UNIT-II

Stacks and Queues

Introduction and primitive operations on stack; Stack application; Infix, postfix, prefix expressions;

Evaluation of postfix expression; Conversion between prefix, infix and postfix, introduction and

primitive operation on queues, D- queues and priority queues.

Unit 2

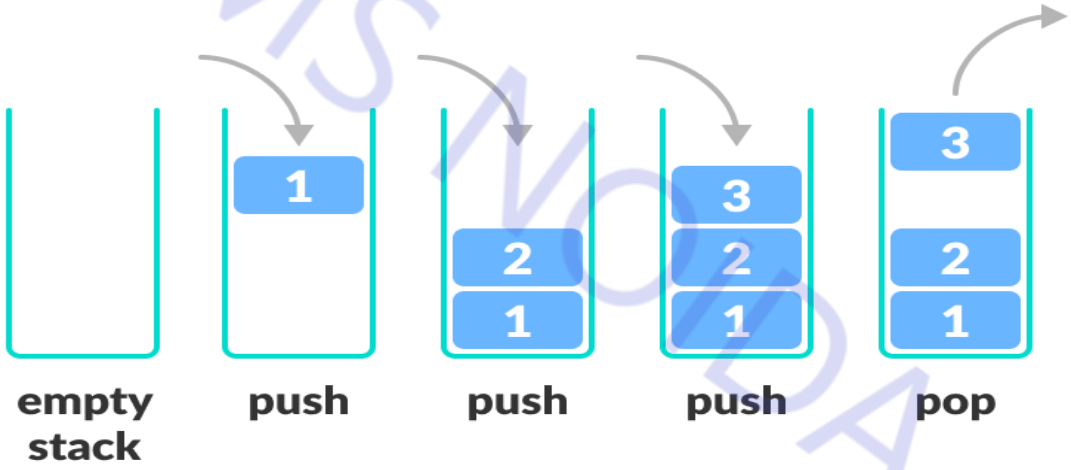
Stack

Stack is a linear data structure that follow the Last In First Out (LIFO) principle. The last item to be inserted into a stack is the first one to be deleted from it.

For example, you have a stack of trays on a table. The tray at the top of the stack is the first item to be moved if you require a tray from that stack.

Operations on stack

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.



Algorithm to push an element into stack.

PUSH_STACK (STACK, TOP, MAX, ITEM)

1) IF TOP = MAX then

   Print “Overflow”;

   Exit;

2) Otherwise TOP: = TOP + 1; /*increment TOP*/

   STACK [TOP]:= ITEM;

3) End of IF

4) Exit

Algorithm to pop an element from stack

POP_STACK(STACK,TOP,ITEM)

1) IF TOP = 0 then

   Print “Underflow”;

   Exit;

2) Otherwise ITEM: =STACK [TOP];

   TOP:=TOP – 1;

3) End of IF

4) Exit

Applications of Stack

- Infix to Postfix or Infix to Prefix Conversion.
- Postfix or Prefix Evaluation
- backtracking
- Function Call
- Memory Management

Infix and Postfix Expressions

Infix Expression : We write expression in **infix** notation, e.g. a - b + c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

form a op b.

<operand><operator><operand>

Ex: a+b

Postfix Expression: This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

form a b op.

<operand><operand><operator>

Ex: ab+

Prefix Expression: In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

form op a b

<operator><operand><operand>

Ex: +ab

Algorithm to convert Infix To Postfix

Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

1) Push “(“onto Stack, and add “)” to the end of X.
2) Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
3) If an operand is encountered, add it to Y.
4) If a left parenthesis is encountered, push it onto Stack.
5) If an operator is encountered ,then:
    1) Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.
    2) Add operator to Stack.
    [End of If]
6) If a right parenthesis is encountered ,then:
    1) Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.
    2) Remove the left Parenthesis.
    [End of If]
    [End of If]
7) END.

Example

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

Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack, that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack, that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Advantage of Postfix Expression over Infix Expression

An infix expression is difficult for the machine to know and keep track of precedence of operators. On the other hand, a postfix expression itself determines the precedence of operators (as the placement of operators in a postfix expression depends upon its precedence).Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

Evaluation of Postfix Expressions:

Algorithm

1) Add `)` to postfix expression.
2) Read postfix expression Left to Right until `)` encountered
3) If operand is encountered, push it onto Stack
[End If]
4) If operator is encountered, Pop two elements
i) `A -> Top` element
ii) `B-> Next to Top` element
iii) Evaluate `B` operator `A`
push `B` operator `A` onto Stack
5) Set `result = pop`
6) END

Let's see an example to better understand the algorithm:

Expression: 456*+

Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

UNIT 2

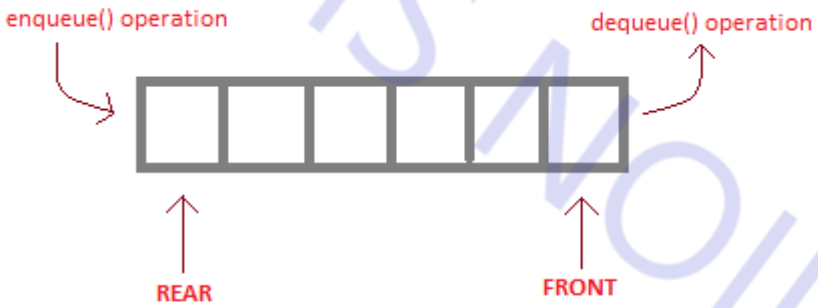
Queue:

Queue is also an abstract data type or a linear data structure, just like stack data structure, in which the first element is inserted from one end called the **REAR**(also called **tail**), and the removal of existing element takes place from the other end called as **FRONT**(also called **head**).

This makes queue as **FIFO**(First in First Out) data structure, which means that element inserted first will be removed first.

Which is exactly how queue system works in real world. If you go to a ticket counter to buy movie tickets, and are first in the queue, then you will be the first one to get the tickets. Right? Same is the case with Queue data structure. Data inserted first, will leave the queue first.

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



enqueue( ) is the operation for adding an element into Queue.

dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

Applications of Queue

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.

3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

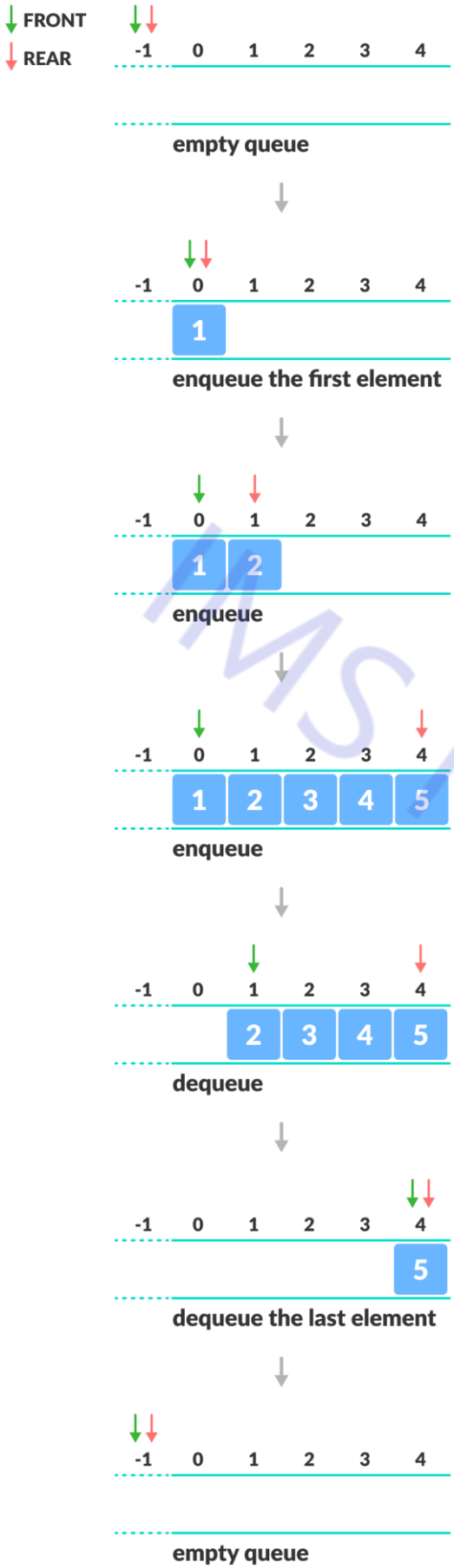
Types of Queues in Data Structure

Queue in data structure is of the following types

1. Simple Queue
2. Circular Queue
3. Priority Queue
4. Dequeue (Double Ended Queue)

The most basic queue operations in data structure are the following

- enqueue() - Adds an element at the beginning of the queue. If the queue is full, then it is an overflow.
- dequeue() - Deletes an element at the end of the queue. If the queue is empty, then it is an underflow.


FRONT
REAR
-1 0 1 2 3 4
empty queue
1
enqueue the first element
2
enqueue
3 4 5
enqueue
dequeue
dequeue the last element
empty queue

INSERT-ITEM(QUEUE,FRONT,REAR,MAX,ITEM)

Algorithm_Enqueue

```
if (REAR == N - 1) // Condition for overflow
Print “Queue Overflow”
end Algorithm_Enqueue
end if
if (FRONT == -1)   //Inserting an element in an empty queue
FRONT = REAR = 0
end if
else
 REAR = REAR +1    // Increment rear
end else
QUEUE [REAR] = element    //Assign the inserted element to the queue
end Enqueue
```

REMOVE-ITEM(QUEUE,FRONT,REAR,ITEM)

Algorithm_Dequeue

```
if(FRONT == -1)  // Condition for underflow
Print “Underflow”
end Dequeue
end if
element = QUEUE[FRONT] // assigning the front element
if(FRONT == REAR)  // Deleting the only element in the queue
FRONT = REAR = -1
else
FRONT = FRONT + 1    // Increment front
end if
end Dequeue
```

Program to implement simple queue using array

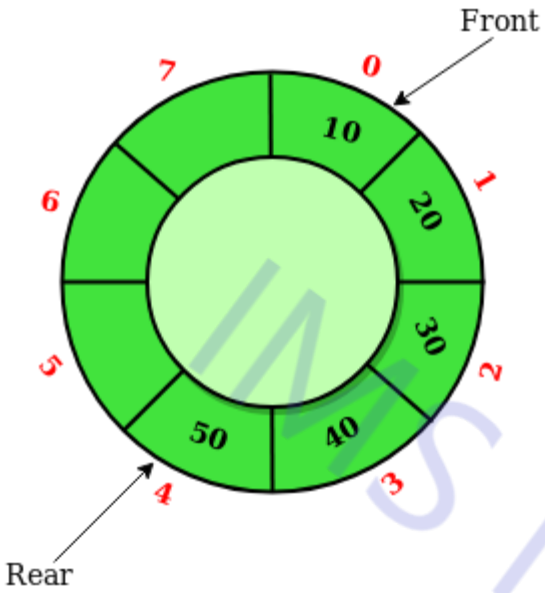
```
#include<stdio.h>
#include<stdlib.h>
#define maxsize 5
void insert();
void delete();
void display();
int front = -1, rear = -1;
int queue[maxsize];

void main ()
{
    int choice;
    while(choice != 4)
    {
        printf("\n*********Main Menu********\n");
        printf("\n=========================\n");
        printf("\n1.insert an element\n2.Delete an element\n3.Display the
queue\n4.Exit\n");
        printf("\nEnter your choice……");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            insert();
            break;
            case 2:
            delete();
            break;
            case 3:
            display();
            break;
            case 4:
            exit(0);
            break;
            default:
            printf("\nEnter valid choice??\n");
        }
    }
}

void insert()
{
    int item;
    printf("\nEnter the element\n");
    scanf("\n%d",&item);
    if(rear == maxsize-1)
    {
        printf("\nOVERFLOW\n");
        return;
    }
```

```
    if(front == -1 && rear == -1)
    {
        front = 0;
        rear = 0;
    }
    else
    {
        rear = rear+1;
    }
    queue[rear] = item;
    printf("\n%d Value inserted ",item);

}

void delete()
{
    int item;
    if (front == -1)
    {
        printf("\nUNDERFLOW\n");
        return;

    }
    else
    {
        item = queue[front];
        if(front == rear)
        {
            front = -1;
            rear = -1;
        }
        else
        {
            front = front + 1;
        }
        printf("\n%d value deleted ", item);
    }


}

void display()
{
    int i;
    if(rear == -1)
    {
        printf("\nEmpty queue\n");
    }
    else
    {
       printf("\nprinting values .....\n");
        for(i=front;i<=rear;i++)
        {
            printf("\n%d\n",queue[i]);
```

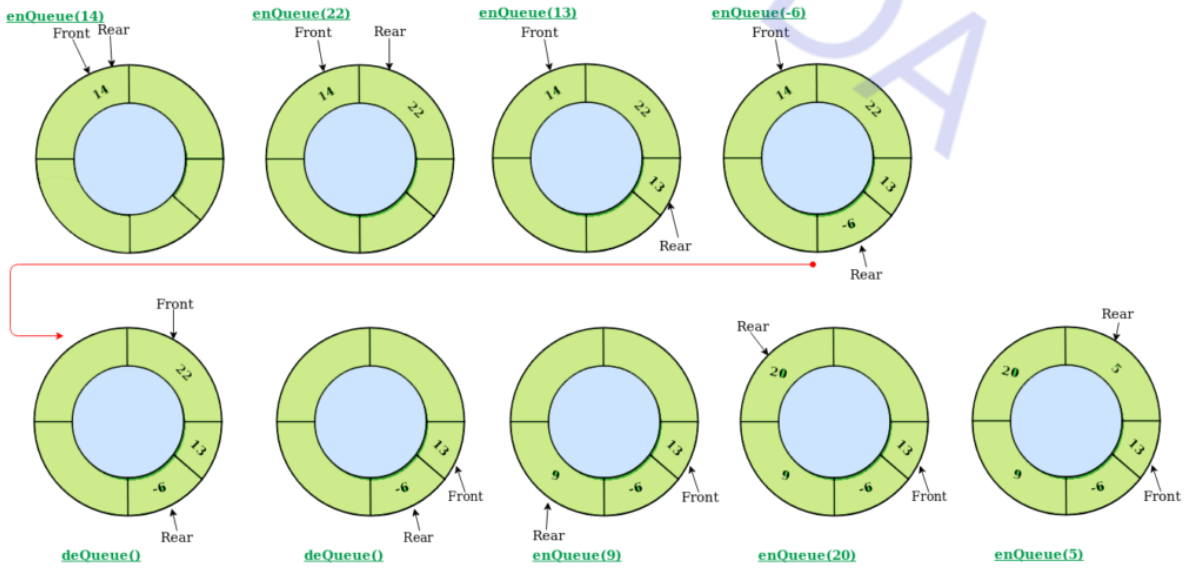
```
        }
    }
}
```

Circular Queue:

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called **'Ring Buffer'**.



In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.



Algorithm to insert an element in circular queue

- **Step 1:** IF (((REAR+1)== FRONT)||((FRONT==0)&&(REAR==MAX-1)))
  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
  SET REAR = 0
  ELSE
  SET REAR = REAR + 1
  [END OF IF]
- **Step 3:** SET QUEUE[REAR] = VAL
- **Step 4:** EXIT

Algorithm to delete an element from 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]
- **Step 4:** EXIT

Program to implement Circular Queue

```
#include <stdio.h>
#define size 5

void insertq(int[], int);
void deleteq(int[]);
void display(int[]);

int front = - 1;
int rear = - 1;

int main()
{
    int n, ch;
    int queue[size];
    do
    {
        printf("\n\n Circular Queue:\n1. Insert \n2. Delete\n3.
Display\n0. Exit");
        printf("\nEnter Choice 0-3? : ");
        scanf("%d", &ch);
        switch (ch)
        {
            case 1:
                printf("\nEnter number: ");
                scanf("%d", &n);
                insertq(queue, n);
                break;
            case 2:
                deleteq(queue);
                break;
            case 3:
                display(queue);
                break;
        }
    }while (ch != 0);
}


void insertq(int queue[], int item)
{
    if ((front == 0 && rear == size - 1) || (front == rear + 1))
    {
        printf("queue is full");
        return;
    }
    else if (rear == - 1)
    {
        rear=0;
        front=0;
    }
    else if (rear == size - 1)
    {
```

```
        rear = 0;
    }
    else
    {
        rear++;
    }
    queue[rear] = item;
    printf(“%d inserted\n”,item);
}

void display(int queue[])
{
    int i;
    printf("\n");
    if(front==-1)
    {
     printf(“queue is empty\n”);
     return;
    }
    else if (front > rear)
    {
        for (i = front; i < size; i++)
        {
            printf("%d\t ", queue[i]);
        }
        for (i = 0; i <= rear; i++)
            printf("%d\t", queue[i]);
    }
    else
    {
        for (i = front; i <= rear; i++)
            printf("%d\t ", queue[i]);
    }
}

void deleteq(int queue[])
{
 Int item;
    if (front == - 1)
    {
        printf("Queue is empty ");
    }
    Item=queue[front];
    if (front == rear)
    {
        front = - 1;
        rear = - 1;
    }
    else if (front == size-1)
    {
       front = 0;
    else
    {
        front++;
```

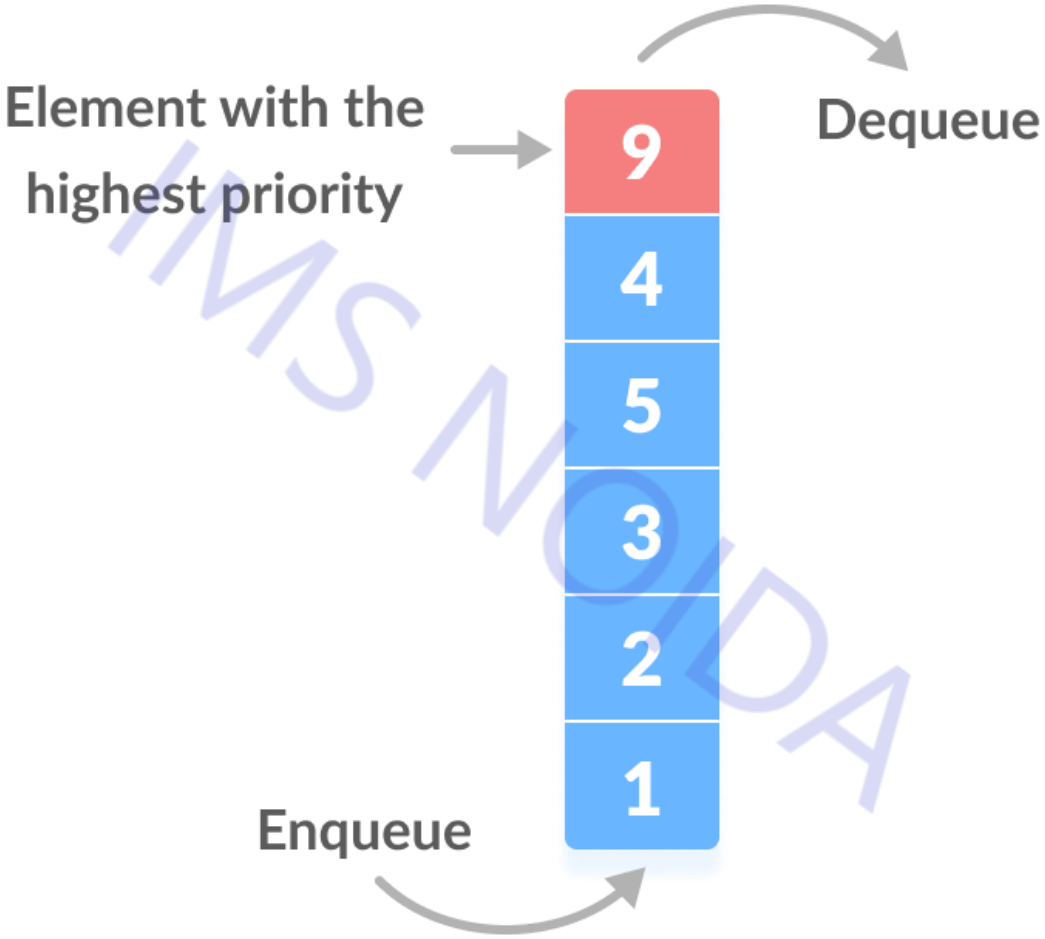
```
    }
printf("\n %d deleted", item);
}
```

Priority Queue:

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.

Generally, the value of the element itself is considered for assigning the priority.

For example, the element with the highest value is considered as the highest priority element. However, in other cases, we can assume the element with the lowest value as the highest priority element. In other cases, we can set priorities according to our needs.



Double Ended Queue:

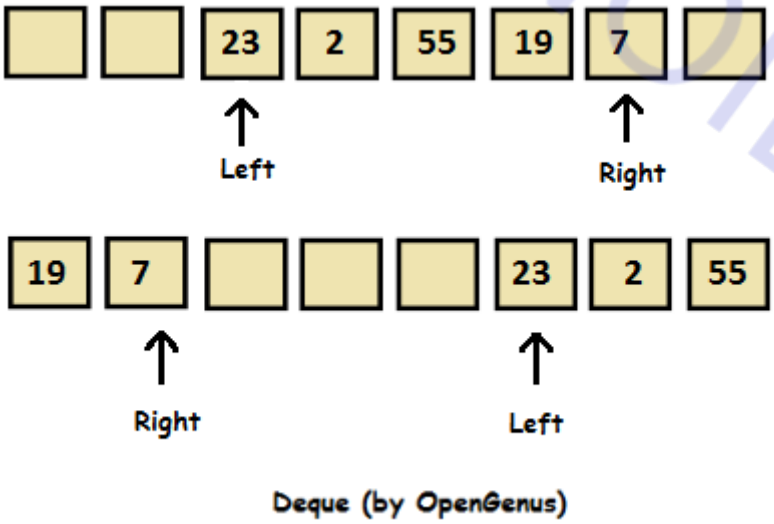
Deque or Double Ended Queue is a type of queue in which insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow FIFO rule (First In First Out).

Types of Deque

Input Restricted Deque
In this deque, input is restricted at a single end but allows deletion at both the ends.

Output Restricted Deque
In this deque, output is restricted at a single end but allows insertion at both the ends.

Insertion and Deletion in Deque



C Program to implement Double Ended Queue (Deque)

```
/* DOUBLE ENDED QUEUE */

#include<stdio.h>
#include<conio.h>
#define MAX 10

int deque[MAX];
int left=-1, right=-1;

void insert_right(void);
void insert_left(void);
void delete_right(void);
void delete_left(void);
void display(void);

int main()
{
 int choice;
 clrscr();
 do
 {
  printf("\n1.Insert at right ");
  printf("\n2.Insert at left ");
  printf("\n3.Delete from right ");
  printf("\n4.Delete from left ");
  printf("\n5.Display ");
  printf("\n6.Exit");
  printf("\n\nEnter your choice ");
  scanf("%d",&choice);
  switch(choice)
  {
   case 1:
    insert_right();
    break;
   case 2:
    insert_left();
    break;
   case 3:
    delete_right();
    break;
   case 4:
    delete_left();
    break;
   case 5:
    display();
    break;
  }
 }while(choice!=6);
 getch();
```

```
 return 0;
}


//-------INSERT AT RIGHT-------
void insert_right()
{
 int val;
 printf("\nEnter the value to be added ");
 scanf("%d",&val);
 if( (left==0 && right==MAX-1) || (left==right+1) )
 {
  printf("\nOVERFLOW");
  return;
 }
 if(left==-1)        //if queue is initially empty
 {
  left=0;
  right=0;
 }
 else
 {
  if(right==MAX-1)
   right=0;
  else
   right=right+1;
 }
 deque[right]=val;
}


//-------INSERT AT LEFT-------
void insert_left()
{
 int val;
 printf("\nEnter the value to be added ");
 scanf("%d",&val);
 if( (left==0 && right==MAX-1) || (left==right+1) )
 {
  printf("\nOVERFLOW");
  return;
 }
 if(left==-1)        //if queue is initially empty
 {
  left=0;
  right=0;
 }
 else
 {
  if(left==0)
   left=MAX-1;
  else
   left=left-1;
 }
```

```
 deque[left]=val;
}


//-------DELETE FROM RIGHT-------
void delete_right()
{
 if(left==-1)
 {
  printf("\nUNDERFLOW");
  return;
 }
 printf("\nThe deleted element is %d\n", deque[right]);
 if(left==right)       //Queue has only one element
 {
  left=-1;
  right=-1;
 }
 else
 {
  if(right==0)
   right=MAX-1;
  else
   right=right-1;
 }
}


//-------DELETE FROM LEFT-------
void delete_left()
{
 if(left==-1)
 {
  printf("\nUNDERFLOW");
  return;
 }
 printf("\nThe deleted element is %d\n", deque[left]);
 if(left==right)          //Queue has only one element
 {
  left=-1;
  right=-1;
 }
 else
 {
  if(left==MAX-1)
   left=0;
  else
   left=left+1;
 }
}


//-------DISPLAY-------
void display()
```

```
{
    int i;
    printf("\n");
    if(left==-1)
    {
     printf(“queue is empty\n”);
     return;
    }
    else if (left > right)
    {
        for (i = left; i < MAX; i++)
        {
            printf("%d\t ", deque[i]);
        }
        for (i = 0; i <= right; i++)
            printf("%d\t", deque[i]);
    }
    else
    {
        for (i = left; i <= right; i++)
            printf("%d\t ", deque[i]);
    }
}
```