UNIT- IV

STACKS & QUEUES

Contents

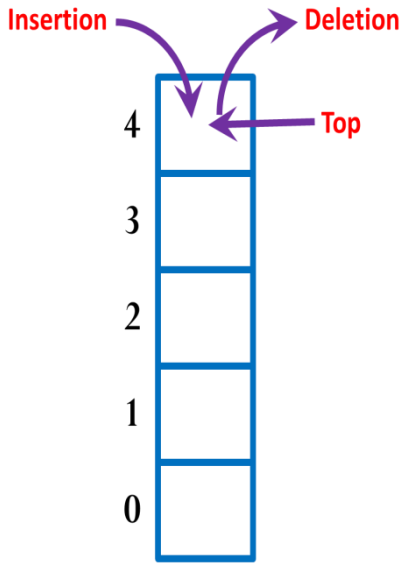
Stacks- concept, Primitive operations, Stack Abstract Data Type, Representation of Stacks Using Sequential Organization, stack operations, Multiple Stacks, Applications of Stack- Expression Evaluation and Conversion, Polish notation and expression conversion, Need for prefix and postfix expressions, Postfix expression evaluation, Linked Stack and Operations. Recursion concept, variants of recursion- direct, indirect, tail and tree, Backtracking algorithmic strategy, use of stack in

backtracking.

Queues - Concept, Queue as Abstract Data Type, Realization of Queues Using Arrays, Circular Queue, Advantages of using circular queues, Multi-queues, Deque, Priority Queue, Array implementation of priority queue, Linked Queue and operations, Case study- Priority queue in Bandwidth management, 4 Queens problem, Android multiple tasks/multiple activities and back stack.

What is a STACK ?

- A stack is a Linear list
- It is a container of elements that are inserted and removed according to the last-in first-out (LIFO) principle.
- A stack is a ordered list of elements of same data type
- In a stack all operation like insertion and deletion are performed at only one end called Top

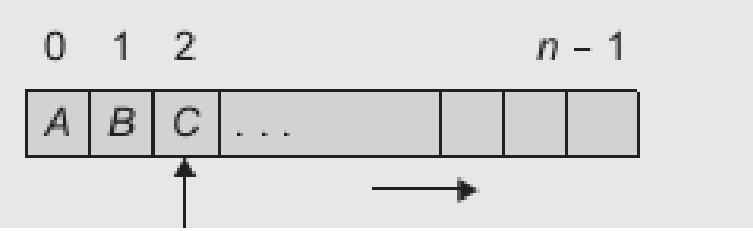


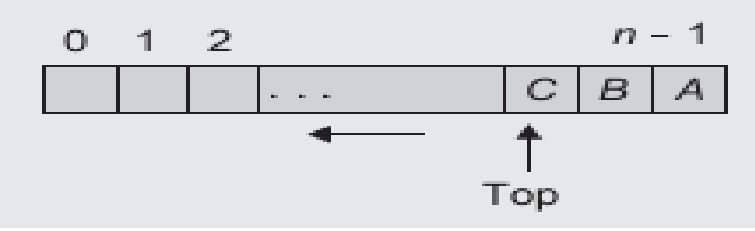
Representation of stacks using sequential organization (arrays)

- A stack can be implemented using both a static data structure (array) and a dynamic data structure (linked list).
- The simplest way to represent a stack is by using a one-dimensional array.
- A stack is an ordered collection of elements. Hence, it would be very simple to manage a stack when represented using an array.
- The only difficulty with an array is its static memory allocation. Once declared, the size cannot be modified during run-time.

Representation of stacks using sequential organization (arrays)

- The simplest way to implement an ADT stack is using arrays.
- We initialize the variable top to -1 using a constructor to denote an empty stack.
- The bottom element is represented using the 0^{th} position, that is, the first element of the array.
- The next element is stored at the 1^{st} position and so on.





Representation of stacks using sequential organization (arrays)

```
class Stack
{
private:
int Stack[50];
int MaxCapacity;
int top;
public:
Stack()
{
MaxCapacity = 50;
top = −1;
currentsize = 0;
}
```

Operations on STACK ?

- Creation
- Insertion
- Deletion
- Displaying

Operations on STACK ?

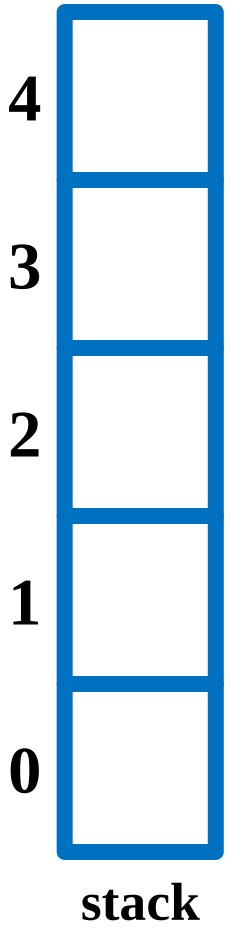
Creation

Insertion

Deletion

Displaying

```
#define SIZE 5
int stack[SIZE];
```



Operations on STACK ?

Insertion operation is called as "push"

Creation

Insertion

Deletion

Displaying

```
void push(element){
    if(Stack is full)
    {
        cout<<"Full"
    }
    else
    {
        Top++;
        stack[Top] = element;
    }
}
```

Operations on STACK ?

Creation

Insertion

Deletion

Displaying

Deletion operation is called as "pop"

```
int pop( ){
    if(Stack is Empty)
    {
    Cout<<"Empty";
        return  Top;
    }
    else
    {
        deleted = stack[Top];
        Top--;
        return  deleted;
    }
}
```

Operations on STACK ?

Creation

Insertion

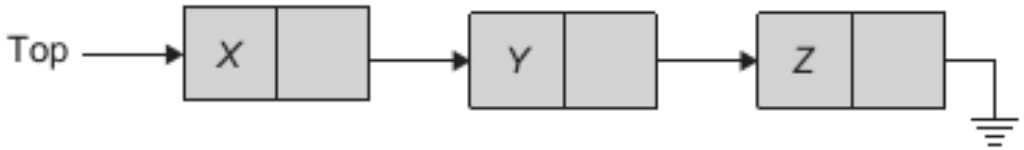
Deletion

Displaying

```
void display( ){
    if(Stack is Empty)
    {
        cout<<“Empty”
    }
    else
    {
        for(i=Top; i>-1; i--
    )
         cout<<stack[i];
    }
}
```

Stack Operations Using Link List

- A stack implemented using a linked list is also called linked stack.
- Each element of the stack will be represented as a node of the list.
- The addition and deletion of a node will be only at one end.
- The first node is considered to be at the top of the stack, and it will be pointed to by a pointer called top.
- The last node is the bottom of the stack, and its link field is set to Null.
- An empty stack will have Top = Null.



Stack Operations Using Link List

Structure of stack node:

```
class Stack_Node
{
   public:
        int data;
        Stack_Node *link;
};
```

Stack Operations Using Link List

IsEmpty(): to check stack is empty

or nor

```
int Stack :: IsEmpty()
{
if(Top == Null)
        return 1;
else
        return 0;
}
```

GetTop(): To get top element

```
int Stack :: GetTop()
{
if(!IsEmpty())
        return(Top->data);
}
```

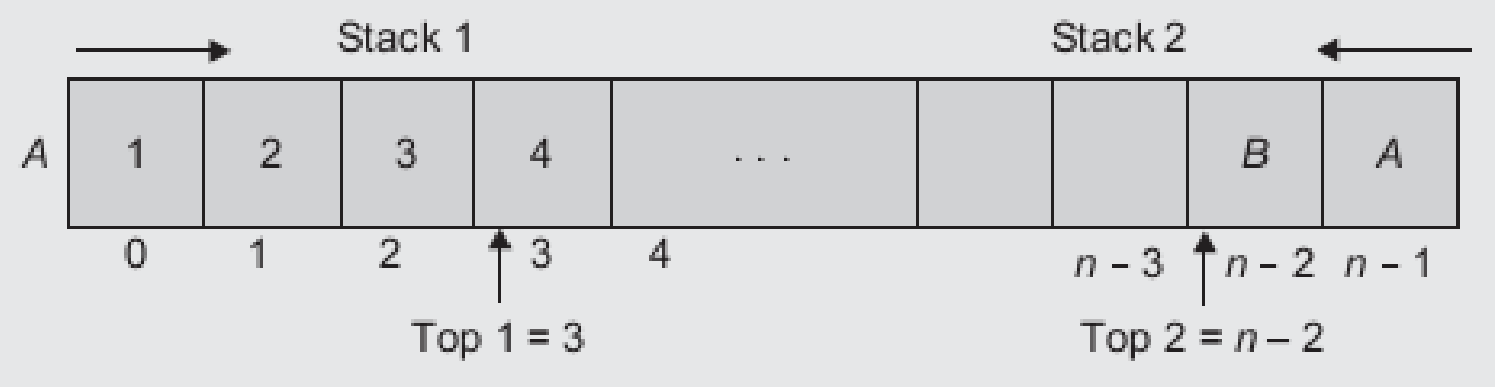
Stack Operations Using Link List

```
void Stack :: Push(int value)
{
Stack_Node* NewNode;
NewNode = new Stack_Node;
NewNode->data = value;
NewNode->link = Null;
NewNode->link = Top;
Top = NewNode;
}
```

```
int Stack :: Pop()
{
Stack_Node* tmp = Top;
int data = Top->data;
if(!IsEmpty())
{
Top = Top->link;
delete tmp;
return(data);
}
}
```

Multiple Stacks

- Multiple stacks can be implemented by sequentially mapping stacks into A[0], ..., A[n − 1].
- The solution is simple if we implement only two stacks. The first stack grows towards A[n -1]from A[0]and the second stack grows towards A[0] from A[n − 1].



Applications of Stack

The stack data structure is used in a wide range of applications. A few of them are the following:

1. Converting infix expression to postfix and prefix expressions
2. Evaluating the postfix expression
3. Checking well-formed (nested) parenthesis
4. Reversing a string
5. Processing function calls
6. Parsing (analyse the structure) of computer programs
7. Simulating recursion
8. In computations such as decimal to binary conversion
9. In backtracking algorithms (often used in optimizations and in games)

Expression evaluation and Conversion

- The most frequent application of stacks is in the evaluation of arithmetic expressions.
- An arithmetic expression is made of operands, operators, and delimiters.
- Consider the following expression:

$$X = a/b * c - d$$

Let a =1, b =2, c =3, and d =4.

- One of the meanings that can be drawn from this expression could be

$$X =(1/2) * (3 -4) = -1/2$$

- Another way to evaluate the same expression could be

$$X =(1/(2 *3)) -4 = -23/6$$

Expression evaluation and Conversion

- To fix the order of evaluation, assign each operator a priority.
- The following operators are written in descending order of their precedence:

1. Exponentiation (^), Unary (+), Unary (-), and not (~)
2. Multiplication (*)and division (/)
3. Addition (+)and subtraction (-)
4. Relational operators <, *, =, π, ≥, >
5. Logical AND
6. Logical OR

Expression evaluation and Conversion

- Expressions such as A + B – C and A x B/C are to be evaluated from left to right.
- However, the expression A ^ B ^ C is to be evaluated from right to left.
- Exponentiation is right associative and all other operators are left associative.

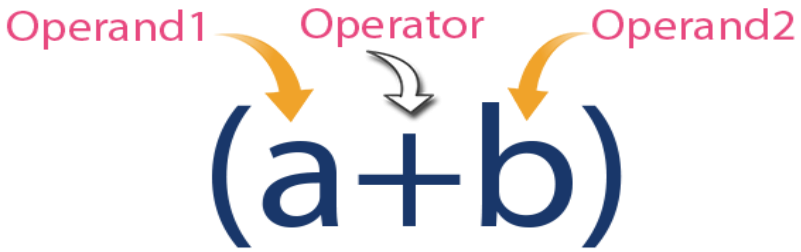
Expression Types

- Based on the operator position, expressions are divided into THREE types. They are as follows...
- Infix Expression--- A+B
- Postfix Expression--- AB+
- Prefix Expression--- +AB

Infix Expression

- In infix expression, operator is used in between operands.

  The general structure of an Infix expression is as follows...
- Operand1 Operator Operand2
- Example



Postfix Expression

- In postfix expression, operator is used after operands. We can say that "Operator follows the Operands".

  The general structure of Postfix expression is as follows...
- Operand1 Operand2 Operator
- Example

Prefix Expression

- In prefix expression, operator is used before operands. We can say that "Operands follows the Operator".

  The general structure of Prefix expression is as follows...
- Operator Operand1 Operand2
- Example

Expression Conversion

- Any expression can be represented using three types of expressions (Infix, Postfix and Prefix).
- We can also convert one type of expression to another type of expression like

1. Infix expression to postfix expression
2. Infix expression to prefix expression
3. Prefix expression to infix expression
4. Prefix expression to postfix expression
5. Postfix expression to infix expression
6. Postfix expression to prefix expression

Infix to Postfix Conversion

1. Scan expression E from left to right, character by character, till character is '#'

```
ch = get_next_token(E)
2. while(ch != '#')
        if(ch = ')') then ch = pop()
                while(ch !='(')
                        Display ch
                        ch = pop()
                end while
        if(ch = operand) display the same
        if(ch = operator) then
                if(ICP > ISP) then push(ch)
                else
                        while(ICP <= ISP)
                                pop the operator and display it
                        end while
                ch = get_next_token(E)
end while
3. if(ch = #) then while(!emptystack()) pop and display
4. stop
```

Infix to Postfix Conversion

1. Read all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is operand, then directly print it to the result (Output).
3. If the reading symbol is left parenthesis '(', then Push it on to the Stack.
4. If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is poped and print each poped symbol to the result.
5. If the reading symbol is operator (+ , - , * , / etc.,), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.

Convert following expression to postfix

A ^B * C - C + D/A/(E + F)

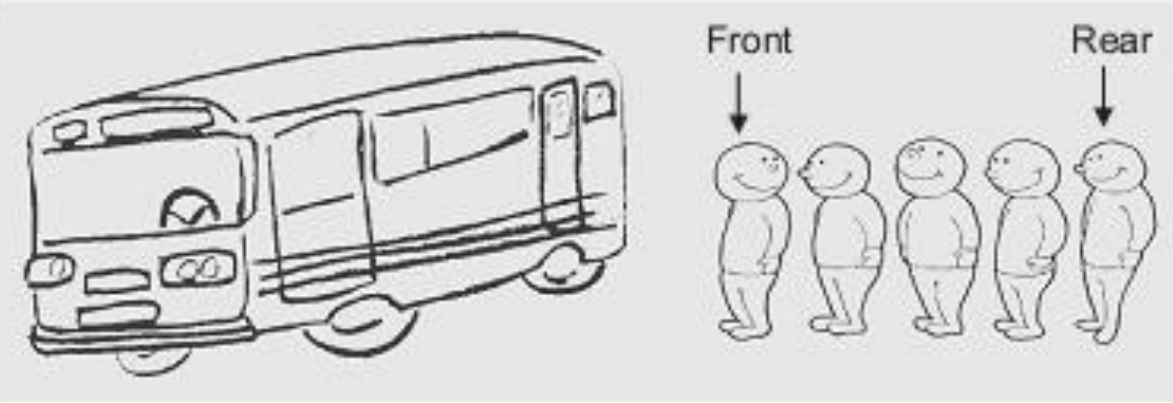
Character scanned	Stack contents	Postfix expression	Remark
A	Empty	A	
^	^	A	
B	^	AB	
*	*	AB^	^ high priority so perform pop operation
C	*	AB^C	
-	-	AB^C*	* high priority so perform pop operation
C	-	AB^C*C	
+	+	AB^C*C-	
D	+	AB^C*C-D	
/	+/	AB^C*C-D	
A	+/	AB^C*C-DA	
/	+/	AB^C*C-DA/	
(	+/(	AB^C*C-DA/	
E	+/(	AB^C*C-DA/E	
+	+/(+	AB^C*C-DA/E	
F	+/(+	AB^C*C-DA/EF	
)	+/	AB^C*C-DA/EF+	
	Empty	AB^C*C-DA/EF+/+	

Queue

- In arrays, element insertion at and deletion from any position causes a lot of data movement.
- On the other hand, in stacks, these operations are performed at only one end, the top.
- A queue is a special type of data structure that performs insertions at one end called the **rear** and deletions at the other end called the **front**.
- a queue follows first in first out (**FIFO**) or last in last out (**LILO**) structure.

queue

- Consider an ordered list L ={a_1, a_2, a_3, a_4, …, a_n}.
- If we assume that L represents a queue, then a_1 is the front-end element and an is the rear-end element. In addition, a_i is behind a_{i-1}.



Queue operation

A set of operations on a queue is as follows:

1. create()—creates an empty queue, Q

2. add(i,Q)—adds the element i to the rear end of the queue, Q and returns the new queue

3. delete(Q)—takes out an element from the front end of the queue and returns the resulting queue

4. getFront(Q)—returns the element that is at the front position of the queue

5. Is_Empty(Q)—returns true if the queue is empty; otherwise returns false

Realization of Queues using Arrays

- Create(): This operation should create an empty queue. Here max is the maximum initial size that is defined.
- #define max 50
- int Queue[max];
- int Front = Rear = −1;

Realization of Queues using Arrays

- Is_Empty(): This operation checks whether the queue is empty or not. This is confirmed by comparing the values of Front and Rear. If Front = Rear, then Is_Empty returns true, else returns false.

```
bool Is_Empty()
{
   if(Front == Rear)
         return 1;
   else
         return 0;
}
```

Realization of Queues using Arrays

- Is_Full : When Rear points to the last location of the array, it indicates that the queue is full, that is, there is no space to accommodate any more elements.

```
bool Is_Full()
{
   if(rear == max − 1)
         return 1;
   else
         return 0;
}
```

Realization of Queues using Arrays

- Add : This operation adds an element in the queue if it is not full.

  As Rear points to the last element of the queue, the new element is added at the (rear +1)th location.

```
void Add(int Element)
{
   if(Is_Full())
         cout << "Error, Queue is full";
   else
         Queue[++Rear] = Element;
}
```

Realization of Queues using Arrays

- Delete : This operation deletes an element from the front of the queue and sets Front to point to the next element. Front can be initialized to one position less than the actual front. We should first increment the value of Front and then remove the element.

```
int Delete()
{
    if(Is_Empty())
        cout << "Sorry, queue is Empty";
    else
        return(Queue[++Front]);
}
```

Realization of Queues using Arrays

- getFront : The operation getFront returns the element at the front, but unlike delete, this does not update the value of Front.

```
int getFront()
{
   if(Is_Empty())
        cout << "Sorry, queue is Empty";
   else
        return(Queue[Front + 1]);
}
```

1

Front=-1 Queue is empty

0 1 2 3 4

Rear=-1

2

Front=0 Insert A

0 1 2 3 4

Rear=0

3

Front=0 Insert B

0 1 2 3 4

Rear=1

4

Front=0 Insert C

0 1 2 3 4

Rear=2

5

Front=1 Delete

0 1 2 3 4

Rear=2

6

Front=2 Delete

0 1 2 3 4

Rear=2

7

Front=3 Delete

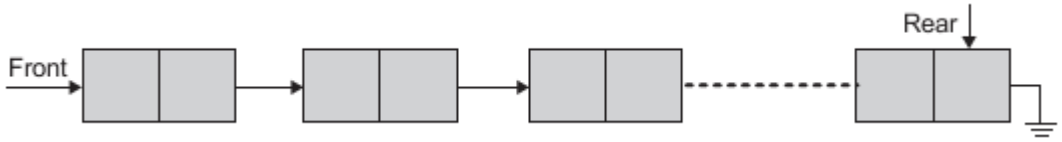
0 1 2 3 4

Rear=2 Queue is empty

Entry point is called Rear &
Exit point is called Front

Linked Queue and operations

- There are many drawbacks of implementing queues using arrays.
- The fixed sizes do not give flexibility to the user to dynamically exceed the maximum size.
- The declaration of arbitrarily maximum size leads to poor utilization of memory.
- In addition, the major drawback is the updating of front and rear.
- Here is a good solution to this problem which uses linked list.
- We need two pointers, front and rear.



Linked Queue and operations

Let us consider the following node structure for studying the linked queue and operations:

```
class QNode
{
public:
int data;
QNode *link;
};
```

Linked Queue and operations

IsEmpty(): to check queue is empty or nor

```
int Queue :: IsEmpty()
{
if(Front == Null)
return 1;
else
return 0;
}
```

```
int Queue :: GetFront()
{
if(!IsEmpty())
return(Front->data);
}
```

Linked Queue and operations


```
void Queue :: Add(int x)
{
QNode *NewNode;
NewNode = new QNode;
NewNode->data = x;
NewNode->link = Null;
// if the new is a node getting added in empty queue then front should be set so as to point to new
if(Rear == Null)
{
    Front = NewNode;
    Rear = NewNode;
}
else
{
    Rear->link = NewNode;
    Rear = NewNode;
}
}
```

Linked Queue and operations


```
int Queue :: Delete()
{
int temp;
QNode *current = Null;
if(!IsEmpty())
{
    temp = Front->data;
    current = Front;
    Front = Front->link;
    delete current;
    if(Front == Null)
            Rear = Null;
    return(temp);
}
}
```

Disadvantages of linear queue

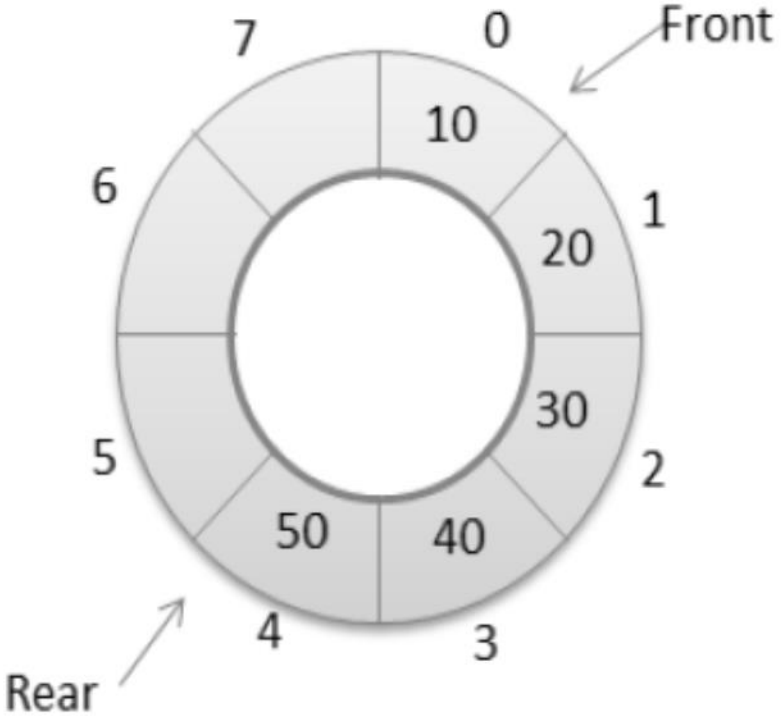
- On deletion of an element from existing queue, front pointer is shifted to next position.
- This results into virtual deletion of an element.
- By doing so memory space which was occupied by deleted element is wasted and hence inefficient memory utilization is occur.

Overcome disadvantage of linear queue:

- To overcome disadvantage of linear queue, **circular queue** is use.
- We can solve this problem by joining the front and rear end of a queue to make the queue as a circular queue .
- Circular queue is a linear data structure. It follows FIFO principle.
- In circular queue the last node is connected back to the first node to make a circle.

Overcome disadvantage of linear queue:

- It is also called as “Ring buffer”.
- Items can inserted and deleted from a queue in O(1) time.



Types Of Queue

1. Circular Queue

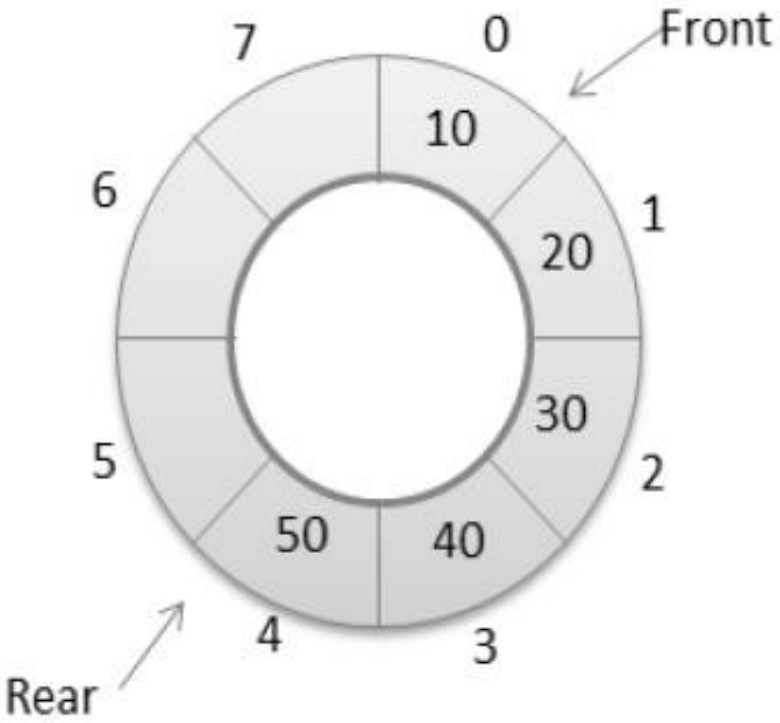
2. Dequeue (Double Ended Queue)

3. Priority Queue

Circular queue

- A queue, in which the last node is connected back to the first node to form a cycle, is called as circular queue.
- Circular queue are the queues implemented in circular form rather than in a straight line.
- Circular queues overcome the problem of unutilized space in linear queue implemented as an array.
- The main disadvantage of linear queue using array is that when elements are deleted from the queue, new elements cannot be added in their place in the queue, i.e. the position cannot be reused.

Circular queue



Circular queue implementation

- After rear reaches the last position, i.e. MAX-1 in order to reuse the vacant positions, we can bring rear back to the 0th position, if it is empty, and continue incrementing rear in same manner as earlier.
- Thus rear will have to be incremented circularly.
- For deletion, front will also have to be incremented circularly..

Dequeue (Delete) operation on Circular Queue:

- **Step 1:** Check for queue empty **if (front = -1)**

  then circular queue is empty and deletion operation is not possible. otherwise go to step 2

- **Step 2:** Check for position of front and rear pointers.

  if front = rear then

  Data = q[front];

  set front=rear=-1

- **Step 3:** **C**heck position of front

  if front = Max-1

  Data = q[front];

  then set front=0;

  otherwise

  Data = q[front];

  front = (front+1)%MAX

PRIORITY QUEUE

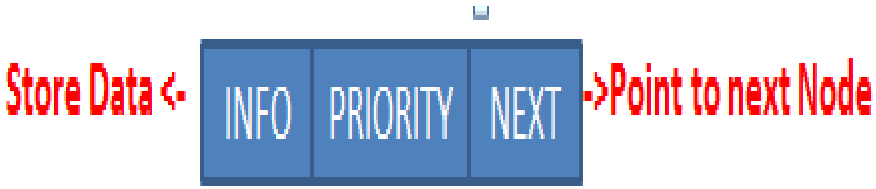
- A priority Queue is a collection of elements where each element is assigned a priority and the order in which elements are deleted and processed is determined from the following rules:
- 1) An element of higher priority is processed before any element of lower priority.
- 2) Two elements with the same priority are processed according to the order in which they are added to the queue.

The priority queue implementation

- The priority queue is again implemented in two way

1. array/sequential representation.
2. Dynamic/ linked representation.

In priority queue node is divided into three parts



Structure of priority queue Node

PRIORITY QUEUE

- An example where priority queue are used is in operating systems.
- The operating system has to handle a large number of jobs.
- These jobs have to be properly scheduled.
- The operating system assigns priorities to each type of job.
- The jobs are placed in a queue and the job with the highest priority will be executed first.

PRIORITY QUEUE

- **Advantages:-**
  - Preferences to the higher priority process are added at the beginning.
  - Keep the list sorted in *increasing* order.

Applications of Queue

- Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for there turn, like in the following scenarios :
  - Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
  - In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
  - Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

Enqueue(Insert) operation on Circular Queue:

- **Step 1:** Check for queue full
- If rear=max–1 and front=0 or if front=rear+1 then circular queue is full and insertion operation is not possible. otherwise go to step 2
- **Step 2:** Check position of rear pointer

  If rear=max–1

  then set rear=0 otherwise increment rear by 1.

  rear=(rear+1)%MAX
- **Step 3:** Insert element at the position pointer by rear pointer.

  q[rear]=Data
- **Step 4:** Check the position of front pointer

  If front=–1 then set front as 0.

Distinguish between stack and queue

Sr.No	STACK	QUEUE
1	It is LIFO(Last In First Out) data structure	It is FIFO (First In First Out) data structure.
2	Insertion and deletion take place at only one end called top	Insertion takes place at rear and deletion takes place at front.
3	It has only one pointer variable	It has two pointer variables.
4	No memory wastage	Memory wastage in linear queue
5	Operations: 1.push() 2.pop()	Operations: 1.enqueue() 2.dequeue()
6	In computer system it is used in procedure calls	In computer system it is used time/resource sharing
7.	Plate counter at marriage reception is an example of stack	Student standing in a line at fee counter is an example of queue.

Thank You