Bölüm 8: Kuyruklar ve Deque

Veri Yapıları

FIFO | enqueue / dequeue | Dizi ile kuyruk | Dairesel kuyruk | Bağlı liste ile kuyruk | Deque | Kayan pencere | Round robin

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

GİRİŞ

Öğrenme Çıktıları

FIFO modeli

Kuyruğun ilk giren ilk çıkar davranışını açıklamak.

Gerçekleme

Kuyruğu dizi ve bağlı liste kullanarak gerçeklemek.

Kuyruk türleri

Dairesel kuyruk ile çift uçlu kuyruğu (deque) ayırt etmek ve uygun yerde kullanmak.

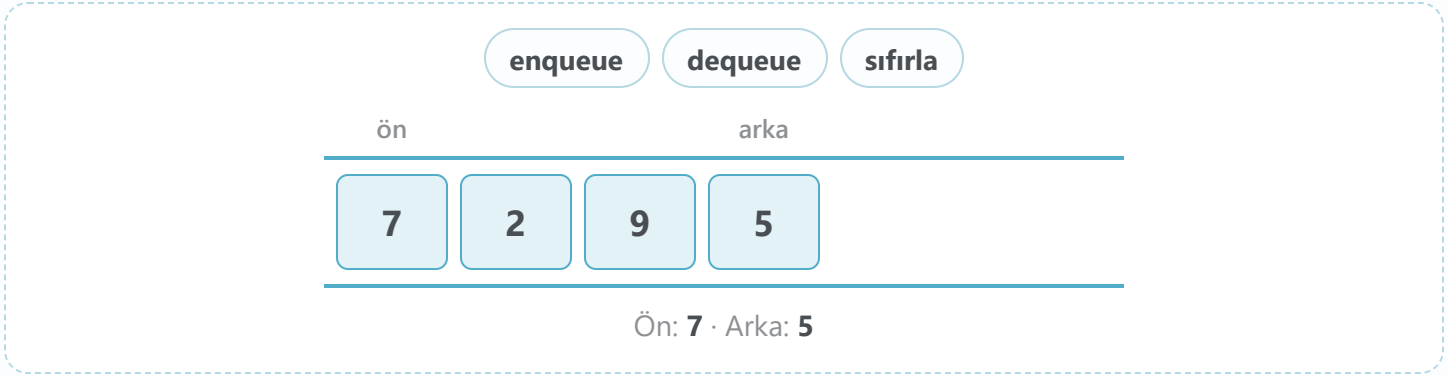
Uygulamalar

Uygun kuyruk türünü gerçek yaşam problemlerinde seçmek.

KUYRUK ADT

Kuyruk (Queue)

- Her iki ucu açık, **doğrusal** bir veri yapısıdır.
- İşlemler **ilk giren, ilk çıkar** (FIFO - First-In, First-Out) sırasıyla yapılır.
- Ekleme bir uçtan (**arka**, rear), çıkarma diğer uçtan (**ön**, front) yapılır.
- Market kasasındaki, bilet gişesindeki sıra gibi: önce gelen önce hizmet alır.



KUYRUK ADT

Temel Kuyruk İşlemleri

Ekleme (enqueue)

Kuyruğun sonuna (arkasına) yeni bir öğe ekler.

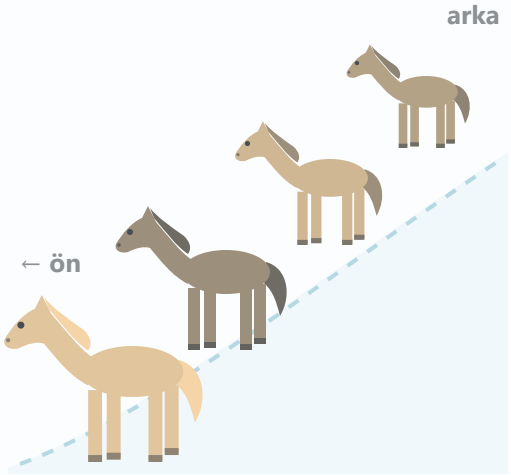
`Zaman karmaşıklığı: O(1)`

Çıkarma (dequeue)

Kuyruğun başındaki (önündeki) öğeyi çıkarır ve döndürür. İlk giren öğe ilk çıkar (FIFO ilkesi).

`Zaman karmaşıklığı: O(1)`

Kuyrukta ortadaki bir öğeye doğrudan erişilmez; yalnızca **ön** ve **arka** uçlar kullanılır.



Tek sıra yürüyen atlar: öndeki önce geçer, yeni gelen sona katılır.

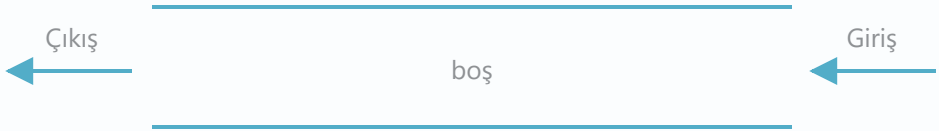
Kuyruk Soyut Veri Tipi (Queue ADT)

İşlem	Açıklama
`enqueue(e)`	e'yi arkaya ekler
`dequeue()`	öndeki öğeyi çıkarır ve döndürür
`first()` / `peek()`	öndeki öğeyi döndürür, çıkarmaz
`rear()`	arkadaki öğeyi döndürür, çıkarmaz
`size()`	öğe sayısı
`isEmpty()`	kuyruk boş mu?
`isFull()`	dolu mu? (yalnızca sabit kapasitede)

```
public interface Queue<E> {
    int size();
    boolean isEmpty();
    void enqueue(E e);
    E first();    // boşsa null
    E dequeue();  // boşsa null
}
```

Boş kuyruktan çıkarma **alt taşma** (underflow), dolu kuyruğa ekleme **taşma** (overflow) hatasıdır. Goodrich'in arayüzü boş kuyrukta `null` döndürür; bazı gerçeklemeler istisna (exception) fırlatır.

Kuyruk İşlemleri Adım Adım



on = arka = null

Başlangıçta kuyruk boştur: `on` ve `arka` hiçbir öğeyi göstermez (null).

Kuyruk İşlemleri Adım Adım

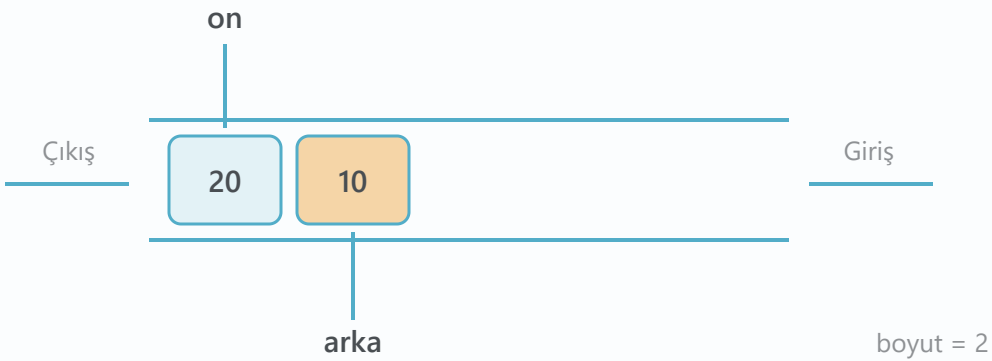
enqueue(20)



İlk öğe eklenir: tek öğe hem **ön** hem **arka**dır.

Kuyruk İşlemleri Adım Adım

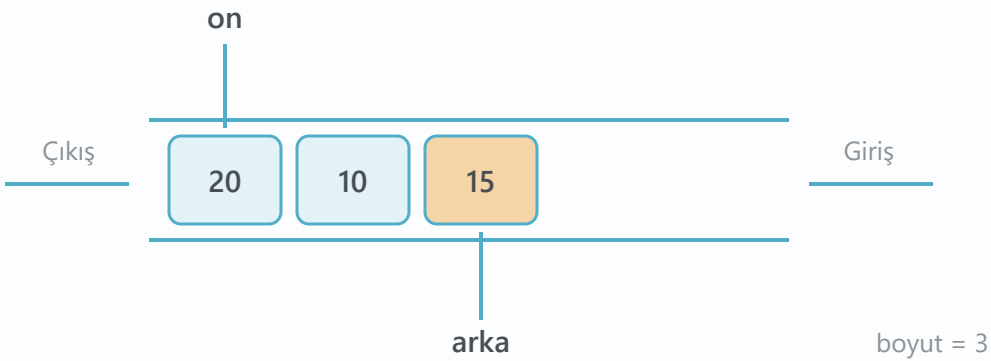
enqueue(10)



10 arkaya eklenir: `arka` ilerler, `on` yerinde kalır.

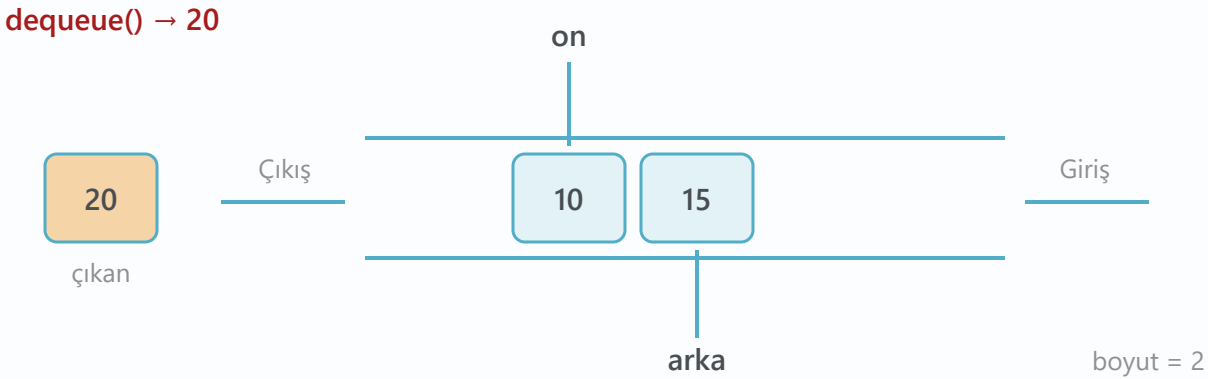
Kuyruk İşlemleri Adım Adım

enqueue(15)



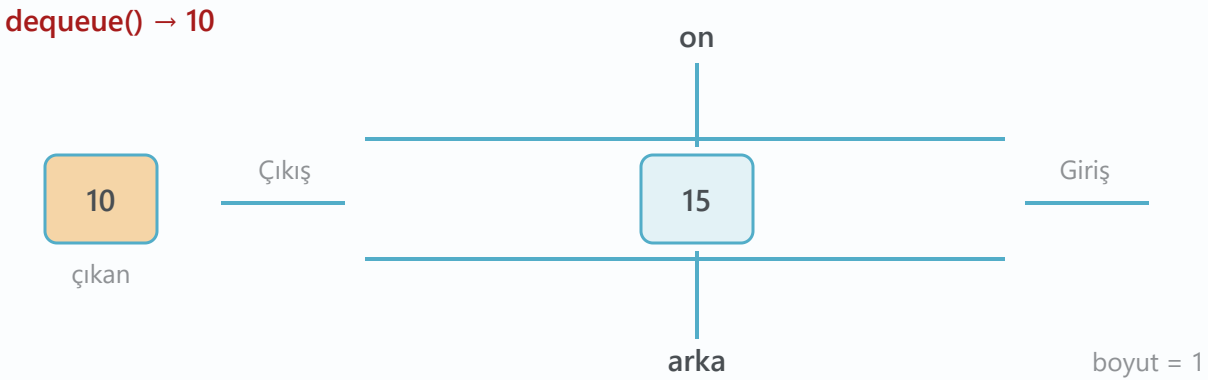
15 arkaya eklenir. Kuyruk: 20, 10, 15 (önden arkaya).

Kuyruk İşlemleri Adım Adım



Öndeki 20 çıkar; `on` bir sonraki öğeye (10) ilerler.

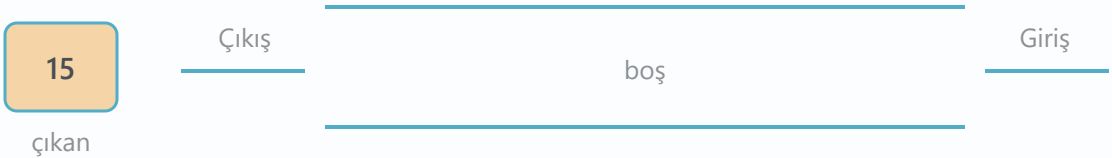
Kuyruk İşlemleri Adım Adım



10 çıkar. Şimdi `on` ve `arka` aynı öğeyi (15) gösterir.

Kuyruk İşlemleri Adım Adım

dequeue() → 15



on = arka = null

Son öğe çıkınca ikisi de null olur. Çıkış sırası 20, 10, 15 = giriş sırası (FIFO).

Java'da Kuyruk: `java.util.Queue`

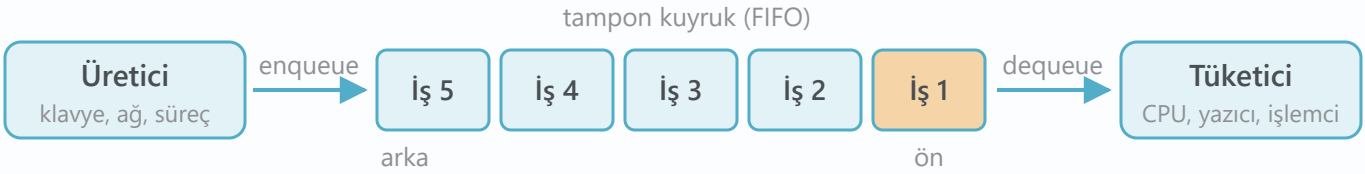
İşlem	Hata fırlatır	Özel değer döner
Ekleme	`add(e)`	`offer(e)` → false
Çıkarma	`remove()`	`poll()` → null
İnceleme	`element()`	`peek()` → null

- `ArrayDeque` : dairesel dizi ile gerçeklenir, genellikle en hızlı seçenektir.
- `LinkedList` : bağlı liste ile gerçeklenir.

```
Queue<Integer> q = new ArrayDeque<>();
q.offer(20);
q.offer(10);
q.offer(15);        // [20, 10, 15]
q.peek();           // 20, çıkarmaz
q.poll();           // 20 → [10, 15]
q.poll();           // 10 → [15]
q.size();           // 1
q.poll();           // 15 → []
q.poll();           // null (boş)
```

Kendi sınıflarımızda `enqueue` / `dequeue` adları, Java kütüphanesinde ise `offer` / `poll` / `peek` üçlüsü kullanılır.

Kuyrukların Kullanım Alanları



İşletim Sistemi Çizelgeleyicisi

Çalışmaya hazır süreçler hazır kuyruğunda bekler; CPU sırayla dağıtılır.

Yazdırma Kuyruğu

Yazıcıya gönderilen belgeler geliş sırasıyla basılır.

Tamponlar (Buffer)

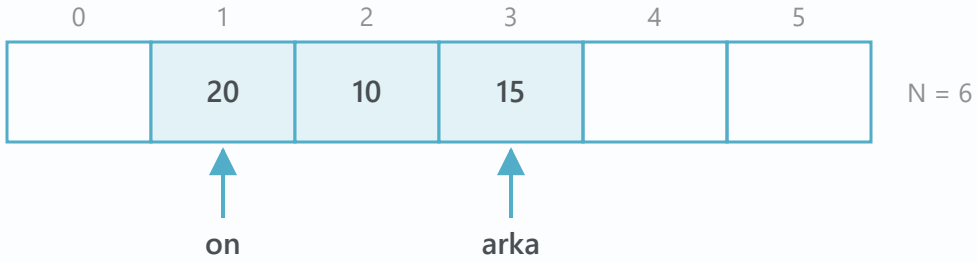
Klavye girdisi, ağ paketleri, video akışı: üretici ile tüketici arasındaki hız farkını dengeler.

Genişlik Öncelikli Arama (BFS)

Keşfedilen düğümler kuyrukta sıraya alınır; düzey düzey gezinme sağlar.

Dizi ile Kuyruk

- Öğeler bir dizide tutulur; iki indis kuyruğun uçlarını gösterir:
  - **on** (ön, front): ilk öğenin indisi.
  - **arka** (rear): son öğenin indisi.
- Eklemede `arka`, çıkarmada `on` bir artar. Öğeler yerinden kaydırılmaz.
- Basit ve hızlıdır, ancak dizinin **kapasitesi sabittir**.



İndis 0'daki öğe daha önce çıkarıldı; o yuva artık boş ama basit dizi kuyruğu onu yeniden kullanamaz.

Basit Dizi Kuyruğunun Sorunu



boş: on = 0, arka = −1

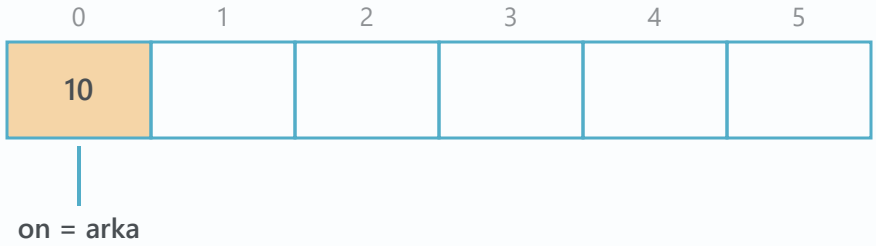
Boş kuyruk, kapasite N = 6. `arka = −1` : henüz öğe yok.

Adım 1 / 11

Çözüm 1: Kaydırma
Her dequeue'da öğeleri sola kaydır → dequeue O(n).

Çözüm 2: Dairesel dizi
İndisler sona gelince başa sarar → her işlem O(1).

Basit Dizi Kuyruğunun Sorunu



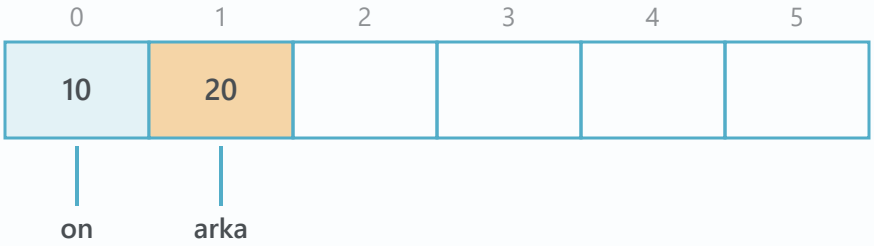
enqueue(10) → arka = 0

`enqueue(10)` : `arka++` → 0, öğe dizi[0]'e yazılır.

Çözüm 1: Kaydırma
Her dequeue'da öğeleri sola kaydır → dequeue O(n).

Çözüm 2: Dairesel dizi
İndisler sona gelince başa sarar → her işlem O(1).

Basit Dizi Kuyruğunun Sorunu



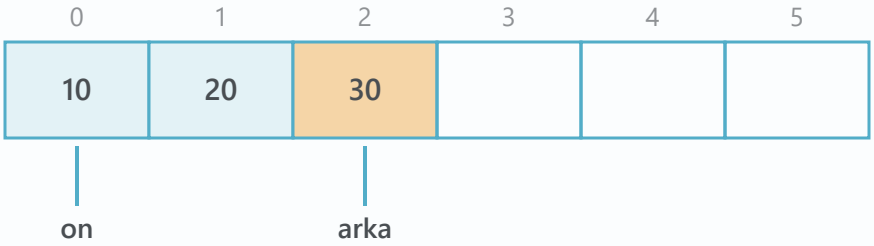
enqueue(20) → arka = 1

`enqueue(20)` : `arka++` → 1, öğe dizi[1]'e yazılır.

Çözüm 1: Kaydırma
Her dequeue'da öğeleri sola kaydır → dequeue O(n).

Çözüm 2: Dairesel dizi
İndisler sona gelince başa sarar → her işlem O(1).

Basit Dizi Kuyruğunun Sorunu



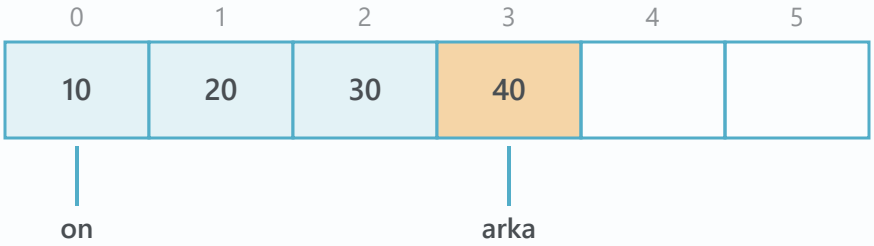
enqueue(30) → arka = 2

`enqueue(30)` : `arka++` → 2, öğe dizi[2]'e yazılır.

Çözüm 1: Kaydırma
Her dequeue'da öğeleri sola kaydır → dequeue O(n).

Çözüm 2: Dairesel dizi
İndisler sona gelince başa sarar → her işlem O(1).

Basit Dizi Kuyruğunun Sorunu



enqueue(40) → arka = 3

`enqueue(40)` : `arka++` → 3, öğe dizi[3]'e yazılır.

Çözüm 1: Kaydırma
Her dequeue'da öğeleri sola kaydır → dequeue O(n).

Çözüm 2: Dairesel dizi
İndisler sona gelince başa sarar → her işlem O(1).

Basit Dizi Kuyruğunun Sorunu



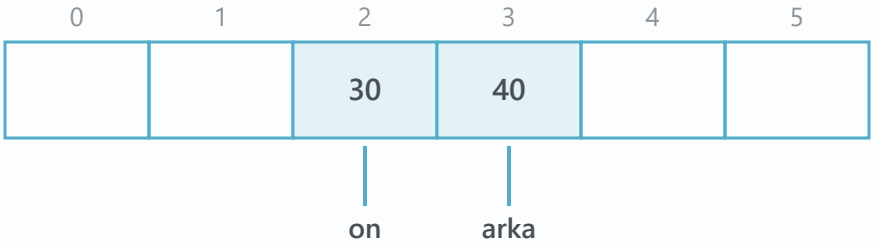
dequeue() → 10, on = 1

`dequeue()` → 10: `on++` → 1. Boşalan yuva bir daha kullanılmaz.

Çözüm 1: Kaydırma
Her dequeue'da öğeleri sola kaydır → dequeue O(n).

Çözüm 2: Dairesel dizi
İndisler sona gelince başa sarar → her işlem O(1).

Basit Dizi Kuyruğunun Sorunu



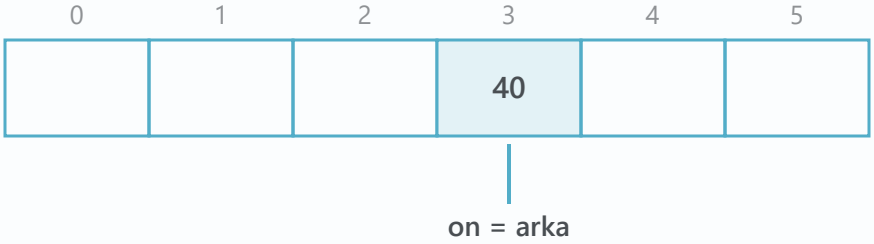
dequeue() → 20, on = 2

`dequeue()` → 20: `on++` → 2. Boşalan yuva bir daha kullanılmaz.

Çözüm 1: Kaydırma
Her dequeue'da öğeleri sola kaydır → dequeue O(n).

Çözüm 2: Dairesel dizi
İndisler sona gelince başa sarar → her işlem O(1).

Basit Dizi Kuyruğunun Sorunu



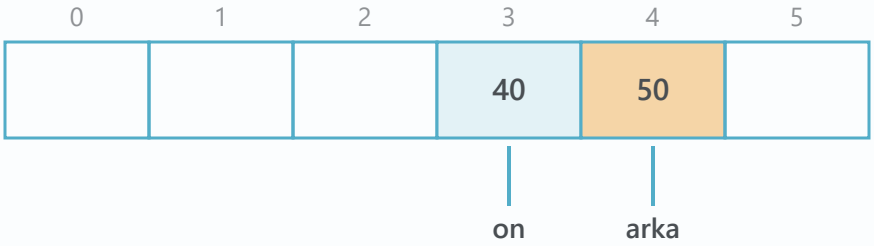
dequeue() → 30, on = 3

`dequeue()` → 30: `on++` → 3. Boşalan yuva bir daha kullanılmaz.

Çözüm 1: Kaydırma
Her dequeue'da öğeleri sola kaydır → dequeue O(n).

Çözüm 2: Dairesel dizi
İndisler sona gelince başa sarar → her işlem O(1).

Basit Dizi Kuyruğunun Sorunu



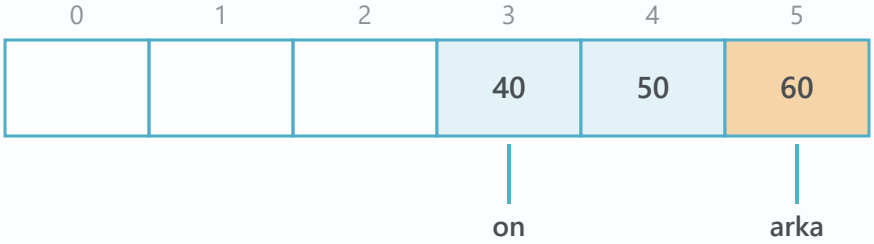
enqueue(50) → arka = 4

`enqueue(50)` : arka = 4.

Çözüm 1: Kaydırma
Her dequeue'da öğeleri sola kaydır → dequeue O(n).

Çözüm 2: Dairesel dizi
İndisler sona gelince başa sarar → her işlem O(1).

Basit Dizi Kuyruğunun Sorunu



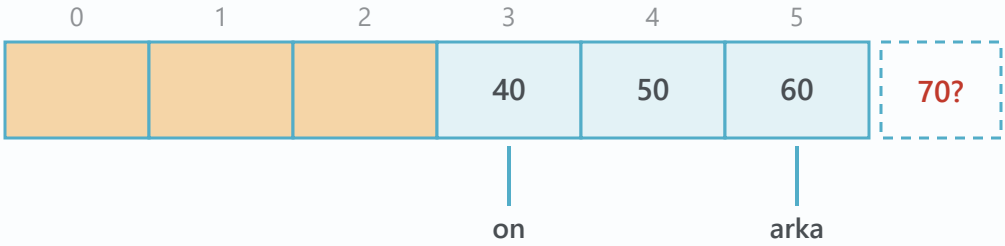
enqueue(60) → arka = 5

`enqueue(60)`: arka = 5 = N − 1, dizinin son indisi.

Çözüm 1: Kaydırma
Her dequeue'da öğeleri sola kaydır → dequeue O(n).

Çözüm 2: Dairesel dizi
İndisler sona gelince başa sarar → her işlem O(1).

Basit Dizi Kuyruğunun Sorunu



enqueue(70): arka + 1 = 6 = N → “dolu”!

Yalnızca 3 öğe varken **“dolu”**: turuncu 3 yuva boşa gidiyor. Kaydırma O(n) sürer; çözüm dairesel kuyruk.

Çözüm 1: Kaydırma

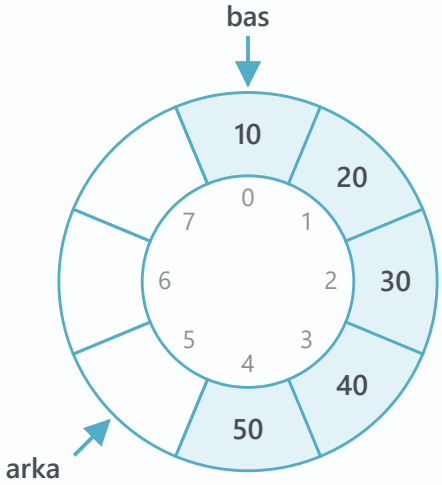
Her dequeue'da öğeleri sola kaydır → dequeue O(n).

Çözüm 2: Dairesel dizi

İndisler sona gelince başa sarar → her işlem O(1).

Dairesel Kuyruk (Circular Queue)

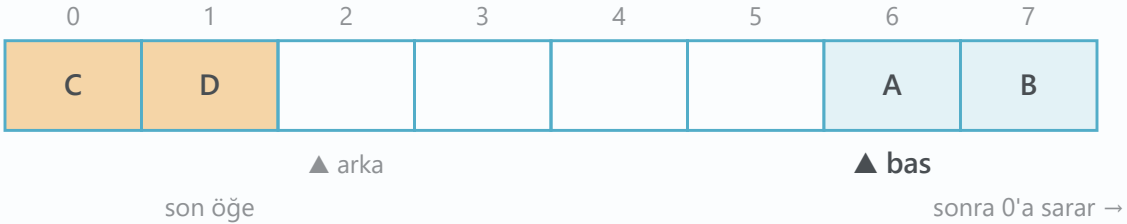
- Normal kuyruk gibi FIFO çalışır; ama dizinin son yuvasından sonra **ilk yuvaya** dönülür.
- Dizinin başı ve sonu mantıksal olarak birleşip bir **halka** oluşturur.
- İndisler **mod (%) işlemiyle** ilerletilir: `(i + 1) % N`.
- Boşalan yuvalar yeniden kullanılır; öğe kaydırmaya gerek kalmaz.
- **Halka tampon** (ring buffer) olarak da bilinir.



Burada **arka**, bir sonraki öğenin yazılacağı boş yuvadır: arka = (bas + boyut) % N = (0 + 5) % 8 = 5.

İndislerin Mod ile Gezinmesi

Ne?	Formül	Örnek: N = 8, bas = 6, boyut = 4
Sonraki boş yuva (ekleme yeri)	`(bas + boyut) % N`	(6 + 4) % 8 = **2**
Son öğe	`(bas + boyut − 1) % N`	(6 + 3) % 8 = **1**
Baştan k. öğe (k = 0, 1, …)	`(bas + k) % N`	k = 2 → (6 + 2) % 8 = **0**
dequeue sonrası bas	`(bas + 1) % N`	(6 + 1) % 8 = **7**



Sıra önden arkaya: A (6), B (7), C (0), D (1). Mod işlemi, dizinin sonundan başına geçişi tek bir ifadeyle çözer.

Dolu mu, Boş mu?



Yalnızca bas ve arka indisleri tutulursa, **boş** ve **dolu** kuyrukta ikisi de aynı yuvayı gösterir; durumlar ayırt edilemez. İki yaygın çözüm:

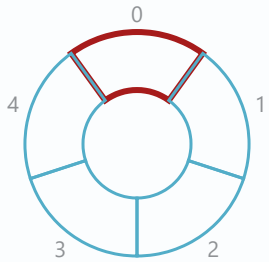
1. Öğe sayacı (boyut)

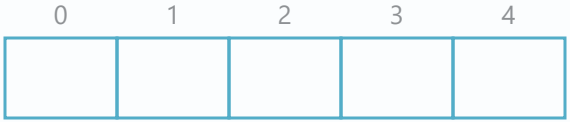
Boş: `boyut == 0` · Dolu: `boyut == N`
Dizinin N yuvasının tamamı kullanılır. Bu derste kullandığımız yöntem.

2. Bir yuvayı boş bırakmak

Boş: `bas == arka` · Dolu: `(arka + 1) % N == bas`
Sayaç gerekmez, ama en fazla N − 1 öğe saklanır.

Dairesel Kuyruk Adım Adım (N = 5)





bas = 0

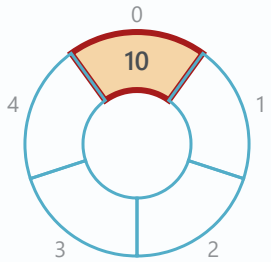
boyut = 0

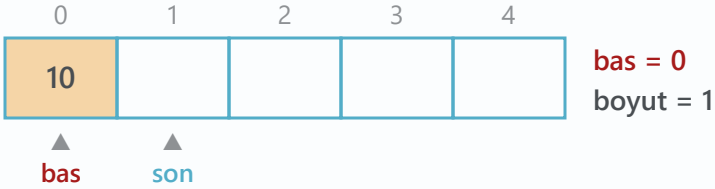
▲
bas, son

son = (bas + boyut) % 5 = (0 + 0) % 5 = 0 · boyut = 0 → BOŞ

Boş kuyruk: boyut = 0. Halkada kırmızı kenarlı dilim **bas**, dizide ▲son sonraki yazma yeridir.

Dairesel Kuyruk Adım Adım (N = 5)



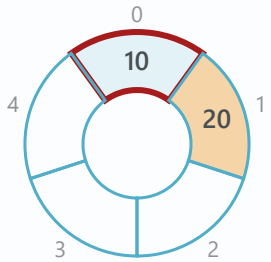


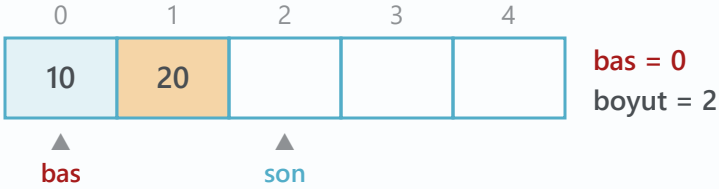
bas = 0
boyut = 1

yazılan yer = (0 + 0) % 5 = 0

`ekle(10)` → dizi[0]; boyut = 1.

Dairesel Kuyruk Adım Adım (N = 5)



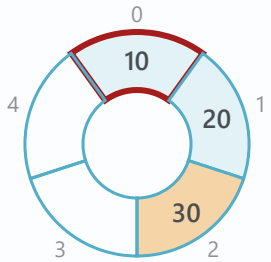


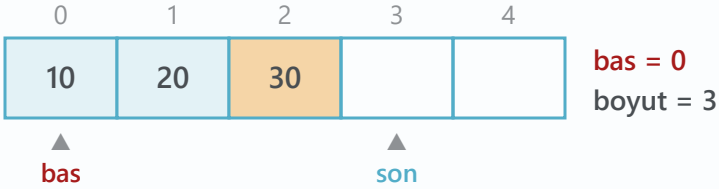
bas = 0
boyut = 2

yazılan yer = (0 + 1) % 5 = 1

`ekle(20)` → dizi[1]; boyut = 2.

Dairesel Kuyruk Adım Adım (N = 5)



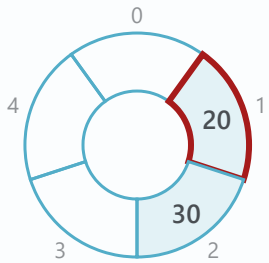


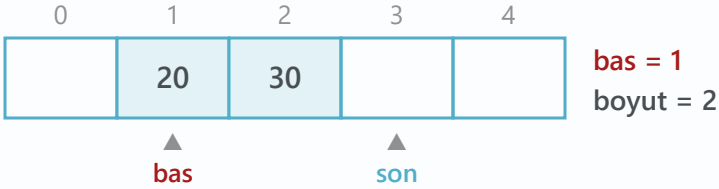
bas = 0
boyut = 3

yazılan yer = (0 + 2) % 5 = 2

`ekle(30)` → dizi[2]; boyut = 3.

Dairesel Kuyruk Adım Adım (N = 5)



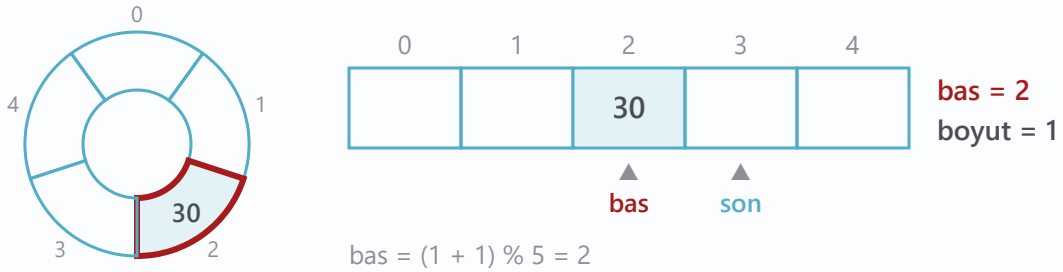


bas = 1
boyut = 2

bas = (0 + 1) % 5 = 1

`cikar()` → 10; bas = (0 + 1) % 5 = 1. Öğeler yerinden oynamaz.

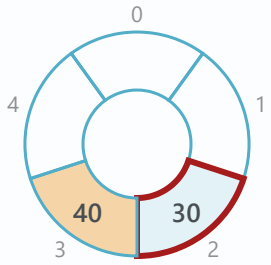
Dairesel Kuyruk Adım Adım (N = 5)

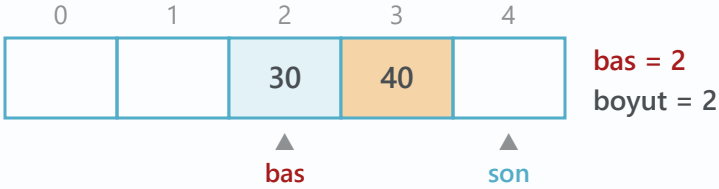


bas = (1 + 1) % 5 = 2

`cikar()` → 20; bas = (1 + 1) % 5 = 2. Öğeler yerinden oynamaz.

Dairesel Kuyruk Adım Adım (N = 5)



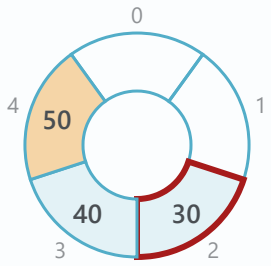


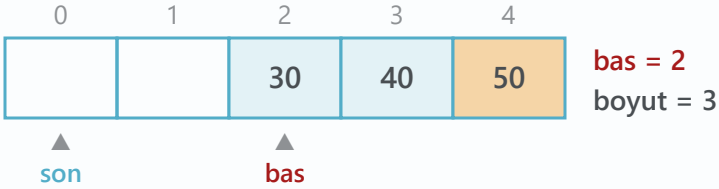
bas = 2
boyut = 2

yazılan yer = (2 + 1) % 5 = 3

`ekle(40)` → dizi[3].

Dairesel Kuyruk Adım Adım (N = 5)



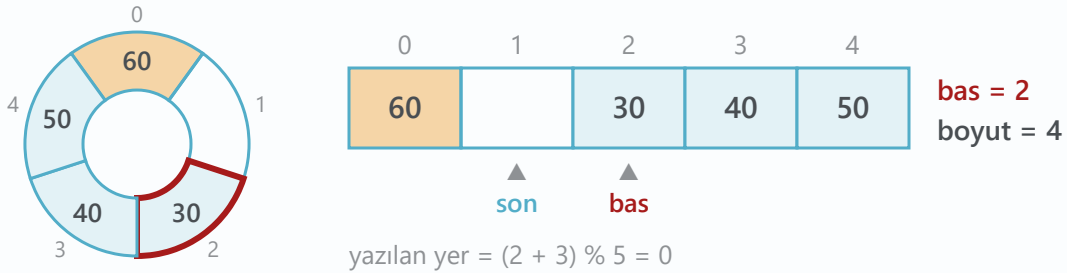


bas = 2
boyut = 3

yazılan yer = (2 + 2) % 5 = 4

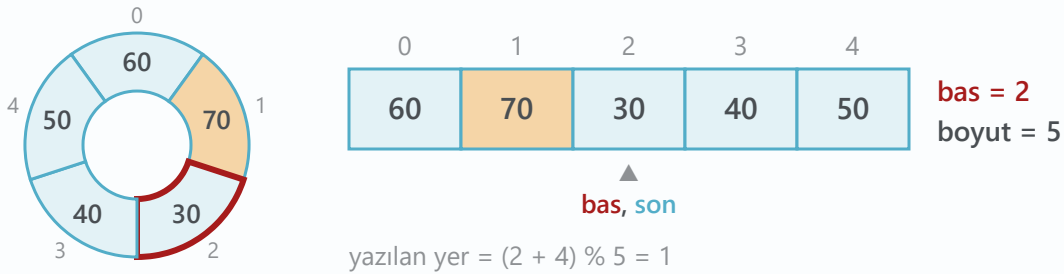
`ekle(50)` → dizi[4].

Dairesel Kuyruk Adım Adım (N = 5)



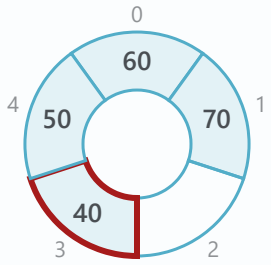
`ekle(60)` → dizi[0]. Dizinin sonu aşıldı, **başa sardı**: basit dizide “dolu” denecek yerde boş yuva kullanıldı.

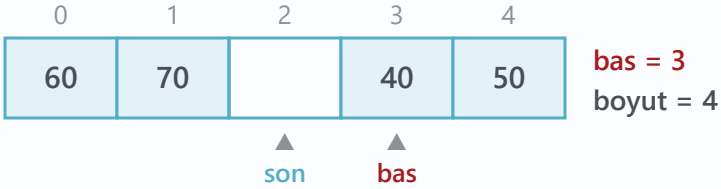
Dairesel Kuyruk Adım Adım (N = 5)



`ekle(70)` → dizi[1]. boyut == N → **dolu**; bas == son olsa da boyut sayesinde boş sanılmaz.

Dairesel Kuyruk Adım Adım (N = 5)



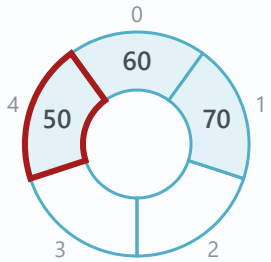


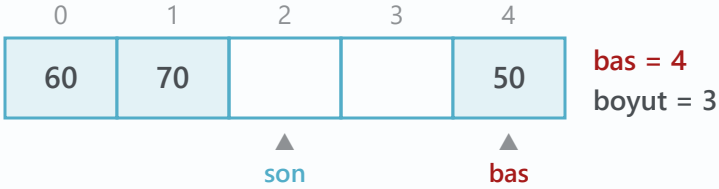
bas = 3
boyut = 4

bas = (2 + 1) % 5 = 3

`cikar()` → 30; bas = 3.

Dairesel Kuyruk Adım Adım (N = 5)



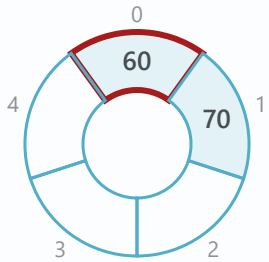


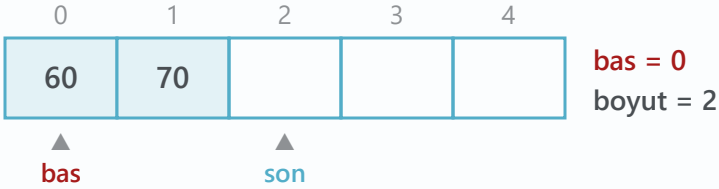
bas = 4
boyut = 3

bas = (3 + 1) % 5 = 4

`cikar()` → 40; bas = 4.

Dairesel Kuyruk Adım Adım (N = 5)



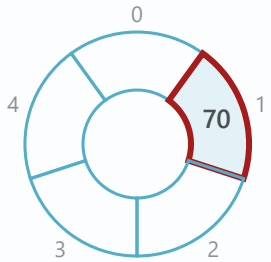


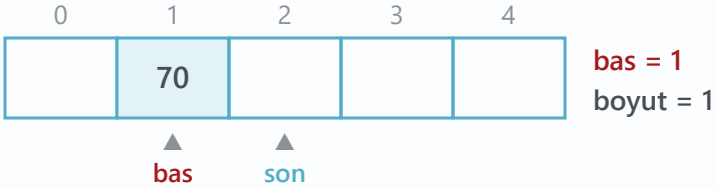
bas = 0
boyut = 2

bas = (4 + 1) % 5 = 0

`cikar()` → 50; bas = 0. (4 + 1) % 5 = 0 → bas da başa sardı.

Dairesel Kuyruk Adım Adım (N = 5)



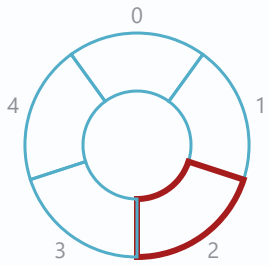


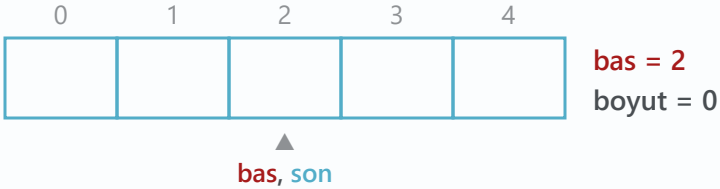
bas = 1
boyut = 1

bas = (0 + 1) % 5 = 1

`cikar()` → 60; bas = 1.

Dairesel Kuyruk Adım Adım (N = 5)





bas = (1 + 1) % 5 = 2

`cikar()` → 70; bas = 2. boyut = 0 → kuyruk yine **boş**; çıkış sırası 30, 40, 50, 60, 70 = giriş sırası (FIFO).

Dairesel Kuyruk: Sınıf ve Durum Sorguları

```
public class DaireselKuyruk<E> {
    private E[] dizi;       // öğeleri saklar
    private int bas = 0;    // ilk öğenin indisi (front)
    private int boyut = 0;  // öğe sayısı (size)

    @SuppressWarnings("unchecked")
    public DaireselKuyruk(int kapasite) {
        dizi = (E[]) new Object[kapasite];
    }
    public int boyut()      { return boyut; }
    public boolean bosMu()  { return boyut == 0; }
    public boolean doluMu() { return boyut == dizi.length; }
    public E ilk()          { return bosMu() ? null : dizi[bas]; }
    // ekle(), cikar()
}
```

Arka indisi ayrıca saklanmaz; gerektiğinde `(bas + boyut) % dizi.length` ile hesaplanır. Böylece iki alanın birbiriyle tutarsız kalma riski ortadan kalkar.

Dairesel Kuyruk: `ekle` ve `cikar`

```
public void ekle(E oge) {           // enqueue
    if (doluMu())
        throw new IllegalStateException("dolu");
    int son = (bas + boyut) % dizi.length;
    dizi[son] = oge;
    boyut++;
}
```

```
public E cikar() {                  // dequeue
    if (bosMu())
        throw new IllegalStateException("boş");
    E oge = dizi[bas];
    dizi[bas] = null;               // referansı bırak
    bas = (bas + 1) % dizi.length;
    boyut--;
    return oge;
}
```

Ekleme yeri

N = 5, bas = 3, boyut = 2

`son = (3 + 2) % 5 = 0`

Önü ilerletme

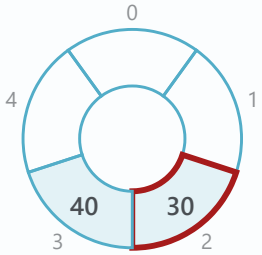
N = 5, bas = 4

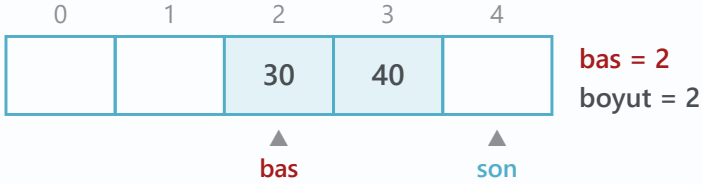
`bas = (4 + 1) % 5 = 0`

Maliyet

Her iki işlem de sabit sayıda adım: **O(1)**. Döngü veya kaydırma yok.

Kod İzleme: Dairesel `ekle()`





Başlangıç: dizi.length = 5, **bas = 2**, **boyut = 2** (30 ve 40). Şimdi 50, 60, 70, 80 eklenecek.

Adım 1 / 15

```
▶ public void ekle(E oge) {
2   if (doluMu())
3     throw new IllegalStateException("dolu");
4   int son = (bas + boyut) % dizi.length;
5   dizi[son] = oge;
6   boyut++;
7 }
```

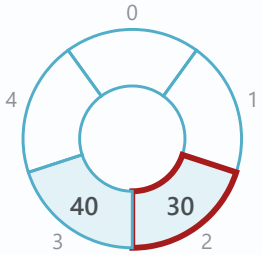
DEĞİŞKENLER

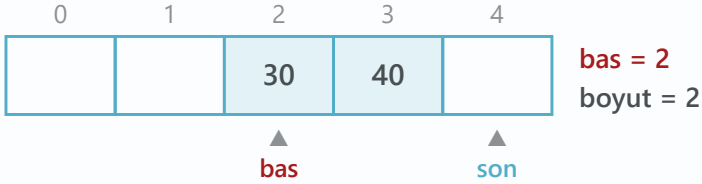
`oge = –` `bas = 2` `boyut = 2` `son = –` `dizi.length = 5`

ÇAĞRI YIĞINI

`main()`

Kod İzleme: Dairesel `ekle()`





bas = 2
boyut = 2

`ekle(50)` : `doluMu()` → boyut 2 == 5? **hayır**.

Adım 2 / 15

```
1 public void ekle(E oge) {
▶   if (doluMu())
3     throw new IllegalStateException("dolu");
4   int son = (bas + boyut) % dizi.length;
5   dizi[son] = oge;
6   boyut++;
7 }
```

DEĞİŞKENLER

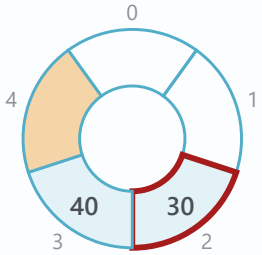
`oge = 50` `bas = 2` `boyut = 2` `son = –` `dizi.length = 5`

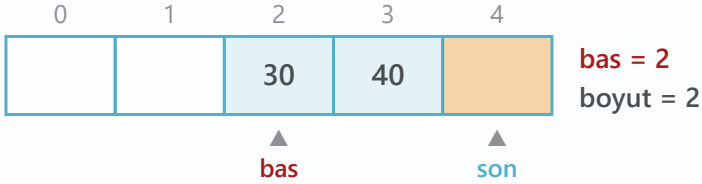
ÇAĞRI YIĞINI

`ekle(50)`

`main()`

Kod İzleme: Dairesel `ekle()`





son = (2 + 2) % 5 = 4 % 5 = 4.

Adım 3 / 15

```
1 public void ekle(E oge) {
2   if (doluMu())
3     throw new IllegalStateException("dolu");
▶   int son = (bas + boyut) % dizi.length;
5   dizi[son] = oge;
6   boyut++;
7 }
```

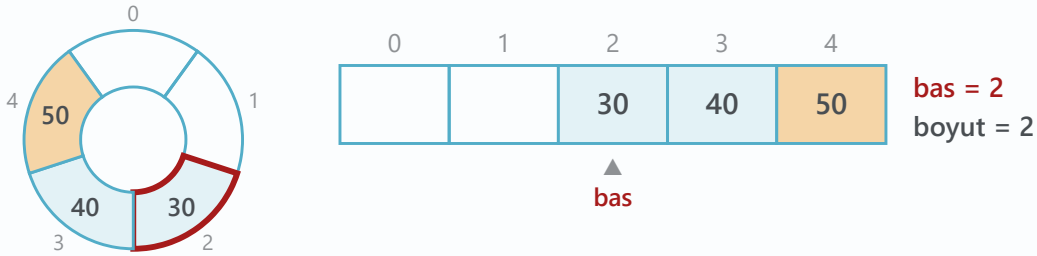
DEĞİŞKENLER

`oge = 50` `bas = 2` `boyut = 2` `son = 4` `dizi.length = 5`

ÇAĞRI YIĞINI

- `ekle(50)`
- `main()`

Kod İzleme: Dairesel `ekle()`



`dizi[4] = 50`. Diğer öğeler yerinden oynamaz.

Adım 4 / 15

```
1 public void ekle(E oge) {
2   if (doluMu())
3     throw new IllegalStateException("dolu");
4   int son = (bas + boyut) % dizi.length;
▶   dizi[son] = oge;
6   boyut++;
7 }
```

DEĞİŞKENLER

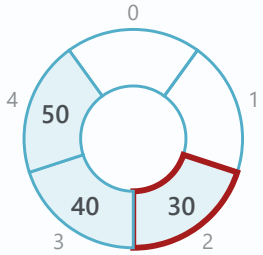
`oge = 50` `bas = 2` `boyut = 2` `son = 4` `dizi.length = 5`

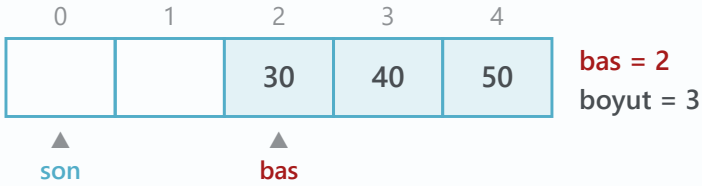
ÇAĞRI YIĞINI

`ekle(50)`

`main()`

Kod İzleme: Dairesel `ekle()`





`boyut++` → 3. bas değişmedi; ön hâlâ dizi[2].

Adım 5 / 15

```
1 public void ekle(E oge) {
2   if (doluMu())
3     throw new IllegalStateException("dolu");
4   int son = (bas + boyut) % dizi.length;
5   dizi[son] = oge;
▶   boyut++;
7 }
```

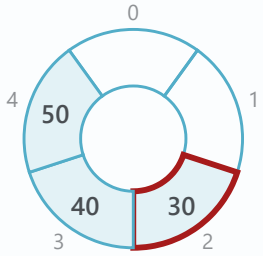
DEĞİŞKENLER

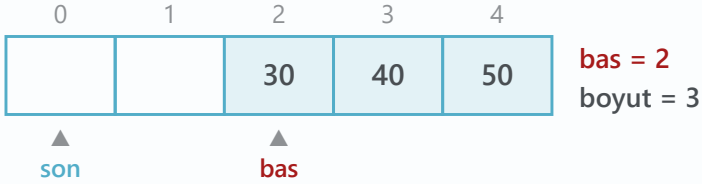
`oge = 50` `bas = 2` `boyut = 3` `son = 4` `dizi.length = 5`

ÇAĞRI YIĞINI

- `ekle(50)`
- `main()`

Kod İzleme: Dairesel `ekle()`





bas = 2
boyut = 3

`ekle(60)` : `doluMu()` → boyut 3 == 5? **hayır**.

Adım 6 / 15

```
1 public void ekle(E oge) {
▶   if (doluMu())
3     throw new IllegalStateException("dolu");
4   int son = (bas + boyut) % dizi.length;
5   dizi[son] = oge;
6   boyut++;
7 }
```

DEĞİŞKENLER

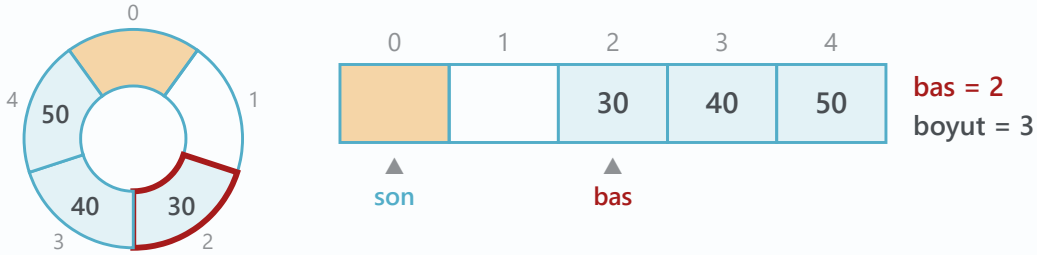
`oge = 60` `bas = 2` `boyut = 3` `son = –` `dizi.length = 5`

ÇAĞRI YIĞINI

`ekle(60)`

`main()`

Kod İzleme: Dairesel `ekle()`



son = (2 + 3) % 5 = 5 % 5 = 0 → dizinin sonu aşıldı, **başa sardı**.

Adım 7 / 15

```
1 public void ekle(E oge) {
2   if (doluMu())
3     throw new IllegalStateException("dolu");
▶   int son = (bas + boyut) % dizi.length;
5   dizi[son] = oge;
6   boyut++;
7 }
```

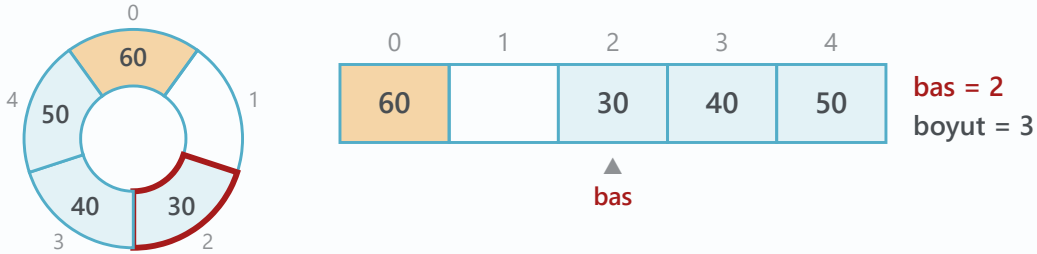
DEĞİŞKENLER

`oge = 60` `bas = 2` `boyut = 3` `son = 0` `dizi.length = 5`

ÇAĞRI YIĞINI

- `ekle(60)`
- `main()`

Kod İzleme: Dairesel `ekle()`



`dizi[0] = 60`. Diğer öğeler yerinden oynamaz.

Adım 8 / 15

```
1 public void ekle(E oge) {
2   if (doluMu())
3     throw new IllegalStateException("dolu");
4   int son = (bas + boyut) % dizi.length;
▶   dizi[son] = oge;
6   boyut++;
7 }
```

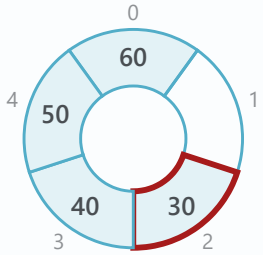
DEĞİŞKENLER

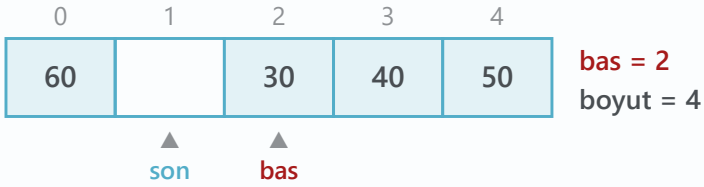
`oge = 60` `bas = 2` `boyut = 3` `son = 0` `dizi.length = 5`

ÇAĞRI YIĞINI

- **`ekle(60)`**
- `main()`

Kod İzleme: Dairesel `ekle()`





`boyut++` → 4. bas değişmedi; ön hâlâ dizi[2].

Adım 9 / 15

```
1 public void ekle(E oge) {
2   if (doluMu())
3     throw new IllegalStateException("dolu");
4   int son = (bas + boyut) % dizi.length;
5   dizi[son] = oge;
▶   boyut++;
7 }
```

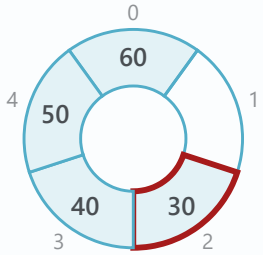
DEĞİŞKENLER

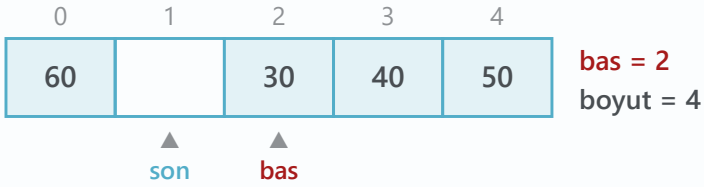
`oge = 60` `bas = 2` `boyut = 4` `son = 0` `dizi.length = 5`

ÇAĞRI YIĞINI

- `ekle(60)`
- `main()`

Kod İzleme: Dairesel `ekle()`





`ekle(70)` : `doluMu()` → boyut 4 == 5? **hayır**.

Adım 10 / 15

```
1 public void ekle(E oge) {
▶   if (doluMu())
3     throw new IllegalStateException("dolu");
4   int son = (bas + boyut) % dizi.length;
5   dizi[son] = oge;
6   boyut++;
7 }
```

DEĞİŞKENLER

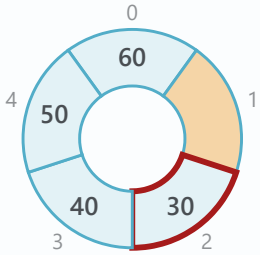
`oge = 70` `bas = 2` `boyut = 4` `son = –` `dizi.length = 5`

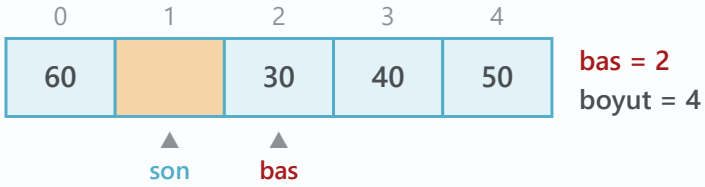
ÇAĞRI YIĞINI

`ekle(70)`

`main()`

Kod İzleme: Dairesel `ekle()`





son = (2 + 4) % 5 = 6 % 5 = 1 → dizinin sonu aşıldı, **başa sardı**.

Adım 11 / 15

```
1 public void ekle(E oge) {
2   if (doluMu())
3     throw new IllegalStateException("dolu");
▶   int son = (bas + boyut) % dizi.length;
5   dizi[son] = oge;
6   boyut++;
7 }
```

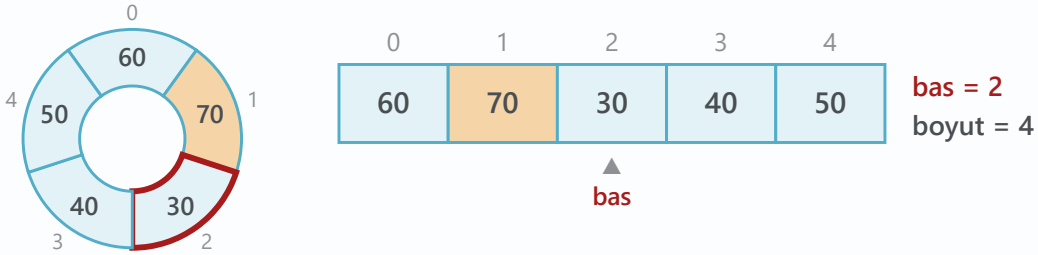
DEĞİŞKENLER

`oge = 70` `bas = 2` `boyut = 4` `son = 1` `dizi.length = 5`

ÇAĞRI YIĞINI

- `ekle(70)`
- `main()`

Kod İzleme: Dairesel `ekle()`



`dizi[1] = 70`. Diğer öğeler yerinden oynamaz.

Adım 12 / 15

```
1 public void ekle(E oge) {
2   if (doluMu())
3     throw new IllegalStateException("dolu");
4   int son = (bas + boyut) % dizi.length;
▶   dizi[son] = oge;
6   boyut++;
7 }
```

DEĞİŞKENLER

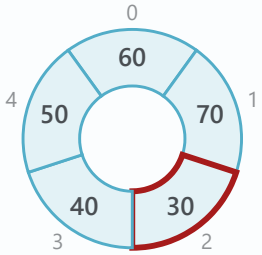
`oge = 70` `bas = 2` `boyut = 4` `son = 1` `dizi.length = 5`

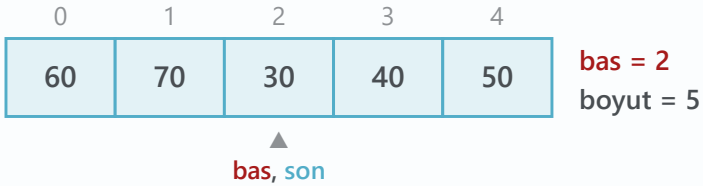
ÇAĞRI YIĞINI

`ekle(70)`

`main()`

Kod İzleme: Dairesel `ekle()`





`boyut++` → 5. bas değişmedi; ön hâlâ dizi[2]. Kuyruk artık dolu.

Adım 13 / 15

```
1 public void ekle(E oge) {
2   if (doluMu())
3     throw new IllegalStateException("dolu");
4   int son = (bas + boyut) % dizi.length;
5   dizi[son] = oge;
▶   boyut++;
7 }
```

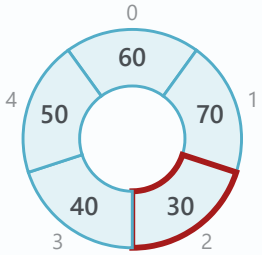
DEĞİŞKENLER

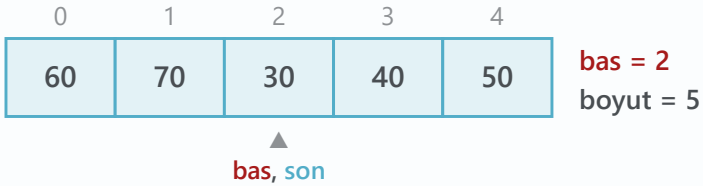
`oge = 70` `bas = 2` `boyut = 5` `son = 1` `dizi.length = 5`

ÇAĞRI YIĞINI

- `ekle(70)`
- `main()`

Kod İzleme: Dairesel `ekle()`





`ekle(80)` : `doluMu()` → boyut 5 == 5? **evet**.

Adım 14 / 15

```
1 public void ekle(E oge) {
▶   if (doluMu())
3     throw new IllegalStateException("dolu");
4   int son = (bas + boyut) % dizi.length;
5   dizi[son] = oge;
6   boyut++;
7 }
```

DEĞİŞKENLER

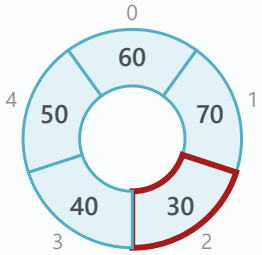
`oge = 80` `bas = 2` `boyut = 5` `son = –` `dizi.length = 5`

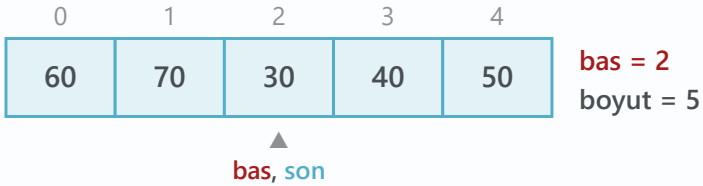
ÇAĞRI YIĞINI

`ekle(80)`

`main()`

Kod İzleme: Dairesel `ekle()`





Kuyruk dolu → `IllegalStateException("dolu")` fırlatılır; dizi değişmez. Kapasite 5 öğede tükendi, hiçbir yuva boşa gitmedi.

Adım 15 / 15

```
1 public void ekle(E oge) {
2   if (doluMu())
▶     throw new IllegalStateException("dolu");
4   int son = (bas + boyut) % dizi.length;
5   dizi[son] = oge;
6   boyut++;
7 }
```

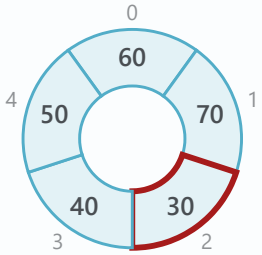
DEĞİŞKENLER

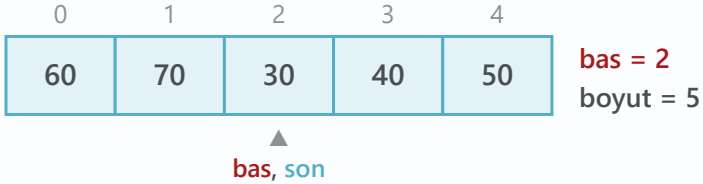
`oge = 80` `bas = 2` `boyut = 5`
`son = –` `dizi.length = 5`

ÇAĞRI YIĞINI

`ekle(80)`
`main()`

Kod İzleme: Dairesel `cikar()`





Başlangıç: dolu kuyruk, bas = 2. Sıra önden arkaya: 30, 40, 50, 60, 70. Üç kez `cikar()` çağrılacak.

Adım 1 / 16

```
▶ public E cikar() {
2   if (bosMu())
3     throw new IllegalStateException("boş");
4   E oge = dizi[bas];
5   dizi[bas] = null;            // referansı bırak
6   bas = (bas + 1) % dizi.length;
7   boyut--;
8   return oge;
9 }
```

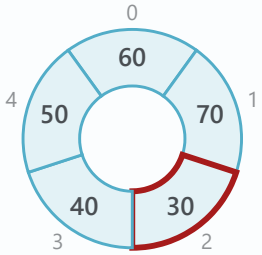
DEĞİŞKENLER

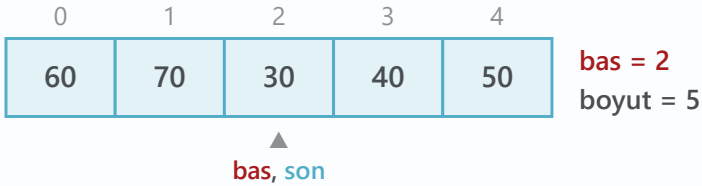
`oge = –` `bas = 2` `boyut = 5` `dizi.length = 5`

ÇAĞRI YIĞINI

`main()`

Kod İzleme: Dairesel `cikar()`





`bosMu()` : boyut = 5 ≠ 0 → boş değil, devam.

Adım 2 / 16

```
1 public E cikar() {
▶   if (bosMu())
3     throw new IllegalStateException("boş");
4   E oge = dizi[bas];
5   dizi[bas] = null;            // referansı bırak
6   bas = (bas + 1) % dizi.length;
7   boyut--;
8   return oge;
9 }
```

DEĞİŞKENLER

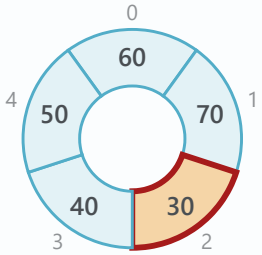
`oge = –` `bas = 2` `boyut = 5` `dizi.length = 5`

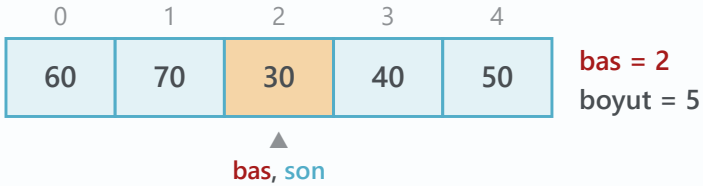
ÇAĞRI YIĞINI

`cikar()`

`main()`

Kod İzleme: Dairesel `cikar()`





oge = dizi[2] = 30: öndeki (en eski) öğe.

Adım 3 / 16

```
1 public E cikar() {
2   if (bosMu())
3     throw new IllegalStateException("boş");
▶   E oge = dizi[bas];
5   dizi[bas] = null;            // referansı bırak
6   bas = (bas + 1) % dizi.length;
7   boyut--;
8   return oge;
9 }
```

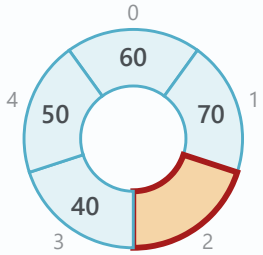
DEĞİŞKENLER

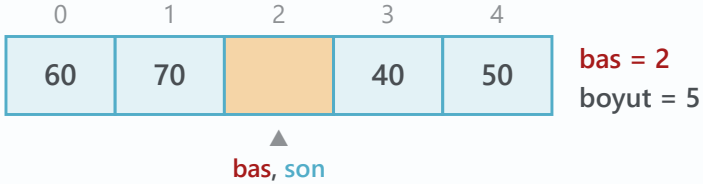
`oge = 30` `bas = 2` `boyut = 5` `dizi.length = 5`

ÇAĞRI YIĞINI

- `cikar()`
- `main()`

Kod İzleme: Dairesel `cikar()`





`dizi[2] = null` : yuva boşalır; referans bırakılınca nesne çöp toplayıcıya kalabilir.

Adım 4 / 16

```
1 public E cikar() {
2   if (bosMu())
3     throw new IllegalStateException("boş");
4   E oge = dizi[bas];
▶   dizi[bas] = null;            // referansı bırak
6   bas = (bas + 1) % dizi.length;
7   boyut--;
8   return oge;
9 }
```

DEĞİŞKENLER

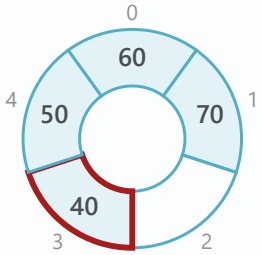
`oge = 30` `bas = 2` `boyut = 5` `dizi.length = 5`

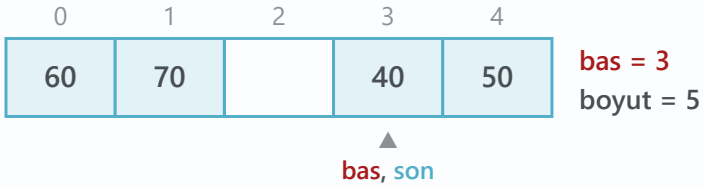
ÇAĞRI YIĞINI

`cikar()`

`main()`

Kod İzleme: Dairesel `cikar()`





bas = (2 + 1) % 5 = 3.

Adım 5 / 16

```
1 public E cikar() {
2   if (bosMu())
3     throw new IllegalStateException("boş");
4   E oge = dizi[bas];
5   dizi[bas] = null;            // referansı bırak
▶   bas = (bas + 1) % dizi.length;
7   boyut--;
8   return oge;
9 }
```

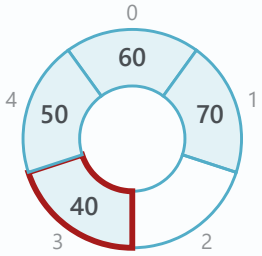
DEĞİŞKENLER

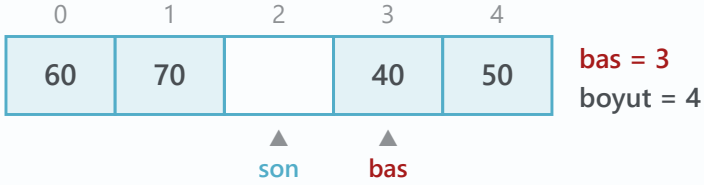
`oge = 30` `bas = 3` `boyut = 5` `dizi.length = 5`

ÇAĞRI YIĞINI

- `cikar()`
- `main()`

Kod İzleme: Dairesel `cikar()`





`boyut--` → 4; **return 30**.

Adım 6 / 16

```
1 public E cikar() {
2   if (bosMu())
3     throw new IllegalStateException("boş");
4   E oge = dizi[bas];
5   dizi[bas] = null;            // referansı bırak
6   bas = (bas + 1) % dizi.length;
▶   boyut--;
▶   return oge;
9 }
```

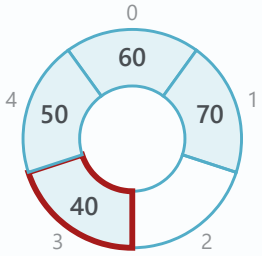
DEĞİŞKENLER

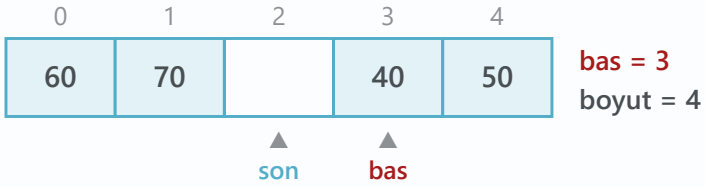
`oge = 30` `bas = 3` `boyut = 4` `dizi.length = 5`

ÇAĞRI YIĞINI

- `cikar()`
- `main()`

Kod İzleme: Dairesel `cikar()`





bas = 3
boyut = 4

`bosMu()` : boyut = 4 ≠ 0 → boş değil, devam.

Adım 7 / 16

```
1 public E cikar() {
▶   if (bosMu())
3     throw new IllegalStateException("boş");
4   E oge = dizi[bas];
5   dizi[bas] = null;            // referansı bırak
6   bas = (bas + 1) % dizi.length;
7   boyut--;
8   return oge;
9 }
```

DEĞİŞKENLER

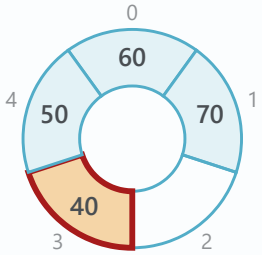
`oge = –` `bas = 3` `boyut = 4` `dizi.length = 5`

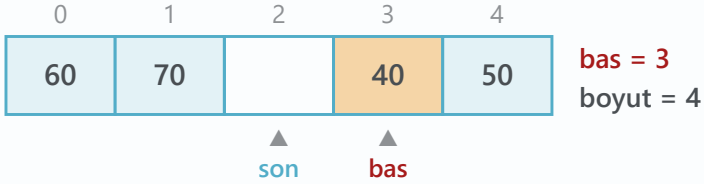
ÇAĞRI YIĞINI

`cikar()`

`main()`

Kod İzleme: Dairesel `cikar()`





oge = dizi[3] = 40: öndeki (en eski) öğe.

Adım 8 / 16

```
1 public E cikar() {
2   if (bosMu())
3     throw new IllegalStateException("boş");
▶   E oge = dizi[bas];
5   dizi[bas] = null;            // referansı bırak
6   bas = (bas + 1) % dizi.length;
7   boyut--;
8   return oge;
9 }
```

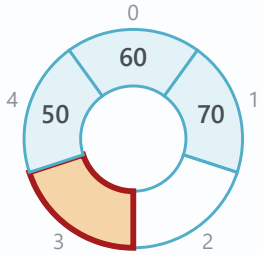
DEĞİŞKENLER

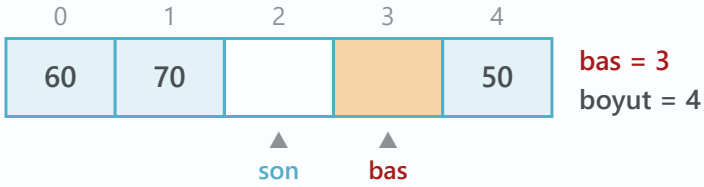
`oge = 40` `bas = 3` `boyut = 4` `dizi.length = 5`

ÇAĞRI YIĞINI

- `cikar()`
- `main()`

Kod İzleme: Dairesel `cikar()`





`dizi[3] = null` : yuva boşalır; referans bırakılınca nesne çöp toplayıcıya kalabilir.

Adım 9 / 16

```
1 public E cikar() {
2   if (bosMu())
3     throw new IllegalStateException("boş");
4   E oge = dizi[bas];
▶   dizi[bas] = null;            // referansı bırak
6   bas = (bas + 1) % dizi.length;
7   boyut--;
8   return oge;
9 }
```

DEĞİŞKENLER

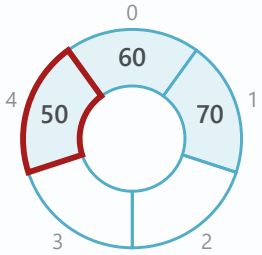
`oge = 40` `bas = 3` `boyut = 4` `dizi.length = 5`

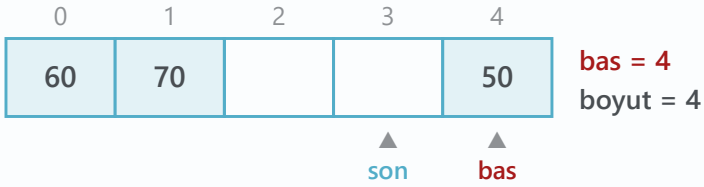
ÇAĞRI YIĞINI

`cikar()`

`main()`

Kod İzleme: Dairesel `cikar()`





bas = (3 + 1) % 5 = 4.

Adım 10 / 16

```
1 public E cikar() {
2   if (bosMu())
3     throw new IllegalStateException("boş");
4   E oge = dizi[bas];
5   dizi[bas] = null;            // referansı bırak
▶   bas = (bas + 1) % dizi.length;
7   boyut--;
8   return oge;
9 }
```

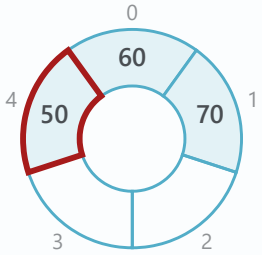
DEĞİŞKENLER

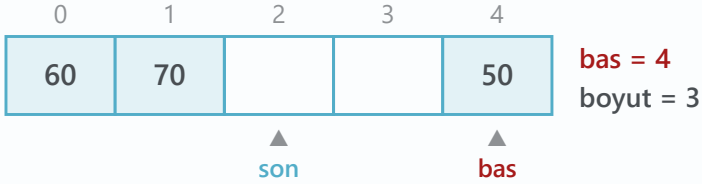
`oge = 40` `bas = 4` `boyut = 4` `dizi.length = 5`

ÇAĞRI YIĞINI

- `cikar()`
- `main()`

Kod İzleme: Dairesel `cikar()`





`boyut--` → 3; **return 40**.

Adım 11 / 16

```
1 public E cikar() {
2   if (bosMu())
3     throw new IllegalStateException("boş");
4   E oge = dizi[bas];
5   dizi[bas] = null;            // referansı bırak
6   bas = (bas + 1) % dizi.length;
▶   boyut--;
▶   return oge;
9 }
```

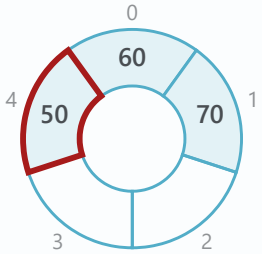
DEĞİŞKENLER

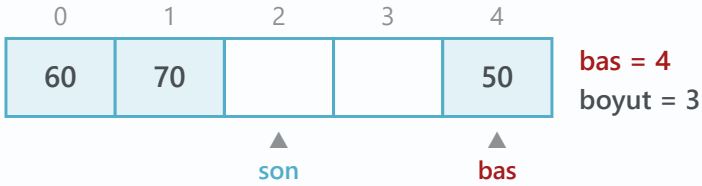
`oge = 40` `bas = 4` `boyut = 3` `dizi.length = 5`

ÇAĞRI YIĞINI

- `cikar()`
- `main()`

Kod İzleme: Dairesel `cikar()`





`bosMu()` : boyut = 3 ≠ 0 → boş değil, devam.

Adım 12 / 16

```
1 public E cikar() {
▶   if (bosMu())
3     throw new IllegalStateException("boş");
4   E oge = dizi[bas];
5   dizi[bas] = null;            // referansı bırak
6   bas = (bas + 1) % dizi.length;
7   boyut--;
8   return oge;
9 }
```

DEĞİŞKENLER

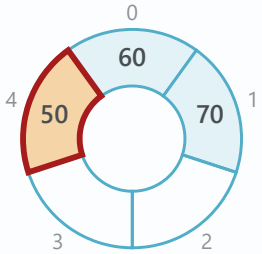
`oge = –` `bas = 4` `boyut = 3` `dizi.length = 5`

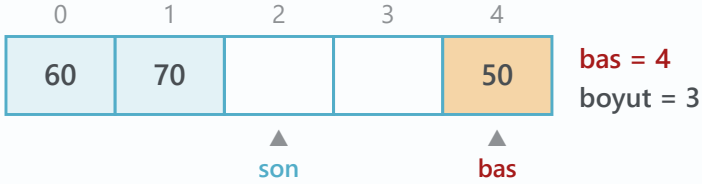
ÇAĞRI YIĞINI

`cikar()`

`main()`

Kod İzleme: Dairesel `cikar()`





oge = dizi[4] = 50: öndeki (en eski) öğe.

Adım 13 / 16

```
1 public E cikar() {
2   if (bosMu())
3     throw new IllegalStateException("boş");
▶   E oge = dizi[bas];
5   dizi[bas] = null;            // referansı bırak
6   bas = (bas + 1) % dizi.length;
7   boyut--;
8   return oge;
9 }
```

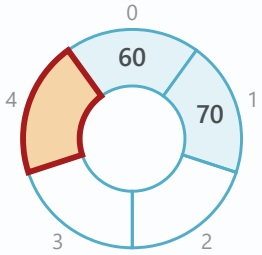
DEĞİŞKENLER

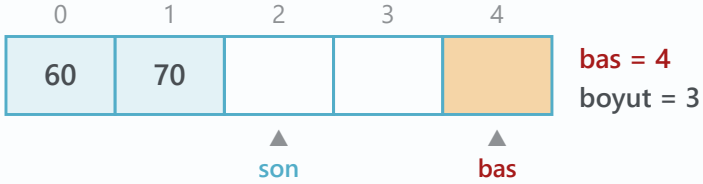
`oge = 50` `bas = 4` `boyut = 3` `dizi.length = 5`

ÇAĞRI YIĞINI

- `cikar()`
- `main()`

Kod İzleme: Dairesel `cikar()`





`dizi[4] = null` : yuva boşalır; referans bırakılınca nesne çöp toplayıcıya kalabilir.

Adım 14 / 16

```
1 public E cikar() {
2   if (bosMu())
3     throw new IllegalStateException("boş");
4   E oge = dizi[bas];
▶   dizi[bas] = null;            // referansı bırak
6   bas = (bas + 1) % dizi.length;
7   boyut--;
8   return oge;
9 }
```

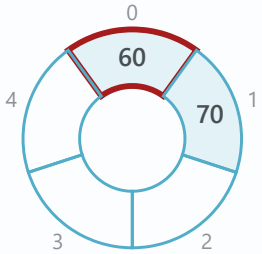
DEĞİŞKENLER

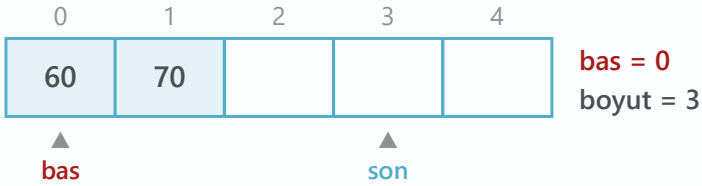
`oge = 50` `bas = 4` `boyut = 3` `dizi.length = 5`

ÇAĞRI YIĞINI

- **`cikar()`**
- `main()`

Kod İzleme: Dairesel `cikar()`





bas = (4 + 1) % 5 = 0 → 5 % 5 = 0, bas **başa sardı**.

Adım 15 / 16

```
1 public E cikar() {
2   if (bosMu())
3     throw new IllegalStateException("boş");
4   E oge = dizi[bas];
5   dizi[bas] = null;            // referansı bırak
▶   bas = (bas + 1) % dizi.length;
7   boyut--;
8   return oge;
9 }
```

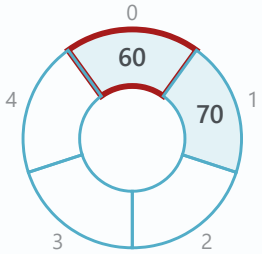
DEĞİŞKENLER

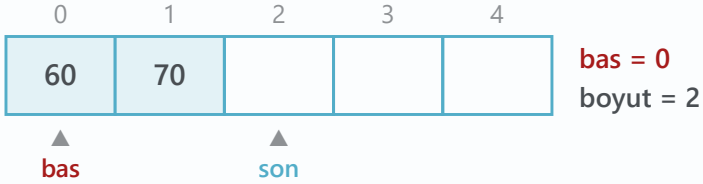
`oge = 50` `bas = 0` `boyut = 3` `dizi.length = 5`

ÇAĞRI YIĞINI

- `cikar()`
- `main()`

Kod İzleme: Dairesel `cikar()`





`boyut--` → 2; **return 50**. Sabit sayıda adım: **O(1)**.

Adım 16 / 16

```
1 public E cikar() {
2   if (bosMu())
3     throw new IllegalStateException("boş");
4   E oge = dizi[bas];
5   dizi[bas] = null;           // referansı bırak
6   bas = (bas + 1) % dizi.length;
    boyut--;
    return oge;
9 }
```

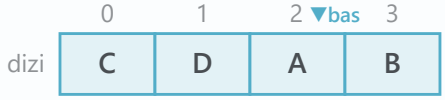
DEĞİŞKENLER

`oge = 50` `bas = 0` `boyut = 2` `dizi.length = 5`

ÇAĞRI YIĞINI

- **`cikar()`**
- `main()`

Kod İzleme: Dolu Dairesel Kuyruğu Büyütme



bas = 2 · boyut = 4 · N = 4

sıra: A B C D

Dolu kuyruk (bas = 2): sıra A, B, C, D, dizide ise **C, D, A, B**. Şimdi ekle(E).

Adım 1 / 11

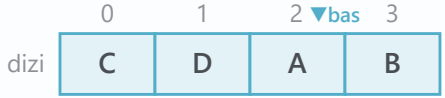
```
1  private void buyut(int yeniKap) {
2    E[] yeni = (E[]) new Object[yeniKap];
3    for (int k = 0; k < boyut; k++)
4      yeni[k] = dizi[(bas + k) % dizi.length];
5    dizi = yeni;
6    bas = 0;
7  }
▶ public void ekle(E oge) {
9    if (boyut == dizi.length) buyut(2 * dizi.length);
10   dizi[(bas + boyut) % dizi.length] = oge;
11   boyut++;
12 }
```

DEĞİŞKENLER

`oge = E` `bas = 2` `boyut = 4`

`dizi = [C, D, A, B]`

Kod İzleme: Dolu Dairesel Kuyruğu Büyütme



bas = 2 · boyut = 4 · N = 4

dolu!

4 == 4 → dolu. Hata yerine **iki katı** kapasiteyle büyüt.

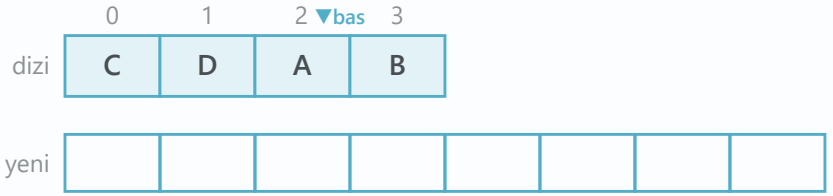
Adım 2 / 11

```
1  private void buyut(int yeniKap) {
2    E[] yeni = (E[]) new Object[yeniKap];
3    for (int k = 0; k < boyut; k++)
4      yeni[k] = dizi[(bas + k) % dizi.length];
5    dizi = yeni;
6    bas = 0;
7  }
8  public void ekle(E oge) {
▶    if (boyut == dizi.length) buyut(2 * dizi.length);
10   dizi[(bas + boyut) % dizi.length] = oge;
11   boyut++;
12 }
```

DEĞİŞKENLER

oge = **E** | bas = **2** | boyut = **4** | dizi = **[C, D, A, B]**

Kod İzleme: Dolu Dairesel Kuyruğu Büyütme



bas = 2 · boyut = 4 · N = 4

8 hücrelik yeni dizi ayrılır.

Adım 3 / 11

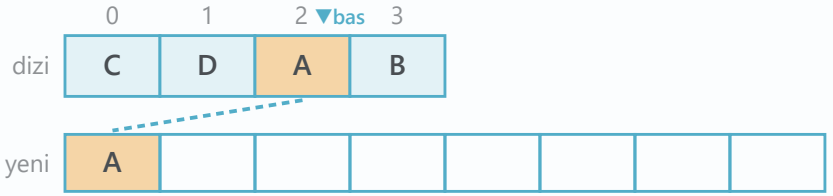
```
1  private void buyut(int yeniKap) {
▶    E[] yeni = (E[]) new Object[yeniKap];
3    for (int k = 0; k < boyut; k++)
4      yeni[k] = dizi[(bas + k) % dizi.length];
5    dizi = yeni;
6    bas = 0;
7  }
8  public void ekle(E oge) {
9    if (boyut == dizi.length) buyut(2 * dizi.length);
10   dizi[(bas + boyut) % dizi.length] = oge;
11   boyut++;
12 }
```

DEĞİŞKENLER

oge = **E** · bas = **2** · boyut = **4** · dizi = **[C, D, A, B]** · yeniKap = **8**

yeni = **[null, null, null, null, null, null, null, null]**

Kod İzleme: Dolu Dairesel Kuyruğu Büyütme



bas = 2 · boyut = 4 · N = 4

k = 0: yeni[0] = dizi[(2 + 0) % 4] = dizi[2] = **A**.


```
1  private void buyut(int yeniKap) {
2    E[] yeni = (E[]) new Object[yeniKap];
▶    for (int k = 0; k < boyut; k++)
▶      yeni[k] = dizi[(bas + k) % dizi.length];
5    dizi = yeni;
6    bas = 0;
7  }
8  public void ekle(E oge) {
9    if (boyut == dizi.length) buyut(2 * dizi.length);
10   dizi[(bas + boyut) % dizi.length] = oge;
11   boyut++;
12 }
```

DEĞİŞKENLER

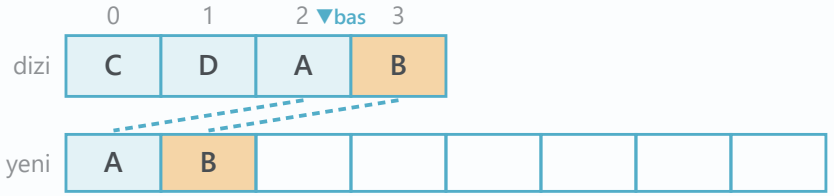
`oge = E` `bas = 2` `boyut = 4`

`dizi = [C, D, A, B]` `yeniKap = 8`

`k = 0`

`yeni = [A, null, null, null, null, null, null, null]`

Kod İzleme: Dolu Dairesel Kuyruğu Büyütme



bas = 2 · boyut = 4 · N = 4

k = 1: yeni[1] = dizi[(2 + 1) % 4] = dizi[3] = **B**.

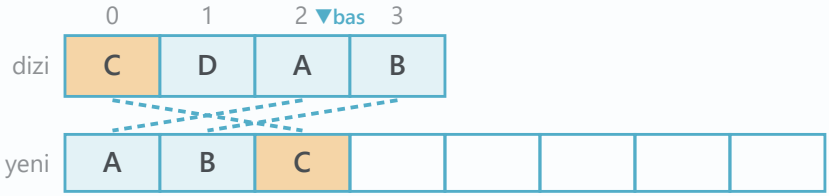

```
1  private void buyut(int yeniKap) {
2    E[] yeni = (E[]) new Object[yeniKap];
▶    for (int k = 0; k < boyut; k++)
▶      yeni[k] = dizi[(bas + k) % dizi.length];
5    dizi = yeni;
6    bas = 0;
7  }
8  public void ekle(E oge) {
9    if (boyut == dizi.length) buyut(2 * dizi.length);
10   dizi[(bas + boyut) % dizi.length] = oge;
11   boyut++;
12 }
```

DEĞİŞKENLER

oge = **E** · bas = **2** · boyut = **4** · dizi = **[C, D, A, B]** · yeniKap = **8** · k = **1**

yeni = **[A, B, null, null, null, null, null, null]**

Kod İzleme: Dolu Dairesel Kuyruğu Büyütme



bas = 2 · boyut = 4 · N = 4

k = 2: yeni[2] = dizi[(2 + 2) % 4] = dizi[0] = **C**. Mod ile başa sarıldı.

Adım 6 / 11

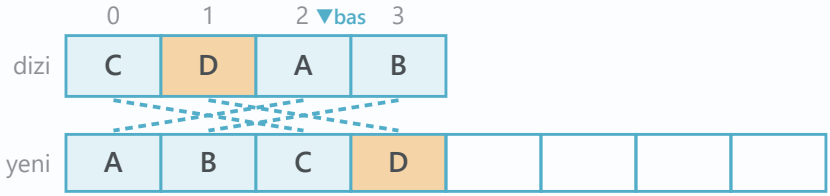
```
1  private void buyut(int yeniKap) {
2    E[] yeni = (E[]) new Object[yeniKap];
▶    for (int k = 0; k < boyut; k++)
▶      yeni[k] = dizi[(bas + k) % dizi.length];
5    dizi = yeni;
6    bas = 0;
7  }
8  public void ekle(E oge) {
9    if (boyut == dizi.length) buyut(2 * dizi.length);
10   dizi[(bas + boyut) % dizi.length] = oge;
11   boyut++;
12 }
```

DEĞİŞKENLER

oge = **E** · bas = **2** · boyut = **4** · dizi = **[C, D, A, B]** · yeniKap = **8** · k = **2**

yeni = **[A, B, C, null, null, null, null, null]**

Kod İzleme: Dolu Dairesel Kuyruğu Büyütme



bas = 2 · boyut = 4 · N = 4

k = 3: yeni[3] = dizi[(2 + 3) % 4] = dizi[1] = **D**.

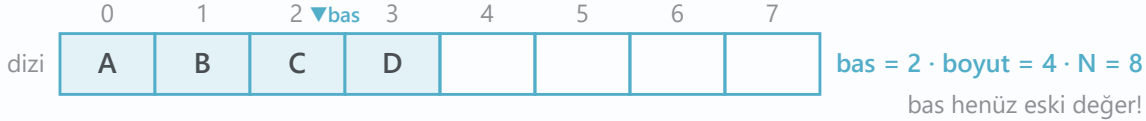

```
1  private void buyut(int yeniKap) {
2    E[] yeni = (E[]) new Object[yeniKap];
▶    for (int k = 0; k < boyut; k++)
▶      yeni[k] = dizi[(bas + k) % dizi.length];
5    dizi = yeni;
6    bas = 0;
7  }
8  public void ekle(E oge) {
9    if (boyut == dizi.length) buyut(2 * dizi.length);
10   dizi[(bas + boyut) % dizi.length] = oge;
11   boyut++;
12 }
```

DEĞİŞKENLER

oge = **E** · bas = **2** · boyut = **4** · dizi = **[C, D, A, B]** · yeniKap = **8** · k = **3**

yeni = **[A, B, C, D, null, null, null, null]**

Kod İzleme: Dolu Dairesel Kuyruğu Büyütme



dizi = yeni: öğeler artık **sıralı** (Arrays.copyOf sırayı bozardı).

Adım 8 / 11

```
1  private void buyut(int yeniKap) {
2    E[] yeni = (E[]) new Object[yeniKap];
3    for (int k = 0; k < boyut; k++)
4      yeni[k] = dizi[(bas + k) % dizi.length];
▶    dizi = yeni;
6    bas = 0;
7  }
8  public void ekle(E oge) {
9    if (boyut == dizi.length) buyut(2 * dizi.length);
10   dizi[(bas + boyut) % dizi.length] = oge;
11   boyut++;
12 }
```

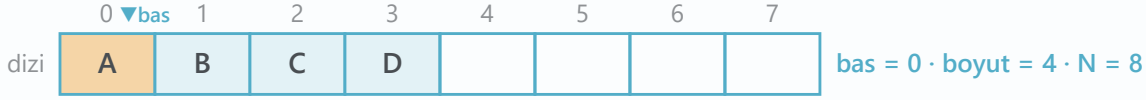
DEĞİŞKENLER

oge = **E** | bas = **2** | boyut = **4**

dizi = **[A, B, C, D, null, null, null, null]**

yeniKap = **8**

Kod İzleme: Dolu Dairesel Kuyruğu Büyütme



`bas = 0` : ön öğe (A) artık dizi[0]'da.

Adım 9 / 11

```
1  private void buyut(int yeniKap) {
2    E[] yeni = (E[]) new Object[yeniKap];
3    for (int k = 0; k < boyut; k++)
4      yeni[k] = dizi[(bas + k) % dizi.length];
5    dizi = yeni;
▶    bas = 0;
7  }
8  public void ekle(E oge) {
9    if (boyut == dizi.length) buyut(2 * dizi.length);
10   dizi[(bas + boyut) % dizi.length] = oge;
11   boyut++;
12 }
```

DEĞİŞKENLER

- `oge = E`
- `bas = 0`
- `boyut = 4`
- `dizi = [A, B, C, D, null, null, null, null]`
- `yeniKap = 8`

Kod İzleme: Dolu Dairesel Kuyruğu Büyütme



ekle'ye dönülür: (0 + 4) % 8 = **4** → dizi[4] = E.

Adım 10 / 11

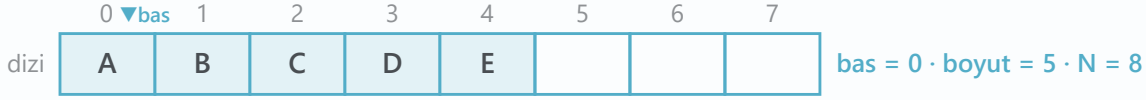
```
1  private void buyut(int yeniKap) {
2    E[] yeni = (E[]) new Object[yeniKap];
3    for (int k = 0; k < boyut; k++)
4      yeni[k] = dizi[(bas + k) % dizi.length];
5    dizi = yeni;
6    bas = 0;
7  }
8  public void ekle(E oge) {
9    if (boyut == dizi.length) buyut(2 * dizi.length);
▶    dizi[(bas + boyut) % dizi.length] = oge;
11   boyut++;
12 }
```

DEĞİŞKENLER

`oge = E` `bas = 0` `boyut = 4`

`dizi = [A, B, C, D, E, null, null, null]`

Kod İzleme: Dolu Dairesel Kuyruğu Büyütme



boyut → 5. Büyütme O(n), ama 2 katına çıktığı için ekle **amortize O(1)**.

Adım 11 / 11

```
1  private void buyut(int yeniKap) {
2    E[] yeni = (E[]) new Object[yeniKap];
3    for (int k = 0; k < boyut; k++)
4      yeni[k] = dizi[(bas + k) % dizi.length];
5    dizi = yeni;
6    bas = 0;
7  }
8  public void ekle(E oge) {
9    if (boyut == dizi.length) buyut(2 * dizi.length);
10   dizi[(bas + boyut) % dizi.length] = oge;
▶    boyut++;
12 }
```

DEĞİŞKENLER

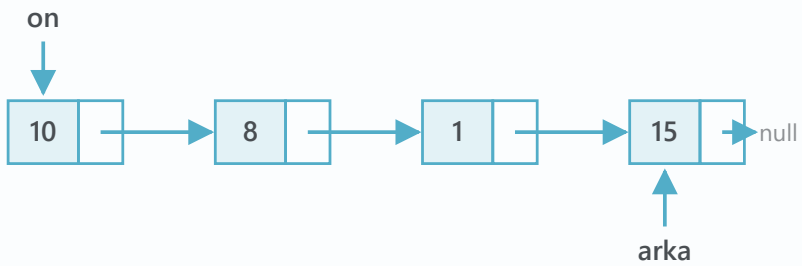
`oge = E` `bas = 0` `boyut = 5`

`dizi = [A, B, C, D, E, null, null, null]`

Bağlı Liste ile Kuyruk

- Her kuyruk öğesi bir **düğümdür** (veri + sonraki referansı).
- `on` listenin başını, `arka` sonunu gösterir: ekleme sona, çıkarma baştan.
- Boyut dinamiktir; kapasite sınırı (dolu kuyruk) yoktur.
- Düğüm başına fazladan bir referans ve nesne oluşturma maliyeti vardır.

```
class Dugum {
    int veri;
    Dugum sonraki;
    Dugum(int veri) {
        this.veri = veri;
    }
}
// alanlar:
Dugum on, arka;
int uzunluk;
```



Neden ekleme sona, çıkarma başa? Tek yönlü listede baştan silmek O(1), ama sondan silmek için sondan bir önceki düğümü bulmak gerekir ve bu O(n) sürer.

Enqueue ve Dequeue Algoritması

enqueue(x)

1. Kuyruk dolu mu? (sabit kapasiteli dizide)
2. Doluysa **taşma** (overflow) hatası ver, bitir.
3. Değilse x'i arka konuma yaz.
4. Arkayı bir sonraki konuma ilerlet.
5. Öğe sayısını artır, başarılı dön.

dequeue()

1. Kuyruk boş mu?
2. Boşsa **alt taşma** (underflow) hatası ver, bitir.
3. Değilse öndeki öğeyi al.
4. Önü bir sonraki öğeye ilerlet.
5. Öğe sayısını azalt, alınan öğeyi döndür.

Adım	Dairesel dizi	Bağlı liste
Arkaya yaz	`dizi[(bas + boyut) % N] = x`	`arka.sonraki = yeni; arka = yeni`
Önü ilerlet	`bas = (bas + 1) % N`	`on = on.sonraki`
Dolu kontrolü	`boyut == N`	gerekmez (bellek yettikçe)

Kod İzleme: Bağlı Liste Kuyruğu `enqueue`

on = null · arka = null · uzunluk = 0

Boş kuyruk: on = arka = null. Sırayla `enqueue(10)`, `enqueue(15)`, `enqueue(20)`.

Adım 1 / 16

```
▶ public void enqueue(int veri) {
2     Dugum gecici = new Dugum(veri);
3     if (bosMu()) {
4         on = gecici;
5     } else {
6         arka.sonraki = gecici;
7     }
8     arka = gecici;
9     uzunluk++;
10 }
```

DEĞİŞKENLER

`veri = –` `gecici = –` `on = null` `arka = null` `uzunluk = 0`

ÇAĞRI YIĞINI

`main()`

Kod İzleme: Bağlı Liste Kuyruğu `enqueue`



on = null · arka = null · uzunluk = 0

`new Dugum(10)` : sonraki = null olan yeni düğüm; henüz listeye bağlı değil.

Adım 2 / 16

```
1  public void enqueue(int veri) {
▶       Dugum gecici = new Dugum(veri);
3       if (bosMu()) {
4           on = gecici;
5       } else {
6           arka.sonraki = gecici;
7       }
8       arka = gecici;
9       uzunluk++;
10 }
```

DEĞİŞKENLER

`veri = 10` `gecici = düğüm(10)`

`on = null` `arka = null` `uzunluk = 0`

ÇAĞRI YIĞINI

`enqueue(10)`

`main()`

Kod İzleme: Bağlı Liste Kuyruğu `enqueue`



on = null · arka = null · uzunluk = 0

`bosMu()` → **true** (uzunluk 0).

Adım 3 / 16

```
1  public void enqueue(int veri) {
2      Dugum gecici = new Dugum(veri);
▶      if (bosMu()) {
4          on = gecici;
5      } else {
6          arka.sonraki = gecici;
7      }
8      arka = gecici;
9      uzunluk++;
10 }
```

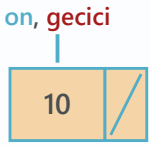
DEĞİŞKENLER

`veri = 10` `gecici = düğüm(10)`
`on = null` `arka = null` `uzunluk = 0`

ÇAĞRI YIĞINI

`enqueue(10)`
`main()`

Kod İzleme: Bağlı Liste Kuyruğu `enqueue`



arka = null · uzunluk = 0

İlk düğüm: `on = gecici` .

Adım 4 / 16

```
1  public void enqueue(int veri) {
2      Dugum gecici = new Dugum(veri);
3      if (bosMu()) {
▶          on = gecici;
5      } else {
6          arka.sonraki = gecici;
7      }
8      arka = gecici;
9      uzunluk++;
10 }
```

DEĞİŞKENLER

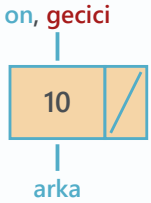
`veri = 10` `gecici = düğüm(10)` `on = düğüm(10)` `arka = null` `uzunluk = 0`

ÇAĞRI YIĞINI

`enqueue(10)`

`main()`

Kod İzleme: Bağlı Liste Kuyruğu `enqueue`



uzunluk = 0

`arka = gecici` : arka artık yeni düğüm. Tek düğüm hem ön hem arka.

Adım 5 / 16

```
1  public void enqueue(int veri) {
2      Dugum gecici = new Dugum(veri);
3      if (bosMu()) {
4          on = gecici;
5      } else {
6          arka.sonraki = gecici;
7      }
▶      arka = gecici;
9      uzunluk++;
10 }
```

DEĞİŞKENLER

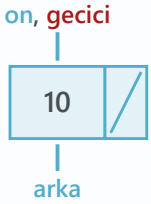
`veri = 10` `gecici = düğüm(10)` `on = düğüm(10)` `arka = düğüm(10)` `uzunluk = 0`

ÇAĞRI YIĞINI

`enqueue(10)`

`main()`

Kod İzleme: Bağlı Liste Kuyruğu `enqueue`



uzunluk = 1

`uzunluk++` → 1.

Adım 6 / 16

```
1  public void enqueue(int veri) {
2      Dugum gecici = new Dugum(veri);
3      if (bosMu()) {
4          on = gecici;
5      } else {
6          arka.sonraki = gecici;
7      }
8      arka = gecici;
▶      uzunluk++;
10 }
```

DEĞİŞKENLER

`veri = 10` `gecici = düğüm(10)`

`on = düğüm(10)` `arka = düğüm(10)`

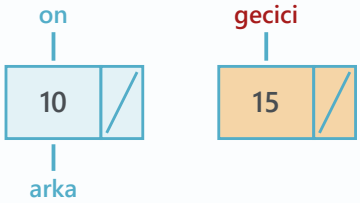
`uzunluk = 1`

ÇAĞRI YIĞINI

`enqueue(10)`

`main()`

Kod İzleme: Bağlı Liste Kuyruğu `enqueue`



uzunluk = 1

`new Dugum(15)` : sonraki = null olan yeni düğüm; henüz listeye bağlı değil.

Adım 7 / 16

```
1  public void enqueue(int veri) {
▶       Dugum gecici = new Dugum(veri);
3       if (bosMu()) {
4           on = gecici;
5       } else {
6           arka.sonraki = gecici;
7       }
8       arka = gecici;
9       uzunluk++;
10 }
```

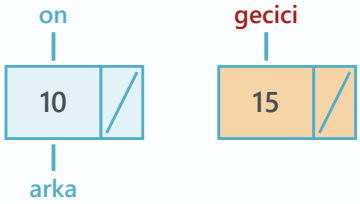
DEĞİŞKENLER

`veri = 15` `gecici = düğüm(15)`
`on = düğüm(10)` `arka = düğüm(10)`
`uzunluk = 1`

ÇAĞRI YIĞINI

`enqueue(15)`
`main()`

Kod İzleme: Bağlı Liste Kuyruğu `enqueue`



uzunluk = 1

`bosMu()` → **false**; else dalına gidilir.

Adım 8 / 16

```
1  public void enqueue(int veri) {
2      Dugum gecici = new Dugum(veri);
▶      if (bosMu()) {
4          on = gecici;
5      } else {
6          arka.sonraki = gecici;
7      }
8      arka = gecici;
9      uzunluk++;
10 }
```

DEĞİŞKENLER

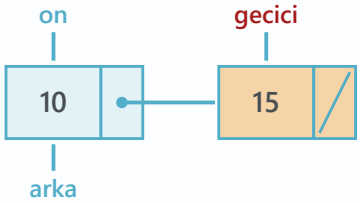
`veri = 15` `gecici = düğüm(15)` `on = düğüm(10)` `arka = düğüm(10)` `uzunluk = 1`

ÇAĞRI YIĞINI

`enqueue(15)`

`main()`

Kod İzleme: Bağlı Liste Kuyruğu `enqueue`



uzunluk = 1

`arka.sonraki = gecici` : eski arka (10) yeni düğüme bağlanır.

Adım 9 / 16

```
1  public void enqueue(int veri) {
2      Dugum gecici = new Dugum(veri);
3      if (bosMu()) {
4          on = gecici;
5      } else {
▶          arka.sonraki = gecici;
7      }
8      arka = gecici;
9      uzunluk++;
10 }
```

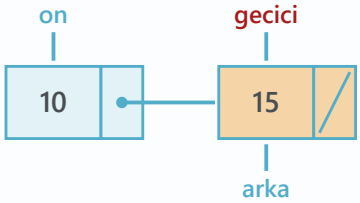
DEĞİŞKENLER

`veri = 15` `gecici = düğüm(15)`
`on = düğüm(10)` `arka = düğüm(10)`
`uzunluk = 1`

ÇAĞRI YIĞINI

`enqueue(15)`
`main()`

Kod İzleme: Bağlı Liste Kuyruğu `enqueue`



uzunluk = 1

`arka = gecici` : arka artık yeni düğüm.

Adım 10 / 16

```
1  public void enqueue(int veri) {
2      Dugum gecici = new Dugum(veri);
3      if (bosMu()) {
4          on = gecici;
5      } else {
6          arka.sonraki = gecici;
7      }
▶      arka = gecici;
9      uzunluk++;
10 }
```

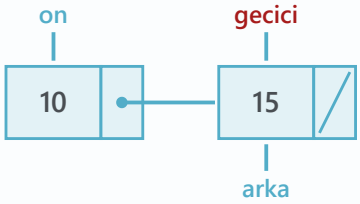
DEĞİŞKENLER

`veri = 15` `gecici = düğüm(15)`
`on = düğüm(10)` `arka = düğüm(15)`
`uzunluk = 1`

ÇAĞRI YIĞINI

`enqueue(15)`
`main()`

Kod İzleme: Bağlı Liste Kuyruğu `enqueue`



uzunluk = 2

`uzunluk++` → 2.

Adım 11 / 16

```
1  public void enqueue(int veri) {
2      Dugum gecici = new Dugum(veri);
3      if (bosMu()) {
4          on = gecici;
5      } else {
6          arka.sonraki = gecici;
7      }
8      arka = gecici;
▶      uzunluk++;
10 }
```

DEĞİŞKENLER

`veri = 15` `gecici = düğüm(15)`

`on = düğüm(10)` `arka = düğüm(15)`

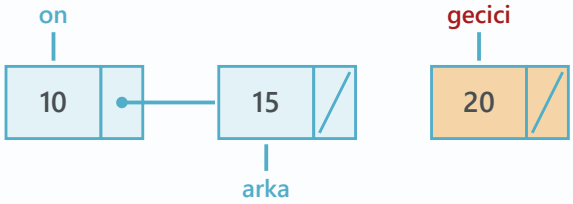
`uzunluk = 2`

ÇAĞRI YIĞINI

`enqueue(15)`

`main()`

Kod İzleme: Bağlı Liste Kuyruğu `enqueue`



uzunluk = 2

`new Dugum(20)` : sonraki = null olan yeni düğüm; henüz listeye bağlı değil.

Adım 12 / 16

```
1  public void enqueue(int veri) {
▶       Dugum gecici = new Dugum(veri);
3       if (bosMu()) {
4           on = gecici;
5       } else {
6           arka.sonraki = gecici;
7       }
8       arka = gecici;
9       uzunluk++;
10 }
```

DEĞİŞKENLER

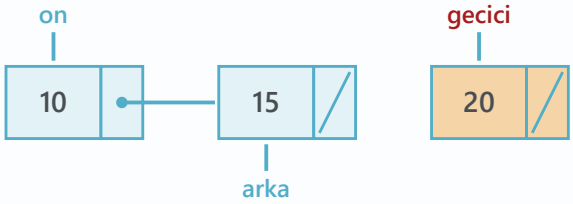
`veri = 20` `gecici = düğüm(20)` `on = düğüm(10)` `arka = düğüm(15)` `uzunluk = 2`

ÇAĞRI YIĞINI

`enqueue(20)`

`main()`

Kod İzleme: Bağlı Liste Kuyruğu `enqueue`



uzunluk = 2

`bosMu()` → **false**; else dalına gidilir.

Adım 13 / 16

```
1  public void enqueue(int veri) {
2      Dugum gecici = new Dugum(veri);
▶      if (bosMu()) {
4          on = gecici;
5      } else {
6          arka.sonraki = gecici;
7      }
8      arka = gecici;
9      uzunluk++;
10 }
```

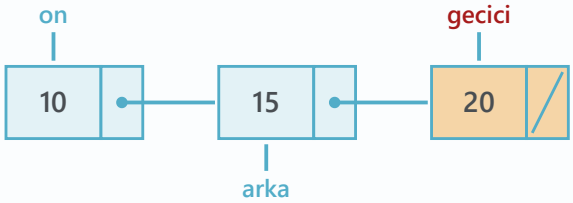
DEĞİŞKENLER

`veri = 20` `gecici = düğüm(20)`
`on = düğüm(10)` `arka = düğüm(15)`
`uzunluk = 2`

ÇAĞRI YIĞINI

`enqueue(20)`
`main()`

Kod İzleme: Bağlı Liste Kuyruğu `enqueue`



uzunluk = 2

`arka.sonraki = gecici` : eski arka (15) yeni düğüme bağlanır.

Adım 14 / 16

```
1  public void enqueue(int veri) {
2      Dugum gecici = new Dugum(veri);
3      if (bosMu()) {
4          on = gecici;
5      } else {
▶          arka.sonraki = gecici;
7      }
8      arka = gecici;
9      uzunluk++;
10 }
```

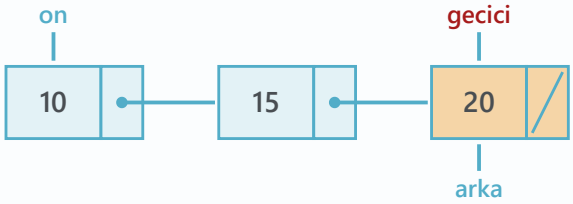
DEĞİŞKENLER

`veri = 20` `gecici = düğüm(20)`
`on = düğüm(10)` `arka = düğüm(15)`
`uzunluk = 2`

ÇAĞRI YIĞINI

`enqueue(20)`
`main()`

Kod İzleme: Bağlı Liste Kuyruğu `enqueue`



uzunluk = 2

`arka = gecici` : arka artık yeni düğüm.

Adım 15 / 16

```
1  public void enqueue(int veri) {
2      Dugum gecici = new Dugum(veri);
3      if (bosMu()) {
4          on = gecici;
5      } else {
6          arka.sonraki = gecici;
7      }
▶      arka = gecici;
9      uzunluk++;
10 }
```

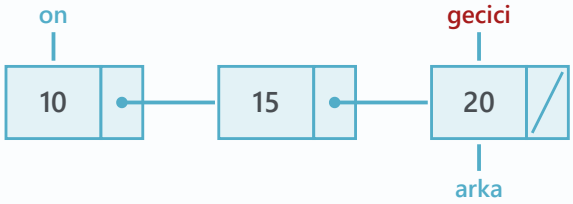
DEĞİŞKENLER

`veri = 20` `gecici = düğüm(20)`
`on = düğüm(10)` `arka = düğüm(20)`
`uzunluk = 2`

ÇAĞRI YIĞINI

`enqueue(20)`
`main()`

Kod İzleme: Bağlı Liste Kuyruğu `enqueue`



uzunluk = 3

`uzunluk++` → 3. Liste hiç gezinilmedi: her enqueue **O(1)**.

Adım 16 / 16

```
1  public void enqueue(int veri) {
2      Dugum gecici = new Dugum(veri);
3      if (bosMu()) {
4          on = gecici;
5      } else {
6          arka.sonraki = gecici;
7      }
8      arka = gecici;
▶      uzunluk++;
10 }
```

DEĞİŞKENLER

`veri = 20` `gecici = düğüm(20)`

`on = düğüm(10)` `arka = düğüm(20)`

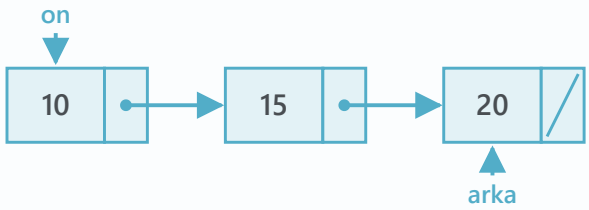
`uzunluk = 3`

ÇAĞRI YIĞINI

`enqueue(20)`

`main()`

Kod İzleme: Bağlı Liste Kuyruğu `dequeue`



uzunluk = 3

Kuyruk: 10, 15, 20. Üç kez `dequeue()` çağrılacak.

Adım 1 / 17

```
▶ public int dequeue() {
2     if (bosMu())
3         throw new NoSuchElementException();
4     int sonuc = on.veri;
5     on = on.sonraki;
6     if (on == null)
7         arka = null;
8     uzunluk--;
9     return sonuc;
10 }
```

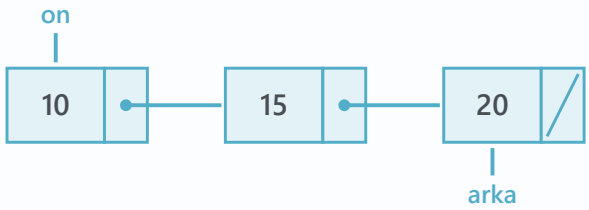
DEĞİŞKENLER

`sonuc = –` `on = düğüm(10)` `arka = düğüm(20)` `uzunluk = 3`

ÇAĞRI YIĞINI

`main()`

Kod İzleme: Bağlı Liste Kuyruğu `dequeue`



uzunluk = 3

`bosMu()` → false (uzunluk 3).

Adım 2 / 17

```
1  public int dequeue() {
▶      if (bosMu())
3          throw new NoSuchElementException();
4      int sonuc = on.veri;
5      on = on.sonraki;
6      if (on == null)
7          arka = null;
8      uzunluk--;
9      return sonuc;
10 }
```

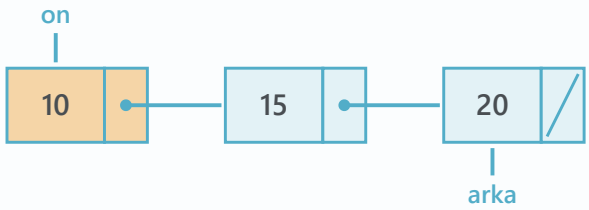
DEĞİŞKENLER

`sonuc = –` `on = düğüm(10)` `arka = düğüm(20)` `uzunluk = 3`

ÇAĞRI YIĞINI

- **dequeue()**
- main()

Kod İzleme: Bağlı Liste Kuyruğu `dequeue`



uzunluk = 3

sonuc = on.veri = 10.

Adım 3 / 17

```
1  public int dequeue() {
2      if (bosMu())
3          throw new NoSuchElementException();
▶      int sonuc = on.veri;
5      on = on.sonraki;
6      if (on == null)
7          arka = null;
8      uzunluk--;
9      return sonuc;
10 }
```

DEĞİŞKENLER

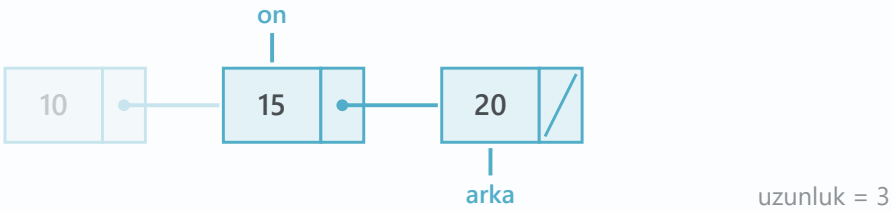
`sonuc = 10` `on = düğüm(10)` `arka = düğüm(20)` `uzunluk = 3`

ÇAĞRI YIĞINI

`dequeue()`

`main()`

Kod İzleme: Bağlı Liste Kuyruğu `dequeue`



`on = on.sonraki` → düğüm(15). Eski düğüme artık kimse bakmıyor (çöp toplayıcı temizler).

Adım 4 / 17

```
1  public int dequeue() {
2      if (bosMu())
3          throw new NoSuchElementException();
4      int sonuc = on.veri;
▶      on = on.sonraki;
6      if (on == null)
7          arka = null;
8      uzunluk--;
9      return sonuc;
10 }
```

DEĞİŞKENLER

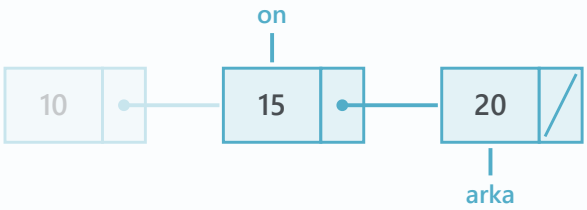
`sonuc = 10` `on = düğüm(15)` `arka = düğüm(20)` `uzunluk = 3`

ÇAĞRI YIĞINI

`dequeue()`

`main()`

Kod İzleme: Bağlı Liste Kuyruğu `dequeue`



uzunluk = 3

`on == null` → false; arka aynı kalır.

Adım 5 / 17

```
1  public int dequeue() {
2      if (bosMu())
3          throw new NoSuchElementException();
4      int sonuc = on.veri;
5      on = on.sonraki;
▶      if (on == null)
7          arka = null;
8      uzunluk--;
9      return sonuc;
10 }
```

DEĞİŞKENLER

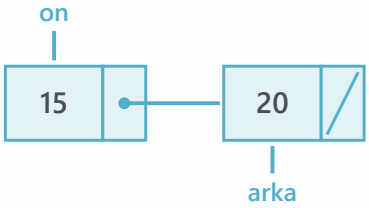
`sonuc = 10` `on = düğüm(15)` `arka = düğüm(20)` `uzunluk = 3`

ÇAĞRI YIĞINI

`dequeue()`

`main()`

Kod İzleme: Bağlı Liste Kuyruğu `dequeue`



uzunluk = 2

`uzunluk--` → 2; **return 10**.

Adım 6 / 17

```
1  public int dequeue() {
2      if (bosMu())
3          throw new NoSuchElementException();
4      int sonuc = on.veri;
5      on = on.sonraki;
6      if (on == null)
7          arka = null;
▶      uzunluk--;
▶      return sonuc;
10 }
```

DEĞİŞKENLER

`sonuc = 10` `on = düğüm(15)`

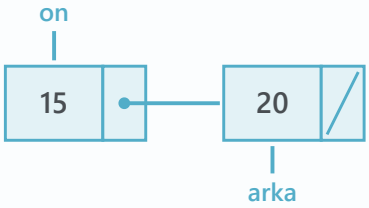
`arka = düğüm(20)` `uzunluk = 2`

ÇAĞRI YIĞINI

dequeue()

main()

Kod İzleme: Bağlı Liste Kuyruğu `dequeue`



uzunluk = 2

`bosMu()` → false (uzunluk 2).

Adım 7 / 17

```
1  public int dequeue() {
▶       if (bosMu())
3           throw new NoSuchElementException();
4       int sonuc = on.veri;
5       on = on.sonraki;
6       if (on == null)
7           arka = null;
8       uzunluk--;
9       return sonuc;
10 }
```

DEĞİŞKENLER

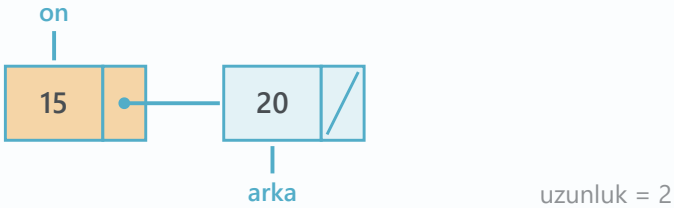
`sonuc = –` `on = düğüm(15)` `arka = düğüm(20)` `uzunluk = 2`

ÇAĞRI YIĞINI

`dequeue()`

`main()`

Kod İzleme: Bağlı Liste Kuyruğu `dequeue`



sonuc = on.veri = 15.

Adım 8 / 17

```
1  public int dequeue() {
2      if (bosMu())
3          throw new NoSuchElementException();
▶      int sonuc = on.veri;
5      on = on.sonraki;
6      if (on == null)
7          arka = null;
8      uzunluk--;
9      return sonuc;
10 }
```

DEĞİŞKENLER

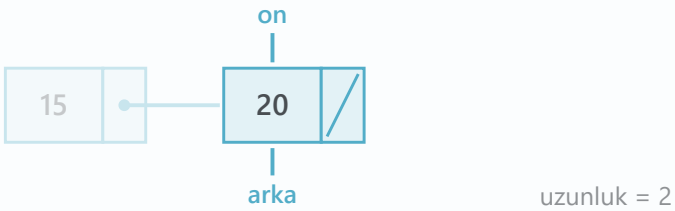
`sonuc = 15` `on = düğüm(15)` `arka = düğüm(20)` `uzunluk = 2`

ÇAĞRI YIĞINI

`dequeue()`

`main()`

Kod İzleme: Bağlı Liste Kuyruğu `dequeue`



`on = on.sonraki` → düğüm(20). Eski düğüme artık kimse bakmıyor (çöp toplayıcı temizler).

Adım 9 / 17

```
1  public int dequeue() {
2      if (bosMu())
3          throw new NoSuchElementException();
4      int sonuc = on.veri;
▶      on = on.sonraki;
6      if (on == null)
7          arka = null;
8      uzunluk--;
9      return sonuc;
10 }
```

DEĞİŞKENLER

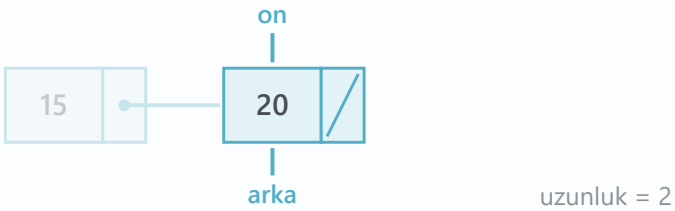
`sonuc = 15` `on = düğüm(20)` `arka = düğüm(20)` `uzunluk = 2`

ÇAĞRI YIĞINI

`dequeue()`

`main()`

Kod İzleme: Bağlı Liste Kuyruğu `dequeue`



`on == null` → false; arka aynı kalır.

Adım 10 / 17

```
1  public int dequeue() {
2      if (bosMu())
3          throw new NoSuchElementException();
4      int sonuc = on.veri;
5      on = on.sonraki;
▶      if (on == null)
7          arka = null;
8      uzunluk--;
9      return sonuc;
10 }
```

DEĞİŞKENLER

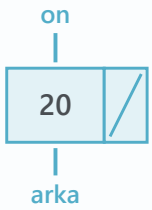
`sonuc = 15` `on = düğüm(20)` `arka = düğüm(20)` `uzunluk = 2`

ÇAĞRI YIĞINI

`dequeue()`

`main()`

Kod İzleme: Bağlı Liste Kuyruğu `dequeue`



uzunluk = 1

`uzunluk--` → 1; **return 15**.

Adım 11 / 17

```
1  public int dequeue() {
2      if (bosMu())
3          throw new NoSuchElementException();
4      int sonuc = on.veri;
5      on = on.sonraki;
6      if (on == null)
7          arka = null;
▶      uzunluk--;
▶      return sonuc;
10 }
```

DEĞİŞKENLER

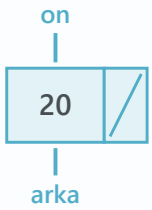
`sonuc = 15` `on = düğüm(20)` `arka = düğüm(20)` `uzunluk = 1`

ÇAĞRI YIĞINI

dequeue()

main()

Kod İzleme: Bağlı Liste Kuyruğu `dequeue`



uzunluk = 1

`bosMu()` → false (uzunluk 1).

Adım 12 / 17

```
1  public int dequeue() {
▶       if (bosMu())
3           throw new NoSuchElementException();
4       int sonuc = on.veri;
5       on = on.sonraki;
6       if (on == null)
7           arka = null;
8       uzunluk--;
9       return sonuc;
10 }
```

DEĞİŞKENLER

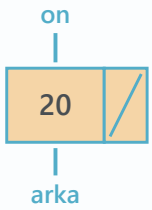
`sonuc = –` `on = düğüm(20)` `arka = düğüm(20)` `uzunluk = 1`

ÇAĞRI YIĞINI

`dequeue()`

`main()`

Kod İzleme: Bağlı Liste Kuyruğu `dequeue`



uzunluk = 1

sonuc = on.veri = 20.

Adım 13 / 17

```
1  public int dequeue() {
2      if (bosMu())
3          throw new NoSuchElementException();
▶      int sonuc = on.veri;
5      on = on.sonraki;
6      if (on == null)
7          arka = null;
8      uzunluk--;
9      return sonuc;
10 }
```

DEĞİŞKENLER

`sonuc = 20` `on = düğüm(20)` `arka = düğüm(20)` `uzunluk = 1`

ÇAĞRI YIĞINI

`dequeue()`

`main()`

Kod İzleme: Bağlı Liste Kuyruğu `dequeue`



on = null · uzunluk = 1

`on = on.sonraki` → **null**. Ama `arka` hâlâ silinen düğümü gösteriyor!

Adım 14 / 17

```
1  public int dequeue() {
2      if (bosMu())
3          throw new NoSuchElementException();
4      int sonuc = on.veri;
▶      on = on.sonraki;
6      if (on == null)
7          arka = null;
8      uzunluk--;
9      return sonuc;
10 }
```

DEĞİŞKENLER

`sonuc = 20` `on = null`
`arka = düğüm(20)` `uzunluk = 1`

ÇAĞRI YIĞINI

dequeue()
main()

Kod İzleme: Bağlı Liste Kuyruğu `dequeue`



on = null · uzunluk = 1

`on == null` → true: kuyruk boşaldı.

Adım 15 / 17

```
1  public int dequeue() {
2      if (bosMu())
3          throw new NoSuchElementException();
4      int sonuc = on.veri;
5      on = on.sonraki;
▶      if (on == null)
7          arka = null;
8      uzunluk--;
9      return sonuc;
10 }
```

DEĞİŞKENLER

`sonuc = 20` `on = null` `arka = düğüm(20)` `uzunluk = 1`

ÇAĞRI YIĞINI

`dequeue()`

`main()`

Kod İzleme: Bağlı Liste Kuyruğu `dequeue`



on = null · arka = null · uzunluk = 1

`arka = null` : bu satır olmasa kuyruk boşken arka silinen düğümü göstermeye devam ederdi (tutarsız durum).

Adım 16 / 17

```
1  public int dequeue() {
2      if (bosMu())
3          throw new NoSuchElementException();
4      int sonuc = on.veri;
5      on = on.sonraki;
6      if (on == null)
▶          arka = null;
8      uzunluk--;
9      return sonuc;
10 }
```

DEĞİŞKENLER

`sonuc = 20` `on = null` `arka = null` `uzunluk = 1`

ÇAĞRI YIĞINI

- **dequeue()**
- main()

Kod İzleme: Bağlı Liste Kuyruğu `dequeue`

on = null · arka = null · uzunluk = 0

`uzunluk--` → 0; **return 20**. Çıkış sırası 10, 15, 20 (FIFO), her işlem **O(1)**.

Adım 17 / 17

```
1  public int dequeue() {
2      if (bosMu())
3          throw new NoSuchElementException();
4      int sonuc = on.veri;
5      on = on.sonraki;
6      if (on == null)
7          arka = null;
▶      uzunluk--;
▶      return sonuc;
10 }
```

DEĞİŞKENLER

`sonuc = 20` `on = null` `arka = null` `uzunluk = 0`

ÇAĞRI YIĞINI

- **dequeue()**
- main()

Dairesel Dizi mi, Bağlı Liste mi?

Dairesel dizi

+ Bellekte ardışık; önbellek dostu ve hızlı.
+ Öğe başına ek referans yok.
+ Sabit kapasite: dolunca taşma ya da daha büyük diziye kopyalama (O(n), amortize O(1)).

Bağlı liste

+ Kapasite sınırı yok; gerektiği kadar büyür.
+ Her ekleme/çıkarma kesin O(1).
+ Her düğüm için ek referans ve nesne oluşturma maliyeti.

İşlem	Dairesel dizi	Bağlı liste
enqueue	O(1)	O(1)
dequeue	O(1)	O(1)
first / size / isEmpty	O(1)	O(1)
Bellek	O(N) (kapasite kadar)	O(n) (öğe sayısı kadar)

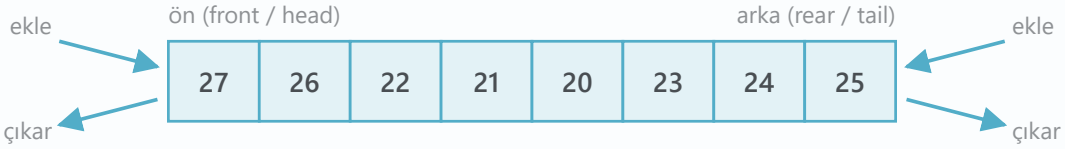
KUYRUK TÜRLERİ

Kuyruk Türleri

Tür	Ekleme	Çıkarma	Not
Basit kuyruk	arka	ön	FIFO
Giriş sınırlı (input restricted)	yalnızca arka	ön ve arka	kısıtlı deque
Çıkış sınırlı (output restricted)	ön ve arka	yalnızca ön	kısıtlı deque
Dairesel (circular)	arka	ön	dizi yuvaları yeniden kullanılır
Çift uçlu (deque)	ön ve arka	ön ve arka	FIFO'ya bağlı değil
Öncelikli (priority)	herhangi	en yüksek öncelik	sıra önceliğe göre

Deque, İngilizce *double-ended queue* kısaltmasıdır ve "dek" diye okunur. `dequeue` ise "kuyruktan çıkar" işleminin adıdır; ikisini karıştırmayın.

Çift Uçlu Kuyruk (Deque)



- Double-ended queue: hem ön hem arka uçtan ekleme ve çıkarma yapılır.
- FIFO kuralına bağlı değildir: yalnızca arka uçtan çalışan bir deque **yığın** (LIFO), arkadan ekleyip önden çıkaran bir deque **kuyruk** olur.

İşlem	Java `ArrayDeque`	Süre
basaEkle(e)	`offerFirst(e)`	O(1)
sonaEkle(e)	`offerLast(e)`	O(1)
bastanCikar()	`pollFirst()`	O(1)
sondanCikar()	`pollLast()`	O(1)
ilk() / son()	`peekFirst()` / `peekLast()`	O(1)

Deque İşlemleri Adım Adım

boş deque: bas = son = null

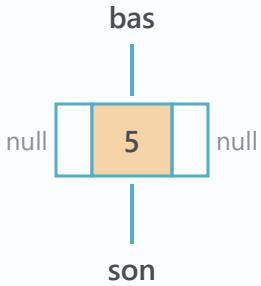
boyut = 0

Boş deque. Çift yönlü bağlı liste ile gerçekleyeceğiz: her düğümde `onceki` ve `sonraki`.

Adım 1 / 7

Tek yönlü listede sondan çıkarma O(n) olurdu (sondan bir önceki düğümü bulmak için). `onceki` referansı bunu O(1) yapar.

Deque İşlemleri Adım Adım



sonaEkle(5)

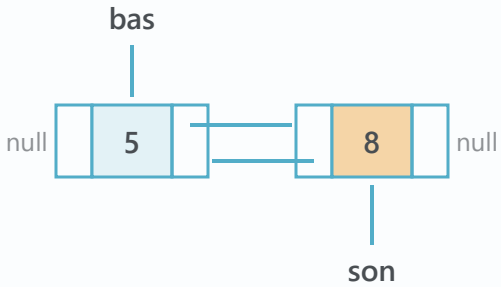
boyut = 1

İlk düğüm: `bas = son = yeni düğüm`.

Adım 2 / 7

Tek yönlü listede sondan çıkarma O(n) olurdu (sondan bir önceki düğümü bulmak için). `onceki` referansı bunu O(1) yapar.

Deque İşlemleri Adım Adım



sonaEkle(8)

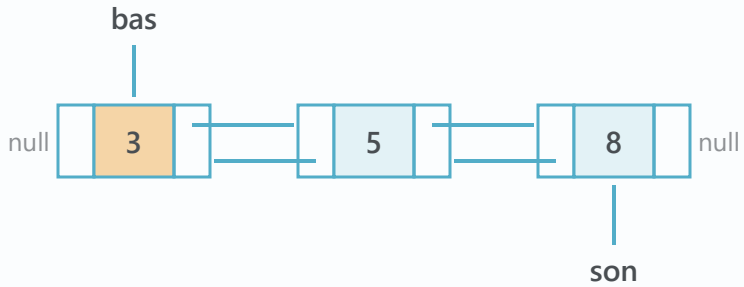
boyut = 2

Sona ekleme: `yeni.onceki = son; son.sonraki = yeni; son = yeni` .

Adım 3 / 7

Tek yönlü listede sondan çıkarma O(n) olurdu (sondan bir önceki düğümü bulmak için). `onceki` referansı bunu O(1) yapar.

Deque İşlemleri Adım Adım



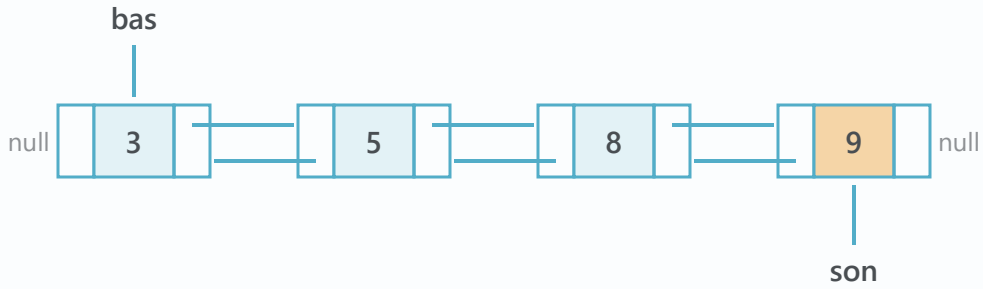
basaEkle(3) boyut = 3

Başa ekleme: `yeni.sonraki = bas; bas.onceki = yeni; bas = yeni` .

Adım 4 / 7

Tek yönlü listede sondan çıkarma O(n) olurdu (sondan bir önceki düğümü bulmak için). `onceki` referansı bunu O(1) yapar.

Deque İşlemleri Adım Adım

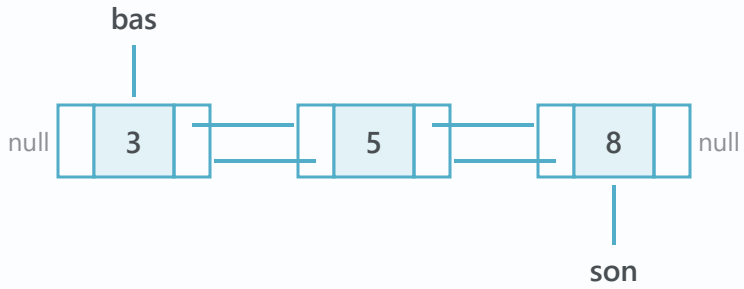


sonaEkle(9) boyut = 4

Her iki uç da referansla tutulduğu için tüm eklemeler O(1).

Tek yönlü listede sondan çıkarma O(n) olurdu (sondan bir önceki düğümü bulmak için). `onceki` referansı bunu O(1) yapar.

Deque İşlemleri Adım Adım



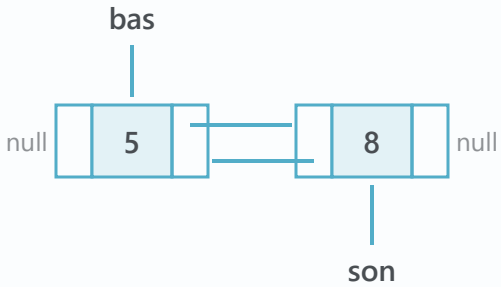
sondanCikar()

9 döndü · boyut = 3

Sondan çıkarma: `son = son.onceki; son.sonraki = null`. Önceki referansı sayesinde O(1).

Tek yönlü listede sondan çıkarma O(n) olurdu (sondan bir önceki düğümü bulmak için). `onceki` referansı bunu O(1) yapar.

Deque İşlemleri Adım Adım



bastanCikar()

3 döndü · boyut = 2

Baştan çıkarma: `bas = bas.sonraki; bas.onceki = null`.

Adım 7 / 7

Tek yönlü listede sondan çıkarma O(n) olurdu (sondan bir önceki düğümü bulmak için). `onceki` referansı bunu O(1) yapar.

Deque: Başa ve Sona Ekleme

```
public void basaEkle(E veri) {
    Dugum<E> d = new Dugum<>(veri);
    if (bosMu()) {
        bas = son = d;
    } else {
        d.sonraki = bas;
        bas.onceki = d;
        bas = d;
    }
    boyut++;
}
```

```
public void sonaEkle(E veri) {
    Dugum<E> d = new Dugum<>(veri);
    if (bosMu()) {
        bas = son = d;
    } else {
        d.onceki = son;
        son.sonraki = d;
        son = d;
    }
    boyut++;
}
```

```
class Dugum<E> {
    E veri;
    Dugum<E> onceki, sonraki;
    Dugum(E veri) { this.veri = veri; }
}
```

İki metot birbirinin aynadaki görüntüsüdür: `bas` ↔ `son`, `onceki` ↔ `sonraki`.

Kod İzleme: Deque `basaEkle()`

bas = null · son = null · boyut = 0

Boş deque. Önce `basaEkle(5)`, sonra `basaEkle(3)`: 3 başa geçecek.

Adım 1 / 11

```
▶ public void basaEkle(E veri) {
2     Dugum<E> d = new Dugum<>(veri);
3     if (bosMu()) {
4         bas = son = d;
5     } else {
6         d.sonraki = bas;
7         bas.onceki = d;
8         bas = d;
9     }
10    boyut++;
11 }
```

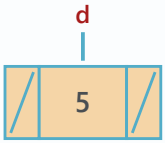
DEĞİŞKENLER

`veri = –` `d = –` `bas = null` `son = null` `boyut = 0`

ÇAĞRI YIĞINI

`main()`

Kod İzleme: Deque `basaEkle()`



bas = null · son = null · boyut = 0

`new Dugum<>(5)` : onceki = sonraki = null.

Adım 2 / 11

```
1  public void basaEkle(E veri) {
▶      Dugum<E> d = new Dugum<>(veri);
3      if (bosMu()) {
4          bas = son = d;
5      } else {
6          d.sonraki = bas;
7          bas.onceki = d;
8          bas = d;
9      }
10     boyut++;
11 }
```

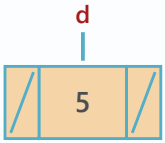
DEĞİŞKENLER

`veri = 5` `d = düğüm(5)` `bas = null` `son = null` `boyut = 0`

ÇAĞRI YIĞINI

- **basaEkle(5)**
- main()

Kod İzleme: Deque `basaEkle()`



bas = null · son = null · boyut = 0

`bosMu()` → **true**.

Adım 3 / 11

```
1  public void basaEkle(E veri) {
2      Dugum<E> d = new Dugum<>(veri);
▶      if (bosMu()) {
4          bas = son = d;
5      } else {
6          d.sonraki = bas;
7          bas.onceki = d;
8          bas = d;
9      }
10     boyut++;
11 }
```

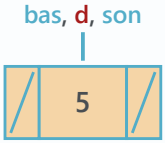
DEĞİŞKENLER

`veri = 5` `d = düğüm(5)` `bas = null` `son = null` `boyut = 0`

ÇAĞRI YIĞINI

- **`basaEkle(5)`**
- `main()`

Kod İzleme: Deque `basaEkle()`



boyut = 0

`bas = son = d` : tek düğüm iki ucu birden oluşturur.

Adım 4 / 11

```
1  public void basaEkle(E veri) {
2      Dugum<E> d = new Dugum<>(veri);
3      if (bosMu()) {
▶          bas = son = d;
5      } else {
6          d.sonraki = bas;
7          bas.onceki = d;
8          bas = d;
9      }
10     boyut++;
11 }
```

DEĞİŞKENLER

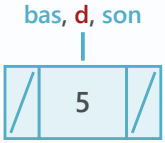
`veri = 5` `d = düğüm(5)` `bas = düğüm(5)` `son = düğüm(5)` `boyut = 0`

ÇAĞRI YIĞINI

`basaEkle(5)`

`main()`

Kod İzleme: Deque `basaEkle()`



boyut = 1

`boyut++` → 1.

Adım 5 / 11

```
1  public void basaEkle(E veri) {
2      Dugum<E> d = new Dugum<>(veri);
3      if (bosMu()) {
4          bas = son = d;
5      } else {
6          d.sonraki = bas;
7          bas.onceki = d;
8          bas = d;
9      }
▶      boyut++;
11 }
```

DEĞİŞKENLER

`veri = 5` `d = düğüm(5)` `bas = düğüm(5)` `son = düğüm(5)` `boyut = 1`

ÇAĞRI YIĞINI

- **basaEkle(5)**
- main()

Kod İzleme: Deque `basaEkle()`



boyut = 1

`new Dugum<>(3)` : onceki = sonraki = null.

Adım 6 / 11

```
1  public void basaEkle(E veri) {
▶       Dugum<E> d = new Dugum<>(veri);
3       if (bosMu()) {
4           bas = son = d;
5       } else {
6           d.sonraki = bas;
7           bas.onceki = d;
8           bas = d;
9       }
10      boyut++;
11 }
```

DEĞİŞKENLER

`veri = 3` `d = düğüm(3)` `bas = düğüm(5)` `son = düğüm(5)` `boyut = 1`

ÇAĞRI YIĞINI

`basaEkle(3)`

`main()`

Kod İzleme: Deque `basaEkle()`



boyut = 1

`bosMu()` → **false**.

Adım 7 / 11

```
1  public void basaEkle(E veri) {
2      Dugum<E> d = new Dugum<>(veri);
▶      if (bosMu()) {
4          bas = son = d;
5      } else {
6          d.sonraki = bas;
7          bas.onceki = d;
8          bas = d;
9      }
10     boyut++;
11 }
```

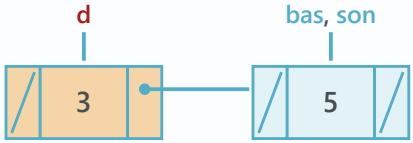
DEĞİŞKENLER

`veri` = **3** | `d` = **düğüm(3)** | `bas` = **düğüm(5)** | `son` = **düğüm(5)** | `boyut` = **1**

ÇAĞRI YIĞINI

- **`basaEkle(3)`**
- `main()`

Kod İzleme: Deque `basaEkle()`



boyut = 1

`d.sonraki = bas` : yeni düğüm eski başı (5) gösterir.

Adım 8 / 11

```
1  public void basaEkle(E veri) {
2      Dugum<E> d = new Dugum<>(veri);
3      if (bosMu()) {
4          bas = son = d;
5      } else {
▶          d.sonraki = bas;
7          bas.onceki = d;
8          bas = d;
9      }
10     boyut++;
11 }
```

DEĞİŞKENLER

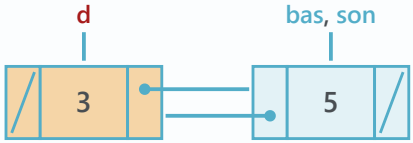
`veri = 3` `d = düğüm(3)` `bas = düğüm(5)` `son = düğüm(5)` `boyut = 1`

ÇAĞRI YIĞINI

`basaEkle(3)`

`main()`

Kod İzleme: Deque `basaEkle()`



boyut = 1

`bas.onceki = d` : geri bağlantı; artık iki yönde de gezilebilir.

Adım 9 / 11

```
1  public void basaEkle(E veri) {
2      Dugum<E> d = new Dugum<>(veri);
3      if (bosMu()) {
4          bas = son = d;
5      } else {
6          d.sonraki = bas;
▶          bas.onceki = d;
8          bas = d;
9      }
10     boyut++;
11 }
```

DEĞİŞKENLER

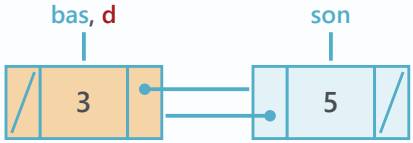
`veri = 3` `d = düğüm(3)` `bas = düğüm(5)` `son = düğüm(5)` `boyut = 1`

ÇAĞRI YIĞINI

`basaEkle(3)`

`main()`

Kod İzleme: Deque `basaEkle()`



boyut = 1

`bas = d` : baş referansı yeni düğüme geçer.

Adım 10 / 11

```
1  public void basaEkle(E veri) {
2      Dugum<E> d = new Dugum<>(veri);
3      if (bosMu()) {
4          bas = son = d;
5      } else {
6          d.sonraki = bas;
7          bas.onceki = d;
▶          bas = d;
9      }
10     boyut++;
11 }
```

DEĞİŞKENLER

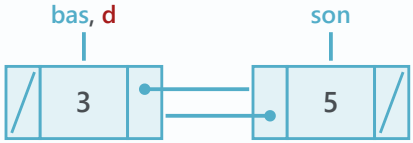
`veri = 3` `d = düğüm(3)` `bas = düğüm(3)` `son = düğüm(5)` `boyut = 1`

ÇAĞRI YIĞINI

`basaEkle(3)`

`main()`

Kod İzleme: Deque `basaEkle()`



boyut = 2

`boyut++` → 2. Hiç gezinme yok: **O(1)**. `sonaEkle` bunun aynadaki görüntüsüdür.

Adım 11 / 11

```
1  public void basaEkle(E veri) {
2      Dugum<E> d = new Dugum<>(veri);
3      if (bosMu()) {
4          bas = son = d;
5      } else {
6          d.sonraki = bas;
7          bas.onceki = d;
8          bas = d;
9      }
▶      boyut++;
11 }
```

DEĞİŞKENLER

`veri = 3` `d = düğüm(3)` `bas = düğüm(3)` `son = düğüm(5)` `boyut = 2`

ÇAĞRI YIĞINI

- **basaEkle(3)**
- main()

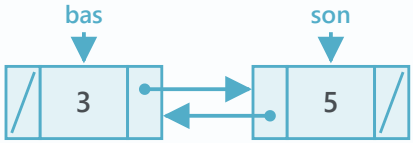
Deque: Baştan ve Sondan Çıkarma

```java
public E bastanCikar() {
    if (bosMu()) return null;
    E veri = bas.veri;
    bas = bas.sonraki;
    if (bas == null)
        son = null;       // boşaldı
    else
        bas.onceki = null;
    boyut--;
    return veri;
}
```

```java
public E sondanCikar() {
    if (bosMu()) return null;
    E veri = son.veri;
    son = son.onceki;
    if (son == null)
        bas = null;       // boşaldı
    else
        son.sonraki = null;
    boyut--;
    return veri;
}
```

Son öğe çıkarıldığında **iki uç da** null yapılmalıdır. Aksi hâlde `bas` null iken `son` silinmiş düğümü göstermeye devam eder; düğüm bellekten atılamaz ve `son()` yanlış değer döndürür. Boş deque'de Goodrich'teki gibi `null` döndürülür.

Kod İzleme: Deque `sondanCikar()`



boyut = 2

Deque: 3 ⇄ 5. İki kez `sondanCikar()` : önce 5, sonra 3 çıkacak ve deque boşalacak.

Adım 1 / 13

```
▶ public E sondanCikar() {
2     if (bosMu()) return null;
3     E veri = son.veri;
4     son = son.onceki;
5     if (son == null)
6         bas = null;        // boşaldı
7     else
8         son.sonraki = null;
9     boyut--;
10    return veri;
11 }
```

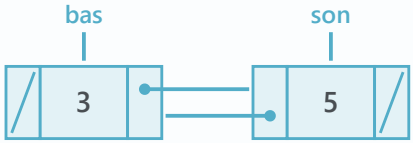
DEĞİŞKENLER

`veri = –` `bas = düğüm(3)` `son = düğüm(5)` `boyut = 2`

ÇAĞRI YIĞINI

`main()`

Kod İzleme: Deque `sondanCikar()`



boyut = 2

`bosMu()` → false (boyut 2).

Adım 2 / 13

```
1  public E sondanCikar() {
▶      if (bosMu()) return null;
3      E veri = son.veri;
4      son = son.onceki;
5      if (son == null)
6          bas = null;        // boşaldı
7      else
8          son.sonraki = null;
9      boyut--;
10     return veri;
11 }
```

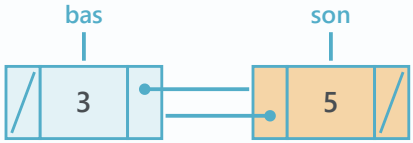
DEĞİŞKENLER

`veri = –` `bas = düğüm(3)` `son = düğüm(5)` `boyut = 2`

ÇAĞRI YIĞINI

- **`sondanCikar()`**
- `main()`

Kod İzleme: Deque `sondanCikar()`



boyut = 2

veri = son.veri = 5.

Adım 3 / 13

```
1  public E sondanCikar() {
2      if (bosMu()) return null;
▶      E veri = son.veri;
4      son = son.onceki;
5      if (son == null)
6          bas = null;        // boşaldı
7      else
8          son.sonraki = null;
9      boyut--;
10     return veri;
11 }
```

DEĞİŞKENLER

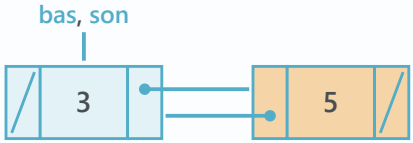
`veri = 5` `bas = düğüm(3)` `son = düğüm(5)` `boyut = 2`

ÇAĞRI YIĞINI

`sondanCikar()`

`main()`

Kod İzleme: Deque `sondanCikar()`



boyut = 2

`son = son.onceki` → düğüm(3). `onceki` sayesinde sondan bir öncekini aramaya gerek yok.

Adım 4 / 13

```
1  public E sondanCikar() {
2      if (bosMu()) return null;
3      E veri = son.veri;
▶      son = son.onceki;
5      if (son == null)
6          bas = null;        // boşaldı
7      else
8          son.sonraki = null;
9      boyut--;
10     return veri;
11 }
```

DEĞİŞKENLER

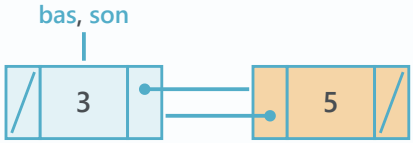
`veri = 5` `bas = düğüm(3)` `son = düğüm(3)` `boyut = 2`

ÇAĞRI YIĞINI

`sondanCikar()`

`main()`

Kod İzleme: Deque `sondanCikar()`



boyut = 2

`son == null` → false: deque boşalmadı.

Adım 5 / 13

```
1  public E sondanCikar() {
2      if (bosMu()) return null;
3      E veri = son.veri;
4      son = son.onceki;
▶      if (son == null)
6          bas = null;        // boşaldı
7      else
8          son.sonraki = null;
9      boyut--;
10     return veri;
11 }
```

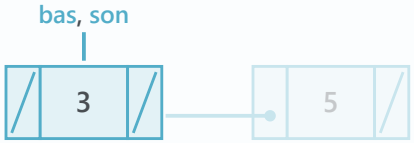
DEĞİŞKENLER

`veri = 5` `bas = düğüm(3)` `son = düğüm(3)` `boyut = 2`

ÇAĞRI YIĞINI

- **`sondanCikar()`**
- `main()`

Kod İzleme: Deque `sondanCikar()`



boyut = 2

`son.sonraki = null` : 5 düğümüne giden bağ kopar; düğüm listeden ayrılır.

Adım 6 / 13

```
1  public E sondanCikar() {
2      if (bosMu()) return null;
3      E veri = son.veri;
4      son = son.onceki;
5      if (son == null)
6          bas = null;        // boşaldı
7      else
▶          son.sonraki = null;
9      boyut--;
10     return veri;
11 }
```

DEĞİŞKENLER

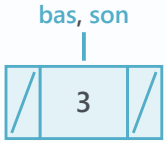
`veri = 5` `bas = düğüm(3)` `son = düğüm(3)` `boyut = 2`

ÇAĞRI YIĞINI

`sondanCikar()`

`main()`

Kod İzleme: Deque `sondanCikar()`



boyut = 1

`boyut--` → 1; **return 5**.

Adım 7 / 13

```
1  public E sondanCikar() {
2      if (bosMu()) return null;
3      E veri = son.veri;
4      son = son.onceki;
5      if (son == null)
6          bas = null;        // boşaldı
7      else
8          son.sonraki = null;
▶      boyut--;
▶      return veri;
11 }
```

DEĞİŞKENLER

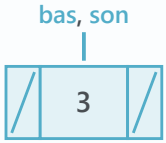
`veri = 5` `bas = düğüm(3)` `son = düğüm(3)` `boyut = 1`

ÇAĞRI YIĞINI

`sondanCikar()`

`main()`

Kod İzleme: Deque `sondanCikar()`



boyut = 1

`bosMu()` → false (boyut 1).

Adım 8 / 13

```
1  public E sondanCikar() {
▶      if (bosMu()) return null;
3      E veri = son.veri;
4      son = son.onceki;
5      if (son == null)
6          bas = null;        // boşaldı
7      else
8          son.sonraki = null;
9      boyut--;
10     return veri;
11 }
```

DEĞİŞKENLER

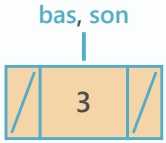
`veri = –` `bas = düğüm(3)` `son = düğüm(3)` `boyut = 1`

ÇAĞRI YIĞINI

`sondanCikar()`

`main()`

Kod İzleme: Deque `sondanCikar()`



boyut = 1

veri = son.veri = 3.

Adım 9 / 13

```
1  public E sondanCikar() {
2      if (bosMu()) return null;
▶      E veri = son.veri;
4      son = son.onceki;
5      if (son == null)
6          bas = null;        // boşaldı
7      else
8          son.sonraki = null;
9      boyut--;
10     return veri;
11 }
```

DEĞİŞKENLER

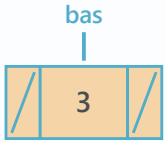
`veri = 3` `bas = düğüm(3)` `son = düğüm(3)` `boyut = 1`

ÇAĞRI YIĞINI

`sondanCikar()`

`main()`

Kod İzleme: Deque `sondanCikar()`



son = null · boyut = 1

`son = son.onceki` → **null** (tek düğüm vardı). Ama bas hâlâ onu gösteriyor!

Adım 10 / 13

```
1  public E sondanCikar() {
2      if (bosMu()) return null;
3      E veri = son.veri;
▶      son = son.onceki;
5      if (son == null)
6          bas = null;        // boşaldı
7      else
8          son.sonraki = null;
9      boyut--;
10     return veri;
11 }
```

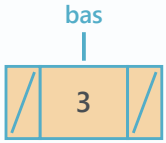
DEĞİŞKENLER

`veri = 3` `bas = düğüm(3)` `son = null` `boyut = 1`

ÇAĞRI YIĞINI

- **sondanCikar()**
- main()

Kod İzleme: Deque `sondanCikar()`



son = null · boyut = 1

`son == null` → true: deque boşaldı.

Adım 11 / 13

```
1  public E sondanCikar() {
2      if (bosMu()) return null;
3      E veri = son.veri;
4      son = son.onceki;
▶      if (son == null)
6          bas = null;        // boşaldı
7      else
8          son.sonraki = null;
9      boyut--;
10     return veri;
11 }
```

DEĞİŞKENLER

`veri = 3` `bas = düğüm(3)` `son = null` `boyut = 1`

ÇAĞRI YIĞINI

- **`sondanCikar()`**
- `main()`

Kod İzleme: Deque `sondanCikar()`



bas = null · son = null · boyut = 1

`bas = null` : iki uç da null olmalı; yoksa bas silinmiş düğümü gösterirdi.

Adım 12 / 13

```
1  public E sondanCikar() {
2      if (bosMu()) return null;
3      E veri = son.veri;
4      son = son.onceki;
5      if (son == null)
▶          bas = null;        // boşaldı
7      else
8          son.sonraki = null;
9      boyut--;
10     return veri;
11 }
```

DEĞİŞKENLER

`veri = 3` `bas = null` `son = null` `boyut = 1`

ÇAĞRI YIĞINI

`sondanCikar()`

`main()`

Kod İzleme: Deque `sondanCikar()`

bas = null · son = null · boyut = 0

`boyut--` → 0; **return 3**. Her iki durumda da **O(1)**.

Adım 13 / 13

```
1  public E sondanCikar() {
2      if (bosMu()) return null;
3      E veri = son.veri;
4      son = son.onceki;
5      if (son == null)
6          bas = null;        // boşaldı
7      else
8          son.sonraki = null;
▶      boyut--;
▶      return veri;
11 }
```

DEĞİŞKENLER

`veri = 3` `bas = null` `son = null` `boyut = 0`

ÇAĞRI YIĞINI

- **sondanCikar()**
- main()

Deque Uygulaması: Kayan Pencerede En Büyük



k = 3. d indis tutar; değerler önden arkaya **azalır**, önde en büyük durur.

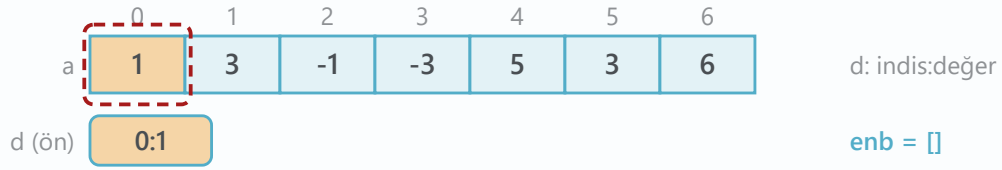
Adım 1 / 9

```
1  int[] pencereMax(int[] a, int k) {
▶    int[] enb = new int[a.length - k + 1];
▶    Deque<Integer> d = new ArrayDeque<>(); // indis
4    for (int i = 0; i < a.length; i++) {
5      if (!d.isEmpty() && d.peekFirst() <= i - k)
6        d.pollFirst();        // pencere dışı
7      while (!d.isEmpty() && a[d.peekLast()] <= a[i])
8        d.pollLast();         // aday değil
9      d.offerLast(i);
10     if (i >= k - 1) enb[i - k + 1] = a[d.peekFirst()];
11   }
12   return enb;
13 }
```

DEĞİŞKENLER

`k = 3` `i = –` `d = []` `enb = []`

Deque Uygulaması: Kayan Pencerede En Büyük



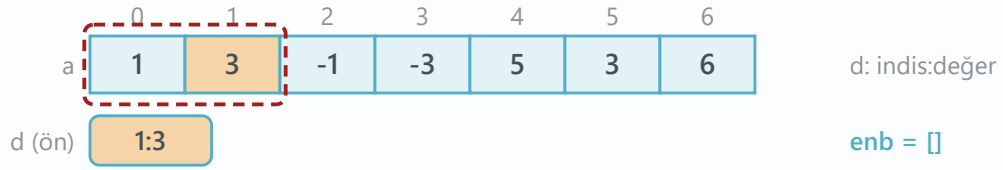
a[0] = 1: 0 arkaya.


```
1  int[] pencereMax(int[] a, int k) {
2    int[] enb = new int[a.length - k + 1];
3    Deque<Integer> d = new ArrayDeque<>(); // indis
▶    for (int i = 0; i < a.length; i++) {
5      if (!d.isEmpty() && d.peekFirst() <= i - k)
6        d.pollFirst();        // pencere dışı
7      while (!d.isEmpty() && a[d.peekLast()] <= a[i])
8        d.pollLast();         // aday değil
▶      d.offerLast(i);
10     if (i >= k - 1) enb[i - k + 1] = a[d.peekFirst()];
11   }
12   return enb;
13 }
```

DEĞİŞKENLER

`k = 3` `i = 0` `d = [0]` `enb = []`

Deque Uygulaması: Kayan Pencerede En Büyük



a[1] = 3: arkadan at 0:1. 1 arkaya.

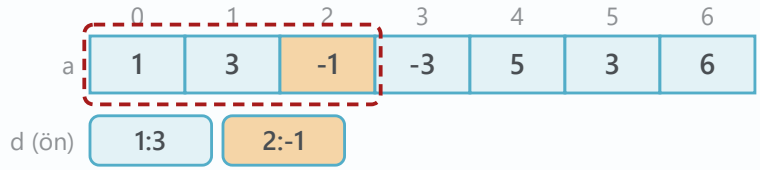
Adım 3 / 9

```
1  int[] pencereMax(int[] a, int k) {
2    int[] enb = new int[a.length - k + 1];
3    Deque<Integer> d = new ArrayDeque<>(); // indis
▶    for (int i = 0; i < a.length; i++) {
5      if (!d.isEmpty() && d.peekFirst() <= i - k)
6        d.pollFirst();        // pencere dışı
7      while (!d.isEmpty() && a[d.peekLast()] <= a[i])
▶        d.pollLast();         // aday değil
▶      d.offerLast(i);
10     if (i >= k - 1) enb[i - k + 1] = a[d.peekFirst()];
11   }
12   return enb;
13 }
```

DEĞİŞKENLER

`k = 3` `i = 1` `d = [1]` `enb = []`

Deque Uygulaması: Kayan Pencerede En Büyük



a[2] = -1: 2 arkaya. En büyük = **3**.

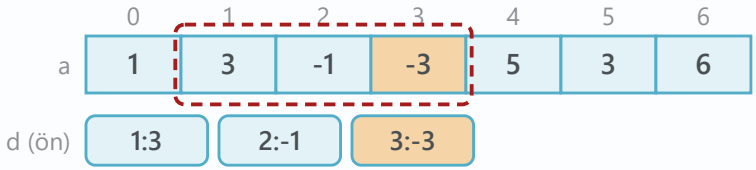
Adım 4 / 9

```
1  int[] pencereMax(int[] a, int k) {
2    int[] enb = new int[a.length - k + 1];
3    Deque<Integer> d = new ArrayDeque<>(); // indis
▶    for (int i = 0; i < a.length; i++) {
5      if (!d.isEmpty() && d.peekFirst() <= i - k)
6        d.pollFirst();        // pencere dışı
7      while (!d.isEmpty() && a[d.peekLast()] <= a[i])
8        d.pollLast();         // aday değil
▶      d.offerLast(i);
▶      if (i >= k - 1) enb[i - k + 1] = a[d.peekFirst()];
11   }
12   return enb;
13 }
```

DEĞİŞKENLER

`k = 3` `i = 2` `d = [1, 2]` `enb = [3]`

Deque Uygulaması: Kayan Pencerede En Büyük



d: indis:değer

enb = [3, 3]

a[3] = -3: 3 arkaya. En büyük = **3**.

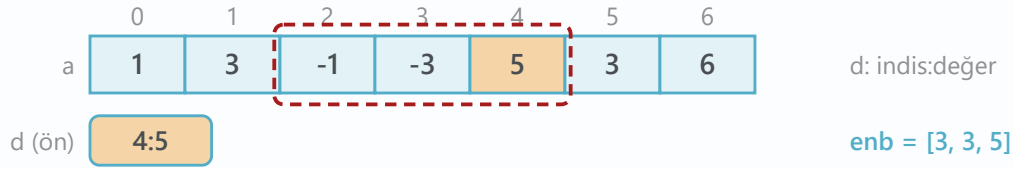
Adım 5 / 9

```
1  int[] pencereMax(int[] a, int k) {
2    int[] enb = new int[a.length - k + 1];
3    Deque<Integer> d = new ArrayDeque<>(); // indis
▶    for (int i = 0; i < a.length; i++) {
5      if (!d.isEmpty() && d.peekFirst() <= i - k)
6        d.pollFirst();        // pencere dışı
7      while (!d.isEmpty() && a[d.peekLast()] <= a[i])
8        d.pollLast();         // aday değil
▶      d.offerLast(i);
▶      if (i >= k - 1) enb[i - k + 1] = a[d.peekFirst()];
11   }
12   return enb;
13 }
```

DEĞİŞKENLER

`k = 3` `i = 3` `d = [1, 2, 3]` `enb = [3, 3]`

Deque Uygulaması: Kayan Pencerede En Büyük



a[4] = 5: önden at 1; arkadan at 3:-3; arkadan at 2:-1. 4 arkaya. En büyük = **5**.

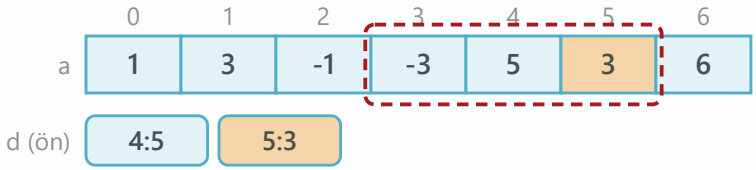
Adım 6 / 9

```
1  int[] pencereMax(int[] a, int k) {
2    int[] enb = new int[a.length - k + 1];
3    Deque<Integer> d = new ArrayDeque<>(); // indis
▶    for (int i = 0; i < a.length; i++) {
5      if (!d.isEmpty() && d.peekFirst() <= i - k)
▶        d.pollFirst();        // pencere dışı
7      while (!d.isEmpty() && a[d.peekLast()] <= a[i])
▶        d.pollLast();         // aday değil
▶      d.offerLast(i);
▶      if (i >= k - 1) enb[i - k + 1] = a[d.peekFirst()];
11   }
12   return enb;
13 }
```

DEĞİŞKENLER

`k = 3` `i = 4` `d = [4]` `enb = [3, 3, 5]`

Deque Uygulaması: Kayan Pencerede En Büyük



d: indis:değer

enb = [3, 3, 5, 5]

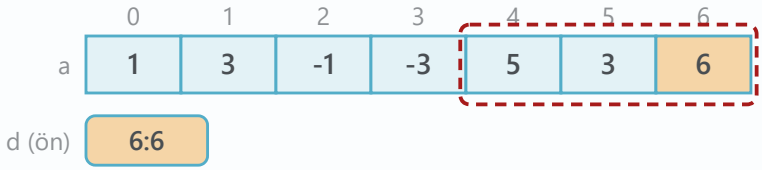
a[5] = 3: 5 arkaya. En büyük = **5**.


```
1  int[] pencereMax(int[] a, int k) {
2    int[] enb = new int[a.length - k + 1];
3    Deque<Integer> d = new ArrayDeque<>(); // indis
▶    for (int i = 0; i < a.length; i++) {
5      if (!d.isEmpty() && d.peekFirst() <= i - k)
6        d.pollFirst();        // pencere dışı
7      while (!d.isEmpty() && a[d.peekLast()] <= a[i])
8        d.pollLast();         // aday değil
▶      d.offerLast(i);
▶      if (i >= k - 1) enb[i - k + 1] = a[d.peekFirst()];
11   }
12   return enb;
13 }
```

DEĞİŞKENLER

`k = 3` `i = 5` `d = [4, 5]` `enb = [3, 3, 5, 5]`

Deque Uygulaması: Kayan Pencerede En Büyük



d: indis:değer

enb = [3, 3, 5, 5, 6]

a[6] = 6: arkadan at 5:3; arkadan at 4:5. 6 arkaya. En büyük = **6**.

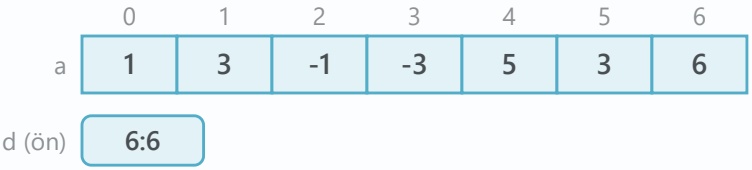
Adım 8 / 9

```
1  int[] pencereMax(int[] a, int k) {
2    int[] enb = new int[a.length - k + 1];
3    Deque<Integer> d = new ArrayDeque<>(); // indis
▶    for (int i = 0; i < a.length; i++) {
5      if (!d.isEmpty() && d.peekFirst() <= i - k)
6        d.pollFirst();        // pencere dışı
7      while (!d.isEmpty() && a[d.peekLast()] <= a[i])
▶        d.pollLast();         // aday değil
▶      d.offerLast(i);
▶      if (i >= k - 1) enb[i - k + 1] = a[d.peekFirst()];
11   }
12   return enb;
13 }
```

DEĞİŞKENLER

k = 3 | i = 6 | d = [6] | enb = [3, 3, 5, 5, 6]

Deque Uygulaması: Kayan Pencerede En Büyük



d: indis:değer

enb = [3, 3, 5, 5, 6]

Her indis bir kez girer, en çok bir kez çıkar → **O(n)** (kaba kuvvet O(n·k)).

Adım 9 / 9

```
1  int[] pencereMax(int[] a, int k) {
2    int[] enb = new int[a.length - k + 1];
3    Deque<Integer> d = new ArrayDeque<>(); // indis
4    for (int i = 0; i < a.length; i++) {
5      if (!d.isEmpty() && d.peekFirst() <= i - k)
6        d.pollFirst();        // pencere dışı
7      while (!d.isEmpty() && a[d.peekLast()] <= a[i])
8        d.pollLast();         // aday değil
9      d.offerLast(i);
10     if (i >= k - 1) enb[i - k + 1] = a[d.peekFirst()];
11   }
▶    return enb;
13 }
```

DEĞİŞKENLER

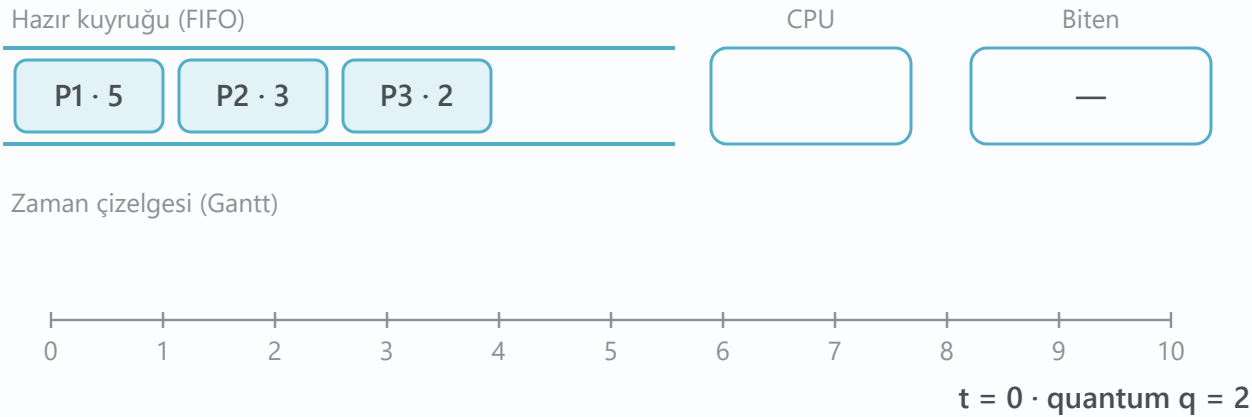
k = **3** | i = **–** | d = **[6]**

enb = **[3, 3, 5, 5, 6]**

UYGULAMALAR

Uygulama: Round Robin Çizelgeleyici

İşletim sistemi CPU'yu süreçler arasında sırayla paylaştırır: kuyruğun önündeki süreç en fazla **q** birim çalışır, bitmediyse kuyruğun arkasına döner.



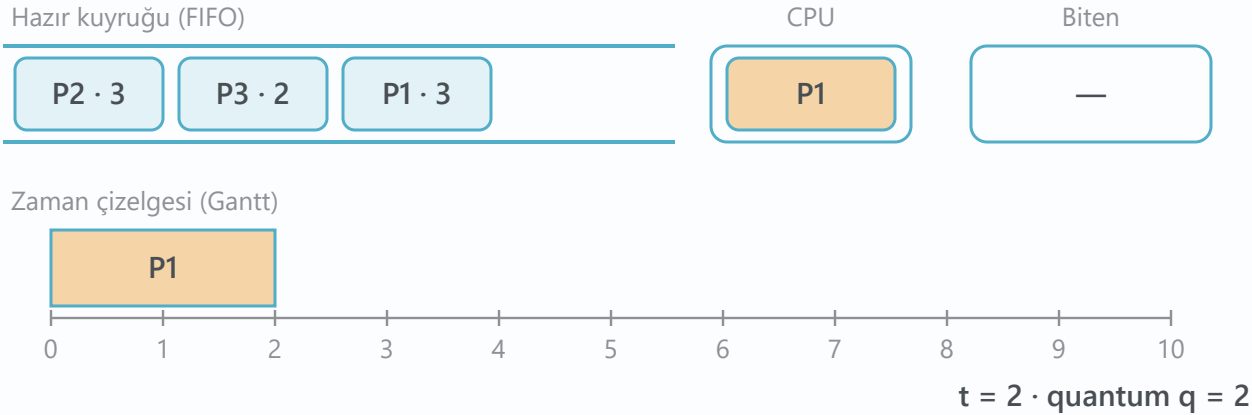
Üç süreç hazır kuyruğunda; kalan süreleri 5, 3, 2 birim. Zaman dilimi (quantum) q = 2.

Adım 1 / 7

Simülatör: Round Robin Çizelgeleyici (ayarlanabilir quantum ve süreçler).

Uygulama: Round Robin Çizelgeleyici

İşletim sistemi CPU'yu süreçler arasında sırayla paylaştırır: kuyruğun önündeki süreç en fazla **q** birim çalışır, bitmediyse kuyruğun arkasına döner.

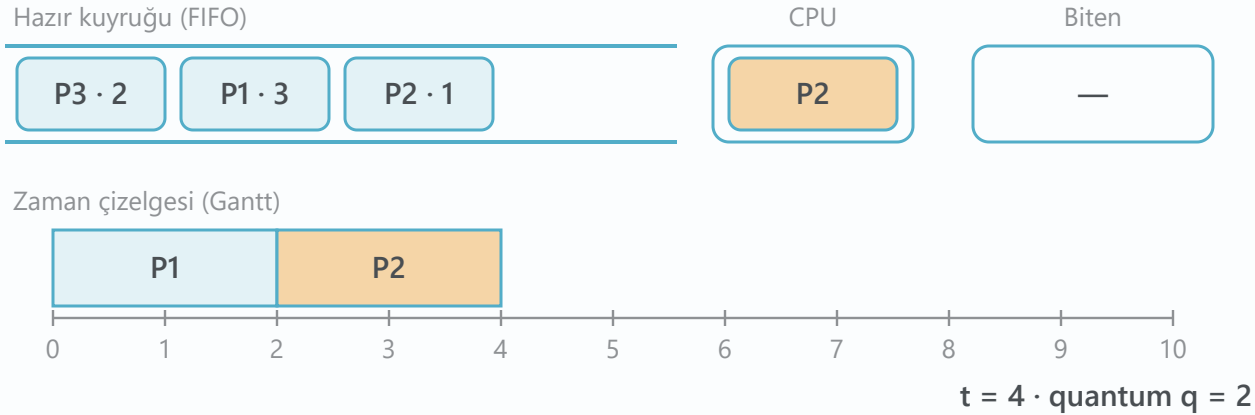


P1 önden alınır (dequeue), 2 birim çalışır, kalan 3 ile arkaya eklenir (enqueue).

Simülatör: Round Robin Çizelgeleyici (ayarlanabilir quantum ve süreçler).

Uygulama: Round Robin Çizelgeleyici

İşletim sistemi CPU'yu süreçler arasında sırayla paylaştırır: kuyruğun önündeki süreç en fazla **q** birim çalışır, bitmediyse kuyruğun arkasına döner.

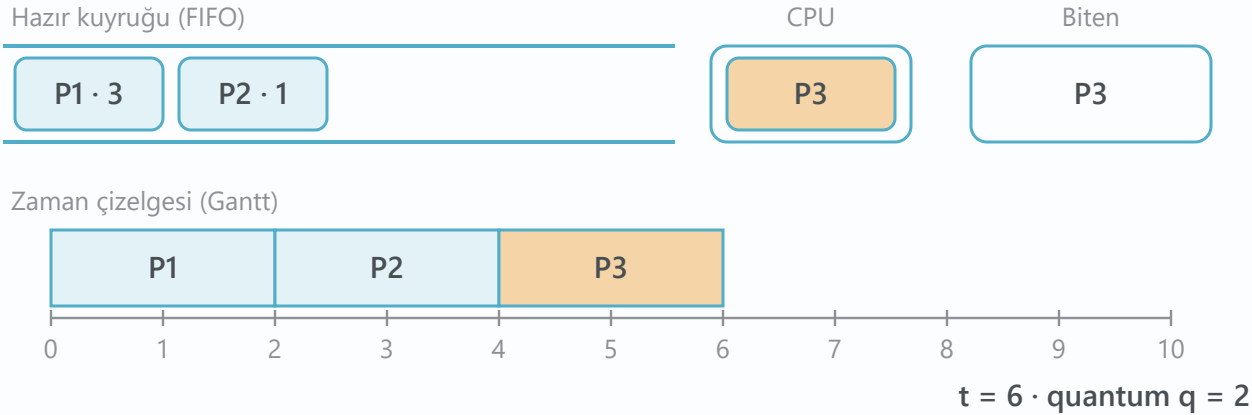


P2 2 birim çalışır, kalan 1 ile arkaya döner.

Simülatör: Round Robin Çizelgeleyici (ayarlanabilir quantum ve süreçler).

Uygulama: Round Robin Çizelgeleyici

İşletim sistemi CPU'yu süreçler arasında sırayla paylaştırır: kuyruğun önündeki süreç en fazla **q** birim çalışır, bitmediyse kuyruğun arkasına döner.

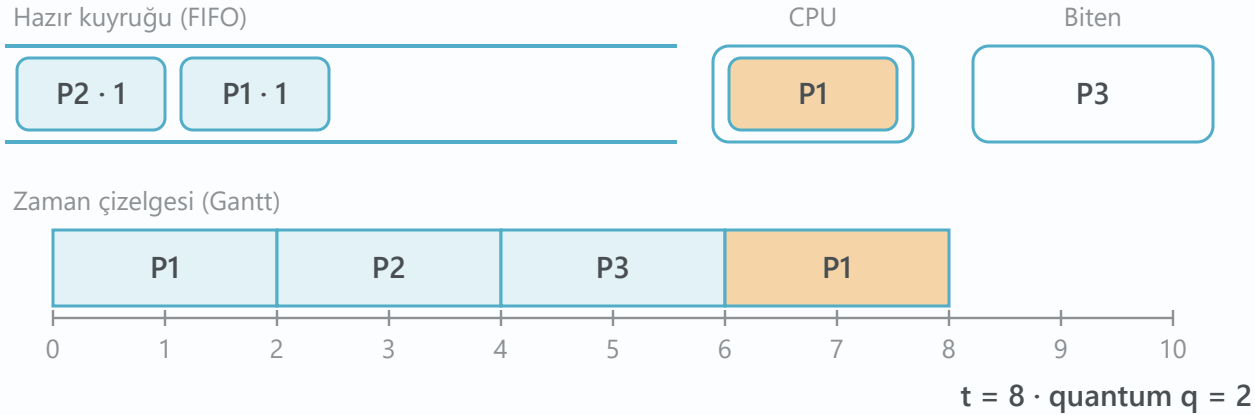


P3'ün 2 birimi bir dilime sığar → t = 6'da biter, kuyruğa dönmez.

Simülatör: Round Robin Çizelgeleyici (ayarlanabilir quantum ve süreçler).

Uygulama: Round Robin Çizelgeleyici

İşletim sistemi CPU'yu süreçler arasında sırayla paylaştırır: kuyruğun önündeki süreç en fazla **q** birim çalışır, bitmediyse kuyruğun arkasına döner.

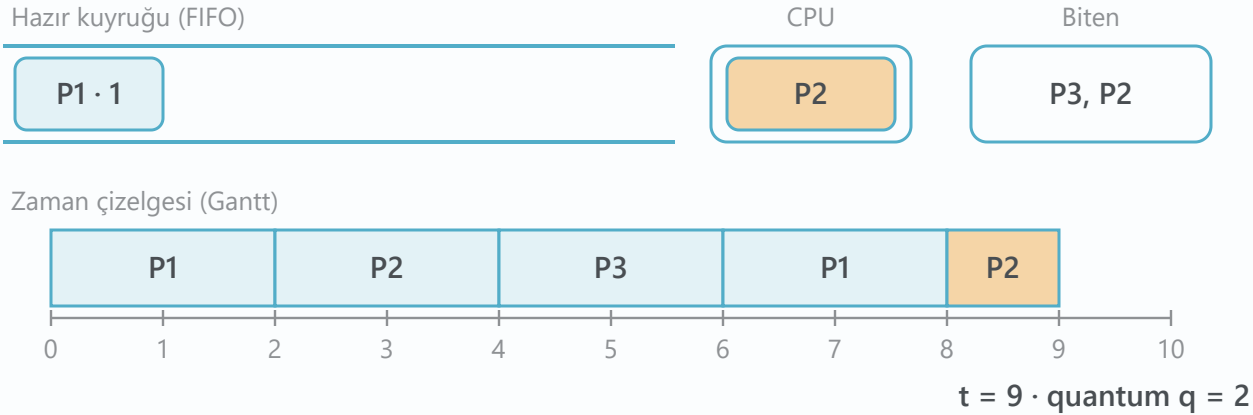


P1 yine 2 birim çalışır; kalan 1.

Simülatör: Round Robin Çizelgeleyici (ayarlanabilir quantum ve süreçler).

Uygulama: Round Robin Çizelgeleyici

İşletim sistemi CPU'yu süreçler arasında sırayla paylaştırır: kuyruğun önündeki süreç en fazla **q** birim çalışır, bitmediyse kuyruğun arkasına döner.



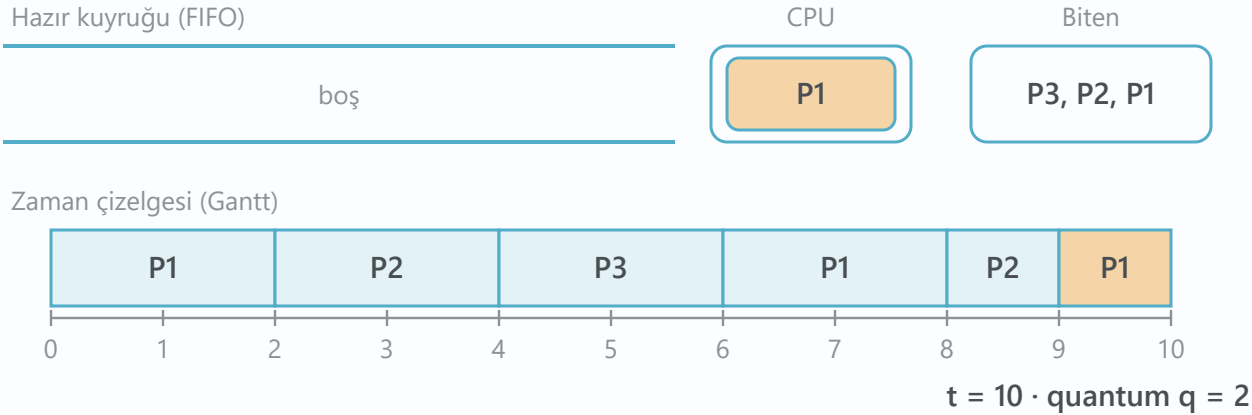
P2'nin kalan 1 birimi biter (t = 9); dilimin tamamı kullanılmaz.

Simülatör: Round Robin Çizelgeleyici (ayarlanabilir quantum ve süreçler).

UYGULAMALAR

Uygulama: Round Robin Çizelgeleyici

İşletim sistemi CPU'yu süreçler arasında sırayla paylaştırır: kuyruğun önündeki süreç en fazla **q** birim çalışır, bitmediyse kuyruğun arkasına döner.

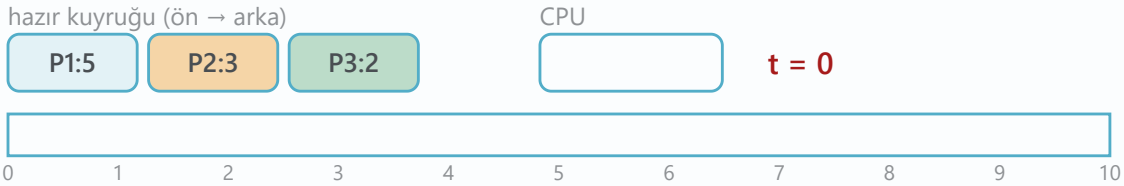


Tümü bitti. Tamamlanma zamanları: P3 = 6, P2 = 9, P1 = 10.

Adım 7 / 7

Simülatör: Round Robin Çizelgeleyici (ayarlanabilir quantum ve süreçler).

Kod İzleme: `roundRobin(q, 2)`



`roundRobin(q, 2)` : kalan süreleri 5, 3, 2 olan üç süreç (Surec: ad, kalan). **t = 0**.

Adım 1 / 20

```
1  void roundRobin(Queue<Surec> q, int dilim) {
▶    int t = 0;
3    while (!q.isEmpty()) {
4      Surec p = q.poll();
5      int calis = Math.min(dilim, p.kalan);
6      t += calis;
7      p.kalan -= calis;
8      if (p.kalan > 0) q.offer(p);
9      else System.out.println(p.ad + " t=" + t);
10   }
11 }
```

DEĞİŞKENLER

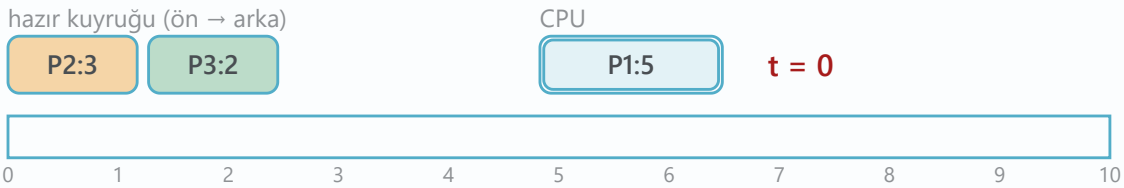
`dilim = 2` `t = 0`

`q = [P1:5, P2:3, P3:2]` `p = –`

`calis = –`

ÇIKTI

Kod İzleme: `roundRobin(q, 2)`



Kuyruk boş değil → **p = q.poll()** = P1 (kalan 5); önden çıkar.

Adım 2 / 20

```
1  void roundRobin(Queue<Surec> q, int dilim) {
2    int t = 0;
▶    while (!q.isEmpty()) {
▶      Surec p = q.poll();
5      int calis = Math.min(dilim, p.kalan);
6      t += calis;
7      p.kalan -= calis;
8      if (p.kalan > 0) q.offer(p);
9      else System.out.println(p.ad + " t=" + t);
10   }
11 }
```

DEĞİŞKENLER

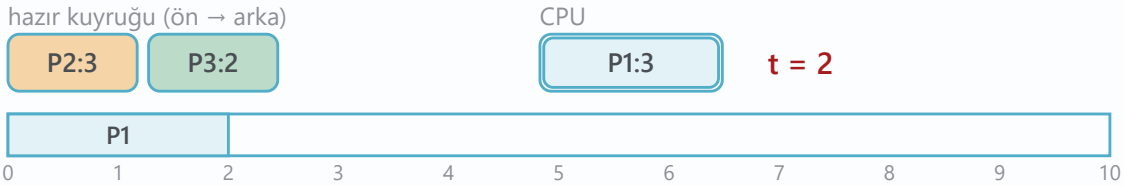
`dilim = 2` `t = 0`

`q = [P2:3, P3:2]` `p = P1:5`

`calis = –`

ÇIKTI

Kod İzleme: `roundRobin(q, 2)`



calis = min(2, 5) = 2; P1 2 birim çalışır → t = 2, kalan 3.

Adım 3 / 20

```
1  void roundRobin(Queue<Surec> q, int dilim) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Surec p = q.poll();
▶      int calis = Math.min(dilim, p.kalan);
▶      t += calis;
▶      p.kalan -= calis;
8      if (p.kalan > 0) q.offer(p);
9      else System.out.println(p.ad + " t=" + t);
10   }
11 }
```

DEĞİŞKENLER

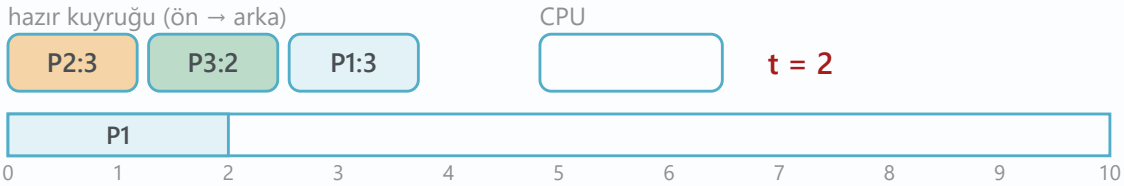
`dilim = 2` `t = 2`

`q = [P2:3, P3:2]` `p = P1:3`

`calis = 2`

ÇIKTI

Kod İzleme: `roundRobin(q, 2)`



kalan 3 > 0 → `q.offer(p)` : P1 kuyruğun **arkasına** döner.

Adım 4 / 20

```
1  void roundRobin(Queue<Surec> q, int dilim) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Surec p = q.poll();
5      int calis = Math.min(dilim, p.kalan);
6      t += calis;
7      p.kalan -= calis;
▶      if (p.kalan > 0) q.offer(p);
9      else System.out.println(p.ad + " t=" + t);
10   }
11 }
```

DEĞİŞKENLER

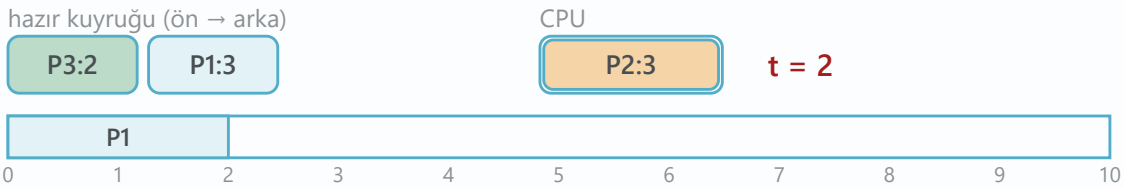
`dilim = 2` `t = 2`

`q = [P2:3, P3:2, P1:3]` `p = P1:3`

`calis = 2`

ÇIKTI

Kod İzleme: `roundRobin(q, 2)`



Kuyruk boş değil → **p = q.poll()** = P2 (kalan 3); önden çıkar.

Adım 5 / 20

```
1  void roundRobin(Queue<Surec> q, int dilim) {
2    int t = 0;
▶    while (!q.isEmpty()) {
▶      Surec p = q.poll();
5      int calis = Math.min(dilim, p.kalan);
6      t += calis;
7      p.kalan -= calis;
8      if (p.kalan > 0) q.offer(p);
9      else System.out.println(p.ad + " t=" + t);
10   }
11 }
```

DEĞİŞKENLER

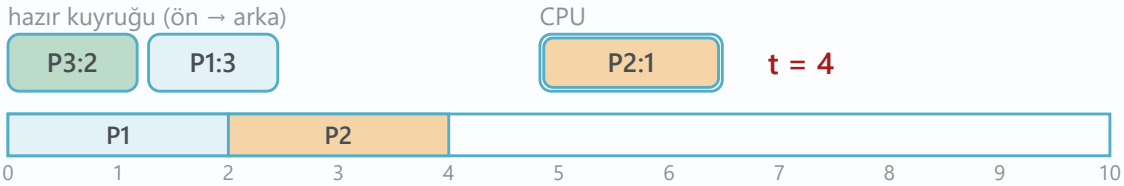
`dilim = 2` `t = 2`

`q = [P3:2, P1:3]` `p = P2:3`

`calis = –`

ÇIKTI

Kod İzleme: `roundRobin(q, 2)`



calis = min(2, 3) = 2; P2 2 birim çalışır → t = 4, kalan 1.

Adım 6 / 20

```
1  void roundRobin(Queue<Surec> q, int dilim) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Surec p = q.poll();
▶      int calis = Math.min(dilim, p.kalan);
▶      t += calis;
▶      p.kalan -= calis;
8      if (p.kalan > 0) q.offer(p);
9      else System.out.println(p.ad + " t=" + t);
10   }
11 }
```

DEĞİŞKENLER

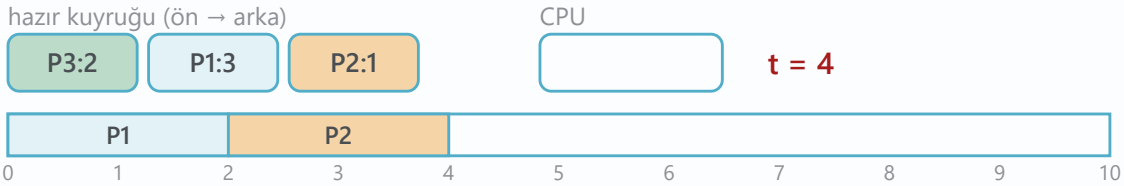
`dilim = 2` `t = 4`

`q = [P3:2, P1:3]` `p = P2:1`

`calis = 2`

ÇIKTI

Kod İzleme: `roundRobin(q, 2)`



kalan 1 > 0 → `q.offer(p)` : P2 kuyruğun **arkasına** döner.

Adım 7 / 20

```
1  void roundRobin(Queue<Surec> q, int dilim) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Surec p = q.poll();
5      int calis = Math.min(dilim, p.kalan);
6      t += calis;
7      p.kalan -= calis;
▶      if (p.kalan > 0) q.offer(p);
9      else System.out.println(p.ad + " t=" + t);
10   }
11 }
```

DEĞİŞKENLER

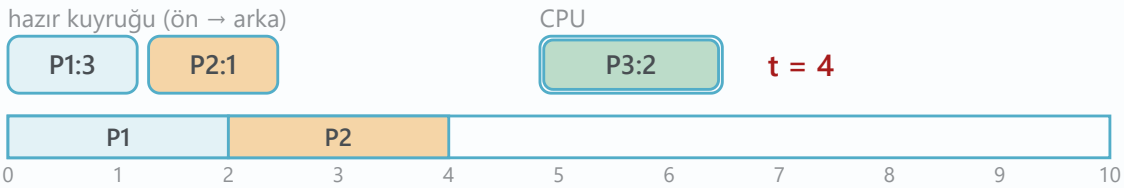
`dilim = 2` `t = 4`

`q = [P3:2, P1:3, P2:1]` `p = P2:1`

`calis = 2`

ÇIKTI

Kod İzleme: `roundRobin(q, 2)`



Kuyruk boş değil → **p = q.poll()** = P3 (kalan 2); önden çıkar.

Adım 8 / 20

```
1  void roundRobin(Queue<Surec> q, int dilim) {
2    int t = 0;
▶    while (!q.isEmpty()) {
▶      Surec p = q.poll();
5      int calis = Math.min(dilim, p.kalan);
6      t += calis;
7      p.kalan -= calis;
8      if (p.kalan > 0) q.offer(p);
9      else System.out.println(p.ad + " t=" + t);
10   }
11 }
```

DEĞİŞKENLER

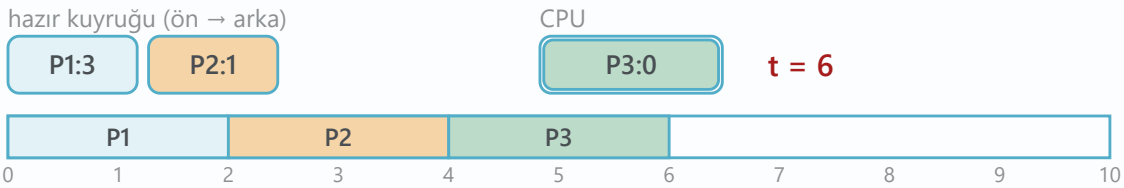
`dilim = 2` `t = 4`

`q = [P1:3, P2:1]` `p = P3:2`

`calis = –`

ÇIKTI

Kod İzleme: `roundRobin(q, 2)`



calis = min(2, 2) = 2; P3 2 birim çalışır → t = 6, kalan 0.

Adım 9 / 20

```
1  void roundRobin(Queue<Surec> q, int dilim) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Surec p = q.poll();
▶      int calis = Math.min(dilim, p.kalan);
▶      t += calis;
▶      p.kalan -= calis;
8      if (p.kalan > 0) q.offer(p);
9      else System.out.println(p.ad + " t=" + t);
10   }
11 }
```

DEĞİŞKENLER

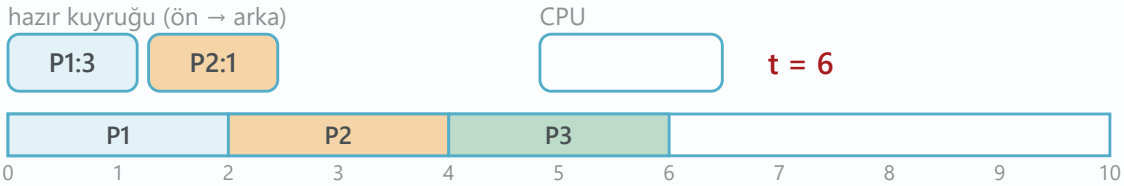
`dilim = 2` `t = 6`

`q = [P1:3, P2:1]` `p = P3:0`

`calis = 2`

ÇIKTI

Kod İzleme: `roundRobin(q, 2)`



kalan 0 → P3 **bitti** (t = 6); kuyruğa dönmez.

Adım 10 / 20

```
1  void roundRobin(Queue<Surec> q, int dilim) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Surec p = q.poll();
5      int calis = Math.min(dilim, p.kalan);
6      t += calis;
7      p.kalan -= calis;
8      if (p.kalan > 0) q.offer(p);
▶      else System.out.println(p.ad + " t=" + t);
10   }
11 }
```

DEĞİŞKENLER

`dilim = 2` `t = 6`

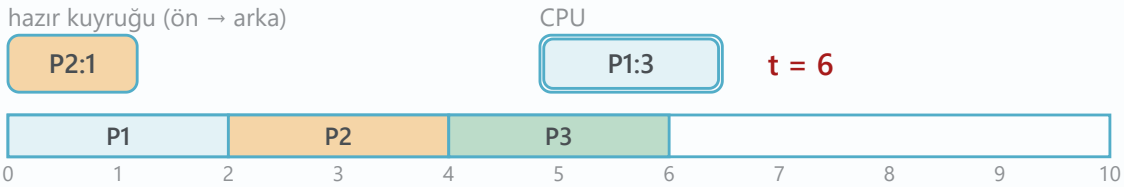
`q = [P1:3, P2:1]` `p = P3:0`

`calis = 2`

ÇIKTI

```
P3 t=6
```

Kod İzleme: `roundRobin(q, 2)`



Kuyruk boş değil → **p = q.poll()** = P1 (kalan 3); önden çıkar.

Adım 11 / 20

```
1  void roundRobin(Queue<Surec> q, int dilim) {
2    int t = 0;
▶    while (!q.isEmpty()) {
▶      Surec p = q.poll();
5      int calis = Math.min(dilim, p.kalan);
6      t += calis;
7      p.kalan -= calis;
8      if (p.kalan > 0) q.offer(p);
9      else System.out.println(p.ad + " t=" + t);
10   }
11 }
```

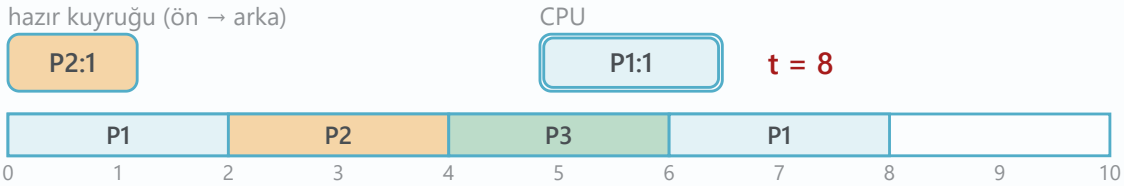
DEĞİŞKENLER

`dilim = 2` `t = 6` `q = [P2:1]` `p = P1:3` `calis = –`

ÇIKTI

```
P3 t=6
```

Kod İzleme: `roundRobin(q, 2)`



calis = min(2, 3) = 2; P1 2 birim çalışır → t = 8, kalan 1.

Adım 12 / 20

```
1  void roundRobin(Queue<Surec> q, int dilim) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Surec p = q.poll();
▶      int calis = Math.min(dilim, p.kalan);
▶      t += calis;
▶      p.kalan -= calis;
8      if (p.kalan > 0) q.offer(p);
9      else System.out.println(p.ad + " t=" + t);
10   }
11 }
```

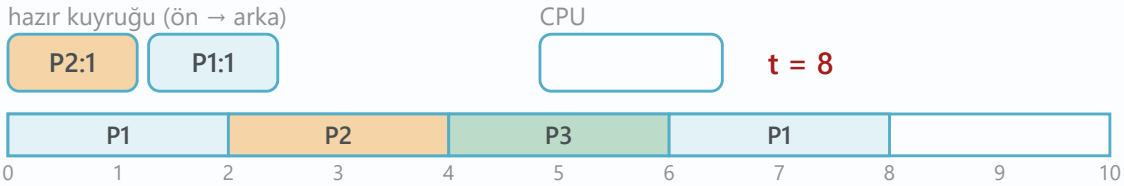
DEĞİŞKENLER

`dilim = 2` `t = 8` `q = [P2:1]` `p = P1:1` `calis = 2`

ÇIKTI

```
P3 t=6
```

Kod İzleme: `roundRobin(q, 2)`



kalan 1 > 0 → `q.offer(p)` : P1 kuyruğun **arkasına** döner.

Adım 13 / 20

```
1  void roundRobin(Queue<Surec> q, int dilim) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Surec p = q.poll();
5      int calis = Math.min(dilim, p.kalan);
6      t += calis;
7      p.kalan -= calis;
▶      if (p.kalan > 0) q.offer(p);
9      else System.out.println(p.ad + " t=" + t);
10   }
11 }
```

DEĞİŞKENLER

`dilim = 2` `t = 8`

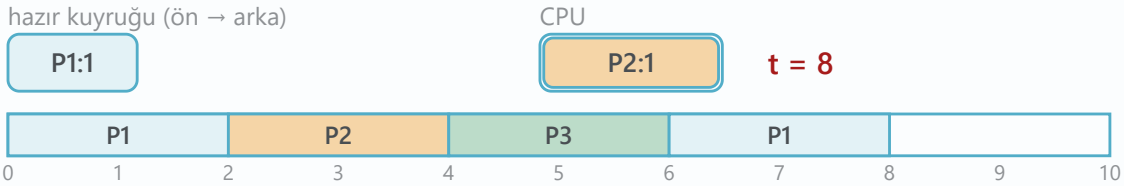
`q = [P2:1, P1:1]` `p = P1:1`

`calis = 2`

ÇIKTI

```
P3 t=6
```

Kod İzleme: `roundRobin(q, 2)`



Kuyruk boş değil → **p = q.poll()** = P2 (kalan 1); önden çıkar.

Adım 14 / 20

```
1  void roundRobin(Queue<Surec> q, int dilim) {
2    int t = 0;
▶    while (!q.isEmpty()) {
▶      Surec p = q.poll();
5      int calis = Math.min(dilim, p.kalan);
6      t += calis;
7      p.kalan -= calis;
8      if (p.kalan > 0) q.offer(p);
9      else System.out.println(p.ad + " t=" + t);
10   }
11 }
```

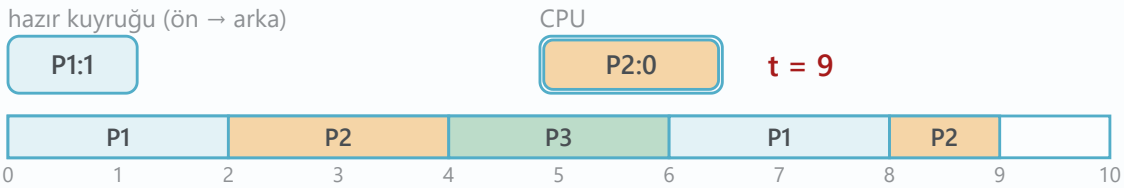
DEĞİŞKENLER

`dilim = 2` `t = 8` `q = [P1:1]` `p = P2:1` `calis = –`

ÇIKTI

```
P3 t=6
```

Kod İzleme: `roundRobin(q, 2)`



calis = min(2, 1) = 1; P2 1 birim çalışır → t = 9, kalan 0. Dilimin tamamı kullanılmadı.

Adım 15 / 20

```
1  void roundRobin(Queue<Surec> q, int dilim) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Surec p = q.poll();
▶      int calis = Math.min(dilim, p.kalan);
▶      t += calis;
▶      p.kalan -= calis;
8      if (p.kalan > 0) q.offer(p);
9      else System.out.println(p.ad + " t=" + t);
10   }
11 }
```

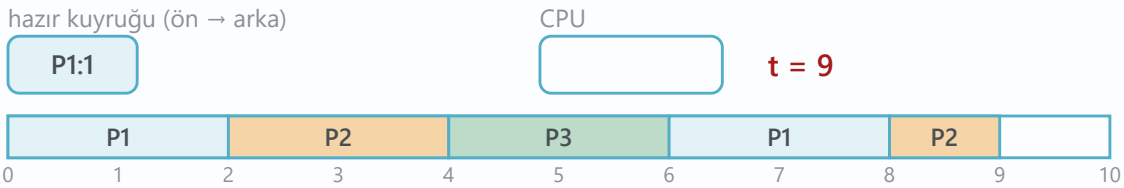
DEĞİŞKENLER

`dilim = 2` `t = 9` `q = [P1:1]` `p = P2:0` `calis = 1`

ÇIKTI

```
P3 t=6
```

Kod İzleme: `roundRobin(q, 2)`



kalan 0 → P2 **bitti** (t = 9); kuyruğa dönmez.

Adım 16 / 20

```
1  void roundRobin(Queue<Surec> q, int dilim) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Surec p = q.poll();
5      int calis = Math.min(dilim, p.kalan);
6      t += calis;
7      p.kalan -= calis;
8      if (p.kalan > 0) q.offer(p);
▶      else System.out.println(p.ad + " t=" + t);
10   }
11 }
```

DEĞİŞKENLER

`dilim = 2` `t = 9` `q = [P1:1]` `p = P2:0` `calis = 1`

ÇIKTI

```
P3 t=6
P2 t=9
```

Kod İzleme: `roundRobin(q, 2)`

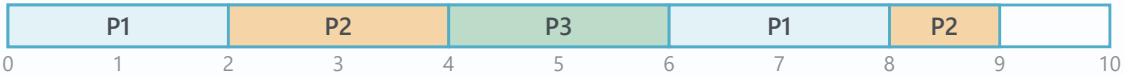
hazır kuyruğu (ön → arka)

boş

CPU

P1:1

t = 9



Kuyruk boş değil → **p = q.poll()** = P1 (kalan 1); önden çıkar.

Adım 17 / 20

```
1  void roundRobin(Queue<Surec> q, int dilim) {
2    int t = 0;
▶    while (!q.isEmpty()) {
▶      Surec p = q.poll();
5      int calis = Math.min(dilim, p.kalan);
6      t += calis;
7      p.kalan -= calis;
8      if (p.kalan > 0) q.offer(p);
9      else System.out.println(p.ad + " t=" + t);
10   }
11 }
```

DEĞİŞKENLER

`dilim = 2` `t = 9` `q = []`

`p = P1:1` `calis = –`

ÇIKTI

```
P3 t=6
P2 t=9
```

Kod İzleme: `roundRobin(q, 2)`

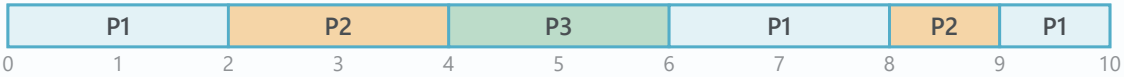
hazır kuyruğu (ön → arka)

boş

CPU

P1:0

t = 10



calis = min(2, 1) = 1; P1 1 birim çalışır → t = 10, kalan 0. Dilimin tamamı kullanılmadı.

Adım 18 / 20

```
1  void roundRobin(Queue<Surec> q, int dilim) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Surec p = q.poll();
▶      int calis = Math.min(dilim, p.kalan);
▶      t += calis;
▶      p.kalan -= calis;
8      if (p.kalan > 0) q.offer(p);
9      else System.out.println(p.ad + " t=" + t);
10   }
11 }
```

DEĞİŞKENLER

`dilim = 2` `t = 10` `q = []` `p = P1:0` `calis = 1`

ÇIKTI

```
P3 t=6
P2 t=9
```

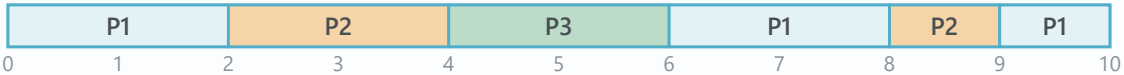
Kod İzleme: `roundRobin(q, 2)`

hazır kuyruğu (ön → arka)

boş

CPU

t = 10



kalan 0 → P1 **bitti** (t = 10); kuyruğa dönmez.

Adım 19 / 20

```
1  void roundRobin(Queue<Surec> q, int dilim) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Surec p = q.poll();
5      int calis = Math.min(dilim, p.kalan);
6      t += calis;
7      p.kalan -= calis;
8      if (p.kalan > 0) q.offer(p);
▶      else System.out.println(p.ad + " t=" + t);
10   }
11 }
```

DEĞİŞKENLER

`dilim = 2` `t = 10` `q = []` `p = P1:0` `calis = 1`

ÇIKTI

```
P3 t=6
P2 t=9
P1 t=10
```

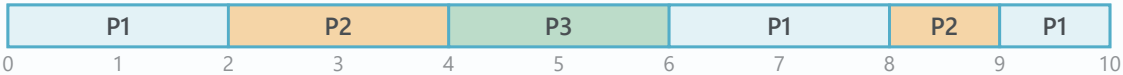
Kod İzleme: `roundRobin(q, 2)`

hazır kuyruğu (ön → arka)

boş

CPU

t = 10



Kuyruk boş → döngü biter. Tamamlanma: P3 = 6, P2 = 9, P1 = 10. Her tur bir poll + en çok bir offer: O(1).

Adım 20 / 20

```
1  void roundRobin(Queue<Surec> q, int dilim) {
2    int t = 0;
▶    while (!q.isEmpty()) {
4      Surec p = q.poll();
5      int calis = Math.min(dilim, p.kalan);
6      t += calis;
7      p.kalan -= calis;
8      if (p.kalan > 0) q.offer(p);
9      else System.out.println(p.ad + " t=" + t);
10   }
11 }
```

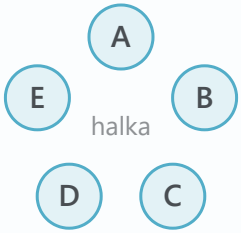
DEĞİŞKENLER

`dilim = 2` `t = 10` `q = []` `p = –` `calis = –`

ÇIKTI

```
P3 t=6
P2 t=9
P1 t=10
```

Kod İzleme: Josephus Problemi (n = 5, k = 3)



kuyruk (ön → arka)



elenenler

—

5 kişi halkada. Her turda öndeki **k − 1 = 2** kişi arkaya geçer, 3. kişi elenir. Kuyruk, halkayı döndürmenin en doğal yoludur.

Adım 1 / 14

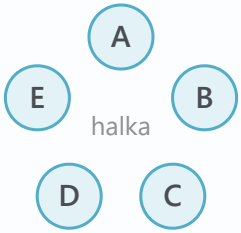
```
1 String josephus(Queue<String> q, int k) {
▶   while (q.size() > 1) {
3     for (int i = 1; i < k; i++)
4       q.offer(q.poll());      // öndekini arkaya
5     String elenen = q.poll();
6     System.out.println(elenen + " elendi");
7   }
8   return q.poll();
9 }
```

DEĞİŞKENLER

`k = 3` `q = [A, B, C, D, E]`

ÇIKTI

Kod İzleme: Josephus Problemi (n = 5, k = 3)



kuyruk (ön → arka)



A öne gelip arkaya taşındı (poll + offer)

elenenler

—

i = 1: `q.offer(q.poll())` → A önden alınıp arkaya eklenir.

Adım 2 / 14

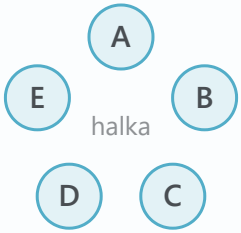
```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
▶     for (int i = 1; i < k; i++)
▶       q.offer(q.poll());      // öndekini arkaya
5     String elenen = q.poll();
6     System.out.println(elenen + " elendi");
7   }
8   return q.poll();
9 }
```

DEĞİŞKENLER

`k = 3` `q = [B, C, D, E, A]` `i = 1`

ÇIKTI

Kod İzleme: Josephus Problemi (n = 5, k = 3)



kuyruk (ön → arka)



B öne gelip arkaya taşındı (poll + offer)

elenenler

—

i = 2: `q.offer(q.poll())` → B önden alınıp arkaya eklenir.

Adım 3 / 14

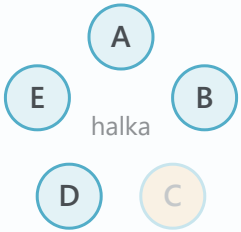
```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
▶     for (int i = 1; i < k; i++)
▶       q.offer(q.poll());      // öndekini arkaya
5     String elenen = q.poll();
6     System.out.println(elenen + " elendi");
7   }
8   return q.poll();
9 }
```

DEĞİŞKENLER

`k = 3` `q = [C, D, E, A, B]` `i = 2`

ÇIKTI

Kod İzleme: Josephus Problemi (n = 5, k = 3)



kuyruk (ön → arka)



C elendi

elenenler

C

Öndeki **C** çıkarılır ve elenir. Kalan: 4 kişi.

Adım 4 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
3     for (int i = 1; i < k; i++)
4       q.offer(q.poll());      // öndekini arkaya
▶     String elenen = q.poll();
▶     System.out.println(elenen + " elendi");
7   }
8   return q.poll();
9 }
```

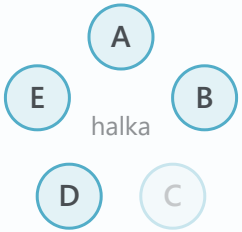
DEĞİŞKENLER

`k = 3` `q = [D, E, A, B]` `elenen = C`

ÇIKTI

```
C elendi
```

Kod İzleme: Josephus Problemi (n = 5, k = 3)



kuyruk (ön → arka)



D öne gelip arkaya taşındı (poll + offer)

elenenler

C

i = 1: `q.offer(q.poll())` → D önden alınıp arkaya eklenir.

Adım 5 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
▶     for (int i = 1; i < k; i++)
▶       q.offer(q.poll());      // öndekini arkaya
5     String elenen = q.poll();
6     System.out.println(elenen + " elendi");
7   }
8   return q.poll();
9 }
```

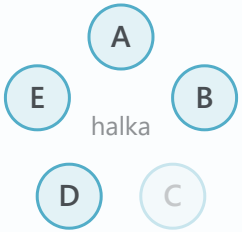
DEĞİŞKENLER

`k = 3` `q = [E, A, B, D]` `i = 1`

ÇIKTI

```
C elendi
```

Kod İzleme: Josephus Problemi (n = 5, k = 3)



kuyruk (ön → arka)

A	B	D	E

E öne gelip arkaya taşındı (poll + offer)

elenenler

C

i = 2: `q.offer(q.poll())` → E önden alınıp arkaya eklenir.

Adım 6 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
▶     for (int i = 1; i < k; i++)
▶       q.offer(q.poll());      // öndekini arkaya
5     String elenen = q.poll();
6     System.out.println(elenen + " elendi");
7   }
8   return q.poll();
9 }
```

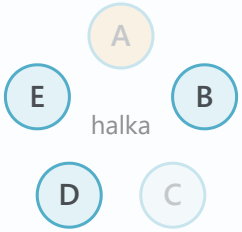
DEĞİŞKENLER

`k = 3` `q = [A, B, D, E]` `i = 2`

ÇIKTI

```
C elendi
```

Kod İzleme: Josephus Problemi (n = 5, k = 3)



kuyruk (ön → arka)

B D E

A elendi

elenenler

C A

Öndeki **A** çıkarılır ve elenir. Kalan: 3 kişi.

Adım 7 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
3     for (int i = 1; i < k; i++)
4       q.offer(q.poll());      // öndekini arkaya
▶     String elenen = q.poll();
▶     System.out.println(elenen + " elendi");
7   }
8   return q.poll();
9 }
```

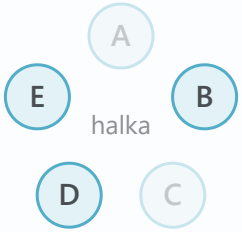
DEĞİŞKENLER

`k = 3` `q = [B, D, E]` `elenen = A`

ÇIKTI

```
C elendi
A elendi
```

Kod İzleme: Josephus Problemi (n = 5, k = 3)



kuyruk (ön → arka)

D E B

B öne gelip arkaya taşındı (poll + offer)

elenenler

C A

i = 1: `q.offer(q.poll())` → B önden alınıp arkaya eklenir.

Adım 8 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
▶     for (int i = 1; i < k; i++)
▶       q.offer(q.poll());      // öndekini arkaya
5     String elenen = q.poll();
6     System.out.println(elenen + " elendi");
7   }
8   return q.poll();
9 }
```

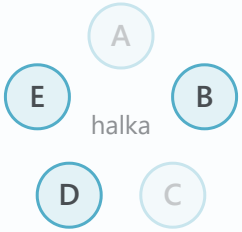
DEĞİŞKENLER

`k = 3` `q = [D, E, B]` `i = 1`

ÇIKTI

```
C elendi
A elendi
```

Kod İzleme: Josephus Problemi (n = 5, k = 3)



kuyruk (ön → arka)

E | B | D

D öne gelip arkaya taşındı (poll + offer)

elenenler

C A

i = 2: `q.offer(q.poll())` → D önden alınıp arkaya eklenir.

Adım 9 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
▶     for (int i = 1; i < k; i++)
▶       q.offer(q.poll());      // öndekini arkaya
5     String elenen = q.poll();
6     System.out.println(elenen + " elendi");
7   }
8   return q.poll();
9 }
```

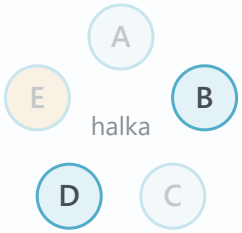
DEĞİŞKENLER

`k = 3` `q = [E, B, D]` `i = 2`

ÇIKTI

```
C elendi
A elendi
```

Kod İzleme: Josephus Problemi (n = 5, k = 3)



kuyruk (ön → arka)

B D

E elendi

elenenler

C A E

Öndeki **E** çıkarılır ve elenir. Kalan: 2 kişi.

Adım 10 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
3     for (int i = 1; i < k; i++)
4       q.offer(q.poll());      // öndekini arkaya
▶     String elenen = q.poll();
▶     System.out.println(elenen + " elendi");
7   }
8   return q.poll();
9 }
```

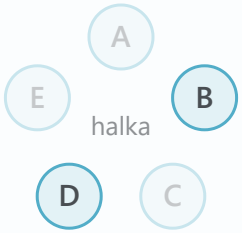
DEĞİŞKENLER

`k = 3` `q = [B, D]` `elenen = E`

ÇIKTI

```
C elendi
A elendi
E elendi
```

Kod İzleme: Josephus Problemi (n = 5, k = 3)



kuyruk (ön → arka)

D B

B öne gelip arkaya taşındı (poll + offer)

elenenler

C A E

i = 1: `q.offer(q.poll())` → B önden alınıp arkaya eklenir.

Adım 11 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
▶     for (int i = 1; i < k; i++)
▶       q.offer(q.poll());      // öndekini arkaya
5     String elenen = q.poll();
6     System.out.println(elenen + " elendi");
7   }
8   return q.poll();
9 }
```

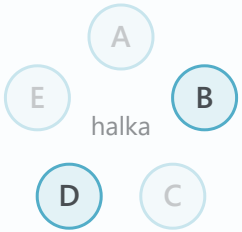
DEĞİŞKENLER

`k = 3` `q = [D, B]` `i = 1`

ÇIKTI

```
C elendi
A elendi
E elendi
```

Kod İzleme: Josephus Problemi (n = 5, k = 3)



kuyruk (ön → arka)

B D

D öne gelip arkaya taşındı (poll + offer)

elenenler

C A E

i = 2: `q.offer(q.poll())` → D önden alınıp arkaya eklenir.

Adım 12 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
▶     for (int i = 1; i < k; i++)
▶       q.offer(q.poll());      // öndekini arkaya
5     String elenen = q.poll();
6     System.out.println(elenen + " elendi");
7   }
8   return q.poll();
9 }
```

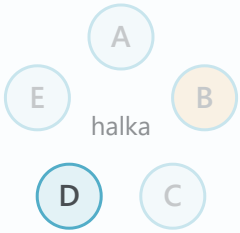
DEĞİŞKENLER

`k = 3` `q = [B, D]` `i = 2`

ÇIKTI

```
C elendi
A elendi
E elendi
```

Kod İzleme: Josephus Problemi (n = 5, k = 3)



kuyruk (ön → arka)

D

B elendi

elenenler

C A E B

Öndeki **B** çıkarılır ve elenir. Kalan: 1 kişi.

Adım 13 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
3     for (int i = 1; i < k; i++)
4       q.offer(q.poll());      // öndekini arkaya
▶     String elenen = q.poll();
▶     System.out.println(elenen + " elendi");
7   }
8   return q.poll();
9 }
```

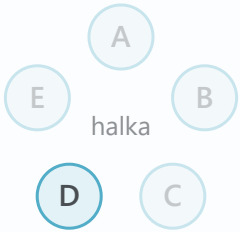
DEĞİŞKENLER

`k = 3` `q = [D]` `elenen = B`

ÇIKTI

```
C elendi
A elendi
E elendi
B elendi
```

Kod İzleme: Josephus Problemi (n = 5, k = 3)



kuyruk (ön → arka)

D

elenenler

C A E B

Tek kişi kaldı: kazanan **D**. n kişi, k adım: O(n·k) kuyruk işlemi.

Adım 14 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
3     for (int i = 1; i < k; i++)
4       q.offer(q.poll());      // öndekini arkaya
5     String elenen = q.poll();
6     System.out.println(elenen + " elendi");
7   }
▶   return q.poll();
9 }
```

DEĞİŞKENLER

`k = 3` `q = [D]`

ÇIKTI

```
C elendi
A elendi
E elendi
B elendi
```

UYGULAMALAR

Örnek: 1'den n'e Kadar İkilik Sayılar

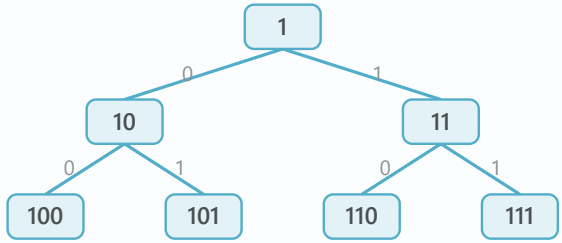
Kuyruk kullanarak 1'den n'e kadar sayıların ikilik (binary) gösterimlerini üretin.

Örnek 1

n = 3 → {"1", "10", "11"}

Örnek 2

n = 5 → {"1", "10", "11", "100", "101"}



Fikir: x'in iki ardılı `x + "0"` (2x) ve `x + "1"` (2x + 1)'dir. Kuyruktan bir sayı alıp sonuca yazar, bu iki sayıyı arkaya eklersek sayılar düzey düzey, yani artan sırada üretilir (genişlik öncelikli gezinme).

Kod İzleme: `ikilikSayiUret(4)`

q (ön →) boş



`sonuc` : 4 boş yuvalı dizi.

Adım 1 / 17

```
1  String[] ikilikSayiUret(int n) {
▶      String[] sonuc = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
6          sonuc[i] = q.poll();
7          q.offer(sonuc[i] + "0");
8          q.offer(sonuc[i] + "1");
9      }
10     return sonuc;
11 }
```

DEĞİŞKENLER

`n = 4` `i = –` `q = –`

ÇAĞRI YIĞINI

`ikilikSayiUret(4)`

`main()`

Kod İzleme: `ikilikSayiUret(4)`

q (ön →) boş



Boş bir kuyruk oluşturulur (LinkedList, Queue arayüzüyle).

Adım 2 / 17

```
1  String[] ikilikSayiUret(int n) {
2      String[] sonuc = new String[n];
▶      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
6          sonuc[i] = q.poll();
7          q.offer(sonuc[i] + "0");
8          q.offer(sonuc[i] + "1");
9      }
10     return sonuc;
11 }
```

DEĞİŞKENLER

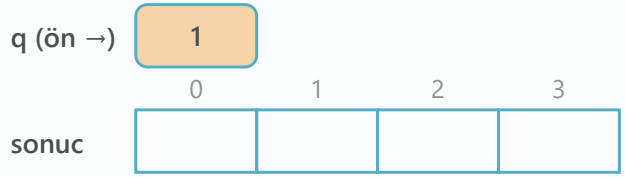
`n = 4` `i = –` `q = []`

ÇAĞRI YIĞINI

`ikilikSayiUret(4)`

`main()`

Kod İzleme: `ikilikSayiUret(4)`



Kök “1” kuyruğa eklenir.

Adım 3 / 17

```
1  String[] ikilikSayiUret(int n) {
2      String[] sonuc = new String[n];
3      Queue<String> q = new LinkedList<>();
▶      q.offer("1");
5      for (int i = 0; i < n; i++) {
6          sonuc[i] = q.poll();
7          q.offer(sonuc[i] + "0");
8          q.offer(sonuc[i] + "1");
9      }
10     return sonuc;
11 }
```

DEĞİŞKENLER

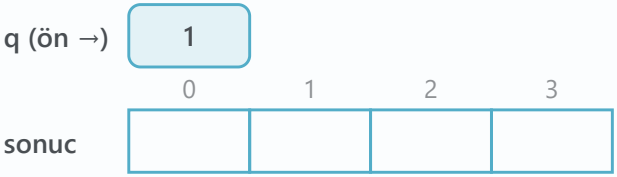
`n = 4` `i = –` `q = [1]`

ÇAĞRI YIĞINI

`ikilikSayiUret(4)`

`main()`

Kod İzleme: `ikilikSayiUret(4)`



i = 0 < 4 → döngü gövdesi.

Adım 4 / 17

```
1  String[] ikilikSayiUret(int n) {
2      String[] sonuc = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
▶      for (int i = 0; i < n; i++) {
6          sonuc[i] = q.poll();
7          q.offer(sonuc[i] + "0");
8          q.offer(sonuc[i] + "1");
9      }
10     return sonuc;
11 }
```

DEĞİŞKENLER

`n = 4` `i = 0` `q = [1]`

ÇAĞRI YIĞINI

- `ikilikSayiUret(4)`
- `main()`

Kod İzleme: `ikilikSayiUret(4)`

q (ön →) boş



sonuc[0] = q.poll() = "1": kuyruğun önündeki en küçük sayı.

Adım 5 / 17

```
1  String[] ikilikSayiUret(int n) {
2      String[] sonuc = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
▶          sonuc[i] = q.poll();
7          q.offer(sonuc[i] + "0");
8          q.offer(sonuc[i] + "1");
9      }
10     return sonuc;
11 }
```

DEĞİŞKENLER

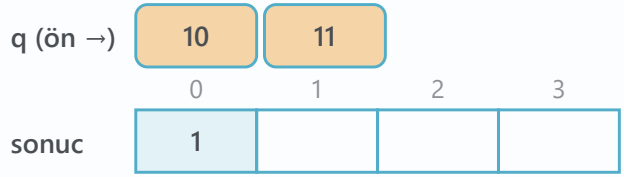
`n = 4` `i = 0` `q = []`

ÇAĞRI YIĞINI

`ikilikSayiUret(4)`

`main()`

Kod İzleme: `ikilikSayiUret(4)`



“10” ve “11” arkaya eklenir: “1” dizgisinin ardılları, sıraları gelince çıkacak.

Adım 6 / 17

```
1  String[] ikilikSayiUret(int n) {
2      String[] sonuc = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
6          sonuc[i] = q.poll();
▶          q.offer(sonuc[i] + "0");
▶          q.offer(sonuc[i] + "1");
9      }
10     return sonuc;
11 }
```

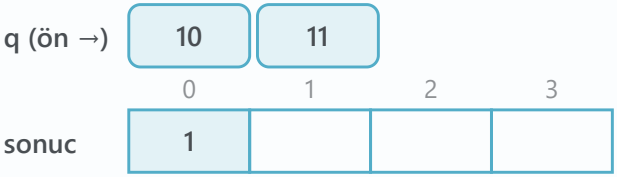
DEĞİŞKENLER

`n = 4` `i = 0` `q = [10, 11]`

ÇAĞRI YIĞINI

- `ikilikSayiUret(4)`
- `main()`

Kod İzleme: `ikilikSayiUret(4)`



i = 1 < 4 → döngü gövdesi.

Adım 7 / 17

```
1  String[] ikilikSayiUret(int n) {
2      String[] sonuc = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
▶      for (int i = 0; i < n; i++) {
6          sonuc[i] = q.poll();
7          q.offer(sonuc[i] + "0");
8          q.offer(sonuc[i] + "1");
9      }
10     return sonuc;
11 }
```

DEĞİŞKENLER

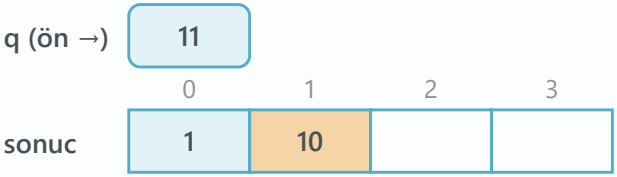
`n = 4` `i = 1` `q = [10, 11]`

ÇAĞRI YIĞINI

`ikilikSayiUret(4)`

`main()`

Kod İzleme: `ikilikSayiUret(4)`



sonuc[1] = q.poll() = "10": kuyruğun önündeki en küçük sayı.

Adım 8 / 17

```
1  String[] ikilikSayiUret(int n) {
2      String[] sonuc = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
▶          sonuc[i] = q.poll();
7          q.offer(sonuc[i] + "0");
8          q.offer(sonuc[i] + "1");
9      }
10     return sonuc;
11 }
```

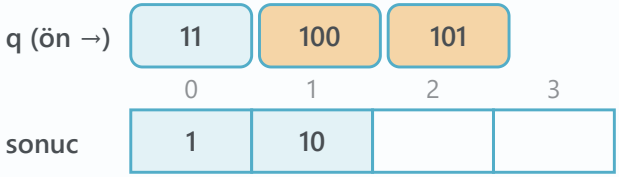
DEĞİŞKENLER

`n = 4` `i = 1` `q = [11]`

ÇAĞRI YIĞINI

- `ikilikSayiUret(4)`
- `main()`

Kod İzleme: `ikilikSayiUret(4)`



“100” ve “101” arkaya eklenir: “10” dizgisinin ardılları, sıraları gelince çıkacak.

Adım 9 / 17

```
1  String[] ikilikSayiUret(int n) {
2      String[] sonuc = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
6          sonuc[i] = q.poll();
▶          q.offer(sonuc[i] + "0");
▶          q.offer(sonuc[i] + "1");
9      }
10     return sonuc;
11 }
```

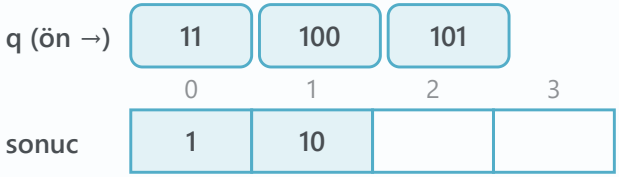
DEĞİŞKENLER

`n = 4` `i = 1` `q = [11, 100, 101]`

ÇAĞRI YIĞINI

- `ikilikSayiUret(4)`
- `main()`

Kod İzleme: `ikilikSayiUret(4)`



i = 2 < 4 → döngü gövdesi.

Adım 10 / 17

```
1  String[] ikilikSayiUret(int n) {
2      String[] sonuc = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
▶      for (int i = 0; i < n; i++) {
6          sonuc[i] = q.poll();
7          q.offer(sonuc[i] + "0");
8          q.offer(sonuc[i] + "1");
9      }
10     return sonuc;
11 }
```

DEĞİŞKENLER

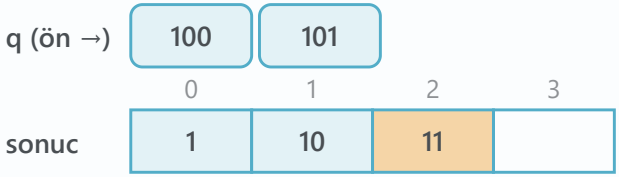
`n = 4` `i = 2` `q = [11, 100, 101]`

ÇAĞRI YIĞINI

`ikilikSayiUret(4)`

`main()`

Kod İzleme: `ikilikSayiUret(4)`



sonuc[2] = q.poll() = "11": kuyruğun önündeki en küçük sayı.

Adım 11 / 17

```
1  String[] ikilikSayiUret(int n) {
2      String[] sonuc = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
▶          sonuc[i] = q.poll();
7          q.offer(sonuc[i] + "0");
8          q.offer(sonuc[i] + "1");
9      }
10     return sonuc;
11 }
```

DEĞİŞKENLER

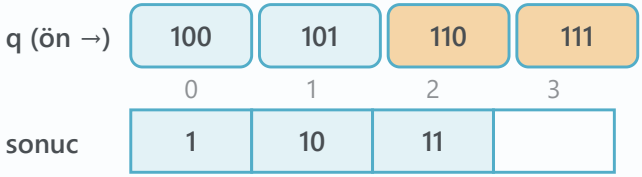
`n = 4` `i = 2` `q = [100, 101]`

ÇAĞRI YIĞINI

`ikilikSayiUret(4)`

`main()`

Kod İzleme: `ikilikSayiUret(4)`



“110” ve “111” arkaya eklenir: “11” dizgisinin ardılları, sıraları gelince çıkacak.

Adım 12 / 17

```
1  String[] ikilikSayiUret(int n) {
2      String[] sonuc = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
6          sonuc[i] = q.poll();
▶          q.offer(sonuc[i] + "0");
▶          q.offer(sonuc[i] + "1");
9      }
10     return sonuc;
11 }
```

DEĞİŞKENLER

`n = 4` `i = 2`

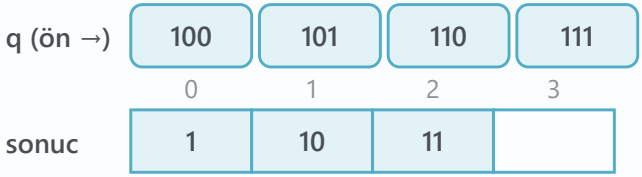
`q = [100, 101, 110, 111]`

ÇAĞRI YIĞINI

`ikilikSayiUret(4)`

`main()`

Kod İzleme: `ikilikSayiUret(4)`



i = 3 < 4 → döngü gövdesi.

Adım 13 / 17

```
1  String[] ikilikSayiUret(int n) {
2      String[] sonuc = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
▶      for (int i = 0; i < n; i++) {
6          sonuc[i] = q.poll();
7          q.offer(sonuc[i] + "0");
8          q.offer(sonuc[i] + "1");
9      }
10     return sonuc;
11 }
```

DEĞİŞKENLER

`n = 4` `i = 3`

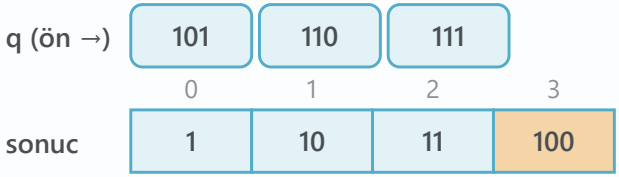
`q = [100, 101, 110, 111]`

ÇAĞRI YIĞINI

`ikilikSayiUret(4)`

`main()`

Kod İzleme: `ikilikSayiUret(4)`



sonuc[3] = q.poll() = "100": kuyruğun önündeki en küçük sayı.

Adım 14 / 17

```
1  String[] ikilikSayiUret(int n) {
2      String[] sonuc = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
▶          sonuc[i] = q.poll();
7          q.offer(sonuc[i] + "0");
8          q.offer(sonuc[i] + "1");
9      }
10     return sonuc;
11 }
```

DEĞİŞKENLER

`n = 4` `i = 3` `q = [101, 110, 111]`

ÇAĞRI YIĞINI

`ikilikSayiUret(4)`

`main()`

Kod İzleme: `ikilikSayiUret(4)`



“1000” ve “1001” arkaya eklenir: “100” dizgisinin ardılları, sıraları gelince çıkacak.

Adım 15 / 17

```
1  String[] ikilikSayiUret(int n) {
2      String[] sonuc = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
6          sonuc[i] = q.poll();
▶          q.offer(sonuc[i] + "0");
▶          q.offer(sonuc[i] + "1");
9      }
10     return sonuc;
11 }
```

DEĞİŞKENLER

`n = 4` `i = 3`

`q = [101, 110, 111, 1000, 1001]`

ÇAĞRI YIĞINI

`ikilikSayiUret(4)`

`main()`

Kod İzleme: `ikilikSayiUret(4)`



i = 4 < 4 değil → döngü biter.

Adım 16 / 17

```
1  String[] ikilikSayiUret(int n) {
2      String[] sonuc = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
▶      for (int i = 0; i < n; i++) {
6          sonuc[i] = q.poll();
7          q.offer(sonuc[i] + "0");
8          q.offer(sonuc[i] + "1");
9      }
10     return sonuc;
11 }
```

DEĞİŞKENLER

`n = 4` `i = 4`

`q = [101, 110, 111, 1000, 1001]`

ÇAĞRI YIĞINI

`ikilikSayiUret(4)`

`main()`

Kod İzleme: `ikilikSayiUret(4)`



return {“1”, “10”, “11”, “100”}. n poll + 2n offer → O(n) kuyruk işlemi (dizgi uzunluğu O(log n) olduğundan toplam karakter işi O(n log n)).

Adım 17 / 17

```
1  String[] ikilikSayiUret(int n) {
2      String[] sonuc = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
6          sonuc[i] = q.poll();
7          q.offer(sonuc[i] + "0");
8          q.offer(sonuc[i] + "1");
9      }
▶      return sonuc;
11 }
```

DEĞİŞKENLER

`n = 4` `i = –`

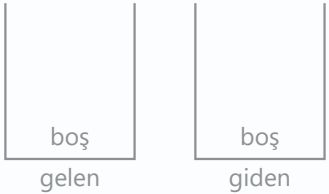
`q = [101, 110, 111, 1000, 1001]`

ÇAĞRI YIĞINI

`ikilikSayiUret(4)`

`main()`

Kod İzleme: İki Yığınla Kuyruk



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

çıkış sırası: —

gelen eklemeler, **giden** çıkarmalar için (yığınların üstü yukarıda).

Adım 1 / 17

```
▶ Deque<Integer> gelen = new ArrayDeque<>();
▶ Deque<Integer> giden = new ArrayDeque<>();
3  void enqueue(int x) {
4    gelen.push(x);
5  }
6  int dequeue() {
7    if (giden.isEmpty())
8      while (!gelen.isEmpty())
9        giden.push(gelen.pop());
10   return giden.pop();
11 }
```

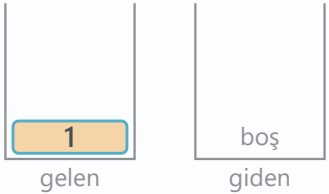
DEĞİŞKENLER

gelen = [] giden = []

ÇAĞRI YIĞINI

main()

Kod İzleme: İki Yığınla Kuyruk



enqueue(1) `gelen.push(1)` → O(1).

enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

çıkış sırası: —

Adım 2 / 17

```
1  Deque<Integer> gelen = new ArrayDeque<>();
2  Deque<Integer> giden = new ArrayDeque<>();
3  void enqueue(int x) {
▶    gelen.push(x);
5  }
6  int dequeue() {
7    if (giden.isEmpty())
8      while (!gelen.isEmpty())
9        giden.push(gelen.pop());
10   return giden.pop();
11 }
```

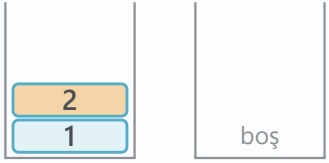
DEĞİŞKENLER

`gelen = [1]` `giden = []` `x = 1`

ÇAĞRI YIĞINI

- **enqueue(1)**
- main()

Kod İzleme: İki Yığınla Kuyruk



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

çıkış sırası: —

enqueue(2): `gelen.push(2)` → O(1).

Adım 3 / 17

```
1  Deque<Integer> gelen = new ArrayDeque<>();
2  Deque<Integer> giden = new ArrayDeque<>();
3  void enqueue(int x) {
▶    gelen.push(x);
5  }
6  int dequeue() {
7    if (giden.isEmpty())
8      while (!gelen.isEmpty())
9        giden.push(gelen.pop());
10   return giden.pop();
11 }
```

DEĞİŞKENLER

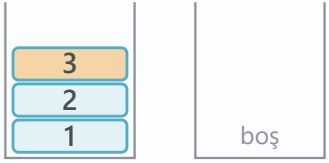
`gelen = [1, 2]` `giden = []` `x = 2`

ÇAĞRI YIĞINI

`enqueue(2)`

`main()`

Kod İzleme: İki Yığınla Kuyruk



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

çıkış sırası: —

enqueue(3): `gelen.push(3)` → O(1).

Adım 4 / 17

```
1  Deque<Integer> gelen = new ArrayDeque<>();
2  Deque<Integer> giden = new ArrayDeque<>();
3  void enqueue(int x) {
▶    gelen.push(x);
5  }
6  int dequeue() {
7    if (giden.isEmpty())
8      while (!gelen.isEmpty())
9        giden.push(gelen.pop());
10   return giden.pop();
11 }
```

DEĞİŞKENLER

`gelen = [1, 2, 3]` `giden = []` `x = 3`

ÇAĞRI YIĞINI

- **enqueue(3)**
- main()

Kod İzleme: İki Yığınla Kuyruk



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

çıkış sırası: —

dequeue: giden **boş** → gelen'deki her şey giden'e aktarılacak.

Adım 5 / 17

```
1  Deque<Integer> gelen = new ArrayDeque<>();
2  Deque<Integer> giden = new ArrayDeque<>();
3  void enqueue(int x) {
4    gelen.push(x);
5  }
6  int dequeue() {
▶    if (giden.isEmpty())
8      while (!gelen.isEmpty())
9        giden.push(gelen.pop());
10   return giden.pop();
11 }
```

DEĞİŞKENLER

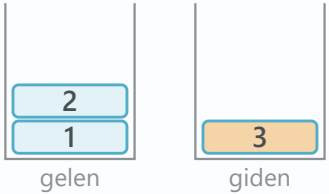
`gelen = [1, 2, 3]` `giden = []`

ÇAĞRI YIĞINI

dequeue()

main()

Kod İzleme: İki Yığınla Kuyruk



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

çıkış sırası: —

3: gelen'den pop, giden'e push. Aktarım sırayı **ters çevirir**; en eski öğe üste çıkar.

Adım 6 / 17

```
1  Deque<Integer> gelen = new ArrayDeque<>();
2  Deque<Integer> giden = new ArrayDeque<>();
3  void enqueue(int x) {
4    gelen.push(x);
5  }
6  int dequeue() {
7    if (giden.isEmpty())
▶      while (!gelen.isEmpty())
▶        giden.push(gelen.pop());
10   return giden.pop();
11 }
```

DEĞİŞKENLER

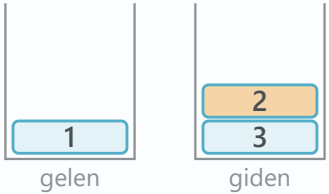
`gelen = [1, 2]` `giden = [3]`

ÇAĞRI YIĞINI

`dequeue()`

`main()`

Kod İzleme: İki Yığınla Kuyruk



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

çıkış sırası: —

2: gelen'den pop, giden'e push. Aktarım sırayı **ters çevirir**; en eski öğe üste çıkar.

Adım 7 / 17

```
1  Deque<Integer> gelen = new ArrayDeque<>();
2  Deque<Integer> giden = new ArrayDeque<>();
3  void enqueue(int x) {
4    gelen.push(x);
5  }
6  int dequeue() {
7    if (giden.isEmpty())
▶      while (!gelen.isEmpty())
▶        giden.push(gelen.pop());
10   return giden.pop();
11 }
```

DEĞİŞKENLER

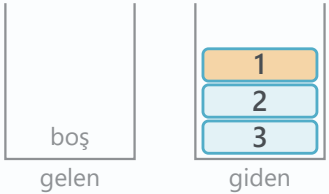
`gelen = [1]` `giden = [3, 2]`

ÇAĞRI YIĞINI

`dequeue()`

`main()`

Kod İzleme: İki Yığınla Kuyruk



enqueue(1) enqueue(2) enqueue(3) **dequeue()**
enqueue(4) dequeue() dequeue() dequeue()

çıkış sırası: —

1: gelen'den pop, giden'e push. Aktarım sırayı **ters çevirir**; en eski öğe üste çıkar.

Adım 8 / 17

```
1  Deque<Integer> gelen = new ArrayDeque<>();
2  Deque<Integer> giden = new ArrayDeque<>();
3  void enqueue(int x) {
4    gelen.push(x);
5  }
6  int dequeue() {
7    if (giden.isEmpty())
▶      while (!gelen.isEmpty())
▶        giden.push(gelen.pop());
10   return giden.pop();
11 }
```

DEĞİŞKENLER

`gelen = []` `giden = [3, 2, 1]`

ÇAĞRI YIĞINI

- **dequeue()**
- main()

Kod İzleme: İki Yığınla Kuyruk



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

çıkış sırası: 1

`giden.pop()` → **1** (FIFO: ilk eklenenlerden).

Adım 9 / 17

```
1  Deque<Integer> gelen = new ArrayDeque<>();
2  Deque<Integer> giden = new ArrayDeque<>();
3  void enqueue(int x) {
4    gelen.push(x);
5  }
6  int dequeue() {
7    if (giden.isEmpty())
8      while (!gelen.isEmpty())
9        giden.push(gelen.pop());
▶    return giden.pop();
11 }
```

DEĞİŞKENLER

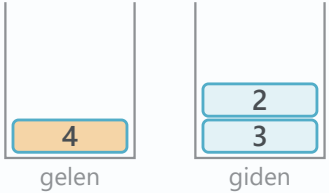
`gelen = []` `giden = [3, 2]`

ÇAĞRI YIĞINI

dequeue()

main()

Kod İzleme: İki Yığınla Kuyruk



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

çıkış sırası: 1

enqueue(4): `gelen.push(4)` → O(1).

Adım 10 / 17

```
1  Deque<Integer> gelen = new ArrayDeque<>();
2  Deque<Integer> giden = new ArrayDeque<>();
3  void enqueue(int x) {
▶    gelen.push(x);
5  }
6  int dequeue() {
7    if (giden.isEmpty())
8      while (!gelen.isEmpty())
9        giden.push(gelen.pop());
10   return giden.pop();
11 }
```

DEĞİŞKENLER

`gelen = [4]` `giden = [3, 2]` `x = 4`

ÇAĞRI YIĞINI

- **enqueue(4)**
- main()

Kod İzleme: İki Yığınla Kuyruk



enqueue(1)	enqueue(2)	enqueue(3)	dequeue()
enqueue(4)	dequeue()	dequeue()	dequeue()

çıkış sırası: 1

dequeue: giden boş değil → aktarım **yok**.

Adım 11 / 17

```
1  Deque<Integer> gelen = new ArrayDeque<>();
2  Deque<Integer> giden = new ArrayDeque<>();
3  void enqueue(int x) {
4    gelen.push(x);
5  }
6  int dequeue() {
▶    if (giden.isEmpty())
8      while (!gelen.isEmpty())
9        giden.push(gelen.pop());
10   return giden.pop();
11 }
```

DEĞİŞKENLER

`gelen = [4]` `giden = [3, 2]`

ÇAĞRI YIĞINI

- **dequeue()**
- main()

Kod İzleme: İki Yığınla Kuyruk



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

çıkış sırası: 1 2

`giden.pop()` → **2** (FIFO: ilk eklenenlerden).

Adım 12 / 17

```
1  Deque<Integer> gelen = new ArrayDeque<>();
2  Deque<Integer> giden = new ArrayDeque<>();
3  void enqueue(int x) {
4    gelen.push(x);
5  }
6  int dequeue() {
7    if (giden.isEmpty())
8      while (!gelen.isEmpty())
9        giden.push(gelen.pop());
▶    return giden.pop();
11 }
```

DEĞİŞKENLER

`gelen = [4]` `giden = [3]`

ÇAĞRI YIĞINI

dequeue()

main()

Kod İzleme: İki Yığınla Kuyruk



enqueue(1)	enqueue(2)	enqueue(3)	dequeue()
enqueue(4)	dequeue()	dequeue()	dequeue()

çıkış sırası: 1 2

dequeue: giden boş değil → aktarım **yok**.

Adım 13 / 17

```
1  Deque<Integer> gelen = new ArrayDeque<>();
2  Deque<Integer> giden = new ArrayDeque<>();
3  void enqueue(int x) {
4    gelen.push(x);
5  }
6  int dequeue() {
▶    if (giden.isEmpty())
8      while (!gelen.isEmpty())
9        giden.push(gelen.pop());
10   return giden.pop();
11 }
```

DEĞİŞKENLER

`gelen = [4]` `giden = [3]`

ÇAĞRI YIĞINI

- **dequeue()**
- main()

Kod İzleme: İki Yığınla Kuyruk



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

çıkış sırası: 1 2 3

`giden.pop()` → **3** (FIFO: ilk eklenenlerden).

Adım 14 / 17

```
1  Deque<Integer> gelen = new ArrayDeque<>();
2  Deque<Integer> giden = new ArrayDeque<>();
3  void enqueue(int x) {
4    gelen.push(x);
5  }
6  int dequeue() {
7    if (giden.isEmpty())
8      while (!gelen.isEmpty())
9        giden.push(gelen.pop());
▶    return giden.pop();
11 }
```

DEĞİŞKENLER

`gelen = [4]` `giden = []`

ÇAĞRI YIĞINI

dequeue()

main()

Kod İzleme: İki Yığınla Kuyruk



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

çıkış sırası: 1 2 3

dequeue: giden **boş** → gelen'deki her şey giden'e aktarılacak.

Adım 15 / 17

```
1  Deque<Integer> gelen = new ArrayDeque<>();
2  Deque<Integer> giden = new ArrayDeque<>();
3  void enqueue(int x) {
4    gelen.push(x);
5  }
6  int dequeue() {
▶    if (giden.isEmpty())
8      while (!gelen.isEmpty())
9        giden.push(gelen.pop());
10   return giden.pop();
11 }
```

DEĞİŞKENLER

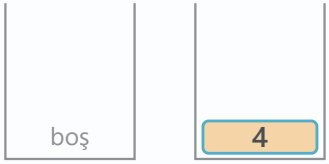
`gelen = [4]` `giden = []`

ÇAĞRI YIĞINI

`dequeue()`

`main()`

Kod İzleme: İki Yığınla Kuyruk



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

çıkış sırası: 1 2 3

4: gelen'den pop, giden'e push. Aktarım sırayı **ters çevirir**; en eski öğe üste çıkar.

Adım 16 / 17

```
1  Deque<Integer> gelen = new ArrayDeque<>();
2  Deque<Integer> giden = new ArrayDeque<>();
3  void enqueue(int x) {
4    gelen.push(x);
5  }
6  int dequeue() {
7    if (giden.isEmpty())
▶      while (!gelen.isEmpty())
▶        giden.push(gelen.pop());
10   return giden.pop();
11 }
```

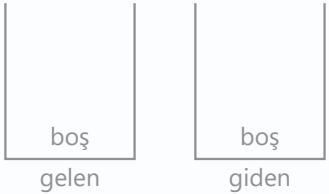
DEĞİŞKENLER

`gelen = []` `giden = [4]`

ÇAĞRI YIĞINI

- **dequeue()**
- main()

Kod İzleme: İki Yığınla Kuyruk



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

çıkış sırası: 1 2 3 4

`giden.pop()` → **4**. Çıkış 1, 2, 3, 4 = giriş sırası. Her öğe en çok bir kez aktarılır → dequeue **amortize O(1)** (tek çağrı en kötü O(n)).

Adım 17 / 17

```
1  Deque<Integer> gelen = new ArrayDeque<>();
2  Deque<Integer> giden = new ArrayDeque<>();
3  void enqueue(int x) {
4    gelen.push(x);
5  }
6  int dequeue() {
7    if (giden.isEmpty())
8      while (!gelen.isEmpty())
9        giden.push(gelen.pop());
▶    return giden.pop();
11 }
```

DEĞİŞKENLER

`gelen = []` `giden = []`

ÇAĞRI YIĞINI

dequeue()

main()

ÖZET

Sık Yapılan Hatalar

Son öğede arka'yı unutmak

Bağlı liste kuyruğunda son düğüm çıkınca `on = null` olur; `arka = null` yapılmazsa silinmiş düğümü gösterir.

Dolu / boş karışıklığı

Yalnızca bas ve arka tutulursa boşta da dolu da `bas == arka` olur; `boyut` sayacı ya da bir boş yuva gerekir.

Büyütmede `Arrays.copyOf`

Başa sarmış dairesel kuyrukta sırayı bozar; öğeler `(bas + k) % N` sırasıyla kopyalanmalı.

add / offer, remove / poll

`add`, `remove`, `element` hata fırlatır; `offer`, `poll`, `peek` false / null döner.

`PriorityQueue` FIFO değildir

`Queue<Integer> q = new PriorityQueue<>()` öğeleri önceliğe göre verir, geliş sırasına göre değil.

`ArrayDeque` ve null

`ArrayDeque` null öğe kabul etmez (`NullPointerException`); null, "boş" sonucuyla karışır.

ÖZET

Özet: Kuyruk Gerçeklemelerinin Maliyetleri

Gerçekleme	Ekleme	Çıkarma	Ön öğe	Not
Basit dizi (kaydırmalı)	O(1)	O(n)	O(1)	her çıkarmada kaydırma
Dairesel dizi	O(1)	O(1)	O(1)	sabit kapasite, `% N`
Bağlı liste	O(1)	O(1)	O(1)	on + arka referansı
Deque (çift yönlü liste)	O(1) iki uç	O(1) iki uç	O(1)	`onceki` referansı
İki yığınla kuyruk	O(1)	O(1) amortize	O(1) amortize	her öğe bir kez aktarılır

FIFO

Arkadan ekle, önden çıkar.

Dairesel dizi

Boş/dolu ayrımı: `boyut` sayacı ya da bir boş yuva.

Uç durumlar

Boşta çıkarma (underflow), son öğe çıkınca `arka = null`.

ÖZET

Etkileşimli Simülatörler

Kuyruk Simülatörü

enqueue, dequeue, front, rear ve arama işlemleri, adım adım.

Deque Simülatörü

Her iki uçtan ekleme ve çıkarmanın etkisi.

Dairesel Kuyruk Simülatörü

Dizi ve halka görünümünde bas, boyut, mod ile başa sarma ve dolu dizinin büyütülmesi.

Round Robin Çizelgeleyici

FIFO hazır kuyruğu, ayarlanabilir quantum ve süreç ölçümleri.

ÖZET

Örnek Problemler

1. Boş kuyrukta: enqueue(4), enqueue(7), dequeue(), enqueue(1), enqueue(9), dequeue(). first() nedir, kuyrukta ne kalır?
2. N = 6, bas = 4, boyut = 3: öğeler hangi indislerde, sonraki enqueue nereye yazılır?
3. “Bir yuva boş” yönteminde N = 10 ise en fazla kaç öğe saklanır? Dolu koşulu nedir?
4. dequeue'dan `if (on == null) arka = null;` silinirse ne olur?
5. İki yığınla kuyruk nasıl yapılır? dequeue'nun amortize maliyeti nedir?
6. Deque ile bir kelimenin palindrom olduğu nasıl kontrol edilir?

▶ **Cevapları göster**

ÖZET

Alıştırmalar

1. Bir kuyruğu yalnızca kuyruk işlemleri ve bir yığınla ters çevirin.
2. Tek bir kuyrukla yığın yapın: push'u nasıl yazarsınız, maliyeti nedir?
3. Josephus: 1..7 numaralı kişiler, k = 3. Eleme sırası ve kazanan?
4. N = 5, bas = 3, boyut = 5 olan dairesel kuyruk büyütülünce yeni dizide öğeler hangi sırada olur?
5. Bir kuyruğun ilk k öğesini, kalanların sırasını bozmadan ters çevirin.
6. Kayan pencere en küçüğü için deque algoritmasında ne değişir?

▶ **İpuçları / cevaplar**

ÖZET

Kaynaklar

- Goodrich, Tamassia ve Goldwasser, *Data Structures and Algorithms in Java*, 6. baskı, 2014.
- Lafore, *Data Structures and Algorithms in Java*, 2. baskı, 2002.
- Oracle, Java SE 17 `Queue`, `Deque`, `ArrayDeque` ve `PriorityQueue` API belgeleri.

Son

Bölüm 8: Kuyruklar ve Deque

Sonraki sunum: Bölüm 9: Ağaçlar · Çalışma soruları · Simülatörler: Kuyruk Simülatörü, Deque Simülatörü, Dairesel Kuyruk Simülatörü, Round Robin Çizelgeleyici · Terim sözlüğü