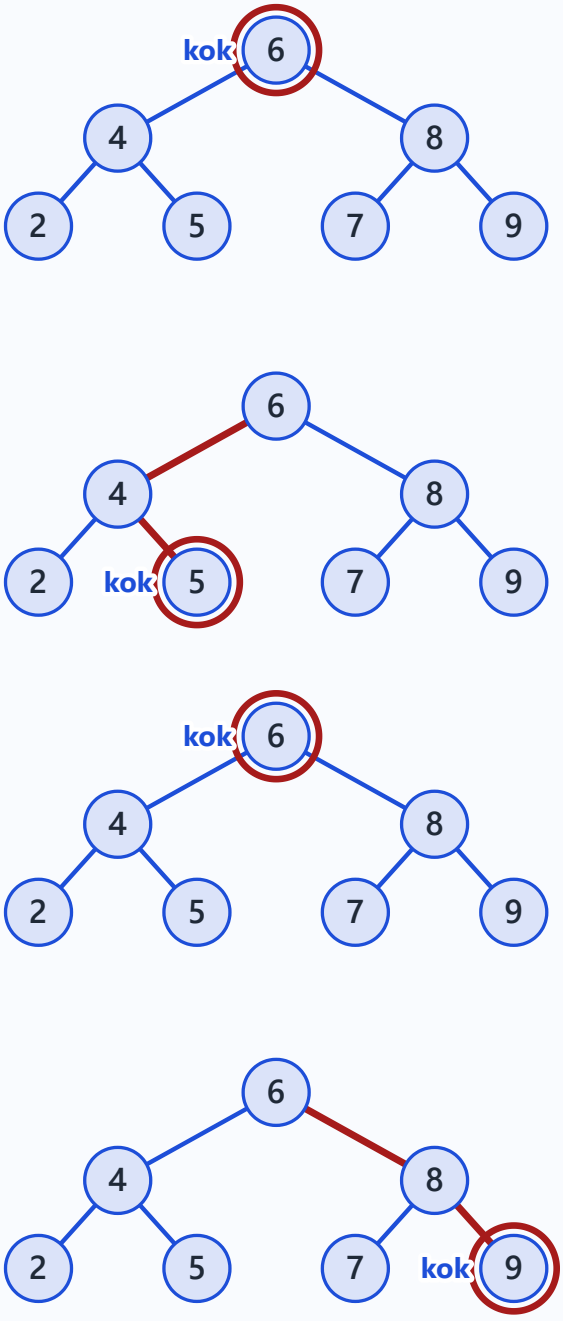
10.3.2 Kod izleme: ara

İki çağrıyı Şekil 10.8'deki ağaçta izleyelim: başarılı `ara(kok, 5)` ve başarısız `ara(kok, 10)`. Şekildeki ağaç 6(4(2, 5), 8(7, 9)) biçimindedir. Tablo satırlarındaki her çağrı, çağrı yığınında yeni bir çerçeve açar.

Çağrı	`kok`	`k`	Karar
`ara(kok, 5)`	6	−1 (5 < 6)	sol alt ağaç: `ara(4, 5)`
	4	1 (5 > 4)	sağ alt ağaç: `ara(5, 5)`
	5	0	bulundu, `return true`; 4 ve 6'daki çerçeveler `true`'yu aynen döndürür
`ara(kok, 10)`	6	1 (10 > 6)	sağ: `ara(8, 10)`
	8	1 (10 > 8)	sağ: `ara(9, 10)`
	9	1 (10 > 9)	sağ: `ara(null, 10)`
	`null`	—	9'un sağ çocuğu yok: `return false`; 9, 8 ve 6'daki çerçeveler `false`'u aynen döndürür

Tablo 10.4: `ara(kok, 5)` ve `ara(kok, 10)` izlemesi.

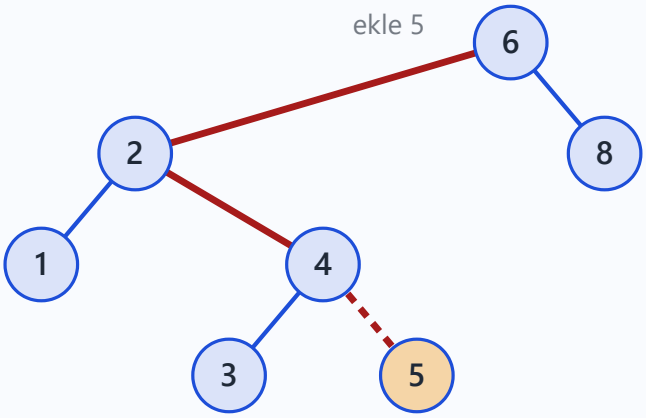
Başarılı aramada üç düğümle karşılaştırma yapıldı. Bu sayı, 5'in derinliği olan 2'nin bir fazlasıdır. Başarısız aramada da üç düğüme bakıldı (6, 8, 9) ve dördüncü çağrı boş bir bağa düştü. Ağacın yüksekliği 2 olduğu için en çok *h + 1 = 3* düğüme bakılabilirdi; iki arama da bu sınıra ulaştı. Yedi düğümlü ağaçta hiçbir arama üç karşılaştırmayı geçmez. Aynı yedi anahtar bir dizide sırasız dursaydı, başarısız bir arama yedi karşılaştırma isterdi.



Şekil 10.9: Arama izlemesinden dört an. (a) `ara(kok, 5)` : 5 < 6, `k = -1` . (b) Üçüncü çerçevede `kok` 5'i gösterir ve `k = 0` . (c) `ara(kok, 10)` : 10 > 6, sağa. (d) `kok == null` : 9'un sağ çocuğu yok, 10 ağaçta değil.

10.3.3 Ekleme

Yeni bir anahtar nereye eklenmelidir? Cevap aramanın kendisidir. Anahtar ağaçta aranıyormuş gibi kökten aşağı inilir. Anahtar ağaçta yoksa arama bir `null` bağda biter. Bu bağ, anahtarın BST özelliğini bozmadan durabileceği tek yerdir; çünkü yol boyunca her düğümde doğru tarafa dönülmüştür. Yeni düğüm oraya yaprak olarak bağlanır.



Şekil 10.10: 6, 2, 8, 1, 4, 3 anahtarlı ağaca 5 ekleniyor. 6'dan sola, 2'den sağa, 4'ten sağa inilir; 4'ün sağ bağı boş olduğu için 5 oraya yaprak olarak bağlanır (kesikli çizgi).

```
IkiliAgacDugum<E> ekle(
        IkiliAgacDugum<E> kok, E veri) {
    if (kok == null)
        return new IkiliAgacDugum<>(veri);
    int k = veri.compareTo(kok.veri);
    if (k < 0)
        kok.solCocuk = ekle(kok.solCocuk, veri);
    else if (k > 0)
        kok.sagCocuk = ekle(kok.sagCocuk, veri);
    // k == 0: anahtar zaten var, eklenmez
    return kok;
}
// kullanım: kok = ekle(kok, 5);
```

Kodun en önemli özelliği **dönüş değeridir**. `ekle`, kendisine verilen alt ağacın ekleme sonrasındaki kökünü döndürür. Boş bir alt ağaca ekleme yapılınca yeni düğüm oluşturulur ve döndürülür; artık o alt ağacın kökü bu yeni düğümdür. Boş olmayan alt ağaçta kök değişmez; metot çocuk bağını özyinelemeli çağrının sonucuyla günceller ve kendi kökünü döndürür.

`kok.solCocuk = ekle(kok.solCocuk, veri)` satırı ilk okumada gereksiz görünebilir: sol çocuk zaten oradadır, neden yeniden atanıyor? Gerçekten iş gören tek atama, inişin sonunda yapılanıdır. Yeni düğümün ebeveyni olacak düğümde `kok.sagCocuk` o ana kadar `null`'dır; çağrı yeni düğümü döndürür ve atama onu bağlar. Yol üzerindeki öteki atamalar bağı kendi değerine yeniden yazar ve hiçbir şeyi değiştirmez. Bu "gereksiz" atamalar sayesinde kod, yeni düğümün ebeveynini ayrıca hatırlamak zorunda kalmaz.

Aynı desen kullanımda da sürer: ağaca ekleme her zaman `kok = ekle(kok, 5);` biçiminde yazılır. Ağaç boşken (`kok == null`) ilk ekleme yeni bir kök oluşturur ve onu ancak bu atama `kok` değişkenine yazar.

Sık yapılan hata

Dönüş değerini atamadan `ekle(kok, 8);` yazmak. Java'da parametreler değerle geçer. Metot içinde `kok` parametresine ne atanırsa atansın, çağıranın `kok` değişkeni değişmez. Boş ağaçta `ekle(null, 8)` yeni düğümü döndürür, ama sonuç bir yere yazılmadığı için kaybolur; ağaç boş kalır. Boş olmayan ağaçta hata gizlenir, çünkü yeni yaprak iç çağrıda `kok.solCocuk = …` atamasıyla bağlanır. Kökün değiştiği her durumda (boş ağaç, kökün silinmesi, dengeli ağaçta kökte döndürme) ağaç bozulur.

Eşit anahtar geldiğinde (`k == 0`) iki `if` de atlanır ve metot hiçbir şeyi değiştirmeden `kok`'u döndürür; anahtar ikinci kez eklenmez. Eklemenin maliyeti aramayla aynıdır: kökten bir yaprağa inilir, her düzeyde sabit iş yapılır. Süre *O(h)*, özyinelemeli sürümde çağrı yığını *O(h)*'dir. Yeni düğüm *her zaman* bir yaprak olarak eklenir; var olan düğümlerin yeri değişmez. Bu basitlik, düz BST'nin şeklinin bütünüyle ekleme sırasına bağlı olmasının da nedenidir.

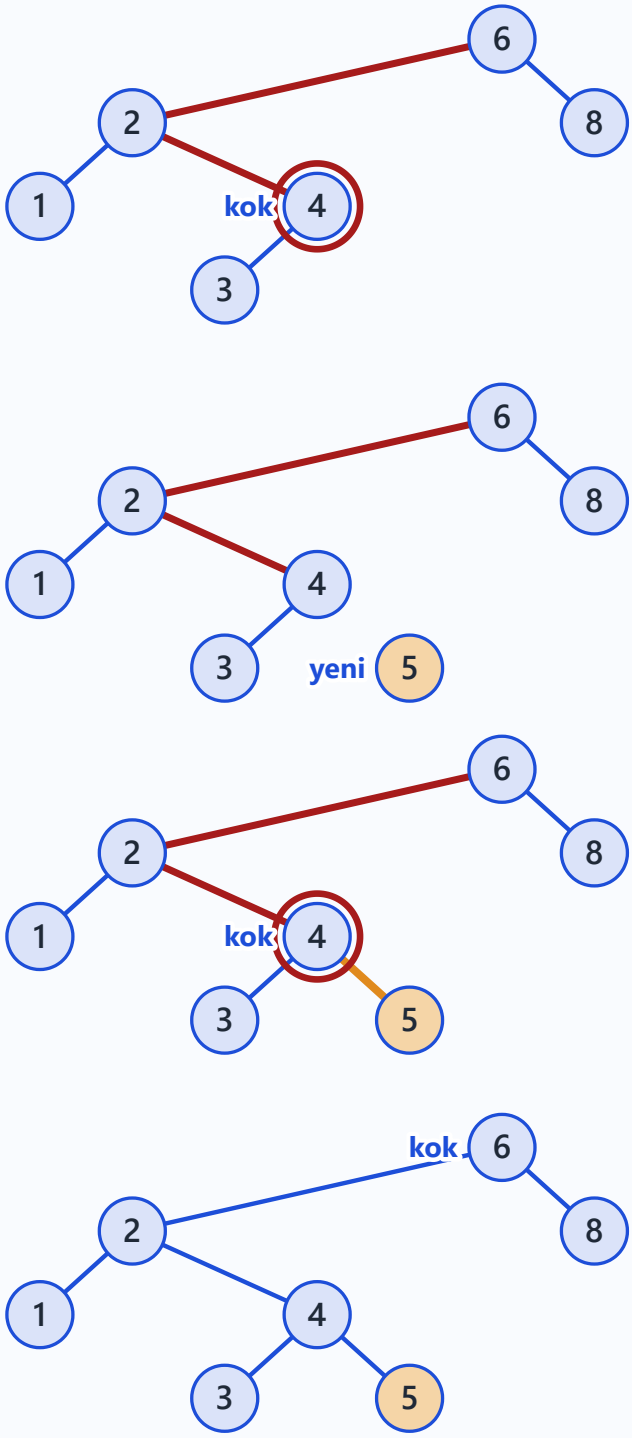
10.3.4 Kod izleme: ekle

Şekil 10.10'daki ağaca `kok = ekle(kok, 5)` çağrısıyla 5 eklenir. Tablo 10.5 hem inişi hem dönüşü gösterir. Dönüş satırlarında, her çerçevenin çocuk bağına hangi değeri atadığı yazılmıştır.

Adım	Çerçeve	k	Olay
1	`kok` = 6	−1	sola in: `ekle(kok.solCocuk, 5)`
2	`kok` = 2	1	sağa in: `ekle(kok.sagCocuk, 5)`
3	`kok` = 4	1	sağa in: `ekle(kok.sagCocuk, 5)`
4	`kok` = null	—	`new` ile düğüm 5 oluşturulur ve döndürülür; henüz hiçbir bağ onu göstermez
5	`kok` = 4	—	`kok.sagCocuk = düğüm(5)` : boş bağın yerine yeni düğüm bağlandı; `return` 4
6	`kok` = 2	—	`kok.sagCocuk = düğüm(4)` : bağ değişmez; `return` 2
7	`kok` = 6	—	`kok.solCocuk = düğüm(2)` : bağ değişmez; `return` 6
8	`main`	—	`kok = 6` : kök değişmedi

Tablo 10.5: `kok = ekle(kok, 5)` izlemesi (ağaç: 6(2(1, 4(3, —)), 8)).

Sonuç ağacı 6(2(1, 4(3, 5)), 8)'dir ve kök ortada gezinmesi 1 2 3 4 5 6 8 verir. Üç düzey inildi, üç çerçeve açıldı ve geri dönüldü. Gerçekte değişen tek bağ 4'ün sağ bağıdır. Şekil 10.11, yeni düğümün oluştuğu ve bağlandığı anları gösterir.



Şekil 10.11: Ekleme izlemesinden dört an. (a) 4'ten sağa inilirken `kok = null`. (b) Düğüm 5 oluşturuldu; hiçbir bağ onu göstermiyor. (c) 4'ün çerçevesinde `kok.sagCocuk` yeni düğüme bağlandı. (d) `main`'de `kok = 6`: kök değişmedi, yeni düğüm yaprak oldu.

10.3.5 Ardıl, öncül, tavan ve taban

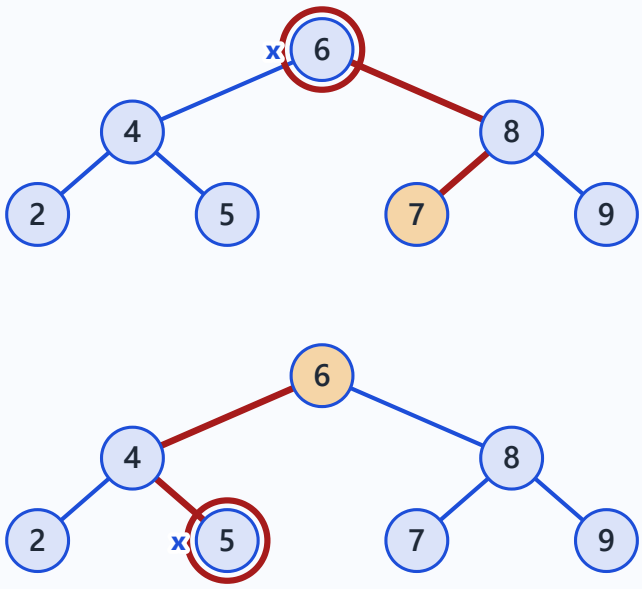
Sıralı bir listede her öğenin bir “sonraki” ve bir “önceki” komşusu vardır. BST'de bu komşulara ardıl ve öncül denir.

Tanım 10.3

Bir x anahtarının **ardılı** (*successor*), ağaçta x'ten büyük en küçük anahtardır. **Öncülü** (*predecessor*) ise x'ten küçük en büyük anahtardır. x'in kendisinin ağaçta bulunması gerekmez. Kök ortada sırada ardıl, x'ten hemen sonra; öncül hemen önce gelir.

x ağaçtaysa ardılını bulmak için iki durum vardır (Şekil 10.12):

1. **Sağ alt ağaç varsa**, ardıl sağ alt ağacın en küçüğüdür: bir kez sağa, sonra hep sola inilir. x'ten büyük ve ona en yakın anahtarlar onun sağ alt ağacındadır; oradaki en küçük olan da ardıldır.
2. **Sağ alt ağaç yoksa**, ardıl, kökten x'e inerken *solundan* geçilen en yakın atadır. Örneğin 5'e 6 → 4 → 5 yoluyla inilir. Bu yolda 6'dan sola dönülmüştür; yani 5, 6'nın sol alt ağacındadır ve 6'dan küçüktür. 4'ten ise sağa dönülmüştür; 4, 5'ten küçüktür ve ardıl olamaz. Ardıl 6'dır.



Şekil 10.12: Ardılın iki durumu, 6(4(2, 5), 8(7, 9)) ağacında. (a) 6'nın sağ alt ağacı var: bir sağa (8), sonra hep sola (7); `ardil(6) = 7`. (b) 5'in sağ alt ağacı yok: solundan geçilen en yakın ata 6; `ardil(5) = 6`.

Ardılın iki yüzlü bu tarifi koda geçirilirken birleştirilebilir. Kökten başlayıp aşağı inen tek bir döngü, yol boyunca "şimdiye kadarki en iyi aday"ı tutar:

```
AgacDugumu ardil(AgacDugumu kok, int x) {
    AgacDugumu aday = null; // x'ten büyük en küçük
    while (kok != null) {
        if (x < kok.veri) {
            aday = kok;     // kok bir aday
            kok = kok.sol;  // daha küçüğü solda
        } else
            kok = kok.sag;  // x'ten büyükler sağda
    }
    return aday;
}
```

Her düğümde iki olasılık vardır. *x < kok.veri* ise bu düğüm *x*'ten büyüktür ve bir ardıl adayıdır. Daha iyi bir aday, yani *x*'ten büyük ama bu düğümden küçük bir anahtar, ancak düğümün solunda olabilir; aday kaydedilir ve sola inilir. *x ≥ kok.veri* ise bu düğüm ve bütün sol alt ağacı *x*'ten küçük ya da ona eşittir; hiçbiri ardıl olamaz. *x*'ten büyükler ancak sağdadır; sağa inilir. Döngü boş bir bağda biter ve son aday döndürülür. Hiç aday bulunmadıysa sonuç `null`'dır; bu, *x*'in ağaçtaki en büyük anahtar ya da ondan büyük olduğu anlamına gelir.

Kod iki durumu kendiliğinden kapsar. Sağ alt ağaç varsa döngü *x*'i geçtikten sonra bir kez sağa, sonra adaylar toplayarak hep sola iner; son aday sağ alt ağacın en küçüğüdür. Sağ alt ağaç yoksa *x*'ten sonra döngü hemen biter ve son aday, solundan geçilen en yakın atadır. Üstelik kod *x*'in ağaçta bulunmasını da gerektirmez.

Ardılın akrabası üç sorgu daha vardır. Tablo 10.6 dördünü bir arada verir. Java'nın `TreeMap` ve `TreeSet` sınıfları bu sorguları hazır metotlar olarak sunar.

İşlem	Anlamı	Java `TreeMap`	Örnek
ardıl	*x*'ten büyük en küçük anahtar	`higherKey(x)`	`ardil(5) = 6`
öncül	*x*'ten küçük en büyük anahtar	`lowerKey(x)`	`oncul(6) = 5`
tavan (*ceiling*)	≥ *x* olan en küçük anahtar	`ceilingKey(x)`	`tavan(3) = 4`
taban (*floor*)	≤ *x* olan en büyük anahtar	`floorKey(x)`	`taban(3) = 2`

Tablo 10.6: Sıralı komşu sorguları; son sütun 6(4(2, 5), 8(7, 9)) ağacı içindir.

Öncül, ardılın aynadaki görüntüsüdür: *x > kok.veri* ise düğüm bir adaydır ve sağa inilir, değilse sola inilir. Tavan ile ardıl arasındaki tek fark eşitliktir: *x* ağaçta varsa tavan *x*'in kendisidir, ardıl ise ondan sonraki anahtardır. Taban da öncülün eşitliği kabul eden sürümüdür. Örnek ağaçta 3 yoktur; bu yüzden tavan(3) ardıl gibi 4'ü, taban(3) öncül gibi 2'yi verir. Dört sorgunun hepsi kökten tek bir yol izler ve *O(h)* sürer.

Örnek 10.2

Aynı ağaçta `taban(kok, 7)` ve `oncul(kok, 7)` bulunsun.

1. `taban` : 7 ≥ 6, 6 aday; sağa (8). 7 < 8, sola (7). 7 ≥ 7, 7 aday; sağa (`null`). Sonuç 7: anahtar ağaçta olduğu için taban kendisidir.
2. `oncul` : 7 > 6, 6 aday; sağa (8). 7 > 8 değil, sola (7). 7 > 7 değil, sola (`null`). Sonuç 6: 7'den küçük en büyük anahtar.

İki sorgu aynı yolu izledi; tek fark 7'nin kendisinin aday sayılıp sayılmamasıdır.

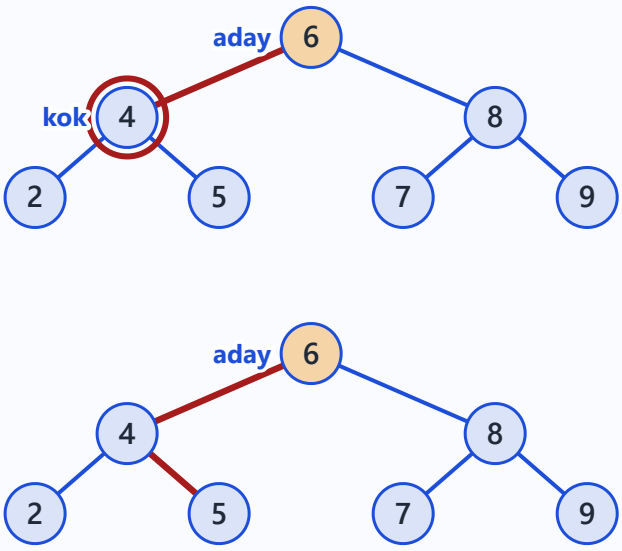
10.3.6 Kod izleme: ardil

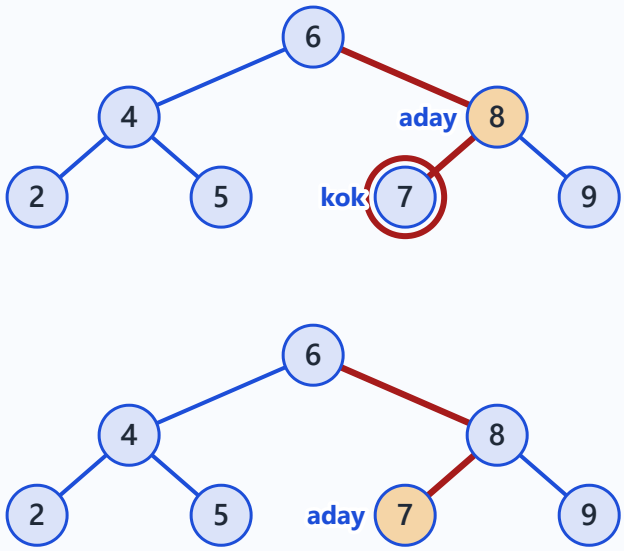
`ardil(kok, 5)` ve `ardil(kok, 6)` çağrıları, ardılın iki durumunu sırasıyla örnekler. Tablo 10.7'de her satır döngünün bir turudur.

Çağrı	kok	Karşılaştırma	aday	Yön
`ardil(kok, 5)`	6	5 < 6	6	sola
	4	5 ≥ 4	6	sağa
	5	5 ≥ 5	6	sağa → `null` ; `return` 6
`ardil(kok, 6)`	6	6 ≥ 6	null	sağa
	8	6 < 8	8	sola
	7	6 < 7	7	sola → `null` ; `return` 7

Tablo 10.7: `ardil(kok, 5)` ve `ardil(kok, 6)` izlemesi.

İlk çağrıda aday kökte bulundu ve bir daha değişmedi: 5'in sağ alt ağacı yoktur, ardıl solundan geçilen ata 6'dır. İkinci çağrıda aday önce 8, sonra 7 oldu: 6'nın sağ alt ağacı vardır ve ardıl onun en küçüğüdür. Her iki çağrıda da üç düğüme bakıldı. Şekil 10.13 izlemenin dört anını gösterir.





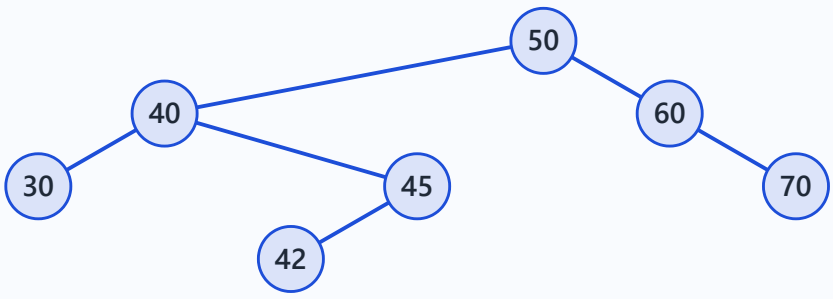
Şekil 10.13: `ardil` izlemesi. (a) 5 < 6: 6 aday, sola inilir. (b) Döngü bitti; `ardil(5) = 6`. (c) `ardil(kok, 6)`: 6 < 8, 8 aday. (d) 7 daha iyi aday; döngü bitti, `ardil(6) = 7`.

Sık yapılan hata

Ardılı "sağ çocuk" sanmak. 6'nın sağ çocuğu 8'dir, ama ardılı 7'dir; ardıl, sağ alt ağacın *en küçüğüdür* ve sağ çocuktan sonra sola inmeyi gerektirir. Öteki yanlış ise sağ alt ağaç yokken ardılın olmadığını düşünmektir. 5'in sağ çocuğu yoktur, ama ardılı 6 vardır; yukarıda, bir atadadır.

10.3.7 Silme: üç durum

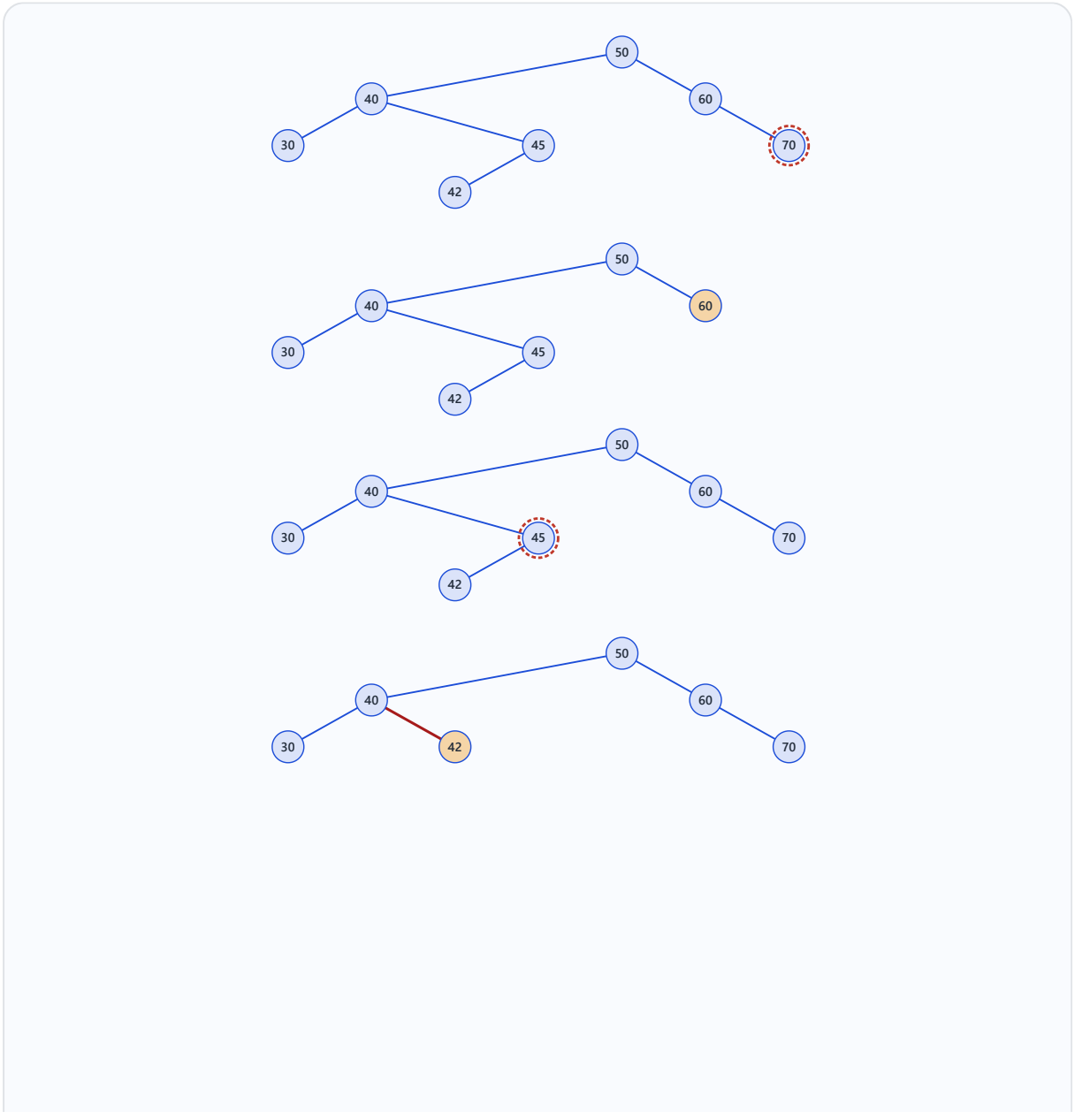
Silme, BST işlemlerinin en dikkat isteyenidir. Önce silinecek düğüm aramayla bulunur; bu *O(h)* sürer. Ardından düğüm, çocuk sayısına göre ağaçtan çıkarılır. Amaç, BST özelliğini bozmadan düğümü koparmaktır. Örneklerde Şekil 10.14'teki ağaç kullanılır: kök 50; 40'ın çocukları 30 ve 45; 45'in sol çocuğu 42; 60'ın sağ çocuğu 70.

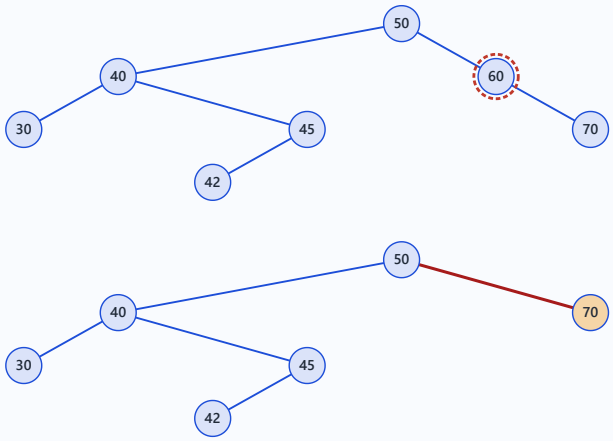


Şekil 10.14: Silme örneklerinin ağacı: 50(40(30, 45(42, —)), 60(—, 70)). 70 ve 30 yaprak, 45 ile 60 tek çocuklu, 40 ve 50 iki çocukludur.

1. **Yaprak.** Silinecek düğümün çocuğu yoktur. Ebeveynin ona giden bağı `null` yapılır. Örnek: 70'in silinmesi; 60'ın sağ bağı `null` olur.
2. **Tek çocuk.** Düğüm atlanır: ebeveyn, doğrudan düğümün tek çocuğuna bağlanır. Örnek: 45 silinirken 40'ın sağ bağı 42'ye yönelir; 60 silinirken 50'nin sağ bağı 70'e yönelir.
3. **İki çocuk.** Düğüm doğrudan çıkarılamaz, çünkü ebeveyninin onun yerine bağlayabileceği tek bir bağ vardır, oysa düğümün iki çocuğu vardır. Bunun yerine düğümün *değeri*, kök ortada ardılının (sağ alt ağacın en küçüğü) ya da öncülünün (sol alt ağacın en büyüğü) değeriyle değiştirilir. Sonra ardıl ya da öncül kendi yerinden silinir. Örnek: 40'ın silinmesi.

Tek çocuk durumunun neden BST özelliğini koruduğunu görmek kolaydır. 45, 40'ın sağ alt ağacındadır; 45'in soyundaki bütün düğümler de öyledir ve 40'tan büyüktür. 42'yi 40'ın sağına bağlamak bu ilişkiyi değiştirmez. Aynı biçimde 42, 50'nin sol alt ağacında kalır ve 50'den küçüktür. Alt ağaç olduğu gibi bir düzey yukarı çekilmiş, sırası değişmemiştir.

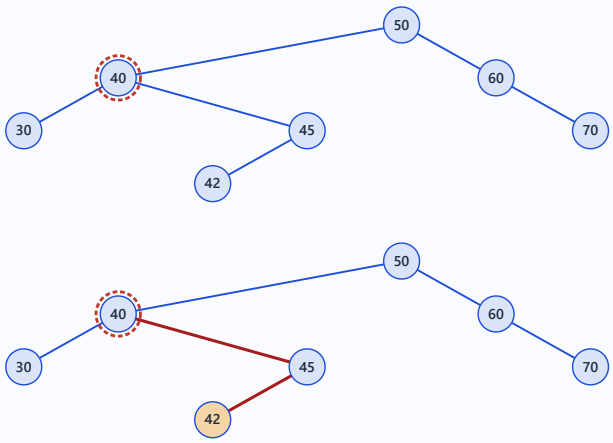


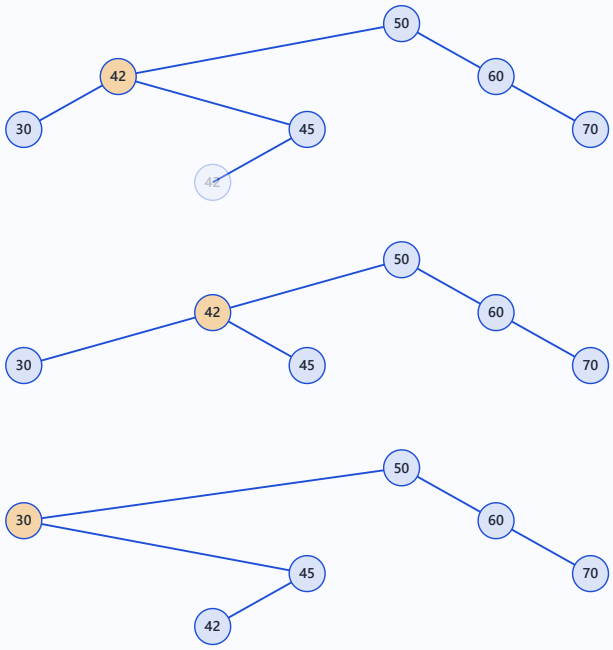


Şekil 10.15: Yaprak ve tek çocuk silme; her satır orijinal ağaçtan başlar. (a)–(b) Yaprak 70 silinir; 60'ın sağ bağı `null` olur. (c)–(d) 45'in yalnız sol çocuğu 42 vardır; 42, 45'in yerine 40'ın sağ çocuğu olur. (e)–(f) 60'ın yalnız sağ çocuğu 70 vardır; 70, 60'ın yerine 50'nin sağ çocuğu olur.

İki çocuk durumunda neden ardıl seçilir? Silinen değerin yerine konacak değer, sol alt ağacın hepsinden büyük ve sağ alt ağacın hepsinden küçük olmalıdır. Kök ortada sırada silinen anahtarın hemen yanındaki iki anahtar bu koşulu sağlar: öncül ve ardıl. 40'ın ardılı 42'dir. 42, 40'ın sağ alt ağacındaki en küçük anahtardır; yani sağ alt ağaçtaki öteki anahtarlardan küçüktür. 40'tan büyük olduğu için de sol alt ağaçtaki her anahtardan büyüktür. 42, 40'ın yerine konduğunda BST özelliği korunur.

Geriye ardılı eski yerinden silmek kalır. Bu silme her zaman kolay durumlardan birine düşer. Ardıl, sağ alt ağaçta hep sola inilerek bulunur ve sol çocuğu olmayan ilk düğümde durulur. Sol çocuğu yoktur; ya yapraktır ya da yalnız sağ çocuğu vardır. Yani ardılın silinmesi Durum 1 ya da Durum 2'dir; özyineleme bir daha iki çocuklu bir düğüme rastlamaz.





Şekil 10.16: İki çocuklu 40'ın silinmesi. (a) 40'ın hem solu (30) hem sağı (45) var. (b) Kök ortada ardıl: 45'e geç, sonra hep sola → 42. (c) 42'nin değeri 40'ın düğümüne kopyalanır; eski 42 fazladan duruyor. (d) Eski 42 sağ alt ağaçtan silinir; yaprak olduğu için Durum 1. (e) Öncül de kullanılabilir: sol alt ağacın en büyüğü 30; iki seçim de geçerli bir BST verir.

10.3.8 Silme kodu

Üç durum, eklemedeki "alt ağacın yeni kökünü döndür" deseniyle tek bir özyinelemeli metotta toplanır:

```
IkiliAgacDugum<E> sil(
        IkiliAgacDugum<E> kok, E veri) {
    if (kok == null) return null;   // yok
    int k = veri.compareTo(kok.veri);
    if (k < 0)
        kok.solCocuk = sil(kok.solCocuk, veri);
    else if (k > 0)
        kok.sagCocuk = sil(kok.sagCocuk, veri);
    else if (kok.solCocuk == null)
        return kok.sagCocuk;  // yaprak/sağ çocuk
    else if (kok.sagCocuk == null)
        return kok.solCocuk;  // yalnız sol çocuk
    else {                    // iki çocuk
        kok.veri = minDeger(kok.sagCocuk).veri;
        kok.sagCocuk = sil(kok.sagCocuk, kok.veri);
    }
    return kok;
}
```

Metodun ilk yarısı aramadır. Anahtar bulunamazsa (`kok == null`) `null` döner; boş alt ağaçtan silme yapılınca alt ağaç boş kalır ve ağaç değişmez. `k < 0` ya da `k > 0` ise silme ilgili alt ağaçta sürer ve sonuç çocuk bağına atanır.

`k == 0` olduğunda silinecek düğüm bulunmuştur ve üç durum üç dalda ele alınır:

- `kok.solCocuk == null` ise `kok.sagCocuk` döndürülür. Bu dal iki durumu birden kapsar. Düğüm yapraksa sağ çocuk da `null`'dır ve `null` döner; ebeveynin bağı boşalır. Düğümün yalnız sağ çocuğu varsa o çocuk döner; ebeveyn doğrudan ona bağlanır.
- `kok.sagCocuk == null` ise (sol çocuk var, sağ yok) sol çocuk döndürülür.
- İki çocuk varsa `minDeger(kok.sagCocuk)` ardılı bulur ve değeri düğüme kopyalanır. Sonra `sil(kok.sagCocuk, kok.veri)` ardılı sağ alt ağaçtan siler. Burada `kok.veri` artık ardılın değeridir.

Düğümün ağaçtan kopması, döndürülen değerin ebeveynin bağına atanmasıyla olur. Yaprak ya da tek çocuk durumunda metot silinen düğümü değil, onun yerine geçecek olanı döndürür. Ebeveyn bu değeri kendi bağına yazar ve silinen düğüme giden referans kaybolur. Java'nın çöp toplayıcısı (*garbage collector*) artık hiçbir referansın göstermediği düğümü bir süre sonra temizler. Kullanım her zaman `kok = sil(kok, 40);` biçimindedir; kök silinirse yeni kök ancak bu atamayla `kok` değişkenine yazılır.

Maliyeti bulalım. Silinecek düğüme kadar iniş *O(h)*'dir. İki çocuk durumunda `minDeger` aynı yoldan aşağı devam eder. İkinci `sil` çağrısı da ardıla kadar yeniden iner. Bu iki iniş silinen düğümün altında kalır ve toplamları *O(h)*'yi geçmez. Silme *O(h)* zamanda yapılır.

> **Sık yapılan hata**
>
> İki çocuklu düğümü silerken ardılın değerini kopyalayıp ardılı yerinden silmeyi unutmak. O zaman 42 ağaçta iki kez bulunur ve ağaç BST olmaktan çıkar. Öteki yanlış da ardılı silerken anahtar olarak eski değeri kullanmaktır: `sil(kok.sagCocuk, veri)` yazılırsa sağ alt ağaçta 40 aranır ve bulunamaz. Doğru çağrı, kopyalanmış yeni değerle yapılır: `sil(kok.sagCocuk, kok.veri)`.

10.3.9 Kod izleme: üç silme

Üç silme de Şekil 10.14'teki orijinal ağaçtan başlar. Tablo 10.8 her çerçevenin karşılaştırmasını ve dönüşte bağına atadığı değeri verir.

Çağrı	İniş (k)	Bulunan düğümde	Değişen bağ	Sonuç ağacı
`sil(kok, 70)`	50 (1), 60 (1), 70 (0)	sol çocuk yok → `return kok.sagCocuk` = `null`	60'ın sağ bağı `null`	50(40(30, 45(42, —)), 60)

Çağrı	İniş (k)	Bulunan düğümde	Değişen bağ	Sonuç ağacı
`sil(kok, 45)`	50 (−1), 40 (1), 45 (0)	sol var, sağ yok → `return kok.solCocuk` = 42	40'ın sağ bağı 42	50(40(30, 42), 60(—, 70))
`sil(kok, 40)`	50 (−1), 40 (0)	iki çocuk: ardıl 42; `kok.veri = 42`; `sil(45'li alt ağaç, 42)` : 45 (−1), 42 (0), yaprak → `null`	45'in sol bağı `null`; 40'ın düğümünde artık 42	50(42(30, 45), 60(—, 70))

Tablo 10.8: `sil(kok, 70)`, `sil(kok, 45)` ve `sil(kok, 40)` izlemesi.

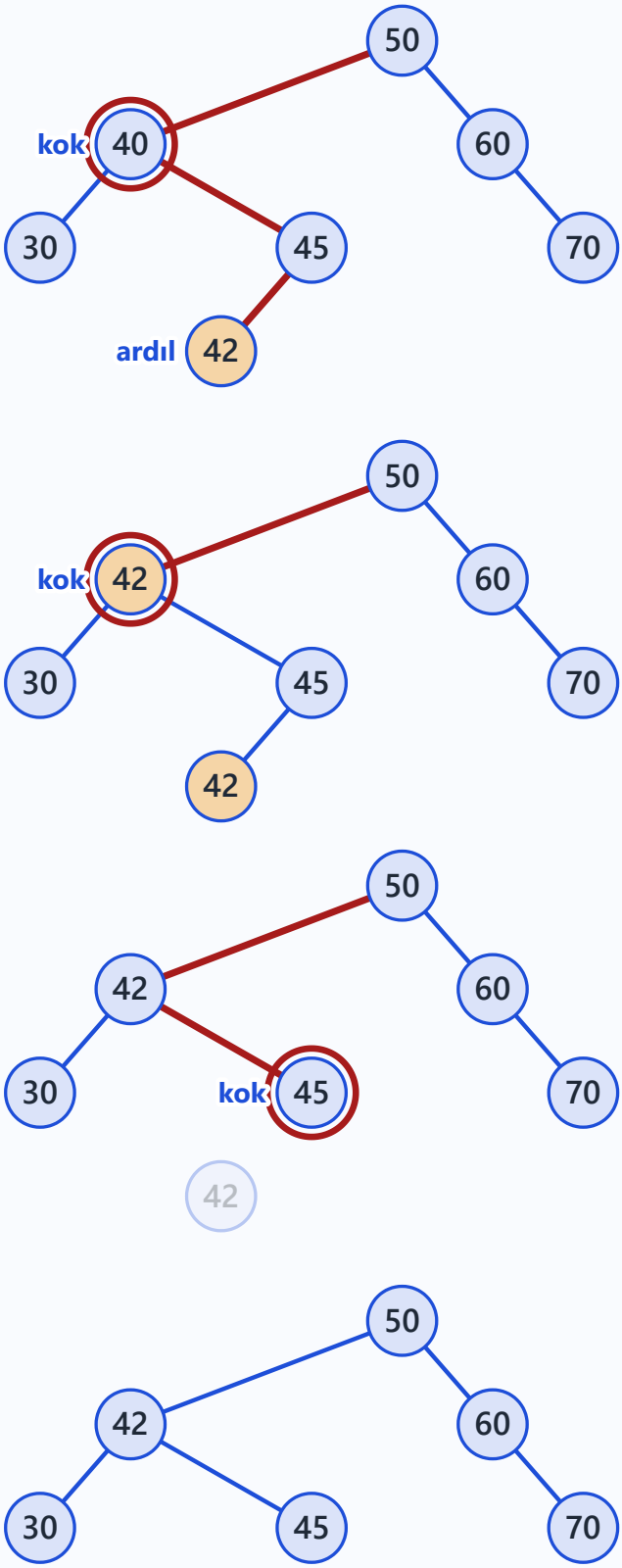
Üç sonucun kök ortada gezinmeleri sırasıyla 30 40 42 45 50 60, 30 40 42 50 60 70 ve 30 42 45 50 60 70'tir. Her birinde silinen anahtar kaybolmuş, kalanların sırası korunmuştur. Üç izlemede kök hiç değişmedi; `main`'deki `kok = sil(…)` ataması her seferinde yine 50'yi yazdı.

Üçüncü izleme ayrıntılı incelenmeye değer, çünkü iki ayrı silme iç içe çalışır. Tablo 10.9 bu çağrıyı adım adım verir.

Adım	Çerçeve	Olay
1	`sil(50, 40)`	k = −1 (40 < 50): `kok.solCocuk = sil(kok.solCocuk, 40)`
2	`sil(40, 40)`	k = 0; iki çocuk var (30 ve 45)
3	`sil(40, 40)`	`minDeger(45)` : 45'ten sola 42'ye; 42'nin solu yok → ardıl 42
4	`sil(40, 40)`	`kok.veri = 42` : düğümde artık 42; ağaçta iki tane 42 var
5	`sil(45, 42)`	k = −1 (42 < 45): `kok.solCocuk = sil(kok.solCocuk, 42)`
6	`sil(42, 42)`	k = 0; sol çocuk yok → `return kok.sagCocuk` = `null`
7	`sil(45, 42)`	`kok.solCocuk = null` : eski 42 koptu; `return` 45
8	`sil(42, 40)`	`kok.sagCocuk = düğüm(45)` (bağ aynı); `return` düğüm (değeri 42)
9	`sil(50, 40)`	`kok.solCocuk` = aynı düğüm; `return` 50; `main` : `kok = 50`

Tablo 10.9: `kok = sil(kok, 40)` izlemesinin ayrıntısı.

Dikkat edilirse 40'ı tutan *düğüm nesnesi* ağaçtan hiç çıkmadı; yalnız içindeki değer 42 oldu. Ağaçtan kopan nesne, eski 42'yi tutan yapraktır. Şekil 10.17 izlemenin üç anını ve sonucu gösterir.



Şekil 10.17: `sil(kok, 40)` izlemesi. (a) `minDeger(kok.sagCocuk)` : ardıl 42. (b) `kok.veri = 42` : 42 iki kez var. (c) Eski 42'nin bağı `null` oldu. (d) Sonuç: 50(42(30, 45), 60(—, 70)); düğüm sayısı bir azaldı, BST özelliği

korundu.

Örnek 10.3

Aynı orijinal ağaçtan bu kez kök 50 silinsin.

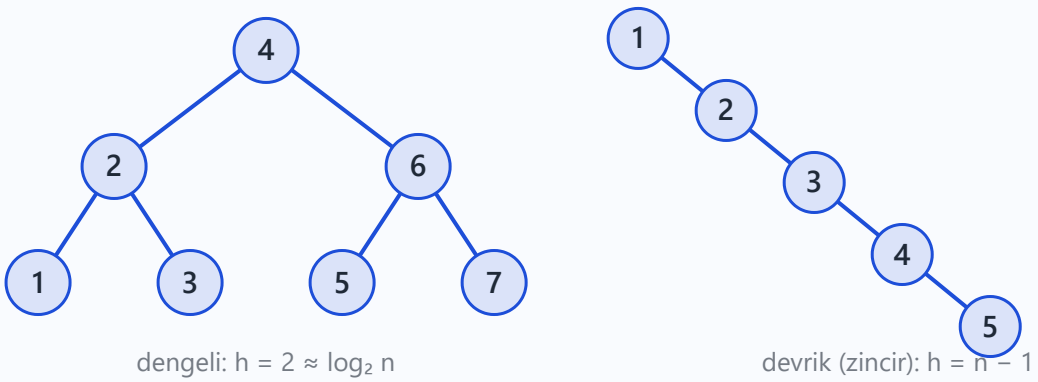
1. `sil(50, 50)` : k = 0; iki çocuk var (40 ve 60).
2. `minDeger(60)` : 60'ın sol çocuğu yok; ardıl 60'ın kendisidir.
3. `kok.veri = 60` ; sonra `sil(60'lı alt ağaç, 60)` : k = 0, sol çocuk yok → `return kok.sagCocuk` = 70. Bu bir tek çocuk silmesidir.
4. Kök düğümün sağ bağı 70'e yönelir. Sonuç: 60(40(30, 45(42, —)), 70). Kök ortada sıra: 30 40 42 45 60 70.

Ardıl, silinen düğümün doğrudan sağ çocuğu çıktı ve bir sağ çocuğu vardı. Bu durum, ardılın her zaman yaprak olmadığını gösterir; ama sol çocuğu hiçbir zaman yoktur.

10.3.10 BST işlemlerinin analizi

Buraya kadar görülen işlemlerin hepsi aynı sonuca vardı: arama, ekleme ve silmenin maliyeti, işlemin eriştiği en derin düğümün derinliğiyle orantılıdır. En kötü durumda bu derinlik ağacın yüksekliğidir; maliyet *O(h)*'dir. Öyleyse asıl soru şudur: *n* anahtarlı bir BST'nin yüksekliği ne kadardır?

Cevap, anahtarların ekleme sırasına bağlıdır. Aynı yedi anahtar 4, 2, 6, 1, 3, 5, 7 sırasıyla eklenirse yüksekliği 2 olan mükemmel bir ağaç oluşur. 1, 2, 3, 4, 5, 6, 7 sırasıyla eklenirse her yeni anahtar bir öncekinin sağına düşer ve ağaç sağa uzanan bir zincire dönüşür; yükseklik 6 olur. Böyle bir ağaca **devrik** (*degenerate*) ağaç denir. Devrik ağaç, fazladan bellek harcayan bir bağlı listeden farksızdır.



Şekil 10.18: Solda 7 düğümlü dengeli ağaç, yükseklik 2 (yaklaşık $log_2 n$). Sağda 1, 2, 3, 4, 5 sırayla eklenince oluşan zincir; yükseklik $n - 1 = 4$.

Durum	Yükseklik h	Arama / ekleme / silme	$n = 10^6$
Dengeli (en iyi)	$\approx \log_2 n$	$O(\log n)$	≈ 20
Rastgele sıra (ortalama)	$\approx 2{,}99 \log_2 n$	$O(\log n)$	≈ 28 (ortalama)
Devrik (en kötü, sıralı girdi)	$n - 1$	$O(n)$	10^6

Tablo 10.10: BST'nin yüksekliği ve işlem maliyeti; son sütun $n = 10^6$ için yaklaşık adım sayısıdır.

Tablonun orta satırı umut vericidir. Anahtarlar rastgele bir sırayla eklenirse, bir düğümün ortalama derinliği yaklaşık $2 \ln n \approx 1{,}39 \log_2 n$'dir. Bir milyon anahtarda bu, ortalama 28 civarında karşılaştırma demektir; en iyi durumdan yalnız %40 kadar fazladır. Ağacın beklenen *yüksekliği* ise biraz daha büyüktür: yaklaşık $4{,}31 \ln n \approx 2{,}99 \log_2 n$, yani bir milyon anahtarda 60 civarı. Tablonun yükseklik sütunu bu değeri, son sütunu ise ortalama derinliği verir. Bu sonuç, hızlı sıralamanın ortalama karşılaştırma sayısıyla aynı hesaptan çıkar: her anahtar kök olarak seçildiğinde kalan anahtarları rastgele iki gruba ayırır.

Sorun, gerçek verinin çoğu zaman rastgele olmamasıdır. Bir dosyadan okunan kayıtlar zaten sıralı olabilir; öğrenci numaraları kayıt sırasıyla artar; zaman damgaları hep büyür. Böyle bir girdi düz BST'yi en kötü duruma iter ve her işlem $O(n)$ olur. n anahtarı sıralı biçimde eklemek de toplam $1 + 2 + \dots + (n - 1) = n(n - 1)/2$, yani $O(n^2)$ karşılaştırma ister. Bir milyon anahtar için bu yarım trilyon karşılaştırmadır.

> **Sık yapılan hata**
>
> "BST'de arama $O(\log n)$'dir" demek. Bu yalnız ağaç dengeliyse doğrudur. Düz BST yükseklik için hiçbir garanti vermez; doğru ifade "$O(h)$"dir ve en kötü durumda $h = n - 1$'dir. $O(\log n)$ garantisi ancak AVL ve kırmızı-siyah gibi dengeli ağaçlarda vardır.

10.3.11 Aralık sorgusu

Aralık sorgusu, *[lo, hi]* aralığına düşen bütün anahtarları sıralı olarak verir. "5 ile 10 arasındaki anahtarlar" ya da "not ortalaması 2,5 ile 3,0 arasındaki öğrenciler" bu tür sorgulardır. Basit bir çözüm, bütün ağacı kök ortada gezip aralıktakileri yazmaktır; ama bu, aralıkta kaç anahtar olursa olsun $O(n)$ sürer. BST özelliği çok daha iyisini yapmaya izin verir: aralığa giremeyecek alt ağaçlara hiç inmemek. Buna **budama** (*pruning*) denir.

```
void aralik(AgacDugumu k, int lo, int hi) {
    if (k == null) return;
    if (lo < k.veri)          // solda aday var
        aralik(k.sol, lo, hi);
    if (lo <= k.veri && k.veri <= hi)
        System.out.print(k.veri + " ");
    if (k.veri < hi)          // sağda aday var
        aralik(k.sag, lo, hi);
}
```

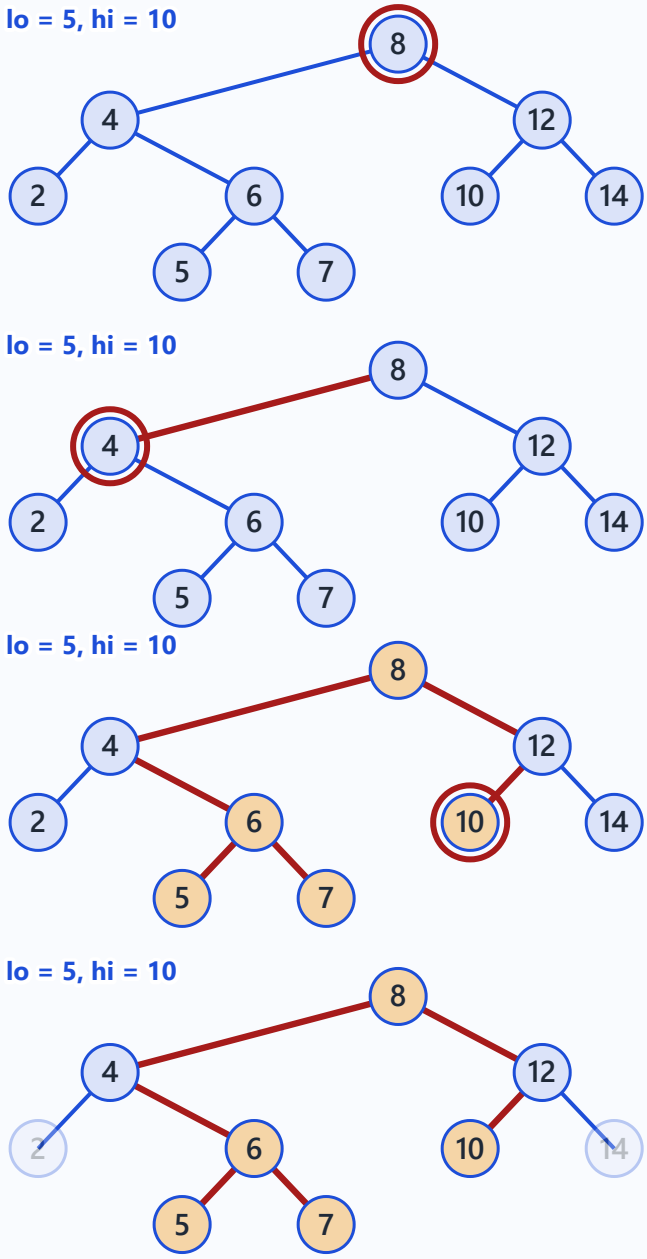
Kod, kök ortada gezinmenin iki koşulla budanmış hâlidir. Sol alt ağaçtaki anahtarların hepsi `k.veri`'den küçüktür. `lo < k.veri` değilse, yani `k.veri ≤ lo` ise, sol alt ağaçtaki her anahtar `lo`'dan küçüktür ve hiçbiri aralığa giremez; sol alt ağaç atlanır. Aynı biçimde `k.veri ≥ hi` ise sağ alt ağaçtaki her anahtar `hi`'den büyüktür; sağ alt ağaç atlanır. Ortadaki `if` düğümün kendisini aralıktaysa yazar. Sıra sol–düğüm–sağ olduğu için çıktı sıralıdır.

Tablo 10.11, 8(4(2, 6(5, 7)), 12(10, 14)) ağacında `aralik(kok, 5, 10)` çağrısını izler.

Düğüm	Sol koşul `5 < k.veri`	Aralıkta mı?	Sağ koşul `k.veri < 10`	Çıktı
8	doğru → sola in	kendisi ve sağı, sol alt ağaç bitince ele alınır		
4	yanlış → 2 budanır	hayır	doğru → sağa in	
6	doğru → sola in	kendisi ve sağı, sol alt ağaç bitince ele alınır		
5	yanlış → solu atlanır	evet	doğru → sağ `null`	5
6	sola dönüldü; 6 yazılır; `aralik(7)`			5 6
7	doğru → sol `null`	evet	doğru → sağ `null`	5 6 7
8	sola dönüldü; 8 yazılır; `aralik(12)`			5 6 7 8
12	doğru → sola in	hayır	yanlış → 14 budanır	
10	doğru → sol `null`	evet	yanlış → sağ budanır	5 6 7 8 10

Tablo 10.11: `aralik(kok, 5, 10)` izlemesi.

Çıktı 5 6 7 8 10'dur ve sıralıdır. Ziyaret edilen düğümler 8, 4, 6, 5, 7, 12 ve 10'dur. 2 ve 14 hiç ziyaret edilmedi: 2, 4'ün solundaydı ve 4 zaten alt sınırdan küçüktü; 14, 12'nin sağındaydı ve 12 zaten üst sınırdan büyüktü. Şekil 10.19 izlemenin başını ve sonunu gösterir.



Şekil 10.19: `aralik(kok, 5, 10)` izlemesi. (a) 5 < 8: solda aralığa giren anahtar olabilir. (b) 5 < 4 yanlış: 4'ün sol alt ağacı budanır. (c) 12 aralık dışında ve 12 ≥ 10: sağ alt ağaç budanır. (d) Sonuç 5 6 7 8 10; 2 ve 14 hiç ziyaret edilmedi (soluk).

Maliyeti bulmak için ziyaret edilen düğümleri iki gruba ayıralım. Birinci grup aralıktaki düğümlerdir; s tane yazılır. İkinci grup aralığın dışında kalıp yine de ziyaret edilen düğümlerdir. Bunlar, aralığın sol ve sağ sınırına doğru inen iki yolun üzerindedir ve her yol en çok $h + 1$ düğüm içerir. Toplam maliyet $O(h + s)$ olur. Dengeli bir ağaçta bir milyon anahtar arasından 10 anahtarlık bir aralık, yalnızca birkaç düzine düğüm ziyaret edilerek bulunur.

10.3.12 Bir ağaç BST mi?

Bir ikili ağacın BST olup olmadığını denetlemek, BST özelliğinin "bütün alt ağaç için" geçerli olduğunu iyi anlamayı gerektirir. Her düğüm yalnız ebeveyniyle değil, bütün atalarıyla da uyumlu olmalıdır. Sol alt ağaca inerken bir üst sınır, sağ alt ağaca inerken bir alt sınır kazanılır. Bir düğüm, kökten ona inen yoldaki bütün bu sınırların içinde kalmalıdır.

```
boolean bstMi(AgacDugumu k, long min, long max) {
    if (k == null) return true;
    if (k.veri <= min || k.veri >= max)
        return false;          // aralık dışında
    return bstMi(k.sol, min, k.veri)
        && bstMi(k.sag, k.veri, max);
}
// ilk çağrı: bstMi(kok, Long.MIN_VALUE,
//                  Long.MAX_VALUE)
```

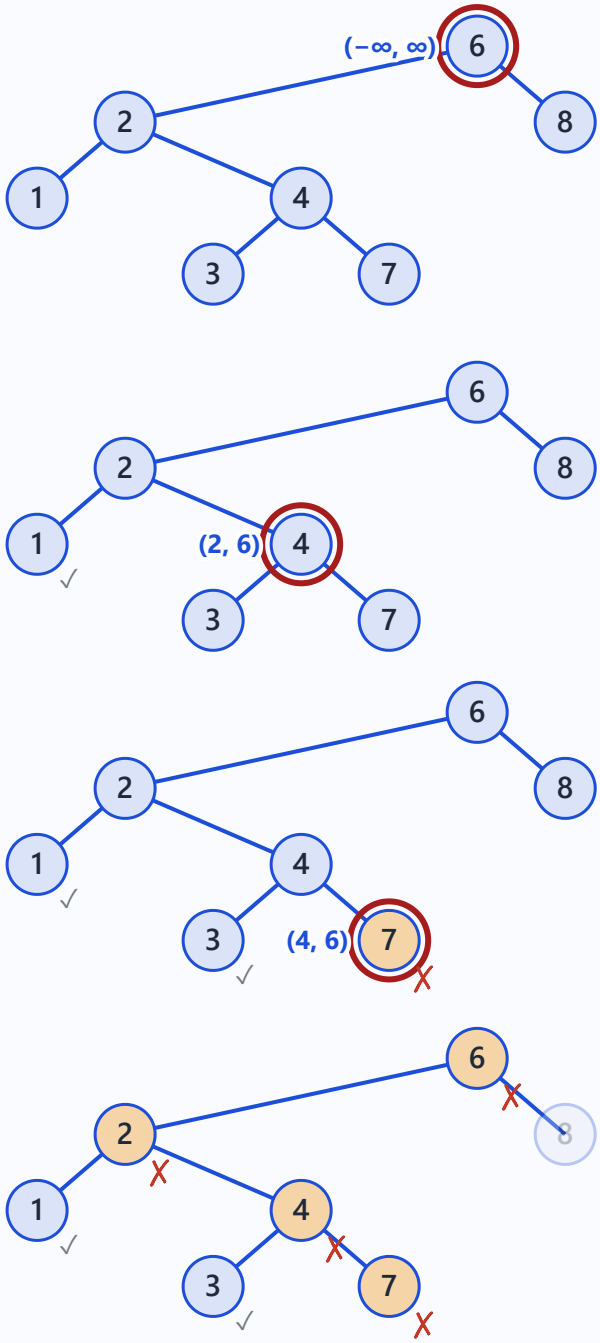
Metot her düğüme bir izinli aralık *(min, max)* taşır. Kök için aralık sınırsızdır; `Long.MIN_VALUE` ve `Long.MAX_VALUE` değerleri eksi ve artı sonsuzun yerine kullanılır. Anahtarlar `int` olduğu için `long` türündeki bu uç değerler hiçbir anahtara eşit olamaz. Düğüm aralığın dışındaysa `false` döner. İçindeyse sol çocuk için üst sınır düğümün kendi değeri olur, sağ çocuk için alt sınır. İki alt ağacın sonuçları `&&` ile birleşir.

Tablo 10.12, sağdaki hatalı ağacı, yani 6(2(1, 4(3, 7)), 8) ağacını denetler.

Çağrı	İzinli aralık	Denetim	Sonuç
`bstMi(6)`	(−∞, ∞)	6 aralıkta	sol: `bstMi(2, −∞, 6)`
`bstMi(2)`	(−∞, 6)	2 aralıkta	sol: `bstMi(1, −∞, 2)`
`bstMi(1)`	(−∞, 2)	1 aralıkta; çocukları `null`	`true`
`bstMi(4)`	(2, 6)	4 aralıkta	sol: `bstMi(3, 2, 4)`
`bstMi(3)`	(2, 4)	3 aralıkta; çocukları `null`	`true`
`bstMi(7)`	(4, 6)	7 ≥ 6	`false`
`bstMi(4)` = true && false; `bstMi(2)` = true && false			`false`
`bstMi(6)` : sol `false` ; `&&` kısa devre yapar, 8 hiç denetlenmez			`false`

Tablo 10.12: `bstMi(kok, Long.MIN_VALUE, Long.MAX_VALUE)` izlemesi.

7 için izinli aralık *(4, 6)*'dır. Alt sınır 4, 7'nin 4'ün sağında olmasından gelir. Üst sınır 6 ise 7'nin 6'nın sol alt ağacında olmasından gelir; bu sınır kökten iki düzey aşağı taşınmıştır. Yalnız ebeveyn–çocuk çiftlerine bakan bir denetim 4 < 7 olduğu için bu ağaca yanlışlıkla "BST" derdi. Doğru denetim her düğümü bir kez ziyaret eder; süre *O(n)*, çağrı yığını *O(h)*'dir.



Şekil 10.20: `bstMi` izlemesi. (a) Kökte aralık $(-\infty, \infty)$. (b) 4 için aralık $(2, 6)$. (c) 7 için aralık $(4, 6)$; $7 \geq 6$ olduğu için `false`. (d) Sonuç: ağaç BST değil.

Sık yapılan hata

Sınırları `int` türünde `Integer.MIN_VALUE` ve `Integer.MAX_VALUE` ile başlatmak. Ağaçta tam bu değerlerden biri varsa, katı eşitsizlik yüzünden geçerli bir ağaç "BST değil" diye reddedilir. Sunumdaki kod bunu `long` sınırlarla önler. Öbür yol, sınırları `Integer` nesnesi olarak taşıyıp "sınır yok" durumunu `null` ile göstermektir.

10.3.13 Uygulama: öğrenci kayıtları

Gerçek uygulamalarda düğüm yalnız bir sayı değil, bütün bir kayıt taşır. Sunumdaki örnekte anahtar öğrenci numarasıdır ve düğüm öğrenci nesnesinin tamamını tutar:

```
public class Ogrenci {
    int ogrenciNo;     // anahtar
    String adSoyad;

    public Ogrenci(int ogrenciNo,
                   String adSoyad) {
        this.ogrenciNo = ogrenciNo;
        this.adSoyad = adSoyad;
    }
}
```

```
public class OgrenciDugum {
    Ogrenci ogr;           // veri
    OgrenciDugum solCocuk;
    OgrenciDugum sagCocuk;

    public OgrenciDugum(Ogrenci ogr) {
        this.ogr = ogr;
        // çocuklar başlangıçta null
    }
}
```

`Ogrenci` sınıfı veriyi, `OgrenciDugum` ise ağaç yapısını taşır. Bu ayrım, aynı öğrenci nesnesinin başka bir yapıda da (örneğin bir listede) kullanılabilmesini sağlar. Ekleme ve arama, tam sayı anahtarlı sürümlerle aynı mantıkla çalışır:

```
OgrenciDugum ekle(OgrenciDugum kok, Ogrenci ogr) {
    if (kok == null) return new OgrenciDugum(ogr);
    if (ogr.ogrenciNo < kok.ogr.ogrenciNo)
        kok.solCocuk = ekle(kok.solCocuk, ogr);   // soldan devam
    else if (ogr.ogrenciNo > kok.ogr.ogrenciNo)
        kok.sagCocuk = ekle(kok.sagCocuk, ogr);   // sağdan devam
    return kok;                                   // aynı numara eklenmez
}
```

```
Ogrenci ara(OgrenciDugum kok, int numara) {
    if (kok == null) return null;                // kayıt yok
    if (numara == kok.ogr.ogrenciNo) return kok.ogr; // bulundu
    if (numara < kok.ogr.ogrenciNo)
        return ara(kok.solCocuk, numara);        // soldan devam
    return ara(kok.sagCocuk, numara);            // sağdan devam
}
```

İki fark dikkat çeker. Birincisi, anahtar `int` olduğu için karşılaştırmalar `<`, `>` ve `==` ile doğrudan yapılır; `compareTo` gerekmez. İkincisi, `ara` `boolean` yerine öğrenci nesnesini döndürür. Bulunursa ad-soyad gibi bilgilere de erişilir; bulunamazsa `null` döner.

Örnek 10.4

Boş ağaca sırasıyla (2205, Ayşe Kaya), (1980, Mehmet Demir), (2410, Zeynep Ak) ve (2205, Tekrar) eklensin.

1. 2205 kök olur.
2. 1980 < 2205: kökün soluna eklenir.
3. 2410 > 2205: kökün sağına eklenir.
4. 2205 kökte zaten vardır; iki koşul da yanlıştır ve kayıt eklenmez. Ağaçta "Ayşe Kaya" kalır.

`ara(kok, 2410).adSoyad` "Zeynep Ak", `ara(kok, 2205).adSoyad` "Ayşe Kaya" verir. `ara(kok, 1000)` ise 2205'ten sola, 1980'den sola iner, boş bağa düşer ve `null` döndürür.

Aynı işi Java'da hazır bir sınıf görür: `TreeMap<Integer, Ogrenci>`. `TreeMap` anahtardan değere bir eşleme tutar ve içeride dengeli bir arama ağacı, kırmızı-siyah ağaç kullanır. `put`, `get` ve `remove` *O(log n)* sürer; `higherKey`, `floorKey` ve `subMap` gibi metotlar da ardıl, taban ve aralık sorgularını yapar. Elle yazılan BST'den farkı, sıralı girdiyle bile zincire dönüşmemesidir.

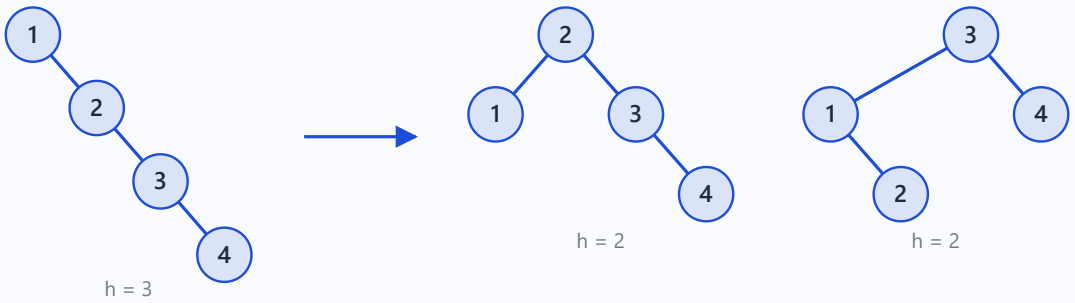
10.4 Denge ve AVL ağaçları

BST işlemlerinin hepsi *O(h)* sürer ve *h*, en kötü durumda *n − 1*'e çıkar. Bu kesimin sorusu, yüksekliği her koşulda *O(log n)*'de tutmanın ucuz bir yolunun olup olmadığıdır. Cevap evettir ve ilk yolu 1962'de Adelson-Velsky ile Landis gösterdi.

10.4.1 Denge ihtiyacı

Düz BST'nin şekli ekleme sırasına bağlıdır. 1, 2, 3, 4 sırayla eklenirse ağaç sağa uzanan bir zincir olur ve yüksekliği 3'tür. Oysa aynı dört anahtar, yüksekliği 2 olan ağaçlarda da durabilir. Şekil 10.21 iki seçeneği gösterir: kökü 2 olan 2(1, 3(—, 4)) ve kökü 3 olan 3(1(—, 2), 4). Üç ağacın da kök ortada gezin-

mesi 1 2 3 4'tür. Yani üçü de aynı sıralı kümeyi tutar; yalnız biçimleri farklıdır.



Şekil 10.21: 1, 2, 3, 4 zinciri (yükseklik 3), aynı anahtarlarla yüksekliği 2 olan iki farklı ağaca dönüştürülebilir: kökü 2 olan ve kökü 3 olan.

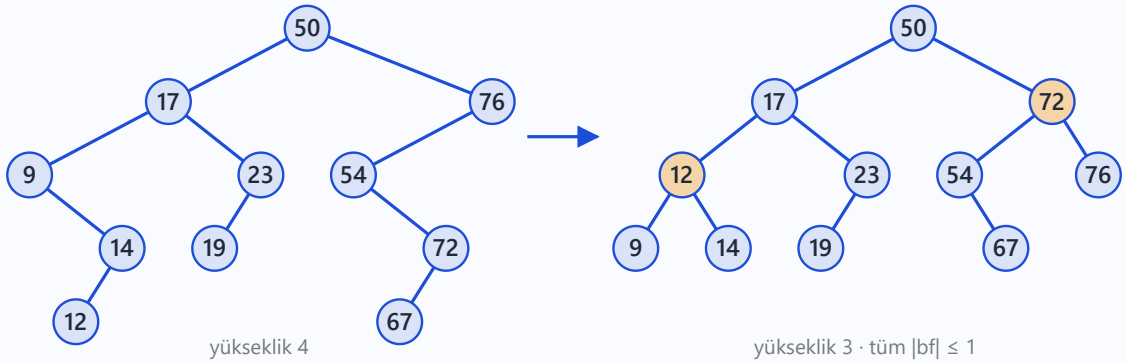
Tanım 10.4

Kendini dengeleyen arama ağacı (*self-balancing search tree*), yüksekliğin *O(log n)* kalmasını garanti eden BST'dir. Bunun için her ekleme ve silmeden sonra yapısını küçük yerel değişikliklerle düzeltir. Yerel değişiklikler döndürme ve renk değiştirmedir. Kısaca **dengeli ağaç** da denir.

Tanımdaki “yerel” sözcüğü önemlidir. Bozulmuş bir ağacı tamamen yıkıp kök ortada sırasından dengeli biçimde yeniden kurmak mümkündür; ama bu her güncellemede *O(n)* sürer ve ağacın bütün avantajını yok eder. Dengeli ağaçlar yalnız ekleme ya da silmenin geçtiği yol üzerindeki birkaç düğüme dokunur. Bu yol *O(log n)* uzunluktadır ve her düğümde yapılan iş sabittir; güncellemenin toplam maliyeti yine *O(log n)* kalır.

10.4.2 Dengeleme örneği

Daha büyük bir örnek, yerel düzeltmenin nasıl işlediğini gösterir. Şekil 10.22'nin solundaki ağaçta 11 anahtar vardır. 9 → 14 → 12 ve 76 → 54 → 72 → 67 kolları uzamıştır ve yükseklik 4'tür. Bu iki kolun alt ağaçları yeniden düzenlenince aynı 11 anahtar yüksekliği 3 olan bir ağaca sığar. Yeni alt ağaç kökleri 12 ve 72'dir.



Şekil 10.22: Solda yüksekliği 4 olan dengesiz ağaç; sağda aynı anahtarlarla yüksekliği 3 olan, her düğümde $|bf| \le 1$ sağlayan ağaç. Turuncu düğümler (12 ve 72) yeni alt ağaç kökleridir.

Yeniden düzenleme sırasında kök ortada sıra değişmez: iki ağaç da 9 12 14 17 19 23 50 54 67 72 76 verir. Değişen yalnız hangi anahtarın hangisinin ebeveyni olduğudur. 17'nin sol alt ağacında 9, 14 ve 12 vardı; bunların ortancası 12 alt ağacın kökü yapıldı, 9 ve 14 onun çocukları oldu. Sağda 76'nın sol alt ağacındaki 54, 72, 67 ve 76'nın kendisi yeniden dizildi; ortadaki 72 köke çıktı. Kökün öbür tarafına, 17'nin sağ alt ağacına ya da 50'ye dokunulmadı.

AVL ve kırmızı-siyah ağaçlar bu tür düzeltmeleri kendiliğinden yapar. Bir ekleme ya da silme yalnız kendi geçtiği yolu bozabilir; düzeltme de yalnız o yol üzerinde, aşağıdan yukarı doğru yapılır.

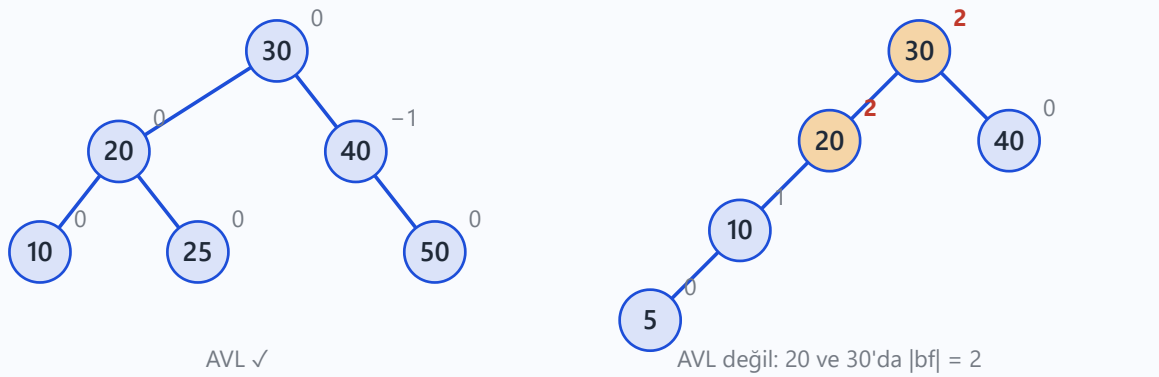
10.4.3 AVL ağacı ve denge faktörü

AVL ağacı, adını Sovyet matematikçiler Georgy Adelson-Velsky ve Evgenii Landis'in soyadlarının baş harflerinden alır. 1962'de yayımlanan AVL, kendini dengeleyen ilk ikili arama ağacıdır. Fikri sadedir: her düğümün iki alt ağacının yükseklikleri arasındaki fark en çok 1 olsun.

Tanım 10.5

Bir x düğümünün **denge faktörü** (*balance factor*) $bf(x) = h(sol) - h(sağ)$'dır. Burada $h(sol)$ ve $h(sağ)$, x'in sol ve sağ alt ağaçlarının yükseklikleridir. Boş alt ağacın yüksekliği −1'dir. **AVL ağacı**, her düğümünde $bf \in \{-1, 0, +1\}$ olan ikili arama ağacıdır.

Pozitif denge faktörü düğümün “sol ağır” olduğunu, negatif değer “sağ ağır” olduğunu söyler. Yaprakta iki alt ağaç da boştur: $bf = -1 - (-1) = 0$. Yalnız sol çocuğu olan bir düğümde $bf = 0 - (-1) = 1$'dir. Bu yüzden boş alt ağacın yüksekliğinin −1 alınması önemlidir; 0 alınsaydı yaprakla tek çocuklu düğüm aynı görünürdü.



Şekil 10.23: Düğümlerin sağ üstündeki sayı denge faktörüdür. Solda geçerli bir AVL ağacı: 30(20(10, 25), 40(—, 50)); 40'ın denge faktörü −1, ötekilerin 0. Sağda 5, 10, 20 zinciri yüzünden 20 ve 30'un denge faktörü 2 olmuş; ağaç AVL değil.

Sağdaki ağaçta iki düğüm dengesizdir. Böyle durumlarda **en alttaki** dengesiz düğüm, yani yeni eklenen düğüme en yakın olan, düzeltilir. Bu örnekte o düğüm 20'dir. 20 düzeltildiğinde alt ağacının yüksekliği bir azalır ve 30 da kendiliğinden dengeye gelir.

AVL ağacında her düğüm kendi yüksekliğini bir alanda saklar. Denge faktörü gerektiğinde iki çocuğun yüksekliğinden hesaplanır. Ekleme ve silme önce düz BST gibi yapılır. Sonra özyineleme geri dönerken yol üzerindeki her düğümün yüksekliği güncellenir ve denge faktörü denetlenir. *|bf| = 2* olan bir düğüme rastlanırsa döndürmeyle düzeltilir.

> **Sık yapılan hata**
>
> Denge faktörünü alt ağaçlardaki *düğüm sayısı* farkıyla hesaplamak. AVL koşulu yükseklikle ilgilidir. 4(2(1, 3), 5) ağacında kökün solunda üç, sağında bir düğüm vardır; ama sol yükseklik 1, sağ yükseklik 0'dır ve *bf = 1*'dir. Ağaç geçerli bir AVL ağacıdır.

10.4.4 AVL yüksekliği neden O(log n)?

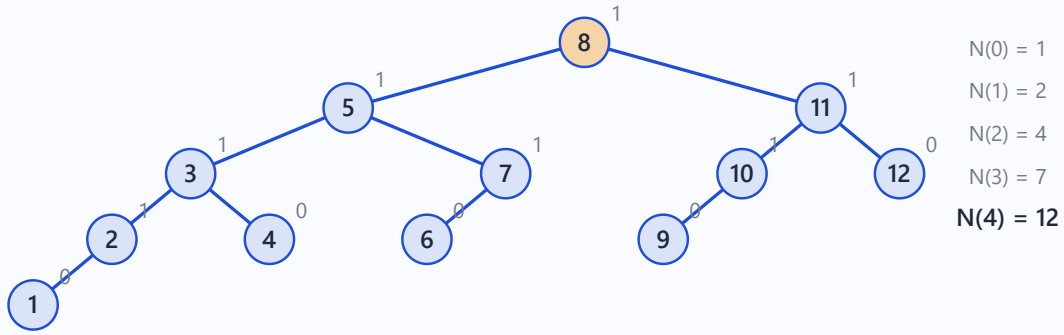
AVL koşulu yerel bir koşuldur; her düğüm yalnız iki çocuğunun yüksekliğine bakar. Bu yerel koşulun bütün ağacın yüksekliğini sınırladığını göstermek için soruyu tersinden soralım: yüksekliği *h* olan bir AVL ağacında *en az* kaç düğüm vardır? Bu sayıya *N(h)* diyelim. En az düğümlü, yani "en seyrek" AVL ağacında kökün bir alt ağacının yüksekliği *h − 1* olmak zorundadır (yoksa ağacın yüksekliği *h* olmazdı). Öbür alt ağaç, AVL koşulunun izin verdiği en kısa yükseklikte, *h − 2*'de olur. İki alt ağaç da kendi yüksekliklerinde en seyrek ağaçlardır. Buradan bir yineleme bağıntısı çıkar:

> $N(0) = 1, \quad N(1) = 2, \quad N(h) = 1 + N(h-1) + N(h-2)$

h	Hesap	*N(h)*
0	tek düğüm	1

h	Hesap	*N(h)*
1	kök + tek çocuk	2
2	1 + N(1) + N(0) = 1 + 2 + 1	4
3	1 + N(2) + N(1) = 1 + 4 + 2	7
4	1 + N(3) + N(2) = 1 + 7 + 4	12
5	1 + N(4) + N(3) = 1 + 12 + 7	20
6	1 + N(5) + N(4) = 1 + 20 + 12	33

Tablo 10.13: Yüksekliği *h* olan AVL ağacındaki en az düğüm sayısı.



Şekil 10.24: Yüksekliği 4 olan en seyrek AVL ağacı: 12 düğüm. Kök (turuncu), yüksekliği 3 olan en seyrek ağacı (7 düğüm) sol, yüksekliği 2 olan en seyrek ağacı (4 düğüm) sağ alt ağaç olarak taşır. Her iç düğümde iki alt ağacın yükseklik farkı tam 1'dir.

Sayılar Fibonacci sayıları gibi büyür; gerçekten de $N(h) = F(h + 3) - 1$'dir ($F(1) = F(2) = 1$). Fibonacci sayıları üstel büyüdüğü için $N(h)$ de üstel büyür. Kesin bir alt sınır şöyle bulunur. $N(h - 1) > N(h - 2)$ olduğundan $N(h) = 1 + N(h - 1) + N(h - 2) > 2 \cdot N(h - 2)$'dir. Her iki düzeyde düğüm sayısı en az iki katına çıkar. Bunu $h/2$ kez uygulayınca $N(h) > 2^{h/2}$ bulunur.

Teorem 10.2 (AVL yüksekliği)

n düğümlü bir AVL ağacının yüksekliği $h < 2 \log_2 n$'dir. Fibonacci sayılarıyla yapılan daha sıkı hesap $h < 1{,}44 \log_2(n + 2)$ verir. Bu yüzden AVL ağacında arama, ekleme ve silme $O(\log n)$ sürer.

İspat. Yüksekliği h olan ağaçta en az $N(h)$ düğüm vardır: $n \geq N(h) > 2^{h/2}$. İki tarafın 2 tabanında logaritması alınınca $\log_2 n > h/2$, yani $h < 2 \log_2 n$ çıkar. □

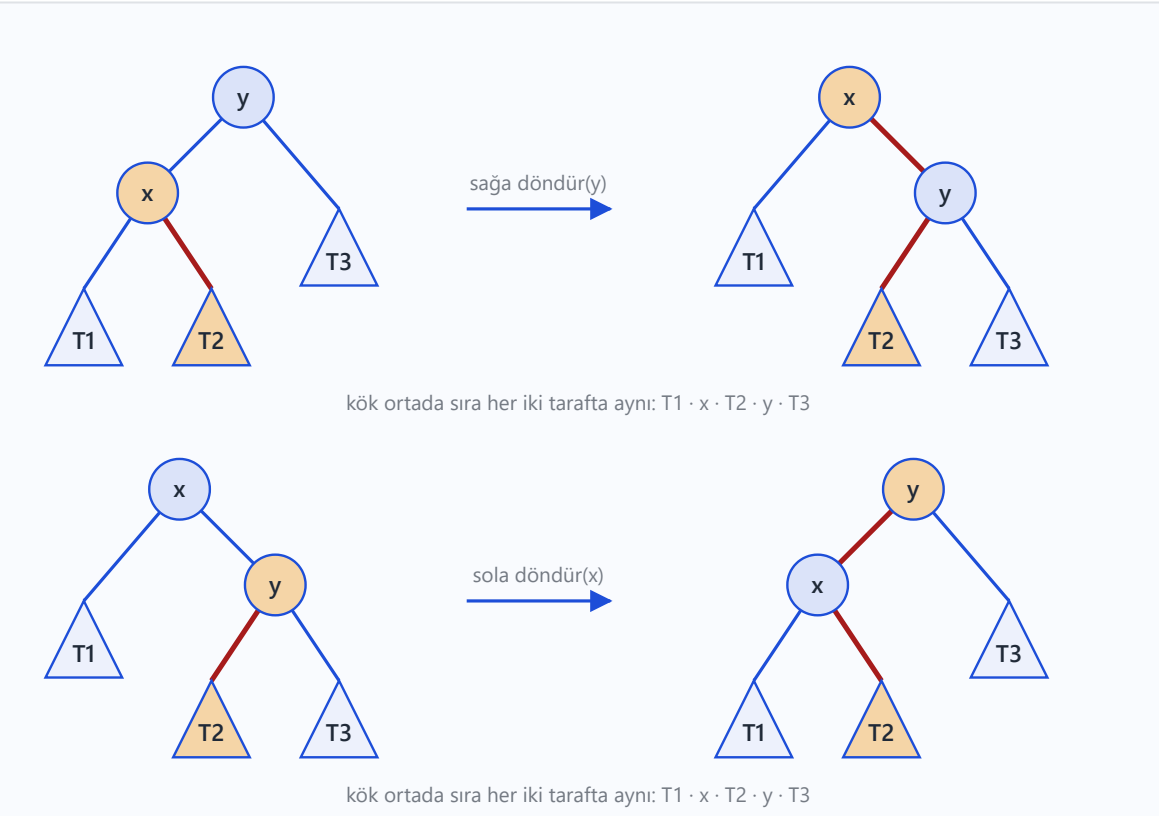
Örnek 10.5

30 düğümlü bir AVL ağacının yüksekliği en çok kaçtır? Yüksekliği 6 olan bir AVL ağacı en az *N(6) = 33* düğüm ister; 30 düğüm buna yetmez. Yüksekliği 5 olan bir ağaç ise en az *N(5) = 20* düğümle kurulabilir. Cevap 5'tir. Karşılaştırma için 30 düğümlü bir düz BST'nin yüksekliği 29'a kadar çıkabilir; mükemmel dengeli bir ağaçta ise en az 4'tür (31 düğümlü mükemmel ağacın yüksekliği 4'tür).

Sonuç şunu söyler: AVL ağacı mükemmel dengeli bir ağaçtan en kötü durumda yalnız %44 kadar daha yüksektir. Bir milyon anahtarlı AVL ağacının yüksekliği 28'i geçmez. Düz BST'deki milyonluk zincirle karşılaştırılınca fark çok büyüktür.

10.4.5 Döndürme

Dengesizliği düzeltmenin temel aracı **döndürmedir** (*rotation*). Döndürme, bir düğüm ile çocuklarından birinin ebeveyn–çocuk ilişkisini tersine çevirir. Çocuk bir düzey yukarı çıkar, eski ebeveyn onun altına iner. Bu sırada BST özelliğinin bozulmaması için bir alt ağacın sahibi değişir.



Şekil 10.25: Üstte sağa döndürme: sol ağır *y*'de *x* yukarı çıkar, *y* onun sağ çocuğu olur; *x*'in sağ alt ağacı T2, *y*'nin soluna geçer. Altta sola döndürme bunun simetriğidir. Kalın çizgiler değişen bağlardır. Kök ortada sıra iki tarafta da T1 · x · T2 · y · T3'tür.

Sağa döndürmeyi ayrıntılı inceleyelim. Başlangıçta *y* ebeveyndir ve *x* onun sol çocuğudur. *x*'in iki alt ağacı T1 ve T2, *y*'nin sağ alt ağacı T3'tür. BST özelliğine göre kök ortada sıra şudur: T1'deki anahtarlar, *x*, T2'deki anahtarlar, *y*, T3'teki anahtarlar. Döndürmeden sonra *x* yukarıdadır ve *y* onun sağ çocuğudur. T1 *x*'in solunda kalır, T3 *y*'nin sağında kalır. T2'nin yeri değişir: *x*'ten büyük ve *y*'den küçük olduğu için *y*'nin sol alt ağacı olur. Kök ortada sıra yine T1 · x · T2 · y · T3'tür.

Teorem 10.3 (Döndürme)

Bir döndürme, ağacın kök ortada sırasını değiştirmez; bu yüzden BST özelliğini korur. Döndürmede yalnız üç bağ değişir ve işlem *O(1)* sürer. Döndürülen alt ağaçta bir taraf bir düzey kısalır, öbür taraf bir düzey uzar.

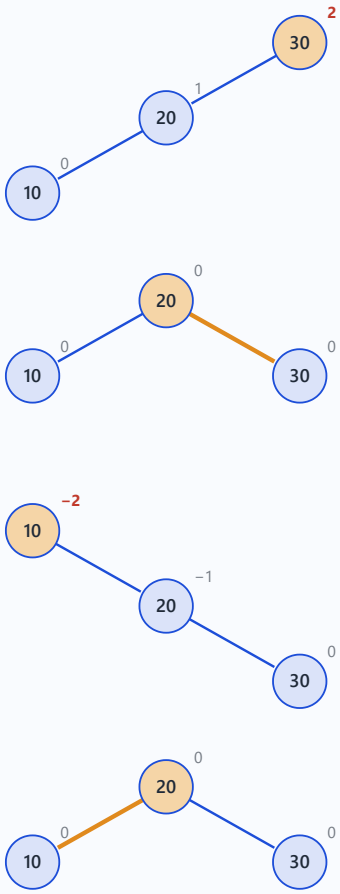
Üç bağ değişir. *y*'nin sol bağı artık T2'yi, *x*'in sağ bağı artık *y*'yi gösterir. Ebeveynden gelen bağ da *y* yerine *x*'i gösterir. Ağacın büyüklüğü ne olursa olsun bu üç atama yeter; döndürme sabit sürelidir. Yükseklik açısından bakıldığında T1 bir düzey yukarı çıkar, T3 bir düzey aşağı iner, T2 aynı düzeyde kalır. Sol ağır bir düğümde ağır taraf T1 ise sağa döndürme ağırlığı dengeler.

Sık yapılan hata

Döndürmede T2'yi unutmak. *x.sag = y* ataması önce yapılırsa *x*'in eski sağ alt ağacına giden tek referans kaybolur ve T2 ağaçtan düşer. Atama sırası önemlidir: önce `y.sol = x.sag` ile T2 güvenli yere taşınır, sonra `x.sag = y` yazılır.

10.4.6 Tek döndürmeler: LL ve RR

Bir eklemeden sonra en alttaki dengesiz düğüm *z* olsun. Dengesizliğin dört biçimi vardır. Adlar, yeni düğüme *z*'den inen yolun ilk iki adımını söyler: L sol (*left*), R sağ (*right*) demektir. LL'de yeni düğüm *z*'nin sol çocuğunun sol alt ağacına eklenmiştir. RR bunun simetriğidir. Bu iki durumda *z* ile çocuğu aynı yöne eğiktir ve tek döndürme yeter.



Şekil 10.26: Tek döndürmeler; düğümlerin sağ üstündeki sayı denge faktörüdür. (a) LL: 10, sol çocuğun (20) soluna eklendi; *bf(30) = 2*, *bf(20) = 1*. (b) 30'da sağa döndürme: 20 yukarı çıkar, 30 onun sağ çocuğu olur; bütün denge faktörleri 0. (c) RR: 30, sağ çocuğun (20) sağına eklendi; *bf(10) = −2*, *bf(20) = −1*. (d) 10'da sola döndürme.

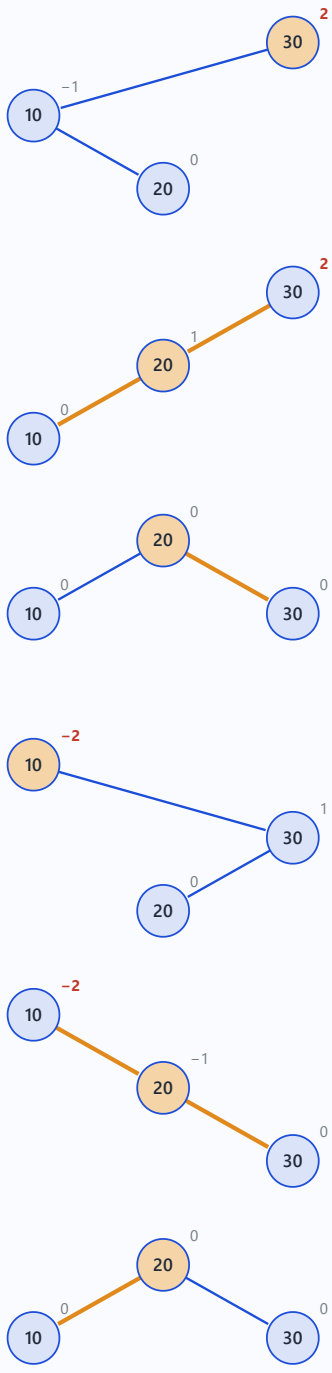
LL durumu denge faktörlerinden tanınır. *bf(z) = 2*'dir, yani *z* sol ağırdır. Sol çocuğun denge faktörü de 1 ya da 0'dır; çocuk sola eğik ya da dengelidir. Çözüm *z*'de sağa döndürmedir. RR'de *bf(z) = −2* ve sağ çocuğun denge faktörü −1 ya da 0'dır; çözüm *z*'de sola döndürmedir.

Ekleme sonrasında tek döndürmenin güzel bir sonucu vardır: döndürülen alt ağacın yüksekliği, eklemeden önceki değerine geri döner. Şekil 10.26'da 30'un alt ağacının yüksekliği 10 eklenmeden önce 1'di (30 ve 20); ekleme 2'ye çıkardı; döndürme yeniden 1'e indirdi. Daha yukarıdaki atalar yüksekliğin değiştiğini hiç görmez. Bu yüzden eklemede tek bir düzeltme yeterlidir.

10.4.7 Çift döndürmeler: LR ve RL

Kalan iki durumda yeni düğüm "zikzak" bir yolun sonundadır. LR'de yeni düğüm *z*'nin sol çocuğunun *sağ* alt ağacına eklenmiştir. *z* sola, çocuğu sağa eğiktir; üç düğüm bir üçgen çizer. Bu durumda *z*'de tek sağa döndürme işe yaramaz: üçgeni ters yöne çevirir ve ağaç yine dengesiz kalır.

Çözüm iki adımlıdır. Önce çocukta sola döndürülür; üçgen düz bir çizgiye, yani LL durumuna dönüşür. Sonra *z*'de sağa döndürülür. Sonuçta üç anahtarın ortancası köke çıkar, küçüğü sola, büyüğü sağa yerleşir. RL bunun simetriğidir: önce çocukta sağa, sonra *z*'de sola döndürülür.



Şekil 10.27: Çift döndürmeler. Üstte LR: (a) 20, sol çocuğun (10) sağına eklendi; *bf(30) = 2*, *bf(10) = −1*: üçgen. (b) 10'da sola döndürme: üçgen düz çizgiye (LL) döndü, *bf(30)* hâlâ 2. (c) 30'da sağa döndürme: ortanca 20 kök oldu. Altta RL: (d) 20, sağ çocuğun (30) soluna eklendi; *bf(10) = −2*, *bf(30) = 1*. (e) 30'da sağa döndürme: düz çizgi (RR). (f) 10'da sola döndürme: 20 kök.

Durum	*bf(z)*	Çocuğun *bf*'si	Düzeltme
LL	+2	sol çocuk ≥ 0	*z*'de sağa döndür
RR	−2	sağ çocuk ≤ 0	*z*'de sola döndür
LR	+2	sol çocuk < 0	sol çocukta sola, sonra *z*'de sağa
RL	−2	sağ çocuk > 0	sağ çocukta sağa, sonra *z*'de sola

Tablo 10.14: AVL'de dört dengesizlik durumu (*z*: en alttaki dengesiz düğüm).

Sık yapılan hata

Durumu yalnız *z*'nin denge faktörüne bakarak seçmek. *bf(z) = 2* yalnız "sol ağır" der; LL mi LR mi olduğu çocuğun eğimine bağlıdır. Boş bir AVL ağacına 50, 20, 30 eklenince *bf(50) = 2* olur, ama *bf(20) = −1*'dir: durum LR'dir. 50'de tek sağa döndürme 20(—, 50(30, —)) verir ve *bf(20) = −2* olur; ağaç yine dengesizdir. Doğru sonuç 30(20, 50)'dir.

10.4.8 AVL kodu

AVL düğümü anahtar, yükseklik ve iki çocuk referansı taşır. Sunumdaki kod, önce üç küçük yardımcı ile sağa döndürmeyi verir:

```
int h(AvlDugum d) {
    return d == null ? -1 : d.yukseklik;
}
int bf(AvlDugum d) {
    return h(d.sol) - h(d.sag);
}
void guncelle(AvlDugum d) {
    d.yukseklik = 1 +
        Math.max(h(d.sol), h(d.sag));
}
AvlDugum sagaDondur(AvlDugum y) {
    AvlDugum x = y.sol;
    y.sol = x.sag;  // T2 → y.sol
    x.sag = y;      // y → x.sag
    guncelle(y);    // önce alttaki
    guncelle(x);
    return x;       // yeni kök
}
```

`h`, boş alt ağaç için −1 döndürür; böylece öteki metotlar `null` kontrolüyle uğraşmaz. `bf`, iki çocuğun yükseklik farkıdır. `guncelle`, bir düğümün yüksekliğini çocuklarınınkinden yeniden hesaplar. Bu hesap, çocukların yüksekliğinin doğru olmasına dayanır. Bu yüzden yükseklikler her zaman aşağıdan yukarı güncellenir.

`sagaDondur(y)`, Şekil 10.25'teki adımları kodlar. `x`, yukarı çıkacak sol çocuktur. `y.sol = x.sag` T2'yi taşır; `x.sag = y` ilişkiyi tersine çevirir. Sonra iki düğümün yüksekliği güncellenir: önce `y`, sonra `x`. Döndürmeden sonra `y`, `x`'in çocuğudur; `x`'in yüksekliği `y`'nin yeni yüksekliğine bağlıdır. Sıra ters olsaydı `x` eski, yanlış bir değerden hesaplanırdı. Metot yeni alt ağaç kökünü, `x`'i döndürür. Üçüncü bağı, yani ebeveynden gelen bağı, çağıran bu dönüş değerini kendi bağına atayarak değiştirir.

Sola döndürme simetriktir; sunumda "sol ↔ sag simetriği" diye kısaltılan kod açık hâliyle şöyledir:

```
AvlDugum solaDondur(AvlDugum x) {
    AvlDugum y = x.sag;
    x.sag = y.sol;  // T2 → x.sag
    y.sol = x;      // x → y.sol
    guncelle(x);    // önce alttaki
    guncelle(y);
    return y;       // yeni kök
}
```

Dört durumu bir araya getiren metot `dengele`'dir:

```
AvlDugum dengele(AvlDugum d) {
    guncelle(d);
    if (bf(d) > 1) {       // sol ağır
        if (bf(d.sol) < 0)  // LR
            d.sol = solaDondur(d.sol);
        return sagaDondur(d); // LL
    }
    if (bf(d) < -1) {      // sağ ağır
        if (bf(d.sag) > 0)  // RL
            d.sag = sagaDondur(d.sag);
        return solaDondur(d); // RR
    }
    return d;
}
// solaDondur: sol ⇔ sag simetriği
// ekle/sil BST gibi; dönüşte:
//   return dengele(kok);
```

`dengele(d)` önce `d`'nin yüksekliğini günceller. Denge faktörü 1'den büyükse düğüm sol ağırdır. Sol çocuk sağa eğikse (`bf(d.sol) < 0`) durum LR'dir; önce sol çocukta sola döndürülür ve sonuç `d.sol`'a yazılır. Ardından her iki durumda da `d`'de sağa döndürülür. Denge faktörü −1'den küçükse aynı mantık simetrik olarak işler. Denge bozuk değilse `d` olduğu gibi döner.

Ekleme, düz BST eklemesinin son satırını değiştirerek elde edilir:

```
AvlDugum ekle(AvlDugum d, int k) {
    if (d == null) return new AvlDugum(k);
    if (k < d.anahtar)
        d.sol = ekle(d.sol, k);
    else if (k > d.anahtar)
        d.sag = ekle(d.sag, k);
    else return d;          // tekrar yok
    return dengele(d);
}
```

Fark tek satırdadır: `return d` yerine `return dengele(d)`. Özyineleme geri dönerken yeni yapraktan köke kadar yol üzerindeki her düğümde `dengele` çağrılır. Böylece yükseklikler aşağıdan yukarı güncellenir ve dengesiz bir düğüm bulunursa hemen düzeltilir. Silme de aynı biçimde düz BST silmesinin dönüşlerine `dengele` eklenerek yazılır.

Maliyet: iniş *O(log n)* düzeydir. Dönüşte her düzeyde `dengele` sabit iş yapar; döndürmeler de sabit sürelidir. Ekleme ve silme *O(log n)* sürer. Eklemede en çok bir düzeltme (bir tek ya da bir çift döndürme) gerekir. Silmede bir düzeltme alt ağacın yüksekliğini azaltabilir ve dengesizlik yukarıya taşınabilir; en kötü durumda yolun her düzeyinde, yani *O(log n)* kez döndürme yapılır.

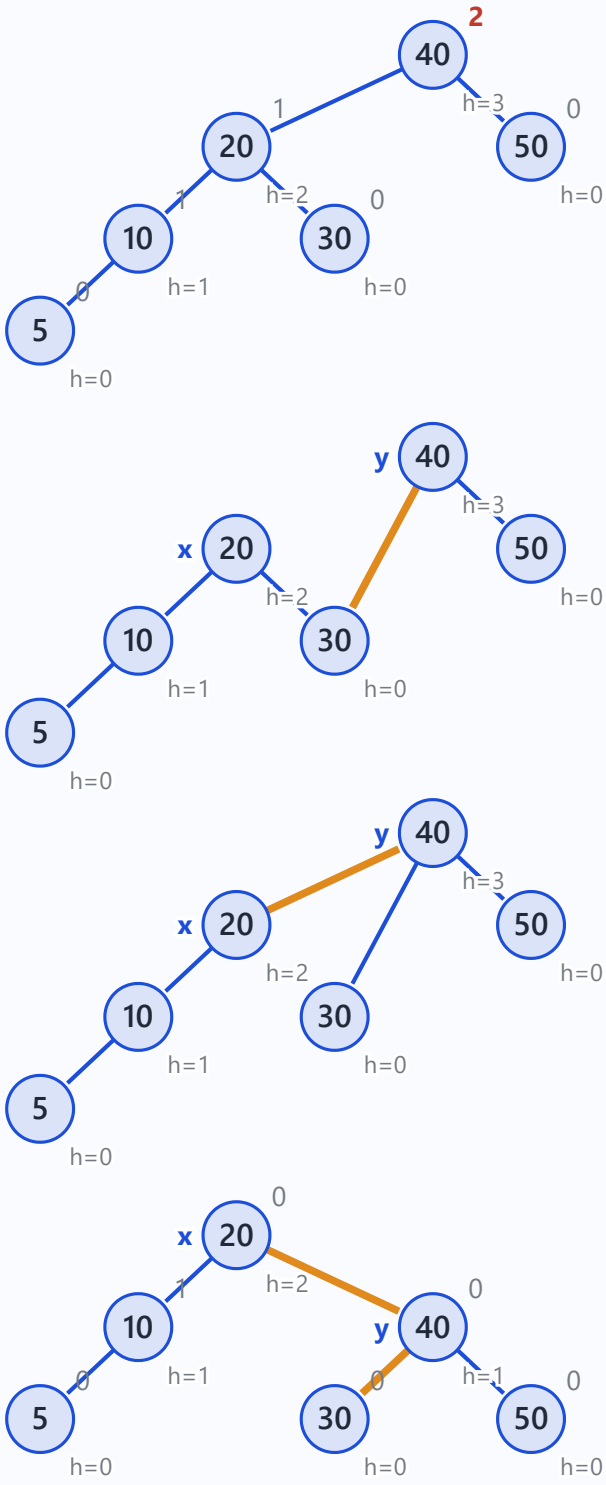
10.4.9 Kod izleme: sagaDondur ve solaDondur

İlk izleme LL durumudur. Ağaç 40(20(10, 30), 50)'dir ve 5 eklenir. 5, 40 → 20 → 10 yolundan inip 10'un soluna yerleşir. Dönüşte `dengele(10)` ve `dengele(20)` döndürme gerektirmez (*bf* sırasıyla 1 ve 1). `dengele(40)` ise *bf = 2* bulur; sol çocuk 20'nin *bf*'si 1'dir, durum LL'dir ve `sagaDondur(40)` çağrılır.

Satır	Etki	Değişen
`x = y.sol`	yukarı çıkacak çocuk 20; henüz bağ değişmedi	—
`y.sol = x.sag`	40'ın sol bağı artık T2'yi, yani 30'u gösterir; 20'ye yalnız `x` ulaşır	bağ 1
`x.sag = y`	20'nin sağ bağı 30 yerine 40'ı gösterir	bağ 2
`guncelle(y)`	40.yukseklik = 1 + max(0, 0) = 1 (önceden 3)	yükseklik
`guncelle(x)`	20.yukseklik = 1 + max(1, 1) = 2 (aynı kaldı, ama yeniden hesaplandı)	yükseklik
`return x`	yeni alt ağaç kökü 20; çağıran onu kendi bağına atar	bağ 3

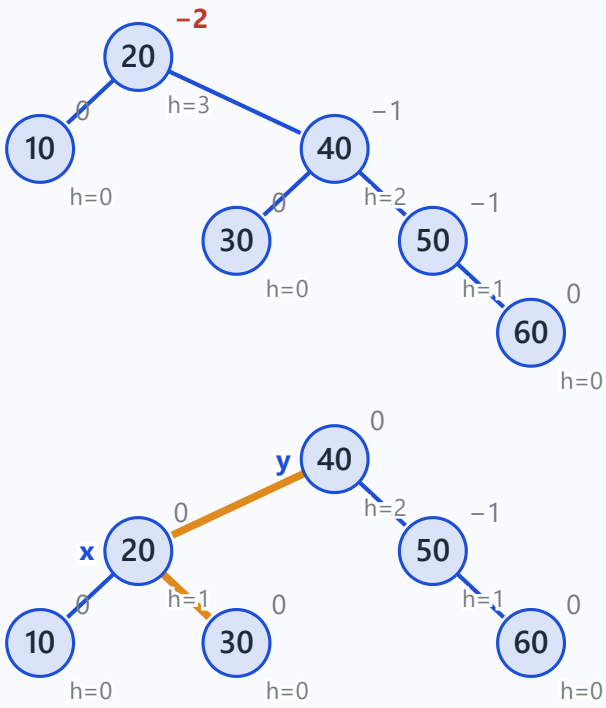
Tablo 10.15: `sagaDondur(y)` izlemesi, `y` = 40 (LL durumu).

Sonuç ağacı 20(10(5, —), 40(30, 50))'dir. Denge faktörleri: 20'de 0, 10'da 1, 40'ta 0, yapraklarda 0. Kök ortada sıra 5 10 20 30 40 50'dir ve döndürmeden önceki sırayla aynıdır. 40'ın yüksekliği 3'ten 1'e indi, alt ağacın kökü 20'nin yüksekliği ise 2 oldu; ekleme öncesindeki alt ağacın yüksekliği de 2'ydi.



Şekil 10.28: `sagaDondur(40)` ; düğümlerin yanında denge faktörü ve yükseklik yazılıdır. (a) 5 eklendi; *bf(40) = 2*, *bf(20) = 1*: LL. (b) `y.sol = x.sag` : 40'ın sol bağı 30'u gösteriyor. (c) `x.sag = y` : 40, 20'nin sağ çocuğu oldu. (d) Yeni alt ağaç kökü 20; bütün denge faktörleri {−1, 0, 1} içinde.

İkinci izleme RR durumudur ve öncekinin aynadaki görüntüsüdür. Ağaç 20(10, 40(30, 50))'dir ve 60 eklenir. `dengele(20)` *bf* = −2 bulur, sağ çocuk 40'ın *bf*'si −1'dir ve `solaDondur(20)` çağrılır. `y = x.sag` 40'ı tutar. `x.sag = y.sol` 20'nin sağ bağını 30'a yöneltir. `y.sol = x` 20'yi 40'ın sol çocuğu yapar. `guncelle(x)` 20'nin yüksekliğini 1 + max(0, 0) = 1 bulur (önceden 3), `guncelle(y)` 40'ınkini 1 + max(1, 1) = 2 bulur. Sonuç 40(20(10, 30), 50(—, 60))'tır.



Şekil 10.29: `solaDondur(20)`. (a) 60 eklendi; *bf(20) = −2*, *bf(40) = −1*: RR. (b) Yeni alt ağaç kökü 40; 20 onun sol çocuğu, 30 da 20'nin sağ çocuğu oldu.

Sık yapılan hata

Döndürmeden sonra yükseklikleri ters sırayla güncellemek. `guncelle(x)` önce çağrılırsa `x`'in yüksekliği, `y`'nin henüz güncellenmemiş eski yüksekliğinden (LL örneğinde 3) hesaplanır ve 4 bulunur. Ardından üst düğümlerin denge faktörleri yanlış çıkar ve gereksiz döndürmeler yapılır. Önce aşağı inen düğüm, sonra yeni kök güncellenir.

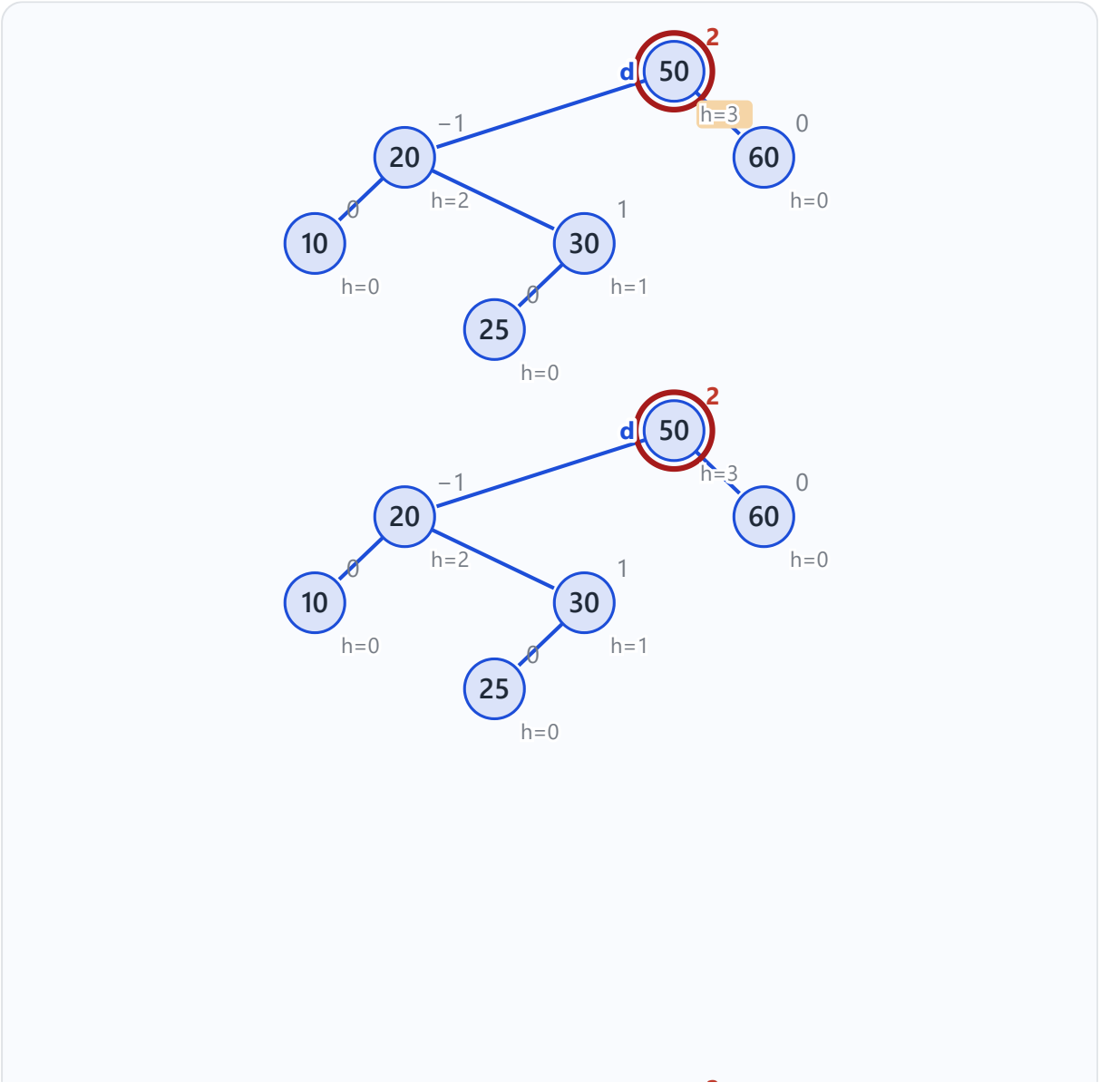
10.4.10 Kod izleme: dengele

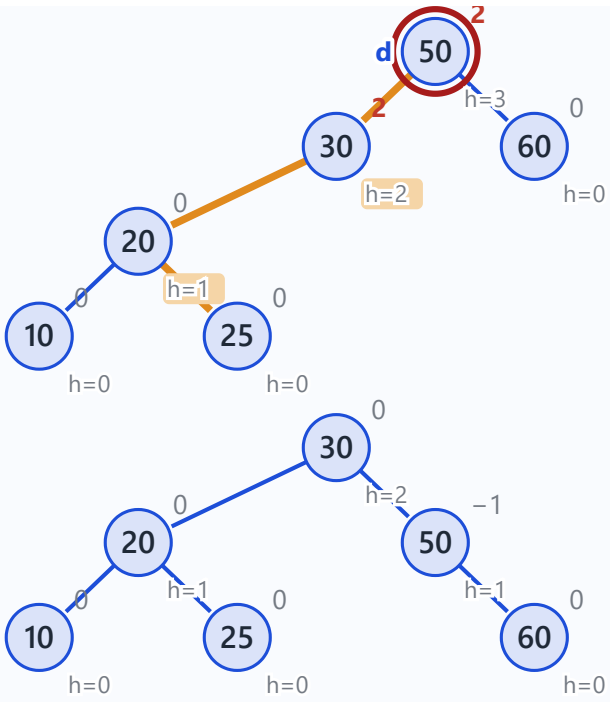
Çift döndürmenin kodda nasıl işlediğini LR örneğinde izleyelim. Ağaç 50(20(10, 30), 60)'tır ve 25 eklenir. 25, 50 → 20 → 30 yolundan inip 30'un soluna yerleşir. Özyineleme geri dönerken `dengele` yol üzerindeki her düğümde çağrılır.

Çağrı	Yükseklik	*bf*	Karar
`dengele(30)`	0 → 1	1	döndürme yok, `return d`
`dengele(20)`	1 → 2	0 − 1 = −1	döndürme yok, `return d`
`dengele(50)`	2 → 3	h(20) − h(60) = 2 − 0 = 2	sol ağır; *bf(20) = −1 < 0* → LR
`d.sol = solaDondur(20)`	30 yukarı çıkar, 25 20'nin sağına geçer		üçgen düz çizgiye döndü; *bf(50)* hâlâ 2
`return sagaDondur(50)`	30 kök olur, 50 onun sağına iner; 30'un eski sağı (`null`) 50'nin soluna geçer		yeni alt ağaç kökü 30

Tablo 10.16: 25 eklendikten sonra dönüşte `dengele` çağrıları (LR durumu).

Sonuç 30(20(10, 25), 50(—, 60))'tır. Yükseklik 3'ten 2'ye indi; ekleme öncesindeki değer de 2'ydi. Denge faktörleri 30'da 0, 20'de 0, 50'de −1, yapraklarda 0'dır. İlk döndürmenin sonucunun `d.sol`'a atanması şarttır: `solaDondur(20)` yeni alt ağaç kökü olarak 30'u döndürür ve 50'nin sol bağı ancak bu atamayla 30'u gösterir.





Şekil 10.30: `dengele(50)` izlemesi. (a) Yükseklik 3, *bf = 2*. (b) *bf(20) = −1*: sol çocuk sağa eğik, LR. (c) `solaDondur(20)` : 30 yukarı çıktı, üçgen düz çizgiye döndü. (d) `sagaDondur(50)` : 30 yeni alt ağaç kökü; yükseklik 3'ten 2'ye indi.

10.4.11 AVL'ye ekleme örneği

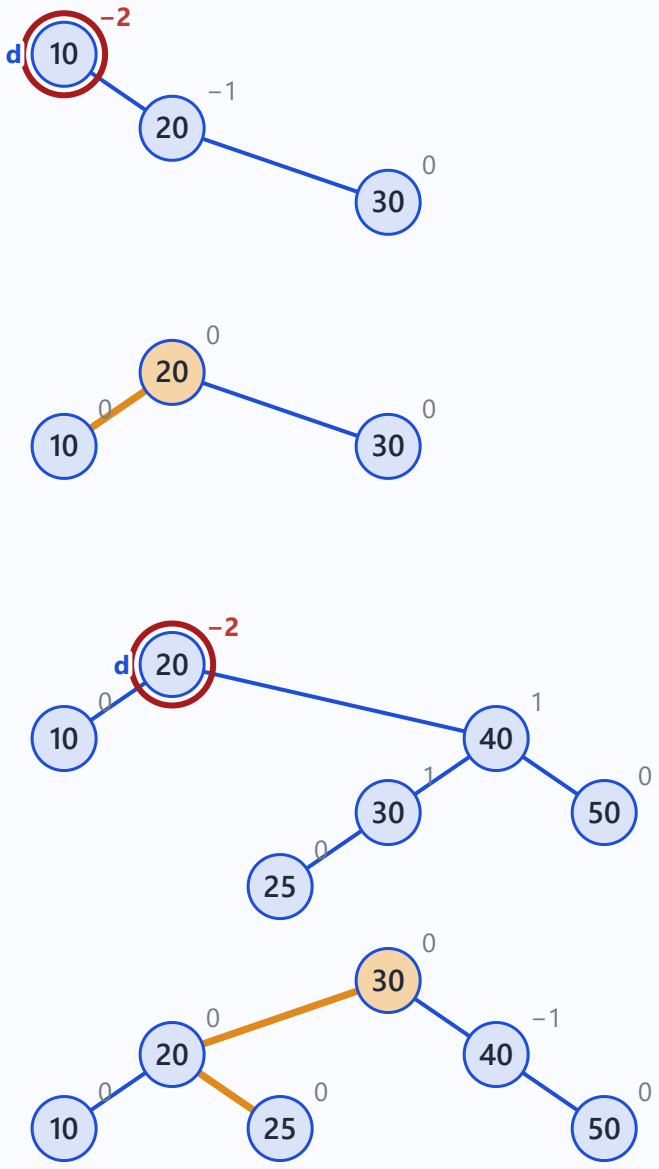
Boş bir AVL ağacına sırasıyla 10, 20, 30, 40, 50, 25 eklensin. Düz BST bu girdiyle (son anahtar dışında) sağa uzanan bir zincir kurardı. Tablo 10.17 her eklemeden sonraki ağacı ve yapılan döndürmeleri verir.

Eklem e	İniş	Dönüşte dengesizlik	Döndürme	Sonuç ağacı
10	boş ağaç	—	—	10
20	10 → sağ	*bf(10) = −1*, sorun yok	—	10(—, 20)
30	10 → 20 → sağ	*bf(10) = −2*, *bf(20) = −1*: RR	10'da sola	20(10, 30)
40	20 → 30 → sağ	*bf(30) = −1*, *bf(20) = −1*, sorun yok	—	20(10, 30(—, 40))
50	20 → 30 → 40 → sağ	*bf(30) = −2*, *bf(40) = −1*: RR	30'da sola	20(10, 40(30, 50))
25	20 → 40 → 30 → sol	*bf(20) = −2*, *bf(40) = 1*: RL	40'ta sağa, 20'de sola	30(20(10, 25), 40(—, 50))

Tablo 10.17: Boş AVL ağacına 10, 20, 30, 40, 50, 25 eklenmesi.

50 eklendiğinde 30 ve 20 aynı anda dengesiz görünebilir mi? Hayır. Dönüşte önce `dengele(40)`, sonra `dengele(30)` çağrılır. 30 dengesizdir ve düzeltilir; alt ağacın yüksekliği eski değeri olan 1'e iner. `dengele(20)` çağrıldığında yükseklik 2 → 2 kalır, *bf(20) = −1*'dir. En alttaki dengesiz ata düzeltildiği için yukarısı etkilenmez.

Son eklemede 25, 30'un soluna düşer. Dönüşte *bf(30) = 1* ve *bf(40) = 1* sorun değildir. *bf(20) = h(10) − h(40) = 0 − 2 = −2*'dir ve sağ çocuk 40 sola eğiktir: RL. Önce 40'ta sağa döndürülür (30 yukarı çıkar), sonra 20'de sola döndürülür. 30 köke çıkar; 25, 20'nin sağına geçer. Altı ekleme sonunda ağacın yüksekliği 2'dir; düz BST'de aynı sıra 4 yüksekliğinde bir ağaç verirdi.



Şekil 10.31: AVL ekleme izlemesinden dört an. (a) 30 eklendi: *bf(10) = −2*, RR. (b) 10'da sola döndürme; 20 kök. (c) 25 eklendi: *bf(20) = −2*, *bf(40) = 1*, RL. (d) İki döndürmeden sonra 30 kök; bütün denge faktörleri {−1, 0, 1} içinde.

Örnek 10.6

Boş bir AVL ağacına 1, 2, …, 7 sırayla eklenirse ne olur? Her döndürme RR durumundadır.

1. 3 eklenince 1'de sola döndürülür: 2(1, 3).
2. 4 eklenir: 2(1, 3(—, 4)); dengesizlik yok.
3. 5 eklenince 3'te sola döndürülür: 2(1, 4(3, 5)).
4. 6 eklenince kök 2'de *bf* = *−2* olur; 2'de sola döndürülür: 4(2(1, 3), 5(—, 6)).
5. 7 eklenince 5'te sola döndürülür: 4(2(1, 3), 6(5, 7)).

Toplam dört döndürme yapılır ve sonuç yüksekliği 2 olan mükemmel ağaçtır. Sıralı girdi, düz BST'nin en kötü durumuydu; AVL ağacı aynı girdiden en iyi şekli üretti.

10.5 Kırmızı-siyah ağaçlar

AVL ağacı dengeyi sıkı tutar: her düğümde iki alt ağacın yükseklik farkı en çok 1'dir. Bu sıkılığın bedeli, silmede dengesizliğin köke kadar yayılabilmesi ve her düzeyde döndürme gerekebilmesidir. Kırmızı-siyah ağaç daha gevşek bir denge kabul eder. Karşılığında güncellemelerde çok az döndürme yapar ve çoğu düzeltmeyi yalnız renk değiştirerek halleder. Yapı, Guibas ve Sedgewick'in 1978'deki çalışmasında bugünkü renk diliyle tanımlanmıştır. Java'nın `TreeMap` ve `TreeSet` sınıfları ile C++'ın `std::map`'i bu yapıyı kullanır.

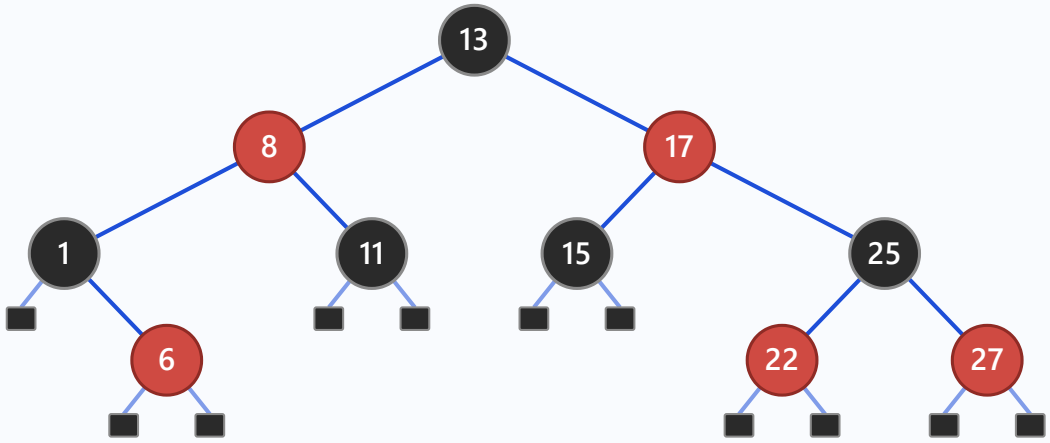
10.5.1 Kurallar

Tanım 10.6

Kırmızı-siyah ağaç (*red-black tree*), her düğümü kırmızı ya da siyah renkli olan ve şu beş kuralı sağlayan ikili arama ağacıdır:

1. Her düğüm kırmızı ya da siyahtır.
2. Kök siyahtır.
3. Bütün yapraklar, yani boş çocuklar (NIL), siyah sayılır.
4. Kırmızı bir düğümün çocukları siyahtır: art arda iki kırmızı düğüm bulunmaz.
5. Bir düğümden altındaki NIL yapraklara giden her yolda aynı sayıda siyah düğüm vardır. Bu sayıya düğümün **siyah yüksekliği** (*black height*) denir.

Üçüncü kural ilk bakışta tuhaf gelir: boş bir çocuğun rengi ne demektir? Kural, kodda `null` çocukların siyah sayılacağını söyler. Böylece kural 4 ve 5, çocuğu eksik düğümlerde de anlamlı kalır. Notta siyah düğüm sayılırken kök sayılır, NIL yapraklar sayılmaz.



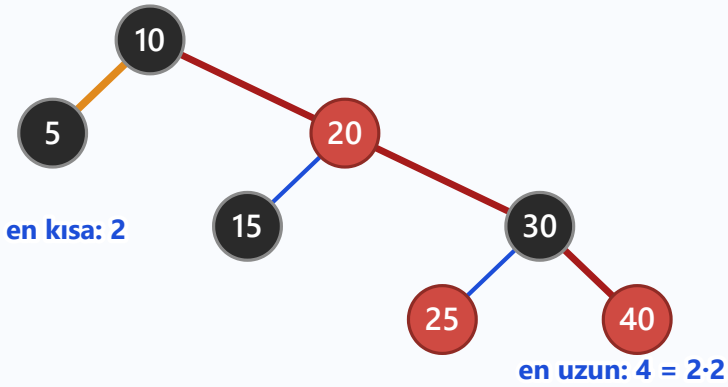
Şekil 10.32: Geçerli bir kırmızı-siyah ağaç; küçük siyah kareler NIL yapraklardır. Kök 13 siyah; 8 ve 17 kırmızı; 1, 11, 15, 25 siyah; 6, 22, 27 kırmızı. Kökten her NIL'e giden yolda 2 siyah düğüm vardır (kök dahil, NIL hariç).

Kuralları Şekil 10.32 üzerinde denetleyelim. Kırmızı 8'in çocukları 1 ve 11 siyahtır; kırmızı 17'nin çocukları 15 ve 25 siyahtır. Kırmızı yaprakların (6, 22, 27) çocukları NIL'dir ve siyah sayılır. Kural 4 sağlanır. Kural 5 için birkaç yol sayalım: 13 → 8 → 1 → NIL yolunda siyahlar 13 ve 1'dir; 13 → 8 → 1 → 6 → NIL yolunda da 13 ve 1'dir, 6 kırmızıdır. 13 → 17 → 25 → 27 → NIL yolunda siyahlar 13 ve 25'tir. Her yolda iki siyah vardır.

> **Sık yapılan hata**
>
> Kural 5'i yalnız en uzun ve en kısa yola bakarak denetlemek. Siyah sayısı *her* yolda eşit olmalıdır ve yollar NIL'e kadar sayılır. Tek çocuklu bir düğümün boş tarafı da bir yoldur. Bir örnek düşünelim: kök siyah 20, çocukları siyah 10 ve siyah 30, 30'un sol çocuğu da siyah 25 olsun. 20 → 10 → NIL yolunda 2 siyah vardır. 20 → 30 → 25 → NIL yolunda ise 3 siyah vardır. Ağaç kural 5'i bozar.

10.5.2 Kurallar dengeyi nasıl garanti eder?

Kurallar ağacın yüksekliğini doğrudan sınırlamaz. Sınır iki kuralın birlikte çalışmasından çıkar. Kökün siyah yüksekliği b olsun. Kökten bir NIL'e giden en kısa yol ne kadar kısa olabilir? Her yolda tam b siyah düğüm vardır; en kısa yol yalnız siyah düğümlerden oluşur ve b düğümlüdür. En uzun yol ne kadar uzun olabilir? Yine b siyah düğüm içerir, ama araya kırmızılar girebilir. Kural 4 iki kırmızının art arda gelmesini yasaklar; en fazla her siyahın altına bir kırmızı girer. En uzun yol en çok $2b$ düğümlüdür.



Şekil 10.33: Siyah yüksekliği 2 olan ağaçta en kısa yol 10 → 5 (2 düğüm, ikisi de siyah); en uzun yol 10 → 20 → 30 → 40 (4 düğüm, siyah ve kırmızı sırayla). En uzun yol, en kısanın 2 katıdır.

Teorem 10.4 (Kırmızı-siyah ağacın yüksekliği)

Kırmızı-siyah ağaçta kökten bir NIL'e giden en uzun yol, en kısa yolun en çok iki katıdır. n iç düğümlü bir kırmızı-siyah ağacın yüksekliği $h \le 2\,\log_2(n+1)$'dir.

İspat. Birinci cümle yukarıdaki sayımdan çıkar. İkincisi için şunu gözleyelim: siyah yüksekliği b olan bir düğümün alt ağacında en az $2^b - 1$ iç düğüm vardır. Bu, tümevarımla gösterilir. Siyah yüksekliği 0 olan alt ağaç NIL'dir ve $2^0 - 1 = 0$ düğüm içerir. Siyah yüksekliği b olan bir düğümün her çocuğunun siyah yüksekliği en az $b - 1$'dir. Tümevarım varsayımına göre her çocuğun alt ağacında en az $2^{b-1} - 1$ düğüm vardır. Düğümle birlikte toplam en az $2(2^{b-1} - 1) + 1 = 2^b - 1$'dir. Kökte kural 4'ten $b \ge h/2$ olduğu için $n \ge 2^{h/2} - 1$, buradan $h \le 2\,\log_2(n+1)$ çıkar. □

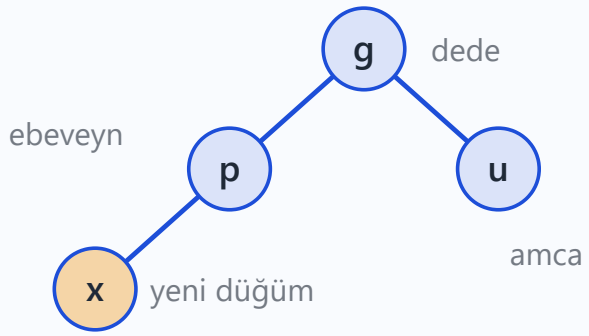
Sonuç olarak kırmızı-siyah ağaçta arama, ekleme ve silme $O(\log n)$ sürer. Ekleme sonrası düzeltmede en çok 2, silme sonrası düzeltmede en çok 3 döndürme yapılır; geri kalan iş renk değiştirmektir. Yükseklik sınırı AVL'ninkinden (yaklaşık $1{,}44\,\log_2 n$) gevşektir; yani kırmızı-siyah ağaç biraz daha uzun olabilir. Buna karşılık güncellemeleri daha ucuzdur ve düğüm başına ek bilgi yalnız bir bitlik renktir.

10.5.3 Ekleme ve düzeltme durumları

Ekleme iki aşamalıdır. Birinci aşamada yeni düğüm düz BST eklemesiyle yaprak olarak bağlanır ve **kırmızıya** boyanır. Neden kırmızı? Siyah eklenseydi, yeni düğümün bulunduğu yollarda siyah sayısı bir artar ve kural 5 bozulurdu. Kırmızı düğüm siyah yüksekliği değiştirmez. Bozulabilecek tek kural 4'tür: yeni düğümün ebeveyni de kırmızıysa art arda iki kırmızı oluşur. İkinci aşama bu bozukluğu gidermektir.

Düzeltmede dört düğüm rol alır (Şekil 10.34): yeni düğüm x, ebeveyni p, dedesi g ve **amcası** u, yani ebeveynin kardeşi. Ebeveyn kırmızıysa kök olamaz (kök siyahtır); demek ki bir dede vardır ve dede siyahtır (yoksa ağaçta önceden art arda iki kırmızı olurdu). Neyin yapılacağına amcanın rengi karar

verir.



Şekil 10.34: Düzeltmedeki roller: yeni düğüm *x*, ebeveyn *p*, dede *g*, amca *u*.

Durum	Yapılan	Sonra
Amca kırmızı	Renk değiştir: *p* ve *u* siyah, *g* kırmızı.	Sorun dedeye taşınabilir; *x* = *g* alınıp döngü sürer.
Amca siyah, üçgen	*x*–*p*–*g* zikzak yapıyorsa *p*'de döndür; *x* ile *p*'nin rolleri değişir.	Çizgi durumuna geçilir.
Amca siyah, çizgi	*g*'de döndür; eski ebeveyn siyah, eski dede kırmızı olur.	Bitti.

Tablo 10.18: Ekleme sonrası düzeltme durumları (ebeveyn kırmızıyken).

Ebeveyn siyahsa hiçbir şey yapılmaz; art arda kırmızı yoktur. En sonda kök her zaman siyaha boyanır. NIL amca siyah sayılır.

Renk değiştirmenin neden kural 5'i koruduğunu görelim. Önce *g* siyah, *p* ve *u* kırmızıdır. Sonra *g* kırmızı, *p* ve *u* siyahtır. *g*'den aşağı inen her yol, *p* ya da *u*'dan tam birinden geçer. Önce bu yolda bir siyah vardı (*g*), şimdi yine bir siyah var (*p* ya da *u*). Siyah sayıları değişmez. Ama *g* kırmızı olduğu için onun ebeveyni de kırmızıysa kural 4 bu kez yukarıda bozulur. Bu yüzden döngü *g*'den devam eder. Sorun her adımda iki düzey yukarı çıkar; en kötü durumda köke ulaşır ve kökün siyaha boyanmasıyla biter.

Amca siyahken renk değiştirmek işe yaramaz. Amca zaten siyahtır; *g*'yi kırmızı yapmak ise amcanın tarafındaki yollarda bir siyah eksiltir. Bunun yerine döndürme yapılır. Çizgi durumunda (*x* ile *p* aynı yönde çocuk) *g*'de döndürülür; *p* yukarı çıkar ve siyah olur, *g* aşağı iner ve kırmızı olur. Üçgen durumunda önce *p*'de döndürülerek çizgi elde edilir. Bu iki adım, AVL'deki tek ve çift döndürmelerin aynısıdır. Döndürmeden sonra alt ağacın kökü siyahtır ve döngü biter.

10.5.4 Düzeltme kodu

Kırmızı-siyah ağaç düğümünde anahtar ve iki çocuğun yanında bir renk alanı ve ebeveyne bir referans vardır. Ebeveyn referansı, düzeltmenin aşağıdan yukarı yürüyebilmesi içindir. Sunumdaki düzeltme kodu şöyledir:

```
void duzelt(Dugum x) {      // x: yeni, kırmızı
    while (x != kok && kirmizi(x.ebeveyn)) {
        Dugum p = x.ebeveyn, g = p.ebeveyn;
        Dugum u = (p == g.sol) ? g.sag : g.sol;
        if (kirmizi(u)) {         // amca kırmızı
            p.renk = u.renk = SIYAH;
            g.renk = KIRMIZI;
            x = g;                // sorun dedeye
        } else {                  // amca siyah
            if (ucgen(x, p, g)) { // zikzak
                dondur(p, x);     // çizgiye çevir
                Dugum t = x; x = p; p = t;
            }
            dondur(g, p);         // çizgi
            p.renk = SIYAH; g.renk = KIRMIZI;
        }
    }
    kok.renk = SIYAH;
}
```

Yardımcılar ve eklemenin bütünü yorum satırlarıyla verilmiştir:

```
// Ekleme: BST gibi yaprağa ekle,
// ebeveyni bağla, kırmızı boya:
//   x = bstEkle(anahtar);
//   x.renk = KIRMIZI; duzelt(x);

// NIL (null) siyah sayılır
boolean kirmizi(Dugum n) {
    return n != null
        && n.renk == KIRMIZI;
}
// x-p-g zikzak (üçgen) mı?
boolean ucgen(Dugum x, Dugum p,
              Dugum g) {
    return (p == g.sol)
        != (x == p.sol);
}
// dondur(a, b): b, a'nın çocuğu;
// b'yi a'nın yerine çıkarır (b
// soldaysa sağa, sağdaysa sola
// döndürme; ebeveyn bağları dahil)
```

`kirmizi(n)`, `null`'ı siyah sayar; böylece NIL amca ayrıca ele alınmaz. `ucgen(x, p, g)`, "*p* dedenin sol çocuğu mu?" ile "*x* ebeveynin sol çocuğu mu?" sorularının cevaplarını karşılaştırır. Cevaplar farklıysa *x*–*p*–*g* zikzaktır. `dondur(a, b)`, *a*'nın çocuğu olan *b*'yi *a*'nın yerine çıkarır; *b* soldaysa sağa, sağ-

daysa sola döndürür ve ebeveyn bağlarını da günceller. İzlemelerde `duzelt`'in satır numaraları kullanılır. Satır 2 döngü koşulu, satır 5 amca kontrolü, satır 10 üçgen kontrolüdür. Satır 14 dedede döndürür, satır 18 kökü siyaha boyar.

Döngüdeki `Dugum t = x; x = p; p = t;` satırı üçgen durumundan sonra rolleri değiştirir. *p*'de döndürmeden sonra *x* yukarı, *p* aşağı geçmiştir. Çizgi durumundaki `dondur(g, p)` çağrısının doğru düğümü döndürmesi için *p* değişkeninin artık yukarıdaki düğümü göstermesi gerekir.

Maliyet: BST eklemesi *O(log n)*'dir. Düzeltme döngüsü her turda en az iki düzey yukarı çıkar; en çok *O(log n)* tur döner. Döndürme yalnız döngünün son turunda yapılır (amca siyah dalı döngüyü bitirir) ve en çok iki tanedir.

> **Sık yapılan hata**
>
> Döngüden sonraki `kok.renk = SIYAH` satırını unutmak. Renk değiştirme dedeyi kırmızı yapar; dede kökse döngü "*x* kök" koşuluyla biter ve kök kırmızı kalır. Kural 2 bozulur. Bu satır, siyah yüksekliğin ağaç genelinde bir artmasının da tek yoludur.

10.5.5 Kod izleme: 10, 20, 30, 15

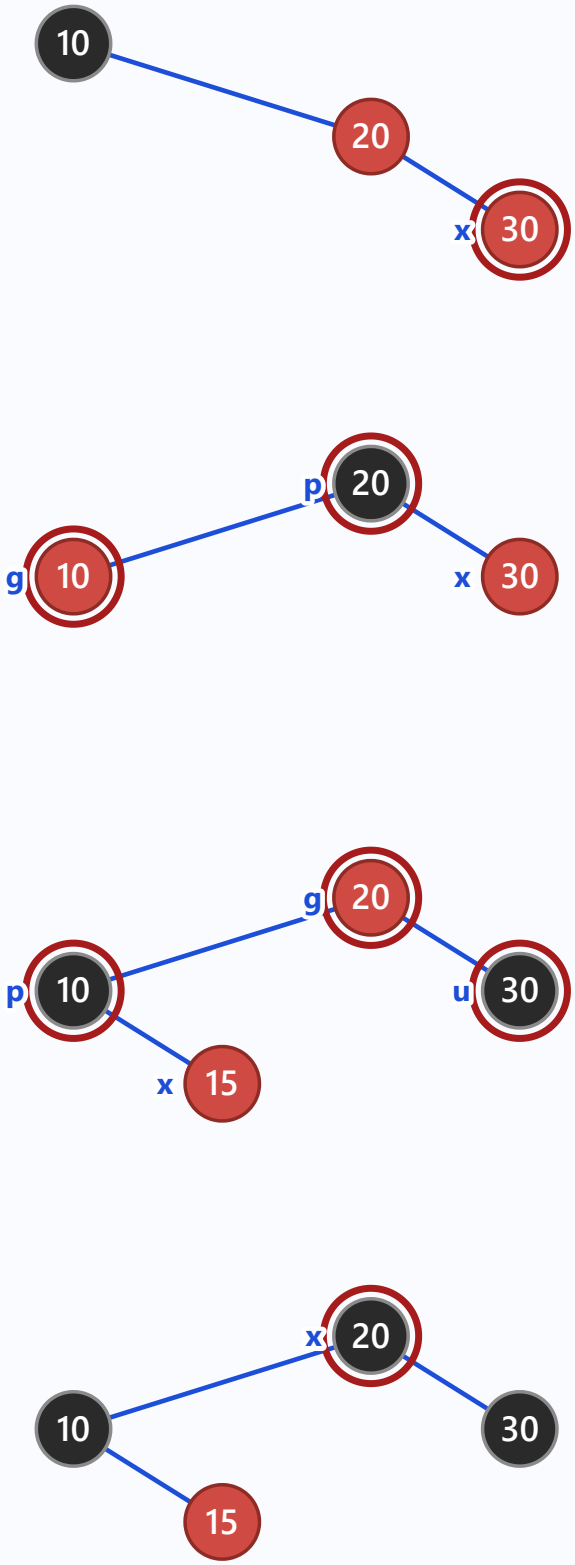
Boş bir kırmızı-siyah ağaca sırasıyla 10, 20, 30 ve 15 eklensin. Metinde renk, düğümün yanına yazılan K (kırmızı) ve S (siyah) harfleriyle gösterilir: **K**, **S**.

Ekleme	Döngü	*p*, *g*, *u*	Yapılan	Sonuç
10	x kök → girilmez	—	satır 18: kök siyah	10S
20	ebeveyn 10 siyah → girilmez	—	—	10S(—, 20K)
30	ebeveyn 20 kırmızı → girilir	20, 10, NIL (siyah)	çizgi: `dondur(10, 20)` sola; 20 siyah, 10 kırmızı	20S(10K, 30K)
15	ebeveyn 10 kırmızı → girilir	10, 20, 30K	amca kırmızı: 10 ve 30 siyah, 20 kırmızı; *x* = *20*; x kök → döngü biter; satır 18: 20 siyah	20S(10S(—, 15K), 30S)

Tablo 10.19: 10, 20, 30, 15 eklenirken `duzelt` izlemesi.

30 eklenirken amca NIL'dir ve siyah sayılır. 30, 20'nin sağ çocuğu; 20 de 10'un sağ çocuğudur. Üçü düz bir çizgi oluşturur; `ucgen` yanlış döner ve doğrudan satır 14'e geçilir. Dededе sola döndürme, AVL'deki RR düzeltmesinin aynısıdır. Renkler değiştirilince alt ağacın kökü (20) siyah olur ve döngü biter.

15 eklendiğinde amca 30 kırmızıdır; döndürme gerekmez, renk değiştirmek yeter. Dede 20 kırmızı olur ve döngü 20'den devam eder. 20 kök olduğu için döngü koşulu yanlıştır. Satır 18 kökü yeniden siyaha boyar. Bu ekleme, ağacın siyah yüksekliğini 1'den 2'ye çıkardı: artık kökten her NIL'e giden yolda iki siyah düğüm vardır.



Şekil 10.35: İlk izlemeden dört an. (a) 30 eklendi ve kırmızı boyandı; ebeveyn 20 de kırmızı. (b) 10'da sola döndürme ve renk değişiminden sonra 20 siyah, 10 ve 30 kırmızı. (c) 15 eklendi; amca 30 kırmızı olduğu için

yalnız renk değişti: 10 ve 30 siyah, 20 kırmızı. (d) Kök yeniden siyah; siyah yükseklik 1 arttı.

10.5.6 Kod izleme: 12 ve 5

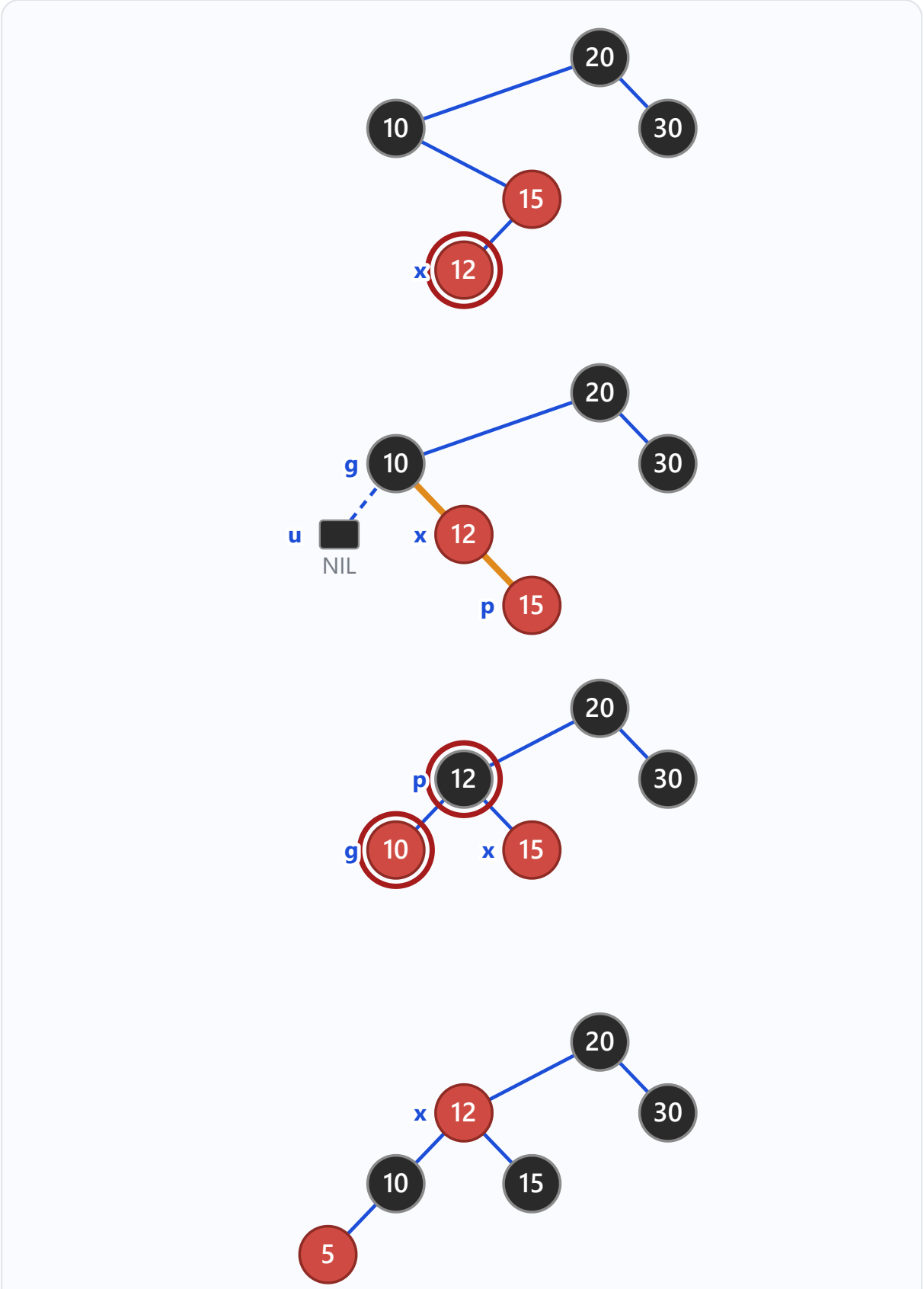
Aynı ağaca, 20S(10S(—, 15K), 30S), önce 12, sonra 5 eklensin. 12 üçgen durumunu, 5 ise renk değiştirmenin bir düzey yukarı taşınmasını gösterir.

Ekleme	Konum	*p*, *g*, *u*	Yapılan	Sonuç
12	15'in sol çocuğu	15K, 10S, NIL	amca siyah, üçgen: `dondur(15, 12)` sağa; roller değişir (x = 15, p = 12); `dondur(10, 12)` sola; 12 siyah, 10 kırmızı	20S(12S(10K, 15K), 30S)
5	10'un sol çocuğu	10K, 12S, 15K	amca kırmızı: 10 ve 15 siyah, 12 kırmızı; *x = 12*; ebeveyn 20 siyah → döngü biter	20S(12K(10S(5K, —), 15S), 30S)

Tablo 10.20: 12 ve 5 eklenirken `duzelt` izlemesi.

12 eklenirken 15, 10'un *sağ* çocuğudur; 12 ise 15'in *sol* çocuğudur. Yol zikzak çizer. Önce 15'te sağa döndürülür: 12 yukarı, 15 aşağı geçer ve 10 → 12 → 15 düz çizgisi oluşur; ama art arda iki kırmızı hâlâ vardır. Roller değiştirildikten sonra 10'da sola döndürülür. 12 alt ağacın köküne çıkar ve siyah olur, 10 kırmızı olur. Bu, AVL'deki RL düzeltmesinin aynısıdır.

5 eklenirken amca 15 kırmızıdır. Renk değiştirme 12'yi kırmızı yapar ve döngü 12'den devam eder. 12'nin ebeveyni 20 siyah olduğu için döngü biter; kök zaten siyahtır. Son ağaçta her yolda 2 siyah düğüm vardır. 20 → 12 → 10 → 5 → NIL yolundaki siyahlar 20 ve 10'dur. 20 → 30 → NIL yolunda da 20 ve 30 siyahtır.



Şekil 10.36: İkinci izlemeden dört an. (a) 12, 15'in sol çocuğu olarak eklendi; 15 de kırmızı. (b) 15'te sağa döndürme: üçgen düz çizgiye döndü. (c) 10'da sola döndürme ve renkler: 12 siyah, 10 ve 15 kırmızı. (d) 5 eklendi,

renk değişti; kurallar sağlanıyor: 20S(12K(10S(5K, —), 15S), 30S).

Örnek 10.7

Boş bir kırmızı-siyah ağaca 1'den 10'a kadar sayılar sırayla eklenirse ne olur? Kod çalıştırıldığında beş renk değişimi ve beş döndürme yapılır. Sonuç ağacı 4S(2S(1S, 3S), 6S(5S, 8K(7S, 9S(—, 10K)))')dur. Yükseklik 4, siyah yükseklik 3'tür. Aynı girdi düz BST'de yüksekliği 9 olan bir zincir verirdi. AVL ağacı ise aynı girdiden yüksekliği 3 olan bir ağaç kurar. Kırmızı-siyah ağaç AVL'den bir düzey uzun kaldı; ama ikisi de logaritmik yüksekliktedir.

10.5.7 Dengeli ağaçlarda silme

AVL ve kırmızı-siyah ağaçlarda silme aynı biçimde başlar: önce düz BST silmesi uygulanır. İki çocuklu düğüm ardılı ya da öncülüyle değiştirilir; gerçekte ağaçtan kopan düğüm her zaman bir yaprak ya da tek çocuklu bir düğümdür. Düğümün kopması dengeyi bozabilir: AVL'de yükseklik farkı, kırmızı-siyah ağaçta siyah yükseklik kuralı ihlal edilebilir.

- **AVL düzeltmesi:** Kopan düğümden köke doğru yükseklikler güncellenir. *|bf| = 2* olan her düğümde LL, RR, LR ya da RL döndürmesi uygulanır. Eklemenin tersine, bir döndürme alt ağacın yüksekliğini eski değerinin altına indirebilir; bu da üstteki bir atayı dengesizleştirebilir. Bu yüzden birden fazla atada düzeltme gerekebilir.
- **Kırmızı-siyah düzeltmesi:** Kırmızı bir düğümün silinmesi kolaydır; siyah sayıları değişmez. Siyah bir düğüm kopunca onun yolları bir siyah eksik kalır. Bu eksiklik, kardeşin rengine ve kardeşin çocuklarının renklerine göre yeniden renklendirme ve döndürmeyle giderilir. Bu düzeltmeye kitaplarda “çift siyah” düzeltmesi denir; eksik siyahlık köke kadar taşınabilir, ama döndürme sayısı en çok üçtür.

Yapı	Yukarı doğru inceleme	Döndürme	Toplam
AVL	*O(log n)*	*O(log n)* olabilir	*O(log n)*
Kırmızı-siyah	*O(log n)*	en çok 3	*O(log n)*

Tablo 10.21: Dengeli ağaçlarda silme maliyeti.

Ekleme ve silme BST sırasını korur. Dengeleme yalnız ağacın biçimini ve düğümlerdeki yardımcı bilgileri (yükseklik ya da renk) değiştirir; kök ortada gezinme her an aynı sıralı listeyi verir.

10.5.8 AVL mi, kırmızı-siyah mı?

İki yapı da bütün işlemleri *O(log n)*'de yapar. Seçim, işlemlerin karışımına bağlıdır. Tablo 10.22 farkları özetler.

Özellik	AVL	Kırmızı-siyah
Denge ölçütü	her düğümde $\lvert bf \rvert \le 1$	renk kuralları, eşit siyah yükseklik
En büyük yükseklik	$\approx 1{,}44 \log_2 n$	$2 \log_2(n + 1)$
Arama	$O(\log n)$, biraz daha hızlı (ağaç daha kısa)	$O(\log n)$
Ekleme döndürmesi	en çok 2 (bir tek ya da bir çift)	en çok 2
Silme döndürmesi	$O(\log n)$ olabilir	en çok 3
Düğümde ek bilgi	yükseklik (ya da denge faktörü)	1 bit renk
Tercih	aramanın baskın olduğu işler	sık ekleme ve silme, kütüphaneler

Tablo 10.22: AVL ve kırmızı-siyah ağacın karşılaştırması.

Uygulamada fark çoğu zaman küçüktür. Bir sözlük bir kez kurulup milyonlarca kez sorgulanacaksa AVL'nin daha kısa ağacı her aramada birkaç karşılaştırma kazandırır. Ekleme ve silmenin sık olduğu genel amaçlı bir kütüphane sınıfında kırmızı-siyah ağacın ucuz güncellemesi ağır basar. Java ve C++ standart kütüphanelerinin kırmızı-siyah ağacı seçmesinin nedeni budur.

10.6 Splay ağaçları

AVL ve kırmızı-siyah ağaçlar her düğümde ek bilgi tutar ve her işlemden sonra dengeyi bu bilgiye bakarak düzeltir. Splay ağacı başka bir yol izler: hiçbir denge bilgisi tutmaz ve ağacın dengeli olmasına da çalışmaz. Bunun yerine her erişimde erişilen düğümü köke taşır. Yöntemi 1985'te Sleator ve Tarjan önerdi; yapıya kendini düzenleyen (*self-adjusting*) BST de denir.

10.6.1 Kendini düzenleyen ağaç

> **Tanım 10.7**
>
> **Splay ağacı** (*splay tree*), her erişimden (arama, ekleme, silme) sonra erişilen düğümü bir dizi döndürmeyle köke taşıyan ikili arama ağacıdır. Bu köke taşıma işlemine **splay** işlemi denir. Düğümlerde yükseklik ya da renk gibi denge bilgisi tutulmaz.

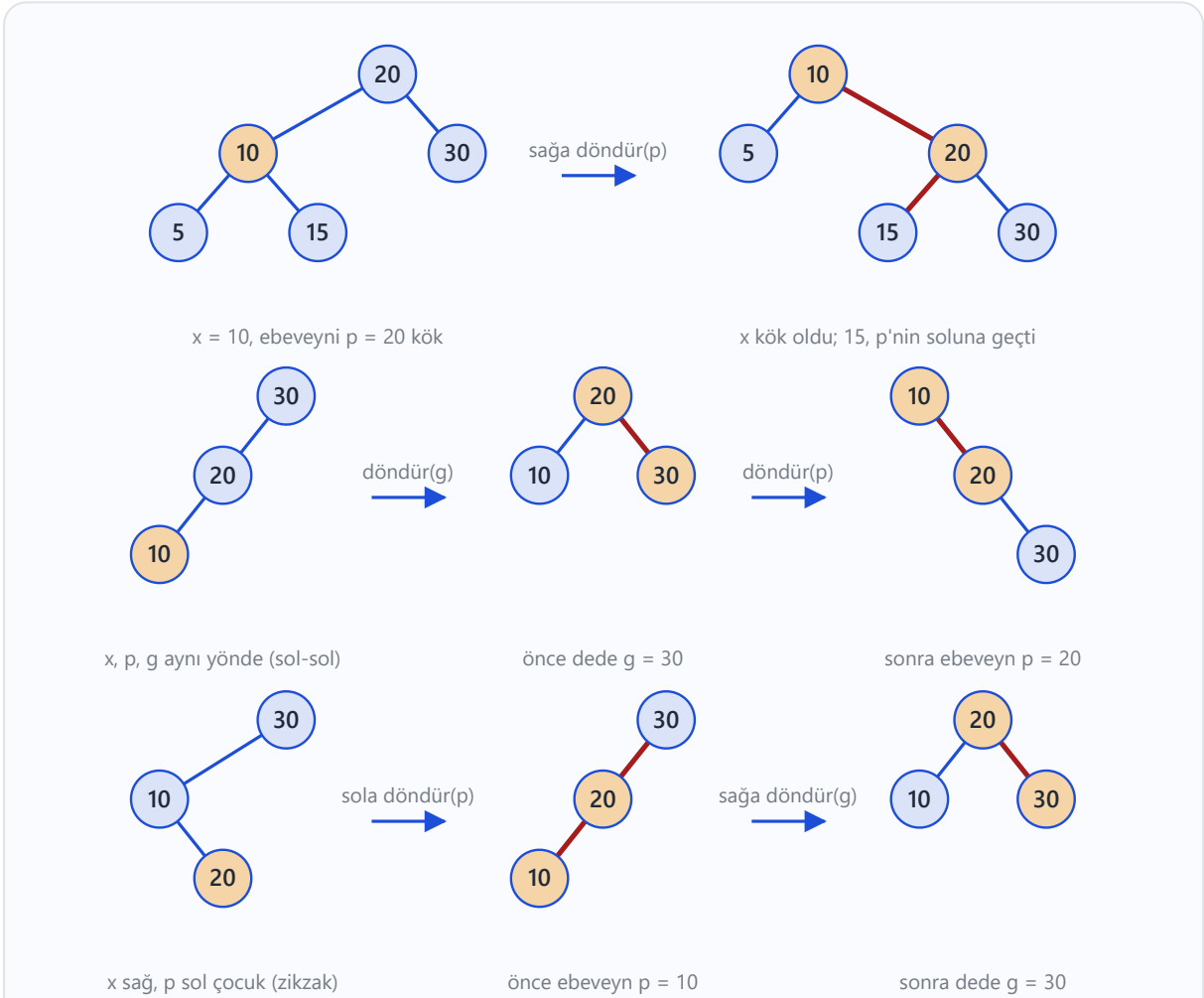
Splay ağacının yüksekliği için hiçbir garanti yoktur; ağaç bir zincire bile dönüşebilir. Böyle bir ağaçta en alttaki düğüme erişmek $O(n)$ sürer. Ama bu pahalı erişim ağacı da düzeltir: erişilen düğüm köke çıkarken yolu üzerindeki düğümlerin derinliği kabaca yarıya iner. Sleator ve Tarjan, boş ağaçtan başlayan m işlemlik her dizinin toplam $O(m \log n)$ sürdüğünü gösterdi. İşlem başına **amortize maliyet** $O(\log n)$'dir. Amortize analiz, tek tek işlemlere değil, işlem dizisinin toplamına söz verir: bazı işlemler pahalı olabilir, ama ucuz işlemler onların maliyetini dengeler.

Yapının üç belirgin özelliği vardır:

- **Yerellik:** Sık erişilen anahtarlar köke yakın durur. Ağaç bir önbellek gibi davranır. Erişimlerin küçük bir anahtar kümesinde yoğunlaştığı uygulamalarda splay ağacı dengeli ağaçlardan hızlı olabilir.
- **Basitlik:** Ek alan gerekmez. Tek bir `splay` yordamı bütün işlemlere yeter: arama bulduğu düğümü, ekleme yeni düğümü, silme ise silinecek düğümü önce köke taşır.
- **Dezavantaj:** Arama bile ağacı değiştirir. Bu, birden çok iş parçacığının aynı ağacı okuduğu durumlarda sorun çıkarır. Tek bir işlemin süresi için garanti yoktur; gerçek zamanlı sistemlerde bu kabul edilemeyebilir.

10.6.2 Zig, zig-zig ve zig-zag

Splay, erişilen düğüm x kök olana kadar üç tür adımı tekrarlar. Adımın türünü x'in ebeveyni p ve dedesi g belirler.



Şekil 10.37: Splay adımları. Üstte zig: $x = 10$, ebeveyni $p = 20$ kök; tek döndürme x'i kök yapar ve 15, p'nin soluna geçer. Ortada zig-zig: x, p, g aynı yönde (sol-sol); önce dede (30), sonra ebeveyn (20) döndürülür. Altta zig-zag: x sağ, p sol çocuk; önce ebeveyn (10), sonra dede (30) döndürülür.

- **Zig:** *x*'in ebeveyni köktür (dede yoktur). *p*'de tek bir döndürme yapılır ve *x* kök olur. Bu adım yalnız splay'in son adımında görülebilir.
- **Zig-zig:** *x* ve *p* aynı yönde çocuktur (ikisi de sol ya da ikisi de sağ). Önce *dedede*, sonra ebeveynde döndürülür.
- **Zig-zag:** *x* ve *p* zıt yönde çocuktur. Önce ebeveynde, sonra dedede döndürülür. Bu, AVL'deki çift döndürmenin aynısıdır.

Zig-zig'deki sıra önemlidir ve AVL'den farklıdır. Önce dedede döndürmek, yoldaki düğümleri "katlar": zincirin üst kısmı da kısalır. Önce ebeveynde döndürmek (yani *x*'i tek döndürmelerle yukarı itmek) *x*'i yine köke çıkarır, ama zincirin geri kalanı uzun kalabilir. Sleator ve Tarjan'ın amortize sınırı tam olarak bu dededen başlayan sıraya dayanır.

> **Sık yapılan hata**
>
> Zig-zig adımında önce ebeveyni döndürmek. Bu, düğümü köke çıkaran "basit köke taşıma"dır ve amortize *O(log n)* garantisini vermez. Örneğin sıralı bir zincirde en alttaki düğüme tekrar tekrar erişildiğinde zincir uzun kalır ve her erişim *O(n)* sürebilir. Doğru sıra: önce dede *g*, sonra ebeveyn *p*.

Sunumdaki splay kodu bu üç adımı bir döngüde toplar:

```
void splay(Dugum x) {
    while (x != kok) {
        Dugum p = x.ebeveyn, g = p.ebeveyn;
        if (g == null)                // zig
            dondur(p, x);
        else if (ayniYon(x, p, g)) {  // zig-zig
            dondur(g, p);   // önce dede
            dondur(p, x);
        } else {                      // zig-zag
            dondur(p, x);   // önce ebeveyn
            dondur(g, x);
        }
    }
}
```

`dondur(a, b)`, kırmızı-siyah koddaki tek döndürmenin aynısıdır: *a*'nın çocuğu olan *b*'yi *a*'nın yerine çıkarır. `ayniYon(x, p, g)`, *x*'in *p*'nin hangi tarafında olduğunu, *p*'nin *g*'nin hangi tarafında olduğuyla karşılaştırır. Zig-zag dalındaki ikinci çağrı `dondur(g, x)` biçimindedir: ilk döndürmeden sonra *x*, *p*'nin yerine geçmiştir ve artık *g*'nin çocuğudur.

10.6.3 Kod izleme: splay(50) ve splay(10)

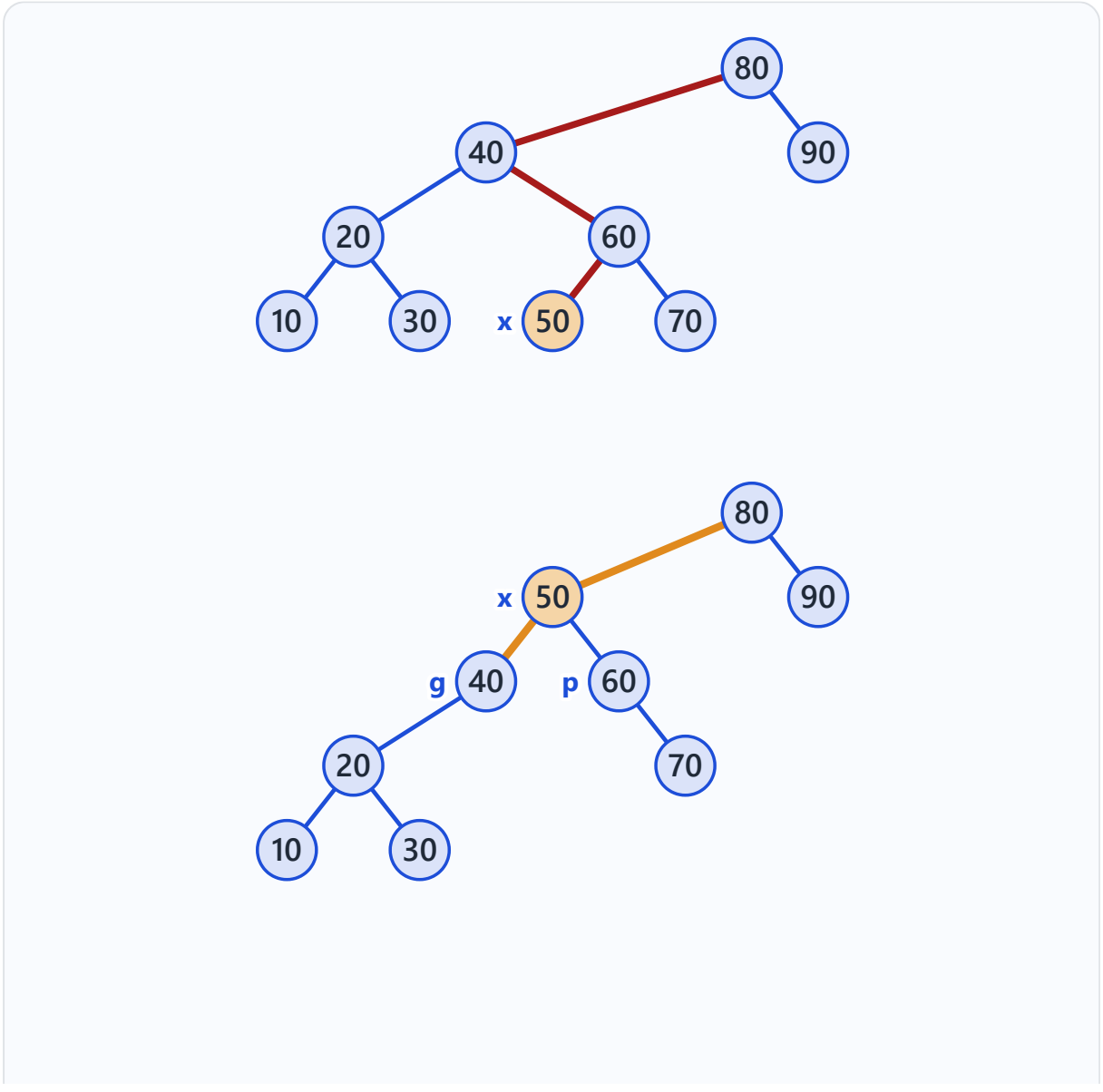
Başlangıç ağacı 80(40(20(10, 30), 60(50, 70)), 90)'dır. Önce 50'ye, sonra 10'a erişilsin.

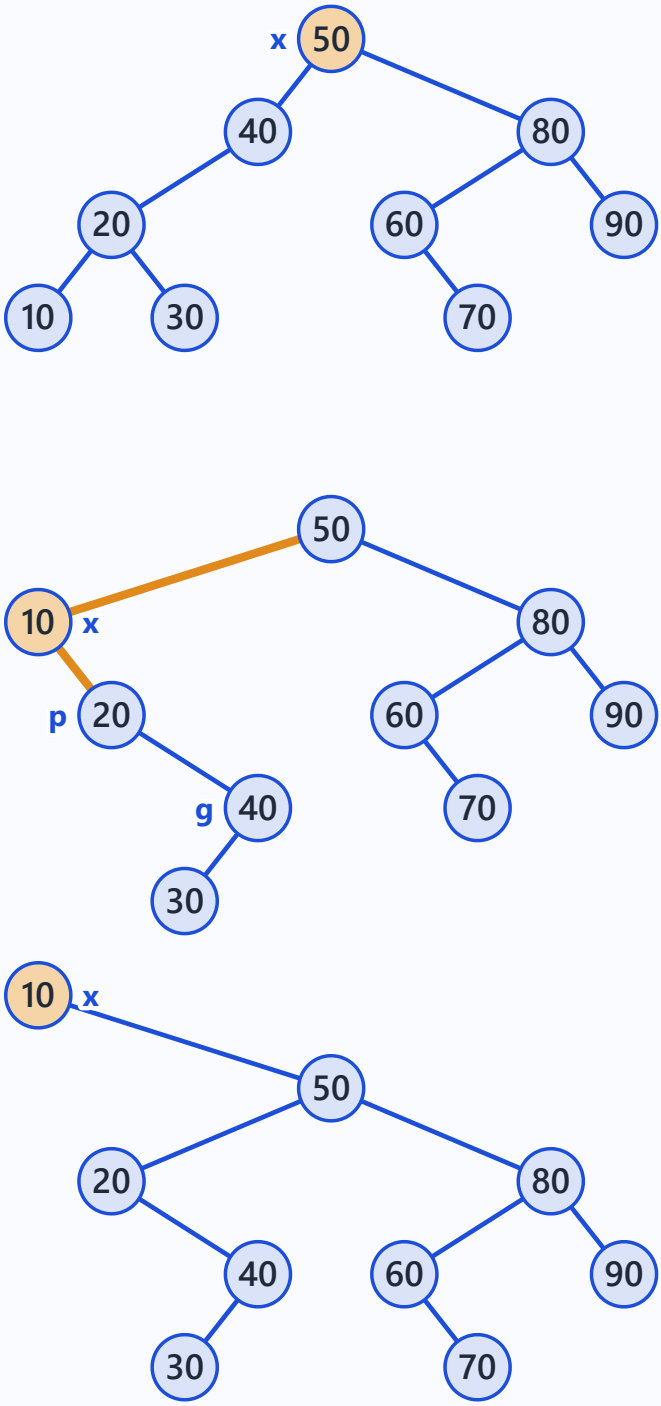
Erişim	Tur	p, g	Adım	Döndürmeler
50 (yol 80 → 40 → 60 → 50)	1	60, 40	zig-zag (50 sol, 60 sağ çocuk)	60'ta sağa, sonra 40'ta sola
	2	80, yok	zig	80'de sağa; 50 kök
10 (yol 50 → 40 → 20 → 10)	1	20, 40	zig-zig (ikisi de sol çocuk)	önce 40'ta sağa, sonra 20'de sağa
	2	50, yok	zig	50'de sağa; 10 kök

Tablo 10.23: `splay(50)` ve ardından `splay(10)` izlemesi.

`splay(50)` 'den sonra ağaç 50(40(20(10, 30), —), 80(60(—, 70), 90))'dır. 50 derinlik 3'ten köke çıktı. Eski kök 80 bir düzey indi ve 90 da onunla birlikte 1'den 2'ye geçti. 40, 20, 60 ve 70'in derinlikleri değişmedi.

`splay(10)` 'dan sonra ağaç 10(—, 50(20(—, 40(30, —)), 80(60(—, 70), 90)))'dır. 10 köke çıktı ve az önce erişilen 50 hâlâ köke yakındır: derinliği 1'dir. Zig-zig adımında önce 40'ta döndürüldüğü için 40, 20'nin altına inmiş ve 30'u sol çocuğu olarak almıştır.





Şekil 10.38: Splay izlemesi. (a) `splay(50)` : BST araması 80 → 40 → 60 → 50. (b) Zig-zag bitti: 50, 40 ile 60'ın ebeveyni oldu. (c) Zig'den sonra 50 kök. (d) `splay(10)` 'da zig-zig: önce 40'ta, sonra 20'de sağa döndürüldü; 10 iki düzey yükseldi. (e) Son zig'den sonra 10 kök; 50'nin derinliği 1.

Tek bir splay işleminin maliyeti, erişilen düğümün derinliğiyle orantılıdır; derin bir düğüm için *O(n)* olabilir. Ama ağaç bu işlemlerle sürekli yeniden şekillenir ve uzun bir işlem dizisinin ortalaması *O(log n)*'de kalır.

10.7 Çok yollu arama ağaçları

Buraya kadar bütün ağaçlar ikiliydi: her düğümde bir anahtar ve en çok iki çocuk. Veri ana bellekte durduğunda bu yeterlidir; bir düğümden ötekine geçmek birkaç nanosaniye sürer. Veri diskte durduğunda durum değişir. Diskten bir blok okumak, bellekten bir değer okumaktan binlerce, eski dönen disklerde yüz binlerce kat yavaştır. Bir milyar kayıtlı bir veritabanı indisinde ikili ağacın yüksekliği en iyi durumda 30'dur. Her düğüm ayrı bir disk okuması isterse tek bir arama 30 disk erişimi tutar. Çok yollu ağaçlar bu sayıyı birkaç erişime indirir.

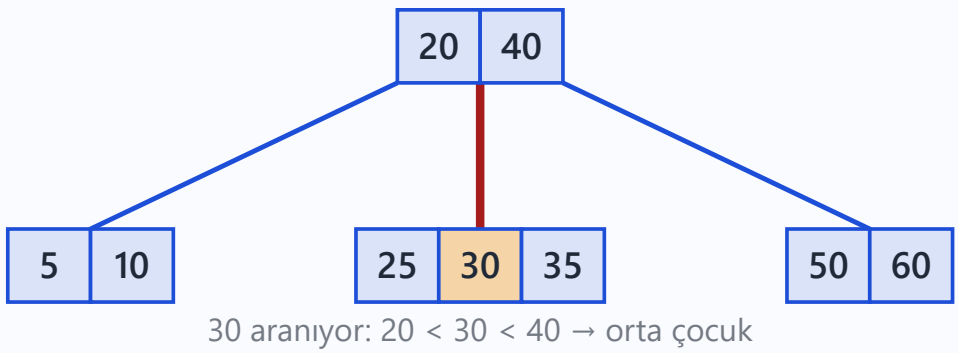
10.7.1 B-ağacı

Tanım 10.8

B-ağacı (*B-tree*), bir düğümde birden çok anahtar ve çok sayıda çocuk bulunan dengeli bir çok yollu arama ağacıdır. **Derecesi** m olan B-ağacında:

- Bir düğüm en çok m çocuk ve $m - 1$ anahtar tutar.
- Kök dışındaki iç düğümlerde en az $\lceil m/2 \rceil$ çocuk bulunur.
- Düğüm içindeki anahtarlar sıralıdır. k anahtarlı bir iç düğümün $k + 1$ çocuğu vardır. Ardışık iki anahtar arasındaki çocukta, değeri bu iki anahtarın arasında kalan anahtarlar bulunur.
- Bütün yapraklar aynı derinliktedir.

Düğüm içindeki anahtarlar, çocukların arasındaki sınırlar gibi çalışır. Şekil 10.39'daki kök [20, 40] üç çocuğa yön verir: 20'den küçükler sol çocukta, 20 ile 40 arasındakiler ortada, 40'tan büyükler sağdadır. Bu, BST özelliğinin çok yollu genellemesidir. İkili arama ağacı, derecesi 2 olan bir B-ağacı gibi düşünülebilir; ama B-ağacının asıl gücü, bütün yaprakların aynı derinlikte olması kuralından gelir. Bu kural ağacı her zaman mükemmel biçimde dengeli tutar.



Şekil 10.39: Derecesi 4 olan bir B-ağacı: kök [20, 40], yapraklar [5, 10], [25, 30, 35], [50, 60]. 30 aranırken kökte 20 < 30 < 40 olduğu için orta çocuğa inilir.

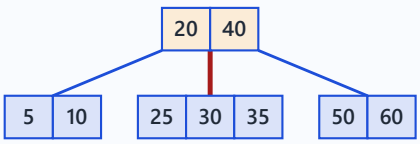
Neden böyle bir yapı? Bir düğüm bir disk bloğuna denk getirilir. Bir blokta yüzlerce anahtar tutulabilir; derecesi örneğin 200 olan bir B-ağacında her düğüm 200'e kadar çocuğa yön verir. Yüksekliği 3 olan böyle bir ağaç 200^4, yani bir milyardan fazla anahtar tutabilir. Bir arama dört blok okur; kök genellikle bellekte tutulduğu için diske gidilen sayı daha da azdır. Geniş ve alçak ağaç az disk erişimi demektir. Bu yüzden veritabanı ve dosya sistemi indislerinin temeli B-ağacıdır. Yapıyı 1972'de Bayer ve McCreight tanıttı.

Derecesi 4 olan B-ağacına 2-3-4 ağacı da denir; düğümleri 2, 3 ya da 4 çocukludur. 2-3-4 ağacı kırmızı-siyah ağacın eşdeğeridir. Kırmızı-siyah ağaçta her siyah düğüm kırmızı çocuklarıyla tek bir düğümde toplanırsa bir 2-3-4 ağacı elde edilir. Kırmızı-siyah ağacın “eşit siyah yükseklik” kuralı, 2-3-4 ağacının “bütün yapraklar aynı derinlikte” kuralının ta kendisidir.

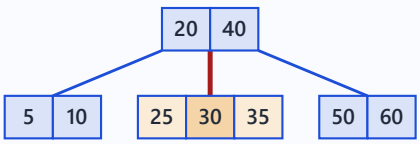
10.7.2 B-ağacında arama ve ekleme

Arama BST aramasının genellemesidir. Kökte, düğümün anahtarları sırayla aranan anahtarla karşılaştırılır. Anahtar bulunursa arama biter. Bulunamazsa, aranan anahtarın düştüğü aralığa karşılık gelen çocuğa inilir. Her düzeyde yalnız bir düğüm, yani bir disk bloğu okunur. Maliyet yükseklikle orantılıdır: $O(\log_m n)$ düğüm okuması. Düğüm içindeki arama da ikili aramayla $O(\log m)$ karşılaştırmada yapılabilir; toplam karşılaştırma sayısı $O(\log n)$'dir.

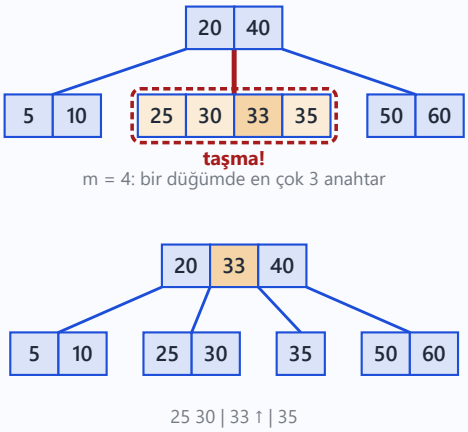
Ekleme her zaman bir yaprakta başlar. Anahtar için aynı arama yolu izlenir ve anahtar yaprakta sıralı yerine konur. Yaprakta yer varsa iş biter. Yaprak taşarsa, yani m anahtara ulaşırsa, **bölünür**: ortanca anahtar ebeveyne çıkar, kalan anahtarlar iki ayrı düğüme ayrılır. Ebeveyne bir anahtar ve bir çocuk eklendiği için ebeveyn de taşabilir; o zaman o da bölünür. Bölünme köke kadar yayılabilir. Kök bölünürse yeni bir kök oluşur ve ağacın yüksekliği bir artar.



kök: 20 < 30 < 40 → orta çocuğa in



yaprak: 25, 30 → bulundu (2 düğüm okundu)



Şekil 10.40: Derecesi 4 olan B-ağacında arama ve ekleme. (a) 30 aranıyor: 20 < 30 < 40, orta çocuğa inilir. (b) 30 yaprakta bulunur; her düzeyde bir düğüm okundu. (c) 33 aynı yoldan inip yaprağa sıralı yerine konur; yaprakta 4 anahtar oldu, kapasite ($m - 1 = 3$) aşıldı. (d) Yaprak bölünür: ortadaki 33 ebeveyne çıkar, kalanlar [25, 30] ve [35] olur. Kök artık [20, 33, 40].

Dört anahtarlı [25, 30, 33, 35] yaprağının tam bir ortancası yoktur; iki ortadaki anahtardan biri seçilmelidir. Sunumdaki örnek sağdakini (33) seçer ve sol yarıda iki, sağ yarıda bir anahtar kalır. Soldakini seçmek de geçerlidir; önemli olan seçimin tutarlı olmasıdır. Derecesi tek olan ağaçlarda bu belirsizlik doğmaz: $m = 3$'te taşan düğümde üç anahtar vardır ve ortanca tektir.

Bölünmenin güzel bir sonucu vardır: B-ağacı **aşağıdan yukarı** büyür. İkili arama ağacında yeni düğümler aşağıya eklenir ve ağaç aşağı doğru uzar. B-ağacında ise yükseklik ancak kök bölünürken, yani en üstte artar. Bütün yapraklar aynı anda bir düzey aşağı itilmiş olur; bu yüzden hepsi her zaman aynı derinlikte kalır.

Silme eklemenin tersidir. Bir düğüm en az doluluk sınırının altına düşerse kardeşinden anahtar ödünç alır ya da kardeşiyle birleşir. Birleşme ebeveynden bir anahtar eksiltir ve eksiklik yukarı taşınabilir. Kökün son anahtarı inerse kök kaldırılır ve yükseklik bir azalır. Böylece hem denge hem doluluk sınırları korunur.

10.7.3 Ekleme ve bölünme adım adım

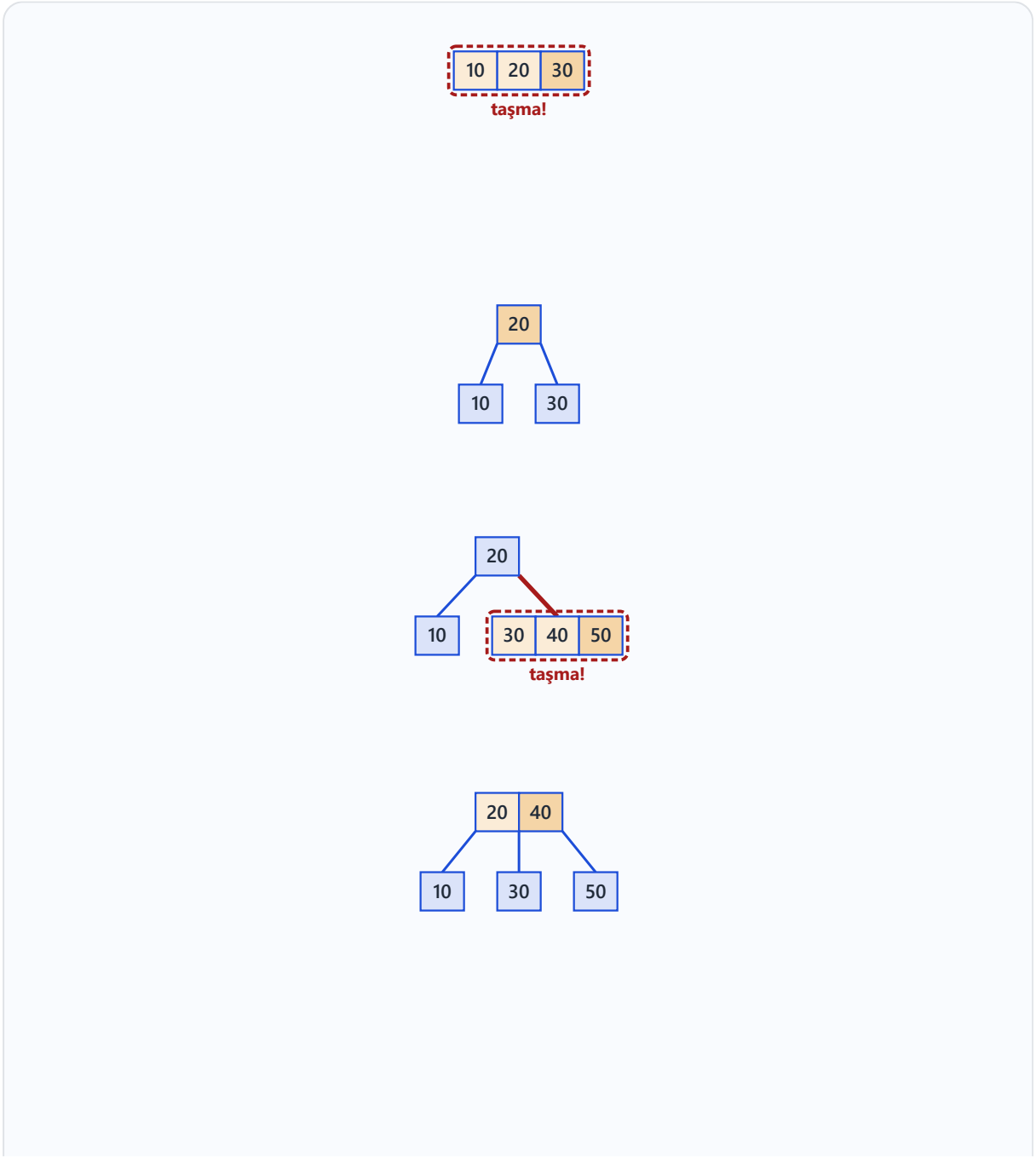
Derecesi $m = 3$ olan boş bir B-ağacına sırasıyla 10, 20, 30, 40, 50, 60, 70 eklensin. Bu ağaçta bir düğüm en çok 2 anahtar ve 3 çocuk tutar; 3 anahtara ulaşan düğüm bölünür.

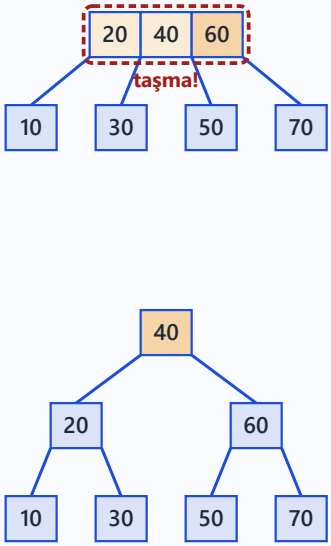
Ekleme	Olay	Sonuç
10	kök yaprağa eklenir; 1 anahtar	[10]
20	kök yaprağa eklenir; 2 anahtar	[10, 20]
30	[10, 20, 30] taşar; ortanca 20 yeni köke çıkar (kök bölünmesi, yükseklik 1)	[20]{[10] [30]}
40	sağ yaprağa eklenir	[20]{[10] [30, 40]}
50	[30, 40, 50] taşar; 40 ebeveyne çıkar; ebeveynde yer var	[20, 40]{[10] [30] [50]}

Ekleme	Olay	Sonuç
60	sağ yaprağa eklenir	[20, 40]{[10] [30] [50, 60]}
70	[50, 60, 70] taşar; 60 ebeveyne çıkar; ebeveyn [20, 40, 60] de taşar; 40 yeni köke çıkar (kök bölünmesi, yükseklik 2)	[40]{[20]{[10] [30]} [60] {[50] [70]}}

Tablo 10.24: *m = 3* olan B-ağacına 10, 20, 30, 40, 50, 60, 70 eklenmesi. Ağaç "kök{çocuklar}" biçiminde yazılmıştır.

Sonuçta 7 anahtar 3 düzeye yerleşti. Her iç düğümün 2 çocuğu vardır ve bütün yapraklar aynı derinliktedir. Kök iki kez bölündü ve yükseklik yalnız bu iki anda arttı. Sıralı girdi düz BST'yi zincire çevirirdi; B-ağacı ise aynı girdiden mükemmel dengeli bir ağaç kurdu.





Şekil 10.41: Bölünmeler. (a) 30 eklenince düğüm [10, 20, 30] oldu, kapasite aşıldı. (b) Kök bölündü: 20 yeni kök. (c) 50 eklenince [30, 40, 50] taştı. (d) 40 ebeveyne çıktı; bölünme orada durdu. (e) 70 eklenince 60 ebeveyne çıktı, ama ebeveyn [20, 40, 60] taştı. (f) Kök bölündü: 40 yeni kök; yükseklik bir arttı ve bütün yapraklar yine aynı derinlikte.

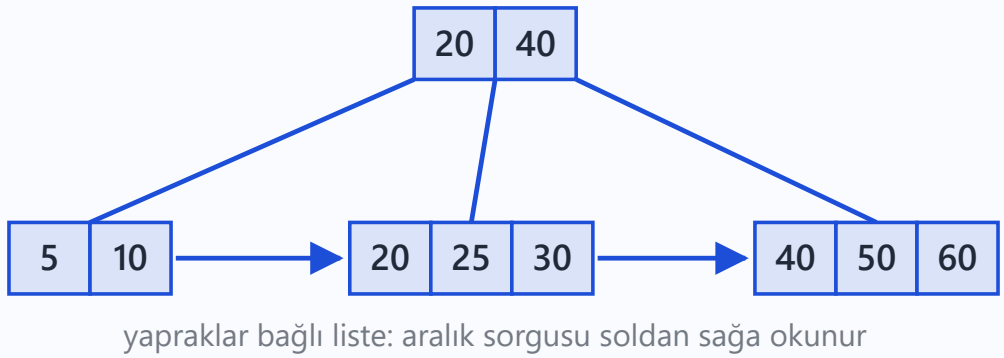
Sık yapılan hata

Taşan yaprağın altına yeni bir yaprak eklemeye çalışmak. B-ağacında yeni düğüm aşağıya değil, yana açılır: bölünme taşan düğümü iki kardeşe ayırır ve ortanca anahtarı yukarı gönderir. Ağaç ancak kök bölünürken uzar. Yaprakların altına düğüm eklemek "bütün yapraklar aynı derinlikte" kuralını bozar.

10.7.4 B+ ağacı

B+ ağacı, B-ağacının veritabanlarında en çok kullanılan çeşididir. İki yapı arasında üç fark vardır:

- **Veri yapraklarda:** Bütün anahtarlar ve kayıtlar yapraklarda bulunur. İç düğümler yalnız yönlendirme anahtarları tutar; bu anahtarların kopyası yapraklarda da vardır.
- **Bağlı yapraklar:** Yapraklar soldan sağa bir bağlı liste gibi birbirine bağlıdır. "20 ile 50 arası" gibi aralık sorguları bu zincir üzerinde hızla yürür.
- **Daha geniş iç düğümler:** İç düğümler kayıt taşımadığı için bir bloğa daha çok anahtar sığar. Derece büyür, ağaç alçalır.



Şekil 10.42: B+ ağacı: kökte 20 ve 40 yönlendirme anahtarları; yapraklarda [5, 10], [20, 25, 30], [40, 50, 60]. Yapraklar soldan sağa bağlıdır; aralık sorgusu bu zinciri izler. 20 ve 40 hem kökte hem yapraklarda bulunur.

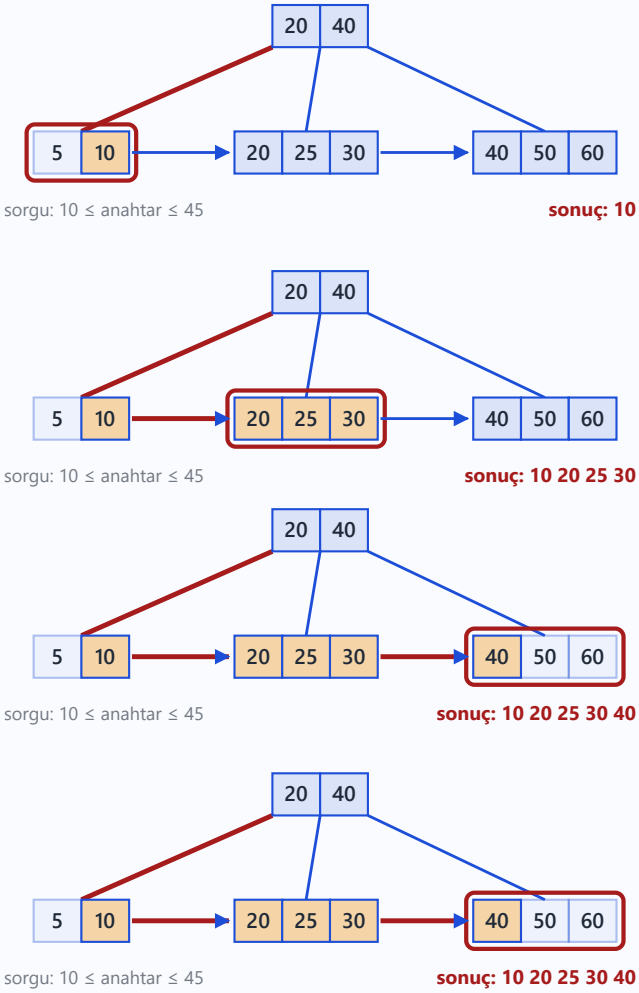
Arama her zaman bir yaprakta biter, çünkü kayıt yalnız orada vardır. B-ağacında aranan anahtar bir iç düğümde bulunursa arama erken biterdi; B+ ağacında bu kısa yol yoktur. Buna karşılık her arama aynı sayıda düzey iner ve maliyet tahmin edilebilir. Çoğu ilişkisel veritabanı yönetim sisteminde tablo indisleri B+ ağacıdır.

10.7.5 B+ ağacında aralık sorgusu

Şekil 10.42'deki ağaçta 10 ile 45 arasındaki anahtarlar istensin.

1. Kökten başlanır: 10 < 20 olduğu için alt sınır 10'u içeren yaprak en soldaki çocuktadır. Bu iniş bir aramadır ve *O(log n)* düzey sürer.
2. İlk yaprak [5, 10]: 5 < 10 olduğu için atlanır, 10 alınır. Yaprağın sonuna gelindi; yukarı çıkılmadan sonraki yaprak bağına geçilir.
3. İkinci yaprak [20, 25, 30]: üçü de aralıkta, alınır. Yine bağ izlenir.
4. Üçüncü yaprak [40, 50, 60]: 40 alınır; 50 > 45 olduğu için durulur.

Sonuç 10 20 25 30 40'tır ve sıralıdır. Maliyet, bir kez aşağı inmek (*O(log n)*) ile okunan yaprakların toplamıdır: *O(log n + k)*, burada *k* sonuç sayısıdır. B-ağacında aynı sorgu, anahtarlar iç düğümlere de dağıldığı için ağaçta tekrar tekrar inip çıkmayı gerektirirdi. B+ ağacındaki yaprak zinciri bu gidiş gelişi ortadan kaldırır.



Şekil 10.43: B+ ağacında 10 ile 45 arası aralık sorgusu. (a) İlk yaprak: 10 alındı. (b) İkinci yaprak: 20, 25, 30 alındı. (c) Üçüncü yaprak: 40 alındı, 50 > 45 olduğu için duruldu. (d) Sonuç 10 20 25 30 40; maliyet $O(\log n + k)$.

Örnek 10.8

Bir öğrenci kayıt sisteminde 2 milyon kayıt, öğrenci numarasına göre derecesi 100 olan bir B+ ağacında tutulsun. İç düğümler en az yarı dolu olduğu için her iç düğüm en az 50 çocuğa yön verir. Yükseklik en çok $\lceil \log_{50}(2 \cdot 10^6) \rceil$ civarındadır: $50^3 = 125.000$, $50^4 = 6.250.000$; yani en çok 4 düzey. Bir öğrenciyi bulmak en çok 4 blok okur. Numarası 2020001 ile 2020150 arasındaki 150 öğrenciyi listelemek ise bir iniş ve birkaç yaprak okumasıyla yapılır. Aynı kayıtlar dengeli bir ikili ağaçta tutulsaydı yükseklik en az 20 olurdu.

10.8 Sık yapılan hatalar

Arama ağacı kodundaki hataların çoğu birkaç tipe ayrılır. Not boyunca konuların yanında verilen hatalar burada bir arada toplanmıştır; bir ağaç kodunu gözden geçirirken kontrol listesi olarak kullanılabilir.

Hata	Ne olur?	Doğrusu
Dönüş değerini atamamak	`sil(kok, 40);` kök silinince ya da boş ağaca eklenince ağaç bozulur.	`kok = sil(kok, 40);` ve `kok = ekle(kok, 5);`
Yalnız çocuklara bakmak	Ebeveyn–çocuk denetimi BST olmayan ağaçları geçirir.	İzinli (min, max) aralığı kökten aşağı taşınır (`bstMi`).
Nesneleri `==` ile karşılaştırmak	Referanslar karşılaştırılır; ağaçta olan anahtar bulunamaz.	Sıra için `compareTo` , eşitlik için `equals` ya da `compareTo(...) == 0` .
Yükseklik güncelleme sırası	AVL döndürmesinden sonra yanlış denge faktörleri, gereksiz döndürmeler.	Önce aşağı inen düğüm, sonra yeni kök güncellenir.
"BST her zaman *O(log n)*"	Sıralı girdide $h = n - 1$, işlemler $O(n)$.	$O(\log n)$ yalnız dengeli ağaçlarda (AVL, kırmızı-siyah) garantidir.
`null` çocuğu unutmak	`NullPointerException` ya da yanlış denge ve renk kararları.	Boş alt ağacın yüksekliği −1, kırmızı-siyahta NIL siyahtır; `h(null)` , `kirmizi(null)` gibi yardımcılar bu yüzden vardır.
Çift döndürme yerine tek döndürme	LR ya da RL durumunda ağaç ters yöne eğilir ve dengesiz kalır.	Çocuğun denge faktörüne bakılır; zikzakta önce çocukta döndürülür.
Zig-zig sırası	Splay'in amortize garantisi kaybolur.	Önce dede, sonra ebeveyn döndürülür.

Tablo 10.25: Arama ağacı kodunda sık görülen hatalar.

Bu hataların arkasında iki ortak düşünce yanlışı vardır. Birincisi, ağacı yalnız "yerel" görmektir: bir düğümü yalnız ebeveyni ve çocuklarıyla düşünmek. Oysa BST özelliği, AVL koşulu ve siyah yükseklik kuralı bütün alt ağaçla ilgili koşullardır. İkincisi, Java'nın referans ve değer ayrımını unutmaktır. Ağaç metotları alt ağacın yeni kökünü döndürür ve çağıran bu değeri bir bağa ya da değişkene yazmadıkça hiçbir şey değişmez.

10.9 Özet

- **Arama ağacı**, öğeleri anahtarlarına göre sıralı tutar; arama, ekleme ve silme kökten tek bir yol izler ve $O(h)$ sürer. n düğümlü ağaçta h, en iyi durumda yaklaşık $\log_2 n$, en kötü durumda $n - 1$'dir.
- **BST özelliği**: her düğümde sol alt ağacın bütün anahtarları küçük, sağ alt ağacın bütün anahtarları büyüktür. Özellik bütün alt ağaç için geçerlidir. **Kök ortada gezinme** anahtarları sıralı verir ve $O(n)$ sürer. En küçük anahtar en soldaki, en büyük en sağdaki düğümdedir.
- **Ekleme** aranıyormuş gibi iner ve düştüğü boş bağa yeni bir yaprak bağlar. Metotlar alt ağacın yeni kökünü döndürür; kullanım `kok = ekle(kok, x)` biçimindedir.

- **Ardıl** *x*'ten büyük en küçük, **öncül** *x*'ten küçük en büyük anahtardır. Tavan ve taban eşitliği de kabul eder. Hepsi *O(h)*'dir.
- **Silme** üç durumludur: yaprak kaldırılır; tek çocuklu düğüm atlanır; iki çocuklu düğümün değeri ardılıyla (ya da öncülüyle) değiştirilir ve ardıl kendi yerinden silinir. Ardılın sol çocuğu olmadığı için bu ikinci silme her zaman kolay durumlardan birine düşer.
- **Aralık sorgusu** budamayla *O(h + s)* sürer. **BST doğrulama**, izinli aralığı kökten aşağı taşıyarak *O(n)*'de yapılır.
- Düz BST'nin şekli ekleme sırasına bağlıdır; sıralı girdi zincir verir. Rastgele sırada ortalama derinlik yaklaşık $1{,}39 \log_2 n$'dir. **Dengeli ağaçlar** küçük yerel değişikliklerle $h = O(\log n)$ garantisi verir.
- **AVL ağacında** her düğümde $bf = h(\text{sol}) - h(\text{sağ}) \in \{-1, 0, 1\}$'dir. En seyrek AVL ağacının düğüm sayısı Fibonacci gibi büyür ve $h < 1{,}44 \log_2(n + 2)$'dir. Dengesizlik LL ve RR'de tek, LR ve RL'de çift **döndürmeyle** düzeltilir. Döndürme *O(1)*'dir ve kök ortada sırayı korur.
- **Kırmızı-siyah ağaçta** kök siyahtır, art arda iki kırmızı yoktur ve her yolda siyah sayısı eşittir. En uzun yol en kısanın en çok iki katıdır ve $h \le 2 \log_2(n + 1)$'dir. Ekleme sonrası düzeltme amcanın rengine bakar: amca kırmızıysa renk değiştirilir, siyahsa en çok iki döndürme yapılır.
- **Splay ağacı** erişilen düğümü zig, zig-zig ve zig-zag adımlarıyla köke taşır. Denge bilgisi tutmaz; tek işlem *O(n)* olabilir, amortize maliyet *O(log n)*'dir.
- **B-ağacında** bir düğüm en çok $m - 1$ anahtar ve m çocuk tutar, bütün yapraklar aynı derinliktedir. Taşan düğüm bölünür ve ortanca anahtar yukarı çıkar; ağaç aşağıdan yukarı büyür. **B+ ağacında** veri yalnız yapraklardadır ve yapraklar bağlıdır; aralık sorgusu *O(log n + k)* sürer.

Yapı	Yükseklik	Arama / ekleme / silme	Nasıl dengelenir?
BST	*log n … n*	*O(h)*	dengelenmez; en kötü *O(n)*
AVL	≤ *1,44 log n*	*O(log n)*	$bf \in \{-1, 0, 1\}$; LL, RR, LR, RL döndürmeleri
Kırmızı-siyah	≤ *2 log(n + 1)*	*O(log n)*	renk değiştirme + en çok 2–3 döndürme
Splay	sınırsız	*O(log n)**	erişilen düğüm zig / zig-zig / zig-zag ile köke
B-ağacı / B+	≈ $\log_m n$	*O(log n)*	taşan düğüm bölünür; bütün yapraklar aynı derinlikte

Tablo 10.26: Arama ağaçlarında işlem maliyetleri. (*) Amortize; tek işlem *O(n)* olabilir. Bütün yapılarda kök ortada gezinme *O(n)*'dir ve anahtarları sıralı verir.

10.10 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

arama ağacı *search tree* · ikili arama ağacı *binary search tree (BST)* · anahtar *key* · kök *root* · yaprak *leaf* · alt ağaç *subtree* · derinlik *depth* · yükseklik *height* · kök ortada *inorder* · gezinme *traversal* · ardıl *successor* · öncül *predecessor* · ata *ancestor* · kardeş *sibling* · mükemmel ikili ağaç *perfect binary tree* · denge faktörü *balance factor* · AVL ağacı *AVL tree*

döndürme *rotation* · kırmızı-siyah ağaç *red-black tree* · splay ağacı *splay tree* · zig, zig-zig, zig-zag *zig, zig-zig, zig-zag* · amortize analiz *amortized analysis* · B-ağacı *B-tree* · B+ ağacı *B+ tree* · derece *degree*

10.11 Kendini Dene

1. Boş bir BST'ye sırasıyla 35, 18, 52, 10, 25, 44, 60, 22, 48 ekleniyor. Ağacın yüksekliği nedir? Kök ortada gezinme ne verir? `ara(kok, 48)` kaç düğümle karşılaştırma yapar?

   Cevap

   Ağaç 35(18(10, 25(22, —)), 52(44(—, 48), 60))'tır. En derin düğümler 22 ve 48'dir (derinlik 3); yükseklik 3'tür. Kök ortada gezinme 10 18 22 25 35 44 48 52 60 verir. 48 aranırken 35 (sağa), 52 (sola), 44 (sağa) ve 48 ile karşılaştırılır: 4 düğüm, yani 48'in derinliği artı bir.

2. Aynı ağaçtan önce 18, sonra 35 siliniyor (iki çocuklu düğümde kök ortada ardıl kullanılıyor). Her silmeden sonra ağaç nasıl olur?

   Cevap

   18'in iki çocuğu vardır. Ardıl, sağ alt ağacın en küçüğüdür: 25'e geçilir, sonra sola 22'ye inilir. 22, 18'in düğümüne kopyalanır ve eski 22 (yaprak) silinir: 35(22(10, 25), 52(44(—, 48), 60)). 35'in de iki çocuğu vardır; ardıl 52'den sola 44'tür. 44 köke kopyalanır, eski 44 tek çocuklu olduğu için atlanır ve 52'nin sol bağı 48'e yönelir: 44(22(10, 25), 52(48, 60)).

3. 1. sorudaki ağaçta `ardil(25)`, `oncul(44)`, `tavan(45)` ve `taban(21)` nedir?

   Cevap

   `ardil(25) = 35`: 25'in sağ alt ağacı yoktur; solundan geçilen en yakın ata köktür. `oncul(44) = 35`: 44'ün sol alt ağacı yoktur; sağından geçilen en yakın ata köktür. `tavan(45) = 48` (45 ağaçta yok, ≥ 45 olan en küçük anahtar). `taban(21) = 18` (21 ağaçta yok, ≤ 21 olan en büyük anahtar).

4. 1. sorudaki ağaçta `aralik(kok, 20, 46)` ne yazdırır? Hangi düğümler hiç ziyaret edilmez?

   Cevap

   Çıktı 22 25 35 44'tür. Ziyaret sırası 35, 18, 25, 22, 52, 44, 48'dir. 18'de `20 < 18` yanlış olduğu için 10 budanır. 52'de `52 < 46` yanlış olduğu için 60 budanır. 48 ziyaret edilir (44'ün sağı, çünkü 44 < 46), ama aralık dışındadır. Hiç ziyaret edilmeyenler 10 ve 60'tır.

5. Boş bir AVL ağacına sırasıyla 30, 20, 10, 40, 50, 35 ekleniyor. Hangi eklemelerde hangi durum oluşur ve son ağaç nedir?

Cevap

10: *bf(30) = 2*, *bf(20) = 1* → LL; 30'da sağa → 20(10, 30). 40: dengesizlik yok. 50: *bf(30) = −2*, *bf(40) = −1* → RR; 30'da sola → 20(10, 40(30, 50)). 35: 30'un sağına düşer; *bf(20) = −2*, *bf(40) = 1* → RL; 40'ta sağa, 20'de sola. Son ağaç 30(20(10, —), 40(35, 50)), yükseklik 2.

6. Boş bir kırmızı-siyah ağaca sırasıyla 30, 20, 10, 25, 27 ekleniyor. Düzeltme gerektiren eklemelerde hangi durum uygulanır? Son ağacı renkleriyle yazın.

Cevap

10: ebeveyn 20 kırmızı, amca NIL, 10–20–30 çizgi → 30'da sağa döndür; 20 siyah, 30 kırmızı: 20S(10K, 30K). 25: ebeveyn 30 kırmızı, amca 10 kırmızı → renk değiştir (10 ve 30 siyah, 20 kırmızı), 20 kök olduğu için yeniden siyah. 27: ebeveyn 25 kırmızı, amca NIL; 27 sağ, 25 sol çocuk → üçgen: 25'te sola, sonra 30'da sağa döndür; 27 siyah, 30 kırmızı. Son ağaç 20S(10S, 27S(25K, 30K)); her yolda 2 siyah düğüm.

7. Splay ağacı 50(30(20(10, —), 40), 70) olsun. `splay(40)` hangi adımla çalışır ve sonuç ağacı nedir?

Cevap

Arama yolu 50 → 30 → 40'tır; p = 30, g = 50. 40, 30'un sağ çocuğu; 30 ise 50'nin sol çocuğudur: zıt yön, zig-zag. Önce 30'da sola, sonra 50'de sağa döndürülür. Sonuç 40(30(20(10, —), —), 50(—, 70)). 40 tek adımda köke çıktı.

8. Derecesi 3 olan boş bir B-ağacına sırasıyla 5, 15, 25, 35, 45 ekleniyor. Hangi eklemelerde bölünme olur ve son ağaç nedir?

Cevap

25 eklenince [5, 15, 25] taşar; 15 yeni köke çıkar: [15]{[5] [25]}. 35 sağ yaprağa eklenir: [25, 35]. 45 eklenince [25, 35, 45] taşar; 35 köke çıkar ve kökte yer vardır: [15, 35]{[5] [25] [45]}. İki bölünme olur, kök bir kez bölünür; yükseklik 1'dir.

9. Kökünün siyah yüksekliği 3 olan bir kırmızı-siyah ağaçta kökten bir NIL'e giden en kısa ve en uzun

yol en çok kaç düğüm içerebilir? Böyle bir ağaçta en az kaç düğüm vardır?

Cevap

Her yolda 3 siyah düğüm vardır. En kısa yol yalnız siyahlardan oluşur: 3 düğüm. En uzun yolda her siyahın altına bir kırmızı girebilir: en çok 6 düğüm (kök siyah olduğu için siyah, kırmızı, siyah, kırmızı, siyah, kırmızı). En az düğüm sayısı $2^3 - 1 = 7$'dir: bütün düğümleri siyah olan, yüksekliği 2 olan mükemmel ağaç.

10. Şekil 10.42'deki B+ ağacında 22 ile 55 arasındaki anahtarlar isteniyor. Sonuç nedir ve kaç yaprak okunur?

Cevap

Kökte $20 \le 22 < 40$ olduğu için orta çocuğa, [20, 25, 30] yaprağına inilir. 20 atlanır; 25 ve 30 alınır. Bağ izlenerek [40, 50, 60] yaprağına geçilir; 40 ve 50 alınır, $60 > 55$ olduğu için durulur. Sonuç 25 30 40 50'dir; 2 yaprak okunur ve köke bir daha çıkılmaz.

10.12 İleri okuma

- M. T. Goodrich, R. Tamassia ve M. H. Goldwasser, *Data Structures and Algorithms in Java* (6. baskı, 2014), Bölüm 11: ikili arama ağaçlarını, AVL, splay, (2, 4) ve kırmızı-siyah ağaçları aynı sırayla işler. Kodları bu nottakilere yakın bir Java diliyle yazılmıştır.
- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 3.2 ve 3.3: BST'yi ve sola yatık kırmızı-siyah ağacı 2-3 ağaçlarıyla ilişkilendirerek anlatır; kırmızı-siyah ağacın neden çalıştığını görmenin en kolay yoludur.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 12, 13 ve 18: BST, kırmızı-siyah ağaç (silme dahil bütün durumlarıyla) ve B-ağaçlarını ispatlarıyla verir.
- D. D. Sleator ve R. E. Tarjan, “Self-Adjusting Binary Search Trees” (1985): splay ağacını tanıtan özgün makaledir; amortize $O(\log n)$ sınırının potansiyel fonksiyonuyla ispatını içerir.

10.13 Kaynaklar

1. G. M. Adelson-Velsky, E. M. Landis, “An algorithm for the organization of information”, *Soviet Mathematics Doklady*, 3, 1259–1263, 1962.
2. R. Bayer, E. M. McCreight, “Organization and maintenance of large ordered indexes”, *Acta Informatica*, 1(3), 173–189, 1972.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022.

4. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6. baskı, Wiley, 2014.
5. L. J. Guibas, R. Sedgewick, "A dichromatic framework for balanced trees", *19th Annual Symposium on Foundations of Computer Science*, 8–21, 1978.
6. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011.
7. D. D. Sleator, R. E. Tarjan, "Self-adjusting binary search trees", *Journal of the ACM*, 32(3), 652–686, 1985.

KISIM III · BÖLÜM 11

Öncelikli Kuyruklar ve Heap

Sıradan bir kuyrukta önce gelen önce çıkar. Gerçek hayatta ise sıra çoğu zaman geliş zamanına değil, işin aciliyetine göre belirlenir: acil serviste kalp krizi geçiren hasta, kendisinden önce gelmiş burkulma vakasının önüne alınır. Öncelikli kuyruk bu düzenin veri yapıları dilindeki karşılığıdır. Not, öncelikli kuyruğun soyut tanımıyla başlar ve onu önce sırasız ya da sıralı bir listeyle gerçekler. Bu iki kolay yolun her birinde işlemlerden biri yavaş kalır. İkisini dengeleyen yapı ikili heap'tir: hem ekleme hem çıkarma logaritmik sürede yapılır. Heap'in dizi gösterimi, yukarı ve aşağı süzme işlemleri, anahtar değiştirme, doğrusal zamanda heap oluşturma ve heap sort ayrıntılı kod izlemeleriyle anlatılır. Son kesim, Java'nın hazır `PriorityQueue` sınıfını, k'ıncı en büyük öğe problemini ve işletim sistemlerindeki öncelikli çizelgelemeyi ele alır.

Bu bölümün kodu: `github.com/sercankulcu/data-structures-java/Bolum11/src`

ÖĞRENME ÇIKTILARI

ADT modeli
Öncelikli kuyruğun temel işlemlerini ve kullanım alanlarını açıklamak; bir işlem dizisinden sonra dönen değerleri ve kuyruğun içeriğini bulmak.

İkili heap
Heap-sıra özelliğini ve tam ikili ağaç özelliğini dizi gösterimiyle ilişkilendirmek; ebeveyn ve çocuk indislerini hesaplamak.

İşlemler
Ekleme, kökü çıkarma, anahtar değiştirme ve heap oluşturma adımlarını elle ve kodla uygulamak.

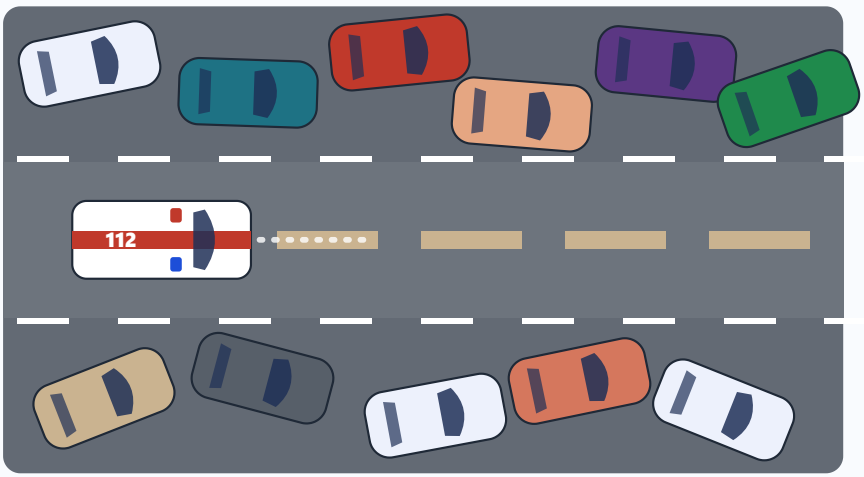
Maliyet analizi
İşlem maliyetlerini gerekçesiyle vermek; aşağıdan yukarıya heap oluşturmanın neden doğrusal olduğunu ve heap sort'un nasıl çalıştığını açıklamak.

11.1 Öncelikli kuyruk ADT'si

Bir veri yapısı iki düzeyde anlatılabilir. Birinci düzey yapının *ne yaptığını* söyler: hangi işlemler vardır, her işlem neyi değiştirir ve ne döndürür. İkinci düzey bunun *nasıl yapıldığını* söyler: veriler bellekte hangi biçimde durur, her işlem hangi adımlarla yürür. Birinci düzeydeki tarife **soyut veri tipi** (*abstract data type*, kısaca ADT) denir. Öncelikli kuyruk da bir ADT'dir. Aynı ADT birbirinden çok farklı veri yapılarıyla gerçeklenebilir ve bu bölümün ana fikirlerinden biri, seçilen veri yapısının çalışma süresini nasıl değiştirdiğidir.

11.1.1 Öncelikli kuyruk nedir?

Sıkışık bir trafiği düşünelim. Araçlar şeritlerde geliş sırasıyla ilerler; öndeki araç önce geçer. Arkadan sireni açık bir ambulans geldiğinde ise sürücüler kenara çekilir ve ambulans hepsinin önüne geçer. Ambulansın öne geçmesinin nedeni erken gelmiş olması değildir. Önceliği yüksektir.



Şekil 11.1: Sıkışık trafikte araçlar kenara çekilir, ambulans açılan şeritten öne geçer. Önce gelen değil, önceliği yüksek olan geçer.

Şekil 11.1 bu düzeni gösteriyor. Aynı düzen bilgisayar sistemlerinde de sık görülür. İşletim sistemi bekleyen süreçler arasından en önemlisini çalıştırır. Bir yazıcı kuyruğunda yöneticinin acil belgesi öne alınabilir. Bir olay simülasyonu, zamanı en erken olan olayı işler. Bu durumların hepsinde bekleyen öğeler bir "öncelik" bilgisi taşır ve sıradaki öğe bu bilgiye göre seçilir.

Tanım 11.1

Öncelikli kuyruk (*priority queue*), her öğesi bir öncelik taşıyan ve çıkarma işleminde her zaman en yüksek öncelikli öğeyi veren soyut veri tipidir. Öğelerin kuyruğa giriş sırası çıkış sırasını belirlemez; çıkış sırasını öncelikler belirler.

Tanımın vurguladığı nokta şudur: öncelikli kuyruk yalnız "sıradaki en önemli öğe hangisi?" sorusuna hızlı cevap vermek zorundadır. Bütün öğeleri sıralı tutmak zorunda değildir. Kuyruğun ortasındaki öğelerin kendi aralarında hangi düzende durduğu kullanıcıyı ilgilendirmez. Bu serbestlik, verimli gerçeklemelerin anahtarıdır.

İşlemlerin maliyeti seçilen gerçeklemeye bağlıdır. Öğeler sırasız bir listede tutulursa en öncelikli öğeye yalnız bakmak bile bütün listeyi taramayı, yani *O(n)* zamanı gerektirir. İkili heap ile ise en öncelikli öğeye bakmak *O(1)*, onu çıkarmak *O(log n)* sürer.

11.1.2 Anahtar, değer ve öncelik

Öncelikli kuyruktaki her öğe iki parçadan oluşur. Birinci parça **anahtardır** (*key*): öğenin önceliğini taşır. İkinci parça **değerdir** (*value*): kuyrukta saklanmak istenen asıl veridir. Acil servis örneğinde `(1, "kalp krizi")` ve `(4, "burkulma")` iki öğedir. Birinci öğenin anahtarı 1, değeri "kalp krizi" metnidir.

> **Tanım 11.2**
>
> Öncelikli kuyruktaki her öğe bir **(anahtar, değer)** çiftidir. Anahtarlar karşılaştırılabilir olmalıdır: her iki anahtar için hangisinin küçük olduğu $<$ ve $\leq$ ile belirlenebilmelidir. **Min-öncelikli kuyrukta** küçük anahtar yüksek öncelik demektir; çıkarma işlemi en küçük anahtarlı öğeyi verir. **Max-öncelikli kuyrukta** büyük anahtar yüksek önceliktir; çıkarma en büyük anahtarlı öğeyi verir.

Hangi yönün seçildiği bir alışkanlık meselesidir. Goodrich, Tamassia ve Goldwasser'in ders kitabı ADT'yi min-öncelikli kuyruk olarak tanımlar ve işlemlere `min()`, `removeMin()` adlarını verir. Bu notta da ADT bu biçimde verilir. Heap kodlarının çoğu ise max-heap üzerindedir; orada işlemler `ekle` ve `silMax` adını alır. İki yön arasındaki tek fark karşılaştırmanın yönüdür: min-kuyruk kodundaki her $<$ işareti $>$ ile değiştirilirse max-kuyruk elde edilir. Bu yüzden bir yönde öğrenilen her şey öbür yöne doğrudan taşınır.

En yüksek öncelikli öğe, kuyruğun "önündeki" öğedir. Min-kuyrukta bu, anahtarı en küçük olan öğedir; max-kuyrukta anahtarı en büyük olan öğedir. Aynı en küçük anahtardan birden çok olabilir. O durumda bunlardan herhangi biri önde sayılabilir.

Bu son gözlem önemli bir soruyu doğurur: **eşit öncelikli öğeler hangi sırayla çıkar?** Öncelikli kuyruk ADT'si bu konuda hiçbir söz vermez. Eşit anahtarlı iki öğeden önce gelenin önce çıkacağı, yani FIFO (*first-in, first-out*: ilk giren ilk çıkar) düzeninin korunacağı kendiliğinden garanti değildir. Bazı gerçeklemeler bu sırayı tesadüfen korur, bazıları korumaz. Java'nın `PriorityQueue` sınıfı da bu konuda bir garanti vermez.

Eşitlikte geliş sırasının korunması gerekiyorsa çözüm basittir: anahtara bir **geliş sıra numarası** eklenir. Her yeni öğe bir sayaçtan artan bir numara alır. Karşılaştırma önce önceliğe bakar; öncelikler eşitse küçük sıra numaralı, yani önce gelmiş öğe önde sayılır. Sıra numaraları hep farklı olduğu için iki

öğe artık hiçbir zaman tam eşit olmaz ve sıra kesinleşir.

Sık yapılan hata

Öncelikli kuyruğun eşit anahtarlarda ekleme sırasını koruduğunu varsaymak. Heap tabanlı gerçeklemeler eşit anahtarlı öğelerin yerini takaslarla değiştirir; çıkış sırası ekleme sırasından farklı olabilir. FIFO gerekiyorsa anahtar (öncelik, sıra numarası) çiftine dönüştürülür.

11.1.3 Kuyruk mu, öncelikli kuyruk mu?

Sıradan kuyrukla öncelikli kuyruk arasındaki farkı küçük bir örnek açıkça gösterir.

Örnek 11.1

Acil servise sırasıyla dört hasta geliyor. Küçük sayı daha acil durumu gösteriyor: Ali (3), Ece (1), Can (2), Naz (1). Hastalar sıradan bir kuyrukta ve min-öncelikli bir kuyrukta beklerse hangi sırayla hizmet alırlar?

1. **Kuyruk (FIFO).** Çıkış sırası geliş sırasıyla aynıdır: Ali, Ece, Can, Naz. Önceliklere hiç bakılmaz.
2. **Öncelikli kuyruk.** İlk çıkan, anahtarı en küçük olan öğedir. Anahtarı 1 olan iki hasta var: Ece ve Naz. Geliş sırası korunursa önce Ece, sonra Naz çıkar.
3. Kalanlar Can (2) ve Ali (3)'tür. Önce Can, en son Ali çıkar. Sıra: Ece, Naz, Can, Ali.

Sıra	1.	2.	3.	4.
Geliş (öncelik)	Ali (3)	Ece (1)	Can (2)	Naz (1)
Kuyruk (FIFO)	Ali	Ece	Can	Naz
Öncelikli kuyruk	Ece	Naz	Can	Ali

Tablo 11.1: Aynı dört hastanın kuyrukta ve öncelikli kuyrukta çıkış sırası. Ece ile Naz eşit öncelikli; burada geliş sırası korunmuştur.

Ali kuyruğa ilk girdiği hâlde öncelikli kuyruktan en son çıkmıştır. Kuyrukta çıkış sırasını geliş zamanı belirler; öncelikli kuyrukta öncelik belirler. Ece ile Naz'ın göreli sırası ise ADT'nin değil, gerçeklemenin kararıdır. Tabloda geliş sırası korunmuş, ama başka bir gerçekleme Naz'ı Ece'den önce de çıkarabilirdi.

Öncelikli kuyruğu sıradan kuyruğun bir genellemesi olarak da görmek mümkündür. Her öğenin anahtarı geliş zamanı olarak seçilirse, min-öncelikli kuyruk en eski öğeyi, yani kuyruğun önündeki öğeyi verir. Aynı şekilde anahtar olarak geliş zamanının eksisi seçilirse en yeni öğe çıkar ve yapı yığın gibi davranır.

11.1.4 Kullanım alanları

Öncelikli kuyruk, "bekleyenler arasından her seferinde en iyisini seç" biçimindeki bütün problemlerde kullanılır. Dört tipik alan şunlardır:

- **İşletim sistemleri.** Görev (süreç) çizelgelemede yüksek öncelikli süreç işlemciyi önce alır. Örneğin bir kesme işleyicisi kullanıcı arayüzünden, arayüz de arka plandaki yedekleme işinden önce gelir.
- **Çizge algoritmaları.** Dijkstra'nın en kısa yol algoritması ve A* araması, sıradaki en yakın düğümü bir öncelikli kuyruktan alır. Anahtar, düğümün kaynağa o ana kadar bilinen uzaklığıdır.
- **Acil durum yönetimi.** Hasta triyajında ve olay yönetiminde en acil vaka önce ele alınır.
- **Veri sıkıştırma.** Huffman kodlaması her adımda frekansı en düşük iki düğümü birleştirir. Bu iki düğüm bir min-öncelikli kuyruktan iki kez `removeMin` ile alınır, birleşim düğümü kuyruğa geri eklenir.

Bu örneklerin ortak yanı, öğelerin kuyruğa farklı zamanlarda girmesi ve çıkarma ile ekleme işlemlerinin iç içe gelmesidir. Bütün veri baştan belli olsaydı diziyi bir kez sıralamak yeterdi. Dijkstra algoritmasında ise bir düğüm çıkarıldıktan sonra komşuları kuyruğa eklenir; sıralı bir diziyi her eklemede yeniden düzenlemek pahalıya mal olur. Öncelikli kuyruk tam bu durum için tasarlanmıştır.

11.1.5 Temel işlemler ve Java arayüzü

Min-öncelikli kuyruk ADT'si beş işlemle tanımlanır. Tablo 11.2 bu işlemleri ve anlamlarını listeliyor.

İşlem	Anlamı
`insert(k, v)`	Ekleme: anahtarı `k`, değeri `v` olan öğe kuyruğa girer ve önceliğine göre yerini alır.
`removeMin()`	Çıkarma: en yüksek öncelikli (en küçük anahtarlı) öğeyi kuyruktan çıkarır ve döndürür.
`min()`	Sorgulama (*peek*): en yüksek öncelikli öğeyi döndürür, ama çıkarmaz.
`size()`	Kuyruktaki öğe sayısını döndürür.
`isEmpty()`	Kuyruk boşsa `true` döndürür.

Tablo 11.2: Min-öncelikli kuyruğun temel işlemleri.

Java'da bir ADT'nin sözleşmesi doğal olarak bir arayüzle (*interface*) yazılır. Arayüz yalnız işlemlerin adlarını, parametrelerini ve dönüş tiplerini verir; işlemlerin nasıl yapılacağını söylemez:

```
public interface PriorityQueue<K, V> {
    void insert(K key, V value);
    Entry<K, V> min();        // boşsa null
    Entry<K, V> removeMin();  // boşsa null
    int size();
    boolean isEmpty();
}
```

Arayüz iki tip parametresi alır: anahtar tipi `K` ve değer tipi `V`. `Entry<K, V>` bir (anahtar, değer) çiftini temsil eden küçük bir tiptir; `min()` ve `removeMin()` öğenin yalnız değerini değil, anahtarını da döndürsün diye bu tip kullanılır. Yorumlar boş kuyruktaki davranışı belirtir: iki metot da `null` döndürür. Bazı kütüphaneler bu durumda istisna fırlatır. Hangisinin seçildiği sözleşmenin parçasıdır ve arayüzü kullanan kodun bunu bilmesi gerekir.

Max-öncelikli kuyrukta aynı işlemler `max()` ve `removeMax()` adını alır. Bu notun heap kodları max-heap kullanır ve işlemleri Türkçe adlarla verir: ekleme `ekle`, en büyüğü çıkarma `silMax`.

Arayüzün kendisi hiçbir maliyet bilgisi taşımaz. Maliyet, arayüzü uygulayan sınıfa bağlıdır. Aynı `PriorityQueue<K, V>` arayüzünü bir sırasız liste, bir sıralı liste ve bir heap ayrı ayrı uygulayabilir; kullanıcı kodu değişmez, yalnız çalışma süresi değişir.

11.1.6 Örnek işlem dizisi

ADT'nin davranışını bir işlem dizisiyle somutlaştıralım. Boş bir min-öncelikli kuyruğa sırasıyla aşağıdaki işlemler uygulanıyor. Tabloda her işlemin döndürdüğü değer ve işlemden sonraki içerik verilmiştir. İçerik küme parantezleriyle yazıldı, çünkü kuyruğun iç düzeni ADT düzeyinde tanımlı değildir; okumayı kolaylaştırmak için öğeler anahtar sırasıyla dizildi.

İşlem	Dönen değer	Kuyruğun içeriği
`insert(5, A)`	—	{ (5,A) }
`insert(9, C)`	—	{ (5,A), (9,C) }
`insert(3, B)`	—	{ (3,B), (5,A), (9,C) }
`min()`	(3,B)	{ (3,B), (5,A), (9,C) }
`removeMin()`	(3,B)	{ (5,A), (9,C) }
`insert(7, D)`	—	{ (5,A), (7,D), (9,C) }
`removeMin()`	(5,A)	{ (7,D), (9,C) }
`removeMin()`	(7,D)	{ (9,C) }
`removeMin()`	(9,C)	{ }
`removeMin()`	`null`	{ } (boş kuyruk)

Tablo 11.3: Boş min-öncelikli kuyrukta bir işlem dizisi.

Tablodaki üç ayrıntıya dikkat etmek gerekir. Birincisi, `min()` ile `removeMin()` aynı öğeyi döndürmüştür, ama yalnız ikincisi onu kuyruktan çıkarır. `min()`'den sonra içerik değişmemiştir. İkincisi, (3,B) kuyruğa (5,A) ve (9,C)'den sonra girmesine karşın ilk çıkan öğedir; sırayı anahtar belirler. Üçüncüsü, (7,D) en son eklenen öğedir ama (9,C)'den önce çıkar. Son satırda boş kuyrukta `removeMin()` çağrılmış ve arayüzün sözleşmesine uygun olarak `null` dönmüştür.

Sık yapılan hata

`min()` 'in öğeyi kuyruktan çıkardığını sanmak. `min()` (Java'da `peek()`) yalnız bakar; kuyruğun içeriğini değiştirmez. Aynı öğe art arda iki `min()` çağrısında da döner. Öğeyi çıkaran işlem `removeMin()` 'dir (Java'da `poll()`).

11.2 Liste ile gerçekleme

Öncelikli kuyruğu gerçeklemenin en kolay yolu, öğeleri bir listede tutmaktır. Bu listede öğelerin düzeni için iki seçenek vardır: öğeler ya geldikleri gibi sırasız durur ya da her zaman önceliğe göre sıralı tutulur. İki seçenek de doğru çalışır, ama maliyetleri birbirinin aynadaki görüntüsü gibidir. Birinde ekleme ucuz, çıkarma pahalıdır; öbüründe tersi geçerlidir. Bu kesim iki seçeneği kodla ve izlemelerle karşılaştırır. Kazanılan sezgi, ikili heap'in neden gerekli olduğunu da açıklar.

11.2.1 Sırasız liste ve sıralı liste

Sırasız liste (*unsorted list*) gerçeklemesinde yeni öğe listenin sonuna eklenir. Hiçbir karşılaştırma yapılmaz; ekleme *O(1)*'dir. Bedel çıkarmada ödenir. En küçük anahtarlı öğe listenin herhangi bir yerinde olabilir. Onu bulmak için listenin bütün öğelerine bakmak gerekir: `min` ve `removeMin` *O(n)* sürer.

Sıralı liste (*sorted list*) gerçeklemesinde öğeler her zaman anahtar sırasıyla durur. En küçük öğe hep listenin bir ucundadır; `min` ve `removeMin` bu uca bakar ve *O(1)* sürer. Bu kez bedel eklemede ödenir. Yeni öğe sona atılamaz; sırayı bozmamak için doğru yerinin bulunması ve öğelerin kaydırılması gerekir. En kötü durumda bütün öğeler yer değiştirir ve ekleme *O(n)* olur.



Şekil 11.2: Solda sırasız liste 5, 9, 3, 7: yeni öğe 4 sona eklenir, en küçüğü bulmak için bütün liste taranır. Sağda sıralı liste: 4, 3 ile 5 arasına yerleşir; en küçük öğe hep baştadır.

Şekil 11.2 iki durumu yan yana gösteriyor. İki gerçekleme de işlemlerden birini hızlı, öbürünü yavaş yapar. Biri eklemede, diğeri çıkarmada üstündür. Öncelikli kuyruk uygulamalarında ekleme ve çıkarma genellikle yaklaşık eşit sayıda yapılır. Bu durumda yavaş işlem toplam süreyi belirler ve iki gerçek-

leme de *n* ekleme ile *n* çıkarma için $O(n^2)$ zaman harcar. İki işlemi dengeleyen yapı ikili heap'tir: orada ikisi de $O(\log n)$ sürer.

Sıralı listede sıralamanın yönü bir gerçekleme kararıdır. Şekildeki liste küçükten büyüğe sıralıdır ve en küçük öğe baştadır. Dizi ile gerçeklemede ise en küçüğü dizinin *sonunda* tutmak daha akıllıcadır. Bunun nedeni, iki temsilde ucuz ucun farklı olmasıdır: dizide son hücreyi silmek, bağlı listede ise baştaki düğümü silmek sabit sürer.

11.2.2 Kod izleme: sırasız dizide removeMin()

Sırasız gerçeklemeyi bir dizi üzerinde yazalım. Öğeler `a[0]`'dan `a[n - 1]`'e kadar dizide durur ve `n` öğe sayısını tutar. Ekleme tek satırdır: `a[n] = x; n++;`. Çıkarma ise şöyledir:

```
// sırasız dizi: insert(x) { a[n] = x; n++; }  O(1)
public int removeMin() {        // n > 0 olmalı
  int enk = 0;                  // en küçüğün indisi
  for (int k = 1; k < n; k++)
    if (a[k] < a[enk]) enk = k;
  int min = a[enk];
  a[enk] = a[n - 1];  n--;      // son öğe boşluğa
  return min;
}
```

Kod üç iş yapar. Birinci iş en küçüğün yerini bulmaktır. `enk` değişkeni o ana kadar görülen en küçük öğenin indisini tutar ve ilk aday olarak 0 alınır. `for` döngüsü `k = 1`'den `n - 1`'e kadar her öğeyi o anki adayla karşılaştırır; daha küçük bir öğe bulunca `enk` güncellenir. Döngü bittiğinde `enk` en küçük öğenin indisidir. İkinci iş bu öğeyi `min` değişkeninde saklamaktır.

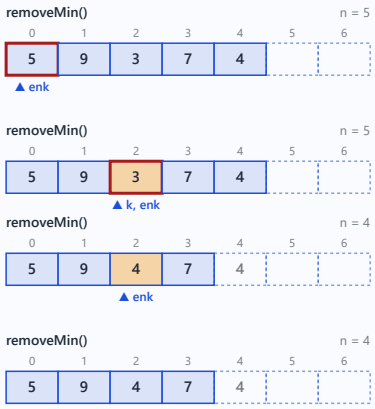
Üçüncü iş, çıkarılan öğenin bıraktığı boşluğu kapatmaktır. Dizi sırasız olduğu için öğelerin düzenini korumak gerekmez. Boşluğu kapatmak için sağdaki bütün öğeleri birer sola kaydırmak yerine son öğe boşluğa taşınır ve `n` bir azaltılır. Bu, tek bir atamadır ve $O(1)$ sürer. Sıralı bir dizide aynı hile kullanılamazdı, çünkü son öğeyi ortaya taşımak sırayı bozardı.

Kodu `a = [5, 9, 3, 7, 4]`, `n = 5` dizisi üzerinde izleyelim. Tablo 11.4 döngünün her adımını veriyor.

Adım	Karşılaştırma	`enk`	Dizi
başlangıç	—	0	[5, 9, 3, 7, 4], `n = 5`
`k = 1`	`a[1] = 9 < a[0] = 5` ? hayır	0	değişmez
`k = 2`	`a[2] = 3 < a[0] = 5` ? **evet**	2	değişmez
`k = 3`	`a[3] = 7 < a[2] = 3` ? hayır	2	değişmez
`k = 4`	`a[4] = 4 < a[2] = 3` ? hayır	2	değişmez
`k = 5 = n`	döngü biter	2	4 karşılaştırma yapıldı
`min = a[2]`	—	2	`min = 3`

Adım	Karşılaştırma	enk	Dizi
`a[2] = a[4]; n--`	—	2	[5, 9, 4, 7], `n = 4`
`return`	—	—	3 döner

Tablo 11.4: `a = [5, 9, 3, 7, 4]` üzerinde `removeMin()` izlemesi.



Şekil 11.3: Sırasız dizide `removeMin()` 'in dört anı. (a) İlk aday `enk = 0`. (b) `k = 2` 'de daha küçük 3 bulunur, `enk = 2`. (c) Son öğe 4 boşluğa taşınmış, `n = 4`. (d) 3 döner; kalan dizi [5, 9, 4, 7] yine sırasızdır.

Beş öğeli dizide 4 karşılaştırma yapıldı. Genel olarak n öğeli dizide döngü $n - 1$ kez döner. En küçük öğe nerede olursa olsun bu sayı değişmez; en küçük ilk hücrede olsa bile algoritma bunu ancak bütün diziyi gördükten sonra bilebilir. Bu yüzden `removeMin` her durumda $\Theta(n)$ zamanlıdır. `min()` da aynı taramayı yapar ve maliyeti aynıdır; yalnız sondaki taşıma adımı yoktur.

Sonuçta kalan dizi [5, 9, 4, 7]'dir ve sırasızdır. Bu bir sorun değildir: sırasız gerçekleme zaten düzen tutmaz. Taşıma hilesinin dizinin düzenini değiştirdiğine dikkat edilmeli. 4 dizinin sonundan ortasına geçmiştir.

Sık yapılan hata

Sırasız dizide boşluğu, sağdaki bütün öğeleri birer sola kaydırarak kapatmak. Sonuç doğrudur ama gereksiz yere $O(n)$ iş yapılır. Düzen önemsiz olduğu için son öğeyi boşluğa taşımak yeterlidir. Tersine, sıralı dizide bu hile kullanılamaz.

11.2.3 Kod izleme: sıralı dizide insert(4)

Sıralı gerçeklemede en küçüğü dizinin hangi ucunda tutacağımızı seçmek gerekir. En küçük `a[0]` 'da durursa onu çıkarmak, kalan bütün öğeleri birer sola kaydırmayı gerektirir; `removeMin` $O(n)$ olur ve sıralı gerçeklemenin tek üstünlüğü kaybolur. Doğrusu, diziyi **büyükten küçüğe** sıralı tutmaktır. O zaman en küçük öğe hep `a[n - 1]` 'dedir ve onu çıkarmak yalnız `n` 'yi bir azaltmaktır:

```
// sıralı dizi: büyükten küçüğe, min hep a[n-1]
public void insert(int x) {
  int k = n - 1;
  while (k >= 0 && a[k] < x) {    // x'ten küçükler
    a[k + 1] = a[k];              // bir sağa kaydır
    k--;
  }
  a[k + 1] = x;  n++;
}
public int removeMin() { n--; return a[n]; }
```

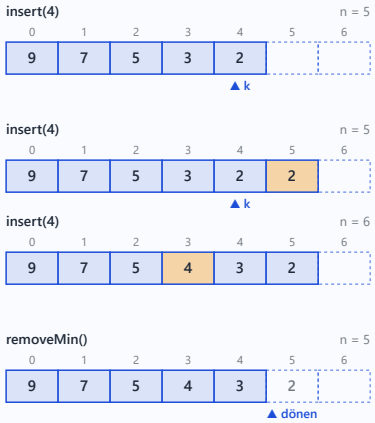
`insert` metodu, yeni öğe `x`'in yerini sondan başa doğru arar. `k` son öğeden başlar. `a[k] < x` olduğu sürece `a[k]`, `x`'ten küçüktür ve büyükten küçüğe düzende `x`'ten sonra gelmelidir. Bu yüzden bir sağa, `a[k + 1]`'e kaydırılır ve `k` bir azalır. Döngü iki durumda biter: `k` −1'e iner (yani `x` bütün öğelerden büyüktür) ya da `a[k] ≥ x` olan bir öğeye gelinir. İki durumda da `x`'in yeri `k + 1`'dir. Koşuldaki `k >= 0` denetiminin `a[k]`'ye erişimden önce yazıldığına dikkat edilmeli: `&&` kısa devre yaptığı için `k = -1` olduğunda `a[-1]` hiç okunmaz.

Kodu `a = [9, 7, 5, 3, 2]`, `n = 5` dizisinde `insert(4)` için izleyelim.

Adım	`k`	Koşul	Dizi
başlangıç	4	—	[9, 7, 5, 3, 2, ·]
karşılaştır	4	`a[4] = 2 < 4` : doğru	[9, 7, 5, 3, 2, ·]
`a[5] = a[4]`	4 → 3	—	[9, 7, 5, 3, 2, 2]
karşılaştır	3	`a[3] = 3 < 4` : doğru	[9, 7, 5, 3, 2, 2]
`a[4] = a[3]`	3 → 2	—	[9, 7, 5, 3, 3, 2]
karşılaştır	2	`a[2] = 5 < 4` : yanlış, döngü biter	[9, 7, 5, 3, 3, 2]
`a[3] = 4; n++`	2	—	[9, 7, 5, 4, 3, 2], `n = 6`
`removeMin()`	—	`n--`, `a[5]` döner	2 döner, `n = 5`

Tablo 11.5: `a = [9, 7, 5, 3, 2]` üzerinde `insert(4)` ve ardından `removeMin()` izlemesi. Dizinin ilk altı hücresi gösterilmiştir; boş hücre "·" ile yazıldı.

İzlemede iki öğe (2 ve 3) birer sağa kaydı. Kaydırma sırasında dizide geçici olarak aynı değerden iki tane görülür: `a[5] = a[4]`'ten sonra 2 iki hücrededir. Bu bir hata değildir. `a[4]`'teki kopyanın üzerine bir sonraki adımda 3 yazılır; o adımda `a[3]`'te kalan kopyanın üzerine de en sonda 4 yazılır. Kaydırma, "bir öğeyi sağa kopyala, boşalan yere sonrakini yaz" düşüncesiyle çalışır ve takastan daha az atama yapar.



Şekil 11.4: Sıralı dizide `insert(4)`. (a) `k = n − 1 = 4`'ten başlanır. (b) 2 bir sağa kaydırılmış; `a[4]` şimdilik kopya. (c) 4, `a[3]`'e yazılmış, `n = 6`; dizi yine sıralı. (d) `removeMin()` sondaki 2'yi kaydırmadan döndürür.

Maliyet, kaydırılan öğe sayısıyla orantılıdır. Yeni öğe en küçükse hiçbir öğe kaymaz ve ekleme sabit sürer. En kötü durum, yeni öğenin bütün öğelerden büyük olmasıdır: o zaman n öğenin hepsi bir sağa kayar ve ekleme $O(n)$ olur. Ortalamada da yaklaşık $n/2$ öğe kayar; bu da $O(n)$'dir. Buna karşılık `removeMin()` tek satırlıktır ve $O(1)$'dir: en küçük zaten sondadır.

> **Sık yapılan hata**
>
> Sıralı dizide en öncelikli öğeyi `a[0]`'da tutmak. O zaman her `removeMin`, kalan öğeleri sola kaydırmak zorunda kalır ve ekleme de çıkarma da $O(n)$ olur. En öncelikli öğe dizinin *sonunda* tutulursa çıkarma $O(1)$'dir.

11.2.4 Dizi ve bağlı liste temsili

Sıralı gerçeklemeyi dizi yerine bir bağlı listeyle de yazmak mümkündür. İki temsilin maliyetleri benzer, ama farklı yerlerden gelir.

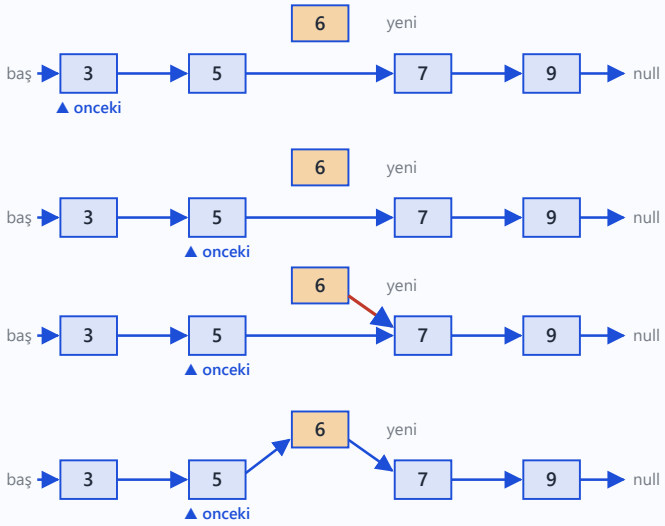
Dizi temsilinde öğeler ardışık hücrelerde durur. Sıralı tutmak için ekleme sırasında öğeler kaydırılır; bu $O(n)$'dir. En öncelikli öğe dizinin başında tutulursa her çıkarmada bütün dizi sola kaydırılmak zorundadır. Bu yüzden en öncelikli öğe dizinin sonunda tutulur ve çıkarma $O(1)$ olur.

Bağlı liste temsilinde kaydırma yoktur. Yeni düğüm için yer aramak yine $O(n)$'dir, çünkü bağlı listede bir konuma ancak baştan yürüyerek ulaşılır. Yer bulunduktan sonra düğümü listeye bağlamak iki referans atamasıdır ve $O(1)$ sürer. Bağlı listede en öncelikli öğe *baştadır*: baştan silmek tek atamalık bir iştir. Sondan silmek ise tek yönlü listede sondan bir önceki düğümü bulmayı gerektirdiği için pahalıdır. Dizide son, bağlı listede baş ucuz uçtur; en öncelikli öğe her temsilde ucuz uca konur. Bağlı listenin bedeli bellektir: her düğüm, verinin yanında bir de sonraki düğümün referansını taşır.

Örnek 11.2

Küçükten büyüğe sıralı 3 → 5 → 7 → 9 bağlı listesine 6 eklensin. Liste min-öncelikli olduğu için en öncelikli öğe 3 baştadır.

1. **Düğüm oluşturma.** Değeri 6 olan `yeni` düğüm oluşturulur; henüz listeye bağlı değildir.
2. **Yer arama.** `onceki` baştan başlar. `onceki.sonraki.deger` 6'dan küçük olduğu sürece ilerler: 3'ün sonrakisi 5'tir (5 < 6, ilerle); 5'in sonrakisi 7'dir (7 > 6, dur). `onceki` 5'te kalır. Bu tarama en kötü durumda bütün listeyi gezer: *O(n)*.
3. **İlk bağ.** `yeni.sonraki = onceki.sonraki` : 6, 7'yi gösterir.
4. **İkinci bağ.** `onceki.sonraki = yeni` : 5, 6'yı gösterir. Liste 3 → 5 → 6 → 7 → 9 olur.



Şekil 11.5: Sıralı bağlı listeye 6 ekleme. (a) Yeni düğüm oluşturuldu. (b) Yer arama `onceki = 5` 'te durur. (c) Önce `yeni.sonraki = onceki.sonraki` (6 → 7). (d) Sonra `onceki.sonraki = yeni` (5 → 6).

Son iki adımın sırası önemlidir. Önce `onceki.sonraki = yeni` yazılsaydı 5'ten 7'ye giden tek bağ silinirdi. 7 ve ondan sonraki düğümlere artık hiçbir referans ulaşmazdı ve listenin geri kalanı kaybolurdu. Doğru sırada önce yeni düğüm listenin geri kalanına bağlanır, sonra liste yeni düğüme bağlanır.

11.2.5 Kod: sıralı dizi ile öncelikli kuyruk

Sıralı dizi gerçeklemesini, öğelerin bir veri ve bir öncelik taşıdığı genel bir sınıfa dönüştürelim. Bu sınıf max-öncelikli çalışır: büyük öncelik değeri önce çıkar. Önce her öğeyi temsil eden küçük bir sınıf tanımlanır:

```
class Oge<T> {
    T veri;
    int oncelik;
    Oge(T v, int p) { veri = v; oncelik = p; }
}
```

`Oge<T>` jenerik bir sınıftır: `veri` alanı her tipten olabilir, `oncelik` ise bir tam sayıdır. Kuyruk sınıfı bu öğeleri bir dizide tutar:

```
// OncelikliKuyruk<E> alanları
private Oge<E>[] dizi =
    (Oge<E>[]) new Oge[10];
private int boyut = 0;
public boolean bosMu() { return boyut == 0; }
```

Dizinin başlangıç kapasitesi 10'dur; `boyut` içerideki öğe sayısını tutar. Java jenerik tipte dizi oluşturmaya izin vermediği için önce ham `Oge` dizisi oluşturulur, sonra `(Oge<E>[])` ile dönüştürülür. Derleyici bu satır için "denetlenmeyen dönüşüm" uyarısı verir; dizi yalnız sınıfın içinde kullanıldığı için bu güvenlidir. Ekleme metodu şöyledir:

```
public void ekle(E veri, int p) {
    if (boyut == dizi.length)
        dizi = Arrays.copyOf(dizi, 2 * boyut);
    int i = boyut;
    while (i > 0 && dizi[i-1].oncelik < p) {
        dizi[i] = dizi[i - 1];  // sağa kaydır
        i--;
    }
    dizi[i] = new Oge<>(veri, p);
    boyut++;
}
```

Metot önce kapasiteyi denetler. Dizi doluysa `Arrays.copyOf` ile iki kat büyüklükte yeni bir dizi oluşturulur ve eski öğeler oraya kopyalanır. Ardından yer arama başlar. `i`, ilk boş hücre olan `boyut`'tan başlar ve sola doğru ilerler. Solundaki öğenin önceliği yeni öğenin önceliği `p`'den küçük olduğu sürece o öğe bir sağa kaydırılır. Döngü durduğunda `i`, yeni öğenin yeridir. Bu düzende en büyük öncelik `dizi[0]`'da durur. Çıkarma metodu bu öğeyi alır:

```
public E cikar() {
    if (bosMu()) return null;
    E sonuc = dizi[0].veri;
    for (int i = 1; i < boyut; i++)
        dizi[i - 1] = dizi[i];
    dizi[--boyut] = null;
    return sonuc;
}
```

`cikar`, boş kuyrukta `null` döndürür. Aksi hâlde `dizi[0]`'daki verinin bir kopyasını alır ve kalan öğeleri birer sola kaydırır. Son hücreye `null` yazılması, çıkan nesnenin referansının dizide kalıp çöp toplayıcıdan kaçmasını önler. `dizi[--boyut]` ifadesi önce `boyut`'u azaltır, sonra o indise yazar.

Bu sınıf, "en öncelikli öğeyi dizinin sonunda tut" önerisine uymaz: en büyük öncelik başta durur ve `cikar` bu yüzden *O(n)* sürer. `ekle` de en kötü durumda bütün öğeleri kaydırdığı için *O(n)*'dir. Kod, okunması kolay olsun diye bu düzeni seçmiştir. Öncelik dizinin sonunda tutulsaydı `cikar` tek satıra inerdi.

Koşuldaki `<` işaretinin bir anlamı daha vardır. Kaydırma yalnız solundaki öğe *kesin olarak* daha düşük öncelikliyse yapılır. Eşit öncelikli bir öğe kaydırılmaz ve yeni öğe onun sağına yerleşir. Böylece eşit öncelikli öğeler geliş sırasıyla durur ve geliş sırasıyla çıkar: kuyruk eşitlikte FIFO'dur.

11.2.6 Kod izleme: eşit öncelikte ekleme

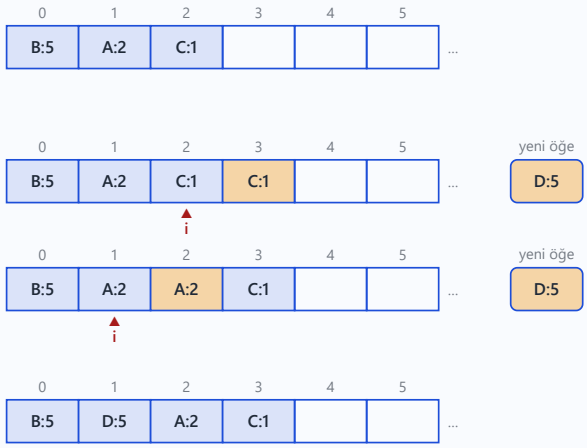
Boş kuyruğa `ekle(A, 2)`, `ekle(B, 5)`, `ekle(C, 1)` uygulandıktan sonra dizi `[B:5, A:2, C:1]` olur (her hücre "veri:öncelik" biçiminde yazıldı). Şimdi `ekle(D, 5)` çağrılıyor. D'nin önceliği B'ninkine eşittir; asıl soru D'nin B'nin önüne mi, arkasına mı gireceğidir.

```
public void ekle(E veri, int p) {
  if (boyut == dizi.length)
    dizi = Arrays.copyOf(dizi, 2 * boyut);
  int i = boyut;
  while (i > 0 && dizi[i-1].oncelik < p) {
    dizi[i] = dizi[i - 1];  // sağa kaydır
    i--;
  }
  dizi[i] = new Oge<>(veri, p);
  boyut++;
}
```

Adım	`i`	Koşul	`dizi`
kapasite denetimi	—	`3 == 10` ? hayır	[B:5, A:2, C:1, ·]
`i = boyut`	3	—	[B:5, A:2, C:1, ·]
karşılaştır	3	`dizi[2].oncelik = 1 < 5` : doğru	[B:5, A:2, C:1, ·]

Adım	`i`	Koşul	`dizi`
`dizi[3] = dizi[2]; i--`	2	—	[B:5, A:2, C:1, C:1]
karşılaştır	2	`dizi[1].oncelik = 2 < 5` : doğru	[B:5, A:2, C:1, C:1]
`dizi[2] = dizi[1]; i--`	1	—	[B:5, A:2, A:2, C:1]
karşılaştır	1	`dizi[0].oncelik = 5 < 5` : **yanlış**	döngü biter
`dizi[1] = (D, 5)`	1	—	[B:5, D:5, A:2, C:1]
`boyut++`	—	—	`boyut = 4`

Tablo 11.6: `[B:5, A:2, C:1]` dizisinde `ekle(D, 5)` izlemesi. "·" boş hücreyi gösterir.



Şekil 11.6: `ekle(D, 5)` 'in dört anı. (a) Başlangıç dizisi. (b) C bir sağa kaymış. (c) A da kaymış, `i = 1`. (d) D, B'nin arkasına yerleşmiş; dizi yine büyükten küçüğe sıralı.

Kritik adım `i = 1`'deki karşılaştırmadır: `5 < 5` yanlıştır ve B kaydırılmaz. D, B'nin hemen arkasına yerleşir. Kuyruktan iki kez `cikar()` yapılırsa önce B, sonra D çıkar. Kuyruk, eşit öncelikli B ile D arasında geliş sırasını korumuştur.

Koşul `<=` olarak yazılsaydı tablonun son karşılaştırması `5 <= 5`, yani doğru olurdu. B de bir sağa kayar ve D başa geçerdi: `[D:5, B:5, A:2, C:1]`. Dizi yine büyükten küçüğe sıralı kalır, ama eşit öncelikte yeni gelen öne geçer. Tek bir karakter, kuyruğun eşitlikteki davranışını FIFO'dan LIFO'ya çevirir.

Sık yapılan hata

Sıralı eklemede karşılaştırmanın katı mı (`<`) yoksa katı olmayan mı (`<=`) olacağına dikkat etmemek. İki seçenek de diziyi sıralı tutar; fark yalnız eşit anahtarlı öğelerin sırasında görülür ve bu yüzden testlerde kolayca gözden kaçar.

11.2.7 Maliyet karşılaştırması

İki liste gerçeklemesini ikili heap ile aynı tabloda karşılaştıralım.

İşlem	Sırasız liste	Sıralı liste	İkili heap
`size`, `isEmpty`	O(1)	O(1)	O(1)
`insert`	O(1)	O(n)	O(log n)
`min`	O(n)	O(1)	O(1)
`removeMin`	O(n)	O(1)	O(log n)
n öğeyle doldurma	O(n)	O(n log n)*	O(n)**

Tablo 11.7: Öncelikli kuyruk gerçeklemelerinin işlem maliyetleri.

> * Öğeler önce sıralanarak. ** Aşağıdan yukarıya heap oluşturma ile.

Tablonun son satırı, kuyruk boşken n öğenin hep birlikte verildiği durumu gösterir. Sırasız listede öğeleri olduğu gibi kopyalamak yeter. Sıralı listede öğeler önce sıralanır; bu, iyi bir sıralama algoritmasıyla $O(n \log n)$'dir. Heap'te ise öğeleri tek tek eklemek yerine diziyi yerinde düzenleyen özel bir algoritma $O(n)$ sürede heap kurar.

Tek tek işlemlerden çok, bir işlem dizisinin toplam maliyeti önemlidir. Boş bir kuyruğa n öğe eklenip ardından n kez `removeMin` yapıldığını düşünelim.

- **Sırasız liste:** $n \cdot O(1) + n \cdot O(n) = O(n^2)$. Eklemeler ucuzdur, ama her çıkarma kalan bütün öğeleri tarar.
- **Sıralı liste:** $n \cdot O(n) + n \cdot O(1) = O(n^2)$. Bu kez pahalı olan eklemelerdir.
- **İkili heap:** $2n \cdot O(\log n) = O(n \log n)$.

Farkın büyüklüğünü bir sayıyla görelim. $n = 10^6$ için $n^2 = 10^{12}$, $n \log_2 n \approx 10^6 \cdot 19{,}93 \approx 2 \cdot 10^7$'dir. Oran $n / \log_2 n \approx 50.000$'dir. Heap'li bir programın bir saniyede bitirdiği iş, liste kullanan programda yaklaşık 50.000 saniye, yani yaklaşık on dört saat sürer. Asimptotik karmaşıklık bu tür farkları önceden görmeye yarar.

11.3 İkili heap

Liste gerçeklemeleri iki uç noktadır. Sırasız liste hiç düzen tutmaz ve ekleme bedava olur; sıralı liste tam düzen tutar ve çıkarma bedava olur. İkili heap ikisinin arasında durur: *kısmi* bir düzen tutar. Bu düzen en öncelikli öğenin hep aynı yerde olmasını garanti edecek kadar güçlü, ama her işlemde yalnız birkaç öğenin yer değiştirmesine izin verecek kadar gevşektir.

11.3.1 İki özellik

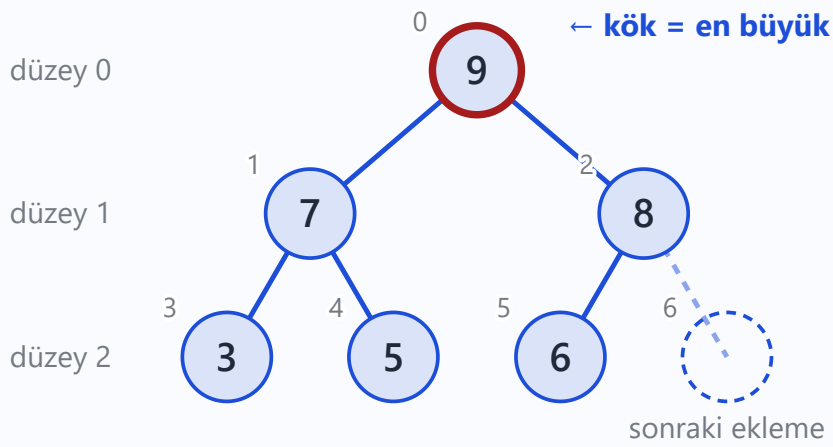
İkili heap (*binary heap*), öncelikli kuyruğu gerçeklemek için en yaygın kullanılan veri yapısıdır. Öğeler bir ikili ağacın düğümlerinde tutulur ve ağaç aynı anda iki koşulu sağlar.

Tanım 11.3

Max-heap, aşağıdaki iki özelliği birlikte sağlayan ikili ağaçtır:

1. **Heap-sıra özelliği** (*heap-order property*): her düğümün anahtarı, çocuklarının anahtarlarından büyük ya da onlara eşittir.
2. **Tam ikili ağaç özelliği** (*complete binary tree*): son düzey dışındaki bütün düzeyler doludur; son düzeydeki düğümler soldan sağa, aralarında boşluk olmadan dizilir.

Birinci özellikte "büyük ya da eşit" yerine "küçük ya da eşit" yazılırsa **min-heap** elde edilir.



Şekil 11.7: Altı düğümlü bir max-heap. Kök 9 en büyük anahtardır. Son düzey soldan sağa dolmuştur; sıradaki ekleme, 8'in sağ çocuğu olan 6 numaralı konuma yapılır.

Şekil 11.7 bir max-heap gösteriyor. Kök 9'dur; çocukları 7 ve 8'dir. 7'nin çocukları 3 ve 5, 8'in tek çocuğu 6'dır. Her ebeveyn çocuklarından büyüktür: $9 \geq 7, 8$; $7 \geq 3, 5$; $8 \geq 6$. Son düzey soldan sağa dolmuştur ve tek boşluk en sağdadır.

Heap-sıra özelliği yalnız ebeveyn ile çocuk arasında bir şey söyler. Ama bu yerel kural zincirleme olarak uygulanınca güçlü bir sonuç verir: bir düğümün anahtarı çocuğundan, çocuğun anahtarı torunundan büyüktür ve bu böyle sürer. Yani her düğüm, kendi alt ağacındaki bütün düğümlerden büyük ya da onlara eşittir. Kök bütün ağacın alt ağacı olduğundan **en büyük anahtar köktedir**. En öncelikli öğeye bakmak bu yüzden $O(1)$'dir.

Aynı kural, kardeşler arasında hiçbir şey söylemez. Şekil 11.7'de 7 ile 8 yer değiştirse ağaç yine heap olur: 9 iki çocuğundan da büyüktür ve 8 ile 7 kendi çocuklarından büyük kalır. Heap'in sıralı bir yapı olmamasının nedeni budur. Kökten başka hiçbir düğümün konumu, onun bütün öğeler içindeki sırası

hakkında kesin bilgi vermez.

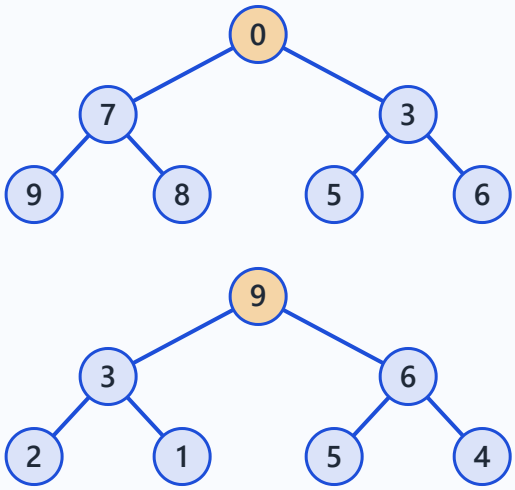
İkinci özellik ağacın biçimini sınırlar. Tam ikili ağaç olabilecek en "sıkışık" ikili ağaçtır: belli sayıda düğümle olabilecek en kısa ağaç budur. Bu özelliğin iki önemli sonucu vardır. Birincisi, ağacın yüksekliği $\lfloor \log_2 n \rfloor$'dir. İkincisi, ağaç bir dizide boşluksuz tutulabilir; işaretçi gerekmez.

Bu iki özellik birlikte heap işlemlerinin hızını açıklar. Ekleme ve çıkarma, kökten bir yaprağa uzanan *tek bir yol* üzerinde çalışır. Yolun uzunluğu en çok ağacın yüksekliği kadardır ve yükseklik logaritmik olduğu için iki işlem de *O(log n)*'dir.

> **Sık yapılan hata**
>
> Heap'i ikili arama ağacıyla karıştırmak. İkili arama ağacında sol alt ağaçtaki her anahtar düğümden küçük, sağdakiler büyüktür; ağaçta bir değeri aramak bu yüzden hızlıdır. Heap'te sol ve sağ çocuk arasında hiçbir sıra yoktur. Heap'te belli bir değeri aramak bütün düğümlere bakmayı gerektirir ve *O(n)*'dir.

11.3.2 Min-heap ve max-heap

İki heap türü, heap-sıra özelliğindeki eşitsizliğin yönüyle ayrılır. **Min-heap**'te her ebeveynin anahtarı çocuklarınınkinden küçük ya da eşittir: *anahtar(ebeveyn) ≤ anahtar(çocuk)*. En küçük anahtar köktedir ve min-heap min-öncelikli kuyruğu gerçekler. **Max-heap**'te eşitsizlik terstir: *anahtar(ebeveyn) ≥ anahtar(çocuk)*. Kökte en büyük anahtar durur.



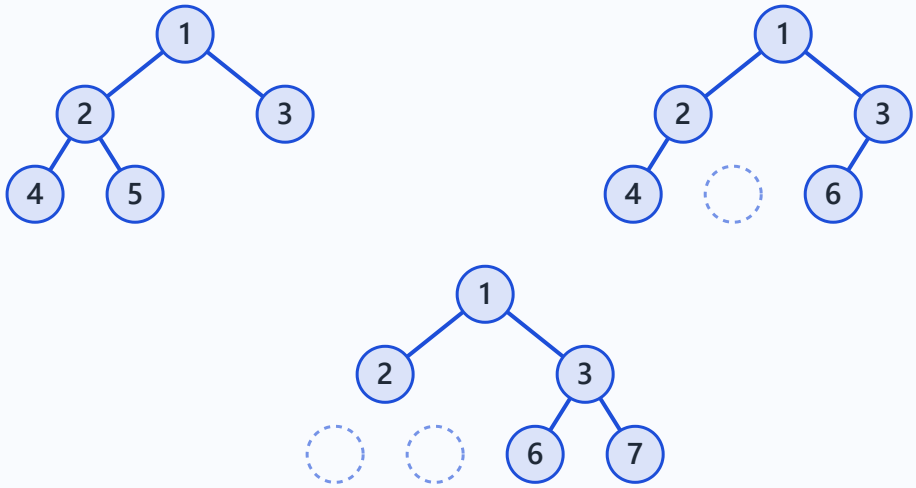
Şekil 11.8: Solda min-heap: kök 0; çocukları 7 ve 3; 7'nin çocukları 9 ve 8; 3'ün çocukları 5 ve 6. Sağda max-heap: kök 9; çocukları 3 ve 6; 3'ün çocukları 2 ve 1; 6'nın çocukları 5 ve 4.

Şekil 11.8'deki min-heap'te 7 ile 3 kardeştir ve 7 > 3'tür. Bu bir sorun değildir: sıra yalnız ebeveyn ile çocuk arasında geçerlidir. Aynı ağaçta 7, 3'ün çocukları olan 5 ve 6'dan da büyüktür. Farklı dallardaki düğümler arasında herhangi bir ilişki beklenmez.

Heap-sıra özelliğinin geometrik bir okunuşu da vardır: kökten herhangi bir yaprağa giden her yol sıralıdır. Min-heap'te yol boyunca anahtarlar artar; max-heap'te azalır. Şekildeki max-heap'te 9 → 3 → 2, 9 → 3 → 1, 9 → 6 → 5 ve 9 → 6 → 4 yollarının dördü de azalan sıradadır. Heap'i, kökte birleşen sıralı yollardan oluşan bir demet gibi düşünmek mümkündür. Ekleme ve çıkarma da bu yollardan yalnız birini düzeltir.

11.3.3 Tam ikili ağaç ve yükseklik

Tam ikili ağaç tanımı ilk okumada "mükemmel ağaç" ile karıştırılabilir. Fark, son düzeydedir. Tam ağaçta son düzeyin dolu olması gerekmez; yalnız düğümlerin *soldan başlayarak boşluksuz* dizilmesi gerekir. Şekil 11.9 üç ağacı karşılaştırıyor.



Şekil 11.9: Soldaki ağaç tamdır: son düzey soldan dolu. Ortadaki tam değildir: son düzeyde 4 ile 6 arasında boşluk var. Sağdaki tam değildir: son düzey sola yaslı değil, düğümler en sağdaki konumlarda.

Soldaki ağaçta 1 kök, 2 ve 3 onun çocukları, 4 ve 5 ise 2'nin çocuklarıdır. Son düzey 2'nin altında soldan dolmuş ve 3'ün altı boş kalmıştır; ağaç tamdır. Ortadaki ağaçta 2'nin sağ çocuğu yokken 3'ün sol çocuğu vardır. Son düzeyde bir boşluk oluşmuştur. Sağdaki ağaçta son düzeydeki iki düğüm 3'ün altındadır; 2'nin altındaki konumlar boştur. Bu ağaçların ikisi de heap olamaz.

Tanım 11.4

Kökün düzeyi 0, çocuklarının düzeyi 1 olmak üzere her düğümün **düzeyi** (*level*) ebeveyninin düzeyinden bir fazladır. Ağacın **yüksekliği** (*height*) en derin yaprağın düzeyidir. d. düzeyde en çok 2^d düğüm bulunabilir. Yüksekliği h olan bir **tam ikili ağaçta** 0, 1, …, $h - 1$ düzeylerinin hepsi doludur ve h. düzeydeki düğümler en soldaki konumlardadır.

Teorem 11.1 (Tam ikili ağacın yüksekliği)

$n \geq 1$ düğümlü bir tam ikili ağacın yüksekliği $h = \lfloor \log_2 n \rfloor$'dir.

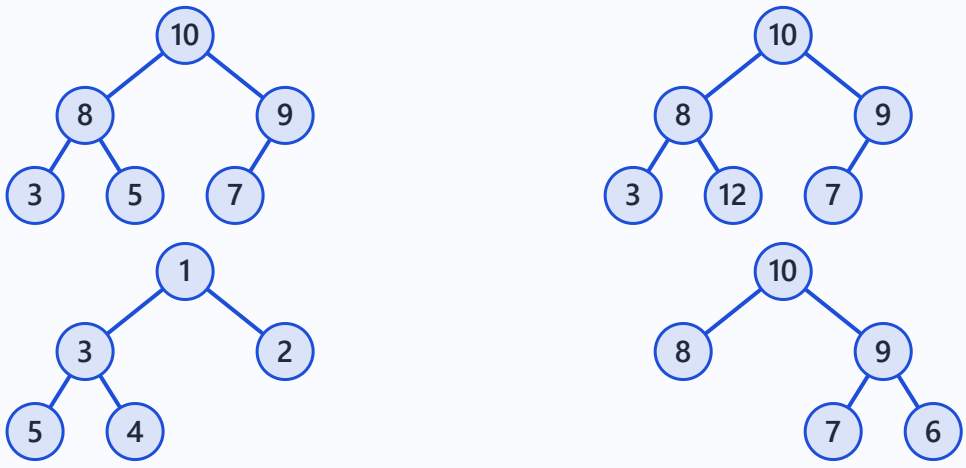
İspat. Yüksekliği h olan tam ağaçta 0'dan $h - 1$'e kadar bütün düzeyler doludur. Bu düzeylerdeki düğüm sayısı $1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$'dir. Son düzeyde en az 1, en çok 2^h düğüm vardır. Toplam düğüm sayısı için şu eşitsizlik çıkar: $2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1}$. Taraflardan iki tabanında logaritma alınırsa $h \leq \log_2 n < h + 1$ bulunur. h bir tam sayı olduğu için bu, $h = \lfloor \log_2 n \rfloor$ demektir. ■

Teoremi iki örnekle sınayalım. $n = 7$ için $\log_2 7 \approx 2{,}81$ ve $h = 2$'dir: kök, iki çocuk ve dört torun tam üç düzey doldurur. $n = 1000$ için $2^9 = 512 \leq 1000 < 1024 = 2^{10}$ olduğundan $h = 9$'dur. Bin öğeli bir heap'te kökten en derin yaprağa giden yol yalnız 9 kenar uzunluğundadır; bir milyon öğede bu sayı 19'a çıkar. Logaritmik büyümenin pratik anlamı budur: öğe sayısı bin kat artarken yol yalnız 10 adım uzar.

Tam ağaç özelliğinin heap için ikinci önemli sonucu, yeni düğümün yerinin hiçbir zaman tartışmalı olmamasıdır. Yeni öğe her zaman “son düzeydeki en soldaki boş yere” gelir. Son düzey doluysa yeni bir düzey açılır ve öğe onun en soluna yerleşir. Dizi gösteriminde bu konum tam olarak `n` indisidir. Silmede de durum benzerdir: tam ağaçtan bir düğüm çıkarmanın biçimi bozmayan tek yolu, son düzeyin en sağındaki düğümü, yani dizinin son öğesini kaldırmaktır.

11.3.4 Heap mi, değil mi?

Bir ağacın heap olup olmadığını anlamak için iki özellik ayrı ayrı denetlenir. Önce biçime bakılır: ağaç tam mı? Sonra her ebeveyn–çocuk çiftine bakılır: eşitsizlik hep aynı yönde mi?



Şekil 11.10: Dört ağaç: (a) üst sol, (b) üst sağ, (c) alt sol, (d) alt sağ.

Örnek 11.3

Şekil 11.10'daki her ağaç için max-heap mi, min-heap mi, yoksa heap değil mi olduğuna karar verelim.

1. **(a)** Kök 10; çocukları 8 ve 9; 8'in çocukları 3 ve 5; 9'un sol çocuğu 7. Ağaç tamdır: son düzeyde 3, 5, 7 soldan sağa boşluksuz dizilmiş. Ebeveyn–çocuk çiftleri: 10 ≥ 8, 10 ≥ 9, 8 ≥ 3, 8 ≥ 5, 9 ≥ 7. Hepsi aynı yönde: **max-heap**.
2. **(b)** Biçim (a) ile aynıdır, ama 8'in sağ çocuğu 12'dir. 12 > 8 olduğu için heap-sıra özelliği bozuktur. Kök 10 ≥ 8 olduğundan ağaç min-heap de olamaz: **heap değil**.
3. **(c)** Kök 1; çocukları 3 ve 2; 3'ün çocukları 5 ve 4. Ağaç tamdır. 1 ≤ 3, 1 ≤ 2, 3 ≤ 5, 3 ≤ 4: **min-heap**.
4. **(d)** Kök 10; çocukları 8 ve 9; 8'in çocuğu yok, 9'un çocukları 7 ve 6. Bütün ebeveynler çocuklarından büyüktür, ama son düzey sola yaslı değildir: 8'in altı boşken 9'un altında düğüm var. Tam ağaç özelliği bozuk: **heap değil**.

(d) ağacı önemli bir dersi gösterir: heap-sıra özelliği tek başına yetmez. Biçim bozukken dizi gösteriminde boşluk oluşur. (d) ağacı dizide `[10, 8, 9, _, _, 7, 6]` biçiminde, iki boş hücreyle yazılmak zorundadır. Heap'in indis formülleri ve "yeni öğe `n` indisine gelir" kuralı boşluksuz dizi varsayımına dayanır.

11.3.5 Kod izleme: heapMi()

Bir dizinin max-heap olup olmadığını denetleyen kod kısadır. Dizi gösteriminde ağaç kendiliğinden tamdır, çünkü dizide boşluk yoktur. Bu yüzden yalnız heap-sıra özelliği denetlenir. Dizideki `i` indisli düğümün ebeveyni `(i - 1) / 2` indisindedir (Tanım 11.5).

```
static boolean heapMi(int[] a) {  // max-heap mi?
  for (int i = 1; i < a.length; i++) {
    int e = (i - 1) / 2;          // ebeveyn
    if (a[e] < a[i]) return false; // sıra bozuk
  }
  return true;
}
```

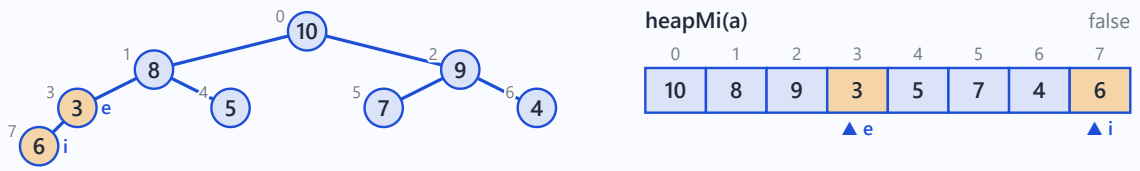
Döngü kök dışındaki her düğümü, yani `i = 1`'den `n - 1`'e kadar her indisi bir kez ziyaret eder. Her düğüm yalnız kendi ebeveyniyle karşılaştırılır. Bir ebeveyn çocuğundan küçükse sıra bozuktur ve metot hemen `false` döndürür. Döngü sona kadar giderse bütün ebeveyn–çocuk çiftleri denetlenmiştir ve dizi bir max-heap'tir.

Her ebeveyn–çocuk çifti tam bir kez denetlenir, çünkü her düğümün tek bir ebeveyni vardır. Ayrıca çocuklar üzerinden gitmek, ebeveynler üzerinden gidip iki çocuğu denetlemekten daha basit bir döngü verir: sağ çocuğun var olup olmadığını sorgulamak gerekmez.

Kodu `a = [10, 8, 9, 3, 5, 7, 4, 6]` dizisi üzerinde izleyelim.

`i`	`e = (i - 1) / 2`	Karşılaştırma	Sonuç
1	0	`a[0] = 10 ≥ a[1] = 8`	devam
2	0	`a[0] = 10 ≥ a[2] = 9`	devam
3	1	`a[1] = 8 ≥ a[3] = 3`	devam
4	1	`a[1] = 8 ≥ a[4] = 5`	devam
5	2	`a[2] = 9 ≥ a[5] = 7`	devam
6	2	`a[2] = 9 ≥ a[6] = 4`	devam
7	3	`a[3] = 3 < a[7] = 6`	`return false`

Tablo 11.8: `heapMi([10, 8, 9, 3, 5, 7, 4, 6])` izlemesi.



Şekil 11.11: `i = 7` 'de sıra bozukluğu bulunur: 6, ebeveyni 3'ten büyüktür.

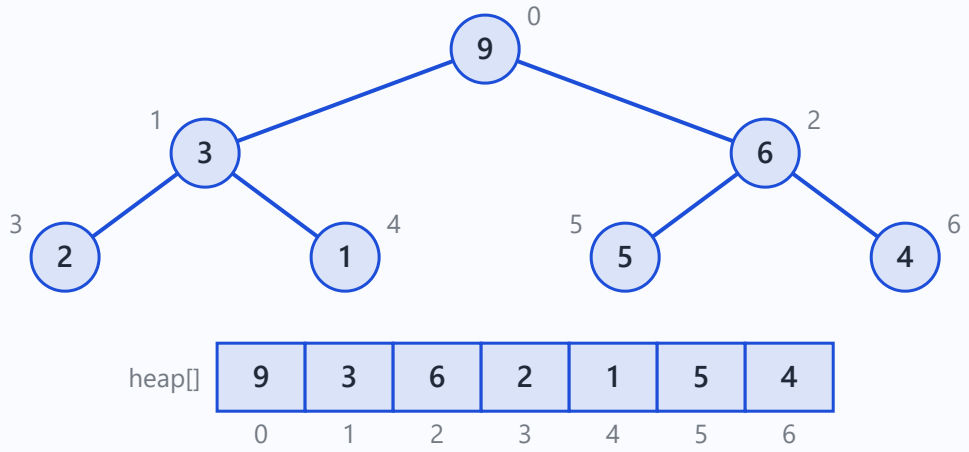
İlk yedi öğe, yani `[10, 8, 9, 3, 5, 7, 4]` geçerli bir max-heap'tir; Java ile çalıştırıldığında bu yedi öğelik dizi için metot `true` döndürür. Sona eklenen 6 ise ebeveyni 3'ten büyüktür ve heap-sıra özelliğini bozar. Bu durum, ekleme işleminin neden ek bir düzeltme adımına ihtiyaç duyduğunu da gösteriyor: yeni öğeyi sona yazmak biçimi korur, sırayı korumayabilir.

En kötü durumda, yani dizi gerçekten bir heap olduğunda, döngü $n - 1$ karşılaştırma yapar. `heapMi` 'nin zaman maliyeti $O(n)$, ek bellek maliyeti $O(1)$'dir. Daha hızlı bir denetim mümkün değildir: herhangi bir ebeveyn–çocuk çifti atlanırsa o çiftteki bozukluk gözden kaçabilir.

Boş ve tek öğeli dizilerde döngü hiç dönmez ve metot `true` döndürür. Bu doğru bir sonuçtur: hiç ebeveyn–çocuk çifti olmayan bir ağaç, heap-sıra özelliğini kendiliğinden sağlar. Metodun min-heap sürümü için koşulu ters çevirmek yeter: `if (a[e] > a[i]) return false;` .

11.3.6 Dizi gösterimi

Ağaçlar çoğunlukla düğüm nesneleri ve aralarındaki referanslarla temsil edilir. Heap'te buna gerek yoktur. Düğümler kökten başlayarak düzey düzey, her düzeyde soldan sağa (*level order*, düzey sırası) bir diziye yazılır. Kök `heap[0]` 'a, onun çocukları `heap[1]` ve `heap[2]` 'ye, torunlar `heap[3]` 'ten `heap[6]` 'ya gider ve bu böyle sürer.



Şekil 11.12: Max-heap ve dizi gösterimi. Düğümlerin yanındaki küçük sayılar dizideki indislerdir. Dizi: [9, 3, 6, 2, 1, 5, 4].

Şekil 11.12 yedi düğümlü bir max-heap'i ve onun dizisini gösteriyor. Ağaç tam olduğu için düzey sırasıyla yazılan dizide boşluk kalmaz: `n` öğe tam olarak `heap[0]`'dan `heap[n - 1]`'e kadar olan hücrelere sığar. Ebeveyn ve çocuk bağlantıları için referans saklamak gerekmez; bunlar indislerden hesaplanır.

Dizi gösteriminin üç pratik üstünlüğü vardır. Birincisi bellektir: düğüm başına iki çocuk referansı yerine yalnız öğenin kendisi saklanır. İkincisi hızdır: ardışık bellek hücreleri işlemcinin önbelleğine bloklar hâlinde gelir ve dizide gezinmek, bellekte dağınık düğümleri izlemekten çok daha hızlıdır. Üçüncüsü sadeliktir: "son öğe" ve "ilk boş konum" tek bir sayaçla, `n` ile bilinir.

Bu notta indisler 0'dan başlar ve kök `heap[0]`'dadır. Bazı kitaplar indisleri 1'den başlatır; iki seçenek eşdeğerdir ve formülleri biraz değişir.

11.3.7 Ebeveyn ve çocuk indisleri

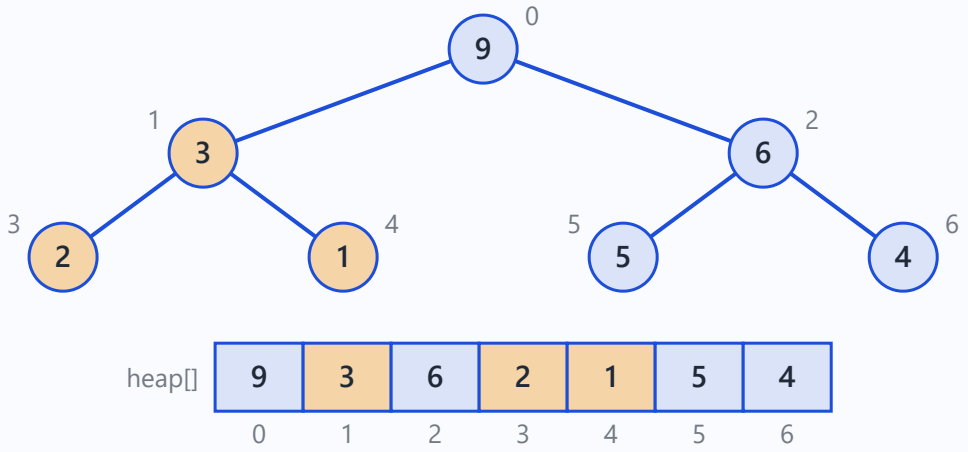
Düzey sırası düzeninde bir düğümün çocuklarının ve ebeveyninin indisi basit formüllerle bulunur.

Tanım 11.5

0 tabanlı dizi gösteriminde `i` indisli düğüm için:

- sol çocuk: $sol(i) = 2i + 1$,
- sağ çocuk: $sağ(i) = 2i + 2$,
- ebeveyn ($i \geq 1$ için): $ebeveyn(i) = \lfloor (i - 1) / 2 \rfloor$.

Çocuk indisi n'den küçük değilse o çocuk yoktur. Özel olarak $2i + 1 \geq n$ ise `i` bir yapraktır.



Şekil 11.13: İndis 1'deki 3'ün çocukları indis 3 ve 4'tedir: 2 ve 1.

Formüllerin neden doğru olduğunu görmek için düzeylerin dizide nereden başladığına bakalım. *d*. düzey $2^d - 1$ indisinden başlar: 0. düzey 0'dan, 1. düzey 1'den, 2. düzey 3'ten, 3. düzey 7'den. Bir düzeyin *j*'inci düğümünün (0'dan sayarak) çocukları, bir alt düzeyin $2j$'inci ve $2j + 1$'inci düğümleridir, çünkü her düğüm soldan sağa sırayla iki çocuk alır. İndis olarak yazılırsa düğüm $i = 2^d - 1 + j$, sol çocuğu $2^{d+1} - 1 + 2j = 2(2^d - 1 + j) + 1 = 2i + 1$'dir. Sağ çocuk bir sonraki hücredir. Ebeveyn formülü bu ilişkinin tersidir: $2i + 1$ ve $2i + 2$'den 1 çıkarıp 2'ye bölünce aşağı yuvarlanmış değer iki durumda da *i* olur.

Örnek 11.4

Şekil 11.13'teki `[9, 3, 6, 2, 1, 5, 4]` dizisinde formülleri sınayalım.

1. `i = 1` (3): çocuklar $2 \cdot 1 + 1 = 3$ ve $2 \cdot 1 + 2 = 4$, yani 2 ve 1.
2. `i = 2` (6): çocuklar 5 ve 6 indislerinde, yani 5 ve 4.
3. `i = 6` (4): ebeveyn $\lfloor (6 - 1)/2 \rfloor = \lfloor 2{,}5 \rfloor = 2$, yani 6. `i = 5` için de $\lfloor 4/2 \rfloor = 2$. İki kardeşin ebeveyni aynıdır.
4. `i = 4` (1): ebeveyn $\lfloor 3/2 \rfloor = 1$, yani 3. Çocuk adayı $2 \cdot 4 + 1 = 9 \geq n = 7$: 4 indisli düğüm yapraktır.
5. Daha büyük bir örnek: indis 13'ün ebeveyni $\lfloor 12/2 \rfloor = 6$, çocukları 27 ve 28'dir.

Java'da tam sayı bölmesi pozitif sayılarda zaten aşağı yuvarlar. Bu yüzden ebeveyn formülü kodda doğrudan `(i - 1) / 2` olarak yazılır. Çocuk formülleri de yalnız toplama ve çarpma içerir. Bu işlemler sabit sürede yapılır; heap'te bir düğümden komşusuna geçmek, bağlı yapıda bir referansı izlemek kadar ucuzdur.

1 tabanlı gösterim. Bazı kitaplar `heap[0]`'ı boş bırakır ve kökü `heap[1]`'e koyar. O zaman *k* indisli düğümün çocukları $2k$ ve $2k + 1$, ebeveyni $\lfloor k/2 \rfloor$'dir. Formüller biraz daha kısadır, ama bir hücre boşa gider. İki gösterim eşdeğerdir; önemli olan birini seçip baştan sona tutarlı kullanmaktır.

Sık yapılan hata

0 tabanlı ve 1 tabanlı formülleri karıştırmak. Kök `heap[0]` 'dayken sol çocuk için `2 * i` yazılırsa kökün sol çocuğu yine kökün kendisi çıkar ($2 \cdot 0 = 0$) ve ağaç bozulur. 0 tabanlı gösterimde formüller $2i + 1$, $2i + 2$, $(i - 1)/2$; 1 tabanlıda $2i$, $2i + 1$, $i/2$'dir.

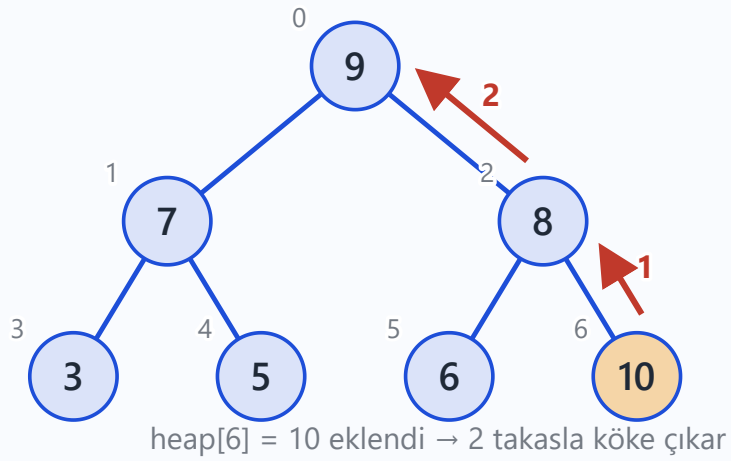
11.4 Ekleme ve silme

Heap'in iki temel işlemi aynı kalıbı izler. Önce ağacın *biçimi* korunur: öğe, tam ağaç özelliğini bozmayacak tek yere konur ya da oradan alınır. Bu adım heap-sıra özelliğini bir noktada bozabilir. İkinci adım bu yerel bozukluğu, öğeyi tek bir yol boyunca yukarı ya da aşağı taşıyarak onarır. Yukarı taşımaya **up-heap** (yukarı süzme), aşağı taşımaya **down-heap** (aşağı süzme) denir. Bu kesimdeki kodlar max-heap içindir.

11.4.1 Ekleme: up-heap

Max-heap'e yeni bir öğe eklemek üç adımdan oluşur:

1. **Biçimi koru.** Yeni öğe ağacın ilk boş konumuna, yani `heap[n]` 'e yazılır ve `n` bir artar. Ağaç tam kalır.
2. **Bozukluğu bul.** Heap-sıra özelliği yalnız yeni öğe ile ebeveyni arasında bozulmuş olabilir. Yeni öğe ebeveyninden büyükse ikisi yer değiştirir.
3. **Tekrarla.** Öğe bir düzey yukarı çıkınca aynı soru yeni ebeveyn için sorulur. Öğe köke ulaşınca ya da ebeveyni kendisinden büyük veya ona eşit olunca durulur.



Şekil 11.14: Max-heap [9, 7, 8, 3, 5, 6]'ya 10 ekleniyor. 10 indis 6'ya yazılır, önce 8 ile, sonra 9 ile yer değiştirip iki takasla köke çıkar.

Şekil 11.14 bir örnek gösteriyor. 10, 8'in sağ çocuğu olarak indis 6'ya yazılır. Ebeveyni 8, 10'dan küçüktür: takas yapılır ve 10 indis 2'ye çıkar. Yeni ebeveyn kök 9'dur; $9 < 10$ olduğu için bir takas daha yapılır. 10 artık köktedir ve işlem biter. Sonuç `[10, 7, 9, 3, 5, 6, 8]`'dir.

Bu işlemin adları kaynaktan kaynağa değişir: up-heap, *bubble-up*, *sift-up*, *swim*. Hepsi aynı şeyi anlatır. Bu notun kodunda metodun adı `yuzdur`'dur: hafif bir cismin suyun yüzeyine çıkması gibi büyük anahtar köke doğru yükselir.

Teorem 11.2 (Up-heap'in doğruluğu ve maliyeti)

Geçerli bir max-heap'in sonuna bir öğe eklenip up-heap uygulanırsa sonuç yine geçerli bir max-heap'tir. İşlem en çok $\lfloor \log_2 n \rfloor$ takas yapar ve $O(\log n)$ zamanda biter.

İspat. Up-heap sırasında yükselen öğeye x diyelim. Her adımda heap-sıra özelliği x ile ebeveyni arasındaki *tek çift* dışında her yerde geçerlidir. Başlangıçta bu doğrudur, çünkü ekleme öncesi ağaç bir heap'ti ve yalnız yeni yaprak eklendi. Bir takası inceleyelim: ebeveyn p, x'ten küçük olduğu için yer değiştiriyorlar. p bir düzey aşağı iner ve x'in eski çocuklarının ebeveyni olur. Bu çocuklar ekleme öncesinde p'nin torunlarıydı ve heap özelliği gereği p'den büyük değildi. p'nin öbür çocuğu da zaten p'den büyük değildi. Yani aşağı inen p altındaki her düğümden büyük ya da eşittir. x ise yeni yerinde p'den ve dolayısıyla p'nin eski çocuğundan büyüktür. Böylece tek bozuk çift bir düzey yukarı taşınmış olur. Up-heap köke ulaşınca ya da ebeveyn $\geq x$ olunca o son çift de düzelir. Her takas öğeyi bir düzey yukarı taşır ve yükseklik $\lfloor \log_2 n \rfloor$ olduğundan takas sayısı bunu aşamaz. ■

İspatın özü tek cümleyle şöyle söylenebilir: yer değiştirmede aşağı inen ebeveyn, zaten diğer çocuğundan büyüktü. Bu yüzden takas, yukarıdaki bozukluğu düzeltirken aşağıda yeni bir bozukluk yaratmaz.

11.4.2 Kod: ekle() ve yuzdur()

Max-heap sınıfı bir `int` dizisi ve bir öğe sayacı tutar. Ekleme metodu ve up-heap şöyledir:

```
public class MaxHeap {
  private int[] heap = new int[4];
  private int n = 0;                // öğe sayısı

public void ekle(int x) {
  if (n == heap.length) buyut(2 * heap.length);
  heap[n] = x;  n++;              // ilk boş konum
  yuzdur(n - 1);
}
private void yuzdur(int i) {      // up-heap
  while (i > 0) {
    int ebeveyn = (i - 1) / 2;
    if (heap[ebeveyn] >= heap[i]) return;
    yerDegistir(i, ebeveyn);  i = ebeveyn;
  }
}
private void yerDegistir(int a, int b) {
  int t = heap[a];  heap[a] = heap[b];  heap[b] = t;
}
}  // MaxHeap
```

Sınıfın başlangıç kapasitesi 4'tür. `ekle` önce dizinin dolu olup olmadığına bakar: `n == heap.length` ise `buyut` kapasiteyi iki katına çıkarır. Ardından `x` ilk boş konum olan `heap[n]`'e yazılır, `n` artırılır ve yeni öğenin indisi `n - 1` ile `yuzdur` çağrılır.

`yuzdur(i)`, `i` kökten farklı olduğu sürece döner. Her turda ebeveynin indisi `(i - 1) / 2` ile hesaplanır. Ebeveyn öğeden büyük ya da ona eşitse sıra sağlanmıştır ve metot `return` ile hemen biter. Aksi hâlde `yerDegistir` iki hücreyi takas eder ve `i` ebeveynin indisine çıkar. `yerDegistir(a, b)` üç atamalık bir yardımcı metottur: geçici bir değişken üzerinden `heap[a]` ile `heap[b]`'yi değiştirir.

Döngünün iki çıkışı vardır. Birincisi `while` koşuludur: `i = 0` olduğunda öğe köktedir ve ebeveyni yoktur. İkincisi döngü içindeki `return`'dür: ebeveyn ≥ öğe olduğunda daha yukarıya bakmaya gerek yoktur, çünkü yukarıdaki yol zaten sıralıdır. Koşulun `>=` olması, eşit anahtarlarda gereksiz takası önler.

Maliyet: `yuzdur` her turda sabit iş yapar ve en çok $\lfloor \log_2 n \rfloor$ tur döner; $O(\log n)$'dir. `ekle`'nin kendisi, `buyut` çağrısı dışında sabit iştir. `buyut` ise nadiren çalışır ve maliyeti eklemelere dağıtılınca amortize $O(1)$'dir. Ekleme bu yüzden amortize $O(\log n)$'dir.

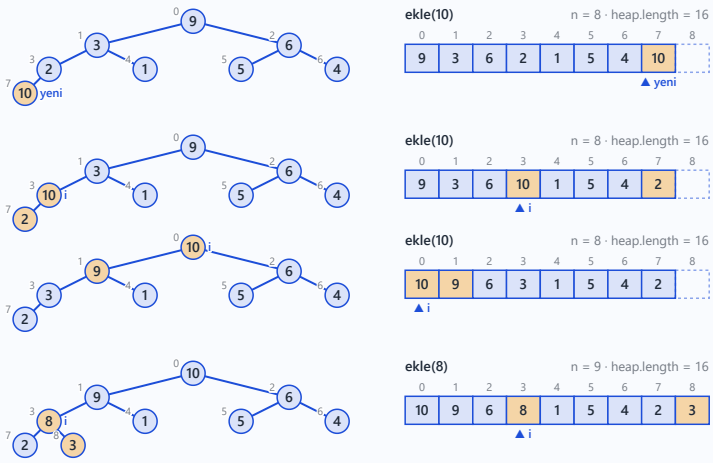
11.4.3 Kod izleme: ekle(10), ekle(8)

Başlangıçta heap `[9, 3, 6, 2, 1, 5, 4]`'tür: `n = 7`, kapasite 16. Her ebeveyn çocuklarından büyük ya da eşittir ve en büyük öğe 9 köktedir. Ardı ardına `ekle(10)` ve `ekle(8)` çağrılıyor.

Adım	`i`	`ebeveyn`	Karşılaştırma	Dizi
`ekle(10)`: `heap[7] = 10`	7	—	—	[9, 3, 6, 2, 1, 5, 4, 10], `n = 8`

Adım	`i`	`ebeveyn`	Karşılaştırma	Dizi
tur 1	7	3	`heap[3] = 2 < 10` : takas	[9, 3, 6, 10, 1, 5, 4, 2]
tur 2	3	1	`heap[1] = 3 < 10` : takas	[9, 10, 6, 3, 1, 5, 4, 2]
tur 3	1	0	`heap[0] = 9 < 10` : takas	[10, 9, 6, 3, 1, 5, 4, 2]
döngü sonu	0	—	`i > 0` yanlış	10 kökte
`ekle(8)` : `heap[8] = 8`	8	—	—	[10, 9, 6, 3, 1, 5, 4, 2, 8], `n = 9`
tur 1	8	3	`heap[3] = 3 < 8` : takas	[10, 9, 6, 8, 1, 5, 4, 2, 3]
tur 2	3	1	`heap[1] = 9 ≥ 8` : `return`	[10, 9, 6, 8, 1, 5, 4, 2, 3]

Tablo 11.9: `[9, 3, 6, 2, 1, 5, 4]` üzerinde `ekle(10)` ve `ekle(8)` izlemesi.



Şekil 11.15: (a) 10, `heap[7]` 'ye yazıldı. (b) İlk takastan sonra 10 indis 3'te. (c) Üç takasla 10 köke çıktı. (d) `ekle(8)` 'de tek takas: 8 indis 3'te, ebeveyni 9 ondan büyük.

`ekle(10)` en kötü durumu gösterir: yeni öğe bütün öğelerden büyüktür ve yapraktan köke kadar çıkar. Sekiz öğeli heap'in yüksekliği $\lfloor \log_2 8 \rfloor = 3$'tür ve tam 3 takas yapılmıştır. `ekle(8)` ise yolun yarısında durur: 8, 3 ile yer değiştirdikten sonra ebeveyni 9 ile karşılaşır ve $9 \geq 8$ olduğu için metot döner. İki eklemede toplam 4 takas yapılmıştır.

Tabloda dikkat çekici bir ayrıntı vardır: `ekle(8)` 'de ilk karşılaştırılan ebeveyn 3'tür. Bu 3, `ekle(10)` sırasında indis 1'den indis 3'e inmiş olan öğedir. Her up-heap, kendi yolundaki öğeleri bir düzey aşağı iter; sonraki eklemeler bu yeni düzen üzerinde çalışır.

11.4.4 Boş heap'e art arda ekleme

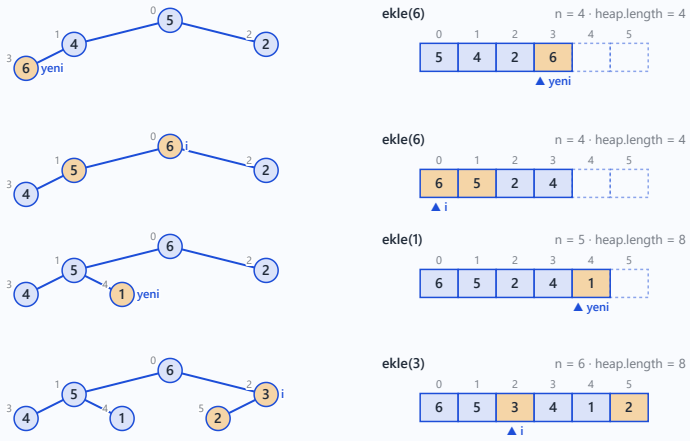
Heap'in boştan nasıl kurulduğunu görmek için kapasitesi 4 olan boş bir max-heap'e sırasıyla 4, 5, 2, 6, 1, 3 ekleyelim. Tablo 11.10 her eklemeden sonraki durumu veriyor.

Çağrı	Yazılan yer	Up-heap adımları	Sonuç dizisi	Kapasite
`ekle(4)`	`heap[0]`	`i = 0` : kök, döngüye girilmez	[4]	4

Çağrı	Yazılan yer	Up-heap adımları	Sonuç dizisi	Kapasite
`ekle(5)`	`heap[1]`	ebeveyn 0: 4 < 5, takas; `i = 0`	[5, 4]	4
`ekle(2)`	`heap[2]`	ebeveyn 0: 5 ≥ 2, `return`	[5, 4, 2]	4
`ekle(6)`	`heap[3]`	ebeveyn 1: 4 < 6, takas; ebeveyn 0: 5 < 6, takas	[6, 5, 2, 4]	4
`ekle(1)`	`heap[4]`	önce `buyut(8)`; ebeveyn 1: 5 ≥ 1, `return`	[6, 5, 2, 4, 1]	8
`ekle(3)`	`heap[5]`	ebeveyn 2: 2 < 3, takas; ebeveyn 0: 6 ≥ 3, `return`	[6, 5, 3, 4, 1, 2]	8

Tablo 11.10: Boş heap'e (kapasite 4) art arda ekleme.

İlk ekleme hiçbir karşılaştırma yapmaz: tek öğe zaten köktür. `ekle(6)`, o ana kadarki en büyük öğe olduğu için yapraktan köke çıkar. `ekle(1)` anında dizi doludur (`n = 4 = heap.length`). Önce kapasite 8'e büyütülür, sonra 1 yazılır. 1, ebeveyni 5'ten küçük olduğu için hiç hareket etmez. Son eklemede 3, ebeveyni 2 ile yer değiştirir ve 6'nın altında durur.



Şekil 11.16: (a) `ekle(6)`: 6 indis 3'e yazıldı. (b) İki takasla 6 kökte: [6, 5, 2, 4]. (c) Büyütmeden sonra 1, indis 4'e yazıldı ve yerinde kaldı. (d) `ekle(3)`: 3, ebeveyni 2 ile yer değiştirdi: [6, 5, 3, 4, 1, 2].

Son dizi `[6, 5, 3, 4, 1, 2]` sıralı değildir, ama geçerli bir max-heap'tir: 6 ≥ 5, 3; 5 ≥ 4, 1; 3 ≥ 2. Aynı öğeler farklı sırayla eklenseydi farklı bir heap ortaya çıkabilirdi. Bir öğe kümesinin tek bir heap'i yoktur; heap'in biçimi kesin, öğelerin yerleşimi ise ekleme sırasına bağlıdır.

11.4.5 Kapasite büyütme

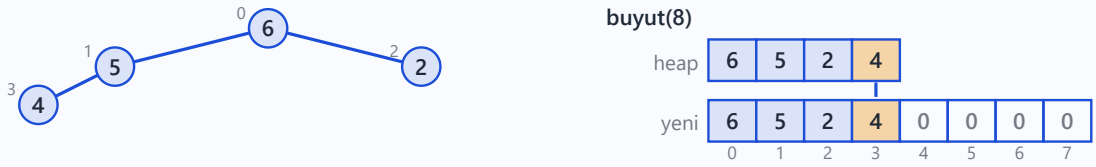
Java dizilerinin boyu oluşturulduktan sonra değişmez. Kapasiteyi büyütmek, daha büyük yeni bir dizi oluşturup eski öğeleri ona kopyalamak demektir:

```
private void buyut(int kapasite) {
  int[] yeni = new int[kapasite];
  for (int k = 0; k < n; k++)
    yeni[k] = heap[k];          // kopyala
  heap = yeni;                  // eski dizi çöpe
}
```

`buyut(kapasite)` önce istenen boyda yeni bir dizi ayırır. Java yeni bir `int` dizisinin bütün hücrelerini 0 ile doldurur. Döngü, eski dizinin ilk `n` öğesini aynı indislerle yeni diziye kopyalar. Son satır `heap` alanını yeni diziye yönlendirir. Eski diziye artık hiçbir referans kalmaz ve çöp toplayıcı onu daha sonra serbest bırakır.

Adım	`k`	`yeni`
`new int[8]`	—	[0, 0, 0, 0, 0, 0, 0, 0]
`yeni[0] = heap[0]`	0	[6, 0, 0, 0, 0, 0, 0, 0]
`yeni[1] = heap[1]`	1	[6, 5, 0, 0, 0, 0, 0, 0]
`yeni[2] = heap[2]`	2	[6, 5, 2, 0, 0, 0, 0, 0]
`yeni[3] = heap[3]`	3	[6, 5, 2, 4, 0, 0, 0, 0]
`k = 4 = n` : döngü biter	4	4 öğe kopyalandı
`heap = yeni`	—	ağaç aynı, kapasite 8

Tablo 11.11: `[6, 5, 2, 4]` dizisinde `buyut(8)` izlemesi.



Şekil 11.17: Kopyalama bitmek üzere: eski dizinin dört öğesi yeni dizinin ilk dört hücresine geçmiş; kalan hücreler 0.

Tek bir büyütmenin maliyeti *O(n)*'dir, çünkü *n* öğe kopyalanır. Ancak büyütme seyrek yapılır. Kapasite her seferinde iki katına çıktığı için büyütmeler 4, 8, 16, … öğe dolduğunda olur. Boş heap'e *n* ekleme yapıldığında kopyalanan toplam öğe sayısı *4 + 8 + 16 + …* biçiminde bir toplamdır ve son terimi *n*'den küçüktür. Her terimi öncekinin iki katı olan böyle bir toplam, son teriminin iki katından küçüktür. Toplam kopya bu yüzden *2n*'den azdır ve ekleme başına düşen kopya maliyeti sabittir: büyütme **amortize** *O(1)*'dir.

Kapasiteyi iki katına çıkarmak yerine sabit bir miktar (örneğin 4 hücre) artırmak kötü bir seçim olurdu. O zaman her 4 eklemede bir büyütme yapılır ve toplam kopya $4 + 8 + 12 + \ldots \approx n^2/8$ olur: ekleme başına *O(n)*.

Silme işleminde de simetrik bir küçültme yapılabilir: öğe sayısı kapasitenin dörtte birine inince dizi yarıya indirilir. Eşik dörtte bir seçilir, yarı seçilmez. Yarı seçilseydi tam eşikte art arda yapılan bir ekleme ve bir silme her seferinde büyütme ve küçültmeye yol açardı. Küçültme isteğe bağlıdır ve bu notun kodunda yer almaz.

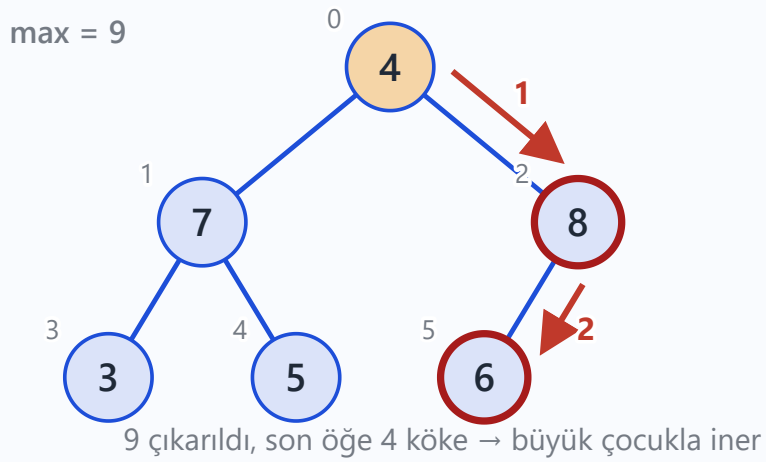
Sık yapılan hata

Büyütmeyi `heap = new int[2 * heap.length]` ile yapıp kopyalamayı unutmak. Yeni dizi sıfırlarla dolu olarak gelir ve heap'in bütün içeriği kaybolur. Büyütme her zaman "yeni dizi ayır, eski öğeleri kopyala, referansı değiştir" sırasıyla yapılır; Java'da bunu tek satırda `Arrays.copyOf(heap, yeniBoy)` yapar.

11.4.6 En büyüğü çıkarma: down-heap

Max-heap'te en büyük öğe köktedir. Onu çıkarmak, ekleme gibi iki aşamada yapılır.

1. **Kökü sakla.** Kökteki öğe döndürülecek değerdir; bir değişkene alınır.
2. **Biçimi koru.** Tam ağaçtan biçimi bozmadan çıkarılabilecek tek düğüm sonuncusudur. Ağacın son öğesi `heap[n - 1]` köke taşınır ve `n` bir azalır.
3. **Bozukluğu onar.** Yeni kök büyük olasılıkla küçük bir öğedir ve çocuklarından küçük olabilir. Kök, çocuklarından *büyük olanla* yer değiştirir.
4. **Tekrarla.** Öğe yaprağa ulaşana ya da iki çocuğundan da büyük veya onlara eşit olana kadar aşağı iner.



Şekil 11.18: [9, 7, 8, 3, 5, 6, 4]'ten 9 çıkarıldıktan sonra son öğe 4 köke taşınmış: 4 önce büyük çocuk 8 ile, sonra 6 ile yer değiştirir.

Şekil 11.18 süreci gösteriyor. 9 çıkarılır ve son öğe 4 köke gelir. Kökün çocukları 7 ve 8'dir; büyük olan 8 ile takas yapılır. 4 şimdi indis 2'dedir ve tek çocuğu 6'dır; 4 < 6 olduğu için bir takas daha yapılır. 4 bir yaprağa ulaşmıştır. Sonuç `[8, 7, 6, 3, 5, 4]`'tür.

Neden büyük çocuk? Kök hangi çocukla yer değiştirirse o çocuk yeni kök olur ve diğer çocuğun ebeveyni olur. Yukarı çıkan çocuk büyük olan ise diğerinden büyük ya da ona eşittir ve heap-sıra özelliği yeni kökte sağlanır. Küçük çocuk yukarı çıksaydı, kendisinden büyük bir çocuğun ebeveyni olurdu; bozukluk düzelmez, yalnız yer değiştirirdi. Min-heap'te aynı akıl yürütme küçük çocuğu seçtirir.

Maliyet: her düzeyde iki karşılaştırma yapılır. Biri iki çocuk arasında büyüğü seçmek, diğeri öğeyi o çocukla karşılaştırmak içindir. Öğe en çok $h = \lfloor \log_2 n \rfloor$ düzey iner. Toplam iş $O(\log n)$'dir.

11.4.7 Kod: silMax() ve batir()

```
public int silMax() {           // n > 0 olmalı
  int max = heap[0];
  yerDegistir(0, n - 1);  n--;  // son öğe köke
  batir(0);
  return max;
}
private void batir(int i) {       // down-heap
  while (2 * i + 1 < n) {       // sol çocuk var mı?
    int j = 2 * i + 1;            // j: büyük çocuk
    if (j + 1 < n && heap[j + 1] > heap[j]) j++;
    if (heap[i] >= heap[j]) return;
    yerDegistir(i, j);  i = j;    // bir düzey in
  }
}
```

`silMax` kökü `max`'ta saklar. Sonra kökü son öğeyle takas eder ve `n`'yi azaltır. Takas, son öğeyi köke getirirken eski kökü de dizinin sonuna, artık heap'e ait olmayan bir hücreye taşır. Heap sort bu ayrıntıdan yararlanır: art arda yapılan çıkarmalar en büyük öğeleri dizinin sonunda biriktirir. Ardından `batir(0)` çağrılır ve `max` döndürülür. Boş heap'te `heap[0]` anlamsız olduğu için gerçek bir sınıfta metodun başına `if (n == 0)` denetimi ve bir istisna eklenir.

`batir(i)` down-heap'tir; adı bir cismin suya batması benzetmesinden gelir. Döngü koşulu `2 * i + 1 < n`, `i`'nin sol çocuğu var mı sorusudur. Tam ağaçta sol çocuğu olmayan düğümün sağ çocuğu da olamaz; bu yüzden sol çocuk yoksa `i` bir yapraktır ve döngü biter. Döngü içinde `j` önce sol çocuğa ayarlanır. Sağ çocuk varsa ve daha büyükse `j` bir artırılır. Böylece `j` büyük çocuğu gösterir. Öğe bu çocuktan büyük ya da ona eşitse sıra sağlanmıştır ve metot döner; değilse takas yapılır ve öğe bir düzey iner.

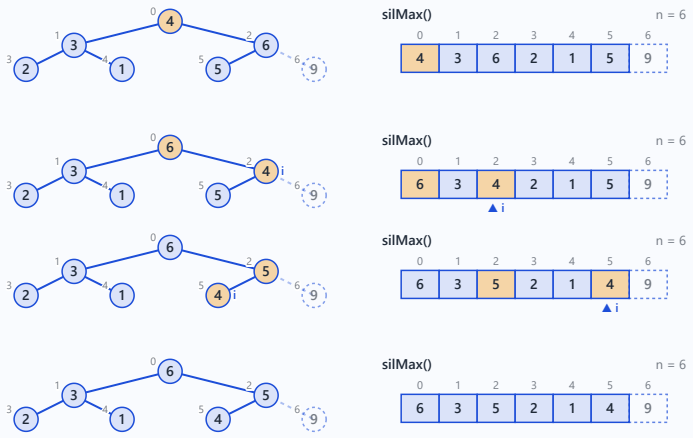
Sağ çocuk denetimindeki `j + 1 < n` koşulu kritik önemdedir. `silMax`'tan sonra eski kök dizinin `n` indisinde durmaya devam eder. Koşul unutulursa `heap[j + 1]` bu silinmiş öğeyi okuyabilir ve onu "büyük çocuk" sanıp heap'e geri taşıyabilir. Bu hatanın etkisi Örnek 11.5'te görülür.

11.4.8 Kod izleme: silMax()

Heap `[9, 3, 6, 2, 1, 5, 4]` (`n = 7`) üzerinde `silMax()` çağrılıyor.

Adım	`i`	Çocuklar → `j`	Karar	Dizi
`max = heap[0]`	—	—	`max = 9`	[9, 3, 6, 2, 1, 5, 4]
`yerDegistir(0, 6); n--`	—	—	`n = 6`	[4, 3, 6, 2, 1, 5 | 9]
tur 1	0	1 (3), 2 (6) → `j = 2`	4 < 6: takas	[6, 3, 4, 2, 1, 5 | 9]
tur 2	2	yalnız 5 (5) → `j = 5`	4 < 5: takas	[6, 3, 5, 2, 1, 4 | 9]
döngü sonu	5	`2 · 5 + 1 = 11 ≥ 6`	yaprak	[6, 3, 5, 2, 1, 4 | 9]
`return max`	—	—	9 döner	—

Tablo 11.12: `[9, 3, 6, 2, 1, 5, 4]` üzerinde `silMax()` izlemesi. Dikey çizginin sağındaki öğe artık heap'e ait değildir.



Şekil 11.19: (a) Kök ile son öğe yer değiştirdi; 9 heap dışında (soluk). (b) 4, büyük çocuk 6 ile takas edildi. (c) 4, tek çocuğu 5 ile takas edildi. (d) Sonuç [6, 3, 5, 2, 1, 4].

İlk turda kökün iki çocuğu vardır ve büyüğü 6'dır. İkinci turda indis 2'deki 4'ün yalnız sol çocuğu vardır: `j = 5`, sağ çocuk adayı `j + 1 = 6` ise `n = 6`'dan küçük değildir. Bu noktada `heap[6]` hâlâ 9'dur. `j + 1 < n` denetimi sayesinde bu silinmiş değer hiç okunmaz. Toplam 3 karşılaştırma ve 2 takas yapılmıştır; yükseklik $\lfloor \log_2 6 \rfloor = 2$ olduğundan bu, olabilecek en çok takas sayısıdır.

11.4.9 Kod izleme: art arda iki silMax()

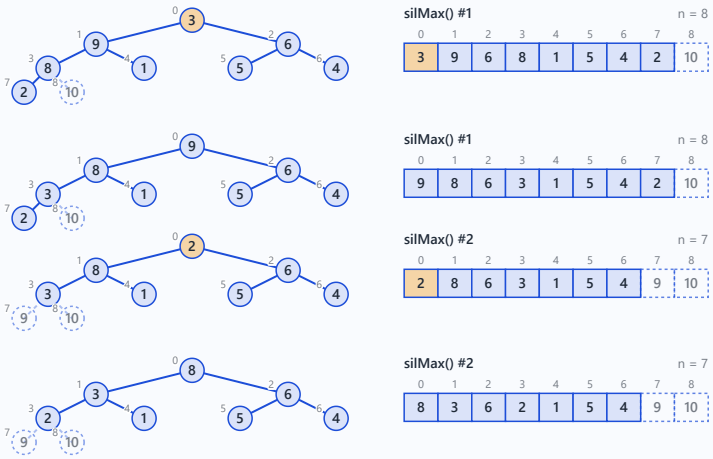
Up-heap izlemesinin sonunda elde edilen `[10, 9, 6, 8, 1, 5, 4, 2, 3]` heap'inden art arda iki kez en büyük öğeyi çıkaralım.

Adım	`i`	Çocuklar → `j`	Karar	Heap kısmı
#1: kök ↔ son, `n = 8`	—	—	`max = 10`	[3, 9, 6, 8, 1, 5, 4, 2]
tur 1	0	9, 6 → `j = 1`	3 < 9: takas	[9, 3, 6, 8, 1, 5, 4, 2]
tur 2	1	8, 1 → `j = 3`	3 < 8: takas	[9, 8, 6, 3, 1, 5, 4, 2]

Adım	`i`	Çocuklar → `j`	Karar	Heap kısmı
tur 3	3	yalnız 2 → `j = 7`	3 ≥ 2: `return`	[9, 8, 6, 3, 1, 5, 4, 2]
#2: kök ↔ son, `n = 7`	—	—	`max = 9`	[2, 8, 6, 3, 1, 5, 4]
tur 1	0	8, 6 → `j = 1`	2 < 8: takas	[8, 2, 6, 3, 1, 5, 4]
tur 2	1	3, 1 → `j = 3`	2 < 3: takas	[8, 3, 6, 2, 1, 5, 4]
döngü sonu	3	`2 · 3 + 1 = 7 ≥ 7`	yaprak	[8, 3, 6, 2, 1, 5, 4]

Tablo 11.13: `[10, 9, 6, 8, 1, 5, 4, 2, 3]` üzerinde art arda iki `silMax()`.

İki çıkarma 10 ve 9'u döndürür. İlk çıkarmada inen öğe 3, yaprağa varmadan durur: indis 3'te tek çocuğu 2'dir ve 3 ≥ 2'dir. İkinci çıkarmada inen öğe 2 bir yaprağa kadar iner. Dizinin tamamına bakıldığında son iki hücrede 9 ve 10 durur: `[8, 3, 6, 2, 1, 5, 4 | 9, 10]`. Çıkarılan öğeler dizinin sonunda büyükten küçüğe doğru, sağdan sola birikmiştir.



Şekil 11.20: (a) İlk çıkarma: 3 köke geldi, 10 heap dışında. (b) İlk çıkarma bitti: [9, 8, 6, 3, 1, 5, 4, 2]. (c) İkinci çıkarma: 2 köke geldi. (d) İkinci çıkarma bitti; 9 ve 10 dizinin sonunda.

Örnek 11.5

`batir`'da `j + 1 < n` denetimi unutulsun: `if (heap[j + 1] > heap[j]) j++;`. Max-heap `[9, 8, 3, 5, 4]` (`n = 5`) üzerinde `silMax()` ne yapar?

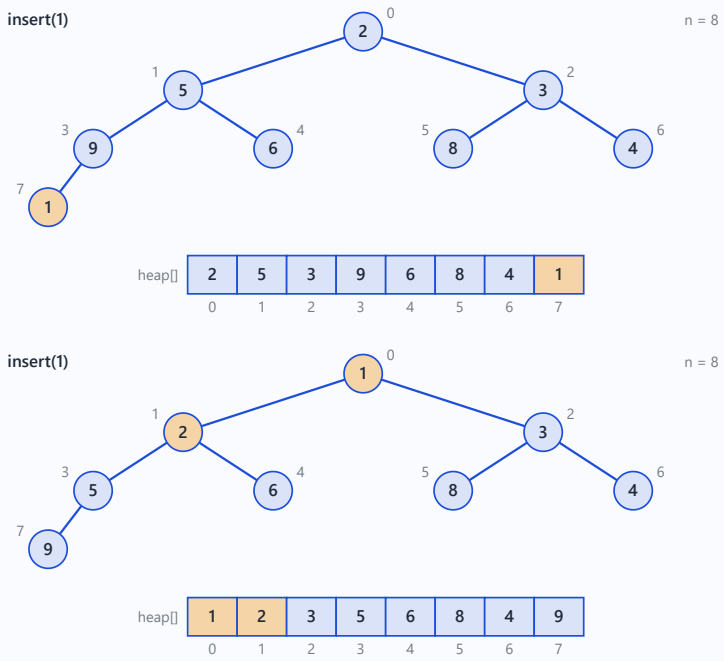
1. `max = 9`; 9 ile 4 yer değiştirir ve `n = 4` olur: `[4, 8, 3, 5 | 9]`.
2. `i = 0`: çocuklar 8 ve 3, `j = 1`. 4 < 8, takas: `[8, 4, 3, 5 | 9]`.
3. `i = 1`: sol çocuk indis 3'te (5). Hatalı kod `heap[4] = 9 > 5` diye `j = 4` seçer. Oysa indis 4 artık heap'e ait değildir; orada silinmiş 9 durur. 4 < 9, takas: `[8, 9, 3, 5 | 4]`.
4. Sonuç: heap'in ilk dört öğesi `[8, 9, 3, 5]`'tir. Silinen 9 geri gelmiş, 4 kaybolmuş ve 9 > ebeveyni 8 olduğu için heap bozulmuştur. Doğru kod `[8, 5, 3, 4]` verirdi. `batir`, `n` azaltılmadan çağrıldığında (örneğin `heapOlustur`'da) dizi tam doluysa aynı hata `ArrayIndexOutOfBoundsException` fırlatır.

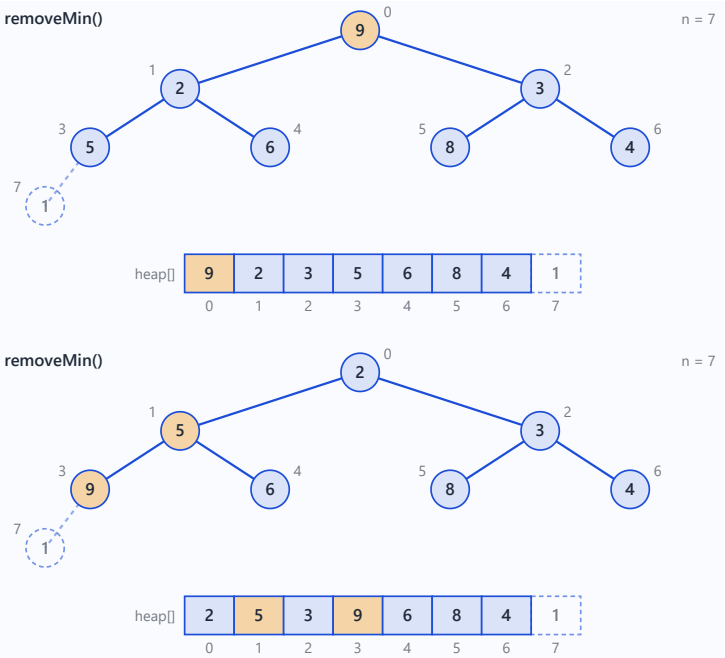
11.4.10 Min-heap'te insert ve removeMin

Min-heap'te iki işlem aynı biçimde çalışır; yalnız karşılaştırmaların yönü terstir. Yukarı süzmede öğe ebeveyninden *küçükse* yukarı çıkar. Aşağı süzmede öğe çocuklarının *küçüğüyle* yer değiştirir. Başlangıç min-heap'i `[2, 5, 3, 9, 6, 8, 4]` olsun: en küçük öğe 2 köktedir.

Adım	Karar	Dizi
`insert(1)`: indis 7'ye yaz	ebeveyn *(7 − 1)/2 = 3* → 9	[2, 5, 3, 9, 6, 8, 4, 1]
up-heap	1 < 9: takas (7 ↔ 3)	[2, 5, 3, 1, 6, 8, 4, 9]
up-heap	1 < 5: takas (3 ↔ 1)	[2, 1, 3, 5, 6, 8, 4, 9]
up-heap	1 < 2: takas (1 ↔ 0); kök, dur	[1, 2, 3, 5, 6, 8, 4, 9]
`removeMin()`	`min = 1`; kök ↔ son (0 ↔ 7), `n = 7`	[9, 2, 3, 5, 6, 8, 4 | 1]
down-heap	çocuklar 2, 3 → küçük 2; 9 > 2: takas (0 ↔ 1)	[2, 9, 3, 5, 6, 8, 4 | 1]
down-heap	çocuklar 5, 6 → küçük 5; 9 > 5: takas (1 ↔ 3)	[2, 5, 3, 9, 6, 8, 4 | 1]
son	indis 3 yaprak (*2 · 3 + 1 = 7 ≥ 7*)	1 döner

Tablo 11.14: Min-heap `[2, 5, 3, 9, 6, 8, 4]` üzerinde `insert(1)` ve `removeMin()`.





Şekil 11.21: Min-heap'te (a) 1 indis 7'ye eklendi, (b) üç takasla köke çıktı, (c) `removeMin`'de 9 köke taşındı, (d) 9 iki takasla yerine indi.

Bu örnekte ekleme ve çıkarmanın birbirini tam olarak geri aldığı görülüyor: son dizi başlangıç dizisiyle aynıdır. Bu bir rastlantıdır. Eklenen öğe en küçük olduğu için yol boyunca her öğeyi bir düzey aşağı itmiştir; çıkarma sırasında köke gelen 9 da aynı yoldan inmiştir. Genel olarak bir ekleme ve onu izleyen bir çıkarma heap'i başlangıç durumuna döndürmez.

Max-heap kodundan min-heap kodu elde etmek için üç karşılaştırmayı çevirmek yeter: `yuzdur`'da `heap[ebeveyn] >= heap[i]` yerine `heap[ebeveyn] <= heap[i]`, `batir`'da büyük çocuk yerine küçük çocuk seçimi ve `heap[i] <= heap[j]` durma koşulu. Bir başka yol, anahtarları eksiyle çarpıp max-heap'e koymaktır: en büyük $-x$, en küçük x'e karşılık gelir.

11.4.11 Heap işlemlerinin maliyeti

İşlem	Maliyet	Neden?
`size`, `isEmpty`	O(1)	`n` sayacı tutulur.
`max` / `min` (peek)	O(1)	Öğe her zaman `heap[0]`'dadır.
`ekle` (insert)	O(log n)*	Up-heap: yapraktan köke tek yol.
`silMax` (removeMin)	O(log n)*	Down-heap: kökten yaprağa tek yol.
Rastgele bir öğeyi arama	O(n)	Heap sıralı değildir; arama için uygun değildir.

Tablo 11.15: İkili heap işlemlerinin maliyeti.

> * Dizi büyütme ve küçültme dahil amortize maliyet.

Logaritmik maliyetin pratik karşılığını yükseklik formülü verir. $n = 10^6$ öğeli bir heap'in yüksekliği $\lfloor \log_2 10^6 \rfloor = 19$'dur; ekleme ya da çıkarma en çok 19 takas yapar. Sıralı bir dizide aynı ekleme yarım milyon öğeyi kaydırabilirdi.

Heap'in bellek maliyeti de düşüktür: n öğe için $O(n)$ hücre. İki katına büyütme nedeniyle dizinin en çok yarısı boş kalabilir. Bağlı ağaç gerçeklemesinde ise her düğüm iki çocuk referansı ve çoğu zaman bir ebeveyn referansı taşırdı.

Tablonun son satırı, heap'in neyi yapmadığını hatırlatır. Heap yalnız en öncelikli öğeyi hızlı bulur. "7 kuyrukta mı?" sorusuna cevap vermek için bütün diziyi taramak gerekir. Belirli bir öğenin önceliğini değiştirmek ya da onu silmek de önce bu öğenin bulunmasını gerektirir. Bu sorunun çözümü uyarlanabilir öncelikli kuyruktur.

11.4.12 Uyarlanabilir öncelikli kuyruk

Bazı uygulamalarda kuyruktaki bir öğenin önceliği bekleme sırasında değişir. Dijkstra algoritmasında bir düğüme daha kısa bir yol bulununca o düğümün uzaklığı, yani anahtarı küçülür. Acil serviste bekleyen bir hastanın durumu ağırlaşır. Bazen de bir öğe çıkmadan kuyruktan ayrılır: hasta beklemekten vazgeçer, bir süreç iptal edilir.

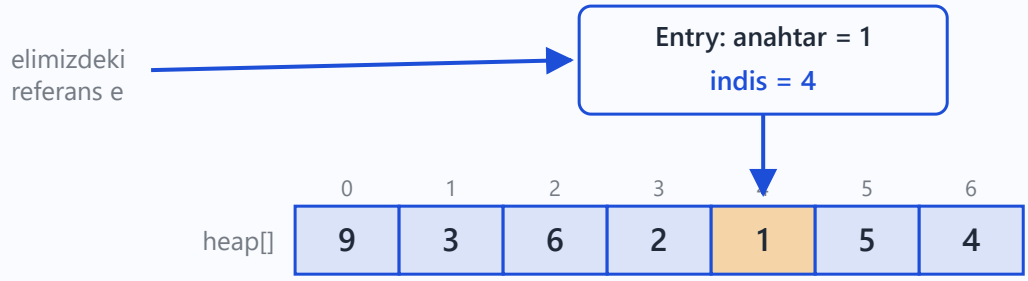
Tanım 11.6

Uyarlanabilir öncelikli kuyruk (*adaptable priority queue*), temel işlemlere ek olarak şu iki işlemi sunan öncelikli kuyruktur:

- `replaceKey(e, k)`: `e` öğesinin anahtarını `k` yapar;
- `remove(e)`: `e` öğesini kuyruktan çıkarır ve döndürür.

Burada `e`, kuyruğa eklenirken kullanıcıya verilen öğe nesnesidir.

Bu işlemlerin asıl zorluğu, `e` öğesinin heap dizisinde nerede olduğunu bilmektir. Heap sıralı olmadığı için öğeyi aramak $O(n)$'dir. Goodrich ve arkadaşlarının çözümü **konum farkında öğedir** (*location-aware entry*). Her öğe nesnesi, anahtarı ve değerinin yanında heap dizisindeki kendi indisini de saklar. Heap'te her takas yapıldığında takas edilen iki öğenin indis alanı da güncellenir.



Şekil 11.22: Konum farkında öğe. Dışarıdaki `e` referansı, anahtarı 1 olan ve heap dizisindeki indisini (4) bilen öğe nesnesini gösterir.

Konum bilindiğinde iki işlem de heap'in bildik araçlarıyla yapılır:

- `replaceKey(e, k)`. `heap[e.indis]`'in anahtarı değiştirilir. Max-heap'te anahtar büyüdüyse öğe yalnız ebeveyniyle çatışabilir ve up-heap uygulanır. Küçüldüyse yalnız çocuklarıyla çatışabilir ve down-heap uygulanır.
- `remove(e)`. Öğe dizinin son öğesiyle takas edilir ve `n` bir azaltılır. Yerine gelen öğe, ebeveyninden büyük de olabilir, çocuklarından küçük de. Bu yüzden ona duruma göre up-heap ya da down-heap uygulanır.

Konum alanı olmadan bu işlemlerin her biri önce *O(n)*'lik bir arama gerektirirdi. Konumla öğeye *O(1)*'de ulaşılır ve işlemler *O(log n)*'e iner. Bedeli, her takasta iki indis alanının da güncellenmesidir; bu da sabit bir ek iştir.

Sık yapılan hata

`remove(e)`'de yerine gelen öğeye yalnız down-heap uygulamak. Son öğe, silinen öğenin bulunduğu alt ağaçtan değil, ağacın başka bir dalından gelebilir; silinen öğenin ebeveyninden büyük olabilir. Örneğin `[9, 3, 8, 1, 2, 7, 6]`'dan indis 3'teki 1 silinirse yerine 6 gelir ve 6 > ebeveyni 3'tür: up-heap gerekir.

11.4.13 Kod izleme: anahtar büyütme ve küçültme

Bu notun basit max-heap sınıfı öğe nesneleri yerine tam sayılar tutar. Bu yüzden öğe indisle verilir: `anahtarDegistir(i, yeni)`, `heap[i]`'yi `yeni` yapar ve sırayı onarır.

```
public void anahtarDegistir(int i, int yeni) {
  int eski = heap[i];
  heap[i] = yeni;
  if (yeni > eski) yuzdur(i);  // büyüdü: yukarı
  else batir(i);               // küçüldü: aşağı
}
private void yuzdur(int i) {     // up-heap
  while (i > 0) {
    int ebeveyn = (i - 1) / 2;
    if (heap[ebeveyn] >= heap[i]) return;
    yerDegistir(i, ebeveyn);  i = ebeveyn;
  }
}
```

Metot eski değeri saklar, yeni değeri yazar ve ikisini karşılaştırarak yönü seçer. Anahtar büyüdüyse `yuzdur(i)`, küçüldüyse `batir(i)` çağrılır. Eşitse `batir` çağrılır ama hiçbir şey yapmadan döner, çünkü öğe çocuklarından zaten büyük ya da eşittir. Ağacın biçimi hiç değişmez; yalnız tek bir yol boyunca değerler yer değiştirir.

Anahtar büyütme. Heap `[9, 3, 6, 2, 1, 5, 4]` üzerinde `anahtarDegistir(4, 8)` çağrılsın.

Adım	`i`	Karar	Dizi
`eski = heap[4]`	4	`eski = 1`	[9, 3, 6, 2, 1, 5, 4]
`heap[4] = 8`	4	8 > 1: büyüdü, `yuzdur(4)`	[9, 3, 6, 2, 8, 5, 4]
tur 1	4	ebeveyn 1: 3 < 8, takas	[9, 8, 6, 2, 3, 5, 4]
tur 2	1	ebeveyn 0: 9 ≥ 8, `return`	[9, 8, 6, 2, 3, 5, 4]

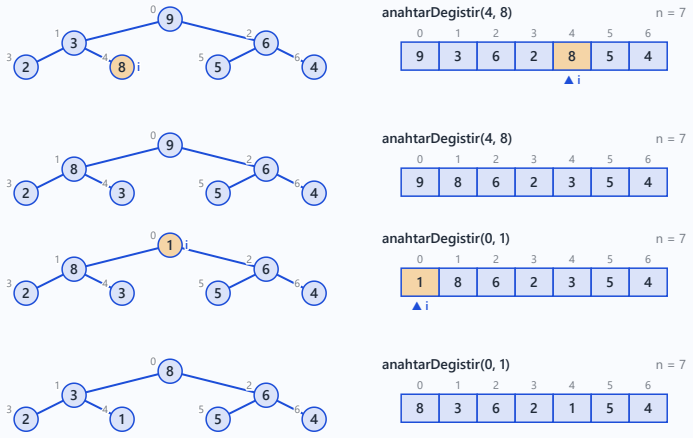
Tablo 11.16: `[9, 3, 6, 2, 1, 5, 4]` üzerinde `anahtarDegistir(4, 8)`.

8 bir düzey yukarı çıkmış ve kökün altında durmuştur. Yalnız $4 \to 1 \to 0$ yolu incelenmiştir. İspattaki gözlem burada da görülüyor: aşağı inen 3, 8'in eski yerine, yani bir yaprağa iner ve altında çocuk kalmaz. 3, indis 1'deyken öbür çocuğu 2'den zaten büyüktü.

Anahtar küçültme. Aynı heap'in yeni hâli `[9, 8, 6, 2, 3, 5, 4]` üzerinde bu kez kökün anahtarı küçültülsün: `anahtarDegistir(0, 1)`.

Adım	`i`	Çocuklar → `j`	Karar	Dizi
`eski = heap[0]` ; `heap[0] = 1`	0	—	1 < 9: küçüldü, `batir(0)`	[1, 8, 6, 2, 3, 5, 4]
tur 1	0	8, 6 → `j = 1`	1 < 8: takas	[8, 1, 6, 2, 3, 5, 4]
tur 2	1	2, 3 → `j = 4`	1 < 3: takas	[8, 3, 6, 2, 1, 5, 4]
döngü sonu	4	`2 · 4 + 1 = 9 ≥ 7`	yaprak	[8, 3, 6, 2, 1, 5, 4]

Tablo 11.17: `[9, 8, 6, 2, 3, 5, 4]` üzerinde `anahtarDegistir(0, 1)`.



Şekil 11.23: Üstte anahtar büyütme: (a) `heap[4] = 8`, (b) 8 bir düzey çıktı. Altta anahtar küçültme: (c) `heap[0] = 1`, (d) 1 iki düzey indi, 8 köke çıktı.

1 iki düzey inmiş, 8 köke çıkmıştır. Kökün anahtarını küçültüp `batir(0)` çağırmak, `silMax`'ın aşağı süzme adımıyla aynı işi yapar. `silMax`'ta köke gelen değer son öğedir, burada ise kullanıcının verdiği yeni değerdir.

İki yönde de maliyet, değişen öğenin izlediği yolun uzunluğuyla sınırlıdır: *O(log n)*. İndis bilinmeseydi önce öğeyi aramak gerekirdi ve maliyet *O(n)*'e çıkardı.

11.5 Heap oluşturma ve heap sort

Şimdiye kadar heap'e öğeler tek tek eklendi. Bazen ise bütün öğeler baştan elimizdedir: bir dosyadan okunan sayılar, bir dizide bekleyen görevler. Bu durumda heap'i öğeleri tek tek ekleyerek kurmak gerekmez; diziyi yerinde düzenleyen daha hızlı bir yöntem vardır. Aynı yöntem, heap'i bir sıralama algoritmasına dönüştürmenin de ilk adımıdır.

11.5.1 n öğeden heap oluşturmanın iki yolu

Birinci yol: n kez ekleme. Boş bir heap'e öğeler sırayla eklenir. Her ekleme up-heap yapar ve *O(log n)* sürer; toplam *O(n log n)*'dir. Öğeler tek tek ve zaman içinde geliyorsa, yani algoritma bütün girdiyi önceden göremiyorsa (*online*, çevrim içi durum) tek seçenek budur.

İkinci yol: aşağıdan yukarıya oluşturma (*bottom-up heap construction*). Bütün öğeler bir dizide hazırsa dizi olduğu gibi tam ikili ağaç sayılır; biçim zaten doğrudur. Yalnız heap-sıra özelliği eksiktir. Bunu sağlamak için son iç düğümden köke doğru her düğüme down-heap uygulanır:

```
for (int k = n / 2 - 1; k >= 0; k--)   // son iç düğümden köke
    batir(k);                           // alt ağaçları heap olan k'yi batır
```

Döngünün başladığı *⌊n/2⌋ − 1* indisi, sol çocuğu olan son düğümdür. Bundan sonraki indislerin hepsi yapraktır, çünkü *i ≥ ⌊n/2⌋* için *2i + 1 ≥ n* olur. Yapraklar tek başlarına zaten heap'tir; onlara dokunmaya gerek yoktur. Dizinin yarısı bu yüzden hiç işlenmez.

Algoritmanın doğruluğu döngünün yönüne dayanır. `batir(k)` çağrıldığında `k`'nın iki alt ağacı zaten heap'tir: o alt ağaçların kökleri `k`'dan büyük indislidir ve onlar daha önce işlenmiştir ya da yapraktır. Down-heap tam bu durumu onarmak için tasarlanmıştır: alt ağaçları heap olan bir kökü doğru yerine indirir. Döngü köke ulaştığında bütün ağaç bir heap olur.

> **Sık yapılan hata**
>
> Aşağıdan yukarıya oluşturmada döngüyü kökten başlatmak: `for (int k = 0; k < n / 2; k++)`. Bu sırada `batir(k)` çağrıldığında alt ağaçlar henüz heap değildir ve down-heap'in varsayımı bozulur. Örneğin `[1, 2, 3, 4]` dizisinde bu döngü `[3, 4, 1, 2]` üretir; 4, ebeveyni 3'ten büyüktür. Döngü son iç düğümden köke doğru, yani azalan `k` ile yürümelidir.

11.5.2 Kod izleme: heapOlustur()

Max-heap sınıfına aşağıdan yukarıya oluşturma metodu eklenir. Metot, dizide hazır duran `n` öğeyi heap'e çevirir:

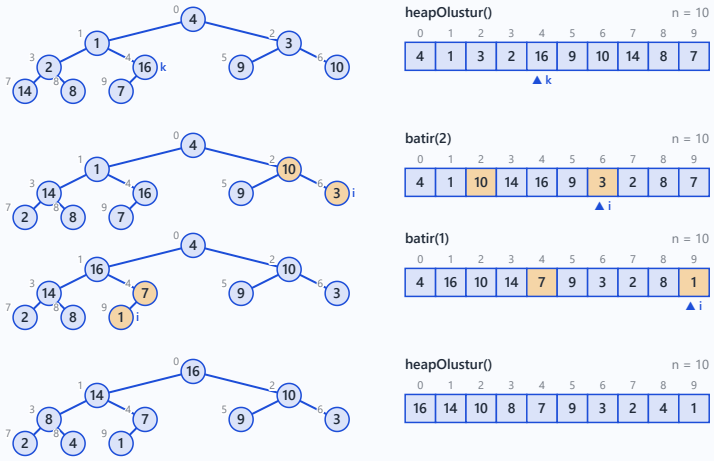
```
public void heapOlustur() {     // rastgele dizi
  for (int k = n / 2 - 1; k >= 0; k--)
    batir(k);                   // alt ağaçlar heap
}
private void batir(int i) {       // down-heap
  while (2 * i + 1 < n) {       // sol çocuk var mı?
    int j = 2 * i + 1;            // j: büyük çocuk
    if (j + 1 < n && heap[j + 1] > heap[j]) j++;
    if (heap[i] >= heap[j]) return;
    yerDegistir(i, j);  i = j;    // bir düzey in
  }
}
```

Girdi, Cormen ve arkadaşlarının kitabındaki klasik örnektir: `[4, 1, 3, 2, 16, 9, 10, 14, 8, 7]`, `n = 10`. Dizi bir heap değildir; örneğin kök 4, çocuğu 16'dan küçüktür. İndis 5'ten 9'a kadar olan düğümler yapraktır. Döngü *k = 10/2 − 1 = 4*'ten başlar.

`k`	`i`	Çocuklar → `j`	Karar	Dizi
4	4	yalnız 9 (7) → `j = 9`	16 ≥ 7: `return`	[4, 1, 3, 2, 16, 9, 10, 14, 8, 7]
3	3	7 (14), 8 (8) → `j = 7`	2 < 14: takas	[4, 1, 3, 14, 16, 9, 10, 2, 8, 7]
3	7	`15 ≥ 10`	yaprak	—
2	2	5 (9), 6 (10) → `j = 6`	3 < 10: takas	[4, 1, 10, 14, 16, 9, 3, 2, 8, 7]

`k`	`i`	Çocuklar → `j`	Karar	Dizi
2	6	`13 ≥ 10`	yaprak	—
1	1	3 (14), 4 (16) → `j = 4`	1 < 16: takas	[4, 16, 10, 14, 1, 9, 3, 2, 8, 7]
1	4	yalnız 9 (7) → `j = 9`	1 < 7: takas	[4, 16, 10, 14, 7, 9, 3, 2, 8, 1]
1	9	`19 ≥ 10`	yaprak	—
0	0	1 (16), 2 (10) → `j = 1`	4 < 16: takas	[16, 4, 10, 14, 7, 9, 3, 2, 8, 1]
0	1	3 (14), 4 (7) → `j = 3`	4 < 14: takas	[16, 14, 10, 4, 7, 9, 3, 2, 8, 1]
0	3	7 (2), 8 (8) → `j = 8`	4 < 8: takas	[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
0	8	`17 ≥ 10`	yaprak	—

Tablo 11.18: `[4, 1, 3, 2, 16, 9, 10, 14, 8, 7]` üzerinde `heapOlustur()`. Her satır bir `batir` turunu gösterir.



Şekil 11.24: (a) Sıradan dizi. (b) `k = 4, 3, 2`'den sonra. (c) `k = 1`'den sonra: 1 iki düzey inmiş. (d) `k = 0`'dan sonra max-heap [16, 14, 10, 8, 7, 9, 3, 2, 4, 1].

İzlemede yedi takas yapılmıştır. Üst düğümler daha uzun yollar yürümüştür: `k = 0`'daki 4 üç düzey inmiştir. Buna karşılık bu düğümlerin sayısı azdır. Alt düzeydeki düğümler ise ya hiç işlenmemiş (yapraklar) ya da en çok bir takas yapmıştır. Toplam işin neden doğrusal kaldığı bu dengeden gelir.

11.5.3 Aşağıdan yukarıya oluşturma neden O(n)?

İlk bakışta aşağıdan yukarıya oluşturmanın maliyeti *O(n log n)* gibi görünür: yaklaşık *n/2* düğüme down-heap uygulanır ve her biri *O(log n)* sürebilir. Bu üst sınır doğrudur ama gevşektir. Bir down-heap'in maliyeti, düğümün *yüksekliği* kadardır, yani düğümden altındaki en derin yaprağa kadar olan uzaklık kadardır. Düğümlerin çoğu ağacın altında ve yükseklikleri küçüktür.

15 düğümlü mükemmel bir ikili ağaçla (bütün düzeyleri dolu, yükseklik 3) hesaplayalım.

Düğümün yüksekliği	Düğüm sayısı	En çok takas	Toplam
0 (yapraklar)	8	0	0

Düğümün yüksekliği	Düğüm sayısı	En çok takas	Toplam
1	4	1	4
2	2	2	4
3 (kök)	1	3	3
n = 15			**11**

Tablo 11.19: 15 düğümlü ağaçta aşağıdan yukarıya oluşturmanın en kötü durumda yaptığı takas sayısı.

Düğümlerin yarısından fazlası yapraktır ve hiç iş yapmaz. Çok iş yapan yüksek düğümlerden ise yalnız birkaç tane vardır. Toplam 11 takas, düğüm sayısı 15'ten bile küçüktür. Java ile 1'den 15'e artan bir dizi (max-heap için en kötü girdi) üzerinde çalıştırıldığında da tam 11 takas sayılır.

Aynı ağacı 15 kez ekleme ile kurmak çok daha pahalıdır. Up-heap'in maliyeti düğümün *derinliğine*, yani kökten uzaklığına bağlıdır ve düğümlerin çoğu derindir. Artan sırada gelen öğelerde her yeni öğe köke kadar çıkar: derinliği 1 olan 2 düğüm, derinliği 2 olan 4 düğüm ve derinliği 3 olan 8 düğüm vardır. Toplam $0 + 2 \cdot 1 + 4 \cdot 2 + 8 \cdot 3 = 34$ takas eder. Java ile sayılan değer de 34'tür.

Teorem 11.3 (Aşağıdan yukarıya oluşturmanın maliyeti)

n öğeli bir diziden aşağıdan yukarıya heap oluşturma en çok *n* takas yapar ve $O(n)$ zamanda çalışır.

İspat. *n* düğümlü bir tam ikili ağaçta yüksekliği *h* olan düğümlerin sayısı en çok $\lceil n / 2^{h+1} \rceil$'dir: yaprakların sayısı yaklaşık yarıdır, bir üst kattakiler yaklaşık dörtte birdir ve her katta sayı yarıya iner. Yüksekliği *h* olan bir düğümün down-heap'i en çok *h* takas yapar. Toplam takas sayısı bu yüzden en çok $\Sigma_{h \geq 0} (n / 2^{h+1}) \cdot h = n \cdot \Sigma_{h \geq 0} h / 2^{h+1}$ kadardır. Sondaki seri $1/4 + 2/8 + 3/16 + 4/32 + \ldots$ biçimindedir ve toplamı tam olarak 1'dir. Böylece takas sayısı *n* ile sınırlanır. Her takasa en çok iki karşılaştırma eşlik ettiği için toplam iş $O(n)$'dir. ■

Serinin toplamının 1 olduğu kolayca görülebilir. $S = 1/4 + 2/8 + 3/16 + \ldots$ olsun. $S/2 = 1/8 + 2/16 + \ldots$'dir. Farkları alınırsa $S - S/2 = 1/4 + 1/8 + 1/16 + \ldots = 1/2$ bulunur; buradan $S = 1$ çıkar.

Teoremin özü şu cümlededir: down-heap'in maliyeti yüksekliğe, up-heap'in maliyeti derinliğe bağlıdır ve düğümlerin çoğu aşağıdadır. Aşağıdaki düğümler için yükseklik küçük, derinlik büyüktür. Aşağıdan yukarıya yöntem işi ucuz yapan yönü seçer.

Sık yapılan hata

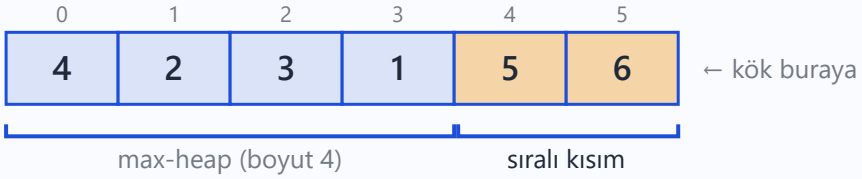
"$n/2$ düğüme $O(\log n)$'lik işlem uygulanıyor, demek ki $O(n \log n)$" diye düşünüp aşağıdan yukarıya oluşturmanın ekleme yoluyla aynı maliyette olduğunu sanmak. Bu hesap yalnız bir üst sınır verir. Düğümlerin çoğu kısa yol yürüdüğü için gerçek toplam doğrusaldır.

11.5.4 Heap sort

Heap, bir sıralama algoritmasının temelini de oluşturur. Fikir şudur: max-heap'ten art arda en büyük öğe çıkarılırsa öğeler büyükten küçüğe sırayla gelir. `silMax` çıkarılan kökü dizinin sonuna, heap'in hemen dışındaki hücreye taşıdığı için bu öğeler ayrı bir diziye bile yazılmaz: dizinin sonunda kendiliğinden birikir.

Heap sort iki aşamadan oluşur:

1. **Heap oluştur.** Dizi aşağıdan yukarıya max-heap'e çevrilir: *O(n)*.
2. ***n − 1* kez en büyüğü sona al.** Kök (o anki en büyük öğe) heap'in son öğesiyle takas edilir. Heap bir küçülür ve kök için down-heap yapılır. Her tur *O(log n)*'dir; toplam *O(n log n)*.



Şekil 11.25: `heapSort([6, 5, 3, 4, 1, 2])` iki tur sonra: [4, 2, 3, 1 | 5, 6]. İlk dört hücre heap, son iki hücre sıralı kısımdır.

Şekil 11.25 algoritmanın ortasından bir anı gösteriyor. Dizi ikiye bölünmüştür. Baştaki kısım bir max-heap'tir ve her turda bir küçülür. Sondaki kısım sıralıdır ve her turda bir büyür. Bu kısımdaki öğeler heap'teki bütün öğelerden büyük ya da onlara eşittir; yani kesin yerlerindedir. İki kısım aynı dizide durduğu için ek dizi gerekmez.

Heap sort'un dört özelliği öne çıkar:

- **Yerinde çalışır.** Heap, sıralanan dizinin kendisinde kurulur; birkaç değişken dışında ek bellek gerekmez: *O(1)*.
- **En kötü durumda bile *O(n log n)*'dir.** Hızlı sıralama (*quick sort*) ortalamada çok hızlıdır ama kötü seçilen pivotlarla $O(n^2)$'ye düşebilir. Heap sort'un böyle bir kötü durumu yoktur.
- **Kararlı değildir.** Kök ile sondaki öğe arasındaki uzak takaslar eşit anahtarlı öğelerin göreli sırasını değiştirebilir.
- **Yön.** Max-heap artan sıralama üretir, çünkü en büyük öğe en sona yerleşir. Min-heap ile aynı algoritma azalan sıralama üretir.

11.5.5 Kod: heapSort()

```
public static void heapSort(int[] a) {
  int n = a.length;
  // 1) oluştur: son iç düğümden köke
  for (int k = n / 2 - 1; k >= 0; k--)
    batir(a, k, n);
  // 2) sırala: en büyüğü sona al
  for (int son = n - 1; son > 0; son--) {
    yerDegistir(a, 0, son);
    batir(a, 0, son);           // heap boyutu = son
  }
}
static void batir(int[] a, int i, int n) {
  while (2 * i + 1 < n) {
    int j = 2 * i + 1;
    if (j + 1 < n && a[j + 1] > a[j]) j++;
    if (a[i] >= a[j]) return;
    yerDegistir(a, i, j);  i = j;
  }
}
```

Kod, sınıftan bağımsız `static` metotlardan oluşur. `batir` burada üç parametre alır: dizi, indis ve heap'in o anki boyu. Gövdesi sınıftaki `batir` ile aynıdır; yalnız `heap` ve `n` alanları yerine parametreler kullanılır. Heap boyunun parametre olması önemlidir: sıralama aşamasında dizinin boyu değişmez, ama heap her turda bir küçülür.

Birinci döngü aşağıdan yukarıya oluşturmadır. İkinci döngüde `son`, heap'in son öğesinin indisidir. Her turda kök ile `a[son]` takas edilir; en büyük öğe kesin yerine oturur. Ardından `batir(a, 0, son)` çağrılır. Üçüncü parametre `son`, heap'in yeni boyudur: `son` indisi ve sağındaki hücreler artık heap'e ait değildir ve `batir` onlara dokunmaz. Döngü `son = 1`'de son kez çalışır; tek öğe kalan heap'e işlem gerekmez.

Kodda kullanılan `yerDegistir(a, i, j)`, dizinin iki hücresini takas eden üç satırlık bir yardımcı metottur:

```
static void yerDegistir(int[] a, int i, int j) {
  int t = a[i];  a[i] = a[j];  a[j] = t;
}
```

Klasik örnek dizi `{4, 1, 3, 2, 16, 9, 10, 14, 8, 7}` heap sort ile `{1, 2, 3, 4, 7, 8, 9, 10, 14, 16}` olur. Tablo 11.20 ikinci aşamanın her turundan sonraki diziyi veriyor.

Aşama	Dizi
oluşturma sonrası	[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]

Aşama	Dizi
`son = 9`	[14, 8, 10, 4, 7, 9, 3, 2, 1 | 16]
`son = 8`	[10, 8, 9, 4, 7, 1, 3, 2 | 14, 16]
`son = 7`	[9, 8, 3, 4, 7, 1, 2 | 10, 14, 16]
`son = 6`	[8, 7, 3, 4, 2, 1 | 9, 10, 14, 16]
`son = 5`	[7, 4, 3, 1, 2 | 8, 9, 10, 14, 16]
`son = 4`	[4, 2, 3, 1 | 7, 8, 9, 10, 14, 16]
`son = 3`	[3, 2, 1 | 4, 7, 8, 9, 10, 14, 16]
`son = 2`	[2, 1 | 3, 4, 7, 8, 9, 10, 14, 16]
`son = 1`	[1 | 2, 3, 4, 7, 8, 9, 10, 14, 16]

Tablo 11.20: `{4, 1, 3, 2, 16, 9, 10, 14, 8, 7}` dizisinde heap sort. Dikey çizginin sağı sıralı kısımdır.

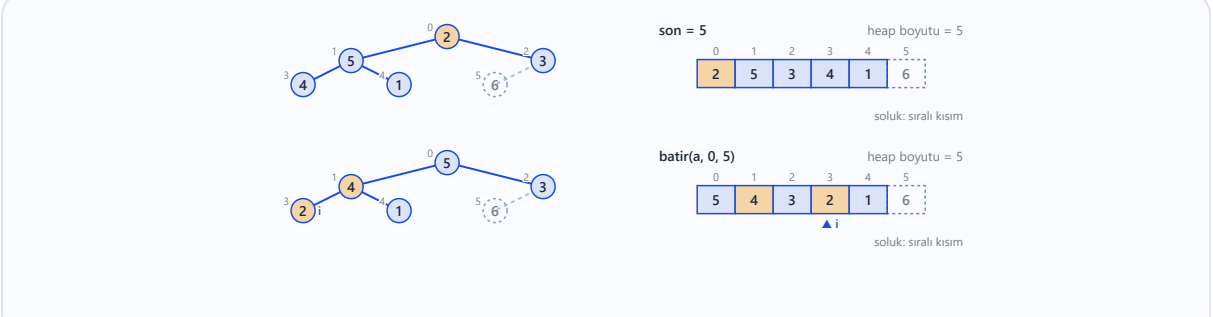
11.5.6 Kod izleme: heapSort([6, 5, 3, 4, 1, 2])

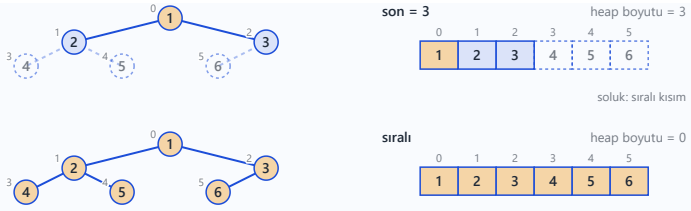
Daha küçük bir dizide ikinci aşamanın içine, yani her `batir` çağrısının adımlarına bakalım. Girdi `a = [6, 5, 3, 4, 1, 2]`'dir.

Birinci aşama. `k = 2, 1, 0` için `batir` çağrılır. Bu dizi zaten bir max-heap'tir: 6 ≥ 5, 3; 5 ≥ 4, 1; 3 ≥ 2. Her çağrı ilk karşılaştırmada döner ve hiç takas yapılmaz.

`son`	Kök ↔ `a[son]`	`batir(a, 0, son)`	Tur sonunda dizi
5	6 ↔ 2: [2, 5, 3, 4, 1 | 6]	`i = 0`: büyük çocuk 5, takas; `i = 1`: büyük çocuk 4, takas; indis 3 yaprak	[5, 4, 3, 2, 1 | 6]
4	5 ↔ 1: [1, 4, 3, 2 | 5, 6]	`i = 0`: büyük çocuk 4, takas; `i = 1`: tek çocuk 2, takas; indis 3 yaprak	[4, 2, 3, 1 | 5, 6]
3	4 ↔ 1: [1, 2, 3 | 4, 5, 6]	`i = 0`: büyük çocuk 3, takas; indis 2 yaprak	[3, 2, 1 | 4, 5, 6]
2	3 ↔ 1: [1, 2 | 3, 4, 5, 6]	`i = 0`: tek çocuk 2, takas; indis 1 yaprak	[2, 1 | 3, 4, 5, 6]
1	2 ↔ 1: [1 | 2, 3, 4, 5, 6]	heap boyu 1, kökün çocuğu yok	[1 | 2, 3, 4, 5, 6]
0	döngü koşulu `son > 0` yanlış		[1, 2, 3, 4, 5, 6]

Tablo 11.21: `heapSort([6, 5, 3, 4, 1, 2])`'nin ikinci aşaması.





Şekil 11.26: (a) `son = 5` : 6 kesin yerine oturdu, heap boyu 5. (b) Down-heap sonrası [5, 4, 3, 2, 1 | 6]. (c) `son = 3` : 4 de yerine oturdu. (d) Sonuç [1, 2, 3, 4, 5, 6].

Her turda sıralı kısım bir hücre büyür ve heap bir hücre küçülür. Beş turda kök ile son öğe arasındaki 5 takasa ek olarak `batir` 6 takas yapmıştır. İlk turlarda `batir` iki düzey iner; heap küçüldükçe yol da kısalır. Toplam maliyet $log_2\ n + log_2(n-1) + \ldots \le n\ log_2\ n$ ile sınırlıdır: $O(n\ log\ n)$.

Örnek 11.6

Heap sort'un kararlı olmadığını en küçük bir örnekle görelim. Dizi `[2a, 2b, 1]` olsun; 2a ve 2b eşit anahtarlı ama farklı iki öğedir ve 2a önde gelir.

1. **Oluşturma.** `k = 0` : çocuklar 2b ve 1; büyük çocuk 2b. 2a ≥ 2b olduğu için takas yok. Dizi değişmez.
2. `son = 2` . Kök 2a ile `a[2] = 1` takas edilir: `[1, 2b | 2a]` . `batir` : 1 < 2b, takas: `[2b, 1 | 2a]` .
3. `son = 1` . Kök 2b ile `a[1] = 1` takas edilir: `[1 | 2b, 2a]` .
4. Sonuç `[1, 2b, 2a]` 'dır. Anahtarlar doğru sıradadır, ama eşit iki öğenin sırası tersine dönmüştür.

11.5.7 Öncelikli kuyrukla sıralama

Heap sort, daha genel bir fikrin özel bir durumudur. Herhangi bir öncelikli kuyrukla sıralama yapılabilir: bütün öğeler kuyruğa eklenir, sonra *n* kez `removeMin` ile sırayla çıkarılır. Çıkan öğeler artan sıradadır. Elde edilen algoritma ve çalışma süresi, kuyruğun hangi veri yapısıyla gerçeklendiğine bağlıdır.

Gerçekleme	*n* × insert	*n* × removeMin	Ortaya çıkan algoritma
Sırasız liste	O(n)	$O(n^2)$	Seçimli sıralama (*selection sort*), $O(n^2)$
Sıralı liste	$O(n^2)$	O(n)	Eklemeli sıralama (*insertion sort*), $O(n^2)$
İkili heap	O(n log n)	O(n log n)	Heap sort, O(n log n)

Tablo 11.22: Öncelikli kuyrukla sıralama: gerçekleme hangi algoritmayı verir?

Sırasız listede her `removeMin` kalan öğeler arasında en küçüğü arar ve onu çıkarır. Bu, seçimli sıralamanın tam tanımıdır: her adımda en küçüğü seç ve sıraya koy. Sıralı listede ise her `insert` yeni öğeyi sıralı kısımda doğru yerine yerleştirir; bu da eklemeli sıralamanın tanımıdır. Heap'te ise iki aşamanın

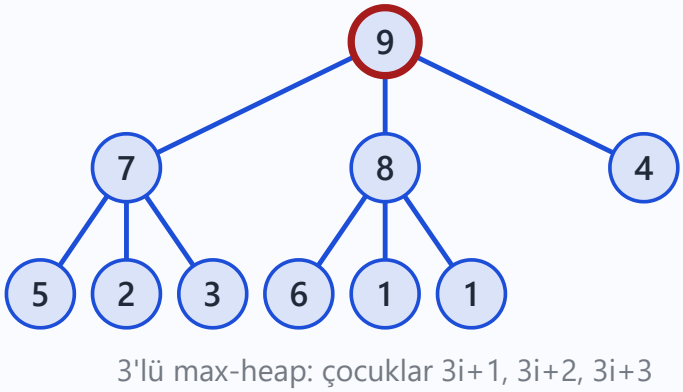
da maliyeti dengelidir.

Bu tablo, ADT ile veri yapısı arasındaki ayrımın somut bir örneğidir. Algoritmanın metni (“hepsini ekle, sonra hepsini çıkar”) üç durumda da aynıdır. Yalnız kuyruğun gerçeklemesi değişmiş ve çalışma süresi $O(n^2)$'den $O(n \log n)$'e inmiştir.

11.5.8 Heap türleri

İkili heap tek seçenek değildir. Farklı işlem karışımları için farklı heap türleri tasarlanmıştır.

d'li heap (*d-ary heap*), her düğümün en çok d çocuğu olduğu tam ağaçtır. Dizi gösterimi aynı mantıkla çalışır: 0 tabanlı gösterimde `i` indisli düğümün çocukları $di + 1$'den $di + d$'ye kadardır, ebeveyni $\lfloor (i-1)/d \rfloor$'dir. Örneğin 3'lü heap'te indis 7'nin ebeveyni $\lfloor 6/3 \rfloor = 2$'dir; indis 2'nin çocukları 7, 8 ve 9'dur.



Şekil 11.27: 3'lü max-heap: kök 9; çocukları 7, 8, 4; 7'nin çocukları 5, 2, 3; 8'in çocukları 6, 1, 1.

d'li heap'in yüksekliği $\log_d n$'dir ve d büyüdükçe ağaç alçalır. Up-heap her düzeyde tek karşılaştırma yaptığı için ekleme hızlanır: $O(\log_d n)$. Down-heap ise her düzeyde d çocuk arasından en büyüğünü seçmek zorundadır: silme $O(d \log_d n)$ olur. Eklemenin ve anahtar küçültmenin çok, silmenin az olduğu uygulamalarda (örneğin yoğun çizgelerde Dijkstra algoritması) d'li heap ikili heap'ten hızlı olabilir.

Binom heap ve **Fibonacci heap** tek bir ağaç değil, bir ağaç ormanıdır. Bu yapılar özellikle iki öncelikli kuyruğu *birleştirme* ve anahtar küçültme işlemlerini hızlandırmak için tasarlanmıştır. Fibonacci heap, Dijkstra ve Prim algoritmalarının kuramsal olarak en hızlı sürümlerinde kullanılır.

Min-heap türü	insert	removeMin	anahtar küçült	birleştir
İkili heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
d'li heap	$O(\log_d n)$	$O(d \log_d n)$	$O(\log_d n)$	$O(n)$
Binom heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Fibonacci heap	$O(1)$	$O(\log n)$*	$O(1)$*	$O(1)$

Tablo 11.23: Min-heap türlerinin işlem maliyetleri. * amortize.

İkili heap'i birleştirmenin *O(n)* sürmesinin nedeni, iki dizinin uç uca eklenip aşağıdan yukarıya yeniden heap oluşturulmasıdır. Fibonacci heap'in kuramsal üstünlüğüne karşın uygulamada ikili heap çoğu zaman en hızlısıdır. Kodu basittir, sabit çarpanları küçüktür ve dizi yapısı önbelleği iyi kullanır. Java'nın `PriorityQueue` sınıfı ve Python'un `heapq` modülü de ikili heap kullanır.

11.6 Java ve uygulamalar

Heap'i kendimiz yazmak, nasıl çalıştığını anlamanın en iyi yoludur. Gerçek programlarda ise çoğu zaman dilin hazır sınıfı kullanılır. Java'da bu sınıf `java.util.PriorityQueue`'dur. Bu kesim sınıfın kullanımını, içeride yaptığı işi ve heap'in iki tipik uygulamasını anlatır.

11.6.1 Java'da hazır sınıf: PriorityQueue

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.offer(5); pq.offer(1); pq.offer(8);
pq.peek();   // 1  (min-heap, O(1))
pq.poll();   // 1  (O(log n))

// max-heap için ters karşılaştırıcı
PriorityQueue<Integer> maxPq =
    new PriorityQueue<>(Comparator.reverseOrder());

// hasta: öncelik küçük olan önce
PriorityQueue<Hasta> acil = new PriorityQueue<>(
    Comparator.comparingInt(h -> h.oncelik));
```

`PriorityQueue<Integer>` varsayılan olarak bir **min-heap**'tir: öğeleri doğal sıralarına göre karşılaştırır ve en küçüğü öne alır. `offer` ekleme, `peek` en öndeki öğeye bakma, `poll` en öndeki öğeyi çıkarma işlemidir. Üç `offer`'dan sonra `peek()` 1'i döndürür ve kuyruğu değiştirmez; `poll()` da 1'i döndürür ama onu çıkarır. Boş kuyrukta `peek` ve `poll` `null` döndürür. Aynı işleri yapan `add`, `element` ve `remove` metotları ise bu durumda istisna fırlatır.

Sıralama ölçütünü değiştirmek için yapıcıya bir `Comparator` verilir. `Comparator.reverseOrder()` doğal sıranın tersini kullanır ve kuyruk bir max-heap gibi davranır: aynı 5, 1, 8 eklenirse `peek()` 8'i döndürür. Kendi tanımladığımız sınıflar için `Comparator.comparingInt(h -> h.oncelik)` gibi bir ifade, nesnenin hangi alanına göre karşılaştırılacağını söyler. Örnekteki `acil` kuyruğuna Ali (3), Ece (1), Can (2) ve Naz (1) eklenip sırayla çıkarılırsa önce önceliği 1 olan iki hasta, sonra Can, en son Ali çıkar.

Sınıfın dört özelliği bilinmelidir:

- **İç yapı.** İçeride dizi tabanlı bir ikili min-heap kullanır. Dizi dolunca kendiliğinden büyür.

- **Maliyetler.** `offer` ve `poll` *O(log n)*; `peek` ve `size` *O(1)*; `remove(nesne)` ve `contains` ise *O(n)*'dir, çünkü belirli bir nesne dizide doğrusal aranır.
- **Eşzamanlılık.** Sınıf iş parçacığı güvenli (*thread-safe*) değildir. Birden çok iş parçacığının aynı kuyruğu kullandığı programlarda `PriorityBlockingQueue` tercih edilir.
- **Eşit öncelik.** Eşit öncelikli öğeler için FIFO garantisi yoktur. Hasta örneğinde Ece'nin Naz'dan önce çıkması bu girdiye özgüdür; başka bir ekleme sırasında eşit öncelikli öğelerin sırası değişebilir.

> **Sık yapılan hata**
>
> `PriorityQueue`'nun varsayılan olarak max-heap olduğunu sanmak. Varsayılan sıralama doğal sıradır ve en küçük öğe öndedir. En büyüğün önde olması isteniyorsa yapıcıya `Comparator.reverseOrder()` verilir.

11.6.2 Kod izleme: PriorityQueue içeride ne yapar?

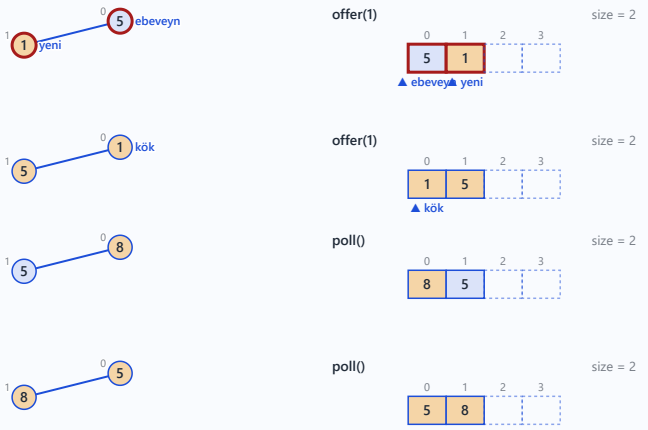
Aşağıdaki kısa program, sınıfın dışarıdan görünen davranışını gösterir. İzleme ise içerideki dizinin (Java kaynak kodundaki adıyla `queue`) her adımdaki durumunu verir.

```
PriorityQueue<Integer> pq =
    new PriorityQueue<>();        // min-heap
pq.offer(5);
pq.offer(1);
pq.offer(8);
System.out.println(pq.peek());    // bakar
System.out.println(pq.poll());    // çıkarır
System.out.println(pq.poll());
System.out.println(pq.size());
```

Çağrı	İçeride ne olur?	`queue`	Konsol
`new PriorityQueue<>()`	boş dizi	[ ]	—
`offer(5)`	5 köke yerleşir	[5]	—
`offer(1)`	indis 1'e eklenir; ebeveyn 5 > 1, `siftUp` ile köke çıkar	[1, 5]	—
`offer(8)`	indis 2'ye eklenir; ebeveyn 1 ≤ 8, up-heap gerekmez	[1, 5, 8]	—
`peek()`	kök döner, çıkarılmaz	[1, 5, 8]	1
`poll()`	kök 1 alınır; son öğe 8 köke taşınır; `siftDown`: 8, küçük çocuk 5 ile yer değiştirir	[5, 8]	1
`poll()`	kök 5 alınır; son öğe 8 köke taşınır; tek öğe, `siftDown` gerekmez	[8]	5

Çağrı	İçeride ne olur?	queue	Konsol
`size()`	sayaç okunur	[8]	1

Tablo 11.24: `PriorityQueue`'nun iç dizisi. Konsol sütunu programın yazdırdığı değerleri verir.



Şekil 11.28: (a) `offer(1)`: 1 indis 1'de, min-heap sırası bozuk. (b) `siftUp` sonrası [1, 5]. (c) `poll()`: son öğe 8 köke taşındı. (d) `siftDown` sonrası [5, 8].

Programın çıktısı dört satırdır: 1 (`peek`), 1 ve 5 (iki `poll`), 1 (`size`). Java'nın kaynak kodundaki `siftUp` ve `siftDown` metotları, bu notta `yuzdur` ve `batir` adıyla yazılan up-heap ve down-heap işlemlerinin min-heap sürümleridir. Hazır sınıf, aynı algoritmayı genel bir `Comparator` ile çalışacak biçimde gerçekler.

`PriorityQueue` nesnesi `println` ile yazdırıldığında iç dizi olduğu gibi yazılır. Sıralı bir liste beklenmemelidir. Örneğin 3, 1, 2 eklenen kuyruk `[1, 3, 2]` olarak yazdırılır. Kuyruğun `for-each` ile gezilmesi de aynı iç düzeni verir. Öğeleri sıralı almanın tek yolu onları `poll` ile tek tek çıkarmaktır.

Sık yapılan hata

"Heap sıralıdır" varsayımıyla `PriorityQueue`'yu yazdırıp ya da gezip sıralı sonuç beklemek. Heap'te yalnız kökün yeri bellidir; öbür öğeler yalnız heap-sıra özelliğini sağlar. `System.out.println(pq)` iç diziyi, yani heap sırasını yazar.

11.6.3 k'ıncı en büyük öğe

Bir dizideki k'ıncı en büyük öğeyi bulmak sık karşılaşılan bir problemdir: bir sınavda en yüksek üçüncü not, bir sunucuda en yoğun onuncu saat gibi. Diziyi sıralamak bir çözümdür, ama *O(n log n)* sürer ve *n* öğenin hepsini bellekte tutmayı gerektirir. Boyutu en çok *k* olan bir min-heap daha iyi bir çözüm verir:

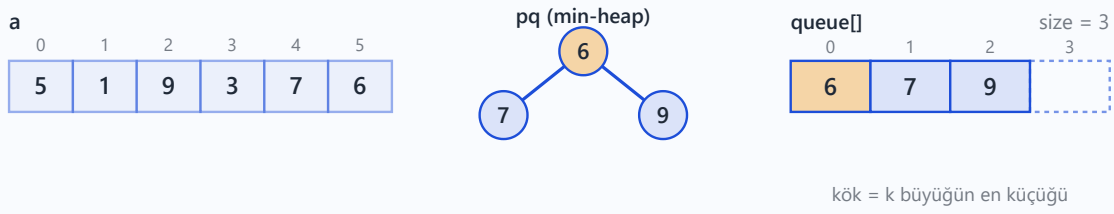
```
static int kinciEnBuyuk(int[] a, int k) {
  PriorityQueue<Integer> pq = new PriorityQueue<>();
  for (int x : a) {
    pq.offer(x);                    // O(log k)
    if (pq.size() > k) pq.poll();  // en küçüğü at
  }
  return pq.peek();       // k büyüğün en küçüğü
}
```

Algoritmanın değişmezi şudur: her adımın sonunda heap, o ana kadar görülen öğelerin en büyük *k* tanesini tutar. Min-heap olduğu için kökte bu *k* öğenin en küçüğü durur. Yeni bir öğe gelince heap'e eklenir. Heap *k + 1* öğeye ulaşırsa en küçüğü atılır, çünkü kendisinden büyük *k* öğe bulunan bir öğe en büyük *k* arasına giremez. Döngü bittiğinde kökteki öğe, dizinin en büyük *k* öğesinin en küçüğü, yani k'ıncı en büyük öğedir.

`kinciEnBuyuk([5, 1, 9, 3, 7, 6], 3)` çağrısını izleyelim.

`x`	`offer(x)` sonrası	`size > k` ?	Atılan	Tur sonunda heap
5	[5]	1 > 3? hayır	—	[5]
1	[1, 5] (1 köke çıktı)	2 > 3? hayır	—	[1, 5]
9	[1, 5, 9]	3 > 3? hayır	—	[1, 5, 9]
3	[1, 3, 9, 5] (3, 5 ile yer değiştirdi)	4 > 3? evet	1	[3, 5, 9]
7	[3, 5, 9, 7]	evet	3	[5, 7, 9]
6	[5, 6, 9, 7] (6, 7 ile yer değiştirdi)	evet	5	[6, 7, 9]
`peek()`				6 döner

Tablo 11.25: `kinciEnBuyuk([5, 1, 9, 3, 7, 6], 3)` . Heap sütunu iç diziyi verir.



Şekil 11.29: Döngü bittiğinde heap, dizinin en büyük üç öğesini (6, 7, 9) tutar; kökteki 6 üçüncü en büyük öğedir.

Atılan öğeler 1, 3 ve 5'tir; hepsi en büyük üçün dışında kalan öğelerdir. Dizinin sıralı hâli [1, 3, 5, 6, 7, 9]'dur ve üçüncü en büyük gerçekten 6'dır.

Maliyet: heap en çok *k + 1* öğe tutar. Her `offer` ve `poll` bu yüzden *O(log k)* sürer. *n* öğe için toplam zaman *O(n log k)*, ek bellek *O(k)*'dir. *k* küçükse bu, sıralamadan belirgin biçimde hızlıdır. Bellek kazancı da önemlidir: öğeler bir dosyadan ya da ağdan akıyorsa hepsini saklamak gerekmez, yalnız *k* tanesi

tutulur.

Sık yapılan hata

k'ıncı en büyük öğe için max-heap kullanmak. Max-heap'te bütün öğeler tutulmalı ve k kez `poll` yapılmalıdır: bellek $O(n)$ olur. Boyutu k ile sınırlı bir min-heap ise kökünde tam aranan öğeyi tutar ve küçük öğeleri hemen atar.

11.6.4 k sıralı listeyi birleştirme

Heap'in bir başka tipik kullanımı, kendi içinde sıralı k listeyi tek bir sıralı listede birleştirmektir. Büyük dosyaların parça parça sıralanıp sonra birleştirildiği dış sıralama bu işi yapar. Toplam öğe sayısı N olsun.

Yöntem şöyledir. Her listenin ilk öğesi bir min-heap'e konur; heap'teki her öğe hangi listeden geldiğini de bilir. Sonra şu adım heap boşalana kadar tekrarlanır: en küçük öğe çıkarılır ve çıktıya yazılır; o öğenin geldiği listede sıradaki öğe varsa heap'e eklenir. Heap her an her listeden en çok bir öğe, yani en çok k öğe tutar.

Örnek 11.7

Üç sıralı liste: A = [1, 7], B = [2, 3, 9], C = [4]. Heap öğeleri "değer(liste)" biçiminde yazılmıştır.

1. Başlangıç: heap {1(A), 2(B), 4(C)}.
2. 1(A) çıkar, çıktı [1]; A'dan 7 eklenir: {2(B), 4(C), 7(A)}.
3. 2(B) çıkar, çıktı [1, 2]; B'den 3 eklenir: {3(B), 4(C), 7(A)}.
4. 3(B) çıkar, çıktı [1, 2, 3]; B'den 9 eklenir: {4(C), 7(A), 9(B)}.
5. 4(C) çıkar; C bitti, ekleme yok: {7(A), 9(B)}. Sonra 7, en son 9 çıkar.
6. Sonuç: [1, 2, 3, 4, 7, 9].

Her öğe heap'e bir kez girer ve bir kez çıkar. Heap'in boyu en çok k olduğu için her işlem $O(\log k)$'dir ve toplam maliyet $O(N \log k)$'dir. Bütün öğeleri tek bir diziye toplayıp sıralamak $O(N \log N)$ sürerdi.

11.6.5 Öncelikli çizelgeleyici

İşletim sistemindeki çizelgeleyici (*scheduler*), işlemciyi bekleyen süreçler arasından hangisinin çalışacağına karar verir. Öncelikli çizelgelemede her sürecin bir öncelik düzeyi vardır. Sık kullanılan bir düzen, her düzey için ayrı bir FIFO kuyruğu tutmaktır.



Şekil 11.30: Üç öncelik düzeyinde üç FIFO kuyruğu: yüksekte P4, ortada P1 ve P5, düşükte P2 ve P3. İşlemci önce yüksek düzeydeki P4'ü seçer.

- **Kesin öncelik.** İşlemci her zaman en yüksek öncelikli, boş olmayan kuyruğun önündeki süreci seçer. Şekil 11.30'da önce P4 çalışır. Yüksek düzey boşalınca orta düzeydeki P1 ve P5'e sıra gelir.
- **Düzey içinde FIFO.** Aynı öncelikteki süreçler kendi kuyruklarında geliş sırasıyla çalışır. Her süreç belli bir zaman dilimi (*quantum*) kadar çalışıp kuyruğun sonuna dönüyorsa düzey içinde round robin uygulanmış olur.
- **Açlık ve yaşlandırma.** Yüksek öncelikli süreçler sürekli geliyorsa düşük öncelikliler hiç çalışamayabilir. Bu duruma **açlık** (*starvation*) denir. Yaygın çözüm **yaşlandırmadır** (*aging*): bir süreç bekledikçe önceliği adım adım artırılır ve sonunda üst düzeylere çıkıp çalışma fırsatı bulur.

Düzey sayısı az olduğunda her düzey için ayrı bir kuyruk tutmak en basit ve en hızlı çözümdür: en yüksek boş olmayan düzeyi bulmak sabit sayıda adımdır. Öncelikler geniş bir aralıktan geliyorsa (örneğin her sürecin bir son teslim zamanı varsa) tek bir heap kullanılır. Anahtar olarak (öncelik, geliş sıra numarası) çifti seçilirse aynı öncelikteki süreçlerin FIFO düzeni de korunur. Yaşlandırma ise bir anahtar değiştirme işlemidir ve uyarlanabilir öncelikli kuyrukla *O(log n)*'de yapılır.

11.7 Sık yapılan hatalar

Bölüm boyunca konuların yanında verilen hataların en sık görülenleri aşağıdaki tabloda toplanmıştır. Bir heap kodunu gözden geçirirken kontrol listesi olarak kullanılabilir.

Hata	Ne olur?	Doğrusu
"Heap sıralıdır"	3, 1, 2 eklenen `PriorityQueue` için `println(pq)` `[1, 3, 2]` yazar; sıralı çıktı beklenirse program yanlış çalışır.	Yalnız kökün yeri bellidir. Sıralı sonuç için `poll` ile tek tek çıkarılır.
İndis formülleri	0 ve 1 tabanlı formüller karışırsa düğümler yanlış ebeveyn ve çocuklara bağlanır, ağaç bozulur.	0 tabanlı: *2i + 1*, *2i + 2*, *(i − 1)/2*. 1 tabanlı: *2i*, *2i + 1*, *i/2*.
Sağ çocuk sınırı	`j + 1 < n` unutulursa heap dışındaki (silinmiş) öğe okunur ya da dizinin dışına taşılır.	`if (j + 1 < n && heap[j + 1] > heap[j]) j++;`
Yanlış çocuk	Max-heap'te down-heap küçük çocukla takas ederse sıra bozulur.	Max-heap'te büyük, min-heap'te küçük çocuk seçilir.

Hata	Ne olur?	Doğrusu
Heap'te arama	`contains` ve `remove(nesne)` *O(n)*'dir; sık kullanılırsa program yavaşlar.	Hızlı erişim için konum farkında öğe ya da ayrı bir eşleme tutulur.
Yön karışıklığı	`PriorityQueue` en büyüğü verecek sanılır.	Varsayılan min-heap'tir; max için `Comparator.reverseOrder()`.
Oluşturma döngüsünün yönü	Kökten yapraklara doğru `batir` çağrılırsa sonuç heap olmayabilir.	`k`, *n/2 − 1*'den 0'a azalarak yürür.
Eşitlikte FIFO beklentisi	Eşit öncelikli öğeler ekleme sırasından farklı çıkabilir.	Anahtar (öncelik, sıra numarası) çifti yapılır.

Tablo 11.26: Heap ve öncelikli kuyruk kodunda sık görülen hatalar.

Bu hataların çoğu küçük testlerde görünmez. Sağ çocuk sınırı hatası yalnız belirli boyutlarda, eşitlik hatası yalnız eşit anahtarlar varken ortaya çıkar. Bir heap gerçeklemesini sınamanın güvenilir yolu, her işlemden sonra `heapMi` ile diziyi denetlemek ve sonuçları hazır bir sınıfla (örneğin `PriorityQueue`) rastgele girdilerde karşılaştırmaktır.

11.8 Özet

- **Öncelikli kuyruk**, öğelerin geliş sırasına göre değil önceliğe göre çıktığı bir ADT'dir. Her öğe bir (anahtar, değer) çiftidir. Temel işlemleri `insert`, `min`, `removeMin`, `size` ve `isEmpty`'dir. Max-öncelikli kuyrukta `max` ve `removeMax` kullanılır. Eşit anahtarlarda FIFO garantisi yoktur; gerekiyorsa anahtara sıra numarası eklenir.
- **Liste ile gerçekleme**de sırasız liste *O(1)*'de ekler ama *O(n)*'de çıkarır; sıralı liste tersini yapar. İkisinde de *n* ekleme ve *n* çıkarma $O(n^2)$ sürer. Sıralı dizide en öncelikli öğe sonda, bağlı listede başta tutulur.
- **İkili heap**, heap-sıra özelliğini (max-heap'te ebeveyn ≥ çocuk) ve tam ikili ağaç özelliğini birlikte sağlar. En öncelikli öğe köktedir. Yükseklik $\lfloor \log_2 n \rfloor$'dir. Heap sıralı değildir ve ikili arama ağacı değildir.
- **Dizi gösterimi**nde düğümler düzey sırasıyla yazılır; boşluk ve referans yoktur. 0 tabanlı indislerde çocuklar *2i + 1*, *2i + 2*; ebeveyn $\lfloor (i-1)/2 \rfloor$'dir.
- **Ekleme** öğeyi `heap[n]`'e yazar ve up-heap ile yukarı taşır; **silMax** son öğeyi köke koyar ve down-heap ile büyük çocuk yönünde aşağı indirir. İkisi de *O(log n)*'dir. Down-heap'te sağ çocuk için `j + 1 < n` denetimi şarttır. Kapasite iki katına büyütülür; büyütme amortize *O(1)*'dir.
- **Uyarlanabilir öncelikli kuyruk** `replaceKey` ve `remove` işlemlerini ekler. Öğe dizideki yerini bilirse bu işlemler bir up-heap ya da down-heap ile *O(log n)*'de yapılır.
- **Aşağıdan yukarıya heap oluşturma** son iç düğümden ($\lfloor n/2 \rfloor - 1$) köke doğru down-heap uygular ve *O(n)*'dir. Down-heap'in maliyeti yüksekliğe bağlıdır ve düğümlerin çoğu alçaktır. *n* kez ekleme ise *O(n log n)*'dir.

- **Heap sort** önce heap oluşturur, sonra $n - 1$ kez kökü sona alıp heap'i küçültür. Yerinde çalışır, en kötü durumda $O(n \log n)$'dir, kararlı değildir. Öncelikli kuyrukla sıralama sırasız listede seçimli sıralamayı, sıralı listede eklemeli sıralamayı verir.
- **Java'nın** `PriorityQueue` sınıfı dizi tabanlı ikili min-heap'tir; max için `Comparator.reverseOrder()` verilir. Boyutu k olan bir min-heap k'ıncı en büyük öğeyi $O(n \log k)$'de bulur; aynı fikir k sıralı listeyi $O(N \log k)$'de birleştirir.

İşlem	Heap	Nasıl?
peek (max/min)	O(1)	`heap[0]`
ekle	O(log n)	`heap[n]` 'e koy, up-heap
silMax / removeMin	O(log n)	son öğeyi köke taşı, down-heap
anahtar değiştirme	O(log n)	büyüdüyse up-heap, küçüldüyse down-heap (max-heap)
heap oluşturma	O(n)	$\lfloor n/2 \rfloor - 1$'den 0'a down-heap
heap sort	O(n log n)	oluştur + $n - 1$ kez kökü sona al

Tablo 11.27: Heap işlemlerinin özeti.

11.9 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* · değer *value* · kuyruk *queue* · ilk giren ilk çıkar *FIFO* · arayüz *interface* · jenerik *generic* · ikili heap *binary heap* · heap *heap* · max-heap *max-heap* · min-heap *min-heap* · tam ikili ağaç *complete binary tree* · mükemmel ikili ağaç *perfect binary tree* · kök *root* · yaprak *leaf* · iç düğüm *internal node* · ebeveyn *parent* · çocuk *child* · kardeş *sibling* · düzey *level* · düzey sırası *level-order* · yükseklik *height* · derinlik *depth* · up-heap (yukarı süzme) *up-heap* · down-heap (aşağı süzme) *down-heap* · takas *swap* · kapasite *capacity* · dinamik dizi *dynamic array* · amortize analiz *amortized analysis* · uyarlanabilir öncelikli kuyruk *adaptable priority queue* · aşağıdan yukarıya heap oluşturma *bottom-up heap construction* · heap sort *heap sort* · yerinde *in-place* · kararlı *stable (sort)* · seçimli sıralama *selection sort* · eklemeli sıralama *insertion sort* · hızlı sıralama *quick sort* · round robin *round robin* · yaşlandırma *aging*

11.10 Kendini Dene

1. Boş bir min-öncelikli kuyruğa sırasıyla `insert(7, a)`, `insert(2, b)`, `insert(9, c)`, `removeMin()`, `insert(1, d)`, `min()`, `removeMin()`, `removeMin()` uygulanıyor. Değer döndüren çağrılar ne döndürür ve sonda kuyrukta ne kalır?

   Cevap

   İlk `removeMin()` (2, b)'yi döndürür; kuyruk {(7,a), (9,c)}. (1, d) eklenince en küçük odur: `min()` (1, d)'yi çıkarmadan döndürür. Sonraki `removeMin()` (1, d)'yi, ardından gelen (7, a)'yı döndürür. Sonda kuyrukta yalnız (9, c) kalır; `size()` 1'dir.

2. Kapasitesi 4 olan boş bir max-heap'e sırasıyla 3, 8, 6, 10, 1, 9 ekleniyor. Son dizi nedir? Toplam kaç takas yapılır ve kapasite ne zaman büyür?

   Cevap

   [3] → 8 eklenince 3 ile takas: [8, 3] → 6 yerinde kalır: [8, 3, 6] → 10 önce 3 ile, sonra 8 ile takas: [10, 8, 6, 3] → 1 eklenmeden önce dizi dolu olduğu için kapasite 8'e büyür; 1 yerinde kalır: [10, 8, 6, 3, 1] → 9, ebeveyni 6 ile takas edilir, 10'un altında durur. Son dizi `[10, 8, 9, 3, 1, 6]`; toplam 4 takas.

3. Max-heap `[20, 15, 18, 7, 12, 16, 4]` üzerinde art arda iki `silMax()` yapılıyor. Dönen değerler ve sondaki heap nedir?

   Cevap

   Birinci: 20 döner. 4 köke gelir: [4, 15, 18, 7, 12, 16]. Büyük çocuk 18: [18, 15, 4, 7, 12, 16]; indis 2'de tek çocuk 16: [18, 15, 16, 7, 12, 4]. İkinci: 18 döner. 4 köke gelir: [4, 15, 16, 7, 12]. Büyük çocuk 16: [16, 15, 4, 7, 12]; indis 2 yaprak. Son heap `[16, 15, 4, 7, 12]`.

4. 25 öğeli bir ikili heap'in yüksekliği nedir? Aşağıdan yukarıya oluşturmada döngü hangi indisten başlar ve kaç düğüm yapraktır? 0 tabanlı gösterimde indis 20'nin ebeveyni hangi indistedir?

   Cevap

   Yükseklik $\lfloor \log_2 25 \rfloor = 4$'tür ($16 \le 25 < 32$). Döngü $\lfloor 25/2 \rfloor - 1 = 11$'den başlar; indis 12'den 24'e kadar olan 13 düğüm yapraktır. İndis 20'nin ebeveyni $\lfloor 19/2 \rfloor = 9$'dur; çocuk adayı $41 \ge 25$ olduğundan 20 bir yapraktır.

5. `[5, 13, 2, 25, 7, 17, 20, 8, 4]` dizisi aşağıdan yukarıya yöntemle max-heap'e çevriliyor. Her

`batir(k)` 'dan sonra dizi nedir ve toplam kaç takas yapılır?

Cevap

n = 9, ilk `k = 3`. `k = 3`: 25 ≥ 8, 4; değişmez. `k = 2`: 2, büyük çocuk 20 ile takas: [5, 13, 20, 25, 7, 17, 2, 8, 4]. `k = 1`: 13, 25 ile takas; inen 13 ≥ 8, 4: [5, 25, 20, 13, 7, 17, 2, 8, 4]. `k = 0`: 5, sırasıyla 25, 13 ve 8 ile takas edilir: `[25, 13, 20, 8, 7, 17, 2, 5, 4]`. Toplam 5 takas.

6. `[5, 2, 8, 1, 9]` dizisi heap sort ile sıralanıyor. Oluşturmadan sonra ve her "kökü sona al + batir" turundan sonra dizi nedir?

Cevap

Oluşturma: `k = 1`'de 2 ↔ 9, `k = 0`'da 5 ↔ 9 ve ardından 5'in çocukları 1 ve 2'den büyük olduğu için durur: [9, 5, 8, 1, 2]. Turlar: [8, 5, 2, 1 | 9] → [5, 1, 2 | 8, 9] → [2, 1 | 5, 8, 9] → [1 | 2, 5, 8, 9]. Sonuç [1, 2, 5, 8, 9].

7. Max-heap `[15, 9, 12, 4, 8, 10, 3]` üzerinde `anahtarDegistir(1, 2)` çağrılıyor. Hangi işlem çalışır ve son dizi nedir?

Cevap

Anahtar 9'dan 2'ye küçüldüğü için `batir(1)` çalışır. İndis 1'in çocukları 4 ve 8'dir; büyük olan 8 (indis 4). 2 < 8, takas; indis 4 yapraktır. Son dizi `[15, 8, 12, 4, 2, 10, 3]`. Kökteki 15 ve sağ alt ağaç hiç incelenmedi.

8. `kinciEnBuyuk([3, 11, 6, 2, 9, 14, 5], 2)` ne döndürür? Döngü boyunca hangi öğeler atılır?

Cevap

Heap en çok iki öğe tutar. 3 ve 11'den sonra 6 gelince en küçük 3 atılır: {6, 11}. 2 eklenince hemen kendisi atılır. 9 gelince 6, 14 gelince 9, 5 gelince 5 atılır. Sonda heap {11, 14}'tür ve `peek()` 11'i döndürür: dizinin ikinci en büyük öğesi. Atılan öğeler sırasıyla 3, 2, 6, 9, 5'tir.

9. 1023 düğümlü mükemmel bir ikili ağaç (yükseklik 9) için en kötü durumda aşağıdan yukarıya oluşturma kaç takas yapar? Aynı öğeler artan sırada tek tek eklenseydi kaç takas yapılırdı?

Cevap

Yüksekliği *h* olan 2^{9-h} düğüm vardır ve her biri en çok *h* takas yapar: $\Sigma\, h \cdot 2^{9-h} = 1013$. Bu, $n - \log_2(n+1) = 1023 - 10$'a eşittir ve *n*'den küçüktür. Artan sırada eklemede her öğe köke kadar çıkar; derinliği *d* olan 2^d düğüm *d* takas yapar: $\Sigma\, d \cdot 2^d = 8 \cdot 2^{10} + 2 = 8194$. Fark sekiz kattan fazladır.

10. Max-heap `[10, 4, 9, 1, 3, 8, 7]`'den indis 4'teki öğe (3) uyarlanabilir kuyruğun `remove` işlemiyle siliniyor. Yerine hangi öğe gelir, hangi süzme gerekir ve son dizi nedir?

    Cevap

    Silinen öğe son öğe 7 ile takas edilir ve `n` 6'ya iner: [10, 4, 9, 1, 7, 8]. 7, ebeveyni 4'ten büyüktür; down-heap değil up-heap gerekir. 7 ile 4 yer değiştirir; yeni ebeveyn 10 ≥ 7 olduğu için durulur. Son dizi `[10, 7, 9, 1, 4, 8]`. Son öğe, silinen düğümün alt ağacından değil, ağacın öbür dalından gelmiştir.

11.11 İleri okuma

- M. T. Goodrich, R. Tamassia ve M. H. Goldwasser, *Data Structures and Algorithms in Java* (6. baskı, 2014), 9. bölüm: öncelikli kuyruk ADT'sini, liste ve heap gerçeklemelerini, konum farkında öğelerle uyarlanabilir öncelikli kuyruğu bu nottakine yakın bir Java koduyla anlatır.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (4. baskı, 2022), 6. bölüm: heap'i, aşağıdan yukarıya oluşturmanın doğrusal maliyetinin ispatını, heap sort'u ve heap tabanlı öncelikli kuyrukları biçimsel bir dille verir; nottaki on öğeli örnek bu kitaptandır.
- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 2.4: ikili heap'i 1 tabanlı dizide *swim* ve *sink* adlı işlemlerle anlatır; indeksli öncelikli kuyruk ve çok yollu birleştirme örnekleri içerir.
- R. W. Floyd, "Algorithm 245: Treesort 3" (1964): aşağıdan yukarıya heap oluşturmanın ilk yayımlandığı bir sayfalık kısa makaledir; Williams'ın heapsort'unu geliştirir.

11.12 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022.
2. R. W. Floyd, "Algorithm 245: Treesort 3", *Communications of the ACM*, 7(12), s. 701, 1964.
3. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6. baskı, Wiley, 2014.
4. Oracle, *Java SE 17 API Specification*: `java.util.PriorityQueue` belgesi.
5. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011.
6. J. W. J. Williams, "Algorithm 232: Heapsort", *Communications of the ACM*, 7(6), s. 347–348, 1964.

KISIM IV

Eşlemeler, Kümeler ve Çizgeler

Bölüm 12–14

KISIM IV · BÖLÜM 12

Eşlemeler ve Hash Tabloları

Bir telefon rehberinde Ayşe Demir'in numarasını ararken rehberi baştan sona okumayız. İsimden hareketle doğrudan doğru sayfaya gideriz. Eşleme, bu "anahtardan değere" erişimi soyut bir veri tipi olarak tanımlar. Hash tablosu ise eşlemeyi gerçeklemenin en yaygın yoludur: anahtarı bir hash fonksiyonundan geçirip bir dizi indisine çevirir ve öğeyi karşılaştırma yapmadan, ortalama sabit sürede bulur. Not, eşleme ADT'si ile başlar. Doğrudan adreslemenin neden yetmediğini gösterir, ardından hash fonksiyonlarını ve Java'daki `equals` / `hashCode` sözleşmesini anlatır. Çakışmaları çözmenin iki ana yolu, ayrı zincirleme ve açık adresleme, kodlarıyla ve adım adım izlemeleriyle işlenir. Son kesimler yük faktörünü, yeniden boyutlandırmayı, Java `HashMap`'in iç yapısını ve hash tablolarının birkaç tipik kullanımını ele alır.

Bu bölümün kodu: `github.com/sercankulcu/data-structures-java/Bolum12/src`

ÖĞRENME ÇIKTILARI

Eşleme modeli
Anahtar-değer ilişkisini ve hash tablosunun temel işleyişini açıklamak; bir işlem dizisinden sonra eşlemenin içeriğini ve dönen değerleri bulmak.

Çakışmalar
Ayrı zincirleme ve açık adresleme (doğrusal, karesel yoklama, çift hash) yöntemlerini elle ve kodla uygulamak; açık adreslemede silmenin neden özel işaret gerektirdiğini göstermek.

Yük faktörü
Yük faktörü, yeniden boyutlandırma ve anahtarların kovalara dağılımı arasındaki ilişkiyi yorumlamak; beklenen yoklama sayısını hesaplamak.

Doğruluk ve maliyet
`equals` / `hashCode` sözleşmesini uygulamak; hash tablosu işlemlerinin ortalama ve en kötü durum maliyetlerini gerekçesiyle karşılaştırmak.

12.1 Eşleme ve hash fikri

Dizi ve bağlı liste gibi yapılarda öğelere ya konumlarıyla (dizinin 5. hücresi) ya da sırayla (listenin başından itibaren) ulaştık. Gündelik hayatta ise bilgiyi çoğu zaman bir *ad* ile ararız. Öğrenci işleri bir öğrenciyi okul numarasıyla bulur. Sözlük bir kelimenin anlamını kelimenin kendisiyle verir. Derleyici bir değişkenin tipini değişkenin adına bakarak öğrenir. Bu kesim, bu tür aramaları tanımlayan eşleme ADT'sini ve onu hızlı yapmanın ardındaki fikri anlatır.

12.1.1 Eşleme (Map)

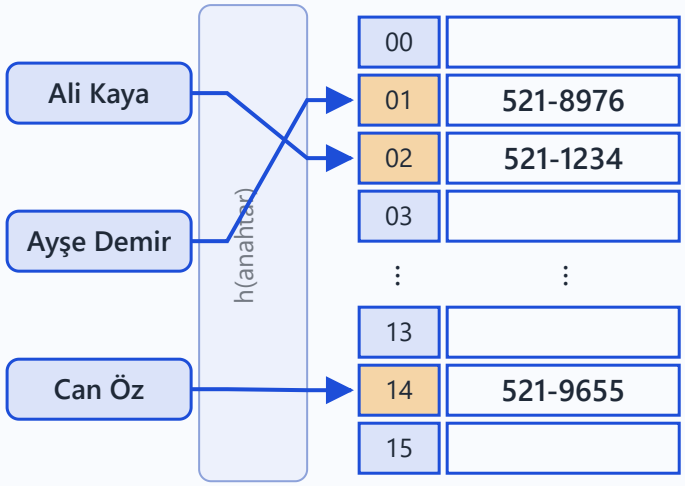
Bir telefon rehberini düşünelim. Rehberdeki her satırda bir isim ve bir telefon numarası vardır. İsim satırı tanımlar; numara ise o isme bağlı bilgidir. "Ayşe Demir'in telefonu nedir?" sorusunu sorarken elimizde isim vardır, aradığımız şey numaradır. Bu düzende isme **anahtar** (*key*), numaraya **değer** (*value*) denir.

Tanım 12.1

Eşleme (*map*), verileri **anahtar-değer** (*key-value*) çiftleri olarak saklayan soyut veri tipidir. Eşlemede her anahtar en çok bir kez bulunur; yani anahtarlar **tekildir**. Değere her zaman anahtar üzerinden erişilir.

Anahtarların tekil olması tanımın en önemli parçasıdır. Rehberde aynı isim iki kez geçerse "bu ismin numarası" sorusunun tek bir cevabı kalmaz. Eşleme bu belirsizliğe izin vermez: aynı anahtarla ikinci kez ekleme yapılırsa yeni bir çift oluşmaz, var olan çiftin değeri değişir. Değerler için böyle bir kısıt yoktur. İki farklı kişinin numarası aynı olabilir; iki farklı öğrencinin notu da aynı olabilir.

Şekil 12.1 eşlemeyi bir rehber örneği üzerinde gösterir. Üç isim, ortadaki bir fonksiyondan geçerek bir tablonun 02, 01 ve 14 numaralı yuvalarına gönderilir. Numaralar da bu yuvalarda saklanır. Şekildeki ortadaki kutu, bu bölümün asıl konusu olan *hash fonksiyonudur*. Görevi, bir ismi tablodaki bir yuva numarasına çevirmektir.



Şekil 12.1: Telefon rehberi bir eşlemedir. Ali Kaya, Ayşe Demir ve Can Öz isimleri `h(anahtar)` fonksiyonundan geçerek 02, 01 ve 14 numaralı yuvalara eşlenir; numaralar bu yuvalarda saklanır. Diğer yuvalar boştur.

Eşlemenin bir *soyut veri tipi* olduğunu vurgulamak gerekir. Soyut veri tipi (*abstract data type*, kısaca ADT) bir yapının *ne yaptığını* söyler: hangi işlemler vardır ve her işlem neyi değiştirir. *Nasıl yapıldığını* söylemez. Eşleme ADT'si sırasız bir diziyle, sıralı bir diziyle, bir arama ağacıyla ya da bir hash tablosuyla gerçeklenebilir. Bu gerçeklemelerin hepsi aynı soruları cevaplar; fark, cevabın ne kadar sürede geldiğindedir.

Hash tablosu, eşlemenin en yaygın gerçeklemesidir. Nedeni basittir: ekleme, arama ve silme hash tablosunda **ortalama *O(1)*** sürer. Tabloda bin öğe de olsa bir milyon öğe de olsa bir anahtarın değerine ortalama olarak aynı sürede ulaşılır. Sıralı dizide ikili arama bile *O(log n)* karşılaştırma yapar. Bu bölümün geri kalanı, bu "ortalama sabit süre"nin nereden geldiğini ve hangi koşullarda bozulduğunu anlatır.

Sık yapılan hata

Eşleme ile hash tablosunu aynı şey sanmak. Eşleme bir arayüzdür, hash tablosu ise bu arayüzü uygulayan bir veri yapısıdır. Java'da `Map` bir arayüzdür; `HashMap` onu hash tablosuyla, `TreeMap` ise dengeli bir arama ağacıyla gerçekler. İkisi de aynı `put` / `get` işlemlerini sunar, ama maliyetleri ve sağladıkları ek özellikler (örneğin sıralı gezinme) farklıdır.

12.1.2 Eşleme ADT'sinin temel işlemleri

Eşleme ADT'si dört temel işlemle tanımlanır: ekleme, arama, silme ve güncelleme. Java'nın `java.util.Map` arayüzü bu işlemleri `put`, `get` ve `remove` adlarıyla verir. Güncelleme için ayrı bir metot yoktur; var olan bir anahtarla yapılan `put` güncelleme anlamına gelir.

İşlem	`java.util.Map`	Açıklama
Ekleme (*insertion*)	`put(k, v)`	Çifti tabloya ekler.
Arama (*search*)	`get(k)`	`k` anahtarının değerini döndürür; anahtar yoksa `null` döndürür.
Silme (*deletion*)	`remove(k)`	`k` anahtarlı çifti çıkarır ve değerini döndürür.
Güncelleme (*update*)	`put(k, v')`	`k` zaten varsa değerini `v'` yapar ve eski değeri döndürür.
Diğer	`size()`, `isEmpty()`, `containsKey(k)`, `keySet()`, `values()`	

Tablo 12.1: Eşleme ADT'sinin temel işlemleri ve `java.util.Map` karşılıkları.

Aşağıdaki kısa kod parçası bu işlemleri art arda kullanır. Satır sonlarındaki yorumlar her çağrının sonucunu gösterir.

```
Map<Integer, String> m = new HashMap<>();
m.put(105, "Murat");   m.put(21, "Leyla");
m.put(21, "Ali");      // güncelleme: "Leyla" → "Ali"
m.get(21);             // "Ali"
m.remove(105);         // "Murat"
m.get(88);             // null: böyle bir anahtar yok
```

İlk satır, anahtarları `Integer`, değerleri `String` olan boş bir eşleme oluşturur. Sol taraftaki tür `Map` arayüzüdür; sağdaki `new HashMap<>()` ise bu arayüzün hash tablosuyla yapılmış gerçeklemesidir. Değişkeni arayüz türünde tanımlamak iyi bir alışkanlıktır: ileride `HashMap` yerine `TreeMap` kullanmak gerekirse yalnız bu satır değişir.

İkinci satırda iki çift eklenir: (105, "Murat") ve (21, "Leyla"). Üçüncü satırdaki `m.put(21, "Ali")` yeni bir çift eklemez, çünkü 21 anahtarı zaten vardır. Bu çağrı bir güncellemedir: 21'in değeri "Leyla"dan "Ali"ye döner ve eşlemede yine iki çift kalır. `m.get(21)` artık "Ali" döndürür. `m.remove(105)`, 105 anahtarlı çifti çıkarır ve değeri olan "Murat"ı döndürür. Son satırdaki `m.get(88)` ise `null` döndürür, çünkü eşlemede 88 anahtarı hiç yoktur.

`put`'un da bir dönüş değeri vardır, ama yukarıdaki kod bunu kullanmaz. Anahtar yeniyse `put` `null` döndürür. Anahtar varsa değeri günceller ve *eski* değeri döndürür. Bu ayrıntı, bir ekleme ile bir güncellemeyi kod içinde ayırt etmenin en kolay yoludur.

Çağrı	Eşleme (sonra)	`size()`	Döner
(başlangıç)	{ }	0	—
`put(105, "Murat")`	{105=Murat}	1	`null` (yeni anahtar)
`put(21, "Leyla")`	{105=Murat, 21=Leyla}	2	`null` (yeni anahtar)
`put(21, "Ali")`	{105=Murat, 21=Ali}	2	"Leyla" (eski değer)
`get(21)`	değişmez	2	"Ali"
`remove(105)`	{21=Ali}	1	"Murat"

Çağrı	Eşleme (sonra)	`size()`	Döner
`get(88)`	değişmez	1	`null` (anahtar yok)

Tablo 12.2: Kod parçasının adım adım izlemesi. `put`'un dönüş değeri de gösterilmiştir.

> **Sık yapılan hata**
>
> `get`'in `null` döndürmesini her zaman "anahtar yok" diye yorumlamak. `HashMap` değer olarak `null` saklamaya izin verir: `m.put(7, null)` geçerli bir çağrıdır. Bu durumda `m.get(7)` da `null` döndürür, ama 7 anahtarı eşlememededir. Anahtarın varlığı sorulacaksa `containsKey(k)` kullanılır.

12.1.3 Dizide sabit sürede erişim

Hash tablosunun fikrine bir soruyla yaklaşalım: bir dizide arama *O(1)* sürede yapılabilir mi? Öğeleri birbiriyle karşılaştırarak bu zordur. Sırasız bir dizide aranan öğe her yerde olabilir; en kötü durumda bütün hücrelere bakmak gerekir. Bu **doğrusal arama**dır ve *O(n)* sürer. Dizi sıralıysa **ikili arama** her adımda aralığı yarıya indirir ve *O(log n)* karşılaştırma yapar. Karşılaştırmaya dayanan hiçbir yöntem bundan iyisini genel olarak sağlayamaz.

Ama dizinin bir özelliği daha vardır. Aranan öğenin *konumu*, yani indisi biliniyorsa erişim tek adımdır:

```
int[] siraliDizi = {1, 3, 5, 7, 9};
int aranacakIndis = 2;                  // erişilmek istenen öğenin indisi
int oge = siraliDizi[aranacakIndis];    // O(1): oge = 5
```

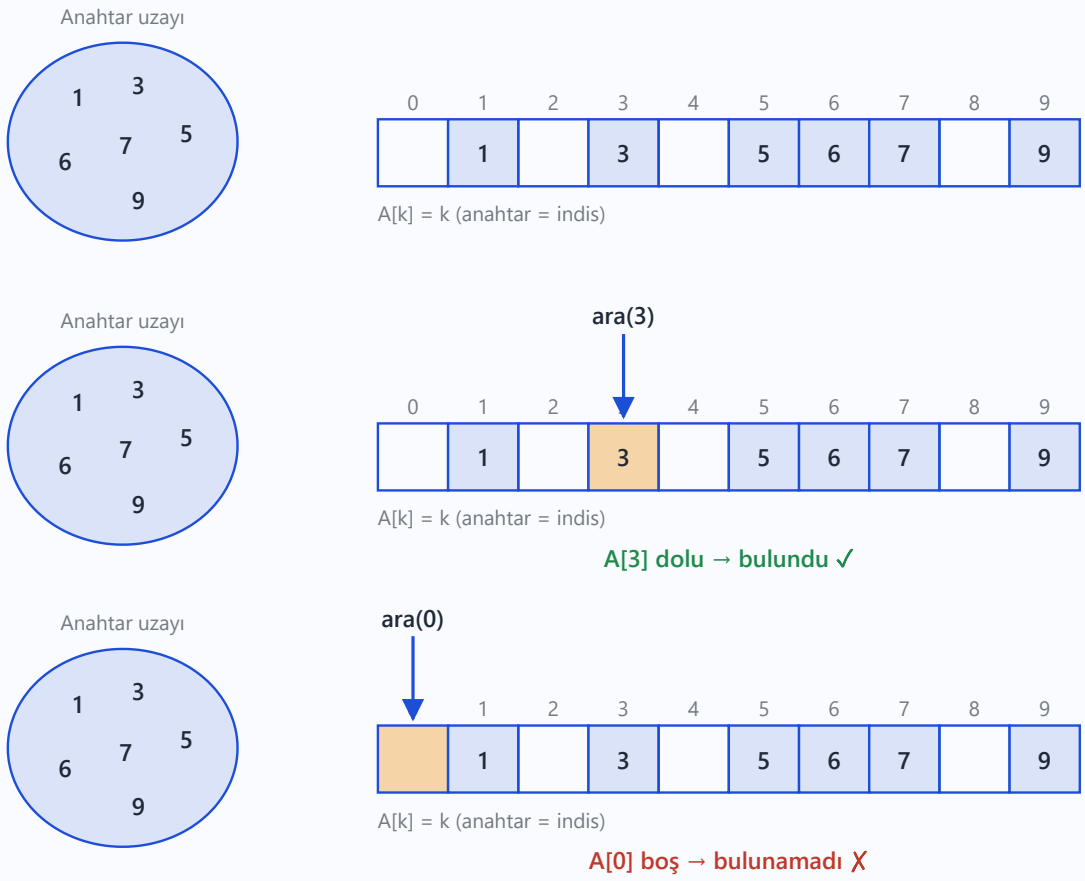
`siraliDizi[aranacakIndis]` ifadesi diziyi taramaz. Dizinin bellekteki başlangıç adresine *indis × hücre boyu* eklenir ve o adrese doğrudan gidilir. Bu hesap dizinin uzunluğundan bağımsızdır. Beş öğeli dizide de beş milyon öğeli dizide de aynı sürede yapılır. Bu yüzden indisle erişim *O(1)*'dir.

Buradan hash tablosunun temel fikri çıkar: **anahtarı doğrudan bir dizi indisine çevirebilirsek, karşılaştırma yapmadan tek adımda doğru hücreye gideriz.** Bölümün geri kalanı bu fikrin nasıl hayata geçirildiğini ve karşılaştığı engelleri anlatır.

12.1.4 İndisleme: anahtar = indis

En basit durumdan başlayalım. Anahtarlar 0 ile 9 arasındaki tam sayılar olsun. O zaman *k* anahtarını dizinin *k* numaralı hücresine koymak yeterlidir: 3 anahtarı `A[3]`'te, 7 anahtarı `A[7]`'de durur. Bir anahtarın tabloda olup olmadığını öğrenmek için yalnız o anahtarın hücresine bakılır. Bu yönteme **indisleme** denir.

Şekil 12.2 anahtar uzayı {1, 3, 5, 6, 7, 9} olan bir örneği gösterir. Her anahtar kendi numarasını taşıyan hücreye yerleşmiştir. Kullanılmayan 0, 2, 4 ve 8 numaralı hücreler boştur.



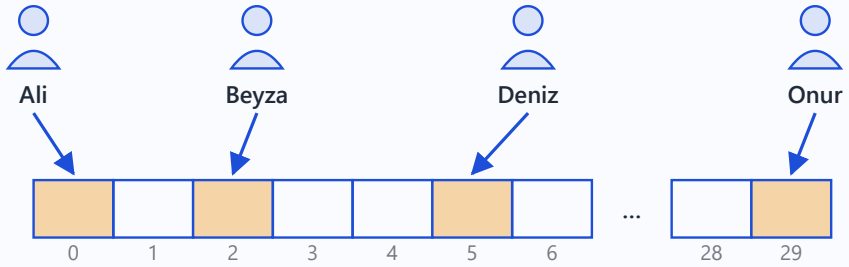
Şekil 12.2: İndisleme. Üstte: anahtar uzayındaki her *k* değeri `A[k]` hücresine yerleştirilir. Ortada: `ara(3)` doğrudan `A[3]`'e bakar; hücre dolu olduğu için 3 bulunur. Altta: `ara(0)` `A[0]`'a bakar; hücre boş olduğu için 0 tabloda yoktur. İki arama da tek adımdır.

Şeklin alt iki parçası, başarılı ve başarısız aramanın aynı maliyette olduğunu gösterir. `ara(3)` yalnız `A[3]`'e bakar ve dolu olduğunu görür. `ara(0)` yalnız `A[0]`'a bakar ve boş olduğunu görür. İkisinde de başka hiçbir hücre incelenmez. Ekleme ve silme de aynı şekilde tek hücreyi değiştirir.

12.1.5 Doğrudan adresleme ve sınırı

İndisleme fikrini genelleştirdiğimizde **doğrudan adresleme** (*direct addressing*) elde edilir. Olası her anahtar için dizide bir hücre ayrılır. Anahtar *k*, `A[k]` hücresinde saklanır. Bu tabloda arama, ekleme ve silme *O(1)*'dir ve tablo hiçbir karşılaştırma yapmaz.

Şekil 12.3 yöntemin zayıf tarafını gösterir. Ali, Beyza, Deniz ve Onur adlı dört kişi, 30 hücrelik bir dizinin 0, 2, 5 ve 29 numaralı hücrelerine yerleşmiştir. Kullanılan hücre dört, ayrılan hücre otuzdur. Diğer 26 hücre boş durur ama yine de bellekte yer kaplar.



Şekil 12.3: Doğrudan adresleme. Dört kişi 30 hücrelik dizinin 0, 2, 5 ve 29 numaralı hücrelerine yerleşir. Her olası anahtar için hücre ayrıldığından diğer 26 hücre boş kalır.

Doğrudan adreslemenin iki ciddi sınırı vardır:

- **Bellek israfı.** Anahtar uzayı büyükse ayrılması gereken dizi de büyür. Dokuz haneli okul numaralarını doğrudan adreslemek için 10^9 hücre gerekir. Her hücre 4 baytlık bir `int` olsa bile bu yaklaşık 4 GB eder. Oysa bir bölümde birkaç yüz öğrenci vardır. Hücrelerin milyonda birinden azı kullanılır.
- **Anahtar tam sayı olmayabilir.** "Ali" ya da "Beyza" gibi bir dizgi, dizi indisi olarak kullanılamaz. `A["Ali"]` diye bir erişim yoktur.

Çözüm, büyük ve belki tam sayı bile olmayan anahtar uzayını küçük bir diziye *sıkıştıran* bir fonksiyon kullanmaktır. Bu fonksiyon her anahtarı 0 ile $N - 1$ arasında bir indise çevirir; N burada dizinin boyudur ve saklanacak öğe sayısıyla orantılı seçilir. Böyle bir fonksiyona **hash fonksiyonu** denir.

Örnek 12.1

Bir fakültede okul numarası 9 haneli olan 800 öğrencinin not ortalaması saklanacaktır. Her ortalama 8 baytlık bir `double` değerdir. Doğrudan adresleme ile hash tablosunun bellek ihtiyacını karşılaştıralım.

1. **Doğrudan adresleme:** Olası numara sayısı 10^9'dur. Dizi $10^9 \times 8$ bayt, yani yaklaşık 8 GB yer tutar.
2. **Hash tablosu:** Öğrenci sayısının iki katı kadar kova yeterlidir: $N \approx 1600$. Her kovada numara (4 bayt) ve ortalama (8 bayt) birlikte saklansa bile tablo $1600 \times 12 = 19.200$ bayt, yani 20 KB'tan az yer tutar.
3. **Sonuç:** İki yöntem de ortalama sabit sürede çalışır, ama hash tablosu yaklaşık 400.000 kat daha az bellek kullanır. Bedel, iki numaranın aynı kovaya düşme olasılığıdır; bu olasılığı yönetmek bölümün asıl konusudur.

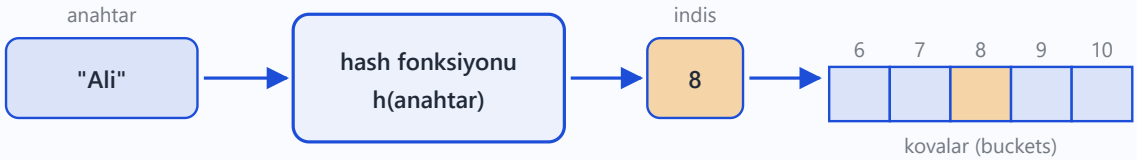
12.1.6 Hash tekniği

Hash tekniği, öğeleri saklamak ve onlara hızlı erişmek için kullanılır. Temel adım, bir öğe için dizide hızlıca bir konum bulmaktır. Bu konuma **kova** (*bucket*) ya da açık adreslemede **yuva** (*slot*) denir. Konum belli olunca öğeyi eklemek, çıkarmak ve güncellemek kolaydır, çünkü dizinin yalnız o hücresiyle uğraşılır.

Tanım 12.2

Hash fonksiyonu (*hash function*) h, her anahtarı 0 ile $N - 1$ arasında bir tam sayıya çeviren fonksiyondur. N kovalık bir dizi ve bir hash fonksiyonundan oluşan yapıya **hash tablosu** (*hash table*) denir. k anahtarlı öğe, $h(k)$ numaralı kovada saklanır.

Şekil 12.4 bu süreci tek bir anahtar için gösterir. "Ali" anahtarı hash fonksiyonuna girer, fonksiyon 8 değerini üretir ve öğe kovalar dizisinin 8 numaralı kovasına yerleşir. Daha sonra "Ali" arandığında aynı hesap yeniden yapılır, yine 8 bulunur ve doğrudan 8. kovaya bakılır.



Şekil 12.4: Hash tekniği. "Ali" anahtarı hash fonksiyonu h'den geçer ve 8 indisini verir; öğe kovalar dizisinin 8 numaralı kovasına yerleşir.

Bu 8 değeri nereden geldi? Java'da her nesnenin bir `hashCode()` metodu vardır. "Ali" dizgisi için bu metot 65918 döndürür. Tabloda 13 kova varsa 65918'in 13'e bölümünden kalan alınır: $65918 \bmod 13 = 8$. Yani “hash fonksiyonu” aslında iki adımlı bir hesaptır. Önce anahtar bir tam sayıya, sonra bu tam sayı tablonun aralığına çevrilir. Bu iki adım ve 65918 sayısının nasıl hesaplandığı sonraki kesimde ayrıntılı işlenir.

Sık yapılan hata

Hash tablosunun öğeleri “sıralı” tuttuğunu sanmak. Öğeler, hash fonksiyonunun verdiği kovalara dağılır. Bu dağılım anahtarların büyüklük sırasıyla ya da ekleme sırasıyla ilgili değildir. 8. kovadaki "Ali", 2. kovadaki bir anahtardan alfabede önce gelebilir. Bu yüzden hash tablosu “en küçük anahtar” ya da “sıralı listeleme” gibi sorulara hızlı cevap veremez.

12.2 Hash fonksiyonları

Hash tablosunun temelinde hash fonksiyonu durur. Fonksiyon anahtarı alır ve öğenin tablodaki konumunu üretir. Tablonun hızlı olup olmaması büyük ölçüde bu fonksiyonun kalitesine bağlıdır. İyi bir fonksiyon anahtarları kovalara dengeli dağıtır ve her işlem bir iki adımda biter. Kötü bir fonksiyon anahtarları birkaç kovaya yığar; o zaman tablo, bir listeden farksız hâle gelir.

12.2.1 İyi bir hash fonksiyonu

İyi bir hash fonksiyonundan dört özellik beklenir:

- **Belirleyici** (*deterministic*) **olmalı.** Aynı anahtar her zaman aynı indisi vermelidir. "Ali" ekleme sırasında 8. kovaya gönderilip arama sırasında 3. kovada aranırsa bir daha bulunamaz. Bu yüzden fonksiyon rastgele sayı, saat ya da başka bir dış etkene bağlı olamaz.
- **Hızlı olmalı.** Hash tablosunun bütün üstünlüğü, aramanın sabit sürede bitmesidir. Hash hesabı uzun sürerse bu üstünlük kaybolur. Fonksiyonun maliyeti *O(1)* kabul edilir. Bir dizginin hash'i aslında dizginin uzunluğu kadar sürer, ama anahtar uzunluğu tablodaki öğe sayısından bağımsız bir sabit sayılır.
- **Düzgün dağılım sağlamalı.** Anahtarları kovalara olabildiğince eşit yaymalıdır. Kovalar dengeli dolarsa her kovada az öğe olur ve aynı kovaya düşen anahtar sayısı, yani çakışma sayısı azalır.
- **Anahtarın tamamını kullanmalı.** Yalnız ilk harfe bakan bir fonksiyon "Ali", "Ayşe" ve "Ahmet" anahtarlarını aynı kovaya atar. Anahtarın her parçası sonucu etkilemelidir; yoksa bir parçada benzeşen anahtarlar hep çakışır.

Dördüncü özelliğin önemini somut sayılarla görelim. 13 kovalı bir tabloya "Ali", "Ayşe" ve "Ahmet" konacak olsun. Yalnız ilk harfin kodunu (A = 65) kullanan bir fonksiyon üçünü de *65 mod 13 = 0* numaralı kovaya gönderir. Java'nın bütün karakterleri kullanan `String.hashCode()` metoduyla hesap yapılırsa aynı üç anahtar 8, 6 ve 5 numaralı kovalara düşer. Aynı tablo ve aynı anahtarlar, yalnız fonksiyon değiştiği için üç çakışmadan sıfır çakışmaya geçmiştir.

Fonksiyonun ne kadar kötü olabileceğine en uç örnek, her anahtar için aynı indisi döndüren fonksiyondur. Bu durumda bütün öğeler tek bir kovada toplanır. Arama, o kovadaki öğeleri tek tek gezmek zorunda kalır ve *O(n)* sürer. Hash tablosu, karmaşık bir bağlı listeye dönüşmüş olur.

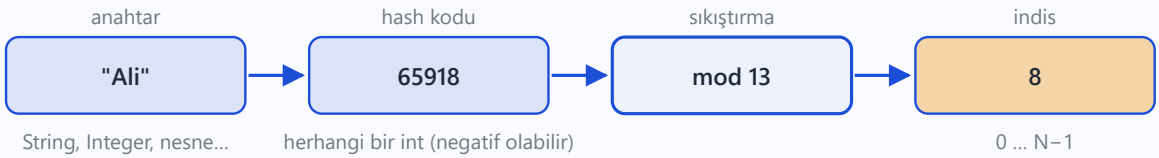
Fonksiyonun yanında **kapasite**, yani kova sayısı *N* da önemlidir. Kapasite büyük seçilirse boş kovalar belleği israf eder. Küçük seçilirse kovalar kalabalıklaşır ve çakışmalar artar. Pratikte *N*, saklanacak öğe sayısıyla orantılı tutulur. Öğe sayısı arttıkça tablo büyütülür (12.5.2).

> **Sık yapılan hata**
>
> Hash fonksiyonunda değişebilen bir değer kullanmak. Örneğin bir nesnenin hash'ini, nesnenin o anki sayacına ya da zamanına göre hesaplamak belirleyiciliği bozar. Öğe eklendikten sonra hash değişirse arama başka bir kovaya bakar ve öğeyi bulamaz. Aynı sorun, tabloya konduktan

sonra alanları değiştirilen anahtarlarda da görülür (12.2.3).

12.2.2 Hash kodu ve sıkıştırma

Bir hash fonksiyonu iki aşamada düşünülür. Şekil 12.5 bu aşamaları "Ali" anahtarı üzerinde gösterir.



Şekil 12.5: Hash fonksiyonunun iki aşaması. Anahtar ("Ali"; dizgi, tam sayı ya da herhangi bir nesne olabilir) önce bir **hash koduna** çevrilir: 65918. Bu kod herhangi bir `int` olabilir, negatif bile. Ardından **sıkıştırma** adımı kodu tablonun aralığına indirir: *65918 mod 13 = 8*, yani 0 ... *N − 1* arasında bir indis.

1. **Hash kodu** (*hash code*): Anahtar, tablodan bağımsız olarak bir tam sayıya çevrilir. Java'da bu iş her nesnenin `hashCode()` metoduyla yapılır. Hash kodu tablonun boyunu bilmez; çok büyük ya da negatif olabilir.
2. **Sıkıştırma** (*compression function*): Hash kodu, 0 ile *N − 1* arasındaki bir indise indirilir. Bu adım tablonun boyuna bağlıdır. Tablo büyütülürse aynı hash kodu farklı bir indise sıkıştırılır.

İki aşamayı ayırmanın pratik bir yararı vardır. Hash kodunu hesaplamak anahtarın sorumluluğudur: `String` sınıfı kendi kodunu, `Integer` sınıfı kendi kodunu hesaplar. Sıkıştırma ise tablonun işidir. Böylece aynı tablo her türden anahtarla çalışabilir ve aynı anahtar her boydaki tabloya konabilir.

En çok kullanılan iki sıkıştırma yöntemi şunlardır:

Bölme yöntemi. *h(k) = k mod N*. Hash kodunun *N*'ye bölümünden kalan alınır. Kalan her zaman 0 ile *N − 1* arasındadır (negatif kodlar için bkz. 12.3.4). Yöntem basit ve hızlıdır. Ama *N*'nin seçimine duyarlıdır. *N* asal seçilirse anahtarlardaki düzenli örüntüler, örneğin hep 10'un katı olan anahtarlar, aynı kovalara yığılmaz.

k	*k mod 10*	*k mod 11*
10	0	10
20	0	9
30	0	8
40	0	7
50	0	6

Tablo 12.3: 10'un katı olan anahtarlar için bölme yöntemi. *N = 10*'da beşi de aynı kovaya düşer; *N = 11*'de

(asal) beşi de farklı kovalara
dağılır.

Tablo 12.3'ün arkasındaki neden şudur. *N = 10* iken *k mod 10* yalnız sayının son hanesine bakar. Bütün anahtarların son hanesi 0 olduğu için hepsi 0. kovaya düşer. Anahtarın diğer haneleri sonucu hiç etkilemez; bu, "anahtarın tamamını kullanma" ilkesine aykırıdır. *N = 11* ise 10 ile ortak böleni olmayan bir sayıdır. 10'un katları 11'e bölündüğünde kalanlar 10, 9, 8, … diye ilerler ve kovalara yayılır. Genel kural: anahtarlar *d*'nin katlarıysa ve *d* ile *N*'nin en büyük ortak böleni *g* ise, anahtarlar kovaların yalnız *1/g*'sini kullanabilir. *N* asal olursa bu ortak bölen ancak *d*, *N*'nin katıysa 1'den büyük olur.

MAD yöntemi (*Multiply-Add-and-Divide*, çarp-topla-böl). $h(k) = ((a \cdot k + b) \bmod p) \bmod N$. Burada p, N'den büyük bir asal sayıdır; a ve b, tablo kurulurken rastgele seçilen sabitlerdir ve $a \neq 0$'dır. Çarpma ve toplama, anahtardaki düzenli örüntüleri karıştırır. Böylece N asal olmasa bile dağılım düzelir.

Örnek 12.2

10 kovalı tabloda ($N = 10$) 10, 20, 30, 40, 50 anahtarları için MAD yöntemi uygulansın: $a = 7$, $b = 3$, $p = 101$.

1. **k = 10:** $7 \cdot 10 + 3 = 73$; $73 \bmod 101 = 73$; $73 \bmod 10 = 3$.
2. **k = 20:** $143 \bmod 101 = 42$; $42 \bmod 10 = 2$.
3. **k = 30:** $213 \bmod 101 = 11$; $11 \bmod 10 = 1$.
4. **k = 40:** $283 \bmod 101 = 81$; $81 \bmod 10 = 1$.
5. **k = 50:** $353 \bmod 101 = 50$; $50 \bmod 10 = 0$.
6. **Sonuç:** Kovalar 3, 2, 1, 1, 0'dır. Bölme yönteminde $N = 10$ ile beş anahtar tek kovada toplanıyordu; MAD ile dört farklı kova kullanılır ve yalnız 30 ile 40 çakışır. Çakışmanın tamamen ortadan kalkmaması doğaldır: amaç çakışmayı azaltmaktır, sıfırlamak çoğu zaman mümkün değildir.

12.2.3 equals ve hashCode sözleşmesi

Java'da hash tablosu iki metoda dayanır. `hashCode()` anahtarın hangi kovaya gideceğini belirler. `equals()` ise kovadaki bir anahtarın aranan anahtarla gerçekten aynı olup olmadığını söyler. Tablonun doğru çalışması için bu iki metodun birbiriyle uyumlu olması gerekir. Java bu uyumu bir **sözleşme** (*contract*) olarak tanımlar.

Tanım 12.3

`equals` / `hashCode` **sözleşmesi:** `a.equals(b)` doğruysa `a.hashCode() == b.hashCode()` olmak zorundadır. Tersi gerekmez: hash kodları aynı olan iki nesne eşit olmayabilir. Ayrıca bir nesnenin `hashCode()` değeri, `equals` 'ta kullanılan alanları değişmediği sürece aynı kalmalıdır.

Sözleşmenin dört sonucu vardır:

- **Eşit nesneler.** `a.equals(b)` doğruysa iki nesnenin hash kodu aynıdır. Bu kural çiğnenirse eşit anahtarlar farklı kovalara düşer. `a` ile eklenen bir değer `b` ile aranınca bulunamaz.
- **Aynı hash kodu.** İki nesnenin hash kodunun aynı olması, nesnelerin eşit olduğu anlamına gelmez. Java'da "Aa" ve "BB" dizgilerinin `hashCode()` değeri aynıdır (ikisi de 2112), ama dizgiler farklıdır. Bu yüzden hash tablosu kovaya ulaştıktan sonra anahtarları ayrıca `equals` ile karşılaştırır.
- **Birlikte ezme.** Bir sınıf `equals`'ı yeniden tanımlıyorsa (*eziyorsa*, *override*) `hashCode`'u da aynı alanlara göre yeniden tanımlamalıdır. Yalnız birini ezmek sözleşmeyi bozar.
- **Değiştirilebilir anahtar.** Tabloya eklendikten sonra `equals` / `hashCode`'da kullanılan alanları değişen bir anahtar, doğru kovada aranamaz. Yeni hash kodu başka bir kovayı gösterir; öğe ise eski kovada kalmıştır.

Kısacası, hash tablosunda kova seçimini `hashCode`, anahtarın gerçekten aynı olup olmadığını `equals` belirler. Java 16 ile gelen `record` türleri bu işi kendiliğinden doğru yapar:

```
record Ogrenci(int no, String ad) {}
Map<Ogrenci, Double> notlar = new HashMap<>();
notlar.put(new Ogrenci(42, "Ada"), 87.5); // record: equals/hashCode hazır (Java 16+)
```

`record Ogrenci(int no, String ad) {}` satırı, `no` ve `ad` alanları olan değişmez bir sınıf tanımlar. Derleyici bu sınıf için iki alanı birlikte karşılaştıran bir `equals` ve iki alandan hesaplanan bir `hashCode` üretir. Bu yüzden `new Ogrenci(42, "Ada")` ile eklenen not, başka bir `new Ogrenci(42, "Ada")` nesnesiyle sorulduğunda bulunur. Ayrıca `record` alanları sonradan değiştirilemez; değiştirilebilir anahtar sorunu da baştan önlenmiş olur.

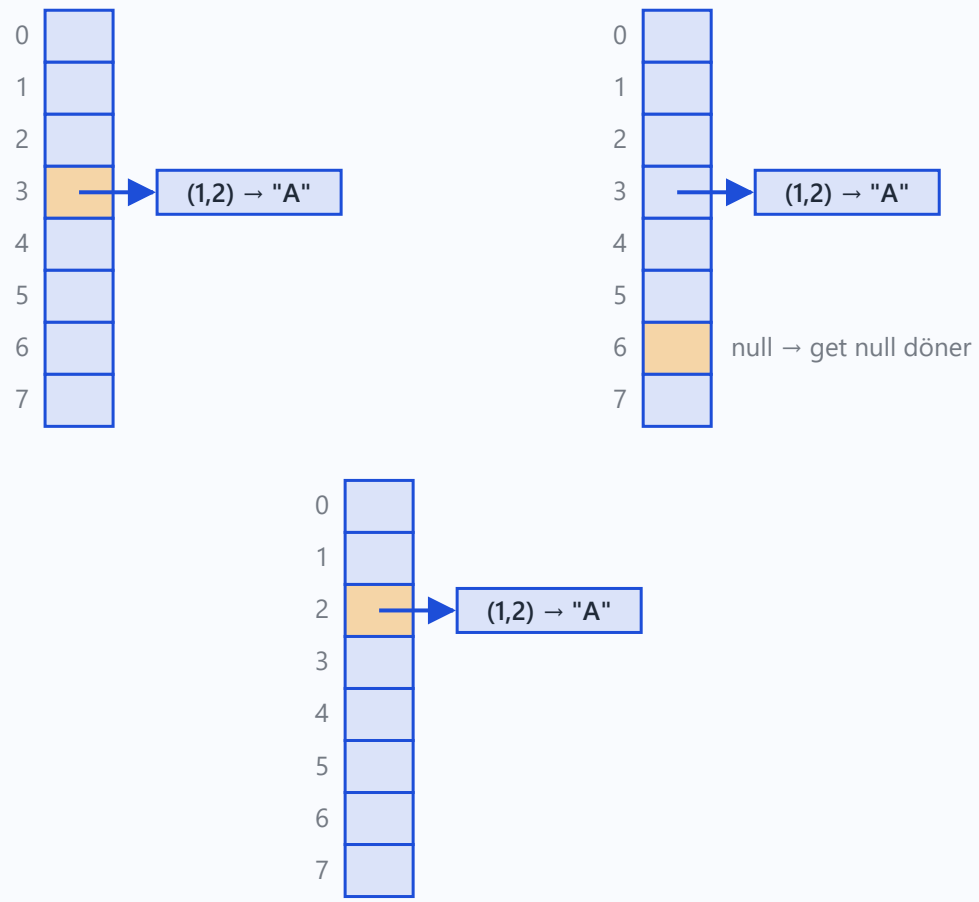
12.2.4 hashCode unutulursa

Sözleşmenin çiğnendiği en yaygın durum, `equals`'ın ezilip `hashCode`'un unutulmasıdır. Aşağıdaki `Nokta` sınıfı iki noktayı koordinatları aynıysa eşit sayar, ama `hashCode()` metodu yoktur.

```
class Nokta {
  int x, y;
  Nokta(int x, int y) { this.x = x; this.y = y; }
  public boolean equals(Object o) {
    return o instanceof Nokta p
        && x == p.x && y == p.y;
  }
  // hashCode() yok -> Object'inki
}
Map<Nokta, String> m = new HashMap<>();
m.put(new Nokta(1, 2), "A");
m.get(new Nokta(1, 2));  // büyük olasılıkla null (!)
```

`equals` metodundaki `o instanceof Nokta p` ifadesi iki iş yapar. Önce `o`'nun bir `Nokta` olup olmadığını sorar. Öyleyse onu `p` adlı bir `Nokta` değişkenine bağlar. Ardından iki koordinat karşılaştırılır. Bu metoda göre `new Nokta(1, 2)` ile ikinci bir `new Nokta(1, 2)` eşittir.

`hashCode()` ezilmediği için Java, `Object` sınıfındaki varsayılan metodu kullanır. Bu metot nesnenin *kimliğine* dayanır: her yeni nesne için genellikle farklı bir değer üretir. İki noktanın koordinatları aynı olsa da ayrı nesneler oldukları için kimlik hash'leri neredeyse her zaman farklıdır. Şekil 12.6 bunun sonucunu adım adım gösterir.



Şekil 12.6: `hashCode` unutulunca ne olur? (a) `put(p1, "A")` : kova `p1.hashCode()` ile seçilir; `Object.hashCode()` kimliğe bağlı olduğundan örneğin kova 3. (b) `get(p2)` : `p2` , `p1`'e eşittir ama ayrı nesnedir; kimlik hash'i farklıdır, örneğin kova 6'ya bakılır; kova boş, sonuç `null` . `equals` hiç çağrılmaz. (c) Çözüm: `hashCode()` da ezilir, `return Objects.hash(x, y);` . Eşit noktalar aynı kovaya düşer (`Objects.hash(1, 2) = 994` , 8 kovalı tabloda *994 mod 8 = 2*) ve kovada `equals` eşleşmeyi bulur.

Şekildeki (a) ve (b) adımlarındaki 3 ve 6 sayıları yalnız örnektir. Kimlik hash'i her çalıştırmada değişebilir. Ama sonuç neredeyse her zaman aynıdır: `get` yanlış kovaya bakar ve `null` döndürür. "Neredeyse" sözcüğü de önemlidir. İki kimlik hash'i tesadüfen aynı kovaya düşebilir. O zaman `get` doğru değeri bulur ve hata gizlenir. Böyle hatalar bazen çalışan, bazen çalışmayan programlara yol açtığı için bulunması en zor hatalardandır.

Burada bir ayrıntıya daha dikkat etmek gerekir. Java `HashMap`'i bir kovadaki düğümü incelerken önce hash kodlarını karşılaştırır, ancak kodlar eşitse `equals`'ı çağırır. Bu yüzden iki nokta tesadüfen aynı kovaya düşse bile kimlik hash'leri farklıysa `equals` çağrılmaz ve eşleşme yine bulunmaz. Kısacası `hashCode` ezilmeyen bir sınıfta, `equals` ne kadar doğru yazılmış olursa olsun, eşleme bu eşitliği görmez.

Çözüm, `hashCode`'u `equals`'ta kullanılan alanlarla ezmektir:

```
public int hashCode() {
  return Objects.hash(x, y);
}
```

`Objects.hash(x, y)`, verilen alanlardan `String.hashCode()`'a benzer bir hesapla tek bir sayı üretir: *31 · (31 · 1 + x) + y*. *x = 1, y = 2* için sonuç *31 · 32 + 2 = 994*'tür. Eşit iki noktanın koordinatları aynı olduğundan bu değer de aynıdır; iki nokta aynı kovaya düşer. Sözleşme sağlanmıştır. `Nokta(2, 1)` için sonuç *31 · 33 + 1 = 1024* olur; koordinatların sırası sonucu etkiler.

Sık yapılan hata

`hashCode`'u `equals`'ta kullanılmayan bir alanla hesaplamak. Örneğin `equals` yalnız öğrenci numarasına, `hashCode` ise numara ile ada birlikte bakıyor olsun. Numarası aynı ama adı farklı yazılmış iki kayıt `equals`'a göre eşittir, ama hash kodları farklıdır. Sözleşme yine bozulur. Kural: `hashCode`'da yalnız `equals`'ın baktığı alanlar kullanılır.

12.2.5 Örnek hash fonksiyonları

Uygulamada pek çok hash fonksiyonu kullanılır. Aşağıda dört tanesi, kısa Java kodlarıyla verilmiştir.

Mod alma (bölme kalanı). Tam sayı anahtarlar için en basit fonksiyondur:

```
int modulusHash(int anahtar, int diziBoyu) {
  return Math.floorMod(anahtar, diziBoyu);
}
```

Burada `%` yerine `Math.floorMod` kullanılır. Java'da `%` işleminin sonucu bölünenin işaretini alır; negatif bir anahtar için negatif bir kalan çıkabilir. `Math.floorMod` ise bölen pozitif olduğu sürece her zaman 0 … `diziBoyu - 1` aralığında bir sonuç verir. Bu fark 12.3.4'te ayrıntılı işlenir.

CRC32. Döngüsel artıklık denetimi (*Cyclic Redundancy Check*), aslında veri iletiminde bozulmaları yakalamak için tasarlanmış bir sağlama toplamıdır. Bitleri iyi karıştırdığı için hash fonksiyonu olarak da kullanılabilir. Java'da `java.util.zip` paketinde hazır gelir:

```
long crc32Hash(String anahtar) {
  CRC32 crc = new CRC32();  // java.util.zip
  crc.update(anahtar.getBytes());
  return crc.getValue();
}
```

Metot dizgiyi bayt dizisine çevirir, `CRC32` nesnesine verir ve 32 bitlik sonucu `long` olarak döndürür. Sonuç 0 ile $2^{32} - 1$ arasındadır; tabloda kullanmak için yine sıkıştırma gerekir. Örneğin "Ali" için bu metot 1933558706 döndürür.

DJB2. Daniel J. Bernstein'ın dizgiler için önerdiği, kısa ve hızlı bir fonksiyondur:

```
int djb2Hash(String s) {
  int h = 5381;
  for (int i = 0; i < s.length(); i++)
    h = (h * 33) ^ s.charAt(i);
  return h;
}
```

Fonksiyon 5381 ile başlar. Her karakterde o ana kadarki sonucu 33 ile çarpar ve karakterin koduyla birleştirir. Klasik DJB2 bu birleştirmeyi toplama ile yapar: `h * 33 + c`. Yukarıdaki kod ise XOR (`^`, bit düzeyinde dışlayan veya) kullanan biçimdir. İki biçim de yaygındır. Çarpma `int` sınırını kolayca aşar; sonuç taşar ve negatif olabilir. "Ali" için sonuç 193442657'dir, ama uzun dizgilerde negatif değerler sık görülür. Bu yüzden indis için yine güvenli bir sıkıştırma gerekir.

Java `String.hashCode()`. Java'nın dizgiler için kullandığı fonksiyondur:

```
// Σ s[i] · 31^(n-1-i)
"abc".hashCode()  // 96354
```

n karakterli bir dizgi için formül $s[0] \cdot 31^{n-1} + s[1] \cdot 31^{n-2} + \ldots + s[n-1]$'dir. Her karakter, dizgideki konumuna göre 31'in farklı bir kuvvetiyle çarpılır. "abc" için $97 \cdot 31^2 + 98 \cdot 31 + 99 = 93217 + 3038 + 99 = 96354$ bulunur. Konumun katsayıya girmesi sayesinde "abc" ile "cba" farklı kodlar alır. Yalnız karakter kodlarını toplayan bir fonksiyon ise bu iki dizgiyi ayırt edemezdi.

Neden 31? 31 hem tek hem asal bir sayıdır. Tek olduğu için çarpma sırasında taşan bitler bilgiyi tamamen yok etmez; çift bir çarpan her adımda en alt biti sıfırlardı. Ayrıca $31 \cdot h$, işlemcide $(h \ll 5) - h$ biçiminde bir kaydırma ve bir çıkarmayla hızla hesaplanabilir.

Sık yapılan hata

Farklı dizgilerin farklı hash kodu alacağını varsaymak. "Aa" ve "BB" dizgilerinin ikisi de 2112 değerini verir: *65 · 31 + 97 = 2112* ve *66 · 31 + 66 = 2112*. Bunlardan kurulan "AaAa", "AaBB" ve "BBBB"nin kodu da aynıdır: 2031744. Hash kodu bir kimlik numarası değildir. Eşitlik her zaman `equals` ile denetlenir.

12.2.6 Kod izleme: "Ali".hashCode()

Aşağıdaki `hash` metodu, `String.hashCode()` ile aynı hesabı açık bir döngüyle yapar. `main` bu metodu "Ali" için çağırır, sonra sonucu 13 kovalı bir tabloya sıkıştırır.

```
// String.hashCode() ile aynı hesap:
int hash(String s) {
  int h = 0;
  for (int i = 0; i < s.length(); i++)
    h = 31 * h + s.charAt(i);
  return h;
}

// main:
int kod = hash("Ali");                // 65918
int indis = Math.floorMod(kod, 13);   // 8
```

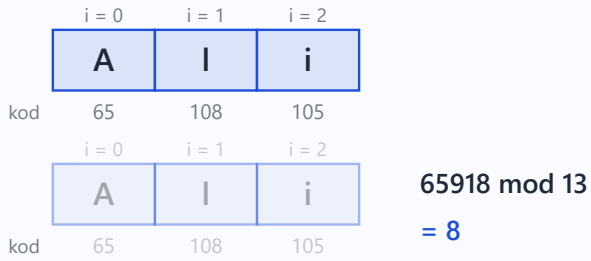
Metot `h = 0` ile başlar. Döngü dizginin her karakteri için bir kez döner. Her turda önceki `h` 31 ile çarpılır ve sıradaki karakterin kodu eklenir. `s.charAt(i)` bir `char` döndürür; Java bu değeri toplamada kendiliğinden karakterin Unicode koduna, yani bir tam sayıya çevirir. "Ali" dizgisinin karakter kodları A = 65, l = 108 ve i = 105'tir. Tablo 12.4 döngünün her turunu verir.

`i`	Koşul `i < 3`	`s.charAt(i)`	Hesap	`h` (sonra)
—	—	—	başlangıç	0
0	doğru	'A' (65)	*31 · 0 + 65*	65
1	doğru	'l' (108)	*31 · 65 + 108*	2123
2	doğru	'i' (105)	*31 · 2123 + 105*	65918
3	yanlış	—	döngü biter, `return`	65918

Tablo 12.4: `hash("Ali")` izlemesi.

Son değer formülle de doğrulanabilir: $65 \cdot 31^2 + 108 \cdot 31 + 105 = 62465 + 3348 + 105 = 65918$. Döngü bu kuvvetleri ayrı ayrı hesaplamaz. Her turda önceki toplamı 31 ile çarpmak, daha önce eklenen her karakterin katsayısını bir kuvvet yükseltir. Bu yöntem, polinom değerlendirmede bilinen **Horner yöntemidir**. *n* karakter için *n* çarpma ve *n* toplama yapar; maliyet *O(n)*, ek bellek *O(1)*'dir.

Son adım sıkıştırmadır: `Math.floorMod(65918, 13)`. *13 · 5070 = 65910* ve *65918 − 65910 = 8*'dir. "Ali" 13 kovalı tabloda 8. kovaya düşer. Şekil 12.7 izlemenin başını ve sonunu gösterir.



Şekil 12.7: "Ali".hashCode() izlemesi. Üstte: karakterler ve Unicode kodları (A = 65, l = 108, i = 105). Altta: döngü bitmiş, `kod = 65918`; sıkıştırma *65918 mod 13 = 8* verir.

Kısa dizgilerde sonuç pozitif çıkar, ama uzun dizgilerde çarpım `int` sınırını ($2^{31} - 1 = 2.147.483.647$) aşar. Java'da tam sayı taşması hata vermez; sayı sessizce sarılır ve negatif olabilir. Örneğin "Sercan".hashCode() −1821977712'dir. Daha ilginç bir örnek "polygenelubricants" dizgisidir: bu dizginin hash kodu tam olarak `Integer.MIN_VALUE`, yani −2147483648'dir. Bu değer 12.3.4'te önemli bir tuzağı gösterecektir.

12.2.7 Kod izleme: modulusHash()

Bölme yöntemini 10 hücrelik bir tablo üzerinde izleyelim. Kod beş anahtarı sırayla alır, her biri için hücre indisini hesaplar ve anahtarı o hücreye yazar.

```
int modulusHash(int anahtar, int diziBoyu) {
  return Math.floorMod(anahtar, diziBoyu);
}

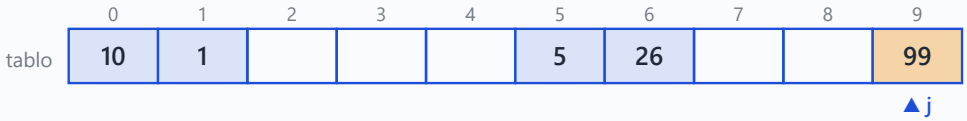
// main:
Integer[] tablo = new Integer[10];
for (int k : new int[]{5, 1, 10, 26, 99}) {
  int j = modulusHash(k, tablo.length);
  tablo[j] = k;
}
```

`Integer[] tablo = new Integer[10]` on hücrelik bir dizi oluşturur. Dizi `int` değil `Integer` türünde olduğu için her hücre başlangıçta `null`'dır; böylece boş hücre ile 0 değeri birbirinden ayrılır. Döngüdeki `new int[]{5, 1, 10, 26, 99}` ifadesi anahtarları sırayla verir. Her turda `modulusHash(k, tablo.length)` anahtarın 10'a bölümünden kalanı döndürür ve anahtar o hücreye yazılır.

`k`	`k mod 10`	**Yazılan hücre**	`tablo` **(sonra)**
—	—	—	[·, ·, ·, ·, ·, ·, ·, ·, ·, ·]

`k`	`k mod 10`	Yazılan hücre	`tablo` (sonra)
5	5	`tablo[5]`	[·, ·, ·, ·, ·, 5, ·, ·, ·, ·]
1	1	`tablo[1]`	[·, 1, ·, ·, ·, 5, ·, ·, ·, ·]
10	0	`tablo[0]`	[10, 1, ·, ·, ·, 5, ·, ·, ·, ·]
26	6	`tablo[6]`	[10, 1, ·, ·, ·, 5, 26, ·, ·, ·]
99	9	`tablo[9]`	[10, 1, ·, ·, ·, 5, 26, ·, ·, 99]

Tablo 12.5: `modulusHash` izlemesi. Tablo 0'dan 9'a yazılmıştır; boş hücreler "·" ile gösterilmiştir.



Şekil 12.8: `modulusHash` izlemesinin sonu: 5, 1, 10, 26 ve 99 anahtarları 5, 1, 0, 6 ve 9 numaralı hücrelerdedir. Beş anahtar beş farklı hücreye düşmüştür.

Beş anahtar beş farklı hücreye yerleşmiş, hiçbir hücreye iki anahtar düşmemiştir. Bu bir şanstır. Sırada 15 olsaydı *15 mod 10 = 5* hesaplanırdı ve 15, 5'in hücresine yazılırdı. Kod 5'in üzerine yazar ve 5 kaybolurdu. Bu tablo, çakışmaları hiç ele almadığı için yalnız gösterim amaçlıdır. Her ekleme *O(1)*, beş ekleme toplam *O(5)* sürmüştür; tablo 10 hücrelik sabit bir alan kullanır.

12.2.8 Çakışmalar

Tanım 12.4

İki farklı anahtar $k_1 \neq k_2$ için $h(k_1) = h(k_2)$ oluyorsa bu duruma **çakışma** (*collision*) denir.

Örneğin 10 kovalı tabloda bölme yöntemiyle $h(5) = h(105) = 5$'tir. 5 ile 105 farklı anahtarlardır ama aynı indisi üretirler. İki öğe aynı hücrede doğrudan saklanamayacağı için tablonun bu durumu çözen bir stratejisi olmalıdır.

Çakışmalar tesadüf değildir; anahtar uzayı tablodan büyük olduğu sürece kaçınılmazdır. Bunun nedeni basit bir sayma ilkesidir.

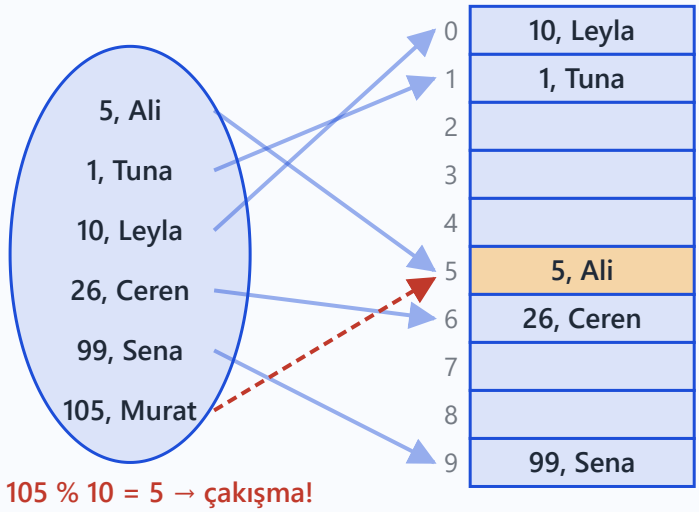
Teorem 12.1 (Güvercin yuvası ilkesi)

N kovalı bir tabloya N'den fazla farklı anahtar gönderen her hash fonksiyonunda en az bir çakışma vardır. Daha genel olarak, olası anahtar sayısı U ise en az bir kovaya en az $\lceil U / N \rceil$ farklı anahtar düşer.

İspat. *N + 1* anahtarın her biri *N* kovadan birine gider. Her kovada en çok bir anahtar olsaydı toplam en çok *N* anahtar olurdu; oysa *N + 1* anahtar vardır. Demek ki en az bir kovada iki anahtar bulunur. Genel durumda da her kovaya *⌈U / N⌉*'den az anahtar düşseydi toplam *U*'dan az olurdu. □

Teorem bize hash fonksiyonu tasarımının sınırını söyler. Dokuz haneli okul numaraları için $U = 10^9$'dur ve tablo 1000 kovalıysa en kalabalık kovaya en az bir milyon olası numara gider. İyi bir fonksiyon çakışmayı yok edemez; yalnız *gerçekte gelen* anahtarların kovalara dengeli dağılmasını umabilir. Çakışma, öngörülenden de erken başlar. 365 kovalı bir tabloya rastgele 23 anahtar konursa en az bir çakışma olma olasılığı yaklaşık 0,507'dir. Bu, "doğum günü problemi" olarak tanınan sonuçtur: 23 kişilik bir sınıfta iki kişinin aynı gün doğmuş olma olasılığı yarıdan fazladır.

Şekil 12.9 çakışmayı bir örnekle gösterir. Altı anahtar-değer çifti 10 yuvalı tabloya bölme yöntemiyle yerleştiriliyor. İlk beşi boş yuvalara düşer. (105, Murat) çifti ise 5. yuvaya gitmek ister, ama orada (5, Ali) vardır.



Şekil 12.9: Çakışma. Çiftler 10 yuvalı tabloya *k mod 10* ile yerleşir: 5 → 5, 1 → 1, 10 → 0, 26 → 6, 99 → 9. (105, Murat) çifti de 5. yuvaya gitmek ister, ama orada (5, Ali) vardır: *105 % 10 = 5*, çakışma.

Çakışmaları çözmenin iki ana stratejisi vardır:

- **Ayrı zincirleme** (*separate chaining*): Her kova, o kovaya düşen bütün öğeleri bir bağlı listede tutar. Çakışan öğe listeye eklenir.
- **Açık adresleme** (*open addressing*): Bütün öğeler dizinin kendisinde durur. Dolu bir yuvaya gelen öğe için belirli bir kurala göre dizide başka bir boş yuva aranır.

Sonraki iki kesim bu stratejileri kodlarıyla ve izlemeleriyle ele alır.

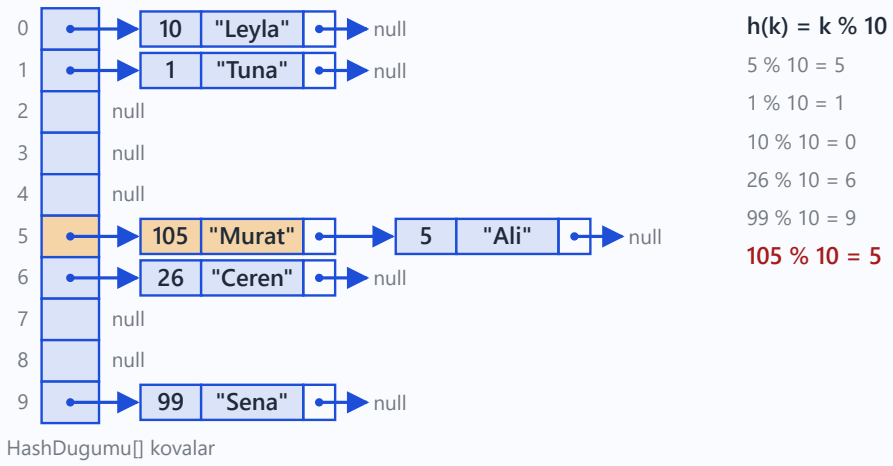
12.3 Ayrı zincirleme

Ayrı zincirleme, çakışmayı en doğrudan yoldan çözer: bir kovaya birden çok öğe düşüyorsa kova bu öğelerin hepsini tutar. Bunun için her kova tek bir öğe yerine bir bağlı listenin başını saklar. Bu listeye **zincir** (*chain*) denir. Yöntemin adındaki "ayrı" sözcüğü, her kovanın kendi zincirini tutmasından gelir.

12.3.1 Zincirleme fikri

10 kovalı bir tabloya *h(k) = k % 10* ile sırasıyla (5, "Ali"), (1, "Tuna"), (10, "Leyla"), (26, "Ceren"), (99, "Sena") ve (105, "Murat") çiftlerini ekleyelim. Her ekleme şu adımları izler:

- *h(5) = 5*: 5. kova boştur; (5, "Ali") düğümü kovaya bağlanır.
- *h(1) = 1*: 1. kova boştur; (1, "Tuna") düğümü kovaya bağlanır.
- *h(10) = 0*: 0. kova boştur; (10, "Leyla") düğümü kovaya bağlanır.
- *h(26) = 6*: 6. kova boştur; (26, "Ceren") düğümü kovaya bağlanır.
- *h(99) = 9*: 9. kova boştur; (99, "Sena") düğümü kovaya bağlanır.
- *h(105) = 5*: 5. kova doludur, orada (5, "Ali") vardır. Bu bir çakışmadır. Yeni düğüm zincirin **başına** eklenir: kova artık önce 105'i, 105 de 5'i gösterir.



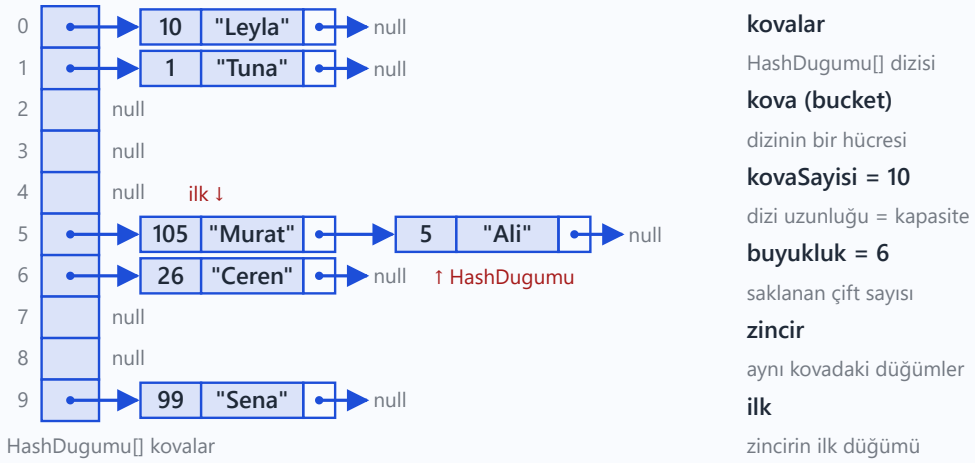
Şekil 12.10: Altı çiftten sonra ayrı zincirlemeli tablo. `HashDugumu[] kovalar` dizisinin her hücresi bir zincirin ilk düğümünü ya da `null` 'ı tutar. 105 ile 5 aynı kovaya düşmüştür (*105 % 10 = 5*); 105 sonra geldiği için zincirin başındadır: 5. kova → (105, "Murat") → (5, "Ali") → `null`.

Yeni düğüm neden zincirin sonuna değil de başına eklenir? Kova, zincirin yalnız ilk düğümünü bilir. Sona eklemek için zincirin sonuna kadar yürümek gerekir. Başa eklemek ise yalnız iki referans ataması ister: yeni düğüm eski başı gösterir, kova yeni düğümü gösterir. Bu işlem zincirin uzunluğundan bağımsızdır, yani *O(1)*'dir. Ancak 12.3.5'te görüleceği gibi ekleme öncesinde anahtarın zincirde zaten olup olmadığı denetlenir. Bu denetim zaten bütün zinciri gezer. Bu yüzden Java `HashMap` gibi bazı gerçeklemeler düğümü sona ekler; iki seçim de doğrudur.

Zincirlemede bir kovaya kaç öğe düşerse düşsün tablo dolmaz. Zincirler gerektiği kadar uzar. Bu esneklik yöntemin en büyük üstünlüğüdür, ama bir bedeli vardır: zincir uzadıkça arama yavaşlar. Bu yüzden zincirlerin ortalama uzunluğu, yani tablonun ne kadar dolu olduğu izlenir.

12.3.2 Hash tablosu terimleri

Kod okumayı kolaylaştırmak için zincirlemeli tablonun parçalarına ad verelim. Şekil 12.11 bu adları aynı altı çiftli tablo üzerinde gösterir.



Şekil 12.11: Zincirlemeli hash tablosunun terimleri: `kovalar` (`HashDugumu[]` dizisi), **kova** (dizinin bir hücresi), `kovaSayisi = 10` (dizi uzunluğu, kapasite), `buyukluk = 6` (saklanan çift sayısı), **zincir** (aynı kovadaki düğümler), `ilk` (zincirin ilk düğümü) ve `HashDugumu` (anahtar, değer ve sonraki referansı tutan düğüm).

Terim	Koddaki ad	Anlamı	Şekildeki değer
Kova dizisi	`kovalar`	Her hücresi bir zincirin başını tutan dizi.	10 hücre
Kova	`kovalar[i]`	Dizinin bir hücresi; boşsa `null`.	0 … 9
Kapasite	`kovaSayisi`	Kova sayısı, *N*.	10
Büyüklük	`buyukluk`	Saklanan çift sayısı, *n*.	6
Zincir	—	Aynı kovaya düşen düğümlerin listesi.	5. kova: 105 → 5
Düğüm	`HashDugumu`	Anahtar, değer ve sonraki düğüm referansı.	(26, "Ceren")

Tablo 12.6: Zincirlemeli hash tablosunun terimleri ve koddaki karşılıkları.

Kapasite ile büyüklüğü karıştırmamak gerekir. Kapasite tablonun kaç kovası olduğunu, büyüklük tabloda kaç çift bulunduğunu söyler. Şekildeki tabloda 10 kovanın 5'i doludur ama çift sayısı 6'dır, çünkü bir kovada iki çift vardır. Zincirlemede büyüklük kapasiteyi aşabilir; 10 kovalı tabloda 25 çift de saklanabilir.

12.3.3 Kod: HashTablosu sınıfı

Aşağıdaki sınıf, anahtarları `Integer`, değerleri `String` olan zincirlemeli bir hash tablosunun iskeletidir.

```
public class HashTablosu {
  private HashDugumu[] kovalar;   // zincirlerin ilk düğümleri
  private int kovaSayisi, buyukluk;

  public HashTablosu(int kapasite) {
    this.kovaSayisi = kapasite;
    this.kovalar = new HashDugumu[kapasite];
    this.buyukluk = 0;
  }
  private class HashDugumu {
    private Integer anahtar;
    private String deger;
    private HashDugumu sonraki;
    public HashDugumu(Integer anahtar, String deger) {
      this.anahtar = anahtar; this.deger = deger;
    }
  }
}
```

Sınıfın alanları ve parçaları şunlardır:

- `kovalar`, `HashDugumu` türünde bir dizidir. Her hücre bir zincirin ilk düğümünü gösterir; zincir boşsa hücre `null`'dır.
- `kovaSayisi` dizinin uzunluğunu, `buyukluk` saklanan çift sayısını tutar.
- Yapıcı, istenen kapasitede bir dizi ayırır. Java'da yeni bir nesne dizisinin bütün hücreleri `null` ile başlar; bu yüzden ayrıca bir döngüyle kovaları boşaltmak gerekmez. `new HashTablosu(10)` çağrısı 10 boş kova oluşturur ve `buyukluk = 0` olur.
- `HashDugumu`, sınıfın içinde tanımlanmış bir iç sınıftır. Her düğüm bir `anahtar`, bir `deger` ve bir sonraki düğümü gösteren `sonraki` referansı tutar. Bu, tek yönlü bağlı listenin düğümüdür; yalnız anahtar alanı eklenmiştir.

Düğümün değerin yanında anahtarı da tutması şarttır. Bir kovada birden çok çift olabileceği için, aranan anahtarın hangi düğümde olduğu ancak düğümdeki anahtara bakılarak anlaşılır. Yalnız değer saklansaydı 5. kovadaki "Murat" ile "Ali"den hangisinin 105'e ait olduğu bilinemezdi.

Alan maliyeti: dizi N referans, her düğüm bir anahtar, bir değer ve bir `sonraki` referansı tutar. Toplam alan $O(N + n)$'dir.

12.3.4 Kova indisi ve negatif hash kodları

Bir anahtarın hangi kovaya gideceğini `getKovaIndisi` metodu hesaplar:

```
// Basit yol: anahtar % kovalar.length → negatif olabilir!
private int getKovaIndisi(Integer anahtar) {
  int h = anahtar.hashCode();  // = anahtar
  return Math.floorMod(h, kovalar.length);
}
```

Metot önce anahtarın hash kodunu alır. `Integer` sınıfında `hashCode()` sayının kendisini döndürür; 41'in hash kodu 41'dir. Ardından kod, `Math.floorMod` ile dizinin uzunluğuna sıkıştırılır. İlk satırdaki yorum önemli bir uyarı içerir: "basit yol" olan `anahtar % kovalar.length` negatif bir sonuç verebilir.

Java'da `%` işleminin sonucu **bölünenin işaretini** alır. *41 % 10 = 1*'dir, ama *−7 % 10 = −7*'dir. Negatif bir sayı dizi indisi olarak kullanılırsa Java `ArrayIndexOutOfBoundsException` fırlatır. `Math.floorMod` ise bölen pozitif olduğu sürece her zaman 0 ile bölen − 1 arasında bir sonuç verir.

`anahtar`	`anahtar % 10`	`Math.floorMod(anahtar, 10)`	**Sonuç**
105	5	5	kova 5
21	1	1	kova 1
41	1	1	kova 1 (21 ile çakışır)
−7	−7	3	`%` ile `kovalar[-7]` → `ArrayIndexOutOfBoundsException`

Tablo 12.7: 10 kovalı tabloda `%` ile `Math.floorMod` karşılaştırması.

Anahtarlar her zaman pozitif tam sayılar olsa bile bu ayrıntı önemlidir. `String` gibi anahtarların `hashCode()` değeri sık sık negatif çıkar; örneğin "Sercan"ın hash kodu −1821977712'dir. Tablo her tür anahtarla çalışacaksa negatif kodlara hazırlıklı olmalıdır.

`Math.floorMod` içeride nasıl çalışır? Bölen *n > 0* iken davranışı aşağıdaki kısa metotla aynıdır:

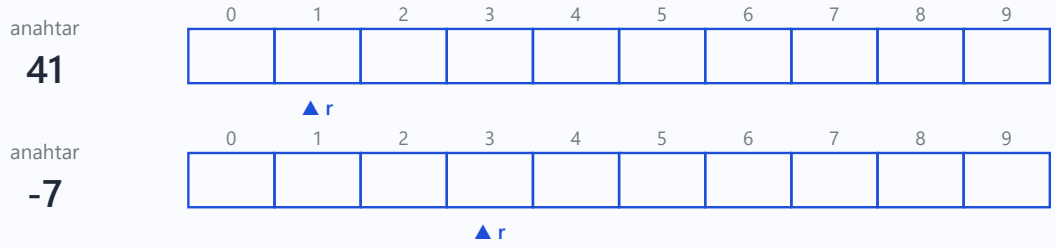
```
private int getKovaIndisi(Integer anahtar) {
  int h = anahtar.hashCode();  // = anahtar
  return Math.floorMod(h, kovalar.length);
}

// floorMod'un n > 0 için eşdeğeri:
static int floorMod(int x, int n) {
  int r = x % n;       // işaret x'ten gelir
  if (r < 0) r += n;   // negatifse n ekle
  return r;
}
```

Metot önce Java'nın `%` işlemini yapar. Sonuç negatifse *n* ekler. *x % n* sonucu her zaman *−(n − 1)* ile *n − 1* arasında olduğundan, negatif bir sonuca *n* eklemek onu 1 ile *n − 1* arasına taşır. Pozitif sonuca dokunulmaz. Tablo 12.8 iki çağrının izlemesini verir.

Adım	`anahtar = 41`	`anahtar = -7`
`h = anahtar.hashCode()`	41	−7
`r = h % 10`	1	−7 (işaret bölünenden gelir)
`r < 0` ?	hayır, düzeltme yok	evet: `r += 10` → 3
Dönüş	1 (21 ile aynı kova: çakışma)	3

Tablo 12.8: `getKovaIndisi(41)` ve `getKovaIndisi(-7)` izlemesi (10 kova).



Şekil 12.12: `getKovaIndisi` sonuçları. Üstte: 41 için `r = 1`; anahtar 1. kovaya gider. Altta: −7 için `%` −7 verir, `r += 10` ile `r = 3` olur; anahtar 3. kovaya gider.

Negatif sorunu için akla gelen bir başka çözüm, mutlak değer almaktır: `Math.abs(h) % N`. Bu çözüm hemen her zaman çalışır, ama bir istisnası vardır. `int` türü −2147483648 ile 2147483647 arasındaki sayıları tutar. En küçük değer olan `Integer.MIN_VALUE = -2147483648`'in pozitif karşılığı `int`'e sığmaz. Java'da `Math.abs(Integer.MIN_VALUE)` taşar ve yine −2147483648 döndürür.

Örnek 12.3

13 kovalı bir tabloya "polygenelubricants" dizgisi anahtar olarak ekleniyor. Dizginin hash kodu −2147483648'dir. Üç farklı indis hesabını karşılaştıralım.

1. `h % 13`: −2147483648 % 13 = −11. Negatif indis; `ArrayIndexOutOfBoundsException`.
2. `Math.abs(h) % 13`: `Math.abs(h)` yine −2147483648 olur; sonuç yine −11. Mutlak değer burada hiçbir işe yaramaz.
3. `Math.floorMod(h, 13)`: −11 negatif olduğu için 13 eklenir; sonuç 2'dir. Anahtar 2. kovaya güvenle yerleşir.
4. **Sonuç:** Tek bir anahtar için bile programı çökerten bir hata, ancak bu nadir değere denk gelen girdiyle ortaya çıkar. Bu yüzden indis hesabında her zaman `Math.floorMod(h, N)` kullanılır.

12.3.5 Kod: yerlestir()

Tabloya anahtar-değer çifti ekleyen metot üç adımda çalışır: kovayı bul, zinciri tara, gerekirse başa ekle.

```
public void yerlestir(Integer anahtar,
                      String deger) {
  int kovaIndisi = getKovaIndisi(anahtar);
  HashDugumu ilk = kovalar[kovaIndisi];
  while (ilk != null) {    // anahtar var mı?
    if (ilk.anahtar.equals(anahtar)) {
      ilk.deger = deger;   // varsa güncelle
      return;
    }
    ilk = ilk.sonraki;
  }
  buyukluk++;
  HashDugumu dugum =
      new HashDugumu(anahtar, deger);
  dugum.sonraki = kovalar[kovaIndisi];
  kovalar[kovaIndisi] = dugum;   // başa ekle
}
```

Kodun satırları şöyle çalışır:

- `kovaIndisi = getKovaIndisi(anahtar)` anahtarın kovasını *O(1)*'de bulur. Bundan sonra yalnız bu kovayla uğraşılır; diğer kovalara hiç bakılmaz.
- `ilk = kovalar[kovaIndisi]`, zincirin ilk düğümünü gösteren bir yardımcı referanstır. Adı "ilk" olsa da döngü içinde zincir boyunca ilerler.
- `while (ilk != null)` döngüsü zinciri baştan sona tarar. Her düğümde `ilk.anahtar.equals(anahtar)` sorulur. Anahtar bulunursa yalnız değer değişir (`ilk.deger = deger`) ve metot döner. Bu bir güncellemedir; `buyukluk` artmaz, çünkü çift sayısı değişmemiştir.
- Döngü `ilk == null` ile biterse anahtar tabloda yoktur. `buyukluk` bir artar, yeni bir düğüm oluşturulur.
- `dugum.sonraki = kovalar[kovaIndisi]` yeni düğümü eski zincirin başına bağlar. `kovalar[kovaIndisi] = dugum` ise kovayı yeni düğüme yöneltir. Bu iki satırın sırası önemlidir.

Karşılaştırmanın `==` ile değil `equals` ile yapıldığına dikkat edilmelidir. Anahtarlar `Integer` nesneleridir. `==` iki referansın aynı nesneyi gösterip göstermediğini sorar. Java küçük sayılar (−128 … 127) için hazır nesneleri yeniden kullandığından `==` bu aralıkta doğru sonuç verir gibi görünür. 128 ve üstünde ise aynı değerli iki `Integer` genellikle farklı nesnedir ve `==` `false` döndürür. `equals` ise değerleri karşılaştırır.

Maliyet. Kova bulma *O(1)*'dir. Zincir tarama, zincirin uzunluğu kadar sürer. Başa ekleme *O(1)*'dir. Toplam *O(1 + zincir uzunluğu)* olur. Zincirlerin ortalama uzunluğu küçük tutulduğu sürece bu ortalamada sabittir (12.3.10).

Sık yapılan hata

Başa eklemede iki satırın sırasını değiştirmek: önce `kovalar[kovaIndisi] = dugum`, sonra `dugum.sonraki = kovalar[kovaIndisi]`. İlk satırdan sonra kova zaten yeni düğümü gösterir; ikinci satır düğümü kendisine bağlar. Eski zincirdeki bütün düğümler kaybolur ve tablo sessizce veri yitirir. Kural: önce yeni düğüm eski zincire bağlanır, sonra kova güncellenir.

12.3.6 Kod izleme: yerlestir()

10 kovalı bir tabloda yalnız (105, "Murat") çifti olsun; `buyukluk = 1`. Sırasıyla `yerlestir(21, "Leyla")`, `yerlestir(41, "Sena")` ve `yerlestir(21, "Ali")` çağrılıyor. Tablo 12.9 her çağrının adımlarını verir.

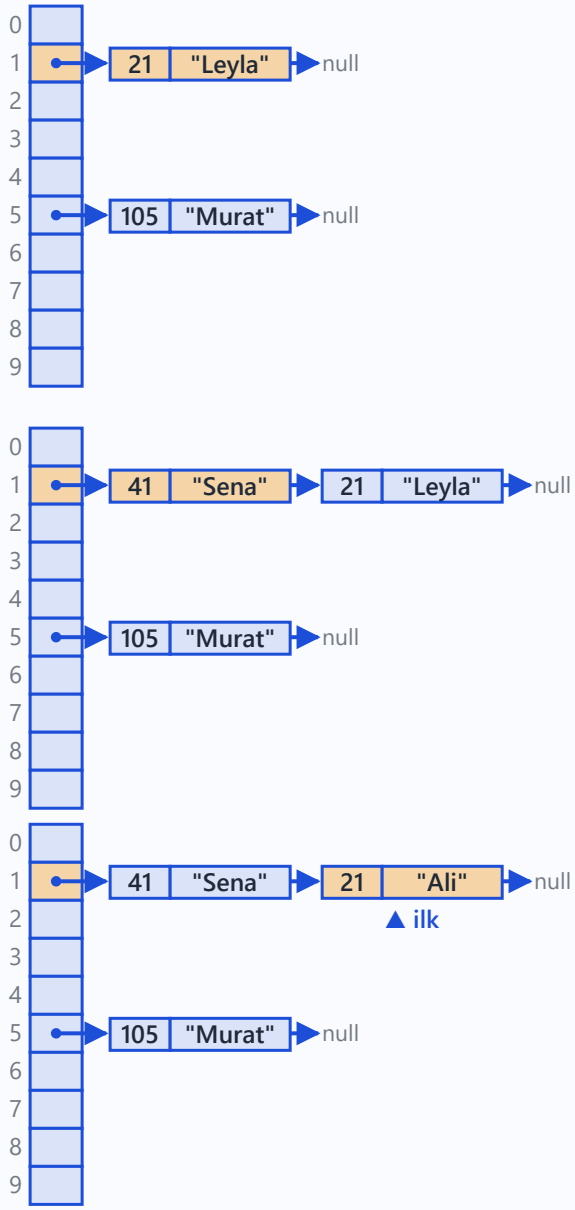
Çağrı	Kova	Zincir taraması	Sonuç	Kova 1 (sonra)	`buyukluk`
`yerlestir(21, "Leyla")`	*21 mod 10 = 1*	kova boş, `ilk = null`; döngüye girilmez	yeni düğüm başa	21 → `null`	2
`yerlestir(41, "Sena")`	*41 mod 10 = 1*	21 ≠ 41; `ilk = null`	yeni düğüm başa	41 → 21 → `null`	3
`yerlestir(21, "Ali")`	*21 mod 10 = 1*	41 ≠ 21; 21 = 21	değer güncellenir: "Leyla" → "Ali"	41 → 21 → `null`	3

Tablo 12.9: Üç `yerlestir` çağrısının izlemesi. “Kova 1” sütunu zinciri baştan sona gösterir.

İlk çağrıda 1. kova boştur. `ilk = null` olduğu için döngüye hiç girilmez; anahtarın tabloda olmadığı hemen anlaşılır. `buyukluk` 2 olur. Yeni düğüm (21, "Leyla") oluşturulur. `dugum.sonraki = kovalar[1]` satırı `null` atar, çünkü kova boştu. Son satır kovayı yeni düğüme yöneltir.

İkinci çağrıda 41 de 1. kovaya düşer: bu bir çakışmadır. Döngü 21 düğümünü inceler; *21 ≠ 41* olduğu için `ilk = ilk.sonraki` ile ilerlenir ve `ilk` `null` olur. Anahtar tabloda yoktur. `buyukluk` 3 olur. Yeni düğüm (41, "Sena") oluşturulur, `sonraki` alanı eski baş olan 21'i gösterir, kova da 41'i gösterir. Zincir artık 41 → 21'dir. Eski zincir kaybolmamış, yeni düğümün arkasına geçmiştir.

Üçüncü çağrı bir güncellemedir. Döngü önce 41'i görür; eşit değildir, ilerlenir. Sonra 21'i görür: `21.equals(21)` doğrudur. Değer "Leyla"dan "Ali"ye döner ve metot `return` ile biter. Yeni düğüm eklenmez, `buyukluk` 3'te kalır. Bu çağrı iki karşılaştırma yapmıştır.



Şekil 12.13: `yerlestir` izlemesinin üç anı. (a) `yerlestir(21, "Leyla")` bitti: 1. kova → 21. (b) `yerlestir(41, "Sena")` bitti: yeni düğüm zincirin başında, 1. kova → 41 → 21. (c) `yerlestir(21, "Ali")` : 21 bulundu, değeri "Ali" oldu; düğüm eklenmedi.

12.3.7 Kod ve izleme: getir()

Bir anahtarın değerini döndüren metot, `yerlestir` 'in tarama kısmının aynısıdır:

```
public String getir(Integer anahtar) {
  int kovaIndisi = getKovaIndisi(anahtar);
  HashDugumu ilk = kovalar[kovaIndisi];
  while (ilk != null) {
    if (ilk.anahtar.equals(anahtar))
      return ilk.deger;        // bulundu
    ilk = ilk.sonraki;
  }
  return null;                 // anahtar yok
}
```

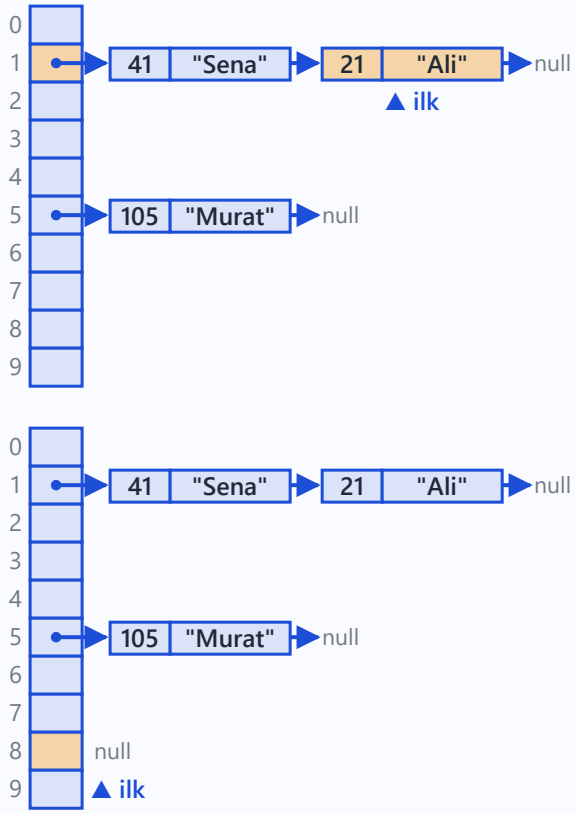
Metot anahtarın kovasını bulur ve yalnız o kovanın zincirini tarar. Diğer kovalara hiç bakılmaz. Anahtar bulunursa değeri döner. Zincir biterse anahtar tabloda yoktur ve metot `null` döndürür. Maliyeti *O(1 + zincir uzunluğu)*'dur. Zincirlerin ortalama uzunluğu *n / N*'dir.

Önceki izlemenin sonundaki tabloda (5. kova → 105; 1. kova → 41 → 21) üç arama yapalım:

Çağrı	Kova	İncelenen düğümler	Karşılaştırma	Dönüş
`getir(105)`	*105 % 10 = 5*	105	1	"Murat"
`getir(21)`	*21 % 10 = 1*	41, 21	2	"Ali"
`getir(88)`	*88 % 10 = 8*	— (kova boş)	0	`null`

Tablo 12.10: Üç `getir` çağrısı.

`getir(105)` zincirin ilk düğümünde eşleşmeyi bulur; bir karşılaştırma yeterlidir. `getir(21)` önce 41'i görür. 41 aynı kovaya düşmüş başka bir anahtardır, yani çakışmanın sonucudur. Zincirde ilerlenir, ikinci düğümde 21 bulunur. `getir(88)` boş bir kovaya bakar; döngüye hiç girilmez ve `null` döner. Başarısız bir arama boş kovada tek adımda biter. Kova dolu olsaydı bütün zincir taranmak zorunda kalırdı, çünkü anahtarın zincirde olmadığı ancak sona gelince anlaşılır.



Şekil 12.14: `getir` izlemesi. (a) `getir(21)` : `ilk` ikinci düğümde, 21 = 21; "Ali" döner (2 karşılaştırma). (b) `getir(88)` : 8. kova boş, `ilk = null` ; `null` döner.

12.3.8 Kod: sil()

Zincirden bir düğümü çıkarmak, tek yönlü bağlı listeden düğüm silmekle aynıdır. Bir düğümü listeden koparmak için ondan *önceki* düğümün `sonraki` alanını değiştirmek gerekir. Bu yüzden tarama sırasında bir önceki düğüm de izlenir.

```
public String sil(Integer anahtar) {
  int kovaIndisi = getKovaIndisi(anahtar);
  HashDugumu ilk = kovalar[kovaIndisi];
  HashDugumu onceki = null;
  while (ilk != null &&
         !ilk.anahtar.equals(anahtar)) {
    onceki = ilk;
    ilk = ilk.sonraki;
  }
  if (ilk == null) return null;   // yok
  buyukluk--;
  if (onceki != null)       // ortadan/sondan
    onceki.sonraki = ilk.sonraki;
  else                      // baştan
    kovalar[kovaIndisi] = ilk.sonraki;
  return ilk.deger;
}
```

Kod satır satır şöyle çalışır:

- `onceki = null` ile başlanır: zincirin ilk düğümünden önce bir düğüm yoktur.
- `while` döngüsünün koşulu iki parçalıdır: zincir bitmemişse *ve* düğümün anahtarı aranan değilse ilerle. Java `&&` işlecinde ilk koşul yanlışsa ikinciye bakmaz. Bu yüzden `ilk == null` iken `ilk.anahtar`'a erişilmez ve `NullPointerException` oluşmaz.
- Döngüde her adımda `onceki = ilk` ve `ilk = ilk.sonraki` yapılır; iki referans birlikte ilerler.
- Döngü bitince iki durum vardır. `ilk == null` ise anahtar zincirde yoktur ve metot `null` döndürür. Değilse `ilk` silinecek düğümü gösterir.
- Silme iki biçimde yapılır. Düğüm zincirin ortasında ya da sonundaysa (`onceki != null`) önceki düğüm silinen düğümü atlar: `onceki.sonraki = ilk.sonraki`. Düğüm zincirin başındaysa (`onceki == null`) kova güncellenir: `kovalar[kovaIndisi] = ilk.sonraki`.
- Son olarak `buyukluk` bir azaltılmış, silinen değer döndürülür. Kopan düğümü artık hiçbir referans göstermediği için Java'nın çöp toplayıcısı onu bellekten temizler.

Maliyet, `getir`'le aynıdır: *O(1 + zincir uzunluğu)*. Düğümü koparmak birkaç referans atamasıdır.

> **Sık yapılan hata**
>
> Baştaki düğümü silerken yalnız `onceki.sonraki = ilk.sonraki` satırını yazmak. Silinecek düğüm zincirin başındaysa `onceki` `null`'dır ve bu satır `NullPointerException` verir. Baştan silme ayrı bir durumdur: kovanın kendisi güncellenmelidir. Kodda bu iki durum `if` / `else` ile ayrılmıştır.

12.3.9 Kod izleme: sil()

Tablo yine 5. kova → (105, "Murat") ve 1. kova → (41, "Sena") → (21, "Ali") biçiminde, `buyukluk = 3` olsun. Sırasıyla `sil(21)`, `sil(105)` ve `sil(88)` çağrılıyor.

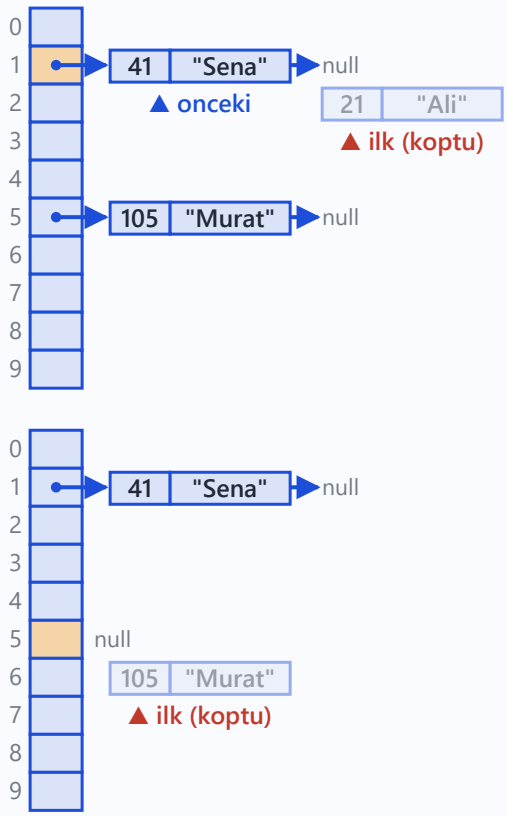
Çağrı	Döngü adımları	Döngü sonunda	Dal	Tablo (sonra)	`buyukluk`	Dönüş
`sil(21)`	41 ≠ 21 → `onceki = 41`, `ilk = 21`; 21 = 21, döngü biter	`onceki` = 41, `ilk` = 21	`onceki.sonraki = ilk.sonraki` (`null`)	1 → 41; 5 → 105	2	"Ali"
`sil(105)`	105 = 105, döngüye girilmez	`onceki = null`, `ilk` = 105	`kovalar[5] = ilk.sonraki` (`null`)	1 → 41	1	"Murat"
`sil(88)`	8. kova boş, `ilk == null`	`ilk = null`	`return null`	değişmez	1	`null`

Tablo 12.11: Üç `sil` çağrısının izlemesi.

`sil(21)` zincirin ortasından silmenin örneğidir. Döngü 41'de eşleşme bulamaz ve iki referansı ilerletir: `onceki` 41'i, `ilk` 21'i gösterir. 21 aranan anahtar olduğu için döngü koşulu yanlış olur ve döngü biter. `ilk != null` olduğundan silme yapılır. `onceki` `null` olmadığı için düğüm ortada ya da sondadır: `onceki.sonraki = ilk.sonraki` ile 41, 21'i atlar ve doğrudan `null`'ı gösterir. 21 zincirden kopar.

`sil(105)` baştan silmenin örneğidir. Döngünün koşulu ilk düğümde yanlış olur; döngü gövdesi hiç çalışmaz ve `onceki` `null` kalır. Bu yüzden `else` dalına girilir ve kovanın kendisi güncellenir: `kovalar[5] = ilk.sonraki`. 105'in ardında düğüm olmadığı için kova boşalır.

`sil(88)` başarısız bir silmedir. Kova boş olduğu için döngü hemen biter, `ilk == null` kontrolü `null` döndürür. Tablo ve `buyukluk` değişmez.



Şekil 12.15: `sil` izlemesi. (a) `sil(21)` : `onceki.sonraki = ilk.sonraki` ; 41 artık 21'i atlar, 21 zincirden koptu. (b) `sil(105)` : düğüm zincirin başındaydı, `kovalar[5] = ilk.sonraki` ile kova boşaldı.

12.3.10 Zincirlemenin maliyeti

Zincirlemeli tabloda bütün işlemler *O(1 + zincir uzunluğu)* sürer. Peki bir zincir ne kadar uzun olur? Bunu tablonun doluluk oranı belirler.

Tanım 12.5

n çift saklayan *N* kovalı bir hash tablosunun **yük faktörü** (*load factor*) $\lambda = n / N$'dir. Zincirlemede λ, zincirlerin ortalama uzunluğudur.

Ortalama uzunluk her zincirin bu uzunlukta olduğu anlamına gelmez. Bir kova boş, bir başkası kalabalık olabilir. Analizde genellikle şu varsayım yapılır: hash fonksiyonu her anahtarı kovalardan birine, öteki anahtarlardan bağımsız olarak, eşit olasılıkla gönderir. Bu varsayıma **basit düzgün hash** (*simple uniform hashing*) denir.

Teorem 12.2 (Zincirlemede beklenen arama maliyeti)

Basit düzgün hash varsayımı altında, yük faktörü λ olan zincirlemeli bir tabloda başarısız bir arama ortalama λ düğüm inceler. Başarılı bir arama ortalama yaklaşık $1 + \lambda/2$ düğüm inceler. Kova hesabıyla birlikte iki aramanın beklenen maliyeti $O(1 + \lambda)$'dır.

İspat. Başarısız aramada aranan anahtar eşit olasılıkla N kovadan birine gider ve o kovanın bütün zinciri taranır. Zincir uzunluklarının toplamı n olduğundan rastgele bir kovanın ortalama uzunluğu $n / N = \lambda$'dır. Başarılı aramada ise aranan anahtar, kendi zincirinde kendisinden sonra eklenmiş anahtarların arkasındadır (yeni düğümler başa eklenir). Bir anahtardan sonra eklenen anahtarların her biri, aynı kovaya $1/N$ olasılıkla düşer. Ortalamada bunların sayısı yaklaşık $(n - 1)/2$'dir. Aranan anahtarın önünde ortalama $(n - 1)/(2N) \approx \lambda/2$ düğüm vardır; kendisiyle birlikte $1 + \lambda/2$ düğüm incelenir. □

Örnek 12.4

Şekil 12.10'daki altı çiftli tabloda ($N = 10$, $n = 6$) beklenen değerlerle gerçek değerleri karşılaştıralım.

1. **Yük faktörü:** $\lambda = 6 / 10 = 0{,}6$.
2. **Başarısız arama:** Rastgele bir kova seçilip zincirinin tamamı taranır. Zincir uzunlukları 0. kovadan 9. kovaya 1, 1, 0, 0, 0, 2, 1, 0, 0, 1'dir. Ortalama $6 / 10 = 0{,}6$ düğüm; bu tam olarak λ'dır.
3. **Başarılı arama:** Tablodaki altı anahtarın her biri bir kez aranırsa 10, 1, 26, 99 ve 105 birer karşılaştırmada, 5 ise iki karşılaştırmada bulunur. Toplam 7, ortalama $7 / 6 \approx 1{,}17$'dir. Teoremin verdiği değer $1 + 0{,}6/2 = 1{,}3$'tür. Bu küçük tablo varsayımın öngördüğünden de dengelidir.

Teoremin asıl mesajı şudur: λ bir sabitin altında tutulursa, örneğin hiçbir zaman 0,75'i aşmazsa, işlemlerin beklenen maliyeti $O(1)$'dir. Bunun için tablo, çift sayısı arttıkça büyütülür. Bu işleme yeniden boyutlandırma denir ve 12.5.2'de ele alınır. Zincirlemede λ 1'i aşabilir; bu yalnız zincirleri uzatır ve işlemleri yavaşlatır, tabloyu kullanılamaz yapmaz.

En kötü durum ise başkadır. Bütün anahtarlar aynı kovaya düşerse tek bir zincir n uzunluğunda olur ve arama $O(n)$ sürer. Bu, kötü bir hash fonksiyonunda ya da özellikle aynı kovaya düşecek biçimde seçilmiş anahtarlarda görülür. Basit düzgün hash varsayımı yalnız ortalamayı güvence altına alır.

12.4 Açık adresleme

Ayrı zincirleme her çakışmada yeni bir düğüm oluşturur. Düğümler belleğin farklı yerlerine dağılır ve her düğüm fazladan bir referans taşır. Açık adresleme bu ek yapıları kullanmaz. Bütün çiftler dizinin kendisinde saklanır ve her yuvada en çok bir çift bulunur. Çakışan bir anahtar için dizide başka bir

boş yuva aranır.

12.4.1 Yoklama dizisi

Açık adreslemede bir anahtar önce kendi **ev yuvasına**, yani *h(k)* numaralı yuvaya bakar. Yuva boşsa anahtar oraya yerleşir. Doluysa belirli bir kurala göre ikinci bir yuva, o da doluysa üçüncü bir yuva denenir. Bir yuvanın dolu olup olmadığına bakma işine **yoklama** (*probing*) denir. Denenen yuvaların sırası da anahtarın **yoklama dizisidir** (*probe sequence*).

Tanım 12.6

Açık adresleme (*open addressing*), bütün çiftleri N yuvalı dizinin içinde saklayan çakışma çözüm yöntemidir. k anahtarı için denenen i. yuva

> $h(k, i) = (h(k) + f(i)) \bmod N,\ i = 0, 1, 2, \ldots$

formülüyle bulunur. $f(0) = 0$ olduğundan ilk deneme her zaman ev yuvasıdır. f fonksiyonunun seçimi, yoklama yöntemini belirler.

Yuva başına en çok bir çift saklandığından çift sayısı yuva sayısını aşamaz: $n \le N$, yani $\lambda \le 1$'dir. Zincirlemenin aksine burada λ'nın 1'i aşması mümkün değildir. Üç yaygın yoklama yöntemi vardır:

Yöntem	*f(i)*	Yoklama dizisi	Zayıf yönü
Doğrusal (*linear probing*)	i	5, 6, 7, 8, 9, …	Birincil kümelenme
Karesel (*quadratic probing*)	i^2	5, 6, 9, 14 → 1, 21 → 8, …	İkincil kümelenme; her yuvayı denemeyebilir
Çift hash (*double hashing*)	$i \cdot h_2(k)$	5, 5 + d, 5 + 2d, … ($d = h_2(k)$)	$h_2(k) \neq 0$ olmalı; adım ile N aralarında asal olmalı

Tablo 12.12: Üç yoklama yöntemi. Örnek yoklama dizileri $h(k) = 5$, $N = 13$ için verilmiştir.

Açık adreslemenin zincirlemeye göre üstünlükleri şunlardır. Ek düğüm ve referans yoktur; bellek yalnız dizi kadardır. Dizi bellekte ardışık durduğu için işlemci önbelleği onu verimli kullanır; ardışık yuvaları yoklamak, dağınık düğümler arasında gezinmekten pratikte hızlıdır. Bedelleri de vardır. Yük faktörü düşük tutulmalıdır, çünkü tablo doldukça yoklama dizileri hızla uzar. Silme de özel bir işlem ister (12.4.5).

Bu kesimin bütün örnekleri aynı veriyi kullanır: $N = 13$ yuvalı tabloya, $h(k) = k \bmod 13$ ile sırasıyla 18, 41, 22, 44, 59, 32, 31 ve 73 anahtarları eklenir. Ev yuvaları sırasıyla 5, 2, 9, 5, 7, 6, 5 ve 8'dir. Aynı veri üç yöntemle eklendiğinde aradaki farklar doğrudan görülür.

12.4.2 Doğrusal yoklama

Doğrusal yoklama en basit yöntemdir: ev yuvası doluysa bir sonrakine, o da doluysa ondan sonrakine bakılır. Dizinin sonuna gelinince başa dönülür. Kod şöyledir:

```
// yerlestir(k): doğrusal yoklama, N = 13
int h = k % 13;                // ev yuvası
for (int i = 0; i < 13; i++) {
  int j = (h + i) % 13;          // h, h+1, h+2…
  if (tablo[j] == null) {      // boş mu?
    tablo[j] = k;
    break;
  }                            // dolu: i++
}
```

`h = k % 13` ev yuvasıdır. Döngü değişkeni `i` deneme sayısını tutar. Her turda `j = (h + i) % 13` denenecek yuvadır: `i = 0` için ev yuvası, sonra bir sonraki yuva, sonra bir sonraki. `% 13` sayesinde 12. yuvadan sonra 0. yuvaya dönülür. Boş yuva bulununca anahtar yazılır ve `break` ile döngüden çıkılır. Döngü en çok 13 kez döner, çünkü 13 farklı `i` değeri bütün yuvaları bir kez dener.

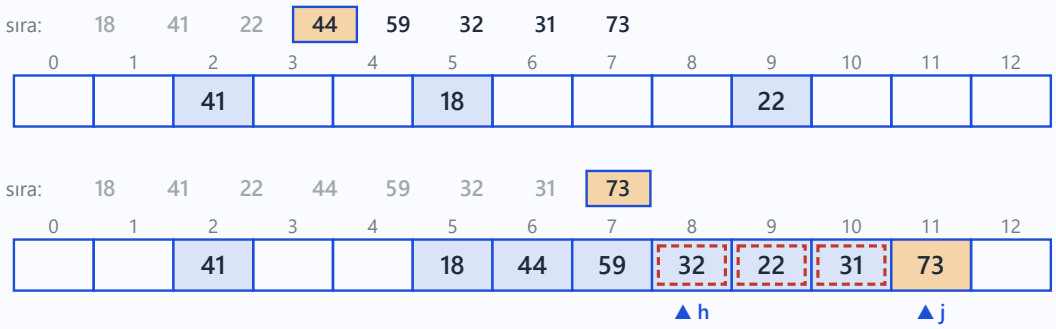
İlk üç anahtar boş tabloya geldiği için ilk denemede yerleşir: 18 → 5, 41 → 2, 22 → 9. Sonraki anahtarların izlemesi Tablo 12.13'te verilmiştir.

k	Ev yuvası *h*	Denenen yuvalar	Yerleştiği yuva	Yoklama
18	5	5	5	1
41	2	2	2	1
22	9	9	9	1
44	5	5 (18), 6	6	2
59	7	7	7	1
32	6	6 (44), 7 (59), 8	8	3
31	5	5 (18), 6 (44), 7 (59), 8 (32), 9 (22), 10	10	6
73	8	8 (32), 9 (22), 10 (31), 11	11	4
Toplam				19

Tablo 12.13: Doğrusal yoklama izlemesi (*N = 13*).

44 için ev yuvası 5'tir, ama orada 18 vardır. Bir sonraki yuva 6 boştur; 44 iki yoklamada yerleşir. 59 ev yuvası 7'de yer bulur. 32'nin ev yuvası 6'dır. Ama 6'ya biraz önce 44 yerleşmiştir. 7'de 59 vardır; 32 ancak 8'de yer bulur. 32, ev yuvası 5 olmadığı hâlde 5'ten başlayan kalabalığın içine düşmüştür.

31'in durumu daha da kötüdür. Ev yuvası 5'tir ve 5'ten 9'a kadar bütün yuvalar doludur. 31 altı yoklamadan sonra 10. yuvaya yerleşir. 73'ün ev yuvası 8'dir; 8, 9 ve 10 dolu olduğundan 11'e kadar gider. Sonunda 5'ten 11'e kadar yedi yuva tek bir dolu blok oluşturur.



Şekil 12.16: Doğrusal yoklama. Üstte: 18 → 5, 41 → 2, 22 → 9 yerleşmiş; sırada 44 var. Altta: 73 eklendi. Ev yuvası 8'dir (▲h); 8, 9 ve 10 dolu olduğu için (kesikli çerçeveler) 11'e (▲j) yerleşir. 5–11 arası tek bir büyük küme oluşturur.

Tanım 12.7

Birincil kümelenme (*primary clustering*), doğrusal yoklamada dolu yuvaların uzun ve kesintisiz bloklar oluşturması ve bu blokların büyüdükçe daha da hızlı büyümesidir.

Kümelenme kendini besler. Bir kümenin herhangi bir yuvasına düşen anahtar, kümenin sonuna kadar yürür ve kümeyi bir yuva uzatır. Küme uzadıkça ona düşen anahtar sayısı da artar. Yukarıdaki örnekte 32 ve 73'ün ev yuvaları 5 değildir, ama ikisi de 5'te başlayan kümeye takılmıştır. Ayrıca iki küme arasındaki tek boş yuva dolduğunda iki küme birleşir ve daha da büyük bir küme oluşur.

Doğrusal yoklamanın maliyeti kümelerin uzunluğuna bağlıdır. Tablo seyrekken neredeyse her anahtar ilk denemede yerleşir. Tablo doldukça kümeler uzar ve yoklama sayısı hızla artar. Bu artışın sayısal karşılığı 12.5.1'de verilmiştir.

12.4.3 Karesel yoklama

Birincil kümelenmenin nedeni, yoklamanın komşu yuvalara bakmasıdır. Karesel yoklama, sıçramaları büyüterek kümeden çabuk çıkmaya çalışır: ev yuvasından 1, 4, 9, 16, … yuva ötesi denenir.

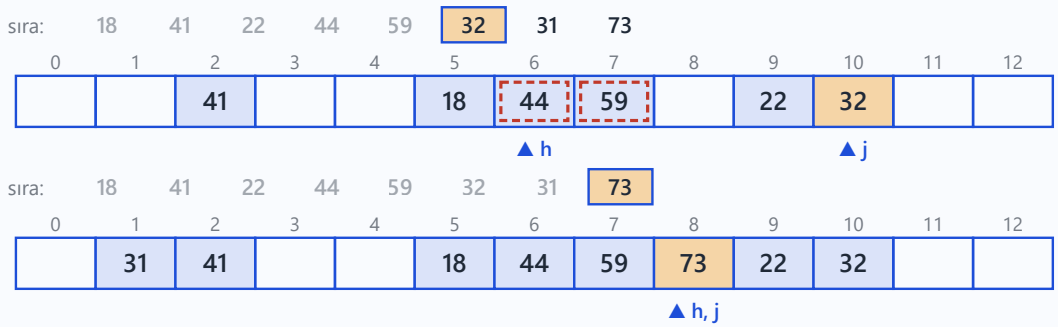
```
// yerlestir(k): karesel yoklama, N = 13
int h = k % 13;                // ev yuvası
for (int i = 0; i < 13; i++) {
  int j = (h + i * i) % 13;      // h, h+1, h+4…
  if (tablo[j] == null) {      // boş mu?
    tablo[j] = k;
    break;
  }                            // dolu: i++
}
```

Kodun doğrusal yoklamadan tek farkı `j` 'nin hesabıdır: `(h + i * i) % 13`. `i = 0, 1, 2, 3` için ev yuvasına sırasıyla 0, 1, 4 ve 9 eklenir. Aynı anahtarları bu kez karesel yoklamayla ekleyelim.

k	*h*	Denenen yuvalar	Yerleştiği yuva	Yoklama
18	5	5	5	1
41	2	2	2	1
22	9	9	9	1
44	5	5 (18), 6	6	2
59	7	7	7	1
32	6	6 (44), 7 (59), *6 + 4 = 10*	10	3
31	5	5 (18), 6 (44), *5 + 4 = 9* (22), *5 + 9 = 14 → 1*	1	4
73	8	8	8	1
Toplam				14

Tablo 12.14: Karesel yoklama izlemesi (*N = 13*).

Fark 32 ile başlar. Doğrusal yoklamada 32, 8. yuvaya yerleşip kümeyi büyütmüştü. Karesel yoklamada üçüncü deneme $6 + 2^2 = 10$'dur ve 32 kümenin dışına, 10. yuvaya yerleşir. 31 dördüncü denemede $5 + 3^2 = 14$, yani $14 \bmod 13 = 1$. yuvaya sıçrar. 8. yuva boş kaldığı için 73 de ev yuvasına ilk denemede yerleşir. Toplam yoklama 19'dan 14'e, en uzun yoklama 6'dan 4'e inmiştir.



Şekil 12.17: Karesel yoklama. Üstte: 32 eklendi; 6 ve 7 dolu (kesikli), üçüncü deneme *6 + 4 = 10* boş. Altta: bütün anahtarlar eklendi; 31 1. yuvaya sıçradı, 73 ev yuvası 8'e ilk denemede yerleşti.

Karesel yoklama birincil kümelenmeyi azaltır, ama başka bir sorun bırakır. Aynı ev yuvasına düşen anahtarlar hep aynı yoklama dizisini izler. Örnekte 18, 44 ve 31'in ev yuvası 5'tir; üçü de 5, 6, 9, 1, … sırasını dener. Ev yuvası aynı olan anahtarlar birbirinin yolunu tıkar. Bu duruma **ikincil kümelenme** (*secondary clustering*) denir. Etkisi birincil kümelenmeden hafiftir, çünkü yalnız aynı ev yuvasına düşen anahtarları etkiler.

Karesel yoklamanın ikinci zayıflığı, her yuvayı denemeyebilmesidir. *N = 13* ve *h = 5* için *i = 0, 1, …, 12* değerleri sırasıyla 5, 6, 9, 1, 8, 4, 2, 2, 4, 8, 1, 9, 6 yuvalarını verir. On üç denemede yalnız yedi farklı yuva görülür: {1, 2, 4, 5, 6, 8, 9}. Bu yuvaların hepsi doluysa tabloda altı boş yuva olsa bile anahtar yer

bulamaz. Bu sorunu önlemek için şu sonuç kullanılır.

Teorem 12.3 (Karesel yoklamada boş yuva güvencesi)

N 2'den büyük bir asal sayıysa karesel yoklamanın ilk *(N + 1)/2* denemesi birbirinden farklı yuvalara bakar. Bu yüzden tablo yarıdan az doluysa ($\lambda < 1/2$) her ekleme boş bir yuva bulur.

İspat. *0 ≤ i < j ≤ (N − 1)/2* olan iki deneme aynı yuvaya baksın: $h + i^2 \equiv h + j^2 \pmod N$. O zaman *N*, $j^2 - i^2 = (j - i)(j + i)$'yi böler. *N* asal olduğu için çarpanlardan birini bölmelidir. Ama *1 ≤ j − i < N* ve *1 ≤ j + i ≤ N − 1*'dir; ikisi de *N*'den küçük pozitif sayılardır ve *N*'ye bölünemez. Çelişki. Demek ki ilk *(N + 1)/2* deneme farklı yuvalardır. Tabloda $\lambda < 1/2$ iken dolu yuva sayısı en çok *(N − 1)/2*'dir. Farklı *(N + 1)/2* yuvanın hepsi dolu olamaz; en az biri boştur. □

N = 13 için teorem ilk yedi denemenin (*i = 0 … 6*) farklı olduğunu söyler. Yukarıda bulunan yedi farklı yuva tam olarak bunlardır; yedinci denemeden sonra yuvalar tekrarlanmaya başlar. Bu yüzden karesel yoklama kullanan tablolarda *N* asal seçilir ve λ 0,5'in altında tutulur.

12.4.4 Çift hash

İkincil kümelenmenin nedeni, yoklama adımının yalnız ev yuvasına bağlı olmasıdır. Çift hash, adımı anahtarın kendisine bağlar. Bunun için ikinci bir hash fonksiyonu h_2 kullanılır ve *i*. deneme

> $h(k, i) = (h(k) + i \cdot h_2(k)) \bmod 13,\ h_2(k) = 7 - (k \bmod 7)$

formülüyle bulunur. $k \bmod 7$ 0 ile 6 arasında olduğundan $h_2(k)$ 1 ile 7 arasındadır; asla 0 olmaz. Adım 0 olsaydı anahtar hep ev yuvasını yoklar ve hiç ilerleyemezdi. Kod şöyledir:

```
// yerlestir(k): çift hash, N = 13
int h = k % 13;
int d = 7 - k % 7;             // adım: 1…7
for (int i = 0; i < 13; i++) {
  int j = (h + i * d) % 13;
  if (tablo[j] == null) {      // boş mu?
    tablo[j] = k;
    break;
  }                            // dolu: i++
}
```

`d = 7 - k % 7` adımdır. Java'da `%` çıkarmadan önce yapıldığı için ifade *7 − (k mod 7)* anlamına gelir. Döngünün geri kalanı öteki yöntemlerle aynıdır; yalnız `j = (h + i * d) % 13` hesabı değişir.

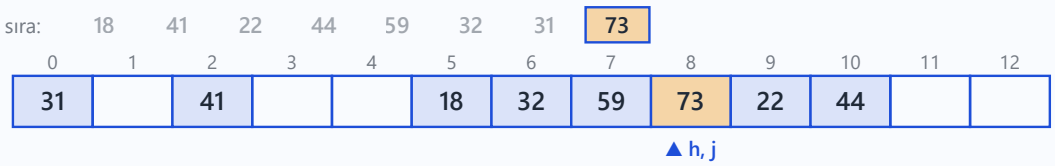
k	*h(k)*	*d* = $h_2(k)$	Denenen yuvalar	Yerleştiği yuva	Yoklama
18	5	7 − 4 = 3	5	5	1

k	$h(k)$	$d = h_2(k)$	Denenen yuvalar	Yerleştiği yuva	Yoklama
41	2	$7 - 6 = 1$	2	2	1
22	9	$7 - 1 = 6$	9	9	1
44	5	$7 - 2 = 5$	5 (18), $5 + 5 = 10$	10	2
59	7	$7 - 3 = 4$	7	7	1
32	6	$7 - 4 = 3$	6	6	1
31	5	$7 - 3 = 4$	5 (18), $5 + 4 = 9$ (22), $5 + 8 = 13 \to 0$	0	3
73	8	$7 - 3 = 4$	8	8	1
Toplam					11

Tablo 12.15: Çift hash izlemesi ($N = 13$, $h_2(k) = 7 - (k \bmod 7)$).

Ev yuvası 5 olan üç anahtara bakalım. 18 ilk denemede yerleşir. 44'ün adımı 5'tir ve 10. yuvaya sıçrar. 31'in adımı 4'tür: önce 5'e, sonra 9'a, sonra $5 + 8 = 13$, yani 0. yuvaya gider. Aynı ev yuvasından başlayan anahtarlar farklı adımlarla ilerler ve birbirinin yolunu tıkamaz. 44 artık 6. yuvayı işgal etmediği için 32 de ev yuvasına ilk denemede yerleşir. Toplam yoklama 11'e iner.

Çift hash ikincil kümelenmeyi tamamen ortadan kaldırmaz. İki anahtar hem aynı ev yuvasını hem aynı adımı alıyorsa yine aynı diziyi izler. Örnekte 59, 31 ve 73'ün adımı 4'tür; ama ev yuvaları farklı olduğu için birbirlerini etkilemezler. İki fonksiyonun birlikte çakışma olasılığı, tek fonksiyonunkinden çok daha küçüktür.



Şekil 12.18: Çift hash sonrası tablo: 31 → 0, 41 → 2, 18 → 5, 32 → 6, 59 → 7, 73 → 8, 22 → 9, 44 → 10. Son eklenen 73 ev yuvasına (▲h, j) ilk denemede yerleşti.

Adımın neden asal bir N ile birlikte kullanıldığını şu sonuç açıklar.

Teorem 12.4 (Çift hash yoklama dizisi)

Adım d ile N aralarında asalsa $h, h + d, h + 2d, \ldots, h + (N - 1)d \pmod N$ denemeleri N yuvanın hepsini birer kez ziyaret eder. N asalsa 1 ile $N - 1$ arasındaki her adım bu koşulu sağlar.

İspat. $0 \le i < j \le N - 1$ için $h + i \cdot d \equiv h + j \cdot d \pmod N$ olsun. O zaman N, $(j - i) \cdot d$'yi böler. d ile N'nin 1'den başka ortak böleni olmadığından N, $j - i$'yi bölmelidir. Ama $0 < j - i < N$'dir. Çelişki. N deneme birbirinden farklıdır ve N yuvanın hepsini kapsar. □

Örnekte *N = 13* asaldır ve adımlar 1 ile 7 arasındadır; her anahtar gerekirse bütün yuvaları dener. *N* asal olmasaydı, örneğin *N = 12* ve adım 4 olsaydı, yoklama yalnız *h, h + 4, h + 8* yuvalarını gezer ve tabloda boş yer olsa bile onu bulamayabilirdi.

12.4.5 Silme: “silindi” işareti

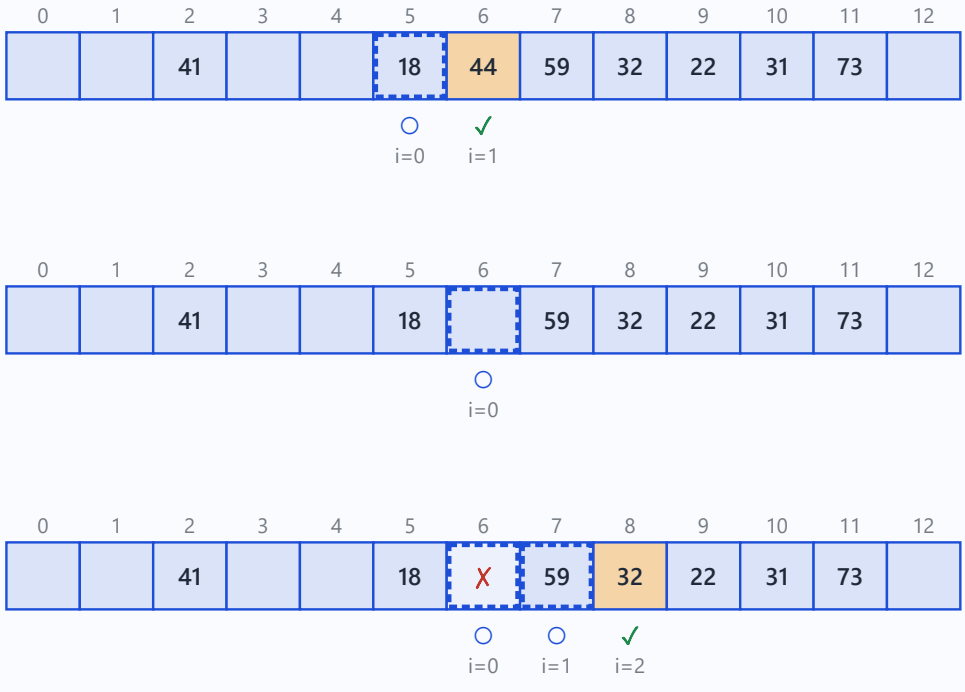
Açık adreslemede silme, ilk bakışta göründüğü kadar basit değildir. Doğrusal yoklama sonrası tabloyu ele alalım: 2'de 41, 5'ten 11'e kadar sırasıyla 18, 44, 59, 32, 22, 31, 73 vardır. 44 silinmek isteniyor.

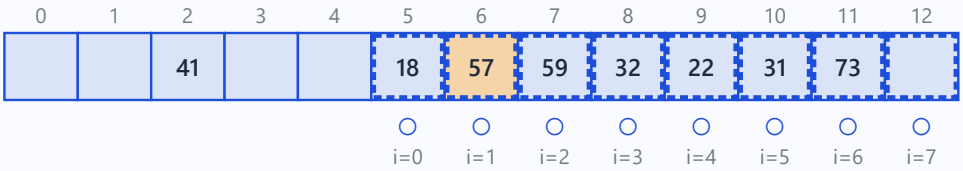
Önce 44 bulunur: *h(44) = 5*, 5. yuvada 18 vardır, eşit değildir; 6. yuvada 44 bulunur. Akla gelen ilk çözüm, 6. yuvayı `null` yapmaktır. Ama bu tabloyu bozar. 32'yi arayalım: *h(32) = 6*, yani arama 6. yuvadan başlar. Yuva boştur. Açık adreslemede boş yuva, “aranan anahtar buradan ötede olamaz” demektir. Çünkü 32 eklenirken bu yuva boş olsaydı 32 buraya yerleşirdi. Arama “32 yok” cevabını verir. Oysa 32 tabloda, 8. yuvadadır.

Sorun, silinen yuvanın başka anahtarların yoklama dizisinin bir halkası olmasıdır. 32, 6 ve 7 dolu olduğu için 8'e yerleşmişti. 6'yı boşaltmak, 32'ye giden yolu koparır. Doğru çözüm yuvayı boşaltmak yerine onu özel bir işaretle işaretlemektir.

Tanım 12.8

Silindi işareti (*DEFUNCT* ya da *tombstone*, “mezar taşı”), açık adreslemede silinen bir çiftin yuvasına konan özel değerdir. Arama bu yuvayı *dolu gibi* atlar ve aramaya devam eder. Ekleme ise bu yuvayı *boş gibi* yeniden kullanabilir. Şekillerde ✗ ile gösterilir.





Şekil 12.19: Açık adreslemede silme (✗: silindi işareti; hücre altındaki ○: yoklanan yuva, ✓: bulundu). (a) `sil(44)` : *h(44) = 5*, 18 ≠ 44; 6. yuvada 44 bulundu. (b) Yanlış yol: 6. yuva boşaltıldı; `getir(32)` 6'dan başlar, boş yuvayı görür ve "yok" der, oysa 32 8. yuvadadır. (c) Doğru yol: 6. yuvaya silindi işareti (✗) kondu; `getir(32)` ✗'i atlar, 7'de 59 ≠ 32, 8'de 32 bulunur. (d) `yerlestir(57)` : *h = 5*; 5, 6 (✗), 7, ..., 11 dolu, 12 boş; 57 tabloda yok. 57, ilk uygun yuva olan 6'ya yerleşir.

Şeklin son parçası işaretin ekleme sırasındaki rolünü gösterir. 57 eklenirken arama 5. yuvadan başlar. 6. yuvadaki ✗, ilk uygun yuva olarak not edilir. Ama arama orada durmaz, çünkü 57 tabloda daha ileride bir yerde zaten olabilir. Arama gerçekten boş bir yuvaya (12) varınca 57'nin tabloda olmadığı kesinleşir. Ancak o zaman 57, not edilen ilk uygun yuvaya, yani 6'ya yazılır. Böylece silinen yuvanın yeri boşa gitmez.

Silindi işaretlerinin bir bedeli vardır. İşaretler tabloda birikir ve aramalar onları dolu yuva gibi geçmek zorundadır. Çok sayıda silme yapılan bir tabloda yoklama dizileri, gerçek öğe sayısı az olsa bile uzar. Bu yüzden işaretli yuvaların sayısı da izlenir ve gerektiğinde tablo baştan oluşturulur. Yeniden oluşturma bütün işaretleri temizler.

Sık yapılan hata

Açık adreslemede silinen yuvaya `null` atamak. Silme işleminin kendisi doğru çalışır gibi görünür, ama sonraki bazı aramalar tabloda olan anahtarları "yok" diye bildirir. Hata ancak silinen yuvanın ötesine yerleşmiş bir anahtar arandığında ortaya çıkar; bu yüzden küçük testlerde gözden kaçabilir.

12.4.6 Kod: yuvaBul()

Açık adreslemede arama ve ekleme aynı yoklamayı yapar. Bu ortak adım, `yuvaBul` adlı tek bir yardımcı metotta toplanabilir. Metot hem anahtarın bulunup bulunmadığını hem de bulunamadıysa nereye eklenebileceğini tek bir tam sayıyla bildirir.

```
private int yuvaBul(int k) {       // findSlot
  int h = Math.floorMod(k, N), uygun = -1;
  for (int i = 0; i < N; i++) {
    int j = (h + i) % N;           // doğrusal
    if (tablo[j] == null ||
        tablo[j] == SILINDI) {
      if (uygun == -1) uygun = j;
      if (tablo[j] == null) break;
    } else if (tablo[j].anahtar == k)
      return j;                    // bulundu
  }
  return -(uygun + 1);         // yok: yer kodu
}
```

Metodun parçaları şunlardır:

- `h` ev yuvasıdır; `Math.floorMod` negatif anahtarları da güvenle yuvaya çevirir. `uygun`, şimdiye kadar görülen ilk boş ya da silinmiş yuvayı tutar. Başta henüz böyle bir yuva görülmediği için −1'dir.
- Döngü doğrusal yoklama yapar: `j = (h + i) % N`.
- Yuva boşsa (`null`) ya da silinmişse (`SILINDI`) ve henüz uygun yuva yoksa, bu yuva `uygun` olarak not edilir. Yuva *gerçekten boşsa* döngü `break` ile biter: anahtar bundan öteye yerleşmiş olamaz. Silinmiş yuvada ise döngü devam eder.
- Yuva doluysa ve anahtarı aranan anahtarsa yuvanın indisi döndürülür. Dönüş değeri 0 ya da pozitiftir ve "bulundu" anlamına gelir.
- Döngü anahtarı bulamadan biterse `-(uygun + 1)` döndürülür. Bu değer her zaman negatiftir ve "bulunamadı" anlamına gelir. İçinde eklenecek yuvanın bilgisi de vardır.

Neden yalnız `-uygun` değil de `-(uygun + 1)`? `uygun = 0` olabilir. O zaman `-uygun` da 0 olurdu ve "0. yuvada bulundu" sonucundan ayırt edilemezdi. 1 eklemek bütün "bulunamadı" sonuçlarını kesin olarak negatif yapar: 0. yuva −1, 6. yuva −7 olarak kodlanır. Yuvayı geri almak için *−(j + 1)* hesaplanır. Java kütüphanesindeki `Arrays.binarySearch` da bulunamayan öğenin ekleneceği yeri aynı yöntemle bildirir.

`yuvaBul` üzerine arama ve ekleme birer satırda kurulur:

- `getir(k)` : `j = yuvaBul(k)`. `j ≥ 0` ise `tablo[j].deger`, değilse `null` döndürülür.
- `yerlestir(k, v)` : `j ≥ 0` ise o yuvadaki değer güncellenir. Değilse yeni çift `tablo[-(j + 1)]` yuvasına yazılır.

Karesel yoklama ya da çift hash istenirse yalnız `j`'nin hesabı değişir: karesel için `(h + i*i) % N`, çift hash için `(h + i*h2) % N`. Metodun geri kalanı aynı kalır.

Kodda örtük bir varsayım vardır: tablo hiçbir zaman tamamen dolmaz. $\lambda < 1$ tutulur ve gerekince tablo büyütülür. Bu yüzden döngü her zaman en az bir boş ya da silinmiş yuva görür ve `uygun` −1'de kalmaz. Tablo tamamen dolu olsaydı `uygun = -1` kalır ve dönüş değeri `-(-1 + 1) = 0` olurdu; bu da yanlışlıkla "0. yuvada bulundu" diye okunurdu.

Maliyet. Her yoklama *O(1)*'dir; toplam süre yoklama sayısıyla orantılıdır. En kötü durumda bütün yuvalar denenir: *O(N)*. Yük faktörü düşük tutulursa beklenen yoklama sayısı küçük bir sabittir.

12.4.7 Kod izleme: yuvaBul()

Doğrusal yoklama sonrası tabloda 44 silinmiş, 6. yuvada ✗ olsun. Tablo: 2 → 41, 5 → 18, 6 → ✗, 7 → 59, 8 → 32, 9 → 22, 10 → 31, 11 → 73; geri kalan yuvalar boş. Önce `yuvaBul(32)`, sonra `yuvaBul(57)` çağrılıyor.

`i`	`j`	Yuva	Karar	`uygun`
—	—	—	başlangıç	−1
0	6	✗ (silindi)	ilk uygun yuva; silindi ≠ boş, devam	6
1	7	59	59 ≠ 32, devam	6
2	8	32	32 = 32, `return 8`	6

Tablo 12.16: `yuvaBul(32)` izlemesi (*h = 32 mod 13 = 6*).

Arama ✗ işaretini gördüğünde durmaz. 6'yı yalnız "gerekirse buraya eklenir" diye not eder ve devam eder. 8. yuvada 32 bulunur, metot 8 döndürür. `getir` bu durumda `tablo[8].deger`'i döndürür. ✗ işareti arama zincirini koparmamıştır.

`i`	`j`	Yuva	Karar	`uygun`
—	—	—	başlangıç	−1
0	5	18	18 ≠ 57, devam	−1
1	6	✗	ilk uygun yuva; devam	6
2	7	59	59 ≠ 57, devam	6
3	8	32	32 ≠ 57, devam	6
4	9	22	22 ≠ 57, devam	6
5	10	31	31 ≠ 57, devam	6
6	11	73	73 ≠ 57, devam	6
7	12	boş	`uygun` zaten 6, değişmez; gerçekten boş: `break`	6
Dönüş: *−(6 + 1) = −7*				—

Tablo 12.17: `yuvaBul(57)` izlemesi (*h = 57 mod 13 = 5*).

`yuvaBul(57)` sekiz yoklama yapar. 6'daki ✗ ilk uygun yuva olarak not edilir. 12. yuva da boştur, ama `if (uygun == -1)` koşulu yanlış olduğu için `uygun` değişmez; ilk uygun yuva korunur. Gerçekten boş yuva görüldüğü için döngü biter: 57 bundan öteye yerleşmiş olamaz. Metot *−7* döndürür.

`yerlestir` bu sonucu şöyle kullanır: *j = −7 < 0*, öyleyse `tablo[-(j + 1)] = tablo[6] = 57`. Silinmiş yuva yeniden kullanılmıştır.



Şekil 12.20: `yuvaBul` izlemesinin sonuçları. Üstte: `yuvaBul(32)` ; ev yuvası 6'da *X* (▲h, uygun), 8'de 32 bulundu (▲j). Altta: `yuvaBul(57)` −7 döndürdü; `yerlestir` 57'yi silinmiş 6. yuvaya yazdı.

Ekleme 6'yı kullanmasaydı, yani 57'yi 12. yuvaya yazsaydı, tablo yine doğru çalışırdı. Ama 6'daki *X* tabloda kalır ve her aramada fazladan bir yoklamaya yol açardı. İlk uygun yuvayı kullanmak, silinmiş yuvaları zamanla temizler ve yoklama dizilerini kısa tutar.

12.4.8 Üç yoklama yönteminin karşılaştırması

Aynı sekiz anahtarın üç yöntemle eklenmesinden çıkan sayılar Tablo 12.18'de toplanmıştır.

Yöntem	Toplam yoklama	Anahtar başına ortalama	En uzun yoklama	Son tablo (yuva: anahtar)
Doğrusal	19	2,38	6 (31)	2: 41, 5: 18, 6: 44, 7: 59, 8: 32, 9: 22, 10: 31, 11: 73
Karesel	14	1,75	4 (31)	1: 31, 2: 41, 5: 18, 6: 44, 7: 59, 8: 73, 9: 22, 10: 32
Çift hash	11	1,38	3 (31)	0: 31, 2: 41, 5: 18, 6: 32, 7: 59, 8: 73, 9: 22, 10: 44

Tablo 12.18: 18, 41, 22, 44, 59, 32, 31, 73 anahtarlarının *N = 13* yuvalı tabloya eklenmesi ($\lambda = 8/13 \approx 0{,}62$).

Bu tek örnek genel eğilimi doğru yansıtır. Doğrusal yoklama kümelenmeye en açık yöntemdir. Karesel yoklama birincil kümelenmeyi kırar, ikincil kümelenmeyi bırakır. Çift hash iki tür kümelenmeyi de büyük ölçüde önler. Buna karşın pratikte doğrusal yoklama hâlâ çok kullanılır. Ardışık yuvaları yoklamak önbellek açısından en ucuz işlemdir ve yük faktörü düşük tutulursa kümeler kısa kalır. Çift hash ise her anahtar için ikinci bir hash hesabı ister ve yuvalar arasında uzun sıçramalar yapar.

Üç yöntemi farklı anahtarlarla denemek için sunumdaki deneme alanı ya da açık adresleme simülatörü kullanılabilir. Simülatör *N = 11* yuvalı bir tabloda *h(k) = k mod 11* ile çalışır; her ekleme ve aramada denenen yuvaları, silindi işaretlerini ve yük faktörünü gösterir.

12.5 Performans ve uygulamalar

Hash tablosunun ortalama *O(1)* vaadi bir koşula bağlıdır: tablo fazla dolmamalıdır. Bu kesim doluluğun maliyete etkisini sayılarla gösterir. Ardından tablo dolarken ne yapıldığını, Java `HashMap`'in bu işleri nasıl çözdüğünü ve hash tablolarının tipik kullanım yerlerini anlatır.

12.5.1 Yük faktörü

Yük faktörü $\lambda = n / N$, çift sayısının kova sayısına oranıdır. Burada n, koddaki `buyukluk`'tur; N ise kova ya da yuva sayısıdır. İki çakışma çözümünde λ'nın anlamı farklıdır:

- **Zincirlemede** λ, zincirlerin ortalama uzunluğudur. $\lambda > 1$ olabilir; tablo yine çalışır, ama işlemler yavaşlar. Beklenen maliyet $O(1 + \lambda)$'dır.
- **Açık adreslemede** $\lambda \le 1$ zorunludur, çünkü her yuvada en çok bir çift vardır. λ 1'e yaklaştıkça boş yuva bulmak zorlaşır ve yoklama sayısı hızla büyür.

Doğrusal yoklamada beklenen yoklama sayısını Donald Knuth hesaplamıştır. Tablo büyük ve hash fonksiyonu iyi olduğunda:

> başarılı arama: $\tfrac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$ başarısız arama: $\tfrac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$

Başarısız arama, boş bir yuvaya varana kadar sürer. Ekleme de aynı yolu izlediği için bir eklemenin maliyeti, başarısız bir aramanınkiyle aynıdır. Formüllerdeki $1 - \lambda$ terimi boş yuvaların oranıdır. Boş yuvalar azaldıkça payda küçülür ve değer büyür. Başarısız aramada payda karesi alındığı için büyüme çok daha hızlıdır.

λ	Başarılı	Başarısız (ve ekleme)	Zincirleme $1 + \lambda$
0,25	1,17	1,39	1,25
0,5	1,5	2,5	1,5
0,75	2,5	8,5	1,75
0,8	3	13	1,8
0,9	5,5	50,5	1,9

Tablo 12.19: Doğrusal yoklamada beklenen yoklama sayısı ve zincirlemedeki $1 + \lambda$ ile karşılaştırma.



Şekil 12.21: Doğrusal yoklamada beklenen yoklama sayısı. Başarısız arama (kırmızı) $\lambda = 0{,}5$'te 2,5, $\lambda = 0{,}75$'te 8,5, $\lambda = 0{,}9$'da 50,5'tir. Başarılı arama (mavi) 1,5, 2,5 ve 5,5'tir. Kesikli çizgi zincirlemenin $1 + \lambda$ maliyetidir ve hemen hemen düz kalır.

Tablo ve şekil iki sonucu açıkça gösterir. Birincisi, açık adreslemede $\lambda = 0{,}5$'e kadar maliyet düşüktür, ama sonrasında hızla artar. Tablonun %50'den %80'e dolması başarısız aramayı 2,5'ten 13 yoklamaya, yani yaklaşık beş katına çıkarır. %90'da ortalama 50 yoklama gerekir. İkincisi, zincirlemede maliyet λ ile yalnız doğrusal artar ve yüksek yük faktörlerinde bile kabul edilebilir kalır.

Bu yüzden pratikte iki farklı eşik kullanılır. Zincirlemede $\lambda \approx 0{,}75$ yaygın bir sınırdır; Java `HashMap` da bu değeri kullanır. Açık adreslemede genellikle $\lambda \le 0{,}5$ tutulur. Eşik aşılınca tablo yaklaşık iki katına büyütülür.

Örnek 12.5

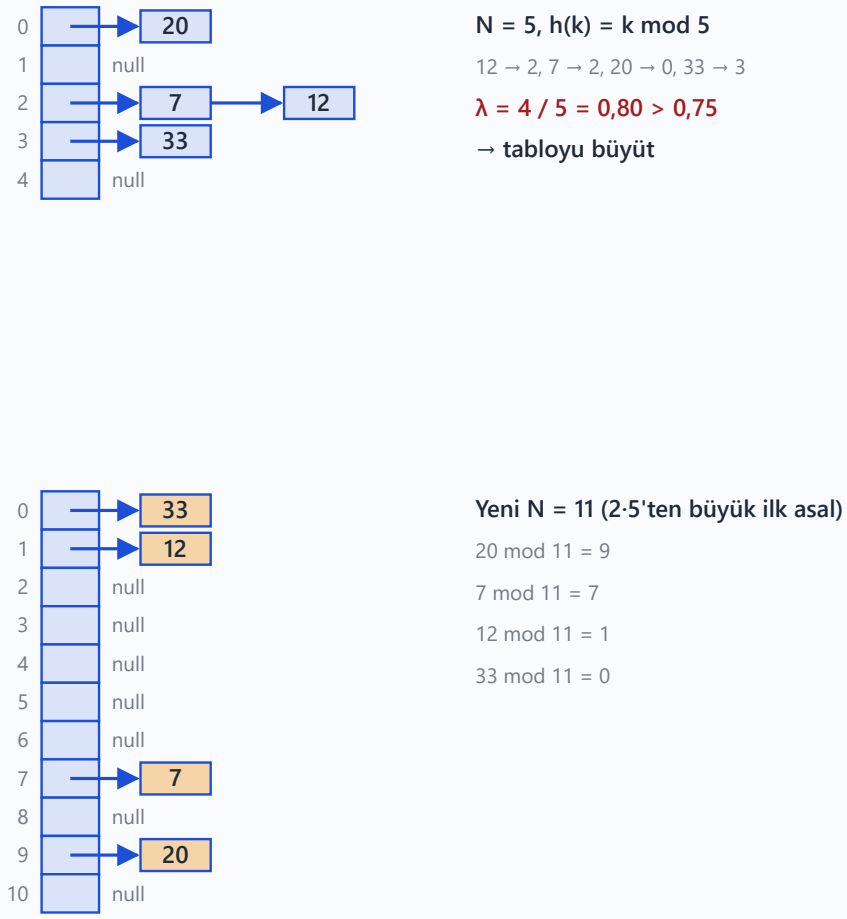
Doğrusal yoklamalı bir tabloda 600 anahtar saklanacaktır. Başarısız bir aramanın beklenen yoklama sayısının 3'ü geçmemesi isteniyor. Tablo en az kaç yuvalı olmalıdır?

1. **Koşul:** $\tfrac{1}{2}\,(1 + 1/(1-\lambda)^2) \le 3$, yani $1/(1-\lambda)^2 \le 5$.
2. **Çözüm:** $(1-\lambda)^2 \ge 1/5 = 0{,}2$, buradan $1 - \lambda \ge \sqrt{0{,}2} \approx 0{,}447$ ve $\lambda \le 0{,}553$ bulunur.
3. **Yuva sayısı:** $N \ge 600 \,/\, 0{,}553 \approx 1085{,}4$, yani $N \ge 1086$. 1086'dan büyük ilk asal sayı 1087'dir; 1087 yuvalı bir tablo yeterlidir. Bu durumda $\lambda = 600 \,/\, 1087 \approx 0{,}552$ ve beklenen başarısız arama yaklaşık 3 yoklamadır.

12.5.2 Yeniden boyutlandırma

Yük faktörü eşiği aşınca tablo büyütülür. Java'da dizinin boyu değiştirilemez; büyütme, daha büyük yeni bir dizi ayırmak ve bütün çiftleri oraya taşımak demektir. Bu işleme **yeniden boyutlandırma** (*resizing*) denir. Taşıma sırasında her anahtar yeniden hash'lenir (*rehashing*), çünkü indis *h(k) mod N* ile hesaplanır ve *N* değişmiştir.

Şekil 12.22 bir örneği adım adım gösterir. 5 kovalı zincirlemeli bir tabloya 12, 7, 20 ve 33 eklenmiştir. 12 ile 7 ikisi de 2. kovaya düşer ve bir zincir oluşturur. Dördüncü eklemeden sonra $\lambda = 4/5 = 0{,}80$ olur ve 0,75 eşiği aşılır. Tablo 11 kovaya büyütülür. Her anahtar *k mod 11* ile yeni kovasına gider: 20 → 9, 7 → 7, 12 → 1, 33 → 0. Büyütmeden sonra 7 ile 12 ayrı kovalardadır; tabloda hiç çakışma kalmamıştır ve $\lambda = 4/11 \approx 0{,}36$'dır.



Şekil 12.22: Yeniden boyutlandırma. Üstte: *N = 5*, *h(k) = k mod 5*; 12 → 2, 7 → 2, 20 → 0, 33 → 3. *λ = 4/5 = 0,80 > 0,75*: tablo büyütülür. Altta: yeni *N = 11* (*2 · 5*'ten büyük ilk asal); 20 mod 11 = 9, 7 mod 11 = 7, 12 mod 11 = 1, 33 mod 11 = 0.

Yeni kapasite neden tam iki kat (10) değil de 11'dir? Bölme yöntemi asal bir *N* ile daha iyi dağılım verir (12.2.2). Bu yüzden bazı gerçeklemeler eski boyun iki katından büyük ilk asal sayıyı seçer. Aşağıdaki kod daha basit bir kural kullanır: *2N + 1*. Bu sayı her zaman tektir ve 5 için 11, yani asaldır;

ama her zaman asal değildir (örneğin 7'den 15 elde edilir).

Bir yeniden boyutlandırma n çiftin hepsini taşıdığı için tek başına $O(n)$ sürer ve pahalıdır. Ama sık yapılmaz. Kapasite her seferinde yaklaşık iki katına çıktığından, bir büyütmeden sonra yeniden büyütme gerekene kadar en az o kadar yeni ekleme yapılır. Pahalı işlemin maliyeti bu eklemelere dağıtılınca ekleme başına sabit bir pay düşer.

Teorem 12.5 (Yeniden boyutlandırmanın amortize maliyeti)

Yük faktörü eşiği aşılınca kapasitesi en az iki katına çıkarılan bir hash tablosunda, boş tablodan başlayan n eklemenin yeniden boyutlandırmalar için yaptığı toplam taşıma sayısı $2n$'den azdır. Bu yüzden eklemenin **amortize** maliyeti $O(1)$'dir.

İspat. Son yeniden boyutlandırma anında tabloda $m \le n$ çift olsun; bu büyütme m taşıma yapar. Kapasite her büyütmede en az iki katına çıktığı için, bir önceki büyütme en çok $m/2$ çift varken, ondan önceki en çok $m/4$ çift varken yapılmıştır. Toplam taşıma $m + m/2 + m/4 + \ldots < 2m \le 2n$'dir. Her ekleme ayrıca kendi yerleştirmesini yapar. n ekleme toplam $O(n)$ iş yapar; ekleme başına ortalama sabittir. □

Teorem tek bir eklemenin hızlı olduğunu söylemez. Büyütmeyi tetikleyen ekleme $O(n)$ sürer. Ortalama hızın yettiği çoğu programda bu sorun değildir. Her işlemin kısa sürmesi gereken gerçek zamanlı sistemlerde ise büyütme parça parça, birkaç eklemeye yayılarak yapılabilir.

12.5.3 Kod izleme: yenidenBoyutlandir()

Zincirlemeli `HashTablosu` sınıfına eklenecek büyütme metodu şöyledir:

```
private void yenidenBoyutlandir() {
  HashDugumu[] eski = kovalar;
  int yeniN = 2 * eski.length + 1;
  kovalar = new HashDugumu[yeniN];
  buyukluk = 0;  kovaSayisi = yeniN;
  for (HashDugumu ilk : eski) {   // her kova
    while (ilk != null) {
      yerlestir(ilk.anahtar, ilk.deger);
      ilk = ilk.sonraki;
    }
  }
}
```

Metodun adımları şunlardır:

- `eski = kovalar`: eski dizi bir değişkende saklanır. Yeni dizi ayrıldıktan sonra eski zincirlere ancak bu değişken üzerinden ulaşılabilir.

- `yeniN = 2 * eski.length + 1`: yaklaşık iki kat ve tek bir kapasite hesaplanır.
- `kovalar = new HashDugumu[yeniN]`: alan artık boş yeni diziyi gösterir. `getKovaIndisi` `kovalar.length`'i kullandığı için bundan sonraki her indis hesabı yeni boya göre yapılır.
- `buyukluk = 0`: `yerlestir` her çifti eklerken `buyukluk`'u bir artırır. Sayaç sıfırlanmazsa çiftler iki kez sayılırdı.
- Dış döngü eski dizinin her kovasını, iç döngü o kovadaki zinciri gezer. Her düğüm için `yerlestir(ilk.anahtar, ilk.deger)` çağrılır; anahtar yeni tabloda yeni kovasına yerleşir.

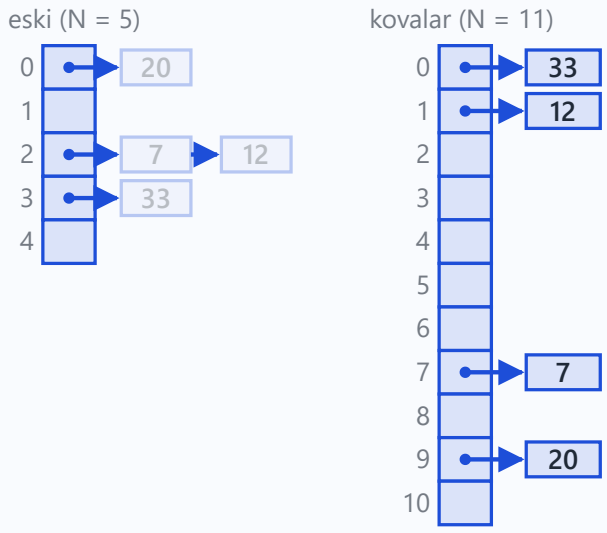
Metot eski düğümleri yeniden kullanmaz; `yerlestir` her çift için yeni bir düğüm oluşturur. Eski dizi ve düğümleri artık hiçbir referansla gösterilmediğinde çöp toplayıcı onları temizler. Daha verimli bir gerçekleme düğümleri yeni zincirlere doğrudan taşıyabilir; ama yukarıdaki kod daha kısa ve anlaşılırdır.

Önceki örnekteki tabloyu izleyelim. Eski tabloda 0. kova → 20, 2. kova → 7 → 12, 3. kova → 33 vardır (7, 12'den sonra eklendiği için zincirin başındadır).

Eski kova	`ilk`	**Eski indis**	**Yeni indis**	`buyukluk`
(hazırlık)	—	—	`yeniN = 11`, yeni dizi boş	0
0	20	*20 mod 5 = 0*	*20 mod 11 = 9*	1
1	`null`	—	— (`while`'a girilmez)	1
2	7	*7 mod 5 = 2*	*7 mod 11 = 7*	2
2	12	*12 mod 5 = 2*	*12 mod 11 = 1*	3
3	33	*33 mod 5 = 3*	*33 mod 11 = 0*	4
4	`null`	—	—	4

Tablo 12.20: `yenidenBoyutlandir()` izlemesi (5 kovadan 11 kovaya).

Eski 2. kovadaki zincir, iç döngünün iki tur dönmesine yol açar: önce 7, `ilk = ilk.sonraki` ile sonra 12 taşınır. Aynı eski kovada olan iki anahtar yeni tabloda 7. ve 1. kovalara ayrılmıştır. Sonunda dört çiftin hepsi taşınmış ve `buyukluk` yine 4 olmuştur. $\lambda = 4/11 \approx 0{,}36$'dır.



Şekil 12.23: `yenidenBoyutlandir()` bitti. Solda soluk renkte eski tablo ($N = 5$), sağda yeni tablo ($N = 11$): 0 → 33, 1 → 12, 7 → 7, 9 → 20.

Maliyet. Dış döngü N_{eski} kez döner, iç döngüler toplam n kez döner. Her `yerlestir` yeni tabloda ortalama $O(1)$'dir. Toplam süre $O(N + n)$'dir; eşik sayesinde N, n ile orantılı olduğundan bu $O(n)$ demektir. Taşıma sırasında eski ve yeni dizi bir süre birlikte bellekte durur.

Sık yapılan hata

Büyütmede eski diziyi yeni dizinin başına kopyalamak (`System.arraycopy` ya da `Arrays.copyOf` ile). Dinamik dizide doğru olan bu yöntem hash tablosunda yanlıştır. 12, eski tabloda 2. kovadadır ve kopyalama sonrası da 2. kovada kalır. Ama yeni tabloda `getir(12)` *12 mod 11 = 1*. kovaya bakar ve 12'yi bulamaz. Büyütmede her anahtar yeniden hash'lenmelidir.

12.5.4 Karmaşıklık karşılaştırması

Eşleme ADT'sinin farklı gerçeklemelerini bir arada karşılaştıralım.

Eşleme gerçeklemesi	`getir`	`yerlestir`	`sil`
Sırasız dizi / liste	O(n)	O(n)*	O(n)
Sıralı dizi (ikili arama)	O(log n)	O(n)	O(n)
Hash tablosu: ortalama	O(1)	O(1)	O(1)
Hash tablosu: en kötü	O(n)	O(n)	O(n)

Tablo 12.21: Eşleme gerçeklemelerinde işlem maliyetleri (n çift).

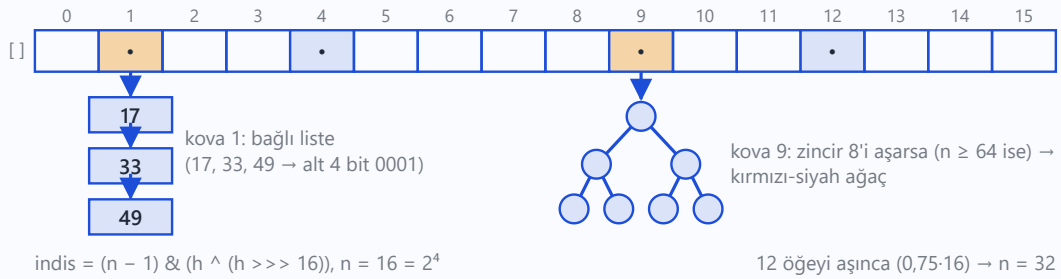
Yıldızlı hücrenin açıklaması şudur: sırasız listede yeni çifti eklemek tek başına *O(1)*'dir, ama eşlemede anahtarlar tekil olduğundan önce anahtarın zaten var olup olmadığı denetlenir. Bu denetim bütün listeyi tarar ve *O(n)* sürer. Sıralı dizide arama ikili aramayla *O(log n)*'dir, ama ekleme ve silme öğeleri kaydırmak zorunda olduğu için *O(n)*'dir.

Hash tablosu için üç noktanın altını çizmek gerekir:

- **En kötü durum.** Bütün anahtarlar aynı kovaya düşerse tek bir uzun zincir ya da tek bir büyük küme oluşur ve işlemler *O(n)* sürer. Ortalama *O(1)*, iyi bir hash fonksiyonuna ve sınırlı yük faktörüne bağlıdır.
- **Alan.** Toplam alan *O(n + N)*'dir: kova dizisi ve çiftler. Zincirlemede her düğüm bir referans fazlası taşır. Yük faktörü sınırlı tutulduğu için *N = O(n)*'dir ve alan *O(n)*'dir.
- **Sıra yok.** Hash tablosu anahtarları sırasız tutar. En küçük anahtarı bulmak, anahtarları sıralı listelemek ya da "50 ile 100 arasındaki anahtarlar" gibi aralık soruları bütün tabloyu gezmeyi gerektirir: *O(n)* ve sıralama gerekiyorsa daha fazlası. Sıralı erişim gereken işlerde arama ağaçları tercih edilir.

12.5.5 Java HashMap içeride nasıl çalışır?

Java'nın `java.util.HashMap` sınıfı, bu bölümde anlatılan fikirleri birkaç akıllı ayrıntıyla birleştirir.



Şekil 12.24: Java `HashMap`'in iç yapısı. 16 kovalı tabloda indis `(n - 1) & (h ^ (h >>> 16))` ile bulunur. 1. kovada bir bağlı liste var: 17, 33, 49 (alt 4 bitleri 0001). Bir zincir 8 düğümü aşarsa ve tablo en az 64 kovalıysa zincir kırmızı-siyah ağaca çevrilir. 12 öğe aşılınca (*0,75 · 16*) tablo 32 kovaya büyür.

- **Ayrı zincirleme ve 2'nin kuvveti kadar kova.** `HashMap` çakışmaları zincirlemeyle çözer. Kova sayısı *n* her zaman 2'nin bir kuvvetidir; varsayılan değer 16'dır. Bu sayede indis bölme yerine hızlı bir bit işlemiyle bulunur: `h & (n - 1)`. *n = 16 = 2⁴* iken *n − 1 = 15*'in ikili gösterimi `1111`'dir; `& 15` sayının yalnız son dört bitini bırakır. Bu, negatif olmayan sayılarda *h mod 16* ile aynı sonucu verir ve negatif kodlarda da 0 … 15 aralığında kalır.
- **Bit karıştırma.** Maske yalnız alt bitleri gördüğü için üst bitleri farklı, alt bitleri aynı olan anahtarlar hep aynı kovaya düşerdi. `HashMap` bu yüzden indisten önce `h ^ (h >>> 16)` hesaplar. `>>> 16`, sayının üst 16 bitini alt yarıya kaydırır; `^` (XOR) bu bitleri alt bitlerle karıştırır. Böylece üst bitler de indisi etkiler.

- ***λ = 0,75* eşiği.** Öğe sayısı *0,75 · n*'yi aşınca kova dizisi iki katına büyür ve düğümler yeni kovalara dağıtılır. 16 kovalı tabloda eşik 12'dir; 13. öğe eklenince tablo 32 kovaya çıkar. Kova sayısı 2'nin kuvveti olduğu için büyütmede her düğüm ya eski indisinde kalır ya da eski indis + eski boy konumuna gider; hangisine gideceğini tek bir bit belirler.
- **Ağaçlaşma (Java 8 ve sonrası).** Bir zincir 8 düğümü aşarsa ve tabloda en az 64 kova varsa zincir dengeli bir arama ağacına, bir kırmızı-siyah ağaca çevrilir. O kovadaki arama en kötü durumda *O(n)*'den *O(log n)*'e iner. Tablo 64 kovadan küçükse ağaca çevirmek yerine tablo büyütülür; küçük tabloda uzun zincirin daha olası nedeni kapasitenin azlığıdır.

Şekildeki 16 kovalı tablo bu ayrıntıları bir arada göstermek için çizilmiştir. Gerçek bir 16 kovalı `HashMap`'te 9. kovadaki uzun zincir ağaca çevrilmez; 64 kova sınırına ulaşılmadığı için tablo büyütülür.

`HashMap`'in gezinme sırası da sık sorulan bir konudur. `keySet()` ya da `for` döngüsüyle gezinme, kovaların sırasını izler; anahtarların ekleme sırasını değil. Ayrıca tablo büyüdüğünde düğümler kovalar arasında yer değiştirdiği için sıra da değişebilir. Ekleme sırası gerekiyorsa `LinkedHashMap`, anahtarların büyüklük sırası gerekiyorsa `TreeMap` kullanılır. Yalnız anahtarların varlığıyla ilgilenen işler için aynı hash tablosu yapısını kullanan `HashSet` vardır.

12.5.6 Kod izleme: indis(k, 16)

Bit karıştırmanın etkisini görmek için `HashMap`'in indis hesabını taklit eden kısa bir metot yazalım. `main` üç anahtarın 16 kovalı tablodaki indisini yazdırır.

```
static int indis(Object key, int n) { // n = 2^k
  int h = key.hashCode();
  int m = h ^ (h >>> 16); // üst bitleri karıştır
  return m & (n - 1); // alt k bit
}
// main:
for (int k : new int[]{1, 65537, 131073})
  System.out.println(k + " -> " + indis(k, 16));
```

`key.hashCode()`, `Integer` anahtarlar için sayının kendisidir. `h >>> 16` işlemi sayıyı 16 bit sağa kaydırır ve soldan sıfır doldurur; üst yarı alt yarıya iner. `h ^ (h >>> 16)`, alt 16 biti üst 16 bitle XOR'lar; üst 16 bit ise değişmez. Son olarak `m & (n - 1)` alt 4 biti alır.

Seçilen üç anahtar özel olarak hazırlanmıştır: 1, 65537 ve 131073. Onaltılık (*hexadecimal*) gösterimde bunlar `0x00000001`, `0x00010001` ve `0x00020001`'dir. Alt 16 bitleri aynıdır (`…0001`); yalnız üst bitleri farklıdır. Karıştırma olmasaydı `h & 15` üçü için de 1 verirdi.

`k` (`h`)	Onaltılık	Maskesiz `h & 15`	`h >>> 16`	`m = h ^ (h >>> 16)`	Alt 4 bit	`m & 15`
1	`0x00000001`	1	0	1	0001	1
65537	`0x00010001`	1	1	65536	0000	0

`k` (`h`)	Onaltılık	Maskesiz `h & 15`	`h >>> 16`	`m = h ^ (h >>> 16)`	Alt 4 bit	`m & 15`
131073	`0x00020001`	1	2	131075	0011	3

Tablo 12.22: `indis(k, 16)` izlemesi.

Anahtar 1'in üst yarısı sıfırdır; kaydırma 0 verir, XOR sayıyı değiştirmez ve indis 1 olur. 65537'nin üst yarısı 1'dir. Alt yarıdaki `0001` ile kaydırılmış `0001` XOR'lanınca alt bitler `0000` olur; *m = 65536*, indis 0'dır. 131073'ün üst yarısı 2'dir (`0010`). `0001 ^ 0010 = 0011` ; *m = 131075*, indis 3'tür. Program şunları yazdırır:

```
1 -> 1
65537 -> 0
131073 -> 3
```



Şekil 12.25: 131073 için karıştırma: `h` , `h>>>16` ve `m` 4 bitlik gruplar hâlinde. `m` 'nin alt 4 biti `0011` 'dir; anahtar 3. kovaya gider. Alttaki tabloda 65537 0., 1 1., 131073 3. kovadadır; kesikli çerçeve maskesiz indisi (1) gösterir.

Karıştırma olmasaydı üç anahtar da 1. kovada çakışırdı; karıştırma ile 1, 0 ve 3 numaralı kovalara dağılmışlardır. Bunun maliyeti yalnız bir kaydırma ve bir XOR işlemidir. Hash kodu her zaman küçük olan anahtarlarda (*h < 65536*) üst yarı sıfır olduğu için karıştırma sonucu değiştirmez; bu anahtarlarda alt bitler zaten anahtarın bütün bilgisini taşır.

12.5.7 Hash tablolarının kullanım alanları

"Bu anahtar var mı, varsa değeri ne?" sorusunun hızla cevaplanması gereken her yerde hash tablosu kullanılır. Birkaç tipik örnek:

- **Sözlük ve yazım denetimi.** Kelime → anlam eşlemesi bir hash tablosunda tutulur. Yazım denetleyicisi her kelime için "bu kelime sözlükte var mı?" diye sorar; sorunun cevabı ortalama sabit sürede gelir.
- **Veritabanları.** Hash indisleri, anahtara göre kayda hızlı erişim sağlar. İki tabloyu ortak bir sütuna göre birleştiren *hash join* algoritması, tablolardan birini bellekte bir hash tablosuna koyar ve ötekinin her satırı için eşini bu tabloda arar.

- **Bellek önbelleği.** Pahalı bir hesabın ya da sorgunun sonucu, bir URL ya da sorgu metni anahtarıyla saklanır. Aynı istek tekrar gelince sonuç hesaplanmadan tablodan alınır. Redis ve memcached gibi sistemler bu fikre dayanır.
- **Web tarayıcı geçmişi.** Ziyaret edilen adres → son ziyaret zamanı. Tarayıcı "bu bağlantı daha önce açıldı mı?" sorusuna göre bağlantının rengini değiştirir.
- **Çerez ve oturum yönetimi.** Çerez adı → değer; oturum kimliği → kullanıcı bilgisi. Sunucu her istekte oturum kimliğiyle kullanıcıyı bulur.
- **Derleyiciler.** Sembol tablosu, değişken adlarını tip, adres ve kapsam bilgisine eşler. Derleyici kodda bir ad gördüğünde onu bu tabloda arar.

Java programlarında bu işlerin çoğu `HashMap` ve `HashSet` ile yapılır. Bir dizideki öğelerin kaç kez geçtiğini saymak, iki listenin ortak öğelerini bulmak ya da tekrar eden öğeleri ayıklamak gibi küçük görevler de hash tablosuyla *O(n)* beklenen sürede çözülür. Bir sonraki uygulama bunun tipik bir örneğidir.

12.5.8 Uygulama: ikiToplam()

Bir tam sayı dizisinde, toplamı verilen bir hedefe eşit olan iki öğenin indislerini bulmak isteyelim. Kaba kuvvet çözümü bütün *(i, j)* çiftlerini dener: iç içe iki döngü, $O(n^2)$ zaman. Hash tablosu bu işi tek geçişte yapar.

```
static int[] ikiToplam(int[] a, int hedef) {
  Map<Integer, Integer> gor = new HashMap<>();
  for (int i = 0; i < a.length; i++) {
    int ek = hedef - a[i]; // gereken eş
    if (gor.containsKey(ek))
      return new int[]{gor.get(ek), i};
    gor.put(a[i], i); // değer -> indis
  }
  return null; // çift yok
}
```

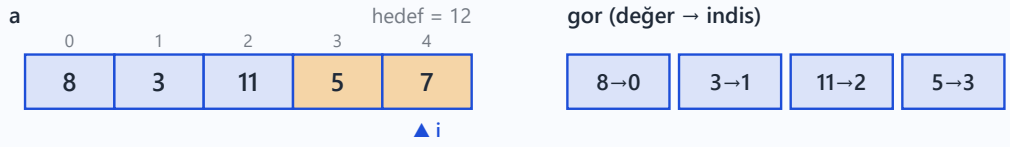
Fikir şudur: `a[i]` 'ye bakarken, onunla toplamı hedefi veren eş `ek = hedef - a[i]` 'dir. Bu eş daha önce görüldüyse çözüm bulunmuştur. Daha önce görülen değerler `gor` eşlemesinde, değer → indis biçiminde tutulur. `containsKey(ek)` sorusu ortalama *O(1)*'de cevaplanır. Eş yoksa `a[i]` eşlemeye eklenir; böylece sonraki öğeler onu eş olarak bulabilir.

`i`	`a[i]`	`ek`	`containsKey(ek)`	`gor` **(sonra)**
0	8	4	false	{8=0}
1	3	9	false	{8=0, 3=1}
2	11	1	false	{8=0, 3=1, 11=2}
3	5	7	false	{8=0, 3=1, 11=2, 5=3}

`i`	`a[i]`	`ek`	`containsKey(ek)`	`gor` **(sonra)**
4	7	5	**true**	— (`return [3, 4]`)

Tablo 12.23: `ikiToplam([8, 3, 11, 5, 7], 12)` izlemesi.

İlk dört adımda aranan eşlerin (4, 9, 1, 7) hiçbiri daha önce görülmemiştir; her değer indisiyle eşlemeye eklenir. Beşinci adımda `a[4] = 7` için gereken eş 5'tir ve 5, 3. indiste görülmüştür. Metot `[gor.get(5), 4] = [3, 4]` döndürür: *a[3] + a[4] = 5 + 7 = 12*.



Şekil 12.26: `ikiToplam` son adımı. Dizi [8, 3, 11, 5, 7], hedef 12; `i = 4`'te eş olan 5, `gor` eşlemesinde (5 → 3) bulunur. Cevap [3, 4].

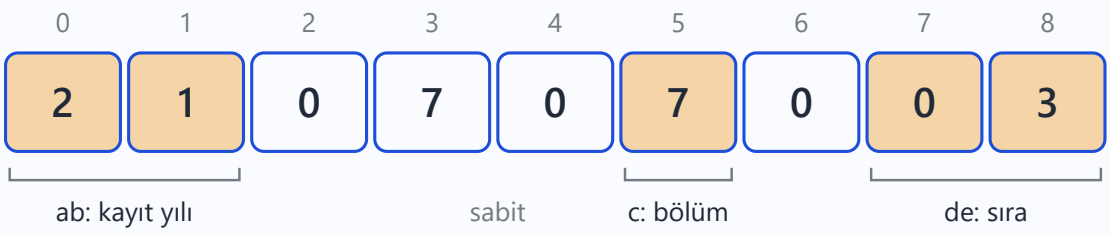
Kontrolün eklemeden *önce* yapılması önemlidir. Önce eklenseydi, hedefin yarısına eşit bir öğe (örneğin hedef 12 iken 6) kendisini eş olarak bulurdu. Ayrıca aynı değerin iki kopyası doğru eşleşir: ilk kopya eşlemeye girer, ikinci kopya onu bulur.

Maliyet. Dizi bir kez gezilir. Her adımda ortalama *O(1)* bir arama ve bir ekleme yapılır. Toplam beklenen süre *O(n)*, eşlemenin kullandığı ek bellek *O(n)*'dir. Kaba kuvvet ek bellek kullanmaz ama *O(n²)* sürer. Hash tablosu burada bellek harcayarak zaman kazanmıştır; bu ödünleşim hash tablosu kullanımının tipik biçimidir.

12.5.9 Uygulama: okul numarası için hash fonksiyonu

Genel amaçlı hash fonksiyonları her türlü anahtarla çalışmak zorundadır. Anahtarın yapısı biliniyorsa bu bilgiyle çok daha iyi bir fonksiyon tasarlanabilir. Bu uygulamada amaç, okul numaralarını 1000 öğelik bir diziye çakışmasız dağıtmaktır.

Numaralar `ab070c0de` biçimindedir; örneğin 210707003. Dokuz hanenin anlamı şöyledir:



Şekil 12.27: Okul numarası 210707003'ün haneleri. 0–1: `ab`, kayıt yılı (21); 2–4: sabit "070"; 5: `c`, bölüm (7); 6: sabit "0"; 7–8: `de`, kayıt sırası (03).

- `ab` : kayıt yılı, 20 ile 24 arasında;
- `c` : bölüm kodu, 6 ya da 7;
- `de` : o yıl o bölümdeki kayıt sırası, 00 ile 99 arasında (iki hane);
- öteki haneler (“070” ve “0”) bütün numaralarda aynıdır ve bilgi taşımaz.

Bu biçimdeki bütün numaraları 0–999 aralığına dağıtan fonksiyon şudur:

> *hash(ab070c0de) = ((ab − 20) × 100 + (c − 6) × 500 + de) mod 1000*

Fonksiyonun her terimi bir işe yarar. *ab − 20*, 0 ile 4 arasındadır ve 100'lük bir blok seçer: 2020 kaydı 0–99, 2021 kaydı 100–199 bloğuna düşer. *c − 6*, 0 ya da 1'dir; 7 kodlu bölüm 500 kaydırılır. *de*, 0–99 arasındadır ve bloğun içindeki yeri verir.

Numara	`ab`	`c`	`de`	Hesap	İndis
210706003	21	6	03	(1 × 100 + 0 × 500 + 3) mod 1000	103
240707050	24	7	50	(4 × 100 + 1 × 500 + 50) mod 1000	950
230707042	23	7	42	(3 × 100 + 1 × 500 + 42) mod 1000	842

Tablo 12.24: Üç okul numarasının indisi.

Olası numara sayısı *5 yıl × 2 bölüm × 100 sıra = 1000*'dir ve fonksiyon bunların her birini farklı bir indise gönderir. Bölüm 6'nın numaraları 0–499, bölüm 7'ninkiler 500–999 aralığını doldurur; her yıl kendi 100'lük bloğunu kullanır. Hiç çakışma yoktur. Bu tür fonksiyona **mükemmel hash** (*perfect hash*) denir. Mükemmel hash ancak anahtar kümesi önceden bilindiğinde mümkündür; burada bu bilgiyi numaranın biçimi sağlar.

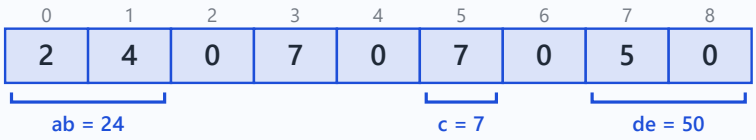
Fonksiyonun Java kodu şöyledir:

```
// s: "ab070c0de" biçiminde okul numarası
public int calculateHash(String s) {
  int ab = Integer.parseInt(s.substring(0, 2));
  int c  = Integer.parseInt(s.substring(5, 6));
  int de = Integer.parseInt(s.substring(7, 9));
  int yil   = (ab - 20) * 100;   // yıl bloğu
  int bolum = (c - 6) * 500;     // bölüm: 0/500
  return (yil + bolum + de) % 1000;
}
```

`substring(başlangıç, bitiş)` metodu başlangıç dahil, bitiş hariç karakterleri döndürür. `s.substring(0, 2)` 0. ve 1. karakterleri, yani yılı verir. `s.substring(5, 6)` yalnız 5. karakteri, yani bölüm kodunu verir. `s.substring(7, 9)` 7. ve 8. karakterleri, yani kayıt sırasını verir. `Integer.parseInt` bu parçaları tam sayıya çevirir; "03" 3 olur. Kalan satırlar formülü adım adım hesaplar.

Değişken	"240707050"	"210706003"
`ab = substring(0, 2)`	"24" → 24	"21" → 21
`c = substring(5, 6)`	"7" → 7	"6" → 6
`de = substring(7, 9)`	"50" → 50	"03" → 3
`yil = (ab - 20) * 100`	400	100
`bolum = (c - 6) * 500`	500	0
Dönüş `(yil + bolum + de) % 1000`	950	103

Tablo 12.25: `calculateHash` izlemesi.



Şekil 12.28: `calculateHash("240707050")` : `ab = 24` (0–1. karakterler), `c = 7` (5. karakter), `de = 50` (7–8. karakterler); sonuç *400 + 500 + 50 = 950*.

Metot sabit sayıda işlem yapar; maliyeti *O(1)*'dir. Ama yalnız biçime uyan numaralar için doğrudur. Biçime uymayan bir girdi, örneğin yılı 25 olan 250707001, *(500 + 500 + 1) mod 1000 = 1* verir ve 200706001 numarasıyla çakışır. Bölüm kodu 8 olan bir numara 1000'i aşan bir toplam üretir ve başka bir numaranın indisine sarılır. Yılı 20'den küçük bir numara negatif sonuç da verebilir. Gerçek bir programda girdi önce doğrulanmalı, biçim dışı numaralar reddedilmeli ya da genel amaçlı bir hash tablosuna yönlendirilmelidir.

Sık yapılan hata

`substring`'in bitiş indisini dahil sanmak. `s.substring(5, 6)` tek karakter döndürür; bölüm kodu için `s.substring(5, 5)` yazılırsa boş dizgi elde edilir ve `Integer.parseInt("")` `NumberFormatException` fırlatır. Bir karakterlik parça için bitiş, başlangıçtan bir fazladır.

12.6 Sık yapılan hatalar

Hash tablosu kullanan programlarda aynı hatalar sık sık tekrarlanır. Bunların çoğu derleme sırasında yakalanmaz ve ancak belirli girdilerde ortaya çıkar. Tablo 12.26 bölüm boyunca anlatılan hataları bir arada verir.

Hata	Sonucu	Doğrusu
`Math.abs(h) % N`	`Math.abs(Integer.MIN_VALUE)` negatiftir; dizi sınırı hatası.	`Math.floorMod(h, N)`

Hata	Sonucu	Doğrusu
Yalnız `equals`'ı ezmek	Eşit anahtarlar farklı kovalara düşer; `get` `null` döner.	`equals` ve `hashCode` aynı alanlarla birlikte ezilir; ya da `record` kullanılır.
Değiştirilebilir anahtar	Tabloya girdikten sonra hash alanı değişen anahtar eski kovada kalır; bulunamaz.	Anahtar olarak `String`, `Integer`, `record` gibi değişmez türler.
`Integer`'ları `==` ile karşılaştırmak	−128 … 127 dışındaki kutulu sayılar ayrı nesnelerdir; `==` `false` verir.	`equals`
Açık adreslemede `null` ile silmek	Yoklama zinciri kopar; ötedeki anahtarlar "yok" sanılır.	"Silindi" (DEFUNCT) işareti
Sıra beklemek	`HashMap`'in gezinme sırası ekleme sırası değildir ve büyütmeyle değişebilir.	`LinkedHashMap` (ekleme sırası) ya da `TreeMap` (anahtar sırası)
Büyütmede diziyi kopyalamak	Anahtarlar eski indislerinde kalır; yeni boyla aranınca bulunamaz.	Her anahtar yeniden hash'lenir.
`get`'in `null`'ını "anahtar yok" saymak	`HashMap` `null` değer saklayabilir.	`containsKey(k)`

Tablo 12.26: Hash tablolarında sık yapılan hatalar.

Değiştirilebilir anahtar hatası özellikle sinsidir. Bir `ArrayList` anahtar olarak eşlemeye konup sonra listeye öğe eklenirse listenin `hashCode()`'u değişir. Örneğin [1, 2] listesinin hash kodu 994'tür; listeye 3 eklenince 30817 olur. `get` yeni kodla başka bir kovaya bakar ve `null` döndürür. Oysa çift hâlâ tablodadır ve `size()` 1'dir. Çift ne bulunabilir ne de silinebilir; tabloda erişilemez bir çöp olarak kalır.

Hataların çoğunun ortak kökü, hash tablosunun iki varsayımına dayanmasıdır: anahtarın hash kodu tabloda kaldığı sürece değişmez, ve eşit anahtarların hash kodu aynıdır. Bu iki varsayımdan biri bozulduğunda tablo hata vermez; yalnız yanlış cevap verir. Hash tablosuyla çalışan kodda ilk denetlenecek yer, anahtar sınıfının `equals` ve `hashCode` metotlarıdır.

12.7 Özet

- **Eşleme** (*map*), tekil anahtarlardan değerlere bir eşlemedir. Temel işlemleri `put`, `get` ve `remove`'dur; var olan bir anahtarla yapılan `put` değeri günceller ve eski değeri döndürür. Eşleme bir ADT'dir; hash tablosu onu gerçeklemenin en yaygın yoludur.
- **Doğrudan adresleme**, her olası anahtara bir hücre ayırır ve *O(1)*'de çalışır, ama büyük anahtar uzaylarında belleği israf eder ve tam sayı olmayan anahtarlarla çalışmaz.
- **Hash fonksiyonu**, anahtarı önce bir **hash koduna**, sonra **sıkıştırma** ile 0 … $N - 1$ aralığında bir indise çevirir. İyi bir fonksiyon belirleyici, hızlı ve düzgün dağılımlıdır ve anahtarın tamamını kullanır. Bölme yönteminde N asal seçilir; MAD yöntemi $((a \cdot k + b) \bmod p) \bmod N$'dir. Java'da indis için `Math.floorMod(h, N)` kullanılır.
- `equals` / `hashCode` **sözleşmesi:** eşit nesnelerin hash kodu aynı olmalıdır; aynı hash kodu eşitlik anlamına gelmez. Kova seçimini `hashCode`, gerçek eşitliği `equals` belirler. İkisi birlikte ezilir.
- **Çakışmalar**, anahtar uzayı tablodan büyük olduğu sürece kaçınılmazdır (güvercin yuvası ilkesi).

- **Ayrı zincirlemede** her kova bir bağlı liste tutar; yeni düğüm başa eklenebilir. İşlemler *O(1 + zincir uzunluğu)*'dur; beklenen maliyet *O(1 + λ)*'dır ve *λ > 1* olabilir. Silmede önceki düğüm izlenir.
- **Açık adreslemede** çiftler dizinin kendisinde durur ve *λ ≤ 1*'dir. **Doğrusal yoklama** birincil kümelenmeye, **karesel yoklama** ikincil kümelenmeye yol açar; karesel yoklama asal *N* ve *λ < 1/2* ile boş yuva bulmayı güvenceye alır. **Çift hash**, adımı anahtara bağlayarak kümelenmeyi azaltır; adım sıfır olmamalı ve *N* ile aralarında asal olmalıdır. Silmede yuvaya **silindi işareti** konur.
- **Yük faktörü** *λ = n/N*'dir. Doğrusal yoklamada başarısız arama ortalama *½(1 + 1/(1 − λ)²)* yoklama ister: *λ = 0,5*'te 2,5, *λ = 0,8*'de 13. Eşik aşılınca (zincirlemede ≈ 0,75, açık adreslemede ≈ 0,5) tablo yaklaşık iki katına büyütülür ve bütün anahtarlar yeniden hash'lenir. Büyütme tek başına *O(n)*, ekleme başına amortize *O(1)*'dir.
- Hash tablosunda ekleme, arama ve silme **ortalama *O(1)*, en kötü *O(n)***; alan *O(n + N)*'dir. Tablo sıra tutmaz.
- **Java** `HashMap` : zincirleme, 2'nin kuvveti kadar kova (varsayılan 16), `h ^ (h >>> 16)` ile bit karıştırma ve `& (n - 1)` maskesi, 0,75 eşiğinde iki katına büyüme, 8'i aşan zincirlerde (en az 64 kova varsa) kırmızı-siyah ağaç.
- Anahtar yapısı biliniyorsa **mükemmel hash** tasarlanabilir; okul numarası örneğinde 1000 olası numara 1000 farklı indise gider.

Yöntem	Çiftler nerede?	λ sınırı	Zayıf yönü	Silme
Ayrı zincirleme	Kovalardaki bağlı listeler	> 1 olabilir (pratikte ≈ 0,75)	Düğüm başına ek referans; dağınık bellek	Düğüm zincirden koparılır
Doğrusal yoklama	Dizinin kendisi	≤ 1 (pratikte ≈ 0,5)	Birincil kümelenme	Silindi işareti
Karesel yoklama	Dizinin kendisi	< 0,5 (asal *N*)	İkincil kümelenme; her yuvayı denemeyebilir	Silindi işareti
Çift hash	Dizinin kendisi	≤ 1 (pratikte ≈ 0,5)	İkinci hash hesabı; önbellek dostu değil	Silindi işareti

Tablo 12.27: Çakışma çözüm yöntemlerinin karşılaştırması.

12.8 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

eşleme *map* · anahtar *key* · değer *value* · doğrudan adresleme *direct addressing* · hash tablosu *hash table* · hash fonksiyonu *hash function* · hash kodu *hash code* · sıkıştırma *compression function* · kova *bucket* · çakışma *collision* · ayrı zincirleme *separate chaining* · açık adresleme *open addressing* · yoklama *probing* · doğrusal yoklama *linear probing* · karesel yoklama *quadratic probing* · çift hash *double hashing* · birincil kümelenme *primary clustering*

yük faktörü *load factor* | yeniden boyutlandırma *resizing (rehashing)* | amortize analiz *amortized analysis*

12.9 Kendini Dene

1. 5 kovalı, *h(k) = k mod 5* kullanan ayrı zincirlemeli bir tabloya sırasıyla 8, 13, 3, 21 ve 6 ekleniyor; yeni düğüm zincirin başına eklenir. Son tablo nedir, λ kaçtır ve `getir(8)` kaç anahtar karşılaştırması yapar?

   Cevap

   Kovalar: 8 → 3, 13 → 3, 3 → 3, 21 → 1, 6 → 1. Başa ekleme nedeniyle son gelen başta durur: 1. kova 6 → 21, 3. kova 3 → 13 → 8; 0, 2 ve 4. kovalar boştur. $\lambda = 5/5 = 1$. `getir(8)` 3. kovada 3, 13 ve 8'i inceler: 3 karşılaştırma.

2. "Can".hashCode() değerini `String.hashCode()` formülüyle hesaplayın (C = 67, a = 97, n = 110). 13 kovalı tabloda bu anahtar hangi kovaya düşer?

   Cevap

   h = 67; *31 · 67 + 97 = 2174*; *31 · 2174 + 110 = 67504*. *13 · 5192 = 67496*, kalan 8: kova 8. "Ali" de 8. kovaya düştüğü için iki anahtar çakışır.

3. 11 yuvalı, *h(k) = k mod 11* kullanan doğrusal yoklamalı boş bir tabloya sırasıyla 4, 15, 26 ve 5 ekleniyor. Her anahtar hangi yuvaya yerleşir ve kaç yoklama yapar?

   Cevap

   4 → 4 (1 yoklama). 15'in ev yuvası 4, dolu; 5 boş (2 yoklama). 26'nın ev yuvası 4; 4 ve 5 dolu, 6 boş (3 yoklama). 5'in ev yuvası 5; 5 ve 6 dolu, 7 boş (3 yoklama). Toplam 9 yoklama; 4–7 arası tek bir kümedir.

4. Önceki sorudaki tabloda 15 silinip 5. yuva `null` yapılırsa `getir(26)` ne bulur? Yuvaya silindi işareti konursa ne değişir?

   Cevap

   `null` ile: arama 4'ten başlar (4 ≠ 26), 5 boştur ve “26 yok” der; oysa 26 6. yuvadadır. Silindi işaretiyle: 4 (4 ≠ 26), 5 (✗, atlanır), 6'da 26 bulunur; 3 yoklama.

5. `yuvaBul` bir aramada −1 döndürdü. Bu sonuç ne anlama gelir? Metot `-(uygun + 1)` yerine `-`

`uygun` döndürseydi hangi durum ayırt edilemezdi?

Cevap

Negatif sonuç "anahtar yok" demektir; eklenecek yuva $-(-1 + 1) = 0$'dır. `-uygun` döndürülseydi uygun yuva 0 olduğunda sonuç da 0 olurdu ve "0. yuvada bulundu" ile karışırdı.

6. 11 yuvalı tabloda $h(k) = k \bmod 11$ ve $h_2(k) = 7 - (k \bmod 7)$ ile çift hash kullanılıyor. Boş tabloya 13, 24 ve 35 eklenirse hangi yuvalara yerleşirler? Doğrusal yoklamada ne olurdu?

Cevap

Üçünün de ev yuvası 2'dir. 13 → 2. 24: adım $7 - 3 = 4$, 2 dolu, $2 + 4 = 6$. 35: adım $7 - 0 = 7$, 2 dolu, $2 + 7 = 9$. Doğrusal yoklamada 2, 3, 4 yuvalarına yerleşir ve bir küme oluştururlardı.

7. Varsayılan ayarlarla oluşturulmuş bir Java `HashMap`'e anahtarlar tek tek ekleniyor. Kaçıncı eklemede tablo ilk kez büyür ve kaç kovaya çıkar? Bir sonraki büyüme kaçıncı eklemededir?

Cevap

Varsayılan kova sayısı 16, eşik $0{,}75 \cdot 16 = 12$'dir. Öğe sayısı 12'yi aşınca, yani 13. eklemede tablo 32 kovaya büyür. Yeni eşik $0{,}75 \cdot 32 = 24$'tür; 25. eklemede tablo 64 kovaya çıkar.

8. Okul numarası 220706099 ve 220707099 için $((ab - 20) \times 100 + (c - 6) \times 500 + de) \bmod 1000$ değerlerini bulun.

Cevap

İkisinde de $ab = 22$, $de = 99$. 220706099: $c = 6$, $200 + 0 + 99 = 299$. 220707099: $c = 7$, $200 + 500 + 99 = 799$.

9. Doğrusal yoklamalı bir tabloda $\lambda = 2/3$'tür. Knuth'un formüllerine göre başarılı ve başarısız bir arama ortalama kaç yoklama yapar?

Cevap

$1 - \lambda = 1/3$. Başarılı: $\tfrac{1}{2}(1 + 3) = 2$. Başarısız: $\tfrac{1}{2}(1 + 9) = 5$.

10. Bir `Nokta` sınıfında `equals` iki koordinatı karşılaştırıyor, `hashCode` ise `x + y` döndürüyor. Sözleşme sağlanır mı? Boş bir `HashMap`'e `new Nokta(1, 2)` ve `new Nokta(2, 1)` anahtarlarıyla

iki `put` yapılırsa `size()` kaç olur?

Cevap

Sözleşme sağlanır: eşit noktaların koordinatları aynı olduğundan `x + y` de aynıdır. (1, 2) ile (2, 1)'in hash kodları aynıdır (3), ama `equals`'a göre farklıdırlar. İkisi aynı kovaya düşer ve ayrı çiftler olarak saklanır: `size()` 2'dir. Fonksiyon doğrudur ama zayıftır; toplamı aynı olan bütün noktalar çakışır.

12.10 İleri okuma

- M. T. Goodrich, R. Tamassia ve M. H. Goldwasser, *Data Structures and Algorithms in Java* (6. baskı, 2014), Bölüm 10: eşleme ADT'sini, hash kodu ve sıkıştırma ayrımını, MAD yöntemini, zincirleme ve doğrusal yoklamalı Java gerçeklemelerini bu nottakine yakın bir yapıyla verir.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 11: zincirlemenin ve açık adreslemenin beklenen maliyetlerini olasılık hesabıyla kanıtlar; evrensel hash fikrini tanıtır.
- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 3.4: zincirleme ve doğrusal yoklamalı sembol tablolarını Java koduyla verir; yük faktörünün etkisini deneylerle gösterir.
- D. E. Knuth, *The Art of Computer Programming, Cilt 3: Sorting and Searching* (2. baskı, 1998), Kesim 6.4: doğrusal yoklamanın beklenen yoklama sayısı formüllerinin kaynağıdır; hash fikrinin tarihçesini de anlatır.

12.11 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022.
2. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6. baskı, Wiley, 2014.
3. D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2. baskı, Addison-Wesley, 1998.
4. Oracle, *Java SE 17 API Specification*: `java.util.Map`, `java.util.HashMap` ve `java.util.HashSet` belgeleri.
5. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011.

KISIM IV · BÖLÜM 13

Kümeler ve Ayrık Kümeler

Bir öğrenci listesinde aynı ad iki kez yazılmış mı, bir kullanıcı yasaklılar arasında mı, iki şehir aynı yol ağında mı? Bu soruların hepsi "şu öğe şu toplulukta var mı?" sorusunun bir biçimidir. Kümeler bu soruyu hızlı yanıtlamak için kurulan yapılardır. Not önce küme kavramını ve Java'daki `Set` arayüzünü anlatır. Ardından aynı kümenin hash tablosu, dengeli arama ağacı, sıralı dizi ve bit vektörüyle nasıl gerçeklendiğini ve bu seçimlerin maliyetlerini karşılaştırır. Tekrarları sayan çoklu küme ile bir anahtara birden çok değer bağlayan çoklu eşleme bunu izler. Notun ikinci yarısı ayrık kümelere ayrılmıştır: öğeleri birbirine değmeyen gruplara bölen ve grupları birleştirmeyi neredeyse sabit sürede yapan Union-Find yapısı. Boyuta ve rütbeye göre birleştirme, yol kısaltma ve bağlantılı bileşen sayma bu yapının ana başlıklarıdır.

Bu bölümün kodu: `github.com/sercankulcu/data-structures-java/Bolum13/src`

ÖĞRENME ÇIKTILARI

Küme modeli
Küme ADT'sini ve Java `Set` arayüzünü açıklamak; birleşim, kesişim, fark ve alt küme işlemlerini hem elle hem `Set` metotlarıyla hesaplamak.

Gerçeklemeler
Hash tabanlı, ağaç tabanlı, sıralı dizi ve bit vektörü gerçeklemelerinin işleyişini izlemek ve problem için uygun olanı maliyetine göre seçmek.

Çoklu küme
Tekrarlı öğe gerektiren problemlerde çoklu küme ve çoklu eşleme yaklaşımını kullanmak.

Union-Find
Ayrık kümeleri ebeveyn dizisiyle temsil etmek; yol kısaltma ile rütbeye (ya da boyuta) göre birleştirme yöntemlerini uygulamak ve etkilerini açıklamak.

Uygulama
Union-Find yapısını bağlantılı bileşen sayma gibi bağlılık problemlerinde kullanmak.

13.1 Küme ADT'si

Bir veri yapısını anlatmanın iki düzeyi vardır. Birinci düzey yapının ne yaptığını söyler: hangi işlemler vardır ve her işlem neyi değiştirir. İkinci düzey bunun nasıl yapıldığını anlatır: veriler bellekte nasıl dizilir, işlemler hangi adımlarla yürür. Birinci düzeydeki tarife **soyut veri tipi** (*abstract data type*, kısaca ADT) denir. Bu kesim kümeyi soyut düzeyde ele alır; hash tablosu, ağaç ve bit vektörü gibi somut gerçeklemeler 13.2'nin konusudur.

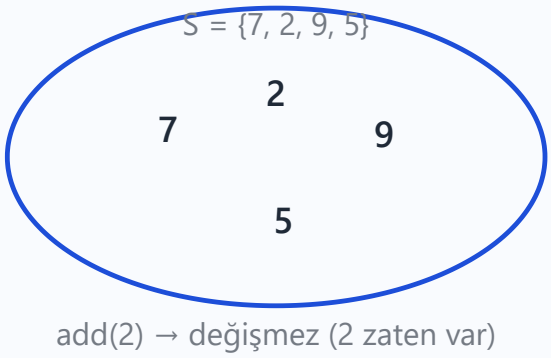
13.1.1 Küme nedir?

Bir sınıfın yoklama listesini düşünelim. Bir öğrencinin adı listeye bir kez yazılır; aynı adı ikinci kez yazmak listeye bilgi katmaz. Adların hangi sırayla yazıldığı da önemli değildir. Önemli olan tek soru şudur: "Ayşe bu sınıfta mı?" Matematikteki küme kavramı tam olarak bu düşünceyi taşır. Veri yapıları da aynı kavramı bilgisayara taşır.

> **Tanım 13.1**
>
> **Küme** (*set*), birbirinden farklı (tekil) öğelerden oluşan bir koleksiyondur. Aynı öğe kümede iki kez bulunmaz ve öğelerin sırası önemsizdir: *{1, 2, 3} = {3, 1, 2}*. Küme ADT'sinin temel işlemleri öğe eklemek, öğe çıkarmak ve **üyelik testidir**: "x kümede var mı?"

Tanımda iki kural öne çıkar. Birincisi *tekrar yasağıdır*. Kümede zaten bulunan bir öğe yeniden eklenirse küme değişmez. Bu kural, sıradan bir listeyle küme arasındaki en belirgin farktır. Listeye aynı değer istendiği kadar eklenebilir; kümeye eklenemez. İkinci kural *sırasızlıktır*. Kümenin "ilk öğesi" ya da "3. öğesi" diye bir kavramı yoktur; bu yüzden küme ADT'sinde indisle erişim işlemi bulunmaz. Bazı gerçeklemeler öğeleri belli bir sırayla gezer, ama bu sıra gerçeklemenin bir yan ürünüdür, kümenin tanımına ait değildir.

Kümeden beklenen en önemli yetenek üyelik testini hızlı yanıtlamaktır. Bir dizide aynı soruyu yanıtlamak için öğeler baştan sona taranır; *n* öğede bu *O(n)* sürer. İyi bir küme gerçeklemesi aynı soruyu ortalama sabit sürede ya da *O(log n)* sürede yanıtlar. Bölümün büyük kısmı bu hızın nasıl elde edildiğini anlatır.



Şekil 13.1: 7, 2, 9 ve 5 öğelerinden oluşan *S* kümesi. Öğeler sırasız çizilmiştir. Kümede zaten bulunan 2 yeniden eklenmek istendiğinde (`add(2)`) küme değişmez.

Şekil 13.1 bu iki kuralı birlikte gösterir. *S = {7, 2, 9, 5}* kümesine `add(2)` uygulandığında 2 zaten kümededir; öğe sayısı 4 olarak kalır. Şekilde öğelerin dağınık çizilmesi de bilinçli bir seçimdir: kümede "önce" ya da "sonra" yoktur.

Kümeler gündelik programlamada üç tür işte karşımıza çıkar:

- **Tekrar ayıklama.** Bir metindeki farklı kelimeleri bulmak için kelimeler bir kümeye eklenir; tekrarlar kendiliğinden düşer. Java'da bir listeden kümeye geçmek tek satırdır: `new HashSet<>(liste)`.
- **Üyelik.** Yasaklı kullanıcılar, daha önce ziyaret edilmiş düğümler, sözlükteki geçerli kelimeler birer kümedir. Her yeni istekte "bu öğe kümede mi?" diye sorulur.
- **Küme cebri.** İki kişinin ortak arkadaşları iki arkadaş kümesinin kesişimidir. Bir sitedeki bütün etiketler, yazıların etiket kümelerinin birleşimidir.

Sık yapılan hata

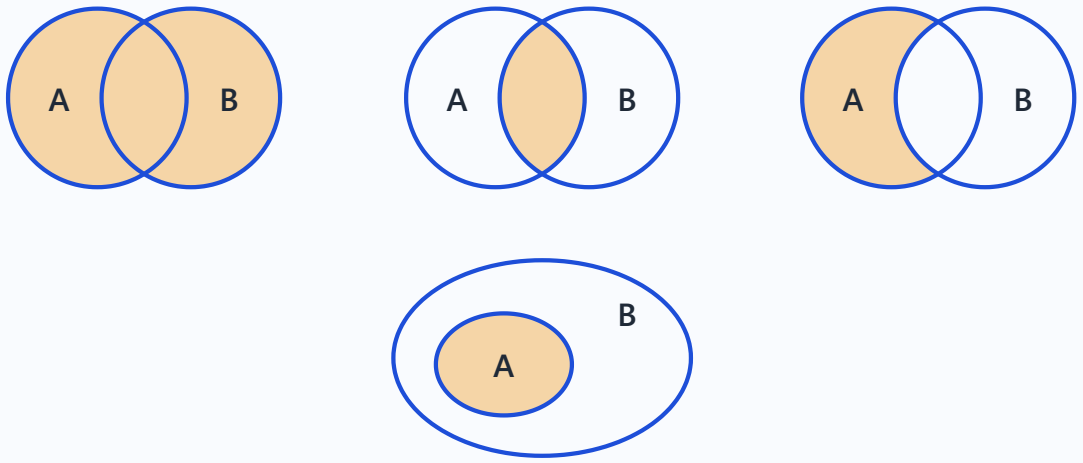
Kümeyi "sıralı liste" sanmak. `HashSet` yazdırıldığında öğeler bazen küçükten büyüğe çıkar; küçük tam sayılarda bu sık olur. Bu bir rastlantıdır: öğeler hash kodlarına göre kovalara yerleşir ve küçük tam sayıların hash kodu kendisine eşittir. Dizgiler ya da büyük sayılar eklendiğinde sıra bozulur. Sıra gerekiyorsa sıralı bir küme (`TreeSet`) seçilir.

13.1.2 Küme işlemleri

Kümeler üzerinde dört temel işlem tanımlanır. Bunlara küme **işleçleri** de denir. *A* ve *B* iki küme olsun.

- **Birleşim** (*union*), $A \cup B$: A'da ya da B'de (ya da ikisinde birden) bulunan öğelerin kümesi.
- **Kesişim** (*intersection*), $A \cap B$: hem A'da hem B'de bulunan öğelerin kümesi.
- **Fark** (*difference*), $A - B$: A'da bulunup B'de bulunmayan öğelerin kümesi.

- **Alt küme** (*subset*), $A \subseteq B$: A'nın her öğesi B'de de bulunuyorsa A, B'nin alt kümesidir. Bu bir işlem değil, doğru ya da yanlış yanıtı olan bir sorudur.



Şekil 13.2: Dört küme işleminin Venn şemaları (soldan sağa). (a) Birleşim $A \cup B$: iki dairenin tamamı taralı. (b) Kesişim $A \cap B$: yalnız ortak bölge taralı. (c) Fark $A - B$: A'nın B ile ortak olmayan kısmı taralı. (d) Alt küme $A \subseteq B$: A dairesi B'nin içinde.

Şekil 13.2 bu işlemleri Venn şemalarıyla gösterir. Venn şemasında her küme bir daire, öğeler dairenin içindeki noktalardır. Birleşimde iki daire de tümüyle taranır. Kesişimde yalnız iki dairenin örtüştüğü mercek biçimli bölge kalır. Farkta A'nın B ile örtüşen kısmı oyulur. Alt küme ilişkisinde A dairesi B'nin tamamen içindedir.

Farkın iki yönü olduğuna dikkat etmek gerekir. Birleşim ve kesişim değişmelidir: $A \cup B = B \cup A$ ve $A \cap B = B \cap A$. Fark ise değişmeli değildir; $A - B$ ile $B - A$ genellikle farklı kümelerdir.

Örnek 13.1

$A = \{1, 2, 3, 4, 5\}$ ve $B = \{3, 4, 5, 6, 7\}$ için birleşim, kesişim, iki yönlü fark ve alt küme sorularını yanıtlayalım.

1. **Birleşim.** A'nın bütün öğeleri alınır: 1, 2, 3, 4, 5. B'den yalnız A'da olmayanlar eklenir: 6 ve 7. Ortak öğeler (3, 4, 5) bir kez yazılır. $A \cup B = \{1, 2, 3, 4, 5, 6, 7\}$.
2. **Kesişim.** A'nın her öğesi için B'de olup olmadığı sorulur. 1 ve 2 B'de yoktur; 3, 4, 5 vardır. $A \cap B = \{3, 4, 5\}$.
3. **Fark.** A'dan B'de bulunanlar çıkarılır: $A - B = \{1, 2\}$. Ters yönde B'den A'da bulunanlar çıkarılır: $B - A = \{6, 7\}$.
4. **Alt küme.** $A \subseteq B$ doğru mu? 1, A'da vardır ama B'de yoktur ($1 \notin B$); bu tek karşı örnek yeter: $A \not\subset B$. Buna karşılık $\{3, 4\} \subseteq A$ doğrudur, çünkü 3 de 4 de A'dadır.
5. **Denetim.** Birleşimin öğe sayısı $|A| + |B| - |A \cap B| = 5 + 5 - 3 = 7$'dir. Sayım tutar.

İşlem	Sonuç
Birleşim $A \cup B$	{1, 2, 3, 4, 5, 6, 7}
Kesişim $A \cap B$	{3, 4, 5}
Fark $A - B$ / $B - A$	{1, 2} / {6, 7}
Alt küme?	$A \not\subset B$ ($1 \notin B$) · $\{3, 4\} \subseteq A$ ✓

Tablo 13.1: $A = \{1, 2, 3, 4, 5\}$, $B = \{3, 4, 5, 6, 7\}$ için küme işlemlerinin sonuçları.

Örnekteki son adım, birleşimin boyutunu denetlemek için kullanışlı bir eşitliği gösterir: $|A \cup B| = |A| + |B| - |A \cap B|$. Ortak öğeler iki kümenin boyutunda da sayıldığı için bir kez çıkarılır. Bu eşitliğe *içerme–dışlama ilkesi* denir.

13.1.3 Java'da Set arayüzü

Java koleksiyon kütüphanesinde küme ADT'sinin karşılığı `java.util.Set<E>` arayüzüdür. Arayüz yalnız işlemleri tanımlar; işlemleri yapan sınıflar `HashSet`, `LinkedHashSet` ve `TreeSet`'tir. `Set`, `Collection` arayüzünü genişletir ve ona tek bir sözleşme ekler: kümede birbirine eşit iki öğe bulunamaz. Eşitlik `equals` metoduyla (sıralı kümede `compare` ile) belirlenir.

Metot	Açıklama
`add(e)`	e'yi ekler; e zaten varsa küme değişmez ve `false` döner.
`remove(o)`, `contains(o)`	öğeyi çıkarır; üyeliği sorgular.
`addAll(T)`	birleşim: $S \leftarrow S \cup T$
`retainAll(T)`	kesişim: $S \leftarrow S \cap T$
`removeAll(T)`	fark: $S \leftarrow S - T$
`containsAll(T)`	alt küme sorusu: $T \subseteq S$ mi?

Tablo 13.2: `java.util.Set` arayüzünün başlıca metotları ve küme işlemleri karşılıkları.

`add` metodunun dönüş değeri küçük ama değerli bir ayrıntıdır. Metot yalnız öğeyi eklemekle kalmaz, ekleyip eklemediğini de bildirir. Öğe yeniyse `true`, zaten varsa `false` döner. Bu tek bit bilgi, 13.1.4'teki tekrar bulma algoritmasının temelidir. `remove` da benzer biçimde öğeyi gerçekten çıkardıysa `true` döndürür.

Küme işlemlerinin Java karşılıkları `addAll`, `retainAll` ve `removeAll`'dur. Aşağıdaki kod Örnek 13.1'deki hesabı Java ile yapar:

```
Set<Integer> a = new HashSet<>(List.of(1,2,3,4,5));
Set<Integer> b = new HashSet<>(List.of(3,4,5,6,7));
Set<Integer> bir = new HashSet<>(a);
bir.addAll(b);     // [1..7]
Set<Integer> kes = new HashSet<>(a);
kes.retainAll(b);  // [3, 4, 5]
Set<Integer> fark = new HashSet<>(a);
fark.removeAll(b); // [1, 2]
a.containsAll(b);  // false
```

Kodu satır satır okuyalım. İlk iki satır `List.of` ile verilen değerlerden iki `HashSet` kurar. Üçüncü satır `a`'nın bir *kopyasını* `bir` adıyla oluşturur; `bir.addAll(b)` bu kopyaya B'nin öğelerini ekler. Kopyada zaten bulunan 3, 4, 5 yeniden eklenmez; 6 ve 7 eklenir. Sonuç 1'den 7'ye kadar olan sayılardır. Kesişim için yine `a`'nın kopyası alınır ve `retainAll(b)` çağrılır. Bu metot, B'de bulunmayan her öğeyi siler; geriye `[3, 4, 5]` kalır. Fark için kopyaya `removeAll(b)` uygulanır; B'de bulunan her öğe silinir, geriye `[1, 2]` kalır. Son satırdaki `a.containsAll(b)`, "B'nin her öğesi A'da mı?" sorusunu, yani $B \subseteq A$ sorusunu sorar. 6, A'da olmadığı için yanıt `false`'tur.

Üç toplu metodun maliyeti, gerçeklemeye göre değişir. `HashSet`'te `bir.addAll(b)`, B'nin her öğesi için bir `add` yapar. Her `add` ortalama $O(1)$ olduğundan toplam $O(|B|)$ sürer; kopyalama da $O(|A|)$ ekler. `retainAll` ve `removeAll` ise kopyadaki her öğe için öteki kümede `contains` sorar; bu da ortalama $O(|A|)$ sürer. Kısacası hash kümelerle iki kümenin birleşimi, kesişimi ve farkı ortalama $O(n + m)$ sürede bulunur.

> **Sık yapılan hata**
>
> `addAll`, `retainAll` ve `removeAll` yeni bir küme döndürmez. Çağrıldıkları kümeyi **yerinde değiştirir** ve yalnız "küme değişti mi?" bilgisini (`boolean`) döndürür. `Set<Integer> kes = a; kes.retainAll(b);` yazılırsa kopya oluşmaz: `kes` ile `a` aynı nesneyi gösterir ve `a` da kesişime dönüşür. A'yı korumak için önce kopya alınır: `new HashSet<>(a)`.

13.1.4 Kod izleme: add dönüşüyle ilk tekrar

Bir dizide ilk tekrar eden öğeyi bulmak isteyelim: soldan sağa okurken daha önce görülmüş ilk değer hangisidir? Kümesiz çözüm her öğeyi kendinden önceki bütün öğelerle karşılaştırır. Bu iç içe iki döngü demektir ve en kötü durumda $n(n-1)/2$ karşılaştırma, yani $O(n^2)$ yapar. Kümeyle çözüm, görülen öğeleri bir kümede biriktirir ve her yeni öğeyi tek bir `add` çağrısıyla denetler:

```
int ilkTekrar(int[] a) {
  Set<Integer> gorulen = new HashSet<>();
  for (int x : a) {
    if (!gorulen.add(x))   // eklenemedi mi?
      return x;            // zaten vardı: tekrar
  }
  return -1;               // tekrar yok
}
```

Metot boş bir `HashSet` ile başlar ve dizinin öğelerinde sırayla gezinir. Döngünün gövdesindeki tek satır iki işi birden yapar. `gorulen.add(x)` öğeyi eklemeyi dener; öğe yeniyse ekler ve `true` döndürür. Başına konan `!` bu değeri tersine çevirir: `!true` yanlıştır, koşul sağlanmaz ve döngü sürer. Öğe daha önce eklenmişse `add` kümeyi değiştirmeden `false` döndürür; `!false` doğrudur ve metot o öğeyi döndürür. Dizide hiç tekrar yoksa döngü biter ve `-1` döner.

Bu kalıp, önce `contains` ile sorup sonra `add` çağırmaktan daha kısadır. Üstelik kümeye bir kez erişir: `contains` + `add` çifti aynı kovayı iki kez arardı.

Metodu `a = [4, 7, 2, 7, 4]` dizisiyle izleyelim. `HashSet` içeride 16 kovalı bir tablo kullanır. Bir öğenin kovası, hash kodunun son dört biti alınarak bulunur: `hash(x) & 15`. `Integer` için hash kodu sayının kendisidir; bu yüzden 4 kova 4'e, 7 kova 7'ye, 2 kova 2'ye düşer.

Tur	x	Kova	add(x)	gorulen (sonra)	Sonuç
1	4	4	`true`	{4}	döngü sürer
2	7	7	`true`	{4, 7}	döngü sürer
3	2	2	`true`	{2, 4, 7}	döngü sürer
4	7	7	`false`	{2, 4, 7}	**7 döner**

Tablo 13.3: `ilkTekrar([4, 7, 2, 7, 4])` izlemesi. Kova, `x & 15` ile bulunur.

İlk üç turda kovalar boştur; her öğe tek bir kova bakışıyla eklenir ve `add` `true` döndürür. Dördüncü turda 7 gelir. Kova 7'de zaten 7 vardır; küme değişmez ve `add` `false` döndürür. Koşul sağlanır, metot 7'yi döndürür. Dizinin son öğesi olan 4'e hiç bakılmaz: metot ilk tekrarı bulduğu anda durur.





Şekil 13.3: `ilkTekrar` izlemesinden iki an. Üstte üçüncü tur: 2 kova 2'ye eklenmiş, küme {2, 4, 7}. Altta dördüncü tur: `x = 7` için kova 7'de 7 zaten bulunuyor; `add` `false` döndürür ve metot 7'yi verir.

Maliyet hesabı kısadır. Her öğe için bir `add` yapılır ve her `add` ortalama *O(1)*'dir. *n* öğeli dizide toplam süre ortalama *O(n)*'dir; iç içe döngü yoktur. Bu hızın bedeli bellektir: küme en çok *n* öğe tutar, ek alan *O(n)*'dir. Zaman ile bellek arasındaki bu takas, kümelerin en tipik kullanımıdır.

13.2 Küme gerçeklemeleri

Küme ADT'si yalnız "ne" sorusunu yanıtlar. Aynı ADT birçok farklı biçimde gerçeklenebilir ve her gerçekleme bazı işlemleri hızlı, bazılarını yavaş yapar. Bu kesim dört gerçeklemeyi inceler: hash tablosu, dengeli ikili arama ağacı, sıralı dizi ve bit vektörü. Sonunda hangisinin hangi durumda seçileceği bir tabloda toplanır.

13.2.1 Java'daki Set sınıfları

Java, `Set` arayüzünü üç hazır sınıfla uygular. Üçü de aynı sözleşmeye uyar: tekrar tutmaz, `add`, `remove` ve `contains` işlemlerini sunar. Aralarındaki fark iç yapıda ve bu yapının getirdiği gezinme sırasındadır.

Sınıf	İç yapı	Gezinme sırası	`add` / `contains` / `remove`
`HashSet`	hash tablosu	hash koduna bağlı, öngörülemez	ortalama O(1)
`LinkedHashSet`	hash tablosu + çift bağlı liste	ekleme sırası	ortalama O(1)
`TreeSet`	kırmızı-siyah ağaç	sıralı (küçükten büyüğe, alfabetik)	O(log n)

Tablo 13.4: `java.util.Set` gerçeklemeleri.

Üç sınıfın farkını görmek için aynı dört kelimeyi üçüne de ekleyelim:

```
for (String s : List.of("elma", "kiraz", "armut", "elma")) {
    hash.add(s);  linked.add(s);  tree.add(s);
}
// HashSet       → [elma, armut, kiraz]   (kova sırası: hash'e bağlı)
// LinkedHashSet → [elma, kiraz, armut]   (eklenme sırası)
// TreeSet       → [armut, elma, kiraz]   (alfabetik)
```

“elma” iki kez eklenmiştir, ama üç kümede de bir kez bulunur: ikinci `add("elma")` üçünde de `false` döndürür. Yazdırılan sıralar ise farklıdır. `HashSet` öğeleri kovalarının sırasıyla verir. “elma”nın hash kodu 16 kovalı tabloda 4. kovaya, “armut”unki 7. kovaya, “kiraz”ınki 8. kovaya düşer. Bu yüzden çıktı `[elma, armut, kiraz]`'dır. Bu sıra ne ekleme sırasıdır ne alfabetik; tablonun boyutu değişirse o da değişebilir. `LinkedHashSet` her öğeyi ayrıca bir çift bağlı listenin sonuna ekler ve gezinmeyi bu listeden yapar; sonuç ekleme sırasıdır: `[elma, kiraz, armut]`. `TreeSet` öğeleri bir arama ağacında tutar ve kök ortada gezinmeyle sıralı verir: `[armut, elma, kiraz]`.

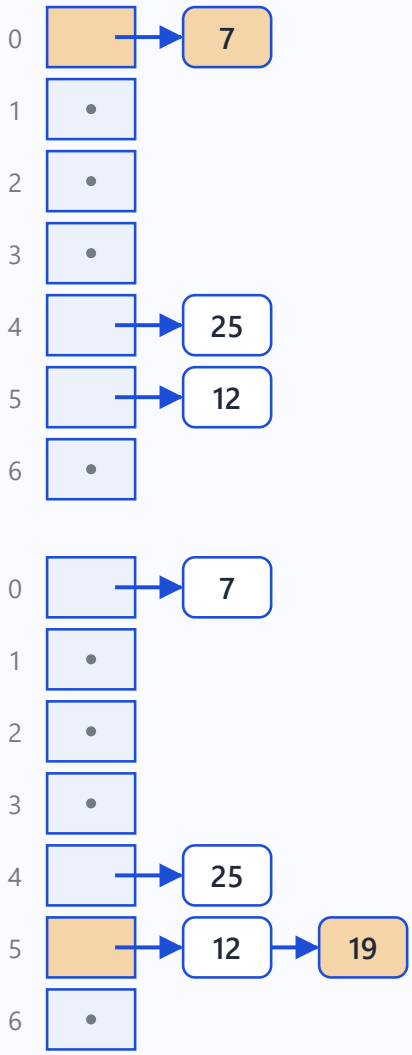
Seçim kuralı buradan çıkar. Sıra önemsizse `HashSet`, ekleme sırası gerekiyorsa `LinkedHashSet`, sıralı erişim gerekiyorsa `TreeSet` kullanılır. `LinkedHashSet` sıranın bedelini bellekle öder: her öğe için iki ek referans tutar. `TreeSet` ise zamanla öder: işlemleri *O(1)* yerine *O(log n)*'dir.

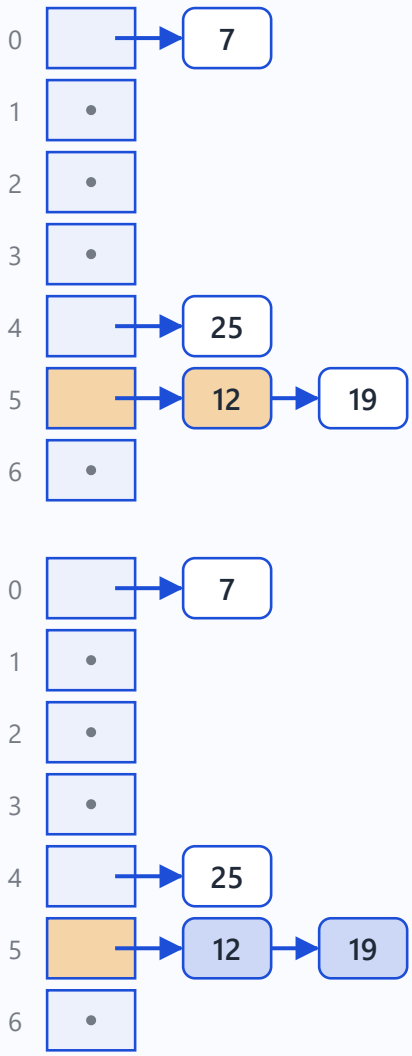
13.2.2 Hash tabanlı küme

Hash tabanlı küme, öğeleri bir **hash tablosunda** (*hash table*) saklar. Hash tablosu, *m* hücreli bir dizidir; her hücreye **kova** (*bucket*) denir. Bir öğenin hangi kovaya gideceği, öğenin **hash kodundan** (*hash code*) hesaplanır. Hash kodu, öğeden üretilen bir tam sayıdır; eşit öğelerin hash kodları da eşittir. Kova numarası çoğu zaman hash kodunun *m*'ye bölümünden kalandır: *i = hashCode(x) mod m*.

Eşlemeler için kullanılan hash tablosu her kovada bir anahtar–değer çifti tutar. Kümede değere gerek yoktur; tabloda yalnız anahtarlar, yani küme öğeleri saklanır. Java'nın `HashSet` sınıfı da içeride bir `HashMap` kullanır: öğeyi anahtar olarak koyar, değer yerine hep aynı boş nesneyi yazar.

İki farklı öğe aynı kovaya düşebilir; buna **çakışma** (*collision*) denir. 7 kovalı bir tabloda 12 ile 19'un ikisi de 5'e düşer, çünkü *12 mod 7 = 5* ve *19 mod 7 = 5*'tir. Çakışmayı çözmenin en basit yolu **ayrı zincirlemedir** (*separate chaining*): her kova bir liste tutar ve aynı kovaya düşen öğeler bu listeye (zincire) sıralanır.


0
7
1
2
3
4
25
5
12
6
0
7
1
2
3
4
25
5
12
19
6



Şekil 13.4: 7 kovalı zincirlemeli hash kümesi; konum *hashCode(x) mod 7*. (a) 12, 25 ve 7 eklendikten sonra: kova 5'te 12, kova 4'te 25, kova 0'da 7. (b) `add(19)` : 19 da kova 5'e düşer, zincirin sonuna eklenir. (c) `add(12)` yeniden: kova 5 taranır, 12 bulunur, eklenmez. (d) `contains(26)` : yalnız kova 5 (12, 19) taranır; 26 bulunamaz.

Şekil 13.4 boş bir 7 kovalı tabloya sırasıyla 12, 25, 7, 19 ve yeniden 12 eklendiğinde olanları gösterir. `add(12)` : *12 mod 7 = 5*, kova 5 boştur, 12 eklenir. `add(25)` : *25 mod 7 = 4*, 25 kova 4'e gider. `add(7)` : *7 mod 7 = 0*, 7 kova 0'a gider. `add(19)` : *19 mod 7 = 5*; kova 5'te 12 vardır. Bu bir çakışmadır; 19, 12'den farklı olduğu için zincirin sonuna eklenir. İkinci `add(12)` 'de kova 5 taranır ve 12 bulunur; öğe eklenmez, metot `false` döndürür. Kümeyi küme yapan işte bu denetimdir.

Son adım hash tablosunun neden hızlı olduğunu açıklar. `contains(26)` için yalnız *26 mod 7 = 5* numaralı kovaya bakılır. Kovadaki iki öğe (12 ve 19) 26 ile karşılaştırılır, ikisi de farklıdır, yanıt `false` 'tur. Kova 0 ve kova 4'teki öğelere hiç dokunulmaz. Tablo ne kadar büyük olursa olsun arama tek bir kovanın zinciriyle sınırlıdır.

Bu zincirlerin uzunluğu ne kadardır? n öğe m kovaya düzgün dağılırsa bir zincirin ortalama uzunluğu n / m'dir. Bu orana **yük faktörü** (*load factor*) denir ve genellikle λ ile gösterilir. Örnekteki tabloda 4 öğe ve 7 kova vardır: $\lambda = 4/7 \approx 0{,}57$. Hash tablosu yük faktörü belli bir eşiği aşınca (Java'da 0,75) kova sayısını yaklaşık iki katına çıkarır ve öğeleri yeni kovalarına dağıtır. Böylece λ sabit bir sınırın altında kalır. Ortalama zincir boyu sabit olduğundan `add`, `contains` ve `remove` ortalama $O(1)$ sürer.

Bu ortalama, hash fonksiyonunun öğeleri kovalara iyi dağıttığı varsayımına dayanır. Kötü bir hash fonksiyonu bütün öğeleri aynı kovaya gönderirse tablo tek bir uzun zincire dönüşür. O durumda her işlem bütün zinciri tarar ve en kötü durum maliyeti $O(n)$ olur.

> **Sık yapılan hata**
>
> Bir sınıfta `equals`'ı ezip `hashCode`'u ezmemek. Bu durumda "eşit" iki nesnenin hash kodları büyük olasılıkla farklı olur ve nesneler farklı kovalara düşer. `HashSet` yalnız aynı kovadaki öğeleri `equals` ile karşılaştırdığı için iki nesneyi birbirinden haberdar olmadan saklar; küme aynı öğeyi **iki kez** içerir. Kural basittir: `equals` ezilirse `hashCode` da birlikte ezilir ve eşit nesneler eşit hash kodu üretir.

13.2.3 Kod izleme: add ve contains

Zincirlemeli hash kümesinin iki temel metodu kısa bir kodla yazılabilir. Alanlar şunlardır: `m` kova sayısı, `kova` her biri bir tam sayı listesi olan dizi, `n` kümedeki öğe sayısı. Kod, öğelerin negatif olmadığını varsayar; böylece kova numarası doğrudan `x % m` ile bulunur.

```
// alanlar: List<Integer>[] kova; n; m = 7
boolean add(int x) {          // x ≥ 0
  int i = x % m;               // kova no
  for (int y : kova[i])
    if (y == x) return false;  // zaten var
  kova[i].add(x);
  n++;
  return true;
}
boolean contains(int x) {
  int i = x % m;
  for (int y : kova[i])
    if (y == x) return true;
  return false;
}
```

`add(x)` önce kova numarasını hesaplar: `i = x % m`. Ardından o kovanın zincirinde baştan sona gezinir. Zincirde `x`'e eşit bir `y` bulursa öğe zaten kümededir; metot hiçbir şey eklemeden `false` döndürür. Zincir biterse `x` yenidir: zincirin sonuna eklenir, `n` bir artar ve metot `true` döndürür.

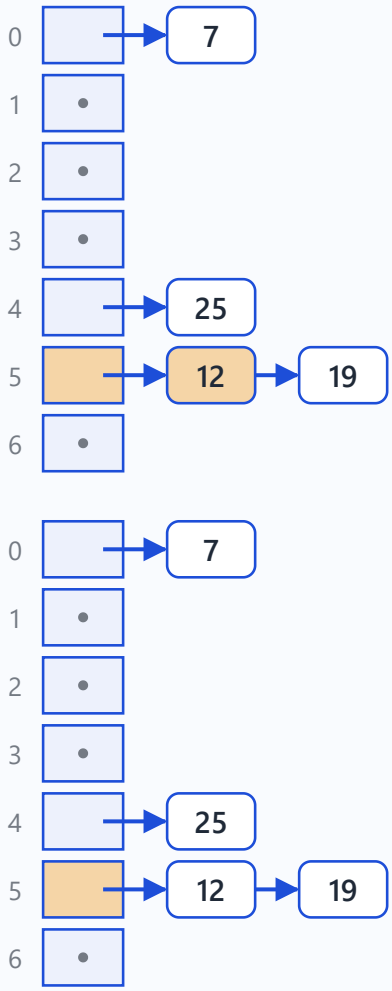
`contains(x)` aynı taramayı yapar; eşit öğe bulursa `true`, zincir biterse `false` döndürür. İki metot da yalnız tek bir kovanın zincirine bakar.

Çağrı	`i = x % 7`	Taranan	Dönüş	`n`	Dolu kovalar (sonra)
`add(12)`	5	—	`true`	1	5: 12
`add(25)`	4	—	`true`	2	4: 25 · 5: 12
`add(7)`	0	—	`true`	3	0: 7 · 4: 25 · 5: 12
`add(19)`	5	12	`true`	4	0: 7 · 4: 25 · 5: 12 → 19
`add(12)`	5	12 (eşit)	`false`	4	değişmez
`contains(26)`	5	12, 19	`false`	4	değişmez
`contains(19)`	5	12, 19 (eşit)	`true`	4	değişmez

Tablo 13.5: *m = 7* kovalı boş kümede çağrı dizisinin izlemesi. “Taranan” sütunu, zincirde `x` ile karşılaştırılan öğeleri sırasıyla verir.

İzlemede dikkat edilecek üç an vardır. İlk üç `add`'de kovalar boştur; döngü hiç dönmez ve öğe hemen eklenir. `add(19)`'da kova 5'te 12 bulunur. Döngü bir kez döner: *12 ≠ 19*, zincirde ilerlenir. Zincir bitince 19 sona eklenir ve kova 5'te artık iki öğe vardır. İkinci `add(12)`'de döngünün ilk adımı *y = 12 == 12* sonucunu verir; metot oradan `false` ile döner ve `n` 4'te kalır.

İki `contains` çağrısı başarılı ve başarısız aramanın farkını gösterir. `contains(26)` zincirin tamamını (iki öğe) tarar ve bulamaz. `contains(19)` de iki öğeye bakar, ama ikincisinde eşitliği bulur ve `true` döndürür. Her iki çağrıda da yalnız kova 5'e bakılmıştır.



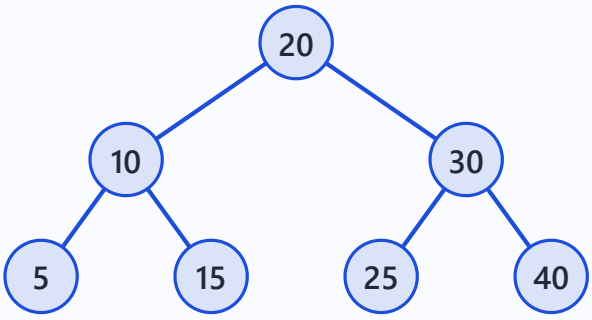
Şekil 13.5: İzlemeden iki an. (a) İkinci `add(12)` : kova 5'in ilk öğesi 12'dir, metot `false` döndürür ve hiçbir şey eklemez. (b) `contains(26)` : kova 5'teki zincir (12, 19) bitmiş, 26 bulunamamıştır; öteki kovalara hiç bakılmaz.

Maliyet. Kova numarasını hesaplamak sabit süredir. Zincir taraması zincirin uzunluğu kadar sürer; ortalama zincir uzunluğu yük faktörüne, yani *n / m*'ye eşittir. Yük faktörü sabit tutulduğu sürece iki metot da ortalama *O(1)*'dir. Bellek, *m* kova ile *n* zincir düğümü için *O(n + m)*'dir.

13.2.4 Ağaç tabanlı (sıralı) küme

Hash tablosu hızlıdır, ama öğelerin sırası hakkında hiçbir şey bilmez. “Kümedeki en küçük öğe hangisi?” ya da “17'den küçük en büyük öğe hangisi?” sorularını hash kümesi ancak bütün öğeleri tarayarak yanıtlar. Bu sorular sık soruluyorsa küme bir **dengeli ikili arama ağacıyla** gerçeklenir.

İkili arama ağacında her düğümün sol alt ağacındaki öğeler düğümden küçük, sağ alt ağacındakiler büyüktür. Bu düzen sayesinde arama kökten başlar ve her adımda sola ya da sağa inerek ağacın yarısını eler. Ağaç dengeli tutulursa yüksekliği *O(log n)* olur ve arama, ekleme, silme *O(log n)* sürer. Java'nın `TreeSet` sınıfı dengeyi **kırmızı-siyah ağaçla** (*red-black tree*) korur.



Şekil 13.6: *S = {5, 10, 15, 20, 25, 30, 40}* kümesini tutan dengeli ikili arama ağacı. Kök 20; sol alt ağaçta 10, 5, 15; sağ alt ağaçta 30, 25, 40. Kök ortada gezinme öğeleri sıralı verir.

Ağaçtaki öğelerin sıralı saklanması, iki önemli sonuç doğurur. Birincisi, **kök ortada** (*inorder*) gezinme (önce sol alt ağaç, sonra düğüm, sonra sağ alt ağaç) öğeleri küçükten büyüğe verir. Şekil 13.6'daki ağaçta bu sıra 5, 10, 15, 20, 25, 30, 40'tır. İkincisi, sıraya dayalı sorgular da *O(log n)* sürede yanıtlanır. `TreeSet` bu sorguları `NavigableSet` arayüzü üzerinden sunar.

İşlem	Anlamı	Sonuç
`first()` / `last()`	en küçük / en büyük öğe	5 / 40
`floor(17)`	≤ 17 olan en büyük öğe	15
`ceiling(17)`	≥ 17 olan en küçük öğe	20
`subSet(10, 30)`	aralık sorgusu: *10 ≤ x < 30*	[10, 15, 20, 25]

Tablo 13.6: Sıralı kümenin ek işlemleri, *S = {5, 10, 15, 20, 25, 30, 40}* üzerinde.

`floor(17)`'nin ağaçta nasıl bulunduğunu izleyelim. Kökte 20 vardır; *20 > 17* olduğu için cevap 20 olamaz, sola inilir. 10'da *10 ≤ 17*'dir; 10 bir aday olur ve daha büyük bir aday için sağa inilir. 15'te *15 ≤ 17*'dir; aday 15 olur, yine sağa inilir. 15'in sağ çocuğu yoktur; arama biter ve son aday 15 döner. Yol kökten yaprağa tek bir iniştir; ağacın yüksekliği kadar adım sürer. `ceiling(17)` aynı yolu ters kuralla yürür ve 20'yi bulur. Bir değer kümede varsa iki metot da onun kendisini döndürür; eşitlik dahildir.

`subSet(10, 30)`'da alt sınır dahil, üst sınır hariçtir; 30 sonuca girmez. Bu, Java'nın aralıklarda genel olarak izlediği yarı açık aralık kuralıdır. Aralık sorgusunun maliyeti, başlangıç noktasını bulmak için *O(log n)* ile çıktıdaki *k* öğe için *O(k)*'nin toplamıdır.

Ağaç tabanlı kümenin bir ön koşulu vardır: öğeler karşılaştırılabilir olmalıdır. Ya öğe sınıfı `Comparable` arayüzünü gerçekler ya da kümeye bir `Comparator` verilir. `TreeSet` iki öğenin eşit olup olmadığına `equals` ile değil, `compare(a, b) == 0` sonucuyla karar verir.

Sık yapılan hata

`equals` ile tutarsız bir sıralama vermek. Bir `TreeSet` kişileri yalnız soyadına göre karşılaştıran bir `Comparator` ile kurulmuş olsun. Kümeye "Ali Kaya" ve "Ayşe Kaya" eklenirse ikinci ekleme reddedilir. `compare` iki kişi için 0 döndürür ve ağaç onları aynı öğe sayar. Sıralama `equals` ile tutarlı olmalıdır; örneğin soyadından sonra ada göre de karşılaştırılmalıdır (`thenComparing`).

13.2.5 Sıralı dizilerle birleşim ve kesişim

Kümenin en sade gerçeklemesi, öğelerini küçükten büyüğe sıralı bir dizide tutmaktır. Üyelik testi ikili aramayla *O(log n)* sürer. Ekleme ve silme yavaştır, çünkü sırayı korumak için öğeler kaydırılır: *O(n)*. Buna karşılık iki sıralı küme arasında birleşim ve kesişim çok verimli hesaplanır. Yöntem, birleştirme sıralamasındaki birleştirme (*merge*) adımının aynısıdır: iki dizide birer imleç tutulur ve iki imleç tek geçişte ilerler. Goodrich, Tamassia ve Goldwasser bu yöntemi genel birleştirme (*generic merge*) adıyla anlatır.

```
void birlesimKesisim(int[] a, int[] b,
    List<Integer> bir, List<Integer> kes) {
  int i = 0, j = 0;
  while (i < a.length && j < b.length) {
    if (a[i] < b[j]) bir.add(a[i++]);
    else if (a[i] > b[j]) bir.add(b[j++]);
    else {                      // ortak öğe
      bir.add(a[i]);  kes.add(a[i]);
      i++;  j++;
    }
  }
  while (i < a.length) bir.add(a[i++]);
  while (j < b.length) bir.add(b[j++]);
}
```

Metot iki imleçle başlar: `i` dizi `a`'da, `j` dizi `b`'de, ikisi de 0'da. Ana döngü iki dizide de öğe kaldığı sürece döner ve her turda `a[i]` ile `b[j]`'yi karşılaştırır:

- `a[i] < b[j]` ise `a[i]` yalnız `a`'dadır. Neden? `b` sıralıdır ve `b[j]`'den sonraki öğeler daha büyüktür; `a[i]`'ye eşit bir öğe `b`'nin geri kalanında bulunamaz. `a[i]` birleşime eklenir ve `i` ilerler.
- `a[i] > b[j]` ise aynı gerekçeyle `b[j]` yalnız `b`'dedir; birleşime eklenir, `j` ilerler.
- İkisi eşitse öğe ortaktır. Birleşime *bir kez* ve kesişime eklenir; iki imleç birlikte ilerler.

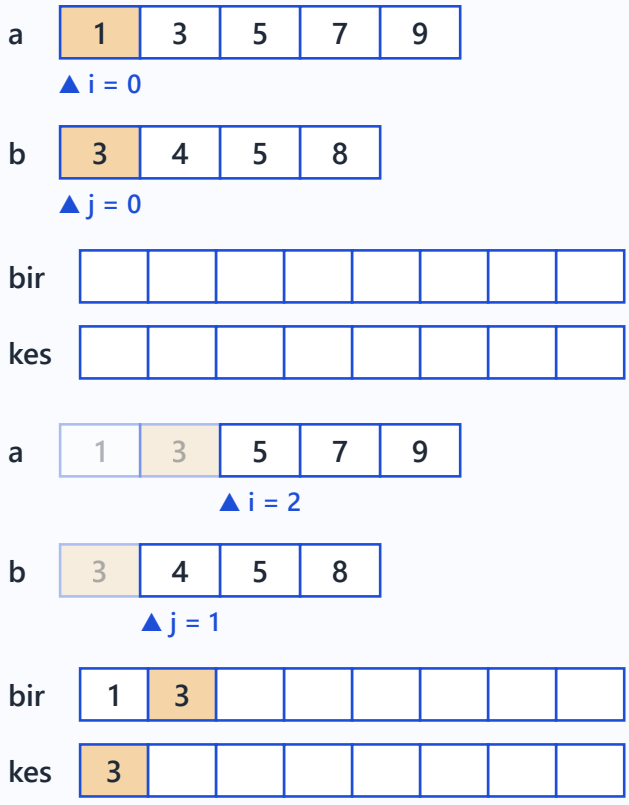
Dizilerden biri bitince ana döngü durur. Artık ortak öğe olamaz, bu yüzden kesişim tamamlanmıştır. Öteki dizide kalan öğeler yalnız o dizide bulunduğu için doğrudan birleşime eklenir. Son iki `while` döngüsünden en çok biri iş yapar.

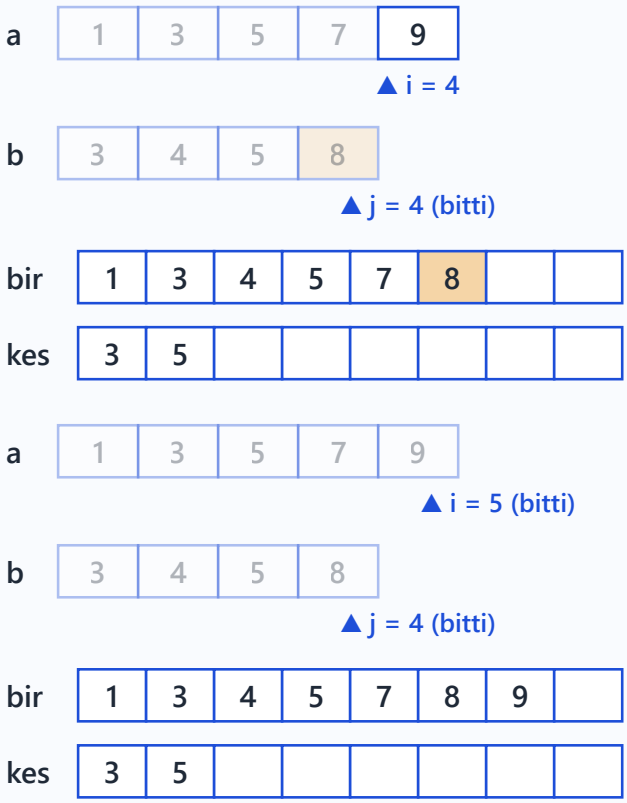
Metodu `a = [1, 3, 5, 7, 9]` ve `b = [3, 4, 5, 8]` ile izleyelim.

Adım	Karşılaştırma	Karar	`i`	`j`	`bir`	`kes`
1	1 < 3	1 yalnız a'da	1	0	[1]	[]
2	3 = 3	ortak	2	1	[1, 3]	[3]
3	5 > 4	4 yalnız b'de	2	2	[1, 3, 4]	[3]
4	5 = 5	ortak	3	3	[1, 3, 4, 5]	[3, 5]
5	7 < 8	7 yalnız a'da	4	3	[1, 3, 4, 5, 7]	[3, 5]
6	9 > 8	8 yalnız b'de	4	4	[1, 3, 4, 5, 7, 8]	[3, 5]
7	`j = 4 = b.length`	ana döngü biter	4	4	değişmez	[3, 5]
8	a'da kalan: 9	birleşime eklenir	5	4	[1, 3, 4, 5, 7, 8, 9]	[3, 5]

Tablo 13.7: `birlesimKesisim([1, 3, 5, 7, 9], [3, 4, 5, 8])` izlemesi. `i`, `j`, `bir` ve `kes` adımdan sonraki değerlerdir.

Ana döngü altı tur döner. 7. adımda `b` tükenmiştir; kesişim `[3, 5]` olarak kesinleşir. `a`'da kalan tek öğe 9'dur ve ilk ek döngü onu birleşime ekler. `b`'de kalan olmadığı için ikinci ek döngü hiç dönmez. Sonuç: birleşim `[1, 3, 4, 5, 7, 8, 9]`, kesişim `[3, 5]`. İki çıktı da kendiliğinden sıralıdır; böylece yeni kümeler de sıralı dizi olarak kullanılabilir.





Şekil 13.7: İki imleçli birleşim ve kesişimin dört anı. (a) Başlangıç: `i = j = 0`. (b) 3 ortak öğe: birleşime bir kez, kesişime de eklenmiş, iki imleç ilerlemiş. (c) 8 eklendikten sonra `j = 4`: `b` bitti. (d) Kalan 9 eklenmiş: birleşim [1, 3, 4, 5, 7, 8, 9], kesişim [3, 5].

Maliyet. Her turda en az bir imleç bir adım ilerler ve hiçbir imleç geri gitmez. $|a| = n$ ve $|b| = m$ ise toplam adım sayısı en çok $n + m$'dir; süre $O(n + m)$'dir. Sıralı olmayan iki listede aynı iş, her öğe için öteki listeyi taramayı gerektirir ve $O(n \cdot m)$ sürer. Fark ($A - B$) da aynı iskeletle bulunur: yalnız `a[i] < b[j]` durumundaki öğeler ve `a`'da kalanlar sonuca eklenir.

> **Sık yapılan hata**
>
> Eşitlik durumunda yalnız bir imleci ilerletmek. Ortak öğede yalnız `i++` yapılırsa bir sonraki turda `b[j]` yine aynı değerdir ve `a[i] > b[j]` dalı onu birleşime ikinci kez ekler. Ortak öğede iki imleç birlikte ilerlemelidir.

13.2.6 Gerçeklemelerin karşılaştırması

Sırasız liste ile üç gerçeklemenin maliyetleri Tablo 13.8'de yan yana verilmiştir. $|A| = n$, $|B| = m$'dir.

İşlem	Sırasız liste	Sıralı dizi	Hash tablosu	Dengeli BST
`contains`	O(n)	O(log n)	**O(1)***	O(log n)
`add` / `remove`	O(n)†	O(n)	**O(1)***	O(log n)

İşlem	Sırasız liste	Sıralı dizi	Hash tablosu	Dengeli BST
A ∪ B, A ∩ B, A − B	O(n · m)	O(n + m)	O(n + m)*	O(m log(n + m))
Sıralı gezinme	O(n log n)	**O(n)**	O(n log n)	**O(n)**
En küçük / `floor` / aralık	O(n)	O(log n)	O(n)	O(log n)

Tablo 13.8: Küme gerçeklemelerinin maliyetleri. * ortalama durum (en kötü O(n)). † tekrar denetimi için önce arama gerekir.

Tablonun satırlarını tek tek gerekçelendirelim. *Sırasız listede* üyelik için bütün liste taranır. Ekleme de tekrar denetimi gerektirdiği için önce bir arama yapar; bu yüzden *O(n)*'dir. Birleşimde B'nin her öğesi A'da aranır: *O(n · m)*. *Sıralı dizide* arama ikili aramayla *O(log n)*'dir, ama ekleme ve silme öğe kaydırdığı için *O(n)*'dir. Küme işlemleri iki imleçle *O(n + m)*'dir. *Hash tablosunda* tek öğe işlemleri ortalama sabittir. Birleşim için B'nin her öğesine `contains` + `add` uygulanır; kopyalamayla birlikte ortalama *O(n + m)* tutar. Hash tablosu sıra bilmediği için sıralı gezinme önce bir sıralama ister (*O(n log n)*). En küçük öğe ya da `floor` sorgusu bütün öğeleri tarar (*O(n)*). *Dengeli ağaçta* her tek öğe işlemi *O(log n)*'dir. Birleşimde B'nin *m* öğesi, en çok *n + m* öğeli ağaca tek tek eklenir: *O(m log(n + m))*. Kök ortada gezinme sıralı çıktıyı *O(n)*'de verir.

Pratikteki seçim üç cümleyle özetlenebilir. **Varsayılan seçim hash tablosudur:** sıra gerekmiyorsa en hızlı üyelik testini o verir. **Sıra gerekiyorsa ağaç seçilir:** her işlem *O(log n)*'dir ve sıralı sorgular ek bir bedel istemez. Sıralı dizi ise küme bir kez kurulup sonra yalnız sorgulanıyorsa ve sık sık birleşim ya da kesişim gerekiyorsa iyi bir seçimdir.

13.2.7 Bit vektörü ile küme

Kümenin öğeleri küçük bir tam sayı aralığından geliyorsa çok daha sıkı bir gösterim kullanılabilir. Bütün olası öğelerin oluşturduğu kümeye **evren** denir. Evren *{0, 1, …, U − 1}* ise küme *U* bitlik bir dizi olarak saklanır: *i*. bit 1 ise *i* kümededir, 0 ise değildir. Bu gösterime **bit vektörü** (*bit vector*) denir.



Şekil 13.8: Evren 0..7 üzerinde bit vektörleri. Bitler sağdan sola 0'dan 7'ye numaralanır. *A = {1, 3, 4}* için 1., 3. ve 4. bitler 1'dir; sayı değeri *2 + 8 + 16 = 26*. *B = {3, 5}* için değer *8 + 32 = 40*. Birleşim `A | B` = 58 → {1, 3, 4, 5}.

Evren en çok 32 öğeliyse bütün küme tek bir `int`'e sığar. Bu durumda her küme işlemi tek bir makine komutudur:

- **Ekleme:** `a |= 1 << x`. `1 << x`, yalnız *x*. biti 1 olan bir **maske** üretir; VEYA işlemi o biti 1 yapar, ötekilere dokunmaz.

- **Üyelik:** `(a >> x & 1) == 1`. Sağa kaydırma *x*. biti en sağa getirir, `& 1` yalnız o biti bırakır.
- **Silme:** `a &= ~(1 << x)`. Maskenin tersi *x*. bit dışında her yerde 1'dir; VE işlemi yalnız o biti sıfırlar.
- **Birleşim, kesişim, fark:** `a | b`, `a & b`, `a & ~b`. Bitlerin her biri ayrı bir öğeyi temsil ettiği için bit düzeyindeki VEYA, VE ve DEĞİL işlemleri küme işlemlerine birebir karşılık gelir.
- **Öğe sayısı:** `Integer.bitCount(a)`, 1 olan bitleri sayar.

Evren 32'den büyükse `java.util.BitSet` kullanılır. Bu sınıf bitleri bir `long` dizisinde tutar ve 64'er bitlik kelimeler üzerinde çalışır. Tek öğe işlemleri yine *O(1)*'dir; ilgili kelime ve bit doğrudan hesaplanır. Küme işlemleri kelime kelime yapılır ve *O(U / 64)* sürer.

İşlem	`int` maskesi	`java.util.BitSet`
ekle x / x var mı?	`a |= 1 << x` · `(a >> x & 1) == 1`	`set(x)` · `get(x)`
birleşim / kesişim	`a | b` · `a & b`	`or(b)` · `and(b)`
fark / öğe sayısı	`a & ~b` · `Integer.bitCount(a)`	`andNot(b)` · `cardinality()`

Tablo 13.9: Bit vektörüyle küme işlemleri: tek `int` maskesi ($U \le 32$) ve `java.util.BitSet`.

Bit vektörünün belleği evrenin boyutuna bağlıdır, kümedeki öğe sayısına değil: *U* bit. Bir milyon öğelik evren $10^6 / 8 = 125.000$ bayt, yani yaklaşık 122 KB tutar. Bu, bir milyon `Integer` nesnesi tutan bir `HashSet`'in kullanacağı birkaç on megabayta göre çok küçüktür. Ama evren büyük ve küme seyrekse hesap tersine döner. $U = 10^9$ evreninden yalnız 3 öğe tutmak için bit vektörü yaklaşık 119 MB ayırır; hash tabanlı küme ise birkaç yüz bayt kullanır. Bu durumda hash tabanlı küme seçilir.

> **Sık yapılan hata**
>
> `int` maskesinde 31'den büyük bit kullanmak. Java'da `1 << x` ifadesinde kaydırma miktarı 32'ye göre kalanla alınır: `1 << 32` 1'e, yani `1 << 0`'a eşittir. 32 eklenmek istenirken 0 eklenir ve hata sessizce oluşur. 64'e kadar `long` ve `1L << x`, daha büyük evrenlerde `BitSet` kullanılır.

13.2.8 Kod izleme: bit işlemleriyle küme cebri

Aşağıdaki kod *a = {1, 3, 4}* ve *b = {3, 5}* kümelerini bit vektörü olarak kurar, sonra birleşim, kesişim ve farkı hesaplar ve 3'ün *a*'da olup olmadığını sorar.

```
int a = 0, b = 0;
for (int x : new int[]{1, 3, 4})
  a |= 1 << x;          // x. biti yak
for (int x : new int[]{3, 5})
  b |= 1 << x;
int bir = a | b;        // birleşim
int kes = a & b;        // kesişim
int fark = a & ~b;      // fark
boolean var3 = (a >> 3 & 1) == 1;
```

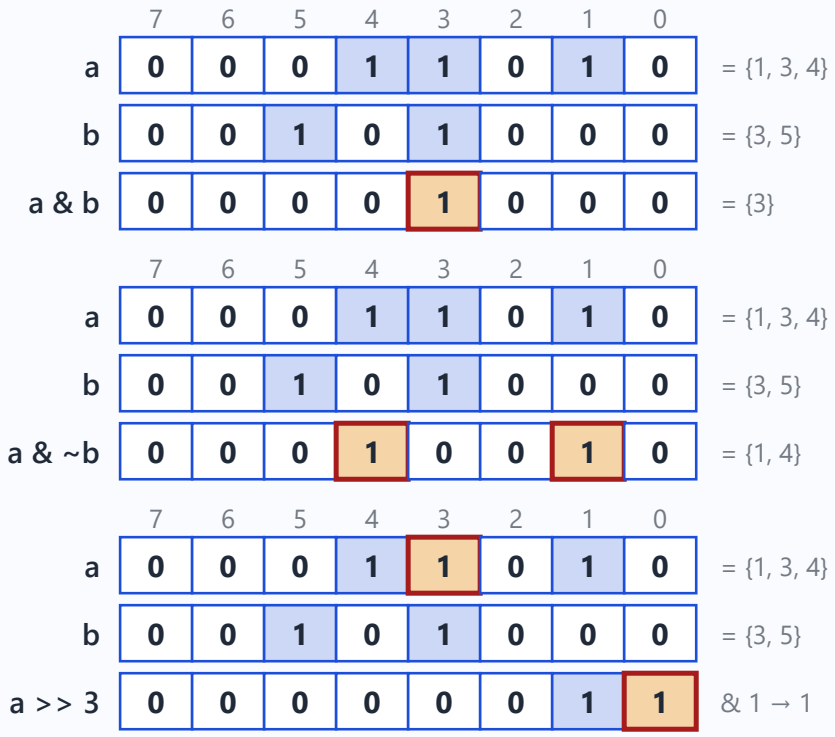
Satır	İşlem	Maske / ara değer	Sonuç (ikili)	Değer	Küme
1	`a = 0, b = 0`	—	0000 0000	0	{}
3	`a |= 1 << 1`	0000 0010 (2)	0000 0010	2	{1}
3	`a |= 1 << 3`	0000 1000 (8)	0000 1010	10	{1, 3}
3	`a |= 1 << 4`	0001 0000 (16)	0001 1010	26	{1, 3, 4}
5	`b |= 1 << 3`	0000 1000 (8)	0000 1000	8	{3}
5	`b |= 1 << 5`	0010 0000 (32)	0010 1000	40	{3, 5}
6	`bir = a | b`	—	0011 1010	58	{1, 3, 4, 5}
7	`kes = a & b`	—	0000 1000	8	{3}
8	`fark = a & ~b`	`~b` = 1101 0111 (son 8 bit)	0001 0010	18	{1, 4}
9	`(a >> 3 & 1) == 1`	`a >> 3` = 0000 0011	1	`true`	3 ∈ a

Tablo 13.10: Bit işlemleri izlemesi. Bitler 7'den 0'a doğru yazılmıştır.

İzlemenin ilk altı satırı küme kurmaktır. Her turda `1 << x` yalnız *x*. biti yanık bir maske üretir: `1 << 1 = 2`, `1 << 3 = 8`, `1 << 4 = 16`. VEYA ile bu bitler `a`'ya eklenir; `a`'nın değeri 2, 10, 26 olur. Aynı yolla `b = 8 + 32 = 40` olur.

Birleşimde bitler tek tek VEYA'lanır: herhangi birinde 1 olan bit sonuçta 1'dir. 1., 3., 4. ve 5. bitler yanar; değer 58, küme {1, 3, 4, 5}'tir. Kesişimde VE kullanılır: yalnız iki sayıda da 1 olan bit kalır. Bu tek bit 3. bittir; değer 8, küme {3}'tür. Fark iki adımdır. `~b`, `b`'nin bütün bitlerini tersine çevirir; artık 3. ve 5. bitler 0, ötekilerin hepsi 1'dir. `a & ~b`, `a`'nın bitlerinden yalnız `b`'de olmayanları bırakır: 1. ve 4. bitler. Değer 18, küme {1, 4}'tür.

Son satır üyelik testidir. `a = 0001 1010` üç kez sağa kaydırılınca `0000 0011` olur; 3. bit en sağa gelmiştir. `& 1` bu biti ayırır ve sonuç 1'dir: 3 kümededir. Bütün bu işlemler döngüsüz, tek komutluk işlemlerdir; her biri *O(1)*'dir.



Şekil 13.9: Bit işlemlerinin üç anı (yukarıdan aşağı). Kesişim `a & b` : yalnız 3. bit iki sayıda da 1'dir → {3}. Fark `a & ~b` : `a` 'nın bitlerinden `b` 'de olmayanlar kalır → {1, 4}. Üyelik: `a >> 3` 3. biti en sağa getirir, `& 1` onu ayırır → 1, yani $3 \in a$.

13.3 Çoklu küme ve çoklu eşleme

Kümenin tekrar yasağı çoğu problemde işe yarar, ama bazı problemlerde tam olarak tekrarlar önemlidir. Bir metinde "ve" kelimesinin kaç kez geçtiği, bir sepette kaç elma olduğu, bir seçimde bir adayın kaç oy aldığı birer sayma sorusudur. Bu sorular için kümenin bir uzantısı kullanılır.

13.3.1 Çoklu küme

> **Tanım 13.2**
>
> **Çoklu küme** (*multiset, bag*), aynı öğenin birden çok kez bulunabildiği bir kümedir. Öğelerin sırası yine önemsizdir; önemli olan her öğenin kaç kez geçtiğidir. Bir öğenin geçme sayısına o öğenin **çokluğu** (*multiplicity*) denir.

Çoklu küme $M = \{a, a, b, c, c, c\}$ düşünelim. Burada a'nın çokluğu 2, b'ninki 1, c'ninki 3'tür. Öğeleri tek tek saklamak yerine her farklı öğeyi bir kez yazıp yanına çokluğunu koymak aynı bilgiyi taşır: $\{a{:}2, b{:}1, c{:}3\}$. Bu, çoklu kümenin doğal gerçeklemesidir: **öğe → sayaç** eşlemesi. Eşleme bir hash tablosu olduğunda her işlem tek bir anahtar erişimine iner.

İşlem	Anlamı	Hash ile
`add(e)` / `add(e, n)`	e'yi 1 / n kez ekle (sayaç += n)	O(1)*
`count(e)`	e kaç kez var? (yoksa 0)	O(1)*
`remove(e)` / `remove(e, n)`	sayacı azalt; 0 olursa girdiyi sil	O(1)*
`size()`	tekrarlar dahil öğe sayısı (*M* için 6)	O(1)

Tablo 13.11: Çoklu küme ADT'si (Goodrich, Tamassia ve Goldwasser) ve hash tabanlı eşlemeyle maliyetleri. * ortalama durum.

Tablodaki `size()`'ın anlamına dikkat etmek gerekir. *M*'nin *farklı* öğe sayısı 3'tür, ama `size()` tekrarları da sayar ve 6 döndürür. Bu değeri her seferinde sayaçları toplayarak bulmak *O(farklı öğe sayısı)* sürerdi. Bunun yerine gerçekleme ayrı bir `n` sayacı tutar; her ekleme onu artırır, her silme azaltır. Böylece `size()` *O(1)* olur.

Çoklu küme pek çok yerde işe yarar. Metindeki kelime sıklıkları, alışveriş sepetindeki ürünler ve oy sayımı bunların en bilinenleridir. İki kelimenin aynı harflerden oluşup oluşmadığı da çoklu kümeyle sınanır: iki kelimenin harf sayaçları karşılaştırılır.

13.3.2 Java ile çoklu küme

Java'nın standart kütüphanesinde hazır bir çoklu küme sınıfı yoktur. Bir `HashMap<E, Integer>` ile kolayca kurulur:

```
class CokluKume<E> {
    private final Map<E, Integer> say = new HashMap<>();
    private int n = 0;                                  // tekrarlar dahil öğe sayısı
    void add(E e) { say.merge(e, 1, Integer::sum); n++; }  // yoksa 1, varsa +1
    int count(E e) { return say.getOrDefault(e, 0); }
    boolean remove(E e) {
        Integer c = say.get(e);
        if (c == null) return false;                    // e yok
        if (c == 1) say.remove(e);                         // sayaç 0 olacak: girdiyi sil
        else say.put(e, c - 1);
        n--;
        return true;
    }
    int size() { return n; }
}
```

Sınıfın iki alanı vardır. `say`, her farklı öğeyi sayacına eşler; `n`, tekrarlar dahil öğe sayısıdır. Metotları sırayla okuyalım.

- `add(e)` tek satırdır: `say.merge(e, 1, Integer::sum)`. `merge` metodu şöyle çalışır: `e` eşlemede yoksa onu verilen değerle (1) ekler. Varsa eski değerle yeni değeri verilen fonksiyonla birleştirir; burada fonksiyon toplamadır ve sayaç bir artar. Ardından `n` artırılır.

- `count(e)`, `getOrDefault(e, 0)` ile sayacı okur. Öğe eşlemede yoksa `null` yerine 0 döner. Bu, çoklu kümenin anlamına uygundur: bulunmayan öğe 0 kez vardır.
- `remove(e)` önce sayacı okur. `null` ise öğe yoktur; metot `false` döndürür ve hiçbir şey değişmez. Sayaç 1 ise silme sonrası 0 olacaktır; girdi eşlemeden tamamen silinir. Sayaç 1'den büyükse bir azaltılır. Her iki durumda `n` azalır ve metot `true` döndürür.
- `size()` doğrudan `n`'yi döndürür.

Kümeyle farkı tek bir örnek açıkça gösterir. `add("a"); add("a");` çağrılarından sonra `count("a")` 2 döndürür. Aynı iki çağrı bir `Set`'e yapılsaydı ikinci `add` etkisiz kalırdı; küme yalnız "a var" bilgisini tutardı. Sıralı gezinme gerekiyorsa `HashMap` yerine `TreeMap<E, Integer>` kullanılır; işlemler *O(log n)*'e çıkar, ama öğeler sıralı gezilir.

> **Sık yapılan hata**
>
> Sayacı 0'a düşen girdiyi eşlemede bırakmak. `remove`'da her zaman `say.put(e, c - 1)` yazılırsa sayacı 0 olan girdiler birikir. Daha kötüsü, `c == null` denetimi artık "öğe yok" durumunu yakalayamaz. 0 sayaçlı bir öğe yeniden silinirse sayaç −1 olur. Metot `true` döndürür ve `n` yanlış azalır. Çoklu kümede 0 sayaç saklanmaz; sayaç 0'a inen girdi silinir.

13.3.3 Kod izleme: CokluKume sayaçları

Boş bir `CokluKume<String>` üzerinde şu çağrıları izleyelim: `add("a")`, `add("b")`, `add("a")`, `add("c")`, `count("a")`, `remove("a")`, `remove("b")`, `remove("x")`, `count("b")`. İzlemede kullanılan metot gövdeleri şunlardır:

```
void add(E e) {
  say.merge(e, 1, Integer::sum);  n++;
}
int count(E e) {
  return say.getOrDefault(e, 0);
}
boolean remove(E e) {
  Integer c = say.get(e);
  if (c == null) return false;
  if (c == 1) say.remove(e);
  else say.put(e, c - 1);
  n--;
  return true;
}
```

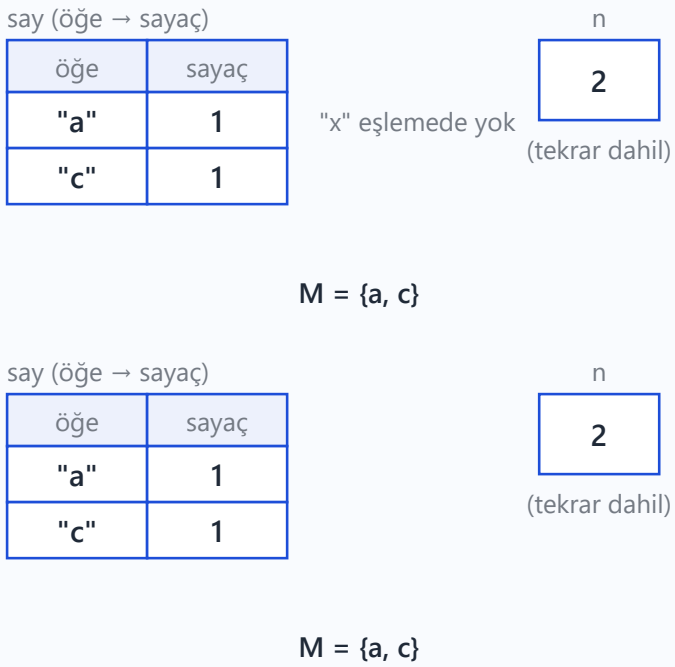
Çağrı	Ne olur?	Dönüş	`say`	`n`
`add("a")`	"a" yoktu → sayaç 1 ile eklenir	—	{a:1}	1
`add("b")`	"b" yoktu → sayaç 1	—	{a:1, b:1}	2

Çağrı	Ne olur?	Dönüş	`say`	`n`
`add("a")`	"a" vardı → *1 + 1 = 2*	—	{a:2, b:1}	3
`add("c")`	"c" yoktu → sayaç 1	—	{a:2, b:1, c:1}	4
`count("a")`	`getOrDefault` → 2	2	değişmez	4
`remove("a")`	`c = 2 > 1` → sayaç 1'e iner	`true`	{a:1, b:1, c:1}	3
`remove("b")`	`c == 1` → girdi silinir	`true`	{a:1, c:1}	2
`remove("x")`	`c == null` → öğe yok	`false`	değişmez	2
`count("b")`	"b" eşlemede yok → 0	0	değişmez	2

Tablo 13.12: `CokluKume` izlemesi. `say` ve `n` çağrıdan sonraki değerlerdir.

İzlemenin üçüncü satırı kümeyle farkı gösterir. "a" ikinci kez eklendiğinde `merge` eski sayaç 1 ile yeni değer 1'i toplar; sayaç 2 olur ve `n` 3'e çıkar. Bir `Set`'te bu çağrı kümeyi değiştirmezdi. İki `remove` çağrısı ise iki dalı ayrı ayrı çalıştırır. "a"nın sayacı 2'dir; `else` dalı onu 1'e indirir ve girdi eşlemede kalır. "b"nin sayacı 1'dir; `if (c == 1)` dalı girdiyi tamamen siler. Var olmayan "x" için `say.get` `null` döndürür ve metot ilk denetimde `false` ile çıkar; `n` değişmez. Son olarak `count("b")` 0 döndürür: silinmiş öğe ile hiç eklenmemiş öğe aynı sonucu verir. Sonuçta *M = {a, c}*, `n = 2`'dir.





Şekil 13.10: `CokluKume` izlemesinden dört an. (a) "a" ikinci kez eklenmiş: sayaç 2, `n = 3`. (b) `remove("b")`: sayaç 1 olduğu için girdi silindi (0 sayaç saklanmaz), `n = 2`. (c) `remove("x")`: "x" eşlemede yok, `false` döner. (d) Sonuç $M = \{a, c\}$, `n = 2`.

Maliyet. Her metot eşlemeye bir kez (`remove`'da en çok iki kez) erişir. Hash tabanlı eşlemede bu erişimler ortalama $O(1)$'dir; bu yüzden bütün işlemler ortalama sabit süredir. Bellek, farklı öğe sayısı d ile orantılıdır: $O(d)$. Tekrarlar ayrı ayrı saklanmadığı için milyonlarca kez geçen bir kelime de tek bir girdi tutar.

Örnek 13.2

"kavak" kelimesinin harfleri sırayla bir `CokluKume<Character>`'e ekleniyor. `count('a')` ve `size()` ne döndürür?

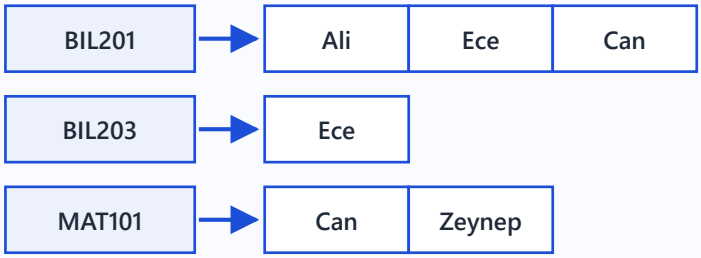
1. **k:** yoktu, sayaç 1. Eşleme {k:1}, `n = 1`.
2. **a:** yoktu, sayaç 1. {k:1, a:1}, `n = 2`.
3. **v:** yoktu. {k:1, a:1, v:1}, `n = 3`.
4. **a:** vardı, *1 + 1 = 2*. {k:1, a:2, v:1}, `n = 4`.
5. **k:** vardı, sayaç 2. {k:2, a:2, v:1}, `n = 5`.
6. **Sonuç.** `count('a')` 2, `size()` 5 döndürür. Farklı harf sayısı ise eşlemenin boyutudur: 3.

13.3.4 Çoklu eşleme

Çoklu küme bir öğenin *kaç kez* geçtiğini tutar. Bazen sayı yetmez; tekrarların kendisi de gerekir. Bir dersi alan öğrencilerin *kimler* olduğunu bilmek istiyorsak her ders anahtarına bir öğrenci listesi bağlanmalıdır.

Tanım 13.3

Çoklu eşleme (*multimap*), bir anahtarın birden çok değere eşlenebildiği eşlemedir. Goodrich, Tamassia ve Goldwasser'in ADT'sinde `get(k)` anahtarın bütün değerlerini döndürür. `put(k, v)` yeni bir (anahtar, değer) çifti ekler. `remove(k, v)` tek bir çifti, `removeAll(k)` anahtarın bütün çiftlerini siler. `size()` bütün (anahtar, değer) çiftlerinin sayısını verir.



Şekil 13.11: Ders → öğrenciler çoklu eşlemesi. BIL201 üç öğrenciye (Ali, Ece, Can), BIL203 bir öğrenciye (Ece), MAT101 iki öğrenciye (Can, Zeynep) eşlenmiştir. Ece ve Can birden çok anahtarın değer listesinde bulunur.

Java'da hazır bir çoklu eşleme sınıfı yoktur. Yapı, her anahtarı bir değer listesine eşleyen bir `Map<K, List<V>>` ile kurulur:

```
Map<String, List<String>> m = new HashMap<>();
void put(String k, String v) {   // yoksa boş liste aç
  m.computeIfAbsent(k, x -> new ArrayList<>()).add(v);
}
List<String> get(String k) {
  return m.getOrDefault(k, List.of());
}
```

`put` metodundaki `computeIfAbsent(k, x -> new ArrayList<>())` çağrısı iki durumu tek satırda çözer. Anahtar eşlemede yoksa yeni bir boş liste oluşturur, eşlemeye koyar ve onu döndürür. Anahtar varsa mevcut listeyi döndürür. Her iki durumda dönen listeye `add(v)` uygulanır. `get` ise anahtar yoksa `null` yerine değiştirilemez boş bir liste (`List.of()`) döndürür; çağıran kod `null` denetimi yapmak zorunda kalmaz.

Şekil 13.11'deki eşleme altı `put` çağrısıyla kurulur: (BIL201, Ali), (BIL201, Ece), (BIL201, Can), (BIL203, Ece), (MAT101, Can), (MAT101, Zeynep). `get("BIL201")` `[Ali, Ece, Can]` döndürür; `get("FIZ101")` boş liste döndürür. Çoklu eşlemenin boyutu çift sayısıdır: 6. Anahtar sayısı ise 3'tür.

Maliyet. Hash tabanlı eşlemede `put` ve `get` ortalama *O(1)*'dir; `get` listeyi kopyalamadan döndürür. `remove(k, v)` önce listeyi bulur (*O(1)*), sonra listede `v`'yi arar; bu, liste boyu kadar sürer: *O(1 + liste boyu)*. Çoklu kümeyle farkı özetlersek: çoklu küme yalnız kaç kez geçtiğini, çoklu eşleme değerlerin kendisini tutar.

13.4 Ayrık kümeler ve Union-Find

Şimdiye kadar tek bir kümeyle ya da iki kümenin birleşimiyle çalıştık. Bazı problemlerde ise elimizde çok sayıda küme vardır ve bu kümeler zamanla birleşir. Bir sosyal ağda arkadaş grupları, bir haritada birbirine yolla bağlı şehirler, bir resimde aynı renkteki bitişik pikseller böyle problemlerdir. Hepsinde iki soru sorulur: "Şu iki öğe aynı grupta mı?" ve "Şu iki grubu birleştir." Bu kesim, bu iki soruyu çok hızlı yanıtlayan yapıyı kurar.

13.4.1 Ayrık kümeler

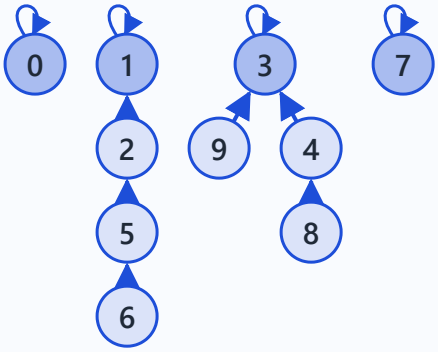
Tanım 13.4

Ayrık kümeler (*disjoint sets*), hiçbir öğesi ortak olmayan kümelerden oluşan bir topluluktur. Topluluktaki her öğe tam olarak *bir* kümeye aittir. Bu kümeleri tutan veri yapısına **Union-Find** yapısı (*Disjoint Set Union*, kısaca DSU) denir. Yapının iki temel işlemi vardır:

- `find(x)` (bul): x'in ait olduğu kümenin **temsilcisini** döndürür. İki öğe aynı kümedeyse ve yalnız o zaman `find(x) == find(y)`'dir.
- `union(x, y)` (birleştir): x'in kümesiyle y'nin kümesini tek bir kümede birleştirir. Örneğin *{1, 2}* ve *{3}* birleşince *{1, 2, 3}* olur.

Temsilci, kümenin "adı" gibidir. Kümenin hangi öğesinin temsilci seçildiği önemli değildir. Önemli olan, aynı kümedeki bütün öğeler için `find`'ın *aynı* temsilciyi döndürmesidir. Böylece "x ile y aynı kümede mi?" sorusu iki `find` çağrısı ve bir karşılaştırmayla yanıtlanır.

Union-Find yapısı her kümeyi bir **ağaç** olarak saklar. Ağaçtaki her düğüm yalnız kendi **ebeveynini** (*parent*) gösterir; oklar çocuktan ataya doğrudur. Kök kendini gösterir ve kümenin temsilcisidir. Birden çok ağaçtan oluşan bu yapıya **orman** denir.



Şekil 13.12: Dört ayrık küme, ağaç olarak. {0} ve {7} tek düğümlük kümelerdir. {1, 2, 5, 6} bir zincirdir: 6 → 5 → 2 → 1. {3, 9, 4, 8} kümesinde 9 ve 4, 3'ü; 8, 4'ü gösterir. Her kök kendini gösterir (döngüsel ok) ve kümesinin temsilcisidir.

Şekil 13.12 bu gösterimi somutlaştırır. `find(6)`, 6'dan başlayıp okları izler: 6 → 5 → 2 → 1. 1 kendini gösterdiği için köktür ve sonuç 1'dir. `find(8)` ise 8 → 4 → 3 yolunu izler ve 3'ü döndürür. *1 ≠ 3* olduğu için 6 ile 8 farklı kümelerdedir. Bu iki kümeyi birleştirmek için tek bir ok eklemek yeter: köklerden biri ötekini gösterecek biçimde değiştirilir, örneğin 1'in ebeveyni 3 yapılır.

Ağaçlar burada arama ağacı değildir. Düğümler arasında "küçük–büyük" düzeni yoktur, bir düğümün kaç çocuğu olacağı da sınırlı değildir. Ağacın tek görevi, aynı kümedeki öğeleri ortak bir köke bağlamaktır.

13.4.2 Mıknatıs örneği

Masanın üzerine dağılmış 14 güçlü mıknatıs düşünelim. İki mıknatıs birbirine değdiğinde yapışır ve bir daha ayrılmaz. Birbirine yapışmış mıknatıslar bir grup oluşturur; bir mıknatıs başka bir grubun herhangi bir üyesine değerse iki grup bütünüyle birleşir. Birleşmeler sürerken her an şu soruyu yanıtlamak istiyoruz: "Şu iki mıknatıs aynı grupta mı?"

Önce her mıknatısa 1'den 14'e bir numara verilir. Başlangıçta her mıknatıs kendi grubundadır; 14 ayrık küme vardır ve her *x* için `find(x) = x`'tir. İki mıknatısın değmesi bir `union`, "aynı grupta mı?" sorusu iki `find` çağrısının karşılaştırılmasıdır.

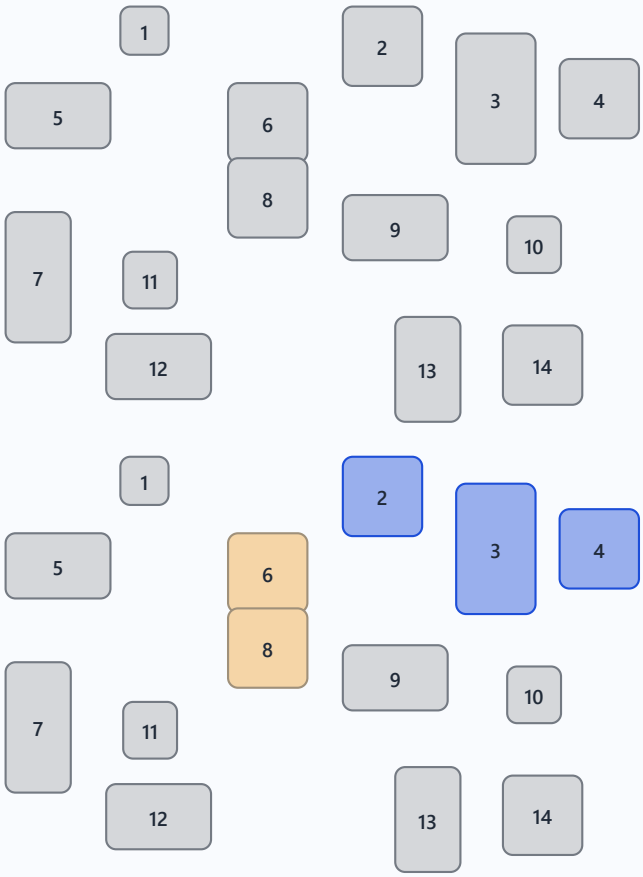
Birleşme	Gruplar (sonra)	Grup sayısı
başlangıç	her mıknatıs tek başına	14
`union(6, 8)`	{6, 8}	13
`union(3, 4)`	{6, 8} · {3, 4}	12
`union(2, 3)`	{6, 8} · {2, 3, 4}	11
`union(9, 10)`, `union(13, 14)`, `union(9, 13)`	{6, 8} · {2, 3, 4} · {9, 10, 13, 14}	8
`union(5, 7)`, `union(7, 11)`, `union(11, 12)`	{6, 8} · {2, 3, 4} · {9, 10, 13, 14} · {5, 7, 11, 12}	5

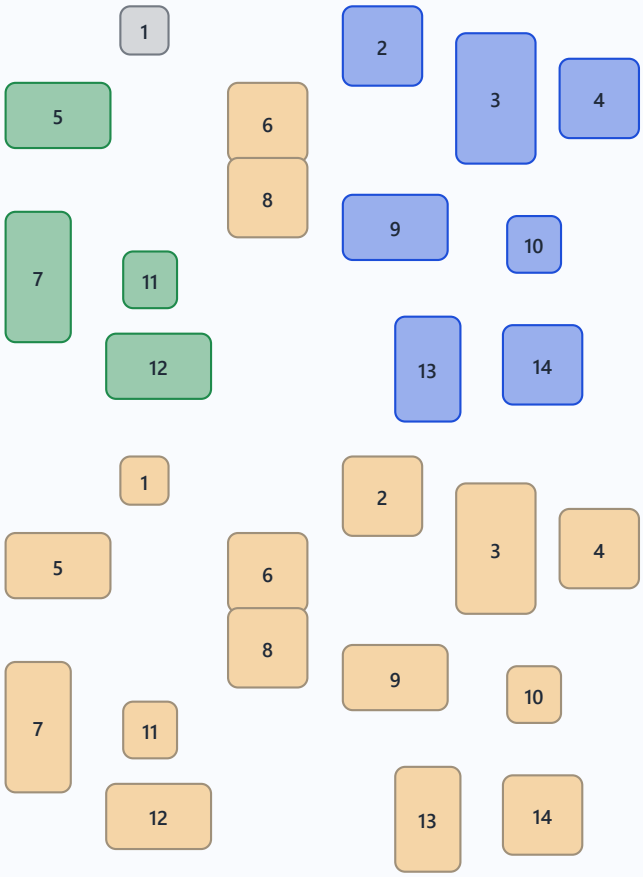
Birleşme	Gruplar (sonra)	Grup sayısı
`union(4, 10)`	{6, 8} · {2, 3, 4, 9, 10, 13, 14} · {5, 7, 11, 12}	4

Tablo 13.13: Mıknatıs örneğinde birleşmeler ve gruplar. Tek başına kalan mıknatıslar yazılmamıştır.

Her başarılı birleşme grup sayısını tam olarak bir azaltır. İlk birleşmede 6 ile 8 yapışır; 13 grup kalır. `union(2, 3)`'te 2 tek başınadır, 3 ise {3, 4} grubundadır; 2 bu gruba katılır ve grup {2, 3, 4} olur. Üç birleşmeden sonra {9, 10, 13, 14}, üç birleşme daha sonra {5, 7, 11, 12} oluşur. Artık beş grup vardır: dört çok üyeli grup ve tek başına kalan 1.

Ardından soru sorulur: 4 ile 10 aynı grupta mı? `find(4)`, {2, 3, 4} grubunun temsilcisini; `find(10)`, {9, 10, 13, 14} grubunun temsilcisini döndürür. İkisi farklıdır, yanıt "hayır"dır. `union(4, 10)` bu iki grubu tek grupta birleştirir. Birleşmeler sürerse sonunda bütün mıknatıslar tek grupta toplanır ve `find` herkes için aynı kökü verir.





Şekil 13.13: Mıknatıs örneğinin dört anı; aynı renkteki mıknatıslar aynı gruptadır. (a) 14 mıknatıs, her biri kendi grubunda. (b) {6, 8} ve {2, 3, 4} oluşmuş. (c) `union(4, 10)` iki grubu birleştirmiş. (d) Bütün mıknatıslar tek grupta.

Örneğin önemli bir dersi vardır: Union-Find, grupların *nasıl* oluştuğunu (kimin kime değdiğini) hatırlamaz; yalnız kimin hangi grupta olduğunu bilir. 2'nin 3'e mi 4'e mi değdiği sonradan sorulamaz. Bu bilgiyi atmak, yapının bu kadar hızlı olmasının nedenlerinden biridir.

13.4.3 find ve union işlemleri

İki işlemi biraz daha kesin tanımlayalım.

`find(x)`, x'i içeren kümenin adını, yani ağacının kökünü döndürür. Bunun için ebeveyn bağlantıları kök bulunana kadar izlenir. Bu işlem arama ağaçlarındaki aramayla karıştırılmamalıdır. Arama ağacında bir *değer* aranır ve kökten aşağı inilir. `find`'da hiçbir değer aranmaz; düğüm zaten bellidir ve yukarı doğru köke çıkılır.

`union(x, y)`, iki kümenin yerine ikisinin birleşimi olan tek bir küme koyar. Bunu yapmak için iki kökten biri ötekinin altına bağlanır. Kümeler baştan ayrık olduğu için birleşimde tekrar denetimi gerekmez; hiçbir öğe kopyalanmaz. Yalnız tek bir ebeveyn bağlantısı değişir. 13.1'deki küme birleşimiyle bu fark dikkat çekicidir: orada birleşim $O(n + m)$ sürüyordu, burada (kökler bulunduktan sonra) $O(1)$'dir.

İki işlemin sözde kodu şöyledir:

```
find(x):
  if x.ebeveyn != x:
    x.ebeveyn = find(x.ebeveyn)  // yol kısaltma
  return x.ebeveyn

union(x, y):
  kök1 = find(x)
  kök2 = find(y)
  if kök1 != kök2:
    // bir kümenin kökünü diğer
    // kümenin köküne ebeveyn
    kök1.ebeveyn = kök2
```

`find` özyinelemelidir. Düğüm kendi ebeveyni değilse kök değildir; ebeveyninin kökü bulunur. Sözde koddaki atama (`x.ebeveyn = find(x.ebeveyn)`) yalnız kökü bulmakla kalmaz, bulunan kökü x'in yeni ebeveyni yapar. Bu ek işe **yol kısaltma** denir ve 13.5.4'te ayrıntısıyla ele alınır. `union` önce iki öğenin köklerini bulur. Kökler farklıysa birinci kökün ebeveyni ikinci kök yapılır; aynıysa öğeler zaten aynı kümededir ve hiçbir şey yapılmaz.

> **Sık yapılan hata**
>
> Kökler yerine öğelerin kendisini bağlamak. `union(x, y)` 'de `ebeveyn[x] = y` yazmak yanlıştır. x kök değilse eski ebeveyn bağlantısı kaybolur ve x'in ağacı bölünür. x ile alt ağacı y'nin kümesine taşınır; x'in eski ataları ise eski kümede kalır. Doğrusu iki kökü bağlamaktır: `ebeveyn[find(x)] = find(y)`.

13.4.4 Nesneleri indislere eşleme

Union-Find en verimli hâliyle bir tam sayı dizisi üzerinde çalışır. Öğeler *0..n − 1* arasındaki tam sayılardır ve `ebeveyn[i]`, *i* öğesinin ebeveyninin indisini tutar. Dizi erişimi sabit süreli olduğundan ebeveyne çıkmak tek bir okumadır.

Gerçek problemlerde öğeler çoğu zaman harf, şehir ya da kişidir. Bu durumda her nesneye önce bir indis verilir. İndisler rastgele seçilebilir, ama bir kez verildikten sonra değişmemelidir. Nesne → indis eşlemesi bir hash tablosunda saklanır. Ardından her öğe için `ebeveyn[i] = i` yazılır; her öğe kendi tek öğeli kümesinin kökü olarak başlar. Bu başlangıç işlemine `makeSet` denir.

Harf	E	F	I	D	C	A	J	L	G	K	B	H
İndis	0	1	2	3	4	5	6	7	8	9	10	11

Tablo 13.14: On iki harfin indislere eşlenmesi.

```
Map<Character, Integer> indis = new HashMap<>();   // 'E'→0, 'F'→1, …
int[] ebeveyn = new int[12];
for (int i = 0; i < 12; i++) ebeveyn[i] = i;             // 12 tekil küme
```

Bu kodun ilk satırı harf → indis eşlemesini tutar. İkinci ve üçüncü satır 12 hücreli ebeveyn dizisini kurar ve her hücreye kendi indisini yazar: 12 tekil küme. Harflerle çalışan bir `union('C', 'K')` çağrısı önce eşlemeden indisleri bulur (4 ve 9), sonra diziyle çalışır. Eşleme ortalama *O(1)* olduğu için bu çeviri işlemlerin maliyetini değiştirmez. Başlangıç *O(n)* sürer.

13.4.5 union işlemi adım adım

Tablo 13.14'teki 12 harf üzerinde bir dizi `union` çağrısını izleyelim. Bu izlemede yol kısaltma kullanılmaz. Birleştirme kuralı şudur: **küçük ağacın kökü büyük ağacın köküne bağlanır; boyutlar eşitse ikinci kök birincinin altına bağlanır.** Ağacın boyutu, içindeki öğe sayısıdır. Bu kuralın neden seçildiği 13.5.2'de açıklanır; burada yalnız uygulayacağız. Her kökte ağacının boyutu tutulur.

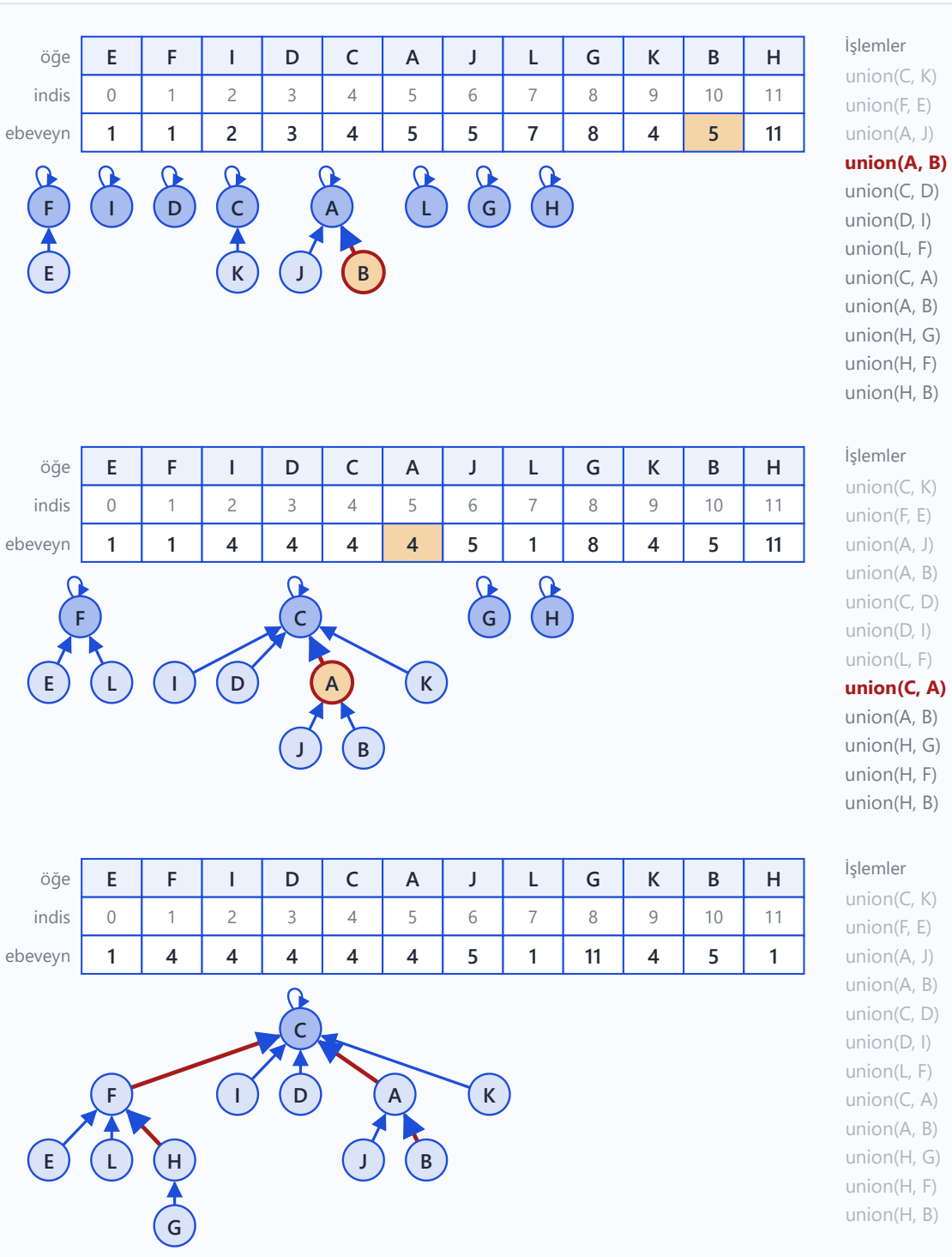
Çağrı	Kökler	Boyutlar	Bağlanan	Değişen hücre
`union(C, K)`	C, K	1, 1 (eşit)	K → C	`ebeveyn[9] = 4`
`union(F, E)`	F, E	1, 1 (eşit)	E → F	`ebeveyn[0] = 1`
`union(A, J)`	A, J	1, 1 (eşit)	J → A	`ebeveyn[6] = 5`
`union(A, B)`	A, B	2, 1	B → A	`ebeveyn[10] = 5`
`union(C, D)`	C, D	2, 1	D → C	`ebeveyn[3] = 4`
`union(D, I)`	C, I	3, 1	I → C	`ebeveyn[2] = 4`
`union(L, F)`	L, F	1, 2	L → F	`ebeveyn[7] = 1`
`union(C, A)`	C, A	4, 3	A → C	`ebeveyn[5] = 4`
`union(A, B)`	C, C	—	zaten aynı küme	yok
`union(H, G)`	H, G	1, 1 (eşit)	G → H	`ebeveyn[8] = 11`
`union(H, F)`	H, F	2, 3	H → F	`ebeveyn[11] = 1`
`union(H, B)`	F, C	5, 7	F → C	`ebeveyn[1] = 4`

Tablo 13.15: Boyuta göre birleştirme izlemesi (yol kısaltma yok). "Kökler" sütunu iki `find` sonucunu, "Boyutlar" o köklerin ağaç boyutlarını verir.

İzlemenin bazı satırları özellikle öğreticidir. `union(D, I)`'da D kök değildir; ebeveyni C'dir. `find(D)` bu yüzden C'yi döndürür ve birleşme C'nin ağacıyla yapılır. Boyut 3 olan C ağacına tekil I bağlanır. `union(C, A)`'da iki büyük ağaç birleşir: C'nin ağacında {C, K, D, I} (boyut 4), A'nınkinde {A, J, B} (boyut 3) vardır. Küçük olan A'nın ağacı, kökü A ile C'nin altına girer; yeni boyut 7'dir.

İkinci `union(A, B)`'da iki `find` de C'yi döndürür: A'nın ebeveyni C'dir; B'nin yolu B → A → C'dir. Kökler aynıdır, hiçbir şey değişmez. Bu satır, aynı kümedeki iki öğeyi birleştirmeye çalışmanın zararsız olduğunu gösterir. Son çağrı `union(H, B)`'da H'nin kökü F (H → F), B'nin kökü C'dir. F'nin ağacı

{F, E, L, H, G} (boyut 5), C'ninki 7 öğelidir; F, C'nin altına bağlanır. Artık 12 öğenin hepsi tek kümededir ve kök C'dir.



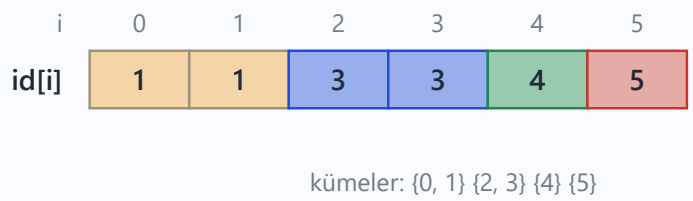
Şekil 13.14: Boyuta göre birleştirme izlemesinden üç an (yukarıdan aşağı). Solda harf, indis ve `ebeveyn` tablosu, ortada işlem listesi, sağda orman. (a) `union(A, B)` sonrası: B, A'nın altında. (b) `union(C, A)` sonrası: A'nın ağacı C'ye bağlanmış, boyut 7. (c) Sonuç: tek küme, kök C.

Son durumda "H ile B aynı kümede mi?" sorusunu yanıtlayalım. `find(H)`: H → F → C, iki adım. `find(B)`: B → A → C, iki adım. Kökler aynıdır; yanıt evettir ve toplam 4 adım sürmüştür. Ağacın yüksekliği 3'tür: en derin düğüm G'dir (G → H → F → C). 12 öğelik bir kümede yüksekliğin bu kadar küçük kalması tesadüf değildir; boyuta göre birleştirmenin sonucudur.

13.4.6 Başka bir fikir: Quick-Find

Ağaçları hiç kullanmayan daha basit bir gösterim de düşünülebilir. Sedgewick ve Wayne'in Quick-Find (hızlı bul) adını verdiği bu yöntemde düz bir `id` dizisi tutulur: `id[i]`, *i* öğesinin küme adıdır. Aynı kümedeki öğelerin `id`'leri aynıdır. Bu durumda `find(p)` tek bir dizi okumasıdır: `id[p]`. Süre *O(1)*'dir.

Bedeli `union` öder. `union(p, q)`'da p'nin kümesindeki *bütün* öğelerin adı q'nun küme adıyla değiştirilmelidir. Bu öğelerin nerede olduğu bilinmediği için bütün dizi taranır: *O(n)*.



Şekil 13.15: Quick-Find `id` dizisi, `union(0, 1)` ve `union(2, 3)` sonrası: `id = [1, 1, 3, 3, 4, 5]`. Dört küme vardır: {0, 1}, {2, 3}, {4}, {5}. Aynı renkteki hücreler aynı kümededir.

Yöntem	`find`	`union`	*n − 1* `union` toplamı
Quick-Find (dizi, küme adı)	O(1)	O(n)	$O(n^2)$
Quick-Union (ebeveyn ağacı, naif)	O(h), en kötü O(n)	O(h)	en kötü $O(n^2)$
Rütbe/boyut + yol kısaltma	$O(\alpha(n))$ amortize		≈ O(n)

Tablo 13.16: Üç Union-Find yönteminin maliyetleri (Sedgewick ve Wayne'in sınıflamasıyla). *h*: ağaç yüksekliği.

Tablo, bu kesimin ve bir sonrakinin yol haritasıdır. Quick-Find `find`'ı hızlı yapar ama her `union` diziyi tarar; *n* öğeyi tek kümede toplamak için gereken *n − 1* birleşme $O(n^2)$ sürer. Ebeveyn ağacının naif hâli (Quick-Union) `union`'ı ucuzlatır, ama ağaç uzayabilir ve `find` yüksekliğe bağlı kalır. En iyi sonuç, ağaçları basık tutan iki iyileştirmenin birlikte kullanılmasıyla elde edilir; tablodaki $\alpha(n)$ 13.5.5'te tanıtılır ve pratikte 4'ten büyük olmaz.

13.4.7 Kod izleme: Quick-Find union

```
int find(int p) { return id[p]; }  // O(1)
void union(int p, int q) {
  int pid = id[p], qid = id[q];
  if (pid == qid) return;
  for (int i = 0; i < id.length; i++)
    if (id[i] == pid)
      id[i] = qid;           // O(n) tarama
}
```

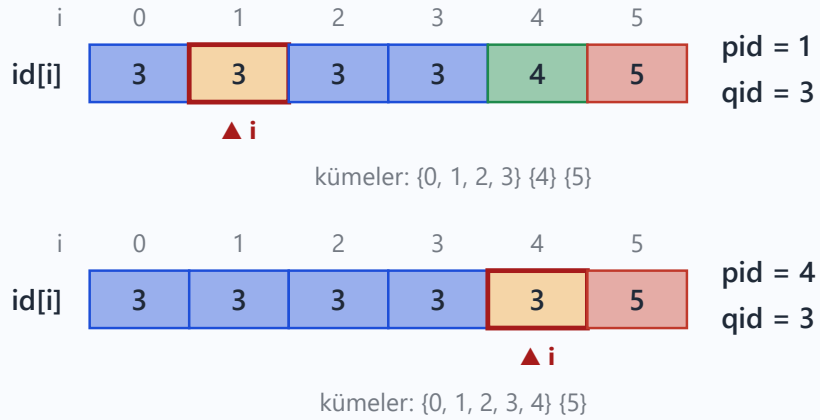
`find` tek satırdır. `union` önce iki küme adını okur: `pid = id[p]`, `qid = id[q]`. Adlar eşitse öğeler zaten aynı kümededir ve metot hemen döner. Değilse dizinin her hücresine bakılır; küme adı `pid` olan her hücreye `qid` yazılır.

Kodu Şekil 13.15'teki durumdan başlayarak izleyelim: `id = [1, 1, 3, 3, 4, 5]`. Önce `union(1, 3)`, sonra `union(4, 3)` çağrılıyor.

Çağrı	`pid`, `qid`	`i`	`id[i] == pid` ?	`id` (sonra)
`union(1, 3)`	1, 3	0	1 = 1 evet → 3	[3, 1, 3, 3, 4, 5]
		1	1 = 1 evet → 3	[3, 3, 3, 3, 4, 5]
		2	3 ≠ 1	değişmez
		3	3 ≠ 1	değişmez
		4	4 ≠ 1	değişmez
		5	5 ≠ 1	değişmez
`union(4, 3)`	4, 3	0	3 ≠ 4	değişmez
		1	3 ≠ 4	değişmez
		2	3 ≠ 4	değişmez
		3	3 ≠ 4	değişmez
		4	4 = 4 evet → 3	[3, 3, 3, 3, 3, 5]
		5	5 ≠ 4	değişmez

Tablo 13.17: Quick-Find `union` izlemesi. Her satır döngünün bir turudur.

`union(1, 3)`'te `pid = 1`'dir; 0 ve 1 numaralı hücrelerin adı 3 olur. 2–5 numaralı hücrelere de bakılır, ama hiçbiri değişmez. `union(4, 3)`'te yalnız 4 numaralı hücre değişir. Buna karşın döngü yine altı hücrenin altısına da bakar. Değişen hücre sayısı 2 ya da 1 olsa da bakılan hücre sayısı her seferinde 6'dır.



Şekil 13.16: Quick-Find izlemesinden iki an. Üstte `union(1, 3)` 'ün ikinci turu: 0 ve 1'in adı 3 olmuş, kümeler {0, 1, 2, 3}, {4}, {5}. Altta `union(4, 3)` 'ün beşinci turu: 4'ün adı 3 olmuş, kümeler {0, 1, 2, 3, 4}, {5}.

Maliyet. Her `union` *n* hücreye bakar: *O(n)*. Bütün öğeleri tek kümede toplamak *n − 1* birleşme gerektirir; toplam *(n − 1) · n* hücre bakışı, yani $O(n^2)$. 8 öğe için bu *7 · 8 = 56* bakıştır; bir milyon öğe için yaklaşık 10^{12}'dir ve pratikte kullanılamaz. Quick-Find'ın asıl dersi, `find`'ı sabit yapmanın `union`'ı pahalılaştırdığıdır. Ebeveyn ağacı bu dengeyi tersine çevirir.

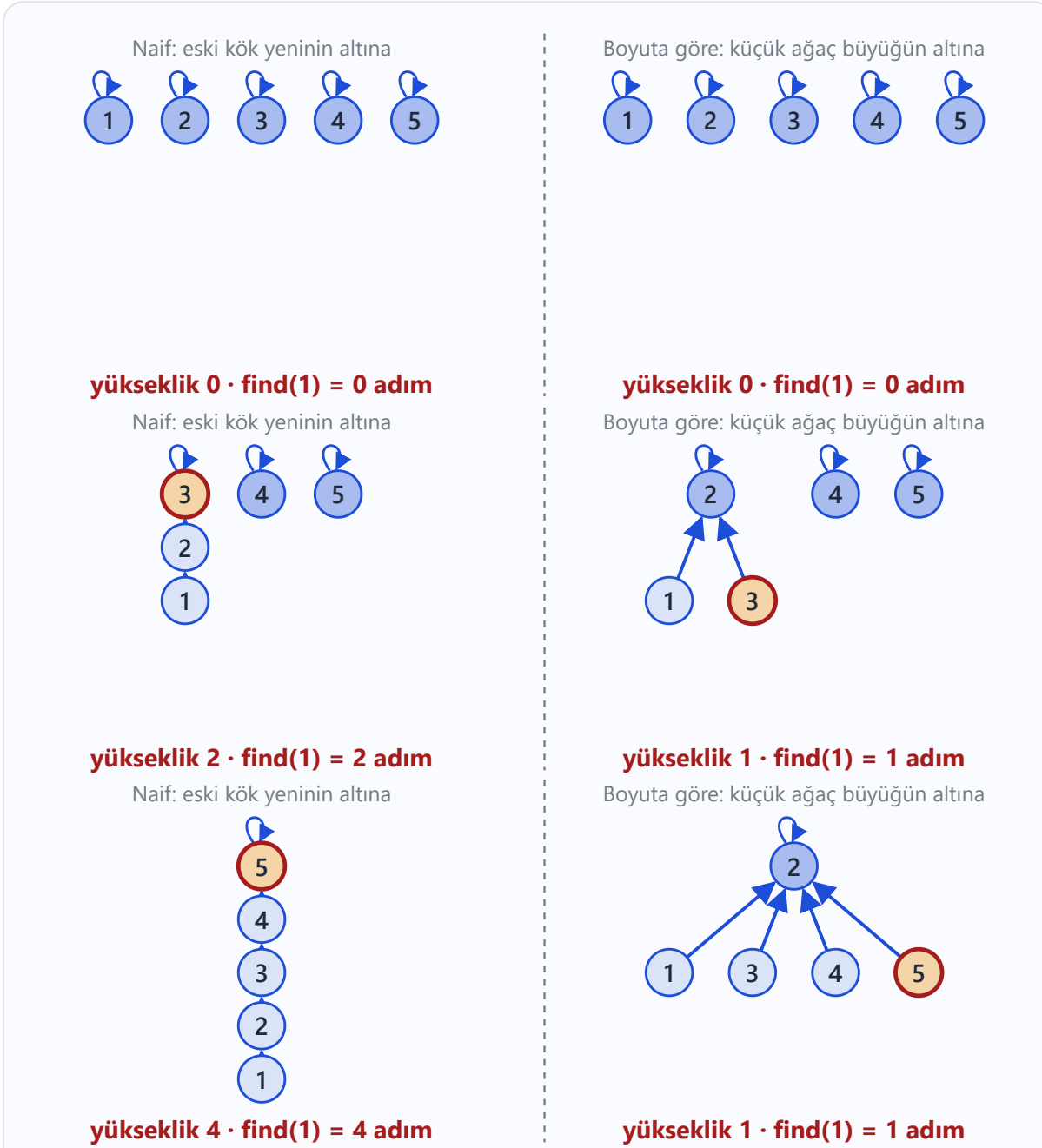
13.5 Union-Find'ı hızlandırmak

Ebeveyn ağacıyla `union`, kökler bulunduktan sonra tek bir atamadır. Bütün maliyet `find`'dadır ve `find`, düğümün kökten uzaklığı kadar adım atar. Bu yüzden Union-Find'ı hızlandırmanın yolu ağaçları *basık* tutmaktan geçer. Bu kesim iki basit fikri ve bunların birlikte verdiği şaşırtıcı sonucu anlatır.

13.5.1 Neden iyileştirme gerekir?

Naif `union(x, y)` her zaman `x`'in kökünü `y`'nin kökünün altına bağlasın. Beş tekil kümede sırasıyla `union(1, 2)`, `union(2, 3)`, `union(3, 4)`, `union(4, 5)` yapılırsa ne olur? İlk çağrıda 1, 2'nin altına girer. İkincide 2'nin iki öğeli ağacı tekil 3'ün altına girer; zincir 1 → 2 → 3 olur. Her yeni çağrıda büyüyen ağaç tek bir düğümün altına girer ve zincir bir kat uzar. Sonunda ağaç 1 → 2 → 3 → 4 → 5 zinciridir. Yüksekliği *n − 1 = 4*'tür ve `find(1)` dört bağlantı izler.

Aynı çağrılar "küçük ağacı büyük ağacın altına bağla" kuralıyla yapılsaydı sonuç bambaşka olurdu. İlk çağrıda iki tekil düğüm vardır; 1 yine 2'nin altına girer. Sonraki her çağrıda tekil düğüm (3, sonra 4, sonra 5) büyük ağacın kökü 2'ye eklenir. Ağaç bir yıldızdır; yüksekliği 1'dir ve her `find` en çok bir adım sürer.



Şekil 13.17: Naif birleştirme (solda) ile boyuta göre birleştirme (sağda) aynı çağrı dizisinde. Üstte başlangıç: beş tekil küme. Ortada iki `union` sonrası: naif ağaç 1 → 2 → 3 zinciri (yükseklik 2), boyuta göre ağaç yıldız (yükseklik 1). Altta dört `union` sonrası: naif ağaç yükseklik 4'lük bir zincir, boyuta göre ağaç hâlâ yükseklik 1.

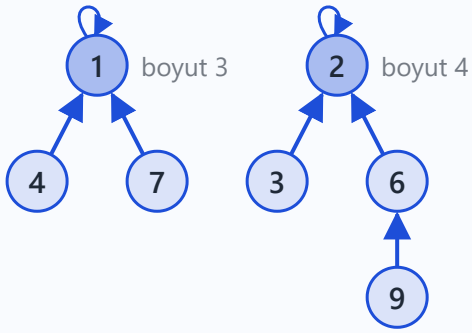
Zincir oluştuğunda her `find` $O(n)$ sürer. Kötü sıralı bir işlem dizisinde m `find` $O(m \cdot n)$ tutar; bu, Quick-Find'dan bile kötü olabilir. İki basit fikir ağaçları basık tutar: (1) **boyuta ya da rütbeye göre birleştirme** ve (2) **yol kısaltma**. Birlikte kullanıldıklarında işlem başına süre neredeyse sabittir: $O(\alpha(n))$.

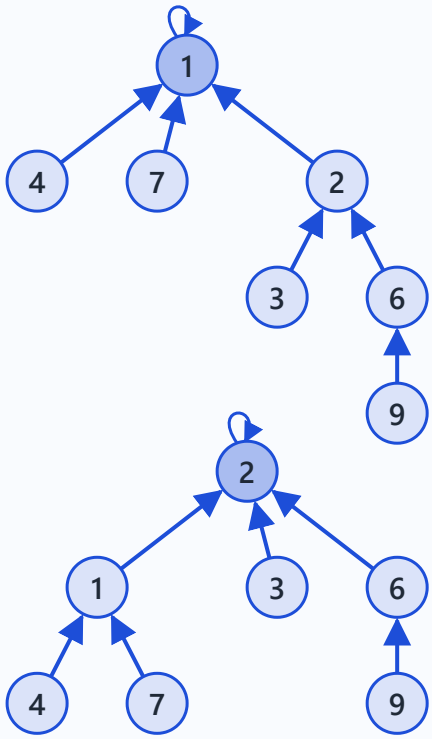
13.5.2 Boyuta göre birleştirme

Tanım 13.5

Boyuta göre birleştirme (*union by size*): her kökte, ağacının **boyutu** (öğe sayısı) saklanır. Birleştirirken küçük ağacın kökü büyük ağacın köküne bağlanır; yeni kökün boyutu iki boyutun toplamı olur. Boyutlar eşitse kökler belirli bir kurala göre (örneğin ikinci kök birincinin altına) bağlanır.

Kuralın gerekçesi şöyle düşünülebilir. Bir ağaç başka bir kökün altına bağlandığında o ağaçtaki *her* düğümün derinliği bir artar. Derinliği artan düğüm sayısını az tutmak için küçük ağaç taşınmalıdır. Büyük ağaç yerinde kalır ve onun düğümlerinin derinliği değişmez.





Şekil 13.18: Boyuta göre birleştirmenin etkisi. Üstte iki küme: kökü 1 olan {1, 4, 7} (boyut 3, yükseklik 1) ve kökü 2 olan {2, 3, 6, 9} (boyut 4, yükseklik 2; 9, 6'nın altında). Altta solda naif birleştirme: büyük ağacın kökü 2, küçük ağacın kökü 1'in altına girer; yükseklik 3'e çıkar. Altta sağda boyuta göre birleştirme: 1, 2'nin altına girer; yükseklik 2 kalır, `boyut[2] = 3 + 4 = 7`.

Şekil 13.18 bunu bir örnekle gösterir. Naif yöntem büyük ağacı küçüğün altına koyarsa 2, 3, 6 ve 9'un derinliği birer artar. 9'un derinliği 2'den 3'e çıkar ve `find(9)` uzar. Boyuta göre yöntem küçük ağacı taşır. 1, 4 ve 7'nin derinliği bir artar, ama bunlar sığ düğümlerdir: 4 ve 7 derinlik 2'ye iner, en derin düğüm yine 9'dur. Yükseklik 2'de kalır.

Teorem 13.1 (Boyuta göre birleştirmede yükseklik)

Boyuta göre birleştirme kullanılsın; yol kısaltmanın olup olmaması sonucu değiştirmez. Bu durumda n öğeli bir ormanda hiçbir düğümün derinliği $\lfloor \log_2 n \rfloor$'yi aşmaz. Dolayısıyla `find` ve `union` en kötü durumda $O(\log n)$'dir.

İspat. Bir x düğümünü izleyelim. x'in derinliği yalnız bir yolla artar: x'in bulunduğu ağaç başka bir kökün altına bağlanır. Kurala göre bu yalnız x'in ağacı öteki ağaçtan büyük değilse olur. Birleşmeden sonra x'in içinde bulunduğu ağacın boyutu en az iki katına çıkar: eski boyut s ise yeni boyut en az $s + s = 2s$'dir. Başlangıçta boyut 1'dir ve hiçbir ağaç n'den büyük olamaz. Boyut en çok $\lfloor \log_2 n \rfloor$ kez iki katına çıkabilir; x'in derinliği de en çok o kadar artar. Yol kısaltma derinlikleri yalnız azaltır, sınırı bozmaz. □

Teorem, 13.4.5'teki izlemeyle uyumludur: 12 öğe için $\lfloor \log_2 12 \rfloor = 3$'tür ve ağacın yüksekliği gerçekten 3 olmuştu. Sınıra eşitlik de ulaşılabilir. 8 öğeyi önce ikişerli, sonra dörderli, sonra sekizli eşit boyutlu ağaçlarda birleştirirsek yüksekliği $\log_2 8 = 3$ olan bir ağaç elde ederiz. Teorem bundan daha kötüsünün olamayacağını söyler.

13.5.3 Rütbeye göre birleştirme

Boyut yerine ağacın yüksekliğini izlemek de aynı işi görür. Ama yol kısaltma kullanıldığında gerçek yüksekliği güncel tutmak pahalıdır: bir `find` ağacın yüksekliğini değiştirebilir ve bunu anlamak için bütün ağaçta gezinmek gerekir. Bu yüzden yüksekliğin yerine onun bir üst sınırı tutulur.

> **Tanım 13.6**
>
> **Rütbe** (*rank*), her kökte tutulan ve ağacın yüksekliğinin üst sınırı olan bir tam sayıdır. Yeni oluşturulan tek öğeli kümede rütbe 0'dır. **Rütbeye göre birleştirmede** (*union by rank*) küçük rütbeli kök büyük rütbeli köke bağlanır ve rütbeler değişmez. Rütbeler eşitse kökler birbirine bağlanır ve yeni kökün rütbesi bir artar.

CLRS'deki sözde kod, işlemi üç parçaya ayırır:

```
makeSet(x):
  ebeveyn[x] = x
  sira[x] = 0

find(x):   // yol kısaltmalı
  if x != ebeveyn[x]:
    ebeveyn[x] = find(ebeveyn[x])
  return ebeveyn[x]

link(x, y):   // x, y kök
  if sira[x] > sira[y]:
    ebeveyn[y] = x
  else:
    ebeveyn[x] = y
    if sira[x] == sira[y]:
      sira[y] = sira[y] + 1

union(x, y):
  link(find(x), find(y))
```

`makeSet(x)` tek öğeli bir küme kurar: x kendinin ebeveynidir, rütbesi 0'dır. `find(x)` yol kısaltmalıdır. `link(x, y)` iki *kökü* bağlar. `x`'in rütbesi büyükse `y`, `x`'in altına girer. Değilse `x`, `y`'nin altına girer; rütbeler eşitse `y`'nin rütbesi bir artar. `union(x, y)` önce iki kökü bulur, sonra onları `link` ile bağlar. Kodda `sira` dizisi rütbeyi tutar.

Eşit rütbede neden artırma yapılır? Rütbesi r olan iki ağaç birleşince birinin kökü ötekinin çocuğu olur. O ağacın en derin düğümü bir kat aşağı iner ve yeni ağacın yüksekliği $r + 1$'e çıkabilir. Rütbeler farklıysa alçak ağaç bağlanır; onun derinleşen düğümleri yüksek ağacın yüksekliğini aşamaz ve rütbe değişmez.

Teorem 13.2 (Rütbe ve öğe sayısı)

Rütbeye göre birleştirmede rütbesi r olan bir kökün ağacında en az 2^r öğe vardır. Bu yüzden n öğede hiçbir rütbe $\lfloor \log_2 n \rfloor$'yi aşmaz ve yol kısaltma olmasa bile yükseklik $O(\log n)$'dir.

İspat. Rütbe üzerinden tümevarım yapalım. Rütbe 0 olan kökün ağacında en az $1 = 2^0$ öğe vardır. Bir kökün rütbesi r'den $r + 1$'e yalnız rütbesi r olan başka bir kökle birleşince çıkar. Tümevarım varsayımına göre iki ağaçta da en az 2^r öğe vardır; birleşimde en az $2 \cdot 2^r = 2^{r+1}$ öğe olur. Öğe sayısı n'yi aşamayacağından $2^r \le n$, yani $r \le \log_2 n$'dir. Rütbe yüksekliğin üst sınırı olduğu için yükseklik de bu sınırı aşmaz. □

Bu sınıra da ulaşılabilir. 16 öğe önce ikişer ikişer, sonra dörder dörder, sonra sekizer sekizer birleştirilsin. En son iki sekizli ağaç birleşince kökün rütbesi $4 = \log_2 16$ olur.

Yol kısaltma devreye girdiğinde rütbe ile gerçek yükseklik birbirinden ayrılabilir. Yol kısaltma ağacı basıklaştırır, ama rütbeyi güncellemez. Bu yüzden rütbe yalnız bir *üst sınırdır*; gerçek yükseklik rütbeden küçük olabilir. Bu bir hata değildir. Rütbe, karar vermek için yeterince iyi bir ölçüdür ve onu güncel tutmak hiçbir ek maliyet gerektirmez.

Sık yapılan hata

Eşit rütbede artırmayı unutmak. `sira[ky]++` satırı yazılmazsa bütün rütbeler 0'da kalır ve her birleşme "eşit rütbe" dalına düşer. Kural fiilen naif birleştirmeye döner: rütbe yüksekliği izlemez ve $O(\log n)$ garantisi kaybolur. 13.5.1'deki gibi bir çağrı dizisi yine zincir üretebilir.

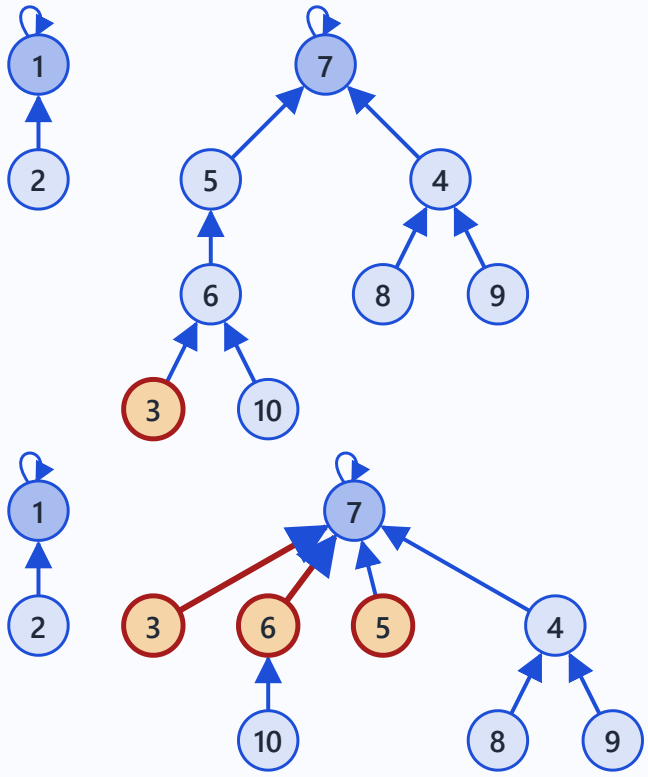
13.5.4 Yol kısaltma

İkinci fikir `find`'ın kendisini kullanır. `find(x)` kökü bulmak için x'ten köke kadar bir yol yürür. Bu yoldaki her düğümün kökü de aynıdır. Yürüyüş bitince bu düğümlerin hepsini doğrudan köke bağlamak, sonraki `find`'ları tek adıma indirir.

Tanım 13.7

Yol kısaltma (*path compression*): `find(x)` sırasında x'ten köke giden yolda rastlanan bütün düğümlerin ebeveyni kök yapılır.

Özyinelemeli `find`'da bu iş kendiliğinden olur: `ebeveyn[x] = find(ebeveyn[x])`. Özyineleme köke kadar iner; dönüşte her çağrı, kendisine dönen kökü düğümünün yeni ebeveyni yapar.



Şekil 13.19: Yol kısaltma. Üstte önce: kökü 7 olan ağaçta 3'ün yolu 3 → 6 → 5 → 7'dir (3 adım); 10 da 6'ya bağlıdır. Yanda kökü 1 olan ayrı bir ağaç ({1, 2}) vardır. Altta `find(3)` sonrası: 3 ve 6 doğrudan 7'ye bağlanmıştır (5 zaten 7'nin çocuğuydu); 10 hâlâ 6'ya bağlıdır, ama onun yolu da kısalmıştır.

Şekil 13.19 üzerinde `find(3)`'ü izleyelim. Özyineleme 3'ten başlar: `ebeveyn[3] = 6`, kök değildir, `find(6)` çağrılır. 6'nın ebeveyni 5'tir, `find(5)` çağrılır. 5'in ebeveyni 7'dir, `find(7)` çağrılır. 7 kendini gösterir; köktür ve 7 döner. Dönüşte `find(5)`, `ebeveyn[5] = 7` yazar; değer zaten 7'dir. `find(6)`, `ebeveyn[6] = 7` yazar; 6 artık doğrudan köke bağlıdır. `find(3)`, `ebeveyn[3] = 7` yazar. Sonuç olarak 3'ün yolu üç adımdan bir adıma iner. 6'nın çocuğu olan 10 bu yolda değildi, ebeveyni değişmedi. Buna karşın 10'un yolu da kısaldı: 10 → 6 → 7, yani iki adım.

Yol kısaltma tek bir `find`'ı hızlandırmaz; aksine o `find` biraz daha çok iş yapar (her düğüme bir yazma). Kazanç sonraki işlemlerdedir. Bir kez uzun bir yol yürüyen `find`, o yolu ortadan kaldırır ve aynı düğümlere yapılan sonraki sorgular ucuzlar. Bu, maliyetin tek tek işlemlere göre değil, işlem dizisi üzerinden ortalandığı amortize analizin tipik bir örneğidir.

13.5.5 Karmaşıklık ve ters Ackermann fonksiyonu

Gerçekleme	`find`	`union`	Not
Naif (rastgele bağlama)	O(n)	O(n)	ağaç zincire dönüşebilir
Boyuta veya rütbeye göre	O(log n)	O(log n)	yükseklik ≤ $\log_2 n$
Yalnız yol kısaltma	O(log n) amortize		m işlem için
Rütbe + yol kısaltma	**O(α(n)) amortize**		pratikte sabit

Tablo 13.18: Union-Find gerçeklemelerinin karmaşıklığı. Amortize sınırlar, m işlemlik diziler için işlem başına ortalamadır.

Tablonun ilk iki satırı önceki kesimlerden bellidir. `union`'ın maliyeti iki `find`'ın maliyetidir; bağlama sabit süredir. Üçüncü satır, yalnız yol kısaltma kullanıldığında da işlem başına ortalama maliyetin logaritmik olduğunu söyler. Tek bir `find` uzun sürebilir, ama uzun yollar bir kez yüründükten sonra kısaldığı için dizinin toplamı sınırlı kalır.

Son satır bölümün en çarpıcı sonucudur. Burada $\alpha(n)$, **ters Ackermann fonksiyonudur**. Ackermann fonksiyonu $A_k(j)$ akıl almaz bir hızla büyür: k bir arttığında yeni fonksiyon, bir öncekinin kendini art arda uygulamasıyla elde edilir. Ters fonksiyon $\alpha(n)$, $A_k(1) \geq n$ olan en küçük k'dir ve tam tersine son derece yavaş büyür. CLRS'deki tanımla $A_4(1)$, 10^{80}'den (gözlenebilir evrendeki atom sayısının kestirimi) kat kat büyüktür. Bu yüzden evrendeki atom sayısı kadar öğe için bile $\alpha(n) \leq 4$'tür. Pratikte $\alpha(n)$ bir sabit gibi davranır.

> **Teorem 13.3 (Tarjan, 1975)**
>
> Rütbeye göre birleştirme ve yol kısaltma birlikte kullanılsın. n öğe üzerinde m tane `makeSet`, `find` ve `union` işleminden oluşan herhangi bir dizi $O(m \cdot \alpha(n))$ sürede biter.

Teoremin ispatı bu notun kapsamını aşar; CLRS'de ayrı bir kesim olarak verilir. Sonucu üç noktayla yorumlamak yeterlidir. Birincisi, sınır *amortizedir*: tek bir `find` yine $O(\log n)$ sürebilir, ama dizinin toplamı $O(m \cdot \alpha(n))$'yi aşmaz. İkincisi, sınır *sıkıdır*: Tarjan aynı makalede bu algoritmanın $m \cdot \alpha$ ile orantılı süre harcadığı işlem dizileri bulunduğunu da göstermiştir. Yani algoritma tam doğrusal süreye inmez; makalenin başlığındaki "iyi ama doğrusal olmayan" nitelemesi buradan gelir. Üçüncüsü, bellek gereksinimi küçüktür: iki tam sayı dizisi, `ebeveyn[n]` ve `sira[n]` (ya da `boyut[n]`), yani $O(n)$.

Union-Find'ın kullanıldığı yerler çeşitlidir. Kruskal'ın minimum kapsayan ağaç algoritması bir kenarın döngü oluşturup oluşturmadığını `find` ile sorar. Bir çizgenin bağlantılı bileşenleri kenarlar geldikçe birleştirilerek bulunur (13.5.10). Resim işlemede aynı renkteki bitişik pikseller bölgelere ayrılır (bölge etiketleme). Bir bilgisayar ağında iki makinenin birbirine ulaşıp ulaşamayacağı sorgulanır.

13.5.6 Java: düğüm tabanlı Union-Find

Union-Find, öğeleri nesne olarak da tutabilir. Her düğüm, verisini, rütbesini ve ebeveyninin referansını saklar:

```
class SetDugum {
  int veri;
  int sira;   // rütbe
  SetDugum ebeveyn;

  SetDugum(int veri) {
    this.veri = veri;
    this.ebeveyn = this;
    this.sira = 0;
  }
}

SetDugum find(SetDugum d) {
  if (d != d.ebeveyn)
    d.ebeveyn = find(d.ebeveyn); // kısaltma
  return d.ebeveyn;
}
```

Yapıcı, düğümü kendi ebeveyni yapar ve rütbesini 0'a ayarlar; bu, `makeSet`'in nesne karşılığıdır. `find`, sözde kodun Java çevirisidir: düğüm kendi ebeveyni değilse ebeveyninin kökü bulunur, bu kök düğümün yeni ebeveyni yapılır (yol kısaltma) ve döndürülür. Karşılaştırma `d != d.ebeveyn` referans karşılaştırmasıdır; burada doğru olan da budur, çünkü sorulan, iki referansın aynı nesneyi gösterip göstermediğidir.

```
void union(SetDugum x, SetDugum y) {
  SetDugum kokX = find(x);
  SetDugum kokY = find(y);
  if (kokX == kokY) return;
  // rütbeye göre birleştirme
  if (kokX.sira < kokY.sira) {
    kokX.ebeveyn = kokY;
  } else if (kokX.sira > kokY.sira) {
    kokY.ebeveyn = kokX;
  } else { // eşit: birini seç
    kokX.ebeveyn = kokY;
    kokY.sira++;
  }
}
```

`union` iki kökü bulur. Kökler aynı nesneyse öğeler zaten aynı kümededir. Değilse rütbeler karşılaştırılır: küçük rütbeli kök büyük rütbelinin altına girer. Rütbeler eşitse `kokX`, `kokY`'nin altına girer ve `kokY`'nin rütbesi bir artar. Bu yaklaşımın üstünlüğü, öğelerin doğrudan nesne olarak kullanılmasıdır;

indis eşlemesine gerek kalmaz. Bedeli bellek ve hızdır: her düğüm ayrı bir nesnedir ve referans izlemek, dizi okumaktan yavaştır.

13.5.7 Java: dizi tabanlı Union-Find

Öğeler *0..n − 1* indisleriyse aynı yapı iki tam sayı dizisiyle yazılır. Bu, pratikte en çok kullanılan biçimdir:

```
class UnionFind {
  private final int[] ebeveyn, sira;
  UnionFind(int n) {
    ebeveyn = new int[n];  sira = new int[n];
    for (int i = 0; i < n; i++) ebeveyn[i] = i;
  }
  int find(int x) {
    if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
    return ebeveyn[x];
  }
  boolean union(int x, int y) {
    int kx = find(x), ky = find(y);
    if (kx == ky) return false;
    if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
    else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
    else { ebeveyn[kx] = ky; sira[ky]++; }
    return true;
  }
}
```

Yapıcı iki diziyi ayırır. Her öğe kendi ebeveynidir; `sira` dizisi Java'nın varsayılanıyla sıfırlarla başlar. `find`, özyinelemeli yol kısaltmayı tek satırda yapar. `union`, iki kökü bulur ve aynılarsa `false` döndürür. Değilse rütbeye göre bağlar ve `true` döndürür.

`union`'ın `boolean` döndürmesi bilinçli bir tasarım kararıdır. Dönüş değeri, "bu çağrı iki farklı kümeyi gerçekten birleştirdi mi?" sorusunun yanıtıdır. Kruskal algoritması bir kenarı eklemeden önce tam olarak bunu sorar: kenarın iki ucu zaten aynı kümedeyse kenar bir döngü kapatır ve atlanır. Bağlantılı bileşen sayma da (13.5.10) aynı bilgiyle çalışır.

Kısa bir kullanım örneği: `UnionFind uf = new UnionFind(5); uf.union(0, 1); uf.union(3, 4);` çağrılarından sonra `uf.find(1) == uf.find(0)` `true`, `uf.find(1) == uf.find(3)` `false` verir. İlk birleşmede rütbeler eşittir; 0, 1'in altına girer ve `sira[1] = 1` olur. İkincide 3, 4'ün altına girer.

> **Sık yapılan hata**
>
> Çok büyük *n*'de özyinelemeli `find`'ın derinliğini unutmak. Rütbeye göre birleştirmeyle yükseklik *O(log n)*'dir ve özyineleme sorun çıkarmaz. Ama yalnız yol kısaltma kullanan ya da naif bağlayan bir gerçeklemede yükseklik *n*'ye yaklaşabilir. Yüz binlerce öğelik bir zincirde özyine-

lemeli `find` `StackOverflowError` verir. Bu durumda `find` döngüyle yazılır (13.5.9).

13.5.8 Kod izleme: find ve union

Dizi tabanlı yapıyı 6 öğe üzerinde izleyelim. Çağrı dizisi şudur: `union(0, 1)`, `union(2, 3)`, `union(0, 2)`, `union(4, 0)`, `union(1, 4)`. İzlemede kullanılan metotlar şunlardır:

```
int find(int x) {
  if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
  return ebeveyn[x];
}
boolean union(int x, int y) {
  int kx = find(x), ky = find(y);
  if (kx == ky) return false;
  if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
  else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
  else { ebeveyn[kx] = ky; sira[ky]++; }
  return true;
}
```

Çağrı	`kx`, `ky`	Rütbeler	Karar	Dönüş	`ebeveyn`	`sira`
başlangıç	—	—	—	—	[0, 1, 2, 3, 4, 5]	[0, 0, 0, 0, 0, 0]
`union(0, 1)`	0, 1	0 = 0	0 → 1, `sira[1]++`	`true`	[1, 1, 2, 3, 4, 5]	[0, 1, 0, 0, 0, 0]
`union(2, 3)`	2, 3	0 = 0	2 → 3, `sira[3]++`	`true`	[1, 1, 3, 3, 4, 5]	[0, 1, 0, 1, 0, 0]
`union(0, 2)`	1, 3	1 = 1	1 → 3, `sira[3]++`	`true`	[1, 3, 3, 3, 4, 5]	[0, 1, 0, 2, 0, 0]
`union(4, 0)`	4, 3	0 < 2	4 → 3 (yol kısaltma: 0 → 3)	`true`	[3, 3, 3, 3, 3, 5]	[0, 1, 0, 2, 0, 0]
`union(1, 4)`	3, 3	—	aynı küme	`false`	[3, 3, 3, 3, 3, 5]	[0, 1, 0, 2, 0, 0]

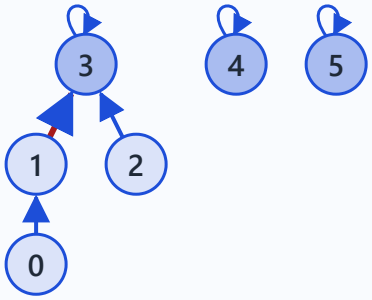
Tablo 13.19: `new UnionFind(6)` üzerinde beş `union` çağrısının izlemesi. Diziler çağrıdan sonraki değerlerdir.

İlk iki çağrıda bütün öğeler köktür; her `find` ilk denetimde kendini döndürür. Rütbeler eşit olduğu için `else` dalı çalışır: `ebeveyn[kx] = ky` ve `sira[ky]++`. Böylece {0, 1} kümesinin kökü 1, {2, 3} kümesinin kökü 3 olur; ikisinin de rütbesi 1'dir.

Üçüncü çağrı `union(0, 2)`'de öğeler artık kök değildir. `find(0)`: `ebeveyn[0] = 1 ≠ 0`, `find(1)` çağrılır; 1 köktür ve döner. Dönüşte `ebeveyn[0] = 1` yazılır; 0 zaten kökün çocuğudur, bir şey değişmez. Aynı biçimde `find(2) = 3`'tür. Kökler 1 ve 3'tür, rütbeleri eşittir (1). 1, 3'ün altına girer ve `sira[3] = 2` olur. Bu çağrıdan sonra ağaç şöyledir: kök 3; çocukları 2 ve 1; 1'in çocuğu 0. Yükseklik 2'dir ve rütbe 2 bunu doğru gösterir.



i	0	1	2	3	4	5
ebeveyn	1	1	2	3	4	5
sira	0	1	0	0	0	0

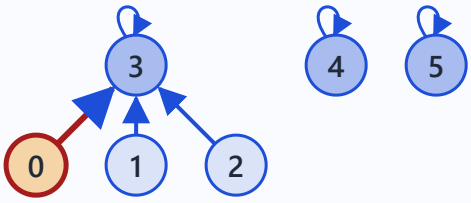


i	0	1	2	3	4	5
ebeveyn	1	3	3	3	4	5
sira	0	1	0	2	0	0

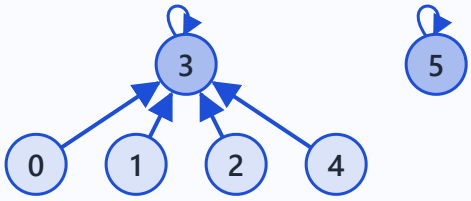
Şekil 13.20: İzlemenin ilk yarısından iki an. (a) `union(0, 1)` sonrası: 0, 1'in altında; `sira[1] = 1`. (b) `union(0, 2)` sonrası: rütbesi 1 olan iki kök birleşmiş, 1 → 3; `sira[3] = 2`. 0'ın yolu 0 → 1 → 3'tür.

Dördüncü çağrı `union(4, 0)` yol kısaltmanın etkisini gösterir. `find(4)` hemen 4'ü döndürür. `find(0)` ise iki katlı bir özyinelemedir: 0 → 1 → 3. `find(3)` 3'ü döndürür. Dönüşte `find(1)`, `ebeveyn[1] = 3` yazar (zaten 3'tür). `find(0)`, `ebeveyn[0] = 3` yazar; 0'ın ebeveyni 1'den 3'e değişir. 0 artık doğrudan köke bağlıdır. Kökler 4 ve 3'tür. `sira[4] = 0 < sira[3] = 2` olduğu için alçak ağacın kökü 4, 3'e bağlanır; rütbe değişmez ve yükseklik artmaz.

Son çağrı `union(1, 4)`'te iki `find` de 3'ü döndürür: 1 ve 4 doğrudan 3'ün çocuklarıdır. `kx == ky` olduğu için metot `false` döndürür. Kruskal algoritmasının diliyle bu kenar bir döngü kapatırdı ve atlanırdı.



i	0	1	2	3	4	5
ebeveyn	3	3	3	3	4	5
sira	0	1	0	2	0	0



i	0	1	2	3	4	5
ebeveyn	3	3	3	3	3	5
sira	0	1	0	2	0	0

Şekil 13.21: İzlemenin ikinci yarısından iki an. (a) `union(4, 0)` içindeki `find(0)` sonrası: yol kısaltma 0'ı doğrudan köke bağlamıştır. (b) Sonuç: {0, 1, 2, 3, 4} tek küme, kök 3; {5} ayrı kümedir. `sira[3] = 2` kalmıştır, ama gerçek yükseklik artık 1'dir.

İzlemenin sonunda rütbe ile yükseklik arasındaki fark da görülür. `sira[3] = 2`'dir, ama yol kısaltma 0'ı köke taşıdığı için gerçek yükseklik 1'dir. Rütbe bir üst sınır olarak doğru kalır. Rütbe ve yol kısaltma birlikte işlem başına yaklaşık $O(\alpha(n))$ maliyet verir.

13.5.9 Döngüyle find ve yol kısaltma

Özyineleme yığın taşması riski taşıdığından `find` iki döngüyle de yazılabilir. Birinci döngü kökü bulur. İkinci döngü aynı yolu yeniden yürür ve yoldaki her düğümü köke bağlar:

```
int find(int x) {
  int kok = x;
  while (ebeveyn[kok] != kok) kok = ebeveyn[kok];
  while (ebeveyn[x] != kok) {    // yol kısaltma
    int sonraki = ebeveyn[x];
    ebeveyn[x] = kok;
    x = sonraki;
  }
  return kok;
}
```

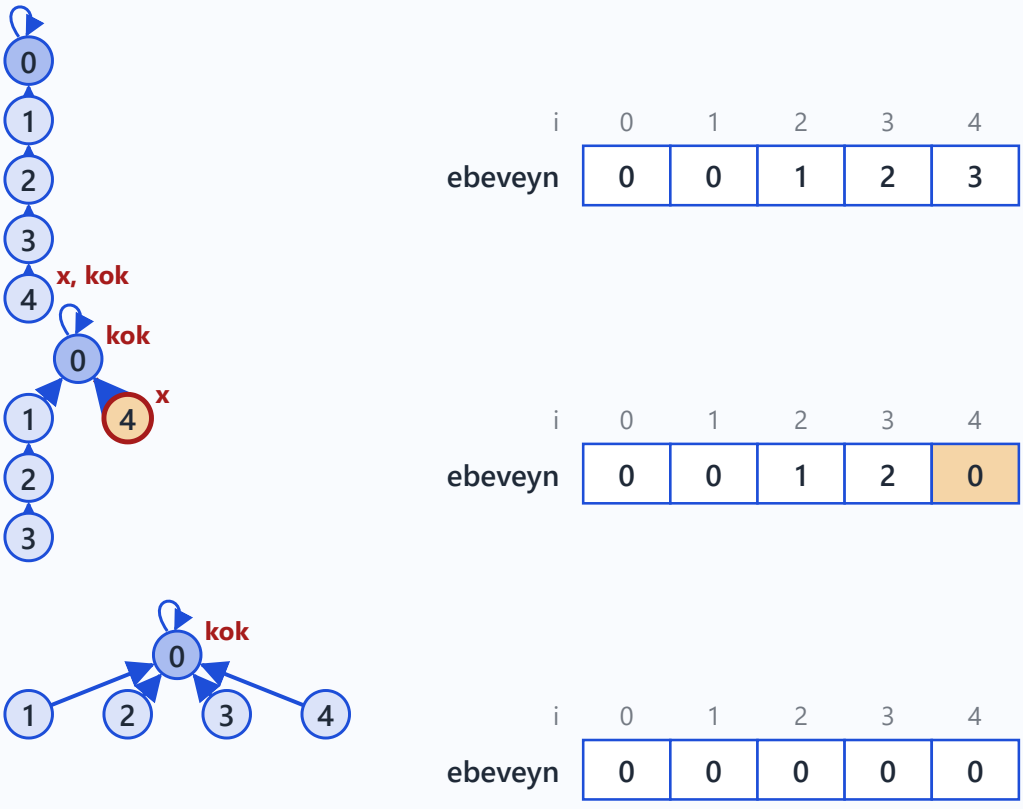
Birinci döngüde `kok` değişkeni x'ten başlar ve ebeveyn bağlantılarını izleyerek kendini gösteren düğüme, yani köke ulaşır. İkinci döngüde `x` yine baştan başlar. Her adımda önce eski ebeveyn `sonraki` değişkenine saklanır; sonra `ebeveyn[x] = kok` yazılır ve `x = sonraki` ile bir yukarı çıkılır. Eski ebeveyni önce saklamak zorunludur: `ebeveyn[x]` değiştirildikten sonra yolun devamı kaybolurdu. Döngü, `ebeveyn[x]` zaten köke eşit olduğunda durur.

Kötü birleştirmeler sonucu oluşmuş `ebeveyn = [0, 0, 1, 2, 3]` zincirinde ($4 \to 3 \to 2 \to 1 \to 0$) `find(4)`'ü izleyelim.

Döngü	Adım	x	kok	sonraki	ebeveyn
—	`kok = x`	4	4	—	[0, 0, 1, 2, 3]
1.	`ebeveyn[4] = 3 ≠ 4`	4	3	—	[0, 0, 1, 2, 3]
1.	`ebeveyn[3] = 2 ≠ 3`	4	2	—	[0, 0, 1, 2, 3]
1.	`ebeveyn[2] = 1 ≠ 2`	4	1	—	[0, 0, 1, 2, 3]
1.	`ebeveyn[1] = 0 ≠ 1`; sonra `ebeveyn[0] = 0`: dur	4	0	—	[0, 0, 1, 2, 3]
2.	`ebeveyn[4] = 3 ≠ 0` → kısalt	3	0	3	[0, 0, 1, 2, **0**]
2.	`ebeveyn[3] = 2 ≠ 0` → kısalt	2	0	2	[0, 0, 1, **0**, 0]
2.	`ebeveyn[2] = 1 ≠ 0` → kısalt	1	0	1	[0, 0, **0**, 0, 0]
2.	`ebeveyn[1] = 0 = kok`: dur	1	0	—	[0, 0, 0, 0, 0]

Tablo 13.20: Döngüyle `find(4)` izlemesi, `ebeveyn = [0, 0, 1, 2, 3]`.

Birinci döngü dört adımda köke ulaşır: $4 \to 3 \to 2 \to 1 \to 0$. İkinci döngü üç düğümü kısaltır: 4, 3 ve 2'nin ebeveyni 0 olur. 1 zaten köke bağlıdır; döngü onda durur. Metot 0'ı döndürür. İşlemden sonra bütün düğümler doğrudan köke bağlıdır ve sonraki her `find` tek adım sürer. Özyineleme kullanılmadığı için zincir ne kadar uzun olursa olsun yığın taşması riski yoktur.



Şekil 13.22: Döngüyle `find(4)`. Üstte başlangıç: 4 → 3 → 2 → 1 → 0 zinciri. Ortada ikinci döngünün ilk adımı: `sonraki = 3` saklanmış, 4 doğrudan 0'a bağlanmış. Altta sonuç: bütün düğümler köke bağlı, `ebeveyn = [0, 0, 0, 0, 0]`.

Maliyet. İki döngü aynı yolu birer kez yürür; süre yol uzunluğunun iki katıdır, yani yine *O(yol uzunluğu)*. Özyinelemeli sürümle aynı kısaltmayı yapar, ek bellek ise *O(1)*'dir; özyinelemeli sürüm yol uzunluğu kadar çağrı çerçevesi kullanır.

13.5.10 Uygulama: bağlantılı bileşen sayma

Bir çizgede birbirine bir yolla ulaşılabilen düğümlerin oluşturduğu en büyük gruplara **bağlantılı bileşen** (*connected component*) denir. *n* düğümlü ve kenarları verilmiş bir çizgede kaç bileşen vardır? Union-Find bu soruyu doğal biçimde yanıtlar: her düğüm başta kendi bileşenindedir ve her kenar, iki ucunun bileşenlerini birleştirir.

```
int bilesenSay(int n, int[][] kenar) {
  UnionFind uf = new UnionFind(n);
  int say = n;          // n tekil küme
  for (int[] e : kenar)
    if (uf.union(e[0], e[1]))
      say--;            // iki bileşen birleşti
  return say;
}
```

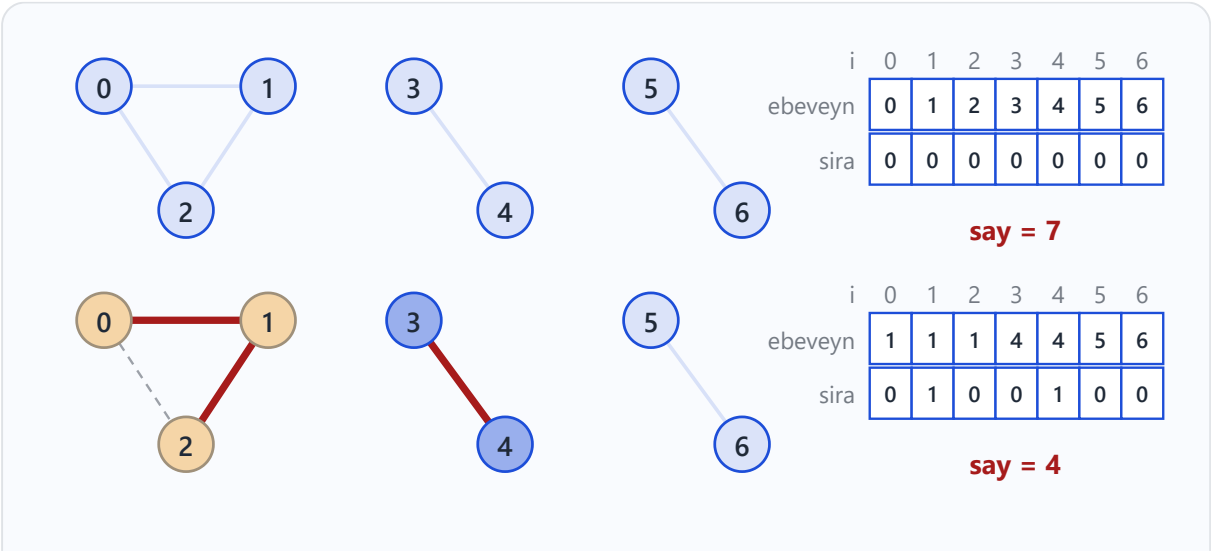
Sayaç `say`, *n* ile başlar: kenar yokken her düğüm ayrı bir bileşendir. Her kenar için `union` çağrılır. `union` `true` döndürürse iki farklı bileşen birleşmiştir ve bileşen sayısı bir azalır. `false` döndürürse kenarın iki ucu zaten aynı bileşendedir; kenar bir döngü kapatır ve sayaç değişmez. `union`'ın dönüş değeri burada da işin kalbidir.

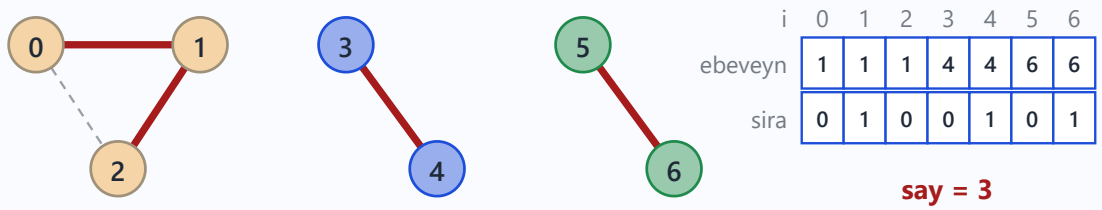
Metodu 7 düğüm ve 5 kenarla izleyelim: `(0, 1)`, `(1, 2)`, `(3, 4)`, `(0, 2)`, `(5, 6)`.

Kenar	Kökler	`union`	`say`	`ebeveyn` (sonra)	Bileşenler
(başlangıç)	—	—	7	[0, 1, 2, 3, 4, 5, 6]	yedi tekil
(0, 1)	0, 1	`true`	6	[1, 1, 2, 3, 4, 5, 6]	{0, 1} ve beş tekil
(1, 2)	1, 2	`true`	5	[1, 1, 1, 3, 4, 5, 6]	{0, 1, 2} ve dört tekil
(3, 4)	3, 4	`true`	4	[1, 1, 1, 4, 4, 5, 6]	{0, 1, 2}, {3, 4}, {5}, {6}
(0, 2)	1, 1	`false`	4	değişmez	değişmez (döngü)
(5, 6)	5, 6	`true`	3	[1, 1, 1, 4, 4, 6, 6]	{0, 1, 2}, {3, 4}, {5, 6}

Tablo 13.21: `bilesenSay(7, kenar)` izlemesi.

Kenar `(1, 2)`'de 1'in rütbesi 1, 2'ninki 0'dır; 2, 1'in altına girer ve rütbe değişmez. Kenar `(0, 2)`'de iki ucun kökü de 1'dir: 0 ile 2, 1 üzerinden zaten bağlıdır. `union` `false` döndürür ve `say` 4'te kalır. Bu kenar 0–1–2 üçgenini kapatan döngü kenarıdır. Son kenardan sonra üç bileşen kalır: {0, 1, 2}, {3, 4}, {5, 6}. Metot 3 döndürür.





Şekil 13.23: Bağlantılı bileşen sayma. Üstte başlangıç: 7 düğüm, kenar yok, `say = 7`. Ortada kenar (0, 2): iki uç zaten aynı bileşende, `union` `false` döndürür ve `say` 4'te kalır. Altta sonuç: üç bileşen, {0, 1, 2}, {3, 4}, {5, 6}.

Maliyet. Yapıyı kurmak *O(n)*'dir. *m* kenarın her biri için bir `union`, yani iki `find` yapılır. Toplam süre *O(n + m · α(n))*'dir. Aynı soru derinlik öncelikli aramayla da *O(n + m)*'de yanıtlanabilir. Union-Find'ın üstünlüğü **dinamik bağlılıktadır**: kenarlar teker teker geldiğinde sayaç her kenardan sonra anında günceldir. Derinlik öncelikli arama ise her yeni kenarda baştan çalıştırılmak zorundadır. "Her kenardan sonra çizge bağlı mı?" sorusu da aynı sayaçla yanıtlanır: `say == 1` ise çizge bağlıdır.

Örnek 13.3

6 düğümlü bir çizgede kenarlar sırayla `(0, 1)`, `(1, 2)`, `(3, 4)`, `(2, 0)` geliyor. Hangi `union` `false` döner, sonda kaç bileşen kalır?

1. **Başlangıç.** `say = 6`.
2. **(0, 1):** kökler 0 ve 1, farklı. Birleşir; `say = 5`.
3. **(1, 2):** 1'in kökü 1, 2'ninki 2. Birleşir; `say = 4`.
4. **(3, 4):** farklı kökler; `say = 3`.
5. **(2, 0):** `find(2) = 1`, `find(0) = 1`. Kökler aynı; `union` `false` döner. Kenar 0–1–2 döngüsünü kapatır, `say` değişmez.
6. **Sonuç.** Üç bileşen: {0, 1, 2}, {3, 4}, {5}. Kenarı olmayan 5 tek başına bir bileşendir.

13.6 Sık yapılan hatalar

Bölüm boyunca yeri geldikçe değinilen hatalar, iki gruba ayrılır. İlk üçü Java kümeleriyle, son üçü Union-Find ve sıralı kümelerle ilgilidir.

Hata	Ne olur?	Doğrusu
Kümedeki öğeyi değiştirmek	Hash koduna giren alan değişen nesne eski kovada kalır; `contains` yeni kovaya bakar ve `false` döner.	Kümedeki nesnelerin hash'e giren alanları değiştirilmez; gerekiyorsa önce çıkarılır, değiştirilir, yeniden eklenir.
`equals` **var,** `hashCode` **yok**	"Eşit" iki nesne farklı kovalara düşer; `HashSet`'te iki kez bulunur.	İkisi birlikte ezilir; eşit nesneler eşit hash kodu üretir.
Gezinirken silmek	for-each içinde `set.remove(x)` → `ConcurrentModificationException`.	`set.removeIf(x -> x % 2 == 0)` ya da yineleyicinin `remove()` metodu.

Hata	Ne olur?	Doğrusu
Kökler yerine öğeleri bağlamak	`ebeveyn[x] = y` : x kök değilse ağacı bölünür.	`ebeveyn[find(x)] = find(y)`
Eşit rütbede artırmayı unutmak	Rütbe yüksekliği izlemez; *O(log n)* garantisi kaybolur.	Eşit rütbede `sira[ky]++` .
`TreeSet` **ile tutarsız** `compare`	`compare(a, b) == 0` olan farklı nesnelerden yalnız biri saklanır.	Sıralama `equals` ile tutarlı olur (`thenComparing`).

Tablo 13.22: Kümeler ve Union-Find ile çalışırken sık yapılan hatalar.

İlk hata özellikle sinsidir, çünkü program hata vermez; yalnız yanlış sonuç üretir. `x` ve `y` alanlarından hash kodu hesaplanan bir `Nokta` nesnesi *(1, 2)* iken kümeye eklensin. Nesne, *(1, 2)*'nin hash kodunun gösterdiği kovaya yerleşir. Ardından `p.x = 5` yazılırsa nesnenin hash kodu değişir, ama nesne eski kovasında durur. `set.contains(p)` yeni hash kodunun kovasına bakar ve nesneyi bulamaz. Kümenin boyutu yine 1'dir; nesne kümededir, ama ona ulaşılamaz. Bu yüzden kümelerde ve eşlemelerde anahtar olarak değiştirilemez nesneler (`String` , `Integer` , `record`) tercih edilir.

Üçüncü hatanın nedeni, yineleyicinin kümenin yapısal değişikliklerini izlemesidir. For-each döngüsü arka planda bir yineleyici kullanır. Döngü sürerken küme başka bir yoldan değişirse yineleyici bunu bir sonraki adımda fark eder ve istisna fırlatır. `removeIf` silme işini yineleyicinin kendi `remove` metoduyla yaptığı için güvenlidir.

13.7 Özet

- **Küme**, tekil öğelerden oluşan sırasız bir koleksiyondur. Temel işlemleri ekleme, çıkarma ve üyelik testidir. Küme işlemleri birleşim ($\cup$), kesişim ($\cap$), fark ($-$) ile alt küme ($\subseteq$) sorusudur; fark değişmeli değildir.
- Java'da küme ADT'si `Set` arayüzüdür. `add` öğe zaten varsa `false` döndürür; bu, tekrarı tek çağrıda bulmayı sağlar. `addAll` , `retainAll` , `removeAll` birleşim, kesişim ve farkı **yerinde** hesaplar; asıl kümeyi korumak için önce kopya alınır.
- **Hash tabanlı kümede** öğe, hash kodundan hesaplanan kovaya yerleşir; aynı kovaya düşen öğeler bir zincirde tutulur. Yük faktörü sabit tutulduğu sürece `add` , `contains` , `remove` ortalama *O(1)*'dir; kötü hash fonksiyonuyla en kötü *O(n)*'dir. `equals` ile `hashCode` birlikte ezilir.
- **Ağaç tabanlı küme** (`TreeSet` , kırmızı-siyah ağaç) her işlemi *O(log n)*'de yapar ve `first` , `floor` , `ceiling` , `subSet` gibi sıralı sorguları da aynı sürede yanıtlar. Eşitlik `compare` ile belirlenir.
- **Sıralı dizilerde** birleşim ve kesişim iki imleçle tek geçişte, *O(n + m)*'de hesaplanır. Ortak öğede iki imleç birlikte ilerler.
- **Bit vektörü**, evren küçükken kümeyi U bitle tutar. Tek öğe işlemleri ve ($U \le 32$ için) küme işlemleri tek makine komutudur; `BitSet` ile küme işlemleri *O(U / 64)*'tür. Seyrek kümelerde hash tabanlı küme daha az bellek kullanır.

- **Çoklu küme** her öğenin çokluğunu tutar; doğal gerçeklemesi öğe → sayaç eşlemesidir. Sayacı 0'a inen girdi silinir; `size()` tekrarları da sayar. **Çoklu eşleme** bir anahtarı bir değer listesine bağlar (`Map<K, List<V>>`).
- **Ayrık kümeler** öğeleri ortak öğesi olmayan gruplara ayırır. **Union-Find** her grubu bir ağaçla tutar; kök kümenin temsilcisidir. `find` köke çıkar, `union` iki kökü bağlar. Nesneler önce indislere eşlenir.
- **Quick-Find** `find`'ı $O(1)$ yapar, ama her `union` bütün diziyi tarar: $n - 1$ birleşme $O(n^2)$. Naif ebeveyn ağacı zincire dönüşebilir ve `find` $O(n)$ olur.
- **Boyuta ya da rütbeye göre birleştirme** yüksekliği $\lfloor \log_2 n \rfloor$ ile sınırlar; rütbesi r olan ağaçta en az 2^r öğe vardır. **Yol kısaltma**, `find`'ın yürüdüğü yoldaki düğümleri doğrudan köke bağlar. İkisi birlikte m işlemi $O(m \cdot \alpha(n))$'de bitirir (Tarjan, 1975); $\alpha(n) \le 4$.
- `union`'ın `boolean` dönüşü "gerçekten birleştirdi mi?" sorusunun yanıtıdır. Bağlantılı bileşen sayımında başarılı her `union` sayacı bir azaltır; Kruskal algoritmasında `false` dönüşü döngü kuran kenarı gösterir.

Yapı	Temel işlemler	Maliyet	Ne zaman?
`HashSet`	`add`, `remove`, `contains`	O(1)*	sıra gerekmiyorsa
`TreeSet`	+ `first`, `floor`, aralık	O(log n)	sıralı erişim gerekiyorsa
Bit vektörü	bit işlemleri	O(1) / O(U / 64)	evren küçük ve yoğunsa
Çoklu küme	`add`, `count`, `remove`	O(1)*	tekrar sayısı önemliyse
Union-Find	`find`, `union`	O(α(n))	ayrık kümeler, bağlantı sorguları

Tablo 13.23: Bölümdeki yapıların özeti. * ortalama durum; Union-Find maliyeti rütbe + yol kısaltma ile amortizedir.

13.8 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

küme *set* · soyut veri tipi *abstract data type (ADT)* · birleşim *union* · kesişim *intersection* · fark *difference* · alt küme *subset* · hash tablosu *hash table* · hash kodu *hash code* · kova *bucket* · çakışma *collision* · ayrı zincirleme *separate chaining* · yük faktörü *load factor* · ikili arama ağacı *binary search tree (BST)* · kırmızı-siyah ağaç *red-black tree* · kök ortada *inorder* · birleştirme sıralaması *merge sort* · bit vektörü *bit vector* · çoklu küme *multiset* · çoklu eşleme *multimap* · eşleme *map* · ayrık kümeler *disjoint sets (union-find)* · ebeveyn *parent* · kök *root* · yükseklik *height* · boyuta göre birleştirme *union by size* · rütbe *rank* · rütbeye göre birleştirme *union by rank* · yol kısaltma *path compression* · amortize analiz *amortized analysis* · bağlantılı bileşen *connected component* · Kruskal algoritması *Kruskal's algorithm* · yineleyici *iterator*

13.9 Kendini Dene

1. $A = \{1, 3, 5, 7, 9\}$ ve $B = \{3, 6, 9\}$ için $A \cup B$, $A \cap B$, $A - B$ ve $B - A$ nedir? $\{3, 9\} \subseteq B$ ve $B \subseteq A$ doğru mu?

   Cevap

   $A \cup B = \{1, 3, 5, 6, 7, 9\}$, $A \cap B = \{3, 9\}$, $A - B = \{1, 5, 7\}$, $B - A = \{6\}$. $\{3, 9\} \subseteq B$ doğrudur. $B \subseteq A$ yanlıştır, çünkü $6 \notin A$. Denetim: $|A \cup B| = 5 + 3 - 2 = 6$.

2. `ilkTekrar(new int[]{5, 1, 9, 1, 5})` ne döndürür? Metot dönene kadar kaç kez `add` çağrılır?

   Cevap

   1 döner. İlk üç `add` (5, 1, 9) `true` döndürür ve küme {1, 5, 9} olur. Dördüncü `add(1)` `false` döndürür; metot 1'i verir. Toplam 4 `add` yapılır; son öğe 5'e hiç bakılmaz. 5 de tekrar eder, ama 1'in tekrarı daha önce görülür.

3. 7 kovalı zincirlemeli bir hash kümesine (kova = `x % 7`) sırasıyla 14, 3, 21, 10, 3 ekleniyor. Her `add` ne döndürür, kovaların son durumu nedir? Ardından `contains(17)` kaç karşılaştırma yapar?

   Cevap

   Kovalar: 14 → 0, 3 → 3, 21 → 0, 10 → 3, 3 → 3. Dönüşler `true, true, true, true, false`'tur. Son durum: kova 0: 14 → 21, kova 3: 3 → 10, öteki kovalar boş, `n = 4`. `contains(17)` : *17 mod 7 = 3*; kova 3'teki 3 ve 10 ile karşılaştırılır (2 karşılaştırma) ve `false` döner.

4. `birlesimKesisim` metodu `a = [2, 5, 8, 11]` ve `b = [1, 5, 11, 12, 15]` ile çağrılıyor. `bir` ve `kes` ne olur? Ana döngü kaç tur döner?

   Cevap

   Turlar: 2 > 1 → 1 eklenir; 2 < 5 → 2 eklenir; 5 = 5 → ortak; 8 < 11 → 8 eklenir; 11 = 11 → ortak. Beşinci turdan sonra `i = 4 = a.length` olur ve ana döngü biter (5 tur). `b`'de kalan 12 ve 15 birleşime eklenir. Sonuç: `bir = [1, 2, 5, 8, 11, 12, 15]`, `kes = [5, 11]`.

5. `a` kümesi {0, 4, 6}, `b` kümesi {4, 5} bitleriyle kuruluyor. `a`, `b`, `a | b`, `a & b` ve `a & ~b`'nin

sayı değerleri ve temsil ettikleri kümeler nedir?

Cevap

a = 1 + 16 + 64 = 81 (0101 0001), *b = 16 + 32 = 48* (0011 0000). `a | b = 113` (0111 0001) → {0, 4, 5, 6}. `a & b = 16` → {4}. `a & ~b = 65` (0100 0001) → {0, 6}. Birleşimde dört bit yanıktır: `Integer.bitCount(a | b) = 4`.

6. "kelebek" kelimesinin harfleri bir `CokluKume<Character>`'e ekleniyor. Ardından iki kez `remove('e')` ve bir kez `remove('l')` çağrılıyor. `count('e')`, `count('l')`, `size()` ne döndürür? Eşlemede kaç girdi kalır?

Cevap

Eklemelerden sonra eşleme {k:2, e:3, l:1, b:1}, `n = 7`'dir. İki `remove('e')` sayacı 3 → 2 → 1'e indirir. `remove('l')`'de sayaç 1 olduğu için girdi silinir. `count('e') = 1`, `count('l') = 0`, `size() = 4`. Eşlemede üç girdi kalır: {k:2, e:1, b:1}.

7. `TreeSet` *S = {4, 9, 13, 18, 22, 30}* için `floor(20)`, `ceiling(20)`, `subSet(9, 22)` ve `floor(3)` ne döndürür?

Cevap

`floor(20) = 18` (≤ 20 olan en büyük), `ceiling(20) = 22` (≥ 20 olan en küçük), `subSet(9, 22) = [9, 13, 18]` (9 dahil, 22 hariç). `floor(3)` `null` döndürür: 3'ten küçük ya da eşit öğe yoktur. Sonucu `int`'e atayan kod burada `NullPointerException` alır.

8. Dizi tabanlı `UnionFind(6)` üzerinde sırasıyla `union(0, 1)`, `union(1, 2)`, `union(3, 4)`, `union(4, 5)`, `union(2, 5)`, `union(0, 3)` çağrılıyor. Son çağrı ne döndürür? `ebeveyn` ve `sira` dizilerinin son hâli nedir?

Cevap

`(0, 1)`: eşit rütbe, 0 → 1, `sira[1] = 1`. `(1, 2)`: rütbe 1 > 0, 2 → 1. `(3, 4)`: 3 → 4, `sira[4] = 1`. `(4, 5)`: 5 → 4. `(2, 5)`: kökler 1 ve 4, rütbeler eşit (1); 1 → 4, `sira[4] = 2`. `(0, 3)`: `find(0)` 0 → 1 → 4 yolunu yürür ve yol kısaltma 0'ı 4'e bağlar; `find(3) = 4`. Kökler aynıdır, çağrı `false` döndürür. Son hâl: `ebeveyn = [4, 4, 1, 4, 4, 4]`, `sira = [0, 1, 0, 0, 2, 0]`. 2 hâlâ 1'e bağlıdır, çünkü hiçbir `find` 2'den geçmemiştir.

9. `ebeveyn = [0, 0, 1, 2, 2, 4]` iken döngüyle yazılmış `find(5)` çağrılıyor. Kök kaç adımda bu-

lunur, dizinin son hâli nedir? 3'ün ebeveyni neden değişmez?

Cevap

Yol 5 → 4 → 2 → 1 → 0'dır; kök 4 adımda bulunur. İkinci döngü 5, 4 ve 2'yi doğrudan 0'a bağlar; 1 zaten köke bağlıdır. Son hâl `[0, 0, 0, 2, 0, 0]` 'dır. 3, 2'nin çocuğudur, ama 5'ten köke giden yolda değildir; yol kısaltma yalnız yürünen yoldaki düğümleri değiştirir. 3'ün yolu yine de kısalmıştır: 3 → 2 → 0.

10. Boyuta göre birleştirmede (eşitlikte ikinci kök birincinin altına) 8 öğeli bir ağacın yüksekliği en çok kaç olabilir? Bu yüksekliği üreten bir `union` dizisi verin.

Cevap

Teoreme göre en çok $\lfloor \log_2 8 \rfloor = 3$. Dizi: `union(0,1)`, `union(2,3)`, `union(4,5)`, `union(6,7)` dört ikili ağaç; `union(0,2)` ve `union(4,6)` iki dörtlü ağaç; `union(0,4)` tek ağaç. Sonuçta `ebeveyn = [0, 0, 0, 2, 0, 4, 4, 6]` 'dır ve 7'nin yolu 7 → 6 → 4 → 0, yani 3 adımdır. Her birleşme eşit boyutlu iki ağacı birleştirdiği için yükseklik her seferinde bir artmıştır.

13.10 İleri okuma

- M. T. Goodrich, R. Tamassia ve M. H. Goldwasser, *Data Structures and Algorithms in Java* (6. baskı, 2014), Kesim 10.5 ve 14.7: küme, çoklu küme ve çoklu eşleme ADT'lerini, genel birleştirme yöntemini ve Kruskal algoritması içinde Union-Find yapısını bu nottakine yakın bir Java diliyle anlatır.
- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 1.5: Quick-Find'dan başlayıp Quick-Union, ağırlıklı birleştirme ve yol kısaltmaya giden adımları deneysel ölçümlerle karşılaştırır.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 19: ayrık küme ormanlarını, rütbeye göre birleştirmeyi, yol kısaltmayı ve $O(m \cdot \alpha(n))$ sınırının ispatını biçimsel olarak verir.
- R. E. Tarjan, “Efficiency of a Good But Not Linear Set Union Algorithm” (1975): rütbe ve yol kısaltmanın birlikte ters Ackermann sınırını verdiğini ve bu sınırın sıkı olduğunu gösteren özgün makale.

13.11 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022.
2. B. A. Galler, M. J. Fischer, “An improved equivalence algorithm”, *Communications of the ACM*, 7(5), 301–303, 1964.
3. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6. baskı, Wiley, 2014.

4. Oracle, *Java SE 17 API Specification*: `java.util.Set`, `HashSet`, `TreeSet` ve `BitSet` belgeleri.
5. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011.
6. R. E. Tarjan, "Efficiency of a Good But Not Linear Set Union Algorithm", *Journal of the ACM*, 22(2), 215–225, 1975.

KISIM IV · BÖLÜM 14

Çizgeler

Şehirler ve aralarındaki yollar, bir sosyal ağdaki kişiler ve arkadaşlıklar, bir bölümdeki dersler ve ön koşulları birbirine benzer. Hepsi aynı soyut yapıyla anlatılır: bir grup nesne ve bu nesneler arasındaki ikili ilişkiler. Bu yapıya çizge denir. Ağaçlarda her düğümün tek bir ebeveyni vardır ve döngü yoktur. Çizgede bu kısıtlar kalkar: bir düğüme pek çok yoldan ulaşılabilir, yollar dönüp aynı düğüme gelebilir. Not, çizgenin temel kavramlarıyla başlar. Ardından çizgenin bellekte nasıl saklandığını ve bu seçimin işlem maliyetlerini nasıl etkilediğini anlatır. Sonra genişlik öncelikli ve derinlik öncelikli aramayı, bunlara dayanan bileşen bulma, iki parçalılık testi, topolojik sıralama ve döngü tespitini işler. Son iki konu en kısa yol (ağırlıksız çizgede BFS, ağırlıklı çizgede Dijkstra) ve minimum kapsayan ağaçtır (Kruskal ve Prim).

Bu bölümün kodu: `github.com/sercankulcu/data-structures-java/Bolum14/src`

ÖĞRENME ÇIKTILARI

Temel kavramlar
Düğüm, kenar, yön, ağırlık, derece, yol, döngü ve bağlantılılık kavramlarını açıklamak; el sıkışma lemmasını kullanarak kenar ve derece sayılarını hesaplamak.

Gösterimler
Bir çizgeyi kenar listesi, komşuluk listesi ve komşuluk matrisi ile saklamak; bu gösterimleri bellek ve işlem maliyetleriyle karşılaştırıp probleme uygun olanı seçmek.

Gezinme ve sıralama
BFS ve DFS'yi elle ve kodla uygulamak; bağlantılı bileşenleri bulmak, iki parçalılığı sınamak, yönlü döngüsüz çizgelerde (DAG) topolojik sıralama yapmak ve yönlü döngüleri bulmak.

En kısa yol ve MKA
Ağırlıksız en kısa yol için BFS'yi, ağırlıklı en kısa yol için Dijkstra'yı, minimum kapsayan ağaç için Kruskal ve Prim algoritmalarını uygulamak ve hangi problemde hangisinin kullanılacağını gerekçelendirmek.

14.1 Temel kavramlar

Çizge algoritmalarını okumak için önce küçük bir sözlüğe ihtiyaç vardır: düğüm, kenar, derece, yol, döngü, bağlantılılık. Bu kavramların çoğu günlük dildeki anlamlarına yakındır. Yine de her birinin kesin bir tanımı vardır. Algoritmaların doğru çalışıp çalışmadığı da çoğu zaman bu kesin tanımlara bakarak anlaşılır.

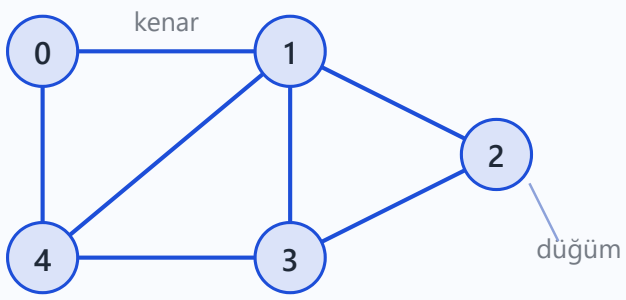
14.1.1 Çizge nedir?

Bir şehir haritasını düşünelim. Haritada kavşaklar ve kavşakları birbirine bağlayan yollar vardır. Bir yolun uzunluğu, eğimi, kaç şeritli olduğu gibi pek çok ayrıntı da vardır. "Hangi kavşaktan hangi kavşağa gidilebilir?" sorusunu yanıtlamak için bu ayrıntıların çoğu gereksizdir. Yalnız iki bilgi yeter: kavşakların listesi ve hangi iki kavşağın doğrudan bir yolla bağlı olduğu. Bu iki bilginin oluşturduğu soyut yapıya **çizge** (*graph*) denir.

Tanım 14.1

Bir **çizge** $G = (V, E)$ ikilisidir. V sonlu bir **düğümler** (*vertex*, *node*) kümesidir. E ise **kenarlar** (*edge*) kümesidir; her kenar iki düğümü birbirine bağlar ve (u, v) biçiminde yazılır. Düğüm sayısı $n = |V|$, kenar sayısı $m = |E|$ ile gösterilir.

Tanımdaki $|V|$ gösterimi, V kümesinin eleman sayısı demektir. Not boyunca n her zaman düğüm sayısını, m her zaman kenar sayısını gösterir. Algoritmaların maliyeti de bu iki sayıyla ifade edilir; örneğin "$O(n + m)$ zaman" ifadesi, sürenin düğüm sayısı ile kenar sayısının toplamıyla orantılı büyüdüğü anlamına gelir.



Şekil 14.1: Beş düğümlü ve yedi kenarlı yönsüz bir çizge. Düğümler 0'dan 4'e numaralanmıştır; daireler düğümleri, çizgiler kenarları gösterir.

Örnek 14.1

Şekil 14.1'deki çizge kümelerle şöyle yazılır:

- $V = \{0, 1, 2, 3, 4\}$, dolayısıyla $n = 5$.
- $E = \{(0,1), (0,4), (1,2), (1,3), (1,4), (2,3), (3,4)\}$, dolayısıyla $m = 7$.

Kenarların yönü yoktur: $(0,1)$ ile $(1,0)$ aynı kenardır. Bu yüzden her kenar listede bir kez yazılır. Şekil ile küme gösterimi aynı bilgiyi taşır; düğümlerin kâğıttaki yeri, çizgilerin uzunluğu ya da eğriliği çizgeye ait değildir.

Çizge, ağacın genelleştirilmiş hâli olarak da düşünülebilir. Ağaçta bir kök vardır, her düğümün (kök dışında) tek bir ebeveyni vardır ve döngü yoktur. Çizgede bu kuralların hiçbiri aranmaz:

- **Kök yoktur.** Hiçbir düğüm ötekilerden ayrıcalıklı değildir. Bir algoritma bir düğümden başlayacaksa bu başlangıç düğümü ayrıca verilir.
- **Döngü olabilir.** Bir düğümden çıkıp kenarlar boyunca ilerleyerek aynı düğüme geri dönmek mümkündür. Şekil 14.1'de $1 \to 2 \to 3 \to 1$ böyle bir dönüştür.
- **Bir düğüme birden çok yoldan ulaşılabilir.** 0'dan 3'e hem $0 \to 1 \to 3$ hem de $0 \to 4 \to 3$ yoluyla gidilir.

Ağaç "bire-çok" ilişkiyi modeller: bir ebeveynin çok çocuğu olur ama her çocuğun tek ebeveyni vardır. Çizge ise "çoğa-çok" ilişkiyi modeller. Bir kişinin pek çok arkadaşı olur; bu arkadaşların her birinin de başka pek çok arkadaşı vardır. Bir ders birkaç dersin ön koşuludur; aynı dersin kendisi de birkaç ön koşula bağlıdır. Bu esneklik çizgeyi güçlü bir model yapar. Bedeli ise algoritmalarda ortaya çıkar: aynı düğüme ikinci kez gelme ihtimali her zaman vardır ve algoritma bunu fark etmek zorundadır.

Sık yapılan hata

Çizgenin çizimini çizgenin kendisi sanmak. Aynı çizge kâğıda pek çok farklı biçimde çizilebilir. Düğümlerin yeri değişince ya da iki kenar şekilde birbirini kesince yeni bir düğüm veya kenar oluşmaz. İki kenarın çizimde kesiştiği nokta bir düğüm değildir. Bir çizgeyi tanımlayan yalnız V ve E kümeleridir.

14.1.2 Düğüm ve kenar

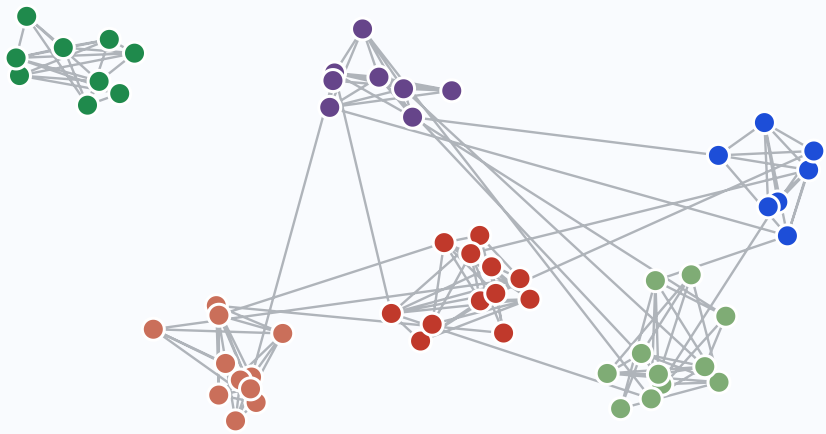
Düğüm, çizgenin temel yapı taşıdır ve modellenen nesneyi temsil eder: bir şehir, bir kişi, bir bilgisayar, bir web sayfası, bir ders. Düğüm yalnız bir numaradan ibaret olmak zorunda değildir; bir etiket, bir ad ya da haritadaki konum gibi bir değer de saklayabilir. Algoritmalar içinse düğümleri 0'dan $n - 1$'e numaralamak en pratik yoldur. Böylece düğüme ait bilgiler dizilerde, düğüm numarası indis olacak biçimde tutulabilir.



Şekil 14.2: Türkiye karayolu ağının basitleştirilmiş bir çizimi. Şehirler düğüm, aralarındaki ana yollar kenardır.

Harita çizge olarak modellenince "İstanbul'dan Van'a en kısa yol hangisi?" sorusu bir **çizge algoritması** problemine dönüşür. Kenarlara yolların uzunluğu yazılırsa bu problemi Dijkstra algoritması çözer (14.4.4). Modelleme, problemin özünü koruyup gereksiz ayrıntıyı atmak demektir. Yolun asfalt mı stabilize mi olduğu en kısa yol için önemli değildir, ama uzunluğu önemlidir.

Kenar, iki düğüm arasındaki ilişkiyi temsil eder: iki şehir arasındaki yol, iki bilgisayar arasındaki ağ bağlantısı, iki kişi arasındaki arkadaşlık. *(v, w)* bir kenar ise *v* ile *w*'ye **bitişik** (*adjacent*) düğümler denir; bitişik iki düğümün her biri ötekinin **komşusudur**. Kenar da bu iki düğüme **bağlıdır** (*incident*). Şekil 14.1'de 1 ile 3 komşudur, 0 ile 2 komşu değildir.



Şekil 14.3: Bir sosyal ağ çizimi. Kişiler düğüm, arkadaşlıklar kenardır. Renkler, kendi içinde sıkı bağlı toplulukları gösterir; topluluklar arasında az sayıda kenar vardır.

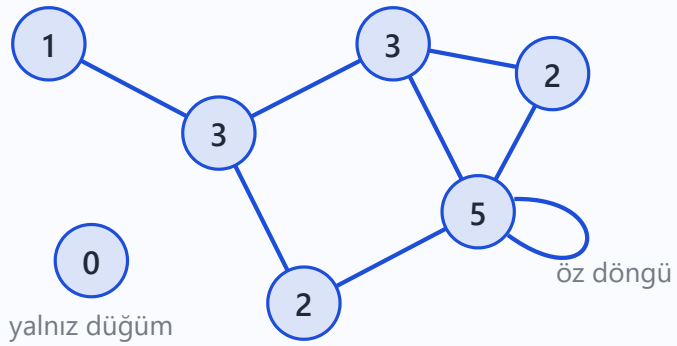
Şekil 14.3 gerçek ağların tipik bir özelliğini gösterir: düğüm sayısı büyüktür ama her düğümün yalnız birkaç komşusu vardır. Bir sosyal ağda milyonlarca kişi bulunur, oysa bir kişinin arkadaş sayısı en çok birkaç yüz ya da birkaç bindir. Kenar sayısı, olabilecek en büyük kenar sayısının çok küçük bir kesridir. Böyle çizgelere **seyrek** çizge denir. Seyreklik, çizgenin bellekte nasıl saklanacağına karar verirken belirleyici olur (14.2.6).

14.1.3 Derece ve el sıkışma lemması

Tanım 14.2

Yönsüz bir çizgede v düğümünün **derecesi** (*degree*), v'ye bağlı kenar sayısıdır ve $deg(v)$ ile gösterilir. Hiçbir kenara bağlı olmayan, yani derecesi 0 olan düğüme **yalnız düğüm** (*isolated vertex*) denir. Bir düğümü kendisine bağlayan kenara **öz döngü** (*self-loop*) denir; öz döngü dereceye 2 katkı yapar.

Günlük hayattaki karşılığı kolaydır: bir kavşağın derecesi, o kavşağa giren yolların sayısıdır. Öz döngünün neden 2 saydığı da bu benzetmeyle görülür. Kavşaktan çıkıp yine aynı kavşağa dönen bir yol kavşağa iki ucuyla bağlanır; kavşakta duran biri o yolun iki ağzını görür.



Şekil 14.4: Her düğümün içine kendi derecesi yazılmıştır: 1, 3, 3, 2, 5, 2 ve 0. Sağ alttaki 5 dereceli düğümde bir öz döngü vardır; soldaki 0 dereceli düğüm yalnızdır.

Şekil 14.4'teki çizgede derecelerin toplamını alalım: $1 + 3 + 3 + 2 + 5 + 2 + 0 = 16$. Çizgede yedi sıradan kenar ve bir öz döngü, toplam sekiz kenar vardır. Toplamın kenar sayısının tam iki katı çıkması rastlantı değildir.

Teorem 14.1 (El sıkışma lemması)

Her yönsüz çizgede düğüm derecelerinin toplamı kenar sayısının iki katıdır: $\Sigma\, deg(v) = 2m$.

İspat. Dereceleri toplarken her kenarı kaç kez saydığımıza bakalım. *(u, v)* kenarı *deg(u)*'ya 1, *deg(v)*'ye 1 katkı yapar; başka hiçbir düğümün derecesini etkilemez. Öz döngüde *u = v*'dir ve tanım gereği yine toplam 2 katkı yapar. Demek ki her kenar toplama tam 2 ekler; *m* kenar toplamı *2m* yapar. □

Adı, bir toplantıdaki el sıkışmalardan gelir: herkes kaç kişiyle el sıkıştığını söylerse bu sayıların toplamı, el sıkışma sayısının iki katıdır. Çünkü her el sıkışmada iki el vardır. Lemmanın iki pratik sonucu vardır:

- Dereceler biliniyorsa kenar sayısı hemen bulunur: $m = \Sigma\, deg(v) \,/\, 2$. Örneğin derecelerin toplamı 30 olan bir çizgede 15 kenar vardır.
- Toplam her zaman çifttir. Bu yüzden tek dereceli düğümlerin sayısı da her zaman çifttir. Çift dereceler toplamı çift yapar; tek derecelerin toplamının da çift olması için bunlardan çift sayıda bulunmalıdır. Dereceleri 3, 3, 2, 2, 1 olan bir çizge kurulamaz: toplam 11 tektir.

Sık yapılan hata

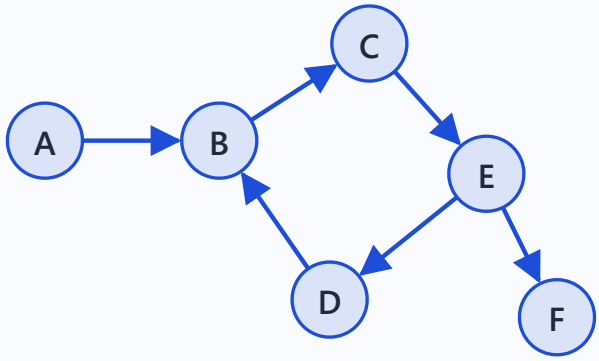
Kenar sayısını bulurken dereceleri toplayıp ikiye bölmeyi unutmak. Her kenar iki ucunda birer kez sayıldığı için toplam, kenar sayısının iki katıdır. Aynı yanılgı bellek hesaplarında da görülür: yönsüz bir çizgenin komşuluk listelerinde toplam *m* değil, *2m* kayıt bulunur.

14.1.4 Yönlü çizge

Bazı ilişkiler simetriktir. A, B'nin arkadaşıysa B de A'nın arkadaşıdır; iki şehir arasındaki çift yönlü yol her iki yönde kullanılır. Bazı ilişkiler ise tek yönlüdür. Bir web sayfası başka bir sayfaya bağlantı verebilir ama o sayfa geri bağlantı vermeyebilir. Bir ders başka bir dersin ön koşuludur, tersi doğru değildir. Tek yönlü sokakta yalnız bir yönde gidilir. Bu ilişkiler **yönlü çizgeyle** (*directed graph*, kısaca *digraph*) modellenir.

Tanım 14.3

Yönlü çizgede her kenar sıralı bir *(u, v)* çiftidir ve *u*'dan *v*'ye giden bir ok olarak çizilir. *u* kenarın **kuyruğu** (başlangıcı), *v* ise **başıdır** (hedefi). Yönlü çizgede *(u, v) ≠ (v, u)*'dur. Bir düğümün **giriş derecesi** (*in-degree*) ona gelen okların, **çıkış derecesi** (*out-degree*) ondan çıkan okların sayısıdır.



Şekil 14.5: Altı düğümlü yönlü bir çizge. Oklar A → B, B → C, C → E, E → D, D → B ve E → F'dir.

Düğüm	A	B	C	D	E	F	Toplam
giriş derecesi	0	2	1	1	1	1	6
çıkış derecesi	1	1	1	1	2	0	6

Tablo 14.1: Şekil 14.5'teki çizgenin giriş ve çıkış dereceleri.

Tablodaki değerler oklar tek tek sayılarak bulunur. B'ye iki ok gelir: A'dan ve D'den. Bu yüzden B'nin giriş derecesi 2'dir. E'den iki ok çıkar: D'ye ve F'ye. Bu yüzden E'nin çıkış derecesi 2'dir. A'ya hiç ok gelmez; F'den hiç ok çıkmaz.

Satır toplamlarının ikisi de 6'dır ve çizgedeki kenar sayısına eşittir. Bu, el sıkışma lemmasının yönlü karşılığıdır: her ok kuyruğundaki düğümün çıkış derecesine 1, başındaki düğümün giriş derecesine 1 ekler. Dolayısıyla Σ *giriş derecesi* $= \Sigma$ *çıkış derecesi* $= m$'dir. Yönsüz çizgedeki $2m$'nin yerini burada iki ayrı toplam alır.

Yönlü çizgede ulaşılabilirlik de yöne bağlıdır. Şekil 14.5'te A'dan B'ye bir ok vardır, ama B'den A'ya hiçbir yol yoktur: A'ya hiç ok gelmez. Öte yandan B'den C'ye, oradan E'ye, D'ye ve yeniden B'ye gidilebilir. Yani B, C, E, D birbirine ulaşabilir; A ve F bu gruba dışarıdan bağlıdır.

Sık yapılan hata

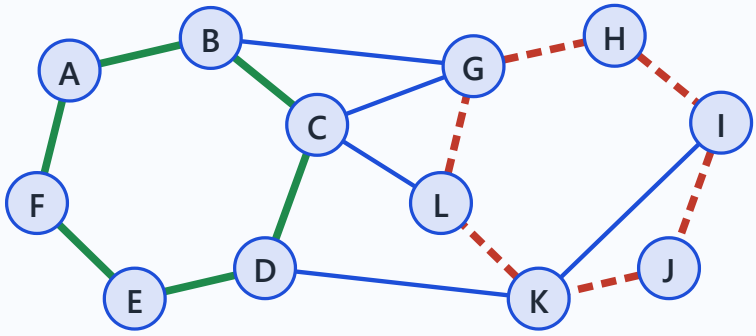
Yönlü çizgede komşuluğu iki yönlü sanmak. *(u, v)* kenarı varsa *v*, *u*'nun komşusudur; ama *(v, u)* kenarı yoksa *u*, *v*'nin komşusu değildir. Bu, "*v*'den *u*'ya hiç gidilemez" demek de değildir: *v*'den başka düğümler üzerinden *u*'ya dolaylı bir yol bulunabilir.

14.1.5 Yol ve döngü

Tanım 14.4

Bir **yol** (*path*), ardışık her iki düğümü bir kenarla bağlı olan bir düğüm dizisidir: $v_0, v_1, \ldots, v_k$ ve her i için (v_i, v_{i+1}) bir kenardır. Hiçbir düğümü tekrar etmeyen yola **basit yol** denir. Aynı düğümde başlayıp biten yola **döngü** (*cycle*) denir. Başlangıç düğümünün sonda tekrar gelmesi dışında hiçbir düğümü tekrar etmeyen döngü **basit döngüdür**.

Yolun **uzunluğu**, ağırlıksız çizgede üzerindeki kenar sayısıdır; $k + 1$ düğümlü bir yol k kenar uzunluğundadır. Yönlü çizgede yol, okların yönünde ilerlemelidir; böyle bir yola **yönlü yol**, yönlü bir döngüye de **yönlü döngü** denir.



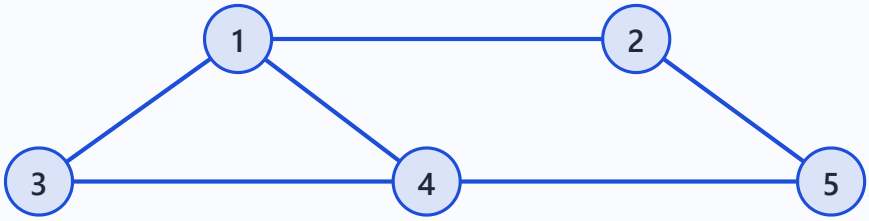
Şekil 14.6: On iki düğümlü bir çizgede iki basit döngü. Yeşil döngü A-B-C-D-E-F-A, kırmızı kesikli döngü G-H-I-J-K-L-G'dir. İki döngü B–G, C–G, C–L ve D–K kenarlarıyla birbirine bağlıdır; kırmızı döngünün içinde ayrıca K–I kenarı vardır.

Şekil 14.6'daki iki döngünün her biri altı kenar uzunluğundadır. Bunlar çizgedeki tek döngüler değildir. Örneğin C → G → L → C de bir döngüdür; C–G, G–L ve L–C kenarları vardır. Bir çizgedeki döngü sayısı, kenar sayısı arttıkça hızla büyür.

Ağaçta döngü yoktur. Ağaçta iki düğüm arasında tek bir basit yol bulunur. Çizgede ise iki düğüm arasında birden çok basit yol olabilir; iki farklı basit yol birleşince bir döngü oluşur. Uçuş rotalarıyla bir örnek verelim. İstanbul → Ankara → İzmir → İstanbul bir döngüdür. Uçak arada aynı şehre ikinci kez uğramadığı için bu basit bir döngüdür. Rota İstanbul → Ankara → İzmir → Ankara → İstanbul olsaydı Ankara iki kez geçerdi; döngü yine döngüdür ama basit değildir.

14.1.6 İki örnek çizge

Kavramları küçük bir yönsüz ve küçük bir yönlü çizge üzerinde birlikte uygulayalım.



Şekil 14.7: Yönsüz örnek çizge: *V = {1, 2, 3, 4, 5}*, *E = {(1,2), (1,3), (1,4), (2,5), (3,4), (4,5)}*.

Örnek 14.2

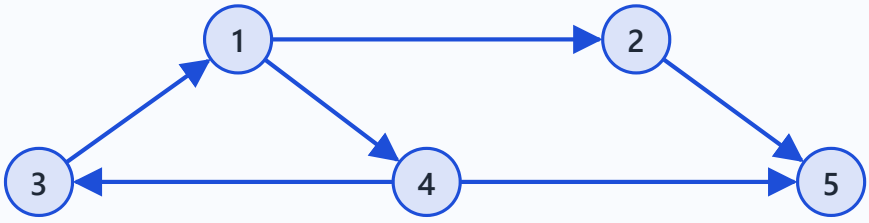
Şekil 14.7'deki çizge için:

1. **Komşuluk.** 1 ile 2 komşudur, çünkü *(1,2)* bir kenardır. 2 ile 3 komşu değildir.
2. **Dereceler.** 1'in komşuları 2, 3, 4'tür: *deg(1) = 3*. 4'ün komşuları 1, 3, 5'tir: *deg(4) = 3*. 2, 3 ve 5'in ikişer komşusu vardır. Toplam *3 + 2 + 2 + 3 + 2 = 12 = 2 · 6*; lemma doğrulanır.
3. **Yol türleri.** Aşağıdaki tablo dört diziyi sınıflandırır.

Dizi	Türü	Gerekçe
1 → 2 → 5	basit yol	1–2 ve 2–5 kenarları var; düğüm tekrarı yok
1 → 3 → 4 → 1	basit döngü	1'de başlayıp 1'de biter; arada tekrar yok
1 → 3 → 4 → 1 → 2	basit olmayan yol	kenarların hepsi var, ama 1 iki kez geçiyor
1 → 3 → 4 → 1 → 4 → 1	basit olmayan döngü	1'de başlayıp 1'de biter, ama 1 ve 4 arada tekrar ediyor

Tablo 14.2: Şekil 14.7'deki çizgede dört düğüm dizisi.

Bir dizinin yol olup olmadığını anlamak için ardışık her çiftin arasında kenar olup olmadığına bakılır. Örneğin 2 → 5 → 3 bir yol değildir, çünkü 5–3 kenarı yoktur. Basitlik ise ikinci bir denetimdir: önce dizi yol olmalı, sonra düğüm tekrarına bakılır.



Şekil 14.8: Yönlü örnek çizge: *V = {1, 2, 3, 4, 5}*, *E = {(1,2), (1,4), (2,5), (4,5), (3,1), (4,3)}*.

Örnek 14.3

Şekil 14.8'deki yönlü çizgede:

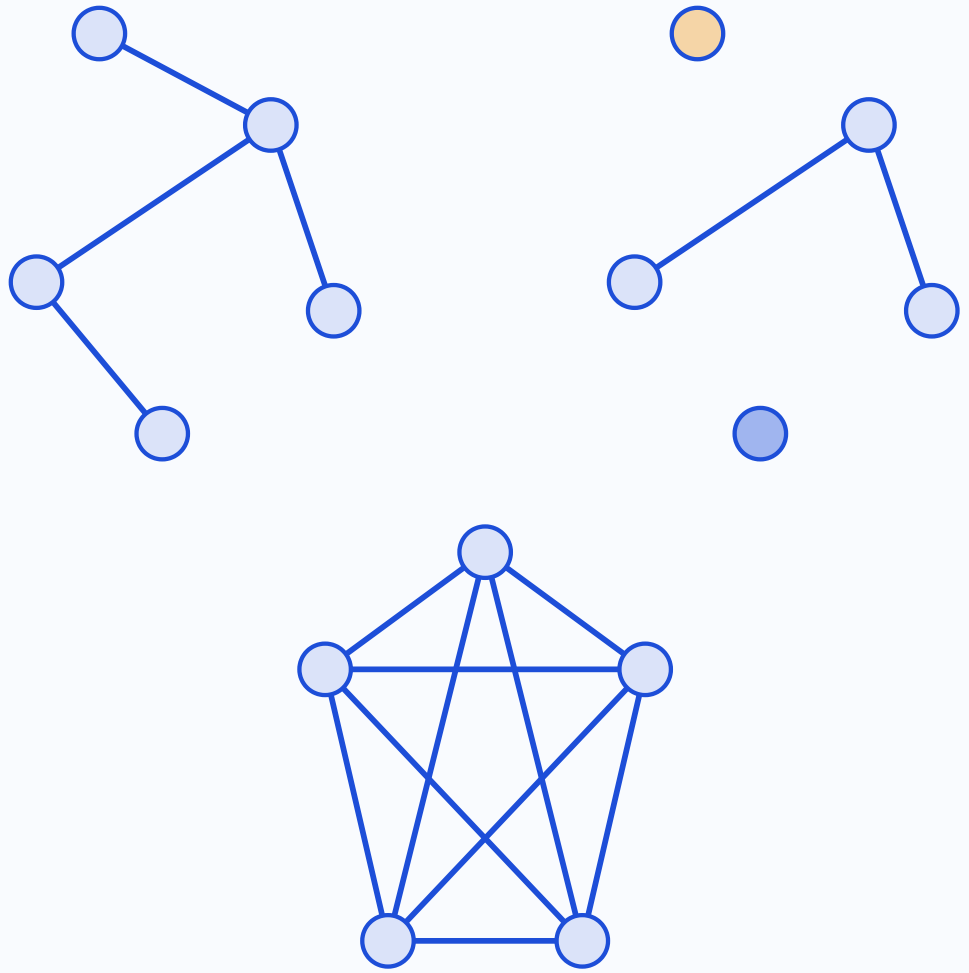
- 2, 1'in komşusudur, çünkü 1 → 2 oku vardır. 1 ise 2'nin komşusu değildir: 2 → 1 oku yoktur.
- 1 → 2 → 5 bir yönlü yoldur.
- 1 → 4 → 3 → 1 bir yönlü döngüdür: 1 → 4, 4 → 3 ve 3 → 1 okları vardır.
- 5'ten çıkan hiçbir ok yoktur. 5'ten başlayan her yol tek düğümden ibarettir; 5'ten başka hiçbir düğüme ulaşılamaz.

14.1.7 Bağlantılı çizge ve tam çizge

Tanım 14.5

Yönsüz bir çizgede her iki farklı düğüm arasında bir yol varsa çizge **bağlantılıdır** (*connected*). Birbirine yolla ulaşabilen düğümlerin en büyük grubuna **bağlantılı bileşen** (*connected component*) denir. Her iki farklı düğüm arasında doğrudan bir kenar varsa çizge **tam çizgedir** (*complete graph*); n düğümlü tam çizge K_n ile gösterilir.

İki tanım arasındaki farka dikkat etmek gerekir. Bağlantılılık *yol* ister, tamlık *kenar* ister. Tam çizge her zaman bağlantılıdır, ama bağlantılı bir çizge tam olmak zorunda değildir. Bileşen tanımındaki “en büyük” sözcüğü de önemlidir: bir bileşene, ondaki herhangi bir düğüme yolla ulaşabilen her düğüm dahildir. Bileşenler çizgenin düğümlerini çakışmayan gruplara ayırır. Bağlantılı bir çizgenin tek bir bileşeni vardır.



Şekil 14.9: Soldan sağa: beş düğümün dört kenarla tek parça olduğu bağlantılı bir çizge (1 bileşen); bir kenarla bağlı iki düğüm ve iki yalnız düğümden oluşan bağlantısız bir çizge (3 bileşen); beş düğümlü tam çizge K_5 (10 kenar).

Kenar sayısı için iki sınır vardır. Basit (öz döngüsüz ve çift kenarsız) yönsüz bir çizgede her düğüm çifti en çok bir kenarla bağlanır. Farklı düğüm çiftlerinin sayısı *n(n − 1)/2*'dir: ilk düğüm *n* biçimde, ikinci düğüm *n − 1* biçimde seçilir ve her çift iki kez sayıldığı için 2'ye bölünür. Bu yüzden *m ≤ n(n − 1)/2*'dir ve eşitlik yalnız tam çizgede sağlanır. K_5'te *5 · 4 / 2 = 10*, 10 düğümlü tam çizgede *10 · 9 / 2 = 45* kenar vardır.

Alt sınır bağlantılılıktan gelir. Bağlantılı bir çizgede *m ≥ n − 1*'dir. Bunu görmek için kenarları boş bir çizgeye tek tek eklediğimizi düşünelim. Başta *n* bileşen vardır; her yalnız düğüm ayrı bir bileşendir. Bir kenar eklemek bileşen sayısını en çok 1 azaltır: ya iki bileşeni birleştirir ya da aynı bileşenin içine düşer. Tek bileşene inmek için en az *n − 1* kenar gerekir. Tam *n − 1* kenarlı bağlantılı çizge bir ağaçtır; bu gözlem kapsayan ağaçta yeniden karşımıza çıkar (14.5.1).

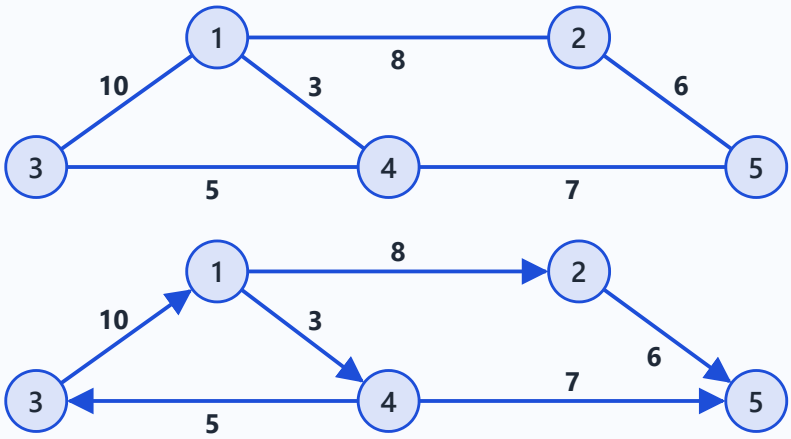
Bu iki sınır, **seyrek** ve **yoğun** çizge ayrımının da temelidir. Kenar sayısı n mertebesindeyse ($m \approx n$) çizge seyrektir; n^2 mertebesindeyse ($m \approx n^2$) yoğundur. Yol ağları ve sosyal ağlar seyrektir. Herkesin herkesle doğrudan bağlantılı olduğu küçük gruplar yoğundur.

14.1.8 Ağırlıklı çizge

Pek çok problemde yalnız "bağlı mı?" sorusu yetmez; bağlantının bir bedeli vardır. İki şehir arasındaki yolun uzunluğu, bir ağ bağlantısının gecikmesi, bir boru hattının maliyeti ya da bir hattın kapasitesi bu türden bilgilerdir.

Tanım 14.6

Ağırlıklı çizgede (*weighted graph*) her kenara bir sayı atanır; bu sayıya kenarın **ağırlığı** (*weight*) denir. Bir yolun **uzunluğu** ya da **ağırlığı**, üzerindeki kenar ağırlıklarının toplamıdır. Ağırlıklı çizge yönsüz de yönlü de olabilir.



Şekil 14.10: Aynı beş düğüm üzerinde ağırlıklı yönsüz (üstte) ve ağırlıklı yönlü (altta) çizge. Yönsüz çizgede kenarlar 1–2: 8, 1–3: 10, 1–4: 3, 2–5: 6, 3–4: 5, 4–5: 7'dir. Yönlü çizgede oklar 1 → 2: 8, 1 → 4: 3, 2 → 5: 6, 4 → 5: 7, 3 → 1: 10, 4 → 3: 5'tir.

Örnek 14.4

Şekil 14.10'daki yönsüz çizgede 3 → 4 → 1 yolunun uzunluğu *5 + 3 = 8*'dir. Yönlü çizgede 1 → 4 → 3 yolunun uzunluğu *3 + 5 = 8*'dir.

Yönsüz çizgede 1 ile 3'ü doğrudan bağlayan bir kenar vardır ve ağırlığı 10'dur. Oysa 1 → 4 → 3 yolu yalnız 8 tutar. Doğrudan kenar, iki düğüm arasındaki en kısa yol olmak zorunda değildir. Daha çok kenardan geçen bir yol, kenarları hafifse daha kısa olabilir.

Bu gözlem, en kısa yol algoritmalarının neden gerektiğini açıklar. Ağırlıksız çizgede en kısa yol en az kenarlı yoldur. Ağırlıklı çizgede ise kenar sayısı yanıltıcıdır; ağırlıkların toplamına bakmak gerekir. Ağırlıksız çizgeyi, her kenarın ağırlığının 1 olduğu ağırlıklı çizge olarak da düşünebiliriz.

14.1.9 Çizge ADT'si

Çizge de öteki veri yapıları gibi önce bir **soyut veri tipi** (*abstract data type*, ADT) olarak tanımlanabilir. ADT, hangi işlemlerin bulunduğunu ve ne yaptıklarını söyler; bu işlemlerin bellekte nasıl yürütüleceğini söylemez. Goodrich, Tamassia ve Goldwasser'in kitabındaki `Graph` arayüzü, aşağıdaki temel işlemleri içerir.

İşlem	Açıklama
`numVertices()`, `numEdges()`	Düğüm sayısı n, kenar sayısı m
`vertices()`, `edges()`	Bütün düğümler / bütün kenarlar üzerinde gezinme
`getEdge(u, v)`	u'dan v'ye kenar varsa onu, yoksa `null` döndürür
`endVertices(e)`, `opposite(v, e)`	Kenarın iki ucu / e kenarının v dışındaki ucu
`outDegree(v)`, `inDegree(v)`	Çıkış / giriş derecesi (yönsüz çizgede ikisi eşittir)
`outgoingEdges(v)`, `incomingEdges(v)`	v'den çıkan / v'ye gelen kenarlar
`insertVertex(x)`, `insertEdge(u, v, x)`	Yeni düğüm / yeni kenar ekler (x: saklanacak değer)
`removeVertex(v)`, `removeEdge(e)`	Düğümü bağlı kenarlarıyla birlikte / kenarı siler

Tablo 14.3: Çizge ADT'sinin temel işlemleri (V: düğüm, E: kenar).

Algoritmaların en çok kullandığı iki işlem öne çıkar. Birincisi `getEdge(u, v)`, yani "u ile v komşu mu?" sorusudur. İkincisi `outgoingEdges(v)`, yani "v'nin komşuları kimler?" sorusudur. BFS, DFS, Dijkstra ve Prim gibi algoritmalar ikinci işlemi her düğüm için bir kez çağırır. Bu yüzden bir gösterimin komşuları ne kadar hızlı verdiği, algoritmaların toplam süresini doğrudan belirler.

`removeVertex(v)`'nin bağlı kenarları da silmesi bir tutarlılık kuralıdır. Ucu olmayan bir kenar anlamsız olur; bu yüzden düğümle birlikte ona bağlı bütün kenarlar da çizgeden çıkar. Bu işlemlerin maliyeti çizgenin bellekte nasıl temsil edildiğine bağlıdır.

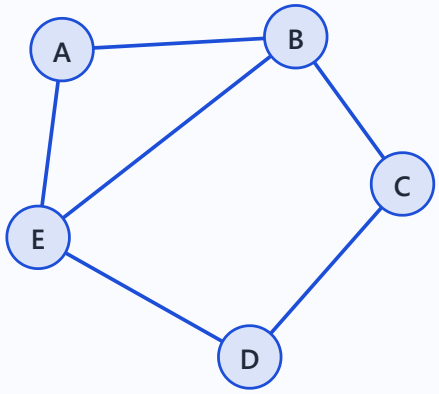
14.2 Çizge gösterimleri

Bir çizgeyi bilgisayarda saklamanın tek bir doğru yolu yoktur. Aynı çizge üç farklı biçimde tutulabilir ve her biçim bazı işlemleri hızlandırırken bazılarını yavaşlatır. Bu kesimde üç temel gösterimi sırayla ele alıyor, sonra bunları bellek ve işlem maliyetleri açısından karşılaştırıyoruz. Kesimin sonunda komşuluk listesinin ve matrisinin Java ile nasıl kurulduğunu satır satır izliyoruz.

14.2.1 Üç temel gösterim ve kenar listesi

Çizgeyi saklamanın üç temel yolu şunlardır:

- **Kenar listesi** (*edge list*): bütün kenarlar tek bir listede tutulur, örneğin `[(A,B), (A,E), …]`.
- **Komşuluk listesi** (*adjacency list*): her düğüm için o düğümün komşularının listesi tutulur, örneğin `V[i] = {j, k, l}`.
- **Komşuluk matrisi** (*adjacency matrix*): $n \times n$ boyutlu bir dizi tutulur; `M[i][j]` hücresi i'den j'ye kenar olup olmadığını gösterir.



Şekil 14.11: Gösterimlerin karşılaştırılmasında kullanılan beş düğümlü yönsüz çizge. Kenarlar A–B, A–E, B–C, B–E, C–D ve D–E'dir.

Kenar listesi en basit gösterimdir. Şekil 14.11'deki çizge için liste `(A, B) (A, E) (B, C) (B, E) (C, D) (D, E)` biçimindedir. Kenarı olmayan düğümleri kaybetmemek için düğümler de ayrı bir listede tutulur; böylece bellek $O(n + m)$ olur. Yeni bir kenar eklemek listenin sonuna bir öğe eklemektir ve $O(1)$ sürer.

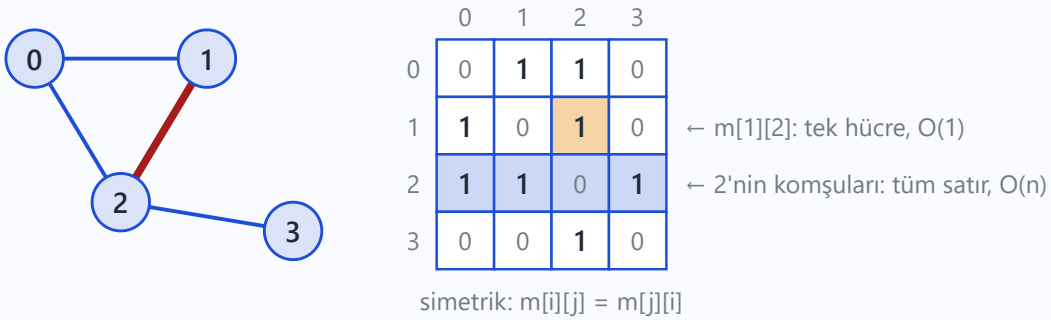
Kenar listesinin zayıflığı sorgulardadır. "u ile v komşu mu?" sorusunu yanıtlamak için listede (u, v) ya da (v, u) aranır. Liste hiçbir düzene göre sıralı olmadığı için, en kötü durumda bütün liste taranır: $O(m)$. "u'nun komşuları kim?" sorusu da aynı biçimde tüm listeyi taramayı gerektirir, çünkü u'ya bağlı kenarlar listenin herhangi bir yerinde olabilir. Gezinme algoritmaları bu soruyu her düğüm için sorduğundan, kenar listesiyle bir gezinme $O(n \cdot m)$'ye çıkar. Kenar listesi yine de bir algoritmada işe yarar: Kruskal algoritması kenarları ağırlığa göre sıralayıp tek tek dener ve komşu sorgusu yapmaz (14.5.3).

14.2.2 Komşuluk matrisi

Tanım 14.7

Düğümleri 0'dan $n - 1$'e numaralanmış bir çizgenin **komşuluk matrisi**, $n \times n$ boyutlu bir `matris` dizisidir. i'den j'ye kenar varsa `matris[i][j] = 1`, yoksa 0'dır. Ağırlıklı çizgede hücre, kenarın ağırlığını tutar; kenar yoksa ∞ ya da bu anlama gelen özel bir değer yazılır.

Yönsüz çizgede her kenar iki yönde de geçerlidir. *(i, j)* kenarı hem `matris[i][j]` hem de `matris[j][i]` hücresine yazılır. Bu yüzden matris ana köşegene göre **simetriktir**: her *i*, *j* için `matris[i][j] == matris[j][i]`'dir. Yönlü çizgede bu simetri yoktur.



Şekil 14.12: Dört düğümlü bir çizge (kenarlar 0–1, 0–2, 1–2, 2–3) ve komşuluk matrisi. `m[1][2]` hücresine tek erişimle bakılır: *O(1)*. 2'nin komşularını bulmak için 2. satırın tamamı okunur: *O(n)*.

Matrisin güçlü yanı doğrudan erişimdir. Java'da iki boyutlu bir dizinin herhangi bir hücresi, indisleri bilindiği sürece sabit sürede okunur ve yazılır. Bu yüzden üç işlem *O(1)*'dir:

- "*i* ile *j* arasında kenar var mı?" sorusu, `matris[i][j]` hücresini okumaktır.
- Kenar eklemek, bir (yönsüzde iki) hücreye 1 yazmaktır.
- Kenar silmek, aynı hücrelere 0 yazmaktır.

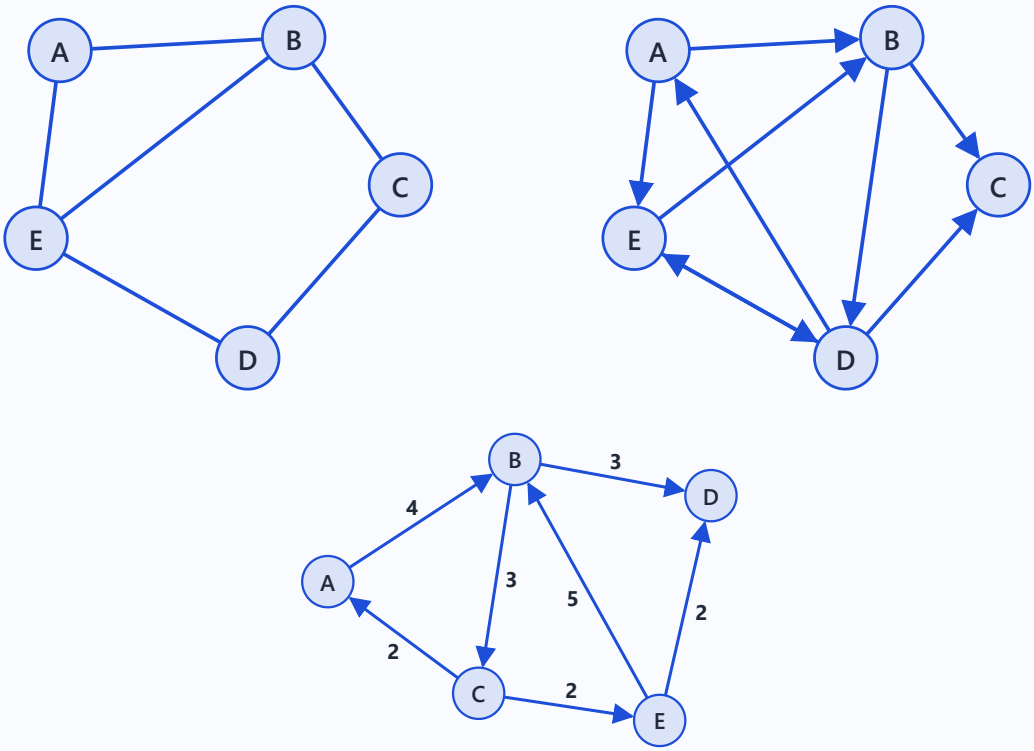
Zayıf yanları da iki tanedir. Birincisi, bir düğümün komşularını bulmak için o düğümün satırının *n* hücresinin hepsine bakmak gerekir. Düğümün tek bir komşusu olsa bile süre *O(n)*'dir. İkincisi bellektir: matris, kenar sayısından bağımsız olarak n^2 hücre kaplar. Seyrek bir çizgede hücrelerin büyük çoğunluğu 0'dır; bellek, var olmayan kenarları kaydetmeye harcanır.

Örnek 14.5

Bir milyon düğümlü ($n = 10^6$) bir yol ağının komşuluk matrisi $n^2 = 10^{12}$ hücre ister. Hücre başına yalnız 1 bit ayrılsa bile bu yaklaşık 125 GB eder. Oysa her kavşakta ortalama üç dört yol birleşir; kenar sayısı birkaç milyon mertebesindedir. Dolu hücrelerin oranı milyonda birkaç düzeyinde kalır.

14.2.3 Komşuluk matrisi: üç örnek

Aynı beş düğüm (A–E) üzerinde yönsüz, yönlü ve ağırlıklı üç çizgenin matrislerini karşılaştıralım. Satırlar ve sütunlar A, B, C, D, E sırasıyla dizilmiştir.



Şekil 14.13: Soldan sağa: yönsüz çizge (Şekil 14.11'deki çizge); dokuz oklu yönlü çizge (A → B, A → E, B → C, B → D, D → A, D → C, D → E, E → B, E → D); ağırlıklı yönlü çizge (A → B: 4, B → C: 3, B → D: 3, C → A: 2, C → E: 2, E → B: 5, E → D: 2).

	A	B	C	D	E
A	0	1	0	0	1
B	1	0	1	0	1
C	0	1	0	1	0
D	0	0	1	0	1
E	1	1	0	1	0

Tablo 14.4: Yönsüz çizgenin komşuluk matrisi.

Yönsüz matriste her kenar iki hücreye yazılmıştır; altı kenar 12 tane 1 verir. Matris köşegene göre simetriktir: örneğin A satırının E sütunu ile E satırının A sütunu ikisi de 1'dir. Bir satırın toplamı, o düğümün derecesidir. B satırında üç tane 1 vardır (A, C, E), dolayısıyla *deg(B) = 3*'tür. Köşegen 0'dır, çünkü çizgede öz döngü yoktur.

	A	B	C	D	E
A	0	1	0	0	1
B	0	0	1	1	0
C	0	0	0	0	0
D	1	0	1	0	1

	A	B	C	D	E
E	0	1	0	1	0

Tablo 14.5: Yönlü çizgenin komşuluk matrisi (satır = kaynak, sütun = hedef).

Yönlü matriste her ok yalnız bir hücreye yazılır: dokuz ok, dokuz tane 1. Satır kaynağı, sütun hedefi gösterir; bu yüzden matris simetrik değildir. A → B oku `m[A][B] = 1` yapar, ama B → A oku olmadığı için `m[B][A] = 0`'dır. Satır toplamı çıkış derecesini, sütun toplamı giriş derecesini verir. D satırında üç tane 1 vardır: D'nin çıkış derecesi 3'tür. C satırının tamamı 0'dır: C'den hiç ok çıkmaz. C sütununda ise iki tane 1 vardır (B ve D'den): C'nin giriş derecesi 2'dir.

	A	B	C	D	E
A	0	4	∞	∞	∞
B	∞	0	3	3	∞
C	2	∞	0	∞	2
D	∞	∞	∞	0	∞
E	∞	5	∞	2	0

Tablo 14.6: Ağırlıklı yönlü çizgenin komşuluk matrisi (kenar yoksa ∞, köşegende 0).

Ağırlıklı matriste hücre, kenarın ağırlığını tutar. A satırının B sütunundaki 4, A → B okunun ağırlığıdır. Kenar olmayan hücrelere ∞ yazılır. Köşegende ise 0 vardır: bir düğümün kendisine uzaklığı sıfırdır. Bu gösterim en kısa yol problemlerinde kullanışlıdır, çünkü ∞ "doğrudan gidilemez" anlamına gelir ve toplamalarda kendiliğinden doğru davranır: ∞ artı herhangi bir sayı yine ∞'dur.

Sık yapılan hata

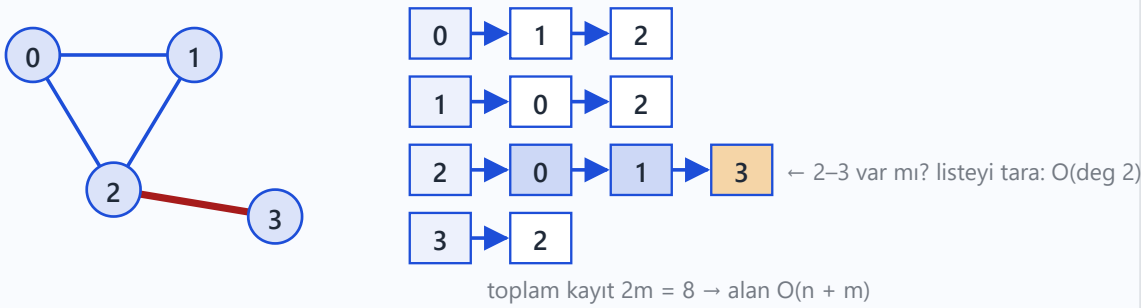
Ağırlıklı matriste kenar olmayan hücreye 0 yazmak. 0, ağırlığı 0 olan gerçek bir kenarla karışır; algoritma var olmayan bir kenarı "bedava yol" sanır. Java bir `int[][]` dizisini oluştururken bütün hücreleri 0 ile başlattığı için bu hata kendiliğinden ortaya çıkar. Kenarsız hücreler açıkça `Integer.MAX_VALUE` gibi bir değerle doldurulmalıdır. Köşegendeki 0 ise kendine uzaklık olarak anlamlıdır ve doğrudur.

14.2.4 Komşuluk listesi

Tanım 14.8

Komşuluk listesi gösteriminde her düğüm için bir liste tutulur; n düğüm için n liste vardır. `liste[i]`, i'nin komşularını (ağırlıklı çizgede ağırlıklarıyla birlikte) içerir. Listeler çoğunlukla bağlı liste ya da dinamik dizi olarak gerçeklenir.

Yönsüz çizgede bir (v, w) kenarı iki listeye birden eklenir: w, v'nin listesine; v, w'nin listesine. Yönlü çizgede ise yalnız kaynağın listesine eklenir. Listede yalnız var olan kenarlar yer aldığı için bellek n liste başı ile toplam kayıt sayısının toplamıdır. Yönsüz çizgede kayıt sayısı $2m$, yönlü çizgede m'dir. Her iki durumda bellek $O(n + m)$'dir.



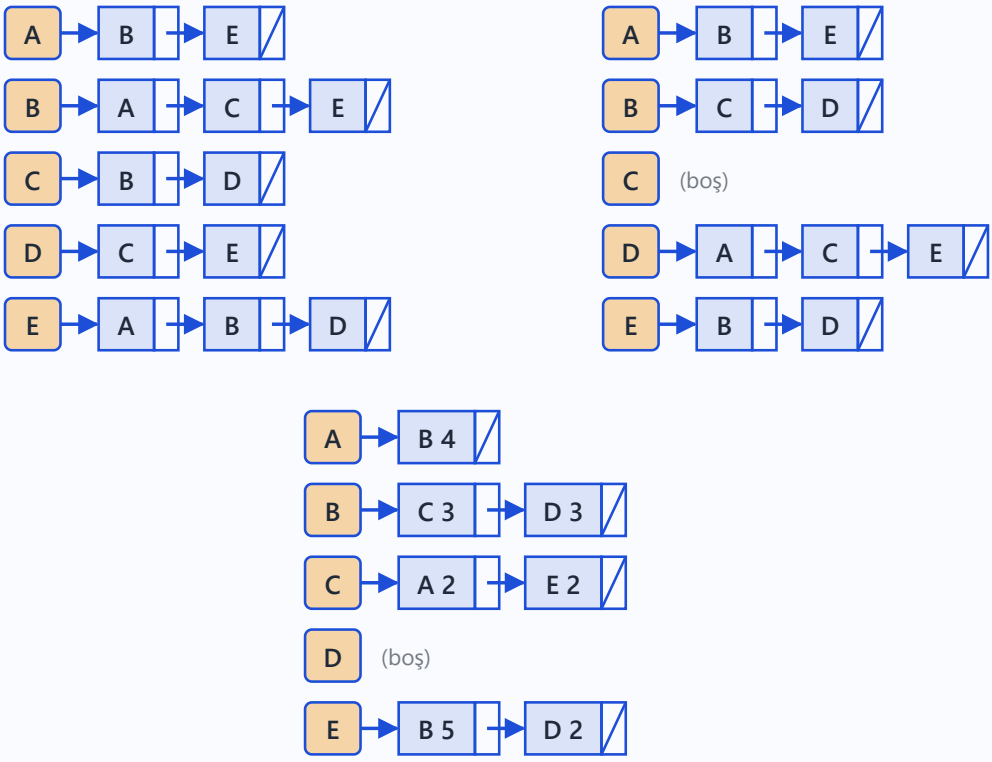
Şekil 14.14: Şekil 14.12'deki dört düğümlü çizgenin komşuluk listesi: 0: 1, 2 · 1: 0, 2 · 2: 0, 1, 3 · 3: 2. "2–3 kenarı var mı?" sorusu için 2'nin listesi taranır: $O(\deg 2)$. Toplam kayıt $2m = 8$'dir.

Komşuluk listesinin güçlü yanı gezinmedir. Bir düğümün komşularını gezmek, onun listesini baştan sona okumaktır ve $O(\deg(v))$ sürer. Matristeki gibi komşu olmayan düğümlere boşuna bakılmaz. Kenar eklemek de listenin sonuna bir öğe eklemektir; dinamik dizide amortize $O(1)$, bağlı listede $O(1)$'dir.

Zayıf yanı, "u ile v arasında kenar var mı?" sorgusudur. Listeler sıralı tutulmadığı sürece v'yi bulmak için u'nun listesi taranır: $O(\deg(u))$. Yönsüz çizgede iki listeden kısa olanı seçilebilir ve süre $O(\min(\deg u, \deg v))$ olur. Bir düğümün derecesi büyükse bu sorgu matristeki $O(1)$'e göre yavaştır.

14.2.5 Komşuluk listesi: üç örnek

Şekil 14.13'teki üç çizgenin komşuluk listeleri aşağıdadır. Komşular alfabetik sırayla yazılmıştır.



Şekil 14.15: Soldan sağa: yönsüz çizgenin listeleri (A: B, E · B: A, C, E · C: B, D · D: C, E · E: A, B, D); yönlü çizgenin listeleri (A: B, E · B: C, D · C boş · D: A, C, E · E: B, D); ağırlıklı yönlü çizgenin listeleri (A: (B, 4) · B: (C, 3), (D, 3) · C: (A, 2), (E, 2) · D boş · E: (B, 5), (D, 2)).

- **Yönsüz.** Altı kenar listelerde 12 kayıt oluşturur; her kenar iki kez yazılır. Bir listenin uzunluğu, düğümün derecesidir: B'nin listesi üç öğelidir ve *deg(B) = 3*'tür.
- **Yönlü.** Yalnız çıkan kenarlar listelenir. Dokuz ok dokuz kayıt verir. C'den çıkan ok olmadığı için C'nin listesi boştur. Liste uzunluğu burada çıkış derecesidir. Giriş derecesini bulmak için bütün listeleri taramak gerekir; bu, listenin yönlü çizgedeki bir eksiğidir.
- **Ağırlıklı.** Her kayıt (komşu, ağırlık) çifti tutar. B'nin listesi `(C, 3), (D, 3)`'tür. Java'da bu çift `int[]{v, w}` dizisiyle ya da küçük bir `Kenar` sınıfıyla temsil edilir.

14.2.6 Gösterimlerin maliyeti

İşlem	Kenar listesi	Komşuluk listesi	Komşuluk matrisi
Bellek	O(n + m)	O(n + m)	O(n^2)
u–v kenarı var mı? (`getEdge`)	O(m)	yönsüz: O(min(deg u, deg v)); yönlü u → v: O(outDeg u)	**O(1)**
v'nin komşularını gez	O(m)	**O(deg v)**	O(n)
Kenar ekle	O(1)	O(1)	O(1)

Tablo 14.7: Üç gösterimin bellek ve işlem maliyetleri (*n* düğüm, *m* kenar).

Tablo, seçimin çizgenin yapısına ve en sık yapılan işleme göre verilmesi gerektiğini gösterir. İki tipik durum vardır:

- **Seyrek çizge** ($m \approx n$). Yol ağları ve sosyal ağlar bu türdendir. Matris n^2 hücre ister ve hücrelerin neredeyse tamamı boş kalır. Komşuluk listesi hem belleği $O(n + m)$'de tutar hem de gezinmeyi hızlandırır. Bu notta incelenen bütün gezinme ve en kısa yol algoritmaları listeyle $O(n + m)$ ya da ona yakın sürede çalışır.
- **Yoğun çizge** ($m \approx n^2$). Kenarların çoğu varsa matrisin boş hücresi azdır ve bellek kaybı önemsizleşir. "Kenar var mı?" sorgusu sık yapılıyorsa matrisin $O(1)$ erişimi belirleyici olur.

Örnek 14.6

$n = 10.000$ düğüm ve $m = 30.000$ kenarlı bir yol ağı saklanacak olsun. Hangi gösterim seçilmelidir?

1. **Matris.** $n^2 = 10^8$ hücre gerekir. Yönsüz çizgede dolu hücre sayısı $2m = 60.000$'dir; oran $60.000 / 10^8 = \%0{,}06$'dır.
2. **Liste.** n liste başı ve $2m$ kayıt: $10.000 + 60.000 = 70.000$ öğe. Bu sayı $n + m = 40.000$ ile aynı mertebededir, yani bellek $O(n + m)$'dir.
3. **Karar.** Çizge seyrektir ($m = 3n$). Komşuluk listesi seçilir. Gezinme de listede $O(n + m)$, matriste $O(n^2)$ sürer.

Sık yapılan hata

Matristeki satır taramasının maliyetini `getEdge` ile karıştırmak. Matriste "kenar var mı?" tek hücre okumasıdır ve $O(1)$'dir; $O(n)$ olan, bir düğümün bütün komşularını bulmak için satırı taramaktır. Listede ise durum tersinedir: komşuları gezmek ucuz, tek bir kenarı sorgulamak görece pahalıdır.

14.2.7 Java ile komşuluk listesi

Düğümler 0'dan $n - 1$'e numaralanırsa komşuluk listesi, listelerden oluşan bir liste olarak yazılabilir. Dış listenin i. öğesi, i düğümünün komşu listesidir.

```
class Cizge {
    private final int n;                                    // düğümler: 0..n-1
    private final List<List<Integer>> komsu = new ArrayList<>();
    Cizge(int n) {
        this.n = n;
        for (int i = 0; i < n; i++)
            komsu.add(new ArrayList<>());
    }
    void kenarEkle(int u, int v) {
        komsu.get(u).add(v);
        komsu.get(v).add(u);                  // yönlü çizgede bu satır yok
    }
    List<Integer> komsular(int u) { return komsu.get(u); }   // O(1)
    int derece(int u) { return komsu.get(u).size(); }
}
```

Kodun parçaları şunlardır:

- `komsu` alanı `List<List<Integer>>` türündedir. Dış liste düğüm başına bir iç liste tutar; iç listeler `ArrayList`, yani dinamik dizidir.
- Yapıcı, *n*'yi saklar ve döngüyle *n* tane boş iç liste ekler. Bu adım *O(n)* zaman alır. Düğümlerin hepsi baştan bilindiği için listeler önceden açılır; böylece `komsu.get(u)` her *u* için geçerli bir liste döndürür.
- `kenarEkle(u, v)` iki satırdır: *v*'yi *u*'nun listesine, *u*'yu *v*'nin listesine ekler. Yönlü çizgede ikinci satır yazılmaz. `ArrayList.add` amortize *O(1)*'dir; metot da öyledir.
- `komsular(u)` listenin kendisini döndürür; kopyalama yapılmadığı için *O(1)*'dir. Listenin gezilmesi ise çağıranın işidir ve *O(deg(u))* sürer.
- `derece(u)`, listenin boyutudur. `ArrayList` boyutunu bir alanda tuttuğu için bu da *O(1)*'dir.

Matris karşılığı daha kısadır: `boolean[][] m = new boolean[n][n];` diziyi oluşturur, `m[u][v] = m[v][u] = true;` kenar ekler, `m[u][v]` sorgular. Ağırlıklı çizgede listenin öğesi `int[]{v, w}` ya da küçük bir `Kenar` sınıfı olur; matris ise `int[][]` olur ve kenarsız hücrelere `Integer.MAX_VALUE` yazılır.

14.2.8 Kod izleme: komşuluk listesi kurma

Beş düğümlü bir çizgeyi altı kenarla kuralım. Kenarlar sırasıyla *{0,1}, {0,4}, {1,2}, {1,4}, {2,3}, {3,4}*'tür. İzlenen kod, sınıfın yapıcısı ile `kenarEkle` metodu ve bunları çağıran `main` parçasıdır.

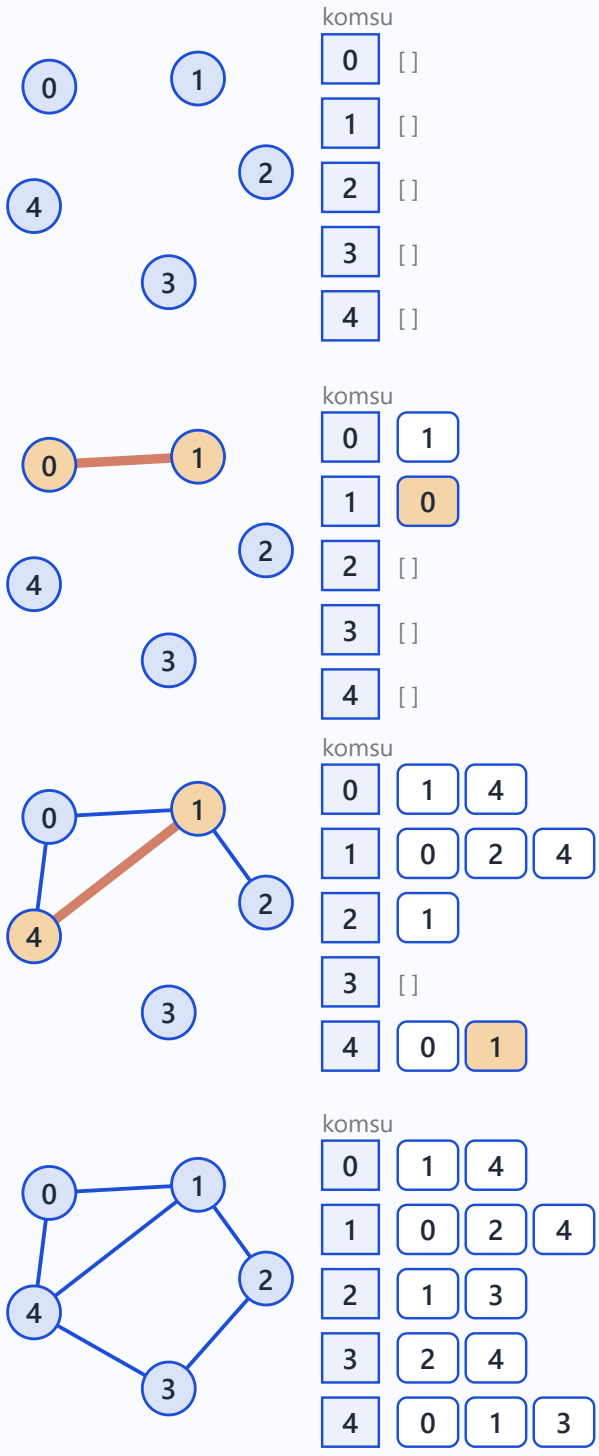
```
Cizge(int n) {
  this.n = n;
  for (int i = 0; i < n; i++)
    komsu.add(new ArrayList<>());
}
void kenarEkle(int u, int v) {
  komsu.get(u).add(v);
  komsu.get(v).add(u);  // yönlüde yok
}
// main:
Cizge g = new Cizge(5);
int[][] E = {{0,1}, {0,4}, {1,2},
             {1,4}, {2,3}, {3,4}};
for (int[] e : E) g.kenarEkle(e[0], e[1]);
```

Adım	0	1	2	3	4
yapıcı (`i = 0 … 4`)	[]	[]	[]	[]	[]
`kenarEkle(0, 1)`	[1]	[0]	[]	[]	[]
`kenarEkle(0, 4)`	[1, 4]	[0]	[]	[]	[0]
`kenarEkle(1, 2)`	[1, 4]	[0, 2]	[1]	[]	[0]
`kenarEkle(1, 4)`	[1, 4]	[0, 2, 4]	[1]	[]	[0, 1]
`kenarEkle(2, 3)`	[1, 4]	[0, 2, 4]	[1, 3]	[2]	[0, 1]
`kenarEkle(3, 4)`	[1, 4]	[0, 2, 4]	[1, 3]	[2, 4]	[0, 1, 3]

Tablo 14.8: `new Cizge(5)` ve altı `kenarEkle` çağrısından sonra komşu listeleri.

İzlemenin ilk aşaması yapıcıdır. `new Cizge(5)` çağrısında `n = 5` atanır; `komsu` henüz boş bir dış listedir. Döngü `i = 0`'dan `i = 4`'e beş tur döner ve her turda boş bir iç liste ekler. Döngü bittiğinde her düğümün boş bir komşu listesi vardır.

İkinci aşama kenarlardır. `kenarEkle(0, 1)`'in ilk satırı 1'i 0'ın listesinin sonuna ekler. İkinci satır aynı kenarı ters yönde işler ve 0'ı 1'in listesine ekler. Sonraki her çağrı aynı iki adımı yapar. Listelerdeki sıra, ekleme sırasıdır: 1'in listesi `[0, 2, 4]` olur, çünkü 0–1, 1–2 ve 1–4 kenarları bu sırayla gelmiştir.



Şekil 14.16: Komşuluk listesi kurulurken dört an. (a) Yapıcıdan sonra beş boş liste. (b) `kenarEkle(0, 1)` 'den sonra 0: [1], 1: [0]. (c) `kenarEkle(1, 4)` 'ten sonra 1: [0, 2, 4], 4: [0, 1]. (d) Son durum: 12 kayıt.

Sonuçta altı kenar, listelerde *2 · 6 = 12* kayıt oluşturur. Liste uzunlukları 2, 3, 2, 2, 3'tür ve toplamları 12'dir; bu da el sıkışma lemmasının kodda görünen biçimidir. `derece(1)`, 1'in listesinin boyutudur ve 3 döndürür. Her `kenarEkle` çağrısı sabit sürelidir; bellek *O(n + m)*'dir.

Sık yapılan hata

Yönsüz çizgede `kenarEkle`'nin ikinci satırını unutmak. Bu durumda çizge sessizce yönlü hâle gelir: 0–1 kenarı 0'ın listesinde görünür ama 1'in listesinde görünmez. Gezinme 1'den başlarsa 0'a hiç ulaşamaz. Kod hata vermez; yanlış sonuç verir. Tersi de bir hatadır: yönlü bir ilişkiyi iki yönlü eklemek, olmayan geri dönüş yolları yaratır.

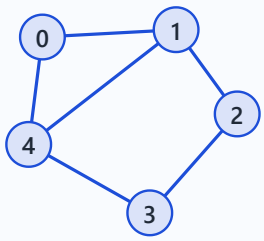
14.2.9 Kod izleme: komşuluk matrisi kurma

Aynı altı kenarı bu kez *5 × 5* boyutlu bir `boolean` matrise ekleyelim.

```
boolean[][] m = new boolean[n][n];
void kenarEkle(int u, int v) {
  m[u][v] = true;
  m[v][u] = true;  // yönlü çizgede yok
}
// main: aynı 6 kenar (n = 5)
for (int[] e : E) g.kenarEkle(e[0], e[1]);
```

`new boolean[n][n]` 25 hücrelik bir matris ayırır. Java, `boolean` dizisinin bütün hücrelerini `false` ile başlatır; başlangıçta hiç kenar yoktur. Matrisin oluşturulması bile $O(n^2)$ sürer, çünkü 25 hücrenin her biri başlangıç değerini almak zorundadır.

Her `kenarEkle(u, v)` iki hücre yazar. `kenarEkle(0, 1)` önce `m[0][1] = true` yapar (satır 0, sütun 1), sonra simetrik hücreyi, `m[1][0]`'ı `true` yapar. Altı kenar sırayla şu hücre çiftlerini doldurur: (0,1)–(1,0), (0,4)–(4,0), (1,2)–(2,1), (1,4)–(4,1), (2,3)–(3,2), (3,4)–(4,3).



	0	1	2	3	4
0	0	**1**	0	0	**1**
1	**1**	0	**1**	0	**1**
2	0	**1**	0	**1**	0
3	0	0	**1**	0	**1**
4	**1**	**1**	0	**1**	0

m[satır][sütun] (1 = true)

Şekil 14.17: Altı kenar eklendikten sonra komşuluk matrisi. 12 hücre `true`, 13 hücre `false`'tur. Matris köşegene göre simetriktir.

	0	**1**	**2**	**3**	**4**
0	0	1	0	0	1
1	1	0	1	0	1
2	0	1	0	1	0

	0	1	2	3	4
3	0	0	1	0	1
4	1	1	0	1	0

Tablo 14.9: Son durumdaki matris (1 = `true`, 0 = `false`).

Bu küçük örnekte bile iki gösterimin farkı görülür. "2–4 kenarı var mı?" sorusu `m[2][4]` hücresine tek erişimle yanıtlanır: `false`, yani kenar yok. Aynı soruyu listeyle yanıtlamak için 2'nin listesi `[1, 3]` taranır. Öte yandan 2'nin komşularını bulmak için matriste 2. satırın beş hücresinin hepsi okunur; listede ise yalnız iki öğe okunur. Matrisin 25 hücresinden 13'ü hiçbir bilgi taşımaz. Çizge büyüdükçe ve seyrekleştikçe bu oran hızla kötüleşir.

14.3 Gezinme: BFS ve DFS

Çizge algoritmalarının çoğu, çizgede sistemli biçimde gezinen bir iskelet üzerine kurulur. Bu iskeletin iki temel türü vardır: genişlik öncelikli arama ve derinlik öncelikli arama. İkisi de aynı düğüm kümesini ziyaret eder; farkları ziyaret sırasındadır. Sıranın farklı olması, her birini farklı problemlere uygun kılar. Kesimin ikinci yarısında bu iki aramanın üzerine kurulan dört uygulamayı ele alıyoruz: bağlantılı bileşenler, iki parçalılık testi, topolojik sıralama ve yönlü döngü tespiti.

14.3.1 Gezinme ve ziyaret işareti

> **Tanım 14.9**
>
> **Gezinme** (*traversal*), bir başlangıç düğümünden ulaşılabilen bütün düğümleri, her birini bir kez olmak üzere, sistemli bir sırayla ziyaret etmektir.

İki temel gezinme yöntemi vardır:

- **Genişlik öncelikli arama** (*breadth-first search*, BFS) önce başlangıcın bütün komşularını, sonra onların komşularını ziyaret eder. Düzey düzey ilerler. Yardımcı yapı olarak **kuyruk** (FIFO: ilk giren ilk çıkar) kullanır.
- **Derinlik öncelikli arama** (*depth-first search*, DFS) bir yol boyunca gidebildiği kadar derine iner. Çıkmaza gelince geri döner ve başka bir dalı dener. Yardımcı yapı olarak **yığın** (LIFO: son giren ilk çıkar) ya da özyineleme kullanır.

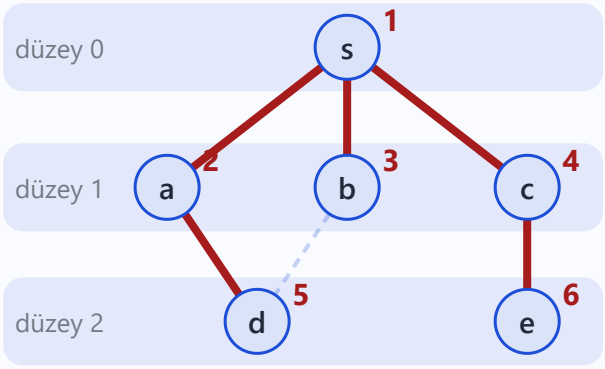
Ağaçta gezinme ile çizgede gezinme arasında önemli bir fark vardır. Ağaçta bir düğüme kökten tek bir yol vardır; çocuklara inerek ilerleyen bir algoritma hiçbir düğüme iki kez gelmez. Çizgede ise döngü olabilir ve aynı düğüme birden çok yoldan ulaşılabilir. Bu yüzden çizge gezinmesi hangi düğümleri

gördüğünü kaydetmek zorundadır. Bu kayıt çoğunlukla düğüm numarasıyla indislenen bir `boolean[] ziyaret` dizisidir.

> **Sık yapılan hata**
>
> Ağaç gezinmesini çizgeye olduğu gibi uygulamak. Ziyaret işareti olmayan bir gezinme döngüye girdiğinde aynı düğümleri tekrar tekrar gezer ve hiç bitmez. Döngü olmasa bile bir düğüme birden çok yoldan gelindiğinde düğüm birden çok kez işlenir ve maliyet üstel olarak büyüyebilir. Ziyaret işareti bir ayrıntı değil, algoritmanın doğruluğunun parçasıdır.

14.3.2 Genişlik öncelikli arama

Suya atılan bir taşı düşünelim. Dalgalar taşın düştüğü noktadan halkalar hâlinde yayılır; önce yakın halkalar, sonra uzaktakiler oluşur. BFS de böyle çalışır. Başlangıca 1 kenar uzaklıktaki düğümleri bulur, sonra 2 kenar uzaklıktakileri, sonra 3 kenar uzaklıktakileri. Başlangıçtan *k* kenar uzaktaki düğümlerin oluşturduğu gruba **düzey *k*** denir.



Şekil 14.18: BFS düzey düzey ilerler. Düğümlerin yanındaki numara ziyaret sırasıdır: s (1); a, b, c (2, 3, 4); d, e (5, 6). Kalın kenarlar BFS ağacını oluşturur; soluk kesikli b–d kenarı ağaca girmez, çünkü d'ye a üzerinden zaten ulaşılmıştır.

Bu sırayı kuyruk sağlar. Bir düğüm keşfedildiğinde kuyruğun arkasına eklenir; işlenecek düğüm her zaman kuyruğun önünden alınır. Düzey 1'deki düğümlerin hepsi, düzey 2'deki herhangi bir düğümden önce kuyruğa girer. Kuyruk ilk giren ilk çıkar ilkesiyle çalıştığı için düzey 1'in hepsi düzey 2'den önce işlenir.

BFS'nin her düğümü keşfettiği kenarlar bir ağaç oluşturur; buna **BFS ağacı** denir. Şekil 14.18'de d'ye hem a hem b üzerinden gidilebilir. a daha önce işlendiği için d, a–d kenarıyla keşfedilir; b–d kenarı ağaca girmez. BFS ağacında kökten bir düğüme giden yol, o düğüme giden en az kenarlı yollardan biridir. Bu özellik, ağırlıksız en kısa yol algoritmasının temelidir (14.4.1).

BFS'nin iki güçlü yanı vardır: ağırlıksız çizgede en kısa yolu bulur ve başlangıca yakın düğümleri önce keşfeder. Sosyal ağda "arkadaşlarımın arkadaşları" sorusu düzey 2'deki düğümlerdir; bir web tarayıcı robotu (*crawler*) bir siteyi bağlantı derinliğine göre katman katman tarar. Zayıf yanı bellektir: bir düzeyin tamamı aynı anda kuyrukta bekleyebilir. Her düğümün çok sayıda komşusu olan geniş çizgelerde kuyruk çok büyüyebilir.

14.3.3 BFS algoritması ve kodu

Önce algoritmayı sözde kodla, sonra Java ile verelim.

```
BFS(başlangıç):
  kuyruk.ekle(başlangıç)
  başlangıç.ziyaretEdildi()
  while kuyruk boş değil:
    w = kuyruk.çıkar()
    for w'nin ziyaret edilmemiş
        her komşusu u:
      u.ziyaretEdildi()
      kuyruk.ekle(u)
```

```
void bfs(int s) {
  boolean[] ziyaret = new boolean[n];
  Queue<Integer> q = new ArrayDeque<>();
  ziyaret[s] = true;  q.add(s);
  while (!q.isEmpty()) {
    int w = q.poll();
    System.out.print(w + " ");
    for (int u : komsu.get(w))
      if (!ziyaret[u]) {
        ziyaret[u] = true;
        q.add(u);
      }
  }
}
```

Java kodu, 14.2.7'deki `Cizge` sınıfının bir metodu olarak yazılmıştır; `n` ve `komsu` alanlarını kullanır. Satır satır inceleyelim:

- `boolean[] ziyaret = new boolean[n];` her düğüm için bir işaret ayırır. Java bu diziyi `false` ile başlatır; başta hiçbir düğüm görülmemiştir.
- `Queue<Integer> q = new ArrayDeque<>();` boş bir kuyruk oluşturur. `ArrayDeque`'te `add` arkaya ekler, `poll` öndeki öğeyi çıkarıp döndürür; ikisi de *O(1)*'dir.
- `ziyaret[s] = true; q.add(s);` başlangıç düğümünü işaretleyip kuyruğa koyar.
- Döngü, kuyruk boşalana kadar sürer. Her turda öndeki düğüm `w` çıkarılır ve yazdırılır.

- İç döngü `w`'nin komşu listesini gezer. Ziyaret edilmemiş her komşu `u` önce işaretlenir, sonra kuyruğa eklenir.

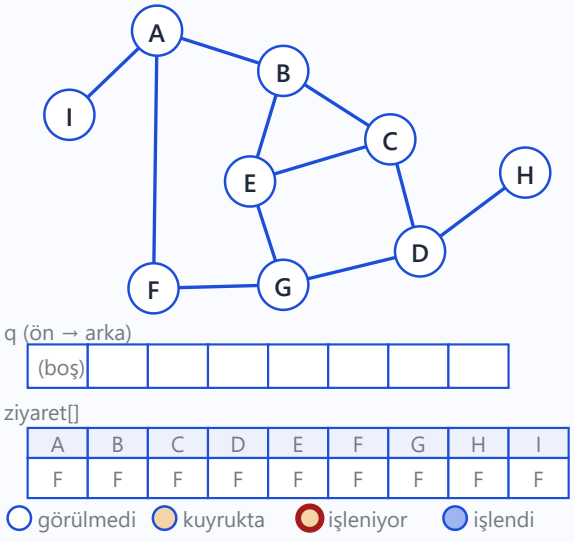
Kodun en ince noktası işaretin *nerede* konduğudur. Düğüm kuyruğa **girerken** işaretlenir, kuyruktan çıkarken değil. Böylece bir düğüm birden çok komşunun listesinde görünse bile kuyruğa yalnız bir kez girer. İşaret kuyruktan çıkarken konsaydı, aynı düğüm kuyrukta beklerken başka bir komşusu onu bir daha ekleyebilirdi.

> **Sık yapılan hata**
>
> Düğümü kuyruktan çıkarırken işaretlemek. Kenarları 0–1, 0–2, 1–2 olan üçgende böyle bir BFS 0'dan başlarsa çıktı `0 1 2 2` olur. 0 çıkınca 1 ve 2 eklenir. 1 çıkınca 2 henüz işaretsiz olduğu için tekrar eklenir. Sonuçta 2 iki kez çıkar ve iki kez yazdırılır. Yoğun çizgelerde kuyruk *O(n)* yerine *O(m)* boyuta büyür.

Maliyet. Her düğüm kuyruğa en çok bir kez girer ve bir kez çıkar: toplam *O(n)*. Her düğüm çıktığında komşu listesi bir kez taranır. Bütün listelerin toplam uzunluğu yönsüz çizgede *2m*'dir; tarama toplamı *O(m)* eder. BFS'nin toplam süresi komşuluk listesiyle *O(n + m)*'dir. Komşuluk matrisiyle her düğümün satırı *n* hücre taranacağı için süre $O(n^2)$ olur. Ek bellek *ziyaret* dizisi ve kuyruk için *O(n)*'dir.

14.3.4 Kod izleme: bfs(A)

Dokuz düğümlü bir çizge üzerinde `bfs(A)`'yı izleyelim. Düğümler A = 0, B = 1, …, I = 8 diye numaralanmıştır. Komşu listeleri alfabetik sıradadır: A: B, F, I · B: A, C, E · C: B, D, E · D: C, G, H · E: B, C, G · F: A, G · G: D, E, F · H: D · I: A. Çizgede 9 düğüm ve 11 kenar vardır.



Şekil 14.19: Gezinme örneklerinde kullanılan çizge ve başlangıç durumu. Kuyruk boştur, hiçbir düğüm ziyaret edilmemiştir. Aşağıdaki şeritte her düğümün `ziyaret[]` durumu görülür.

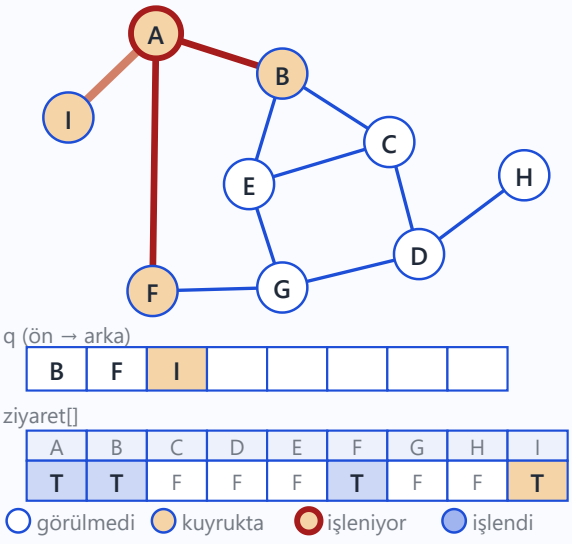
Tur	Çıkan `w` (yazdırılır)	Taranan komşular	İşaretlenip eklenenler	Kuyruk (tur sonunda, önden arkaya)
başlangıç	—	—	A	[A]
1	A	B, F, I	B, F, I (düzey 1)	[B, F, I]
2	B	A, C, E	C, E (düzey 2)	[F, I, C, E]
3	F	A, G	G (düzey 2)	[I, C, E, G]
4	I	A	—	[C, E, G]
5	C	B, D, E	D (düzey 3)	[E, G, D]
6	E	B, C, G	—	[G, D]
7	G	D, E, F	—	[D]
8	D	C, G, H	H (düzey 4)	[H]
9	H	D	—	[]

Tablo 14.10: `bfs(A)` izlemesi. Her satır dış döngünün bir turudur.

İzlemenin önemli anları şunlardır. Başlangıçta A işaretlenir ve kuyruğa girer. Birinci turda A çıkar ve yazdırılır; komşuları B, F, I ziyaret edilmemiştir, üçü de işaretlenip sırayla kuyruğa eklenir. Bunlar düzey 1'dir.

İkinci turda kuyruğun önündeki B çıkar. B'nin ilk komşusu A'dır ve zaten işaretlidir; atlanır. Ziyaret işareti olmasaydı A burada kuyruğa ikinci kez girerdi. C ve E işaretlenip kuyruğun arkasına eklenir. Kuyruk artık `[F, I, C, E]`'dir: önde hâlâ düzey 1'in düğümleri, arkada düzey 2'nin düğümleri vardır. Kuyruk, düzeylerin karışmasını kendiliğinden önler.

Üçüncü turda F çıkar, G eklenir. Dördüncü turda I çıkar; tek komşusu A işaretli olduğu için kuyruğa bir şey eklenmez. Böylece düzey 1 biter. Beşinci turda C çıkar ve D'yi düzey 3 olarak ekler. E ile G çıkınca bütün komşuları işaretli bulunur. Sekizinci turda D, H'yi düzey 4 olarak ekler. H çıkınca kuyruk boşalır ve döngü biter.



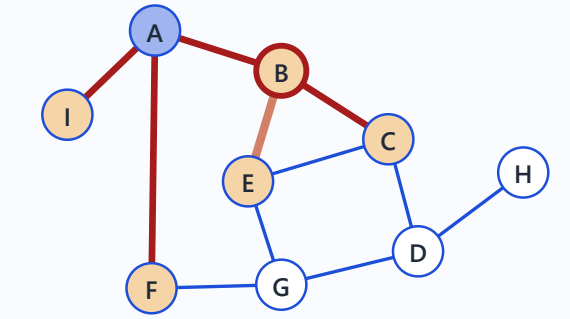
q (ön → arka)

B	F	I					

ziyaret[]

A	B	C	D	E	F	G	H	I
T	T	F	F	F	T	F	F	T

görülmedi kuyrukta işleniyor işlendi



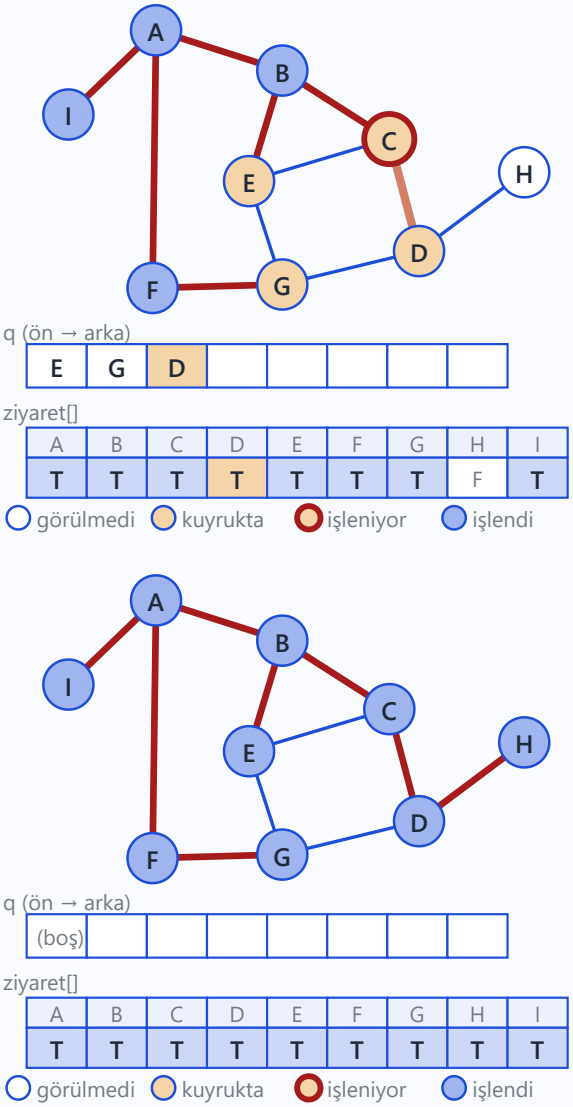
q (ön → arka)

F	I	C	E				

ziyaret[]

A	B	C	D	E	F	G	H	I
T	T	T	F	T	T	F	F	T

görülmedi kuyrukta işleniyor işlendi

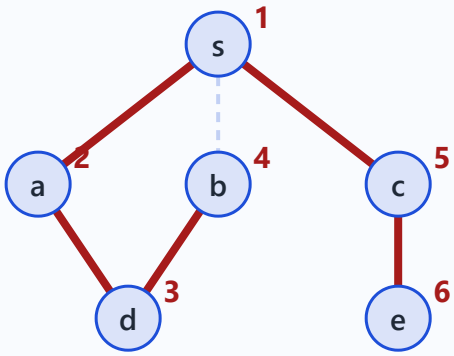


Şekil 14.20: `bfs(A)` 'nın dört anı. (a) A işlendi; kuyruk B, F, I. (b) B işlendi; kuyruk F, I, C, E. (c) C işlendi; kuyruk E, G, D. (d) Son durum: bütün düğümler işlendi, kalın kenarlar BFS ağacını gösterir.

Yazdırılan sıra `A B F I C E G D H`'dir. Düzeyler bu sırada ardışık bloklar hâlinde görülür: {A}, {B, F, I}, {C, E, G}, {D}, {H}. BFS ağacının kenarları A–B, A–F, A–I, B–C, B–E, F–G, C–D ve D–H'dir. Dokuz düğümlü bir ağaçta beklendiği gibi sekiz kenar vardır. Her düğüm kuyruğa bir kez girmiş, her komşu listesi bir kez taranmıştır; taranan toplam kayıt sayısı $2m = 22$'dir.

14.3.5 Derinlik öncelikli arama

Bir labirentte çıkış arayan biri genellikle şöyle davranır: bir koridora girer, sonuna kadar yürür. Çıkmaza girerse son yol ayrımına döner ve henüz denemediği bir koridora girer. Bu davranış DFS'nin ta kendisidir. DFS gidebildiği kadar derine iner. Ziyaret edilmemiş komşusu kalmayan bir düğüme gelince **geri döner** (*backtracking*) ve başka bir dalı dener.



Şekil 14.21: DFS bir yol boyunca derine iner. Numaralar ziyaret sırasıdır: s (1), a (2), d (3), b (4); b'de çıkmaz vardır, geri dönülür; c (5), e (6). Kalın kenarlar DFS ağacıdır. s–b kenarı ağaca girmez, çünkü b'ye d üzerinden ulaşılmıştır.

Şekil 14.21'i Şekil 14.18 ile karşılaştıralım. Çizge aynıdır, ziyaret sıraları farklıdır. BFS s'den sonra a, b, c'yi, yani bütün komşuları ziyaret eder. DFS ise s'den a'ya, a'dan d'ye, d'den b'ye iner. b'nin bütün komşuları görülmüştür; DFS geri dönmek zorunda kalır. s'ye kadar geri çıkıp c'ye, oradan e'ye iner.

DFS'nin artıları şunlardır: bellekte yalnız o anki yol tutulur; geniş ama sığ çizgelerde BFS'den az bellek kullanır. Bir yolun sonuna kadar gitmek gereken problemlerde (labirent, oyun ağacı) doğal bir yöntemdir. Eksisi de açıktır: bulduğu yol en kısa yol olmak zorunda değildir. Şekil 14.21'de DFS b'ye s → a → d → b yoluyla, üç kenarla ulaşmıştır; oysa s–b kenarı doğrudan vardır. DFS'nin kullanıldığı başlıca yerler labirent çözme, döngü tespiti, bağlantılı bileşenleri bulma ve yönlü döngüsüz çizgelerde topolojik sıralamadır.

14.3.6 DFS kodu: özyinelemeli ve açık yığınla

DFS'nin en kısa yazılışı özyinelemelidir.

```
void dfs(int v, boolean[] ziyaret) {
  ziyaret[v] = true;
  System.out.print(v + " ");
  for (int u : komsu.get(v))
    if (!ziyaret[u])
      dfs(u, ziyaret);
}
```

Metot önce `v`'yi işaretler ve yazdırır. Sonra `v`'nin komşularını sırayla dener; ziyaret edilmemiş ilk komşu için kendini çağırır. Bu çağrı dönmeden döngü devam etmez. Yani DFS, ilk komşunun altındaki her şeyi bitirmeden ikinci komşuya geçmez; "derine önce" davranışı buradan gelir. Komşuların hepsi denendiğinde metot döner ve çağıran düğüm kaldığı yerden devam eder. Bu dönüş, geri dönmenin kodda görünen biçimidir.

Özyineleme yığını bizim yerimize **çağrı yığını** ile tutar. Her etkin `dfs` çağrısı çağrı yığınında bir çerçevedir; çerçevelerin sırası, başlangıçtan o anki düğüme giden yoldur. Aynı algoritma açık bir yığınla da yazılabilir:

```
void dfsYigin(int s) {
  boolean[] ziyaret = new boolean[n];
  Deque<Integer> yigin = new ArrayDeque<>();
  yigin.push(s);  ziyaret[s] = true;
  while (!yigin.isEmpty()) {
    int v = yigin.peek();
    Integer yeni = null;
    for (int u : komsu.get(v))
      if (!ziyaret[u]) { yeni = u; break; }
    if (yeni == null) yigin.pop(); // geri
    else { ziyaret[yeni] = true;
           yigin.push(yeni); }
  }
}
```

Açık yığınlı sürüm özyinelemeyi adım adım taklit eder. Yığının üstü (`peek`) o anda bulunduğumuz düğümdür. Her turda üstteki düğümün ziyaret edilmemiş *ilk* komşusu aranır. Bulunursa işaretlenir ve yığına itilir; bu, özyinelemeli bir çağrıya karşılık gelir. Bulunamazsa (`yeni == null`) üstteki düğüm yığından çıkarılır; bu, bir çağrının dönmesine karşılık gelir. `break` önemlidir: ilk uygun komşu bulunca arama durur ve hemen o komşuya inilir.

Bu sürüm, derin çizgelerde özyinelemenin `StackOverflowError` vermesini önler. Java'nın çağrı yığını sınırlıdır; yüz binlerce düğümlü bir yol çizgesinde özyinelemeli DFS yığını taşırabilir. Açık yığın ise yığın belleğinde (*heap*) tutulur ve çok daha büyüyebilir.

Maliyet. Özyinelemeli sürümde her düğüm için `dfs` bir kez çağrılır ve o düğümün listesi bir kez gezilir: $O(n + m)$. Açık yığınlı sürümde ise her `peek` turunda komşu listesi *baştan* taranır. Bir düğüm yığının üstüne birkaç kez geri geldiğinde listesinin başındaki işaretli komşular yeniden atlanır. Her düğüm yığının üstüne en çok derecesi kadar artı bir kez gelir; bu yüzden en kötü durumda tarama toplamı $\Sigma\, deg(v)^2$ mertebesine çıkabilir. Her liste için "nerede kaldım" bilgisi bir indis dizisinde tutulursa açık yığınlı sürüm de $O(n + m)$ olur. Sunumdaki kod sadeliği öne çıkarır; küçük derecelerde bu fark önemsizdir.

14.3.7 Kod izleme: özyinelemeli dfs(A)

Şekil 14.19'daki çizgede `dfs(A, ziyaret)` çağrısını izleyelim. Komşular yine alfabetik sıradadır.

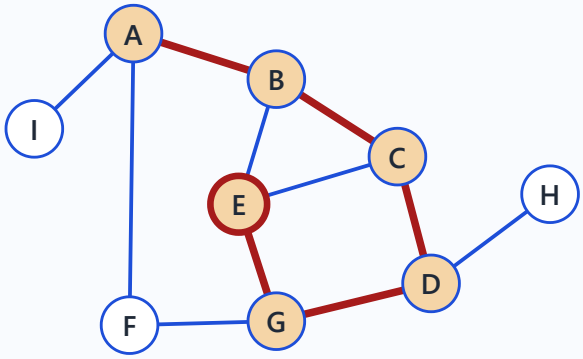
Adım	Olay	Çağrı yığını	Yazdırılan (toplam)
1	`dfs(A)` : A işaretlenir	A	A
2	A'nın komşusu B yeni → `dfs(B)`	A B	A B
3	B: A işaretli, atla; C yeni → `dfs(C)`	A B C	A B C

Adım	Olay	Çağrı yığını	Yazdırılan (toplam)
4	C: B atla; D yeni → `dfs(D)`	A B C D	A B C D
5	D: C atla; G yeni → `dfs(G)`	A B C D G	A B C D G
6	G: D atla; E yeni → `dfs(E)`	A B C D G E	A B C D G E
7	E: B, C, G işaretli → `dfs(E)` döner	A B C D G	—
8	G: F yeni → `dfs(F)`	A B C D G F	A B C D G E F
9	F: A, G işaretli → döner; G'nin komşusu kalmadı → döner	A B C D	—
10	D: H yeni → `dfs(H)`	A B C D H	A B C D G E F H
11	H: D atla → döner; D biter → döner	A B C	—
12	C: E işaretli → döner; B: E işaretli → döner	A	—
13	A: F işaretli, atla; I yeni → `dfs(I)`	A I	A B C D G E F H I
14	I: A atla → döner; A biter → döner	(boş)	—

Tablo 14.11: Özyinelemeli `dfs(A)` izlemesi. Çağrı yığını alttan üste yazılmıştır.

Tablo, DFS'nin iki evresini açıkça gösterir. 1.–6. adımlarda algoritma hep ileri gider ve çağrı yığını her adımda bir çerçeve büyür: A → B → C → D → G → E. E'de bütün komşular işaretlidir; ilk geri dönüş burada olur. Geri dönen algoritma G'ye gelir ve G'nin listesinde kaldığı yerden devam eder: D ve E denenmiştir, sırada F vardır.

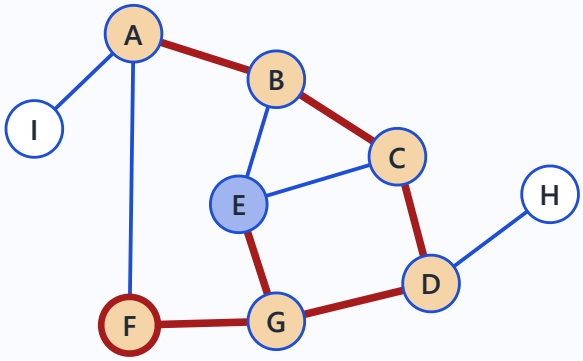
Dikkat edilecek bir nokta da şudur: bir düğüm döndükten sonra çağıran düğüm döngüsünü *baştan* değil, kaldığı yerden sürdürür. Örneğin 12. adımda C'ye dönülür; C'nin listesi B, D, E'dir ve sırada E kalmıştır. E ise G'nin altında zaten işaretlenmiştir. Her düğümün listesi tam bir kez gezilir.



ziyaret[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	F	T	F	F

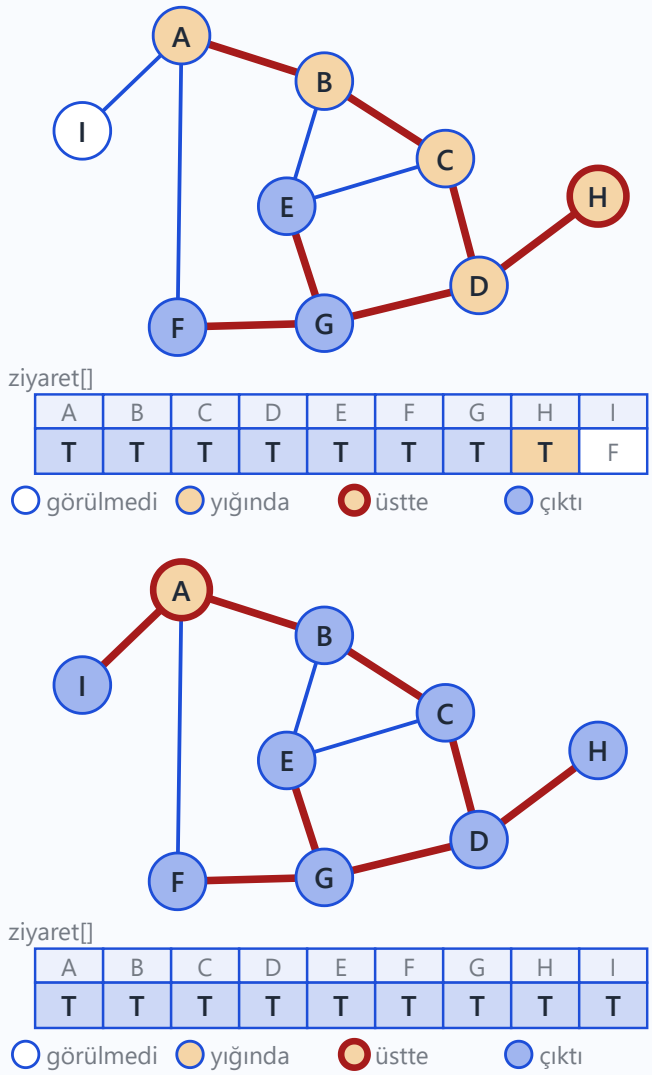
görülmedi yığında üstte çıktı



ziyaret[]

A	B	C	D	E	F	G	H	I
T	T	T	T	T	T	T	F	F

görülmedi yığında üstte çıktı



Şekil 14.22: Özyinelemeli `dfs(A)` 'nın dört anı ve sağdaki çağrı yığını. (a) En derin nokta: yığın A B C D G E. (b) E döndü, G'nin altında F'ye inildi. (c) G bitti, D'nin altında H'ye inildi. (d) Son adım: `dfs(I)` döndü, `dfs(A)` da bitmek üzere; bütün düğümler ziyaret edildi.

Ziyaret sırası `A B C D G E F H I` 'dır. DFS ağacının kenarları A–B, B–C, C–D, D–G, G–E, G–F, D–H ve A–I'dır. Çağrı yığınının en derin hâli altı çerçevedir. BFS ağacıyla karşılaştırıldığında DFS ağacı daha uzun ve daha ince bir yapıdadır: kökten E'ye BFS ağacında iki kenar, DFS ağacında beş kenar vardır.

14.3.8 Kod izleme: dfsYigin(A)

Aynı çizgede açık yığınlı sürümü izleyelim. Her turda yığının üstüne bakılır, ziyaret edilmemiş ilk komşu aranır ve ya push ya da pop yapılır.

Tur	peek	İlk ziyaret edilmemiş komşu	İşlem	Yığın (tur sonunda)
başlangıç	—	—	push A	A
1	A	B	push B	A B
2	B	C	push C	A B C

Tur	peek	İlk ziyaret edilmemiş komşu	İşlem	Yığın (tur sonunda)
3	C	D	push D	A B C D
4	D	G	push G	A B C D G
5	G	E	push E	A B C D G E
6	E	yok	pop	A B C D G
7	G	F	push F	A B C D G F
8	F	yok	pop	A B C D G
9	G	yok	pop	A B C D
10	D	H	push H	A B C D H
11	H	yok	pop	A B C D
12	D	yok	pop	A B C
13	C	yok	pop	A B
14	B	yok	pop	A
15	A	I	push I	A I
16	I	yok	pop	A
17	A	yok	pop	(boş)

Tablo 14.12: `dfsYigin(A)` izlemesi. Yığın alttan üste yazılmıştır.

Döngü 17 tur döner: 8 push (A dışındaki sekiz düğüm) ve 9 pop (bütün düğümler). Push turları yeni bir düğümün ziyaret edildiği anlardır ve sıra `A B C D G E F H I`'dır; özyinelemeli sürümle tıpatıp aynıdır. Yığının her andaki içeriği de Tablo 14.11'deki çağrı yığınıyla aynıdır. İki kod aynı algoritmadır; yalnız yığını tutan taraf değişmiştir.

Sık yapılan hata

Açık yığınlı DFS'yi BFS kodundaki kuyruğu yığınla değiştirerek yazmak: bir düğümü çıkarıp bütün komşularını birden yığına itmek. Bu sürüm de her düğümü bir kez ziyaret eder, ama ziyaret sırası özyinelemeli DFS'nin sırası olmaz. Komşular ters sırada çıkar ve bir düğüm, derine inilmeden önce kardeşleriyle birlikte işaretlenmiş olur. Sunumdaki `peek` ve "ilk komşu, `break`" düzeni bu yüzden seçilmiştir.

14.3.9 BFS ile DFS'nin karşılaştırması

	BFS	DFS
Yardımcı yapı	kuyruk (FIFO)	yığın (LIFO) ya da özyineleme
Zaman (komşuluk listesi)	O(n + m)	O(n + m)
Ek bellek	O(n): en geniş düzey kuyrukta	O(n): en uzun yol yığında
En kısa yolu verir mi?	evet (ağırlıksız çizgede)	hayır

	BFS	DFS
Tipik kullanım	en az adım, düzey bulma	döngü tespiti, bileşenler, labirent, topolojik sıralama

Tablo 14.13: BFS ve DFS'nin karşılaştırması.

İkisi de *O(n + m)* zaman alır, çünkü her düğüm bir kez işlenir ve bütün komşuluk kayıtları bir kez taranır. Yönsüz çizgede her kenar iki listede göründüğü için taranan kayıt sayısı *2m*'dir; sabit çarpan büyük-O gösteriminde düşer ve sonuç yine *O(m)*'dir.

Ek bellek ikisinde de en kötü durumda *O(n)*'dir, ama kötü durumlar farklıdır. BFS'yi zorlayan çizge geniştir: bir düğümün çok sayıda komşusu varsa hepsi aynı anda kuyruğa girer. DFS'yi zorlayan çizge derindir: uzun bir zincirde yığın zincirin boyu kadar büyür.

İzlemelerdeki sıralar, komşuların alfabetik sırayla gezildiği varsayımıyla elde edilmiştir. Komşu listelerinin sırası değişirse ziyaret sırası da değişir; ama ziyaret edilen düğüm kümesi ve maliyet değişmez. Örneğin aynı çizgede H'den başlanırsa BFS `H D C G B E F A I`, DFS `H D C B A F G E I` sırasını verir. Bir başka gözlem de şudur: açık yığınlı DFS'deki yığın bir kuyrukla değiştirilirse ve komşuların hepsi birden eklenirse algoritma BFS'ye dönüşür. İki arama arasındaki fark, yardımcı yapının hangi ucundan okunduğudur.

14.3.10 Bağlantılı bileşenler

Bağlantısız bir çizgede bir düğümden başlayan gezinme yalnız o düğümün bileşenini ziyaret eder. Bütün bileşenleri bulmak için gezinmeyi, henüz hiçbir bileşene atanmamış her düğümden yeniden başlatmak yeterlidir. Her yeni başlangıç yeni bir bileşen demektir.

```
int bilesenler(int[] bil) {
  Arrays.fill(bil, -1);
  int say = 0;
  for (int s = 0; s < n; s++)
    if (bil[s] == -1) dfs(s, say++, bil);
  return say;
}
void dfs(int v, int c, int[] bil) {
  bil[v] = c;
  for (int u : komsu.get(v))
    if (bil[u] == -1) dfs(u, c, bil);
}
```

`bil` dizisi her düğümün bileşen numarasını tutar. Başta bütün değerler −1'dir: “henüz atanmadı”. Bu dizi aynı zamanda ziyaret işareti görevi görür; ayrı bir `boolean` diziye gerek kalmaz. Dış döngü düğümleri 0'dan *n − 1*'e tarar. −1 değerli bir düğüm bulunca `dfs(s, say++, bil)` çağrılır. `say++` sonradan artırma olduğu için çağrıya `say`'ın eski değeri gider; ilk bileşen 0 numarasını alır. `dfs`, ulaştığı her düğüme bileşen numarasını yazar.



Şekil 14.23: Altı düğümlü çizge (kenarlar 0–1, 0–2, 3–4) ve `bilesenler`'in sonucu. Üç bileşen üç renkle gösterilmiştir: `bil = [0, 0, 0, 1, 1, 2]`.

`s`	`bil[s]`	**Yapılan**	`say`	`bil` **(sonra)**
—	—	`Arrays.fill`	0	[−1, −1, −1, −1, −1, −1]
0	−1	`dfs(0, 0)`: 0, 1, 2 boyanır	1	[0, 0, 0, −1, −1, −1]
1	0	atla	1	aynı
2	0	atla	1	aynı
3	−1	`dfs(3, 1)`: 3, 4 boyanır	2	[0, 0, 0, 1, 1, −1]
4	1	atla	2	aynı
5	−1	`dfs(5, 2)`: yalnız 5	3	[0, 0, 0, 1, 1, 2]

Tablo 14.14: Şekil 14.23'teki çizgede `bilesenler` izlemesi.

`dfs(0, 0)`'ın içi şöyle yürür: 0 boyanır; 0'ın ilk komşusu 1 boyanır ve 1'in tek komşusu 0 zaten boyalı olduğu için 1'den dönülür. Sonra 0'ın ikinci komşusu 2 boyanır; onun da tek komşusu 0'dır. Metot sonunda 3 döndürür: çizgede üç bileşen vardır, *{0, 1, 2}*, *{3, 4}* ve *{5}*.

Maliyet *O(n + m)*'dir. Dış döngü *n* tur döner. Her düğüm tam bir kez boyanır ve listesi bir kez gezilir; her kenara iki ucundan birer kez bakılır. Bir kez hesaplanan `bil` dizisiyle "*u* ile *v* aynı bileşende mi?" sorusu `bil[u] == bil[v]` karşılaştırmasıyla *O(1)*'de yanıtlanır. Aynı soru her seferinde yeni bir gezinmeyle yanıtlansaydı her sorgu *O(n + m)* sürerdi.

14.3.11 İki parçalı çizge testi

Tanım 14.10

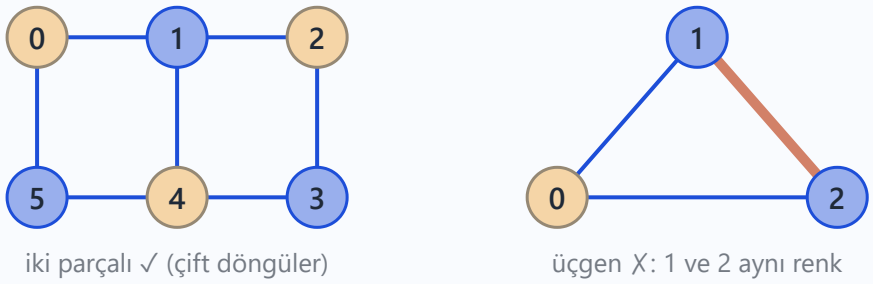
Düğümleri iki gruba ayrılabilen ve her kenarı iki grup *arasında* bulunan çizgeye **iki parçalı çizge** (*bipartite graph*) denir. Böyle bir çizgede hiçbir kenar aynı grubun iki düğümünü bağlamaz.

İki parçalı çizgeler eşleştirme problemlerinde doğal olarak ortaya çıkar. Öğrenciler bir grup, projeler öbür grup olsun; kenar "bu öğrenci bu projeyi istiyor" anlamına gelsin. Hiçbir kenar iki öğrenciyi ya da iki projeyi bağlamaz. İki vardiyalı çizelgeleme ve çakışmasız gruplama da aynı yapıdadır.

Grupları iki renk olarak düşünelim. Çizge iki parçalıysa her düğüme, komşularının hepsinin rengi kendisininkinden farklı olacak biçimde iki renkten biri verilebilir. Bunu BFS ile sınamak kolaydır: başlangıca 0 rengi verilir; her yeni keşfedilen düğüm, onu keşfeden düğümün zıt rengini alır. Daha önce boyanmış bir komşunun rengi şimdiki düğümün rengiyle aynıysa çizge iki parçalı değildir.

```
// renk[] = -1 ile başlar; s'den BFS
renk[s] = 0;  q.add(s);
while (!q.isEmpty()) {
  int w = q.poll();
  for (int u : komsu.get(w))
    if (renk[u] == -1) {
      renk[u] = 1 - renk[w];  q.add(u);
    } else if (renk[u] == renk[w]) return false;
}
return true;
```

`renk` dizisi −1 ile başlar; −1 "boyanmadı", 0 ve 1 iki renktir. `1 - renk[w]` ifadesi zıt rengi verir: 0'ın zıttı 1, 1'in zıttı 0'dır. Kod BFS'nin aynısıdır; tek fark, boyalı bir komşuda rengin denetlenmesidir.



Şekil 14.24: Solda iki renkle boyanabilen altı düğümlü çizge: 0–1–2–3–4–5–0 döngüsü ve 1–4 kenarı. Sağda üçgen: 0'dan BFS ile boyanınca 1 ve 2 aynı rengi alır, aralarındaki kenar çakışmayı gösterir.

Örnek 14.7

Soldaki çizge. 0 rengi 0 alır. Komşuları 1 ve 5 rengi 1 alır. 1 işlenince 2 ve 4 rengi 0 alır. 5 işlenince komşuları 4 (renk 0) ve 0 (renk 0) zıt renklidir; sorun yok. 2 işlenince 3 rengi 1 alır. Kalan denetimlerin hiçbiri çakışma vermez. Sonuç `renk = [0, 1, 0, 1, 0, 1]` : çift numaralılar bir grupta, tek numaralılar öbür gruptadır. Çizgedeki döngülerin hepsi çift uzunluktadır: 0–1–2–3–4–5–0 altı, 0–1–4–5–0 ve 1–2–3–4–1 dörder kenarlıdır.

Üçgen. 0 rengi 0 alır, komşuları 1 ve 2 rengi 1 alır. 1 işlenirken komşusu 2 zaten boyalıdır ve rengi 1'in rengiyle aynıdır. Metot `false` döndürür. Üçgen iki parçalı değildir: üç düğümü iki renge, komşular farklı olacak biçimde dağıtmak imkânsızdır.

Bu iki örnek genel bir kuralı yansıtır: **bir çizgenin iki parçalı olması, tek uzunlukta döngü içermemesiyle eşdeğerdir.** Bir döngü boyunca renkler 0, 1, 0, 1 diye değişir. Döngünün başladığı düğüme geri dönüldüğünde renk başlangıçtakiyle aynı olmalıdır; bu ancak döngünün kenar sayısı çiftse mümkündür. Örneğin 5 düğümlü döngü C_5 iki parçalı değildir, 6 düğümlü döngü C_6 ise iki parçalıdır. Testin maliyeti BFS'ninkiyle aynıdır: $O(n + m)$. Çizge bağlantısızsa test, bileşenlerdeki gibi her boyanmamış düğümden yeniden başlatılır.

14.3.12 Topolojik sıralama ve Kahn algoritması

Bir öğrencinin alacağı dersleri düşünelim. Bazı dersler başka derslerin ön koşuludur. Öğrenci dersleri öyle bir sırayla almalıdır ki her ders, ön koşullarının hepsinden sonra gelsin. Bu ilişki yönlü bir çizgedir: "u, v'nin ön koşulu" ise $u \to v$ oku vardır. Aranan sıraya topolojik sıra denir.

> **Tanım 14.11**
>
> Yönlü bir çizgenin **topolojik sıralaması** (*topological sort*), düğümlerin her $u \to v$ okunda u önce gelecek biçimde dizilmesidir. Yönlü döngü içermeyen yönlü çizgeye **yönlü döngüsüz çizge** (*directed acyclic graph*, DAG) denir. Topolojik sıra yalnız DAG'lerde vardır.

Döngülü bir çizgede neden topolojik sıra olamayacağı kolayca görülür. $a \to b \to c \to a$ döngüsü olsun. Sırada a, b'den; b, c'den; c de a'dan önce gelmelidir. Bu, a'nın kendisinden önce gelmesi demektir. Ders örneğinde her ders, döngüdeki bir başka dersin önce alınmasını bekler ve hiçbiri başlayamaz.

Topolojik sıra tek olmak zorunda değildir. Aynı anda hazır olan, yani ön koşulu kalmamış birden çok ders varsa bunlar herhangi bir sırayla alınabilir. Örneğin $0 \to 1$, $0 \to 2$, $1 \to 3$, $2 \to 3$ DAG'ında hem *0 1 2 3* hem de *0 2 1 3* geçerli sıralardır. Topolojik sıralama ders planlarında, görev bağımlılıklarında, derleme sırasının belirlenmesinde ve iş akışı planlamada kullanılır.

Kahn algoritmasının fikri basittir. Giriş derecesi 0 olan bir düğümün hiçbir ön koşulu yoktur; sırada ilk o gelebilir. Bu düğüm sıraya yazılınca ondan çıkan oklar "silinmiş" sayılır. Silinen her ok, hedefinin giriş derecesini bir azaltır. Giriş derecesi böylece 0'a düşen düğümler de hazır hâle gelir. Hazır düğümler bir kuyrukta bekletilir.

```
Kahn(G):
  q = girişDerecesi 0 olan düğümler
  sonuc = []
  while q boş değil:
    u = q'dan çıkar
    sonuc'a u ekle
    for u'dan çıkan her (u, v):
      girişDerecesi[v]--
      if girişDerecesi[v] == 0:
        q'ya v ekle
  if sonuc.size < n:
    döngü var
  return sonuc
```

```
List<Integer> kahn(List<List<Integer>> adj) {
  int n = adj.size();
  int[] giris = new int[n];
  for (List<Integer> l : adj)
    for (int v : l) giris[v]++;
  Queue<Integer> q = new ArrayDeque<>();
  for (int v = 0; v < n; v++)
    if (giris[v] == 0) q.add(v);
  List<Integer> sonuc = new ArrayList<>();
  while (!q.isEmpty()) {
    int u = q.poll();
    sonuc.add(u);
    for (int v : adj.get(u))
      if (--giris[v] == 0) q.add(v);
  }
  return sonuc;  // boyut < n → döngü
}
```

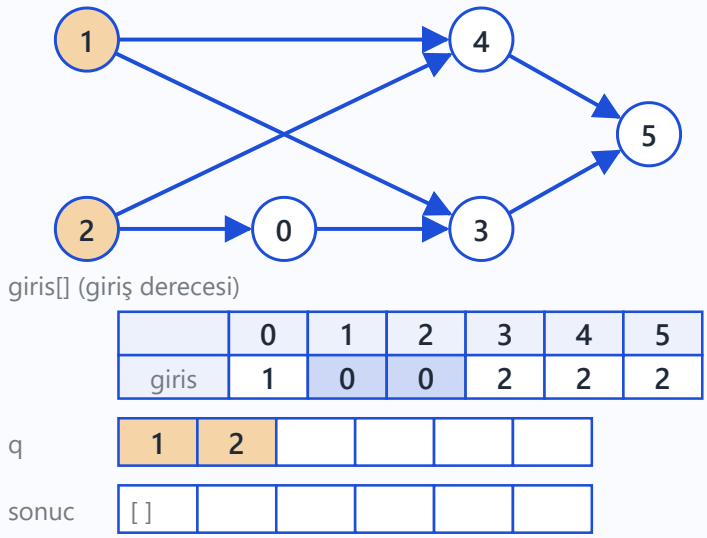
Java kodu üç evreden oluşur:

1. **Sayma.** İç içe iki döngü bütün listeleri gezer ve her okun hedefinin `giris` değerini bir artırır. Sonunda `giris[v]`, *v*'nin giriş derecesidir. Bu evre *O(n + m)* sürer.
2. **Başlatma.** Giriş derecesi 0 olan bütün düğümler kuyruğa konur.
3. **Ana döngü.** Kuyruktan bir düğüm *u* çıkar ve sonuca eklenir. *u*'nun her okunun hedefi için `--giris[v]` yapılır. Ön ek azaltma önce azaltır, sonra yeni değeri verir; değer 0'a düşmüşse *v* kuyruğa eklenir.

Kuyruk boşaldığında sonuç listesinde *n* düğüm varsa bulunan sıra bir topolojik sıradır. *n*'den az düğüm varsa çizgede yönlü bir döngü vardır. Döngüdeki düğümlerin giriş dereceleri hiçbir zaman 0'a inmez, çünkü her biri döngüdeki bir önceki düğümü bekler. Algoritma bu düğümleri hiç kuyruğa koymaz.

14.3.13 Kod izleme: kahn()

Altı dersli bir ön koşul çizgesi alalım. Oklar *0 → 3, 1 → 3, 1 → 4, 2 → 0, 2 → 4, 3 → 5, 4 → 5*'tir. Komşu listeleri 0: [3] · 1: [3, 4] · 2: [0, 4] · 3: [5] · 4: [5] · 5: [] biçimindedir.



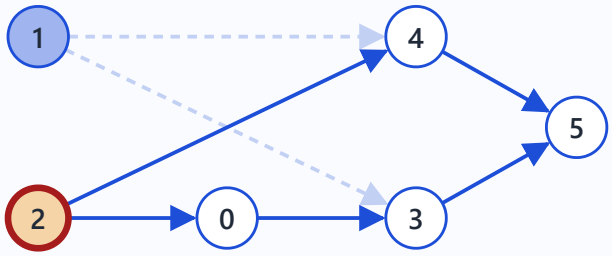
Şekil 14.25: Ders ön koşul çizgesi; sayma evresinden sonraki durum. Giriş dereceleri `giris = [1, 0, 0, 2, 2, 2]` 'dir; ön koşulu olmayan 1 ve 2 kuyruktadır.

Sayma evresi. 0'ın listesi `giris[3]` 'ü 1 yapar. 1'in listesi `giris[3]` 'ü 2'ye, `giris[4]` 'ü 1'e çıkarır. 2'nin listesi `giris[0]` 'ı 1'e, `giris[4]` 'ü 2'ye çıkarır. 3 ve 4'ün listeleri `giris[5]` 'i 2'ye çıkarır. 5'in listesi boştur ve bir şey değiştirmez. Sonuç `[1, 0, 0, 2, 2, 2]` 'dir. Giriş derecesi 0 olan 1 ve 2 kuyruğa girer.

Çıkan `u`	Silinen oklar ve yeni giriş dereceleri	Kuyruğa girenler	`giris` (sonra)	Kuyruk	`sonuc`
—	—	1, 2	[1, 0, 0, 2, 2, 2]	[1, 2]	[]
1	1 → 3: giris[3] = 1; 1 → 4: giris[4] = 1	—	[1, 0, 0, 1, 1, 2]	[2]	[1]
2	2 → 0: giris[0] = 0; 2 → 4: giris[4] = 0	0, 4	[0, 0, 0, 1, 0, 2]	[0, 4]	[1, 2]
0	0 → 3: giris[3] = 0	3	[0, 0, 0, 0, 0, 2]	[4, 3]	[1, 2, 0]
4	4 → 5: giris[5] = 1	—	[0, 0, 0, 0, 0, 1]	[3]	[1, 2, 0, 4]
3	3 → 5: giris[5] = 0	5	[0, 0, 0, 0, 0, 0]	[5]	[1, 2, 0, 4, 3]
5	(liste boş)	—	[0, 0, 0, 0, 0, 0]	[]	[1, 2, 0, 4, 3, 5]

Tablo 14.15: `kahn()` ana döngüsünün izlemesi.

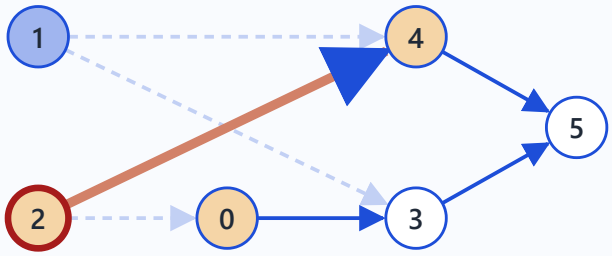
İzlemede şu anlara dikkat etmek gerekir. 1 çıkınca 3 ve 4'ün giriş dereceleri azalır ama 0'a inmez: ikisi de hâlâ başka bir ön koşul bekler. 3, 0'ı; 4 ise 2'yi bekler. 2 çıkınca hem 0 hem 4 hazır olur. 0 çıkınca 3'ün son ön koşulu da tamamlanır. 5'in iki ön koşulu vardır (3 ve 4); 4 çıkınca `giris[5]` 1'e iner, 3 çıkınca 0 olur.



giris[] (giriş derecesi)

	0	1	2	3	4	5
giris	1	0	0	1	1	2

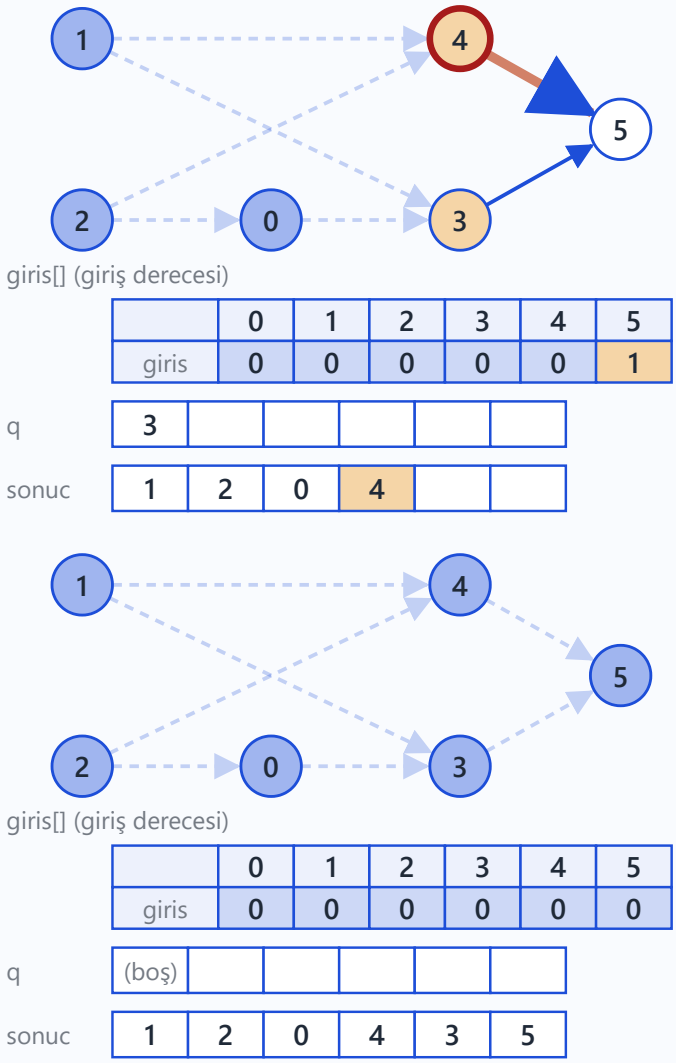
q	(boş)					
sonuc	1	2				



giris[] (giriş derecesi)

	0	1	2	3	4	5
giris	0	0	0	1	0	2

q	0	4				
sonuc	1	2				



Şekil 14.26: `kahn()` 'ın dört anı. (a) 2 kuyruktan çıktı ve sonuca eklendi; okları henüz işlenmedi, kuyruk boş. (b) 2 → 4 de silindi; kuyruk 0, 4. (c) 4 → 5 silindi; 5 hâlâ 3'ü bekler. (d) Son: sonuç 1 2 0 4 3 5.

Sonuç `1 2 0 4 3 5` 'tir ve boyutu $6 = n$'dir; çizgede döngü yoktur. Sıra gerçekten geçerlidir: her ok için kaynak hedeften önce gelir. Örneğin *2 → 0* okunda 2 ikinci, 0 üçüncü sıradadır. Her düğüm bir kez kuyruğa girmiş, her ok bir kez azaltılmıştır: maliyet $O(n + m)$'dir.

Örnek 14.8

Oklar *0 → 1, 1 → 2, 2 → 0, 2 → 3* olsun. Giriş dereceleri `[1, 1, 1, 1]` 'dir; hiçbir düğümün giriş derecesi 0 değildir. Kuyruk baştan boştur ve ana döngü hiç dönmez. Sonuç boş listedir; boyutu 0 < 4 olduğu için döngü vardır. 3 döngünün parçası değildir, ama döngüdeki 2'ye bağlı olduğu için o da sıraya giremez.

Topolojik sıralamanın bir başka yolu DFS'dir. DFS'de bir düğümün çağrısı, ondan ulaşılabilen her düğümün çağrısı bittikten sonra biter. Düğümler bitiş zamanlarına göre ters sırada dizilirse topolojik sıra elde edilir. Bu yöntemde döngü, bir sonraki kesimdeki geri kenar denetimiyle bulunur.

> **Sık yapılan hata**
>
> Kahn algoritmasının dönüş değerini denetlememek. Döngülü bir çizgede algoritma hata vermez; yalnız eksik bir liste döndürür. Sonuç boyutu *n* ile karşılaştırılmazsa eksik liste geçerli bir sıra sanılır ve döngüdeki dersler plandan sessizce düşer.

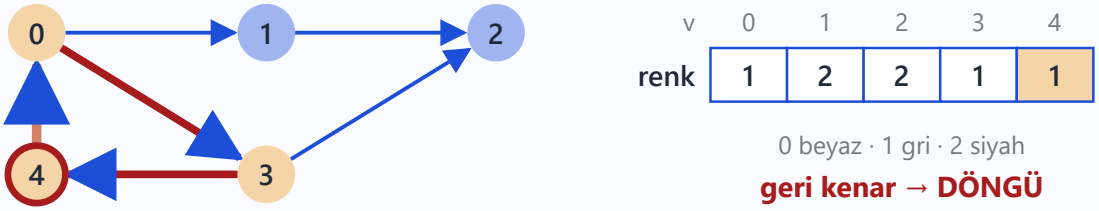
14.3.14 Yönlü çizgede döngü tespiti

Yönsüz çizgede döngü bulmak görece kolaydır: DFS sırasında, ebeveyn dışında işaretli bir komşuya rastlamak döngü demektir. Yönlü çizgede ise işaretli bir düğüme ok gitmesi döngü için yeterli değildir. İşaretli düğümün o anki DFS yolunun üzerinde olup olmadığı önemlidir. Bu ayrımı yapmak için her düğüm üç renkten biriyle işaretlenir:

- **beyaz (0):** henüz ziyaret edilmedi;
- **gri (1):** ziyaret edildi ve şu anki DFS yolunun üzerinde; çağrısı henüz bitmedi;
- **siyah (2):** ziyaret edildi ve çağrısı bitti; ondan ulaşılabilen her şey incelendi.

```
// renk: 0 beyaz, 1 gri (yolda), 2 siyah
boolean donguVar(int v, int[] renk) {
  renk[v] = 1;
  for (int u : komsu.get(v)) {
    if (renk[u] == 1) return true;  // geri kenar
    if (renk[u] == 0 && donguVar(u, renk))
      return true;
  }
  renk[v] = 2;
  return false;
}
```

Bir okun hedefi gri ise, hedef düğümden şu anki düğüme bir yol vardır (DFS yolu boyunca). Bu yola okun kendisi eklenince döngü kapanır. Böyle bir oka **geri kenar** denir. Hedef siyah ise ondan şu anki düğüme dönen bir yol yoktur; olsaydı o düğümün çağrısı sırasında şu anki düğüm de ziyaret edilmiş olurdu. Siyah düğüme giden ok döngü kurmaz.



Şekil 14.27: Oklar 0 → 1, 1 → 2, 0 → 3, 3 → 2, 3 → 4, 4 → 0. 4 → 0 oku incelenirken 0 gridir, yani şu anki yolun üzerindedir: döngü 0 → 3 → 4 → 0 bulunur. 1 ve 2 siyahtır.

Adım	Olay	`renk` (0…4)
1	0 gri olur	[1, 0, 0, 0, 0]
2	0 → 1: 1 beyaz, çağrı; 1 gri olur	[1, 1, 0, 0, 0]
3	1 → 2: 2 beyaz, çağrı; 2 gri olur	[1, 1, 1, 0, 0]
4	2'nin oku yok: 2 siyah, `false`	[1, 1, 2, 0, 0]
5	1'in okları bitti: 1 siyah, `false`	[1, 2, 2, 0, 0]
6	0 → 3: 3 beyaz, çağrı; 3 gri olur	[1, 2, 2, 1, 0]
7	3 → 2: 2 siyah, döngü değil, devam	[1, 2, 2, 1, 0]
8	3 → 4: 4 beyaz, çağrı; 4 gri olur	[1, 2, 2, 1, 1]
9	4 → 0: 0 **gri** → geri kenar, `true`	[1, 2, 2, 1, 1]
10	`true` 3'e, oradan 0'a geri iletilir	[1, 2, 2, 1, 1]

Tablo 14.16: `donguVar(0, renk)` izlemesi.

7. adım algoritmanın inceliğini gösterir. 3 → 2 okunda 2 daha önce ziyaret edilmiştir. Yalnız "ziyaret edildi mi?" diye bakan bir algoritma burada yanlışlıkla döngü ilan ederdi. Oysa 2'den 3'e dönen bir yol yoktur; 2 siyahtır, işi bitmiştir. 9. adımda ise 0 gridir: 0 → 3 → 4 yolu hâlâ çağrı yığınındadır ve 4 → 0 oku bu yolu kapatır. Bulunan döngü 0 → 3 → 4 → 0'dır. `true` değeri her düzeyde hemen döndürülür; geri kalan oklara bakılmaz.

Sık yapılan hata

`renk[v] = 2` satırını unutmak. Bu durumda işi biten düğümler gri kalır. Okları 0 → 1, 0 → 2, 1 → 2 olan DAG'da 0, 1, 2 sırayla gri olur; 2 ve 1 döner ama gri kalır. 0'a dönülüp 0 → 2 oku incelenince `renk[2] == 1` görülür ve döngü olmadığı hâlde `true` döner. İki renk (ziyaret edildi / edilmedi) yönlü çizgede yetmez; gri ile siyahın ayrımı şarttır.

Döngü tespitinin maliyeti DFS'ninkiyle aynıdır: her düğüm bir kez gri, bir kez siyah olur ve her ok bir kez incelenir. Çizge bağlantısızsa ya da 0'dan her düğüme ulaşılamıyorsa, `donguVar` beyaz kalan her düğümden yeniden çağrılır; toplam yine $O(n + m)$'dir.

14.4 En kısa yol

Bir harita uygulaması iki nokta arasındaki en kısa güzergâhı, bir ağ yönlendiricisi bir paketin en hızlı yolunu, bir oyun karakteri hedefe giden en kısa rotayı arar. Bu soruların hepsi aynı çizge problemidir. Problemin iki sürümü vardır. Ağırlıksız çizgede yolun uzunluğu kenar sayısıdır ve problemi BFS çözer. Ağırlıklı çizgede uzunluk kenar ağırlıklarının toplamıdır ve problemi Dijkstra algoritması çözer.

14.4.1 Ağırlıksız en kısa yol problemi

Tanım 14.12

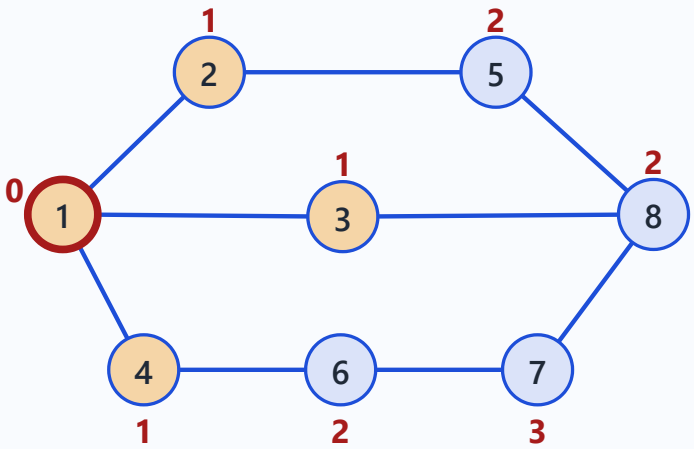
Tek kaynaklı en kısa yol probleminde bir s kaynak düğümü verilir ve s'den çizgedeki her düğüme olan en kısa yolun uzunluğu aranır. Bu uzunluğa s'den o düğüme **uzaklık** denir. s'den ulaşılamayan bir düğümün uzaklığı ∞ kabul edilir.

Ağırlıksız çizgede her kenarın ağırlığı 1 sayılır; yolun uzunluğu kenar sayısıdır. Bu durumda çözüm BFS'dir. BFS düğümleri düzey düzey keşfeder ve düzey k'deki bir düğümün uzaklığı tam olarak k'dir.

Teorem 14.2 (BFS ve en kısa yol)

Ağırlıksız bir çizgede s'den başlayan BFS, her düğümü s'ye uzaklık sırasıyla keşfeder. Bir düğüme ilk kez w düğümünden ulaşıldığında o düğümün uzaklığı $uzaklık(w) + 1$'dir.

İspat. Kuyruğun her an şu biçimde olduğunu gösterelim: önde uzaklığı k olan düğümler, arkada uzaklığı $k + 1$ olan düğümler vardır. Başta kuyrukta yalnız s vardır ve uzaklığı 0'dır. Uzaklığı k olan bir w çıktığında, keşfedilmemiş komşuları kuyruğun arkasına $k + 1$ ile eklenir; düzen bozulmaz. Şimdi uzaklığı $k + 1$ olan bir u düşünelim. s'den u'ya $k + 1$ kenarlı bir yol vardır; bu yolun sondan bir önceki düğümü, uzaklığı k olan bir w"dür. Uzaklığı k'den küçük hiçbir düğüm u'ya komşu değildir; komşu olsaydı u'nun uzaklığı k'den küçük olurdu. Uzaklığı k olanlar ise uzaklığı $k + 1$ olanların hepsinden önce kuyruktan çıkar. Bu yüzden u, uzaklığı k olan bir düğümden keşfedilir ve ona yazılan değer $k + 1$'dir. □



Şekil 14.28: Sekiz düğümlü ağırlıksız çizge; kaynak 1. Kenarlar 1–2, 1–3, 1–4, 2–5, 3–8, 4–6, 5–8, 6–7, 7–8'dir. Düğümlerin yanındaki sayı 1'e uzaklıktır: 2, 3, 4 için 1; 5, 6, 8 için 2; 7 için 3.

Şekil 14.28'de 7'ye iki en kısa yol vardır: 1 → 4 → 6 → 7 ve 1 → 3 → 8 → 7. İkisi de üç kenarlıdır. BFS bunlardan yalnız birini bulur; hangisini bulacağı komşu listelerinin sırasına bağlıdır. Uzunluk ise her durumda 3'tür. En kısa yol tek olmak zorunda değildir, ama en kısa uzunluk tektir.

14.4.2 Algoritma ve kod

```
mesafe[tüm v] = ∞
mesafe[s] = 0
kuyruk.ekle(s)
while kuyruk boş değil:
  w = kuyruk.çıkar()
  for w'nin her komşusu u:
    if mesafe[u] == ∞:
      mesafe[u] = mesafe[w] + 1
      kuyruk.ekle(u)
return mesafe
```

Sözde kod BFS'nin küçük bir değişikliğidir. Ziyaret dizisinin yerini bir mesafe dizisi alır. “mesafe = ∞” aynı zamanda “ziyaret edilmedi” anlamına gelir; ayrı bir işaret dizisine gerek kalmaz. Bir düğüm ilk kez keşfedildiğinde mesafesi, onu keşfeden düğümün mesafesinin bir fazlası olarak yazılır. Teorem 14.2 bu ilk değerin doğru olduğunu söyler; değer bir daha değiştirilmez.

```
Map<Integer, Integer> enKisaYol(int s) {
  final int INF = Integer.MAX_VALUE;  // ∞
  Map<Integer, Integer> mesafe =
      new HashMap<>();
  for (int v : cizge.keySet())
    mesafe.put(v, INF);
  mesafe.put(s, 0);
  Queue<Integer> q = new ArrayDeque<>();
  q.offer(s);
  while (!q.isEmpty()) {
    int w = q.poll();
    for (int u : cizge.get(w))
      if (mesafe.get(u) == INF) {
        mesafe.put(u, mesafe.get(w) + 1);
        q.offer(u);
      }
  }
  return mesafe;
}
```

Java sürümünde çizge `cizge` adlı bir `Map<Integer, List<Integer>>` içinde tutulur: anahtar düğüm, değer onun komşu listesidir. Bu örnekte düğümler 1'den başlayarak numaralandığı için dizi yerine eşleme kullanılmıştır. Kod satır satır şunları yapar:

- `INF`, `Integer.MAX_VALUE` değeriyle ∞'u temsil eder.
- İlk döngü her düğümün mesafesini `INF` yapar; sonra kaynağın mesafesi 0 yapılır ve kaynak kuyruğa konur. `offer`, `add` ile aynı işi yapar; kuyruğun arkasına ekler.
- Ana döngü BFS'dir. Mesafesi hâlâ `INF` olan her komşu için mesafe `mesafe.get(w) + 1` yapılır ve komşu kuyruğa eklenir.

Maliyet BFS'ninki gibidir: *O(n + m)*. `HashMap` işlemleri ortalama *O(1)* kabul edilir.

> **Sık yapılan hata**
>
> ∞ değerine 1 eklemek. `Integer.MAX_VALUE + 1` Java'da taşma yapar ve en küçük negatif sayıya, −2.147.483.648'e döner. Koddaki koşul bu yüzden önemlidir: mesafeye 1 eklenen `w` her zaman kuyruktan çıkmış, yani mesafesi sonlu bir düğümdür. Ağırlıklı çizgede bu tehlike daha büyüktür; Dijkstra'nın Java sürümü mesafeleri bu yüzden `long` türünde tutar ve yalnız kesinleşmiş düğümlerden gevşetme yapar.

14.4.3 Kod izleme: enKisaYol(1)

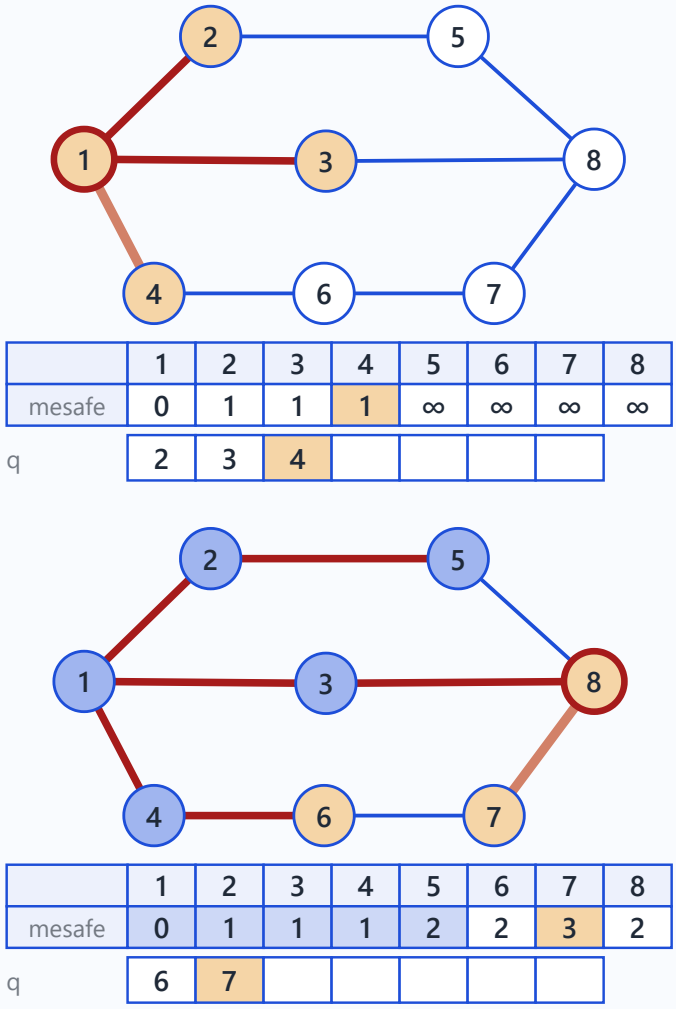
Şekil 14.28'deki çizgede kaynak 1 olsun. Komşu listeleri küçükten büyüğe sıralıdır: 1: [2, 3, 4] · 2: [1, 5] · 3: [1, 8] · 4: [1, 6] · 5: [2, 8] · 6: [4, 7] · 7: [6, 8] · 8: [3, 5, 7].

Çıkan `w`	Komşular	Güncellenen	mesafe (1…8)	Kuyruk
—	—	mesafe[1] = 0	0 ∞ ∞ ∞ ∞ ∞ ∞ ∞	[1]
1	2, 3, 4	2, 3, 4 ← 0 + 1 = 1	0 1 1 1 ∞ ∞ ∞ ∞	[2, 3, 4]
2	1, 5	5 ← 1 + 1 = 2	0 1 1 1 2 ∞ ∞ ∞	[3, 4, 5]
3	1, 8	8 ← 1 + 1 = 2	0 1 1 1 2 ∞ ∞ 2	[4, 5, 8]
4	1, 6	6 ← 1 + 1 = 2	0 1 1 1 2 2 ∞ 2	[5, 8, 6]
5	2, 8	— (ikisi de belli)	0 1 1 1 2 2 ∞ 2	[8, 6]
8	3, 5, 7	7 ← 2 + 1 = 3	0 1 1 1 2 2 3 2	[6, 7]
6	4, 7	— (ikisi de belli)	0 1 1 1 2 2 3 2	[7]
7	6, 8	—	0 1 1 1 2 2 3 2	[]

Tablo 14.17: `enKisaYol(1)` izlemesi. Mesafe sütunu 1…8 düğümlerinin sırasıyla mesafeleridir (∞ = `INF`).

İzlemenin üç noktası öğreticidir. Birincisi, 2 çıktığında komşusu 1'in mesafesi zaten 0'dır ve değişmez. BFS'de bir düğümün ilk bulunan mesafesi en kısadır; sonradan gelen hiçbir yol onu kısaltamaz. İkincisi, 5 çıktığında komşusu 8'in mesafesi 2'dir. 5 üzerinden 8'e gitmek *2 + 1 = 3* verirdi; bu daha uzundur ve kod ona bakmaz bile, çünkü 8'in mesafesi artık `INF` değildir.

Üçüncüsü, 7'nin mesafesini 8 yazmıştır. 8, kuyruğa 6'dan önce girmişti; bu yüzden 7'ye 6'dan önce o ulaşır. Bulunan yol 1 → 3 → 8 → 7'dir. Komşu sırası farklı olsaydı 1 → 4 → 6 → 7 bulunabilirdi; mesafe yine 3 olurdu. Son tabloda her düğümün mesafesi Şekil 14.28'deki değerlerle aynıdır.

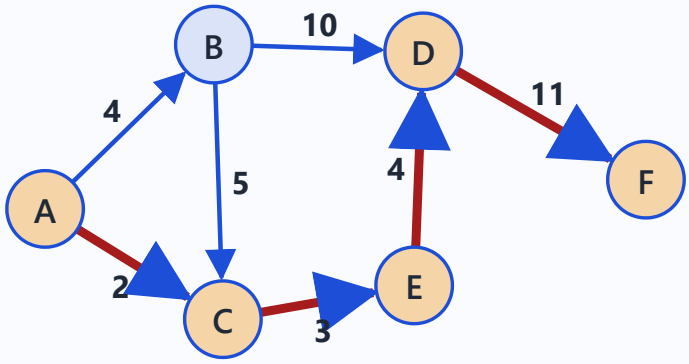


Şekil 14.29: `enKisaYol(1)` 'in iki anı. (a) 1 işlendi; 2, 3, 4'ün mesafesi 1, kuyruk 2, 3, 4. (b) 8 işlendi; 7'nin mesafesi 3 yazıldı, kuyruk 6, 7.

Kod yalnız uzaklıkları verir, yolların kendisini vermez. Yolu da istemek için her düğümü ilk keşfeden düğüm bir `onceki` dizisinde saklanır. Dijkstra'da aynı yöntem kullanılır ve yolun hedeften geriye doğru nasıl kurulduğu 14.4.8'de ayrıntılı olarak izlenir.

14.4.4 Ağırlıklı en kısa yol problemi

Ağırlıklı çizgede problem aynıdır: kaynaktan her düğüme en kısa yol aranır. Bu kez yolun uzunluğu kenar ağırlıklarının toplamıdır. Ağırlık uzaklık, süre ya da maliyet olabilir.



Şekil 14.30: Ağırlıklı yönlü çizge. Oklar A → B: 4, A → C: 2, B → C: 5, B → D: 10, C → E: 3, E → D: 4, D → F: 11'dir. A'dan F'ye en kısa yol A → C → E → D → F'dir: *2 + 3 + 4 + 11 = 20*.

Örnek 14.9

Şekil 14.30'daki çizgede A'dan F'ye en az kenarlı yol A → B → D → F'dir; üç kenarlıdır. BFS bu yolu bulur. Ağırlıklı uzunluğu ise *4 + 10 + 11 = 25*'tir. Dört kenarlı A → C → E → D → F yolu yalnız 20 tutar. Kenar sayısını en aza indiren yol, ağırlık toplamını en aza indiren yol değildir; ağırlıklı çizgede BFS yetmez.

Çözüm, Edsger W. Dijkstra'nın 1956'da bulduğu ve 1959'da yayımladığı algoritmadır. Algoritmanın tek koşulu vardır: **kenar ağırlıkları negatif olmamalıdır**. Bu koşulun neden gerektiği algoritmanın doğruluk gerekçesinde görülür (14.4.9).

14.4.5 Dijkstra algoritması: fikir ve sözde kod

Dijkstra algoritması, her düğüm için bir **geçici mesafe** tutar. Bu değer, o ana kadar bulunan en kısa yolun uzunluğudur ve başta kaynak için 0, öbür düğümler için ∞'dur. Algoritma her adımda şunları yapar:

1. Henüz **kesinleşmemiş** düğümler arasından geçici mesafesi en küçük olanı seçer. Bu düğümün mesafesi artık kesindir.
2. Seçilen düğümün komşularının mesafelerini **gevşetme** ile günceller.

Tanım 14.13

(u, v) kenarını **gevşetmek** (*relaxation*), *u* üzerinden *v*'ye giden yolun bilinen yoldan kısa olup olmadığını denemektir: *alt = mesafe[u] + ağırlık(u, v)* hesaplanır; *alt < mesafe[v]* ise *mesafe[v] = alt* ve *önceki[v] = u* yapılır.

"Gevşetme" adı, mesafe üst sınırının bir yay gibi gevşeyip küçülmesinden gelir. Geçici mesafe hiçbir zaman gerçek uzaklığın altına inmez; her gevşetme onu ya olduğu gibi bırakır ya da gerçek değere biraz daha yaklaştırır.

```
Dijkstra(G, s):
  for her düğüm v:
    mesafe[v] = ∞
    önceki[v] = tanımsız
  mesafe[s] = 0
  Q = tüm düğümler  // öncelikli kuyruk
  while Q boş değil:
    u = Q.çıkar()   // en küçük mesafe[u]
    for u'nun Q'daki her komşusu v:
      alt = mesafe[u] + ağırlık(u, v)
      if alt < mesafe[v]:
        mesafe[v] = alt
        önceki[v] = u
        // Q'da v'nin önceliğini azalt
```

Sözde kodda `Q`, kesinleşmemiş düğümlerin öncelikli kuyruğudur; öncelik, düğümün geçici mesafesidir. `Q.çıkar()` mesafesi en küçük düğümü verir. Bir düğümün mesafesi gevşetmeyle küçülünce kuyruktaki önceliği de küçültülmelidir. `önceki[]` dizisi her düğüm için en kısa yolda kendisinden önce gelen düğümü tutar; yol bu diziden geriye doğru okunarak kurulur.

Sık yapılan hata

Öncelikli kuyrukta önceliği düğüm numarasına göre vermek. Kuyruk, mesafesi en küçük düğümü vermelidir; numarası en küçük düğümü değil. Java'da `PriorityQueue<long[]>` kullanılırken karşılaştırıcı bu yüzden kaydın ikinci öğesini, yani mesafeyi karşılaştırır.

14.4.6 Dijkstra: Java

```
long[] dijkstra(List<List<int[]>> adj, int s, int[] onceki) { // ağırlıklar ≥ 0
    long[] d = new long[adj.size()];
    Arrays.fill(d, Long.MAX_VALUE);  d[s] = 0;  Arrays.fill(onceki, -1);
    PriorityQueue<long[]> pq =                    // {düğüm, mesafe}
        new PriorityQueue<>((a, b) -> Long.compare(a[1], b[1]));
    pq.add(new long[]{s, 0});
    while (!pq.isEmpty()) {
        long[] e = pq.poll();  int u = (int) e[0];
        if (e[1] > d[u]) continue;                  // bayat kayıt: atla
        for (int[] k : adj.get(u)) {
            int v = k[0];  long alt = d[u] + k[1];
            if (alt < d[v]) { d[v] = alt; onceki[v] = u; pq.add(new long[]{v, alt}); }
        }
    }
    return d;
}
```

Kodu parça parça inceleyelim:

- **Girdi.** `adj.get(u)`, *u*'nun komşularını `int[]{v, w}` çiftleri olarak verir: komşu ve kenar ağırlığı. `onceki` dizisini çağıran metot ayırır; `dijkstra` onu doldurur.
- **Başlangıç.** `d` dizisi `long` türündedir ve `Long.MAX_VALUE` ile, yani ∞ ile doldurulur. `d[s] = 0` yapılır, `onceki` −1 ile doldurulur; −1 "önceki yok" anlamındadır.
- **Kuyruk.** Kuyruğa `{düğüm, mesafe}` çiftleri girer. Karşılaştırıcı `Long.compare(a[1], b[1])` mesafeye göre sıralar; `poll` en küçük mesafeli kaydı verir.
- **Bayat kayıt denetimi.** `if (e[1] > d[u]) continue;` satırı, kayıttaki mesafe düğümün güncel mesafesinden büyükse kaydı atlar.
- **Gevşetme.** Her komşu için `alt = d[u] + k[1]` hesaplanır. `alt < d[v]` ise mesafe ve `onceki` güncellenir, kuyruğa yeni bir `{v, alt}` kaydı eklenir.

Java'nın `PriorityQueue` sınıfında "önceliği azalt" işlemi yoktur. Sözde koddaki "Q'da *v*'nin önceliğini azalt" adımının yerine kod, *v* için yeni bir kayıt ekler. Eski kayıt kuyrukta kalır. Eski kayıt sonunda kuyruktan çıktığında, taşıdığı mesafe `d[v]`'den büyüktür, çünkü `d[v]` o arada küçülmüştür. Bu kayda **bayat kayıt** denir ve `continue` ile atlanır. Bu yönteme "tembel silme" de denir: eski kaydı hemen silmek yerine, çıktığında görmezden gelmek.

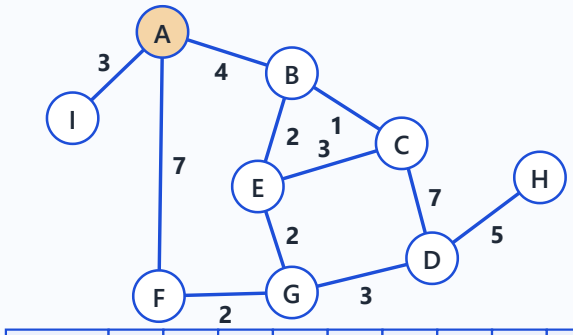
Sık yapılan hata

Bayat kayıt satırını doğruluk için gerekli sanmak ya da tam tersine gereksiz sayıp silmek. Ağırlıklar negatif değilse bu satır silindiğinde mesafeler yine doğru çıkar. Bayat kayıt çıktığında `d[u]` zaten daha küçüktür ve komşular bu küçük değerle önceden gevşetilmiştir; `alt < d[v]` bir daha sağlanmaz. Satırın görevi verimdir: bayat kayıtların komşularını boşuna yeni-

den taramayı önler. Yoğun çizgelerde bu tarama zamanı belirgin biçimde artırır.

14.4.7 Kod izleme: dijkstra(adj, A)

Gezinme örneklerindeki dokuz düğümlü çizgeye ağırlık verelim: A–B: 4, A–F: 7, A–I: 3, B–C: 1, B–E: 2, C–D: 7, C–E: 3, D–G: 3, D–H: 5, E–G: 2, F–G: 2. Çizge yönsüzdür; her kenar iki komşu listesinde de vardır. Kaynak A'dır.



	A	B	C	D	E	F	G	H	I
d[]	0	∞	∞	∞	∞	∞	∞	∞	∞
onceki	–	–	–	–	–	–	–	–	–

pq (mesafeye göre sıralı gösterim; soluk = bayat)

A,0

Şekil 14.31: Dijkstra izlemesinin başlangıcı. `d[A] = 0`, öbür mesafeler ∞; `onceki` tümüyle boş; kuyrukta yalnız `{A, 0}` vardır.

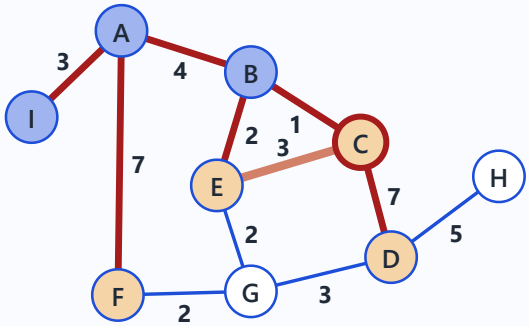
Aşağıdaki tabloda her satır bir `poll` işlemidir. Kuyruk sütunu, o satırdaki gevşetmelerden sonra kuyrukta kalan kayıtları mesafeye göre sıralı gösterir.

Çıkan kayıt	Durum	Gevşetmeler	Kuyruk (sonra)
{A, 0}	A kesinleşir (0)	B: 0 + 4 = 4 < ∞ ✓; F: 0 + 7 = 7 ✓; I: 0 + 3 = 3 ✓	{I,3} {B,4} {F,7}
{I, 3}	I kesinleşir (3)	A: 6 ≥ 0	{B,4} {F,7}
{B, 4}	B kesinleşir (4)	A: 8 ≥ 0; C: 4 + 1 = 5 ✓; E: 4 + 2 = 6 ✓	{C,5} {E,6} {F,7}
{C, 5}	C kesinleşir (5)	B: 6 ≥ 4; D: 5 + 7 = 12 ✓; E: 8 ≥ 6	{E,6} {F,7} {D,12}
{E, 6}	E kesinleşir (6)	B: 8 ≥ 4; C: 9 ≥ 5; G: 6 + 2 = 8 ✓	{F,7} {G,8} {D,12}
{F, 7}	F kesinleşir (7)	A: 14 ≥ 0; G: 9 ≥ 8	{G,8} {D,12}
{G, 8}	G kesinleşir (8)	D: 8 + 3 = 11 < 12 ✓; E: 10 ≥ 6; F: 10 ≥ 7	{D,11} {D,12}
{D, 11}	D kesinleşir (11)	C: 18 ≥ 5; G: 14 ≥ 8; H: 11 + 5 = 16 ✓	{D,12} {H,16}
{D, 12}	**bayat**: 12 > d[D] = 11	atlanır	{H,16}
{H, 16}	H kesinleşir (16)	D: 21 ≥ 11	(boş)

Tablo 14.18: `dijkstra(adj, A)` izlemesi.

İzlemenin kritik anı G'nin işlendiği satırdır. D'nin geçici mesafesi o ana kadar 12'dir; C üzerinden bulunmuştur (A → B → C → D: *4 + 1 + 7*). G kesinleşince D–G kenarı gevşetilir: *8 + 3 = 11 < 12*. D'nin mesafesi 11'e iner, `onceki[D] = G` olur ve kuyruğa `{D, 11}` eklenir. Eski `{D, 12}` kaydı kuyrukta kalır. Önce `{D, 11}` çıkar ve D kesinleşir. Hemen ardından `{D, 12}` çıkar; *12 > 11* olduğu için bayattır ve atlanır.

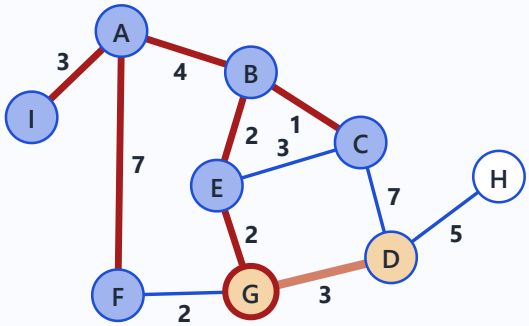
Bir başka gözlem: düğümler mesafe sırasıyla kesinleşir. Sıra A (0), I (3), B (4), C (5), E (6), F (7), G (8), D (11), H (16)'dır. Bu sıra Dijkstra'nın temel özelliğidir ve doğruluğunun da dayanağıdır. Toplam 10 `poll`, 10 ekleme (1 başlangıç ve 9 başarılı gevşetme) ve 22 gevşetme denemesi yapılmıştır; deneme sayısı *2m = 22*'ye eşittir.



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	12	6	7	∞	∞	3
onceki	–	A	B	C	B	A	–	–	A

pq (mesafeye göre sıralı gösterim; soluk = bayat)

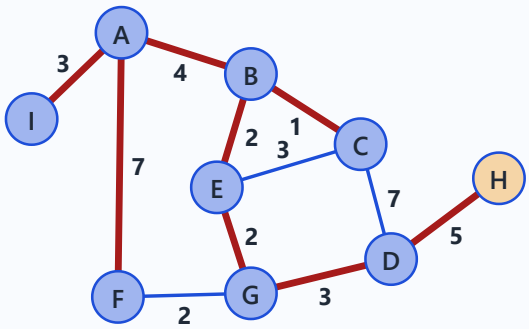
E,6 | F,7 | D,12



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	11	6	7	8	∞	3
onceki	–	A	B	G	B	A	E	–	A

pq (mesafeye göre sıralı gösterim; soluk = bayat)

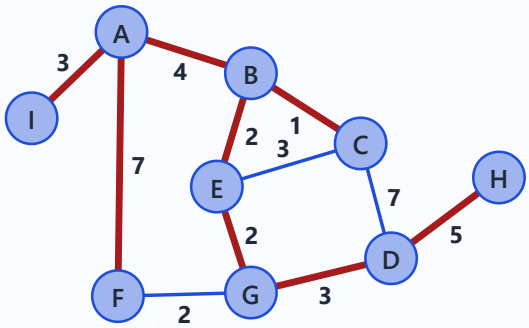
D,11 | ~~D,12~~



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	11	6	7	8	16	3
onceki	–	A	B	G	B	A	E	D	A

pq (mesafeye göre sıralı gösterim; soluk = bayat)

H,16



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	11	6	7	8	16	3
onceki	–	A	B	G	B	A	E	D	A

pq (mesafeye göre sıralı gösterim; soluk = bayat)

(boş)

Şekil 14.32: Dijkstra izlemesinin dört anı. (a) C kesinleşti; `d[D] = 12` (C üzerinden). (b) G üzerinden daha kısa yol bulundu: `d[D] = 11`, `onceki[D] = G`; eski `{D, 12}` kaydı bayat. (c) Bayat kayıt çıktı ve atlandı. (d) Son durum: kalın kenarlar en kısa yol ağacıdır.

	A	B	C	D	E	F	G	H	I
`d`	0	4	5	11	6	7	8	16	3
`onceki`	–	A	B	G	B	A	E	D	A

Tablo 14.19: Dijkstra'nın sonucu (kaynak A).

`onceki` dizisi en kısa yol ağacını tutar. H'den geriye doğru okunursa H ← D ← G ← E ← B ← A elde edilir. A'dan H'ye en kısa yol A → B → E → G → D → H'dir ve uzunluğu *4 + 2 + 2 + 3 + 5 = 16*'dır. Bu yol beş kenarlıdır. A'dan H'ye en az kenarlı yollardan biri ise dört kenarlı A → F → G → D → H'dir; ama bu yolun uzunluğu *7 + 2 + 3 + 5 = 17*'dir.

14.4.8 Yolu geri kurma

Dijkstra'nın bıraktığı `onceki` dizisinden *s*'den *t*'ye yolu çıkarmak için *t*'den başlanır ve `onceki` zinciri −1'e kadar izlenir. Zincir hedeften kaynağa doğru ilerler; yol ise kaynaktan hedefe yazılmalıdır. Bu yüzden her düğüm listenin *başına* eklenir.

```
List<Integer> yol(int[] onceki, int s, int t) {
  LinkedList<Integer> y = new LinkedList<>();
  for (int v = t; v != -1; v = onceki[v])
    y.addFirst(v);
  if (y.getFirst() != s) return List.of();
  return y;                 // s ... t
}
```

Döngü `v = t` ile başlar, her turda `v`'yi listenin başına ekler ve `v = onceki[v]` ile bir adım geri gider. Kaynağın `onceki` değeri −1 olduğu için kaynak eklendikten sonra döngü biter. `LinkedList.addFirst` sabit sürelidir. Döngü, yoldaki düğüm sayısı kadar döner; maliyet *O(yol uzunluğu)*'dur ve en çok *O(n)*'dir.

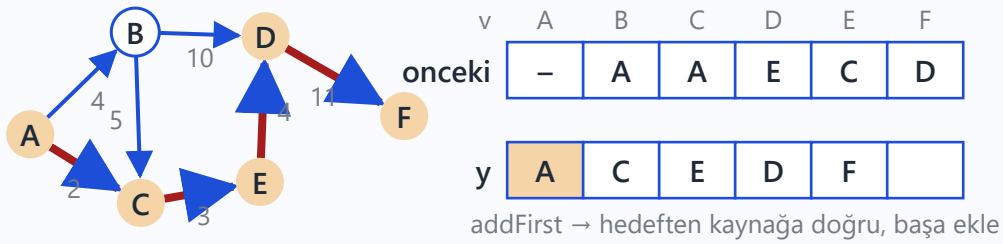
Son satırdaki denetim ulaşılamayan hedefi yakalar. *t*'ye *s*'den ulaşılamıyorsa `onceki[t] = -1` kalmıştır; zincir hemen biter ve listenin ilk öğesi *t*'nin kendisi olur. İlk öğe *s* değilse metot boş liste döndürür.

Örnek 14.10

Şekil 14.30'daki çizgede Dijkstra A'dan çalıştırılınca `onceki` şöyle olur: B ← A, C ← A, D ← E, E ← C, F ← D. `yol(onceki, A, F)` çağrısı:

1. `v = F` : liste [F]; `v = onceki[F] = D` .
2. `v = D` : liste [D, F]; `v = onceki[D] = E` .
3. `v = E` : liste [E, D, F]; `v = onceki[E] = C` .
4. `v = C` : liste [C, E, D, F]; `v = onceki[C] = A` .
5. `v = A` : liste [A, C, E, D, F]; `v = onceki[A] = -1` , döngü biter.

Beş tur dönülmüştür; yolda beş düğüm vardır. İlk öğe A'dır, yani F'ye ulaşılabilir. Sonuç A → C → E → D → F'dir ve uzunluğu `d[F] = 2 + 3 + 4 + 11 = 20` 'dir.



Şekil 14.33: Yolu geri kurmanın sonu. `onceki` zinciri F'den A'ya izlenmiş, her düğüm listenin başına eklenmiştir: y = [A, C, E, D, F].

Sık yapılan hata

`onceki` dizisini başka bir kaynak için kullanmak. Dizi, Dijkstra hangi kaynaktan çalıştırıldıysa o kaynağın en kısa yol ağacıdır. `onceki = [-1, 0, 0, 1, 3]` dizisi 0'dan kurulmuştur; 0'dan 4'e yol 0 → 1 → 3 → 4'tür. Aynı diziyle 2'den 4'e yol aranırsa zincir 4, 3, 1, 0 diye gider ve 2'ye hiç uğramaz; metot boş liste döndürür. 2'den yol bulmak için Dijkstra 2'den yeniden çalıştırılmalıdır.

14.4.9 Dijkstra'nın analizi ve doğruluğu

Zaman. İki gerçekleme yaygındır:

Gerçekleme	Karmaşıklık	Neden?
Öncelikli kuyruk (ikili heap)	O((n + m) log n)	*n* kez en küçüğü çıkarma, en çok *m* kez ekleme ya da güncelleme; her biri O(log n)
Kuyruksuz (dizi taraması)	$O(n^2 + m) = O(n^2)$	Her adımda en küçük mesafeli düğümü bulmak için *n* düğüm taranır; komşu denetimleri toplam *m*

Tablo 14.20: Dijkstra algoritmasının iki gerçeklemesi.

Heap'li sürümde her başarılı gevşetme kuyruğa bir kayıt ekler. Kuyrukta en çok $m + 1$ kayıt bulunur; her ekleme ve çıkarma $O(\log m)$'dir. $m \le n^2$ olduğu için $\log m \le 2 \log n$'dir ve $O(\log m) = O(\log n)$ yazılabilir. Toplam $O((n + m) \log n)$ eder.

Hangisinin seçileceği çizgeye bağlıdır. Seyrek çizgede ($m \approx n$) heap'li sürüm $O(n \log n)$'dir ve dizi taramasının $O(n^2)$'sinden çok daha hızlıdır. Çok yoğun çizgede ($m \approx n^2$) heap'li sürüm $O(n^2 \log n)$'ye çıkar; dizi taraması $O(n^2)$ ile daha iyi olabilir.

Teorem 14.3 (Dijkstra'nın doğruluğu)

Kenar ağırlıkları negatif değilse, Dijkstra algoritmasında bir düğüm kesinleştiği anda geçici mesafesi o düğümün kaynağa gerçek uzaklığına eşittir.

İspat. *u*, kesinleşmemiş düğümler arasında geçici mesafesi en küçük olan düğüm olsun. *s*'den *u*'ya herhangi bir yolu düşünelim. Bu yol *s*'de, yani kesinleşmiş bölgede başlar ve bir yerde bölgeden ilk kez çıkar; ilk dışarıdaki düğüm *y*, ondan önceki düğüm *x* olsun. *x* kesinleşirken *(x, y)* kenarı gevşetilmiştir; bu yüzden *y*'nin geçici mesafesi en çok yolun *s*'den *y*'ye kadarki kısmının uzunluğudur. *u* en küçük geçici mesafeli düğüm olduğu için *mesafe[u] ≤ mesafe[y]*'dir. Yolun *y*'den *u*'ya kalan kısmı negatif olmayan kenarlardan oluşur; uzunluğu ≥ 0'dır. Demek ki yolun tamamı en az *mesafe[y] ≥ mesafe[u]* uzunluğundadır. Hiçbir yol *mesafe[u]*'dan kısa olamaz. □

Kanıtın tek bir yerinde ağırlıkların negatif olmaması kullanılmıştır: "kalan kısmın uzunluğu ≥ 0". Negatif bir kenar varsa, kesinleşmiş bir düğümün mesafesi sonradan negatif kenar üzerinden kısalabilir. Sözde koddaki Dijkstra bu durumda yanlış sonuç verir. Bayat kayıtlı Java sürümü bir düğümü yeniden işleyerek bazen doğru sonucu bulabilir, ama süre garantisi kaybolur; negatif ağırlıklı bir döngü varsa algoritma hiç bitmez. Negatif ağırlıklı çizgelerde Bellman-Ford algoritması kullanılır.

Örnek 14.11

Oklar s → a: 2, s → b: 3, b → a: −2, a → c: 1 olsun. Dijkstra önce s'yi (0), sonra a'yı (2) kesinleştirir; a → c gevşetilir ve c'nin mesafesi 3 olur. Sonra b (3) kesinleşir. b → a üzerinden *3 + (−2) = 1* elde edilir; a'nın gerçek uzaklığı 1'dir. Oysa a, 2 değeriyle çoktan kesinleşmiştir ve sözde koddaki algoritma kesinleşmiş düğümü bir daha işlemez. c de 3 değeriyle kalır; gerçek uzaklığı ise *1 + 1 = 2*'dir. Bayat kayıtlı Java sürümü bu küçük örnekte a'yı 1 değeriyle kuyruğa yeniden ekler. a ikinci kez işlenir ve c 2'ye düzelir. Sonuç bu kez doğrudur, ama bir düğüm birden çok kez işlenmiştir.

14.5 Minimum kapsayan ağaç

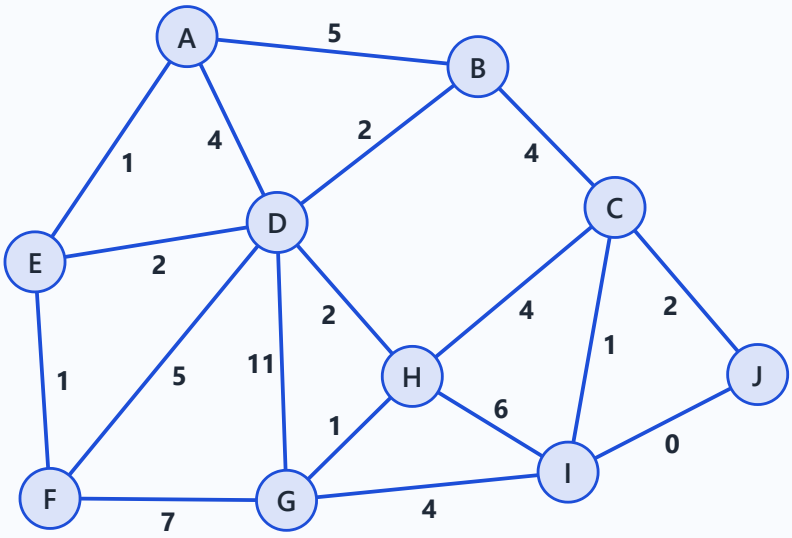
Bir belediye, ilçedeki köylerin hepsine su götürmek istiyor. Her iki köy arasına boru döşemenin bir maliyeti var. Her köyün her köye doğrudan bağlanması gerekmez; suyun her köye bir yolla ulaşması yeterlidir. Toplam maliyeti en düşük boru ağı hangisidir? Bu soru, minimum kapsayan ağaç problemidir. Aynı problem elektrik ve iletişim ağlarının tasarımında, kümeleme yöntemlerinde ve başka algoritmaların alt adımı olarak karşımıza çıkar.

14.5.1 Kapsayan ağaç ve MKA

Tanım 14.14

Bağlantılı bir çizgenin **kapsayan ağacı** (*spanning tree*), çizgenin bütün düğümlerini içeren ve döngü içermeyen bağlantılı bir alt çizgesidir; *n* düğüm için tam *n − 1* kenarı vardır. Ağırlıklı bir çizgenin **minimum kapsayan ağacı** (MKA; *minimum spanning tree*, MST), toplam kenar ağırlığı en düşük olan kapsayan ağaçtır.

Kapsayan ağaçta neden döngü istemiyoruz? Ağırlıklar pozitifse döngüdeki herhangi bir kenarı çıkarmak toplamı azaltır ve bağlantılılığı bozmaz: döngünün öbür kenarları iki ucu hâlâ birbirine bağlar. Bu yüzden en ucuz bağlantılı alt çizge her zaman döngüsüzdür, yani bir ağaçtır. $n - 1$ sayısı da 14.1.7'deki alt sınırdan gelir: n düğümü bağlamak için en az $n - 1$ kenar gerekir ve döngüsüz bağlantılı çizgede tam bu kadar kenar vardır.



Şekil 14.34: On düğüm ve on sekiz ağırlıklı kenardan oluşan yönsüz çizge. Kenarlar ağırlığa göre: I–J 0; A–E, C–I, E–F, G–H 1; B–D, C–J, D–E, D–H 2; A–D, B–C, C–H, G–I 4; A–B, D–F 5; H–I 6; F–G 7; D–G 11.

Bu çizgenin kenar ağırlıklarının toplamı 62'dir. MKA ise yalnız dokuz kenar seçer ve toplamı 14'tür. Olası kapsayan ağaçların sayısı çok büyüktür; hepsini tek tek denemek pratik değildir. İki klasik algoritma, Kruskal ve Prim, MKA'yı açgözlü (*greedy*) bir yöntemle bulur: her adımda o an en iyi görünen seçimi yapar ve bu seçimden hiç geri dönmez. Açgözlü yöntemin bu problemde doğru sonuç vermesinin nedeni aşağıdaki özelliktir.

Teorem 14.4 (Kesim özelliği)

Bir çizgenin düğümleri iki boş olmayan gruba ayrılsın. İki grubu birbirine bağlayan kenarlar arasında ağırlığı en küçük olan kenar, bir minimum kapsayan ağaçta bulunur.

İspat. En hafif kesişen kenar $e = (u, v)$ olsun ve bir MKA T, e'yi içermesin. T bağlantılı olduğu için içinde u'dan v'ye bir yol vardır. u ile v farklı gruplarda olduğundan bu yol iki grup arasında en az bir kez geçer; geçtiği kenar f olsun. T'ye e'yi ekleyip f'yi çıkaralım. Sonuç yine bir kapsayan ağaçtır: e eklenince oluşan tek döngü f çıkınca kırılır, bağlantı bozulmaz. e kesişen kenarların en hafifi olduğu için $ağırlık(e) \le ağırlık(f)$'dir. Yeni ağacın toplamı T'ninkinden büyük olamaz; o da bir MKA'dır ve e'yi içerir. □

Kruskal ve Prim bu teoremin iki farklı kullanımıdır. Prim'de gruplardan biri o ana kadar kurulan ağaçtır. Kruskal'da ise seçilen kenarın iki ucunun ait olduğu ağaçlar ayrı gruplar olarak düşünülür.

14.5.2 Ayrık kümeler (Union-Find)

Kruskal algoritması kenarları hafiften ağıra dener ve her kenar için aynı soruyu sorar: "Bu kenar, şimdiye kadar seçilenlerle bir döngü kurar mı?" Kenarın iki ucu seçilen kenarlarla zaten birbirine bağlıysa yeni kenar bir döngü kapatır. Bu soruyu hızlı yanıtlayan yapı **ayrık kümeler** (*disjoint sets*, *union-find*) yapısıdır.

Yapı, çakışmayan kümelerden oluşan bir aileyi iki işlemle yönetir:

- `find(x)` : x'in kümesinin **temsilcisini** döndürür. İki öğe aynı kümedeyse `find` ikisi için aynı temsilciyi verir.
- `union(u, v)` : u ile v farklı kümelerdeyse iki kümeyi birleştirir ve `true` döndürür. Zaten aynı kümedeyseler hiçbir şey yapmaz ve `false` döndürür.

Sunum yalnız bu iki işlemin davranışını kullanır; yapının iç ayrıntısını vermez. Aşağıdaki sınıf, sunumdaki izlemenin ebeveyn oklarını aynen üreten bir gerçeklemedir.

```
class UnionFind {
    private final int[] ebeveyn, boyut;
    UnionFind(int n) {                       // her düğüm kendi kümesi
        ebeveyn = new int[n];  boyut = new int[n];
        for (int i = 0; i < n; i++) { ebeveyn[i] = i; boyut[i] = 1; }
    }
    int find(int x) {                        // kökü bul, yolu kısalt
        if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
        return ebeveyn[x];
    }
    boolean union(int u, int v) {
        int a = find(u), b = find(v);
        if (a == b) return false;            // aynı küme: döngü kurar
        if (boyut[a] > boyut[b]) { int t = a; a = b; b = t; }
        ebeveyn[a] = b;  boyut[b] += boyut[a];  // küçük ağaç büyüğün altına
        return true;
    }
}
```

Her küme bir ağaç olarak tutulur. `ebeveyn[x]`, x'in ağaçtaki ebeveynidir; kökün ebeveyni kendisidir ve kök, kümenin temsilcisidir. Başta her düğüm kendi kümesidir: n tane tek düğümlü ağaç vardır. İki iyileştirme yapının hızını belirler:

- **Boyuta göre birleştirme.** `union`, küçük ağacın kökünü büyük ağacın köküne bağlar. Böylece ağaçların yüksekliği en çok $\log n$ olur. Boyutlar eşitse u'nun kökü v'nin köküne bağlanır.

- **Yol kısaltma** (*path compression*). `find` kökü bulurken yol üzerindeki her düğümü doğrudan köke bağlar. Aynı düğüm bir daha sorulduğunda kök tek adımda bulunur.

İki iyileştirme birlikte kullanılınca bir dizi işlemin işlem başına maliyeti $O(\alpha(n))$ olur. Buradaki α, ters Ackermann fonksiyonudur ve son derece yavaş büyür. Evrendeki atom sayısı kadar büyük bir n için bile $\alpha(n) \le 4$'tür. Pratikte `find` ve `union` sabit süreli kabul edilebilir.

14.5.3 Kruskal algoritması

```
Kruskal(G):
  T = ∅          // seçilen kenarlar
  for her düğüm v: makeSet(v)
  kenarları ağırlığa göre sırala
  for her (u, v), küçükten büyüğe:
    if find(u) != find(v): // döngü yok
      T.ekle((u, v))
      union(u, v)
    if |T| == n - 1: break
  return T  // MST (bağlı değilse: orman)
```

Kruskal'ın kuralı tek cümledir: **döngü oluşturmayan en hafif kenarı seç.** Algoritma başta n ayrık küme ile başlar; her düğüm tek başına bir ağaçtır. Kenarlar ağırlığa göre küçükten büyüğe sıralanır ve sırayla denenir. Kenarın iki ucu farklı kümelerdeyse kenar T'ye eklenir ve iki küme birleşir. Aynı kümedeyse kenar bir döngü kapatır ve atlanır. $n - 1$ kenar seçilince ağaç tamamlanmıştır; kalan kenarlara bakılmaz.

Ara durumda seçilen kenarlar tek bir ağaç değil, birden çok ağaçtan oluşan bir **ormandır**. Ağaçlar algoritma ilerledikçe birleşir ve sonunda tek ağaç kalır. Çizge bağlantısızsa $n - 1$ kenara hiç ulaşılamaz. Bu durumda algoritma bütün kenarları dener ve her bağlantılı bileşen için bir ağaçtan oluşan **minimum kapsayan ormanı** döndürür.

Aşama	Maliyet
Kenarları sıralama	$O(m \log m)$
$2m$ `find` + $(n-1)$ `union`	$O(m \cdot \alpha(n))$
Toplam	**$O(m \log n)$**

Tablo 14.21: Kruskal algoritmasının maliyeti.

Baskın terim sıralamadır. $m \le n^2$ olduğu için $\log m \le 2 \log n$'dir; bu yüzden $O(m \log m) = O(m \log n)$ yazılır. Union-Find işlemleri neredeyse doğrusaldır ve toplamı değiştirmez. Döngü denetimi Union-Find yerine her kenar için bir DFS ile yapılsaydı, her denetim $O(n)$ sürer ve toplam $O(m \cdot n)$'ye çıkardı.

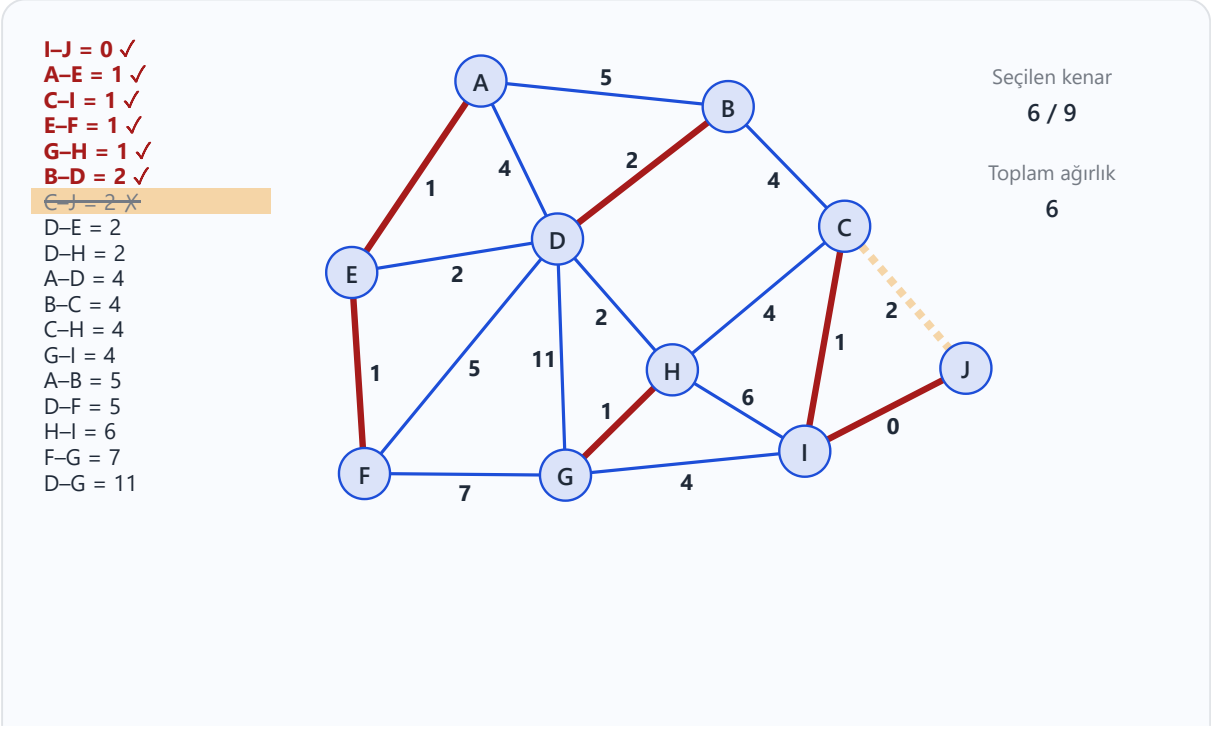
14.5.4 Kruskal adım adım

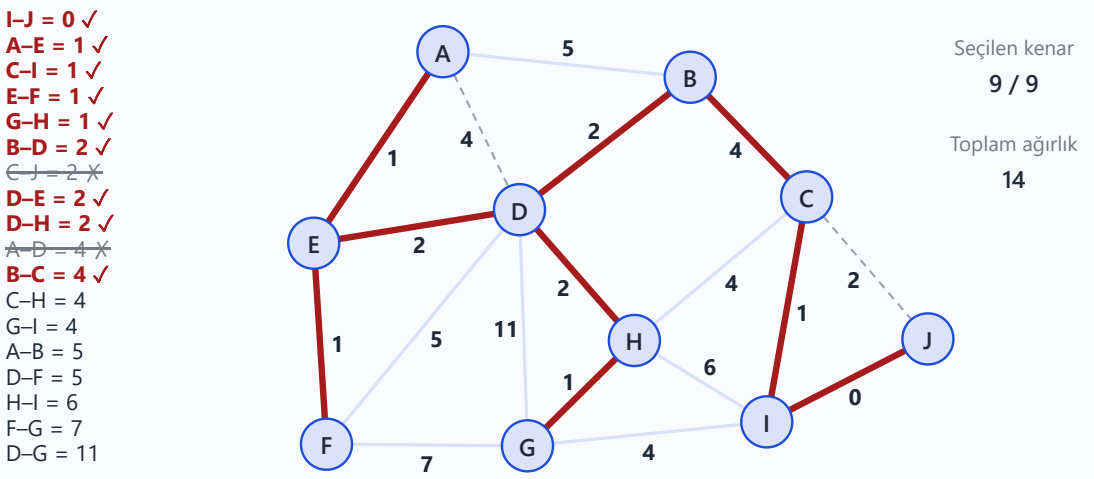
Kruskal'ı Şekil 14.34'teki çizgede çalıştıralım. Eşit ağırlıklı kenarlar Şekil 14.34'ün altyazısındaki sırayla denenir.

Kenar	Ağırlık	Karar	Birleşen kümeler / gerekçe	|T|	Toplam
I–J	0	kabul	{I} + {J}	1	0
A–E	1	kabul	{A} + {E}	2	1
C–I	1	kabul	{C} + {I, J}	3	2
E–F	1	kabul	{A, E} + {F}	4	3
G–H	1	kabul	{G} + {H}	5	4
B–D	2	kabul	{B} + {D}	6	6
C–J	2	**ret**	C ve J zaten {C, I, J} kümesinde	6	6
D–E	2	kabul	{B, D} + {A, E, F}	7	8
D–H	2	kabul	{A, B, D, E, F} + {G, H}	8	10
A–D	4	**ret**	A ve D aynı kümede	8	10
B–C	4	kabul	büyük küme + {C, I, J}	9	14

Tablo 14.22: Kruskal'ın denediği kenarlar ve kararları.

C–J kenarının neden reddedildiğine bakalım. Daha önce C–I ve I–J kenarları seçilmiştir; C ile J, C–I–J yoluyla zaten bağlıdır. C–J eklenseydi C–I–J–C döngüsü oluşurdu. A–D de aynı nedenle reddedilir: A ile D, A–E–D yoluyla bağlıdır. B–C kabul edilince $|T| = 9 = n - 1$ olur ve algoritma durur. Toplam ağırlık $0 + 1 + 1 + 1 + 1 + 2 + 2 + 2 + 4 = 14$'tür. C–H, G–I, A–B, D–F, H–I, F–G ve D–G kenarlarına, yani kalan yedi kenara hiç bakılmaz.





Şekil 14.35: Kruskal'ın iki anı. Üstte C–J kenarı reddediliyor: C ve J aynı ağaçtadır. Altta son durum: seçilen dokuz kenar A–E, E–F, D–E, B–D, D–H, G–H, B–C, C–I, I–J'dir ve toplam ağırlık 14'tür. Soldaki listede kabul edilen kenarlar işaretli, reddedilenler üstü çizilidir.

Sık yapılan hata

Bir kenarın döngü kurup kurmadığına yalnız iki ucunun daha önce seçilmiş bir kenara dokunup dokunmadığına bakarak karar vermek. D–E denendiğinde D de E de daha önce seçilmiş kenarlara bağlıdır (B–D ve A–E), ama farklı ağaçlardadır; kenar kabul edilir. Belirleyici olan uçların aynı *kümede* olup olmadığıdır, seçilmiş olup olmadıkları değil.

14.5.5 Kod izleme: kruskal() ve Union-Find

```
List<int[]> kruskal(int n, List<int[]> E) {
  E.sort((a, b) ->          // {u, v, w}
      Integer.compare(a[2], b[2]));
  UnionFind uf = new UnionFind(n);
  List<int[]> T = new ArrayList<>();
  for (int[] e : E) {
    if (uf.union(e[0], e[1]))  // döngüsüz
      T.add(e);
    if (T.size() == n - 1) break;
  }
  return T;
}
```

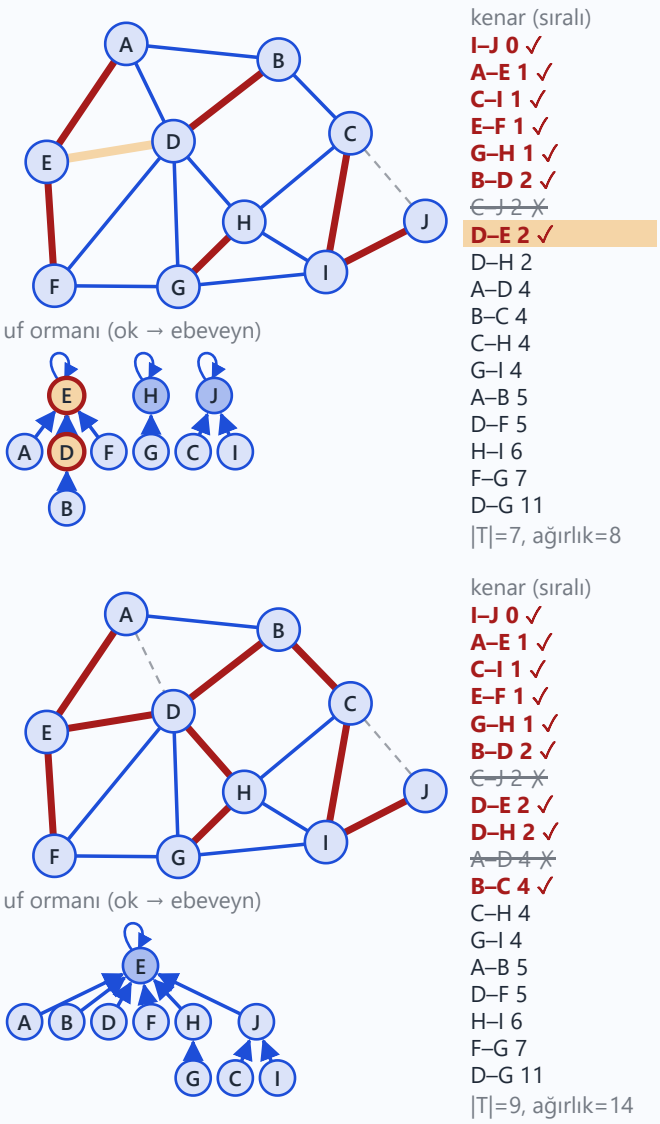
Kod, sözde kodun doğrudan karşılığıdır. Kenarlar `{u, v, w}` dizileri olarak verilir. `E.sort` kenarları üçüncü öğeye, yani ağırlığa göre sıralar. Java'nın nesne listelerini sıralama yöntemi **kararlıdır**: eşit ağırlıklı kenarlar girdideki sıralarını korur. `uf.union(e[0], e[1])` hem döngü denetimini hem birleştirmeyi tek çağrıda yapar: `true` dönerse kenar eklenir. $|T| = n - 1$ olunca döngü `break` ile kesilir.

Union-Find'ın içini izlemek için düğümleri A = 0, …, J = 9 diye numaralayalım. Aşağıdaki tabloda “bağlanan” sütunu, `union` içinde hangi kökün hangi köke bağlandığını gösterir.

Kenar	`find` kökleri	`union`	Bağlanan	Değişen ebeveynler
I–J (0)	I, J	true	I → J (boyutlar eşit)	I → J
A–E (1)	A, E	true	A → E (eşit)	A → E
C–I (1)	C, J	true	C → J (1 < 2)	C → J
E–F (1)	E, F	true	F → E (1 < 2)	F → E
G–H (1)	G, H	true	G → H (eşit)	G → H
B–D (2)	B, D	true	B → D (eşit)	B → D
C–J (2)	J, J	**false**	—	—
D–E (2)	D, E	true	D → E (2 < 3)	D → E
D–H (2)	E, H	true	H → E (2 < 5)	H → E
A–D (4)	E, E	**false**	—	—
B–C (4)	E, J	true	J → E (3 < 7)	B → E (yol kısaltma), J → E

Tablo 14.23: `kruskal()` sırasında Union-Find ormanı.

Tablodaki iki an özellikle öğreticidir. C–J denendiğinde `find(C)`, C → J okunu izleyip J'yi; `find(J)` doğrudan J'yi verir. Kökler aynıdır, `union` `false` döndürür ve kenar atlanır. Son adımda `find(B)`, B → D → E yolunu izler ve yol kısaltma B'yi doğrudan E'ye bağlar. Sonra J'nin ağacı (3 düğüm) E'nin ağacının (7 düğüm) altına girer. Son ormanda tek kök E'dir; I ve C, J üzerinden, G ise H üzerinden E'ye ulaşır.

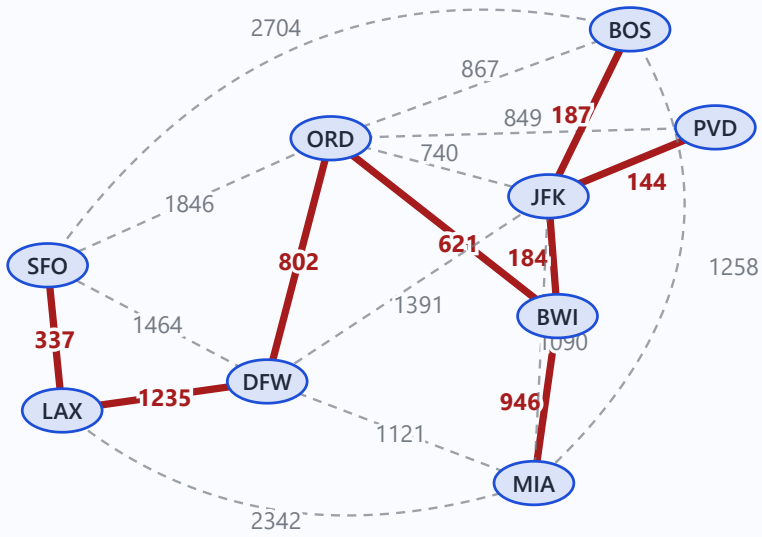


Şekil 14.36: Union-Find ormanının iki anı (oklar ebeveyni gösterir). (a) D–E kabul edildi: D'nin ağacı E'nin altına girdi. (b) Son durum: tek ağaç, kök E. B'nin oku yol kısaltmayla doğrudan E'yi gösterir.

`return T` dokuz kenar döndürür; toplam ağırlık 14'tür. On sekiz kenardan on biri denenmiş, yedisine hiç bakılmamıştır. Sıralama yine de on sekiz kenarın hepsini kapsar; algoritmanın baskın maliyeti budur.

14.5.6 İkinci örnek: havalimanları

Goodrich, Tamassia ve Goldwasser'in kitabındaki örnekte dokuz ABD havalimanı ve aralarındaki 19 uçuş hattı vardır; ağırlıklar mil cinsinden uzaklıklardır. Bütün havalimanlarını en kısa toplam hat uzunluğuyla birbirine bağlayan ağ aranıyor.



Şekil 14.37: Dokuz havalimanı ve 19 hat. Kalın kenarlar Kruskal'ın bulduğu MKA'dır: JFK–PVD 144, JFK–BWI 184, JFK–BOS 187, SFO–LAX 337, ORD–BWI 621, ORD–DFW 802, BWI–MIA 946, LAX–DFW 1235. Kesikli kenarlar seçilmez.

Hat (mil)	Karar	Gerekçe
JFK–PVD 144, JFK–BWI 184, JFK–BOS 187	kabul	JFK çevresinde dört havalimanı birleşir
SFO–LAX 337	kabul	batıda ayrı bir ağaç
ORD–BWI 621	kabul	ORD doğu ağacına katılır
ORD–JFK 740	ret	ORD ile JFK, BWI üzerinden zaten bağlı (döngü)
ORD–DFW 802	kabul	DFW doğu ağacına katılır
ORD–PVD 849, ORD–BOS 867	ret	iki uç da doğu ağacında
BWI–MIA 946	kabul	MIA doğu ağacına katılır
JFK–MIA 1090, DFW–MIA 1121	ret	iki uç da doğu ağacında
LAX–DFW 1235	kabul	batı ve doğu ağaçları birleşir; 8 = n − 1 kenar

Tablo 14.24: Havalimanı örneğinde Kruskal'ın kararları.

Toplam *144 + 184 + 187 + 337 + 621 + 802 + 946 + 1235 = 4456* mildir. On üç hat denenmiş, en uzun altı hat (1258, 1391, 1464, 1846, 2342, 2704) hiç denenmemiştir. Örnek, algoritmanın davranışını iyi gösterir: kısa hatlar önce birbirine yakın havalimanlarını gruplar; uzun hatlar yalnız gruplar arasında köprü gerektiğinde seçilir. LAX–DFW, batı grubu ile doğu grubu arasındaki en kısa köprüdür; kesim özelliğine göre MKA'da bulunmak zorundadır.

14.5.7 Prim algoritması

```
Prim(G, s):
  ağaç = {s}, T = ∅
  s'nin kenarlarını kuyruğa ekle
  while |T| < n − 1:
    (u, v) = en hafif kenar  // u ağaçta
    if v ağaçta: atla        // döngü
    v'yi ağaca, (u, v)'yi T'ye ekle
    v'nin dışarı giden kenarlarını
      kuyruğa ekle
  return T
```

Prim algoritması Kruskal'dan farklı bir yol izler. Ağaç tek bir düğümle başlar ve her adımda bir düğüm büyür. Her adımda, ağaçtaki bir düğümü ağaç *dışındaki* bir düğüme bağlayan kenarların en hafifi seçilir. Kesim özelliği bu seçimin güvenli olduğunu söyler: ağaç ile dışarısı iki gruptur; ağaç dışarıya bir yolla bağlanmak zorundadır ve seçilen kenar bu bağlantıların en ucuzudur.

Prim'in Dijkstra'ya benzerliği dikkat çekicidir. İkisi de bir düğümden başlar, bir öncelikli kuyruk kullanır ve her adımda kuyruktan en küçük öğeyi alır. Fark kuyruğun anahtarındadır. Dijkstra'da anahtar düğümün kaynağa *toplam* uzaklığıdır. Prim'de anahtar, düğümü ağaca bağlayan *tek kenarın* ağırlığıdır. Kruskal'dan farkı ise ara durumdadır: Prim'de her an tek bir ağaç vardır. Bir kenarın döngü kurup kurmadığını anlamak için "hedef düğüm zaten ağaçta mı?" sorusu yeterlidir; Union-Find gerekmez.

14.5.8 Prim: Java

Sunumdaki Java sürümü öncelikli kuyrukla yazılmış **tembel** sürümdür.

```
List<int[]> prim(List<List<int[]>> adj, int s) {
  int n = adj.size();                 // adj.get(u): {v, w}
  boolean[] agacta = new boolean[n];
  PriorityQueue<int[]> pq = new PriorityQueue<>(
      (a, b) -> Integer.compare(a[2], b[2]));  // {u, v, w}
  List<int[]> T = new ArrayList<>();
  agacta[s] = true;
  for (int[] e : adj.get(s)) pq.add(new int[]{s, e[0], e[1]});
  while (!pq.isEmpty() && T.size() < n - 1) {
    int[] e = pq.poll();               // en hafif aday
    if (agacta[e[1]]) continue;        // bayat: döngü kurar
    agacta[e[1]] = true;
    T.add(e);
    for (int[] f : adj.get(e[1]))
      if (!agacta[f[0]]) pq.add(new int[]{e[1], f[0], f[1]});
  }
  return T;
}
```

Kuyrukta `{u, v, w}` kayıtları bulunur: `u` ağaçtaki uç, `v` aday düğüm, `w` kenarın ağırlığıdır. Karşılaştırıcı yalnız ağırlığa bakar. `agacta` dizisi hangi düğümlerin ağaca girdiğini tutar. Kod şu adımları izler:

1. Başlangıç düğümü `s` ağaca konur; `s`'nin bütün kenarları kuyruğa eklenir.
2. Döngü, kuyruk boşalana ya da $n - 1$ kenar seçilene kadar sürer. Her turda en hafif kayıt alınır.
3. Kaydın aday ucu `e[1]` zaten ağaçtaysa kayıt **bayattır**: bu kenar iki ağaç düğümünü bağlar ve döngü kurar. `continue` ile atlanır.
4. Değilse aday ağaca alınır, kenar T'ye eklenir. Yeni düğümün ağaç dışına giden bütün kenarları kuyruğa eklenir.

"Tembel" sözcüğü bayat kayıtların ele alınış biçimini anlatır. v ağaca girdiğinde, v'ye giden ve kuyrukta bekleyen eski kayıtlar silinmez; kuyruktan çıktıklarında atlanırlar. Dijkstra'daki bayat kayıt yöntemiyle aynı fikirdir.

Maliyet. Her kenar kuyruğa en çok bir kez girer: kenar, ağaca ilk giren ucundan eklenir; öbür ucu ağaca girdiğinde karşı uç zaten ağaçta olduğu için `!agacta[f[0]]` denetimi kenarı yeniden eklemez. Kuyrukta en çok m ekleme ve m çıkarma yapılır; her biri $O(\log m)$'dir. Toplam $O(m \log m) = O(m \log n)$, ek bellek ise kuyruk için $O(m)$'dir. Komşuluk matrisiyle ve kuyruksuz, dizi taramalı bir sürümle Prim $O(n^2)$'de çalışır; çok yoğun çizgelerde bu sürüm daha iyidir.

> **Sık yapılan hata**
>
> Bayat kayıt denetimini (`if (agacta[e[1]]) continue;`) gereksiz sanıp silmek. Dijkstra'da benzer satır yalnız verim içindi; Prim'de ise doğruluk için gereklidir. Satır silinirse, v ağaca girdikten sonra kuyrukta kalan ve v'ye giden bir kayıt çıktığında iki ucu da ağaçta olan bir kenar T'ye eklenir. Ağaçta döngü oluşur ve sonuç kapsayan ağaç olmaktan çıkar.

14.5.9 Prim adım adım

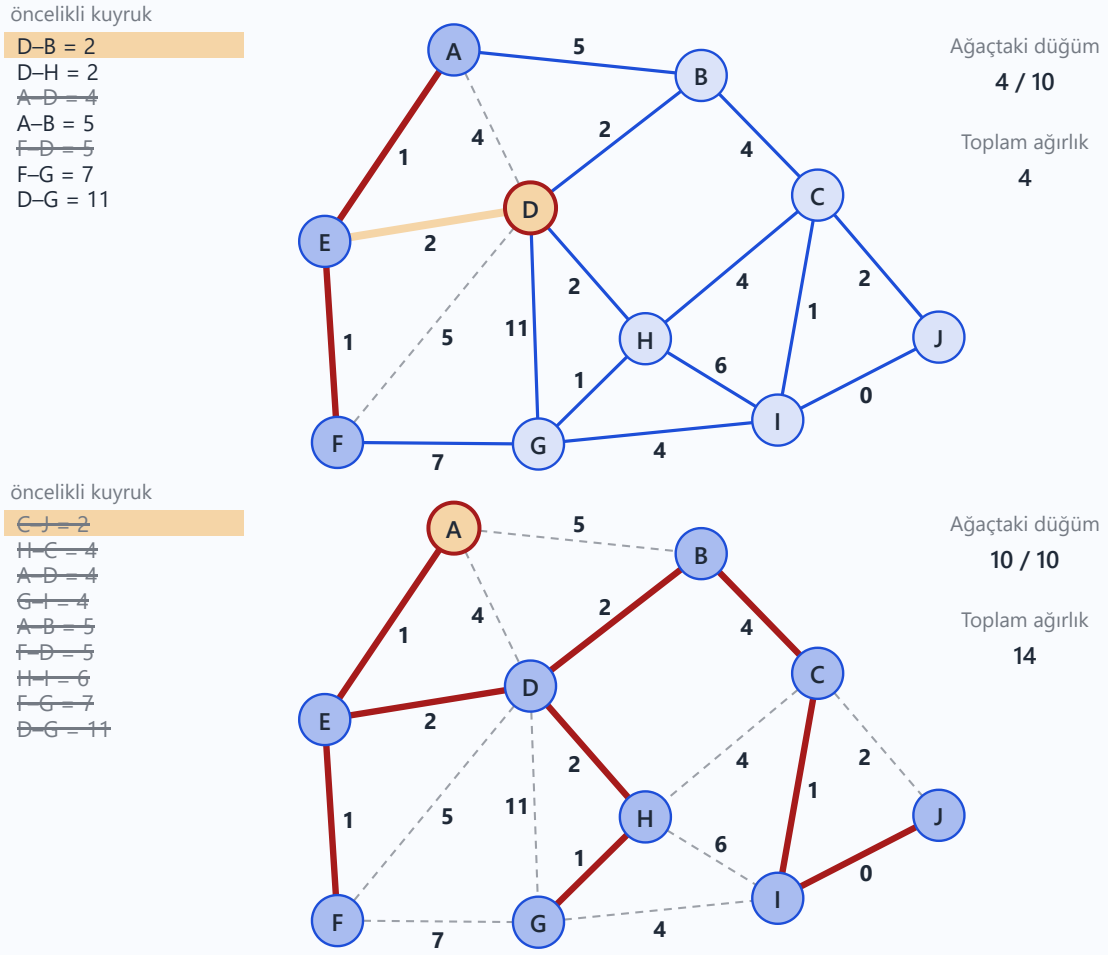
Prim'i Şekil 14.34'teki çizgede A'dan başlatalım. Elle çalıştırırken eşit ağırlıklı kayıtlar alfabetik sırayla alınsın: önce ağaçtaki uç, sonra aday uç alfabeye göre karşılaştırılır.

Adım	Seçilen kenar	Ağaca giren	Kuyruğa yeni eklenen adaylar	Toplam
0	—	A	A–E 1, A–D 4, A–B 5	0
1	A–E (1)	E	E–D 2, E–F 1	1
2	E–F (1)	F	F–D 5, F–G 7	2
3	E–D (2)	D	D–B 2, D–G 11, D–H 2	4
4	D–B (2)	B	B–C 4	6
5	D–H (2)	H	H–C 4, H–G 1, H–I 6	8
6	H–G (1)	G	G–I 4	9

Adım	Seçilen kenar	Ağaca giren	Kuyruğa yeni eklenen adaylar	Toplam
7	B–C (4) (önce bayat A–D 4 atlanır)	C	C–I 1, C–J 2	13
8	C–I (1)	I	I–J 0	14
9	I–J (0)	J	—	14

Tablo 14.25: Prim'in A'dan izlemesi (eşit ağırlıklarda alfabetik sıra).

3. adımda D ağaca girer. Bu anda kuyruktaki A–D 4 ve F–D 5 kayıtları bayatlamıştır: ikisinin de iki ucu ağaçtadır. Kuyruktan hemen silinmezler. 7. adımda en hafif kayıtlar dörder ağırlıklıdır ve alfabetik sırada önce A–D gelir; A–D bayat olduğu için atlanır, ardından B–C seçilir. 9. adımda J ağaca girer, $n - 1 = 9$ kenar seçilmiştir ve algoritma durur. Kuyrukta hâlâ sekiz kayıt vardır; bunlara bakılmaz.



Şekil 14.38: Prim'in iki anı. Üstte E–D seçildi ve D ağaca girdi; üstü çizili kayıtlar bayatlamıştır. Altta son durum: ağaçta on düğüm, toplam ağırlık 14.

Prim ağacı A–E, E–F, E–D, D–B, D–H, H–G, B–C, C–I, I–J'dir. Kenar kümesi Kruskal'ınkiyle aynıdır, yalnız kenarlar farklı sırada seçilmiştir. Kruskal işe ağırlığı 0 olan I–J ile başlamıştı; Prim bu kenarı ancak en son, I ağaca girdikten sonra görebilir.

Alfabetik sıra elle izleme için seçilmiş bir kuraldır. Sunumdaki Java kodu eşit ağırlıklı kayıtları `PriorityQueue`'nun iç düzenine göre çıkarır ve bu düzen hiçbir sıra garantisi vermez. Kodun eşit ağırlıklarda hangi kaydı önce çıkaracağı, kayıtların kuyruğa hangi sırayla girdiğine, dolayısıyla komşu listelerinin sırasına bağlıdır. Kod bu çizgede çalıştırıldığında komşu listeleri alfabetik sıralıysa 7. adımda B–C yerine G–I seçilir, ardından I–J ve I–C gelir; ağaç A–E, E–F, E–D, D–B, D–H, H–G, G–I, I–J, I–C olur. Kenarlar listelere Şekil 14.34'ün altyazısındaki sırayla eklenmişse 7. adımda H–C seçilir. Bu ağaçların her biri bir MKA'dır ve toplamları yine 14'tür. Eşit ağırlıklı kenarlar varsa birden çok MKA olabilir; toplam ağırlık ise her zaman aynıdır.

Örnek 14.12

Prim aynı çizgede J'den başlatılsın (eşitlikte alfabetik sıra).

1. J'nin kenarları kuyruğa girer: J–I 0, J–C 2. En hafifi J–I 0'dır; I ağaca girer. Yeni adaylar I–C 1, I–G 4, I–H 6.
2. En hafif kayıt I–C 1'dir; C ağaca girer. Yeni adaylar C–B 4, C–H 4.
3. Sıradaki en hafif kayıt J–C 2'dir, ama C artık ağaçtadır: kayıt bayattır ve atlanır. Kuyrukta dörder ağırlıklı C–B, C–H ve I–G vardır; alfabetik sırada C–B önce gelir ve B ağaca girer.

İlk üç seçilen kenar J–I (0), I–C (1) ve C–B (4)'tür. Algoritma B–D, D–E, E–A, E–F, D–H, H–G ile devam eder ve toplam yine 14 olur. Başlangıç düğümü ağacın kuruluş sırasını değiştirir, MKA'nın toplam ağırlığını değiştirmez.

14.5.10 Kruskal ile Prim'in karşılaştırması

	Kruskal	**Prim**
Fikir	kenarları küçükten büyüğe dener, ağaçları birleştirir	tek ağacı bir düğümden başlayıp büyütür
Ara durum	orman (birden çok ağaç)	tek ağaç
Veri yapısı	sıralı kenar listesi ve Union-Find	öncelikli kuyruk, işaret dizisi, komşuluk listesi
Döngü denetimi	`find(u) == find(v)`	*v* zaten ağaçta mı?
Maliyet	O(m log n) (sıralama baskın)	O(m log n) heap ile; matrisle $O(n^2)$
Ne zaman?	seyrek çizge, kenarlar liste olarak hazır	komşuluk listesi hazır; çok yoğun çizgede $O(n^2)$ sürüm

Tablo 14.26: Kruskal ve Prim algoritmalarının karşılaştırması.

İki algoritma da açgözlüdür ve kesim özelliğine dayanır. Aynı çizgede aynı toplam ağırlığı bulurlar; örneğimizde ikisi de 14'tür. Eşit ağırlıklı kenarlar varsa buldukları ağaçlar farklı olabilir. Seçim, verinin elde hangi biçimde olduğuna da bağlıdır. Kenarlar bir dosyadan liste olarak okunuyorsa Kruskal doğaldır. Çizge zaten komşuluk listesiyle tutuluyorsa ve bir düğümden başlamak anlamlıysa Prim doğaldır.

Sık yapılan hata

MKA ile en kısa yol ağacını karıştırmak. Dijkstra'nın `onceki` dizisi de bir ağaç kurar, ama bu ağaç kaynaktan her düğüme olan *yol uzunluklarını* en aza indirir. MKA ise kenarların *toplamını* en aza indirir. Kenarları A–B 2, B–C 2, A–C 3 olan üçgende MKA, A–B ve B–C kenarlarından oluşur (toplam 4). Bu ağaçta A'dan C'ye yol 4 uzunluğundadır; oysa A'dan C'ye en kısa yol doğrudan A–C kenarıdır ve 3 uzunluğundadır. İki problemin cevabı genel olarak farklıdır.

14.6 Özet

- **Çizge** $G = (V, E)$, düğümler ve onları bağlayan kenarlardan oluşur; n düğüm, m kenar sayısıdır. Ağaçtan farklı olarak kökü yoktur, döngü içerebilir ve bir düğüme birden çok yoldan ulaşılabilir. Kenarlar yönlü ya da yönsüz, ağırlıklı ya da ağırlıksız olabilir.
- **Derece** bir düğüme bağlı kenar sayısıdır. El sıkışma lemmasına göre $\Sigma \deg(v) = 2m$'dir; yönlü çizgede giriş ve çıkış derecelerinin toplamlarının ikisi de m'dir. Basit yönsüz çizgede $m \le n(n-1)/2$, bağlantılı çizgede $m \ge n - 1$'dir.
- **Yol** kenarlarla bağlı düğüm dizisi, **döngü** aynı düğümde başlayıp biten yoldur. Her iki düğümü arasında yol olan çizge **bağlantılıdır**; birbirine ulaşabilen düğümlerin en büyük grubu bir **bağlantılı bileşendir**.
- **Komşuluk matrisi** $O(n^2)$ bellek kullanır; kenar sorgusu $O(1)$, komşuları gezmek $O(n)$'dir. **Komşuluk listesi** $O(n + m)$ bellek kullanır; komşuları gezmek $O(\deg v)$'dir. Seyrek çizgede liste, yoğun çizgede ve sık kenar sorgusunda matris seçilir.
- **BFS** kuyrukla düzey düzey, **DFS** yığınla ya da özyinelemeyle derine doğru ilerler. İkisi de komşuluk listesiyle $O(n + m)$'dir. Düğüm kuyruğa ya da yığına girerken işaretlenir; işaret olmadan gezinme döngülerde sonsuza kadar döner.
- Üç uygulama bu gezinmelerin üzerine kurulur. **Bağlantılı bileşenler**, boyanmamış her düğümden yeni bir gezinme başlatılarak bulunur. **İki parçalılık testi** BFS ile 2-renklemedir; tek uzunlukta döngü yoksa çizge iki parçalıdır. **Yönlü döngü tespiti** üç renkli DFS'dir; gri düğüme giden ok bir geri kenardır.
- **Topolojik sıralama** yalnız DAG'lerde vardır. **Kahn algoritması** giriş derecesi 0 olan düğümleri kuyruktan sırayla çıkarır ve oklarını siler; sonuçta n'den az düğüm varsa döngü vardır. Maliyet $O(n + m)$'dir.
- **Ağırlıksız en kısa yolu** BFS bulur: ilk bulunan mesafe en kısadır. **Ağırlıklı en kısa yolu**, ağırlıklar negatif değilse **Dijkstra** bulur: mesafesi en küçük kesinleşmemiş düğümü seçer ve komşularını gevşetir. Heap ile $O((n + m) \log n)$, dizi taramasıyla $O(n^2)$'dir. Yol, `onceki` dizisinden hedeften geriye okunarak kurulur.
- **Minimum kapsayan ağaç**, bütün düğümleri en düşük toplam ağırlıkla bağlayan $n - 1$ kenarlı ağaçtır. **Kruskal** kenarları sıralayıp döngü kurmayanları Union-Find ile seçer; **Prim** tek ağacı her adımda en hafif dış kenarla büyütür. İkisi de $O(m \log n)$'dir ve kesim özelliğine dayanır.

Algoritma	Yardımcı yapı	Zaman (komşuluk listesi ile)	Ne verir?
BFS	kuyruk	O(n + m)	düzey sırası, ağırlıksız en kısa yol
DFS	yığın / özyineleme	O(n + m)	derinlik öncelikli gezinme, döngü, bileşenler
Topolojik sıralama (Kahn)	kuyruk / DFS	O(n + m)	DAG için bağımlılık sırası
Dijkstra	öncelikli kuyruk	O((n + m) log n)	ağırlıklı en kısa yol (ağırlık ≥ 0)
Kruskal	sıralama ve Union-Find	O(m log n)	minimum kapsayan ağaç
Prim	öncelikli kuyruk	O(m log n)	minimum kapsayan ağaç

Tablo 14.27: Bölümdeki algoritmalar.

Tablo, bölümün ortak dersini de özetler: veri yapısı algoritmanın davranışını belirler. Aynı gezinme iskeleti kuyrukla BFS, yığınla DFS, mesafeye göre sıralı öncelikli kuyrukla Dijkstra, kenar ağırlığına göre sıralı öncelikli kuyrukla Prim olur.

14.7 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

çizge *graph* · düğüm *vertex, node* · kenar *edge* · bitişik *adjacent* · derece *degree* · giriş derecesi *in-degree* · çıkış derecesi *out-degree* · yönlü çizge *directed graph (digraph)* · ağırlıklı çizge *weighted graph* · yol *path* · döngü *cycle* · bağlantılı bileşen *connected component* · tam çizge *complete graph* · kenar listesi *edge list* · komşuluk listesi *adjacency list* · komşuluk matrisi *adjacency matrix* · gezinme *traversal* · genişlik öncelikli arama *breadth-first search (BFS)* · derinlik öncelikli arama *depth-first search (DFS)* · iki parçalı çizge *bipartite graph* · yönlü döngüsüz çizge *directed acyclic graph (DAG)* · topolojik sıralama *topological sort* · gevşetme *relaxation* · öncelikli kuyruk *priority queue* · kapsayan ağaç *spanning tree* · minimum kapsayan ağaç *minimum spanning tree (MST)* · ayrık kümeler *disjoint sets (union-find)* · yol kısaltma *path compression* · Kruskal algoritması *Kruskal's algorithm* · Prim algoritması *Prim's algorithm*

14.8 Kendini Dene

1. 8 düğümlü basit, yönsüz ve bağlantılı bir çizgede en az ve en çok kaç kenar bulunabilir?

> **Cevap**
>
> En az $n - 1 = 7$ kenar (bu durumda çizge bir ağaçtır), en çok $n(n-1)/2 = 8 \cdot 7 / 2 = 28$ kenar (tam çizge K_8).

2. Okları 0 → 1, 0 → 2, 2 → 1, 3 → 0 olan yönlü çizgenin komşuluk matrisinde kaç hücre 1'dir? 0'ın çıkış derecesi ve 1'in giriş derecesi matristen nasıl okunur?

> **Cevap**
>
> Her ok bir hücre doldurur: 4 hücre 1'dir. 0'ın çıkış derecesi 0. satırın toplamıdır: `m[0][1] + m[0][2] = 2`. 1'in giriş derecesi 1. sütunun toplamıdır: `m[0][1] + m[2][1] = 2`.

3. Gezinme örneklerindeki dokuz düğümlü çizgede (komşular alfabetik) `bfs(E)` hangi sırayı yazdırır? A, E'den kaç kenar uzaktadır?

> **Cevap**
>
> Sıra `E B C G A D F I H`'dir. E'nin komşuları B, C, G düzey 1'dir; B'den A, C'den D, G'den F düzey 2'ye girer; A'dan I ve D'den H düzey 3'tür. A düzey 2'dedir: E'den 2 kenar uzaktadır (E → B → A).

4. Aynı çizgede özyinelemeli `dfs(F)` hangi sırayı yazdırır?

> **Cevap**
>
> `F A B C D G E H I`. F'nin ilk komşusu A, A'nın ilk ziyaret edilmemiş komşusu B, sonra C, D, G'ye inilir. G'nin komşusu E ziyaret edilir. E'den ve G'den geri dönülür; D'nin altında H ziyaret edilir. En sonda A'ya dönülünce A'nın son komşusu I ziyaret edilir.

5. Komşu listeleri 0: [1, 2] · 1: [4] · 2: [4] · 3: [1] · 4: [] olan DAG için `kahn()` hangi sırayı döndürür?

> **Cevap**
>
> Giriş dereceleri `[0, 2, 1, 0, 2]`'dir; kuyruk 0, 3 ile başlar. 0 çıkar: `giris[1] = 1`, `giris[2] = 0` ve 2 kuyruğa girer. 3 çıkar: `giris[1] = 0` ve 1 girer. 2 çıkar: `giris[4] = 1`. 1 çıkar: `giris[4] = 0` ve 4 girer. Sonuç `0 3 2 1 4`'tür; boyut 5 = n, döngü yok.

6. Okları 0 → 1, 1 → 2, 2 → 3, 3 → 1 olan çizgede `donguVar(0, renk)` hangi oku incelerken `true`

döndürür? O anda `renk` dizisi nedir?

Cevap

0, 1, 2, 3 sırayla gri olur: `renk = [1, 1, 1, 1]`. 3 → 1 oku incelenirken 1 gridir, yani şu anki yolun üzerindedir. Bu bir geri kenardır; bulunan döngü 1 → 2 → 3 → 1'dir.

7. Yönsüz kenarları A–B 1, A–C 4, B–C 2, B–D 6, C–D 3 olan çizgede Java Dijkstra A'dan çalıştırılırsa son `d` ve `onceki` değerleri ne olur? Kuyruktan kaç bayat kayıt çıkar?

Cevap

`d = [0, 1, 3, 6]` (A, B, C, D) ve `onceki = [-, A, B, C]`. A işlenince C 4, B 1 olur. B işlenince C 3'e iner (`{C, 3}` eklenir) ve D 7 olur. C işlenince D 6'ya iner (`{D, 6}` eklenir). Eski `{C, 4}` ve `{D, 7}` kayıtları bayat olarak çıkar ve atlanır: 2 bayat kayıt.

8. Kenarları A–B 3, A–C 1, B–C 3, B–D 2, C–D 4, C–E 5, D–E 6 olan çizgede Kruskal'ın seçtiği kenarlar ve toplam ağırlık nedir? Eşit ağırlıklı A–B ile B–C'nin deneme sırası değişirse ne olur?

Cevap

Sıra A–C 1 (kabul), B–D 2 (kabul), A–B 3 (kabul: {A, C} ile {B, D} birleşir), B–C 3 (ret), C–D 4 (ret), C–E 5 (kabul). Dört kenar seçilmiştir, D–E'ye bakılmaz. Toplam *1 + 2 + 3 + 5 = 11*'dir. B–C önce denenirse B–C kabul, A–B ret olur. Ağaç farklıdır (A–C, B–D, B–C, C–E), toplam yine 11'dir.

9. Bir önceki sorudaki çizgede tembel Prim A'dan başlatılırsa (eşitlikte alfabetik sıra) kenarlar hangi sırayla seçilir ve kaç bayat kayıt atlanır?

Cevap

A–C 1, A–B 3, B–D 2, C–E 5. A–B seçildikten sonra kuyrukta C–B 3 kalır; B ağaçta olduğu için bu kayıt bayattır. D girince C–D 4 de bayatlar. C–E'den önce bu iki kayıt çıkar ve atlanır: 2 bayat kayıt. Toplam yine 11'dir.

10. Bütün kenarlarının ağırlığı 5 olan bir çizgede tek kaynaklı en kısa yollar için Dijkstra yerine hangi algoritma kullanılabilir? Neden?

Cevap

BFS. Bütün ağırlıklar eşit olduğunda bir yolun uzunluğu *5 × kenar sayısı*'dır; en az kenarlı yol aynı zamanda en kısa yoldur. BFS'nin bulduğu mesafeler 5 ile çarpılır. BFS *O(n + m)*'dir; heap'li Dijkstra'nın *log n* çarpanından kurtulunur.

14.9 İleri okuma

- M. T. Goodrich, R. Tamassia ve M. H. Goldwasser, *Data Structures and Algorithms in Java* (6. baskı, 2014), Bölüm 14: çizge ADT'sini, gösterimleri, DFS ve BFS'yi, topolojik sıralamayı, Dijkstra'yı, Kruskal ve Prim'i Java koduyla anlatır; havalimanı örneği bu bölümdendir.
- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Bölüm 4: yönsüz ve yönlü çizgeleri, MKA'yı ve en kısa yolları ayrı kesimlerde işler; tembel ve istekli Prim sürümlerini karşılaştırır.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest ve C. Stein, *Introduction to Algorithms* (4. baskı, 2022), Bölüm 20–22: BFS ve DFS'nin doğruluk kanıtlarını, kesim özelliğini ve Bellman-Ford algoritmasını biçimsel bir dille verir; ayrık kümeler yapısının analizi Bölüm 19'dadır.

14.10 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022.
2. E. W. Dijkstra, “A note on two problems in connexion with graphs”, *Numerische Mathematik*, 1, 269–271, 1959.
3. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6. baskı, Wiley, 2014.
4. A. B. Kahn, “Topological sorting of large networks”, *Communications of the ACM*, 5(11), 558–562, 1962.
5. J. B. Kruskal, “On the shortest spanning subtree of a graph and the traveling salesman problem”, *Proceedings of the American Mathematical Society*, 7(1), 48–50, 1956.
6. R. C. Prim, “Shortest connection networks and some generalizations”, *Bell System Technical Journal*, 36(6), 1389–1401, 1957.
7. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011.

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

accumulator → biriktirici (kuyruk özyinelemesinde ara sonucu taşıyan parametre) 4

adaptable priority queue → uyarlanabilir öncelikli kuyruk 11

adjacency list → komşuluk listesi 14

adjacency matrix → komşuluk matrisi 14

adjacent → bitişik 14

aging → yaşlandırma 11

algorithm → algoritma 1

algorithm analysis → algoritma analizi 2

aliasing → takma ad 1

amortized analysis → amortize analiz 6

ancestor → ata 9

array → dizi 3

array covariance → dizi kovaryansı 3

ASCII (American Standard Code for Information Interchange) → ASCII 1

asymptotic → asimptotik 2

autoboxing → otomatik kutulama 6

average case → ortalama durum 2

AVL tree → AVL ağacı 10

B

B+ tree → B+ ağacı 10

B-tree → B-ağacı 10

back edge → geri kenar 14

backtracking → geri izleme 4

balance factor → denge faktörü 10

balanced parentheses → parantez kontrolü 7

base case → taban durum 4

best case → en iyi durum 2

Big O → büyük O 2

Big Omega → büyük Omega 2

Big Theta → büyük Teta 2

big-O notation → büyük O gösterimi 3

binary heap → ikili heap 11

binary recursion → ikili özyineleme 4

binary search → ikili arama 2

binary search tree (BST) → ikili arama ağacı 10

binary tree → ikili ağaç 9

binomial heap → binom heap 11

bipartite graph → iki parçalı çizge 14

bit (binary digit) → bit 1

bit mask → maske 13

bit vector → bit vektörü 13

black height → siyah yüksekliği 10

bottom-up heap construction → aşağıdan yukarıya heap oluşturma 11

bounded stack → sınırlı yığın 7

breadth-first search (BFS) → genişlik öncelikli arama 14

brute force → kaba kuvvet 7

BST property → BST özelliği 10

bubble sort → kabarcık sıralaması 3

bucket → kova 12

buffer → tampon 8

byte → bayt 1

C

cache → önbellek 3

cache miss → önbellek ıskası 6

call stack → çağrı yığını 7

call tree → çağrı ağacı 4

capacity → kapasite 3

chain → zincir 12

character encoding → karakter kodlaması 1

child → çocuk 9

circular queue → dairesel kuyruk 8

circularly linked list → dairesel bağlı liste 5

client code → istemci kodu 1

code point → kod noktası 1

collection → koleksiyon 6

collision → çakışma 12

comparator → karşılaştırıcı 6

compile time → derleme zamanı 1

complete binary tree → tam ikili ağaç 9
complete graph → tam çizge 14
complexity → karmaşıklık 2
compression function → sıkıştırma fonksiyonu 12
connected → bağlantılı 14
connected component → bağlantılı bileşen 14
contiguous → ardışık 3
contract (equals/hashCode) → sözleşme 12
correctness → doğruluk 2
covariance → kovaryans 3
covariant → kovaryant 3
cursor → imleç 6
cycle → döngü (çizgede: aynı düğümde başlayıp biten yol (bkz. loop)) 14
cycle detection → döngü tespiti 5

D

d-ary heap → d'li heap 11
data structure → veri yapısı 1
data type → veri tipi 1
decision tree → karar ağacı 4
deep copy → derin kopya 3
degenerate tree → devrik ağaç 10
degree → derece 9
dense graph → yoğun çizge 14
depth → derinlik 9
depth-first search (DFS) → derinlik öncelikli arama 14
deque (double-ended queue) → çift uçlu kuyruk 8
dequeue → dequeue (önden çıkarma) 8
descendant → soy 9
deterministic → belirleyici 12
difference → fark 13
digital data → dijital veri 1
Dijkstra's algorithm → Dijkstra algoritması 14
direct addressing → doğrudan adresleme 12
directed acyclic graph (DAG) → yönlü döngüsüz çizge 14
directed cycle → yönlü döngü 14
directed graph (digraph) → yönlü çizge 14
disjoint sets (union-find) → ayrık kümeler 13
double hashing → çift hash 12
doubling experiment → katlama deneyi 2
doubly linked list → çift yönlü bağlı liste 5
down-heap → down-heap (aşağı süzme) 11
dynamic array → dinamik dizi 3
dynamic size stack → dinamik boyutlu yığın 7

E

edge → kenar 9
edge list → kenar listesi 14
element → öğe 3
empirical analysis → deneysel analiz 2
enqueue → enqueue (arkaya ekleme) 8
Euler tour → Euler turu 9
exception → istisna 7
expression tree → ifade ağacı 9
external node → dış düğüm 9

F

fail-fast → hızlı başarısız 6
Fibonacci heap → Fibonacci heap 11
FIFO → ilk giren ilk çıkar 8
first-in, last-out (FILO) → ilk giren son çıkar 7
fixed size stack → sabit boyutlu yığın 7
floating point → kayan nokta 1
Floyd's cycle detection algorithm → Floyd'un döngü tespit algoritması 5
forest → orman 9
front → ön 8
full binary tree → dolu ikili ağaç 9

G

garbage collector → çöp toplayıcı 1
generic → jenerik 1
generics → jenerikler 1
graph → çizge 14

H

half-open interval → yarı açık aralık 6
hash code → hash kodu 12
hash function → hash fonksiyonu 12
hash set → hash kümesi 13
hash table → hash tablosu 12
head → baş 5
heap → heap 11
heap memory → heap bellek 3
heap sort → heap sort 11
heap-order property → heap-sıra özelliği 11
height → yükseklik 9
home slot → ev yuvası 12
Horner's method → Horner yöntemi 12

I

immutability → değiştirilemezlik 1
immutable → değiştirilemez 6
implementation → gerçekleme 1
in-degree → giriş derecesi 14
in-place → yerinde 3
incident → bağlı (kenar ile düğüm) 14
index → indis 3
infix → ara ek 7
initializer → ilklendirici 3
inorder → kök ortada 9
insertion sort → eklemeli sıralama 3

interface → arayüz 1
internal node → iç düğüm 9
intersection → kesişim 13
invariant → değişmez 2
inverse Ackermann function → ters Ackermann fonksiyonu 13
inversion → ters sıralı çift 3
isolated vertex → yalnız düğüm 14
iterator → yineleyici 6

J

jagged array → düzensiz dizi 3
Java Collections Framework → Java Koleksiyon Çerçevesi 6

K

Kahn's algorithm → Kahn algoritması 14
key → anahtar 11
key-value pair → anahtar-değer çifti 12
Kruskal's algorithm → Kruskal algoritması 14

L

lambda expression → lambda ifadesi 6
leaf → yaprak 9
level → düzey 9
level-order → düzey sırası 9
LIFO → son giren ilk çıkar 7
linear probing → doğrusal yoklama 12
linear recursion → doğrusal özyineleme 4
linear search → doğrusal arama 3
linked list → bağlı liste 5
list → liste 6
listener → olay dinleyicisi 6
load factor → yük faktörü 12
location-aware entry → konum farkında öğe 11
loitering → oyalanma 7
loop → döngü (programda: for/while tekrar yapısı (bkz. cycle)) 2
loop invariant → döngü değişmezi 2
lower bound → alt sınır 2

M

MAD method (Multiply-Add-and-Divide) → MAD yöntemi (çarp-topla-böl) 12
map → eşleme 12
mathematical induction → tümevarım 4
matrix → matris 3
max-heap → max-heap 11
median → ortanca 3
membership test → üyelik testi 13
memoization → bellekleme 4
memory leak → bellek sızıntısı 3
merge sort → birleştirme sıralaması 3
min-heap → min-heap 11
minimum spanning tree (MST) → minimum kapsayan ağaç 14
mode → mod 3
monotonic stack → monoton yığın 7
multidimensional array → çok boyutlu dizi 3
multimap → çoklu eşleme 13
multiple recursion → çoklu özyineleme 4
multiplicity → çokluk 13
multiset → çoklu küme 13
mutable → değiştirilebilir 6

N

next greater element → sonraki büyük öğe 7
node → düğüm (bağlı liste, ağaç ve çizgede; çizgede vertex ile aynı kavram) 5
non-primitive → ilkel olmayan 1
null → null 5

O

off-by-one error → sınır hatası 3
open addressing → açık adresleme 12
operand → işlenen 7
operator → işleç 7
out-degree → çıkış derecesi 14
overflow → taşma 7
overload → aşırı yükleme 6

P

parent → ebeveyn 9
pass by value → değerle geçirme 1
path → yol 9
path compression → yol kısaltma 13
perfect binary tree → mükemmel ikili ağaç 9
perfect hash → mükemmel hash 12
persistent data structure → kalıcı veri yapısı 6
pointer → işaretçi 5
pop → pop (üstten çıkarma) 7
position → konum 6
positional list → konumsal liste 6
postfix → sonek 7
postorder → kök sonda 9
predecessor → öncül 10
prefix → önek 7
prefix averages → önek ortalamaları 2
preorder → kök önde 9
Prim's algorithm → Prim algoritması 14
primary clustering → birincil kümelenme 12
primitive operation → ilkel işlem 2
primitive type → ilkel tip 1
priority queue → öncelikli kuyruk 11

probe sequence → yoklama dizisi 12
probing → yoklama 12
pruning → budama 10
push → push (üste ekleme) 7

Q

quadratic probing → karesel yoklama 12
queue → kuyruk 8
quick sort → hızlı sıralama 3
Quick-Find → Quick-Find 13

R

race condition → yarış durumu 6
random access → rastgele erişim 3
range query → aralık sorgusu 10
rank → rütbe 13
rear → arka 8
recurrence relation → yineleme bağıntısı 4
recursion → özyineleme 4
recursive step → özyinelemeli adım 4
red-black tree → kırmızı-siyah ağaç 10
reference → referans 1
relaxation → gevşetme 14
resizing (rehashing) → yeniden boyutlandırma 12
return address → dönüş adresi 4
reverse Polish notation (RPN) → ters Polonya gösterimi 7
ring buffer → halka tampon 8
robustness → sağlamlık 2
root → kök 9
rotation → döndürme 10
round robin → round robin (sırayla dönen çizelgeleme) 8
run time → çalışma zamanı 1

S

scheduler → çizelgeleyici 8
search tree → arama ağacı 10
secondary clustering → ikincil kümelenme 12
selection sort → seçimli sıralama 3
self-balancing search tree → kendini dengeleyen arama ağacı 10
self-loop → öz döngü 14
self-referential structure → öz-referanslı yapı 5
sentinel → bekçi 5
separate chaining → ayrı zincirleme 12
set → küme 13
shallow copy → sığ kopya 3
short-circuit evaluation → kısa devre 3
shortest path → en kısa yol 14
sibling → kardeş 9
simple path → basit yol 9
simple uniform hashing → basit düzgün hash 12
single-source shortest path → tek kaynaklı en kısa yol 14
singly linked list → tek yönlü bağlı liste 5
skewed tree → çarpık ağaç 9
sliding window → kayan pencere 8
slot → yuva 12
slow and fast pointers → yavaş ve hızlı işaretçi 5
sorted list → sıralı liste 11
space complexity → alan karmaşıklığı 2
spanning tree → kapsayan ağaç 14
sparse graph → seyrek çizge 14
splay tree → splay ağacı 10
stable (sort) → kararlı 3
stack → yığın 7
stack frame → yığın çerçevesi (çağrı yığınında bir metot çağrısının kaydı) 4
stack trace → yığın izi 4
stale entry → bayat kayıt 14
starvation → açlık 11
static array → statik dizi 3
stock span → stock span 7
stream → akış 6
structural modification → yapısal değişiklik 6
subset → alt küme 13
subtree → alt ağaç 9
successor → ardıl 10
swap → takas 3

T

tail → son 5
tail call → kuyruk çağrısı 4
tail call optimization → kuyruk çağrısı eniyilemesi 4
tail pointer → son işaretçisi 5
tail recursion → kuyruk özyinelemesi 4
tentative distance → geçici mesafe 14
thread → iş parçacığı 6
tight bound → sıkı sınır 2
time complexity → zaman karmaşıklığı 2
time slice (quantum) → zaman dilimi 8
TimSort → TimSort 6
tombstone (DEFUNCT) → silindi işareti 12
top → üst 7
topological sort → topolojik sıralama 14
Towers of Hanoi → Hanoi kuleleri 4
trade-off → ödünleşim 1
transpose → transpoz 3
traversal → gezinme 5
tree → ağaç 9
two-dimensional array → iki boyutlu dizi 3
type parameter → tür parametresi 1
type safety → tip güvenliği 1

U

unboxing → kutudan çıkarma 6

underflow → alt taşma 7

undirected graph → yönsüz çizge 14

undo → geri al 7

Unicode → Unicode 1

union → birleşim 13

union by rank → rütbeye göre birleştirme 13

union by size → boyuta göre birleştirme 13

universe → evren 13

unsorted list → sırasız liste 11

unwinding → çözülme 4

up-heap → up-heap (yukarı süzme) 11

upper bound → üst sınır 2

V

value → değer 12

varargs → değişken sayıda argüman 3

vertex → düğüm (çizgede; node ile aynı karşılık (bkz. node)) 14

view → görünüm 6

W

weight → ağırlık 14

weighted graph → ağırlıklı çizge 14

worst case → en kötü durum 2

wrapper class → sarmalayıcı sınıf 6

Z

zig, zig-zig, zig-zag → zig, zig-zig, zig-zag 10

EK B

Veri Yapıları İçin Ön Bilgiler

Veri yapıları; programlamayı, soyutlamayı, algoritmaları ve matematiksel akıl yürütmeyi bir araya getirir. Java, referanslar, diziler, özyineleme, karmaşıklık ve temel ayrık matematik bilgisi derse sağlam bir zemin hazırlar.

Temel birikim

En yararlı hazırlık, uygulamalı programlama deneyimidir. Bunun yanında kodu adım adım izleyebilmek, referansların davranışını kavramak ve işlem maliyetlerini karşılaştırabilmek gerekir.

Önerilen hazırlık sırası

Programlama **Java nesneleri** **Referanslar** **Diziler** **Özyineleme**

Ayrık matematik **Büyük O** **İzleme ve hata ayıklama**

Temel ön bilgiler

Programlama Temelleri

Değişkenler · kontrol akışı · metotlar

Gerçek bir veri yapısını kodlamadan önce temel programlamada rahat olmak gerekir.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Bir veri yapısını yalnız sakladığı değerler tanımlamaz; sunduğu işlemler ve kodun koruduğu değişmezler (*invariant*) de tanımlar.

- Atama, koşul, döngü ve metot çağrılarını adım adım izleyebilmek.
- Parametre alan ve değer döndüren metotlar yazabilmek.
- Yerel değişkenleri ve kapsamı (*scope*) anlamak.
- İç içe döngüleri okuyup kaç kez çalışacaklarını kestirebilmek.

Veri yapılarıyla bağlantısı: Gezinme, ekleme, silme, arama ve güncelleme işlemleri.

Değişkenler **Döngüler** **Metotlar**

Java Sınıfları ve Nesneleri

Nesneler · sınıflar · yapıcılar

Derste gerçeklemeler Java ile yazılır. Bu yüzden sınıf, nesne, alan, yapıcı (*constructor*) ve metot kavramları önceden tanıdık olmalıdır.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Veri yapılarının çoğu sınıf olarak gerçeklenir; iç gösterim, işlemlerin arkasında gizli kalır.

- Sınıf ile nesneyi ayırt edebilmek.
- new ile nesne oluşturmak ve yapıcıları kullanmak.
- Örnek alanlarını, yerel değişkenleri ve this anahtar sözcüğünü anlamak.
- Kapsüllemeyi, erişim belirleyicilerini ve static üyelerle örnek üyeleri arasındaki farkı tanımak.
- İstisna (*exception*) fırlatıp yakalayabilmek; örneğin boş bir yapıdan öğe çıkarılmak istendiğinde.
- Arayüzleri ve List<Integer> gibi jenerik tipleri okuyabilmek; Bölüm 1 ikisini de yeniden ele alır.

Veri yapılarıyla bağlantısı: Düğüm sınıfları, kap sınıfları, ADT'ler ve kapsüllenmiş değişmezler.

Java **Sınıflar** **Nesneler**

Referanslar ve Takma Ad

Referanslar · null · takma ad

Bağlı yapılar, nesneleri çalışma anında birbirine bağlayan referanslara dayanır.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Bağlı listeler, ağaçlar, çizgeler ve nesne tabanlı yapılar için referansları anlamak şarttır.

- Nesne değişkeninin nesnenin kopyasını değil, nesneye giden referansı tuttuğunu bilmek.
- İki değişkenin aynı nesneyi gösterebileceğini fark etmek (takma ad, *aliasing*).
- null değerini güvenle kullanmak ve neyi temsil ettiğini bilmek.
- Nesneyi değiştirmek ile referansa yeni değer atamak arasındaki farkı görmek.

Veri yapılarıyla bağlantısı: Sonraki düğüm bağlantıları, çocuk referansları, çizge düğümleri ve yapısal güncellemeler.

Referanslar **null** **Takma ad**

Diziler

İndisli saklama · ardışık konumlar

Diziler; dinamik dizilerin, heap'lerin, hash tablolarının, komşuluk matrislerinin ve birçok iç gösterimin temelidir. Temel dizi kullanımının bilindiği varsayılır. Bölüm 3 dizileri ve dizi algoritmalarını ayrıntılı işler; ArrayList'in dinamik dizi olarak büyümesi Bölüm 6'dadır.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

İndisle sabit sürede erişim, diziyi veri yapısı tasarımının en önemli yapı taşlarından biri yapar.

- Sıfırdan başlayan indislemeyi doğru kullanmak.
- Dizi üzerinde gezinmek ve indisi saklanan değerden ayırt etmek.
- Tek ve iki boyutlu dizileri tanımlamak, doldurmak ve yazdırmak.
- Sınır dışı indislerden kaçınmak; ArrayIndexOutOfBoundsException hatasının ne zaman oluştuğunu bilmek.

Veri yapılarıyla bağlantısı: ArrayList, ikili heap'ler, hash tablosu kovaları ve komşuluk matrisleri.

Diziler **İndisleme** **Bellek**

Özyineleme

Taban durum · özyinelemeli durum · çağrı yığını

Ağaç ve çizge algoritmaları çoğunlukla özyinelemeli (*recursive*) yazılır. Özyineleme, ağaçlarda ve çizgelerde kullanılmadan önce Bölüm 4'te yeniden ele alınır.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Özyineleme, aynı problemi alt yapılar üzerinde çözerek hiyerarşik yapıları işlemenin doğal yoludur.

- Özyinelemeyi sonlandıran taban durumu belirlemek.
- Özyinelemeli çağrıları ve dönüşleri izlemek.
- Yığın derinliğini ve sonsuz özyinelemeyi anlamak.
- Ağaç gezinmesinde ve derinlik öncelikli aramada özyinelemeli kalıpları tanımak.

Veri yapılarıyla bağlantısı: Ağaç gezinmeleri, DFS, özyinelemeli ekleme ve silme, böl ve fethet yaklaşımı.

Özyineleme **Ağaçlar** **Çağrı yığını**

Büyük O Gösterimi

Büyüme hızları · üst sınırlar · ölçeklenebilirlik

Veri yapıları, girdi büyüdükçe işlemlerinin ne kadar maliyetli olduğuna göre karşılaştırılır. Gösterimi Bölüm 2 biçimsel olarak tanıtır; yaygın büyüme hızlarını önceden tanımak bu bölümü kolaylaştırır.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Büyük O, bir gösterimin neden ötekinden daha iyi ölçeklendiğini açıklamanın dilidir.

- $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$ ve $O(n^2)$ sınıflarını tanımak.
- İç içe döngülerin maliyetini kestirmek.
- Art arda ikiye bölmenin neden logaritmik davranış verdiğini anlamak.
- Tek bir ölçüte bağlı kalmadan zaman ve alan ödünleşimlerini karşılaştırmak.

Veri yapılarıyla bağlantısı: Diziler, listeler, ağaçlar, heap'ler, hash tabloları ve çizgeler için işlem maliyeti tabloları.

Büyük O **Karmaşıklık** **Ölçeklenebilirlik**

Algoritma İzleme

Durum değişimleri · değişmezler · elle çalıştırma

Bir algoritmayı elle izleyebilmek, bir veri yapısını anlamanın en yararlı becerilerinden biridir.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

İzleme; işaretçilerin, indislerin, yığınların, kuyrukların ve ağaç bağlantılarının bir adımdan ötekine nasıl değiştiğini gösterir.

- Her işlemden sonra değişkenlerin ve yapının durumunu takip etmek.
- Döngüleri ve özyinelemeli çağrıları çalışma sırasıyla izlemek.
- Güncellemelerden sonra yapısal değişmezlerin hâlâ geçerli olup olmadığını denetlemek.
- Uç durumları küçük örnekler üzerinde düşünmek.

Veri yapılarıyla bağlantısı: Simülatör kullanımı, hata ayıklama, doğruluk denetimi ve sınav sorularındaki akıl yürütme.

İzleme **Değişmezler** **Elle çalıştırma**

Temel Ayrık Matematik

Kümeler · bağıntılar · logaritmalar

Biraz ayrık matematik bilgisi; kümeler, ağaçlar, çizgeler ve karmaşıklık üzerine akıl yürütmeyi kolaylaştırır.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Matematiksel gösterim; ilişkileri, sınırları, bağlantılılığı ve yapısal özellikleri kesin biçimde ifade etmeye yarar.

- Küme, üyelik, birleşim, kesişim ve fark kavramlarında rahat olmak.
- Basit bağıntıları ve çizgedeki gibi bağlantıları anlamak.
- Dengeli ağaçlar ve art arda ikiye bölme üzerine düşünürken üsleri ve logaritmaları kullanmak.
- Algoritma maliyetlerini karşılaştırırken basit toplamları ve eşitsizlikleri okuyabilmek.

Veri yapılarıyla bağlantısı: Kümeler, çizgede bağlantılılık, ağaç yüksekliği ve asimptotik analiz.

Kümeler **Bağıntılar** **Logaritmalar**

Tümevarım ve Kanıt Yöntemleri

Tümevarım · çelişki · bölünebilme

Ağaç yüksekliği sınırları, özyinelemeli algoritmaların doğruluğu ve hash tablolarındaki yoklama dizileri gibi birçok sonuç derste kısa kanıtlarla gösterilir. Bu kanıtlar çoğunlukla tümevarıma ya da çelişkiye dayanır.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Kanıt, bir yapının her girdide beklenen sınırı koruduğunu gösterir; birkaç örnek üzerinde denemek bu güvenceyi vermez.

- Tümevarımda taban adımını ve tümevarım adımını kurabilmek; tümevarım varsayımının nerede kullanıldığını görmek.
- Düğüm sayısı, yükseklik ya da rütbe gibi bir büyüklük üzerinden yapılan tümevarımı izleyebilmek.
- Tersini varsayıp bir çelişkiye ulaşarak kanıtlama yöntemini tanımak.
- Bölünebilme, asal sayı ve ortak bölen kavramlarını bilmek.

Veri yapılarıyla bağlantısı: Özyinelemenin doğruluğu, ikili ağaçta düğüm sayısı ile yükseklik arasındaki bağıntılar, kırmızı-siyah ağaç ve ayrık küme ağaçlarının yükseklik sınırları, karesel yoklamada ve çift hash'te denenen yuvaların farklı olması.

Tümevarım **Çelişki** **Bölünebilme**

Hata Ayıklama ve Test

Assert · uç durumlar · ara durumları inceleme

Veri yapısı hataları çoğu zaman yanlış bağlantılardan, yanlış indislerden, sınır koşullarından ya da bozulan değişmezlerden kaynaklanır.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Sistemli hata ayıklama, çıktı doğru görünene kadar kodu deneme yanılmayla değiştirmekten daha güvenilirdir.

- Boş, tek öğeli ve sınırdaki durumları test etmek.
- Yalnız son çıktıya değil, ara durumlara da bakmak.
- Değişmezler için assert ifadeleri ya da açık denetimler kullanmak.
- Girdi hatalarını yapının mantık hatalarından ayırmak.

Veri yapılarıyla bağlantısı: Bağlı liste güncellemeleri, döndürmeler, heap düzeni, hash tablosunu yeniden boyutlandırma ve çizgede gezinme.

Hata ayıklama **Test** **Uç durumlar**

EK C

Veri Yapılarında Önemli İsimler

Bu dersteki birçok yapı ve algoritma hâlâ onu bulanların adını taşır: Dijkstra algoritması, AVL ağacı, Kruskal algoritması. Aşağıdaki tanıtımlar dersin bölüm sırasını izler ve her katkıyı işlendiği bölüme bağlar.

Dizilerden ve bağlı listelerden kümelere ve çizgelere

Amaç yaşam öyküsü anlatmak değil; her ismi somut bir yapıya, algoritmaya, değişmeze ya da performans fikrine bağlamaktır.

Öne çıkan katkılar

Niklaus Wirth

Program: veri üzerinde çalışan algoritma

Pascal dilini tasarladı ve Algorithms + Data Structures = Programs (Algoritmalar + Veri Yapıları = Programlar, 1976) kitabını yazdı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Kitabın adı bu dersin ana fikrini özetler: program, iyi seçilmiş bir veri yapısı üzerinde çalışan bir algoritmadır.

Derste: Bölüm 1 · Giriş

Pascal **Algoritmalar** **Veri yapıları**

Barbara Liskov

Soyut veri tipleri

Stephen Zilles ile birlikte soyut veri tipi (*abstract data type*, ADT) kavramını ortaya koydu (1974) ve bu kavramı CLU dilinin temeline yerleştirdi. 2008 Turing Ödülü'nü aldı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

ADT, işlemlerin ne yaptığını nasıl gerçeklendiklerinden ayırır. Derste yığın, kuyruk, liste, eşleme ve küme tam da bu yolla tanımlanır.

Derste: Bölüm 1 · Giriş

ADT **Arayüzler** **Kapsülleme**

Donald E. Knuth

Algoritma analizi

The Art of Computer Programming (Bilgisayar Programlama Sanatı) dizisini yazdı. Algoritmaların matematiksel analizini, O, Ω ve Θ gösterimleriyle birlikte, standart bir uygulama hâline getirdi.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Arama, sıralama, ağaçlar ve hash'leme (*hashing*) konularını sistemli biçimde ele aldı; gerçekleme teknikleri böylece bir mühendislik disiplinine dönüştü. Doğrusal yoklamada (*linear probing*) beklenen yoklama sayıları da onun sonuçlarıdır.

Derste: Bölüm 2 · Algoritma Analizi, Bölüm 12 · Eşlemeler ve Hash Tabloları

Büyük O **Analiz** **Hash'leme**

John von Neumann

Sıralama · Birleştirme sıralaması

Birleştirme sıralamasını (*merge sort*) 1945'te tanımladı. Bu algoritma, programı bellekte saklayan bir bilgisayar için yazılmış ilk algoritmalardan biridir.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Birleştirme sıralaması, böl ve fethet (*divide and conquer*) yaklaşımının ve kararlı (*stable*) bir O(n log n) sıralamanın standart örneğidir.

Derste: Bölüm 3 · Sıralama Algoritmaları

Birleştirme sıralaması **Böl ve fethet** **Kararlı sıralama**

C. A. R. Hoare

Sıralama · Hızlı sıralama

Hızlı sıralamayı (*quicksort*) 1959–1961 yıllarında geliştirdi. Daha sonra program doğruluğunu kanıtlamak için Hoare mantığını kurdu.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Hızlı sıralama; bölümleme (*partitioning*), özyineleme, yerinde düzenleme ve pivot seçiminin ortalama ve en kötü durum maliyetine etkisini bir arada gösterir.

Derste: Bölüm 3 · Sıralama Algoritmaları

Hızlı sıralama **Bölümleme** **Özyineleme**

Allen Newell, Cliff Shaw ve Herbert A. Simon

Bağlı listeler

Bağlı listeyi (*linked list*) 1955–1956'da geliştirdikleri Information Processing Language (IPL) dilinde ilk kez kullandılar.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Referanslarla birbirine bağlanan düğümler, çalışma anında büyüyüp küçülen yapıları mümkün kıldı. Bağlı listeler, bağlı yığınlar, kuyruklar ve ağaçlar bu fikrin üzerine kurulur.

Derste: Bölüm 5 · Bağlı Listeler

Bağlı liste **Düğümler** **Referanslar**

Robert W. Floyd

Döngü tespiti · Heap oluşturma

Kaplumbağa ve tavşan döngü tespiti algoritması ona atfedilir. 1964 tarihli Treesort 3 algoritması heap'i aşağıdan yukarıya kurar.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Farklı hızlarda ilerleyen iki işaretçi, döngüyü O(n) sürede ve O(1) ek bellekle bulur. Aşağıdan yukarıya heap oluşturma (*bottom-up heap construction*) O(n log n) yerine O(n) sürer.

Derste: Bölüm 5 · Bağlı Listeler, Bölüm 11 · Öncelikli Kuyruklar ve Heap

Döngü tespiti **İki işaretçi** **Heap oluşturma**

Joshua Bloch

Java Collections Framework

Java 1.2 (1998) ile gelen Java Collections Framework'ü (Java koleksiyonlar çerçevesi) tasarladı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

List, Set, Map ve Iterator arayüzleri ile ArrayList, LinkedList ve HashMap gibi sınıflar, tıpkı derste olduğu gibi ADT'yi onu gerçekleyen veri yapısından ayırır.

Derste: Bölüm 6 · Liste ADT ve Java Koleksiyonları

List **Iterator** **Java koleksiyonları**

Friedrich L. Bauer ve Klaus Samelson

Yığınlar

Aritmetik formülleri çevirmek ve hesaplamak için yığın (*stack*) ilkesini ortaya koydular (Kellerprinzip, 1955–1957).

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Son giren ilk çıkar (*LIFO*) ilkesiyle çalışan yığın; ifade hesaplama, derleyiciler ve metot çağrılarının yönetimi için standart araç oldu.

Derste: Bölüm 7 · Yığınlar

Yığın **LIFO** **İfadeler**

Jan Łukasiewicz

Önek gösterimi

1920'lerde Polonya gösterimini, yani önek (*prefix*) gösterimini ortaya koydu. Ters Polonya gösterimi, yani sonek (*postfix*) gösterimi bundan türetildi.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Önek ve sonek ifadeler parantez gerektirmez; tek bir yığınla tek geçişte hesaplanır.

Derste: Bölüm 7 · Yığınlar

Önek **Sonek** **Gösterim**

Edsger W. Dijkstra

Shunting-yard · En kısa yollar

Ara ek (*infix*) ifadeyi sonek ifadeye çeviren shunting-yard (manevra sahası) algoritmasını (1961) ve ağırlıklı çizgelerde en kısa yol algoritmasını (1959) geliştirdi.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Shunting-yard bir işleç yığını ve öncelik kurallarıyla çalışır. En kısa yol algoritması ise gevşetmeyi (*relaxation*) anahtarı en küçük öğeyi veren bir öncelikli kuyrukla birleştirir.

Derste: Bölüm 7 · Yığınlar, Bölüm 14 · Çizgeler

Ara ek → sonek **En kısa yol** **Öncelikli kuyruk**

Georgy Adelson-Velsky ve Evgenii Landis

AVL ağaçları

İlk kendini dengeleyen ikili arama ağacı olan AVL ağacını 1962'de ortaya koydular.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Denge faktörü (*balance factor*) ile tek ya da çift döndürmeler (*rotation*) yüksekliği O(log n) tutar. Böylece arama, ekleme ve silme logaritmik kalır.

Derste: Bölüm 10 · Arama Ağaçları

AVL **Denge faktörü** **Döndürmeler**

Leonidas J. Guibas ve Robert Sedgewick

Kırmızı-siyah ağaçlar

Kırmızı-siyah ağacı 1978'de ortaya koydular. Çalışmaları Rudolf Bayer'in simetrik ikili B-ağaçlarına (1972) dayanır.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Düğüm renkleri ve birkaç döndürme ağacı dengede tutar. Java'daki TreeMap ve TreeSet kırmızı-siyah ağaçla gerçeklenir.

Derste: Bölüm 10 · Arama Ağaçları

Kırmızı-siyah ağaç **Dengeleme** **TreeMap**

Daniel D. Sleator ve Robert E. Tarjan

Splay ağaçları

Splay ağacını 1985'te ortaya koydular.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Splay ağacı denge bilgisi tutmaz. Erişilen her düğümü köke taşır; bu sayede işlemler amortize O(log n) sürer ve yakın zamanda kullanılan anahtarlara erişim hızlanır.

Derste: Bölüm 10 · Arama Ağaçları

Splay ağacı **Amortize analiz** **Yerellik**

Rudolf Bayer ve Edward M. McCreight

B-ağaçları

B-ağacını Boeing Scientific Research Labs'ta geliştirdiler (1970; yayımlanışı 1972).

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Düğüm başına çok sayıda çocuk (*fan-out*) ağacı sığ tutar; bu yüzden her aramada az sayıda disk bloğu okunur. B-ağaçları ve B+ ağaçları, veritabanı ve dosya sistemi indekslerinin temelidir.

Derste: Bölüm 10 · Arama Ağaçları

B-ağacı **B+ ağacı** **İndeksleme**

J. W. J. Williams

İkili heap · Heap sort

İkili heap'i (*binary heap*) ve heap sort algoritmasını 1964'te ortaya koydu.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Diziyle tutulan tam ikili ağaç, O(log n) sürede ekleme ve en küçük (ya da en büyük) öğeyi çıkarma sağlar. Aynı yapı, yerinde (*in-place*) çalışan O(n log n) bir sıralama da verir.

Derste: Bölüm 11 · Öncelikli Kuyruklar ve Heap

İkili heap **Heap sort** **Öncelikli kuyruk**

Michael L. Fredman ve Robert E. Tarjan

Fibonacci heap'leri

Fibonacci heap'i 1984'te ortaya koydular.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Anahtar küçültme (*decrease-key*) işleminin amortize O(1) maliyeti, özel bir heap'in en kısa yol gibi çizge algoritmalarını nasıl hızlandırabileceğini gösterir.

Derste: Bölüm 11 · Öncelikli Kuyruklar ve Heap

Fibonacci heap **Amortize** **Öncelikli kuyruk**

Hans Peter Luhn

Hash'leme

Zincirlemeli hash'leme fikrini 1953'te IBM içinde yazdığı bir notta önerdi.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Anahtarı doğrudan bir kovaya (*bucket*) eşlemek, aramayı beklenen O(1) süreye indirir. Ayrı zincirleme (*separate chaining*) bugün de çakışmaları çözmenin standart yollarından biridir.

Derste: Bölüm 12 · Eşlemeler ve Hash Tabloları

Hash'leme **Zincirleme** **Kovalar**

Bernard A. Galler ve Michael J. Fischer

Ayrık küme ormanları

Ayrık kümelerin orman gösterimini (*disjoint-set forest*) 1964'te ortaya koydular.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Her küme bir ağaçtır ve ağacın kökü kümenin temsilcisidir. Bu yüzden birleştirme (*union*) ve bulma (*find*) işlemleri için yalnız ebeveyn bağlantıları yeter.

Derste: Bölüm 13 · Kümeler ve Ayrık Kümeler

Ayrık kümeler **Birleştir-bul** **Ormanlar**

Robert E. Tarjan

Ayrık kümelerin analizi

Rütbeye göre birleştirme ile yol kısaltma birlikte kullanıldığında m işlemin O(m·α(n)) sürede bittiğini 1975'te kanıtladı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Ters Ackermann fonksiyonu α(n) o kadar yavaş büyür ki her işlem pratikte sabit süre alır.

Derste: Bölüm 13 · Kümeler ve Ayrık Kümeler

Rütbeye göre birleştirme **Yol kısaltma** **Amortize**

Edward F. Moore

Genişlik öncelikli arama

Genişlik öncelikli aramayı (*breadth-first search*, BFS) 1959'da bir labirentteki en kısa yolu bulma yöntemi olarak yayımladı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

BFS, bir kuyruk yardımıyla düğümleri düzey düzey gezer ve ağırlıksız çizgelerde en kısa yolları O(n + m) sürede bulur.

Derste: Bölüm 14 · Çizgeler

BFS **Kuyruk** **En kısa yol**

Arthur B. Kahn

Topolojik sıralama

Giriş derecesine (*in-degree*) dayanan topolojik sıralama algoritmasını 1962'de yayımladı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Giriş derecesi sıfır olan düğümleri art arda çıkarmak, yönlü bir çizgenin düğümlerini topolojik sıraya dizer; çizge yönlü döngüsüz değilse (*directed acyclic graph*, DAG) döngüyü ortaya çıkarır.

Derste: Bölüm 14 · Çizgeler

Topolojik sıralama **DAG** **Giriş derecesi**

Joseph B. Kruskal

Minimum kapsayan ağaçlar

Minimum kapsayan ağaç (*minimum spanning tree*, MST) algoritmasını 1956'da yayımladı.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Kruskal algoritması kenarları ağırlığa göre sıralar. Döngü oluşturacak kenarları ayrık kümeler (*union-find*) yapısıyla eler.

Derste: Bölüm 14 · Çizgeler

MST **Açgözlü** **Ayrık kümeler**

Robert C. Prim

Prim algoritması

Bell Laboratuvarları'nda çalışırken minimum kapsayan ağacı tek bir düğümden başlayıp büyüten algoritmayı “Shortest connection networks and some generalizations” (1957) makalesiyle yayımladı. Aynı yöntemi Vojtěch Jarník 1930'da, Edsger W. Dijkstra da 1959'da bağımsız olarak verdi.

MÜHENDİSLİK AÇISINDAN ÖNEMİ

Prim algoritması her adımda ağaçla ağacın dışındaki düğümler arasındaki en hafif kenarı ekler; kesim özelliği bu kenarın bir minimum kapsayan ağaçta bulunduğunu güvenceye alır. Aday kenarlar min-öncelikli kuyrukta tutulduğunda algoritma O(m log n) sürer.

Derste: Bölüm 14 · Prim Algoritması, Bölüm 14 · Kruskal ve Prim

MST **Kesim özelliği** **Öncelikli kuyruk**