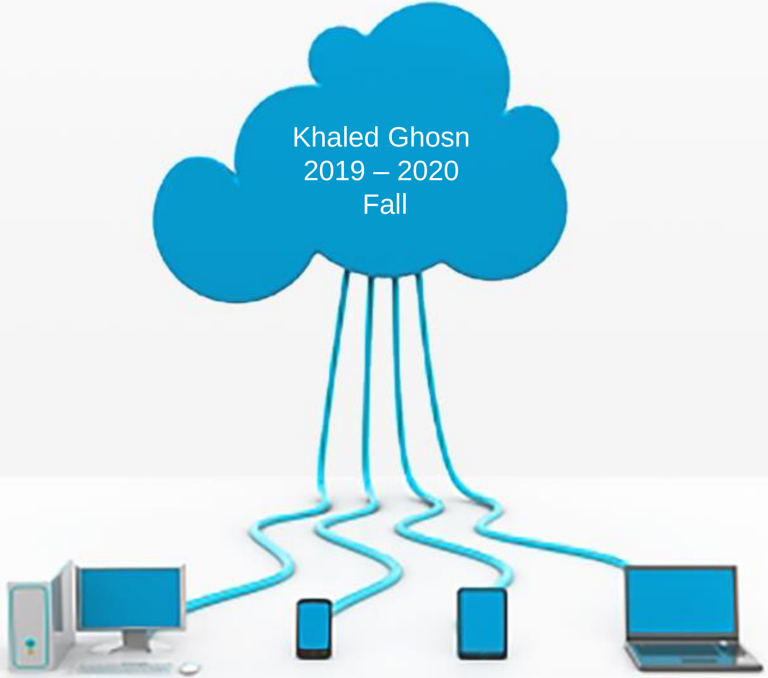
ARTS, SCIENCES & TECHNOLOGY
UNIVERSITY IN LEBANON
AUL

Queues

ADT Queue, Array Queue, Linked Queue



Khaled Ghosn
2019 – 2020
Fall

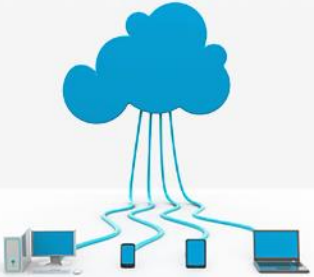
Queue

ADT Queue

- **Queue**:
  - ✓ is an ordered collection of data items in which:
    - ➢ all additions are made at one end called **Back** (**Rear**) of the queue
    - ➢ and all deletions are made from the other end called the **Front** of the queue

- Alternatively, in a queue the element deleted is the one that stayed in the queue the longest

- This is also called first-in-first-out (**FIFO**)



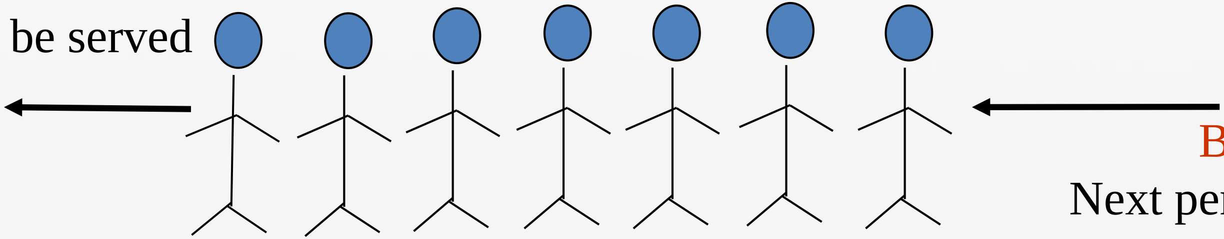
Queue

ADT Queue

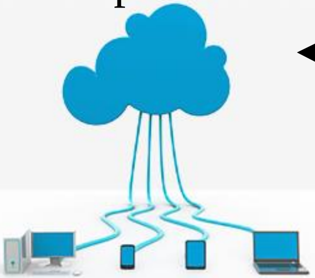
- The insert operation is often called **Enqueue** (at the Back)
- The delete operation is often called **Dequeue** (from the Front)

Queue of people waiting to pay bills

Front of the queue
Next person to be served



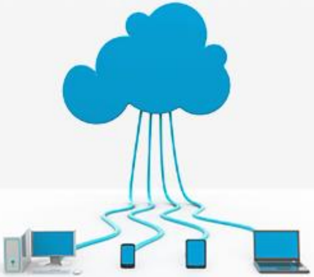
Back of the queue
Next person will join the queue from the Back



Queue

ADT Queue

- Common queue operations:
  - ✓ Constructor
  - ✓ getFront ( ) – Return the item at the front
  - ✓ Enqueue (item) – Add the item to the end of the Q
  - ✓ Dequeue ( ) – Remove & return the item at the front



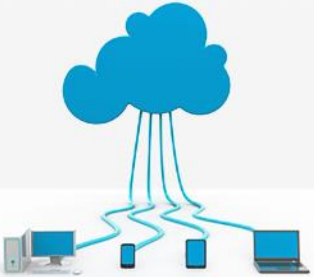
Queue

Implementation

2 ways to implement a Queue

1- Using an array:
ArrayQueue

2- Using a linked list:
LinkedQueue

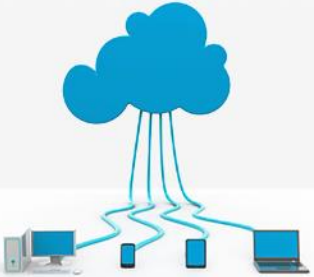


Queue

Interface in Java

- ✓ Java interface corresponding to our Queue ADT
- ✓ Requires the definition of class EmptyQueueException
- ✓ No corresponding built-in Java class

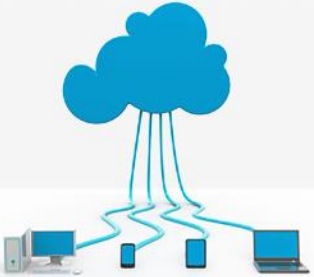
```
public interface Queue {
   public int Length();
   public boolean isEmpty();
   public Anytype getFront()
                    throws EmptyQueueException;
   public void Enqueue(Anytype value);
   public Anytype Dequeue()
                    throws EmptyQueueException;
 }
```



Queue

Application of Queue

- When a resource is shared among multiple consumers.
- When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes
- Load Balancing



Queue

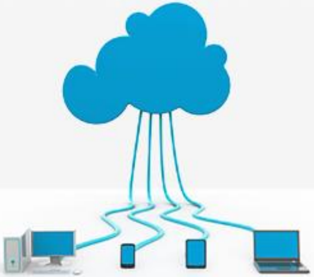
Application of Queue

(i) Queue is used in time sharing system in which programs with the same priority form a queue while waiting to be executed.

(ii) Queue is used for performing level order traversal of a binary tree and for performing breadth first search at a graph.

(iii) Used in simulation related problem.

(iv) When jobs are submitted to a networked printer, they are arranged in order of arrival, i.e.. Jobs are placed in a queue.

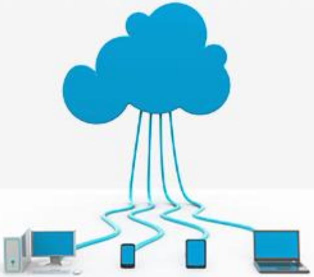


Queue

More Examples of Queue

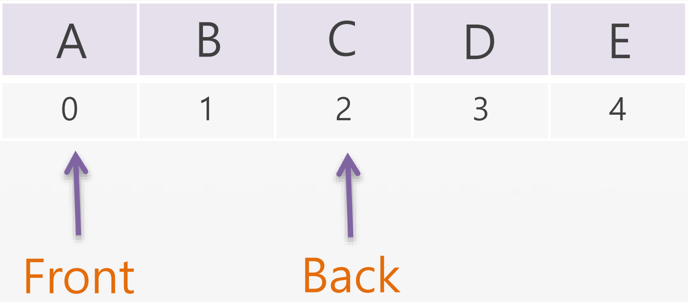
In our daily life

- Airport Security Check
- Cinema Ticket Office
- Bank, ATM
- Printing Job Management
- Packet Forwarding in Routers
- Message queue in Windows
- I/O buffer
- Anything else ?



Array Queue

ArrayQueue operations



A	B	C	D	E
0	1	2	3	4

Size = 5

How to Enqueue again ???

Queue

- *Anytype* **theArray**
- *int* **Front**
- *int* **Back**
- *int* **Size**

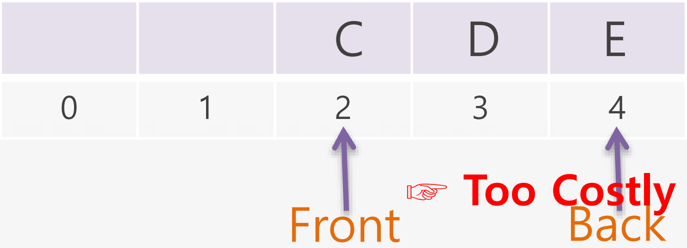
+ **Queue** ( )
+ **Queue** (*int* size)

+ *boolean* **isEmpty** ( )
+ *boolean* **isFull** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *Anytype* **getFront** ( )
+ *void* **Enqueue** (*Anytype* value)
+ *Anytype* **Dequeue** ( )
+ *void* **Print** ( )

Array Queue

Repairing Array Queue !!!

In Dequeue operation: shift all items to Front in the array



		C	D	E
0	1	2	3	4

☞ **Too Costly**

➔ Solution: Wrapped around array

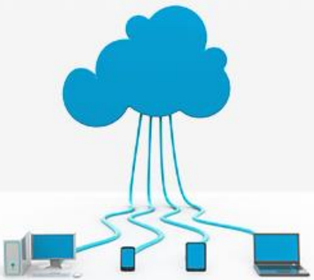
Circular Array

Queue

- *Anytype* **theArray**
- *int* **Front**
- *int* **Back**
- *int* **Size**

+ **Queue** ( )
+ **Queue** (*int* size)

+ *boolean* **isEmpty** ( )
+ *boolean* **isFull** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *Anytype* **getFront** ( )
+ *void* **Enqueue** (*Anytype* value)
+ *Anytype* **Dequeue** ( )
+ *void* **Print** ( )



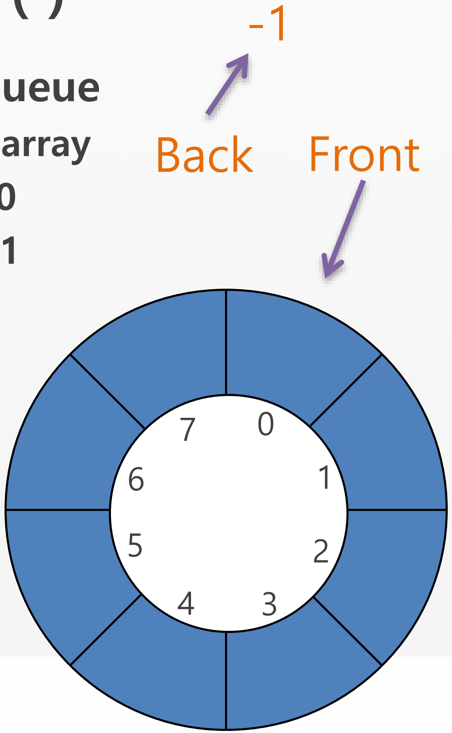
Array Queue

Queue ( )

✓ **Initialize the queue**

1. **Initialize the array**
2. **Set Front to 0**
3. **Set Back to -1**
4. **Set Size to 0**

Size = 0

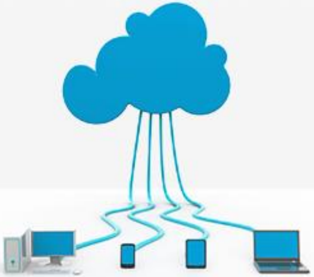


Queue
- *Anytype* **theArray** - *int* **Front** - *int* **Back** - *int* **Size**
+ **Queue** ( ) + **Queue** (*int* size) + *boolean* **isEmpty** ( ) + *boolean* **isFull** ( ) + *void* **makeEmpty** ( ) + *int* **Length** ( ) + *Anytype* **getFront** ( ) + *void* **Enqueue** (*Anytype* value) + *Anytype* **Dequeue** ( ) + *void* **Print** ( )

Array Queue

Queue ( )

```
public ArrayQueue()
{
   int maxSize = 10;
   theArray =  (Anytype[]) new Object[maxSize];

   Front =  0;
   Back  = -1;
   Size  =  0;
}
```

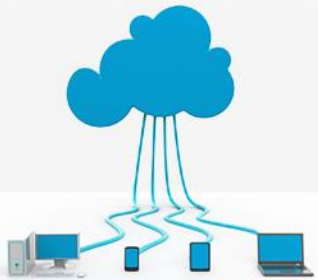


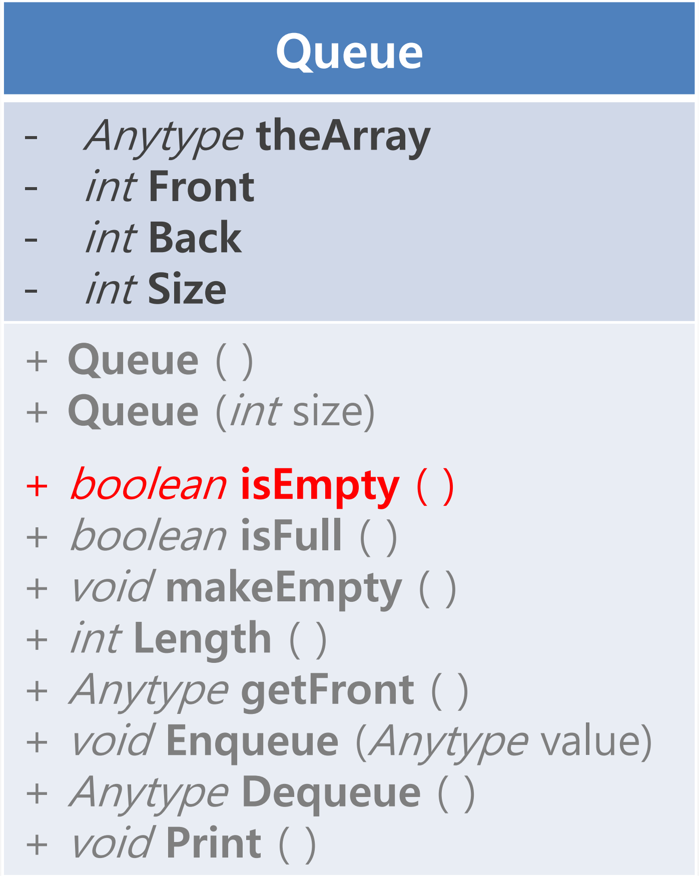
Array Queue

isEmpty ( )

✓ Check if Size is 0

```
public boolean isEmpty()
{
    return Size == 0;
}
```





Queue

- *Anytype* **theArray**
- *int* **Front**
- *int* **Back**
- *int* **Size**

+ **Queue** ( )
+ **Queue** (*int* size)

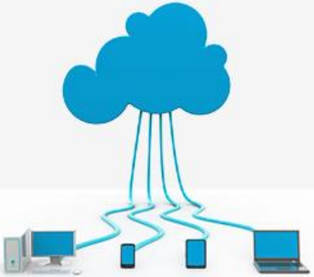
+ *boolean* **isEmpty** ( )
+ *boolean* **isFull** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *Anytype* **getFront** ( )
+ *void* **Enqueue** (*Anytype* value)
+ *Anytype* **Dequeue** ( )
+ *void* **Print** ( )

Array Queue

isFull ( )

- ✓ Check if Size is equal to the length of the array (max Size)

```
public boolean isFull()
{
    return Size == theArray.length;
}
```



Queue

- *Anytype* **theArray**
- *int* **Front**
- *int* **Back**
- *int* **Size**

+ **Queue** ( )
+ **Queue** (*int* size)

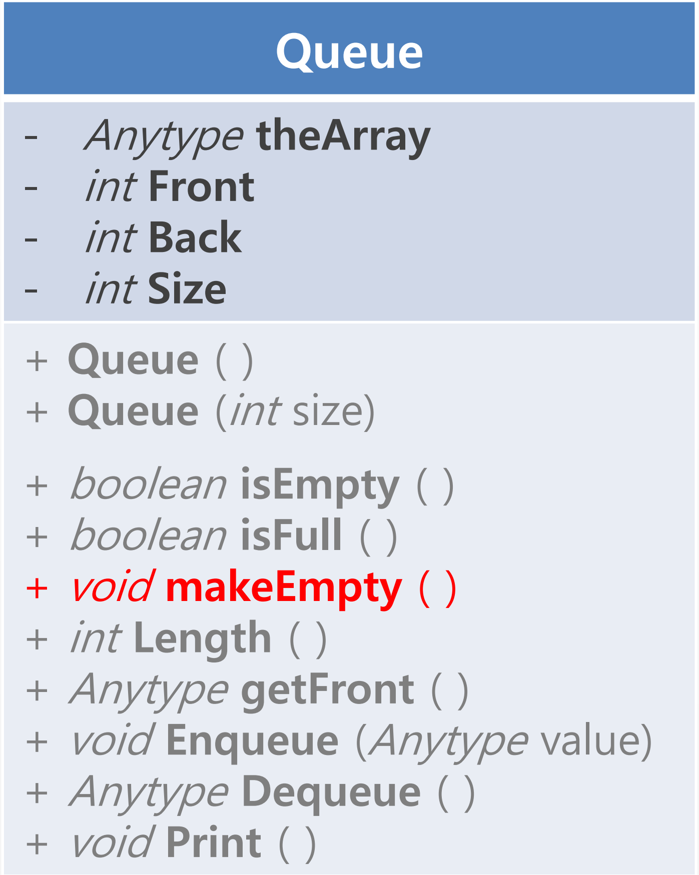
+ *boolean* **isEmpty** ( )
+ *boolean* **isFull** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *Anytype* **getFront** ( )
+ *void* **Enqueue** (*Anytype* value)
+ *Anytype* **Dequeue** ( )
+ *void* **Print** ( )

Array Queue

makeEmpty ( )

1. Check if the queue is empty
   (if empty stop here)
2. Set Front to 0
3. Set Back to -1
4. Set Size to 0

```
public void makeEmpty() {
    if (! isEmpty()) {
        Front =  0;
        Back  = -1;
        Size  =  0;
    }
}
```

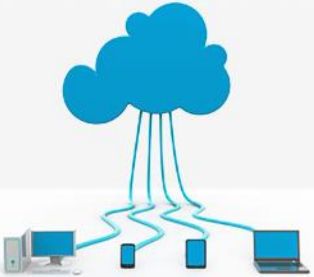


Queue

- *Anytype* **theArray**
- *int* **Front**
- *int* **Back**
- *int* **Size**

+ **Queue** ( )
+ **Queue** (*int* size)

+ *boolean* **isEmpty** ( )
+ *boolean* **isFull** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *Anytype* **getFront** ( )
+ *void* **Enqueue** (*Anytype* value)
+ *Anytype* **Dequeue** ( )
+ *void* **Print** ( )



Array Queue

Length ( )

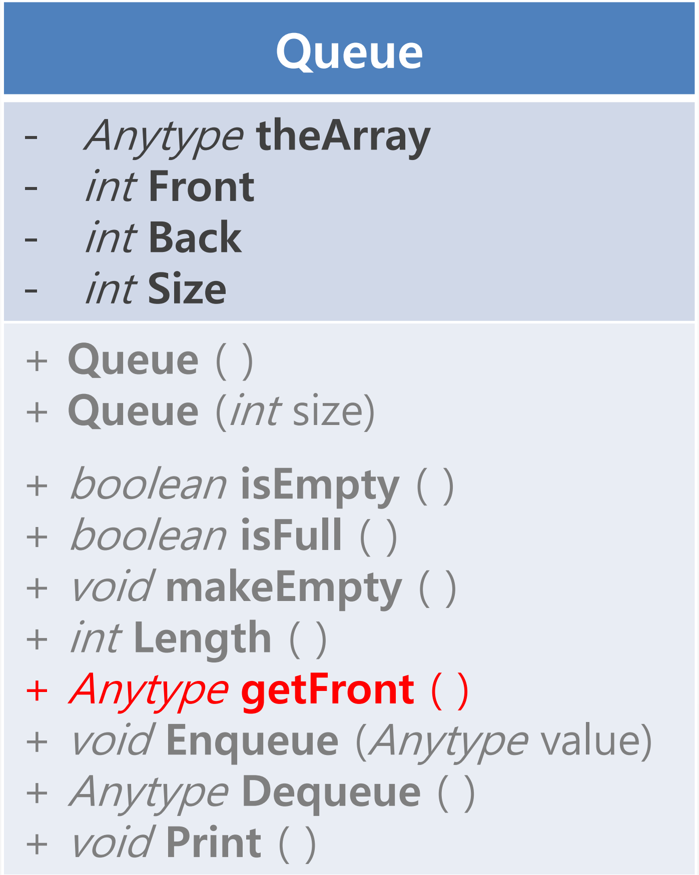
```
public int Length()
{
    if (Front <= Back)
       Size = Back - Front + 1;
    else
       Size = (theArray.length - Front)
            + (Back + 1);


     return Size;
}
```

Queue
- *Anytype* **theArray**
- *int* **Front**
- *int* **Back**
- ~~*int* **Size**~~
+ **Queue** ( )
+ **Queue** (*int* size)
+ *boolean* **isEmpty** ( )
+ *boolean* **isFull** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *Anytype* **getFront** ( )
+ *void* **Enqueue** (*Anytype* value)
+ *Anytype* **Dequeue** ( )
+ *void* **Print** ( )

Array Queue

getFront ( )

```
public Anytype getFront() {
    if(isEmpty())
        throw new RuntimeException();

    return theArray[Front];
}
```



Queue

- *Anytype* **theArray**
- *int* **Front**
- *int* **Back**
- *int* **Size**

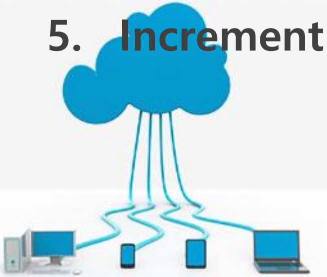
+ **Queue** ( )
+ **Queue** (*int* size)

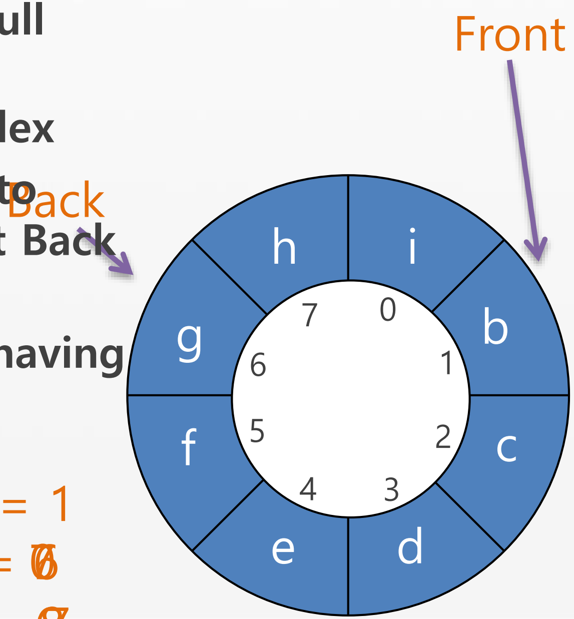
+ *boolean* **isEmpty** ( )
+ *boolean* **isFull** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *Anytype* **getFront** ( )
+ *void* **Enqueue** (*Anytype* value)
+ *Anytype* **Dequeue** ( )
+ *void* **Print** ( )

Array Queue

Enqueue (value)

1. Check if the queue is full
   (if full stop here)
2. Increment the Back index
3. Check if Back is equal to array’s max size => set Back to 0
4. Set the array element having the Back as an index
5. Increment Size





Front= 1

Back= 7

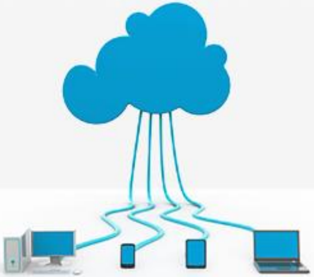
Size= 8

Queue
- *Anytype* **theArray** - *int* **Front** - *int* **Back** - *int* **Size**
+ **Queue** ( ) + **Queue** (*int* size) + *boolean* **isEmpty** ( ) + *boolean* **isFull** ( ) + *void* **makeEmpty** ( ) + *int* **Length** ( ) + *Anytype* **getFront** ( ) + *void* **Enqueue** (*Anytype* value) + *Anytype* **Dequeue** ( ) + *void* **Print** ( )

Array Queue

Enqueue (value)

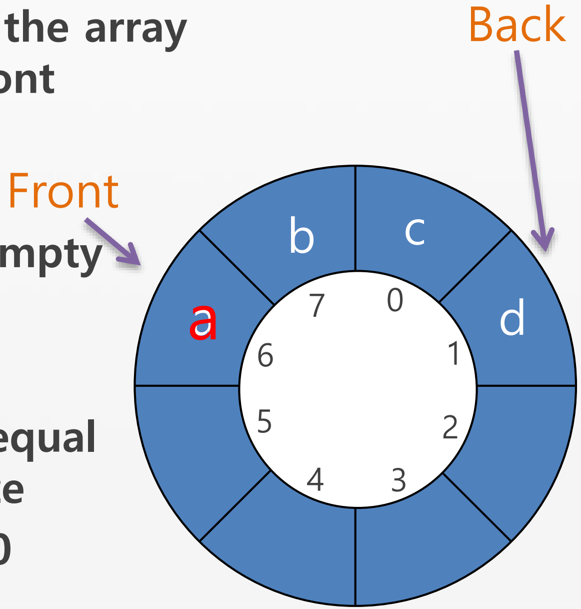
```
public void Enqueue(Anytype value)
{
   if (isFull())
      throw new RuntimeException();

   if(++Back == theArray.length)
      Back = 0;

   theArray[Back] = value;
   Size++;
}
```



Array Queue

Dequeue ( )

1. Check if the queue is empty
   (if empty stop here)
2. Get the data stored in the array element having the Front as an index
3. If size equal to 1 => just make the queue empty
2. If not
   - i. increment Front
   - ii. check if Front is equal to array's max size
     => set Front to 0
5. Decrement Size
6. Return data stored (in step 2)



Queue

- *Anytype* **theArray**
- *int* **Front**
- *int* **Back**
- *int* **Size**

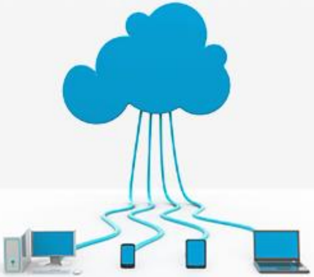
+ **Queue** ( )
+ **Queue** (*int* size)

+ *boolean* **isEmpty** ( )
+ *boolean* **isFull** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *Anytype* **getFront** ( )
+ *void* **Enqueue** (*Anytype* value)
+ *Anytype* **Dequeue** ( )
+ *void* **Print** ( )

Array Queue

Dequeue ( )

```
public Anytype Dequeue() {
    if(isEmpty())
        throw new RuntimeException();

    Anytype removedValue = theArray[Front];

    if(Size == 1)
        makeEmpty();
    else
        if(++Front == theArray.length)
            Front = 0;

    Size--;
    return removedValue;
}
```



Array Queue

Print ( )

Queue

- *Anytype* **theArray**
- *int* **Front**
- *int* **Back**
- *int* **Size**

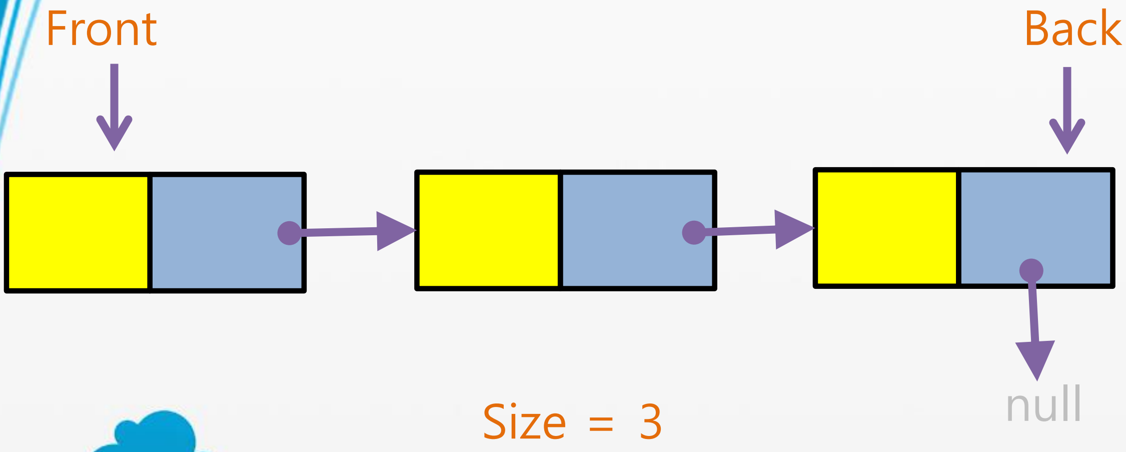
+ **Queue** ( )
+ **Queue** (*int* size)

+ *boolean* **isEmpty** ( )
+ *boolean* **isFull** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *Anytype* **getFront** ( )
+ *void* **Enqueue** (*Anytype* value)
+ *Anytype* **Dequeue** ( )
+ *void* **Print** ( )

```
public void Print() {
 String s = "\n ";

 if (this.isEmpty()) {
   System.out.println("\n The queue is empty");
   return;
 }

 if (this.Front <= this.Back)
   for (int i = this.Front; i <= this.Back; i++)
     s += this.theArray[i] + " ";
 else {
   for (int i = this.Front; i <= this.theArray.length - 1; i++)
     s += this.theArray[i] + " ";

   for (int i = 0; i <= this.Back; i++)
     s += this.theArray[i] + " ";
   }

   System.out.println(s); }
```

Linked Queue

LinkedQueue operations



Queue

- *Node* **Front**
- *Node* **Back**
- *int* **Size**

+ **Queue** ( )

+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *Anytype* **getFront** ( )
+ *void* **Enqueue** (*Anytype* value)
+ *Anytype* **Dequeue** ( )
+ *void* **Print** ( )

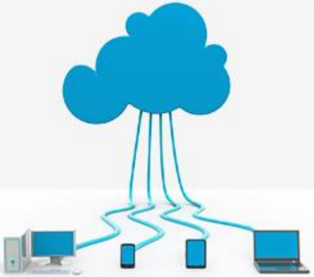
Linked Queue

Queue ( )

✓ Initialize the queue

1. Set Front to null
2. Set Back to null
3. Set Size to 0

```
public LinkedQueue()
{
   Front = Back = null;
   Size = 0;
}
```



Queue

- *Node* **Front**
- *Node* **Back**
- *int* **Size**

+ **Queue** ( )
+ **Queue** (*int* size)

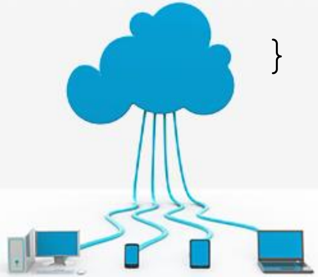
+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *Anytype* **getFront** ( )
+ *void* **Enqueue** (*Anytype* value)
+ *Anytype* **Dequeue** ( )
+ *void* **Print** ( )

Linked Queue

isEmpty ( )

✓ **Check if Front is null**
or if Size is 0
or if Back is null

```
public boolean isEmpty()
{
    return Front == null;
    // return Size == 0;
}
```



Queue

- *Node* **Front**
- *Node* **Back**
- *int* **Size**

+ **Queue** ( )
+ **Queue** (*int* size)

+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *Anytype* **getFront** ( )
+ *void* **Enqueue** (*Anytype* value)
+ *Anytype* **Dequeue** ( )
+ *void* **Print** ( )

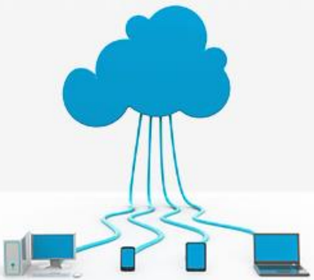
Linked Queue

makeEmpty ( )

1. Set Front to null
2. Set Back to null
3. Set Size to 0

```
public void makeEmpty()
{
    Front = Back = null;
    Size = 0;
}
```

Queue
- *int* **Front** - *int* **Back** - *int* **Size**
+ **Queue** ( ) + **Queue** (*int* size) + *boolean* **isEmpty** ( ) + *void* **makeEmpty** ( ) + *int* **Length** ( ) + *Anytype* **getFront** ( ) + *void* **Enqueue** (*Anytype* value) + *Anytype* **Dequeue** ( ) + *void* **Print** ( )



Linked Queue

Length ( )

```
public int Length()
{
    int Size=0;
    Node cn=Front;
    while(cn!=null) {
        cn = cn.getNextNode();
        Size++;
    }

    return Size;
}
```

Queue

- *int* **Front**
- *int* **Back**
- ~~*int* **Size**~~

+ **Queue** ( )
+ **Queue** (*int* size)

+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *Anytype* **getFront** ( )
+ *void* **Enqueue** (*Anytype* value)
+ *Anytype* **Dequeue** ( )
+ *void* **Print** ( )

Linked Queue

getFront ( )

```
public Anytype getFront() {
    if(isEmpty())
        throw new RuntimeException();

    return Front.getData();
}
```

Queue

- *int* **Front**
- *int* **Back**
- *int* **Size**

+ **Queue** ( )
+ **Queue** (*int* size)

+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *Anytype* **getFront** ( )
+ *void* **Enqueue** (*Anytype* value)
+ *Anytype* **Dequeue** ( )
+ *void* **Print** ( )

Linked Queue

Enqueue (value)

1. Create a new node storing value
2. If Queue is empty:

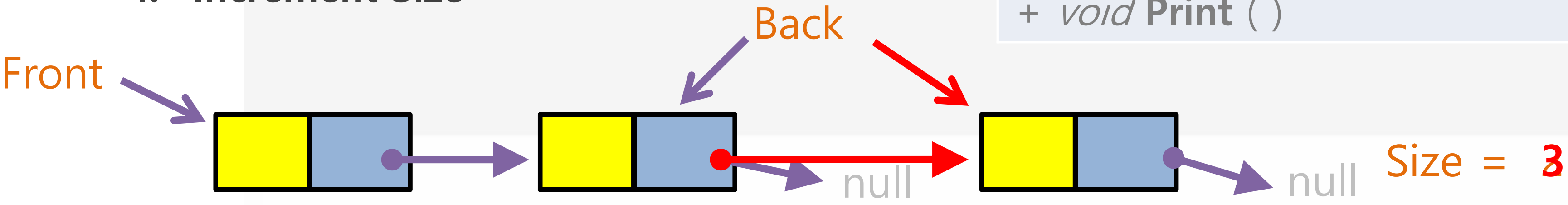
   Let Front & Back refer to the new node
3. If Queue is not empty:
   a. Let the nextNode reference of the Back refers to new node
   b. Let Back refers to the new node
4. Increment Size

Queue

- *int* **Front**
- *int* **Back**
- *int* **Size**

+ **Queue** ( )
+ **Queue** (*int* size)

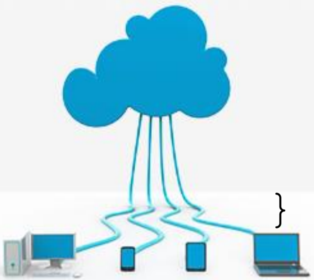
+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *Anytype* **getFront** ( )
+ *void* **Enqueue** (*Anytype* value)
+ *Anytype* **Dequeue** ( )
+ *void* **Print** ( )



Linked Queue

Enqueue (value)

```
public void Enqueue(Anytype value) {
   Node <Anytype> newNode = new Node <Anytype> (value);

   if (isEmpty())
      Front = Back = newNode;
   else {
      Back.setNextNode(newNode);
      Back = newNode;
   }

   Size++;
}
```



Linked Queue

Dequeue ( )

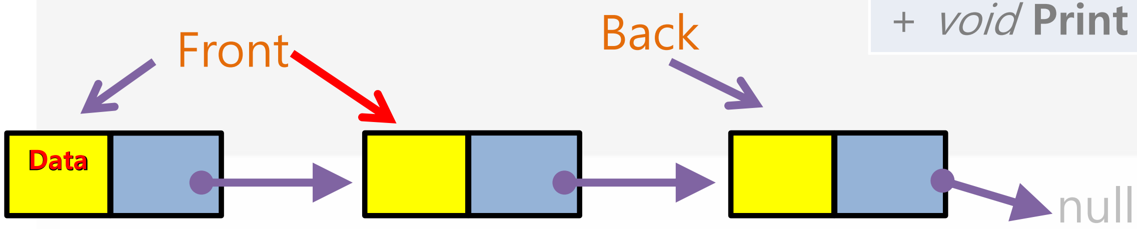
1. Check if the queue is empty
   (if empty stop here)
2. Get the data stored in the Front
3. If size equal to 1 => just make the queue empty (set Front & Back to be null)
4. If size more than 1, let the Front refers to its next node.
5. Decrement Size
6. Return data stored (in step 2)

Queue

- *int* Front
- *int* Back
- *int* Size

+ Queue ( )
+ Queue (*int* size)

+ *boolean* isEmpty ( )
+ *void* makeEmpty ( )
+ *int* Length ( )
+ *Anytype* getFront ( )
+ *void* Enqueue (*Anytype* value)
+ *Anytype* Dequeue ( )
+ *void* Print ( )

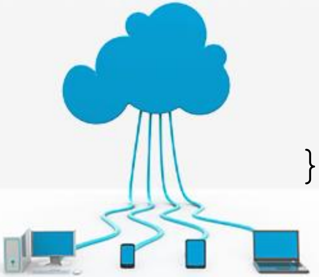


Size = 2

Linked Queue

Dequeue ( )

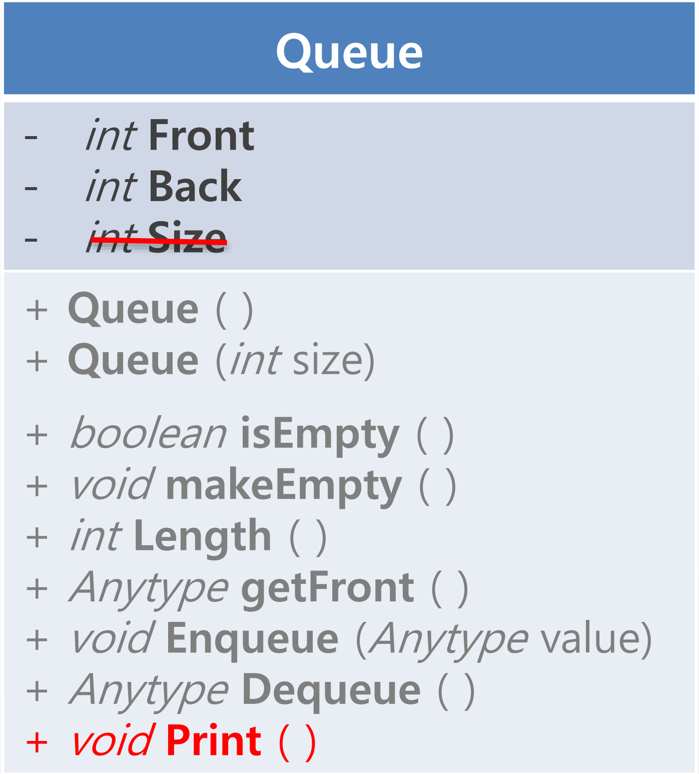
```
public Anytype Dequeue()   {
    if (isEmpty())
        throw new RuntimeException();

    Anytype removedValue = Front.getData();

    if (Front == Back)     // if (Size == 1)
        Front = Back = null;
    else
        Front = Front.getNextNode();

    Size--;
    return removedValue;
}
```



Linked Queue

Print ( )

1. Check if the queue is not empty
   (if empty stop here)
2. Start from Front node
3. If Node is not null:
   a. Print data stored in the node
   b. Move to next node
4. Repeat step 2 until Node is null

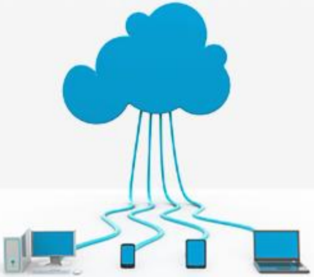


Queue

- *int* **Front**
- *int* **Back**
- ~~*int* **Size**~~

+ **Queue** ( )
+ **Queue** (*int* size)

+ *boolean* **isEmpty** ( )
+ *void* **makeEmpty** ( )
+ *int* **Length** ( )
+ *Anytype* **getFront** ( )
+ *void* **Enqueue** (*Anytype* value)
+ *Anytype* **Dequeue** ( )
+ *void* **Print** ( )



Linked Queue

Print ( )

```
public void Print() {
  if(isEmpty())
    System.out.println("The Queue is empty");
  else {
    Node < Anytype > currentNode = Front;

    while (currentNode!=null) {
      System.out.print(currentNode.getData().toString() + " --> ");
      currentNode = currentNode.getNextNode();
    }

    System.out.println("");
  }
}
```

Queue

Exercises

R-6.7 , R-6.9 , C-6.28 , C-6.29