# Bölüm 3: Arama Algoritmaları

## Algoritmalar

Arama problemi | Doğrusal arama | İkili arama | Hash ile arama | Ara değer arama | Atlayarak arama | Üstel arama | Karşılaştırma

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

GİRİŞ

# Öğrenme Çıktıları

**Temel aramalar**

Doğrusal ve ikili aramayı (ikiliyi yinelemeli ve özyinelemeli) Java ile yazmak, bir dizi üzerinde adım adım izlemek; en iyi, ortalama ve en kötü durumlarını hesaplamak.

**Hash ile arama**

Hash fonksiyonu, kova, çakışma, doğrusal yoklama ve zincirleme kavramlarını açıklamak; yük faktörünün maliyete etkisini yorumlamak.

**Sıralı dizide gelişmiş aramalar**

Ara değer, atlayarak ve üstel aramanın fikrini, kodunu ve O(log log n), O(√n), O(log i) karmaşıklıklarını gerekçelendirmek.

**Seçim**

Verinin sıralı olup olmamasına, dağılımına ve boyutuna göre uygun arama algoritmasını seçmek; yaygın uygulama hatalarından kaçınmak.

GİRİŞ

# Arama Problemi

- Arama algoritmaları, bir veri kümesinde **istenen değeri** bulmak için kullanılır.
- Verileri bir **arama ölçütüne** (çoğunlukla bir **anahtar**, key) göre tarar ve eşleşen öğeyi bulmaya çalışır.

**Girdi**
n öğeli bir `dizi` ve bir `aranan` değer.

**Çıktı**
`dizi[i] == aranan` olan bir **indis** i; aranan yoksa **−1** (başarısız arama).

**Maliyet ölçüsü:** aramalarda süre, genellikle aranan değerin dizideki öğelerle kaç kez **karşılaştırıldığı** (yoklama, probe sayısı) ile ölçülür. Bu bölümdeki tüm karmaşıklıklar bu sayıyı verir.

GİRİŞ

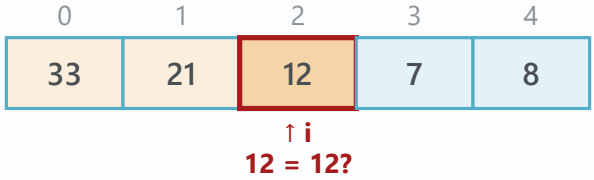
# Arama Algoritmalarının Çeşitleri

Farklı arama algoritmaları, farklı çalışma prensiplerine ve önkoşullara sahiptir.

| Algoritma | Önkoşul | Çalışma prensibi |
|---|---|---|
| Doğrusal (Linear Search) | yok | Verileri tek tek karşılaştırarak arar. |
| İkili (Binary Search) | sıralı dizi | Veri kümesini ikiye bölerek arama alanını daraltır. |
| Hash (Hash Search) | hash tablosu | Veriler bir hash tablosuna yerleştirilir; arama, anahtarın kovasında yapılır. |
| Ara değer (Interpolation) | sıralı, düzgün dağılımlı | Konumu, değerlerin oranından tahmin eder. |
| Atlayarak (Jump Search) | sıralı dizi | √n büyüklüğünde bloklar atlar, sonra blok içinde doğrusal arar. |
| Üstel (Exponential) | sıralı dizi | Sınırı 1, 2, 4, 8, … diye büyütür, sonra ikili arama yapar. |

# Doğrusal Arama (Linear Search)

- Listedeki **her öğeyi tek tek** kontrol ederek arama yapar.
- Alışveriş listesindeki ürünleri sırayla kontrol etmeye ya da kayıp bir eşyayı bulmak için odayı **sistematik** olarak taramaya benzer.
- Aranan değer, listenin **başından** başlanarak her öğe ile karşılaştırılır:
  - bulunursa **konumu** (indisi) döndürülür;
  - listede yoksa başarısız sonuç **−1** döndürülür.
- Bazen aranan bilgiyi bulmak için tek tek bakmak **en iyi** yoldur: dizi sırasızsa başka seçenek yoktur.



# Doğrusal Arama Algoritması

```
int dogrusalArama(int[] dizi, int n,
                  int aranan) {
  int i = 0;
  while (i < n && dizi[i] != aranan)
    i++;
  if (i < n)
    return i;      // bulundu: indis
  else
    return -1;     // bulunamadı
}
```

- **Parametreler:** `dizi` aranacak öğeler, `n` dizinin boyutu, `aranan` aranacak öğe.
- **Dönüş:** aranan öğenin indisi; bulunamazsa −1.
- `i` dizideki öğeler üzerinde gezinir; `while` döngüsü **i < n** ve **dizi[i] ≠ aranan** olduğu sürece devam eder, her turda i bir artar.
- Döngüden sonra `if`, öğenin bulunup bulunmadığını denetler: **i < n** ise i (öğenin indisi), değilse −1 döner.

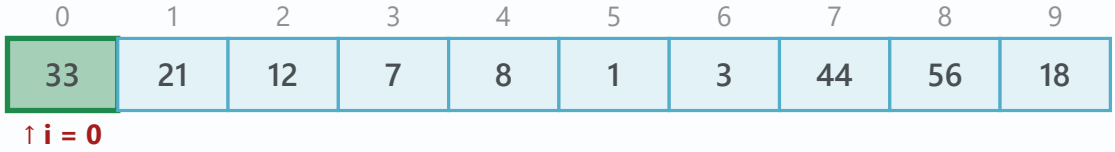
Koşulların sırası önemlidir: `i < n` önce sınanır; Java'nın kısa devre `&&` işleci sayesinde `i == n` olduğunda `dizi[n]`'e hiç erişilmez.

# Doğrusal Arama: En İyi ve En Kötü Durum

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 33 | 21 | 12 | 7 | 8 | 1 | 3 | 44 | 56 | 18 |

**En iyi durum: 33 aranır.** Dizi sırasızdır: 33, 21, 12, 7, 8, 1, 3, 44, 56, 18 (n = 10).

# Doğrusal Arama: En İyi ve En Kötü Durum



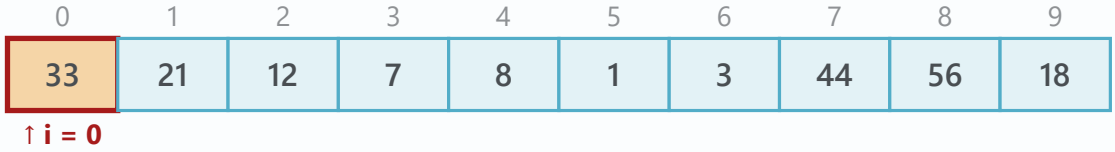

dizi[0] = 33 = aranan → döngü hemen biter; **0 döner**. Yalnızca **1 karşılaştırma**: en iyi durum O(1).

# Doğrusal Arama: En İyi ve En Kötü Durum

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 33 | 21 | 12 | 7 | 8 | 1 | 3 | 44 | 56 | 18 |

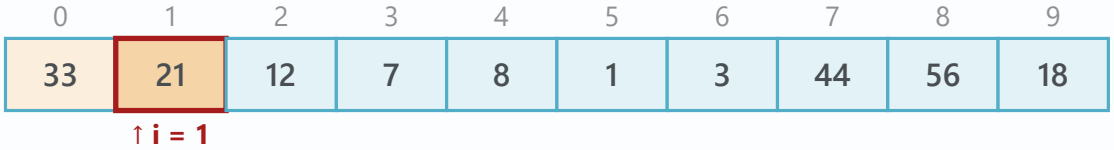
**En kötü durum: 17 aranır.** 17 dizide yoktur; her öğeye bakmak gerekir.

# Doğrusal Arama: En İyi ve En Kötü Durum



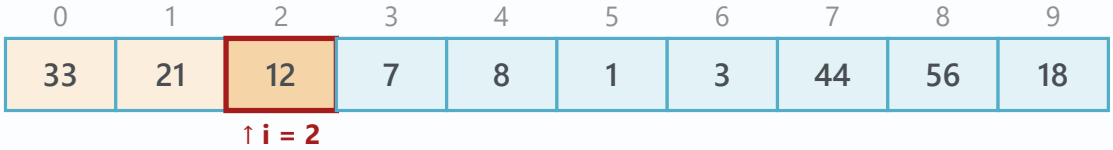

i = 0: dizi[0] = 33 ≠ 17 → i = 1.

# Doğrusal Arama: En İyi ve En Kötü Durum



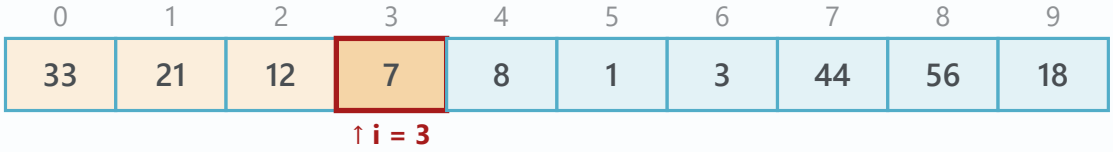

i = 1: dizi[1] = 21 ≠ 17 → i = 2.

# Doğrusal Arama: En İyi ve En Kötü Durum



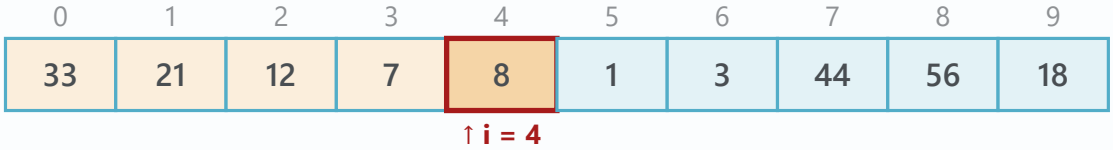

i = 2: dizi[2] = 12 ≠ 17 → i = 3.

# Doğrusal Arama: En İyi ve En Kötü Durum



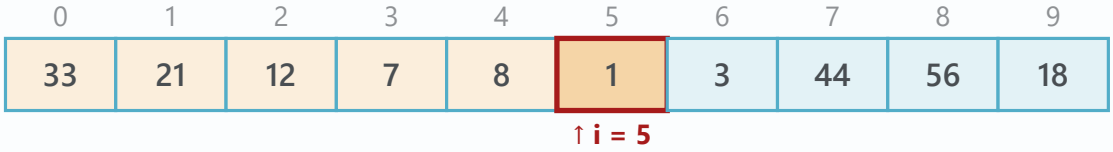

i = 3: dizi[3] = 7 ≠ 17 → i = 4.

# Doğrusal Arama: En İyi ve En Kötü Durum



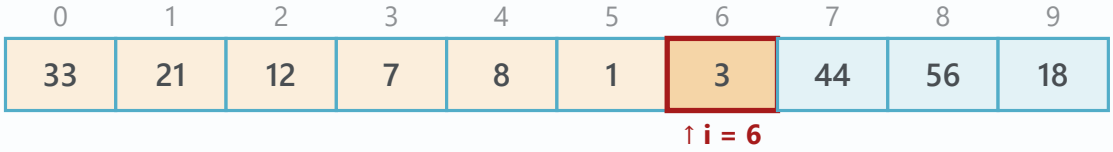

i = 4: dizi[4] = 8 ≠ 17 → i = 5.

# Doğrusal Arama: En İyi ve En Kötü Durum



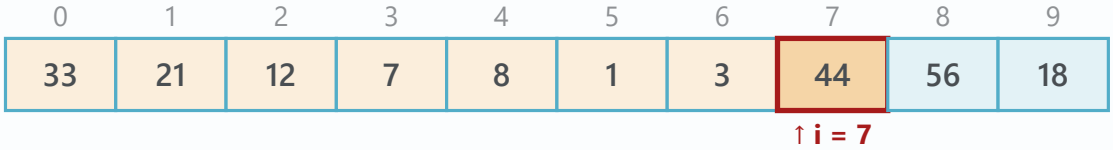

i = 5: dizi[5] = 1 ≠ 17 → i = 6.

# Doğrusal Arama: En İyi ve En Kötü Durum



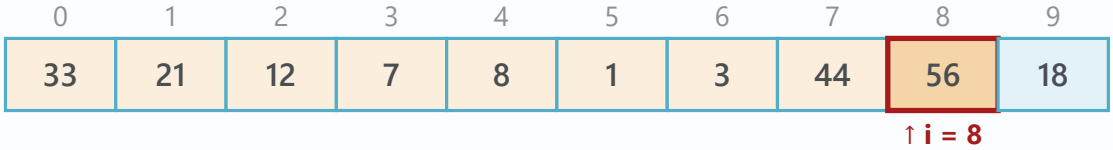

i = 6: dizi[6] = 3 ≠ 17 → i = 7.

# Doğrusal Arama: En İyi ve En Kötü Durum



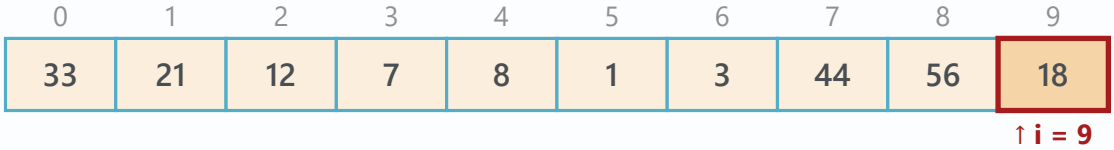

i = 7: dizi[7] = 44 ≠ 17 → i = 8.

# Doğrusal Arama: En İyi ve En Kötü Durum



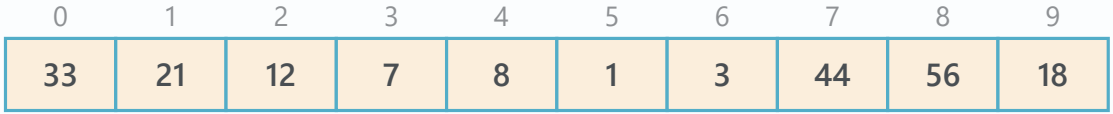

i = 8: dizi[8] = 56 ≠ 17 → i = 9.

# Doğrusal Arama: En İyi ve En Kötü Durum




i = 9: dizi[9] = 18 ≠ 17 → i = 10.

# Doğrusal Arama: En İyi ve En Kötü Durum




i = 10 = n → döngü biter; `i < n` yanlış, **−1 (bulunamadı) döner**. **n = 10 karşılaştırma**: en kötü durum O(n).

DOĞRUSAL ARAMA · KODU ADIM ADIM ÇALIŞTIR

# dogrusalArama(dizi, 10, 1): Kodu Adım Adım İzle

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 33 | 21 | 12 | 7 | 8 | 1 | 3 | 44 | 56 | 18 |

↑ i

`i` = 0 ile başlanır. Aranan: 1 (**ortalama durum** örneği).

Adım 1 / 13

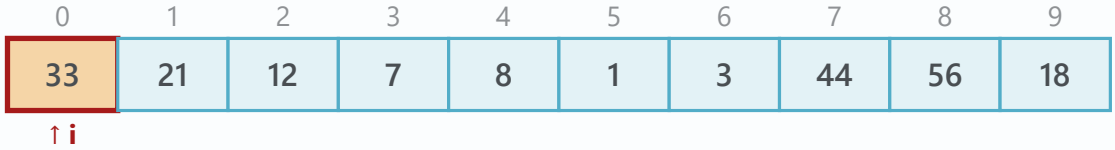
```
1  int dogrusalArama(int[] dizi, int n,
2                    int aranan) {
▶    int i = 0;
4    while (i < n && dizi[i] != aranan)
5      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

DEĞİŞKENLER

`n = 10` `aranan = 1` `i = 0`

ÇIKTI

# dogrusalArama(dizi, 10, 1): Kodu Adım Adım İzle




i < n ve dizi[0] = 33 ≠ 1 → döngü gövdesi çalışır.

Adım 2 / 13

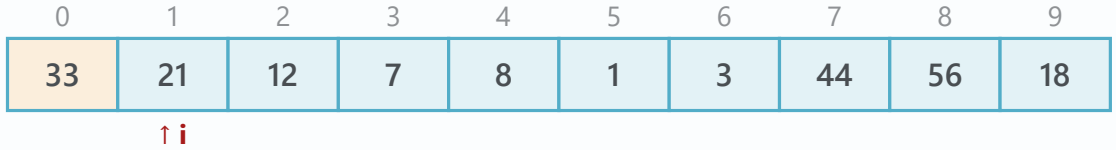
```
1  int dogrusalArama(int[] dizi, int n,
2                    int aranan) {
3    int i = 0;
▶    while (i < n && dizi[i] != aranan)
5      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

**DEĞİŞKENLER**

`n = 10` `aranan = 1` `i = 0`

**ÇIKTI**

# dogrusalArama(dizi, 10, 1): Kodu Adım Adım İzle




`i++` → i = 1.

Adım 3 / 13

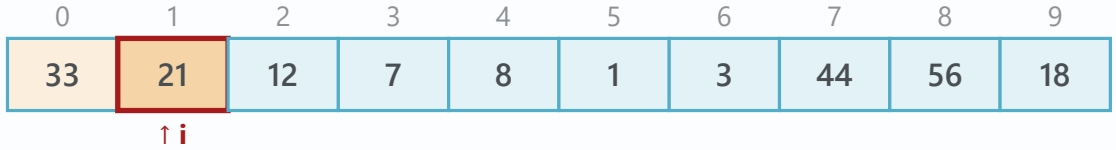
```
1  int dogrusalArama(int[] dizi, int n,
2                    int aranan) {
3    int i = 0;
4    while (i < n && dizi[i] != aranan)
▶      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

**DEĞİŞKENLER**

`n = 10` `aranan = 1` `i = 1`

**ÇIKTI**

# dogrusalArama(dizi, 10, 1): Kodu Adım Adım İzle




i < n ve dizi[1] = 21 ≠ 1 → döngü gövdesi çalışır.

Adım 4 / 13

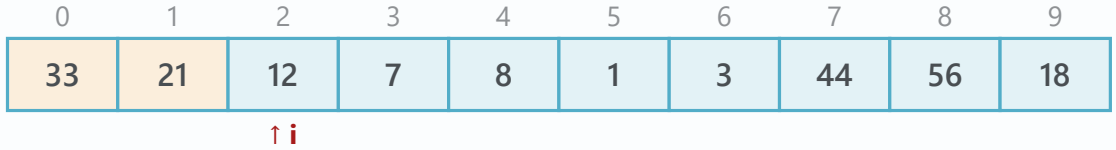
```
1  int dogrusalArama(int[] dizi, int n,
2                    int aranan) {
3    int i = 0;
▶    while (i < n && dizi[i] != aranan)
5      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

**DEĞİŞKENLER**

`n = 10` `aranan = 1` `i = 1`

**ÇIKTI**

# dogrusalArama(dizi, 10, 1): Kodu Adım Adım İzle




`i++` → i = 2.

Adım 5 / 13

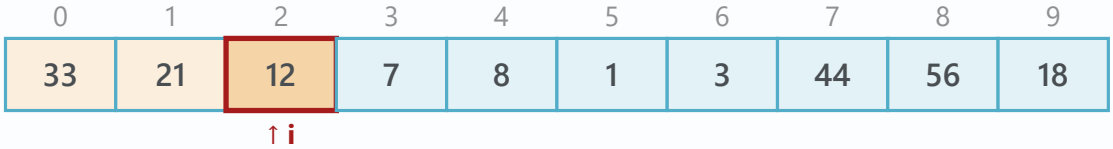
```
1  int dogrusalArama(int[] dizi, int n,
2                    int aranan) {
3    int i = 0;
4    while (i < n && dizi[i] != aranan)
▶      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

**DEĞİŞKENLER**

`n = 10` `aranan = 1` `i = 2`

**ÇIKTI**

# dogrusalArama(dizi, 10, 1): Kodu Adım Adım İzle




i < n ve dizi[2] = 12 ≠ 1 → döngü gövdesi çalışır.

Adım 6 / 13

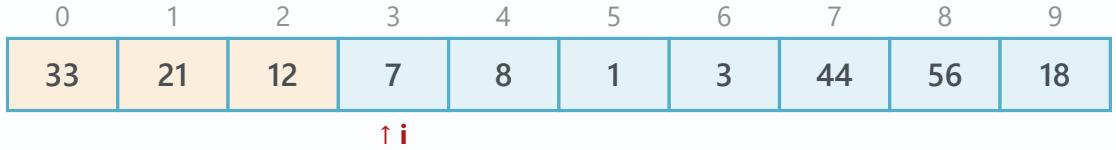
```
1  int dogrusalArama(int[] dizi, int n,
2                    int aranan) {
3    int i = 0;
▶    while (i < n && dizi[i] != aranan)
5      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

**DEĞİŞKENLER**

`n = 10` `aranan = 1` `i = 2`

**ÇIKTI**

# dogrusalArama(dizi, 10, 1): Kodu Adım Adım İzle




`i++` → i = 3.

Adım 7 / 13

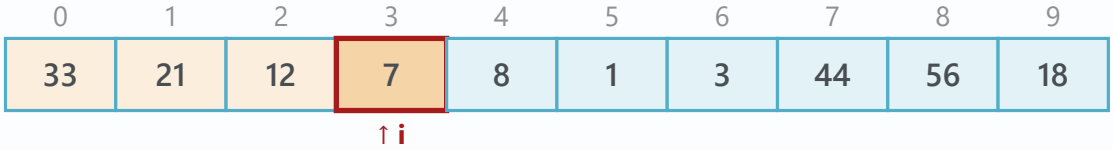
```
1  int dogrusalArama(int[] dizi, int n,
2                    int aranan) {
3    int i = 0;
4    while (i < n && dizi[i] != aranan)
▶      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

**DEĞİŞKENLER**

`n = 10` `aranan = 1` `i = 3`

**ÇIKTI**

# dogrusalArama(dizi, 10, 1): Kodu Adım Adım İzle




i < n ve dizi[3] = 7 ≠ 1 → döngü gövdesi çalışır.

Adım 8 / 13

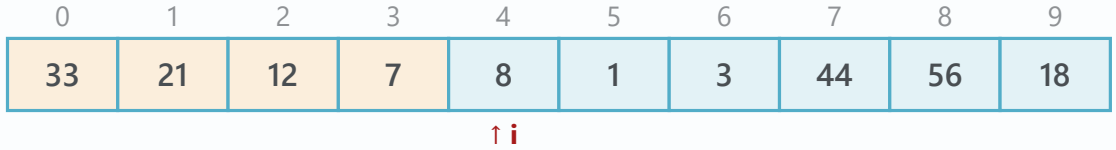
```
1  int dogrusalArama(int[] dizi, int n,
2                    int aranan) {
3    int i = 0;
▶    while (i < n && dizi[i] != aranan)
5      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

**DEĞİŞKENLER**

`n = 10` `aranan = 1` `i = 3`

**ÇIKTI**

# dogrusalArama(dizi, 10, 1): Kodu Adım Adım İzle




`i++` → i = 4.

Adım 9 / 13

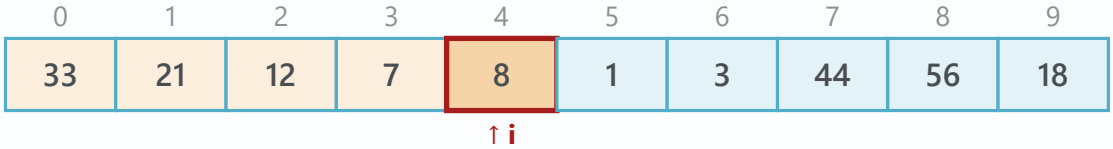
```
1 int dogrusalArama(int[] dizi, int n,
2                   int aranan) {
3   int i = 0;
4   while (i < n && dizi[i] != aranan)
▶     i++;
6   if (i < n)
7     return i;
8   else
9     return -1;
10 }
```

**DEĞİŞKENLER**

`n = 10` `aranan = 1` `i = 4`

**ÇIKTI**

# dogrusalArama(dizi, 10, 1): Kodu Adım Adım İzle




i < n ve dizi[4] = 8 ≠ 1 → döngü gövdesi çalışır.

Adım 10 / 13

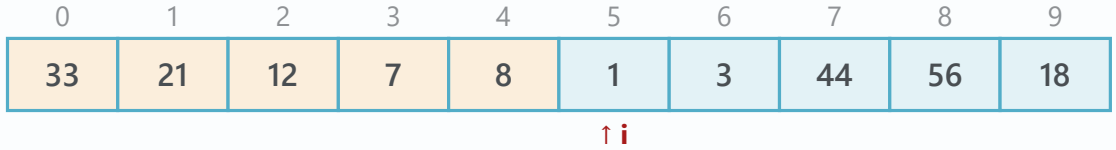
```
1  int dogrusalArama(int[] dizi, int n,
2                    int aranan) {
3    int i = 0;
▶    while (i < n && dizi[i] != aranan)
5      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

**DEĞİŞKENLER**

`n = 10` `aranan = 1` `i = 4`

**ÇIKTI**

# dogrusalArama(dizi, 10, 1): Kodu Adım Adım İzle




`i++` → i = 5.

Adım 11 / 13

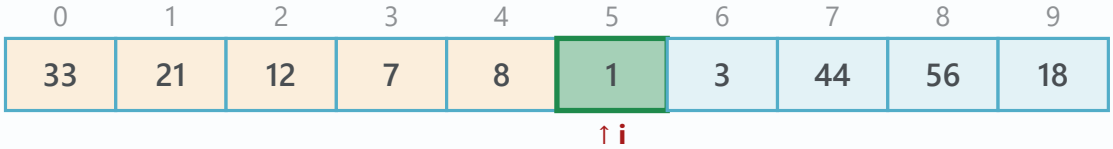
```
1  int dogrusalArama(int[] dizi, int n,
2                    int aranan) {
3    int i = 0;
4    while (i < n && dizi[i] != aranan)
▶      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

**DEĞİŞKENLER**

`n = 10` `aranan = 1` `i = 5`

**ÇIKTI**

# dogrusalArama(dizi, 10, 1): Kodu Adım Adım İzle




dizi[5] = 1 = aranan → `while` koşulu yanlış, döngüden çıkılır.

Adım 12 / 13

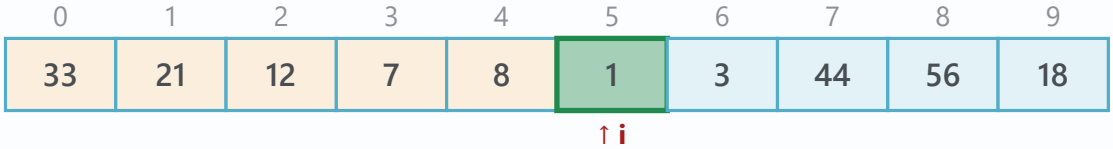
```
1  int dogrusalArama(int[] dizi, int n,
2                    int aranan) {
3    int i = 0;
▶    while (i < n && dizi[i] != aranan)
5      i++;
6    if (i < n)
7      return i;
8    else
9      return -1;
10 }
```

**DEĞİŞKENLER**

`n = 10` `aranan = 1` `i = 5`

**ÇIKTI**

# dogrusalArama(dizi, 10, 1): Kodu Adım Adım İzle




i = 5 < n → **5 döner**. 6 karşılaştırma yapıldı.

Adım 13 / 13

```
1  int dogrusalArama(int[] dizi, int n,
2                    int aranan) {
3    int i = 0;
4    while (i < n && dizi[i] != aranan)
5      i++;
▶    if (i < n)
▶      return i;
8    else
9      return -1;
10 }
```

**DEĞİŞKENLER**

`n = 10` `aranan = 1` `i = 5`

**ÇIKTI**

```
dogrusalArama(dizi, 10, 1) → 5
```

# Doğrusal Arama: Karmaşıklık ve Değerlendirme

| Durum | Karşılaştırma | Mertebe | Örnek (n = 10) |
|---|---|---|---|
| En iyi (ilk öğe) | 1 | O(1) | 33 → 1 karşılaştırma |
| Ortalama (başarılı, eşit olasılık) | (1 + 2 + … + n) / n = (n + 1)/2 | O(n) | ortalama 5,5 |
| En kötü / başarısız | n | O(n) | 17 → 10 karşılaştırma |

### Avantajlar

+ Kodlaması ve anlaşılması kolay; karmaşık veri yapısı gerektirmez.
+ **Ön hazırlık yoktur**: dizinin sıralı olması gerekmez.
+ Küçük veri kümelerinde ideal (ör. rehber listesinde bir numara).
+ Bağlı liste gibi yalnızca sıralı erişilen yapılarda da çalışır.

### Dezavantajlar

– Büyük veride yavaş: $10^6$ öğede ortalama 500 000 karşılaştırma.
– Dizi sıralı olsa bile bu bilgiden **yararlanmaz**.
– Çok sayıda sorgu varsa diziyi bir kez sıralayıp ikili arama yapmak daha ucuzdur.

# Bekçili (Sentinel) Doğrusal Arama

```
int gozcuIleAra(int[] dizi, int n,
                int aranan) {
  int sonEleman = dizi[n - 1];
  dizi[n - 1] = aranan;     // bekçi
  int i = 0;
  while (dizi[i] != aranan)
    i++;
  dizi[n - 1] = sonEleman;  // geri koy
  if (i < n - 1 || sonEleman == aranan)
    return i;
  return -1;
}
```

- Klasik döngü her turda **iki** karşılaştırma yapar: `i < n` ve `dizi[i] != aranan`.
- Aranan değer son hücreye **bekçi** (sentinel) olarak yazılırsa döngü mutlaka durur; `i < n` testine gerek kalmaz.
- Karşılaştırma sayısı ≈ 2n'den ≈ n + 2'ye iner; mertebe yine **O(n)**'dir.

Son öğe geçici olarak değiştirildiği için dizi **salt okunur** olmamalı ve aynı anda başka bir iş parçacığı tarafından okunmamalıdır.

# gozculleAra(dizi, 5, 5): Kodu Adım Adım İzle

```
1  int gozcuIleAra(int[] dizi, int n,
2                  int aranan) {
▶    int sonEleman = dizi[n - 1];
4    dizi[n - 1] = aranan;      // bekçi
5    int i = 0;
6    while (dizi[i] != aranan)
7      i++;
8    dizi[n - 1] = sonEleman;  // geri koy
9    if (i < n - 1 || sonEleman == aranan)
10     return i;
11   return -1;
12 }
```




sonEleman = 6 **aranan = 5**

**karşılaştırma: 0**

dizi = [7, 3, 9, 4, 6], aranan = 5 (dizide yok). Son öğe 6, `sonEleman`'da saklanır.

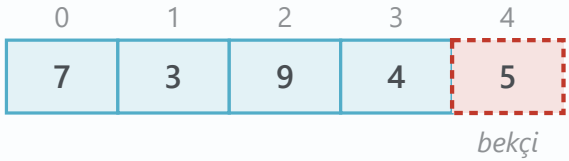
Adım 1 / 9

DEĞİŞKENLER

`n = 5` `aranan = 5` `sonEleman = 6` `i = –`

# gozculleAra(dizi, 5, 5): Kodu Adım Adım İzle

```
1  int gozcuIleAra(int[] dizi, int n,
2                  int aranan) {
3    int sonEleman = dizi[n - 1];
▶    dizi[n - 1] = aranan;      // bekçi
▶    int i = 0;
6    while (dizi[i] != aranan)
7      i++;
8    dizi[n - 1] = sonEleman;  // geri koy
9    if (i < n - 1 || sonEleman == aranan)
10     return i;
11   return -1;
12 }
```




sonEleman = 6 **aranan = 5**

**karşılaştırma: 0**

Son hücreye aranan 5 **bekçi** olarak yazılır: döngü artık en geç burada durur. `i = 0`.

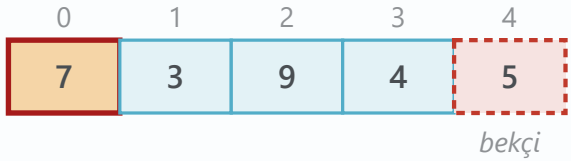
Adım 2 / 9

DEĞİŞKENLER

`n = 5` `aranan = 5` `sonEleman = 6` `i = 0`

# gozculleAra(dizi, 5, 5): Kodu Adım Adım İzle

```
1  int gozcuIleAra(int[] dizi, int n,
2                  int aranan) {
3    int sonEleman = dizi[n - 1];
4    dizi[n - 1] = aranan;      // bekçi
5    int i = 0;
▶    while (dizi[i] != aranan)
▶      i++;
8    dizi[n - 1] = sonEleman;  // geri koy
9    if (i < n - 1 || sonEleman == aranan)
10     return i;
11   return -1;
12 }
```




sonEleman = 6 **aranan = 5**

**karşılaştırma: 1**

dizi[0] = 7 ≠ 5 → `i++`. Yalnızca **bir** karşılaştırma; `i < n` testi yok.

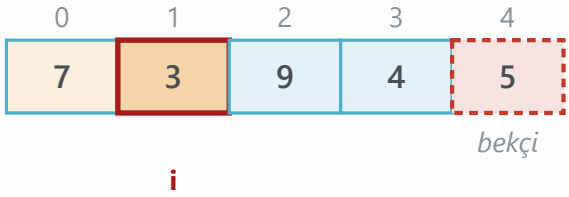
Adım 3 / 9

**DEĞİŞKENLER**

`n = 5` `aranan = 5` `sonEleman = 6` `i = 0`

# gozculleAra(dizi, 5, 5): Kodu Adım Adım İzle

```
1  int gozcuIleAra(int[] dizi, int n,
2                  int aranan) {
3    int sonEleman = dizi[n - 1];
4    dizi[n - 1] = aranan;      // bekçi
5    int i = 0;
▶    while (dizi[i] != aranan)
▶      i++;
8    dizi[n - 1] = sonEleman;  // geri koy
9    if (i < n - 1 || sonEleman == aranan)
10     return i;
11   return -1;
12 }
```




sonEleman = 6 **aranan = 5**

**karşılaştırma: 2**

dizi[1] = 3 ≠ 5 → `i++`. Yalnızca **bir** karşılaştırma; `i < n` testi yok.

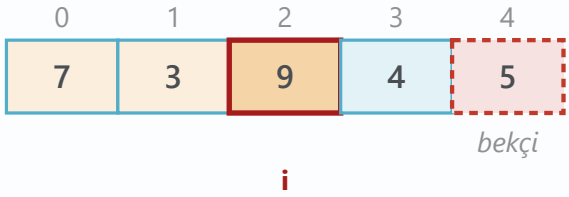
Adım 4 / 9

DEĞİŞKENLER

`n = 5` `aranan = 5` `sonEleman = 6` `i = 1`

# gozculleAra(dizi, 5, 5): Kodu Adım Adım İzle

```
1  int gozcuIleAra(int[] dizi, int n,
2                  int aranan) {
3    int sonEleman = dizi[n - 1];
4    dizi[n - 1] = aranan;      // bekçi
5    int i = 0;
▶    while (dizi[i] != aranan)
▶      i++;
8    dizi[n - 1] = sonEleman;  // geri koy
9    if (i < n - 1 || sonEleman == aranan)
10     return i;
11   return -1;
12 }
```




sonEleman = 6 **aranan = 5**

**karşılaştırma: 3**

dizi[2] = 9 ≠ 5 → `i++`. Yalnızca **bir** karşılaştırma; `i < n` testi yok.

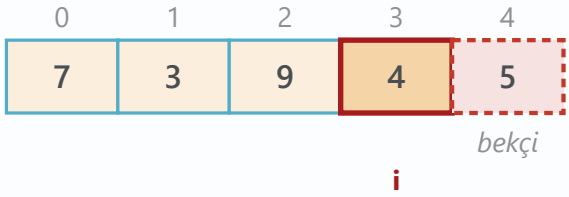
Adım 5 / 9

DEĞİŞKENLER

`n = 5` `aranan = 5` `sonEleman = 6` `i = 2`

# gozculleAra(dizi, 5, 5): Kodu Adım Adım İzle

```
1  int gozcuIleAra(int[] dizi, int n,
2                  int aranan) {
3    int sonEleman = dizi[n - 1];
4    dizi[n - 1] = aranan;      // bekçi
5    int i = 0;
▶    while (dizi[i] != aranan)
▶      i++;
8    dizi[n - 1] = sonEleman;  // geri koy
9    if (i < n - 1 || sonEleman == aranan)
10     return i;
11   return -1;
12 }
```




sonEleman = 6 **aranan = 5**

**karşılaştırma: 4**

dizi[3] = 4 ≠ 5 → `i++`. Yalnızca **bir** karşılaştırma; `i < n` testi yok.

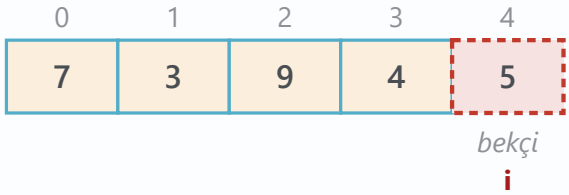
Adım 6 / 9

DEĞİŞKENLER

`n = 5` `aranan = 5` `sonEleman = 6` `i = 3`

# gozculleAra(dizi, 5, 5): Kodu Adım Adım İzle

```
1  int gozcuIleAra(int[] dizi, int n,
2                  int aranan) {
3    int sonEleman = dizi[n - 1];
4    dizi[n - 1] = aranan;      // bekçi
5    int i = 0;
▶    while (dizi[i] != aranan)
7      i++;
8    dizi[n - 1] = sonEleman;  // geri koy
9    if (i < n - 1 || sonEleman == aranan)
10     return i;
11   return -1;
12 }
```




sonEleman = 6 **aranan = 5**

**karşılaştırma: 5**

dizi[4] = 5 = aranan: döngü durur. Ama bulunan gerçek öğe değil, bekçidir.

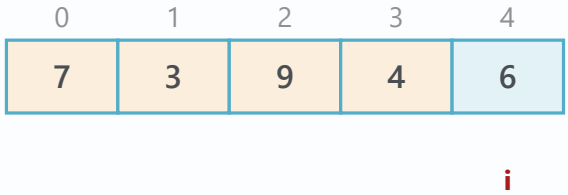
Adım 7 / 9

**DEĞİŞKENLER**

`n = 5` `aranan = 5` `sonEleman = 6` `i = 4`

# gozculleAra(dizi, 5, 5): Kodu Adım Adım İzle

```
1  int gozcuIleAra(int[] dizi, int n,
2                  int aranan) {
3    int sonEleman = dizi[n - 1];
4    dizi[n - 1] = aranan;     // bekçi
5    int i = 0;
6    while (dizi[i] != aranan)
7      i++;
▶    dizi[n - 1] = sonEleman;  // geri koy
9    if (i < n - 1 || sonEleman == aranan)
10     return i;
11   return -1;
12 }
```




sonEleman = 6 **aranan = 5**

**karşılaştırma: 5**

Son öğe geri konur: dizi yine [7, 3, 9, 4, 6].

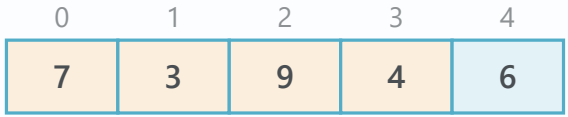
Adım 8 / 9

**DEĞİŞKENLER**

`n = 5` `aranan = 5` `sonEleman = 6` `i = 4`

# gozculleAra(dizi, 5, 5): Kodu Adım Adım İzle

```
1  int gozcuIleAra(int[] dizi, int n,
2                  int aranan) {
3    int sonEleman = dizi[n - 1];
4    dizi[n - 1] = aranan;      // bekçi
5    int i = 0;
6    while (dizi[i] != aranan)
7      i++;
8    dizi[n - 1] = sonEleman;  // geri koy
▶    if (i < n - 1 || sonEleman == aranan)
10     return i;
▶    return -1;
12 }
```




sonEleman = 6 **aranan = 5**

**karşılaştırma: 7**

i < n − 1? 4 < 4 yanlış; sonEleman == 5? 6 ≠ 5 → `return -1`. Toplam **7** karşılaştırma = n + 2; klasik döngü 2n = 10 yapardı.

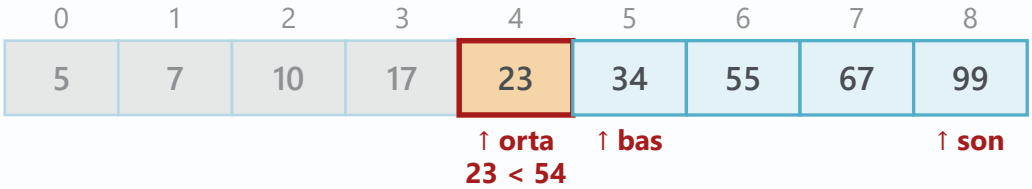
Adım 9 / 9

DEĞİŞKENLER

`n = 5` `aranan = 5` `sonEleman = 6` `i = 4` `sonuç = -1`

# İkili Arama (Binary Search)

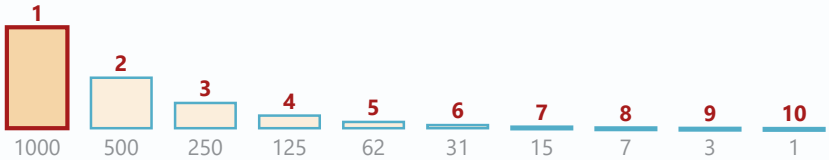
- **Kütüphanede kitap aramaya** benzer: kitaplar yazarın soyadına göre alfabetik sıralanmıştır.
- Tüm rafları tek tek aramak yerine önce **orta** bölüme gidilir.
  - Kitap ortanın **solundaysa** sol taraftaki raflarda,
  - **sağındaysa** sağ taraftaki raflarda aramaya devam edilir.
- Her karşılaştırma, aranması gereken kısmı **yarıya** indirir.
- Önkoşul: dizi **sıralı** olmalıdır ve öğelere indisle O(1)'de erişilebilmelidir.



# İkili Arama Neden Hızlı?

- Sıralı listelerde arama yaparken hızlıdır: her adımda kalan kısım **yarıya** iner.
- 1000 öğeli bir listede doğrusal arama **ortalama 500** kontrol yaparken ikili arama en fazla **10 adımda** aramayı tamamlar.
- 1000 → 500 → 250 → 125 → 62 → 31 → 15 → 7 → 3 → 1: her aralık boyu bir karşılaştırma, toplam 10.
- Büyük ve sıralı veri kümelerinde arama için idealdir.

| n | Doğrusal (en kötü) | İkili (en kötü) |
|---|---|---|
| 10 | 10 | 4 |
| 1 000 | 1 000 | 10 |
| $10^6$ | $10^6$ | 20 |
| $10^9$ | $10^9$ | 30 |




İkili arama en fazla $\lfloor \log_2 n \rfloor + 1$ karşılaştırma yapar. Veri 1000 kat büyüdüğünde yalnızca ≈ 10 adım eklenir.

# İkili Arama Algoritması

```
int ikiliArama(int[] dizi, int n, int aranan) {
  int bas = 0, son = n - 1;
  while (bas <= son) {
    int orta = (bas + son) / 2;
    if (aranan < dizi[orta])
      son = orta - 1;       // sol yarı
    else if (aranan > dizi[orta])
      bas = orta + 1;       // sağ yarı
    else
      return orta;          // bulundu
  }
  return -1;
}
```

- `bas` ve `son`, arama yapılacak aralığın alt ve üst sınırlarıdır.
- `while`, `bas ≤ son` olduğu (aralık boş olmadığı) sürece devam eder.
- `orta`, her adımda aralığın ortasındaki öğenin indisidir.
- Aranan ortadan **küçükse** alt yarıda, **büyükse** üst yarıda devam edilir; **eşitse** `orta` döndürülür.
- Aralık boşalırsa (`bas > son`) −1 döner.

# Özyinelemeli İkili Arama

```
int ara(int[] dizi, int bas,
        int son, int aranan) {
  if (bas > son) return -1;  // boş aralık
  int orta = (bas + son) / 2;
  if (dizi[orta] == aranan) return orta;
  if (dizi[orta] < aranan)           // sağ yarı
    return ara(dizi, orta + 1, son, aranan);
  return ara(dizi, bas, orta - 1, aranan);
}
```

- İlk çağrı `ara(dizi, 0, n - 1, aranan)` ; döngüdeki `bas` / `son` güncellemesinin yerini çağrı alır.
- Böl ve fethet bakışıyla: **böl** ortadaki öğe, **yönet** yalnızca **bir** yarıda devam, **birleştir** yok.
- Tek dallı olduğu için **azalt ve fethet** (decrease and conquer) de denir: T(n) = T(n/2) + 1.
- Ek bellek: O(log n) çağrı yığını (yinelemeli sürümde O(1)).

# İkili Arama = Dengeli İkili Arama Ağacı

5 11 12 14 15 18 19 21 23 25 27 28 30 32 37

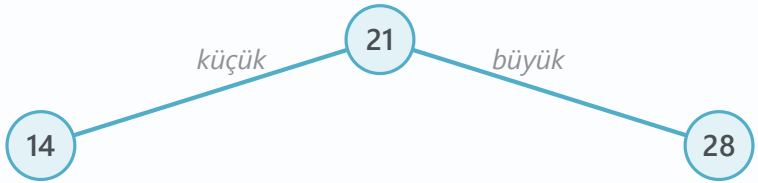
Sıralı dizi: 15 öğe. Bu diziden **en iyi (dengeli) ikili arama ağacını** kuralım.

# İkili Arama = Dengeli İkili Arama Ağacı




Ortadaki öğe **21** kök olur: ikili aramanın **ilk** karşılaştırması daima budur.

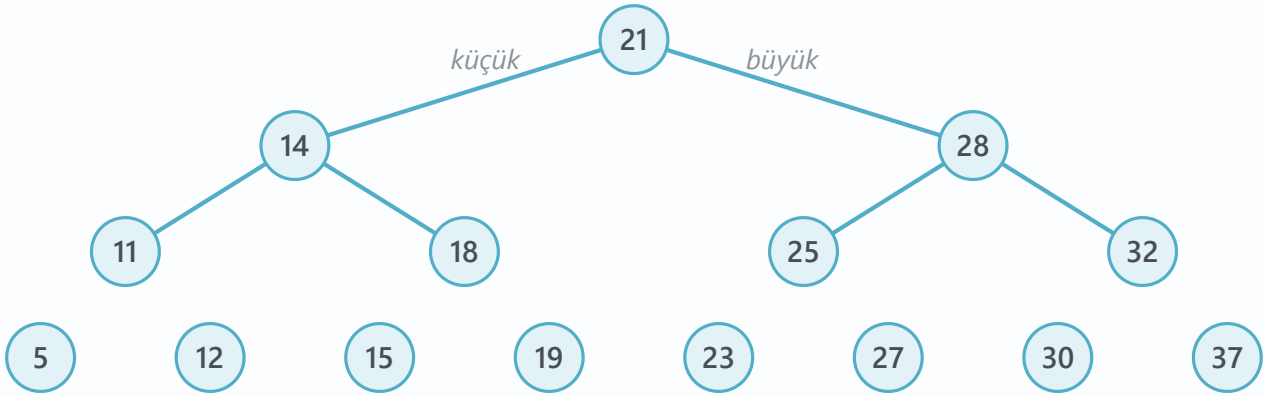
# İkili Arama = Dengeli İkili Arama Ağacı




5 11 12 15 18 19 23 25 27 30 32 37

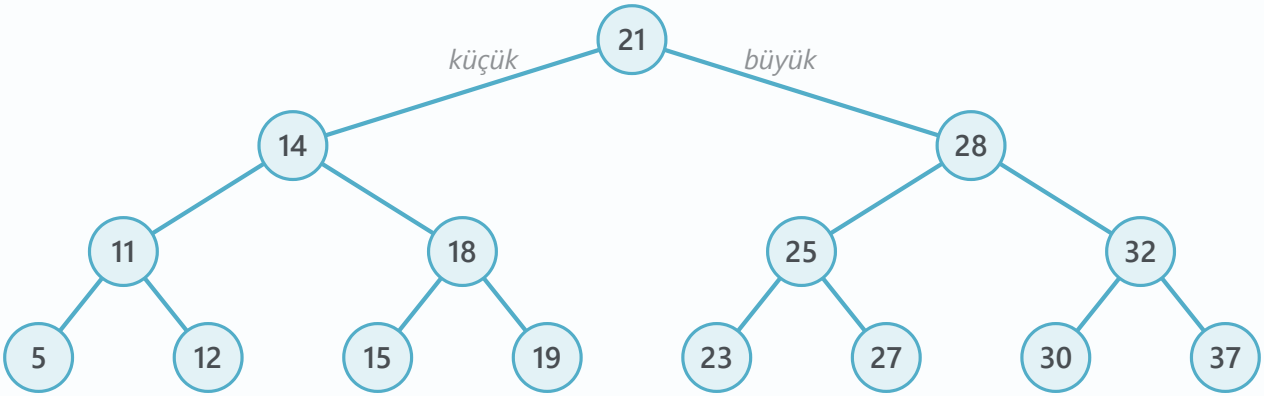
Sol yarının ortası 14, sağ yarının ortası 28: 21'den **küçükler** solda, **büyükler** sağda.

# İkili Arama = Dengeli İkili Arama Ağacı



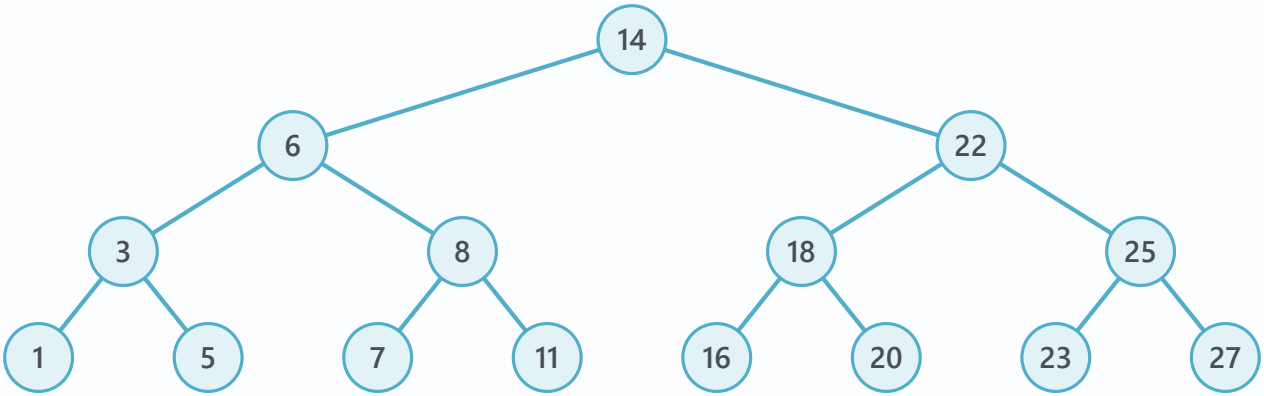

Her alt aralığın ortası bir sonraki düzeyi oluşturur: 11, 18, 25, 32.

# İkili Arama = Dengeli İkili Arama Ağacı



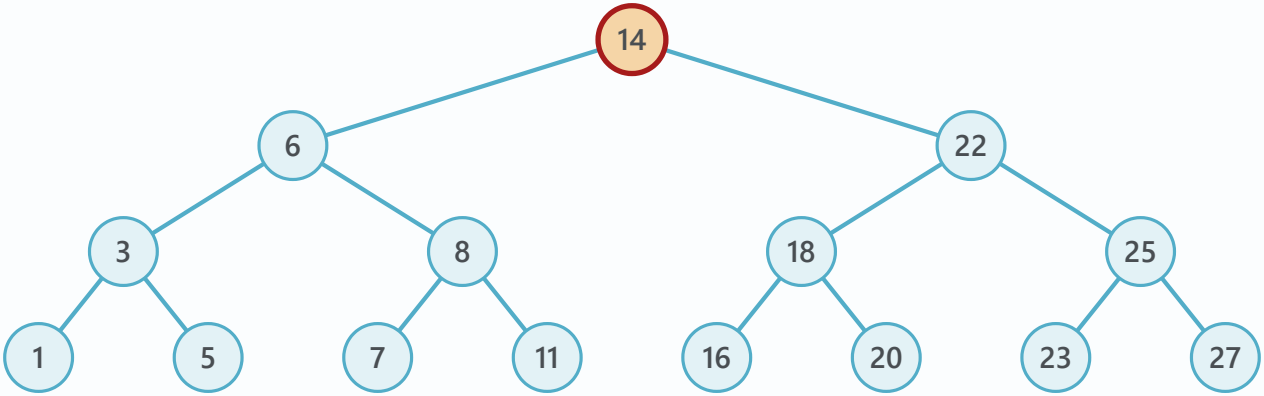

Kalan öğeler yapraklardır. Ağacın yüksekliği 3, düzey sayısı 4 = $\lfloor \log_2 15 \rfloor + 1$: her arama **en fazla 4** karşılaştırma yapar.

# Sıralı Dizi Örtük Bir Ağaçtır



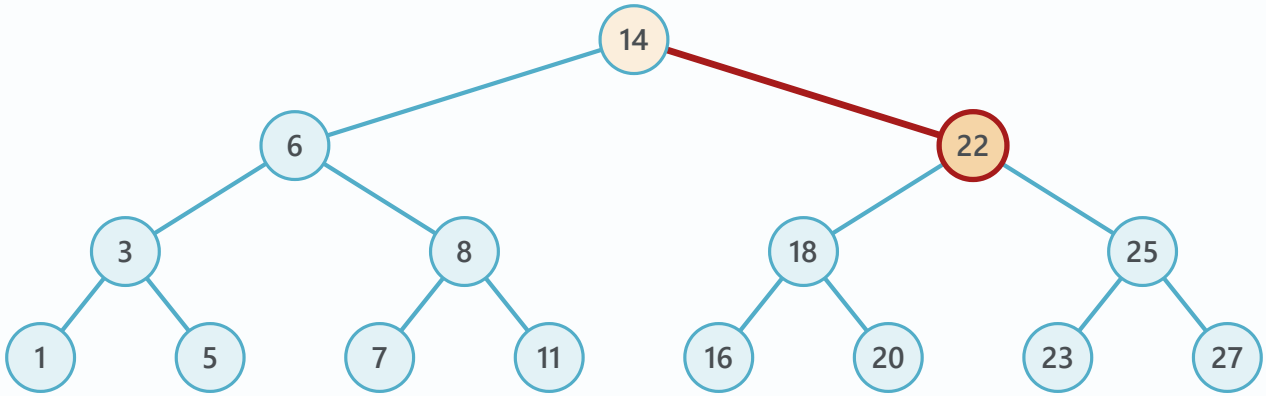

İkinci bir sıralı dizi (15 öğe: 1, 3, 5, 6, …, 25, 27) de aynı şekilde ağaç olarak görülebilir. Ağaç bellekte **kurulmaz**: `orta` hesabı onu örtük olarak gezer. 20'yi arayalım.

# Sıralı Dizi Örtük Bir Ağaçtır



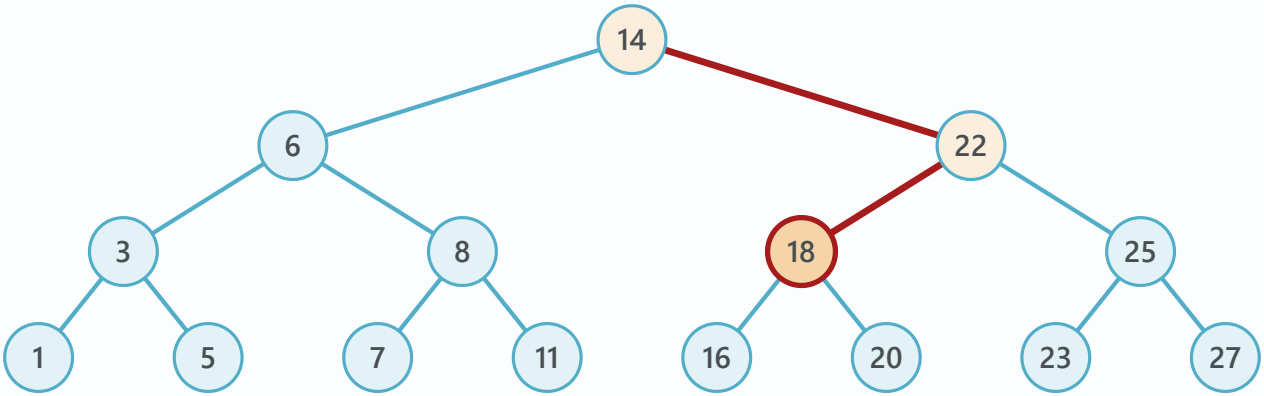

orta = 7 → dizi[7] = 14; 20 > 14 → sağa.

# Sıralı Dizi Örtük Bir Ağaçtır



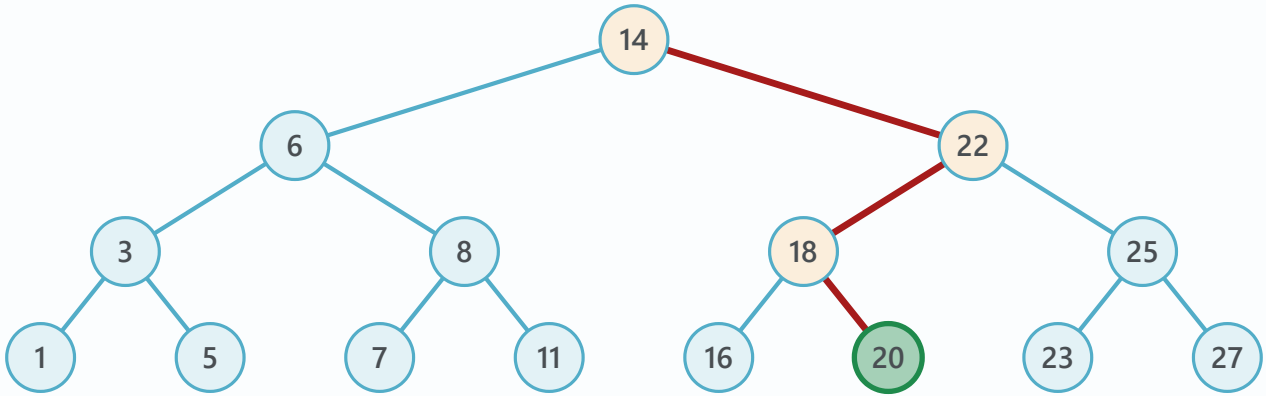

orta = 11 → dizi[11] = 22; 20 < 22 → sola.

# Sıralı Dizi Örtük Bir Ağaçtır



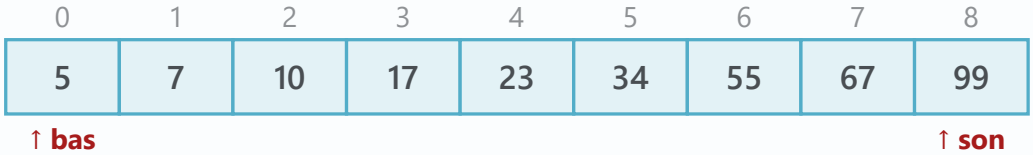

orta = 9 → dizi[9] = 18; 20 > 18 → sağa.

# Sıralı Dizi Örtük Bir Ağaçtır



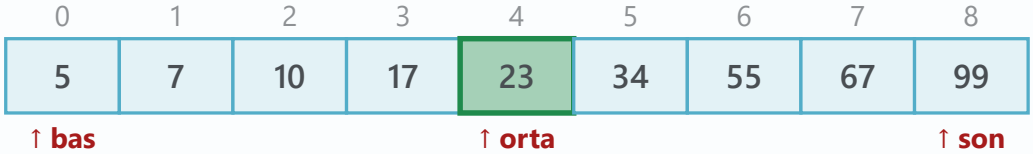

orta = 10 → dizi[10] = 20: **bulundu**. Kökten yaprağa 4 karşılaştırma; yol 14 → 22 → 18 → 20.

# İkili Arama: En İyi ve En Kötü Durum



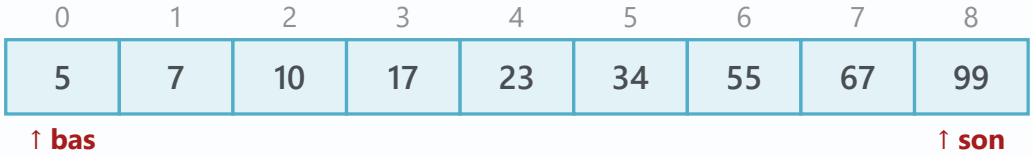

**En iyi durum: 23 aranır.** bas = 0, son = 8.

# İkili Arama: En İyi ve En Kötü Durum



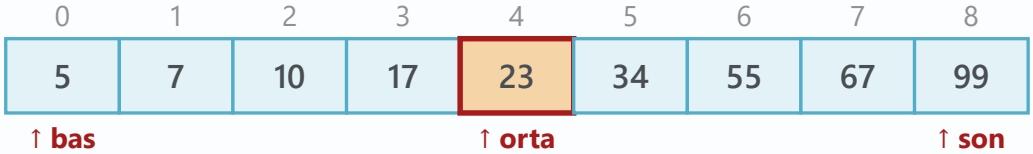

orta = (0 + 8) / 2 = 4 → dizi[4] = 23 = aranan: **1 adımda bulundu**, 4 döner.

# İkili Arama: En İyi ve En Kötü Durum




**En kötü durum: 54 aranır** (dizide yok).

# İkili Arama: En İyi ve En Kötü Durum



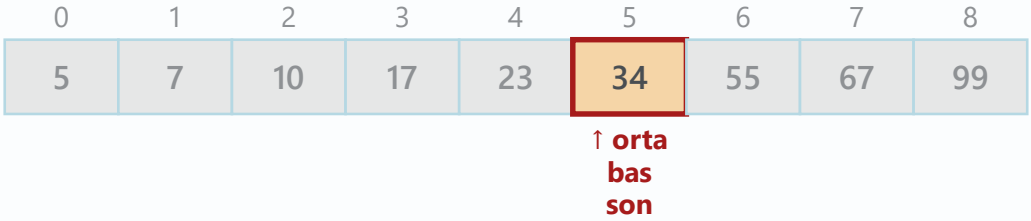

Adım 1: orta = 4 → 23 < 54 → `bas = 5` ; sol yarı elenir.

# İkili Arama: En İyi ve En Kötü Durum




Adım 2: orta = (5 + 8) / 2 = 6 → 55 > 54 → `son = 5`.

# İkili Arama: En İyi ve En Kötü Durum




Adım 3: orta = (5 + 5) / 2 = 5 → 34 < 54 → `bas = 6`.

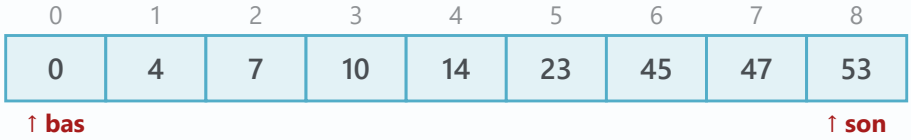
# İkili Arama: En İyi ve En Kötü Durum




bas = 6 > son = 5: aralık boş → **3 adımda bulunamadı**, −1 döner. 54 olsaydı 34 ile 55 arasında (indis 6'da) olurdu.

İKİLİ ARAMA · KODU ADIM ADIM ÇALIŞTIR

# ikiliArama(dizi, 9, 47): Kodu Adım Adım İzle




Aranan 47; `bas` = 0, `son` = 8.

Adım 1 / 7

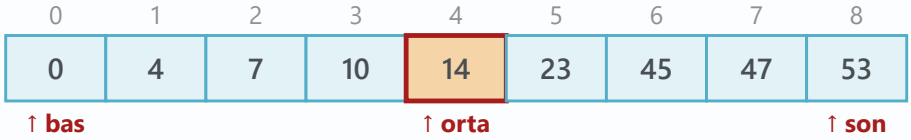
```
1  int ikiliArama(int[] dizi, int n, int aranan) {
▶    int bas = 0, son = n - 1;
3    while (bas <= son) {
4      int orta = (bas + son) / 2;
5      if (aranan < dizi[orta])
6        son = orta - 1;       // sol yarı
7      else if (aranan > dizi[orta])
8        bas = orta + 1;       // sağ yarı
9      else
10       return orta;          // bulundu
11   }
12   return -1;
13 }
```

**DEĞİŞKENLER**

`aranan = 47` `bas = 0` `son = 8` `orta = –`

**ÇIKTI**

# ikiliArama(dizi, 9, 47): Kodu Adım Adım İzle




bas ≤ son → orta = (0 + 8) / 2 = 4; dizi[4] = 14.

Adım 2 / 7

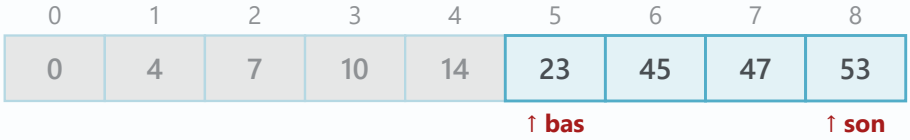
```
1  int ikiliArama(int[] dizi, int n, int aranan) {
2    int bas = 0, son = n - 1;
▶    while (bas <= son) {
▶      int orta = (bas + son) / 2;
5      if (aranan < dizi[orta])
6        son = orta - 1;       // sol yarı
7      else if (aranan > dizi[orta])
8        bas = orta + 1;       // sağ yarı
9      else
10       return orta;          // bulundu
11   }
12   return -1;
13 }
```

**DEĞİŞKENLER**

`aranan = 47` `bas = 0` `son = 8` `orta = 4`

**ÇIKTI**

# ikiliArama(dizi, 9, 47): Kodu Adım Adım İzle




47 > 14 → `bas = 5` ; sol kısım elenir.

Adım 3 / 7

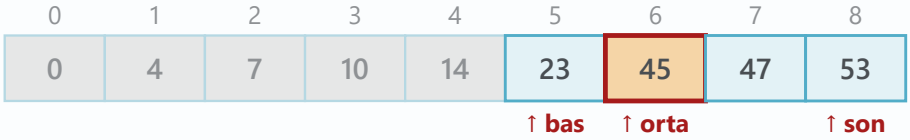
```
1  int ikiliArama(int[] dizi, int n, int aranan) {
2    int bas = 0, son = n - 1;
3    while (bas <= son) {
4      int orta = (bas + son) / 2;
5      if (aranan < dizi[orta])
6        son = orta - 1;       // sol yarı
▶      else if (aranan > dizi[orta])
▶        bas = orta + 1;       // sağ yarı
9      else
10       return orta;          // bulundu
11   }
12   return -1;
13 }
```

**DEĞİŞKENLER**

`aranan = 47` `bas = 5` `son = 8` `orta = 4`

**ÇIKTI**

# ikiliArama(dizi, 9, 47): Kodu Adım Adım İzle




bas ≤ son → orta = (5 + 8) / 2 = 6; dizi[6] = 45.

Adım 4 / 7

```
1  int ikiliArama(int[] dizi, int n, int aranan) {
2    int bas = 0, son = n - 1;
▶    while (bas <= son) {
▶      int orta = (bas + son) / 2;
5      if (aranan < dizi[orta])
6        son = orta - 1;       // sol yarı
7      else if (aranan > dizi[orta])
8        bas = orta + 1;       // sağ yarı
9      else
10       return orta;          // bulundu
11   }
12   return -1;
13 }
```

**DEĞİŞKENLER**

`aranan = 47` `bas = 5` `son = 8` `orta = 6`

**ÇIKTI**

# ikiliArama(dizi, 9, 47): Kodu Adım Adım İzle




47 > 45 → `bas = 7` ; sol kısım elenir.

Adım 5 / 7

```
1  int ikiliArama(int[] dizi, int n, int aranan) {
2    int bas = 0, son = n - 1;
3    while (bas <= son) {
4      int orta = (bas + son) / 2;
5      if (aranan < dizi[orta])
6        son = orta - 1;       // sol yarı
▶      else if (aranan > dizi[orta])
▶        bas = orta + 1;       // sağ yarı
9      else
10       return orta;          // bulundu
11   }
12   return -1;
13 }
```

**DEĞİŞKENLER**

`aranan = 47` `bas = 7` `son = 8` `orta = 6`

**ÇIKTI**

İKİLİ ARAMA · KODU ADIM ADIM ÇALIŞTIR

# ikiliArama(dizi, 9, 47): Kodu Adım Adım İzle




bas ≤ son → orta = (7 + 8) / 2 = 7; dizi[7] = 47.

Adım 6 / 7

```
1  int ikiliArama(int[] dizi, int n, int aranan) {
2    int bas = 0, son = n - 1;
▶    while (bas <= son) {
▶      int orta = (bas + son) / 2;
5      if (aranan < dizi[orta])
6        son = orta - 1;       // sol yarı
7      else if (aranan > dizi[orta])
8        bas = orta + 1;       // sağ yarı
9      else
10       return orta;          // bulundu
11   }
12   return -1;
13 }
```

DEĞİŞKENLER

aranan = 47 | bas = 7 | son = 8 | orta = 7

ÇIKTI

İKİLİ ARAMA · KODU ADIM ADIM ÇALIŞTIR

# ikiliArama(dizi, 9, 47): Kodu Adım Adım İzle




47 == dizi[7] → **7 döner**. 9 öğede 3 karşılaştırma (en fazla $\lfloor \log_2 9 \rfloor + 1 = 4$).

Adım 7 / 7

```
1  int ikiliArama(int[] dizi, int n, int aranan) {
2    int bas = 0, son = n - 1;
3    while (bas <= son) {
4      int orta = (bas + son) / 2;
5      if (aranan < dizi[orta])
6        son = orta - 1;       // sol yarı
7      else if (aranan > dizi[orta])
8        bas = orta + 1;       // sağ yarı
▶      else
▶        return orta;          // bulundu
11   }
12   return -1;
13 }
```

**DEĞİŞKENLER**

`aranan = 47` `bas = 7`
`son = 8` `orta = 7`

**ÇIKTI**

```
ikiliArama(dizi, 9, 47) → 7
```

# İkili Arama: Karmaşıklık

$T(n) = T(n/2) + 1 \Rightarrow T(n) = O(\log n)$

- Her turda aralık en az yarıya iner: n → n/2 → n/4 → … → 1.
- k tur sonra $n/2^k \leq 1$ olur: $k \approx \log_2 n$.
- Ek bellek: yinelemeli sürümde **O(1)**, özyinelemeli sürümde O(log n) yığın.

| Durum | Mertebe |
|---|---|
| En iyi (ilk orta eşleşir) | O(1) |
| Ortalama | O(log n) |
| En kötü / başarısız | O(log n) |
| Ön hazırlık (sıralama) | O(n log n), bir kez |

**Ne zaman değer?** Dizi bir kez sıralanıp çok kez aranıyorsa: k sorgu için doğrusal O(k · n), sırala + ikili O(n log n + k log n). k büyüdükçe fark açılır.

# İkili Arama: Uygulama Ayrıntıları

### Taşma (overflow)

`(bas + son) / 2`, bas + son > $2^{31} - 1$ olduğunda negatif olur. Güvenli yazım: `bas + (son - bas) / 2` ya da `(bas + son) >>> 1`.

### Sınırların güncellenmesi

`son = orta` ya da `bas = orta` yazmak aralığı küçültmeyebilir → **sonsuz döngü**. Orta öğe zaten denetlendiği için ±1 şarttır.

### Tekrarlı öğeler

Eşitlikte hemen dönmek **herhangi bir** eşleşmeyi verir. İlk konum için eşitlikte sonucu kaydedip `son = orta - 1` ile sola devam edilir.

### Java kütüphanesi

`Arrays.binarySearch(dizi, x)`: bulunamazsa −(ekleme noktası) − 1 döndürür; böylece x'in sırayı bozmadan nereye ekleneceği de öğrenilir.

# Tekrarlı Öğelerde İlk Konum: ilkKonum(dizi, 4)

```
1  int ilkKonum(int[] dizi, int aranan) {
▶    int bas = 0, son = dizi.length - 1, sonuc =
   -1;
3    while (bas <= son) {
4      int orta = bas + (son - bas) / 2;
5      if (dizi[orta] == aranan) {
6        sonuc = orta;       // kaydet,
7        son = orta - 1;     // solda ara
8      } else if (dizi[orta] < aranan)
9        bas = orta + 1;
10     else
11       son = orta - 1;
12   }
13   return sonuc;
14 }
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 2 | 4 | 4 | 4 | 4 | 7 | 9 | 12 |

**bas** **son**

**sonuc = -1**

dizi = [2, 4, 4, 4, 4, 7, 9, 12], aranan = 4: dört kez geçiyor. Hedef, **ilk** 4'ün indisi. sonuc = −1 ile başlanır.

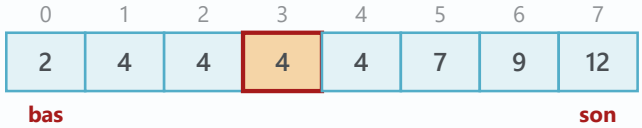
Adım 1 / 5

DEĞİŞKENLER

`bas = 0` `son = 7` `orta = –` `sonuc = -1`

# Tekrarlı Öğelerde İlk Konum: ilkKonum(dizi, 4)

```
1  int ilkKonum(int[] dizi, int aranan) {
2    int bas = 0, son = dizi.length - 1, sonuc =
   -1;
3    while (bas <= son) {
▶      int orta = bas + (son - bas) / 2;
▶      if (dizi[orta] == aranan) {
▶        sonuc = orta;       // kaydet,
▶        son = orta - 1;     // solda ara
8      } else if (dizi[orta] < aranan)
9        bas = orta + 1;
10     else
11       son = orta - 1;
12   }
13   return sonuc;
14 }
```




**sonuc = 3**

orta = 0 + (7 − 0) / 2 = 3: dizi[3] = 4 = aranan. Hemen dönmek yerine `sonuc = 3` kaydedilir ve `son = 2` ile **solda** daha önce bir 4 aranır.

Adım 2 / 5

DEĞİŞKENLER

`bas = 0` `son = 2` `orta = 3` `sonuc = 3`

# Tekrarlı Öğelerde İlk Konum: ilkKonum(dizi, 4)

```
1  int ilkKonum(int[] dizi, int aranan) {
2    int bas = 0, son = dizi.length - 1, sonuc =
   -1;
3    while (bas <= son) {
▶      int orta = bas + (son - bas) / 2;
▶      if (dizi[orta] == aranan) {
▶        sonuc = orta;       // kaydet,
▶        son = orta - 1;     // solda ara
8      } else if (dizi[orta] < aranan)
9        bas = orta + 1;
10     else
11       son = orta - 1;
12   }
13   return sonuc;
14 }
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 2 | 4 | 4 | 4 | 4 | 7 | 9 | 12 |

bas son

**sonuc = 1**

orta = 0 + (2 − 0) / 2 = 1: dizi[1] = 4 = aranan. Hemen dönmek yerine `sonuc = 1` kaydedilir ve `son = 0` ile **solda** daha önce bir 4 aranır.

Adım 3 / 5

DEĞİŞKENLER

`bas = 0` `son = 0` `orta = 1` `sonuc = 1`

# Tekrarlı Öğelerde İlk Konum: ilkKonum(dizi, 4)

```
1  int ilkKonum(int[] dizi, int aranan) {
2    int bas = 0, son = dizi.length - 1, sonuc =
   -1;
3    while (bas <= son) {
▶      int orta = bas + (son - bas) / 2;
▶      if (dizi[orta] == aranan) {
6        sonuc = orta;      // kaydet,
7        son = orta - 1;    // solda ara
▶      } else if (dizi[orta] < aranan)
▶        bas = orta + 1;
10     else
11       son = orta - 1;
12   }
13   return sonuc;
14 }
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 2 | 4 | 4 | 4 | 4 | 7 | 9 | 12 |

**bas/son**

**sonuc = 1**

orta = 0: dizi[0] = 2 < 4 → `bas = 1`.

Adım 4 / 5

DEĞİŞKENLER

`bas = 1` `son = 0` `orta = 0` `sonuc = 1`

# Tekrarlı Öğelerde İlk Konum: ilkKonum(dizi, 4)

```
1  int ilkKonum(int[] dizi, int aranan) {
2    int bas = 0, son = dizi.length - 1, sonuc =
   -1;
▶    while (bas <= son) {
4      int orta = bas + (son - bas) / 2;
5      if (dizi[orta] == aranan) {
6        sonuc = orta;       // kaydet,
7        son = orta - 1;     // solda ara
8      } else if (dizi[orta] < aranan)
9        bas = orta + 1;
10     else
11       son = orta - 1;
12   }
▶    return sonuc;
14 }
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 2 | 4 | 4 | 4 | 4 | 7 | 9 | 12 |
| | son | bas | | | | | |

**sonuc = 1**

bas = 1 > son = 0: aralık boş, döngü biter. `return 1` : ilk 4 indis 1'de. Yine O(log n): 3 yoklama.

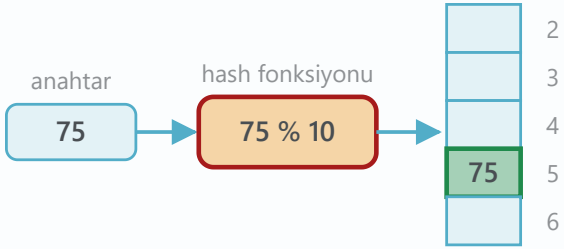
Adım 5 / 5

DEĞİŞKENLER

`bas = 1` `son = 0` `orta = –` `sonuc = 1`

# Hash Tablo Arama Algoritması

- Bir **anahtarın kilide** tam oturmasına benzer mantıkla çalışır.
- Veriler, **anahtarlar** ve onlara karşılık gelen **değerlerden** oluşur.
- Anahtar, **hash fonksiyonu** ile bir tablo indisine (hash değeri) dönüştürülür; bu değer verinin tablodaki yerini işaret eder.
- Arama, anahtarın hash değerine karşılık gelen konumda (**kova**, bucket) yapılır.
- Kova boş değilse anahtar, kovadaki değerlerle karşılaştırılır; eşleşme bulunursa işlem tamamlanır.




Diğer aramalar öğeleri **karşılaştırarak** yer arar; hash araması yeri **hesaplar**. Bu yüzden ortalama süre n'den bağımsızdır: O(1).

# Hash ile Arama: Avantajlar ve Dezavantajlar

## Avantajlar

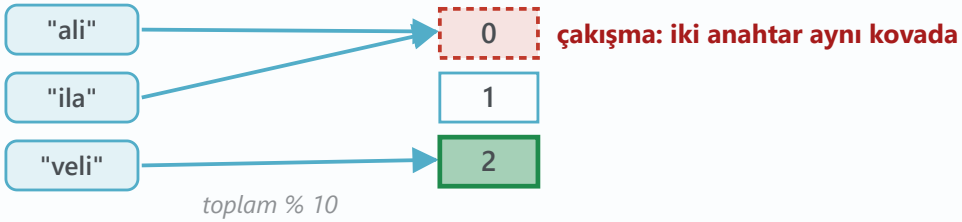
+ Ortalama durumda çok hızlı arama: **O(1)**.
+ Verilerin önceden **sıralanmasına gerek yoktur**.
+ Büyük veri kümelerinde aramalar için idealdir.
+ Hash fonksiyonu iyi seçilmişse aranan öğeye doğrudan erişilir.

## Dezavantajlar

– Hash fonksiyonu **çakışmalara** (collision) yol açabilir: iki farklı anahtarın aynı hash değerine sahip olması.

– Çakışma olduğunda ek işlem adımları gerekir; en kötü durumda **O(n)**.

– Ek bellek (boş kovalar) ister; sıralı gezinme, en küçük / aralık sorguları desteklenmez.



# Hash Fonksiyonu

```
int hashFonksiyonu(String anahtar,
                   int m) {
  int toplam = 0;
  for (char k : anahtar.toCharArray())
    toplam += k;          // Unicode kodu
  return toplam % m;      // m: tablo boyu
}
```

| anahtar | karakter kodları | toplam % 10 |
|---|---|---|
| "ali" | 97 + 108 + 105 = 310 | 0 |
| "ila" | 105 + 108 + 97 = 310 | 0 |
| "veli" | 118 + 101 + 108 + 105 = 432 | 2 |

- Bu fonksiyon, karakterlerin Unicode kodlarını (Java'da `char`'ın sayısal değeri) toplar ve tablo boyuna göre **mod** alır.
- Basit ama zayıftır: harfleri aynı olan kelimeler (**"ali"** ve **"ila"**) her zaman **çakışır**.
- İyi bir hash fonksiyonu hızlı hesaplanır ve anahtarları kovalara **düzgün** dağıtır; Java'nın `String.hashCode()`'u konuma bağlı ağırlık kullanır: $\Sigma\, s[i] \cdot 31^{n-1-i}$.

# hashFonksiyonu("veli", 10): Kodu Adım Adım İzle




anahtar = "veli", m = 10 kova. `toplam = 0`.

Adım 1 / 6

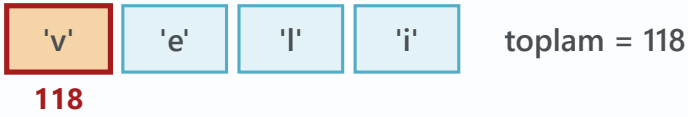
```
1 int hashFonksiyonu(String anahtar,
2                    int m) {
▶   int toplam = 0;
4   for (char k : anahtar.toCharArray())
5     toplam += k;          // Unicode kodu
6   return toplam % m;     // m: tablo boyu
7 }
```

**DEĞİŞKENLER**

`anahtar = "veli"` `m = 10`

`k = –` `toplam = 0`

# hashFonksiyonu("veli", 10): Kodu Adım Adım İzle




k = 'v': Unicode kodu **118** eklenir → toplam = 118.

Adım 2 / 6

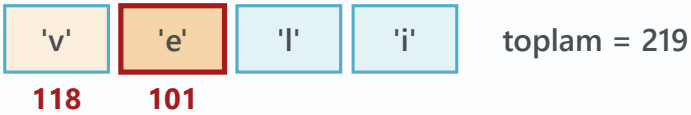
```
1 int hashFonksiyonu(String anahtar,
2                    int m) {
3   int toplam = 0;
▶   for (char k : anahtar.toCharArray())
▶     toplam += k;          // Unicode kodu
6   return toplam % m;     // m: tablo boyu
7 }
```

**DEĞİŞKENLER**

`anahtar = "veli"` `m = 10`

`k = 'v'` `toplam = 118`

# hashFonksiyonu("veli", 10): Kodu Adım Adım İzle




k = 'e': Unicode kodu **101** eklenir → toplam = 219.

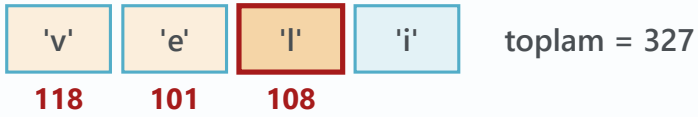
Adım 3 / 6

```
1 int hashFonksiyonu(String anahtar,
2                    int m) {
3   int toplam = 0;
▶   for (char k : anahtar.toCharArray())
▶     toplam += k;          // Unicode kodu
6   return toplam % m;     // m: tablo boyu
7 }
```

**DEĞİŞKENLER**

`anahtar = "veli"` `m = 10` `k = 'e'` `toplam = 219`

# hashFonksiyonu("veli", 10): Kodu Adım Adım İzle




k = 'l': Unicode kodu **108** eklenir → toplam = 327.

Adım 4 / 6

```
1 int hashFonksiyonu(String anahtar,
2                    int m) {
3   int toplam = 0;
▶   for (char k : anahtar.toCharArray())
▶     toplam += k;          // Unicode kodu
6   return toplam % m;     // m: tablo boyu
7 }
```

**DEĞİŞKENLER**

`anahtar = "veli"` `m = 10` `k = 'l'` `toplam = 327`

# hashFonksiyonu("veli", 10): Kodu Adım Adım İzle




k = 'i': Unicode kodu **105** eklenir → toplam = 432.

Adım 5 / 6

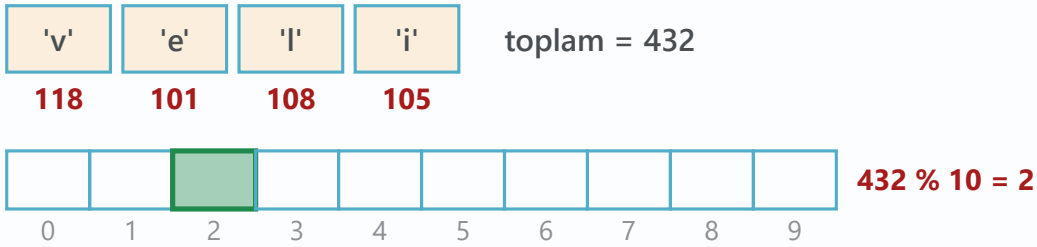
```
1 int hashFonksiyonu(String anahtar,
2                    int m) {
3   int toplam = 0;
▶   for (char k : anahtar.toCharArray())
▶     toplam += k;          // Unicode kodu
6   return toplam % m;     // m: tablo boyu
7 }
```

**DEĞİŞKENLER**

`anahtar = "veli"` `m = 10`

`k = 'i'` `toplam = 432`

# hashFonksiyonu("veli", 10): Kodu Adım Adım İzle




432 % 10 = **2**: "veli" 2 numaralı kovaya düşer. "ali" ve "ila" ise aynı harflerden oluştuğu için ikisi de 310 → 0 numaralı kovada çakışır.

Adım 6 / 6

```
1 int hashFonksiyonu(String anahtar,
2                    int m) {
3   int toplam = 0;
4   for (char k : anahtar.toCharArray())
5     toplam += k;          // Unicode kodu
▶   return toplam % m;     // m: tablo boyu
7 }
```

**DEĞİŞKENLER**

anahtar = **"veli"** | m = **10** | k = **–** | toplam = **432** | sonuç = **2**

# Hash Tablo Arama Algoritması (Doğrusal Yoklama)

```
int hashArama(Integer[] tablo, int anahtar) {
  int m = tablo.length;
  int indis = anahtar % m;    // hash
  while (tablo[indis] != null) {
    if (tablo[indis] == anahtar)
      return indis;           // bulundu
    indis = (indis + 1) % m; // sonraki kova
  }
  return -1;                  // boş kova: yok
}
```

- **Veri yapıları:** `tablo` anahtarları saklar (genel uygulamada anahtar-değer çiftleri); hash fonksiyonu anahtarı indise dönüştürür.
- `while`, `tablo[indis]` boş (null) olana kadar sürer.
- Her turda kovadaki anahtar aranan ile karşılaştırılır; eşleşirse konum (anahtar-değer tablosunda değer) döner.
- Eşleşme yoksa **bir sonraki** kovaya bakılır; `% m` ile sondan başa dönülür (dairesel arama).
- Boş kovaya varılırsa anahtar yoktur.

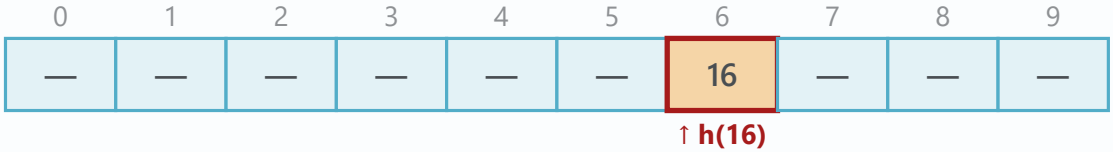
Bu çakışma çözümüne **açık adresleme** (doğrusal yoklama, linear probing) denir. Örneklerde anahtarlar tam sayıdır ve h(x) = x % 10 kullanılır.

# Doğrusal Yoklama ile Ekleme



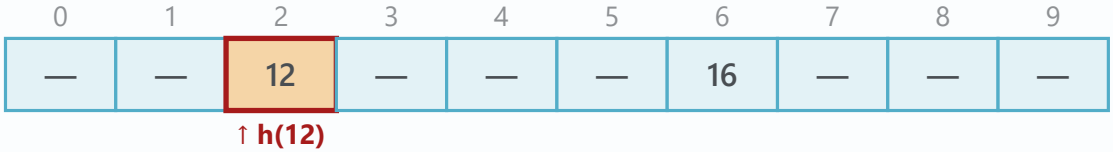

Boş tablo (m = 10). Anahtarlar sırayla eklenecek: 16, 12, 45, 29, 6, 122, 5, 58, 75; h(x) = x % 10.

# Doğrusal Yoklama ile Ekleme



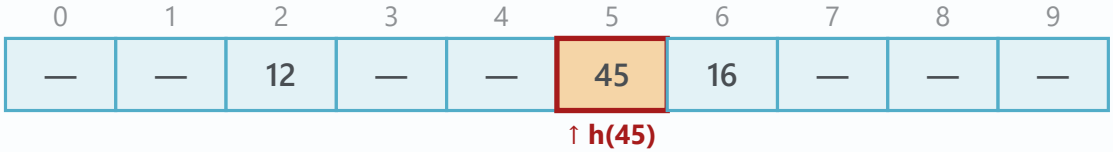

h(16) = 6: kova boş → 16 buraya yerleşir.

# Doğrusal Yoklama ile Ekleme



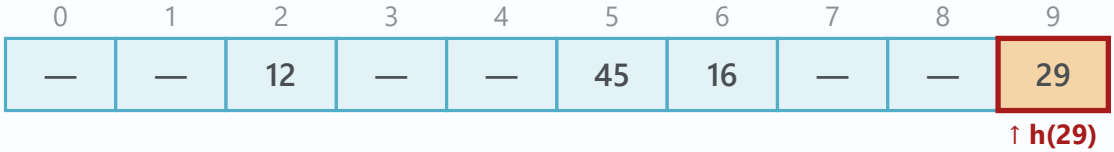

h(12) = 2: kova boş → 12 buraya yerleşir.

# Doğrusal Yoklama ile Ekleme



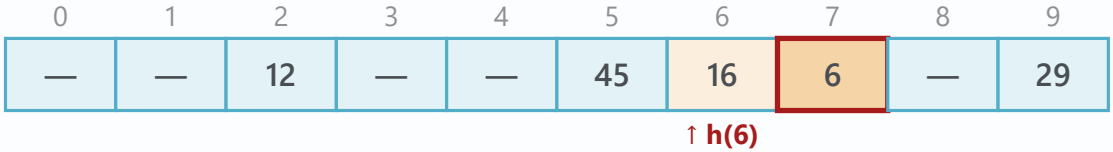

h(45) = 5: kova boş → 45 buraya yerleşir.

# Doğrusal Yoklama ile Ekleme



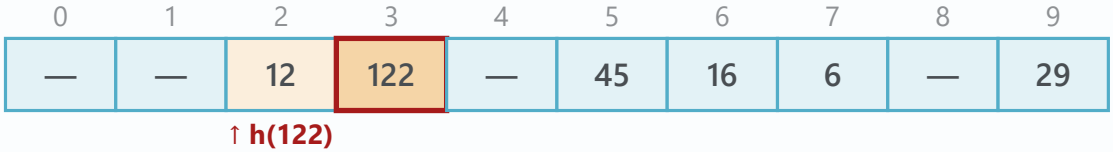

h(29) = 9: kova boş → 29 buraya yerleşir.

# Doğrusal Yoklama ile Ekleme



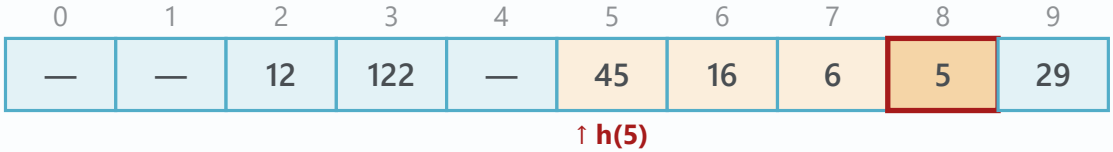

h(6) = 6 dolu → 7 yoklanır; ilk boş kova **7**. (2 yoklama)

# Doğrusal Yoklama ile Ekleme



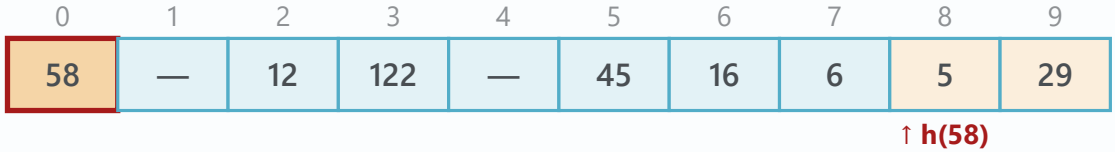

h(122) = 2 dolu → 3 yoklanır; ilk boş kova **3**. (2 yoklama)

# Doğrusal Yoklama ile Ekleme



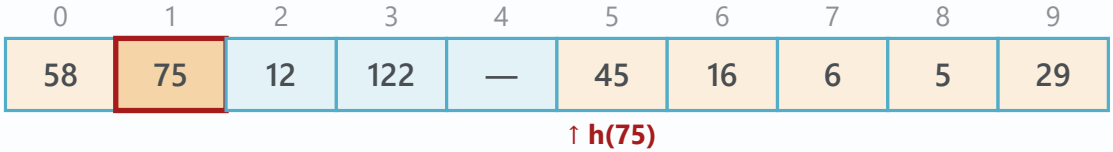

h(5) = 5 dolu → 6, 7, 8 yoklanır; ilk boş kova **8**. (4 yoklama)

# Doğrusal Yoklama ile Ekleme




h(58) = 8 dolu → 9, 0 yoklanır; ilk boş kova **0**. (3 yoklama)

# Doğrusal Yoklama ile Ekleme




h(75) = 5 dolu → 6, 7, 8, 9, 0, 1 yoklanır; ilk boş kova **1**. (7 yoklama) Dolu kovalar bitişik bir **küme** (5, 6, …, 9, 0, …, 3; yalnızca kova 4 boş) oluşturdu: birincil kümelenme.

# hashArama(tablo, 75): Kodu Adım Adım İzle

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 58 | 75 | 12 | 122 | — | 45 | 16 | 6 | 5 | 29 |

↑ **indis**

m = 10, `indis` = 75 % 10 = 5.

Adım 1 / 8

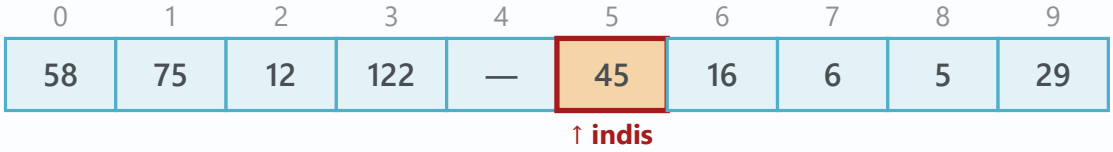
```
1  int hashArama(Integer[] tablo, int anahtar) {
▶    int m = tablo.length;
▶    int indis = anahtar % m;    // hash
4    while (tablo[indis] != null) {
5      if (tablo[indis] == anahtar)
6        return indis;           // bulundu
7      indis = (indis + 1) % m; // sonraki kova
8    }
9    return -1;                  // boş kova: yok
10 }
```

**DEĞİŞKENLER**

`anahtar` = **75** `m` = **10** `indis` = **5**

**ÇIKTI**

# hashArama(tablo, 75): Kodu Adım Adım İzle




tablo[5] = 45 ≠ 75 → indis = (5 + 1) % 10 = 6.

Adım 2 / 8

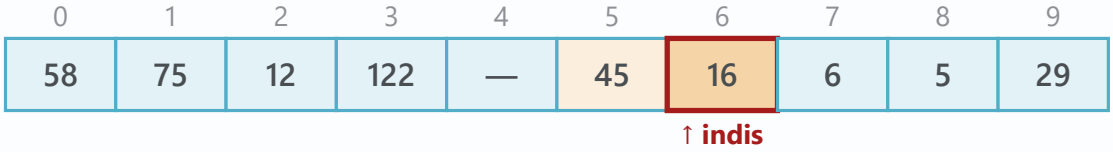
```
1  int hashArama(Integer[] tablo, int anahtar) {
2    int m = tablo.length;
3    int indis = anahtar % m;    // hash
▶    while (tablo[indis] != null) {
▶      if (tablo[indis] == anahtar)
6        return indis;           // bulundu
▶      indis = (indis + 1) % m; // sonraki kova
8    }
9    return -1;                  // boş kova: yok
10 }
```

**DEĞİŞKENLER**

`anahtar = 75` `m = 10` `indis = 5`

**ÇIKTI**

# hashArama(tablo, 75): Kodu Adım Adım İzle




tablo[6] = 16 ≠ 75 → indis = (6 + 1) % 10 = 7.

Adım 3 / 8

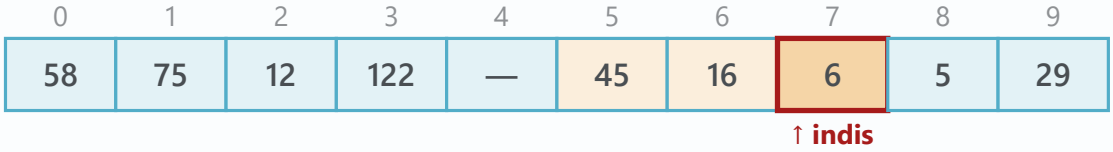
```
1  int hashArama(Integer[] tablo, int anahtar) {
2    int m = tablo.length;
3    int indis = anahtar % m;    // hash
▶    while (tablo[indis] != null) {
▶      if (tablo[indis] == anahtar)
6        return indis;           // bulundu
▶      indis = (indis + 1) % m; // sonraki kova
8    }
9    return -1;                  // boş kova: yok
10 }
```

**DEĞİŞKENLER**

`anahtar = 75` `m = 10` `indis = 6`

**ÇIKTI**

# hashArama(tablo, 75): Kodu Adım Adım İzle




tablo[7] = 6 ≠ 75 → indis = (7 + 1) % 10 = 8.

Adım 4 / 8

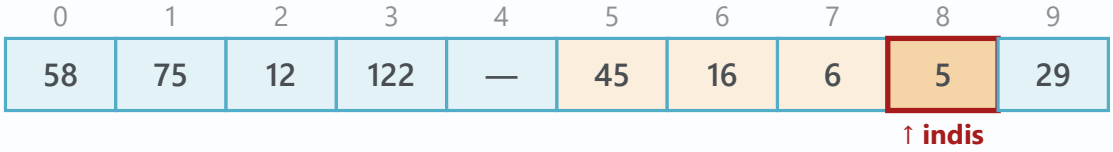
```
1  int hashArama(Integer[] tablo, int anahtar) {
2    int m = tablo.length;
3    int indis = anahtar % m;    // hash
▶    while (tablo[indis] != null) {
▶      if (tablo[indis] == anahtar)
6        return indis;           // bulundu
▶      indis = (indis + 1) % m; // sonraki kova
8    }
9    return -1;                  // boş kova: yok
10 }
```

**DEĞİŞKENLER**

`anahtar = 75` `m = 10` `indis = 7`

**ÇIKTI**

# hashArama(tablo, 75): Kodu Adım Adım İzle




tablo[8] = 5 ≠ 75 → indis = (8 + 1) % 10 = 9.

Adım 5 / 8

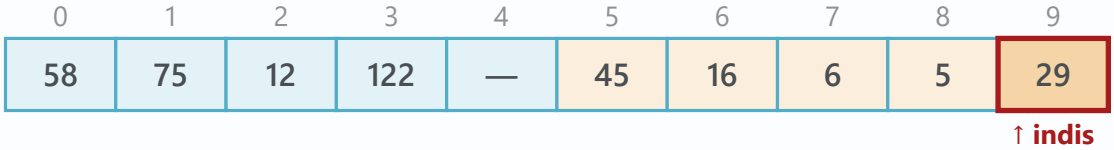
```
1  int hashArama(Integer[] tablo, int anahtar) {
2    int m = tablo.length;
3    int indis = anahtar % m;    // hash
▶    while (tablo[indis] != null) {
▶      if (tablo[indis] == anahtar)
6        return indis;           // bulundu
▶      indis = (indis + 1) % m; // sonraki kova
8    }
9    return -1;                  // boş kova: yok
10 }
```

**DEĞİŞKENLER**

`anahtar = 75` `m = 10` `indis = 8`

**ÇIKTI**

# hashArama(tablo, 75): Kodu Adım Adım İzle




tablo[9] = 29 ≠ 75 → indis = (9 + 1) % 10 = 0. Sondan başa dönüldü.

Adım 6 / 8

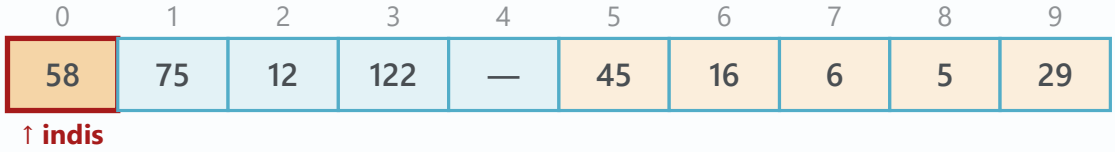
```
1  int hashArama(Integer[] tablo, int anahtar) {
2    int m = tablo.length;
3    int indis = anahtar % m;    // hash
▶    while (tablo[indis] != null) {
▶      if (tablo[indis] == anahtar)
6        return indis;           // bulundu
▶      indis = (indis + 1) % m; // sonraki kova
8    }
9    return -1;                  // boş kova: yok
10 }
```

**DEĞİŞKENLER**

`anahtar = 75` `m = 10` `indis = 9`

**ÇIKTI**

# hashArama(tablo, 75): Kodu Adım Adım İzle




tablo[0] = 58 ≠ 75 → indis = (0 + 1) % 10 = 1.

Adım 7 / 8

```
1  int hashArama(Integer[] tablo, int anahtar) {
2    int m = tablo.length;
3    int indis = anahtar % m;    // hash
▶    while (tablo[indis] != null) {
▶      if (tablo[indis] == anahtar)
6        return indis;           // bulundu
▶      indis = (indis + 1) % m; // sonraki kova
8    }
9    return -1;                  // boş kova: yok
10 }
```

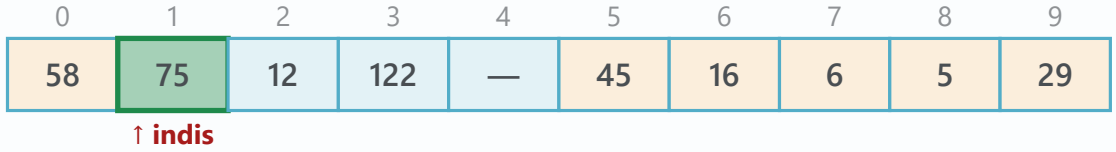
**DEĞİŞKENLER**

anahtar = **75** m = **10**

indis = **0**

**ÇIKTI**

# hashArama(tablo, 75): Kodu Adım Adım İzle




tablo[1] = 75 → **1 döner**. 7 yoklama: 75'in yerine kümenin sonunda ulaşıldı.

Adım 8 / 8

```
1  int hashArama(Integer[] tablo, int anahtar) {
2    int m = tablo.length;
3    int indis = anahtar % m;    // hash
▶    while (tablo[indis] != null) {
▶      if (tablo[indis] == anahtar)
▶        return indis;           // bulundu
7      indis = (indis + 1) % m; // sonraki kova
8    }
9    return -1;                  // boş kova: yok
10 }
```

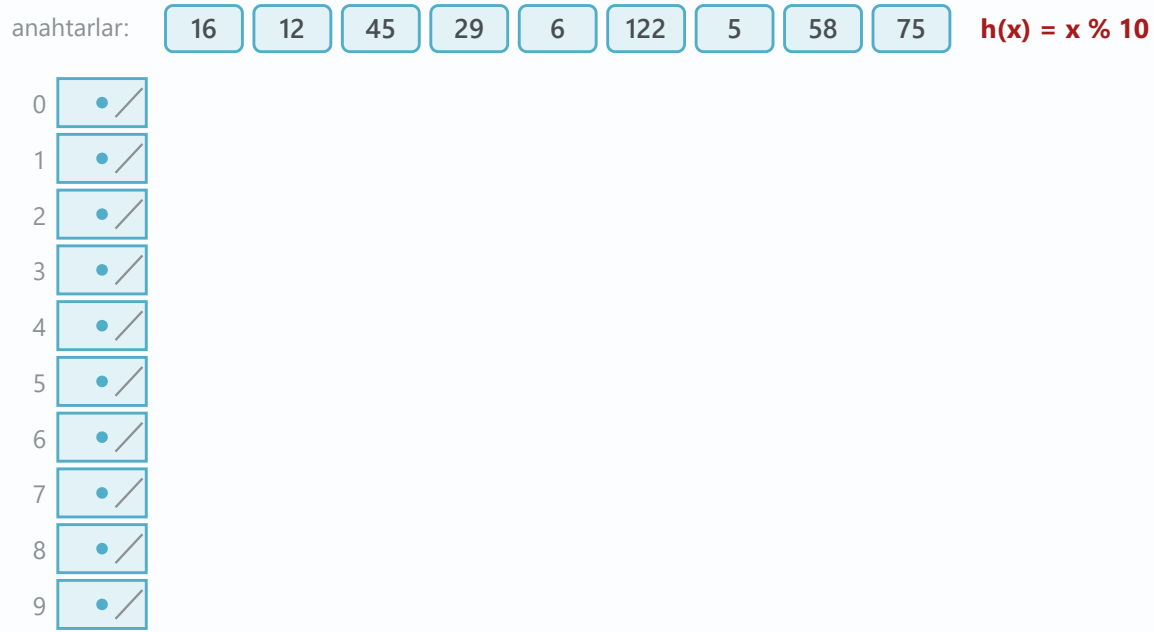
**DEĞİŞKENLER**

`anahtar = 75` `m = 10` `indis = 1`

**ÇIKTI**

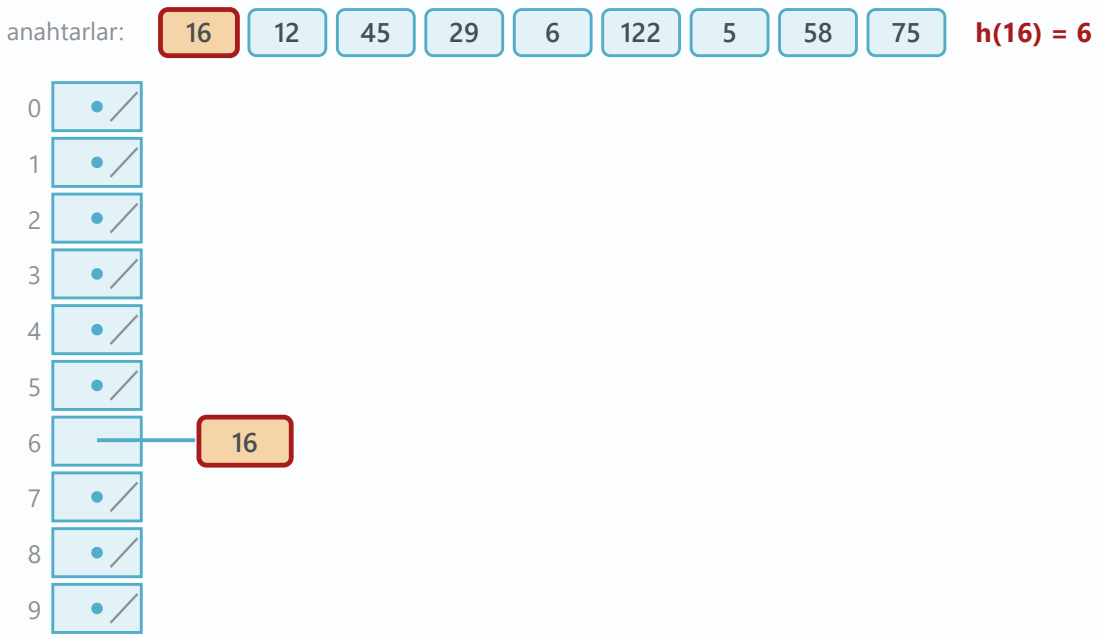
```
hashArama(tablo, 75) → 1
```

# Zincirleme (Chaining) ile Ekleme ve Arama



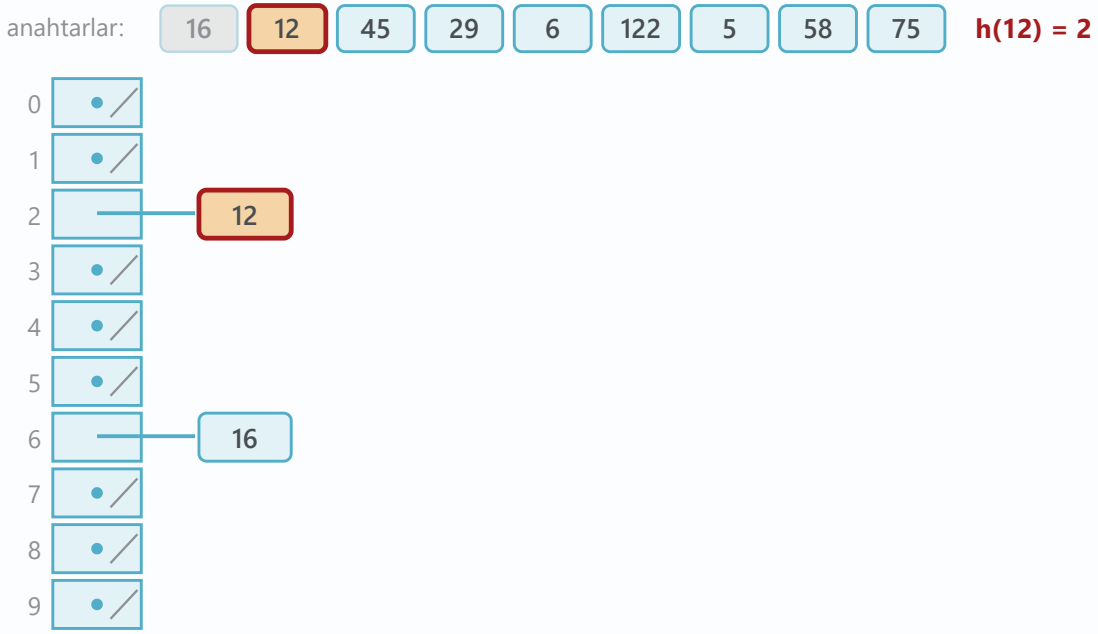

**Ayrı zincirleme** (separate chaining): her kova bir bağlı listenin başını tutar. Aynı anahtarlar, h(x) = x % 10.

# Zincirleme (Chaining) ile Ekleme ve Arama



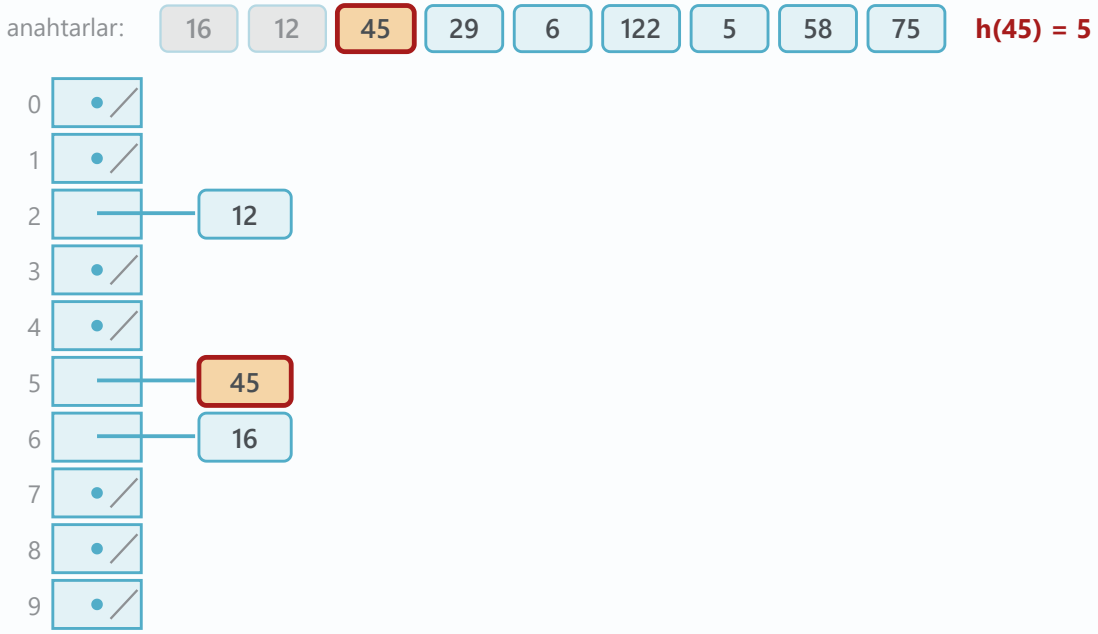

h(16) = 6 → kova 6 boş; 16 listenin **sonuna** eklenir.

# Zincirleme (Chaining) ile Ekleme ve Arama



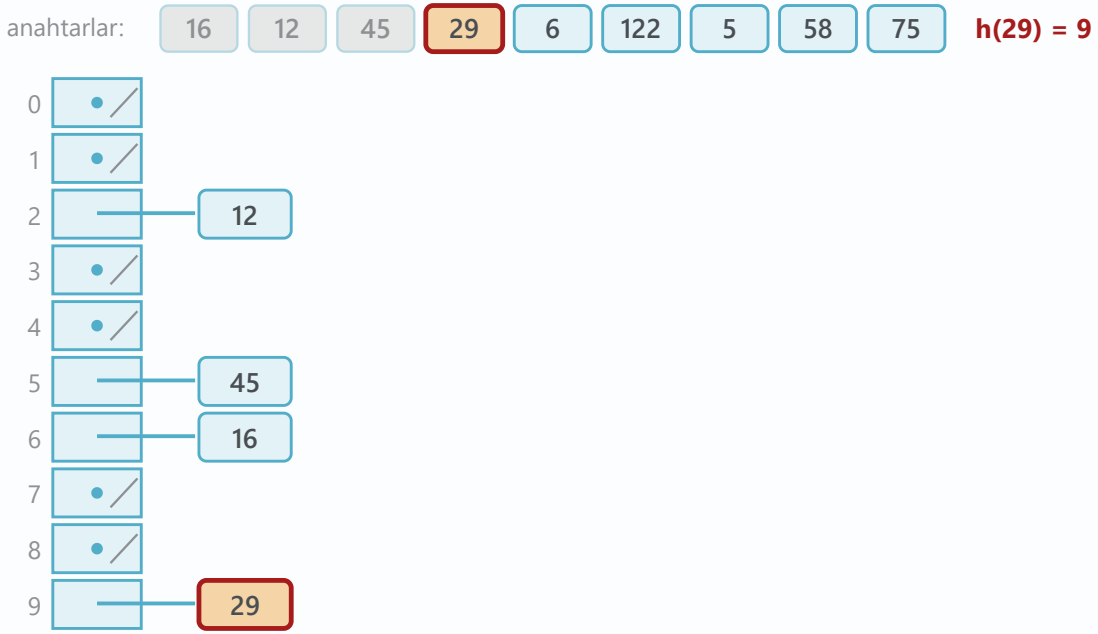

h(12) = 2 → kova 2 boş; 12 listenin **sonuna** eklenir.

# Zincirleme (Chaining) ile Ekleme ve Arama



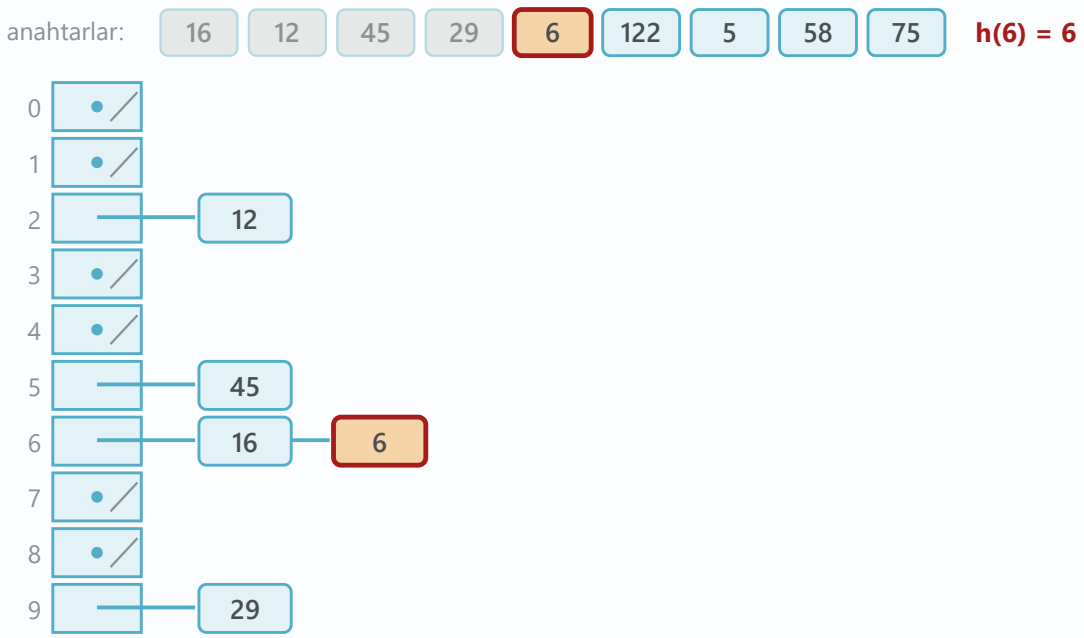

h(45) = 5 → kova 5 boş; 45 listenin **sonuna** eklenir.

# Zincirleme (Chaining) ile Ekleme ve Arama



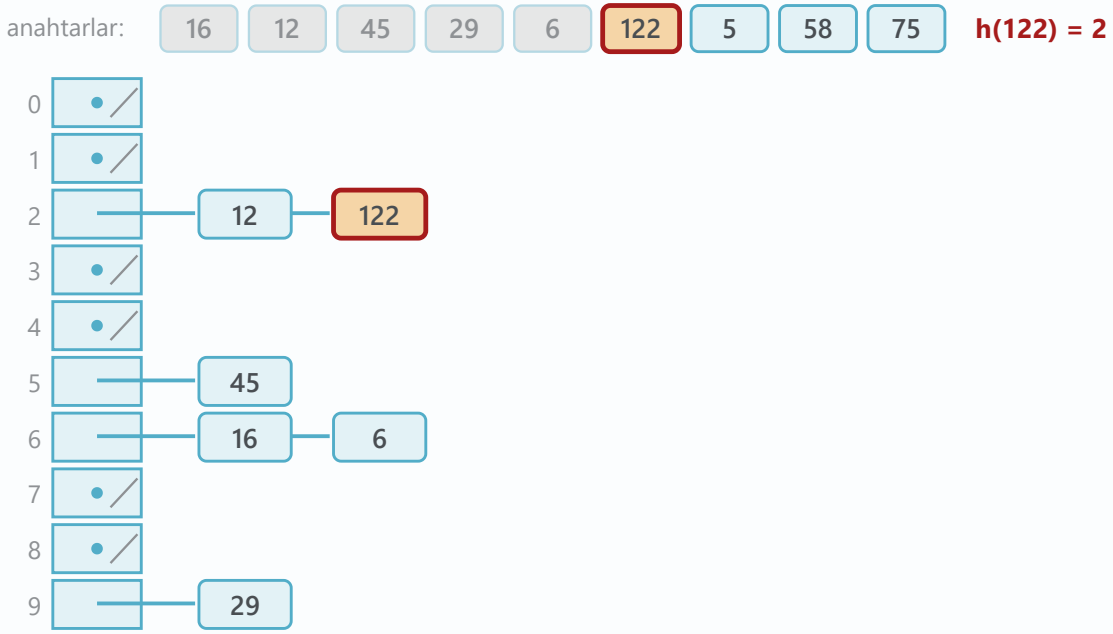

h(29) = 9 → kova 9 boş; 29 listenin **sonuna** eklenir.

# Zincirleme (Chaining) ile Ekleme ve Arama



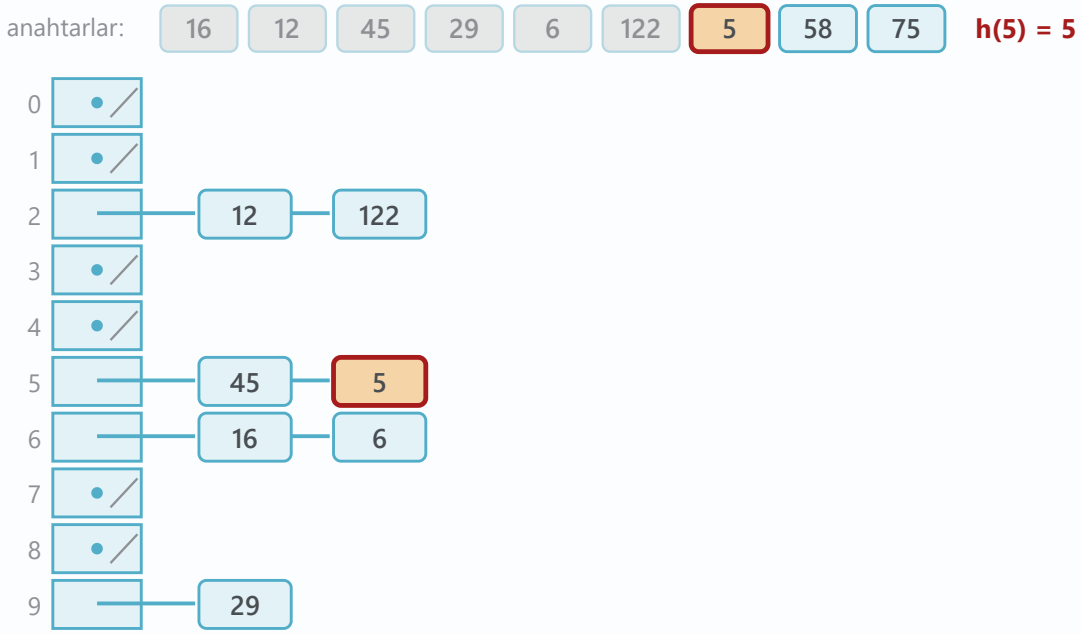

h(6) = 6 → kova 6 dolu (çakışma); 6 listenin **sonuna** eklenir.

# Zincirleme (Chaining) ile Ekleme ve Arama



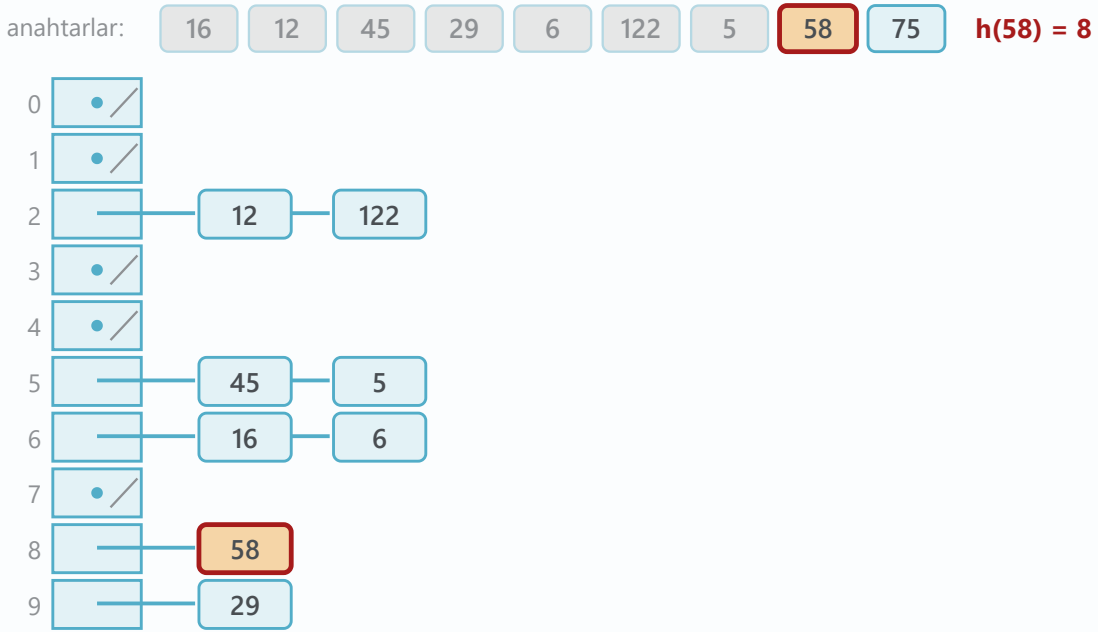

h(122) = 2 → kova 2 dolu (çakışma); 122 listenin **sonuna** eklenir.

# Zincirleme (Chaining) ile Ekleme ve Arama



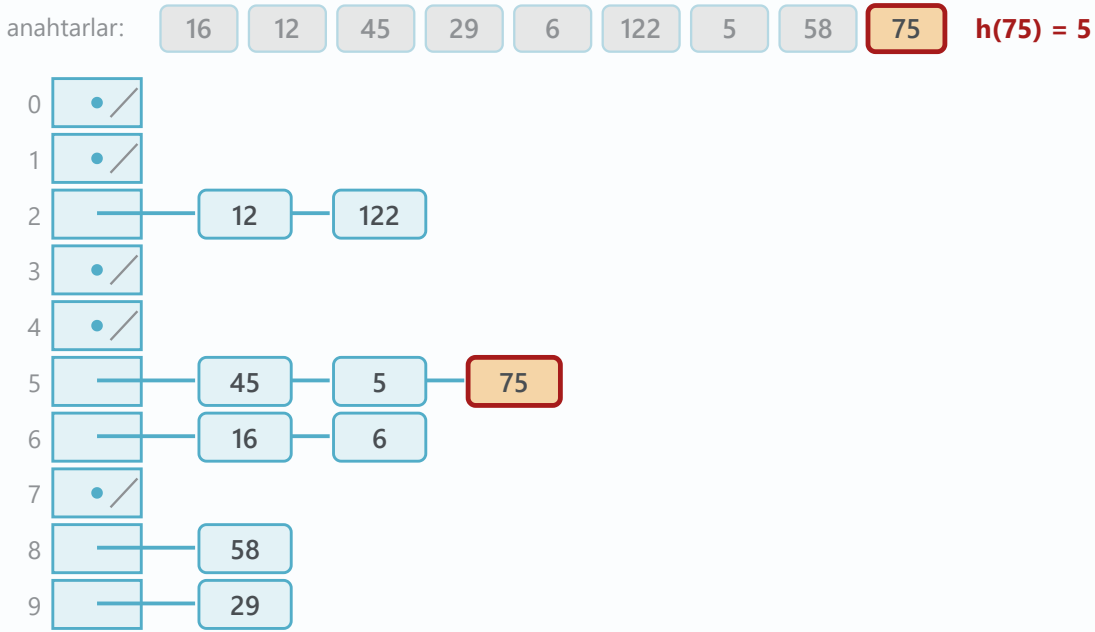

h(5) = 5 → kova 5 dolu (çakışma); 5 listenin **sonuna** eklenir.

# Zincirleme (Chaining) ile Ekleme ve Arama



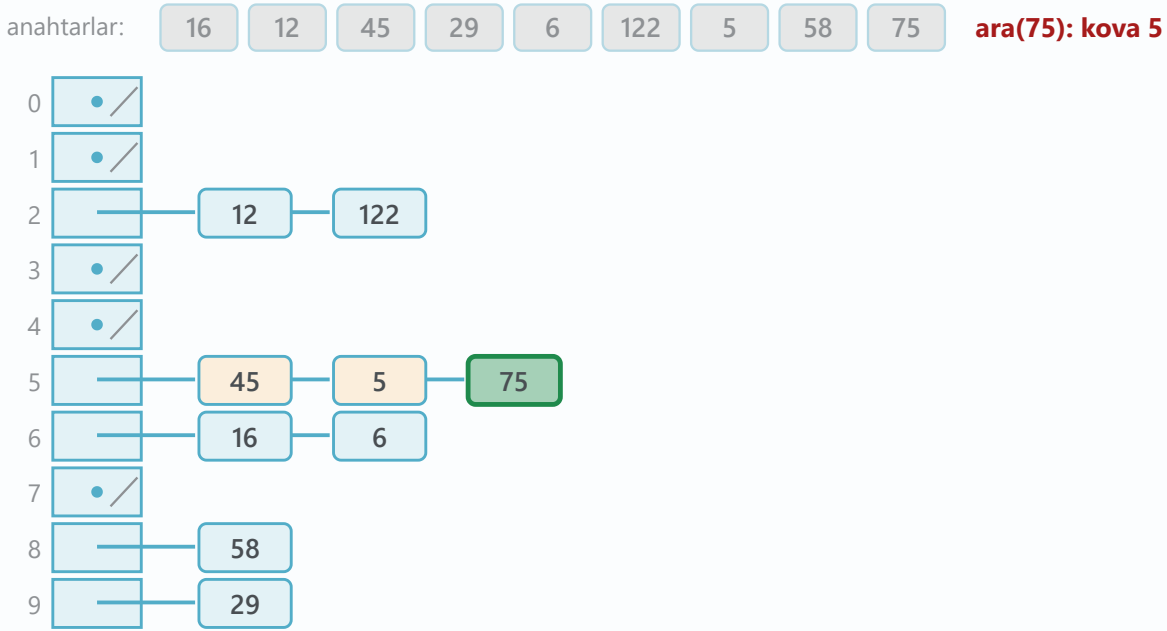

h(58) = 8 → kova 8 boş; 58 listenin **sonuna** eklenir.

# Zincirleme (Chaining) ile Ekleme ve Arama




h(75) = 5 → kova 5 dolu (çakışma); 75 listenin **sonuna** eklenir.

# Zincirleme (Chaining) ile Ekleme ve Arama




**Arama** 75: kova 5'in listesi gezilir: 45 ≠ 75, 5 ≠ 75, 75 = 75 → bulundu (3 karşılaştırma). Doğrusal yoklamadaki 7 yoklamaya göre daha az.

# Yük Faktörü ve Karmaşıklık

`yük faktörü α = n / m  (n: anahtar sayısı, m: kova sayısı)`

| Yöntem | Başarılı arama (ort.) | Başarısız arama (ort.) | α = 0,5 | α = 0,9 |
|---|---|---|---|---|
| Zincirleme | ≈ 1 + α/2 | ≈ 1 + α | 1,25 / 1,5 | 1,45 / 1,9 |
| Doğrusal yoklama | ≈ ½ (1 + 1/(1 − α)) | ≈ ½ (1 + 1/(1 − α)$^2$) | 1,5 / 2,5 | 5,5 / 50,5 |

- α sabit tutulursa (ör. α ≤ 0,75 olunca tabloyu iki katına çıkarıp yeniden yerleştirmek) ortalama arama **O(1)** olur.
- En kötü durum **O(n)**: tüm anahtarlar aynı kovaya düşerse (kötü hash fonksiyonu) arama doğrusal aramaya dönüşür.
- Doğrusal yoklamada α → 1 iken kümelenme maliyeti hızla büyütür; tablo tamamen dolarsa başarısız aramada döngü **hiç bitmez**.



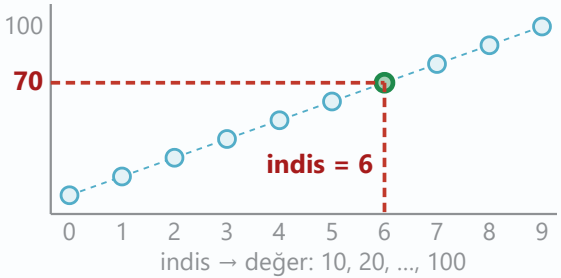

Formüller Knuth'un düzgün hash varsayımı altındaki yaklaşık değerleridir.

# Ara Değer (Interpolation) Arama

- **Sıralı** veri kümelerinde kullanılır.
- Verilerin listede **eşit aralıklarla** dağıldığını varsayar: başlangıç değerinden sonra ardışık öğeler arasındaki fark yaklaşık aynıdır.
- Bu varsayımla aranan değerin konumu, **en küçük ve en büyük** öğe kullanılarak **tahmin** edilir.
- Tahmin edilen konum kontrol edilir; yanlışsa arama daha dar bir aralıkta (sağda veya solda) sürer.
- Telefon rehberinde "Yılmaz"ı ortadan değil, sona yakın bir yerden açmaya benzer.

```
indis = bas + (aranan - dizi[bas]) ·
(son - bas) / (dizi[son] - dizi[bas])
```

**Örnek:** 10, 20, …, 100 dizisinde 70: indis = 0 + (70 − 10) · 9 / 90 = 6 → dizi[6] = 70. Düzgün dağılımda **tek yoklamada** bulunur.



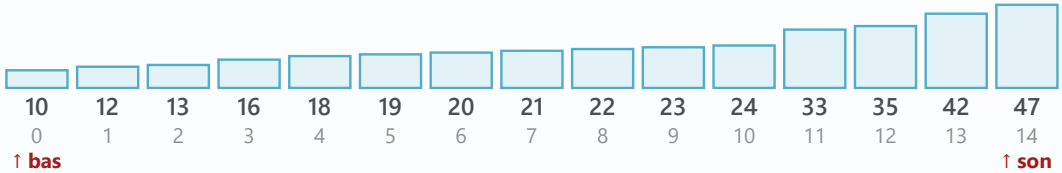

İkili arama daima ortaya bakar (oran ½); ara değer arama oranı değerlerden hesaplar.

# Ara Değer Arama Algoritması

```
int araDegerArama(int[] dizi, int n,
                  int aranan) {
  int bas = 0, son = n - 1;
  while (bas <= son && aranan >= dizi[bas]
                    && aranan <= dizi[son]) {
    if (dizi[son] == dizi[bas]) return bas; // 0'a bölme
    int pay = (aranan - dizi[bas]) * (son - bas);
    int indis = bas + pay / (dizi[son] - dizi[bas]);
    if (dizi[indis] == aranan) return indis;
    if (dizi[indis] < aranan) bas = indis + 1;
    else son = indis - 1;
  }
  return -1;
}
```

- `bas` ve `son` arama aralığının sınırlarıdır; `dizi[son] == dizi[bas]` ise aranan bu değere eşittir (0'a bölme önlenir).
- Aranan değer aralığın uç değerlerinden **küçük** veya **büyükse** döngüye girilmez, −1 döner.
- Tahmini `indis` hesaplanır ve kontrol edilir:
  - eşitse indis döner;
  - `dizi[indis]` küçükse `indis + 1` 'den,
  - büyükse `indis − 1` 'e kadar devam edilir.
- Aralık boşalırsa −1 döner.

# araDegerArama(dizi, 15, 24): Kodu Adım Adım İzle




Aranan 24. Çubuk yüksekliği değeri gösterir: dağılım **yaklaşık** düzgündür.

Adım 1 / 10

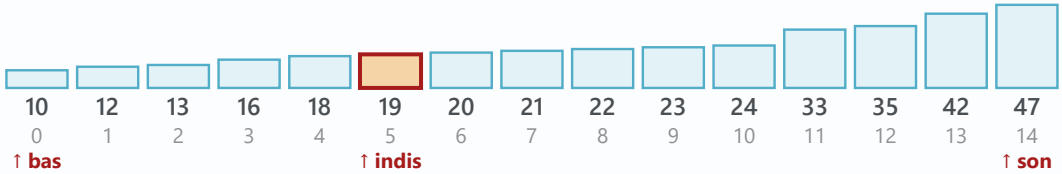
```
1  int araDegerArama(int[] dizi, int n,
2                    int aranan) {
▶  int bas = 0, son = n - 1;
4    while (bas <= son && aranan >= dizi[bas]
5                      && aranan <= dizi[son]) {
6      if (dizi[son] == dizi[bas]) return bas; // 0'a bölme
7      int pay = (aranan - dizi[bas]) * (son - bas);
8      int indis = bas + pay / (dizi[son] - dizi[bas]);
9      if (dizi[indis] == aranan) return indis;
10     if (dizi[indis] < aranan) bas = indis + 1;
11     else son = indis - 1;
12   }
13   return -1;
14 }
```

**DEĞİŞKENLER**

`aranan = 24` `bas = 0` `son = 14` `indis = –`

**ÇIKTI**

# araDegerArama(dizi, 15, 24): Kodu Adım Adım İzle




indis = 0 + (24 − 10)·14 / (47 − 10) = 0 + 196/37 → **5**; dizi[5] = 19.

Adım 2 / 10

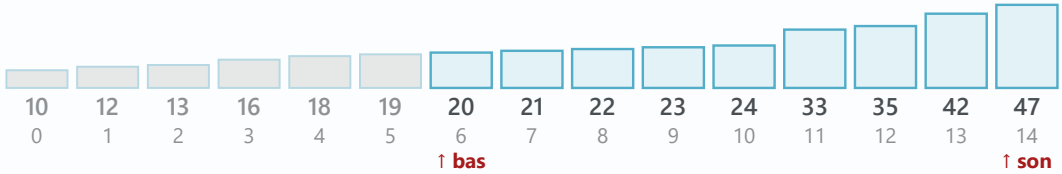
```
1  int araDegerArama(int[] dizi, int n,
2                    int aranan) {
3    int bas = 0, son = n - 1;
▶    while (bas <= son && aranan >= dizi[bas]
▶                      && aranan <= dizi[son]) {
6      if (dizi[son] == dizi[bas]) return bas; // 0'a bölme
▶      int pay = (aranan - dizi[bas]) * (son - bas);
▶      int indis = bas + pay / (dizi[son] - dizi[bas]);
9      if (dizi[indis] == aranan) return indis;
10     if (dizi[indis] < aranan) bas = indis + 1;
11     else son = indis - 1;
12   }
13   return -1;
14 }
```

**DEĞİŞKENLER**

`aranan = 24` `bas = 0` `son = 14` `indis = 5`

**ÇIKTI**

# araDegerArama(dizi, 15, 24): Kodu Adım Adım İzle




19 < 24 → `bas = 6` .

Adım 3 / 10

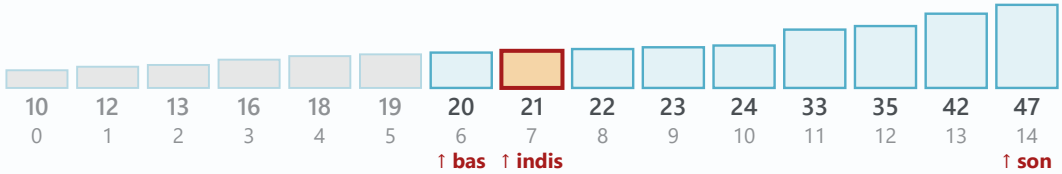
```
1  int araDegerArama(int[] dizi, int n,
2                    int aranan) {
3    int bas = 0, son = n - 1;
4    while (bas <= son && aranan >= dizi[bas]
5                      && aranan <= dizi[son]) {
6      if (dizi[son] == dizi[bas]) return bas; // 0'a bölme
7      int pay = (aranan - dizi[bas]) * (son - bas);
8      int indis = bas + pay / (dizi[son] - dizi[bas]);
9      if (dizi[indis] == aranan) return indis;
▶      if (dizi[indis] < aranan) bas = indis + 1;
11     else son = indis - 1;
12   }
13   return -1;
14 }
```

**DEĞİŞKENLER**

`aranan = 24` `bas = 6` `son = 14` `indis = 5`

**ÇIKTI**

# araDegerArama(dizi, 15, 24): Kodu Adım Adım İzle




indis = 6 + (24 − 20)·8 / (47 − 20) = 6 + 32/27 → **7**; dizi[7] = 21.

Adım 4 / 10

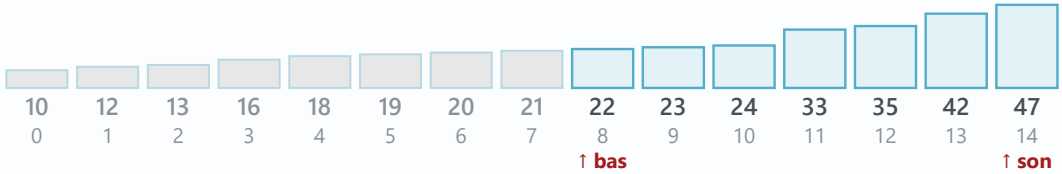
```
1  int araDegerArama(int[] dizi, int n,
2                    int aranan) {
3    int bas = 0, son = n - 1;
▶    while (bas <= son && aranan >= dizi[bas]
▶                      && aranan <= dizi[son]) {
6      if (dizi[son] == dizi[bas]) return bas; // 0'a bölme
▶      int pay = (aranan - dizi[bas]) * (son - bas);
▶      int indis = bas + pay / (dizi[son] - dizi[bas]);
9      if (dizi[indis] == aranan) return indis;
10     if (dizi[indis] < aranan) bas = indis + 1;
11     else son = indis - 1;
12   }
13   return -1;
14 }
```

**DEĞİŞKENLER**

aranan = **24** bas = **6** son = **14** indis = **7**

**ÇIKTI**

# araDegerArama(dizi, 15, 24): Kodu Adım Adım İzle




21 < 24 → `bas = 8`.

Adım 5 / 10

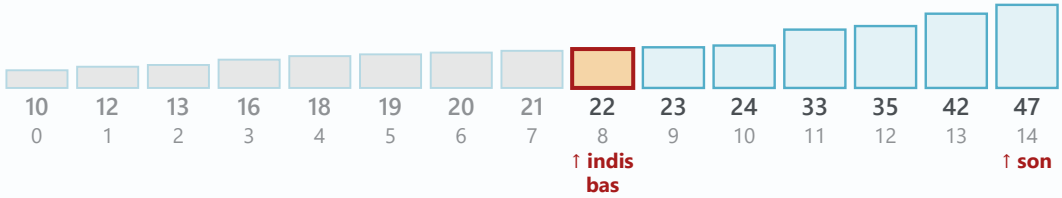
```
1  int araDegerArama(int[] dizi, int n,
2                    int aranan) {
3    int bas = 0, son = n - 1;
4    while (bas <= son && aranan >= dizi[bas]
5                      && aranan <= dizi[son]) {
6      if (dizi[son] == dizi[bas]) return bas; // 0'a bölme
7      int pay = (aranan - dizi[bas]) * (son - bas);
8      int indis = bas + pay / (dizi[son] - dizi[bas]);
9      if (dizi[indis] == aranan) return indis;
▶      if (dizi[indis] < aranan) bas = indis + 1;
11     else son = indis - 1;
12   }
13   return -1;
14 }
```

**DEĞİŞKENLER**

`aranan = 24` `bas = 8` `son = 14` `indis = 7`

**ÇIKTI**

# araDegerArama(dizi, 15, 24): Kodu Adım Adım İzle




indis = 8 + (24 − 22)·6 / (47 − 22) = 8 + 12/25 → **8**; dizi[8] = 22.

Adım 6 / 10

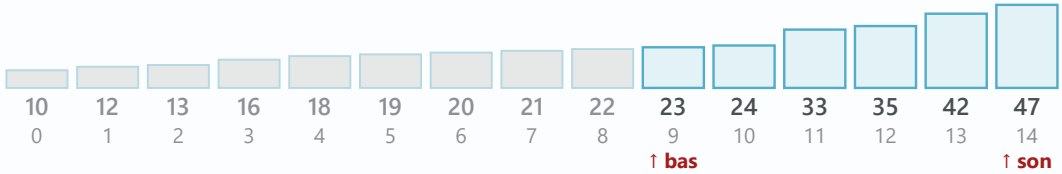
```
1  int araDegerArama(int[] dizi, int n,
2                    int aranan) {
3    int bas = 0, son = n - 1;
▶    while (bas <= son && aranan >= dizi[bas]
▶                      && aranan <= dizi[son]) {
6      if (dizi[son] == dizi[bas]) return bas; // 0'a bölme
▶      int pay = (aranan - dizi[bas]) * (son - bas);
▶      int indis = bas + pay / (dizi[son] - dizi[bas]);
9      if (dizi[indis] == aranan) return indis;
10     if (dizi[indis] < aranan) bas = indis + 1;
11     else son = indis - 1;
12   }
13   return -1;
14 }
```

**DEĞİŞKENLER**

`aranan = 24` `bas = 8` `son = 14` `indis = 8`

**ÇIKTI**

# araDegerArama(dizi, 15, 24): Kodu Adım Adım İzle




22 < 24 → `bas = 9`.

Adım 7 / 10

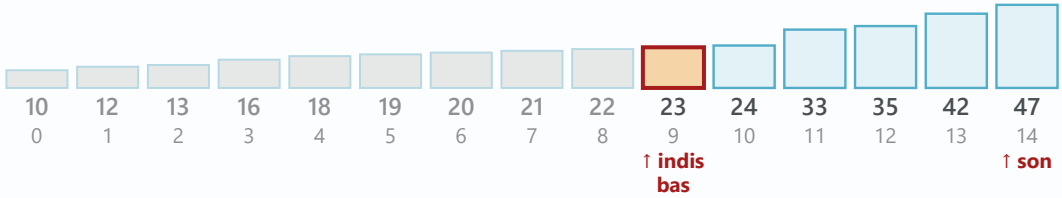
```
1  int araDegerArama(int[] dizi, int n,
2                    int aranan) {
3    int bas = 0, son = n - 1;
4    while (bas <= son && aranan >= dizi[bas]
5                      && aranan <= dizi[son]) {
6      if (dizi[son] == dizi[bas]) return bas; // 0'a bölme
7      int pay = (aranan - dizi[bas]) * (son - bas);
8      int indis = bas + pay / (dizi[son] - dizi[bas]);
9      if (dizi[indis] == aranan) return indis;
▶      if (dizi[indis] < aranan) bas = indis + 1;
11     else son = indis - 1;
12   }
13   return -1;
14 }
```

**DEĞİŞKENLER**

`aranan = 24` `bas = 9` `son = 14` `indis = 8`

**ÇIKTI**

# araDegerArama(dizi, 15, 24): Kodu Adım Adım İzle




indis = 9 + (24 − 23)·5 / (47 − 23) = 9 + 5/24 → **9**; dizi[9] = 23.

Adım 8 / 10

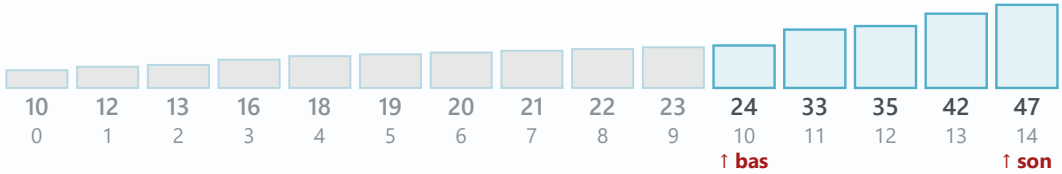
```
1  int araDegerArama(int[] dizi, int n,
2                    int aranan) {
3    int bas = 0, son = n - 1;
▶    while (bas <= son && aranan >= dizi[bas]
▶                      && aranan <= dizi[son]) {
6      if (dizi[son] == dizi[bas]) return bas; // 0'a bölme
▶      int pay = (aranan - dizi[bas]) * (son - bas);
▶      int indis = bas + pay / (dizi[son] - dizi[bas]);
9      if (dizi[indis] == aranan) return indis;
10     if (dizi[indis] < aranan) bas = indis + 1;
11     else son = indis - 1;
12   }
13   return -1;
14 }
```

**DEĞİŞKENLER**

`aranan = 24` `bas = 9` `son = 14` `indis = 9`

**ÇIKTI**

# araDegerArama(dizi, 15, 24): Kodu Adım Adım İzle




23 < 24 → `bas = 10`.

Adım 9 / 10

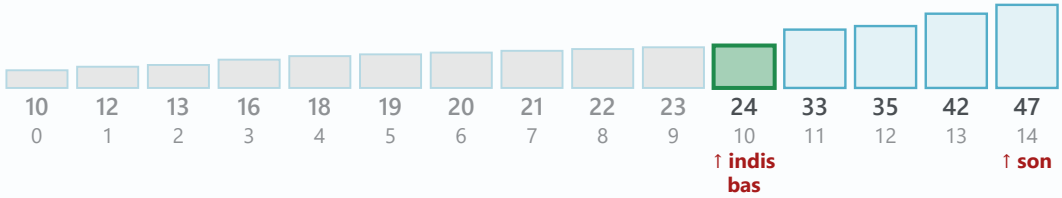
```
1  int araDegerArama(int[] dizi, int n,
2                    int aranan) {
3    int bas = 0, son = n - 1;
4    while (bas <= son && aranan >= dizi[bas]
5                      && aranan <= dizi[son]) {
6      if (dizi[son] == dizi[bas]) return bas; // 0'a bölme
7      int pay = (aranan - dizi[bas]) * (son - bas);
8      int indis = bas + pay / (dizi[son] - dizi[bas]);
9      if (dizi[indis] == aranan) return indis;
▶      if (dizi[indis] < aranan) bas = indis + 1;
11     else son = indis - 1;
12   }
13   return -1;
14 }
```

**DEĞİŞKENLER**

`aranan = 24` `bas = 10` `son = 14` `indis = 9`

**ÇIKTI**

# araDegerArama(dizi, 15, 24): Kodu Adım Adım İzle




indis = 10 + (24 − 24)·4 / (47 − 24) = 10 + 0/23 → **10**; dizi[10] = 24. Eşit → **10 döner** (5 yoklama).

Adım 10 / 10

```
1  int araDegerArama(int[] dizi, int n,
2                    int aranan) {
3    int bas = 0, son = n - 1;
4    while (bas <= son && aranan >= dizi[bas]
5                      && aranan <= dizi[son]) {
6      if (dizi[son] == dizi[bas]) return bas; // 0'a bölme
7      int pay = (aranan - dizi[bas]) * (son - bas);
8      int indis = bas + pay / (dizi[son] - dizi[bas]);
▶      if (dizi[indis] == aranan) return indis;
10     if (dizi[indis] < aranan) bas = indis + 1;
11     else son = indis - 1;
12   }
13   return -1;
14 }
```

**DEĞİŞKENLER**

`aranan = 24` `bas = 10` `son = 14` `indis = 10`

**ÇIKTI**

```
araDegerArama(dizi, 15, 24) → 10
```

# Ara Değer Arama: Karmaşıklık ve Kötü Durum

| Durum | Mertebe |
|---|---|
| En iyi | O(1) |
| Ortalama (düzgün dağılım) | O(log log n) |
| En kötü (çarpık dağılım) | O(n) |
| Ek bellek | O(1) |

- $n = 10^6$ için $\log_2 \log_2 n \approx 4{,}3$; ikili arama ≈ 20 yoklama.
- Her yoklama ikili aramadan pahalıdır (çarpma, bölme).
- Ara çarpım `(aranan - dizi[bas]) · (son - bas)` büyük değerlerde `int`'i taşırabilir; çarpım `long` ile yapılmalıdır.

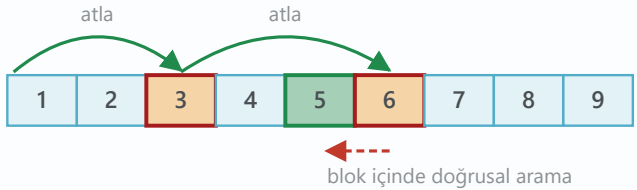



**Kötü durum örneği:** dizi = 1, 2, ..., 9, 1000 ve aranan 9. Uç değer 1000 tahmini hep sola çeker: indisler 0, 1, 2, 3, 4, 5, 6, 7, 8 → **9 yoklama**. İkili arama aynı aramayı 4, 7, 8 indisleriyle **3 yoklamada** bitirir.

ATLAYARAK ARAMA

# Atlayarak Ara Bul (Jump Search)

- Engelli koşuda engelleri aşarak ilerleyen **atlet** mantığıyla çalışır; dizi **sıralı** olmalıdır.
- Liste belirlenen büyüklükte (m = √n) **parçalara** (bloklara) ayrılır.
- Listenin başından büyük bir adım atlanır ve o konumdaki öğe (bloğun son öğesi) kontrol edilir.
- Aranan değer bu konumdaysa arama biter; daha büyükse atlamaya devam edilir.
- Değer atlanan parçadaysa, **o aralığa geri dönülerek doğrusal arama** yapılır.




Geri dönüş, ikili aramadaki gibi büyük sıçramalar gerektirmez: veri **sırayla** okunur. Bu yüzden geriye gitmenin pahalı olduğu ortamlarda (manyetik bant, blok blok okunan disk) tercih edilir.

# Atlayarak Ara Bul Algoritması

```
int atlayarakAraBul(int[] dizi, int eleman,
                    int n) {
  int blok = (int) Math.sqrt(n);
  int atlama = blok, konum = 0;
  while (dizi[Math.min(atlama, n) - 1] < eleman) {
    konum = atlama; atlama += blok; // atla
    if (konum >= n) return -1;
  }
  while (dizi[konum] < eleman) {     // doğrusal
    konum++;
    if (konum == Math.min(atlama, n)) return -1;
  }
  if (dizi[konum] == eleman) return konum;
  return -1;
}
```

- Atlama mesafesi `Math.sqrt(n)` ile hesaplanır; `konum` başta 0'dır.
- İlk döngüde her bloğun **son öğesine** bakılır; aranandan küçük oldukça bir blok atlanır.
- Döngü sonunda aranan, `[konum, atlama)` aralığında olabilir.
- İkinci döngü bu aralığı **doğrusal** tarar.
- Bulunursa indis, yoksa −1 döner.

ATLAYARAK ARAMA · KODU ADIM ADIM ÇALIŞTIR

# atlayarakAraBul(dizi, 69, 10): Kodu Adım Adım İzle

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 9 | 30 | 32 | 38 | 47 | 61 | 69 | 79 | 81 |

↑ konum ↑ atlama−1

Aranan 69; n = 10 → blok = ⌊√10⌋ = 3; atlama = 3, konum = 0.

Adım 1 / 6

```
1  int atlayarakAraBul(int[] dizi, int eleman,
2                      int n) {
▶    int blok = (int) Math.sqrt(n);
▶    int atlama = blok, konum = 0;
5    while (dizi[Math.min(atlama, n) - 1] < eleman) {
6      konum = atlama; atlama += blok; // atla
7      if (konum >= n) return -1;
8    }
9    while (dizi[konum] < eleman) {     // doğrusal
10     konum++;
11     if (konum == Math.min(atlama, n)) return -1;
12   }
13   if (dizi[konum] == eleman) return konum;
14   return -1;
15 }
```

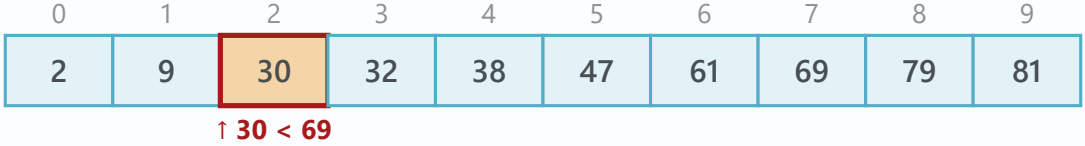
**DEĞİŞKENLER**

eleman = 69 | blok = 3 | atlama = 3 | konum = 0

**ÇIKTI**

ATLAYARAK ARAMA · KODU ADIM ADIM ÇALIŞTIR

# atlayarakAraBul(dizi, 69, 10): Kodu Adım Adım İzle




dizi[min(3, 10) − 1] = dizi[2] = 30 < 69 → bir blok atla: konum = 3, atlama = 6.

Adım 2 / 6

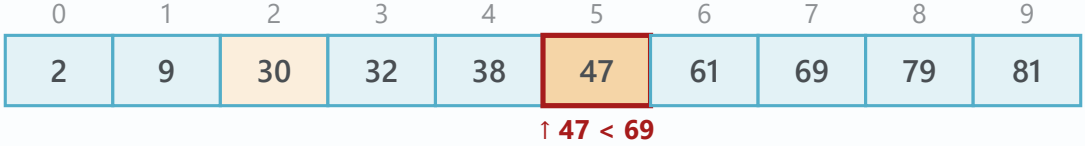
```
1  int atlayarakAraBul(int[] dizi, int eleman,
2                      int n) {
3    int blok = (int) Math.sqrt(n);
4    int atlama = blok, konum = 0;
▶    while (dizi[Math.min(atlama, n) - 1] < eleman) {
▶      konum = atlama; atlama += blok; // atla
7      if (konum >= n) return -1;
8    }
9    while (dizi[konum] < eleman) {     // doğrusal
10     konum++;
11     if (konum == Math.min(atlama, n)) return -1;
12   }
13   if (dizi[konum] == eleman) return konum;
14   return -1;
15 }
```

**DEĞİŞKENLER**

eleman = 69 | blok = 3 | atlama = 6 | konum = 3

**ÇIKTI**

# atlayarakAraBul(dizi, 69, 10): Kodu Adım Adım İzle




dizi[min(6, 10) − 1] = dizi[5] = 47 < 69 → bir blok atla: konum = 6, atlama = 9.

Adım 3 / 6

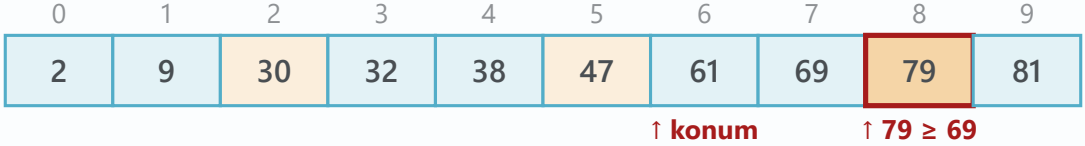
```
1  int atlayarakAraBul(int[] dizi, int eleman,
2                      int n) {
3    int blok = (int) Math.sqrt(n);
4    int atlama = blok, konum = 0;
▶    while (dizi[Math.min(atlama, n) - 1] < eleman) {
▶      konum = atlama; atlama += blok; // atla
7      if (konum >= n) return -1;
8    }
9    while (dizi[konum] < eleman) {     // doğrusal
10     konum++;
11     if (konum == Math.min(atlama, n)) return -1;
12   }
13   if (dizi[konum] == eleman) return konum;
14   return -1;
15 }
```

**DEĞİŞKENLER**

eleman = 69 | blok = 3 | atlama = 9 | konum = 6

**ÇIKTI**

# atlayarakAraBul(dizi, 69, 10): Kodu Adım Adım İzle




dizi[8] = 79 ≥ 69 → atlama biter. Aranan, [6, 9) bloğunda olabilir.

Adım 4 / 6

```
1  int atlayarakAraBul(int[] dizi, int eleman,
2                      int n) {
3    int blok = (int) Math.sqrt(n);
4    int atlama = blok, konum = 0;
▶    while (dizi[Math.min(atlama, n) - 1] < eleman) {
6      konum = atlama; atlama += blok; // atla
7      if (konum >= n) return -1;
8    }
9    while (dizi[konum] < eleman) {     // doğrusal
10     konum++;
11     if (konum == Math.min(atlama, n)) return -1;
12   }
13   if (dizi[konum] == eleman) return konum;
14   return -1;
15 }
```

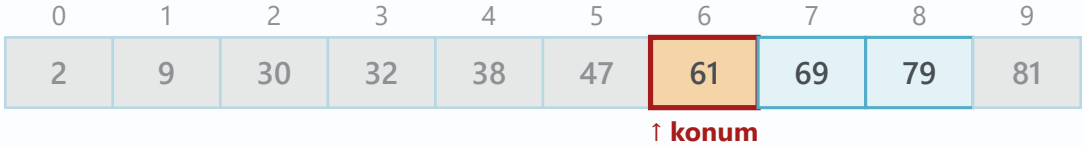
**DEĞİŞKENLER**

`eleman = 69` `blok = 3`

`atlama = 9` `konum = 6`

**ÇIKTI**

# atlayarakAraBul(dizi, 69, 10): Kodu Adım Adım İzle




Doğrusal arama: dizi[6] = 61 < 69 → konum = 7.

Adım 5 / 6

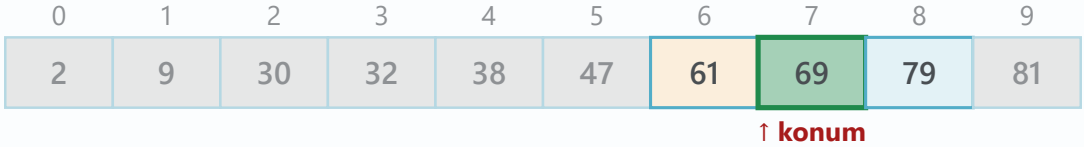
```
1  int atlayarakAraBul(int[] dizi, int eleman,
2                      int n) {
3    int blok = (int) Math.sqrt(n);
4    int atlama = blok, konum = 0;
5    while (dizi[Math.min(atlama, n) - 1] < eleman) {
6      konum = atlama; atlama += blok; // atla
7      if (konum >= n) return -1;
8    }
▶    while (dizi[konum] < eleman) {     // doğrusal
▶      konum++;
▶      if (konum == Math.min(atlama, n)) return -1;
12   }
13   if (dizi[konum] == eleman) return konum;
14   return -1;
15 }
```

**DEĞİŞKENLER**

`eleman = 69` `blok = 3`
`atlama = 9` `konum = 7`

**ÇIKTI**

# atlayarakAraBul(dizi, 69, 10): Kodu Adım Adım İzle




dizi[7] = 69 → **7 döner**. Yoklananlar: 2, 5, 8, 6, 7.

Adım 6 / 6

```
1  int atlayarakAraBul(int[] dizi, int eleman,
2                      int n) {
3    int blok = (int) Math.sqrt(n);
4    int atlama = blok, konum = 0;
5    while (dizi[Math.min(atlama, n) - 1] < eleman) {
6      konum = atlama; atlama += blok; // atla
7      if (konum >= n) return -1;
8    }
9    while (dizi[konum] < eleman) {     // doğrusal
10     konum++;
11     if (konum == Math.min(atlama, n)) return -1;
12   }
▶    if (dizi[konum] == eleman) return konum;
14   return -1;
15 }
```

**DEĞİŞKENLER**

`eleman = 69` `blok = 3`

`atlama = 9` `konum = 7`

**ÇIKTI**

```
atlayarakAraBul(dizi, 69, 10) → 7
```

# Atlayarak Arama: Neden √n?

**`en kötü yoklama ≈ n/m + (m − 1)`**

- n/m: en fazla atlama sayısı; m − 1: blok içindeki doğrusal arama.
- Türev: $-n/m^2 + 1 = 0 \Rightarrow$ **m = √n**; toplam ≈ 2√n − 1.
- Karmaşıklık **O(√n)**: doğrusal O(n) ile ikili O(log n) arasında.
- Ek bellek O(1); dizi sıralı olmalıdır.

| n | m = √n | ≈ 2√n − 1 | ikili |
|---|---|---|---|
| 100 | 10 | 19 | 7 |
| 10 000 | 100 | 199 | 14 |
| $10^6$ | 1 000 | 1 999 | 20 |

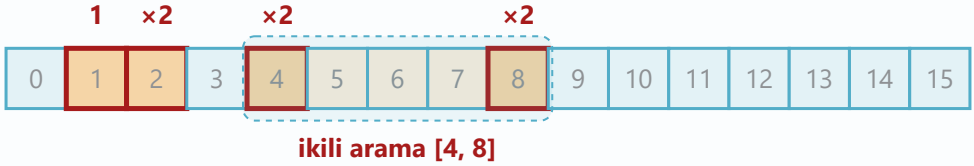
Blok bulunduktan sonra blok sondan geriye (8 → 7) ya da `atlayarakAraBul` kodundaki gibi baştan ileri (6 → 7) taranabilir. İkisi de en fazla m − 1 ek yoklama yapar.

İyileştirme: blok bulunduktan sonra içeride doğrusal yerine yine atlayarak (√m adımla) ya da ikili arama yapılabilir. İkili arama varken atlayarak arama, yalnızca **geri gitmenin pahalı** olduğu durumlarda anlamlıdır.

ÜSTEL ARAMA

# Üstel (Exponential) Arama

- Sıralı listeyi **üstel olarak büyüyen** parçalara ayırır: sınır (konum) 1, 2, 4, 8, 16, … diye **ikiye katlanır**.
- `dizi[konum]` aranandan büyük olunca durulur: aranan **[konum/2, konum]** aralığındadır.
- Bu aralıkta **ikili arama** yapılır (Bentley ve Yao, 1976).
- Örneğin 1024 öğeli bir listede aranan indis 600 civarındaysa sınırlar 1, 2, …, 512, 1024'tür; arama 512 ile 1023 arasında ikili aramayla sürer.




**Neden?** Dizinin boyu bilinmiyorsa (sınırsız / akan veri) ya da aranan başa yakınsa idealdir: maliyet n'ye değil, aranan öğenin **konumuna** (i) bağlıdır.

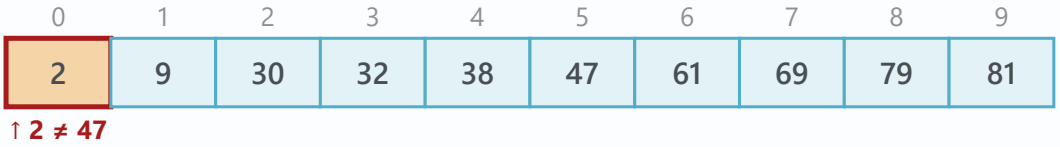
# Üstel Arama Algoritması

```
int ustelArama(int[] dizi, int aranan, int n) {
  if (dizi[0] == aranan) return 0;
  int konum = 1;
  while (konum < n && dizi[konum] <= aranan)
    konum *= 2;                // 1, 2, 4, 8, ...
  int bas = konum / 2, son = Math.min(konum, n - 1);
  while (bas <= son) {         // ikili arama
    int orta = (bas + son) / 2;
    if (dizi[orta] == aranan) return orta;
    if (dizi[orta] < aranan) bas = orta + 1;
    else son = orta - 1;
  }
  return -1;
}
```

- `konum` başta 1'dir; `dizi[konum] ≤ aranan` oldukça 2 ile çarpılır.
- Döngü sonunda `konum`, aranılan değerin olabileceği aralığın **sonunu** işaret eder; `konum / 2` başıdır.
- `Math.min(konum, n − 1)`: sınır dizinin dışına taşmamalıdır.
- Kalan kısım ikili aramayla taranır.

Son aşamada ikili arama yerine **doğrusal** tarama ( `konum / 2` 'den sona doğru) yapılırsa maliyet O(i) olur ve aranan yoksa dizinin sonuna kadar gidilir. İkili arama ile maliyet O(log i)'ye iner.

# ustelArama(dizi, 47, 10): Kodu Adım Adım İzle




Aranan 47. Önce dizi[0] = 2 ≠ 47.

Adım 1 / 8

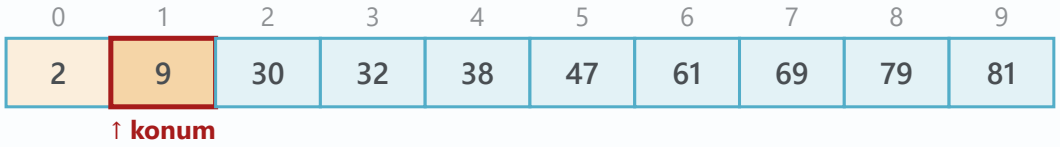
```
1  int ustelArama(int[] dizi, int aranan, int n) {
▶    if (dizi[0] == aranan) return 0;
▶    int konum = 1;
4    while (konum < n && dizi[konum] <= aranan)
5      konum *= 2;               // 1, 2, 4, 8, ...
6    int bas = konum / 2, son = Math.min(konum, n - 1);
7    while (bas <= son) {        // ikili arama
8      int orta = (bas + son) / 2;
9      if (dizi[orta] == aranan) return orta;
10     if (dizi[orta] < aranan) bas = orta + 1;
11     else son = orta - 1;
12   }
13   return -1;
14 }
```

**DEĞİŞKENLER**

aranan = 47 | konum = 1 | bas = – | son = – | orta = –

**ÇIKTI**

# ustelArama(dizi, 47, 10): Kodu Adım Adım İzle




dizi[1] = 9 ≤ 47 → konum = 2.

Adım 2 / 8

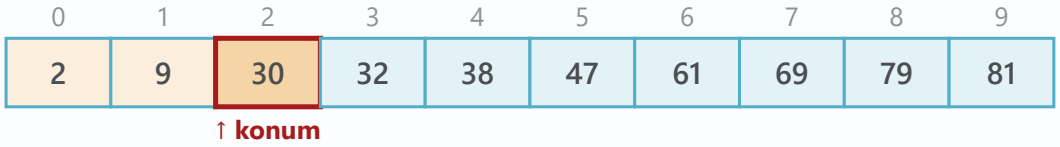
```
1  int ustelArama(int[] dizi, int aranan, int n) {
2    if (dizi[0] == aranan) return 0;
3    int konum = 1;
▶    while (konum < n && dizi[konum] <= aranan)
▶      konum *= 2;               // 1, 2, 4, 8, ...
6    int bas = konum / 2, son = Math.min(konum, n - 1);
7    while (bas <= son) {         // ikili arama
8      int orta = (bas + son) / 2;
9      if (dizi[orta] == aranan) return orta;
10     if (dizi[orta] < aranan) bas = orta + 1;
11     else son = orta - 1;
12   }
13   return -1;
14 }
```

**DEĞİŞKENLER**

aranan = 47 | konum = 2 | bas = – | son = – | orta = –

**ÇIKTI**

# ustelArama(dizi, 47, 10): Kodu Adım Adım İzle




dizi[2] = 30 ≤ 47 → konum = 4.

Adım 3 / 8

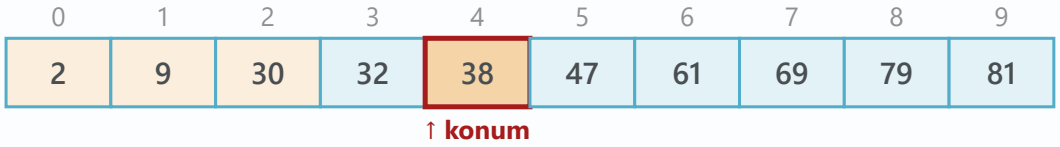
```
 1 int ustelArama(int[] dizi, int aranan, int n) {
 2   if (dizi[0] == aranan) return 0;
 3   int konum = 1;
▶  while (konum < n && dizi[konum] <= aranan)
▶    konum *= 2;                // 1, 2, 4, 8, ...
 6   int bas = konum / 2, son = Math.min(konum, n - 1);
 7   while (bas <= son) {         // ikili arama
 8     int orta = (bas + son) / 2;
 9     if (dizi[orta] == aranan) return orta;
10     if (dizi[orta] < aranan) bas = orta + 1;
11     else son = orta - 1;
12   }
13   return -1;
14 }
```

**DEĞİŞKENLER**

aranan = 47 | konum = 4 | bas = – | son = – | orta = –

**ÇIKTI**

# ustelArama(dizi, 47, 10): Kodu Adım Adım İzle




dizi[4] = 38 ≤ 47 → konum = 8.

Adım 4 / 8

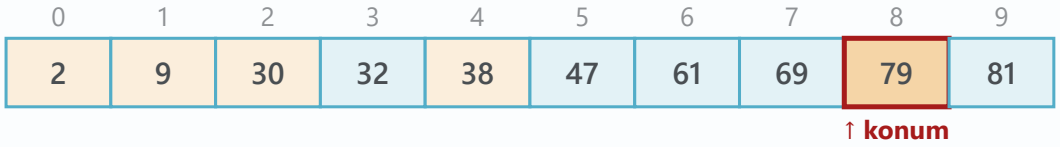
```
1  int ustelArama(int[] dizi, int aranan, int n) {
2    if (dizi[0] == aranan) return 0;
3    int konum = 1;
▶    while (konum < n && dizi[konum] <= aranan)
▶      konum *= 2;               // 1, 2, 4, 8, ...
6    int bas = konum / 2, son = Math.min(konum, n - 1);
7    while (bas <= son) {         // ikili arama
8      int orta = (bas + son) / 2;
9      if (dizi[orta] == aranan) return orta;
10     if (dizi[orta] < aranan) bas = orta + 1;
11     else son = orta - 1;
12   }
13   return -1;
14 }
```

**DEĞİŞKENLER**

aranan = 47 | konum = 8 | bas = – | son = – | orta = –

**ÇIKTI**

# ustelArama(dizi, 47, 10): Kodu Adım Adım İzle




dizi[8] = 79 > 47 → katlama biter. Aranan [8/2, min(8, 9)] = [4, 8] aralığındadır.

Adım 5 / 8

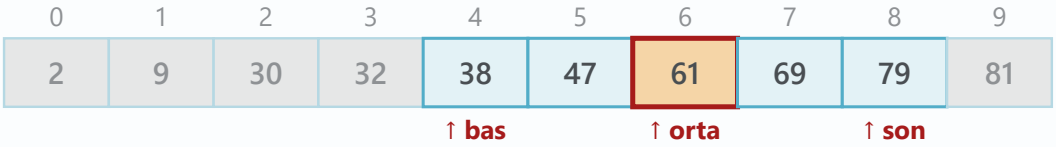
```
1  int ustelArama(int[] dizi, int aranan, int n) {
2    if (dizi[0] == aranan) return 0;
3    int konum = 1;
▶    while (konum < n && dizi[konum] <= aranan)
5      konum *= 2;               // 1, 2, 4, 8, ...
▶    int bas = konum / 2, son = Math.min(konum, n - 1);
7    while (bas <= son) {        // ikili arama
8      int orta = (bas + son) / 2;
9      if (dizi[orta] == aranan) return orta;
10     if (dizi[orta] < aranan) bas = orta + 1;
11     else son = orta - 1;
12   }
13   return -1;
14 }
```

**DEĞİŞKENLER**

aranan = 47 | konum = 8 | bas = 4 | son = 8 | orta = –

**ÇIKTI**

# ustelArama(dizi, 47, 10): Kodu Adım Adım İzle




İkili arama: orta = (4 + 8) / 2 = 6 → dizi[6] = 61 > 47 → son = 5.

Adım 6 / 8

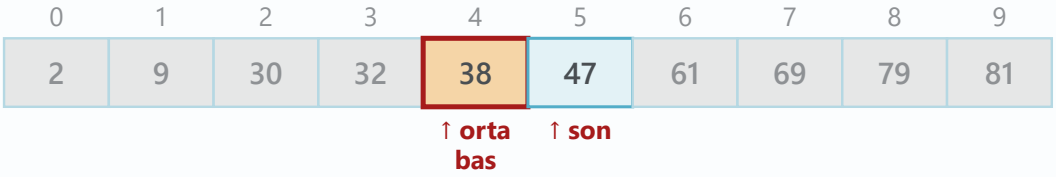
```
1  int ustelArama(int[] dizi, int aranan, int n) {
2    if (dizi[0] == aranan) return 0;
3    int konum = 1;
4    while (konum < n && dizi[konum] <= aranan)
5      konum *= 2;                // 1, 2, 4, 8, ...
6    int bas = konum / 2, son = Math.min(konum, n - 1);
7    while (bas <= son) {         // ikili arama
▶      int orta = (bas + son) / 2;
▶      if (dizi[orta] == aranan) return orta;
▶      if (dizi[orta] < aranan) bas = orta + 1;
▶      else son = orta - 1;
12   }
13   return -1;
14 }
```

**DEĞİŞKENLER**

aranan = 47 | konum = 8 | bas = 4 | son = 8 | orta = 6

**ÇIKTI**

# ustelArama(dizi, 47, 10): Kodu Adım Adım İzle




İkili arama: orta = (4 + 5) / 2 = 4 → dizi[4] = 38 < 47 → bas = 5.

Adım 7 / 8

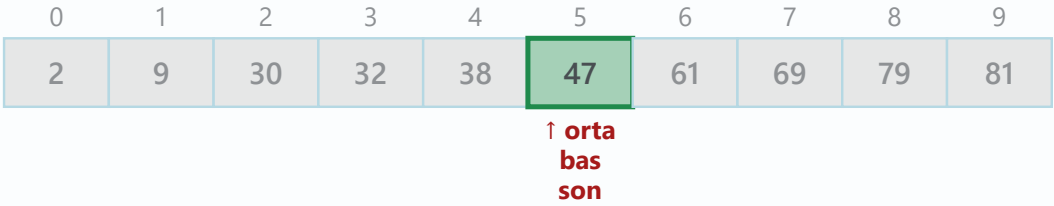
```
1  int ustelArama(int[] dizi, int aranan, int n) {
2    if (dizi[0] == aranan) return 0;
3    int konum = 1;
4    while (konum < n && dizi[konum] <= aranan)
5      konum *= 2;               // 1, 2, 4, 8, ...
6    int bas = konum / 2, son = Math.min(konum, n - 1);
7    while (bas <= son) {        // ikili arama
▶      int orta = (bas + son) / 2;
▶      if (dizi[orta] == aranan) return orta;
▶      if (dizi[orta] < aranan) bas = orta + 1;
11     else son = orta - 1;
12   }
13   return -1;
14 }
```

**DEĞİŞKENLER**

aranan = 47 | konum = 8 | bas = 4 | son = 5 | orta = 4

**ÇIKTI**

# ustelArama(dizi, 47, 10): Kodu Adım Adım İzle




İkili arama: orta = (5 + 5) / 2 = 5 → dizi[5] = 47 = 47 → **5 döner**.

Adım 8 / 8

```
 1 int ustelArama(int[] dizi, int aranan, int n) {
 2   if (dizi[0] == aranan) return 0;
 3   int konum = 1;
 4   while (konum < n && dizi[konum] <= aranan)
 5     konum *= 2;               // 1, 2, 4, 8, ...
 6   int bas = konum / 2, son = Math.min(konum, n - 1);
 7   while (bas <= son) {        // ikili arama
▶      int orta = (bas + son) / 2;
▶      if (dizi[orta] == aranan) return orta;
10     if (dizi[orta] < aranan) bas = orta + 1;
11     else son = orta - 1;
12   }
13   return -1;
14 }
```

**DEĞİŞKENLER**

`aranan = 47` `konum = 8`

`bas = 5` `son = 5` `orta = 5`

**ÇIKTI**

```
ustelArama(dizi, 47, 10) → 5
```

# Üstel Arama: Karmaşıklık

- Aranan i indisindeyse katlama aşaması $\lfloor \log_2 i \rfloor + 1$ adım sürer.
- Son aralık [i/2, i] en fazla i/2 + 1 öğelidir → ikili arama ≈ $\log_2 i$ adım.
- Toplam: **O(log i)**; en kötü (i ≈ n) **O(log n)**, en iyi O(1). Ek bellek O(1).
- Aranan başa yakınsa ikili aramadan hızlıdır; sona yakınsa ≈ 2 kat yavaştır.

| i ($n = 10^6$) | üstel ($\approx 2 \log_2 i$) | ikili ($\approx \log_2 n$) |
|---|---|---|
| 10 | ≈ 7 | 20 |
| 1 000 | ≈ 20 | 20 |
| 500 000 | ≈ 38 | 20 |

**Kullanım:** sınırsız (boyu bilinmeyen) sıralı listeler, büyük bir dosyada başa yakın kayıtlar ve sıralı veri akışlarında "galop" (galloping) araması; örneğin Timsort birleştirmede bu fikri kullanır.

KARŞILAŞTIRMA

# Arama Algoritmalarının Karşılaştırılması

| Algoritma | Önkoşul | En iyi | Ortalama | En kötü | Ek bellek |
|---|---|---|---|---|---|
| Doğrusal | yok | O(1) | O(n) | O(n) | O(1) |
| İkili | sıralı | O(1) | O(log n) | O(log n) | O(1) |
| Hash (tablo) | hash tablosu | O(1) | O(1) | O(n) | O(m) |
| Ara değer | sıralı, düzgün | O(1) | O(log log n) | O(n) | O(1) |
| Atlayarak | sıralı | O(1) | O(√n) | O(√n) | O(1) |
| Üstel | sıralı | O(1) | O(log i) | O(log n) | O(1) |

i: aranan öğenin indisi, m: kova sayısı. Hash tablosunun ortalama O(1)'i, yük faktörünün sabit tutulmasına bağlıdır. Sıralama gerekiyorsa ona bir kez O(n log n) eklenir.

KARŞILAŞTIRMA

# Aynı Dizide Yoklama Sayıları

Dizi: **2, 9, 30, 32, 38, 47, 61, 69, 79, 81** (atlayarak ve üstel arama örneklerindeki dizi).
Hücre: yoklama sayısı (yoklanan indisler).

| aranan | Doğrusal | İkili | Atlayarak | Ara değer | Üstel | sonuç |
|---|---|---|---|---|---|---|
| **69** | 8 (0,1,2,3,4,5,6,7) | 2 (4,7) | 5 (2,5,8,6,7) | 1 (7) | 7 (0,1,2,4,8,6,7) | 7 |
| **47** | 6 (0,1,2,3,4,5) | 3 (4,7,5) | 5 (2,5,3,4,5) | 1 (5) | 8 (0,1,2,4,8,6,4,5) | 5 |
| **2** | 1 (0) | 3 (4,1,0) | 2 (2,0) | 1 (0) | 1 (0) | 0 |
| **50** | 10 (0,1,2,3,4,5,6,7,8,9) | 4 (4,7,5,6) | 4 (2,5,8,6) | 1 (5) | 8 (0,1,2,4,8,6,4,5) | -1 |

Dizi neredeyse düzgün dağıldığı için ara değer arama her sorguda **tek yoklama** yapar. 50 aranırken ara değer arama 47'yi yoklar, ardından aranan aralık dışında kaldığı için hemen −1 döner. Üstel aramada 4 indisi iki kez (katlama ve ikili arama aşamasında) yoklanır.

KARŞILAŞTIRMA

# Hangi Algoritma Ne Zaman?

### Doğrusal

Sırasız ya da küçük veri; bağlı liste; tek seferlik arama.

### İkili

Sıralı dizi, çok sayıda sorgu; genel amaçlı varsayılan seçim.

### Hash

Yalnızca tam eşleşme sorgusu ve çok sayıda ekleme / arama; sıra gerekmiyor.

### Ara değer

Sıralı ve düzgün dağılmış sayısal anahtarlar (ör. sıralı kimlik numaraları).

### Atlayarak

Geriye gitmenin pahalı olduğu sıralı ortamlar (bant, blok okuma).

### Üstel

Boyu bilinmeyen sıralı liste ya da aranan başa yakın.

Soru sırası: **Veri sıralı mı?** Hayır → doğrusal ya da hash tablosu kur. Evet → ikili arama; dağılım düzgünse ara değer, boy bilinmiyorsa üstel, geri gitmek pahalıysa atlayarak.

KARŞILAŞTIRMA

# Arama Yarışı: Aynı Dizide Beş Algoritma

Aranan 65

doğrusal | ikili | atlayarak | ara değer | üstel | düzgün dizi | çarpık dizi

| 3 | 8 | 15 | 21 | 27 | 34 | 40 | 46 | 52 | 59 | 65 | 71 | 78 | 84 | 90 | 97 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

Bir değer girip algoritma seçin (Enter = ikili).

Hücrenin üstündeki sayı yoklama sırasıdır; yeşil: bulundu. Tablo, son aranan değer için beş algoritmanın yoklama sayılarını karşılaştırır. Çarpık dizide ara değer arama yavaşlar.

ÖZET

# Özet

| Konu | Hatırlanacaklar |
|---|---|
| **Doğrusal** | baştan sona tek tek karşılaştır; önkoşul yok; ortalama (n + 1)/2, en kötü n → O(n) |
| **İkili** | sıralı dizi; orta ile karşılaştır, yarıyı ele; en fazla $\lfloor \log_2 n \rfloor + 1$ → O(log n); özyinelemeli sürüm O(log n) yığın |
| **Hash** | yeri hesapla: h(anahtar); çakışma → doğrusal yoklama ya da zincirleme; ort. O(1), en kötü O(n) |
| **Ara değer** | konumu değer oranıyla tahmin et; düzgün dağılımda O(log log n), çarpıkta O(n) |
| **Atlayarak** | √n'lik bloklar atla, sonra blok içinde doğrusal ara → O(√n) |
| **Üstel** | sınırı 1, 2, 4, … katla, [konum/2, konum]'da ikili arama → O(log i) |

ÖZET

# Etkileşimli Simülatörler

## Arama Algoritmaları

Rastgele 16 öğeli bir dizide doğrusal, ikili ve atlayarak aramayı adım adım canlandırır; ikili ve atlayarak aramadan önce dizi sıralanır, yoklanan hücre vurgulanır.

## Hash ve Ara Değer Arama

8 kovalı (x mod 8) bir hash tablosunda kova aramasını ve 12 öğeli sıralı dizide ara değer aramasını yan yana dener; yoklama sayısını raporlar.

## Üstel Arama

20 öğeli sıralı dizide sınırı 1, 2, 4, 8, … diye büyütür, ardından bulunan aralıkta ikili arama yapar ve tüm yoklama dizisini yazar.

ÖZET

# Sık Yapılan Hatalar

### Sırasız dizide ikili arama

Hata vermez, ama yanlış sonuç (−1) döndürebilir. İkili, ara değer, atlayarak ve üstel arama **sıralı** dizi ister.

### Sınırlarda ±1 hatası

`while (bas < son)` tek öğeli aralığı atlar; `son = orta` sonsuz döngüye sokabilir.

### Orta hesabında taşma

Çok büyük dizilerde `(bas + son) / 2` taşar; `bas + (son - bas) / 2` taşmaz.

### Ara değerde bölme

`dizi[son] == dizi[bas]` ise sıfıra bölme; aranan uç değerlerin dışındaysa indis dizi dışına çıkar. Önce aralık denetlenmelidir.

### Dolu hash tablosu

Doğrusal yoklamada boş kova kalmazsa arama döngüsü bitmez. Yük faktörü sınırlanmalı ya da m yoklamadan sonra durulmalıdır.

### Üstel aramada sınır

`Math.min(konum, n - 1)` unutulursa `ArrayIndexOutOfBoundsException`; son aşamada doğrusal tarama O(log i)'yi bozar.

ÖZET

# Örnek Problemler

Dizi D = **3, 8, 15, 21, 27, 34, 40, 46, 52, 59, 65, 71, 78, 84, 90, 97** (n = 16, indisler 0'dan).

1. İkili aramada 65 için hangi indisler yoklanır?
2. Atlayarak aramada 65 için yoklanan indisler nelerdir?
3. Ara değer arama 65 ve 30'u kaç yoklamada sonuçlandırır?
4. Üstel aramada 65 için yoklanan indisler nelerdir?
5. m = 7, h(x) = x % 7, doğrusal yoklama: 10, 24, 17, 31, 5 eklenir. Son tablo? 12 aranırken kaç kova yoklanır?

▶ **Cevapları göster**

ÖZET

# Alıştırmalar

1. Aralığı her adımda iki yerine **üç** parçaya bölen (iki orta öğeye bakan) üçlü arama, ikili aramadan daha az karşılaştırma yapar mı?
2. Tekrarlı öğeler içeren sıralı dizide aranan değerin **ilk** konumunu bulan ikili arama yazın.
3. Başarılı doğrusal aramada ortalama karşılaştırma sayısının $(n + 1)/2$ olduğunu gösterin.
4. Atlayarak aramada en iyi blok boyunun $\sqrt{n}$ olduğunu gösterin; $n = 10\,000$ için en kötü yoklama sayısı?
5. 10, 20, ..., 100 dizisinde ara değer arama 70'i kaç yoklamada bulur? 1, 2, ..., 9, 1000 dizisinde 9'u?
6. Zincirleme hash tablosunda ($m = 10$) 16, 12, 45, 29, 6, 122, 5, 58, 75 varken 35 aranırsa kaç anahtar karşılaştırılır?

▶ **Cevaplar / ipuçları**

ÖZET

# Kaynaklar


- Cormen, Leiserson, Rivest ve Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022 (Bölüm 2.3, 11).
- Sedgewick ve Wayne, *Algorithms*, 4. baskı, 2011 (Bölüm 3.1 İkili arama, 3.4 Hash tabloları).
- D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2. baskı, 1998.
- J. L. Bentley ve A. C. Yao, "An Almost Optimal Algorithm for Unbounded Searching", *Information Processing Letters* 5(3), 1976.
- Y. Perl, A. Itai ve H. Avni, "Interpolation Search — A Log Log N Search", *Communications of the ACM* 21(7), 1978.
- B. Shneiderman, "Jump Searching: A Fast Sequential Search Technique", *Communications of the ACM* 21(10), 1978.

# Son

## Bölüm 3: Arama Algoritmaları

Sonraki bölüm: Bölüm 4.1: Çizge Gezinme Algoritmaları · Terim sözlüğü