# Stacks:

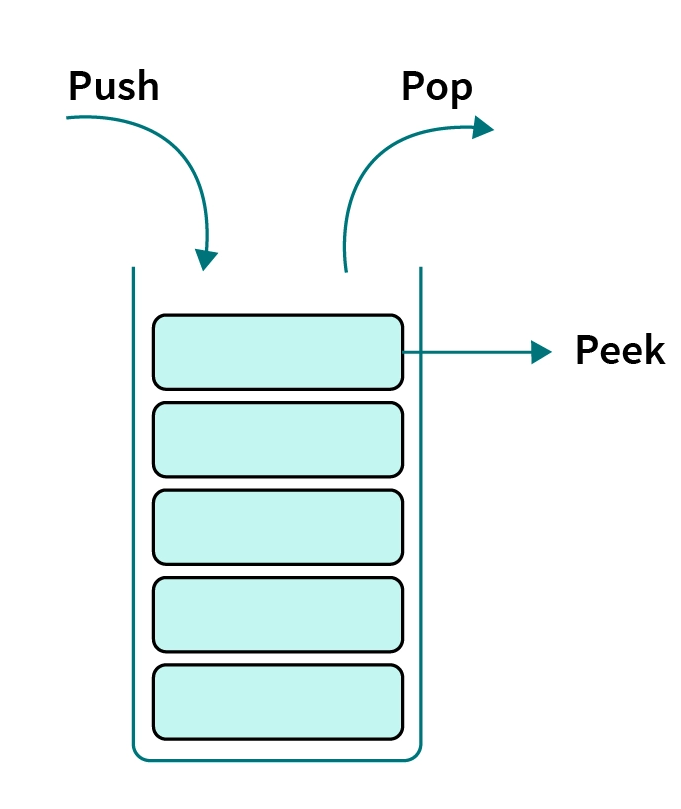
Stacks is a linear data structure, that follows the LIFO (Last-In-First- Out) principle and allows insertion and deletion operations from one end, we can insert and delete the elements from the same end and from top of the Stack.

* The stack can perform the two types of operations Push and Pop

* Inserting the elements into the stack is called Push and Deletion of the elements from the Stack is called Pop

* in which insertion and deletion can be done from the one end known as the top of the stack.

* Initially top the stack is, top = -1.



# Stack Operations:

**The following are some common operations implemented on the stack:**

***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.

# PUSH operation:

**The steps involved in the PUSH operation is given below:**



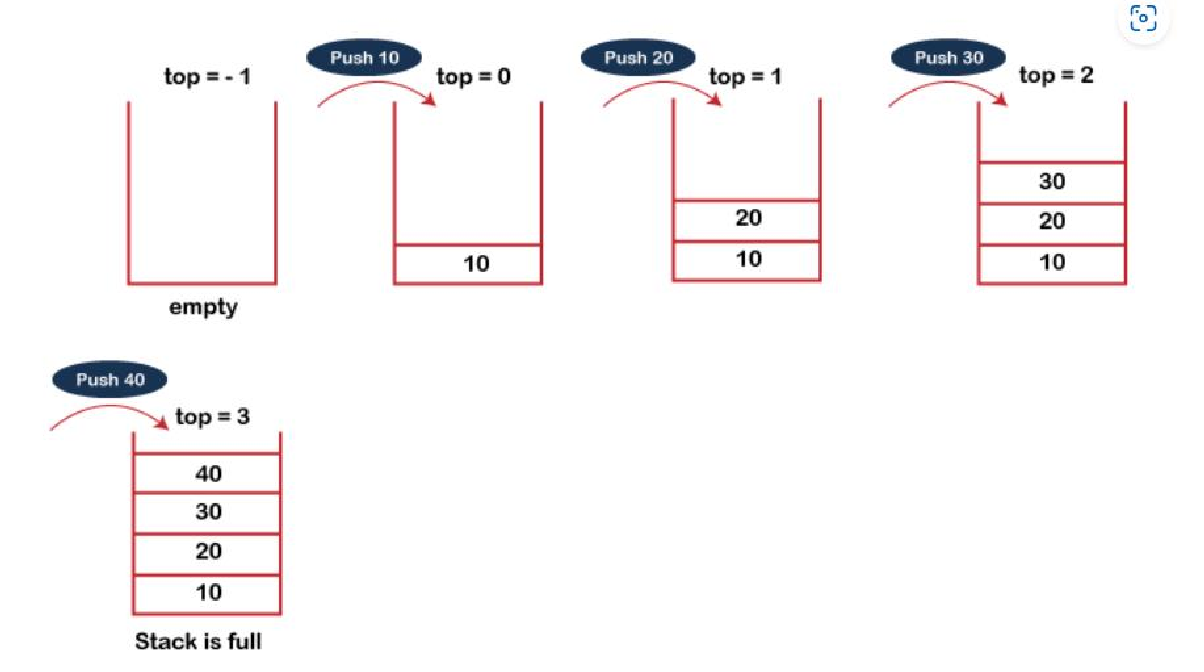
Before inserting an element in a stack, we check whether the stack is full.

If we try to insert the element in a stack, and the stack is full, then the **overflow** condition occurs.

When we initialize a stack, we set the value of top as **-1** to check that the stack is empty.

When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.

The elements will be inserted until we reach the ***max*** size of the stack.



# POP operation:



**The steps involved in the POP operation is given below:**



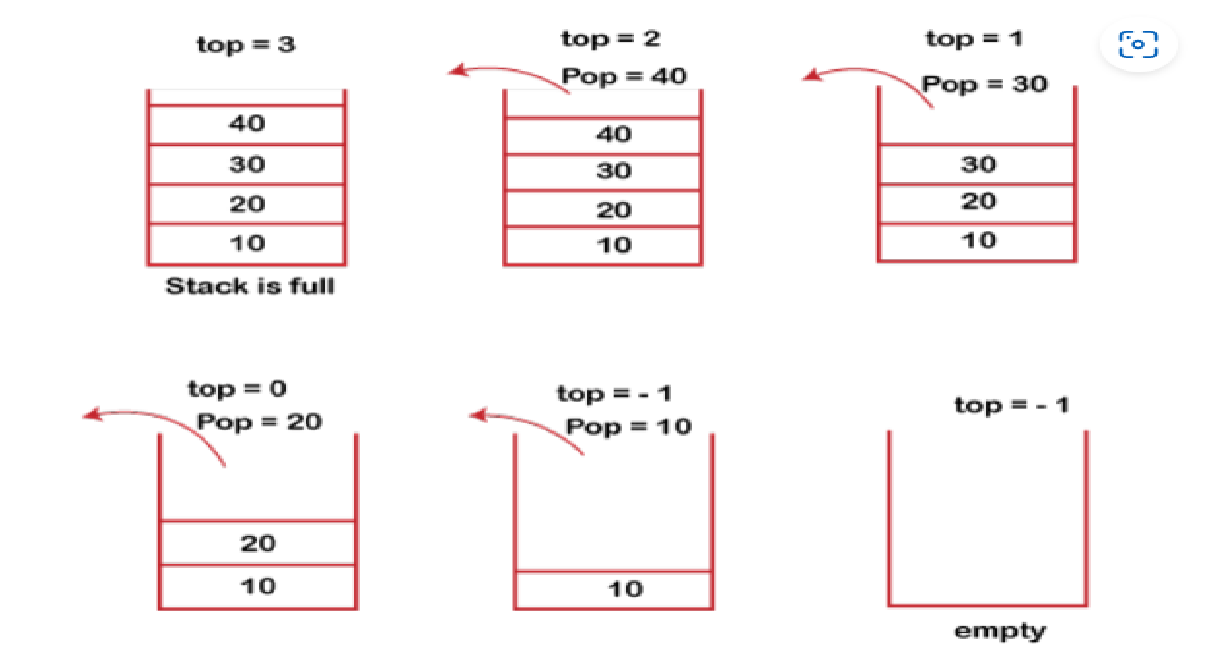
Before deleting the element from the stack, we check whether the stack is empty.

If we try to delete the element from the empty stack, then the **underflow** condition occurs.

If the stack is not empty, we first access the element which is pointed by the ***top***

Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.





# Example:

#include <stdio.h>

**int** MAXSIZE = 10;

**int** stack[10];

**int** top = -1;



**int** isempty() /\* check if the stack is empty \*/



{

if(top == -1)



return 1; else return 0;

}

**int** isfull() /\* Check if the stack is full \*/

{

|  |
| --- |
| 50 |
| 40 |
| 30 |
| 20 |
| 10 |

if(top **==** MAXSIZE) return 1;

else return 0;

}

/\* Function to return the topmost element in the stack \*/

10

30 20

40

50

**int** peek() element

{

return stack[top];

}

Peep

**int** pop() /\* Function to delete from the stack \*/

{

**int** data; if(!isempty())

{



data = stack[top]; top = top - 1; return data;



}

else

{

printf("Could not retrieve data, Stack is empty.\n");

}

}

int push(**int** data) /\* Function to insert into the stack \*/

{

if(!isfull())

{

top = top + 1; stack[top] = data;

}

else

{

printf("Could not insert data, Stack is full.\n");

}

}

int main() /\* Main function \*/

{

push(10);

push(20);

push(30);



push(40);



push(50);

printf("Element at top of the stack: %d\n" ,peek()); printf("Elements: \n");



// print stack data while(!isempty())

{

**int** data = **pop**(); printf("%d\n", data);

}

printf("Stack full: %s\n" , isfull()?"true": "false"); printf("Stack empty: %s\n" , isempty()?"true": "false"); return 0;

}

# Output:

Element at top of the stack: 50 Elements:

50

40

30

20

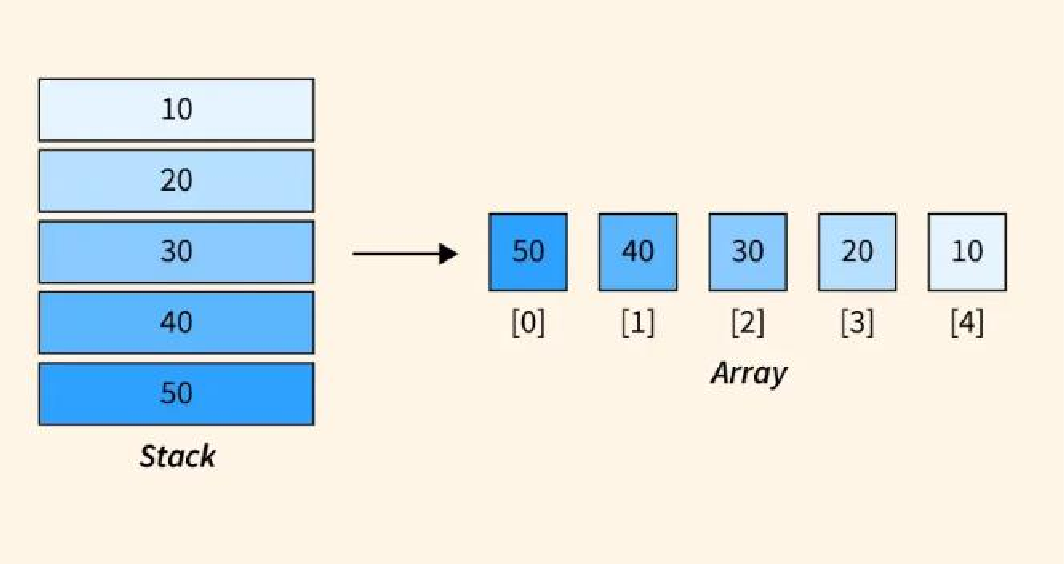
10

Stack full: False Stack empty: true

# Stack Representation as Array:

Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.





Example: // C Program for Implmentation of stack (array) using structure #include <stdio.h>

#include <stdlib.h> #include <limits.h>

// A structure to represent a stack struct Stack

{



**int** top;



**int** maxSize;

**int** \*array;

};



struct Stack \*create (**int** max)

{

struct Stack \*stack = (struct Stack \*) malloc (sizeof (struct Stack)); stack->maxSize = max;

stack->top = -1;

stack->array = (int \*) malloc (stack->maxSize \* sizeof (**int**));

//here above memory for array is being created

// size would be 10\*4 = 40 return stack;

}

// Checking with this function is stack is full or not

// Will return true is stack is full else false

//Stack is full when top is equal to the last index int isFull (struct Stack \*stack)

{

if (stack->top == stack->maxSize - 1)

{

printf ("Will not be able to push maxSize reached\n");

}

// Since array starts from 0, and maxSize starts from 1 return stack->top == stack->maxSize - 1;

}

// By definition the Stack is empty when top is equal to -1

// Will return true if top is -1

int isEmpty (struct Stack \*stack)

{

return stack->top == -1;

}

// Push function here, inserts value in stack and increments stack top by 1 void push (struct Stack \*stack, **int** item)

{



if (isFull (stack)) return;



stack->array[++stack->top] = item;



printf ("We have pushed %d to stack\n", item);

}

// Function to remove an item from stack. It decreases top by 1 int pop (struct Stack \*stack)

{

if (isEmpty (stack)) return INT\_MIN;

return stack->array[stack->top--];

}

// Function to return the top from stack without removing it

**int** peek (struct Stack \*stack)

{

**if** (isEmpty (stack))

**return** INT\_MIN;

**return** stack->array[stack->top];

}

**int** main ()

{

struct Stack \*stack = create (10);

push (stack, 10);

push (stack, 20);

push (stack, 30); int flag = 1;

while (flag)

{

**if** (!isEmpty (stack))

printf ("We have popped %d from stack\n", pop (stack));

# else

printf ("We can't Pop stack must be empty\n"); flag = 0;



}

return 0;

}



# Output:

We have pushed 10 to stack We have pushed 20 to stack We have pushed 30 to stack We have popped 30 from stack

# Stacks using Dynamic Arrays:

A Dynamic Stack is a stack data structure whose capacity (maximum elements that can be stored) **increases** or decreases in runtime, based on the operations (**push or pop**) performed on it.

**Example: //**Write a program to implement following operations of dynamic Stack.



1. Push the element. 2. Pop the element. 3. Display 4. Exit.



#include <stdio.h>

**int** stack [10], i, j, choice=0, n, top=-1;

**void** push (); **void** pop (); **void** show (); **void** main ()

{

printf("Enter the number of elements in the stack "); scanf("%d", &n);

printf ("\*\*\*\*\*\*\*\*\*Stack operations using array\*\*\*\*\*\*\*\*\*"); printf("\n \n");

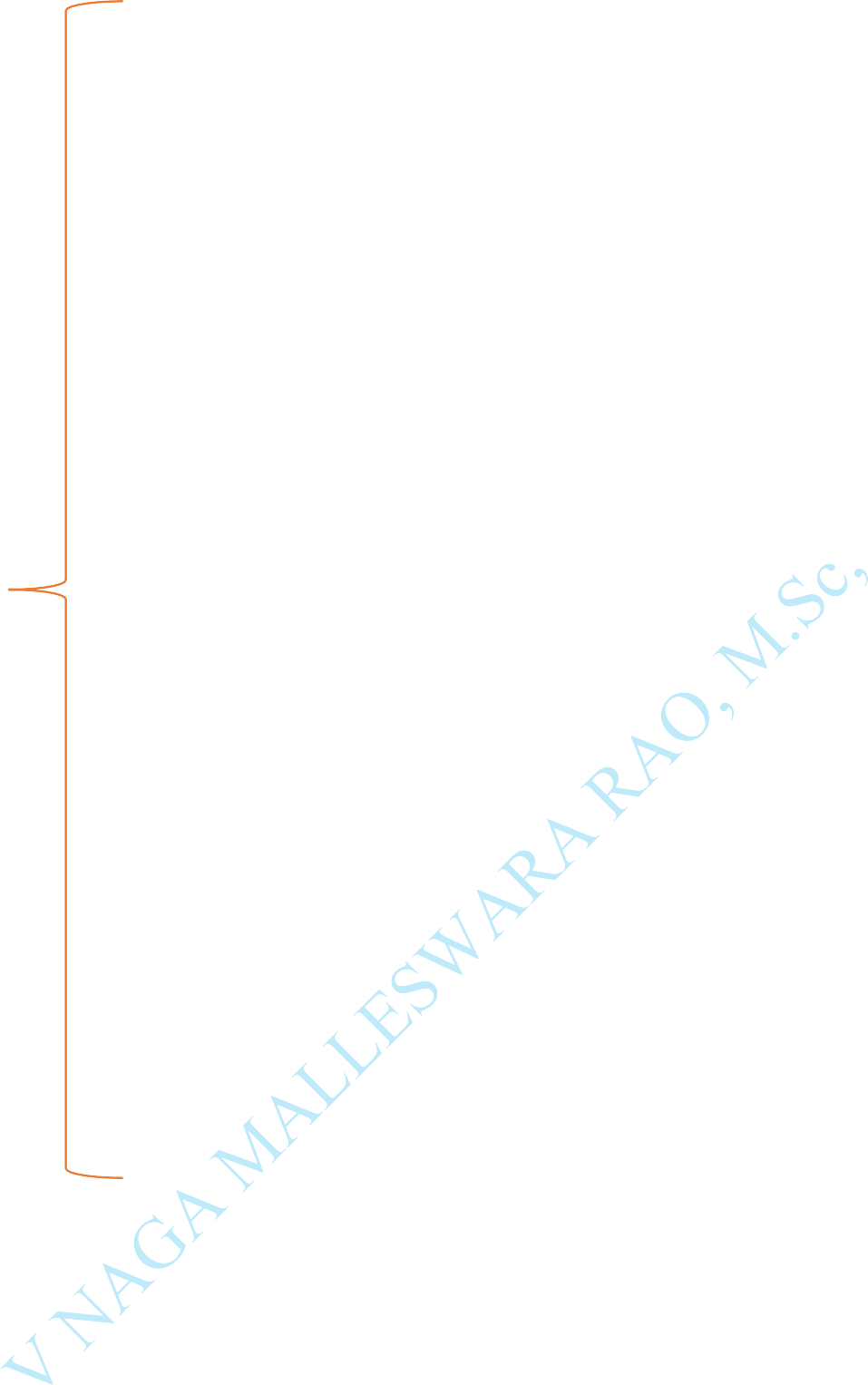
**while** (choice! = 4)

{

printf("Choose one from the below options...\n"); printf("\n**1**.Push\n **2**.Pop\n **3**.Show\n **4**.Exit"); printf("\n Enter your choice \n");

scanf("%d", &choice);





**switch**(**choice**)

{

**case** 1:

{

push();

**break**;

}

**case** 2:

{

pop();

**break**;

}

**case** 3:

{

show();

**break**;

}

**case** 4:

{

printf("Exiting ");

**break**;

}

**default**:

{

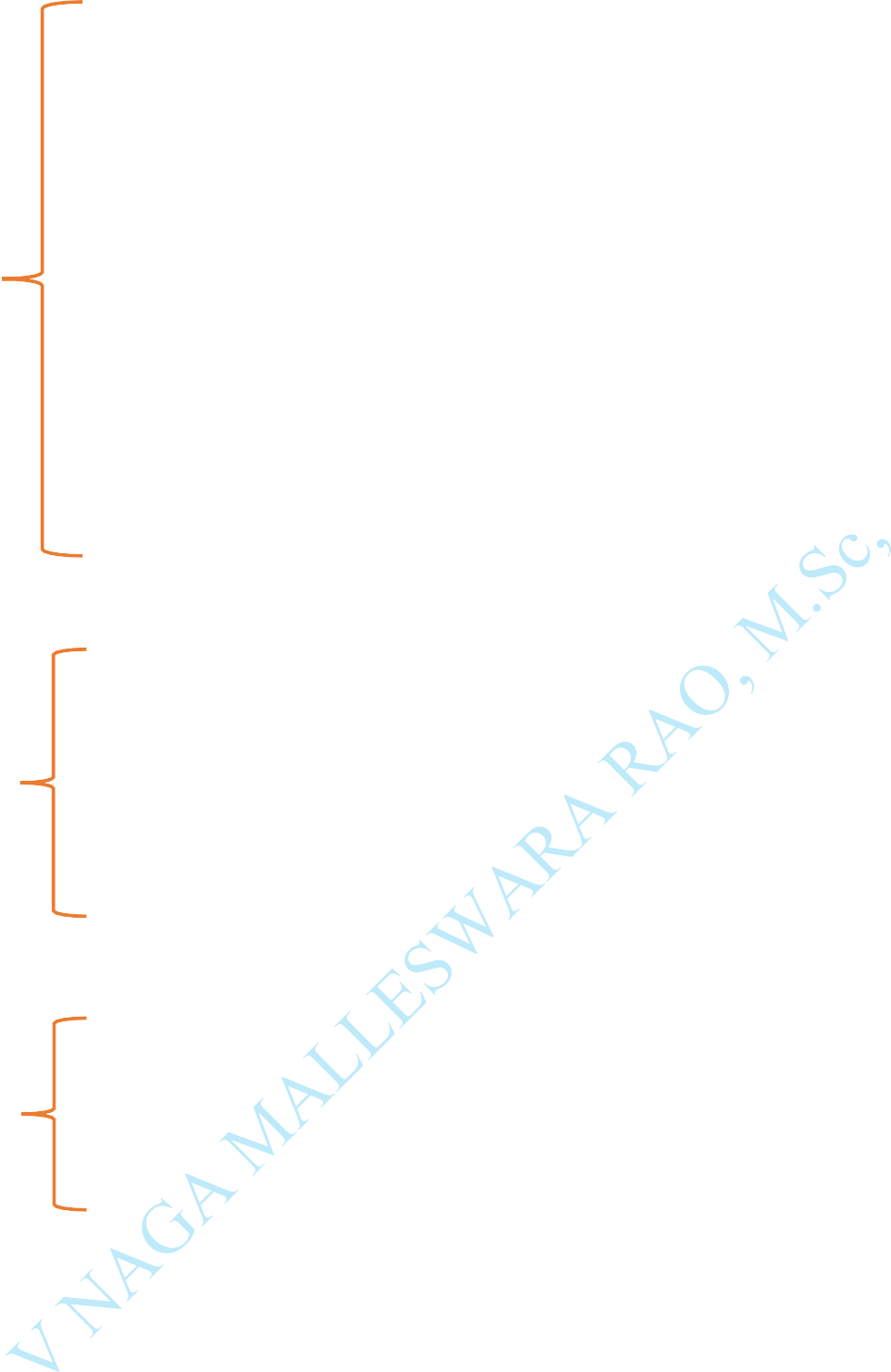
printf("Please Enter valid choice ");

}

};

}

}

**void** push ()

{

**int** val;

**if** (top == **n** ) printf("\n Overflow"); **else**



{

printf("Enter the value?"); scanf("%d",&**val**);



top = **top** +1; stack[**top**] = **val**;

}

}

# void pop ()

{

**if**(**top** == -1) printf("Underflow"); **else**

top = **top** -1;

}

# void show()

{

**for** (i=**top**; i>=0;i--)

{

printf ("%d\n", **stack**[ **i** ]);

}

**if** (**top** == -1)

{

printf ("Stack is empty");

}

}

# Output:

Enter the number of elements in the stack 2

\*\*\*\*\*\*\*\*\*Stack operations using array\*\*\*\*\*\*\*\*\*

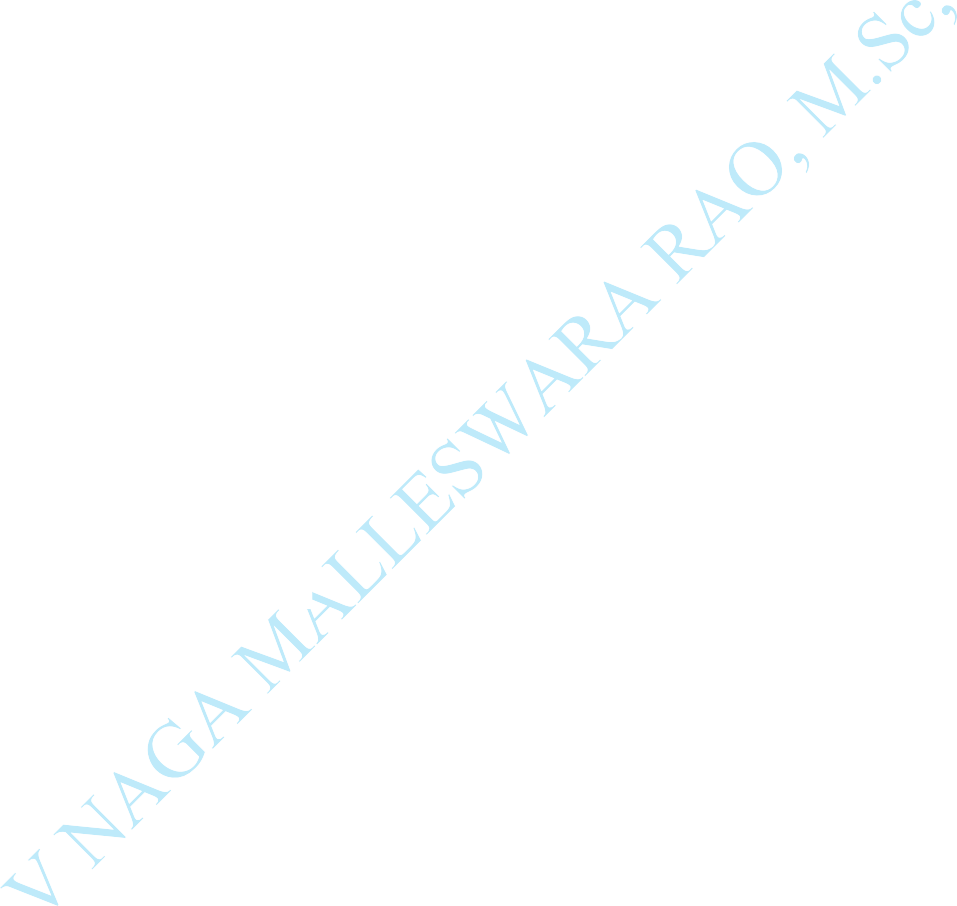
Choose one from the below options... 1.Push



1. Pop 3.Show 4.Exit



Enter your choice



**1**

Enter the value10

Choose one from the below options... 1.Push

2.Pop 3.Show 4.Exit

Enter your choice

**1**

Enter the value20

Choose one from the below options... 1.Push

2.Pop 3.Show

4.Exit

# Enter your choice 1

Enter the value30

Chose one from the below options...

1.Push 2.Pop 3.Show 4.Exit



# Enter your choice 3

30

20

10

# Choose one from the below options...

1.Push 2.Pop 3.Show 4.Exit

# Enter your choice

**Stack Applications:**

**Polish notation:** Infix to postfix conversion, evaluation of postfix expression:

** Polish notation is a notation form for expressing arithmetic, logic and algebraic equations.



Its most basic distinguishing feature is that operators are placed on the left of their operands.

If the operator has a defined fixed number of operands, the syntax **does**

# *not require brackets or parenthesis.



1. **Evaluation of Arithmetic Expressions:**

A stack is a very effective Data Structure for evaluating arithmetic expressions in programming languages. An arithmetic expression consists of **operands** and **operators.**

Evaluation of Arithmetic Expression requires two steps:



First, convert the given expression into special notation.

Evaluate the expression in this new notation.

Notations for Arithmetic Expression:

There are three notations to represent an arithmetic expression:

# Infix Notation

* 1. **Prefix Notation**

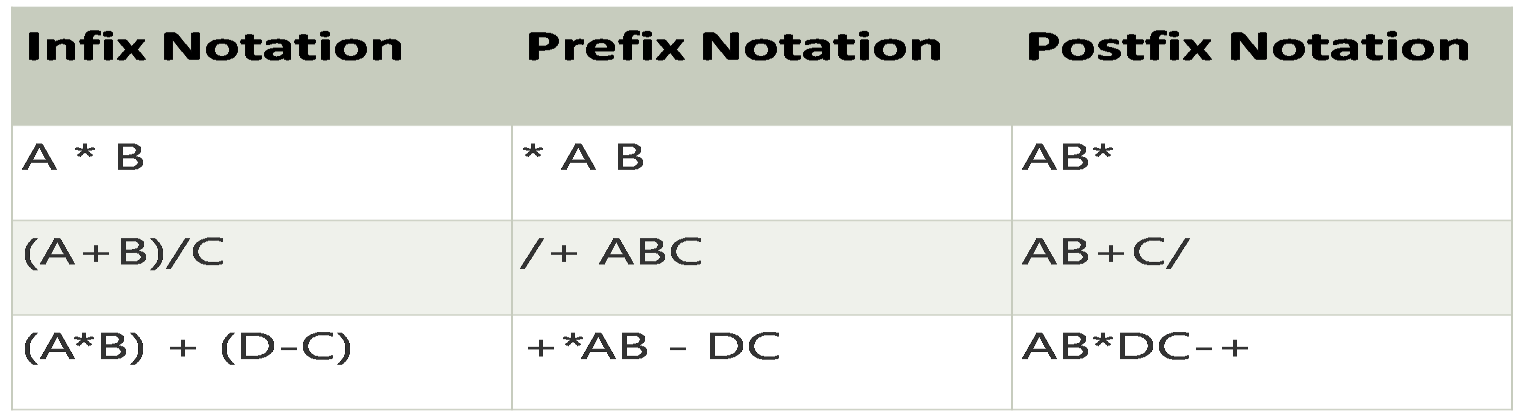
# Postfix Notation Infix Notation:

The infix notation is a convenient way of writing an expression in which each operator is placed between the operands. Infix expressions can be parenthesized or un parenthesized depending upon the problem requirement.

**Example: A + B** (**C - D**) (Operand, Operator, Operand)

# Prefix Notation:

The prefix notation places the operator before the operands. This notation was introduced by the Polish mathematician and hence often referred to as polish notation.



**Example: +** A B **-** CD (Operator, Operand, Operand)

**Postfix Notation:**

The postfix notation places the operator after the operands. This notation is just the reverse of Polish notation and also known as Reverse Polish notation.

**Example:** AB **+** CD **-** (Operand, Operand, Operator)

# Evaluation of Postfix: Example: 1 2 3 \* + 4 -



**Step:1** Push the operand 1 into the Stack



**Step:2**

Push the operand 2 into the Stack

**Step:3**

Push the operand 3 into the Stack

**Step:4**

Push the operator \* into the Stack

Now evaluate 2**\***3 and store the result 6

**Step:5**

Push the Operator **+** into the Stack

Now evaluate 1**+6** and store the result 7

**Step:6**

Push the operand 4 into the Stack

**Step:7**

Push the Operator - into the Stack

Now evaluate 7 - 4 and store the result 3

**3**

**7**

**4**

**7**

**1**

**6**

**+**

**1**

**6**

**1**

**2**

**1**

|  |
| --- |
|  |
| **3** |
| **2** |
| **1** |

|  |
| --- |
| \* |
| **3** |
| **2** |
| **1** |

|  |
| --- |
|  |
| **-** |
| **4** |
| **7** |

# Example:

#include <stdio.h> #include <ctype.h> #include <stdlib.h>



#define SIZE 10

**int** pop();

void push(**int**); char postfix[SIZE]; **int** stack[SIZE]; **top** = -1;

**int** main()

{

**int** i, val1, **val2**, result, pEval; char **ch**;

**for**(i=0; i<SIZE; i++)

{

stack[i] = -1;

}

printf("\n Enter a postfix expression: "); scanf("%s", postfix);

**for**(i=0; postfix[i] != '\0'; i++)

{

ch = postfix[i];

**if**( isdigit (**ch**))

{

push(ch-'0');

}

else if(**ch** == '**+**' || **ch** == '-' || **ch** == '\*' || **ch** == '**/**')

{

val2 = pop(); val1 = pop(); switch(ch)



{

**case** '+': result = val1 + val2; break;

**case** '-': result = val1 - val2; break;

**case** '\*': result = val1 \* val2; break;

**case** '/': result = val1 / val2; break;

**case** '%': result = val1 % val2; break;

}

printf("\n %d", result); push(result);

}

}

pEval = pop();

printf("\n The postfix evaluation is: %d\n", pEval); return 0;

}

void push(**int** n)

{

**if** (top < SIZE -1)

{

stack[**++**top] = n;

}

else

{

printf("Stack is full!\n");



exit(-1);

}

}

**int** pop()

{

**int n**;

**if** (top > -1)

{

n = stack[top]; stack[top--] = -1; return **n**;

}

else

{

printf("Stack is empty!\n"); exit(-1);

}

}

**Output:**

Enter a postfix expression:

1 2 3 \* + 4 -

The postfix evaluation is: 1

# Recursion:

Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem. Any function which calls itself is called recursive function, and such function calls are called recursive



calls. Recursion involves several numbers of recursive calls.



Performing the same operations multiple times with different inputs. In every step, we try smaller inputs to make the problem smaller.

Base condition is needed to stop the recursion otherwise infinite loop will occur.

approach (1): – Simply adding one by one f(n) = 1 + 2 + 3 + + n



approach (2): – Recursive adding f(n) = 1 n=1

f(n) = n + f(n-1) n>1

# Algorithm of Factorial Program in C:



The algorithm of a C program to find factorial of a number is: Start program

Ask the user to enter an integer to find the factorial Read the integer and assign it to a variable

From the value of the integer up to 1, multiply each digit and update the final value

The final value at the end of all the multiplication till 1 is the factorial End program.

**Example:**

#include <stdio.h>

**int** fact (**int**);

**int** main()

{

**int** n,f;



printf("Enter the number whose factorial you want to calculate?"); scanf("%d",&n);



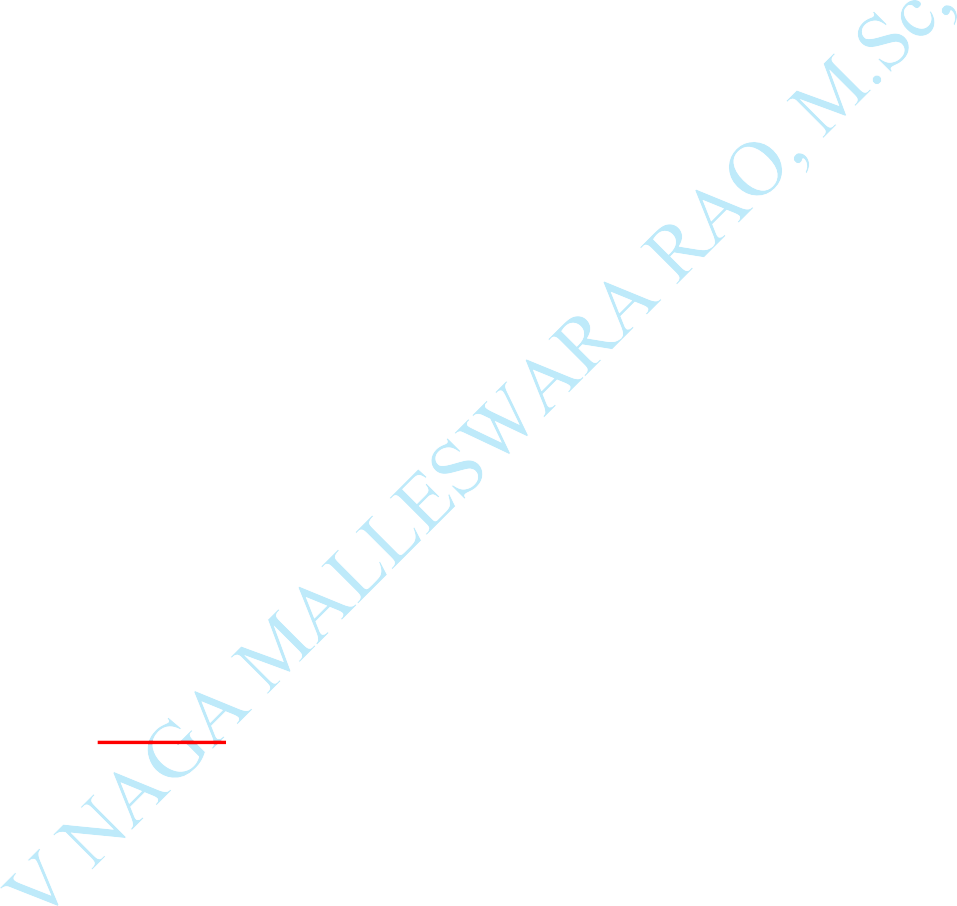
f = fact(n); printf("factorial = %d",f);



}

**int** fact(**int** n)

{

**if** (n==0)

{

**return** 0;

}

**else if** ( n == 1)

{

**return** 1;

}

# else

{

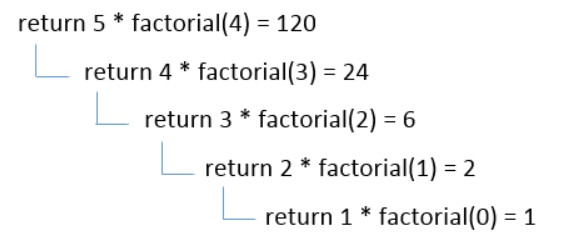
**return** n\*fact(n-1);

}

}

# Output:

Enter the number whose factorial you want to calculate?5 factorial = 120



**GCD:**

GCD stands for Greatest Common Divisor. GCD of two numbers in C is the largest positive integer that completely divides both the given numbers. **Example:**

GCD (10,15) = 5, GCD (12,15) = 3.



The GCD is a mathematical term for the **Greatest Common Divisor** of two or more numbers.

It is the Greatest common divisor that completely divides two or more numbers without leaving any remainder.

Therefore, it is also known as the **Highest Common Factor (HCF)** of

two numbers.

**Example:**

The GCD of two numbers, 20 and 28, is 4 because both 20 and 28 are

completely divisible by 1, 2, 4 (the remainder is 0),

**Example:** #include <stdio.h> **int** main()

{



**int** n1, n2;

printf("Enter two positive integers: "); scanf("%d %d",&n1,&n2); while(n1**!=** n2)

{

**if**(n1 > n2)

**n1** -= **n2**;

**else**

**n2** -= **n1**;

}

printf("**GCD** = %d",n1); return 0;

}

**Output:**

Enter two positive integers: 50

75

**GCD** = 25

# Fibonacci Series in C:

In case of Fibonacci series, next number is the sum of previous two numbers for example 0, 1, 1, 2, 3, 5, 8, 13, 21 etc. The first two numbers of Fibonacci series are 0 and 1.

There are two ways to write the Fibonacci series program:



Fibonacci Series without recursion Fibonacci Series using recursion

# Fibonacci Series in C without recursion:



#include<stdio.h>

**int** main()

{

**int n1**=0, **n2**=1, **n3**, **i** , number; printf("Enter the number of elements:"); scanf("%d", &number);

printf("\n %d %d",**n1**,**n2**); //printing 0 and 1

**for**(i=2; i<number; ++i)

//loop starts from 2 because 0 and 1 are already printed

{

**n3**=**n1**+**n2**; printf(" %d",**n3**); **n1**=**n2**;

**n2**=**n3**;

}

**return** 0;

}

# Output:

Enter the number of elements: 9 0 1 1 2 3 5 8 13 21

# Fibonacci Series using recursion in C

#include<stdio.h>

**void** printFibonacci(**int n**)

{

**static int n1**=0, **n2**=1, **n3**; **if**(n>0)

{

**n3** = **n1** + **n2**; **n1** = **n2**;

**n2** = **n3**; printf("%d ", **n3**);

printFibonacci(**n-1**);

}

}

**int** main()

{

# int n;

printf("Enter the number of elements: "); scanf("%d", &n);

printf("Fibonacci Series: "); printf("%d %d ",0,1);

printFibonacci(**n-2**); //n-2 because 2 numbers are already printed

**return** 0;

}

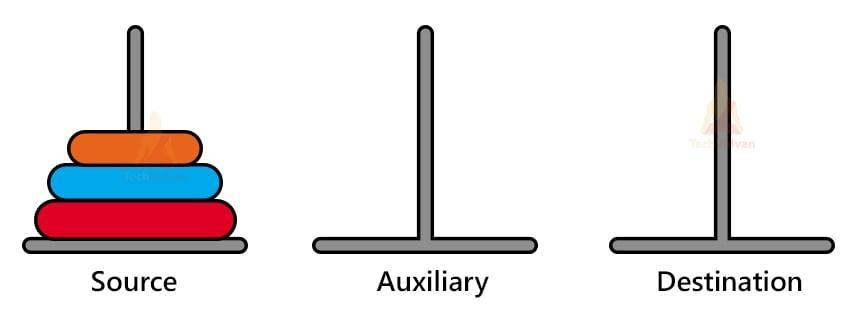
# Output:

Enter the number of elements: 9 Fibonacci Series: 0 1 1 2 3 5 8 13 21

# Tower of Hanoi:

What is Tower of Hanoi in data structure?

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted − These rings are of different sizes and stacked upon in an ascending order. i.e., the smaller one sits over the larger one.



# Rules:

The mission is to move all the disks to some another tower without

violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −

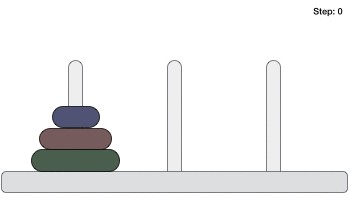
**Only one disk can be moved among the towers at any given time. Only the "top" disk can be removed.

*No large disk can sit over a small disk.

*First, we move the smaller (top) disk to aux peg.

*Then, we move the larger (bottom) disk to destination peg.

*And finally, we move the smaller disk from aux to destination peg.



**A**

**B**

**C**

# Example:

# #include <stdio.h>

# 

# // C recursive function to solve tower of hanoi puzzle

# void towerOfHanoi(int n, char from\_rod, char to\_rod, char aux\_rod)

# {

# if (n == 1)

# {

# printf("\n Move disk 1 from rod %c to rod %c", from\_rod, to\_rod);

# return;

# }

# towerOfHanoi(n-1, from\_rod, aux\_rod, to\_rod);

# printf("\n Move disk %d from rod %c to rod %c", n, from\_rod, to\_rod);

# towerOfHanoi(n-1, aux\_rod, to\_rod, from\_rod);

# }

# 

# int main()

# {

# int n = 4; // Number of disks

# towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods

# return 0;

# }

#include <stdio.h>

void towers(**int**, **char**, **char**, **char**); **int** main()

{

**int** num;

printf("Enter the number of disks : "); scanf("%d", &num);

printf("The sequence of moves involved in the Tower of Hanoi are :\n"); towers(num, '**A**', '**C**', '**B**');

return 0;

}

void towers(**int** num, **char** frompeg, **char** topeg, **char** auxpeg)

{

// Base Condition if no of disks are

**if** (num == **1**)

{

printf("\n Move disk 1 from peg %c to peg %c", frompeg, topeg); return;

}

// Recursively calling function twice towers(num - 1, frompeg, auxpeg, topeg);

printf("\n Move disk %d from peg %c to peg %c", num, frompeg, topeg); towers(num - 1, auxpeg, topeg, frompeg);

}

# Output:

Enter the number of disks: 3

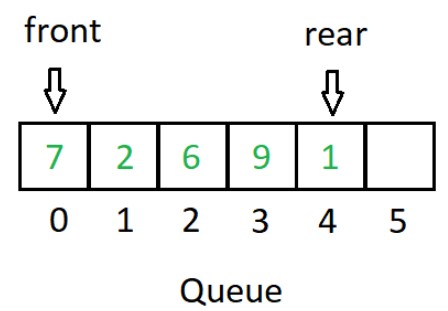
The sequence of moves to Tower of Hanoi are: Move disk 1 from peg A to peg C

Move disk 2 from peg A to peg B Move disk 1 from peg C to peg B Move disk 3 from peg A to peg C Move disk 1 from peg B to peg A Move disk 2 from peg B to peg C Move disk 1 from peg A to peg C

# Queues:

**What is a queue in C?**

A queue in C is basically a linear data structure to store and manipulate the data elements. It follows the order of First In First Out



(FIFO). In queues, the first element entered into the array is the first element to

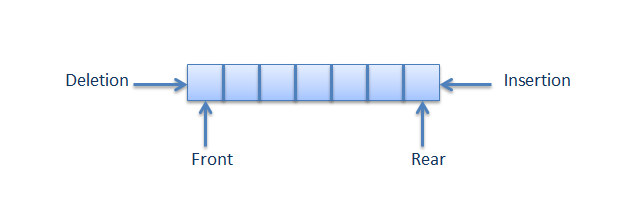
be removed from the array.

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

# Simple Queue:

**Simple Queue** is also called as Linear Queue, an insertion takes

place from one end and 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.

# Algorithm:



**Enqueue:**

▶ **Step 1:** IF REAR = MAX-1 PRINT “OVERFLOW”

Goto step 4 [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] = VAL

▶ **Step 4:** EXIT

# Dequeue:

▶ **Step 1:** IF FRONT = -1

PRINT “UNDERFLOW” [END OF IF]

▶ **Step 2:**



SET VAL = QUEUE[FRONT] SET FRONT = FRONT+1



IF FRONT > REAR FRONT = REAR = -1 [END OF IF]

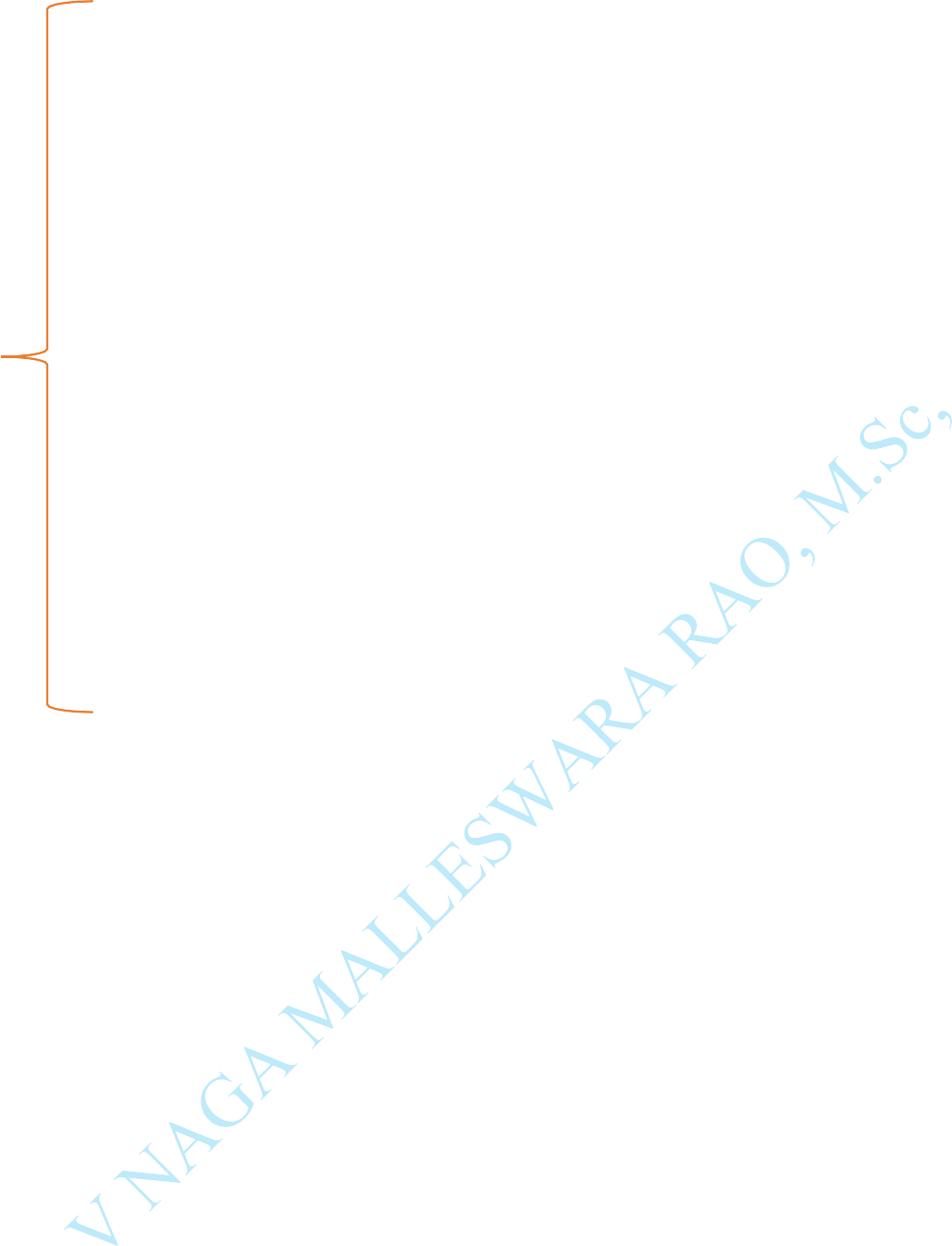


▶ **Step 3:** EXIT

**Example:** #include<stdio.h> #define MAX 5







**int** front = -1; **int** rear = -1; **int** queue[];

**void** enqueue(**int** queue[], **int val**)

{

**if**(**rear** == MAX -1) printf("\n OVERFLOW");

**if**(**front** == -1 && **rear** == -1) front = rear = 0;

**else rear++**;

queue[**rear**] = **val**;

}

**int** dequeue (**int queue**[])

{

**if**(**front** == -1 || front > **rear**)

{

printf("\n UNDERFLOW"); return -1;

}

**else**

{

**int val** = queue[**front**]; front++;

return **val**;

}

}

**void** display(**int** queue[])

{

**if**(front == -1 && rear == -1) printf("|n Queue is empty!"); **else**



{

**int** i; printf("\n");



**for**(i = front; i <= rear; i++) printf("\t %d", queue[i]);

}

}

**int** main()

{

**int** option, val; do

{

printf("\n \*\*\*\*\* Enter Your Choice \*\*\*\*\*"); printf("\n **1.** Insert an element");

printf("\n **2.** Delete an element"); printf("\n **3.** display the queue"); printf("\n **4. EXIT**");

printf("\n \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"); printf("\n\n ENTER your option: "); scanf("%d", &option);

switch(option)

{

case 1:

printf("\n Enter the element to the queue: "); scanf("%d", &**val**);



enqueue(queue, val ); break;

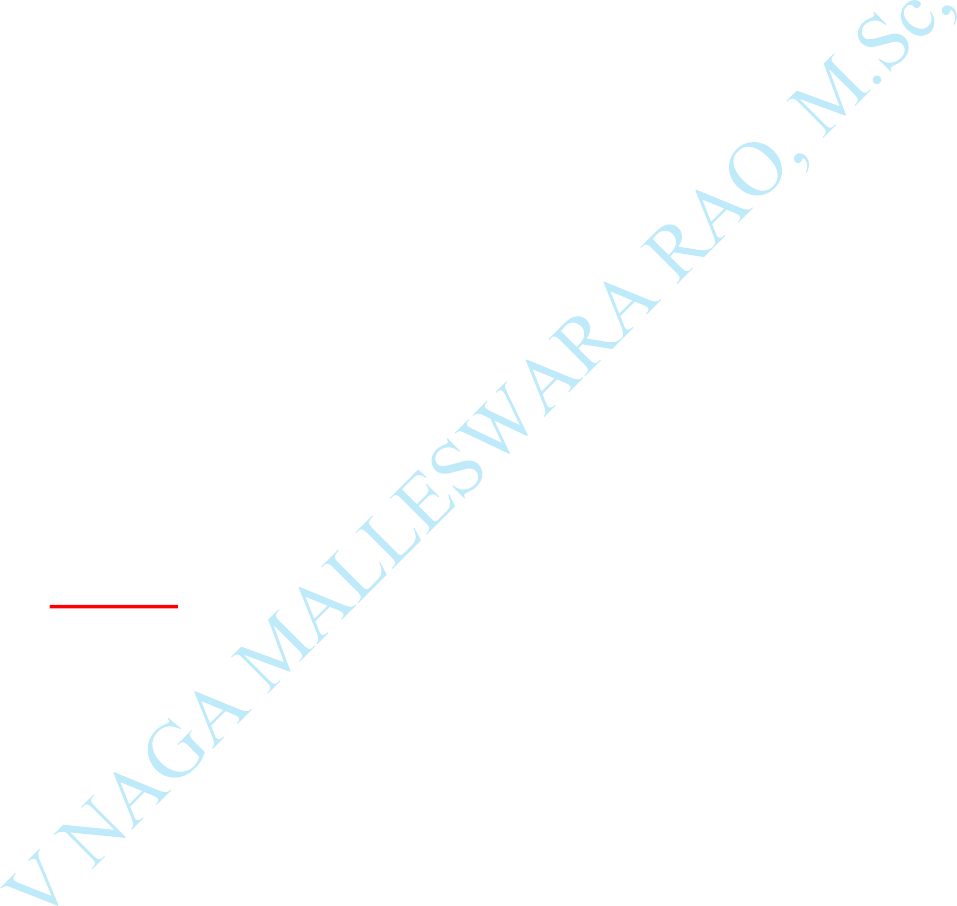


case 2:



val = dequeue(queue);

**if** (val != -1)

printf("\n The number deleted is : %d", **val**); break;

case 3: display(queue); break;

}

}

while(option!= 4);

}

# Output:

\*\*\*\*\* Enter Your Choice \*\*\*\*\*

1. Insert an element
2. Delete an element
3. display the queue
4. EXIT

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ENTER your option: 1

Enter the element to the queue: 10

# 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.

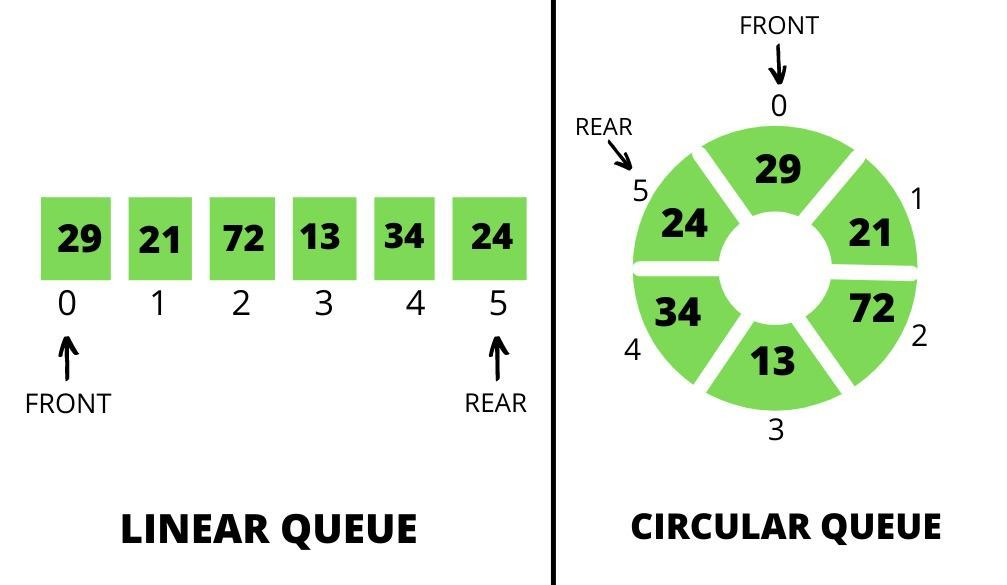
**Front:** It is used to get the front element from the Queue.

**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.



# 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

# 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

# Example:

#include<stdio.h> #include<stdlib.h>

# define N 5 //N = Maximum Size of the Q

**int cq** [N]; //cq = Circular Queue



**int** front=-1;



**int** rear=-1;

**void** enqueue (**int item**); **int** dequeue ();



**int** isEmpty ();

**int** isFull ();

void enqueue(**int** item)

{

**if**(isFull ())

{

printf("Queue Overflow: \n"); exit(1);

}

**if**(front == -1)

{

front=0;

}

rear=(rear+1) % N; // rear is incremented

**cq**[**rear**]=item; // assigning a value to the queue at the rear position.

}

**int** dequeue()

{

**int** deleted\_item;

**if**(isEmpty ())

{



printf("Queue Underflow: \n");



**exit**(1); // To show Failure Exit.

}



deleted\_item = **cq** [front];

**if**(front==rear) // cq has only 1 element

{

front=-1; rear=-1;

}

else

{

front = (front + 1) %N;

}

return deleted\_item;

}

**int** isEmpty ()

{

return ( (front == -1) ? 1 : 0);

}

**int** isFull()

{

return ( ( (rear + 1)%N == front) ? 1 : 0);

}

void display()

{

**int** i;

**if**(isEmpty ())

{

printf("Queue Underflow: \n."); return;



}



**i** = front;

**if** (front < rear)

{

while(i<=rear)

{

printf("%d,", cq[i]); i=i+1;

}

}

else

{

while(i != rear)

{

printf("%d,", **cq**[i]); i=(i+1)%N;

}

printf("%d,", **cq**[i]);

}

}

**int** main()

{

enqueue (**11**); // Insert elements: 11,22,33,44,55

enqueue (**22**);

enqueue (**33**);

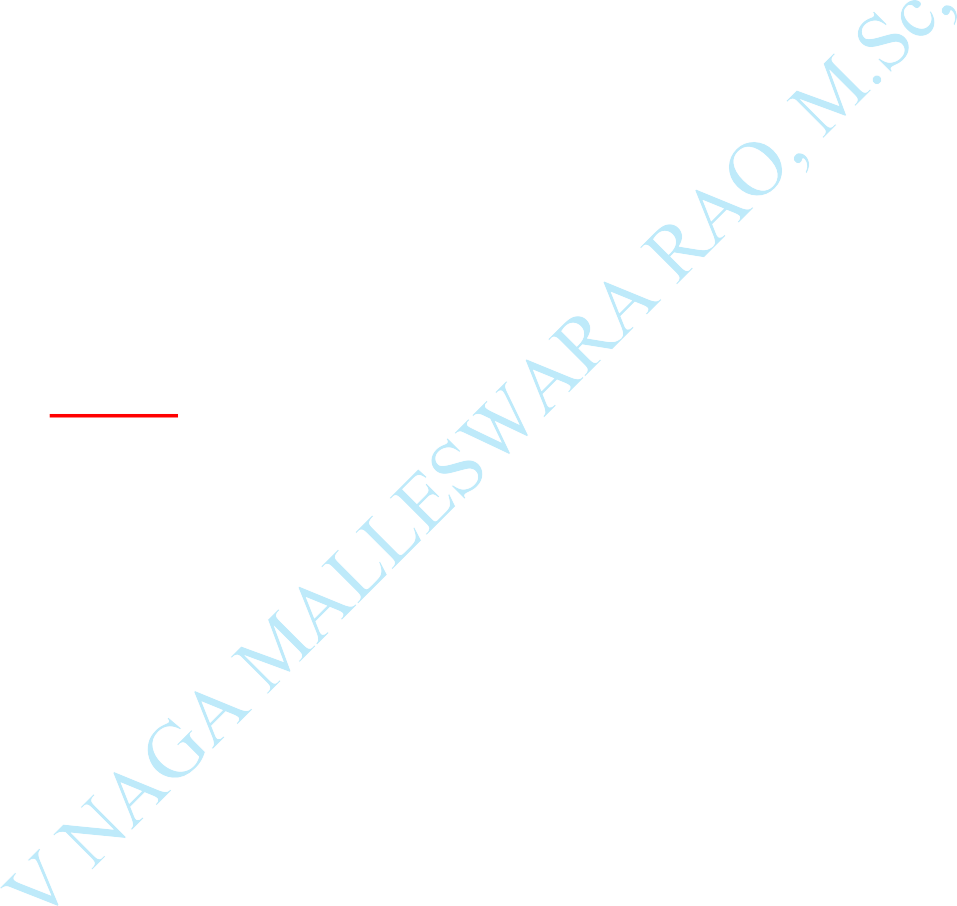


enqueue (**44**);



enqueue (**55**); display ();



printf("Deleted Element is: %d\n", dequeue()); // Delete first elements printf("Deleted Element is: %d\n", dequeue()); // Delete Second elements display ();

enqueue (**66**); // Insert 66 display ();

return **0**;

}

# Output:

11,22,33,44,55

**Deleted Element is:** 11 Deleted Element is: 22 33,44,55

33,44,55,66

# Priority Queue:

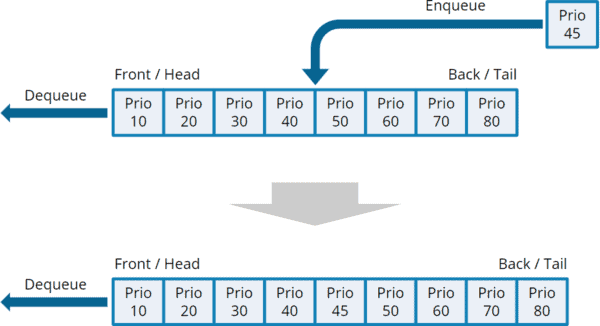
*It is a special type of queue in which all the elements are arranged based on the priority and every element has a priority associated with it.

*Suppose some elements occur with the same priority, they will be



arranged according to the **FIFO** principle.

An ascending order priority queue gives the highest priority to the lower number in that queue.



**There are two types of priority queue:**



**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.

# Example:

#include<stdio.h> #include<limits.h> #define MAX 100 **int** idx = -1;

**int** pqVal[**MAX**];



**int** pqPriority[**MAX**];

**int** isEmpty()

{



return idx == -1;

}

**int** isFull()

{

**return** idx == MAX - 1;

}

**void** enqueue(int data, int priority)

{

**if**(!isFull())

{

**idx**++;

pqVal[idx] = data; pqPriority[idx] = priority;

}

}

**int** peek()

{

**int** maxPriority = INT\_MIN;

**int** indexPos = -1;

**for** (int i = 0; i <= idx; i++)

{

**if** (maxPriority == pqPriority[i] && indexPos > -1 && pqVal[indexPos] < pqVal[i])

{

**maxPriority** = pqPriority[i];

**indexPos** = i;

}

**else if** (maxPriority < pqPriority[i])

{

maxPriority = pqPriority[i]; indexPos = i;

}

}

return indexPos;



}



**void** dequeue()



{

**if**(!isEmpty())

{

**int** indexPos = peek();

**for** (**int** i = indexPos; i < idx; i++)

{

**pqVal**[i] = pqVal[i + 1]; pqPriority[i] = pqPriority[i + 1];

}

**idx**--;

}

}

**void** display()

{

**for** (**int** i = 0; i <= **idx**; i++)

{

printf("(%d, %d)\n",pqVal[i], pqPriority[i]);

}

}

**int** main()

{

**enqueue**(5, 1);

**enqueue**(10, 3);



**enqueue**(15, 4);

**enqueue**(20, 5);

**enqueue**(25, 2);

**printf**("Priority Queue Before Dequeue : \n");

**display**(); **dequeue**(); **dequeue**(); **dequeue**();

**printf**("\nPriority Queue After Dequeue : \n");

**display**(); **return** 0;

}

**Output:**

Priority Queue Before Dequeue:

**(5, 1)**

**(10, 3)**

**(15, 4)**

**(20, 5)**

**(25, 2)**

Priority Queue After Dequeue:

# (5, 1)

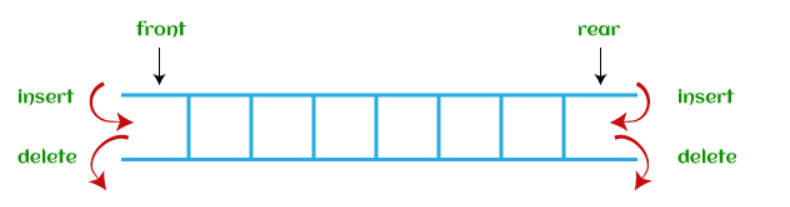
**(25, 2)**

# Deque (Double Ended Queue):

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

* It means that we can insert and delete elements from both **front** and **rear**







ends of the queue.

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

# Operations:



**Enqueue:**

The Enqueue operation is used to insert the element at the rear end of the queue. It returns void.

**Dequeue:**

It performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value.

**Peek:**

This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.



**Queue overflow (isfull):**

It shows the overflow condition when the queue

is completely full.

# *Queue underflow (isempty):

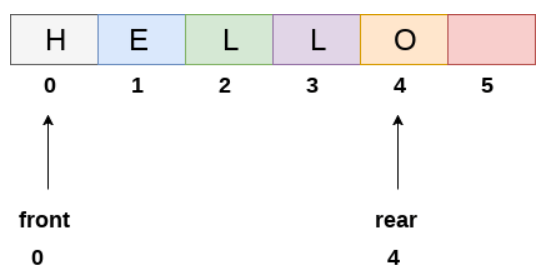
It shows the underflow condition when the

Queue is empty, i.e., no elements are in the Queue.

# implement of queue:

There are two ways of implementing the Queue:

# Implementation using array:

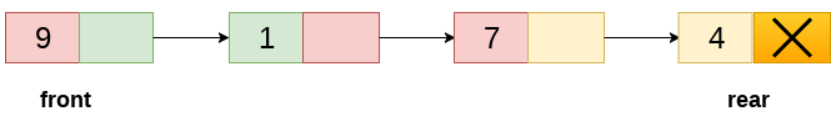


The sequential allocation in a Queue can be implemented using an array.

**Implementation using Linked list:**



The linked list allocation in a Queue can be implemented using a linked list.



# Types of DE Queues:

1. **Input Restricted DE Queue:** in this Dequeue input is restricted at a single end but it allows deletion at both ends.
2. **Output Restricted DE Queue:** in this Dequeue output is restricted at a single end but it allows insertion at both ends.



# Operations on DE Queue:



**Operations performed on deque:**



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

* 1. Insertion at Begin / Insertion at front
  2. Insertion at End / Insertion at rear
  3. Deletion at Begin / Deletion at front
  4. Deletion at End / Deletion at rear

# Insertion at Begin / Insertion at front:

**Explanation:**

int dq[4];

int front =-1; int rear = - 1; int c = 5;





if(front = =0 && rear= =3)

{

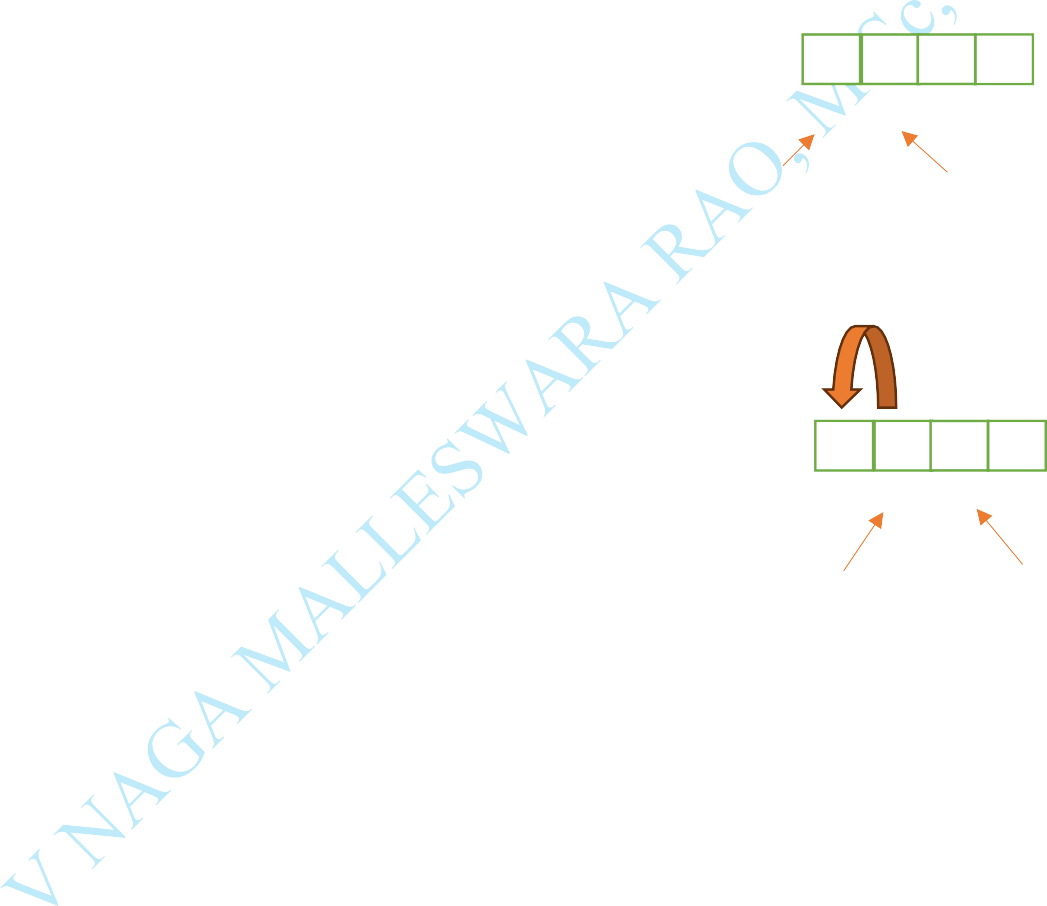
|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |

0 1 2 3



printf(“ DE queue is full”); Front Rear

}



if(front = = 0) 1 2

{ 0 1 2 3

printf(“insertion not possible”); Front Rear

}

else Front

{

front= front-1; 1 2

dq[front]=c; 0 1 2 3

# Front Rear

**Insertion at End / Insertion at front: Explanation:**

int dq[10]; int front =-1; int rear = - 1; int c = 5;



if(front = =0 && rear= =3)

{

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |

0 1 2 3



printf(“ DE queue is full”); Front Rear

}

if(front = = 3)

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 1 | 2 |

{ 0 1 2 3

printf(“insertion not possible”); Front Rear

}

else

{

Rear = Rear + 1;

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 5 |

dq[rear]=c; 0 1 2 3 0 1 2 3

# Front Rear Front

**Rear**

# deletion at Begin / Insertion at Rear: Explanation:

int dq[4];

int front =-1; int rear = - 1; int c = 5;



if(front = =0 && rear= =3)

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

{ 0

printf(“ DE queue is Empty”);

}

printf(“delete the element”,dq[front]); if(front = = rear)

|  |  |  |  |
| --- | --- | --- | --- |
| 1 |  |  |  |

{ 0

Front=rear = -1; Front Rear

}

else

{

front = front + 1;

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 |  |

} 0 1 2 3

1 2 3



1 2 3

|  |  |  |  |
| --- | --- | --- | --- |
|  | 2 | 3 |  |

0 1 2

3

# Front Rear Front Rear

**deletion at End / Insertion at Front: Explanation:**

int dq[5];

int front =-1; int rear = - 1; int c = 5;



if(front = =0 && rear= =3)

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

{ 0

printf(“ DE queue is Empty”);

}

printf(“delete the element”,dq[front]); if(front = = rear)

|  |  |  |  |
| --- | --- | --- | --- |
| 1 |  |  |  |

{ 0

front=rear = -1; Front Rear

}

else

{

rear = rear - 1;

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 |  |

} 0 1 2 3

1 2 3



1 2 3

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 |  |  |

0 1 3

2

# Front Rear Front Rear

**Example:**

#include <stdio.h> #define MAX 10

**void** addFront (**int** \*, **int**, **int** \*, **int** \*); **void** addRear (**int** \*, **int**, **int** \*, **int** \*); **int** delFront (**int** \*, **int** \*, **int** \*);



**int** delRear (**int** \*, **int** \*, **int** \*);

**void** display (**int** \*);



**int** count(**int** \*);

**int** main()

{

**int** arr[MAX];

**int** front, **rear**, i, n;

**front** = **rear** = -1;

**for** (i = 0; i < **MAX**; i++)

**arr**[i] = 0;

addRear( **arr**, 10, &front, &rear); addFront (**arr**, 20, &front, &rear); addRear (**arr**, 30, &front, &rear);

addFront (**arr**, 40, &front, &rear); // insert element // addRear (**arr**, 50, &front, &rear);

addFront (**arr**, 60, &front, &rear); printf("\n Elements in a deque: "); display(arr); // display the list of elements i = **delFront** (**arr**, &front, &rear); printf("\n removed item: %d", i);

printf("\n Elements in a deque after deletion: "); display(**arr**);

addRear (**arr**, 100, &**front**, &**rear**); addRear (**arr**, 200, &**front**, &**rear**);

printf("\n Elements in a deque after addition: "); display(**arr**);







i = **delRear** (**arr**, &**front**, &**rear**); printf("\n removed item: %d", i);

printf("\n Elements in a deque after deletion: "); display(**arr**);

**n** = **count**(**arr**); // count number elements //

printf("\n Total number of elements in deque: %d", n);

}

void addFront(**int** \*arr, **int** item, **int** \*pfront, **int** \*prear)

{

**int** i, k, c;

**if** (\***pfront** == 0 **&&** \***prear** == **MAX** - 1)

{

printf("\n Deque is full.\n"); return;

}

**if** (\***pfront** == -1)

{

\*pfront = \*prear = 0; **arr**[\*pfront] = item; return;

}

**if** (\*prear != **MAX** - 1)

{

c = **count**(**arr**); k = \*prear + 1;

**for** (i = 1; i <= c; i++)



{



**arr**[k] = arr[k - 1]; k--;



}

**arr**[k] = **item**;

\***pfront** = k; (\***prear**) ++;

}

else

{

(\*pfront) --;

**arr**[\***pfront**] = **item**;

}

}

void addRear(**int** \***arr**, **int item**, **int** \***pfront**, **int** \***prear**)

{

**int** i, k;

**if** (\***pfront** == 0 **&&** \***prear** == **MAX** - 1)

{

printf("\n Deque is full.\n"); return;

}

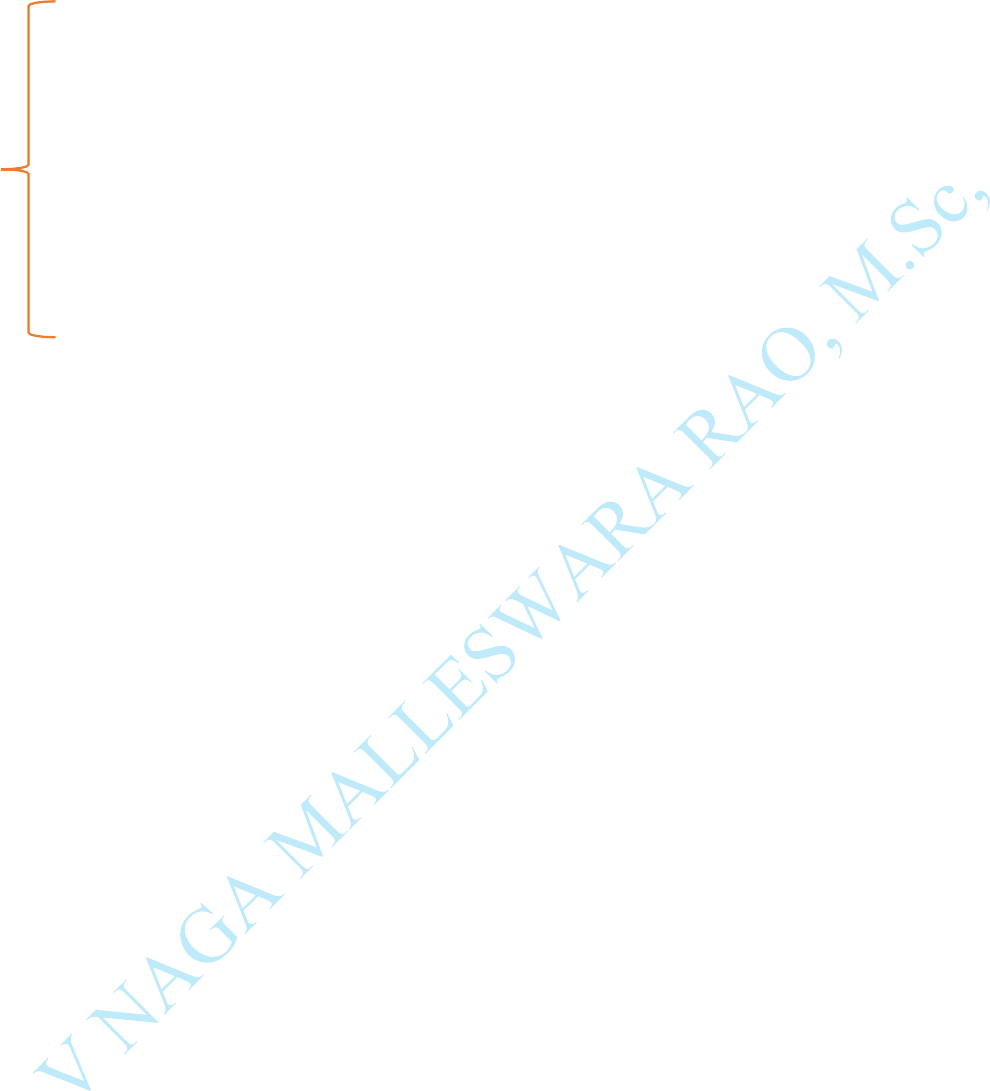
**if** (\***pfront** == -1)

{

\***prear** = \***pfront** = 0; **arr**[\***prear**] = **item**; return;



}



**if** (\***prear** == **MAX** - 1)

{

**k** = \*pfront - 1;

**for** (i = \***pfront** - 1; i < \***prear**; i++)

{

k = i;

**if** (k == **MAX** - 1)

**arr**[k] = 0; else

**arr**[k] = **arr**[i + 1];

}

(\***prear**) --;

(\***pfront**) --;

}

(\***prear**) ++;

**arr**[\***prear**] = **item**;

}

# int delFront(int \*arr, int \*pfront, int \*prear)

{

# int item;

**if** (\***pfront** == -1)

{

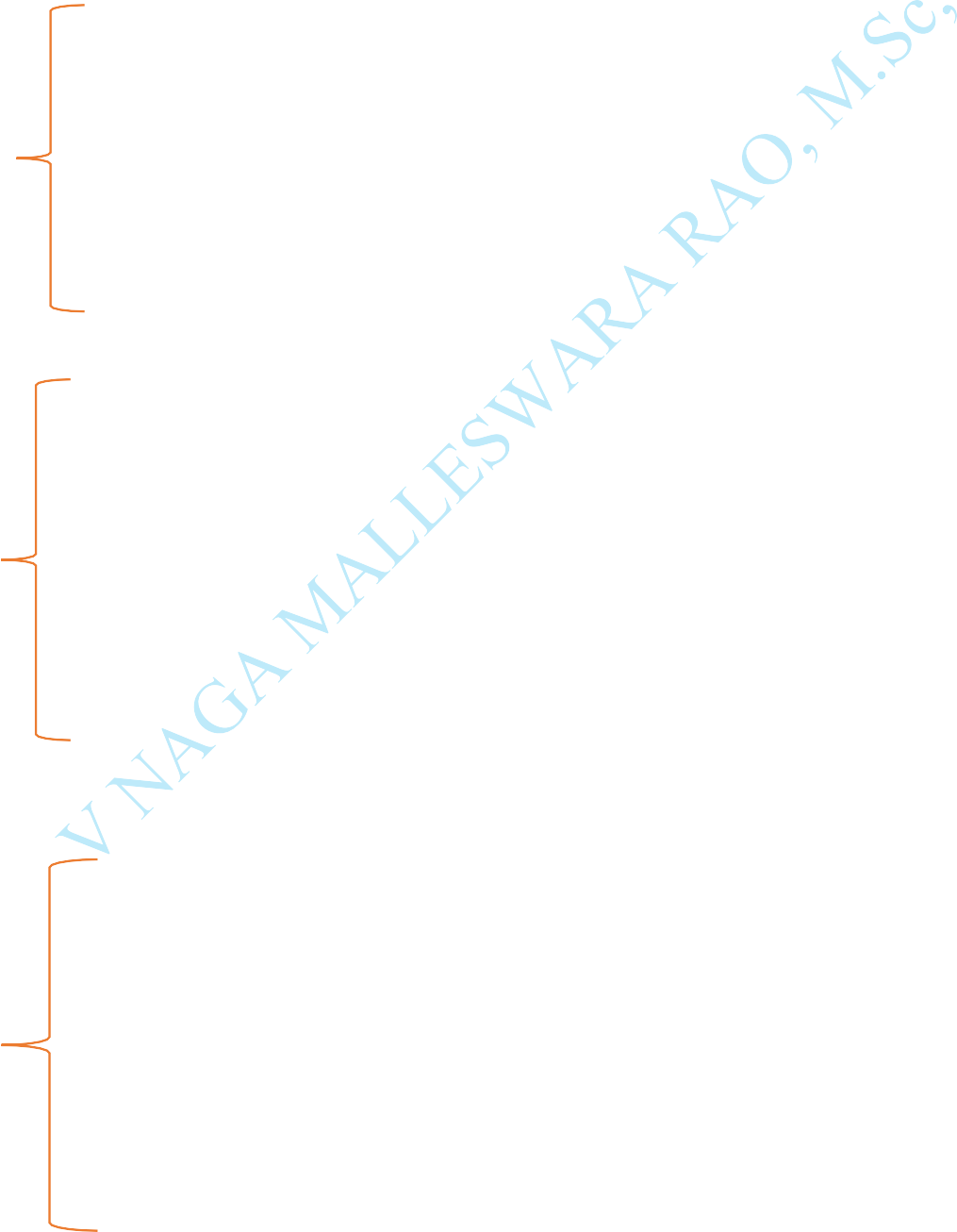
printf("\n Deque is empty.\n"); return 0;



}



item = **arr**[\***pfront**]; **arr**[\***pfront**] = 0;



**if** (\***pfront** == \***prear**)

\***pfront** = \***prear** = -1; else

(\***pfront**)++; return **item**;

}

**int delRear** (**int** \*arr, **int** \*pfront, **int** \*prear)

{

**int** item;

**if** (\***pfront** == -1)

{

printf("\n Deque is empty.\n"); return 0;

}

item = **arr**[\***prear**]; **arr**[\***prear**] = 0; (\***prear**)--;

**if** (\***prear** == -1)

\***pfront** = -1;

return **item**;

}

**void display**(**int** \*arr)

{

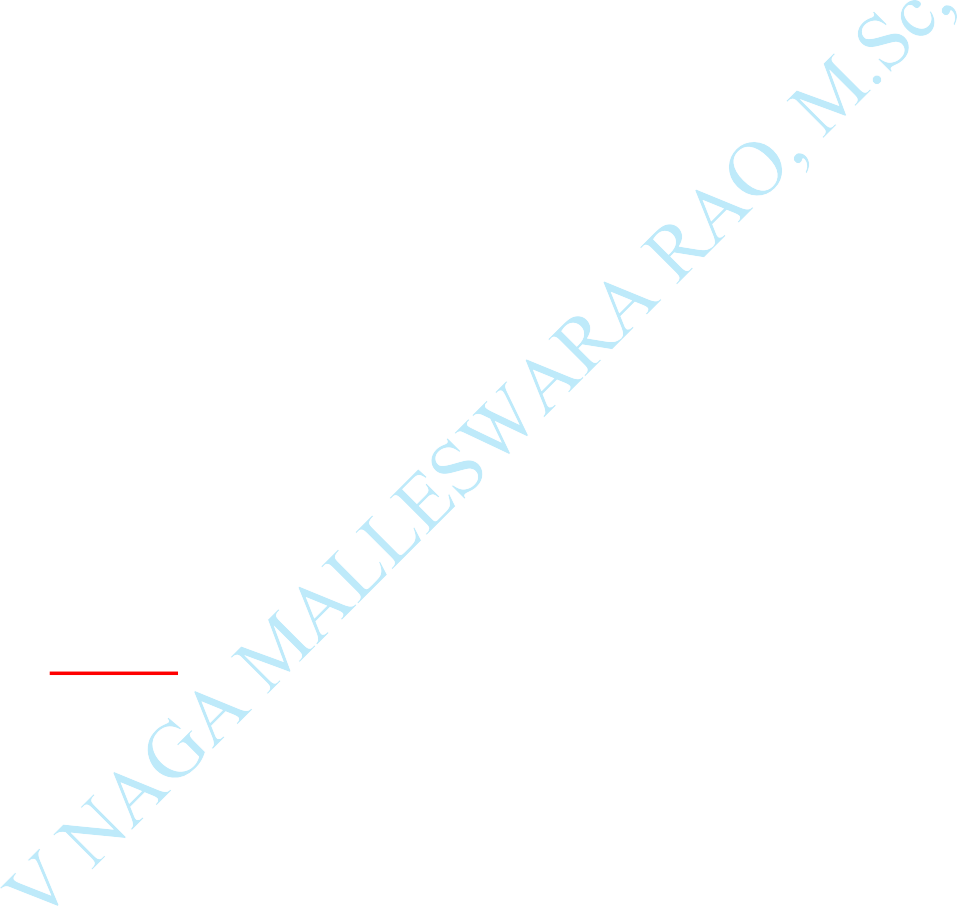
**int** i;

printf("\n **front**: ");

**for** (i = 0; i < **MAX**; i++) printf(" %d", **arr**[i]);

printf(" :**rear**");

}

**int** count (**int** \*arr)

{

**int** c = 0, i;

**for** (i = 0; i < **MAX**; i++)

{

**if** (**arr**[i] != 0) c++;

}

return c;

}

# Output:

Elements in a deque:

front: 60 40 20 10 30 50 0 0 0 0 :rear

removed item: 60

Elements in a deque after deletion:

front: 0 40 20 10 30 50 0 0 0 0 :rear Elements in a deque after addition:

front: 0 40 20 10 30 50 100 200 0 0 :rear

removed item: 200

Elements in a deque after deletion:

front: 0 40 20 10 30 50 100 0 0 0 :rear Total number of elements in deque: 6