Data Structure Lab (ETCS-255).

Faculty Name: Mrs. NEETU GARG

Student Name: NIKHIL MATHUR

Roll No.: 05214802719

Semester: 3rd

Group: 3C2

Maharaja Agrasen Institute of Technology, PSP Area,

Sector – 22, Rohini, New Delhi – 110085

Index

EXP. NO	EXPERIMENT NAME	DATE OF PERFORMANCE	DATE OF CHECKING	MARKS	SIGNATURE
1	Write a program to perform linear and binary search on array				
2	Write a program to perform array traversl, insertion,deletion.				
3	To perform push pop and display operations on stack using array.				
4	To implement sparse matrix using array				
5	To Create a linked list and perform following operations: I. Insert a new node at specified position. II. Delete of a node with a specified position				
6	Create a Doubly linked list and perform following operations:				

	Insertion and Deletion operations.				

EXP. NO	EXPERIMENT NAME	DATE OF PERFORMANCE	DATE OF CHECKING	MARKS	SIGNATURE
7	To evaluate a postfix expression by using an array.				
8	To implement a circular queue to perform functions like addition, deletion.				
9	To implement stack using linked list				
10	To implement queue using linked list				
11	To implement binary tree preorder , postorder , inorder.				
12	To implement insertion and deletion in binary tree				

Data Structure Lab
(ETCS-255)

Name: NIKHIL MATHUR

Roll No.: 05214802719

GROUP : 3C2

EXPERIMENT-1.

AIM: Perform Linear Search and Binary Search on an array. Description of programs:

a. Read an array of type integer.

b. Input element from user for searching.

c. Search the element by passing the array to a function and then returning the position of the element from the function else return -1 if the element is not found.

d. Display the position where the element has been found.

1. LINEAR SEARCH.

ALGORITHM:

Step 1: Select the **first element** as the **current element**.

Step 2: Compare the **current element** with the **target element**. If matches, then go to *step 5*.

Step 3: If there is a **next element**, then set **current element** to **next element** and go to *Step 2*.

Step 4: **Target element** not found. Go to *Step 6*.

Step 5: **Target element** found and return **location**.

Step 6: Exit process.

CODE:

```c
#include <stdio.h>

void main()
{  int num;

    int i,  keynum, found = 0;

    printf("Enter the number of elements ");
    scanf("%d", &num);
    int array[num];
    printf("Enter the elements one by one \n");
    for (i = 0; i < num; i++)
    {
        scanf("%d", &array[i]);
    }

    printf("Enter the element to be searched ");
    scanf("%d", &keynum);
    /*  Linear search begins */
    for (i = 0; i < num ; i++)
    {
        if (keynum == array[i] )
        {
            found = 1;
            break;
        }
    }
    if (found == 1)
        printf("Element is present in the array at position %d",i+1);
    else
        printf("Element is not present in the array\n");
}
```

OUTPUT:

```
Enter the number of elements 4
Enter the elements one by one
1 3 5 7
Enter the element to be searched 5
Element is present in the array at position 3

...Program finished with exit code 0
Press ENTER to exit console.
```

2. BINARY SEARCH.

ALGORITHM:

Step 1: Find middle element of the array.

Step 2: Compare the value of the middle element with the target value.

Step 3: If they match, it is returned.

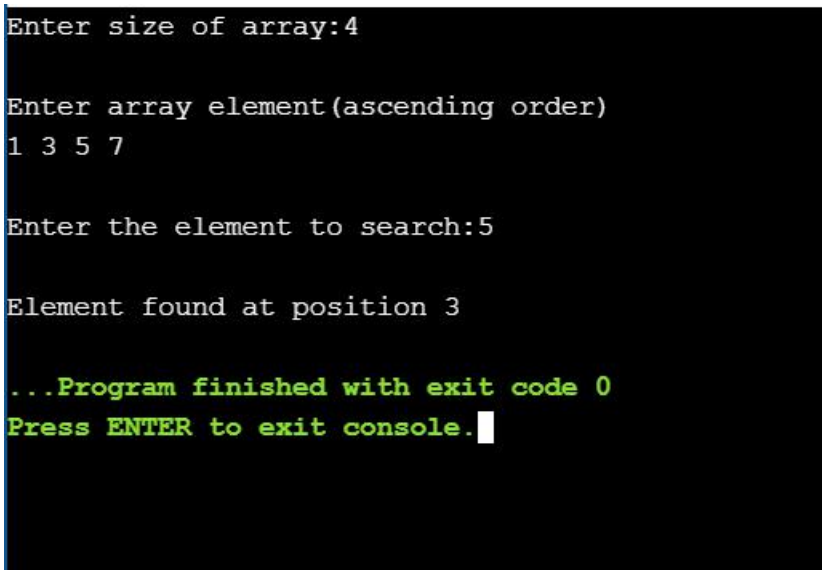
Step 4: If the value is less or greater than the target, the search continues in the lower or upper half of the array accordingly.

Step 5: The same procedure as in *step 2-4* continues, but with a smaller part of the array. This continues until the target element is found or until there are no elements left.

CODE:

```c
#include<stdio.h>
 int main()
 {
    int arr[50],i,n,x,flag=0,first,last,mid;
     printf("Enter size of array:");
    scanf("%d",&n);
    printf("\nEnter array element(ascending order)\n");
     for(i=0;i<n;++i)
        scanf("%d",&arr[i]);

    printf("\nEnter the element to search:");
    scanf("%d",&x);
     first=0;
    last=n-1;
     while(first<=last)
    {
        mid=(first+last)/2;

        if(x==arr[mid]){
            flag=1;
            break;
        }
        else
            if(x>arr[mid])
                first=mid+1;
            else
                last=mid-1;
    }
     if(flag==1)
        printf("\nElement found at position %d",mid+1);
    else
        printf("\nElement not found");
     return 0;
}
```

OUTPUT:



Viva Questions:

Q1. The sequential search, also known as ?

A1. Linear Search.

Q2. What is the primary requirement for implementation of binary search in an array?

A2. The array must be sorted first to be able to apply binary search.

Q3. Is binary search applicable on array and linked list both?

A3. Yes, Binary search is possible on the linked list if the list is ordered and you know the count of elements in list. But While sorting the list, you can access a single element at a time through a pointer to that node i.e. either a previous node or next node. This increases the traversal steps per element in linked list just to find the middle element. This makes it slow and inefficient.

Whereas the binary search on an array is fast and efficient because of its ability to access any element in the array in constant time. You can get to the middle of the array just by saying "array[middle]"! Now, can you do the same with a linked list? The answer is No. You will have to write your own, possibly

inefficient algorithm to get the value of the middle node of a linked list. In a linked list, you loosse the ability to get the value of any node in a constant time.

Q4. What is the principle of working of Binary search?

A4. Binary search works on sorted arrays. Binary search begins by comparing an element in the middle of the array with the target value. If the target value matches the element, its position in the array is returned. If the target value is less than the element, the search continues in the lower half of the array.

Q5. Which searching algorithm is efficient one?

A5. Binary Search is a more efficient search algorithm which relies on the elements in the list being sorted. We apply the same search process to progressively smaller sub-lists of the original list, starting with the whole list and approximately halving the search area every time.

Q6. Under what circumstances, binary search cannot be applied to a list of elements?

A6. Binary Search cannot be applied to :

1. Unsorted Arrays.
2. Descending order Arrays.
3. If elements in the string are not in dictionary order.

Data Structure Lab
(ETCS-255)

Name: NIKHIL MATHUR

Roll No.: 05214802719

GROUP : 3C2

EXPERIMENT-2.

AIM: Implement sparse matrix using array. Description of program:

a. Read a 2D array from the user.

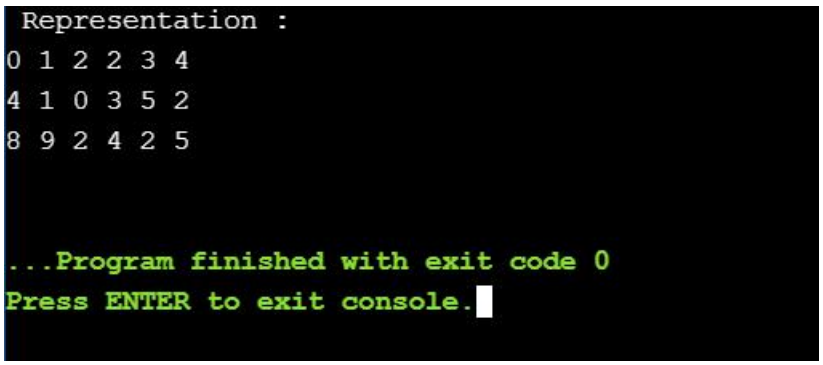
b. b. Store it in the sparse matrix form, use array of structures.

c. c. Print the final array.

CODE:

```cpp
#include<iostream>
using namespace std;
int main()
{

    int sparseMatrix[5][6] =
    {
        {0 , 0 , 0 , 0 , 8, 0 },
        {0 , 9 , 0 , 0 , 0, 0 },
        {2 , 0 , 0 , 4 , 0, 0 },
        {0 , 0 , 0 , 0 , 0, 2 },
        {0 , 0 , 5 , 0 , 0, 0 }
    };

    int size = 0;
    for (int row = 0; row < 5; row++)
        for (int column = 0; column < 6; column++)
            if (sparseMatrix[row][column] != 0)
                size++;

    int resultMatrix[3][size];

    int k = 0;
    for (int row = 0; row < 5; row++)
        for (int column = 0; column < 6; column++)
            if (sparseMatrix[row][column] != 0)
            {
                resultMatrix[0][k] = row;
                resultMatrix[1][k] = column;
                resultMatrix[2][k] = sparseMatrix[row][column];
                k++;
            }

    cout<<"Triplet Representation : "<<endl;
    for (int row=0; row<3; row++)
    {
        for (int column = 0; column<size; column++)
            cout<<resultMatrix[row][column]<<" ";

        cout<<endl;
    }
    return 0;
}
```

OUTPUT:



VIVA QUESTIONS:

Q1. What is sparse matrix?

A1. A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have 0 value, then it is called a sparse matrix .

Q2. What are the different ways of representing sparse matrix in memory?

A2. Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation
2. Linked list representation

Q3. What is the advantage of representing only non-zero values in sparse matrix?

A3. The sparse attribute allows MATLAB to:

Store only the nonzero elements of the matrix, together with their indices.

Reduce computation time by eliminating operations on zero elements.

Q4. Which data structure is used to implement a matrix?

A4. Arrays are used to implement mathematical vectors and matrices, as well as other kinds of rectangular tables. Many databases, small and large, consist of (or include) one-dimensional arrays whose elements are records.

Q5. What are various types of representation of matrix?

A5. Matrix representation is a method used by a computer language to store matrices of more than one dimension in memory. Fortran and C use different schemes for their native arrays. C uses "Row Major", which stores all the elements for a given row contiguously in memory. LAPACK defines various matrix representations in memory. There is also Sparse matrix representation and Morton-order matrix representation. According to the documentation, in LAPACK the unitary matrix representation is optimized.

Q6. How much space does a tria-diagonal sparse matrix take when it is stored in an array?

A6. A tridiagonal matrix has elements only on the main diagonal, the first superdiagonal, and the first subdiagonal. It is stored using three 1-dimensional arrays.

$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & a_{43} & a_{44} \end{bmatrix}$	$\begin{bmatrix} a_{21} \\ a_{32} \\ a_{43} \end{bmatrix}$ $\begin{bmatrix} a_{11} \\ a_{22} \\ a_{33} \\ a_{44} \end{bmatrix}$ $\begin{bmatrix} a_{12} \\ a_{23} \\ a_{34} \end{bmatrix}$
Tridiagonal Matrix	Tridiagonal Array in Tridiagonal Storage

Data Structure Lab
(ETCS-255)

Name: NIKHIL MATHUR

Roll No.: 05214802719

GROUP : 3C2

EXPERIMENT 3.

AIM: Create a circular linked list having information about a college and perform Insertion at front and perform Deletion at end.

ALGORITHM:

Operations

In a circular linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

Step 1 - Include all the header files which are used in the program.
Step 2 - Declare all the user defined functions.
Step 3 - Define a Node structure with two members data and next
Step 4 - Define a Node pointer 'head' and set it to NULL.
Step 5 - Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the circular linked list...

- Step 1 - Create a newNode with given value.
- Step 2 - Check whether list is Empty (head == NULL)
- Step 3 - If it is Empty then, set head = newNode and newNode → next = head .
- Step 4 - If it is Not Empty then, define a Node pointer 'temp' and initialize with 'head'.
- Step 5 - Keep moving the 'temp' to its next node until it reaches to the last node (until 'temp → next == head').
- Step 6 - Set 'newNode → next =head', 'head = newNode' and 'temp → next = head'.

Inserting At End of the list

We can use the following steps to insert a new node at end of the circular linked list...

- Step 1 - Create a newNode with given value.
- Step 2 - Check whether list is Empty (head == NULL).
- Step 3 - If it is Empty then, set head = newNode and newNode → next = head.
- Step 4 - If it is Not Empty then, define a node pointer temp and initialize with head.
- Step 5 - Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next == head).
- Step 6 - Set temp → next = newNode and newNode → next = head.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the circular linked list...

- Step 1 - Create a newNode with given value.
- Step 2 - Check whether list is Empty (head == NULL)
- Step 3 - If it is Empty then, set head = newNode and newNode → next = head.
- Step 4 - If it is Not Empty then, define a node pointer temp and initialize with head.
- Step 5 - Keep moving the temp to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is

equal to location, here location is the node value after which we want to insert the newNode).

- Step 6 - Every time check whether temp is reached to the last node or not. If it is reached to last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp to next node.
- Step 7 - If temp is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).
- Step 8 - If temp is last node then set temp → next = newNode and newNode → next = head.
- Step 8 - If temp is not last node then set newNode → next = temp → next and temp → next = newNode.

Deletion

In a circular linked list, the deletion operation can be performed in three ways those are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the circular linked list...

- Step 1 - Check whether list is Empty (head == NULL)
- Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
- Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize both 'temp1' and 'temp2' with head.
- Step 4 - Check whether list is having only one node (temp1 → next == head)
- Step 5 - If it is TRUE then set head = NULL and delete temp1 (Setting Empty list conditions)
- Step 6 - If it is FALSE move the temp1 until it reaches to the last node. (until temp1 → next == head)

- Step 7 - Then set head = temp2 → next, temp1 → next = head and delete temp2.

Deleting from End of the list

We can use the following steps to delete a node from end of the circular linked list...

- Step 1 - Check whether list is Empty (head == NULL)
- Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
- Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.
- Step 4 - Check whether list has only one Node (temp1 → next == head)
- Step 5 - If it is TRUE. Then, set head = NULL and delete temp1. And terminate from the function. (Setting Empty list condition)
- Step 6 - If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until temp1 reaches to the last node in the list. (until temp1 → next == head)
- Step 7 - Set temp2 → next = head and delete temp1.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the circular linked list...

- Step 1 - Check whether list is Empty (head == NULL)
- Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
- Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.
- Step 4 - Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.
- Step 5 - If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.
- Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node (temp1 → next == head)
- Step 7 - If list has only one node and that is the node to be deleted then set head = NULL and delete temp1 (free(temp1)).

- Step 8 - If list contains multiple nodes then check whether temp1 is the first node in the list (temp1 == head).
- Step 9 - If temp1 is the first node then set temp2 = head and keep moving temp2 to its next node until temp2 reaches to the last node. Then set head = head → next, temp2 → next = head and delete temp1.
- Step 10 - If temp1 is not first node then check whether it is last node in the list (temp1 → next == head).
- Step 1 1- If temp1 is last node then set temp2 → next = head and delete temp1 (free(temp1)).
- Step 12 - If temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free(temp1)).

Displaying a circular Linked List

We can use the following steps to display the elements of a circular linked list...

- Step 1 - Check whether list is Empty (head == NULL)
- Step 2 - If it is Empty, then display 'List is Empty!!!' and terminate the function.
- Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
- Step 4 - Keep displaying temp → data with an arrow (--->) until temp reaches to the last node
- Step 5 - Finally display temp → data with arrow pointing to head → data.

CODE:

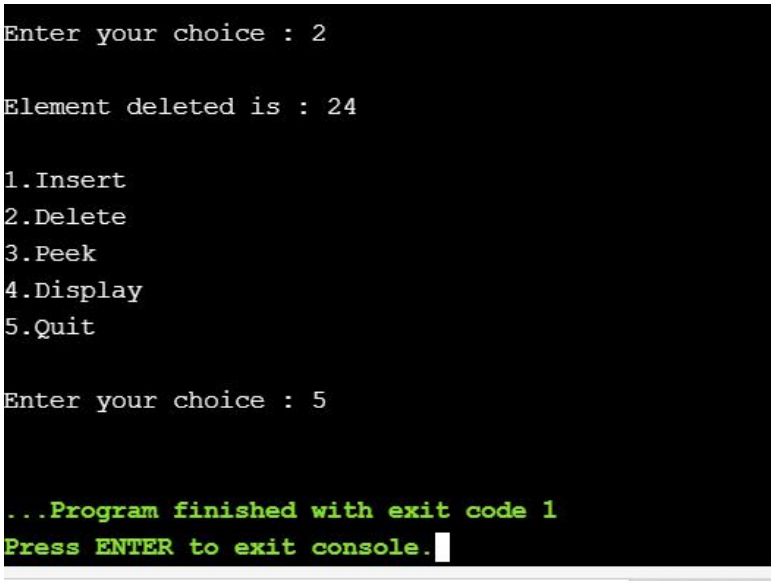
```c
#include<stdio.h>
#include<stdlib.h>
#define MAX 10

int cqueue_arr[MAX];
int front=-1;
int rear=-1;

void display( );
void insert(int item);
int del();
int peek();
int isEmpty();
int isFull();

int main()
{
        int choice,item;
        while(1)
        {
                printf("\n1.Insert\n");
                printf("2.Delete\n");
                printf("3.Peek\n");
                printf("4.Display\n");
                printf("5.Quit\n");
                printf("\nEnter your choice : ");
                scanf("%d",&choice);

                switch(choice)
                {
                case 1 :
                        printf("\nInput the element for insertion : ");
                        scanf("%d",&item);
                        insert(item);
                        break;
                case 2 :
                        printf("\nElement deleted is : %d\n",del());
                        break;
                case 3:
                        printf("\nElement at the front is  : %d\n",peek());
                        break;
                case 4:
                        display();
                        break;
                case 5:
                        exit(1);
                default:
                        printf("\nWrong choice\n");
                }
        }
        return 0;

}

void insert(int item)
{
        if( isFull() )
        {
                printf("\nQueue Overflow\n");
                return;
        }
        if(front == -1 )
                front=0;

        if(rear==MAX-1)
                rear=0;
        else
                rear=rear+1;
        cqueue_arr[rear]=item ;
}/*End of insert()*/

int del()
{
        int item;
```

```
        if( isEmpty() )
        {
                printf("\nQueue Underflow\n");
                exit(1);
        }
        item=cqueue_arr[front];
        if(front==rear)
        {
                front=-1;
                rear=-1;
        }
        else if(front==MAX-1)
                front=0;
        else
                front=front+1;
        return item;
}

int isEmpty()
{
        if(front==-1)
                return 1;
        else
                return 0;
}

int isFull()
{
        if((front==0 && rear==MAX-1) || (front==rear+1))
                return 1;
        else
                return 0;
}

int peek()
{
        if( isEmpty() )
        {
                printf("\nQueue Underflow\n");
                exit(1);
        }
        return cqueue_arr[front];
}

void display()
{
        int i;
        if(isEmpty())
        {
                printf("\nQueue is empty\n");
                return;
        }
        printf("\nQueue elements :\n");
        i=front;
        if( front<=rear )
        {
                while(i<=rear)
                        printf("%d ",cqueue_arr[i++]);
        }
        else
        {
                while(i<=MAX-1)
                        printf("%d ",cqueue_arr[i++]);
                i=0;
                while(i<=rear)
                        printf("%d ",cqueue_arr[i++]);
        }
        printf("\n");
}
```

OUTPUT:

```
1.Insert
2.Delete
3.Peek
4.Display
5.Quit

Enter your choice : 1

Input the element for insertion : 24

1.Insert
2.Delete
3.Peek
4.Display
5.Quit

Enter your choice : 1

Input the element for insertion : 26
```

```
Enter your choice : 1

Input the element for insertion : 26

1.Insert
2.Delete
3.Peek
4.Display
5.Quit

Enter your choice : 4

Queue elements :
24 26

1.Insert
2.Delete
3.Peek
4.Display
5.Quit
```


Enter your choice : 2
Element deleted is : 24
1.Insert
2.Delete
3.Peek
4.Display
5.Quit
Enter your choice : 5
...Program finished with exit code 1
Press ENTER to exit console.

Data Structure Lab
(ETCS-255)

Name: NIKHIL MATHUR

Roll No.: 05214802719

GROUP : 3C2

EXPERIMENT 4.

AIM: Create a stack and perform Pop, Push, Traverse operations on the stack using Linear Linked list.

ALGORITHM:

In stack related algorithms TOP initially point 0, index of elements in stack is start from 1, and index of last element is MAX.

INIT_STACK (STACK, TOP)

Algorithm to initialize a stack using array.
TOP points to the top-most element of stack.

1) TOP: = 0;
2) Exit

Push operation is used to insert an element into stack.

PUSH_STACK(STACK,TOP,MAX,ITEM)

Algorithm to push an item into stack.

1) IF TOP = MAX then
Print "Stack is full" ;
Exit;
2) Otherwise
TOP: = TOP + 1; /*increment TOP*/
STACK (TOP):= ITEM;
3) End of IF
4) Exit

Pop operation is used to remove an item from stack, first get the element and then decrease TOP pointer.

POP_STACK(STACK,TOP,ITEM)

Algorithm to pop an element from stack.

1) IF TOP = 0 then
   Print "Stack is empty" ;
   Exit;
2) Otherwise
   ITEM: =STACK (TOP);
   TOP:=TOP – 1;
3) End of IF
4) Exit

IS_FULL(STACK,TOP,MAX,STATUS)

Algorithm to check stack is full or not.
STATUS contains the result status.

1) IF TOP = MAX then
   STATUS:=true;
2) Otherwise
   STATUS:=false;
3) End of IF
4) Exit

IS_EMPTY(STACK,TOP,MAX,STATUS)

Algorithm to check stack is empty or not.
STATUS contains the result status.

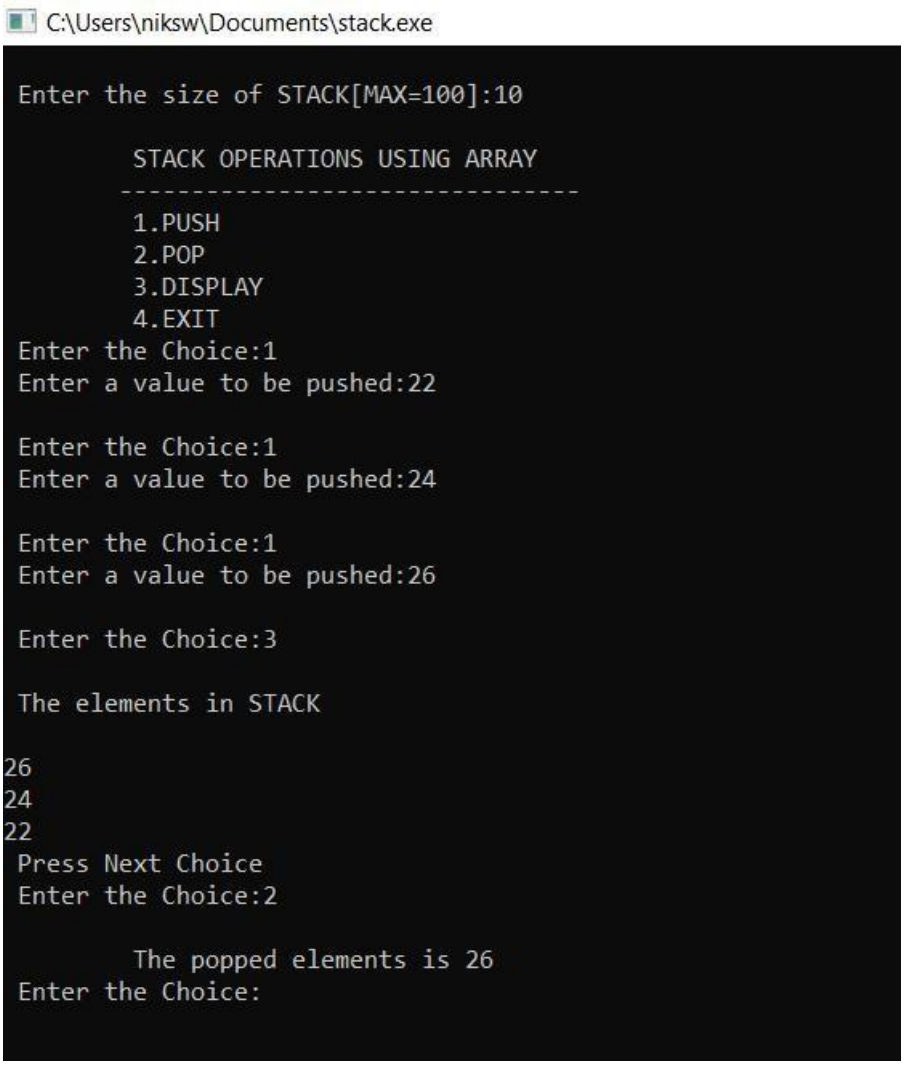
1) IF TOP = 0 then
   STATUS:=true;
2) Otherwise
   STATUS:=false;
3) End of IF
4) Exit

CODE:

```c
#include<stdio.h>
int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{

    top=-1;
    printf("\n Enter the size of STACK[MAX=100]:");
    scanf("%d",&n);
    printf("\n\t STACK OPERATIONS USING ARRAY");
    printf("\n\t--------------------------------");
    printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
    do
    {
        printf("\n Enter the Choice:");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            {
                push();
                break;
            }
            case 2:
            {
                pop();
                break;
            }
            case 3:
            {
                display();
                break;
            }
            case 4:
            {
                printf("\n\t EXIT POINT ");
                break;
            }
            default:
            {
                printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
```

```
44
45          }
46      }
47      while(choice!=4);
48      return 0;
49  }
50  void push()
51  {
52      if(top>=n-1)
53      {
54          printf("\n\tSTACK is over flow");
55
56      }
57      else
58      {
59          printf(" Enter a value to be pushed:");
60          scanf("%d",&x);
61          top++;
62          stack[top]=x;
63      }
64  }
65  void pop()
66  {
67      if(top<=-1)
68      {
69          printf("\n\t Stack is under flow");
70      }
71      else
72      {
73          printf("\n\t The popped elements is %d",stack[top]);
74          top--;
75      }
76  }
77  void display()
78  {
79      if(top>=0)
80      {
81          printf("\n The elements in STACK \n");
82          for(i=top; i>=0; i--)
83              printf("\n%d",stack[i]);
84          printf("\n Press Next Choice");
85      }
86      else
87      {
88          printf("\n The STACK is empty");
89      }
90  }
91
```

OUTPUT:



VIVA QUESTIONS:

Q1. What is the principle of working of stack?

A1. A stack works on the principle of Last In - First Out (LIFO) since removing a plate other than the top one on the stack is not very easy without first removing those plates above it in the stack.

Q2. Give any application of stack.

A2. The stack can be used to convert some infix expression into its postfix equivalent, or prefix equivalent. These postfix or prefix notations are used in computers to express some expressions.

Q3. What are the various operations that can be applied over stack?

A3. Mainly the following three basic operations are performed in the 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. ...

Peek or Top: Returns top element of stack.

isEmpty: Returns true if stack is empty, else false.

Q4. Which type of data structure is used to implement stack?

A4. Stack is alinear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

Q5. Which type of memory allocation does stack uses?

A5. Stack is used for static memory allocation.

Data Structure Lab
(ETCS-255)

Name: NIKHIL MATHUR

Roll No.: 05214802719

GROUP : 3C2

EXPERIMENT 5.

AIM: Design, develop and execute a program in C to evaluate a valid postfix expression using stack. Assume that the postfix expression is read as a single line consisting of non-negative single digit operands and binary arithmetic operators. The operators are +(add), -(subtract), *(multiply), /(divide).

ALGORITHM:

1) Create a stack to store operands (or values).
2) Scan the given expression and do following for every scanned element.
   a. If the element is a number, push it into the stack
   b. If the element is a operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack
3) When the expression is ended, the number in the stack is the final answer

CODE:

```
#include <iostream>
#include <string.h>

using namespace std;

struct Stack
{
    int top;
    unsigned capacity;
    int* array;
};

struct Stack* createStack( unsigned capacity )
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));

    if (!stack) return NULL;

    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (int*) malloc(stack->capacity * sizeof(int));

    if (!stack->array) return NULL;

    return stack;
}

int isEmpty(struct Stack* stack)
```

```
{
    return stack->top == -1 ;
}

char peek(struct Stack* stack)
{
    return stack->array[stack->top];
}

char pop(struct Stack* stack)
{
    if (!isEmpty(stack))
        return stack->array[stack->top--] ;
    return '$';
}

void push(struct Stack* stack, char op)
{
    stack->array[++stack->top] = op;
}


int evaluatePostfix(char* exp)
{
    struct Stack* stack = createStack(strlen(exp));
    int i;

    for (i = 0; exp[i]; ++i)
    {
        if (isdigit(exp[i]))
            push(stack, exp[i] - '0');

        else
        {
            int val1 = pop(stack);
            int val2 = pop(stack);
            switch (exp[i])
            {
            case '+': push(stack, val2 + val1); break;
            case '-': push(stack, val2 - val1); break;
            case '*': push(stack, val2 * val1); break;
            case '/': push(stack, val2/val1); break;
            }
        }
    }
    return pop(stack);
}

int main()
{
    char exp[] = "53+62/*35*+";
    cout<<"postfix evaluation: "<< evaluatePostfix(exp);
    return 0;
}
```

OUTPUT:

```
postfix evaluation: 39

...Program finished with exit code 0
Press ENTER to exit console.
```

Data Structure Lab
(ETCS-255)

Name: NIKHIL MATHUR

Roll No.: 05214802719

GROUP : 3C2

EXPERIMENT 6.

AIM: Implement a Singly Linked List in C.

ALGORITHM:

Operations on Single Linked List

The following operations are performed on a Single Linked List

Insertion
Deletion
Display

Before we implement actual operations, first we need to set up an empty list. First, perform the following steps before implementing actual operations.

- Step 1 - Include all the header files which are used in the program.
- Step 2 - Declare all the user defined functions.
- Step 3 - Define a Node structure with two members data and next
- Step 4 - Define a Node pointer 'head' and set it to NULL.
- Step 5 - Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **newNode → next** = **NULL** and **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, set **newNode → next** = **head** and **head** = **newNode**.

Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode → next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**).
- **Step 3 -** If it is **Empty** then, set **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
- **Step 6 -** Set **temp → next** = **newNode**.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the single linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **newNode → next** = **NULL** and **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is

equal to **location**, here location is the node value after which we want to insert the newNode).

- **Step 6 -** Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7 -** Finally, Set '**newNode → next** = **temp → next**' and '**temp → next** = **newNode**'

Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list is having only one node (**temp → next** == **NULL**)
- **Step 5 -** If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE** then set **head** = **temp → next**, and delete **temp**.

Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

- Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
- Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.
- Step 4 - Check whether list has only one Node (temp1 → next == NULL)
- Step 5 - If it is TRUE. Then, set head = NULL and delete temp1. And terminate the function. (Setting Empty list condition)
- Step 6 - If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until it reaches to the last node in the list. (until temp1 → next == NULL)
- Step 7 - Finally, Set temp2 → next = NULL and delete temp1.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked list...

- Step 1 - Check whether list is Empty (head == NULL)
- Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
- Step 3 - If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.
- Step 4 - Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.
- Step 5 - If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.
- Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- Step 7 - If list has only one node and that is the node to be deleted, then set head = NULL and delete temp1 (free(temp1)).
- Step 8 - If list contains multiple nodes, then check whether temp1 is the first node in the list (temp1 == head).
- Step 9 - If temp1 is the first node then move the head to the next node (head = head → next) and delete temp1.
- Step 10 - If temp1 is not first node then check whether it is last node in the list (temp1 → next == NULL).

- Step 11 - If temp1 is last node then set temp2 → next = NULL and delete temp1 (free(temp1)).
- Step 12 - If temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free(temp1)).

Displaying a Single Linked List

We can use the following steps to display the elements of a single linked list...

- Step 1 - Check whether list is Empty (head == NULL)
- Step 2 - If it is Empty then, display 'List is Empty!!!' and terminate the function.
- Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
- Step 4 - Keep displaying temp → data with an arrow (--->) until temp reaches to the last node
- Step 5 - Finally display temp → data with arrow pointing to NULL (temp → data ---> NULL).

CODE:

```c
#include<stdio.h>
#include<conio.h>

struct node
{
    int data;
    struct node *next;
}*start=NULL,*q,*t;

int main()
{
    int ch;
    void insert_beg();
    void insert_end();
    int insert_pos();
    void display();
    void delete_beg();
    void delete_end();
    int delete_pos();

    while(1)
    {
        printf("\n\n---- Singly Linked List(SLL) Menu ----");
        printf("\n1.Insert\n2.Display\n3.Delete\n4.Exit\n\n");
        printf("Enter your choice(1-4):");
        scanf("%d",&ch);

        switch(ch)
        {
            case 1:
                    printf("\n---- Insert Menu ----");
                    printf("\n1.Insert at beginning\n2.Insert at end\n3.Insert at specified position\n4.Exit");
                    printf("\n\nEnter your choice(1-4):");
                    scanf("%d",&ch);

                    switch(ch)
                    {
                        case 1: insert_beg();
                                break;
                        case 2: insert_end();
                                break;
                        case 3: insert_pos();
                                break;
                        case 4: exit(0);
                        default: printf("Wrong Choice!!");
                    }
                    break;

            case 2: display();
                    break;

            case 3: printf("\n---- Delete Menu ----");
                    printf("\n1.Delete from beginning\n2.Delete from end\n3.Delete from specified position\n4.Exit");
                    printf("\n\nEnter your choice(1-4):");
                    scanf("%d",&ch);

                    switch(ch)
                    {
                        case 1: delete_beg();
                                break;
                        case 2: delete_end();
                                break;
                        case 3: delete_pos();
                                break;
                        case 4: exit(0);
                        default: printf("Wrong Choice!!");
                    }
                    break;
            case 4: exit(0);
                    default: printf("Wrong Choice!!");
        }
    }
    return 0;
}
```

```c

void insert_beg()
{
    int num;
    t=(struct node*)malloc(sizeof(struct node));
    printf("Enter data:");
    scanf("%d",&num);
    t->data=num;

    if(start==NULL)        //If list is empty
    {
        t->next=NULL;
        start=t;
    }
    else
    {
        t->next=start;
        start=t;
    }
}

void insert_end()
{
    int num;
    t=(struct node*)malloc(sizeof(struct node));
    printf("Enter data:");
    scanf("%d",&num);
    t->data=num;
    t->next=NULL;

    if(start==NULL)        //If list is empty
    {
        start=t;
    }
    else
    {
        q=start;
        while(q->next!=NULL)
        q=q->next;
        q->next=t;
    }
}

int insert_pos()
{
    int pos,i,num;
    if(start==NULL)
    {
        printf("List is empty!!");
        return 0;
    }

    t=(struct node*)malloc(sizeof(struct node));
    printf("Enter data:");
    scanf("%d",&num);
    printf("Enter position to insert:");
    scanf("%d",&pos);
    t->data=num;

    q=start;
    for(i=1;i<pos-1;i++)
    {
        if(q->next==NULL)
        {
            printf("There are less elements!!");
            return 0;
        }

        q=q->next;
    }

    t->next=q->next;
    q->next=t;
    return 0;
```

```
}

void display()
{
    if(start==NULL)
    {
        printf("List is empty!!");
    }
    else
    {
        q=start;
        printf("The linked list is:\n");
        while(q!=NULL)
        {
            printf("%d->",q->data);
            q=q->next;
        }
    }
}

void delete_beg()
{
    if(start==NULL)
    {
        printf("The list is empty!!");
    }
    else
    {
        q=start;
        start=start->next;
        printf("Deleted element is %d",q->data);
        free(q);
    }
}

void delete_end()
{
    if(start==NULL)
    {
        printf("The list is empty!!");
    }
    else
    {
        q=start;
        while(q->next->next!=NULL)
        q=q->next;

        t=q->next;
        q->next=NULL;
        printf("Deleted element is %d",t->data);
        free(t);
    }
}

int delete_pos()
{
    int pos,i;

    if(start==NULL)
    {
        printf("List is empty!!");
        return 0;
    }

    printf("Enter position to delete:");
    scanf("%d",&pos);

    q=start;
    for(i=1;i<pos-1;i++)
    {
        if(q->next==NULL)
        {
            printf("There are less elements!!");
            return 0;
```

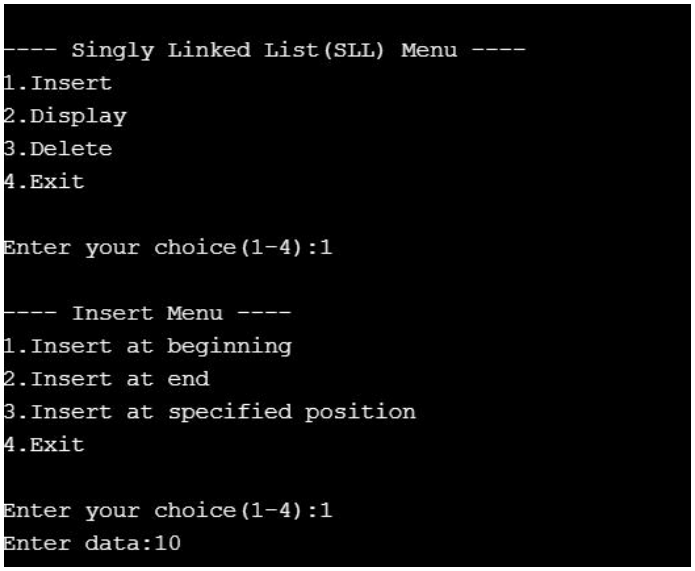
```
223         }
224         q=q->next;
225     }
226 
227     t=q->next;
228     q->next=t->next;
229     printf("Deleted element is %d",t->data);
230     free(t);
231 
232     return 0;
233 }
```

OUTPUT:

```
---- Singly Linked List(SLL) Menu ----
1.Insert
2.Display
3.Delete
4.Exit

Enter your choice(1-4):3

---- Delete Menu ----
1.Delete from beginning
2.Delete from end
3.Delete from specified position
4.Exit

Enter your choice(1-4):3
Enter position to delete:2
Deleted element is 20

---- Singly Linked List(SLL) Menu ----
1.Insert
2.Display
3.Delete
4.Exit

Enter your choice(1-4):2
The linked list is:
10->30->
```

```
---- Singly Linked List(SLL) Menu ----
1.Insert
2.Display
3.Delete
4.Exit

Enter your choice(1-4):1

---- Insert Menu ----
1.Insert at beginning
2.Insert at end
3.Insert at specified position
4.Exit

Enter your choice(1-4):3
Enter data:20
Enter position to insert:2


---- Singly Linked List(SLL) Menu ----
1.Insert
2.Display
3.Delete
4.Exit

Enter your choice(1-4):2
The linked list is:
10->20->30->
```

```
---- Singly Linked List(SLL) Menu ----
1.Insert
2.Display
3.Delete
4.Exit

Enter your choice(1-4):1

---- Insert Menu ----
1.Insert at beginning
2.Insert at end
3.Insert at specified position
4.Exit

Enter your choice(1-4):2
Enter data:30


---- Singly Linked List(SLL) Menu ----
1.Insert
2.Display
3.Delete
4.Exit

Enter your choice(1-4):2
The linked list is:
10->30->
```


---- Singly Linked List(SLL) Menu ----
1.Insert
2.Display
3.Delete
4.Exit
Enter your choice(1-4):1
---- Insert Menu ----
1.Insert at beginning
2.Insert at end
3.Insert at specified position
4.Exit
Enter your choice(1-4):1
Enter data:10

Data Structure Lab
(ETCS-255)

Name: NIKHIL MATHUR

Roll No.: 05214802719

GROUP : 3C2

EXPERIMENT 7.

AIM: Implement a Doubly Linked List in C.

ALGORITHM:

Operations on Double Linked List

In a double linked list, we perform the following operations…

1. Insertion
2. Deletion
3. Display

Insertion

In a double linked list, the insertion operation can be performed in three ways as follows…

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list…

> **Step 1 -** Create a **newNode** with given value and **newNode → previous** as **NULL**.
> **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
> **Step 3 -** If it is **Empty** then, assign **NULL** to **newNode → next** and **newNode** to **head**.
> **Step 4 -** If it is **not Empty** then, assign **head** to **newNode → next** and **newNode** to **head**.

Inserting At End of the list

We can use the following steps to insert a new node at end of the double linked list...

- Step 1 - Create a newNode with given value and newNode → next as NULL.
- Step 2 - Check whether list is Empty (head == NULL)
- Step 3 - If it is Empty, then assign NULL to newNode → previous and newNode to head.
- Step 4 - If it is not Empty, then, define a node pointer temp and initialize with head.
- Step 5 - Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).
- Step 6 - Assign newNode to temp → next and temp to newNode → previous.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the double linked list...

- Step 1 - Create a newNode with given value.
- Step 2 - Check whether list is Empty (head == NULL)
- Step 3 - If it is Empty then, assign NULL to both newNode → previous & newNode → next and set newNode to head.
- Step 4 - If it is not Empty then, define two node pointers temp1 & temp2 and initialize temp1 with head.
- Step 5 - Keep moving the temp1 to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).
- Step 6 - Every time check whether temp1 is reached to the last node. If it is reached to the last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp1 to next node.
- Step 7 - Assign temp1 → next to temp2, newNode to temp1 → next, temp1 to newNode → previous, temp2 to newNode → next and newNode to temp2 → previous.

Deletion

In a double linked list, the deletion operation can be performed in three ways as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the double linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list is having only one node (**temp → previous** is equal to **temp → next**)
- **Step 5 -** If it is **TRUE**, then set **head** to **NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE**, then assign **temp → next** to **head**, **NULL** to **head → previous** and delete **temp**.

Deleting from End of the list

We can use the following steps to delete a node from end of the double linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty**, then display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list has only one Node (**temp → previous** and **temp → next** both are **NULL**)

- Step 5 - If it is TRUE, then assign NULL to head and delete temp. And terminate from the function. (Setting Empty list condition)
- Step 6 - If it is FALSE, then keep moving temp until it reaches to the last node in the list. (until temp → next is equal to NULL)
- Step 7 - Assign NULL to temp → previous → next and delete temp.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the double linked list...

- Step 1 - Check whether list is Empty (head == NULL)
- Step 2 - If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
- Step 3 - If it is not Empty, then define a Node pointer 'temp' and initialize with head.
- Step 4 - Keep moving the temp until it reaches to the exact node to be deleted or to the last node.
- Step 5 - If it is reached to the last node, then display 'Given node not found in the list! Deletion not possible!!!' and terminate the fuction.
- Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- Step 7 - If list has only one node and that is the node which is to be deleted then set head to NULL and delete temp (free(temp)).
- Step 8 - If list contains multiple nodes, then check whether temp is the first node in the list (temp == head).
- Step 9 - If temp is the first node, then move the head to the next node (head = head → next), set head of previous to NULL (head → previous = NULL) and delete temp.
- Step 10 - If temp is not the first node, then check whether it is the last node in the list (temp → next == NULL).
- Step 11 - If temp is the last node then set temp of previous of next to NULL (temp → previous → next = NULL) and delete temp (free(temp)).
- Step 12 - If temp is not the first node and not the last node, then set temp of previous of next to temp of next (temp → previous → next

= temp → next), temp of next of previous to temp of previous (temp → next → previous = temp → previous) and delete temp (free(temp)).

Displaying a Double Linked List

We can use the following steps to display the elements of a double linked list...

- Step 1 - Check whether list is Empty (head == NULL)
- Step 2 - If it is Empty, then display 'List is Empty!!!' and terminate the function.
- Step 3 - If it is not Empty, then define a Node pointer 'temp' and initialize with head.
- Step 4 - Display 'NULL <--- '.
- Step 5 - Keep displaying temp → data with an arrow (<===>) until temp reaches to the last node
- Step 6 - Finally, display temp → data with arrow pointing to NULL (temp → data ---> NULL).

CODE:

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

struct node
{
    struct node *prev;
    int n;
    struct node *next;
}*h,*temp,*temp1,*temp2,*temp4;

void insert1();
void insert2();
void insert3();
void traversebeg();
void traverseend(int);
void sort();
void search();
void update();
void delete();

int count = 0;

void main()
{
    int ch;

    h = NULL;
    temp = temp1 = NULL;

    printf("\n 1 - Insert at beginning");
    printf("\n 2 - Insert at end");
    printf("\n 3 - Insert at position i");
    printf("\n 4 - Delete at i");
    printf("\n 5 - Display from beginning");
    printf("\n 6 - Display from end");
    printf("\n 7 - Search for element");
    printf("\n 8 - Sort the list");
    printf("\n 9 - Update an element");
    printf("\n 10 - Exit");

    while (1)
    {
        printf("\n Enter choice : ");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
            insert1();
            break;
        case 2:
            insert2();
            break;
        case 3:
            insert3();
            break;
        case 4:
            delete();
            break;
        case 5:
            traversebeg();
            break;
        case 6:
            temp2 = h;
            if (temp2 == NULL)
                printf("\n Error : List empty to display ");
            else
```

```
            {
                printf("\n Reverse order of linked list is : ");
                traverseend(temp2->n);
            }
            break;
        case 7:
            search();
            break;
        case 8:
            sort();
            break;
        case 9:
            update();
            break;
        case 10:
            exit(0);
        default:
            printf("\n Wrong choice menu");
        }
    }
}

void create()
{
    int data;

    temp =(struct node *)malloc(1*sizeof(struct node));
    temp->prev = NULL;
    temp->next = NULL;
    printf("\n Enter value to node : ");
    scanf("%d", &data);
    temp->n = data;
    count++;
}

void insert1()
{
    if (h == NULL)
    {
        create();
        h = temp;
        temp1 = h;
    }
    else
    {
        create();
        temp->next = h;
        h->prev = temp;
        h = temp;
    }
}

void insert2()
{
    if (h == NULL)
    {
        create();
        h = temp;
        temp1 = h;
    }
    else
    {
        create();
        temp1->next = temp;
        temp->prev = temp1;
        temp1 = temp;
    }
}

void insert3()
{
    int pos, i = 2;

```

```
    printf("\n Enter position to be inserted : ");
    scanf("%d", &pos);
    temp2 = h;

    if ((pos < 1) || (pos >= count + 1))
    {
        printf("\n Position out of range to insert");
        return;
    }
    if ((h == NULL) && (pos != 1))
    {
        printf("\n Empty list cannot insert other than 1st position");
        return;
    }
    if ((h == NULL) && (pos == 1))
    {
        create();
        h = temp;
        temp1 = h;
        return;
    }
    else
    {
        while (i < pos)
        {
            temp2 = temp2->next;
            i++;
        }
        create();
        temp->prev = temp2;
        temp->next = temp2->next;
        temp2->next->prev = temp;
        temp2->next = temp;
    }
}

void delete()
{
    int i = 1, pos;

    printf("\n Enter position to be deleted : ");
    scanf("%d", &pos);
    temp2 = h;

    if ((pos < 1) || (pos >= count + 1))
    {
        printf("\n Error : Position out of range to delete");
        return;
    }
    if (h == NULL)
    {
        printf("\n Error : Empty list no elements to delete");
        return;
    }
    else
    {
        while (i < pos)
        {
            temp2 = temp2->next;
            i++;
        }
        if (i == 1)
        {
            if (temp2->next == NULL)
            {
                printf("Node deleted from list");
                free(temp2);
                temp2 = h = NULL;
                return;
            }
        }
        if (temp2->next == NULL)
        {
```

```
            temp2->prev->next = NULL;
            free(temp2);
            printf("Node deleted from list");
            return;
        }
        temp2->next->prev = temp2->prev;
        if (i != 1)
            temp2->prev->next = temp2->next;
        if (i == 1)
            h = temp2->next;
        printf("\n Node deleted");
        free(temp2);
    }
    count--;
}

void traversebeg()
{
    temp2 = h;

    if (temp2 == NULL)
    {
        printf("List empty to display \n");
        return;
    }
    printf("\n Linked list elements from begining : ");

    while (temp2->next != NULL)
    {
        printf(" %d ", temp2->n);
        temp2 = temp2->next;
    }
    printf(" %d ", temp2->n);
}

void traverseend(int i)
{
    if (temp2 != NULL)
    {
        i = temp2->n;
        temp2 = temp2->next;
        traverseend(i);
        printf(" %d ", i);
    }
}

void search()
{
    int data, count = 0;
    temp2 = h;

    if (temp2 == NULL)
    {
        printf("\n Error : List empty to search for data");
        return;
    }
    printf("\n Enter value to search : ");
    scanf("%d", &data);
    while (temp2 != NULL)
    {
        if (temp2->n == data)
        {
            printf("\n Data found in %d position",count + 1);
            return;
        }
        else
            temp2 = temp2->next;
            count++;
    }
    printf("\n Error : %d not found in list", data);
}

void update()
{
    int data, data1;
```

```
288
289     printf("\n Enter node data to be updated : ");
290     scanf("%d", &data);
291     printf("\n Enter new data : ");
292     scanf("%d", &data1);
293     temp2 = h;
294     if (temp2 == NULL)
295     {
296         printf("\n Error : List empty no node to update");
297         return;
298     }
299     while (temp2 != NULL)
300     {
301         if (temp2->n == data)
302         {
303
304             temp2->n = data1;
305             traversebeg();
306             return;
307         }
308         else
309             temp2 = temp2->next;
310     }
311
312     printf("\n Error : %d not found in list to update", data);
313 }
314
315 void sort()
316 {
317     int i, j, x;
318
319     temp2 = h;
320     temp4 = h;
321
322     if (temp2 == NULL)
323     {
324         printf("\n List empty to sort");
325         return;
326     }
327
328     for (temp2 = h; temp2 != NULL; temp2 = temp2->next)
329     {
330         for (temp4 = temp2->next; temp4 != NULL; temp4 = temp4->next)
331         {
332             if (temp2->n > temp4->n)
333             {
334                 x = temp2->n;
335                 temp2->n = temp4->n;
336                 temp4->n = x;
337             }
338         }
339     }
340     traversebeg();
341 }
```

OUTPUT:

```
1 - Insert at beginning
2 - Insert at end
3 - Insert at position i
4 - Delete at i
5 - Display from beginning
6 - Display from end
7 - Search for element
8 - Sort the list
9 - Update an element
10 - Exit
Enter choice : 1

Enter value to node : 10

Enter choice : 2

Enter value to node : 30

Enter choice : 3

Enter position to be inserted : 2

Enter value to node : 20

Enter choice : 5

Linked list elements from begining :  10  20  30
Enter choice : 4

Enter position to be deleted : 2

Node deleted
Enter choice : 5

Linked list elements from begining :  10  30
Enter choice :
```

Data Structure Lab
(ETCS-255)

Name: NIKHIL MATHUR

Roll No.: 05214802719

GROUP : 3C2

EXPERIMENT 8.

AIM: WAP to implement a circular queue and allow user to perform functions like addition, deletion and display through a menu driven program.

ALGORITHM:

1. Insertion:

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

2. Deletion:

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

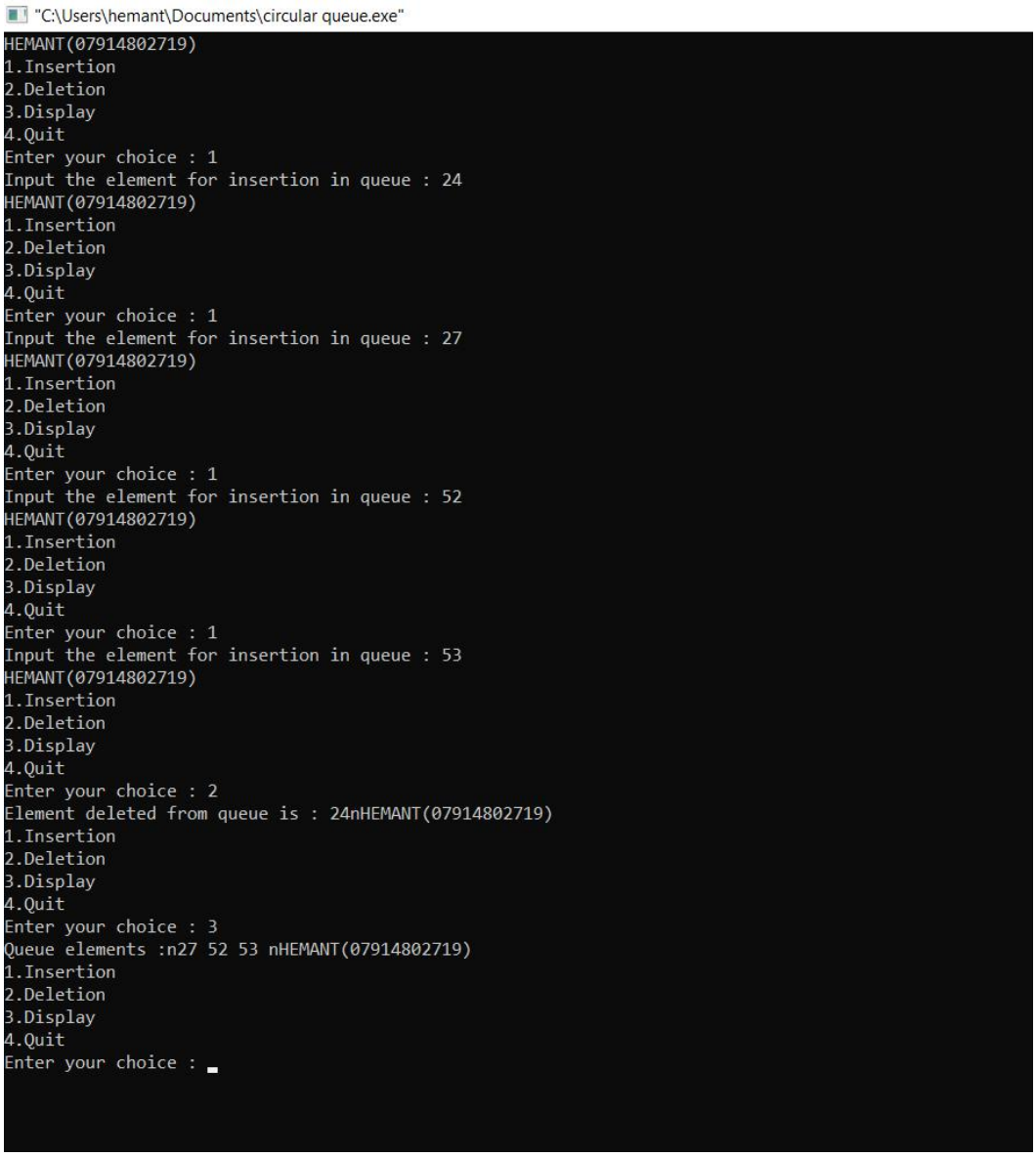
CODE:

```
#include<stdio.h>
# define MAX 5
int cqueue_arr[MAX];
int front = -1;
int rear = -1;
void insert(int item)
{
if((front == 0 && rear == MAX-1) || (front == rear+1))
{
printf("Queue Overflow n");
return;
}
if(front == -1)
{
front = 0;
rear = 0;
}
else
{
if(rear == MAX-1)
rear = 0;
else
rear = rear+1;
}
cqueue_arr[rear] = item ;
}
void deletion()
{
if(front == -1)
{
printf("Queue Underflown");
return ;
}
```

```
printf("Element deleted from queue is : %dn",cqueue_arr[front]);
if(front == rear)
{
front = -1;
rear=-1;
}
else
{
if(front == MAX-1)
front = 0;
else
front = front+1;
}
}
void display()
{
int front_pos = front,rear_pos = rear;
if(front == -1)
{
printf("Queue is emptyn");
return;
}
printf("Queue elements :n");
if( front_pos <= rear_pos )
while(front_pos <= rear_pos)
{
printf("%d ",cqueue_arr[front_pos]);
front_pos++;
}
else
{
while(front_pos <= MAX-1)
{
printf("%d ",cqueue_arr[front_pos])
front_pos++;
}
front_pos = 0;
while(front_pos <= rear_pos)
{
printf("%d ",cqueue_arr[front_pos]);
front_pos++;
}
}
printf("n");
}
int main()
{
int choice,item;
do
{
printf("HEMANT(07914802719)\n1.Insertion\n");
printf("2.Deletion\n");
printf("3.Displayn\n");
printf("4.Quit\n");
```

```
printf("Enter your choice : ");
scanf("%d",&choice);
switch(choice)
{
case 1 :
printf("Input the element for insertion in queue : ");
scanf("%d", &item);
insert(item);
break;
case 2 :
deletion();
break;
case 3:
display();
break;
case 4:
break;
default:
printf("Wrong choicen");
}
}while(choice!=4);
return 0;
}
```

OUTPUT:


"C:\Users\hemant\Documents\circular queue.exe"
HEMANT(07914802719)
1.Insertion
2.Deletion
3.Display
4.Quit
Enter your choice : 1
Input the element for insertion in queue : 24
HEMANT(07914802719)
1.Insertion
2.Deletion
3.Display
4.Quit
Enter your choice : 1
Input the element for insertion in queue : 27
HEMANT(07914802719)
1.Insertion
2.Deletion
3.Display
4.Quit
Enter your choice : 1
Input the element for insertion in queue : 52
HEMANT(07914802719)
1.Insertion
2.Deletion
3.Display
4.Quit
Enter your choice : 1
Input the element for insertion in queue : 53
HEMANT(07914802719)
1.Insertion
2.Deletion
3.Display
4.Quit
Enter your choice : 2
Element deleted from queue is : 24nHEMANT(07914802719)
1.Insertion
2.Deletion
3.Display
4.Quit
Enter your choice : 3
Queue elements :n27 52 53 nHEMANT(07914802719)
1.Insertion
2.Deletion
3.Display
4.Quit
Enter your choice :

Data Structure Lab
(ETCS-255)

Name: NIKHIL MATHUR

Roll No.: 05214802719

GROUP : 3C2

EXPERIMENT 9.

AIM: Implement a queue using singly linked list.

ALGORITHM:

Operations

To implement queue using linked list, we need to set the following things before implementing actual operations.

Step 1 - Include all the header files which are used in the program. And declare all the user defined functions.
Step 2 - Define a 'Node' structure with two members data and next.
Step 3 - Define two Node pointers 'front' and 'rear' and set both to NULL.
Step 4 - Implement the main method by displaying Menu of list of operations and make suitable function calls in the main method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...

- Step 1 - Create a newNode with given value and set 'newNode → next' to NULL.
- Step 2 - Check whether queue is Empty (rear == NULL)
- Step 3 - If it is Empty then, set front = newNode and rear = newNode.
- Step 4 - If it is Not Empty then, set rear → next = newNode and rear = newNode.

deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue...

- Step 1 - Check whether queue is Empty (front == NULL).
- Step 2 - If it is Empty, then display "Queue is Empty!!! Deletion is not possible!!!" and terminate from the function
- Step 3 - If it is Not Empty then, define a Node pointer 'temp' and set it to 'front'.
- Step 4 - Then set 'front = front → next' and delete 'temp' (free(temp)).

display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue...

- Step 1 - Check whether queue is Empty (front == NULL).
- Step 2 - If it is Empty then, display 'Queue is Empty!!!' and terminate the function.
- Step 3 - If it is Not Empty then, define a Node pointer 'temp' and initialize with front.
- Step 4 - Display 'temp → data --->' and move it to the next node. Repeat the same until 'temp' reaches to 'rear' (temp → next != NULL).
- Step 5 - Finally! Display 'temp → data ---> NULL'.

CODE:

```
#include<iostream>
#include<stdlib.h>

using namespace std;

struct node
{
  int data;
  struct node *next;
}*front=NULL,*rear,*temp;

void insert()
{
  temp=new node;
  cout<<"Enter data:";
  cin>>temp->data;
  temp->next=NULL;

  if(front==NULL)
    front=rear=temp;
  else
  {
    rear->next=temp;
    rear=temp;
  }
}

void remove()
```

```
{
  if(front==NULL)
    cout<<"Queue is empty\n";
  else
  {
    temp=front;
    front=front->next;
    cout<<"Deleted node is "<<temp->data<<"\n";
    delete(temp);
  }
}

void display()
{
  if(front==NULL)
    cout<<"Queue is empty\n";
  else
  {
    temp=front;
    while(temp!=NULL)
    {
      cout<<temp->data<<"->";
      temp=temp->next;
    }
  }
}

int main()
{
  int ch;
  while(1)
  {
    cout<<"\n"<<"\n1.Insert\n2.Delete\n3.Display\n4.Exit";
    cout<<"\n\nEnter your choice:";
    cin>>ch;
    cout<<"\n";

    switch(ch)
    {
      case 1: insert();
          break;
      case 2: remove();
          break;
      case 3: display();
          break;
      case 4: exit(0);
          break;
      default: cout<<"Wrong Choice!!!";
    }
  }

  return 0;
}
```

OUTPUT:

```
1.Insert
2.Delete
3.Display
4.Exit

Enter your choice:1
Enter data:10
1.Insert
2.Delete
3.Display
4.Exit

Enter your choice:1

Enter data:20
1.Insert
2.Delete
3.Display
4.Exit

Enter your choice:1
Enter data:30
```

```
1.Insert
2.Delete
3.Display
4.Exit

Enter your choice:3
10->20->30->

1.Insert
2.Delete
3.Display
4.Exit

Enter your choice:2


Deleted node is 10
```

Data Structure Lab
(ETCS-255)

Name: NIKHIL MATHUR

Roll No.: 05214802719

GROUP : 3C2

EXPERIMENT 10.

AIM: Implement insertion, deletion and display (inorder, preorder and postorder) on binary search tree .

ALGORITHM:

Definition: A binary tree is said to be a binary search tree if it is the empty tree or

1. if there is a left-child, then the data in the left-child is less than the data in the root,
2. if there is a right-child, then the data in the right-child is no less than the data in the root, and every sub-tree is a binary search tree. // algorithm to implement insertion operation in BST

Insertion in Binary Search Tree:

Check whether root node is present or not(tree available or not). If root is NULL, create root node.
If the element to be inserted is less than the element present in the root node, traverse the left sub-tree recursively until we reach T->left/T->right is NULL and place the new node at T->left(key in new node < key in T)/T->right (key in new node > key in T).
If the element to be inserted is greater than the element present in root node, traverse the right sub-tree recursively until we reach T->left/T->right is NULL and place the new node at T->left/T->right.

Algorithm for insertion in Binary Search Tree:

```
TreeNode insert(int data, TreeNode T) {
        if T is NULL {
                T = (TreeNode *)malloc(sizeof (Struct TreeNode));
```

```
                    (Allocate Memory of new node and load the data into
 it)
                    T->data = data;
                    T->left    = NULL;
                    T->right = NULL;
            } else if T is less than T->left {
                    T->left = insert(data, T->left);
                    (Then node needs to be inserted in left sub-tree.So,
                      recursively traverse left sub-tree to find the place
                      where the new node needs to be inserted)
            } else if T is greater than T->right {
                     T->right = insert(data, T->right);
                     (Then node needs to be inserted in right sub-tree
                      So, recursively traverse right sub-tree to find the
                       place where the new node needs to be inserted.)
           }
           return T;
}
// algorithm for implementing the delete operation in BST
```

Delete operation on binary search tree is more complicated, then add and search. Basically, in can be divided into two stages:

- search for a node to delete;
- if the node is found, run delete algorithm.

Delete algorithm in detail:

Now, let's see more detailed description of a remove algorithm. First stage is identical to algorithm for lookup, except we should track the parent of the current node. Second part is trickier. There are three cases, which are described below.

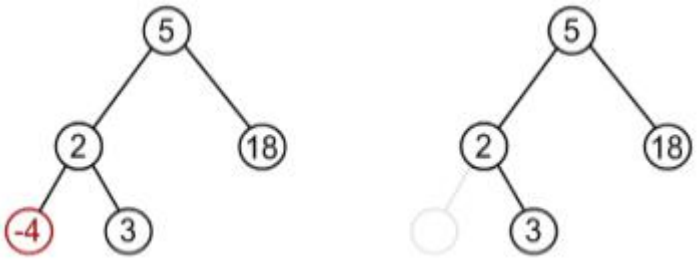
1.

Node to be deleted has no children.

2.

This case is quite simple. Algorithm sets corresponding link of the parent to NULL and disposes the node. Example. Remove -4 from a BST.

3.



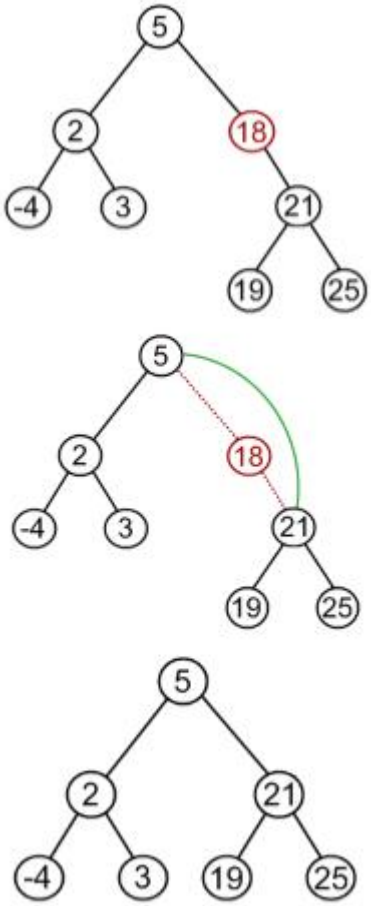
1.

Node to be deleted has one child.

2.

It this case, node is cut from the tree and algorithm links single child (with it's subtree) directly to the parent of the removed node. Example. Remove 18 from a BST.

3.



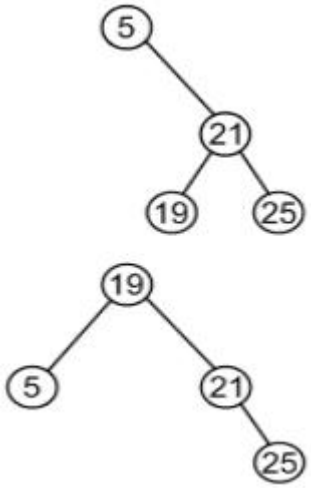
1.

Node to be deleted has two children.

2.

This is the most complex case. To solve it, let us see one useful BST property first. We are going to use the idea, that the same set of values may be represented as different binary-search trees. For example those BSTs:

3.



contains the same values {5, 19, 21, 25}. To transform first tree into second one, we can do following:

- choose minimum element from the right subtree (19 in the example);
- replace 5 by 19;
- hang 5 as a left child.

The same approach can be utilized to remove a node, which has two children:

- replace value of the node to be removed with found minimum. Now, right subtree contains a duplicate!
- apply remove to the right subtree to remove a duplicate.

CODE:

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

class bst
{
    int data;
    bst *left;
    bst *right;

public:
    bst();
    bst(int);
    bst *Insert(bst *, int);
    bst *minValueNode(bst *);
    bst *deleteNode(bst *, int);
    void Inorder(bst *);
    void Postorder(bst *);
    void Preorder(bst *);
};

bst::bst()
    : data(0), left(NULL), right(NULL)
{}

bst::bst(int val)
    : data(val), left(NULL), right(NULL)
{}

bst *bst::Insert(bst *root, int val)
{
    if (!root)
    {
        return new bst(val);
    }
    if (val >= root->data)
    {
        root->right = Insert(root->right, val);
```

```
38      }
39      else
40      {
41          root->left = Insert(root->left, val);
42      }
43      return root;
44  }
45
46  bst *bst::minValueNode(bst *node)
47  {
48      bst *current = node;
49
50      while (current && current->left != NULL)
51          current = current->left;
52
53      return current;
54  }
55
56  bst *bst::deleteNode(bst *root, int value)
57  {
58      if (root == NULL)
59          return root;
60
61      if (value < root->data)
62          root->left = deleteNode(root->left, value);
63
64      else if (value > root->data)
65          root->right = deleteNode(root->right, value);
66
67      else
68      {
69          if (root->left == NULL)
70          {
71              bst *temp = root->right;
72              delete root;
73              return temp;
74          }
```

```
        else if (root->right == NULL)
        {
            bst *temp = root->left;
            delete root;
            return temp;
        }
        bst *temp = minValueNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

void bst ::Inorder(bst *root)
{
    if (!root)
    {
        return;
    }
    Inorder(root->left);
    cout << root->data << "\t";
    Inorder(root->right);
}

void bst ::Postorder(bst *root)
{
    if (!root)
    {
        return;
    }
    Postorder(root->left);
    Postorder(root->right);
    cout << root->data << "\t";
}

void bst ::Preorder(bst *root)
{
```

```
    if (!root)
    {
        return;
    }
    cout << root->data << "\t";
    Preorder(root->left);
    Preorder(root->right);
}

int main()
{

    bst b, *root = NULL;
    root = b.Insert(root, 7);
    b.Insert(root, 18);
    b.Insert(root, 5);
    b.Insert(root, 2);
    b.Insert(root, 99);
    b.Insert(root, 84);
    b.Insert(root, 10);
    cout << "In Order Traversal: \n";
    b.Inorder(root);
    cout << "\nPost Order Traversal: \n";
    b.Postorder(root);
    cout << "\nPre Order Traversal: \n";
    b.Preorder(root);
    root = b.deleteNode(root, 5);
    cout << "\nIn Order Traversal after deleting node 5: \n";
    b.Inorder(root);
    root = b.deleteNode(root, 7);
    cout << "\nIn Order Traversal after deleting root node 7: \n";
    b.Inorder(root);
    return 0;
}
```

OUTPUT:

```
In Order Traversal:
2       5       7       10      18      84      99
Post Order Traversal:
2       5       10      84      99      18      7
Pre Order Traversal:
7       5       2       18      10      99      84
In Order Traversal after deleting node 5:
2       7       10      18      84      99
In Order Traversal after deleting root node 7:
2       10      18      84      99

...Program finished with exit code 0
Press ENTER to exit console.
```

Data Structure Lab
(ETCS-255)

Name: NIKHIL MATHUR

Roll No.: 05214802719

GROUP : 3C2

EXPERIMENT 8.

AIM: Implement a stack using singly linked list.

ALGORITHM:

Stack Operations using Linked List

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

- Step 1 - Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- Step 2 - Define a '**Node**' structure with two members **data** and **next**.
- Step 3 - Define a **Node** pointer '**top**' and set it to **NULL**.
- Step 4 - Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

- Step 1 - Create a **newNode** with given value.
- Step 2 - Check whether stack is **Empty** (**top** == **NULL**)
- Step 3 - If it is **Empty**, then set **newNode** → **next** = **NULL**.
- Step 4 - If it is **Not Empty**, then set **newNode** → **next** = **top**.
- Step 5 - Finally, set **top** = **newNode**.

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

- Step 1 - Check whether **stack** is **Empty** (**top == NULL**).
- Step 2 - If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
- Step 3 - If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
- Step 4 - Then set '**top** = **top** → **next**'.
- Step 5 - Finally, delete '**temp**'. (**free(temp)**).

display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

- Step 1 - Check whether stack is Empty (top == NULL).
- Step 2 - If it is Empty, then display 'Stack is Empty!!!' and terminate the function.
- Step 3 - If it is Not Empty, then define a Node pointer 'temp' and initialize with top.
- Step 4 - Display 'temp → data --->' and move it to the next node. Repeat the same until temp reaches to the first node in the stack. (temp → next != NULL).
- Step 5 - Finally! Display 'temp → data ---> NULL'.

CODE:

```
#include<iostream>
#include<stdio.h>

using namespace std;

struct Node
{
    int data;
    Node *next;
}*top=NULL,*p;

Node* newnode(int x)
{
    p=new Node;
    p->data=x;
    p->next=NULL;
    return(p);
}

void push(Node *q)
{
    if(top==NULL)
        top=q;
    else
    {
        q->next=top;
        top=q;
    }
}

void pop(){
    if(top==NULL){
        cout<<"Stack is empty!!";
    }
    else{
        cout<<"Deleted element is "<<top->data;
        p=top;
```

```
        top=top->next;
        delete(p);
    }
}

void showstack()
{
    Node *q;
    q=top;

    if(top==NULL){
        cout<<"Stack is empty!!";
    }
    else{
        while(q!=NULL)
        {
            cout<<q->data<<" ";
            q=q->next;
        }
    }
}

int main()
{
    int ch,x;
    Node *nptr;

    while(1)
    {
        cout<<"\n\n1.Push\n2.Pop\n3.Display\n4.Exit";
        cout<<"\nEnter your choice(1-4):";
        cin>>ch;

        switch(ch){
            case 1: cout<<"\nEnter data:";
                    cin>>x;
                    nptr=newnode(x);
                    push(nptr);
                    break;

            case 2: pop();
                    break;

            case 3: showstack();
                    break;

            case 4: exit(0);

            default: cout<<"\nWrong choice!!";
        }
    }

    return 0;
}
```

OUTPUT:

```
1.Push
2.Pop
3.Display
4.Exit
Enter your choice(1-4):3
40 30 20

1.Push
2.Pop
3.Display
4.Exit
Enter your choice(1-4):2
Deleted element is 40


1.Push
2.Pop
3.Display
4.Exit
Enter your choice(1-4):1

Enter data:20


1.Push
2.Pop
3.Display
4.Exit
Enter your choice(1-4):1

Enter data:30


1.Push
2.Pop
3.Display
4.Exit
Enter your choice(1-4):1

Enter data:40
```

Data Structure Lab
(ETCS-255)

Name: NIKHIL MATHUR

Roll No.: 05214802719

GROUP : 3C2

EXPERIMENT 11.

AIM: Implement insertion, deletion and display (inorder, preorder and postorder) on binary search tree .

ALGORITHM:

Sorting in Data Structure

Sorting is nothing but storage of data in sorted order, it can be in ascending or descending order. The term Sorting comes into picture with the term Searching.

There are so many things in our real life that we need to search, like a particular record in database, roll numbers in merit list, a particular telephone number, any particular page in a book etc.

Sorting arranges data in a sequence which makes searching easier. Every record which is going to be sorted will contain one key. Based on the key the record will be sorted.

For example, suppose we have a record of students, every such record will have the following data: Roll No. Name Age

Here Student roll no. can be taken as key for sorting the records in ascending or descending order. Now suppose we have to search a Student with roll no. 15, we don't need to search the complete record we will simply search between the Students with roll no. 10 to 20.

Sorting Efficiency

- There are many techniques for sorting. Implementation of particular sorting technique depends upon situation. Sorting techniques mainly depends on two parameters.
- First parameter is the execution time of program, which means time taken for execution of program.
- Second is the space, which means space taken by the program.

Type of Sorting

- Bubble Sort
- Selection Sort
- Insertion Sort
- Quick Sort
- Merge Sort
- Heap Sort
- Shell Sort
- Bucket or Radix Sort

Bubble Sort

- Bubble sorting is a simple sorting technique in which we arrange the elements of the list by forming pairs of adjacent elements
- Bubble Sort is an algorithm which is used to sort N elements that are given in a memory for eg: an Array with N number of elements.
- Bubble Sort compares all the element one by one and sort them based on their values.
- It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.
- Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.

Selection Sort

- The selection sort technique is based upon the extension of the minimum/maximum technique.
- Selection sorting is conceptually the simplest sorting algorithm.
- This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted.
- Selection sort is a simplicity sorting algorithm. It works as its name as it is. Here are basic steps of selection sort algorithm:

    - Find the minimum element in the list
    - Swap it with the element in the first position of the list
    - Repeat the steps above for all remainder elements of the list starting at the second position.

Advantages of Selection Sort

- It is simple and easy to implement.
- It can be used for small data sets.
- It is 60 per cent more efficient than bubble sort.

Insertion Sort

- An insertion sort is one that sorts a set of values by inserting values into an existing sorted file.
- It is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. This algorithm is less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.

Advantages of Insertion Sort

The advantages of this sorting algorithm are as follows:

- It is easy to implement and efficient to use on small sets of data.
- It can be efficiently implemented on data sets that are already substantially sorted.
- It performs better than algorithmslike selection sort and bubble sort. Insertion sort algorithm is simpler than shell sort, with only a small trade-off in efficiency. It is over twice as fast as the bubble sort and almost 40 per cent faster than the selection sort.
- It requires less memory space (only O(1) of additional memory space).
- It is said to be online, as it can sort a list as and when it receives new elements.

Quick Sort

- It is one of the most popular sorting techniques. Quick sort possesses a very good average-case behaviour among all the sorting techniques. This is developed by C.A.R. Hoare.

- Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not stable search, but it is very fast and requires very less additional space. It is based on the rule of Divide and Conquer (also called partition-exchange sort).
- This algorithm divides the list into three main parts:
- Elements less than the Pivot element
- Pivot element
- Elements greater than the pivot element

Merge Sort

- Merge sort is a sorting technique which divides the array into subarrays of size 2 and merge adjacent pairs.
- Merge Sort follows the rule of Divide and Conquer. But it doesn't divide the list into two halves.
- In merge sort the unsorted list is divided into N sub lists, each having one element, because a list of one element is considered sorted.
- Then, it repeatedly merge these sub lists, to produce new sorted sub lists, and at lasts one sorted list is produced.
- Merge Sort is quite fast, and has a time complexity of O(n log n). It is also a stable sort, which means the equal elements are ordered in the same order in the sorted list.

Heap Sort

- Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios.
- Heap is a special tree-based data structure that satisfies the following special heap properties
- Shape Property: Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.
- Heap Property: All nodes are either greater than or equal to or less than or equal to each of its children. If the parent nodes are greater than their children, heap is called a Max-Heap, and if the parent nodes are smaller than their child nodes, heap is called Min-Heap.

Shell Sort

- Shell sort, invented by Donald Shell in 1959, is a sorting algorithm that is a generalization of insertion sort. While discussing insertion sort, we have observed two things:
    - First, insertion sort works well when the input data is 'almost sorted'.
    - Second, insertion sort is quite inefficient to use as it moves the values just one position at a time.
- Shell sort is considered an improvement over insertion sort as it compares elements separated by a gap of several positions. This enables the element to take bigger steps towards its expected position.
- In Shell sort, elements are sorted in multiple passes and in each pass, data are taken with smaller and smaller gap sizes.

Radix Sort

- The idea is to consider the key one character at a time and to divide the entries, not into two sub lists, but into as many sub lists as there are possibilities for the given character from the key.
- If our keys, for example, are words or other alphabetic strings, then we divide the list into 26 sub lists at each stage.

That is, we set up a table of 26 lists and distribute the entries into the lists according to one of the characters in the key.

CODE:

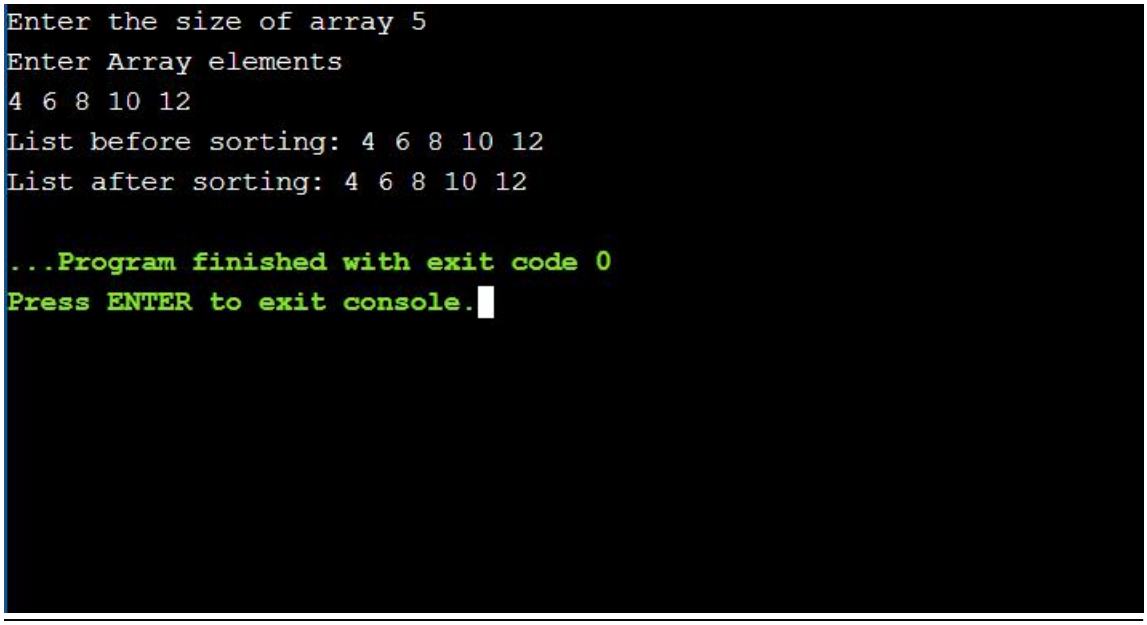
```
#include <iostream>
using namespace std;

void selectionsort(int arr[], int n)
{

    for(int i=0;i<n-1;i++)
    {
        int min=i;
        for(int j=i+1;j<n;j++)
        {
            if(arr[j]<arr[min])
            {
                min=j;
            }
        }
        if(min!=i)
        {
            swap(arr[min],arr[i]);
        }

    }

}
void insertionsort(int a[],int n)
{
    int i,j,key;
    for(int i=1;i<n;i++)
    {
        key=a[i];
        j=i-1;

        if(j>=0 and a[j]>key)
        {
            a[j+1]=a[j];
            j--;
        }
```

```
38          a[j+1]=key;
39      }
40  }
41  void bubblesort(int a[], int n)
42  {
43      int i,j;
44      for(int i=0;i<n;i++)
45      {
46          for(int j=0;j<n-i-1;j++)
47          {
48              if(a[j]>a[j+1])
49              {
50                  swap(a[j],a[j+1]);
51              }
52          }
53      }
54  }
55
56  void merge1(int a[], int l, int m, int r, int size)
57  {
58      int i=l;
59      int j=m+1;
60      int k=l;
61      int temp[size];
62
63      while(i<=m && j<=r)
64      {
65          if(a[i]<=a[j])
66          {
67              temp[k]=a[i];
68              k++;
69              i++;
70          }
71          else
72          {
73              temp[k]=a[j];
74              j++;
```

```
            k++;
        }
    }
        while(i<=m)
        {
            temp[k]=a[i];
            i++;
            j++;
        }
        while(j<=r)
        {
            temp[k]=a[j];
            k++;
            j++;
        }


     for(int p=l;p<=r;p++)
        {
            a[p]=temp[p];
        }
}
void mergesort(int arr[], int l, int r, int size)
{
    if(l<r)
    {
    int mid=(l+r)/2;

    mergesort(arr,l,mid,size);
    mergesort(arr,mid+1,r,size);
    merge1(arr,l,mid,r,size);
    }
}

int partition(int a[],int start, int end);

void quicksort(int arr[], int s, int e)
```

```
{
    int p;
    if(s<e)
    {
        p=partition(arr,s,e);
        quicksort(arr,s,p-1);
        quicksort(arr,p+1,e);
    }
}
int partition(int a[],int s, int e)
{
    int pivot=a[e];
    int pindex=s;

    for(int i=s;i<e;i++)
    {
        if(a[i]<=pivot)
        {
            swap(a[i],a[pindex]);
            pindex++;
        }
    }
    swap(a[e],a[pindex]);
    return pindex;
}

int main()
{
    int n;
    cout<<"Enter the size of array ";
    cin>>n;
    int a[n];
    cout<<"Enter Array elements "<<endl;
    for(int i=0;i<n;i++)
    {
        cin>>a[i];
    }
    cout<<"List before sorting: ";
    for(int i=0;i<n;i++)
    {
        cout<<a[i]<<" ";
    }
    cout<<endl;
   // mergesort(a,0,(n-1),n);
   quicksort(a,0,(n-1));
    cout<<"List after sorting: ";
    for(int i=0;i<n;i++)
    {
        cout<<a[i]<<" ";
    }

    return 0;
}
```

OUTPUT:


Enter the size of array 5
Enter Array elements
4 6 8 10 12
List before sorting: 4 6 8 10 12
List after sorting: 4 6 8 10 12
...Program finished with exit code 0
Press ENTER to exit console.