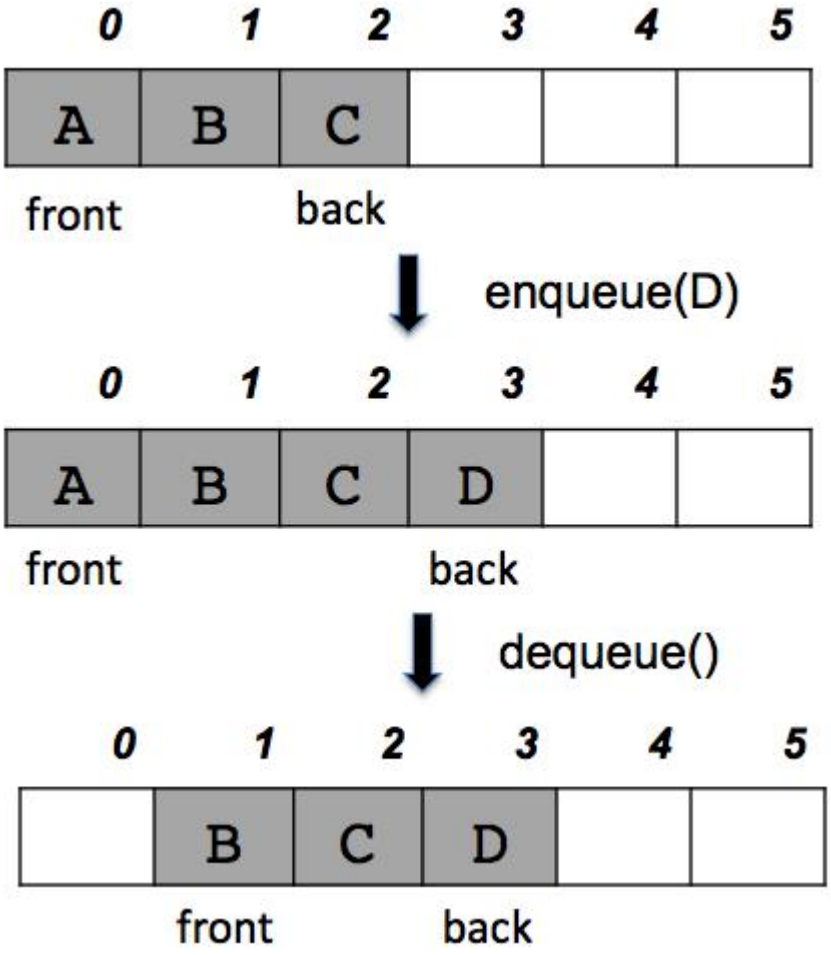
Queues

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −



As in stacks, a queue can also be implemented using Arrays or Linked-lists. For the sake of simplicity, we shall implement queues using one-dimensional array.

Using Inbuilt Queue Interface

The java.util.Queue is a subtype of java.util.Collection interface. Since it is an interface, we need a concrete class during its declaration.

```
import java.util.Queue;
import java.util.LinkedList;
```

```
class QueueUse{
  public static void main(String[] args){
    Queue<Integer> queue = new LinkedList<>();

    queue.add(10);// Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available.

    queue.peek();//Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

    queue.element();//Similar to peek(). Throws NoSuchElementException if queue is empty.

    queue.poll();//Retrieves and removes the head of this queue, or returns null if this queue is empty.

    queue.remove();//Retrieves and removes the head of this queue. This method differs from poll only in that it throws an exception if this queue is empty. Avoid using it.

    queue.size();//Returns the number of elements in this collection
  }
}
```

PriorityQueue class can also be used to initialize queue object.

Basic Operations

Queue operations may involve initializing or defining the queue and then utilizing it. Here we shall try to understand the basic operations associated with queues −

1) enqueue() − add (store) an item to the queue.
2) dequeue() − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

3) peek() − Gets the element at the front of the queue without removing it.
4) isempty() − Checks if the queue is empty.
5) size() - Returns current size of queue.

In queue, we always dequeue (or access) data, pointed by front pointer and while enqueing (or storing) data in the queue we take help of rear/back pointer.

Enqueue Operation

Queues maintain two data pointers, front and rear. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

Step 1 − Check if the queue is full.
Step 2 − If the queue is full, produce overflow error and exit.
Step 3 − If the queue is not full, increment rear pointer to point the next empty space.
Step 4 − Add data element to the queue location, where the rear is pointing.
Step 5 − return success.

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where front is pointing and remove the data after access. The following steps are taken to perform dequeue operation −

Step 1 − Check if the queue is empty.

Step 2 − If the queue is empty, produce underflow error and exit.

Step 3 − If the queue is not empty, access and store the data where front is pointing.

Step 4 − Increment front pointer to point to the next available data element.

Step 5 − Return the data.

```
class node<T>{
  T data;
    node<T> next;
    node(T data){
        this.data = data;
    }
}

public class queue<T>{
    private node<T> front, rear;
    private int size ;
    queue(){
        front = rear = null;
        size = 0;
    }

    public void enqueue(T data){
        node<T> newnode = new node<>(data);
        if(front == null){
            front = rear = newnode;
        }
        else{
            rear.next = newnode;
            rear = newnode;
        }
        size++;
    }

    public T dequeue() throws QueueUnderflowException{
        T temp = null;
        if(front == null){
            throw new QueueUnderflowException();
        }
        else if(front == rear){
            temp = front.data;
            front = rear = null;
        }
        else{
```

```
            temp = front.data;
            front = front.next;
        }
        size--;
        return temp;
    }

    public boolean isempty(){
        return (front == null)? true:false;
    }

  public int size(){
    return size;
  }
}
```

Applications of Queue Data Structure

Queue is used when things don't have to be processed immediatly, but have to be processed in First In First Out order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.

1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.

2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

3) In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.

4) Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

Deque Implementations

Being a Queue subtype all methods in the Queue and Collection interfaces are also available in the Deque interface.

Since Deque is an interface you need to instantiate a concrete implementation of the interface in order to use it. You can choose between the following Deque implementations in the Java Collections API:

java.util.ArrayDeque

java.util.LinkedList

LinkedList is a pretty standard deque / queue implementation.

ArrayDeque stores its elements internally in an array. If the number of elements exceeds the space in the array, a new array is allocated, and all elements moved over. In other words, the ArrayDeque grows as needed, even if it stores its elements in an array.

Here are a few examples of how to create a Deque instance:

```
Deque dequeA = new LinkedList();
Deque dequeB = new ArrayDeque();
```

Adding and Accessing Elements

To add elements to the tail of a Deque you call its add() method. You can also use the addFirst() and addLast() methods, which add elements to the head and tail of the deque.

```
Deque dequeA = new LinkedList();

dequeA.add     ("element 1"); //add element at tail
dequeA.addFirst("element 2"); //add element at head
dequeA.addLast ("element 3"); //add element at tail
```

The order in which the elements added to the Deque are stored internally, depends on the implementation. The two implementations mentioned earlier both store the elements in the order (first or last) in which they are inserted.

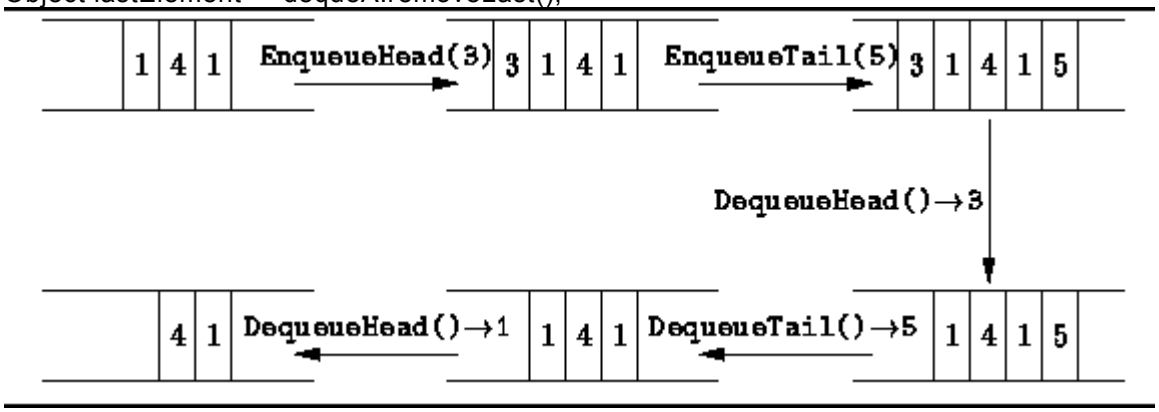
You can peek at the element at the head of the queue without taking the element out of the queue. This is done via the element() method. You can also use the getFirst and getLast() methods, which return the first and last element in the Deque. Here is how that looks:

```
Object firstElement = dequeA.element();
Object firstElement = dequeA.getFirst();
Object lastElement  = dequeA.getLast();
```

Removing Elements

To remove elements from a deque, you call the remove(), removeFirst() and removeLast methods. Here are a few examples:

```
Object firstElement = dequeA.remove();
Object firstElement = dequeA.removeFirst();
Object lastElement  = dequeA.removeLast();
```



Generic Deque

By default you can put any Object into a Deque, but from Java 5, Java Generics makes it possible to limit the types of object you can insert into a Deque. Here is an example:

```
Deque<MyObject> deque = new LinkedList<MyObject>();
```

Circular Queues

In a standard queue data structure re-buffering problem occurs(if we are using arrays) for each dequeue operation. To solve this problem by joining the front and

rear ends of a queue to make the queue as a circular queue Circular queue is a linear data structure. It follows FIFO principle.

1) In circular queue the last node is connected back to the first node to make a circle.

2) Circular linked list fallow the First In First Out principle.

3) Elements are added at the rear end and the elements are deleted at front end of the queue

4) Both the front and the rear pointers points to the beginning of the array.

6) It is also called as “Ring buffer”.

7) Items can inserted and deleted from a queue in O(1) time.

Circular Queue can be created in three ways they are

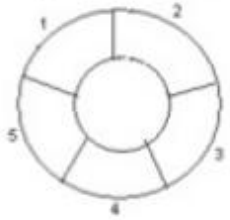
1) Using single linked list

2) Using double linked list

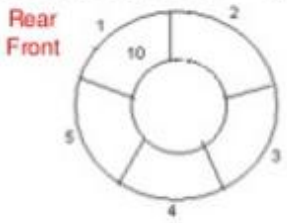
3) Using arrays

Example: Consider the following circular queue with N = 5.

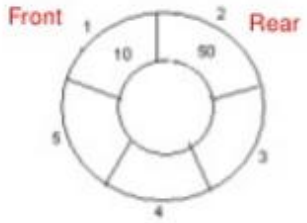
1. Initially, Rear = 0, Front = 0.



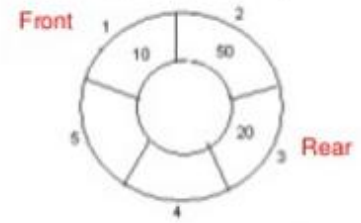
2. Insert 10, Rear = 1, Front = 1.



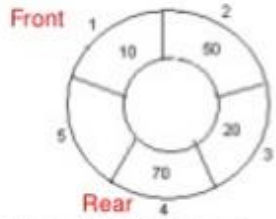
3. Insert 50, Rear = 2, Front = 1.



4. Insert 20, Rear = 3, Front = 0.



5. Insert 70, Rear = 4, Front = 1.



6. Delete front, Rear = 4, Front = 2.