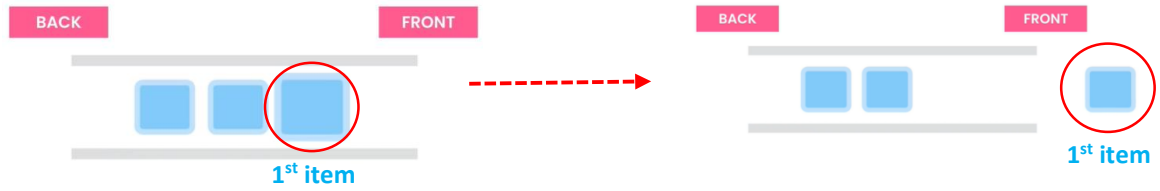
Queues

V-(1+2)-Introduction to Queue:- (Just Like Stack but (FIFO))



FIFO (First in First Out):- The basic difference between stack and queue is that. Stack = LIFO but Queue = FIFO.

The picture above clearly stated that the item that first inserted in the queue will remove first from the queue.

More RealLife Example:-

1. Printer : Use queue cause the first file that is selected for printing will be printed first.

2. Operating System: Use queue for processes as there are many CPU scheduling Algorithms (FCFS,Round Robin,SRTF,Priority) to optimize the time and space .

3. Web Servers: Use queue to manage incoming requests.

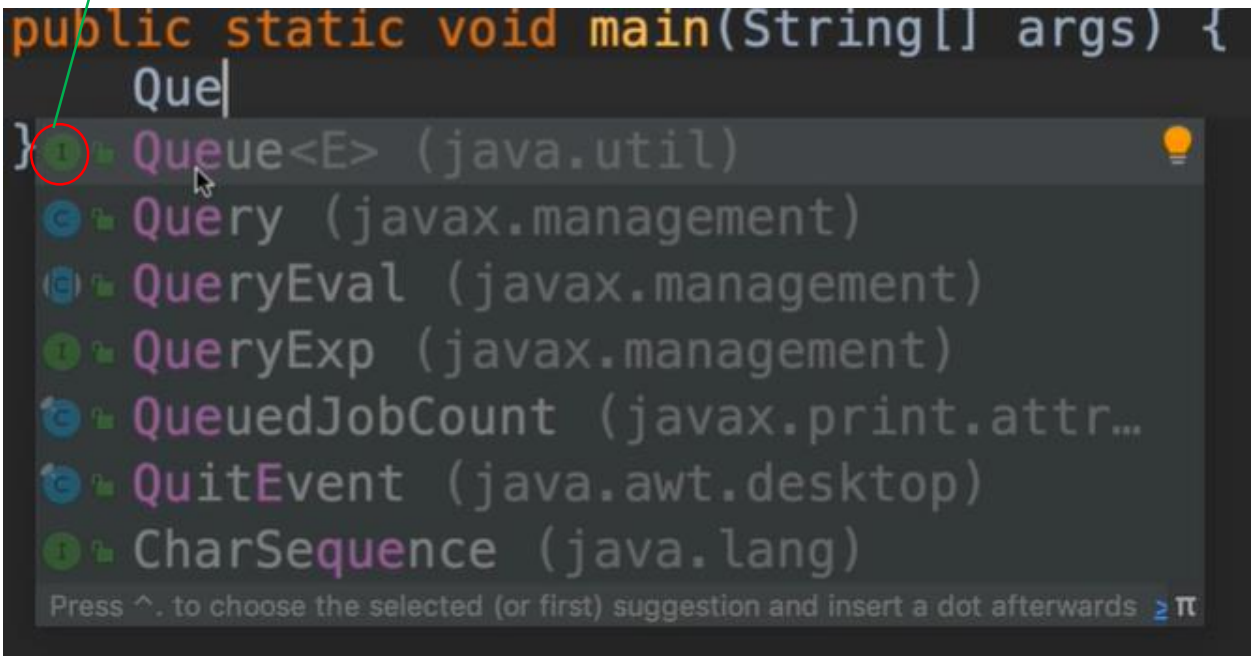
4. Live Supports System: In Live Support systems for managing customers they serve per customer requests (Serially).

OPERATIONS

enqueue	O(1) Adding items at the back of the queue
dequeue	O(1) Removing items from the front of the queue
peek	O(1) Getting items from the front of the queue without removing
isEmpty	O(1) Checking if queue is empty or not
isFull	O(1) Checking if queue is full or not

Its an interface. that means we can't instantiate Queue Object .

Queue<String> q = new Queue<>();[not possible]



All Known Implementing Classes:

AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, SynchronousQueue

---

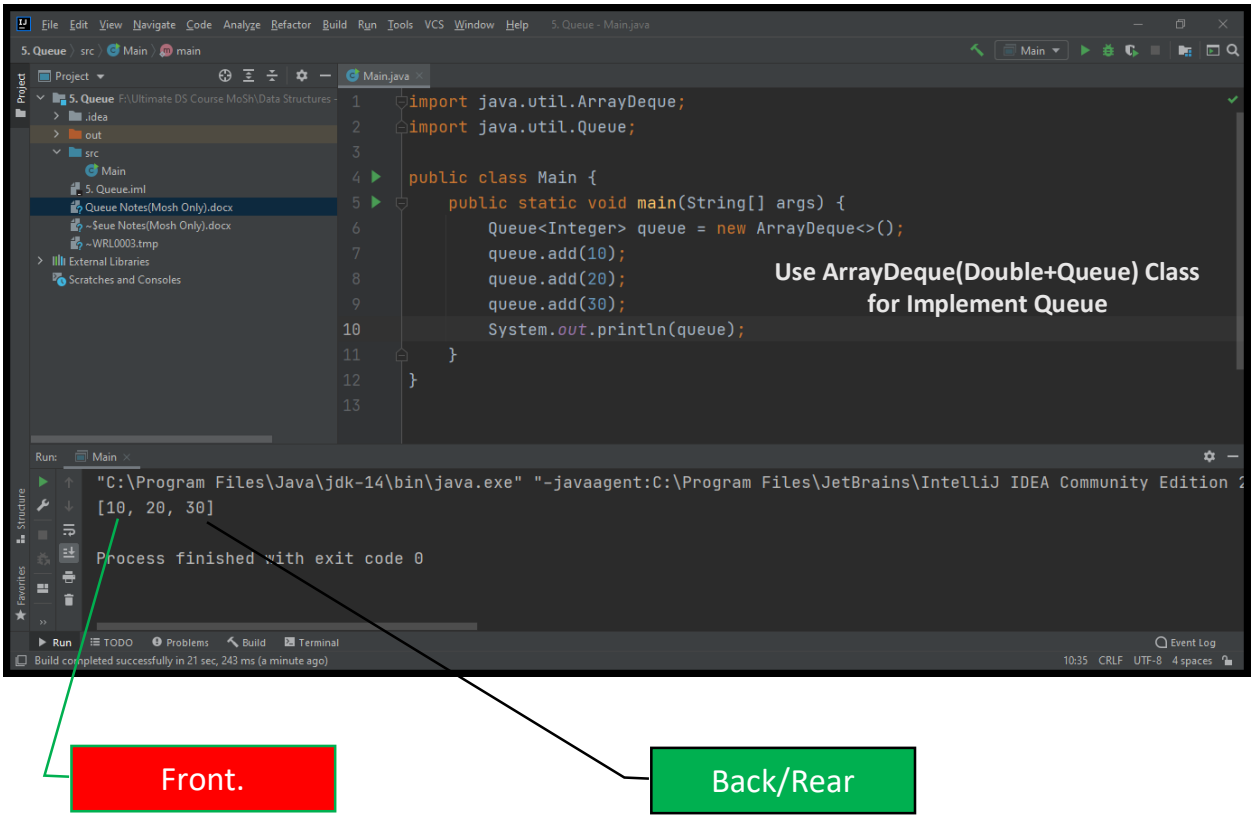
We see that many classes implemented Queue Interface. Among them 90% of our time we will use ArrayDeque and LinkedList Classes. In ArrayDeque class it has two ends (Front and Back).

Methods

Modifier and Type	Method and Description
boolean Queue Full then throw exception	`add(E e)` enque 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.
`E`	`element()` Retrieves, but does not remove, the head of this queue.
boolean Queue Full then Don't take that value.	`offer(E e)` enque Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.

E	No Exception occur, Just return null if queue is empty.	**poll()** Retrieves and removes the head of this queue, or returns `null` if this queue is empty.
E	Exception thrown for empty queue	**remove()** Retrieves and removes the head of this queue.

Queue Implementation(V-3):-




```
Queue<Integer> queue = new ArrayDeque<>();
queue.add(10);// [10]
queue.add(20);//[10,20]
queue.add(30);//[10,20,30]
System.out.println(queue);//[10,20,30]
System.out.println(queue.remove());//10 cause 10 added first
System.out.println(queue);//[20,30]
```

Output:-

```
[10, 20, 30]
10
[20, 30]
```

V-(4+5)-Reversing a queue:-(Use only add, remove ,isEmpty Method):-

```
public static void reverse(Queue<Integer> queue){
    //add , remove an isEmpty methods to build this
method
    if(queue.isEmpty()) throw new
IllegalArgumentException();
    // Idea is that we use both stack and queue to
implement this
    // Queue = [10,20,30]
    // Stack = []
    //Now remove every items from queue and push to stack
    // Then Stack = [10,20,30] and Queue = []
    // Again pop every items from stack and enqueue it to
the Queue
    // Then it will like Stack = [],Queue = [30,20,10]
    // We easily use stack to reverse anything cause it
is LIFO structured
    Stack<Integer> stack = new Stack<>();
    while (!queue.isEmpty())
        stack.push(queue.remove());
    while (!stack.empty())
        queue.add(stack.pop());

}
```

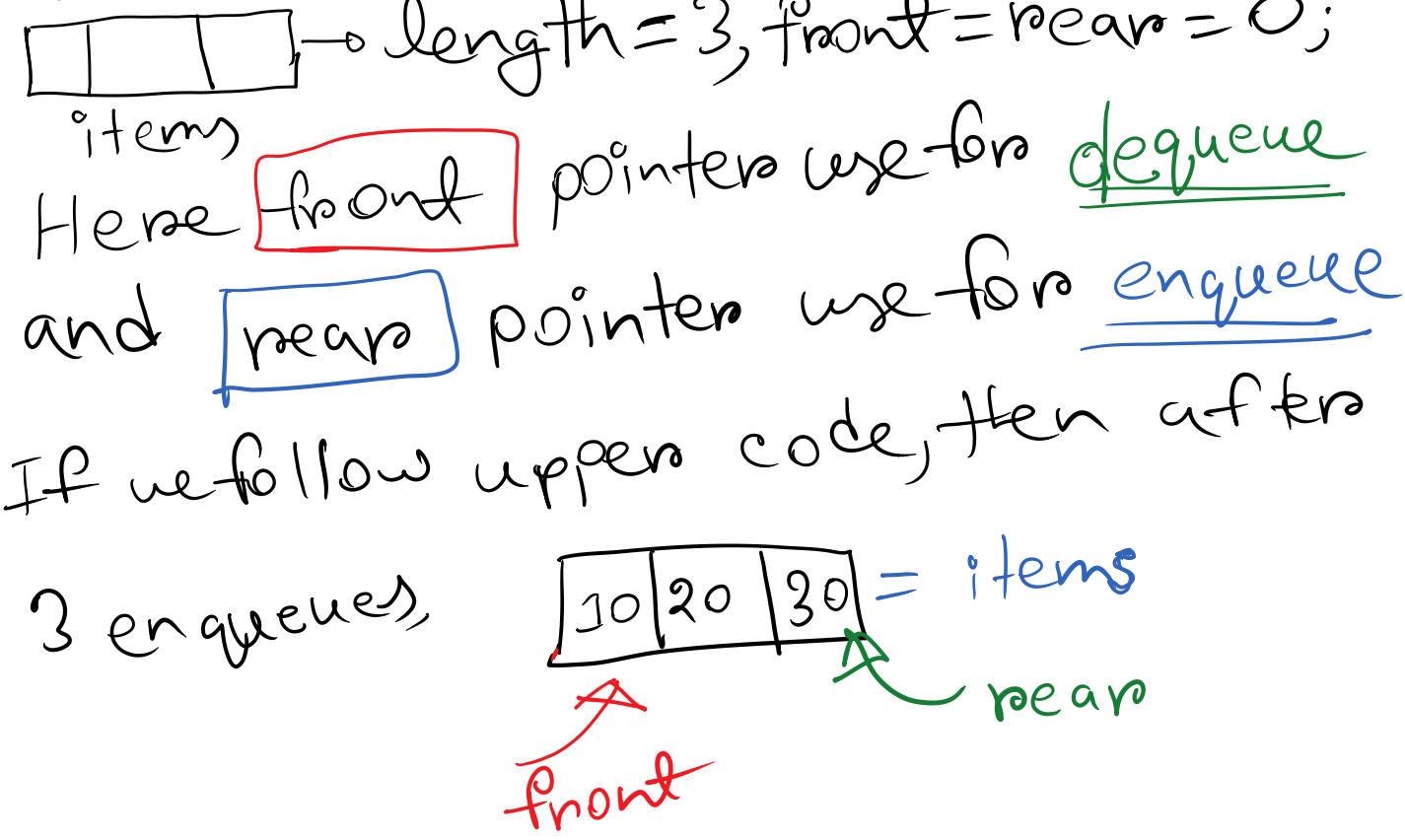
V-(6+7+8)-Building a queue using an array.(But we should also know about building a queue using linkedlist and stack to pass any interview)

```
public class Main {
    public static void main(String[] args) {
        // ArrayQueue (ArrayDeque)
        // enqueue
        // dequeue
        // peek
        // isEmpty
        // isFull
        // [10, 20, 30, 40, 50]
        //      F           R
        //  F = 0
        //  R = 3
    }
```

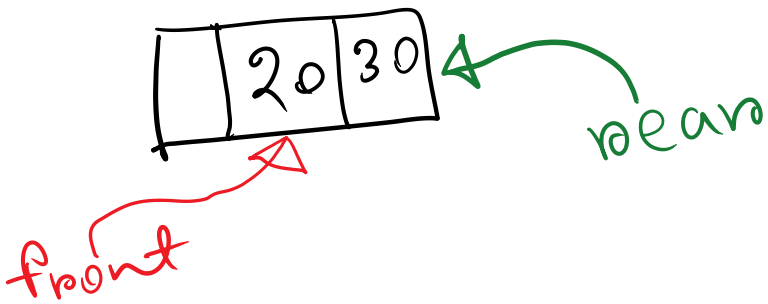
Use Two Pointers To implement

```
public void enqueue(int item) {
  if (count == items.length)
    throw new IllegalStateException();

  items[rear++] = item;
  count++;
}

public int dequeue() {
  var item = items[front];
  items[front++] = 0;
  count--;
  return item;
}
```

The Methods are not correct. Because rear will reach at the end of the array but we may have spaces left after dequeue. But once rear pointer will reach the end of the array it will give exception for next enqueue.



If we dequeue 1 item, then item array will look like below:-



We already see that, our rear reference where every time new element is going to be inserted is now at the end index of the items array. So, though we dequeue 10 from the front and we should insert 1 new item in the items array. But when we try to do that it will give us and exception (ArrayIndexoutofBound). Which is not right . So we have to fixed this using circular array.

Correct way of enqueue and dequeue is given below(Using circular array):-

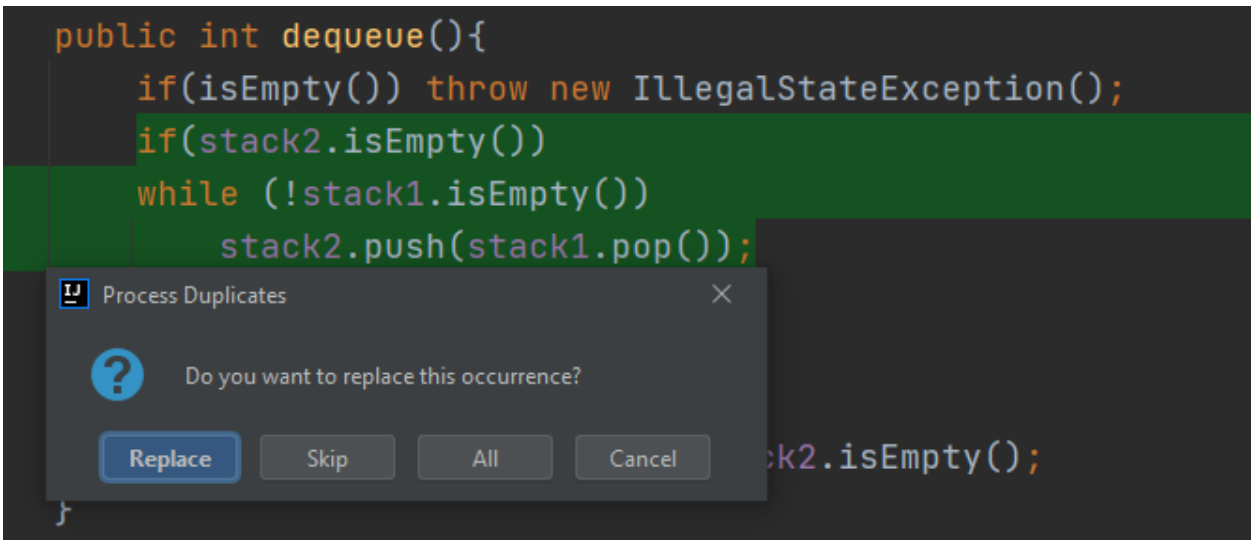
```
public void enqueue(int item){
    //Here rear is needed because in rear the items
inserted everytime
    if(isFull()) throw new IllegalStateException();
    queue[rear]=item;
    rear=(rear+1)%queue.length;
    count++;//it will help to count the items that is
//inserted
}
public int dequeue(){
   if(isEmpty()) throw new IllegalArgumentException();
   var first = queue[front];
   queue[front]=0;
   front = (front+1)%queue.length;
   count--;
   return first;
}
```

other supporting methods of implementation queue using an array:-

```
private boolean isEmpty(){
    return count==0;//count 0 means no items is inserted
}
private boolean isFull(){
    return count==queue.length;
}
```

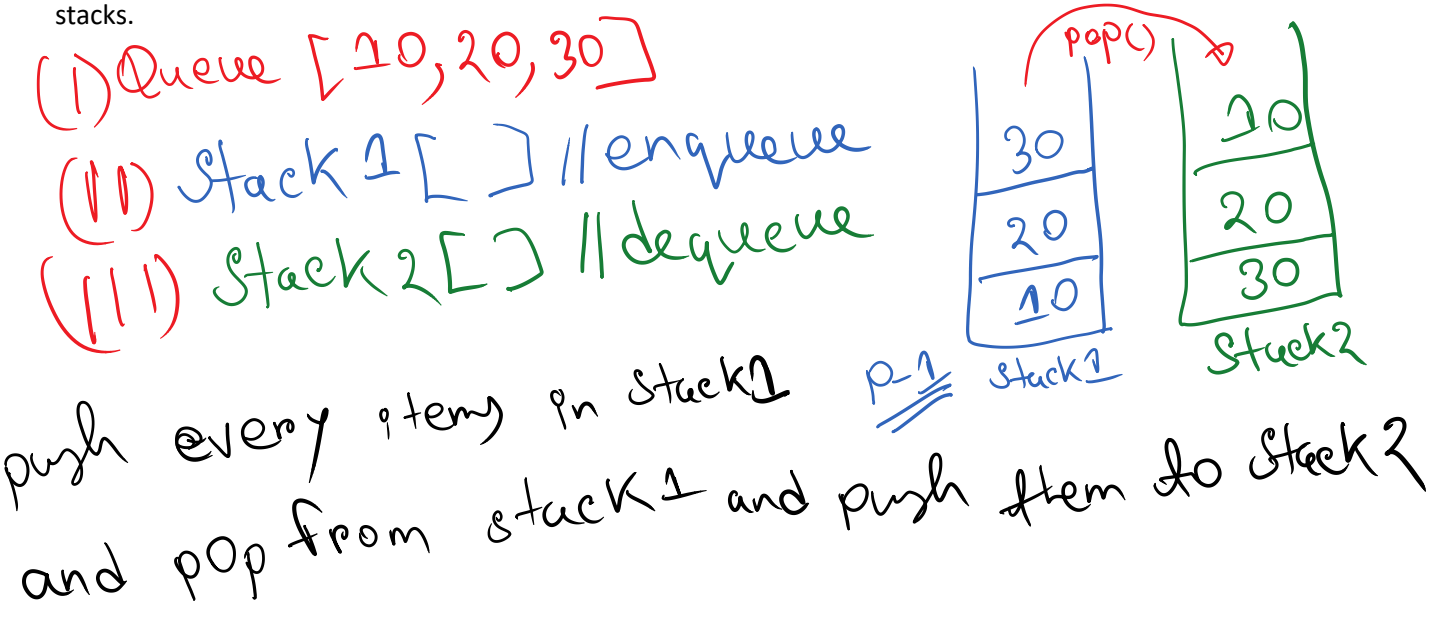
```java
public int peek(){
    if(isEmpty()) throw new IllegalStateException();
    return queue[front];
}
@Override
public String toString(){
    return Arrays.toString(queue);
}
```

The way of refactoring method if same lines of logic use multiple times(same like field extracting):-

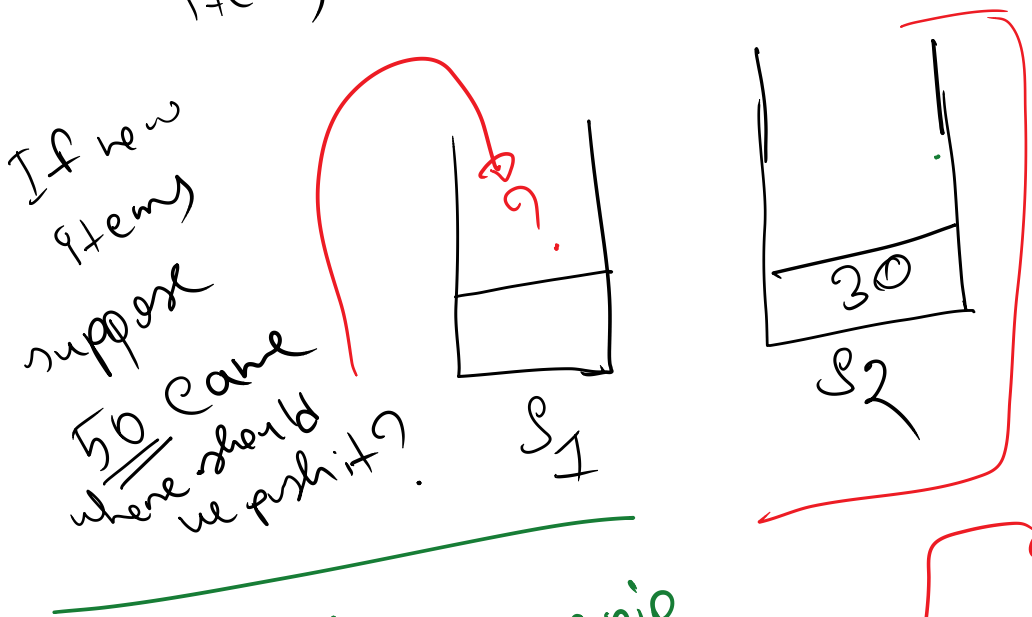


V-(9+10)-Build queue with Stack(ONLY):-

As stack is LIFO it is totally impossible to implement with one stack only. We have to use atleast two stacks.



Now there might be a problem when :-
InQueue[2] it is empty now. If new
items come these can be a confusion

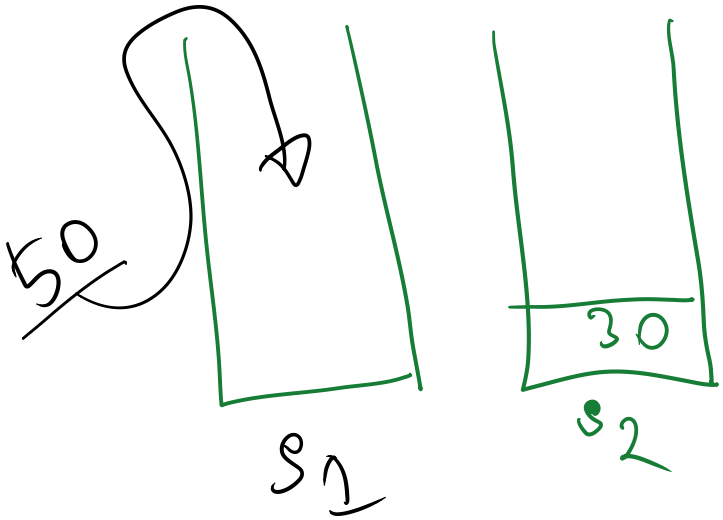


If we push 50 in S2

Queue [30, 50]

S2 [30, 50]

The right scenario would be



30 should be dequeue first but it is under 50 which breaks FIFO structure

So as long as S1 is empty we should push every new item in

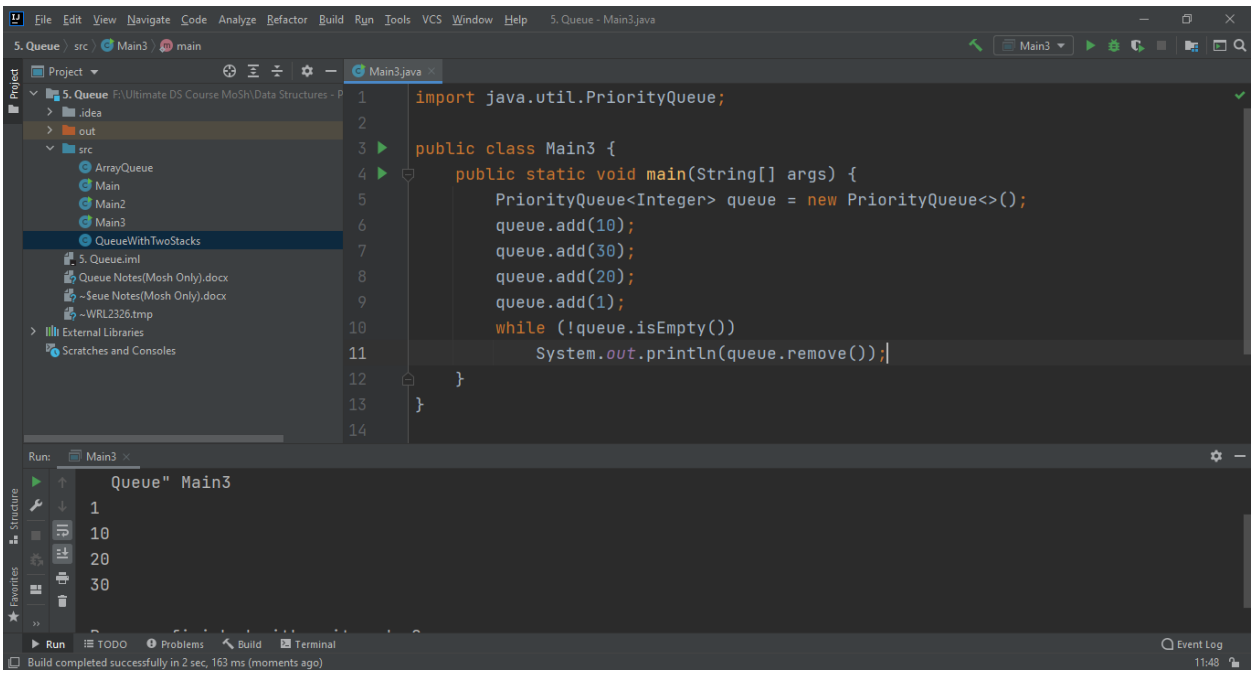
S_1), then reverse (pop from S_1 to S_2) then dequeue (pop) from S_2

```
import java.util.Stack;

public class QueueWithTwoStacks {
    // Popular Interview Question
    // Here we have to build queue with stacks
    // But there is a problem
    // queue is FIFO and Stack is LIFO
    // But we can use at least two Stacks to solve this
problem
    // One Stack will be dedicated for enqueue and other
one is for dequeue
    // suppose Q[10,20]
    // S1 for enqueue S1 [10,20]
    // S2 will store the value from S1 in reverse order
that means we have to pop from S1
    // S2 [20,10] now if we pop from S2 as dequeue then
we get 10 which is like queue FIFO
    // But there is a problem with dequeue
    // if S2 pop 10 then S2 = [20] , Here if we enqueue
30 now in S1=[20,30] it will confused us where to add
those and it will mess up the whole idea
    // so we have to pop all items from S1 and push it to
s2 then only it will be resolved
    Stack<Integer> stack1 = new Stack<>();// for enqueue
    Stack<Integer> stack2 = new Stack<>();// for dequeue
    public void enqueue(int item){
        stack1.push(item);// O(1)
    }
    public int dequeue(){
        if(isEmpty()) throw new IllegalStateException();
        moveStack1toStack2();
        return stack2.pop();
    }
    public boolean isEmpty(){
        return stack1.isEmpty() && stack2.isEmpty();
    }
    public int peek(){
        if(isEmpty()) throw new IllegalStateException();
        moveStack1toStack2();
        return stack2.peek();
    }
```

```
    private void moveStack1toStack2() {
        if (stack2.isEmpty())
            while (!stack1.isEmpty())
                stack2.push(stack1.pop());
    }


}
```

V-11-(Priority Queues):-[By default we see that items removed from this priority queue hasn't follow the traditional order, it is removed followed by ascending orders].

This sort of implementation is needed when we have to prioritize something . For ex- If we have to give token to various types of patients[emergency,normal] in hospital we have to give token follow prioritizing. Then another example can be if we have received 100000 mail and we have to find the important mails as soon as possible then we can prioritize them.



V-(12-14)-Building priority queues with array.[we can use heap also to implement that]

```
import java.util.Arrays;

public class CustomPriorityQueue {
    private int items[] = new int[5];
    private int count;
    public void add(int item){
     // As we set the priority of smaller number to be
inserted at the beginning
     // [0,0,0,0,0] items
     // add(5)
     // add(1)
     // add(3)
     // the array should be [1,3,5,0,0]
     // that means we have to iterate from the last item
of the array
     // if the item is less than the item exist in
current index
     // that means we have to do right shift and for not
missing any value we shift values from end of the array
     if(isFull())
         throw new IllegalStateException();
     var i = shiftItemsToInsert(item);
     items[i+1]=item;
     count++;
    }
    public boolean isFull(){
        return count== items.length;
    }
    public int shiftItemsToInsert(int item){
        int i;
        for(i=count-1;i>=0;i--){
            if(items[i]>item)
                items[i+1]=items[i];//shifting items
            else
                break;
        }
        return i+1;
    }
    public boolean isEmpty(){
        return count==0;
```

```
    }
    public int remove(){// considering highest number get
the higher priority
        if(isEmpty())
            throw new IllegalStateException();
        return items[--count];
    }
    @Override
    public String toString(){
        return Arrays.toString(items);
    }

}
```

Exercise Queue:-

Ex-1 Code answer:-

1- Given an integer K and a queue of integers, write code to reverse the order of the first K elements of the queue.

```
Input:  Q = [10, 20, 30, 40, 50], K = 3

Output: Q = [30, 20, 10, 40, 50]
```

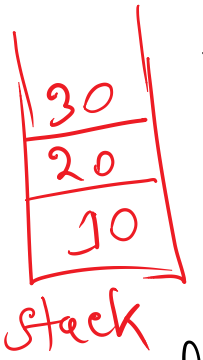
```
import java.util.Queue;

import java.util.Stack;

public class QueueReverser {

   public static void reverse(Queue<Integer> queue, int k) {

      if (k < 0 || k > queue.size())

         throw new IllegalArgumentException();


      java.util.Stack<Integer> stack = new Stack<>();


      // Dequeue the first K elements from the queue

      // and push them onto the stack

      for (int i = 0; i < k; i++)

         stack.push(queue.remove());


      // Enqueue the content of the stack at the

      // back of the queue
```

```
        while (!stack.empty())
            queue.add(stack.pop());

        // Add the remaining items in the queue (items
        // after the first K elements) to the back of the
        // queue and remove them from the beginning of the queue
        for (int i = 0; i < queue.size() - k; i++)
            queue.add(queue.remove());

    }
}
```

Simulation:-

[10,20,30,40,50] Queue:-

K=3, that first 3 elements needs to be reversed. We can use stack to solve this.

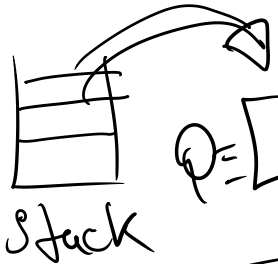


Now our Queue is

Q=[40,50]

Step-1 Just push the items (reversed) onto the stack.

Step-2 Add all the items in Queue that previously pushed on the stacks



Q=[40,50,30,20,10]

we see that the item reversed allready but first two items need to be in the back

Step-3

i < Queue.size()-k

প্রতিটি item নিয়ে Queue থেকে remove করে Queue এর back এ add করবো

So এখন Queue এর সামনে থাকা item remove করে সেটা Queue তে add করবো

code

```
for(int i=0 ; i < Queue.size()-k;i++){
    Queue.add(Queue.remove());
}
```

এখন Q=[30,20,10,40,50] ✓ done.

Exercise -2:-

2- Build a queue using a linked list from scratch. Implement the following operations and calculate their runtime complexities.

- enqueue
- dequeue
- peek
- size
- isEmpty

Ans:-

```
import java.util.ArrayList;

public class LinkedListQueue {

    private class Node{
        private Node next;
        private int value;
     public Node(int value){
         this.value=value;
     }
    }
    private Node head;
    private Node tail;
    private int count;
    //O(1)
    public void enqueue(int value){
        Node NewNode = new Node(value);
        if(isEmpty()){
            head=tail=NewNode;
        }
        else{
            tail.next=NewNode;
            tail = NewNode;
        }
        count++;
    }
```

```
//O(1)
    public int dequeue(){
        if(isEmpty())
            throw new IllegalArgumentException();
        int value;
        if(head==tail){
        value = head.value;
        head=tail=null;
        }
        else {
            value=head.value;
            var second = head.next;
            head.next=null;
            head=second;
        }
        return value;
    }
    // O(1)
    public boolean isEmpty() {
        return head == null;
    }
    //O(1)
    public int peek(){
        return head.value;
    }
    // O(1)
    public int size() {
        return count;
    }
    // O(n)
    public String toString() {
        ArrayList<Integer> list = new ArrayList<>();

        Node current = head;
        while (current != null) {
            list.add(current.value);
            current = current.next;
        }

        return list.toString();
    }
}
```

Exercise -3:-

3- Build a stack using two queues. Implement the following operations and calculate their runtime complexities.

- push

- pop
- peek
- size
- isEmpty

Ans:- We can implement Stacks using two queues. One is with making push operation heavy and another with making pop operation heavy.

Process -1 :- (Making Push Operation Heavy)

Conditions:-

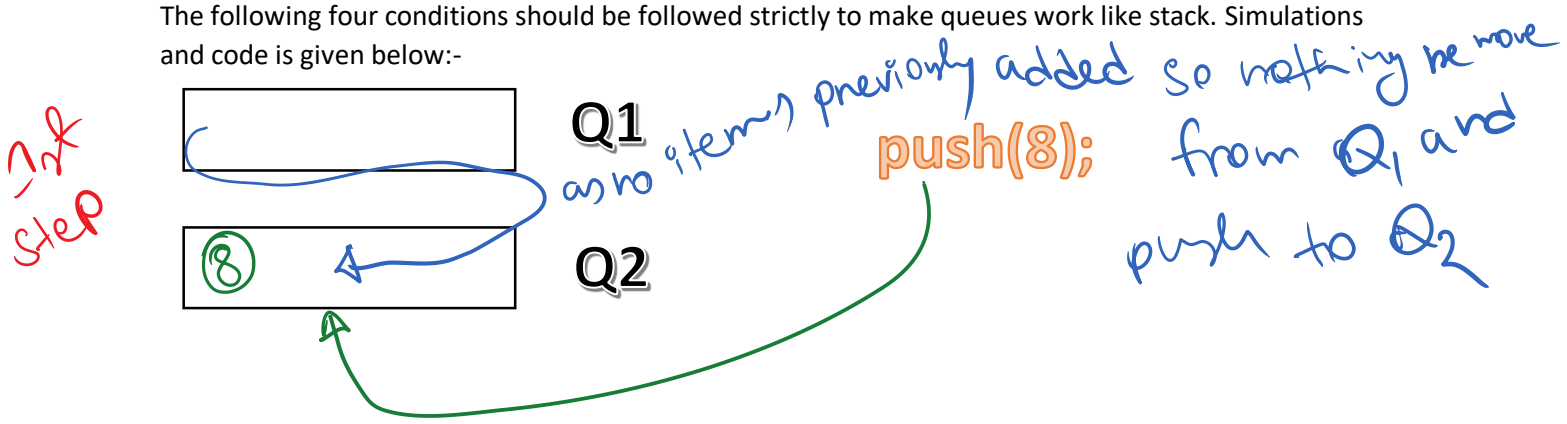
a) Always pop from q1.

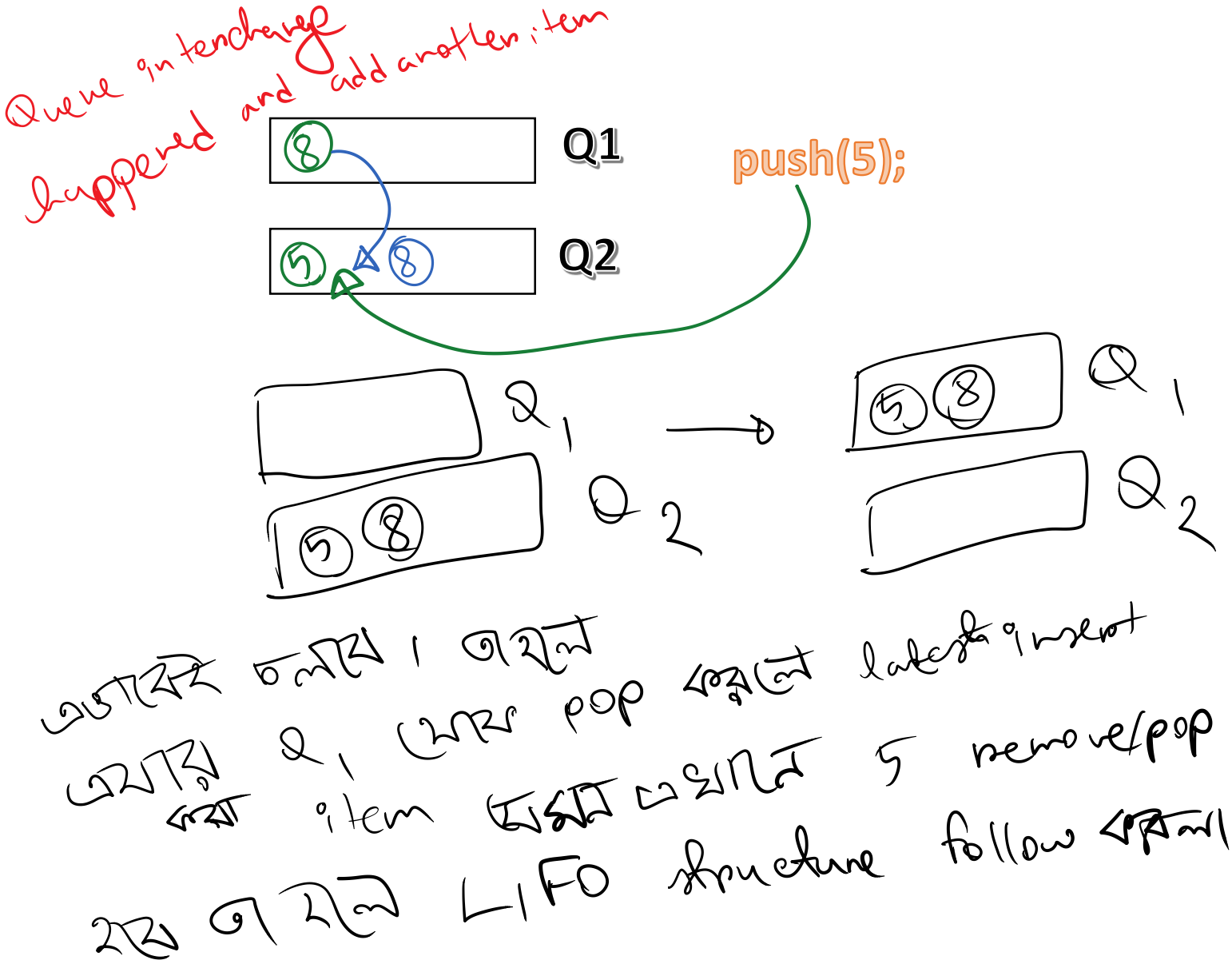
b) Any new item will always push to q2.

c) Push items (that remove from q1) to q2. [q2.add(q1.remove());]

d) Then exchange the queue.

The following four conditions should be followed strictly to make queues work like stack. Simulations and code is given below:-





এভাবে চলবে। তারপর যেহেতু Q1 থেকে pop করতে latest item insert করা হয়েছে তাই remove/pop হবে তারপর LIFO structure follow করবে।

Code:-

```
import java.util.LinkedList;
import java.util.Queue;
// video url :- https://youtu.be/XTouyMSE0Y8
public class StackWithQueues {
    private Queue<Integer> q1 = new LinkedList<>();
    private Queue<Integer> q2 = new LinkedList<>();
    private int size;
```

```
    public void enqueue1(int item){// O(n)
        q2.add(item);// always add items in q2 first
        size++;// increment size for adding new items
        // add all the datas from q1 to q2
        while (!q1.isEmpty())
            q2.add(q1.remove());
        // swap the queues
        Queue<Integer> temp = q1;
        q1 = q2;
        q2 = temp;
    }
    public int dequeue1(){
        if(isEmpty()) throw new IllegalArgumentException();

        size--;
        return q1.remove();
    }
    public int size(){
        return size;
    }
    public int peek1(){
        return q1.peek();
    }
    private boolean isEmpty(){
        return (q1.isEmpty() && q2.isEmpty());
    }

    public static void main(String[] args) {
        StackWithQueues s1 = new StackWithQueues();
        s1.enqueue1(10);
        s1.enqueue1(20);
        s1.enqueue1(30);
        System.out.println(s1.dequeue1());
        System.out.println(s1.dequeue1());
        System.out.println(s1.dequeue1());
      //  System.out.println(s1.dequeue1());
    }
}
```