Implement Queue Using Array
Problem Statement: Implement Queue Data Structure using Array with all functions like pop, push, top, size, etc.
Example:

Input: push(4)
push(14)
push(24)
push(34)
top()
size()
pop()

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

Solution

***Disclaimer**: Don't jump directly to the solution, try it out yourself first.*

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.
Code:

- C++ Code
- Java Code
- Python Code

```cpp
#include<bits/stdc++.h>

using namespace std;
class Queue {
  int * arr;
  int start, end, currSize, maxSize;
  public:
    Queue() {
      arr = new int[16];
      start = -1;
      end = -1;
      currSize = 0;
    }

  Queue(int maxSize) {
    ( * this).maxSize = maxSize;
    arr = new int[maxSize];
    start = -1;
    end = -1;
```

```
        currSize = 0;
    }
    void push(int newElement) {
        if (currSize == maxSize) {
            cout << "Queue is full\nExiting..." << endl;
            exit(1);
        }
        if (end == -1) {
            start = 0;
            end = 0;
        } else
            end = (end + 1) % maxSize;
        arr[end] = newElement;
        cout << "The element pushed is " << newElement << endl;
        currSize++;
    }
    int pop() {
        if (start == -1) {
            cout << "Queue Empty\nExiting..." << endl;
        }
        int popped = arr[start];
        if (currSize == 1) {
            start = -1;
            end = -1;
        } else
            start = (start + 1) % maxSize;
        currSize--;
        return popped;
    }
    int top() {
        if (start == -1) {
            cout << "Queue is Empty" << endl;
            exit(1);
        }
        return arr[start];
    }
    int size() {
        return currSize;
    }

};

int main() {
    Queue q(6);
    q.push(4);
    q.push(14);
    q.push(24);
    q.push(34);
    cout << "The peek of the queue before deleting any element " << q.top() << endl;
    cout << "The size of the queue before deletion " << q.size() << endl;
    cout << "The first element to be deleted " << q.pop() << endl;
    cout << "The peek of the queue after deleting an element " << q.top() << endl;
    cout << "The size of the queue after deleting an element " << q.size() << endl;

    return 0;
}
```

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

Time Complexity:

pop function: O(1)

push function: O(1)

top function: O(1)

size function: O(1)

Space Complexity:

Whole Queue: O(n)