# Bölüm 13: Kümeler ve Ayrık Kümeler

## Veri Yapıları

Küme ADT | Birleşim, kesişim, fark | Hash ve ağaç tabanlı küme | Çoklu küme | Union-Find | Yol kısaltma, rütbe | Bağlantılı bileşenler

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

GİRİŞ

# Öğrenme Çıktıları

**Küme modeli**

Küme ADT'sini ve Java `Set` arayüzünü açıklamak.

**Çoklu küme**

Tekrarlı öğe gerektiren problemlerde çoklu küme yaklaşımını kullanmak.

**Union-Find**

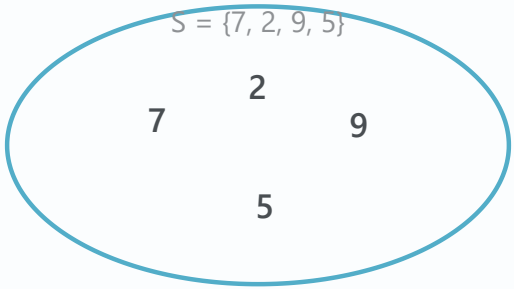
Yol kısaltma ve rütbeye göre birleştirme yöntemlerini uygulamak.

**Uygulama**

Union-Find yapısını bağlantılı bileşen sayma gibi bağlılık problemlerinde kullanmak.

# Küme (Set)

- Küme (set), **benzersiz (tekil)** öğelerden oluşan bir koleksiyondur: aynı öğe iki kez bulunmaz.
- Matematikteki küme kavramının veri yapısı karşılığıdır.
- Öğelerin sırası önemsizdir: {1, 2, 3} = {3, 1, 2}.
- Temel soru: **"x kümede var mı?"** (üyelik testi) hızlı yanıtlanmalıdır.



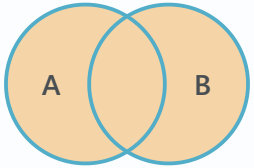

add(2) → değişmez (2 zaten var)

**Tekrar ayıklama**
Bir listedeki farklı kelimeler.
`new HashSet<>(liste)`

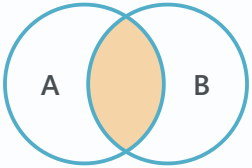
**Üyelik**
Yasaklı kullanıcılar, ziyaret edilen düğümler.

**Küme cebri**
Ortak arkadaşlar = kesişim; tüm etiketler = birleşim.

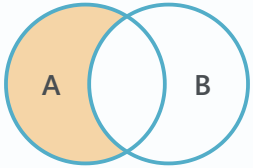
# Küme İşlemleri (İşleçler)



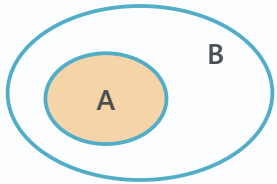

**Birleşim** A ∪ B
A'da *veya* B'de olanlar




**Kesişim** A ∩ B
Hem A'da *hem* B'de olanlar




**Fark** A − B
A'da olup B'de olmayanlar




**Alt küme** A ⊆ B
A'nın her öğesi B'de

| **A = {1, 2, 3, 4, 5}, B = {3, 4, 5, 6, 7}** | **Sonuç** |
|---|---|
| Birleşim A ∪ B | {1, 2, 3, 4, 5, 6, 7} |
| Kesişim A ∩ B | {3, 4, 5} |
| Fark A − B / B − A | {1, 2} / {6, 7} |
| Alt küme? | A ⊄ B (1 ∉ B) · {3, 4} ⊆ A ✓ |

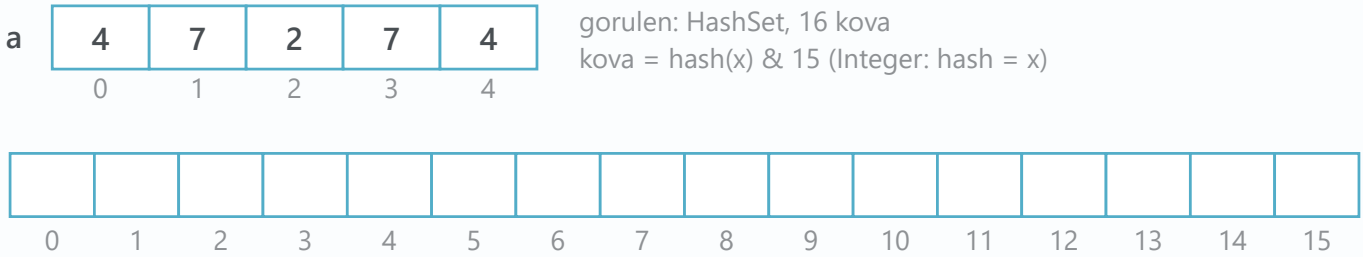
# java.util.Set Arayüzü

| Metot | Açıklama |
|---|---|
| `add(e)` | ekler; varsa `false` |
| `remove(o)`, `contains(o)` | çıkarır; üyeliği sorgular |
| `addAll(T)` | **birleşim**: S ← S ∪ T |
| `retainAll(T)` | **kesişim**: S ← S ∩ T |
| `removeAll(T)` | **fark**: S ← S − T |

```
Set<Integer> a = new HashSet<>(List.of(1,2,3,4,5));
Set<Integer> b = new HashSet<>(List.of(3,4,5,6,7));
Set<Integer> bir = new HashSet<>(a);
bir.addAll(b);     // [1..7]
Set<Integer> kes = new HashSet<>(a);
kes.retainAll(b);  // [3, 4, 5]
Set<Integer> fark = new HashSet<>(a);
fark.removeAll(b); // [1, 2]
a.containsAll(b);  // false
```

`addAll`, `retainAll`, `removeAll` kümeyi **yerinde değiştirir**; a'yı korumak için önce kopyalanır.

# Kod İzleme: `add` dönüşüyle ilk tekrar




Boş bir `HashSet` oluşturulur: 16 boş kova. Her öğe `hash & 15` kovasına gider.

Adım 1 / 10

```
1  int ilkTekrar(int[] a) {
▶    Set<Integer> gorulen = new HashSet<>();
3    for (int x : a) {
4      if (!gorulen.add(x))   // eklenemedi mi?
5        return x;            // zaten vardı: tekrar
6    }
7    return -1;               // tekrar yok
8  }
```

**DEĞİŞKENLER**

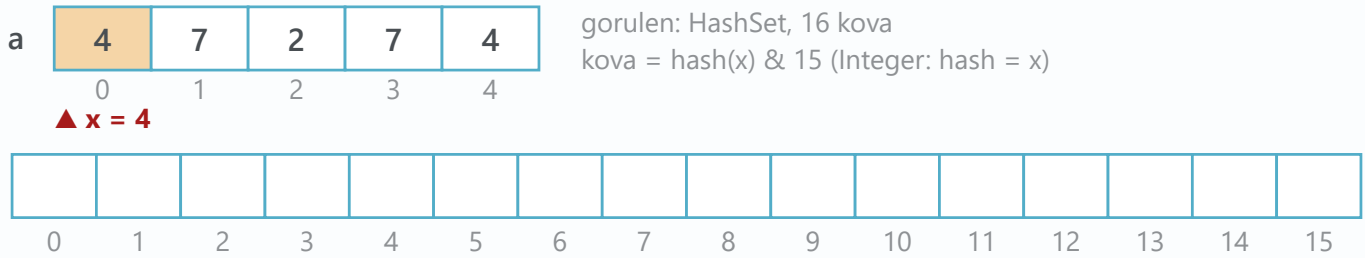
`a = [4, 7, 2, 7, 4]` `gorulen = {}`

**ÇAĞRI YIĞINI**

`ilkTekrar(a)`

`main()`

# Kod İzleme: `add` dönüşüyle ilk tekrar




Döngü sıradaki öğeyi alır: **x = 4**.

Adım 2 / 10

```
1 int ilkTekrar(int[] a) {
2   Set<Integer> gorulen = new HashSet<>();
▶   for (int x : a) {
4     if (!gorulen.add(x))   // eklenemedi mi?
5       return x;            // zaten vardı: tekrar
6   }
7   return -1;               // tekrar yok
8 }
```

**DEĞİŞKENLER**

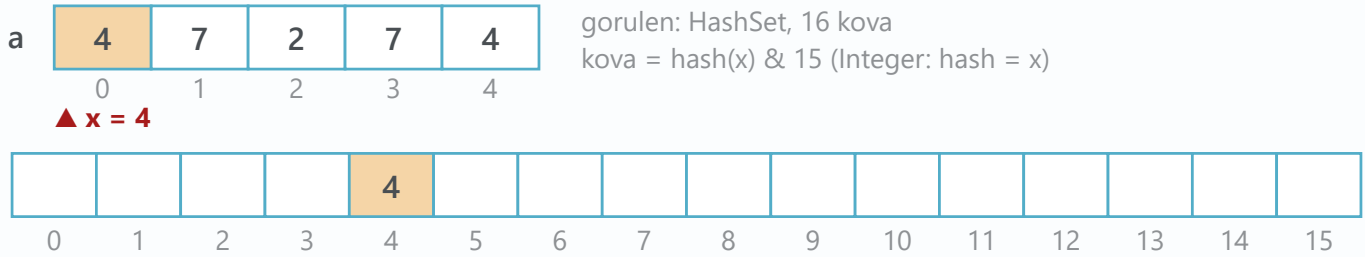
`a = [4, 7, 2, 7, 4]` `gorulen = {}` `x = 4`

**ÇAĞRI YIĞINI**

`ilkTekrar(a)`

`main()`

# Kod İzleme: `add` dönüşüyle ilk tekrar




`add(4)` : kova 4 boş, öğe eklenir ve **true** döner → `!true` yanlış, döngü sürer. Tek kova bakıldı: O(1).

Adım 3 / 10

```
1  int ilkTekrar(int[] a) {
2    Set<Integer> gorulen = new HashSet<>();
3    for (int x : a) {
▶      if (!gorulen.add(x))   // eklenemedi mi?
5        return x;            // zaten vardı: tekrar
6    }
7    return -1;               // tekrar yok
8  }
```

**DEĞİŞKENLER**

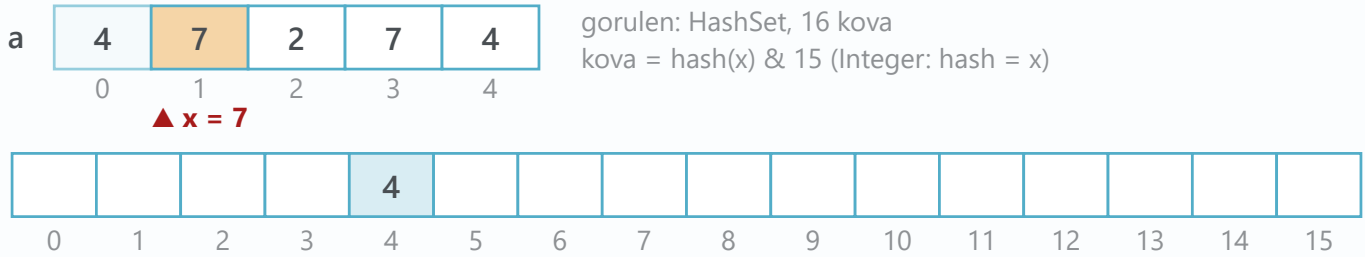
`a = [4, 7, 2, 7, 4]` `gorulen = {4}` `x = 4` `add = true`

**ÇAĞRI YIĞINI**

`ilkTekrar(a)`

`main()`

# Kod İzleme: `add` dönüşüyle ilk tekrar




Döngü sıradaki öğeyi alır: **x = 7**.

Adım 4 / 10

```
1 int ilkTekrar(int[] a) {
2   Set<Integer> gorulen = new HashSet<>();
▶   for (int x : a) {
4     if (!gorulen.add(x))    // eklenemedi mi?
5       return x;             // zaten vardı: tekrar
6   }
7   return -1;                // tekrar yok
8 }
```

**DEĞİŞKENLER**

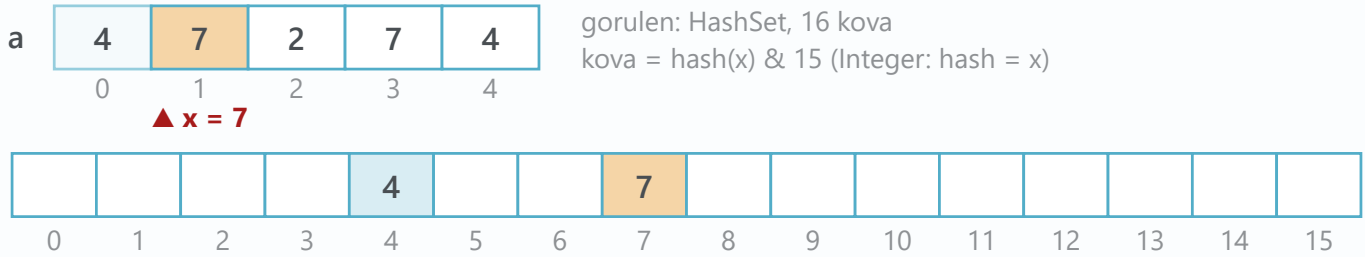
`a = [4, 7, 2, 7, 4]` `gorulen = {4}` `x = 7`

**ÇAĞRI YIĞINI**

`ilkTekrar(a)`

`main()`

# Kod İzleme: `add` dönüşüyle ilk tekrar




`add(7)` : kova 7 boş, öğe eklenir ve **true** döner → `!true` yanlış, döngü sürer. Tek kova bakıldı: O(1).

Adım 5 / 10

```
1 int ilkTekrar(int[] a) {
2   Set<Integer> gorulen = new HashSet<>();
3   for (int x : a) {
▶     if (!gorulen.add(x))   // eklenemedi mi?
5       return x;            // zaten vardı: tekrar
6   }
7   return -1;               // tekrar yok
8 }
```

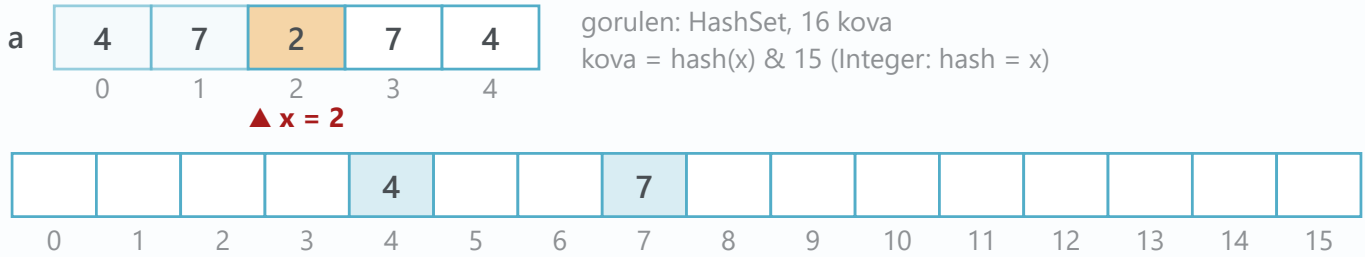
**DEĞİŞKENLER**

- `a = [4, 7, 2, 7, 4]`
- `gorulen = {4, 7}`
- `x = 7`
- `add = true`

**ÇAĞRI YIĞINI**

- `ilkTekrar(a)`
- `main()`

# Kod İzleme: `add` dönüşüyle ilk tekrar




Döngü sıradaki öğeyi alır: **x = 2**.

Adım 6 / 10

```
1 int ilkTekrar(int[] a) {
2   Set<Integer> gorulen = new HashSet<>();
▶   for (int x : a) {
4     if (!gorulen.add(x))    // eklenemedi mi?
5       return x;             // zaten vardı: tekrar
6   }
7   return -1;                // tekrar yok
8 }
```

**DEĞİŞKENLER**

`a = [4, 7, 2, 7, 4]`

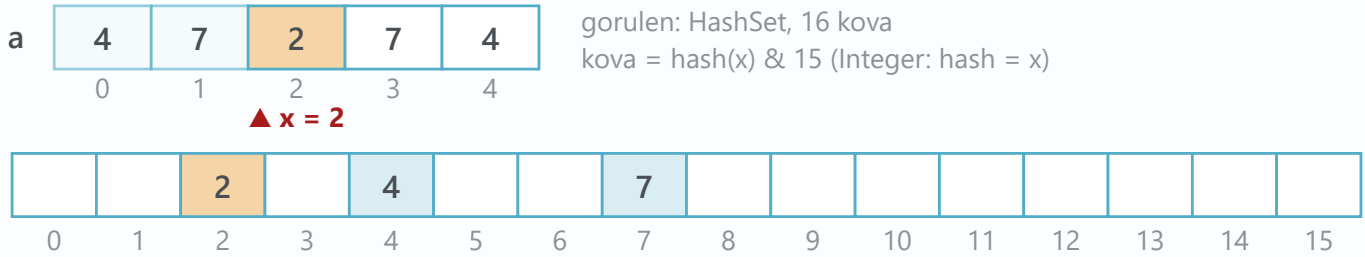
`gorulen = {4, 7}` `x = 2`

**ÇAĞRI YIĞINI**

`ilkTekrar(a)`

`main()`

# Kod İzleme: `add` dönüşüyle ilk tekrar




`add(2)` : kova 2 boş, öğe eklenir ve **true** döner → `!true` yanlış, döngü sürer. Tek kova bakıldı: O(1).

Adım 7 / 10

```
1 int ilkTekrar(int[] a) {
2   Set<Integer> gorulen = new HashSet<>();
3   for (int x : a) {
▶     if (!gorulen.add(x))    // eklenemedi mi?
5       return x;             // zaten vardı: tekrar
6   }
7   return -1;                // tekrar yok
8 }
```

**DEĞİŞKENLER**

`a = [4, 7, 2, 7, 4]`

`gorulen = {2, 4, 7}` `x = 2`

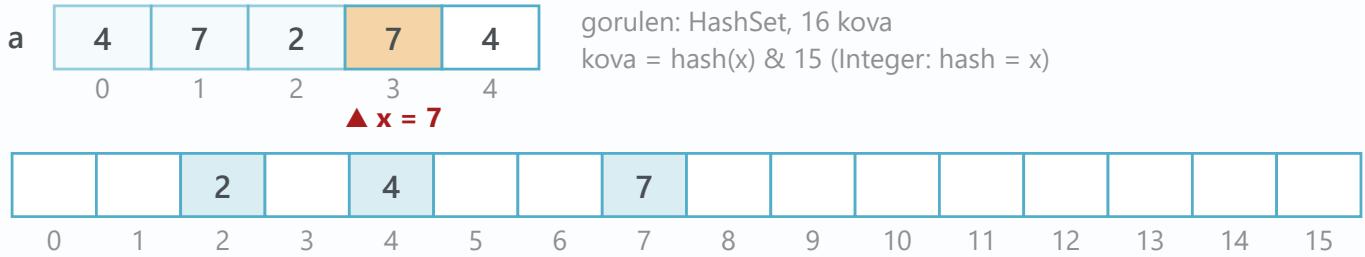
`add = true`

**ÇAĞRI YIĞINI**

`ilkTekrar(a)`

`main()`

# Kod İzleme: `add` dönüşüyle ilk tekrar




Döngü sıradaki öğeyi alır: **x = 7**.

Adım 8 / 10

```
1 int ilkTekrar(int[] a) {
2   Set<Integer> gorulen = new HashSet<>();
▶   for (int x : a) {
4     if (!gorulen.add(x))    // eklenemedi mi?
5       return x;             // zaten vardı: tekrar
6   }
7   return -1;                // tekrar yok
8 }
```

**DEĞİŞKENLER**

`a = [4, 7, 2, 7, 4]`

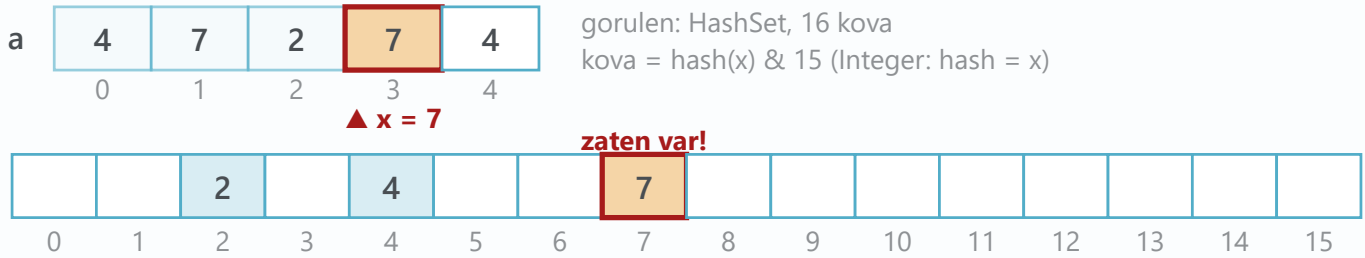
`gorulen = {2, 4, 7}` `x = 7`

**ÇAĞRI YIĞINI**

`ilkTekrar(a)`

`main()`

# Kod İzleme: `add` dönüşüyle ilk tekrar




`add(7)` : kova 7'de **7 zaten var**; küme değişmez ve **false** döner → koşul doğru.

Adım 9 / 10

```
1 int ilkTekrar(int[] a) {
2   Set<Integer> gorulen = new HashSet<>();
3   for (int x : a) {
▶     if (!gorulen.add(x))    // eklenemedi mi?
5       return x;             // zaten vardı: tekrar
6   }
7   return -1;                // tekrar yok
8 }
```

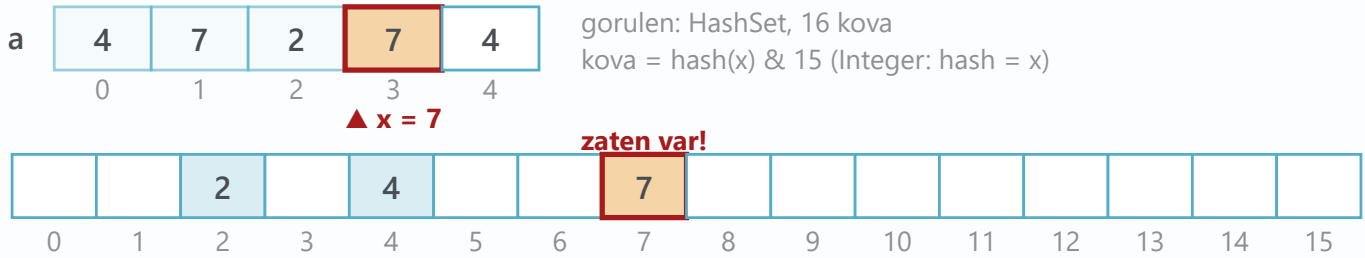
**DEĞİŞKENLER**

- `a = [4, 7, 2, 7, 4]`
- `gorulen = {2, 4, 7}`
- `x = 7`
- `add = false`

**ÇAĞRI YIĞINI**

- `ilkTekrar(a)`
- `main()`

# Kod İzleme: `add` dönüşüyle ilk tekrar




İlk tekrar eden öğe **7** döndürülür. Toplam iş O(n): her öğe için bir `add`; iç içe döngü yok.

Adım 10 / 10

```
1 int ilkTekrar(int[] a) {
2   Set<Integer> gorulen = new HashSet<>();
3   for (int x : a) {
4     if (!gorulen.add(x))    // eklenemedi mi?
▶       return x;             // zaten vardı: tekrar
6   }
7   return -1;                // tekrar yok
8 }
```

**DEĞİŞKENLER**

`a = [4, 7, 2, 7, 4]`

`gorulen = {2, 4, 7}` `x = 7`

`dönüş = 7`

**ÇAĞRI YIĞINI**

`ilkTekrar(a)`

`main()`

KÜME UYGULAMALARI

# java.util.Set Uygulamaları

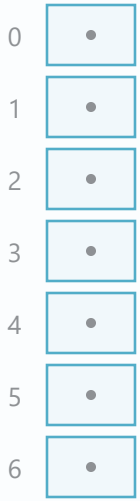
| Sınıf | İç yapı | Gezinme sırası | add / contains / remove |
|---|---|---|---|
| **HashSet** | hash tablosu | hash koduna bağlı, öngörülemez | O(1) ortalama |
| **LinkedHashSet** | hash tablosu + çift bağlı liste | ekleme sırası | O(1) ortalama |
| **TreeSet** | kırmızı-siyah ağaç | sıralı (küçükten büyüğe, alfabetik) | O(log n) |

```
for (String s : List.of("elma", "kiraz", "armut", "elma")) {
    hash.add(s);  linked.add(s);  tree.add(s);
}
// HashSet       → [elma, armut, kiraz]   (kova sırası: hash'e bağlı)
// LinkedHashSet → [elma, kiraz, armut]   (eklenme sırası)
// TreeSet       → [armut, elma, kiraz]   (alfabetik)
```

Tekrar eden "elma" üçünde de **bir kez** bulunur. Sıra önemsizse HashSet, ekleme sırası için LinkedHashSet, sıralı erişim için TreeSet.

# Hash Tabanlı Küme

- Küme bir **hash tablosu** olarak temsil edilir; yalnızca anahtar saklanır, değer yoktur.
- Her öğenin konumu hash koduna göre hızla hesaplanır.
- Ekleme, silme ve arama ortalama **O(1)**; kötü hash ile en kötü O(n).
- Java'da `HashSet` içeride bir `HashMap` kullanır.

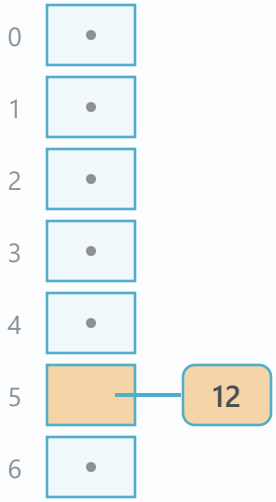



Başlangıç: 7 kovalı boş tablo. Konum = `hashCode(x) mod 7`.

Adım 1 / 7

# Hash Tabanlı Küme

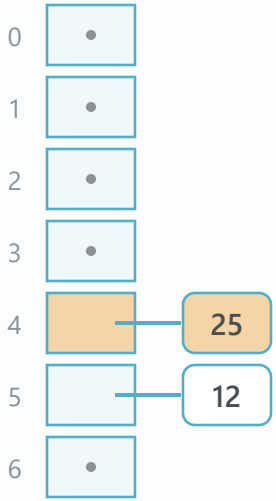
- Küme bir **hash tablosu** olarak temsil edilir; yalnızca anahtar saklanır, değer yoktur.
- Her öğenin konumu hash koduna göre hızla hesaplanır.
- Ekleme, silme ve arama ortalama **O(1)**; kötü hash ile en kötü O(n).
- Java'da `HashSet` içeride bir `HashMap` kullanır.




add(12): 12 mod 7 = **5** → kova 5 boş, eklenir.

# Hash Tabanlı Küme

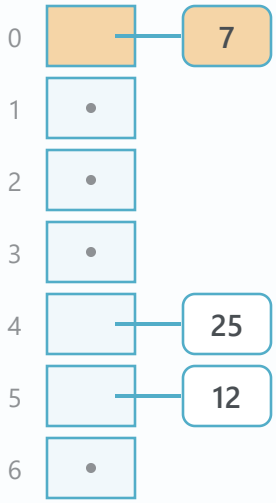
- Küme bir **hash tablosu** olarak temsil edilir; yalnızca anahtar saklanır, değer yoktur.
- Her öğenin konumu hash koduna göre hızla hesaplanır.
- Ekleme, silme ve arama ortalama **O(1)**; kötü hash ile en kötü O(n).
- Java'da `HashSet` içeride bir `HashMap` kullanır.




add(25): 25 mod 7 = **4** → kova 4'e eklenir.

# Hash Tabanlı Küme

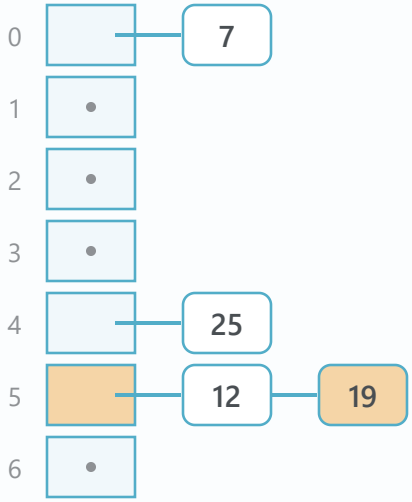
- Küme bir **hash tablosu** olarak temsil edilir; yalnızca anahtar saklanır, değer yoktur.
- Her öğenin konumu hash koduna göre hızla hesaplanır.
- Ekleme, silme ve arama ortalama **O(1)**; kötü hash ile en kötü O(n).
- Java'da `HashSet` içeride bir `HashMap` kullanır.




add(7): 7 mod 7 = **0** → kova 0'a eklenir.

# Hash Tabanlı Küme

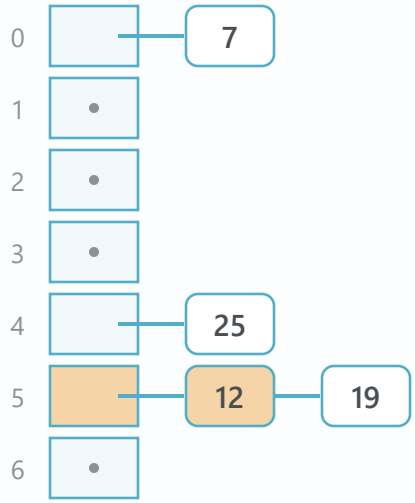
- Küme bir **hash tablosu** olarak temsil edilir; yalnızca anahtar saklanır, değer yoktur.
- Her öğenin konumu hash koduna göre hızla hesaplanır.
- Ekleme, silme ve arama ortalama **O(1)**; kötü hash ile en kötü O(n).
- Java'da `HashSet` içeride bir `HashMap` kullanır.




add(19): 19 mod 7 = **5** → çakışma! Kova 5'te 12 var; 19 zincirin sonuna eklenir.

# Hash Tabanlı Küme

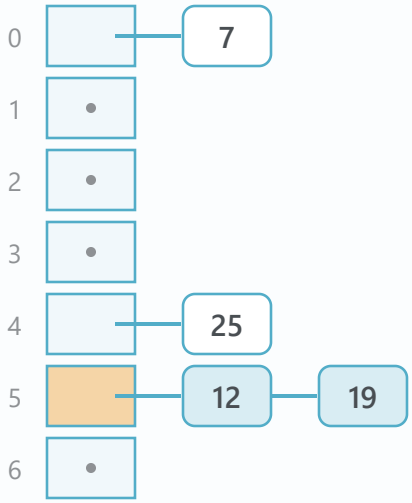
- Küme bir **hash tablosu** olarak temsil edilir; yalnızca anahtar saklanır, değer yoktur.
- Her öğenin konumu hash koduna göre hızla hesaplanır.
- Ekleme, silme ve arama ortalama **O(1)**; kötü hash ile en kötü O(n).
- Java'da `HashSet` içeride bir `HashMap` kullanır.




add(12) tekrar: kova 5 taranır, 12 bulunur → **eklenmez**, `false` döner. Kümeyi küme yapan budur.

# Hash Tabanlı Küme

- Küme bir **hash tablosu** olarak temsil edilir; yalnızca anahtar saklanır, değer yoktur.
- Her öğenin konumu hash koduna göre hızla hesaplanır.
- Ekleme, silme ve arama ortalama **O(1)**; kötü hash ile en kötü O(n).
- Java'da `HashSet` içeride bir `HashMap` kullanır.




contains(26): 26 mod 7 = 5 → yalnızca kova 5 (12, 19) taranır, bulunamaz → `false`. Diğer kovalara hiç bakılmaz.

# Kod İzleme: `add` / `contains` (m = 7)

```
1  // alanlar: List<Integer>[] kova; n; m = 7
2  boolean add(int x) {          // x ≥ 0
▶    int i = x % m;               // kova no
4    for (int y : kova[i])
5      if (y == x) return false;  // zaten var
6    kova[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : kova[i])
13     if (y == x) return true;
14   return false;
15 }
```




`add(12)`: kova no = 12 % 7 = **5**. Kova boş; tarama hemen biter.

Adım 1 / 18

**ÇAĞRI YIĞINI**

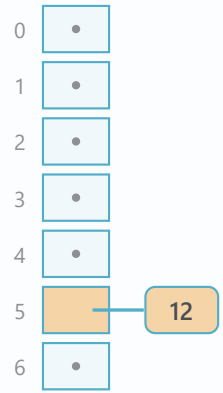
- `add(12)`
- `main()`

**DEĞİŞKENLER**

`x = 12` `i = 5` `n = 0`

# Kod İzleme: `add` / `contains` (m = 7)

```
1  // alanlar: List<Integer>[] kova; n; m = 7
2  boolean add(int x) {          // x ≥ 0
3    int i = x % m;               // kova no
4    for (int y : kova[i])
5      if (y == x) return false;  // zaten var
▶    kova[i].add(x);
▶    n++;
▶    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : kova[i])
13     if (y == x) return true;
14   return false;
15 }
```




12 kovada yok → zincirin sonuna eklenir, **n = 1**, `return true`.



**ÇAĞRI YIĞINI**

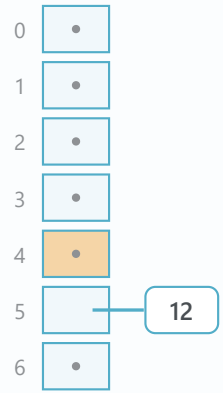
- **add(12)**
- main()

**DEĞİŞKENLER**

`x = 12` `i = 5` `n = 1` `dönüş = true`

# Kod İzleme: `add` / `contains` (m = 7)

```
1  // alanlar: List<Integer>[] kova; n; m = 7
2  boolean add(int x) {          // x ≥ 0
▶    int i = x % m;               // kova no
4    for (int y : kova[i])
5      if (y == x) return false;  // zaten var
6    kova[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : kova[i])
13     if (y == x) return true;
14   return false;
15 }
```




`add(25)`: kova no = 25 % 7 = **4**. Kova boş; tarama hemen biter.

Adım 3 / 18

**ÇAĞRI YIĞINI**

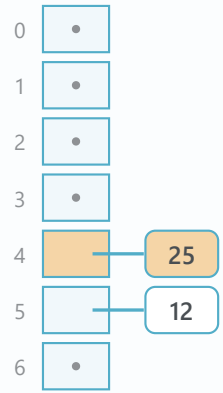
- `add(25)`
- `main()`

**DEĞİŞKENLER**

`x = 25` `i = 4` `n = 1`

# Kod İzleme: `add` / `contains` (m = 7)

```
1  // alanlar: List<Integer>[] kova; n; m = 7
2  boolean add(int x) {          // x ≥ 0
3    int i = x % m;               // kova no
4    for (int y : kova[i])
5      if (y == x) return false;  // zaten var
▶    kova[i].add(x);
▶    n++;
▶    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : kova[i])
13     if (y == x) return true;
14   return false;
15 }
```




25 kovada yok → zincirin sonuna eklenir, **n = 2**, `return true`.



**ÇAĞRI YIĞINI**

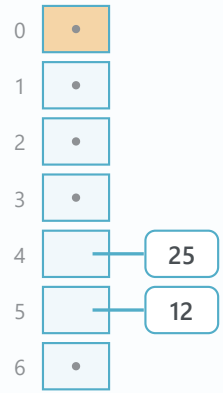
- `add(25)`
- `main()`

**DEĞİŞKENLER**

`x = 25` `i = 4` `n = 2` `dönüş = true`

# Kod İzleme: `add` / `contains` (m = 7)

```
1  // alanlar: List<Integer>[] kova; n; m = 7
2  boolean add(int x) {          // x ≥ 0
▶    int i = x % m;               // kova no
4    for (int y : kova[i])
5      if (y == x) return false;  // zaten var
6    kova[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : kova[i])
13     if (y == x) return true;
14   return false;
15 }
```




`add(7)` : kova no = 7 % 7 = **0**. Kova boş; tarama hemen biter.

Adım 5 / 18

**ÇAĞRI YIĞINI**

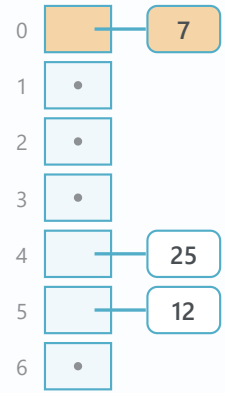
- `add(7)`
- `main()`

**DEĞİŞKENLER**

`x = 7` `i = 0` `n = 2`

# Kod İzleme: `add` / `contains` (m = 7)

```
1  // alanlar: List<Integer>[] kova; n; m = 7
2  boolean add(int x) {          // x ≥ 0
3    int i = x % m;               // kova no
4    for (int y : kova[i])
5      if (y == x) return false;  // zaten var
▶    kova[i].add(x);
▶    n++;
▶    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : kova[i])
13     if (y == x) return true;
14   return false;
15 }
```




7 kovada yok → zincirin sonuna eklenir, **n = 3**, `return true`.



**ÇAĞRI YIĞINI**

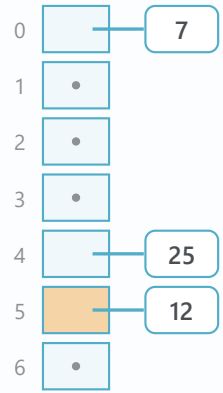
- **add(7)**
- main()

**DEĞİŞKENLER**

`x = 7` `i = 0` `n = 3` `dönüş = true`

# Kod İzleme: `add` / `contains` (m = 7)

```
1  // alanlar: List<Integer>[] kova; n; m = 7
2  boolean add(int x) {          // x ≥ 0
▶    int i = x % m;               // kova no
4    for (int y : kova[i])
5      if (y == x) return false;  // zaten var
6    kova[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : kova[i])
13     if (y == x) return true;
14   return false;
15 }
```




`add(19)`: kova no = 19 % 7 = **5**. Kovadaki zincir taranmalı: 12.

Adım 7 / 18

**ÇAĞRI YIĞINI**

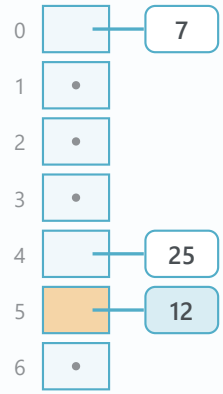
- `add(19)`
- `main()`

**DEĞİŞKENLER**

`x = 19` `i = 5` `n = 3`

# Kod İzleme: `add` / `contains` (m = 7)

```
1  // alanlar: List<Integer>[] kova; n; m = 7
2  boolean add(int x) {          // x ≥ 0
3    int i = x % m;               // kova no
4    for (int y : kova[i])
▶      if (y == x) return false;  // zaten var
6    kova[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : kova[i])
13     if (y == x) return true;
14   return false;
15 }
```




y = 12 ≠ 19 → zincirde ilerle.

Adım 8 / 18

**ÇAĞRI YIĞINI**

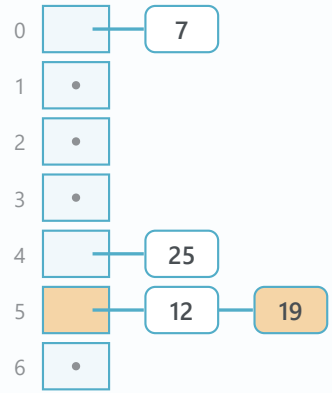
- `add(19)`
- `main()`

**DEĞİŞKENLER**

`x = 19` `i = 5` `n = 3` `y = 12`

# Kod İzleme: `add` / `contains` (m = 7)

```
1  // alanlar: List<Integer>[] kova; n; m = 7
2  boolean add(int x) {          // x ≥ 0
3    int i = x % m;               // kova no
4    for (int y : kova[i])
5      if (y == x) return false;  // zaten var
▶    kova[i].add(x);
▶    n++;
▶    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : kova[i])
13     if (y == x) return true;
14   return false;
15 }
```




19 kovada yok → zincirin sonuna eklenir, **n = 4**, `return true`. (Çakışma: kova 5'te artık 2 öğe var.)

Adım 9 / 18

**ÇAĞRI YIĞINI**

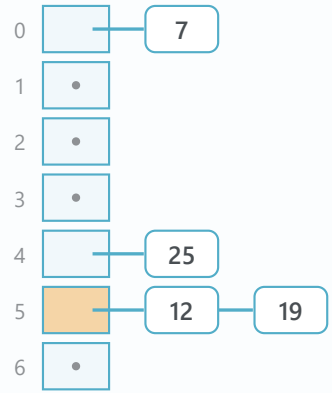
- `add(19)`
- `main()`

**DEĞİŞKENLER**

`x = 19` `i = 5` `n = 4` `dönüş = true`

# Kod İzleme: `add` / `contains` (m = 7)

```
1  // alanlar: List<Integer>[] kova; n; m = 7
2  boolean add(int x) {          // x ≥ 0
▶    int i = x % m;               // kova no
4    for (int y : kova[i])
5      if (y == x) return false;  // zaten var
6    kova[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : kova[i])
13     if (y == x) return true;
14   return false;
15 }
```




`add(12)`: kova no = 12 % 7 = **5**. Kovadaki zincir taranmalı: 12, 19.

Adım 10 / 18

**ÇAĞRI YIĞINI**

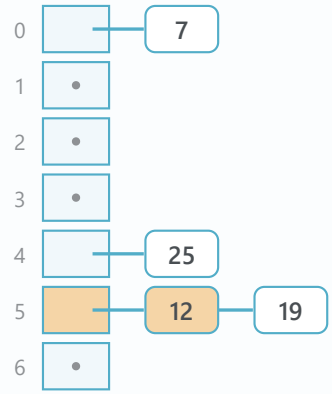
- `add(12)`
- `main()`

**DEĞİŞKENLER**

`x = 12` `i = 5` `n = 4`

# Kod İzleme: `add` / `contains` (m = 7)

```
1  // alanlar: List<Integer>[] kova; n; m = 7
2  boolean add(int x) {          // x ≥ 0
3    int i = x % m;               // kova no
4    for (int y : kova[i])
▶      if (y == x) return false;  // zaten var
6    kova[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : kova[i])
13     if (y == x) return true;
14   return false;
15 }
```




y = 12 == 12 → öğe **zaten kümede**: `return false`, hiçbir şey eklenmez. Kümeyi küme yapan budur.

Adım 11 / 18

**ÇAĞRI YIĞINI**

- `add(12)`
- `main()`

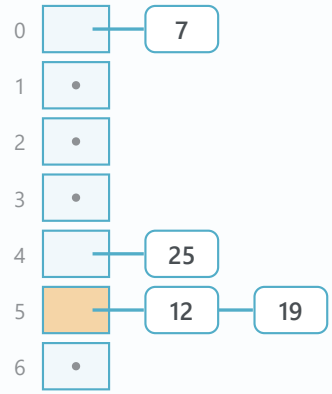
**DEĞİŞKENLER**

`x = 12` `i = 5` `n = 4` `y = 12`

`dönüş = false`

# Kod İzleme: `add` / `contains` (m = 7)

```
1  // alanlar: List<Integer>[] kova; n; m = 7
2  boolean add(int x) {          // x ≥ 0
3    int i = x % m;               // kova no
4    for (int y : kova[i])
5      if (y == x) return false;  // zaten var
6    kova[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
▶    int i = x % m;
12   for (int y : kova[i])
13     if (y == x) return true;
14   return false;
15 }
```




`contains(26)` : 26 % 7 = **5** → yalnızca kova 5 taranır; diğer kovalara hiç bakılmaz.

Adım 12 / 18

**ÇAĞRI YIĞINI**

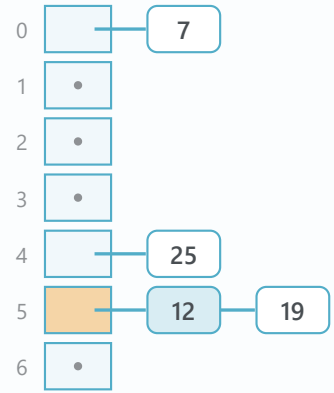
- `contains(26)`
- `main()`

**DEĞİŞKENLER**

`x = 26` `i = 5` `n = 4`

# Kod İzleme: `add` / `contains` (m = 7)

```
1  // alanlar: List<Integer>[] kova; n; m = 7
2  boolean add(int x) {           // x ≥ 0
3    int i = x % m;                // kova no
4    for (int y : kova[i])
5      if (y == x) return false;  // zaten var
6    kova[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : kova[i])
▶      if (y == x) return true;
14   return false;
15 }
```




y = 12 ≠ 26 → devam.

Adım 13 / 18

**ÇAĞRI YIĞINI**

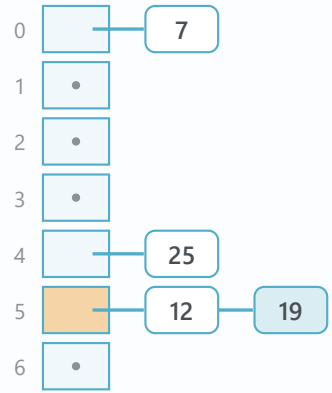
- `contains(26)`
- `main()`

**DEĞİŞKENLER**

`x = 26` `i = 5` `n = 4` `y = 12`

# Kod İzleme: `add` / `contains` (m = 7)

```
1  // alanlar: List<Integer>[] kova; n; m = 7
2  boolean add(int x) {          // x ≥ 0
3    int i = x % m;               // kova no
4    for (int y : kova[i])
5      if (y == x) return false;  // zaten var
6    kova[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : kova[i])
▶      if (y == x) return true;
14   return false;
15 }
```




y = 19 ≠ 26 → devam.

Adım 14 / 18

**ÇAĞRI YIĞINI**

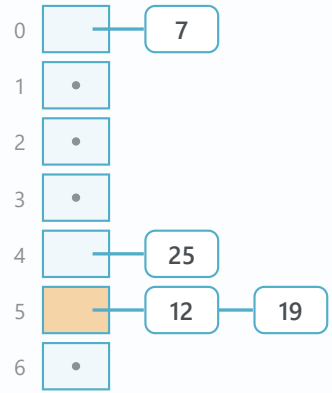
- `contains(26)`
- `main()`

**DEĞİŞKENLER**

`x = 26` `i = 5` `n = 4` `y = 19`

# Kod İzleme: `add` / `contains` (m = 7)

```
1  // alanlar: List<Integer>[] kova; n; m = 7
2  boolean add(int x) {          // x ≥ 0
3    int i = x % m;               // kova no
4    for (int y : kova[i])
5      if (y == x) return false;  // zaten var
6    kova[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : kova[i])
13     if (y == x) return true;
▶    return false;
15 }
```




Zincir bitti, 26 yok → `return false`. Ortalama zincir boyu n/m sabit → **O(1)**.



**ÇAĞRI YIĞINI**

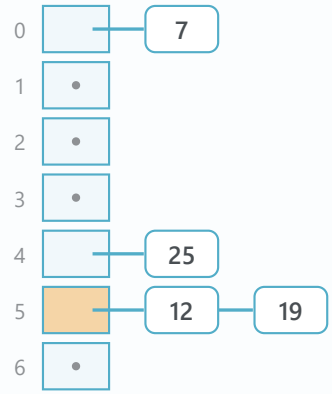
- `contains(26)`
- `main()`

**DEĞİŞKENLER**

`x = 26` `i = 5` `n = 4` `dönüş = false`

# Kod İzleme: `add` / `contains` (m = 7)

```
1  // alanlar: List<Integer>[] kova; n; m = 7
2  boolean add(int x) {          // x ≥ 0
3    int i = x % m;               // kova no
4    for (int y : kova[i])
5      if (y == x) return false;  // zaten var
6    kova[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
▶    int i = x % m;
12   for (int y : kova[i])
13     if (y == x) return true;
14   return false;
15 }
```




`contains(19)` : 19 % 7 = **5** → yalnızca kova 5 taranır; diğer kovalara hiç bakılmaz.

Adım 16 / 18

**ÇAĞRI YIĞINI**

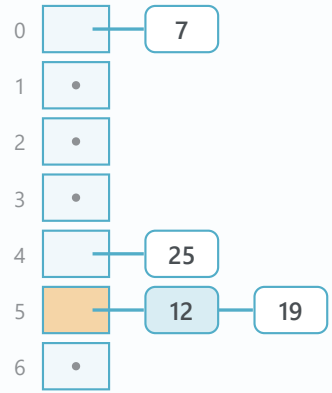
- `contains(19)`
- `main()`

**DEĞİŞKENLER**

`x = 19` `i = 5` `n = 4`

# Kod İzleme: `add` / `contains` (m = 7)

```
1  // alanlar: List<Integer>[] kova; n; m = 7
2  boolean add(int x) {           // x ≥ 0
3    int i = x % m;                // kova no
4    for (int y : kova[i])
5      if (y == x) return false;  // zaten var
6    kova[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : kova[i])
▶      if (y == x) return true;
14   return false;
15 }
```




y = 12 ≠ 19 → devam.

Adım 17 / 18

**ÇAĞRI YIĞINI**

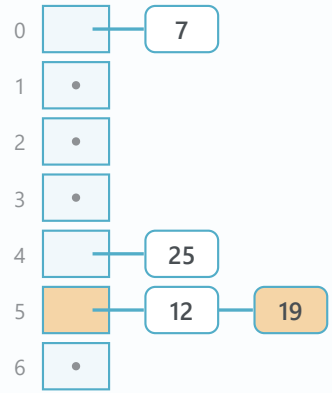
- `contains(19)`
- `main()`

**DEĞİŞKENLER**

`x = 19` `i = 5` `n = 4` `y = 12`

# Kod İzleme: `add` / `contains` (m = 7)

```
1  // alanlar: List<Integer>[] kova; n; m = 7
2  boolean add(int x) {          // x ≥ 0
3    int i = x % m;               // kova no
4    for (int y : kova[i])
5      if (y == x) return false;  // zaten var
6    kova[i].add(x);
7    n++;
8    return true;
9  }
10 boolean contains(int x) {
11   int i = x % m;
12   for (int y : kova[i])
▶      if (y == x) return true;
14   return false;
15 }
```




y = 19 == 19 → **bulundu**, `return true`.

Adım 18 / 18

**ÇAĞRI YIĞINI**

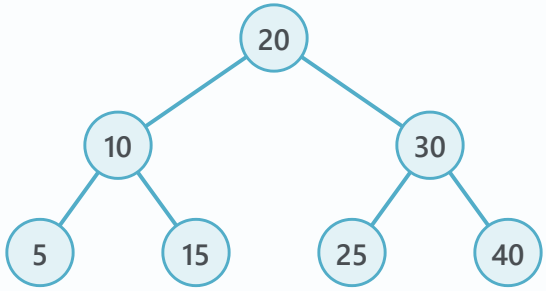
- **contains(19)**
- main()

**DEĞİŞKENLER**

`x = 19` `i = 5` `n = 4` `y = 19` `dönüş = true`

# Ağaç Tabanlı (Sıralı) Küme

- Küme bir **dengeli ikili arama ağacı** olarak temsil edilir (Java `TreeSet` : kırmızı-siyah ağaç).
- Öğeler **sıralı** saklanır; kök ortada (inorder) gezinme sıralı çıktı verir.
- Arama, ekleme, silme **O(log n)**.
- Öğeler karşılaştırılabilir olmalı ( `Comparable` veya `Comparator` ).
- `compare(a, b) == 0` ise `TreeSet` iki öğeyi aynı kabul eder; bu nedenle sıralama `equals` ile tutarlı olmalıdır.




**Sıralı yapının getirdiği ek işlemler** (S = {5, 10, 15, 20, 25, 30, 40}):

**first / last**
`5 / 40`

**floor(17)**
≤ 17 olan en büyük
`15`

**ceiling(17)**
≥ 17 olan en küçük
`20`

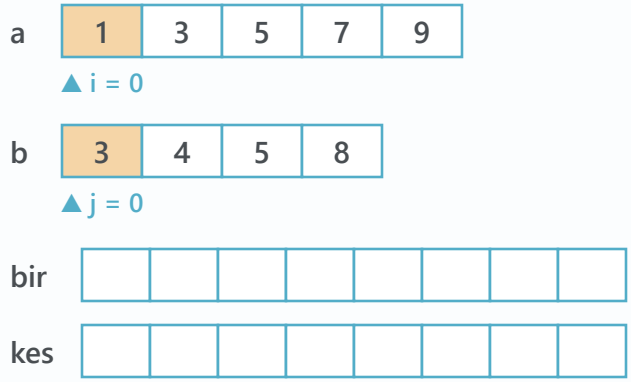
**subSet(10, 30)**
aralık sorgusu
`[10, 15, 20, 25]`

# Sıralı Dizilerle Birleşim ve Kesişim (iki imleç)

```
1  void birlesimKesisim(int[] a, int[] b,
2      List<Integer> bir, List<Integer> kes) {
▶  int i = 0, j = 0;
4    while (i < a.length && j < b.length) {
5      if (a[i] < b[j]) bir.add(a[i++]);
6      else if (a[i] > b[j]) bir.add(b[j++]);
7      else {                          // ortak öğe
8        bir.add(a[i]);  kes.add(a[i]);
9        i++;  j++;
10     }
11   }
12   while (i < a.length) bir.add(a[i++]);
13   while (j < b.length) bir.add(b[j++]);
14 }
```

**DEĞİŞKENLER**

`i = 0` `j = 0` `a[i] = 1` `b[j] = 3`




Kümeler **sıralı** dizilerde. Birleştirme sıralamasındaki *merge* gibi iki imleç tek geçişte ilerler (Goodrich: *generic merge*).

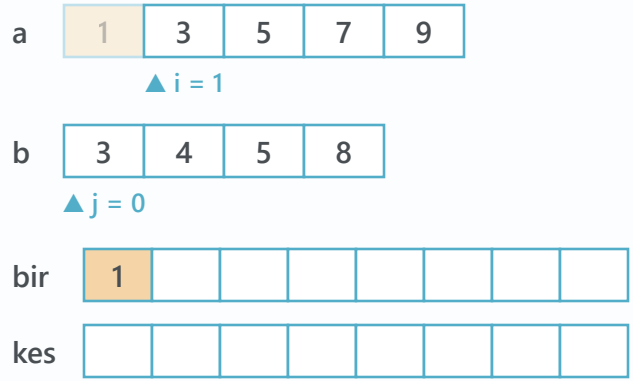
Adım 1 / 10

# Sıralı Dizilerle Birleşim ve Kesişim (iki imleç)

```
1  void birlesimKesisim(int[] a, int[] b,
2      List<Integer> bir, List<Integer> kes) {
3    int i = 0, j = 0;
4    while (i < a.length && j < b.length) {
▶      if (a[i] < b[j]) bir.add(a[i++]);
6      else if (a[i] > b[j]) bir.add(b[j++]);
7      else {                          // ortak öğe
8        bir.add(a[i]);  kes.add(a[i]);
9        i++;  j++;
10     }
11   }
12   while (i < a.length) bir.add(a[i++]);
13   while (j < b.length) bir.add(b[j++]);
14 }
```

**DEĞİŞKENLER**

`i = 0` `j = 0` `a[i] = 1` `b[j] = 3`




a[i] = 1 < b[j] = 3 → 1 yalnızca a'da: **birleşime** eklenir, i++.

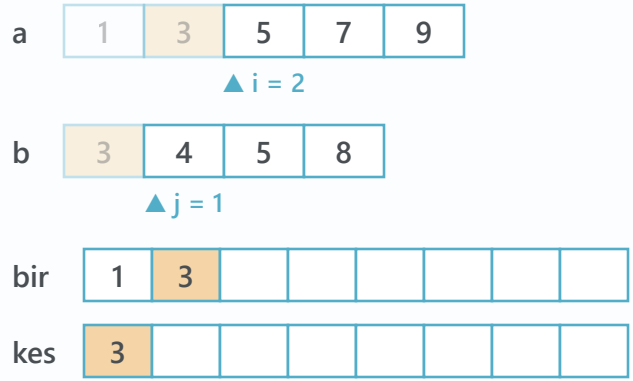
Adım 2 / 10

# Sıralı Dizilerle Birleşim ve Kesişim (iki imleç)

```
1  void birlesimKesisim(int[] a, int[] b,
2      List<Integer> bir, List<Integer> kes) {
3    int i = 0, j = 0;
4    while (i < a.length && j < b.length) {
5      if (a[i] < b[j]) bir.add(a[i++]);
6      else if (a[i] > b[j]) bir.add(b[j++]);
7      else {                          // ortak öğe
▶        bir.add(a[i]);  kes.add(a[i]);
▶        i++;  j++;
10     }
11   }
12   while (i < a.length) bir.add(a[i++]);
13   while (j < b.length) bir.add(b[j++]);
14 }
```

**DEĞİŞKENLER**

`i = 1` `j = 0` `a[i] = 3` `b[j] = 3`



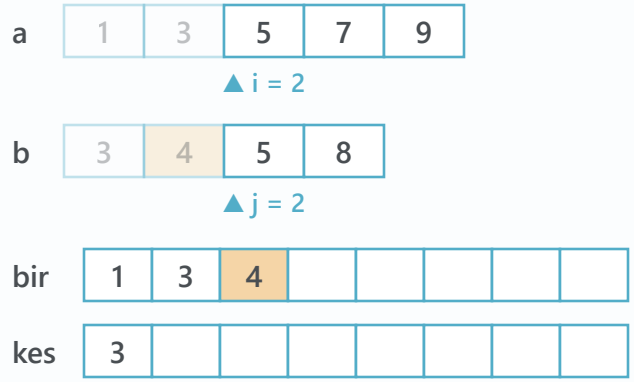

a[i] = b[j] = 3 → **ortak öğe**: birleşime bir kez ve kesişime eklenir; iki imleç de ilerler.

# Sıralı Dizilerle Birleşim ve Kesişim (iki imleç)

```
1  void birlesimKesisim(int[] a, int[] b,
2      List<Integer> bir, List<Integer> kes) {
3    int i = 0, j = 0;
4    while (i < a.length && j < b.length) {
5      if (a[i] < b[j]) bir.add(a[i++]);
▶      else if (a[i] > b[j]) bir.add(b[j++]);
7      else {                          // ortak öğe
8        bir.add(a[i]);  kes.add(a[i]);
9        i++;  j++;
10     }
11   }
12   while (i < a.length) bir.add(a[i++]);
13   while (j < b.length) bir.add(b[j++]);
14 }
```

**DEĞİŞKENLER**

`i = 2` `j = 1` `a[i] = 5` `b[j] = 4`



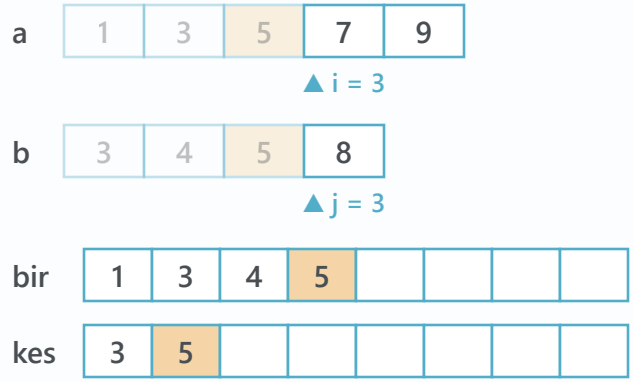

a[i] = 5 > b[j] = 4 → 4 yalnızca b'de: **birleşime** eklenir, j++.

# Sıralı Dizilerle Birleşim ve Kesişim (iki imleç)

```
1  void birlesimKesisim(int[] a, int[] b,
2      List<Integer> bir, List<Integer> kes) {
3    int i = 0, j = 0;
4    while (i < a.length && j < b.length) {
5      if (a[i] < b[j]) bir.add(a[i++]);
6      else if (a[i] > b[j]) bir.add(b[j++]);
7      else {                          // ortak öğe
▶        bir.add(a[i]);  kes.add(a[i]);
▶        i++;  j++;
10     }
11   }
12   while (i < a.length) bir.add(a[i++]);
13   while (j < b.length) bir.add(b[j++]);
14 }
```

**DEĞİŞKENLER**

`i = 2` `j = 2` `a[i] = 5` `b[j] = 5`



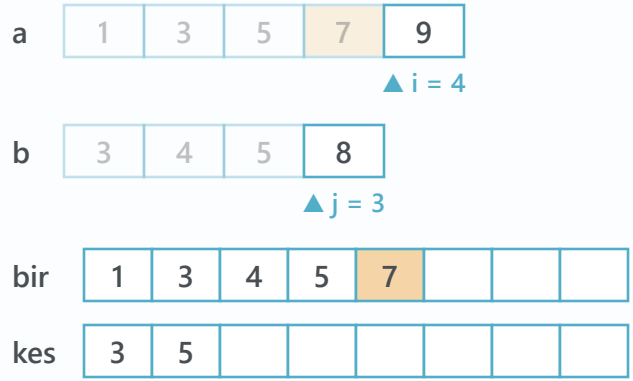

a[i] = b[j] = 5 → **ortak öğe**: birleşime bir kez ve kesişime eklenir; iki imleç de ilerler.

# Sıralı Dizilerle Birleşim ve Kesişim (iki imleç)

```
1  void birlesimKesisim(int[] a, int[] b,
2      List<Integer> bir, List<Integer> kes) {
3    int i = 0, j = 0;
4    while (i < a.length && j < b.length) {
▶      if (a[i] < b[j]) bir.add(a[i++]);
6      else if (a[i] > b[j]) bir.add(b[j++]);
7      else {                          // ortak öğe
8        bir.add(a[i]);  kes.add(a[i]);
9        i++;  j++;
10     }
11   }
12   while (i < a.length) bir.add(a[i++]);
13   while (j < b.length) bir.add(b[j++]);
14 }
```

**DEĞİŞKENLER**

`i = 3` `j = 3` `a[i] = 7` `b[j] = 8`



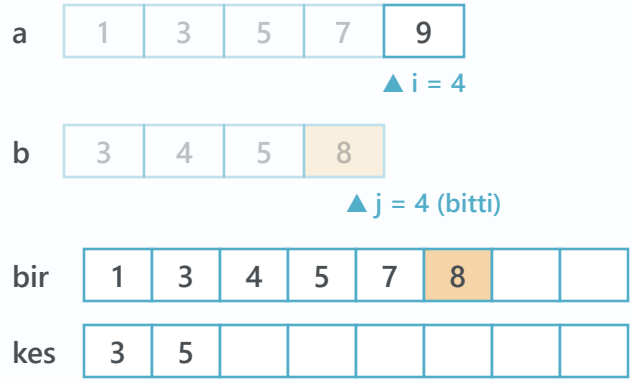

a[i] = 7 < b[j] = 8 → 7 yalnızca a'da: **birleşime** eklenir, i++.

# Sıralı Dizilerle Birleşim ve Kesişim (iki imleç)

```
1  void birlesimKesisim(int[] a, int[] b,
2      List<Integer> bir, List<Integer> kes) {
3    int i = 0, j = 0;
4    while (i < a.length && j < b.length) {
5      if (a[i] < b[j]) bir.add(a[i++]);
▶      else if (a[i] > b[j]) bir.add(b[j++]);
7      else {                          // ortak öğe
8        bir.add(a[i]);  kes.add(a[i]);
9        i++;  j++;
10     }
11   }
12   while (i < a.length) bir.add(a[i++]);
13   while (j < b.length) bir.add(b[j++]);
14 }
```

**DEĞİŞKENLER**

`i = 4` `j = 3` `a[i] = 9` `b[j] = 8`



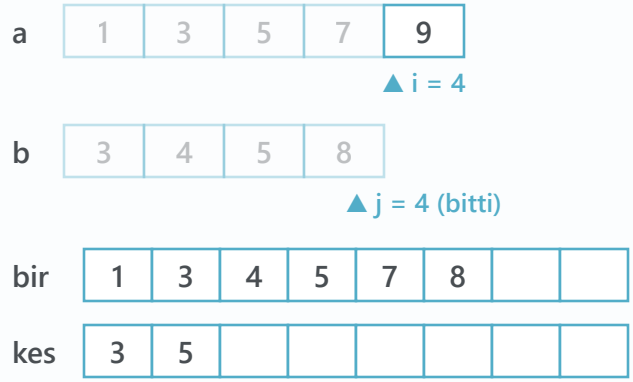

a[i] = 9 > b[j] = 8 → 8 yalnızca b'de: **birleşime** eklenir, j++.

# Sıralı Dizilerle Birleşim ve Kesişim (iki imleç)

```
1  void birlesimKesisim(int[] a, int[] b,
2      List<Integer> bir, List<Integer> kes) {
3    int i = 0, j = 0;
▶    while (i < a.length && j < b.length) {
5      if (a[i] < b[j]) bir.add(a[i++]);
6      else if (a[i] > b[j]) bir.add(b[j++]);
7      else {                          // ortak öğe
8        bir.add(a[i]);  kes.add(a[i]);
9        i++;  j++;
10     }
11   }
12   while (i < a.length) bir.add(a[i++]);
13   while (j < b.length) bir.add(b[j++]);
14 }
```

**DEĞİŞKENLER**

`i = 4` `j = 4` `a[i] = 9`



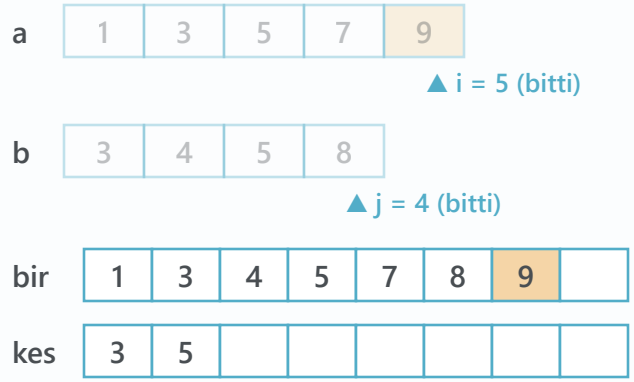

j = 4 = b.length → b bitti, ana döngüden çıkılır. Kesişim tamam: artık ortak öğe olamaz.

# Sıralı Dizilerle Birleşim ve Kesişim (iki imleç)

```
1  void birlesimKesisim(int[] a, int[] b,
2      List<Integer> bir, List<Integer> kes) {
3    int i = 0, j = 0;
4    while (i < a.length && j < b.length) {
5      if (a[i] < b[j]) bir.add(a[i++]);
6      else if (a[i] > b[j]) bir.add(b[j++]);
7      else {                          // ortak öğe
8        bir.add(a[i]);  kes.add(a[i]);
9        i++;  j++;
10     }
11   }
▶  12  while (i < a.length) bir.add(a[i++]);
13   while (j < b.length) bir.add(b[j++]);
14 }
```

**DEĞİŞKENLER**

`i = 4` `j = 4` `a[i] = 9`



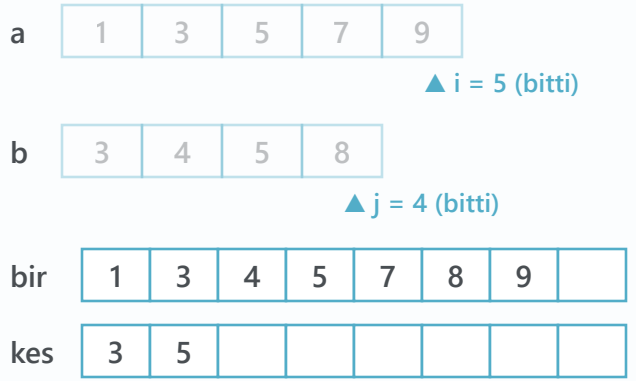

a'da kalan 9 doğrudan birleşime eklenir, i++.

# Sıralı Dizilerle Birleşim ve Kesişim (iki imleç)

```
1  void birlesimKesisim(int[] a, int[] b,
2      List<Integer> bir, List<Integer> kes) {
3    int i = 0, j = 0;
4    while (i < a.length && j < b.length) {
5      if (a[i] < b[j]) bir.add(a[i++]);
6      else if (a[i] > b[j]) bir.add(b[j++]);
7      else {                          // ortak öğe
8        bir.add(a[i]);  kes.add(a[i]);
9        i++;  j++;
10     }
11   }
12   while (i < a.length) bir.add(a[i++]);
▶    while (j < b.length) bir.add(b[j++]);
14 }
```

**DEĞİŞKENLER**

`i = 5` `j = 4`




b'de kalan yok. Birleşim [1, 3, 4, 5, 7, 8, 9], kesişim [3, 5]. Her adımda bir imleç ilerler → en fazla n + m adım: **O(n + m)**.

KÜME UYGULAMALARI

# Küme Uygulamalarının Karşılaştırması

| İşlem (\|A\| = n, \|B\| = m) | Sırasız liste | Sıralı dizi | Hash tablosu | Dengeli BST |
|---|---|---|---|---|
| `contains` | O(n) | O(log n) | **O(1)*** | O(log n) |
| `add` / `remove` | O(n)† | O(n) | **O(1)*** | O(log n) |
| A ∪ B, A ∩ B, A − B | O(n·m) | O(n + m) | O(n + m)* | O(m log(n+m)) |
| Sıralı gezinme | O(n log n) | **O(n)** | O(n log n) | **O(n)** |
| En küçük / floor / aralık | O(n) | O(log n) | O(n) | O(log n) |

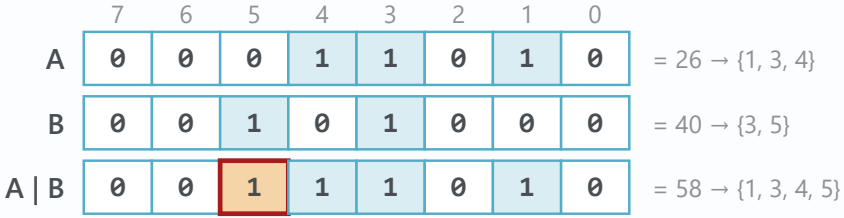
* ortalama durum (en kötü O(n)). † tekrar kontrolü için önce arama gerekir. Hash ile birleşim: B'nin her öğesi için `contains` + `add`.

**Varsayılan seçim: hash**
Sıra gerekmiyorsa en hızlı üyelik testi.

**Sıra gerekiyorsa: ağaç**
Her işlem O(log n) ve sıralı sorgular bedava.

KÜME UYGULAMALARI

# Bit Vektörü ile Küme (Bitset)




**Fikir**
Evren 0..U−1 ise i. bit 1 ⇔ i kümede.

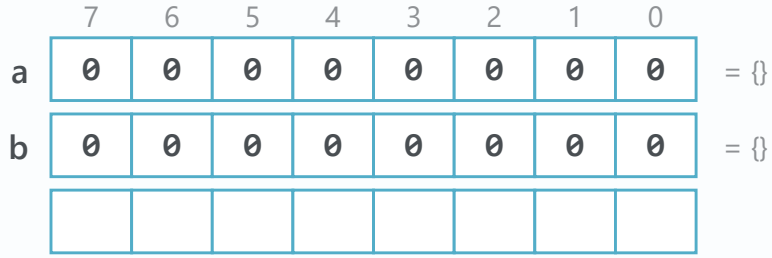
**Tek öğe: O(1)**
ekle, sil, üyelik = tek bit işlemi.

**Küme cebri: O(U / 64)**
∪ ∩ − kelime kelime `|` `&` `&~`.

| İşlem | int maskesi (U ≤ 32) | java.util.BitSet |
|---|---|---|
| ekle x / var mı x? | `a \|= 1 << x` · `(a >> x & 1) == 1` | `set(x)` · `get(x)` |
| birleşim / kesişim | `a \| b` · `a & b` | `or(b)` · `and(b)` |
| fark / öğe sayısı | `a & ~b` · `Integer.bitCount(a)` | `andNot(b)` · `cardinality()` |

Bellek U bit: $10^6$ öğelik evren ≈ 122 KB. Evren büyük ama küme seyrekse (ör. 3 öğe, U = $10^9$) hash tabanlı küme seçilir.

# Kod İzleme: bit işlemleriyle küme cebri




İki boş küme: tüm bitler 0.

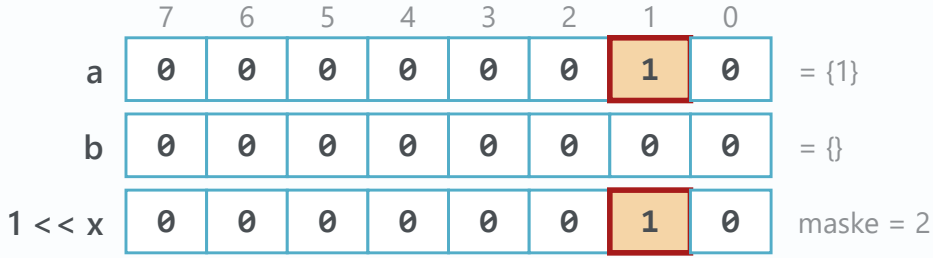
Adım 1 / 10

```
▶ int a = 0, b = 0;
2 for (int x : new int[]{1, 3, 4})
3   a |= 1 << x;          // x. biti yak
4 for (int x : new int[]{3, 5})
5   b |= 1 << x;
6 int bir = a | b;        // birleşim
7 int kes = a & b;        // kesişim
8 int fark = a & ~b;      // fark
9 boolean var3 = (a >> 3 & 1) == 1;
```

**DEĞİŞKENLER**

- a = 0 (00000000)
- b = 0 (00000000)

# Kod İzleme: bit işlemleriyle küme cebri




`1 << 1` = 2 maskesi; `a |= maske` → 1. bit 1 olur: 1 kümeye eklendi.

Adım 2 / 10

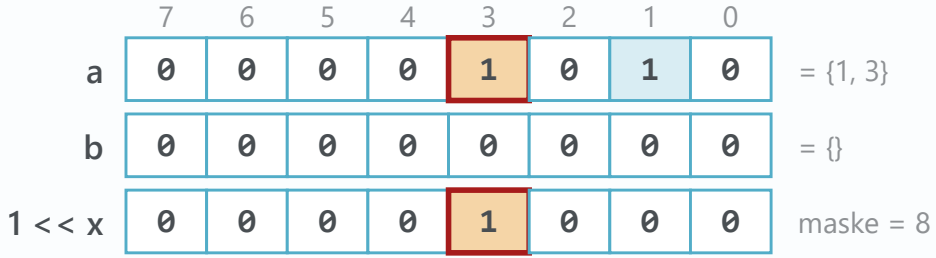
```
1 int a = 0, b = 0;
2 for (int x : new int[]{1, 3, 4})
▶   a |= 1 << x;          // x. biti yak
4 for (int x : new int[]{3, 5})
5   b |= 1 << x;
6 int bir = a | b;        // birleşim
7 int kes = a & b;        // kesişim
8 int fark = a & ~b;      // fark
9 boolean var3 = (a >> 3 & 1) == 1;
```

**DEĞİŞKENLER**

`a = 2 (00000010)`

`b = 0 (00000000)` `x = 1`

# Kod İzleme: bit işlemleriyle küme cebri




`1 << 3` = 8 maskesi; `a |= maske` → 3. bit 1 olur: 3 kümeye eklendi.

Adım 3 / 10

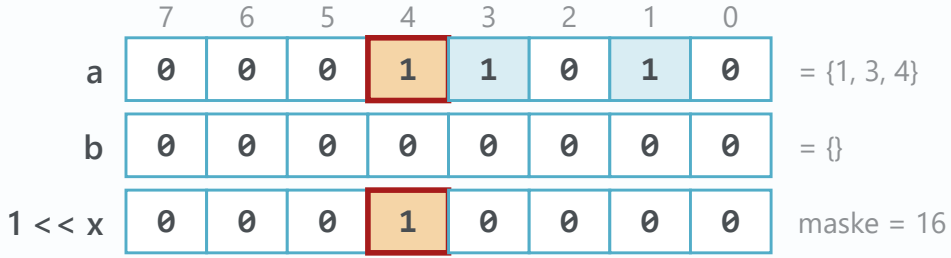
```
1 int a = 0, b = 0;
2 for (int x : new int[]{1, 3, 4})
▶   a |= 1 << x;          // x. biti yak
4 for (int x : new int[]{3, 5})
5   b |= 1 << x;
6 int bir = a | b;        // birleşim
7 int kes = a & b;        // kesişim
8 int fark = a & ~b;      // fark
9 boolean var3 = (a >> 3 & 1) == 1;
```

**DEĞİŞKENLER**

`a = 10 (00001010)`

`b = 0 (00000000)` `x = 3`

# Kod İzleme: bit işlemleriyle küme cebri




`1 << 4` = 16 maskesi; `a |= maske` → 4. bit 1 olur: 4 kümeye eklendi.

Adım 4 / 10

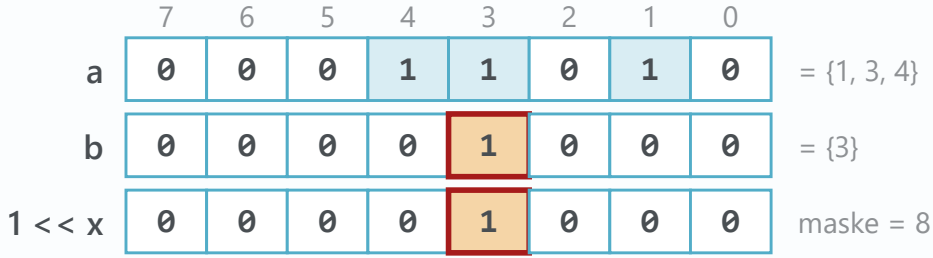
```
1 int a = 0, b = 0;
2 for (int x : new int[]{1, 3, 4})
▶   a |= 1 << x;          // x. biti yak
4 for (int x : new int[]{3, 5})
5   b |= 1 << x;
6 int bir = a | b;        // birleşim
7 int kes = a & b;        // kesişim
8 int fark = a & ~b;      // fark
9 boolean var3 = (a >> 3 & 1) == 1;
```

**DEĞİŞKENLER**

`a = 26 (00011010)`

`b = 0 (00000000)` `x = 4`

# Kod İzleme: bit işlemleriyle küme cebri




b kümesine 3 eklenir: 3. bit yanar.

Adım 5 / 10

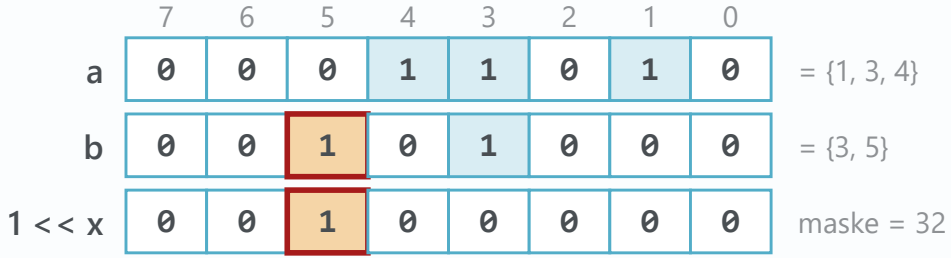
```
1 int a = 0, b = 0;
2 for (int x : new int[]{1, 3, 4})
3   a |= 1 << x;          // x. biti yak
4 for (int x : new int[]{3, 5})
▶   b |= 1 << x;
6 int bir = a | b;        // birleşim
7 int kes = a & b;        // kesişim
8 int fark = a & ~b;      // fark
9 boolean var3 = (a >> 3 & 1) == 1;
```

**DEĞİŞKENLER**

a = **26 (00011010)**

b = **8 (00001000)** x = **3**

# Kod İzleme: bit işlemleriyle küme cebri




b kümesine 5 eklenir: 5. bit yanar.

Adım 6 / 10

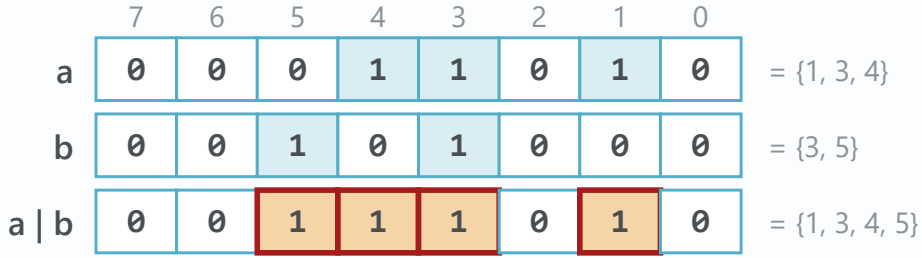
```
1 int a = 0, b = 0;
2 for (int x : new int[]{1, 3, 4})
3   a |= 1 << x;          // x. biti yak
4 for (int x : new int[]{3, 5})
▶   b |= 1 << x;
6 int bir = a | b;        // birleşim
7 int kes = a & b;        // kesişim
8 int fark = a & ~b;      // fark
9 boolean var3 = (a >> 3 & 1) == 1;
```

**DEĞİŞKENLER**

a = **26 (00011010)**

b = **40 (00101000)** x = **5**

# Kod İzleme: bit işlemleriyle küme cebri




**VEYA**: bitlerden biri 1 ise 1 → birleşim {1, 3, 4, 5}. Tek makine komutu.

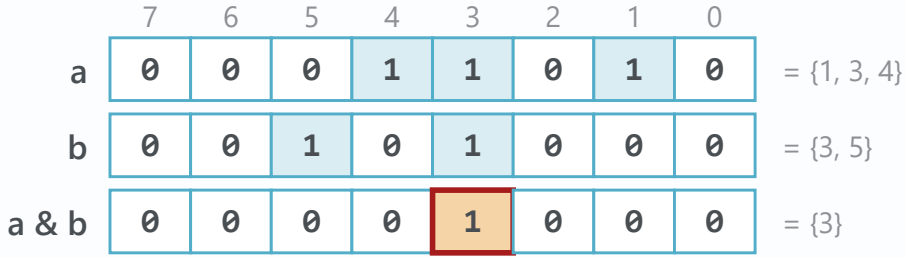
Adım 7 / 10

```
1 int a = 0, b = 0;
2 for (int x : new int[]{1, 3, 4})
3   a |= 1 << x;          // x. biti yak
4 for (int x : new int[]{3, 5})
5   b |= 1 << x;
▶ int bir = a | b;        // birleşim
7 int kes = a & b;        // kesişim
8 int fark = a & ~b;      // fark
9 boolean var3 = (a >> 3 & 1) == 1;
```

**DEĞİŞKENLER**

- a = **26 (00011010)**
- b = **40 (00101000)**
- bir = **58 (00111010)**

# Kod İzleme: bit işlemleriyle küme cebri




**VE**: iki bit de 1 ise 1 → kesişim {3}.

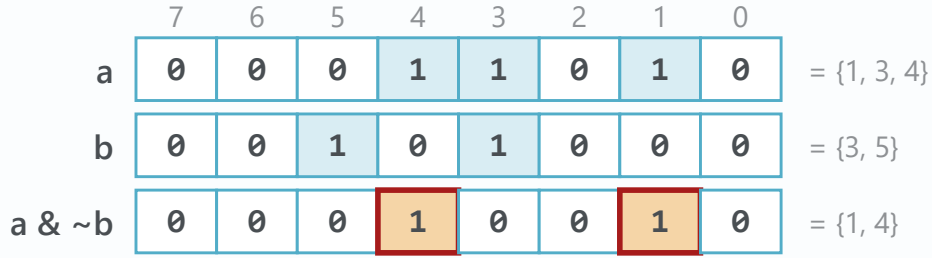
Adım 8 / 10

```
1 int a = 0, b = 0;
2 for (int x : new int[]{1, 3, 4})
3   a |= 1 << x;          // x. biti yak
4 for (int x : new int[]{3, 5})
5   b |= 1 << x;
6 int bir = a | b;        // birleşim
▶ int kes = a & b;        // kesişim
8 int fark = a & ~b;      // fark
9 boolean var3 = (a >> 3 & 1) == 1;
```

**DEĞİŞKENLER**

- a = 26 (00011010)
- b = 40 (00101000)
- bir = 58 (00111010)
- kes = 8 (00001000)

# Kod İzleme: bit işlemleriyle küme cebri




`~b` b'nin bitlerini ters çevirir; `a & ~b` : a'da olup b'de olmayanlar → fark {1, 4}.

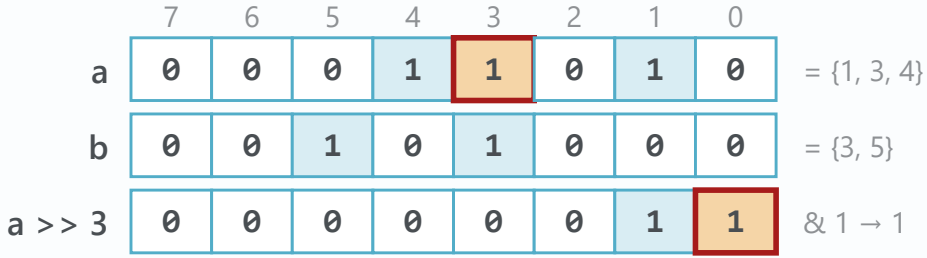
Adım 9 / 10

```
1 int a = 0, b = 0;
2 for (int x : new int[]{1, 3, 4})
3   a |= 1 << x;          // x. biti yak
4 for (int x : new int[]{3, 5})
5   b |= 1 << x;
6 int bir = a | b;        // birleşim
7 int kes = a & b;        // kesişim
▶ int fark = a & ~b;      // fark
9 boolean var3 = (a >> 3 & 1) == 1;
```

**DEĞİŞKENLER**

- `a = 26 (00011010)`
- `b = 40 (00101000)`
- `bir = 58 (00111010)`
- `kes = 8 (00001000)`
- `fark = 18 (00010010)`

# Kod İzleme: bit işlemleriyle küme cebri




Üyelik testi: `a >> 3` 3. biti en sağa kaydırır, `& 1` onu ayırır → 1, yani **3** ∈ **a**. O(1).

Adım 10 / 10

```
1  int a = 0, b = 0;
2  for (int x : new int[]{1, 3, 4})
3    a |= 1 << x;          // x. biti yak
4  for (int x : new int[]{3, 5})
5    b |= 1 << x;
6  int bir = a | b;        // birleşim
7  int kes = a & b;        // kesişim
8  int fark = a & ~b;      // fark
▶  boolean var3 = (a >> 3 & 1) == 1;
```

**DEĞİŞKENLER**

- `a = 26 (00011010)`
- `b = 40 (00101000)`
- `bir = 58 (00111010)`
- `kes = 8 (00001000)`
- `fark = 18 (00010010)`
- `var3 = true`

# Çoklu Küme (Multiset / Bag)

- **Çoklu küme** (multiset, *bag*): aynı öğenin **birden çok kez** bulunabildiği küme.
- Sıra yine önemsizdir; önemli olan her öğenin **kaç kez** geçtiğidir (çokluk, multiplicity).
- Doğal uygulama: **öğe** → **sayaç** eşlemesi.
- Örnek: bir metindeki kelime frekansları, alışveriş sepeti, oy sayımı.

| Öğe | Sayaç |
|---|---|
| a | 2 |
| b | 1 |
| c | 3 |

M = {a, a, b, c, c, c} → eşleme {a:2, b:1, c:3}

| Multiset ADT (Goodrich) | Anlamı | Hash ile |
|---|---|---|
| `add(e)` / `add(e, n)` | e'yi 1 / n kez ekle (sayaç += n) | O(1)* |
| `count(e)` | e kaç kez var? (yoksa 0) | O(1)* |
| `remove(e)` / `remove(e, n)` | sayacı azalt; 0 olursa girdiyi sil | O(1)* |
| `size()` | tekrarlar dahil öğe sayısı (örnekte 6) | O(1) |

ÇOKLU KÜME

# Java ile Çoklu Küme

```
class CokluKume<E> {
    private final Map<E, Integer> say = new HashMap<>();
    private int n = 0;                                  // tekrarlar dahil öğe sayısı
    void add(E e) { say.merge(e, 1, Integer::sum); n++; }  // yoksa 1, varsa +1
    int count(E e) { return say.getOrDefault(e, 0); }
    boolean remove(E e) {
        Integer c = say.get(e);
        if (c == null) return false;                    // e yok
        if (c == 1) say.remove(e);                         // sayaç 0 olacak: girdiyi sil
        else say.put(e, c - 1);
        n--;
        return true;
    }
    int size() { return n; }
}
```

### Kullanım

`add("a"); add("a");`
`count("a") → 2`

### Küme ile farkı

Set'te ikinci `add("a")` etkisiz; burada sayaç artar.

### Sıralı çoklu küme

`TreeMap<E, Integer>` ile sıralı gezinme.

# Kod İzleme: `CokluKume` sayaçları

```
1  void add(E e) {
▶    say.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return say.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
8    Integer c = say.get(e);
9    if (c == null) return false;
10   if (c == 1) say.remove(e);
11   else say.put(e, c - 1);
12   n--;
13   return true;
14 }
```

ÇAĞRI YIĞINI

- **add("a")**
- main()

say (öğe → sayaç)

| öğe | sayaç |
|---|---|
| "a" | 1 |

n

1

(tekrar dahil)

M = {a}

`merge` : "a" yoktu → sayaç **1** ile eklenir; n = 1.

Adım 1 / 13

DEĞİŞKENLER

e = **"a"** n = **1**

# Kod İzleme: `CokluKume` sayaçları

```
1  void add(E e) {
▶    say.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return say.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
8    Integer c = say.get(e);
9    if (c == null) return false;
10   if (c == 1) say.remove(e);
11   else say.put(e, c - 1);
12   n--;
13   return true;
14 }
```

ÇAĞRI YIĞINI

- **`add("b")`**
- `main()`

say (öğe → sayaç)

| öğe | sayaç |
|---|---|
| "a" | 1 |
| "b" | 1 |

n

**2**

(tekrar dahil)

M = {a, b}

`merge` : "b" yoktu → sayaç **1** ile eklenir; n = 2.

Adım 2 / 13

DEĞİŞKENLER

`e = "b"` `n = 2`

# Kod İzleme: `CokluKume` sayaçları

```
1  void add(E e) {
▶    say.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return say.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
8    Integer c = say.get(e);
9    if (c == null) return false;
10   if (c == 1) say.remove(e);
11   else say.put(e, c - 1);
12   n--;
13   return true;
14 }
```

ÇAĞRI YIĞINI

- **add("a")**
- main()

say (öğe → sayaç)

| öğe | sayaç |
|---|---|
| "a" | 2 |
| "b" | 1 |

n

3

(tekrar dahil)

M = {a, a, b}

`merge` : "a" vardı → `Integer::sum` ile 1 + 1 = **2**; n = 3. Set olsaydı değişmezdi.

Adım 3 / 13

DEĞİŞKENLER

`e = "a"` `n = 3`

# Kod İzleme: `CokluKume` sayaçları

```
1  void add(E e) {
▶    say.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return say.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
8    Integer c = say.get(e);
9    if (c == null) return false;
10   if (c == 1) say.remove(e);
11   else say.put(e, c - 1);
12   n--;
13   return true;
14 }
```

ÇAĞRI YIĞINI

- **add("c")**
- main()

say (öğe → sayaç)

| öğe | sayaç |
|---|---|
| "a" | 2 |
| "b" | 1 |
| "c" | 1 |

M = {a, a, b, c}

n

4

(tekrar dahil)

`merge` : "c" yoktu → sayaç **1** ile eklenir; n = 4.

Adım 4 / 13

DEĞİŞKENLER

e = "c"  n = 4

# Kod İzleme: `CokluKume` sayaçları

```
1  void add(E e) {
2    say.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
▶    return say.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
8    Integer c = say.get(e);
9    if (c == null) return false;
10   if (c == 1) say.remove(e);
11   else say.put(e, c - 1);
12   n--;
13   return true;
14 }
```

ÇAĞRI YIĞINI

- `count("a")`
- `main()`

say (öğe → sayaç)

| öğe | sayaç |
|---|---|
| "a" | 2 |
| "b" | 1 |
| "c" | 1 |

n = 4 (tekrar dahil)

M = {a, a, b, c}

count("a"): `getOrDefault` → **2**. Tek hash araması: O(1).

Adım 5 / 13

DEĞİŞKENLER

`e = "a"` `dönüş = 2`

# Kod İzleme: `CokluKume` sayaçları

```
1  void add(E e) {
2    say.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return say.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
▶    Integer c = say.get(e);
9    if (c == null) return false;
10   if (c == 1) say.remove(e);
11   else say.put(e, c - 1);
12   n--;
13   return true;
14 }
```

ÇAĞRI YIĞINI

- **`remove("a")`**
- `main()`

say (öğe → sayaç)

| öğe | sayaç |
|---|---|
| "a" | 2 |
| "b" | 1 |
| "c" | 1 |

n

4

(tekrar dahil)

M = {a, a, b, c}

remove("a"): `c = say.get(e)` = **2**.

Adım 6 / 13

DEĞİŞKENLER

`e = "a"` `c = 2` `n = 4`

# Kod İzleme: `CokluKume` sayaçları

```
1  void add(E e) {
2    say.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return say.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
8    Integer c = say.get(e);
9    if (c == null) return false;
10   if (c == 1) say.remove(e);
▶    else say.put(e, c - 1);
▶    n--;
▶    return true;
14 }
```

ÇAĞRI YIĞINI

- **`remove("a")`**
- `main()`

say (öğe → sayaç)

| öğe | sayaç |
|---|---|
| "a" | 1 |
| "b" | 1 |
| "c" | 1 |

n

3

(tekrar dahil)

M = {a, b, c}

c = 2 > 1 → sayaç **1**'e iner (girdi kalır); n = 3.

Adım 7 / 13

DEĞİŞKENLER

`e = "a"` `c = 2` `n = 3` `dönüş = true`

# Kod İzleme: `CokluKume` sayaçları

```
1  void add(E e) {
2    say.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return say.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
▶    Integer c = say.get(e);
9    if (c == null) return false;
10   if (c == 1) say.remove(e);
11   else say.put(e, c - 1);
12   n--;
13   return true;
14 }
```

ÇAĞRI YIĞINI

- **`remove("b")`**
- `main()`

say (öğe → sayaç)

| öğe | sayaç |
|---|---|
| "a" | 1 |
| "b" | 1 |
| "c" | 1 |

n

**3**

(tekrar dahil)

M = {a, b, c}

remove("b"): `c = say.get(e)` = **1**.

Adım 8 / 13

DEĞİŞKENLER

`e = "b"` `c = 1` `n = 3`

# Kod İzleme: `CokluKume` sayaçları

```
1  void add(E e) {
2    say.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return say.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
8    Integer c = say.get(e);
9    if (c == null) return false;
▶    if (c == 1) say.remove(e);
11   else say.put(e, c - 1);
▶    n--;
▶    return true;
14 }
```

ÇAĞRI YIĞINI

- **`remove("b")`**
- `main()`

say (öğe → sayaç)

| öğe | sayaç |
|---|---|
| "a" | 1 |
| "c" | 1 |
| "b" | 0 (silindi) |

n

**2**

(tekrar dahil)

M = {a, c}

c == 1 → sayaç 0 olacaktı: girdi **tamamen silinir** (0 sayaç saklanmaz); n = 2.

Adım 9 / 13

DEĞİŞKENLER

`e = "b"` `c = 1` `n = 2` `dönüş = true`

# Kod İzleme: `CokluKume` sayaçları

```
1  void add(E e) {
2    say.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return say.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
▶    Integer c = say.get(e);
9    if (c == null) return false;
10   if (c == 1) say.remove(e);
11   else say.put(e, c - 1);
12   n--;
13   return true;
14 }
```

**ÇAĞRI YIĞINI**

- **`remove("x")`**
- `main()`

say (öğe → sayaç)

| öğe | sayaç |
|---|---|
| "a" | 1 |
| "c" | 1 |

"x" eşlemede yok

n

2

(tekrar dahil)

M = {a, c}

remove("x"): `c = say.get(e)` = **null**.

Adım 10 / 13

**DEĞİŞKENLER**

`e = "x"` `c = null` `n = 2`

# Kod İzleme: `CokluKume` sayaçları

```
1  void add(E e) {
2    say.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return say.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
8    Integer c = say.get(e);
▶    if (c == null) return false;
10   if (c == 1) say.remove(e);
11   else say.put(e, c - 1);
12   n--;
13   return true;
14 }
```

**ÇAĞRI YIĞINI**

- `remove("x")`
- `main()`

say (öğe → sayaç)

| öğe | sayaç |
|---|---|
| "a" | 1 |
| "c" | 1 |

"x" eşlemede yok

n = 2 (tekrar dahil)

M = {a, c}

c == null → öğe yok: `return false`, n değişmez.

Adım 11 / 13

**DEĞİŞKENLER**

`e = "x"` `c = null` `n = 2` `dönüş = false`

# Kod İzleme: `CokluKume` sayaçları

```
1  void add(E e) {
2    say.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
▶    return say.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
8    Integer c = say.get(e);
9    if (c == null) return false;
10   if (c == 1) say.remove(e);
11   else say.put(e, c - 1);
12   n--;
13   return true;
14 }
```

ÇAĞRI YIĞINI

- **`count("b")`**
- `main()`

say (öğe → sayaç)

| öğe | sayaç |
|---|---|
| "a" | 1 |
| "c" | 1 |

"b" eşlemede yok

n

2

(tekrar dahil)

M = {a, c}

count("b"): `getOrDefault` → **0**. Tek hash araması: O(1).

Adım 12 / 13

DEĞİŞKENLER

`e = "b"` `dönüş = 0`

# Kod İzleme: `CokluKume` sayaçları

```
1  void add(E e) {
2    say.merge(e, 1, Integer::sum);  n++;
3  }
4  int count(E e) {
5    return say.getOrDefault(e, 0);
6  }
7  boolean remove(E e) {
8    Integer c = say.get(e);
9    if (c == null) return false;
10   if (c == 1) say.remove(e);
11   else say.put(e, c - 1);
12   n--;
13   return true;
▶ }
```

ÇAĞRI YIĞINI

`main()`

say (öğe → sayaç)

| öğe | sayaç |
|---|---|
| "a" | 1 |
| "c" | 1 |

n

2

(tekrar dahil)

M = {a, c}

Sonuç M = {a, c}, n = 2. Tüm işlemler tek hash erişimi → ortalama **O(1)**.

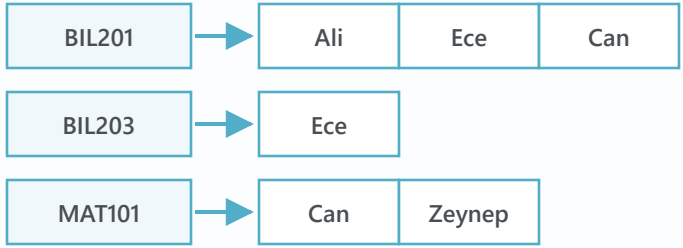
Adım 13 / 13

DEĞİŞKENLER

n = 2

# Çoklu Eşleme (Multimap)

- **Çoklu eşleme** (multimap): bir anahtar **birden çok değere** eşlenebilir (ders → öğrenciler).
- Goodrich ADT: `get(k)` tüm değerleri, `put(k, v)`, `remove(k, v)`, `removeAll(k)`, `size()` tüm (k, v) çiftleri.
- Java'da hazır sınıf yok: `Map<K, List<V>>` ile kurulur.




```
Map<String, List<String>> m = new HashMap<>();
void put(String k, String v) {   // yoksa boş liste aç
  m.computeIfAbsent(k, x -> new ArrayList<>()).add(v);
}
List<String> get(String k) {
  return m.getOrDefault(k, List.of());
}
```

Çoklu kümeden farkı: yalnız *kaç kez* değil, değerlerin kendisi tutulur. Hash ile `put` / `get` O(1)*, `remove(k, v)` O(1 + liste boyu).

# Ayrık Kümeler ve Union-Find

- **Ayrık kümeler** (disjoint sets): hiçbir öğesi ortak olmayan kümeler topluluğu. Her öğe tam olarak **bir** kümeye aittir.
- **Union-Find** (Disjoint Set Union, DSU) bu kümeleri tutan veri yapısıdır. İki ana işlemi vardır:

**find(x) · bul**

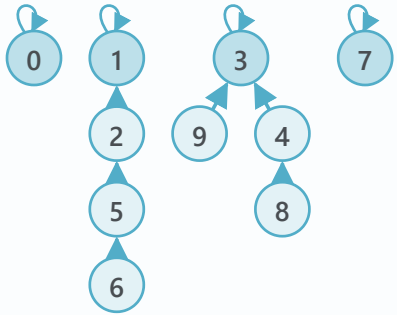
x'in ait olduğu kümenin *temsilcisini* (kökünü) döndürür.

`find(x) == find(y) ⇔ aynı küme`

**union(x, y) · birleştir**

x'in ve y'nin kümelerini tek kümede birleştirir.

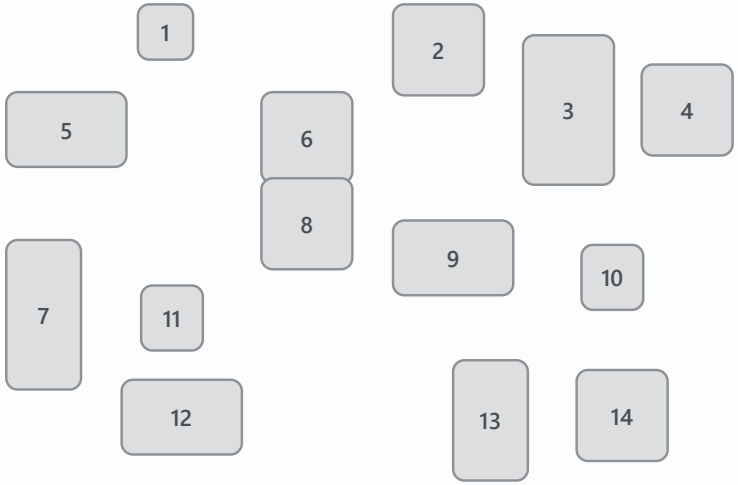
`{1,2} ∪ {3} → {1,2,3}`




Her küme bir ağaç: her düğüm ebeveynini (parent) gösterir; kök kendini gösterir ve kümenin temsilcisidir.

# Mıknatıs Örneği

- Birbirini güçlü şekilde çeken mıknatıslar düşünelim: iki mıknatıs değince **ayrılmaz**.
- Birleşmeler sürerken "hangi mıknatıs hangi grupta?" sorusunu izlemek istiyoruz.
- Önce her mıknatısa bir **numara** verilir.
- Birleşme = `union`, "aynı grupta mı?" = iki `find` karşılaştırması.

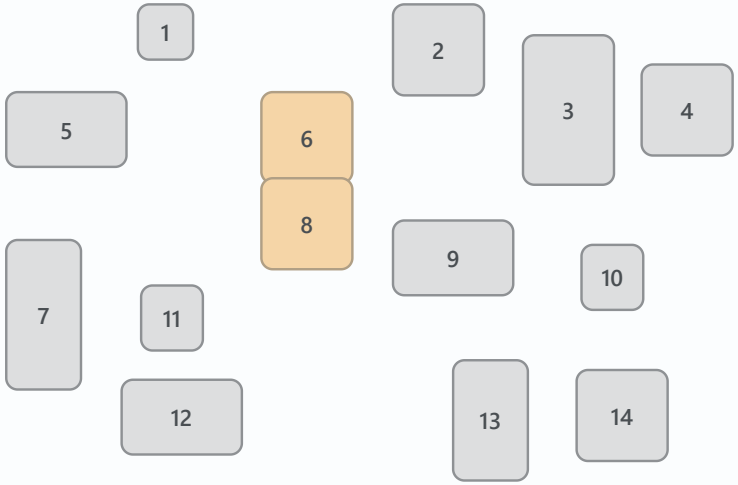



14 mıknatıs, her biri kendi grubunda: 14 ayrık küme. find(x) = x.

Adım 1 / 8

# Mıknatıs Örneği

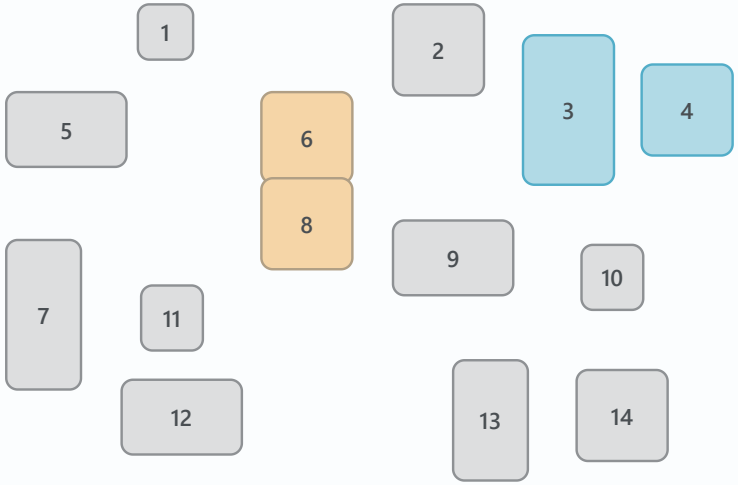
- Birbirini güçlü şekilde çeken mıknatıslar düşünelim: iki mıknatıs değince **ayrılmaz**.
- Birleşmeler sürerken "hangi mıknatıs hangi grupta?" sorusunu izlemek istiyoruz.
- Önce her mıknatısa bir **numara** verilir.
- Birleşme = `union`, "aynı grupta mı?" = iki `find` karşılaştırması.




**union(6, 8)**: en yakın olan 6 ile 8 birleşir; artık find(6) = find(8).

# Mıknatıs Örneği

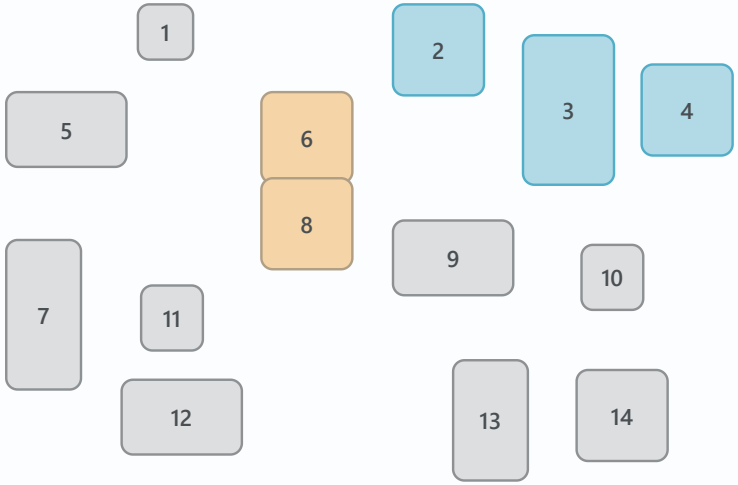
- Birbirini güçlü şekilde çeken mıknatıslar düşünelim: iki mıknatıs değince **ayrılmaz**.
- Birleşmeler sürerken "hangi mıknatıs hangi grupta?" sorusunu izlemek istiyoruz.
- Önce her mıknatısa bir **numara** verilir.
- Birleşme = `union`, "aynı grupta mı?" = iki `find` karşılaştırması.




**union(3, 4)**: 3 ile 4 birbirini çeker, yeni grup {3, 4}.

# Mıknatıs Örneği

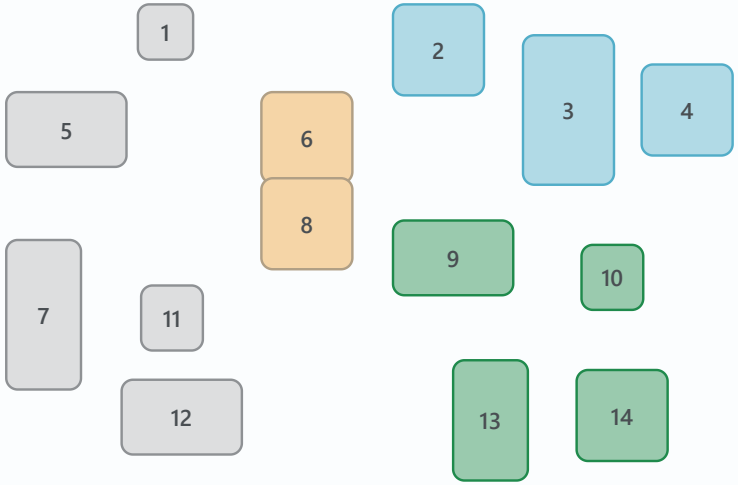
- Birbirini güçlü şekilde çeken mıknatıslar düşünelim: iki mıknatıs değince **ayrılmaz**.
- Birleşmeler sürerken "hangi mıknatıs hangi grupta?" sorusunu izlemek istiyoruz.
- Önce her mıknatısa bir **numara** verilir.
- Birleşme = `union`, "aynı grupta mı?" = iki `find` karşılaştırması.




**union(2, 3)**: 2, {3, 4} grubuna katılır → {2, 3, 4}.

# Mıknatıs Örneği

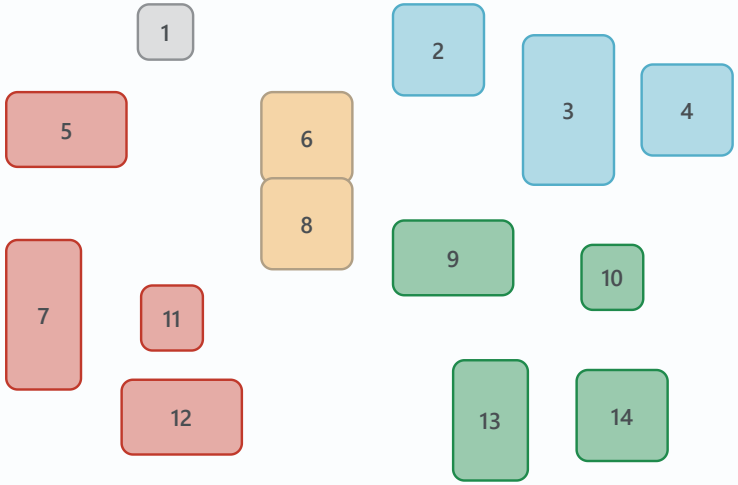
- Birbirini güçlü şekilde çeken mıknatıslar düşünelim: iki mıknatıs değince **ayrılmaz**.
- Birleşmeler sürerken "hangi mıknatıs hangi grupta?" sorusunu izlemek istiyoruz.
- Önce her mıknatısa bir **numara** verilir.
- Birleşme = `union`, "aynı grupta mı?" = iki `find` karşılaştırması.




**union(9, 10), union(13, 14), union(9, 13)** → {9, 10, 13, 14}.

# Mıknatıs Örneği

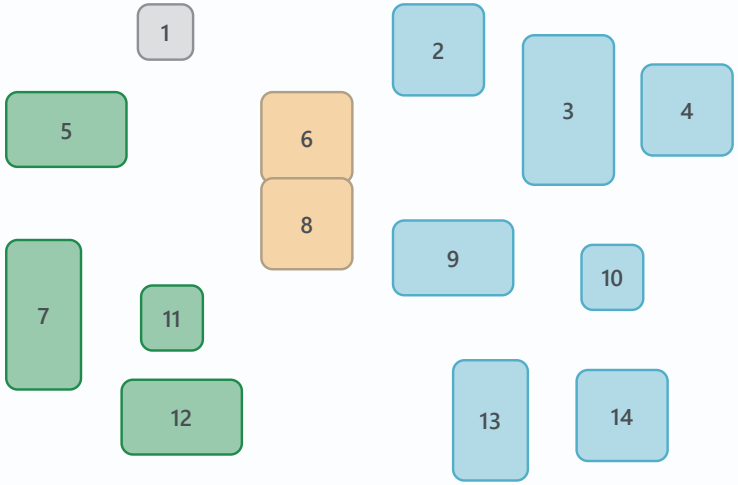
- Birbirini güçlü şekilde çeken mıknatıslar düşünelim: iki mıknatıs değince **ayrılmaz**.
- Birleşmeler sürerken "hangi mıknatıs hangi grupta?" sorusunu izlemek istiyoruz.
- Önce her mıknatısa bir **numara** verilir.
- Birleşme = `union`, "aynı grupta mı?" = iki `find` karşılaştırması.




**union(5, 7), union(7, 11), union(11, 12)** → {5, 7, 11, 12}. Beş grup kaldı (1 tek başına).

# Mıknatıs Örneği

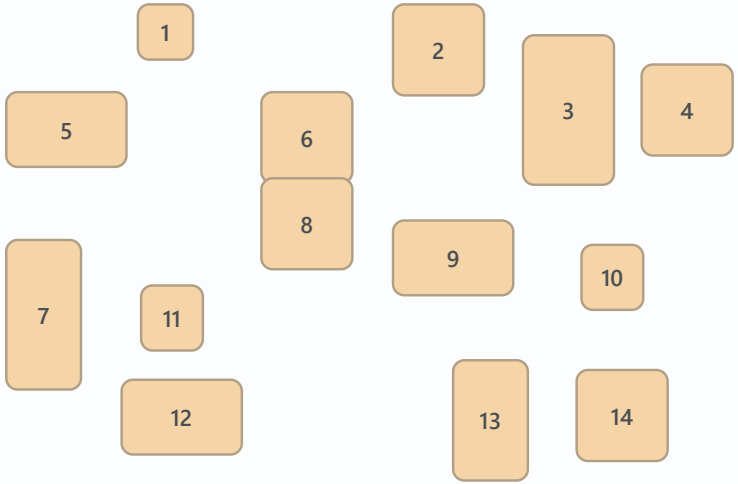
- Birbirini güçlü şekilde çeken mıknatıslar düşünelim: iki mıknatıs değince **ayrılmaz**.
- Birleşmeler sürerken "hangi mıknatıs hangi grupta?" sorusunu izlemek istiyoruz.
- Önce her mıknatısa bir **numara** verilir.
- Birleşme = `union`, "aynı grupta mı?" = iki `find` karşılaştırması.




4 ile 10 aynı grupta mı? find(4) ≠ find(10) → hayır. **union(4, 10)** iki grubu birleştirir.

# Mıknatıs Örneği

- Birbirini güçlü şekilde çeken mıknatıslar düşünelim: iki mıknatıs değince **ayrılmaz**.
- Birleşmeler sürerken "hangi mıknatıs hangi grupta?" sorusunu izlemek istiyoruz.
- Önce her mıknatısa bir **numara** verilir.
- Birleşme = `union`, "aynı grupta mı?" = iki `find` karşılaştırması.




Birleşmeler sürer ve sonunda tüm mıknatıslar **tek grupta** toplanır: find(x) herkes için aynı kökü verir.

# Find ve Union İşlemleri

- **find(x)**: x'i içeren kümenin adını (kökünü) döndürür. Ebeveyn bağlantıları kök bulunana kadar izlenir.
- Bu, arama ağaçlarındaki *arama* ile aynı şey değildir: değer aranmaz, yukarı doğru köke çıkılır.
- **union(x, y)**: iki kümenin yerine, ikisinin birleşimi olan tek bir küme yaratır: bir kökü diğerinin altına bağlar.

Kümeler baştan ayrık olduğu için birleşimde tekrar kontrolü yok: yalnızca **bir ebeveyn bağlantısı** değişir.

```
find(x):
  if x.ebeveyn != x:
    x.ebeveyn = find(x.ebeveyn)  // yol kısaltma
  return x.ebeveyn

union(x, y):
  kök1 = find(x)
  kök2 = find(y)
  if kök1 != kök2:
    // bir kümenin kökünü diğer
    // kümenin köküne ebeveyn
    kök1.ebeveyn = kök2
```

# Nesneleri İndislere Eşleme

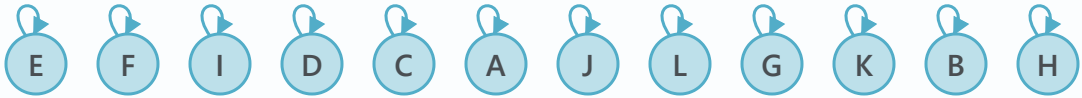
- Union-Find en verimli hâliyle **dizi** üzerinde çalışır: öğeler 0..n−1 tam sayılarıdır.
- Öğeler harf, şehir, kişi… ise önce her nesne bir indise **rastgele (ama kalıcı) eşlenir** ve bu eşleme bir hash tablosunda saklanır.
- Sonra `ebeveyn[i] = i` ile her öğe kendi kümesinin kökü olarak başlar ( `makeSet` ).

| | | | | | |
|---|---|---|---|---|---|
| E → 0 | F → 1 | I → 2 | D → 3 | C → 4 | A → 5 |
| J → 6 | L → 7 | G → 8 | K → 9 | B → 10 | H → 11 |

```
Map<Character, Integer> indis = new HashMap<>();   // 'E'→0, 'F'→1, …
int[] ebeveyn = new int[12];
for (int i = 0; i < 12; i++) ebeveyn[i] = i;             // 12 tekil küme
```

# Union İşlemi Adım Adım

| öğe | E | F | I | D | C | A | J | L | G | K | B | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| indis | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| ebeveyn | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |



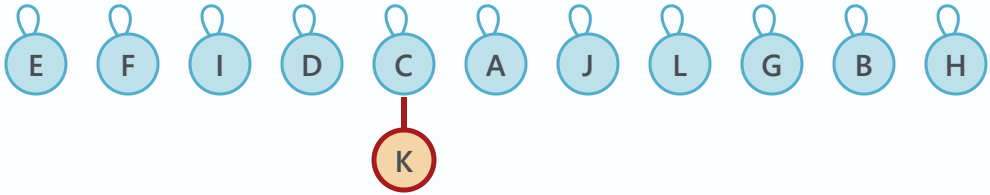

İşlemler

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

Başlangıç: 12 tekil küme, `ebeveyn[i] = i`. Kural: **küçük ağacın kökü büyüğün köküne** bağlanır; eşitse ikinci kök birinciye bağlanır. Yol kısaltma yok.

# Union İşlemi Adım Adım

| öğe | E | F | I | D | C | A | J | L | G | K | B | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| indis | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| ebeveyn | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 4 | 10 | 11 |



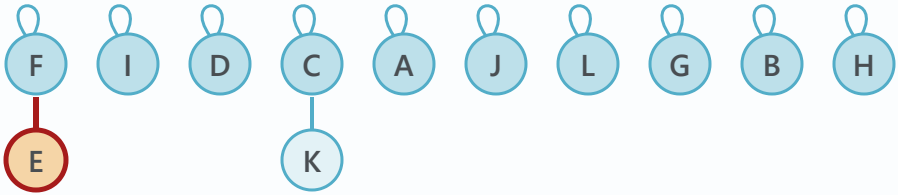

İşlemler

**union(C, K)**
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)

**union(C, K)**: find(C) = C, find(K) = K; boyutlar eşit (1) → **K**, C'nin altına bağlanır (ebeveyn[9] = 4).

# Union İşlemi Adım Adım

| öğe | E | F | I | D | C | A | J | L | G | K | B | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| indis | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| ebeveyn | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 4 | 10 | 11 |



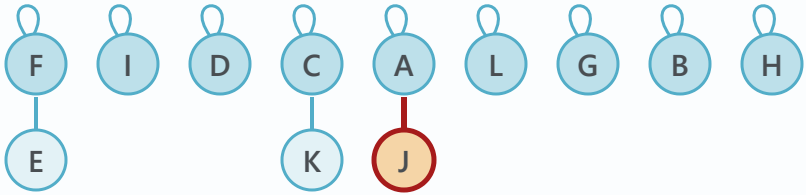

İşlemler

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

**union(F, E)**: find(F) = F, find(E) = E; boyutlar eşit (1) → **E**, F'nin altına bağlanır (ebeveyn[0] = 1).

# Union İşlemi Adım Adım

| öğe | E | F | I | D | C | A | J | L | G | K | B | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| indis | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| ebeveyn | 1 | 1 | 2 | 3 | 4 | 5 | 5 | 7 | 8 | 4 | 10 | 11 |



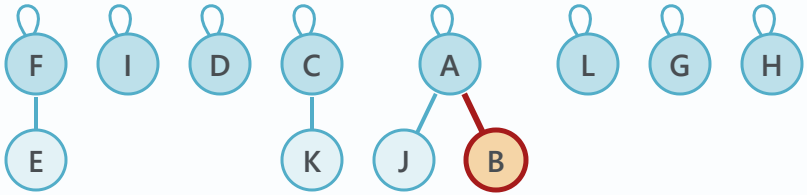

İşlemler

union(C, K)
union(F, E)
**union(A, J)**
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)

**union(A, J)**: find(A) = A, find(J) = J; boyutlar eşit (1) → **J**, A'nın altına bağlanır (ebeveyn[6] = 5).

# Union İşlemi Adım Adım

| öğe | E | F | I | D | C | A | J | L | G | K | B | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| indis | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| ebeveyn | 1 | 1 | 2 | 3 | 4 | 5 | 5 | 7 | 8 | 4 | 5 | 11 |



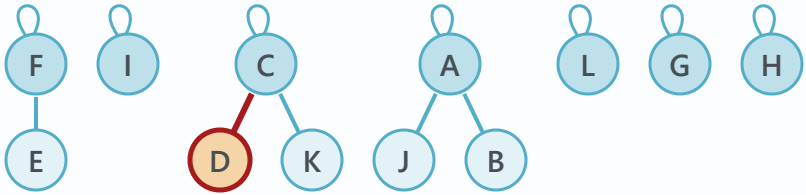

İşlemler

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

**union(A, B)**: find(A) = A, find(B) = B; boyut 1 < 2 → **B**, A'nın altına bağlanır (ebeveyn[10] = 5).

# Union İşlemi Adım Adım

| öğe | E | F | I | D | C | A | J | L | G | K | B | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| indis | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| ebeveyn | 1 | 1 | 2 | 4 | 4 | 5 | 5 | 7 | 8 | 4 | 5 | 11 |



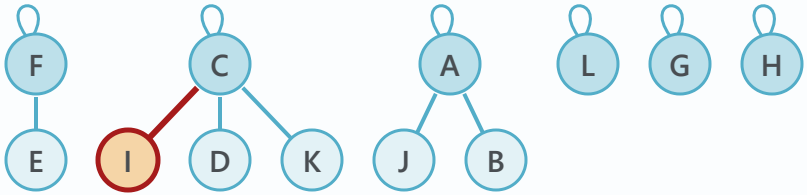

İşlemler

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

**union(C, D)**: find(C) = C, find(D) = D; boyut 1 < 2 → **D**, C'nin altına bağlanır (ebeveyn[3] = 4).

# Union İşlemi Adım Adım

| öğe | E | F | I | D | C | A | J | L | G | K | B | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| indis | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| ebeveyn | 1 | 1 | 4 | 4 | 4 | 5 | 5 | 7 | 8 | 4 | 5 | 11 |



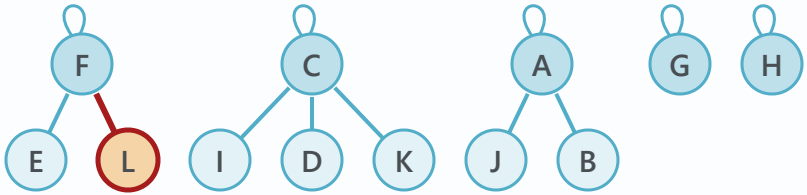

İşlemler

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

**union(D, I)**: find(D) = C, find(I) = I; boyut 1 < 3 → **I**, C'nin altına bağlanır (ebeveyn[2] = 4).

# Union İşlemi Adım Adım

| öğe | E | F | I | D | C | A | J | L | G | K | B | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| indis | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| ebeveyn | 1 | 1 | 4 | 4 | 4 | 5 | 5 | 1 | 8 | 4 | 5 | 11 |



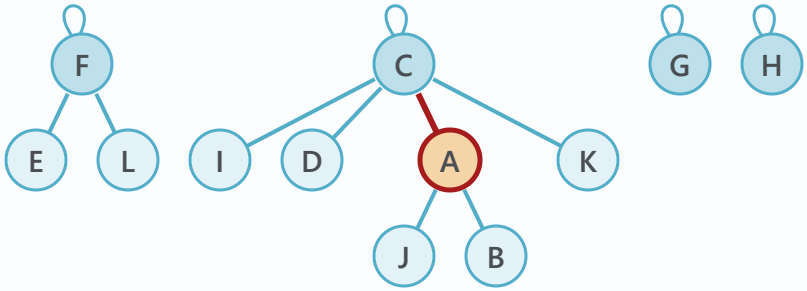

İşlemler

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

**union(L, F)**: find(L) = L, find(F) = F; boyut 1 < 2 → **L**, F'nin altına bağlanır (ebeveyn[7] = 1).

# Union İşlemi Adım Adım

| öğe | E | F | I | D | C | A | J | L | G | K | B | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| indis | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| ebeveyn | 1 | 1 | 4 | 4 | 4 | 4 | 5 | 1 | 8 | 4 | 5 | 11 |



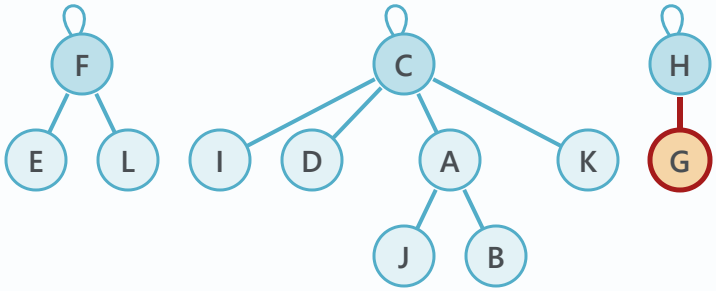

İşlemler

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

**union(C, A)**: find(C) = C, find(A) = A; boyut 3 < 4 → **A**, C'nin altına bağlanır (ebeveyn[5] = 4).

# Union İşlemi Adım Adım

| öğe | E | F | I | D | C | A | J | L | G | K | B | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| indis | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| ebeveyn | 1 | 1 | 4 | 4 | 4 | 4 | 5 | 1 | 11 | 4 | 5 | 11 |



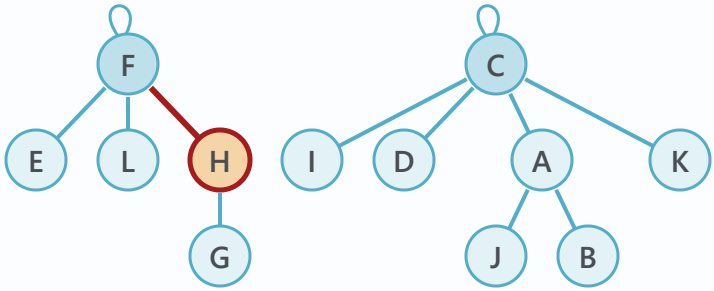

İşlemler

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

**union(A, B)**: find(A) = find(B) = C → zaten aynı kümede, hiçbir şey değişmez. **union(H, G)**: find(H) = H, find(G) = G; boyutlar eşit (1) → **G**, H'nin altına bağlanır (ebeveyn[8] = 11).

# Union İşlemi Adım Adım

| öğe | E | F | I | D | C | A | J | L | G | K | B | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| indis | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| ebeveyn | 1 | 1 | 4 | 4 | 4 | 4 | 5 | 1 | 11 | 4 | 5 | 1 |



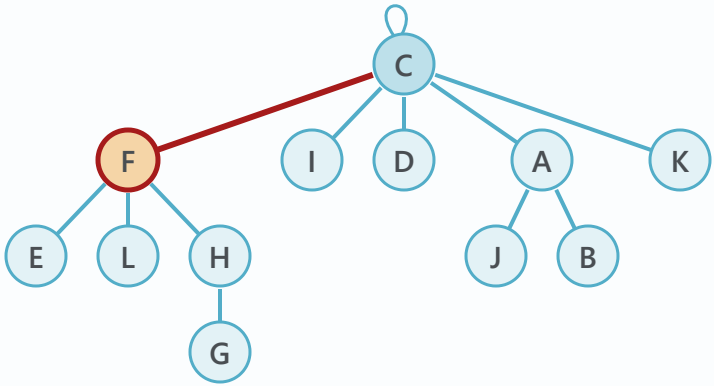

İşlemler

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

**union(H, F)**: find(H) = H, find(F) = F; boyut 2 < 3 → **H**, F'nin altına bağlanır (ebeveyn[11] = 1).

# Union İşlemi Adım Adım

| öğe | E | F | I | D | C | A | J | L | G | K | B | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| indis | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| ebeveyn | 1 | 4 | 4 | 4 | 4 | 4 | 5 | 1 | 11 | 4 | 5 | 1 |



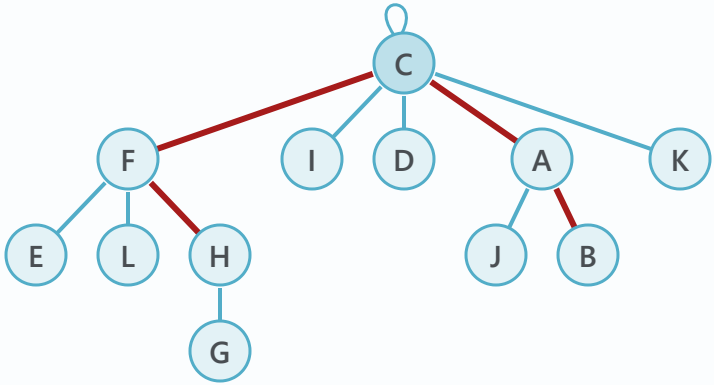

İşlemler

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

**union(H, B)**: find(H) = F, find(B) = C; boyut 5 < 7 → **F**, C'nin altına bağlanır (ebeveyn[1] = 4).

# Union İşlemi Adım Adım

| öğe | E | F | I | D | C | A | J | L | G | K | B | H |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| indis | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| ebeveyn | **1** | **4** | **4** | **4** | **4** | **4** | **5** | **1** | **11** | **4** | **5** | **1** |



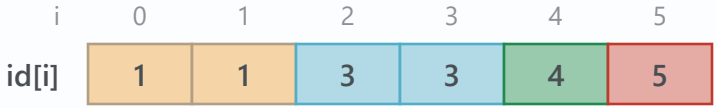

İşlemler

```
union(C, K)
union(F, E)
union(A, J)
union(A, B)
union(C, D)
union(D, I)
union(L, F)
union(C, A)
union(A, B)
union(H, G)
union(H, F)
union(H, B)
```

Sonuç: tek küme, kök **C**. H ile B aynı kümede mi? find(H): H → F → C, find(B): B → A → C; kökler aynı → **evet** (toplam 4 adım).

# Başka Bir Fikir: Quick-Find (Hızlı Bul)

- Ağaç yerine düz dizi: `id[i]` = i'nin **küme adı**; aynı kümedekilerin id'si aynı. **find(p)** = `id[p]` → O(1).
- **union(p, q)**: id'si `id[p]` olan *herkesi* `id[q]` yap → tüm dizi taranır, **O(n)**.

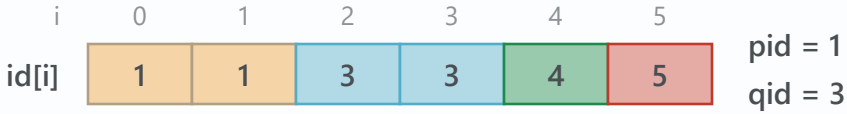



kümeler: {0, 1} {2, 3} {4} {5}

union(0,1), union(2,3) sonrası: 4 küme (aynı renk = aynı küme)

| Yöntem (Sedgewick) | find | union | n − 1 union toplamı |
|---|---|---|---|
| Quick-Find (dizi, küme adı) | **O(1)** | O(n) | $O(n^2)$ |
| Quick-Union (ebeveyn ağacı, naif) | O(h), en kötü O(n) | O(h) | en kötü $O(n^2)$ |
| Rütbe/boyut + yol kısaltma | O(α(n)) amortize | | ≈ O(n) |

# Kod İzleme: Quick-Find `union`




kümeler: {0, 1} {2, 3} {4} {5}

union(1, 3): p'nin küme adı **pid = id[1] = 1**, q'nunki **qid = id[3] = 3**.

Adım 1 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
▶   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
5   for (int i = 0; i < id.length; i++)
6     if (id[i] == pid)
7       id[i] = qid;            // O(n) tarama
8 }
```

**DEĞİŞKENLER**

`p = 1` `q = 3` `pid = 1` `qid = 3`

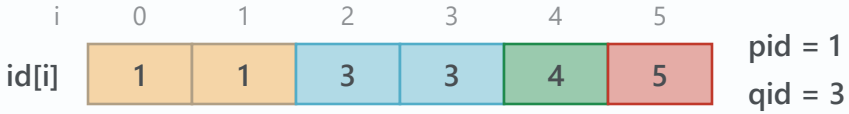
`id = [1, 1, 3, 3, 4, 5]`

**ÇAĞRI YIĞINI**

`union(1, 3)`

`main()`

# Kod İzleme: Quick-Find `union`




kümeler: {0, 1} {2, 3} {4} {5}

pid ≠ qid: farklı kümeler, birleştirme gerekli. Şimdi **tüm dizi** taranacak.

Adım 2 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
▶  if (pid == qid) return;
5   for (int i = 0; i < id.length; i++)
6     if (id[i] == pid)
7       id[i] = qid;            // O(n) tarama
8 }
```

**DEĞİŞKENLER**

`p = 1` `q = 3` `pid = 1` `qid = 3`

`id = [1, 1, 3, 3, 4, 5]`

**ÇAĞRI YIĞINI**

`union(1, 3)`

`main()`

# Kod İzleme: Quick-Find `union`




kümeler: {0, 2, 3} {1} {4} {5}

id[0] = 1 = pid → **id[0] = 3** yapılır.

Adım 3 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
5   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
▶       id[i] = qid;            // O(n) tarama
8 }
```

**DEĞİŞKENLER**

`p = 1` `q = 3` `pid = 1` `qid = 3`

`i = 0` `id = [3, 1, 3, 3, 4, 5]`

**ÇAĞRI YIĞINI**

`union(1, 3)`

`main()`

# Kod İzleme: Quick-Find `union`




kümeler: {0, 1, 2, 3} {4} {5}

id[1] = 1 = pid → **id[1] = 3** yapılır.

Adım 4 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
5   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
▶       id[i] = qid;            // O(n) tarama
8 }
```

**DEĞİŞKENLER**

`p = 1` `q = 3` `pid = 1` `qid = 3`

`i = 1` `id = [3, 3, 3, 3, 4, 5]`

**ÇAĞRI YIĞINI**

`union(1, 3)`

`main()`

# Kod İzleme: Quick-Find `union`




kümeler: {0, 1, 2, 3} {4} {5}

id[2] = 3 ≠ pid: dokunulmaz (ama yine de bakıldı).

Adım 5 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
▶   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
7       id[i] = qid;            // O(n) tarama
8 }
```

**DEĞİŞKENLER**

`p = 1` `q = 3` `pid = 1` `qid = 3`

`i = 2` `id = [3, 3, 3, 3, 4, 5]`

**ÇAĞRI YIĞINI**

`union(1, 3)`

`main()`

# Kod İzleme: Quick-Find `union`




kümeler: {0, 1, 2, 3} {4} {5}

id[3] = 3 ≠ pid: dokunulmaz (ama yine de bakıldı).

Adım 6 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
▶   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
7       id[i] = qid;            // O(n) tarama
8 }
```

**DEĞİŞKENLER**

`p = 1` `q = 3` `pid = 1` `qid = 3`

`i = 3` `id = [3, 3, 3, 3, 4, 5]`

**ÇAĞRI YIĞINI**

`union(1, 3)`

`main()`

# Kod İzleme: Quick-Find `union`




kümeler: {0, 1, 2, 3} {4} {5}

id[4] = 4 ≠ pid: dokunulmaz (ama yine de bakıldı).



```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
▶   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
7       id[i] = qid;            // O(n) tarama
8 }
```

**DEĞİŞKENLER**

`p = 1` `q = 3` `pid = 1` `qid = 3`

`i = 4` `id = [3, 3, 3, 3, 4, 5]`

**ÇAĞRI YIĞINI**

`union(1, 3)`

`main()`

# Kod İzleme: Quick-Find `union`




kümeler: {0, 1, 2, 3} {4} {5}

id[5] = 5 ≠ pid: dokunulmaz (ama yine de bakıldı).

Adım 8 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
▶   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
7       id[i] = qid;            // O(n) tarama
8 }
```

**DEĞİŞKENLER**

`p = 1` `q = 3` `pid = 1` `qid = 3`

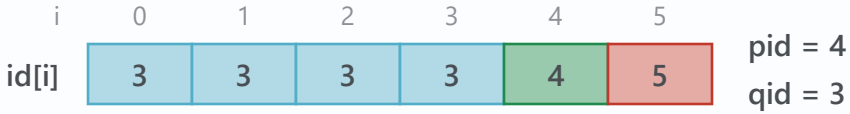
`i = 5` `id = [3, 3, 3, 3, 4, 5]`

**ÇAĞRI YIĞINI**

`union(1, 3)`

`main()`

# Kod İzleme: Quick-Find `union`




kümeler: {0, 1, 2, 3} {4} {5}

union(4, 3): p'nin küme adı **pid = id[4] = 4**, q'nunki **qid = id[3] = 3**.

Adım 9 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
▶   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
5   for (int i = 0; i < id.length; i++)
6     if (id[i] == pid)
7       id[i] = qid;            // O(n) tarama
8 }
```

**DEĞİŞKENLER**

`p = 4` `q = 3` `pid = 4` `qid = 3`

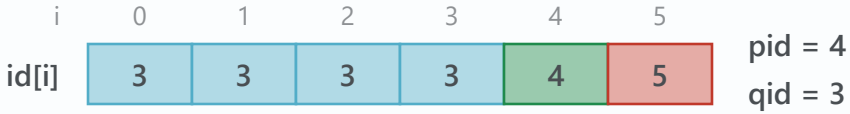
`id = [3, 3, 3, 3, 4, 5]`

**ÇAĞRI YIĞINI**

`union(4, 3)`

`main()`

# Kod İzleme: Quick-Find `union`




kümeler: {0, 1, 2, 3} {4} {5}

pid ≠ qid: farklı kümeler, birleştirme gerekli. Şimdi **tüm dizi** taranacak.

Adım 10 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
▶  if (pid == qid) return;
5   for (int i = 0; i < id.length; i++)
6     if (id[i] == pid)
7       id[i] = qid;            // O(n) tarama
8 }
```

**DEĞİŞKENLER**

`p = 4` `q = 3` `pid = 4` `qid = 3`

`id = [3, 3, 3, 3, 4, 5]`

**ÇAĞRI YIĞINI**

`union(4, 3)`

`main()`

# Kod İzleme: Quick-Find `union`




kümeler: {0, 1, 2, 3} {4} {5}

id[0] = 3 ≠ pid: dokunulmaz (ama yine de bakıldı).

Adım 11 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
▶   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
7       id[i] = qid;            // O(n) tarama
8 }
```

**DEĞİŞKENLER**

`p = 4` `q = 3` `pid = 4` `qid = 3` `i = 0` `id = [3, 3, 3, 3, 4, 5]`

**ÇAĞRI YIĞINI**

- `union(4, 3)`
- `main()`

# Kod İzleme: Quick-Find `union`




kümeler: {0, 1, 2, 3} {4} {5}

id[1] = 3 ≠ pid: dokunulmaz (ama yine de bakıldı).

Adım 12 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
▶   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
7       id[i] = qid;            // O(n) tarama
8 }
```

**DEĞİŞKENLER**

`p = 4` `q = 3` `pid = 4` `qid = 3`

`i = 1` `id = [3, 3, 3, 3, 4, 5]`

**ÇAĞRI YIĞINI**

`union(4, 3)`

`main()`

# Kod İzleme: Quick-Find `union`




kümeler: {0, 1, 2, 3} {4} {5}

id[2] = 3 ≠ pid: dokunulmaz (ama yine de bakıldı).

Adım 13 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
▶   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
7       id[i] = qid;            // O(n) tarama
8 }
```

**DEĞİŞKENLER**

`p = 4` `q = 3` `pid = 4` `qid = 3`

`i = 2` `id = [3, 3, 3, 3, 4, 5]`

**ÇAĞRI YIĞINI**

`union(4, 3)`

`main()`

# Kod İzleme: Quick-Find `union`




kümeler: {0, 1, 2, 3} {4} {5}

id[3] = 3 ≠ pid: dokunulmaz (ama yine de bakıldı).

Adım 14 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
▶   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
7       id[i] = qid;            // O(n) tarama
8 }
```

**DEĞİŞKENLER**

`p = 4` `q = 3` `pid = 4` `qid = 3`

`i = 3` `id = [3, 3, 3, 3, 4, 5]`

**ÇAĞRI YIĞINI**

`union(4, 3)`

`main()`

# Kod İzleme: Quick-Find `union`




kümeler: {0, 1, 2, 3, 4} {5}

id[4] = 4 = pid → **id[4] = 3** yapılır.

Adım 15 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
5   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
▶       id[i] = qid;            // O(n) tarama
8 }
```

**DEĞİŞKENLER**

`p = 4` `q = 3` `pid = 4` `qid = 3` `i = 4` `id = [3, 3, 3, 3, 3, 5]`

**ÇAĞRI YIĞINI**

- `union(4, 3)`
- `main()`

# Kod İzleme: Quick-Find `union`




kümeler: {0, 1, 2, 3, 4} {5}

id[5] = 5 ≠ pid: dokunulmaz. Her union **6 hücrenin hepsine** baktı: n öğede O(n); n − 1 union → O($n^2$).

Adım 16 / 16

```
1 int find(int p) { return id[p]; }  // O(1)
2 void union(int p, int q) {
3   int pid = id[p], qid = id[q];
4   if (pid == qid) return;
▶   for (int i = 0; i < id.length; i++)
▶     if (id[i] == pid)
7       id[i] = qid;            // O(n) tarama
8 }
```

**DEĞİŞKENLER**

`p = 4` `q = 3` `pid = 4` `qid = 3` `i = 5` `id = [3, 3, 3, 3, 3, 5]`

**ÇAĞRI YIĞINI**

`union(4, 3)`

`main()`

# Neden İyileştirme Gerekir?

- find, kökten uzaklık kadar adım atar. Birleştirme sırası kötü seçilirse ağaç bir **zincire** dönüşür → en kötü **O(n)**.

Naif: eski kök yeninin altına




**yükseklik 0 · find(1) = 0 adım**

Boyuta göre: küçük ağaç büyüğün altına




**yükseklik 0 · find(1) = 0 adım**

Başlangıç: 5 tekil küme; her düğüm kendi kökü. Sırayla union(1,2), union(2,3), union(3,4), union(4,5) yapılacak.

Adım 1 / 5

İki basit fikir ağaçları basık tutar: **(1) boyuta / rütbeye göre birleştirme** ve **(2) yol kısaltma**. Birlikte işlem başına neredeyse sabit süre: **O(α(n))**.

# Neden İyileştirme Gerekir?

- find, kökten uzaklık kadar adım atar. Birleştirme sırası kötü seçilirse ağaç bir **zincire** dönüşür → en kötü **O(n)**.

Naif: eski kök yeninin altına




**yükseklik 1 · find(1) = 1 adım**

Boyuta göre: küçük ağaç büyüğün altına




**yükseklik 1 · find(1) = 1 adım**

union(1, 2): iki tekil küme, iki yöntem de 1'i 2'nin altına bağlar.

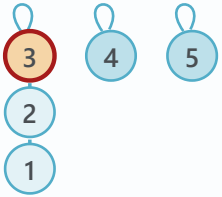


İki basit fikir ağaçları basık tutar: **(1) boyuta / rütbeye göre birleştirme** ve **(2) yol kısaltma**. Birlikte işlem başına neredeyse sabit süre: **O(α(n))**.

# Neden İyileştirme Gerekir?

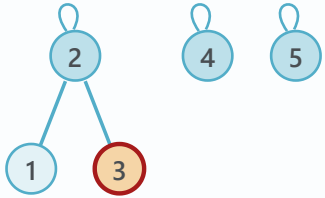
- find, kökten uzaklık kadar adım atar. Birleştirme sırası kötü seçilirse ağaç bir **zincire** dönüşür → en kötü **O(n)**.

Naif: eski kök yeninin altına




**yükseklik 2 · find(1) = 2 adım**

Boyuta göre: küçük ağaç büyüğün altına




**yükseklik 1 · find(1) = 1 adım**

union(2, 3): naif yöntem 2 kökünü (2 öğeli ağaç) tekil 3'ün altına bağlar → zincir uzar. Boyuta göre: tekil 3, büyük ağacın kökü 2'ye bağlanır.

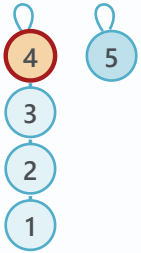


İki basit fikir ağaçları basık tutar: **(1) boyuta / rütbeye göre birleştirme** ve **(2) yol kısaltma**. Birlikte işlem başına neredeyse sabit süre: **O(α(n))**.

# Neden İyileştirme Gerekir?

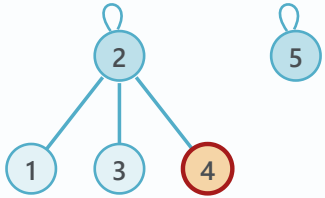
- find, kökten uzaklık kadar adım atar. Birleştirme sırası kötü seçilirse ağaç bir **zincire** dönüşür → en kötü **O(n)**.

Naif: eski kök yeninin altına




**yükseklik 3 · find(1) = 3 adım**

Boyuta göre: küçük ağaç büyüğün altına




**yükseklik 1 · find(1) = 1 adım**

union(3, 4): naif zincir bir kat daha uzar; boyuta göre 4 yine köke eklenir.



İki basit fikir ağaçları basık tutar: **(1) boyuta / rütbeye göre birleştirme** ve **(2) yol kısaltma**. Birlikte işlem başına neredeyse sabit süre: **O(α(n))**.

# Neden İyileştirme Gerekir?

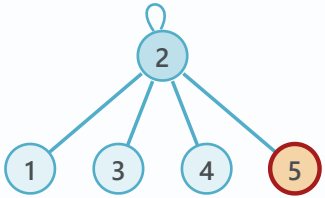
- find, kökten uzaklık kadar adım atar. Birleştirme sırası kötü seçilirse ağaç bir **zincire** dönüşür → en kötü **O(n)**.

Naif: eski kök yeninin altına




**yükseklik 4 · find(1) = 4 adım**

Boyuta göre: küçük ağaç büyüğün altına




**yükseklik 1 · find(1) = 1 adım**

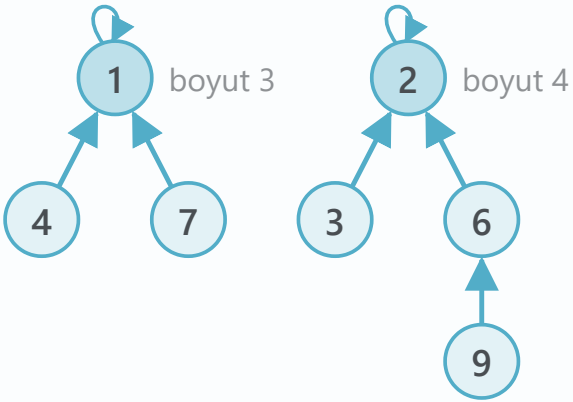
union(4, 5): naif ağaç **zincir** oldu (yükseklik n − 1 → find O(n)); boyuta göre ağaç **yıldız** (yükseklik 1).



İki basit fikir ağaçları basık tutar: **(1) boyuta / rütbeye göre birleştirme** ve **(2) yol kısaltma**. Birlikte işlem başına neredeyse sabit süre: **O(α(n))**.

# Boyuta Göre Birleştirme (Union by Size)

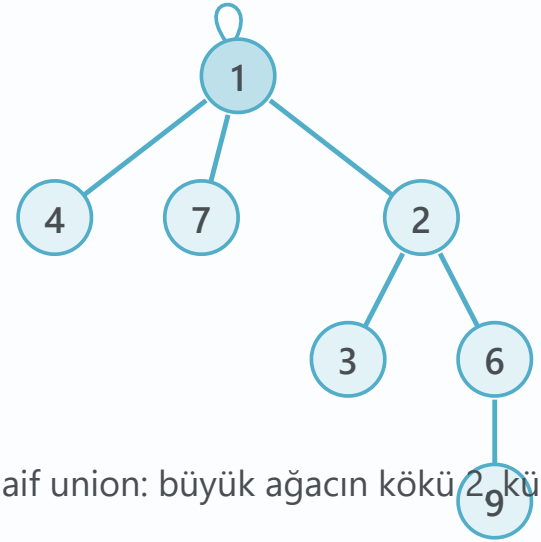
- Her kökte ağacın **boyutunu** (öğe sayısı) sakla.
- Birleştirirken **küçük ağacın kökünü büyük ağacın köküne** bağla; yeni boyut = iki boyutun toplamı.




İki küme: {1, 4, 7} (boyut 3) ve {2, 3, 6, 9} (boyut 4). Yükseklikler 1 ve 2.

# Boyuta Göre Birleştirme (Union by Size)

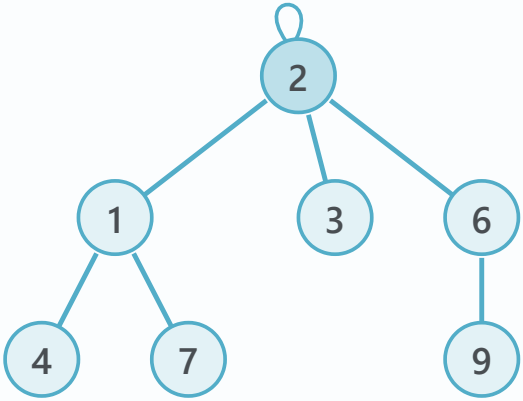
- Her kökte ağacın **boyutunu** (öğe sayısı) sakla.
- Birleştirirken **küçük ağacın kökünü büyük ağacın köküne** bağla; yeni boyut = iki boyutun toplamı.




Naif union: büyük ağacın kökü 2 küçüğün altına bağlanırsa yükseklik **3**'e çıkar; 9'un find'ı uzar.

# Boyuta Göre Birleştirme (Union by Size)

- Her kökte ağacın **boyutunu** (öğe sayısı) sakla.
- Birleştirirken **küçük ağacın kökünü büyük ağacın köküne** bağla; yeni boyut = iki boyutun toplamı.




Boyuta göre: küçük ağacın kökü 1, büyüğün kökü 2'ye bağlanır. Yükseklik **2** kalır, boyut[2] = 3 + 4 = **7**.

# Rütbeye Göre Birleştirme (Union by Rank)

- **Rütbe**, yüksekliğin üst sınırıdır; yeni kümede 0'dır.
- Küçük rütbeli kök, büyük rütbeli köke bağlanır.
- Rütbeler eşitse yeni kökün rütbesi bir artar.
- Bu seçim, yol kısaltma olmadan bile yüksekliği O(log n) tutar.

Yol kısaltma yüksekliği azaltır ama rütbeyi güncellemez; bu yüzden rütbe yalnızca bir *üst sınırdır*.

```
makeSet(x):
  ebeveyn[x] = x
  sira[x] = 0

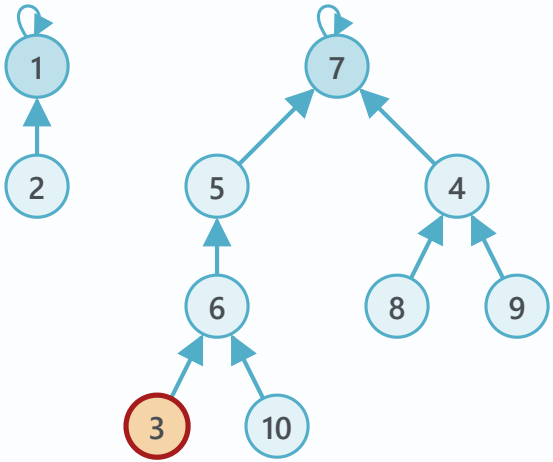
find(x):   // yol kısaltmalı
  if x != ebeveyn[x]:
    ebeveyn[x] = find(ebeveyn[x])
  return ebeveyn[x]

link(x, y):   // x, y kök
  if sira[x] > sira[y]:
    ebeveyn[y] = x
  else:
    ebeveyn[x] = y
    if sira[x] == sira[y]:
      sira[y] = sira[y] + 1

union(x, y):
  link(find(x), find(y))
```

# Yol Kısaltma (Path Compression)

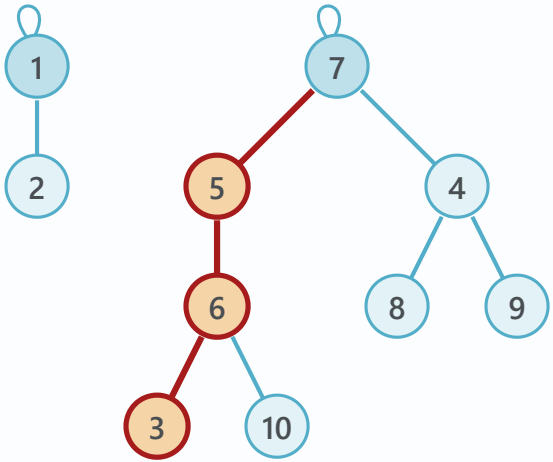
**Yol kısaltma** (path compression): find sırasında rastlanan tüm düğümleri doğrudan **köke bağla**. Sonraki find'lar tek adımda biter.




Başlangıç: 3'ün kökü 7, ama aradaki yol 3 → 6 → 5 → 7 (3 adım).

# Yol Kısaltma (Path Compression)

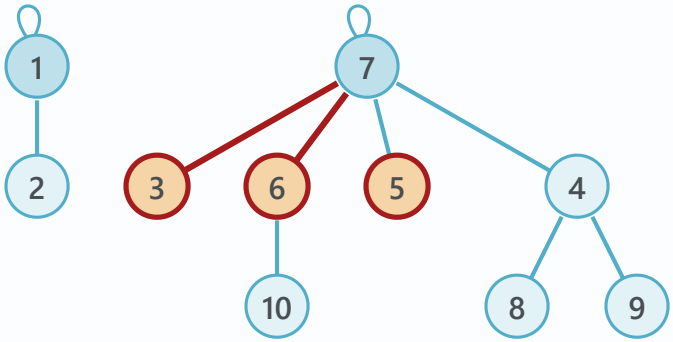
**Yol kısaltma** (path compression): find sırasında rastlanan tüm düğümleri doğrudan **köke bağla**. Sonraki find'lar tek adımda biter.




**find(3)**: özyineleme 3 → 6 → 5 → 7 diye köke çıkar; kök 7 bulunur.

# Yol Kısaltma (Path Compression)

**Yol kısaltma** (path compression): find sırasında rastlanan tüm düğümleri doğrudan **köke bağla**. Sonraki find'lar tek adımda biter.




Dönüşte her düğümün ebeveyni 7 yapılır: `ebeveyn[3] = ebeveyn[6] = 7` (5 zaten 7'deydi). Artık find(3), find(6) tek adım; 10'un yolu da kısaldı.

# Union-Find Karmaşıklığı

| Uygulama | find | union | Not |
|---|---|---|---|
| Naif (rastgele bağlama) | O(n) | O(n) | ağaç zincire dönüşebilir |
| Boyuta veya rütbeye göre | O(log n) | O(log n) | yükseklik ≤ $\log_2 n$ |
| Yalnızca yol kısaltma | O(log n) amortize | | m işlem için |
| **Rütbe + yol kısaltma** | **O(α(n)) amortize** | | pratikte sabit |

**α(n): ters Ackermann fonksiyonu**

Son derece yavaş büyür: evrendeki atom sayısı kadar öğe için bile α(n) ≤ 4.

**Bellek**

İki tam sayı dizisi: `ebeveyn[n]` ve `sira[n]` (veya `boyut[n]`) → O(n).

Tarjan (1975): m işlem dizisi O(m · α(n)) sürede biter. Kruskal, bağlantılı bileşenler, resim işleme (bölge etiketleme), ağ bağlantı sorguları gibi yerlerde kullanılır.

# Java: Düğüm (Nesne) Tabanlı Union-Find

```java
class SetDugum {
  int veri;
  int sira;   // rütbe
  SetDugum ebeveyn;

  SetDugum(int veri) {
    this.veri = veri;
    this.ebeveyn = this;
    this.sira = 0;
  }
}

SetDugum find(SetDugum d) {
  if (d != d.ebeveyn)
    d.ebeveyn = find(d.ebeveyn); // kısaltma
  return d.ebeveyn;
}
```

```java
void union(SetDugum x, SetDugum y) {
  SetDugum kokX = find(x);
  SetDugum kokY = find(y);
  if (kokX == kokY) return;
  // rütbeye göre birleştirme
  if (kokX.sira < kokY.sira) {
    kokX.ebeveyn = kokY;
  } else if (kokX.sira > kokY.sira) {
    kokY.ebeveyn = kokX;
  } else { // eşit: birini seç
    kokX.ebeveyn = kokY;
    kokY.sira++;
  }
}
```

# Java: Dizi Tabanlı Union-Find

```java
class UnionFind {
  private final int[] ebeveyn, sira;
  UnionFind(int n) {
    ebeveyn = new int[n];  sira = new int[n];
    for (int i = 0; i < n; i++) ebeveyn[i] = i;
  }
  int find(int x) {
    if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
    return ebeveyn[x];
  }
  boolean union(int x, int y) {
    int kx = find(x), ky = find(y);
    if (kx == ky) return false;
    if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
    else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
    else { ebeveyn[kx] = ky; sira[ky]++; }
    return true;
  }
}
```

### Kullanım

```
UnionFind uf = new UnionFind(5);
uf.union(0, 1); uf.union(3, 4);
uf.find(1) == uf.find(0) → true
uf.find(1) == uf.find(3) → false
```

### `union` neden boolean?

Birleşme olduysa true: Kruskal'da "kenar döngü oluşturuyor mu?" sorusunun cevabı.

### Derin özyineleme

Çok büyük n'de find döngüyle de yazılabilir.

# Kod İzleme: `find` + `union`

```
▶ int find(int x) {
2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
3   return ebeveyn[x];
4 }
5 boolean union(int x, int y) {
6   int kx = find(x), ky = find(y);
7   if (kx == ky) return false;
8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10  else { ebeveyn[kx] = ky; sira[ky]++; }
11  return true;
12 }
```

**ÇAĞRI YIĞINI**

`main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 0 | 1 | 2 | 3 | 4 | 5 |
| sira | 0 | 0 | 0 | 0 | 0 | 0 |

6 öğe, `ebeveyn[i] = i`, `sira[i] = 0`: 6 tekil küme (her kök kendini gösterir).

Adım 1 / 14

**DEĞİŞKENLER**

—

# Kod İzleme: `find` + `union`

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

ÇAĞRI YIĞINI

- `find(0)`
- `union(0, 1)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 0 | 1 | 2 | 3 | 4 | 5 |
| sira | 0 | 0 | 0 | 0 | 0 | 0 |

find(0): ebeveyn[0] == 0 → **0 köktür**, döndürülür.

Adım 2 / 14

DEĞİŞKENLER

`x = 0`

# Kod İzleme: `find` + `union`

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

ÇAĞRI YIĞINI

- `find(1)`
- `union(0, 1)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 0 | 1 | 2 | 3 | 4 | 5 |
| sira | 0 | 0 | 0 | 0 | 0 | 0 |

find(1): ebeveyn[1] == 1 → **1 köktür**, döndürülür.

Adım 3 / 14

DEĞİŞKENLER

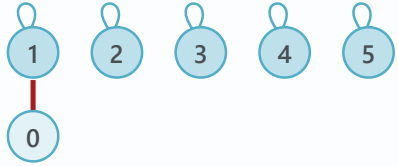
`x = 1`

# Kod İzleme: `find` + `union`

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
   ▶ else { ebeveyn[kx] = ky; sira[ky]++; }
   ▶ return true;
12 }
```

**ÇAĞRI YIĞINI**

- **`union(0, 1)`**
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 1 | 1 | 2 | 3 | 4 | 5 |
| sira | 0 | 1 | 0 | 0 | 0 | 0 |

Rütbeler eşit (0) → ebeveyn[0] = 1 ve yeni kökün rütbesi artar: **sira[1] = 1**.

Adım 4 / 14

**DEĞİŞKENLER**

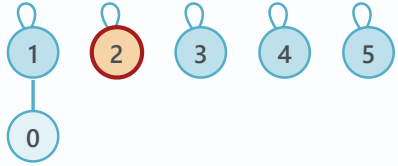
`x = 0` `y = 1` `kx = 0` `ky = 1` `dönüş = true`

# Kod İzleme: `find` + `union`

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

**ÇAĞRI YIĞINI**

- `find(2)`
- `union(2, 3)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 1 | 1 | 2 | 3 | 4 | 5 |
| sira | 0 | 1 | 0 | 0 | 0 | 0 |

find(2): ebeveyn[2] == 2 → **2 köktür**, döndürülür.

Adım 5 / 14

**DEĞİŞKENLER**

x = 2

# Kod İzleme: `find` + `union`

```
1  int find(int x) {
2    if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
▶    return ebeveyn[x];
4  }
5  boolean union(int x, int y) {
6    int kx = find(x), ky = find(y);
7    if (kx == ky) return false;
8    if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
9    else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

**ÇAĞRI YIĞINI**

- `find(3)`
- `union(2, 3)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 1 | 1 | 2 | 3 | 4 | 5 |
| sira | 0 | 1 | 0 | 0 | 0 | 0 |

find(3): ebeveyn[3] == 3 → **3 köktür**, döndürülür.

Adım 6 / 14

**DEĞİŞKENLER**

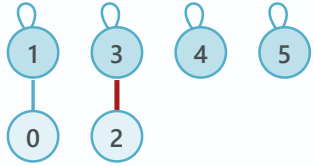
`x = 3`

# Kod İzleme: `find` + `union`

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
   ▶ else { ebeveyn[kx] = ky; sira[ky]++; }
   ▶ return true;
12 }
```

**ÇAĞRI YIĞINI**

- **`union(2, 3)`**
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 1 | 1 | 3 | 3 | 4 | 5 |
| sira | 0 | 1 | 0 | 1 | 0 | 0 |

Rütbeler eşit (0) → ebeveyn[2] = 3 ve yeni kökün rütbesi artar: **sira[3] = 1**.

Adım 7 / 14

**DEĞİŞKENLER**

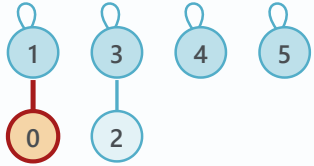
`x = 2` `y = 3` `kx = 2` `ky = 3` `dönüş = true`

# Kod İzleme: `find` + `union`

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

ÇAĞRI YIĞINI

- **`find(0)`**
- `union(0, 2)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 1 | 1 | 3 | 3 | 4 | 5 |
| sira | 0 | 1 | 0 | 1 | 0 | 0 |

find(0): ebeveyn[0] = 1 ≠ 0 → kök değil; `find(1)` ile yukarı çıkılır.

Adım 8 / 14

DEĞİŞKENLER

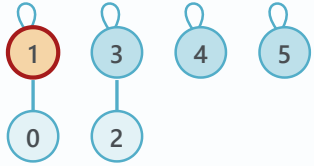
`x = 0`

# Kod İzleme: `find` + `union`

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

ÇAĞRI YIĞINI

- **`find(1)`**
- `find(0)`
- `union(0, 2)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 1 | 1 | 3 | 3 | 4 | 5 |
| sira | 0 | 1 | 0 | 1 | 0 | 0 |

find(1): ebeveyn[1] == 1 → **1 köktür**, döndürülür.

Adım 9 / 14

DEĞİŞKENLER

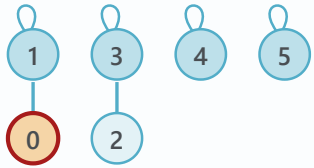
`x = 1`

# Kod İzleme: `find` + `union`

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

ÇAĞRI YIĞINI

- **`find(0)`**
- `union(0, 2)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 1 | 1 | 3 | 3 | 4 | 5 |
| sira | 0 | 1 | 0 | 1 | 0 | 0 |

Dönüşte ebeveyn[0] = 1 (zaten kökün çocuğu, değişmez); find(0) = 1 döner.

Adım 10 / 14

DEĞİŞKENLER

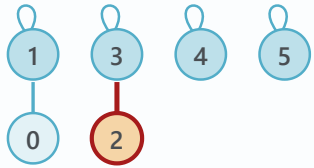
`x = 0`

# Kod İzleme: `find` + `union`

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

ÇAĞRI YIĞINI

- **`find(2)`**
- `union(0, 2)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 1 | 1 | 3 | 3 | 4 | 5 |
| sira | 0 | 1 | 0 | 1 | 0 | 0 |

find(2): ebeveyn[2] = 3 ≠ 2 → kök değil; `find(3)` ile yukarı çıkılır.

Adım 11 / 14

DEĞİŞKENLER

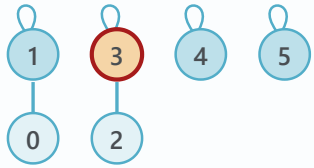
x = 2

# Kod İzleme: `find` + `union`

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

ÇAĞRI YIĞINI

- **`find(3)`**
- `find(2)`
- `union(0, 2)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 1 | 1 | 3 | 3 | 4 | 5 |
| sira | 0 | 1 | 0 | 1 | 0 | 0 |

find(3): ebeveyn[3] == 3 → **3 köktür**, döndürülür.

Adım 12 / 14

DEĞİŞKENLER

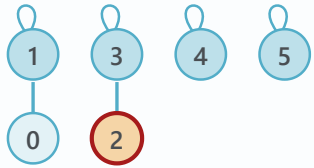
`x = 3`

# Kod İzleme: `find` + `union`

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

ÇAĞRI YIĞINI

- **`find(2)`**
- `union(0, 2)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 1 | 1 | 3 | 3 | 4 | 5 |
| sira | 0 | 1 | 0 | 1 | 0 | 0 |

Dönüşte ebeveyn[2] = 3 (zaten kökün çocuğu, değişmez); find(2) = 3 döner.

Adım 13 / 14

DEĞİŞKENLER

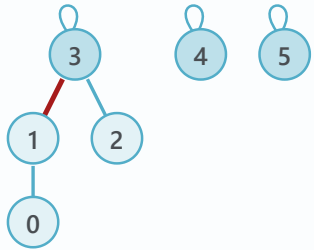
x = 2

# Kod İzleme: `find` + `union`

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
▶    else { ebeveyn[kx] = ky; sira[ky]++; }
▶    return true;
12 }
```

ÇAĞRI YIĞINI

- **union(0, 2)**
- main()




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 1 | 3 | 3 | 3 | 4 | 5 |
| sira | 0 | 1 | 0 | 2 | 0 | 0 |

Rütbeler eşit (1) → ebeveyn[1] = 3 ve yeni kökün rütbesi artar: **sira[3] = 2**.

Adım 14 / 14

DEĞİŞKENLER

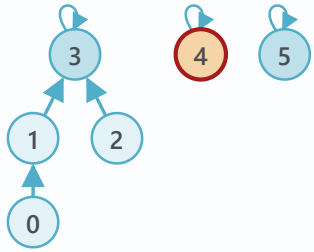
`x = 0` `y = 2` `kx = 1` `ky = 3` `dönüş = true`

# Kod İzleme: `find` + `union` (devam)

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

**ÇAĞRI YIĞINI**

- `find(4)`
- `union(4, 0)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 1 | 3 | 3 | 3 | 4 | 5 |
| sira | 0 | 1 | 0 | 2 | 0 | 0 |

find(4): ebeveyn[4] == 4 → **4 köktür**, döndürülür.

Adım 1 / 15

**DEĞİŞKENLER**

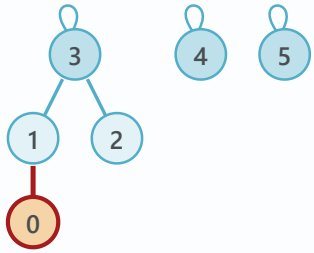
`x = 4`

# Kod İzleme: `find` + `union` (devam)

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

**ÇAĞRI YIĞINI**

- **`find(0)`**
- `union(4, 0)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 1 | 3 | 3 | 3 | 4 | 5 |
| sira | 0 | 1 | 0 | 2 | 0 | 0 |

find(0): ebeveyn[0] = 1 ≠ 0 → kök değil; `find(1)` ile yukarı çıkılır.

Adım 2 / 15

**DEĞİŞKENLER**

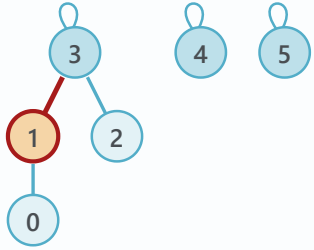
`x = 0`

# Kod İzleme: `find` + `union` (devam)

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

**ÇAĞRI YIĞINI**

- **`find(1)`**
- `find(0)`
- `union(4, 0)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 1 | 3 | 3 | 3 | 4 | 5 |
| sira | 0 | 1 | 0 | 2 | 0 | 0 |

find(1): ebeveyn[1] = 3 ≠ 1 → kök değil; `find(3)` ile yukarı çıkılır.

Adım 3 / 15

**DEĞİŞKENLER**

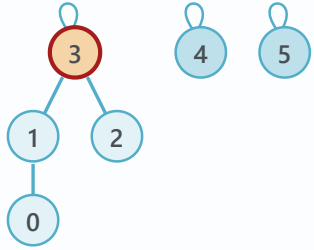
`x = 1`

# Kod İzleme: `find` + `union` (devam)

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

ÇAĞRI YIĞINI

- **`find(3)`**
- `find(1)`
- `find(0)`
- `union(4, 0)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 1 | 3 | 3 | 3 | 4 | 5 |
| sira | 0 | 1 | 0 | 2 | 0 | 0 |

find(3): ebeveyn[3] == 3 → **3 köktür**, döndürülür.



DEĞİŞKENLER

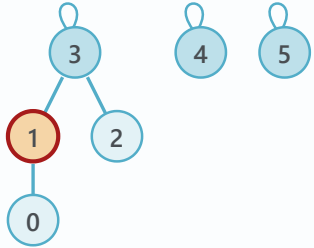
`x = 3`

# Kod İzleme: `find` + `union` (devam)

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

**ÇAĞRI YIĞINI**

- **`find(1)`**
- `find(0)`
- `union(4, 0)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 1 | 3 | 3 | 3 | 4 | 5 |
| sira | 0 | 1 | 0 | 2 | 0 | 0 |

Dönüşte ebeveyn[1] = 3 (zaten kökün çocuğu, değişmez); find(1) = 3 döner.



**DEĞİŞKENLER**

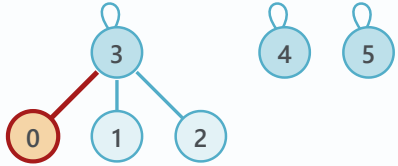
`x = 1`

# Kod İzleme: `find` + `union` (devam)

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

ÇAĞRI YIĞINI

- **`find(0)`**
- `union(4, 0)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 3 | 3 | 3 | 3 | 4 | 5 |
| sira | 0 | 1 | 0 | 2 | 0 | 0 |

**Yol kısaltma**: dönüşte ebeveyn[0] = 1 → **3**; 0 artık doğrudan köke bağlı. find(0) = 3.



DEĞİŞKENLER

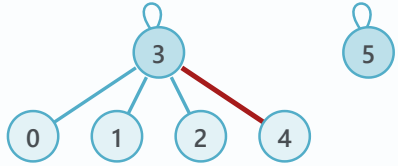
`x = 0`

# Kod İzleme: `find` + `union` (devam)

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 ▶   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
 ▶   return true;
12 }
```

**ÇAĞRI YIĞINI**

- **`union(4, 0)`**
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 3 | 3 | 3 | 3 | 3 | 5 |
| sira | 0 | 1 | 0 | 2 | 0 | 0 |

sira[4] = 0 < sira[3] = 2 → alçak ağacın kökü 4, 3'e bağlanır: ebeveyn[4] = 3. Yükseklik artmaz.



**DEĞİŞKENLER**

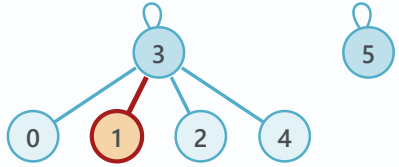
`x = 4` `y = 0` `kx = 4` `ky = 3` `dönüş = true`

# Kod İzleme: `find` + `union` (devam)

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

**ÇAĞRI YIĞINI**

- **`find(1)`**
- `union(1, 4)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 3 | 3 | 3 | 3 | 3 | 5 |
| sira | 0 | 1 | 0 | 2 | 0 | 0 |

find(1): ebeveyn[1] = 3 ≠ 1 → kök değil; `find(3)` ile yukarı çıkılır.



**DEĞİŞKENLER**

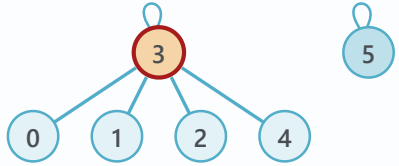
`x = 1`

# Kod İzleme: `find` + `union` (devam)

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

**ÇAĞRI YIĞINI**

- **`find(3)`**
- `find(1)`
- `union(1, 4)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 3 | 3 | 3 | 3 | 3 | 5 |
| sira | 0 | 1 | 0 | 2 | 0 | 0 |

find(3): ebeveyn[3] == 3 → **3 köktür**, döndürülür.

Adım 9 / 15

**DEĞİŞKENLER**

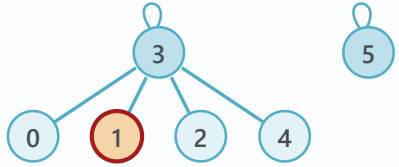
`x = 3`

# Kod İzleme: `find` + `union` (devam)

```
 1 int find(int x) {
 ▶   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 ▶   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

ÇAĞRI YIĞINI

- **`find(1)`**
- `union(1, 4)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 3 | 3 | 3 | 3 | 3 | 5 |
| sira | 0 | 1 | 0 | 2 | 0 | 0 |

Dönüşte ebeveyn[1] = 3 (zaten kökün çocuğu, değişmez); find(1) = 3 döner.

Adım 10 / 15

DEĞİŞKENLER

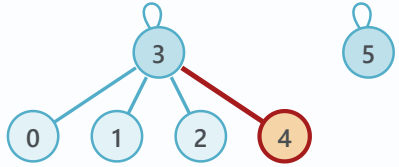
x = 1

# Kod İzleme: `find` + `union` (devam)

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

**ÇAĞRI YIĞINI**

- **`find(4)`**
- `union(1, 4)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 3 | 3 | 3 | 3 | 3 | 5 |
| sira | 0 | 1 | 0 | 2 | 0 | 0 |

find(4): ebeveyn[4] = 3 ≠ 4 → kök değil; `find(3)` ile yukarı çıkılır.

Adım 11 / 15

**DEĞİŞKENLER**

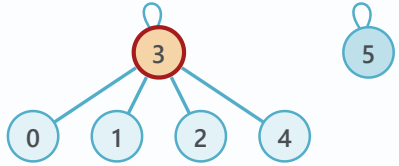
`x = 4`

# Kod İzleme: `find` + `union` (devam)

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

**ÇAĞRI YIĞINI**

- `find(3)`
- `find(4)`
- `union(1, 4)`
- `main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 3 | 3 | 3 | 3 | 3 | 5 |
| sira | 0 | 1 | 0 | 2 | 0 | 0 |

find(3): ebeveyn[3] == 3 → **3 köktür**, döndürülür.

Adım 12 / 15

**DEĞİŞKENLER**

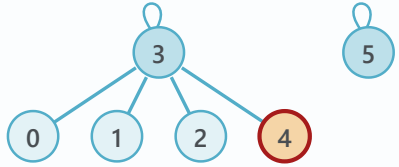
`x = 3`

# Kod İzleme: `find` + `union` (devam)

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

ÇAĞRI YIĞINI

- **find(4)**
- union(1, 4)
- main()




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 3 | 3 | 3 | 3 | 3 | 5 |
| sira | 0 | 1 | 0 | 2 | 0 | 0 |

Dönüşte ebeveyn[4] = 3 (zaten kökün çocuğu, değişmez); find(4) = 3 döner.

Adım 13 / 15

DEĞİŞKENLER

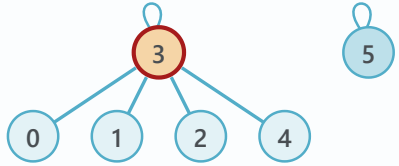
x = 4

# Kod İzleme: `find` + `union` (devam)

```
 1 int find(int x) {
 2   if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
 3   return ebeveyn[x];
 4 }
 5 boolean union(int x, int y) {
 6   int kx = find(x), ky = find(y);
 7   if (kx == ky) return false;
 8   if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
 9   else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

ÇAĞRI YIĞINI

- **union(1, 4)**
- main()




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 3 | 3 | 3 | 3 | 3 | 5 |
| sira | 0 | 1 | 0 | 2 | 0 | 0 |

kx = ky = 3 → 1 ve 4 **zaten aynı kümede**: `return false` (Kruskal: döngü).



DEĞİŞKENLER

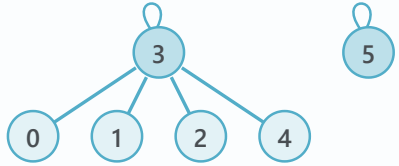
`x = 1` `y = 4` `kx = 3` `ky = 3` `dönüş = false`

# Kod İzleme: `find` + `union` (devam)

```
1  int find(int x) {
2    if (ebeveyn[x] != x) ebeveyn[x] = find(ebeveyn[x]);
▶    return ebeveyn[x];
4  }
5  boolean union(int x, int y) {
6    int kx = find(x), ky = find(y);
7    if (kx == ky) return false;
8    if (sira[kx] < sira[ky]) ebeveyn[kx] = ky;
9    else if (sira[kx] > sira[ky]) ebeveyn[ky] = kx;
10   else { ebeveyn[kx] = ky; sira[ky]++; }
11   return true;
12 }
```

ÇAĞRI YIĞINI

`main()`




| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| ebeveyn | 3 | 3 | 3 | 3 | 3 | 5 |
| sira | 0 | 1 | 0 | 2 | 0 | 0 |

Tek küme, kök 3. Yol kısaltma 0'ı köke taşıdı; sira[3] = 2 yalnızca üst sınır. Birlikte: işlem başına ≈ **O(α(n))**.

Adım 15 / 15

DEĞİŞKENLER

—

# Kod İzleme: döngüyle `find` + yol kısaltma




| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| ebeveyn | 0 | 0 | 1 | 2 | 3 |

Zincir 4 → 3 → 2 → 1 → 0 (kötü birleştirmeler sonucu). find(4) önce kökü arar: **kok = 4**.

Adım 1 / 16

```
1  int find(int x) {
▶    int kok = x;
3    while (ebeveyn[kok] != kok) kok = ebeveyn[kok];
4    while (ebeveyn[x] != kok) {    // yol kısaltma
5      int sonraki = ebeveyn[x];
6      ebeveyn[x] = kok;
7      x = sonraki;
8    }
9    return kok;
10 }
```

**DEĞİŞKENLER**

`x = 4` `kok = 4`

`ebeveyn = [0, 0, 1, 2, 3]`

**ÇAĞRI YIĞINI**

`find(4)`

`main()`

# Kod İzleme: döngüyle `find` + yol kısaltma




| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| ebeveyn | 0 | 0 | 1 | 2 | 3 |

1. tur: ebeveyn[4] = 3 ≠ 4 → yukarı çık, **kok = 3**.

Adım 2 / 16

```
1  int find(int x) {
2    int kok = x;
▶    while (ebeveyn[kok] != kok) kok = ebeveyn[kok];
4    while (ebeveyn[x] != kok) {    // yol kısaltma
5      int sonraki = ebeveyn[x];
6      ebeveyn[x] = kok;
7      x = sonraki;
8    }
9    return kok;
10 }
```

**DEĞİŞKENLER**

`x = 4` `kok = 3`

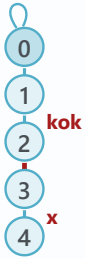
`ebeveyn = [0, 0, 1, 2, 3]`

**ÇAĞRI YIĞINI**

`find(4)`

`main()`

# Kod İzleme: döngüyle `find` + yol kısaltma




| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| ebeveyn | 0 | 0 | 1 | 2 | 3 |

1. tur: ebeveyn[3] = 2 ≠ 3 → yukarı çık, **kok = 2**.

Adım 3 / 16

```
1  int find(int x) {
2    int kok = x;
▶   while (ebeveyn[kok] != kok) kok = ebeveyn[kok];
4    while (ebeveyn[x] != kok) {    // yol kısaltma
5      int sonraki = ebeveyn[x];
6      ebeveyn[x] = kok;
7      x = sonraki;
8    }
9    return kok;
10 }
```

**DEĞİŞKENLER**

`x = 4` `kok = 2`

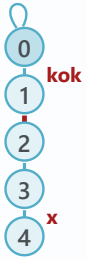
`ebeveyn = [0, 0, 1, 2, 3]`

**ÇAĞRI YIĞINI**

`find(4)`

`main()`

# Kod İzleme: döngüyle `find` + yol kısaltma




| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| ebeveyn | 0 | 0 | 1 | 2 | 3 |

1. tur: ebeveyn[2] = 1 ≠ 2 → yukarı çık, **kok = 1**.

Adım 4 / 16

```
1  int find(int x) {
2    int kok = x;
▶   while (ebeveyn[kok] != kok) kok = ebeveyn[kok];
4    while (ebeveyn[x] != kok) {    // yol kısaltma
5      int sonraki = ebeveyn[x];
6      ebeveyn[x] = kok;
7      x = sonraki;
8    }
9    return kok;
10 }
```

**DEĞİŞKENLER**

`x = 4` `kok = 1`

`ebeveyn = [0, 0, 1, 2, 3]`

**ÇAĞRI YIĞINI**

`find(4)`

`main()`

# Kod İzleme: döngüyle `find` + yol kısaltma




| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| ebeveyn | 0 | 0 | 1 | 2 | 3 |

ebeveyn[1] = 0 → **kok = 0**; ebeveyn[0] = 0 olduğundan döngü biter: kök bulundu (4 adım).

Adım 5 / 16

```
1  int find(int x) {
2    int kok = x;
▶   while (ebeveyn[kok] != kok) kok = ebeveyn[kok];
4    while (ebeveyn[x] != kok) {    // yol kısaltma
5      int sonraki = ebeveyn[x];
6      ebeveyn[x] = kok;
7      x = sonraki;
8    }
9    return kok;
10 }
```

**DEĞİŞKENLER**

`x = 4` `kok = 0`

`ebeveyn = [0, 0, 1, 2, 3]`

**ÇAĞRI YIĞINI**

`find(4)`

`main()`

# Kod İzleme: döngüyle `find` + yol kısaltma




| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| ebeveyn | 0 | 0 | 1 | 2 | 3 |

2. tur: ebeveyn[4] = 3, kök değil → bu düğüm kısaltılacak.



```
1  int find(int x) {
2    int kok = x;
3    while (ebeveyn[kok] != kok) kok = ebeveyn[kok];
▶    while (ebeveyn[x] != kok) {    // yol kısaltma
5      int sonraki = ebeveyn[x];
6      ebeveyn[x] = kok;
7      x = sonraki;
8    }
9    return kok;
10 }
```

**DEĞİŞKENLER**

`x = 4` `kok = 0`

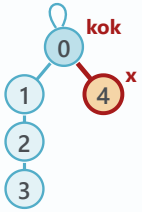
`ebeveyn = [0, 0, 1, 2, 3]`

**ÇAĞRI YIĞINI**

`find(4)`

`main()`

# Kod İzleme: döngüyle `find` + yol kısaltma




| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| ebeveyn | 0 | 0 | 1 | 2 | 0 |

Önce eski ebeveyn saklanır (**sonraki = 3**), sonra **ebeveyn[4] = 0**: 4 artık doğrudan köke bağlı.

Adım 7 / 16

```
1  int find(int x) {
2    int kok = x;
3    while (ebeveyn[kok] != kok) kok = ebeveyn[kok];
4    while (ebeveyn[x] != kok) {    // yol kısaltma
▶      int sonraki = ebeveyn[x];
▶      ebeveyn[x] = kok;
7      x = sonraki;
8    }
9    return kok;
10 }
```

**DEĞİŞKENLER**

`x = 4` `kok = 0` `sonraki = 3`

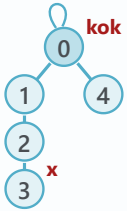
`ebeveyn = [0, 0, 1, 2, 0]`

**ÇAĞRI YIĞINI**

`find(4)`

`main()`

# Kod İzleme: döngüyle `find` + yol kısaltma




| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| ebeveyn | 0 | 0 | 1 | 2 | 0 |

Yolda bir yukarı: **x = 3**.

Adım 8 / 16

```
1  int find(int x) {
2    int kok = x;
3    while (ebeveyn[kok] != kok) kok = ebeveyn[kok];
4    while (ebeveyn[x] != kok) {    // yol kısaltma
5      int sonraki = ebeveyn[x];
6      ebeveyn[x] = kok;
▶      x = sonraki;
8    }
9    return kok;
10 }
```

**DEĞİŞKENLER**

`x = 3` `kok = 0` `sonraki = 3`

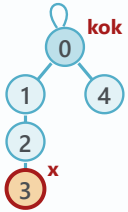
`ebeveyn = [0, 0, 1, 2, 0]`

**ÇAĞRI YIĞINI**

`find(4)`

`main()`

# Kod İzleme: döngüyle `find` + yol kısaltma




| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| ebeveyn | 0 | 0 | 1 | 2 | 0 |

2. tur: ebeveyn[3] = 2, kök değil → bu düğüm kısaltılacak.

Adım 9 / 16

```
1  int find(int x) {
2    int kok = x;
3    while (ebeveyn[kok] != kok) kok = ebeveyn[kok];
▶    while (ebeveyn[x] != kok) {    // yol kısaltma
5      int sonraki = ebeveyn[x];
6      ebeveyn[x] = kok;
7      x = sonraki;
8    }
9    return kok;
10 }
```

**DEĞİŞKENLER**

`x = 3` `kok = 0`

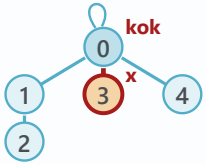
`ebeveyn = [0, 0, 1, 2, 0]`

**ÇAĞRI YIĞINI**

`find(4)`

`main()`

# Kod İzleme: döngüyle `find` + yol kısaltma




| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| ebeveyn | 0 | 0 | 1 | 0 | 0 |

Önce eski ebeveyn saklanır (**sonraki = 2**), sonra **ebeveyn[3] = 0**: 3 artık doğrudan köke bağlı.

Adım 10 / 16

```
 1 int find(int x) {
 2   int kok = x;
 3   while (ebeveyn[kok] != kok) kok = ebeveyn[kok];
 4   while (ebeveyn[x] != kok) {    // yol kısaltma
 ▶     int sonraki = ebeveyn[x];
 ▶     ebeveyn[x] = kok;
 7     x = sonraki;
 8   }
 9   return kok;
10 }
```

**DEĞİŞKENLER**

`x = 3` `kok = 0` `sonraki = 2`

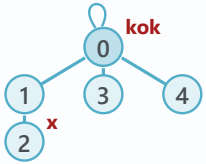
`ebeveyn = [0, 0, 1, 0, 0]`

**ÇAĞRI YIĞINI**

`find(4)`

`main()`

# Kod İzleme: döngüyle `find` + yol kısaltma




| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| ebeveyn | 0 | 0 | 1 | 0 | 0 |

Yolda bir yukarı: **x = 2**.

Adım 11 / 16

```
1  int find(int x) {
2    int kok = x;
3    while (ebeveyn[kok] != kok) kok = ebeveyn[kok];
4    while (ebeveyn[x] != kok) {    // yol kısaltma
5      int sonraki = ebeveyn[x];
6      ebeveyn[x] = kok;
▶      x = sonraki;
8    }
9    return kok;
10 }
```

**DEĞİŞKENLER**

`x = 2` `kok = 0` `sonraki = 2`

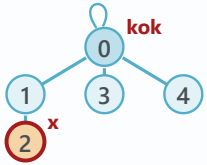
`ebeveyn = [0, 0, 1, 0, 0]`

**ÇAĞRI YIĞINI**

`find(4)`

`main()`

# Kod İzleme: döngüyle `find` + yol kısaltma




| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| ebeveyn | 0 | 0 | 1 | 0 | 0 |

2. tur: ebeveyn[2] = 1, kök değil → bu düğüm kısaltılacak.

Adım 12 / 16

```
1  int find(int x) {
2    int kok = x;
3    while (ebeveyn[kok] != kok) kok = ebeveyn[kok];
▶    while (ebeveyn[x] != kok) {    // yol kısaltma
5      int sonraki = ebeveyn[x];
6      ebeveyn[x] = kok;
7      x = sonraki;
8    }
9    return kok;
10 }
```

**DEĞİŞKENLER**

`x = 2` `kok = 0`

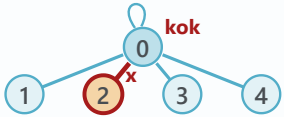
`ebeveyn = [0, 0, 1, 0, 0]`

**ÇAĞRI YIĞINI**

`find(4)`

`main()`

# Kod İzleme: döngüyle `find` + yol kısaltma




| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| ebeveyn | 0 | 0 | 0 | 0 | 0 |

Önce eski ebeveyn saklanır (**sonraki = 1**), sonra **ebeveyn[2] = 0**: 2 artık doğrudan köke bağlı.

Adım 13 / 16

```
1  int find(int x) {
2    int kok = x;
3    while (ebeveyn[kok] != kok) kok = ebeveyn[kok];
4    while (ebeveyn[x] != kok) {    // yol kısaltma
▶      int sonraki = ebeveyn[x];
▶      ebeveyn[x] = kok;
7      x = sonraki;
8    }
9    return kok;
10 }
```

**DEĞİŞKENLER**

`x = 2` `kok = 0` `sonraki = 1`

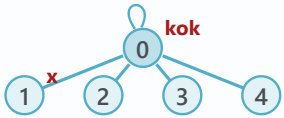
`ebeveyn = [0, 0, 0, 0, 0]`

**ÇAĞRI YIĞINI**

`find(4)`

`main()`

# Kod İzleme: döngüyle `find` + yol kısaltma




| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| ebeveyn | 0 | 0 | 0 | 0 | 0 |

Yolda bir yukarı: **x = 1**.



```
1  int find(int x) {
2    int kok = x;
3    while (ebeveyn[kok] != kok) kok = ebeveyn[kok];
4    while (ebeveyn[x] != kok) {    // yol kısaltma
5      int sonraki = ebeveyn[x];
6      ebeveyn[x] = kok;
▶      x = sonraki;
8    }
9    return kok;
10 }
```

**DEĞİŞKENLER**

`x = 1` `kok = 0` `sonraki = 1`

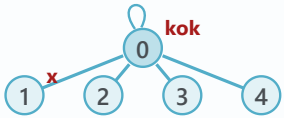
`ebeveyn = [0, 0, 0, 0, 0]`

**ÇAĞRI YIĞINI**

`find(4)`

`main()`

# Kod İzleme: döngüyle `find` + yol kısaltma




| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| ebeveyn | 0 | 0 | 0 | 0 | 0 |

ebeveyn[1] = 0 = kok: 1 zaten köke bağlı, döngü biter.

Adım 15 / 16

```
1  int find(int x) {
2    int kok = x;
3    while (ebeveyn[kok] != kok) kok = ebeveyn[kok];
▶    while (ebeveyn[x] != kok) {    // yol kısaltma
5      int sonraki = ebeveyn[x];
6      ebeveyn[x] = kok;
7      x = sonraki;
8    }
9    return kok;
10 }
```

**DEĞİŞKENLER**

`x = 1` `kok = 0`

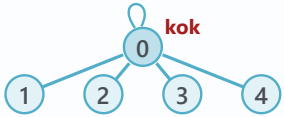
`ebeveyn = [0, 0, 0, 0, 0]`

**ÇAĞRI YIĞINI**

`find(4)`

`main()`

# Kod İzleme: döngüyle `find` + yol kısaltma




| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| ebeveyn | 0 | 0 | 0 | 0 | 0 |

Kök **0** döner. Yol üzerindeki herkes köke bağlandı: sonraki find'lar 1 adım. Özyineleme yok → yığın taşması riski de yok.

Adım 16 / 16

```
1  int find(int x) {
2    int kok = x;
3    while (ebeveyn[kok] != kok) kok = ebeveyn[kok];
4    while (ebeveyn[x] != kok) {    // yol kısaltma
5      int sonraki = ebeveyn[x];
6      ebeveyn[x] = kok;
7      x = sonraki;
8    }
▶    return kok;
10 }
```

**DEĞİŞKENLER**

`x = 1` `kok = 0`

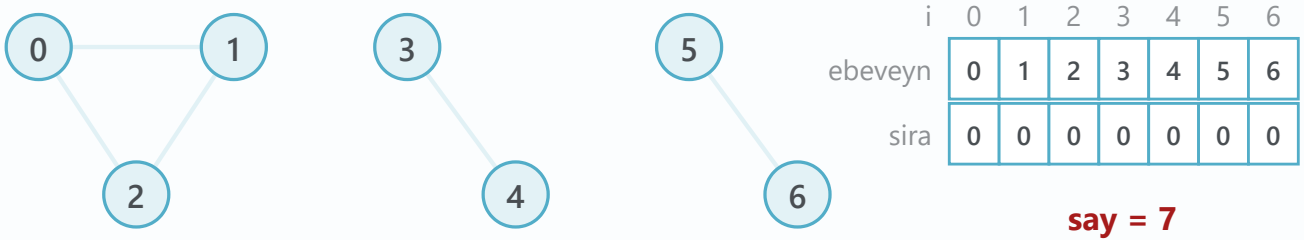
`ebeveyn = [0, 0, 0, 0, 0]` `dönüş = 0`

**ÇAĞRI YIĞINI**

`find(4)`

`main()`

# Kod İzleme: Union-Find ile bağlantılı bileşen sayısı




7 düğüm, 5 kenar. `UnionFind(7)` : her düğüm kendi kümesinin kökü (ebeveyn[i] = i).

Adım 1 / 12

```
1 int bilesenSay(int n, int[][] kenar) {
▶   UnionFind uf = new UnionFind(n);
3   int say = n;          // n tekil küme
4   for (int[] e : kenar)
5     if (uf.union(e[0], e[1]))
6       say--;            // iki bileşen birleşti
7   return say;
8 }
```

**DEĞİŞKENLER**

`n = 7`

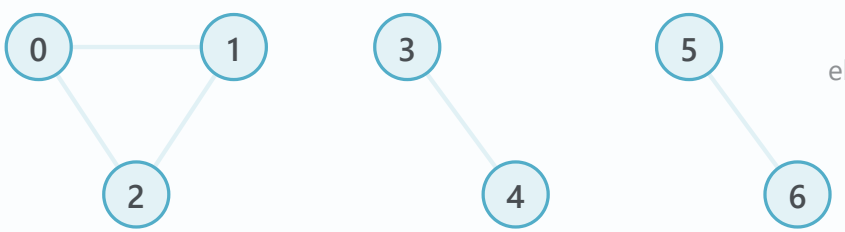
`kenar = [[0,1], [1,2], [3,4], [0,2], [5,6]]`

**ÇAĞRI YIĞINI**

`bilesenSay(7, kenar)`

`main()`

# Kod İzleme: Union-Find ile bağlantılı bileşen sayısı




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

**say = 7**

Başta hiç kenar yok: **7 bileşen**.

Adım 2 / 12

```
1 int bilesenSay(int n, int[][] kenar) {
2   UnionFind uf = new UnionFind(n);
▶   int say = n;          // n tekil küme
4   for (int[] e : kenar)
5     if (uf.union(e[0], e[1]))
6       say--;            // iki bileşen birleşti
7   return say;
8 }
```

**DEĞİŞKENLER**

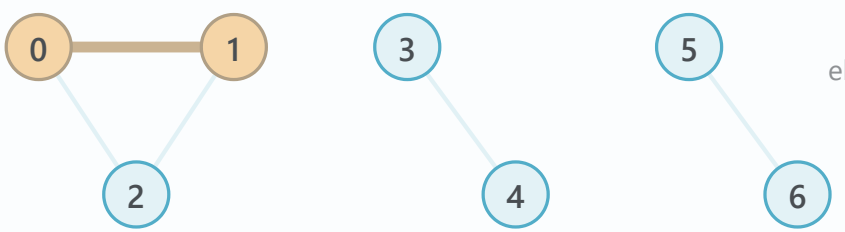
`n = 7` `say = 7`

**ÇAĞRI YIĞINI**

`bilesenSay(7, kenar)`

`main()`

# Kod İzleme: Union-Find ile bağlantılı bileşen sayısı




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

**say = 7**

Kenar (0, 1): kökler farklı → `union` kümeleri birleştirir, **true** döner.

Adım 3 / 12

```
1 int bilesenSay(int n, int[][] kenar) {
2   UnionFind uf = new UnionFind(n);
3   int say = n;           // n tekil küme
▶   for (int[] e : kenar)
▶     if (uf.union(e[0], e[1]))
6       say--;             // iki bileşen birleşti
7   return say;
8 }
```

**DEĞİŞKENLER**

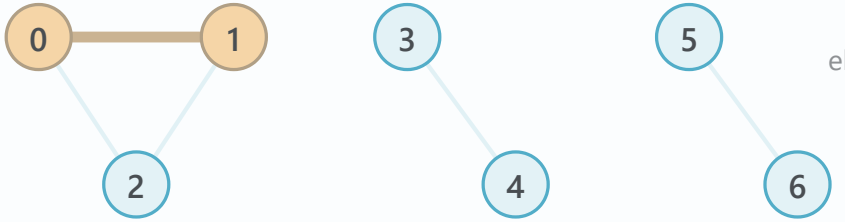
`n = 7` `say = 7` `e = [0, 1]` `union = true`

**ÇAĞRI YIĞINI**

`bilesenSay(7, kenar)`

`main()`

# Kod İzleme: Union-Find ile bağlantılı bileşen sayısı




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

**say = 6**

İki bileşen bire indi: **say = 6**.

Adım 4 / 12

```
1 int bilesenSay(int n, int[][] kenar) {
2   UnionFind uf = new UnionFind(n);
3   int say = n;          // n tekil küme
4   for (int[] e : kenar)
5     if (uf.union(e[0], e[1]))
▶       say--;            // iki bileşen birleşti
7   return say;
8 }
```

**DEĞİŞKENLER**

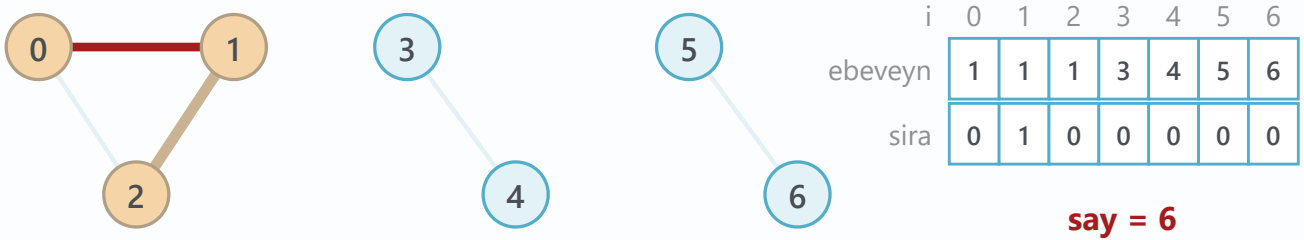
`n = 7` `say = 6` `e = [0, 1]` `union = true`

**ÇAĞRI YIĞINI**

`bilesenSay(7, kenar)`

`main()`

# Kod İzleme: Union-Find ile bağlantılı bileşen sayısı




Kenar (1, 2): kökler farklı → `union` kümeleri birleştirir, **true** döner.

Adım 5 / 12

```
1 int bilesenSay(int n, int[][] kenar) {
2   UnionFind uf = new UnionFind(n);
3   int say = n;           // n tekil küme
▶   for (int[] e : kenar)
▶     if (uf.union(e[0], e[1]))
6       say--;             // iki bileşen birleşti
7   return say;
8 }
```

**DEĞİŞKENLER**

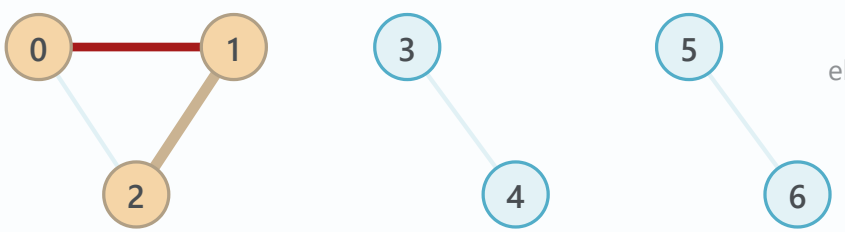
`n = 7` `say = 6` `e = [1, 2]` `union = true`

**ÇAĞRI YIĞINI**

`bilesenSay(7, kenar)`

`main()`

# Kod İzleme: Union-Find ile bağlantılı bileşen sayısı




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

**say = 5**

İki bileşen bire indi: **say = 5**.

Adım 6 / 12

```
1 int bilesenSay(int n, int[][] kenar) {
2   UnionFind uf = new UnionFind(n);
3   int say = n;          // n tekil küme
4   for (int[] e : kenar)
5     if (uf.union(e[0], e[1]))
▶       say--;            // iki bileşen birleşti
7   return say;
8 }
```

**DEĞİŞKENLER**

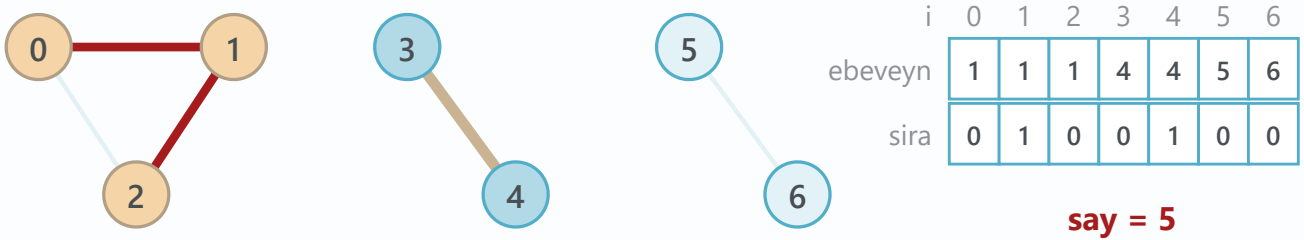
`n = 7` `say = 5` `e = [1, 2]` `union = true`

**ÇAĞRI YIĞINI**

`bilesenSay(7, kenar)`

`main()`

# Kod İzleme: Union-Find ile bağlantılı bileşen sayısı




Kenar (3, 4): kökler farklı → `union` kümeleri birleştirir, **true** döner.

Adım 7 / 12

```
1 int bilesenSay(int n, int[][] kenar) {
2   UnionFind uf = new UnionFind(n);
3   int say = n;           // n tekil küme
▶   for (int[] e : kenar)
▶     if (uf.union(e[0], e[1]))
6       say--;             // iki bileşen birleşti
7   return say;
8 }
```

**DEĞİŞKENLER**

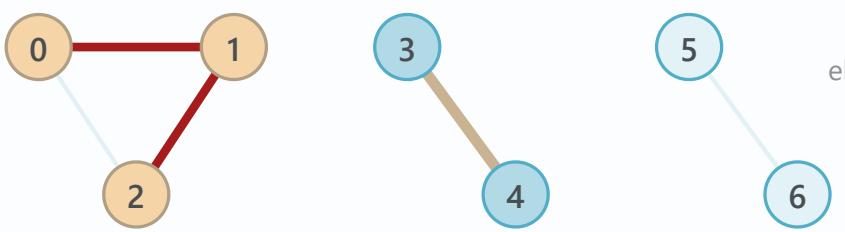
`n = 7` `say = 5` `e = [3, 4]` `union = true`

**ÇAĞRI YIĞINI**

`bilesenSay(7, kenar)`

`main()`

# Kod İzleme: Union-Find ile bağlantılı bileşen sayısı




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

**say = 4**

İki bileşen bire indi: **say = 4**.

Adım 8 / 12

```
1 int bilesenSay(int n, int[][] kenar) {
2   UnionFind uf = new UnionFind(n);
3   int say = n;           // n tekil küme
4   for (int[] e : kenar)
5     if (uf.union(e[0], e[1]))
▶       say--;             // iki bileşen birleşti
7   return say;
8 }
```

**DEĞİŞKENLER**

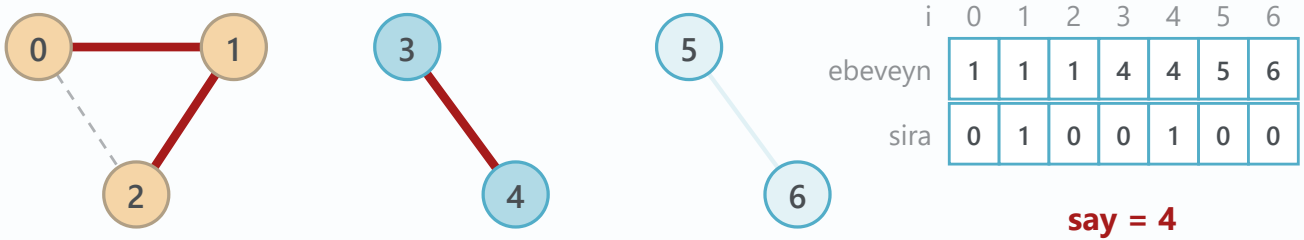
`n = 7` `say = 4` `e = [3, 4]` `union = true`

**ÇAĞRI YIĞINI**

`bilesenSay(7, kenar)`

`main()`

# Kod İzleme: Union-Find ile bağlantılı bileşen sayısı




Kenar (0, 2): find(0) = find(2) → zaten **aynı bileşende**; `union` false, say değişmez (bu kenar bir döngü kapatır).

Adım 9 / 12

```
1 int bilesenSay(int n, int[][] kenar) {
2   UnionFind uf = new UnionFind(n);
3   int say = n;           // n tekil küme
▶   for (int[] e : kenar)
▶     if (uf.union(e[0], e[1]))
6       say--;             // iki bileşen birleşti
7   return say;
8 }
```

**DEĞİŞKENLER**

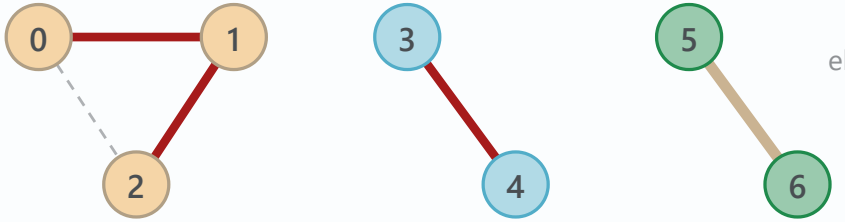
`n = 7` `say = 4` `e = [0, 2]` `union = false`

**ÇAĞRI YIĞINI**

`bilesenSay(7, kenar)`

`main()`

# Kod İzleme: Union-Find ile bağlantılı bileşen sayısı




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

**say = 4**

Kenar (5, 6): kökler farklı → `union` kümeleri birleştirir, **true** döner.

Adım 10 / 12

```
1 int bilesenSay(int n, int[][] kenar) {
2   UnionFind uf = new UnionFind(n);
3   int say = n;           // n tekil küme
▶   for (int[] e : kenar)
▶     if (uf.union(e[0], e[1]))
6       say--;             // iki bileşen birleşti
7   return say;
8 }
```

**DEĞİŞKENLER**

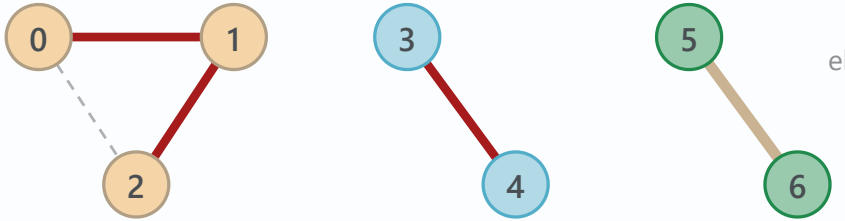
`n = 7` `say = 4` `e = [5, 6]` `union = true`

**ÇAĞRI YIĞINI**

`bilesenSay(7, kenar)`

`main()`

# Kod İzleme: Union-Find ile bağlantılı bileşen sayısı




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

**say = 3**

İki bileşen bire indi: **say = 3**.

Adım 11 / 12

```
1 int bilesenSay(int n, int[][] kenar) {
2   UnionFind uf = new UnionFind(n);
3   int say = n;          // n tekil küme
4   for (int[] e : kenar)
5     if (uf.union(e[0], e[1]))
▶       say--;            // iki bileşen birleşti
7   return say;
8 }
```

**DEĞİŞKENLER**

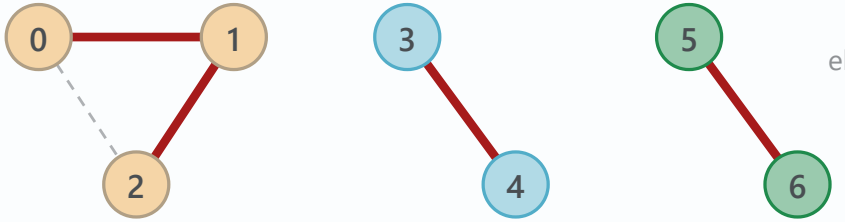
`n = 7` `say = 3` `e = [5, 6]` `union = true`

**ÇAĞRI YIĞINI**

`bilesenSay(7, kenar)`

`main()`

# Kod İzleme: Union-Find ile bağlantılı bileşen sayısı




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

**say = 3**

Sonuç **3 bileşen**: {0, 1, 2}, {3, 4}, {5, 6}. Toplam O(n + m·α(n)); kenarlar sonradan gelse de sayaç anında güncel (dinamik bağlılık).

Adım 12 / 12

```
1 int bilesenSay(int n, int[][] kenar) {
2   UnionFind uf = new UnionFind(n);
3   int say = n;          // n tekil küme
4   for (int[] e : kenar)
5     if (uf.union(e[0], e[1]))
6       say--;            // iki bileşen birleşti
▶   return say;
8 }
```

**DEĞİŞKENLER**

`n = 7` `say = 3` `dönüş = 3`

**ÇAĞRI YIĞINI**

`bilesenSay(7, kenar)`

`main()`

ÖZET

# Sık Yapılan Hatalar

### 1. Kümedeki öğeyi değiştirmek

Eklendikten sonra hashCode'u değişen nesne yanlış kovada kalır: `contains` false döner.

```
p.x = 5; set.contains(p) → false
```

### 2. equals var, hashCode yok

"Eşit" iki nesne büyük olasılıkla farklı kovalara düşer → HashSet'te **iki kez** bulunur. İkisi birlikte ezilmeli.

### 3. Gezinirken silmek

for-each içinde `set.remove(x)` → `ConcurrentModificationException`.

```
set.removeIf(x -> x % 2 == 0);
```

### 4. Kökler yerine öğeleri bağlamak

`ebeveyn[x] = y` yanlış: x'in ağacı bölünür. Doğrusu:

```
ebeveyn[find(x)] = find(y);
```

### 5. Eşit rütbede artırmayı unutmak

`sira[ky]++` yoksa rütbe yüksekliği izlemez; O(log n) garantisi kaybolur.

### 6. TreeSet ile tutarsız compare

`compare(a, b) == 0` olan farklı nesnelerden yalnız biri saklanır (ör. yalnız soyadına göre sıralama).

# Laboratuvar: Küme İşlemleri

Sıralı birleşim ve kesişim: {1, 3, 5, 7, 9} ve {3, 4, 5, 8}

Girdi (sıralı a | sıralı b) `1, 3, 5, 7, 9 | 3, 4, 5, 8`

KOD · **adım 1 / 22**

```
 1 void birlesimKesisim(int[] a, int[] b,
 2     List<Integer> bir, List<Integer> ke…
 3   int i = 0, j = 0;
 4   while (i < a.length && j < b.length) {
 5     if (a[i] < b[j]) bir.add(a[i++]);
 6     else if (a[i] > b[j]) bir.add(b[j++…
 7     else {                        // ortak…
 8       bir.add(a[i]);  kes.add(a[i]);
 9       i++;  j++;
10     }
11   }
12   while (i < a.length) bir.add(a[i++]);
13   while (j < b.length) bir.add(b[j++]);
```

a = {1, 3, 5, 7, 9} ve b = {3, 4, 5, 8} sıralı dizileriyle çağrılır.

a 1 3 5 7 9

b 3 4 5 8

bir boş

kes boş

—

SONUÇ –

ÖZET

# Özet

| Yapı | Temel işlemler | Maliyet | Ne zaman? |
|---|---|---|---|
| **HashSet** | add, remove, contains | O(1)* | sıra gerekmiyorsa |
| **TreeSet** | + first, floor, aralık | O(log n) | sıralı erişim gerekiyorsa |
| **Çoklu küme** | add, count, remove | O(1)* | tekrar sayısı önemliyse |
| **Union-Find** | find, union | O(α(n)) | ayrık kümeler, bağlantı sorguları |

* ortalama durum. Union-Find maliyeti rütbe + yol kısaltma ile amortize.

**Seçim:** üyelik için `HashSet`, sıralı erişim için `TreeSet`, dinamik bağlılık için Union-Find.

ÖZET

# Etkileşimli Simülatörler

## Küme ADT

Öğe ekleme/çıkarma, üyelik testi, birleşim, kesişim ve fark işlemleri.

## Ayrık Küme Birleşimi (Union-Find)

find ve union işlemleri, rütbeye göre birleştirme ve yol kısaltma, ağaç üzerinde.

ÖZET

# Örnek Problemler

1. A = {2, 4, 6, 8}, B = {1, 2, 3, 4} için A ∪ B, A ∩ B, B − A nedir?
2. Bir sözlükteki kelimeleri alfabetik sırayla ve tekrarsız yazdırmak için hangi Set uygulamasını seçersiniz?
3. "kavak" kelimesinin harflerini çoklu kümede tutun: count('a') ve size() ne döner?
4. 0..5 için sırayla union(0,1), union(2,3), union(1,3), union(4,5) yapılıyor (rütbeye göre, eşitlikte ebeveyn[x] = y). Kaç küme kalır? find(0) nedir?
5. 0..4 için sırayla union(0,1), union(1,2), union(3,4), union(2,0) yapılıyor. Hangi union false döner? Kaç bağlantılı bileşen kalır?

▶ **Cevapları göster**

ÖZET

# Alıştırmalar

1. Boş HashSet'e (16 kova, kova = x & 15) 12, 28, 44, 7 eklenir. Kovalar? `add(28)` ne döner?
2. U = {0..7}: A = 1011 $0010_2$, B = 0111 $0100_2$. A | B, A & B, A & ~B kümeleri nedir?
3. Quick-Find ile n öğede n − 1 union: toplam kaç hücreye bakılır?
4. `ebeveyn = [1, 2, 3, 3, 3]` iken yinelemeli `find(0)` sonrası `ebeveyn`?
5. Rütbeye göre birleştirmede rütbe en fazla kaç olabilir? Neden?
6. Kenarlar tek tek geliyor; her kenardan sonra "çizge bağlı mı?" Hangi yapı, maliyet?

▶ **Cevaplar / 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.
- Sedgewick ve Wayne, *Algorithms*, 4. baskı, 2011.
- Oracle, Java SE 17 `Set`, `HashSet` ve `TreeSet` API belgeleri.

# Son

## Bölüm 13: Kümeler ve Ayrık Kümeler

Sonraki sunum: Bölüm 14: Çizgeler · Çalışma soruları · Simülatörler: Küme ADT Simülatörü, Ayrık Kümeler Simülatörü · Terim sözlüğü