Queue Data Structures

Queue is a linear data structure, in which the first element is inserted from one end called **REAR**(also called tail), and the deletion of existing element takes place from the other end called as **FRONT**(also called head). This makes queue as FIFO(First in First Out) data structure, which means that element inserted first will also be removed first.

The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.

Basic features of Queue

1. Like Stack, Queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO(First in First Out) structure.

   Once a new element is inserted into the Queue(*Enqueue(also called* enq, enque, add, or insert), all the elements inserted before the new element in the queue must be removed, to remove the new element(called *Dequeue(*also called deq, deque, remove, or serve.).

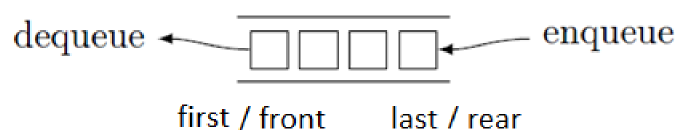
3. **peek( )** function is used to return the value of first element without dequeuing it.

Applications of Queue

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following points :

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

Logical Structure

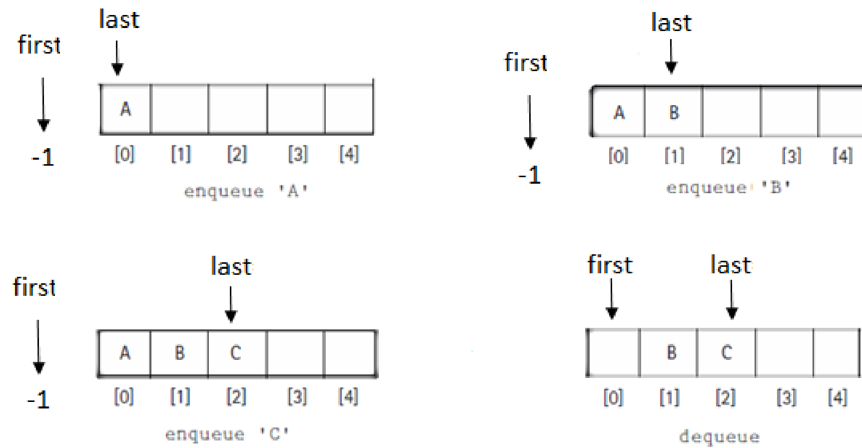


Storage (physical) Structure

Storage structure depends on the implementation of queue , array or linked list structure.

Example: Draw queue in each the following cases:

1. enqueue('A') 2. enqueue('B') 3. enqueue('C') 4. dequeue();

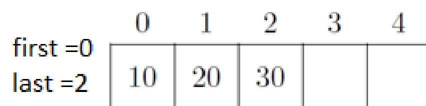


Implementation of Queue

Queue can be implemented using an Array, Stack or Linked List.

Implementation of Queue using Array

The easiest way of implementing a queue is by using an Array. Initially the **first(head, front)** and the **last(tail, rear)** of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the tail keeps on moving ahead, always pointing to the position where the next element will be inserted, while the head remains at the first index.



In the following Queue class:

```
class queue {
   int first=-1;
   int last=-1;
   int qu []=new int [10];

   boolean isempty() {
      return (first == last); }

   boolean isfull() {
      return (last==qu.length-1);}

   void enQueue(int val) {
      if (isfull())
        System.out.println("Queue is full");
      else {
         last++;
         qu[last] = val; }
   }
```

```
int deQueue() {
    int val=-1;
    if (isempty()) {
      System.out.println("Queue is empty, cant dequeue");
    else {
       first++;
       val = qu[first];}
    return val;}
}
```

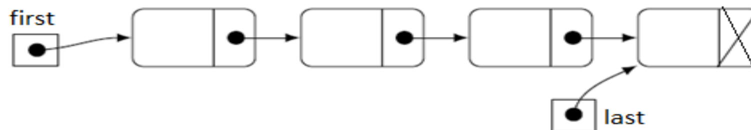
Exercise: Suppose the following operations

1. dequeue(): Returns and removes item from front of queue.
2. void enqueue(int item): Adds item to last of queue.
3. boolean isEmpty(): Returns true if queue has no elements in it.
4. boolean isFull(): Returns true if queue full.
5. int peek() : Returns item at front of queue without removing it.
6. int size() : Number of elements in queue.

Show the results of these operations on an initially empty integer queue named q with Draw the queue contents after each operation, making clear where the front is, and give the return value of each non-void method.

1. q.enqueue(5)
2. q.enqueue(8)
3. q.peek()
4. q.enqueue(3)
5. q.dequeue()
6. q.enqueue(10)
7. q.size()
8. q.enqueue(4)
9. q.dequeue()
10. q.dequeue()
11. q.enqueue(1)
12. q.dequeue()
13. q.enqueue(2)
14. q.dequeue()
15. q.enqueue(3)
16. q.dequeue()
17. q.size()
18. q.enqueue(4)
19. q.peek()
20. q.dequeue()

☒ Implementation of Queue using linked list

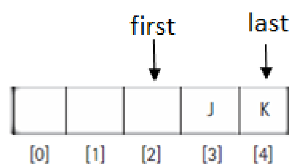


In the following Queue class in this case:

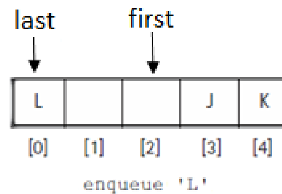
```
class QueueLinkedList{
  node first = null;
  node last = null;

   boolean isEmpty() {
      return (first == null);}

   void enque(int data) {
      node n = new node(data);
      if (isEmpty()) {
         n.next = first;
         first = n;
         last = n;}
      else {
         last.next = n;
         last = n;
         last.next = null;}
   }

  void deque() {
        if (first==null)
           System.out.println("Queue is empty, cant dequeue");
        else
          first = first.next;}

   void displayList() {
      node current = first;
      while (current != null) {
         System.out.print(current.data+" ");
         current = current.next;}
   }
}
```

Circular Queue

Suppose the following queue:



and we went to execute enqueue('L'), in this case queue was full, it is possible for the last of the queue to reach the end of the (physical) array when the (logical) queue is not yet full. Because there may still be space available at the beginning of the array, the solution is to let the array can be treated as a circular structure in which the last slot is followed by the first slot as shown in below figure.



To get the next position for the last indicator, we can use an *if* statement in enQueue method:

```
if (last == (qu.length - 1))
   last = 0;
else
   last = last + 1;
```

And also in deQueue method:

```
if (first == (qu.length - 1))
   first = 0;
else
   first = first + 1;
```

Numeric for Circular Queues

Another way to reset last is to use the modulo (%) operator:

- ❑ Front increases by (1 modulo size(length of queue) after each dequeue( ).
- ❑ Front = (Front+1) % size;
- ❑ Rear(or last) increases by (1 modulo size(length of queue)) after each enqueue( ):
- ❑ Rear= (Rear +1) % size;

Example:

if last=3 and size=5 then:

1%5=1

2%5=2

3%5=3

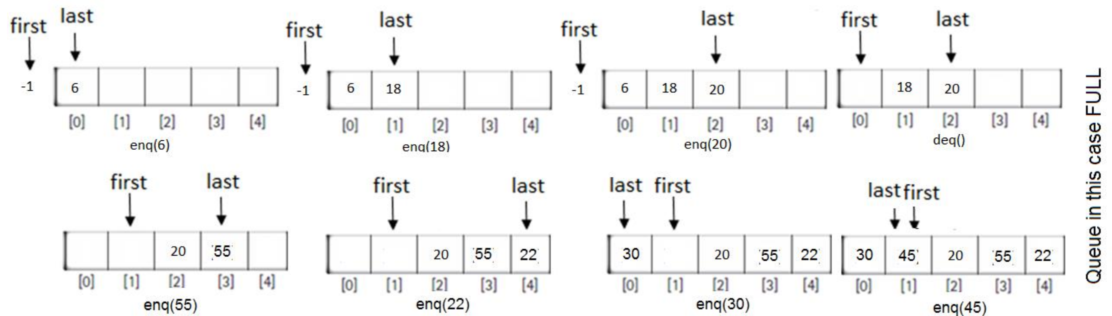
4%5=4

5%5=0

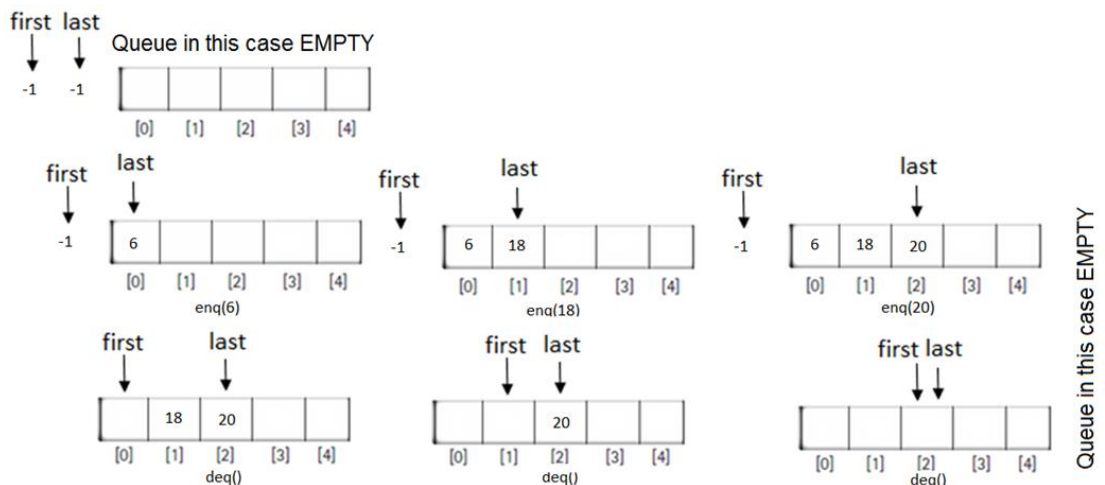
6%5=1

7%5=2 ...etc

The following example shown the queue is become full when first=last



And also from the following cases shown the queue is empty when first=last



Then, the condition last=first (in isempty() method) and the condition last=qu.length(in isfull() method) are not suitable in circular queue, these methods must be as follow:

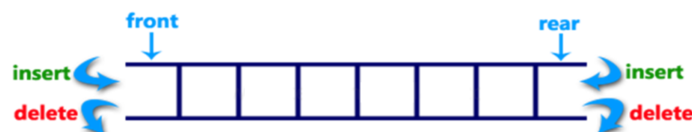
```
boolean isfull() {
    return (first==last && first !=-1); }

  boolean isempty() {
    return (last==-1 && first==-1);}
```

and in dequeue method if (last=first) must restart the queue as:

```
first=-1;
last=-1;
```

Double Ended Queue

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front** and **rear**). That means, we can insert at both front and rear positions and can delete from both front and rear positions.



Double Ended Queue can be represented in TWO ways, those are as follows...

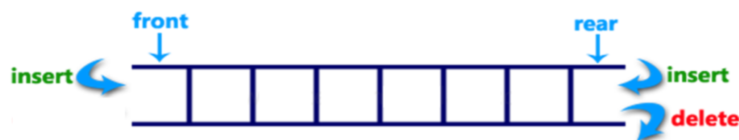
1. Input Restricted Double Ended Queue
2. Output Restricted Double Ended Queue

Input Restricted Double Ended Queue

In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.

Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and deletion operation is performed at both the ends.



Problem: Implementing the enqueue, dequeue, Peek, Clear in the Double Ended Queue.

Hint: In the following algorithm for each above operation

enqueue operation

Create new object(NewItem)
If rear points to null
    rear= NewItem
    Front = NewItem
Else
    NewItem.next = rear
    rear= NewItem

DeQueue operation

If front points to null
    print (the queue is empty)
Else
    for (current = Rear, current.next != Front, current = current.next);
      Front = current;

Peek operation

If Front points to null
    print (the queue is empty)
Else
    return top

Clear operation

```
If Front points to null
        print (the stack is empty)
Else
        Rear = null;
        Front = null;
```

Q1: Write a **printAll** method which display all elements' data of the queue.
Q2: Write a **find** method which search about a specific data in double ended queue.
Q3: Write a **sortAsc** method which sorts the elements of the queue ascending.

Implementation of Queue using Stacks

A Queue is defined by its property of FIFO, which means First in First Out, i.e the element which is added first is taken out first. Hence we can implement a Queue using Stack for storage instead of array.

For performing **enqueue** we require only one stack as we can directly **push** data into stack, but to perform **dequeue** we will require two Stacks, because we need to follow queue's FIFO property and if we directly **pop** any data element out of Stack, it will follow LIFO approach(Last in First Out).

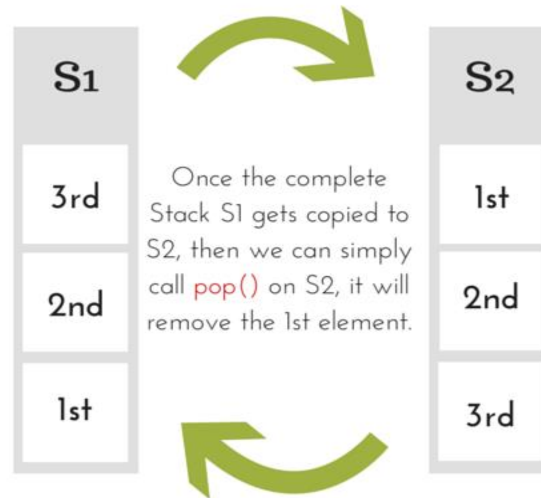
In all we will require two Stacks, we will call them InStack and OutStack.

```
class Queue {
  Stack S1, S2;
  //defining methods
   void enqueue(int x);
   int dequeue();
}
```

As our Queue has Stack for data storage, hence we will be adding data to Stack, which can be done using the push() method, hence :

```
void enqueue(int x) {
  S1.push(x);
}
```

When we say remove data from Queue, it always means taking out the First element first and so on, as we have to follow the FIFO approach. But if we simply perform S1.pop() in our **dequeue** method, then it will remove the Last element first.




```
int  dequeue() {
  while(S1.isEmpty()) {
    x = S1.pop();
    S2.push(); }

    //removing the element
    x = S2.pop();
    while(!S2.isEmpty()) {
       x = S2.pop();
       S1.push(x); }
    return x;}
```

Example1: Draw the queue contents after each operation:

maxsize=4:enq(3), enq(4), enq(5), deq(), deq(), enq(6), enq(7), deq(), deq(), deq()

Example2: Draw the queue contents after each operation

maxsize=4:enq(3), enq(4), enq(5), deq(), deq(),deq()

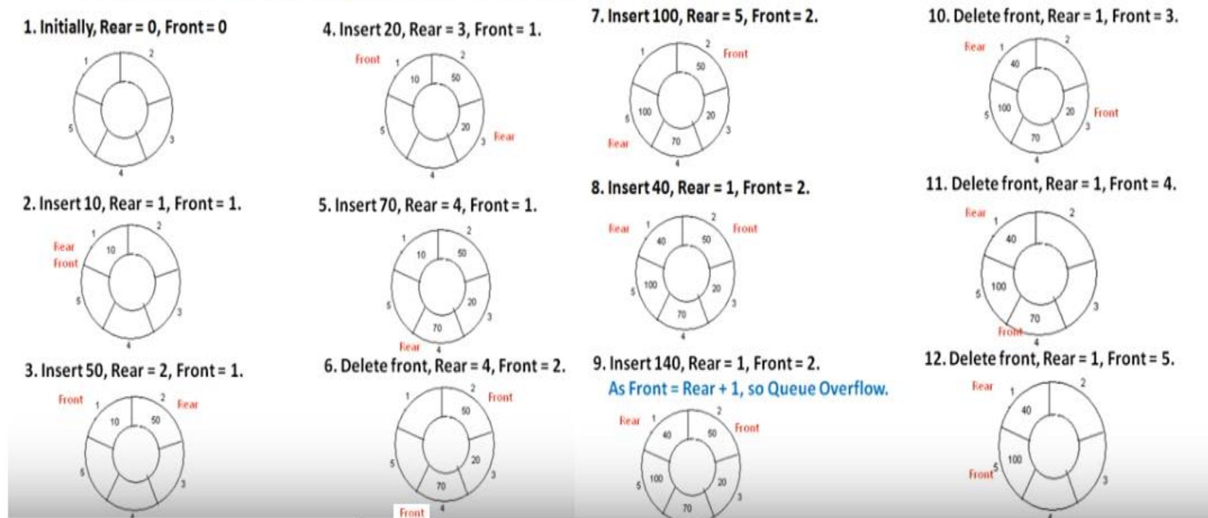
Note:

Insertion of element	**Deletion of element**
full: if front=-1 & rear=maxsize	empty: if front=rear
rear=rear+1	front=front+1
queue(rear)=item	item=queue[front]

Circular Queue

In this queue the rear point to the beginning of queue when it reaches end of the queue

insertion	**deletion**
queue(rear)=item	**item=queue(front)**
rear=(rear+1) mod maxzise	**front=(front+1) mod maxzise**
full: first==last && first !=-1	**empty: last==-1 && first==-1**



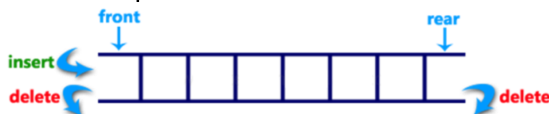
Deque(Double Ends Queue)

In this data structure insertion and delectation can be perform at both ends.

Input Restricted queue

insertion is performed at only one end(rear)

deletion is performed at both the ends.



Output Restricted queue

deletion is performed at only one end(rear)

insertion is performed at both the ends.