The Queue Data Structure

Queue is a linear data structure.

FRONT points to the beginning of the queue and REAR points to the end of the queue.

The element inserted in queue called REAR.

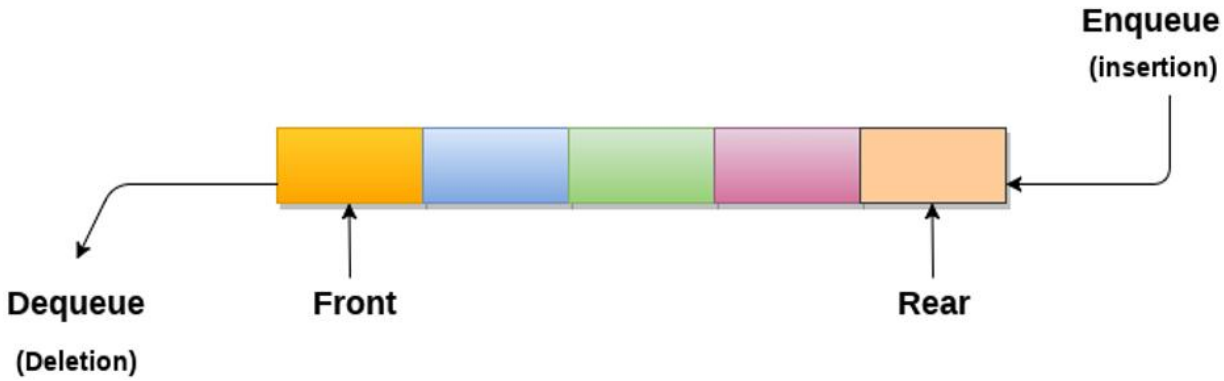
The elements deleted from queue called FRONT.

Queue follows the FIFO (First In First Out) structure.

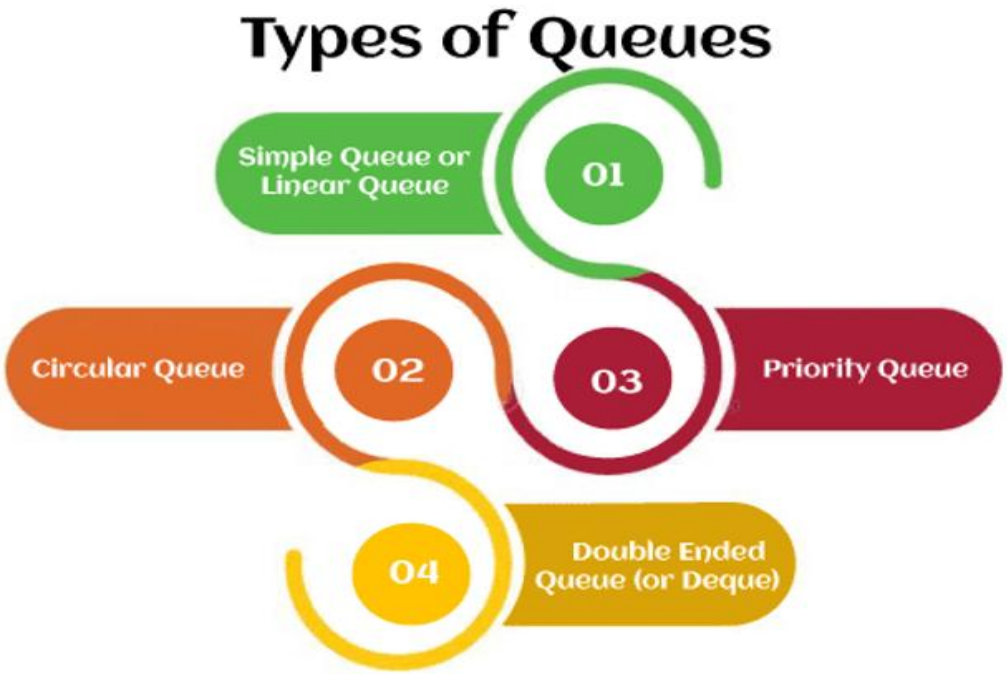
The process to add an element into queue is called Enqueue and the process of removal of an element from queue is called Dequeue.

Key Characteristics:

- Queue is a data structure where insertion takes place at the REAR of the queue and deletion takes place at the FRONT of the queue.



Types



Simple Queue or Linear Queue

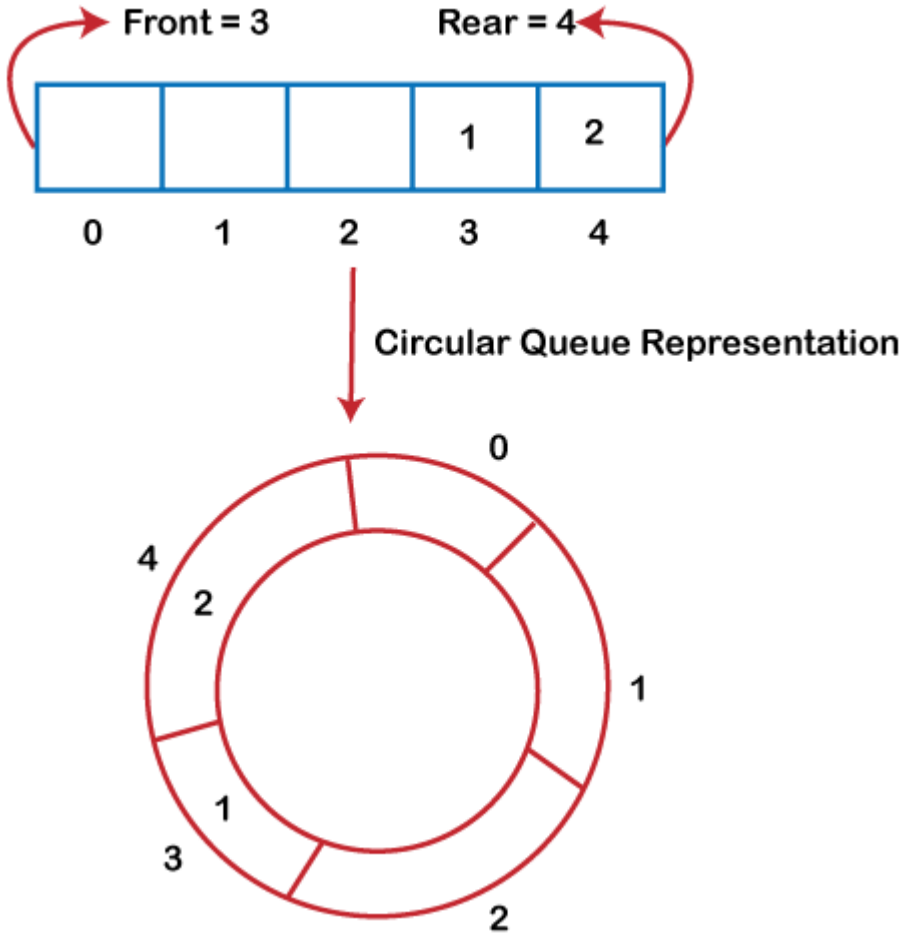
In Linear Queue, an insertion takes place from one end while the deletion occurs from another end.

The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.

Circular Queue

In Circular Queue, all the nodes are represented as circular.

It is similar to the linear Queue except that the last element of the queue is connected to the first element.



Circular Queue

The drawback that occurs in a linear queue is overcome by using the circular queue.

If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear.

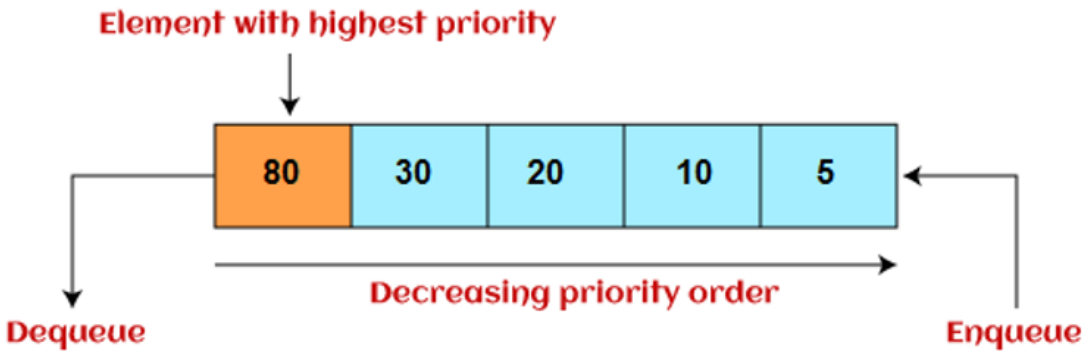
The main advantage of using the circular queue is better memory utilization

Priority Queue

It is a special type of queue in which the elements are arranged based on the priority.

It is a special type of queue data structure in which every element has a priority associated with it.

Suppose some elements occur with the same priority, they will be arranged according to the FIFO principle.

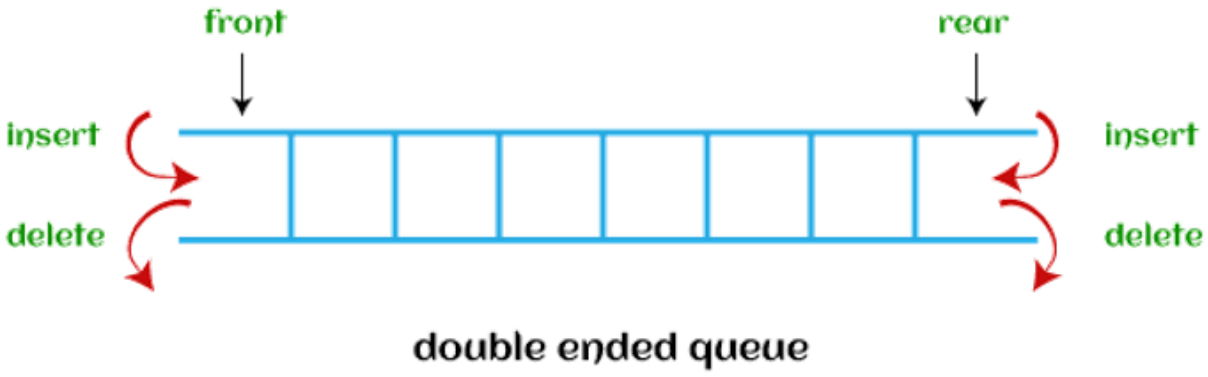


Deque (or, Double Ended Queue)

In Deque or Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear.

It means that we can insert and delete elements from both front and rear ends of the queue.

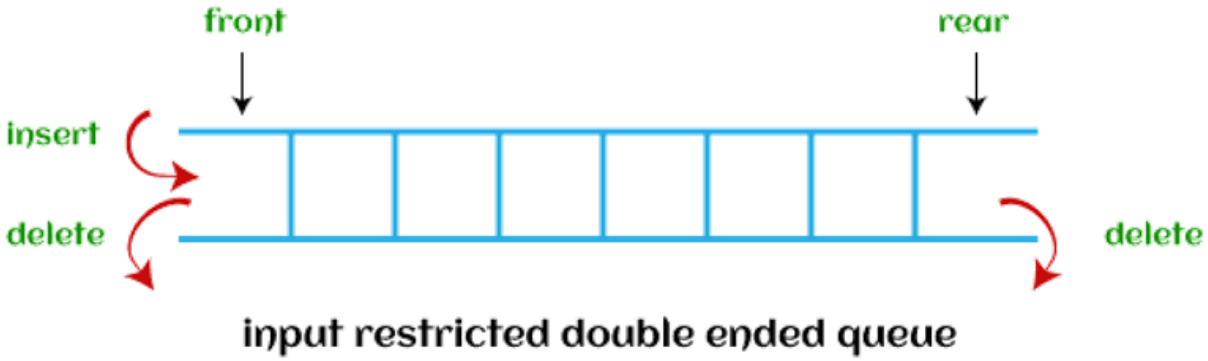
Deque can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.



Types of Deque

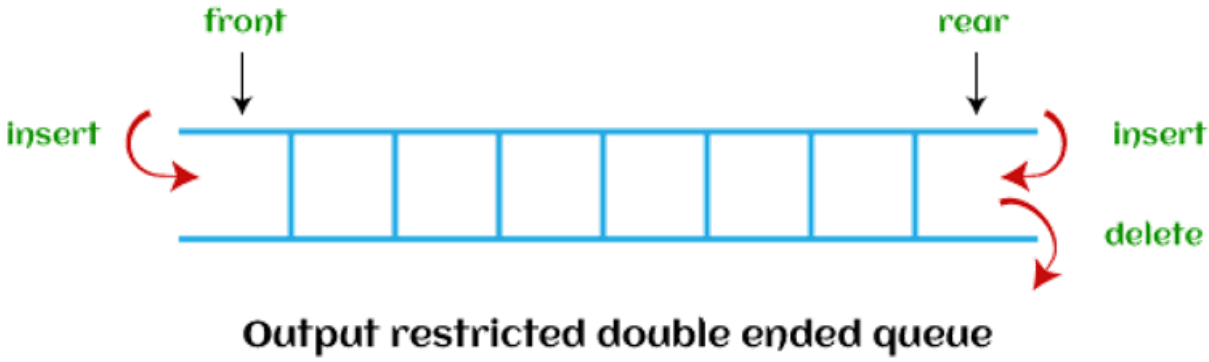
Input restricted dequeue:

As the name implies, in input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



Output restricted dequeue

As the name implies, in output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Application of queue

- Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
- In real life scenario, Call Center phone systems uses Queues to hold people calling.
- Buffer for devices like keyboard.
- Queues in routers/ switches .
- It can be used as a palindrome checker
- Traffic system

Operations of QUEUE

enqueue() – add (store) an item to the queue.

dequeue() – remove (access) an item from the queue.

peek() – Gets the element at the front of the queue without removing it.

isfull() – Checks if the queue is full.

isempty() – Checks if the queue is empty.

Algorithm for ENQUEUE operation

1. Check if the queue is full.
2. If the queue is full then print “Queue overflow”.
3. Else increment REAR by one.
4. Assign QUEUE[REAR]=element.

Algorithm for DEQUEUE operation

1. Check if the queue is empty.
2. If the queue is empty then print “Queue Underflow”.
3. Increment FRONT by one.

Algorithm for peek() operation

This function helps to see the data at the front of the queue.

int peek() {

```
  return queue[front];
}
```

Algorithm for isfull() operation

```
bool isfull() {
  if(rear == MAXSIZE - 1)
    return true;
  else
    return false;
}
```

Algorithm for isempty() operation

```
bool isempty() {
  if(rear == - 1 && front == - 1)
    return true;
  else
    return false;
}
```

Example

```
#include<iostream>
#define size 100
using namespace std;
int queue[size];
class queues{
    public:
        int rear , front;
        queues(){
```

```
        rear=-1;
        front=-1;
}
```

Example 2

```
//insertion
void enqueue(int x){
    if(front==-1){
        front++;
    }
    if(rear==size-1){
        cout<<"Queue is Overflow !!!";
    }else{
        rear++;
        queue[rear]=x;
    }
}
```

Example 3

```
//deletion
void dequeue(){
    if(front==-1){
        cout<<"Queue is Uderflow !!!";
    }else{
        front++;
    }
}
```

Example 4

```
//display
```

```
        void print(){
                for(int i=front;i<=rear;i++){
                        cout<<queue[i]<<endl;
                }
        }
};
```

Practical code:

```
#include <iostream>
#include <queue> // this library is used for defining simple queue, circular
queue
#include <deque> //it is used for pirority queue and double queue
using namespace std;
int main(){
//syntax : queue<DataType> queueName = {values};
//queueName.push(element)
queue<int> numbers, temp;
numbers.push(12);
numbers.push(14);
numbers.push(45);
numbers.push(47);
//numbers.pop();
//access
// cout<<numbers.front();
//traversal

//while(!numbers.empty()){
    //numbers.front();
//  numbers.pop();
//}
temp = numbers;

//for(int i = 0; i < numbers.size(); i++){
//  cout<<temp.front()<<"\n";
//  temp.pop();
//}

//---------------------------
//Double queue
```

```
deque<int> num;
num.push_back(12);
num.push_front(33);
num.push_back(23);

/*for(int i = 0; i < 3; i++){
    cout<<num.front()<<endl;
    num.pop_front();
}*/
//access
//cout<<num.font();
//-------------------------------------

priority_queue<int> abc, test;
abc.push(45);
abc.push(23);
abc.push(12);
//cout<<abc.top();

test = abc;

for (int i = 0; i < 3; i++){
    cout<<test.top()<<endl;
    test.pop();
}
//Traversal


return 0;
}
```