Queue and Its Applications

Dr. GC Jana
Assistant Professor

Today’s discussion…

Queue

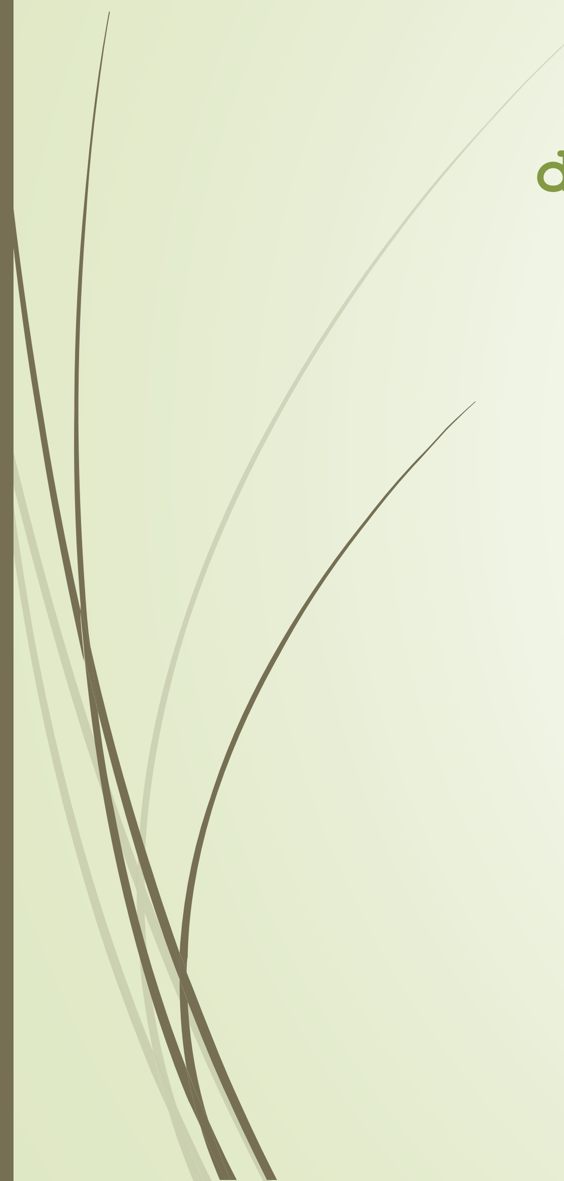
- Basic principles
- Operation of queue
- Queue using Array
- Queue using Linked List
- Applications of queue

Basic Idea

- Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks**, a queue is open at both its ends.** One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Works on FIFO-first in first out.

Queue Representation

- As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures.



enqueue

dequeue

create

isempty

size

QUEUE

QUEUE: First-In-First-Out (FIFO)

```
void enqueue (queue *q, int element);
                /* Insert an element in the queue */
int dequeue (queue *q);
                /* Remove an element from the queue */
queue *create();
                /* Create a new queue */
bool isempty (queue *q);
                /* Check if queue is empty */
int size (queue *q);
                /* Return the no. of elements in queue */
```

Assumption: queue contains integer elements!

```
#include <iostream>
using namespace std;
class Queue
{
public:
	int n=6;
	int ar[n];
	void create();
	void enqueue(int);
	int dequeue();

};

void create()
{
	int front = - 1;
	int rear = - 1;
}
```

```
void enqueue(int item)
{
	// checking overflow condition
	if (rear == n - 1)
	{
		cout<<"Overflow!"<<endl;
		return;
	}
	else
	{
		// front and rear both are at -1 then
		// set front and rear at 0 otherwise increment rear
		if (front == - 1 && rear==-1)
		{
			front = 0; rear=0;
		}
		else
			rear++;

		//inserting element at rear
		ar[rear] = item;
		cout<<"Element inserted"<<endl;
	}
}
```

```
void dequeue()
{
	if (front == - 1  | |  front > rear)
	{
		cout<<"Underflow!";
		return ;
	}
	else
	{
		int item=ar[front];
		cout<<"Element deleted from queue is : "<< item <<endl;
		// if front and rear reach at end then initialize it at -1
		if(front == rear)
		{
			front = -1;
			rear = -1;
		}
		else
			front++;
	}
}
```

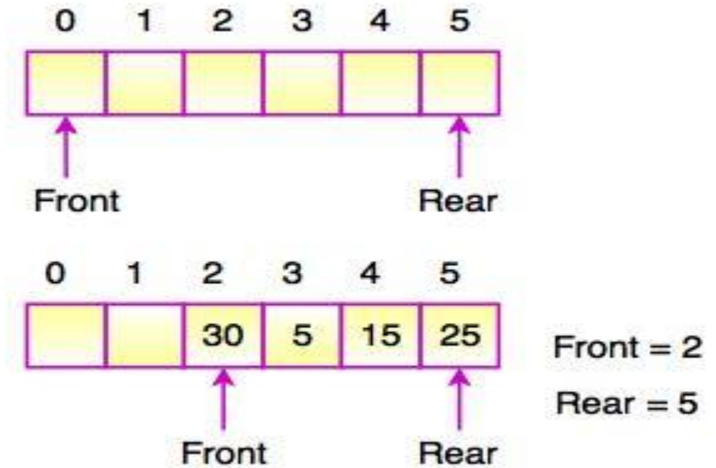


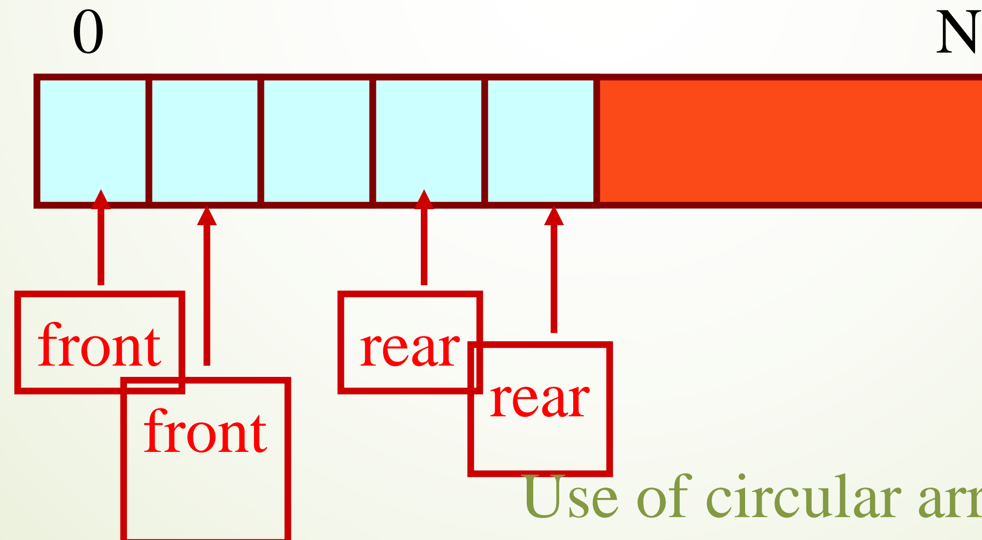
Fig. Implementation of Queue using Array

Problem with Array Implementation

- The size of the queue depends on the number and order of enqueue and dequeue.
- It may be situation where memory is available but enqueue is not possible.

ENQUEUE DEQUEUE

Effective queuing storage area of array gets reduced.



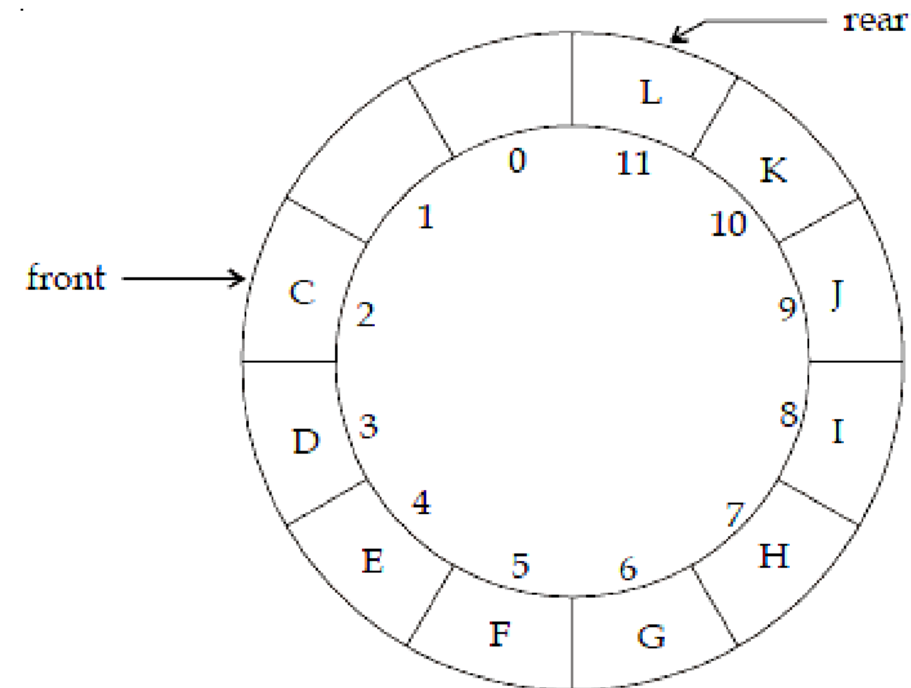
Use of circular array indexing

Circular Queue using Arrays

Array implementation Of Queue : For implementing a *queue*, we need to keep track of two indices - front and rear. We enqueue an item at the rear and dequeue an item from the front. If we simply increment front and rear indices, then there may be problems, the front may reach the end of the array. The solution to this problem is to increase front and rear in a circular manner.

Consider that an Array of size **N** is taken to implement a queue. Initially, the size of the queue will be zero(0). The total capacity of the queue will be the size of the array i.e. N. Now initially, the index *front* will be equal to 0, and *rear* will be equal to N-1.

Every time an item is inserted, so the index *rear* will increment by one, hence increment it as: **rear = (rear + 1)%N** and everytime an item is removed, so the front index will shift to right by 1 place, hence increment it as: **front = (front + 1)%N**

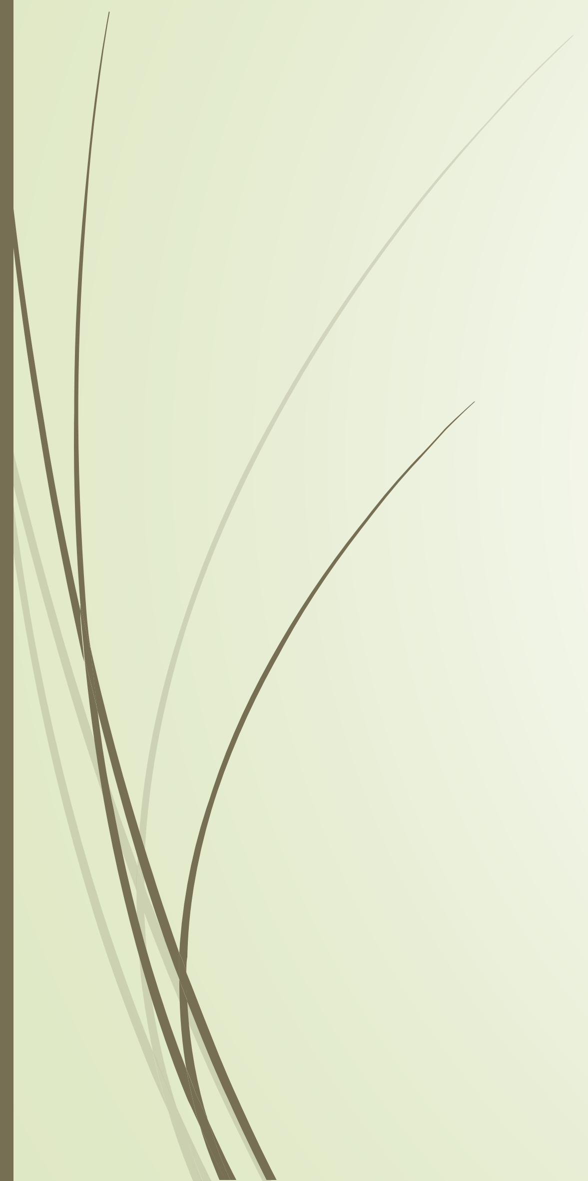


(c) Circular queue after deletion of two elements (A, B).

```
#include <iostream>
using namespace std;
class Queue
{
public:
        int n=6;
        int ar[n];
        void create();
        void enqueue(int);
        int dequeue();

};

void create()
{
        int front = - 1;
        int rear = - 1;
}
```

```
void enqueue(int item)
{
        // checking overflow condition
        if (rear == n - 1)
        {
                cout<<"Overflow!"<<endl;
                return;
        }
        else
        {
                // front and rear both are at -1 then
                // set front and rear at 0 otherwise increment rear
                if (front == - 1 && rear==-1)
                {
                        front = 0; rear=0;
                }
                else
                        rear=(rear+1)%n;

                //inserting element at rear
                ar[rear] = item;
                cout<<"Element inserted"<<endl;
        }
}
```

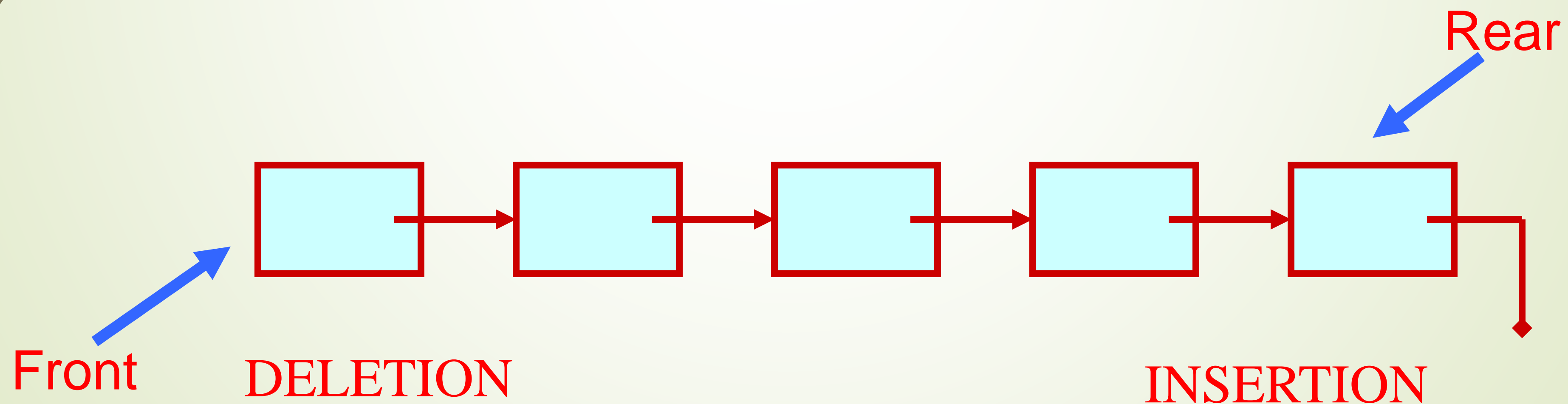
```
void dequeue()
{
    if (front == - 1 | | front > rear)
    {
        cout<<"Underflow!";
        return ;
    }
    else
    {
        int item=ar[front];
        cout<<"Element deleted from queue is : "<< item <<endl;
        // if front and rear reach at end then initialize it at -1
        if(front == rear)
        {
            front = -1;
            rear = -1;
        }
        else
            front=(front+1)%n;
    }
}
```



Queue using Linked List

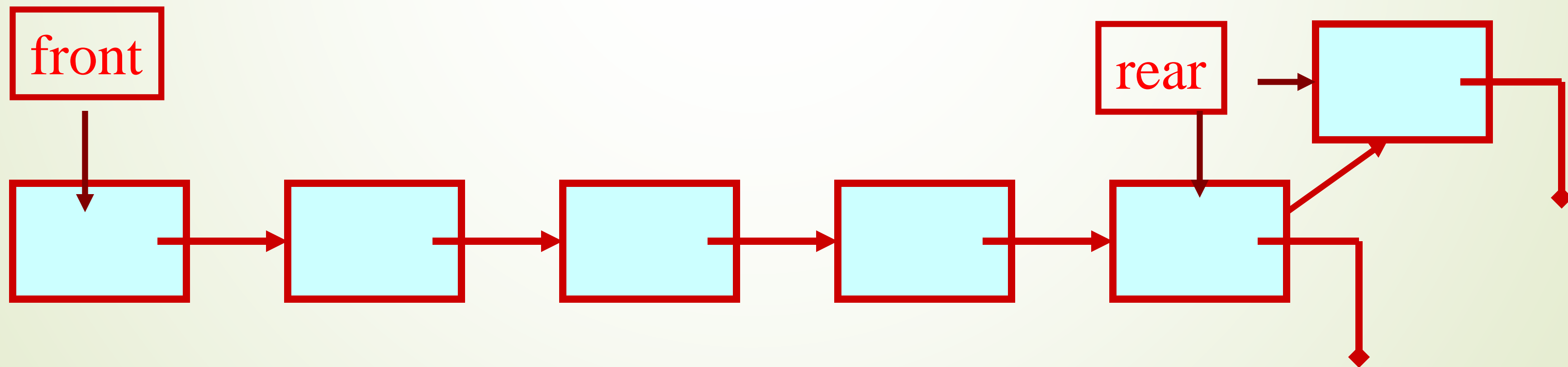
Basic Idea

- Create a linked list to which items would be added to one end and deleted from the other end.
- Two pointers will be maintained:
  - One pointing to the beginning of the list (point from where elements will be deleted).
  - Another pointing to the end of the list (point where new elements will be inserted).



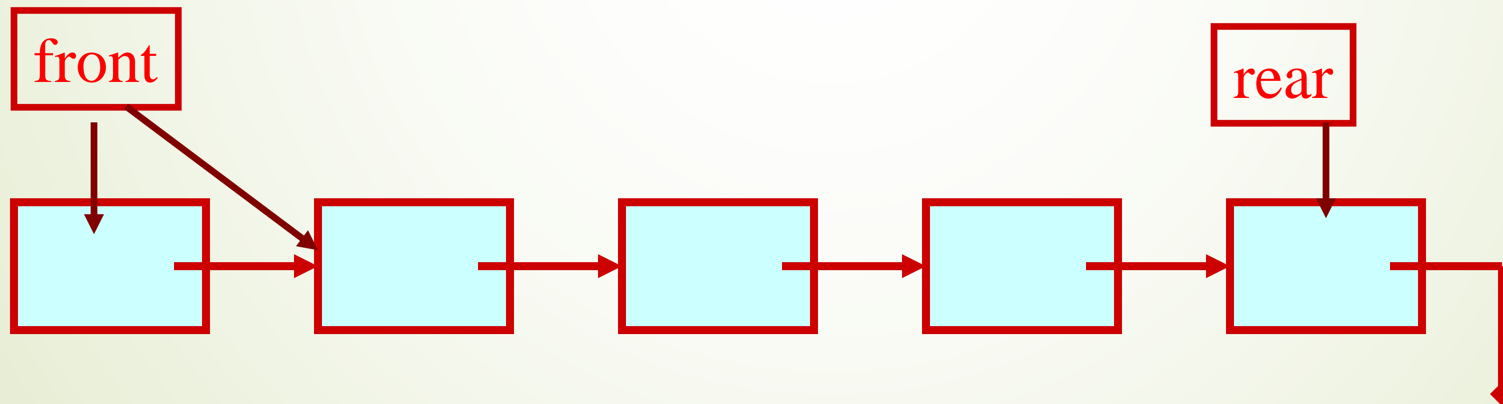
Queue: Linked List Structure

ENQUEUE



Queue: Linked List Structure

DEQUEUE



Example :Queue using Linked List

```
struct QueueNode{
    int data;
    QueueNode *next;
    QueueNode(int a){
        data = a;
        next = NULL;
    }
};

struct MyQueue {
    QueueNode *front;
    QueueNode *rear;
    void enqueue(int);
    int dequeue();
    MyQueue() {
        front = rear = NULL;
    }
};
```

```
void MyQueue:: enqueue(int x){

    QueueNode *qn=new QueueNode(x);
    if (front==NULL)
        front=rear=qn;
    else
    {
        rear->next=qn;
        rear=qn;
    }
}
```

```
int MyQueue :: dequeue(){

    if (front==NULL)
        return -1;
    QueueNode *temp=front;
    front=front->next;
    if (front==NULL)
        rear=NULL;
    return temp->data;
}
```

Applications of Queues

- Waiting lists
- Access to shared resources (e.g., printer)
- Auxiliary data structure for algorithms
- Data items to transfer through a Router on a network,
- To manage the ready queue, waiting queue, etc. for the execution of a task in CPU scheduling
- Job sequencing through an operating system,
- Priority management of the different tasks
- Manage Time-sharing system.
- Buffering data in I/O systems
- Producer-consumer problem
- Thread synchronization
- Graph Applications: BFS, Dijkstra's algorithm
- Traffic management

Queue in C++ STL

Method	Definition
queue::empty()	Returns whether the queue is empty. It return true if the queue is empty otherwise returns false.
queue::size()	Returns the size of the queue.
queue::swap()	Exchange the contents of two queues but the queues must be of the same data type, although sizes may differ.
queue::emplace()	Insert a new element into the queue container, the new element is added to the end of the queue.
queue::front()	Returns a reference to the first element of the queue.
queue::back()	Returns a reference to the last element of the queue.
queue::push(g)	Adds the element 'g' at the end of the queue.
queue::pop()	Deletes the first element of the queue.

Queue in C++ STL

- Methods

queue::empty()	O(1)
queue::size()	O(1)
queue::emplace()	O(1)
queue::front()	O(1)
queue::back()	O(1)
queue::push(g)	O(1)
queue::pop()	O(1)

```
// Print the queue
void print_queue(queue<int> q){
    queue<int> temp = q;
    while (!temp.empty()) {
        cout << temp.front()<<" ";
        temp.pop();
    }
    cout << '\n';
}
int main()
{
    queue<int> q1;
    q1.push(1);
    q1.push(2);
    q1.push(3);
    cout << "The first queue is : ";
    print_queue(q1);
    queue<int> q2;
    q2.push(4);
    q2.push(5);
    q2.push(6);


    cout << "The second queue is : ";
    print_queue(q2);

      q1.swap(q2);

    cout << "After swapping, the first queue is : ";
    print_queue(q1);
    cout << "After swapping the second queue is : ";
    print_queue(q2);

    cout<<q1.empty();  //returns false since q1 is not
empty
     return 0;
}
```

DQueue in C++

- Deque Data Structure
- Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends.

ADD ELEMENT AT REAR

ADD ELEMNET AT FRONT

REAR

FRONT

10	15	20	30	40	50	60	70

REMOVE ELEMENT FROM REAR

REMOVE ELEMENT FROM FRONT

DQueue in C++ STL

- **Operations on Deque:** Mainly the following four basic operations are performed on queue:
  insertFront(): Adds an item at the front of Deque.
  insertLast(): Adds an item at the rear of Deque.
  deleteFront(): Deletes an item from the front of Deque.
  deleteLast(): Deletes an item from the rear of Deque.
- In addition to the above operations, the following operations are also supported.

  getFront(): Gets the front item from the queue.
  getRear(): Gets the last item from queue.
  isEmpty(): Checks whether Deque is empty or not.
  isFull(): Checks whether Deque is full or not.
- **Some Practical Applications of Deque**:
  - Applied as both stack and queue, as it supports both operations.
  - Storing a web browser's history.
  - Storing a software application's list of undo operations.
  - Job scheduling algorithm

DQueue in C++ STL

Method	Definition
deque::insert()	Inserts an element. And returns an iterator that points to the first of the newly inserted elements.
deque::rbegin()	Returns a reverse iterator which points to the last element of the deque (i.e., its reverse beginning).
deque::rend()	Returns a reverse iterator which points to the position before the beginning of the deque (which is considered its reverse end).
deque::cbegin()	Returns a constant iterator pointing to the first element of the container, that is, the iterator cannot be used to modify, only traverse the deque.
deque::max_size()	Returns the maximum number of elements that a deque container can hold.
deque::assign()	Assign values to the same or different deque container.
deque::resize()	Function which changes the size of the deque.
deque::push_front()	It is used to push elements into a deque from the front.
deque::push_back()	This function is used to push elements into a deque from the back.

DQueue in C++ STL

deque::pop_front() and deque::pop_back()	pop_front() function is used to pop or remove elements from a deque from the front. pop_back() function is used to pop or remove elements from a deque from the back.
deque::front() and deque::back()	front() function is used to reference the first element of the deque container. back() function is used to reference the last element of the deque container.
deque::clear() and deque::erase()	clear() function is used to remove all the elements of the deque container, thus making its size 0. erase() function is used to remove elements from a container from the specified position or range.
deque::empty() and deque::size()	empty() function is used to check if the deque container is empty or not. size() function is used to return the size of the deque container or the number of elements in the deque container.
deque::operator= and deque::operator[]	operator= operator is used to assign new contents to the container by replacing the existing contents. operator[] operator is used to reference the element present at position given inside the operator.
deque::at() and deque::swap()	at() function is used reference the element present at the position given as the parameter to the function. swap() function is used to swap the contents of one deque with another deque of same type and size.
deque::begin() and deque::end	begin() function is used to return an iterator pointing to the first element of the deque container. end() function is used to return an iterator pointing to the last element of the deque container.

DQueue in C++ STL

```
void showdq(deque<int> g)
{
    deque<int>::iterator it;
    for (it = g.begin(); it != g.end(); ++it)
        cout << '\t' << *it;
    cout << '\n';
}

int main()
{
    deque<int> gquiz;
    gquiz.push_back(10);
    gquiz.push_front(20);
    gquiz.push_back(30);
    gquiz.push_front(15);
    cout << "The deque gquiz is : ";
    showdq(gquiz);
```

```
cout << "\ngquiz.size() : " << gquiz.size();
cout << "\ngquiz.max_size() : " << gquiz.max_size();

cout << "\ngquiz.at(2) : " << gquiz.at(2);
cout << "\ngquiz.front() : " << gquiz.front();
cout << "\ngquiz.back() : " << gquiz.back();

cout << "\ngquiz.pop_front() : ";
gquiz.pop_front();
showdq(gquiz);

cout << "\ngquiz.pop_back() : ";
gquiz.pop_back();
showdq(gquiz);

return 0;
```

PriorityQueue in C++ STL

A **C++ priority queue** is a type of container adapter, specifically designed such that the first element of the queue is either the greatest or the smallest of all elements in the queue, and elements are in non-increasing or non-decreasing order (hence we can see that each element of the queue has a priority {fixed order}).

In C++ STL, the top element is always the greatest by default. We can also change it to the smallest element at the top. Priority queues are built on the top of the max heap and use an array or vector as an internal structure. In simple terms, **STL Priority Queue** is the implementation of Heap Data Structure.

Syntax:

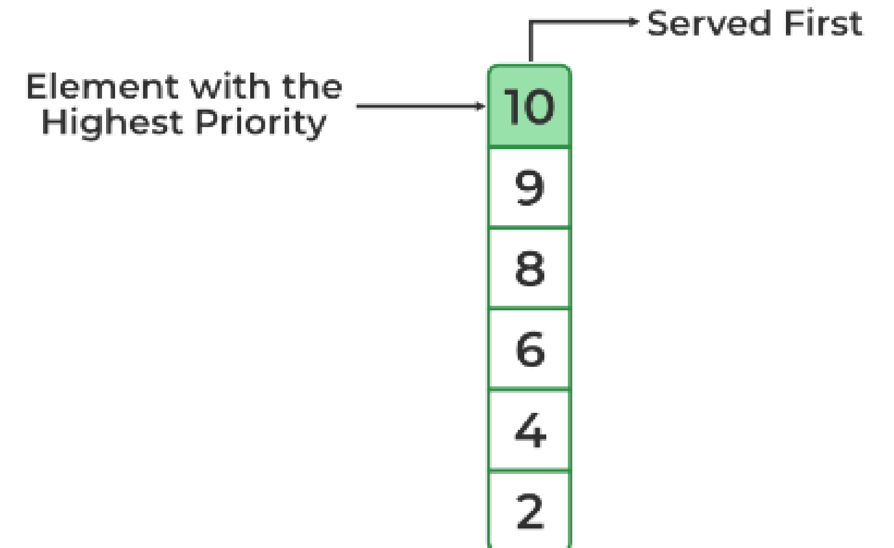
```
priority_queue<int> pq;      //MaxHeap

priority_queue <int, vector<int>, greater<int>> gq;
                             //MinHeap
```

PriorityQueue in C++ STL

```
#include <iostream>
#include <queue>
using namespace std;

// driver code
int main()
{
    int arr[6] = { 10, 2, 4, 8, 6, 9 };
    // defining priority queue
    priority_queue<int> pq;

    // printing array
    cout << "Array: ";
    for (auto i : arr) {
        cout << i << ' ';
    }
    cout << endl;
    // pushing array sequentially one by one
    for (int i = 0; i < 6; i++) {
        pq.push(arr[i]);
    }

    // printing priority queue
    cout << "Priority Queue: ";
    while (!pq.empty()) {
        cout << pq.top() << ' ';
        pq.pop();
    }

    return 0;
}
```

Output

```
Array: 10 2 4 8 6 9
Priority Queue: 10 9 8 6 4 2
```



Max Heap Priority Queue (default scheme)

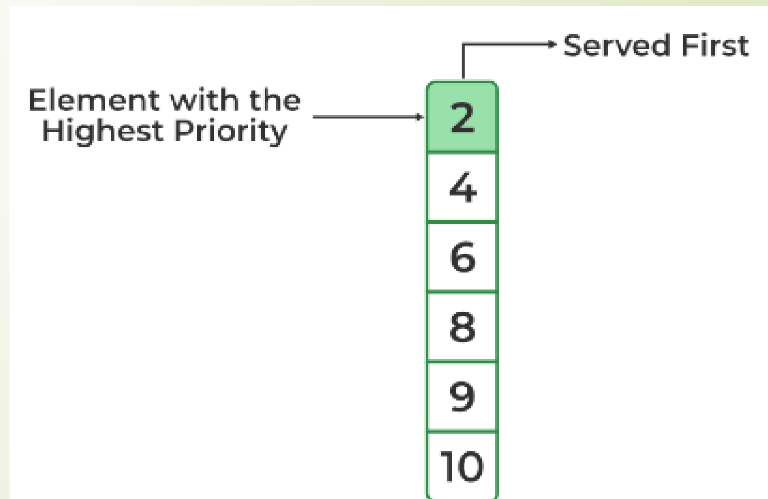
PriorityQueue in C++ STL

```
#include <iostream>
#include <queue>
using namespace std;

void showpq(
	priority_queue<int, vector<int>, greater<int> > g)
{
	while (!g.empty()) {
		cout << ' ' << g.top();
		g.pop();
	}
	cout << '\n';
}

void showArray(int* arr, int n)
{
	for (int i = 0; i < n; i++) {
		cout << arr[i] << ' ';
	}
	cout << endl;
}
```

```
// Driver Code
int main()
{
	int arr[6] = { 10, 2, 4, 8, 6, 9 };
	priority_queue<int, vector<int>,
greater<int> > gquiz(arr, arr + 6);

	cout << "Array: ";
	showArray(arr, 6);

	cout << "Priority Queue : ";
	showpq(gquiz);

	return 0;
}
```

Output

```
Array: 10 2 4 8 6 9
Priority Queue : 2 4 6 8 9 10
```



Min Heap Priority Queue

PriorityQueue in C++ STL

Methods of Priority Queue

The following list of all the methods of std::priority_queue class:

Method	Definition
priority_queue::empty()	Returns whether the queue is empty.
priority_queue::size()	Returns the size of the queue.
priority_queue::top()	Returns a reference to the topmost element of the queue.
priority_queue::push()	Adds the element 'g' at the end of the queue.
priority_queue::pop()	Deletes the first element of the queue.
priority_queue::swap()	Used to swap the contents of two queues provided the queues must be of the same type, although sizes may differ.
priority_queue::emplace()	Used to insert a new element into the priority queue container.
priority_queue value_type	Represents the type of object stored as an element in a priority_queue. It acts as a synonym for the template parameter.

main.cpp

```cpp
#include <iostream>
#include<queue>

using namespace std;
int main() {
  //max heap
  priority_queue<int> maxi;

  //min - heap
  priority_queue<int,vector<int> , greater<int> > mini;

  maxi.push(1);
  maxi.push(3);
  maxi.push(2);
  maxi.push(0);
  cout<<"size-> "<<maxi.size()<<endl;
  int n = maxi.size();
  for(int i=0;i<n;i++) {
    cout<<maxi.top()<<" ";
    maxi.pop();
  }cout<<endl;

  mini.push(5);
  mini.push(1);
  mini.push(0);
  mini.push(4);
  mini.push(3);

  int m = mini.size();
  for(int i=0;i<m;i++) {
    cout<<mini.top()<<" ";
    mini.pop();
  }cout<<endl;


cout<<"khaali h kya bhai "
```

Console Shell

```
> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
size-> 4
3 2 1 0
0 1 3 4 5
> 
```

Any question?