UNIT II

Stacks and Queues: Representation of Stacks, Representation of Queue, Operations on Stacks, Operations on Queues, Types of Queues.

2.1 Stacks And Queues

2.1.1 Representation of Stacks

Q1. What is stack? Write Stack ADT.

(Imp.)

Ans :

A *stack* is an ordered collection of homogeneous data elements where the insertion and deletion operations take place at one end only. The insertion and 'deletion operations in the case of a stack are specially termed PUSH and POP, respectively, and the position of the stack where these operations are performed is known as the TOP of the stack. An element in a stack is termed an ITEM. The maximum number of elements that a stack can accommodate is termed SIZE. Figure shows a typical view of a stack data structure.

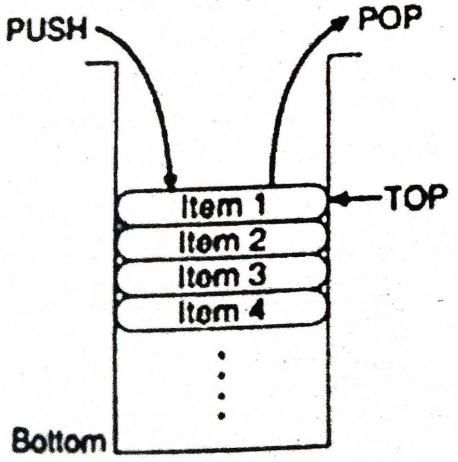
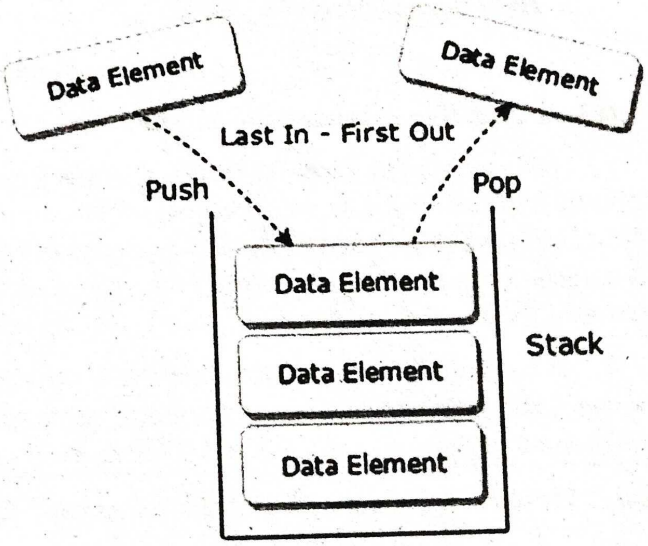


Fig.: Schematic diagram of a stack

Stack as ADT

He stacks of elements of any particular type is a finite sequence of elements of that type together with the following operations:

- Initialize the stack to be empty
- Determine whether the stack is empty or not
- Check whether the stack is full or not
- If the stack is not full, add or insert a new node at the top of the stack. This operation is termed as Push Operation
- If the stack is not empty, then retrieve the node at its top.
- If the stack is not empty, the delete the node at its top. This operation is called as Pop operation.



Q2. What are the various ways to represent the stack?

Ans :

A stack may be represented in the memory in various ways. There are two main ways: using a one-dimensional array and a single linked list.

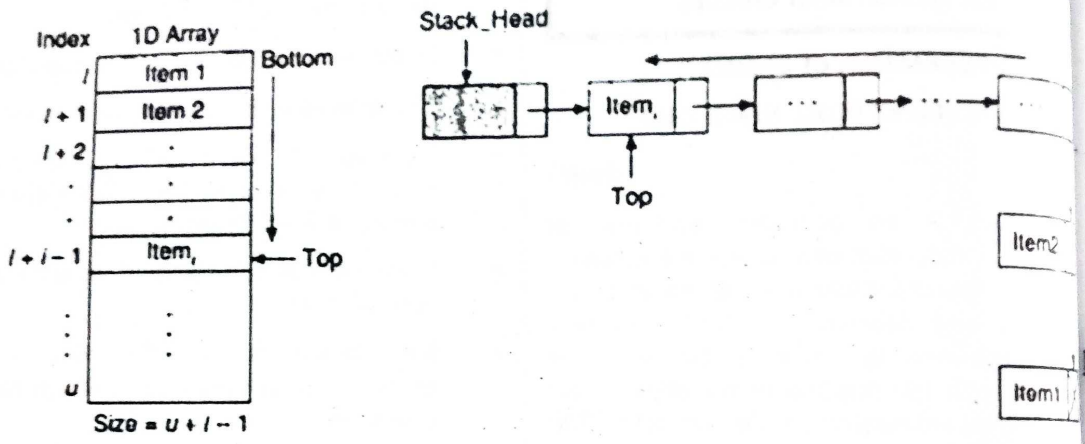
Array Representation of Stacks

First we have to allocate a memory block of sufficient size to accommodate the full capacity of the stack. Then, starting from the first location of the memory block, the items of the stack can be stored in a sequential fashion.

In Figure, Itemi denotes the *ith* item in the stack; l and *u* denote the index range of the array, usually the values of these indices are 1 and SIZE respectively. TOP is a pointer to point the position in array up to which it is filled with the items of the stack. With this representation, the following two tests can be stated:

EMPTY: TOP < l

FULL: TOP e" *u*



(a) Array representation of a stack (b) Linked list representation of a stack

Fig. : Two ways of representing stacks

Linked List Representation of Stacks

Although array representation of stacks is very easy and convenient but it allows the representation of only fixed sized stacks. In several applications, the size of the stack may vary during program execution. An obvious solution to this problem is to represent a stack using a linked list. A single linked list structure is sufficient to represent any stack. Here, the DATA field is for the ITEM, and the LINK field is, as usual, to point to the next' item. Above Figure b depicts such a stack using a single linked list.

In the linked list representation, the first node on the list is the current item that is the item at the top of the stack and the last node is the node containing the bottom-most item. Thus, a PUSH operation will add a new node in the front and a POP operation will remove a node from the front of the list.

Q3. How to represent the stack as assay? Explain.

Ans :

Representation as an Array

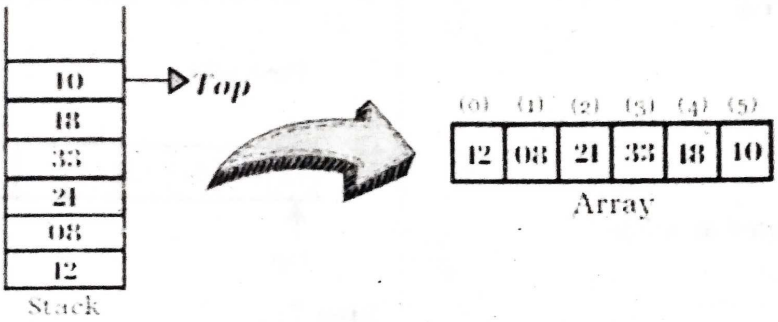
A stack is a data structure that can be represented as an array. Let us learn more about array representation of a stack.

An array is used to store an ordered list of elements. Using an array for representation of stack is one of the easy techniques to manage the data. But there is a major difference between an array and a stack.

- Size of an array is fixed.
- While, in a stack, there is no fixed size since the size of stack changed with the number of elements inserted or deleted to and from it.

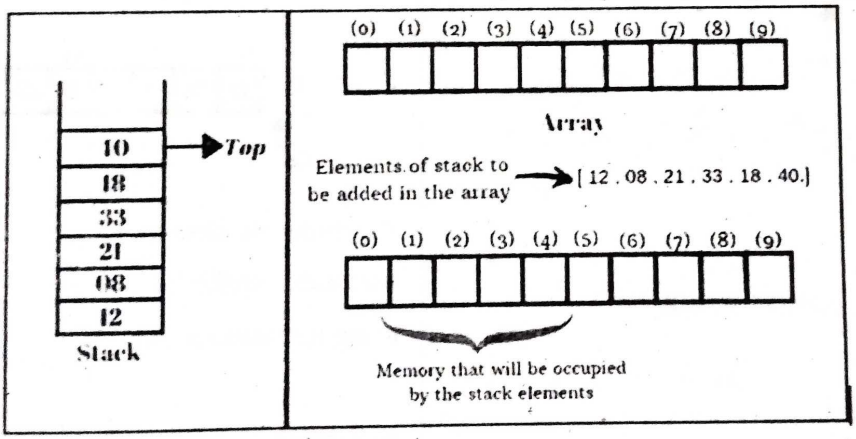
Despite the difference, an array can be used to represent a stack by taking an array of maximum size; big enough to manage a stack.

Stack as an Array



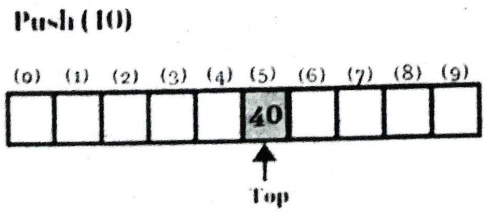
For Example:

We are given a stack of elements: 12 , 08 , 21 , 33 , 18 , 40.



Step 1:

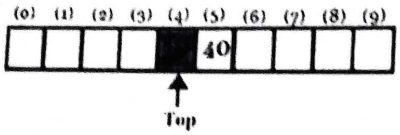
- Push (40).
- Top = 40
- Element is inserted at **a[5].**



Step 2:

- Push (18).
- Top = 18
- Element is inserted at **a[4]**.

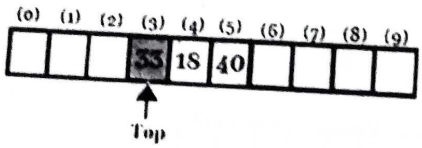
Push (18)



Step 3:

- Push (33).
- Top = 33
- Element is inserted at **a[3]**.

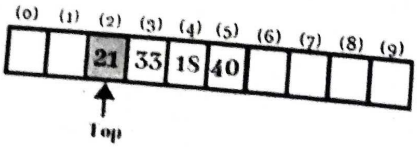
Push (33)



Step 4:

- Push (21).
- Top = 21
- Element is inserted at **a[2]**.

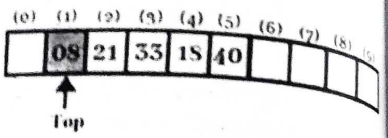
Push (21)



Step 5:

- Push (08).
- Top = 08
- Element is inserted at **a[1]**.

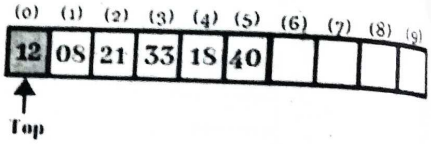
Push (08)



Step 6:

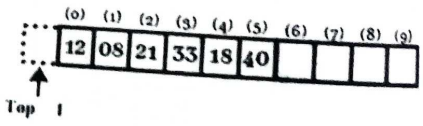
- Push (12).
- Top = 12
- Element is inserted at **a[0]**.

Push (12)



Step 7:

- Top = -1.
- End of array.




```
#include <iostream>
#include<stdlib.h>
using namespace std;

class stack {
   int stk[5];
   int top;

public:
   stack()
   {
      top = -1;
   }
```



```
void push(int x)
{
   if (top >= 4) {
      cout << "stack overflow
      return;
   }
   stk[++top] = x;
   cout << "inserted " <<
}
void pop()
{
   if (top < 0) {
      cout << "stack under
      return;
   }
   cout << "deleted " <<
}
void display()
{
   if (top < 0) {
      cout << " stack emp
= top; i >= 0; i—)
      cout << stk[i] <<
   }
};

int main()
{
   int ch;
   stack st;
   while (1) {
      cout << "\n1.push
      Enter ur choice: ";
      switch (ch) {
```

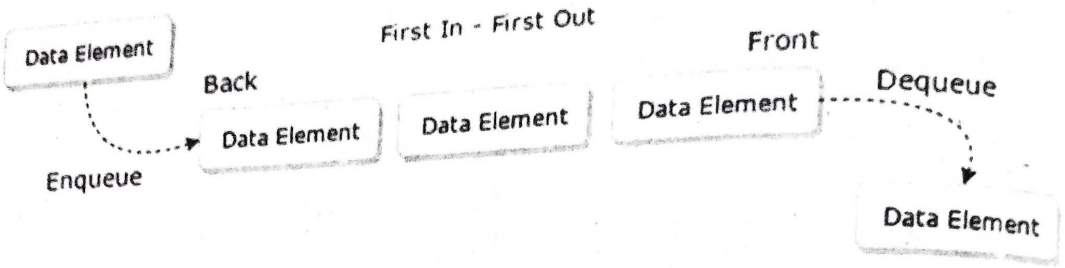
```
    void push(int x)
    {
      if (top >= 4) {
        cout << "stack overflow";
        return;
      }
      stk[++top] = x;
      cout << "inserted " << x;
    }
    void pop()
    {
      if (top < 0) {
        cout << "stack underflow";
        return;
      }
      cout << "deleted " << stk[top--];
    }
    void display()
    {
      if (top < 0) {
        cout << " stack empty"; return; } for (int i
= top; i >= 0; i--)
        cout << stk[i] << " ";
    }
};

int main()
{
  int ch;
  stack st;
  while (1) {
    cout << "\n1.push 2.pop 3.display 4.exit\n
    Enter ur choice: "; cin >> ch;
    switch (ch) {
    case 1:
      cout << "enter the element: "; cin >> ch;
      st.push(ch);
      break;
    case 2:
      st.pop();
      break;
    case 3:
      st.display();
      break;
    case 4:
      exit(0);
    }
  }
}
```

2.1.2 Representation of Queue

Q4. Define queue. Write queue ADT.

Ans : **(Imp.)**

Like a stack, a queue is an ordered collection of homogeneous data elements; in contrast with the stack, here, insertion and deletion operations take place at two extreme ends. A queue is also a linear data structure like an array, a stack and a linked list where the ordering of elements is in a-linear fashion. The only difference between a stack and a queue is that in the case of stack insertion and deletion (PUSH and POP) operations are at one end (TOP) only, but in a queue insertion (called ENQUEUE) and deletion (called DEQUEUE) operations take place at two ends called the REAR and FRONT of the queue, respectively. Figure represents a model of a queue structure. Queue is also termed first-in first-out (FIFO).



Queue as an ADT (Abstract Data Type)

The meaning of an abstract data type clearly says that for a data structure to be abstract, it should have the below-mentioned characteristics:

- First, there should be a particular way in which components are related to each other
- Second, a statement for the operation that can be performed on elements of abstract data type must have to be specified.

Thus for defining a Queue as an abstract data type, these are the following criteria:

- Initialize a queue to be empty
- Check whether a queue is empty or not
- Check whether a queue is full or not
- Insert a new element after the last element in a queue, if the queue is not full
- Retrieve the first element of the queue, if it is not empty
- Delete the first element in a queue, if it is not empty

Q5. Explain the array representation of queue with an example program,

Ans :

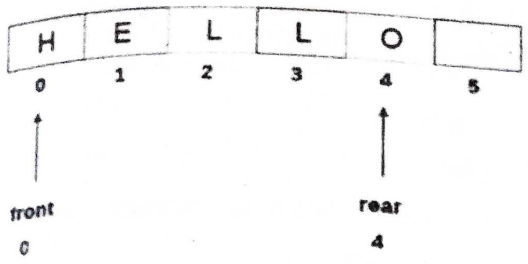
(Imp.)

There are two ways to represent a queue in memory: Using an array & Using a linked list.

The first kind of representation uses a one-dimensional array and it is a better choice where a queue of fixedsize is required. The other representation uses a double linked list and provides a queue whose size can varyduring processing.

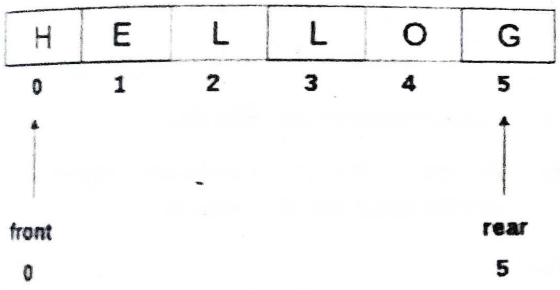
Array Representation of Queue

We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



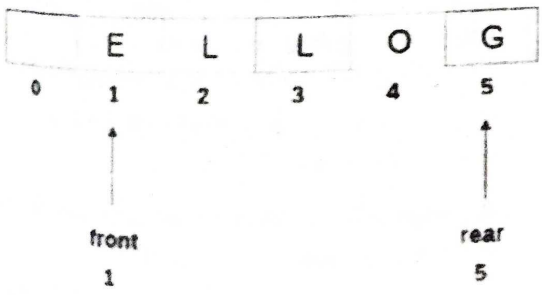
Queue

The above figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

Algorithm to Insert any Element in a Queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

- **Step 1:** IF REAR = MAX - 1
  Write OVERFLOW
  Go to step
  [END OF IF]
- **Step 2:** IF FRONT = -1 and REAR = -1
  SET FRONT = REAR = 0
  ELSE
  SET REAR = REAR + 1
  [END OF IF]
- **Step 3:** Set QUEUE[REAR] = NUM
- **Step 4:** EXIT

```
#include <iostream>
#include<stdlib.h>
using namespace std;
class queuearr {
        int queue1[5];
   int rear, front;

public:
   queuearr()
   {
      rear = -1;
      front = -1;
   }
   void insert(int x)
```

```
    {
      if (rear > 4) {
        cout << "queue over flow";
        front = rear = -1;
        return;
      }
      queue1[++rear] = x;
      cout << "inserted " << x;
    }

    void delet()
    {
      if (front == rear) {
        cout << "queue under flow";
        return;
      }
      cout << "deleted " << queue1[++front];
    }

    void display()
    {
      if (rear == front) {
        cout << " queue empty";
        return;
      }
      for (int i = front + 1; i <= rear; i++)
        cout << queue1[i] << " ";
    }
};

int main()
{
  int ch;
  queuearr qu;
  while (1) {
    cout << "\n1.insert 2.delet 3.display 4.e
    Enter ur choice: "; cin >> ch;
    switch (ch) {
    case 1:
      cout << "enter the element: "; cin >>
      qu.insert(ch);
      break;
    case 2:
      qu.delet();
      break;
    case 3:
      qu.display();
      break;
    case 4:
      exit(0);
    }
  }
}
```

2.1.3 Operations on Stacks

Q6. Write about various operations performed on the stack.

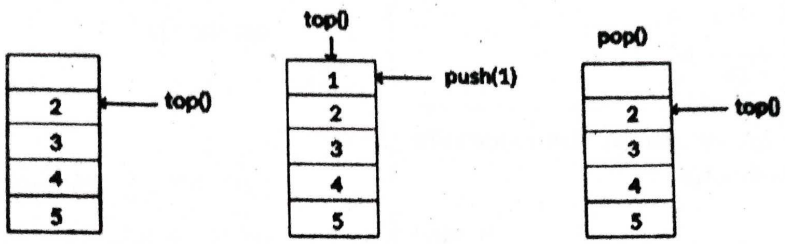
Ans :

(Imp.)

The most basic stack operations in the data structure are the following.

- **push()**: When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop()**: When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **isEmpty()**: It determines whether the stack is empty or not.
- **isFull()**: It determines whether the stack is full or not.

- **peek()**: It returns the element at the given position.
- **count()**: It returns the total number of elements available in a stack.
- **change()**: It changes the element at the given position.
- **display()**: It prints all the elements available in the stack.



1) **push()**

- Check if the stack is full.
- If the stack is full, then display **"Stack overflow"**.
- If the stack is not full, increment **top** to the next location.
- Assign data to the top element.

```
Algorithm_push()
If TOP >= SIZE – 1 then  //  Stack Overflow indicating that the Stack is FULL.
TOP = TOP + 1
STACK [TOP] = ELEMENT
```

2) **pop()**

- Check if the stack is empty.
- If the stack is empty, then display **"Stack Underflow"**.
- If the stack is not empty, copy top in a temporary variable.
- Decrement **top** to the previous location.
- Delete the temporary variable.

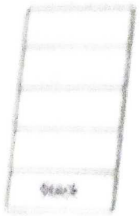
```
Algorithm_pop()
If TOP = -1 then  //  Stack Underflow indicating that the Stack is EMPTY.
Return STACK [TOP]
TOP = TOP - 1
```

3) **peek()**

- Print the top most element from the stack.

```
Algorithm_peek()
Return STACK [TOP]
```

Stack Operations



Q7. Write a c++ program to demonstrate the stack operations.

Ans : **(Imp.)**

```
#include<iostream>
using namespace std;
#define MAX 1000 //max size for stack
class Stack
{
  int top;
  public:
  int myStack[MAX]; //stack array

  Stack() { top = -1; }
  bool push(int x);
  int pop();
  bool isEmpty();
};

  //pushes element on to the stack
  bool Stack::push(int item)
  {
    if (top >= (MAX-1)) {
    cout << "Stack Overflow!!!";
    return false;
  }
else {
  myStack[++top] = item;
  cout<<item<<endl;
  return true;
  }
}

//removes or pops elements out of the stack
int Stack::pop()
{
  if (top < 0) {
    cout << "Stack Underflow!!";
    return 0;
  }
else {
    int item = myStack[top--];
    return item;
  }
}

//check if stack is empty
bool Stack::isEmpty()
{
  return (top < 0);
}

// main program to demonstrate stack functions
int main()
{
  class Stack stack;
  cout<<"The Stack Push "<<endl;
  stack.push(2);
  stack.push(4);
  stack.push(6);
  cout<<"The Stack Pop : "<<endl;
  while(!stack.isEmpty())
    {
    cout<<stack.pop()<<endl;
    }
  return 0;
}
```

Output:

```
The Stack Push
2
4
6
The Stack Pop:
6
4
2
```

2.1.4 Operations On Queues

Q8. What are the operations that can be performed on the queue? Explain.

Ans : **(Imp.)**

The queue data structure includes the following operations:

- **EnQueue:** Adds an item to the queue. Addition of an item to the queue is always done at the rear of the queue.
- **DeQueue:** Removes an item from the queue. An item is removed or de-queued always from the front of the queue.
- **isEmpty:** Checks if the queue is empty.
- **isFull:** Checks if the queue is full.
- **peek:** Gets an element at the front of the queue without removing it.

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.

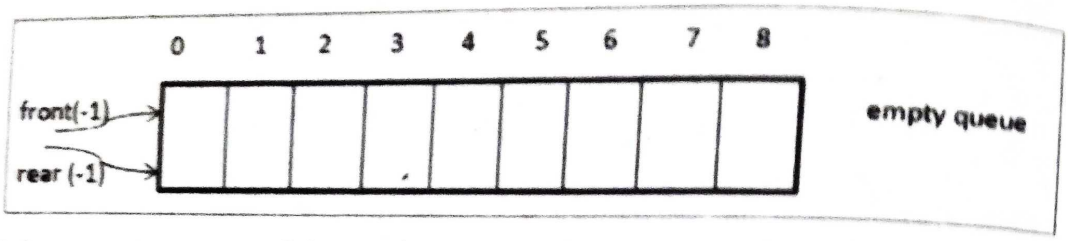
Dequeue

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.

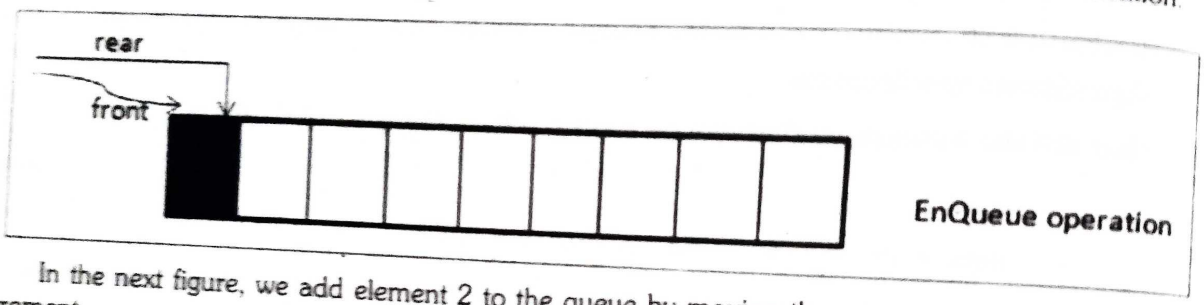
Next, we will see a detailed illustration of insertion and deletion operations in queue.

Illustration

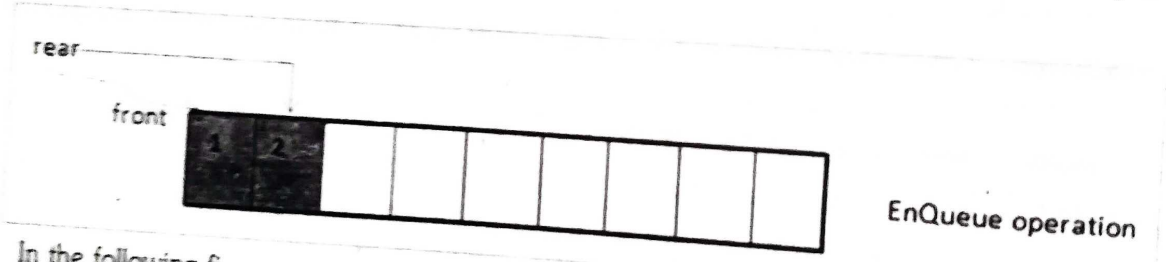


This is an empty queue and thus we have rear and empty set to -1.

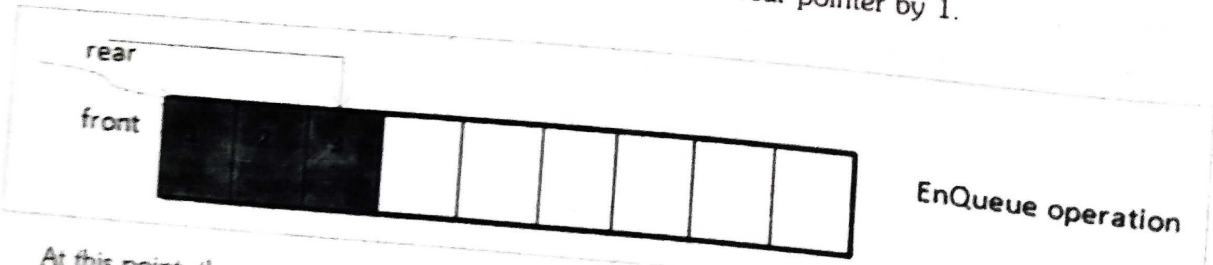
Next, we add 1 to the queue and as a result, the rear pointer moves ahead by one location.



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

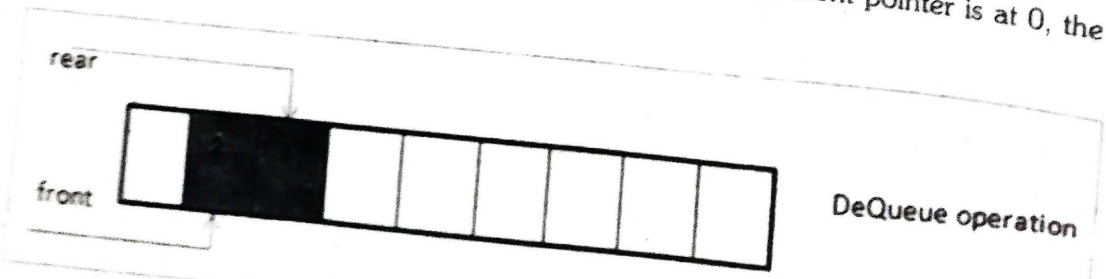


In the following figure, we add element 3 and move the rear pointer by 1.



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.



UNIT - II

Thus the
i.e. 1 happens to
the queue. As a
front pointer now
location which is

Q9. Write a queue op

Ans :

```
#include <iostr
#define MAX_S
usingnamespace
 classQueue {
private:
intmyqueue[MA
    public:
Queue(){
front = -1;
rear = -1;
    }
boolisFull(){
if(front == 0 &
returntrue;
        }
returnfalse;
    }
boolisEmpty(){
if(front == -1)
elsereturnfalse;
    }
voidenQueue(i
if(isFull()){
cout << endl <
        } els
if(front == -1)
rear++;
myque
```

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.

Q9. Write a c++ program to implement queue opearations.

Ans :

```
#include <iostream>
#define MAX_SIZE 5
usingnamespacestd;
 classQueue {
private:
intmyqueue[MAX_SIZE], front, rear;
    public:
Queue(){
front = -1;
rear = -1;
    }
boolisFull(){
if(front == 0 && rear == MAX_SIZE - 1){
returntrue;
        }
returnfalse;
    }
boolisEmpty(){
if(front == -1) returntrue;
elsereturnfalse;
    }
voidenQueue(intvalue){
if(isFull()){
cout << endl<< "Queue is full!!";
        } else{
if(front == -1) front = 0;
rear++;
myqueue[rear] = value;
cout << value << " ";
        }
    }
intdeQueue(){
intvalue;
if(isEmpty()){
cout << "Queue is empty!!"<< endl; return(-1);
} else{ value = myqueue[front]; if(front >= rear)
{       //only one element in queue
front = -1;
rear = -1;
            }
else{
front++;
            }
cout << endl << "Deleted => "<< value << "
from myqueue";
return(value);
        }
    }

    /* Function to display elements of Queue */
voiddisplayQueue()
    {
inti;
if(isEmpty()) {
cout << endl << "Queue is Empty!!"<< endl;
        }
else{
cout << endl << "Front = "<< front;
cout << endl << "Queue elements : ";
for(i=front; i<=rear; i++)
cout << myqueue[i] << "\t";
cout << endl << "Rear = "<< rear << endl;
        }
```

```
    }
};
intmain()
{
    Queue myq;

myq.deQueue();        //deQueue

cout<<"Queue created:"<<endl; myq.enQueue(10); myq.enQueue(20); myq.enQueue(30);
myq.enQueue(40); myq.enQueue(50); //enqueue 60 => queue is full
myq.enQueue(60);

myq.displayQueue();

    //deQueue =>removes 10
myq.deQueue();

    //queue after dequeue
myq.displayQueue();

return0;
}
```

Output:

```
Queue is empty!!
Queue created:
10   20   30   40   50
Queue is full!!
Front = 0
Queue elements : 10        20        30        40        50
Rear = 4
Deleted => 10 from myqueue
Front = 1
Queue elements: 20        30        40        50
Rear = 4
```

2.1.5 Types of Queues

Q10. Explain various types of queues.

Ans : **(Imp.)**

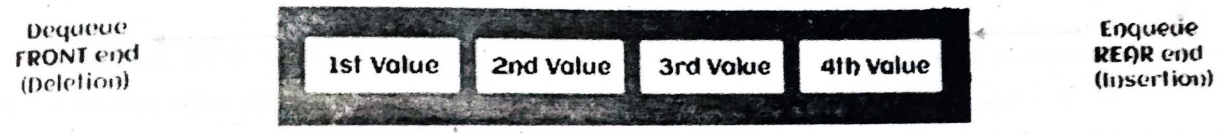
There are four different types of queue that are listed as follows :

1. Simple Queue or Linear Queue
2. Circular Queue
3. Priority Queue
4. Double Ended Queue (or Deque)

Let's discuss each of the type of queue.

1. Simple Queue or Linear Queue

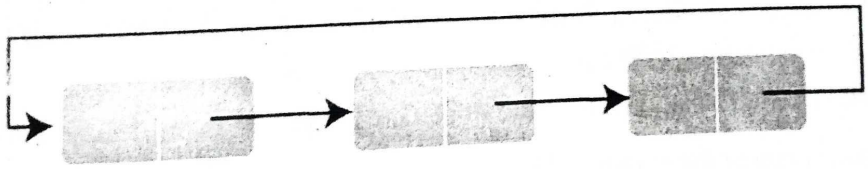
In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.



The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

2. 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. It is also known as Ring Buffer, as all the ends are connected to another end. The representation of circular queue is shown in the below image

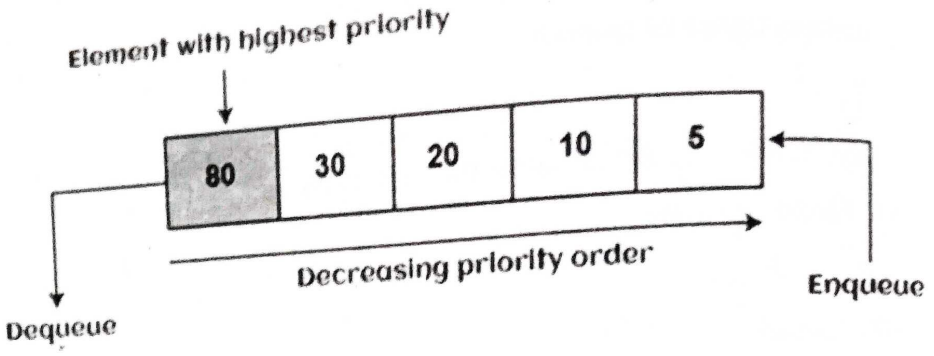


Circular Queue

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.

3. 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. The representation of priority queue is shown in the below image.



Insertion in priority queue takes place based on the arrival, while deletion in the priority ... occurs based on the priority. Priority queue is mainly used to implement the CPU scheduling algo...

There are two types of priority queue that are discussed as follows:

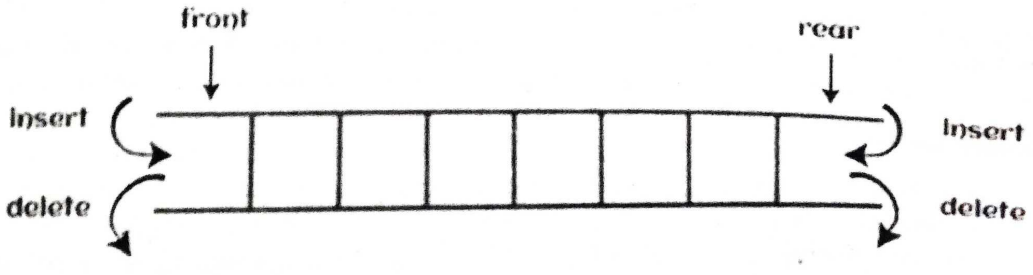
- **Ascending priority queue:** In ascending priority queue, elements can be inserted in arb... order, but only smallest can be deleted first. Suppose an array with elements 7, 5, and 3 in the ... order, so, insertion can be done with the same sequence, but the order of deleting the eleme... 3, 5, 7.
- **Descending priority queue:** In descending priority queue, elements can be inserted in arbi... order, but only the largest element can be deleted first. Suppose an array with elements 7, 3, a... 5 in the same order, so, insertion can be done with the same sequence, but the order of deleting ... elements is 7, 5, 3.

4. Deque (or, Double Ended Queue)

In Deque or Double Ended Queue, insertion and deletion can be done from both ends of t... queue either from the front or rear. It means that we can insert and delete elements from both front a... rear ends of the queue. Deque can be used as a palindrome checker means that if we read the string fro... both ends, then the string would be the same.

Deque can be used both as stack and queue as it allows the insertion and deletion operations o... both ends. Deque can be considered as stack because stack follows the LIFO (Last In First Out) princip... in which insertion and deletion both can be performed only from one end. And in deque, it is possible ... perform both insertion and deletion from one end, and Deque does not follow the FIFO principle.

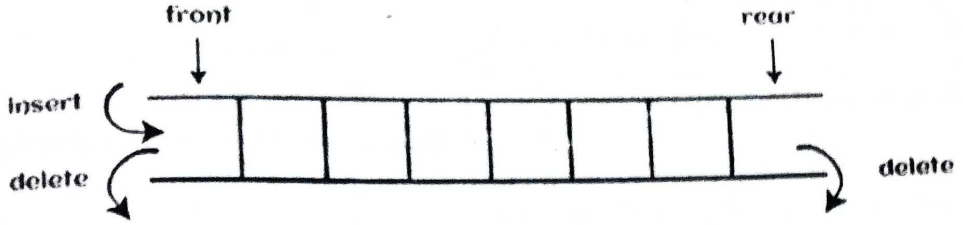
The representation of the deque is shown in the below image:



double ended queue

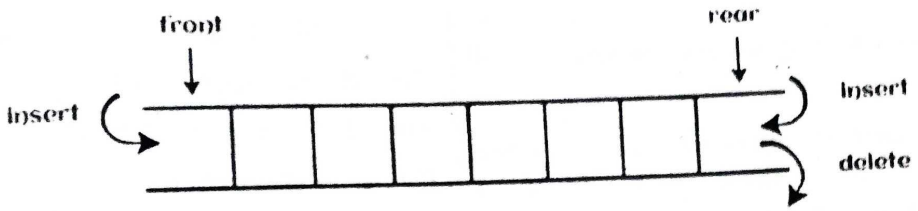
There are two types of deque that are discussed as follows:

- **Input restricted deque:** 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.



Input restricted double ended queue

- **Output restricted deque:** 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.



Output restricted double ended queue

Q11. What is a Circular Queue? Explain its working.

Ans :

A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a ***Ring Buffer***.

Operations on Circular Queue

The following are the operations that can be performed on a circular queue:

- **Front:** It is used to get the front element from the Queue.
- **Rear:** It is used to get the rear element from the Queue.
- **enQueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
- **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end.

Applications of Circular Queue

The circular Queue can be used in the following scenarios:

- **Memory management:** The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.

- **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.
- **Traffic system:** In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every jinterval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

Enqueue Operation

The steps of enqueue operation are given below:

- First, we will check whether the Queue is full or not.
- Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
- When we insert a new element, the rear gets incremented, i.e., ***rear=rear+1***.

Scenarios for inserting an element

There are two scenarios in which queue is not full:

- **If rear != max - 1**, then rear will be incremented to **mod(maxsize)** and the new value will be inserted at the rear end of the queue.
- **If front != 0 and rear = max - 1**, it means that queue is not full, then set the value of rear to 0 and insert the new element there.

There are two cases in which the element cannot be inserted:

- When **front ==0 && rear = max-1**, which means that front is at the first position of the Queue and rear is at the last position of the Queue.

front== rear + 1;

Algorithm to insert an element in a circular queue

Step 1: IF (REAR+1)%MAX = FRONT
Write " OVERFLOW "
Goto step 4
[End OF IF]

Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE IF REAR = MAX - 1 and FRONT != 0
SET REAR = 0
ELSE
SET REAR = (REAR + 1) % MAX
[END OF IF]

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT

Dequeue Operation

The steps of dequeue operation are given below:

- First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the value of front gets decremented by 1.
- If there is only one element left which is to be deleted, then the front and rear are reset to -1.

Algorithm to delete an element from the circular queue

Step 1: IF FRONT = -1
Write " UNDERFLOW "
Goto Step 4
[END of IF]

Step 2: SET VAL = QUEUE[FRONT]

Step 3: IF FRONT = REAR
SET FRONT = REAR = -1
ELSE
IF FRONT = MAX -1

SET FRONT = 0

ELSE

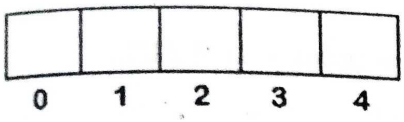
SET FRONT = FRONT + 1

[END of IF]

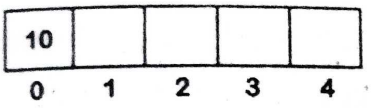
[END OF IF]

Step 4: EXIT

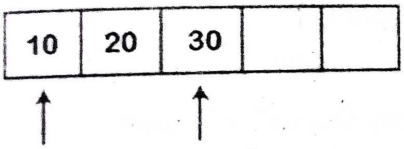
Let's understand the enqueue and dequeue operation through the diagrammatic representation.

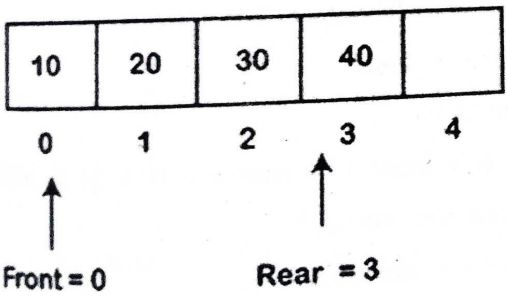


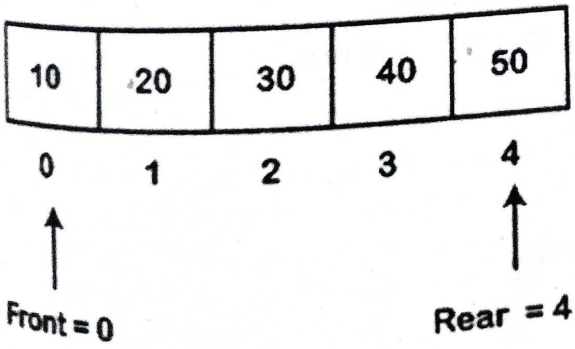
Front = -1
Rear = -1

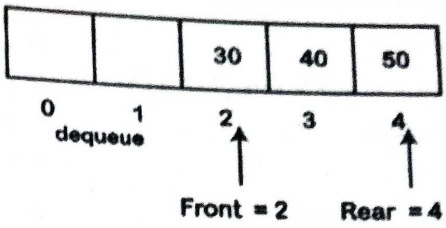


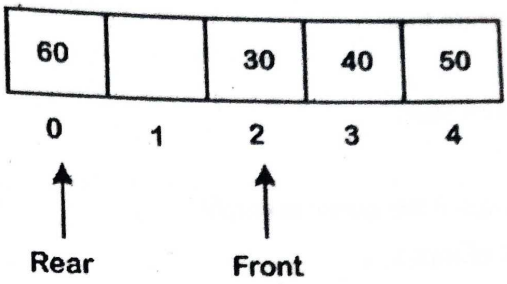
Front = 0
Rear = 0

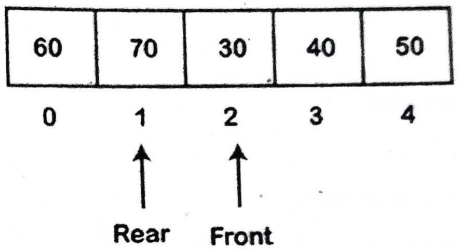












Q12. Write a C++ program to implement circular queue using Array.

Ans :

Implementation of circular queue using Array

```
#include <iostream>
#define SIZE 5 /* Size of Circular Queue */


using namespace std;
class Queue {
  private:
  int items[SIZE], front, rear;
  public:
  Queue() {
    front = -1;
    rear = -1;
  }
```

```
// Check if the queue is full
bool isFull() {
 if (front == 0 && rear == SIZE - 1) {
  return true;
 }
 if (front == rear + 1) {
  return true;
 }
 return false;
}
// Check if the queue is empty
bool isEmpty() {
 if (front == -1)
  return true;
 else
  return false;
}
// Adding an element
void enQueue(int element) {
 if (isFull()) {
  cout << "Queue is full";
 } else {
  if (front == -1) front = 0;
  rear = (rear + 1) % SIZE;
  items[rear] = element;
  cout << endl
<< "Inserted " << element << endl;
 }
}
// Removing an element
int deQueue() {
 int element;
 if (isEmpty()) {
  cout << "Queue is empty" << endl;
  return (-1);
 } else {
  element = items[front];
  if (front == rear) {
   front = -1;
   rear = -1;
  }
  // Q has only one element,
  // so we reset the queue after deleting it.
  else {
   front = (front + 1) % SIZE;
  }
  return (element);
 }
}

void display() {
 // Function to display status of Circular Queue
 int i;
 if (isEmpty()) {
  cout << endl
<< "Empty Queue" << endl;
 } else {
  cout << "Front -> " << front;
  cout << endl
<< "Items -> ";
  for (i = front; i != rear; i = (i + 1) % SIZE)
   cout << items[i];
  cout << items[i];
  cout << endl
<< "Rear -> " << rear;
 }
}
};
int main() {
 Queue q;
```



```
Fails because front = -1
.deQueue();

.enQueue(1);
.enQueue(2);
.enQueue(3);
.enQueue(4);
.enQueue(5);

// Fails to enqueue becau
= SIZE - 1
q.enQueue(6);
q.display();
int elem = q.deQueue(

if (elem != -1)
 cout << endl
< "Deleted Element is
q.display();
q.enQueue(7);
q.display();

// Fails to enqueue b
q.enQueue(8);
return 0;
```

Q13. What is a prio it.

Ans.

A priority que behaves similarly to each element has with the highest prio queue. The priorit queue will determi are removed from

```
// Fails because front = -1
q.deQueue();

q.enQueue(1);
q.enQueue(2);
q.enQueue(3);
q.enQueue(4);
q.enQueue(5);

// Fails to enqueue because front == 0 && rear
== SIZE - 1
q.enQueue(6);
q.display();
int elem = q.deQueue();

if (elem != -1)
  cout << endl
<< "Deleted Element is " << elem;
q.display();
q.enQueue(7);
q.display();

// Fails to enqueue because front == rear + 1
q.enQueue(8);
return 0;
}
```

Q13. What is a priority queue? Explain about it.

Ans :

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Characteristics of a Priority Queue

A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

Let's understand the priority queue through an example.

We have a priority queue that contains the following values:

1, 3, 4, 8, 14, 22

All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

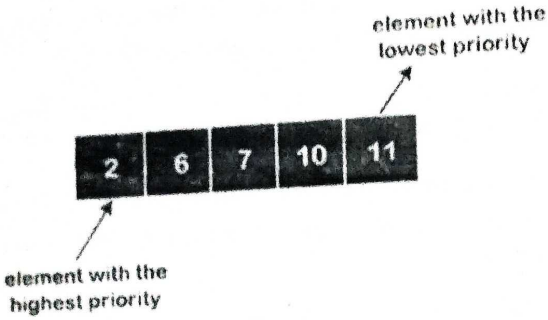
- **poll():** This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- **add(2):** This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
- **poll():** It will remove '2' element from the priority queue as it has the highest priority queue.
- **add(5):** It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

Types of Priority Queue

There are two types of priority queue:

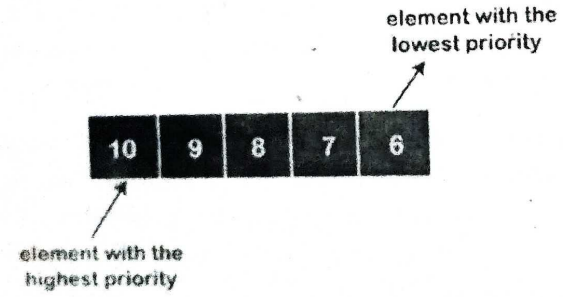
Ascending Order Priority Queue

In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.



Descending Order Priority Queue

In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.



Representation of Priority Queue

Now, we will see how to represent the priority queue through a one-way list.

We will create the priority queue by using the list given below in which **INFO** list contains the data elements, **PRN** list contains the priority numbers of each data element available in the **INFO** list, and LINK basically contains the address of the next node.

	INFO	PNR	LINK
0	200	2	4
1	400	4	2
2	600	4	0
3	300	1	0
4	100	2	5
5	600	3	1
6	700	4	

Let's create the priority queue step by step

In the case of priority queue, lower priority number is considered the higher priority, i.e., priority number = higher priority.

Step 1:

In the list, lower priority number is 1, whose data value is 333, so it will be inserted in the list as shown in the below diagram:

Step 2:

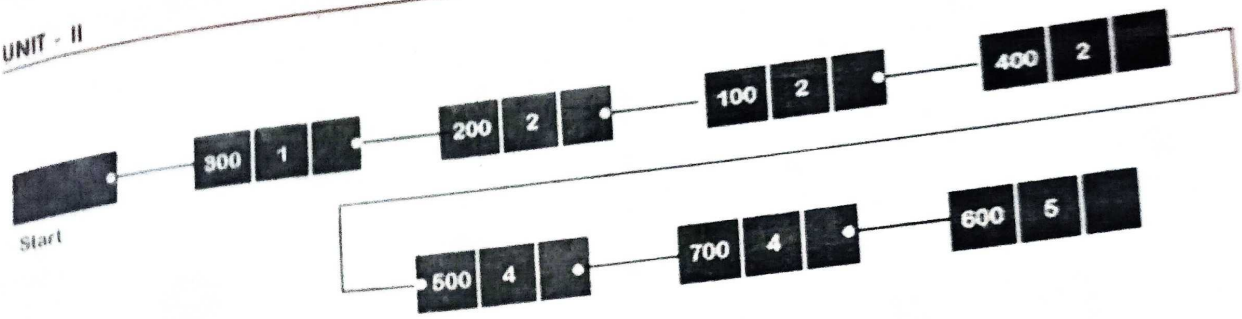
After inserting 333, priority number 2 having a higher priority, and data values associated with this priority are 222 and 111. So, this data will be inserted based on the FIFO principle; therefore 222 will be added first and then 111.

Step 3:

After inserting the elements of priority 2, the next higher priority number is 4 and the elements associated with 4 priority numbers are 444, 555, 777. In this case, elements would be inserted based on the FIFO principle; therefore, 444 will be added first, then 555, and then 777.

Step 4:

After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 666, so it will be inserted at the end of the queue.



Implementation of Priority Queue

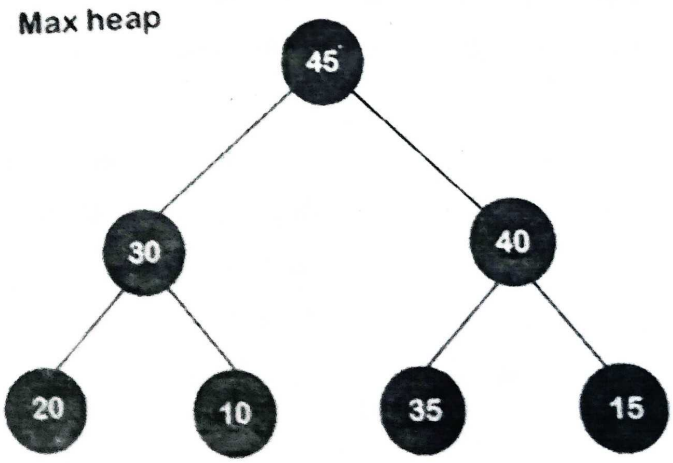
The priority queue can be implemented in four ways that include arrays, linked list, heap data structure and binary search tree. The heap data structure is the most efficient way of implementing the priority queue, so we will implement the priority queue using a heap data structure in this topic. Now, first we understand the reason why heap is the most efficient way among all the other data structures.

Q14. What is Heap? Write about , how we use max an min heaps in priority queues. **(Imp.)**

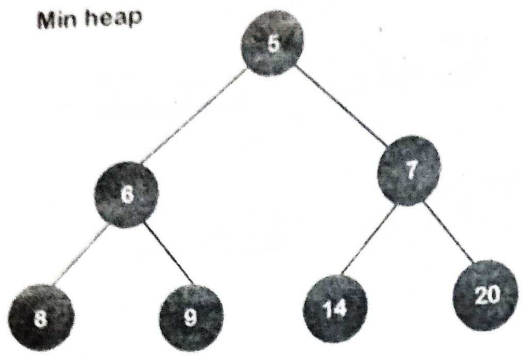
Ans :

A heap is a tree-based data structure that forms a complete binary tree, and satisfies the heap property. If A is a parent node of B, then A is ordered with respect to the node B for all nodes A and B in a heap. It means that the value of the parent node could be more than or equal to the value of the child node, or the value of the parent node could be less than or equal to the value of the child node. Therefore, we can say that there are two types of heaps:

- **Max heap:** The max heap is a heap in which the value of the parent node is greater than the value of the child nodes.



- **Min heap:** The min heap is a heap in which the value of the parent node is less than the value of the child nodes.



Both the heaps are the binary heap, as each has exactly two child nodes.

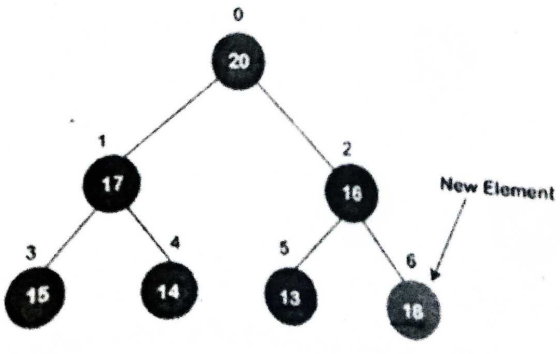
Priority Queue Operations

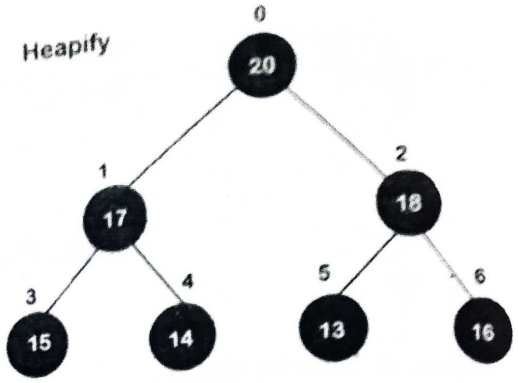
The common operations that we can perform on a priority queue are insertion, deletion and peek. Let's see how we can maintain the heap data structure.

Inserting the element in a priority queue (max heap)

If we insert an element in a priority queue, it will move to the empty slot by looking from top to bottom and left to right.

If the element is not in a correct place then it is compared with the parent node; if it is found out of order, elements are swapped. This process continues until the element is placed in a correct position.





Removing the minimum element from the priority queue

As we know that in a max heap, the maximum element is the root node. When we remove the root node, it creates an empty slot. The last inserted element will be added in this empty slot. Then, this element is compared with the child nodes, i.e., left-child and right child, and swap with the smaller of the two. It keeps moving down the tree until the heap property is restored.

Q15. Write a C++ Program to create the priority queue using the binary max heap.

Ans : **(Imp.)**

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

// Function to swap position of two elements
void swap(int *a, int *b) {
  int temp = *b;
  *b = *a;
  *a = temp;
}

// Function to heapify the tree
void heapify(vector<int>&hT, int i) {
```



```
int size = hT.size();

// Find the largest amon
child
int largest = i;
int l = 2 * i + 1;
int r = 2 * i + 2;
if (l < size && hT[l]
  largest = l;
if (r < size && hT[r]
  largest = r;

// Swap and continu
if (largest != i) {
  swap(&hT[i], &h
  heapify(hT, large
}
}

// Function to inse
void insert(vector
int size = hT.siz
if (size == 0) {
  hT.push_bac
} else {
  hT.push_ba
  for (int i =
    heapify(hT
  }
}
}

// Function t
void deleteN
int size =
```

```
  int size = hT.size();

  // Find the largest among root, left child and right
child
  int largest = i;
  int l = 2 * i + 1;
  int r = 2 * i + 2;
  if (l < size && hT[l] > hT[largest])
    largest = l;
  if (r < size && hT[r] > hT[largest])
    largest = r;

  // Swap and continue heapifying if root is not largest
  if (largest != i) {
    swap(&hT[i], &hT[largest]);
    heapify(hT, largest);
  }
}

// Function to insert an element into the tree
void insert(vector<int>&hT, int newNum) {
  int size = hT.size();
  if (size == 0) {
    hT.push_back(newNum);
  } else {
    hT.push_back(newNum);
    for (int i = size / 2 - 1; i >= 0; i--) {
      heapify(hT, i);
    }
  }
}

// Function to delete an element from the tree
void deleteNode(vector<int>&hT, int num) {
  int size = hT.size();
  int i;
  for (i = 0; i < size; i++) {
    if (num == hT[i])
      break;
  }
  swap(&hT[i], &hT[size - 1]);

  hT.pop_back();
  for (int i = size / 2 - 1; i >= 0; i--) {
    heapify(hT, i);
  }
}

// Print the tree
void printArray(vector<int>&hT) {
  for (int i = 0; i < hT.size(); ++i)
    cout << hT[i] << " ";
  cout << "\n";
}
// Driver code
int main() {
  vector<int> heapTree;

  insert(heapTree, 3);
  insert(heapTree, 4);
  insert(heapTree, 9);
  insert(heapTree, 5);
  insert(heapTree, 2);

  cout << "Max-Heap array: ";
  printArray(heapTree);
  deleteNode(heapTree, 4);
  cout << "After deleting an element: ";
  printArray(heapTree);
}
```

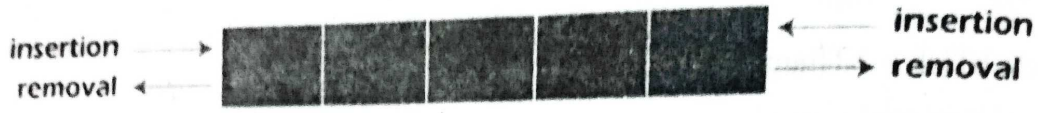
Q16. What is a Deque (or double-ended queue)? What are the various operations performed on deque? Explain.

(Imp

Ans :

The deque stands for Double Ended Queue. Deque is a linear data structure where the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue

Though the insertion and deletion in a deque can be performed on both ends, it does not follow the FIFO rule. The representation of a deque is given as follows -



Representation of deque

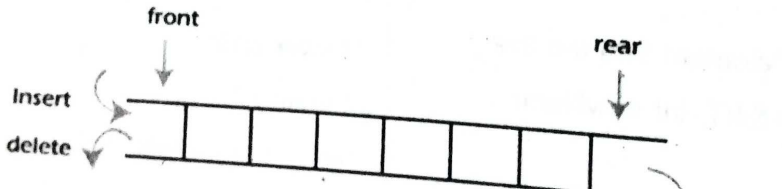
Types of Deque

There are two types of deque:

- Input restricted queue
- Output restricted queue

Input Restricted Queue

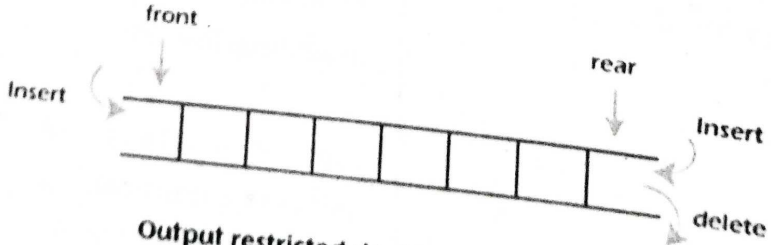
In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



input restricted double ended queue

Output restricted Queue

In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Output restricted double ended queue

Operations Performed on Deque

There are the following operations that can be applied on a deque -

- Insertion at front

- Insertion at rear
- Deletion at front
- Deletion at rear

We can also perform peek operations in the deque along with the operations listed above. Through peek operation, we can get the deque's front and rear elements of the deque. So, in addition to the above operations, following operations are also supported in deque

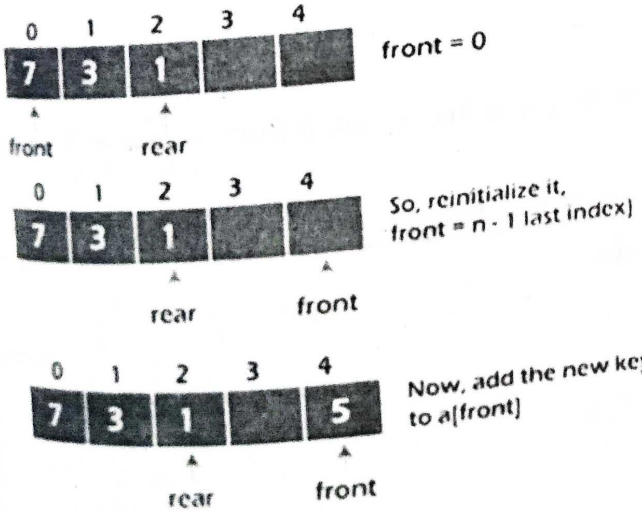
- Get the front item from the deque
- Get the rear item from the deque
- Check whether the deque is full or not
- Checks whether the deque is empty or not

Now, let's understand the operation performed on deque using an example.

Insertion at the Front End

In this operation, the element is inserted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is full or not. If the queue is not full, then the element can be inserted from the front end by using the below conditions:

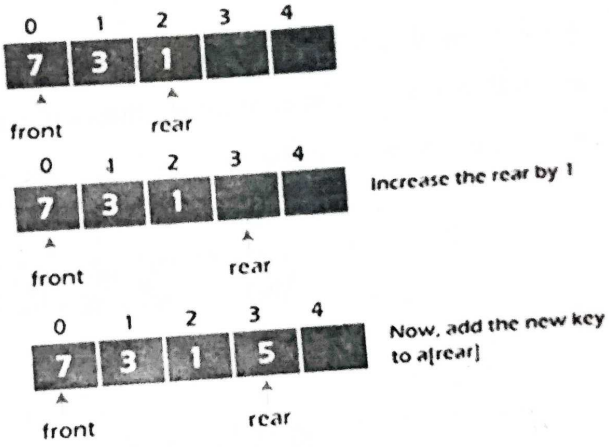
- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by **front = n - 1**, i.e., the last index of the array.



Insertion at the Rear End

In this operation, the element is inserted from the rear end of the queue. Before implementing the operation, we first have to check again whether the queue is full or not. If the queue is not full, then the element can be inserted from the rear end by using the below conditions.

- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, increment the rear by 1. If the rear is at last index (or size - 1), then instead of increasing it by 1, we have to make it equal to 0.

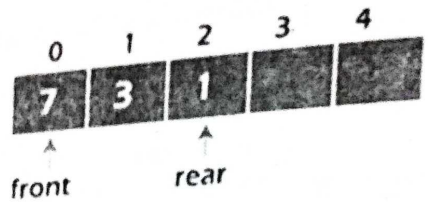


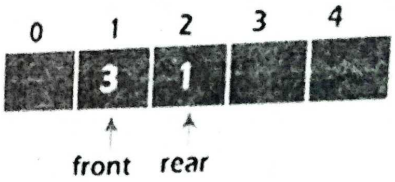
Deletion at the Front End

In this operation, the element is deleted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion. If the queue is not full, then the element can be inserted from the front end by using the below conditions:

- If the deque has only one element, set rear = -1 and front = -1.
- Else if front is at end (that means front = size - 1), set front = 0.
- Else increment the front by 1, (i.e., front = front + 1).





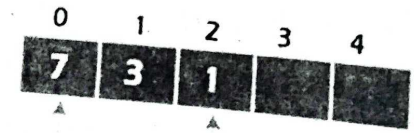
After deleting the element 7 front end

Deletion at the Rear end

In this operation, the element is deleted from the rear end of the queue. Before implementing operation, we first have to check whether the queue is empty or not.

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform deletion.

- If the deque has only one element, set rear = -1 and front = -1.
- If rear = 0 (rear is at front), then set rear = n - 1.
- Else, decrement the rear by 1 (or, rear = rear -1).



After deleting element 1 from rear end

Check empty

This operation is performed to check whether the deque is empty or not. If front = -1, it means the deque is empty.

Check full

This operation is performed to check whether the deque is full or not. If front = rear + 1, or front = 0 and rear = n - 1 it means that the deque is full.

The time complexity of all of the above operations of the deque is O(1), i.e., constant.

Q17. Write a C++ program to implement dequeue. **(Imp.)**

Ans :

Implementation of deque

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

```
#define MAX 10

class Deque {
  int arr[MAX];
  int front;
  int rear;
  int size;

  public:
  Deque(int size) {
    front = -1;
    rear = 0;
    this->size = size;
  }

  void insertfront(int key);
  void insertrear(int key);
  void deletefront();
  void deleterear();
  bool isFull();
  bool isEmpty();
  int getFront();
  int getRear();
};

bool Deque::isFull() {
  return ((front == 0 && rear == size - 1) ||
     front == rear + 1);
}

bool Deque::isEmpty() {
  return (front == -1);
}

void Deque::insertfront(int key) {
  if (isFull()) {
    cout << "Overflow\n"
<< endl;
    return;
  }

  if (front == -1) {
    front = 0;
    rear = 0;
  }

  else if (front == 0)
    front = size - 1;

  else
    front = front - 1;

  arr[front] = key;
}

void Deque ::insertrear(int key) {
  if (isFull()) {
    cout << " Overflow\n " << endl;
    return;
  }

  if (front == -1) {
    front = 0;
    rear = 0;
  }

  else if (rear == size - 1)
    rear = 0;

  else
    rear = rear + 1;

  arr[rear] = key;
}

void Deque ::deletefront() {
  if (isEmpty()) {
    cout << "Queue Underflow\n"
```

```
<< endl;
    return;
  }

  if (front == rear) {
   front = -1;
   rear = -1;
  } else if (front == size - 1)
   front = 0;

  else
   front = front + 1;
 }

 void Deque::deleterear() {
  if (isEmpty()) {
   cout << " Underflow\n"
<< endl;
   return;
  }

  if (front == rear) {
   front = -1;
   rear = -1;
  } else if (rear == 0)
   rear = size - 1;
  else
   rear = rear - 1;
 }

 int Deque::getFront() {
  if (isEmpty()) {
   cout << " Underflow\n"
<< endl;
   return -1;
  }
  return arr[front];
}

int Deque::getRear() {
 if (isEmpty() || rear < 0) {
  cout << " Underflow\n"
<< endl;
  return -1;
 }
 return arr[rear];
}

int main() {
 Deque dq(4);

 cout << "insert element at rear end \n";
 dq.insertrear(5);
 dq.insertrear(11);

 cout << "rear element: "
<< dq.getRear() << endl;

 dq.deleterear();
 cout << "after deletion of the rear element. the
new rear element: " << dq.getRear() << endl;

 cout << "insert element at front end \n";

 dq.insertfront(8);

 cout << "front element: " << dq.getFront() <<
endl;

 dq.deletefront();

 cout << "after deletion of front element new front
element: " << dq.getFront() << endl;
}
```

Short Question and Answers

1. What is stack?

Ans :

A stack is an ordered collection of homogeneous data elements where the insertion and deletion operations take place at one end only. The insertion and deletion operations in the case of a stack are specially termed PUSH and POP, respectively, and the position of the stack where these operations are performed is known as the TOP of the stack. An element in a stack is termed an ITEM. The maximum number of elements that a stack can accommodate is termed SIZE.

2. What are the various ways to represent the stack?

Ans :

A stack may be represented in the memory in various ways. There are two main ways: using a one-dimensional array and a single linked list.

Array Representation of Stacks

First we have to allocate a memory block of sufficient size to accommodate the full capacity of the stack. Then, starting from the first location of the memory block, the items of the stack can be stored in a sequential fashion.

In Figure, Itemi denotes the *ith* item in the stack; l and u denote the index range of the array in use; usually the values of these indices are 1 and SIZE respectively. TOP is a pointer to point the position of the array up to which it is filled with the items of the stack.

3. Define queue.

Ans :

A queue is an ordered collection of homogeneous data elements; in contrast with the stack, here, insertion and deletion operations take place at two extreme ends. A queue is also a linear data structure like an array, a stack and a linked list where the ordering of elements is in a-linear fashion. The only difference between a stack and a queue is that in the case of stack insertion and deletion (PUSH and POP) operations are at one end (TOP) only, but in a queue insertion (called ENQUEUE) and deletion (called DEQUEUE) operations take place at two ends called the REAR and FRONT of the queue, respectively. Figure represents a model of a queue structure. Queue is also termed first-in first-out (FIFO).

4. Queue as an ADT

Ans :

The meaning of an abstract data type clearly says that for a data structure to be abstract, it should have the below-mentioned characteristics:

- First, there should be a particular way in which components are related to each other
- Second, a statement for the operation that can be performed on elements of abstract data type must have to be specified.

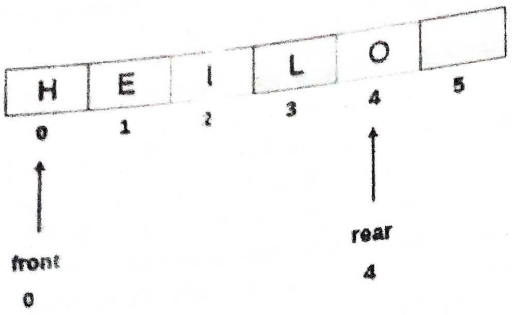
Thus for defining a Queue as an abstract data type, these are the following criteria:

- Initialize a queue to be empty
- Check whether a queue is empty or not
- Check whether a queue is full or not
- Insert a new element after the last element in a queue, if the queue is not full
- Retrieve the first element of the queue, if it is not empty
- Delete the first element in a queue, if it is not empty

5. Array Representation of Queue

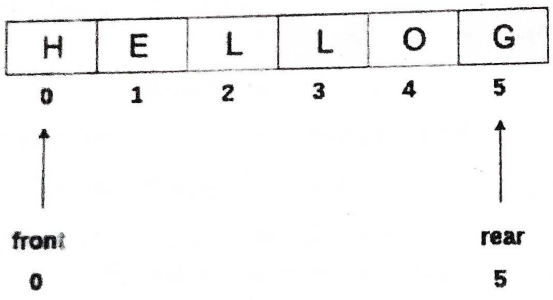
Ans :

There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



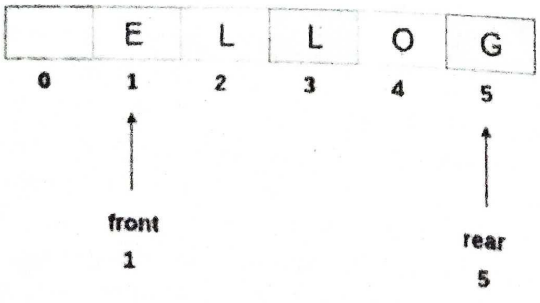
Queue

The above figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

6. Operations on Stacks

Ans :

The most basic stack operations in the structure are the following.

- **push():** When we insert an element in a then the operation is known as a push stack is full then the overflow condition o
- **pop():** When we delete an element fro stack, the operation is known as a pop. stack is empty means that no element in the stack, this state is known a underflow state.
- **isEmpty():** It determines whether the is empty or not.
- **isFull():** It determines whether the sta full or not.'
- **peek():** It returns the element at the position.
- **count():** It returns the total numbe elements available in a stack.
- **change():** It changes the element at the position.
- **display():** It prints all the elements avail in the stack.

7. What are the operations that can performed on the queue?

Ans :

The queue data structure includes following operations:

- **EnQueue:** Adds an item to the que Addition of an item to the queue is al done at the rear of the queue.
- **DeQueue:** Removes an item from queue. An item is removed or de-que always from the front of the queue.
- **isEmpty:** Checks if the queue is empty
- **isFull:** Checks if the queue is full.
- **peek:** Gets an element at the front of queue without removing it.

Enqueue

In this process

- Check if the q
- If full, produc
- Else, increme
- Add an elem
- Return succe

8. Simple Qu

Ans :

In Linear Q end. The end at deletion takes pla

Dequeue
FRONT end
(Deletion)

The majo first three eleme is available in a pointing to the

9. Circular

Ans :

In Circul that the last el the ends are c

The d space is ava incrementing

Enqueue

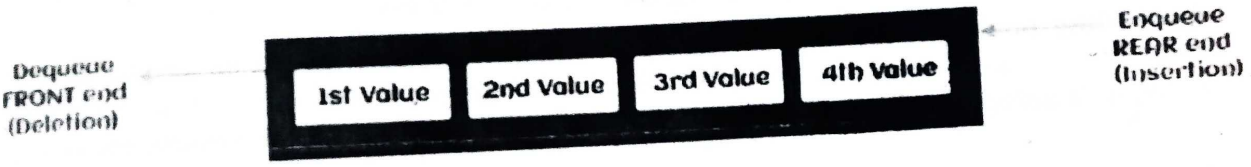
In this process, the following steps are performed:

- 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.

8. Simple Queue

Ans :

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.

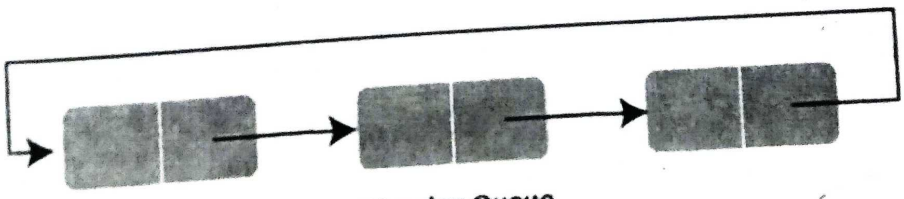


The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

9. Circular Queue

Ans :

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. It is also known as Ring Buffer, as all the ends are connected to another end. The representation of circular queue is shown in the below image



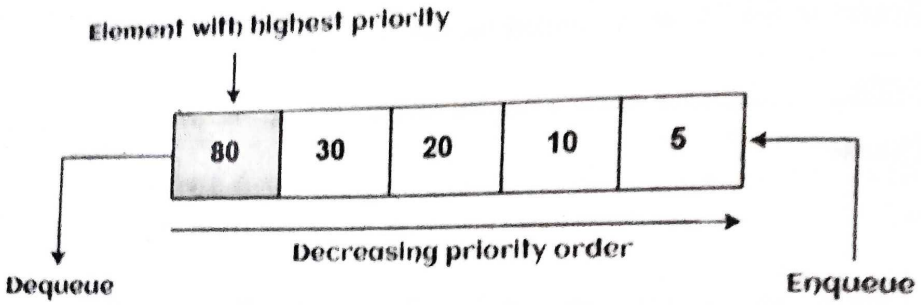
Circular Queue

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.

10. Priority Queue

Ans :

It is a special type of queue in which the elements are arranged based on the priority. It is a type of queue data structure in which every element has a priority associated with it. Suppose elements occur with the same priority, they will be arranged according to the FIFO principle representation of priority queue is shown in the below image.



Insertion in priority queue takes place based on the arrival, while deletion in the priority queue occurs based on the priority. Priority queue is mainly used to implement the CPU scheduling algorithms.

There are two types of priority queue that are discussed as follows:

- **Ascending priority queue:** In ascending priority queue, elements can be inserted in arbitrary order, but only smallest can be deleted first. Suppose an array with elements 7, 5, and 3 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 3, 5, 7.
- **Descending priority queue:** In descending priority queue, elements can be inserted in arbitrary order, but only the largest element can be deleted first. Suppose an array with elements 7, 3, and 5 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 7, 5, 3.

1. A queu
(a) L
(c) L
2. The ti
(a)
(c)
3. Whic
(a)
(c)
4. Whic
(a)
(c)
5. We
(a)
(c)
6. Wh
(a)
(c)
7. Wh
(a)
(c)
8. If
(a
(c
9.

Choose the Correct Answers

1. A queue follows _____. [b]
   (a) LIFO principle (b) FIFO principle
   (c) Linear tree (d) Ordered array

2. The time complexity used for inserting a node in a priority queue on the basis of key is: [a]
   (a) O(n) (b) O(n2)
   (c) O(nlogn) (d) O(logn)

3. Which of these is a postfix expression? [c]
   (a) a+b-c (b) +ab
   (c) abc*+de-+ (d) a*b(c+d)

4. Which data structure do we use for testing a palindrome? [d]
   (a) Heap (b) Tree
   (c) Priority queue (d) Stack

5. We can use a self–balancing binary search tree for implementing the: [b]
   (a) Hash table (b) Priority queue
   (c) Heap sort and Priority queue (d) Heap sort

6. Which of the following is the infix expression? [a]
   (a) A+B*C (b) +A*BC
   (c) ABC+* (d) None of the above

7. What is another name for the circular queue among the following options? [c]
   (a) Square buffer (b) Rectangle buffer
   (c) Ring buffer (d) None of the above

8. If the elements '1', '2', '3' and '4' are inserted in a queue, what would be order for the removal? [a]
   (a) 1234 (b) 4321
   (c) 3241 (d) None of the above

9. Which one of the following is the overflow condition if linear queue is implemented using an array with a size MAX_SIZE? [c]
   (a) rear = front (b) rear = front+1
   (c) rear=MAX_SIZE -1 (d) rear = MAX_SIZE

10. How many Queues are required to implement a Stack? [b]
    (a) 3 (b) 2
    (c) 1 (d) 4

Fill in the Blanks

1. _____ is a data structure in which insertion takes place from one end, and deletion takes place from one end.
2. When the user tries to delete the element from the empty stack then the condition is said to be _____.
3. If the size of the stack is 10 and we try to add the 11th element in the stack then the condition is known as _____.
4. The time complexity of enqueue operation in Queue is _____.
5. The necessary condition to be checked before deletion from the Queue is _____.
6. A linear data structure in which insertion and deletion operations can be performed from both the ends is _____.
7. The _____ data structure required to check whether an expression contains a balanced parenthesis is?
8. Insertion and Deletion operation in Queue is known as _____
9. _____ node is considered the top of the stack if the stack is implemented using the linked list?
10. The minimum number of stacks required to implement a stack is _____

Answers

1. Queue
2. under flow
3. overflow
4. O(1)
5. under flow
6. Deque
7. Stack
8. Enqueue and Dequeue
9. First node
10. 2