Module on Queue Data Structure

Introduction to Queues:

A queue is a linear data structure that follows the FIFO (First In First Out) principle. This means that the element which is inserted first will be the one to be removed first. Think of it like a queue of people waiting for a bus to go to a particular destination, where the person who arrived first gets on the bus first and the person who enters at last will exit first.

Properties of Queue:
->FIFO (First In First Out): The element that is added first is the one that gets removed first.
->Linear Data Structure: Queue elements are arranged in a linear sequence.
->Two Ends: A queue has two ends, namely the front and the rear. Elements are added to the rear end and removed from the front end.
->Dynamic Size: Queues can be of fixed size or dynamic size, depending on the implementation.

Types of Queues:
1. Simple Queue: A basic queue where elements are inserted at the rear end and removed from the front end.
Circular Queue: A queue where the rear end is connected to the front end, forming a circle. This allows for efficient space utilization in a fixed-size queue.

2. Priority Queue: A queue where elements have associated priorities, and the element with the highest priority is dequeued first.
3. Double Ended Queue (Deque): A queue that supports insertion and deletion at both ends.

Applications of Queue:
1. Job Scheduling: Queues are used in job scheduling algorithms like First-Come-First-Serve (FCFS) scheduling.
2. Breadth-First Search (BFS): Queues are utilized in graph traversal algorithms like BFS to explore nodes level by level.
Buffer Management: Queues are used in managing buffers in operating systems and networking.
3. Printing Queues: Queues are used to manage print jobs in printers.
4. CPU Scheduling: Queues are used in CPU scheduling algorithms to manage processes waiting to be executed.
Implementation:
5. Queues can be implemented using arrays, linked lists, or other data structures. The choice of implementation depends on the requirements such as dynamic size, efficiency, etc.
Array Implementation:
->Uses an array to store queue elements.
->Requires additional logic to handle wrap-around in circular queues.
->Can have fixed or dynamic size.
Linked List Implementation:
->Uses a linked list to store queue elements.

Allows for dynamic size without worrying about resizing.
Efficient for frequent insertions and deletions.
Complexity Analysis:

Time Complexity:
->Enqueue: O(1)
->Dequeue: O(1)
->Front: O(1)
->Rear: O(1)

Space Complexity: O(n)

Operations on Queue:

1. Enqueue (Insertion): Adds an element to the rear of the queue.
2. Dequeue (Deletion): Removes an element from the front of the queue.
3. Front: Returns the element at the front of the queue without removing it.
4. Rear: Returns the element at the rear of the queue without removing it.
5. isEmpty: Checks if the queue is empty.
6. isFull: Checks if the queue is full (applicable in a fixed-size array implementation).

Queue Implementation using STL:

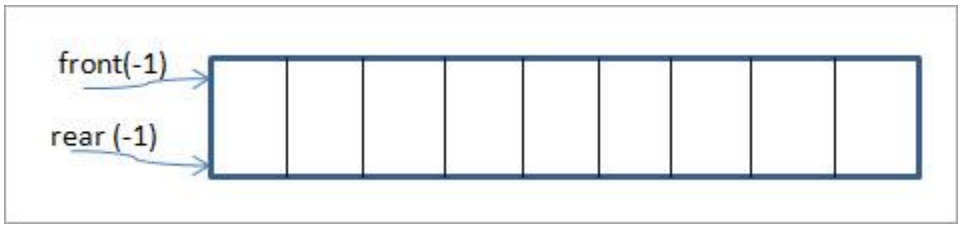
01_queue_stl.cpp | 02_queueImpliUsingArray.cpp

Queue > lec60 > 01_queue_stl.cpp > main()

```cpp
#include<bits/stdc++.h>
using namespace std;

int main(){
    queue<int> q;
    //Enqueue
    q.push(11);
    q.push(45);

    cout << "Front is: " << q.front() << endl;

    cout << "Size is: " << q.size() << endl;

    //Dequeue
    q.pop();

    cout << "Front is: " << q.front() << endl;

    cout << "Size is: " << q.size() << endl;

    //check whether queue is empty
    if(q.empty()){
        cout << "Q is empty" << endl;
    }
    else{
        cout << "Q is not empty" << endl;
    }
    return 0;
}
```

Queue Implementation:

In software terms, the queue can be viewed as a set or collection of elements as shown below. The elements are arranged linearly.



We have two ends i.e. “front” and “rear” of the queue. When the queue is empty, then both the pointers are set to -1.

The “rear” end pointer is the place from where the elements are inserted in the queue. The operation of adding /inserting elements in the queue is called “enqueue”.

The “front” end pointer is the place from where the elements are removed from the queue. The operation to remove/delete elements from the queue is called “dequeue”.

When the rear pointer value is size-1, then we say that the queue is full. When the front is null, then the queue is empty.

Enqueue

In this process, the following steps are performed:

- **Check if the queue is full.**
- **If full, produce overflow error and exit.**
- **Else, increment 'rear'.**
- **Add an element to the location pointed by 'rear'.**
- **Return success.**

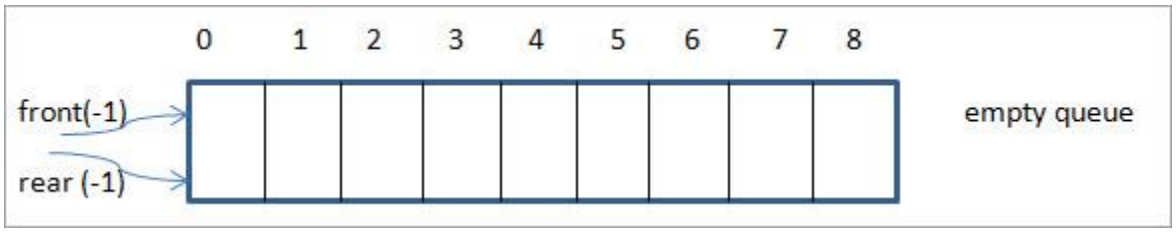
Dequeue

Dequeue operation consists of the following steps:

- **Check if the queue is empty.**
- **If empty, display an underflow error and exit.**
- **Else, the access element is pointed out by 'front'.**
- **Increment the 'front' to point to the next accessible data.**
- **Return success.**

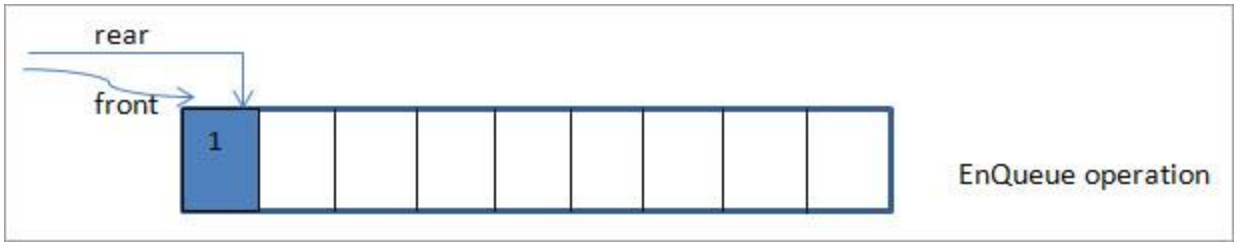
Next, we will see a detailed illustration of insertion and deletion operations in queue.

Illustration

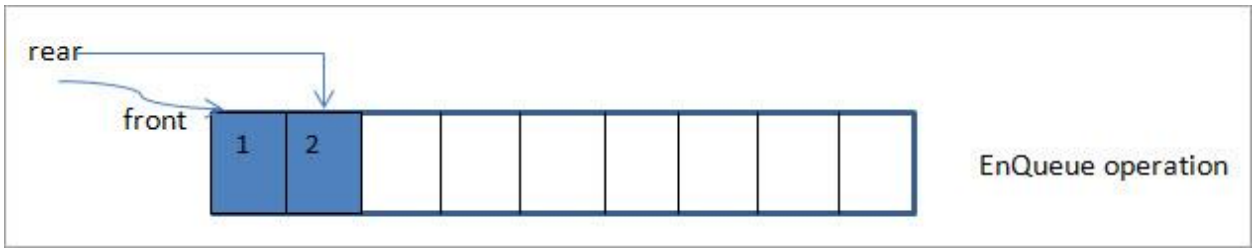


This is an empty queue and thus we have rear and empty set to -1.

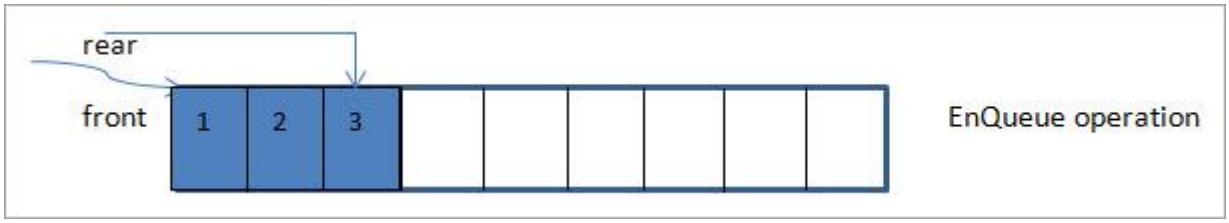
Next, we add 1 to the queue and as a result, the rear pointer moves ahead by one location.



In the next figure, we add element 2 to the queue by moving the rear pointer ahead by another increment.

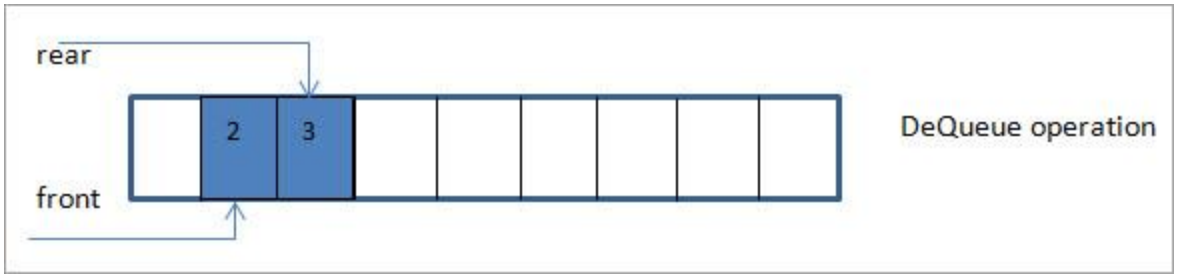


In the following figure, we add element 3 and move the rear pointer by 1.



At this point, the rear pointer has value 2 while the front pointer is at the 0th location.

Next, we delete the element pointed by the front pointer. As the front pointer is at 0, the element that is deleted is 1.



Thus the first element entered in the queue i.e. 1 happens to be the first element removed from the queue. As a result, after the first dequeue, the front pointer now will be moved ahead t0 the next location which is 1.

Array Implementation For Queue:

->Uses an array to store queue elements.
->Requires additional logic to handle wrap-around in circular queues.
->Can have fixed or dynamic size.

Using array Queues can be implemented as given below:

01_queue_stl.cpp | 02_queueImpliUsingArray.cpp

Queue > lec60 > 02_queueImpliUsingArray.cpp > Queue

```cpp
#include<bits/stdc++.h>
using namespace std;

class Queue{
    int *arr;
    int front;
    int rear;
    int size;

    public:
    Queue(){
        size = 100001;
        arr = new int[size];
        front = 0;
        rear = 0;
    }

    bool isEmpty(){
        if(front == rear){
            //cout << "Yes it is empty" << endl;
            return true;
        }
        else{
            //cout << "Not empty" << endl;
            return false;
        }
    }

    void enQueue(int data){
        if(rear == size){
            cout << "Queue is full" << endl;
```

```
32          }
33          else{
34              arr[rear] = data;
35              rear++;
36          }
37      }
38
39      int deQueue(){
40          if(front == rear){
41              //cout << "Queue is empty" << endl;
42              return -1;
43          }
44          else{
45              int ans = arr[front];
46              arr[front] = -1;
47              front++;
48              if(front == rear){
49                  front = 0;
50                  rear = 0;
51              }
52              return ans;
53          }
54      }
55
56      int frontElement(){
57          if(front == rear){
58              return -1;
59          }
60          else{
```

```
60          else{
61              return arr[front];
62          }
63      }
64
65  };
66
67  int main(){
68      Queue q;
69      q.enQueue(6);
70      q.enQueue(8);
71      q.enQueue(12);
72
73      cout << q.frontElement() << endl;
74
75      q.deQueue();
76      q.deQueue();
77      //q.deQueue();
78
79      cout << q.frontElement() << endl;
80
81      if(q.isEmpty()){
82          cout << "Queue is empty" << endl;
83      }
84      else{
85          cout << "Queue is not empty" << endl;
86      }
87      return 0;
88  }
```

Additional resources:

1. Queue - LeetCode
2. Queue Data Structure - GeeksforGeeks
3. Queue Data Structure - Tutorialspoint
4. Queues - Solve Data Structures | HackerRank

Practice problems as much as possible in order to get a good command on queue from above mentioned resources….
Happy coding…