STACK QUESTION

Question-> Queue using 2 stack

Code->

```
package Day53Queue;
import java.util.*;
public class Ques1QueueUsing2Stack {
    static class QueueUsing2Stack {
       static Stack<Integer> s1 = new Stack<>();
       static Stack<Integer> s2 = new Stack<>();

       public static boolean isEmpty(){
        return s1.isEmpty();
       }
       //add - O(n)
       public static void add(int data){
         // Move all elements from s1 to s2
        while(!s1.isEmpty()){
            s2.push(s1.pop());
        }
        // Push new element into s1
        s1.push(data);

        // Move everything back to s1 from s2
        while(!s2.isEmpty()){
            s1.push(s2.pop());
        }
       }
       //remove - O(1)
       public static int remove(){
        if(isEmpty()){
            System.out.println("Queue is empty");
            return -1;
        }
        return s1.pop();

       }
       //peek
       public static int peek(){
        if(isEmpty()){
            System.out.println("Queue is empty");
            return -1;
        }
        return s1.peek();
       }

    }
    public static void main(String[] args) {
        QueueUsing2Stack q = new QueueUsing2Stack();
```

STACK QUESTION

```
        q.add(1);
        q.add(2);
        q.add(3);
        while(!q.isEmpty()){
            System.out.println(q.peek());
            q.remove();
        }

    }


}
```

Pseudocode→

Class QueueUsing2Stack

Declare two stacks 's1' and 's2'

Method isEmpty()

Return true if 's1' is empty

Method add(data)

While 's1' is not empty

Move elements from 's1' to 's2' (s1.pop() → s2.push())

Push 'data' into 's1'

While 's2' is not empty

Move elements from 's2' back to 's1' (s2.pop() → s1.push())

Method remove()

If queue is empty (isEmpty() == true)

Print "Queue is empty"

Return -1

Else

Pop and return the top element from 's1'

```
Method peek()
  If queue is empty (isEmpty() == true)
    Print "Queue is empty"
    Return -1
  Else
    Return the top element from 's1'

Main method
  Create an instance of QueueUsing2Stack, 'q'
  Call q.add(1)
  Call q.add(2)
  Call q.add(3)

  While q is not empty (q.isEmpty() == false)
    Print q.peek()
    Call q.remove()
```

Explanation:

1. **Class Structure**: The `QueueUsing2Stack` class implements a queue using two stacks (`s1` and `s2`).
   - **s1**: Holds the elements in queue order.
   - **s2**: A temporary stack used to reverse the order of elements when adding a new element.
2. **isEmpty()**: This method checks if the queue is empty by checking if `s1` is empty. If `s1` is empty, the queue is empty.
3. **add(data)**:
   - This method adds a new element (`data`) to the queue.
   - It first moves all elements from `s1` to `s2` (to reverse the order).
   - Then, the new element is pushed onto `s1`.
   - Finally, all elements are moved back from `s2` to `s1`, restoring the queue order.
4. **remove()**:
   - This method removes and returns the front element of the queue by simply popping from `s1` (since the front element is at the top of `s1` after the add process).
   - If the queue is empty, it returns `-1`.
5. **peek()**:
   - This method returns the front element of the queue without removing it by checking the top of `s1`. If `s1` is empty, it returns `-1`.
6. **Main Method**:

- A `QueueUsing2Stack` object (`q`) is created.
- The elements `1`, `2`, and `3` are added to the queue.
- Then, a loop runs while the queue is not empty, printing the front element (`q.peek()`) and removing it (`q.remove()`).

This code demonstrates how to implement a queue with two stacks, where the enqueue operation (`add`) has a time complexity of O(n)O(n)O(n) and the dequeue operation (`remove`) has a time complexity of O(1)O(1)O(1).

4o

Algorithm for Queue Using Two Stacks:

1. isEmpty()

- **Input**: None
- **Output**: Returns `true` if the queue is empty, `false` otherwise
- **Steps**:
  1. Check if `s1` is empty.
  2. If `s1` is empty, return `true`.
  3. Else, return `false`.

2. add(data)

- **Input**: `data` (the element to be added to the queue)
- **Output**: None
- **Steps**:
  1. While `s1` is not empty:
     - Pop the top element from `s1` and push it onto `s2`.
  2. Push the new element (`data`) onto `s1`.
  3. While `s2` is not empty:
     - Pop the top element from `s2` and push it back onto `s1`.

3. remove()

- **Input**: None
- **Output**: Returns the front element of the queue or `-1` if the queue is empty
- **Steps**:
  1. If the queue is empty (call `isEmpty()`):
     - Print "Queue is empty" and return `-1`.
  2. Otherwise, pop the top element from `s1` and return it.

4. peek()

- **Input**: None
- **Output**: Returns the front element of the queue or `-1` if the queue is empty
- **Steps**:
  1. If the queue is empty (call `isEmpty()`):
     - Print "Queue is empty" and return `-1`.
  2. Otherwise, return the top element from `s1`.

5. Main Method

- **Input**: None
- **Output**: Prints the elements at the front of the queue while removing them
- **Steps**:
    1. Create a `QueueUsing2Stack` object `q`.
    2. Call `q.add(1)`, `q.add(2)`, and `q.add(3)` to add elements to the queue.
    3. While `q.isEmpty()` returns `false`:
        - Call `q.peek()` to print the front element.
        - Call `q.remove()` to remove the front element.

This algorithm follows a **two-stack approach** for simulating the queue where the `add` operation involves shifting elements between the two stacks to maintain order, and `remove` is efficient with a direct pop from the stack.

Question-> Stack Using 2 Queue?

Code-

```
package Day53Queue;

import java.util.*;

public class Ques3StackUsing2Queue {
    static class Stack {
        Queue<Integer> q1 = new LinkedList<>();
        Queue<Integer> q2 = new LinkedList<>();

        // Check if the stack is empty
        public boolean isEmpty() {
            return q1.isEmpty() && q2.isEmpty();
        }

        // Push an element onto the stack
        public void push(int data) {
            // Always add to the non-empty queue
            if (!q1.isEmpty()) {
                q1.add(data);
            } else {
                q2.add(data);
            }
        }

        // Pop an element from the stack
        public int pop() {
            if (isEmpty()) {
```

```
                System.out.println("Empty Stack");
                return -1;
            }
            int top = -1;

            // Case 1: If q1 is not empty, shift elements from q1 to q2
            if (!q1.isEmpty()) {
                while (!q1.isEmpty()) {
                    top = q1.remove();
                    if (q1.isEmpty()) {
                        break;  // We've reached the last element, which is
the top of the stack
                    }
                    q2.add(top);
                }
            }
            // Case 2: If q2 is not empty, shift elements from q2 to q1
            else {
                while (!q2.isEmpty()) {
                    top = q2.remove();
                    if (q2.isEmpty()) {
                        break;  // We've reached the last element, which is
the top of the stack
                    }
                    q1.add(top);
                }
            }
            return top;
        }

        // Peek the top element of the stack
        public int peek() {
            if (isEmpty()) {
                System.out.println("Empty Stack");
                return -1;
            }
            int top = -1;

            // Case 1: If q1 is not empty, shift elements and get the top
            if (!q1.isEmpty()) {
                while (!q1.isEmpty()) {
                    top = q1.remove();
                    q2.add(top);  // Move elements back to q2
                }
            }
            // Case 2: If q2 is not empty, shift elements and get the top
            else {
                while (!q2.isEmpty()) {
```

```
                    top = q2.remove();
                    q1.add(top);  // Move elements back to q1
                }
            }
            return top;
        }
    }

    public static void main(String[] args) {
        Stack s = new Stack();
        s.push(1);
        s.push(2);
        s.push(3);

        // Pop and print elements until the stack is empty
        while (!s.isEmpty()) {
            System.out.println(s.pop());
        }
    }
}
```

Pseudocode for Stack Using Two Queues→

Class Stack:

Initialize two empty queues: q1, q2

Function isEmpty():

Return True if both q1 and q2 are empty, otherwise return False

Function push(data):

If q1 is not empty:

Add data to q1

Else:

Add data to q2

Function pop():

If isEmpty():

Print "Empty Stack"

Return -1

Else:

If q1 is not empty:

While q1 has more than one element:

Move element from q1 to q2

Remove the last element from q1 (this is the top of the stack)

Return the removed element

Else:

While q2 has more than one element:

Move element from q2 to q1

Remove the last element from q2 (this is the top of the stack)

Return the removed element

Function peek():

If isEmpty():

Print "Empty Stack"

Return -1

Else:

If q1 is not empty:

While q1 is not empty:

Remove element from q1

Add it to q2

Track the last removed element (this is the top of the stack)

Return the last removed element

Else:

While q2 is not empty:

Remove element from q2

Add it to q1

Track the last removed element (this is the top of the stack)

Return the last removed element

Main:

STACK QUESTION

Initialize stack s

Call s.push(1)

Call s.push(2)

Call s.push(3)

While s is not empty:

Print s.pop()

Algorithm for Stack Using Two Queues→

1. isEmpty():

- **Input**: None
- **Output**: Returns `True` if both queues `q1` and `q2` are empty, otherwise `False`.
- **Steps**:
    1. Check if `q1` and `q2` are both empty.
    2. Return the result.

2. push(data):

- **Input**: `data` (the element to be pushed onto the stack).
- **Output**: None.
- **Steps**:
    1. Check if `q1` is not empty:
        - If so, add `data` to `q1`.
    2. Else, add `data` to `q2`.

3. pop():

- **Input**: None.
- **Output**: Returns the element at the top of the stack or `-1` if the stack is empty.
- **Steps**:
    1. If the stack is empty, print "Empty Stack" and return `-1`.
    2. If `q1` is not empty:
        - Move all elements except the last one from `q1` to `q2`.
        - Remove the last element from `q1` and return it.
    3. Else:
        - Move all elements except the last one from `q2` to `q1`.
        - Remove the last element from `q2` and return it.

4. peek():

- **Input**: None.
- **Output**: Returns the top element of the stack without removing it, or `-1` if the stack is empty.
- **Steps**:
    1. If the stack is empty, print "Empty Stack" and return `-1`.
    2. If `q1` is not empty:

- Move all elements from `q1` to `q2` while tracking the last removed element (this is the top of the stack).
- Return the tracked element.

3. Else:
   - Move all elements from `q2` to `q1` while tracking the last removed element (this is the top of the stack).

Return the trackeDetailed Explanation:

The goal is to implement a **stack** (LIFO - Last In, First Out) using two **queues** (FIFO - First In, First Out). Queues and stacks work with opposite principles, so to simulate a stack, we need to use some queue operations cleverly.

1. push(data):

When we push an element to the stack, we need to place it in one of the queues. The idea is that the stack's last added element should always be at the front of the queue so that it can be popped or peeked immediately. We achieve this by:

- If `q1` has elements, we add the new element to `q1`.
- If `q1` is empty and `q2` has elements, we add the new element to `q2`. This way, the newly pushed element will always be the first element to be accessed.

2. pop():

To simulate a stack's pop operation (removing the last added element), we need to:

- Move all elements from the non-empty queue (either `q1` or `q2`) to the other queue, except the last element.
- The last remaining element is the top of the stack, which we remove and return.

3. peek():

To peek the top element of the stack without removing it:

- We follow the same process as `pop()` but instead of removing the last element, we just return its value and move it to the other queue.

4. isEmpty():

This is a simple check to see if both queues are empty. If both are empty, the stack is empty.

Time Complexity Analysis:

- **Push Operation**: O(n) where `n` is the number of elements in the queue. This is because every time we push an element, we may have to move all existing elements between the two queues.

- **Pop Operation**: O(n), similar to push, since all elements except the last one have to be moved between queues.
- **Peek Operation**: O(n) for the same reason as pop.
- **isEmpty Operation**: O(1), as it only checks if both queues are empty.

Thus, while this implementation is functional, it's not optimal. However, it serves as a useful exercise to understand how stacks and queues differ and how one can be used to simulate the other.