Bölüm 11: Arama Ağaçları

C ile Veri Yapıları ve Algoritmalar

İkili arama ağacı · Ekleme · Arama · Derinlik · Silme · AVL döndürmeleri · Kırmızı-siyah, B-ağacı

Kaynak: Horowitz–Sahni 5.7, 10.2 · CLRS 12–13

GİRİŞ

Öğrenme Çıktıları

Tanımı uygulamak
İkili arama ağacının dört özelliğini bir ağaçta denetlemek.

Ekleme ve arama
`arama_agacina_ekle` ve `arama_agacinda_ara` yordamlarını döngüsel ve özyinelemeli yazmak.

Derinliği hesaplamak
Derinliği özyinelemeli yazmak; ek parametre ve global değişkenden kaçınmak.

Silmek
Çocuksuz, tek çocuklu ve iki çocuklu düğüm silmenin dört yöntemini karşılaştırmak.

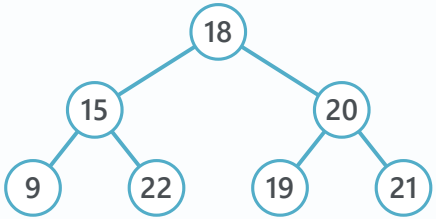
Dengeyi korumak
Dengesizliğin maliyetini hesaplamak; AVL'de LL, RR, LR, RL döndürmelerini uygulamak.

Yapı seçmek
BST, AVL, kırmızı-siyah ve B-ağacını kullanım yerine göre ayırmak.

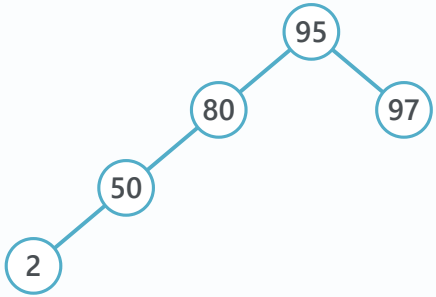
İkili arama ağacı (binary search tree)

1. Her öğenin değeri farklıdır; değerler **tektir** (unique).
2. Boş olmayan **sol** alt ağaçtaki değerler, alt ağacın kökündekinden **küçüktür**.
3. Boş olmayan **sağ** alt ağaçtaki değerler kökündekinden **büyüktür**.
4. Sol ve sağ alt ağaçlar da ikili arama ağacıdır.

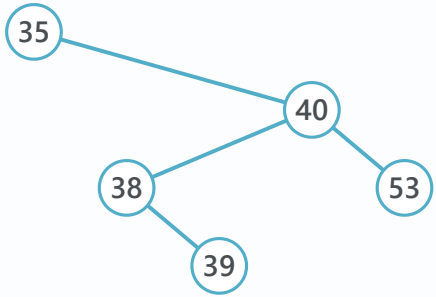
İkili arama ağacı, ikili ağacın bir alt kümesidir. Kök ortada gezinme değerleri **sıralı** verir.



ikili arama ağacı DEĞİL

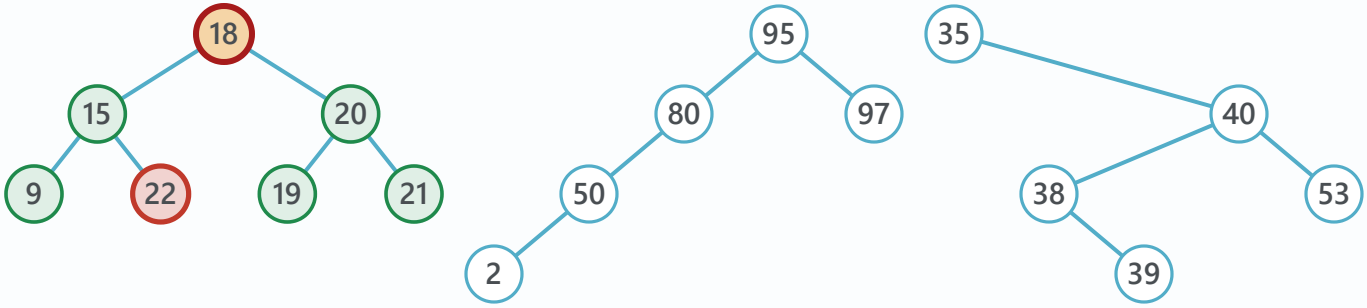


ikili arama ağacı



ikili arama ağacı

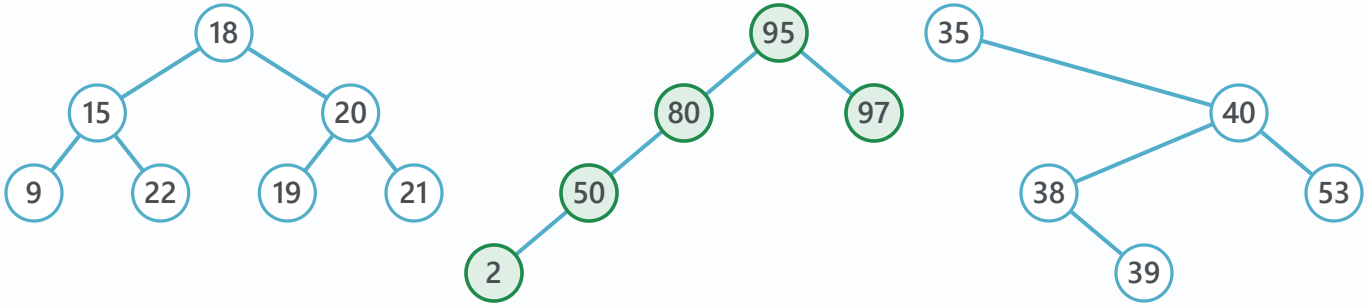
Hangisi ikili arama ağacı?



ikili arama ağacı DEĞİL | **ikili arama ağacı** | **ikili arama ağacı**

22, 15'in sağında olduğu için 15'ten büyük (doğru) ama 18'in **sol** alt ağacında olduğu hâlde 18'den büyük → tanıma uymaz.

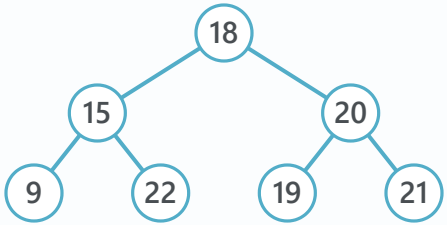
Hangisi ikili arama ağacı?



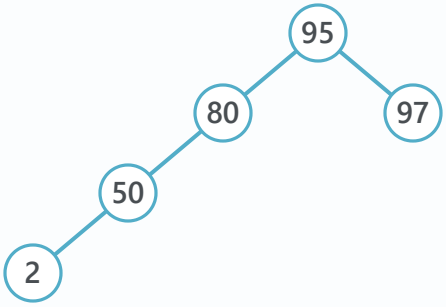
ikili arama ağacı DEĞİL | **ikili arama ağacı** | **ikili arama ağacı**

95'in solundaki her değer (80, 50, 2) 95'ten küçük; 80'in solu 50, 50'nin solu 2: sıralama bozulmamış.

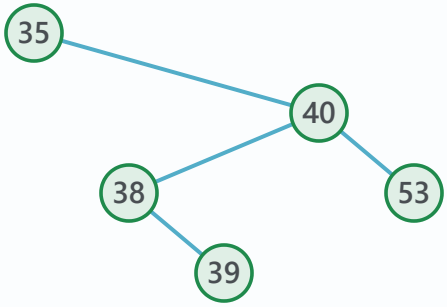
Hangisi ikili arama ağacı?



ikili arama ağacı DEĞİL



ikili arama ağacı



ikili arama ağacı

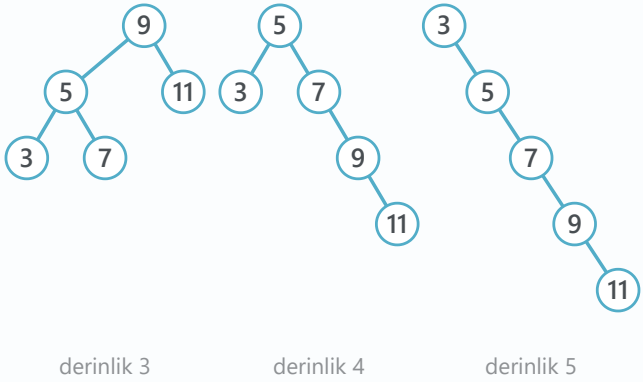
35'in sağındaki her değer büyük; 40'ın solunda 38 ve 39 (40'tan küçük, 35'ten büyük): geçerli.

Ekleme: yeri ağaç belirler

HOCADAN “Bir değer söylensin, ben ekleyeyim. “Hocam 15’in sağına ekleyin.” Sağa eklerim, sola eklerim; nereye ekleyeceğim benim bileceğim iş! ☺”

SORU: 3, 5, 7, 9, 11 kümesiyle derinliği 3, 4 ve 5 olan ikili arama ağaçları çiziniz. 2 derinliğinde çizilebilir mi?

▶ **Çözüm**



derinlik 3 derinlik 4 derinlik 5

EKLEME

Verilerin ağaca eklenme sırası önemli midir?



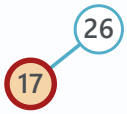
47 (sol) ve **26** (sağ) eklenir: kökten başlanıp küçükse sola, büyükse sağa inilir, ilk NULL'a konur.

Adım 1 / 9

EKLEME

Verilerin ağaca eklenme sırası önemli midir?





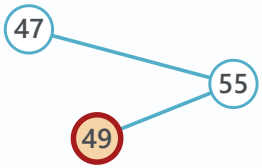
47 **55**
derinlik 2

26 **17**
derinlik 2

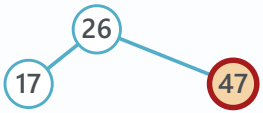
55 (sol) ve **17** (sağ) eklenir: kökten başlanıp küçükse sola, büyükse sağa inilir, ilk NULL'a konur.

EKLEME

Verilerin ağaca eklenme sırası önemli midir?



47 55 49
derinlik 3

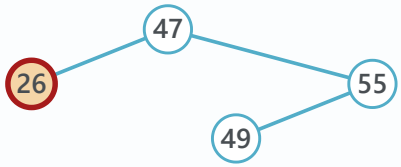


26 17 47
derinlik 2

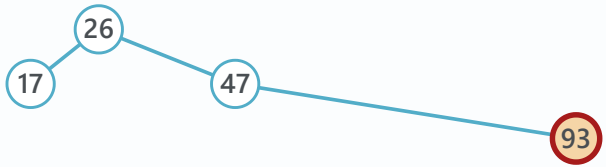
49 (sol) ve **47** (sağ) eklenir: kökten başlanıp küçükse sola, büyükse sağa inilir, ilk NULL'a konur.

EKLEME

Verilerin ağaca eklenme sırası önemli midir?



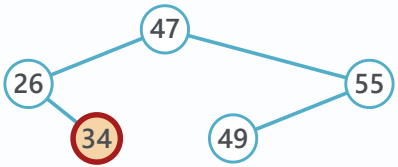
47 55 49 26
derinlik 3



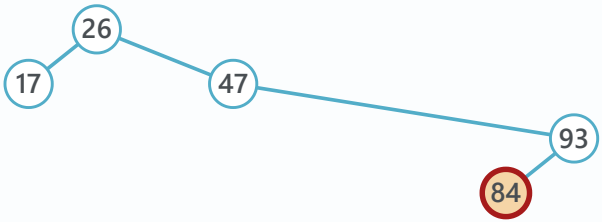
26 17 47 93
derinlik 3

26 (sol) ve **93** (sağ) eklenir: kökten başlanıp küçükse sola, büyükse sağa inilir, ilk NULL'a konur.

Verilerin ağaca eklenme sırası önemli midir?



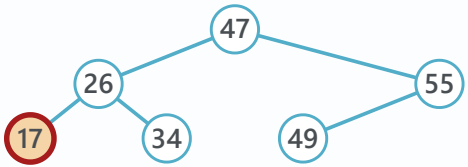
47 55 49 26 34
derinlik 3



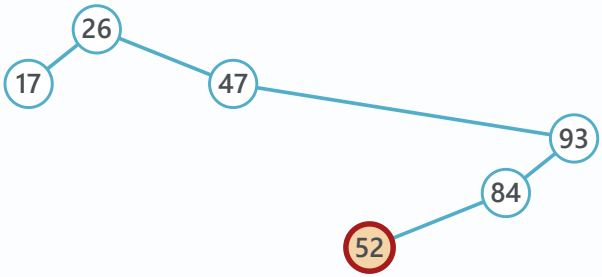
26 17 47 93 84
derinlik 4

34 (sol) ve **84** (sağ) eklenir: kökten başlanıp küçükse sola, büyükse sağa inilir, ilk NULL'a konur.

Verilerin ağaca eklenme sırası önemli midir?



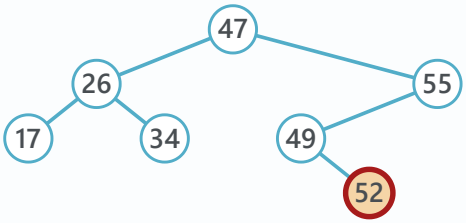
47 55 49 26 34 17
derinlik 3



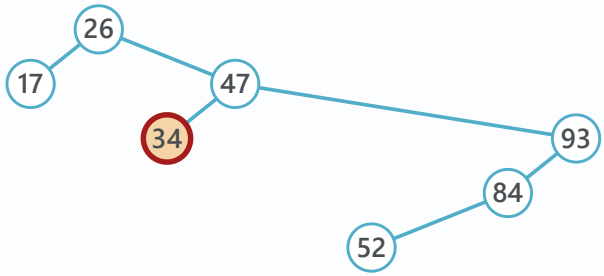
26 17 47 93 84 52
derinlik 5

17 (sol) ve **52** (sağ) eklenir: kökten başlanıp küçükse sola, büyükse sağa inilir, ilk NULL'a konur.

Verilerin ağaca eklenme sırası önemli midir?



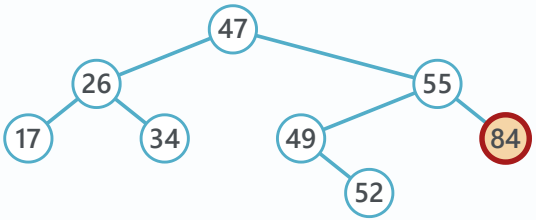
47 55 49 26 34 17 52
derinlik 4



26 17 47 93 84 52 34
derinlik 5

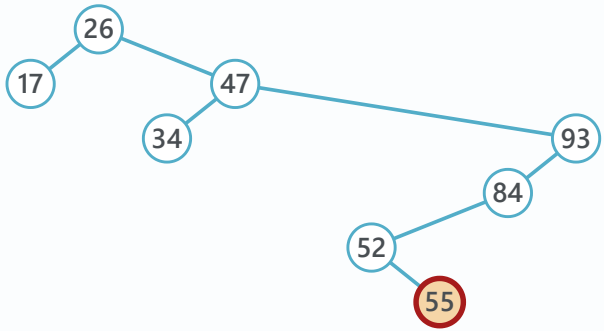
52 (sol) ve **34** (sağ) eklenir: kökten başlanıp küçükse sola, büyükse sağa inilir, ilk NULL'a konur.

Verilerin ağaca eklenme sırası önemli midir?



47 55 49 26 34 17 52 84

derinlik 4

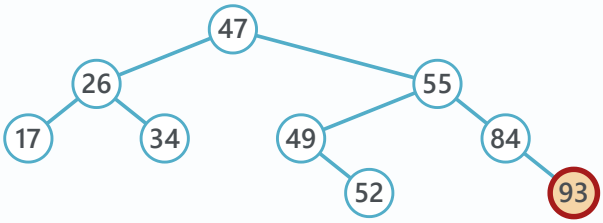


26 17 47 93 84 52 34 55

derinlik 6

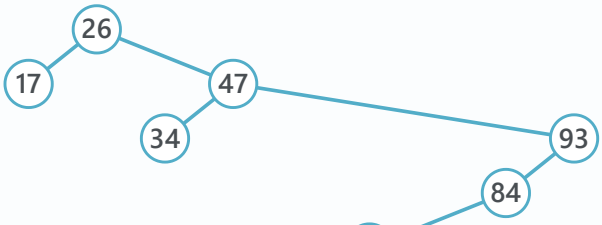
84 (sol) ve **55** (sağ) eklenir: kökten başlanıp küçükse sola, büyükse sağa inilir, ilk NULL'a konur.

Verilerin ağaca eklenme sırası önemli midir?



47 55 49 26 34 17 52 84 93

derinlik 4



26 17 47 93 84 52 34 55 49

derinlik 6

Sonuç: derinlik **4** ve **6**. Kök ortada gezinme ikisinde de aynı: 17 26 34 47 49 52 55 84 93.

EKLEME

Çarpıklık, dengeli ağaçlar ve sık kullanılan düğümler

- Veriler **sıralı** gelirse ağaç sağa ya da sola çarpık olur, derinliği **n** olur: liste gibi.
- Derinlik ≈ lg n ise **dengeli** ikili arama ağacı: ekleme, silme, arama O(lg n).
- **Dengeli ağaçlar** (self-balancing): veri geldikçe ağacı yeniden dengeleyen algoritmalar — kırmızı-siyah, AVL.
- **Sık kullanılan düğümler**: arama sıklığı yüksek düğümleri köke yaklaştıran algoritmalar ("zig-zag"; literatürdeki adıyla *splay tree*, Sleator–Tarjan).

> **HOCADAN** "Ağacın dengeli olması çok önemliyse ağacı dengeli kuran algoritmalar kullanılmalıdır. Çok zor değil, kitaplarda da var, yalnızca biraz daha uzun. Ödev olarak sorabilirim."

EKLEME

??? — düşündüren iki soru

??? Ağaç görüntüsünden verilerin eklenme sırası elde edilebilir mi? Gelen veriye göre oluşan ağaç *unique* mi?

▶ **Cevap**

??? Kök önde, kök ortada, kök sonda gezinme sırasından ağaç görüntüsü elde edilebilir mi?

▶ **Cevap**

HOCADAN "Şimdiye kadar arka planda bir ağaç olmadığı için kod çalıştırılarak test edilemiyordu."

Artık ekleme yordamıyla örnek ağaçlar kurup ikili ağaç yordamlarını (gezinme, kopyalama, eşitlik) gerçekten çalıştırabiliriz.

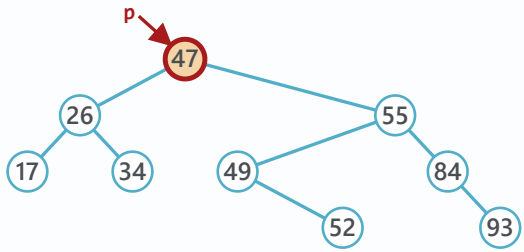
Ekleme nasıl yapılır?

- Her iki çocuğu da olan düğüme eklenemez; **en az bir çocuğu olmayan** düğüme eklenir.
- Gelen veri kökten itibaren aranır: daha büyük değerli düğümde **sola**, daha küçük değerlide **sağa** ilerlenir.
- Aynı değer bulunursa tanıma aykırı olacağından ekleme yapılmaz.
- Arama işaretçisinin ilk rastladığı **NULL**'un yerine yeni öğe eklenir: bunun için bir adım **arkadan** gelen işaretçi (q) gerekir.

```
typedef enum { FALSE, TRUE } boolean;
typedef struct dugum *agac_gost;
struct dugum {
    agac_gost solcocuk;
    int       veri;
    agac_gost sagcocuk;
};
/* ogeal(): malloc + NULL denetimi,
   çocukları NULL yapar */
```

arama_agacina_ekle(50, &L) adım adım

```
1  boolean arama_agacina_ekle(int x, agac_gost *L) {
▶    agac_gost q = NULL, p = *L, r; // q arka, p ön
3    while (p) {                    // NULL'a dek in
4      q = p;                       // arkayı günle
5      if (x < p->veri) p = p->solcocuk;
6      else if (x > p->veri) p = p->sagcocuk;
7      else { printf("Benzer deger var.\n");
8             return FALSE; }
9    }
10   r = ogeal();                   // çocuklar NULL
11   r->veri = x;
12   if (*L == NULL) *L = r;        // boş ağaç: kök
13   else if (x < q->veri) q->solcocuk = r;
14   else q->sagcocuk = r;
15   return TRUE;
16 }
```



x = 50

p kökü gösteriyor (47), q (arkadaki işaretçi) NULL.

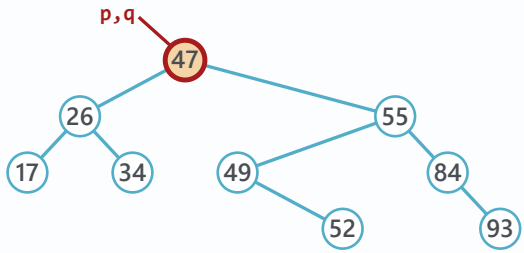
Adım 1 / 13

DEĞİŞKENLER

`x = 50` `p = &düğüm(47)` `q = NULL` `r = ?`

arama_agacina_ekle(50, &L) adım adım

```
1  boolean arama_agacina_ekle(int x, agac_gost *L) {
2    agac_gost q = NULL, p = *L, r; // q arka, p ön
3    while (p) {                    // NULL'a dek in
▶      q = p;                       // arkayı günle
5      if (x < p->veri) p = p->solcocuk;
6      else if (x > p->veri) p = p->sagcocuk;
7      else { printf("Benzer deger var.\n");
8             return FALSE; }
9    }
10   r = ogeal();                   // çocuklar NULL
11   r->veri = x;
12   if (*L == NULL) *L = r;        // boş ağaç: kök
13   else if (x < q->veri) q->solcocuk = r;
14   else q->sagcocuk = r;
15   return TRUE;
16 }
```



x = 50

q = p = 47: arka işaretçi bir adım geriden gelir; eklemeyi q'ya yapacağız.

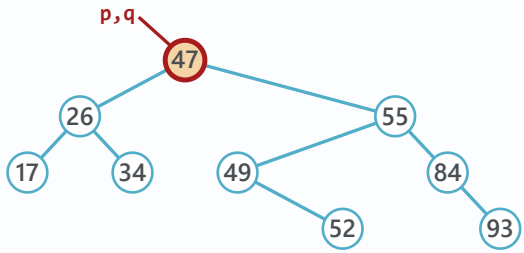
Adım 2 / 13

DEĞİŞKENLER

x = 50 | p = &düğüm(47) | q = &düğüm(47) | r = ?

arama_agacina_ekle(50, &L) adım adım

```
1  boolean arama_agacina_ekle(int x, agac_gost *L) {
2    agac_gost q = NULL, p = *L, r; // q arka, p ön
3    while (p) {                    // NULL'a dek in
4      q = p;                       // arkayı günle
5      if (x < p->veri) p = p->solcocuk;
▶      else if (x > p->veri) p = p->sagcocuk;
7      else { printf("Benzer deger var.\n");
8             return FALSE; }
9    }
10   r = ogeal();                   // çocuklar NULL
11   r->veri = x;
12   if (*L == NULL) *L = r;        // boş ağaç: kök
13   else if (x < q->veri) q->solcocuk = r;
14   else q->sagcocuk = r;
15   return TRUE;
16 }
```



x = 50

50 > 47 → yeri **sağ** alt ağaçta: p = 47->sagcocuk = 55.

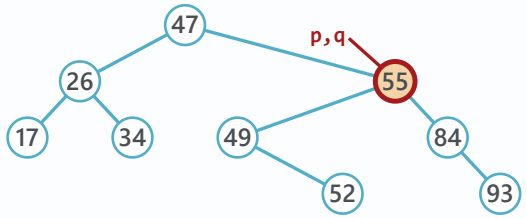
Adım 3 / 13

DEĞİŞKENLER

x = **50** | p = **&düğüm(47)** | q = **&düğüm(47)** | r = **?**

arama_agacina_ekle(50, &L) adım adım

```
1  boolean arama_agacina_ekle(int x, agac_gost *L) {
2    agac_gost q = NULL, p = *L, r; // q arka, p ön
3    while (p) {                    // NULL'a dek in
▶      q = p;                       // arkayı günle
5      if (x < p->veri) p = p->solcocuk;
6      else if (x > p->veri) p = p->sagcocuk;
7      else { printf("Benzer deger var.\n");
8             return FALSE; }
9    }
10   r = ogeal();                   // çocuklar NULL
11   r->veri = x;
12   if (*L == NULL) *L = r;        // boş ağaç: kök
13   else if (x < q->veri) q->solcocuk = r;
14   else q->sagcocuk = r;
15   return TRUE;
16 }
```



x = 50

q = p = 55: arka işaretçi bir adım geriden gelir; eklemeyi q'ya yapacağız.

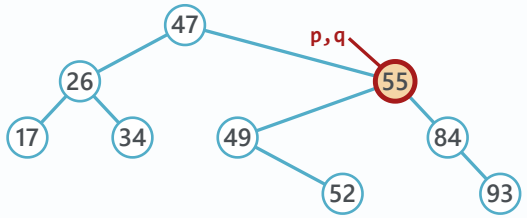
Adım 4 / 13

DEĞİŞKENLER

`x = 50` `p = &düğüm(55)` `q = &düğüm(55)` `r = ?`

arama_agacina_ekle(50, &L) adım adım

```
1  boolean arama_agacina_ekle(int x, agac_gost *L) {
2    agac_gost q = NULL, p = *L, r; // q arka, p ön
3    while (p) {                    // NULL'a dek in
4      q = p;                       // arkayı günle
▶      if (x < p->veri) p = p->solcocuk;
6      else if (x > p->veri) p = p->sagcocuk;
7      else { printf("Benzer deger var.\n");
8             return FALSE; }
9    }
10   r = ogeal();                   // çocuklar NULL
11   r->veri = x;
12   if (*L == NULL) *L = r;        // boş ağaç: kök
13   else if (x < q->veri) q->solcocuk = r;
14   else q->sagcocuk = r;
15   return TRUE;
16 }
```



x = 50

50 < 55 → yeni öğenin yeri **sol** alt ağaçta: p = 55->solcocuk = 49.

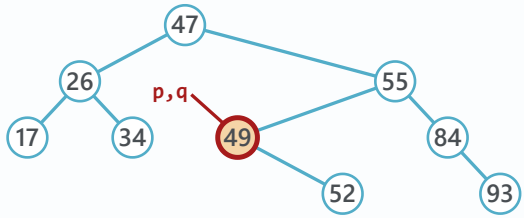
Adım 5 / 13

DEĞİŞKENLER

x = **50** | p = **&düğüm(55)** | q = **&düğüm(55)** | r = **?**

arama_agacina_ekle(50, &L) adım adım

```
1  boolean arama_agacina_ekle(int x, agac_gost *L) {
2    agac_gost q = NULL, p = *L, r; // q arka, p ön
3    while (p) {                    // NULL'a dek in
▶      q = p;                       // arkayı günle
5      if (x < p->veri) p = p->solcocuk;
6      else if (x > p->veri) p = p->sagcocuk;
7      else { printf("Benzer deger var.\n");
8             return FALSE; }
9    }
10   r = ogeal();                   // çocuklar NULL
11   r->veri = x;
12   if (*L == NULL) *L = r;        // boş ağaç: kök
13   else if (x < q->veri) q->solcocuk = r;
14   else q->sagcocuk = r;
15   return TRUE;
16 }
```



x = 50

q = p = 49: arka işaretçi bir adım geriden gelir; eklemeyi q'ya yapacağız.

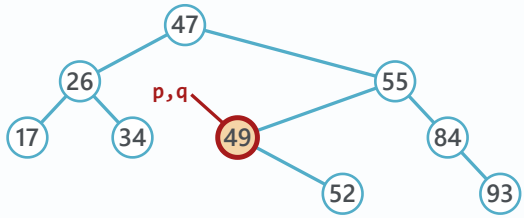
Adım 6 / 13

DEĞİŞKENLER

`x = 50` `p = &düğüm(49)` `q = &düğüm(49)` `r = ?`

arama_agacina_ekle(50, &L) adım adım

```
1  boolean arama_agacina_ekle(int x, agac_gost *L) {
2    agac_gost q = NULL, p = *L, r; // q arka, p ön
3    while (p) {                    // NULL'a dek in
4      q = p;                       // arkayı günle
5      if (x < p->veri) p = p->solcocuk;
▶      else if (x > p->veri) p = p->sagcocuk;
7      else { printf("Benzer deger var.\n");
8             return FALSE; }
9    }
10   r = ogeal();                   // çocuklar NULL
11   r->veri = x;
12   if (*L == NULL) *L = r;        // boş ağaç: kök
13   else if (x < q->veri) q->solcocuk = r;
14   else q->sagcocuk = r;
15   return TRUE;
16 }
```



x = 50

50 > 49 → yeri **sağ** alt ağaçta: p = 49->sagcocuk = 52.

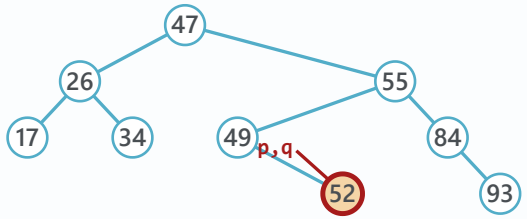
Adım 7 / 13

DEĞİŞKENLER

`x = 50` `p = &düğüm(49)` `q = &düğüm(49)` `r = ?`

arama_agacina_ekle(50, &L) adım adım

```
1  boolean arama_agacina_ekle(int x, agac_gost *L) {
2    agac_gost q = NULL, p = *L, r; // q arka, p ön
3    while (p) {                    // NULL'a dek in
▶      q = p;                       // arkayı günle
5      if (x < p->veri) p = p->solcocuk;
6      else if (x > p->veri) p = p->sagcocuk;
7      else { printf("Benzer deger var.\n");
8             return FALSE; }
9    }
10   r = ogeal();                   // çocuklar NULL
11   r->veri = x;
12   if (*L == NULL) *L = r;        // boş ağaç: kök
13   else if (x < q->veri) q->solcocuk = r;
14   else q->sagcocuk = r;
15   return TRUE;
16 }
```



x = 50

q = p = 52: arka işaretçi bir adım geriden gelir; eklemeyi q'ya yapacağız.

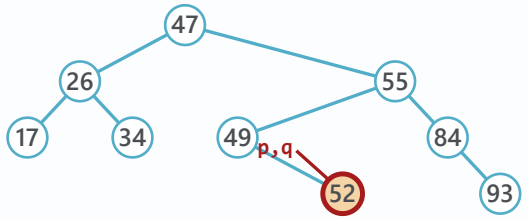
Adım 8 / 13

DEĞİŞKENLER

`x = 50` `p = &düğüm(52)` `q = &düğüm(52)` `r = ?`

arama_agacina_ekle(50, &L) adım adım

```
1  boolean arama_agacina_ekle(int x, agac_gost *L) {
2    agac_gost q = NULL, p = *L, r; // q arka, p ön
3    while (p) {                    // NULL'a dek in
4      q = p;                       // arkayı günle
▶      if (x < p->veri) p = p->solcocuk;
6      else if (x > p->veri) p = p->sagcocuk;
7      else { printf("Benzer deger var.\n");
8             return FALSE; }
9    }
10   r = ogeal();                   // çocuklar NULL
11   r->veri = x;
12   if (*L == NULL) *L = r;        // boş ağaç: kök
13   else if (x < q->veri) q->solcocuk = r;
14   else q->sagcocuk = r;
15   return TRUE;
16 }
```



x = 50

50 < 52 → yeni öğenin yeri **sol** alt ağaçta: p = 52->solcocuk = NULL.

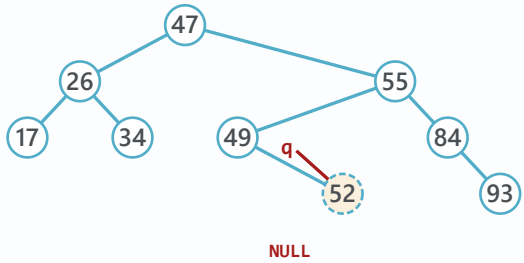
Adım 9 / 13

DEĞİŞKENLER

`x = 50` `p = &düğüm(52)` `q = &düğüm(52)` `r = ?`

arama_agacina_ekle(50, &L) adım adım

```
1  boolean arama_agacina_ekle(int x, agac_gost *L) {
2    agac_gost q = NULL, p = *L, r; // q arka, p ön
▶    while (p) {                    // NULL'a dek in
4      q = p;                       // arkayı günle
5      if (x < p->veri) p = p->solcocuk;
6      else if (x > p->veri) p = p->sagcocuk;
7      else { printf("Benzer deger var.\n");
8             return FALSE; }
9    }
10   r = ogeal();                   // çocuklar NULL
11   r->veri = x;
12   if (*L == NULL) *L = r;        // boş ağaç: kök
13   else if (x < q->veri) q->solcocuk = r;
14   else q->sagcocuk = r;
15   return TRUE;
16 }
```



x = 50

p NULL oldu: döngü biter. Yeni öğe q = 52'nin boş çocuğunun yerine gelecek.

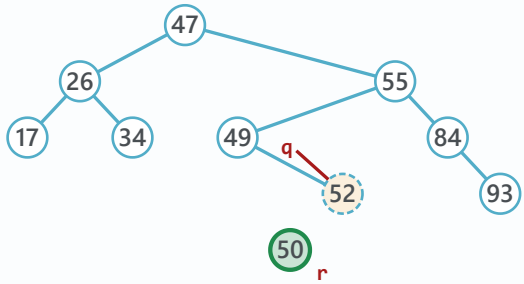
Adım 10 / 13

DEĞİŞKENLER

x = 50 | p = NULL | q = &düğüm(52) | r = ?

arama_agacina_ekle(50, &L) adım adım

```
1  boolean arama_agacina_ekle(int x, agac_gost *L) {
2    agac_gost q = NULL, p = *L, r; // q arka, p ön
3    while (p) {                    // NULL'a dek in
4      q = p;                       // arkayı günle
5      if (x < p->veri) p = p->solcocuk;
6      else if (x > p->veri) p = p->sagcocuk;
7      else { printf("Benzer deger var.\n");
8             return FALSE; }
9    }
▶    r = ogeal();                   // çocuklar NULL
▶    r->veri = x;
12   if (*L == NULL) *L = r;        // boş ağaç: kök
13   else if (x < q->veri) q->solcocuk = r;
14   else q->sagcocuk = r;
15   return TRUE;
16 }
```



x = 50

Yeni düğüm ayrılır (çocukları NULL) ve verisi 50 yapılır.

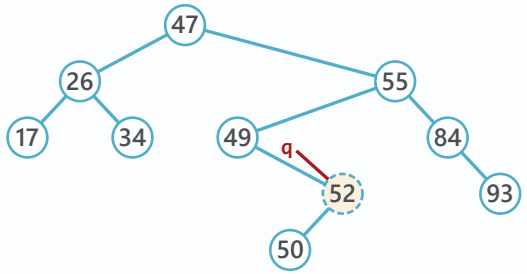
Adım 11 / 13

DEĞİŞKENLER

`x = 50` `p = NULL` `q = &düğüm(52)` `r = &düğüm(50)`

arama_agacina_ekle(50, &L) adım adım

```
1  boolean arama_agacina_ekle(int x, agac_gost *L) {
2    agac_gost q = NULL, p = *L, r; // q arka, p ön
3    while (p) {                    // NULL'a dek in
4      q = p;                       // arkayı günle
5      if (x < p->veri) p = p->solcocuk;
6      else if (x > p->veri) p = p->sagcocuk;
7      else { printf("Benzer deger var.\n");
8             return FALSE; }
9    }
10   r = ogeal();                   // çocuklar NULL
11   r->veri = x;
12   if (*L == NULL) *L = r;        // boş ağaç: kök
▶    else if (x < q->veri) q->solcocuk = r;
14   else q->sagcocuk = r;
15   return TRUE;
16 }
```



x = 50

Ağaç boş değil; 50 < 52 → `q→solcocuk = r`. Bağ kuruldu.

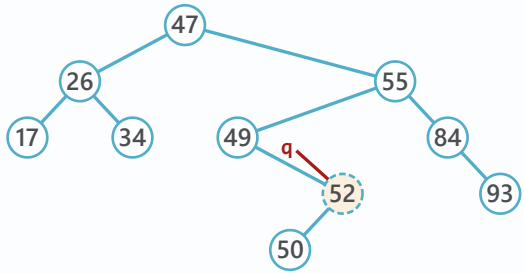
Adım 12 / 13

DEĞİŞKENLER

`x = 50` `p = NULL` `q = &düğüm(52)` `r = &düğüm(50)`

arama_agacina_ekle(50, &L) adım adım

```
1  boolean arama_agacina_ekle(int x, agac_gost *L) {
2    agac_gost q = NULL, p = *L, r; // q arka, p ön
3    while (p) {                    // NULL'a dek in
4      q = p;                       // arkayı günle
5      if (x < p->veri) p = p->solcocuk;
6      else if (x > p->veri) p = p->sagcocuk;
7      else { printf("Benzer deger var.\n");
8             return FALSE; }
9    }
10   r = ogeal();                   // çocuklar NULL
11   r->veri = x;
12   if (*L == NULL) *L = r;        // boş ağaç: kök
13   else if (x < q->veri) q->solcocuk = r;
14   else q->sagcocuk = r;
▶    return TRUE;
16 }
```



x = 50

TRUE döner. Yalnızca **bir** işaretçi değişti; ağacın geri kalanına dokunulmadı.

Adım 13 / 13

DEĞİŞKENLER

x = 50 | p = NULL | q = &düğüm(52) | r = &düğüm(50)

EKLEME

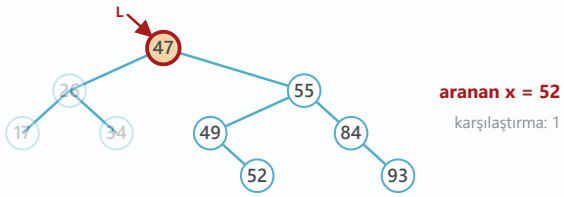
Neden `agac_gost *L`? Maliyet?

- Ağaç boşsa (`*L == NULL`) eklenen öğe **kök** olur: çağıranın kök işaretçisi değişmelidir.
- İşaretçinin kendisini değiştirmek için **adresi** gönderilir (call by reference): `arama_agacina_ekle(x, &kok)` .
- Maliyet: derinlik kadar karşılaştırma → **Ω(lg n) – O(n)**.
- C99 düzeltmesi: `q = NULL` ilk değeri.

```
/* Özyinelemeli alternatif: yeni kökü döndürür */
agac_gost ekle(agac_gost L, int x) {
  if (!L) {
    L = ogeal();
    L->veri = x;
  } else if (x < L->veri)
    L->solcocuk = ekle(L->solcocuk, x);
  else if (x > L->veri)
    L->sagcocuk = ekle(L->sagcocuk, x);
  return L;        /* kok = ekle(kok, x); */
}
```

ARAMA · KODU ADIM ADIM ÇALIŞTIR

arama_agacinda_ara(52, L) — döngüsel



aranan x = 52

karşılaştırma: 1

52 > 47 → sağa. Sol alt ağaç elenir.

Adım 1 / 4

```
1 agac_gost arama_agacinda_ara(int x, agac_gost L) {
2   while (L) {           // uygun alt ağaçta in
3     if (x == L->veri) return L;
4     if (x < L->veri) L = L->solcocuk;
▶     else             L = L->sagcocuk;
6   }
7   return NULL;          // bulunamadı
8 }
```

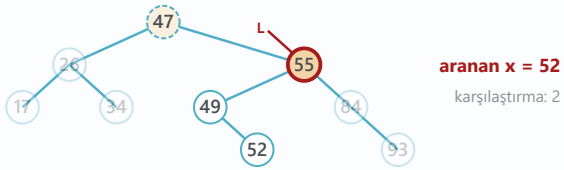
DEĞİŞKENLER

`x = 52` `L = &düğüm(47)`

“Memura görevi tanımlarken müdür çok doğru tanımlamalı. Adam bir daha gelip kapıyı çalmamalı.”

ARAMA · KODU ADIM ADIM ÇALIŞTIR

arama_agacinda_ara(52, L) — döngüsel



52 < 55 → sola. Sağ alt ağaç (soluk) hiç incelenmez: aramanın gücü buradan gelir.

Adım 2 / 4

```
1 agac_gost arama_agacinda_ara(int x, agac_gost L) {
2   while (L) {           // uygun alt ağaçta in
3     if (x == L->veri) return L;
▶     if (x < L->veri) L = L->solcocuk;
5     else             L = L->sagcocuk;
6   }
7   return NULL;          // bulunamadı
8 }
```

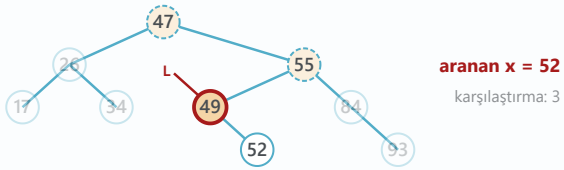
DEĞİŞKENLER

x = 52 L = &düğüm(55)

“Memura görevi tanımlarken müdür çok doğru tanımlamalı. Adam bir daha gelip kapıyı çalmamalı.”

ARAMA · KODU ADIM ADIM ÇALIŞTIR

arama_agacinda_ara(52, L) — döngüsel



aranan x = 52

karşılaştırma: 3

52 > 49 → sağa. Sol alt ağaç elenir.

Adım 3 / 4

```
1 agac_gost arama_agacinda_ara(int x, agac_gost L) {
2   while (L) {           // uygun alt ağaçta in
3     if (x == L->veri) return L;
4     if (x < L->veri) L = L->solcocuk;
▶     else             L = L->sagcocuk;
6   }
7   return NULL;          // bulunamadı
8 }
```

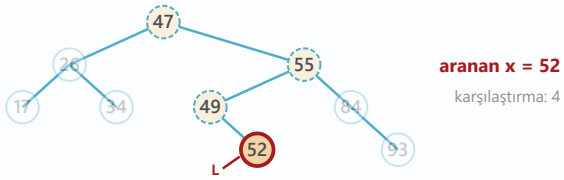
DEĞİŞKENLER

`x = 52` `L = &düğüm(49)`

“Memura görevi tanımlarken müdür çok doğru tanımlamalı. Adam bir daha gelip kapıyı çalmamalı.”

ARAMA · KODU ADIM ADIM ÇALIŞTIR

arama_agacinda_ara(52, L) — döngüsel



52 == 52: **bulundu**, düğümün adresi döndürülür. Toplam 4 karşılaştırma.

Adım 4 / 4

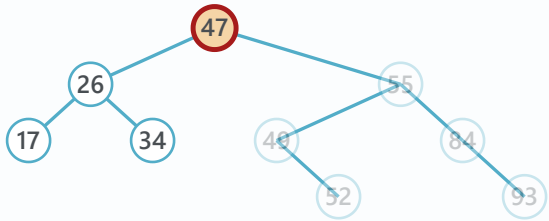
```
1 agac_gost arama_agacinda_ara(int x, agac_gost L) {
2   while (L) {           // uygun alt ağaçta in
▶     if (x == L->veri) return L;
4     if (x < L->veri) L = L->solcocuk;
5     else             L = L->sagcocuk;
6   }
7   return NULL;          // bulunamadı
8 }
```

DEĞİŞKENLER

`x = 52` `L = &düğüm(52)`

“Memura görevi tanımlarken müdür çok doğru tanımlamalı. Adam bir daha gelip kapıyı çalmamalı.”

Özyinelemeli arama: ara(34, L)



x = 34

34 < 47: `return ara(x, L→solcocuk)` — sonucu ne olursa olsun **aynen** yukarı iletilecek.

Adım 1 / 5

```
1 agac_gost ara(int x, agac_gost L) {
2   if (!L) return NULL;          // yok
3   if (x == L->veri) return L;   // bulundu
4   if (x < L->veri)
▶     return ara(x, L->solcocuk);
6   return ara(x, L->sagcocuk);
7 }
```

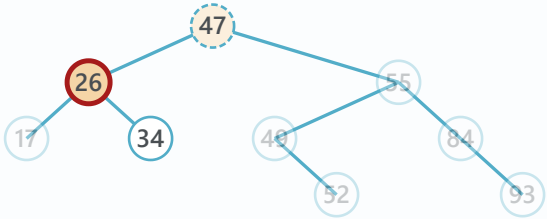
DEĞİŞKENLER

x = 34 | L = &düğüm(47)

ÇAĞRI YIĞINI

ara(34, 47)

Özyinelemeli arama: ara(34, L)



x = 34

34 > 26: `return ara(x, L→sagcocuk)`.

Adım 2 / 5

```
1 agac_gost ara(int x, agac_gost L) {
2   if (!L) return NULL;          // yok
3   if (x == L->veri) return L;   // bulundu
4   if (x < L->veri)
5     return ara(x, L->solcocuk);
▶   return ara(x, L->sagcocuk);
7 }
```

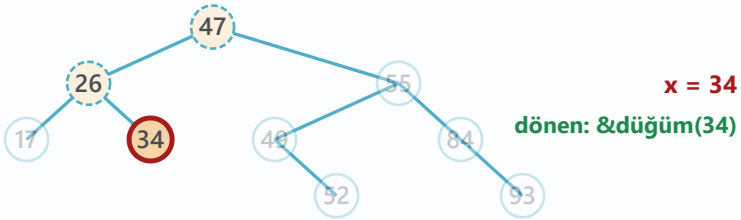
DEĞİŞKENLER

x = 34 | L = &düğüm(26)

ÇAĞRI YIĞINI

- ara(34, 26)
- ara(34, 47)

Özyinelemeli arama: ara(34, L)



34 == 34: bulundu, bu çerçeve L'yi döndürür.

Adım 3 / 5

```
1 agac_gost ara(int x, agac_gost L) {
2   if (!L) return NULL;          // yok
▶   if (x == L->veri) return L;   // bulundu
4   if (x < L->veri)
5     return ara(x, L->solcocuk);
6   return ara(x, L->sagcocuk);
7 }
```

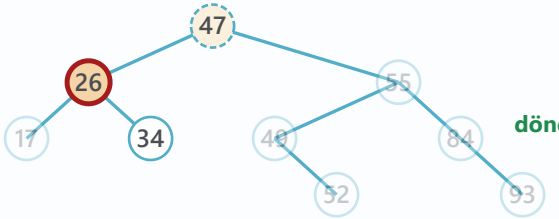
DEĞİŞKENLER

`x = 34` `L = &düğüm(34)`

ÇAĞRI YIĞINI

- `ara(34, 34)`
- `ara(34, 26)`
- `ara(34, 47)`

Özyinelemeli arama: ara(34, L)



x = 34

dönen: &düğüm(34)

Alt çağrı &düğüm(34) döndürdü; ara(34, 26) da onu döndürür.

Adım 4 / 5

```
1 agac_gost ara(int x, agac_gost L) {
2   if (!L) return NULL;          // yok
3   if (x == L->veri) return L;   // bulundu
4   if (x < L->veri)
5     return ara(x, L->solcocuk);
▶   return ara(x, L->sagcocuk);
7 }
```

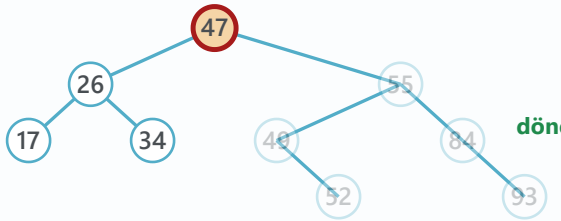
DEĞİŞKENLER

x = 34 | L = &düğüm(26)

ÇAĞRI YIĞINI

- ara(34, 26)
- ara(34, 47)

Özyinelemeli arama: ara(34, L)



x = 34

dönen: &düğüm(34)

Alt çağrı &düğüm(34) döndürdü; ara(34, 47) da onu döndürür.

Adım 5 / 5

```
1 agac_gost ara(int x, agac_gost L) {
2   if (!L) return NULL;          // yok
3   if (x == L->veri) return L;   // bulundu
4   if (x < L->veri)
      return ara(x, L->solcocuk);
6   return ara(x, L->sagcocuk);
7 }
```

DEĞİŞKENLER

x = 34 | L = &düğüm(47)

ÇAĞRI YIĞINI

ara(34, 47)

Örnek: “Bu örnek çalışır mı?”

```
if (L) {
  if (x == L->veri) return L;
  if (x <  L->veri) return arama(x, L->solcocuk);
  return arama(x, L->sagcocuk);
}
return NULL;   /* L == NULL: boş (alt) ağaç */
```

3. satırdaki `if` bloğuna girilirse 4. satırdaki sağ alt ağaç araması **asla** yapılmaz: alt çağrı ne döndürürse o döner.

- **İkili arama ağacında** bu davranış tam istenendir: x küçükse sağda olamaz; kod doğrudur ve O(h)'dir.
- **Sıralı olmayan genel ikili ağaçta** ise x sağ alt ağaçta olabilir; bu kod onu kaçırır.
- Genel ağaç için çözüm iki tarafı da arar → O(n).

HOCADAN “Özyinelemeli kod yazılırken yandaki örnekteki gibi return satırından sonra işlem yapılıyor.”

Örnek çözüm: her ikili ağaçta arama

```
agac_gost ara_ozyineli(int veri, agac_gost L) {
  if (!L) return NULL;              /* ağaç boş mu   */
  if (L->veri == veri) return L;    /* aranan bulundu */
  agac_gost T = ara_ozyineli(veri, L->solcocuk);
  return T ? T : ara_ozyineli(veri, L->sagcocuk);
}
```

- Önce solda arar; bulamazsa (T == NULL) sağda arar.
- Sıralama bilgisi kullanılmaz → en kötü O(n); BST'de gereksiz yere yavaş.
- Ağırlık bulma sınav sorusunun çözümünde bu yordam kullanılır.

HOCADAN "Bu yazdığımız yordamları buradaki herkesin gözü kapalı yazması lazım! Hadi bir gözü açık olsun ☺"

SORU: derinliği döngüsel yazınız

HOCADAN "İteratif kod yazmak acemi işi! ... Sınavda bu şekilde yazılırsa, çalışsa bile puan alamaz. Neden? Çünkü global değişkenler kullanılıyor. — "Ama doğru çalışıyorsa 1 puan da mı vermezsiniz?" — Hayır, 0.25 bile vermem!"

```
/* global değişkensiz döngüsel çözüm:
   düzey düzey gezin, düzeyleri say */
int derinlik_d(agac_gost L) {
  agac_gost k[MAX_SIZE];
  int on = 0, arka = 0, d = 0;
  if (L) k[arka++] = L;
  while (on < arka) {          /* bir düzey */
    int son = arka;
    d++;
    for (; on < son; on++) {
      if (k[on]->solcocuk) k[arka++] = k[on]->solcocuk;
      if (k[on]->sagcocuk) k[arka++] = k[on]->sagcocuk;
    }
  }
  return d;       /* MAX_SIZE >= n olmalı */
}
```

DERİNLİK

Neden global değişken yok?

HOCADAN “Büyük firmalarda büyük işlerin yapılabilmesi için entegre çalışmak gerekir. Birinin yazdığı yordamı ta öbür uçtaki biri kullanıyor. Birbirlerini tanımıyorlar, o ta Hindistan’da! Telefonu açıp “Senin bir max değişkenin var, bu ne oluyor?” diye soramaz ki!”

Yordam kendi içinde tamam olmalı

Girdi parametreden, çıktı dönüş değerinden; gizli durum yok.

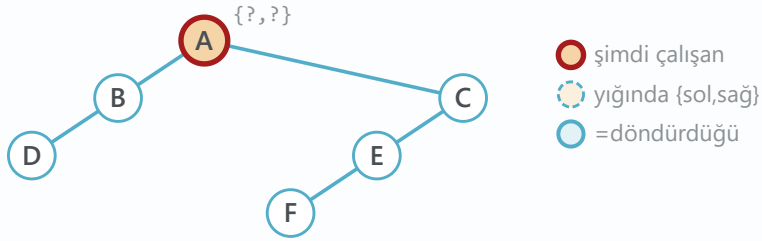
Yerel prototip olur

Yordamın içinde başka bir yordamın prototipi yerel olarak bildirilip kullanılabilir; tanım dışarıda kalır (standart C iç içe yordam tanımına izin vermez).

Özyinelemeli çözüm

Her çağrının kendi yerel sol/sağ değişkeni vardır; global sayaç gerekmez.

Özyinelemeli derinlik: {sol, sağ} çerçeveleri



L = A: önce sol alt ağacın derinliği: `derinlik(B)` çağrılır.

Adım 1 / 11

```
1 int derinlik(agac_gost L) {
2   int sol, sag;
3   if (L) {                    // işaretçi NULL mı
▶     sol = derinlik(L->solcocuk);
5     sag = derinlik(L->sagcocuk);
6     return MAX(sol, sag) + 1;
7   }
8   else return 0;              // uç yaprağa ulaşıldı
9 }
```

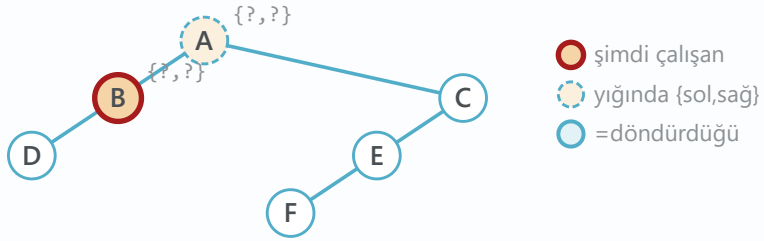
DEĞİŞKENLER

`L = &düğüm(A)` `sol = ?` `sag = ?`

ÇAĞRI YIĞINI

`derinlik(A)`

Özyinelemeli derinlik: {sol, sağ} çerçeveleri



L = B: önce sol alt ağacın derinliği: `derinlik(D)` çağrılır.

Adım 2 / 11

```
1 int derinlik(agac_gost L) {
2   int sol, sag;
3   if (L) {                    // işaretçi NULL mı
▶     sol = derinlik(L->solcocuk);
5     sag = derinlik(L->sagcocuk);
6     return MAX(sol, sag) + 1;
7   }
8   else return 0;              // uç yaprağa ulaşıldı
9 }
```

DEĞİŞKENLER

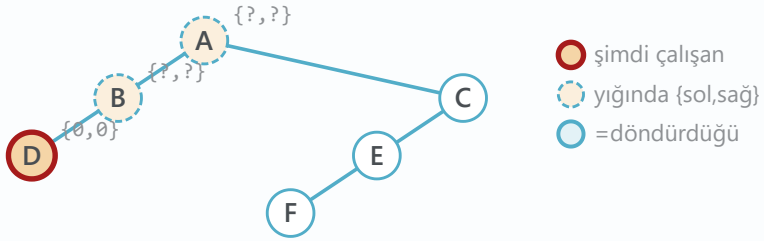
`L = &düğüm(B)` `sol = ?` `sag = ?`

ÇAĞRI YIĞINI

`derinlik(B)`

`derinlik(A)`

Özyinelemeli derinlik: {sol, sağ} çerçeveleri



L = D: sol = 0, sağ = 0 (NULL → 0) → MAX + 1 = **1** döner; yerel sol/sağ silinir.

Adım 3 / 11

```
1 int derinlik(agac_gost L) {
2   int sol, sag;
3   if (L) {                    // işaretçi NULL mı
4     sol = derinlik(L->solcocuk);
5     sag = derinlik(L->sagcocuk);
▶     return MAX(sol, sag) + 1;
7   }
8   else return 0;              // uç yaprağa ulaşıldı
9 }
```

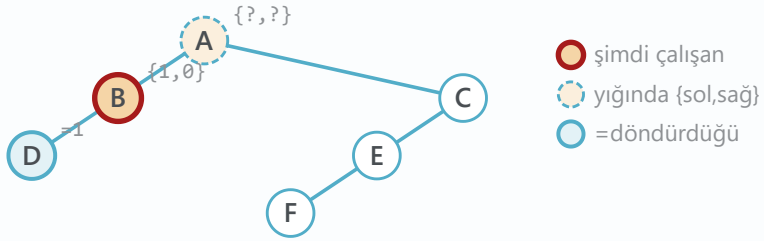
DEĞİŞKENLER

`L = &düğüm(D)` `sol = 0` `sag = 0`

ÇAĞRI YIĞINI

- **derinlik(D)**
- derinlik(B)
- derinlik(A)

Özyinelemeli derinlik: {sol, sağ} çerçeveleri



L = B: sol = 1, sağ = 0 (NULL → 0) → MAX + 1 = **2** döner; yerel sol/sağ silinir.

Adım 4 / 11

```
1 int derinlik(agac_gost L) {
2   int sol, sag;
3   if (L) {                    // işaretçi NULL mı
4     sol = derinlik(L->solcocuk);
5     sag = derinlik(L->sagcocuk);
▶     return MAX(sol, sag) + 1;
7   }
8   else return 0;              // uç yaprağa ulaşıldı
9 }
```

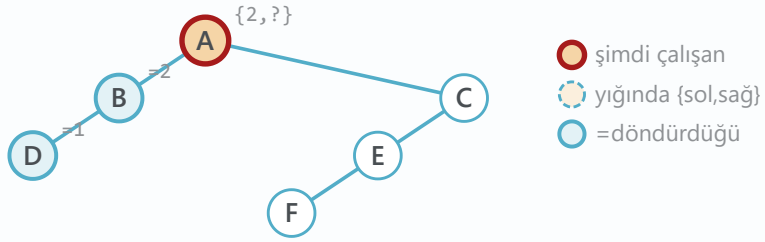
DEĞİŞKENLER

`L = &düğüm(B)` `sol = 1` `sag = 0`

ÇAĞRI YIĞINI

- **derinlik(B)**
- derinlik(A)

Özyinelemeli derinlik: {sol, sağ} çerçeveleri



sol = 2; şimdi `derinlik(C)` .

Adım 5 / 11

```
1 int derinlik(agac_gost L) {
2   int sol, sag;
3   if (L) {                    // işaretçi NULL mı
4     sol = derinlik(L->solcocuk);
▶     sag = derinlik(L->sagcocuk);
6     return MAX(sol, sag) + 1;
7   }
8   else return 0;              // uç yaprağa ulaşıldı
9 }
```

DEĞİŞKENLER

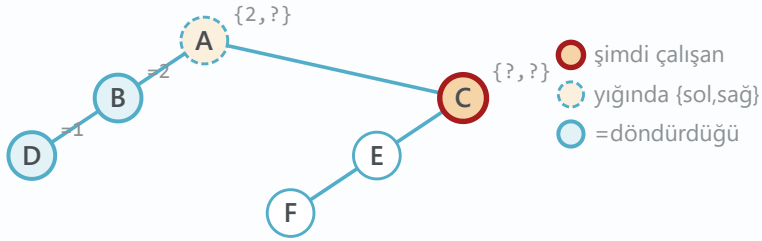
`L = &düğüm(A)` `sol = 2`

`sag = ?`

ÇAĞRI YIĞINI

`derinlik(A)`

Özyinelemeli derinlik: {sol, sağ} çerçeveleri



L = C: önce sol alt ağacın derinliği: `derinlik(E)` çağrılır.

Adım 6 / 11

```
1 int derinlik(agac_gost L) {
2   int sol, sag;
3   if (L) {                    // işaretçi NULL mı
▶     sol = derinlik(L->solcocuk);
5     sag = derinlik(L->sagcocuk);
6     return MAX(sol, sag) + 1;
7   }
8   else return 0;              // uç yaprağa ulaşıldı
9 }
```

DEĞİŞKENLER

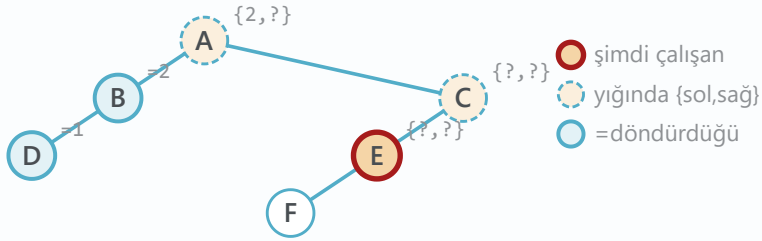
`L = &düğüm(C)` `sol = ?` `sag = ?`

ÇAĞRI YIĞINI

`derinlik(C)`

`derinlik(A)`

Özyinelemeli derinlik: {sol, sağ} çerçeveleri



L = E: önce sol alt ağacın derinliği: `derinlik(F)` çağrılır.

Adım 7 / 11

```
1 int derinlik(agac_gost L) {
2   int sol, sag;
3   if (L) {                    // işaretçi NULL mı
▶     sol = derinlik(L->solcocuk);
5     sag = derinlik(L->sagcocuk);
6     return MAX(sol, sag) + 1;
7   }
8   else return 0;              // uç yaprağa ulaşıldı
9 }
```

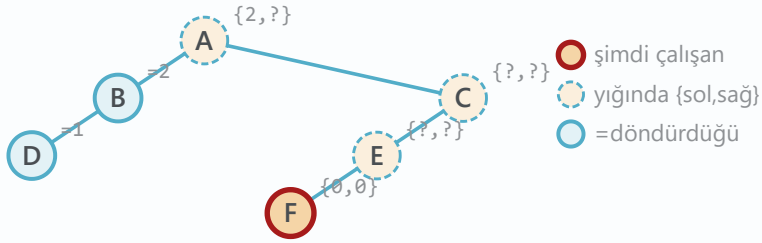
DEĞİŞKENLER

`L = &düğüm(E)` `sol = ?` `sag = ?`

ÇAĞRI YIĞINI

- **derinlik(E)**
- derinlik(C)
- derinlik(A)

Özyinelemeli derinlik: {sol, sağ} çerçeveleri



L = F: sol = 0, sağ = 0 (NULL → 0) → MAX + 1 = **1** döner; yerel sol/sağ silinir.

Adım 8 / 11

```
1 int derinlik(agac_gost L) {
2   int sol, sag;
3   if (L) {                    // işaretçi NULL mı
4     sol = derinlik(L->solcocuk);
5     sag = derinlik(L->sagcocuk);
▶     return MAX(sol, sag) + 1;
7   }
8   else return 0;              // uç yaprağa ulaşıldı
9 }
```

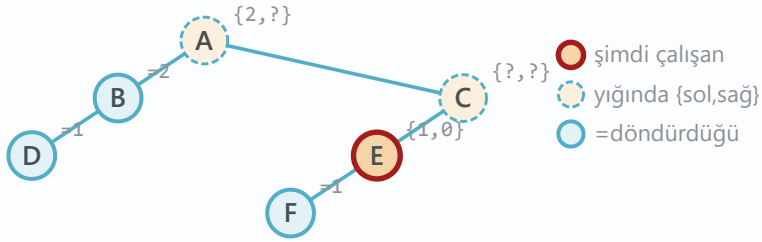
DEĞİŞKENLER

`L = &düğüm(F)` `sol = 0` `sag = 0`

ÇAĞRI YIĞINI

- **derinlik(F)**
- derinlik(E)
- derinlik(C)
- derinlik(A)

Özyinelemeli derinlik: {sol, sağ} çerçeveleri



L = E: sol = 1, sağ = 0 (NULL → 0) → MAX + 1 = **2** döner; yerel sol/sağ silinir.

Adım 9 / 11

```
1 int derinlik(agac_gost L) {
2   int sol, sag;
3   if (L) {                    // işaretçi NULL mı
4     sol = derinlik(L->solcocuk);
5     sag = derinlik(L->sagcocuk);
▶     return MAX(sol, sag) + 1;
7   }
8   else return 0;              // uç yaprağa ulaşıldı
9 }
```

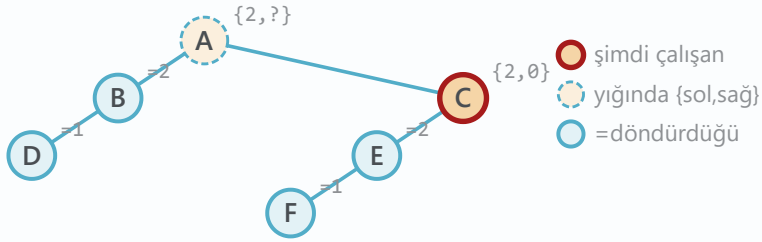
DEĞİŞKENLER

`L = &düğüm(E)` `sol = 1` `sag = 0`

ÇAĞRI YIĞINI

- **derinlik(E)**
- derinlik(C)
- derinlik(A)

Özyinelemeli derinlik: {sol, sağ} çerçeveleri



L = C: sol = 2, sağ = 0 (NULL → 0) → MAX + 1 = **3** döner; yerel sol/sağ silinir.

Adım 10 / 11

```
1 int derinlik(agac_gost L) {
2   int sol, sag;
3   if (L) {                    // işaretçi NULL mı
4     sol = derinlik(L->solcocuk);
5     sag = derinlik(L->sagcocuk);
▶     return MAX(sol, sag) + 1;
7   }
8   else return 0;              // uç yaprağa ulaşıldı
9 }
```

DEĞİŞKENLER

`L = &düğüm(C)` `sol = 2`

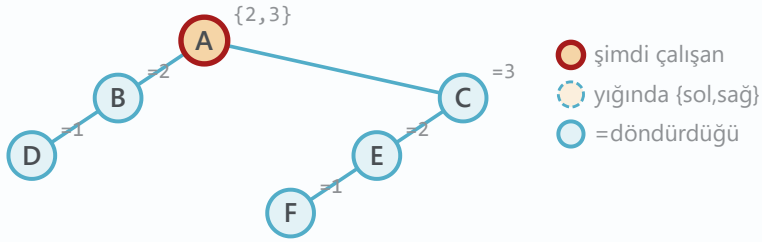
`sag = 0`

ÇAĞRI YIĞINI

derinlik(C)

derinlik(A)

Özyinelemeli derinlik: {sol, sağ} çerçeveleri



L = A: sol = 2, sağ = 3 → MAX + 1 = **4** döner; yerel sol/sağ silinir.

Adım 11 / 11

```
1 int derinlik(agac_gost L) {
2   int sol, sag;
3   if (L) {                    // işaretçi NULL mı
4     sol = derinlik(L->solcocuk);
5     sag = derinlik(L->sagcocuk);
▶     return MAX(sol, sag) + 1;
7   }
8   else return 0;              // uç yaprağa ulaşıldı
9 }
```

DEĞİŞKENLER

`L = &düğüm(A)` `sol = 2` `sag = 3`

ÇAĞRI YIĞINI

`derinlik(A)`

DERİNLİK

Derinlik: problem yeterince küçüldü mü?

- Ağacın derinliği, sol ve sağ alt ağaçlarının en derin olanının **1 fazlasıdır**: `MAX(sol, sağ) + 1`.
- Her düğümün kendi 2 yerel değişkeni vardır: **sol** ve **sağ** derinlik. NULL ile yapılan çağrıların yerelleri kullanılmadan silinir.
- Zaman O(n), bellek O(h). `#define MAX(a,b) ((a) > (b) ? (a) : (b))`

```
int derinlik(agac_gost L) {    /* tek satır */
  return !L ? 0 : MAX(derinlik(L->solcocuk),
                      derinlik(L->sagcocuk)) + 1;
}
```

`MAX` makrosu argümanını iki kez değerlendirir; bu tek satırlıkta özyinelemeli çağrılar iki kez yapılır → üstel süre! Bu yüzden makro yerine `max()` işlevi kullanılır (sınav çözümündeki gibi).

HOCADAN "Problem yeterince küçüldü mü? Başta da NULL gelebilir!"

HOCADAN "Bu iki yaklaşımı (özyinelemeli düşünme, özyinelemeli bağıntıları çözme) bizim mühendis arkadaşlara kabul ettirmek zordur. Adamın bir düşünce tarzı var, ne güzel kod yazıyor!"

DERİNLİK

Ek parametreler koyma mevzusu

> **“If you have a procedure with 10 parameters, you probably missed some.”**
> — Alan J. Perlis

Soru: “Hocam, `int derinlik(agac_gost L, int a)` diye fazladan bir parametre ekleyip `derinlik(L, 0)` diye çağırsak olur mu?”

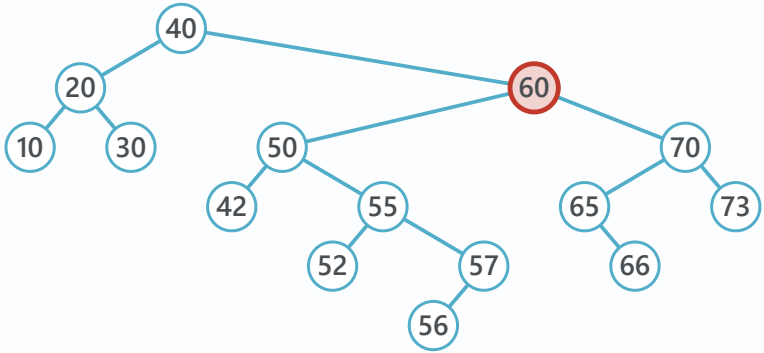
HOCADAN “Neden fazladan parametre tanımlanıp ilk çağrıda sıfır aktarılıyor? Yordamı kullanana ne denecek? —“a’ya sıfır aktarmayı sakın ha unutma!” Sıralama yordamı yazılmış; 0 aktarınca sıralıyor da, 1 aktarınca sıralayamıyor… ☺ Olur mu?”

Ama şu olur: küçükten büyüğe mi büyükten küçüğe mi sıralanacağını belirten (−1/+1, '<'/'>') ya da verinin türünü bildiren bir parametre. Parametre **davranışı seçmeli**, gizli bir başlangıç değeri taşımamalı.

SİLME

İkili arama ağacından öğe silme

- Silerken ağacın **ikili arama ağacı özelliği** bozulmamalı.
- **Çocuksuz** (yaprak): ebeveynin ilgili işaretçisi NULL yapılır.
- **Tek çocuklu**: ebeveynin işaretçisi silinenin tek çocuğunu gösterir.
- **İki çocuklu**: farklı yöntemler var; örnek ağaçtan **60**'ı silelim.

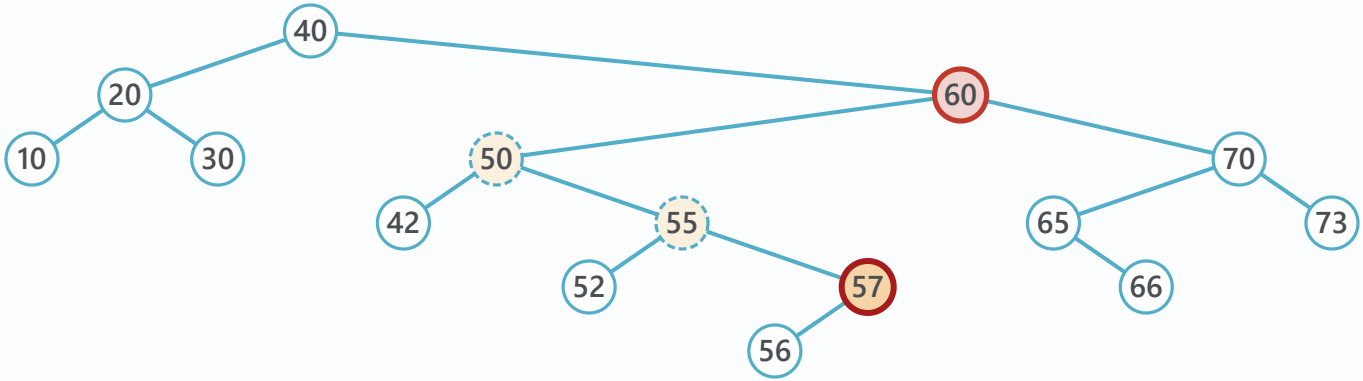


silinecek öğe: 60

Örnek ağaç: 40, 20, 60, 10, 30, 50, 70, 42, 55, 65, 73, 52, 57, 66, 56 sırasıyla eklenmiştir (derinlik 6).

SİLME

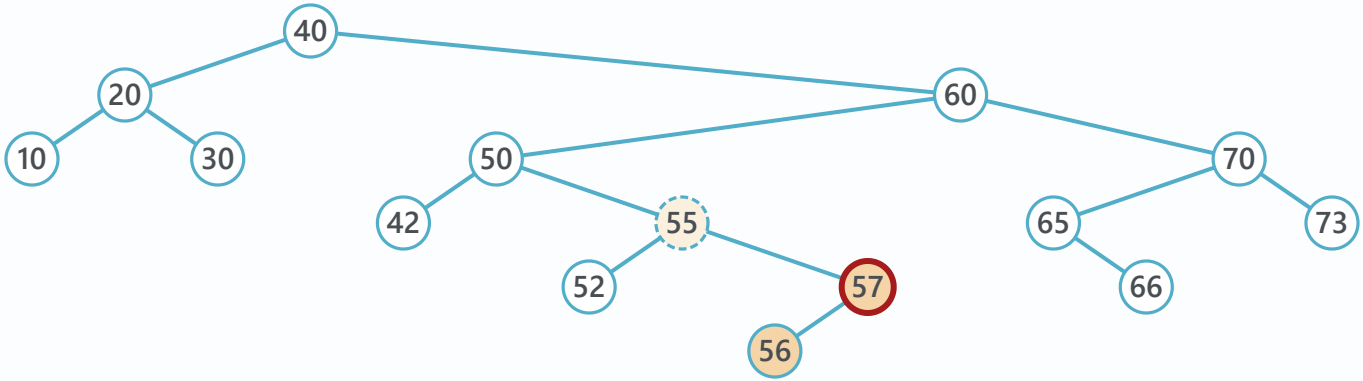
1. yöntem: sol alt ağacın en büyüğü



sol alt ağaçta sürekli sağa: 50 → 55 → 57

Sol alt ağaçtaki en büyük değer: sol alt ağaçta **sağ** işaret alanları üzerinden ilerlenir; sağ çocuğu olmayan öğe (57) en büyüktür.

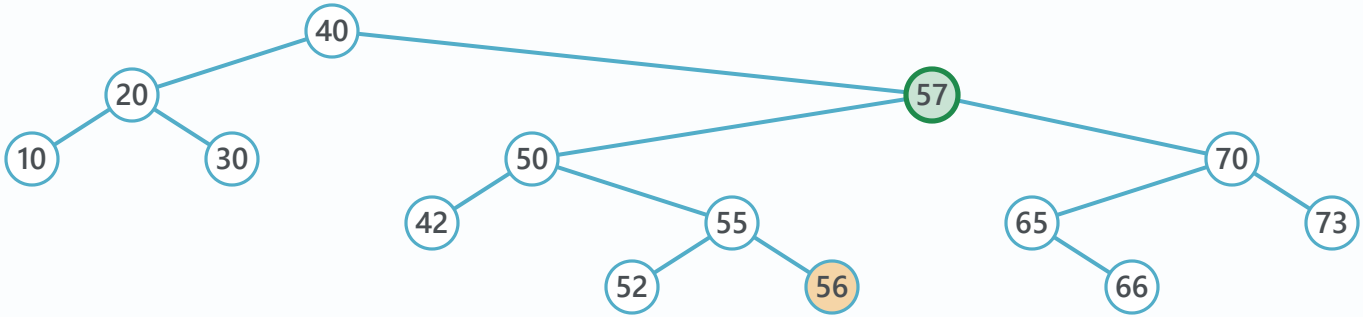
1. yöntem: sol alt ağacın en büyüğü



57'nin sol çocuğu 56 var!

Bu öğenin bir sol alt ağacı olabilir (57'nin 56'sı). Bu sol çocuk, taşınacak öğenin ebeveyninin **sağ** bağına aktarılmalı (55 → 56).

1. yöntem: sol alt ağacın en büyüğü

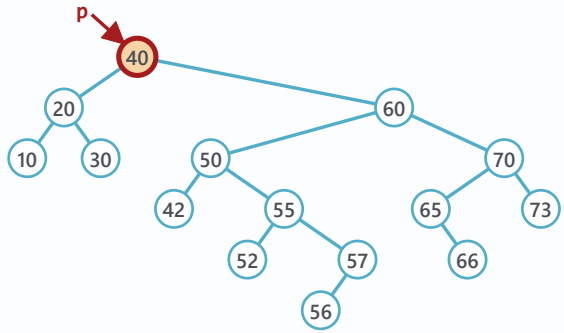


veri kopyalama: 1 bağ günlenir · derinlik 6 → 5

İkinci yol: 60 free edilmez; veri alanına 57 yazılır, yalnızca 55'in sağ bağı 56'ya günlenir → **1 bağ**. **Birinci yol** (düğümü taşımak) 4 bağ ister: 40→57, 55→56, 57.sol→50, 57.sağ→70.

sil(60, &L): iki çocuklu düğüm

```
1  void sil(int x, agac_gost *L) {
▶    agac_gost p = *L, *bag = L;  // bag: p'ye giden
3    while (p && p->veri != x) {  // x'i ara
4      bag = (x < p->veri) ? &p->solcocuk
5                          : &p->sagcocuk;
6      p = *bag;
7    }
8    if (!p) return;              // x yok
9    if (p->solcocuk && p->sagcocuk) {  // 2 çocuk
10     agac_gost *b = &p->solcocuk;     // sol alt
11     while ((*b)->sagcocuk)           // ağacın en
12       b = &(*b)->sagcocuk;           // büyüğü
13     p->veri = (*b)->veri;      // veriyi taşı
14     bag = b;  p = *b;          // onu sileceğiz
15   }
16   *bag = p->solcocuk ? p->solcocuk : p->sagcocuk;
17   free(p);
18 }
```



p kökü, bag kök işaretçisinin **adresini** tutar: silinecek düğüme giden bağı değiştirebilmek için.

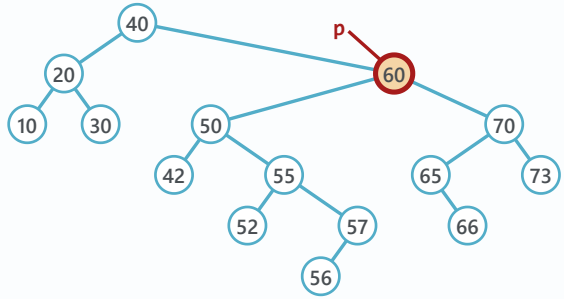
Adım 1 / 11

DEĞİŞKENLER

`x = 60` `p = &düğüm(40)` `bag = &L (kök)`

sil(60, &L): iki çocuklu düğüm

```
1  void sil(int x, agac_gost *L) {
2    agac_gost p = *L, *bag = L;  // bag: p'ye giden
3    while (p && p->veri != x) {  // x'i ara
▶      bag = (x < p->veri) ? &p->solcocuk
▶                          : &p->sagcocuk;
▶      p = *bag;
7    }
8    if (!p) return;              // x yok
9    if (p->solcocuk && p->sagcocuk) {  // 2 çocuk
10     agac_gost *b = &p->solcocuk;     // sol alt
11     while ((*b)->sagcocuk)           // ağacın en
12       b = &(*b)->sagcocuk;           // büyüğü
13     p->veri = (*b)->veri;      // veriyi taşı
14     bag = b;  p = *b;          // onu sileceğiz
15   }
16   *bag = p->solcocuk ? p->solcocuk : p->sagcocuk;
17   free(p);
18 }
```



60 > 40 → bag = &40->sagcocuk, p = 60.

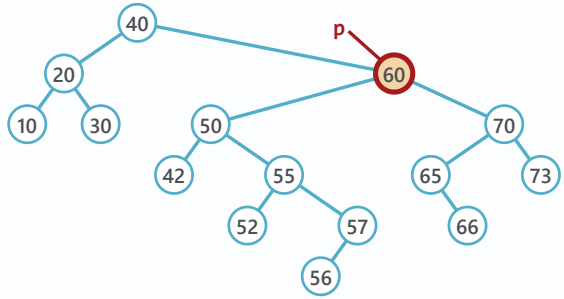
Adım 2 / 11

DEĞİŞKENLER

x = 60 | p = &düğüm(60) | bag = &40→sagcocuk

sil(60, &L): iki çocuklu düğüm

```
1  void sil(int x, agac_gost *L) {
2    agac_gost p = *L, *bag = L;  // bag: p'ye giden
3    while (p && p->veri != x) {  // x'i ara
4      bag = (x < p->veri) ? &p->solcocuk
5                          : &p->sagcocuk;
6      p = *bag;
7    }
8    if (!p) return;              // x yok
▶    if (p->solcocuk && p->sagcocuk) {  // 2 çocuk
10     agac_gost *b = &p->solcocuk;     // sol alt
11     while ((*b)->sagcocuk)           // ağacın en
12       b = &(*b)->sagcocuk;           // büyüğü
13     p->veri = (*b)->veri;      // veriyi taşı
14     bag = b;  p = *b;          // onu sileceğiz
15   }
16   *bag = p->solcocuk ? p->solcocuk : p->sagcocuk;
17   free(p);
18 }
```



60 bulundu ve **iki çocuğu** var (50, 70) → sol alt ağacın en büyüğü aranır.

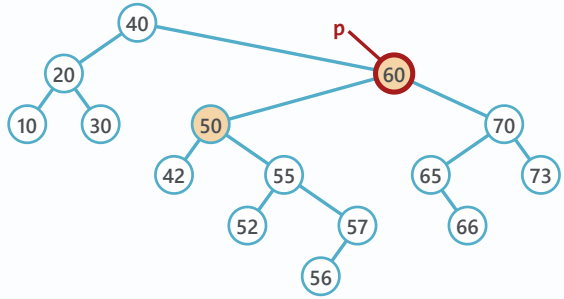
Adım 3 / 11

DEĞİŞKENLER

`x = 60` `p = &düğüm(60)` `bag = &40→sagcocuk`

sil(60, &L): iki çocuklu düğüm

```
1  void sil(int x, agac_gost *L) {
2    agac_gost p = *L, *bag = L;  // bag: p'ye giden
3    while (p && p->veri != x) {  // x'i ara
4      bag = (x < p->veri) ? &p->solcocuk
5                          : &p->sagcocuk;
6      p = *bag;
7    }
8    if (!p) return;              // x yok
9    if (p->solcocuk && p->sagcocuk) {  // 2 çocuk
▶      agac_gost *b = &p->solcocuk;     // sol alt
11     while ((*b)->sagcocuk)           // ağacın en
12       b = &(*b)->sagcocuk;           // büyüğü
13     p->veri = (*b)->veri;      // veriyi taşı
14     bag = b;  p = *b;          // onu sileceğiz
15   }
16   *bag = p->solcocuk ? p->solcocuk : p->sagcocuk;
17   free(p);
18 }
```



b = &60->solcocuk → *b = 50.

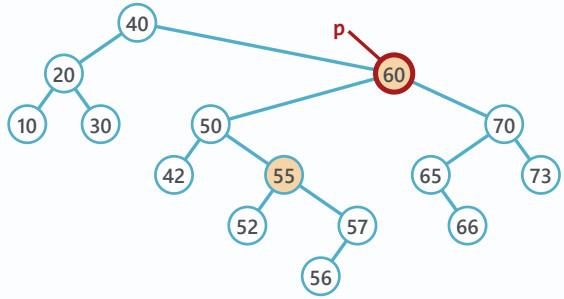
Adım 4 / 11

DEĞİŞKENLER

x = **60** | p = **&düğüm(60)** | bag = **&40→sagcocuk**

sil(60, &L): iki çocuklu düğüm

```
1  void sil(int x, agac_gost *L) {
2    agac_gost p = *L, *bag = L;  // bag: p'ye giden
3    while (p && p->veri != x) {  // x'i ara
4      bag = (x < p->veri) ? &p->solcocuk
5                          : &p->sagcocuk;
6      p = *bag;
7    }
8    if (!p) return;              // x yok
9    if (p->solcocuk && p->sagcocuk) {  // 2 çocuk
10     agac_gost *b = &p->solcocuk;     // sol alt
▶      while ((*b)->sagcocuk)           // ağacın en
▶        b = &(*b)->sagcocuk;           // büyüğü
13     p->veri = (*b)->veri;       // veriyi taşı
14     bag = b;  p = *b;           // onu sileceğiz
15   }
16   *bag = p->solcocuk ? p->solcocuk : p->sagcocuk;
17   free(p);
18 }
```



*b'nin sağı var → b = &50->sagcocuk, *b = 55.

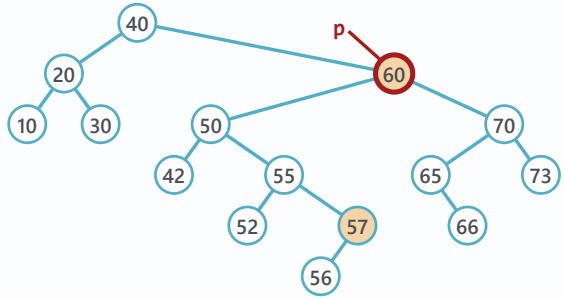
Adım 5 / 11

DEĞİŞKENLER

`x = 60` `p = &düğüm(60)` `bag = &40→sagcocuk`

sil(60, &L): iki çocuklu düğüm

```
1  void sil(int x, agac_gost *L) {
2    agac_gost p = *L, *bag = L;  // bag: p'ye giden
3    while (p && p->veri != x) {  // x'i ara
4      bag = (x < p->veri) ? &p->solcocuk
5                          : &p->sagcocuk;
6      p = *bag;
7    }
8    if (!p) return;              // x yok
9    if (p->solcocuk && p->sagcocuk) {  // 2 çocuk
10     agac_gost *b = &p->solcocuk;     // sol alt
▶      while ((*b)->sagcocuk)           // ağacın en
▶        b = &(*b)->sagcocuk;           // büyüğü
13     p->veri = (*b)->veri;       // veriyi taşı
14     bag = b;  p = *b;           // onu sileceğiz
15   }
16   *bag = p->solcocuk ? p->solcocuk : p->sagcocuk;
17   free(p);
18 }
```



*b'nin sağı var → b = &55->sagcocuk, *b = 57.

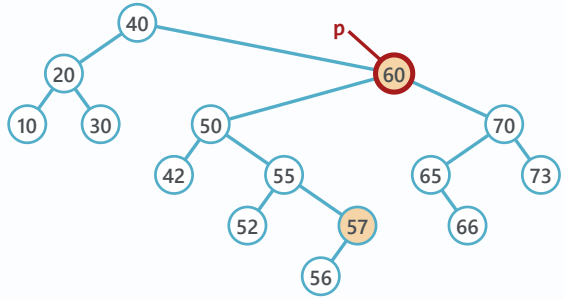
Adım 6 / 11

DEĞİŞKENLER

`x = 60` `p = &düğüm(60)` `bag = &40→sagcocuk`

sil(60, &L): iki çocuklu düğüm

```
1  void sil(int x, agac_gost *L) {
2    agac_gost p = *L, *bag = L;  // bag: p'ye giden
3    while (p && p->veri != x) {  // x'i ara
4      bag = (x < p->veri) ? &p->solcocuk
5                          : &p->sagcocuk;
6      p = *bag;
7    }
8    if (!p) return;              // x yok
9    if (p->solcocuk && p->sagcocuk) {  // 2 çocuk
10     agac_gost *b = &p->solcocuk;     // sol alt
▶      while ((*b)->sagcocuk)           // ağacın en
12       b = &(*b)->sagcocuk;           // büyüğü
13     p->veri = (*b)->veri;      // veriyi taşı
14     bag = b;  p = *b;          // onu sileceğiz
15   }
16   *bag = p->solcocuk ? p->solcocuk : p->sagcocuk;
17   free(p);
18 }
```



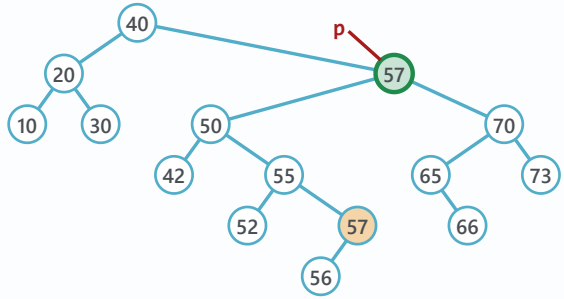
57'nin sağ çocuğu yok: sol alt ağacın en büyüğü **57**.

DEĞİŞKENLER

`x = 60` `p = &düğüm(60)` `bag = &40→sagcocuk`

sil(60, &L): iki çocuklu düğüm

```
1  void sil(int x, agac_gost *L) {
2    agac_gost p = *L, *bag = L;  // bag: p'ye giden
3    while (p && p->veri != x) {  // x'i ara
4      bag = (x < p->veri) ? &p->solcocuk
5                          : &p->sagcocuk;
6      p = *bag;
7    }
8    if (!p) return;              // x yok
9    if (p->solcocuk && p->sagcocuk) {  // 2 çocuk
10     agac_gost *b = &p->solcocuk;     // sol alt
11     while ((*b)->sagcocuk)           // ağacın en
12       b = &(*b)->sagcocuk;           // büyüğü
▶      p->veri = (*b)->veri;      // veriyi taşı
14     bag = b;  p = *b;          // onu sileceğiz
15   }
16   *bag = p->solcocuk ? p->solcocuk : p->sagcocuk;
17   free(p);
18 }
```



60'ın veri alanına 57 yazılır (düğüm taşınmaz, 1. yöntemin 2. yolu).

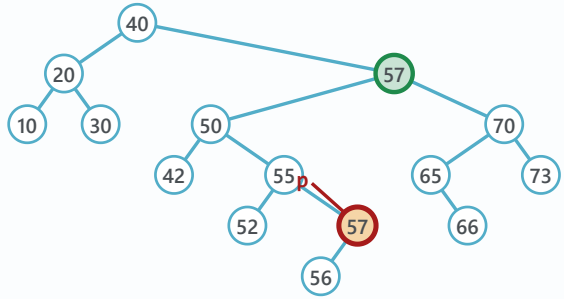
Adım 8 / 11

DEĞİŞKENLER

`x = 60` `p = &düğüm(57)` `bag = &40→sagcocuk`

sil(60, &L): iki çocuklu düğüm

```
1  void sil(int x, agac_gost *L) {
2    agac_gost p = *L, *bag = L;  // bag: p'ye giden
3    while (p && p->veri != x) {  // x'i ara
4      bag = (x < p->veri) ? &p->solcocuk
5                          : &p->sagcocuk;
6      p = *bag;
7    }
8    if (!p) return;              // x yok
9    if (p->solcocuk && p->sagcocuk) {  // 2 çocuk
10     agac_gost *b = &p->solcocuk;     // sol alt
11     while ((*b)->sagcocuk)           // ağacın en
12       b = &(*b)->sagcocuk;           // büyüğü
13     p->veri = (*b)->veri;      // veriyi taşı
▶      bag = b;  p = *b;          // onu sileceğiz
15   }
16   *bag = p->solcocuk ? p->solcocuk : p->sagcocuk;
17   free(p);
18 }
```



Artık silinecek olan eski 57 düğümü; bag = &55->sagcocuk.

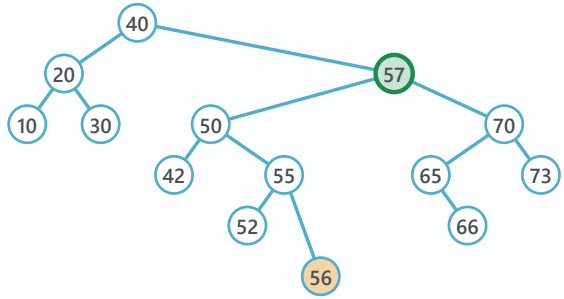
Adım 9 / 11

DEĞİŞKENLER

`x = 60` `p = &düğüm(57)` `bag = &55→sagcocuk`

sil(60, &L): iki çocuklu düğüm

```
1  void sil(int x, agac_gost *L) {
2    agac_gost p = *L, *bag = L;  // bag: p'ye giden
3    while (p && p->veri != x) {  // x'i ara
4      bag = (x < p->veri) ? &p->solcocuk
5                          : &p->sagcocuk;
6      p = *bag;
7    }
8    if (!p) return;              // x yok
9    if (p->solcocuk && p->sagcocuk) {  // 2 çocuk
10     agac_gost *b = &p->solcocuk;     // sol alt
11     while ((*b)->sagcocuk)           // ağacın en
12       b = &(*b)->sagcocuk;           // büyüğü
13     p->veri = (*b)->veri;      // veriyi taşı
14     bag = b;  p = *b;          // onu sileceğiz
15   }
▶  *bag = p->solcocuk ? p->solcocuk : p->sagcocuk;
17   free(p);
18 }
```



*bag = 57'nin tek çocuğu (56): 55'in sağ bağı artık 56'yı gösterir.

Adım 10 / 11

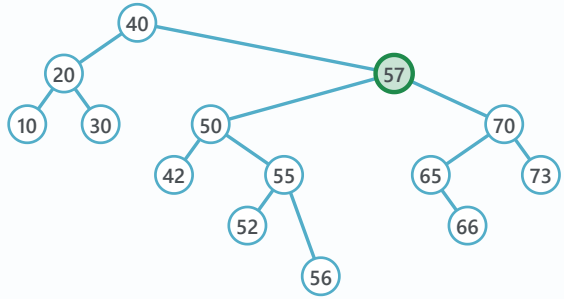
DEĞİŞKENLER

x = **60** p = **NULL**

bag = **&55→sagcocuk**

sil(60, &L): iki çocuklu düğüm

```
1  void sil(int x, agac_gost *L) {
2    agac_gost p = *L, *bag = L;  // bag: p'ye giden
3    while (p && p->veri != x) {  // x'i ara
4      bag = (x < p->veri) ? &p->solcocuk
5                          : &p->sagcocuk;
6      p = *bag;
7    }
8    if (!p) return;              // x yok
9    if (p->solcocuk && p->sagcocuk) {  // 2 çocuk
10     agac_gost *b = &p->solcocuk;     // sol alt
11     while ((*b)->sagcocuk)           // ağacın en
12       b = &(*b)->sagcocuk;           // büyüğü
13     p->veri = (*b)->veri;      // veriyi taşı
14     bag = b;  p = *b;          // onu sileceğiz
15   }
16   *bag = p->solcocuk ? p->solcocuk : p->sagcocuk;
▶    free(p);
18 }
```



Eski düğüm `free` edilir. Ağaç yine ikili arama ağacı; derinlik 6 → 5.

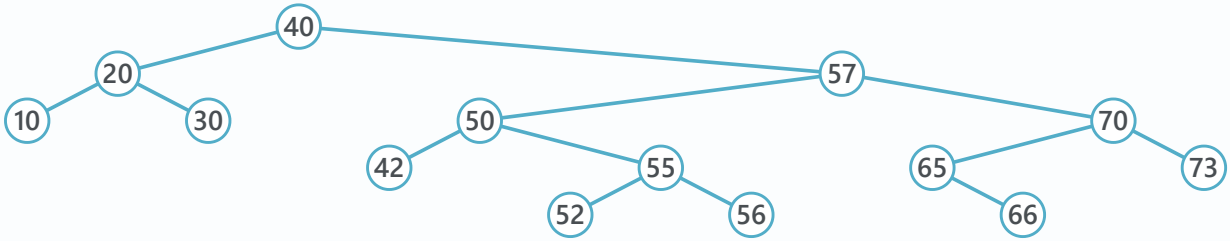
Adım 11 / 11

DEĞİŞKENLER

`x = 60` `p = NULL` `bag = –`

SİLME

Dört yöntem ve derinliğe etkisi



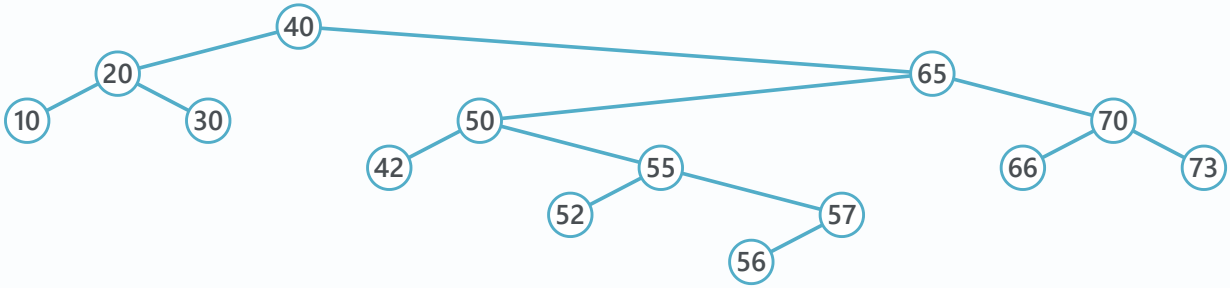
1. yöntem (sol alt ağacın en büyüğü): derinlik 6 → 5

1. yöntem: 57 yukarı, 56 55'in sağına. Derinlik 5'e iner.

Adım 1 / 4

HOCADAN “Sildiğimiz öğenin kökü olduğu ağacın olabildiğince dengeli olmasını sağlayacak şekilde silmeliyiz.”

Dört yöntem ve derinliğe etkisi

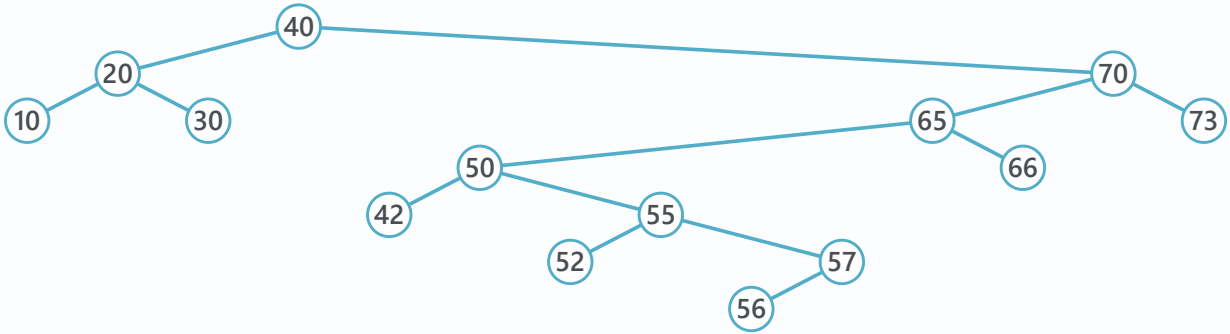


2. yöntem (sağ alt ağacın en küçüğü): derinlik 6 → 6

2. yöntem: sağ alt ağacın en küçüğü 65 taşınır; 65'in sağ çocuğu 66, 70'in soluna bağlanır. Derinlik 6.

HOCADAN "Sildiğimiz öğenin kökü olduğu ağacın olabildiğince dengeli olmasını sağlayacak şekilde silmeliyiz."

Dört yöntem ve derinliğe etkisi

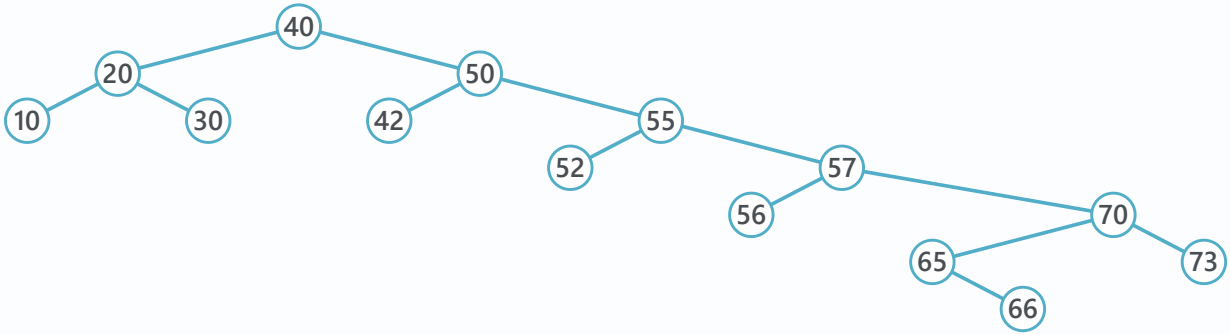


3. yöntem (sağ çocuk yukarı kayar): derinlik 6 → 7

3. yöntem: silinenin yerine sağ çocuğu (70) kayar; 50 alt ağacı, 70 alt ağacının en küçüğünün (65) soluna asılır. Derinlik 7.

HOCADAN "Sildiğimiz öğenin kökü olduğu ağacın olabildiğince dengeli olmasını sağlayacak şekilde silmeliyiz."

Dört yöntem ve derinliğe etkisi



4. yöntem (sol çocuk yukarı kayar): derinlik 6 → 7

4. yöntem: yerine sol çocuğu (50) kayar; 70 alt ağacı, 50 alt ağacının en büyüğünün (57) sağına asılır. Derinlik 7 (40-50-55-57-70-65-66 yolu 7 düğümdür).

HOCADAN "Sildiğimiz öğenin kökü olduğu ağacın olabildiğince dengeli olmasını sağlayacak şekilde silmeliyiz."

Dengesizliğin maliyeti

n	dengeli: $\lceil \lg(n+1) \rceil$	çarpık: n
1.000	10	1.000
1.000.000	20	1.000.000
2.000.000	21	2.000.000
10^9	30	10^9

- Rastgele sırada eklenen n anahtarlı BST'nin **ortalama** derinliği Θ(lg n)'dir; ortalama arama ≈ 1,39 lg n karşılaştırma (Sedgewick; CLRS 12.4).
- Ama **en kötü durum** (sıralı veya neredeyse sıralı veri, ki gerçekte sık görülür) Θ(n)'dir.
- Çözüm: her ekleme/silmeden sonra yerel **döndürmelerle** derinliği O(lg n) tutan ağaçlar: AVL, kırmızı-siyah, B-ağacı.

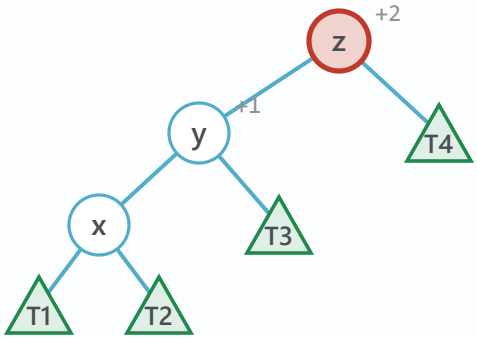
HOCADAN "O(n) ile O(lg n) arasındaki fark, 2.000.000 ile 21 arasındaki fark gibidir."

AVL ağacı (Adelson-Velski ve Landis, 1962)

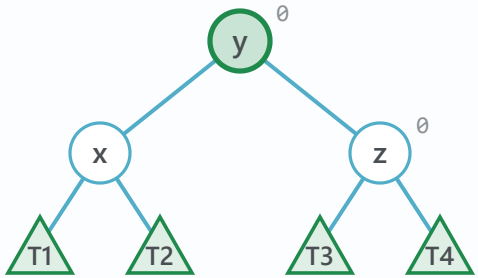
- Her düğümde **denge faktörü** = yük(sol) − yük(sağ) ∈ {−1, 0, +1}.
- Yükseklik en çok ≈ 1,44 lg(n+2): arama, ekleme, silme **O(lg n)**.
- Ekleme sıradan BST eklemesiyle yapılır; sonra eklenen yoldan köke doğru denge faktörleri güncellenir.
- Faktörü ±2 olan **ilk** ata bulunur; yeni düğümün o atanın hangi torununda olduğuna göre 4 durum: **LL, RR, LR, RL**.
- Eklemede tek (LL/RR) ya da çift (LR/RL) döndürme dengeyi yeniden kurar.

```
typedef struct avl *avl_gost;
struct avl {
    int      veri;
    int      yuk;        /* alt ağaç yüksekliği */
    avl_gost sol, sag;
};
int yuk(avl_gost t) { return t ? t->yuk : 0; }
int denge(avl_gost t) {
    return yuk(t->sol) - yuk(t->sag);
}
```

Tek döndürme: LL ve RR



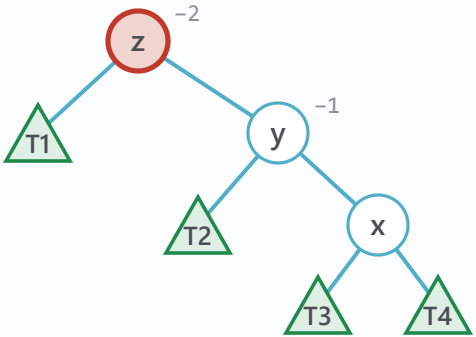
LL: z'nin solunun solu ağır



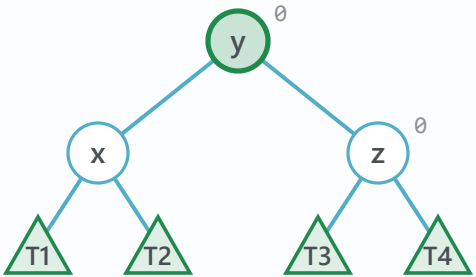
sağa döndür(z): y kök olur

LL: yeni düğüm z'nin sol çocuğunun (y) sol alt ağacına girdi, z'nin faktörü +2. z **sağa** döndürülür: y yukarı çıkar, y'nin sağ alt ağacı T3 z'nin soluna geçer. Sıra korunur: T1 x T2 y T3 z T4.

Tek döndürme: LL ve RR



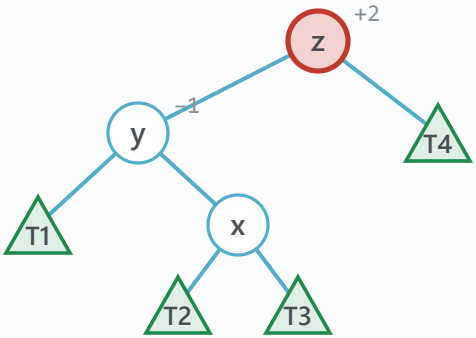
RR: z'nin sağının sağı ağır



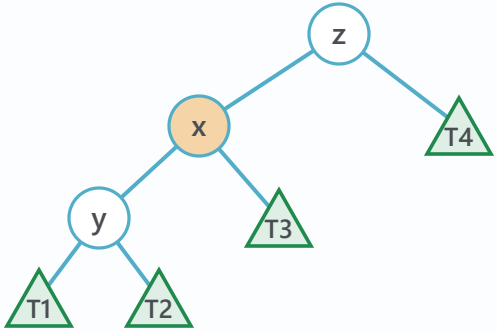
sola döndür(z): y kök olur

RR ayna durumdur: z **sola** döndürülür; y'nin sol alt ağacı (T2) z'nin sağına geçer. Her iki durumda alt ağacın yüksekliği eklemeden önceki değerine döner.

Çift döndürme: LR ve RL



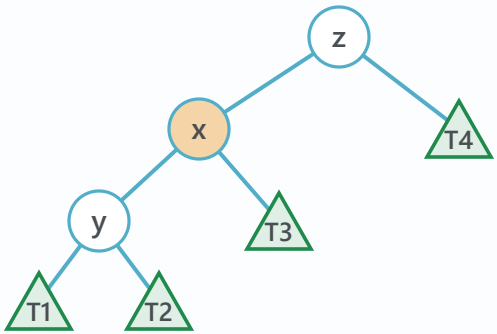
LR: z'nin solunun sağı ağır



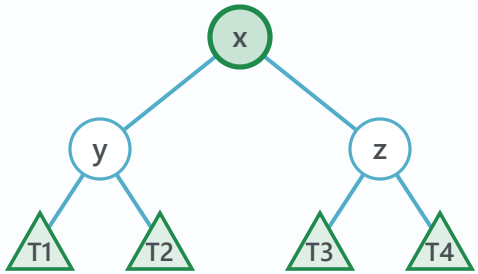
1) sola döndür(y) → LL biçimi

LR: yeni düğüm z'nin sol çocuğunun (y) **sağ** alt ağacında. Tek sağa döndürme işe yaramaz; önce y **sola** döndürülür → LL biçimi.

Çift döndürme: LR ve RL



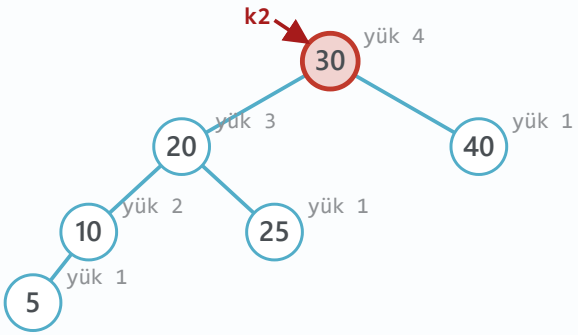
1) sola döndür(y) → LL biçimi



2) sağa döndür(z): x kök

Sonra z **sağa** döndürülür: torun x iki atasının üstüne çıkar. **RL** bunun aynasıdır: önce sağ çocuk sağa, sonra z sola.

sag_dondur(): LL durumunda iki işaretçi



5 eklendi: 30'un faktörü yük(20)=3 − yük(40)=1 = **+2**, sol-sol ağır → LL: `sag_dondur(30)`.

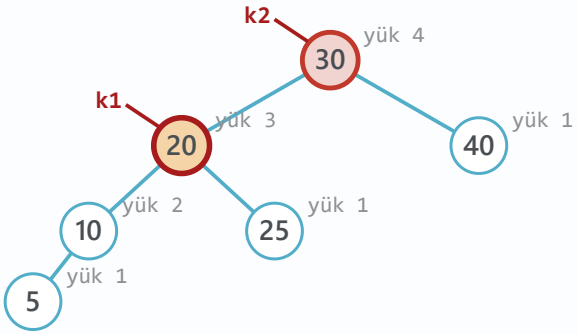
Adım 1 / 7

```
avl_gost sag_dondur(avl_gost k2) {  // LL
  avl_gost k1 = k2->sol;
  k2->sol = k1->sag;
  k1->sag = k2;
  k2->yuk = 1 + MAX(yuk(k2->sol), yuk(k2->sag));
  k1->yuk = 1 + MAX(yuk(k1->sol), k2->yuk);
  return k1;                        // yeni kök
}
```

DEĞİŞKENLER

`k2 = &düğüm(30)` `k1 = ?` `k2→yuk = 4` `k1→yuk = 3`

sag_dondur(): LL durumunda iki işaretçi



k1 = k2->sol = 20: yeni kök olacak düğüm.

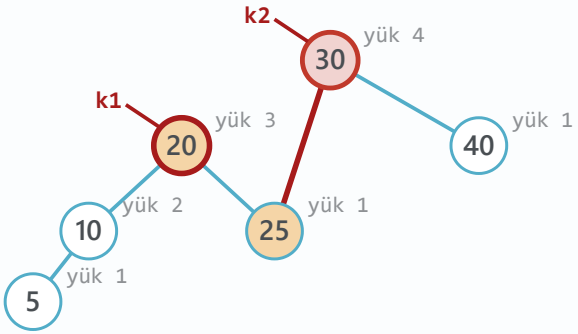
Adım 2 / 7

```
1 avl_gost sag_dondur(avl_gost k2) {  // LL
▶   avl_gost k1 = k2->sol;
3   k2->sol = k1->sag;
4   k1->sag = k2;
5   k2->yuk = 1 + MAX(yuk(k2->sol), yuk(k2->sag));
6   k1->yuk = 1 + MAX(yuk(k1->sol), k2->yuk);
7   return k1;                        // yeni kök
8 }
```

DEĞİŞKENLER

- k2 = &düğüm(30)
- k1 = &düğüm(20)
- k2→yuk = 4
- k1→yuk = 3

sag_dondur(): LL durumunda iki işaretçi



k2->sol = k1->sag: 25, 30'un soluna geçer (20 < 25 < 30, sıra korunur). 25'e şimdilik iki işaretçi var.

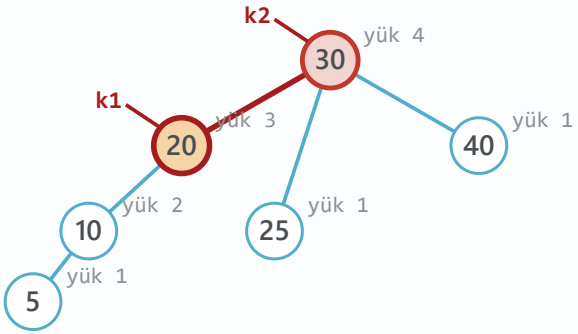
Adım 3 / 7

```
1 avl_gost sag_dondur(avl_gost k2) {  // LL
2   avl_gost k1 = k2->sol;
▶   k2->sol = k1->sag;
4   k1->sag = k2;
5   k2->yuk = 1 + MAX(yuk(k2->sol), yuk(k2->sag));
6   k1->yuk = 1 + MAX(yuk(k1->sol), k2->yuk);
7   return k1;                        // yeni kök
8 }
```

DEĞİŞKENLER

- `k2 = &düğüm(30)`
- `k1 = &düğüm(20)`
- `k2→yuk = 4`
- `k1→yuk = 3`

sag_dondur(): LL durumunda iki işaretçi



k1->sag = k2: 30, 20'nin sağ çocuğu olur. Bağlar bitti; çizimi düzeltelim →

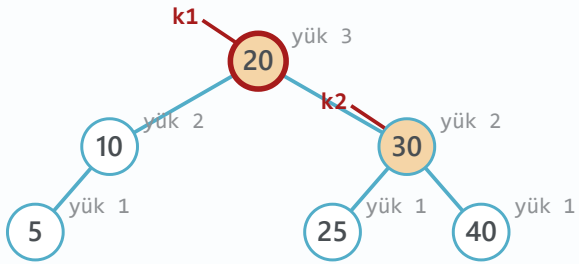
Adım 4 / 7

```
1 avl_gost sag_dondur(avl_gost k2) {  // LL
2   avl_gost k1 = k2->sol;
3   k2->sol = k1->sag;
▶   k1->sag = k2;
5   k2->yuk = 1 + MAX(yuk(k2->sol), yuk(k2->sag));
6   k1->yuk = 1 + MAX(yuk(k1->sol), k2->yuk);
7   return k1;                       // yeni kök
8 }
```

DEĞİŞKENLER

- k2 = **&düğüm(30)**
- k1 = **&düğüm(20)**
- k2→yuk = **4**
- k1→yuk = **3**

sag_dondur(): LL durumunda iki işaretçi



Yükseklikler aşağıdan yukarı: önce k2 (30): 1 + MAX(1, 1) = **2**.

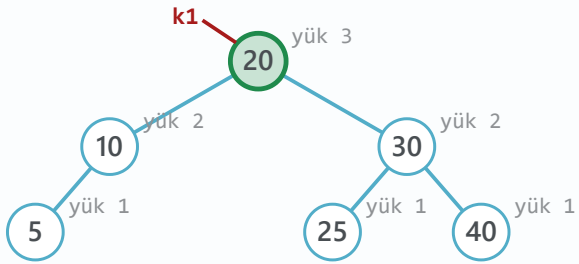
Adım 5 / 7

```
1 avl_gost sag_dondur(avl_gost k2) {  // LL
2   avl_gost k1 = k2->sol;
3   k2->sol = k1->sag;
4   k1->sag = k2;
▶   k2->yuk = 1 + MAX(yuk(k2->sol), yuk(k2->sag));
6   k1->yuk = 1 + MAX(yuk(k1->sol), k2->yuk);
7   return k1;                        // yeni kök
8 }
```

DEĞİŞKENLER

- `k2 = &düğüm(30)`
- `k1 = &düğüm(20)`
- `k2→yuk = 2`
- `k1→yuk = 3`

sag_dondur(): LL durumunda iki işaretçi



Sonra k1 (20): 1 + MAX(yük(10)=2, 2) = **3**. Alt ağacın yüksekliği eklemeden önceki gibi 3.

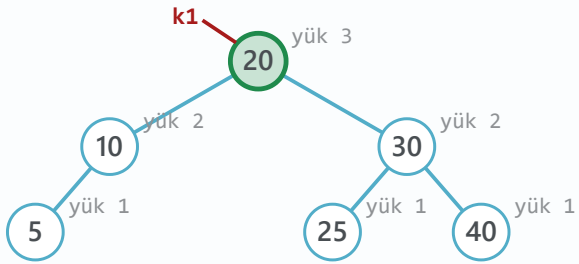
Adım 6 / 7

```
1 avl_gost sag_dondur(avl_gost k2) {  // LL
2   avl_gost k1 = k2->sol;
3   k2->sol = k1->sag;
4   k1->sag = k2;
5   k2->yuk = 1 + MAX(yuk(k2->sol), yuk(k2->sag));
▶   k1->yuk = 1 + MAX(yuk(k1->sol), k2->yuk);
7   return k1;                        // yeni kök
8 }
```

DEĞİŞKENLER

- `k2 = &düğüm(30)`
- `k1 = &düğüm(20)`
- `k2→yuk = 2`
- `k1→yuk = 3`

sag_dondur(): LL durumunda iki işaretçi



Yeni kök k1 döner; çağıran, ebeveynin işaretçisini buna bağlar. Yalnızca 2 işaretçi değişti: **O(1)**.

Adım 7 / 7

```
1 avl_gost sag_dondur(avl_gost k2) {  // LL
2   avl_gost k1 = k2->sol;
3   k2->sol = k1->sag;
4   k1->sag = k2;
5   k2->yuk = 1 + MAX(yuk(k2->sol), yuk(k2->sag));
6   k1->yuk = 1 + MAX(yuk(k1->sol), k2->yuk);
▶   return k1;                        // yeni kök
8 }
```

DEĞİŞKENLER

`k2 = &düğüm(30)` `k1 = &düğüm(20)` `k2→yuk = 2` `k1→yuk = 3`

AVL'ye 3, 2, 1, 4, 5, 6, 7, 16, 15 ekleme



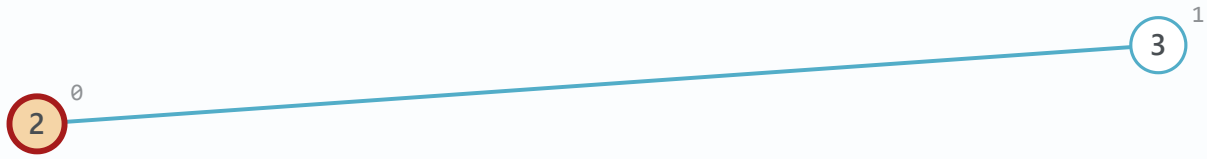
küçük sayı: denge faktörü

3 eklendi. Tüm faktörler −1..+1: döndürme gerekmedi.

Adım 1 / 9

Aynı sıra sıradan BST'ye eklenseydi 3–4–5–6–7–16–15 zinciri yüzünden derinlik 7 olurdu; AVL'de 4. (Örnek: Weiss, *Data Structures and Algorithm Analysis in C*, 4.4)

AVL'ye 3, 2, 1, 4, 5, 6, 7, 16, 15 ekleme

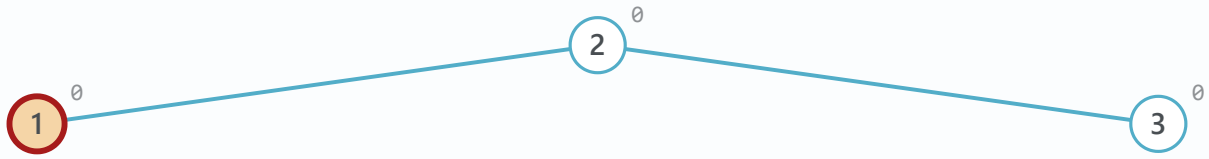


küçük sayı: denge faktörü

2 eklendi. Tüm faktörler −1..+1: döndürme gerekmedi.

Aynı sıra sıradan BST'ye eklenseydi 3–4–5–6–7–16–15 zinciri yüzünden derinlik 7 olurdu; AVL'de 4. (Örnek: Weiss, *Data Structures and Algorithm Analysis in C*, 4.4)

AVL'ye 3, 2, 1, 4, 5, 6, 7, 16, 15 ekleme

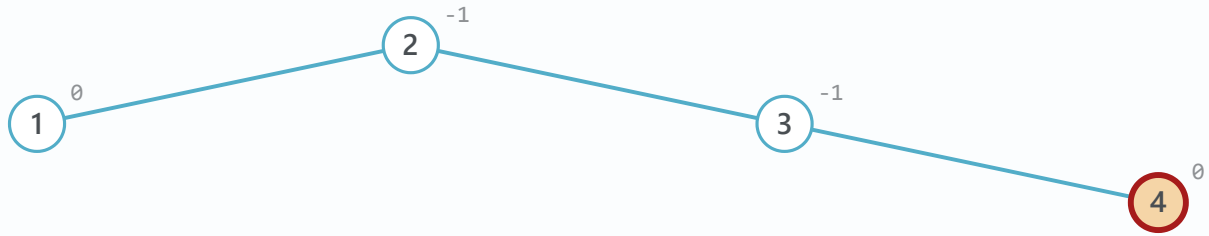


küçük sayı: denge faktörü

1 eklendi. Dengesizlik → LL (3'te sağa döndürme).

Aynı sıra sıradan BST'ye eklenseydi 3–4–5–6–7–16–15 zinciri yüzünden derinlik 7 olurdu; AVL'de 4. (Örnek: Weiss, *Data Structures and Algorithm Analysis in C*, 4.4)

AVL'ye 3, 2, 1, 4, 5, 6, 7, 16, 15 ekleme

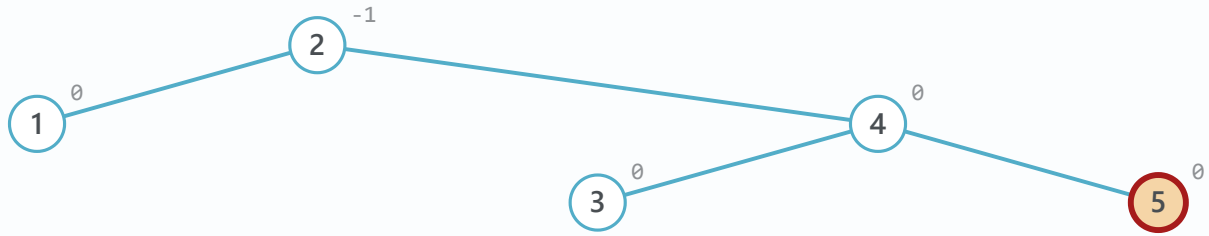


küçük sayı: denge faktörü

4 eklendi. Tüm faktörler −1..+1: döndürme gerekmedi.

Aynı sıra sıradan BST'ye eklenseydi 3–4–5–6–7–16–15 zinciri yüzünden derinlik 7 olurdu; AVL'de 4. (Örnek: Weiss, *Data Structures and Algorithm Analysis in C*, 4.4)

AVL'ye 3, 2, 1, 4, 5, 6, 7, 16, 15 ekleme

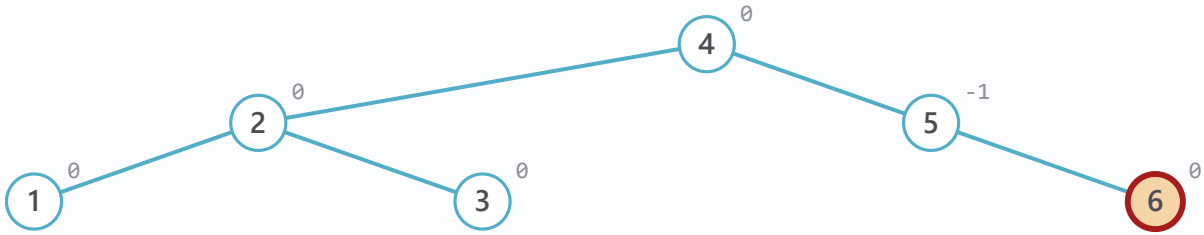


küçük sayı: denge faktörü

5 eklendi. Dengesizlik → RR (3'te sola döndürme).

Aynı sıra sıradan BST'ye eklenseydi 3–4–5–6–7–16–15 zinciri yüzünden derinlik 7 olurdu; AVL'de 4. (Örnek: Weiss, *Data Structures and Algorithm Analysis in C*, 4.4)

AVL'ye 3, 2, 1, 4, 5, 6, 7, 16, 15 ekleme



küçük sayı: denge faktörü

6 eklendi. Dengesizlik → RR (2'de sola döndürme).

Aynı sıra sıradan BST'ye eklenseydi 3–4–5–6–7–16–15 zinciri yüzünden derinlik 7 olurdu; AVL'de 4. (Örnek: Weiss, *Data Structures and Algorithm Analysis in C*, 4.4)

AVL'ye 3, 2, 1, 4, 5, 6, 7, 16, 15 ekleme

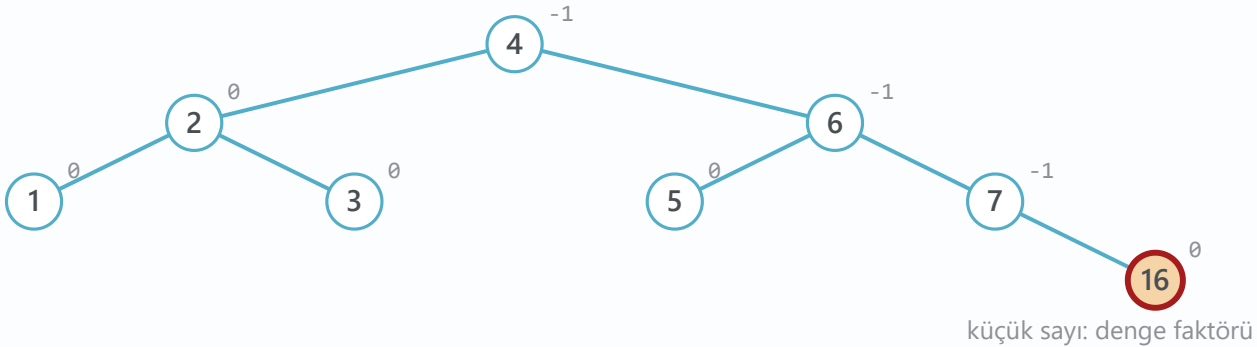


küçük sayı: denge faktörü

7 eklendi. Dengesizlik → RR (5'te sola döndürme).

Aynı sıra sıradan BST'ye eklenseydi 3–4–5–6–7–16–15 zinciri yüzünden derinlik 7 olurdu; AVL'de 4. (Örnek: Weiss, *Data Structures and Algorithm Analysis in C*, 4.4)

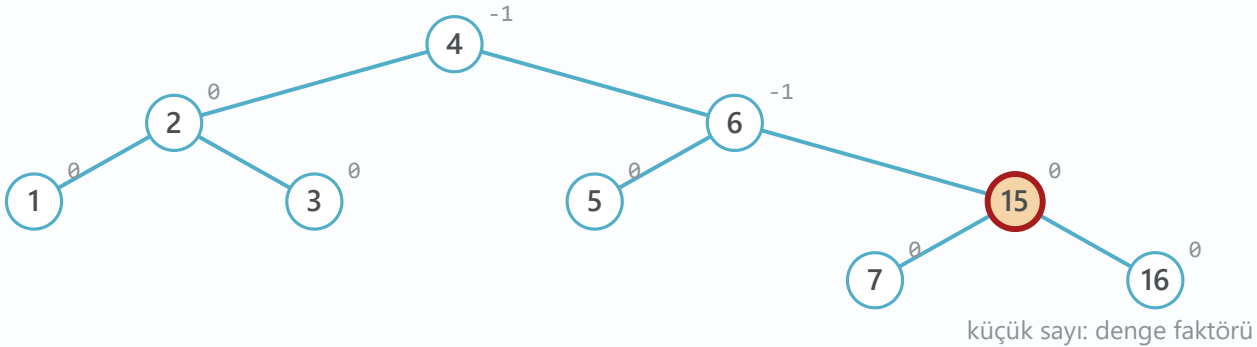
AVL'ye 3, 2, 1, 4, 5, 6, 7, 16, 15 ekleme



16 eklendi. Tüm faktörler −1..+1: döndürme gerekmedi.

Aynı sıra sıradan BST'ye eklenseydi 3–4–5–6–7–16–15 zinciri yüzünden derinlik 7 olurdu; AVL'de 4. (Örnek: Weiss, *Data Structures and Algorithm Analysis in C*, 4.4)

AVL'ye 3, 2, 1, 4, 5, 6, 7, 16, 15 ekleme



15 eklendi. Dengesizlik → RL (16'da sağa, 7'de sola).

Adım 9 / 9

Aynı sıra sıradan BST'ye eklenseydi 3–4–5–6–7–16–15 zinciri yüzünden derinlik 7 olurdu; AVL'de 4. (Örnek: Weiss, *Data Structures and Algorithm Analysis in C*, 4.4)

AVL ekleme: BST eklemesi + dönüşte denge

```
avl_gost avl_ekle(avl_gost t, int x) {
  if (!t) return yeni_avl(x);              /* malloc, yuk = 1        */
  if (x < t->veri)      t->sol = avl_ekle(t->sol, x);
  else if (x > t->veri) t->sag = avl_ekle(t->sag, x);
  else return t;                           /* tekrar eden değer yok  */
  t->yuk = 1 + MAX(yuk(t->sol), yuk(t->sag));
  int d = denge(t);
  if (d >  1 && x < t->sol->veri) return sag_dondur(t);            /* LL */
  if (d < -1 && x > t->sag->veri) return sol_dondur(t);            /* RR */
  if (d >  1) { t->sol = sol_dondur(t->sol); return sag_dondur(t); } /* LR */
  if (d < -1) { t->sag = sag_dondur(t->sag); return sol_dondur(t); } /* RL */
  return t;
}
```

- Özyinelemeden dönerken her ata güncellenir; eklemede en çok **bir** (tek ya da çift) döndürme yeter.
- Kullanım: `kok = avl_ekle(kok, x);` — yeni kök döndüğü için `*L` gerekmez. Silme de benzer ama O(lg n) döndürme gerekebilir.

Kısa bakış: kırmızı-siyah ağaç ve B-ağacı

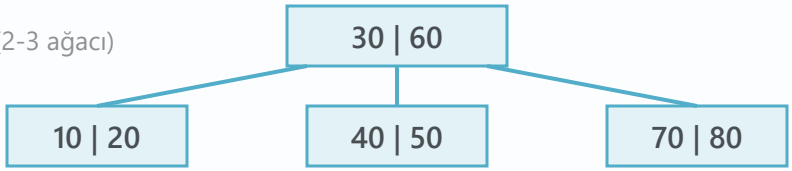
Kırmızı-siyah ağaç (CLRS 13)

- Her düğüm kırmızı ya da siyah; kök ve NULL yapraklar siyah.
- Kırmızı düğümün çocukları siyah.
- Her düğümden NULL'lara giden her yolda **aynı sayıda siyah** düğüm.
- Sonuç: yükseklik ≤ 2 lg(n+1); ekleme/silmede en çok 2–3 döndürme + renk değişimi.
- C++ `std::map`, Java `TreeMap`, Linux çekirdeği.

B-ağacı (Bayer–McCreight 1972)

- m yollu dengeli arama ağacı: düğüm başına $\lceil m/2 \rceil - 1$ … $m-1$ anahtar.
- Tüm yapraklar aynı düzeyde.
- Dolan düğüm ortadaki anahtarı ebeveyne çıkararak **bölünür**; ağaç kökten büyür.
- Bir düğüm = bir disk bloğu → çok az disk erişimi: veritabanı dizinleri, dosya sistemleri (B+ ağacı).

3. dereceden B-ağacı (2-3 ağacı)



Hangi arama ağacı?

yapı	arama / ekleme / silme (en kötü)	denge koşulu	nerede?
ikili arama ağacı	O(n) (ortalama O(lg n))	yok	küçük, rastgele veri; öğretim
AVL	O(lg n)	|yük(sol) − yük(sağ)| ≤ 1	arama ağırlıklı işler
kırmızı-siyah	O(lg n)	siyah yükseklik eşit	kütüphane eşlemeleri (map)
B-ağacı / B+	$O(\log_m n)$ disk erişimi	yapraklar aynı düzeyde	veritabanı, dosya sistemi
splay (“zig-zag”)	O(lg n) amortize	yok; erişilen köke çıkar	sık erişilen öğeler

SINAV SORULARI

Sınav sorusu: bir öğenin ağırlığı

Değerlerin tekrarlı olmadığı bir ikili ağaçta x değerini içeren öğenin **ağırlığı**: "uç düğümlere kadar olan yolların en uzunu". x yoksa ya da L = NULL ise 0. `int AgirlikBul(agac_gost L, int x)` yordamını yazınız. *Global değişken kullanmayın.*

▶ **Örnek çözüm**

Tasarım: iki iyi tanımlı yordamın bileşimi; derinlik(NULL) = 0 sayesinde "bulunamadı" ayrıca ele alınmaz.

Sınav sorusu: AgirlikBul kaç kez işletilir?

```
int AgirlikBul(oge_gost L, int x) {
  if (!L) return 0;
  else if (L->veri == x) return agirlik(L);
  if (AgirlikBul(L->solcocuk, x) >
      AgirlikBul(L->sagcocuk, x))
    AgirlikBul(L->solcocuk, x);
  if (AgirlikBul(L->sagcocuk, x) >
      AgirlikBul(L->solcocuk, x))
    AgirlikBul(L->sagcocuk, x);
  return 0;
}
```

Ağaç: kök 2, sol 10, sağ 20. **i)** x = 20 ve x = 30 için kaç çağrı? **ii)** En kötü durum için özyinelemeli ilişkiyi yazıp çözünüz.

▶ **Çözüm**

SINAV SORULARI

Sınav sorusu: SWAP iki özyinelemeli çağrının arasında

{44, 30, 15, 18, 86, 97, 65, 113, 74, 129} ile kurulan BST için `yordam` 'ı işletiniz; SWAP satırından her geçişte ağaç görünümü değişmişse yeniden çiziniz; son hâli yorumlayınız.

```
void yordam(agac_gost L) {
  agac_gost p;
  if (L) {
    yordam(L->solcocuk);
    SWAP(L->solcocuk, L->sagcocuk, p);
    yordam(L->sagcocuk);
  }
}
```

▶ **Çözüm (adım adım)**

Yorum: SWAP öncesine ya da sonrasına yazılsaydı ağacın tam aynası elde edilirdi; arada olunca sol çocuk SWAP ile sağa geçip aynı düğüm için bir kez daha işlenir.

SINAV SORULARI

Sınav sorusu: SWAP iki özyinelemeli çağrının arasında

{44, 30, 15, 18, 86, 97, 65, 113, 74, 129} ile kurulan BST için `yordam` 'ı işletiniz; SWAP satırından her geçişte ağaç görünümü değişmişse yeniden çiziniz; son hâli yorumlayınız.

```
void yordam(agac_gost L) {
  agac_gost p;
  if (L) {
    yordam(L->solcocuk);
    SWAP(L->solcocuk, L->sagcocuk, p);
    yordam(L->sagcocuk);
  }
}
```

▶ **Çözüm (adım adım)**

Yorum: SWAP öncesine ya da sonrasına yazılsaydı ağacın tam aynası elde edilirdi; arada olunca sol çocuk SWAP ile sağa geçip aynı düğüm için bir kez daha işlenir.

SINAV SORULARI

Sınav sorusu: SWAP iki özyinelemeli çağrının arasında

{44, 30, 15, 18, 86, 97, 65, 113, 74, 129} ile kurulan BST için `yordam` 'ı işletiniz; SWAP satırından her geçişte ağaç görünümü değişmişse yeniden çiziniz; son hâli yorumlayınız.

```
void yordam(agac_gost L) {
  agac_gost p;
  if (L) {
    yordam(L->solcocuk);
    SWAP(L->solcocuk, L->sagcocuk, p);
    yordam(L->sagcocuk);
  }
}
```

▶ **Çözüm (adım adım)**

Yorum: SWAP öncesine ya da sonrasına yazılsaydı ağacın tam aynası elde edilirdi; arada olunca sol çocuk SWAP ile sağa geçip aynı düğüm için bir kez daha işlenir.

SINAV SORULARI

Sınav sorusu: SWAP iki özyinelemeli çağrının arasında

{44, 30, 15, 18, 86, 97, 65, 113, 74, 129} ile kurulan BST için `yordam` 'ı işletiniz; SWAP satırından her geçişte ağaç görünümü değişmişse yeniden çiziniz; son hâli yorumlayınız.

```
void yordam(agac_gost L) {
  agac_gost p;
  if (L) {
    yordam(L->solcocuk);
    SWAP(L->solcocuk, L->sagcocuk, p);
    yordam(L->sagcocuk);
  }
}
```

▶ **Çözüm (adım adım)**

Yorum: SWAP öncesine ya da sonrasına yazılsaydı ağacın tam aynası elde edilirdi; arada olunca sol çocuk SWAP ile sağa geçip aynı düğüm için bir kez daha işlenir.

SINAV SORULARI

Sınav sorusu: SWAP iki özyinelemeli çağrının arasında

{44, 30, 15, 18, 86, 97, 65, 113, 74, 129} ile kurulan BST için `yordam` 'ı işletiniz; SWAP satırından her geçişte ağaç görünümü değişmişse yeniden çiziniz; son hâli yorumlayınız.

```
void yordam(agac_gost L) {
  agac_gost p;
  if (L) {
    yordam(L->solcocuk);
    SWAP(L->solcocuk, L->sagcocuk, p);
    yordam(L->sagcocuk);
  }
}
```

▶ **Çözüm (adım adım)**

Yorum: SWAP öncesine ya da sonrasına yazılsaydı ağacın tam aynası elde edilirdi; arada olunca sol çocuk SWAP ile sağa geçip aynı düğüm için bir kez daha işlenir.

SINAV SORULARI

Sınav sorusu: SWAP iki özyinelemeli çağrının arasında

{44, 30, 15, 18, 86, 97, 65, 113, 74, 129} ile kurulan BST için `yordam` 'ı işletiniz; SWAP satırından her geçişte ağaç görünümü değişmişse yeniden çiziniz; son hâli yorumlayınız.

```
void yordam(agac_gost L) {
  agac_gost p;
  if (L) {
    yordam(L->solcocuk);
    SWAP(L->solcocuk, L->sagcocuk, p);
    yordam(L->sagcocuk);
  }
}
```

▶ **Çözüm (adım adım)**

Yorum: SWAP öncesine ya da sonrasına yazılsaydı ağacın tam aynası elde edilirdi; arada olunca sol çocuk SWAP ile sağa geçip aynı düğüm için bir kez daha işlenir.

SINAV SORULARI

Sınav sorusu: SWAP iki özyinelemeli çağrının arasında

{44, 30, 15, 18, 86, 97, 65, 113, 74, 129} ile kurulan BST için `yordam` 'ı işletiniz; SWAP satırından her geçişte ağaç görünümü değişmişse yeniden çiziniz; son hâli yorumlayınız.

```
void yordam(agac_gost L) {
  agac_gost p;
  if (L) {
    yordam(L->solcocuk);
    SWAP(L->solcocuk, L->sagcocuk, p);
    yordam(L->sagcocuk);
  }
}
```

▶ **Çözüm (adım adım)**

Yorum: SWAP öncesine ya da sonrasına yazılsaydı ağacın tam aynası elde edilirdi; arada olunca sol çocuk SWAP ile sağa geçip aynı düğüm için bir kez daha işlenir.

ÖZET

Karmaşıklık özeti

işlem	en iyi / dengeli	en kötü (çarpık)	ek bellek
arama_agacinda_ara (döngüsel)	O(lg n)	O(n)	O(1)
ara (özyinelemeli)	O(lg n)	O(n)	O(h) yığın
arama_agacina_ekle	Ω(lg n)	O(n)	O(1)
sil	O(lg n)	O(n)	O(1)
derinlik	O(n)	O(n)	O(h)
genel ikili ağaçta ara	O(n)	O(n)	O(h)
AVL arama / ekleme / silme	O(lg n)	O(lg n)	O(lg n) özyineleme
tek döndürme	O(1)	O(1)	—

ÖZET

Sık yapılan hatalar

1. Boş ağaca eklerken kök işaretçisini güncelleyememek (`agac_gost L` yerine `*L` gerekir).
2. Arkadaki işaretçiyi (q) tutmadan NULL'a kadar inip bağlanacak düğümü kaybetmek.
3. Özyinelemeli aramada alt çağrının sonucunu `return` etmemek.
4. İki çocuklu silmede taşınan öğenin sol çocuğunu unutmak (56 kaybolur).
5. Derinlik için global değişken ya da gereksiz ek parametre kullanmak.
6. `MAX` makrosuna özyinelemeli çağrı koyup üstel süreye düşmek.

ÖZET

Alıştırmalar

1. Boş bir BST'ye sırasıyla 50, 30, 70, 20, 40, 60, 80, 35 ekleyin; kök önde sırasını yazın.
2. Aynı ağaçtan 30'u 1. ve 2. yöntemle silin; derinlikleri karşılaştırın.
3. En küçük ve en büyük değeri döndüren döngüsel yordamları yazın.
4. Bir ikili ağacın BST olup olmadığını O(n)'de denetleyen yordamı yazın.
5. Boş AVL'ye 10, 20, 30, 25, 28 ekleyin; hangi döndürmeler yapılır?
6. x'ten küçük anahtar sayısını bulan yordamı yazın (kök ortada gezinmeden yararlanılabilir).

▶ **İpuçları / cevaplar**

ÖZET

Bölüm özeti

Tanım

Sol alt ağaç küçük, sağ büyük, değerler tek; kök ortada = sıralı.

Ekleme

İlk NULL'a eklenir; arka işaretçi q ve `*L` ile boş ağaç.

Arama

Her karşılaştırma bir alt ağacı eler: O(h).

Derinlik

MAX(sol, sağ) + 1; global değişken ve gereksiz parametre yok.

Silme

0/1 çocuk kolay; 2 çocukta öncül/ardıl ile veri değiştir (1 bağ).

Denge

Sıralı veri O(n) yapar; AVL/kırmızı-siyah O(lg n) garanti eder.

ÖZET

Kaynaklar ve devam

- Horowitz–Sahni–Anderson-Freed, *Fundamentals of Data Structures in C*, 5.7 (BST), 10.2 (AVL).
- Cormen vd., *Introduction to Algorithms*, Bölüm 12 (BST), 13 (kırmızı-siyah), 18 (B-ağacı).
- M. A. Weiss, *Data Structures and Algorithm Analysis in C*, 4.4 (AVL); R. Sedgewick, *Algorithms in C*, Bölüm 12–13.

← Bölüm 10: Ağaçlar | Ders sayfası | Bölüm 12: Kılavuzlu İkili Ağaçlar →

Son

Bölüm 11: Arama Ağaçları

Sonraki bölüm: Bölüm 12: Kılavuzlu İkili Ağaçlar · Ders sayfası