
Stack and Queue(Theory)

Stack understanding

- What is ?
- Implementation using Arrays
- Implementation using LinkedList
- Implementation using Queues

Queue (FIFO)

- What is ?
- Implementation using Arrays
- Implementation using LinkedList
- Implementation using Stack

Stack :

a linear data structure that follows the **Last In, First Out (LIFO)** principle.

- **Operations**:
  - **Push**: Adds an element to the top of the stack.
  - **Pop**: Removes and returns the top element of the stack.
  - **Peek/Top**: Returns the top element without removing it.
  - **isEmpty**: Checks if the stack is empty.
  - **isFull**: Checks if the stack is full (in case of a fixed-size stack).
- **Time Complexity**:
  - Push: **O(1)**
  - Pop: **O(1)**
  - Peek: **O(1)**
  - isEmpty: **O(1)**
  - isFull: **O(1)**
  - **size()**
- **Implementation**:
  - Can be implemented using **arrays** or **linked lists**.
  - Array-based stacks have a fixed size, while linked list-based stacks are dynamic.
- **Key Points**:
  - Stacks are efficient for adding and removing elements from one end (top).
  - They are used in scenarios where the order of operations matters.
  - Stack overflow occurs when pushing to a full stack (in fixed-size implementations).
  - Stack underflow occurs when popping from an empty stack.
- **Example Use Case**:
  - Reversing a string using a stack.

    - Balancing parentheses in an expression.

```
import java.util.Stack;

public class StackExample {
   public static void main(String[] args) {
      // Create a stack
      Stack<Integer> stack = new Stack<>();

      // Push elements onto the stack
      stack.push(10);
      stack.push(20);
      stack.push(30);

      // Peek at the top element
      System.out.println("Top element: " + stack.peek()); // Output: 30

      // Pop elements from the stack
      System.out.println("Popped element: " + stack.pop()); // Output: 30
      System.out.println("Popped element: " + stack.pop()); // Output: 20

      // Check if the stack is empty
      System.out.println("Is stack empty? " + stack.isEmpty()); // Output: false

      // Push another element
      stack.push(40);
      System.out.println("Top element after push: " + stack.peek()); // Output: 40
   }
}
```

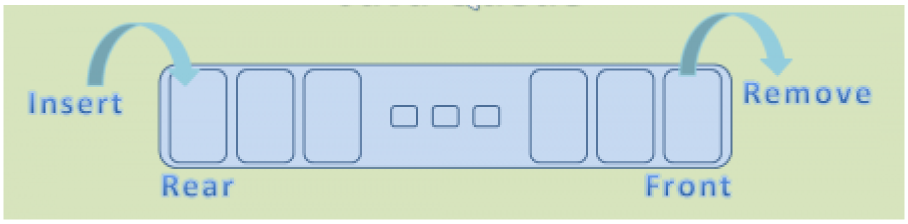
```
Top of Stack
   ↓
+-------+
|  30  | ← Last element pushed (will be the first to pop)
+-------+
|  20  |
+-------+
|  10  | ← First element pushed (will be the last to pop)
+-------+
```

Queue Overview in Java

- **Definition**:
    - A queue is a linear data structure that follows the **First In, First Out (FIFO)** principle.
    - The first element added is the first one to be removed.

- **Operation**

Operation	*Throws exception*	*Special value*
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()



- **Key Methods in Java's Queue Interface**:
    - `add(E e)`: Inserts the specified element into the queue (throws an exception if the queue is full).
    - `offer(E e)`: Inserts the specified element into the queue (returns `false` if the queue is full).
    - `remove()`: Removes and returns the element at the front of the queue (throws an exception if the queue is empty).
    - `poll()`: Removes and returns the element at the front of the queue (returns `null` if the queue is empty).
    - `element()`: Retrieves the element at the front without removing it (throws an exception if the queue is empty).
    - `peek()`: Retrieves the element at the front without removing it (returns `null` if the queue is empty).
    - size():

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

public class QueueExample {
   public static void main(String[] args) {
      // Create a queue using LinkedList
      Queue<Integer> queue = new LinkedList<>();

      // Add elements using offer()
      queue.offer(10); // Queue: [10]
      queue.offer(20); // Queue: [10, 20]
      queue.offer(30); // Queue: [10, 20, 30]

      // Peek at the front element
      System.out.println("Front element: " + queue.peek()); // Output: 10

      // Remove elements using poll()
      System.out.println("Polled element: " + queue.poll()); // Output: 10, Queue: [20, 30]
```

```
        // Check if the queue is empty
        System.out.println("Is queue empty? " + queue.isEmpty()); // Output: false

        // Get the size of the queue
        System.out.println("Queue size: " + queue.size()); // Output: 2
    }
}
```

Stack using the Arrays ..

```
public class StackUsingArray {
    private int maxSize; // Maximum size of the stack
    private int[] stackArray; // Array to store stack elements
    private int top; // Index of the top element

    // Constructor to initialize the stack
    public StackUsingArray(int size) {
        this.maxSize = size;
        this.stackArray = new int[maxSize];
        this.top = -1; // Stack is initially empty
    }
```

```
    // Push operation: Add an element to the top of the stack
    public void push(int value) {
        if (isFull()) {
            System.out.println("Stack is full. Cannot push " + value);
            return;
        }
        stackArray[++top] = value; // Increment top and insert value
    }
```

```
    // Pop operation: Remove and return the top element from the stack
    public int pop() {
        if (isEmpty()) {
            System.out.println("Stack is empty. Cannot pop.");
            return -1; // Return a default value indicating underflow
        }
        return stackArray[top--]; // Return top value and decrement top
    }
```

```
    // Peek operation: Return the top element without removing it
    public int peek() {
        if (isEmpty()) {
            System.out.println("Stack is empty. Cannot peek.");
            return -1; // Return a default value indicating underflow
        }
        return stackArray[top]; // Return the top element
    }
```

```
    // Check if the stack is empty
    public boolean isEmpty() {
        return (top == -1);
    }
```

```
    // Check if the stack is full
    public boolean isFull() {
        return (top == maxSize - 1);
    }
}
```

Main class

```
public class Main {
    public static void main(String[] args) {
        // Create a stack of size 5
        StackUsingArray stack = new StackUsingArray(5);

        // Push elements onto the stack
        stack.push(10);
        stack.push(20);
        stack.push(30);

        // Peek at the top element
        System.out.println("Top element: " + stack.peek()); // Output: 30

        // Pop elements from the stack
        System.out.println("Popped element: " + stack.pop()); // Output: 30
        System.out.println("Popped element: " + stack.pop()); // Output: 20

        // Check if the stack is empty
        System.out.println("Is stack empty? " + stack.isEmpty()); // Output: false

        // Push another element
        stack.push(40);
        System.out.println("Top element after push: " + stack.peek()); // Output: 40

        // Test edge cases
        stack.pop(); // Output: 40
        stack.pop(); // Output: 10
        stack.pop(); // Output: Stack is empty. Cannot pop.
        stack.peek(); // Output: Stack is empty. Cannot peek.
        stack.push(50);
        stack.push(60);
        stack.push(70);
        stack.push(80);
        stack.push(90);
        stack.push(100); // Output: Stack is full. Cannot push 100
    }
}
```

Output

```
Top element: 30
Popped element: 30
Popped element: 20
Is stack empty? false
Top element after push: 40
Stack is empty. Cannot pop.
Stack is empty. Cannot peek.
Stack is full. Cannot push 100
```

Implementation of Queue using Array ..

```
class Queue {
    private int arr[];
    private int start, end, currSize, maxSize;

    // Default constructor
    public Queue() {
        arr = new int[16];
        start = -1;
        end = -1;
        currSize = 0;
    }

    // Parameterized constructor
    public Queue(int maxSize) {
        this.maxSize = maxSize;
        arr = new int[maxSize];
        start = -1;
        end = -1;
        currSize = 0;
    }
    // Push operation: Add an element to the queue
    public void push(int newElement) {
        if (currSize == maxSize) {
            System.out.println("Queue is full. Cannot push " + newElement);
            return;
        }
        if (currSize == 0) { // Queue is empty
            start = 0;
            end = 0;
        } else {
            end = (end + 1) % maxSize; // Circular increment
        }
        arr[end] = newElement;
        currSize++;
        System.out.println("The element pushed is " + newElement);
    }

    // Pop operation: Remove and return the front element
    public int pop() {
        if (currSize==0) { // Queue is empty
            System.out.println("Queue is empty. Cannot pop.");
            return -1; // Return a default value indicating underflow
        }
        int popped = arr[start];
        if (currSize == 1) { // Only one element in the queue
            start = -1;
            end = -1;
        } else {
            start = (start + 1) % maxSize; // Circular increment
        }
        currSize--;
        return popped;
    }

    // Peek operation: Return the front element without removing it
    public int peek() {
        if (start == -1) { // Queue is empty
            System.out.println("Queue is empty. Cannot peek.");
            return -1; // Return a default value indicating underflow
        }
        return arr[start];
    }

    // Return the current size of the queue
    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.peek());
        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.peek());
        System.out.println("The size of the queue after deleting an element: " + q.size());

        // Test edge cases
        q.pop(); // 14
        q.pop(); // 24
        q.pop(); // 34
        q.pop(); // Queue is empty. Cannot pop.
        q.peek(); // Queue is empty. Cannot peek.
        q.push(50);
        q.push(60);
        q.push(70);
        q.push(80);
        q.push(90);
        q.push(100);
        q.push(110); // Queue is full. Cannot push 110
    }
}
```

Output :

```
 The element pushed is 4
The element pushed is 14
The element pushed is 24
The element pushed is 34
The peek of the queue before deleting any element: 4
The size of the queue before deletion: 4
The first element to be deleted: 4
The peek of the queue after deleting an element: 14
The size of the queue after deleting an element: 3
Queue is empty. Cannot pop.
Queue is empty. Cannot peek.
The element pushed is 50
The element pushed is 60
The element pushed is 70
The element pushed is 80
The element pushed is 90
The element pushed is 100
Queue is full. Cannot push 110
```

Time complexity of operation

Operation	Time Complexity
push(E o)	O(1)
pop()	O(1)
peek()	O(1)
size()	O(1)

Stack Using LL

- Advantage of stack using linked list is that in Dynamic size.

```
class stack {


private class stackNode {
   int data;
   stackNode next;
   stackNode(int d) {
     data = d;
     next = null;
   }
 }
 stackNode top;
 int size;
 stack() {
   this.top = null;
   this.size = 0;
 }


 public void stackPush(int x) {
   stackNode element = new stackNode(x);
   element.next = top;
   top = element;
   System.out.println("Element pushed");
   size++;
 }
```

```
public int stackPop() {
  if (top == null) return -1;
  int topData = top.data;
  size - - ;
  top = top.next;
  return topData;
 }
```

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

```
 public boolean stackIsEmpty() {
  return top == null;
 }
```

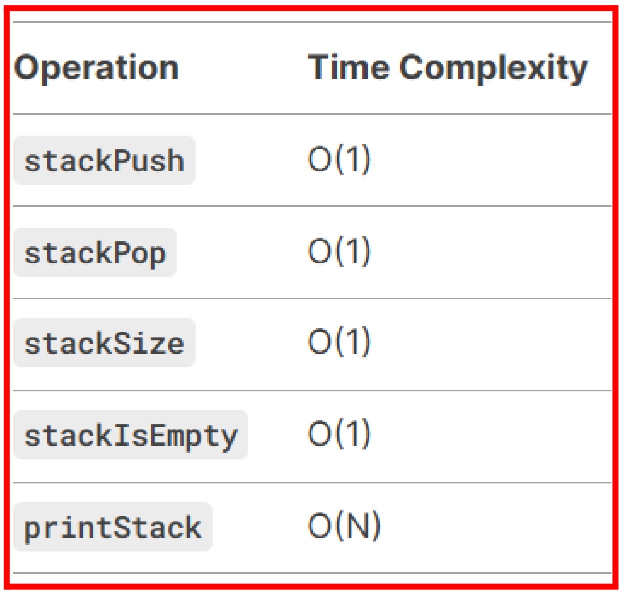
```
 public void printStack() {
  stackNode current = top;
  while (current != null) {
    System.out.print(current.data + " ");
    current = current.next;
  }
  System.out.println();
 }
}
```

```
class Main {
  public static void main(String args[]) {
    stack s = new stack();
    s.stackPush(10);
    s.stackPush(20);
    s.printStack();
    System.out.println("Element popped " + s.stackPop());
    System.out.println("Stack size: " + s.stackSize());
    System.out.println("Stack is empty or not: " + s.stackIsEmpty());

  }
}
```

Output:

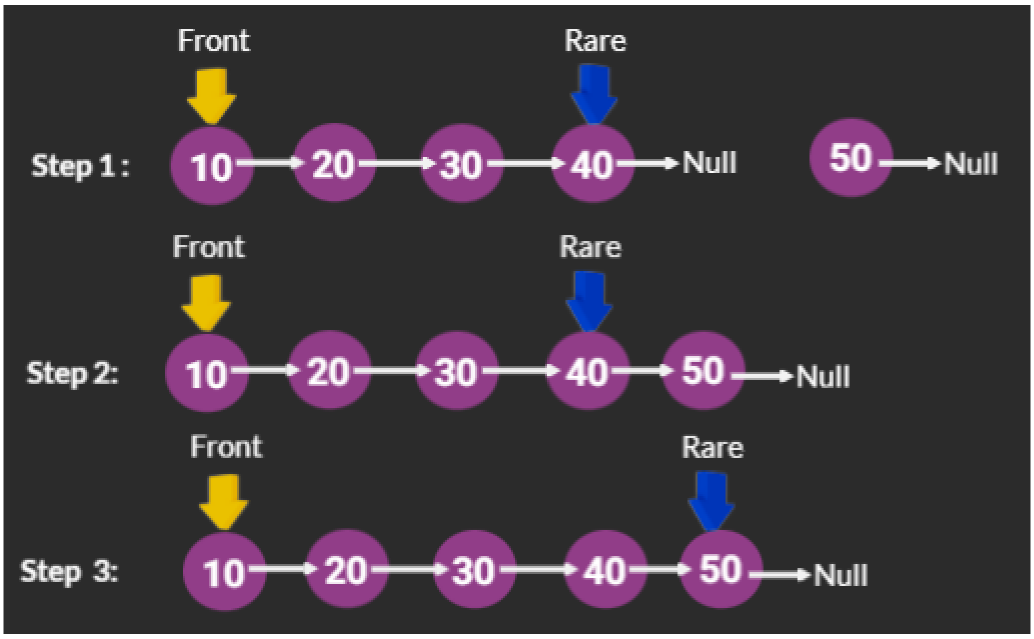
```
Element pushed
Element pushed
20 10
Element popped 20
Stack size: 2
Stack is empty or not: false
```

Time complexity :



Operation	Time Complexity
`stackPush`	O(1)
`stackPop`	O(1)
`stackSize`	O(1)
`stackIsEmpty`	O(1)
`printStack`	O(N)

Queue Using LL



```
import java.util.*;
```

```
class QueueNode {
   int val;
   QueueNode next;

   // Constructor to initialize a node
   QueueNode(int data) {
      this.val = data;
      this.next = null;
   }
}
```

```
class Queue {
   private QueueNode front; // Front of the queue
   private QueueNode rear;  // Rear of the queue
   private int size;        // Current size of the queue
```

```
   // Constructor to initialize an empty queue
   public Queue() {
      this.front = null;
      this.rear = null;
      this.size = 0;
   }
```

```
   // Check if the queue is empty
   public boolean isEmpty() {
      return front == null;
   }
```

```
   // Get the front element of the queue (peek)
   public int peek() {
      if (isEmpty()) {

         return -1; // Return a default value indicating underflow
      }
      return front.val;
   }
```

```
   // Add an element to the rear of the queue (offer)
   public boolean offer(int value) {
    QueueNode newNode =new QueueNode(value);

    if (isEmpty()) { // If the queue is empty, set front and rear to the new node
        front = newNode;
        rear = newNode;
    } else { // Otherwise, add the new node to the rear
        rear.next = newNode;
        rear = newNode;
    }
    size++;

    return true;
}
```

```
// Remove an element from the front of the queue (poll)
public int poll() {
    if (isEmpty()) {
    return -1; // Return a default value indicating underflow
    }
    int polledValue = front.val;
    front = front.next; // Move front to the next node
    size--;
    return polledValue;
}
```

```
// Get the current size of the queue
public int size() {
    return size;
}
```

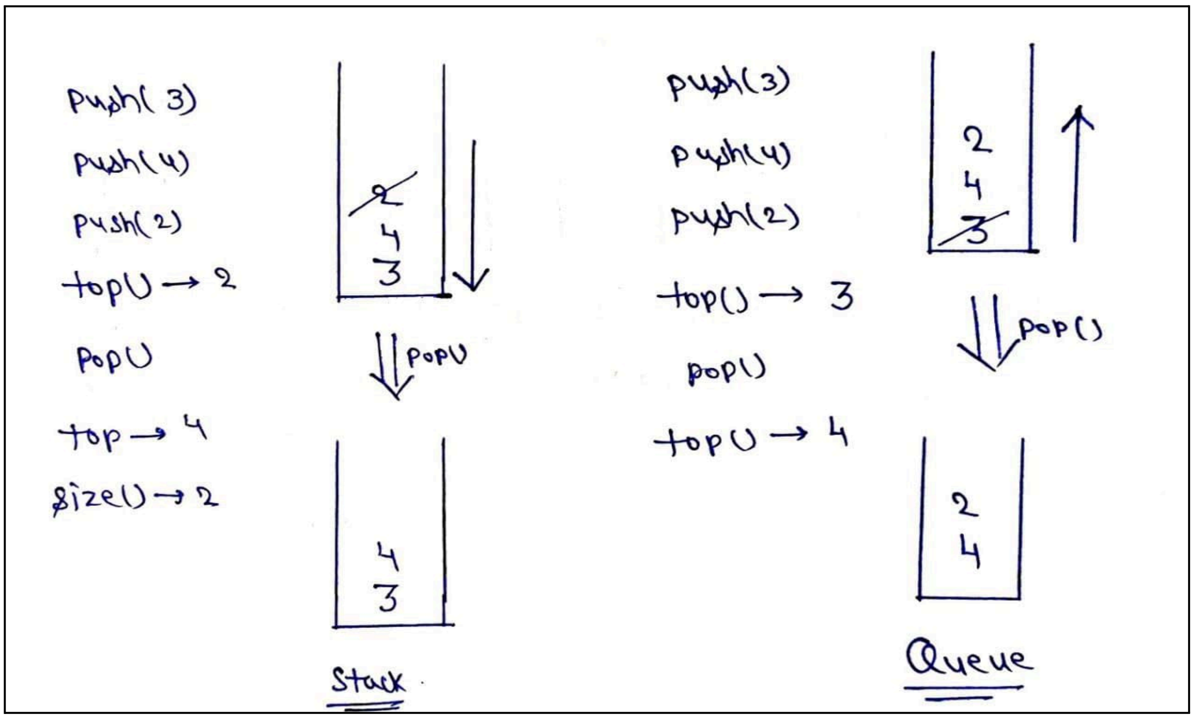
```
// Print the queue (for debugging purposes)
public void printQueue() {
    if (isEmpty()) {
        System.out.println("Queue is empty.");
        return;
    }
    QueueNode current = front;

    while (current != null) {
        System.out.print(current.val + " ");
        current = current.next;
    }
    System.out.println();
}
```

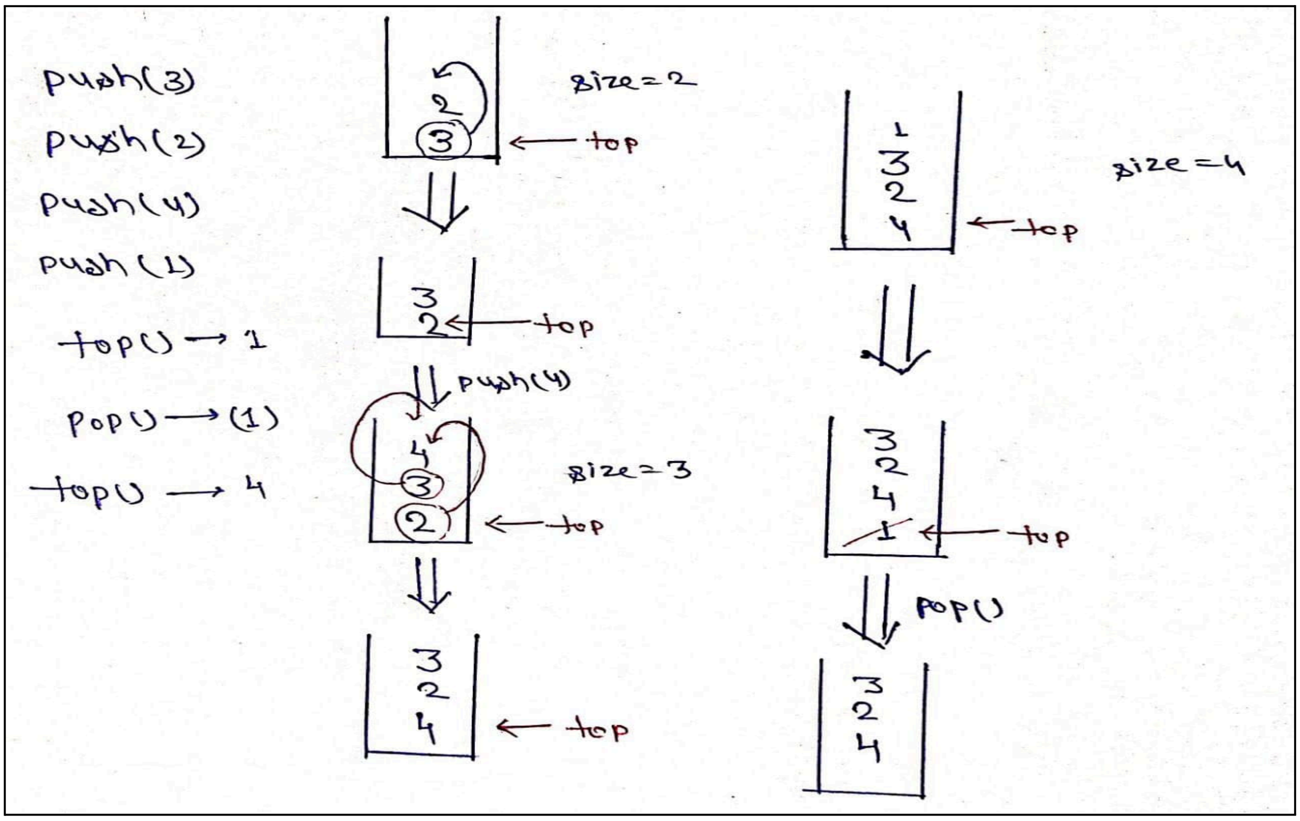
```
    public static void main(String[] args) {
        Queue queue = new Queue();

        // Offer elements
        queue.offer(10);
        queue.offer(20);
        queue.offer(30);
        queue.offer(40);
        queue.offer(50);

        // Print the queue
        queue.printQueue();

        // Poll elements
        queue.poll();
        queue.poll();

        // Print the queue after polling
        queue.printQueue();

        // Peek at the front element
        System.out.println("Peek element: " + queue.peek());

        // Get the size of the queue
        System.out.println("Queue size: " + queue.size());

        // Check if the queue is empty
        System.out.println("Is queue empty? " + queue.isEmpty());
    }
}
```

Output : :

```
10 20 30 40 50
30 40 50
Peek element: 30
Queue size: 3
Is queue empty? false
```

Implement Stack Using Queue





```
import java.util.*;


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

```
public class tuf {

    public static void main(String[] args) {
        stack s = new stack();
        s.push(3);
        s.push(2);
        s.push(4);
        s.push(1);
        System.out.println("Top of the stack: " + s.top());
        System.out.println("Size of the stack before removing element: " + s.size());
        System.out.println("The deleted element is: " + s.pop());
        System.out.println("Top of the stack after removing element: " + s.top());
        System.out.println("Size of the stack after removing element: " + s.size());
    }

}
```

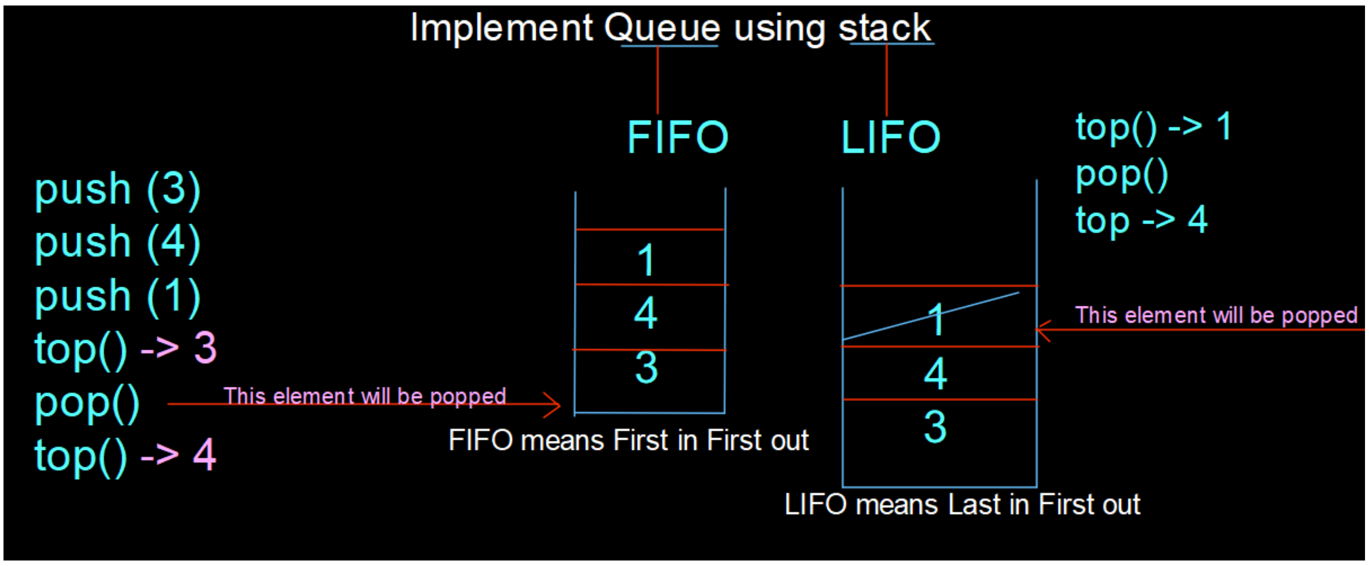
Output:

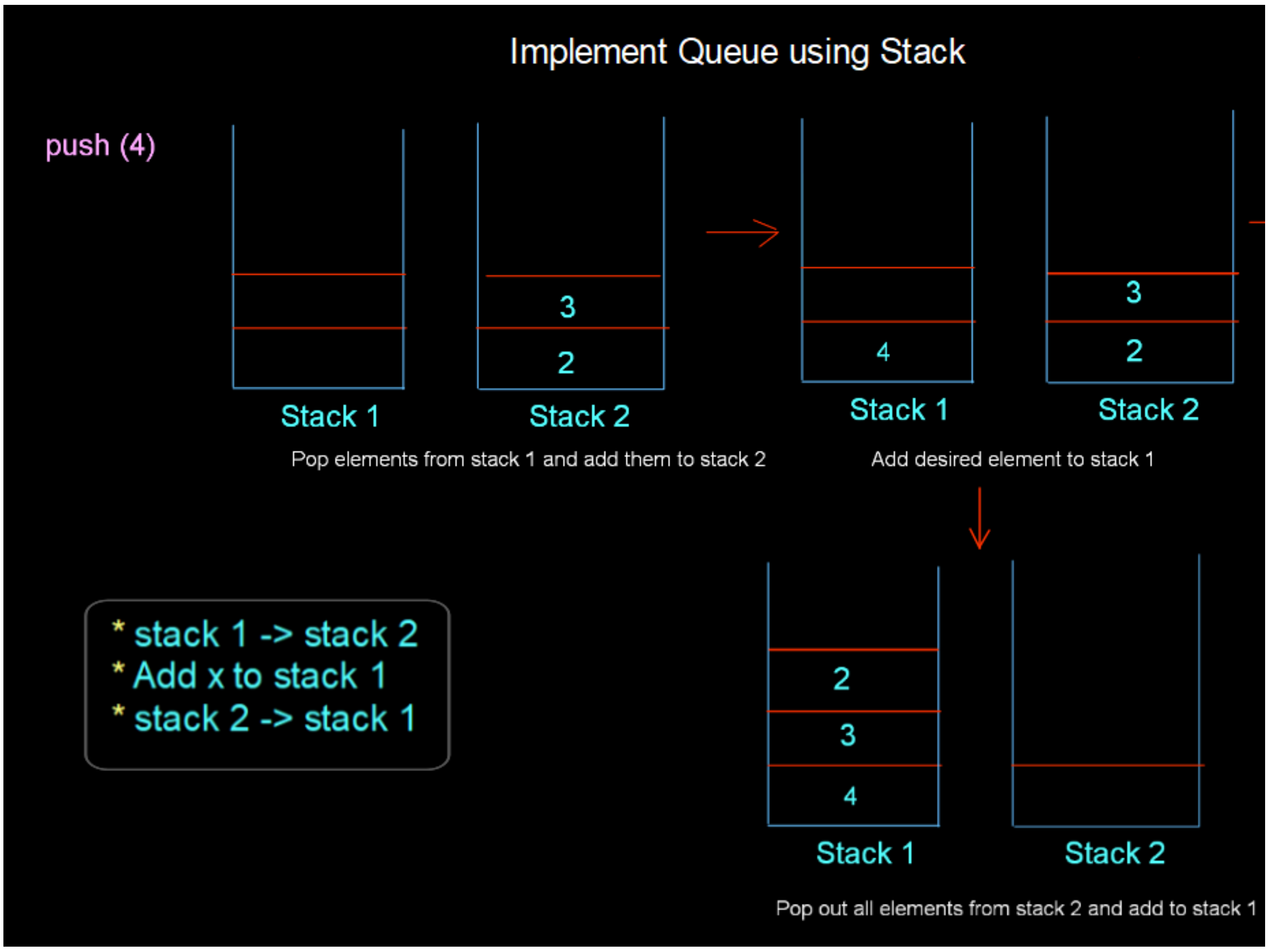
```
Top of the stack: 1
Size of the stack before removing element: 4
The deleted element is: 1
Top of the stack after removing element: 4
Size of the stack after removing element: 3
```

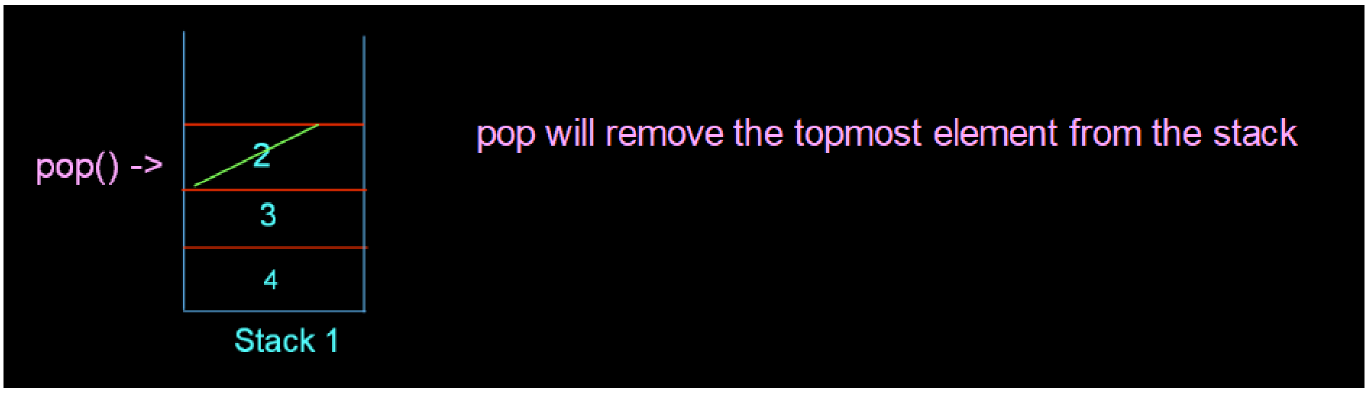
Time Complexity: O(N)

Space Complexity: O(N)

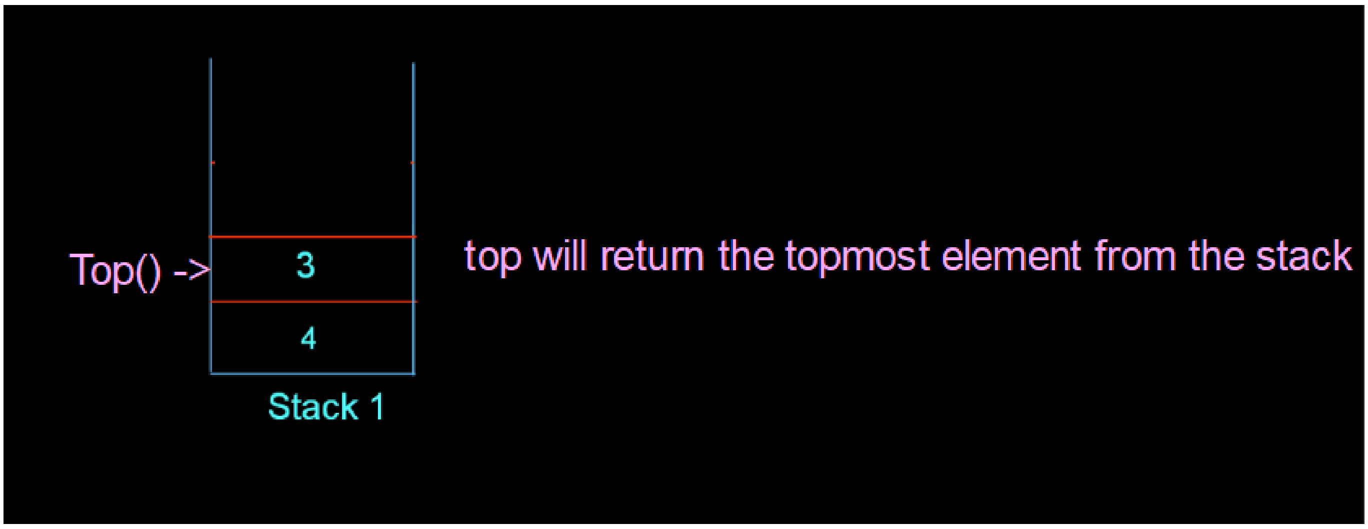
Implement of queue using stack


Implement Queue using stack
FIFO
LIFO
push (3)
push (4)
push (1)
top() -> 3
pop()
top() -> 4
This element will be popped
1
4
3
FIFO means First in First out
top() -> 1
pop()
top -> 4
This element will be popped
1
4
3
LIFO means Last in First out


Implement Queue using Stack
push (4)
3
2
Stack 1
Stack 2
Pop elements from stack 1 and add them to stack 2
4
3
2
Stack 1
Stack 2
Add desired element to stack 1
* stack 1 -> stack 2
* Add x to stack 1
* stack 2 -> stack 1
2
3
4
Stack 1
Stack 2
Pop out all elements from stack 2 and add to stack 1



peek()



```
import java.util.*;

class MyQueue {

    Stack < Integer > input = new Stack < > ();
    Stack < Integer > output = new Stack < > ();
    /** Initialize your data structure here. */
    public MyQueue() {

    }


 /** Push element x to the back of queue. */

 public void push(int x) {
        while (input.empty() == false) {
           output.push(input.peek());
           input.pop();
        }
        // Insert the desired element in the stack input
        System.out.println("The element pushed is " + x);
        input.push(x);
        // Pop out elements from the stack output and push them into the stack input
        while (output.empty() == false) {
            input.push(output.peek());
            output.pop();
        }

    }
```

```
/** Removes the element from in front of queue and returns that element. */
```

```
    public int pop() {
        // shift input to output
        if (input.empty()) {
            System.out.println("Stack is empty");

        }
        int val = input.peek();
        input.pop();
        return val;

    }
```

```
  /** Get the front element. */
```

```
   public int peek() {
        // shift input to output
        if (input.empty()) {
            System.out.println("Stack is empty");

        }
        return input.peek();
    }
```

```
 int size() {
        return input.size();
    }
}
```

```
class TUF {
   public static void main(String args[]) {
      MyQueue q = new MyQueue();
      q.push(3);
      q.push(4);
      System.out.println("The element poped is " + q.pop());
      q.push(5);
      System.out.println("The top element is " + q.peek());
      System.out.println("The size of the queue is " + q.size());

   }
}
```

Output:

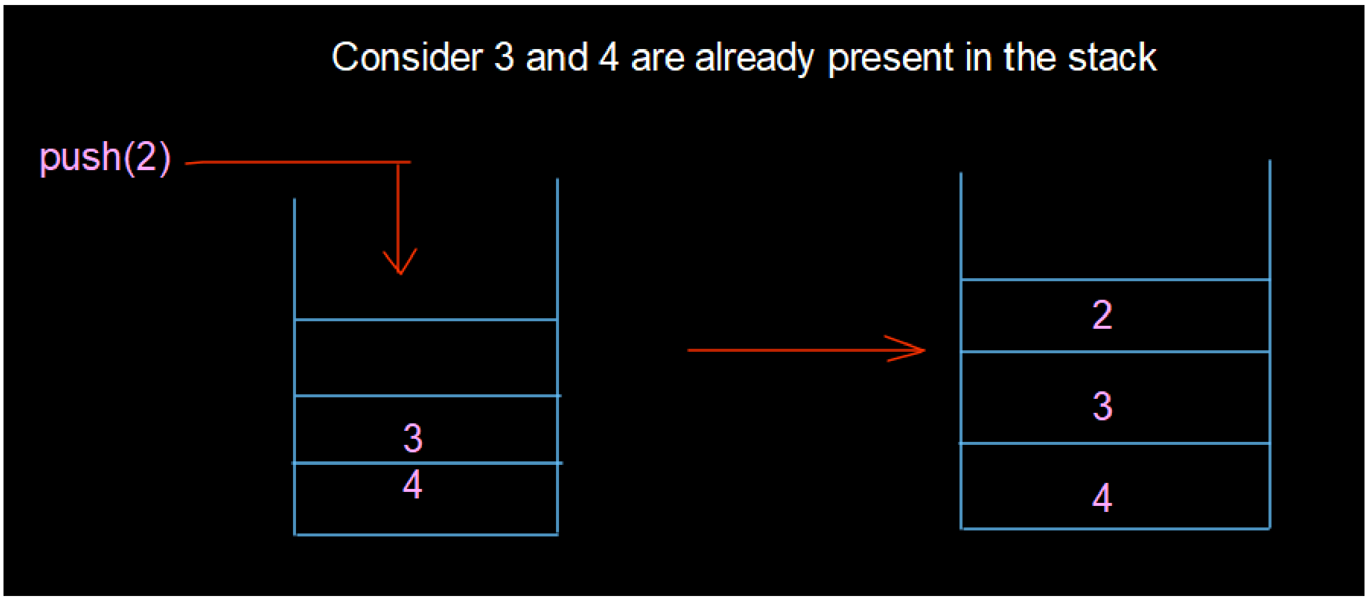
```
The element pushed is 3
The element pushed is 4
The element poped is 3
The element pushed is 5
The top element is 4
The size of the queue is 2
```

Time Complexity: O(N)

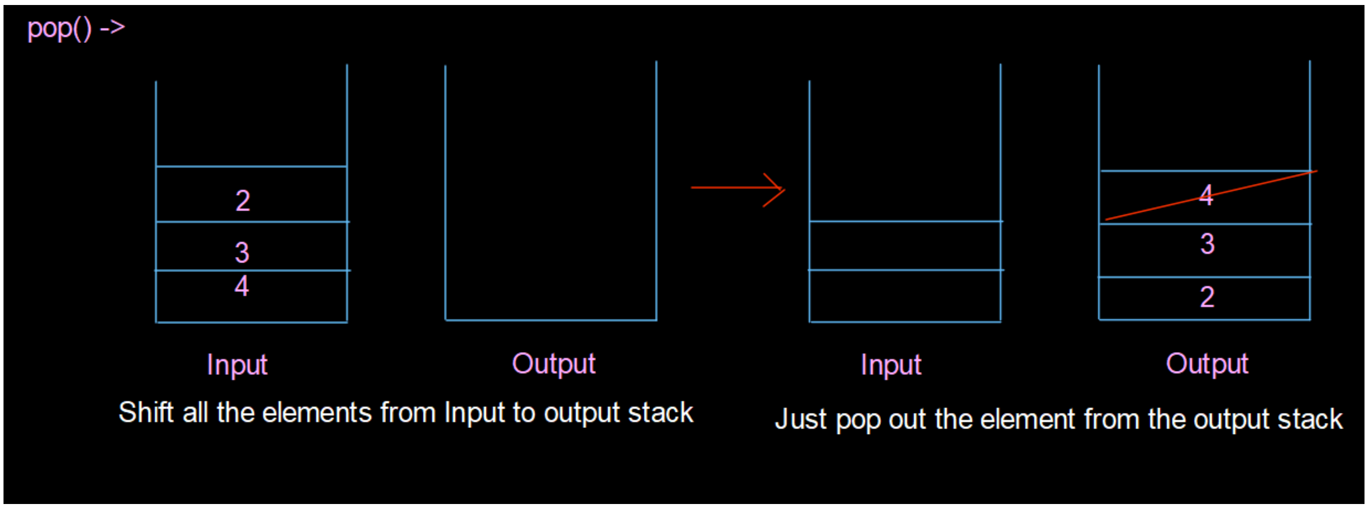
Space Complexity: O(2N)

Solution 2: Using two Stacks where push operation is O(1)

Push()->



Pop()->



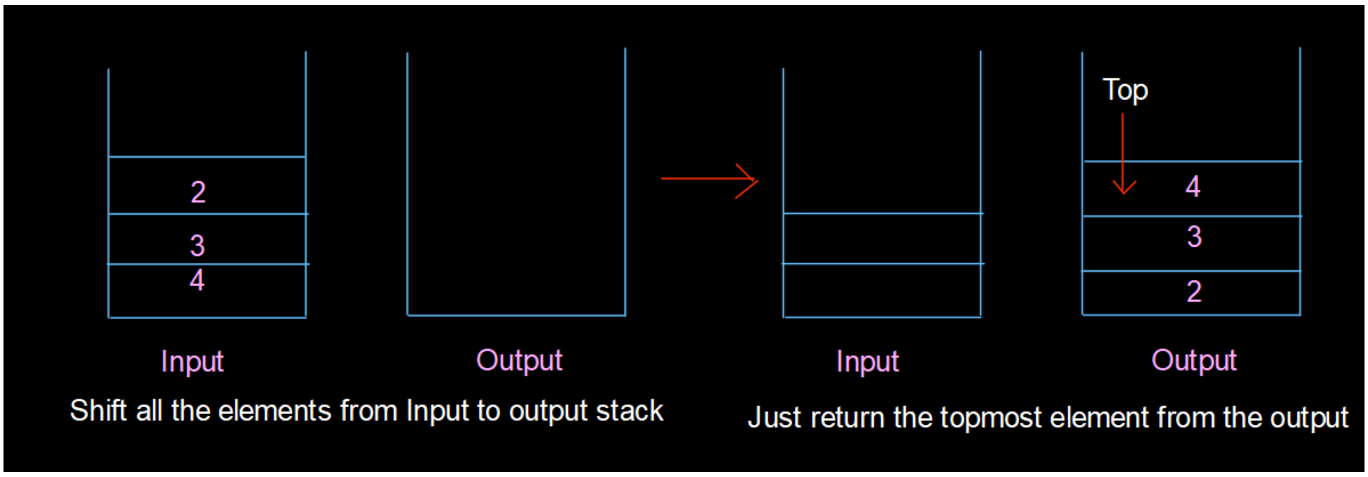
Shift all the elements from Input to output stack

Just pop out the element from the output stack

pop() -> pop can be O(1) or O(n) .

O(1) when all elements are already shifted . So we just need to delete the element . So here we can say that time complexity can be O(1) or amortised O(1)

top()->



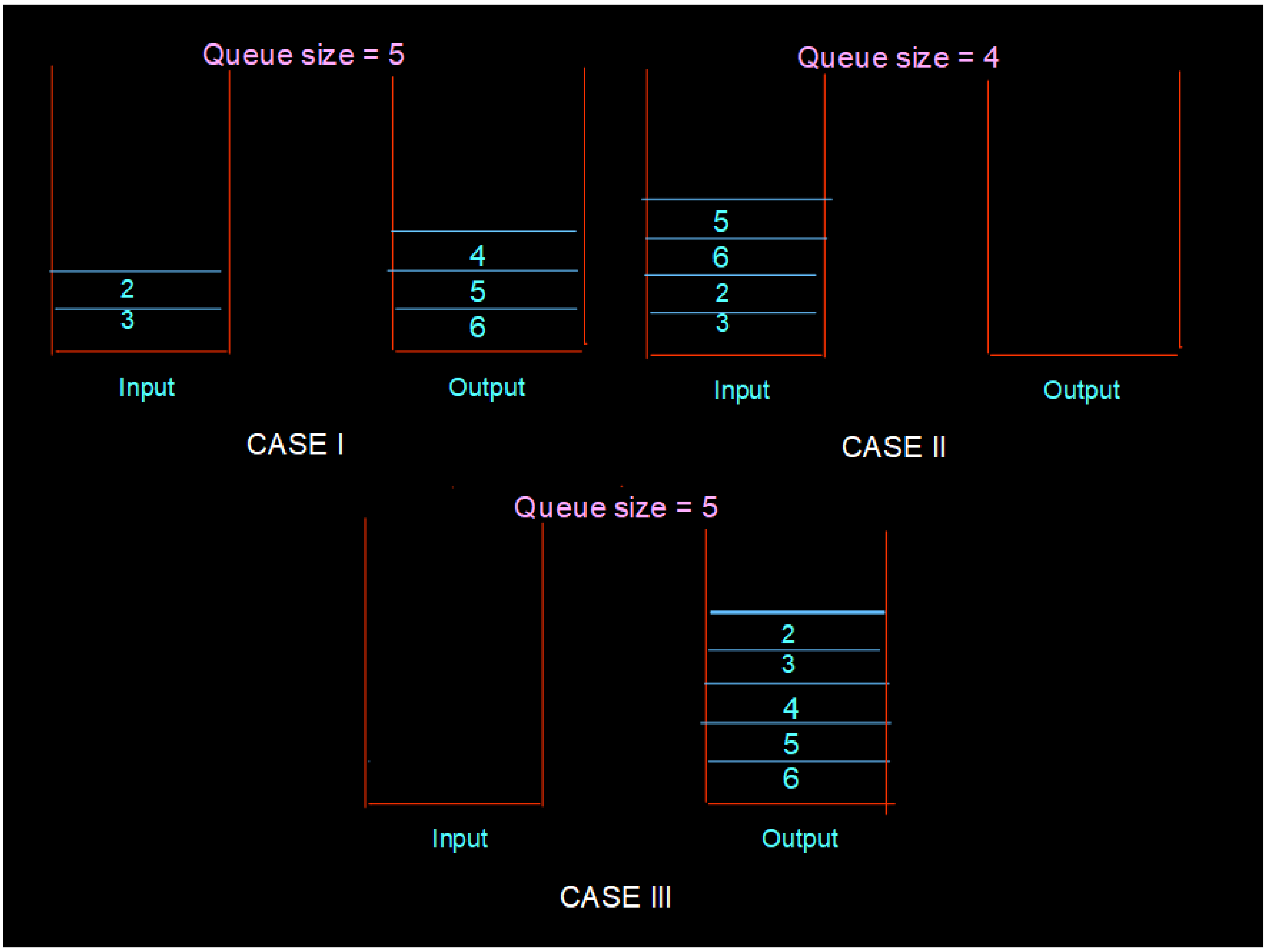
Shift all the elements from Input to output stack

Just return the topmost element from the output

Top() -> Top can be O(1) or O(n) .

O(1) when all elements are already shifted . So we just need to return the element . So here we can say that time complexity can be O(1) or amortised O(1)

Size()



```
import java.util.*;

class MyQueue {

Stack < Integer > input = new Stack < > ();
  Stack < Integer > output = new Stack < > ();


 /** Initialize your data structure here. */

public MyQueue() {

  }

 /** Push element x to the back of queue. */

public void push(int x) {
      System.out.println("The element pushed is " + x);
      input.push(x);
  }
  /** Removes the element from in front of queue and returns that element. */

 public int pop() {
      // shift input to output
      if (output.empty())
         while (input.empty() == false) {
            output.push(input.peek());
            input.pop();
         }


      int x = output.peek();
      output.pop();
      return x;
  }
```

```
 /** Get the front element. */

 public int peek() {
     // shift input to output
     if (output.empty())
        while (input.empty() == false) {
           output.push(input.peek());
           input.pop();
        }
     return output.peek();
  }
  int size() {
     return (output.size() + input.size());
  }


}
```

```
class TUF {

public static void main(String args[]) {
     MyQueue q = new MyQueue();
     q.push(3);
     q.push(4);
     System.out.println("The element poped is " + q.pop());
     q.push(5);
```

```
        System.out.println("The top element is " + q.peek());
        System.out.println("The size of the queue is " + q.size());

    }

}
```

Output:

```
Time Complexity: O(1 )

Space Complexity: O(2N)
```