**Implement Stack using Queues.**

**Intuition:**

As we know stack follows last in first out, which means we get the most recently inserted element whenever we remove an element from the stack. But queue follows first in first out, it means we get that element which we inserted in the starting at each deletion, it means if we want to use the queue like a stack we have to arrange elements in the queue such that we get the most recent element at each deletion.

**Approach:**

Take a single queue.

push(x): Push the element in the queue.

Use a for loop of size()-1, remove element from queue and again push back to the queue, hence the most recent element becomes the most former element and vice versa.

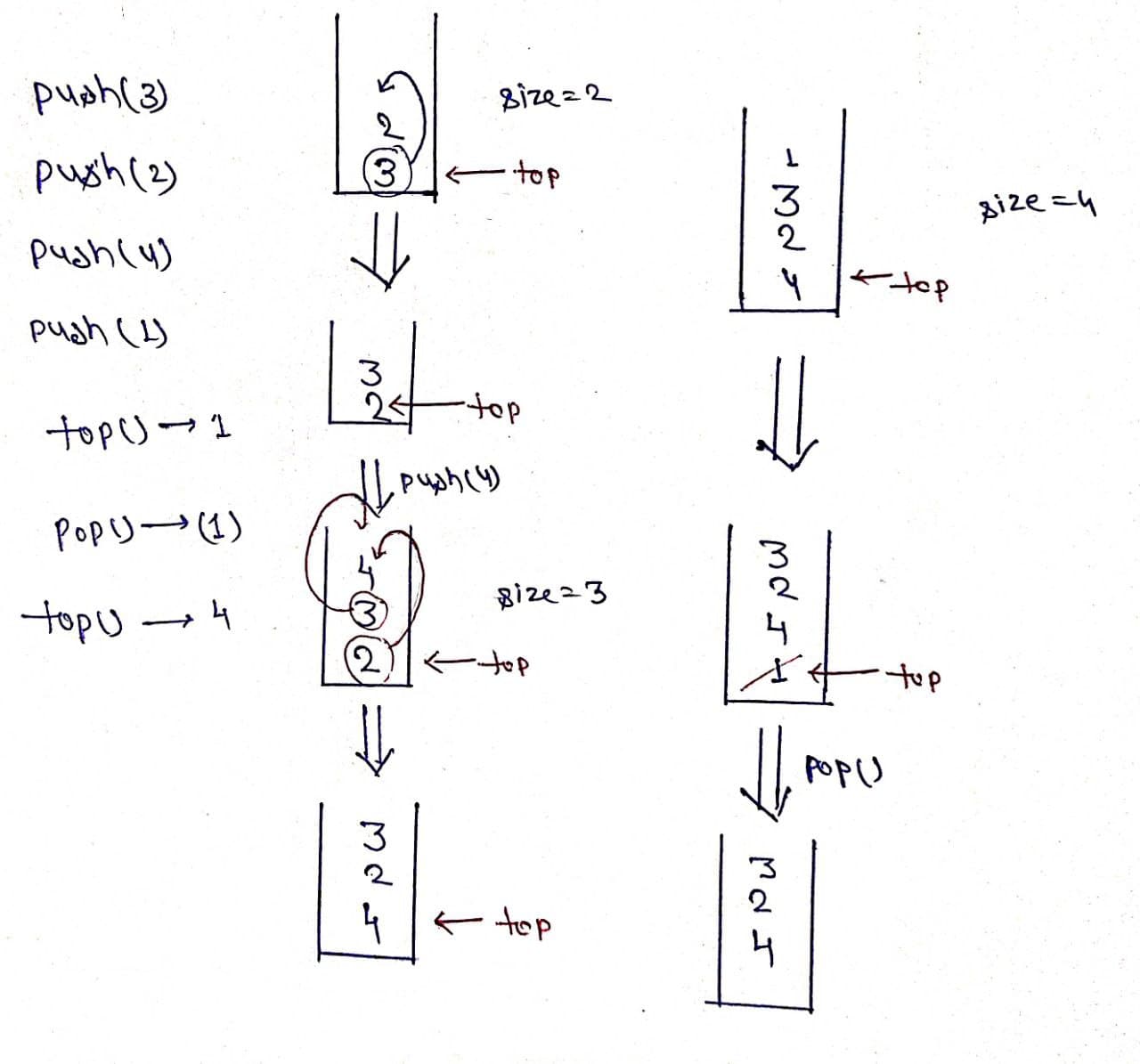
pop(): remove the element from the queue.

top(): show the element at the top of the queue.

size(): size of the current queue.

Repeat step3 at every insertion of the element.

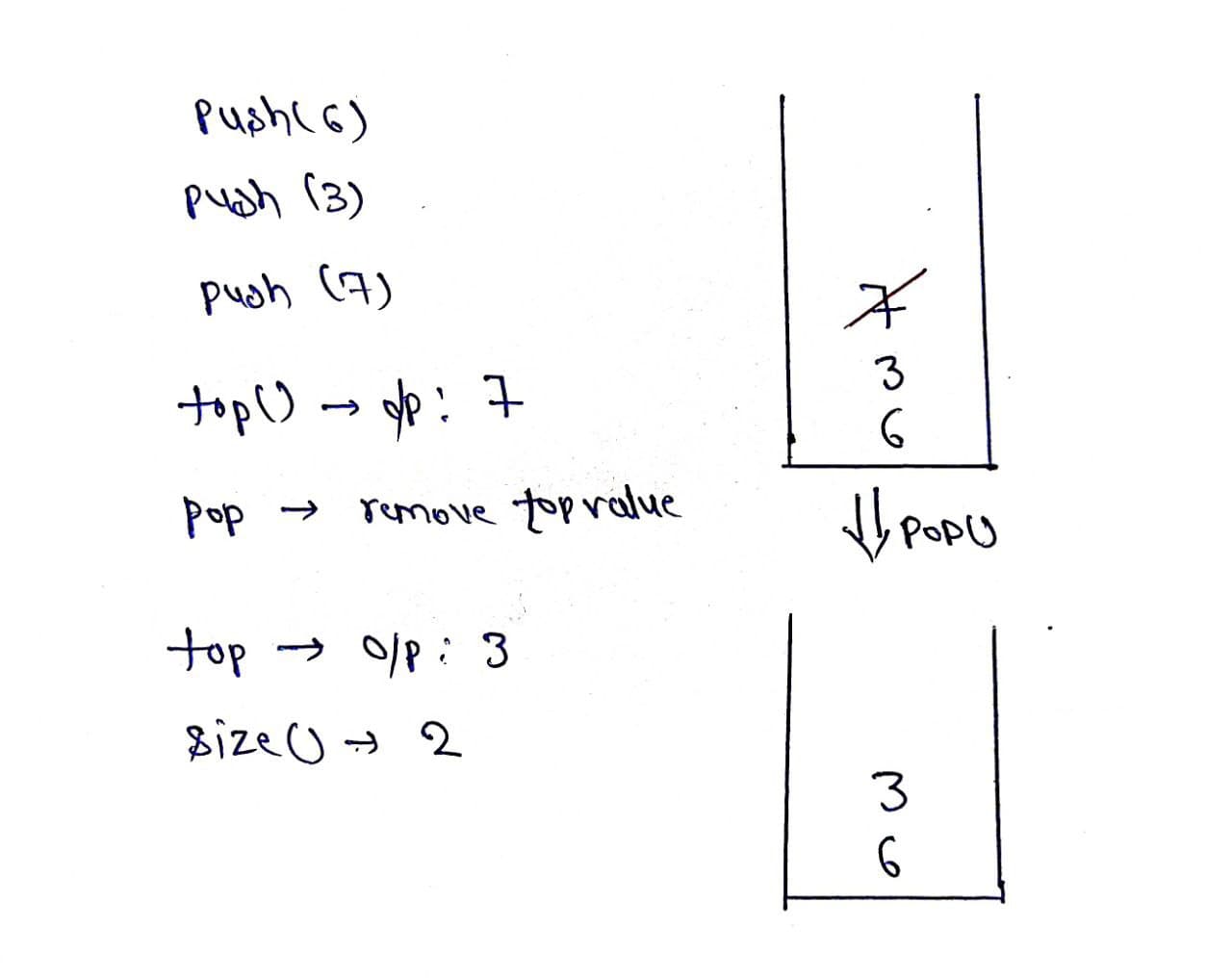
public class SuQ {  
 Queue<Integer> q;  
 public SuQ() {  
 q = new LinkedList<>();  
 }  
  
 public void push(int x) {  
 q.add(x);  
 for(int i=0;i<q.size()-1;i++)  
 {  
 q.add(q.poll());  
 }  
 }  
  
 public int pop() {  
 return q.remove();  
 }  
  
 public int top() {  
 return q.peek();  
 }  
  
 public boolean empty() {  
 return q.size()==0;  
 }  
   
}



**Time Complexity: O(N) Space Complexity: O(N)**

**Implement Stack using Arrays.**

Stack is a data structure that follows the Last In First Out (LIFO) rule.



**Explanation**:

push(): Insert the element in the stack.

pop(): Remove and return the topmost element of the stack.

top(): Return the topmost element of the stack

size(): Return the number of remaining elements in the stack.

Intuition: As we know stack works on the principle of last in first out, so we have to put elements in an array such that it keeps track of the most recently inserted element. Hence we can think of using a Top variable which will help in keeping track of recent elements inserted in the array.

Approach:

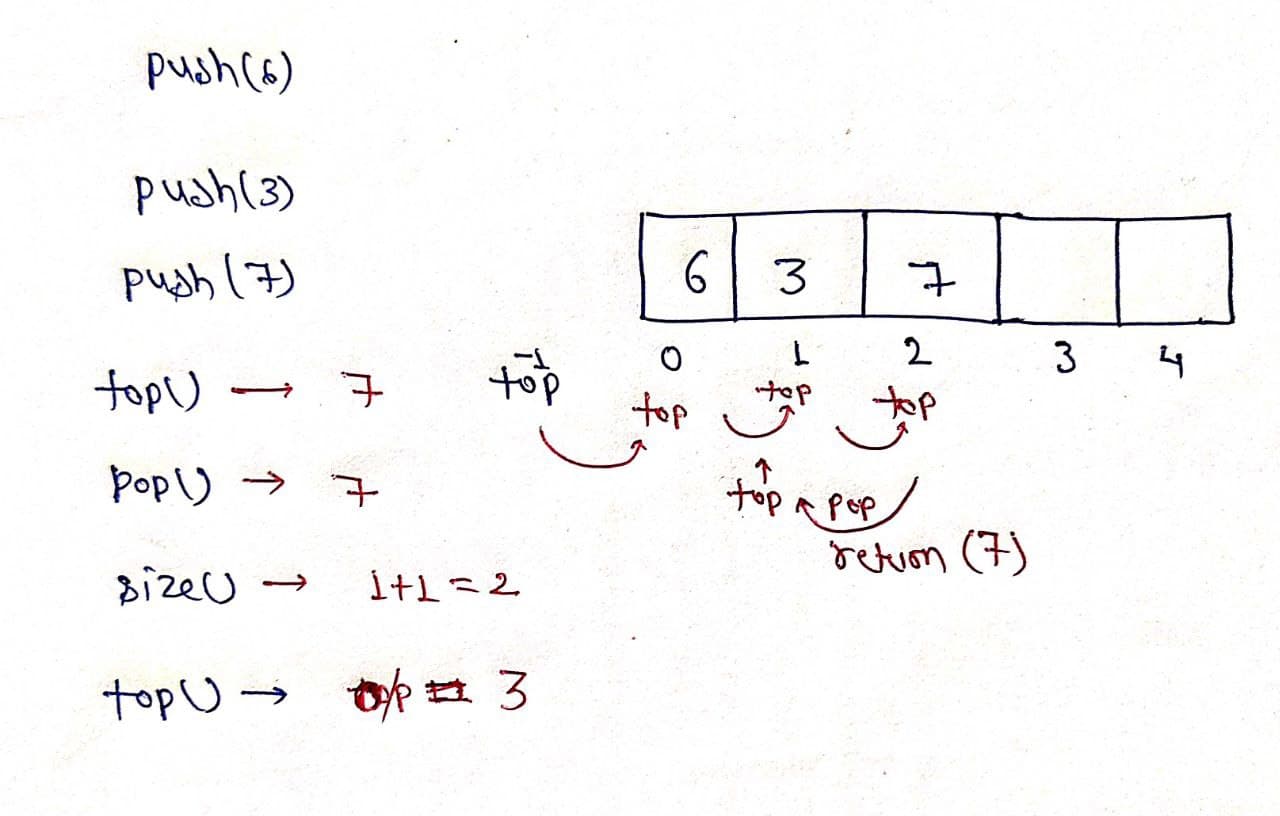
Declare an array of particular size.

Define a variable “Top” and initialize it as -1.

push(x): insert element is the array by increasing top by one.

pop(): check whether top is not equal to -1 if it is so, return top and decrease its value by one.

size(): return top+1.

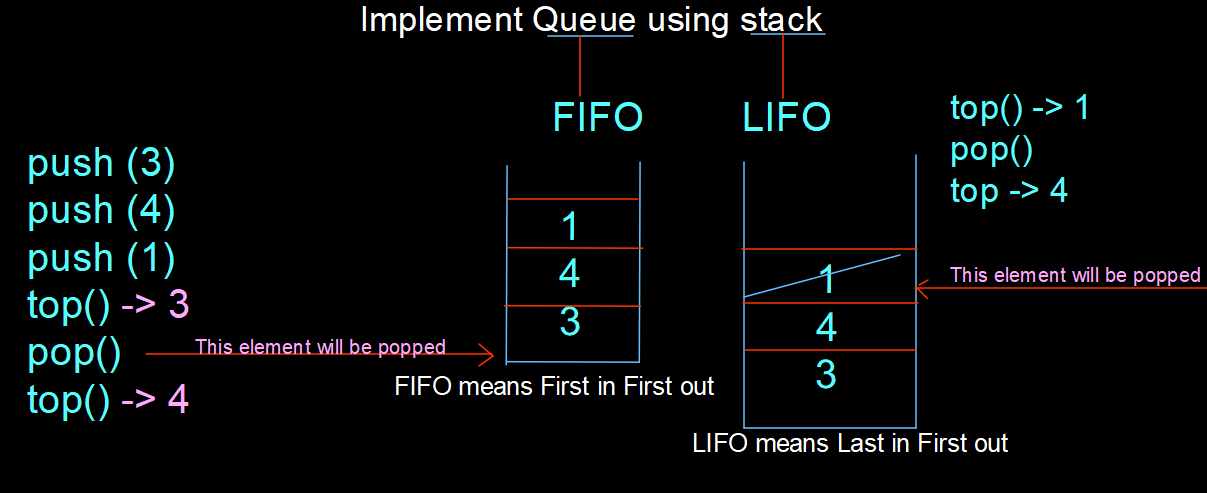
  
public class SuA {  
 private int[] arr;  
 private int top;  
  
 public SuA() {  
 arr = new int[1000];  
 top = -1;  
 }  
  
 public void push(int x) {  
 top++;  
 arr[top] = x;  
 }  
  
 public int pop() {  
 if(top==-1)return -1;  
 top--;  
 return arr[top+1];  
 }  
}

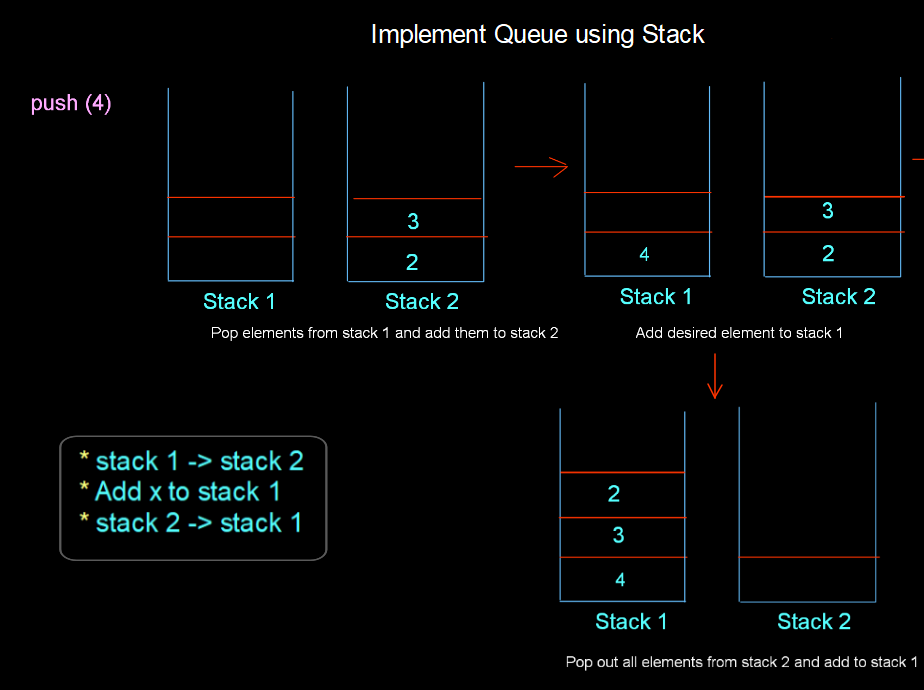
**Time Complexity: O(1) Space Complexity: O(N)**

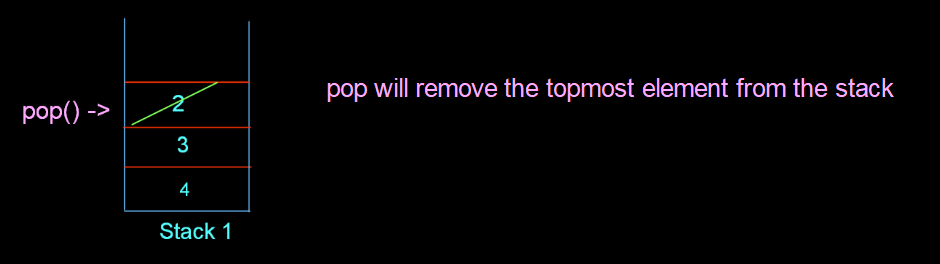
**Implement Queue using Stack.**

**Approach 1:**

class MyQueue {  
 Stack<Integer> st1,st2;  
 public MyQueue() {  
 st1 = new Stack<>();  
 st2 = new Stack<>();  
 }  
  
 public void push(int x) {  
 while(st1.isEmpty()==false)  
 {  
 st2.push(st1.pop());  
 }  
 st1.push(x);  
 while(st2.isEmpty()==false)  
 {  
 st1.push(st2.pop());  
 }  
 }  
  
 public int pop() {  
 return st1.pop();  
 }  
  
 public int peek() {  
 return st1.peek();  
 }  
  
 public boolean empty() {  
 return st1.isEmpty();  
 }  
}





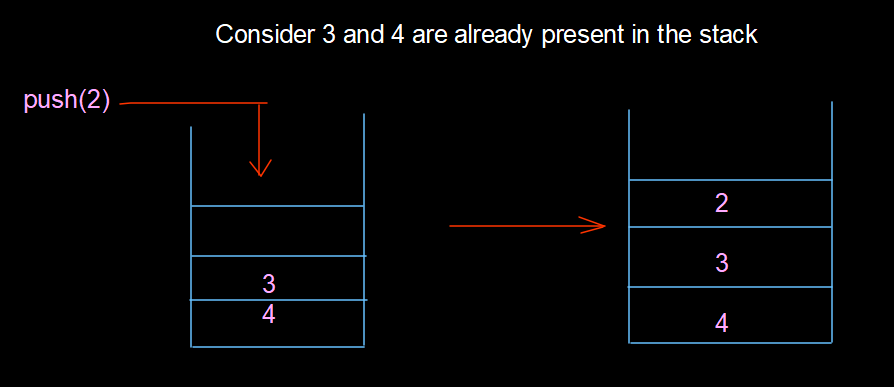


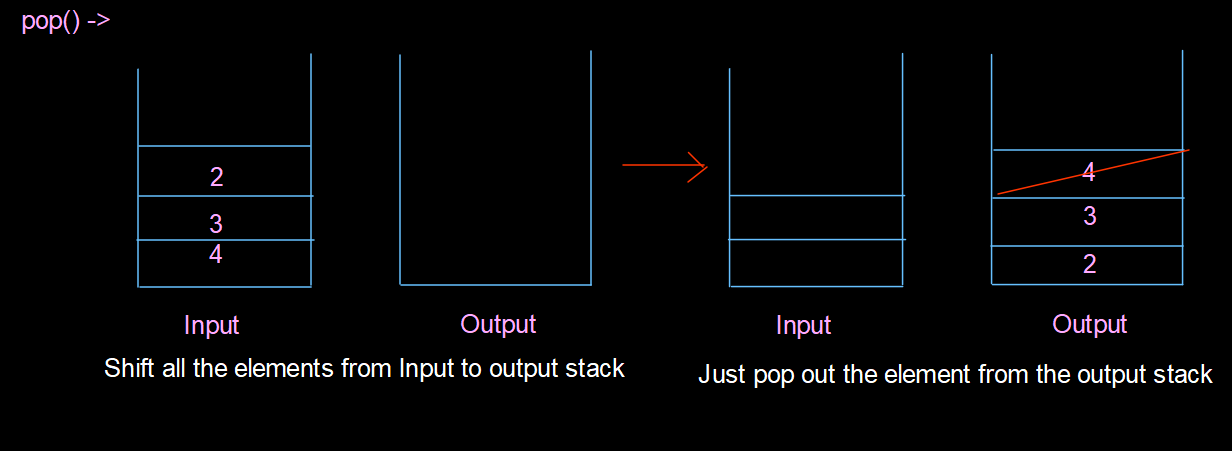


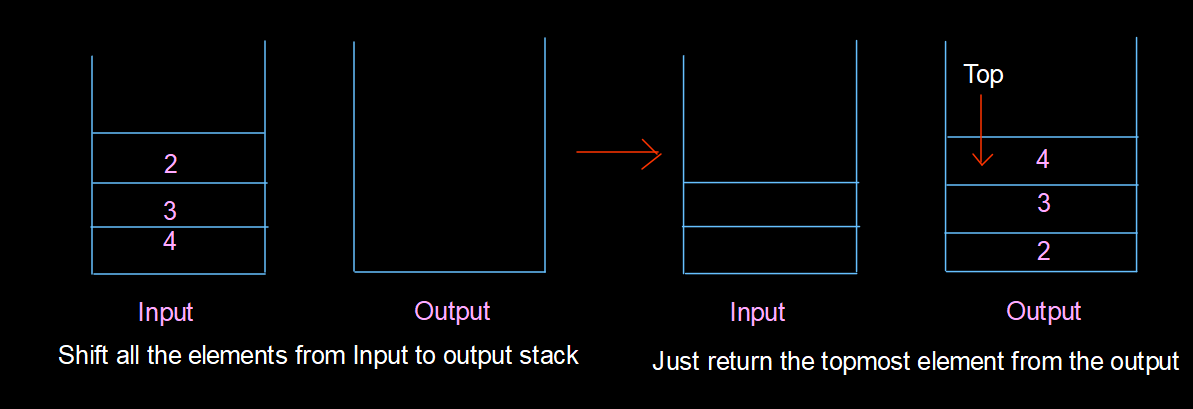
**Time Complexity: O(N) Space Complexity: O(2N)**

**Approach 2:**

class MyQueue {  
 Stack<Integer> st1,st2;  
 public MyQueue() {  
 st1 = new Stack<>();  
 st2 = new Stack<>();  
 }  
  
 public void push(int x) {  
 st1.push(x);  
 }  
  
 public int pop() {  
 if(st2.isEmpty())  
 {  
 while(st1.isEmpty()==false)  
 st2.push(st1.pop());  
 return st2.pop();  
  
 }  
 return st2.pop();  
 }  
  
 public int peek() {  
 if(st2.isEmpty())  
 {  
 while(st1.isEmpty()==false)  
 st2.push(st1.pop());  
 return st2.peek();  
 }  
 return st2.peek();  
 }  
  
 public boolean empty() {  
 if(st1.isEmpty()&&st2.isEmpty())return true;  
 return false;  
 }  
}







**Time Complexity: O(1) Space Complexity: O(2N)**

**Implement Queue using Arrays.**

**Intuition:**

The intuition is to fill the array in a circular manner, (ie) after popping from the front, rather than moving all the elements towards the front. We can have 2 variables to keep track of the start and end indexes of the sequence. Mod addition is done to handle boundary conditions.

**Approach:**

The basic approach is to maintain two variables to point to the START and END of the filled elements in the array. START pointer is used to point to the starting index of the elements and the same case for the END pointer(ending index). Initially, both have value -1(indicating empty queue).

First, let's see the implementation of the push function. Push basically inserts a new element at the end. So only the END variable is going to be incremented.

Corner case 1: What if we have no empty places in the array? So, first check that, if we don't have we exit, in the other case we increment the START variable and put the new element.

Corner case 2: What if END reaches the last index? We are doing mod with addition. So, END goes back to index 0([0-(n-1)] will always be the range for END).

Second, let us see the pop function. In Queue pop removes and returns the front element. So, START needs to be modified. The general approach is to copy the current element pointed by START and increase the START variable to the next index.

Corner case 3: What if the Queue is empty? That's why we are checking the START variable. If it is -1, then the queue is empty, we just exit.

Corner case 4: What if START goes out of bound? As done for END, mod addition comes to the rescue.

Corner case 5: What happens after popping the last element? We check this state with the currSize variable. Queue returns to the initial state, both START and END are set to -1.

Third, let's see the top function. It behaves more like a pop function. We need to return the element pointed by the START variable. Since we are not actually removing any element, it's fine to ignore corner cases 4 and 5.

That's all about the Queue class implementation. In the main function, we just initialize the Queue class to check all corner cases.

class Queue {  
  
 private int arr[];  
 private int start, end, currSize, maxSize;  
 public Queue() {  
 arr = new int[16];  
 start = -1;  
 end = -1;  
 currSize = 0;  
 }  
  
 public Queue(int maxSize) {  
 this.maxSize = maxSize;  
 arr = new int[maxSize];  
 start = -1;  
 end = -1;  
 currSize = 0;  
 }  
 public void push(int newElement) {  
 if (currSize == maxSize) {  
 System.out.println("Queue is full\nExiting...");  
 System.exit(1);  
 }  
 if (end == -1) {  
 start = 0;  
 end = 0;  
 } else  
 end = (end + 1) % maxSize;  
 arr[end] = newElement;  
 System.out.println("The element pushed is " + newElement);  
 currSize++;  
 }  
 public int pop() {  
 if (start == -1) {  
 System.out.println("Queue Empty\nExiting...");  
 System.exit(1);  
 }  
 int popped = arr[start];  
 if (currSize == 1) {  
 start = -1;  
 end = -1;  
 } else  
 start = (start + 1) % maxSize;  
 currSize--;  
 return popped;  
 }  
 public int top() {  
 if (start == -1) {  
 System.*out*.println("Queue is Empty");  
 System.*exit*(1);  
 }  
 return arr[start];  
 }  
 public int size() {  
 return currSize;  
 }  
  
}  
  
public class TUF {  
 public static void main(String args[]) {  
 Queue q = new Queue(6);  
 q.push(4);  
 q.push(14);  
 q.push(24);  
 q.push(34);  
 System.*out*.println("The peek of the queue before deleting any element " + q.top());  
 System.out.println("The size of the queue before deletion " + q.size());  
 System.out.println("The first element to be deleted " + q.pop());  
 System.out.println("The peek of the queue after deleting an element " + q.top());  
 System.out.println("The size of the queue after deleting an element " + q.size());  
 }  
}

**Time Complexity: O(1) Space Complexity: O(N)**