# Bölüm 11: Öncelikli Kuyruklar ve Heap

## Veri Yapıları

Öncelikli kuyruk ADT | Liste ile gerçekleme | İkili heap | Dizi gösterimi | Up-heap / down-heap | Heap oluşturma | Heap sort | Öncelikli çizelgeleyici

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

GİRİŞ

# Öğrenme Çıktıları

**ADT modeli**

Öncelikli kuyruğun temel işlemlerini ve kullanım alanlarını açıklamak.

**İkili heap**

Heap düzeni ve tam ikili ağaç özelliğini dizi gösterimiyle ilişkilendirmek.

**İşlemler**

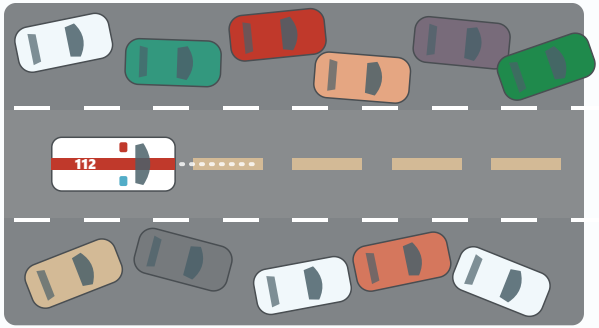
Ekleme, kökü çıkarma ve heap oluşturma adımlarını uygulamak.

**Maliyet analizi**

İşlem maliyetlerini ve Heap Sort algoritmasını değerlendirmek.

# Öncelikli Kuyruk (Priority Queue)

- Her öğenin bir **önceliği** (anahtarı) vardır; öğeler öncelik sırasına göre çıkar.
- Geliş sırası değil, öncelik belirleyicidir: ambulans trafikte öne geçer.
- Öncelikli işlerin yönetiminde kullanılır: görev çizelgeleme, acil servis, olay simülasyonu.
- Maliyetler gerçeklemeye bağlıdır: heap ile en öncelikli öğeye **bakmak** O(1), onu **çıkarmak** O(log n); sırasız listede bakmak bile O(n).




Önce gelen değil, önceliği yüksek olan geçer.

# Temel Kavramlar

### Öğe = (anahtar, değer)

Anahtar (key) önceliği, değer (value) asıl veriyi taşır.

`(1, "kalp krizi") · (4, "burkulma")`

### Öncelik

Anahtarlar karşılaştırılabilir olmalı (<, ≤). Genelde *küçük anahtar = yüksek öncelik* (min-öncelikli kuyruk).

### En yüksek öncelik

Kuyruğun "önündeki" öğe: min-kuyrukta en küçük, max-kuyrukta en büyük anahtar.

### Eşit öncelik

Eşit anahtarlı öğeler için FIFO sırası *kendiliğinden garanti değildir*; gerekiyorsa anahtara bir geliş sıra numarası eklenir.

ADT tanımında Goodrich'teki gibi min-öncelikli kuyruk (removeMin) kullanılır; heap gerçekleme örnekleri ise max-heap (ekle / silMax) üzerindedir. Hangisinin seçildiği yalnızca karşılaştırma yönünü değiştirir.

# Kuyruk mu, Öncelikli Kuyruk mu?

Acil servise sırasıyla dört hasta geliyor (küçük sayı = daha acil):

| **Geliş** | **1.** | **2.** | **3.** | **4.** |
|---|---|---|---|---|
| **Hasta (öncelik)** | Ali (3) | Ece (1) | Can (2) | Naz (1) |
| **Kuyruk (FIFO)** | Ali | Ece | Can | Naz |
| **Öncelikli kuyruk** | Ece | Naz | Can | Ali |

**Kuyruk**

Çıkış sırası = geliş sırası. Önce gelen önce hizmet alır.

**Öncelikli kuyruk**

Çıkış sırası = öncelik sırası. Ece ve Naz eşit (1); burada geliş sırasını koruduk.

# Kullanım Alanları

### İşletim Sistemleri

Görev (süreç) çizelgeleme: yüksek öncelikli süreç işlemciyi önce alır.

`kesme işleyici > arayüz > yedekleme`

### Çizge Algoritmaları

Dijkstra en kısa yol ve A* araması, sıradaki en yakın düğümü öncelikli kuyruktan alır.

### Acil Durum Yönetimi

Hasta triyajı ve olay yönetimi: en acil vaka önce ele alınır.

### Veri Sıkıştırma

Huffman kodlaması, her adımda en düşük frekanslı iki düğümü birleştirir.

# Temel İşlemler

| İşlem | Anlamı |
|---|---|
| `insert(k, v)` | **Ekleme**: öğe, önceliğine göre yerini alır. |
| `removeMin()` | **Çıkarma**: en yüksek öncelikli öğeyi çıkarır ve döndürür. |
| `min()` | **Sorgulama** (peek): öğeyi döndürür, çıkarmaz. |
| `size()` | Öğe sayısı. |
| `isEmpty()` | Kuyruk boş mu? |

```
public interface PriorityQueue<K, V> {
    void insert(K key, V value);
    Entry<K, V> min();         // boşsa null
    Entry<K, V> removeMin();   // boşsa null
    int size();
    boolean isEmpty();
}
```

Max-öncelikli kuyrukta aynı işlemler `max()` ve `removeMax()` adını alır. Bu bölümün heap kod örnekleri **max-heap** (`ekle`, `silMax`) kullanır.

# Örnek İşlem Dizisi (min-öncelikli)

| İşlem | Dönen değer | Kuyruğun içeriği |
|---|---|---|
| `insert(5, A)` | | { (5,A) } |
| `insert(9, C)` | | { (5,A), (9,C) } |
| `insert(3, B)` | | { (3,B), (5,A), (9,C) } |
| `min()` | (3,B) | { (3,B), (5,A), (9,C) } |
| `removeMin()` | (3,B) | { (5,A), (9,C) } |
| `insert(7, D)` | | { (5,A), (7,D), (9,C) } |
| `removeMin()` | (5,A) | { (7,D), (9,C) } |
| `removeMin()` | (7,D) | { (9,C) } |
| `removeMin()` | (9,C) | { } |
| `removeMin()` | null | { } boş kuyruk |

LİSTE İLE GERÇEKLEME

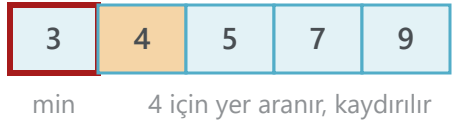
# Sırasız Liste ve Sıralı Liste

### Sırasız liste (unsorted)




- `insert` : sona ekle → **O(1)**
- `min` , `removeMin` : tüm listeyi tara → **O(n)**

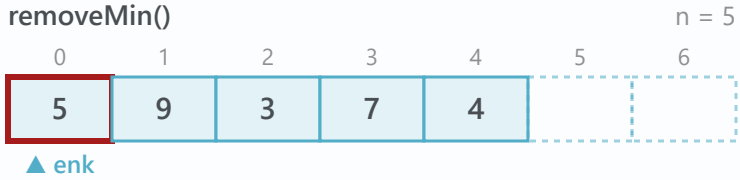
### Sıralı liste (sorted)




- `insert` : doğru yeri bul → **O(n)**
- `min` , `removeMin` : en küçük hep uçta → **O(1)**

Biri eklemede, diğeri çıkarmada hızlı. İkisini dengeleyen yapı: **ikili heap** (her ikisi de O(log n)).

# Kod İzleme: sırasız dizide `removeMin()`




Sırasız dizi a = [5, 9, 3, 7, 4]: ekleme sona yapıldığı için en küçüğün yeri bilinmez. İlk aday: **enk = 0**.

Adım 1 / 9

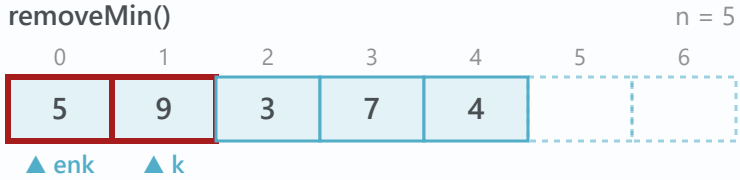
```
1 // sırasız dizi: insert(x) { a[n] = x; n++; }  O(1)
2 public int removeMin() {          // n > 0 olmalı
▶   int enk = 0;                    // en küçüğün indisi
4   for (int k = 1; k < n; k++)
5     if (a[k] < a[enk]) enk = k;
6   int min = a[enk];
7   a[enk] = a[n - 1];  n--;        // son öğe boşluğa
8   return min;
9 }
```

**DEĞİŞKENLER**

`n = 5` `enk = 0`

`a = [5, 9, 3, 7, 4]`

# Kod İzleme: sırasız dizide `removeMin()`




k = 1: a[1] = 9 ≥ a[0] = 5 → enk değişmez.

Adım 2 / 9

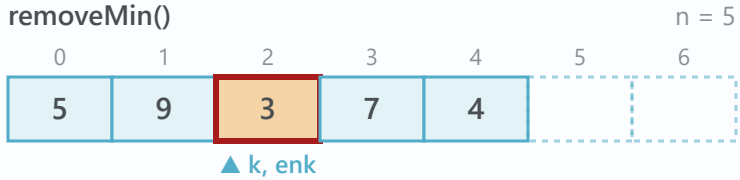
```
1 // sırasız dizi: insert(x) { a[n] = x; n++; }  O(1)
2 public int removeMin() {          // n > 0 olmalı
3   int enk = 0;                    // en küçüğün indisi
▶   for (int k = 1; k < n; k++)
▶     if (a[k] < a[enk]) enk = k;
6   int min = a[enk];
7   a[enk] = a[n - 1];  n--;        // son öğe boşluğa
8   return min;
9 }
```

**DEĞİŞKENLER**

`n = 5` `k = 1` `enk = 0`

`a = [5, 9, 3, 7, 4]`

# Kod İzleme: sırasız dizide `removeMin()`




k = 2: a[2] = 3 < a[0] = 5 → yeni en küçük, **enk = 2**.

Adım 3 / 9

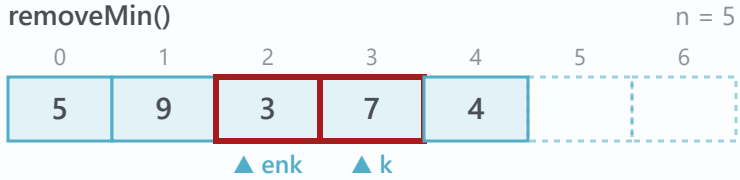
```
1 // sırasız dizi: insert(x) { a[n] = x; n++; }  O(1)
2 public int removeMin() {          // n > 0 olmalı
3   int enk = 0;                    // en küçüğün indisi
▶   for (int k = 1; k < n; k++)
▶     if (a[k] < a[enk]) enk = k;
6   int min = a[enk];
7   a[enk] = a[n - 1];  n--;        // son öğe boşluğa
8   return min;
9 }
```

**DEĞİŞKENLER**

`n = 5` `k = 2` `enk = 2`

`a = [5, 9, 3, 7, 4]`

# Kod İzleme: sırasız dizide `removeMin()`




k = 3: a[3] = 7 ≥ a[2] = 3 → enk değişmez.

Adım 4 / 9

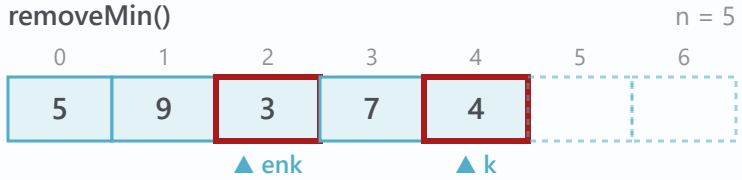
```
1 // sırasız dizi: insert(x) { a[n] = x; n++; }  O(1)
2 public int removeMin() {          // n > 0 olmalı
3   int enk = 0;                    // en küçüğün indisi
  for (int k = 1; k < n; k++)
    if (a[k] < a[enk]) enk = k;
6   int min = a[enk];
7   a[enk] = a[n - 1];  n--;        // son öğe boşluğa
8   return min;
9 }
```

**DEĞİŞKENLER**

`n = 5` `k = 3` `enk = 2`

`a = [5, 9, 3, 7, 4]`

# Kod İzleme: sırasız dizide `removeMin()`




k = 4: a[4] = 4 ≥ a[2] = 3 → enk değişmez.

Adım 5 / 9

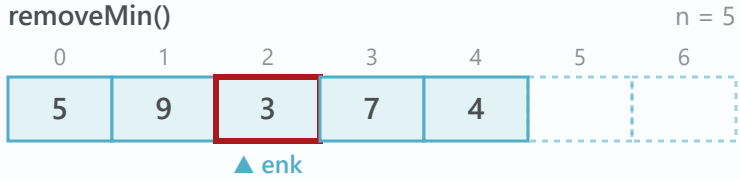
```
1 // sırasız dizi: insert(x) { a[n] = x; n++; }  O(1)
2 public int removeMin() {          // n > 0 olmalı
3   int enk = 0;                    // en küçüğün indisi
▶   for (int k = 1; k < n; k++)
▶     if (a[k] < a[enk]) enk = k;
6   int min = a[enk];
7   a[enk] = a[n - 1];  n--;        // son öğe boşluğa
8   return min;
9 }
```

**DEĞİŞKENLER**

`n = 5` `k = 4` `enk = 2`

`a = [5, 9, 3, 7, 4]`

# Kod İzleme: sırasız dizide `removeMin()`




k = 5 = n → tarama bitti: 4 karşılaştırma. Her removeMin tüm diziyi tarar → **O(n)**.

Adım 6 / 9

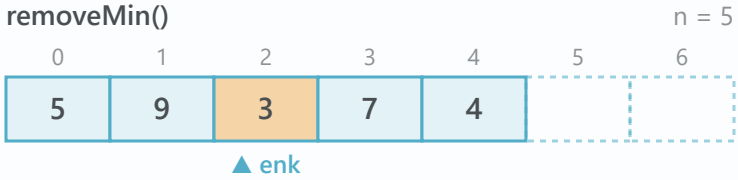
```
1 // sırasız dizi: insert(x) { a[n] = x; n++; }  O(1)
2 public int removeMin() {          // n > 0 olmalı
3   int enk = 0;                    // en küçüğün indisi
▶   for (int k = 1; k < n; k++)
5     if (a[k] < a[enk]) enk = k;
6   int min = a[enk];
7   a[enk] = a[n - 1];  n--;        // son öğe boşluğa
8   return min;
9 }
```

**DEĞİŞKENLER**

`n = 5` `k = 5` `enk = 2`

`a = [5, 9, 3, 7, 4]`

# Kod İzleme: sırasız dizide `removeMin()`




**min = a[2] = 3** saklanır.

Adım 7 / 9

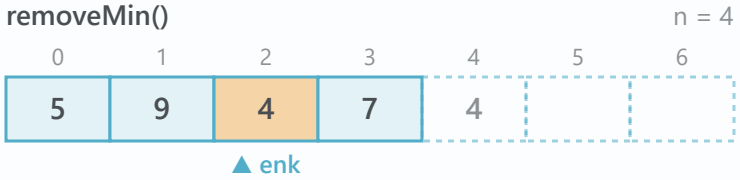
```
1 // sırasız dizi: insert(x) { a[n] = x; n++; }  O(1)
2 public int removeMin() {          // n > 0 olmalı
3   int enk = 0;                    // en küçüğün indisi
4   for (int k = 1; k < n; k++)
5     if (a[k] < a[enk]) enk = k;
▶   int min = a[enk];
7   a[enk] = a[n - 1];  n--;        // son öğe boşluğa
8   return min;
9 }
```

**DEĞİŞKENLER**

`n = 5` `enk = 2` `min = 3`

`a = [5, 9, 3, 7, 4]`

# Kod İzleme: sırasız dizide `removeMin()`




Boşluğu kaydırmadan kapatmak için son öğe (4) indis 2'ye taşınır, **n = 4**. Sıra önemli olmadığından bu O(1).

Adım 8 / 9

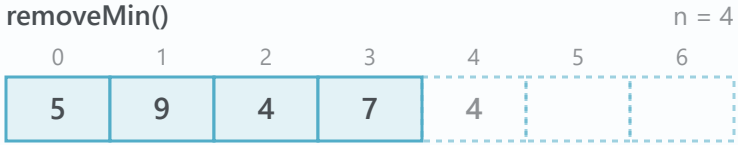
```
1 // sırasız dizi: insert(x) { a[n] = x; n++; }  O(1)
2 public int removeMin() {          // n > 0 olmalı
3   int enk = 0;                    // en küçüğün indisi
4   for (int k = 1; k < n; k++)
5     if (a[k] < a[enk]) enk = k;
6   int min = a[enk];
▶   a[enk] = a[n - 1];  n--;        // son öğe boşluğa
8   return min;
9 }
```

**DEĞİŞKENLER**

`n = 4` `enk = 2` `min = 3`

`a = [5, 9, 4, 7 | 4]`

# Kod İzleme: sırasız dizide `removeMin()`




**return 3**. Kalan dizi [5, 9, 4, 7] yine sırasız.

Adım 9 / 9

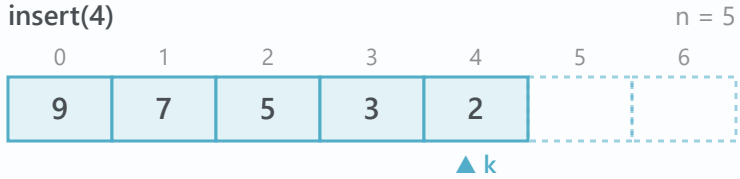
```
1 // sırasız dizi: insert(x) { a[n] = x; n++; }  O(1)
2 public int removeMin() {          // n > 0 olmalı
3   int enk = 0;                    // en küçüğün indisi
4   for (int k = 1; k < n; k++)
5     if (a[k] < a[enk]) enk = k;
6   int min = a[enk];
7   a[enk] = a[n - 1];  n--;        // son öğe boşluğa
▶   return min;
9 }
```

**DEĞİŞKENLER**

`n = 4` `min = 3`

`a = [5, 9, 4, 7 | 4]`

# Kod İzleme: sıralı dizide `insert(4)`




Dizi büyükten küçüğe sıralı; en öncelikli (en küçük) öğe hep sonda. **k = n − 1 = 4**: sondan başa doğru yer aranır.

Adım 1 / 10

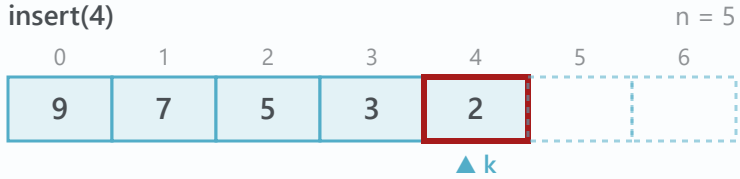
```
1  // sıralı dizi: büyükten küçüğe, min hep a[n-1]
2  public void insert(int x) {
▶    int k = n - 1;
4    while (k >= 0 && a[k] < x) {    // x'ten küçükler
5      a[k + 1] = a[k];              // bir sağa kaydır
6      k--;
7    }
8    a[k + 1] = x;  n++;
9  }
10 public int removeMin() { n--; return a[n]; }
```

**DEĞİŞKENLER**

`x = 4` `n = 5` `k = 4`

`a = [9, 7, 5, 3, 2]`

# Kod İzleme: sıralı dizide `insert(4)`




a[4] = 2 < 4 → 2, x'ten sonra gelmeli; kaydırılacak.

Adım 2 / 10

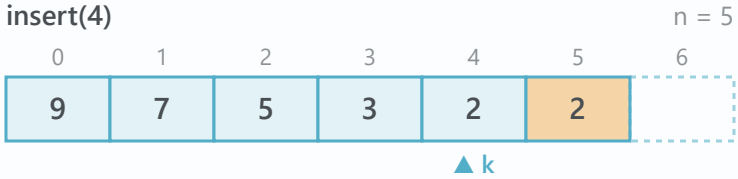
```
1  // sıralı dizi: büyükten küçüğe, min hep a[n-1]
2  public void insert(int x) {
3    int k = n - 1;
▶4   while (k >= 0 && a[k] < x) {     // x'ten küçükler
5      a[k + 1] = a[k];               // bir sağa kaydır
6      k--;
7    }
8    a[k + 1] = x;  n++;
9  }
10 public int removeMin() { n--; return a[n]; }
```

**DEĞİŞKENLER**

`x = 4` `n = 5` `k = 4`

`a = [9, 7, 5, 3, 2, None]`

# Kod İzleme: sıralı dizide `insert(4)`




**a[5] = a[4] = 2**: bir sağa kaydırıldı (a[4] şimdilik kopya; üzerine yazılacak).

Adım 3 / 10

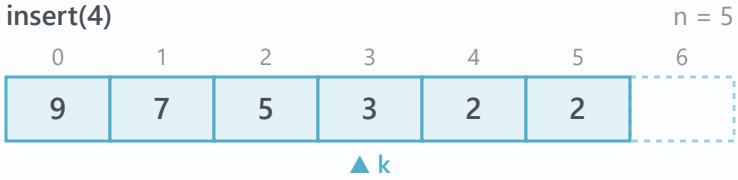
```
1  // sıralı dizi: büyükten küçüğe, min hep a[n-1]
2  public void insert(int x) {
3    int k = n - 1;
4    while (k >= 0 && a[k] < x) {     // x'ten küçükler
▶      a[k + 1] = a[k];               // bir sağa kaydır
6      k--;
7    }
8    a[k + 1] = x;  n++;
9  }
10 public int removeMin() { n--; return a[n]; }
```

**DEĞİŞKENLER**

`x = 4` `n = 5` `k = 4`

`a = [9, 7, 5, 3, 2, 2]`

# Kod İzleme: sıralı dizide `insert(4)`




**k = 3**: bir sola geç.

Adım 4 / 10

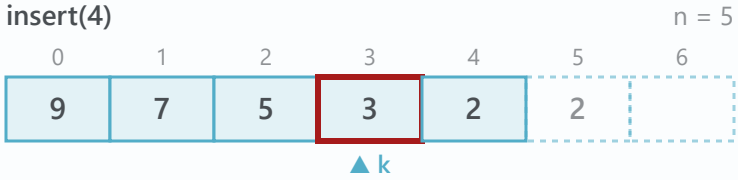
```
1  // sıralı dizi: büyükten küçüğe, min hep a[n-1]
2  public void insert(int x) {
3    int k = n - 1;
4    while (k >= 0 && a[k] < x) {    // x'ten küçükler
5      a[k + 1] = a[k];              // bir sağa kaydır
▶      k--;
7    }
8    a[k + 1] = x;  n++;
9  }
10 public int removeMin() { n--; return a[n]; }
```

**DEĞİŞKENLER**

`x = 4` `n = 5` `k = 3`

`a = [9, 7, 5, 3, 2, 2]`

# Kod İzleme: sıralı dizide `insert(4)`




a[3] = 3 < 4 → 3, x'ten sonra gelmeli; kaydırılacak.

Adım 5 / 10

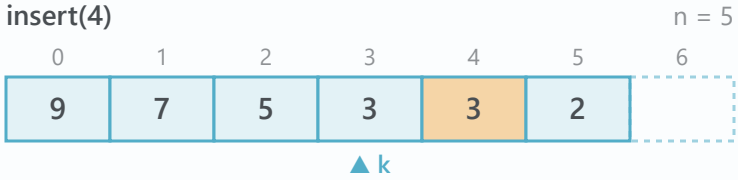
```
1  // sıralı dizi: büyükten küçüğe, min hep a[n-1]
2  public void insert(int x) {
3    int k = n - 1;
▶    while (k >= 0 && a[k] < x) {     // x'ten küçükler
5      a[k + 1] = a[k];               // bir sağa kaydır
6      k--;
7    }
8    a[k + 1] = x;  n++;
9  }
10 public int removeMin() { n--; return a[n]; }
```

**DEĞİŞKENLER**

`x = 4` `n = 5` `k = 3`

`a = [9, 7, 5, 3, 2]`

# Kod İzleme: sıralı dizide `insert(4)`




**a[4] = a[3] = 3**: bir sağa kaydırıldı (a[3] şimdilik kopya; üzerine yazılacak).

Adım 6 / 10

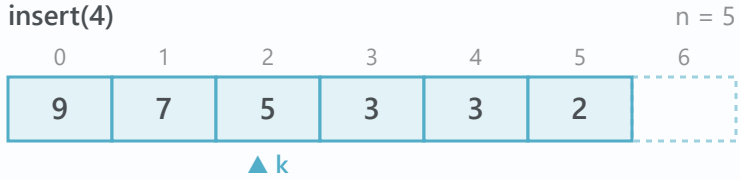
```
1  // sıralı dizi: büyükten küçüğe, min hep a[n-1]
2  public void insert(int x) {
3    int k = n - 1;
4    while (k >= 0 && a[k] < x) {     // x'ten küçükler
▶      a[k + 1] = a[k];               // bir sağa kaydır
6      k--;
7    }
8    a[k + 1] = x;  n++;
9  }
10 public int removeMin() { n--; return a[n]; }
```

**DEĞİŞKENLER**

`x = 4` `n = 5` `k = 3`

`a = [9, 7, 5, 3, 3, 2]`

# Kod İzleme: sıralı dizide `insert(4)`




**k = 2**: bir sola geç.

Adım 7 / 10

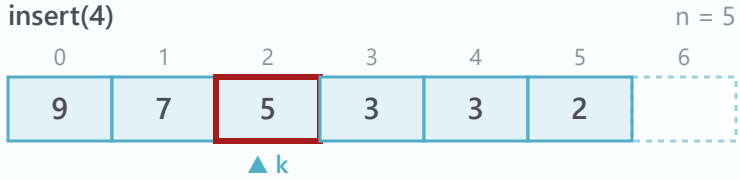
```
1  // sıralı dizi: büyükten küçüğe, min hep a[n-1]
2  public void insert(int x) {
3    int k = n - 1;
4    while (k >= 0 && a[k] < x) {    // x'ten küçükler
5      a[k + 1] = a[k];              // bir sağa kaydır
▶      k--;
7    }
8    a[k + 1] = x;  n++;
9  }
10 public int removeMin() { n--; return a[n]; }
```

**DEĞİŞKENLER**

`x = 4` `n = 5` `k = 2`

`a = [9, 7, 5, 3, 3, 2]`

# Kod İzleme: sıralı dizide `insert(4)`




a[2] = 5 ≥ 4 → döngü biter. 4'ün yeri **k + 1 = 3**.

Adım 8 / 10

```
1  // sıralı dizi: büyükten küçüğe, min hep a[n-1]
2  public void insert(int x) {
3    int k = n - 1;
▶    while (k >= 0 && a[k] < x) {     // x'ten küçükler
5      a[k + 1] = a[k];               // bir sağa kaydır
6      k--;
7    }
8    a[k + 1] = x;  n++;
9  }
10 public int removeMin() { n--; return a[n]; }
```

**DEĞİŞKENLER**

`x = 4` `n = 5` `k = 2`

`a = [9, 7, 5, 3, 3, 2]`

# Kod İzleme: sıralı dizide `insert(4)`




**a[3] = 4**, n = 6. Dizi yine sıralı. En kötü durumda (x en büyükse) tüm öğeler kayar → **O(n)**.

Adım 9 / 10

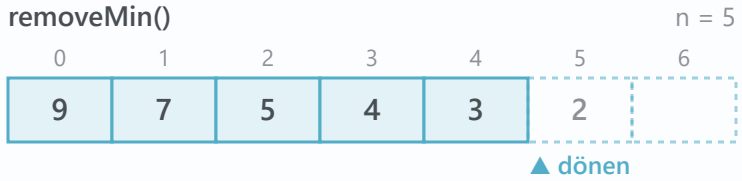
```
1  // sıralı dizi: büyükten küçüğe, min hep a[n-1]
2  public void insert(int x) {
3    int k = n - 1;
4    while (k >= 0 && a[k] < x) {     // x'ten küçükler
5      a[k + 1] = a[k];               // bir sağa kaydır
6      k--;
7    }
▶    a[k + 1] = x;  n++;
9  }
10 public int removeMin() { n--; return a[n]; }
```

**DEĞİŞKENLER**

`x = 4` `n = 6` `k = 2`

`a = [9, 7, 5, 4, 3, 2]`

# Kod İzleme: sıralı dizide `insert(4)`




Karşılığında **removeMin()** çok ucuz: en küçük (2) zaten sonda → n = 5, **return 2**. Kaydırma yok → **O(1)**.

Adım 10 / 10

```
1 // sıralı dizi: büyükten küçüğe, min hep a[n-1]
2 public void insert(int x) {
3   int k = n - 1;
4   while (k >= 0 && a[k] < x) {     // x'ten küçükler
5     a[k + 1] = a[k];               // bir sağa kaydır
6     k--;
7   }
8   a[k + 1] = x;  n++;
9 }
▶ public int removeMin() { n--; return a[n]; }
```

**DEĞİŞKENLER**

`n = 5` `a = [9, 7, 5, 4, 3 | 2]`

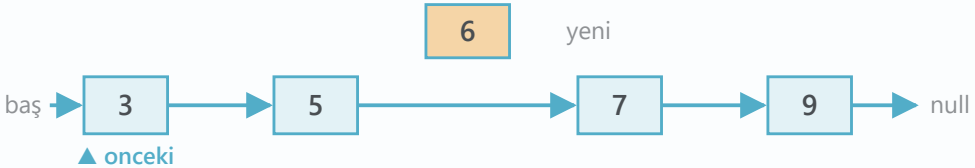
# Dizi ve Bağlı Liste Temsili

**Dizi temsili**

- Öğeler dizide tutulur; sıralı tutmak için ekleme sırasında öğeler kaydırılır (O(n)).
- Öncelikli öğe dizinin *başında* tutulursa her çıkarmada tüm dizi sola kaydırılır: O(n).
- Doğrusu: en öncelikli öğeyi dizinin **sonunda** tutmak → çıkarma O(1).

**Bağlı liste temsili**

- Öğeler önceliklerine göre listedeki uygun konuma eklenir: yer arama O(n), bağlama O(1).
- Öncelikli öğe listenin başındadır; çıkarma = baştan silme, O(1).
- Kaydırma yok, ama her düğüm ek bir referans taşır.




Liste büyükten küçüğe değil, **küçükten büyüğe** (min-öncelikli): en öncelikli 3 baştadır. Yeni düğüm 6 oluşturuldu.

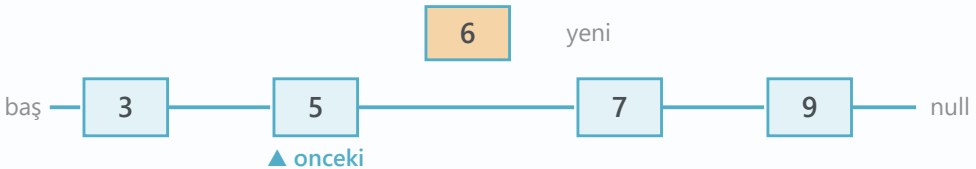
# Dizi ve Bağlı Liste Temsili

**Dizi temsili**

- Öğeler dizide tutulur; sıralı tutmak için ekleme sırasında öğeler kaydırılır (O(n)).
- Öncelikli öğe dizinin *başında* tutulursa her çıkarmada tüm dizi sola kaydırılır: O(n).
- Doğrusu: en öncelikli öğeyi dizinin **sonunda** tutmak → çıkarma O(1).

**Bağlı liste temsili**

- Öğeler önceliklerine göre listedeki uygun konuma eklenir: yer arama O(n), bağlama O(1).
- Öncelikli öğe listenin başındadır; çıkarma = baştan silme, O(1).
- Kaydırma yok, ama her düğüm ek bir referans taşır.




Yer arama: `onceki.sonraki.deger = 7 > 6` olana dek ilerle → **onceki = 5**. Bu tarama O(n) sürer.

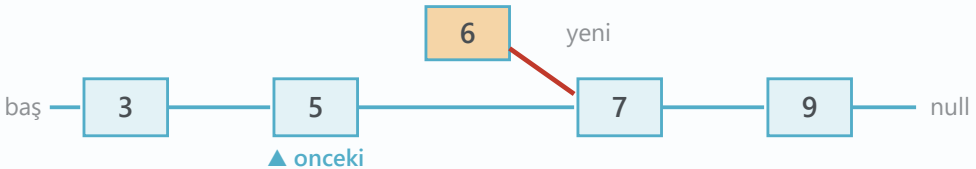
# Dizi ve Bağlı Liste Temsili

### Dizi temsili

- Öğeler dizide tutulur; sıralı tutmak için ekleme sırasında öğeler kaydırılır (O(n)).
- Öncelikli öğe dizinin *başında* tutulursa her çıkarmada tüm dizi sola kaydırılır: O(n).
- Doğrusu: en öncelikli öğeyi dizinin **sonunda** tutmak → çıkarma O(1).

### Bağlı liste temsili

- Öğeler önceliklerine göre listedeki uygun konuma eklenir: yer arama O(n), bağlama O(1).
- Öncelikli öğe listenin başındadır; çıkarma = baştan silme, O(1).
- Kaydırma yok, ama her düğüm ek bir referans taşır.




Önce `yeni.sonraki = onceki.sonraki` (6 → 7). Sıra önemli: önce bu bağ kurulmazsa 7'ye giden yol kaybolur.

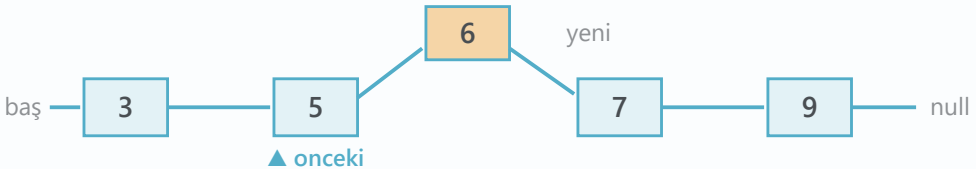
# Dizi ve Bağlı Liste Temsili

**Dizi temsili**

- Öğeler dizide tutulur; sıralı tutmak için ekleme sırasında öğeler kaydırılır (O(n)).
- Öncelikli öğe dizinin *başında* tutulursa her çıkarmada tüm dizi sola kaydırılır: O(n).
- Doğrusu: en öncelikli öğeyi dizinin **sonunda** tutmak → çıkarma O(1).

**Bağlı liste temsili**

- Öğeler önceliklerine göre listedeki uygun konuma eklenir: yer arama O(n), bağlama O(1).
- Öncelikli öğe listenin başındadır; çıkarma = baştan silme, O(1).
- Kaydırma yok, ama her düğüm ek bir referans taşır.




Sonra `onceki.sonraki = yeni` (5 → 6). Bağlama O(1), kaydırma yok; çıkarma her zaman baştan: O(1).

LİSTE İLE GERÇEKLEME

# Sıralı Dizi ile Öncelikli Kuyruk: Kod

```
class Oge<T> {
    T veri;
    int oncelik;
    Oge(T v, int p) { veri = v; oncelik = p; }
}
```

```
// OncelikliKuyruk<E> alanları
private Oge<E>[] dizi =
    (Oge<E>[]) new Oge[10];
private int boyut = 0;
public boolean bosMu() { return boyut == 0; }
```

```
public void ekle(E veri, int p) {
    if (boyut == dizi.length)
        dizi = Arrays.copyOf(dizi, 2 * boyut);
    int i = boyut;
    while (i > 0 && dizi[i-1].oncelik < p) {
        dizi[i] = dizi[i - 1];  // sağa kaydır
        i--;
    }
    dizi[i] = new Oge<>(veri, p);
    boyut++;
}
```

```
public E cikar() {
    if (bosMu()) return null;
    E sonuc = dizi[0].veri;
    for (int i = 1; i < boyut; i++)
        dizi[i - 1] = dizi[i];
    dizi[--boyut] = null;
    return sonuc;
}
```

Öğeler (veri, öncelik) çifti olarak saklanır; büyük öncelik önde, dizi[0]'da. Bu basit sürüm önceki slayttaki öneriye uymaz: ekle O(n) (kaydırma), cikar da O(n) (sola kaydırma). Öğeler artan sırada tutulsaydı cikar tek satıra ( `dizi[--boyut]` ) ve O(1)'e inerdi. Eşit öncelikte `<` karşılaştırması geliş sırasını korur.

# Kod İzleme: Eşit Öncelikte `ekle(D, 5)`




ekle(A, 2), ekle(B, 5), ekle(C, 1) sonrası dizi [B:5, A:2, C:1]. Şimdi `ekle(D, 5)` : iki öğe kayacak.

Adım 1 / 10

```
▶ public void ekle(E veri, int p) {
2   if (boyut == dizi.length)
3     dizi = Arrays.copyOf(dizi, 2 * boyut);
4   int i = boyut;
5   while (i > 0 && dizi[i-1].oncelik < p) {
6     dizi[i] = dizi[i - 1];  // sağa kaydır
7     i--;
8   }
9   dizi[i] = new Oge<>(veri, p);
10  boyut++;
11 }
```

**DEĞİŞKENLER**

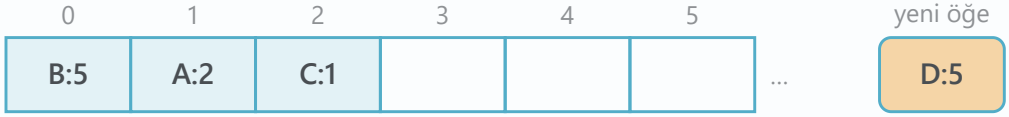
`veri = –` `p = –` `i = –`

`boyut = 3` `dizi = [B:5, A:2, C:1]`

**ÇAĞRI YIĞINI**

`main()`

# Kod İzleme: Eşit Öncelikte `ekle(D, 5)`




`ekle(D, 5)` : boyut 3 == 10? Hayır, büyütme gerekmez.

Adım 2 / 10

```
1  public void ekle(E veri, int p) {
▶    if (boyut == dizi.length)
3      dizi = Arrays.copyOf(dizi, 2 * boyut);
4    int i = boyut;
5    while (i > 0 && dizi[i-1].oncelik < p) {
6      dizi[i] = dizi[i - 1];  // sağa kaydır
7      i--;
8    }
9    dizi[i] = new Oge<>(veri, p);
10   boyut++;
11 }
```

**DEĞİŞKENLER**

`veri = D` `p = 5` `i = –`

`boyut = 3` `dizi = [B:5, A:2, C:1]`

**ÇAĞRI YIĞINI**

`ekle(D, 5)`

`main()`

# Kod İzleme: Eşit Öncelikte `ekle(D, 5)`




**i = boyut = 3**: ilk boş yuvadan başlayıp sola doğru yer arayacağız.

Adım 3 / 10

```
1  public void ekle(E veri, int p) {
2    if (boyut == dizi.length)
3      dizi = Arrays.copyOf(dizi, 2 * boyut);
4    int i = boyut;
5    while (i > 0 && dizi[i-1].oncelik < p) {
6      dizi[i] = dizi[i - 1];  // sağa kaydır
7      i--;
8    }
9    dizi[i] = new Oge<>(veri, p);
10   boyut++;
11 }
```

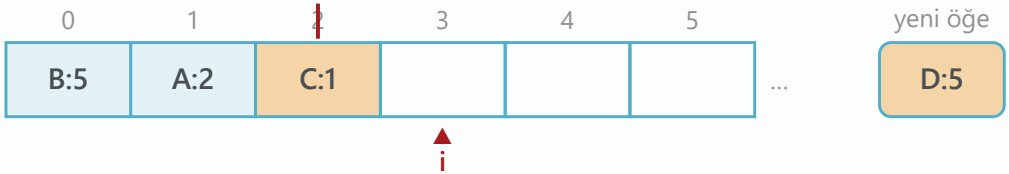
**DEĞİŞKENLER**

`veri = D` `p = 5` `i = 3`
`boyut = 3` `dizi = [B:5, A:2, C:1]`

**ÇAĞRI YIĞINI**

`ekle(D, 5)`
`main()`

# Kod İzleme: Eşit Öncelikte `ekle(D, 5)`




i > 0 ve dizi[2].oncelik = 1 < 5 → **true**: solundaki öğe daha düşük öncelikli.

Adım 4 / 10

```
1  public void ekle(E veri, int p) {
2    if (boyut == dizi.length)
3      dizi = Arrays.copyOf(dizi, 2 * boyut);
4    int i = boyut;
▶   while (i > 0 && dizi[i-1].oncelik < p) {
6      dizi[i] = dizi[i - 1];  // sağa kaydır
7      i--;
8    }
9    dizi[i] = new Oge<>(veri, p);
10   boyut++;
11 }
```

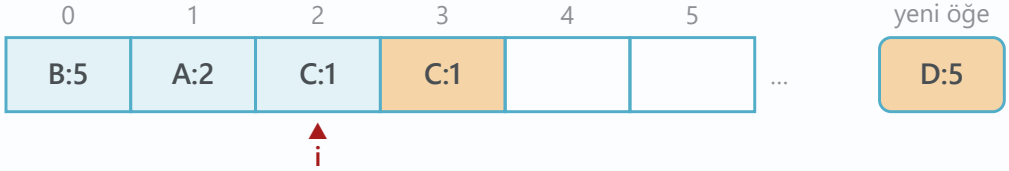
**DEĞİŞKENLER**

`veri = D` `p = 5` `i = 3`
`boyut = 3` `dizi = [B:5, A:2, C:1]`

**ÇAĞRI YIĞINI**

`ekle(D, 5)`
`main()`

# Kod İzleme: Eşit Öncelikte `ekle(D, 5)`




dizi[3] = dizi[2]: C bir sağa kayar; **i = 2**.

Adım 5 / 10

```
1  public void ekle(E veri, int p) {
2    if (boyut == dizi.length)
3      dizi = Arrays.copyOf(dizi, 2 * boyut);
4    int i = boyut;
5    while (i > 0 && dizi[i-1].oncelik < p) {
▶      dizi[i] = dizi[i - 1];  // sağa kaydır
▶      i--;
8    }
9    dizi[i] = new Oge<>(veri, p);
10   boyut++;
11 }
```

**DEĞİŞKENLER**

`veri = D` `p = 5` `i = 2`

`boyut = 3`

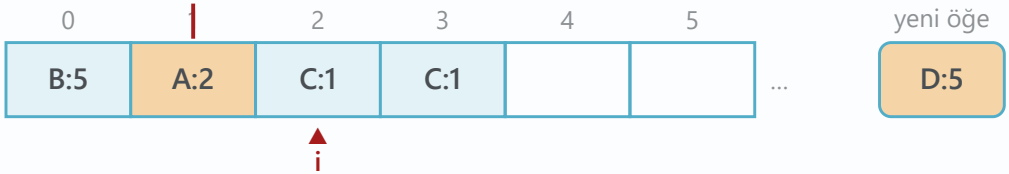
`dizi = [B:5, A:2, C:1, C:1]`

**ÇAĞRI YIĞINI**

`ekle(D, 5)`

`main()`

# Kod İzleme: Eşit Öncelikte `ekle(D, 5)`




i > 0 ve dizi[1].oncelik = 2 < 5 → **true**: solundaki öğe daha düşük öncelikli.

Adım 6 / 10

```
1  public void ekle(E veri, int p) {
2    if (boyut == dizi.length)
3      dizi = Arrays.copyOf(dizi, 2 * boyut);
4    int i = boyut;
▶   while (i > 0 && dizi[i-1].oncelik < p) {
6      dizi[i] = dizi[i - 1];  // sağa kaydır
7      i--;
8    }
9    dizi[i] = new Oge<>(veri, p);
10   boyut++;
11 }
```

**DEĞİŞKENLER**

`veri = D` `p = 5` `i = 2`

`boyut = 3`

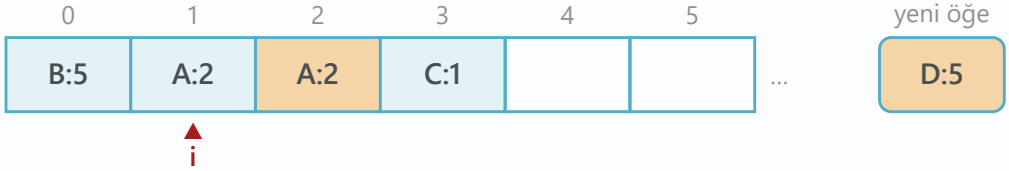
`dizi = [B:5, A:2, C:1, C:1]`

**ÇAĞRI YIĞINI**

`ekle(D, 5)`

`main()`

# Kod İzleme: Eşit Öncelikte `ekle(D, 5)`




dizi[2] = dizi[1]: A bir sağa kayar; **i = 1**.

Adım 7 / 10

```
1  public void ekle(E veri, int p) {
2    if (boyut == dizi.length)
3      dizi = Arrays.copyOf(dizi, 2 * boyut);
4    int i = boyut;
5    while (i > 0 && dizi[i-1].oncelik < p) {
▶      dizi[i] = dizi[i - 1];  // sağa kaydır
▶      i--;
8    }
9    dizi[i] = new Oge<>(veri, p);
10   boyut++;
11 }
```

**DEĞİŞKENLER**

`veri = D` `p = 5` `i = 1`

`boyut = 3`

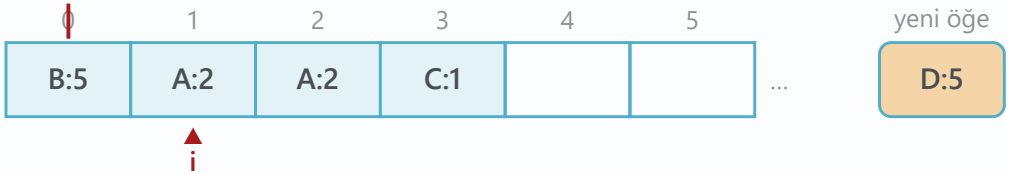
`dizi = [B:5, A:2, A:2, C:1]`

**ÇAĞRI YIĞINI**

`ekle(D, 5)`

`main()`

# Kod İzleme: Eşit Öncelikte `ekle(D, 5)`




dizi[0].oncelik = 5 < 5 değil (eşit öncelikte önce gelen önde kalır: FIFO) → **false**, döngü biter.

Adım 8 / 10

```
1  public void ekle(E veri, int p) {
2    if (boyut == dizi.length)
3      dizi = Arrays.copyOf(dizi, 2 * boyut);
4    int i = boyut;
▶   while (i > 0 && dizi[i-1].oncelik < p) {
6      dizi[i] = dizi[i - 1];  // sağa kaydır
7      i--;
8    }
9    dizi[i] = new Oge<>(veri, p);
10   boyut++;
11 }
```

**DEĞİŞKENLER**

`veri = D` `p = 5` `i = 1`

`boyut = 3`

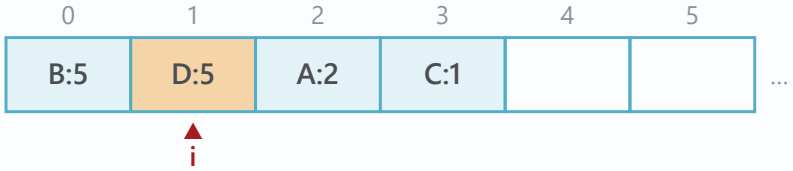
`dizi = [B:5, A:2, A:2, C:1]`

**ÇAĞRI YIĞINI**

`ekle(D, 5)`

`main()`

# Kod İzleme: Eşit Öncelikte `ekle(D, 5)`




`dizi[1] = (D, 5)` : açılan boşluğa yerleşir.

Adım 9 / 10

```
1  public void ekle(E veri, int p) {
2    if (boyut == dizi.length)
3      dizi = Arrays.copyOf(dizi, 2 * boyut);
4    int i = boyut;
5    while (i > 0 && dizi[i-1].oncelik < p) {
6      dizi[i] = dizi[i - 1];  // sağa kaydır
7      i--;
8    }
▶    dizi[i] = new Oge<>(veri, p);
10   boyut++;
11 }
```

**DEĞİŞKENLER**

`veri = D` `p = 5` `i = 1`

`boyut = 3`

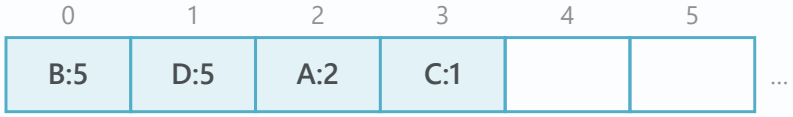
`dizi = [B:5, D:5, A:2, C:1]`

**ÇAĞRI YIĞINI**

`ekle(D, 5)`

`main()`

# Kod İzleme: Eşit Öncelikte `ekle(D, 5)`




`boyut++` → 4. Dizi yine büyükten küçüğe sıralı.

Adım 10 / 10

```
1  public void ekle(E veri, int p) {
2    if (boyut == dizi.length)
3      dizi = Arrays.copyOf(dizi, 2 * boyut);
4    int i = boyut;
5    while (i > 0 && dizi[i-1].oncelik < p) {
6      dizi[i] = dizi[i - 1];  // sağa kaydır
7      i--;
8    }
9    dizi[i] = new Oge<>(veri, p);
▶    boyut++;
11 }
```

**DEĞİŞKENLER**

`veri = D` `p = 5` `i = 1`

`boyut = 4`

`dizi = [B:5, D:5, A:2, C:1]`

**ÇAĞRI YIĞINI**

`ekle(D, 5)`

`main()`

# Maliyet Karşılaştırması

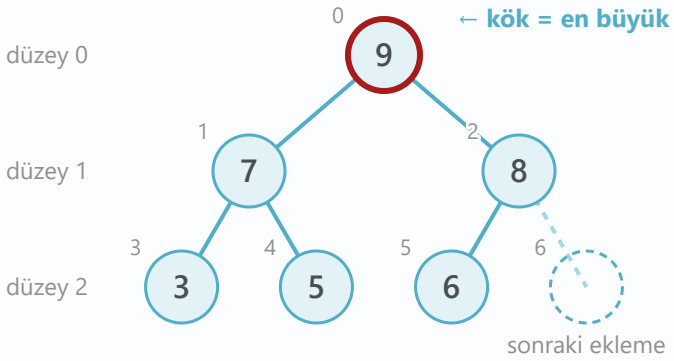
| İşlem | Sırasız liste | Sıralı liste | İkili heap |
|---|---|---|---|
| `size`, `isEmpty` | O(1) | O(1) | O(1) |
| `insert` | **O(1)** | O(n) | O(log n) |
| `min` | O(n) | **O(1)** | **O(1)** |
| `removeMin` | O(n) | **O(1)** | O(log n) |
| n öğeyle doldurma | O(n) | O(n log n)* | O(n)** |

* Önce sıralayarak. ** Aşağıdan yukarıya heap oluşturma ile.

n ekleme + n çıkarma: sırasız ve sıralı listede **$O(n^2)$**, heap'te **$O(n \log n)$**. $n = 10^6$ için fark ≈ 50.000 kat.

# İkili Heap (Binary Heap)

Öncelikli kuyruğu gerçeklemek için en yaygın veri yapısıdır. İki koşulu birlikte sağlayan bir ikili ağaçtır:




Kardeşler arasında sıra yok: 7 ile 8 yer değişse de heap olur.

### 1. Heap-sıra özelliği

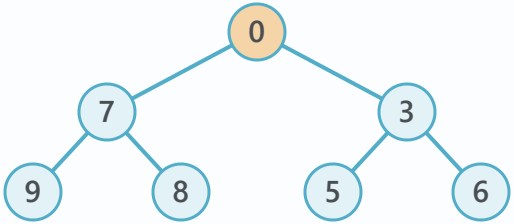
Her düğüm çocuklarından *önce gelir*: max-heap'te ebeveyn ≥ çocuk. En yüksek öncelikli öğe **kök**tedir.

### 2. Tam ikili ağaç özelliği

Son düzey hariç tüm düzeyler dolu, son düzey soldan sağa dolar → yükseklik $\lfloor \log_2 n \rfloor$, dizide boşluk yok.

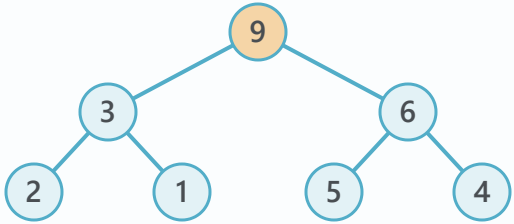
Ekleme ve çıkarma yalnızca kökten yaprağa uzanan *tek bir yol* üzerinde çalışır → **O(log n)**. Heap, ikili arama ağacı *değildir*.

# Min-Heap ve Max-Heap




**Min-Heap**: kökte en küçük anahtar

`anahtar(ebeveyn) ≤ anahtar(çocuk)`

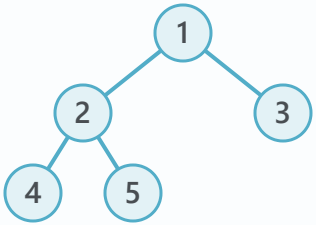



**Max-Heap**: kökte en büyük anahtar

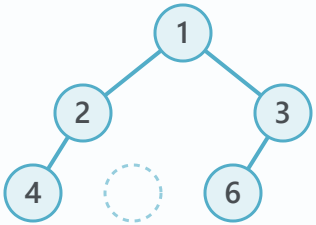
`anahtar(ebeveyn) ≥ anahtar(çocuk)`

- Sıra yalnızca **ebeveyn–çocuk** arasında geçerlidir: min-heap'te 7 ile 3 kardeştir, 7 > 3 olması sorun değildir.
- Kökten herhangi bir yaprağa giden her yol sıralıdır (min-heap'te artan, max-heap'te azalan).

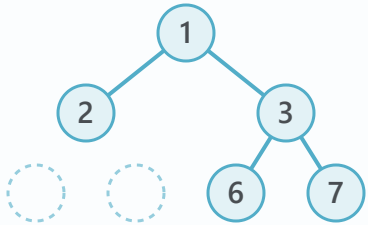
# Tam İkili Ağaç (Complete Binary Tree)




**Tam** ✓
son düzey soldan dolu




**Tam değil** ✗
son düzeyde boşluk var




**Tam değil** ✗
son düzey sola yaslı değil

### Yükseklik

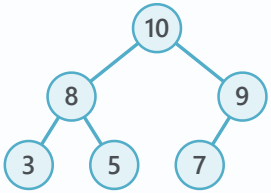
d. düzeyde en çok $2^d$ düğüm bulunur. n düğümlü tam ikili ağacın yüksekliği **h = ⌊log₂ n⌋**.

`n = 7 → h = 2 · n = 1000 → h = 9`

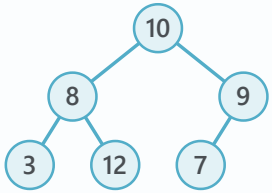
### Neden önemli?

Yeni öğe her zaman "son düzeyde en soldaki boş yere" gelir. Bu konum dizide tam olarak **indis n**'dir.

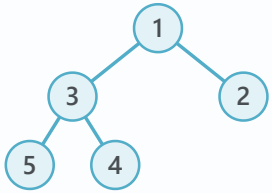
# Heap mi, Değil mi?



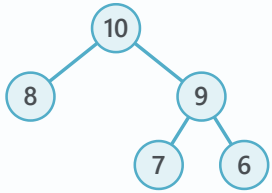

**(a)**




**(b)**




**(c)**

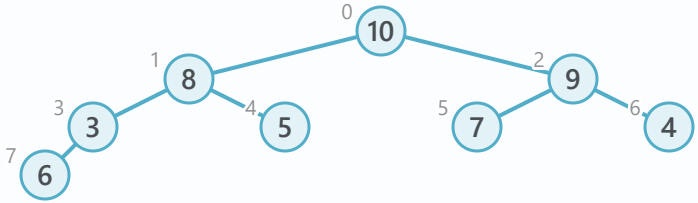



**(d)**

Her ağaç için karar verin: max-heap mi, min-heap mi, yoksa heap değil mi? Hangi özellik bozuluyor?

▶ **Cevapları göster**

# Kod İzleme: `heapMi([10, 8, 9, 3, 5, 7, 4, 6])`




heapMi(a) n = 8

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 10 | 8 | 9 | 3 | 5 | 7 | 4 | 6 |

heapMi(a) çağrıldı: 8 öğeli dizi. Kök (indis 0) hariç her düğüm, **ebeveyniyle** bir kez karşılaştırılacak.

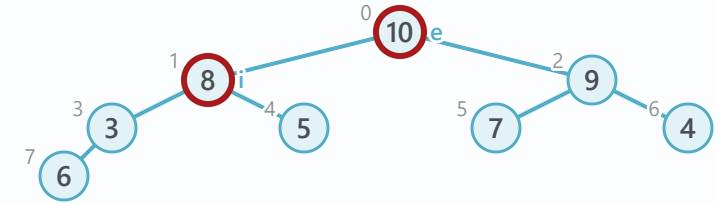
Adım 1 / 9

```
▶ static boolean heapMi(int[] a) {  // max-heap mi?
2   for (int i = 1; i < a.length; i++) {
3     int e = (i - 1) / 2;           // ebeveyn
4     if (a[e] < a[i]) return false; // sıra bozuk
5   }
6   return true;
7 }
```

**DEĞİŞKENLER**

`a = [10, 8, 9, 3, 5, 7, 4, 6]`

# Kod İzleme: `heapMi([10, 8, 9, 3, 5, 7, 4, 6])`




heapMi(a) n = 8

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 10 | 8 | 9 | 3 | 5 | 7 | 4 | 6 |
| ▲ e | ▲ i | | | | | | |

i = 1: ebeveyn e = (1 − 1) / 2 = 0. a[0] = 10 ≥ a[1] = 8 ✓ → devam.

Adım 2 / 9

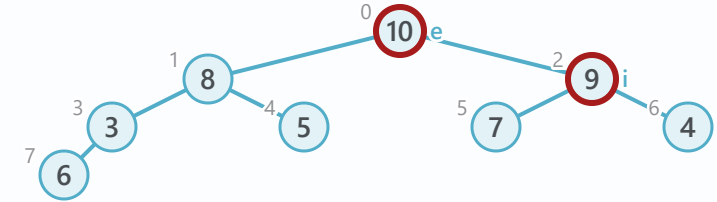
```
1 static boolean heapMi(int[] a) {  // max-heap mi?
▶   for (int i = 1; i < a.length; i++) {
▶     int e = (i - 1) / 2;           // ebeveyn
▶     if (a[e] < a[i]) return false; // sıra bozuk
5   }
6   return true;
7 }
```

**DEĞİŞKENLER**

`a = [10, 8, 9, 3, 5, 7, 4, 6]`

`i = 1` `e = 0`

# Kod İzleme: `heapMi([10, 8, 9, 3, 5, 7, 4, 6])`




heapMi(a) n = 8

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 10 | 8 | 9 | 3 | 5 | 7 | 4 | 6 |
| ▲ e | | ▲ i | | | | | |

i = 2: ebeveyn e = (2 − 1) / 2 = 0. a[0] = 10 ≥ a[2] = 9 ✓ → devam.

Adım 3 / 9

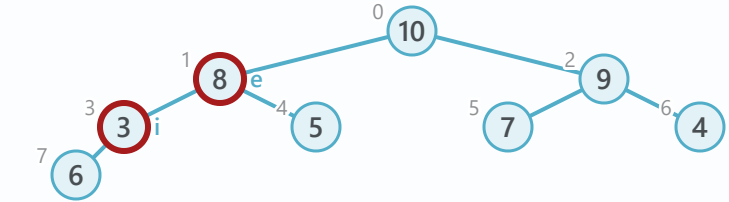
```
1 static boolean heapMi(int[] a) {  // max-heap mi?
▶   for (int i = 1; i < a.length; i++) {
▶     int e = (i - 1) / 2;           // ebeveyn
▶     if (a[e] < a[i]) return false; // sıra bozuk
5   }
6   return true;
7 }
```

**DEĞİŞKENLER**

`a = [10, 8, 9, 3, 5, 7, 4, 6]`

`i = 2` `e = 0`

# Kod İzleme: `heapMi([10, 8, 9, 3, 5, 7, 4, 6])`




heapMi(a) n = 8

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 10 | 8 | 9 | 3 | 5 | 7 | 4 | 6 |
| | ▲ e | | ▲ i | | | | |

i = 3: ebeveyn e = (3 − 1) / 2 = 1. a[1] = 8 ≥ a[3] = 3 ✓ → devam.

Adım 4 / 9

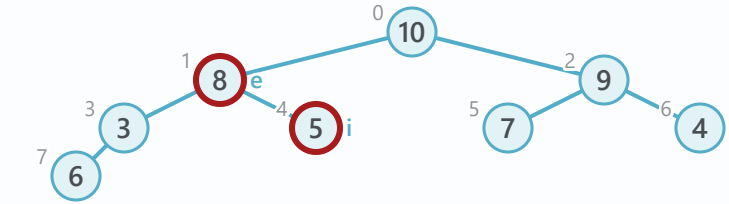
```
1 static boolean heapMi(int[] a) {  // max-heap mi?
▶   for (int i = 1; i < a.length; i++) {
▶     int e = (i - 1) / 2;           // ebeveyn
▶     if (a[e] < a[i]) return false; // sıra bozuk
5   }
6   return true;
7 }
```

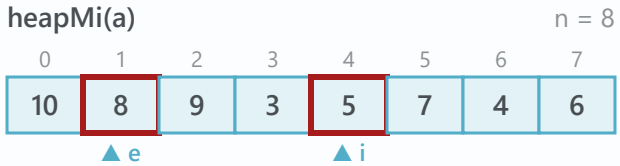
**DEĞİŞKENLER**

`a = [10, 8, 9, 3, 5, 7, 4, 6]`

`i = 3` `e = 1`

# Kod İzleme: `heapMi([10, 8, 9, 3, 5, 7, 4, 6])`




i = 4: ebeveyn e = (4 − 1) / 2 = 1. a[1] = 8 ≥ a[4] = 5 ✓ → devam.

Adım 5 / 9

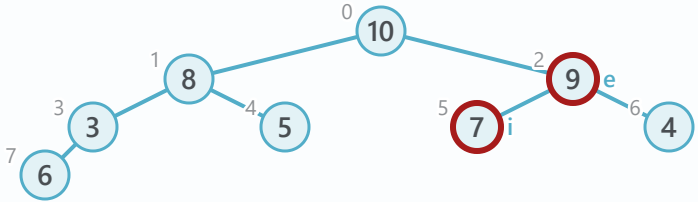
```
1 static boolean heapMi(int[] a) {  // max-heap mi?
▶   for (int i = 1; i < a.length; i++) {
▶     int e = (i - 1) / 2;           // ebeveyn
▶     if (a[e] < a[i]) return false; // sıra bozuk
5   }
6   return true;
7 }
```

**DEĞİŞKENLER**

`a = [10, 8, 9, 3, 5, 7, 4, 6]`

`i = 4` `e = 1`

# Kod İzleme: `heapMi([10, 8, 9, 3, 5, 7, 4, 6])`




heapMi(a) n = 8

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 10 | 8 | 9 | 3 | 5 | 7 | 4 | 6 |
| | | ▲ e | | | ▲ i | | |

i = 5: ebeveyn e = (5 − 1) / 2 = 2. a[2] = 9 ≥ a[5] = 7 ✓ → devam.

Adım 6 / 9

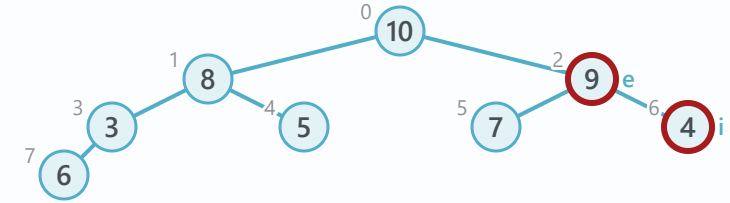
```
1 static boolean heapMi(int[] a) {  // max-heap mi?
▶   for (int i = 1; i < a.length; i++) {
▶     int e = (i - 1) / 2;           // ebeveyn
▶     if (a[e] < a[i]) return false; // sıra bozuk
5   }
6   return true;
7 }
```

**DEĞİŞKENLER**

`a = [10, 8, 9, 3, 5, 7, 4, 6]`

`i = 5` `e = 2`

# Kod İzleme: `heapMi([10, 8, 9, 3, 5, 7, 4, 6])`




heapMi(a) n = 8

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 10 | 8 | 9 | 3 | 5 | 7 | 4 | 6 |
| | | ▲ e | | | | ▲ i | |

i = 6: ebeveyn e = (6 − 1) / 2 = 2. a[2] = 9 ≥ a[6] = 4 ✓ → devam.

Adım 7 / 9

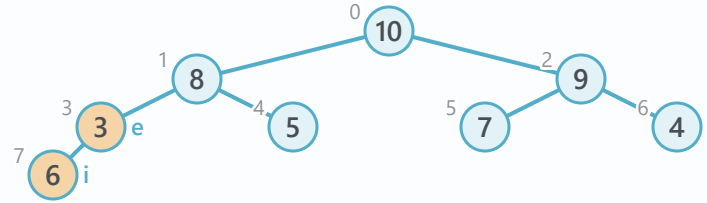
```
1 static boolean heapMi(int[] a) {  // max-heap mi?
▶   for (int i = 1; i < a.length; i++) {
▶     int e = (i - 1) / 2;           // ebeveyn
▶     if (a[e] < a[i]) return false; // sıra bozuk
5   }
6   return true;
7 }
```

**DEĞİŞKENLER**

`a = [10, 8, 9, 3, 5, 7, 4, 6]`

`i = 6` `e = 2`

# Kod İzleme: `heapMi([10, 8, 9, 3, 5, 7, 4, 6])`




heapMi(a) false

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 10 | 8 | 9 | 3 | 5 | 7 | 4 | 6 |
| | | | ▲ e | | | | ▲ i |

i = 7: e = 3. a[3] = 3 < a[7] = 6 ✗ → çocuk ebeveynden büyük: **return false**.

Adım 8 / 9

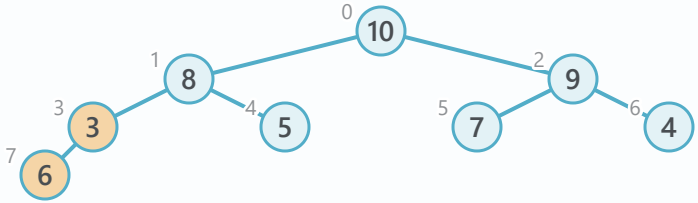
```
1 static boolean heapMi(int[] a) {  // max-heap mi?
2   for (int i = 1; i < a.length; i++) {
      int e = (i - 1) / 2;           // ebeveyn
      if (a[e] < a[i]) return false; // sıra bozuk
5   }
6   return true;
7 }
```

**DEĞİŞKENLER**

`a = [10, 8, 9, 3, 5, 7, 4, 6]`

`i = 7` `e = 3`

# Kod İzleme: `heapMi([10, 8, 9, 3, 5, 7, 4, 6])`




heapMi(a) false

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 10 | 8 | 9 | 3 | 5 | 7 | 4 | 6 |

Sonuç **false**: ilk yedi öğe geçerli bir heap olsa da 6, ebeveyni 3'ten büyük. En kötü durumda n − 1 karşılaştırma → **O(n)**.

Adım 9 / 9

```
1 static boolean heapMi(int[] a) {  // max-heap mi?
2   for (int i = 1; i < a.length; i++) {
3     int e = (i - 1) / 2;           // ebeveyn
▶     if (a[e] < a[i]) return false; // sıra bozuk
5   }
6   return true;
7 }
```

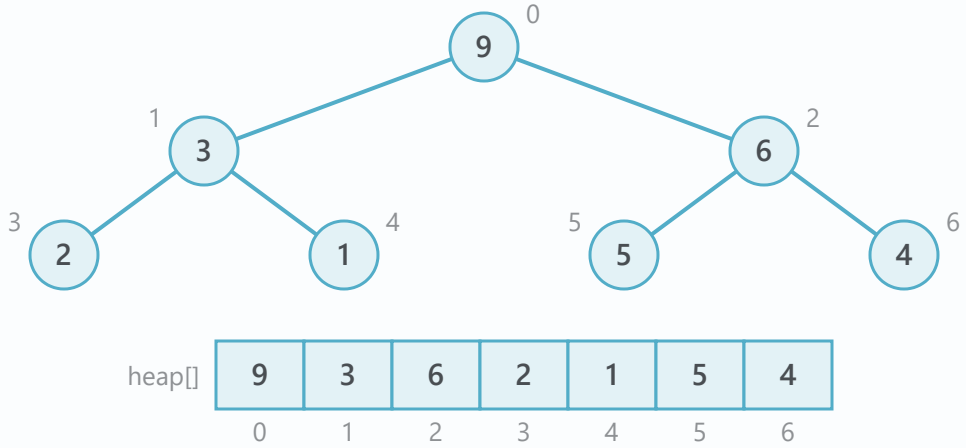
**DEĞİŞKENLER**

`a = [10, 8, 9, 3, 5, 7, 4, 6]`

`i = 7` `e = 3`

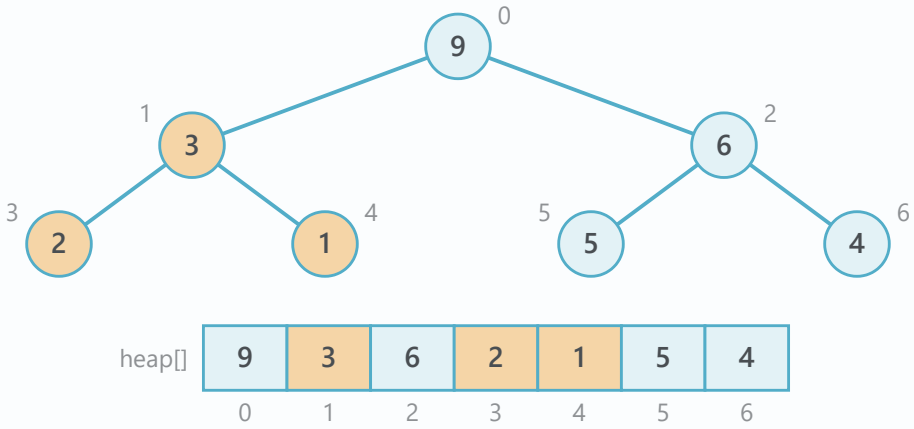
# Dizi Gösterimi

- Düğümler kökten başlayarak **düzey düzey, soldan sağa** (level order) diziye yazılır.
- Tam ağaç olduğu için dizide boşluk kalmaz; işaretçi (referans) gerekmez.




Düğümlerin yanındaki küçük sayılar dizideki indisleridir. Bu sunumda indisler **0'dan** başlar; kök heap[0]'dadır.

# Ebeveyn ve Çocuk Hesaplamaları




**Çocuklar**

`sol(i) = 2i + 1 · sağ(i) = 2i + 2`

`i = 1 → 3 ve 4 · i = 2 → 5 ve 6`

**Ebeveyn**

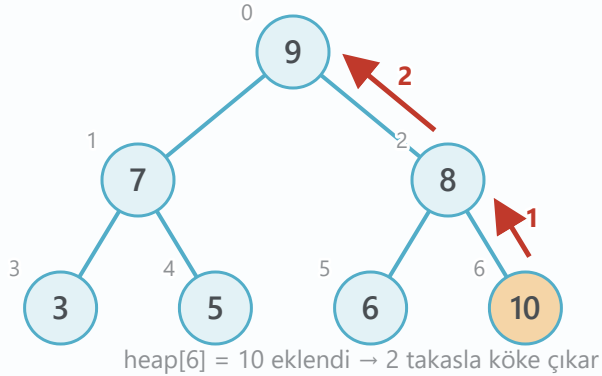
`ebeveyn(i) = ⌊(i − 1) / 2⌋`

`i = 6 → 2 · i = 5 → 2 · i = 4 → 1`

- Java'da tam sayı bölmesi zaten aşağı yuvarlar: `(i − 1) / 2`. Çocuk yoksa 2i + 1 ≥ n olur.
- 1 tabanlı alternatif: heap[0] boş bırakılır; çocuklar 2k, 2k + 1; ebeveyn ⌊k/2⌋. İkisi eşdeğerdir, biri seçilip tutarlı kullanılmalıdır.

# Öğe Ekleme: Up-Heap (yukarı süzme)

1. Yeni öğe ağacın **ilk boş konumuna** (heap[n]) eklenir. Ağaç tam kalır.
2. Heap-sıra özelliği bozulmuş olabilir: yeni öğe ebeveyninden büyükse (max-heap) yer değiştirilir.
3. Öğe köke ulaşana ya da ebeveyni kendisinden büyük/eşit olana kadar **yukarı** taşınır.



heap[6] = 10 eklendi → 2 takasla köke çıkar

**Maliyet**

En çok $h = \lfloor \log_2 n \rfloor$ yer değiştirme → **O(log n)**.

**Neden doğru?**

Yer değiştirmede aşağı inen ebeveyn, zaten diğer çocuğundan büyüktü.

**Adları**

up-heap, bubble-up, sift-up, swim; bu bölümün kodunda `yuzdur`.

# Max-Heap'e Ekleme: Java Kodu

```
public class MaxHeap {
  private int[] heap = new int[4];
  private int n = 0;                 // öğe sayısı

public void ekle(int x) {
  if (n == heap.length) buyut(2 * heap.length);
  heap[n] = x;  n++;                 // ilk boş konum
  yuzdur(n - 1);
}
private void yuzdur(int i) {         // up-heap
  while (i > 0) {
    int ebeveyn = (i - 1) / 2;
    if (heap[ebeveyn] >= heap[i]) return;
    yerDegistir(i, ebeveyn);  i = ebeveyn;
  }
}
private void yerDegistir(int a, int b) {
  int t = heap[a];  heap[a] = heap[b];  heap[b] = t;
}
}  // MaxHeap
```

**buyut**

Dizi dolunca kapasite iki katına çıkar (amortize O(1)).

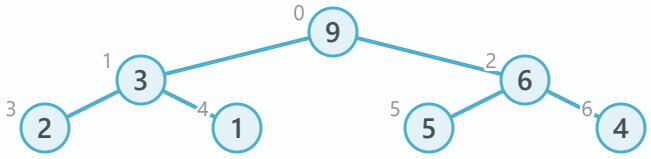
**yerDegistir**

heap[a] ↔ heap[b] takası.

**Durma**

Kökte (i = 0) ya da ebeveyn ≥ öğe olunca.

# Kod İzleme: `ekle(10)`, `ekle(8)`




başlangıç — n = 7 · heap.length = 16

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 9 | 3 | 6 | 2 | 1 | 5 | 4 | | |

Başlangıç max-heap: her ebeveyn ≥ çocukları, en büyük (9) kökte. Kapasite 16, yer var.

Adım 1 / 14

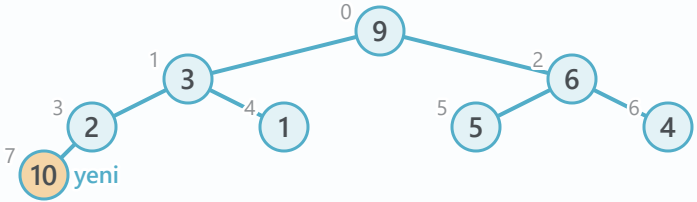
```
1  public void ekle(int x) {
▶    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;                // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {       // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

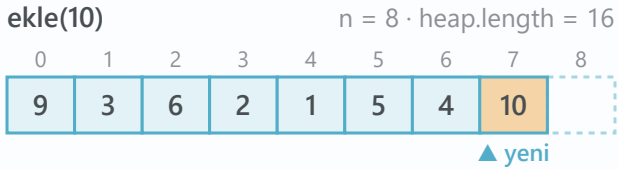
**DEĞİŞKENLER**

`n = 7` `heap.length = 16`

`heap = [9, 3, 6, 2, 1, 5, 4]`

# Kod İzleme: `ekle(10)`, `ekle(8)`




**ekle(10)**: 10, ilk boş konuma **heap[7]**'ye yazılır; n = 8. Ağaç tam kalır ama sıra bozulmuş olabilir.

Adım 2 / 14

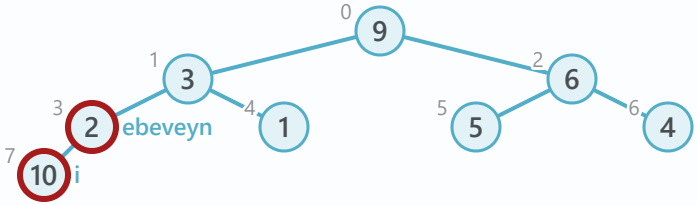
```
1  public void ekle(int x) {
▶    if (n == heap.length) buyut(2 * heap.length);
▶    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

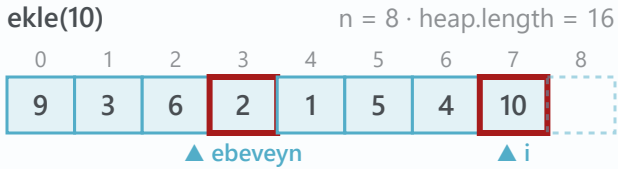
**DEĞİŞKENLER**

`x = 10` `n = 8` `heap.length = 16`

`heap = [9, 3, 6, 2, 1, 5, 4, 10]`

# Kod İzleme: `ekle(10)`, `ekle(8)`




i = 7 > 0 → **ebeveyn = (7 − 1) / 2 = 3**. Karşılaştır: heap[3] = 2 < heap[7] = 10.

Adım 3 / 14

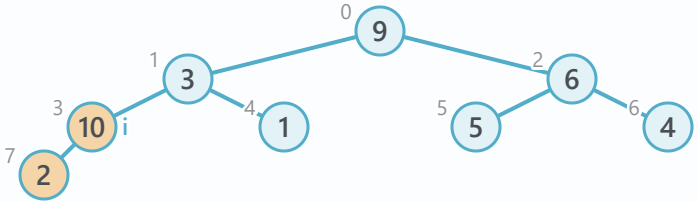
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
▶    while (i > 0) {
▶      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

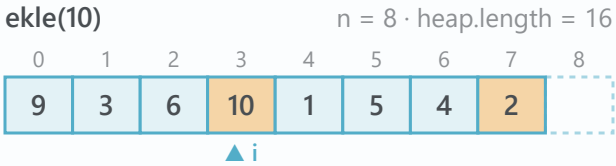
**DEĞİŞKENLER**

`i = 7` `ebeveyn = 3` `n = 8`

`heap = [9, 3, 6, 2, 1, 5, 4, 10]`

# Kod İzleme: `ekle(10)`, `ekle(8)`




10 > 2 → **yer değiştir** (indis 7 ↔ 3); öğe bir düzey yukarı çıktı, **i = 3**.

Adım 4 / 14

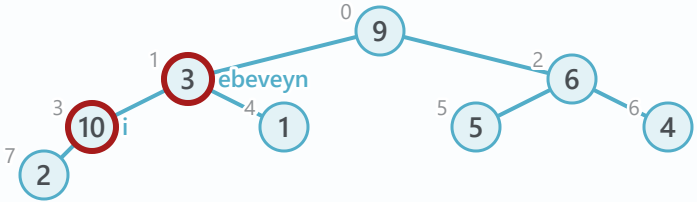
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
▶      yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

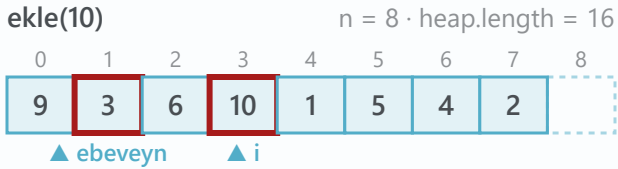
**DEĞİŞKENLER**

`i = 3` `ebeveyn = 3` `n = 8`

`heap = [9, 3, 6, 10, 1, 5, 4, 2]`

# Kod İzleme: `ekle(10)`, `ekle(8)`




i = 3 > 0 → **ebeveyn = (3 − 1) / 2 = 1**. Karşılaştır: heap[1] = 3 < heap[3] = 10.

Adım 5 / 14

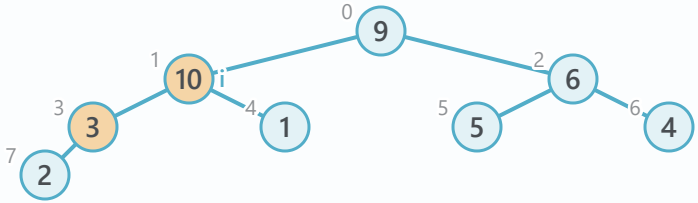
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;                 // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {         // up-heap
▶    while (i > 0) {
▶      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

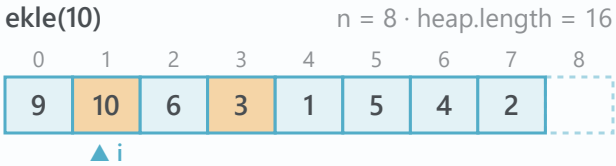
**DEĞİŞKENLER**

`i = 3` `ebeveyn = 1` `n = 8`

`heap = [9, 3, 6, 10, 1, 5, 4, 2]`

# Kod İzleme: `ekle(10)`, `ekle(8)`




10 > 3 → **yer değiştir** (indis 3 ↔ 1); öğe bir düzey yukarı çıktı, **i = 1**.

Adım 6 / 14

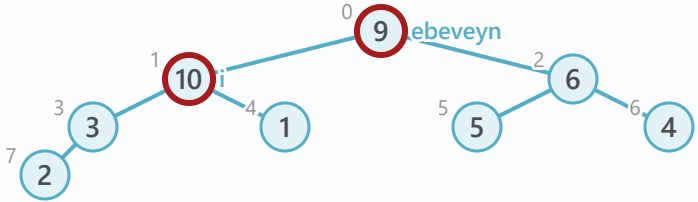
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
▶      yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

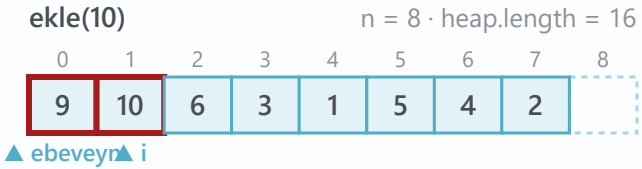
**DEĞİŞKENLER**

`i = 1` `ebeveyn = 1` `n = 8`

`heap = [9, 10, 6, 3, 1, 5, 4, 2]`

# Kod İzleme: `ekle(10)`, `ekle(8)`




i = 1 > 0 → **ebeveyn = (1 − 1) / 2 = 0**. Karşılaştır: heap[0] = 9 < heap[1] = 10.

Adım 7 / 14

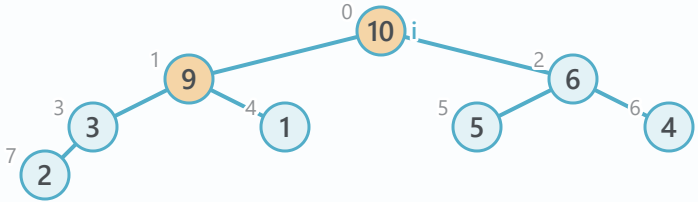
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
▶    while (i > 0) {
▶      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

**DEĞİŞKENLER**

`i = 1` `ebeveyn = 0` `n = 8`

`heap = [9, 10, 6, 3, 1, 5, 4, 2]`

# Kod İzleme: `ekle(10)`, `ekle(8)`




**ekle(10)** n = 8 · heap.length = 16

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 10 | 9 | 6 | 3 | 1 | 5 | 4 | 2 | |

▲ i

10 > 9 → **yer değiştir** (indis 1 ↔ 0); öğe bir düzey yukarı çıktı, **i = 0**.

Adım 8 / 14

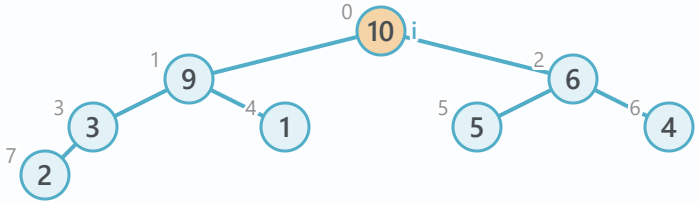
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
▶      yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

**DEĞİŞKENLER**

`i = 0` `ebeveyn = 0` `n = 8`

`heap = [10, 9, 6, 3, 1, 5, 4, 2]`

# Kod İzleme: `ekle(10)`, `ekle(8)`




ekle(10) n = 8 · heap.length = 16

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 10 | 9 | 6 | 3 | 1 | 5 | 4 | 2 | |

▲ i

i = 0: öğe **kökte**, while koşulu (i > 0) yanlış → yuzdur biter. 10 yerine oturdu.

Adım 9 / 14

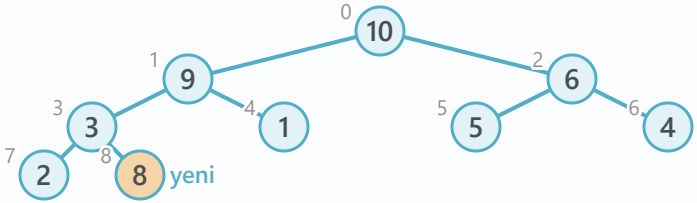
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
▶    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

**DEĞİŞKENLER**

`i = 0` `n = 8`

`heap = [10, 9, 6, 3, 1, 5, 4, 2]`

# Kod İzleme: `ekle(10)`, `ekle(8)`




ekle(8) n = 9 · heap.length = 16

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 10 | 9 | 6 | 3 | 1 | 5 | 4 | 2 | 8 |

▲ yeni

**ekle(8)**: 8, ilk boş konuma **heap[8]**'e yazılır; n = 9. Ağaç tam kalır ama sıra bozulmuş olabilir.

Adım 10 / 14

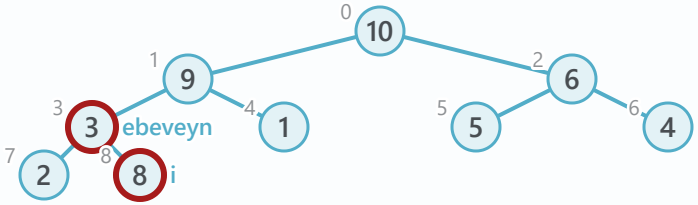
```
1  public void ekle(int x) {
▶    if (n == heap.length) buyut(2 * heap.length);
▶    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

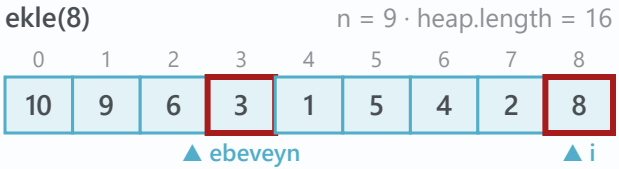
**DEĞİŞKENLER**

`x = 8` `n = 9` `heap.length = 16`

`heap = [10, 9, 6, 3, 1, 5, 4, 2, 8]`

# Kod İzleme: `ekle(10)`, `ekle(8)`




i = 8 > 0 → **ebeveyn = (8 − 1) / 2 = 3**. Karşılaştır: heap[3] = 3 < heap[8] = 8.

Adım 11 / 14

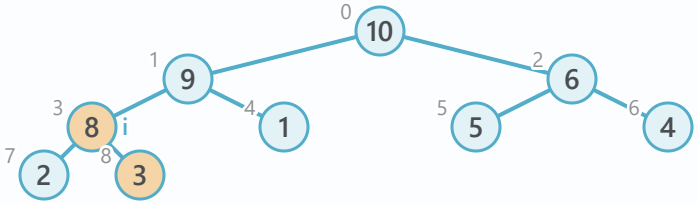
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
▶    while (i > 0) {
▶      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

**DEĞİŞKENLER**

`i = 8` `ebeveyn = 3` `n = 9`

`heap = [10, 9, 6, 3, 1, 5, 4, 2, 8]`

# Kod İzleme: `ekle(10)`, `ekle(8)`




ekle(8) n = 9 · heap.length = 16

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 10 | 9 | 6 | 8 | 1 | 5 | 4 | 2 | 3 |
| | | | ▲ i | | | | | |

8 > 3 → **yer değiştir** (indis 8 ↔ 3); öğe bir düzey yukarı çıktı, **i = 3**.

Adım 12 / 14

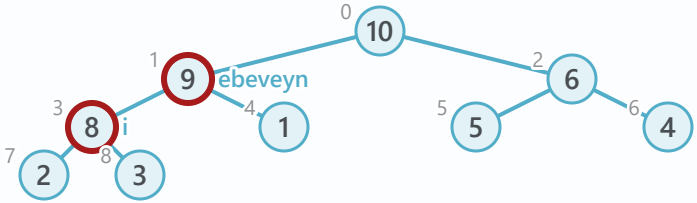
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
▶      yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

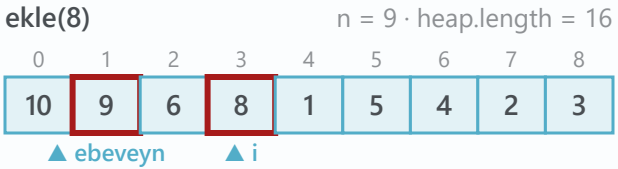
**DEĞİŞKENLER**

`i = 3` `ebeveyn = 3` `n = 9`

`heap = [10, 9, 6, 8, 1, 5, 4, 2, 3]`

# Kod İzleme: `ekle(10)`, `ekle(8)`




i = 3 > 0 → **ebeveyn = (3 − 1) / 2 = 1**. Karşılaştır: heap[1] = 9 ≥ heap[3] = 8.

Adım 13 / 14

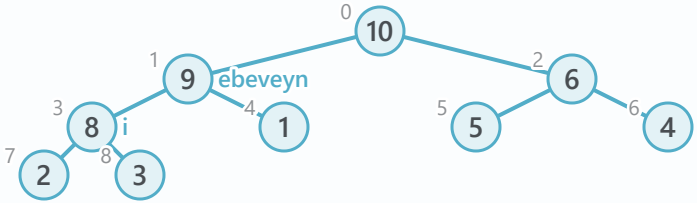
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
▶    while (i > 0) {
▶      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

**DEĞİŞKENLER**

`i = 3` `ebeveyn = 1` `n = 9`

`heap = [10, 9, 6, 8, 1, 5, 4, 2, 3]`

# Kod İzleme: `ekle(10)`, `ekle(8)`




ekle(8) — n = 9 · heap.length = 16

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 10 | 9 | 6 | 8 | 1 | 5 | 4 | 2 | 3 |
| | ▲ ebeveyn | | ▲ i | | | | | |

9 ≥ 8: ebeveyn zaten büyük/eşit → heap-sıra sağlandı, **return**. 8 yerine oturdu.

Adım 14 / 14

```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
▶      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

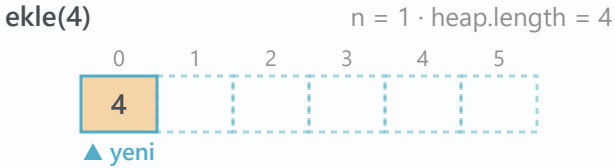
**DEĞİŞKENLER**

`i = 3` `ebeveyn = 1` `n = 9`

`heap = [10, 9, 6, 8, 1, 5, 4, 2, 3]`

# Kod İzleme: boş heap'e 4, 5, 2, 6 ekle




**ekle(4)**: 4, ilk boş konuma **heap[0]**'a yazılır; n = 1. Ağaç tam kalır ama sıra bozulmuş olabilir.

Adım 1 / 15

```
1  public void ekle(int x) {
▶    if (n == heap.length) buyut(2 * heap.length);
▶    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

**DEĞİŞKENLER**

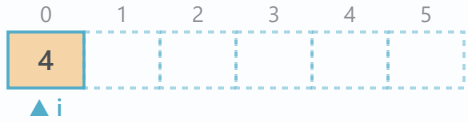
`x = 4` `n = 1` `heap.length = 4` `heap = [4]`

# Kod İzleme: boş heap'e 4, 5, 2, 6 ekle




ekle(4)

n = 1 · heap.length = 4




i = 0: öğe **kökte**, while koşulu (i > 0) yanlış → yuzdur biter. 4 yerine oturdu.

Adım 2 / 15

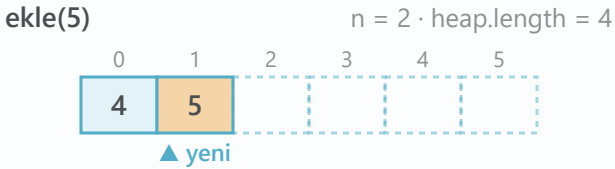
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
▶    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

DEĞİŞKENLER

i = 0 | n = 1 | heap = [4]

# Kod İzleme: boş heap'e 4, 5, 2, 6 ekle




**ekle(5)**: 5, ilk boş konuma **heap[1]**'e yazılır; n = 2. Ağaç tam kalır ama sıra bozulmuş olabilir.

Adım 3 / 15

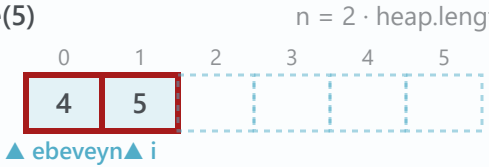
```
1  public void ekle(int x) {
▶    if (n == heap.length) buyut(2 * heap.length);
▶    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

**DEĞİŞKENLER**

`x = 5` `n = 2` `heap.length = 4` `heap = [4, 5]`

# Kod İzleme: boş heap'e 4, 5, 2, 6 ekle




i = 1 > 0 → **ebeveyn = (1 − 1) / 2 = 0**. Karşılaştır: heap[0] = 4 < heap[1] = 5.

Adım 4 / 15

```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
▶    while (i > 0) {
▶      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

**DEĞİŞKENLER**

`i = 1` `ebeveyn = 0` `n = 2`

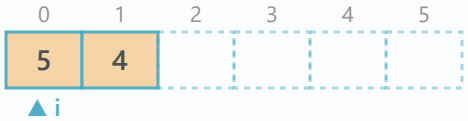
`heap = [4, 5]`

# Kod İzleme: boş heap'e 4, 5, 2, 6 ekle




ekle(5)

n = 2 · heap.length = 4




5 > 4 → **yer değiştir** (indis 1 ↔ 0); öğe bir düzey yukarı çıktı, **i = 0**.

Adım 5 / 15

```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
▶      yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

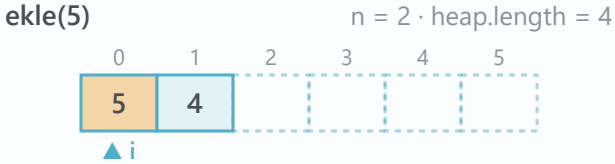
DEĞİŞKENLER

i = 0 | ebeveyn = 0 | n = 2

heap = [5, 4]

# Kod İzleme: boş heap'e 4, 5, 2, 6 ekle




i = 0: öğe **kökte**, while koşulu (i > 0) yanlış → yuzdur biter. 5 yerine oturdu.

Adım 6 / 15

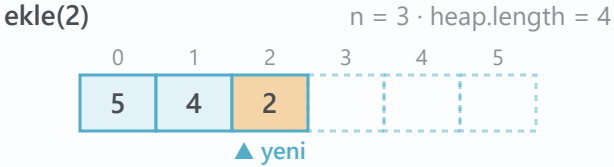
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
▶    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

**DEĞİŞKENLER**

`i = 0` `n = 2` `heap = [5, 4]`

# Kod İzleme: boş heap'e 4, 5, 2, 6 ekle




**ekle(2)**: 2, ilk boş konuma **heap[2]**'ye yazılır; n = 3. Ağaç tam kalır ama sıra bozulmuş olabilir.

Adım 7 / 15

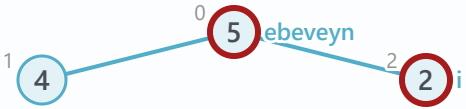
```
1  public void ekle(int x) {
▶    if (n == heap.length) buyut(2 * heap.length);
▶    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

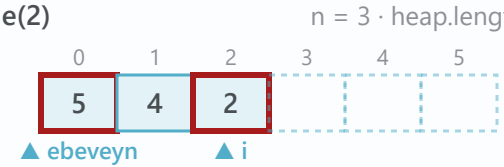
**DEĞİŞKENLER**

`x = 2` `n = 3` `heap.length = 4`

`heap = [5, 4, 2]`

# Kod İzleme: boş heap'e 4, 5, 2, 6 ekle




i = 2 > 0 → **ebeveyn = (2 − 1) / 2 = 0**. Karşılaştır: heap[0] = 5 ≥ heap[2] = 2.

Adım 8 / 15

```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
▶    while (i > 0) {
▶      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

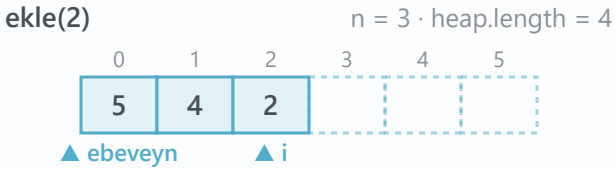
**DEĞİŞKENLER**

`i = 2` `ebeveyn = 0` `n = 3`

`heap = [5, 4, 2]`

# Kod İzleme: boş heap'e 4, 5, 2, 6 ekle




5 ≥ 2: ebeveyn zaten büyük/eşit → heap-sıra sağlandı, **return**. 2 yerine oturdu.

Adım 9 / 15

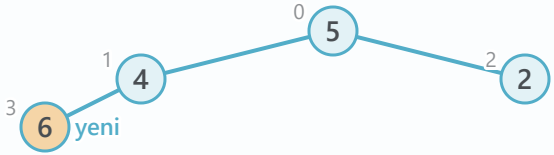
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
▶      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

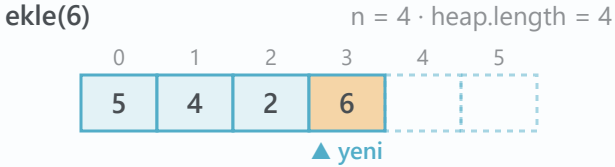
**DEĞİŞKENLER**

`i = 2` `ebeveyn = 0` `n = 3`

`heap = [5, 4, 2]`

# Kod İzleme: boş heap'e 4, 5, 2, 6 ekle




**ekle(6)**: 6, ilk boş konuma **heap[3]**'e yazılır; n = 4. Ağaç tam kalır ama sıra bozulmuş olabilir.

Adım 10 / 15

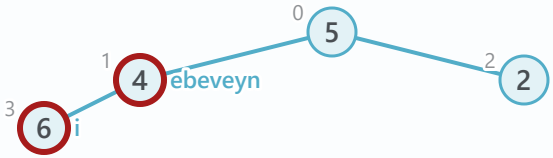
```
1  public void ekle(int x) {
▶    if (n == heap.length) buyut(2 * heap.length);
▶    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

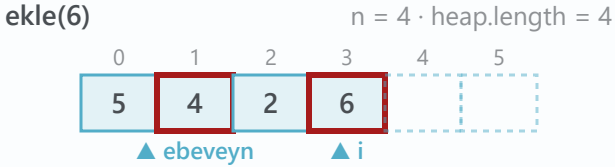
**DEĞİŞKENLER**

`x = 6` `n = 4` `heap.length = 4`

`heap = [5, 4, 2, 6]`

# Kod İzleme: boş heap'e 4, 5, 2, 6 ekle




i = 3 > 0 → **ebeveyn = (3 − 1) / 2 = 1**. Karşılaştır: heap[1] = 4 < heap[3] = 6.

Adım 11 / 15

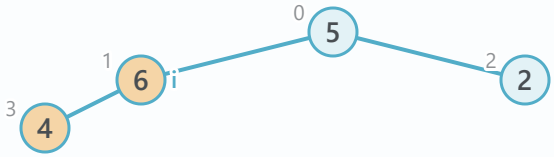
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
▶    while (i > 0) {
▶      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

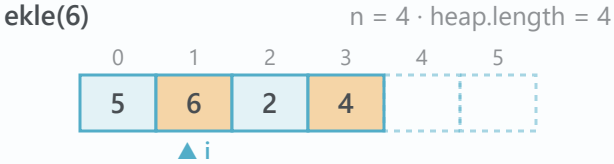
**DEĞİŞKENLER**

`i = 3` `ebeveyn = 1` `n = 4`

`heap = [5, 4, 2, 6]`

# Kod İzleme: boş heap'e 4, 5, 2, 6 ekle




6 > 4 → **yer değiştir** (indis 3 ↔ 1); öğe bir düzey yukarı çıktı, **i = 1**.

Adım 12 / 15

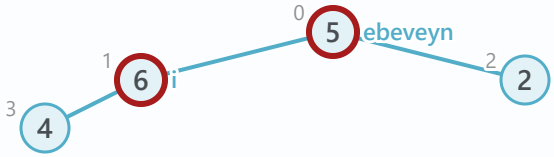
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
▶      yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

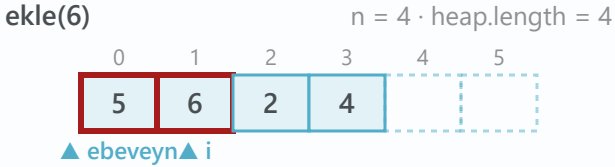
**DEĞİŞKENLER**

`i = 1` `ebeveyn = 1` `n = 4`

`heap = [5, 6, 2, 4]`

# Kod İzleme: boş heap'e 4, 5, 2, 6 ekle




i = 1 > 0 → **ebeveyn = (1 − 1) / 2 = 0**. Karşılaştır: heap[0] = 5 < heap[1] = 6.

Adım 13 / 15

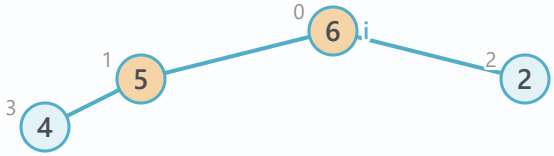
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
▶    while (i > 0) {
▶      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

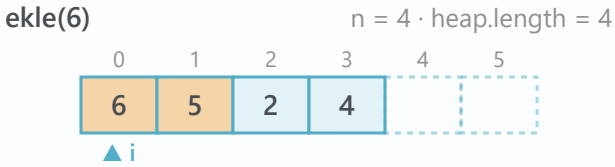
DEĞİŞKENLER

i = 1 | ebeveyn = 0 | n = 4

heap = [5, 6, 2, 4]

# Kod İzleme: boş heap'e 4, 5, 2, 6 ekle




6 > 5 → **yer değiştir** (indis 1 ↔ 0); öğe bir düzey yukarı çıktı, **i = 0**.

Adım 14 / 15

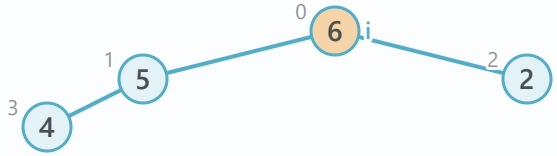
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
▶      yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

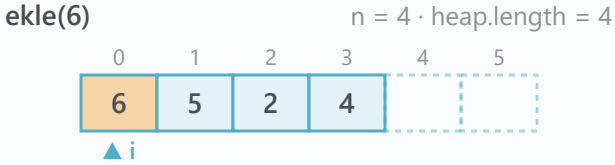
**DEĞİŞKENLER**

`i = 0` `ebeveyn = 0` `n = 4`

`heap = [6, 5, 2, 4]`

# Kod İzleme: boş heap'e 4, 5, 2, 6 ekle




i = 0: öğe **kökte**, while koşulu (i > 0) yanlış → yuzdur biter. 6 yerine oturdu.

Adım 15 / 15

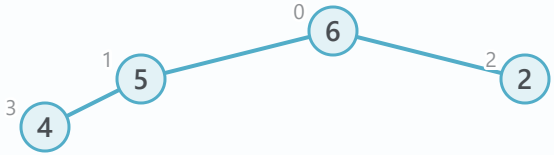
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
▶    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

**DEĞİŞKENLER**

`i = 0` `n = 4`

`heap = [6, 5, 2, 4]`

# Kod İzleme (devam): ekle(1), ekle(3)




**durum** n = 4 · heap.length = 4




Başlangıç: heap = [6, 5, 2, 4], n = 4 ve kapasite 4 → dizi **tamamen dolu**.

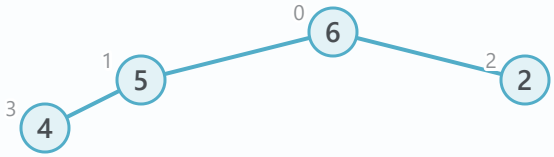
Adım 1 / 10

```
1  public void ekle(int x) {
▶    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

**DEĞİŞKENLER**

`n = 4` `heap.length = 4` `heap = [6, 5, 2, 4]`

# Kod İzleme (devam): ekle(1), ekle(3)




ekle(1)

n = 4 · heap.length = 8




**ekle(1)**: n == heap.length (4) → dizi dolu. **buyut(8)**: kapasite iki katına çıkar, öğeler yeni diziye kopyalanır.

Adım 2 / 10

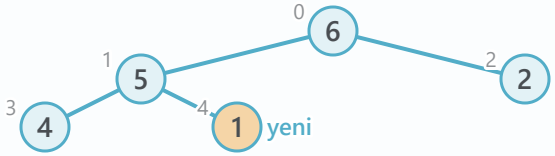
```
1  public void ekle(int x) {
▶    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

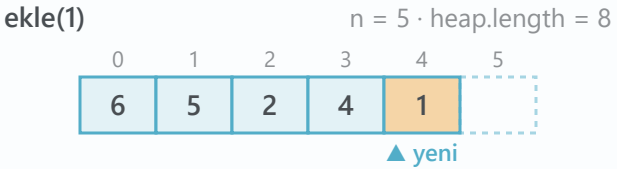
DEĞİŞKENLER

x = 1 | n = 4 | heap.length = 8

heap = [6, 5, 2, 4]

# Kod İzleme (devam): ekle(1), ekle(3)




1, ilk boş konuma **heap[4]**'e yazılır; n = 5. Ağaç tam kalır ama sıra bozulmuş olabilir.

Adım 3 / 10

```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
▶    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

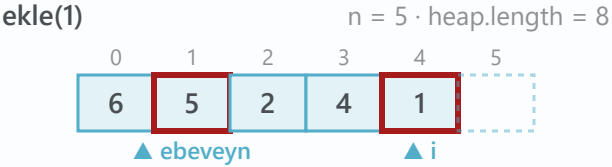
**DEĞİŞKENLER**

`x = 1` `n = 5` `heap.length = 8`

`heap = [6, 5, 2, 4, 1]`

# Kod İzleme (devam): ekle(1), ekle(3)




i = 4 > 0 → **ebeveyn = (4 − 1) / 2 = 1**. Karşılaştır: heap[1] = 5 ≥ heap[4] = 1.

Adım 4 / 10

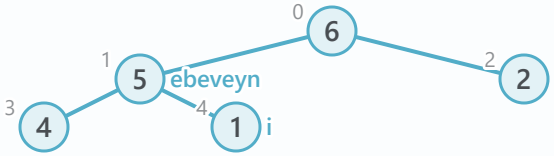
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
▶    while (i > 0) {
▶      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

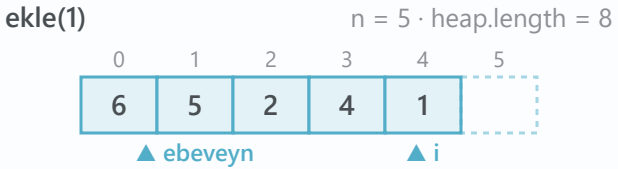
**DEĞİŞKENLER**

`i = 4` `ebeveyn = 1` `n = 5`

`heap = [6, 5, 2, 4, 1]`

# Kod İzleme (devam): ekle(1), ekle(3)




5 ≥ 1: ebeveyn zaten büyük/eşit → heap-sıra sağlandı, **return**. 1 yerine oturdu.

Adım 5 / 10

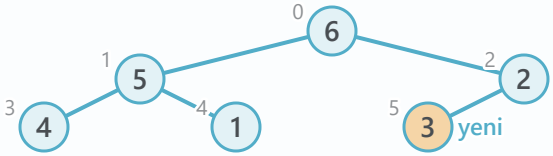
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
▶      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

**DEĞİŞKENLER**

`i = 4` `ebeveyn = 1` `n = 5`

`heap = [6, 5, 2, 4, 1]`

# Kod İzleme (devam): ekle(1), ekle(3)




ekle(3) n = 6 · heap.length = 8

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 6 | 5 | 2 | 4 | 1 | 3 |
| | | | | | ▲ yeni |

**ekle(3)**: 3, ilk boş konuma **heap[5]**'e yazılır; n = 6. Ağaç tam kalır ama sıra bozulmuş olabilir.

Adım 6 / 10

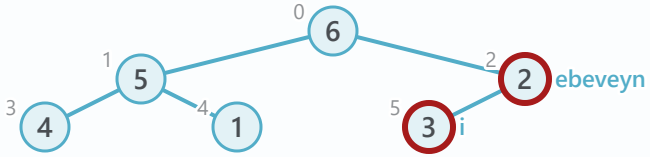
```
1  public void ekle(int x) {
▶    if (n == heap.length) buyut(2 * heap.length);
▶    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

DEĞİŞKENLER

x = 3 | n = 6 | heap.length = 8

heap = [6, 5, 2, 4, 1, 3]

# Kod İzleme (devam): ekle(1), ekle(3)




ekle(3) n = 6 · heap.length = 8

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 6 | 5 | 2 | 4 | 1 | 3 |
| | | ▲ ebeveyn | | | ▲ i |

i = 5 > 0 → **ebeveyn = (5 − 1) / 2 = 2**. Karşılaştır: heap[2] = 2 < heap[5] = 3.

Adım 7 / 10

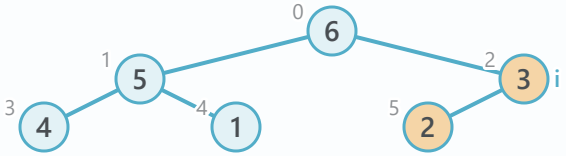
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
▶    while (i > 0) {
▶      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

**DEĞİŞKENLER**

`i = 5` `ebeveyn = 2` `n = 6`

`heap = [6, 5, 2, 4, 1, 3]`

# Kod İzleme (devam): ekle(1), ekle(3)




ekle(3) n = 6 · heap.length = 8

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 6 | 5 | 3 | 4 | 1 | 2 |
| | | ▲ i | | | |

3 > 2 → **yer değiştir** (indis 5 ↔ 2); öğe bir düzey yukarı çıktı, **i = 2**.

Adım 8 / 10

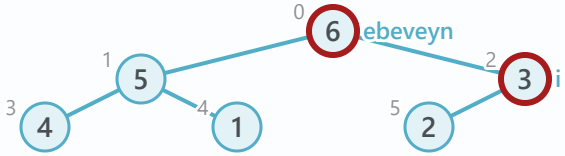
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
▶      yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

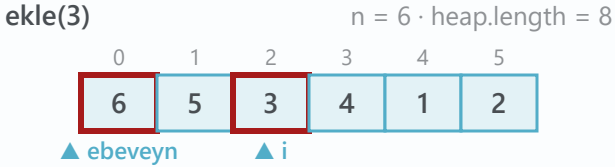
DEĞİŞKENLER

i = 2 | ebeveyn = 2 | n = 6

heap = [6, 5, 3, 4, 1, 2]

# Kod İzleme (devam): ekle(1), ekle(3)




i = 2 > 0 → **ebeveyn = (2 − 1) / 2 = 0**. Karşılaştır: heap[0] = 6 ≥ heap[2] = 3.

Adım 9 / 10

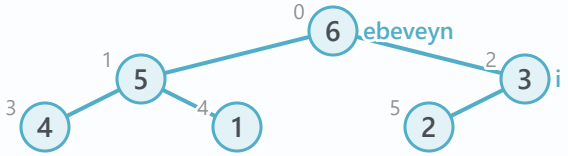
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
▶    while (i > 0) {
▶      int ebeveyn = (i - 1) / 2;
9      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

**DEĞİŞKENLER**

`i = 2` `ebeveyn = 0` `n = 6`

`heap = [6, 5, 3, 4, 1, 2]`

# Kod İzleme (devam): ekle(1), ekle(3)




ekle(3) n = 6 · heap.length = 8

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 6 | 5 | 3 | 4 | 1 | 2 |
| ▲ ebeveyn | | ▲ i | | | |

6 ≥ 3: ebeveyn zaten büyük/eşit → heap-sıra sağlandı, **return**. 3 yerine oturdu.

Adım 10 / 10

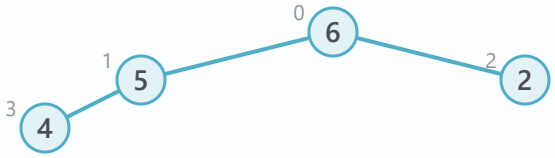
```
1  public void ekle(int x) {
2    if (n == heap.length) buyut(2 * heap.length);
3    heap[n] = x;  n++;              // ilk boş konum
4    yuzdur(n - 1);
5  }
6  private void yuzdur(int i) {      // up-heap
7    while (i > 0) {
8      int ebeveyn = (i - 1) / 2;
▶      if (heap[ebeveyn] >= heap[i]) return;
10     yerDegistir(i, ebeveyn);  i = ebeveyn;
11   }
12 }
```

**DEĞİŞKENLER**

`i = 2` `ebeveyn = 0` `n = 6`

`heap = [6, 5, 3, 4, 1, 2]`

# Kod İzleme: `buyut(8)`




buyut(8)




**ekle(1)** çağrıldı; n == heap.length = 4 → **buyut(8)**. Dizinin boyu sabittir; büyütmek = yeni dizi + kopyalama.

Adım 1 / 9

```
▶ private void buyut(int kapasite) {
2   int[] yeni = new int[kapasite];
3   for (int k = 0; k < n; k++)
4     yeni[k] = heap[k];          // kopyala
5   heap = yeni;                  // eski dizi çöpe
6 }
```

**DEĞİŞKENLER**

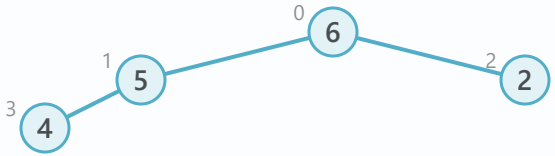
`kapasite = 8` `n = 4`

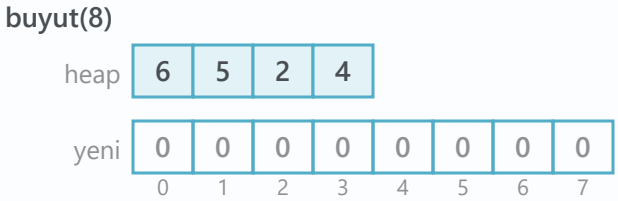
`heap = [6, 5, 2, 4]`

**ÇAĞRI YIĞINI**

- **buyut(8)**
- ekle(1)

# Kod İzleme: `buyut(8)`




8 hücrelik **yeni** dizi ayrılır; Java int dizisini 0 ile doldurur.

Adım 2 / 9

```
1 private void buyut(int kapasite) {
▶   int[] yeni = new int[kapasite];
3   for (int k = 0; k < n; k++)
4     yeni[k] = heap[k];          // kopyala
5   heap = yeni;                  // eski dizi çöpe
6 }
```

**DEĞİŞKENLER**

`kapasite = 8` `n = 4`

`heap = [6, 5, 2, 4]`

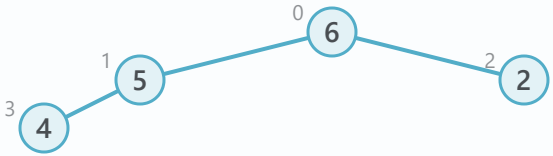
`yeni = [0, 0, 0, 0, 0, 0, 0, 0]`

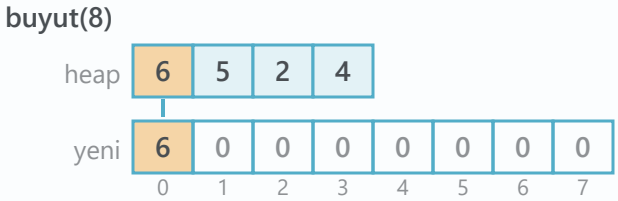
**ÇAĞRI YIĞINI**

`buyut(8)`

`ekle(1)`

# Kod İzleme: `buyut(8)`




k = 0 < n → **yeni[0] = heap[0] = 6**.

Adım 3 / 9

```
1 private void buyut(int kapasite) {
2   int[] yeni = new int[kapasite];
▶   for (int k = 0; k < n; k++)
▶     yeni[k] = heap[k];          // kopyala
5   heap = yeni;                  // eski dizi çöpe
6 }
```

**DEĞİŞKENLER**

`kapasite = 8` `n = 4` `k = 0`

`heap = [6, 5, 2, 4]`

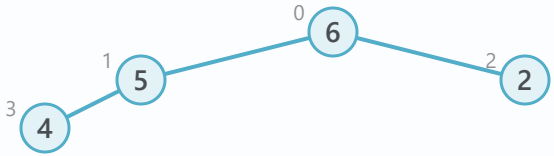
`yeni = [6, 0, 0, 0, 0, 0, 0, 0]`

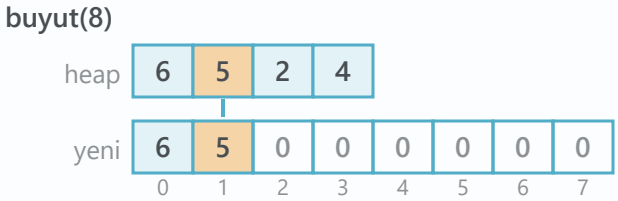
**ÇAĞRI YIĞINI**

`buyut(8)`

`ekle(1)`

# Kod İzleme: `buyut(8)`




k = 1 < n → **yeni[1] = heap[1] = 5**.

Adım 4 / 9

```
1 private void buyut(int kapasite) {
2   int[] yeni = new int[kapasite];
▶   for (int k = 0; k < n; k++)
▶     yeni[k] = heap[k];          // kopyala
5   heap = yeni;                  // eski dizi çöpe
6 }
```

**DEĞİŞKENLER**

`kapasite = 8` `n = 4` `k = 1`

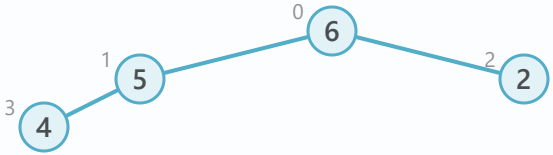
`heap = [6, 5, 2, 4]`

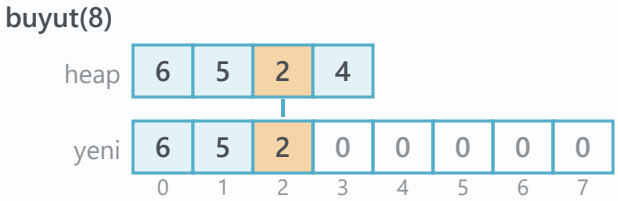
`yeni = [6, 5, 0, 0, 0, 0, 0, 0]`

**ÇAĞRI YIĞINI**

- `buyut(8)`
- `ekle(1)`

# Kod İzleme: `buyut(8)`




k = 2 < n → **yeni[2] = heap[2] = 2**.

Adım 5 / 9

```
1 private void buyut(int kapasite) {
2   int[] yeni = new int[kapasite];
▶   for (int k = 0; k < n; k++)
▶     yeni[k] = heap[k];          // kopyala
5   heap = yeni;                  // eski dizi çöpe
6 }
```

**DEĞİŞKENLER**

`kapasite = 8` `n = 4` `k = 2`

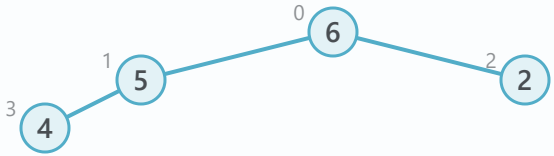
`heap = [6, 5, 2, 4]`

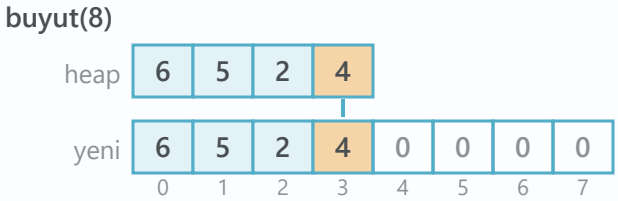
`yeni = [6, 5, 2, 0, 0, 0, 0, 0]`

**ÇAĞRI YIĞINI**

- **`buyut(8)`**
- `ekle(1)`

# Kod İzleme: `buyut(8)`




k = 3 < n → **yeni[3] = heap[3] = 4**.

Adım 6 / 9

```
1 private void buyut(int kapasite) {
2   int[] yeni = new int[kapasite];
▶   for (int k = 0; k < n; k++)
▶     yeni[k] = heap[k];          // kopyala
5   heap = yeni;                  // eski dizi çöpe
6 }
```

**DEĞİŞKENLER**

`kapasite = 8` `n = 4` `k = 3`

`heap = [6, 5, 2, 4]`

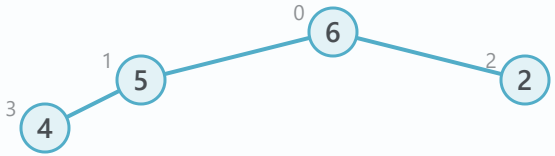
`yeni = [6, 5, 2, 4, 0, 0, 0, 0]`

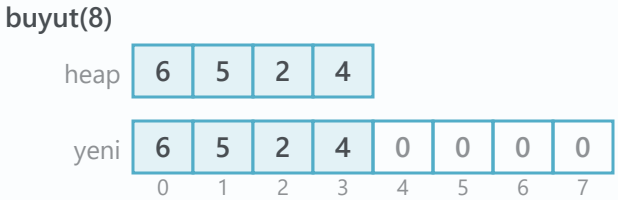
**ÇAĞRI YIĞINI**

`buyut(8)`

`ekle(1)`

# Kod İzleme: `buyut(8)`




k = 4 = n → döngü biter. n öğe kopyalandı: bu büyütmenin maliyeti **O(n)**.

Adım 7 / 9

```
1 private void buyut(int kapasite) {
2   int[] yeni = new int[kapasite];
▶   for (int k = 0; k < n; k++)
4     yeni[k] = heap[k];          // kopyala
5   heap = yeni;                  // eski dizi çöpe
6 }
```

**DEĞİŞKENLER**

`kapasite = 8` `n = 4` `k = 4`

`heap = [6, 5, 2, 4]`

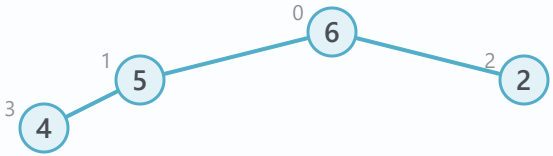
`yeni = [6, 5, 2, 4, 0, 0, 0, 0]`

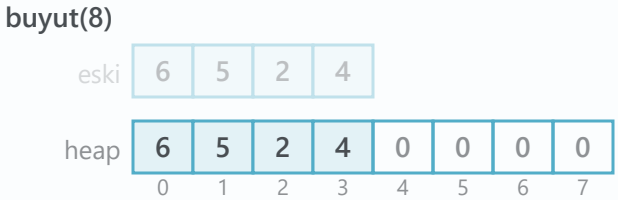
**ÇAĞRI YIĞINI**

`buyut(8)`

`ekle(1)`

# Kod İzleme: `buyut(8)`




**heap = yeni**: alan artık yeni diziyi gösterir; eski dizi çöp toplayıcıya kalır. Ağaç aynı, yalnızca kapasite 8.

Adım 8 / 9

```
1 private void buyut(int kapasite) {
2   int[] yeni = new int[kapasite];
3   for (int k = 0; k < n; k++)
4     yeni[k] = heap[k];          // kopyala
▶   heap = yeni;                  // eski dizi çöpe
6 }
```

**DEĞİŞKENLER**

`kapasite = 8` `n = 4`

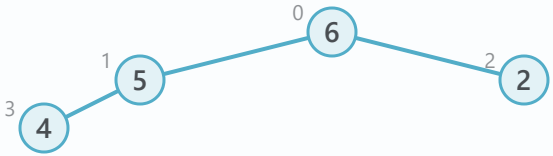
`heap = [6, 5, 2, 4, 0, 0, 0, 0]`

**ÇAĞRI YIĞINI**

`buyut(8)`

`ekle(1)`

# Kod İzleme: `buyut(8)`




buyut(8)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| eski | 6 | 5 | 2 | 4 | | | | |
| heap | 6 | 5 | 2 | 4 | 0 | 0 | 0 | 0 |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

buyut döner, ekle(1) devam eder. Kapasite her seferinde **iki katına** çıktığı için n eklemede toplam kopyalama < 2n → ekleme başına **amortize O(1)**.

Adım 9 / 9

```
1 private void buyut(int kapasite) {
2   int[] yeni = new int[kapasite];
3   for (int k = 0; k < n; k++)
4     yeni[k] = heap[k];          // kopyala
▶   heap = yeni;                  // eski dizi çöpe
6 }
```

**DEĞİŞKENLER**

`kapasite = 8` `n = 4`

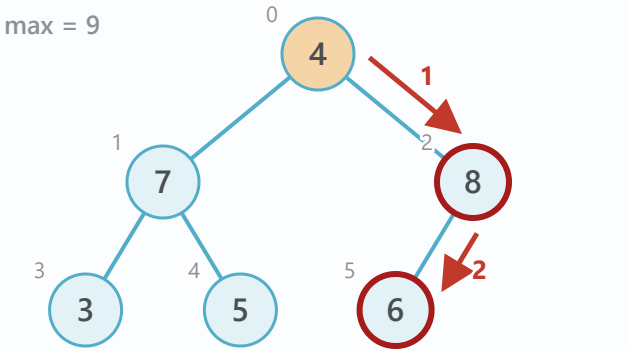
`heap = [6, 5, 2, 4, 0, 0, 0, 0]`

**ÇAĞRI YIĞINI**

`ekle(1)`

# En Büyüğü Çıkarma: Down-Heap (aşağı süzme)

1. Kökteki öğe (en büyük) saklanır; döndürülecek değer budur.
2. Ağacın **son öğesi** (heap[n − 1]) köke taşınır, n bir azalır.
3. Kök **büyük olan çocukla** yer değiştirir.
4. Öğe yaprağa ulaşana ya da iki çocuğundan da büyük/eşit olana kadar **aşağı** iner.



9 çıkarıldı, son öğe 4 köke → büyük çocukla iner

**Neden büyük çocuk?**

Küçük çocuk yukarı çıksaydı, diğer (büyük) çocuktan küçük bir ebeveyn olurdu. Min-heap'te ise *küçük* çocuk seçilir.

**Maliyet**

Her düzeyde 2 karşılaştırma, en çok h düzey → **O(log n)**.

# En Büyüğü Çıkarma: Java Kodu

```
public int silMax() {           // n > 0 olmalı
  int max = heap[0];
  yerDegistir(0, n - 1);  n--;  // son öğe köke
  batir(0);
  return max;
}
private void batir(int i) {       // down-heap
  while (2 * i + 1 < n) {       // sol çocuk var mı?
    int j = 2 * i + 1;            // j: büyük çocuk
    if (j + 1 < n && heap[j + 1] > heap[j]) j++;
    if (heap[i] >= heap[j]) return;
    yerDegistir(i, j);  i = j;    // bir düzey in
  }
}
```

**Boş heap**

Gerçek kodda önce `if (n == 0)` ile istisna fırlatılır.

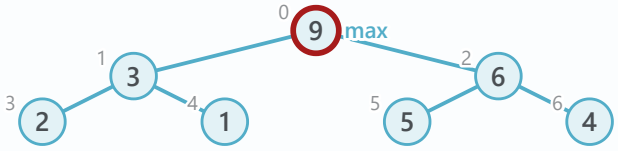
**kucult**

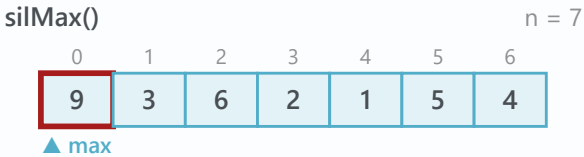
İsteğe bağlı: n = ¼ kapasite olunca dizi yarıya indirilir.

**Takas**

Max dizinin sonunda kalır → heap sort bunu kullanır.

# Kod İzleme: `silMax()` ve `batir`




**silMax()**: en büyük öğe her zaman kökte → **max = heap[0] = 9**.

Adım 1 / 8

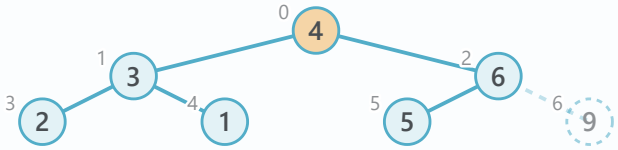
```
1  public int silMax() {           // n > 0 olmalı
▶    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {       // sol çocuk var mı?
9      int j = 2 * i + 1;            // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

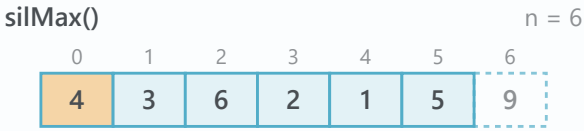
**DEĞİŞKENLER**

`max = 9` `n = 7`

`heap = [9, 3, 6, 2, 1, 5, 4]`

# Kod İzleme: `silMax()` ve `batir`




Kök ile son öğe yer değiştirir (0 ↔ 6), **n = 6**: 9 artık heap dışında (soluk). Ağaç tam kaldı; yeni kök 4 sırayı bozabilir.

Adım 2 / 8

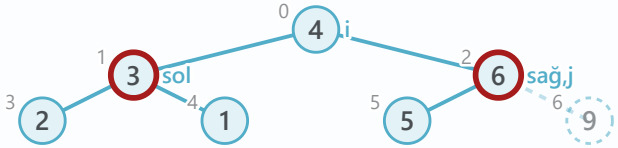
```
1  public int silMax() {           // n > 0 olmalı
2    int max = heap[0];
▶    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {       // sol çocuk var mı?
9      int j = 2 * i + 1;            // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

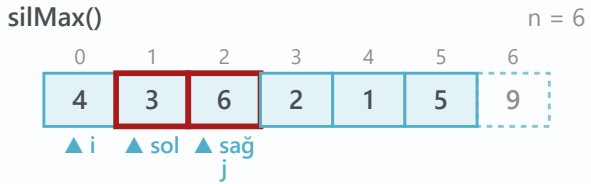
**DEĞİŞKENLER**

`max = 9` `n = 6`

`heap = [4, 3, 6, 2, 1, 5 | 9]`

# Kod İzleme: `silMax()` ve `batir`




Sol çocuk 1 (3), sağ çocuk 2 (6) → büyük olan **6**, **j = 2**.

Adım 3 / 8

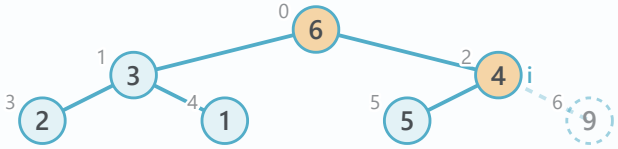
```
1  public int silMax() {            // n > 0 olmalı
2    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
▶    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
▶    while (2 * i + 1 < n) {       // sol çocuk var mı?
▶      int j = 2 * i + 1;           // j: büyük çocuk
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

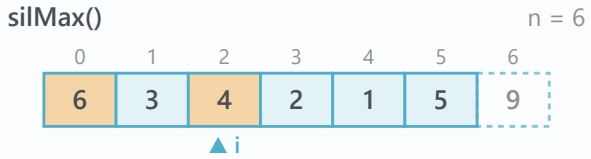
**DEĞİŞKENLER**

`max = 9` `i = 0` `j = 2` `n = 6`

`heap = [4, 3, 6, 2, 1, 5 | 9]`

# Kod İzleme: `silMax()` ve `batir`




4 < 6 → **yer değiştir** (indis 0 ↔ 2); büyük çocuk yukarı çıktı, **i = 2**.

Adım 4 / 8

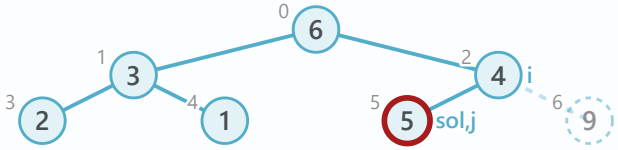
```
1  public int silMax() {            // n > 0 olmalı
2    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {        // sol çocuk var mı?
9      int j = 2 * i + 1;            // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
▶      yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

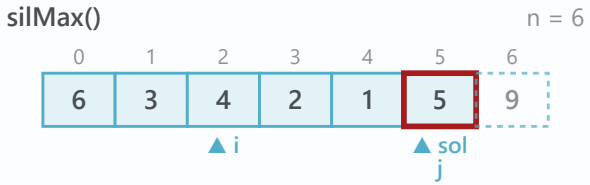
**DEĞİŞKENLER**

`max = 9` `i = 2` `j = 2` `n = 6`

`heap = [6, 3, 4, 2, 1, 5 | 9]`

# Kod İzleme: `silMax()` ve `batir`




Yalnızca sol çocuk var: indis 5 (5) → **j = 5**.

Adım 5 / 8

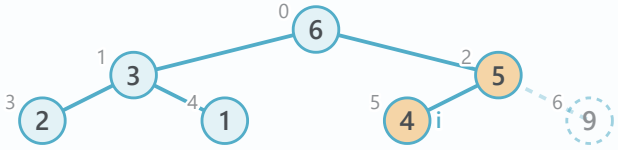
```
1  public int silMax() {            // n > 0 olmalı
2    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
▶    while (2 * i + 1 < n) {        // sol çocuk var mı?
▶      int j = 2 * i + 1;            // j: büyük çocuk
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

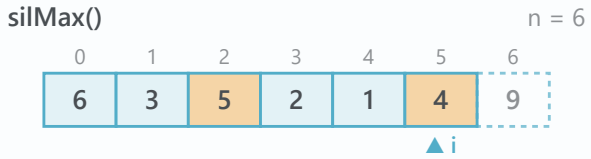
**DEĞİŞKENLER**

`max = 9` `i = 2` `j = 5` `n = 6`

`heap = [6, 3, 4, 2, 1, 5 | 9]`

# Kod İzleme: `silMax()` ve `batir`




4 < 5 → **yer değiştir** (indis 2 ↔ 5); büyük çocuk yukarı çıktı, **i = 5**.

Adım 6 / 8

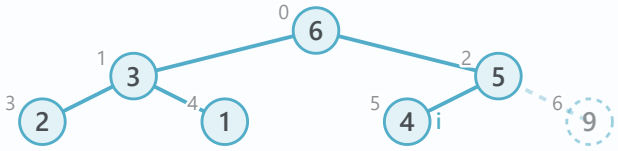
```
1  public int silMax() {            // n > 0 olmalı
2    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {        // sol çocuk var mı?
9      int j = 2 * i + 1;            // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
▶      yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

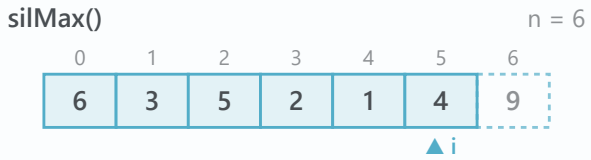
**DEĞİŞKENLER**

`max = 9` `i = 5` `j = 5` `n = 6`

`heap = [6, 3, 5, 2, 1, 4 | 9]`

# Kod İzleme: `silMax()` ve `batir`




i = 5: 2·5 + 1 = 11 ≥ n = 6 → çocuk yok (yaprak), döngü biter.

Adım 7 / 8

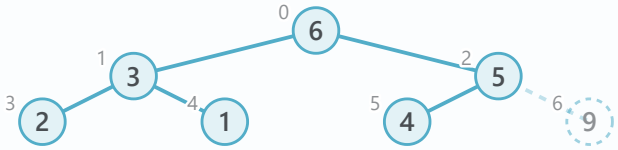
```
1  public int silMax() {            // n > 0 olmalı
2    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
▶    while (2 * i + 1 < n) {        // sol çocuk var mı?
9      int j = 2 * i + 1;            // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

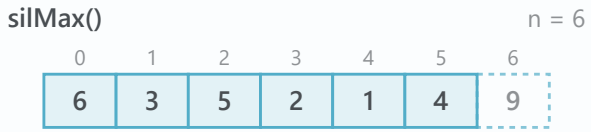
**DEĞİŞKENLER**

`max = 9` `i = 5` `n = 6`

`heap = [6, 3, 5, 2, 1, 4 | 9]`

# Kod İzleme: `silMax()` ve `batir`




batir bitti; yine geçerli bir max-heap. **return 9**. Yol boyu en çok ⌊log₂ n⌋ düzey → **O(log n)**.

Adım 8 / 8

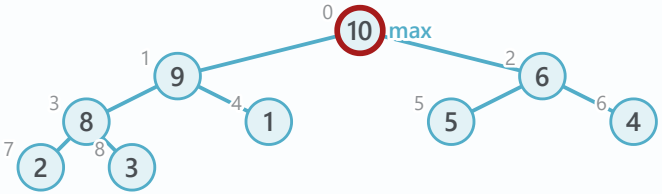
```
1  public int silMax() {            // n > 0 olmalı
2    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
▶    return max;
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {        // sol çocuk var mı?
9      int j = 2 * i + 1;            // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

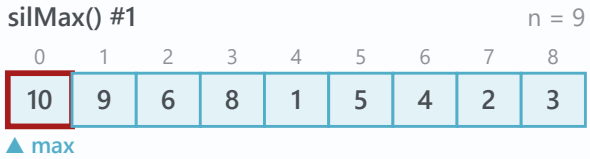
**DEĞİŞKENLER**

`max = 9` `n = 6`

`heap = [6, 3, 5, 2, 1, 4 | 9]`

# Kod İzleme: art arda iki `silMax()`




**silMax() #1**: en büyük öğe her zaman kökte → **max = heap[0] = 10**.

Adım 1 / 17

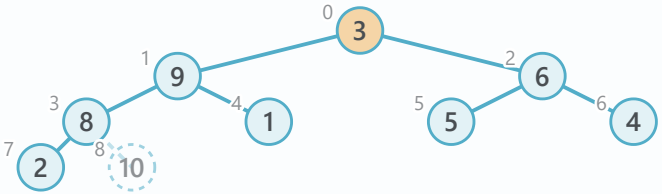
```
1  public int silMax() {           // n > 0 olmalı
▶    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {       // sol çocuk var mı?
9      int j = 2 * i + 1;           // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

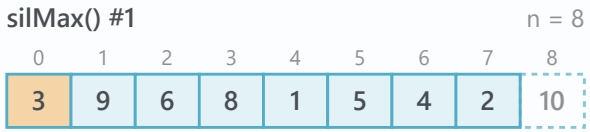
**DEĞİŞKENLER**

`max = 10` `n = 9`

`heap = [10, 9, 6, 8, 1, 5, 4, 2, 3]`

# Kod İzleme: art arda iki `silMax()`




Kök ile son öğe yer değiştirir (0 ↔ 8), **n = 8**: 10 artık heap dışında (soluk). Ağaç tam kaldı; yeni kök 3 sırayı bozabilir.

Adım 2 / 17

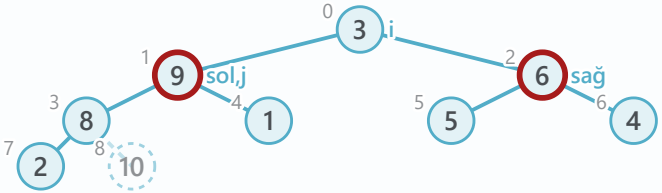
```
1  public int silMax() {           // n > 0 olmalı
2    int max = heap[0];
▶    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {       // sol çocuk var mı?
9      int j = 2 * i + 1;            // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

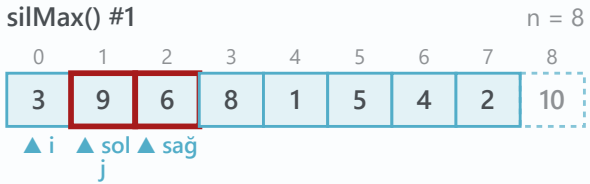
**DEĞİŞKENLER**

`max = 10` `n = 8`

`heap = [3, 9, 6, 8, 1, 5, 4, 2 | 10]`

# Kod İzleme: art arda iki `silMax()`




Sol çocuk 1 (9), sağ çocuk 2 (6) → büyük olan **9**, **j = 1**.

Adım 3 / 17

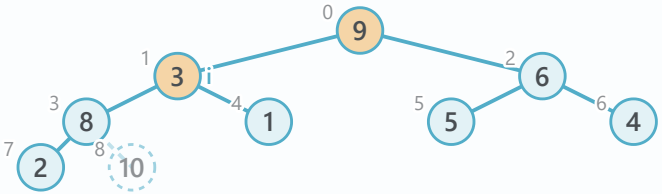
```
1  public int silMax() {            // n > 0 olmalı
2    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
▶    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
▶    while (2 * i + 1 < n) {       // sol çocuk var mı?
▶      int j = 2 * i + 1;           // j: büyük çocuk
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

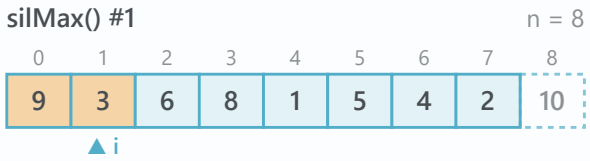
**DEĞİŞKENLER**

`max = 10` `i = 0` `j = 1` `n = 8`

`heap = [3, 9, 6, 8, 1, 5, 4, 2 | 10]`

# Kod İzleme: art arda iki `silMax()`




3 < 9 → **yer değiştir** (indis 0 ↔ 1); büyük çocuk yukarı çıktı, **i = 1**.



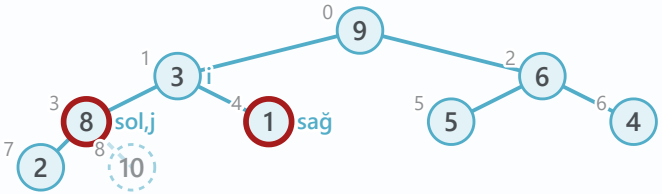
```
1  public int silMax() {            // n > 0 olmalı
2    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {        // sol çocuk var mı?
9      int j = 2 * i + 1;           // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
▶      yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

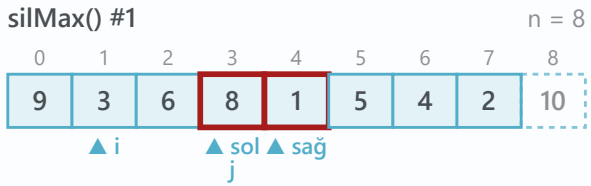
**DEĞİŞKENLER**

`max = 10` `i = 1` `j = 1` `n = 8`

`heap = [9, 3, 6, 8, 1, 5, 4, 2 | 10]`

# Kod İzleme: art arda iki `silMax()`




Sol çocuk 3 (8), sağ çocuk 4 (1) → büyük olan **8**, **j = 3**.

Adım 5 / 17

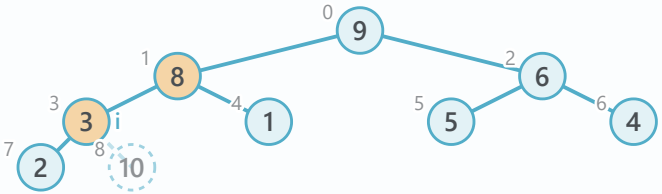
```
1  public int silMax() {            // n > 0 olmalı
2    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
▶    while (2 * i + 1 < n) {       // sol çocuk var mı?
▶      int j = 2 * i + 1;           // j: büyük çocuk
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

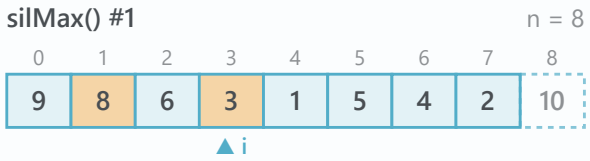
**DEĞİŞKENLER**

`max = 10` `i = 1` `j = 3` `n = 8`

`heap = [9, 3, 6, 8, 1, 5, 4, 2 | 10]`

# Kod İzleme: art arda iki `silMax()`




3 < 8 → **yer değiştir** (indis 1 ↔ 3); büyük çocuk yukarı çıktı, **i = 3**.

Adım 6 / 17

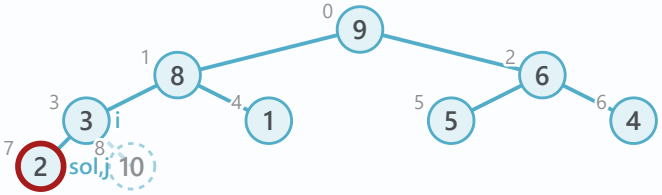
```
1  public int silMax() {            // n > 0 olmalı
2    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {        // sol çocuk var mı?
9      int j = 2 * i + 1;           // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
▶      yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

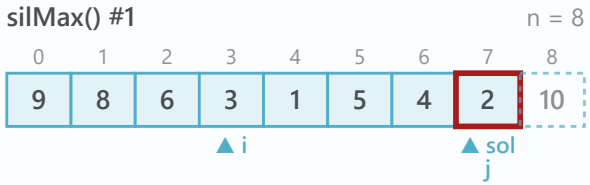
**DEĞİŞKENLER**

`max = 10` `i = 3` `j = 3` `n = 8`

`heap = [9, 8, 6, 3, 1, 5, 4, 2 | 10]`

# Kod İzleme: art arda iki `silMax()`




Yalnızca sol çocuk var: indis 7 (2) → **j = 7**.

Adım 7 / 17

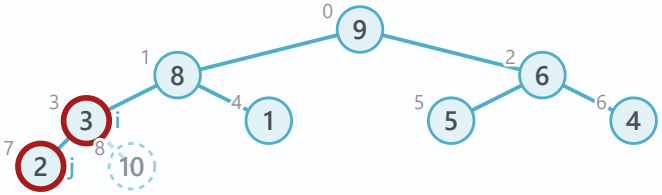
```
1  public int silMax() {            // n > 0 olmalı
2    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
▶    while (2 * i + 1 < n) {       // sol çocuk var mı?
▶      int j = 2 * i + 1;           // j: büyük çocuk
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

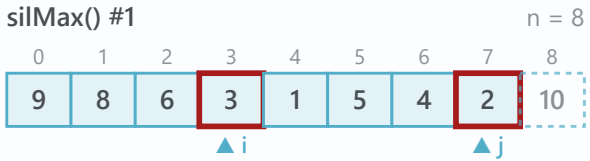
**DEĞİŞKENLER**

`max = 10` `i = 3` `j = 7` `n = 8`

`heap = [9, 8, 6, 3, 1, 5, 4, 2 | 10]`

# Kod İzleme: art arda iki `silMax()`




heap[3] = 3 ≥ heap[7] = 2 → iki çocuktan da büyük/eşit, sıra sağlandı: **return**.

Adım 8 / 17

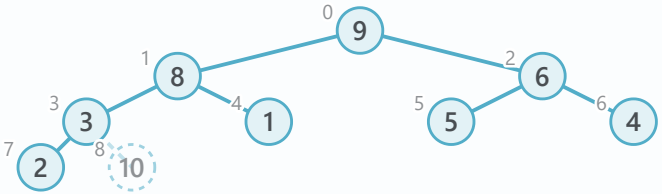
```
1  public int silMax() {            // n > 0 olmalı
2    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {        // sol çocuk var mı?
9      int j = 2 * i + 1;            // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
▶      if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

**DEĞİŞKENLER**

`max = 10` `i = 3` `j = 7` `n = 8`

`heap = [9, 8, 6, 3, 1, 5, 4, 2 | 10]`

# Kod İzleme: art arda iki `silMax()`




silMax() #1 n = 8

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 9 | 8 | 6 | 3 | 1 | 5 | 4 | 2 | 10 |

batir bitti; yine geçerli bir max-heap. **return 10**. Yol boyu en çok $\lfloor \log_2 n \rfloor$ düzey → **O(log n)**.

Adım 9 / 17

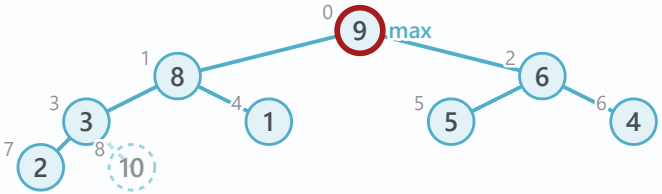
```
1  public int silMax() {            // n > 0 olmalı
2    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
▶    return max;
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {        // sol çocuk var mı?
9      int j = 2 * i + 1;            // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

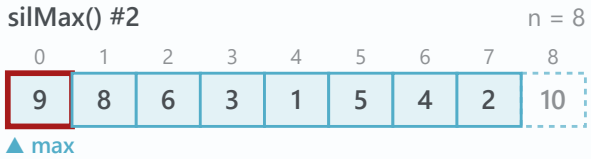
**DEĞİŞKENLER**

`max = 10` `n = 8`

`heap = [9, 8, 6, 3, 1, 5, 4, 2 | 10]`

# Kod İzleme: art arda iki `silMax()`




**silMax() #2**: en büyük öğe her zaman kökte → **max = heap[0] = 9**.

Adım 10 / 17

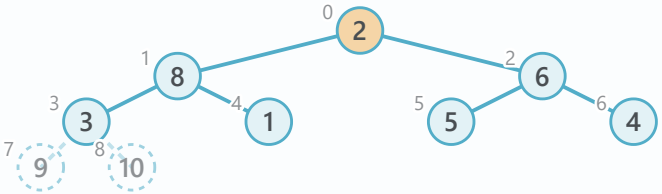
```
1  public int silMax() {            // n > 0 olmalı
▶    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {        // sol çocuk var mı?
9      int j = 2 * i + 1;           // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

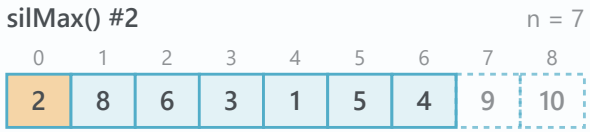
**DEĞİŞKENLER**

`max = 9` `n = 8`

`heap = [9, 8, 6, 3, 1, 5, 4, 2 | 10]`

# Kod İzleme: art arda iki `silMax()`




Kök ile son öğe yer değiştirir (0 ↔ 7), **n = 7**: 9 artık heap dışında (soluk). Ağaç tam kaldı; yeni kök 2 sırayı bozabilir.

Adım 11 / 17

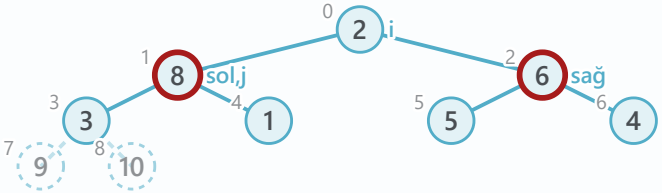
```
1  public int silMax() {           // n > 0 olmalı
2    int max = heap[0];
▶    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {       // sol çocuk var mı?
9      int j = 2 * i + 1;            // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

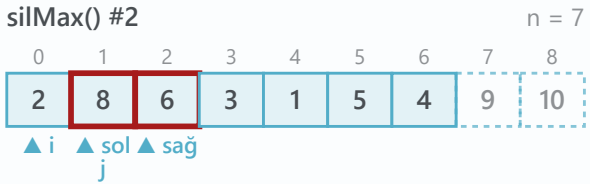
**DEĞİŞKENLER**

`max = 9` `n = 7`

`heap = [2, 8, 6, 3, 1, 5, 4 | 9, 10]`

# Kod İzleme: art arda iki `silMax()`




Sol çocuk 1 (8), sağ çocuk 2 (6) → büyük olan **8**, **j = 1**.

Adım 12 / 17

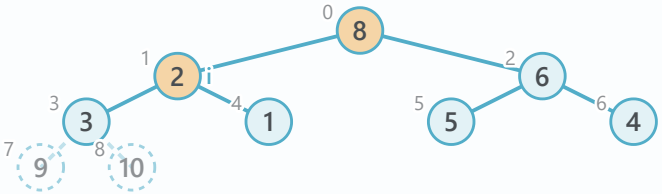
```
1  public int silMax() {            // n > 0 olmalı
2    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
▶    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
▶    while (2 * i + 1 < n) {       // sol çocuk var mı?
▶      int j = 2 * i + 1;           // j: büyük çocuk
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

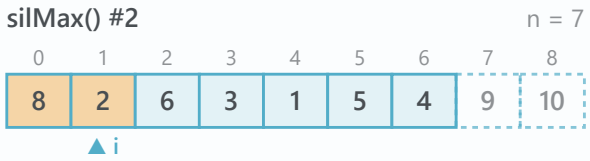
**DEĞİŞKENLER**

`max = 9` `i = 0` `j = 1` `n = 7`

`heap = [2, 8, 6, 3, 1, 5, 4 | 9, 10]`

# Kod İzleme: art arda iki `silMax()`




2 < 8 → **yer değiştir** (indis 0 ↔ 1); büyük çocuk yukarı çıktı, **i = 1**.

Adım 13 / 17

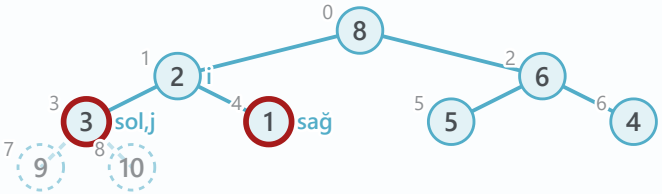
```
1  public int silMax() {            // n > 0 olmalı
2    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {        // sol çocuk var mı?
9      int j = 2 * i + 1;           // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
▶      yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

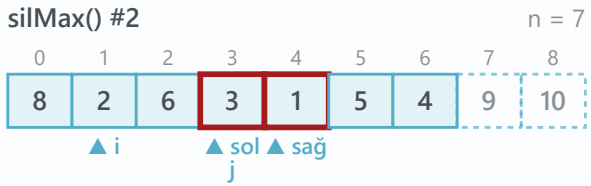
**DEĞİŞKENLER**

`max = 9` `i = 1` `j = 1` `n = 7`

`heap = [8, 2, 6, 3, 1, 5, 4 | 9, 10]`

# Kod İzleme: art arda iki `silMax()`




Sol çocuk 3 (3), sağ çocuk 4 (1) → büyük olan **3**, **j = 3**.

Adım 14 / 17

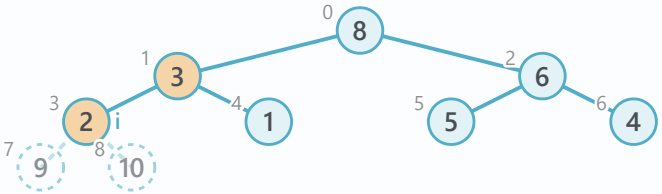
```
1  public int silMax() {            // n > 0 olmalı
2    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
▶    while (2 * i + 1 < n) {       // sol çocuk var mı?
▶      int j = 2 * i + 1;           // j: büyük çocuk
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

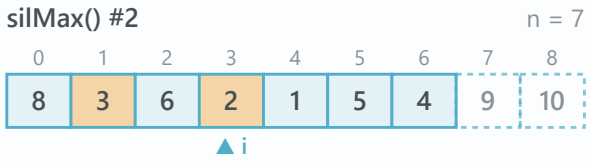
**DEĞİŞKENLER**

`max = 9` `i = 1` `j = 3` `n = 7`

`heap = [8, 2, 6, 3, 1, 5, 4 | 9, 10]`

# Kod İzleme: art arda iki `silMax()`




2 < 3 → **yer değiştir** (indis 1 ↔ 3); büyük çocuk yukarı çıktı, **i = 3**.

Adım 15 / 17

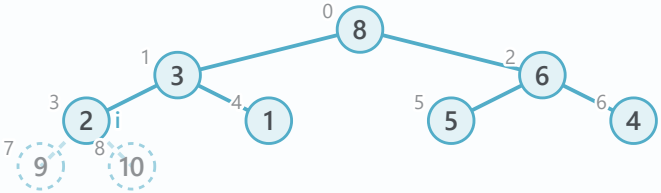
```
1  public int silMax() {            // n > 0 olmalı
2    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {        // sol çocuk var mı?
9      int j = 2 * i + 1;            // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
▶      yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

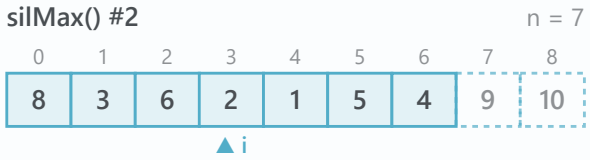
**DEĞİŞKENLER**

`max = 9` `i = 3` `j = 3` `n = 7`

`heap = [8, 3, 6, 2, 1, 5, 4 | 9, 10]`

# Kod İzleme: art arda iki `silMax()`




i = 3: 2·3 + 1 = 7 ≥ n = 7 → çocuk yok (yaprak), döngü biter.

Adım 16 / 17

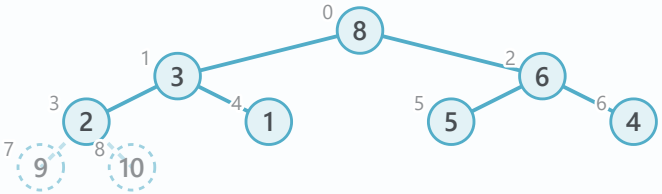
```
1  public int silMax() {            // n > 0 olmalı
2    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
5    return max;
6  }
7  private void batir(int i) {       // down-heap
▶    while (2 * i + 1 < n) {        // sol çocuk var mı?
9      int j = 2 * i + 1;            // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

**DEĞİŞKENLER**

`max = 9` `i = 3` `n = 7`

`heap = [8, 3, 6, 2, 1, 5, 4 | 9, 10]`

# Kod İzleme: art arda iki `silMax()`




silMax() #2 — n = 7

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 8 | 3 | 6 | 2 | 1 | 5 | 4 | 9 | 10 |

batir bitti; yine geçerli bir max-heap. **return 9**. Yol boyu en çok $\lfloor \log_2 n \rfloor$ düzey → **O(log n)**.

Adım 17 / 17

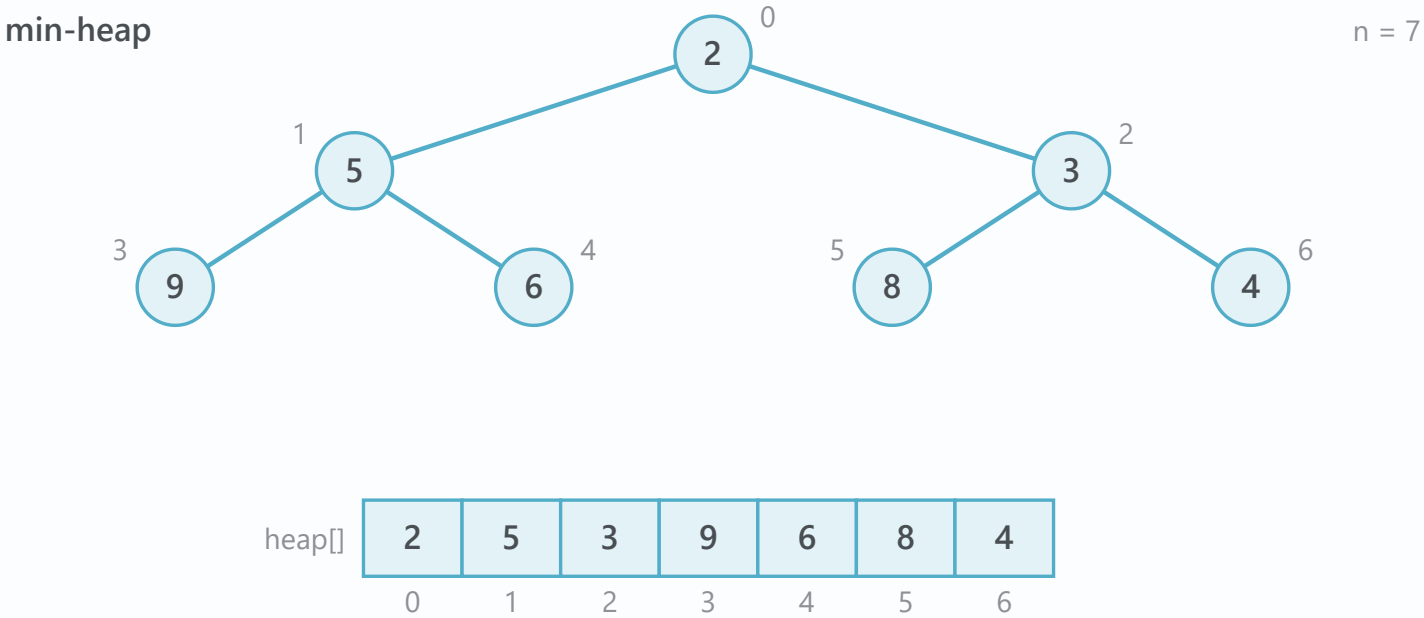
```
1  public int silMax() {            // n > 0 olmalı
2    int max = heap[0];
3    yerDegistir(0, n - 1);  n--;  // son öğe köke
4    batir(0);
▶    return max;
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {        // sol çocuk var mı?
9      int j = 2 * i + 1;            // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;    // bir düzey in
13   }
14 }
```

**DEĞİŞKENLER**

`max = 9` `n = 7`

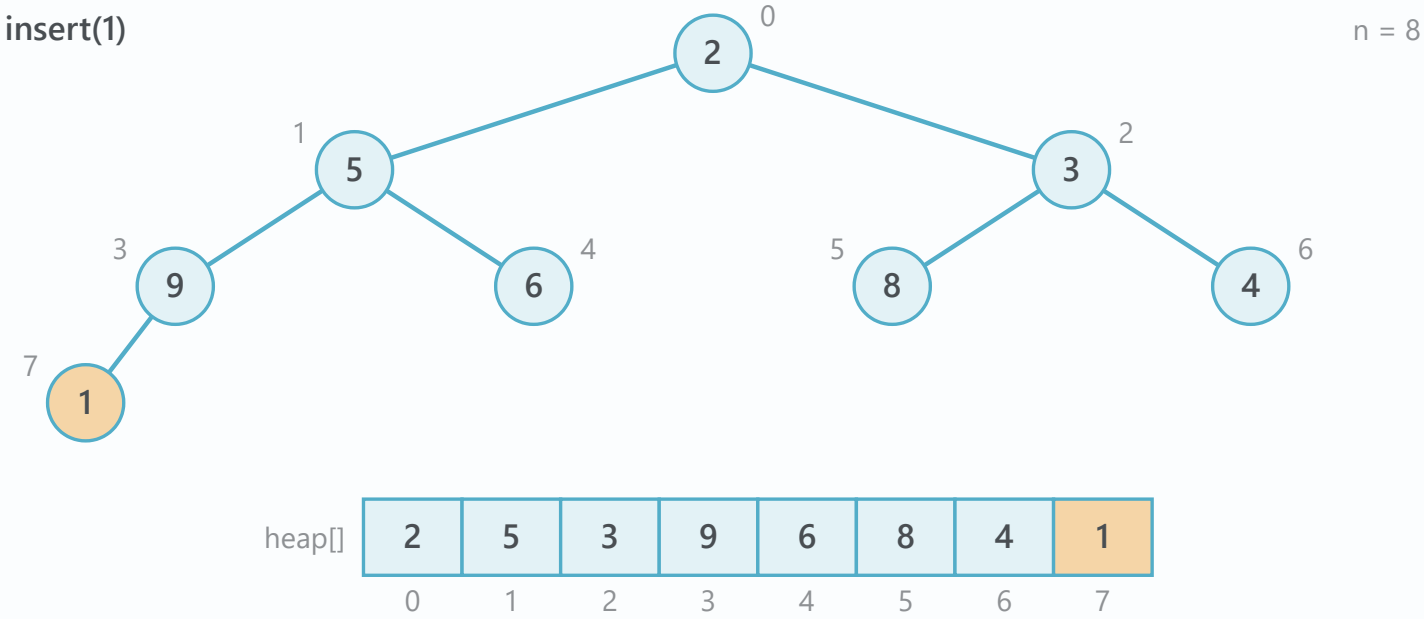
`heap = [8, 3, 6, 2, 1, 5, 4 | 9, 10]`

# Min-Heap: insert(1) ve removeMin()



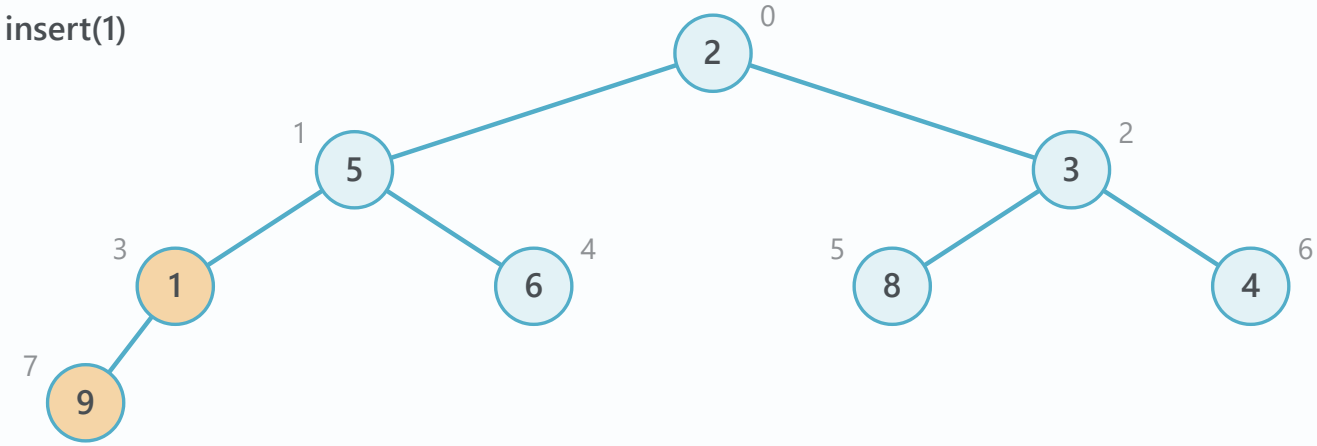

Başlangıç min-heap: her ebeveyn ≤ çocukları; en küçük öğe (2) kökte.

# Min-Heap: insert(1) ve removeMin()




**insert(1)**: 1, ilk boş konuma (indis 7) eklenir. Ebeveyni: (7−1)/2 = 3 → 9.

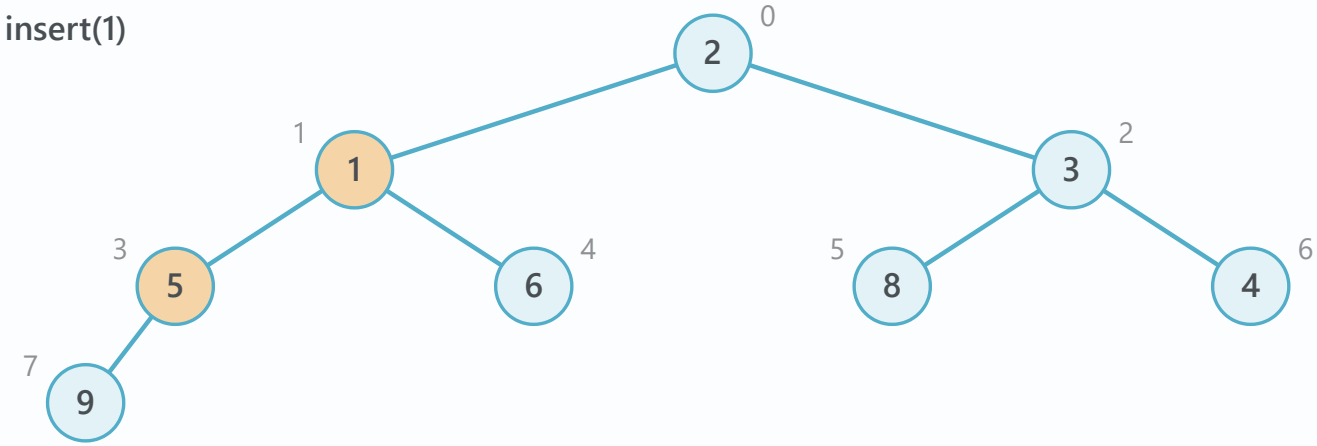
# Min-Heap: insert(1) ve removeMin()




| heap[] | 2 | 5 | 3 | 1 | 6 | 8 | 4 | 9 |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

1 < 9 → yer değiştir (indis 7 ↔ 3).

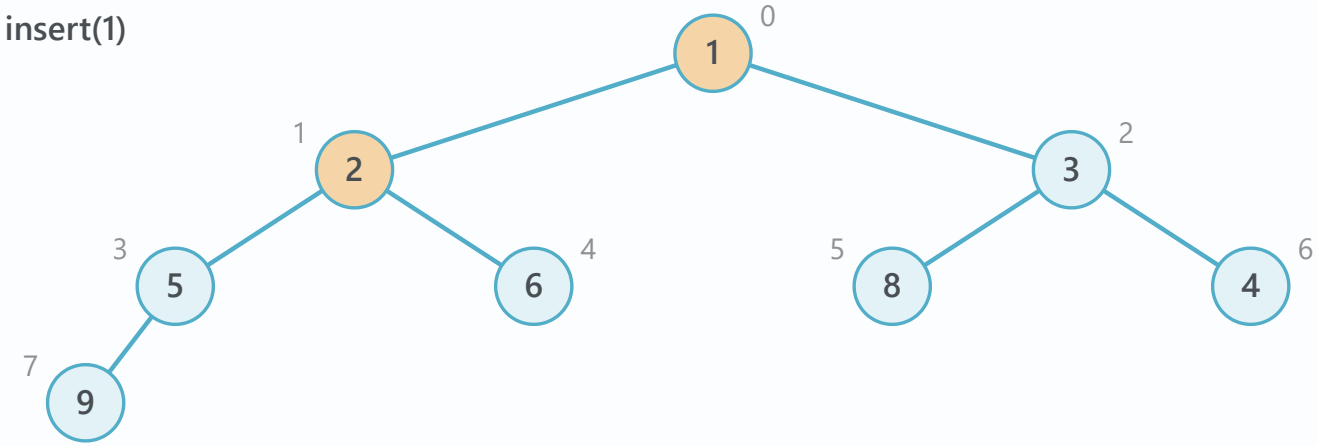
# Min-Heap: insert(1) ve removeMin()




| heap[] | 2 | 1 | 3 | 5 | 6 | 8 | 4 | 9 |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

1 < 5 → yer değiştir (indis 3 ↔ 1).

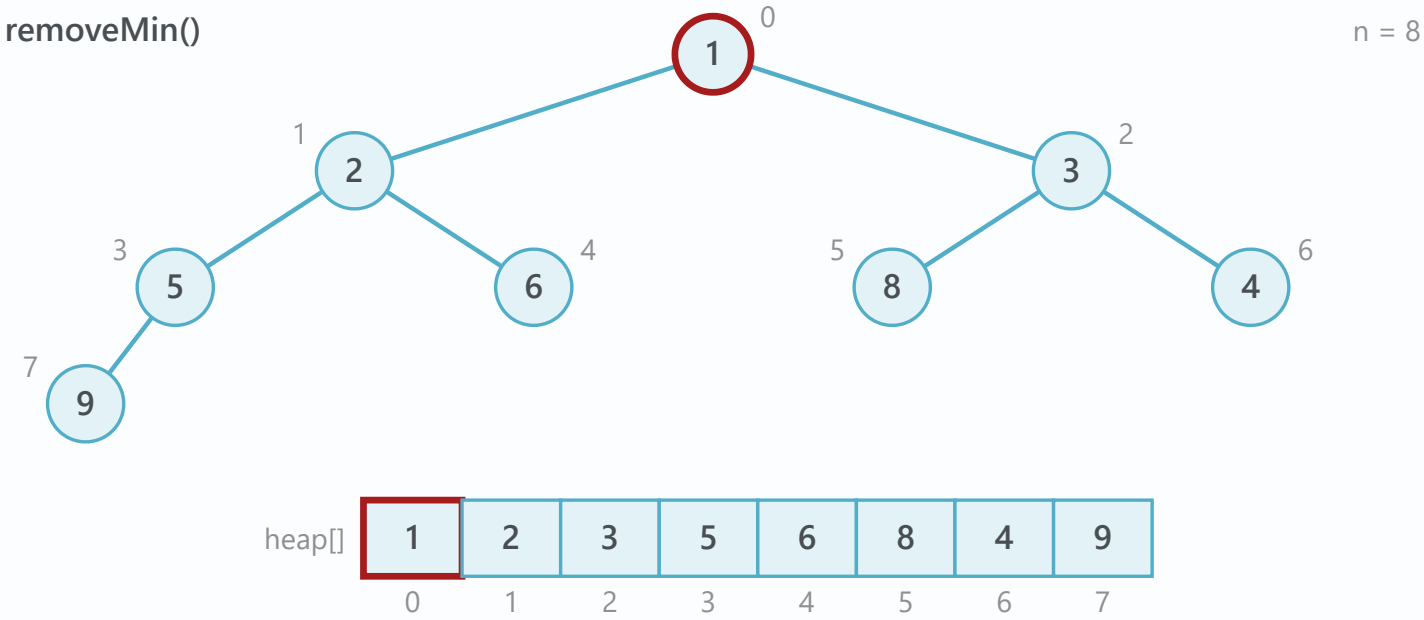
# Min-Heap: insert(1) ve removeMin()




| heap[] | 1 | 2 | 3 | 5 | 6 | 8 | 4 | 9 |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

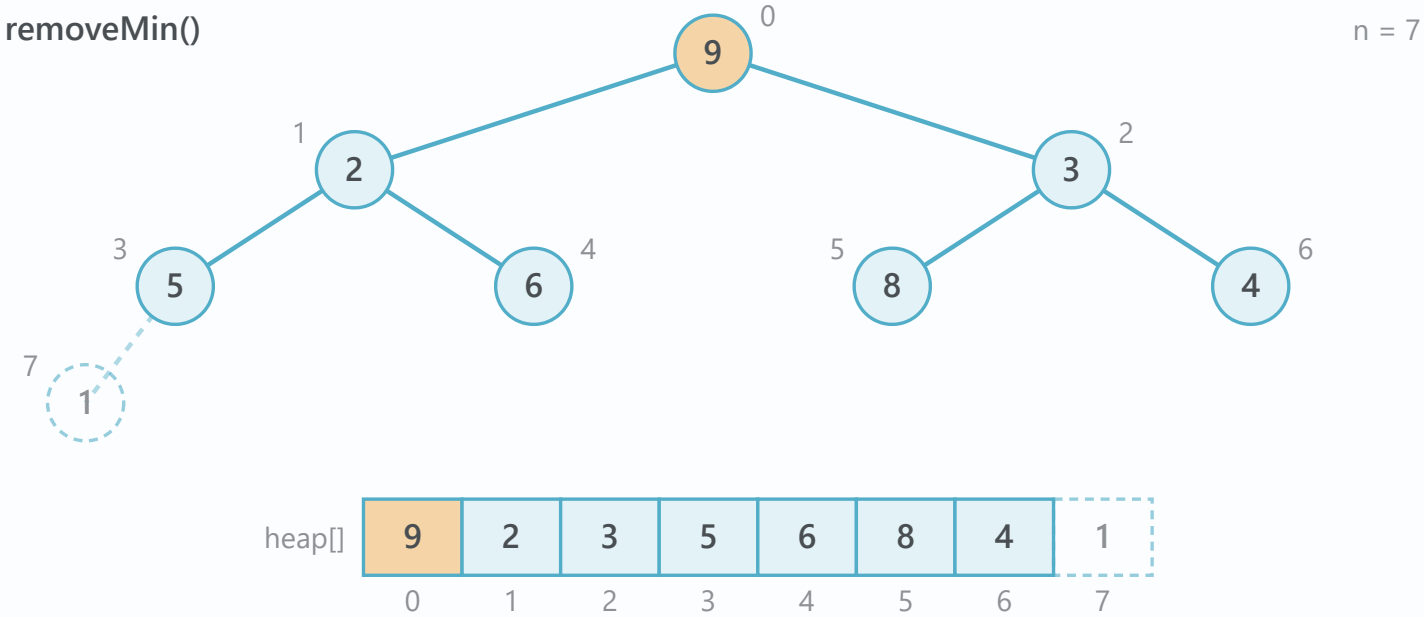
1 < 2 → yer değiştir (indis 1 ↔ 0). Kök → dur.

# Min-Heap: insert(1) ve removeMin()



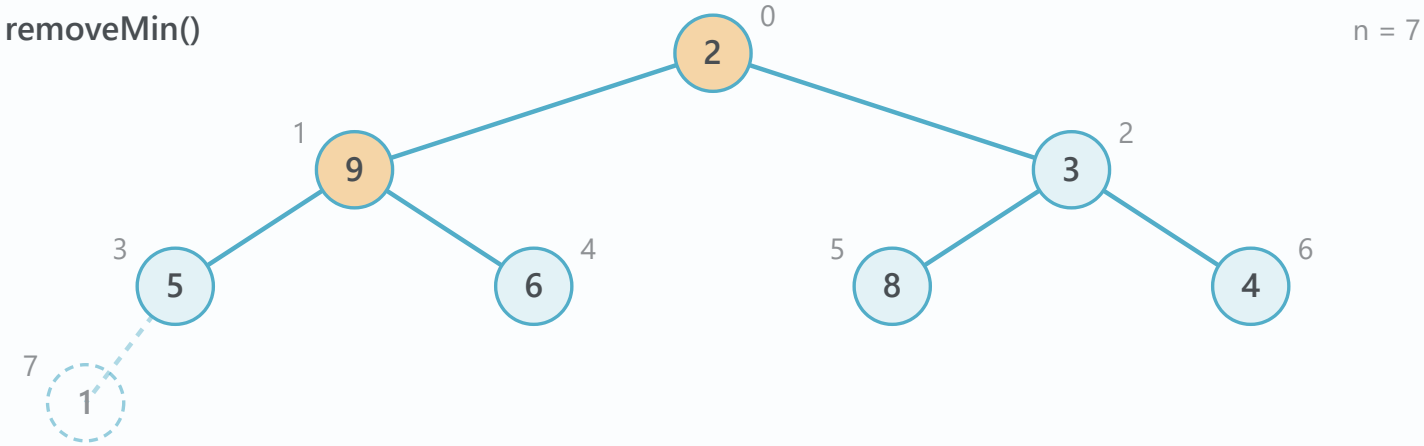

**removeMin()**: min = heap[0] = 1 (kök); döndürülecek değer.

# Min-Heap: insert(1) ve removeMin()



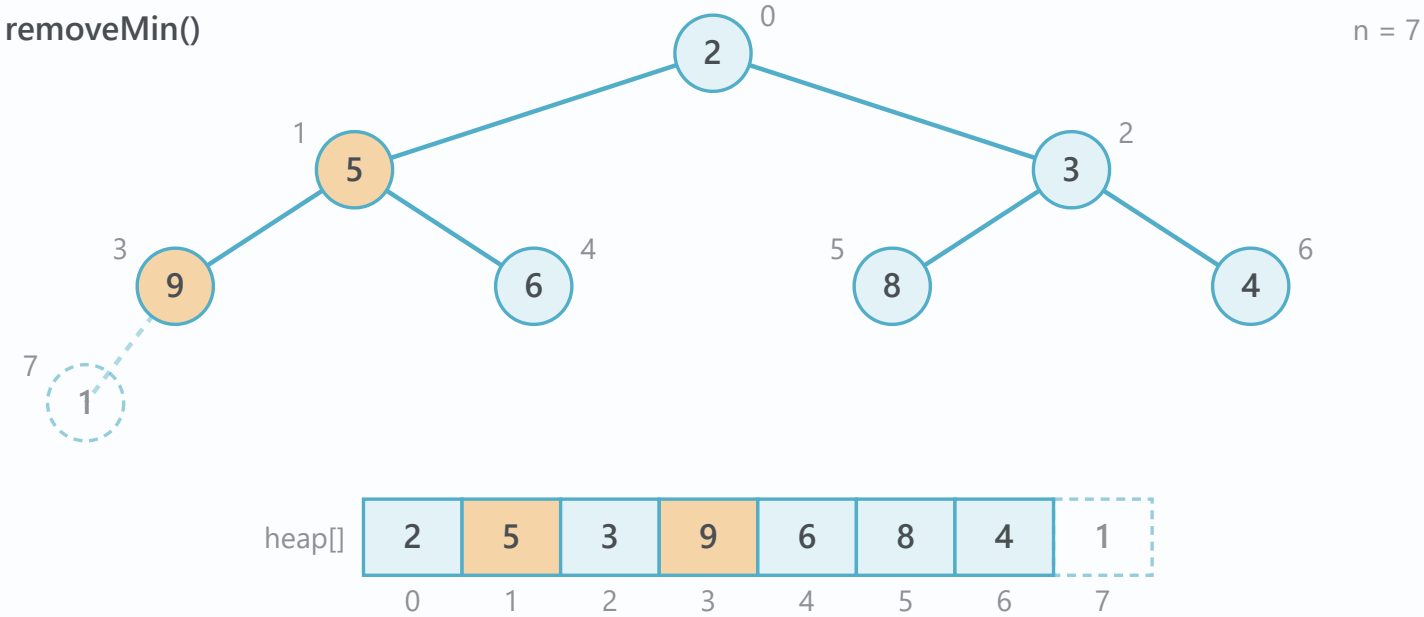

Kök ile son öğe yer değiştirir (0 ↔ 7); n = 7, 1 heap'in dışında. Yeni kök 9 özelliği bozabilir → down-heap.

# Min-Heap: insert(1) ve removeMin()




Çocuklar: 2 ve 3 → küçük olan 2. 9 > 2 → yer değiştir (0 ↔ 1).

# Min-Heap: insert(1) ve removeMin()




Çocuklar: 5 ve 6 → küçük olan 5. 9 > 5 → yer değiştir (1 ↔ 3). İndis 3 yaprak → dur.

EKLEME VE SİLME

# Heap İşlemlerinin Maliyeti

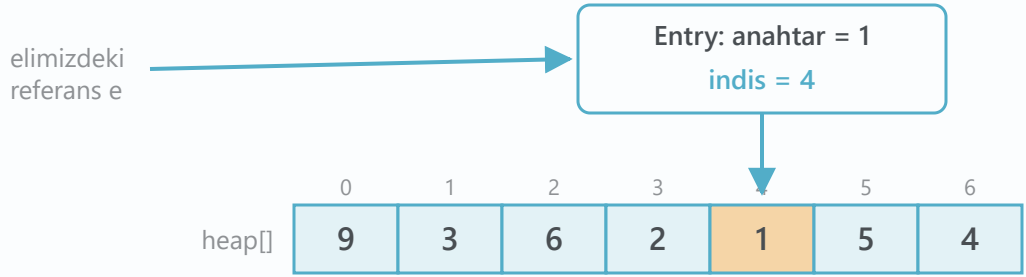
| İşlem | Maliyet | Neden? |
|---|---|---|
| `size`, `isEmpty` | O(1) | n sayacı tutulur |
| `max` / `min` (peek) | O(1) | Öğe hep heap[0]'dadır |
| `ekle` (insert) | O(log n)* | Up-heap: yapraktan köke tek yol |
| `silMax` (removeMin) | O(log n)* | Down-heap: kökten yaprağa tek yol |
| Rastgele öğeyi arama | O(n) | Heap sıralı değildir; arama için uygun değil |

* Dizi büyütme/küçültme dahil amortize maliyet. Yükseklik $\lfloor \log_2 n \rfloor$: $n = 10^6$ öğede en çok 19 yer değiştirme.

EKLEME VE SİLME

# Uyarlanabilir Öncelikli Kuyruk (Adaptable PQ)

Bazen kuyruktaki bir öğenin **önceliği değişir** (Dijkstra'da mesafe kısalır, acil serviste hasta ağırlaşır) ya da öğe sıradan **çıkarılır**. Goodrich bu ek işlemleri tanımlar:




Konum (locator): öğe, heap dizisindeki yerini bilir; her takasta güncellenir.

**replaceKey(e, k)**

`heap[e.indis]` değişir; büyüdüyse **up-heap**, küçüldüyse **down-heap** (max-heap).

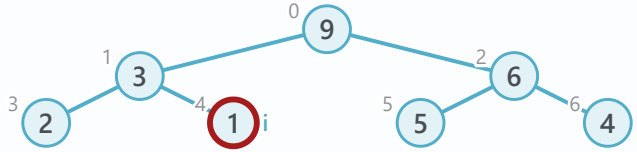
**remove(e)**

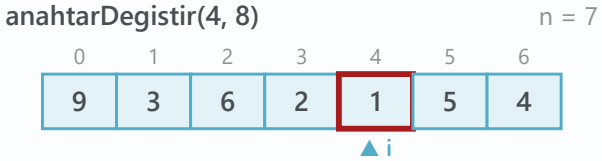
Son öğeyle takas, n − 1; yerine gelen öğe için up- ya da down-heap.

**Neden konum?**

Konumsuz öğeyi bulmak O(n); konumla O(1) → işlemler **O(log n)**.

# Kod İzleme: `anahtarDegistir(4, 8)`




**anahtarDegistir(4, 8)**: indis 4'teki anahtar 1. Eski değeri sakla: **eski = 1**.

Adım 1 / 7

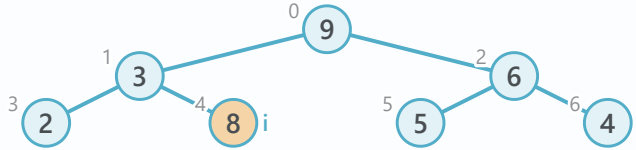
```
1  public void anahtarDegistir(int i, int yeni) {
▶    int eski = heap[i];
3    heap[i] = yeni;
4    if (yeni > eski) yuzdur(i);  // büyüdü: yukarı
5    else batir(i);               // küçüldü: aşağı
6  }
7  private void yuzdur(int i) {     // up-heap
8    while (i > 0) {
9      int ebeveyn = (i - 1) / 2;
10     if (heap[ebeveyn] >= heap[i]) return;
11     yerDegistir(i, ebeveyn);  i = ebeveyn;
12   }
13 }
```

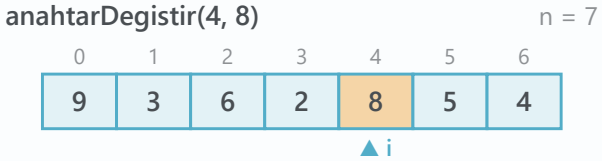
**DEĞİŞKENLER**

`i = 4` `yeni = 8` `eski = 1` `n = 7`

`heap = [9, 3, 6, 2, 1, 5, 4]`

# Kod İzleme: `anahtarDegistir(4, 8)`




**heap[4] = 8**. Ağaç hâlâ tam, ama heap-sıra özelliği bu düğümün çevresinde bozulmuş olabilir.

Adım 2 / 7

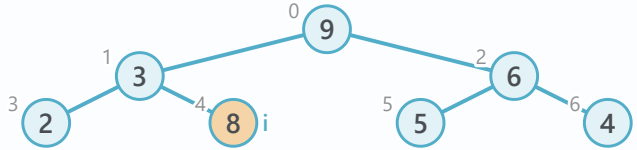
```
1  public void anahtarDegistir(int i, int yeni) {
2    int eski = heap[i];
▶    heap[i] = yeni;
4    if (yeni > eski) yuzdur(i);  // büyüdü: yukarı
5    else batir(i);               // küçüldü: aşağı
6  }
7  private void yuzdur(int i) {     // up-heap
8    while (i > 0) {
9      int ebeveyn = (i - 1) / 2;
10     if (heap[ebeveyn] >= heap[i]) return;
11     yerDegistir(i, ebeveyn);  i = ebeveyn;
12   }
13 }
```

**DEĞİŞKENLER**

`i = 4` `yeni = 8` `eski = 1` `n = 7`

`heap = [9, 3, 6, 2, 8, 5, 4]`

# Kod İzleme: `anahtarDegistir(4, 8)`




anahtarDegistir(4, 8) n = 7

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 9 | 3 | 6 | 2 | 8 | 5 | 4 |
| | | | | ▲ i | | |

8 > 1: anahtar **büyüdü** → yalnızca ebeveynle çatışabilir, **yuzdur(4)** çağrılır.

Adım 3 / 7

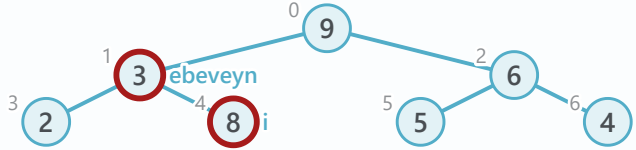
```
1  public void anahtarDegistir(int i, int yeni) {
2    int eski = heap[i];
3    heap[i] = yeni;
▶    if (yeni > eski) yuzdur(i);  // büyüdü: yukarı
5    else batir(i);               // küçüldü: aşağı
6  }
7  private void yuzdur(int i) {     // up-heap
8    while (i > 0) {
9      int ebeveyn = (i - 1) / 2;
10     if (heap[ebeveyn] >= heap[i]) return;
11     yerDegistir(i, ebeveyn);  i = ebeveyn;
12   }
13 }
```

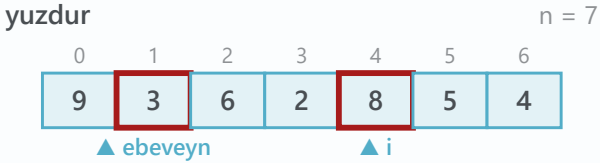
**DEĞİŞKENLER**

`i = 4` `yeni = 8` `eski = 1` `n = 7`

`heap = [9, 3, 6, 2, 8, 5, 4]`

# Kod İzleme: `anahtarDegistir(4, 8)`




Ebeveyn (4 − 1) / 2 = 1: heap[1] = 3 < 8 → yer değiştirmeli.

Adım 4 / 7

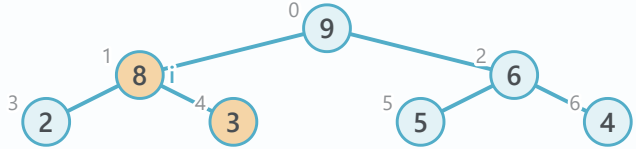
```
1  public void anahtarDegistir(int i, int yeni) {
2    int eski = heap[i];
3    heap[i] = yeni;
4    if (yeni > eski) yuzdur(i);  // büyüdü: yukarı
5    else batir(i);               // küçüldü: aşağı
6  }
7  private void yuzdur(int i) {     // up-heap
▶    while (i > 0) {
▶      int ebeveyn = (i - 1) / 2;
10     if (heap[ebeveyn] >= heap[i]) return;
11     yerDegistir(i, ebeveyn);  i = ebeveyn;
12   }
13 }
```

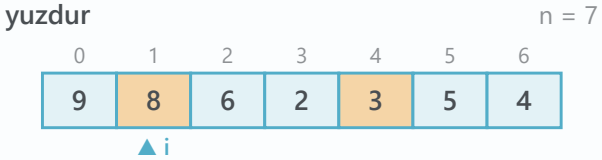
**DEĞİŞKENLER**

`i = 4` `ebeveyn = 1` `n = 7`

`heap = [9, 3, 6, 2, 8, 5, 4]`

# Kod İzleme: `anahtarDegistir(4, 8)`




Takas (indis 4 ↔ 1); **i = 1**. Aşağı inen 3, zaten diğer çocuğundan büyüktü.

Adım 5 / 7

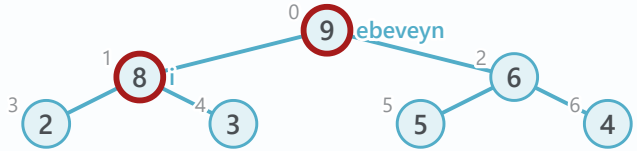
```
1  public void anahtarDegistir(int i, int yeni) {
2    int eski = heap[i];
3    heap[i] = yeni;
4    if (yeni > eski) yuzdur(i);  // büyüdü: yukarı
5    else batir(i);               // küçüldü: aşağı
6  }
7  private void yuzdur(int i) {     // up-heap
8    while (i > 0) {
9      int ebeveyn = (i - 1) / 2;
10     if (heap[ebeveyn] >= heap[i]) return;
▶      yerDegistir(i, ebeveyn);  i = ebeveyn;
12   }
13 }
```

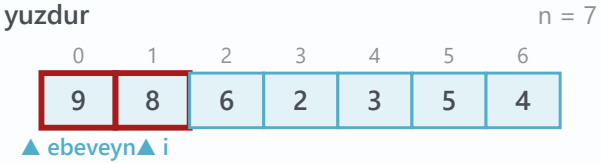
**DEĞİŞKENLER**

`i = 1` `ebeveyn = 1` `n = 7`

`heap = [9, 8, 6, 2, 3, 5, 4]`

# Kod İzleme: `anahtarDegistir(4, 8)`




Ebeveyn 0: heap[0] = 9 ≥ 8 → sıra sağlandı, **return**.

Adım 6 / 7

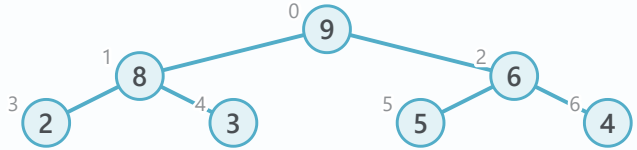
```
 1 public void anahtarDegistir(int i, int yeni) {
 2   int eski = heap[i];
 3   heap[i] = yeni;
 4   if (yeni > eski) yuzdur(i);  // büyüdü: yukarı
 5   else batir(i);               // küçüldü: aşağı
 6 }
 7 private void yuzdur(int i) {     // up-heap
▶   while (i > 0) {
▶     int ebeveyn = (i - 1) / 2;
▶     if (heap[ebeveyn] >= heap[i]) return;
11    yerDegistir(i, ebeveyn);  i = ebeveyn;
12  }
13 }
```

**DEĞİŞKENLER**

`i = 1` `ebeveyn = 0` `n = 7`

`heap = [9, 8, 6, 2, 3, 5, 4]`

# Kod İzleme: `anahtarDegistir(4, 8)`




anahtarDegistir(4, 8) n = 7

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 9 | 8 | 6 | 2 | 3 | 5 | 4 |

anahtarDegistir bitti: 8 bir düzey yukarı çıktı. Yalnızca 4 → 1 → 0 yolu incelendi: **O(log n)**.

Adım 7 / 7

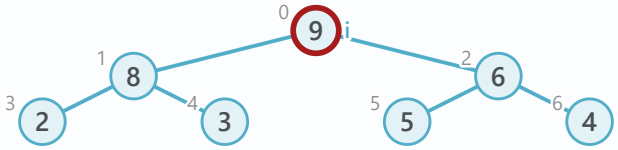
```
1  public void anahtarDegistir(int i, int yeni) {
2    int eski = heap[i];
3    heap[i] = yeni;
4    if (yeni > eski) yuzdur(i);  // büyüdü: yukarı
5    else batir(i);               // küçüldü: aşağı
▶  }
7  private void yuzdur(int i) {     // up-heap
8    while (i > 0) {
9      int ebeveyn = (i - 1) / 2;
10     if (heap[ebeveyn] >= heap[i]) return;
11     yerDegistir(i, ebeveyn);  i = ebeveyn;
12   }
13 }
```

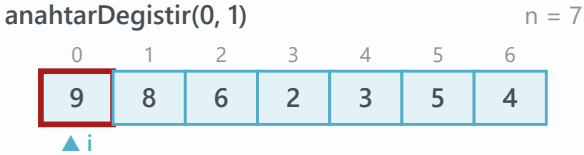
**DEĞİŞKENLER**

`n = 7`

`heap = [9, 8, 6, 2, 3, 5, 4]`

# Kod İzleme (devam): `anahtarDegistir(0, 1)`




**anahtarDegistir(0, 1)**: indis 0'daki anahtar 9. Eski değeri sakla: **eski = 9**.

Adım 1 / 9

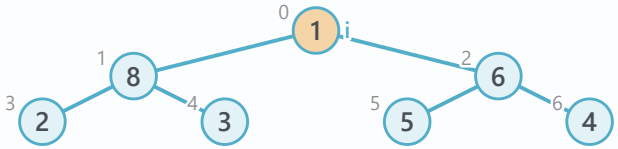
```
1  public void anahtarDegistir(int i, int yeni) {
2    int eski = heap[i];
3    heap[i] = yeni;
4    if (yeni > eski) yuzdur(i);  // büyüdü: yukarı
5    else batir(i);               // küçüldü: aşağı
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {
9      int j = 2 * i + 1;              // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;
13   }
14 }
```

**DEĞİŞKENLER**

`i = 0` `yeni = 1` `eski = 9` `n = 7`

`heap = [9, 8, 6, 2, 3, 5, 4]`

# Kod İzleme (devam): `anahtarDegistir(0, 1)`




anahtarDegistir(0, 1) n = 7

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 8 | 6 | 2 | 3 | 5 | 4 |
| ▲ i | | | | | | |

**heap[0] = 1**. Ağaç hâlâ tam, ama heap-sıra özelliği bu düğümün çevresinde bozulmuş olabilir.

Adım 2 / 9

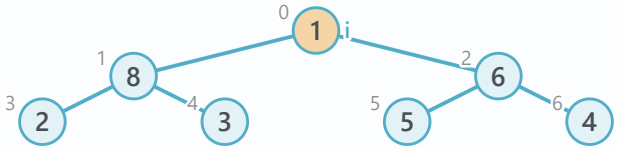
```
1  public void anahtarDegistir(int i, int yeni) {
2    int eski = heap[i];
▶    heap[i] = yeni;
4    if (yeni > eski) yuzdur(i);  // büyüdü: yukarı
5    else batir(i);               // küçüldü: aşağı
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {
9      int j = 2 * i + 1;              // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;
13   }
14 }
```

**DEĞİŞKENLER**

`i = 0` `yeni = 1` `eski = 9` `n = 7`

`heap = [1, 8, 6, 2, 3, 5, 4]`

# Kod İzleme (devam): `anahtarDegistir(0, 1)`




anahtarDegistir(0, 1) n = 7

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 8 | 6 | 2 | 3 | 5 | 4 |
| ▲ i | | | | | | |

1 < 9: anahtar **küçüldü** → yalnızca çocuklarla çatışabilir, **batir(0)** çağrılır.

Adım 3 / 9

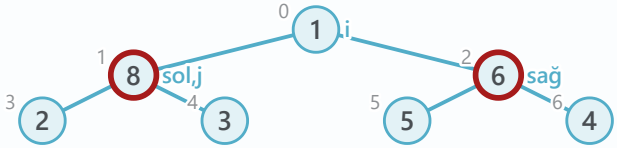
```
1  public void anahtarDegistir(int i, int yeni) {
2    int eski = heap[i];
3    heap[i] = yeni;
▶    if (yeni > eski) yuzdur(i);  // büyüdü: yukarı
▶    else batir(i);               // küçüldü: aşağı
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {
9      int j = 2 * i + 1;              // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;
13   }
14 }
```

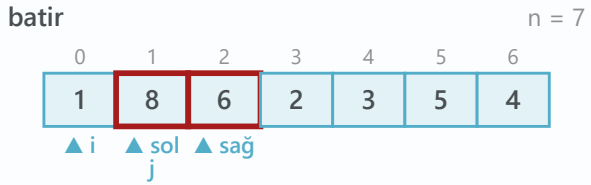
**DEĞİŞKENLER**

`i = 0` `yeni = 1` `eski = 9` `n = 7`

`heap = [1, 8, 6, 2, 3, 5, 4]`

# Kod İzleme (devam): `anahtarDegistir(0, 1)`




Çocuklar 8 ve 6 → büyük olan **8**, **j = 1**.

Adım 4 / 9

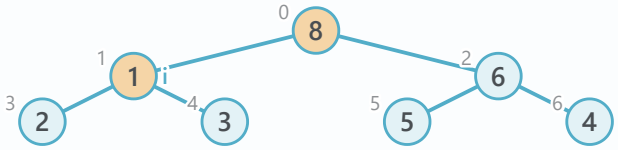
```
1  public void anahtarDegistir(int i, int yeni) {
2    int eski = heap[i];
3    heap[i] = yeni;
4    if (yeni > eski) yuzdur(i);  // büyüdü: yukarı
5    else batir(i);               // küçüldü: aşağı
6  }
7  private void batir(int i) {       // down-heap
▶    while (2 * i + 1 < n) {
▶      int j = 2 * i + 1;              // j: büyük çocuk
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;
13   }
14 }
```

**DEĞİŞKENLER**

`i = 0` `j = 1` `n = 7`

`heap = [1, 8, 6, 2, 3, 5, 4]`

# Kod İzleme (devam): `anahtarDegistir(0, 1)`




batir n = 7

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 8 | 1 | 6 | 2 | 3 | 5 | 4 |
| | ▲ i | | | | | |

1 < 8 → **takas** (0 ↔ 1), büyük çocuk yukarı çıktı; **i = 1**.

Adım 5 / 9

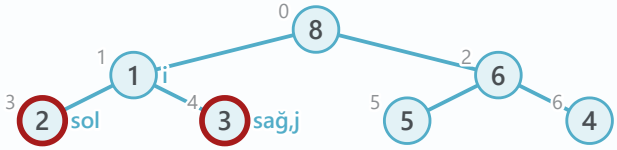
```
1  public void anahtarDegistir(int i, int yeni) {
2    int eski = heap[i];
3    heap[i] = yeni;
4    if (yeni > eski) yuzdur(i);  // büyüdü: yukarı
5    else batir(i);               // küçüldü: aşağı
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {
9      int j = 2 * i + 1;              // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
▶      yerDegistir(i, j);  i = j;
13   }
14 }
```

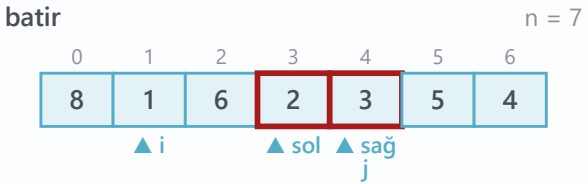
**DEĞİŞKENLER**

`i = 1` `j = 1` `n = 7`

`heap = [8, 1, 6, 2, 3, 5, 4]`

# Kod İzleme (devam): `anahtarDegistir(0, 1)`




Çocuklar 2 ve 3 → büyük olan **3**, **j = 4**.

Adım 6 / 9

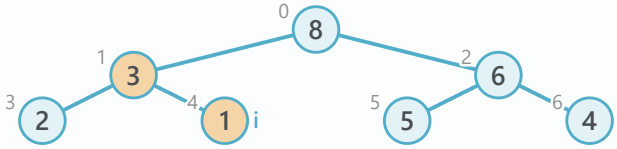
```
1  public void anahtarDegistir(int i, int yeni) {
2    int eski = heap[i];
3    heap[i] = yeni;
4    if (yeni > eski) yuzdur(i);  // büyüdü: yukarı
5    else batir(i);               // küçüldü: aşağı
6  }
7  private void batir(int i) {       // down-heap
▶    while (2 * i + 1 < n) {
▶      int j = 2 * i + 1;              // j: büyük çocuk
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;
13   }
14 }
```

**DEĞİŞKENLER**

`i = 1` `j = 4` `n = 7`

`heap = [8, 1, 6, 2, 3, 5, 4]`

# Kod İzleme (devam): `anahtarDegistir(0, 1)`




batir n = 7

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 8 | 3 | 6 | 2 | 1 | 5 | 4 |
| | | | | ▲ i | | |

1 < 3 → **takas** (1 ↔ 4), büyük çocuk yukarı çıktı; **i = 4**.

Adım 7 / 9

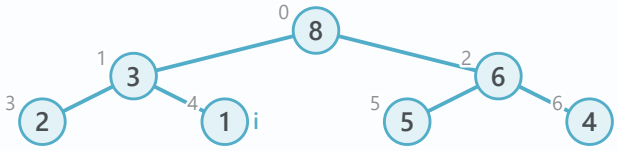
```
1  public void anahtarDegistir(int i, int yeni) {
2    int eski = heap[i];
3    heap[i] = yeni;
4    if (yeni > eski) yuzdur(i);  // büyüdü: yukarı
5    else batir(i);               // küçüldü: aşağı
6  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {
9      int j = 2 * i + 1;              // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
▶      yerDegistir(i, j);  i = j;
13   }
14 }
```

**DEĞİŞKENLER**

`i = 4` `j = 4` `n = 7`

`heap = [8, 3, 6, 2, 1, 5, 4]`

# Kod İzleme (devam): `anahtarDegistir(0, 1)`




batir n = 7

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 8 | 3 | 6 | 2 | 1 | 5 | 4 |
| | | | | ▲ i | | |

i = 4: 2·4 + 1 = 9 ≥ n → yaprak, döngü biter.

Adım 8 / 9

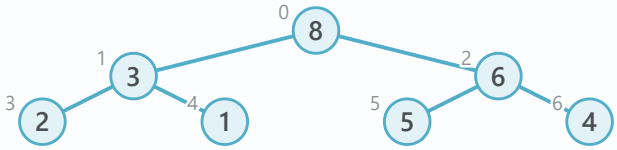
```
1  public void anahtarDegistir(int i, int yeni) {
2    int eski = heap[i];
3    heap[i] = yeni;
4    if (yeni > eski) yuzdur(i);  // büyüdü: yukarı
5    else batir(i);               // küçüldü: aşağı
6  }
7  private void batir(int i) {       // down-heap
▶    while (2 * i + 1 < n) {
9      int j = 2 * i + 1;              // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;
13   }
14 }
```

**DEĞİŞKENLER**

`i = 4` `n = 7`

`heap = [8, 3, 6, 2, 1, 5, 4]`

# Kod İzleme (devam): `anahtarDegistir(0, 1)`




anahtarDegistir(0, 1) n = 7

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 8 | 3 | 6 | 2 | 1 | 5 | 4 |

Bitti: 1 iki düzey indi, 8 köke çıktı. Kök için `anahtarDegistir(0, x)` + küçük x, silMax'ın batırma adımıyla aynı işi yapar.

Adım 9 / 9

```
1  public void anahtarDegistir(int i, int yeni) {
2    int eski = heap[i];
3    heap[i] = yeni;
4    if (yeni > eski) yuzdur(i);  // büyüdü: yukarı
5    else batir(i);               // küçüldü: aşağı
▶  }
7  private void batir(int i) {       // down-heap
8    while (2 * i + 1 < n) {
9      int j = 2 * i + 1;              // j: büyük çocuk
10     if (j + 1 < n && heap[j + 1] > heap[j]) j++;
11     if (heap[i] >= heap[j]) return;
12     yerDegistir(i, j);  i = j;
13   }
14 }
```

**DEĞİŞKENLER**

`n = 7`

`heap = [8, 3, 6, 2, 1, 5, 4]`

# n Öğeden Heap Oluşturma

### Yol 1: n kez ekle

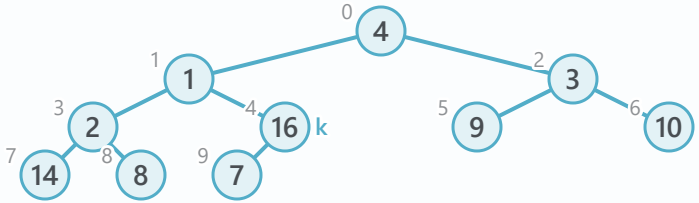
- – Her ekleme up-heap yapar: O(log n).
- – Toplam **O(n log n)**.
- – Öğeler tek tek geliyorsa (çevrim içi) tek seçenektir.

### Yol 2: aşağıdan yukarıya

- + Tüm öğeler baştan elimizdeyse diziyi olduğu gibi al.
- + Son iç düğümden (⌊n/2⌋ − 1) köke doğru her düğüme **down-heap** uygula.
- + Toplam **O(n)**.

```
for (int k = n / 2 - 1; k >= 0; k--)   // son iç düğümden köke
    batir(k);                           // alt ağaçları heap olan k'yi batır
```

# Kod İzleme: `heapOlustur()` (k = 4, 3, 2)




heapOlustur() n = 10

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |

▲ k

Sıradan bir dizi (heap değil). İndis 5…9 yapraktır, tek başına zaten heap. **k = n/2 − 1 = 4** (son iç düğüm) ile başlanır.

Adım 1 / 9

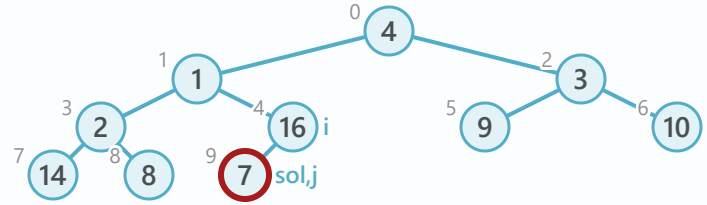
```
1  public void heapOlustur() {      // rastgele dizi
▶    for (int k = n / 2 - 1; k >= 0; k--)
3      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
6    while (2 * i + 1 < n) {        // sol çocuk var mı?
7      int j = 2 * i + 1;             // j: büyük çocuk
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     yerDegistir(i, j);  i = j;     // bir düzey in
11   }
12 }
```

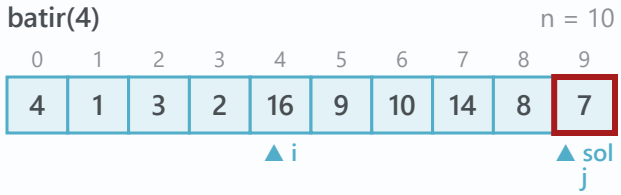
**DEĞİŞKENLER**

`k = 4` `n = 10`

`heap = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]`

# Kod İzleme: `heapOlustur()` (k = 4, 3, 2)




**k = 4** → batir(4). Yalnızca sol çocuk var: indis 9 (7) → **j = 9**.

Adım 2 / 9

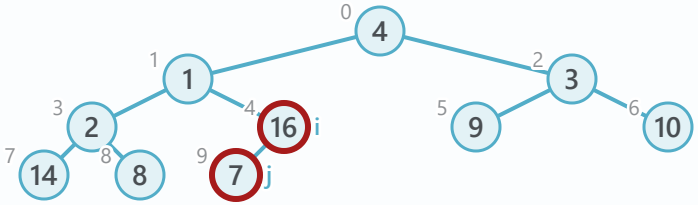
```
1  public void heapOlustur() {      // rastgele dizi
▶    for (int k = n / 2 - 1; k >= 0; k--)
▶      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
▶    while (2 * i + 1 < n) {        // sol çocuk var mı?
▶      int j = 2 * i + 1;             // j: büyük çocuk
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     yerDegistir(i, j);  i = j;     // bir düzey in
11   }
12 }
```

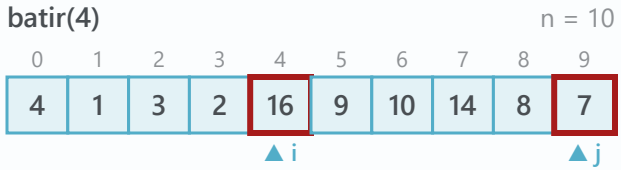
**DEĞİŞKENLER**

`k = 4` `i = 4` `j = 9` `n = 10`

`heap = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]`

# Kod İzleme: `heapOlustur()` (k = 4, 3, 2)




heap[4] = 16 ≥ heap[9] = 7 → iki çocuktan da büyük/eşit, sıra sağlandı: **return**.

Adım 3 / 9

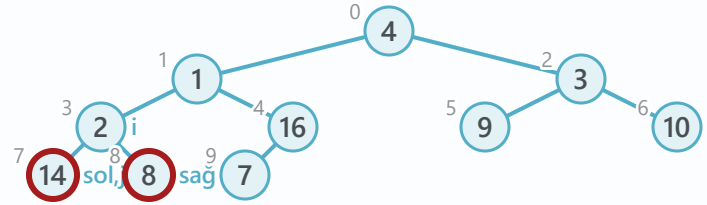
```
1  public void heapOlustur() {      // rastgele dizi
2    for (int k = n / 2 - 1; k >= 0; k--)
3      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
6    while (2 * i + 1 < n) {        // sol çocuk var mı?
7      int j = 2 * i + 1;             // j: büyük çocuk
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
▶      if (heap[i] >= heap[j]) return;
10     yerDegistir(i, j);  i = j;     // bir düzey in
11   }
12 }
```

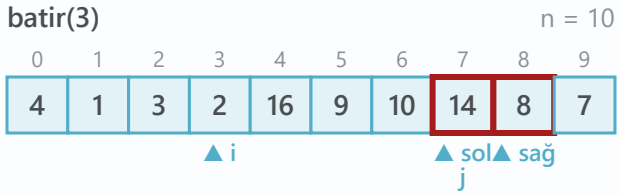
**DEĞİŞKENLER**

`k = 4` `i = 4` `j = 9` `n = 10`

`heap = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]`

# Kod İzleme: `heapOlustur()` (k = 4, 3, 2)




**k = 3** → batir(3). Sol çocuk 7 (14), sağ çocuk 8 (8) → büyük olan **14**, **j = 7**.

Adım 4 / 9

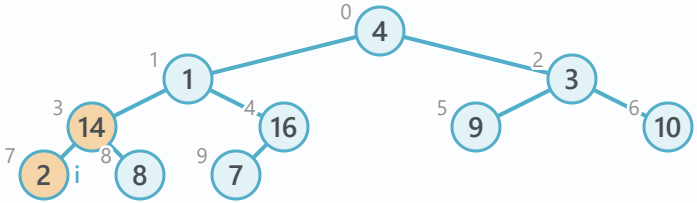
```
1  public void heapOlustur() {      // rastgele dizi
▶    for (int k = n / 2 - 1; k >= 0; k--)
▶      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
▶    while (2 * i + 1 < n) {        // sol çocuk var mı?
▶      int j = 2 * i + 1;             // j: büyük çocuk
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     yerDegistir(i, j);  i = j;     // bir düzey in
11   }
12 }
```

**DEĞİŞKENLER**

`k = 3` `i = 3` `j = 7` `n = 10`

`heap = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]`

# Kod İzleme: `heapOlustur()` (k = 4, 3, 2)




batir(3) n = 10

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 1 | 3 | 14 | 16 | 9 | 10 | 2 | 8 | 7 |
| | | | | | | | ▲ i | | |

2 < 14 → **yer değiştir** (indis 3 ↔ 7); büyük çocuk yukarı çıktı, **i = 7**.

Adım 5 / 9

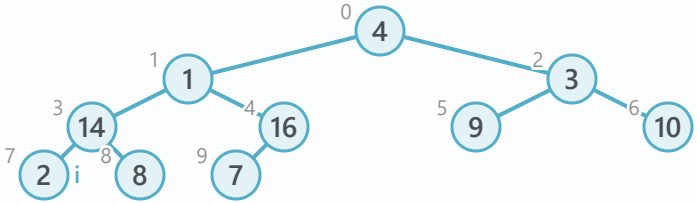
```
1  public void heapOlustur() {      // rastgele dizi
2    for (int k = n / 2 - 1; k >= 0; k--)
3      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
6    while (2 * i + 1 < n) {        // sol çocuk var mı?
7      int j = 2 * i + 1;             // j: büyük çocuk
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
▶      yerDegistir(i, j);  i = j;     // bir düzey in
11   }
12 }
```

**DEĞİŞKENLER**

`k = 3` `i = 7` `j = 7` `n = 10`

`heap = [4, 1, 3, 14, 16, 9, 10, 2, 8, 7]`

# Kod İzleme: `heapOlustur()` (k = 4, 3, 2)




batir(3) n = 10

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 1 | 3 | 14 | 16 | 9 | 10 | 2 | 8 | 7 |
| | | | | | | | ▲ i | | |

i = 7: 2·7 + 1 = 15 ≥ n = 10 → çocuk yok (yaprak), döngü biter.

Adım 6 / 9

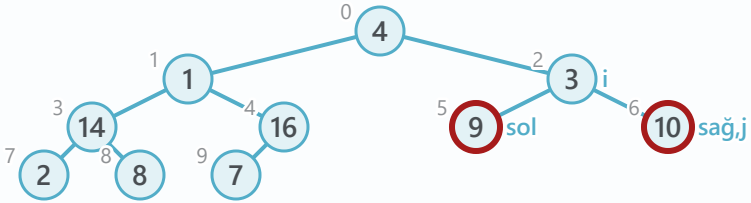
```
1  public void heapOlustur() {      // rastgele dizi
2    for (int k = n / 2 - 1; k >= 0; k--)
3      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
▶    while (2 * i + 1 < n) {        // sol çocuk var mı?
7      int j = 2 * i + 1;             // j: büyük çocuk
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     yerDegistir(i, j);  i = j;     // bir düzey in
11   }
12 }
```

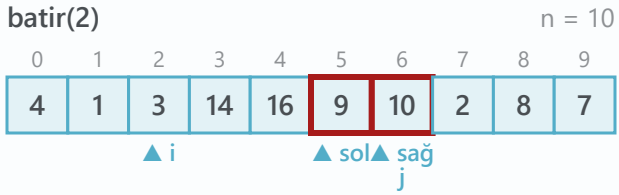
**DEĞİŞKENLER**

`k = 3` `i = 7` `n = 10`

`heap = [4, 1, 3, 14, 16, 9, 10, 2, 8, 7]`

# Kod İzleme: `heapOlustur()` (k = 4, 3, 2)




**k = 2** → batir(2). Sol çocuk 5 (9), sağ çocuk 6 (10) → büyük olan **10**, **j = 6**.

Adım 7 / 9

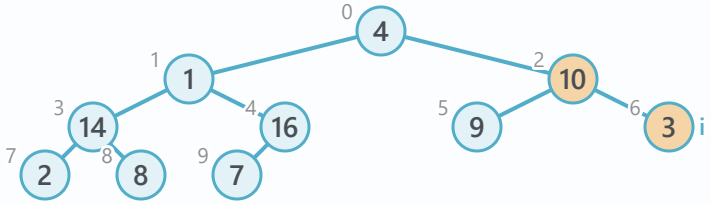
```
1  public void heapOlustur() {      // rastgele dizi
▶    for (int k = n / 2 - 1; k >= 0; k--)
▶      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
▶    while (2 * i + 1 < n) {        // sol çocuk var mı?
▶      int j = 2 * i + 1;             // j: büyük çocuk
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     yerDegistir(i, j);  i = j;     // bir düzey in
11   }
12 }
```

**DEĞİŞKENLER**

`k = 2` `i = 2` `j = 6` `n = 10`

`heap = [4, 1, 3, 14, 16, 9, 10, 2, 8, 7]`

# Kod İzleme: `heapOlustur()` (k = 4, 3, 2)




batir(2) n = 10

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 1 | 10 | 14 | 16 | 9 | 3 | 2 | 8 | 7 |
| | | | | | | ▲ i | | | |

3 < 10 → **yer değiştir** (indis 2 ↔ 6); büyük çocuk yukarı çıktı, **i = 6**.

Adım 8 / 9

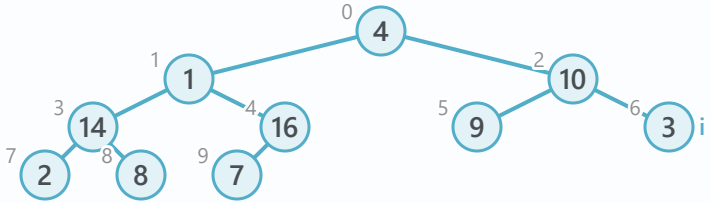
```
1  public void heapOlustur() {      // rastgele dizi
2    for (int k = n / 2 - 1; k >= 0; k--)
3      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
6    while (2 * i + 1 < n) {        // sol çocuk var mı?
7      int j = 2 * i + 1;             // j: büyük çocuk
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
▶      yerDegistir(i, j);  i = j;     // bir düzey in
11   }
12 }
```

**DEĞİŞKENLER**

`k = 2` `i = 6` `j = 6` `n = 10`

`heap = [4, 1, 10, 14, 16, 9, 3, 2, 8, 7]`

# Kod İzleme: `heapOlustur()` (k = 4, 3, 2)




batir(2) n = 10

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 1 | 10 | 14 | 16 | 9 | 3 | 2 | 8 | 7 |

▲ i

i = 6: 2·6 + 1 = 13 ≥ n = 10 → çocuk yok (yaprak), döngü biter.

Adım 9 / 9

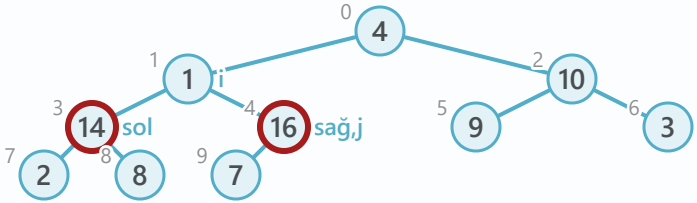
```
1  public void heapOlustur() {      // rastgele dizi
2    for (int k = n / 2 - 1; k >= 0; k--)
3      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {       // down-heap
▶    while (2 * i + 1 < n) {        // sol çocuk var mı?
7      int j = 2 * i + 1;           // j: büyük çocuk
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     yerDegistir(i, j);  i = j;   // bir düzey in
11   }
12 }
```

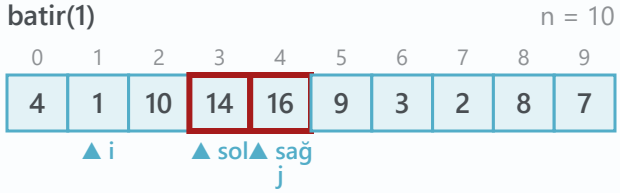
**DEĞİŞKENLER**

`k = 2` `i = 6` `n = 10`

`heap = [4, 1, 10, 14, 16, 9, 3, 2, 8, 7]`

# Kod İzleme (devam): k = 1, 0




**k = 1** → batir(1). Sol çocuk 3 (14), sağ çocuk 4 (16) → büyük olan **16**, **j = 4**.

Adım 1 / 13

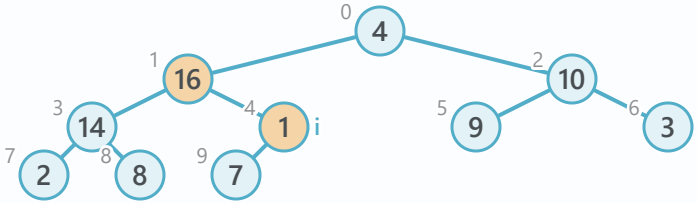
```
1  public void heapOlustur() {      // rastgele dizi
▶    for (int k = n / 2 - 1; k >= 0; k--)
▶      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
▶    while (2 * i + 1 < n) {        // sol çocuk var mı?
▶      int j = 2 * i + 1;           // j: büyük çocuk
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     yerDegistir(i, j);  i = j;    // bir düzey in
11   }
12 }
```

**DEĞİŞKENLER**

`k = 1` `i = 1` `j = 4` `n = 10`

`heap = [4, 1, 10, 14, 16, 9, 3, 2, 8, 7]`

# Kod İzleme (devam): k = 1, 0




batir(1) n = 10

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 16 | 10 | 14 | 1 | 9 | 3 | 2 | 8 | 7 |
| | | | | ▲ i | | | | | |

1 < 16 → **yer değiştir** (indis 1 ↔ 4); büyük çocuk yukarı çıktı, **i = 4**.

Adım 2 / 13

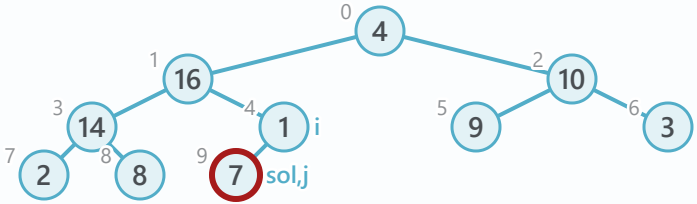
```
1  public void heapOlustur() {      // rastgele dizi
2    for (int k = n / 2 - 1; k >= 0; k--)
3      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
6    while (2 * i + 1 < n) {        // sol çocuk var mı?
7      int j = 2 * i + 1;             // j: büyük çocuk
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
▶      yerDegistir(i, j);  i = j;     // bir düzey in
11   }
12 }
```

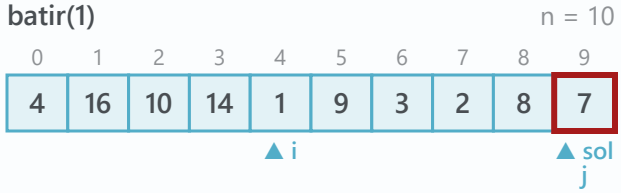
**DEĞİŞKENLER**

`k = 1` `i = 4` `j = 4` `n = 10`

`heap = [4, 16, 10, 14, 1, 9, 3, 2, 8, 7]`

# Kod İzleme (devam): k = 1, 0




Yalnızca sol çocuk var: indis 9 (7) → **j = 9**.



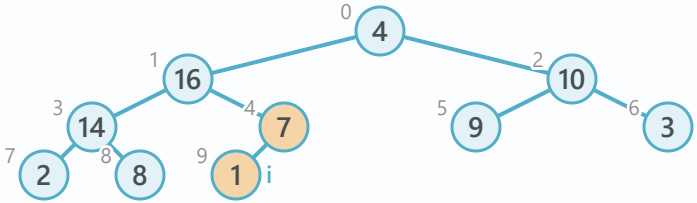
```
1  public void heapOlustur() {      // rastgele dizi
2    for (int k = n / 2 - 1; k >= 0; k--)
3      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
▶    while (2 * i + 1 < n) {        // sol çocuk var mı?
▶      int j = 2 * i + 1;           // j: büyük çocuk
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     yerDegistir(i, j);  i = j;    // bir düzey in
11   }
12 }
```

**DEĞİŞKENLER**

`k = 1` `i = 4` `j = 9` `n = 10`

`heap = [4, 16, 10, 14, 1, 9, 3, 2, 8, 7]`

# Kod İzleme (devam): k = 1, 0




batir(1) n = 10

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 16 | 10 | 14 | 7 | 9 | 3 | 2 | 8 | 1 |
| | | | | | | | | | ▲ i |

1 < 7 → **yer değiştir** (indis 4 ↔ 9); büyük çocuk yukarı çıktı, **i = 9**.

Adım 4 / 13

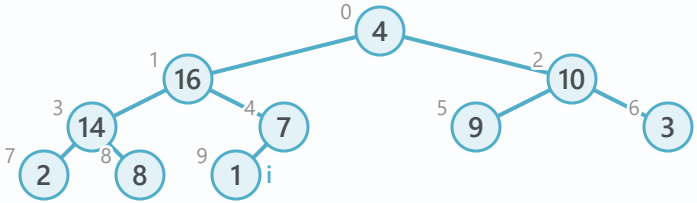
```
1  public void heapOlustur() {      // rastgele dizi
2    for (int k = n / 2 - 1; k >= 0; k--)
3      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
6    while (2 * i + 1 < n) {        // sol çocuk var mı?
7      int j = 2 * i + 1;             // j: büyük çocuk
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
▶      yerDegistir(i, j);  i = j;     // bir düzey in
11   }
12 }
```

**DEĞİŞKENLER**

`k = 1` `i = 9` `j = 9` `n = 10`

`heap = [4, 16, 10, 14, 7, 9, 3, 2, 8, 1]`

# Kod İzleme (devam): k = 1, 0




batir(1) n = 10

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 4 | 16 | 10 | 14 | 7 | 9 | 3 | 2 | 8 | 1 |
| | | | | | | | | | ▲ i |

i = 9: 2·9 + 1 = 19 ≥ n = 10 → çocuk yok (yaprak), döngü biter.

Adım 5 / 13

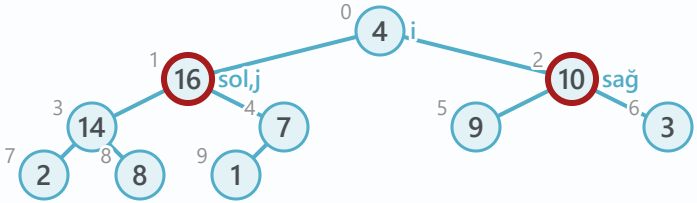
```
1  public void heapOlustur() {      // rastgele dizi
2    for (int k = n / 2 - 1; k >= 0; k--)
3      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
▶    while (2 * i + 1 < n) {        // sol çocuk var mı?
7      int j = 2 * i + 1;             // j: büyük çocuk
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     yerDegistir(i, j);  i = j;     // bir düzey in
11   }
12 }
```

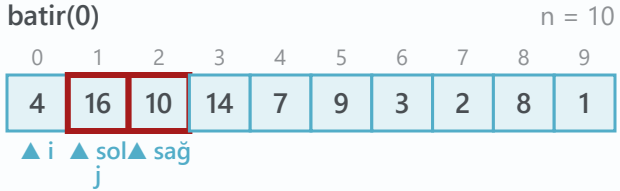
**DEĞİŞKENLER**

`k = 1` `i = 9` `n = 10`

`heap = [4, 16, 10, 14, 7, 9, 3, 2, 8, 1]`

# Kod İzleme (devam): k = 1, 0




**k = 0** → batir(0). Sol çocuk 1 (16), sağ çocuk 2 (10) → büyük olan **16**, **j = 1**.

Adım 6 / 13

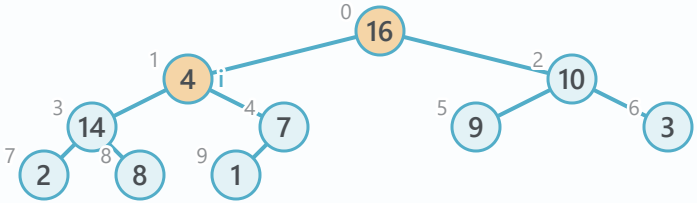
```
1  public void heapOlustur() {      // rastgele dizi
▶    for (int k = n / 2 - 1; k >= 0; k--)
▶      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
▶    while (2 * i + 1 < n) {        // sol çocuk var mı?
▶      int j = 2 * i + 1;           // j: büyük çocuk
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     yerDegistir(i, j);  i = j;    // bir düzey in
11   }
12 }
```

**DEĞİŞKENLER**

k = 0 | i = 0 | j = 1 | n = 10

heap = [4, 16, 10, 14, 7, 9, 3, 2, 8, 1]

# Kod İzleme (devam): k = 1, 0




batir(0) n = 10

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 16 | 4 | 10 | 14 | 7 | 9 | 3 | 2 | 8 | 1 |
| | ▲ i | | | | | | | | |

4 < 16 → **yer değiştir** (indis 0 ↔ 1); büyük çocuk yukarı çıktı, **i = 1**.

Adım 7 / 13

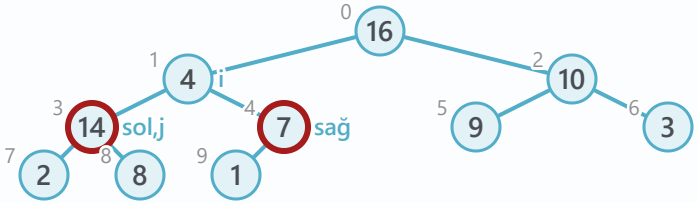
```
1  public void heapOlustur() {      // rastgele dizi
2    for (int k = n / 2 - 1; k >= 0; k--)
3      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
6    while (2 * i + 1 < n) {        // sol çocuk var mı?
7      int j = 2 * i + 1;             // j: büyük çocuk
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
▶      yerDegistir(i, j);  i = j;     // bir düzey in
11   }
12 }
```

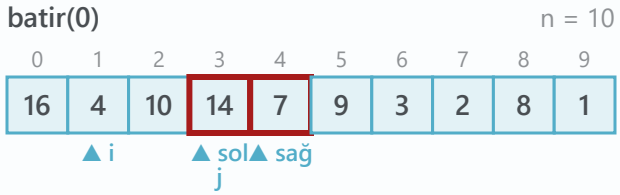
**DEĞİŞKENLER**

`k = 0` `i = 1` `j = 1` `n = 10`

`heap = [16, 4, 10, 14, 7, 9, 3, 2, 8, 1]`

# Kod İzleme (devam): k = 1, 0




Sol çocuk 3 (14), sağ çocuk 4 (7) → büyük olan **14**, **j = 3**.

Adım 8 / 13

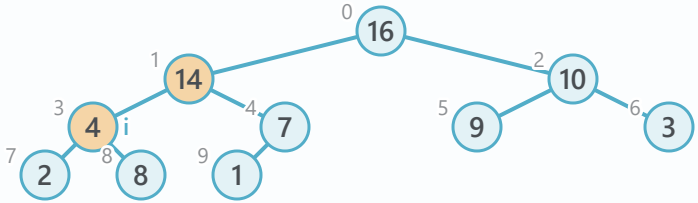
```
1  public void heapOlustur() {      // rastgele dizi
2    for (int k = n / 2 - 1; k >= 0; k--)
3      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
▶    while (2 * i + 1 < n) {        // sol çocuk var mı?
▶      int j = 2 * i + 1;             // j: büyük çocuk
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     yerDegistir(i, j);  i = j;     // bir düzey in
11   }
12 }
```

**DEĞİŞKENLER**

`k = 0` `i = 1` `j = 3` `n = 10`

`heap = [16, 4, 10, 14, 7, 9, 3, 2, 8, 1]`

# Kod İzleme (devam): k = 1, 0




batir(0) n = 10

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 16 | 14 | 10 | 4 | 7 | 9 | 3 | 2 | 8 | 1 |
| | | | ▲ i | | | | | | |

4 < 14 → **yer değiştir** (indis 1 ↔ 3); büyük çocuk yukarı çıktı, **i = 3**.

Adım 9 / 13

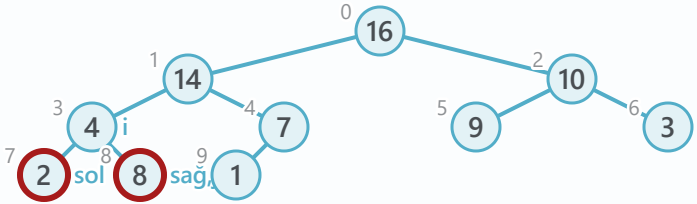
```
1  public void heapOlustur() {      // rastgele dizi
2    for (int k = n / 2 - 1; k >= 0; k--)
3      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
6    while (2 * i + 1 < n) {        // sol çocuk var mı?
7      int j = 2 * i + 1;             // j: büyük çocuk
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
▶      yerDegistir(i, j);  i = j;     // bir düzey in
11   }
12 }
```

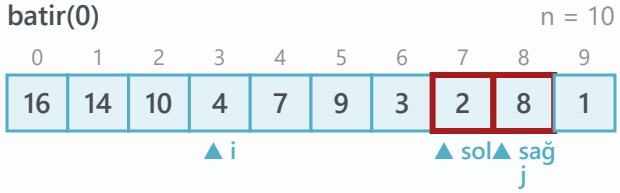
**DEĞİŞKENLER**

`k = 0` `i = 3` `j = 3` `n = 10`

`heap = [16, 14, 10, 4, 7, 9, 3, 2, 8, 1]`

# Kod İzleme (devam): k = 1, 0




Sol çocuk 7 (2), sağ çocuk 8 (8) → büyük olan **8**, **j = 8**.

Adım 10 / 13

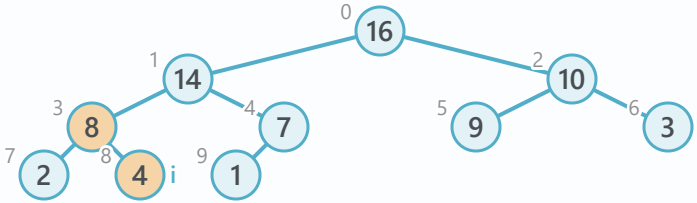
```
1  public void heapOlustur() {      // rastgele dizi
2    for (int k = n / 2 - 1; k >= 0; k--)
3      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
▶    while (2 * i + 1 < n) {        // sol çocuk var mı?
▶      int j = 2 * i + 1;           // j: büyük çocuk
▶      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     yerDegistir(i, j);  i = j;     // bir düzey in
11   }
12 }
```

**DEĞİŞKENLER**

`k = 0` `i = 3` `j = 8` `n = 10`

`heap = [16, 14, 10, 4, 7, 9, 3, 2, 8, 1]`

# Kod İzleme (devam): k = 1, 0




batir(0) n = 10

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |
| | | | | | | | | ▲ i | |

4 < 8 → **yer değiştir** (indis 3 ↔ 8); büyük çocuk yukarı çıktı, **i = 8**.

Adım 11 / 13

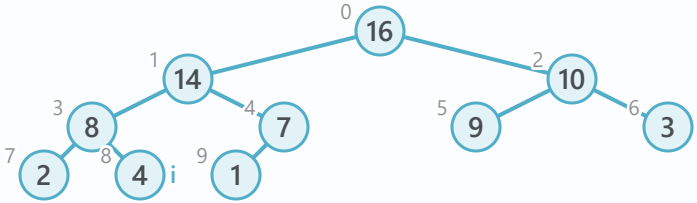
```
1  public void heapOlustur() {      // rastgele dizi
2    for (int k = n / 2 - 1; k >= 0; k--)
3      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
6    while (2 * i + 1 < n) {        // sol çocuk var mı?
7      int j = 2 * i + 1;             // j: büyük çocuk
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
▶      yerDegistir(i, j);  i = j;     // bir düzey in
11   }
12 }
```

**DEĞİŞKENLER**

k = 0 | i = 8 | j = 8 | n = 10

heap = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]

# Kod İzleme (devam): k = 1, 0




batir(0) n = 10

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |
| | | | | | | | | ▲ i | |

i = 8: 2·8 + 1 = 17 ≥ n = 10 → çocuk yok (yaprak), döngü biter.

Adım 12 / 13

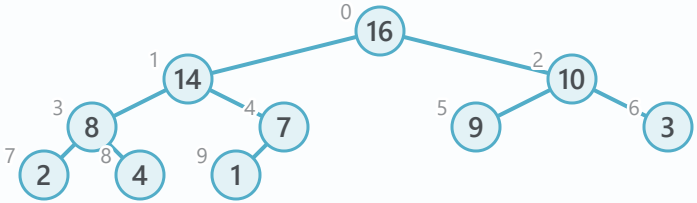
```
1  public void heapOlustur() {      // rastgele dizi
2    for (int k = n / 2 - 1; k >= 0; k--)
3      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
▶    while (2 * i + 1 < n) {        // sol çocuk var mı?
7      int j = 2 * i + 1;             // j: büyük çocuk
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     yerDegistir(i, j);  i = j;     // bir düzey in
11   }
12 }
```

**DEĞİŞKENLER**

`k = 0` `i = 8` `n = 10`

`heap = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]`

# Kod İzleme (devam): k = 1, 0




heapOlustur() n = 10

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |

k = −1 → döngü biter. Dizi artık **max-heap**: her ebeveyn ≥ çocukları, en büyük (16) kökte. Toplam iş O(n).

Adım 13 / 13

```
1  public void heapOlustur() {      // rastgele dizi
▶    for (int k = n / 2 - 1; k >= 0; k--)
3      batir(k);                    // alt ağaçlar heap
4  }
5  private void batir(int i) {        // down-heap
6    while (2 * i + 1 < n) {        // sol çocuk var mı?
7      int j = 2 * i + 1;             // j: büyük çocuk
8      if (j + 1 < n && heap[j + 1] > heap[j]) j++;
9      if (heap[i] >= heap[j]) return;
10     yerDegistir(i, j);  i = j;     // bir düzey in
11   }
12 }
```

**DEĞİŞKENLER**

k = −1 | n = 10

heap = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]

# Aşağıdan Yukarıya Oluşturma Neden O(n)?

| Düğümün yüksekliği | Düğüm | En çok takas | Toplam |
|---|---|---|---|
| 0 (yapraklar) | 8 | 0 | 0 |
| 1 | 4 | 1 | 4 |
| 2 | 2 | 2 | 4 |
| 3 (kök) | 1 | 3 | 3 |
| **n = 15** | | | **11** |

- Düğümlerin yarısı yapraktır ve hiç iş yapmaz; çok iş yapan (yüksek) düğüm sayısı azdır.
- Genel olarak: $\Sigma_h \, (n / 2^{h+1}) \cdot h \leq n \rightarrow$ **O(n)**.
- n kez eklemede ise çok sayıdaki derin düğüm uzun yol yürür: aynı ağaçta en kötü 0 + 2·1 + 4·2 + 8·3 = **34** takas.

Down-heap'in maliyeti düğümün *yüksekliğine*, up-heap'in maliyeti *derinliğine* bağlıdır. Düğümlerin çoğu aşağıdadır.

# Heap Sort

**Aşama 1: heap oluştur**

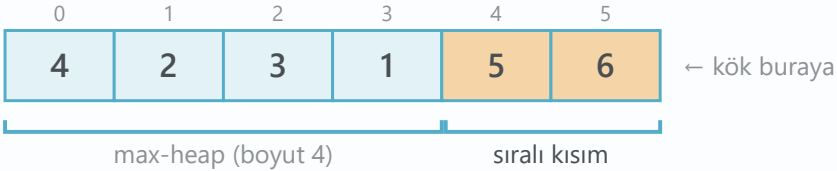
Dizi aşağıdan yukarıya max-heap'e çevrilir.

O(n)

**Aşama 2: n − 1 kez silMax**

Kök (en büyük) dizinin sonundaki öğeyle takas edilir, heap bir küçülür, kök için down-heap.

O(n log n)




- Dizinin sonunda sıralı kısım büyür, başında heap küçülür; **ek dizi gerekmez** (yerinde, O(1) ek bellek).
- En kötü durumda bile **O(n log n)**: hızlı sıralamanın (quick sort) $O(n^2)$ kötü durumu yoktur.
- **Kararlı değildir** (stable değil): eşit anahtarların göreli sırası değişebilir.
- Max-heap artan sıralama, min-heap azalan sıralama üretir.

# Heap Sort: Java Kodu

```java
public static void heapSort(int[] a) {
  int n = a.length;
  // 1) oluştur: son iç düğümden köke
  for (int k = n / 2 - 1; k >= 0; k--)
    batir(a, k, n);
  // 2) sırala: en büyüğü sona al
  for (int son = n - 1; son > 0; son--) {
    yerDegistir(a, 0, son);
    batir(a, 0, son);            // heap boyutu = son
  }
}
static void batir(int[] a, int i, int n) {
  while (2 * i + 1 < n) {
    int j = 2 * i + 1;
    if (j + 1 < n && a[j + 1] > a[j]) j++;
    if (a[i] >= a[j]) return;
    yerDegistir(a, i, j);  i = j;
  }
}
```

### Aynı batir

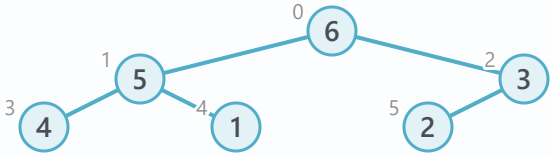
MaxHeap sınıfındakiyle aynı; yalnızca dizi ve heap boyutu parametre olarak geçer.

### Deneme

`{4, 1, 3, 2, 16, 9, 10, 14, 8, 7}`

→ `{1, 2, 3, 4, 7, 8, 9, 10, 14, 16}`

# Kod İzleme: `heapSort([6, 5, 3, 4, 1, 2])`




heapSort(a) heap boyutu = 6

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 6 | 5 | 3 | 4 | 1 | 2 |

Girdi a = [6, 5, 3, 4, 1, 2]. Tüm dizi heap bölgesi; sağda sıralı kısım henüz boş.

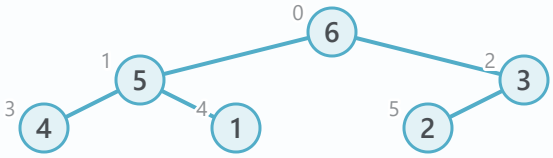
Adım 1 / 15

```
1  public static void heapSort(int[] a) {
▶    int n = a.length;
3    // 1) oluştur: son iç düğümden köke
4    for (int k = n / 2 - 1; k >= 0; k--)
5      batir(a, k, n);
6    // 2) sırala: en büyüğü sona al
7    for (int son = n - 1; son > 0; son--) {
8      yerDegistir(a, 0, son);
9      batir(a, 0, son);            // heap boyutu = son
10   }
11 }
```

**DEĞİŞKENLER**

`n = 6` `a = [6, 5, 3, 4, 1, 2]`

# Kod İzleme: `heapSort([6, 5, 3, 4, 1, 2])`




**1) oluştur** heap boyutu = 6

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 6 | 5 | 3 | 4 | 1 | 2 |

Aşama 1: k = 2, 1, 0 için batir çağrılır. Bu dizi zaten max-heap olduğundan her çağrı ilk karşılaştırmada döner; takas yok.

Adım 2 / 15

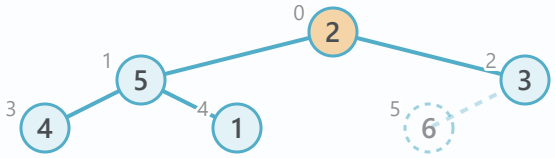
```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) oluştur: son iç düğümden köke
▶    for (int k = n / 2 - 1; k >= 0; k--)
▶      batir(a, k, n);
6    // 2) sırala: en büyüğü sona al
7    for (int son = n - 1; son > 0; son--) {
8      yerDegistir(a, 0, son);
9      batir(a, 0, son);            // heap boyutu = son
10   }
11 }
```

**DEĞİŞKENLER**

`n = 6` `k = 0`

`a = [6, 5, 3, 4, 1, 2]`

# Kod İzleme: `heapSort([6, 5, 3, 4, 1, 2])`




son = 5

heap boyutu = 5

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 2 | 5 | 3 | 4 | 1 | 6 |

soluk: sıralı kısım

**son = 5**: kök (en büyük, 6) ile a[5] = 2 yer değiştirir → 6 kesin yerine oturdu, heap boyutu 5.

Adım 3 / 15

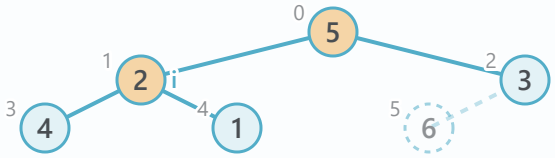
```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) oluştur: son iç düğümden köke
4    for (int k = n / 2 - 1; k >= 0; k--)
5      batir(a, k, n);
6    // 2) sırala: en büyüğü sona al
▶    for (int son = n - 1; son > 0; son--) {
▶      yerDegistir(a, 0, son);
9      batir(a, 0, son);            // heap boyutu = son
10   }
11 }
```

**DEĞİŞKENLER**

`n = 6` `son = 5`

`a = [2, 5, 3, 4, 1 | 6]`

# Kod İzleme: `heapSort([6, 5, 3, 4, 1, 2])`




batir(a, 0, 5)

heap boyutu = 5

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 5 | 2 | 3 | 4 | 1 | 6 |
| | ▲ i | | | | |

soluk: sıralı kısım

batir(a, 0, 5): i = 0 için büyük çocuk a[1] = 5 > 2 → yer değiştir (0 ↔ 1).

Adım 4 / 15

```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) oluştur: son iç düğümden köke
4    for (int k = n / 2 - 1; k >= 0; k--)
5      batir(a, k, n);
6    // 2) sırala: en büyüğü sona al
7    for (int son = n - 1; son > 0; son--) {
8      yerDegistir(a, 0, son);
▶      batir(a, 0, son);            // heap boyutu = son
10   }
11 }
```

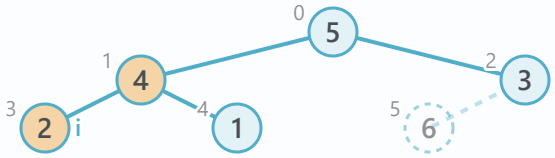
**DEĞİŞKENLER**

`n = 6` `son = 5`

`batir: i = 1, j=1`

`a = [5, 2, 3, 4, 1 | 6]`

# Kod İzleme: `heapSort([6, 5, 3, 4, 1, 2])`




batir(a, 0, 5)

heap boyutu = 5

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 5 | 4 | 3 | 2 | 1 | 6 |
| | | | ▲ i | | |

soluk: sıralı kısım

batir(a, 0, 5): i = 1 için büyük çocuk a[3] = 4 > 2 → yer değiştir (1 ↔ 3). İndis 3 yaprak → batir biter.

Adım 5 / 15

```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) oluştur: son iç düğümden köke
4    for (int k = n / 2 - 1; k >= 0; k--)
5      batir(a, k, n);
6    // 2) sırala: en büyüğü sona al
7    for (int son = n - 1; son > 0; son--) {
8      yerDegistir(a, 0, son);
▶      batir(a, 0, son);            // heap boyutu = son
10   }
11 }
```

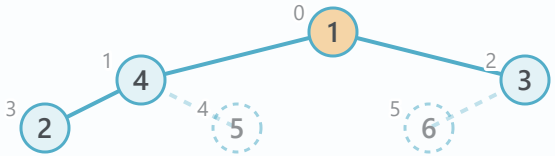
**DEĞİŞKENLER**

`n = 6` `son = 5`

`batir: i = 3, j=3`

`a = [5, 4, 3, 2, 1 | 6]`

# Kod İzleme: `heapSort([6, 5, 3, 4, 1, 2])`




son = 4

heap boyutu = 4

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 4 | 3 | 2 | 5 | 6 |

soluk: sıralı kısım

**son = 4**: kök (en büyük, 5) ile a[4] = 1 yer değiştirir → 5 kesin yerine oturdu, heap boyutu 4.

Adım 6 / 15

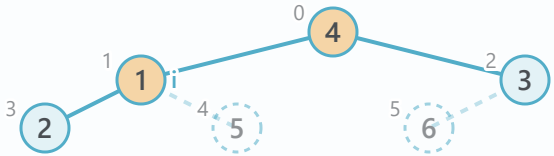
```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) oluştur: son iç düğümden köke
4    for (int k = n / 2 - 1; k >= 0; k--)
5      batir(a, k, n);
6    // 2) sırala: en büyüğü sona al
▶    for (int son = n - 1; son > 0; son--) {
▶      yerDegistir(a, 0, son);
9      batir(a, 0, son);            // heap boyutu = son
10   }
11 }
```

**DEĞİŞKENLER**

`n = 6` `son = 4`

`a = [1, 4, 3, 2 | 5, 6]`

# Kod İzleme: `heapSort([6, 5, 3, 4, 1, 2])`




batir(a, 0, 4)

heap boyutu = 4

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 4 | 1 | 3 | 2 | 5 | 6 |
| | ▲ i | | | | |

soluk: sıralı kısım

batir(a, 0, 4): i = 0 için büyük çocuk a[1] = 4 > 1 → yer değiştir (0 ↔ 1).

Adım 7 / 15

```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) oluştur: son iç düğümden köke
4    for (int k = n / 2 - 1; k >= 0; k--)
5      batir(a, k, n);
6    // 2) sırala: en büyüğü sona al
7    for (int son = n - 1; son > 0; son--) {
8      yerDegistir(a, 0, son);
▶      batir(a, 0, son);            // heap boyutu = son
10   }
11 }
```

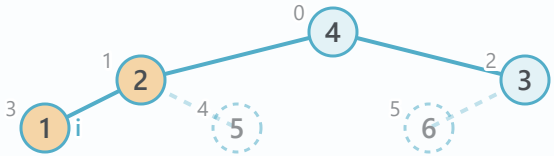
**DEĞİŞKENLER**

`n = 6` `son = 4`

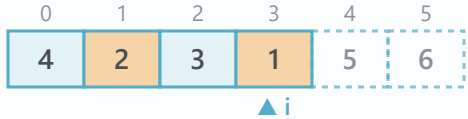
`batir: i = 1, j=1`

`a = [4, 1, 3, 2 | 5, 6]`

# Kod İzleme: `heapSort([6, 5, 3, 4, 1, 2])`




batir(a, 0, 4) — heap boyutu = 4




soluk: sıralı kısım

batir(a, 0, 4): i = 1 için büyük çocuk a[3] = 2 > 1 → yer değiştir (1 ↔ 3). İndis 3 yaprak → batir biter.

Adım 8 / 15

```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) oluştur: son iç düğümden köke
4    for (int k = n / 2 - 1; k >= 0; k--)
5      batir(a, k, n);
6    // 2) sırala: en büyüğü sona al
7    for (int son = n - 1; son > 0; son--) {
8      yerDegistir(a, 0, son);
▶      batir(a, 0, son);            // heap boyutu = son
10   }
11 }
```

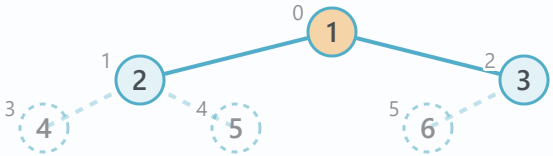
**DEĞİŞKENLER**

`n = 6` `son = 4`

`batir: i = 3, j=3`

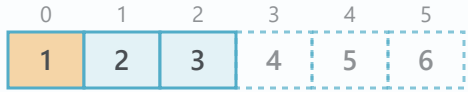
`a = [4, 2, 3, 1 | 5, 6]`

# Kod İzleme: `heapSort([6, 5, 3, 4, 1, 2])`




son = 3

heap boyutu = 3




soluk: sıralı kısım

**son = 3**: kök (en büyük, 4) ile a[3] = 1 yer değiştirir → 4 kesin yerine oturdu, heap boyutu 3.

Adım 9 / 15

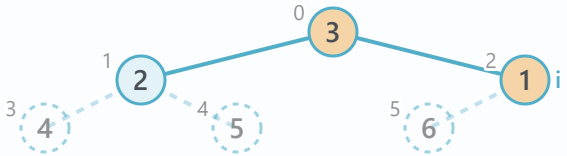
```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) oluştur: son iç düğümden köke
4    for (int k = n / 2 - 1; k >= 0; k--)
5      batir(a, k, n);
6    // 2) sırala: en büyüğü sona al
▶    for (int son = n - 1; son > 0; son--) {
▶      yerDegistir(a, 0, son);
9      batir(a, 0, son);            // heap boyutu = son
10   }
11 }
```

**DEĞİŞKENLER**

`n = 6` `son = 3`

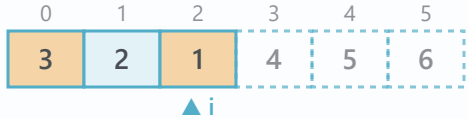
`a = [1, 2, 3 | 4, 5, 6]`

# Kod İzleme: `heapSort([6, 5, 3, 4, 1, 2])`




batir(a, 0, 3)

heap boyutu = 3




soluk: sıralı kısım

batir(a, 0, 3): i = 0 için büyük çocuk a[2] = 3 > 1 → yer değiştir (0 ↔ 2). İndis 2 yaprak → batir biter.

Adım 10 / 15

```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) oluştur: son iç düğümden köke
4    for (int k = n / 2 - 1; k >= 0; k--)
5      batir(a, k, n);
6    // 2) sırala: en büyüğü sona al
7    for (int son = n - 1; son > 0; son--) {
8      yerDegistir(a, 0, son);
▶      batir(a, 0, son);            // heap boyutu = son
10   }
11 }
```

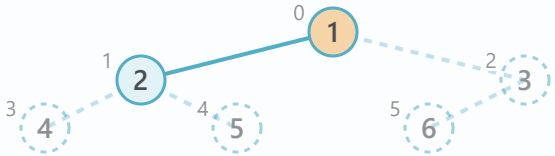
**DEĞİŞKENLER**

`n = 6` `son = 3`

`batir: i = 2, j=2`

`a = [3, 2, 1 | 4, 5, 6]`

# Kod İzleme: `heapSort([6, 5, 3, 4, 1, 2])`




son = 2

heap boyutu = 2




soluk: sıralı kısım

**son = 2**: kök (en büyük, 3) ile a[2] = 1 yer değiştirir → 3 kesin yerine oturdu, heap boyutu 2.

Adım 11 / 15

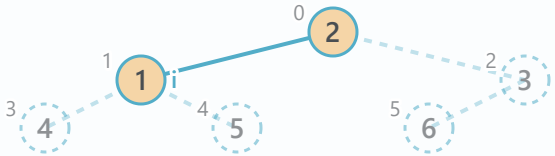
```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) oluştur: son iç düğümden köke
4    for (int k = n / 2 - 1; k >= 0; k--)
5      batir(a, k, n);
6    // 2) sırala: en büyüğü sona al
▶    for (int son = n - 1; son > 0; son--) {
▶      yerDegistir(a, 0, son);
9      batir(a, 0, son);            // heap boyutu = son
10   }
11 }
```

**DEĞİŞKENLER**

`n = 6` `son = 2`

`a = [1, 2 | 3, 4, 5, 6]`

# Kod İzleme: `heapSort([6, 5, 3, 4, 1, 2])`




batir(a, 0, 2)

heap boyutu = 2

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 2 | 1 | 3 | 4 | 5 | 6 |
| | ▲ i | | | | |

soluk: sıralı kısım

batir(a, 0, 2): i = 0 için büyük çocuk a[1] = 2 > 1 → yer değiştir (0 ↔ 1). İndis 1 yaprak → batir biter.

Adım 12 / 15

```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) oluştur: son iç düğümden köke
4    for (int k = n / 2 - 1; k >= 0; k--)
5      batir(a, k, n);
6    // 2) sırala: en büyüğü sona al
7    for (int son = n - 1; son > 0; son--) {
8      yerDegistir(a, 0, son);
▶      batir(a, 0, son);          // heap boyutu = son
10   }
11 }
```

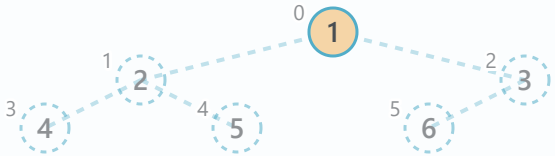
**DEĞİŞKENLER**

`n = 6` `son = 2`

`batir: i = 1, j=1`

`a = [2, 1 | 3, 4, 5, 6]`

# Kod İzleme: `heapSort([6, 5, 3, 4, 1, 2])`




son = 1

heap boyutu = 1




soluk: sıralı kısım

**son = 1**: kök (en büyük, 2) ile a[1] = 1 yer değiştirir → 2 kesin yerine oturdu, heap boyutu 1.

Adım 13 / 15

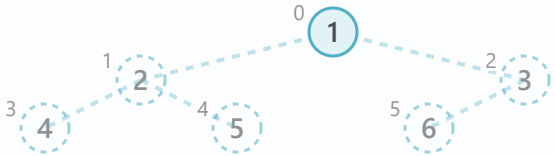
```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) oluştur: son iç düğümden köke
4    for (int k = n / 2 - 1; k >= 0; k--)
5      batir(a, k, n);
6    // 2) sırala: en büyüğü sona al
▶    for (int son = n - 1; son > 0; son--) {
▶      yerDegistir(a, 0, son);
9      batir(a, 0, son);            // heap boyutu = son
10   }
11 }
```

**DEĞİŞKENLER**

`n = 6` `son = 1`

`a = [1 | 2, 3, 4, 5, 6]`

# Kod İzleme: `heapSort([6, 5, 3, 4, 1, 2])`




batir(a, 0, 1)

heap boyutu = 1

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |

soluk: sıralı kısım

batir(a, 0, 1): heap boyutu 1, kökün çocuğu yok → hiçbir şey yapılmaz.

Adım 14 / 15

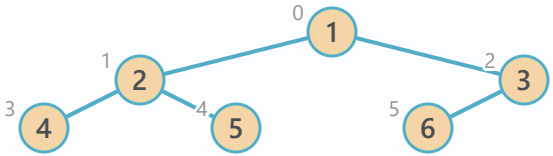
```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) oluştur: son iç düğümden köke
4    for (int k = n / 2 - 1; k >= 0; k--)
5      batir(a, k, n);
6    // 2) sırala: en büyüğü sona al
7    for (int son = n - 1; son > 0; son--) {
8      yerDegistir(a, 0, son);
▶      batir(a, 0, son);            // heap boyutu = son
10   }
11 }
```

**DEĞİŞKENLER**

`n = 6` `son = 1`

`a = [1 | 2, 3, 4, 5, 6]`

# Kod İzleme: `heapSort([6, 5, 3, 4, 1, 2])`




sıralı

heap boyutu = 0

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |

son = 0 → döngü biter. Dizi **artan sırada**: [1, 2, 3, 4, 5, 6]. Ek dizi kullanılmadı (yerinde), toplam **O(n log n)**.

Adım 15 / 15

```
1  public static void heapSort(int[] a) {
2    int n = a.length;
3    // 1) oluştur: son iç düğümden köke
4    for (int k = n / 2 - 1; k >= 0; k--)
5      batir(a, k, n);
6    // 2) sırala: en büyüğü sona al
▶    for (int son = n - 1; son > 0; son--) {
8      yerDegistir(a, 0, son);
9      batir(a, 0, son);            // heap boyutu = son
10   }
11 }
```

**DEĞİŞKENLER**

`n = 6` `son = 0`

`a = [1, 2, 3, 4, 5, 6]`

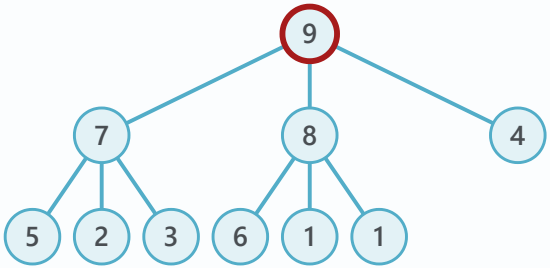
# Öncelikli Kuyrukla Sıralama

Genel fikir: tüm öğeleri öncelikli kuyruğa ekle, sonra n kez `removeMin` ile sırayla çıkar. Hangi sıralamayı elde ettiğimiz gerçeklemeye bağlıdır:

| Gerçekleme | n × insert | n × removeMin | Ortaya çıkan algoritma |
|---|---|---|---|
| **Sırasız liste** | O(n) | $O(n^2)$ | Seçimli sıralama (selection sort), $O(n^2)$ |
| **Sıralı liste** | $O(n^2)$ | O(n) | Eklemeli sıralama (insertion sort), $O(n^2)$ |
| **İkili heap** | O(n log n) | O(n log n) | Heap sort, **O(n log n)** |

Aynı ADT, farklı veri yapısı: çalışma süresi $O(n^2)$'den O(n log n)'e iner. ADT ile veri yapısı ayrımının somut bir örneği.

# Heap Türleri: d'li, Binom, Fibonacci




3'lü max-heap: çocuklar 3i+1, 3i+2, 3i+3

- **d'li heap**: her düğümün d çocuğu; ebeveyn (i − 1) / d. Yükseklik $\log_d n$ → ekleme hızlanır, silmede her düzeyde d çocuk karşılaştırılır.
- **Binom / Fibonacci heap**: ağaç ormanı; iki kuyruğu *birleştirme* ve anahtar küçültme hızlıdır (Dijkstra, Prim).

| Min-heap türü | insert | removeMin | anahtar küçült | birleştir |
|---|---|---|---|---|
| İkili heap | O(log n) | O(log n) | O(log n) | O(n) |
| d'li heap | $O(\log_d n)$ | $O(d \log_d n)$ | $O(\log_d n)$ | O(n) |
| Binom heap | O(log n) | O(log n) | O(log n) | O(log n) |
| Fibonacci heap | O(1) | O(log n)* | O(1)* | O(1) |

* amortize. Uygulamada ikili heap, basitliği ve önbellek dostu dizi yapısı nedeniyle çoğu zaman en hızlısıdır.

# Java'da Hazır Sınıf: PriorityQueue

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.offer(5); pq.offer(1); pq.offer(8);
pq.peek();   // 1  (min-heap, O(1))
pq.poll();   // 1  (O(log n))

// max-heap için ters karşılaştırıcı
PriorityQueue<Integer> maxPq =
    new PriorityQueue<>(Comparator.reverseOrder());

// hasta: öncelik küçük olan önce
PriorityQueue<Hasta> acil = new PriorityQueue<>(
    Comparator.comparingInt(h -> h.oncelik));
```

- İçeride dizi tabanlı **ikili min-heap** kullanır.
- `offer` / `poll` O(log n); `peek`, `size` O(1); `remove(nesne)` ve `contains` O(n).
- **Thread-safe değildir**; çok iş parçacıklı kullanımda `PriorityBlockingQueue` tercih edilir.
- Eşit öncelikte FIFO garantisi yoktur.

# Kod İzleme: `PriorityQueue` içeride ne yapar?

new PriorityQueue

size = 0




Boş bir PriorityQueue: içeride bir dizi ( `queue` ) üzerinde **ikili min-heap** tutar; kökte en küçük öğe.

Adım 1 / 11

```
PriorityQueue<Integer> pq =
    new PriorityQueue<>();        // min-heap
pq.offer(5);
pq.offer(1);
pq.offer(8);
System.out.println(pq.peek());    // bakar
System.out.println(pq.poll());    // çıkarır
System.out.println(pq.poll());
System.out.println(pq.size());
```

**DEĞİŞKENLER**

`pq.size() = 0` `queue = []`

**ÇIKTI**

# Kod İzleme: `PriorityQueue` içeride ne yapar?




offer(5)

size = 1




**offer(5)**: heap boştu; 5 köke (indis 0) yerleşir.

Adım 2 / 11

```
1 PriorityQueue<Integer> pq =
2     new PriorityQueue<>();        // min-heap
▶ pq.offer(5);
4 pq.offer(1);
5 pq.offer(8);
6 System.out.println(pq.peek());    // bakar
7 System.out.println(pq.poll());    // çıkarır
8 System.out.println(pq.poll());
9 System.out.println(pq.size());
```

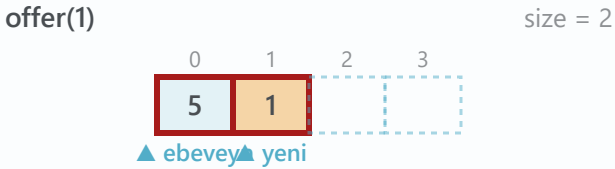
**DEĞİŞKENLER**

`pq.size() = 1` `queue = [5]`

**ÇIKTI**

# Kod İzleme: `PriorityQueue` içeride ne yapar?




**offer(1)**: indis 1'e eklenir; ebeveyni 5 > 1 → min-heap sırası bozuk.

Adım 3 / 11

```
1 PriorityQueue<Integer> pq =
2     new PriorityQueue<>();        // min-heap
3 pq.offer(5);
▶ pq.offer(1);
5 pq.offer(8);
6 System.out.println(pq.peek());    // bakar
7 System.out.println(pq.poll());    // çıkarır
8 System.out.println(pq.poll());
9 System.out.println(pq.size());
```

**DEĞİŞKENLER**

`pq.size() = 2` `queue = [5, 1]`

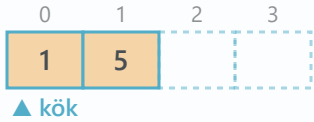
**ÇIKTI**

# Kod İzleme: `PriorityQueue` içeride ne yapar?




offer(1)

size = 2




siftUp (up-heap): 1 yukarı çıkar ve köke ulaşır → queue = [1, 5]. Maliyet O(log n).

Adım 4 / 11

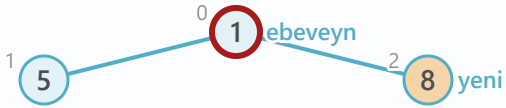
```
1 PriorityQueue<Integer> pq =
2     new PriorityQueue<>();        // min-heap
3 pq.offer(5);
▶ pq.offer(1);
5 pq.offer(8);
6 System.out.println(pq.peek());    // bakar
7 System.out.println(pq.poll());    // çıkarır
8 System.out.println(pq.poll());
9 System.out.println(pq.size());
```

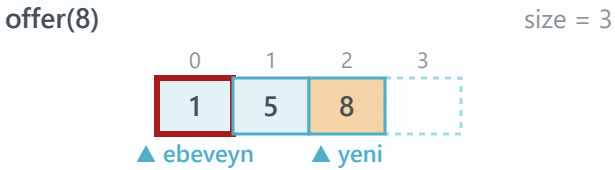
**DEĞİŞKENLER**

`pq.size() = 2` `queue = [1, 5]`

**ÇIKTI**

# Kod İzleme: `PriorityQueue` içeride ne yapar?




**offer(8)**: indis 2'ye eklenir; ebeveyn 1 ≤ 8 → up-heap gerekmez.

Adım 5 / 11

```
1 PriorityQueue<Integer> pq =
2     new PriorityQueue<>();        // min-heap
3 pq.offer(5);
4 pq.offer(1);
▶ pq.offer(8);
6 System.out.println(pq.peek());    // bakar
7 System.out.println(pq.poll());    // çıkarır
8 System.out.println(pq.poll());
9 System.out.println(pq.size());
```

**DEĞİŞKENLER**

`pq.size() = 3`

`queue = [1, 5, 8]`

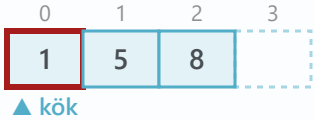
**ÇIKTI**

# Kod İzleme: `PriorityQueue` içeride ne yapar?




peek()

size = 3




**peek()**: kökteki 1 döner ama **çıkarılmaz** → O(1).

Adım 6 / 11

```
1 PriorityQueue<Integer> pq =
2     new PriorityQueue<>();        // min-heap
3 pq.offer(5);
4 pq.offer(1);
5 pq.offer(8);
▶ System.out.println(pq.peek());    // bakar
7 System.out.println(pq.poll());    // çıkarır
8 System.out.println(pq.poll());
9 System.out.println(pq.size());
```

**DEĞİŞKENLER**

`pq.size() = 3`

`queue = [1, 5, 8]`

**ÇIKTI**

```
1
```

# Kod İzleme: `PriorityQueue` içeride ne yapar?




poll()

size = 2




**poll()**: kök 1 alınır; son öğe 8 köke taşınır, size = 2.

Adım 7 / 11

```
1 PriorityQueue<Integer> pq =
2     new PriorityQueue<>();        // min-heap
3 pq.offer(5);
4 pq.offer(1);
5 pq.offer(8);
6 System.out.println(pq.peek());    // bakar
▶ System.out.println(pq.poll());    // çıkarır
8 System.out.println(pq.poll());
9 System.out.println(pq.size());
```

**DEĞİŞKENLER**

`pq.size() = 2` `queue = [8, 5]` `dönen = 1`

**ÇIKTI**

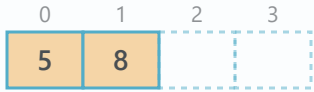
```
1
```

# Kod İzleme: `PriorityQueue` içeride ne yapar?




poll()

size = 2




siftDown (down-heap): 8 küçük çocukla yer değiştirir → queue = [5, 8]. Konsola 1 yazılır.

Adım 8 / 11

```
1 PriorityQueue<Integer> pq =
2     new PriorityQueue<>();        // min-heap
3 pq.offer(5);
4 pq.offer(1);
5 pq.offer(8);
6 System.out.println(pq.peek());    // bakar
▶ System.out.println(pq.poll());    // çıkarır
8 System.out.println(pq.poll());
9 System.out.println(pq.size());
```

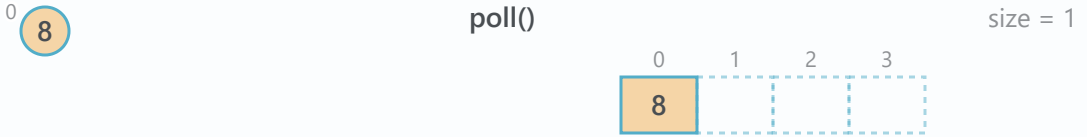
**DEĞİŞKENLER**

`pq.size() = 2` `queue = [5, 8]` `dönen = 1`

**ÇIKTI**

```
1
1
```

# Kod İzleme: `PriorityQueue` içeride ne yapar?




**poll()**: kök 5 alınır; son öğe 8 köke taşınır, size = 1.

Adım 9 / 11

```
1 PriorityQueue<Integer> pq =
2     new PriorityQueue<>();        // min-heap
3 pq.offer(5);
4 pq.offer(1);
5 pq.offer(8);
6 System.out.println(pq.peek());    // bakar
7 System.out.println(pq.poll());    // çıkarır
▶ System.out.println(pq.poll());
9 System.out.println(pq.size());
```

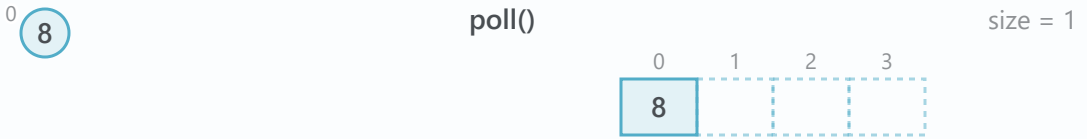
**DEĞİŞKENLER**

`pq.size() = 1` `queue = [8]` `dönen = 5`

**ÇIKTI**

```
1
1
```

# Kod İzleme: `PriorityQueue` içeride ne yapar?




Tek öğe kaldı; siftDown gerekmez. Konsola 5 yazılır.

Adım 10 / 11

```
1 PriorityQueue<Integer> pq =
2     new PriorityQueue<>();        // min-heap
3 pq.offer(5);
4 pq.offer(1);
5 pq.offer(8);
6 System.out.println(pq.peek());    // bakar
7 System.out.println(pq.poll());    // çıkarır
▶ System.out.println(pq.poll());
9 System.out.println(pq.size());
```

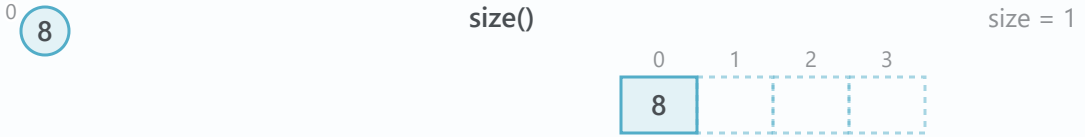
**DEĞİŞKENLER**

`pq.size() = 1` `queue = [8]` `dönen = 5`

**ÇIKTI**

```
1
1
5
```

# Kod İzleme: `PriorityQueue` içeride ne yapar?




**size()** = 1 (yalnızca 8 kaldı) → O(1). Konsol çıktısı: 1 (peek), 1, 5 (poll), 1 (size).

Adım 11 / 11

```
1 PriorityQueue<Integer> pq =
2     new PriorityQueue<>();        // min-heap
3 pq.offer(5);
4 pq.offer(1);
5 pq.offer(8);
6 System.out.println(pq.peek());    // bakar
7 System.out.println(pq.poll());    // çıkarır
8 System.out.println(pq.poll());
▶ System.out.println(pq.size());
```

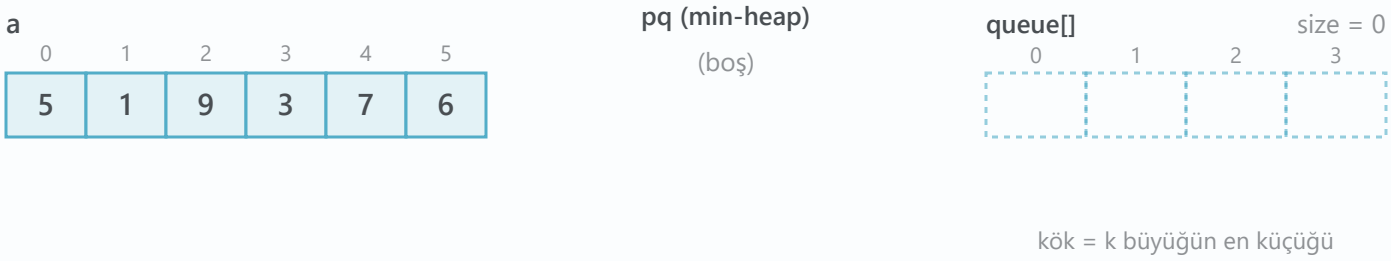
**DEĞİŞKENLER**

`pq.size() = 1` `queue = [8]`

**ÇIKTI**

```
1
1
5
1
```

# Uygulama: `kinciEnBuyuk([5, 1, 9, 3, 7, 6], 3)`




k = 3: boyutu en çok k olan bir **min-heap** tutacağız. Kökte, o ana dek görülen en büyük k öğenin *en küçüğü* durur.

Adım 1 / 14

```
1 static int kinciEnBuyuk(int[] a, int k) {
▶   PriorityQueue<Integer> pq = new PriorityQueue<>();
3   for (int x : a) {
4     pq.offer(x);                    // O(log k)
5     if (pq.size() > k) pq.poll();  // en küçüğü at
6   }
7   return pq.peek();        // k büyüğün en küçüğü
8 }
```

**DEĞİŞKENLER**

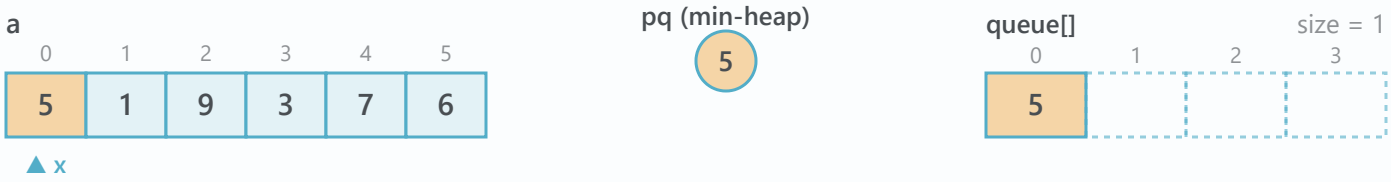
`k = 3` `pq = []`

**ÇAĞRI YIĞINI**

`kinciEnBuyuk(a, 3)`

`main()`

# Uygulama: `kinciEnBuyuk([5, 1, 9, 3, 7, 6], 3)`




x = 5: **offer(5)** → sona eklenir, up-heap. size = 1.

Adım 2 / 14

```
1 static int kinciEnBuyuk(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
▶   for (int x : a) {
▶     pq.offer(x);                    // O(log k)
5     if (pq.size() > k) pq.poll();  // en küçüğü at
6   }
7   return pq.peek();        // k büyüğün en küçüğü
8 }
```

**DEĞİŞKENLER**

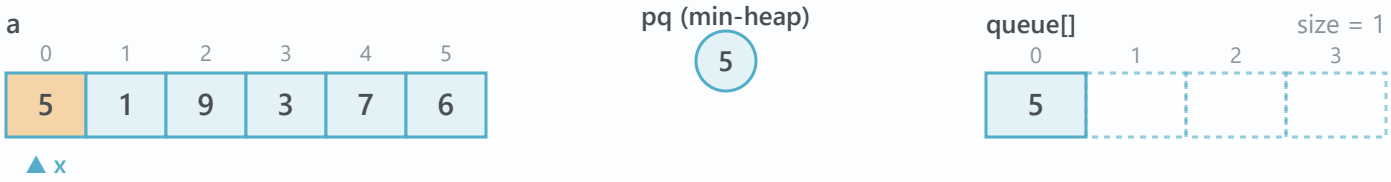
`k = 3` `x = 5` `pq = [5]`

**ÇAĞRI YIĞINI**

`kinciEnBuyuk(a, 3)`

`main()`

# Uygulama: `kinciEnBuyuk([5, 1, 9, 3, 7, 6], 3)`




size = 1 ≤ k = 3 → henüz atılacak öğe yok.

Adım 3 / 14

```
1 static int kinciEnBuyuk(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
3   for (int x : a) {
4     pq.offer(x);                      // O(log k)
▶     if (pq.size() > k) pq.poll();  // en küçüğü at
6   }
7   return pq.peek();        // k büyüğün en küçüğü
8 }
```

**DEĞİŞKENLER**

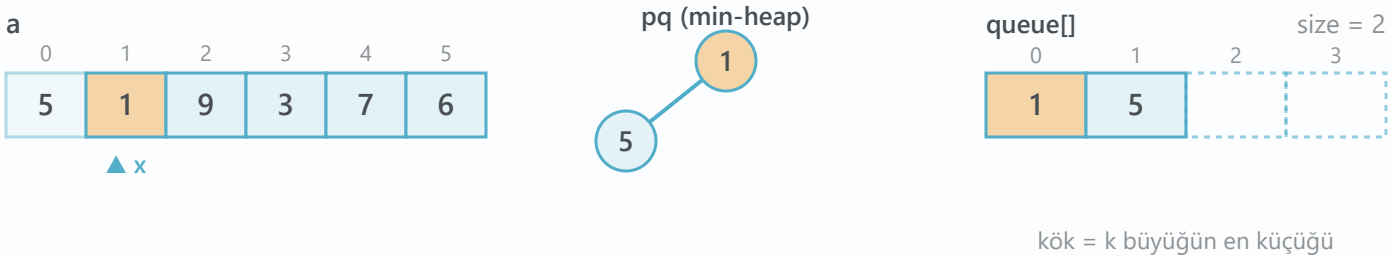
`k = 3` `x = 5` `pq = [5]`

**ÇAĞRI YIĞINI**

`kinciEnBuyuk(a, 3)`

`main()`

# Uygulama: `kinciEnBuyuk([5, 1, 9, 3, 7, 6], 3)`




x = 1: **offer(1)** → sona eklenir, up-heap (yukarı yüzdü: 1 kaydırma). size = 2.

Adım 4 / 14

```
1 static int kinciEnBuyuk(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
▶   for (int x : a) {
▶     pq.offer(x);                    // O(log k)
5     if (pq.size() > k) pq.poll();  // en küçüğü at
6   }
7   return pq.peek();        // k büyüğün en küçüğü
8 }
```

**DEĞİŞKENLER**

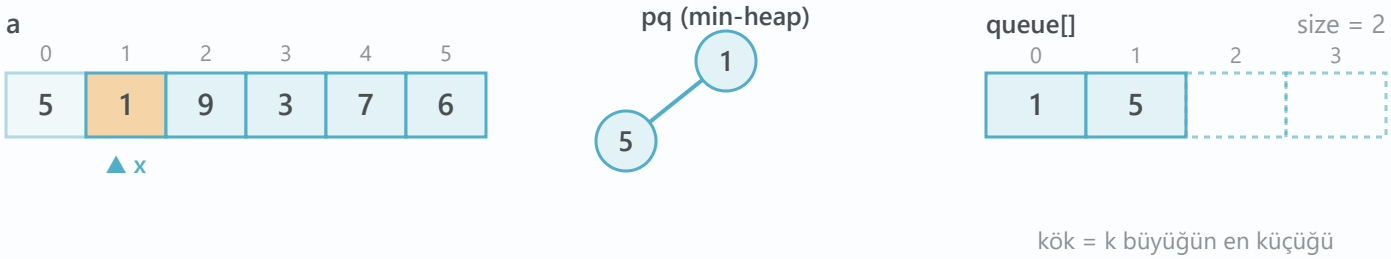
`k = 3` `x = 1` `pq = [1, 5]`

**ÇAĞRI YIĞINI**

`kinciEnBuyuk(a, 3)`

`main()`

# Uygulama: `kinciEnBuyuk([5, 1, 9, 3, 7, 6], 3)`




size = 2 ≤ k = 3 → henüz atılacak öğe yok.

Adım 5 / 14

```
1 static int kinciEnBuyuk(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
3   for (int x : a) {
4     pq.offer(x);                      // O(log k)
▶     if (pq.size() > k) pq.poll();  // en küçüğü at
6   }
7   return pq.peek();        // k büyüğün en küçüğü
8 }
```

**DEĞİŞKENLER**

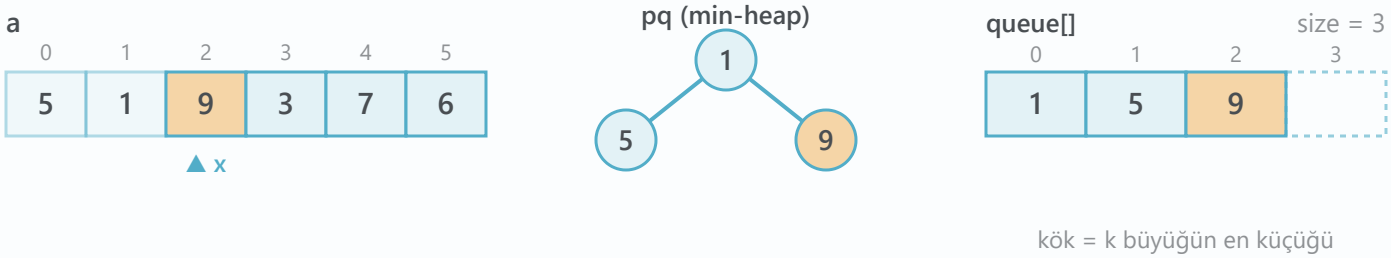
`k = 3` `x = 1` `pq = [1, 5]`

**ÇAĞRI YIĞINI**

`kinciEnBuyuk(a, 3)`

`main()`

# Uygulama: `kinciEnBuyuk([5, 1, 9, 3, 7, 6], 3)`




x = 9: **offer(9)** → sona eklenir, up-heap. size = 3.

Adım 6 / 14

```
1 static int kinciEnBuyuk(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
▶   for (int x : a) {
▶     pq.offer(x);                    // O(log k)
5     if (pq.size() > k) pq.poll();  // en küçüğü at
6   }
7   return pq.peek();        // k büyüğün en küçüğü
8 }
```

**DEĞİŞKENLER**

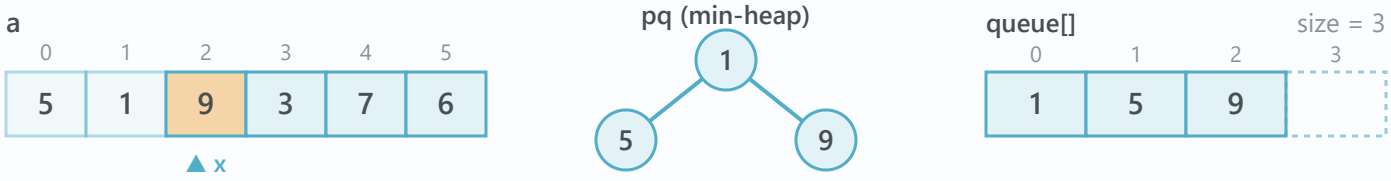
`k = 3` `x = 9` `pq = [1, 5, 9]`

**ÇAĞRI YIĞINI**

`kinciEnBuyuk(a, 3)`

`main()`

# Uygulama: `kinciEnBuyuk([5, 1, 9, 3, 7, 6], 3)`




size = 3 ≤ k = 3 → henüz atılacak öğe yok.

Adım 7 / 14

```
1 static int kinciEnBuyuk(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
3   for (int x : a) {
4     pq.offer(x);                    // O(log k)
▶     if (pq.size() > k) pq.poll();  // en küçüğü at
6   }
7   return pq.peek();        // k büyüğün en küçüğü
8 }
```

**DEĞİŞKENLER**

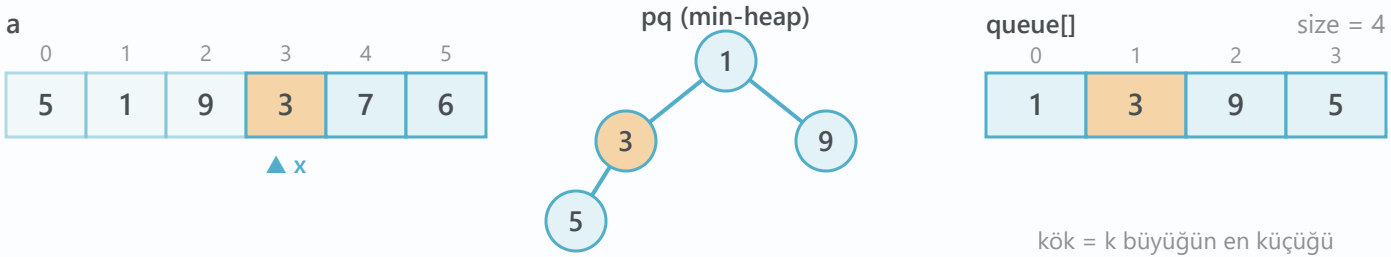
`k = 3` `x = 9` `pq = [1, 5, 9]`

**ÇAĞRI YIĞINI**

`kinciEnBuyuk(a, 3)`

`main()`

# Uygulama: `kinciEnBuyuk([5, 1, 9, 3, 7, 6], 3)`




x = 3: **offer(3)** → sona eklenir, up-heap (yukarı yüzdü: 1 kaydırma). size = 4.

Adım 8 / 14

```
1 static int kinciEnBuyuk(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
▶   for (int x : a) {
▶     pq.offer(x);                    // O(log k)
5     if (pq.size() > k) pq.poll();  // en küçüğü at
6   }
7   return pq.peek();        // k büyüğün en küçüğü
8 }
```

**DEĞİŞKENLER**

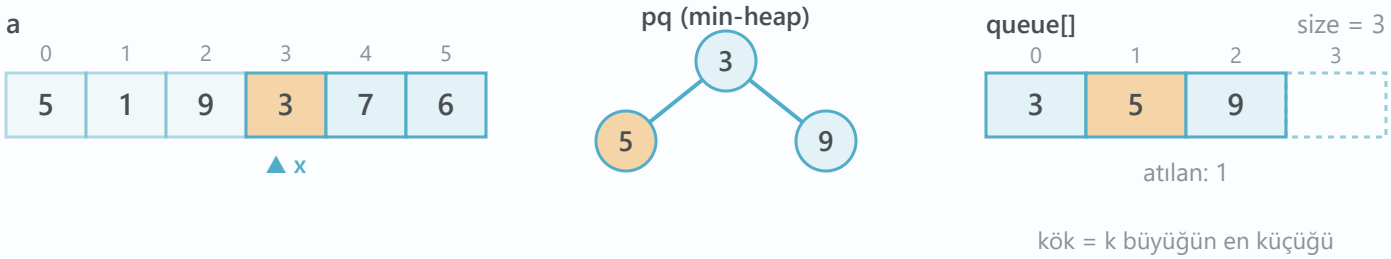
`k = 3` `x = 3` `pq = [1, 3, 9, 5]`

**ÇAĞRI YIĞINI**

`kinciEnBuyuk(a, 3)`

`main()`

# Uygulama: `kinciEnBuyuk([5, 1, 9, 3, 7, 6], 3)`




size = 4 > k → **poll()**: en küçük 1 atılır (k büyüğe giremez). Son öğe köke, down-heap.

Adım 9 / 14

```
1 static int kinciEnBuyuk(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
3   for (int x : a) {
4     pq.offer(x);                      // O(log k)
▶     if (pq.size() > k) pq.poll();  // en küçüğü at
6   }
7   return pq.peek();        // k büyüğün en küçüğü
8 }
```

**DEĞİŞKENLER**

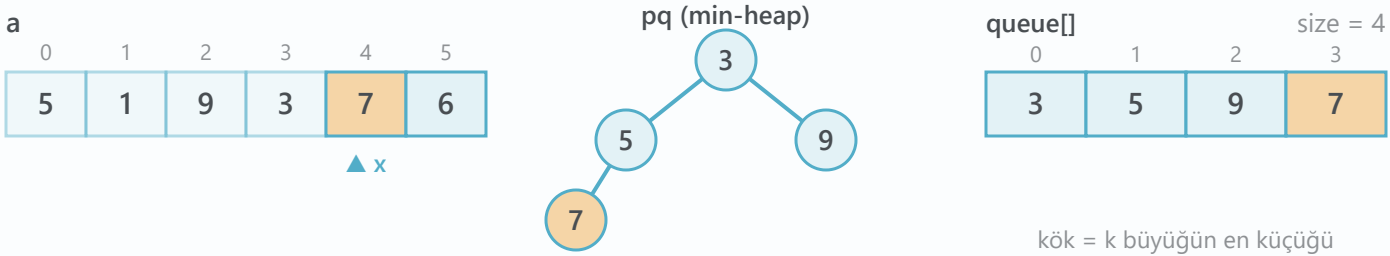
`k = 3` `x = 3` `pq = [3, 5, 9]`

**ÇAĞRI YIĞINI**

`kinciEnBuyuk(a, 3)`

`main()`

# Uygulama: `kinciEnBuyuk([5, 1, 9, 3, 7, 6], 3)`




x = 7: **offer(7)** → sona eklenir, up-heap. size = 4.

Adım 10 / 14

```
1 static int kinciEnBuyuk(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
▶   for (int x : a) {
▶     pq.offer(x);                    // O(log k)
5     if (pq.size() > k) pq.poll();  // en küçüğü at
6   }
7   return pq.peek();        // k büyüğün en küçüğü
8 }
```

**DEĞİŞKENLER**

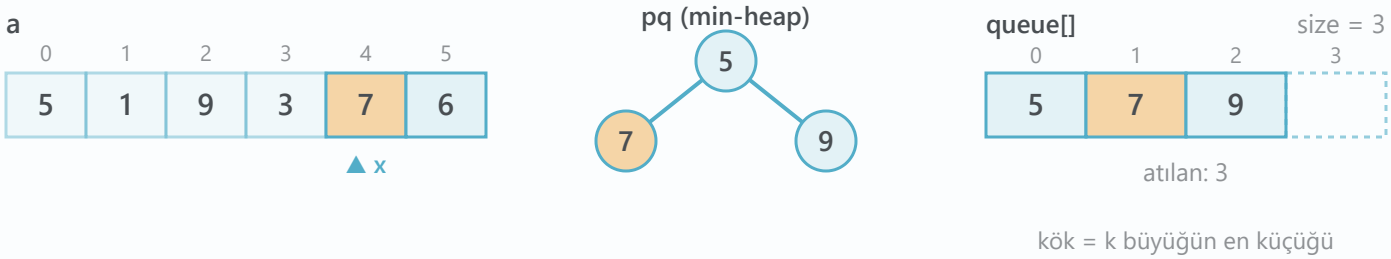
`k = 3` `x = 7` `pq = [3, 5, 9, 7]`

**ÇAĞRI YIĞINI**

`kinciEnBuyuk(a, 3)`

`main()`

# Uygulama: `kinciEnBuyuk([5, 1, 9, 3, 7, 6], 3)`




size = 4 > k → **poll()**: en küçük 3 atılır (k büyüğe giremez). Son öğe köke, down-heap.

Adım 11 / 14

```
1 static int kinciEnBuyuk(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
3   for (int x : a) {
4     pq.offer(x);                    // O(log k)
▶     if (pq.size() > k) pq.poll();  // en küçüğü at
6   }
7   return pq.peek();        // k büyüğün en küçüğü
8 }
```

**DEĞİŞKENLER**

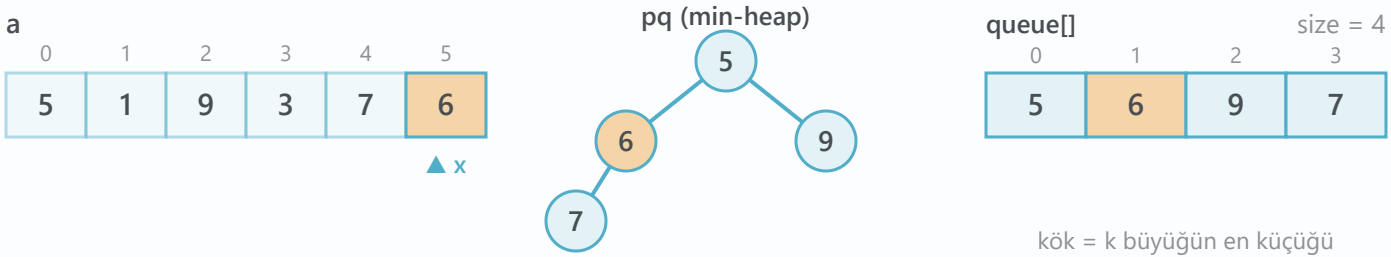
`k = 3` `x = 7` `pq = [5, 7, 9]`

**ÇAĞRI YIĞINI**

`kinciEnBuyuk(a, 3)`

`main()`

# Uygulama: `kinciEnBuyuk([5, 1, 9, 3, 7, 6], 3)`




x = 6: **offer(6)** → sona eklenir, up-heap (yukarı yüzdü: 1 kaydırma). size = 4.

Adım 12 / 14

```
1 static int kinciEnBuyuk(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
▶   for (int x : a) {
▶     pq.offer(x);                    // O(log k)
5     if (pq.size() > k) pq.poll();  // en küçüğü at
6   }
7   return pq.peek();        // k büyüğün en küçüğü
8 }
```

**DEĞİŞKENLER**

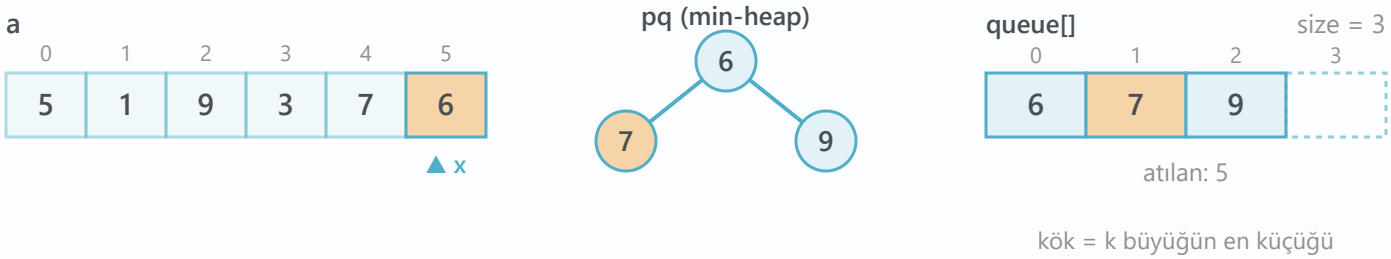
`k = 3` `x = 6` `pq = [5, 6, 9, 7]`

**ÇAĞRI YIĞINI**

`kinciEnBuyuk(a, 3)`

`main()`

# Uygulama: `kinciEnBuyuk([5, 1, 9, 3, 7, 6], 3)`




size = 4 > k → **poll()**: en küçük 5 atılır (k büyüğe giremez). Son öğe köke, down-heap.

Adım 13 / 14

```
1 static int kinciEnBuyuk(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
3   for (int x : a) {
4     pq.offer(x);                    // O(log k)
▶     if (pq.size() > k) pq.poll();  // en küçüğü at
6   }
7   return pq.peek();        // k büyüğün en küçüğü
8 }
```

**DEĞİŞKENLER**

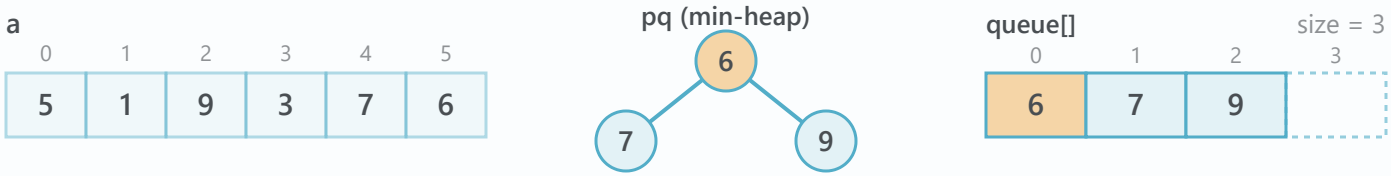
`k = 3` `x = 6` `pq = [6, 7, 9]`

**ÇAĞRI YIĞINI**

`kinciEnBuyuk(a, 3)`

`main()`

# Uygulama: `kinciEnBuyuk([5, 1, 9, 3, 7, 6], 3)`




Döngü bitti: pq = en büyük üç öğe {6, 7, 9}. **peek() = 6** → 3'üncü en büyük. Toplam **O(n log k)**, bellek O(k).

Adım 14 / 14

```
1 static int kinciEnBuyuk(int[] a, int k) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>();
3   for (int x : a) {
4     pq.offer(x);                    // O(log k)
5     if (pq.size() > k) pq.poll();  // en küçüğü at
6   }
▶   return pq.peek();        // k büyüğün en küçüğü
8 }
```

**DEĞİŞKENLER**

`k = 3` `pq = [6, 7, 9]`

**ÇAĞRI YIĞINI**

`kinciEnBuyuk(a, 3)`

`main()`

# Uygulama: Öncelikli Çizelgeleyici




**Kesin öncelik**

CPU her zaman en yüksek öncelikli, boş olmayan kuyruğun önündeki süreci seçer.

**Düzey içinde FIFO**

Aynı öncelikteki süreçler kendi kuyruklarında geliş sırasıyla (gerekirse round robin) çalışır.

**Açlık ve yaşlandırma**

Düşük öncelikliler hiç çalışamayabilir (starvation). Bekledikçe önceliği artırmak (aging) çözümdür.

Simülatör: Öncelikli Çizelgeleyici Simülatörü (üç düzey, quantum, isteğe bağlı yaşlandırma).

ÖZET

# Özet

**Öncelikli kuyruk**
insert, min, removeMin. Öğeler öncelik sırasıyla çıkar.

**Heap özelliği**
Tam ikili ağaç + ebeveyn ≥ çocuk (max) veya ≤ çocuk (min).

**Dizi gösterimi**
`çocuklar 2i+1, 2i+2 · ebeveyn ⌊(i−1)/2⌋`

| İşlem | Heap | Nasıl? |
|---|---|---|
| peek (max/min) | O(1) | heap[0] |
| ekle | O(log n) | heap[n]'e koy, up-heap |
| silMax / removeMin | O(log n) | son öğeyi köke taşı, down-heap |
| heap oluşturma | O(n) | ⌊n/2⌋ − 1'den 0'a down-heap |
| heap sort | O(n log n) | oluştur + n − 1 kez kökü sona al |

ÖZET

# Etkileşimli Simülatörler

### İkili Heap Simülatörü

Ekleme, en büyüğü/en küçüğü çıkarma; ağaç ve dizi görünümünde up-heap ve down-heap adımları.

### Öncelikli Kuyruk Simülatörü

Öncelikli öğe ekleme ve çıkarmayla ADT davranışı.

### Öncelikli Çizelgeleyici Simülatörü

İşletim sistemi tarzı öncelikli çizelgeleme.

ÖZET

# Sık Yapılan Hatalar

**"Heap sıralıdır"**

Değildir; yalnızca kök bellidir. 3, 1, 2 eklenen `PriorityQueue` için `println(pq)` → `[1, 3, 2]`.

**İndis formülleri**

0 tabanlı: 2i+1, 2i+2, (i−1)/2. 1 tabanlı: 2i, 2i+1, i/2. İkisini karıştırmak ağacı bozar.

**Sağ çocuk sınırı**

`j + 1 < n` kontrolü unutulursa heap dışındaki (silinmiş) öğe okunur.

**Yanlış çocuk**

Max-heap'te down-heap *büyük* çocukla takas eder; küçük çocuk seçilirse sıra bozulur.

**Heap'te arama**

`contains` ve `remove(nesne)` O(n)'dir. Hızlı erişim için konum (locator) ya da eşleme gerekir.

**Yön karışıklığı**

`PriorityQueue` varsayılan *min*-heap'tir; max için `Comparator.reverseOrder()`.

ÖZET

# Örnek Problemler

1. Boş bir max-heap'e sırasıyla 5, 12, 8, 3, 15, 10 ekleyin. Son dizi nedir?
2. 1. sorudaki heap'e bir kez `silMax()` uygulayın. Dönen değer ve yeni dizi nedir?
3. {3, 1, 6, 5, 2, 4} dizisini aşağıdan yukarıya yöntemle max-heap'e çevirin.
4. 0 tabanlı dizide indis 13'teki düğümün ebeveyni ve çocukları hangi indislerdedir?
5. {10, 7, 9, 8, 2, 5} bir max-heap midir? Değilse neden?
6. Bir min-heap'teki *en büyük* öğeyi bulmanın maliyeti nedir?

▶ **Cevapları göster**

ÖZET

# Alıştırmalar

1. [4, 10, 3, 5, 1] dizisine heap sort uygulayın; her adımdan sonra dizi nedir?
2. 100 öğeli bir ikili heap'in yüksekliği nedir? Bir eklemede en çok kaç takas yapılır?
3. Farklı anahtarlı max-heap'te 2. ve 3. en büyük öğe hangi indislerde olabilir?
4. `heapMi` yordamını min-heap için yeniden yazın. Boş ve tek öğeli dizide ne döner?
5. Toplam N öğeli k sıralı liste min-heap ile nasıl birleştirilir? Maliyet?
6. [9, 7, 8, 3, 5, 6, 4] max-heap'inde `anahtarDegistir(5, 10)` sonrası dizi nedir?

▶ **Cevap ve ipuçları**

# Kaynaklar


- Goodrich, Tamassia ve Goldwasser, *Data Structures and Algorithms in Java*, 6. baskı, 2014.
- Cormen, Leiserson, Rivest ve Stein, *Introduction to Algorithms*, 4. baskı, 2022.
- Oracle, Java SE 17 `PriorityQueue` API belgesi.

# Son

## Bölüm 11: Öncelikli Kuyruklar ve Heap

Sonraki sunum: Bölüm 12: Eşlemeler ve Hash Tabloları · Çalışma soruları · Simülatörler: İkili Heap Simülatörü, Öncelikli Kuyruk Simülatörü, Öncelikli Çizelgeleyici Simülatörü · Terim sözlüğü