Bölüm 10: Ağaçlar

C ile Veri Yapıları ve Algoritmalar

Terminoloji | İkili ağaç | Dizi / bağlı gösterim | Özyinelemeli gezinmeler | İteratif gezinmeler | Düzey düzey, kopyala, eşit mi?

Kaynak: Horowitz–Sahni–Anderson-Freed, Bölüm 5

GİRİŞ

Öğrenme Çıktıları

Terminolojiyi kullanmak

Kök, ebeveyn, çocuk, kardeş, ata, soy, derece, düzey, derinlik ve yaprak kavramlarını bir ağaç üzerinde göstermek.

İkili ağacı çözümlemek

Lemmayı (2^{i-1}, $2^k - 1$) kanıtlamak; tam dolu, tam ve tam olmayan ağaçları ayırmak.

İki gösterimi kurmak

Diziyle (ebeveyn = $\lfloor i/2 \rfloor$) ve bağlı yapıyla (`agac_gost`) gösterim; bellek maliyetini karşılaştırmak.

Gezinme

Kök önde / ortada / sonda gezinmeleri özyinelemeli ve yığınla döngüsel yazmak; düzey düzey gezmek.

Ağaç yordamları yazmak

Uç düğüm yazma, kopyalama, eşbiçimlilik ve sol–sağ değiş tokuşu özyinelemeli kurmak.

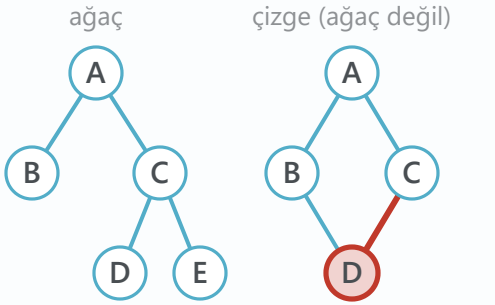
Maliyeti hesaplamak

Her yordamın zaman O(n) ve bellek O(h) maliyetini gerekçelendirmek.

TERMİNOLOJİ

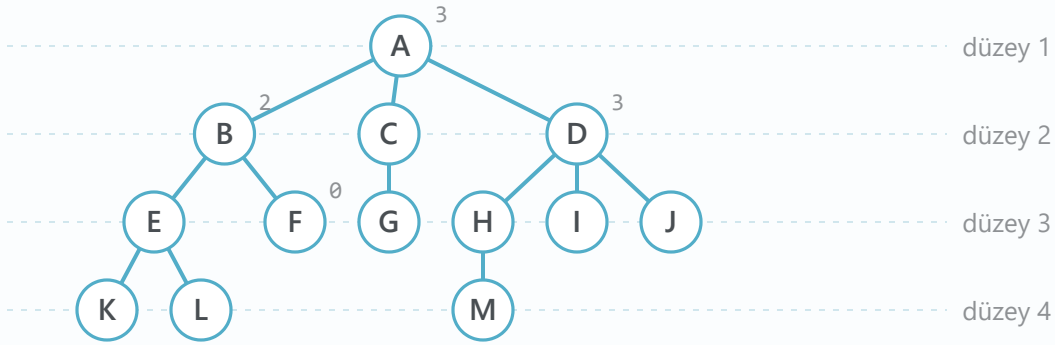
Ağaç (tree): tek ebeveynli hiyerarşi

- Arama, sıralama, veri sıkıştırma, söz dizim çözümleme (*syntax analysis*), kod optimizasyonu, derleyici ara süreçleri…
- Günlük hayatta: soy ağacı, bir kurumun yönetim şeması, dosya sistemi.
- **Ağaç, çizgenin (graph) bir alt kümesidir**: ağaçta her düğümün **tek bir ebeveyni** olur; çizgede birden çok olabilir.
- Her alt ağaç yine bir ağaçtır → ağaç yordamları doğal olarak **özyinelemelidir**.



D'nin iki ebeveyni var

Terminoloji: bir örnek ağaç üzerinde



Ebeveyn / çocuk / kardeş

B; E ve F'nin **ebeveyni**, E ve F B'nin **çocukları**. Aynı ebeveynin çocukları **kardeştir** (sibling).

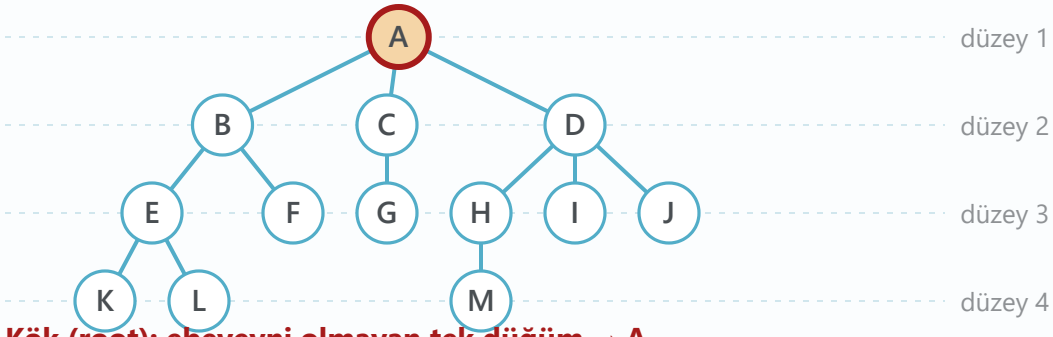
Derece (degree)

Bir düğümün alt ağaç sayısı: A'nın derecesi 3, F'nin 0 (sağ üstteki küçük sayılar).

Yaprak / uç düğüm

Derecesi 0 olan düğüm (leaf, terminal): K, L, F, G, M, I, J.

Terminoloji adım adım

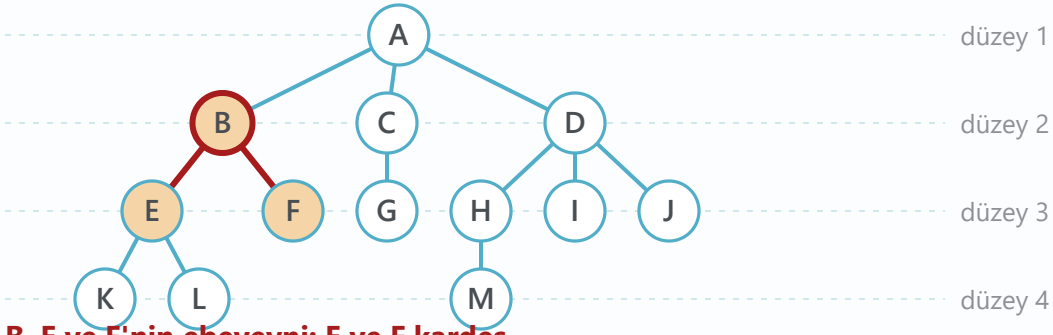


Kök (root): ebeveyni olmayan tek düğüm → A

Kök A'dır. Düzeyler kökten başlayarak numaralanır: bu derste **kök 1. düzeydedir**.

Derece ile derinlik ayrı kavramlardır; ikisini “derecelerin maksimumu” diye aynı tanımla vermek sık yapılan bir hatadır.

Terminoloji adım adım

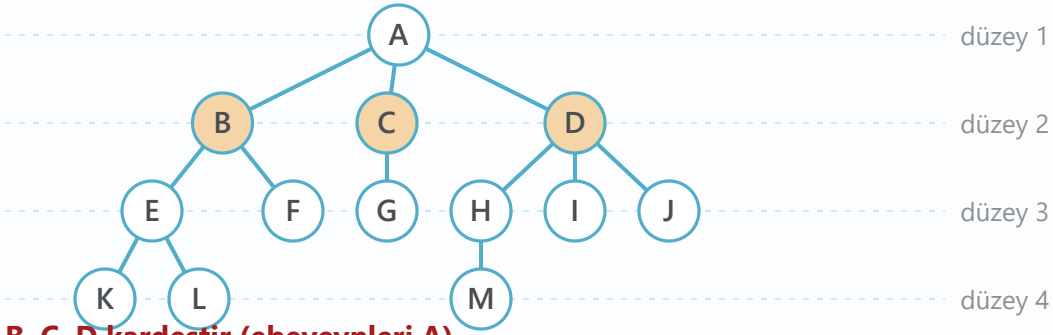


B, E ve F'nin ebeveyni; E ve F kardeş

Alt ağaçları olan düğüm, alt ağaçlarının köklerinin **ebeveynidir** (parent); onlar da onun **çocukları**dır (child).

Derece ile derinlik ayrı kavramlardır; ikisini “derecelerin maksimumu” diye aynı tanımla vermek sık yapılan bir hatadır.

Terminoloji adım adım

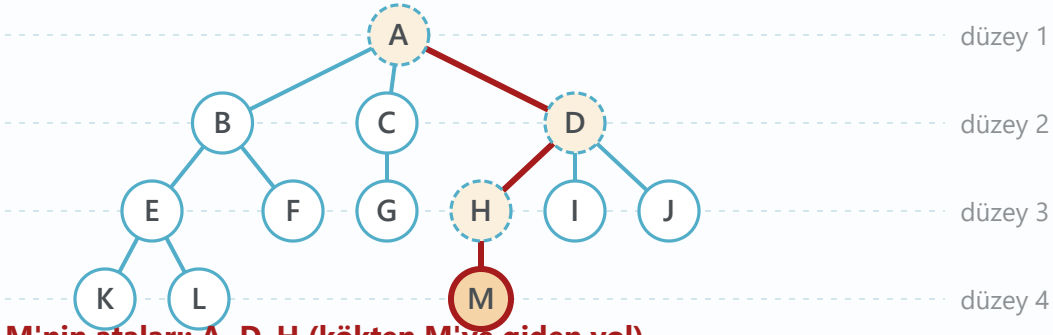


B, C, D kardeştir (ebeveynleri A)

Aynı ebeveynin çocukları **kardeş** düğümlerdir. Terminoloji büyükbaba–torun diye diğer kuşaklara genişler.

Derece ile derinlik ayrı kavramlardır; ikisini “derecelerin maksimumu” diye aynı tanımla vermek sık yapılan bir hatadır.

Terminoloji adım adım

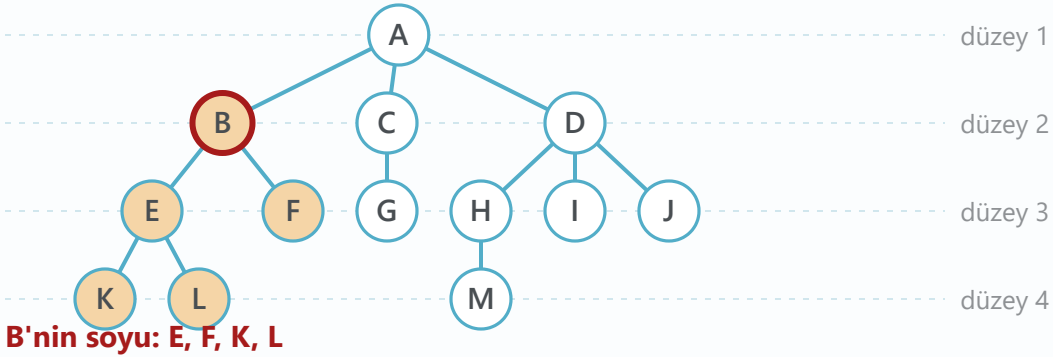


M'nin ataları: A, D, H (kökten M'ye giden yol)

Bir düğümün **ataları** (ancestor) kökten o düğüme kadar olan **yol** (path) üzerindeki tüm düğümlerdir. D, M'nin büyükbabasıdır.

Derece ile derinlik ayrı kavramlardır; ikisini “derecelerin maksimumu” diye aynı tanımla vermek sık yapılan bir hatadır.

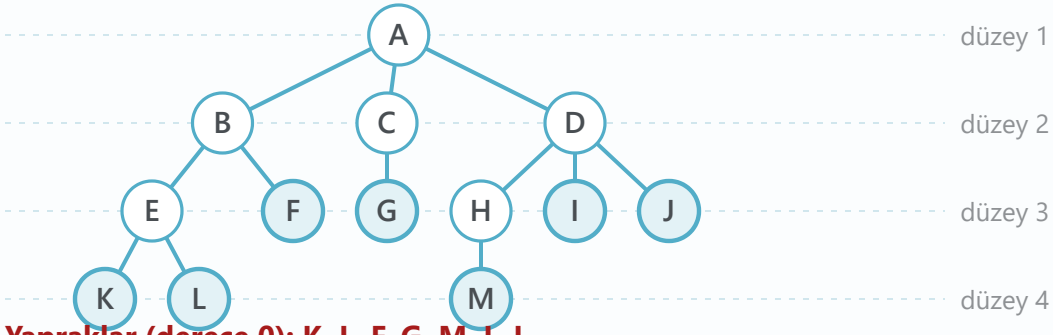
Terminoloji adım adım



Bir düğümün alt ağacındaki tüm düğümler onun **soyudur** (descendant). Çocukların çocukları torundur: E, F, G, H, I, J A'nın torunlarıdır.

Derece ile derinlik ayrı kavramlardır; ikisini “derecelerin maksimumu” diye aynı tanımla vermek sık yapılan bir hatadır.

Terminoloji adım adım

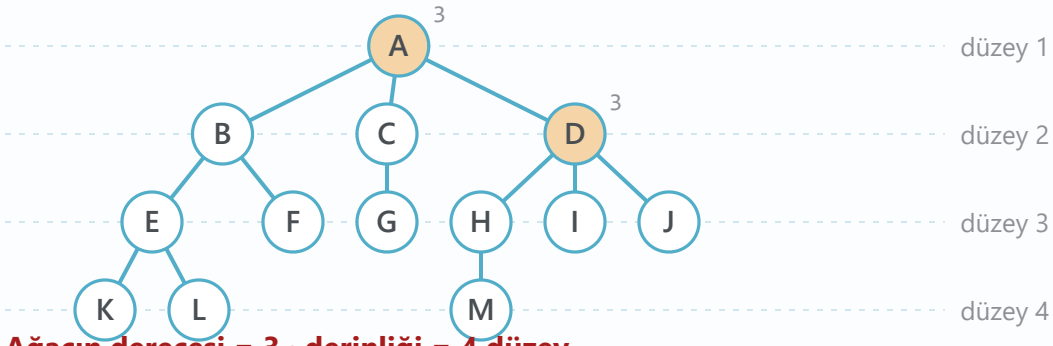


Yapraklar (derece 0): K, L, F, G, M, I, J

Derecesi 0 olan düğümlere **yaprak** ya da **uç düğüm** denir.

Derece ile derinlik ayrı kavramlardır; ikisini “derecelerin maksimumu” diye aynı tanımla vermek sık yapılan bir hatadır.

Terminoloji adım adım



Ağacın derecesi = 3 · derinliği = 4 düzey

Ağacın derecesi düğüm derecelerinin en büyüğü (3), **derinliği** ise en büyük düzey numarasıdır (4).

Derece ile derinlik ayrı kavramlardır; ikisini “derecelerin maksimumu” diye aynı tanımla vermek sık yapılan bir hatadır.

TERMİNOLOJİ

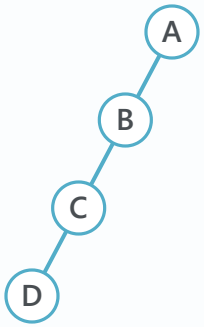
Derinlik, yükseklik: kitaplar farklı sayar

- Bu ders (Horowitz–Sahni): kök **1. düzey**; ağacın **derinliği** = en büyük düzey = düğüm sayısıyla ölçülen en uzun yol. Örnek ağaçta 4.
- CLRS / Sedgewick: kök **derinlik 0**; **yükseklik** = kökten en uzak yaprağa **kenar** sayısı. Aynı ağaçta 3.
- Sınavda hangi tanımın kullanıldığı belirtilmeli; formüller buna göre 1 kayar.
- Bu bölümde `derinlik(NULL) = 0`, tek düğümlü ağacın derinliği 1.

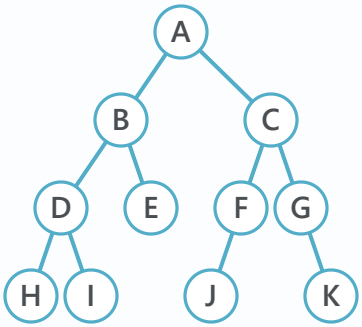
kavram	bu ders	CLRS
kökün düzeyi	1	0
M'nin düzeyi	4	3
ağacın derinliği	4	yükseklik 3
boş ağaç	0	−1

İkili ağaç (binary tree)

- Her düğümün **yalnızca sol alt ağacı** ve/veya **sağ alt ağacı** vardır ya da hiç çocuğu yoktur.
- Düğüm dereceleri 0, 1 ya da 2 olabilir.
- Sol ile sağ **ayrıdır**: tek çocuklu bir düğümde çocuğun solda mı sağda mı olduğu önemlidir.
- Kümeler: **ikili ağaç** ⊂ **ağaç** ⊂ **çizge**.



sola çarpık



4 düzeyli bir ikili ağaç

Lemma: bir düzeyde ve bir ağaçta en çok kaç düğüm?



düzey 1: 2^0 = 1

düzey 2: —

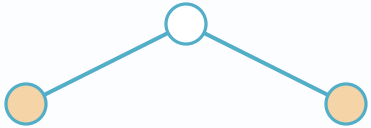
düzey 3: —

düzey 4: —

toplam = 2^1 − 1 = 1

i) i. düzeydeki en fazla düğüm sayısı $\mathbf{2^{i-1}}$ ($i \geq 1$). Taban: $i = 1$ için $2^0 = 1$ (yalnız kök).

Lemma: bir düzeyde ve bir ağaçta en çok kaç düğüm?



düzey 1: 2^0 = 1

düzey 2: 2^1 = 2

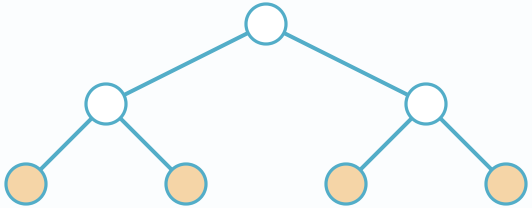
düzey 3: —

düzey 4: —

toplam = 2^2 − 1 = 3

Tümevarım: (n−1). düzeyde 2^{n-2} düğüm varsa, her birinin en çok 2 çocuğu olduğundan n. düzeyde $2 \cdot 2^{n-2} = 2^{n-1}$ düğüm olur.

Lemma: bir düzeyde ve bir ağaçta en çok kaç düğüm?



düzey 1: 2^0 = 1

düzey 2: 2^1 = 2

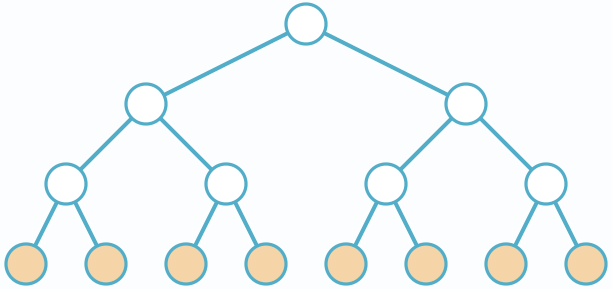
düzey 3: 2^2 = 4

düzey 4: —

toplam = 2^3 − 1 = 7

ii) k derinliğinde bir ikili ağaçta en fazla $\mathbf{2^k - 1}$ düğüm: $\Sigma_{i=1..k}\, 2^{i-1} = 2^0 + 2^1 + \ldots + 2^{k-1}$.

Lemma: bir düzeyde ve bir ağaçta en çok kaç düğüm?



düzey 1: 2^0 = 1

düzey 2: 2^1 = 2

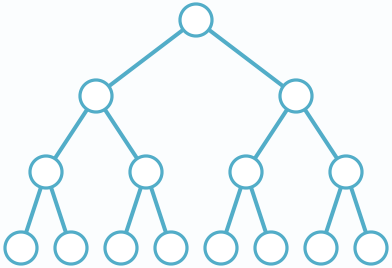
düzey 3: 2^2 = 4

düzey 4: 2^3 = 8

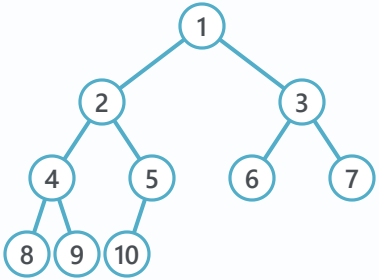
toplam = 2^4 − 1 = 15

Geometrik toplam $2^k - 1$ verir. k = 4 → en çok 15 düğüm. Lemma: “doğruluğu hemen görülebilen, teoremin yumuşatılmış hâli”.

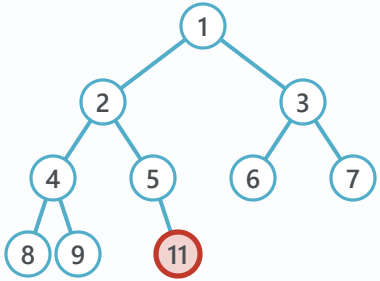
Tam dolu (full) ve tam (complete) ikili ağaç



tam dolu (full): 2^4 − 1 = 15



tam (complete): 1..10 kesintisiz



tam değil: 10 numaralı düğüm yok!

TANIM: Tam dolu

Derinliği k olan ve **$2^k - 1$** düğümlü ikili ağaç: tüm düzeyler tam dolu.

TANIM: Tam

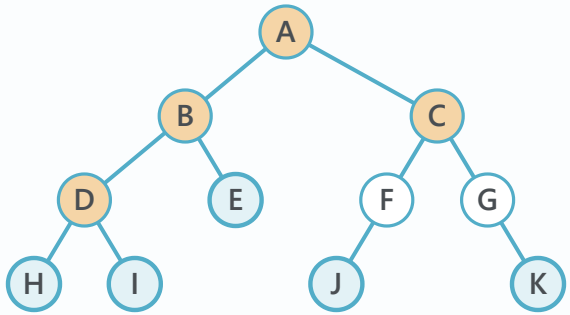
n düğümlü; ebeveyne i, sol çocuğa 2i, sağ çocuğa 2i+1 verildiğinde **1'den n'e atlamasız** numaralanabilen ağaç.

Neden önemli?

Tam ağaç diziye boşluksuz sığar ve derinliği ⌈lg(n+1)⌉ ile en küçüktür.

Derinlik ve düğüm sayısı ilişkileri

- k derinlikli ikili ağaçta: **$k \le n \le 2^k - 1$**; dolayısıyla **$\lceil \lg(n+1) \rceil \le k \le n$**.
- Tam (complete) ağaçta derinlik en küçüktür: $k = \lceil \lg(n+1) \rceil = \lfloor \lg n \rfloor + 1$.
- **$n_0 = n_2 + 1$**: yaprak sayısı, iki çocuklu düğüm sayısından bir fazladır. Kanıt: $n = n_0+n_1+n_2$ ve kenar sayısı $n-1 = n_1+2n_2$.
- 2n işaretçi alanından n−1'i dolu → **n+1 NULL işaretçi** (kılavuzlu ikili ağaçların çıkış noktası).



n = 11: n0 = 5 yaprak, n1 = 2, n2 = 4 → 5 = 4 + 1

Kaynak: Horowitz–Sahni–Anderson-Freed, Lemma 5.2–5.3; CLRS B.5.

İkili ağacın dizi ile gösterimi

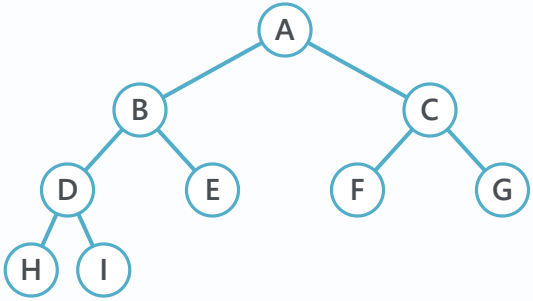
- n düğümlü **tam** ikili ağaç bir diziye yerleştirilebilir.
- **Formüller gereği 0 değil 1 numaralı indisten başlanır!**
- `baba(i) = ⌊i/2⌋` (i = 1 ise kök, ebeveyni yok)
- `solcocuk(i) = 2i` (2i > n ise sol çocuk yok)
- `sagcocuk(i) = 2i+1` (2i+1 > n ise sağ çocuk yok)
- Derinlik k ise $2^k - 1$ gözlük dizi açılmalıdır.

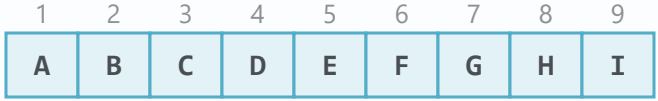
```c
#define MAX_DUGUM 63   /* 2^k - 1 */
char agac[MAX_DUGUM + 1];  /* [0] kullanılmaz */
int  n;                    /* düğüm sayısı */

int baba(int i)     { return i / 2; }
int solcocuk(int i) { return 2 * i; }
int sagcocuk(int i) { return 2 * i + 1; }
/* çocuk var mı?  solcocuk(i) <= n */
```

GÖSTERİM

Dizi gösteriminde bellek: tam ağaç ve çarpık ağaç





9 göz açıldı, 9'u da dolu: kayıp yok

baba(9) = ⌊9/2⌋ = 4 → D; sol(4) = 8 → H

Tam ağaç: dizi gözlerinin neredeyse tamamı dolar; bellek kaybı ya hiç olmaz ya da çok az olur.

Adım 1 / 2

HOCADAN "Depolama ve erişim, bilgisayar bilimlerinin iki ana konusudur. … Eğer ağacımız tam ikili ağaçsa ve ekleme/çıkarma yapılmayacaksa, ağaç yapısını dizi ile gerçeklemek uygun olur."

GÖSTERİM

Dizi gösteriminde bellek: tam ağaç ve çarpık ağaç



Ağaç tam olmaktan uzaklaştıkça âtıl bellek artar. Sola **çarpık** (skewed) 5 derinlikli ağaç 31 göz ister, 5'ini kullanır.

Adım 2 / 2

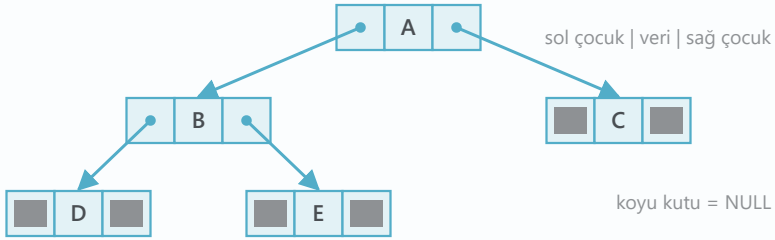
HOCADAN “Depolama ve erişim, bilgisayar bilimlerinin iki ana konusudur. … Eğer ağacımız tam ikili ağaçsa ve ekleme/çıkarma yapılmayacaksa, ağaç yapısını dizi ile gerçeklemek uygun olur.”

İkili ağacın bağlı (işaretçili) gösterimi

```
typedef struct agac *agac_gost;
struct agac {
    agac_gost solcocuk;   /* sol alt ağaç */
    char      veri;
    agac_gost sagcocuk;   /* sağ alt ağaç */
};
```

wide-left

- Her düğüm: sol işaretçi, veri, sağ işaretçi.
- Ekleme / silme yalnızca işaretçi atamasıdır.
- Boş alt ağaç = NULL (koyu kutu).



GÖSTERİM

Yeni düğüm: ogeal()

```
agac_gost ogeal(void) {          /* yeni düğüm */
    agac_gost p = malloc(sizeof *p);
    if (p == NULL) {
        fprintf(stderr, "Bellek yok!\n");
        exit(EXIT_FAILURE);
    }
    p->solcocuk = p->sagcocuk = NULL;
    return p;
}

void agac_sil(agac_gost L) {     /* kök sonda */
    if (!L) return;
    agac_sil(L->solcocuk);
    agac_sil(L->sagcocuk);
    free(L);
}
```

`ogeal()` burada `malloc` sonucu denetlenerek ve çocuklar NULL yapılarak verilmiştir; tüm örnekler bunu kullanır.

- Ağacı bırakırken **kök sonda** sıra zorunludur: kök önce `free` edilirse çocuklara giden işaretçiler kaybolur.

GÖSTERİM

Dizi mi, bağlı gösterim mi?

Dizi

+ İşaretçi alanı yok; ebeveyn/çocuğa O(1) indis hesabı.
+ Tam ağaçta bellek kaybı yok.
+ Örn. ağdaki bilgisayarlar sabitse, ileride eklenip çıkarılmayacaksa uygundur.

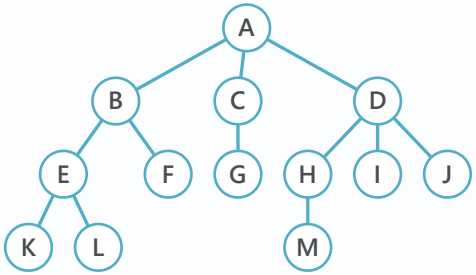
Bağlı gösterim

– Her düğüme 2 işaretçi; çarpık ağaçta bile n düğüm yeter.
– Ekleme / silme işaretçi atamasıyla yapılır.
– Ebeveyne doğrudan erişim yok (gerekirse `ebeveyn` alanı eklenir).

Kural: **ağaç tam ve sabit** ise dizi (ör. heap); **şekli değişen** ağaç için bağlı gösterim.

GÖSTERİM

Genel ağacı ikili ağaca çevirme: sol çocuk – sağ kardeş



Genel ağaçta düğüm başına çocuk sayısı değişkendir; sabit alanlı bir `struct` ile tutmak zordur.

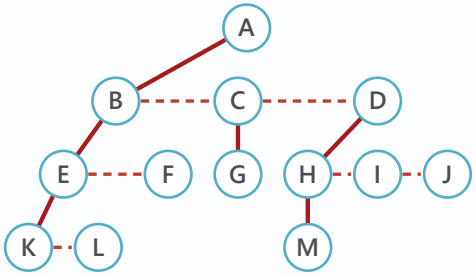
Adım 1 / 3

```
struct gdugum {
    char veri;
    struct gdugum *ilk_cocuk;   /* = sol */
    struct gdugum *kardes;      /* = sağ */
};
```

- İkili ağacın kök önde sırası = genel ağacın kök önde sırası.
- İkili ağacın kök ortada sırası = genel ağacın kök sonda sırası. (Horowitz 5.1.2; Knuth 2.3.2)

GÖSTERİM

Genel ağacı ikili ağaca çevirme: sol çocuk – sağ kardeş



turuncu: ilk çocuk (sol) · kesikli: kardeş (sağ)

Her düğüm için yalnızca **ilk çocuğa** bağ kalır; kardeşler kendi aralarında **sağ kardeş** bağıyla zincirlenir.

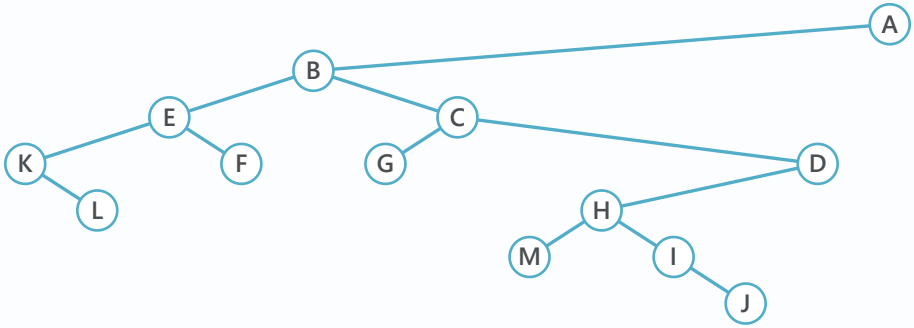
Adım 2 / 3

```
struct gdugum {
    char veri;
    struct gdugum *ilk_cocuk;   /* = sol */
    struct gdugum *kardes;      /* = sağ */
};
```

- İkili ağacın kök önde sırası = genel ağacın kök önde sırası.
- İkili ağacın kök ortada sırası = genel ağacın kök sonda sırası. (Horowitz 5.1.2; Knuth 2.3.2)

GÖSTERİM

Genel ağacı ikili ağaca çevirme: sol çocuk – sağ kardeş



İlk çocuk = **sol**, sağ kardeş = **sağ** işaretçi: sonuç bir ikili ağaçtır. Kökün sağ bağı hep NULL'dur.

Adım 3 / 3

```
struct gdugum {
    char veri;
    struct gdugum *ilk_cocuk;   /* = sol */
    struct gdugum *kardes;      /* = sağ */
};
```

- İkili ağacın kök önde sırası = genel ağacın kök önde sırası.
- İkili ağacın kök ortada sırası = genel ağacın kök sonda sırası. (Horowitz 5.1.2; Knuth 2.3.2)

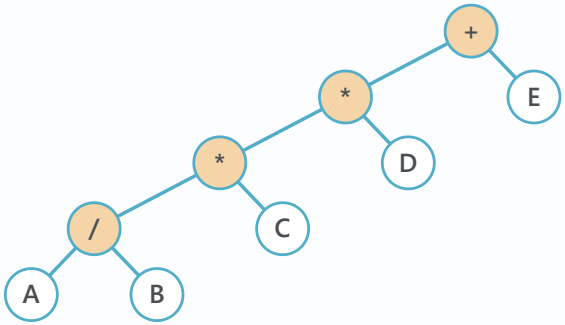
GEZİNMELER

İkili ağaç gezinmeleri (traversals)

- Amaç: her düğümü **yalnızca bir kez** ziyaret ederek tüm ağacı gezmek.
- Bir düğümde üç şey var: **veri** (V), **sol** alt ağaç (L), **sağ** alt ağaç (R) → 3! = **6** gezinme sırası.
- Solun sağdan önce gezildiği üçü standarttır: **inorder**, **preorder**, **postorder**.
- Uygulama isterse diğer üçü de (sağdan sola) kullanılabilir.

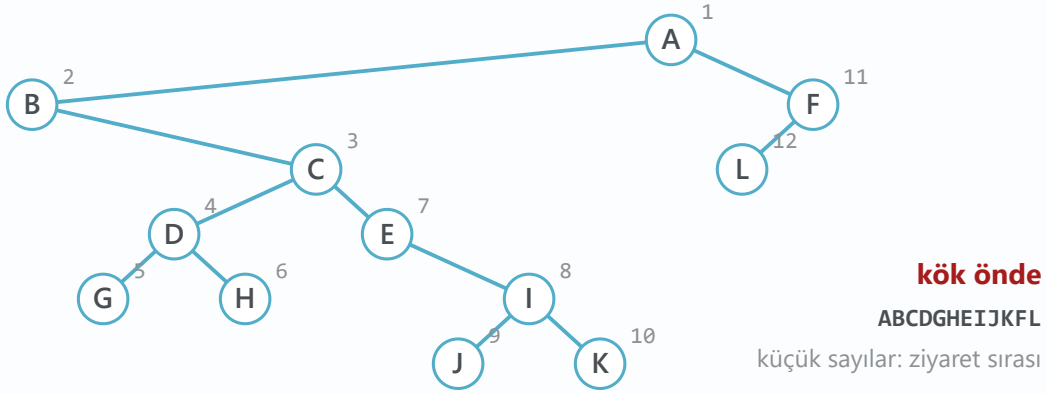
	sıra	ad
1	Sol – **Veri** – Sağ	kök ortada (inorder)
2	Sol – Sağ – **Veri**	kök sonda (postorder)
3	**Veri** – Sol – Sağ	kök önde (preorder)
4	Sağ – **Veri** – Sol	ters kök ortada
5	Sağ – Sol – **Veri**	ters kök sonda
6	**Veri** – Sağ – Sol	ters kök önde

Aritmetik ifade ağacı ve üç gösterim



- Kök ortada → ara-işleç (infix): `((((A/B)*C)*D)+E)`
- Kök önde → ön-işleç (prefix): `+**/ABCDE`
- Kök sonda → son-işleç (postfix): `AB/C*D*E+`
- İç düğümler işleç (turuncu), yapraklar işlenen. Parantezler ağacın **şeklinde** saklıdır; infix yazarken geri eklenir.
- Derleyiciler ifadeleri böyle bir ağaca çevirir; postfix, yığınla değerlendirilir.

Aynı ağaç, üç gezinme



kök önde

`ABCDGHEIJKFL`

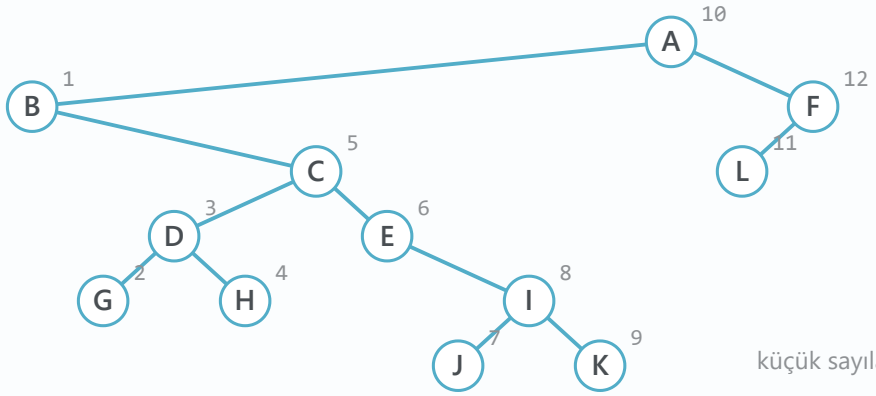
küçük sayılar: ziyaret sırası

Kök önde (V-L-R): önce kök, sonra sol alt ağacın tamamı, en son sağ alt ağaç →

`ABCDGHEIJKFL`

Adım 1 / 3

Aynı ağaç, üç gezinme



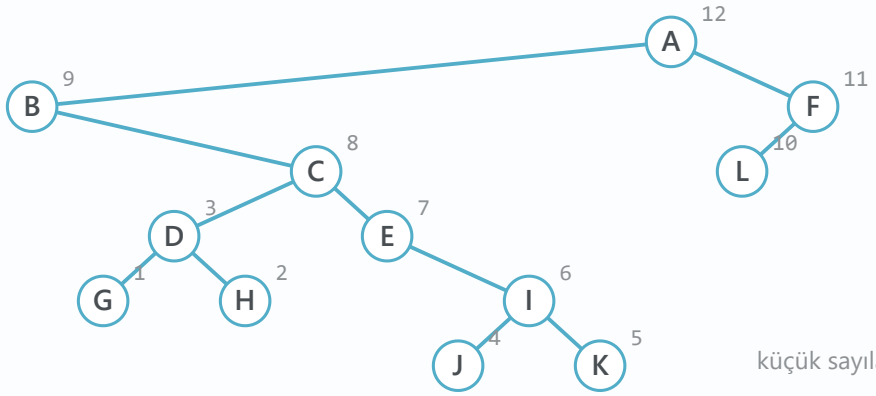
kök ortada

`BGDHCEJIKALF`

küçük sayılar: ziyaret sırası

Kök ortada (L-V-R): B'nin solu boş olduğundan ilk B yazılır → `BGDHCEJIKALF`

Aynı ağaç, üç gezinme



kök sonda

`GHDJKIECBLFA`

küçük sayılar: ziyaret sırası

Kök sonda (L-R-V): kök hep en sonda; A son harf → `GHDJKIECBLFA`

Gezinme yordamları — özyinelemeli

```c
void kok_onde_dolas(agac_gost L) {
  if (L) {
    printf("%c", L->veri);
    kok_onde_dolas(L->solcocuk);
    kok_onde_dolas(L->sagcocuk);
  }
}
```

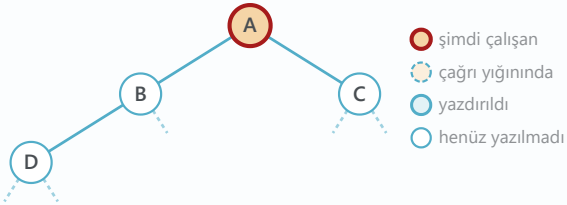
```c
void kok_ortada_dolas(agac_gost L) {
  if (L) {
    kok_ortada_dolas(L->solcocuk);
    printf("%c", L->veri);
    kok_ortada_dolas(L->sagcocuk);
  }
}
```

```c
void kok_sonda_dolas(agac_gost L) {
  if (L) {
    kok_sonda_dolas(L->solcocuk);
    kok_sonda_dolas(L->sagcocuk);
    printf("%c", L->veri);
  }
}
```

> **HOCADAN** "NOT: Ağaç işaretçisi NULL gelebilir, buna hiç dikkat edilmiyor!"

Üç yordam aynı iskelettir; yalnızca `printf` satırının yeri değişir. `if (L)` hem boş ağacı hem özyinelemenin durma koşulunu karşılar.

Kök ortada gezinme adım adım (çağrı yığınıyla)



L = A: sol alt ağaç için `kok_ortada_dolas(B)` çağrılır; bu çerçeve bekler.

Adım 1 / 18

```
1 void kok_ortada_dolas(agac_gost L) {
2   if (L) {
▶     kok_ortada_dolas(L->solcocuk);
4     printf("%c", L->veri);
5     kok_ortada_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

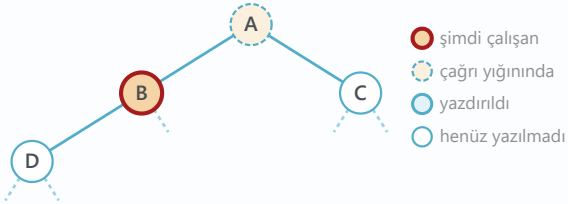
`L = &düğüm(A)`

ÇAĞRI YIĞINI

`kok_ortada_dolas(A)`

ÇIKTI

Kök ortada gezinme adım adım (çağrı yığınıyla)



L = B: sol alt ağaç için `kok_ortada_dolas(D)` çağrılır; bu çerçeve bekler.

Adım 2 / 18

```
1 void kok_ortada_dolas(agac_gost L) {
2   if (L) {
▶     kok_ortada_dolas(L->solcocuk);
4     printf("%c", L->veri);
5     kok_ortada_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

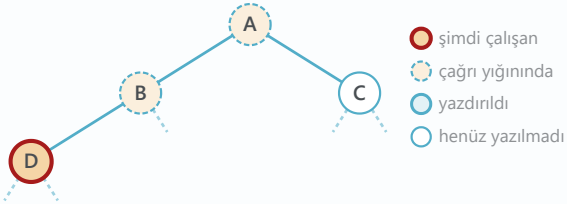
`L = &düğüm(B)`

ÇAĞRI YIĞINI

- `kok_ortada_dolas(B)`
- `kok_ortada_dolas(A)`

ÇIKTI

Kök ortada gezinme adım adım (çağrı yığınıyla)



L = D: sol alt ağaç için `kok_ortada_dolas(NULL)` çağrılır; bu çerçeve bekler.

Adım 3 / 18

```
1 void kok_ortada_dolas(agac_gost L) {
2   if (L) {
▶     kok_ortada_dolas(L->solcocuk);
4     printf("%c", L->veri);
5     kok_ortada_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

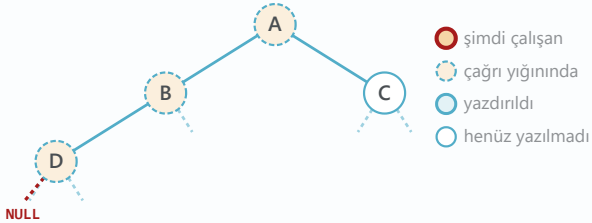
`L = &düğüm(D)`

ÇAĞRI YIĞINI

- `kok_ortada_dolas(D)`
- `kok_ortada_dolas(B)`
- `kok_ortada_dolas(A)`

ÇIKTI

Kök ortada gezinme adım adım (çağrı yığınıyla)



`L == NULL` → `if (L)` yanlış; bu çağrı hiçbir şey yapmadan döner (D'nin sol çocuğu yok).

Adım 4 / 18

```
1 void kok_ortada_dolas(agac_gost L) {
▶   if (L) {
3     kok_ortada_dolas(L->solcocuk);
4     printf("%c", L->veri);
5     kok_ortada_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

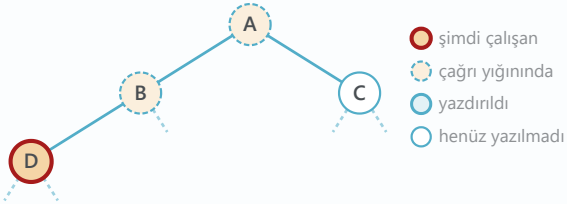
`L = NULL`

ÇAĞRI YIĞINI

- `kok_ortada_dolas(NULL)`
- `kok_ortada_dolas(D)`
- `kok_ortada_dolas(B)`
- `kok_ortada_dolas(A)`

ÇIKTI

Kök ortada gezinme adım adım (çağrı yığınıyla)



L = D: sol alt ağaç bitti; şimdi kökü yazdırır → çıktıya **D** eklenir.

Adım 5 / 18

```
1 void kok_ortada_dolas(agac_gost L) {
2   if (L) {
3     kok_ortada_dolas(L->solcocuk);
▶     printf("%c", L->veri);
5     kok_ortada_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(D)`

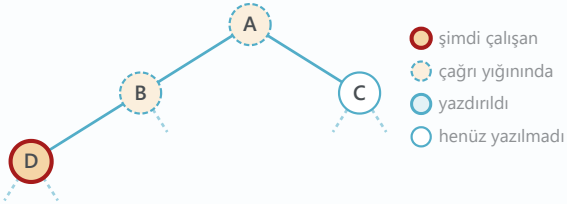
ÇAĞRI YIĞINI

- `kok_ortada_dolas(D)`
- `kok_ortada_dolas(B)`
- `kok_ortada_dolas(A)`

ÇIKTI

```
D
```

Kök ortada gezinme adım adım (çağrı yığınıyla)



L = D: sağ alt ağaç için `kok_ortada_dolas(NULL)` çağrılır; bu çerçeve bekler.

Adım 6 / 18

```
1 void kok_ortada_dolas(agac_gost L) {
2   if (L) {
3     kok_ortada_dolas(L->solcocuk);
4     printf("%c", L->veri);
▶     kok_ortada_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(D)`

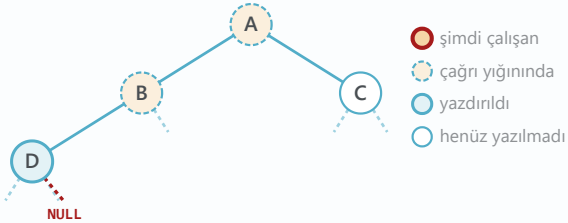
ÇAĞRI YIĞINI

- `kok_ortada_dolas(D)`
- `kok_ortada_dolas(B)`
- `kok_ortada_dolas(A)`

ÇIKTI

```
D
```

Kök ortada gezinme adım adım (çağrı yığınıyla)



`L == NULL` → `if (L)` yanlış; bu çağrı hiçbir şey yapmadan döner (D'nin sağ çocuğu yok).

Adım 7 / 18

```
1 void kok_ortada_dolas(agac_gost L) {
▶   if (L) {
3     kok_ortada_dolas(L->solcocuk);
4     printf("%c", L->veri);
5     kok_ortada_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = NULL`

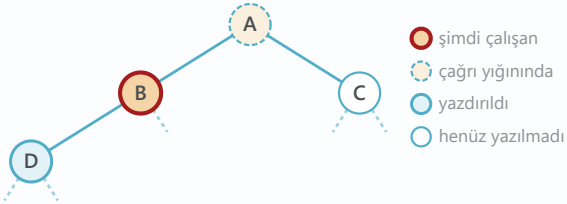
ÇAĞRI YIĞINI

- `kok_ortada_dolas(NULL)`
- `kok_ortada_dolas(D)`
- `kok_ortada_dolas(B)`
- `kok_ortada_dolas(A)`

ÇIKTI

```
D
```

Kök ortada gezinme adım adım (çağrı yığınıyla)



L = B: sol alt ağaç bitti; şimdi kökü yazdırır → çıktıya **B** eklenir.

Adım 8 / 18

```
1 void kok_ortada_dolas(agac_gost L) {
2   if (L) {
3     kok_ortada_dolas(L->solcocuk);
▶     printf("%c", L->veri);
5     kok_ortada_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(B)`

ÇAĞRI YIĞINI

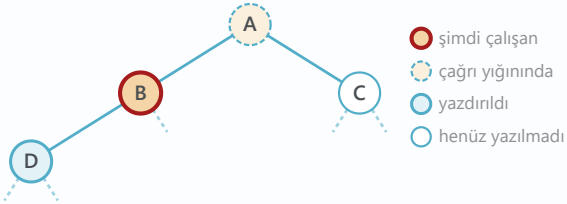
`kok_ortada_dolas(B)`

`kok_ortada_dolas(A)`

ÇIKTI

```
DB
```

Kök ortada gezinme adım adım (çağrı yığınıyla)



L = B: sağ alt ağaç için `kok_ortada_dolas(NULL)` çağrılır; bu çerçeve bekler.

Adım 9 / 18

```
1 void kok_ortada_dolas(agac_gost L) {
2   if (L) {
3     kok_ortada_dolas(L->solcocuk);
4     printf("%c", L->veri);
▶     kok_ortada_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(B)`

ÇAĞRI YIĞINI

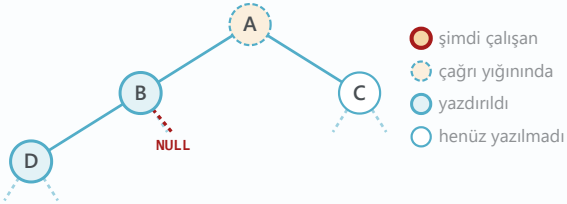
`kok_ortada_dolas(B)`

`kok_ortada_dolas(A)`

ÇIKTI

```
DB
```

Kök ortada gezinme adım adım (çağrı yığınıyla)



`L == NULL` → `if (L)` yanlış; bu çağrı hiçbir şey yapmadan döner (B'nin sağ çocuğu yok).

Adım 10 / 18

```
1 void kok_ortada_dolas(agac_gost L) {
▶   if (L) {
3     kok_ortada_dolas(L->solcocuk);
4     printf("%c", L->veri);
5     kok_ortada_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = NULL`

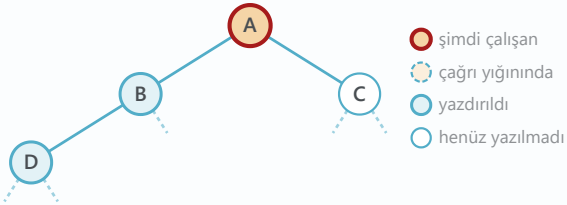
ÇAĞRI YIĞINI

- `kok_ortada_dolas(NULL)`
- `kok_ortada_dolas(B)`
- `kok_ortada_dolas(A)`

ÇIKTI

```
DB
```

Kök ortada gezinme adım adım (çağrı yığınıyla)



L = A: sol alt ağaç bitti; şimdi kökü yazdırır → çıktıya **A** eklenir.


```
1 void kok_ortada_dolas(agac_gost L) {
2   if (L) {
3     kok_ortada_dolas(L->solcocuk);
▶     printf("%c", L->veri);
5     kok_ortada_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(A)`

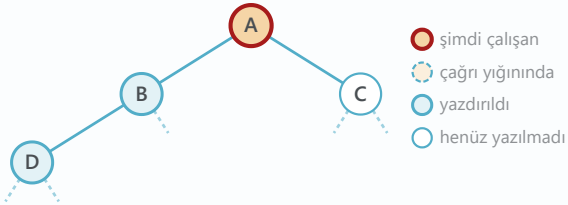
ÇAĞRI YIĞINI

`kok_ortada_dolas(A)`

ÇIKTI

```
DBA
```

Kök ortada gezinme adım adım (çağrı yığınıyla)



L = A: sağ alt ağaç için `kok_ortada_dolas(C)` çağrılır; bu çerçeve bekler.

Adım 12 / 18

```
1 void kok_ortada_dolas(agac_gost L) {
2   if (L) {
3     kok_ortada_dolas(L->solcocuk);
4     printf("%c", L->veri);
▶     kok_ortada_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(A)`

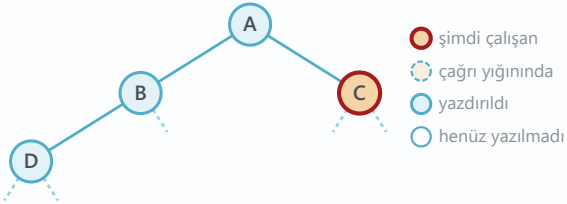
ÇAĞRI YIĞINI

`kok_ortada_dolas(A)`

ÇIKTI

```
DBA
```

Kök ortada gezinme adım adım (çağrı yığınıyla)



L = C: sol alt ağaç için `kok_ortada_dolas(NULL)` çağrılır; bu çerçeve bekler.

Adım 13 / 18

```
1 void kok_ortada_dolas(agac_gost L) {
2   if (L) {
▶     kok_ortada_dolas(L->solcocuk);
4     printf("%c", L->veri);
5     kok_ortada_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(C)`

ÇAĞRI YIĞINI

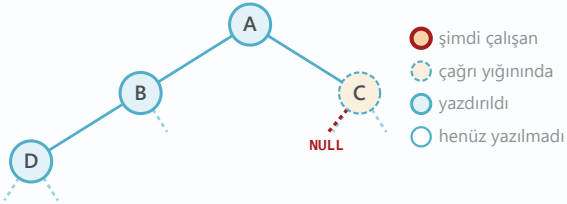
`kok_ortada_dolas(C)`

`kok_ortada_dolas(A)`

ÇIKTI

```
DBA
```

Kök ortada gezinme adım adım (çağrı yığınıyla)



`L == NULL` → `if (L)` yanlış; bu çağrı hiçbir şey yapmadan döner (C'nin sol çocuğu yok).

Adım 14 / 18

```
1 void kok_ortada_dolas(agac_gost L) {
▶   if (L) {
3     kok_ortada_dolas(L->solcocuk);
4     printf("%c", L->veri);
5     kok_ortada_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = NULL`

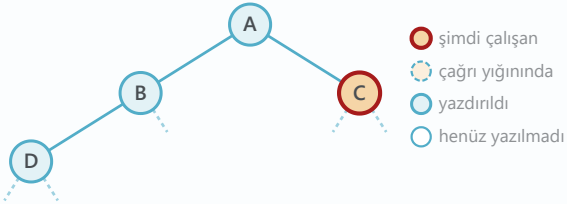
ÇAĞRI YIĞINI

- `kok_ortada_dolas(NULL)`
- `kok_ortada_dolas(C)`
- `kok_ortada_dolas(A)`

ÇIKTI

```
DBA
```

Kök ortada gezinme adım adım (çağrı yığınıyla)



L = C: sol alt ağaç bitti; şimdi kökü yazdırır → çıktıya **C** eklenir.

Adım 15 / 18

```
1 void kok_ortada_dolas(agac_gost L) {
2   if (L) {
3     kok_ortada_dolas(L->solcocuk);
▶     printf("%c", L->veri);
5     kok_ortada_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(C)`

ÇAĞRI YIĞINI

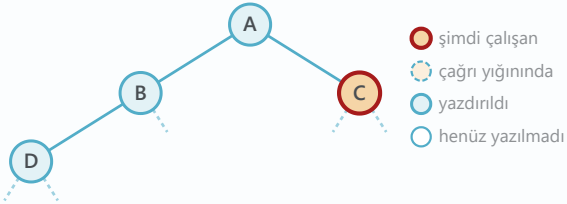
`kok_ortada_dolas(C)`

`kok_ortada_dolas(A)`

ÇIKTI

DBAC

Kök ortada gezinme adım adım (çağrı yığınıyla)



L = C: sağ alt ağaç için `kok_ortada_dolas(NULL)` çağrılır; bu çerçeve bekler.

Adım 16 / 18

```
1 void kok_ortada_dolas(agac_gost L) {
2   if (L) {
3     kok_ortada_dolas(L->solcocuk);
4     printf("%c", L->veri);
▶     kok_ortada_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(C)`

ÇAĞRI YIĞINI

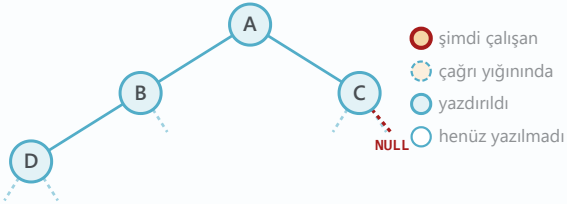
`kok_ortada_dolas(C)`

`kok_ortada_dolas(A)`

ÇIKTI

```
DBAC
```

Kök ortada gezinme adım adım (çağrı yığınıyla)



`L == NULL` → `if (L)` yanlış; bu çağrı hiçbir şey yapmadan döner (C'nin sağ çocuğu yok).

Adım 17 / 18

```
1 void kok_ortada_dolas(agac_gost L) {
▶   if (L) {
3     kok_ortada_dolas(L->solcocuk);
4     printf("%c", L->veri);
5     kok_ortada_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = NULL`

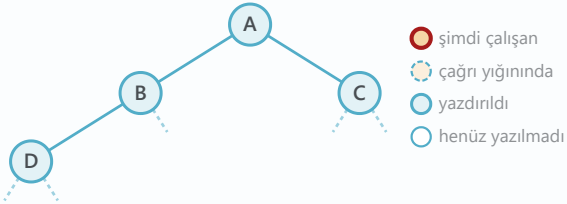
ÇAĞRI YIĞINI

- `kok_ortada_dolas(NULL)`
- `kok_ortada_dolas(C)`
- `kok_ortada_dolas(A)`

ÇIKTI

```
DBAC
```

Kök ortada gezinme adım adım (çağrı yığınıyla)



Tüm çağrılar döndü. Çıktı: **DBAC**. Her düğüme bir kez girildi, bir kez çıkıldı: 2n adım.

Adım 18 / 18

```
1 void kok_ortada_dolas(agac_gost L) {
2   if (L) {
3     kok_ortada_dolas(L->solcocuk);
4     printf("%c", L->veri);
5     kok_ortada_dolas(L->sagcocuk);
6   }
▶ }
```

DEĞİŞKENLER

`L = –`

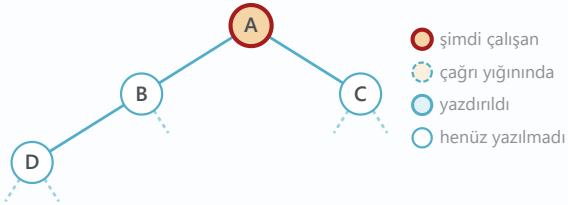
ÇAĞRI YIĞINI

`main()`

ÇIKTI

```
DBAC
```

Kök önde gezinme adım adım



L = A: kök önde: önce kendini yazdırır → çıktıya **A** eklenir.

Adım 1 / 18

```
1 void kok_onde_dolas(agac_gost L) {
2   if (L) {
▶     printf("%c", L->veri);
4     kok_onde_dolas(L->solcocuk);
5     kok_onde_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(A)`

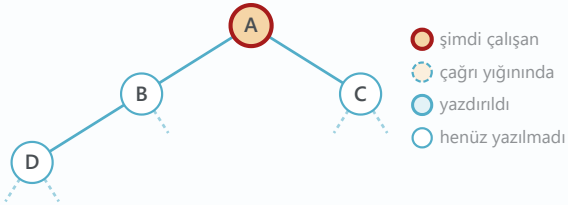
ÇAĞRI YIĞINI

`kok_onde_dolas(A)`

ÇIKTI

```
A
```

Kök önde gezinme adım adım



L = A: sol alt ağaç için `kok_onde_dolas(B)` çağrılır; bu çerçeve bekler.

Adım 2 / 18

```
1 void kok_onde_dolas(agac_gost L) {
2   if (L) {
3     printf("%c", L->veri);
▶     kok_onde_dolas(L->solcocuk);
5     kok_onde_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(A)`

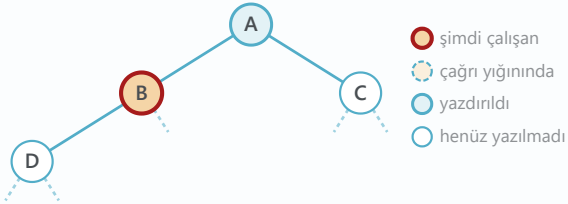
ÇAĞRI YIĞINI

`kok_onde_dolas(A)`

ÇIKTI

```
A
```

Kök önde gezinme adım adım



L = B: kök önde: önce kendini yazdırır → çıktıya **B** eklenir.

Adım 3 / 18

```
1 void kok_onde_dolas(agac_gost L) {
2   if (L) {
▶     printf("%c", L->veri);
4     kok_onde_dolas(L->solcocuk);
5     kok_onde_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(B)`

ÇAĞRI YIĞINI

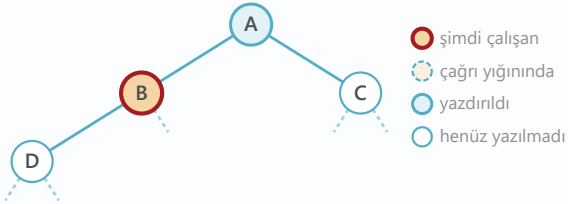
`kok_onde_dolas(B)`

`kok_onde_dolas(A)`

ÇIKTI

AB

Kök önde gezinme adım adım



L = B: sol alt ağaç için `kok_onde_dolas(D)` çağrılır; bu çerçeve bekler.

Adım 4 / 18

```
1 void kok_onde_dolas(agac_gost L) {
2   if (L) {
3     printf("%c", L->veri);
▶     kok_onde_dolas(L->solcocuk);
5     kok_onde_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(B)`

ÇAĞRI YIĞINI

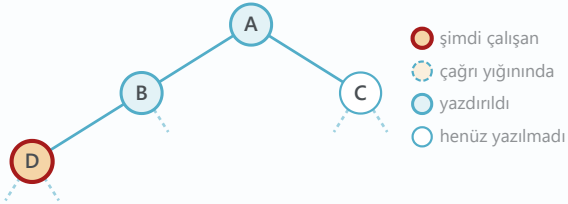
`kok_onde_dolas(B)`

`kok_onde_dolas(A)`

ÇIKTI

```
AB
```

Kök önde gezinme adım adım



L = D: kök önde: önce kendini yazdırır → çıktıya **D** eklenir.

Adım 5 / 18

```
1 void kok_onde_dolas(agac_gost L) {
2   if (L) {
▶     printf("%c", L->veri);
4     kok_onde_dolas(L->solcocuk);
5     kok_onde_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(D)`

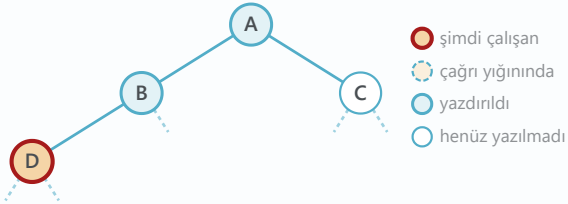
ÇAĞRI YIĞINI

- `kok_onde_dolas(D)`
- `kok_onde_dolas(B)`
- `kok_onde_dolas(A)`

ÇIKTI

```
ABD
```

Kök önde gezinme adım adım



L = D: sol alt ağaç için `kok_onde_dolas(NULL)` çağrılır; bu çerçeve bekler.

Adım 6 / 18

```
1 void kok_onde_dolas(agac_gost L) {
2   if (L) {
3     printf("%c", L->veri);
▶     kok_onde_dolas(L->solcocuk);
5     kok_onde_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

L = &düğüm(D)

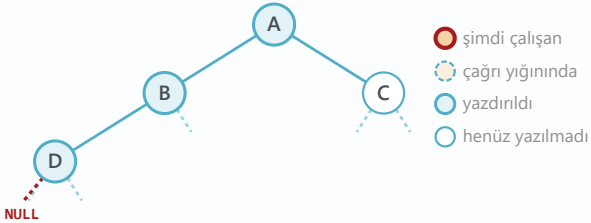
ÇAĞRI YIĞINI

- kok_onde_dolas(D)
- kok_onde_dolas(B)
- kok_onde_dolas(A)

ÇIKTI

ABD

Kök önde gezinme adım adım



`L == NULL` → `if (L)` yanlış; bu çağrı hiçbir şey yapmadan döner (D'nin sol çocuğu yok).

Adım 7 / 18

```
1 void kok_onde_dolas(agac_gost L) {
▶   if (L) {
3     printf("%c", L->veri);
4     kok_onde_dolas(L->solcocuk);
5     kok_onde_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = NULL`

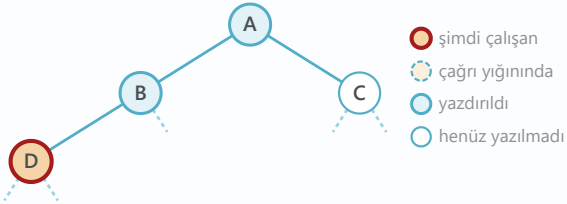
ÇAĞRI YIĞINI

- `kok_onde_dolas(NULL)`
- `kok_onde_dolas(D)`
- `kok_onde_dolas(B)`
- `kok_onde_dolas(A)`

ÇIKTI

```
ABD
```

Kök önde gezinme adım adım



L = D: sağ alt ağaç için `kok_onde_dolas(NULL)` çağrılır; bu çerçeve bekler.

Adım 8 / 18

```
1 void kok_onde_dolas(agac_gost L) {
2   if (L) {
3     printf("%c", L->veri);
4     kok_onde_dolas(L->solcocuk);
▶     kok_onde_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(D)`

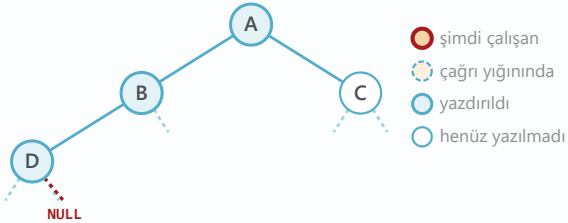
ÇAĞRI YIĞINI

- `kok_onde_dolas(D)`
- `kok_onde_dolas(B)`
- `kok_onde_dolas(A)`

ÇIKTI

```
ABD
```

Kök önde gezinme adım adım



`L == NULL` → `if (L)` yanlış; bu çağrı hiçbir şey yapmadan döner (D'nin sağ çocuğu yok).

Adım 9 / 18

```
1 void kok_onde_dolas(agac_gost L) {
▶   if (L) {
3     printf("%c", L->veri);
4     kok_onde_dolas(L->solcocuk);
5     kok_onde_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = NULL`

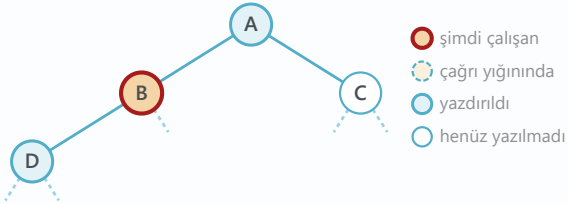
ÇAĞRI YIĞINI

- `kok_onde_dolas(NULL)`
- `kok_onde_dolas(D)`
- `kok_onde_dolas(B)`
- `kok_onde_dolas(A)`

ÇIKTI

```
ABD
```

Kök önde gezinme adım adım



L = B: sağ alt ağaç için `kok_onde_dolas(NULL)` çağrılır; bu çerçeve bekler.

Adım 10 / 18

```
1 void kok_onde_dolas(agac_gost L) {
2   if (L) {
3     printf("%c", L->veri);
4     kok_onde_dolas(L->solcocuk);
▶     kok_onde_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(B)`

ÇAĞRI YIĞINI

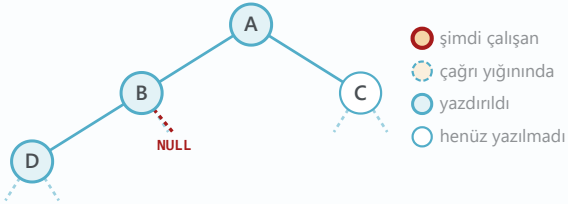
`kok_onde_dolas(B)`

`kok_onde_dolas(A)`

ÇIKTI

```
ABD
```

Kök önde gezinme adım adım



`L == NULL` → `if (L)` yanlış; bu çağrı hiçbir şey yapmadan döner (B'nin sağ çocuğu yok).

Adım 11 / 18

```
1 void kok_onde_dolas(agac_gost L) {
▶   if (L) {
3     printf("%c", L->veri);
4     kok_onde_dolas(L->solcocuk);
5     kok_onde_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = NULL`

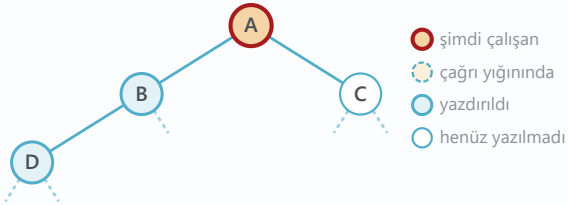
ÇAĞRI YIĞINI

- `kok_onde_dolas(NULL)`
- `kok_onde_dolas(B)`
- `kok_onde_dolas(A)`

ÇIKTI

ABD

Kök önde gezinme adım adım



L = A: sağ alt ağaç için `kok_onde_dolas(C)` çağrılır; bu çerçeve bekler.

Adım 12 / 18

```
1 void kok_onde_dolas(agac_gost L) {
2   if (L) {
3     printf("%c", L->veri);
4     kok_onde_dolas(L->solcocuk);
▶     kok_onde_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(A)`

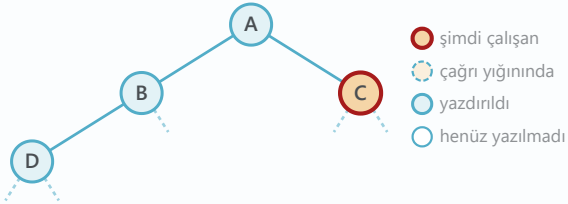
ÇAĞRI YIĞINI

`kok_onde_dolas(A)`

ÇIKTI

```
ABD
```

Kök önde gezinme adım adım



L = C: kök önde: önce kendini yazdırır → çıktıya **C** eklenir.

Adım 13 / 18

```
1 void kok_onde_dolas(agac_gost L) {
2   if (L) {
▶     printf("%c", L->veri);
4     kok_onde_dolas(L->solcocuk);
5     kok_onde_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(C)`

ÇAĞRI YIĞINI

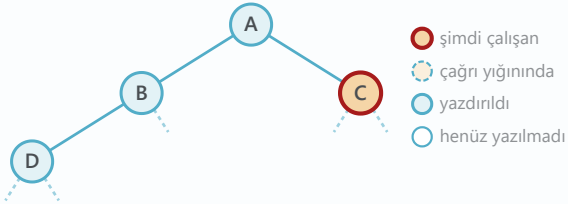
`kok_onde_dolas(C)`

`kok_onde_dolas(A)`

ÇIKTI

ABDC

Kök önde gezinme adım adım



L = C: sol alt ağaç için `kok_onde_dolas(NULL)` çağrılır; bu çerçeve bekler.

Adım 14 / 18

```
1 void kok_onde_dolas(agac_gost L) {
2   if (L) {
3     printf("%c", L->veri);
▶     kok_onde_dolas(L->solcocuk);
5     kok_onde_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(C)`

ÇAĞRI YIĞINI

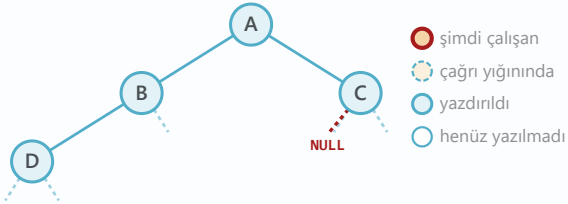
`kok_onde_dolas(C)`

`kok_onde_dolas(A)`

ÇIKTI

```
ABDC
```

Kök önde gezinme adım adım



`L == NULL` → `if (L)` yanlış; bu çağrı hiçbir şey yapmadan döner (C'nin sol çocuğu yok).

Adım 15 / 18

```
1 void kok_onde_dolas(agac_gost L) {
▶   if (L) {
3     printf("%c", L->veri);
4     kok_onde_dolas(L->solcocuk);
5     kok_onde_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = NULL`

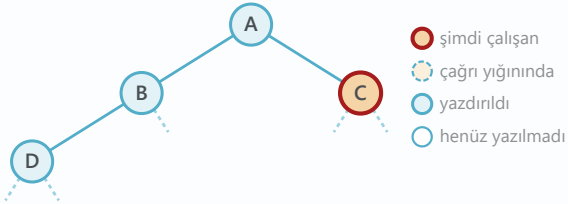
ÇAĞRI YIĞINI

- `kok_onde_dolas(NULL)`
- `kok_onde_dolas(C)`
- `kok_onde_dolas(A)`

ÇIKTI

ABDC

Kök önde gezinme adım adım



L = C: sağ alt ağaç için `kok_onde_dolas(NULL)` çağrılır; bu çerçeve bekler.

Adım 16 / 18

```
1 void kok_onde_dolas(agac_gost L) {
2   if (L) {
3     printf("%c", L->veri);
4     kok_onde_dolas(L->solcocuk);
▶     kok_onde_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(C)`

ÇAĞRI YIĞINI

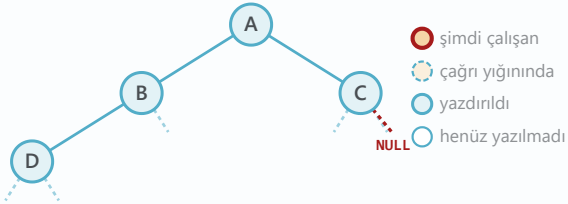
`kok_onde_dolas(C)`

`kok_onde_dolas(A)`

ÇIKTI

```
ABDC
```

Kök önde gezinme adım adım



`L == NULL` → `if (L)` yanlış; bu çağrı hiçbir şey yapmadan döner (C'nin sağ çocuğu yok).

Adım 17 / 18

```
1 void kok_onde_dolas(agac_gost L) {
▶   if (L) {
3     printf("%c", L->veri);
4     kok_onde_dolas(L->solcocuk);
5     kok_onde_dolas(L->sagcocuk);
6   }
7 }
```

DEĞİŞKENLER

`L = NULL`

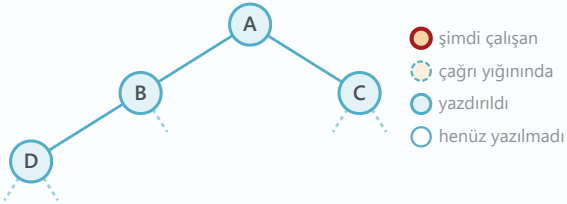
ÇAĞRI YIĞINI

- `kok_onde_dolas(NULL)`
- `kok_onde_dolas(C)`
- `kok_onde_dolas(A)`

ÇIKTI

ABDC

Kök önde gezinme adım adım



Tüm çağrılar döndü. Çıktı: **ABDC**. Her düğüme bir kez girildi, bir kez çıkıldı: 2n adım.

Adım 18 / 18

```
1 void kok_onde_dolas(agac_gost L) {
2   if (L) {
3     printf("%c", L->veri);
4     kok_onde_dolas(L->solcocuk);
5     kok_onde_dolas(L->sagcocuk);
6   }
▶ }
```

DEĞİŞKENLER

`L = –`

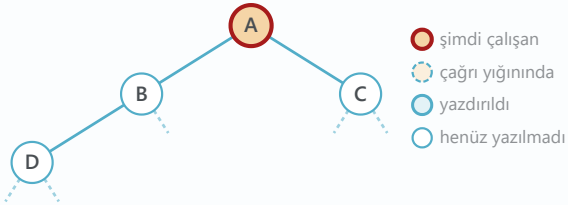
ÇAĞRI YIĞINI

`main()`

ÇIKTI

ABDC

Kök sonda gezinme adım adım



L = A: sol alt ağaç için `kok_sonda_dolas(B)` çağrılır; bu çerçeve bekler.

Adım 1 / 18

```
1 void kok_sonda_dolas(agac_gost L) {
2   if (L) {
▶     kok_sonda_dolas(L->solcocuk);
4     kok_sonda_dolas(L->sagcocuk);
5     printf("%c", L->veri);
6   }
7 }
```

DEĞİŞKENLER

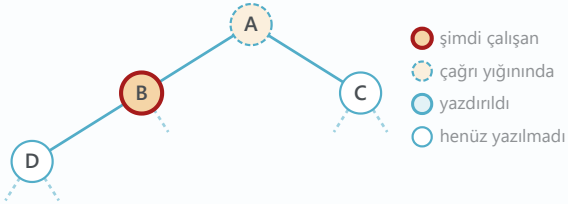
L = &düğüm(A)

ÇAĞRI YIĞINI

kok_sonda_dolas(A)

ÇIKTI

Kök sonda gezinme adım adım



L = B: sol alt ağaç için `kok_sonda_dolas(D)` çağrılır; bu çerçeve bekler.

Adım 2 / 18

```
1 void kok_sonda_dolas(agac_gost L) {
2   if (L) {
▶     kok_sonda_dolas(L->solcocuk);
4     kok_sonda_dolas(L->sagcocuk);
5     printf("%c", L->veri);
6   }
7 }
```

DEĞİŞKENLER

L = &düğüm(B)

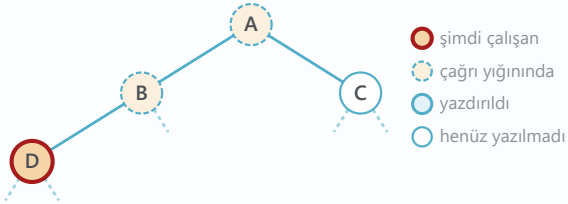
ÇAĞRI YIĞINI

kok_sonda_dolas(B)

kok_sonda_dolas(A)

ÇIKTI

Kök sonda gezinme adım adım



L = D: sol alt ağaç için `kok_sonda_dolas(NULL)` çağrılır; bu çerçeve bekler.

Adım 3 / 18

```
1 void kok_sonda_dolas(agac_gost L) {
2   if (L) {
▶     kok_sonda_dolas(L->solcocuk);
4     kok_sonda_dolas(L->sagcocuk);
5     printf("%c", L->veri);
6   }
7 }
```

DEĞİŞKENLER

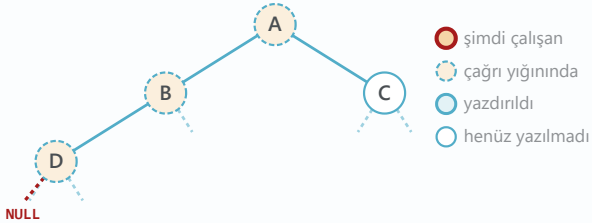
`L = &düğüm(D)`

ÇAĞRI YIĞINI

- `kok_sonda_dolas(D)`
- `kok_sonda_dolas(B)`
- `kok_sonda_dolas(A)`

ÇIKTI

Kök sonda gezinme adım adım



`L == NULL` → `if (L)` yanlış; bu çağrı hiçbir şey yapmadan döner (D'nin sol çocuğu yok).

Adım 4 / 18

```
1 void kok_sonda_dolas(agac_gost L) {
▶   if (L) {
3     kok_sonda_dolas(L->solcocuk);
4     kok_sonda_dolas(L->sagcocuk);
5     printf("%c", L->veri);
6   }
7 }
```

DEĞİŞKENLER

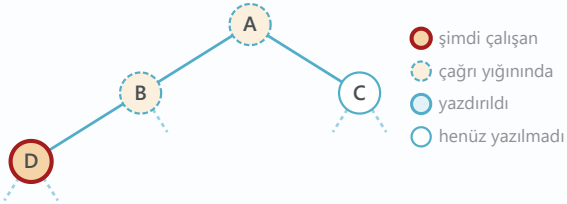
`L = NULL`

ÇAĞRI YIĞINI

- `kok_sonda_dolas(NULL)`
- `kok_sonda_dolas(D)`
- `kok_sonda_dolas(B)`
- `kok_sonda_dolas(A)`

ÇIKTI

Kök sonda gezinme adım adım



L = D: sağ alt ağaç için `kok_sonda_dolas(NULL)` çağrılır; bu çerçeve bekler.

Adım 5 / 18

```
1 void kok_sonda_dolas(agac_gost L) {
2   if (L) {
3     kok_sonda_dolas(L->solcocuk);
▶     kok_sonda_dolas(L->sagcocuk);
5     printf("%c", L->veri);
6   }
7 }
```

DEĞİŞKENLER

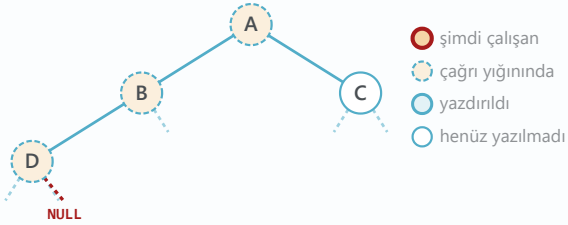
`L = &düğüm(D)`

ÇAĞRI YIĞINI

- `kok_sonda_dolas(D)`
- `kok_sonda_dolas(B)`
- `kok_sonda_dolas(A)`

ÇIKTI

Kök sonda gezinme adım adım



`L == NULL` → `if (L)` yanlış; bu çağrı hiçbir şey yapmadan döner (D'nin sağ çocuğu yok).

Adım 6 / 18

```
1 void kok_sonda_dolas(agac_gost L) {
▶   if (L) {
3     kok_sonda_dolas(L->solcocuk);
4     kok_sonda_dolas(L->sagcocuk);
5     printf("%c", L->veri);
6   }
7 }
```

DEĞİŞKENLER

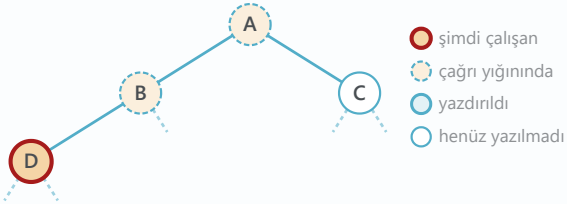
`L = NULL`

ÇAĞRI YIĞINI

- `kok_sonda_dolas(NULL)`
- `kok_sonda_dolas(D)`
- `kok_sonda_dolas(B)`
- `kok_sonda_dolas(A)`

ÇIKTI

Kök sonda gezinme adım adım



L = D: iki alt ağaç da bitti; en son kökü yazdırır → çıktıya **D** eklenir.

Adım 7 / 18

```
1 void kok_sonda_dolas(agac_gost L) {
2   if (L) {
3     kok_sonda_dolas(L->solcocuk);
4     kok_sonda_dolas(L->sagcocuk);
▶     printf("%c", L->veri);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(D)`

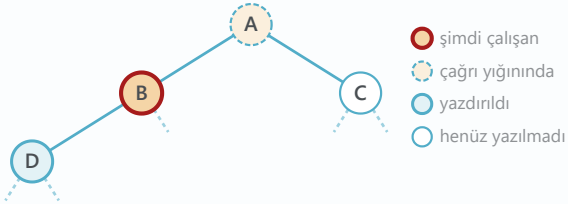
ÇAĞRI YIĞINI

- `kok_sonda_dolas(D)`
- `kok_sonda_dolas(B)`
- `kok_sonda_dolas(A)`

ÇIKTI

```
D
```

Kök sonda gezinme adım adım



L = B: sağ alt ağaç için `kok_sonda_dolas(NULL)` çağrılır; bu çerçeve bekler.

Adım 8 / 18

```
1 void kok_sonda_dolas(agac_gost L) {
2   if (L) {
3     kok_sonda_dolas(L->solcocuk);
▶    kok_sonda_dolas(L->sagcocuk);
5     printf("%c", L->veri);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(B)`

ÇAĞRI YIĞINI

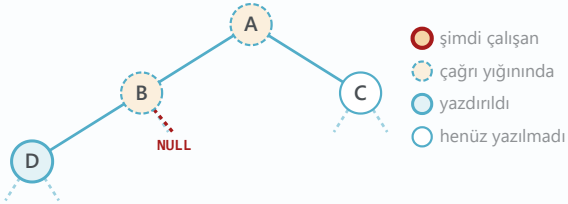
`kok_sonda_dolas(B)`

`kok_sonda_dolas(A)`

ÇIKTI

```
D
```

Kök sonda gezinme adım adım



`L == NULL` → `if (L)` yanlış; bu çağrı hiçbir şey yapmadan döner (B'nin sağ çocuğu yok).

Adım 9 / 18

```
1 void kok_sonda_dolas(agac_gost L) {
▶   if (L) {
3     kok_sonda_dolas(L->solcocuk);
4     kok_sonda_dolas(L->sagcocuk);
5     printf("%c", L->veri);
6   }
7 }
```

DEĞİŞKENLER

`L = NULL`

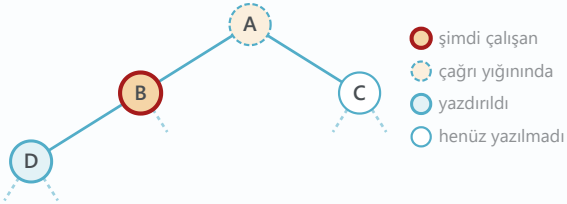
ÇAĞRI YIĞINI

- `kok_sonda_dolas(NULL)`
- `kok_sonda_dolas(B)`
- `kok_sonda_dolas(A)`

ÇIKTI

```
D
```

Kök sonda gezinme adım adım



L = B: iki alt ağaç da bitti; en son kökü yazdırır → çıktıya **B** eklenir.

Adım 10 / 18

```
1 void kok_sonda_dolas(agac_gost L) {
2   if (L) {
3     kok_sonda_dolas(L->solcocuk);
4     kok_sonda_dolas(L->sagcocuk);
▶     printf("%c", L->veri);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(B)`

ÇAĞRI YIĞINI

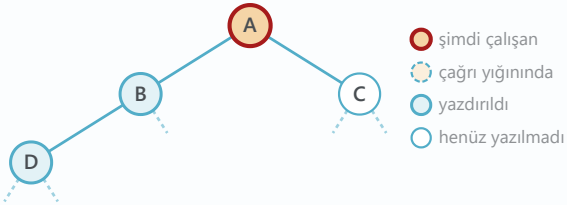
`kok_sonda_dolas(B)`

`kok_sonda_dolas(A)`

ÇIKTI

```
DB
```

Kök sonda gezinme adım adım



L = A: sağ alt ağaç için `kok_sonda_dolas(C)` çağrılır; bu çerçeve bekler.

Adım 11 / 18

```
1 void kok_sonda_dolas(agac_gost L) {
2   if (L) {
3     kok_sonda_dolas(L->solcocuk);
▶    kok_sonda_dolas(L->sagcocuk);
5     printf("%c", L->veri);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(A)`

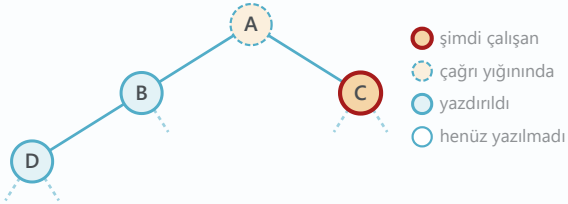
ÇAĞRI YIĞINI

`kok_sonda_dolas(A)`

ÇIKTI

```
DB
```

Kök sonda gezinme adım adım



L = C: sol alt ağaç için `kok_sonda_dolas(NULL)` çağrılır; bu çerçeve bekler.

Adım 12 / 18

```
1 void kok_sonda_dolas(agac_gost L) {
2   if (L) {
▶     kok_sonda_dolas(L->solcocuk);
4     kok_sonda_dolas(L->sagcocuk);
5     printf("%c", L->veri);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(C)`

ÇAĞRI YIĞINI

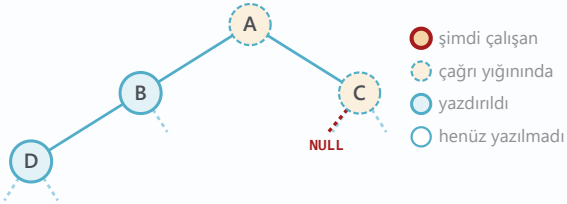
`kok_sonda_dolas(C)`

`kok_sonda_dolas(A)`

ÇIKTI

```
DB
```

Kök sonda gezinme adım adım



`L == NULL` → `if (L)` yanlış; bu çağrı hiçbir şey yapmadan döner (C'nin sol çocuğu yok).

Adım 13 / 18

```
1 void kok_sonda_dolas(agac_gost L) {
▶   if (L) {
3     kok_sonda_dolas(L->solcocuk);
4     kok_sonda_dolas(L->sagcocuk);
5     printf("%c", L->veri);
6   }
7 }
```

DEĞİŞKENLER

`L = NULL`

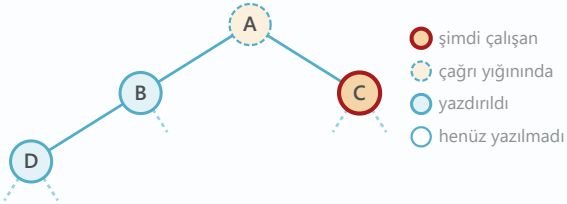
ÇAĞRI YIĞINI

- `kok_sonda_dolas(NULL)`
- `kok_sonda_dolas(C)`
- `kok_sonda_dolas(A)`

ÇIKTI

```
DB
```

Kök sonda gezinme adım adım



L = C: sağ alt ağaç için `kok_sonda_dolas(NULL)` çağrılır; bu çerçeve bekler.

Adım 14 / 18

```
1 void kok_sonda_dolas(agac_gost L) {
2   if (L) {
3     kok_sonda_dolas(L->solcocuk);
▶     kok_sonda_dolas(L->sagcocuk);
5     printf("%c", L->veri);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(C)`

ÇAĞRI YIĞINI

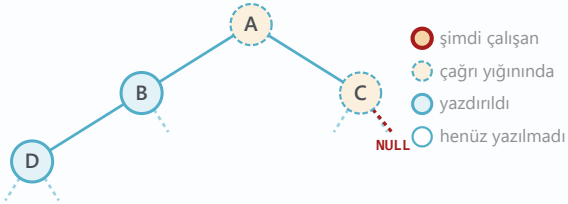
`kok_sonda_dolas(C)`

`kok_sonda_dolas(A)`

ÇIKTI

```
DB
```

Kök sonda gezinme adım adım



`L == NULL` → `if (L)` yanlış; bu çağrı hiçbir şey yapmadan döner (C'nin sağ çocuğu yok).

Adım 15 / 18

```
1 void kok_sonda_dolas(agac_gost L) {
▶   if (L) {
3     kok_sonda_dolas(L->solcocuk);
4     kok_sonda_dolas(L->sagcocuk);
5     printf("%c", L->veri);
6   }
7 }
```

DEĞİŞKENLER

`L = NULL`

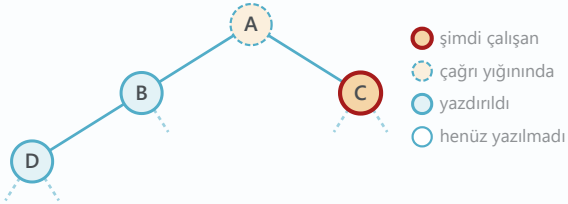
ÇAĞRI YIĞINI

- `kok_sonda_dolas(NULL)`
- `kok_sonda_dolas(C)`
- `kok_sonda_dolas(A)`

ÇIKTI

```
DB
```

Kök sonda gezinme adım adım



L = C: iki alt ağaç da bitti; en son kökü yazdırır → çıktıya **C** eklenir.

Adım 16 / 18

```
1 void kok_sonda_dolas(agac_gost L) {
2   if (L) {
3     kok_sonda_dolas(L->solcocuk);
4     kok_sonda_dolas(L->sagcocuk);
▶     printf("%c", L->veri);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(C)`

ÇAĞRI YIĞINI

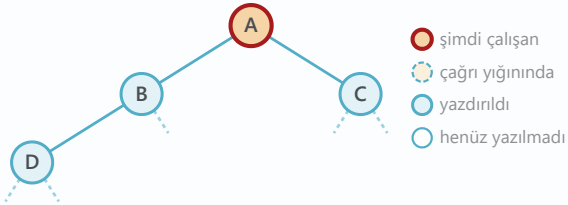
`kok_sonda_dolas(C)`

`kok_sonda_dolas(A)`

ÇIKTI

```
DBC
```

Kök sonda gezinme adım adım



L = A: iki alt ağaç da bitti; en son kökü yazdırır → çıktıya **A** eklenir.

Adım 17 / 18

```
1 void kok_sonda_dolas(agac_gost L) {
2   if (L) {
3     kok_sonda_dolas(L->solcocuk);
4     kok_sonda_dolas(L->sagcocuk);
▶     printf("%c", L->veri);
6   }
7 }
```

DEĞİŞKENLER

`L = &düğüm(A)`

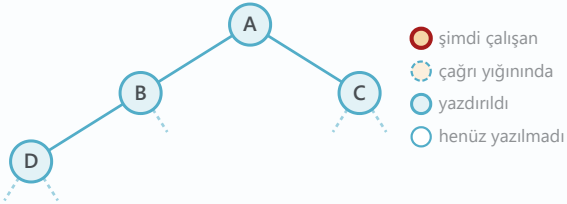
ÇAĞRI YIĞINI

`kok_sonda_dolas(A)`

ÇIKTI

```
DBCA
```

Kök sonda gezinme adım adım



Tüm çağrılar döndü. Çıktı: **DBCA**. Her düğüme bir kez girildi, bir kez çıkıldı: 2n adım.

Adım 18 / 18

```
1 void kok_sonda_dolas(agac_gost L) {
2   if (L) {
3     kok_sonda_dolas(L->solcocuk);
4     kok_sonda_dolas(L->sagcocuk);
5     printf("%c", L->veri);
6   }
▶ }
```

DEĞİŞKENLER

`L = –`

ÇAĞRI YIĞINI

`main()`

ÇIKTI

```
DBCA
```

Gezinmenin maliyeti ve NULL çağrıları

- **Zaman**: her düğüme bir girip bir çıkılır → 2n = **O(n)**, değişmez.
- **Bellek**: çağrı yığını en fazla derinlik (h) kadar çerçeve tutar → **O(h)**. Dengeli ağaçta O(lg n); çarpık ağaçta O(n).
- n düğümlü ağaçta n+1 NULL çağrısı yapılır. Çağrıdan önce denetlenirse bunlar kalkar:

```
/* önce: n+1 NULL çağrısı */
kok_onde_dolas(L->solcocuk);
kok_onde_dolas(L->sagcocuk);
/* sonra: çağırmadan denetle */
if (L->solcocuk) kok_onde_dolas(L->solcocuk);
if (L->sagcocuk) kok_onde_dolas(L->sagcocuk);
```

İlk çağrıdaki `if (L)` yine kalmalı: ağaç boş gelebilir. Bellek için O(lg n) sınırı yalnız dengeli ağaçta geçerlidir; genel sınır O(h).

HOCADAN "'Hocam, özyinelemeli algoritmalar çok bellek yer' tarzındaki iddialar, özyinelemeli kod yazmamak için kaçamaktır."

HOCADAN "Algoritmayı yazdıktan sonra peşini bırakmıyoruz. Bilgisayar bilimlerinde algoritmanın analizini yapmak zorundayız."

GEZİNMELER

Özyinelemeli kod bizzat yazılarak öğrenilir

HOCADAN “Geçmişteki tüm sorularımı internete koysam, bu yanıltıcı olur. Naim Süleymanoğlu’nu halter kaldırırken 100 kere seyretsem, yine de ben o halteri kaldıramam. ‘Bu nasıl yazıldı’ sorusuna yoğunlaşmak gerekir.”

HOCADAN “Bu algoritma hâlâ anlaşılmadıysa, ağaç şeklindeki stack frame’i çizmek anlamaya yardımcı olacaktır.”

Duyarsam unuturum, görürsem hatırlarım, yaparsam öğrenirim.

— Çin atasözü

Çağrı ağacı

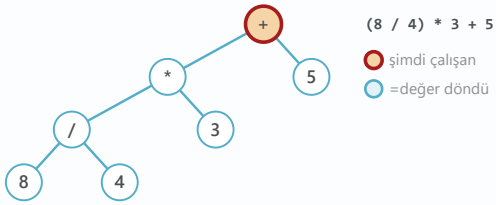
Her çağrı bir düğüm, her alt çağrı bir çocuk: gezinmenin çağrı ağacı, NULL çağrıları yaprak olmak üzere ağacın kendisidir (n iç + n+1 yaprak).

İfade ağacını hesaplamak: kök sonda gezinme

```
typedef struct ifade *ifade_gost;
struct ifade {
    char      islec;   /* '+', '-', '*', '/' */
    int       deger;   /* yalnız yapraklarda */
    ifade_gost sol, sag;
};
```

- Bir işlecin değeri, iki alt ifadenin değerleri bilinmeden hesaplanamaz → önce sol, sonra sağ, en son kök: **kök sonda**.
- Her iç düğümün tam iki çocuğu vardır (ikili işleçler); yaprakta işleç yok, sayı var.
- Maliyet O(n) zaman, O(h) yığın. Tekli eksi (−x) için tek çocuklu düğüm gerekir.

hesapla(): (8 / 4) * 3 + 5 = 11



`+` bir işleç: önce sol alt ifade hesaplanır.

Adım 1 / 13

```
1  int hesapla(ifade_gost t) {
2    if (!t->sol && !t->sag)        /* yaprak: sayı */
3      return t->deger;
▶    int a = hesapla(t->sol);
5    int b = hesapla(t->sag);
6    switch (t->islec) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

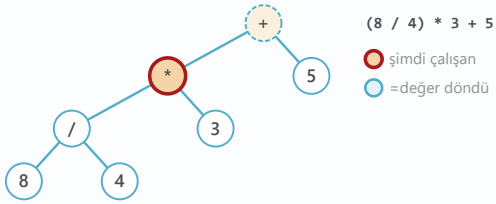
DEĞİŞKENLER

`t = &düğüm(+)` `a = ?` `b = ?`

ÇAĞRI YIĞINI

`hesapla(+)`

hesapla(): (8 / 4) * 3 + 5 = 11



`*` bir işleç: önce sol alt ifade hesaplanır.

Adım 2 / 13

```
1  int hesapla(ifade_gost t) {
2    if (!t->sol && !t->sag)        /* yaprak: sayı */
3      return t->deger;
▶    int a = hesapla(t->sol);
5    int b = hesapla(t->sag);
6    switch (t->islec) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

DEĞİŞKENLER

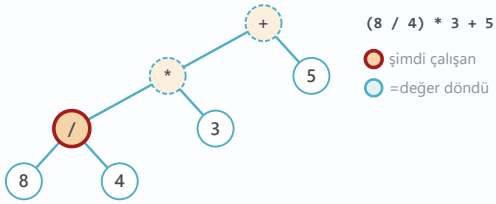
`t = &düğüm(*)` `a = ?` `b = ?`

ÇAĞRI YIĞINI

`hesapla(*)`

`hesapla(+)`

hesapla(): (8 / 4) * 3 + 5 = 11



`/` bir işleç: önce sol alt ifade hesaplanır.

Adım 3 / 13

```
1  int hesapla(ifade_gost t) {
2    if (!t->sol && !t->sag)        /* yaprak: sayı */
3      return t->deger;
▶    int a = hesapla(t->sol);
5    int b = hesapla(t->sag);
6    switch (t->islec) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

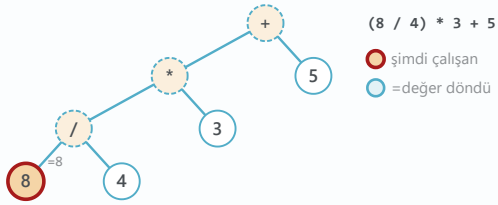
DEĞİŞKENLER

`t = &düğüm(/)` `a = ?` `b = ?`

ÇAĞRI YIĞINI

- `hesapla(/)`
- `hesapla(*)`
- `hesapla(+)`

hesapla(): (8 / 4) * 3 + 5 = 11



`8` bir yaprak: sayıyı döndürür → **8**.


```
1  int hesapla(ifade_gost t) {
2    if (!t->sol && !t->sag)        /* yaprak: sayı */
▶      return t->deger;
4    int a = hesapla(t->sol);
5    int b = hesapla(t->sag);
6    switch (t->islec) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

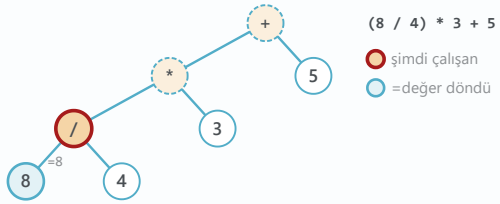
DEĞİŞKENLER

`t = &düğüm(8)` `a = ?` `b = ?`

ÇAĞRI YIĞINI

- `hesapla(8)`
- `hesapla(/)`
- `hesapla(*)`
- `hesapla(+)`

hesapla(): (8 / 4) * 3 + 5 = 11



sol değer a = 8. Şimdi sağ alt ifade hesaplanır.

Adım 5 / 13

```
1  int hesapla(ifade_gost t) {
2    if (!t->sol && !t->sag)        /* yaprak: sayı */
3      return t->deger;
4    int a = hesapla(t->sol);
▶    int b = hesapla(t->sag);
6    switch (t->islec) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

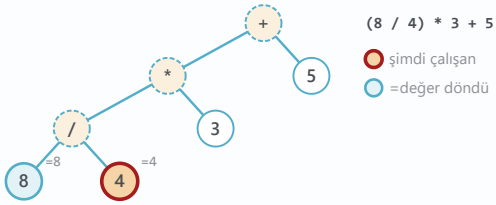
DEĞİŞKENLER

`t = &düğüm(/)` `a = 8` `b = ?`

ÇAĞRI YIĞINI

- `hesapla(/)`
- `hesapla(*)`
- `hesapla(+)`

hesapla(): (8 / 4) * 3 + 5 = 11



`4` bir yaprak: sayıyı döndürür → **4**.


```
1  int hesapla(ifade_gost t) {
2    if (!t->sol && !t->sag)        /* yaprak: sayı */
▶      return t->deger;
4    int a = hesapla(t->sol);
5    int b = hesapla(t->sag);
6    switch (t->islec) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

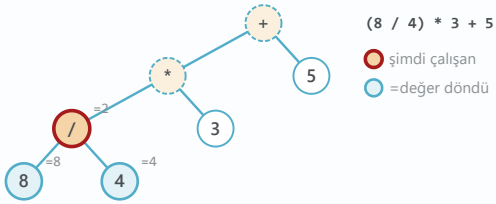
DEĞİŞKENLER

`t = &düğüm(4)` `a = ?` `b = ?`

ÇAĞRI YIĞINI

- `hesapla(4)`
- `hesapla(/)`
- `hesapla(*)`
- `hesapla(+)`

hesapla(): (8 / 4) * 3 + 5 = 11



b = 4; `8 / 4` = **2** döndürülür (kök sonda sırası: sol, sağ, kök).

Adım 7 / 13

```
1  int hesapla(ifade_gost t) {
2    if (!t->sol && !t->sag)        /* yaprak: sayı */
3      return t->deger;
4    int a = hesapla(t->sol);
5    int b = hesapla(t->sag);
6    switch (t->islec) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
▶      default:  return a / b;
11   }
12 }
```

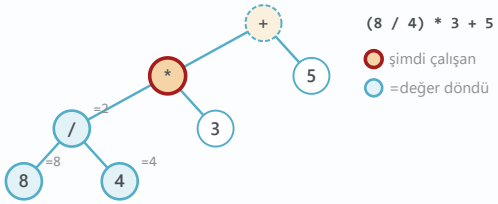
DEĞİŞKENLER

`t = &düğüm(/)` `a = 8` `b = 4`

ÇAĞRI YIĞINI

- `hesapla(/)`
- `hesapla(*)`
- `hesapla(+)`

hesapla(): (8 / 4) * 3 + 5 = 11



sol değer a = 2. Şimdi sağ alt ifade hesaplanır.

Adım 8 / 13

```
1  int hesapla(ifade_gost t) {
2    if (!t->sol && !t->sag)        /* yaprak: sayı */
3      return t->deger;
4    int a = hesapla(t->sol);
▶    int b = hesapla(t->sag);
6    switch (t->islec) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

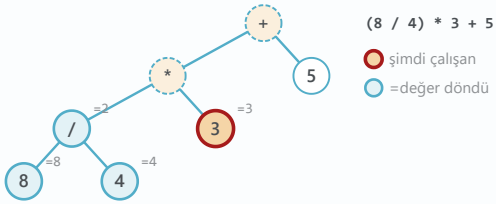
DEĞİŞKENLER

`t = &düğüm(*)` `a = 2` `b = ?`

ÇAĞRI YIĞINI

- `hesapla(*)`
- `hesapla(+)`

hesapla(): (8 / 4) * 3 + 5 = 11



`3` bir yaprak: sayıyı döndürür → **3**.


```
1  int hesapla(ifade_gost t) {
2    if (!t->sol && !t->sag)        /* yaprak: sayı */
▶      return t->deger;
4    int a = hesapla(t->sol);
5    int b = hesapla(t->sag);
6    switch (t->islec) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

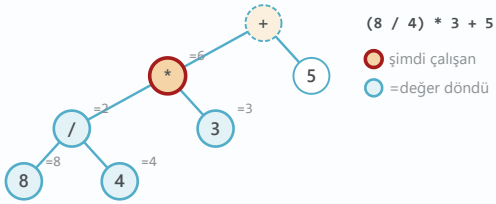
DEĞİŞKENLER

`t = &düğüm(3)` `a = ?` `b = ?`

ÇAĞRI YIĞINI

- `hesapla(3)`
- `hesapla(*)`
- `hesapla(+)`

hesapla(): (8 / 4) * 3 + 5 = 11



b = 3; `2 * 3` = **6** döndürülür (kök sonda sırası: sol, sağ, kök).

Adım 10 / 13

```
1  int hesapla(ifade_gost t) {
2    if (!t->sol && !t->sag)        /* yaprak: sayı */
3      return t->deger;
4    int a = hesapla(t->sol);
5    int b = hesapla(t->sag);
6    switch (t->islec) {
7      case '+': return a + b;
8      case '-': return a - b;
▶      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

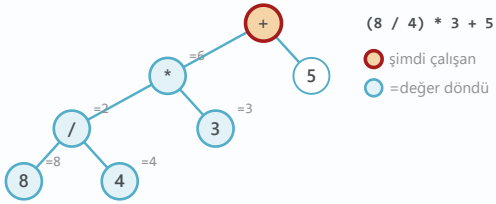
DEĞİŞKENLER

`t = &düğüm(*)` `a = 2` `b = 3`

ÇAĞRI YIĞINI

- `hesapla(*)`
- `hesapla(+)`

hesapla(): (8 / 4) * 3 + 5 = 11



sol değer a = 6. Şimdi sağ alt ifade hesaplanır.

Adım 11 / 13

```
1  int hesapla(ifade_gost t) {
2    if (!t->sol && !t->sag)        /* yaprak: sayı */
3      return t->deger;
4    int a = hesapla(t->sol);
▶    int b = hesapla(t->sag);
6    switch (t->islec) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

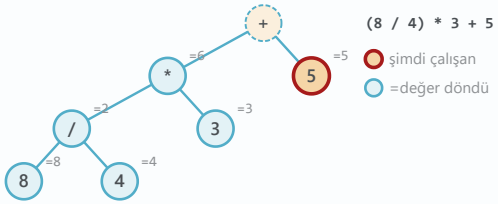
DEĞİŞKENLER

`t = &düğüm(+)` `a = 6` `b = ?`

ÇAĞRI YIĞINI

`hesapla(+)`

hesapla(): (8 / 4) * 3 + 5 = 11



`5` bir yaprak: sayıyı döndürür → **5**.


```
1  int hesapla(ifade_gost t) {
2    if (!t->sol && !t->sag)        /* yaprak: sayı */
▶      return t->deger;
4    int a = hesapla(t->sol);
5    int b = hesapla(t->sag);
6    switch (t->islec) {
7      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

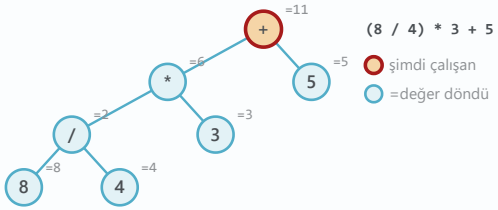
DEĞİŞKENLER

`t = &düğüm(5)` `a = ?` `b = ?`

ÇAĞRI YIĞINI

- `hesapla(5)`
- `hesapla(+)`

hesapla(): (8 / 4) * 3 + 5 = 11



b = 5; `6 + 5` = **11** döndürülür (kök sonda sırası: sol, sağ, kök).

Adım 13 / 13

```
1  int hesapla(ifade_gost t) {
2    if (!t->sol && !t->sag)        /* yaprak: sayı */
3      return t->deger;
4    int a = hesapla(t->sol);
5    int b = hesapla(t->sag);
6    switch (t->islec) {
▶      case '+': return a + b;
8      case '-': return a - b;
9      case '*': return a * b;
10     default:  return a / b;
11   }
12 }
```

DEĞİŞKENLER

`t = &düğüm(+)` `a = 6` `b = 5`

ÇAĞRI YIĞINI

`hesapla(+)`

İteratif gezinme: yığın nereden çıktı?

- **Her özyinelemeli algoritma döngüsel hâle getirilebilir.** İki yol: (1) davranışı döngüyle ifade etmek, (2) işletim sisteminin yaptığı gibi dönüş adreslerini bir yığına atmak.
- Kök ortada: kök basılmadan önce sol alt ağaç basılmalı → son eklenene önce erişilmek istenir: **LIFO**.
- Alan karmaşıklığı artmasın diye yığına öğelerin kopyası değil **adresleri** konur.
- Yığında aynı anda en fazla **sol derinlik** kadar öğe bulunur.

> **HOCADAN** "Öyleyse düğümleri bir sıraya koyduğumuzda son eklediğimize önce erişmek isteyeceğiz. Ya hu bu bir stack yapısı! Ne kadar estetik anlattık!"

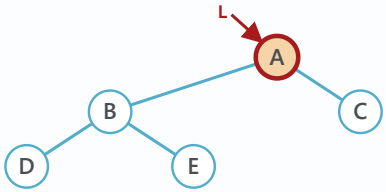
Özyinelemesiz kök ortada gezinme

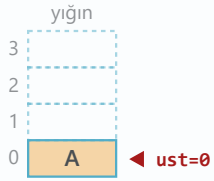
```
void kok_ortada_dolas(agac_gost L) {
  agac_gost yigit[MAX_SIZE];
  int ust = -1;
  for (;;) {
    for (; L; L = L->solcocuk)  // solları at
      yigit_ekle(&ust, L, yigit);
    L = yigit_al(&ust, yigit);  // boşsa NULL
    if (!L) break;
    printf("%c", L->veri);
    L = L->sagcocuk;
  }
}
```

```
void yigit_ekle(int *ust, agac_gost p,
                agac_gost y[]) {
  if (*ust == MAX_SIZE - 1) {
    fprintf(stderr, "Yigit dolu\n");
    exit(EXIT_FAILURE);
  }
  y[++*ust] = p;
}
agac_gost yigit_al(int *ust,
                   agac_gost y[]) {
  return (*ust == -1) ? NULL : y[(*ust)--];
}
```

Yığın boşsa `yigit_al` NULL döndürür; döngünün tek çıkışı budur.

Yığınla kök ortada: A(B(D,E),C)





`A` yığına eklenir (adresi!), sonra sola: L = A->solcocuk = B.

Adım 1 / 16

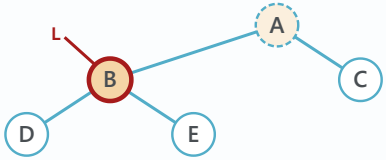
```
1  void kok_ortada_dolas(agac_gost L) {
2    agac_gost yigit[MAX_SIZE];
3    int ust = -1;
4    for (;;) {
5      for (; L; L = L->solcocuk)  // solları at
▶        yigit_ekle(&ust, L, yigit);
7      L = yigit_al(&ust, yigit);  // boşsa NULL
8      if (!L) break;
9      printf("%c", L->veri);
10     L = L->sagcocuk;
11   }
12 }
```

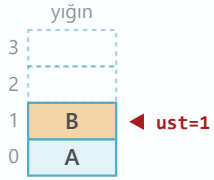
DEĞİŞKENLER

`L = &düğüm(A)` `ust = 0` `yigit = [A]`

ÇIKTI

Yığınla kök ortada: A(B(D,E),C)





`B` yığına eklenir (adresi!), sonra sola: L = B->solcocuk = D.

Adım 2 / 16

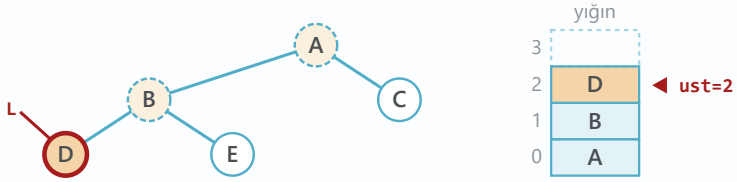
```
1  void kok_ortada_dolas(agac_gost L) {
2    agac_gost yigit[MAX_SIZE];
3    int ust = -1;
4    for (;;) {
5      for (; L; L = L->solcocuk)  // solları at
▶        yigit_ekle(&ust, L, yigit);
7      L = yigit_al(&ust, yigit);  // boşsa NULL
8      if (!L) break;
9      printf("%c", L->veri);
10     L = L->sagcocuk;
11   }
12 }
```

DEĞİŞKENLER

`L = &düğüm(B)` `ust = 1`
`yigit = [A, B]`

ÇIKTI

Yığınla kök ortada: A(B(D,E),C)



`D` yığına eklenir (adresi!), sonra sola: L = D->solcocuk = NULL. L NULL → iç döngü biter.

Adım 3 / 16

```
1  void kok_ortada_dolas(agac_gost L) {
2    agac_gost yigit[MAX_SIZE];
3    int ust = -1;
4    for (;;) {
5      for (; L; L = L->solcocuk)  // solları at
▶        yigit_ekle(&ust, L, yigit);
7      L = yigit_al(&ust, yigit);  // boşsa NULL
8      if (!L) break;
9      printf("%c", L->veri);
10     L = L->sagcocuk;
11   }
12 }
```

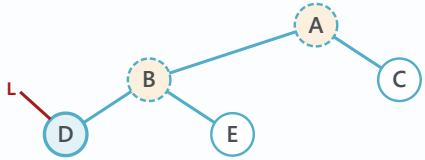
DEĞİŞKENLER

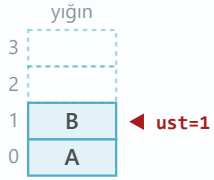
`L = &düğüm(D)` `ust = 2`

`yigit = [A, B, D]`

ÇIKTI

Yığınla kök ortada: A(B(D,E),C)





Tepeden `D` alınır; solu bitmiş demektir → **D** yazılır.


```
1  void kok_ortada_dolas(agac_gost L) {
2    agac_gost yigit[MAX_SIZE];
3    int ust = -1;
4    for (;;) {
5      for (; L; L = L->solcocuk)  // solları at
6        yigit_ekle(&ust, L, yigit);
▶      L = yigit_al(&ust, yigit);  // boşsa NULL
8      if (!L) break;
▶      printf("%c", L->veri);
10     L = L->sagcocuk;
11   }
12 }
```

DEĞİŞKENLER

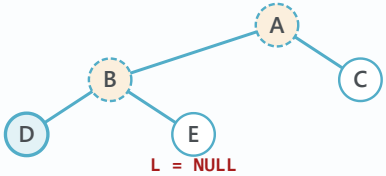
`L = &düğüm(D)` `ust = 1`

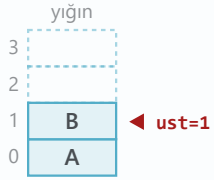
`yigit = [A, B]`

ÇIKTI

```
D
```

Yığınla kök ortada: A(B(D,E),C)





L = D->sagcocuk = NULL: sağ alt ağaç yok, yığından devam.


```
1  void kok_ortada_dolas(agac_gost L) {
2    agac_gost yigit[MAX_SIZE];
3    int ust = -1;
4    for (;;) {
5      for (; L; L = L->solcocuk)  // solları at
6        yigit_ekle(&ust, L, yigit);
7      L = yigit_al(&ust, yigit);  // boşsa NULL
8      if (!L) break;
9      printf("%c", L->veri);
▶      L = L->sagcocuk;
11   }
12 }
```

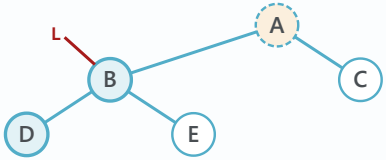
DEĞİŞKENLER

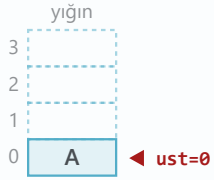
`L = NULL` `ust = 1` `yigit = [A, B]`

ÇIKTI

```
D
```

Yığınla kök ortada: A(B(D,E),C)





Tepeden `B` alınır; solu bitmiş demektir → **B** yazılır.

Adım 6 / 16

```
1  void kok_ortada_dolas(agac_gost L) {
2    agac_gost yigit[MAX_SIZE];
3    int ust = -1;
4    for (;;) {
5      for (; L; L = L->solcocuk)  // solları at
6        yigit_ekle(&ust, L, yigit);
▶      L = yigit_al(&ust, yigit);  // boşsa NULL
8      if (!L) break;
▶      printf("%c", L->veri);
10     L = L->sagcocuk;
11   }
12 }
```

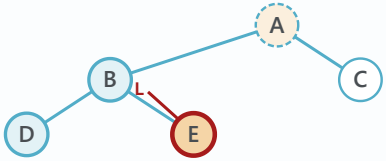
DEĞİŞKENLER

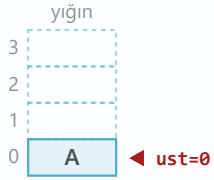
`L = &düğüm(B)` `ust = 0`
`yigit = [A]`

ÇIKTI

```
DB
```

Yığınla kök ortada: A(B(D,E),C)





L = B->sagcocuk = E: sağ alt ağaç aynı biçimde gezilecek.


```
1  void kok_ortada_dolas(agac_gost L) {
2    agac_gost yigit[MAX_SIZE];
3    int ust = -1;
4    for (;;) {
5      for (; L; L = L->solcocuk)  // solları at
6        yigit_ekle(&ust, L, yigit);
7      L = yigit_al(&ust, yigit);  // boşsa NULL
8      if (!L) break;
9      printf("%c", L->veri);
▶      L = L->sagcocuk;
11   }
12 }
```

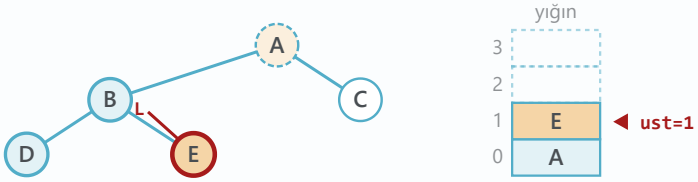
DEĞİŞKENLER

`L = &düğüm(E)` `ust = 0` `yigit = [A]`

ÇIKTI

```
DB
```

Yığınla kök ortada: A(B(D,E),C)



`E` yığına eklenir (adresi!), sonra sola: L = E->solcocuk = NULL. L NULL → iç döngü biter.


```
1  void kok_ortada_dolas(agac_gost L) {
2    agac_gost yigit[MAX_SIZE];
3    int ust = -1;
4    for (;;) {
5      for (; L; L = L->solcocuk)  // solları at
▶        yigit_ekle(&ust, L, yigit);
7      L = yigit_al(&ust, yigit);  // boşsa NULL
8      if (!L) break;
9      printf("%c", L->veri);
10     L = L->sagcocuk;
11   }
12 }
```

DEĞİŞKENLER

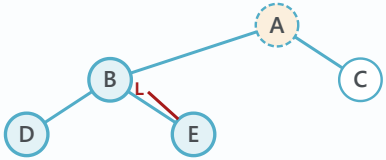
`L = &düğüm(E)` `ust = 1`

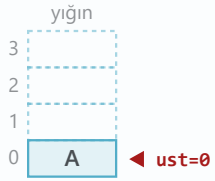
`yigit = [A, E]`

ÇIKTI

```
DB
```

Yığınla kök ortada: A(B(D,E),C)





Tepeden `E` alınır; solu bitmiş demektir → **E** yazılır.


```
1  void kok_ortada_dolas(agac_gost L) {
2    agac_gost yigit[MAX_SIZE];
3    int ust = -1;
4    for (;;) {
5      for (; L; L = L->solcocuk)  // solları at
6        yigit_ekle(&ust, L, yigit);
▶      L = yigit_al(&ust, yigit);  // boşsa NULL
8      if (!L) break;
▶      printf("%c", L->veri);
10     L = L->sagcocuk;
11   }
12 }
```

DEĞİŞKENLER

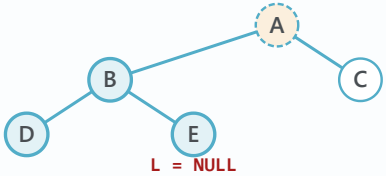
`L = &düğüm(E)` `ust = 0`

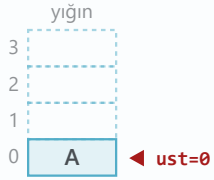
`yigit = [A]`

ÇIKTI

```
DBE
```

Yığınla kök ortada: A(B(D,E),C)





L = E->sagcocuk = NULL: sağ alt ağaç yok, yığından devam.


```
1  void kok_ortada_dolas(agac_gost L) {
2    agac_gost yigit[MAX_SIZE];
3    int ust = -1;
4    for (;;) {
5      for (; L; L = L->solcocuk)  // solları at
6        yigit_ekle(&ust, L, yigit);
7      L = yigit_al(&ust, yigit);  // boşsa NULL
8      if (!L) break;
9      printf("%c", L->veri);
▶      L = L->sagcocuk;
11   }
12 }
```

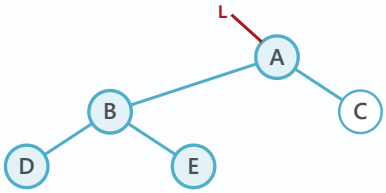
DEĞİŞKENLER

`L = NULL` `ust = 0` `yigit = [A]`

ÇIKTI

```
DBE
```

Yığınla kök ortada: A(B(D,E),C)





Tepeden `A` alınır; solu bitmiş demektir → **A** yazılır.


```
1  void kok_ortada_dolas(agac_gost L) {
2    agac_gost yigit[MAX_SIZE];
3    int ust = -1;
4    for (;;) {
5      for (; L; L = L->solcocuk)  // solları at
6        yigit_ekle(&ust, L, yigit);
▶      L = yigit_al(&ust, yigit);  // boşsa NULL
8      if (!L) break;
▶      printf("%c", L->veri);
10     L = L->sagcocuk;
11   }
12 }
```

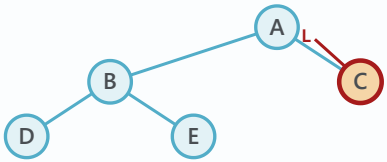
DEĞİŞKENLER

`L = &düğüm(A)` `ust = -1`
`yigit = []`

ÇIKTI

```
DBEA
```

Yığınla kök ortada: A(B(D,E),C)





L = A->sagcocuk = C: sağ alt ağaç aynı biçimde gezilecek.

Adım 12 / 16

```
1  void kok_ortada_dolas(agac_gost L) {
2    agac_gost yigit[MAX_SIZE];
3    int ust = -1;
4    for (;;) {
5      for (; L; L = L->solcocuk)  // solları at
6        yigit_ekle(&ust, L, yigit);
7      L = yigit_al(&ust, yigit);  // boşsa NULL
8      if (!L) break;
9      printf("%c", L->veri);
▶      L = L->sagcocuk;
11   }
12 }
```

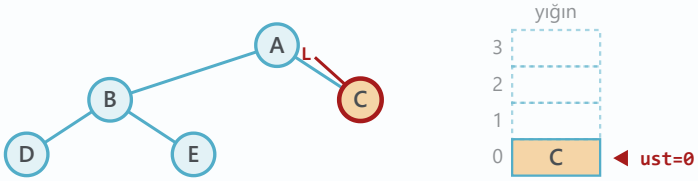
DEĞİŞKENLER

`L = &düğüm(C)` `ust = -1` `yigit = []`

ÇIKTI

```
DBEA
```

Yığınla kök ortada: A(B(D,E),C)



`C` yığına eklenir (adresi!), sonra sola: L = C->solcocuk = NULL. L NULL → iç döngü biter.


```
1  void kok_ortada_dolas(agac_gost L) {
2    agac_gost yigit[MAX_SIZE];
3    int ust = -1;
4    for (;;) {
5      for (; L; L = L->solcocuk)  // solları at
▶        yigit_ekle(&ust, L, yigit);
7      L = yigit_al(&ust, yigit);  // boşsa NULL
8      if (!L) break;
9      printf("%c", L->veri);
10     L = L->sagcocuk;
11   }
12 }
```

DEĞİŞKENLER

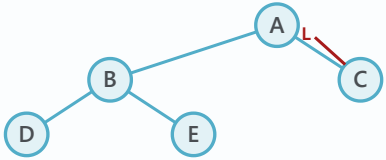
`L = &düğüm(C)` `ust = 0`

`yigit = [C]`

ÇIKTI

```
DBEA
```

Yığınla kök ortada: A(B(D,E),C)





Tepeden `C` alınır; solu bitmiş demektir → **C** yazılır.


```
1  void kok_ortada_dolas(agac_gost L) {
2    agac_gost yigit[MAX_SIZE];
3    int ust = -1;
4    for (;;) {
5      for (; L; L = L->solcocuk)  // solları at
6        yigit_ekle(&ust, L, yigit);
▶      L = yigit_al(&ust, yigit);  // boşsa NULL
8      if (!L) break;
▶      printf("%c", L->veri);
10     L = L->sagcocuk;
11   }
12 }
```

DEĞİŞKENLER

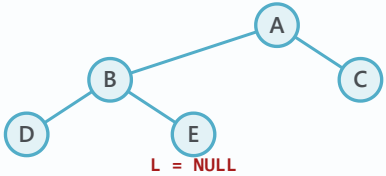
`L = &düğüm(C)` `ust = -1`

`yigit = []`

ÇIKTI

```
DBEAC
```

Yığınla kök ortada: A(B(D,E),C)





L = C->sagcocuk = NULL: sağ alt ağaç yok, yığından devam.


```
1  void kok_ortada_dolas(agac_gost L) {
2    agac_gost yigit[MAX_SIZE];
3    int ust = -1;
4    for (;;) {
5      for (; L; L = L->solcocuk)  // solları at
6        yigit_ekle(&ust, L, yigit);
7      L = yigit_al(&ust, yigit);  // boşsa NULL
8      if (!L) break;
9      printf("%c", L->veri);
▶      L = L->sagcocuk;
11   }
12 }
```

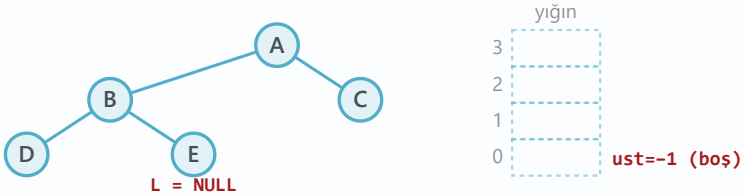
DEĞİŞKENLER

`L = NULL` `ust = -1` `yigit = []`

ÇIKTI

```
DBEAC
```

Yığınla kök ortada: A(B(D,E),C)



Yığın boş: `yigit_al` NULL döndürür → `break`. Gezinme bitti.

Adım 16 / 16

```
1  void kok_ortada_dolas(agac_gost L) {
2    agac_gost yigit[MAX_SIZE];
3    int ust = -1;
4    for (;;) {
5      for (; L; L = L->solcocuk)  // solları at
6        yigit_ekle(&ust, L, yigit);
▶      L = yigit_al(&ust, yigit);  // boşsa NULL
▶      if (!L) break;
9      printf("%c", L->veri);
10     L = L->sagcocuk;
11   }
12 }
```

DEĞİŞKENLER

`L = NULL` `ust = -1` `yigit = []`

ÇIKTI

```
DBEAC
```

Maliyet ve seçici yazdırma

- Her düğüm yığına bir kez eklenir, bir kez alınır: 2n adım → **O(n)** zaman.
- Bellek: yığın en kötü (çarpık) durumda **O(h)**; sağa çarpık ağaçta yığında hep ≤ 1 öğe → O(1).
- 9. satıra (printf) bir koşul konursa yalnızca seçilen düğümleri yazan yordam elde edilir: uç düğümler, tek çocuklular, sol çocuklular…
- `if(!(L→sagcocuk)||!(L→solcocuk))` koşulu **en çok bir çocuklu** düğümleri yazar; yalnız yapraklar için `&&` gerekir.

HOCADAN "Bizim bu dersteki amacımız, büyük otomasyon sistemleri vs. geliştirmek değil, küçük modülleri ustaca yazabilmektir. Küçük kodlar defalarca farklı şekillerde yazılmalı."

"Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it."

— Alan J. Perlis

Sorular: gezinme kalıbını değiştirerek yaz

1. Kök önde gezinme yordamını **döngüsel** yazınız.
2. Kök sonda gezinme yordamını döngüsel yazınız. *(Bu daha zordur.)*
3. Uç düğümleri soldan sağa yazan **özyinelemeli** yordam.
4. Uç düğümleri sağdan sola yazan özyinelemeli yordam.
5. Uç düğümleri soldan sağa yazan döngüsel yordam.
6. Uç düğümleri sağdan sola yazan döngüsel yordam.
7. Yalnız sol çocuğu olan düğümleri yazan özyinelemeli yordam.
8. Yalnız sağ çocuğu olan düğümleri yazan özyinelemeli yordam.
9. Her iki çocuğu da olan düğümleri yazan özyinelemeli yordam.

Hepsi iki kalıptan birine indirgenir: **özyinelemeli gezinme + koşullu printf** ya da **yığınlı gezinme + koşullu printf**. "Sağdan sola" için `solcocuk` ile `sagcocuk` yer değiştirir.

SORU 1: döngüsel kök önde

```
void kok_onde_dolas_d(agac_gost L) {
  agac_gost yigit[MAX_SIZE];
  int ust = -1;
  for (;;) {
    for (; L; L = L->solcocuk) {
      printf("%c", L->veri);   // girerken yaz
      yigit_ekle(&ust, L, yigit);
    }
    L = yigit_al(&ust, yigit);
    if (!L) break;
    L = L->sagcocuk;
  }
}
```

- İteratif kök ortada kodunun **aynısı**; yalnızca `printf` düğüm yığına girerken yapılır.
- Böylece kök, sol alt ağacından önce yazılır: V-L-R.
- Alternatif: kök yığına atılır; her adımda alınır, yazılır, önce **sağ** sonra **sol** çocuk atılır.

▶ **İkinci çözümü göster**

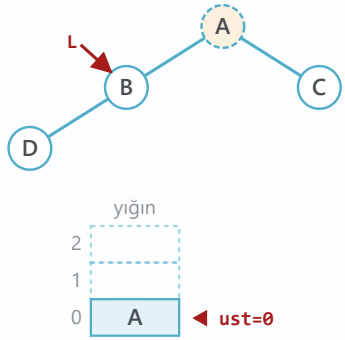
SORU 2: döngüsel kök sonda — neden daha zor?

- Bir düğüm yığından alınmadan önce **iki kez** tepeye gelir: sol alt ağaçtan dönünce ve sağ alt ağaçtan dönünce.
- Hangisinden döndüğümüzü bilmek için **son yazılan düğümü** (`son`) tutarız.
- Tepedeki t'nin sağ çocuğu varsa ve son yazılan o değilse → önce sağa inilir; değilse t yazılır ve çıkarılır.
- Alternatif: iki yığın (V-R-L sırasını ikinci yığında tersine çevirmek).

Zaman O(n): her düğüm bir kez eklenir, bir kez çıkarılır, en çok iki kez tepeye bakılır. Bellek O(h).

SORU 2 adım adım: A(B(D),C)

```
1  void kok_sonda_dolas_d(agac_gost L) {
2    agac_gost yigit[MAX_SIZE], son = NULL, t;
3    int ust = -1;
4    while (L || ust != -1) {
5      if (L) {                        // sola in
▶        yigit_ekle(&ust, L, yigit);
▶        L = L->solcocuk;
8      } else {
9        t = yigit[ust];               // tepeye bak
10       if (t->sagcocuk && t->sagcocuk != son)
11         L = t->sagcocuk;            // önce sağ
12       else {
13         printf("%c", t->veri);
14         son = yigit_al(&ust, yigit);
15       }
16     }
17   }
18 }
```



L = A yığına atılır, sola inilir: L = B.

Adım 1 / 10

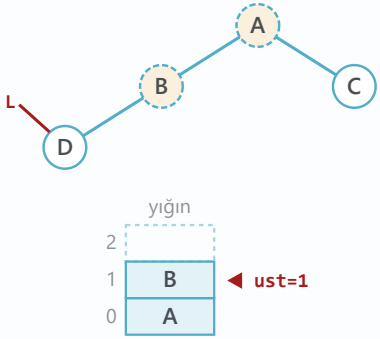
DEĞİŞKENLER

`L = &düğüm(B)` `t = NULL` `son = NULL` `yigit = [A]`

ÇIKTI

SORU 2 adım adım: A(B(D),C)

```
1  void kok_sonda_dolas_d(agac_gost L) {
2    agac_gost yigit[MAX_SIZE], son = NULL, t;
3    int ust = -1;
4    while (L || ust != -1) {
5      if (L) {                        // sola in
▶        yigit_ekle(&ust, L, yigit);
▶        L = L->solcocuk;
8      } else {
9        t = yigit[ust];               // tepeye bak
10       if (t->sagcocuk && t->sagcocuk != son)
11         L = t->sagcocuk;            // önce sağ
12       else {
13         printf("%c", t->veri);
14         son = yigit_al(&ust, yigit);
15       }
16     }
17   }
18 }
```



L = B yığına atılır, sola inilir: L = D.

Adım 2 / 10

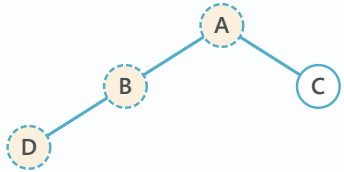
DEĞİŞKENLER

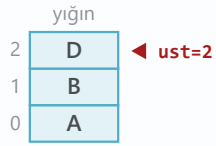
`L = &düğüm(D)` `t = NULL` `son = NULL` `yigit = [A, B]`

ÇIKTI

SORU 2 adım adım: A(B(D),C)

```
1  void kok_sonda_dolas_d(agac_gost L) {
2    agac_gost yigit[MAX_SIZE], son = NULL, t;
3    int ust = -1;
4    while (L || ust != -1) {
5      if (L) {                        // sola in
▶        yigit_ekle(&ust, L, yigit);
▶        L = L->solcocuk;
8      } else {
9        t = yigit[ust];               // tepeye bak
10       if (t->sagcocuk && t->sagcocuk != son)
11         L = t->sagcocuk;            // önce sağ
12       else {
13         printf("%c", t->veri);
14         son = yigit_al(&ust, yigit);
15       }
16     }
17   }
18 }
```





L = D yığına atılır, sola inilir: L = NULL.

Adım 3 / 10

DEĞİŞKENLER

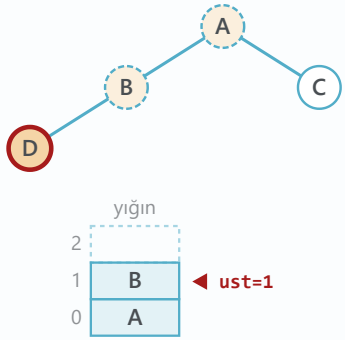
`L = NULL` `t = NULL` `son = NULL`

`yigit = [A, B, D]`

ÇIKTI

SORU 2 adım adım: A(B(D),C)

```
1  void kok_sonda_dolas_d(agac_gost L) {
2    agac_gost yigit[MAX_SIZE], son = NULL, t;
3    int ust = -1;
4    while (L || ust != -1) {
5      if (L) {                        // sola in
6        yigit_ekle(&ust, L, yigit);
7        L = L->solcocuk;
8      } else {
▶        t = yigit[ust];               // tepeye bak
10       if (t->sagcocuk && t->sagcocuk != son)
11         L = t->sagcocuk;            // önce sağ
12       else {
▶          printf("%c", t->veri);
▶          son = yigit_al(&ust, yigit);
15       }
16     }
17   }
18 }
```



Tepe D: sağ çocuğu yok → **D** yazılır, yığından alınır; son = D.

Adım 4 / 10

DEĞİŞKENLER

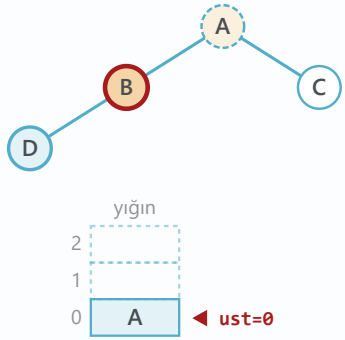
`L = NULL` `t = &düğüm(D)` `son = &düğüm(D)` `yigit = [A, B]`

ÇIKTI

```
D
```

SORU 2 adım adım: A(B(D),C)

```
1  void kok_sonda_dolas_d(agac_gost L) {
2    agac_gost yigit[MAX_SIZE], son = NULL, t;
3    int ust = -1;
4    while (L || ust != -1) {
5      if (L) {                      // sola in
6        yigit_ekle(&ust, L, yigit);
7        L = L->solcocuk;
8      } else {
▶        t = yigit[ust];             // tepeye bak
10       if (t->sagcocuk && t->sagcocuk != son)
11         L = t->sagcocuk;          // önce sağ
12       else {
▶          printf("%c", t->veri);
▶          son = yigit_al(&ust, yigit);
15       }
16     }
17   }
18 }
```



Tepe B: sağ çocuğu yok → **B** yazılır, yığından alınır; son = B.

Adım 5 / 10

DEĞİŞKENLER

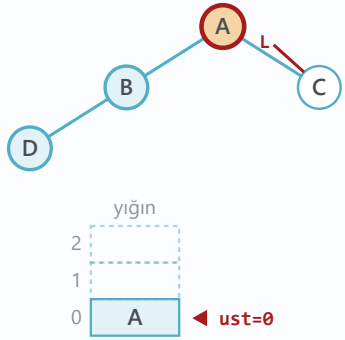
`L = NULL` `t = &düğüm(B)` `son = &düğüm(B)` `yigit = [A]`

ÇIKTI

```
DB
```

SORU 2 adım adım: A(B(D),C)

```
1  void kok_sonda_dolas_d(agac_gost L) {
2    agac_gost yigit[MAX_SIZE], son = NULL, t;
3    int ust = -1;
4    while (L || ust != -1) {
5      if (L) {                        // sola in
6        yigit_ekle(&ust, L, yigit);
7        L = L->solcocuk;
8      } else {
▶        t = yigit[ust];               // tepeye bak
▶        if (t->sagcocuk && t->sagcocuk != son)
▶          L = t->sagcocuk;            // önce sağ
12       else {
13         printf("%c", t->veri);
14         son = yigit_al(&ust, yigit);
15       }
16     }
17   }
18 }
```



Tepe A: sağ çocuğu C var ve henüz yazılmadı → önce sağ alt ağaç (L = C).

Adım 6 / 10

DEĞİŞKENLER

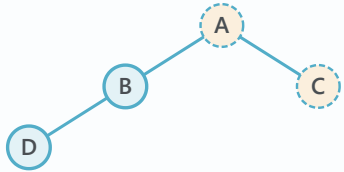
`L = &düğüm(C)` `t = &düğüm(A)` `son = &düğüm(B)` `yigit = [A]`

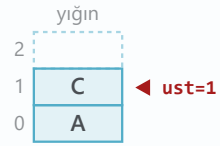
ÇIKTI

```
DB
```

SORU 2 adım adım: A(B(D),C)

```
1  void kok_sonda_dolas_d(agac_gost L) {
2    agac_gost yigit[MAX_SIZE], son = NULL, t;
3    int ust = -1;
4    while (L || ust != -1) {
5      if (L) {                        // sola in
▶        yigit_ekle(&ust, L, yigit);
▶        L = L->solcocuk;
8      } else {
9        t = yigit[ust];               // tepeye bak
10       if (t->sagcocuk && t->sagcocuk != son)
11         L = t->sagcocuk;            // önce sağ
12       else {
13         printf("%c", t->veri);
14         son = yigit_al(&ust, yigit);
15       }
16     }
17   }
18 }
```





L = C yığına atılır, sola inilir: L = NULL.

Adım 7 / 10

DEĞİŞKENLER

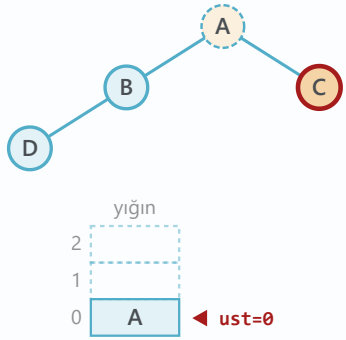
`L = NULL` `t = NULL` `son = &düğüm(B)` `yigit = [A, C]`

ÇIKTI

```
DB
```

SORU 2 adım adım: A(B(D),C)

```
1  void kok_sonda_dolas_d(agac_gost L) {
2    agac_gost yigit[MAX_SIZE], son = NULL, t;
3    int ust = -1;
4    while (L || ust != -1) {
5      if (L) {                        // sola in
6        yigit_ekle(&ust, L, yigit);
7        L = L->solcocuk;
8      } else {
▶        t = yigit[ust];               // tepeye bak
10       if (t->sagcocuk && t->sagcocuk != son)
11         L = t->sagcocuk;            // önce sağ
12       else {
▶          printf("%c", t->veri);
▶          son = yigit_al(&ust, yigit);
15       }
16     }
17   }
18 }
```



Tepe C: sağ çocuğu yok → **C** yazılır, yığından alınır; son = C.

Adım 8 / 10

DEĞİŞKENLER

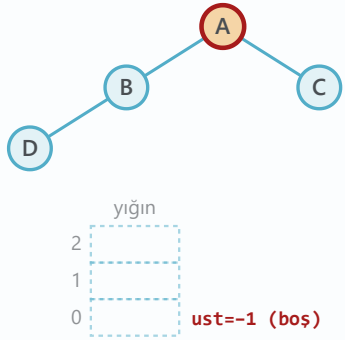
`L = NULL` `t = &düğüm(C)` `son = &düğüm(C)` `yigit = [A]`

ÇIKTI

```
DBC
```

SORU 2 adım adım: A(B(D),C)

```
1  void kok_sonda_dolas_d(agac_gost L) {
2    agac_gost yigit[MAX_SIZE], son = NULL, t;
3    int ust = -1;
4    while (L || ust != -1) {
5      if (L) {                      // sola in
6        yigit_ekle(&ust, L, yigit);
7        L = L->solcocuk;
8      } else {
▶9       t = yigit[ust];             // tepeye bak
10       if (t->sagcocuk && t->sagcocuk != son)
11         L = t->sagcocuk;          // önce sağ
12       else {
▶          printf("%c", t->veri);
▶          son = yigit_al(&ust, yigit);
15       }
16     }
17   }
18 }
```



Tepe A: sağ çocuğu C az önce yazıldı (son) → **A** yazılır, yığından alınır; son = A.

Adım 9 / 10

DEĞİŞKENLER

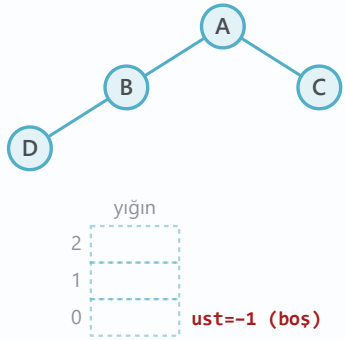
`L = NULL` `t = &düğüm(A)` `son = &düğüm(A)` `yigit = []`

ÇIKTI

```
DBCA
```

SORU 2 adım adım: A(B(D),C)

```
1  void kok_sonda_dolas_d(agac_gost L) {
2    agac_gost yigit[MAX_SIZE], son = NULL, t;
3    int ust = -1;
▶    while (L || ust != -1) {
5      if (L) {                      // sola in
6        yigit_ekle(&ust, L, yigit);
7        L = L->solcocuk;
8      } else {
9        t = yigit[ust];             // tepeye bak
10       if (t->sagcocuk && t->sagcocuk != son)
11         L = t->sagcocuk;         // önce sağ
12       else {
13         printf("%c", t->veri);
14         son = yigit_al(&ust, yigit);
15       }
16     }
17   }
18 }
```



L NULL ve yığın boş: döngü biter. Kök en sona kaldı: kök sonda sırası.

Adım 10 / 10

DEĞİŞKENLER

`L = NULL` `t = NULL` `son = &düğüm(A)` `yigit = []`

ÇIKTI

```
DBCA
```

SORU 3–6: uç düğümleri yazmak

```
void uc_dugum_yaz(agac_gost L) {
  if (L) {
    uc_dugum_yaz(L->solcocuk);
    if (!L->solcocuk && !L->sagcocuk)
      printf("%c", L->veri);
    uc_dugum_yaz(L->sagcocuk);
  }
}
```

SORU 3'ün çözümü. Örnek ağaçta (A…L) çıktı: `GHJKL`.

▶ **SORU 4: sağdan sola (özyinelemeli)**

▶ **SORU 5: soldan sağa (döngüsel)**

▶ **SORU 6: sağdan sola (döngüsel)**

SORU 7–9: çocuk sayısına göre seçmek

```c
void yalniz_sol_yaz(agac_gost L) {          /* SORU 7 */
  if (!L) return;
  if (L->solcocuk && !L->sagcocuk)
    printf("%c", L->veri);
  yalniz_sol_yaz(L->solcocuk);
  yalniz_sol_yaz(L->sagcocuk);
}
```

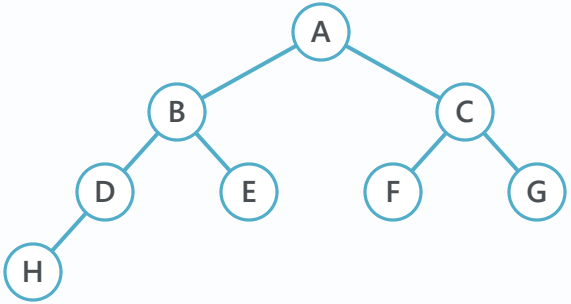
▶ **SORU 8: yalnız sağ çocuklu**

▶ **SORU 9: iki çocuklu**

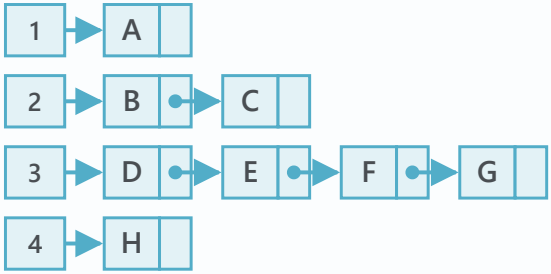
Ortak kalıp: “ağacı gez, **koşul** sağlanıyorsa yaz”. `!L→solcocuk` = “sol çocuğu yok”.

Örnek ağaçta yalnız sol çocuklu: `F`, yalnız sağ çocuklu: `BE`.

SORU: düğümleri düzey düzey yazdırmak — I. yöntem



kök ortada: HDBEAFCG · kök önde: ABDHECFG · kök sonda: HDEBFGCA



- Ağaç (herhangi bir gezinmeyle) gezilir; her düğüm düzeyinin listesinin sonuna eklenir; sonra listeler sırayla yazılır.
- Her öğe 3 kez ele alınır → 3n = **O(n)** zaman; bellek **O(n + h)** ($\lg n \le h \le n$).

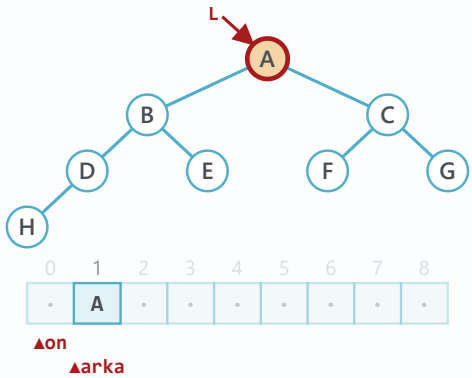
II. yöntem: kuyrukla düzey düzey gezinme

- Düzey düzey yazmak için **önce gelen önce çıkmalı**: bir düğümün çocukları, aynı düzeydeki diğer düğümlerden sonra yazılır → **kuyruk** (FIFO).
- Kök kuyruğa konur; her adımda önden bir düğüm alınır, yazılır, çocukları arkaya eklenir.
- Kuyruk boşalınca (`kuyruk_al` NULL) gezinme biter. Ağaç boş gelebilir: önce `if (!L) return;`
- Kuyruğa da düğümlerin **adresleri** konur (dairesel kuyruk).

```
void kuyruk_ekle(int on, int *arka,
                 agac_gost p, agac_gost k[]) {
  *arka = (*arka + 1) % MAX_SIZE;
  if (*arka == on) {            /* dolu */
    fprintf(stderr, "Kuyruk dolu\n");
    exit(EXIT_FAILURE);
  }
  k[*arka] = p;
}
agac_gost kuyruk_al(int *on, int arka,
                    agac_gost k[]) {
  if (*on == arka) return NULL; /* boş */
  *on = (*on + 1) % MAX_SIZE;
  return k[*on];
}
```

Kuyrukla düzey düzey: A B C D E F G H

```
1  void duzey_dolas(agac_gost L) {
2    agac_gost kuyruk[MAX_SIZE];
3    int on = 0, arka = 0;
4    if (!L) return;              // ağaç boş olabilir
▶    kuyruk_ekle(on, &arka, L, kuyruk);
6    while ((L = kuyruk_al(&on, arka, kuyruk))) {
7      printf("%c", L->veri);
8      if (L->solcocuk)
9        kuyruk_ekle(on, &arka, L->solcocuk, kuyruk);
10     if (L->sagcocuk)
11       kuyruk_ekle(on, &arka, L->sagcocuk, kuyruk);
12   }
13 }
```



Kök kuyruğa eklenir (arka = 1). Dairesel kuyruk: `on` ilk öğenin bir öncesini gösterir.

Adım 1 / 14

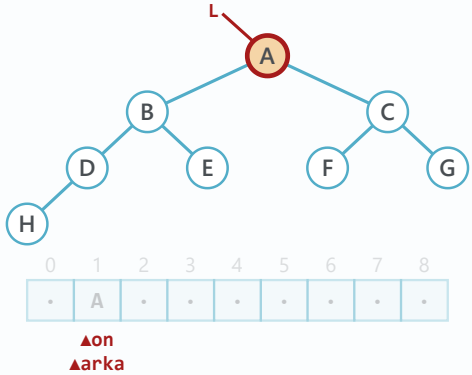
DEĞİŞKENLER

`L = &düğüm(A)` `on = 0` `arka = 1`

ÇIKTI

Kuyrukla düzey düzey: A B C D E F G H

```
1  void duzey_dolas(agac_gost L) {
2    agac_gost kuyruk[MAX_SIZE];
3    int on = 0, arka = 0;
4    if (!L) return;              // ağaç boş
   olabilir
5    kuyruk_ekle(on, &arka, L, kuyruk);
▶    while ((L = kuyruk_al(&on, arka, kuyruk))) {
▶      printf("%c", L->veri);
8      if (L->solcocuk)
9        kuyruk_ekle(on, &arka, L->solcocuk, kuyruk);
10     if (L->sagcocuk)
11       kuyruk_ekle(on, &arka, L->sagcocuk, kuyruk);
12   }
13 }
```



Kuyruğun önünden `A` alınır ve yazılır (FIFO: önce gelen önce çıkar).

Adım 2 / 14

DEĞİŞKENLER

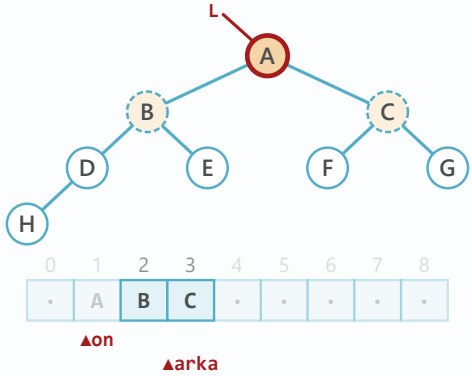
`L = &düğüm(A)` `on = 1` `arka = 1`

ÇIKTI

```
A
```

Kuyrukla düzey düzey: A B C D E F G H

```
1  void duzey_dolas(agac_gost L) {
2    agac_gost kuyruk[MAX_SIZE];
3    int on = 0, arka = 0;
4    if (!L) return;              // ağaç boş olabilir
5    kuyruk_ekle(on, &arka, L, kuyruk);
6    while ((L = kuyruk_al(&on, arka, kuyruk))) {
7      printf("%c", L->veri);
8      if (L->solcocuk)
▶        kuyruk_ekle(on, &arka, L->solcocuk, kuyruk);
10     if (L->sagcocuk)
▶        kuyruk_ekle(on, &arka, L->sagcocuk, kuyruk);
12   }
13 }
```



A'nın çocukları (B, C) kuyruğun arkasına eklenir: bir sonraki düzey sırada bekler.

Adım 3 / 14

DEĞİŞKENLER

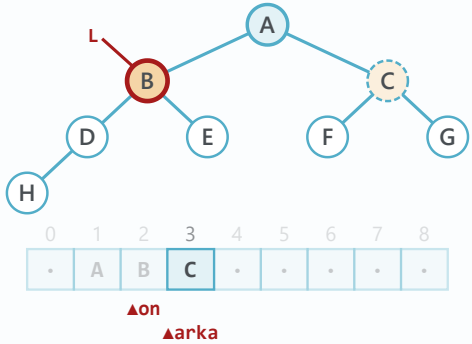
`L = &düğüm(A)` `on = 1` `arka = 3`

ÇIKTI

```
A
```

Kuyrukla düzey düzey: A B C D E F G H

```
1  void duzey_dolas(agac_gost L) {
2    agac_gost kuyruk[MAX_SIZE];
3    int on = 0, arka = 0;
4    if (!L) return;              // ağaç boş olabilir
5    kuyruk_ekle(on, &arka, L, kuyruk);
▶    while ((L = kuyruk_al(&on, arka, kuyruk))) {
▶      printf("%c", L->veri);
8      if (L->solcocuk)
9        kuyruk_ekle(on, &arka, L->solcocuk, kuyruk);
10     if (L->sagcocuk)
11       kuyruk_ekle(on, &arka, L->sagcocuk, kuyruk);
12   }
13 }
```



Kuyruğun önünden `B` alınır ve yazılır (FIFO: önce gelen önce çıkar).

Adım 4 / 14

DEĞİŞKENLER

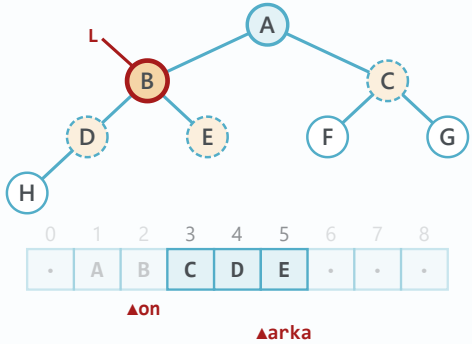
`L = &düğüm(B)` `on = 2` `arka = 3`

ÇIKTI

```
AB
```

Kuyrukla düzey düzey: A B C D E F G H

```
1  void duzey_dolas(agac_gost L) {
2    agac_gost kuyruk[MAX_SIZE];
3    int on = 0, arka = 0;
4    if (!L) return;              // ağaç boş
   olabilir
5    kuyruk_ekle(on, &arka, L, kuyruk);
6    while ((L = kuyruk_al(&on, arka, kuyruk))) {
7      printf("%c", L->veri);
8      if (L->solcocuk)
▶        kuyruk_ekle(on, &arka, L->solcocuk, kuyruk);
10     if (L->sagcocuk)
▶        kuyruk_ekle(on, &arka, L->sagcocuk, kuyruk);
12   }
13 }
```



B'nin çocukları (D, E) kuyruğun arkasına eklenir: bir sonraki düzey sırada bekler.

Adım 5 / 14

DEĞİŞKENLER

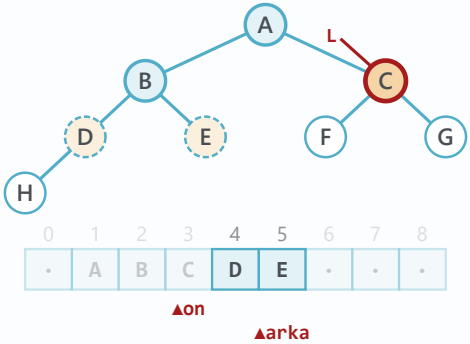
`L = &düğüm(B)` `on = 2` `arka = 5`

ÇIKTI

```
AB
```

Kuyrukla düzey düzey: A B C D E F G H

```
1  void duzey_dolas(agac_gost L) {
2    agac_gost kuyruk[MAX_SIZE];
3    int on = 0, arka = 0;
4    if (!L) return;              // ağaç boş
   olabilir
5    kuyruk_ekle(on, &arka, L, kuyruk);
▶    while ((L = kuyruk_al(&on, arka, kuyruk))) {
▶      printf("%c", L->veri);
8      if (L->solcocuk)
9        kuyruk_ekle(on, &arka, L->solcocuk, kuyruk);
10     if (L->sagcocuk)
11       kuyruk_ekle(on, &arka, L->sagcocuk, kuyruk);
12   }
13 }
```



Kuyruğun önünden `C` alınır ve yazılır (FIFO: önce gelen önce çıkar).

Adım 6 / 14

DEĞİŞKENLER

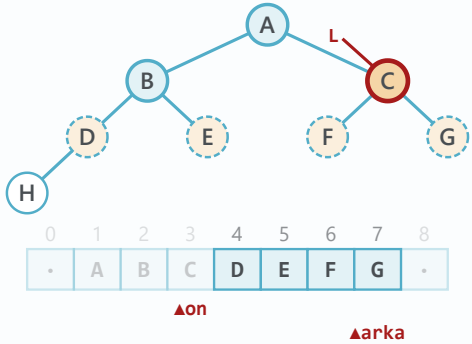
`L = &düğüm(C)` `on = 3` `arka = 5`

ÇIKTI

```
ABC
```

Kuyrukla düzey düzey: A B C D E F G H

```
1  void duzey_dolas(agac_gost L) {
2    agac_gost kuyruk[MAX_SIZE];
3    int on = 0, arka = 0;
4    if (!L) return;              // ağaç boş
   olabilir
5    kuyruk_ekle(on, &arka, L, kuyruk);
6    while ((L = kuyruk_al(&on, arka, kuyruk))) {
7      printf("%c", L->veri);
8      if (L->solcocuk)
▶        kuyruk_ekle(on, &arka, L->solcocuk, kuyruk);
10     if (L->sagcocuk)
▶        kuyruk_ekle(on, &arka, L->sagcocuk, kuyruk);
12   }
13 }
```



C'nin çocukları (F, G) kuyruğun arkasına eklenir: bir sonraki düzey sırada bekler.

Adım 7 / 14

DEĞİŞKENLER

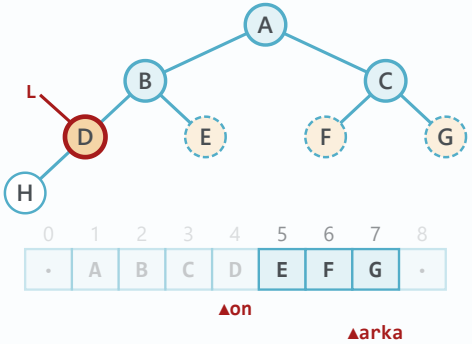
`L = &düğüm(C)` `on = 3` `arka = 7`

ÇIKTI

```
ABC
```

Kuyrukla düzey düzey: A B C D E F G H

```
1  void duzey_dolas(agac_gost L) {
2    agac_gost kuyruk[MAX_SIZE];
3    int on = 0, arka = 0;
4    if (!L) return;              // ağaç boş olabilir
5    kuyruk_ekle(on, &arka, L, kuyruk);
▶    while ((L = kuyruk_al(&on, arka, kuyruk))) {
▶      printf("%c", L->veri);
8      if (L->solcocuk)
9        kuyruk_ekle(on, &arka, L->solcocuk, kuyruk);
10     if (L->sagcocuk)
11       kuyruk_ekle(on, &arka, L->sagcocuk, kuyruk);
12   }
13 }
```



Kuyruğun önünden `D` alınır ve yazılır (FIFO: önce gelen önce çıkar).

Adım 8 / 14

DEĞİŞKENLER

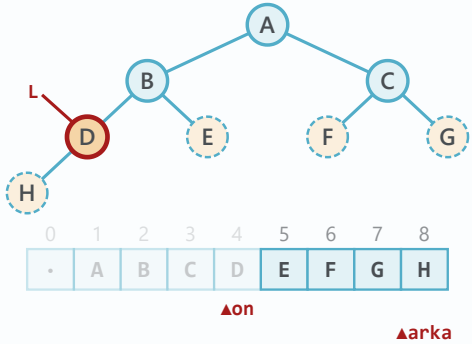
`L = &düğüm(D)` `on = 4` `arka = 7`

ÇIKTI

```
ABCD
```

Kuyrukla düzey düzey: A B C D E F G H

```
1  void duzey_dolas(agac_gost L) {
2    agac_gost kuyruk[MAX_SIZE];
3    int on = 0, arka = 0;
4    if (!L) return;              // ağaç boş
   olabilir
5    kuyruk_ekle(on, &arka, L, kuyruk);
6    while ((L = kuyruk_al(&on, arka, kuyruk))) {
7      printf("%c", L->veri);
8      if (L->solcocuk)
▶        kuyruk_ekle(on, &arka, L->solcocuk, kuyruk);
10     if (L->sagcocuk)
11       kuyruk_ekle(on, &arka, L->sagcocuk, kuyruk);
12   }
13 }
```



D'nin çocukları (H) kuyruğun arkasına eklenir: bir sonraki düzey sırada bekler.

Adım 9 / 14

DEĞİŞKENLER

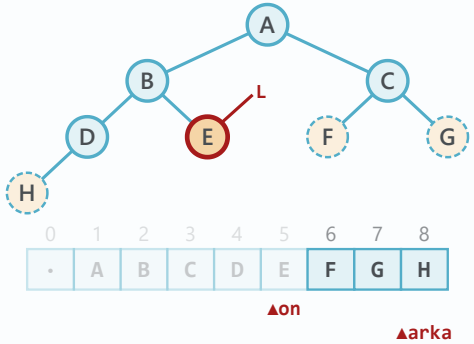
`L = &düğüm(D)` `on = 4` `arka = 8`

ÇIKTI

```
ABCD
```

Kuyrukla düzey düzey: A B C D E F G H

```
1  void duzey_dolas(agac_gost L) {
2    agac_gost kuyruk[MAX_SIZE];
3    int on = 0, arka = 0;
4    if (!L) return;              // ağaç boş olabilir
5    kuyruk_ekle(on, &arka, L, kuyruk);
▶    while ((L = kuyruk_al(&on, arka, kuyruk))) {
▶      printf("%c", L->veri);
8      if (L->solcocuk)
9        kuyruk_ekle(on, &arka, L->solcocuk, kuyruk);
10     if (L->sagcocuk)
11       kuyruk_ekle(on, &arka, L->sagcocuk, kuyruk);
12   }
13 }
```



Kuyruğun önünden `E` alınır ve yazılır (FIFO: önce gelen önce çıkar).

Adım 10 / 14

DEĞİŞKENLER

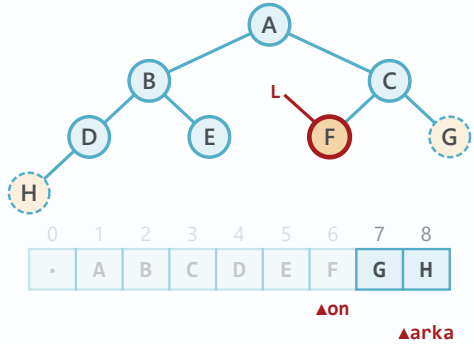
`L = &düğüm(E)` `on = 5` `arka = 8`

ÇIKTI

```
ABCDE
```

Kuyrukla düzey düzey: A B C D E F G H

```
1  void duzey_dolas(agac_gost L) {
2    agac_gost kuyruk[MAX_SIZE];
3    int on = 0, arka = 0;
4    if (!L) return;              // ağaç boş
   olabilir
5    kuyruk_ekle(on, &arka, L, kuyruk);
▶    while ((L = kuyruk_al(&on, arka, kuyruk))) {
▶      printf("%c", L->veri);
8      if (L->solcocuk)
9        kuyruk_ekle(on, &arka, L->solcocuk, kuyruk);
10     if (L->sagcocuk)
11       kuyruk_ekle(on, &arka, L->sagcocuk, kuyruk);
12   }
13 }
```



Kuyruğun önünden `F` alınır ve yazılır (FIFO: önce gelen önce çıkar).

Adım 11 / 14

DEĞİŞKENLER

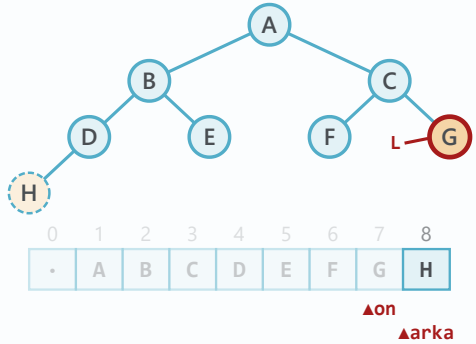
`L = &düğüm(F)` `on = 6` `arka = 8`

ÇIKTI

```
ABCDEF
```

Kuyrukla düzey düzey: A B C D E F G H

```
1  void duzey_dolas(agac_gost L) {
2    agac_gost kuyruk[MAX_SIZE];
3    int on = 0, arka = 0;
4    if (!L) return;              // ağaç boş
   olabilir
5    kuyruk_ekle(on, &arka, L, kuyruk);
▶    while ((L = kuyruk_al(&on, arka, kuyruk))) {
▶      printf("%c", L->veri);
8      if (L->solcocuk)
9        kuyruk_ekle(on, &arka, L->solcocuk, kuyruk);
10     if (L->sagcocuk)
11       kuyruk_ekle(on, &arka, L->sagcocuk, kuyruk);
12   }
13 }
```



Kuyruğun önünden `G` alınır ve yazılır (FIFO: önce gelen önce çıkar).

Adım 12 / 14

DEĞİŞKENLER

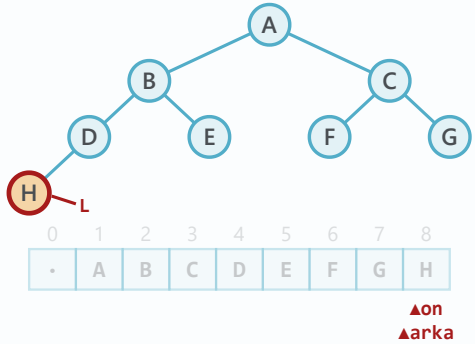
`L = &düğüm(G)` `on = 7` `arka = 8`

ÇIKTI

```
ABCDEFG
```

Kuyrukla düzey düzey: A B C D E F G H

```
1  void duzey_dolas(agac_gost L) {
2    agac_gost kuyruk[MAX_SIZE];
3    int on = 0, arka = 0;
4    if (!L) return;              // ağaç boş
   olabilir
5    kuyruk_ekle(on, &arka, L, kuyruk);
▶    while ((L = kuyruk_al(&on, arka, kuyruk))) {
▶      printf("%c", L->veri);
8      if (L->solcocuk)
9        kuyruk_ekle(on, &arka, L->solcocuk, kuyruk);
10     if (L->sagcocuk)
11       kuyruk_ekle(on, &arka, L->sagcocuk, kuyruk);
12   }
13 }
```



Kuyruğun önünden `H` alınır ve yazılır (FIFO: önce gelen önce çıkar).

Adım 13 / 14

DEĞİŞKENLER

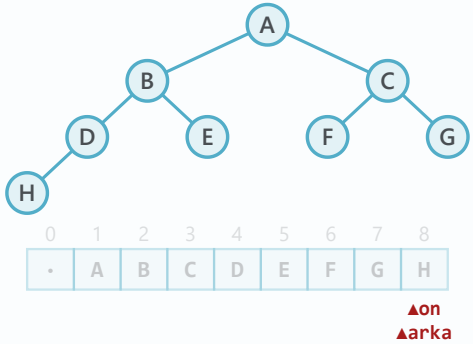
`L = &düğüm(H)` `on = 8` `arka = 8`

ÇIKTI

```
ABCDEFGH
```

Kuyrukla düzey düzey: A B C D E F G H

```
1  void duzey_dolas(agac_gost L) {
2    agac_gost kuyruk[MAX_SIZE];
3    int on = 0, arka = 0;
4    if (!L) return;              // ağaç boş
   olabilir
5    kuyruk_ekle(on, &arka, L, kuyruk);
▶    while ((L = kuyruk_al(&on, arka, kuyruk))) {
7      printf("%c", L->veri);
8      if (L->solcocuk)
9        kuyruk_ekle(on, &arka, L->solcocuk, kuyruk);
10     if (L->sagcocuk)
11       kuyruk_ekle(on, &arka, L->sagcocuk, kuyruk);
12   }
13 }
```



Kuyruk boş (on == arka): `kuyruk_al` NULL döner, döngü biter. Çıktı düzey düzey: ABCDEFGH.

Adım 14 / 14

DEĞİŞKENLER

`L = NULL` `on = 8` `arka = 8`

ÇIKTI

```
ABCDEFGH
```

Düzey sırası gezinmenin maliyeti

- Her düğüm kuyruğa bir kez girer, bir kez çıkar: 2n = **O(n)** zaman.
- Kuyrukta aynı anda en fazla **bir düzeyin genişliği** (w) kadar düğüm bulunur (artı birkaç sonraki düzey düğümü).
- Tam ağaçta son düzey ≈ n/2 düğüm → bellek **O(n)**; çarpık ağaçta O(1).
- O(lg n) bellek sınırı kuyruklu düzey sırası gezinme için değil, (dengeli ağaçta) yığınlı, derinlik öncelikli gezinmeler için geçerlidir.
- Aynı fikir çizgelerde genişlik öncelikli aramadır (BFS); kuyruk yerine yığın kullanılırsa derinlik öncelikli aramaya (DFS) döner.

gezinme	yardımcı yapı	ek bellek
kök önde/ortada/sonda	yığın (LIFO)	O(h)
düzey düzey	kuyruk (FIFO)	O(w) ≤ O(n)

HOCADAN "Nasıl ki derinlik öncelikli aramada problemin doğası gereği yığın kullanma zorunluluğu vardı; bunda da kuyruk kullanma zorunluluğu vardır."

SORU: ağacın kopyasını çıkartmak

```
agac_gost agac_kopyala(agac_gost L) {
  if (!L) return NULL;
  agac_gost T = malloc(sizeof *T);
  if (!T) { perror("malloc"); exit(1); }
  T->solcocuk = agac_kopyala(L->solcocuk);
  T->sagcocuk = agac_kopyala(L->sagcocuk);
  T->veri = L->veri;
  return T;
}
```

HOCADAN "Önce kâğıt üzerinde benim tahtada yaptığım çalışma yapılmalı, özyinelemeli çözüldüğü görülmeli. Bu yordam bottom-up çalışıyor, top-down da yazılabilirdi."

- Veri, iki çocuk kopyalandıktan sonra yazılıyor: yordam ağacı **kök sonda** biçiminde geziyor.
- Satırların yeri değiştirilerek kök önde / kök ortada biçimine kolayca çevrilir.
- `ogeal()` yerine doğrudan `malloc` + denetim kullanıldı.

agac_kopyala(): asıl ağaç ve kopyası



A için yeni düğüm ayrılır (`malloc` denetlendi); verisi henüz yazılmadı.

Adım 1 / 16

```
1 agac_gost agac_kopyala(agac_gost L) {
2   if (!L) return NULL;
▶   agac_gost T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->solcocuk = agac_kopyala(L->solcocuk);
6   T->sagcocuk = agac_kopyala(L->sagcocuk);
7   T->veri = L->veri;
8   return T;
9 }
```

DEĞİŞKENLER

`L = &düğüm(A)` `T = &yeni(A)`

ÇAĞRI YIĞINI

`agac_kopyala(A)`

agac_kopyala(): asıl ağaç ve kopyası



B için yeni düğüm ayrılır (`malloc` denetlendi); verisi henüz yazılmadı.

Adım 2 / 16

```
1 agac_gost agac_kopyala(agac_gost L) {
2   if (!L) return NULL;
▶   agac_gost T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->solcocuk = agac_kopyala(L->solcocuk);
6   T->sagcocuk = agac_kopyala(L->sagcocuk);
7   T->veri = L->veri;
8   return T;
9 }
```

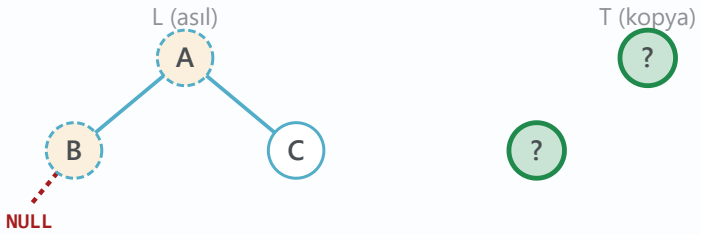
DEĞİŞKENLER

`L = &düğüm(B)` `T = &yeni(B)`

ÇAĞRI YIĞINI

- **agac_kopyala(B)**
- agac_kopyala(A)

agac_kopyala(): asıl ağaç ve kopyası



L NULL → kopyalanacak bir şey yok, **NULL** döner (B'nin sol kopyası NULL olacak).

Adım 3 / 16

```
1 agac_gost agac_kopyala(agac_gost L) {
▶   if (!L) return NULL;
3   agac_gost T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->solcocuk = agac_kopyala(L->solcocuk);
6   T->sagcocuk = agac_kopyala(L->sagcocuk);
7   T->veri = L->veri;
8   return T;
9 }
```

DEĞİŞKENLER

L = NULL T = ?

ÇAĞRI YIĞINI

- agac_kopyala(NULL)
- agac_kopyala(B)
- agac_kopyala(A)

agac_kopyala(): asıl ağaç ve kopyası



Sol çağrı döndü → T->solcocuk = NULL.

Adım 4 / 16

```
1 agac_gost agac_kopyala(agac_gost L) {
2   if (!L) return NULL;
3   agac_gost T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
▶   T->solcocuk = agac_kopyala(L->solcocuk);
6   T->sagcocuk = agac_kopyala(L->sagcocuk);
7   T->veri = L->veri;
8   return T;
9 }
```

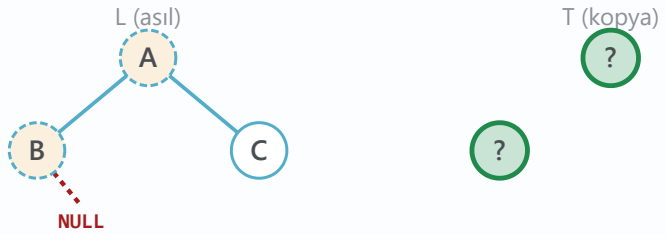
DEĞİŞKENLER

`L = &düğüm(B)` `T = &yeni(B)`

ÇAĞRI YIĞINI

- `agac_kopyala(B)`
- `agac_kopyala(A)`

agac_kopyala(): asıl ağaç ve kopyası



L NULL → kopyalanacak bir şey yok, **NULL** döner (B'nin sağ kopyası NULL olacak).

Adım 5 / 16

```
1 agac_gost agac_kopyala(agac_gost L) {
▶   if (!L) return NULL;
3   agac_gost T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->solcocuk = agac_kopyala(L->solcocuk);
6   T->sagcocuk = agac_kopyala(L->sagcocuk);
7   T->veri = L->veri;
8   return T;
9 }
```

DEĞİŞKENLER

`L = NULL` `T = ?`

ÇAĞRI YIĞINI

- `agac_kopyala(NULL)`
- `agac_kopyala(B)`
- `agac_kopyala(A)`

agac_kopyala(): asıl ağaç ve kopyası



Sağ çağrı döndü → T->sagcocuk = NULL.

Adım 6 / 16

```
1 agac_gost agac_kopyala(agac_gost L) {
2   if (!L) return NULL;
3   agac_gost T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->solcocuk = agac_kopyala(L->solcocuk);
▶   T->sagcocuk = agac_kopyala(L->sagcocuk);
7   T->veri = L->veri;
8   return T;
9 }
```

DEĞİŞKENLER

`L = &düğüm(B)` `T = &yeni(B)`

ÇAĞRI YIĞINI

- **agac_kopyala(B)**
- agac_kopyala(A)

agac_kopyala(): asıl ağaç ve kopyası



Veri kopyalanır (B) ve yeni düğümün adresi çağırana döndürülür.

Adım 7 / 16

```
1 agac_gost agac_kopyala(agac_gost L) {
2   if (!L) return NULL;
3   agac_gost T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->solcocuk = agac_kopyala(L->solcocuk);
6   T->sagcocuk = agac_kopyala(L->sagcocuk);
▶   T->veri = L->veri;
▶   return T;
9 }
```

DEĞİŞKENLER

`L = &düğüm(B)` `T = &yeni(B)`

ÇAĞRI YIĞINI

`agac_kopyala(B)`

`agac_kopyala(A)`

agac_kopyala(): asıl ağaç ve kopyası



Sol çağrı döndü → T->solcocuk = kopya(B).

Adım 8 / 16

```
1 agac_gost agac_kopyala(agac_gost L) {
2   if (!L) return NULL;
3   agac_gost T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
▶   T->solcocuk = agac_kopyala(L->solcocuk);
6   T->sagcocuk = agac_kopyala(L->sagcocuk);
7   T->veri = L->veri;
8   return T;
9 }
```

DEĞİŞKENLER

`L = &düğüm(A)` `T = &yeni(A)`

ÇAĞRI YIĞINI

`agac_kopyala(A)`

agac_kopyala(): asıl ağaç ve kopyası



C için yeni düğüm ayrılır (`malloc` denetlendi); verisi henüz yazılmadı.

Adım 9 / 16

```
1 agac_gost agac_kopyala(agac_gost L) {
2   if (!L) return NULL;
▶   agac_gost T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->solcocuk = agac_kopyala(L->solcocuk);
6   T->sagcocuk = agac_kopyala(L->sagcocuk);
7   T->veri = L->veri;
8   return T;
9 }
```

DEĞİŞKENLER

`L = &düğüm(C)` `T = &yeni(C)`

ÇAĞRI YIĞINI

`agac_kopyala(C)`

`agac_kopyala(A)`

agac_kopyala(): asıl ağaç ve kopyası



L NULL → kopyalanacak bir şey yok, **NULL** döner (C'nin sol kopyası NULL olacak).

Adım 10 / 16

```
1 agac_gost agac_kopyala(agac_gost L) {
▶   if (!L) return NULL;
3   agac_gost T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->solcocuk = agac_kopyala(L->solcocuk);
6   T->sagcocuk = agac_kopyala(L->sagcocuk);
7   T->veri = L->veri;
8   return T;
9 }
```

DEĞİŞKENLER

L = NULL T = ?

ÇAĞRI YIĞINI

- agac_kopyala(NULL)
- agac_kopyala(C)
- agac_kopyala(A)

agac_kopyala(): asıl ağaç ve kopyası



Sol çağrı döndü → T->solcocuk = NULL.

Adım 11 / 16

```
1 agac_gost agac_kopyala(agac_gost L) {
2   if (!L) return NULL;
3   agac_gost T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
▶   T->solcocuk = agac_kopyala(L->solcocuk);
6   T->sagcocuk = agac_kopyala(L->sagcocuk);
7   T->veri = L->veri;
8   return T;
9 }
```

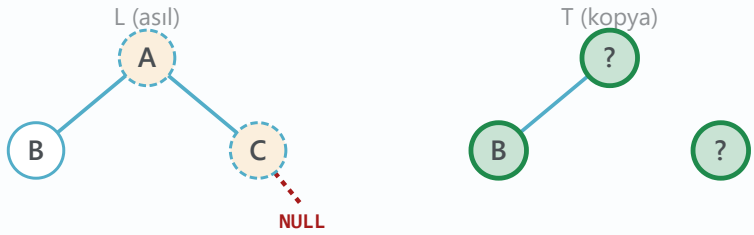
DEĞİŞKENLER

L = &düğüm(C) T = &yeni(C)

ÇAĞRI YIĞINI

agac_kopyala(C)
agac_kopyala(A)

agac_kopyala(): asıl ağaç ve kopyası



L NULL → kopyalanacak bir şey yok, **NULL** döner (C'nin sağ kopyası NULL olacak).

Adım 12 / 16

```
1 agac_gost agac_kopyala(agac_gost L) {
▶   if (!L) return NULL;
3   agac_gost T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->solcocuk = agac_kopyala(L->solcocuk);
6   T->sagcocuk = agac_kopyala(L->sagcocuk);
7   T->veri = L->veri;
8   return T;
9 }
```

DEĞİŞKENLER

L = NULL T = ?

ÇAĞRI YIĞINI

- agac_kopyala(NULL)
- agac_kopyala(C)
- agac_kopyala(A)

agac_kopyala(): asıl ağaç ve kopyası



Sağ çağrı döndü → T->sagcocuk = NULL.

Adım 13 / 16

```
1 agac_gost agac_kopyala(agac_gost L) {
2   if (!L) return NULL;
3   agac_gost T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->solcocuk = agac_kopyala(L->solcocuk);
▶   T->sagcocuk = agac_kopyala(L->sagcocuk);
7   T->veri = L->veri;
8   return T;
9 }
```

DEĞİŞKENLER

`L = &düğüm(C)` `T = &yeni(C)`

ÇAĞRI YIĞINI

- **agac_kopyala(C)**
- agac_kopyala(A)

agac_kopyala(): asıl ağaç ve kopyası



Veri kopyalanır (C) ve yeni düğümün adresi çağırana döndürülür.

Adım 14 / 16

```
1 agac_gost agac_kopyala(agac_gost L) {
2   if (!L) return NULL;
3   agac_gost T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->solcocuk = agac_kopyala(L->solcocuk);
6   T->sagcocuk = agac_kopyala(L->sagcocuk);
▶   T->veri = L->veri;
▶   return T;
9 }
```

DEĞİŞKENLER

`L = &düğüm(C)` `T = &yeni(C)`

ÇAĞRI YIĞINI

`agac_kopyala(C)`

`agac_kopyala(A)`

agac_kopyala(): asıl ağaç ve kopyası



Sağ çağrı döndü → T->sagcocuk = kopya(C).

Adım 15 / 16

```
1 agac_gost agac_kopyala(agac_gost L) {
2   if (!L) return NULL;
3   agac_gost T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->solcocuk = agac_kopyala(L->solcocuk);
▶   T->sagcocuk = agac_kopyala(L->sagcocuk);
7   T->veri = L->veri;
8   return T;
9 }
```

DEĞİŞKENLER

L = &düğüm(A) T = &yeni(A)

ÇAĞRI YIĞINI

agac_kopyala(A)

agac_kopyala(): asıl ağaç ve kopyası



Veri kopyalanır (A) ve yeni düğümün adresi çağırana döndürülür.

Adım 16 / 16

```
1 agac_gost agac_kopyala(agac_gost L) {
2   if (!L) return NULL;
3   agac_gost T = malloc(sizeof *T);
4   if (!T) { perror("malloc"); exit(1); }
5   T->solcocuk = agac_kopyala(L->solcocuk);
6   T->sagcocuk = agac_kopyala(L->sagcocuk);
▶   T->veri = L->veri;
▶   return T;
9 }
```

DEĞİŞKENLER

`L = &düğüm(A)` `T = &yeni(A)`

ÇAĞRI YIĞINI

`agac_kopyala(A)`

SORU: iki ağaç eş biçimli (isomorphic) mi?

```
int esitmi(agac_gost L, agac_gost T) {
  return (!L && !T) ||
         (L && T && L->veri == T->veri
          && esitmi(L->solcocuk, T->solcocuk)
          && esitmi(L->sagcocuk, T->sagcocuk));
}
```

Yalnızca **biçim** benzerliği isteniyorsa (veriler önemsizse) `L→veri == T→veri` ifadesi kaldırılır.

- İkisi de boşsa eşittir; biri boş diğeri doluysa eşit değildir.
- İkisi de doluysa: kökler eşit **ve** sol alt ağaçlar eşit **ve** sağ alt ağaçlar eşit.
- `&&` ve `||` kısa devre yapar: ilk farkta özyineleme durur.

HOCADAN "İkili ağaçların izomorfik olup olmadığını denetlemek bu kadar kolay! İki çizgenin (graf) eş biçimliliğini bulmak ise oldukça zor bir problemdir."

esitmi(): A(B,C) ile A(B,D)



İkisi de dolu ve veriler eşit (A = A) → sol alt ağaçlar karşılaştırılır.

Adım 1 / 9

```
1 int esitmi(agac_gost L, agac_gost T) {
2   return (!L && !T) ||
          (L && T && L->veri == T->veri
4          && esitmi(L->solcocuk, T->solcocuk)
5          && esitmi(L->sagcocuk, T->sagcocuk));
6 }
```

DEĞİŞKENLER

`L = &düğüm(A)` `T = &düğüm(A)`

ÇAĞRI YIĞINI

`esitmi(A, A)`

esitmi(): A(B,C) ile A(B,D)



İkisi de dolu ve veriler eşit (B = B) → sol alt ağaçlar karşılaştırılır.

Adım 2 / 9

```
1 int esitmi(agac_gost L, agac_gost T) {
2   return (!L && !T) ||
▶          (L && T && L->veri == T->veri
4           && esitmi(L->solcocuk, T->solcocuk)
5           && esitmi(L->sagcocuk, T->sagcocuk));
6 }
```

DEĞİŞKENLER

`L = &düğüm(B)` `T = &düğüm(B)`

ÇAĞRI YIĞINI

`esitmi(B, B)`

`esitmi(A, A)`

esitmi(): A(B,C) ile A(B,D)



İkisi de NULL: `(!L && !T)` doğru → **1** (boş ağaçlar eşittir); `||` 'nin sağı hiç hesaplanmaz.

Adım 3 / 9

```
1 int esitmi(agac_gost L, agac_gost T) {
▶   return (!L && !T) ||
3          (L && T && L->veri == T->veri
4           && esitmi(L->solcocuk, T->solcocuk)
5           && esitmi(L->sagcocuk, T->sagcocuk));
6 }
```

DEĞİŞKENLER

`L = NULL` `T = NULL`

ÇAĞRI YIĞINI

- `esitmi(NULL, NULL)`
- `esitmi(B, B)`
- `esitmi(A, A)`

esitmi(): A(B,C) ile A(B,D)



B: sol alt ağaçlar eşit çıktı → sağ alt ağaçlar karşılaştırılır.

Adım 4 / 9

```
1 int esitmi(agac_gost L, agac_gost T) {
2   return (!L && !T) ||
3          (L && T && L->veri == T->veri
4           && esitmi(L->solcocuk, T->solcocuk)
▶           && esitmi(L->sagcocuk, T->sagcocuk));
6 }
```

DEĞİŞKENLER

`L = &düğüm(B)` `T = &düğüm(B)`

ÇAĞRI YIĞINI

`esitmi(B, B)`

`esitmi(A, A)`

esitmi(): A(B,C) ile A(B,D)



İkisi de NULL: `(!L && !T)` doğru → **1** (boş ağaçlar eşittir); `||` 'nin sağı hiç hesaplanmaz.

Adım 5 / 9

```
1 int esitmi(agac_gost L, agac_gost T) {
▶   return (!L && !T) ||
3          (L && T && L->veri == T->veri
4           && esitmi(L->solcocuk, T->solcocuk)
5           && esitmi(L->sagcocuk, T->sagcocuk));
6 }
```

DEĞİŞKENLER

`L = NULL` `T = NULL`

ÇAĞRI YIĞINI

- `esitmi(NULL, NULL)`
- `esitmi(B, B)`
- `esitmi(A, A)`

esitmi(): A(B,C) ile A(B,D)



B: sağ taraf 1 döndü → bu çağrı **1** döndürür.

Adım 6 / 9

```
1 int esitmi(agac_gost L, agac_gost T) {
2   return (!L && !T) ||
3          (L && T && L->veri == T->veri
4           && esitmi(L->solcocuk, T->solcocuk)
▶           && esitmi(L->sagcocuk, T->sagcocuk));
6 }
```

DEĞİŞKENLER

`L = &düğüm(B)` `T = &düğüm(B)`

ÇAĞRI YIĞINI

- **esitmi(B, B)**
- esitmi(A, A)

esitmi(): A(B,C) ile A(B,D)



A: sol alt ağaçlar eşit çıktı → sağ alt ağaçlar karşılaştırılır.

Adım 7 / 9

```
1 int esitmi(agac_gost L, agac_gost T) {
2   return (!L && !T) ||
3          (L && T && L->veri == T->veri
4           && esitmi(L->solcocuk, T->solcocuk)
▶           && esitmi(L->sagcocuk, T->sagcocuk));
6 }
```

DEĞİŞKENLER

`L = &düğüm(A)` `T = &düğüm(A)`

ÇAĞRI YIĞINI

`esitmi(A, A)`

esitmi(): A(B,C) ile A(B,D)



Veriler farklı (C ≠ D) → `&&` kısa devre yapar; alt ağaçlara bakılmadan **0** döner.

Adım 8 / 9

```
1 int esitmi(agac_gost L, agac_gost T) {
2   return (!L && !T) ||
          (L && T && L->veri == T->veri
4          && esitmi(L->solcocuk, T->solcocuk)
5          && esitmi(L->sagcocuk, T->sagcocuk));
6 }
```

DEĞİŞKENLER

`L = &düğüm(C)` `T = &düğüm(D)`

ÇAĞRI YIĞINI

`esitmi(C, D)`

`esitmi(A, A)`

esitmi(): A(B,C) ile A(B,D)



A: sağ taraf 0 döndü → bu çağrı **0** döndürür.

Adım 9 / 9

```
1 int esitmi(agac_gost L, agac_gost T) {
2   return (!L && !T) ||
3          (L && T && L->veri == T->veri
4           && esitmi(L->solcocuk, T->solcocuk)
▶           && esitmi(L->sagcocuk, T->sagcocuk));
6 }
```

DEĞİŞKENLER

`L = &düğüm(A)` `T = &düğüm(A)`

ÇAĞRI YIĞINI

`esitmi(A, A)`

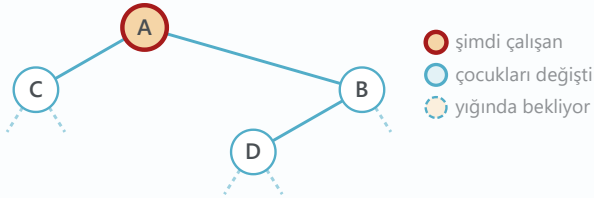
SORU: esitmi() yordamını farklı biçimde yazınız

```
int esitmi2(agac_gost L, agac_gost T) {
  if (L == NULL || T == NULL)
    return L == T;          /* ikisi de NULL mı? */
  if (L->veri != T->veri)
    return 0;
  if (!esitmi2(L->solcocuk, T->solcocuk))
    return 0;
  return esitmi2(L->sagcocuk, T->sagcocuk);
}
```

- Tek bir `return` ifadesi yerine erken dönüşlü (*guard*) yazım; davranış aynı.
- `L == T` : biri NULL iken ikisi de NULL ise doğru.
- Her özyinelemeli çağrının **dönüş değeri** kullanılmalı.

HOCADAN ""Hocam, benimki çok güzel çalışıyor da geri dönmüyor. Çağırıyorum, çağırıyorum, ama dönmüyor, tıkanıp kalıyor." O zaman yine özyinelemeli algoritma yazmayı öğrenememişiz demektir."

SORU: sol ve sağ çocukları değiştirmek



A: sol (B) ve sağ (C) **işaretçileri** takas edilir; veri yerinde kalır.

Adım 1 / 10

```
1 #define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
2 void cocukdegistir(agac_gost L) {
3   agac_gost p;
4   if (L) {
▶     SWAP(L->solcocuk, L->sagcocuk, p);
6     cocukdegistir(L->solcocuk);
7     cocukdegistir(L->sagcocuk);
8   }
9 }
```

DEĞİŞKENLER

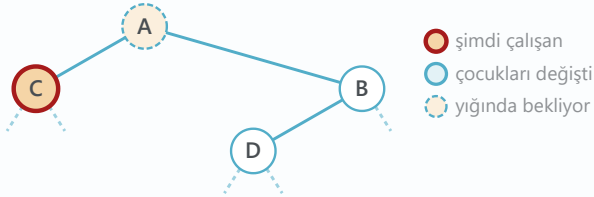
`L = &düğüm(A)`

ÇAĞRI YIĞINI

`cocukdegistir(A)`

“Verileri değiştirmiyorum, sadece işaretçileri takas ediyorum.”

SORU: sol ve sağ çocukları değiştirmek



C: sol (NULL) ve sağ (NULL) **işaretçileri** takas edilir; veri yerinde kalır.

Adım 2 / 10

```
1 #define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
2 void cocukdegistir(agac_gost L) {
3   agac_gost p;
4   if (L) {
▶     SWAP(L->solcocuk, L->sagcocuk, p);
6     cocukdegistir(L->solcocuk);
7     cocukdegistir(L->sagcocuk);
8   }
9 }
```

DEĞİŞKENLER

`L = &düğüm(C)`

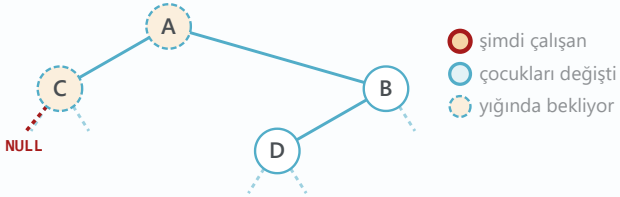
ÇAĞRI YIĞINI

`cocukdegistir(C)`

`cocukdegistir(A)`

“Verileri değiştirmiyorum, sadece işaretçileri takas ediyorum.”

SORU: sol ve sağ çocukları değiştirmek



L NULL → hiçbir şey yapmadan döner.

Adım 3 / 10

```
1 #define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
2 void cocukdegistir(agac_gost L) {
3   agac_gost p;
▶   if (L) {
5     SWAP(L->solcocuk, L->sagcocuk, p);
6     cocukdegistir(L->solcocuk);
7     cocukdegistir(L->sagcocuk);
8   }
9 }
```

DEĞİŞKENLER

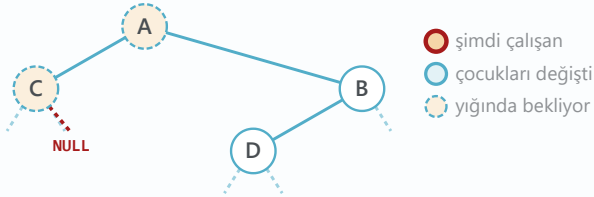
L = NULL

ÇAĞRI YIĞINI

- cocukdegistir(NULL)
- cocukdegistir(C)
- cocukdegistir(A)

“Verileri değiştirmiyorum, sadece işaretçileri takas ediyorum.”

SORU: sol ve sağ çocukları değiştirmek



L NULL → hiçbir şey yapmadan döner.

Adım 4 / 10

```
1 #define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
2 void cocukdegistir(agac_gost L) {
3   agac_gost p;
▶   if (L) {
5     SWAP(L->solcocuk, L->sagcocuk, p);
6     cocukdegistir(L->solcocuk);
7     cocukdegistir(L->sagcocuk);
8   }
9 }
```

DEĞİŞKENLER

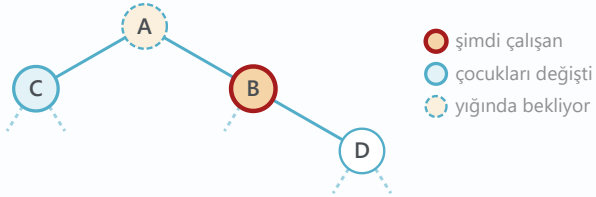
L = NULL

ÇAĞRI YIĞINI

- cocukdegistir(NULL)
- cocukdegistir(C)
- cocukdegistir(A)

“Verileri değiştirmiyorum, sadece işaretçileri takas ediyorum.”

SORU: sol ve sağ çocukları değiştirmek



B: sol (D) ve sağ (NULL) **işaretçileri** takas edilir; veri yerinde kalır.

Adım 5 / 10

```
1 #define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
2 void cocukdegistir(agac_gost L) {
3   agac_gost p;
4   if (L) {
▶     SWAP(L->solcocuk, L->sagcocuk, p);
6     cocukdegistir(L->solcocuk);
7     cocukdegistir(L->sagcocuk);
8   }
9 }
```

DEĞİŞKENLER

`L = &düğüm(B)`

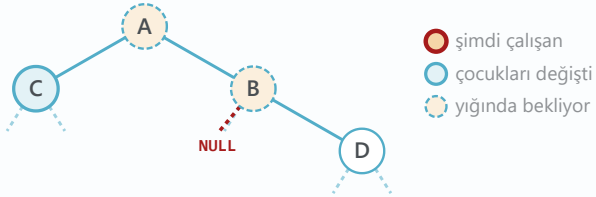
ÇAĞRI YIĞINI

`cocukdegistir(B)`

`cocukdegistir(A)`

“Verileri değiştirmiyorum, sadece işaretçileri takas ediyorum.”

SORU: sol ve sağ çocukları değiştirmek



L NULL → hiçbir şey yapmadan döner.

Adım 6 / 10

```
1 #define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
2 void cocukdegistir(agac_gost L) {
3   agac_gost p;
▶   if (L) {
5     SWAP(L->solcocuk, L->sagcocuk, p);
6     cocukdegistir(L->solcocuk);
7     cocukdegistir(L->sagcocuk);
8   }
9 }
```

DEĞİŞKENLER

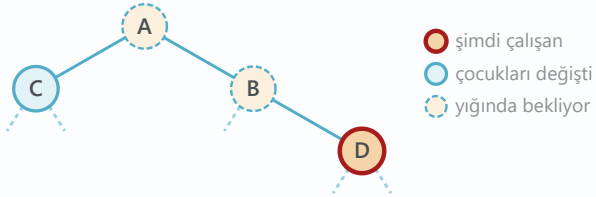
L = NULL

ÇAĞRI YIĞINI

- cocukdegistir(NULL)
- cocukdegistir(B)
- cocukdegistir(A)

“Verileri değiştirmiyorum, sadece işaretçileri takas ediyorum.”

SORU: sol ve sağ çocukları değiştirmek



D: sol (NULL) ve sağ (NULL) **işaretçileri** takas edilir; veri yerinde kalır.

Adım 7 / 10

```
1 #define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
2 void cocukdegistir(agac_gost L) {
3   agac_gost p;
4   if (L) {
▶     SWAP(L->solcocuk, L->sagcocuk, p);
6     cocukdegistir(L->solcocuk);
7     cocukdegistir(L->sagcocuk);
8   }
9 }
```

DEĞİŞKENLER

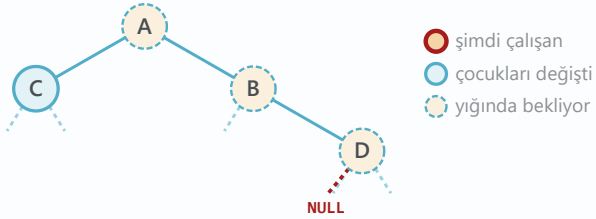
`L = &düğüm(D)`

ÇAĞRI YIĞINI

- `cocukdegistir(D)`
- `cocukdegistir(B)`
- `cocukdegistir(A)`

“Verileri değiştirmiyorum, sadece işaretçileri takas ediyorum.”

SORU: sol ve sağ çocukları değiştirmek



L NULL → hiçbir şey yapmadan döner.

Adım 8 / 10

```
1 #define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
2 void cocukdegistir(agac_gost L) {
3   agac_gost p;
▶   if (L) {
5     SWAP(L->solcocuk, L->sagcocuk, p);
6     cocukdegistir(L->solcocuk);
7     cocukdegistir(L->sagcocuk);
8   }
9 }
```

DEĞİŞKENLER

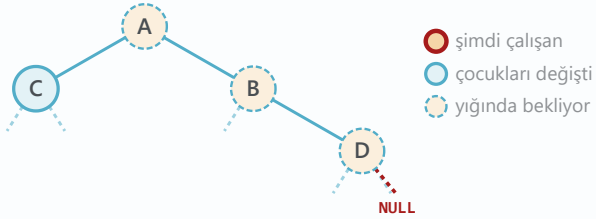
`L = NULL`

ÇAĞRI YIĞINI

- `cocukdegistir(NULL)`
- `cocukdegistir(D)`
- `cocukdegistir(B)`
- `cocukdegistir(A)`

“Verileri değiştirmiyorum, sadece işaretçileri takas ediyorum.”

SORU: sol ve sağ çocukları değiştirmek



L NULL → hiçbir şey yapmadan döner.

Adım 9 / 10

```
1 #define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
2 void cocukdegistir(agac_gost L) {
3   agac_gost p;
▶   if (L) {
5     SWAP(L->solcocuk, L->sagcocuk, p);
6     cocukdegistir(L->solcocuk);
7     cocukdegistir(L->sagcocuk);
8   }
9 }
```

DEĞİŞKENLER

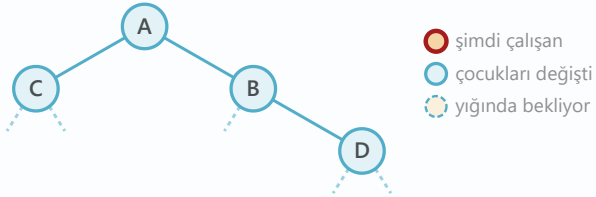
L = NULL

ÇAĞRI YIĞINI

- cocukdegistir(NULL)
- cocukdegistir(D)
- cocukdegistir(B)
- cocukdegistir(A)

“Verileri değiştirmiyorum, sadece işaretçileri takas ediyorum.”

SORU: sol ve sağ çocukları değiştirmek



Bitti: ağaç aynadaki görüntüsüne döndü (kök ortada sırası tersine çevrildi: CABD).

Adım 10 / 10

```
1 #define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
2 void cocukdegistir(agac_gost L) {
3   agac_gost p;
4   if (L) {
5     SWAP(L->solcocuk, L->sagcocuk, p);
6     cocukdegistir(L->solcocuk);
7     cocukdegistir(L->sagcocuk);
8   }
▶ }
```

DEĞİŞKENLER

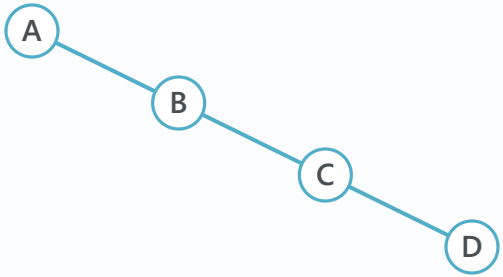
L = –

ÇAĞRI YIĞINI

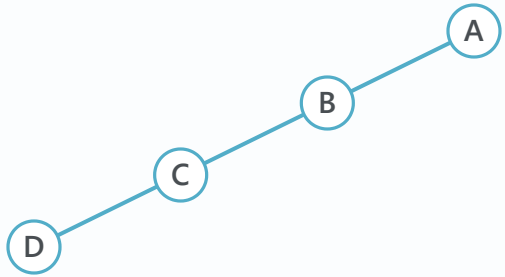
main()

“Verileri değiştirmiyorum, sadece işaretçileri takas ediyorum.”

SWAP özyinelemeli çağrıların öncesinde, arasında, sonunda



başlangıç: sağa çarpık A-B-C-D



SWAP öncesinde → tam ayna

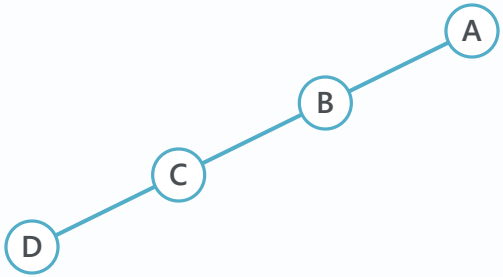
Örnek ağaç: tüm sağ alt ağaçlar tek kollu. SWAP çağrılardan **önce**: her düğümde takas, sonuç sola çarpık ayna.

Adım 1 / 2

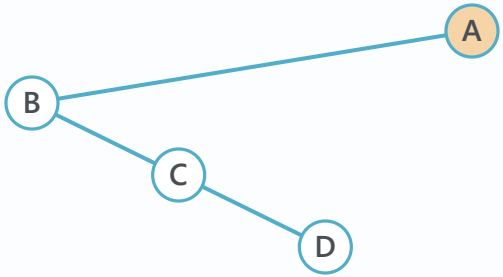
"Aynı sonucu üretir" gözlemi önce/sonra için doğrudur; ara konumda sonuç farklıdır.

DİĞER YORDAMLAR

SWAP özyinelemeli çağrıların öncesinde, arasında, sonunda



SWAP sonunda → yine tam ayna



SWAP arasında → yalnız kök takaslandı

SWAP **sonda** (kök sonda) aynı sonucu verir. **Arasında** ise: sol NULL işlenir, takas olur, sonra "sağ" (artık NULL) işlenir → B alt ağacına hiç dokunulmaz.

Adım 2 / 2

"Aynı sonucu üretir" gözlemi önce/sonra için doğrudur; ara konumda sonuç farklıdır.

SINAV SORUSU

Sınav sorusu: kök ortada + kök sonda → ağaç

Kök ortada sırası `C M D A K N B F E G`, kök sonda sırası `D A N F E B K M G C` olan ikili ağacı çiziniz. *Tek bir ağaç görünümü vardır.*

▶ **Çözüm (adım adım)**

Yöntem: kök sonda sırası sondan başa okunur (her harf bir alt ağacın kökü); kök ortada o harfin solunu/sağını ayırır. Kök önde + kök ortada da tek ağaç verir; kök önde + kök sonda genelde vermez (tek çocuğun solda mı sağda mı olduğu anlaşılmaz).

SINAV SORUSU

Sınav sorusu: kök ortada + kök sonda → ağaç

Kök ortada sırası `C M D A K N B F E G`, kök sonda sırası `D A N F E B K M G C` olan ikili ağacı çiziniz. *Tek bir ağaç görünümü vardır.*

▶ **Çözüm (adım adım)**

Yöntem: kök sonda sırası sondan başa okunur (her harf bir alt ağacın kökü); kök ortada o harfin solunu/sağını ayırır. Kök önde + kök ortada da tek ağaç verir; kök önde + kök sonda genelde vermez (tek çocuğun solda mı sağda mı olduğu anlaşılmaz).

SINAV SORUSU

Sınav sorusu: kök ortada + kök sonda → ağaç

Kök ortada sırası `C M D A K N B F E G`, kök sonda sırası `D A N F E B K M G C` olan ikili ağacı çiziniz. *Tek bir ağaç görünümü vardır.*

▶ **Çözüm (adım adım)**

Yöntem: kök sonda sırası sondan başa okunur (her harf bir alt ağacın kökü); kök ortada o harfin solunu/sağını ayırır. Kök önde + kök ortada da tek ağaç verir; kök önde + kök sonda genelde vermez (tek çocuğun solda mı sağda mı olduğu anlaşılmaz).

SINAV SORUSU

Sınav sorusu: kök ortada + kök sonda → ağaç

Kök ortada sırası `C M D A K N B F E G`, kök sonda sırası `D A N F E B K M G C` olan ikili ağacı çiziniz. *Tek bir ağaç görünümü vardır.*

▶ **Çözüm (adım adım)**

Yöntem: kök sonda sırası sondan başa okunur (her harf bir alt ağacın kökü); kök ortada o harfin solunu/sağını ayırır. Kök önde + kök ortada da tek ağaç verir; kök önde + kök sonda genelde vermez (tek çocuğun solda mı sağda mı olduğu anlaşılmaz).

SINAV SORUSU

Sınav sorusu: kök ortada + kök sonda → ağaç

Kök ortada sırası `C M D A K N B F E G`, kök sonda sırası `D A N F E B K M G C` olan ikili ağacı çiziniz. *Tek bir ağaç görünümü vardır.*

▶ **Çözüm (adım adım)**

Yöntem: kök sonda sırası sondan başa okunur (her harf bir alt ağacın kökü); kök ortada o harfin solunu/sağını ayırır. Kök önde + kök ortada da tek ağaç verir; kök önde + kök sonda genelde vermez (tek çocuğun solda mı sağda mı olduğu anlaşılmaz).

SINAV SORUSU

Sınav sorusu: kök ortada + kök sonda → ağaç

Kök ortada sırası `C M D A K N B F E G`, kök sonda sırası `D A N F E B K M G C` olan ikili ağacı çiziniz. *Tek bir ağaç görünümü vardır.*

▶ **Çözüm (adım adım)**

Yöntem: kök sonda sırası sondan başa okunur (her harf bir alt ağacın kökü); kök ortada o harfin solunu/sağını ayırır. Kök önde + kök ortada da tek ağaç verir; kök önde + kök sonda genelde vermez (tek çocuğun solda mı sağda mı olduğu anlaşılmaz).

SINAV SORUSU

Sınav sorusu: kök ortada + kök sonda → ağaç

Kök ortada sırası `C M D A K N B F E G`, kök sonda sırası `D A N F E B K M G C` olan ikili ağacı çiziniz. *Tek bir ağaç görünümü vardır.*

▶ **Çözüm (adım adım)**

Yöntem: kök sonda sırası sondan başa okunur (her harf bir alt ağacın kökü); kök ortada o harfin solunu/sağını ayırır. Kök önde + kök ortada da tek ağaç verir; kök önde + kök sonda genelde vermez (tek çocuğun solda mı sağda mı olduğu anlaşılmaz).

SINAV SORUSU

Sınav sorusu: kök ortada + kök sonda → ağaç

Kök ortada sırası `C M D A K N B F E G`, kök sonda sırası `D A N F E B K M G C` olan ikili ağacı çiziniz. *Tek bir ağaç görünümü vardır.*

▶ **Çözüm (adım adım)**

Yöntem: kök sonda sırası sondan başa okunur (her harf bir alt ağacın kökü); kök ortada o harfin solunu/sağını ayırır. Kök önde + kök ortada da tek ağaç verir; kök önde + kök sonda genelde vermez (tek çocuğun solda mı sağda mı olduğu anlaşılmaz).

SINAV SORUSU

Sınav sorusu: kök ortada + kök sonda → ağaç

Kök ortada sırası `C M D A K N B F E G`, kök sonda sırası `D A N F E B K M G C` olan ikili ağacı çiziniz. *Tek bir ağaç görünümü vardır.*

▶ **Çözüm (adım adım)**

Yöntem: kök sonda sırası sondan başa okunur (her harf bir alt ağacın kökü); kök ortada o harfin solunu/sağını ayırır. Kök önde + kök ortada da tek ağaç verir; kök önde + kök sonda genelde vermez (tek çocuğun solda mı sağda mı olduğu anlaşılmaz).

SINAV SORUSU

Sınav sorusu: kök ortada + kök sonda → ağaç

Kök ortada sırası `C M D A K N B F E G`, kök sonda sırası `D A N F E B K M G C` olan ikili ağacı çiziniz. *Tek bir ağaç görünümü vardır.*

▶ **Çözüm (adım adım)**

Yöntem: kök sonda sırası sondan başa okunur (her harf bir alt ağacın kökü); kök ortada o harfin solunu/sağını ayırır. Kök önde + kök ortada da tek ağaç verir; kök önde + kök sonda genelde vermez (tek çocuğun solda mı sağda mı olduğu anlaşılmaz).

ÖZET

Karmaşıklık özeti (n düğüm, h derinlik)

işlem	zaman	ek bellek	not
ebeveyn / çocuk (dizi gösterimi)	O(1)	$2^k - 1$ göz	tam ağaçta kayıp yok
özyinelemeli kök önde / ortada / sonda	O(n)	O(h)	n+1 NULL çağrısı
yığınlı kök ortada / önde / sonda	O(n)	O(h)	yığına adres konur
düzey düzey (kuyruk)	O(n)	O(w) ≤ O(n)	w: en geniş düzey
uç düğüm yaz, SORU 7–9	O(n)	O(h)	gezinme + koşul
agac_kopyala	O(n)	O(h) + n düğüm	kök sonda
esitmi	O(min(n_1, n_2))	O(h)	ilk farkta durur
cocukdegistir	O(n)	O(h)	yalnız işaretçiler

h: lg n ≤ h ≤ n (dengeli ↔ çarpık). Bu derste derinlik düzey sayısıyla ölçülür.

ÖZET

Sık yapılan hatalar

1. `L→veri`'ye NULL kontrolü yapmadan erişmek (“ağaç işaretçisi NULL gelebilir”).
2. Dizi gösteriminde 0'dan başlayıp 1 tabanlı formülleri (2i, 2i+1) kullanmak.
3. Yığına düğümün kopyasını koymak; adres yetmez mi?
4. Özyinelemeli çağrının dönüş değerini kullanmamak ya da `return`'den sonra iş yapmaya çalışmak.
5. Uç düğüm koşulunda `&&` yerine `||` yazmak.
6. Derece ile derinliği, düzey 1 ile derinlik 0 tanımlarını karıştırmak.

ÖZET

Alıştırmalar

1. $n_0 = n_2 + 1$ eşitliğini kanıtlayınız.
2. `(A+B)*(C-D)` ifade ağacını çizip üç gezinmeyi yazınız.
3. Düğüm sayısını döndüren `int say(agac_gost L)` yordamını yazınız.
4. Kök önde `A B D E C F`, kök ortada `D B E A F C` olan ağacı çiziniz.
5. Düzeyleri **sağdan sola** yazan kuyruklu yordamı yazınız.
6. İki ağacın birbirinin **aynası** olup olmadığını denetleyen yordamı yazınız.

▶ **İpuçları / cevaplar**

ÖZET

Bölüm özeti

Terminoloji

Kök, ebeveyn, çocuk, kardeş, ata, soy, derece, yaprak; düzey 1'den başlar, derinlik = en büyük düzey.

İkili ağaç

i. düzeyde ≤ 2^{i-1}, k derinlikte ≤ $2^k - 1$ düğüm; tam ağaç diziye sığar.

Gösterim

Tam ve sabit ağaç → dizi; değişen ağaç → `agac_gost` bağları.

Gezinme

Üç özyinelemeli gezinme aynı iskelet; yığınla döngüsel, kuyrukla düzey düzey.

Yordamlar

Uç düğüm, kopyala, esitmi, cocukdegistir: hepsi "gez + işle" kalıbı.

Maliyet

Zaman O(n); bellek derinlikle orantılı O(h).

ÖZET

Kaynaklar ve devam

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. bs., Bölüm 5 (Trees).
- T. H. Cormen vd., *Introduction to Algorithms*, Ek B.5 ve Bölüm 12.
- R. Sedgewick, *Algorithms in C*, Kısım 1–4, Bölüm 5 (Trees).

← Bölüm 9: Liste ADT ve Genelleştirilmiş Listeler | Ders sayfası | Bölüm 11: Arama Ağaçları →

Konunun devamı: ikili arama ağaçları ve kılavuzlu ikili ağaçlar.

Son

Bölüm 10: Ağaçlar

Sonraki bölüm: Bölüm 11: Arama Ağaçları · Ders sayfası