YZM 2116- VERİ YAPILARI

DERS#4: KUYRUK VERİ YAPISI

İÇERİK

Bu bölümde,

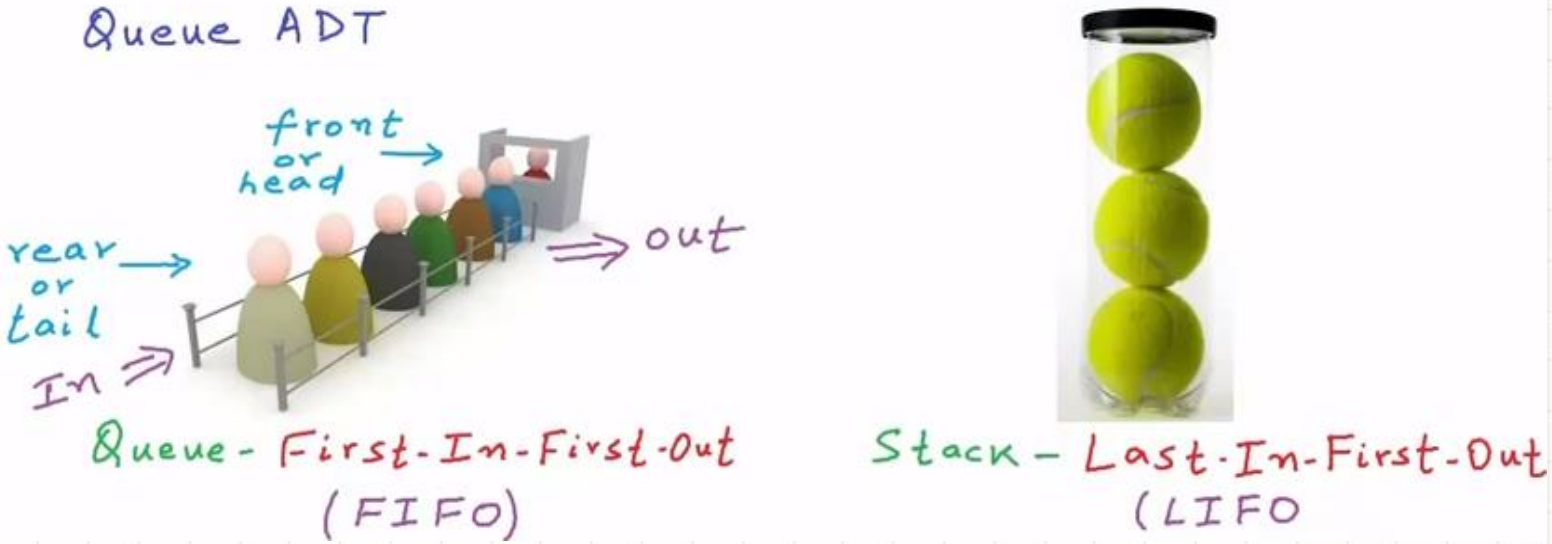
- Kuyruk VY ve ADT
- Basit Kuyruk (Simple Queue)
- Döngüsel Kuyruk (Circular Queue)
- Öncelik Kuyruğu (Priority Queue)

konusuna değinilecektir.

Kuyruk Giriş

- Kuyruk, eleman eklemelerin sondan (**rear**) ve eleman çıkarmaların bastan (**front**) yapıldığı, (First In First Out-İlk Gelen İlk Çıkar – FIFO) olarak modellenen, doğrusal bir veri saklama yapısıdır.
- Bir elemanın kuyruğa girmesi **insert** (literatürde *put*, *add* veya *enqueue* olarak da geçer) işlemi iken listeden silinmesi **remove** (*delete* veya *dequeue*) işlemidir.
- Insert'ler *kuyruğun arkasından yapılırken*, remove'lar *kuyruğun önünden yapılırlar*.
- **Boş bir kuyruktan** eleman silmeye çalışmak ***underflow*** hatası üretirken, **dolu bir kuyruğa** eleman eklemeye çalışmak ***overflow*** hatası üretir.

Kuyruk Giriş (devam…)



- Kuyruk yapısı, yığın yapısına oldukça benzemektedir. İkisinde de eleman ekleme işlemi en sondan yapılmaktadır.
- Aralarındaki fark eleman çıkartmanın yığın yapısında **en sondan**, kuyruk yapısında ise **en baştan** yapılmasıdır.

Kuyruk ADT

```
public interface IQueue
{
        void Insert(object o);
        object Remove();
        object Peek();
        Boolean IsEmpty();
}
```

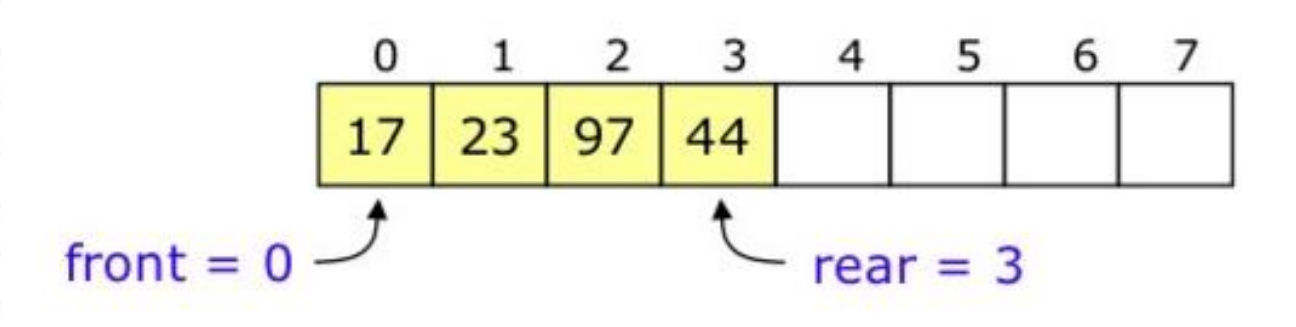
Insert(obj)	Kuyruğun sonuna (**rear**) eleman ekler.
obj Remove()	Kuyruğun önündeki (**front**) ilk elemanı yani işi biten elemanı siler. Kuyruk boşsa hata döner.
obj Peek()	Kuyruğun önündeki (**front**) elemanı geriye döndürür.
bool IsEmpty()	Kuyruk boşsa true değilse false döner.

Kuyruk Dizi Gerçekleştirim

Queue dizi implementasyonu için kurallar

- Queue dizi implemetasyonunda dizimizi **queue[n]** olarak tanımlarsak,
  - ***n*** kuyruktaki maksimum eleman sayısıdır.
- Implementasyonda **front** ve **rear** olmak üzere 2 tane değişken tanımlanır.
  - **front:** kuyruğun önündeki elemanı temsil eder.
    - `front = -1` ise kuyruk boştur.
    - Kuyruktan her eleman çıkartıldığında (REMOVE) front bir artar.
  - **rear:** kuyruğun sonundaki elemanı temsil eder.
    - Kuyruğa her eleman eklendiğinde (INSERT) rear bir artar.

Kuyruk Dizi Gerçekleştirim (devam…)

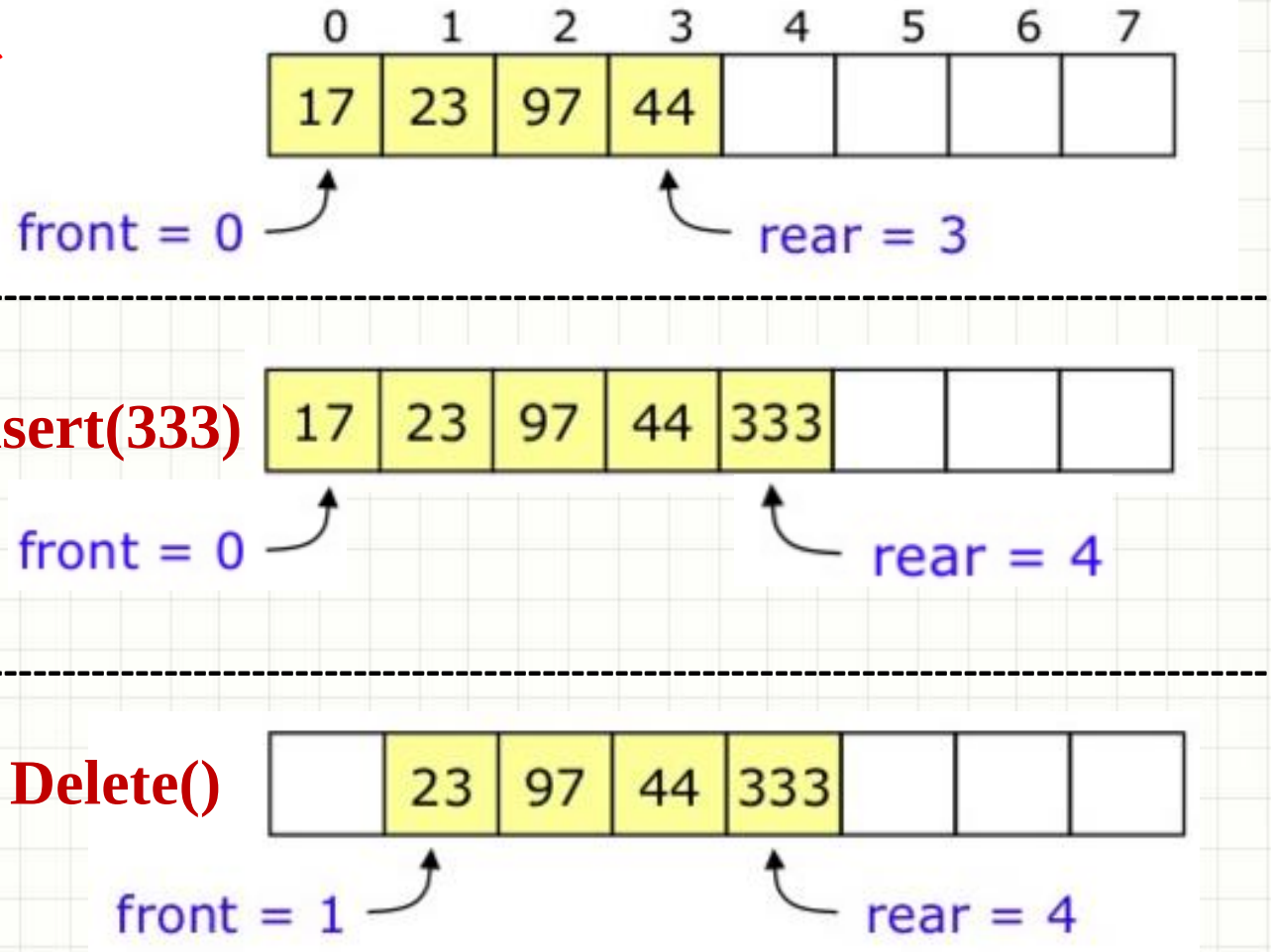


front ve rear

- Kuyruğa bir eleman **eklenince** ne olur?
- Kuyruktan bir eleman **çıkartılınca** (işi bitince ne olur?)

Kuyruk Dizi Gerçekleştirim (devam…)

ÖRNEK 1:



Kuyruk Dizi Gerçekleştirim (devam…)

ÖRNEK 2:

Adım 1: Kuyruk boş.

```
front = -1
rear = -1
```

[0]	[1]	[2]	[3]	[4]

Adım 2: A, B, C elemanlarını sırayla kuyruğa ekle.

```
front = 0
rear = 2
```

[0]	[1]	[2]	[3]	[4]
A	B	C		

Kuyruk Dizi Gerçekleştirim (devam…)

ÖRNEK 2:

Adım 3: Kuyruktan 2 tane elemanı sil.

front = 2

rear = 2

[0]	[1]	[2]	[3]	[4]
		C		

Adım 4: Kuyruğa D, E, F ekle.

front = 2

rear = 4

[0]	[1]	[2]	[3]	[4]
		C	D	E

F için yer kalmadı !!!

2 elemanlık alan kullanılamaz hale geldi !!!

Simple Queue

- Simple Queue (**basit kuyruk**) olarak adlandırılan bu kuyruk tipi
  - *hep ileri yönde hareket etmekte* ve
  - *verimsiz alan kullanımı*na neden olmaktadır.

Simple Queue Kaynak Kod

```
//Sınıf Tanımı
public class SimpleArrayTypedQueue: IQueue

//Üye Değişkenleri
private object[] Queue;
private int front = -1;
private int rear = -1;
private int size = 0;
private int count = 0;

//Constructor
public SimpleArrayTypedQueue(int size)
{
   this.size = size;
   Queue = new object[size];
}
```

Simple Queue Kaynak Kod (devam…)

```
public void Insert(object o)
{
   if ((count == size) || (rear == size -1))
     throw new Exception("Queue dolu.");

   if (front == -1)
     front = 0;

   Queue[++rear] = o;
   count++;
}
```

Simple Queue Kaynak Kod (devam…)

```
public object Remove()
{
   if (IsEmpty())
      throw new Exception("Queue boş.");

   object temp = Queue[front];
   Queue[front] = null;
   front++;
   count--;

   return temp;
}

public bool IsEmpty()
{
    return (count == 0);
}
```

Simple Queue İyileştirmeler

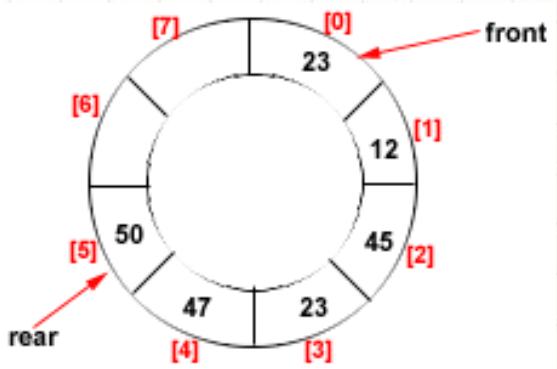
Basit kuyruk gerçekleştiriminde *çeşitli iyileştirmeler* yapılabilir:

- **İyileştirme 1:** Silme sonucunda kuyrukta hiç eleman kalmazsa, kuyruk sıfırdan oluşturulmuş gibi ***ilk durumuna*** getirilebilir.
- **İyileştirme 2:** Kaydırma (**shift**) işlemi yapılarak öndeki boş yerler kullanıma sokulmak üzere arkaya taşınabilir, fakat kaydırmalar aşırı zaman alır ve maliyetlidir.
- **İyileştirme 3:** Diğer iyileştirme kuyruğun boşta kalan öndeki alanlarını kullanmaya yönelik bir geliştirme yapılabilir (**Circular -** Döngüsel Kuyruk).

Circular Queue

- Basit kuyrukta karşılaşılan ve kuyruğun başında kalan *kullanılamayan alan problemini çözmek için* döngüsel kuyruk veri yapısı geliştirilmiştir.
- Döngüsel kuyrukta,
  - Kuyruğun **başı** ile **sonu** **birleştirilmiştir**.

- Önde ***boşalan yerler***, arkadaymış gibi otomatik olarak kullanıma sokulur.



Döngüsel Kuyruk Dizi Gerçekleştirim

ÖRNEK 3:

Adım 1: Kuyruk boş.

```
front = -1
rear = -1
```

[0]	[1]	[2]	[3]	[4]

Adım 2: A, B, C elemanlarını sırayla kuyruğa ekle.

```
front = 0
rear = 2
```

[0]	[1]	[2]	[3]	[4]
A	B	C		

Döngüsel Kuyruk Dizi Gerçekleştirim (devam…)

ÖRNEK 3:

Adım 3: Kuyruktan 2 tane elemanı sil.

front = 2

rear = 2

[0]	[1]	[2]	[3]	[4]
		C		

Adım 4: Kuyruğa D, E, F ekle.

front = 2

rear = 0

[0]	[1]	[2]	[3]	[4]
F		C	D	E

Döngüsel Kuyruk Dizi Gerçekleştirim (devam…)

ÖRNEK 3:

Adım 5: Kuyruktan 1 tane eleman sil.

	[0]	[1]	[2]	[3]	[4]
front = 3 rear = 0	F			D	E

Adım 6: Kuyruğa G elemanını ekle.

	[0]	[1]	[2]	[3]	[4]
front = 3 rear = 1	F	G		D	E

Döngüsel Kuyruk Dizi Gerçekleştirim (devam…)

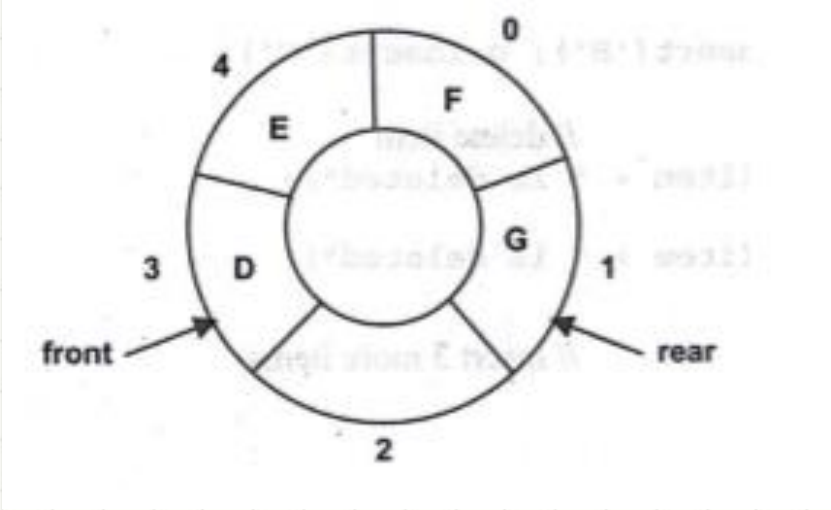
ÖRNEK 3:

Adım 6 sonrasında oluşan son durumun döngüsel kuyruk ile gösterimi aşağıdaki gibidir:

front = 3

rear = 1

[0]	[1]	[2]	[3]	[4]
F	G		D	E



Döngüsel Kuyruk Kaynak Kod (devam…)

```
public void Insert(object o)
{
   if ((count == size) || (rear == size -1))
     throw new Exception("Queue dolu.");

   if (front == -1)
     front = 0;

   //Circular Code Değişikliği
   if (rear == size - 1)
   {
      rear = 0;
      Queue[rear] = o;
   }
   else
      Queue[++rear] = o;
   count++;
}
```

Döngüsel Kuyruk Kaynak Kod (devam…)

```
public object Remove()
{
    if (IsEmpty())
        throw new Exception("Queue boş.");

    object temp = Queue[front];
    Queue[front] = null;

    //Circular Code Değişikliği
    if (front == size - 1)
        front = 0;
    else
        front++;
    count--;

    return temp;
}
```

```
public bool IsEmpty()
{
    return (count == 0);
}
```

Priority Queue

- Standart kuyruk veri yapısı önceliklendirme eksikliği nedeniyle, birçok durumda (problemde) kullanılmak için **uygun olmayabilir**.
- Gerçek hayatta uygulanan kuyruk yapılarında *öncelik durumu dikkate alınır, önceliği* ***yüksek*** *olanlar önce işlem görürler*.

Priority Queue (devam…)

- **Örneğin;** yazılım bakım sürecinde yazılım departmanına iletilen **hataları** düşünelim. İletilen her hata bir havuza atılır ve sırasıyla uygun yazılımcıya iletilir. **Bazı hatalar diğerlerinden daha önceliklidir.**
- Mesela sistemdeki tüm modülleri etkileyen hatalar çok daha kritik olup diğerlerinden daha önce tamamlanmalıdır.
- Aynı şekilde **işletim sistemlerinde** bazı **prosesler** diğerlerinden *daha öncelikli* olarak çalışmak durumundadır.

Priority Queue (devam…)

- Öncelik kuyrukları,
  - **artan** ve **azalan**

  olmak üzere ikiye ayrılırlar.
- Diğer veri yapılarında olduğu gibi kuyrukta bulunan elemanlar, `string veya integer` gibi **basit veri türünde** olabileceği gibi *özelliklere (attribute) sahip* bir **nesne** de olabilir.
- Öncelik kriterinin ne olacağı kuyruktan kuyruğa değişkenlik gösterir.
- Kuyruğa eklenen *elemanın kendisi* veya herhangi bir özelliği, öncelik kriteri olabilir.

Priority Queue (devam…)

- **Örneğin;** telefon rehberi uygulamasında,
  - Kuyruktaki her eleman soyad, ad, adres ve telefon numarası özelliklerinden oluşmakta ve kuyruk **soyada** göre sıralanmaktadır.
- Öncelik kuyrukları;
  - *Dizi,*
  - *Bağlı Liste*
  - *Binary Heap*

  kullanılarak implemente edilebilir.

PQ Dizi Gerçekleştirim

- Artan tipteki öncelik kuyruğunda en küçük değere sahip eleman en öncelikli olarak kuyruktan silinir (*yani ilk olarak işlem görür*).

ÖRNEK 4:

Adım 1: Insert 33.

```
front = 0;
rear = 0
```

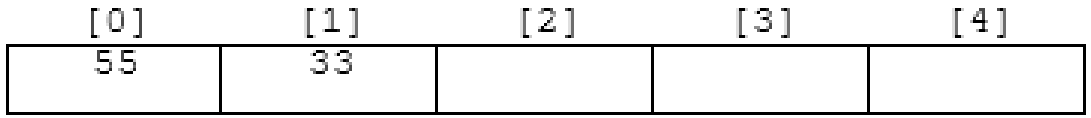
[0]	[1]	[2]	[3]	[4]
33				

PQ Dizi Gerçekleştirim (devam…)

ÖRNEK 4:

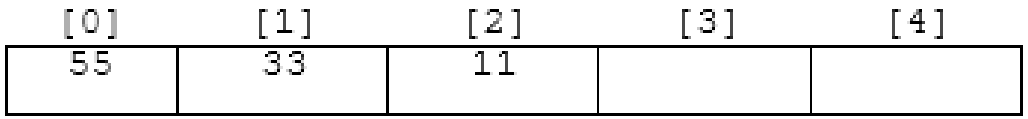
Adım 2: Insert 55.

```
55>33 için 33'ü sağa kaydır

front = 1

rear = 0
```



Adım 3: Insert 11.

```
11>33 olmadığı için 11'i
sağa ekle

front = 2

rear = 0
```

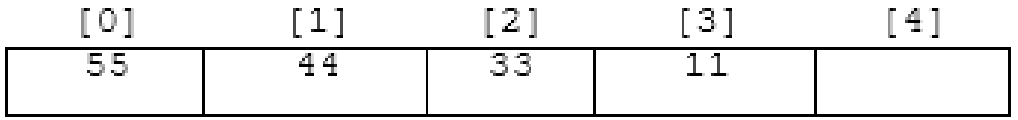


PQ Dizi Gerçekleştirim (devam…)

ÖRNEK 4:

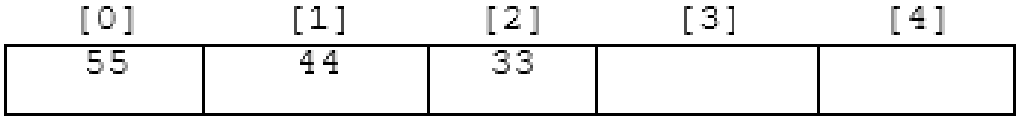
Adım 4: Insert 44.

```
44>11 için 11'i sağa kaydır

Sonra, 44>33 için 33'ü sağa
kaydır

Sonra, 44>55 olmadığı için
44'ü 1 indisli yere koy.

front = 3

rear = 0
```



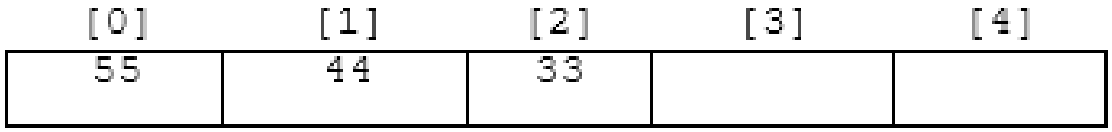
Adım 5: Delete.

```
front elemanı silinir

front = 2

rear = 0
```



PQ Dizi Gerçekleştirim (devam…)

ÖRNEK 4:

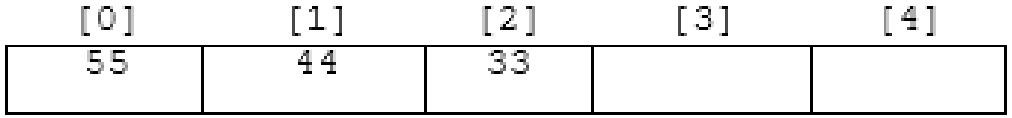


Adım 6: Insert 22.

22>33 olmadığı için 22'yi sona ekle (kaydırmaya gerek yok)

front = 3

rear = 0



Priority Queue Kaynak Kod

```
public class PriorityQueue: IQueue

private object[] Queue;
private int front = -1;
//Not1: rear değeri hep 0 olduğu için değişmez.
//Not2: size ve count değişkenlerinden birisi
//istenirse kullanılmayabilir
private int size = 0;
private int count = 0;
public PriorityQueue(int size)
{
   this.size = size;
   Queue = new object[size];
}
```

Priority Queue Kaynak Kod (devam…)

```
public void Insert(object o)
{
    if (count == size)
        throw new Exception("Queue is full");

    if (IsEmpty())
    {
        front++;
        Queue[front] = o;
    }
    else
    {
        int i;
        //Not3:
        //Son elemandan başlayarak geriye doğru kuyruk kontrol ediliyor
        //Eklenecek elemanın pozisyonu belirleniyor
        //Var olan elemanlar kaydırılıyor
        for (i = count - 1; i >= 0; i--)
        {
            if ((int)o > (int)Queue[i])
                Queue[i + 1] = Queue[i];
            else
                break;
        }
        Queue[i + 1] = o;
        front++;
    }
    count++;
}
```

Priority Queue Kaynak Kod (devam…)

```
public object Remove()
{
    if (this.IsEmpty())
    {
        throw new Exception("Queue is empty...");
    }
    object temp = Queue[front];
    Queue[front] = null;
    front--;
    count--;
    return temp;
}
```

Queue İşlem Karmaşıklığı

- Dizi ile implemente edilmiş kuyruk türlerinin işlem karmaşıklığı aşağıdaki tabloda verilmiştir.

İşlem	Basit Kuyruk	Döngüsel Kuyruk	Öncelik Kuyruğu
Insert	O(1)	O(1)	O(n)
Remove	O(1)	O(1)	O(1)
Peek	O(1)	O(1)	O(1)
IsEmpty	O(1)	O(1)	O(1)