Practical -01

AIM: Implementation of different searching & sorting techniques.

1.Write a C++ Program for Insertion Sort

Sorting: Sorting in data structure is the process of arranging different elements in a particular order based on a particular set of criteria.

Sorting Techniques: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, Radix Sort.

<u>Code:</u>

```
#include<iostream>
using namespace std;
int main ()
{
  //int myarray[5] = { 12,4,3,1,15};
  int myarray[5];
  cout<<"Enter the 5 elements for the array";
  for (int i = 0; i < 5; i++)
  {
   cin>>myarray[i];
  cout<<"\nInput list is \n";
  for(int i=0;i<5;i++)
  {
    cout <<myarray[i]<<"\t";
  }
  for(int k=1; k<5; k++)
  {
    int temp = myarray[k];
    int j= k-1;
    while(j>=0 && temp <= myarray[j])
    {
      myarray[j+1] = myarray[j];
      j = j-1;
    }
  myarray[j+1] = temp;
```

```
  }
cout<<"\nSorted list is \n";
for(int i=0;i<5;i++)
   {
   cout <<myarray[i]<<"\t";
   }
}
```

Output:-

```
PS C:\Users\Harshali>
cd "c:\Users\Harshali\Downloads\" ; if ($?) { g++ Insertion.cpp -o Insertion } ; if ($?) { .\Insertion }
Enter the 5 elements for the array2
3
5
6
8

Input list is
2       3       5       6       8
Sorted list is
2       3       5       6       8
PS C:\Users\Harshali\Downloads>
```

2) Write a C++ Program for Binary Search

Searching: Searching is the fundamental process of locating a specific element or item within a collection of data. This collection of data can take various forms, such as arrays, lists, trees, or other structured representations.

Searching Techniques: linear Search, Binary Search, Hashing, Depth-First Search , Breadth-First Search

Code:

```
#include <iostream>

#include <string>

using namespace std;

int binarySearch(int myarray[], int beg, int end, int key)

{

   int mid;

   if(end >= beg) {

      mid = (beg + end)/2;
```

```
if(myarray[mid] == key)
      {
return mid+1;
      }
      else if(myarray[mid] < key) {
         return binarySearch(myarray,mid+1,end,key);
      }
      else {
         return binarySearch(myarray,beg,mid-1,key);
      }  }
   return -1;
}
int main ()
{
   int myarray[1121] = {5,8,10,13,21,23,25,43,54,75,80};
   int key, location=-1;
   cout<<"The input array is"<<endl;
   for(int i=0;i<10;i++){
      cout<<myarray[i]<<" ";
   }
   cout<<endl;
   cout<<"Enter the key that is to be searched:";
   cin>>key;
   location = binarySearch(myarray, 0, 10, key);
   if(location != -1)
   {
      cout<<"Key found at location "<<location;
   }
   else   {
      cout<<"Requested key not found";
   }
 }
```

Output:

```
PS C:\Users\Harshali\Downloads\event-organizers-Website-main> cd "c:\Users\Harshali\Downloads\" ; if ($?) { g++ binary.cpp -o binary } ; if ($?) { .\binary }
The input array is
5 8 10 13 21 23 25 43 54 75
Enter the key that is to be searched:5
Key found at location 1
PS C:\Users\Harshali\Downloads>
```

Practical-02

AIM: Perform various hashing techniques with Linear Probe as collision resolution.

Hashing Overview: Hashing involves mapping data to a fixed-size table (hash table) using a hash function. However, collisions occur when two keys hash to the same index.

Linear Probing; Linear probing is a collision resolution strategy where, upon a collision, the algorithm probes the next slot in the table (in a linear sequence) until an empty slot is found

Code:-

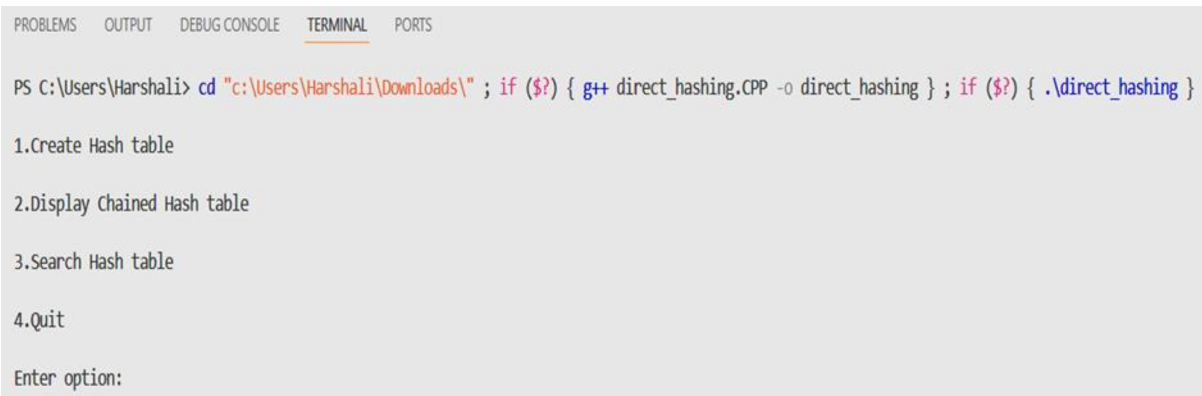
```
//It direct hashing technique with linear probe
#include <iostream>
#include <stdlib.h>
#include <math.h>
#include<conio.h>
#define SIZE 7
using namespace std;
class hashlist
{
    int no_elts;
    int arr[SIZE],addr[SIZE];
    public:
    void createht();
    void search();
    void displayht();
    int collision(int);
};
void hashlist::createht()
{
    int i,tempaddr,j,flag;
    cout<<"Input number of keys (integers)  to be stored in the Hash table\n";
    cin>>no_elts;
    // input keys
    cout<<"\n Input keys to be stored \n";
    for(i=0; i<no_elts; i++)
    {
```

```
        addr[i]=0;
        cin>>arr[i];
  tempaddr = arr[i];
        tempaddr=collision(tempaddr);
  addr[i] = tempaddr;
    }
}
void hashlist::displayht()
{
    int i;
    cout<<"Hashed list is:\n";
    cout<<"Elements"<<"\t"<<"Address"<<"\n";
    for (i=0; i<no_elts; i++)
    {
        cout<<arr[i]<<"\t"<<addr[i]<<"\n";
    }
}
void hashlist::search()
{
    int i,key,addr1;
    cout<<"\nEnter key to be searched:\n";
    cin>>key;
    addr1 = key;
    i=0;
    while (i<SIZE)
    {
           l1:  if(addr1 == addr[i])
            if(key==arr[i])
            {
                cout<<"\nMatch is found";
                break;
            }
            else
```

```
  {
          addr1 = addr1+1;
i++;
          goto l1;
 }
        i++;
  }
  if(i==SIZE)
    cout<<"\nMatch is not found\n";
}
int hashlist::collision(int tadd)
{
  int i;
  for(i=0;i<SIZE;i++)
  {
    if (tadd==addr[i])
    {
      cout<<"Collision ";
      tadd=tadd+1;
      //break;
    }
  }
  return tadd;
}
int main(void)
{
  int opt ;
  //clrscr();
  hashlist hl;
  do                    /* main menu*/
  {
    cout<<"\n1.Create Hash table \n";
    cout<<"\n2.Display Chained Hash table\n";
```

```
cout<<"\n3.Search Hash table \n";
        cout<<"\n4.Quit\n";
cout<<"\nEnter option:   ";
        cin>>opt;
        switch(opt)
    {
        case 1:
            hl.createht();
        break;
        case 2:
            hl.displayht();
        break;
        case 3:
            hl.search();
        break;
        case 4:
              exit(1);
              break;
        }
     }
    while(1);
    return 0;
}
```

Output:



```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\Users\Harshali> cd "c:\Users\Harshali\Downloads\" ; if ($?) { g++ direct_hashing.CPP -o direct_hashing } ; if ($?) { .\direct_hashing }

1.Create Hash table

2.Display Chained Hash table

3.Search Hash table

4.Quit

Enter option:
```

Practical-03

AIM: Implementation of Stacks & Ordinary Queue (Using arrays).

1)Stacks:

A stack follows the LIFO (Last In, First Out) principle.

Operations:

- Push: Add an element to the top of the stack.
- Pop: Remove and return the top element.
- Peek: View the top element without removing it.
- isEmpty: Check if the stack is empty.
-

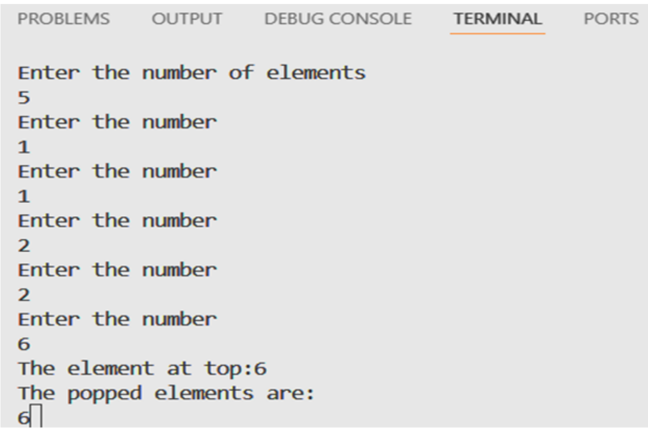
Code:

```
// staken.cpp

// overloading functions in base and derived classes

#include<iostream>

using namespace std;

#include<conio.h>

#include<process.h>            //for exit()

//////////////////////////////////////////////////////////////////

class Stack

{

protected:                    //NOTE: cant be private

int MAX;          //size of stack array

int st[10];            //stack: array of integers

int top;

int count,size;

        //index to top of stack

public:

Stack()                  //constructor

{

   top = -1;

   count=0;

   //size=10;

}

void push(int var)        //put number on stack
```

```
{
   if(top >= size-1)        //error if stack full
   {
  cout << "\nError: stack is full";
exit(1);
   }
   else
   top++;
   st[top] =var;
   //st[++top] = var;
}
int pop()                //take number off stack
{
   if(top < 0)            //error if stack empty    if (top==-1)
   {
      cout << "\nError: stack is empty \n";
      exit(1);
   }
   return st[top--];
}
int stacktop()
{
   if(top < 0)            //error if stack empty
   {
      cout << "\nError: stack is empty \n";
      exit(1);
   }
   return st[top];
}};
/////////////////////////////////////////////////////////////////
int main()
{
Stack s1;
```

```
int num,n,i;
cout<<"Enter the number of elements\n";
cin>>n;
for(i=0;i<n;i++)
{
   cout<<"Enter the number\n";
   cin>>num;
   s1.push(num);            //push some values onto stack
}
cout<<"The element at top:"<<s1.stacktop()<<endl;
cout<<"The popped elements are:";
for(i=0;i<n;i++)
{
   cout<<endl<<s1.pop();
   getch();
}
cout << endl << s1.pop();     //Stack Underflow
cout << endl;
return 0;
}
```

Output:



2)Queue:-

Ordinary Queue : An Ordinary Queue, also known simply as a Queue, is a linear data structure that follows the FIFO (First In, First Out) principle.

Operations:

- Enqueue: Add an element to the rear of the queue.
- Dequeue: Remove and return the front element.
- Peek: View the front element without removing it.
- isEmpty: Check if the queue is empty.
- isFull: Check if the queue is full.

Code:

```
#include<iostream>

#include<conio.h>

#include<stdlib.h>

using namespace std;

/* Queue definitions for Array implementation*/

class Queue

{

  int qAry[10];

  int front;

  int count;

  int rear;

  public:

Queue()

{

   front=-1;

   rear=-1;

   count=0;

}

void enqueue(int element)

{

   if (count == 10)

   {

 cout<<"Can't enqueue an element in queue: Queue Overflow.\n";

      //exit(1);  /* Exit, returning error code. */
```

```
    }
    // Put information in array; update queue.
    else
    {
  rear++;
        qAry[rear]=element;
        if(count==0)
        {
        //inserting into null queue
            front = 0;
            count=1;
        }
        else
            count++;
    }
    cout<<"Queue count is:"<<count<<endl;
}
void dequeue()
{
    int i;
    if (!count)
    {
        cout<<"Can't deque element from queue: queue underflow.\n";
          //   exit(1);  // Exit, returning error code.
    }
    else
    {
        i = qAry[front];
        qAry[front]=0;
        front++;
        if(count==1)//deleted single item from queue
            rear=front=-1;
        count--;
```

```
cout<<"\nThe element dequeued is:"<<i<<endl;
    cout<<"\nQueue count in "<<count<<endl;
  }}};
int main()
{
  int item, n,cnt,flag;
  Queue q1;
  //clrscr();
  while (1)
  {
    cout<<"   1. Enqueue\n";
    cout<<"   2. Dequeue\n";
    cout<<"   3. Quit\n";
    cout<<"\n      Enter option";
    cin>>n;
    switch(n)
    {
    case 1:
      cout<<"Input item to be inserted\n";
      cin>>item;
      q1.enqueue(item);
    break;
    case 2:
      q1.dequeue();
    break;
    case 3:
      exit(1);
      break;
    default: cout<<"Illegal option\n";
    break;
   }
```

```
}

 return 0;

}
```

Output:-

```
PS C:\Users\Harshali> cd "c:\Users\Harshali\Downloads\" ; if ($?) { g++ QUEUES.CPP -o QUEUES } ; if ($?) { .\QUEUES }
   1. Enqueue
   2. Dequeue
   3. Quit

      Enter option 1
Input item to be inserted
15
Queue count is:1
   1. Enqueue
   2. Dequeue
   3. Quit

      Enter option 2

The element dequeued is:15

Queue count in 0
   1. Enqueue
   2. Dequeue
   3. Quit
```

Practical-04

AIM: Implementation of Stack Applications like:

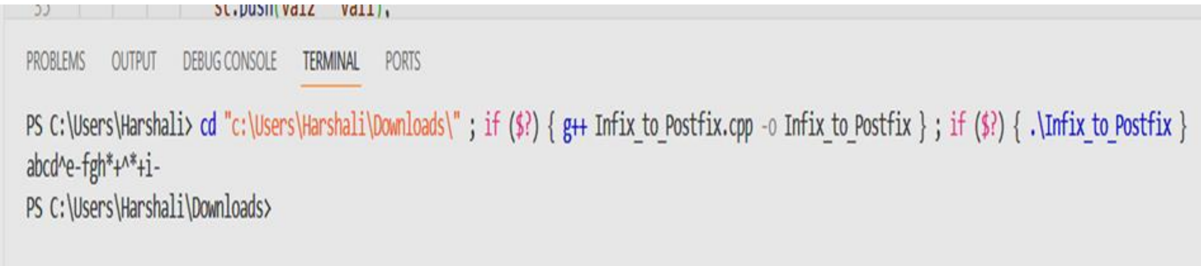
1) **infix to postfix**
   - infix Notation: Operators are placed between operands.
     Example: A + B
   - Postfix Notation :Operators follow the operands.
     Example: A B +

Code:

```
#include <bits/stdc++.h>

using namespace std;

// Function to return precedence of operators
int prec(char c) {
    if (c == '^')
        return 3;
    else if (c == '/' || c == '*')
        return 2;
    else if (c == '+' || c == '-')
        return 1;
    else
        return -1;
}

// The main function to convert infix expression
// to postfix expression
void infixToPostfix(string s) {
    stack<char> st;
    string result;

    for (int i = 0; i < s.length(); i++) {
        char c = s[i];

        // If the scanned character is
        // an operand, add it to the output string.
        if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9'))
            result += c;

        // If the scanned character is an
        // '(', push it to the stack.
        else if (c == '(')
            st.push('(');

        // If the scanned character is an ')',
        // pop and add to the output string from the stack
        // until an '(' is encountered.
        else if (c == ')') {
            while (st.top() != '(') {
                result += st.top();
                st.pop();
            }
```

```
            st.pop();
        }
    // If an operator is scanned

    else {
            while (!st.empty() && prec(c) < prec(st.top()) ||
                   !st.empty() && prec(c) == prec(st.top())) {
                result += st.top();
                st.pop();
            }
            st.push(c);
        }
    }

    // Pop all the remaining elements from the stack
    while (!st.empty()) {
        result += st.top();
        st.pop();
    }

    cout << result << endl;
}

int main() {
    string exp = "a+b*(c^d-e)^(f+g*h)-i";
    infixToPostfix(exp);
    return 0;
}
```

Output:-



2) Postfix evaluation

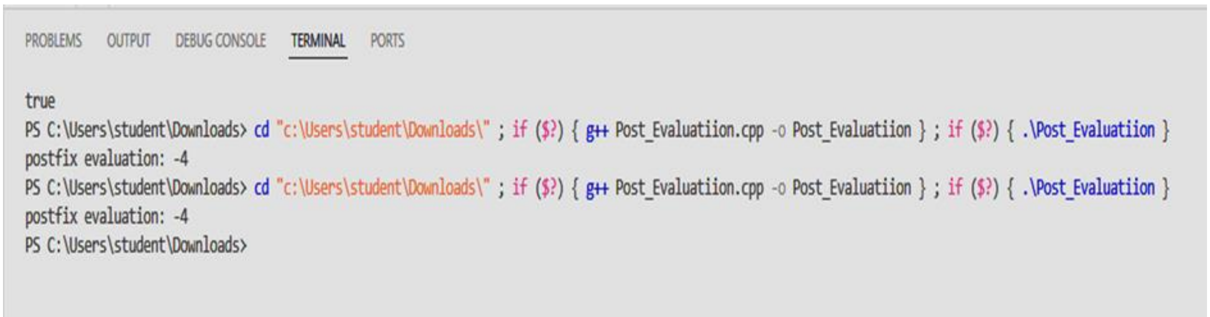
Postfix Evaluation refers to the process of calculating the value of a mathematical expression written in Postfix Notation (Reverse Polish Notation). Postfix notation places operators after their operands and eliminates the need for parentheses or operator precedence.

Code:

```
// C++ program to evaluate value of a postfix expression
#include <bits/stdc++.h>
using namespace std;
// The main function that returns value
// of a given postfix expression
int evaluatePostfix(string exp)
{
```

```
// Create a stack of capacity equal to expression size
    stack<int> st;
    // Scan all characters one by one
    for (int i = 0; i < exp.size(); ++i) {
        // If the scanned character is an operand
        // (number here), push it to the stack.
        if (isdigit(exp[i]))
            st.push(exp[i] - '0');
        // If the scanned character is an operator,
        // pop two elements from stack apply the operator
        else {
            int val1 = st.top();
            st.pop();
            int val2 = st.top();
            st.pop();
            switch (exp[i]) {
            case '+':
                st.push(val2 + val1);
                break;
            case '-':
                st.push(val2 - val1);
                break;
            case '*':
                st.push(val2 * val1);
                break;
            case '/':
                st.push(val2 / val1);
                break;
            }} }
    return st.top();
}

// Driver code
int main()
{
    string exp = "231*+9-";

    // Function call
    cout << "postfix evaluation: " << evaluatePostfix(exp);
    return 0;
}
```

Output:-



3)Balancing of Parenthesis:

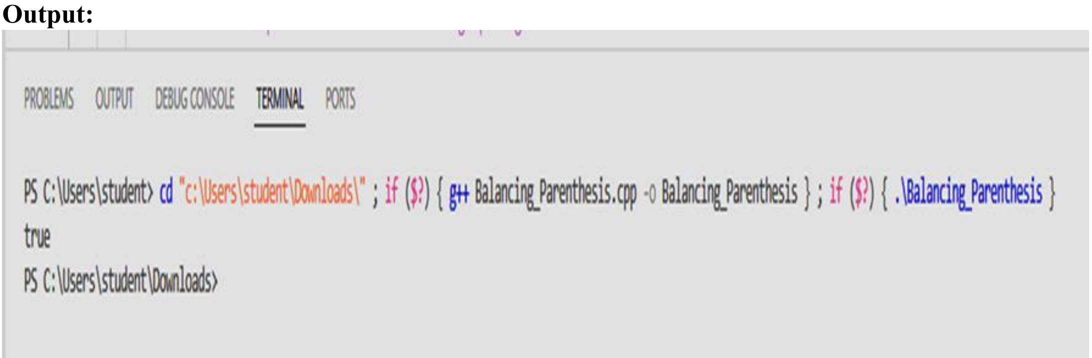
Balancing of Parentheses is a common problem in data structures that involves checking whether the parentheses (or other brackets) in an expression are properly matched and nested. This is crucial in scenarios such as validating mathematical expressions, programming code, or markup languages.

Code:

// C++ program to check for balanced brackets.

```
#include <bits/stdc++.h>
using namespace std;
// Function to check if brackets are balanced
bool ispar(const string& s) {  // Pass string by reference
  // Declare a stack to hold the previous brackets.
    stack<char> stk;
    for (int i = 0; i < s.length(); i++) {
        // Check if the character is an opening bracket
        if (s[i] == '(' || s[i] == '{' || s[i] == '[') {
            stk.push(s[i]);
        }
    else {
            // If it's a closing bracket, check if the stack is non-empty
            // and if the top of the stack is a matching opening bracket
            if (!stk.empty() &&
                ((stk.top() == '(' && s[i] == ')') ||
                 (stk.top() == '{' && s[i] == '}') ||
                 (stk.top() == '[' && s[i] == ']'))) {
                stk.pop(); // Pop the matching opening bracket
            }
            else {
                return false; // Unmatched closing bracket
            } }}
      // If stack is empty, return true (balanced), otherwise false
    return stk.empty();
}
int main() {
  string s = "{()}[]";
    // Function call
    if (ispar(s))
        cout << "true";
    else
        cout << "false";
    return 0; }
```

Output:



Practical-05

AIM: Implementation of all types of linked lists.

1) Write a C++ Program to Demonstrate single linked list

A singly linked list is a fundamental data structure, it consists of nodes where each node contains a data field and a reference to the next node in the linked list. The next of the last node is null, indicating the end of the list. Linked Lists support efficient insertion and deletion operations.

Operations: Insertion, Deletion, Traversal

Code:

```
#include<stdlib.h>
#include<iostream>
#include<conio.h>
using namespace std;
//This program creates sorted linked list.
typedef struct node
{
   int data;
   struct node *link;
}NODE;
typedef struct {
  int count;
  int *pos;
   NODE *head;
}HEAD;
class linklist
{
    HEAD *pNew;
    public:
    HEAD *createHead()
    {
        pNew = new HEAD;
        if(pNew)
        {
            pNew->head = NULL;
```

```
        pNew->count = 0;
        cout<<"Memory allocated\n";
    }
    else
        pNew = NULL;
    return pNew;
}
void insertNode(HEAD *pList,int dain)
{
int loc,i;
NODE *pNew,*pPre,*temp;
pPre =pList->head;
pNew = new NODE;
if(pNew == NULL)
    cout<<"\nMemory overflow.\n";
else
{
    pNew->data = dain;
}
temp = pList->head;
if (dain<temp->data) //Data is less than first node's data
{
    pPre=NULL;
    temp=NULL;
}
while (temp!=NULL) // Inserting at middle position: finding position to insert
{
    if(dain>temp->data)
    {
        pPre=temp;
        temp=temp->link;
    }
    else
        temp=temp->link;
```

```
   }
 if(pPre==NULL) //adding before first node or to empty list
   {
      pNew->link = pList->head;
      pList->head =pNew;
      //pNew->link=NULL;
      pList->count = pList->count+1;
      cout<<"\ndata is inserted.\n";
   }
   else //adding at last or middle position
   {
         pNew->link = pPre->link;
         pPre->link = pNew;
         pList->count = pList->count+1;
         cout<<"\ndata is inserted.\n";
    }
}
void traverse(HEAD *pList)
{
   NODE *pWalk;
   int i;
   if(pList->head==NULL)
      cout<<"List is empty\n";
   else
   {
      pWalk = pList->head;
      //while(pWalk!=NULL)
      i=0;
      while (i<pList->count)
      {
         cout<<pWalk->data<<"\t";
         pWalk=pWalk->link;
 i++;
      }
```

```
 cout<<"\n";
        cout<<"Num ber of elements in linked list is \n"<<pList->count;
    }
}
void deleteNode(HEAD *pList)
{
    NODE *pWalk,*pPre;
    int num,i;
    cout<<"Enter the element to be deleted:\n";
    cin>>num;
    pWalk=pList->head;
    while(pWalk!=NULL)
    {
        if(pWalk->data==num)
            break;
        else
        {
            pPre= pWalk;
            //There should be some initialization for pPre.
            //So pPre=pWalk
            pWalk=pWalk->link;
        }
    }
    if(pWalk==pList->head) //To delete first node
    {
        pList->head = pWalk->link;
        free(pWalk);
        pList->count = pList->count-1;
        cout<<"Number is deleted\n";
    }
 else if(pWalk==NULL)
        cout<<"Number not found in list\n";
    else
   {
```

```
 pPre->link = pWalk->link;
        free(pWalk);
        pList->count = pList->count-1;
        cout<<"Number is deleted\n";
    }
    }
void searchList(HEAD *pList,int data)
{
    NODE *pWalk;
    int i,flag;
    pWalk=pList->head;
    i=1;
    while(pWalk!=NULL)
    {
        if(pWalk->data==data)
        {
            cout<<"Data is present at "<<i<<"th location\n ";
            flag=1;
            break;
        }
        else
            pWalk=pWalk->link;
        i++;
    }
    if(pWalk==NULL)
    {
        cout<<"Data is not present\n";
        flag=0;
    }
}
void retrieveNode(HEAD *pList, int locn)
{
 NODE *pWalk;
```

```
    int i,flag;
    pWalk=pList->head;
    for(i=1;i<=locn-1;i++)
    {
        pWalk =pWalk->link;
    }
    cout<<"Data at"<<locn<< "is\t"<<pWalk->data;
}
void emptyList(HEAD *pList)
{
    if(pList->count==0)
        cout<<"List is empty\n";
    else
        cout<<"List is not empty\n";
}
void reverseList(HEAD *pList)
{
    NODE *pWalk;
    int i;
    int *arr;
    arr = new int[pList->count];
    pWalk=pList->head;
    i=-1;
    while(pWalk!=NULL) //To store linked list into array
    {
        i++;
        arr[i]=pWalk->data;
        pWalk=pWalk->link;
   }
cout<<"\n";
    for(;i>=0;i--) // print the array in reverse
        cout<<arr[i]<<"\t";
}
void destroyList(HEAD *pList)
```

```
{
    NODE *pWalk;
    while (pList->count!=0)
    {
 pWalk=pList->head;
        pList->head = pWalk->link;
        pList->count = pList->count-1;
free(pWalk);
    }
    free(pList);
    cout<<"Linked list destroyed\n";
}
};
int flag;
int main()
{
    int choice,datain;
    HEAD *pList;
    //clrscr();
    linklist l;
    pList=l.createHead();
    while(1)
    {
        cout<<"1. Insert linked list node\n";
cout<<"2. Traverse list\n";
 cout<<"3.Delete Node\n";
 cout<<"4.Search Element\n";
        cout<<"5. Retrieve Node\n";
        cout<<"6. Check empty List\n";
        cout<<"7. Reverse the list\n";
        cout<<"8. Exit\n";
        cout<<"\nEnter choice:";
        cin>>choice;
        switch(choice)
```

```
    {
        case 1:
cout<<"\nEnter the element to be inserted\n";
            cin>>datain;
            l.insertNode(pList,datain);
        break;
        case 2:
            l.traverse(pList);
        break;
        case 3:
            l.deleteNode(pList);
        break;
        case 4:
            cout<<"\nEnter the element to be searched:\n";
            cin>>datain;
            l.searchList(pList,datain);
        break;
        case 5:
            cout<<"\nEnter the location to be retrieved:\n";
            cin>>datain;
            l.retrieveNode(pList,datain);
        break;
        case 6:
            l.emptyList(pList);
break;
        case 7:
            l.reverseList(pList);
        break;
        case 8:
            l.destroyList(pList);
            exit(1);
            break;
default: cout<<"Illegal option\n";
    }
```

```
}
return 0;
}
```

Output:

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS                Code
[Running] cd "c:\Users\shaikh aasiya\Downloads\" && g++ LINKEDLIST1.CPP -o LINKEDLIST1 && "c:\Users\shaikh aasiya\Downloads\"LINKEDLIST1
Memory allocated
1. Insert linked list node
2. Traverse list
3.Delete Node
4.Search Element
5. Retrieve Node
6. Check empty List
7. Reverse the list
8. Exit

Enter choice:
```

2) . Write a C++ Program to Demonstrate Double linked list

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, forward as well as backward easily as compared to Single Linked List.

Operations : Insertion, Deletion, Traversal

Code:

```
#include<iostream>
#include<conio.h>
#include<stdlib.h>
using namespace std;
typedef struct {
  int count;
  struct node *rear;
  struct node *back;
  struct node *fore;
}HEAD;
typedef struct node
{
   int data;
   struct node *back;
   struct node *fore;
}NODE;
class dll
{
HEAD *pNew;
public:
HEAD *createHead()
{
    HEAD *pNew;
    pNew = new HEAD;
    if(pNew)
    {
```

```
 pNew->count = 0;
      pNew->rear=NULL;
      pNew->back=NULL;
      pNew->fore=NULL;
      cout<<"Memory allocated"<<"\n";
   }
   else
      pNew = NULL;
   return pNew;
}
void insertNode(HEAD *,int);
int searchNode(HEAD *,int);
void traverse(HEAD *);
void deleteNode(HEAD *);
void reverseList(HEAD *);
};
void dll::insertNode(HEAD *pList, int element)
{
int loc,i;
   NODE *pNew,*pPre,*temp,*pSucc;
   pPre =pList->fore;
   pNew = new NODE;
   if(pNew == NULL)
      cout<<"Memory overflow.\n";
   else
   {
      pNew->data = element;
   }
   //to locate a position according to ascending order
   //To insert into empty list or at first position
   if(pPre==NULL)
   {
      //pPre->back=pNew;
      pNew->fore=pList->fore;
      pNew->back=NULL;
      pList->fore=pNew;
      if (pPre==NULL)
         pList->rear=pNew;
   }
   else if (element<pPre->data) // To insert first position
   {
      pNew->fore= pList->fore;
      pNew->back=NULL;
      pPre->back = pNew;
      pList->fore= pNew;
   }

    //to insert at last position

   //temp=pList->rear;
   else if (element>pList->rear->data)
   {
      pPre=pList->rear;
      pNew->fore=NULL;
      pNew->back=pPre;
```

```
        pPre->fore=pNew;
        pList->rear=pNew;
      }
      //to insert at middle position
      else
      {
        temp=pPre=pList->fore;
        while (element>temp->data)
        {
          pPre=temp;
          temp=temp->fore;
        }
        pSucc=pPre->fore;
        pNew->fore=pSucc;
        pSucc->back=pNew;
        pPre->fore=pNew;
        pNew->back=pPre;
       }
    pList->count= pList->count+1;
      cout<<"Data is inserted\n";
    cout<<"Number of elements in the list is "<<pList->count;
    }
    void dll::deleteNode(HEAD *pList)
    {
      NODE *pDlt,*pSucc,*pPre;
      int num,i;
      cout<<"Enter the element to be deleted:\n";
      cin>>num;
      pDlt=pList->fore;
      i=1;
      while(i<=pList->count)
      {
        if(pDlt->data==num)
          break;
        else
          pDlt=pDlt->fore;
        i++;
      }
      pPre=pDlt->back;
      pSucc=pDlt->fore;
      if(pSucc==NULL)    //To delete last node
      {
        pPre->fore=NULL;
        pList->rear=pPre;
      }
      else  //To delete middle or first node
      {
        pPre->fore=pSucc;
        pSucc->back=pPre;
      }
      free(pDlt);

     if(i==1)   //To delete first node
        pList->fore=pSucc;
      pList->count=pList->count-1;
```

```
    cout<<"Node is deleted\n";
    cout<<"Nodes remaining are: \n"<<pList->count;
}
void dll::traverse(HEAD *pList)
{
    NODE *pWalk;
    int i;
    if(pList->count==0)
        cout<<"List is empty\n";
    else
    {
        pWalk= pList->fore;
        i=1;
        while(i<=pList->count)
{
            cout<<pWalk->data<<"\t";

  pWalk=pWalk->fore;
            i++;
        }
        cout<<"\n";
    }
}
int dll::searchNode(HEAD *pList,int data)
{
    NODE *pS,*p1;
    int flag=0;
    pS=pList->fore;
    if (pS->data==data) //To check at first position
        flag=1;
    else
    {
        p1=pList->rear;     //To check at last position
        if(p1->data==data)
            flag=1;
    }
    if(flag==0)
    while(pS!=NULL)
    {
        if(pS->data==data)
        {
            flag=1;
            p1=pS;
            break;
        }
        else
            pS=pS->fore;
    }
    if(pS==NULL)
        flag=0;
    return flag;
}
void dll::reverseList(HEAD *pList)
{
    NODE *pWalk,*pPre;
```

```
    pWalk=pList->rear;
    while(pWalk!=NULL)
    {
       cout<<pWalk->data<<"\t";
       pWalk=pWalk->back;
    }
    cout<<endl;
}
int main()
{
    HEAD *pList;
    int ch,datain,flag;
  dll d;
    pList = d.createHead();
    while(1)
{
       cout<<"1. Insert node\n";
       cout<<"2.Display List\n";
       cout<<"3. Delete Node\n";
       cout<<"4.Search Node\n";
       //printf("5.Copy List\n");
       cout<<"6.Reverse List\n";
       cout<<"7.Exit\n";
       cout<<"Enter choice\n";
       cin>>ch;
       switch(ch)
       {
          case 1:
             cout<<"\nEnter the element to be inserted\n";
             cin>>datain;
             d.insertNode(pList,datain);
          break;
          case 2:
             d.traverse(pList);
          break;
          case 3:
             d.deleteNode(pList);
          break;
          case 4:
             cout<<"\nEnter the element to be searched\n";
             cin>>datain;
             flag=d.searchNode(pList,datain);
             if(flag==1)
                cout<<"Data is present\n";
             else
                cout<<"Data is not present\n";
          break;
          /*case 5:

 copyList(pList);
          break;*/
          case 6:
               d.reverseList(pList);
          break;
          case 7:
```

```
          exit(1);
        break;
      } }
   return 0;
}

/*void copyList(HEAD *pList1)
{
   int i;
 HEAD *pList2;
  NODE *pNew,*pWalk,*pPre;
   pList2 = createHead();
   pWalk=pList1->fore;

 i=1;
   pNew = (NODE *)malloc(sizeof(NODE));
   if(pNew == NULL)
      printf("Memory overflow.\n");
   else
   {
      pNew->data = pWalk->data;
      pPre = pNew;
   }
   if(i==1)
   {
      pList2->fore=pNew;
   }
   i++;
   while(i<=pList1->count)
   {
      pWalk=pWalk->fore;
      pNew = (NODE *)malloc(sizeof(NODE));
      pNew->data=pWalk->data;
      pNew->back = pPre;
      pPre->fore = pNew;
      pPre=pNew;
      i++;
   }
   if(i==pList1->count)
      pList1->rear=pNew;
   pList2->count=pList1->count;
   printf("Number of nodes copied are: %d\n", pList2->count);
   traverse(pList2);
}
```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\Harshali> cd "c:\Users\Harshali\Downloads\" ; if ($?) { g++ DLL1.CPP -o DLL1 } ; if ($?) { .\DLL1 }
Memory allocated
1. Insert node
2.Display List
3. Delete Node
4.Search Node
6.Reverse List
7.Exit
Enter choice
```

Practical-06

AIM: Create and perform various operations on BST.

The binary tree that is ordered is called the binary search tree. In a binary search tree, the nodes to the left are less than the root node while the nodes to the right are greater than or equal to the root node.

Operations on BST: Insertion, Search, Deletion, Traversal(In-Order, pre-order, post-order).

Code:

```
// C++ Program to implement binary search tree

#include <iostream>

using namespace std;

// Node structure for a Binary Search Tree

struct Node {

    int data;

    Node* left;

    Node* right;

};

// Function to create a new Node

Node* createNode(int data)

{

    Node* newNode = new Node();

    newNode->data = data;

    newNode->left = newNode->right = nullptr;

    return newNode;

}

// Function to insert a node in the BST

Node* insertNode(Node* root, int data)

{

    if (root == nullptr) { // If the tree is empty, return a

                           // new node

        return createNode(data);

    }

// Otherwise, recur down the tree

    if (data < root->data) {

        root->left = insertNode(root->left, data);
```

```
    }
  else if (data > root->data) {
  root->right = insertNode(root->right, data);
    }
 // return the (unchanged) node pointer
    return root;
}
// Function to do inorder traversal of BST
void inorderTraversal(Node* root)
{
    if (root != nullptr) {
        inorderTraversal(root->left);
        cout << root->data << " ";
        inorderTraversal(root->right);
    }
}
// Function to search a given key in a given BST
Node* searchNode(Node* root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root == nullptr || root->data == key) {
        return root;
    }
 // Key is greater than root's key
    if (root->data < key) {
        return searchNode(root->right, key);
    }
 // Key is smaller than root's key
    return searchNode(root->left, key);
}
// Function to find the inorder successor
Node* minValueNode(Node* node)
{
    Node* current = node;
```

```
  // loop down to find the leftmost leaf
   while (current && current->left != nullptr) {
       current = current->left;
    }
    return current;
}
// Function to delete a node
Node* deleteNode(Node* root, int data)
{
    if (root == nullptr)
        return root;
 // If the data to be deleted is smaller than the root's
    // data, then it lies in the left subtree
    if (data < root->data) {
        root->left = deleteNode(root->left, data);
    }
    // If the data to be deleted is greater than the root's
    // data, then it lies in the right subtree
    else if (data > root->data) {
        root->right = deleteNode(root->right, data);
    }
    // if data is same as root's data, then This is the node
    // to be deleted
    else {
        // node with only one child or no child
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
```

```
return temp;
}
 // node with two children: Get the inorder successor
   // (smallest in the right subtree)
        Node* temp = minValueNode(root->right);
// Copy the inorder successor's content to this node
        root->data = temp->data;
 // Delete the inorder successor
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
// Main function to demonstrate the operations of BST
int main()
{
Node* root = nullptr;
    // create a BST
    root = insertNode(root, 50);
    root = insertNode(root, 30);
    root = insertNode(root, 25);
    root = insertNode(root, 40);
    root = insertNode(root, 70);
    root = insertNode(root, 60);
    root = insertNode(root, 80);
// Print the inorder traversal of a BST
    cout << "Inorder traversal of the given Binary Search "
            "Tree is: ";
    inorderTraversal(root);
    cout << endl;
// delete a node in BST
    root = deleteNode(root, 65);
    cout << "After deletion of 20: ";
    inorderTraversal(root);
```

```
cout << endl;
// Insert a node in BST
   root = insertNode(root, 63);
   cout << "After insertion of 25: ";
   inorderTraversal(root);
   cout << endl;
 // Search a key in BST
   Node* found = searchNode(root, 40);
// check if the key is found or not
   if (found != nullptr) {
      cout << "Node  found in the BST." << endl;
   }
   else {
      cout << "Node  not found in the BST." << endl;
   }
   return 0;
}
```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    SEARCH ERROR

PS C:\Users\student> cd "c:\Users\student\Downloads\" ; if ($?) { g++ BST.cpp -o BST } ; if ($?) { .\BST }
Inorder traversal of the given Binary Search Tree is: 25 30 40 50 60 70 80
After deletion of 20: 25 30 40 50 60 70 80
After insertion of 25: 25 30 40 50 60 63 70 80
Node  found in the BST.
PS C:\Users\student\Downloads>
```

Practical-07

AIM: Implementing Heap with different operations.

A heap is a specialized binary tree-based data structure that satisfies the heap property. There are two types of heaps:

- Max Heap: In a max heap, the value of each node is greater than or equal to the values of its children, and the largest element is at the root.
- Min Heap: In a min heap, the value of each node is less than or equal to the values of its children, and the smallest element is at the root.

Heap Operations: Insertion, Deletion, Heapify, Heapify Down, Building a Heap

Code:

```
#include<iostream>
#include <bits/stdc++.h>
using namespace std;
class BinaryHeap {
  public:
    int capacity; /*Maximum elements that can be stored in heap*/
  int size; /*Current no of elements in heap*/
  int * arr; /*array for storing the keys*/
 BinaryHeap(int cap) {
    capacity = cap; /*Assigning the capacity*/
    size = 0; /*Intailly size of hepa is zero*/
    arr = new int[capacity]; /*Creating a array*/
  }
  /*returns the parent of ith Node*/
  int parent(int i) {
    return (i - 1) / 2;
  }
  /*returns the left child of ith Node*/
  int left(int i) {
    return 2 * i + 1;
  }
  /*Returns the right child of the ith Node*/
  int right(int i) {
    return 2 * i + 2;
```

```
 }
 /*Insert a new key x*/
 void Insert(int x) {
  if (size == capacity) {
    cout << "Binary Heap Overflwon" << endl;
    return;
 }
  arr[size] = x; /*Insert new element at end*/
  int k = size; /*store the index ,for checking heap property*/
  size++; /*Increase the size*/
 /*Fix the min heap property*/
  while (k != 0 && arr[parent(k)] > arr[k]) {
    swap( & arr[parent(k)], & arr[k]);
    k = parent(k);
  }}
void Heapify(int ind) {
  int ri = right(ind); /*right child*/
  int li = left(ind); /*left child*/
  int smallest = ind; /*intially assume violated value in Min value*/
  if (li < size && arr[li] < arr[smallest])
    smallest = li;
  if (ri < size && arr[ri] < arr[smallest])
    smallest = ri;
  /*smallest will store the minvalue index*/
  /*If the Minimum among the three nodes is the parent itself,
  that is Heap property satisfied so stop else call function recursively on Minvalue node*/
  if (smallest != ind) {
    swap( & arr[ind], & arr[smallest]);
    Heapify(smallest);
  }}
 int getMin() {
  return arr[0];
 }
 int ExtractMin() {
```

```
  if (size <= 0)
      return INT_MAX;
    if (size == 1) {
      size--;
      return arr[0];
}
int mini = arr[0];
    arr[0] = arr[size - 1]; /*Copy last Node value to root Node value*/
    size--;
    Heapify(0); /*Call heapify on root node*/
    return mini;
  }
  void Decreasekey(int i, int val) {
    arr[i] = val; /*Updating the new_val*/
    while (i != 0 && arr[parent(i)] > arr[i]) /*Fixing the Min heap*/ {
      swap( & arr[parent(i)], & arr[i]);
      i = parent(i);
    }}
  void Delete(int i) {
    Decreasekey(i, INT_MIN);
    ExtractMin();
  }
  void swap(int * x, int * y) {
    int temp = * x;
    * x = * y;
    * y = temp;
  }
  void print() {
    for (int i = 0; i < size; i++)
      cout << arr[i] << " ";
    cout << endl;
  }};
int main()
```

```
{
  BinaryHeap h(20);
  h.Insert(4);
  h.Insert(1);
  h.Insert(2);
 h.Insert(6);
h.Insert(7);
  h.Insert(3);
  h.Insert(8);
  h.Insert(5);
  cout << "Min value is " << h.getMin() << endl;
  h.Insert(-1);
  cout << "Min value is " << h.getMin() << endl;
  h.Decreasekey(3, -2);
  cout << "Min value is " << h.getMin() << endl;
  h.ExtractMin();
  cout << "Min value is " << h.getMin() << endl;
  h.Delete(0);
  cout << "Min value is " << h.getMin() << endl;
  return 0;
}
```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\student\Downloads> cd "c:\Users\student\Downloads\" ; if ($?) { g++ Heap.cpp -o Heap } ; if ($?) { .\Heap }
Min value is 1
Min value is -1
Min value is -2
Min value is -1
Min value is 1
PS C:\Users\student\Downloads>
```

Practical-08

AIM: Create a Graph storage structure (eg. Adjacency matrix)

Adjacency Matrix is a square matrix used to represent a finite graph by storing the relationships between the nodes in their respective cells. For a graph with V vertices, the adjacency matrix A is an V X V matrix or 2D array.

Operations:

- Add an Edge: Set the matrix entry to 1 (or the weight of the edge).
- Remove an Edge: Set the matrix entry to 0 (or infinity if it's a weighted graph).
- Check if There is an Edge: Check the matrix entry at the position [i][j][i][j][i][j].
- Display the Graph: Print the adjacency matrix.

<u>Code:</u>

```
#include<iostream>

using namespace std;

int vertArr[20][20]; //the adjacency matrix initially 0

int count = 0;

void displayMatrix(int v) {

  int i, j;

  for(i = 0; i < v; i++) {

    for(j = 0; j < v; j++) {

      cout << vertArr[i][j] << " ";

    }

    cout << endl;

  }}

void add_edge(int u, int v) {       //function to add edge into the matrix

  vertArr[u][v] = 1;

  vertArr[v][u] = 1; }

main(int argc, char* argv[]) {

  int v = 6;    //there are 6 vertices in the graph

  add_edge(0, 4);

  add_edge(0, 3);

  add_edge(1, 2);

  add_edge(1, 4);

  add_edge(1, 5);

  add_edge(2, 3);
```

```
 add_edge(2, 5);
  add_edge(5, 3);
  add_edge(5, 4);
  displayMatrix(v);
}
```

Output:

```
PROBLEMS 2   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\Users\student\Downloads> cd "c:\Users\student\Downloads\" ; if ($?) { g++ Graph_AM.cpp -o Graph_AM } ; if ($?) { .\Graph_AM }
0 0 0 1 1 0
0 0 1 0 1 1
0 1 0 1 0 1
1 0 1 0 0 1
1 1 0 0 0 1
0 1 1 1 1 0
PS C:\Users\student\Downloads>
```

Practical-09

AIM: Implementation of Graph traversal. (DFS and BFS)

1)DFS

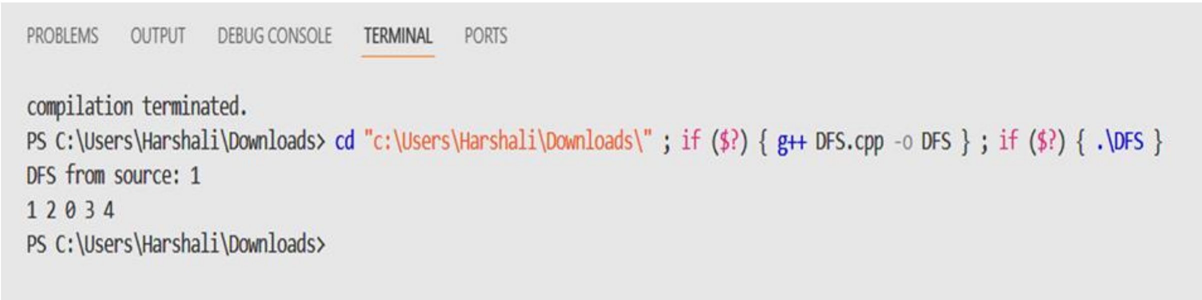
Depth First Search (or DFS) for a graph is similar to Depth first Traversal of a tree Like trees, we traverse all adjacent vertices one by one. When we traverse an adjacent vertex, we completely finish the traversal of all vertices reachable through that adjacent vertex. After we finish traversing one adjacent vertex and its reachable vertices, we move to the next adjacent vertex and repeat the process.

Code:

```
#include <bits/stdc++.h>

using namespace std;

// Recursive function for DFS traversal

void DFSRec(vector<vector<int>> &adj, vector<bool> &visited, int s){

  visited[s] = true;

// Print the current vertex

    cout << s << " ";

 // Recursively visit all adjacent vertices

    // that are not visited yet

    for (int i : adj[s])

        if (visited[i] == false)

            DFSRec(adj, visited, i);

}

// Main DFS function that initializes the visited array

// and call DFSRec

void DFS(vector<vector<int>> &adj, int s){

    vector<bool> visited(adj.size(), false);

    DFSRec(adj, visited, s);

}

// To add an edge in an undirected graph

void addEdge(vector<vector<int>> &adj, int s, int t){

    adj[s].push_back(t);

    adj[t].push_back(s);

}

int main(){

    int V = 5;
```

```
vector<vector<int>> adj(V);
 // Add edges
vector<vector<int>> edges={{1, 2},{1, 0},{2, 0},{2, 3},{2, 4}};
  for (auto &e : edges)
        addEdge(adj, e[0], e[1]);
 int s = 1; // Starting vertex for DFS
    cout << "DFS from source: " << s << endl;
    DFS(adj, s); // Perform DFS starting from the source vertex
  return 0;
}
```

Output:-



2)BFS

Breadth First Search (BFS) is a fundamental graph traversal algorithm. It begins with a node, then first traverses all its adjacent. Once all adjacent are visited, then their adjacent are traversed. This is different from DFS in a way that closest vertices are visited before others.

Code:

```
// C++ program for BFS of an undirected graph
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
// BFS from given source s
void bfs(vector<vector<int>>& adj, int s)
{
    // Create a queue for BFS
    queue<int> q;
```

```
    // Initially mark all the vertices as not visited
// When we push a vertex into the q, we mark it as
    // visited
vector<bool> visited(adj.size(), false);
    // Mark the source node as visited and
    // enqueue it
    visited[s] = true;
    q.push(s);
    // Iterate over the queue
    while (!q.empty()) {
        // Dequeue a vertex from queue and print it
        int curr = q.front();
        q.pop();
        cout << curr << " ";
        // Get all adjacent vertices of the dequeued
        // vertex curr If an adjacent has not been
        // visited, mark it visited and enqueue it
        for (int x : adj[curr]) {
            if (!visited[x]) {
                visited[x] = true;
                q.push(x);
            }
        }
    }
}
// Function to add an edge to the graph
void addEdge(vector<vector<int>>& adj, int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u); // Undirected Graph
}
int main()
{
```

```
// Number of vertices in the graph
    int V = 5;
  // Adjacency list representation of the graph
  vector<vector<int>> adj(V);
    // Add edges to the graph
    addEdge(adj, 0, 1);
    addEdge(adj, 0, 2);
    addEdge(adj, 1, 3);
    addEdge(adj, 1, 4);
    addEdge(adj, 2, 4);
    // Perform BFS traversal starting from vertex 0
    cout << "BFS starting from 0 : \n";
    bfs(adj, 0);
    return 0;
}
```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\Harshali> cd "c:\Users\Harshali\Downloads\" ; if ($?) { g++ BFS.cpp -o BFS } ; if ($?) { .\BFS }
BFS starting from 0 :
0 1 2 3 4
PS C:\Users\Harshali\Downloads>
```

Practical-10

AIM: Create a minimum spanning tree using Kruskal's Algorithm

In Kruskal's algorithm, sort all edges of the given graph in increasing order. Then it keeps on adding new edges and nodes in the MST if the newly added edge does not form a cycle. It picks the minimum weighted edge at first and the maximum weighted edge at last. Thus we can say that it makes a locally optimal choice in each step in order to find the optimal solution.

Complexity: O(E log E) or O(E log V), due to edge sorting.

<u>Code:</u>

```
// C++ program for Kruskal's algorithm to find Minimum
// Spanning Tree of a given connected, undirected and
// weighted graph
#include<bits/stdc++.h>
using namespace std;
// Creating shortcut for an integer pair
typedef pair<int, int> iPair;
// Structure to represent a graph
struct Graph
{
   int V, E;
   vector< pair<int, iPair> > edges;
   // Constructor
   Graph(int V, int E)
   {
      this->V = V;
      this->E = E;
   }
   // Utility function to add an edge
   void addEdge(int u, int v, int w)
   {
      edges.push_back({w, {u, v}});
   }
   // Function to find MST using Kruskal's
   // MST algorithm
```

```
    int kruskalMST();
};
// To represent Disjoint Sets
struct DisjointSets
{
    int *parent, *rnk;
    int n;
 // Constructor.
    DisjointSets(int n)
    {
     // Allocate memory
        this->n = n;
        parent = new int[n+1];
        rnk = new int[n+1];
// Initially, all vertices are in
 // different sets and have rank 0.
        for (int i = 0; i <= n; i++)
        {
            rnk[i] = 0;
 //every element is parent of itself
            parent[i] = i;
        }
    }
 // Find the parent of a node 'u'
 //Path Compression
    int find(int u)
    {
        /* Make the parent of the nodes in the path
        from u--> parent[u] point to parent[u] */
        if (u != parent[u])
            parent[u] = find(parent[u]);
        return parent[u];
    }
// Union by rank
```

```
   void merge(int x, int y)
{
x = find(x), y = find(y);
/* Make tree with smaller height
       a subtree of the other tree */
       if (rnk[x] > rnk[y])
          parent[y] = x;
       else // If rnk[x] <= rnk[y]
          parent[x] = y;
if (rnk[x] == rnk[y])
          rnk[y]++;
   } };
/* Functions returns weight of the MST*/
int Graph::kruskalMST()
{
  int mst_wt = 0; // Initialize result
// Sort edges in increasing order on basis of cost
   sort(edges.begin(), edges.end());
// Create disjoint sets
   DisjointSets ds(V);
 // Iterate through all sorted edges
   vector< pair<int, iPair> >::iterator it;
   for (it=edges.begin(); it!=edges.end(); it++)
   {
       int u = it->second.first;
       int v = it->second.second;
       int set_u = ds.find(u);
       int set_v = ds.find(v);
      // Check if the selected edge is creating
       // a cycle or not (Cycle is created if u
       // and v belong to same set)
       if (set_u != set_v)
       {
```

```
// Current edge will be in the MST
 // so print it
cout << u << " - " << v << endl;
 // Update MST weight
   mst_wt += it->first;
// Merge two sets
            ds.merge(set_u, set_v);
        } }
 return mst_wt;
}
// Driver program to test above functions
int main()
{
    /* Let us create above shown weighted
    and undirected graph */
    int V = 9, E = 14;
    Graph g(V, E);
// making above shown graph
 g.addEdge(0, 1, 4);
    g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8);
    g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);
    g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4);
    g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14);
    g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2);
    g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6);
    g.addEdge(7, 8, 7);
cout << "Edges of MST are \n";
```

```
int mst_wt = g.kruskalMST();
cout << "\nWeight of MST is " << mst_wt;
return 0;
}
```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\Harshali\Downloads> cd "c:\Users\Harshali\Downloads\" ; if ($?) { g++ Kruskal.cpp -o Kruskal } ; if ($?) { .\Kruskal }
Edges of MST are
6 - 7
2 - 8
5 - 6
0 - 1
2 - 5
2 - 3
0 - 7
3 - 4

Weight of MST is 37
PS C:\Users\Harshali\Downloads>
```