VERİ YAPILARI

Veri Yapıları

Ders notları ve alıştırmalar

0 Algoritma analizi

1 Listeler

2 Yığın ve kuyruk

3 Hash tabloları

4 Ağaçlar ve ikili ağaç

5 İkili arama ağacı

6 Dengeli ağaçlar

7 Heap ve öncelikli kuyruk

8 Sıralama algoritmaları

9 Graflar

10 Minimum yayılan ağaç

11 En kısa yol

12 Metin arama

HEKZON

hekzon.com/dersler

13 bölüm · 35 konu

İçindekiler

Bu notlar, YouTube'daki Veri Yapıları dersinin yazılı hâlidir. Her konunun yanındaki zaman, uzun videoda o konunun başladığı yeri gösterir. Her bölümün sonunda sorular var; cevaplarını ve çözümlerini en sonda bulabilirsin. Aynı soruları hekzon.com/dersler adresinde, cevabını hemen göreceğin alıştırmalar olarak da çözebilirsin.

0 **Algoritma analizi** 3
Bir milyon kitap · Bin kat hızlı bilgisayar · Çalıştırmadan hız

1 **Listeler** 13
Dolmayan liste · Adres elimizde · Durmayan döngü

2 **Yığın ve kuyruk** 24
Ctrl+Z · 2 + 3 × 4 · Bip

3 **Hash tabloları** 34
905 nerede? · Kaç kişi? · Kayıp 82

4 **Ağaçlar ve ikili ağaç** 44
Kaç kademe? · Hangi sırayla?

5 **İkili arama ağacı** 52
Aynı yedi sayı · Kökün yerine kim?

6 **Dengeli ağaçlar** 60
Sırayı bozmadan · Neden AVL değil? · Dört okuma

7 **Heap ve öncelikli kuyruk** 70
Sıradaki iş hangisi? · İlk on

8 **Sıralama algoritmaları** 78
Şehre göre sırala · Ek dizi olmadan · Hızın sınırı

9 **Graflar** 89
Kaç bayt? · En kısa çıkış · Hangi sırayla?

10 **Minimum yayılan ağaç** 99
Yirmi köy · Zaten bağlı mı?

11 **En kısa yol** 106
Navigasyon nasıl bulur? · Kafede yirmi dakika · Eksi yol

12 **Metin arama** 117
Bir milyon harfte bir kelime · Geri dönmek şart mı? · Kopya nasıl yakalanır?

Cevaplar ve çözümler 128

VERİ YAPILARI

0 Algoritma analizi

Bu bölümde veri yapısının ne olduğunu ve bir yapının ne yaptığını nasıl yaptığından ayırmayı göreceğiz. Sonra bir algoritmanın hızını girdi büyüdükçe nasıl ölçtüğümüze geçeceğiz: büyük O, büyük Omega ve büyük Teta; en iyi, ortalama ve en kötü durum; bir kodun karmaşıklığını onu çalıştırmadan çıkarmak.

Bir milyon kitap

videoda 0:11

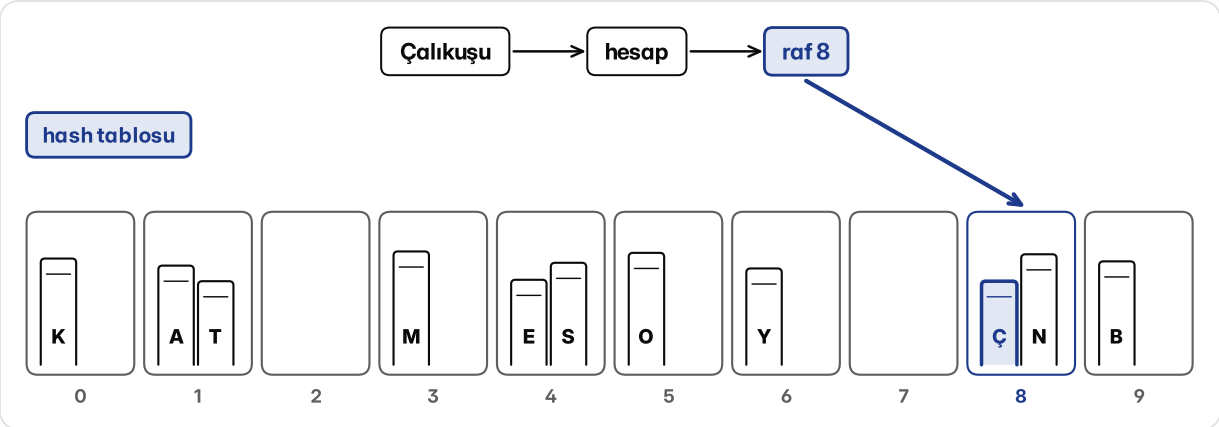
Bir kütüphanedeki 1.000.000 kitap raflara gelişigüzel dizilmiş olsun. Aradığımız kitap ilk rafta da olabilir, en sondakinde de; şansımız yaver gitmezse kitapların hepsine tek tek bakarız. En kötü durumda bu 1.000.000 bakış demektir.

Kitaplar adlarına göre alfabetik dizilseydi önce ortadaki rafa bakardık. Aradığımız ad oradakinden önce geliyorsa arka yarıyı hiç gezmeyiz. Her bakış kalanı yarıya indirir; 1.000.000 kitap 20 yarılamada teke iner, bu yüzden kitap en çok 20 bakışta bulunur. Ama bu düzenin bir bedeli var: araya yeni bir kitap girerse, arkasındaki bütün kitaplar birer yer kaymak zorundadır.



Şekil 0.1 Alfabetik raflarda her bakış kalanı yarıya indirir, 1.000.000 kitapta en çok 20 bakış yeter; araya giren kitap arkadakilerin hepsini kaydırır.

Bir düzen daha düşünelim: kitabın adından basit bir hesapla bir raf numarası çıkıyor ve kitap hep o rafa konuyor. Aradığımız kitabın raf numarasını adından hesaplar, o rafa gider, oradaki bir iki kitaba bakarız. Bilgisayarda bu düzenin adı **hash tablosu**dur.



Şekil 0.2 Çalıkuşu adından yapılan hesapla raf 8'e konur; aranırken de yalnız o raftaki bir iki kitaba bakılır.

Bilgisayar da veriyi bellekte böyle düzenler: geldiği sırayla bir dizide, sıralı bir dizide ya da bir hash tablosunda. Verinin bellekteki bu düzenine **veri yapısı** denir. Kayıtlar aynı, bilgisayar aynı, yalnız düzen farklı; ama 1.000.000 kayıtta 1.000.000 adımla 20 adım arasında 50.000 kat fark vardır.



Şekil 0.3 Bulmak gelişigüzel dizide 1.000.000, sıralı dizide en çok 20, hash tablosunda 1-2 bakış sürer; sıralı diziye eklemek 1.000.000'a kadar kaydırma ister.

Kütüphanenin kataloğunu kullanan bir program, kitapların içeride hangi düzende tutulduğunu bilmek zorunda değildir; ona üç işlem yeter: ekle, sil, bul. Bir yapının ne yaptığını söyleyen bu işlem listesine **soyut veri tipi** denir; işlemlerin içinin nasıl yazıldığına da **gerçekleme**.

C'de bu ayrım iki dosyayla yapılır. Başlık dosyası katalog.h yalnız işlemlerin adlarını verir. İlk satır katalog adında bir tip olduğunu söyler ama içinde ne olduğunu söylemez:

```
typedef struct katalog katalog;

katalog *katalog_yeni(void);
int katalog_ekle(katalog *k, char *ad);
void katalog_sil(katalog *k, char *ad);
int katalog_bul(katalog *k, char *ad);
void katalog_bitir(katalog *k);
```

Kaynak dosyası hem yapının içini hem işlemlerin kodunu tutar. katalog_dizi.c adları geldiği sırayla bir dizide tutar ve ararken baştan sona bakar; strcmp iki metni karşılaştırır, eşitlerse 0 verir:

```
struct katalog {
    char *ad[KAPASITE];
    int adet;
};

int katalog_bul(katalog *k, char *ad) {
    for (int i = 0; i < k->adet; i++)
        if (strcmp(k->ad[i], ad) == 0)
            return 1;
    return 0;
}
```

Kataloğu kullanan main.c yalnız katalog.h'yi görür; yapının içini göremez:

```
int main(void) {
    katalog *k = katalog_yeni();
    if (k == NULL)
        return 1;
    katalog_ekle(k, "Kuyucaklı Yusuf");
    katalog_ekle(k, "Çalıkuşu");
    int var = katalog_bul(k, "Çalıkuşu");
    printf("%d\n", var);
    katalog_sil(k, "Çalıkuşu");
    printf("%d\n", katalog_bul(k, "Çalıkuşu"));
    katalog_bitir(k);
    return 0;
}
```

Çıktı:

```
1
0
```

Aynı main.c, adları sıralı tutan ve ikili aramayla bulan katalog_sirali.c ile derlenince de aynı çıktıyı verir. Gelişigüzel diziyi sıralı diziyle değiştirdik; kullanan programın tek satırı değişmedi, yalnız hızı değişti.

Bir problemi bilgisayarla çözerken izlenen yol da buradan çıkar. Önce problemi modelleriz: hangi veri var, hangi işlem ne sıklıkla istenecek? Diyelim kütüphanede günde binlerce arama yapılıyor, haftada birkaç yeni kitap geliyor. Sonra bu işlemlere uyan yapıyı seçeriz: arama sık, ekleme seyrek; sıralı dizi ya da hash tablosu iş görür. Ardından algoritmayı kurar, en sonda iki şeyi ölçeriz: doğru sonucu veriyor mu, kitap sayısı arttıkça ne kadar uzuyor? Veri yapısı seçmek, aslında hangi işin hızlı olacağına karar vermektir.

Bin kat hızlı bilgisayar

videoda 2:15

Bilgisayar 1.000 kat hızlansa, yavaş bir algoritmayı kurtarabilir mi? 1.000.000 sayıyı sıralayan iki yöntem düşünelim: yavaş olanı n^2 adım atar, yani 1 trilyon; hızlı olanı n log n, yaklaşık 20 milyon. Saniyede 1 milyar adım atan bir bilgisayarda yavaş yöntem yaklaşık 17 dakika, hızlı yöntem saniyenin 50'de biri kadar sürer.

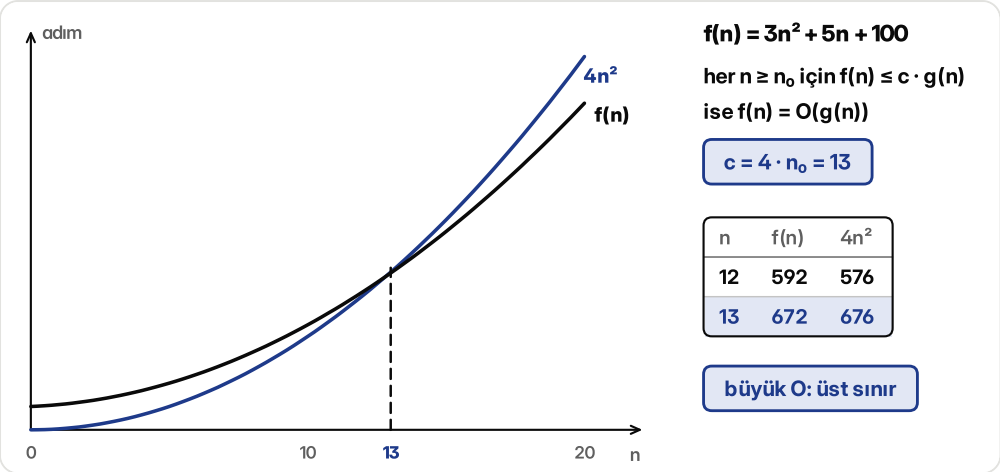
Yavaş yöntemi 1.000 kat hızlı bir bilgisayara verelim: 17 dakika 1 saniyeye iner, ama eski bilgisayardaki hızlı yöntem yine 50 kat öndedir. Veri 10 katına çıkınca yavaş yöntemin işi 100 katına çıkar, hızlı bilgisayarda bile 100 saniye sürer; hızlı yöntem eski bilgisayarda saniyenin dörtte birinde biter. Donanım her adımı aynı oranda kısaltır, yani sabit bir çarpandır; işin büyüme biçimini yalnız algoritma değiştirir.



Şekil 0.4 Hızlı bilgisayarda n^2 1 saniye, eski bilgisayarda n log n 0,02 saniye; veri 10 katına çıkınca n^2'nin işi 100 katına çıkar.

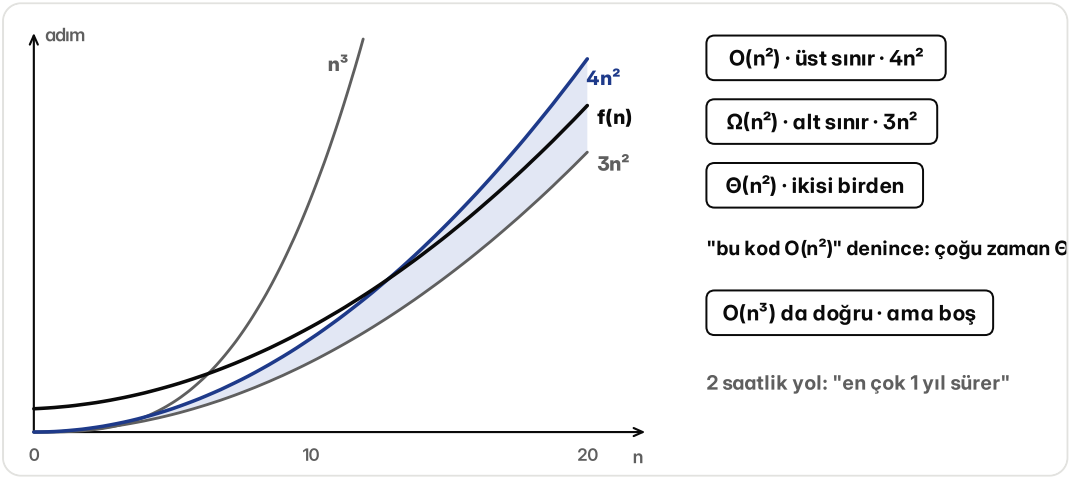
Büyük O bu yüzden sabit çarpanları ve küçük terimleri atar. Bir program n elemanlı girdide $3n^2 + 5n + 100$ adım atsın. Girdi 10 elemanlıyken 450 adım; 1.000 elemanlıyken 3.000.000 + 5.000 + 100 = 3.005.100 adım. Bunun binde 2'si bile 5n ile 100'den gelmez; neredeyse tamamı $3n^2$'den gelir. Baştaki 3 ise donanımın hızı gibi bir çarpandır. Geriye yalnız n^2 kalır: program $O(n^2)$'dir.

Tanımı şöyle: girdi belli bir büyüklüğü (n_0) geçtikten sonra adım sayısı g(n)'nin sabit bir katını hiç aşmıyorsa, yani her $n \geq n_0$ için $f(n) \leq c \cdot g(n)$ ise, $f(n) = O(g(n))$ yazılır. Bizim program için $c = 4$, $n_0 = 13$: $n = 12$'de $f(n) = 592$, $4n^2 = 576$'yı geçer; $n = 13$'te $f(n) = 672 < 676$ olur ve bundan sonra hep altında kalır. Büyük O bir **üst sınır**dır: adım sayısı bundan hızlı büyümez.



Şekil 0.5 f(n) = 3n² + 5n + 100 eğrisi n = 13'ten sonra 4n²'nin hep altında kalır: c = 4, n_0 = 13.

Alt sınırın adı **büyük Omega**: adım sayısı bir yerden sonra g(n)'nin sabit bir katının altına hiç inmiyorsa f(n) = Ω(g(n)). Bizim program 3n²'nin altına hiç düşmez; Ω(n²)'dir. Hem O(n²) hem Ω(n²) olan programa Θ(n²) denir (**büyük Teta**): 3n² ile 4n² arasında sıkışmıştır, tam bu hızda büyür. Günlük konuşmada "bu kod O(n²)" denirken kastedilen çoğu zaman budur. O(n³) demek de yanlış olmaz, çünkü büyük O yalnız üst sınırdır; ama bu, 2 saatlik bir yolculuğa "en çok 1 yıl sürer" demeye benzer: doğrudur, ama işe yaramaz.



Şekil 0.6 f(n), 3n² ile 4n² arasındaki bantta kalır: Θ(n²); O(n³) da doğru ama boştur.

Bu sınırlar en iyi ve en kötü durumla karıştırılmamalı. Karışık bir dizide bir sayıyı aramak, sayı baştaysa 1 adım, sondaysa n adım sürer; sayı her yerde aynı olasılıkla durabiliyorsa ortalaması n/2 civarındadır. Durum, hangi girdiyle çalıştığımızı seçer; sınır, o girdide adım sayısının nasıl büyüdüğünü söyler. Ortalama da en kötü durum da O(n)'dir; 2'ye bölmek de bir sabittir. Çoğu zaman en kötü durumu söyleriz, çünkü programın her girdide tutabileceği söz odur.

Çalıştırmadan hız

videoda 4:39

Bir kodun ne kadar hızlı çalışacağını, onu hiç çalıştırmadan nasıl anlarız? Her satırın kaç kez çalıştığını sayarak. Çoğu kodu üç kural çözer; kendini çağıran kod için de bir ağaç çizeriz.

Birinci kural: art arda gelen bloklar toplanır. 1.000 sayıyı toplayan bir döngünün ardından aynı sayıları 2'yle çarpan ikinci bir döngü geliyor; adim sayacı her turda bir artıyor:

```
for (int i = 0; i < n; i++) {
    toplam += a[i];
    adim++;
}
for (int i = 0; i < n; i++) {
    a[i] = a[i] * 2;
    adim++;
}
```

Dizi 1'den 1.000'e kadar sayılarla doluyken program şunu yazar:

```
toplam 500500, 2000 adım
```

1.000 + 1.000 = 2.000 adım. n elemanlı dizide bu 2n eder; sabit atılır, O(n) kalır. Bloklardan biri $O(n^2)$ olsaydı toplamda büyük olan kalırdı: $O(n^2) + O(n) = O(n^2)$.

İkinci kural: iç içe döngüler çarpılır. 1'den 1.000'e kadar sayıların içinde toplamı 1.001 eden çiftleri arayalım:

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++) {
        if (a[i] + a[j] == hedef)
            cift++;
        adim++;
    }
```

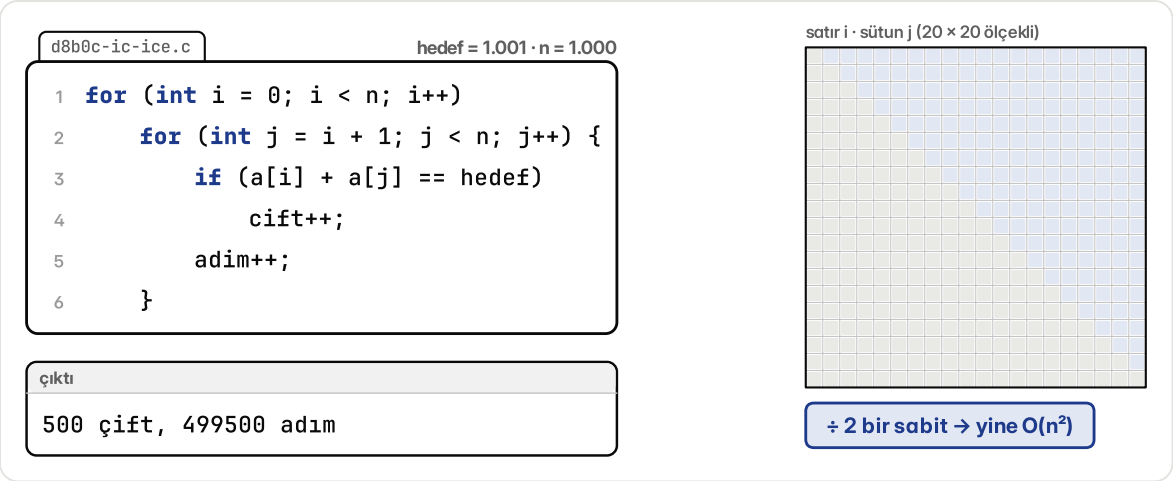
Çıktı:

```
1000 çift, 1000000 adım
```

Dıştaki döngünün her turunda içteki döngü baştan sona döner; içteki satır 1.000 × 1.000 = 1.000.000 kez çalışır: $O(n^2)$. Bu sürüm her çifti iki kez sayar, (1, 1000) ile (1000, 1) gibi. Her çifte bir kez bakmak için içteki döngüyü int j = i + 1'den başlatırsak iş yarıya iner:

```
500 çift, 499500 adım
```

Ama yarı da bir sabittir; yine $O(n^2)$.



Şekil 0.7 1.000 × 1.000 adımı temsil eden 20 × 20'lik ızgara; içteki döngü j = i + 1'den başlayınca yalnız köşegenin üstündeki üçgen kalır, iş yarıya iner ama yine O(n²).

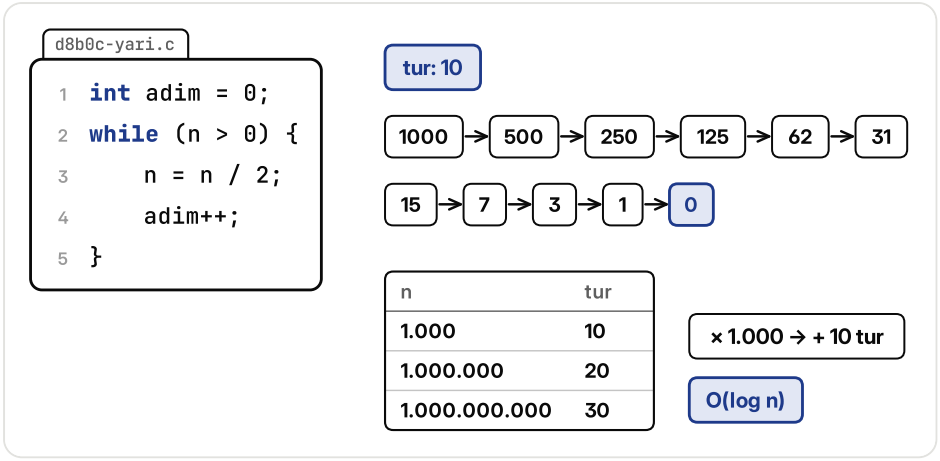
Üçüncü kural: her turda sayıyı yarıya indiren döngü log n kez döner.

```
int tur_say(int n) {
    int adim = 0;
    while (n > 0) {
        n = n / 2;
        adim++;
    }
    return adim;
}

int main(void) {
    printf("%d tur\n", tur_say(1000));
    printf("%d tur\n", tur_say(1000000));
    printf("%d tur\n", tur_say(1000000000));
    return 0;
}
```

Çıktı:

```
10 tur
20 tur
30 tur
```

1.000 sayısı 2'ye bölüne bölüne, küsurat atılarak 10 turda 0'a iner; 1 milyon 20, 1 milyar 30 turda. Veri 1.000 kat büyüyünce döngü yalnız 10 tur uzar: O(log n).



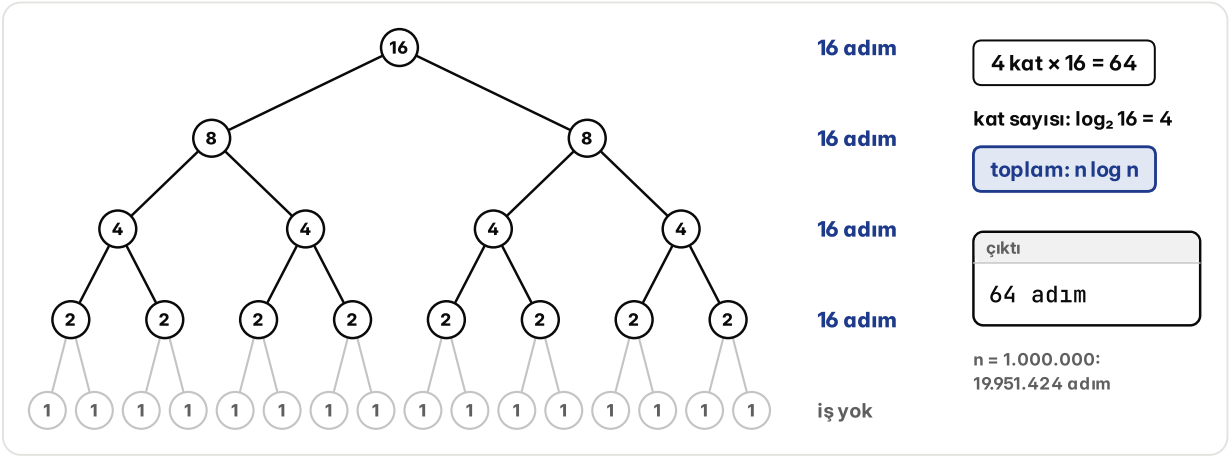
Şekil 0.8 1000 → 500 → … → 1 → 0 zinciri 10 turda biter; 1 milyon 20, 1 milyar 30 tur: veri 1.000 kat büyüyünce 10 tur eklenir.

Kendini çağıran kodda iş çağrıların içine dağılmıştır; döngüleri saymak yetmez. Aşağıdaki fonksiyon diziyi ikiye bölüyor, her yarı için kendini çağırıyor, sonra iki yarıyı n adımda birleştiriyor (birleştirme işini döngüdeki sayaç temsil ediyor). Birleştirmeli sıralamanın iskeleti budur:

```
void bol(int n) {
    if (n <= 1)
        return;
    bol(n / 2);
    bol(n - n / 2);
    for (int i = 0; i < n; i++)
        adim++;   /* birleştirme işi */
}
```

Maliyetini bir denklemle yazarız: n elemanın maliyeti, yarım dizinin maliyetinin 2 katı artı n, yani T(n) = 2T(n/2) + n. Maliyeti kendi küçük hâliyle veren böyle bir denkleme **yineleme bağıntısı** denir.

Çözmek için çağrıları bir ağaç gibi çizeriz; 16 elemanla başlayalım. Tepedeki çağrı 16 adım birleştirir. Altındaki 2 çağrı 8'er adım atar: toplam yine 16. Sonra 4 çağrı 4'er, 8 çağrı 2'şer adım; her kat 16 tutar. Tek elemanlık parçada iş yoktur. 4 kat, her kat 16: 64 adım. Genel hâli: her kat n tutar, kat sayısı log n; toplam n log n.



Şekil 0.9 Çağrı ağacında her kat 16 adım tutar ve $\log_2 16 = 4$ kat vardır: 4 × 16 = 64 adım.

main, bol(16)'yı ve ardından bol(1000000)'u çağırıp sayacı yazdırınca şunu görürüz:

```
64 adım
19951424 adım
```

64'ü kodu hiç çalıştırmadan, kâğıt üstünde bulmuştuk. 1.000.000 elemanda sayaç 19.951.424'tür; $n \log_2 n$ de yaklaşık 19.931.569 eder.

Özet

- Veri yapısı verinin bellekteki düzenidir; aynı veride düzen değişince bir işin süresi 1.000.000 adımdan 20 adıma inebilir.
- Soyut veri tipi bir yapının ne yaptığını, gerçekleme nasıl yaptığını söyler; C'de başlık dosyası ile kaynak dosyası bu ikisini ayırır.
- Donanım sabit bir çarpandır; büyük O sabitleri ve küçük terimleri atar: $3n^2 + 5n + 100$ adım $O(n^2)$'dir.
- Büyük O üst sınır, büyük Omega alt sınır, büyük Teta ikisi birden; en iyi, ortalama ve en kötü durum ise hangi girdiyle çalışıldığını seçer.
- Art arda bloklar toplanır, iç içe döngüler çarpılır, her turda yarıya inen döngü log n kez döner.
- Kendini çağıran kodun maliyeti yineleme bağıntısıyla yazılır; $T(n) = 2T(n/2) + n$ ağaçla çözülünce n log n çıkar.

Sorular

1 **Aynı kitap kayıtları bir kez geldiği sırayla bir dizide, bir kez de alfabetik sıralı bir dizide tutuluyor. İki durum arasındaki fark neye örnektir?**

A Farklı algoritma dillerine
B Farklı veri yapılarına
C Farklı soyut veri tiplerine
D Farklı donanımlara

2 **main.c yalnız katalog.h'yi görüyor ve katalog_ekle ile katalog_bul'u çağırıyor. Program katalog_dizi.c yerine katalog_sirali.c ile derlenirse main.c'de ne değişmelidir?**

A struct katalog tanımı main.c'ye kopyalanmalıdır.
B katalog_bul çağrıları ikili arama yapan bir çağrıyla değiştirilmelidir.
C Hiçbir satır değişmez; yalnız programın hızı değişir.
D katalog.h'deki prototipler main.c'ye taşınmalıdır.

3 **Bir program n elemanlı girdide $3n^2 + 5n + 100$ adım atıyor. Girdi 20 elemanlıyken kaç adım atar?**

Cevap: _____

4 **$f(n) = 3n^2 + 5n + 100$ için aşağıdakilerden hangisi yanlıştır?**

A $f(n) = O(n^2)$

B $f(n) = O(n^3)$

C $f(n) = \Omega(n^2)$

D $f(n) = \Omega(n^3)$

5 **Karışık bir dizide bir sayı aranıyor; sayı her yerde aynı olasılıkla durabiliyor. Ortalama durum için hangisi doğrudur?**

A Ortalama yaklaşık n/2 adımdır ve bu O(n)'dir.

B Ortalama O(1)'dir, çünkü sayı bazen ilk gözdedir.

C Ortalama O(log n)'dir, çünkü her bakışta aranacak yer yarıya iner.

D O(n/2), O(n)'den ayrı ve daha hızlı bir sınıftır.

6 **Aşağıdaki kod ekrana ne yazar?**

```
int n = 64, tur = 0;
while (n > 0) {
    n = n / 2;
    tur++;
}
printf("%d\n", tur);
```

Cevap: _____

7 **Aşağıdaki kodda adim++ satırı kaç kez çalışır, büyük O ile?**

```
for (int i = 0; i < n; i++)
    for (int j = n; j > 1; j = j / 2)
        adim++;
```

A $O(n^2)$

B $O(n \log n)$

C $O(\log n)$

D $O(n)$

8 **Bölümdeki bol fonksiyonu adim = 0 iken bol(8) ile çağrılıyor. Çağrı bitince adim kaç olur?**

Cevap: _____

VERİ YAPILARI

1 Listeler

Bu bölümde listenin iki gerçeklemesini karşılaştıracağız. Dinamik dizinin dolunca nasıl büyüdüğünü ve bu büyümenin ortalamada neden ucuz kaldığını, çift yönlü bağlı listede bir düğümün yürümeden nasıl silindiğini, gözcü düğümü ve dairesel listeyi göreceğiz. Son olarak üç klasik liste problemini tek yürüyüşle çözeceğiz: ters çevirme, ortadaki düğüm ve döngü bulma.

Dolmayan liste

videoda 6:40

Boş bir Python listesine arka arkaya 1.000.000 sayı ekleyelim:

```
liste = []
for i in range(1000000):
    liste.append(i)
```

Liste bir kez bile dolmaz. Oysa bellekten alınan her blok belli bir boydadır, kendiliğinden büyümez. Peki Python'ın listesi dolunca ne yapar?

Liste dört iş vaat eder: eleman eklemek, silmek, bir değeri bulmak ve sıradaki herhangi bir elemanı, örneğin 100.'yü getirmek. Bu vaadi iki yapı tutar: **dinamik dizi** ve **bağlı liste** (linked list). Adına rağmen Python'ın listesi bağlı liste değil, dinamik dizidir: gözleri bellekte yan yana durur, 100. gözün adresi tek bir hesapla bulunur: başlangıç adresi + 100 × 8.

Dinamik dizi, elemanların durduğu bloğun adresinin yanında iki sayı tutar: kaç gözü olduğu, yani **kapasitesi**, ve kaçının dolu olduğu (boy):

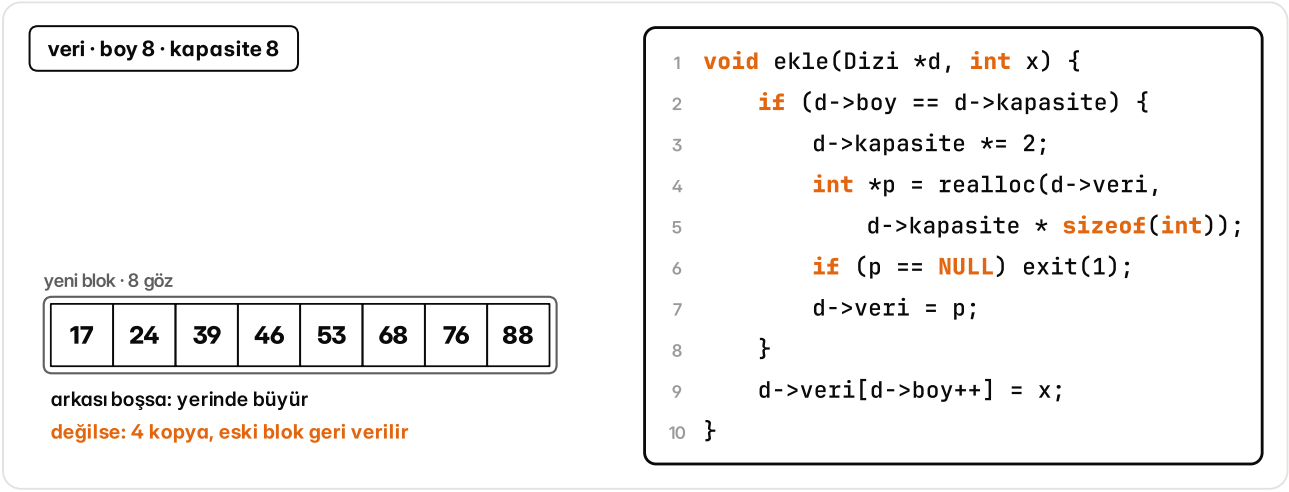
```
typedef struct {
    int *veri;
    int boy, kapasite;
} Dizi;
```

Kapasite 4; 17, 24, 39 ve 46 yerleşti. 53 gelince yer yok. Ekleme fonksiyonu önce buna bakar: boy kapasiteye eşitse kapasiteyi ikiye katlar ve realloc ile 8 gözlük yer ister:

```
void ekle(Dizi *d, int x) {
    if (d->boy == d->kapasite) {
        d->kapasite *= 2;
        int *p = realloc(d->veri,
            d->kapasite * sizeof(int));
        if (p == NULL) exit(1);
        d->veri = p;
    }
    d->veri[d->boy++] = x;
}
```

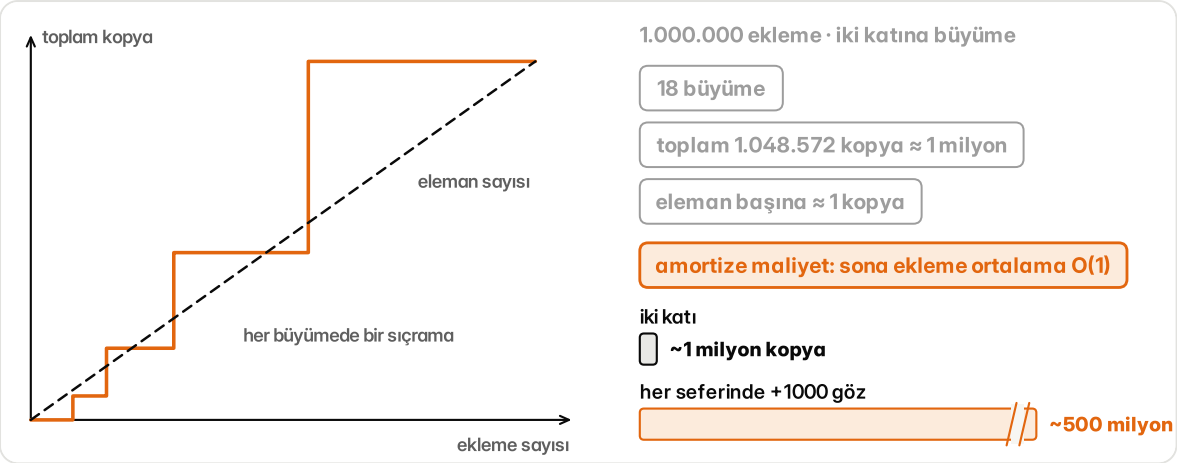
realloc, malloc ile alınmış bir bloğu yeni boya getirir. Bloğun arkası boşsa blok yerinde büyür; değilse yeni bir yer alınır, 4 eleman oraya kopyalanır, eski blok geri verilir. Yer bulunamazsa realloc NULL döndürür ve eski blok yerinde kalır; bu örnek o durumda programı exit(1) ile durdurur. 53 ilk boş göze yazılır; sonraki 3 ekleme hiç kopya istemez. 8 sayıyı ekleyen program her adımda boyu ve kapasiteyi yazar:

```
17 eklendi: boy 1, kapasite 4
24 eklendi: boy 2, kapasite 4
39 eklendi: boy 3, kapasite 4
46 eklendi: boy 4, kapasite 4
53 eklendi: boy 5, kapasite 8  (buyudu, 4 eleman kopyalanabilir)
68 eklendi: boy 6, kapasite 8
76 eklendi: boy 7, kapasite 8
88 eklendi: boy 8, kapasite 8
17 24 39 46 53 68 76 88
```



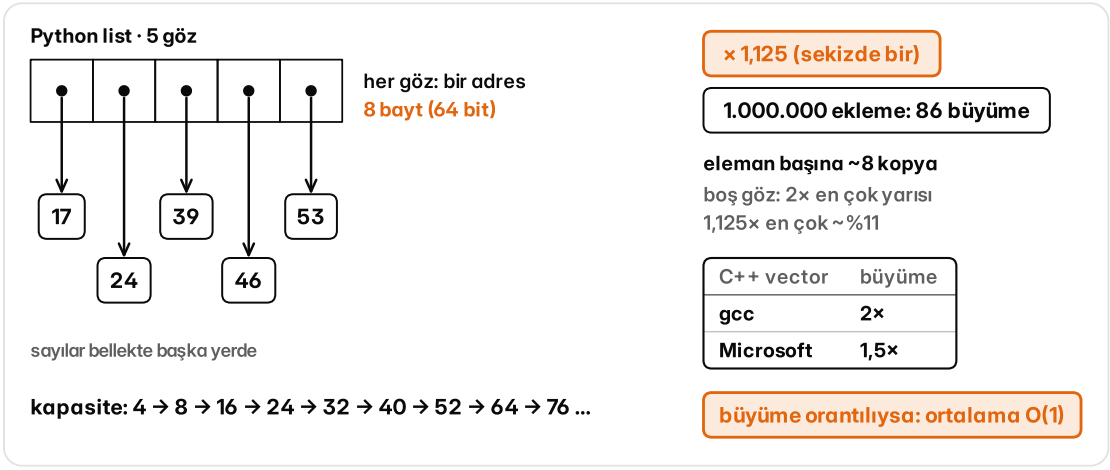
Şekil 1.1 Kapasite 4 dolunca 8 gözlük yeni blok alınır, 4 eleman kopyalanır; 53, 68, 76 ve 88 kopyasız yerleşir: boy 8, kapasite 8.

Kopya pahalı görünür ama seyrek gelir. Kapasite her dolduğunda iki katına çıkarsa, 1.000.000 eklemede dizi yalnız 18 kez büyür. Kopyalar 4, 8, 16, ... 524.288 diye ikiye katlanır; her büyümede hepsi kopyalansa bile toplam 1.048.572, yaklaşık 1 milyon kopya eder. Yani eleman başına ortalama 1 kopya. Tek tek bakınca bazı eklemeler pahalıdır ama ortalaması sabittir. Buna **amortize maliyet** denir: dizinin sonuna ekleme ortalamada O(1)'dir. Kapasiteyi her seferinde 1000 göz artırsaydık, aynı 1.000.000 ekleme yaklaşık 500 milyon kopya isterdi.



Şekil 1.2 İki katına büyüyen dizide 1.000.000 eklemede 18 büyüme ve yaklaşık 1 milyon kopya; her seferinde 1000 göz eklenseydi yaklaşık 500 milyon.

Bir Python listesi de böyle büyür, iki farkla. Gözlerde sayıların kendisi değil adresleri durur; her göz 8 bayt. Dolunca da iki katına değil, sekizde bir oranında büyür: 1.000.000 eklemede 86 kez. Kopya biraz artar, eleman başına 8 kadar olur; karşılığında boş bekleyen göz azalır. Benzer biçimde C++'ın vector'ü de bir dinamik dizidir; kütüphaneye göre iki ya da bir buçuk katına büyür. Oran ne olursa olsun, yeni blok eskisiyle orantılı büyüdükçe ortalama maliyet sabit kalır.



Şekil 1.3 Python listesinin gözlerinde 8 baytlık adresler durur; kapasite 4, 8, 16, 24, 32 … diye sekizde bir oranında büyür.

Dinamik dizinin maliyeti:

İşlem	Maliyet
i'nci elemana gitmek	O(1)
Sona eklemek	ortalama O(1)
Araya eklemek, silmek	O(n)
Aramak	O(n)

Adres elimizde

videoda 8:50

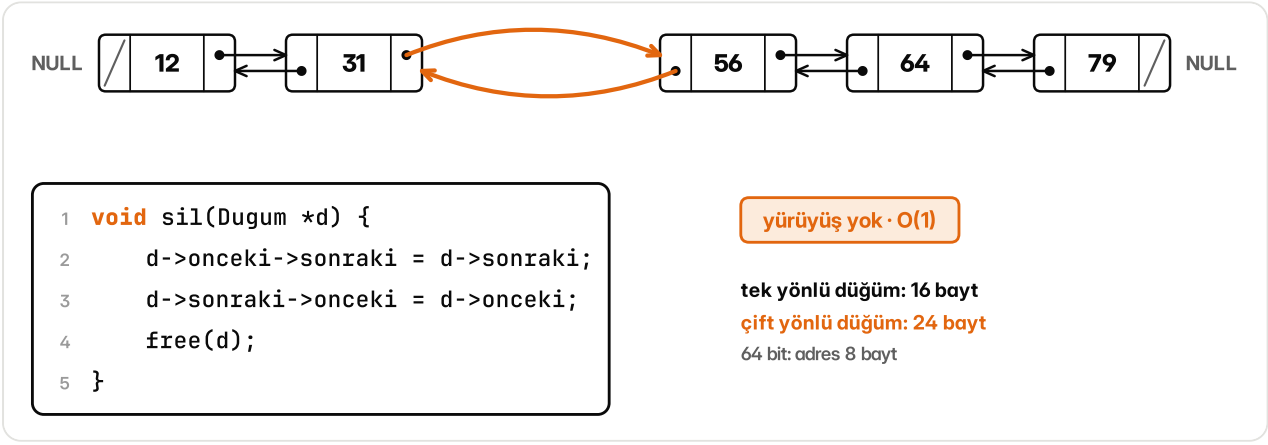
Bir bağlı listede silmek istediğimiz düğümün adresi elimizde: 12, 31, 47, 56, 64, 79 listesinde 47'nin düğümü. Yine de tek yönlü listede onu silmek için baştan yürümek gerekiyor. Adres elimizdeyken neden?

Bağlı listede 100. elemana gitmenin tek yolu baştan yürümektir: 100 adım; yarım milyonuncuya yarım milyon adım, yani O(n). Ama elimizdeki bir düğümün arkasına eklemek, liste ne kadar uzun olursa olsun 2 atamadır: O(1). Silmek o kadar kolay değil. 47'yi çıkarmak için ondan önceki düğümün, 31'in sonraki adresi değişmeli. 47 kendinden öncekini bilmez; 31'i bulmak için baştan yürümek gerekir.

Çift yönlü listede her düğüm, sonrakinin yanında öncekinin adresini de taşır. 47'yi silmek 2 atamadır: 31'in sonraki adresi 56'yı, 56'nın önceki adresi 31'i gösterir. Sonra free ile düğüm belleğe geri verilir:

```
typedef struct dugum {
    int veri;
    struct dugum *onceki, *sonraki;
} Dugum;

void sil(Dugum *d) {
    d->onceki->sonraki = d->sonraki;
    d->sonraki->onceki = d->onceki;
    free(d);
}
```

Liste 1 milyon düğümlü de olsa yürüyüş yok: silme O(1). Bedeli düğüm başına bir adres daha: 64 bitlik bilgisayarda düğüm 16 bayttan 24 bayta çıkar.



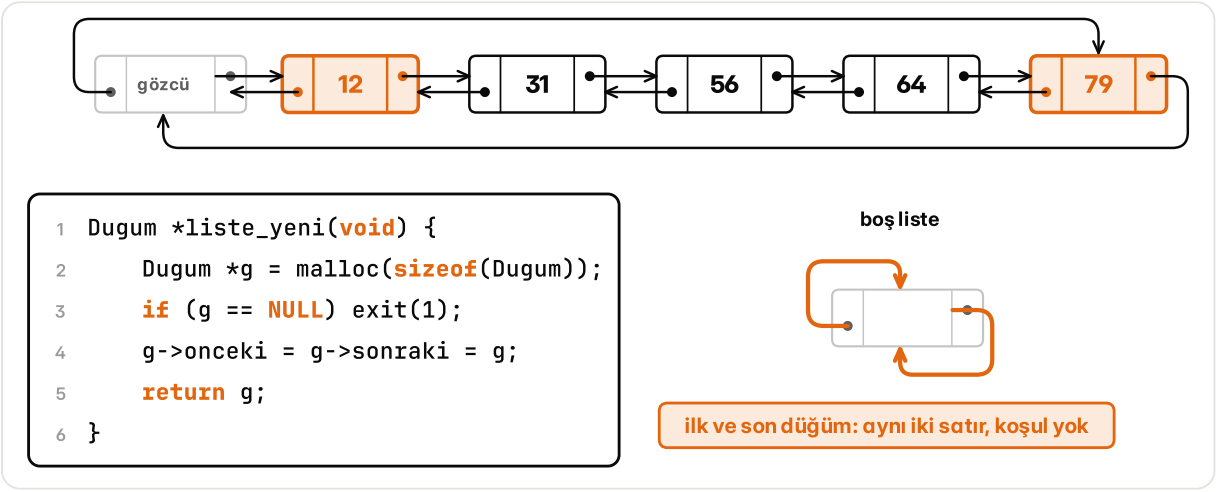
Şekil 1.4 47 iki atamayla çıkar: 31'in sonraki adresi 56'yı, 56'nın önceki adresi 31'i gösterir; yürüyüş yok, O(1).

Bu iki satır uçlarda sorun çıkarır. 12'nin öncesi yok, 79'un sonrası yok; ikisinde de adres yerine NULL durur. NULL bir düğüm değildir, onun alanına yazılamaz. Her uç için ayrı bir koşul yazmak yerine listeye veri taşımayan bir düğüm eklenir: **gözcü düğüm** (sentinel). İlk düğümün öncesi de son düğümün sonrası da gözcüdür. Boş liste, kendini gösteren tek bir gözcüden ibarettir:

```
Dugum *liste_yeni(void) {
    Dugum *g = malloc(sizeof(Dugum));
    if (g == NULL) exit(1);
    g->onceki = g->sonraki = g;
    return g;
}
```

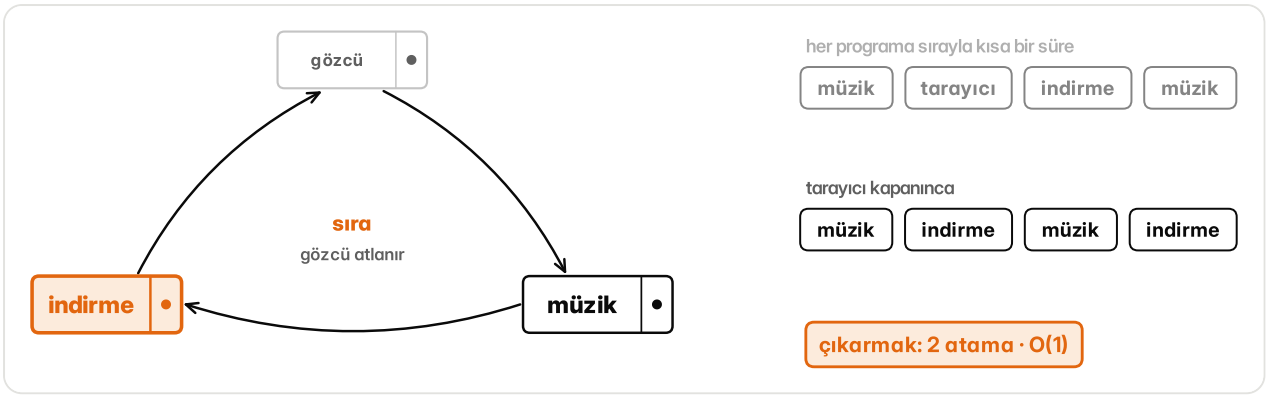
Artık her düğümün iki komşusu var; ilk düğümü de son düğümü de aynı iki satır siler. Program 47'yi, sonra 12'yi ve 79'u aynı sil ile, koşulsuz siliyor ve listeyi iki yönde yazıyor:

```
bos liste: gozcu kendini gosteriyor mu? evet
liste:        12 31 47 56 64 79 | geri: 79 64 56 47 31 12
47 silindi:   12 31 56 64 79 | geri: 79 64 56 31 12
12 silindi:   31 56 64 79 | geri: 79 64 56 31
79 silindi:   31 56 64 | geri: 64 56 31
hepsi silindi: gozcu yine kendini gosteriyor mu? evet
sizeof(Dugum) = 24 bayt
```



Şekil 1.5 Gözcü, ilk düğümün öncesi ve son düğümün sonrasıdır; boş listede kendini gösterir, uçlardaki düğümler koşulsuz silinir.

Gözcüyle kurulan liste bir dairedir: son düğümden sonra gözcü, ondan sonra yine ilk düğüm gelir. **Dairesel liste**, sırayla dönen işler için biçilmiş kaftandır. Zaman paylaşımında işlemci her programa sırayla kısa bir süre verir: müzik, tarayıcı, indirme, sonra yine müzik. Tarayıcı kapanınca düğümü iki atamayla çıkar; sıra müzik, indirme diye dönmeye devam eder.



Şekil 1.6 Sıra daire etrafında döner, gözcü atlanır; tarayıcı 2 atamayla çıkınca sıra müzik, indirme diye sürer.

Ama düğüm çoğu zaman elde değildir; önce aranır, aramak da yürümektir. Bu bilgisayarda 20.000 rastgele sayıyı, sıralı kalacak biçimde tek tek ekledik. Dizi her seferinde yeri bulup sağını kaydırdı; liste yeri bulup düğümü bağladı. Dizi saniyenin onda birinden kısa sürede (yaklaşık 0,08 sn) bitirdi; liste 1 saniyeyi geçti (yaklaşık 1,3 sn), 10 kattan fazla yavaş. Dizinin elemanları yan yana durur, işlemci onları bloklar hâlinde okur; dağınık düğümlerin her biri belleğe ayrı bir gidiştir. Bu yüzden bağlı liste, düğümün adresi zaten elimizdeyken kazanır.

Çift yönlü listenin maliyeti:

İşlem	Maliyet
i'nci elemana gitmek	O(n)
Aramak	O(n)
Eldeki düğümün yanına eklemek	O(1)
Eldeki düğümü silmek	O(1)

Durmayan döngü

videoda 11:06

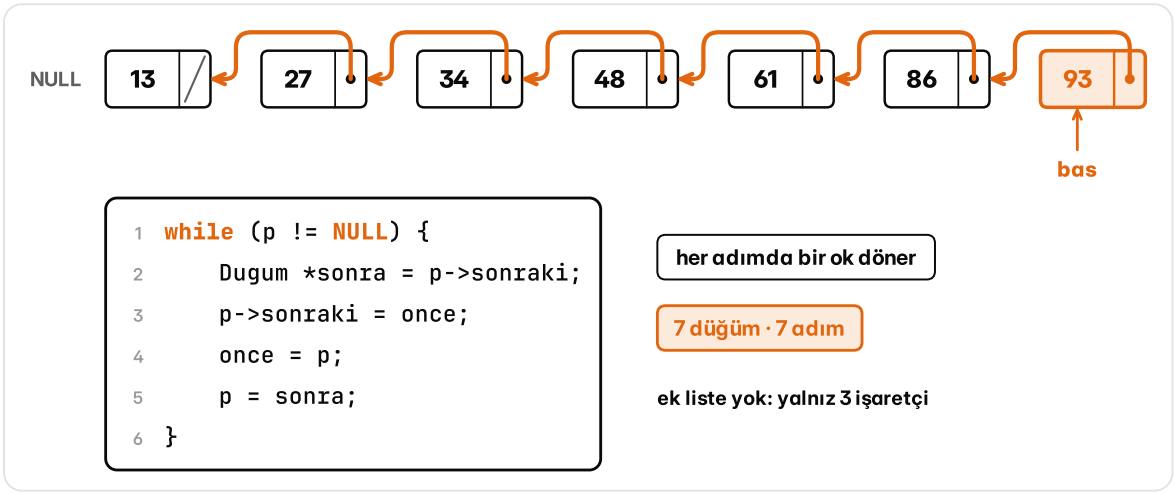
Bir bağlı listeyi dolaşan program durmuyor. Liste çok mu uzun, yoksa son düğüm NULL yerine geride bir düğümü mü gösteriyor? Gezdiğimiz düğümleri bir yere yazmadan, yalnız iki işaretçiyle nasıl anlarız? Önce aynı fikirle çözülen iki probleme bakalım.

Bağlı listenin kuralı: her düğüm yalnız sonrakini bilir, geri dönülmez. Listeyi ters çevirmek için yeni liste kurmayız, okları çevririz. Üç işaretçi birlikte yürür: önceki (once), şimdiki (p) ve sonraki (sonra) düğüm. Her adımda sonraki düğümün adresi önce saklanır, yoksa ok dönünce kaybolur. Sonra şimdiki düğümün oku geriyi gösterir ve hepsi bir düğüm ilerler:

```
Dugum *ters_cevir(Dugum *bas) {
    Dugum *once = NULL, *p = bas;
    while (p != NULL) {
        Dugum *sonra = p->sonraki;
        p->sonraki = once;
        once = p;
        p = sonra;
    }
    return once;
}
```

13, 27, 34, 48, 61, 86, 93 listesinde 7 düğüm, 7 adım; liste artık 93'le başlar. Program önce listeyi, sonra adım adım çevrilmiş hâlini ve adım sayısını, en sonda ters_cevir ile geri çevrilmiş hâlini yazar:

```
13 27 34 48 61 86 93
93 86 61 48 34 27 13
7 adim
13 27 34 48 61 86 93
```

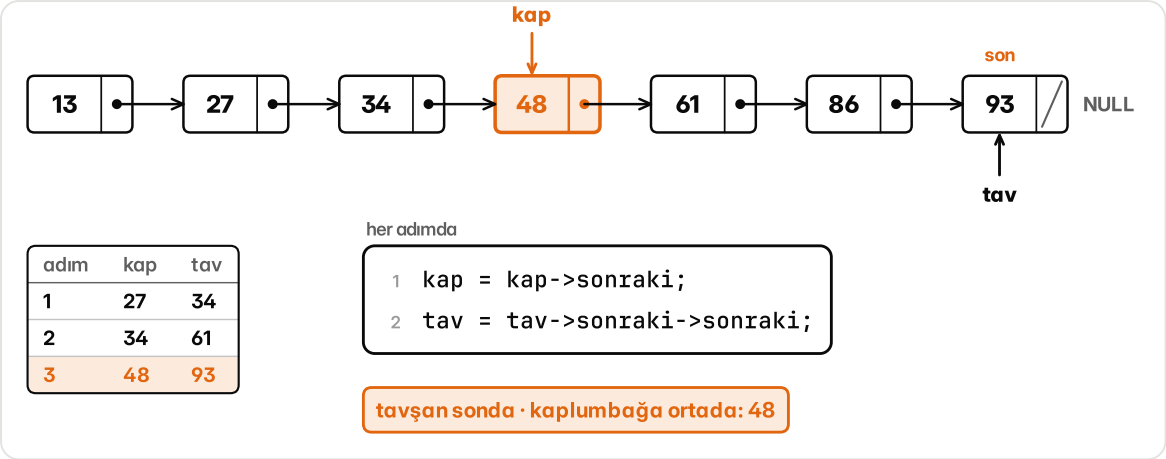


Şekil 1.7 Her adımda bir ok geriye döner; 7 düğüm 7 adımda çevrilir, ek liste yok, yalnız 3 işaretçi.

Ortadaki düğümü bulmak için uzunluğu saymak gerekmez. İki işaretçi baştan çıkar: kaplumbağa (kap) her adımda bir düğüm, tavşan (tav) iki düğüm ilerler. Tavşan sona vardığında kaplumbağa yolun yarısındadır:

```
Dugum *orta(Dugum *bas) {
    Dugum *kap = bas, *tav = bas;
    while (tav != NULL &&
           tav->sonraki != NULL) {
        kap = kap->sonraki;
        tav = tav->sonraki->sonraki;
    }
    return kap;
}
```

7 düğümde 3 adım yeter: kaplumbağa 27, 34, 48'e; tavşan 34, 61, 93'e gider. Tavşan 93'te, kaplumbağa 48'de, tam ortada.



Şekil 1.8 Kaplumbağa bir, tavşan iki düğüm ilerler; 3 adımda tavşan 93'te, kaplumbağa ortadaki 48'de.

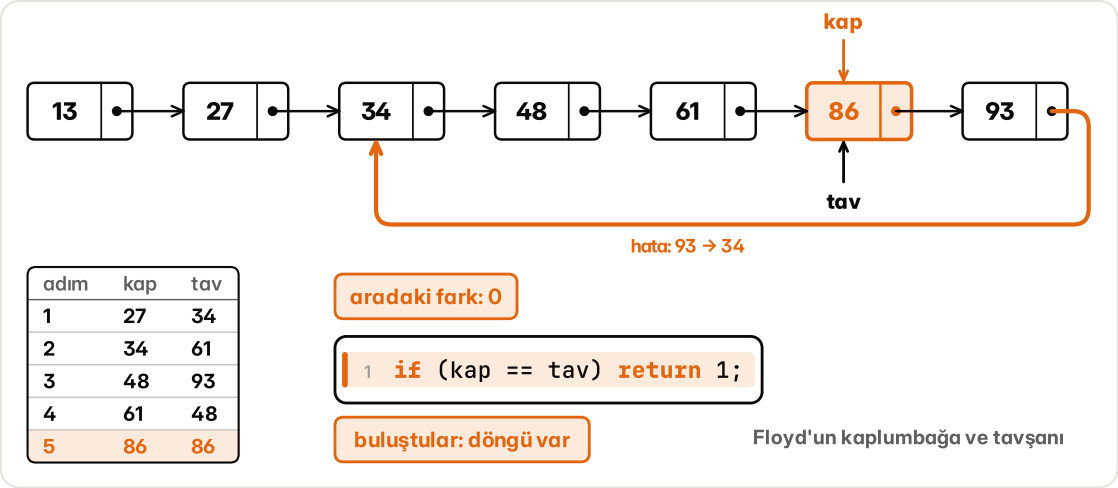
Aynı iki işaretçi döngüyü de bulur. Bir hata yüzünden 93, NULL yerine 34'ü gösteriyor. Döngü olmasaydı tavşan sonunda NULL'a varır, iş biterdi. Döngü varsa ikisi de daireye girer; orada tavşan her adımda kaplumbağaya bir düğüm yaklaşır. Aradaki fark birer birer azaldığı için tavşan onun üstünden atlayamaz, mutlaka yetişir. orta'dan farkı tek satır:

```
int dongu_var(Dugum *bas) {
    Dugum *kap = bas, *tav = bas;
    while (tav != NULL &&
           tav->sonraki != NULL) {
        kap = kap->sonraki;
        tav = tav->sonraki->sonraki;
        if (kap == tav) return 1;
    }
    return 0;
}
```

Program fonksiyonu önce düzgün listede, sonra 93'ün 34'ü gösterdiği listede çağırır ve ikinci listede adımları izler:

```
93 -> NULL: dongu yok
93 -> 34:   dongu var
adim 1: kaplumbaga 27, tavsan 34
adim 2: kaplumbaga 34, tavsan 61
adim 3: kaplumbaga 48, tavsan 93
adim 4: kaplumbaga 61, tavsan 48
adim 5: kaplumbaga 86, tavsan 86
5. adimda bulustular: 86
```

5 adım sonra ikisi 86'da buluşur: buluşma varsa döngü var. Bu yönteme **kaplumbağa ve tavşan** denir; bilgisayar bilimci Floyd'un adıyla anılır.



Şekil 1.9 93 hatayla 34'ü gösterir; daire içinde aradaki fark her adımda bir azalır ve 5. adımda kaplumbağa ile tavşan 86'da buluşur.

Üç problemin çözümü aynı biçimde: liste bir kez yürünür, yanımızda yalnız iki üç işaretçi taşırız. Süre listenin boyuyla büyür; ek bellek liste ne kadar uzun olursa olsun aynıdır. Bu üç soru yazılım iş görüşmelerinde sık sorulur; üçü de işaretçileri doğru sırada ve doğru hızda yürütmekle çözülür.

Problem	Süre	Ek bellek
Ters çevirme	O(n)	O(1)
Ortadaki düğüm	O(n)	O(1)
Döngü bulma	O(n)	O(1)

Özet

- Dinamik dizi dolunca realloc ile daha büyük bir blok alır; kapasite iki katına çıkarsa 1.000.000 eklemede yalnız 18 büyüme ve yaklaşık 1 milyon kopya olur.
- Bazı eklemeler pahalı olsa da ortalaması sabittir: sona ekleme amortize O(1)'dir; sabit miktarda büyüyen dizide bu bozulur.
- Python'ın listesi ve C++'ın vector'ü dinamik dizidir; büyüme orantılı olduğu sürece ortalama maliyet sabit kalır.
- Çift yönlü listede eldeki düğüm 2 atamayla silinir; gözcü düğüm uçlardaki koşulları kaldırır ve listeyi daireye çevirir.
- Düğümün yeri önce aranıyorsa dizi kazanır: bu bilgisayarda 20.000 sıralı eklemede dizi yaklaşık 0,08 sn, liste yaklaşık 1,3 sn sürdü.
- Ters çevirme, ortadaki düğüm ve döngü bulma tek yürüyüşle, O(n) sürede ve O(1) ek bellekle çözülür; kaplumbağa ve tavşan buluşursa döngü vardır.

Sorular

1 **Python'ın listesi (list) içeride hangi yapıyla tutulur?**

A Tek yönlü bağlı liste
B Dinamik dizi
C Çift yönlü bağlı liste
D Dairesel liste

2 **Kapasitesi 4 olan boş bir dinamik diziye 17 eleman ekleniyor. Dizi her dolduğunda kapasite iki katına çıkıyor. Son eklemeden sonra kapasite kaçtır?**

Cevap:

3 **Kapasitesi her dolduğunda iki katına çıkan bir dinamik dizide sona ekleme için hangisi doğrudur?**

A Her ekleme tek adımdır, hiç kopya gerekmez.
B Her ekleme bütün elemanları kopyaladığı için O(n)'dir.
C Bazı eklemeler bütün elemanları kopyalar, ama ortalama maliyet O(1)'dir.
D Ortalama maliyet O(log n)'dir, çünkü dizi log n kez büyür.

4 **Tek yönlü bağlı listede silinecek düğümün adresi elimizde olduğu hâlde silme neden O(n) sürer?**

A Önceki düğümün sonraki adresi değişmelidir ve önceki düğümü bulmak için baştan yürümek gerekir.
B free fonksiyonu listenin bütün düğümlerini tek tek dolaşır.
C Silinen düğümden sonraki bütün düğümler bir öne kaydırılır.
D Silmeden önce listenin uzunluğu sayılmalıdır.

5 **Çift yönlü 12 31 47 56 64 79 listesinde sil(d) çağrılıyor; d, 56'nın düğümünü gösteriyor. Silmeden sonra 64'ün önceki adresi hangi düğümü gösterir? Düğümün değerini yaz.**

Cevap:

6 **Gözcü düğümlü boş bir çift yönlü listede gözcünün önceki ve sonraki adresleri neyi gösterir?**

A İkisi de NULL'dır.
B Önceki NULL, sonraki gözcünün kendisidir.
C Rastgele bir adrestir, liste boş olduğu için önemi yoktur.
D İkisi de gözcünün kendisini gösterir.

7 **Bölümdeki orta fonksiyonu 5 → 8 → 13 → 21 → 34 → NULL listesinde çağrılıyor. Dönen düğümün değeri kaçtır?**

Cevap: _____

8 **ters_cevir'in döngüsünde Dugum *sonra = p→sonraki; satırı neden p→sonraki = once; satırından önce gelir?**

A once işaretçisi NULL olabileceği için önce o denetlenir.

B p'nin oku geriye döndükten sonra sonraki düğümün adresi kaybolur.

C Böylece liste kopyalanır ve eski liste bozulmaz.

D Döngünün durması için sonra'nın NULL olması beklenir.

VERİ YAPILARI

2 Yığın ve kuyruk

Bu bölümde iki sıra düzenini göreceğiz: son giren ilk çıkar (yığın) ve ilk giren ilk çıkar (kuyruk). İkisini de önce dizide, sonra bağlı listeyle kuracağız. Yığının geri al tuşunda, parantez denetiminde ve hesap makinesinde işlem önceliğini çözmede nasıl kullanıldığına; kuyruğun dizide neden halka gibi düşünüldüğüne ve eski bilgisayarların neden bip diye öttüğüne bakacağız.

Ctrl+Z

videoda 12:46

Bir yazıda art arda 4 şey yapalım: bir başlık yazalım, onu kalın yapalım, bir satır silelim, sonra bir kelimenin rengini değiştirelim. Ctrl+Z'ye her basıldığında program neyi geri alacağını nereden bilir?

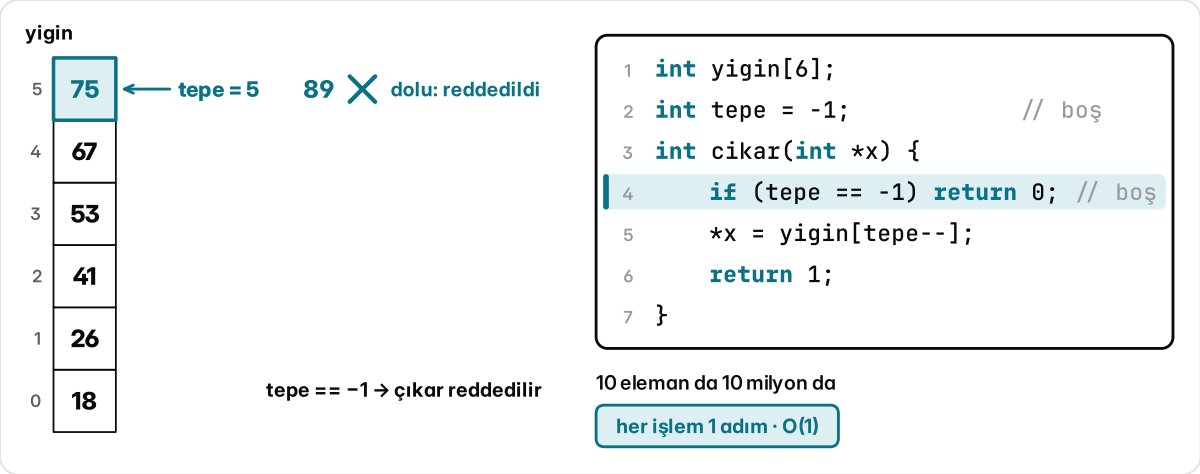
Program her değişikliği, üst üste kâğıt koyar gibi en üste ekler. Ctrl+Z'ye basınca en üstteki kâğıdı, yani son değişikliği alır ve geri çevirir: önce renk geri gelir, sonra silinen satır. Son giren ilk çıkar; bu düzene **yığın** (stack) denir. Yığının 3 işi var: en üste ekle (push), en üsttekini çıkar (pop) ve çıkarmadan tepeye bak (top). Aradaki bir elemana dokunulmaz.

C'de yığın kurmanın en kolay yolu bir dizi ve tepe adında bir tam sayıdır. Tepe, en üstteki elemanın indisidir; yığın boşken −1. Eklerken tepe önce bir artar, sayı o göze yazılır. Çıkarırken tepedeki sayı okunur, sonra tepe bir azalır. cikar sayıyı x işaretçisinin gösterdiği yere yazar; iki fonksiyon da işin olup olmadığını 1 ya da 0 ile söyler:

```
int yigin[6];
int tepe = -1;                // boş
int ekle(int x) {
    if (tepe == 5) return 0;  // dolu
    yigin[++tepe] = x;
    return 1;
}
int cikar(int *x) {
    if (tepe == -1) return 0; // boş
    *x = yigin[tepe--];
    return 1;
}
```

18, 26 ve 41 girince tepe 2 olur; çıkarınca 41 okunur, tepe 1'e iner. Hiçbir eleman yerinden kıpırdamaz; yığında 10 eleman da olsa 10 milyon da, her işlem tek adımdır: O(1). 6 gözlük dizide 7. elemana yer yoktur, ekle onu reddeder; boş yığından çıkarmak da reddedilir. Aşağıdaki çıktıda program önce bunları yapıyor, sonra 41, 53, 67, 75 ve 89'u eklemeye çalışıyor ve yığını boşaltıyor:

```
tepe = 2
cikti: 41, tepe = 1
ekle(41): tamam
ekle(53): tamam
ekle(67): tamam
ekle(75): tamam
ekle(89): dolu, reddedildi
75 67 53 41 26 18
bos yigindan cikar: 0
```



Şekil 2.1 6 gözlük dizide yığın: tepe 5'teyken 89 reddedilir; tepe −1 iken çıkarmak da reddedilir, her işlem O(1).

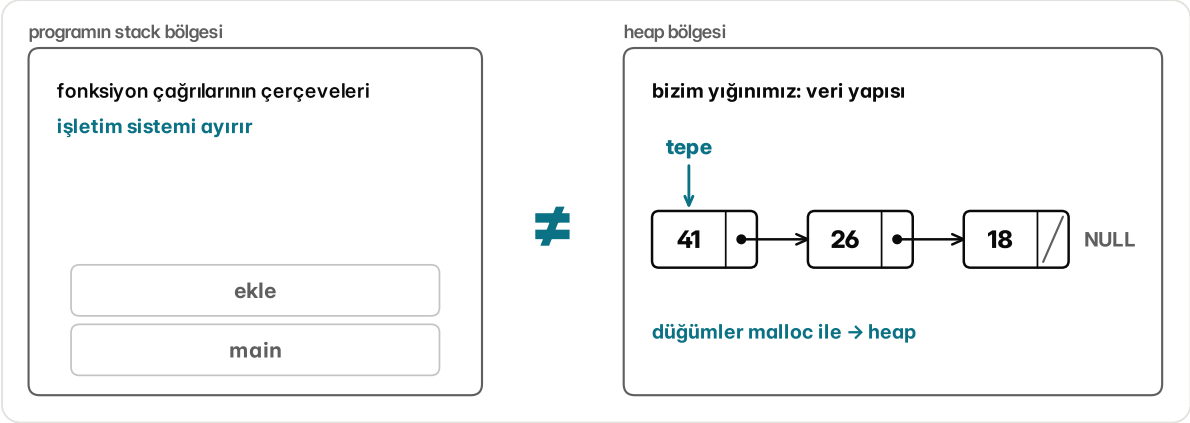
Eleman sayısına sınır istemiyorsak yığını bağlı listeyle kurarız; tepe, listenin ilk düğümüdür ve başta NULL'dır. ekle, malloc ile düğüm boyu kadar yeni bir yer alır ve onu listenin başına bağlar; tepe artık yeni düğümü gösterir. cikar ilk düğümün verisini alır, tepeyi ikinci düğüme geçirir ve eski düğümü free ile geri verir:

```
void ekle(int x) {
    Dugum *d = malloc(sizeof(Dugum));
    d->veri = x;
    d->sonraki = tepe;
    tepe = d;
}
int cikar(void) {
    Dugum *d = tepe;
    int x = d->veri;
    tepe = d->sonraki;
    free(d);
    return x;
}
```

İkisi de birkaç atama, yine O(1). Bedeli, her eleman için fazladan bir adres.

İşlem	Yığın (dizi ya da bağlı liste)
Ekle	O(1)
Çıkar	O(1)
Tepeye bak	O(1)

Bir ad karışıklığına dikkat: programın bellekteki stack bölgesi de, fonksiyon çağrılarını bu kuralla dizdiği için aynı adı taşır. Ama orası işletim sisteminin programa ayırdığı bir bellek alanıdır. Bizim yığınımız ise kendi kurduğumuz bir veri yapısıdır; düğümleri malloc ile alındığı için bellekte heap bölgesinde durur.



Şekil 2.2 Programın stack bölgesinde fonksiyon çağrılarının çerçeveleri durur; bizim yığınımızın düğümleri malloc ile alındığı için heap bölgesindedir.

Tarayıcının geri tuşu da bir yığın gibi çalışır: açılan her sayfa yığına eklenir, geri tuşu en üsttekini çıkarır ve bir önceki sayfaya dönülür.

Parantezlerin doğru kapanıp kapanmadığı da yığınla denetlenir. Açılan her parantez yığına girer. Kapanan gelince tepedeki açılanla eşleşmeli; eşleşirse tepedeki çıkar. ([ ] { ( ) }) satırında her kapanan kendi eşini tepede bulur ve satır bitince yığın boş kalır: satır doğru. ([ )] satırında köşeli parantez açıkken yuvarlak parantez kapanıyor; tepede [var, eşleşmiyor: hata. (( ) satırı bitince yığında (kalır, yani biri kapanmamıştır. Kod yazarken unutulan bir parantez için alınan uyarı da bu tür bir denetimden gelir.



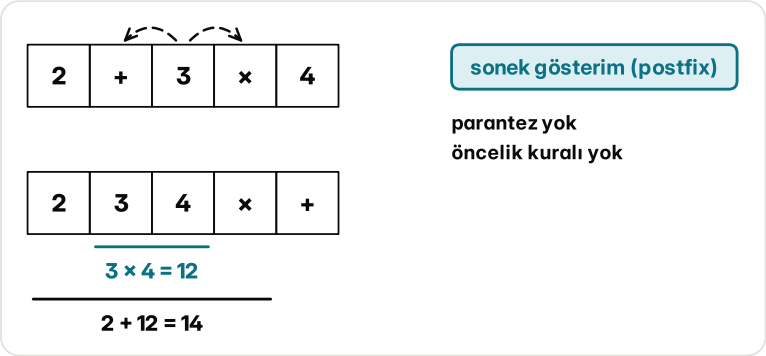
Şekil 2.3 Açılan parantez yığına girer, kapanan tepedekiyle eşleşmelidir: ilk satır doğru, ikincide [ile) eşleşmez, üçüncüde (açık kalır.

2 + 3 × 4

videoda 15:04

Çoğu basit cep hesap makinesinde 2 + 3 × 4, 20 çıkar. Makine her işlem tuşunda o ana kadarkini hesaplar: 2 + 3, 5; × 4, 20. Bilimsel hesap makinesi ise 14 bulur, çünkü önce çarpar. Önce çarpması gerektiğini nasıl bilir?

Zorluk, işaretin sayıların arasında durmasında: 3'ün önce artıyla mı, çarpıyla mı işleneceği, ötekilere bakmadan anlaşılmaz. Çözüm, işareti sayılardan sonra yazmak: 2 3 4 × +. Bunu soldan okuyan, çarpıyı görünce hemen önündeki iki sayıyı, 3 ile 4'ü çarpar; artı gelince 2'yi bu sonuçla toplar. Buna **sonek gösterim** (postfix) denir. Ne parantez gerekir ne öncelik kuralı; sıra, yazılışın kendisindedir.



Şekil 2.4 2 + 3 × 4 sonek gösterimde 2 3 4 × + olur: önce 3 × 4 = 12, sonra 2 + 12 = 14.

Sonek ifade, son koyduğunu ilk aldığımız bir sırayla, yani bir yığınla, soldan sağa tek geçişte hesaplanır. Kural 2 satır: sayı gelirse yığına ekle; işaret gelirse tepeden 2 sayı çıkar, işlemi yap, sonucu yığına koy. 2, 3 ve 4 yığına girer. Çarpı gelince 4 ve 3 çıkar, 12 girer. Artı gelince 12 ve 2 çıkar, 14 girer. İfade bitince yığında tek sayı kalır: sonuç.

Aşağıdaki hesapla fonksiyonu tek basamaklı sayılarla çalışır. isdigit bir karakterin rakam olup olmadığına bakar; *s - '0' rakam karakterini sayıya çevirir. islem(a, op, b) dört işlemden birini yapan küçük bir yardımcıdır; C'de çarpı * ile yazılır:

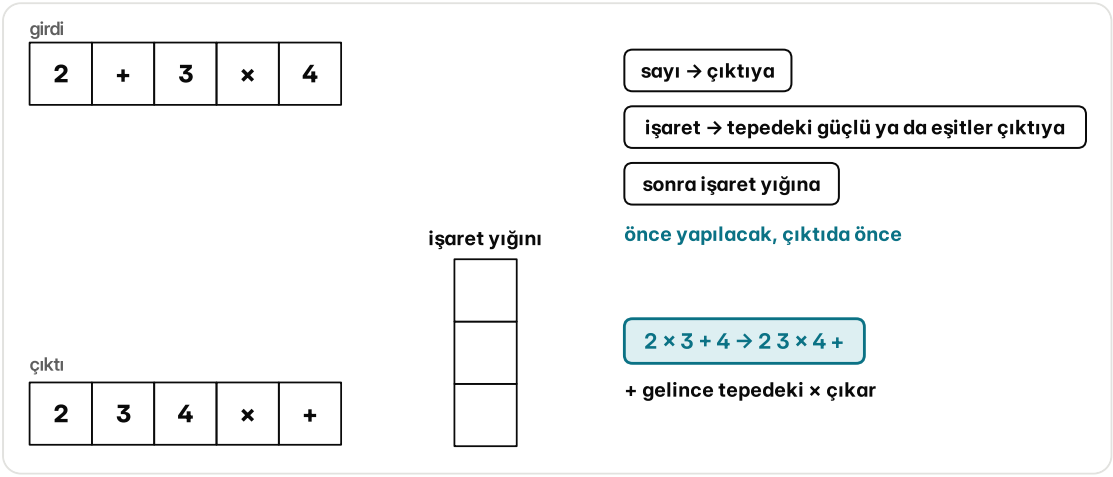
```
int hesapla(const char *s) {
    int y[16], t = -1;
    for (; *s; s++) {
        if (isdigit(*s)) y[++t] = *s - '0';
        else if (*s != ' ') {
            int b = y[t--];   // sağdaki
            int a = y[t--];   // soldaki
            y[++t] = islem(a, *s, b);
        }
    }
    return y[t];
}
```

Dört ifadeyle çağrılınca:

```
2 3 4 * +   = 14
2 3 + 4 *   = 20
2 3 * 4 +   = 10
9 4 -       = 5
```

Çıkarmada sıraya dikkat: 9 4 − ifadesinde önce 4 çıkar, sonra 9. İlk çıkan sağdaki sayıdır (b), sonuç 9 − 4 = 5.

Peki 2 + 3 × 4'ü bu yazıma kim çevirir? Yine bir yığın, bu kez işaretleri bekletir. Sayılar doğrudan çıktıya gider. Bir işaret gelince, tepede ondan güçlü ya da ona eşit işaretler varsa önce onlar çıktıya alınır, sonra yeni işaret yığına girer; önce yapılacak olan, çıktıda da önce yer alsın diye. 2 çıktıya, artı yığına, 3 çıktıya. Çarpı gelir; tepedeki artı ondan zayıf, beklemeye devam eder, çarpı onun üstüne biner. 4 çıktıya. İfade bitince yığın tepeden boşalır: önce çarpı, sonra artı. Sonuç: 2 3 4 × +. 2 × 3 + 4 olsaydı, artı geldiğinde tepedeki çarpı daha güçlü olurdu ve önce o çıkardı: 2 3 × 4 +.

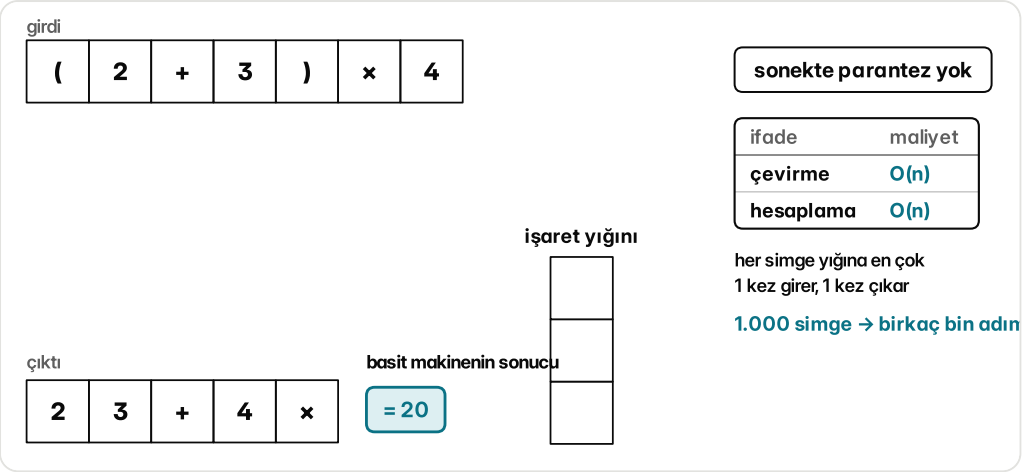


Şekil 2.5 Sayılar çıktıya gider, işaretler yığında bekler; × gelince zayıf olan + bekler, ifade bitince yığın tepeden boşalır: 2 3 4 × +.

Açılan parantez de yığına girer ve bir duvar gibi durur: üstüne gelen işaret, onun altındakileri çıkaramaz. Kapanan parantez gelince duvara kadar ne varsa çıktıya alınır, duvar da kalkar. (2 + 3) × 4'te çıktı 2 3 + 4 × olur ve 20 verir. Sonek yazımda parantez yoktur; onun işini sıra görür.

Çevirme de hesaplama da her simgeyi bir kez okur; her simge yığına en çok bir kez girer, bir kez çıkar. 1000 simgelik ifade birkaç bin adımda biter.

İş	Maliyet
Sonek'e çevirme	O(n)
Sonek ifadeyi hesaplama	O(n)



Şekil 2.6 (2 + 3) × 4 sonek'e 2 3 + 4 × diye çevrilir ve 20 verir; çevirme de hesaplama da O(n).

Python da kargo + adet * fiyat satırını çalıştırmadan önce bu sıraya çevirir. Satırın ara koduna Python 3.14'ün dis modülüyle bakınca komutlar şu sırayla çıkar (işlem numaraları atıldı):

```
LOAD_NAME kargo
LOAD_NAME adet
LOAD_NAME fiyat
BINARY_OP *
BINARY_OP +
```

kargo, adet, fiyat, çarp, topla. Python'ın yorumlayıcısı bu komutları sırayla, bir yığının üstünde işler.

Bip

videoda 17:37

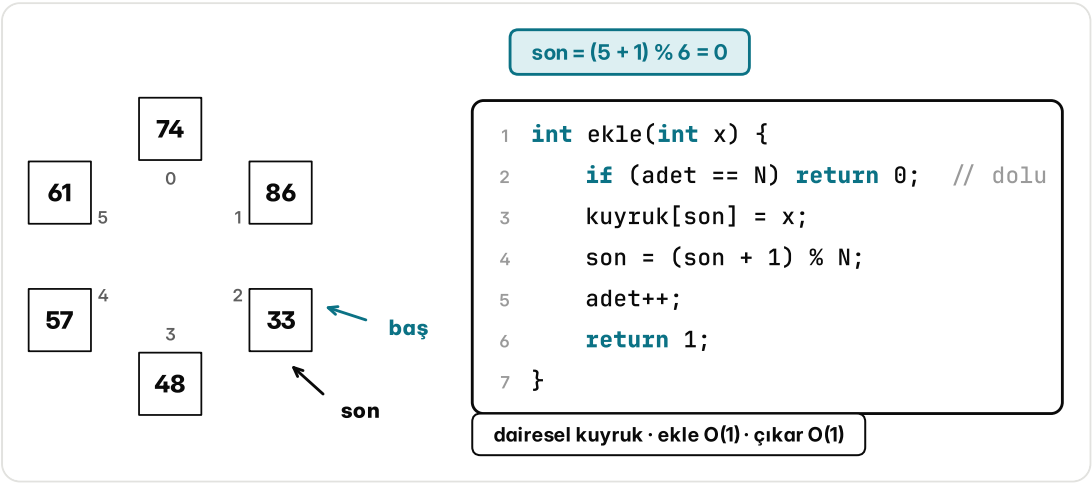
Eski bilgisayarlarda bir program meşgulken tuşlara art arda basınca, bir noktadan sonra her tuşta bip sesi gelirdi ve o tuşlar ekrana hiç düşmezdi. Bilgisayar neden öterdi, tuşlar nerede bekliyordu?

Basılan tuşlar program okuyana kadar bekler ve basıldıkları sırayla okunur. İlk giren ilk çıkar; bu düzene **kuyruk** (queue) denir. İki işi var: sona ekle (enqueue), baştan çıkar (dequeue). Yazıcıya gönderilen belgeler ve işletim sisteminde işlemciyi bekleyen görevler de böyle sıraya girer.

Kuyruğu 6 gözlük bir dizide tutalım: 16, 29, 33 ve 48 girsin. Baş hep 0. gözde kalırsa, her çıkışta geridekilerin hepsi bir öne kaymalıdır; 1000 elemanlı kuyrukta her çıkışta neredeyse 1000 kopya, yani O(n). Kaydırmak yerine iki indis tutarız: baş sıradakini, son da yeni elemanın gireceği gözü gösterir. 16 ve 29 çıkınca baş 2'ye geçer; kimse yerinden kıpırdamaz.

Ama önde iki göz boş kaldı. 57, 4 numaralı göze; 61, 5 numaralı göze girer. Son 6'ya, yani dizinin dışına varacak; önde yer var ama yeni eleman giremez. Çözüm, diziyi bir halka gibi düşünmek: 5 numaralı gözden sonra 0 gelir. Bunu bölümden kalan (%) yapar: 5 + 1 = 6, 6'yı 6'ya bölünce kalan 0; son başa sarar. 74, 0. göze; 86, 1. göze girer. Buna **dairesel kuyruk** denir; ekle de çıkar da tek adımdır: O(1).

```
#define N 6
int kuyruk[N];
int bas = 0, son = 0, adet = 0;

int ekle(int x) {
    if (adet == N) return 0;  // dolu
    kuyruk[son] = x;
    son = (son + 1) % N;
    adet++;
    return 1;
}
```



Şekil 2.7 Dizi halkaya kıvrılır, 5'ten sonra 0 gelir: son = (5 + 1) % 6 = 0; 74 0. göze, 86 1. göze girer.

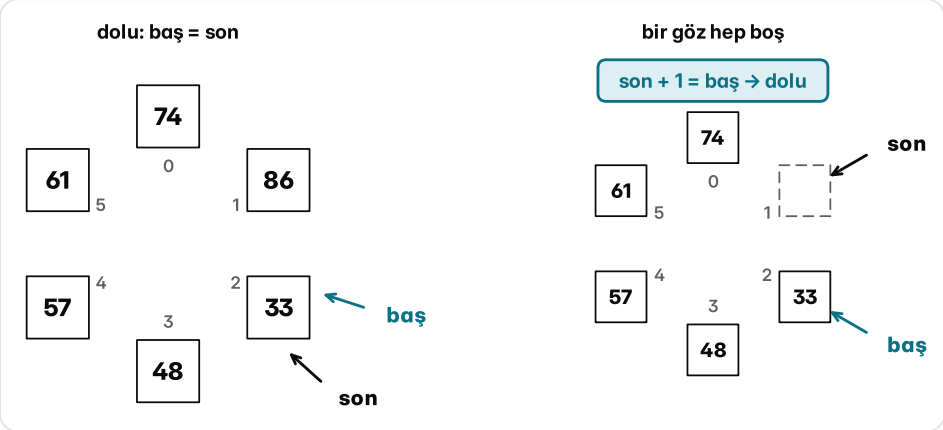
Şimdi 6 göz de dolu ve son, başın üstüne geldi. Oysa kuyruk boşken de baş ile son aynı gözdeydi. İkisini ayırmak için eleman sayısı ayrıca tutulur: adet 6'ysa dolu, 0'sa boş. cikar da başı aynı biçimde halkada yürütür:

```
int cikar(int *x) {
    if (adet == 0) return 0;  // boş
    *x = kuyruk[bas];
    bas = (bas + 1) % N;
    adet--;
    return 1;
}
```

Program her adımda baş, son, adet ve gözleri yazar (nokta boş göz demek):

```
bos        bas=0 son=0 adet=0 |  .  .  .  .  .  .
ekle 16    bas=0 son=1 adet=1 | 16  .  .  .  .  .
ekle 29    bas=0 son=2 adet=2 | 16 29  .  .  .  .
ekle 33    bas=0 son=3 adet=3 | 16 29 33  .  .  .
ekle 48    bas=0 son=4 adet=4 | 16 29 33 48  .  .
cikar 16   bas=1 son=4 adet=3 |  . 29 33 48  .  .
cikar 29   bas=2 son=4 adet=2 |  .  . 33 48  .  .
ekle 57    bas=2 son=5 adet=3 |  .  . 33 48 57  .
ekle 61    bas=2 son=0 adet=4 |  .  . 33 48 57 61
ekle 74    bas=2 son=1 adet=5 | 74  . 33 48 57 61
ekle 86    bas=2 son=2 adet=6 | 74 86 33 48 57 61
ekle 90: dolu, reddedildi
sira: 33 48 57 61 74 86
bos        bas=2 son=2 adet=0 |  .  .  .  .  .  .
```

Dolu kuyrukta da boşalmış kuyrukta da baş ve son 2'dedir; ikisini adet ayırır. Başka bir yol, bir gözü hep boş bırakmaktır: son, başın hemen arkasına gelince kuyruk dolu sayılır.



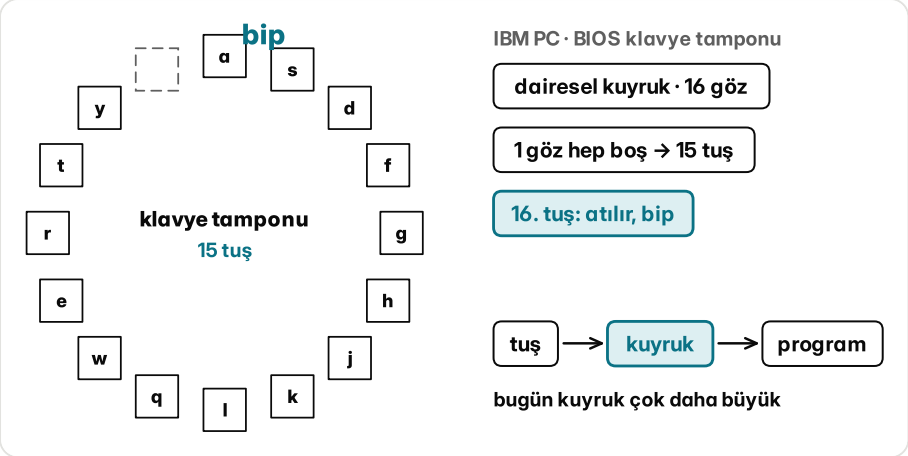
Şekil 2.8 Dolu kuyrukta baş ile son aynı gözdedir; bir göz hep boş bırakılırsa son + 1 = baş olduğunda kuyruk dolu sayılır.

Kuyruğu bağlı listeyle kurarsak iki işaretçi tutarız: baş ilk düğümü, son da son düğümü gösterir; ikisi de başta NULL'dır. ekle yeni düğümü sondakinin arkasına bağlar, cikar baştaki düğümü alır; ikisi de O(1). Son işaretçisi olmasa her eklemede listenin sonuna kadar yürümek gerekirdi:

```
void ekle(int x) {
    Dugum *d = malloc(sizeof(Dugum));
    d->veri = x;
    d->sonraki = NULL;
    if (son == NULL) bas = d;
    else son->sonraki = d;
    son = d;
}
```

İşlem	Kaydırmalı dizi	Dairesel kuyruk	Bağlı liste (baş ve son)
Ekle	O(1)	O(1)	O(1)
Çıkar	O(n)	O(1)	O(1)

Eski bilgisayarlarda (IBM PC) basılan tuşlar da 16 gözlük dairesel bir kuyrukta beklerdi. Bir göz hep boş kaldığı için 15 tuş birikince kuyruk dolardı; yeni tuşa yer yoktu, bilgisayar onu atıp bip diye öterdi. Bugün kuyruk çok daha büyük, ama basılan her tuş hâlâ program okuyana kadar bir kuyrukta sırasını bekler.



Şekil 2.9 IBM PC'nin klavye tamponu 16 gözlük dairesel kuyruktur; bir göz hep boş kaldığı için 15 tuş sığar, 16. tuş atılır ve bip çalar.

Özet

- Yığında son giren ilk çıkar; ekle, çıkar ve tepeye bak dizide de bağlı listede de O(1)'dir.
- Veri yapısı olan yığın, programın bellekteki stack bölgesiyle yalnız kuralı ve adı paylaşır; malloc ile alınan düğümleri heap'te durur.
- Geri al, tarayıcının geri tuşu ve parantez denetimi yığınla çalışır: kapanan parantez tepedeki açılanla eşleşmelidir.
- Sonek gösterimde işaret sayılardan sonra gelir; ifade bir yığınla tek geçişte hesaplanır, araek ifade de işaretleri yığında bekleterek O(n)'de sonek'e çevrilir.
- Kuyrukta ilk giren ilk çıkar; dizide baş ve son indisleri halka gibi döndürülürse (son = (son + 1) % N) ekle ve çıkar O(1) olur.
- Dolu ile boş kuyrukta baş ve son aynı gözdedir; ayrım eleman sayısıyla ya da bir gözü hep boş bırakarak yapılır.

Sorular

1 **Boş bir yığına sırayla 5, 8 ve 3 ekleniyor, sonra bir kez çıkar yapılıyor, en sonda 7 ekleniyor. Yığındaki sayıları tepeden aşağı doğru yaz.**

Cevap:

2 **Bölümdeki dizi yığınında (int tepe = -1; başta) sırayla ekle(18); ekle(26); ekle(41); cikar(&x); ekle(53); çağrılıyor. Sonunda tepe kaçtır?**

A 1
B 2
C 3
D 4

3 **Bağlı listeyle kurulan bir yığının düğümleri malloc ile alınıyor. Bu düğümler bellekte nerede durur?**

A Heap bölgesinde
B Stack bölgesinde, çünkü yığının İngilizcesi stack'tir
C Programın kod bölgesinde
D İşlemcinin yazmaçlarında

4 **Parantezler yığınla denetleniyor: açılan yığına girer, kapanan tepedekiyle eşleşmelidir, satır bitince yığın boş kalmalıdır. Hangi satır doğru çıkar?**

A { ( ] }
B ([ )]
C [( ) { }]
D (( )

5 **Sonek ifadeyi yığınla hesapla: 5 1 2 + 4 × + 3 −**

Cevap: _____

6 **(5 + 1) × 2 ifadesini sonek gösterimle yaz (simgeleri boşlukla ayır, çarpı için × ya da * kullanabilirsin).**

Cevap: _____

7 **Bağlı listeyle kurulan bir kuyrukta yalnız baş işaretçisi tutuluyor, son işaretçisi tutulmuyor. Hangi işlem O(n) olur?**

A Baştan çıkarmak
B Baştaki elemana bakmak
C Sona eklemek
D Kuyruğun boş olup olmadığına bakmak

8 **6 gözlük dairesel kuyrukta (son = (son + 1) % 6) bas = 4, son = 4, adet = 0. Kuyruğa sırayla 10, 20 ve 30 ekleniyor. son kaç olur?**

A 7
B 0
C 2
D 1

VERİ YAPILARI

3 Hash tabloları

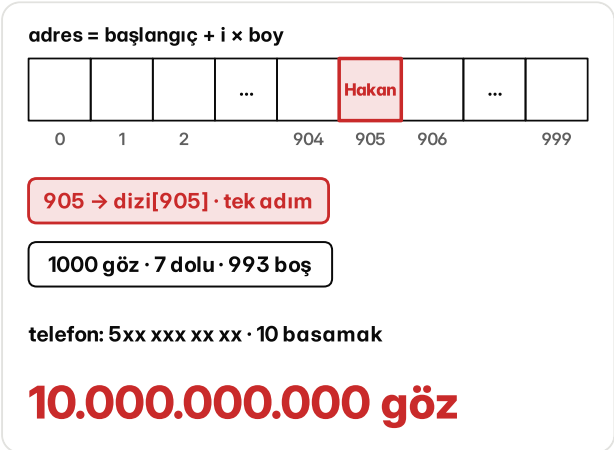
Bu bölümde bir kaydı listede aramadan, yerini anahtarından hesaplayarak bulan yapıyı, hash tablosunu göreceğiz. Anahtarı tablo boyuna indiren hash fonksiyonunu, iki anahtarın aynı göze düşmesi demek olan çakışmayı ve onu çözmenin iki yolunu, zincirleme ile açık adreslemeyi tanıyacağız.

905 nerede?

videoda 19:47

Yedi kişilik bir ofiste herkesin üç basamaklı bir dahili numarası var: 214, 530, 128, 905, 341, 879 ve 657. 905 numaralı kişiyi bulmak için listeye tek tek mi bakmak gerekir, yoksa yerini numaradan hesaplayabilir miyiz?

Dizide sırası bilinen elemana tek adımda gidilir, çünkü adresi hesaplanır: adres = başlangıç + i × boy. Aranan şey sıranın kendisi olsaydı arama da tek adım olurdu. Numarayı doğrudan indeks yapabiliriz: 905 numaralı kişi dizi[905]'te durur. Ama bunun için yedi kişiye 1.000 gözlü bir dizi gerekir; gözlerin 993'ü boş kalır. Anahtar on basamaklı bir telefon numarası olsaydı, her numaraya bir göz ayırmak 10 milyar göz ederdi.

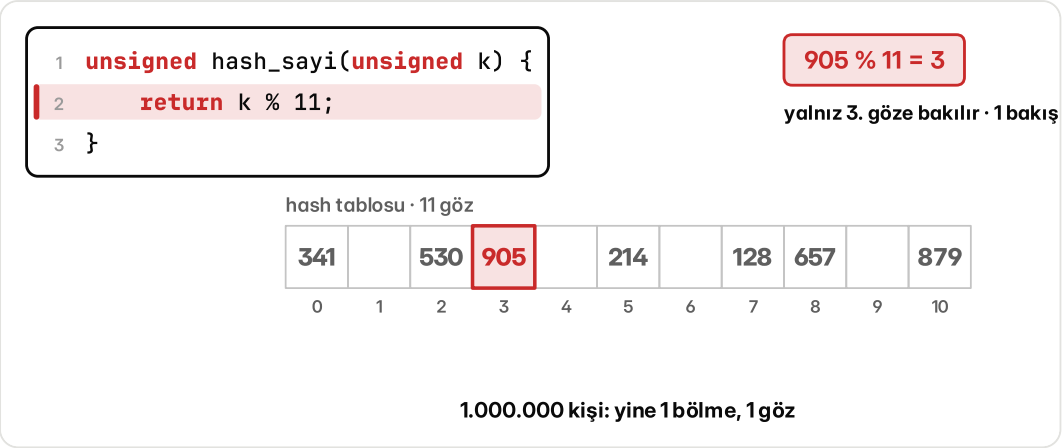


Şekil 3.1 Numara indeks olunca 905 tek adımda bulunur, ama 1.000 gözün yalnız 7'si doludur; on basamaklı telefon numarası 10 milyar göz ister.

Çözüm, büyük anahtarı tablo boyuna indiren bir hesaptır. Bu hesaba **hash fonksiyonu**, kayıtların durduğu diziye de **hash tablosu** denir. En basit hash, anahtarın tablo boyuna bölümünden kalandır. 11 gözlü bir tabloda 214, 11'e bölününce 5 kalır (214 = 19 × 11 + 5); kişi 5. göze yazılır. Öteki numaralar da sırayla 2., 7., 3., 0., 10. ve 8. gözlere düşer. 905'i ararken aynı hesap yapılır: 905 % 11 = 3, yalnız 3. göze bakılır. C'de % işleci bölümden kalanı verir:

```
unsigned hash_sayi(unsigned k) {
    return k % 11;
}
```

unsigned, eksi değer almayan tam sayı tipidir. Tablo bir milyon kişiyi tutsa da iş yine tek bölme ve tek göze bakmaktır.



Şekil 3.2 11 gözlü hash tablosunda yedi numara yedi ayrı gözdedir; 905 % 11 = 3 olduğu için yalnız 3. göze bakılır.

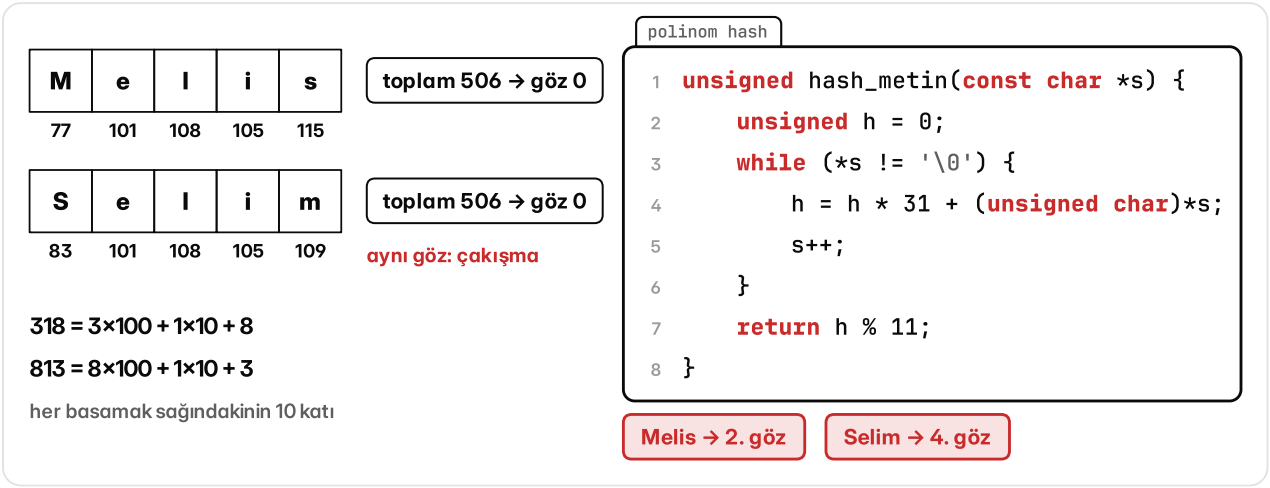
Rehberde anahtar isimdir; isim önce sayıya çevrilir. Bilgisayarda her harfin bir sayı kodu vardır: Melis'in ilk harfi M 77, ikincisi e 101. Kodları toplamak yetmez. Melis ile Selim aynı harflerden oluşur, toplamları da aynıdır: 506. 506 % 11 = 0 olduğu için ikisi de 0. göze düşer. Sayılarda bu sorun yoktur: 318 ile 813 aynı rakamlardan oluşur ama farklıdır, çünkü her basamak sağındakinin 10 katı değer taşır (318 = 3 × 100 + 1 × 10 + 8). İsimde de hesap, her harften önce o ana kadarki toplamı bir sayıyla, örneğin 31'le çarpar:

```
unsigned hash_metin(const char *s) {
    unsigned h = 0;
    while (*s != '\0') {
        h = h * 31 + (unsigned char)*s;
        s++;
    }
    return h % 11;
}
```

Döngü metnin sonundaki '\0' karakterine kadar harf harf ilerler; (unsigned char)*s o harfin kodunu verir. Böylece harfin sıradaki yeri sonucu değiştirir. Buna **polinom hash** denir. Programdaki harf_toplami fonksiyonu aynı döngüyü çarpmadan, yalnız toplayarak yapar. Yedi numara ve iki isim bu fonksiyonlardan geçince program şunu yazar:

Çıktı:

```
214 -> goz 5
530 -> goz 2
128 -> goz 7
905 -> goz 3
341 -> goz 0
879 -> goz 10
657 -> goz 8
Melis: toplam 506 -> goz 0, polinom -> goz 2
Selim: toplam 506 -> goz 0, polinom -> goz 4
```



Şekil 3.3 Melis ve Selim'in harf kodlarının toplamı 506'dır ve ikisi aynı göze düşer; polinom hash Melis'i 2., Selim'i 4. göze yazar.

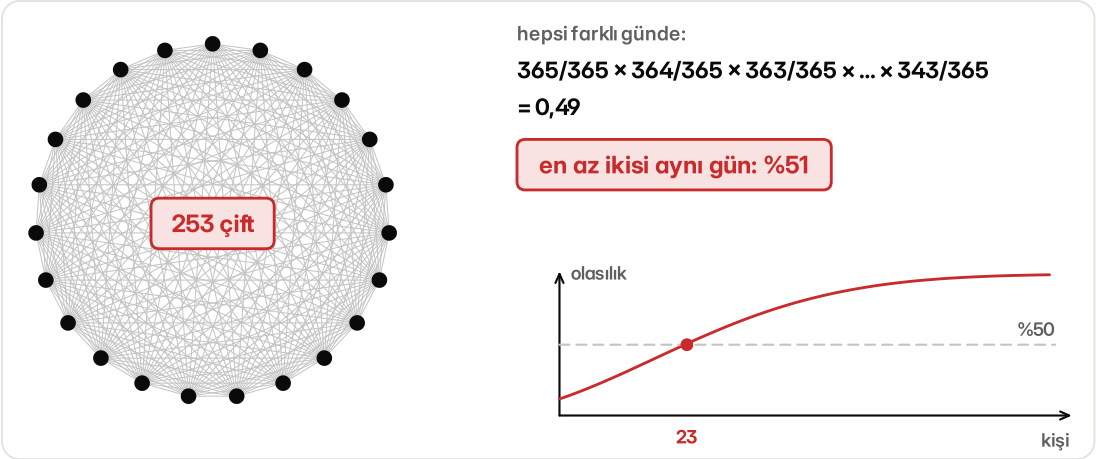
Hash tablosunun tek bir sorunu var: iki farklı anahtar aynı göze düşebilir. Buna **çakışma** denir. Her hash tablosu çakışmayı bir yolla çözer; bu yüzden tek adımda bulmak bir ortalamadır. Ekleme, arama ve silme ortalamada O(1) sürer. Şifre saklamadan bildiğimiz hash ise aynı adı taşıyan başka bir araçtır: orada amaç hızlı bulmak değil, sonuçtan şifreye geri dönülememesidir.

Kaç kişi?

videoda 21:34

Yılda 365 gün var. Bir sınıfta kaç kişi olursa iki kişinin aynı gün doğmuş olma olasılığı yarıyı geçer: 180 mi, 100 mü?

Cevap 23. Sezgi yanıltır, çünkü saymamız gereken kişiler değil çiftlerdir: 23 kişiden 23 × 22 / 2 = 253 çift çıkar ve her çift aynı günü paylaşabilir. Herkesin farklı günde doğma olasılığını kişi kişi çarparız. İkinci kişi birinciden farklı günde doğmalı: 365'te 364. Her yeni kişide pay bir azalır. 23. kişiye kadar çarpınca 0,49 çıkar; demek ki en az iki kişinin aynı gün doğma olasılığı %51'dir. Hesapta yılın 365 gün olduğu ve her günün eşit olası olduğu varsayıldı.



Şekil 3.4 23 kişiden 253 çift çıkar; herkesin farklı günde doğma olasılığı 0,49, en az ikisinin aynı gün doğma olasılığı %51'dir.

Hash tablosu da aynı durumdadır: günler gözler, kişiler anahtarlardır. 365 gözlü tabloya 23 anahtar konunca, tablo yalnız %6 doluyken iki anahtarın aynı göze düşme olasılığı yarıyı geçer. 1.000.000 gözlü tabloda bu eşik 1.178 anahtardır. Çakışma kaçınılmazdır; iyi bir tablo onu önlemeye değil, çözmeye çalışır.

En yaygın çözüm **zincirleme**dir (chaining): her göz tek bir kayıt değil, bir bağlı listenin başını tutar. 7 gözlü bir tabloya, 7'ye bölümden kalanla 16, 29, 38, 44, 57, 61 ve 93'ü ekleyelim. 16, 44 ve 93'ün üçü de 2. göze düşer. Yeni gelen listenin başına eklenir: yeni düğüm gözün eski başını gösterir, göz de yeni düğümü. Böylece listede yürümek gerekmez. Düğüm, bir sayı (veri) ve sonraki düğümün adresini (sonraki) tutan bir struct'tır; malloc yeni düğüme bellekte yer ayırır. Tablo dizisi program başlarken NULL'larla, yani hiçbir yeri göstermeyen adreslerle doludur:

```
typedef struct dugum {
    int veri;
    struct dugum *sonraki;
} Dugum;

Dugum *tablo[7];   /* hepsi NULL */

void ekle(int x) {
    int i = x % 7;
    Dugum *yeni = malloc(sizeof(Dugum));
    yeni->veri = x;
    yeni->sonraki = tablo[i];
    tablo[i] = yeni;
}
```

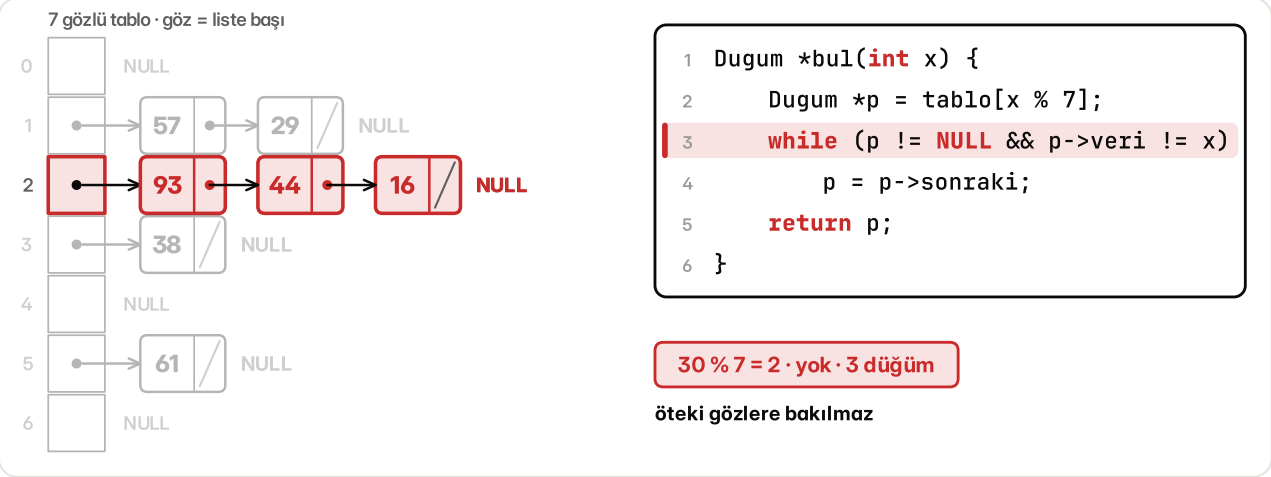
Ararken yalnız anahtarın gözündeki listede yürünür:

```
Dugum *bul(int x) {
    Dugum *p = tablo[x % 7];
    while (p != NULL && p->veri != x)
        p = p->sonraki;
    return p;
}
```

44'ü ararken 2. gözün listesinde önce 93'e, sonra 44'e bakılır. 30 tabloda yok: 30 % 7 = 2, listede 93, 44 ve 16'ya bakılır ve listenin sonuna varılınca arama biter; öteki gözlere hiç bakılmaz. Program tablonun son hâlini ve iki aramayı yazar:

Çıktı:

```
goz 0: NULL
goz 1: 57 -> 29 -> NULL
goz 2: 93 -> 44 -> 16 -> NULL
goz 3: 38 -> NULL
goz 4: NULL
goz 5: 61 -> NULL
goz 6: NULL
44: bulundu, 2 dugume bakildi
30: yok, 3 dugume bakildi
```

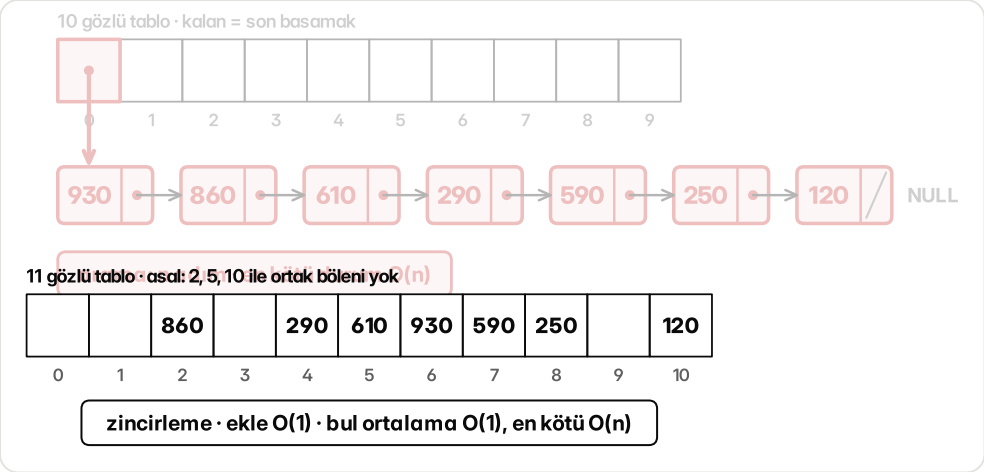


Şekil 3.5 7 gözlü tabloda her göz bir listenin başıdır; 30'u arayan 2. gözün listesinde 3 düğüme bakar ve öteki gözlere dokunmaz.

Anahtar sayısının göz sayısına oranına **doluluk oranı** (load factor) denir; zincirlemede ortalama liste boyu tam bu orandır. Burada 7 gözde 7 anahtar var: oran 1, liste boyları 0, 2, 3, 1, 0, 1, 0 ve ortalaması 1. Oran 1 civarında kaldıkça arama 1-2 adım sürer; anahtarlar çoğalınca tablo büyütülür.

Kötü bir hash bütün bunları bozar. 10 gözlü tabloda bölümden kalan, sayının son basamağıdır. Anahtarlar 120, 250, 290, 590, 610, 860 ve 930 gibi sonu 0'la biten fiyatlarsa hepsi 0. göze düşer ve tablo tek bir uzun listeye döner: 1 milyon anahtarda arama 1 milyon adım sürer, en kötü durum

O(n) olur. 11 gibi asal bir tablo boyu bu düzeni kırar. Asal sayının 2, 5 ve 10 ile ortak böleni yoktur; aynı yedi fiyat 10, 8, 4, 7, 5, 2 ve 6. gözlere dağılır. Ortalamadaki tek adımı, anahtarları gözlere eşit dağıtan iyi bir hash sağlar.



Şekil 3.6 10 gözlü tabloda sonu 0'la biten yedi fiyat tek listede toplanır; 11 gözlü tabloda yedi ayrı göze dağılır.

Kayıp 82

videoda 23:41

Bir hash tablosunda 82, 7. gözde duruyor. Tablodan 49'u sildik ve arama artık 82'yi bulamıyor. 82'ye hiç dokunmadık; neden?

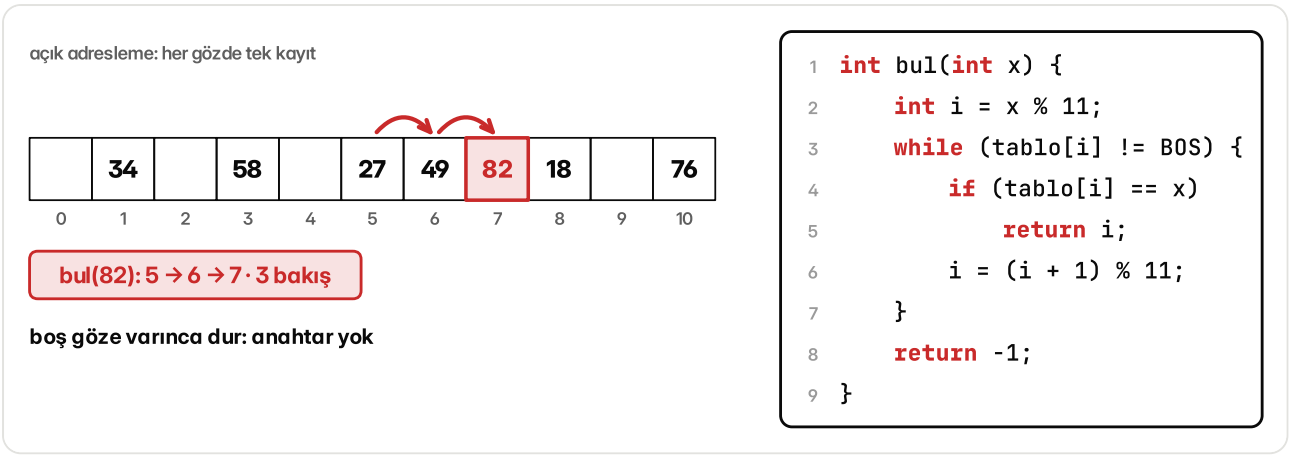
Bu tabloda her gözde tek kayıt durur, liste yoktur; buna **açık adresleme** (open addressing) denir. Tablo 11 gözlü, hash 11'e bölümden kalan. Anahtarlar sırayla 34, 58, 27, 49, 82, 18 ve 76. 27, 5. göze yazılır. 49'un kalanı da 5; göz dolu olduğu için bir sağdaki 6. göze yazılır. 82'nin kalanı da 5: 5 ve 6 dolu, 82 7. göze yazılır. Göz doluysa bir sağdakine geçmeye **doğrusal yoklama** (linear probing) denir; son gözden sonra başa, 0. göze dönülür.

Arama aynı yolu izler; anahtarı bulunca ya da boş bir göze varınca durur. Anahtar tabloda olsaydı, ekleme onu en geç o boş göze yazardı. Boş göz BOS, silinmiş göz SILINDI değeriyle işaretlenir:

```
#define BOS -1
#define SILINDI -2
```

Tablo 11 tam sayılık bir dizidir (int tablo[11]); arama şöyle yazılır:

```
int bul(int x) {
    int i = x % 11;
    while (tablo[i] != BOS) {
        if (tablo[i] == x)
            return i;
        i = (i + 1) % 11;
    }
    return -1;
}
```



Şekil 3.7 Doğrusal yoklamada dolu gözün sağındakine geçilir; bul(82) 5, 6 ve 7. gözlere bakar.

49'u silip 6. gözü BOS yaptık. 82'yi ararken 5. gözde 27 var, 6. göz boş: arama burada durur ve "yok" der. Gözü boşaltan program şunu yazar:

Çıktı:

```
82 -> goz 7
49 silindi; 82 -> -1 (yok)
tabloda 7. goz: 82
```

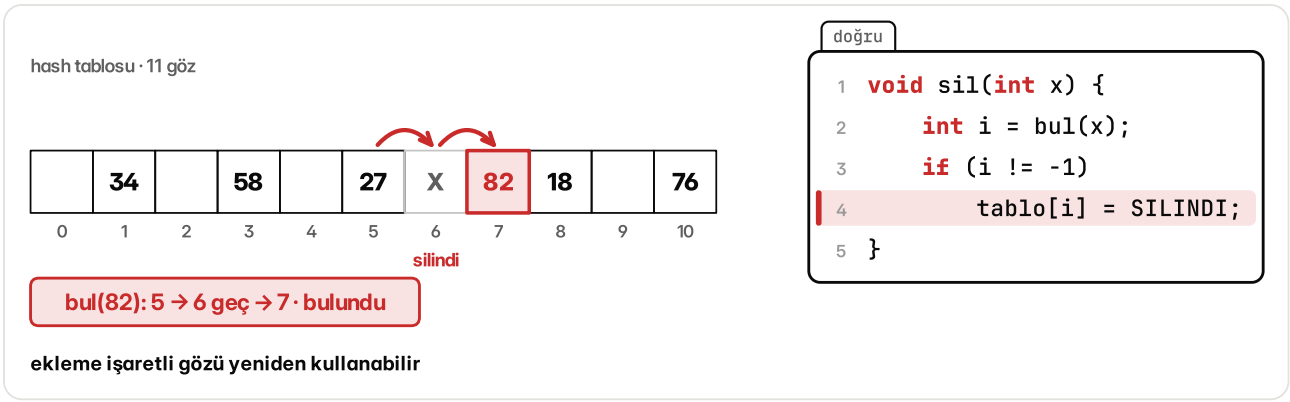
Çözüm, silinen gözü boşaltmak değil, "silindi" diye işaretlemektir:

```
void sil(int x) {
    int i = bul(x);
    if (i != -1)
        tablo[i] = SILINDI;
}
```

bul yalnız BOS'ta durduğu için işaretli gözün üstünden geçer ve 82'yi 7. gözde bulur. Ekleme ise işaretli gözü yeniden kullanabilir: kalanı 5 olan 60, 5. göz dolu olduğu için 6. gözdeki işaretin yerine yazılır. Çıktıda X silindi işaretidir:

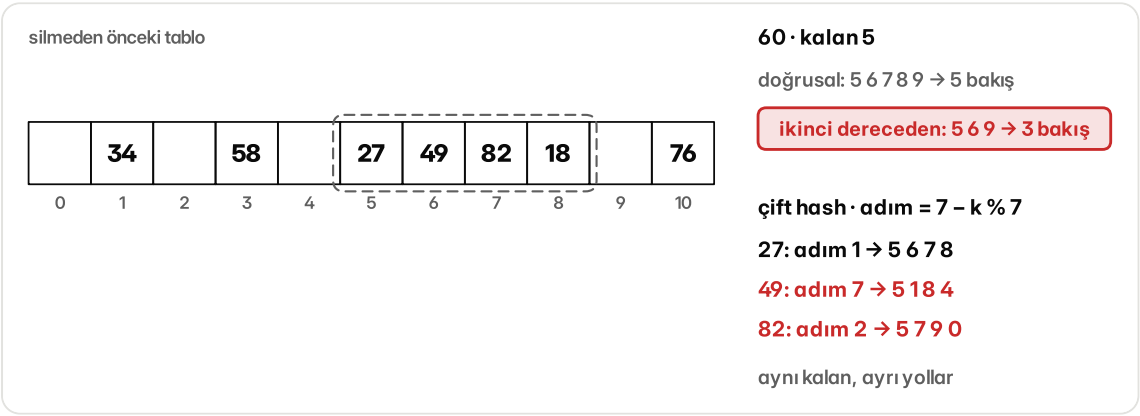
Çıktı:

```
 . 34 . 58 . 27 49 82 18 . 76
82 -> goz 7
 . 34 . 58 . 27 X 82 18 . 76
49 silindi; 82 -> goz 7
 . 34 . 58 . 27 60 82 18 . 76
60 -> goz 6
```



Şekil 3.8 49'un gözü silindi olarak işaretlenince bul(82) o gözün üstünden geçer ve 82'yi 7. gözde bulur.

Doğrusal yoklamada dolu gözler yan yana birikip **küme** olur. Silmeden önceki tabloda 5'ten 8'e dört gözlük küme, kalanı 5 ile 9 arasında olan her yeni anahtarı 9. göze yollar: bu göz 11 olası kalanın 5'ini yakalar, öteki boş gözler ikişer. Küme büyüdükçe daha çok anahtar yakalar ve daha da büyür. İkinci dereceden yoklamada 1, 4, 9 göz ileri sıçranır: kalanı 5 olan 60, doğrusal yoklamada 5 bakışta, ikinci dereceden yoklamada 5, 6, 9 diye 3 bakışta boş göz bulur. Çift hash'te sıçrama boyunu ikinci bir hash verir, örneğin 7 − k % 7; böylece aynı göze düşen 27, 49 ve 82 bile ayrı yollardan ilerler.



Şekil 3.9 60 doğrusal yoklamada 5, ikinci dereceden yoklamada 3 bakışta boş göz bulur; çift hash'te 27, 49 ve 82 ayrı yollardan ilerler.

Açık adreslemede tablo doldukça hızla yavaşlar. Doğrusal yoklamada olmayan bir anahtarı aramanın beklenen maliyeti yarı dolu tabloda ortalama 2,5 göz, %90 dolu tabloda yaklaşık 50 gözdür (100.003 gözlü bir tabloda ölçüm: 2,49 ve 45,55 göz). Bu yüzden tablo belli bir doluluğa varınca, Python'un sözlüğünde üçte ikiye, daha büyük bir tabloya taşınır. Bölen değiştiği için her anahtar yeni gözüne tek tek taşınır. Bu pahalı ama seyrek bir iştir: tablo her seferinde iki katına çıktığı için her eklemeye ikiden az taşıma düşer.

Hash tablosunun maliyeti:

İşlem	Ortalama	En kötü
ekle	O(1)	O(n)
bul	O(1)	O(n)
sil	O(1)	O(n)

Python'un sözlüğü açık adreslemeyle, C++'ın hash tablosu (unordered_map) zincirlemeyle çalışır. Hesaplanmış sonuçları saklayan önbellekler de çoğu zaman birer hash tablosudur. 1 milyon kayıtta tekrar edeni bulmak için her kaydı ötekilerle karşılaştırmak 499.999.500.000, yani yarım trilyon karşılaştırma ister; her kaydı tabloya koymadan önce gözüne bakmak ise 1 milyon adımda biter.

Özet

- Hash fonksiyonu büyük anahtarı tablo boyuna indirir; en basiti tablo boyuna bölümden kalandır.
- Metinde polinom hash harflerin yerini de hesaba katar; harf kodlarını toplamak aynı harfli isimleri aynı göze yollar.
- İki anahtarın aynı göze düşmesine çakışma denir; doğum günü problemindeki gibi tablo az doluyken bile kaçınılmazdır.
- Zincirlemede her göz bir bağlı listenin başıdır; ortalama liste boyu doluluk oranına eşittir.
- Açık adreslemede dolu gözün yanındakine geçilir; silinen göz boşaltılmaz, "silindi" diye işaretlenir.
- Ekleme, arama ve silme ortalamada O(1), en kötü durumda O(n) sürer; iyi bir hash ve zamanında büyütme ortalamayı korur.

Sorular

1 **11 gözlü bir hash tablosunda hash fonksiyonu k % 11. 467 anahtarı kaçıncı göze yazılır? Göz numarasını yaz.**

Cevap: _____

2 **İsmi sayıya çevirirken harf kodlarını yalnız toplamak neden yetmez?**

A Toplam, tablo boyundan her zaman büyük çıkar.

B Aynı harflerden oluşan isimlerin toplamı aynı olur ve hepsi aynı göze düşer.

C Harf kodları toplanınca sonuç eksi çıkabilir.

D Toplama, çarpmadan daha yavaş bir işlemdir.

3 **40 kişilik bir sınıfta kaç farklı kişi çifti vardır?**

A 780

B 40

C 1.600

D 365

4 **Zincirlemeli, 7 gözlü bir hash tablosunda hash x % 7 ve yeni gelen listenin başına ekleniyor. Boş tabloya sırayla 10, 24, 3 ve 17 eklenince 3. gözün listesini baştan sona yaz.**

Cevap:

5 **Zincirlemeli bir hash tablosunda 4 göze 12 anahtar yerleştirildi ve hash anahtarları gözlere eşit dağıtıyor. Bir listenin ortalama boyu nedir?**

A 12
B 4
C 3
D 1

6 **Anahtarlar sonu 0 ile biten fiyatlar (120, 250, 590 ...). Hash, tablo boyuna bölümden kalan. Aşağıdaki tablo boylarından hangisinde anahtarlar bütün gözlere dağılabilir?**

A 20
B 30
C 10
D 11

7 **Doğrusal yoklamalı, 11 gözlü (0-10) bir tabloda hash x % 11. Boş tabloya sırayla 21, 32 ve 43 eklenir. 43 kaçıncı göze yazılır?**

Cevap:

8 **Doğrusal yoklamalı bir tabloda bir anahtar silinirken gözü boş (BOS) yapılırsa ne olur?**

A Tablo bir göz küçülür ve öteki anahtarlar sola kayar.
B Hiçbir şey olmaz; arama her zaman bütün gözlere bakar.
C Arama o boş gözde durur ve ondan sonraki gözlere yoklamayla yerleşmiş anahtarları bulamayabilir.
D Silinen anahtar bir sonraki aramada yeniden ortaya çıkar.

VERİ YAPILARI

4 Ağaçlar ve ikili ağaç

Bu bölümde elemanları tek sıra değil, dallanarak duran yapıyı, ağacı göreceğiz: terimlerini, her düğümün en çok iki çocuğu olan ikili ağacı ve C'deki düğümünü, kaç düğümün kaç kata sığdığını. Sonra ağacın her düğümüne bir kez uğramanın yollarını, yani dolaşmayı ve bir işlemin ağaç olarak nasıl saklandığını inceleyeceğiz.

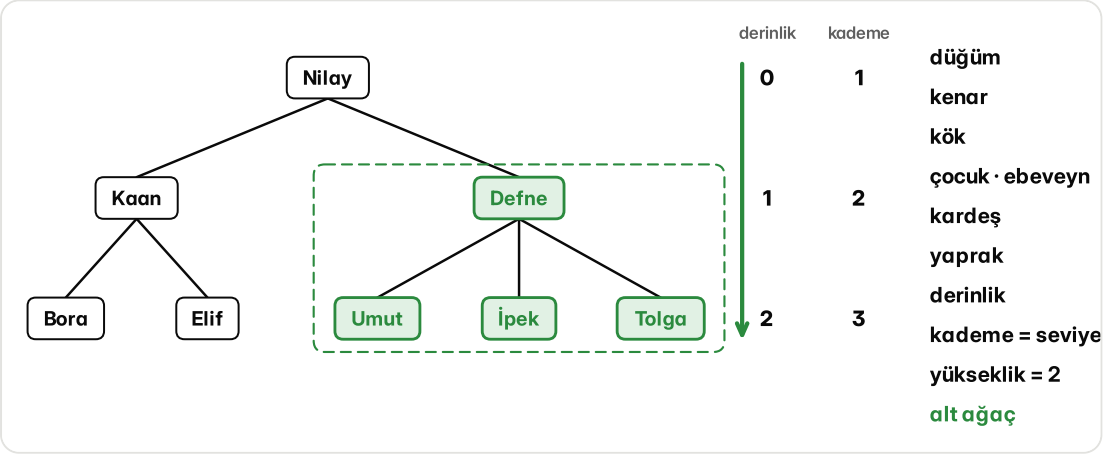
Kaç kademe?

videoda 26:00

100.000 kişinin çalıştığı bir şirkette her yönetici en çok 2 kişiyi yönetiyor. Genel müdür dahil, bu şirkette en az kaç kademe vardır?

Şirketin örgüt şeması tepedeki genel müdürden aşağı doğru dallanır: altında müdürler, onların altında ekipler. Kitapların bölümleri de böyle iç içedir, web sayfalarının parçaları da. Dizide ve listede elemanlar tek sıra durur; burada her eleman altındaki birkaç elemana bağlanır. Bu yapıya **ağaç** (tree) denir; yalnız kökü yukarıda çizilir.

Sekiz kişilik bir şirkette her çalışan bir **düğüm**dür; iki düğümü birleştiren çizgiye **kenar** denir. En tepedeki düğüm **kök**tür: genel müdür Nilay. Bir düğümün hemen altındakiler onun **çocuk**larıdır, kendisi de onların **ebeveyn**idir. Aynı ebeveynin çocukları **kardeş**tir: iki müdür Kaan ile Defne gibi. Çocuğu olmayan düğüm **yaprak**tır: kimseyi yönetmeyen Bora, Elif, Umut, İpek ve Tolga. Bir düğümün kökten kaç kenar aşağıda olduğu onun **derinliği**dir; aynı derinliktekiler bir **seviye**, şirket diliyle bir kademe oluşturur. En derindeki yaprağın derinliği ağacın **yüksekliği**dir: burada 2. Bir düğüm, altındaki her şeyle birlikte bir **alt ağaç**tır: Defne ve ekibi kendi başına küçük bir şemadır.



Şekil 4.1 Sekiz kişilik şemada Nilay kök, Kaan ile Defne kardeş, beş çalışan yapraktır; derinlik 0, 1, 2 ve ağacın yüksekliği 2'dir.

Ekipler aynı büyüklükte değildir: bir ekip 2 kişilik de olabilir, 20 kişilik de. Her düğümde 20 adreslik yer açmak israftır. Bunun yerine her düğüm 2 adres tutar: ilk çocuğunun ve sağındaki kardeşinin. Defne'nin bütün ekibine, Umut'tan başlayıp kardeşten kardeşe geçerek varılır: Umut, İpek, Tolga, sonra NULL.

Her düğümün en çok 2 çocuğu olan ağaca **ikili ağaç** (binary tree) denir; çocuklar sol ve sağ diye ayrılır: Kaan'ın ekibinde Bora solda, Elif sağda. C'de düğüm bir struct'tır: veri, sol çocuğun adresi ve sağ çocuğun adresi.

```
typedef struct dugum {
    int veri;
    struct dugum *sol;
    struct dugum *sag;
} Dugum;
```

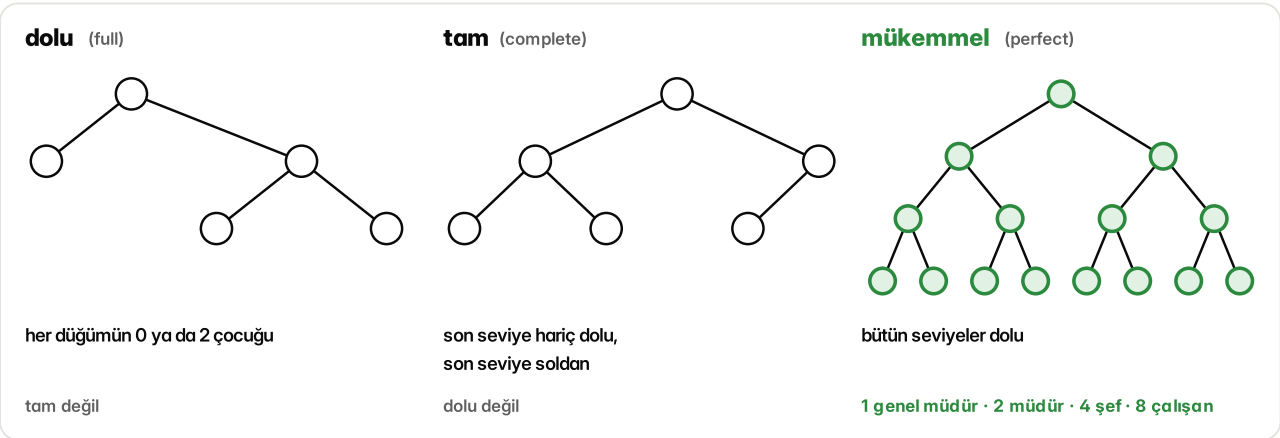
Yeni düğümün yerini malloc verir; bellek alınamazsa exit(1) programı durdurur. İki adres de NULL, yani hiçbir yeri göstermeyen adres yapılır, çünkü düğümün henüz çocuğu yoktur:

```
Dugum *yeni_dugum(int veri) {
    Dugum *d = malloc(sizeof(Dugum));
    if (d == NULL) exit(1);
    d->veri = veri;
    d->sol = d->sag = NULL;
    return d;
}
```

İkili ağacın üç özel biçimi vardır. Her düğümün ya 2 çocuğu varsa ya da hiç yoksa ağaç **dolu**dur (full). Son seviye dışındaki bütün seviyeler doluysa, son seviye de soldan başlayıp boşluk bırakmadan doluyorsa ağaç **tam**dır (complete). Bütün seviyeleri ağzına kadar doluysa ağaç **mükemmel**dir (perfect): 1 genel müdür, 2 müdür, 4 şef, 8 çalışan. Her mükemmel ağaç hem dolu hem tamdır; tersi doğru değildir. Bölümün programı üç örnek şemayı kurup sayar:

Çıktı:

```
mukemmel   15 kisi, 8 yaprak, yukseklik 3, dolu: evet  tam: evet
kademeler: 1 2 4 8
dolu        5 kisi, 3 yaprak, yukseklik 2, dolu: evet  tam: hayir
tam         6 kisi, 3 yaprak, yukseklik 2, dolu: hayir tam: evet
```



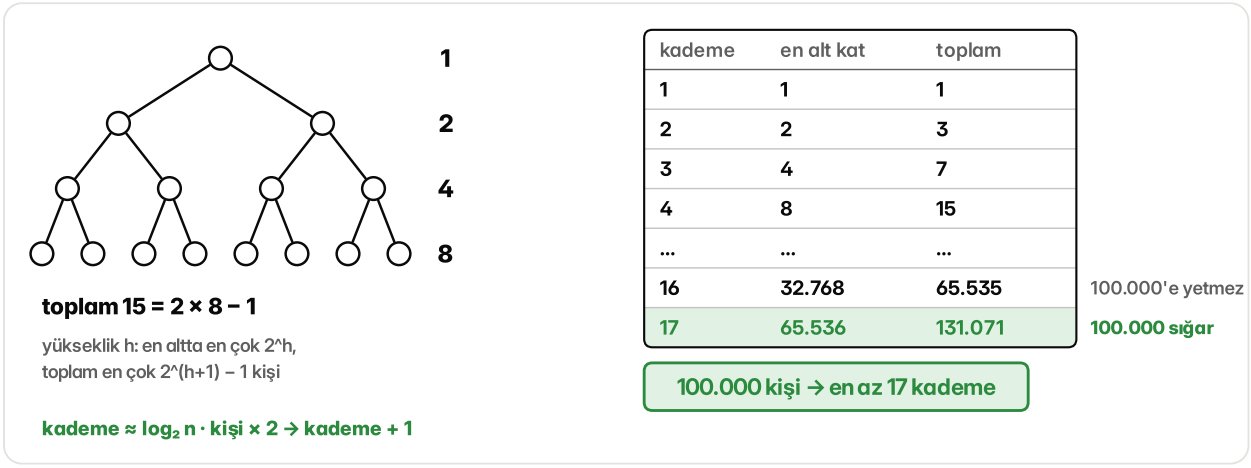
Şekil 4.2 Dolu ağaçta her düğümün 0 ya da 2 çocuğu, tam ağaçta soldan dolan son seviye, mükemmel ağaçta 1, 2, 4 ve 8 kişilik dolu seviyeler vardır.

Mükemmel şirkette kademeler sırayla 1, 2, 4 ve 8 kişi tutar. Her kademe bir öncekinin iki katıdır; toplam 15, yani en alttakinin iki katından bir eksiktir. Yüksekliği h olan ikili ağacın en alt seviyesinde en çok 2^h, bütününde en çok 2^(h+1) − 1 düğüm bulunur. 16 kademe en çok 65.535 kişiyi alır; 100.000 kişiye yetmez. 17. kademe 65.536 kişi daha ekler ve şirket 131.071 kişiye kadar büyüyebilir. Demek ki 100.000 kişi için en az 17 kademe gerekir.

Çıktı:

```
15 kisi: en az 4 kademe
16 kisi: en az 5 kademe
65535 kisi: en az 16 kademe
65536 kisi: en az 17 kademe
100000 kisi: en az 17 kademe
```

n kişi için gereken en az kademe sayısı tam olarak $\lceil \log_2(n + 1) \rceil$'dir, kabaca log n: kişi sayısı iki katına çıktıkça şirkete yalnız bir kademe eklenir.



Şekil 4.3 Kademeler 1, 2, 4, 8 diye ikiye katlanır; 16 kademe 65.535 kişide kalır, 17 kademe 131.071 kişiyi alır.

Hangi sırayla?

videoda 1:00:11

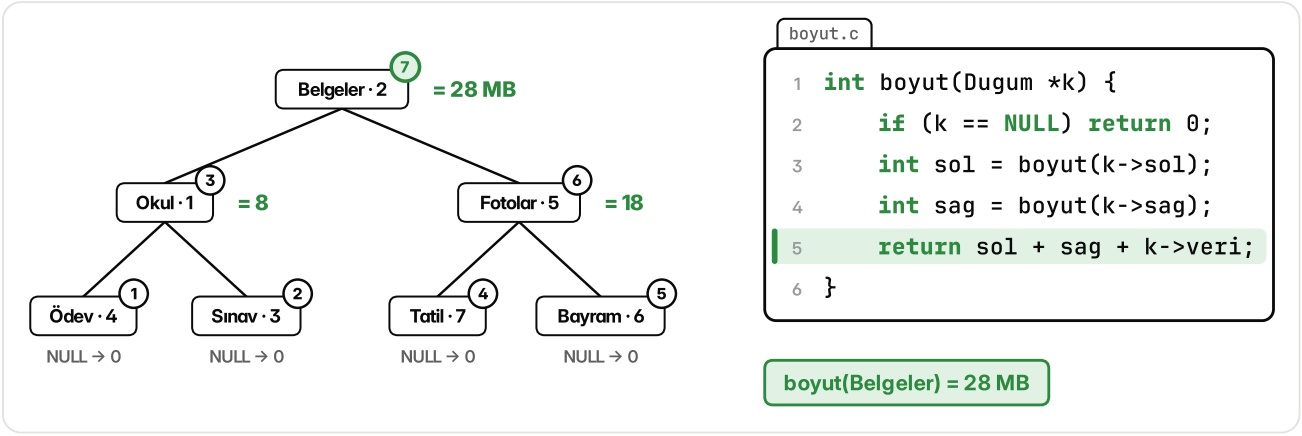
Bir klasörün içinde alt klasörler, onların içinde de başka klasörler var. Klasörün boyutunu bulmak için bilgisayar bu klasörleri hangi sırayla gezer?

Klasörler bir ağaç oluşturur: en üstte Belgeler, altında Okul ve Fotolar; Okul'un altında Ödev ile Sınav, Fotolar'ın altında Tatil ile Bayram. Her klasörün en çok iki alt klasörü olsun; o zaman bu bir ikili ağaçtır. Belgeler'in boyutu, kendi dosyalarıyla Okul ve Fotolar'ın boyutlarının toplamıdır. Ama Okul'un boyutu için de önce Ödev ile Sınav bilinmelidir. Sıra şudur: önce alt klasörler, en sonda klasörün kendisi.

Her alt klasör de kendi başına bir ağaçtır. Bu yüzden boyutu hesaplayan fonksiyon aynı soruyu daha küçük ağaca sorar, yani kendini çağırır (özyineleme). Adresi NULL olan ağaç boştur, boyutu 0'dır; özyineleme orada durur. Burada veri, klasörün kendi dosyalarının megabayt (MB) cinsinden boyutudur:

```
int boyut(Dugum *k) {
    if (k == NULL) return 0;
    int sol = boyut(k->sol);
    int sag = boyut(k->sag);
    return sol + sag + k->veri;
}
```

Fonksiyon önce solu, sonra sağı hesaplar, en sonda düğümün kendi dosyalarını ekler. Ödev 4, Sınav 3 MB; Okul kendi 1 MB'ıyla 8 olur. Tatil 7, Bayram 6, Fotolar kendi 5 MB'ıyla 18 olur. Belgeler 2 + 8 + 18 = 28 MB tutar.



Şekil 4.4 boyut önce alt klasörleri toplar: Okul 8, Fotolar 18 ve Belgeler 28 MB; halkalardaki 1-7 hesap sırasıdır.

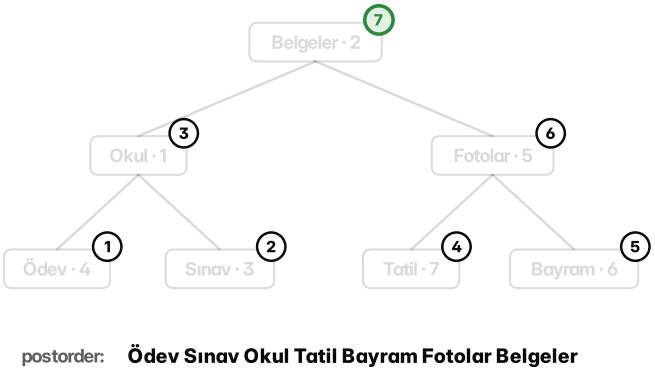
Her düğüme birer kez uğramaya **dolaşma** (traversal) denir. Her düğümde üç iş var: sol alt ağacı dolaşmak, sağ alt ağacı dolaşmak ve düğümün kendisini işlemek. Üç dolaşmanın tek farkı, düğümün kendisinin ne zaman işlendiğidir. Önce kendisi, sonra çocuklar: **preorder**. Klasör kopyalamak böyledir, çünkü klasör oluşturulmadan içi kopyalanamaz:

```
void preorder(Dugum *k) {
    if (k == NULL) return;
    yaz(k);
    preorder(k->sol);
    preorder(k->sag);
}
```

yaz(k) satırı iki çağrının arasına alınırsa sol, kendisi, sağ sırası çıkar: **inorder**. En sona alınırsa önce çocuklar, en sonda kendisi: **postorder**. Klasör boyutu ve klasör silmek böyledir; içi boşalmadan klasör silinemez. C'de ağacı bellekten silmek de postorder'dır: düğüm önce free ile geri verilirse çocuklarının adresi onunla kaybolur.

Çıktı:

```
boyut: 28 MB
hesap sirasi: Ödev 4, Sınav 3, Okul 8, Tatil 7, Bayram 6, Fotolar 18, Belgeler 28,
preorder : Belgeler Okul Ödev Sınav Fotolar Tatil Bayram
inorder  : Ödev Okul Sınav Belgeler Tatil Fotolar Bayram
postorder: Ödev Sınav Okul Tatil Bayram Fotolar Belgeler
seviye   : Belgeler Okul Fotolar Ödev Sınav Tatil Bayram
```



dolas.c

```c
void postorder(Dugum *k) {
    if (k == NULL) return;
    postorder(k->sol);
    postorder(k->sag);
    yaz(k);
}
```

boyut ve silme: önce içi

free: önce çocuklar, sonra düğüm

Şekil 4.5 Postorder sırası Ödev Sınav Okul Tatil Bayram Fotolar Belgeler'dir; boyut ve silme önce içi, sonra klasörü işler.

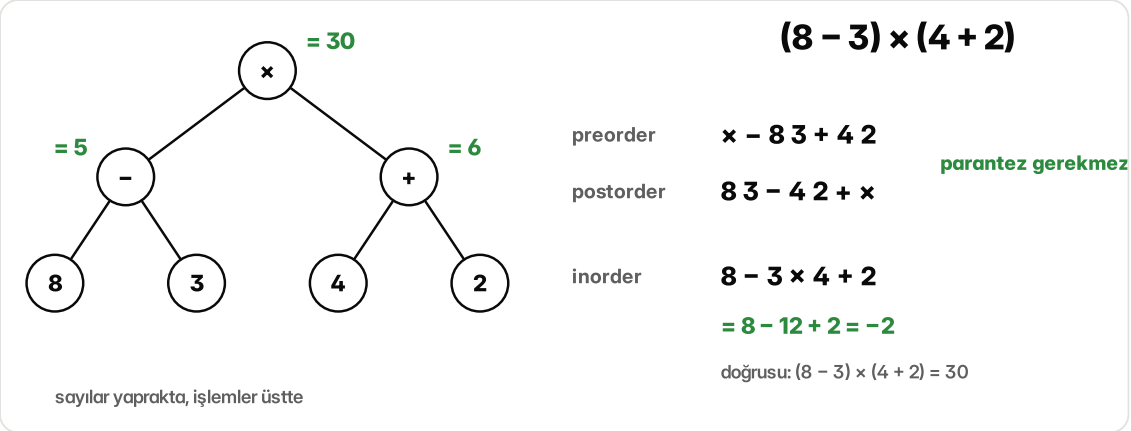
Ağaç kat kat da gezilebilir: önce Belgeler, sonra Okul ve Fotolar, sonra en alttakiler. Bunun için ilk gireni ilk çıkaran bir sıra, yani kuyruk kullanılır. Kök kuyruğa girer; her adımda baştaki çıkar, işlenir ve çocukları kuyruğun sonuna eklenir. Belgeler çıkar, Okul ile Fotolar girer; Okul çıkar, Ödev ile Sınav arkaya dizilir. Kuyruk geliş sırasını koruduğu için bir kat bitmeden alttakine geçilmez. Kodda kuyruk bir dizidir:

```c
void seviye(Dugum *kok) {
    Dugum *kuyruk[16] = {kok};
    int bas = 0, son = 1;
    while (bas < son) {
        Dugum *k = kuyruk[bas++];
        yaz(k);
        if (k->sol) kuyruk[son++] = k->sol;
        if (k->sag) kuyruk[son++] = k->sag;
    }
}
```

Bir işlem de ağaç olabilir: (8 − 3) × (4 + 2). Sayılar yapraklarda, işlemler üstteki düğümlerde durur: kökte ×, solunda − (8 ve 3), sağında + (4 ve 2). Ağacın değeri postorder ile bulunur: önce çocuklar, sonra işlem. Eksi düğümü 5, artı düğümü 6, kök 30 verir. Aynı ağacı preorder ile yazarsak her işlem işareti iki tarafından önce gelir, postorder ile sonra; ikisi de parantez olmadan doğru hesaplanır. Inorder alışık olduğumuz yazımı verir ama parantezleri kaybeder: çarpma önce yapılır ve 8 − 12 + 2 = −2 çıkar. Programda çarpma * ile yazılır:

Çıktı:

```
preorder : * - 8 3 + 4 2
inorder  : 8 - 3 * 4 + 2
postorder: 8 3 - 4 2 + *
seviye   : * - + 8 3 4 2
agacin sonucu: 30
eksi dugumu: 5, arti dugumu: 6
parantezsiz 8 - 3 * 4 + 2 = -2
```



Şekil 4.6 (8 − 3) × (4 + 2) ağacında eksi 5, artı 6, kök 30 verir; inorder yazım parantezi kaybedip −2 sonucunu verir.

Hangi sırayla olursa olsun her düğüme bir kez uğranır:

Dolaşma	Maliyet
preorder	O(n)
inorder	O(n)
postorder	O(n)
seviye seviye	O(n)

İşlem sırası ağacın şeklinde saklıdır; çoğu derleyici de koddaki işlemleri önce böyle bir ağaca çevirir.

Özet

- Ağaçta her düğüm altındaki birkaç düğüme bağlanır; en tepedeki kök, çocuğu olmayanlar yapraktır.
- Yükseklik en derindeki yaprağın kökten kaç kenar aşağıda olduğudur; ikili ağaçta her düğümün en çok iki çocuğu vardır.
- C'de ikili ağacın düğümü veri, sol ve sağ adresini tutan bir struct'tır; yeni düğümün iki adresi NULL'dır.
- Yüksekliği h olan ikili ağaç en çok 2^(h+1) – 1 düğüm alır; n düğüm için en az kabaca log n kat gerekir.
- Preorder önce düğümü, inorder solu, düğümü ve sağı, postorder önce çocukları işler; seviye seviye dolaşma kuyruk kullanır.
- Her dolaşma her düğüme bir kez uğrar ve O(n) sürer.

Sorular

1 **Bir ağaçta kök A'dır. A'nın çocukları B ve C, B'nin çocukları D ve E, E'nin tek çocuğu F'dir. Ağacın yüksekliği kaçtır?**

A 2
B 3
C 4
D 6

2 **Bir ikili ağaçta her düğümün ya 2 çocuğu var ya da hiç çocuğu yok, ama yapraklar farklı derinliklerde. Bu ağaç için hangisi kesinlikle doğrudur?**

A Ağaç mükemmeldir.
B Ağaç tamdır ama dolu değildir.
C Ağaç doludur ama mükemmel değildir.
D Ağaç ikili ağaç değildir.

3 **Yüksekliği 4 olan bir ikili ağaçta en çok kaç düğüm bulunabilir?**

Cevap:

4 **1.000 kişilik bir şirkette her yönetici en çok 2 kişiyi yönetiyor. Genel müdür dahil en az kaç kademe gerekir?**

A 9
B 11
C 500
D 10

5 **Bir ikili ağaçta kök K'dir. K'nin sol çocuğu A, sağ çocuğu B'dir. A'nın sol çocuğu C, sağ çocuğu D'dir. B'nin yalnız sağ çocuğu vardır: E. Postorder dolaşmanın çıktısını harfler arasında boşluk bırakarak yaz.**

Cevap: _____

6 **C'de bir ikili ağacı free ile bellekten silerken hangi dolaşma sırası kullanılmalıdır?**

A Postorder
B Preorder
C Inorder
D Seviye seviye

7 **Bir ağacı kat kat (seviye seviye) dolaşmak için hangi yapı kullanılır?**

A Yığın: son giren ilk çıkar.
B Kuyruk: ilk giren ilk çıkar.
C Hash tablosu.
D Sıralı dizi.

8 **Bir ifade ağacının kökünde × var. Sol çocuğu + düğümüdür ve onun çocukları 6 ile 2'dir. Kökün sağ çocuğu 3 yaprağıdır. Ağacın değeri kaçtır?**

Cevap: _____

VERİ YAPILARI

5 İkili arama ağacı

Bu bölümde hem hızlı bulan hem de kaydırmadan ekleyen bir yapıyı, ikili arama ağacını göreceğiz: kuralını, aramayı ve eklemeyi, ekleme sırasının ağacın şeklini nasıl belirlediğini. Sonra en küçük ve en büyük sayıyı bulmayı, sayıları sıralamadan sıralı yazmayı ve üç durumuyla silmeyi inceleyeceğiz.

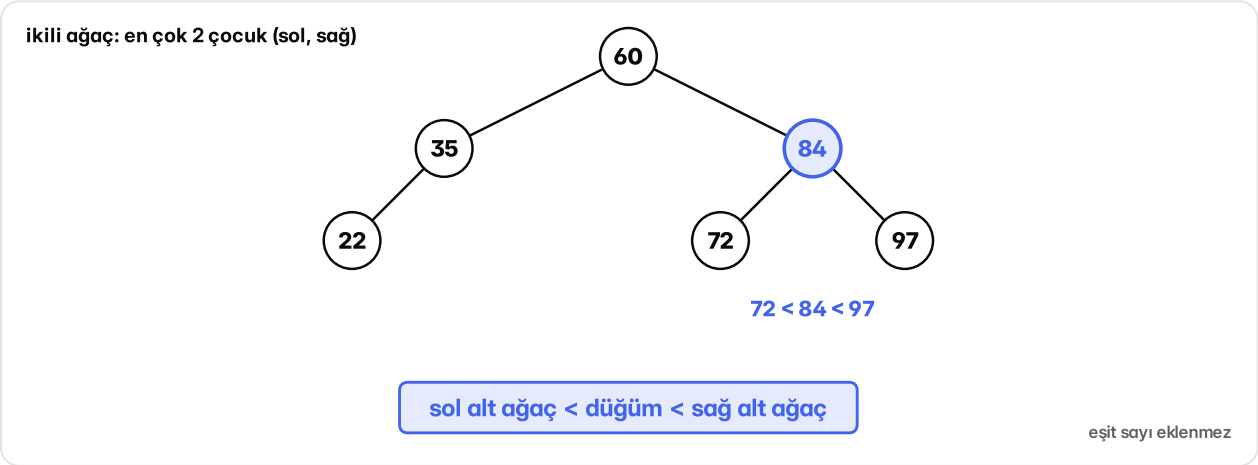
Aynı yedi sayı

videoda 31:00

Aynı 7 sayıyı bir ikili arama ağacına iki ayrı sırayla ekliyoruz. Birinde her sayı en çok 3 adımda bulunuyor, ötekinde en büyüğüne 7 adımda varılıyor. Değişen yalnız sıra; fark nereden geliyor?

Sıralı dizide arama hızlıdır: her adımda ortadakine bakılır, kalanın yarısı bırakılır. Ama araya sayı eklemek pahalıdır. 22, 35, 60, 72, 84 ve 97 sıralı duruyor; 43'ü eklemek için ondan büyük 4 sayı birer göz sağa kayar. 10.000 sayılık bir dizide en başa eklemek 10.000 kaydırma demektir. Hem hızlı bulan hem kaydırmadan ekleyen bir yapı gerekiyor.

Aynı altı sayıyı bir ağaca dizelim. İkili ağaçta her düğümün en çok iki çocuğu vardır, sol ve sağ. **İkili arama ağacı** (binary search tree) buna tek bir kural ekler: bir düğümün sol alt ağacındaki her sayı ondan küçük, sağ alt ağacındaki her sayı ondan büyüktür. Kökte 60 var: solunda 35 ve 22, sağında 84, 72 ve 97. Kural her düğümde ayrı ayrı geçerlidir: 84'ün solunda 72, sağında 97. Eşit sayı ağaca ikinci kez eklenmez.

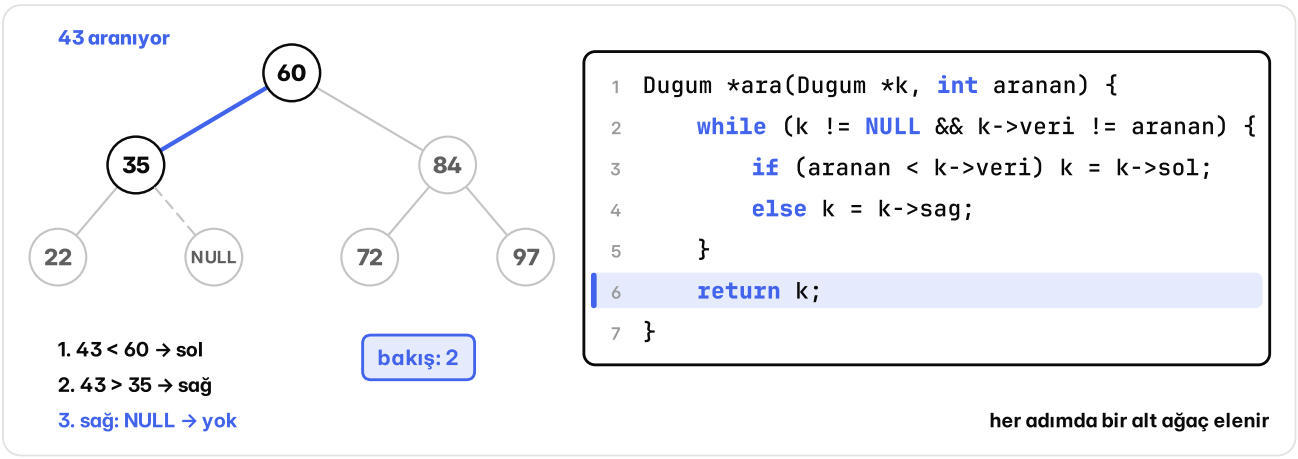


Şekil 5.1 Kökteki 60'ın solunda 35 ve 22, sağında 84, 72 ve 97 durur; kural her düğümde geçerlidir: 72 < 84 < 97.

72'yi arıyoruz. Kökteki 60'tan büyük; öyleyse sol tarafa bakmaya gerek yok, oradakilerin hepsi tek bakışta elenir. Sağa ineriz; 72, 84'ten küçük, sola. Üçüncü bakışta bulunur. Her adımda bir alt ağaç bırakılır, tıpkı sıralı dizide yarısını bırakmak gibi:

```
Dugum *ara(Dugum *k, int aranan) {
    while (k != NULL && k->veri != aranan) {
        if (aranan < k->veri) k = k->sol;
        else k = k->sag;
    }
    return k;
}
```

43'ü ararsak 60'tan küçük, sola; 35'ten büyük, sağa. Ama 35'in sağında kimse yok, adresi NULL: 43 ağaçta yok ve ara NULL döndürür.



Şekil 5.2 43 aranırken 60'tan sola, 35'ten sağa inilir; 35'in sağı NULL olduğu için iki bakışta "yok" denir.

43'ü eklemek için aynı yol yürünür ve aramanın bittiği boş yere yeni düğüm takılır: 43, 35'in sağ çocuğu olur. Hiçbir sayı yerinden oynamaz, yalnız bir adres değişir. Ekleme fonksiyonu kendini çağırır: boş bir yere geldiyse yeni düğümü kurar; değilse sayı küçükse sola, büyükse sağa iner ve inilen dalın adresini yeniden yazar:

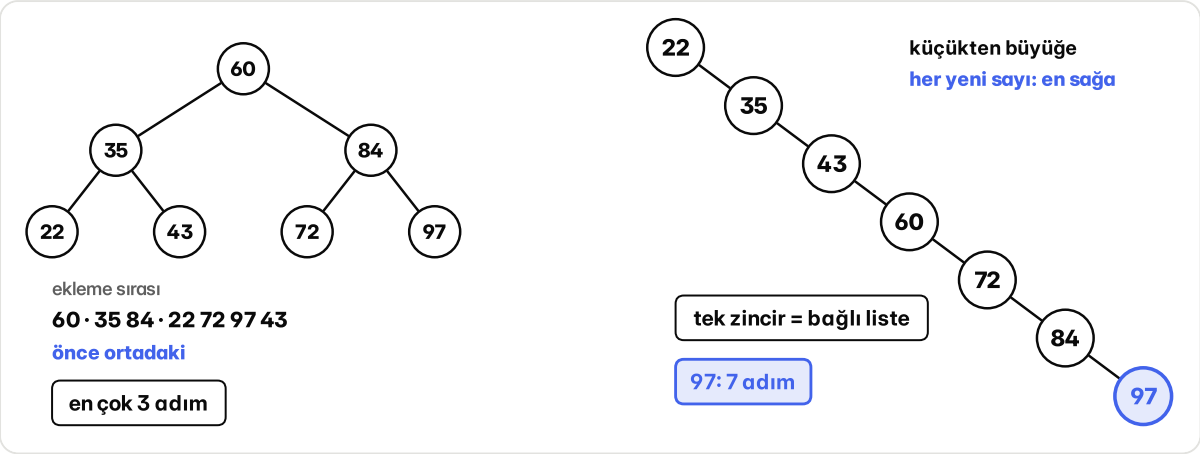
```
Dugum *ekle(Dugum *k, int veri) {
    if (k == NULL) return yeni_dugum(veri);
    if (veri < k->veri)
        k->sol = ekle(k->sol, veri);
    else if (veri > k->veri)
        k->sag = ekle(k->sag, veri);
    return k;
}
```

Program ağacı katlarıyla, aramaları da yollarıyla yazar:

Çıktı:

```
ekleme sirasi 60 35 84 22 72 97: 60 | 35 84 | 22 72 97
72 ara: 60 sag 84 sol 72 -> bulundu (3 bakis)
43 ara: 60 sol 35 sag -> NULL, yok (2 bakis)
43 eklendi: 35'in sag cocugu mu? evet; katlar: 60 | 35 84 | 22 43 72 97
```

Bu ağaçta her sayı en çok 3 adımda bulunur, çünkü önce ortadaki sayı geldi: 60; sonra iki yanın ortadakileri, 35 ve 84. Aynı 7 sayı küçükten büyüğe gelseydi 22 kök olurdu. 35 ondan büyük, sağına gider; 43 ikisinden de büyük, onların sağına. Her yeni sayı öncekilerin hepsinden büyük olduğu için hep en sağa eklenir. Ağaç sağa uzanan tek bir zincire, yani bağlı listeye döner: 97'yi bulmak 7 adım sürer.



Şekil 5.3 Ortadaki sayı önce gelince ağaç 3 katlı kalır; küçükten büyüğe eklenen aynı 7 sayı bağlı listeye döner ve 97'ye 7 adımda varılır.

Arama da ekleme de kökten aşağı tek bir yol yürür; adım sayısı en çok ağacın kat sayısı kadardır. Dengeli, yani katları dolu bir ağaçta her kat bir öncekinin iki katı düğüm alır: 1 + 2 + 4 + ... + 8.192 = 16.383, bu yüzden 10.000 sayı 14 kata sığar ve arama en çok 14 adım sürer. Küçükten büyüğe eklenen 10.000 sayı ise 10.000 kat olur.

Çıktı:

```
10000 dengeli: 14 kat, en cok 14 bakis
10000 kucukten buyuge: en buyuk icin 10000 bakis, 10000 kat
10000 sayilik diziye en basa ekleme: 10000 kayma
```

Maliyet dengeli ağaçta O(log n), zincirde O(n). Sipariş numaraları ve tarihler gibi hep artarak gelen veride sıradan bir ikili arama ağacı bu yüzden listeye döner.

Kökün yerine kim?

videoda 33:30

Bir ikili arama ağacının kökünü sileceğiz. Kökün hem solunda hem sağında dal var. Kuralı bozmadan köke hangi sayı geçebilir?

8 sayı sıralı bir dizide değil, bir ağaçta duruyor; kökte 51. 51'in solunda 32, 19 ve 40; sağında 81, 66, 75 ve 94. 32'nin çocukları 19 ve 40, 81'in çocukları 66 ve 94; 75, 66'nın sağ çocuğu. Köke kim geçerse geçsin, sol alt ağaçtakilerin düğümden küçük, sağ alt ağaçtakilerin büyük olması kuralı bütün düğümlerde korunmalıdır.

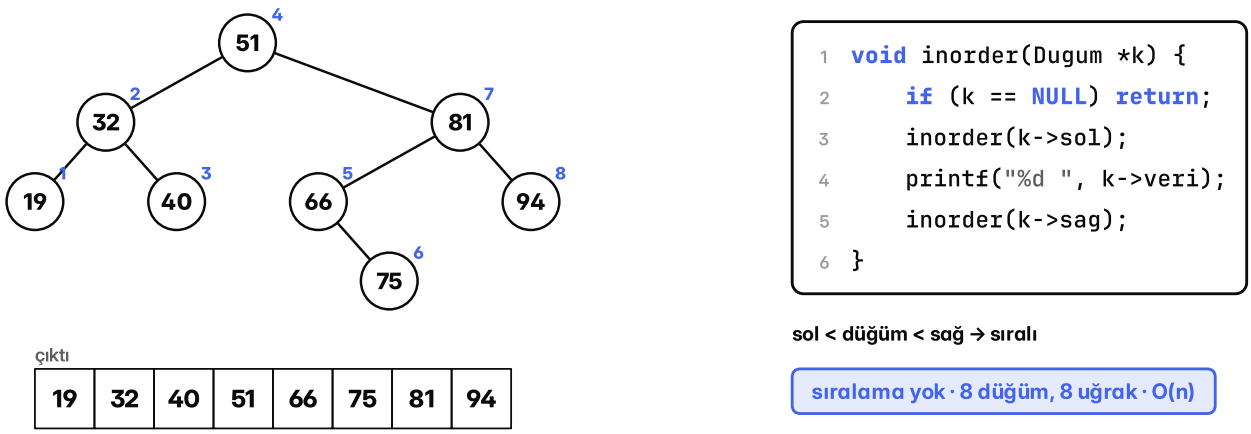
Önce iki kolay soru: en küçük ve en büyük sayı nerede? Küçükler hep solda olduğu için kökten sola inilir, solda kimse kalmayınca durulur: 51, 32, 19. En büyük için hep sağa: 81, 94. Adım sayısı ağacın kat sayısını, burada 4'ü geçmez. En büyüğü bulan fonksiyonda sol yerine sag yazılır:

```
Dugum *en_kucuk(Dugum *k) {
    while (k->sol != NULL) k = k->sol;
    return k;
}
```

Bu sayıları küçükten büyüğe yazmak için sıralama gerekmez. Inorder dolaşma her düğümde önce sol alt ağacı, sonra düğümün kendisini, en son sağ alt ağacı yazar:

```
void inorder(Dugum *k) {
    if (k == NULL) return;
    inorder(k->sol);
    printf("%d ", k->veri);
    inorder(k->sag);
}
```

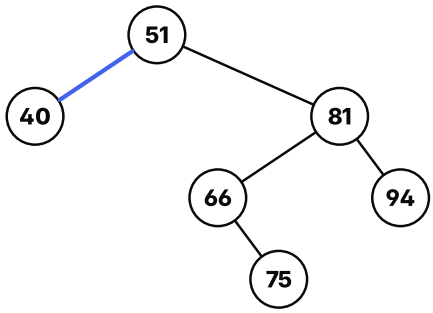
Soldakilerin hepsi düğümden küçük, sağdakilerin hepsi büyük olduğu için her düğüm tam yerinde yazılır: 19 32 40 51 66 75 81 94. 8 düğümün her birine bir kez uğranır: O(n).



Şekil 5.4 Inorder dolaşma düğümleri 1'den 8'e numaralar ve sıralama yapmadan 19 32 40 51 66 75 81 94 yazar.

Silmenin iki kolay durumu var. Çocuğu olmayan düğüm, yani yaprak, doğrudan sökülür: 19 silinince 32'nin sol adresi NULL olur, düğüm de free ile belleğe geri verilir. Artık 32'nin tek çocuğu var, 40. Tek çocuklu düğüm silinince çocuğu onun yerine geçer: 51, sol adresini doğrudan 40'a bağlar. Kural bozulmaz, çünkü 40 zaten 51'in solundaydı. Kodda iki durum aynı satırdır: düğümün yerine varsa çocuğu, yoksa NULL geçer. Aşağıdaki fonksiyon silinecek düğümü alır ve yerine geçeni döndürür; düğümü bulup bu fonksiyonu çağıran sil, ekle gibi yürür:

```
Dugum *sil_dugum(Dugum *k) {
    if (k->sol != NULL && k->sag != NULL) {
        Dugum *ardil = en_kucuk(k->sag);
        k->veri = ardil->veri;
        k->sag = sil(k->sag, ardil->veri);
        return k;
    }
    Dugum *cocuk = k->sol ? k->sol : k->sag;
    free(k);
    return cocuk;
}
```



40 < 51: kural sağlam

yaprak → NULL · tek çocuk → o çocuk

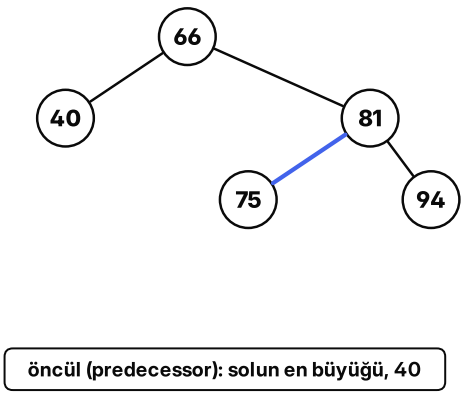
```
1  Dugum *sil_dugum(Dugum *k) {
2      if (k->sol != NULL && k->sag != NULL) {
3          Dugum *ardil = en_kucuk(k->sag);
4          k->veri = ardil->veri;
5          k->sag = sil(k->sag, ardil->veri);
6          return k;
7      }
8      Dugum *cocuk = k->sol ? k->sol : k->sag;
9      free(k);
10     return cocuk;
11 }
```

Şekil 5.5 19 yaprak olarak sökülür, tek çocuklu 32'nin yerine 40 geçer; 40 < 51 olduğu için kural sağlamdır.

Şimdi kökün iki çocuğu var. Yerine geçecek sayı soldakilerin hepsinden büyük, sağda kalanların hepsinden küçük olmalı. Inorder sırada 51'in hemen arkasından gelen sayı tam böyledir; buna **ardıl** (successor) denir. Ardıl, sağ alt ağacın en küçüğüdür: bir kez sağa, sonra hep sola; 81'den 66'ya. Fonksiyonun üst bloğu bunu yapar: 66 köke yazılır, sonra eski yerinden silinir. Ardılın sol çocuğu olamaz, olsaydı en küçük o olurdu; bu yüzden ikinci silme hep kolay durumlardan biridir: 66'nın yerine tek çocuğu 75 geçer. Solun en büyüğü olan 40, yani öncül (predecessor) de köke geçebilirdi.

Çıktı:

```
19 silindi (yaprak): 32 40 51 66 75 81 94 [7 sayi, 4 kat]
32(sol NULL, sag 40)
32 silindi (tek cocuk): 40 51 66 75 81 94 [6 sayi, 4 kat]
51(sol 40, sag 81)
51'in ardili (sag alt agacin en kucugu): 66, yol: 51 81 66 | oncul (solun en buyugu): 40
51 silindi (iki cocuk): 40 66 75 81 94 [5 sayi, 3 kat]
66(sol 40, sag 81) 81(sol 75, sag 94)
```




```c
Dugum *sil_dugum(Dugum *k) {
    if (k->sol != NULL && k->sag != NULL) {
        Dugum *ardil = en_kucuk(k->sag);
        k->veri = ardil->veri;
        k->sag = sil(k->sag, ardil->veri);
        return k;
    }
    Dugum *cocuk = k->sol ? k->sol : k->sag;
    free(k);
    return cocuk;
}
```

Şekil 5.6 51 silinince ardılı 66 köke geçer; son ağaçta kökte 66, solunda 40, sağında 81 ve onun çocukları 75 ile 94 vardır.

Silmek de aramak gibi kökten aşağı tek bir yol yürür: düğümü bulmak, gerekirse ardılı bulmak, birkaç adresi değiştirmek. Adım sayısı kat sayısı kadardır: katlar doluysa 30.000 sayıda en çok 15 adım, yani O(log n); ağaç tek bir zincire dönmüşse 30.000 adım, O(n). Aşağıdaki tabloda h ağacın yüksekliğidir:

İşlem	Sıralı dizi	İkili arama ağacı
ara	O(log n)	O(h)
ekle	O(n)	O(h)
sil	O(n)	O(h)
en küçük	O(1)	O(h)
sıralı yaz	O(n)	O(n)

Aynı sayılar sıralı bir dizide dursaydı, 51'i silmek arkasındaki 4 sayıyı birer göz geri çekmek demekti; ağaçta yalnız 66 yer değiştirdi.

Özet

- İkili arama ağacında her düğümün sol alt ağacındaki sayılar ondan küçük, sağ alt ağacındakiler büyüktür.
- Arama ve ekleme kökten aşağı tek bir yol yürür; ekleme aramanın bittiği boş yere yeni düğüm takar, hiçbir sayıyı kaydırmaz.
- Ağacın şekli ekleme sırasına bağlıdır; küçükten büyüğe gelen sayılar ağacı bağlı listeye çevirir.
- Arama, ekleme ve silme O(h) sürer: katlar doluysa O(log n), zincirde O(n).
- En küçük sayı hep sola, en büyük hep sağa inilerek bulunur; inorder dolaşma sayıları sıralı yazar.
- Yaprak doğrudan sökülür, tek çocuklu düğümün yerine çocuğu geçer; iki çocuklu düğümün yerine ardılı, yani sağ alt ağacın en küçüğü geçer.

Sorular

1 **Aşağıdaki ağaçlardan hangisi ikili arama ağacının kuralını bozar?**

A Kök 50; sol çocuğu 30, sağ çocuğu 70.

B Kök 50; sol çocuğu 30, 30'un sağ çocuğu 40.

C Kök 50; sağ çocuğu 70, 70'in sol çocuğu 60.

D Kök 50; sol çocuğu 30, 30'un sağ çocuğu 60.

2 **Boş bir ikili arama ağacına sırayla 50, 30, 70, 20, 40, 60, 80 ekleniyor. 60 aranırken kaç düğüme bakılır?**

Cevap:

3 **Boş bir ikili arama ağacına sırayla 40, 20, 60, 10, 30 ekleniyor. Inorder dolaşma ne yazar? Sayıları aralarında boşluk bırakarak yaz.**

Cevap:

4 **Aynı beş sayı boş bir ikili arama ağacına aşağıdaki sıralarla ekleniyor. Hangi sıra ağacı tek bir zincire, yani bağlı listeye çevirir?**

A 30 10 50 20 40

B 10 20 30 40 50

C 30 20 40 10 50

D 30 50 10 40 20

5 **Kendini dengelemeyen sıradan bir ikili arama ağacında n sayı varken bir sayıyı aramanın en kötü durumdaki maliyeti nedir?**

A O(1)

B O(log n)

C O(n)

D O(n log n)

6 **İkili arama ağacında tek çocuklu bir düğüm silinirken ne yapılır?**

A Düğümün çocuğu, düğümün yerine geçer; ebeveyn adresini doğrudan çocuğa bağlar.

B Düğümün yerine ağacın en küçük sayısı yazılır.

C Düğümün çocuğu da onunla birlikte silinir.

D Düğümden sonraki bütün sayılar birer adım yukarı kaydırılır.

7 **Boş bir ikili arama ağacına sırayla 45, 25, 65, 15, 35, 55, 75, 30 ekleniyor. Sonra 25 siliniyor ve yerine ardılı geçiyor. 25'in yerine hangi sayı geçer?**

Cevap: _____

8 **İkili arama ağacında en küçük sayı nasıl bulunur?**

A Bütün düğümler dolaşılıp karşılaştırılır.

B Kökten başlayıp hep sağa inilir.

C Kökün sol çocuğuna bakılır.

D Kökten başlayıp solda kimse kalmayana kadar hep sola inilir.

VERİ YAPILARI

6 Dengeli ağaçlar

Bu bölümde ekleme sırası ne olursa olsun yüksekliğini log n düzeyinde tutan ağaçları göreceğiz. AVL ağacının kuralını, tek ve çift dönmeyi; daha gevşek dengeli kırmızı-siyah ağacı ve neden tercih edildiğini; son olarak diskteki büyük veri için geliştirilen B-ağacını inceleyeceğiz.

Sırayı bozmadan

videoda 36:02

Bir uygulamada üye numaraları hep artarak gelir: 1, 2, 3, 4. Böyle sırayla gelen sayılar ikili arama ağacını tek kollu bir listeye çevirir. Ağaç, sayıların sırasını bozmadan kendini nasıl yeniden dizer?

Arama ağacın yüksekliği kadar sürer; amaç, yüksekliği log n düzeyinde tutmak. Bunu sağlayan ilk ağacı 1962'de iki Sovyet matematikçisi, Adelson-Velski ve Landis tanımladı; adı soyadlarının baş harflerinden gelir: **AVL ağacı**. Kuralı tek cümledir: her düğümde sol ve sağ alt ağacın yükseklikleri en çok 1 farklı olabilir. Bunun için her düğüm kendi yüksekliğini de saklar:

```
typedef struct dugum {
    int veri;
    int yukseklik;
    struct dugum *sol;
    struct dugum *sag;
} Dugum;
```

Boş dalın yüksekliği −1, yaprağınki 0 sayılır. fark, sol yükseklikten sağ yüksekliği çıkarır:

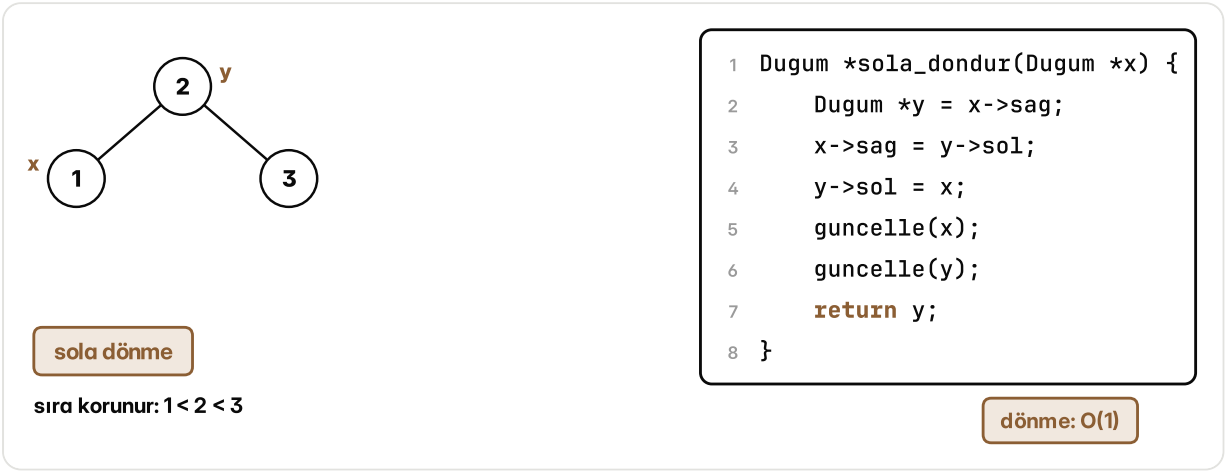
```
int yuk(Dugum *d) {
    return d ? d->yukseklik : -1;
}

int fark(Dugum *d) {
    return yuk(d->sol) - yuk(d->sag);
}
```

Bir eklemeden sonra fark 2'ye (ya da −2'ye) çıkarsa ağaç o düğümde dengesini kaybetmiştir.

1 ve 2 eklenir. 3 gelince 1'in solu boştur, sağında iki katlık bir kol vardır: fark −1 − 1 = −2. Düzeltme şöyle: ortadaki 2 yukarı çıkar, 1 onun soluna iner, 3 sağında kalır. Buna **sola dönme** denir. Sıra bozulmaz: 1 yine 2'nin solunda, 3 sağında. C'de bu iş 3 adres değişikliğidir; guncelle, düğümün yüksekliğini çocuklarından yeniden hesaplar:

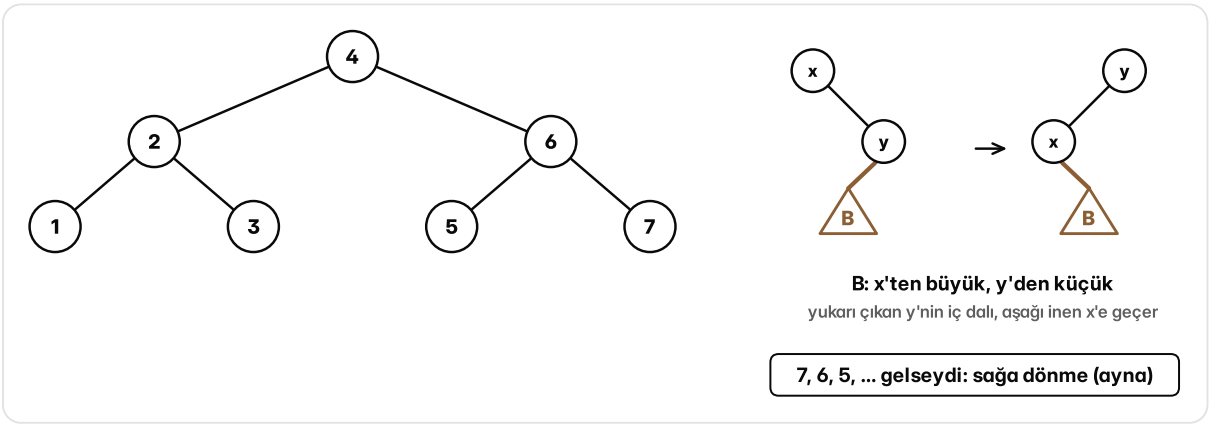
```
Dugum *sola_dondur(Dugum *x) {
    Dugum *y = x->sag;
    x->sag = y->sol;
    y->sol = x;
    guncelle(x);
    guncelle(y);
    return y;
}
```

Kodda iki atama görünür; üçüncü değişiklik üstteki düğümün adresidir: fonksiyon y'yi döndürür, çağıran onu x'in eski yerine yazar. Ağaç ne kadar büyük olursa olsun dönme aynı iştir, yani O(1).



Şekil 6.1 1, 2, 3 sağa doğru bir kol oluşturur; sola dönmede 2 yukarı çıkar, 1 soluna iner ve sıra korunur.

4 gelir, denge bozulmaz. 5'te 3 dengesini kaybeder; 4 yukarı çıkar. 6 gelince denge kökte bozulur: 4 köke yükselir, 2 soluna iner. Peki 4'ün solundaki 3 ne olur? 2'den büyük, 4'ten küçük; bu yüzden 2'nin sağına geçer. Dönmenin tek inceliği budur: yukarı çıkan düğümün iç taraftaki dalı, aşağı inen düğüme geçer. 7'de 5'te bir sola dönme daha yapılır. Sayılar büyükten küçüğe gelseydi aynı işin aynası yapılırdı: **sağa dönme**.



Şekil 6.2 6 eklenince 4 köke çıkar, 2 soluna iner; 4'ün iç dalındaki 3, 2'nin sağına geçer.

Her eklemede yöntem aynıdır: yeni üye her zamanki gibi yerine eklenir, sonra eklendiği yerden köke doğru geri çıkılır. Özyinelemeli ekle fonksiyonunda bu geri çıkış kendiliğinden olur; her düğüm dönüşte dengele'den geçer:

```
Dugum *ekle(Dugum *d, int x) {
    if (d == NULL)
        return yeni_dugum(x);
    if (x < d->veri)
        d->sol = ekle(d->sol, x);
    else
        d->sag = ekle(d->sag, x);
    return dengele(d);
}
```

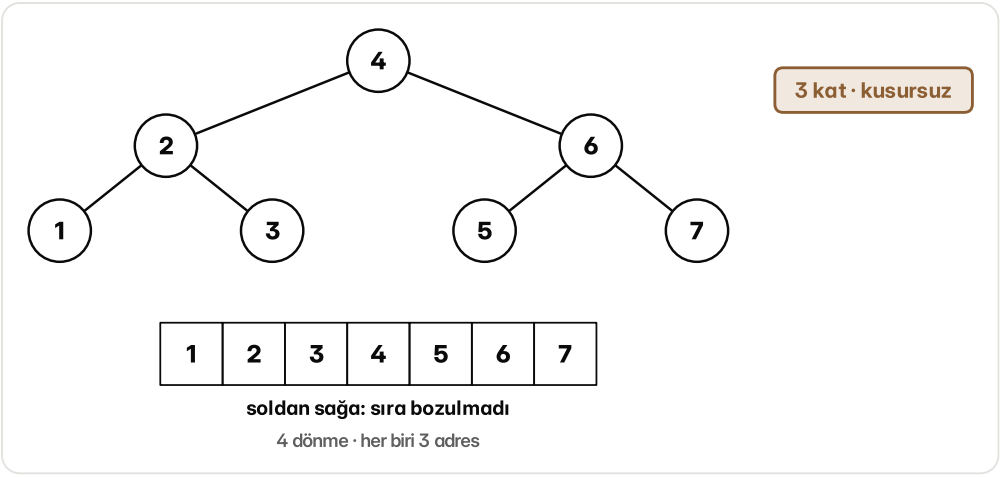
Yol üstündeki her düğüm yüksekliğini günceller; fark ilk kez 2'ye çıktığı yerde dönme yapılır. Dönme o kısmın yüksekliğini eklemeden önceki hâline döndürdüğü için yukarısı etkilenmez; bir ekleme en çok bir düzeltme ister. Programın ilk adımlardaki çıktısında 2(1,3), kökü 2, solu 1, sağı 3 olan ağaç demektir; tire boş dalı gösterir:

Çıktı:

```
3 eklendi
   1'de fark -2: sola donme (2 yukari)
   agac: 2(1,3)   (kat 2)
4 eklendi
   agac: 2(1,3(-,4))   (kat 3)
5 eklendi
   3'de fark -2: sola donme (4 yukari)
   agac: 2(1,4(3,5))   (kat 3)
```

AVL ağacı	Maliyet
arama	O(log n)
ekleme	O(log n)
dönme	O(1)

7 üye sırayla geldi ama ağaç 3 katlı ve kusursuz: kökte 4, altında 2 ile 6, en altta 1, 3, 5 ve 7. Sayıların sırası hiç bozulmadı; ağaç 4 dönmede, yalnız birkaç adresi değiştirerek kendini yeniden dizdi.



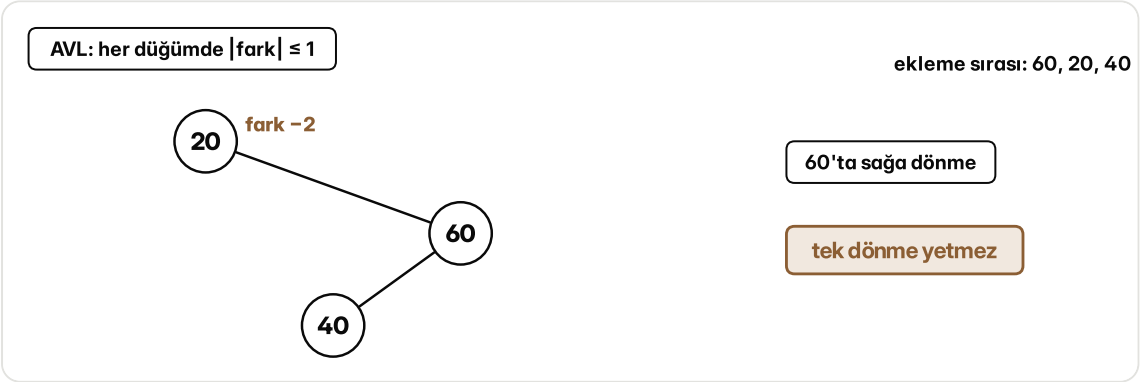
Şekil 6.3 Sırayla gelen 7 üye 3 katlı ağaçta durur: kökte 4, altında 2 ve 6, en altta 1, 3, 5, 7; soldan sağa sıra bozulmamıştır.

Neden AVL değil?

videoda 38:07

C++'ın sıralı sözlüğü de Java'nınki de anahtarları kendini dengeleyen bir ağaçta tutar. Ama daha sıkı dengeli AVL ağacını değil, daha gevşek bir ağacı seçerler. Neden?

AVL ağacında sol ve sağ yükseklik en çok 1 farklıdır; fark 2'ye çıkınca bir dönme ortadaki düğümü yukarı alır. Şimdi 60, 20 ve 40 sırayla eklensin. 20, 60'ın soluna, 40 da 20'nin sağına iner: kol bir sola, bir sağa kırılır, zikzak olur. 60'ta fark 2. Sağa döndürülünce 20 yukarı çıkar, ama 40, 60'ın altına geçer ve ağaç bu kez öbür yana yatar. Tek dönme yetmez.



Şekil 6.4 60, 20, 40 zikzak oluşturur; 60'ta sağa dönme 20'yi yukarı alır ama ağaç öbür yana yatar ve fark −2 olur.

Önce alttaki kırık kol düzeltilir: 20'de sola dönülür, 40 yukarı çıkar ve 3 sayı tek çizgiye dizilir. Sonra 60'ta sağa dönülür: ortanca 40 tepeye çıkar, 20 solunda, 60 sağında. Bu iki adımlı işe **çift dönme** denir. C'deki dengeleme fonksiyonu iki şey sorar: ağaç hangi yana ağır, ağır kol içe mi kırılmış?

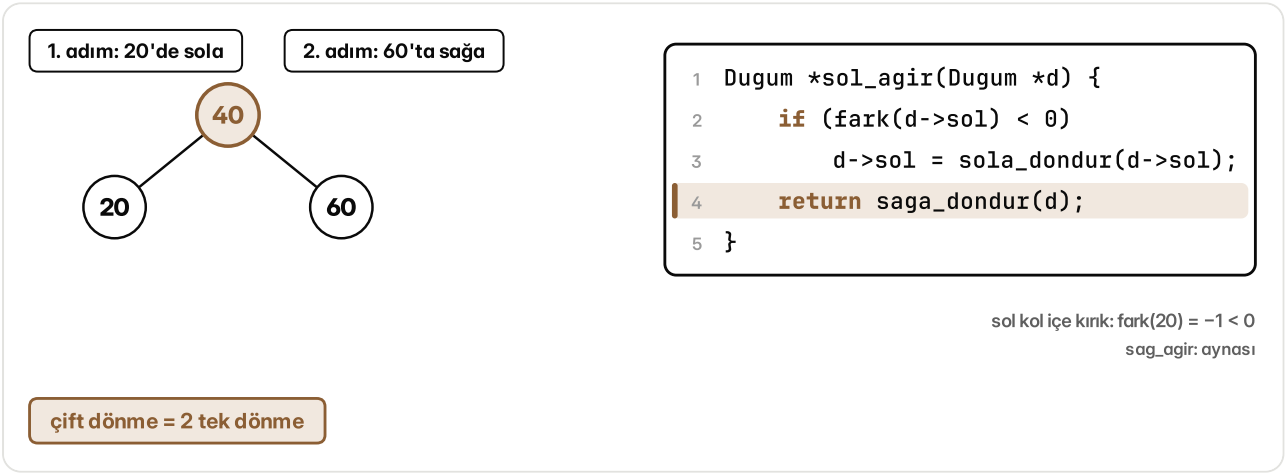
```
Dugum *dengele(Dugum *d) {
    guncelle(d);
    if (fark(d) < -1) return sag_agir(d);
    if (fark(d) > 1) return sol_agir(d);
    return d;
}
```

İçe kırıksa önce o kol döndürülür, sonra düğümün kendisi. sag_agir bunun aynasıdır:

```
Dugum *sol_agir(Dugum *d) {
    if (fark(d->sol) < 0)
        d->sol = sola_dondur(d->sol);
    return saga_dondur(d);
}
```

Çıktı:

```
zikzak:      60(20(-,40),-)   60'ta fark 2, 20'de fark -1
tek donme:   20(-,60(40,-))   kokte fark -2 (hala dengesiz)
1. adim:     60(40(20,-),-)   (20'de sola: duz cizgi)
2. adim:     40(20,60)   (60'ta saga) kokte fark 0
AVL ekle 60 20 40: 40(20,60)
```



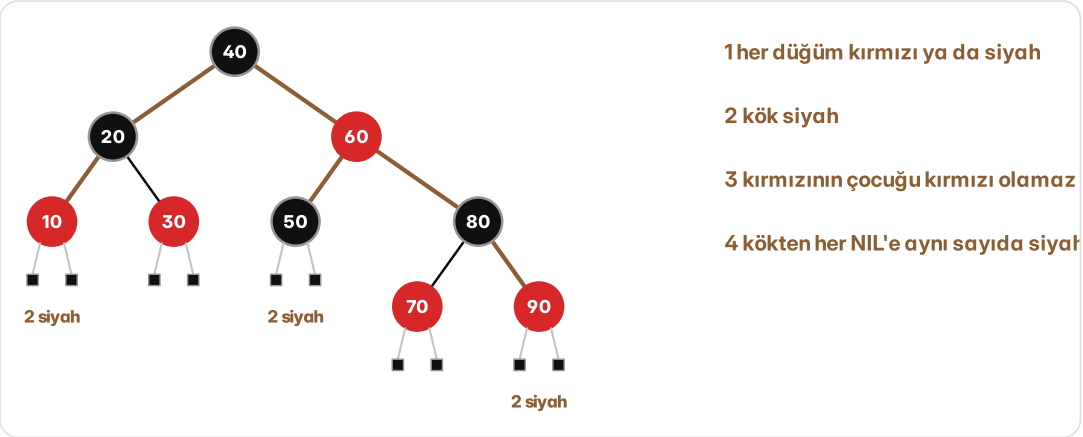
Şekil 6.5 Çift dönme: önce 20'de sola dönülür ve kol düzelir, sonra 60'ta sağa dönülür; 40 tepeye çıkar.

Fark en çok 1 olduğu için en dengesiz AVL ağacı bile kısa kalır. Her düğümde bir yanı öbüründen bir kat kısa olan en seyrek ağaçları sayalım: 1, 2, 4, 7, 12, 20 düğüm. Her biri önceki ikisinin toplamından 1 fazladır; Fibonacci sayıları gibi büyür. Bu yüzden yükseklik en çok 1,44 × log n olur. 50.000 düğümde en kısa ağaç 16 kattır, en kötü AVL ağacı 22 kat.

Sıkı dengenin bir bedeli var: ağaç sık sık döner. 50.000 karışık sayı eklediğimizde AVL ağacı 34.908 kez döndü. Tek bir silme ise köke kadar art arda dönmeler isteyebilir; 50.000 sayıyı karışık sırayla silen ölçümde tek bir silme en çok 7 dönme istedi.

Kırmızı-siyah ağaçta (red-black tree) her düğüm bir renk taşır ve ağaç 4 kurala uyar:

- Her düğüm kırmızı ya da siyahtır.
- Kök siyahtır.
- Kırmızı bir düğümün çocuğu kırmızı olamaz; iki kırmızı art arda gelmez.
- Kökten herhangi bir boş dala inen her yolda siyah düğüm sayısı aynıdır.



Şekil 6.6 Dokuz düğümlü kırmızı-siyah ağaçta kök 40 siyahtır, iki kırmızı art arda gelmez ve kökten her boş dala giden yolda 2 siyah vardır.

En kısa yol yalnız siyahlardan oluşabilir; en uzun yolda ise siyahla kırmızı sırayla dizilir, çünkü iki kırmızı yan yana gelemez. Siyah sayısı her yolda aynı olduğu için en uzun yol, en kısanın en çok iki katıdır. Yükseklik en çok 2 log n: 50.000 düğümde 31 kat. Arama biraz uzar, ama dengeyi korumak ucuzlar: bozulan kural kimi zaman yalnız renk değiştirerek düzelir; bir ekleme en çok 2, bir silme en çok 3 dönme ister. Aynı 50.000 sayıda kırmızı-siyah ağaç 28.911 kez döndü.

	AVL	Kırmızı-siyah
en kötü yükseklik (50.000 düğüm)	22 kat	31 kat
ekleme başına dönme	en çok 2	en çok 2
silme başına dönme	köke kadar	en çok 3
50.000 karışık ekleme	34.908 dönme	28.911 dönme

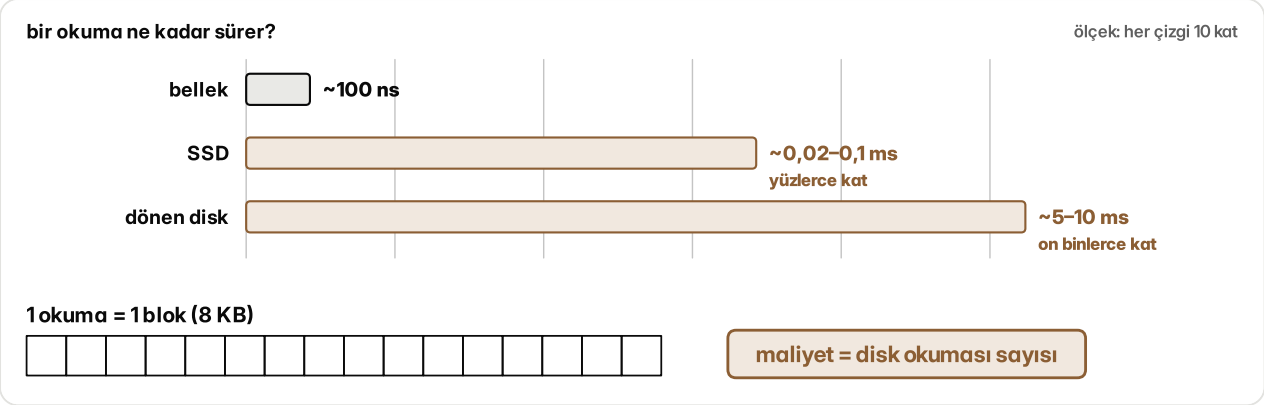
Sürekli ekleyip silen programlar için bu pazarlık kârlıdır. C++'ın sıralı sözlüğü std::map (gcc, clang ve MSVC kütüphanelerinde) de Java'nın TreeMap'i de kırmızı-siyah ağaçtır. İkisinde de arama, ekleme ve silme O(log n) sürer.

Dört okuma

videoda 40:37

Bir veritabanında 3 milyar kayıt var, hepsi diskte duruyor. Veritabanı aradığı kaydı nasıl oluyor da 3-4 disk okumasıyla bulur?

Diskten bir okuma, bellekten okumanın yüzlerce katı sürer; SSD'de bile böyle. Bellekten okuma yaklaşık 100 ns, SSD'den 0,02-0,1 ms sürer; eski tip dönen sabit diskte 5-10 ms ile fark on binlerce kata çıkar. Disk tek bir sayı da vermez; her okumada 8 KB gibi koca bir blok getirir. Burada hızı adım sayısı değil, disk okuması sayısı belirler.

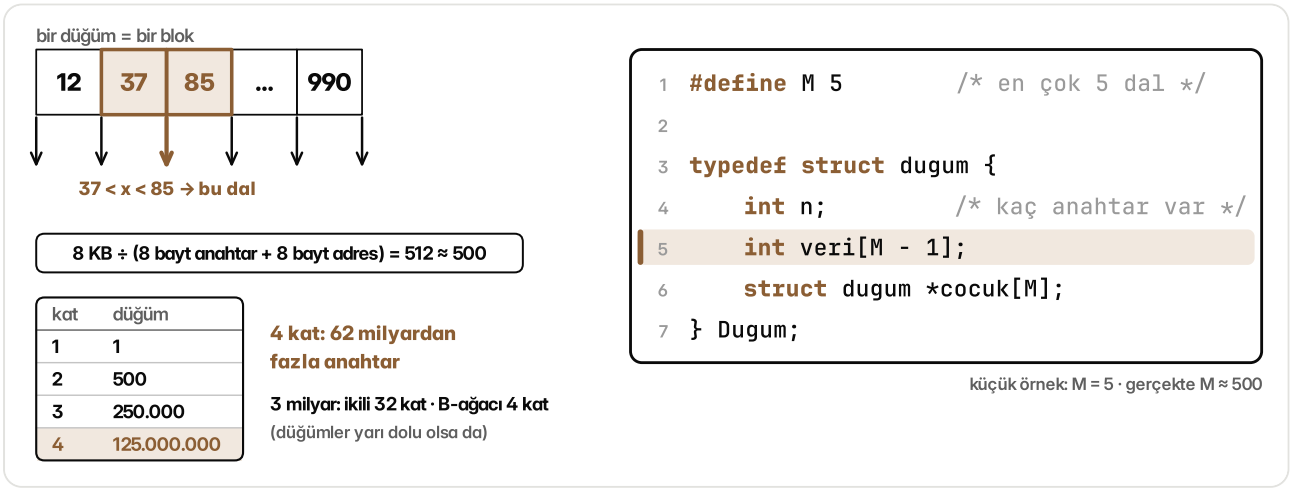


Şekil 6.7 Bellek ~100 ns, SSD ~0,02-0,1 ms, dönen disk ~5-10 ms; her okuma 8 KB'lık bir blok getirir ve maliyeti okuma sayısı belirler.

3 milyar kayıtlı dengeli bir ikili ağaç 32 kat derindir: 31 kat en çok 2^31 − 1, yaklaşık 2,15 milyar kayıt alır. Her kattaki düğüm diskte ayrı bir yerde durur; aramada her kat bir disk okuması demektir, toplam 32. Üstelik her okumada gelen koca bloktan yalnız tek bir sayı kullanılır, gerisi boşa gider.

B-ağacı (B-tree) bu israfı kaldırır: bir düğüm tam bir blok büyüklüğündedir ve içinde yüzlerce anahtar sıralı durur. 8 KB'lık bir bloğa, 8 baytlık anahtar ve 8 baytlık dal adresinden 8.192 / 16 = 512, yani 500 kadar çift sığar. Aranan sayı düğümde yoksa iki komşu anahtarın arasındaki dala inilir; böylece her düğümden 500 dal çıkar. Kök tek düğüm, 2. katta 500, 3. katta 250.000, 4. katta 125 milyon düğüm olur. 4 kat 60 milyarı aşkın anahtar alır; 3 milyar kayıt için, düğümler yarı boş olsa bile 4 kat yeter. C'de küçük bir örnek düğüm, en çok 5 dal ve 4 anahtarla şöyle tanımlanır; gerçekte M 500 kadardır:

```
#define M 5        /* en çok 5 dal */

typedef struct dugum {
    int n;         /* kaç anahtar var */
    int veri[M - 1];
    struct dugum *cocuk[M];
} Dugum;
```

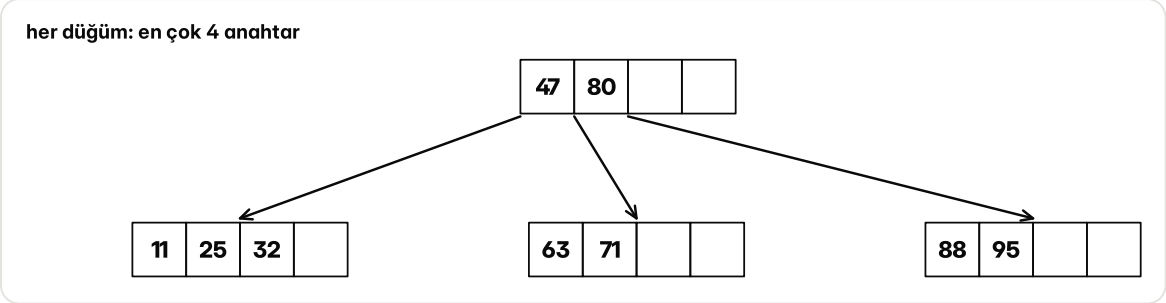


Şekil 6.8 B-ağacında bir düğüm bir bloktur; 500 dallı ağaç katlarda 1, 500, 250.000 ve 125.000.000 düğüm alır ve 3 milyar kayda 4 kat yeter.

Ekleme küçük bir örnekle görülsün; her düğüme en çok 4 anahtar sığsın. 25, 80, 47 ve 11 aynı düğüme sıralı dizilir: [11 25 47 80]. 63 gelince düğüm taşar ve ortadan ikiye bölünür: ortanca 47 yukarı çıkar, solunda 2, sağında 2 anahtar kalır. Yukarıda düğüm olmadığı için 47 yeni kök olur. Sonra 95, 32, 71 ve 88 gelir; sağdaki yaprak yine taşar, bu kez ortanca 80 köke çıkar.

Çıktı:

```
11 eklendi
   [11 25 47 80]   (kat 1)
63 eklendi
   kok bolundu: 47 yukari cikti, agac bir kat uzadi
   [47]([11 25] [63 80])   (kat 2)
```



Şekil 6.9 Her düğüme en çok 4 anahtar sığan B-ağacında taşan düğümler bölünür; kök [47 80], yapraklar [11 25 32], [63 71] ve [88 95] olur.

İkili arama ağacı aşağı doğru uzar; B-ağacı ise yukarı doğru büyür. Yeni kat yalnız kök bölündüğünde, en tepeye eklenir. Bu yüzden bütün yapraklar hep aynı derinlikte kalır: ağaç dönmeye gerek kalmadan kendiliğinden dengelidir. 71'i aramak için 2 düğüm okunur: kökte 71, 47 ile 80 arasına düşer; o dala inilir ve yaprakta bulunur. Kodda iç döngü anahtarın düğümdeki yerini bulur:

```
int ara(Dugum *d, int x) {
    while (d != NULL) {
        int i = 0;
        while (i < d->n && x > d->veri[i])
            i++;
        if (i < d->n && x == d->veri[i])
            return 1;
        d = d->cocuk[i];
    }
    return 0;
}
```

B-ağacı 1970'te Boeing'in laboratuvarında, büyük dosyaların indeksleri için doğdu (Rudolf Bayer ve Edward McCreight). Bugün veritabanlarının indeksleri bu ağaçla çalışır; çoğu, kayıtları yalnız yapraklarda tutan türünü, B+ ağacını kullanır. Windows'un dosya sistemi NTFS de bir klasördeki dosya adlarını böyle bir ağaçta tutar. 3 milyar kayıtta 4 kat, en çok 4 disk okuması gerekir; üst iki kat (501 düğüm, yaklaşık 4 MB) zaten bellekte durduğu için çoğu zaman daha azı yeter. Arama ve ekleme O(log n) sürer; disk okuması sayısı kat sayısı kadardır.

Özet

- AVL ağacında her düğümün sol ve sağ alt ağaç yükseklikleri en çok 1 farklıdır; fark 2 olunca dönmeyle düzeltilir.
- Dönme birkaç adres değiştirir, O(1) sürer ve sayıların sırasını bozmaz; zikzakta önce kol, sonra düğüm döndürülür.
- AVL ağacının yüksekliği en çok 1,44 log n'dir ama ağaç sık döner; tek silme köke kadar dönme isteyebilir.
- Kırmızı-siyah ağaç dört renk kuralıyla yüksekliği en çok 2 log n tutar ve daha az döner; C++ std::map ve Java TreeMap bu ağaçtır.
- Diskte maliyeti okuma sayısı belirler; B-ağacının düğümü bir bloktur ve yüzlerce anahtar tutar.
- B-ağacında taşan düğüm bölünür, ağaç yukarı doğru büyür ve bütün yapraklar aynı derinlikte kalır; 3 milyar kayda 4 kat yeter.

Sorular

1 **AVL ağacının kuralı nedir?**

A Bütün yapraklar aynı derinlikte olmalıdır.

B Her düğümde sol ve sağ alt ağacın yükseklikleri en çok 1 farklı olabilir.

C Kökten her boş dala giden yolda siyah düğüm sayısı aynı olmalıdır.

D Her düğümün tam 2 çocuğu olmalıdır.

2 **Boş bir AVL ağacına sırayla 30, 10 ve 20 ekleniyor. Ne olur?**

A Hiç dönme gerekmez; kök 30 kalır.

B Tek bir sağa dönme yeter; kök 10 olur.

C Kol içe kırık olduğu için çift dönme gerekir; kök 20 olur.

D Tek bir sola dönme yeter; kök 30 kalır.

3 **Boş bir AVL ağacına sırayla 3, 2, 1, 4, 5 ekleniyor. Son ağacı seviye seviye (yukarıdan aşağı, her katta soldan sağa) yaz.**

Cevap:

4 **AVL ağacında 5 düğümü sola döndürülüyor; 5'in sağ çocuğu 8, 8'in sol çocuğu 6. Dönmeden sonra 6 nereye geçer?**

A 8'in sağ çocuğu olur.

B Ağaçtan çıkarılır ve yeniden eklenir.

C 8'in sol çocuğu olarak kalır.

D 5'in sağ çocuğu olur.

5 **C++'ın std::map'i ve Java'nın TreeMap'i neden AVL ağacı yerine kırmızı-siyah ağaç kullanır?**

A Kırmızı-siyah ağaç daha az döner; sürekli ekleyip silen programlarda dengeyi korumak ucuzdur.

B Kırmızı-siyah ağaç her zaman AVL ağacından daha kısadır.

C AVL ağacında silme yapılamaz.

D Kırmızı-siyah ağaçta arama O(1) sürer.

6 **Kırmızı-siyah bir ağaçta kırmızı bir düğümün kırmızı bir çocuğu var. Hangi kural bozulmuştur?**

A Kök siyahtır.

B Her düğüm kırmızı ya da siyahtır.

C Kökten her boş dala giden yolda siyah sayısı aynıdır.

D Kırmızı bir düğümün çocuğu kırmızı olamaz.

7 **Her düğüme en çok 4 anahtar sığan boş bir B-ağacına sırayla 10, 40, 20, 30, 50 ekleniyor. Taşan düğüm ortadan bölündükten sonra kökte hangi anahtar durur?**

Cevap: _____

8 **Her düğümünden 100 dal çıkan, düğümleri tam dolu bir B-ağacında her düğüm 99 anahtar tutar. 3 katlı böyle bir ağaç en çok kaç anahtar alır?**

Cevap: _____

VERİ YAPILARI

7 Heap ve öncelikli kuyruk

Bu bölümde en öncelikli elemanı her an tek adımda veren öncelikli kuyruğu ve onu gerçekleştiren heap yapısını göreceğiz: heap kuralını, tam ağacın bir diziye işaretçisiz nasıl sığdığını, eklemeyi, bir diziyi aşağıdan yukarı heap'e çevirmeyi ve kökü almayı.

Sıradaki iş hangisi?

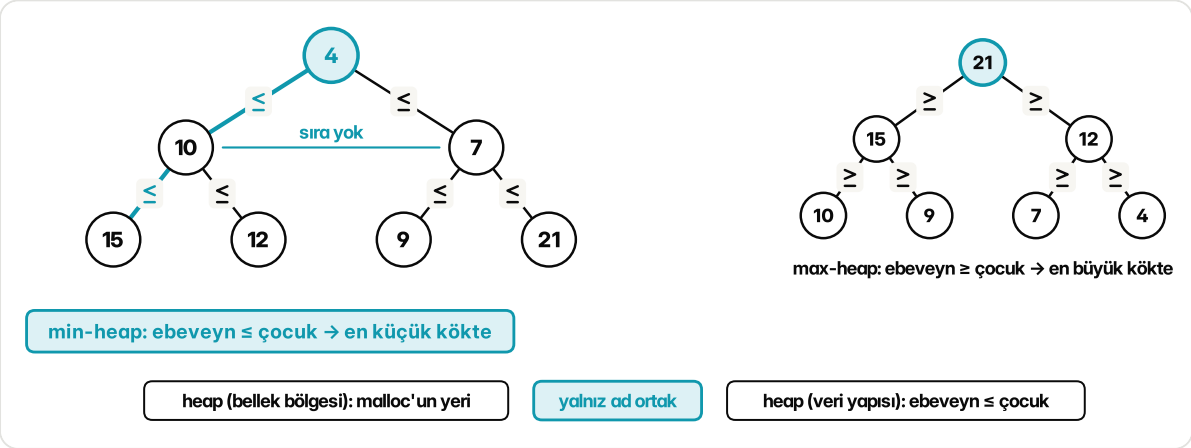
videoda 42:57

Bir sunucuda 10.000 iş, çalışacağı saniyeyi bekliyor ve her an yenileri ekleniyor. Zamanı en yakın işi, her yeni gelişten sonra da anında nasıl buluruz?

Acil servis de böyle bir sıradır: kalp krizi geçiren, saattir bekleyen grip hastasından önce girer. Sıradan kuyrukta ilk giren ilk çıkar; burada önce en öncelikli olan çıkar. Buna **öncelikli kuyruk** (priority queue) denir. İki işi vardır: eklemek ve en önceliklisini çıkarmak. İşletim sistemi de sıradaki işi önceliğe göre seçer. Sunucudaki işlerde öncelik zamandır: saniye küçüldükçe iş öne geçer.

İki kolay yol var, ikisi de aksar. İşleri geldikleri gibi dizinin sonuna yazarsak ekleme tek adımdır, ama en yakın işi bulmak için 10.000 işin hepsine bakmak gerekir. Diziyi sıralı tutarsak en yakın iş hep baştadır, ama her yeni iş yerine girerken arkasındakiler birer göz kayar: ortalama 5.000 kaydırma.

Çözüm, tam sıralamadan vazgeçmektir. İşleri bir ağaca dizer, tek kural koyarız: her düğümün saniyesi, çocuklarınınkinden küçük ya da ona eşittir. Kardeşler arasında sıra aranmaz: aşağıdaki ağaçta kök 4'ün solunda 10, sağında 7 durur. Her düğüm çocuklarından küçük olduğu için en küçük saniye kendiliğinden köke yerleşir. Bu yapıya **min-heap** denir. Kural ters çevrilirse kökte en büyük durur; adı da **max-heap** olur. malloc'un (çalışırken bellek ayıran fonksiyon) kullandığı heap bölgesiyle bu yapının yalnız adı ortaktır.



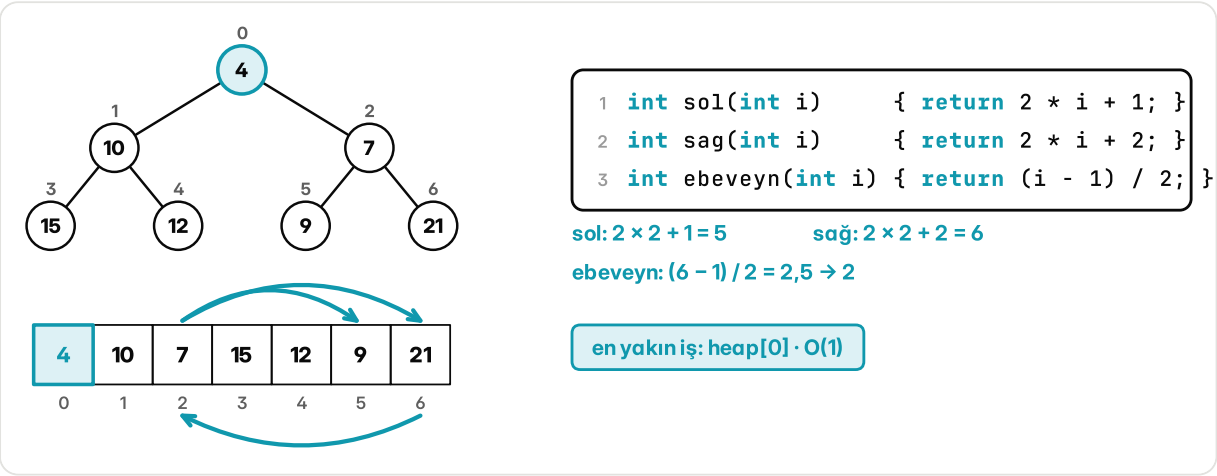
Şekil 7.1 Min-heap 4 10 7 15 12 9 21: her ebeveyn çocuklarından küçük ya da eşittir, 10 ile 7 arasında sıra yoktur; yanında en büyüğü kökte tutan max-heap.

Heap ayrıca **tam ikili ağaçtır**: katlar yukarıdan aşağı dolar, son kat da soldan boşluksuz. Boşluk kalmadığı için ağaç kısa kalır: 14 kat en çok 16.383 düğüm alır, bu yüzden 10.000 iş 14 kata sığar ve kökten en alttaki yaprağa 13 adım vardır.

Tam ağacın bir kazancı daha var: işaretçiye gerek kalmaz. Düğümleri kat kat, soldan sağa bir diziye yazarız: kök 0. gözde, çocukları 1'de ve 2'de, onların çocukları 3'ten 6'ya. Bir gözün numarasını 2 ile çarpıp 1 eklersek sol çocuğa, 2 eklersek sağ çocuğa varırız: 2. gözdeki 7'nin çocukları 5. ve 6. gözlerdedir. Geri dönmek için 1 çıkarıp 2'ye böleriz, küsurat atılır: (6 − 1) / 2 = 2. C'de pozitif tam sayıların bölümü küsuratı zaten atar:

```
int sol(int i)     { return 2 * i + 1; }
int sag(int i)     { return 2 * i + 2; }
int ebeveyn(int i) { return (i - 1) / 2; }
```

Zamanı en yakın iş hep 0. gözdedir; ona bakmak tek adımdır, O(1).



Şekil 7.2 Ağacın düğümleri 0-6. gözlere yazılır: 2. gözün çocukları 2 × 2 + 1 = 5 ve 2 × 2 + 2 = 6, 6. gözün ebeveyni (6 − 1) / 2 = 2; en yakın iş heap[0].

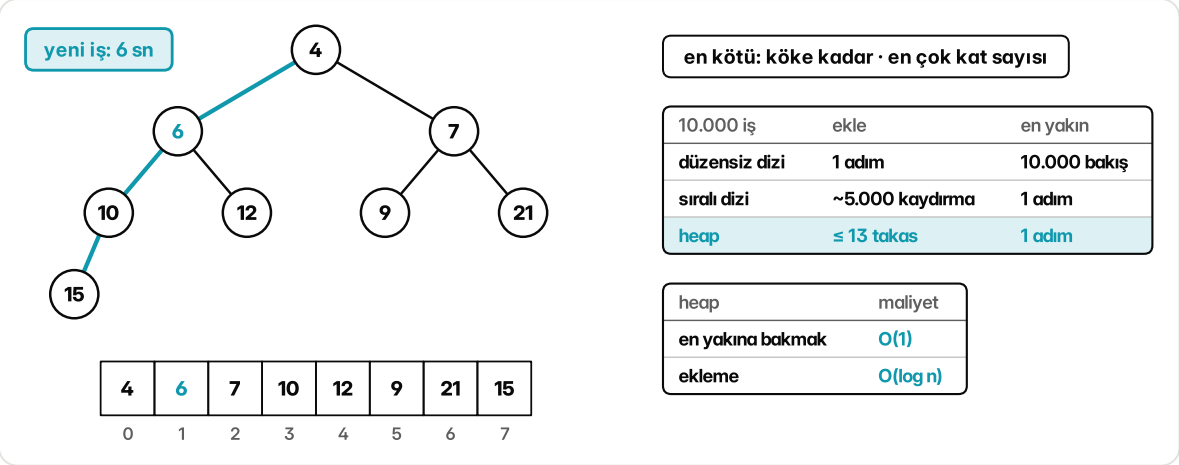
Yeni bir iş geldi: 6. saniyede çalışacak. Önce dizinin ilk boş gözüne, yani son kattaki ilk boş yere konur; ağaç tam kalır. Sonra ebeveyniyle karşılaştırılır: 15, 6'dan büyük, kural bozulmuş; ikisi yer değiştirir. Yeni ebeveyn 10, yine büyük; bir takas daha. Üstte 4 var, 6'dan küçük: tırmanış biter. Bunu ekle yapar; takas, iki gözün değerini yer değiştiren bir yardımcıdır:

```
void ekle(int heap[], int *n, int x) {
    int i = (*n)++;
    heap[i] = x;
    while (i > 0) {
        int e = ebeveyn(i);
        if (heap[e] <= heap[i]) break;
        takas(heap, e, i);
        i = e;
    }
}
```

Tam program her takası yazdırır:

Çıktı:

```
hip: 4 10 7 15 12 9 21
2. gözün çocukları: 5 ve 6 (9, 21)
6. gözün ebeveyni: 2 (7)
1. gözün çocukları: 3 ve 4 (15, 12)
6 eklenir, 7. göze: 4 10 7 15 12 9 21 6
takas 15 <-> 6: 4 10 7 6 12 9 21 15
takas 10 <-> 6: 4 6 7 10 12 9 21 15
sonuç: 4 6 7 10 12 9 21 15
en yakın iş: 4 (0. göz)
```



Şekil 7.3 6 sona konur, 15 ve 10 ile yer değiştirip 1. göze çıkar: 4 6 7 10 12 9 21 15; tabloda heap'in ekleme ve en yakına bakma maliyeti.

Yeni iş ne kadar acil olursa olsun en çok köke kadar çıkar: 10.000 işte en çok 13 takas. Kat sayısı log n kadar olduğu için ekleme O(log n)'dir.

10.000 iş	Ekleme	En yakını bulmak
Düzensiz dizi	1 adım	10.000 bakış
Sıralı dizi	ortalama 5.000 kaydırma	1 adım
Min-heap	en çok 13 takas, O(log n)	1 adım, O(1)

Pek çok programlama dili, "şu kadar süre sonra çalıştır" denen işleri tam bu yapıda, min-heap'te tutar; zamanı en yakın olan hep ilk gözde bekler.

İlk on

videoda 45:34

Bir alışveriş sitesinde 10 milyon ürünün satış sayısı bir dizide duruyor. En çok satan 10 ürünü bulmak için bu diziyi baştan sona sıralamak gerekir mi?

Sayıları birbiriyle karşılaştırarak sıralamak, en hızlı yöntemle bile 200 milyonu aşkın karşılaştırma ister: ürün başına 20 küsur. Oysa bize yalnız ilk 10 lazım; diziyi daha gevşek bir düzene, max-heap'e sokmak yeter: her düğüm çocuklarından büyük ya da onlara eşittir, en büyük kökte durur.

Heap tam ikili ağaç olduğu için dizinin kendisi kat kat okununca ağaçtır; i. gözün çocukları 2i + 1 ve 2i + 2'dedir. 10 milyon ürün bu ağaçta 24 kat tutar.

İlk akla gelen yol ürünleri tek tek eklemektir. Satışlar küçükten büyüğe sıralı geldiyse her ürün köke kadar tırmanır; toplam 200 milyonu aşkın takas, sıralamak kadar pahalı.

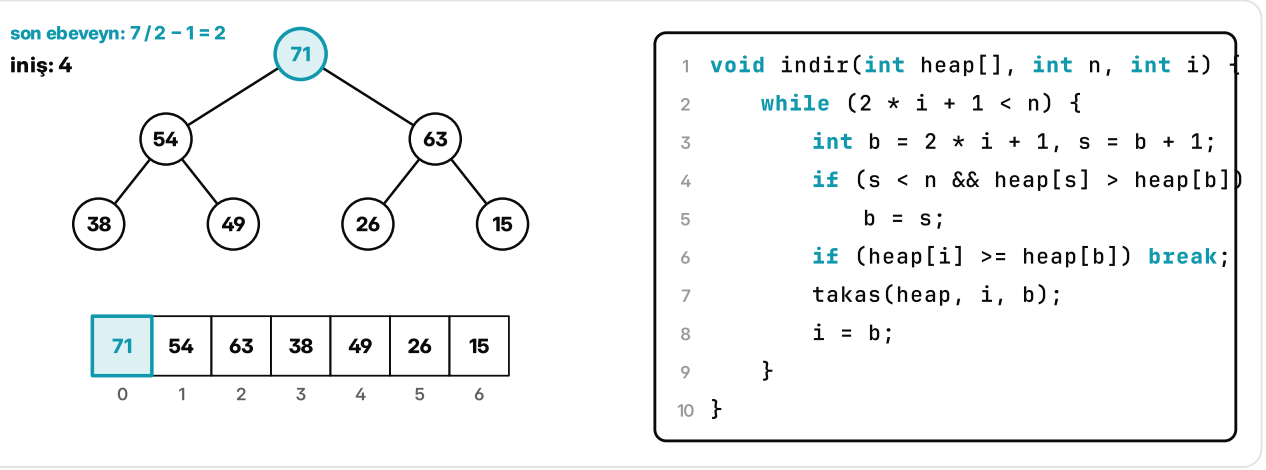
Daha iyi yol, aşağıdan yukarı kurmaktır. Dizinin arka yarısı yapraktır ve her yaprak tek başına zaten heap'tir. Son ebeveynden başlayıp geriye doğru her düğümü aşağı indiririz. Geriye gittiğimiz için bir düğüme geldiğimizde altındaki iki alt ağaç zaten heap'tir; bozuk olan yalnız düğümün kendisidir. Düğüm, büyük çocuğuyla karşılaştırılır; ondan küçükse yer değiştirir. İki çocuğundan da büyük olunca ya da yaprağa varınca durur. Kodda b büyük çocuğun, s sağ çocuğun gözüdür:

```
void indir(int heap[], int n, int i) {
    while (2 * i + 1 < n) {
        int b = 2 * i + 1, s = b + 1;
        if (s < n && heap[s] > heap[b])
            b = s;
        if (heap[i] >= heap[b]) break;
        takas(heap, i, b);
        i = b;
    }
}
```

heap_kur, son ebeveynden (n / 2 − 1) köke kadar her düğümü indirir:

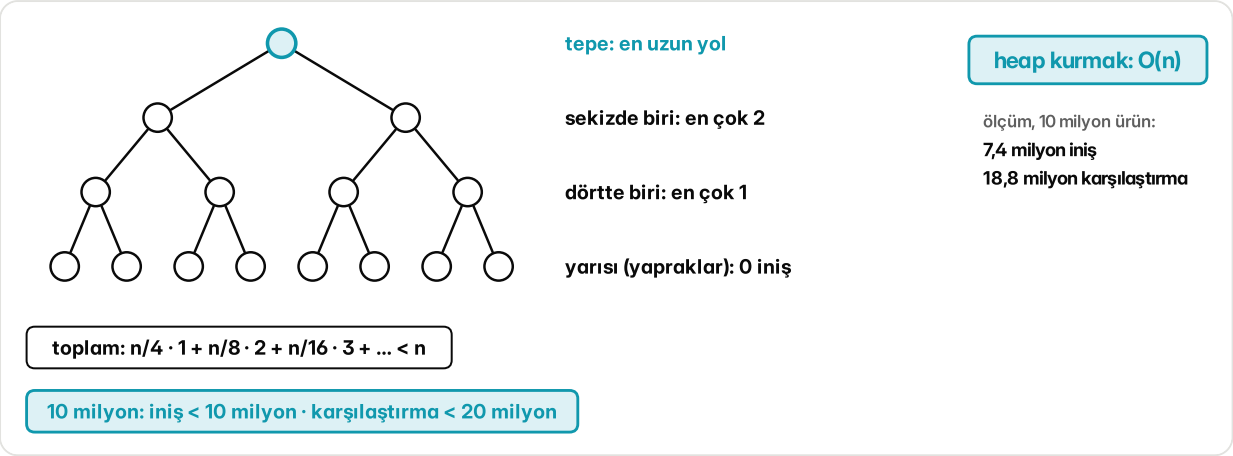
```
void heap_kur(int heap[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        indir(heap, n, i);
}
```

7 ürünlük dizi 38 54 26 71 49 63 15'te son ebeveyn 7 / 2 − 1 = 2. gözdür. Bu gözdeki 26, büyük çocuğu 63'le yer değiştirir. Sonra 1. gözdeki 54, 71'le. En sonda kökteki 38 önce 71'le, sonra 54'le yer değiştirip iner. 4 inişte dizi heap olur: 71 54 63 38 49 26 15, kökte 71.



Şekil 7.4 38 54 26 71 49 63 15 son ebeveynden (2. göz) geriye doğru indirilir; 4 inişte heap 71 54 63 38 49 26 15 olur.

Neden bu kadar ucuz? Düğümlerin yarısı yapraktır, hiç inmez. Dörtte biri en çok 1 kat iner, sekizde biri en çok 2 kat; uzun yolu yalnız tepedeki birkaç düğüm gider. Hepsi toplanınca (n/4 · 1 + n/8 · 2 + n/16 · 3 + ...) iniş sayısı eleman sayısını geçmez: 10 milyon üründe 10 milyondan az iniş, 20 milyondan az karşılaştırma. Heap kurmanın maliyeti O(n)'dir.



Şekil 7.5 Yapraklar hiç inmez, dörtte biri en çok 1, sekizde biri en çok 2 kat iner; toplam iniş n'den az, heap kurmak O(n).

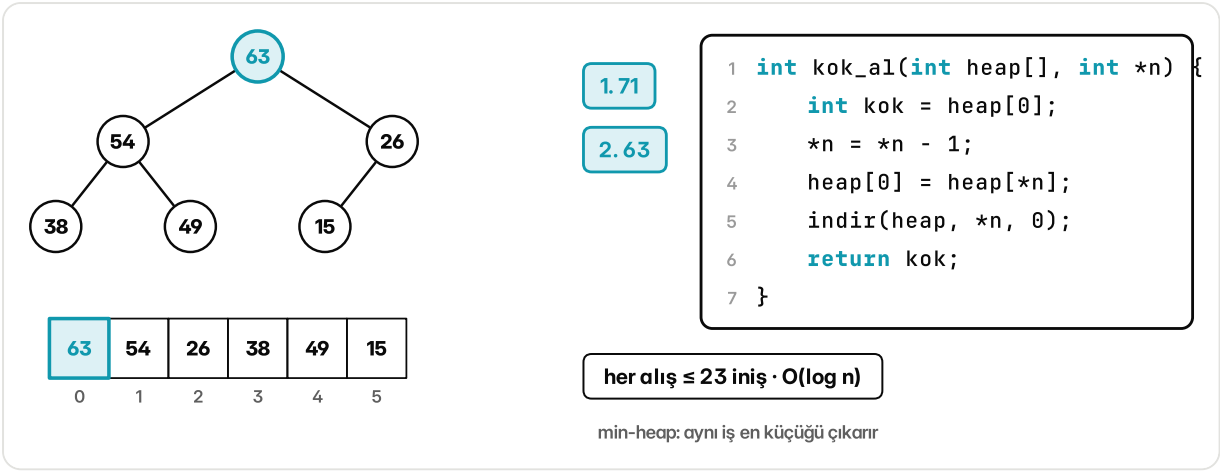
Şimdi en çok satan kökte. Onu alırız; ağaç tam kalsın diye boşalan köke dizinin son elemanı gelir ve aşağı indirilir:

```
int kok_al(int heap[], int *n) {
    int kok = heap[0];
    *n = *n - 1;
    heap[0] = heap[*n];
    indir(heap, *n, 0);
    return kok;
}
```

Örnekte 71 çıkar, 15 köke gelir; önce 63'le, sonra 26'yla yer değiştirir. Kökte 63 kalır: ikinci en çok satan. Tam program kurmayı ve ilk üç alışı yazdırır:

Çıktı:

```
dizi: 38 54 26 71 49 63 15
son ebeveyn: 2. göz
  26 iner, 63 çıkar: 38 54 63 71 49 26 15
  54 iner, 71 çıkar: 38 71 63 54 49 26 15
  38 iner, 71 çıkar: 71 38 63 54 49 26 15
  38 iner, 54 çıkar: 71 54 63 38 49 26 15
hip: 71 54 63 38 49 26 15
toplam 4 iniş
  15 iner, 63 çıkar: 63 54 15 38 49 26
  15 iner, 26 çıkar: 63 54 26 38 49 15
1. en büyük: 71, kalan: 63 54 26 38 49 15
2. en büyük: 63, kalan: 54 49 26 38 15
3. en büyük: 54, kalan: 49 38 26 15
```



Şekil 7.6 Kök 71 alınır, son eleman 15 köke gelip 63 ve 26 ile yer değiştirir; kökte 63 kalır, her alış en çok 23 iniştir.

Her alış en çok kat sayısı kadar iniştir: 10 milyon üründe 23. Maliyeti O(log n); min-heap'te aynı iş en küçüğü çıkarır. İlk 10'u bulmak main'de üç satırdır:

```
heap_kur(satis, n);
for (int k = 0; k < 10; k++)
    printf("%d\n", kok_al(satis, &n));
```

20 milyondan az karşılaştırmayla heap kurulur, 10 kez kök almak 500'den az karşılaştırma daha tutar. Sıralamanın onda birinden az işle ilk 10 elde. Bu bilgisayardaki ölçüm 10 milyon rastgele satışta bunu C'nin hazır sıralama fonksiyonu qsort ile karşılaştırır:

Çıktı:

```
hip kurmak: 18813259 karşılaştırma, 7439037 iniş
on kez kök almak: 446 karşılaştırma
sıralamak (qsort): 271659853 karşılaştırma
oran: 14.4
ilk on aynı mı: evet
ilk on: 4999999 4999998 4999998 4999998 4999997 4999997 4999997 4999994 4999992 4999991
```

Bu çalıştırmada heap yolu 14,4 kat az karşılaştırmayla aynı ilk 10'u buldu. En çok satanlar, en çok izlenenler, en yüksek puanlılar: hepsi sıralamadan bulunabilir.

Heap işlemi	Maliyet
En öncelikliye bakmak (heap[0])	O(1)
Ekleme (sona koy, yukarı taşı)	O(log n)
Kökü almak (son eleman köke, aşağı indir)	O(log n)
Diziden heap kurmak (aşağıdan yukarı)	O(n)
İlk 10'u bulmak	O(n + 10 log n)

Özet

- Öncelikli kuyruk iki iş yapar: eleman eklemek ve en öncelikli elemanı çıkarmak; heap bunu hızlı yapan yapıdır.
- Min-heap'te her düğüm çocuklarından küçük ya da eşittir ve en küçük köktedir; max-heap'te tersi, en büyük köktedir. Kardeşler arasında sıra yoktur.
- Heap tam ikili ağaçtır; dizide i. gözün çocukları 2i + 1 ve 2i + 2, ebeveyni (i – 1) / 2'dir, işaretçi gerekmez.
- Ekleme yeni elemanı sona koyup ebeveyninden küçük oldukça yukarı taşır; kök almak son elemanı köke koyup aşağı indirir; ikisi de O(log n).
- Diziyi aşağıdan yukarı, son ebeveynden geriye indirerek heap'e çevirmek O(n)'dir; tek tek eklemek en kötü durumda sıralamak kadar pahalıdır.
- En büyük 10'u bulmak için diziyi sıralamak yerine heap kurup 10 kez kök almak yeter.

Sorular

1 **Bir min-heap için aşağıdakilerden hangisi her zaman doğrudur?**

A Dizi küçükten büyüğe sıralıdır.

B Sol çocuk her zaman sağ çocuktan küçüktür.

C En küçük eleman köktedir, yani dizinin 0. gözündedir.

D En büyük eleman dizinin son gözündedir.

2 **Heap'i bir diziye işaretçi kullanmadan yazabilmemizi sağlayan özelliği nedir?**

A Tam ikili ağaç olması: katlar soldan boşluksuz dolduğu için i. gözün çocukları 2i + 1 ve 2i + 2. gözlerdedir.

B Kardeşlerin kendi aralarında sıralı olması.

C Bütün elemanların birbirinden farklı olması.

D Ağacın her eklemeden sonra baştan sıralanması.

3 **0'dan başlayan bir heap dizisinde 11. gözdeki elemanın ebeveyni kaçıncı gözdedir?**

Cevap: _____

4 **10.000 elemanlı bir min-heap'e yeni bir eleman ekleniyor. En kötü durumda kaç takas yapılır?**

A 14

B 5.000

C 10.000

D 13

5 **Min-heap dizisi 4 10 7 15 12 9 21. Bu heap'e 3 eklenince dizi ne olur? Elemanları aralarında boşlukla yaz.**

Cevap: _____

6 **n elemanlı bir diziyi aşağıdan yukarı, son ebeveynden geriye doğru her düğümü indirerek heap'e çevirmenin maliyeti nedir?**

A O(log n)
B O(n)
C O(n log n)
D O(n^2)

7 **Max-heap dizisi 71 54 63 38 49 26 15. Kök alınınca dizi ne olur?**

A 63 54 15 38 49 26
B 54 63 26 38 49 15
C 63 54 26 38 49 15
D 15 54 63 38 49 26

8 **Aşağıdaki dizi heap_kur ile max-heap'e çevriliyor. Sonuçta dizi ne olur? Elemanları aralarında boşlukla yaz.**

```
int satis[5] = {20, 50, 30, 10, 40};
heap_kur(satis, 5);
```

Cevap: _____

VERİ YAPILARI

8 Sıralama algoritmaları

Bu bölümde üç karşılaştırmalı sıralama yöntemini ve sıralamanın hız sınırını göreceğiz: birleştirmeli sıralamayı ve kararlılığı, hızlı sıralamayı ve pivot seçimini, heap sıralamasını; sonra karşılaştırarak sıralamanın neden n log n'in altına inemediğini ve sayma ile taban sıralamasının bu sınırı nasıl aştığını.

Şehre göre sırala

videoda 48:13

Adlara göre alfabetik sıralı bir müşteri listesini bir de şehre göre sıralarsak aynı şehirdekiler hâlâ alfabetik kalır mı? Cevap, kullanılan sıralama algoritmasına bağlıdır; **birleştirmeli sıralama** (merge sort) bu sırayı korur.

Altı kayıt alalım, adlara göre sıralı: Arda İzmir, Burcu Ankara, Can Bursa, Derya Bursa, Emre İzmir, Figen Ankara. Birleştirmeli sıralama listeyi ikiye böler, her yarıyı aynı yolla sıralar, sonra iki sıralı yarıyı tek listede birleştirir. sirala fonksiyonunda b parçanın ilk gözü, s son gözünden bir sonraki göz (s parçaya dahil değil), o ortadır; iki elemandan kısa bir parça zaten sıralıdır:

```
void sirala(int b, int s) {
    if (s - b < 2)
        return;
    int o = (b + s) / 2;
    sirala(b, o);
    sirala(o, s);
    birlestir(b, o, s);
}
```

Yarılar şehre göre sıralanınca solda Burcu Ankara, Can Bursa, Arda İzmir; sağda Figen Ankara, Derya Bursa, Emre İzmir kalır. Asıl iş son adımda, birleştirmededir.

Birleştirme iki yarının yalnızca başına bakar, şehri alfabede önce geleni alır ve yandaki ikinci bir diziye yazar. Burcu ile Figen'in ikisi de Ankara'dan; eşitlikte soldakini, Burcu'yu alırız. Can ile Figen'de Ankara önce gelir: Figen. Can ile Derya yine eşit: soldaki Can. Arda ile Derya'dan Derya; Arda ile Emre eşit: Arda; en son, karşılaştırmasız, Emre. 5 karşılaştırmada 6 kayıt yerine kondu; sonra ikinci dizi, asıl dizinin üstüne geri kopyalanır. Kodda a asıl dizi, gec ikinci dizidir; her kaydın ad ve il alanı vardır. strcmp iki metni karşılaştırır: ilki önce geliyorsa negatif, eşitse 0, sonra geliyorsa pozitif bir sayı döndürür:

```
void birlestir(int b, int o, int s) {
    int i = b, j = o, k = b;
    while (i < o && j < s)
        if (strcmp(a[i].il, a[j].il) <= 0)
            gec[k++] = a[i++];
        else
            gec[k++] = a[j++];
    while (i < o) gec[k++] = a[i++];
    while (j < s) gec[k++] = a[j++];
    for (k = b; k < s; k++) a[k] = gec[k];
}
```

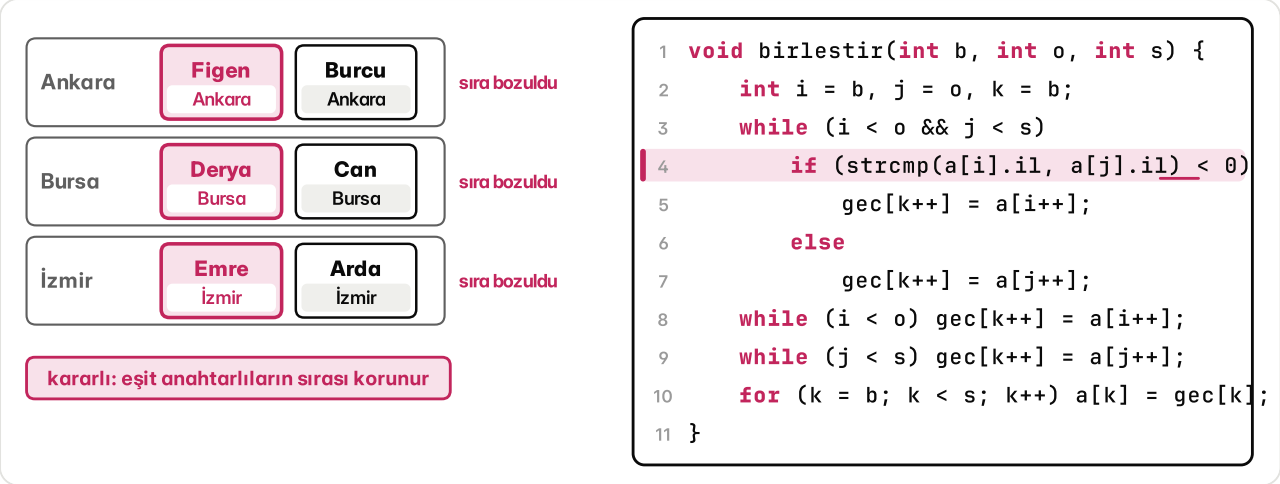
Çıktı:

```
Burcu Ankara
Figen Ankara
Can Bursa
Derya Bursa
Arda İzmir
Emre İzmir
```



Şekil 8.1 Sol ve sağ yarının başları karşılaştırılır, öndeki ikinci diziye iner; 5 karşılaştırmada altı kayıt yerine konur.

Sonuçta Ankara'da Burcu ile Figen, Bursa'da Can ile Derya, İzmir'de Arda ile Emre var; her şehrin içinde adlar hâlâ alfabetik. Eşit anahtarlı kayıtların eski sırasını koruyan sıralamaya **kararlı sıralama** (stable sort) denir. C kodunda bunu tek bir karşılaştırma sağlar: soldaki küçük ya da eşitse (<= 0) soldakini al. Eşitliği çıkarıp yalnızca küçükse (< 0) dersek, eşitlikte sağdaki alınır: Figen Burcu'nun, Derya Can'ın, Emre Arda'nın önüne geçer. Liste yine şehre göre sıralıdır, ama adların sırası bozulur.



Şekil 8.2 Kararlı birleştirmede her şehrin içinde adlar alfabetik kalır; eşitlikte sağdaki alınırsa her şehirde adların sırası bozulur.

Birleştirmeli sıralama girdiye bakmadan hep yarıya böler; her katta her kayıt bir kez yazılır. Bu yüzden hiçbir girdi onu yavaşlatamaz. 1.000 sayıyla yapılan denemede sayılar sıralı gelince 4.932, ters sıralı gelince 5.044, karışık gelince 8.700 civarı karşılaştırma yapıldı; en kötü durumda bile 8.977'yi, yani 9.000'i geçmez. Her girdide O(n log n). Bedeli bellek: birleştirme, dizinin kendisi kadar ikinci bir alan ister, O(n). int 4 bayt olduğu için 10 milyon tam sayıyı sıralamak 40 MB'lık ikinci bir dizi gerektirir.



Şekil 8.3 1.000 sayıda sıralı 4.932, ters 5.044, karışık ≈ 8.700, en kötü ≤ 8.977 karşılaştırma; asıl dizinin yanında aynı boyda ikinci dizi.

C'nin hazır sıralama fonksiyonu qsort kararlılık sözü vermez; dilin standardı, eşit elemanların sırasını belirsiz bırakır. Java'nın nesne dizileri ve Chrome tarayıcısındaki diziler ise Timsort ile sıralanır. Timsort, birleştirmeli sıralamanın, listede zaten sıralı duran parçaları bulup doğrudan birleştiren bir türüdür ve kararlıdır. Bir tabloyu önce ada, sonra şehre göre sıraladığımızda adların dağılmaması bu yüzdendir.

Ek dizi olmadan

videoda 50:30

8 sayıyı, yanına ikinci bir dizi açmadan, yalnızca yer değiştirerek nasıl sıralarız? Birleştirmeli sıralama bunu yapamaz; ona dizi kadar ek yer gerekir. **Hızlı sıralama** (quicksort) ise işi dizinin içinde bitirir.

Dizi 38 71 14 92 55 26 63 47. Sondaki sayıyı, 47'yi **pivot** seçeriz; amaç ondan küçükleri sola toplamak. Dizinin başına bir sınır koyarız: sınırın solu küçüklerin bölgesidir, şimdilik boş. Sonra soldan sağa yürürüz. 38 küçük: bölgeye girer, sınır bir ilerler. 71 büyük, geçeriz. 14 küçük: sınırdaki 71'le yer değiştirir, sınır yine ilerler. 92 ile 55 geçilir; 26 da sınırdaki sayıyla yer değiştirir, 63 geçilir. Son adımda pivot, sınırdaki 92'yle yer değiştirir. Kodda i sınırı, j tarayan gözü tutar; takas iki int'in yerini değiştirir:

```
int ayir(int a[], int b, int s) {
    int pivot = a[s], i = b;
    for (int j = b; j < s; j++)
        if (a[j] < pivot) {
            takas(&a[i], &a[j]);
            i++;
        }
    takas(&a[i], &a[s]);
    return i;
}
```

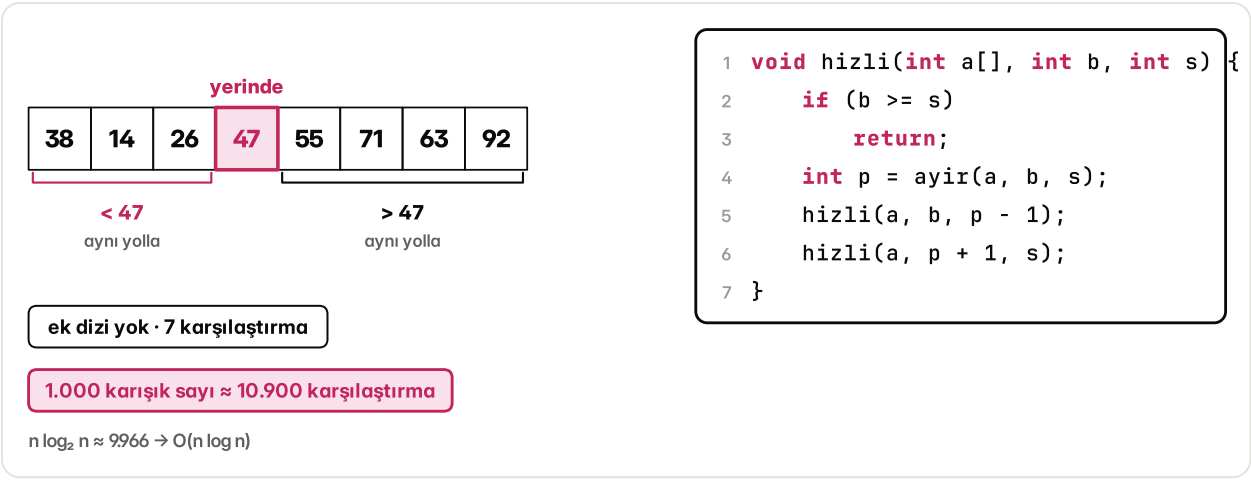
47 artık kesin yerinde: solunda 3 küçük, sağında 4 büyük sayı var; hiç ek dizi açılmadı. İki taraf aynı yolla sıralanır:

```
void hizli(int a[], int b, int s) {
    if (b >= s)
        return;
    int p = ayir(a, b, s);
    hizli(a, b, p - 1);
    hizli(a, p + 1, s);
}
```

Örneği çalıştıran program, ilk ayırmada her yer değiştirmeden sonra diziyi, sonra pivotun yerini ve sıralı diziyi yazar:

Çıktı:

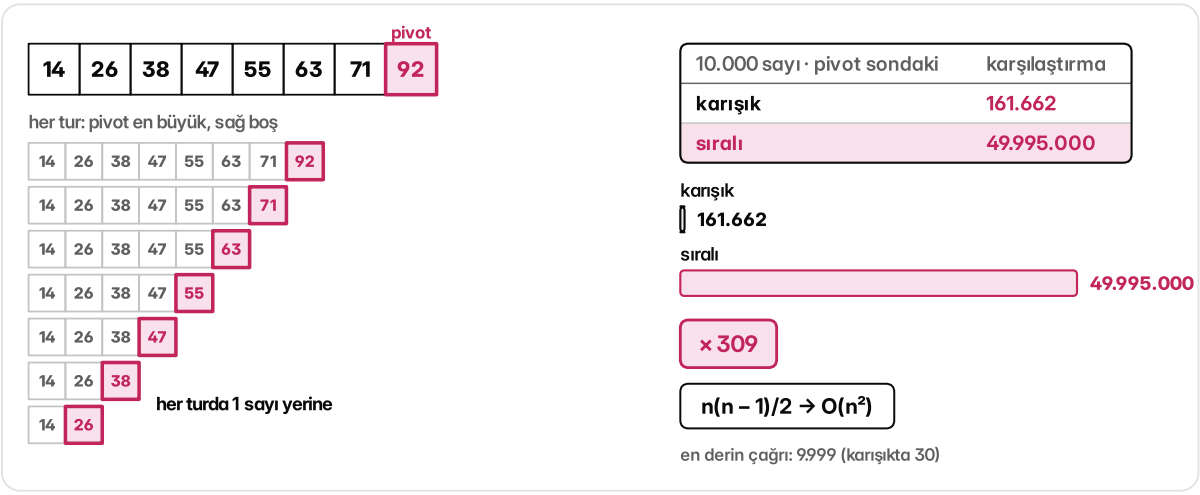
```
  j=0: 38 71 14 92 55 26 63 47
  j=2: 38 14 71 92 55 26 63 47
  j=5: 38 14 26 92 55 71 63 47
ilk ayirma, pivot yeri 3: 38 14 26 47 55 71 63 92
sirali: 14 26 38 47 55 63 71 92
```



Şekil 8.4 Pivot 47 sınırdaki 92 ile yer değiştirip yerine oturur: 38 14 26 47 55 71 63 92; solda küçükler, sağda büyükler, ek dizi yok.

Pivot çoğu zaman ortalardan çıktığı için dizi ikiye yakın bölünür: karışık 1.000 sayıda 11.000 civarı karşılaştırma (denemede 10.868), yani ortalamada O(n log n).

Ama dizi zaten sıralıysa sondaki sayı hep en büyüğüdür. Pivot hiçbir şeyi bölmez: her şey solunda kalır ve her turda yalnızca bir sayı yerine oturur. Karşılaştırmalar (n − 1) + (n − 2) + ... + 1 = n(n − 1)/2 eder. 10.000 karışık sayıda 161.662 olan karşılaştırma, sayılar sıralı gelince 49.995.000'e çıkar: 300 kattan fazla. Bu $O(n^2)$'dir; sıralı listeyi yeniden sıralamak sık yapılan bir iş olduğu için tuzak gerçektir.



Şekil 8.5 Sıralı dizide pivot hep en büyüktür ve her turda yalnız bir sayı yerine oturur; 10.000 sayıda 161.662 karşılaştırma 49.995.000'e çıkar.

Çözüm, pivotu dikkatle seçmektir. Birinci yol rastgele seçmek: şanssız pivotun her turda tekrarlanması neredeyse imkânsızdır; sıralı 10.000 sayı bu yolla 161.299 karşılaştırma tuttu. İkinci yol **üçün ortancası**: baştaki, ortadaki ve sondaki sayının ortancasını pivot yapmak. Sıralı dizide bu tam ortadaki sayıdır; dizi iki eşit yarıya bölünür ve 10.000 sıralı sayı 126.000 civarı karşılaştırmada sıralanır (ortanca seçimi dahil 125.916).

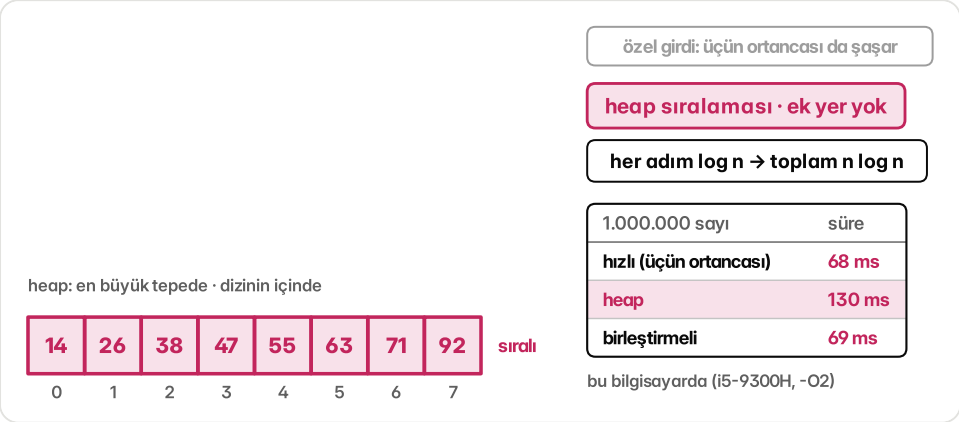
Yine de üçün ortancasını şaşırtan, özel hazırlanmış girdiler var. Hem hiç yavaşlamayan hem ek yer istemeyen bir yol daha var: **heap sıralaması** (heapsort). Heap, en büyük sayıyı hep tepede tutan, dizinin içine sığan bir ağaçtır. Dizi önce heap'e çevrilir. Tepedeki 92 en sona alınır, kalanlar

düzenlenir; yeni tepe 71 sondan ikinci yere geçer. Dizinin sağında sıralı bölge her adımda bir göz büyür. Program önce heap'i, sonra her adımı yazar:

Çıktı:

```
heap: 92 71 63 47 55 26 14 38
adim: 71 55 63 47 38 26 14 92
adim: 63 55 26 47 38 14 71 92
adim: 55 47 26 14 38 63 71 92
adim: 47 38 26 14 55 63 71 92
adim: 38 14 26 47 55 63 71 92
adim: 26 14 38 47 55 63 71 92
adim: 14 26 38 47 55 63 71 92
sirali: 14 26 38 47 55 63 71 92
```

Her adım log n sürer: hangi girdi gelirse gelsin toplam O(n log n). Ama heap sıralaması daha çok karşılaştırır ve dizide uzağa sıçrar. Bu bilgisayarda 1 milyon karışık sayıda hızlı sıralama (üçün ortancası) 68 ms, heap sıralaması 130 ms sürdü: 2 katına yakın.



Şekil 8.6 Heap sıralamasında en büyük her adımda sona alınır ve sağdaki sıralı bölge büyür; 1.000.000 sayıda hızlı, heap ve birleştirmeli sıralama 68, 130 ve 69 ms.

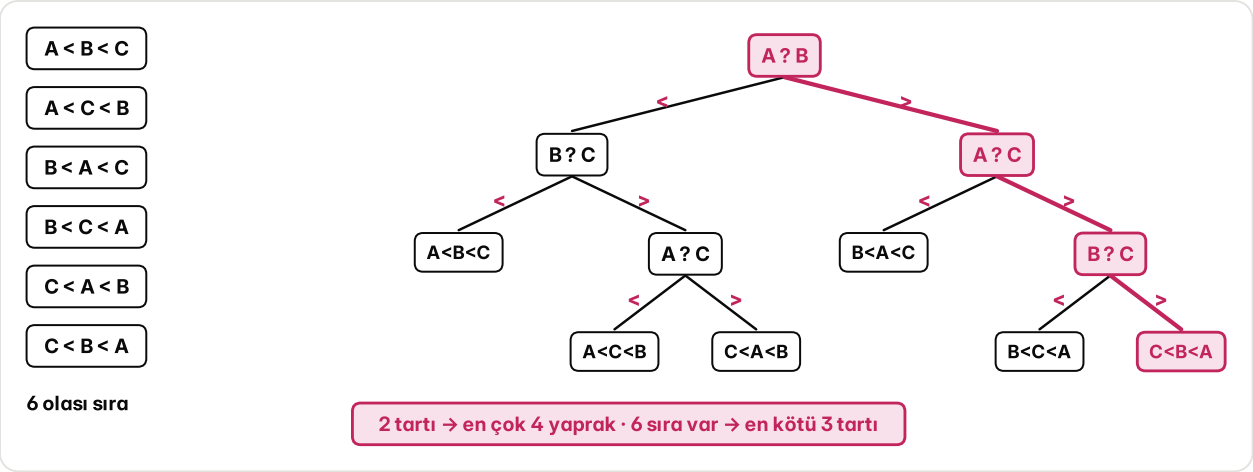
C++'ın hazır sıralama fonksiyonu bu yüzden, yaygın derleyicilerde, üçünü birleştirir. Hızlı sıralamayla başlar; bölme fazla derinleşirse pivotlar kötü gidiyor demektir, o parçayı heap sıralamasıyla bitirir. Küçük parçaları eklemeli sıralamaya bırakır. Adı **introsort**: hızlı sıralamanın ortalama hızı, heap sıralamasının garantisiyle.

Hızın sınırı

videoda 52:56

Bir sıralama algoritması neden belli bir hızın altına inemez, bazıları bu sınırı nasıl aşar? Önce sınırı bulalım: 3 paketi iki kefeli terazide tartarak hafiften ağıra dizmek en az kaç tartı ister?

3 paket 6 farklı sırada olabilir; tartılar doğrusunu bulmalı. Her tartının iki sonucu var: sol ağır ya da sağ ağır. Tartılar dallanan bir ağaç çizer; buna **karar ağacı** denir, her yaprağı bir sıralamadır. 2 tartıda en çok 4 yaprak olur, 6'ya yetmez: en kötü durumda 3 tartı şart.



Şekil 8.7 3 paketin 6 olası sırası; karar ağacında her tartı ikiye ayırır, 2 tartı en çok 4 yaprak verir, en kötü durumda 3 tartı gerekir.

Paket arttıkça olası sıralar patlar: n paket n! (1 · 2 · … · n) farklı sırada olabilir, 10 pakette 3.628.800. Her tartı olasılıkları en iyi ihtimalle yarıya indirir; 3.628.800'ü teke indirmek için 22 kez yarıya bölmek gerekir (2^{21} = 2.097.152 yetmez, 2^{22} = 4.194.304 yeter). Bilgisayarın karşılaştırmaları da birer tartıdır: 1.000 sayıda sınır $\log_2(1.000!) \approx 8.529{,}4$, yani 8.530 karşılaştırma; birleştirmeli sıralama en kötü durumda 8.977 ile sınırın hemen üstündedir. Genel hâli n log n'dir: yalnızca karşılaştırarak sıralayan hiçbir algoritma bunun altına inemez. Bu alt sınır $\Omega(n \log n)$ diye yazılır.

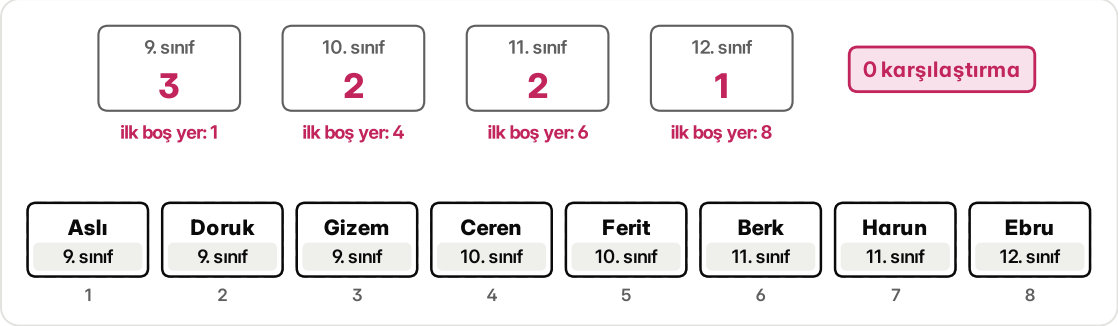
Sınırı aşmanın yolu karşılaştırmayı bırakmak. Bir lisede 8 öğrenciyi sınıfına göre dizelim: Aslı 9, Berk 11, Ceren 10, Doruk 9, Ebru 12, Ferit 10, Gizem 9, Harun 11. Sınıf yalnızca 4 değer alır. Öğrencileri karşılaştırmak yerine sayarız: 9. sınıftan 3, 10.'dan 2, 11.'den 2, 12.'den 1 öğrenci. Sayılar yerleri verir: 9'lar ilk 3 yere, sonra 10'lar, 11'ler, en sona 12; ilk boş yerler 1, 4, 6 ve 8. Sonra her öğrenci listedeki sırayla, sınıfının ilk boş yerine yazılır. Hiç karşılaştırma yok; buna **sayma sıralaması** (counting sort) denir. Kodda sınıf numarası doğrudan dizi gözüdür; yer 0'dan saydığı için ilk boş yerler 0, 3, 5 ve 7'dir:

```
void sayarak(kayit a[], kayit b[], int n) {
    int say[13] = {0}, yer[13] = {0};
    for (int i = 0; i < n; i++)
        say[a[i].sinif]++;
    for (int s = 10; s <= 12; s++)
        yer[s] = yer[s - 1] + say[s - 1];
    for (int i = 0; i < n; i++)
        b[yer[a[i].sinif]++] = a[i];
}
```

Çıktı:

```
say: 9->3 10->2 11->2 12->1
1. Aslı 9
2. Doruk 9
3. Gizem 9
4. Ceren 10
5. Ferit 10
6. Berk 11
7. Harun 11
8. Ebru 12
adim: 19
```

Aynı sınıftakiler listedeki sırasını korudu: sayma sıralaması kararlıdır.



Şekil 8.8 8 öğrenci sınıflarına göre sayılır (3, 2, 2, 1), ilk boş yerler 1, 4, 6, 8; hiç karşılaştırma yapılmaz.

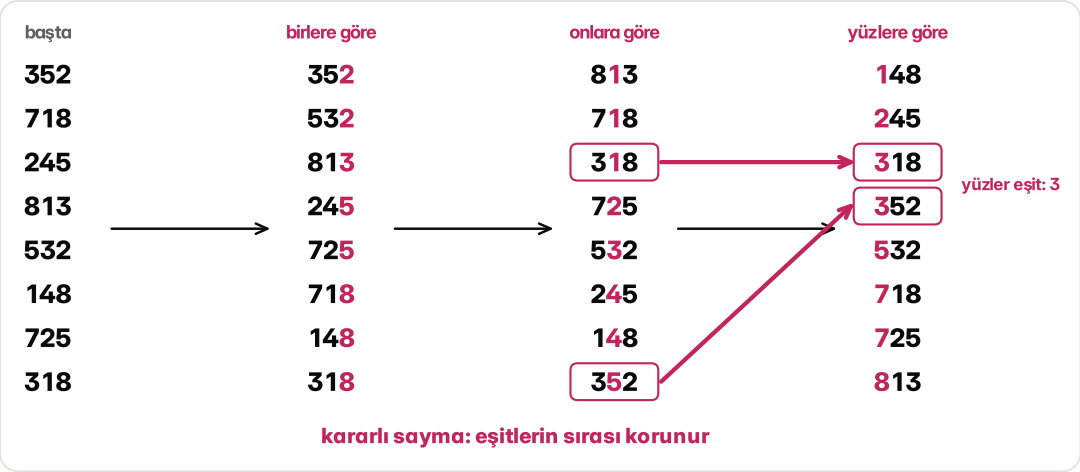
İş, bir kez saymak, 4 sınıfın yerini hesaplamak ve bir kez yerleştirmektir; değer sayısına k dersek O(n + k). Örnekte 8 + 3 + 8 = 19 adım. 2.000 öğrencide 4.000 adım civarı tutar; karşılaştırmanın sınırı ise 19.053'tür. Ama k büyürse yöntem işe yaramaz: sayılar 1 milyara kadar çıkabiliyorsa 1 milyar sayaç gerekir.

Taban sıralaması (radix sort) sayıyı basamaklarına ayırır: her basamak 10 değer alır, 10 sayaç yeter. 3 basamaklı 8 sayı (352 718 245 813 532 148 725 318) önce birler basamağına göre sayarak dizilir, sonra onlara, en son yüzlere göre. Program her turdan sonraki diziyi yazar:

Çıktı:

```
birler : 352 532 813 245 725 718 148 318
onlar  : 813 718 318 725 532 245 148 352
yuzler : 148 245 318 352 532 718 725 813
```

Son turda 318 ile 352 yüzler basamağında eşittir; aralarındaki sırayı önceki tur, onlar basamağı belirlemişti. Sayma sıralaması eşitlerin sırasını bozmadığı için bu sıra korunur ve dizi sıralanır.



Şekil 8.9 8 sayı birlere, onlara, yüzlere göre üç turda dizilir; yüzleri eşit 318 ile 352'nin sırasını önceki tur belirler.

Her tur n + 10 adımdır: 9 basamaklı sayılarda 9 tur, kaç sayı olursa olsun. Ekran kartında sayı sıralayan hazır kütüphaneler de bu yolu seçer; sayma işi binlerce çekirdeğe paylaştırılır. Bedeli, anahtarın basamaklara bölünebilmesi; karşılaştırmalı yöntemler ise her şeyi sıralar. Yani n log n sınırı karşılaştırmanın sınırıdır; sayının içine bakan bir algoritma onu aşabilir.

Yöntem	Ortalama	En kötü durum	Ek yer
Birleştirmeli	O(n log n)	O(n log n)	n göz, O(n)
Hızlı (pivot sondaki)	O(n log n)	$O(n^2)$	ek dizi yok
Heap sıralaması	O(n log n)	O(n log n)	ek dizi yok
Sayma	O(n + k)	O(n + k)	k sayaç ve n göz
Taban (d basamak)	O(d · (n + 10))	O(d · (n + 10))	10 sayaç ve n göz

Özet

- Birleştirmeli sıralama böler, yarıları sıralar ve birleştirir; her girdide O(n log n) sürer ama dizi kadar ek yer ister.
- Eşit anahtarlıların eski sırasını koruyan sıralama kararlıdır; birleştirmede eşitlikte soldakini almak bunu sağlar, qsort bu sözü vermez.
- Hızlı sıralama pivottan küçükleri dizinin içinde sola toplar; ortalamada O(n log n), sıralı dizide sondaki pivotla $O(n^2)$'dir. Rastgele pivot ya da üçün ortancası bu tuzağı önler.
- Heap sıralaması her girdide O(n log n) sürer ve ek dizi istemez, ama daha çok karşılaştırır; bu bilgisayarda hızlı sıralamanın 2 katına yakın sürdü. Introsort üç yöntemi birleştirir.
- Karşılaştırarak sıralayan hiçbir algoritma en kötü durumda $\log_2(n!)$, yani yaklaşık n log n karşılaştırmanın altına inemez.
- Sayma sıralaması O(n + k), taban sıralaması basamak başına bir sayma turuyla çalışır; karşılaştırma yapmadıkları için bu sınırı aşarlar.

Sorular

1 **Bir sıralama algoritmasına kararlı denmesi ne anlama gelir?**

A Her girdide aynı sayıda karşılaştırma yapar.

B Eşit anahtarlı kayıtların aralarındaki eski sırayı korur.

C Ek bellek kullanmadan, dizinin içinde sıralar.

D En kötü durumda bile O(n log n) sürede biter.

2 **birlestir fonksiyonunda if (strcmp(a[i].il, a[j].il) <= 0) satırı if (strcmp(a[i].il, a[j].il) < 0) yapılırsa ne olur?**

A Liste artık şehre göre sıralanmaz.

B Program derlenmez.

C Liste yine şehre göre sıralanır, ama eşitlikte sağdaki alındığı için aynı şehirdekilerin sırası bozulur.

D Hiçbir şey değişmez, çünkü eşit şehirler zaten yan yana gelir.

3 **Birleştirmeli sıralamanın bedeli nedir?**

A Sıralı girdide $O(n^2)$ sürer.

B Eşit elemanların sırasını bozar.

C Karşılaştırma sayısı girdiye göre çok değişir.

D Dizinin kendisi kadar ikinci bir diziye, yani O(n) ek belleğe ihtiyaç duyar.

4 **ayir fonksiyonu son elemanı pivot alıp küçükleri sola toplar. Aşağıdaki diziyle ayir(a, 0, 5) çağrılınca dizi ne olur? Elemanları aralarında boşlukla yaz.**

```
int a[6] = {30, 80, 10, 60, 20, 50};
```

Cevap: _____

5 **Hızlı sıralama her zaman sondaki elemanı pivot seçiyor. Dizi zaten küçükten büyüğe sıralıysa karşılaştırma sayısı nasıl büyür?**

A O(n log n)

B $O(n^2)$

C O(n)

D O(log n)

6 **5 paketi iki kefeli terazide tartarak sıralamak için en kötü durumda en az kaç tartı gerekir?**

Cevap: _____

7 **Sayma sıralaması hangi durumda işe yaramaz?**

A Eleman sayısı çok büyükse.

B Aynı değerden birden çok eleman varsa.

C Değerlerin alabileceği aralık çok büyükse, örneğin sayılar 1 milyara kadar çıkabiliyorsa.

D Elemanların eski sırası korunmak isteniyorsa.

8 **Taban (radix) sıralamasının ilk turu sayıları birler basamağına göre kararlı biçimde dizer. Aşağıdaki dizi ilk turdan sonra nasıl olur? Elemanları aralarında boşlukla yaz.**

```
431 207 325 118 540 213
```

Cevap: _____

VERİ YAPILARI

9 Graflar

Bu bölümde düğüm ve kenarlardan oluşan grafları, onları bellekte saklamanın iki yolunu ve graf üzerinde iki temel dolaşmayı göreceğiz: ağırlıksız grafta en kısa yolu bulan genişlik öncelikli aramayı ve ön koşullu işleri sıraya koyan derinlik öncelikli aramayı.

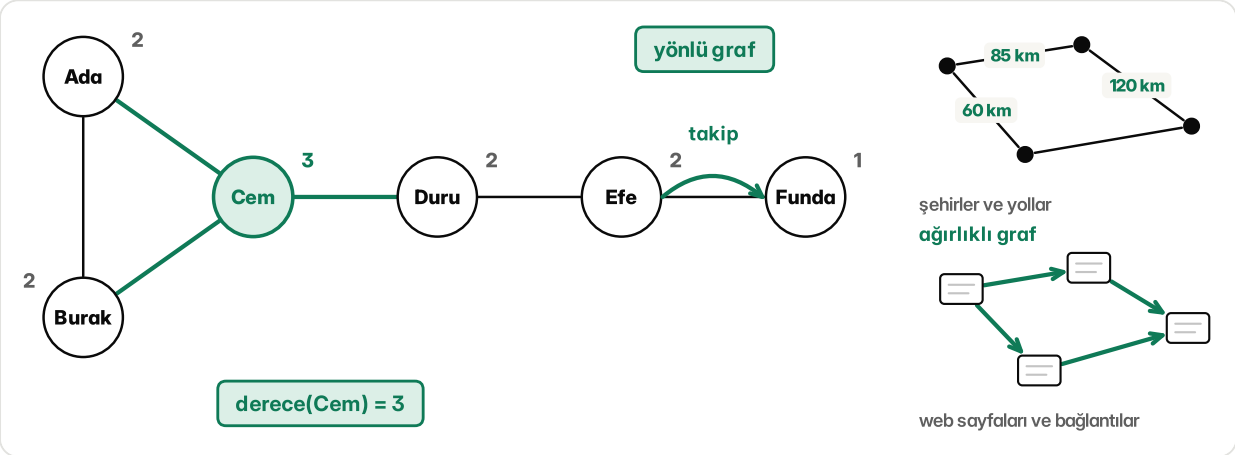
Kaç bayt?

videoda 55:26

2 milyon kullanıcılı bir uygulamada herkesin ortalama 100 arkadaşı var. Kimin kimle arkadaş olduğunu saklamak için ne kadar yer gerekir?

Önce 6 kişilik bir grup: Ada, Burak, Cem, Duru, Efe ve Funda. Ada, Burak ve Cem birbirinin arkadaşı; ayrıca Cem ile Duru, Duru ile Efe, Efe ile Funda arkadaş. Her kişi bir nokta, her arkadaşlık iki noktayı birleştiren bir çizgi olsun. Noktalara **düğüm** (vertex), çizgilere **kenar** (edge), bu yapının tamamına **graf** (graph) denir. Şehirler ve yollar da, web sayfaları ve bağlantılar da birer graftır.

Arkadaşlık karşılıklıdır, kenarın yönü yoktur. Takip etmek öyle değil: Efe Funda'yı takip eder, Funda Efe'yi etmeyebilir. Bu kenar tek yönlü bir oktur; böyle grafa **yönlü graf** denir. Web sayfalarındaki bağlantılar da tek yönlüdür. Yol haritasında kenarın üstünde kilometre yazar; bu da **ağırlıklı graf**. Bir düğüme bağlı kenar sayısı onun **derecesi**dir: Cem'in 3 arkadaşı var, derecesi 3.



Şekil 9.1 Efe'den Funda'ya tek yönlü takip oku, kilometreli yol haritası ve her kişinin derecesi: Cem'in derecesi 3.

Ada'dan Funda'ya arkadaştan arkadaşa bir zincir uzanır: Ada, Cem, Duru, Efe, Funda. Buna **yol** denir. Ada, Burak, Cem ve yeniden Ada: başladığı yere dönen yol bir **döngü**dür. Herkesten herkese bir yol varsa graf **bağlı**dır. Ada ile Burak küserse döngü kalmaz. Bağlı ve döngüsüz graf bir ağaçtır; 6 düğümlü her ağaçta tam 5 kenar, n düğümlü ağaçta n – 1 kenar vardır.

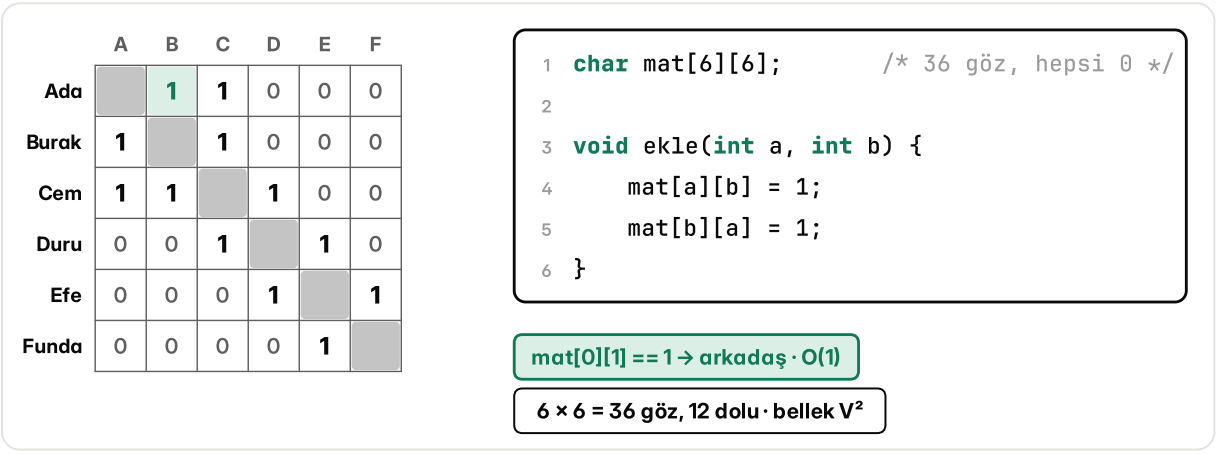
Grafı saklamanın ilk yolu 6 satır, 6 sütunluk bir tablodur. Kişileri 0'dan 5'e numaralayalım (Ada 0, Burak 1, ...). Ada ile Burak arkadaşsa Ada'nın satırında Burak'ın sütununa 1 yazılır, Burak'ın satırında Ada'nınkine de; öteki gözler 0 kalır. Buna **komşuluk matrisi** (adjacency matrix) denir:

```
char mat[6][6];       /* 36 göz, hepsi 0 */

void ekle(int a, int b) {
    mat[a][b] = 1;
    mat[b][a] = 1;
}
```

Çıktı:

```
matris:      A B C D E F
  Ada        0 1 1 0 0 0
  Burak      1 0 1 0 0 0
  Cem        1 1 0 1 0 0
  Duru       0 0 1 0 1 0
  Efe        0 0 0 1 0 1
  Funda      0 0 0 0 1 0
dolu goz: 12 / 36
Ada-Burak: 1  Cem-Funda: 0
```

İki kişi arkadaş mı sorusu tek bakıştır: mat[0][1] 1 ise Ada ile Burak arkadaştır; O(1). Ama tablo, düğüm sayısının karesi kadar göz tutar: 36 gözün yalnız 12'si dolu.

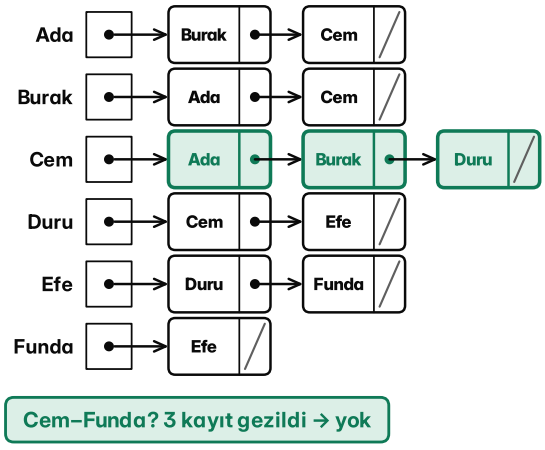


Şekil 9.2 6 × 6 komşuluk matrisi: her arkadaşlık iki göze 1 yazar, 36 gözün 12'si dolu; mat[0][1] tek bakışta cevap verir.

İkinci yolda her kişinin yanına yalnız arkadaşları yazılır. C'de bu, her kişiye bir bağlı liste demektir; yeni arkadaş listenin başına eklenir. Liste düğümü Dugum, arkadaşın numarasını (veri) ve sonraki düğümün adresini (sonraki) tutar:

```
Dugum *komsu[6];      /* başta hepsi NULL */

void yanina(int a, int b) {
    Dugum *d = malloc(sizeof(Dugum));
    if (d == NULL) exit(1);
    d->veri = b;
    d->sonraki = komsu[a];
    komsu[a] = d;
}
```

Arkadaşlık karşılıklı olduğu için iki listeye birden girer (yanina(a, b) ve yanina(b, a)): 6 kenar, 12 kayıt. Buna **komşuluk listesi** (adjacency list) denir. Yer, düğüm sayısı ile kenar sayısının toplamı kadardır. Bedeli de var: iki kişinin arkadaş olup olmadığını anlamak için listeyi gezmek gerekir. Cem ile Funda arkadaş mı diye sorunca Cem'in listesindeki 3 kayıt gezilir ve Funda bulunmaz.




```
1  Dugum *komsu[6];      /* başta hepsi NULL */
2
3  void yanina(int a, int b) {
4      Dugum *d = malloc(sizeof(Dugum));
5      if (d == NULL) exit(1);
6      d->veri = b;
7      d->sonraki = komsu[a];
8      komsu[a] = d;
9  }
```

Şekil 9.3 Her kişinin yanında arkadaşlarının bağlı listesi; 6 kenar 12 kayıt tutar, Cem–Funda sorusu için 3 kayıt gezilir.

Şimdi 2 milyon kullanıcı. Tablo 4 trilyon göz ister; her göze 1 bayt versek 4 terabayt. Üstelik 20.000 gözden yalnız biri dolu. Listede 100 milyon arkadaşlık 2 kez yazılır: 200 milyon kayıt; 64 bitlik bilgisayarda düğüm başına 16 baytla 3 gigabayt kadar. Bir kişinin arkadaşlarını bulmak için de tabloda 2 milyon gözlük satır taranır, listede 100 kayıt. Olası kenarların çoğu varsa graf **yoğun**dur ve tablo boşa gitmez. Ama yol haritası da arkadaşlık ağı da **seyrek**tir: bir kavşaktan yalnız birkaç yol çıkar. Büyük graflar bu yüzden çoğu zaman listeyle saklanır. V düğüm, E kenar sayısıysa:

	Komşuluk matrisi	Komşuluk listesi
Bellek	V^2	V + E
İki düğüm komşu mu?	O(1)	O(derece)
Bir düğümün komşularını gezmek	O(V)	O(derece)

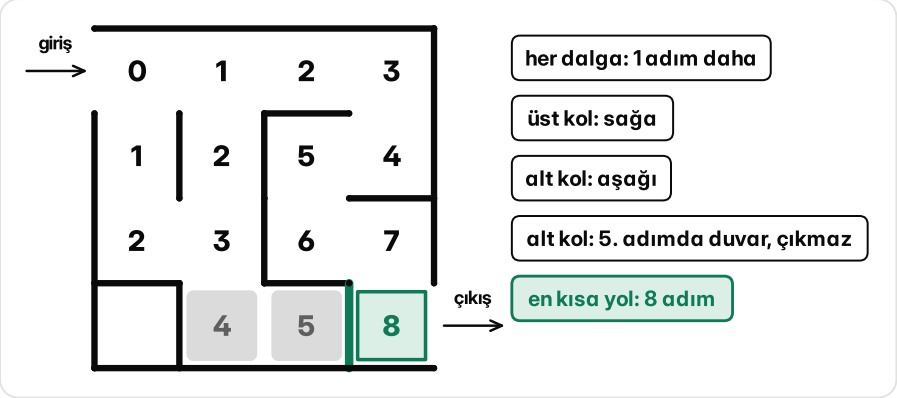
En kısa çıkış

videoda 57:49

16 karelik bir labirentte girişten çıkışa giden en kısa yol hangisi? Bilgisayar bütün yolları tek tek denemeden bunu nasıl bulur?

Labirent 4 × 4 karedir; giriş sol üstte, çıkış sağ altta. Her kare bir düğümdür. Aralarında duvar olmayan iki komşu kare bir kenarla bağlanır; duvar, kenar yok demektir. Bu labirentte 16 kare ve 15 geçit, yani 16 düğüm ve 15 kenar var. Bir kareden komşusuna geçmek 1 adım. Soru şuna döner: girişten çıkışa en az kaç kenar?

Girişe 0 yazalım. Ondan 1 adımda varılan 2 kareye 1 yazılır, onların numarasız komşularına 2, sonra 3. Numaralar, suya atılan taşın halkaları gibi girişten dışa yayılır. Dalga 2 kola ayrılır: biri üst kenardan sağa, öteki sol kenardan aşağı gider. Alttaki kol 5. adımda çıkışın hemen yanındaki kareye varır, ama arada duvar var; o kol orada biter. Üstteki kol dolanarak sürer ve 8. adımda çıkışa ulaşır. En kısa yol 8 adımdır.



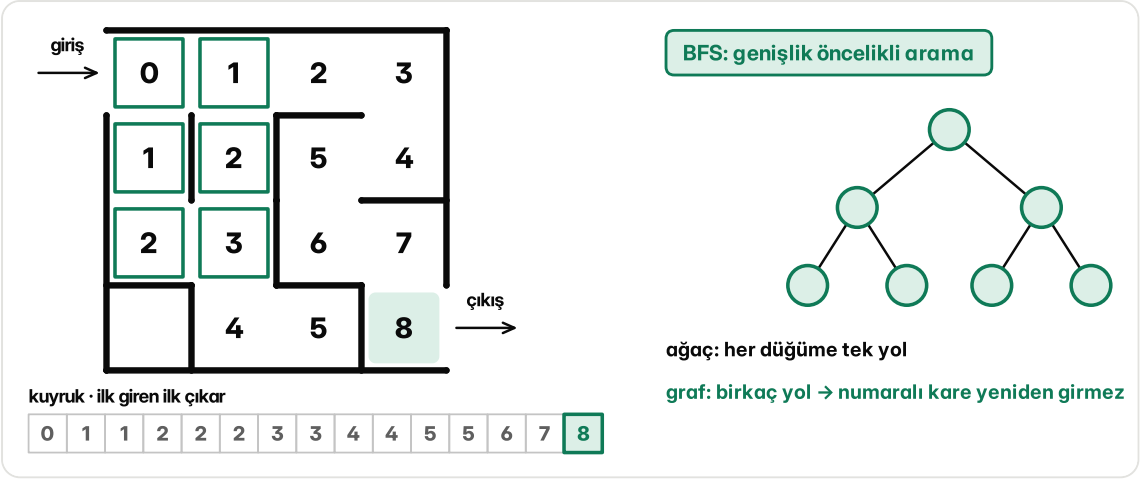
Şekil 9.4 Numaralar girişten dalga dalga yayılır; alt kol 5. adımda duvara takılır, üst kol 8. adımda çıkışa varır.

Dalgaların karışmamasını kuyruk sağlar: ilk gireni ilk çıkaran bir sıra. Önce giriş kuyruğa girer. Her adımda baştaki kare çıkar, numarasız komşularına kendi numarasının bir fazlasını yazar ve onları kuyruğun sonuna ekler. 1'ler 2'lerden önce girdiği için önce onlar çıkar; bir dalga bitmeden sonrakine geçilmez. Bu yüzden bir kareye ilk yazılan numara, oraya giden en kısa yolun adım sayısıdır; daha kısa bir yol olsaydı oraya daha önceki bir dalga varırdı. Bu aramaya **genişlik öncelikli arama** (breadth-first search), kısaca BFS denir.

Kodda kareler satır satır 0'dan 15'e numaralıdır; giriş 0, çıkış 15. uzak başta her karede −1'dir (numarasız); giriş, uzaklığı 0 yapılıp kuyruğa önceden konur. bas kuyruğun başını, son sonunu gösterir:

```
while (bas < son) {
    int u = kuyruk[bas++];
    Dugum *p = komsu[u];
    for (; p != NULL; p = p->sonraki)
        if (uzak[p->veri] == -1) {
            uzak[p->veri] = uzak[u] + 1;
            onceki[p->veri] = u;
            kuyruk[son++] = p->veri;
        }
}
```

BFS, bir ağacı kat kat gezmenin graftaki hâlidir. Tek fark, bir kareye birkaç yoldan varılabilmesidir; numara, karenin kuyruğa ikinci kez girmesini önler.



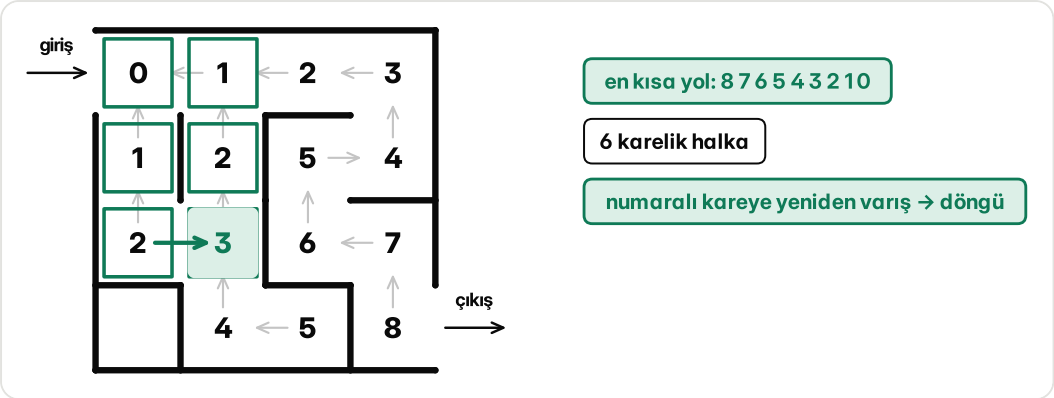
Şekil 9.5 Kuyruk ilk gireni ilk çıkarır, bu yüzden dalgalar sırayla işlenir; ağaçta her düğüme tek yol varken grafta numaralı kare yeniden girmez.

Numara alan her kare, numarayı hangi kareden aldığını da onceki dizisine not eder. Çıkıştan bu notlar geri izlenince numaralar 7, 6, 5 diye azalarak girişe iner: bu, en kısa yolun kendisidir. Sol üstteki 6 karelik halkada ise 2 dalga aynı kareye varır. Numaralı bir kareye yeniden varılması, iki ayrı yolun buluştuğunu, yani labirentte bir döngü olduğunu gösterir.

Sol alt köşedeki kare hiç numara almaz: dört yanı duvar, girişten ona yol yok. Bu graf bağlı değildir; o kare ayrı bir parçadır. Bütün parçaları bulmak için numarasız ilk kareden yeni bir arama başlatılır; kaç arama başlarsa graf o kadar parçadır. Program uzaklıkları, kuyruktan çıkış sırasını, geri izlenen yolu ve parçaları yazar:

Çıktı:

```
uzakliklar:
  0  1  2  3
  1  2  5  4
  2  3  6  7
  -  4  5  8
kuyruk sirasi: 0(0) 1(1) 4(1) 2(2) 5(2) 8(2) 3(3) 9(3) 7(4) 13(4) 6(5) 14(5) 10(6) 11(7)
15(8)
yol (cikistan geri): 15 11 10 6 7 3 2 1 0  -> 8 adim
ikinci varis: 8-9 (dongu)
gecide bakis 30 (2 x 15), numara alan 15 kare, agac kenari 14
yeni arama 12. kareden, parca 2
parca sayisi: 2
```



Şekil 9.6 Çıkıştan geri izlenen en kısa yol 8 7 6 5 4 3 2 1 0; sol üstteki 6 karelik halkada iki dalganın buluşması döngüyü gösterir.

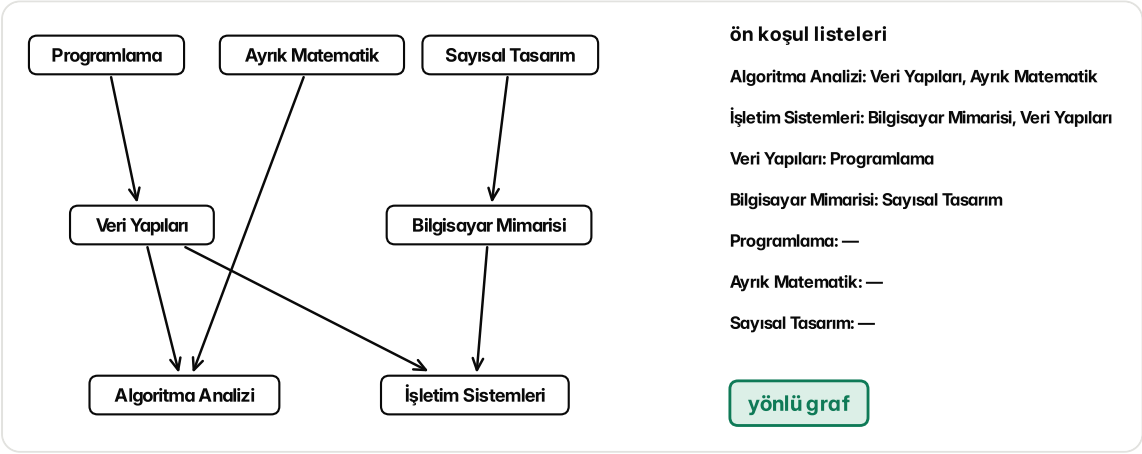
Her kare kuyruğa en çok bir kez girer, her geçide iki yönden birer kez bakılır: 15 geçit, 30 bakış. İş, kare sayısı ile geçit sayısının toplamı kadardır: O(V + E). Labirent 10 kat büyürse iş de 10 kat büyür. Yöntem, her adımın aynı uzunlukta olduğu, yani ağırlıksız her grafta en kısa yolu verir. Resim programlarındaki boya kovası da böyle yayılır: tıklanan noktadan aynı renkteki komşulara geçer, çizginin kapattığı bölgenin dışına taşmaz.

Hangi sırayla?

videoda 1:00:11

7 dersin ön koşulları belli: Veri Yapıları'ndan önce Programlama, İşletim Sistemleri'nden önce Bilgisayar Mimarisi alınmalı. Hiçbir ön koşulu çiğnemeyen bir sırayı bir program nasıl bulur?

Her ders bir düğüm; ön koşuldan derse giden ok bir kenar. Oklar tek yönlü, yani graf yönlü. Her dersin yanında ön koşullarının listesi durur. Algoritma Analizi, Veri Yapıları'nı ve Ayrık Matematik'i ister; İşletim Sistemleri, Bilgisayar Mimarisi'ni ve Veri Yapıları'nı. Veri Yapıları'nın ön koşulu Programlama, Bilgisayar Mimarisi'ninki Sayısal Tasarım. Programlama, Ayrık Matematik ve Sayısal Tasarım'ın ön koşulu yok.



Şekil 9.7 Yedi ders yönlü bir graf oluşturur, oklar ön koşuldan derse gider; yanda her dersin ön koşul listesi.

Kural şu: bir dersi yazmadan önce ön koşullarını yaz. Algoritma Analizi'nden başlayalım. İlk ön koşulu olan Veri Yapıları'na inilir, oradan Programlama'ya. Programlama'nın ön koşulu yok; listeye ilk o yazılır. Geri çıkılır; Veri Yapıları'nın başka ön koşulu yok, o da yazılır. Sonra Ayrık Matematik,

en sonda Algoritma Analizi. Gidebildiği kadar derine inen, gidecek yer kalmayınca bir adım geri çıkan bu aramaya **derinlik öncelikli arama** (depth-first search), kısaca DFS denir. Labirentte ip çözerek yürüyen biri de böyle yapar: koridorun sonuna gider, çıkmazda ipi sararak son kavşağa döner.

Kodda hal dizisi her dersin durumunu tutar (YENI, YOLDA ya da BITTI; başta hepsi YENI), onkosul[d] dersin ön koşul listesidir, yaz(d) dersi sıraya ekleyip ekrana yazar:

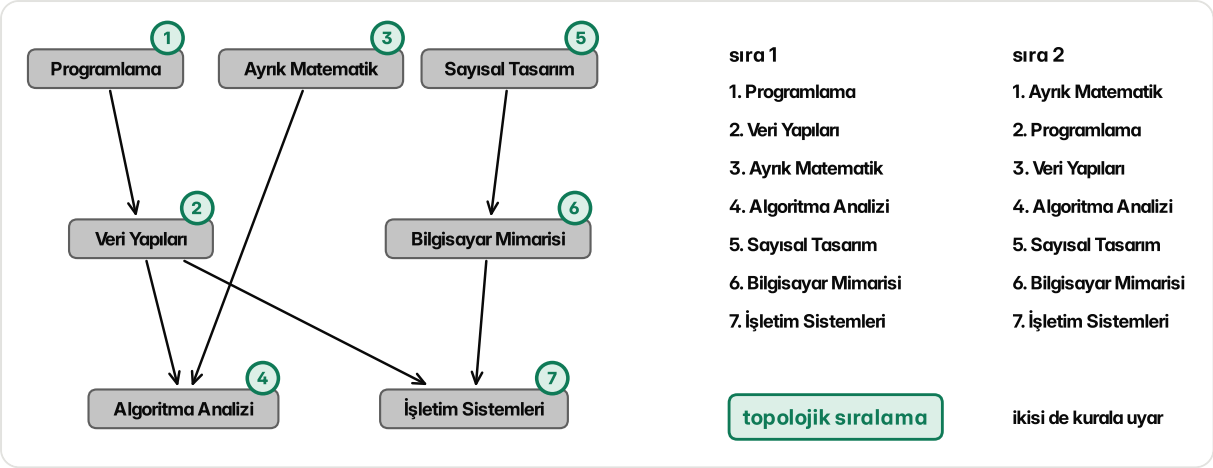
```
void dfs(int d) {
    if (hal[d] == YOLDA) dongu = 1;
    if (hal[d] != YENI) return;
    hal[d] = YOLDA;
    Dugum *p = onkosul[d];
    for (; p != NULL; p = p->sonraki)
        dfs(p->veri);
    hal[d] = BITTI;
    yaz(d);
}
```

Nereye geri dönüleceğini bir yığın (stack) tutar: son gireni ilk çıkaran bir sıra. Fonksiyon her ön koşul için kendini çağırır; çağrılar üst üste biner, en üstteki bitince bir alttakine dönülür. Programlama yazılırken 3 çağrı üst üste durur: dfs(Algoritma Analizi), dfs(Veri Yapıları) ve dfs(Programlama). Aynı arama, açık bir yığınla döngü olarak da yazılabilir.

Sıra İşletim Sistemleri'nde. Bilgisayar Mimarisi'ne, oradan Sayısal Tasarım'a inilir; önce Sayısal Tasarım, sonra Bilgisayar Mimarisi yazılır. Öteki ön koşul Veri Yapıları zaten bitti; bitmiş derse yeniden inilmez, if (hal[d] != YENI) return; satırı bunu sağlar. En sonda İşletim Sistemleri yazılır. 7 ders sıralandı; her biri kendi ön koşullarından sonra geliyor. Buna **topolojik sıralama** denir. Tek doğru sıra yoktur: Ayrık Matematik başa alınsa da kural bozulmaz. Programın çıktısının ilk satırları sırayı ve iki denetimi gösterir:

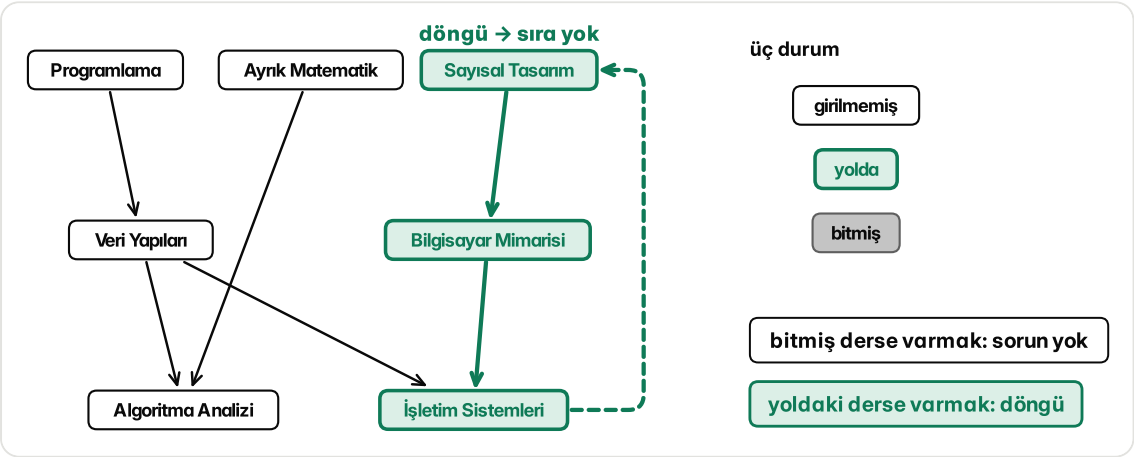
Çıktı:

```
sira (dfs):
  1. Programlama
  2. Veri Yapıları
  3. Ayrık Matematik
  4. Algoritma Analizi
  5. Sayısal Tasarım
  6. Bilgisayar Mimarisi
  7. İşletim Sistemleri
dongu: 0, kural tamam: 1
Ayrik basta: kural tamam: 1
```



Şekil 9.8 DFS'in bulduğu sıra ve Ayrık Matematik'in başa alındığı ikinci sıra; ikisi de topolojik sıralamadır.

Tabloya yanlış bir satır girsin: Sayısal Tasarım, İşletim Sistemleri'ni istesin. Arama İşletim Sistemleri'nden Bilgisayar Mimarisi'ne, oradan Sayısal Tasarım'a iner ve yine İşletim Sistemleri'ne varır. Ama o ders hâlâ yolda; kendi ön koşulları bitmeden ona geri dönüldü. 3 ders birbirini bekliyor: bu bir döngüdür ve hiçbir sıra kurulamaz. Bunu yakalamak için her ders 3 durumdan birinde tutulur: girilmemiş, yolda ya da bitmiş. Bitmiş derse varmak sorun değil; yoldaki derse varmak döngü demektir. dfs'in ilk satırı, if (hal[d] == YOLDA) dongu = 1;, tam bunu denetler.



Şekil 9.9 Yanlış satırla İşletim Sistemleri, Bilgisayar Mimarisi ve Sayısal Tasarım bir döngü kurar; üç durum: girilmemiş, yolda, bitmiş.

Her derse bir kez girilir, her oka bir kez bakılır: 7 ders, 6 ok. İş, ders sayısı ile ok sayısının toplamı kadardır: O(V + E). Program kurarken de aynı sıra gerekir: kurulum aracı önce programın ihtiyaç duyduğu paketleri, onlardan da önce onların ihtiyaç duyduklarını kurar.

	BFS	DFS
Sırayı tutan	kuyruk	yığın (çağrı yığını)
Bu bölümde bulduğu	ağırlıksız grafta en kısa yol, bağlı parçalar	topolojik sıra, yönlü grafta döngü
Maliyet	O(V + E)	O(V + E)

Özet

- Graf düğümlerden ve onları birleştiren kenarlardan oluşur; kenarlar yönlü ya da ağırlıklı olabilir, bir düğümün kenar sayısı derecesidir.
- Bağlı ve döngüsüz graf bir ağaçtır; n düğümlü ağaçta n – 1 kenar vardır.
- Komşuluk matrisi V^2 göz tutar ve komşuluk sorusunu O(1)'de cevaplar; komşuluk listesi V + E yer tutar ve seyrek büyük graflar için uygundur.
- Genişlik öncelikli arama (BFS) kuyrukla dalga dalga ilerler; ağırlıksız grafta bir düğüme ilk yazılan uzaklık en kısa yoldur.
- Derinlik öncelikli arama (DFS) gidebildiği kadar derine inip geri çıkar; ön koşul listelerinde bitiş sırası topolojik sıralamayı verir.
- Yoldaki bir düğüme yeniden varmak yönlü grafta döngü demektir; BFS de DFS de O(V + E) sürer.

Sorular

1 **10 düğümlü, bağlı ve döngüsüz bir grafta kaç kenar vardır?**

A 10
B 11
C 9
D 45

2 **1.000 düğümlü bir graf komşuluk matrisiyle saklanıyor. Tabloda kaç göz vardır?**

Cevap:

3 **Yönsüz bir grafta 8 kenar var ve graf komşuluk listesiyle saklanıyor. Listelerde toplam kaç kayıt bulunur?**

A 16
B 8
C 64
D 4

4 **Bir yol haritası grafında 1 milyon kavşak var ve her kavşaktan yalnız 3-4 yol çıkıyor. Bu grafı saklamak için hangisi daha uygundur, neden?**

A Komşuluk matrisi, çünkü iki kavşağın komşu olup olmadığına O(1)'de bakılır.
B Komşuluk listesi, çünkü graf seyrektir; matris 1 trilyon gözün neredeyse hepsini boş tutardı.
C Komşuluk matrisi, çünkü yol haritası yoğun bir graftır.
D İkisi aynı yeri tutar, fark yoktur.

5 **Genişlik öncelikli aramada (BFS) sıradaki düğümleri hangi yapı tutar ve neden?**

A Yığın, çünkü en son bulunan düğüme önce gidilmelidir.

B Sıralı dizi, çünkü düğümler numaraya göre işlenmelidir.

C Heap, çünkü en küçük numaralı düğüm her an tepede olmalıdır.

D Kuyruk, çünkü ilk giren ilk çıktığı için bir dalga bitmeden sonrakine geçilmez.

6 **Yönsüz bir grafın kenarları: A-B, A-C, B-D, C-D, D-E. A'dan başlayan genişlik öncelikli arama E'ye kaç uzaklık yazar?**

Cevap: _____

7 **Ön koşul grafında derinlik öncelikli arama yaparken hangi durum bir döngü olduğunu gösterir?**

A Bitmiş bir derse yeniden varmak.

B Henüz girilmemiş bir derse varmak.

C Hâlâ yolda olan, yani ön koşulları bitmemiş bir derse yeniden varmak.

D Ön koşulu olmayan bir derse varmak.

8 **Ön koşul listeleri: D'nin ön koşulları B ve C (bu sırayla), B'nin ön koşulu A, C'nin ön koşulu A; A'nın ön koşulu yok. Bölümdeki dfs fonksiyonu yalnız dfs(D) ile çağrılınca dersler hangi sırayla yazılır? Aralarında boşlukla yaz.**

Cevap: _____

VERİ YAPILARI

10 Minimum yayılan ağaç

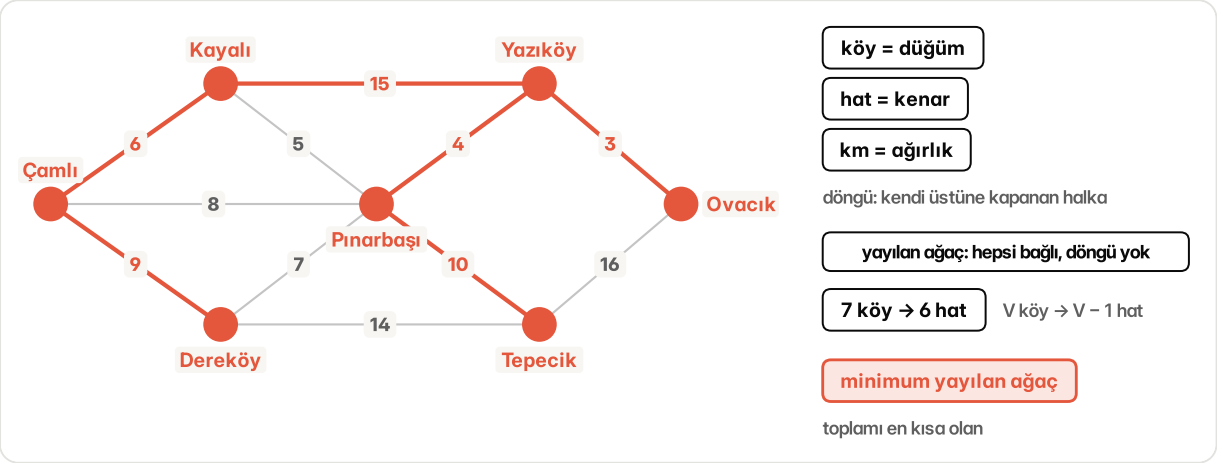
Bu bölümde bir grafın bütün düğümlerini en kısa toplam kenarla bağlamayı göreceğiz: Prim'in yöntemini ve onu doğru kılan kesit özelliğini, sonra Kruskal'ın yöntemini ve onun "zaten bağlı mı?" sorusunu hızla cevaplayan ayrık kümeleri.

Yirmi köy

videoda 1:02:34

Yirmi köy elektrik hattıyla birbirine bağlanacak. Her köyden her köye hat çekilebilir ve her hattın kaç kilometre tuttuğu belli. Bütün köylere elektriği en kısa toplam hatla götürmenin yolu hangisi?

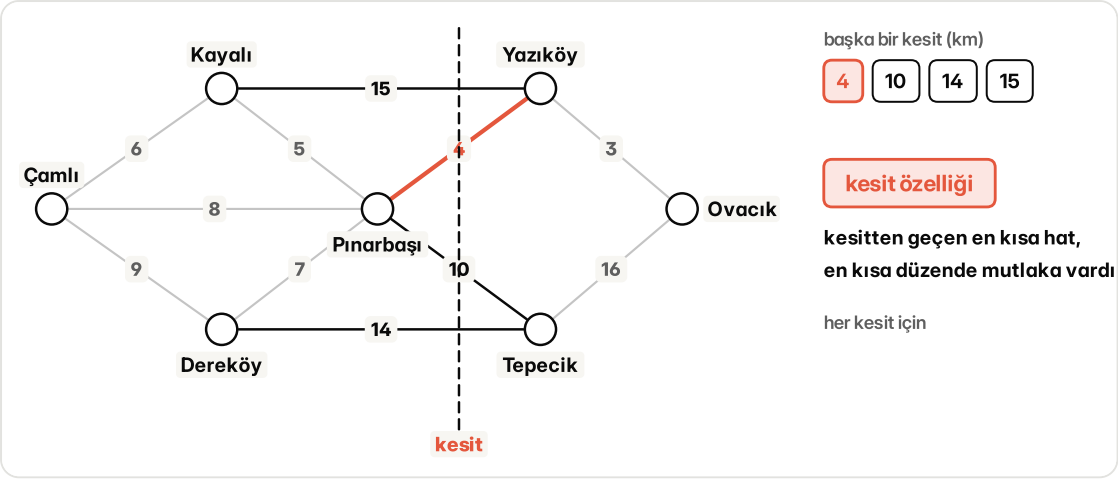
Küçük bir örnek: 7 köy, aralarında 11 olası hat. Bu bir graftır: köyler düğüm, hatlar kenar, her hattın kilometresi de o kenarın ağırlığı. Köyler tek parça olmalı, ama döngüye, yani kendi üstüne kapanan bir hat halkasına gerek yok: halkanın bir hattı sökülse de elektrik halkadaki bütün köylere ulaşır. Bütün köyleri bağlayan, döngüsü olmayan düzene **yayılan ağaç** denir. İlk köyden sonra her hat ağaca bir köy daha kattığı için V köyde tam V − 1 hat vardır; 7 köyde 6 hat. Toplamı en kısa olana **minimum yayılan ağaç** (minimum spanning tree) denir.



Şekil 10.1 Yedi köy ve on bir hat: köy düğüm, hat kenar, km ağırlık; yayılan ağaçta hepsi bağlı, döngü yok ve 7 köy 6 hatla bağlanır.

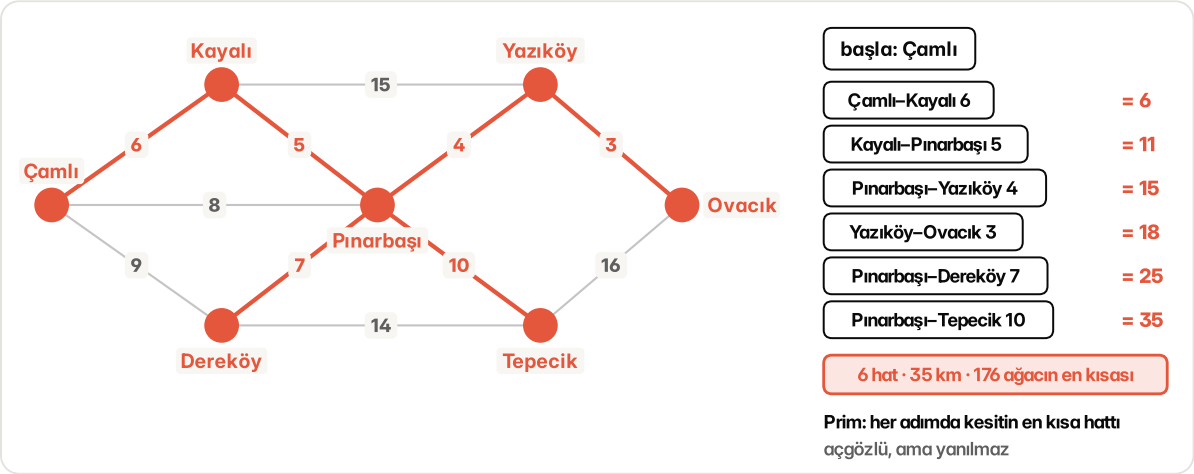
Bu 7 köyde bile 176 ayrı yayılan ağaç var. 20 köyde ağaç sayısı 20^{18}, 24 basamaklı bir sayı; saniyede 1 milyar ağaç deneyen bir bilgisayar 8 milyon yıldan fazla uğraşır. Denemek yerine bir kural gerekir.

Köyleri iki gruba ayıralım: batıda Çamlı, Kayalı ve Dereköy, doğuda kalan 4 köy. Aradaki çizgiye **kesit** denir. Kesitten 5 hat geçiyor; en kısası Kayalı–Pınarbaşı arası, 5 km. Bir düzen iki tarafı yalnız 15 km'lik Kayalı–Yazıköy hattıyla bağlamış olsun. O hattı söküp 5 km'lik hattı çekersek iki taraf yine bağlı kalır, düzen 10 km kısalır. Gruplar nasıl seçilirse seçilsin, kesitten geçen en kısa hat, en kısa düzende mutlaka vardır. Buna **kesit özelliği** (cut property) denir.



Şekil 10.2 Kesik çizgi köyleri iki gruba ayırır; kesitten geçen en kısa hat en kısa düzende vardır ve bu, her kesit için geçerlidir.

Bu özellik yöntemi de verir. Çamlı'dan başlayalım: bir yanda elektrik alan köyler, öbür yanda geri kalanlar. Aradaki en kısa hattı çekeriz; ucundaki köy gruba katılır, soru yeniden sorulur. Önce 6 km'lik hatla Kayalı katılır; sonra Pınarbaşı 5, Yazıköy 4, Ovacık 3, Dereköy 7 ve Tepecik 10 km'yle. 6 hat, 35 km: 176 ağacın en kısası. Buna **Prim'in yöntemi** denir. Her adımda o an en kısa görüneni seçen bu açgözlü yol yanılmaz, çünkü her seçim bir kesitten geçen en kısa hattır.



Şekil 10.3 Prim Çamlı'dan başlar ve her adımda kesitin en kısa hattını çeker: 6, 5, 4, 3, 7, 10; toplam 35 km.

Her adımda bütün hatlara bakmamak için Prim, gruba katılan köyün hatlarını bir **öncelikli kuyruğa** (priority queue) atar; kuyruk en kısa hattı hep önde tutar. Kodda komsu[u], u köyünün hatlarını tutan bağlı listedir (Kenar: hedef, agirlik). Kuyruğun her kaydı (Oge) bir uzunluk ve bir köydür; ekle kayıt koyar, kok_al en kısasını çıkarır. Başta Çamlı 0 uzunlukla kuyruğa konur:

```
while (n > 0) {
    Oge o = kok_al();
    if (agacta[o.dugum]) continue;
    agacta[o.dugum] = 1;
    toplam += o.agirlik;
    Kenar *k = komsu[o.dugum];
    for (; k != NULL; k = k->sonraki)
        if (!agacta[k->hedef])
            ekle(k->hedef, k->agirlik);
}
```

Kuyruktan çıkan hattın ucundaki köy çoktan ağaçtaysa hat atılır, çünkü döngü kurar. Programın çıktısının son 11 satırı:

```
al: Yazıköy-Ovacık 3 km, toplam 18
   kuyruğa: Ovacık-Tepecik 16
al: Pınarbaşı-Dereköy 7 km, toplam 25
   kuyruğa: Dereköy-Tepecik 14
at: Çamlı-Pınarbaşı 8 km (Pınarbaşı zaten ağaçta)
at: Çamlı-Dereköy 9 km (Dereköy zaten ağaçta)
al: Pınarbaşı-Tepecik 10 km, toplam 35
at: Dereköy-Tepecik 14 km (Tepecik zaten ağaçta)
at: Kayalı-Yazıköy 15 km (Yazıköy zaten ağaçta)
at: Ovacık-Tepecik 16 km (Tepecik zaten ağaçta)
toplam 35 km, kuyruğa giren 12 kayıt, atılan 5
```

Her hat kuyruğa en çok bir kez girip çıkar; her giriş ya da çıkış log E kadar adım tutar. Toplam maliyet O(E log E); E burada hat sayısı. 20 köyün 190 hattında Prim, 100 rastgele köy düzeninde ortalama 3.577 adım sürdü; 8 milyon yıl yerine.

Bu soru gerçek bir elektrik şirketinin sorusuydu: 1926'da Çek matematikçi Otakar Borůvka, Moravya'ya en ucuz şebekeyi kurmak isteyen şirket için ilk çözümü yayımladı.

Zaten bağlı mı?

videoda 1:05:12

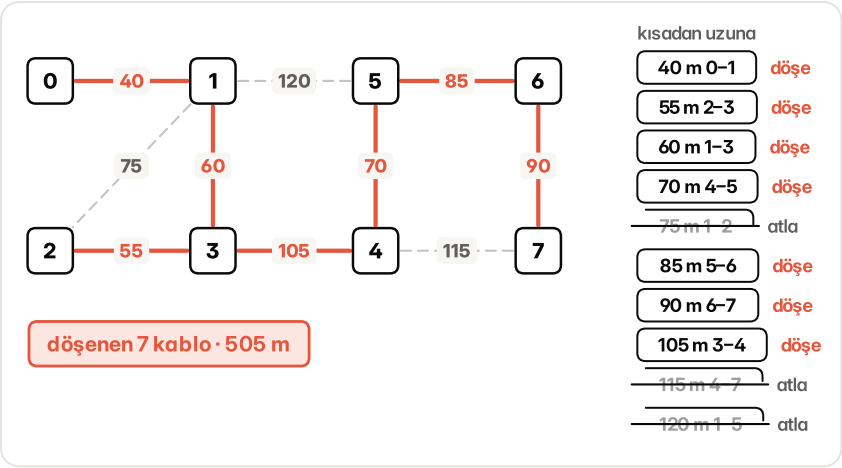
Bir ilçede 30.000 bina fiber kabloyla bağlanacak; her bina yalnız yanındaki binalara kablo çekebilir, 59.650 olası kablo var. Kablolar kısadan uzuna döşenecek, ama her kablodan önce bir soru var: bu iki bina zaten bağlı mı? Bunu her seferinde baştan aramadan nasıl anlarız?

Kruskal'ın yöntemi şöyledir:

- Bütün kabloları kısadan uzuna sırala ve sırayla bak.
- Kablonun iki ucu ayrı parçalardaysa kabloyu döşe.
- İki ucu aynı parçadaysa atla: kablo döngü kurar ve boşa gider.

Neden doğru? Daha kısa kabloların hepsine çoktan bakıldı; bir parçadan dışarı çıkan daha kısa kablo olsaydı, o çoktan döşenirdi. Yani döşenen her kablo, parçasını dışarıya bağlayan en kısa kablodur; iki grubu bağlayan en kısa kablo da en kısa düzende mutlaka vardır.

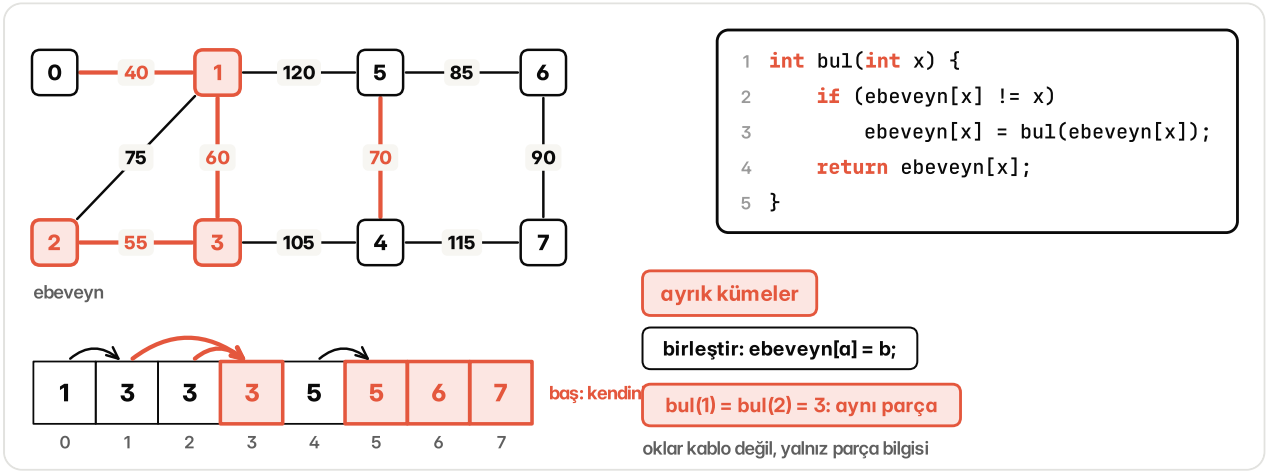
Sekiz binayla bakalım: 0'dan 7'ye numaralı, aralarında 10 olası kablo. 0–1 (40 m), 2–3 (55), 1–3 (60) ve 4–5 (70) döşenir. Sıradaki 1–2 (75); ama 1, 3 üzerinden 2'ye zaten bağlı: atlanır. Sonra 5–6, 6–7 ve 3–4 döşenir. 7 kablo, 505 metre; kalan 2 kablo da atlanır.



Şekil 10.4 Sekiz bina ve on kablo kısadan uzuna sıralı: yedisi döşenir, 75, 115 ve 120 metrelik kablolar döngü kuracağı için atlanır; toplam 505 m.

Zor kısım, "zaten bağlı mı?" sorusu. Baştan aramak, döşenmiş kablolar üstünde bir binadan ötekine yol aramaktır ve parça büyüdükçe pahalanır: 30.000 binada kablo başına ortalama 6.298 bina gezildi, toplam 375.703.370.

Daha iyisi, parçaları bir dizide tutmaktır: **ayrık kümeler** (union-find). Her binanın ebeveyn dizisinde bir gözü vardır; başta her bina kendini gösterir, kendi parçasının başıdır. **Bul** okları izler ve kendini gösteren binada, parçanın başında durur; iki binanın başı aynıysa ikisi aynı parçadadır. **Birleştir** bir başı ötekine bağlar: ebeveyn[a] = b;. 1–2 sorulduğunda dizi 1 3 3 3 5 5 6 7'dir: 1'den 3'e gidilir, 2'den de 3'e; aynı parça. Bu oklar kablo değildir, yalnız kimin hangi parçada olduğunu tutar.

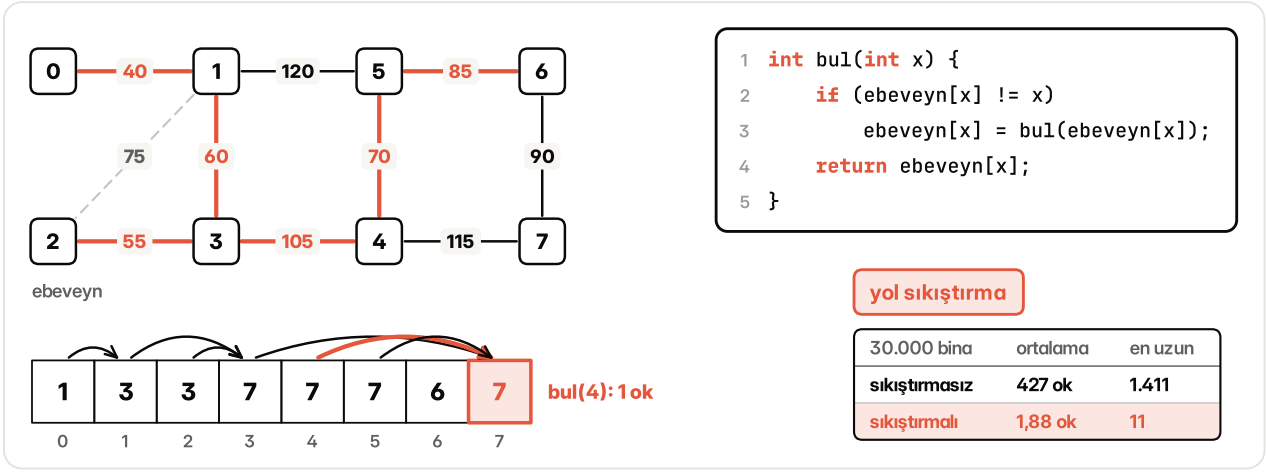


Şekil 10.5 Ebeveyn dizisi 1 3 3 3 5 5 6 7: bul(1) ve bul(2) aynı başa, 3'e varır, yani 1 ile 2 aynı parçadadır.

Oklar uzun zincir kurabilir. 3–4 sorulduğunda dizi 1 3 3 3 5 6 7 7'dir; 4'ten 5'e, 5'ten 6'ya, 6'dan 7'ye 3 ok izlenir. Bul, dönüşte geçtiği her binayı doğrudan başa bağlar: 4 de 5 de artık 7'yi gösterir, bir dahaki soru tek ok. Buna **yol sıkıştırma** (path compression) denir:

```
int bul(int x) {
    if (ebeveyn[x] != x)
        ebeveyn[x] = bul(ebeveyn[x]);
    return ebeveyn[x];
}
```

Kablo döşenince ebeveyn[3] = 7 olur ve dizi 1 3 3 7 7 7 7 7'ye döner. 30.000 binada sıkıştırma olmadan bir soru ortalama 427 ok izledi, en uzun zincir 1.411 ok; sıkıştırmayla ortalama 1,88 ok.



Şekil 10.6 Yol sıkıştırmadan sonra 4 ve 5 doğrudan 7'yi gösterir; 30.000 binada ortalama 427 ok 1,88'e iner.

Kruskal'ın kodunda hat dizisi kabloları tutar (bas, son, metre); qsort C'nin standart sıralama fonksiyonudur, kisa_once iki kabloyu uzunluğa göre karşılaştırır:

```
void kruskal(void) {
    qsort(hat, M, sizeof hat[0], kisa_once);
    for (int i = 0; i < M; i++) {
        int a = bul(hat[i].bas);
        int b = bul(hat[i].son);
        if (a == b) continue;
        ebeveyn[a] = b;
        toplam += hat[i].metre;
    }
}
```

İş (30.000 bina, 59.650 kablo)	Ölçülen
Baştan aramak	375.703.370 bina ziyareti
Ayrık kümeler	224.662 ok
Sıralama (qsort)	1.079.524 karşılaştırma

En pahalı iş artık baştaki sıralamadır. Kruskal'ın maliyeti de bu yüzden sıralamanınki kadardır: O(E log E).

Özet

- Yayılan ağaç bütün düğümleri döngüsüz bağlar ve V düğümde V − 1 kenar tutar; toplamı en kısa olanı minimum yayılan ağaçtır.
- Kesit özelliği: iki grubu ayıran kesitten geçen en kısa kenar, en kısa düzende mutlaka vardır.
- Prim ağacı bir düğümden büyütür, her adımda kesitin en kısa kenarını öncelikli kuyruktan alır: O(E log E).
- Kruskal kenarları kısadan uzuna sıralar, iki ucu ayrı parçadaysa ekler, aynı parçadaysa atlar.
- Ayrık kümelerde bul başa gider, birleştir bir başı ötekine bağlar, yol sıkıştırma zinciri kısaltır; Kruskal'da en pahalı iş sıralamadır.

Sorular

1 **12 köyü bağlayan bir yayılan ağaçta kaç hat vardır?**

A 12
B 11
C 13
D 66

2 **Yedi köylük haritada Dereköy'ün hatları şunlardır: Çamlı–Dereköy 9 km, Dereköy–Pınarbaşı 7 km, Dereköy–Tepecik 14 km. Dereköy bir yanda, kalan altı köy öbür yanda kalacak biçimde bir kesit çizilirse kesit özelliği hangi hattın en kısa düzende mutlaka bulunduğunu söyler?**

A Dereköy–Pınarbaşı 7 km
B Çamlı–Dereköy 9 km
C Dereköy–Tepecik 14 km
D Hiçbirini, çünkü kesitin bir yanında tek köy var.

3 **Yedi köylük haritanın hatları (km): Çamlı–Kayalı 6, Çamlı–Dereköy 9, Çamlı–Pınarbaşı 8, Kayalı–Pınarbaşı 5, Dereköy–Pınarbaşı 7, Kayalı–Yazıköy 15, Pınarbaşı–Yazıköy 4, Pınarbaşı–Tepecik 10, Dereköy–Tepecik 14, Yazıköy–Ovacık 3, Ovacık–Tepecik 16. Prim Tepecik'ten başlarsa ağaca giren ilk üç hattın uzunluklarını sırayla yaz.**

Cevap:

4 **Kuyruklu Prim'de kuyruktan çıkan kaydın köyü zaten ağaçtaysa ne yapılır?**

A Kayıt ağaca eklenir ve toplam güncellenir.

B Kuyruk baştan kurulur.

C Kayıt atılır, çünkü o hat bir döngü kurar.

D Köy ağaçtan çıkarılıp yeniden eklenir.

5 **Öncelikli kuyruk kullanan Prim'in maliyeti nedir? (E: hat sayısı)**

A O(E)

B $O(E^2)$

C O(2^E)

D O(E log E)

6 **Kruskal sekiz binada kabloları kısadan uzuna işlerken 1–2 kablosunu (75 m) neden atlar?**

A 75 m, o ana kadar döşenen kabloların ortalamasından uzundur.

B 1 ile 2, 3 üzerinden zaten aynı parçadadır; kablo döngü kurar.

C 2 numaralı bina henüz hiçbir kabloya bağlanmamıştır.

D Kruskal her parçaya en çok iki kablo döşer.

7 **0'dan 5'e numaralı altı binanın ebeveyn dizisi şöyledir: 1 2 5 3 3 5. Yol sıkıştırmalı bul(0) çağrıldıktan sonra dizi ne olur? Altı sayıyı sırayla yaz.**

Cevap: _____

8 **30.000 binalık ilçede yol sıkıştırmalı ayrık kümelerle çalışan Kruskal'ın en pahalı işi hangisidir?**

A Bul çağrılarının izlediği oklar

B Ebeveyn dizisinin ilk değerlerini yazmak

C Kabloların baştaki sıralaması

D Döşenen kabloların toplamını tutmak

VERİ YAPILARI

11 En kısa yol

Bu bölümde ağırlıklı bir grafta bir düğümden ötekine, ağırlıkları toplamı en küçük yolu bulmayı göreceğiz: Dijkstra'nın yöntemini, onu milyonlarca kavşaklı haritalarda hızlandıran öncelikli kuyruğu ve A*'ı, sonra eksi ağırlıklı yollarda da doğruyu bulan Bellman-Ford'u.

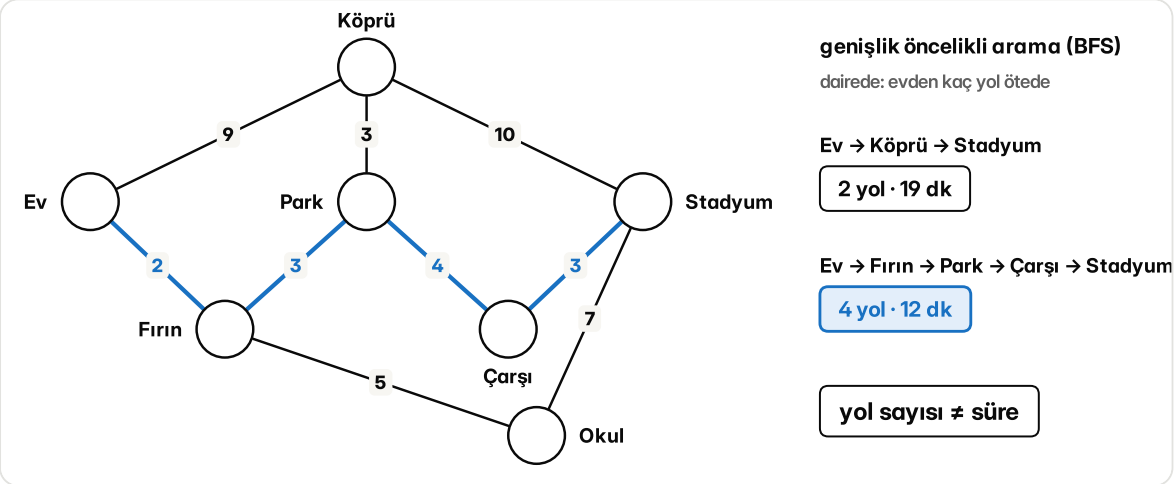
Navigasyon nasıl bulur?

videoda 1:07:47

Navigasyona evden stadyuma yol sorduğumuzda bir saniye geçmeden en kısa yolu çizer. Şehirde binlerce kavşak ve yol varken en kısasını nasıl bulur?

Harita bir graftır: kavşaklar düğüm, yollar kenar. Her yolun üstüne o yoldan geçmenin kaç dakika sürdüğünü yazarız. Bu sayıya kenarın ağırlığı, böyle bir grafa da **ağırlıklı graf** denir. Aradığımız, ağırlıklarının toplamı en küçük olan yoldur. Aynı soru internette de var: bir operatörün ağı içinde paket, yönlendiriciden yönlendiriciye en kısa yolu izler.

Örnek şehrimizde 7 kavşak ve 9 yol var (dakika): Ev–Köprü 9, Köprü–Stadyum 10, Ev–Fırın 2, Fırın–Park 3, Park–Çarşı 4, Çarşı–Stadyum 3, Fırın–Okul 5, Okul–Stadyum 7, Park–Köprü 3. İlk akla gelen genişlik öncelikli arama (BFS): evden çıkıp önce bir yol ötedeki kavşaklara, sonra iki yol ötedekilere bakmak. Bu yöntem en az yollu güzergâhı bulur: Ev → Köprü → Stadyum, 2 yol, 19 dakika. Oysa Fırın, Park ve Çarşı üstünden giden güzergâh 4 yol tutar ama 12 dakika sürer. Yol saymak, bütün yollar aynı uzunluktayken en kısa yolu verir; burada her yolun süresi farklı.

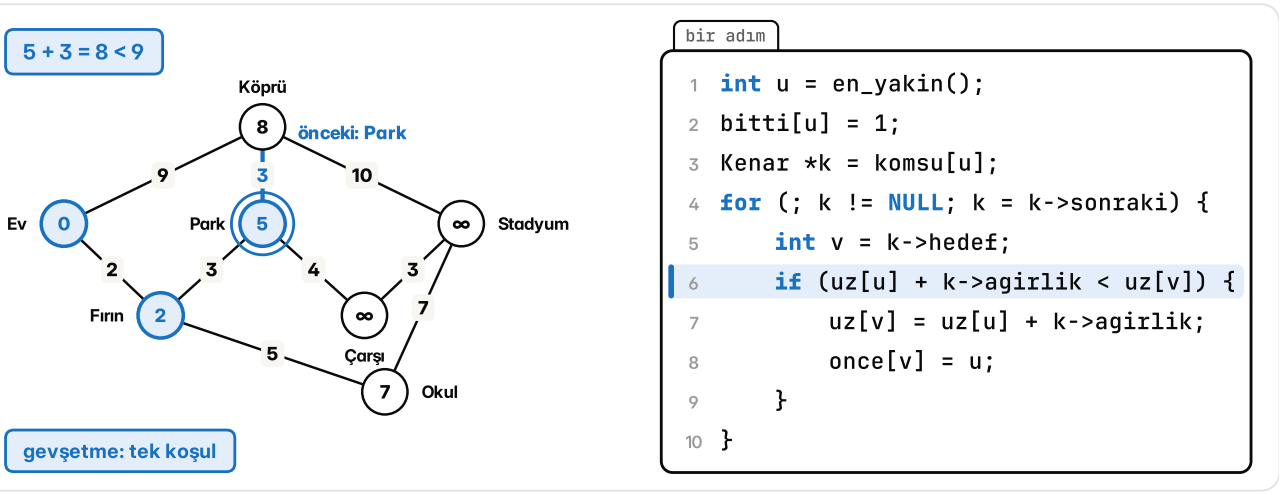


Şekil 11.1 Genişlik öncelikli arama 2 yollu, 19 dakikalık güzergâhı bulur; 4 yollu güzergâh 12 dakika sürer: yol sayısı süre değildir.

Dijkstra'nın yöntemi kavşakları yakından uzağa doğru, sırayla kesinleştirir. Her kavşağa evden şimdiye kadar bulunan en kısa süreyi yazarız: eve 0, ötekilere sonsuz. Her adımda, henüz kesinleşmemiş kavşaklar arasından süresi en küçük olanı seçeriz. Onun süresi artık kesindir: oraya başka bir yoldan gelmek, ondan daha uzaktaki bir kavşaktan geçmek demektir ve süreler eksi olmadığı için yol ancak uzar.

Sonra seçilen kavşaktan çıkan her yola bakarız. Komşuya bu kavşağın üstünden gitmek komşunun yazılı süresinden kısaysa yeni süreyi yazar, nereden geldiğimizi de not ederiz. Bu güncellemeye **gevşetme** denir; kodda tek bir koşul satırıdır. Graf, her kavşağın yollarını bir bağlı listede tutar: komsu[u], u'dan çıkan yolların listesidir ve her Kenar yolun öbür ucunu (hedef) ve süresini (agirlik) tutar. uz[] süreleri, once[] nereden gelindiğini, bitti[] kavşağın kesinleşip kesinleşmediğini tutar; en_yakin, kesinleşmemiş kavşaklar arasından süresi en küçüğü bütün kavşaklara bakarak bulur:

```
int adim(void) {
    int u = en_yakin();
    bitti[u] = 1;
    Kenar *k = komsu[u];
    for (; k != NULL; k = k->sonraki) {
        int v = k->hedef;
        if (uz[u] + k->agirlik < uz[v]) {
            uz[v] = uz[u] + k->agirlik;
            once[v] = u;
        }
    }
    return u;
}
```

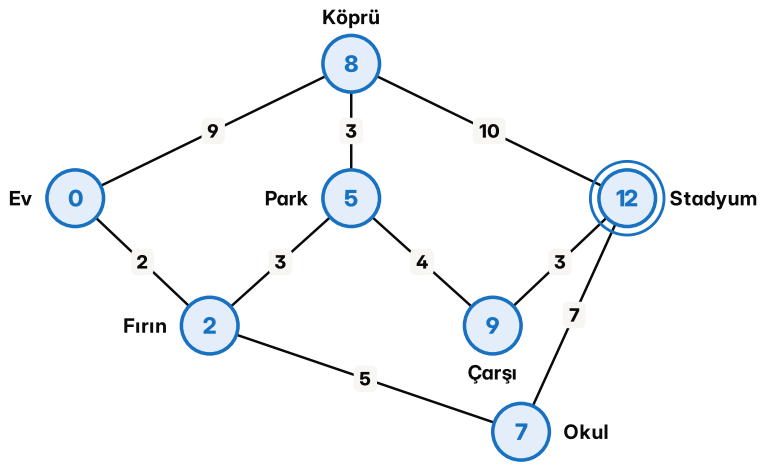


Şekil 11.2 Park 5'te kesinleşir; Köprü'ye Park üstünden 5 + 3 = 8 < 9 dakikada gidilir, Köprü'nün süresi 8 olur ve önceki kavşağı Park diye not edilir.

Adım adım bakalım. Ev kesinleşir: Fırın'a 2, Köprü'ye 9 yazılır. En küçük Fırın; kesinleşir, Park'a 5, Okul'a 7 gelir. Sonra Park: Çarşı'ya 9 yazılır, Köprü'nün 9'u 8'e iner, çünkü Park'ın üstünden daha kısa. Okul kesinleşince Stadyum'a 14 yazılır. Köprü 8'le kesinleşir ama Stadyum'a kısa yol getirmez: 8 + 10 = 18. Çarşı kesinleşince Stadyum'un 14'ü 12'ye iner. Sonunda Stadyum 12'yle kesinleşir. Program önce en az yollu güzergâhı, sonra her kesinleşmeyi ve değişen süreleri yazdırır:

Çıktı:

```
en az yol: Ev > Köprü > Stadyum · 2 yol · 19 dk
1. kesinleşti Ev 0 | Fırın ∞→2 Köprü ∞→9
2. kesinleşti Fırın 2 | Park ∞→5 Okul ∞→7
3. kesinleşti Park 5 | Köprü 9→8 Çarşı ∞→9
4. kesinleşti Okul 7 | Stadyum ∞→14
5. kesinleşti Köprü 8 |
6. kesinleşti Çarşı 9 | Stadyum 14→12
7. kesinleşti Stadyum 12 |
en kısa: Ev > Fırın > Park > Çarşı > Stadyum · 12 dk
```



sıra	kavşak	süre
1	Ev	0
2	Fırın	2
3	Park	5
4	Okul	7
5	Köprü	8
6	Çarşı	9
7	Stadyum	12

Stadyum: 14 → 12

Şekil 11.3 Yedi kesinleşme sırayla: Ev 0, Fırın 2, Park 5, Okul 7, Köprü 8, Çarşı 9, Stadyum 12.

Hedef kesinleşince arama biter; yol, nereden gelindiği notlarından geriye okunur: Stadyum'a Çarşı'dan, Çarşı'ya Park'tan, Park'a Fırın'dan, Fırın'a Ev'den. Bu basit hâlde en yakını seçmek için her adımda bütün kavşaklara bakarız: kavşak sayısı kadar adım, her seferinde kavşak sayısı kadar bakış, yani $O(V^2)$. 7 kavşakta 49 bakış, 10.000 kavşakta 100 milyon.

Kafede yirmi dakika

videoda 1:10:03

1956'da bir kafe terasında, kâğıt kalem kullanmadan 20 dakikada bulunan bir yöntem, bugün milyonlarca kavşaklı haritalarda nasıl çalışıyor?

Edsger W. Dijkstra o sırada Amsterdam'daki Matematik Merkezi'nde (Mathematisch Centrum) programcıydı. Yeni bilgisayar ARMAC'ın tanıtımı için, bilgisayar bilmeyenlerin de anlayacağı bir soru arıyordu: Rotterdam'dan kuzeydeki Groningen'e en kısa yol hangisi? Bir gün nişanlısıyla alışverişten yorulup bir kafe terasına oturdular; kahvesini içerken yöntemi kafasında kurdu. Sonra sadeleştirilmiş bir Hollanda haritasıyla bilgisayarda çalıştırdı. Yöntemi anlatan kısa makale, "A Note on Two Problems in Connexion with Graphs", 1959'da Numerische Mathematik dergisinde çıktı.

Yöntemde yavaş olan, en yakını seçmektir. Bütün kavşaklara tek tek bakarak seçersek, 4 milyon kavşaklı bir haritada her seçim 4 milyon bakış tutar, seçim de 4 milyon kez yapılır: 16 trilyon adım.

Çözüm, adayları öncelikli kuyrukta tutmaktır: en küçüğü hep tepede tutan yapı, yani **hip** (heap). Tepedekini almak da yeni bir aday eklemek de en çok ağacın kat sayısı kadar adım sürer; 4 milyon adayda 22 kat.

Altı kavşaklı küçük bir haritada Gar'dan Liman'a bakalım. Yollar (dakika): Gar–Kule 1, Gar–Han 4, Kule–Han 2, Kule–Meydan 6, Han–Meydan 2, Han–Saray 6, Meydan–Liman 3, Saray–Liman 1. Kuyrukta Gar 0'la durur; çıkar, kesinleşir, Kule'yi 1, Han'ı 4 dakikayla kuyruğa koyar. Tepede Kule var; o da kesinleşir ve Han'a daha kısa bir yol bulur: 3 dakika. Han kuyruğa bir kez daha girer. Eski 4'ü hipin içinde arayıp silmek zahmetli, bırakmak ucuzdur. 3'lü kayıt önce çıkar ve Han kesinleşir. Sıra eski 4'e gelince Han zaten kesinleşmiştir, kayıt atlanır. Meydan'da da aynısı olur; sonunda Liman 8 dakikayla kesinleşir.

Kodda kuyruk (uzaklık, kavşak) çiftlerini tutan bir hiptir; n kayıt sayısı, kok_al tepedekini çıkarır, ekle yeni kayıt koyar:

```
void dijkstra(void) {
    while (n > 0) {
        int u = kok_al().dugum;
        if (bitti[u]) continue;
        bitti[u] = 1;
        Kenar *k = komsu[u];
        for (; k != NULL; k = k->sonraki)
            gevset(u, k);
    }
}
```

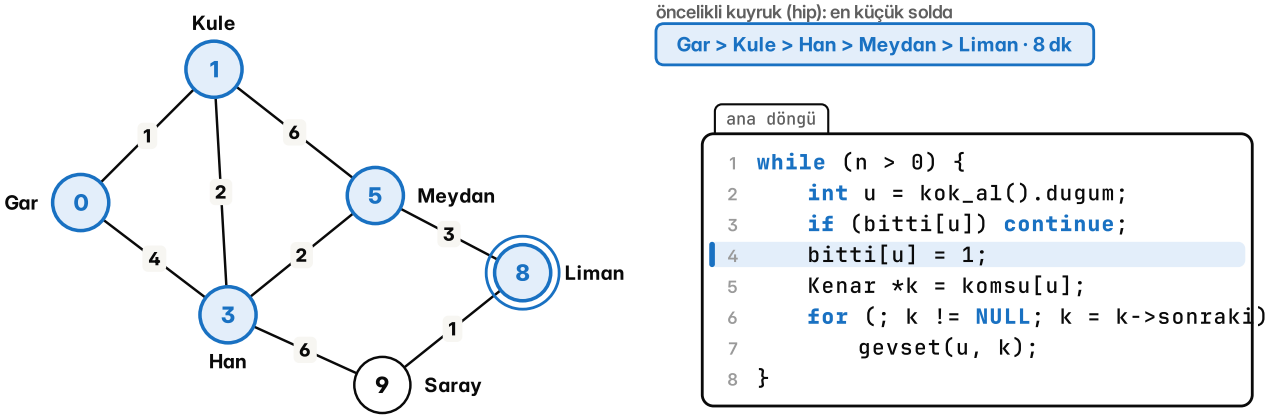
Kuyruktan çıkan kavşak zaten kesinleşmişse kayıt eskidir ve continue ile atlanır. gevset bir yolu gevşetir; süre kısaldıysa kavşağı yeni süresiyle kuyruğa ekler:

```
void gevset(int u, Kenar *k) {
    int v = k->hedef;
    int yeni = uz[u] + k->agirlik;
    if (yeni < uz[v]) {
        uz[v] = yeni;
        once[v] = u;
        ekle(v, yeni);
    }
}
```

Program kuyruğa giren (+) ve çıkan kayıtları yazdırır:

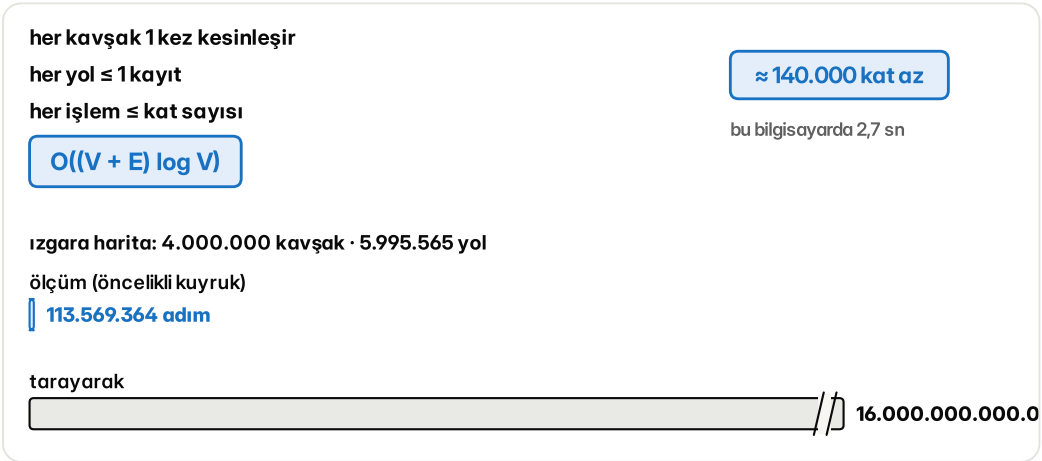
Çıktı:

```
başlangıç: +0 Gar
çıktı 0 Gar: kesinleşti | +1 Kule +4 Han
çıktı 1 Kule: kesinleşti | +7 Meydan +3 Han
çıktı 3 Han: kesinleşti | +9 Saray +5 Meydan
çıktı 4 Han: kesinleşmiş, atla
çıktı 5 Meydan: kesinleşti | +8 Liman
çıktı 7 Meydan: kesinleşmiş, atla
çıktı 8 Liman: kesinleşti |
çıktı 9 Saray: kesinleşti |
en kısa: Gar > Kule > Han > Meydan > Liman · 8 dk
```



Şekil 11.4 Gar'dan Liman'a: Han ve Meydan kuyruğa ikişer kez girer, eski kayıtlar sırası gelince atlanır; en kısa yol Gar > Kule > Han > Meydan > Liman, 8 dakika.

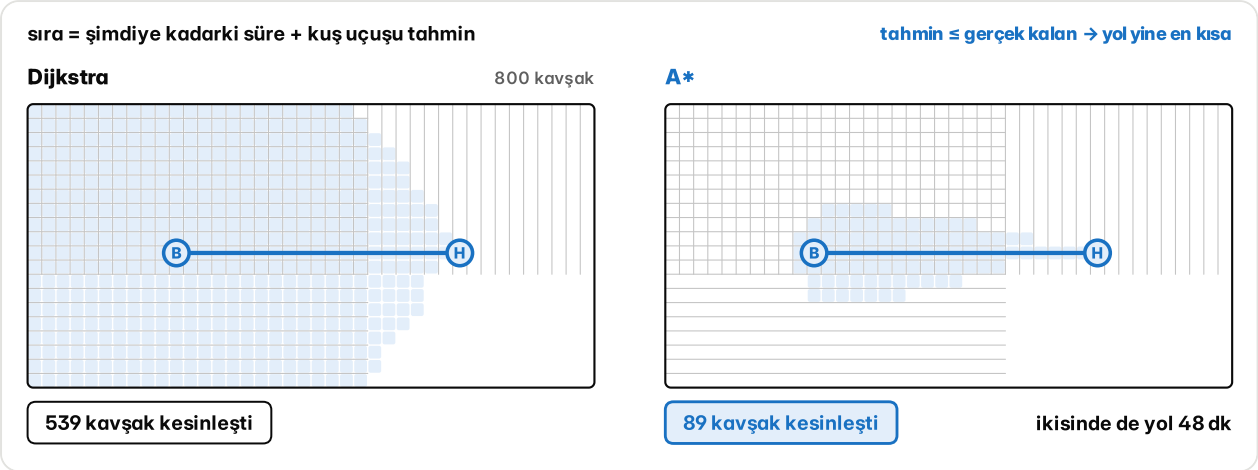
Her kavşak bir kez kesinleşir, her yol kuyruğa en çok bir kayıt ekler, her kuyruk işlemi de en çok kat sayısı kadar sürer; maliyet O((V + E) log V) olur. 4.000.000 kavşaklı, 5.995.565 yollu bir ızgara haritada ölçtük: 113.569.364 adım, bu bilgisayarda 2,7 saniye. 16 trilyonun yüz binde birinden az.



Şekil 11.5 4.000.000 kavşaklı ızgarada öncelikli kuyrukla 113.569.364 adım; tarayarak 16.000.000.000.000 adım, yaklaşık 140.000 kat fazla.

Navigasyon bir adım daha atar. Dijkstra her yöne aynı hızda yayılır, hedefin ters yönündeki sokakları da tarar. **A*** yönteminde bir kavşağın sırası, oraya kadarki süreye hedefe kalan kuş uçuşu tahmin eklenerek belirlenir; arama hedefe doğru uzanır. Tahmin gerçek kalan yoldan hiç

uzun değilse bulunan yol yine en kısadır. 800 kavşaklı bir ızgarada Dijkstra 539 kavşağı kesinleştirdi, A* 89'unu; yol ikisinde de 48 dakika.



Şekil 11.6 Aynı 800 kavşaklı ızgarada Dijkstra 539, A* 89 kavşağı kesinleştirir; ikisinde de yol 48 dakika.

Dijkstra	En yakını seçmek	Maliyet
Tarayarak	bütün kavşaklara bakarak	$O(V^2)$
Öncelikli kuyrukla	hipin tepesinden	$O((V + E) \log V)$

Navigasyonların bugün kullandığı yöntemler bu fikri hızlandırır ama değiştirmez: en yakını kesinleştir, komşuları güncelle.

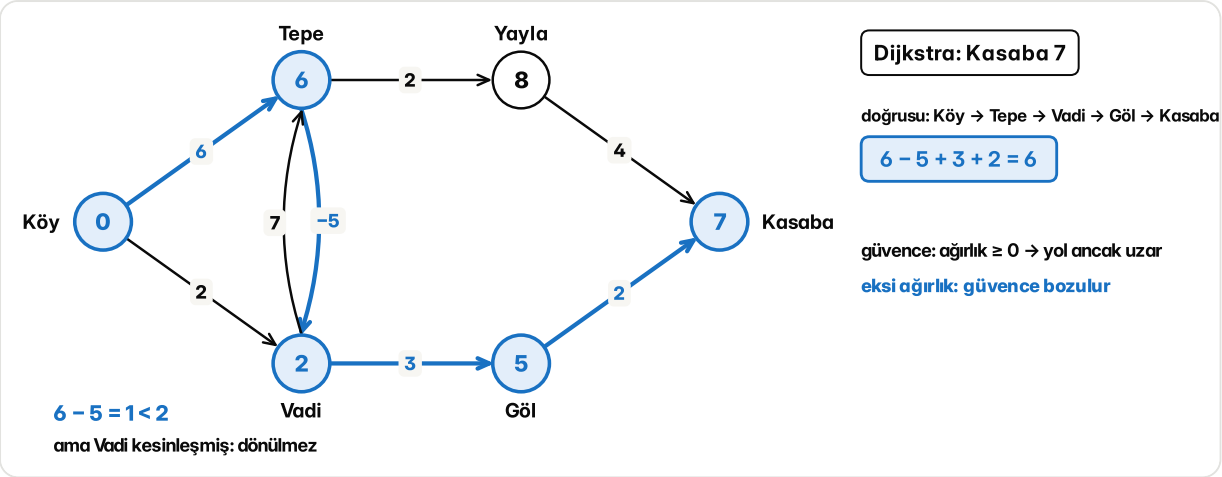
Eksi yol

videoda 1:12:32

Bir yolun uzunluğu eksi olabilir mi? Elektrikli araç için olabilir: yokuş aşağı inerken bataryası şarj olur. Böyle bir haritada Dijkstra'nın yöntemi neden yanılır?

Köyden kasabaya gidecek bir aracın haritasında her yolun üstüne bataryanın yüzde kaçını harcadığını yazalım (değerler temsilidir). Köy'den Tepe'ye yokuş: %6. Tepe'den Vadi'ye iniş: batarya %5 şarj olur, ağırlık −5. Yokuşun dönüşü ayrı bir yoldur, Vadi'den Tepe'ye %7; graf yönlüdür. Öteki yollar: Köy→Vadi 2, Vadi→Göl 3, Göl→Kasaba 2, Tepe→Yayla 2, Yayla→Kasaba 4.

Dijkstra kesinleştirdiği kavşağa bir daha dönmez. Köy'den başlar: Vadi'ye 2, Tepe'ye 6 yazar. En küçük Vadi; kesinleşir, Göl'e 5 yazılır. Sonra Göl kesinleşir, Kasaba'ya 7 gelir. Tepe kesinleşince inişten Vadi'ye bir yol bulunur: 6 − 5 = 1. Ama Vadi çoktan kesinleşmişti; ondan sonraki Göl ve Kasaba yeniden hesaplanmaz. Yöntem Kasaba için 7 der, oysa Tepe ve Vadi üstünden 6 − 5 + 3 + 2 = 6 yeter. Yöntemin güvencesi, yolun ancak uzayabilmesiydi; eksi bir yol bu güvenceyi bozar.



Şekil 11.7 Dijkstra Vadi'yi 2'yle erken kesinleştirir; Tepe üstünden gelen 6 − 5 = 1 işlenmez ve Kasaba için 7 bulunur, doğrusu 6.

Eksi yollarda doğruyu bulan yöntemin adı **Bellman-Ford**. Hiçbir kavşağı erken kesinleştirmez; bütün yolları sırayla gevşetir ve bunu turlarla tekrarlar. Kodda yollar bir dizide durur; her Yol başlangıcını (u), ucunu (v) ve ağırlığını (w) tutar. SONSUZ henüz ulaşılmamış kavşağın değeridir:

```
int gevset(Yol y) {
    if (uz[y.u] == SONSUZ) return 0;
    if (uz[y.u] + y.w >= uz[y.v]) return 0;
    uz[y.v] = uz[y.u] + y.w;
    return 1;
}
```

Yollar her turda Göl→Kasaba, Vadi→Göl, Tepe→Vadi, Vadi→Tepe, Tepe→Yayla, Yayla→Kasaba, Köy→Tepe, Köy→Vadi sırasıyla gevşetilir. Birinci turda Köy'den doğrudan gidilen Tepe ve Vadi bulunur. İkinci turda Vadi, Tepe'nin inişiyle 1'e düşer. Üçüncü turda Göl 4'e, dördüncü turda Kasaba 6'ya iner. Beşinci turda hiçbir şey değişmez.



Şekil 11.8 Bellman-Ford'un turları: Vadi 2. turda 1'e, Göl 3. turda 4'e, Kasaba 4. turda 6'ya iner; 5. turda değişiklik yok.

Kaç tur gerekir? En kısa yol bir kavşaktan iki kez geçmez; 6 kavşaklı haritada en çok 5 yol kullanır. Her tur bu yolun bir adımını daha yerine oturtur; kavşak sayısının bir eksiği kadar, yani V − 1 turda en uzunu bile oturur. Bir turda hiçbir şey değişmezse beklemeden durulabilir.

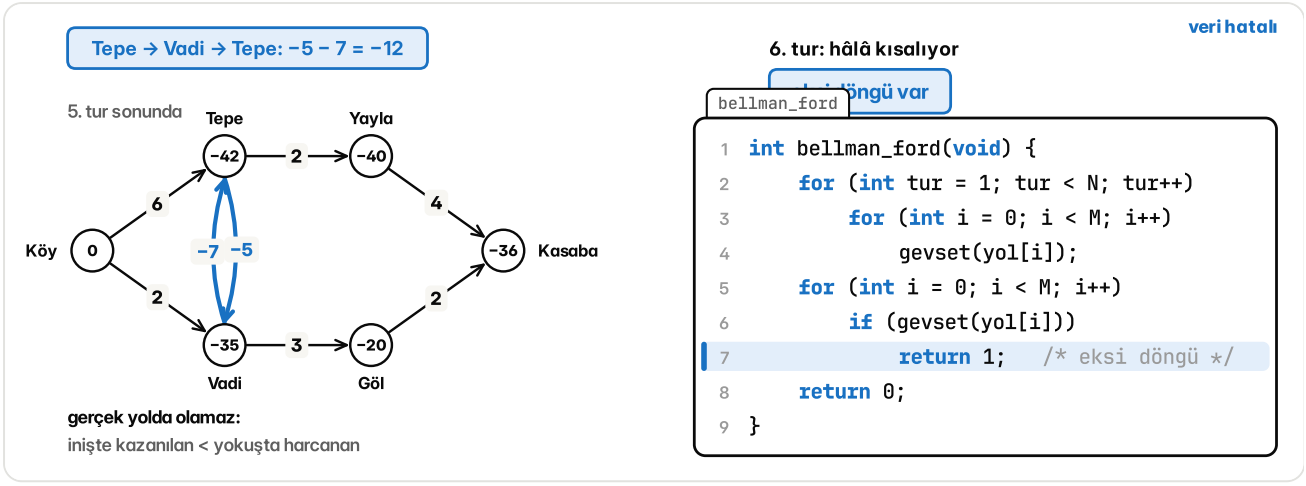
Şimdi haritaya bir hata girsin: Vadi'den Tepe'ye çıkış yanlışlıkla −7 yazılmış olsun. Tepe'den Vadi'ye inip yine Tepe'ye çıkan her tur bataryayı %12 doldurur; döndükçe yol kısalır, en kısa yol diye bir şey kalmaz. Buna **eksi döngü** denir. Bellman-Ford bunu yakalar: V − 1 turdan sonra bir tur daha yapar; hâlâ kısalan bir değer varsa haritada eksi döngü vardır. Gerçek bir yolda böyle döngü olamaz, aynı yere dönen araç inişte kazandığından fazlasını yokuşta harcar; demek ki veri hatalıdır.

```
int bellman_ford(void) {
    for (int tur = 1; tur < N; tur++)
        for (int i = 0; i < M; i++)
            gevset(yol[i]);
    for (int i = 0; i < M; i++)
        if (gevset(yol[i]))
            return 1;   /* eksi döngü */
    return 0;
}
```

N kavşak, M yol sayısıdır. Programın çıktısında önce Dijkstra, sonra Bellman-Ford'un turları ve hatalı harita var:

Çıktı:

```
dijkstra sırası: Köy 0 Vadi 2 Göl 5 Tepe 6 Kasaba 7 Yayla 8
dijkstra sonucu: Köy 0 Tepe 6 Vadi 2 Göl 5 Yayla 8 Kasaba 7
1. tur: Köy 0 Tepe 6 Vadi 2 Göl ∞ Yayla ∞ Kasaba ∞
2. tur: Köy 0 Tepe 6 Vadi 1 Göl 5 Yayla 8 Kasaba 12
3. tur: Köy 0 Tepe 6 Vadi 1 Göl 4 Yayla 8 Kasaba 7
4. tur: Köy 0 Tepe 6 Vadi 1 Göl 4 Yayla 8 Kasaba 6
5. tur: Köy 0 Tepe 6 Vadi 1 Göl 4 Yayla 8 Kasaba 6
eksi döngü: yok
bellman-ford sonucu: Köy 0 Tepe 6 Vadi 1 Göl 4 Yayla 8 Kasaba 6
hatalı harita, 6. tur kısalttı mı: evet, eksi döngü var
```



Şekil 11.9 Vadi→Tepe −7 girilince Tepe→Vadi→Tepe döngüsü −12 tutar; 5. tur sonunda Tepe −42, Kasaba −36'dır ve 6. tur hâlâ kısaltır: eksi döngü var.

Bu güvenliğin bedeli var: Bellman-Ford her turda bütün yollara bakar: 10.000 kavşaklı, 30.000 yollu bir haritada en kötü durumda 299.970.000 gevşetme. Dijkstra aynı haritada en çok yarım milyon kadar adım atar. Her kavşaktan her kavşağa uzaklık tablosu gerekirse Floyd-Warshall adlı yöntem kullanılır; maliyeti kavşak sayısının küpüdür, 1.000 kavşakta 1 milyar adım.

Yöntem	Ne zaman	Maliyet
Dijkstra	eksi ağırlık yok	$O((V + E) \log V)$
Bellman-Ford	eksi ağırlık olur, eksi döngüyü yakalar	$O(V \cdot E)$
Floyd-Warshall	her kavşaktan her kavşağa	$O(V^3)$

Elektrikli araçların rota planlayıcıları da yokuşu ve inişi hesaba katar; inişte bataryaya dönen enerji, onların haritasında eksi bir sayıdır.

Özet

- Ağırlıklı grafta en kısa yol, ağırlıkları toplamı en küçük yoldur; genişlik öncelikli arama yalnız yol sayısını en aza indirir.
- Dijkstra kesinleşmemiş en yakın düğümü kesinleştirir ve komşularını gevşetir; ağırlıklar eksi olmadığı için kesinleşen değer bir daha değişmez.
- En yakını tarayarak seçmek $O(V^2)$, öncelikli kuyrukla (hip) seçmek $O((V + E) \log V)$ tutar; kuyruktaki eski kayıt sırası gelince atlanır.
- A* sırayı şimdiye kadarki süreye hedefe kalan tahmini ekleyerek belirler; tahmin gerçeği aşmazsa yol yine en kısadır.
- Eksi ağırlıkta Dijkstra yanılabilir; Bellman-Ford bütün yolları $V - 1$ tur gevşetir, bir tur daha kısaltırsa eksi döngü vardır.
- Bellman-Ford $O(V \cdot E)$, bütün çiftler için Floyd-Warshall $O(V^3)$ tutar.

Sorular

1 **Genişlik öncelikli arama (BFS) ağırlıklı bir haritada neden en kısa süreli yolu her zaman bulamaz?**

- **A** Yalnız yönlü graflarda çalışır.
- **B** Yol sayısını en aza indirir; yolların süreleri farklıysa az yollu güzergâh uzun sürebilir.
- **C** Her kavşağa iki kez uğrar ve süreleri karıştırır.
- **D** Hedefe vardığında durmaz, bütün haritayı dolaşır.

2 **Yönsüz bir grafın yolları: A–B 4, A–C 1, C–B 2, B–D 1, C–D 5. Dijkstra'nın yöntemiyle A'dan D'ye en kısa yolun uzunluğunu bul.**

Cevap:

3 **Dijkstra'da Park kavşağı uz = 5 ile kesinleşti. Park'tan Köprü'ye giden yol 3 dakika, Köprü'nün yazılı süresi 9. Gevşetmeden sonra ne olur?**

- **A** Köprü'nün süresi 9 kalır, çünkü Köprü'ye önce Ev'den ulaşılmıştır.
- **B** Köprü'nün süresi 3 olur.
- **C** Köprü'nün süresi 8 olur ve önceki kavşağı Park diye not edilir.
- **D** Köprü hemen kesinleşir.

4 **En yakını bütün kavşaklara bakarak seçen Dijkstra, 10.000 kavşaklı bir haritada kabaca kaç bakış yapar?**

- **A** 100.000.000
- **B** 10.000
- **C** 20.000
- **D** 140.000

5 **Öncelikli kuyruklu Dijkstra'da Han kuyrukta hem 4 hem 3 dakikalık kayıtla durur. 3'lü kayıt çıkıp Han kesinleştikten sonra 4'lü kayıt kuyruktan çıkınca ne olur?**

- **A** Han'ın süresi 4'e çıkarılır.
- **B** Program hata verir, çünkü bir kavşak kuyrukta iki kez olamaz.
- **C** Han yeniden kesinleşir ve komşuları yeniden gevşetilir.
- **D** Han zaten kesinleşmiş olduğu için kayıt atlanır.

6 **A* yönteminin bulduğu yolun yine en kısa olması için hedefe kalan tahmin hangi koşulu sağlamalıdır?**

- **A** Tahmin, gerçek kalan yoldan hiç uzun olmamalıdır.
- **B** Tahmin her zaman 0 olmalıdır.
- **C** Tahmin gerçek kalan yolun iki katı olmalıdır.
- **D** Tahmin yalnız yönlü graflarda kullanılmalıdır.

7 **Yönlü bir grafın yolları: S→A 4, S→B 3, B→A −2, A→T 2. Bellman-Ford her turda yolları A→T, B→A, S→A, S→B sırasıyla gevşetiyor; başta S 0, ötekiler sonsuz. 2. turun sonunda T'nin değeri kaçtır?**

Cevap: _____

8 **Bellman-Ford bir haritada eksi döngü olduğunu nasıl anlar?**

A Herhangi bir yolun ağırlığı eksiyse eksi döngü var der.

B V − 1 turdan sonra bir tur daha yapar; hâlâ kısalan bir değer varsa eksi döngü vardır.

C İlk turda hiçbir değer değişmezse eksi döngü var der.

D Dijkstra'yla aynı sonucu bulamazsa eksi döngü var der.

VERİ YAPILARI

12 Metin arama

Bu bölümde uzun bir metinde bir kalıbı, yani aranan sözcüğü ya da parçayı bulmanın yollarını göreceğiz: her konumu baştan deneyen kaba kuvveti, harflerin çoğuna bakmadan atlayan Boyer-Moore'u, metinde hiç geri dönmeyen KMP'yi, pencereyi bir sayıya çeviren Rabin-Karp'ı ve bir kelimenin başını yazılan harf kadar adımda bulan önek ağacını.

Bir milyon harfte bir kelime

videoda 1:15:06

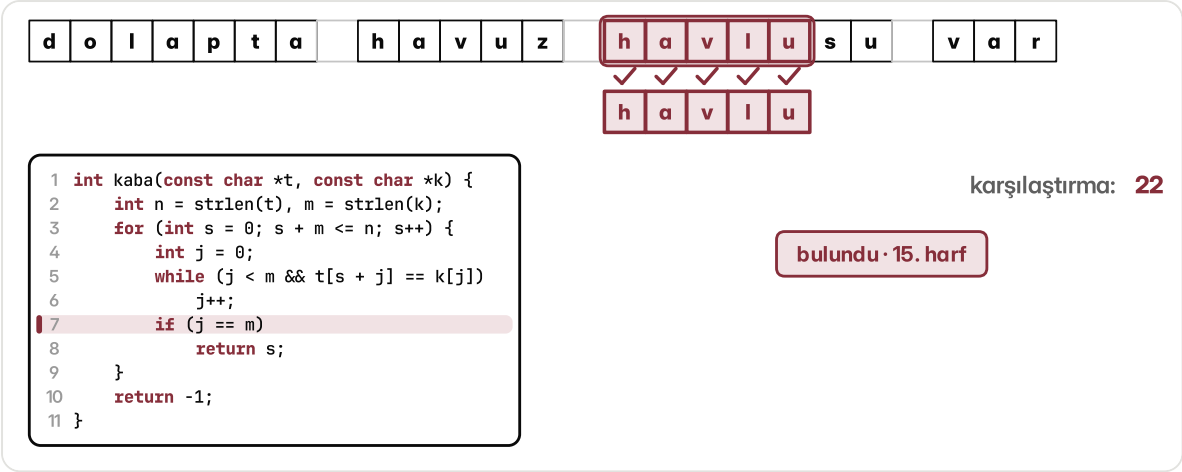
500 sayfalık bir romanda yaklaşık 1 milyon harf var. Arama kutusuna bir kahramanın adını yazınca ilk geçtiği satır bir göz kırpımında ekrana gelir. Ctrl+F, 1 milyon harfin içinde bir kelimeyi bu kadar hızlı nasıl bulur? Aynı iş, 3 milyar harflik insan genomunda bir geni ararken de yapılır.

En basit yol **kaba kuvvet**. "dolapta havuz havlusu var" cümlesinde (boşluklarla 25 harf) "havlu" sözcüğünü arayalım. Kalıbı metnin başına koyar, harf harf karşılaştırırız; ilk uyuşmazlıkta kalıbı bir harf sağa kaydırıp baştan başlarız. İlk 8 konumda daha ilk harf tutmaz. "havuz" sözcüğünde 3 harf tutar, dördüncüde bozulur. Kalıp 15. harfte bulunur; toplam 22 karşılaştırma. Kodda t metin, k kalıp, s kalıbın metindeki konumu, j de tutan harf sayısıdır; strlen bir metnin uzunluğunu verir:

```
int kaba(const char *t, const char *k) {
    int n = strlen(t), m = strlen(k);
    for (int s = 0; s + m <= n; s++) {
        int j = 0;
        while (j < m && t[s + j] == k[j])
            j++;
        if (j == m)
            return s;
    }
    return -1;
}
```

Örnek program her konumu ve karşılaştırma sayısını yazdırır. Çıktının son 11 satırı:

```
   8. harf:  havu  0 harf tuttu, 1 karşılaştırma
   9. harf: havuz  3 harf tuttu, 4 karşılaştırma
  10. harf: avuz   0 harf tuttu, 1 karşılaştırma
  11. harf: vuz h  0 harf tuttu, 1 karşılaştırma
  12. harf: uz ha  0 harf tuttu, 1 karşılaştırma
  13. harf: z hav  0 harf tuttu, 1 karşılaştırma
  14. harf:  havl  0 harf tuttu, 1 karşılaştırma
  15. harf: havlu  5 harf tuttu, 5 karşılaştırma
metin 25 harf, kalıp 5 harf
kaba: 15. harfte bulundu, toplam 22 karşılaştırma
kontrol: kaba() -> 15
```



Şekil 12.1 Kaba kuvvet "havlu" kalıbını her konumda soldan karşılaştırır; 15. harfte bulur, toplam 22 karşılaştırma.

Gerçek metinde çoğu konumda daha ilk harf tutmaz. 1 milyon harflik Türkçe bir metinde "zaman" sözcüğünü aramak 1.016.000 karşılaştırma gerektirdi: harf başına yaklaşık 1. Bu bilgisayar bunu yaklaşık 5 milisaniyede bitirdi. Ama metin tekrarlıysa iş değişir: metin 1 milyon kez aynı harf (a), kalıp 1.000 harf; kalıbın da hepsi a, yalnız sonuncusu farklı (b). Her konumda 999 harf tutar, sonuncusu bozulur: 999.001.000 karşılaştırma, neredeyse 1 milyar; bu bilgisayarda 1 saniyeye yakın. Kaba kuvvetin en kötü durumu budur: iş, metin boyu (n) × kalıp boyu (m) kadar, yani O(n·m).

Daha hızlısı, her harfe bakmamaktır. **Boyer-Moore** yöntemi kalıbı yine metnin başına koyar ama karşılaştırmaya sağdan, kalıbın son harfinden başlar. Kalıbın son harfinin altında p var; "havlu"da p yok. Kalıp, bu harfin üstünden geçen hiçbir konumda tutamaz: 5 harf birden atlarız. Tutmayan metin harfine bakıp atlamaya **kötü karakter kuralı** denir. Bunun için arama başlamadan son[] tablosu kurulur: her harfin kalıptaki en sağdaki yeri (h 0, a 1, v 2, l 3, u 4), kalıpta olmayan harfler için −1. Uyuşmazlık kalıbın j. gözünde, c harfinde çıktıysa kalıp j − son[c] kadar, en az 1 harf kayar.



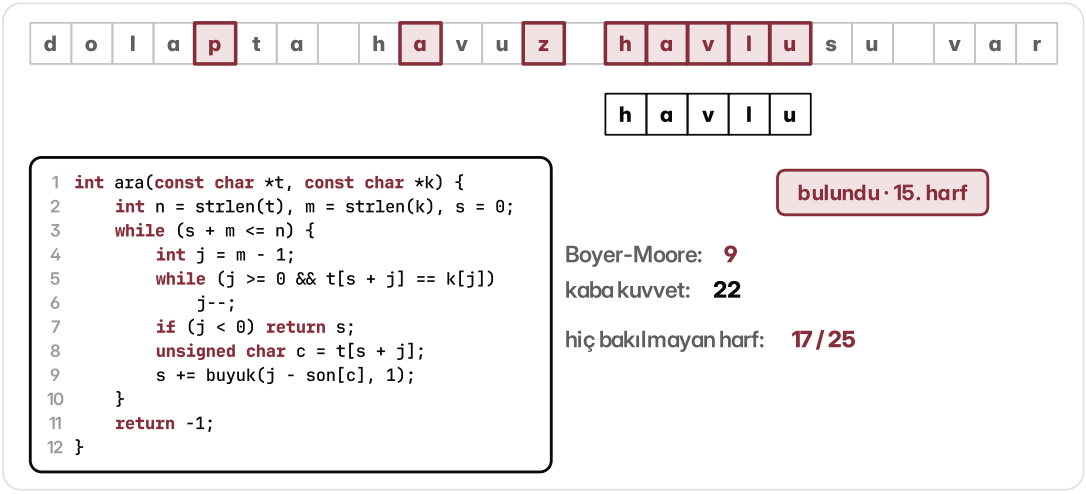
Şekil 12.2 Son harfin altındaki p kalıpta yok: kalıp 5 göz birden atlar; son[ ] tablosu her harfin kalıptaki en sağdaki yerini tutar.

Yeni konumda son harfin altında a var; a kalıbın 2. harfi. Kalıbı, kendi a'sı bu a'nın altına gelecek kadar, 3 harf kaydırırız. Sonra z: kalıpta yok, yine 5 harf. Sonra l: kalıbın 4. harfi, 1 harf. Şimdi sağdan 5 harf tutar: "havlu" bulundu. Kaba kuvvetin 22 karşılaştırması yerine 9; metnin 25 harfinden 17'sine hiç bakılmadı. Kodda buyuk iki sayının büyüğünü verir:

```
int ara(const char *t, const char *k) {
    int n = strlen(t), m = strlen(k), s = 0;
    while (s + m <= n) {
        int j = m - 1;
        while (j >= 0 && t[s + j] == k[j])
            j--;
        if (j < 0) return s;
        unsigned char c = t[s + j];
        s += buyuk(j - son[c], 1);
    }
    return -1;
}
```

Çıktı:

```
   1. harften: altta 'p', kalıpta yok -> 5 harf atla
   6. harften: altta 'a', kalıbın 2. harfi -> 3 harf kaydır
   9. harften: altta 'z', kalıpta yok -> 5 harf atla
  14. harften: altta 'l', kalıbın 4. harfi -> 1 harf kaydır
  15. harften: kalıp tuttu
Boyer-Moore: 15. harfte bulundu, 9 karşılaştırma
metnin 25 harfinden 8 tanesine bakıldı, 17 tanesine hiç bakılmadı
kontrol: ara() -> 15
```



Şekil 12.3 Sıçramalar 5, 3, 5, 1: Boyer-Moore 9 karşılaştırmada bulur, kaba kuvvet 22; 25 harfin 17'sine hiç bakılmaz.

Kalıp uzadıkça atlamalar büyür. Aynı 1 milyon harflik metinde 5 harflik "zaman" aranırken 237.000 harfe bakıldı; 10 harflik "bilgisayar" için bu sayı yaklaşık 140.000'e iner. Bu yöntemi Boyer ile Moore 1977'de yayımladı; programcıların metin arama aracı grep de onunla çalışır. Koca kitapta anında bulmanın sırrı bu: kaba kuvvet bile birkaç milisaniye sürer, atlayan yöntem harflerin çoğuna hiç bakmaz.

Yöntem	Maliyet
Kaba kuvvet, en kötü	O(n·m)
Boyer-Moore, çoğu metinde	yaklaşık n / m bakış
Boyer-Moore, en kötü (yalnız bu kuralla)	O(n·m)

Geri dönmek şart mı?

videoda 1:17:23

Bir mesajda, tuş takılınca "durdurdurmak" yazılmış; içinde "durdurmak" sözcüğünü arıyoruz. İlk 6 harf tutuyor, yedincide bozuluyor. Metinde geri dönüp aynı harfleri yeniden okumak zorunda mıyız?

Kaba kuvvet kalıbı bir harf sağa kaydırır ve metnin 2. harfine, yani 5 harf geri döner; okuduğu harfleri yeniden okur. Bu kısa metinde 18 karşılaştırma eder.

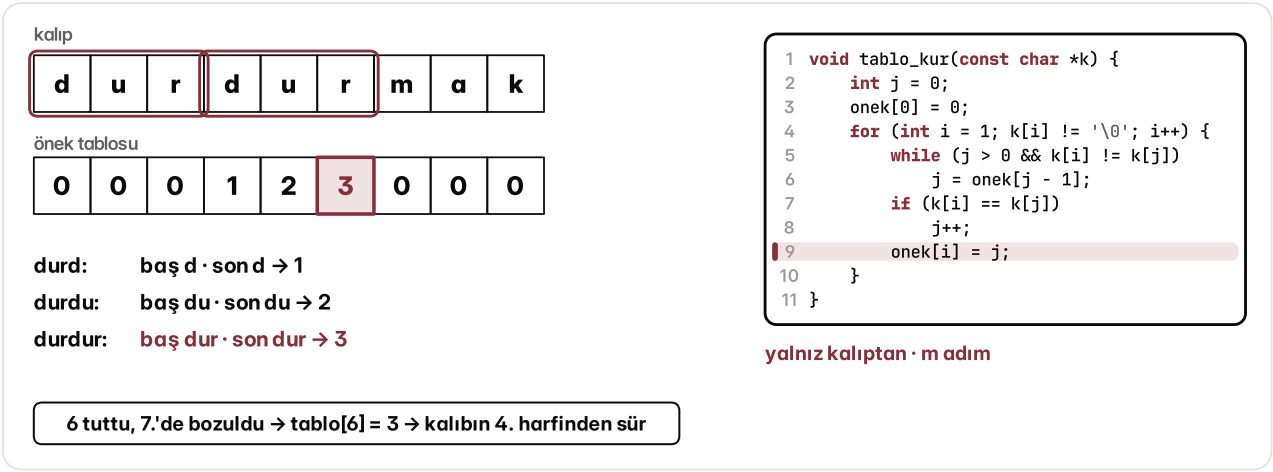
Oysa tutan 6 harf belli: "durdur". Bunu bilmek için metne bakmak bile gerekmez, çünkü bunlar kalıbın kendi ilk 6 harfi. Bu 6 harfin sonu "dur", başı da "dur". Kalıbı 3 harf kaydırırsak, kalıbın ilk 3 harfi metindeki son 3 harfin altına gelir; ikisi de "dur", yani bu 3 harf zaten tutuyor. Karşılaştırma metinde kaldığımız yerden, 7. harften sürer: metnin 7. harfi kalıbın 4. harfiyle tutar, sonra 5 harf daha. "durdurmak", metnin 4. harfinden başlayarak bulundu; 13 karşılaştırmada.



Şekil 12.4 Tutan "durdur"un başı ve sonu "dur": kalıp 3 harf kayar, karşılaştırma metnin 7. harfinden sürer; 13 karşılaştırma, kaba kuvvette 18.

Her uyuşmazlıkta bunu bilmek için kalıbın her başlangıç parçasına tek bir sayı yazarız: o parçanın hem başında hem sonunda duran, kendisinden kısa en uzun kısmın boyu. Bu sayılara **önek tablosu** denir. "durdurmak" için tablo ilk 3 harfte 0, sonra 1, 2, 3, sonra yine 0: 0 0 0 1 2 3 0 0 0. Altıncı sayı 3: ilk 6 harfin hem başında hem sonunda "dur" var. 6 harf tutup yedincide bozulunca tabloya bakarız: 3 harf hâlâ tutuyor, kalıbın 4. harfinden devam. Tablo aramadan önce, yalnız kalıba bakılarak, kalıp boyu kadar adımda kurulur:

```
void tablo_kur(const char *k) {
    int j = 0;
    onek[0] = 0;
    for (int i = 1; k[i] != '\0'; i++) {
        while (j > 0 && k[i] != k[j])
            j = onek[j - 1];
        if (k[i] == k[j])
            j++;
        onek[i] = j;
    }
}
```



Şekil 12.5 "durdurmak"ın önek tablosu 0 0 0 1 2 3 0 0 0; 6 harf tutup 7.'de bozulunca tablo[6] = 3, kalıbın 4. harfinden sürülür.

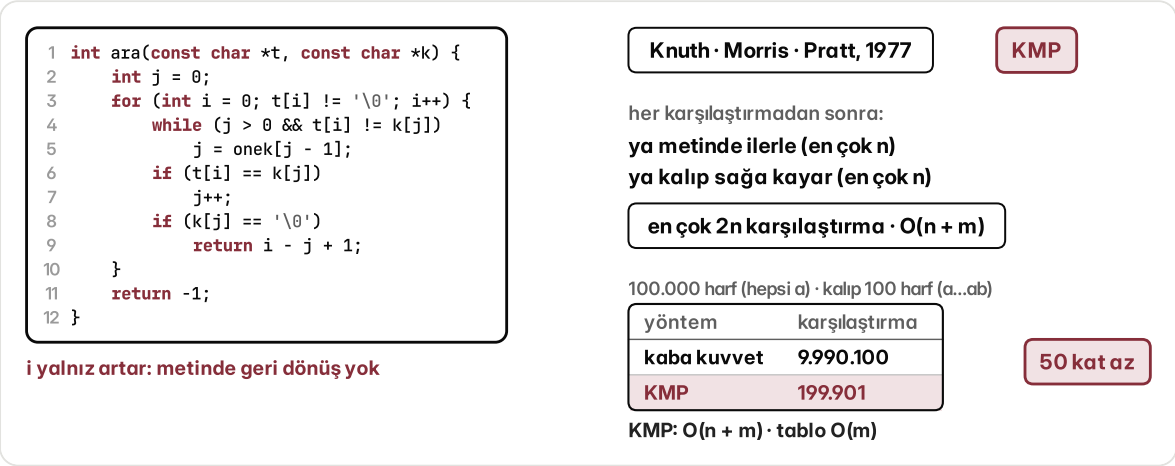
Bu yönteme, onu 1977'de yayımlayan üç araştırmacının (Knuth, Morris, Pratt) soyadlarının baş harfleriyle **KMP** denir. Aramada i metindeki yeri, j kalıpta o ana kadar tutan harf sayısını tutar:

```
int ara(const char *t, const char *k) {
    int j = 0;
    for (int i = 0; t[i] != '\0'; i++) {
        while (j > 0 && t[i] != k[j])
            j = onek[j - 1];
        if (t[i] == k[j])
            j++;
        if (k[j] == '\0')
            return i - j + 1;
    }
    return -1;
}
```

i yalnız artar: metinde hiç geri dönülmez. Her karşılaştırmadan sonra ya metinde bir harf ilerlenir ya da kalıp sağa kayar; ikisi de en çok n kez olur. Bu yüzden arama en çok 2n karşılaştırma, tabloyla birlikte iş metin boyu + kalıp boyu kadardır: O(n + m). Örnek program önce tabloyu yazar (önek tablosu (durdurmak): 0 0 0 1 2 3 0 0 0), sonra her karşılaştırmayı. Çıktının son 11 satırı:

```
  metin 6. 'r' / kalıp 6. 'r': tuttu
  metin 7. 'd' / kalıp 7. 'm': tutmadı, tabloda 3 -> kalıbın 4. harfinden sür
  metin 7. 'd' / kalıp 4. 'd': tuttu
  metin 8. 'u' / kalıp 5. 'u': tuttu
  metin 9. 'r' / kalıp 6. 'r': tuttu
  metin 10. 'm' / kalıp 7. 'm': tuttu
  metin 11. 'a' / kalıp 8. 'a': tuttu
  metin 12. 'k' / kalıp 9. 'k': tuttu
KMP: 4. harften başlayarak bulundu, 13 karşılaştırma
kaba kuvvet: 18 karşılaştırma
kontrol: ara() -> 4
```

Fark tekrarlı metinde büyür: 100.000 harflik bir metinde (hepsi a) 100 harflik bir kalıbı (99 a ve bir b) ararken kaba kuvvetin 9.990.100 karşılaştırması, KMP ile 199.901'e iner: 50 kat az.



Şekil 12.6 KMP'de i yalnız artar; en çok 2n karşılaştırma, O(n + m). Tekrarlı metinde kaba kuvvet 9.990.100, KMP 199.901 karşılaştırma.

Yöntem	Maliyet
KMP, arama	O(n + m)
Önek tablosu	O(m)

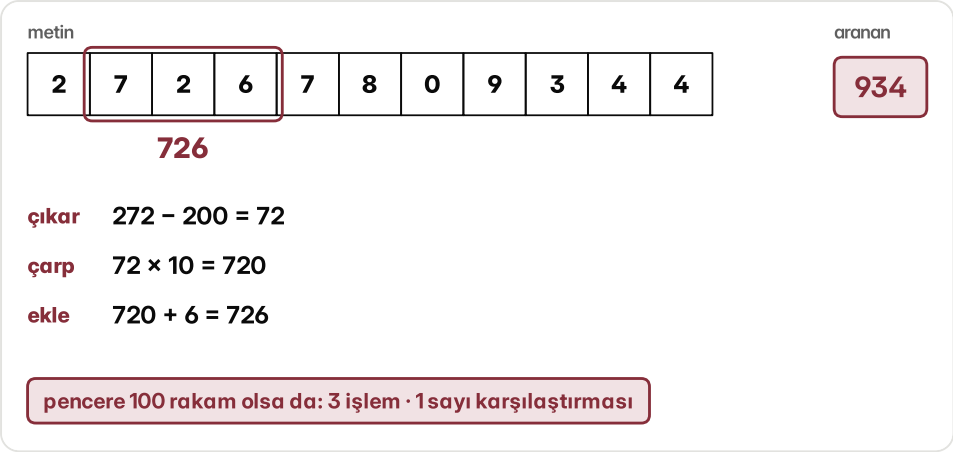
Geri dönmediği için metni saklamak da gerekmez: ağdan akan bir veride kalıp, metni bir kez baştan sona okuyarak aranır. Aynı harfleri yeniden okumamanın bütün sırrı, kalıbın kendi içindeki tekrarı aramadan önce öğrenmektir.

Kopya nasıl yakalanır?

videoda 1:19:20

Kopya denetleyen bir program, bir ödevi binlerce sayfalık bir arşivle nasıl karşılaştırır? Ödevden 100 parçayı, 10 milyon harflik arşivin her konumunda tek tek denemek 1 milyar deneme eder.

Önce rakamla düşünelim: metin 2 7 2 6 7 8 0 9 3 4 4 diye gidiyor, aranan 934. Metnin üstünde 3 rakamlık bir pencere kayar ve her pencere bir sayıdır: önce 272. Bir sağa kayınca baştaki 2'nin payını, 200'ü çıkarırız, 72 kalır; 10'la çarparız, 720; yeni rakamı, 6'yı ekleriz: 726. Pencere 100 rakamlık olsa da kaydırmak 3 işlemdir ve her konumda 100 rakam yerine tek bir sayı karşılaştırılır.



Şekil 12.7 Pencere 272'den 726'ya üç işlemle kayar: 272 − 200 = 72, 72 × 10 = 720, 720 + 6 = 726.

Ama 100 rakamlık sayı bilgisayara sığmaz (64 bitlik bir tam sayı en çok 20 basamak tutar); bu yüzden sayının kendisi değil, bir asala, burada 13'e bölümünden kalanı tutulur. Pencerenin hash'i budur; kayarken aynı 3 işlem kalanlarla yapılır. Kodda ASAL 13'tür; US, baştaki rakamın payının kalanıdır: 100'ün 13'e bölümünden kalan, 9:

```
int kaydir(int h, int eski, int yeni) {
    h = h - eski * US % ASAL;
    if (h < 0)
        h += ASAL;
    return (h * 10 + yeni) % ASAL;
}
```

Aranan 934'ün kalanı 11; kalanı 11 olmayan pencere kesinlikle farklıdır, geçilir. Ama 726'nın kalanı da 11: rakamlar tek tek karşılaştırılır ve tutmaz. Kalan aynı olduğu hâlde kalıbın tutmamasına **sahte eşleşme** denir. 934'te rakamlar da tutar: bulundu. Aramada h pencerenin, p kalıbın kalanıdır; esit, M rakamı tek tek karşılaştırır:

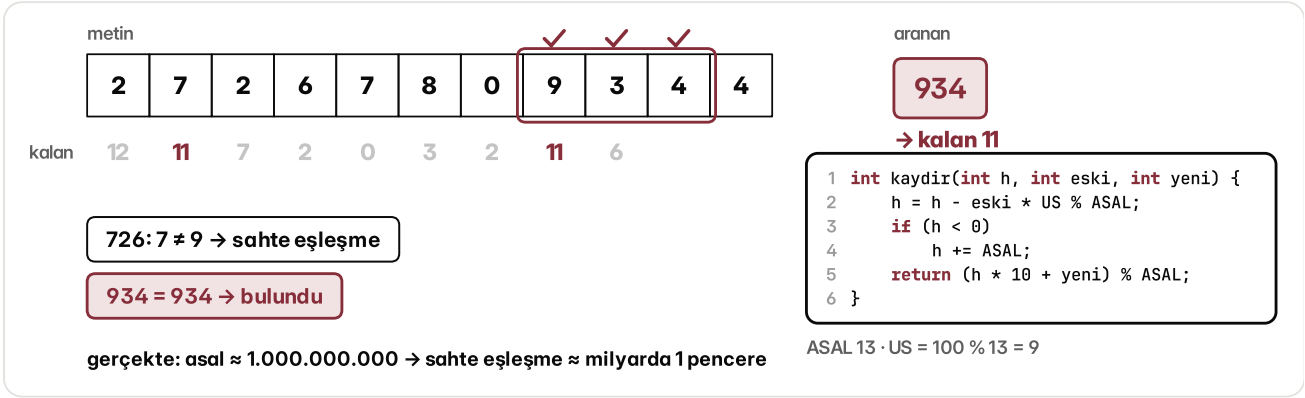
```
void ara(int h, int p) {
    for (int s = 0; s + M <= N; s++) {
        if (h == p && esit(t + s, k))
            printf("%d. konumda\n", s + 1);
        if (s + M < N)
            h = kaydir(h, t[s], t[s + M]);
    }
}
```

Örnek program her pencereyi ve kalanını da yazar.

Çıktı:

```
aranan 934, kalanı 11
  1. pencere 272: kalan 12
  2. pencere 726: kalan 11  -> sahte eşleşme, rakamlar tutmadı
  3. pencere 267: kalan  7
  4. pencere 678: kalan  2
  5. pencere 780: kalan  0
  6. pencere 809: kalan  3
  7. pencere 093: kalan  2
  8. pencere 934: kalan 11  -> rakamlar da tuttu: bulundu
  9. pencere 344: kalan  6
ara(): 8. konumda
272 -> (272 - 2*100) * 10 + 6 = 726
```

Gerçekte asal milyar civarında seçilir; sahte eşleşme milyarda 1 pencerede çıkar.

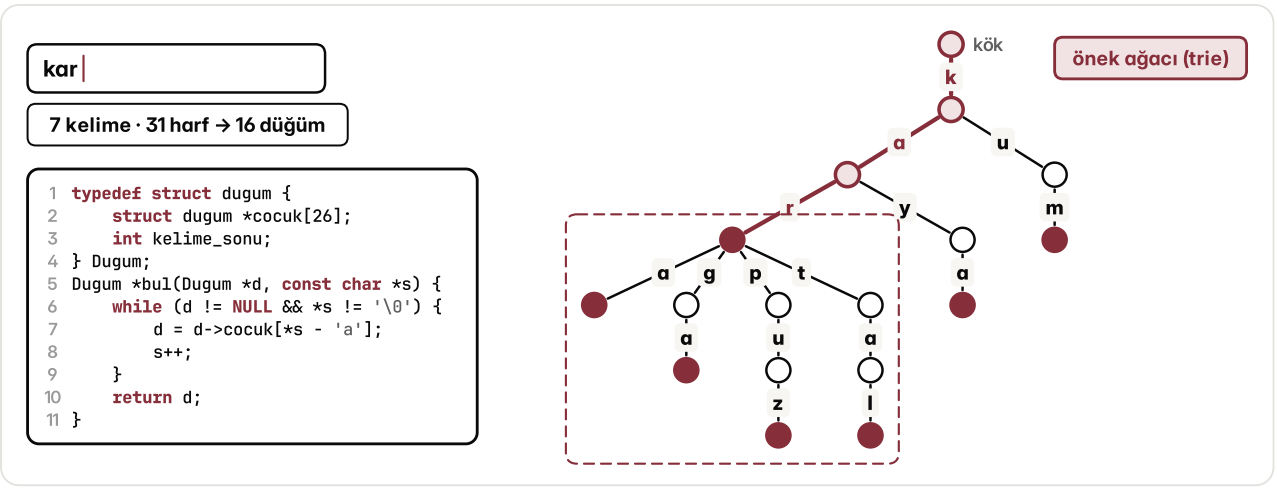


Şekil 12.8 Pencerelerin 13'e bölümünden kalanları 12 11 7 2 0 3 2 11 6: kalanı 11 olan 726 sahte eşleşme, 934 gerçek eşleşme.

Harflerde de aynısı geçerlidir: polinom hash, her harfin sayı kodundan önce toplamı bir sayıyla, örneğin 31'le çarpar; 10 yerine 31'le aynı 3 işlem harf penceresini kaydırır. Bu yönteme, onu bulan iki araştırmacının adıyla **Rabin-Karp** denir. Asıl gücü, çok kalıbı birden aramaktır: ödevin 50 harflik bütün parçalarının hash'leri bir hash tablosuna konur. Arşivde kayan pencerenin hash'i her adımda güncellenip tabloda tek bakışla aranır. 10 milyon harflik arşiv, kaç parça aranırsa aransın, yaklaşık 10 milyon adımda taranır. Ödev kopyası arayan programlar da metnin kısa parçalarını böyle hash'lere, parmak izlerine çevirir.

Aranan bir kelimenin başıysa başka bir yapı işe yarar. Arama kutusuna "kar" yazınca kar, kara, karga, karpuz ve kartal önerilsin. Sözlükteki kelimeler harf harf bir ağaca dizilir: kökten inen her kenar bir harf, ortak başlangıçlar tek bir yol. Sözlükte kaya ve kum da olsun: 7 kelimenin 31 harfi, kök dışında 16 düğüme sığar. "kar" yazınca kökten 3 adım inilir; o düğümün altındaki bütün kelimeler önerilerdir. Bu ağaca **önek ağacı** (trie) denir. Her düğüm (Dugum) 26 küçük harf için birer çocuk işaretçisi ve orada bir kelime bitip bitmediğini tutar:

```
typedef struct dugum {
    struct dugum *cocuk[26];
    int kelime_sonu;
} Dugum;
```



Şekil 12.9 Yedi kelimelik önek ağacı: ortak başlangıçlar tek yol, 31 harf 16 düğüme sığar; "kar" yolu kökten 3 adımdır.

bul, yazılan her harf için bir kat iner; yol yoksa NULL döner:

```
Dugum *bul(Dugum *d, const char *s) {
    while (d != NULL && *s != '\0') {
        d = d->cocuk[*s - 'a'];
        s++;
    }
    return d;
}
```

Çıktı:

```
7 kelime, toplam 31 harf, ağaçta 17 düğüm (kök dahil)
'kar' ile başlayanlar (kökten 3 adım):
  kar
  kara
  karga
  karpuz
  kartal
'kas': yok
'karpuz': 6 adım, kelime
```

Bir kelimenin başına inmek, sözlük 100.000 kelime de olsa, yazılan harf sayısı kadar adım sürer: karpuz 6 adım. Öneriler daha üçüncü harfte bu yüzden gelebilir.

Yöntem	Maliyet
Rabin-Karp, ortalama	O(n + m)
Rabin-Karp, en kötü	O(n·m)
Önek ağacında bul	O(kelime boyu)

Sözlük aynı sözlük, yalnız düzeni farklı: aynı veri doğru düzende durunca, aranan şeye en kısa yoldan varılır.

Özet

- Kaba kuvvet kalıbı her konumda soldan karşılaştırır; gerçek metinde hızlıdır ama en kötü durumda O(n·m) tutar.
- Boyer-Moore sağdan karşılaştırır ve kötü karakter kuralıyla atlar; kalıp uzadıkça harflerin çoğuna hiç bakmaz.
- KMP önek tablosuyla uyuşmazlıktan sonra kaç harfin hâlâ tuttuğunu bilir, metinde geri dönmez: O(n + m).
- Rabin-Karp pencereyi bir sayının kalanına çevirir ve üç işlemle kaydırır; kalan tutunca harfler karşılaştırılır, çünkü sahte eşleşme olabilir.
- Önek ağacında ortak başlangıçlar tek yoldur; bir kelimenin başına yazılan harf kadar adımda inilir.

Sorular

1 **Kaba kuvvet "bebek" metninde "bek" kalıbını arıyor: her konumda soldan karşılaştırır, ilk uyuşmazlıkta kalıbı bir harf sağa kaydırır. Kalıp bulunana kadar toplam kaç karşılaştırma yapılır?**

Cevap:

2 **Metin n, kalıp m harf uzunluğunda. Kaba kuvvetin en kötü durumdaki maliyeti nedir?**

A O(n + m)
B O(n·m)
C O(n / m)
D O(m)

3 **Boyer-Moore "havlu" kalıbını arıyor (son[ ] tablosu: h 0, a 1, v 2, l 3, u 4). Kalıbın son harfinin altında metindeki v harfi var ve tutmuyor. Kalıp kaç harf kaydırılır?**

A 1
B 5
C 3
D 2

4 **"ababa" kalıbının önek tablosunu yaz (beş sayı, sırayla).**

Cevap:

5 **KMP'nin metin boyu + kalıp boyu kadar, yani O(n + m) adımda çalışmasının nedeni nedir?**

A Metindeki i yalnız artar; her karşılaştırmadan sonra ya metinde ilerlenir ya da kalıp sağa kayar.

B Kalıbı sağdan karşılaştırır ve kalıpta olmayan harflerin üstünden atlar.

C Metni bir sayının kalanına çevirip tek sayı karşılaştırır.

D Metni önce alfabe sırasına dizer.

6 **Rakamlar üstünde 3 rakamlık bir pencere kayıyor. Pencere 481 ve sıradaki rakam 5. Pencere bir sağa kayınca yeni pencerenin sayısı nedir?**

Cevap: _____

7 **Rabin-Karp'ta bir pencerenin kalanı aranan kalıbın kalanına eşit çıktı. Sonra ne yapılır?**

A Kalıp bulundu sayılır, başka bir şeye bakılmaz.

B Pencere geçilir, çünkü eşit kalan sahte eşleşme demektir.

C Harfler (rakamlar) tek tek karşılaştırılır; tutmazsa bu bir sahte eşleşmedir.

D Asal değiştirilip kalan yeniden hesaplanır.

8 **kar, kara, karga, karpuz, kartal, kaya ve kum kelimelerini tutan önek ağacına "kumru" eklenirse kaç yeni düğüm açılır?**

A 5

B 2

C 3

D 0

Cevaplar ve çözümler

0 Algoritma analizi

1. **B** · Farklı veri yapılarına

   Kayıtlar ve bilgisayar aynı, yalnız verinin bellekteki düzeni farklı; verinin bellekteki düzenine veri yapısı denir. İki dizi de aynı işleri (ekle, sil, bul) vaat ettiği için soyut veri tipi aynıdır.

2. **C** · Hiçbir satır değişmez; yalnız programın hızı değişir.

   Başlık dosyası yalnız işlemlerin adlarını verir; yapının içi ve işlemlerin kodu kaynak dosyasındadır. Kullanan program yapının içini görmediği için gerçekleme değişince onun tek satırı değişmez, yalnız hızı değişir.

3. **1400**

   $3 \times 20^2 = 3 \times 400 = 1.200$; $5 \times 20 = 100$; sabit 100. Toplam $1.200 + 100 + 100 = 1.400$.

4. **D** · $f(n) = \Omega(n^3)$

   $f(n)$ $3n^2$ ile $4n^2$ arasında kalır, yani $\Theta(n^2)$'dir. $O(n^3)$ gevşek ama doğru bir üst sınırdır. $\Omega(n^3)$ ise $f(n)$'nin bir yerden sonra n^3'ün sabit bir katının altına inmediğini söyler; n büyüdükçe n^3 çok daha hızlı büyüdüğü için bu yanlıştır.

5. **A** · Ortalama yaklaşık n/2 adımdır ve bu O(n)'dir.

   Sayı baştaysa 1, sondaysa n adım sürer; ortalaması n/2 civarındadır. 2'ye bölmek bir sabit çarpandır, büyük O sabitleri atar: ortalama da O(n)'dir. Yarıya inme ancak sıralı dizide olur.

6. **7**

   n her turda yarıya iner, küsurat atılır: 64, 32, 16, 8, 4, 2, 1, 0. 0'a inmek için 7 tur gerekir.

7. **B** · O(n log n)

   İçteki döngü j'yi her turda yarıya indirir, log n kez döner. İç içe döngüler çarpılır: dıştaki n turun her birinde log n adım, toplam O(n log n).

8. **24**

   Çağrı ağacında tepe 8 adım, altındaki 2 çağrı 4'er, sonra 4 çağrı 2'şer adım atar; tek elemanlık parçada iş yoktur. Her kat 8 tutar, kat sayısı $\log_2 8 = 3$: $3 \times 8 = 24$.

1 Listeler

1. **B** · Dinamik dizi

   Adına rağmen Python'ın listesi bir dinamik dizidir: gözleri bellekte yan yana durur, i'nci gözün adresi tek bir hesapla bulunur. Dolunca daha büyük bir blok alır.

2. **32**

   5. elemanda kapasite 4'ten 8'e, 9. elemanda 16'ya, 17. elemanda 32'ye çıkar. 17 eleman 32 gözlük blokta durur.

3. **C** · Bazı eklemeler bütün elemanları kopyalar, ama ortalama maliyet O(1)'dir.

   Büyüme seyrek gelir: 1.000.000 eklemede yalnız 18 büyüme ve toplam yaklaşık 1 milyon kopya olur, eleman başına ortalama 1 kopya. Buna amortize maliyet denir: sona ekleme ortalamada O(1)'dir.

4. **A** · Önceki düğümün sonraki adresi değişmelidir ve önceki düğümü bulmak için baştan yürümek gerekir.

   Düğüm kendinden öncekini bilmez; onu bulmak için baştan yürümek gerekir. Çift yönlü listede düğüm öncekinin adresini de taşıdığı için silme 2 atamayla, O(1)'de yapılır.

5. **47**

   sil'in ikinci satırı d→sonraki→onceki = d→onceki; yazar. d→sonraki 64, d→onceki 47'dir; yani 64'ün önceki adresi 47'yi gösterir.

6. **D** · İkisi de gözcünün kendisini gösterir.

   liste_yeni g→onceki = g→sonraki = g; yazar: boş liste kendini gösteren tek bir gözcüden ibarettir. Böylece hiçbir yerde NULL kalmaz ve uçlar için ayrı koşul gerekmez.

7. **13**

   1. adımda kaplumbağa 8'e, tavşan 13'e; 2. adımda kaplumbağa 13'e, tavşan 34'e gider. 34'ün sonrası NULL olduğu için döngü biter ve kaplumbağanın durduğu 13 döner.

8. **B** · p'nin oku geriye döndükten sonra sonraki düğümün adresi kaybolur.

   p→sonraki = once; satırı p'nin okunu geriye çevirir; ondan sonra listenin geri kalanına giden tek yol kaybolur. Bu yüzden sonraki düğümün adresi önce sonra'da saklanır.

2 Yığın ve kuyruk

1. **7 8 5**

   Son giren ilk çıkar: çıkar, en son eklenen 3'ü alır. Geriye alttan üste 5 ve 8 kalır; 7 en üste biner. Tepeden aşağı: 7 8 5.

2. **B** · 2

   Her ekle tepeyi bir artırır, her cikar bir azaltır: −1'den üç eklemeyle 2'ye, çıkarla 1'e, son eklemeyle yine 2'ye gelir. Tepe, en üstteki 53'ün indisidir.

3. **A** · Heap bölgesinde

   malloc ile alınan bellek heap bölgesindedir. Stack bölgesi, işletim sisteminin fonksiyon çağrılarının çerçeveleri için ayırdığı alandır; veri yapısı olan yığınla yalnız kuralı ve adı ortaktır.

4. **C** · [( ) { }]

   [( ) { }] satırında her kapanan tepede kendi eşini bulur ve sonda yığın boş kalır. İlk ikisinde kapanan, tepedekiyle eşleşmez; sonuncuda bir (açık kalır.

5. **14**

   5, 1, 2 yığına girer; + ile 1 + 2 = 3; 4 girer, × ile 3 × 4 = 12; + ile 5 + 12 = 17; 3 girer, − ile 17 − 3 = 14. Çıkarmada ilk çıkan (3) sağdaki sayıdır.

6. **5 1 + 2 ×**

   Açılan parantez yığına bir duvar gibi girer; + onun üstüne biner.) gelince + çıktıya alınır, duvar kalkar. Sonra × yığına girer, 2 çıktıya gider, ifade bitince × çıkar: 5 1 + 2 ×.

7. **C** · Sona eklemek

   Yeni düğüm sondakinin arkasına bağlanır; son işaretçisi yoksa sondaki düğümü bulmak için baştan sona yürümek gerekir. Son işaretçisiyle ekle de çıkar da O(1)'dir.

8. D · 1

10 4. göze girer, son 5 olur; 20 5. göze girer, son (5 + 1) % 6 = 0 olur; 30 0. göze girer, son 1 olur. Dizinin dışındaki 7. göz yoktur; son başa sarar.

3 Hash tabloları

1. 5

467'yi 11'e böl: 11 × 42 = 462, kalan 467 − 462 = 5. Anahtar 5. göze yazılır.

2. B · Aynı harflerden oluşan isimlerin toplamı aynı olur ve hepsi aynı göze düşer.

Toplama harflerin yerini hesaba katmaz: Melis ile Selim'in toplamı da 506'dır. Polinom hash her harften önce toplamı 31 gibi bir sayıyla çarptığı için harfin yeri sonucu değiştirir.

3. A · 780

Her kişi öteki 39 kişiyle çift oluşturur; her çift iki kez sayıldığı için 40 × 39 / 2 = 780. Doğum günü probleminde olasılığı hızla büyüten bu çift sayısıdır.

4. 17 3 24 10

10, 24, 3 ve 17'nin 7'ye bölümünden kalanı 3'tür; dördü de 3. göze düşer. Her yeni anahtar listenin başına geçtiği için son gelen 17 en başta, ilk gelen 10 en sonda durur: 17 3 24 10.

5. C · 3

Doluluk oranı anahtar sayısının göz sayısına oranıdır: 12 / 4 = 3. Zincirlemede ortalama liste boyu bu orana eşittir.

6. D · 11

Sonu 0 olan bir sayının 10, 20 ya da 30'a bölümünden kalanı hep 10'un katıdır: 10 gözde yalnız 0. göz, 20 gözde 0 ve 10, 30 gözde 0, 10 ve 20. gözler kullanılır. 11 asaldır, 10 ile ortak böleni yoktur; kalan her göze düşebilir.

7. 1

Üçünün de kalanı 10'dur. 21 10. göze yazılır. 32 için 10 dolu, sonraki göz başa dönerek 0'dır: 32 0. göze. 43 için 10 ve 0 dolu, 1. göze yazılır.

8. C · Arama o boş gözde durur ve ondan sonraki gözlere yoklamayla yerleşmiş anahtarları bulamayabilir.

Arama boş bir göze varınca anahtarın tabloda olmadığına karar verir. 49'u silip gözünü boşaltınca 82'yi arayan 6. gözde durdu ve 7. gözdeki 82'yi bulamadı. Bu yüzden silinen göz "silindi" diye işaretlenir.

4 Ağaçlar ve ikili ağaç

1. B · 3

Yükseklik, en derindeki yaprağın kökten kaç kenar aşağıda olduğudur. En derin yaprak F'dir: A-B, B-E, E-F, yani 3 kenar. 6 düğüm sayısıdır, 4 ise kat sayısıdır.

2. C · Ağaç doludur ama mükemmel değildir.

Her düğümün 0 ya da 2 çocuğu olması dolu ağacın tanımıdır. Mükemmel ağaçta bütün seviyeler dolu olduğu için bütün yapraklar aynı derinlikte olurdu; burada öyle değil.

3. 31

Seviyeler en çok 1, 2, 4, 8 ve 16 düğüm alır (derinlik 0'dan 4'e). Toplam 2^(4+1) − 1 = 32 − 1 = 31.

4. **D** · 10

   L kademe en çok 2^L − 1 kişi alır. 9 kademe 511 kişide kalır, 10 kademe 1.023 kişiyi alır; 1.000 kişi için en az 10 kademe gerekir.

5. **C D A E B K**

   Postorder her düğümde önce sol alt ağacı, sonra sağ alt ağacı, en sonda düğümün kendisini işler. A'nın alt ağacından C D A, B'nin alt ağacından E B çıkar, en sonda kök K gelir: C D A E B K.

6. **A** · Postorder

   Önce çocuklar, en sonda düğüm geri verilmelidir. Düğüm önce free edilirse sol ve sağ çocuklarının adresi onunla birlikte kaybolur.

7. **B** · Kuyruk: ilk giren ilk çıkar.

   Kök kuyruğa girer; her adımda baştaki çıkar ve çocukları sona eklenir. Kuyruk geliş sırasını koruduğu için bir kat bitmeden alttakine geçilmez.

8. **24**

   Değer postorder ile bulunur: önce + düğümü 6 + 2 = 8 verir, sonra kök 8 × 3 = 24 verir. Inorder yazım 6 + 2 × 3 parantezi kaybettiği için 12 sonucunu verirdi.

5 İkili arama ağacı

1. **D** · Kök 50; sol çocuğu 30, 30'un sağ çocuğu 60.

   Kural yalnız çocuklar için değil, bütün alt ağaç için geçerlidir. 60, 30'un sağında durabilir ama 50'nin sol alt ağacındadır ve 50'den büyüktür; bu yüzden kural bozulur.

2. **3**

   Arama kökten başlar: 60, 50'den büyük, sağa; 70'ten küçük, sola; üçüncü bakışta 60 bulunur.

3. **10 20 30 40 60**

   Inorder her düğümde önce solu, sonra düğümü, en son sağı yazar. Soldakiler küçük, sağdakiler büyük olduğu için sayılar sıralama yapılmadan küçükten büyüğe çıkar: 10 20 30 40 60.

4. **B** · 10 20 30 40 50

   Küçükten büyüğe gelen her sayı öncekilerin hepsinden büyüktür ve hep en sağa eklenir; ağaç 5 katlı bir zincir olur. Ötekilerde ilk gelen 30 ortancadır, ağaç iki yana dallanır.

5. **C** · O(n)

   Arama kat sayısı kadar sürer. Sayılar küçükten büyüğe eklenmişse ağaç n katlı bir zincirdir ve arama n adım sürer. O(log n) yalnız katları dolu, dengeli ağaçta geçerlidir.

6. **A** · Düğümün çocuğu, düğümün yerine geçer; ebeveyn adresini doğrudan çocuğa bağlar.

   Çocuğun alt ağacı zaten silinen düğümle aynı tarafta durduğu için çocuk onun yerine geçince kural bozulmaz. Örnekte 32 silinince 51, sol adresini doğrudan 40'a bağladı.

7. **30**

   25'in iki çocuğu var: 15 ve 35. Ardıl, sağ alt ağacın en küçüğüdür: bir kez sağa (35), sonra hep sola (30). 30'un solu boş olduğu için durulur; 30, 25'in yerine yazılır ve eski yerinden silinir.

8. **D** · Kökten başlayıp solda kimse kalmayana kadar hep sola inilir.

   Küçükler hep solda olduğu için kökten sola inilir ve solu NULL olan düğümde durulur. Kökün sol çocuğunun da solunda daha küçük sayılar olabilir; adım sayısı kat sayısını geçmez.

6 Dengeli ağaçlar

1. **B** · Her düğümde sol ve sağ alt ağacın yükseklikleri en çok 1 farklı olabilir.
   AVL ağacı her düğümde iki alt ağacın yüksekliklerini karşılaştırır; fark 2'ye çıkınca dönmeyle düzeltir. Siyah düğüm kuralı kırmızı-siyah ağacın, yaprakların aynı derinlikte olması B-ağacının özelliğidir.

2. **C** · Kol içe kırık olduğu için çift dönme gerekir; kök 20 olur.
   10, 30'un soluna, 20 de 10'un sağına iner: zikzak. 30'da fark 2'dir ve sol kol içe kırıktır. Önce 10'da sola dönülür, sonra 30'da sağa; ortanca 20 tepeye çıkar, 10 solunda, 30 sağında durur.

3. **2 1 4 3 5**
   1 gelince 3'te fark 2 olur; sağa dönmeyle ağaç 2(1, 3) olur. 4, 3'ün sağına iner. 5 gelince 3'te fark −2 olur; sola dönmeyle 4 yukarı çıkar: 2(1, 4(3, 5)). Seviye seviye: 2 1 4 3 5.

4. **D** · 5'in sağ çocuğu olur.
   Sola dönmede 8 yukarı çıkar, 5 onun soluna iner. Yukarı çıkan düğümün iç taraftaki dalı aşağı inen düğüme geçer: 6, 5'ten büyük ve 8'den küçük olduğu için 5'in sağ çocuğu olur.

5. **A** · Kırmızı-siyah ağaç daha az döner; sürekli ekleyip silen programlarda dengeyi korumak ucuzdur.
   Kırmızı-siyah ağaç daha gevşek dengelidir: yüksekliği en çok 2 log n, AVL'ninki en çok 1,44 log n. Buna karşılık bir silme en çok 3 dönme ister, AVL'de ise köke kadar dönme gerekebilir.

6. **D** · Kırmızı bir düğümün çocuğu kırmızı olamaz.
   Kırmızı-siyah ağaçta iki kırmızı art arda gelemez. Bu kural en uzun yolun en kısanın iki katını geçmemesini sağlar.

7. **30**
   İlk dört anahtar tek düğüme sıralı dizilir: [10 20 30 40]. 50 gelince düğüm taşar: [10 20 30 40 50]. Ortanca 30 yukarı çıkar ve yeni kök olur; solunda [10 20], sağında [40 50] kalır.

8. **999999**
   Katlarda 1, 100 ve 10.000 düğüm vardır; toplam 10.101 düğüm × 99 anahtar = 999.999. Kısa yol: k katlı, 100 dallı ağaç en çok 100^k − 1 anahtar alır; 100^3 − 1 = 999.999.

7 Heap ve öncelikli kuyruk

1. **C** · En küçük eleman köktedir, yani dizinin 0. gözündedir.
   Kural yalnız ebeveynle çocuk arasındadır: her düğüm çocuklarından küçük ya da onlara eşittir; bu yüzden en küçük köke yerleşir. Kardeşler arasında sıra aranmaz: 4 10 7 15 12 9 21 sıralı değildir, kökün solunda 10, sağında 7 durur. En büyük eleman ise yapraklardan herhangi birinde olabilir.

2. **A** · Tam ikili ağaç olması: katlar soldan boşluksuz dolduğu için i. gözün çocukları 2i + 1 ve 2i + 2. gözlerdedir.
   Tam ağaçta boşluk olmadığı için düğümler kat kat, soldan sağa diziye yazılınca her düğümün yeri numarasından hesaplanır: çocuklar 2i + 1 ve 2i + 2, ebeveyn (i − 1) / 2.

3. **5**
   Ebeveyn (i − 1) / 2 ile bulunur, küsurat atılır: (11 − 1) / 2 = 5. Sağlama: 5. gözün çocukları 2 × 5 + 1 = 11 ve 2 × 5 + 2 = 12'dir.

4. **D** · 13
   10.000 eleman 14 kata sığar; kökten en alttaki yaprağa 13 adım vardır. Yeni eleman en çok köke kadar çıkar, yani en çok 13 takas yapılır. 14 kat sayısıdır, adım sayısı değil.

5. **3 4 7 10 12 9 21 15**

   3 son göze (7. göz) konur. Ebeveyni 3. gözdeki 15'tir: takas, 4 10 7 3 12 9 21 15. Yeni ebeveyn 1. gözdeki 10: takas, 4 3 7 10 12 9 21 15. Sonra 0. gözdeki 4 de 3'ten büyüktür: takas, 3 4 7 10 12 9 21 15. 3 köke ulaştı.

6. **B** · O(n)

   Düğümlerin yarısı yapraktır ve hiç inmez; dörtte biri en çok 1, sekizde biri en çok 2 kat iner. Toplam iniş sayısı n'yi geçmez, bu yüzden heap kurmak O(n)'dir. O(n log n), elemanları tek tek eklemenin en kötü durumudur.

7. **C** · 63 54 26 38 49 15

   71 alınır, son eleman 15 köke gelir: 15 54 63 38 49 26. 15 büyük çocuğu 63'le yer değiştirir: 63 54 15 38 49 26. 2. gözdeki 15'in tek çocuğu 26'dır (5. göz) ve ondan küçüktür: bir takas daha, 63 54 26 38 49 15. İlk seçenekte indirme yarıda bırakılmıştır.

8. **50 40 30 10 20**

   Son ebeveyn 5 / 2 − 1 = 1. gözdür. 1. gözdeki 50, çocukları 10 ve 40'tan büyük: iniş yok. Sonra kökteki 20, büyük çocuğu 50'yle yer değiştirir: 50 20 30 10 40. 1. göze inen 20'nin büyük çocuğu 40'tır: takas, 50 40 30 10 20.

8 Sıralama algoritmaları

1. **B** · Eşit anahtarlı kayıtların aralarındaki eski sırayı korur.

   Kararlılık eşit anahtarlıların sırasıyla ilgilidir: ada göre sıralı listeyi şehre göre kararlı bir yöntemle sıralayınca aynı şehirdekiler alfabetik kalır. Hız ve ek bellek ayrı özelliklerdir.

2. **C** · Liste yine şehre göre sıralanır, ama eşitlikte sağdaki alındığı için aynı şehirdekilerin sırası bozulur.

   Eşitlikte artık soldaki değil sağdaki kayıt alınır. Şehirler yine sıralı çıkar, ama örnekte Figen Burcu'nun, Derya Can'ın, Emre Arda'nın önüne geçer: sıralama kararsız olur.

3. **D** · Dizinin kendisi kadar ikinci bir diziye, yani O(n) ek belleğe ihtiyaç duyar.

   Birleştirme, sonucu yandaki ikinci diziye yazıp sonra asıl dizinin üstüne kopyalar; 10 milyon int için 40 MB ek yer gerekir. Süresi her girdide O(n log n)'dir ve eşitlikte soldakini aldığı için kararlıdır.

4. **30 10 20 50 80 60**

   Pivot 50, sınır 0. gözde. 30 küçük: kendisiyle yer değiştirir, sınır 1. 80 geçilir. 10 küçük: sınırdaki 80'le yer değiştirir (30 10 80 60 20 50), sınır 2. 60 geçilir. 20 küçük: sınırdaki 80'le yer değiştirir (30 10 20 60 80 50), sınır 3. Son olarak pivot sınırdaki 60'la yer değiştirir: 30 10 20 50 80 60.

5. **B** · $O(n^2)$

   Sondaki eleman hep en büyüktür; pivot hiçbir şeyi bölmez ve her turda yalnız bir sayı yerine oturur. Karşılaştırmalar $(n-1) + (n-2) + \dots + 1 = n(n-1)/2$ eder: 10.000 sayıda 49.995.000. Rastgele pivot ya da üçün ortancası bunu önler.

6. **7**

   5 paket 5! = 120 farklı sırada olabilir. Her tartı olasılıkları en iyi ihtimalle yarıya indirir: $2^6 = 64$ yaprak yetmez, $2^7 = 128$ yeter. Bu yüzden en az 7 tartı gerekir.

7. **C** · Değerlerin alabileceği aralık çok büyükse, örneğin sayılar 1 milyara kadar çıkabiliyorsa.

   Sayma sıralaması her olası değer için bir sayaç tutar ve O(n + k) sürer. k küçükse çok hızlıdır, ama değerler 1 milyara kadar çıkabiliyorsa 1 milyar sayaç gerekir. Eşit değerler sorun değildir; sayma sıralaması onların sırasını da korur.

8. **540 431 213 325 207 118**

   Birler basamakları 1, 7, 5, 8, 0, 3. Küçükten büyüğe: 0 → 540, 1 → 431, 3 → 213, 5 → 325, 7 → 207, 8 → 118. Sonuç 540 431 213 325 207 118.

9 Graflar

1. **C** · 9

   Bağlı ve döngüsüz graf bir ağaçtır; n düğümlü her ağaçta n − 1 kenar vardır: 10 − 1 = 9. 6 kişilik örnekte de 6 düğümlü ağaçta 5 kenar vardı. 45, 10 düğüm arasındaki bütün olası kenarların sayısıdır.

2. **1000000**

   Komşuluk matrisi düğüm sayısının karesi kadar göz tutar: 1.000 × 1.000 = 1.000.000. Grafta kaç kenar olduğu bu sayıyı değiştirmez.

3. **A** · 16

   Yönsüz grafta her kenar iki düğümü birbirine bağladığı için iki listeye birden girer: 8 × 2 = 16 kayıt. 6 kişilik örnekte de 6 kenar 12 kayıt tutuyordu.

4. **B** · Komşuluk listesi, çünkü graf seyrektir; matris 1 trilyon gözün neredeyse hepsini boş tutardı.

   Matris V^2 = 1 trilyon göz ister ve gözlerin çok küçük bir kısmı dolu olur. Liste yalnız var olan yolları yazar, yeri V + E kadardır. Seyrek büyük graflar bu yüzden çoğu zaman listeyle saklanır.

5. **D** · Kuyruk, çünkü ilk giren ilk çıktığı için bir dalga bitmeden sonrakine geçilmez.

   Kuyruk ilk gireni ilk çıkarır: 1 adımdaki düğümler 2 adımdakilerden önce girdiği için önce onlar işlenir. Bu yüzden bir düğüme ilk yazılan uzaklık en kısa yoldur. Yığın derinlik öncelikli aramanın yapısıdır.

6. **3**

   A'ya 0 yazılır. A'nın komşuları B ve C 1 alır. Onların numarasız komşusu D 2 alır (C'den ikinci kez varılınca D zaten numaralıdır). D'nin komşusu E 3 alır.

7. **C** · Hâlâ yolda olan, yani ön koşulları bitmemiş bir derse yeniden varmak.

   Yoldaki bir derse yeniden varılması, o dersin kendi ön koşullarından birinin yine onu beklediğini gösterir: dersler birbirini bekler ve sıra kurulamaz. Bitmiş derse varmak sorun değildir, örnekte Veri Yapıları'na ikinci kez inilmedi.

8. **A B C D**

   dfs(D) önce B'ye, B de A'ya iner; A'nın ön koşulu yok, ilk A yazılır, sonra B. D'nin ikinci ön koşulu C'ye inilir; C'nin ön koşulu A bitmiştir, inilmez; C yazılır. En sonda D. Sıra A B C D.

10 Minimum yayılan ağaç

1. **B** · 11

   Yayılan ağaçta ilk köyden sonra her hat ağaca bir köy daha katar; V köy için V − 1 hat gerekir: 12 − 1 = 11. 12 hat olsaydı bir döngü oluşurdu; 66, her köyden her köye çekilebilecek bütün hatların sayısıdır.

2. **A** · Dereköy–Pınarbaşı 7 km

   Kesitten Dereköy'ün üç hattı geçer; en kısası 7 km'lik Dereköy–Pınarbaşı hattıdır ve kesit özelliğine göre en kısa düzende mutlaka vardır. Grupların büyüklüğü önemli değildir; tek köylük bir grup da geçerli bir kesittir.

3. **10 4 3**

   Tepecik'in hatları 10, 14, 16: Pınarbaşı 10 km'yle katılır. Grubun dışına çıkan hatlar 14, 16, 8, 5, 7, 4: Yazıköy 4 km'yle katılır. Sonra adaylara Yazıköy–Ovacık 3 eklenir ve en kısası odur: Ovacık 3 km'yle katılır.

4. **C** · Kayıt atılır, çünkü o hat bir döngü kurar.

   Köy ağaçtaysa o hat iki ağaç köyünü bağlar ve döngü kurar; kodda if (agacta[o.dugum]) continue; satırı bu kaydı atlar. Örnekte Çamlı–Pınarbaşı 8 ve Çamlı–Dereköy 9 böyle atıldı.

5. **D** · O(E log E)

   Her hat kuyruğa en çok bir kez girer ve bir kez çıkar; her giriş ya da çıkış log E kadar adım tutar. Toplam O(E log E). O(2^E) bütün düzenleri denemenin büyüme hızına yakındır.

6. **B** · 1 ile 2, 3 üzerinden zaten aynı parçadadır; kablo döngü kurar.

   75 m'den önce 0–1, 2–3, 1–3 ve 4–5 döşenmiştir; 1, 3 üzerinden 2'ye bağlıdır. bul(1) ve bul(2) aynı başı, 3'ü verir; iki ucu aynı parçada olan kablo döngü kurduğu için atlanır.

7. **5 5 5 3 3 5**

   bul(0) okları izler: 0 → 1 → 2 → 5; 5 kendini gösterdiği için baştır. Dönüşte geçilen her göze başın numarası yazılır: ebeveyn[2], ebeveyn[1] ve ebeveyn[0] 5 olur. 3 ve 4'ün gözlerine dokunulmaz.

8. **C** · Kabloların baştaki sıralaması

   Ölçümde bütün bul çağrıları toplam 224.662 ok izledi; sıralama ise 1.079.524 karşılaştırma tuttu. Bu yüzden Kruskal'ın maliyeti sıralamanınki kadardır: O(E log E).

11 En kısa yol

1. **B** · Yol sayısını en aza indirir; yolların süreleri farklıysa az yollu güzergâh uzun sürebilir.

   BFS önce bir yol ötedeki, sonra iki yol ötedeki kavşaklara bakar; bulduğu, en az yollu güzergâhtır. Örnekte Ev → Köprü → Stadyum 2 yol, 19 dakika; 4 yollu güzergâh ise 12 dakika.

2. **4**

   A 0 ile kesinleşir: C 1, B 4. C kesinleşir: B için 1 + 2 = 3 < 4, D için 1 + 5 = 6. B 3 ile kesinleşir: D için 3 + 1 = 4 < 6. D 4 ile kesinleşir; yol A → C → B → D.

3. **C** · Köprü'nün süresi 8 olur ve önceki kavşağı Park diye not edilir.

   Gevşetme, uz[u] + ağırlık < uz[v] koşuluna bakar: 5 + 3 = 8 < 9. Köprü'ye 8 yazılır ve once[Köprü] = Park olur. Köprü ancak kesinleşmemişler arasında en küçük olduğunda kesinleşir.

4. **A** · 100.000.000

   Kavşak sayısı kadar seçim yapılır ve her seçimde bütün kavşaklara bakılır: V × V = 10.000 × 10.000 = 100.000.000. Maliyet $O(V^2)$.

5. **D** · Han zaten kesinleşmiş olduğu için kayıt atlanır.

   Eski kaydı hipin içinde arayıp silmek zahmetli olduğu için bırakılır. Sırası gelince if (bitti[u]) continue; satırı onu atlar; Han'ın süresi 3 kalır.

6. **A** · Tahmin, gerçek kalan yoldan hiç uzun olmamalıdır.

   A* bir kavşağın sırasını şimdiye kadarki süre ile tahmini toplayarak belirler. Tahmin gerçek kalanı aşmadıkça bulunan yol en kısadır; kuş uçuşu uzaklık bu koşulu sağlar.

7. 6

1. turda A→T ve B→A gevşetilemez (A ve B henüz sonsuz); sonra A 4, B 3 olur. 2. turda önce A→T: T = 4 + 2 = 6; sonra B→A: 3 − 2 = 1 < 4, A 1 olur. T'nin 3'e inmesi 3. turu bekler.

8. B · V − 1 turdan sonra bir tur daha yapar; hâlâ kısalan bir değer varsa eksi döngü vardır.

Eksi döngü yoksa en kısa yollar en çok V − 1 yol kullanır ve V − 1 turda oturur. Bir tur daha değer kısaltıyorsa yol döndükçe kısalıyordur. Tek bir eksi yol döngü demek değildir: doğru haritada Tepe→Vadi −5 vardı ama eksi döngü yoktu.

12 Metin arama

1. 7

1. konumda b ve e tutar, üçüncü harf b ≠ k: 3 karşılaştırma. 2. konumda e ≠ b: 1. 3. konumda b, e, k tutar: 3. Toplam 3 + 1 + 3 = 7.

2. B · O(n·m)

Tekrarlı metinde (hepsi a) ve kalıpta (a…ab) her konumda kalıbın neredeyse hepsi tutar, yalnız son harf bozulur; her konumda m karşılaştırma yapılır. 1 milyon harf ve 1.000 harflik kalıpta 999.001.000 karşılaştırma.

3. D · 2

Uyuşmazlık kalıbın son gözünde, j = 4'te çıktı; v'nin kalıptaki en sağdaki yeri son['v'] = 2. Kayma j − son[c] = 4 − 2 = 2: kalıbın kendi v'si metindeki v'nin altına gelir. 5 harf, harf kalıpta hiç olmasaydı atlanırdı.

4. 0 0 1 2 3

Her başlangıç parçası için hem başta hem sonda duran, parçadan kısa en uzun kısım: a → 0; ab → 0; aba → a, 1; abab → ab, 2; ababa → aba, 3.

5. A · Metindeki i yalnız artar; her karşılaştırmadan sonra ya metinde ilerlenir ya da kalıp sağa kayar.

KMP önek tablosu sayesinde tutan harfleri yeniden okumaz; metinde hiç geri dönmez. İki adım türünün her biri en çok n kez olur: arama en çok 2n karşılaştırma, tablo da O(m). Sağdan atlamak Boyer-Moore'un, kalan karşılaştırmak Rabin-Karp'ın yoludur.

6. 815

Baştaki 4'ün payını çıkar: 481 − 400 = 81. 10'la çarp: 810. Yeni rakamı ekle: 810 + 5 = 815.

7. C · Harfler (rakamlar) tek tek karşılaştırılır; tutmazsa bu bir sahte eşleşmedir.

Kalan farklıysa pencere kesinlikle farklıdır; ama kalan aynıyken sayılar farklı olabilir. Örnekte 726 ve 934'ün 13'e bölümünden kalanı da 11'dir: 726'da rakamlar tutmadı, 934'te tuttu.

8. B · 2

k, u, m yolu kum için zaten var; ortak başlangıç tek yoldur. Yalnız r ve u için 2 yeni düğüm açılır ve son düğüm kelime sonu diye işaretlenir.

Öğrendiğini pekiştir

Bu notlardaki bütün soruları ve daha fazlasını sitede, her cevabın hemen ardından çözümüyle birlikte çözebilirsin.

hekzon.com/dersler

HEKZON