J.C BOSE UNIVERSITY OF SCIENCE AND TECHNOLOGY, Y.M.C.A

Advance data structure

Workshop FIle

NAME:: ABHISHEK KUMAR

ROLL NO:19001015003

BRANCH::ENC 7TH SEM

S.No	Date	Program	Page No.	Remarks
1.		Implementation of Dictionary ADT- Log file- a) Insertion b) Deletion c) Updating d) Searching e) Display.		
2.		Implementation of Look-up table- a) Insertion b) Deletion c) Updating d) Searching e) Display		
3.		Program to implement different applications using Divide and Conquer and Computation of time complexity-Binary Search, Ternary Search, Merge-sort, Quick-sort.		
4.		Program to implement randomized quick-sort and compute its complexity.		
5.		Program to implement Count-Sort and compute its complexity.		
6.		Program for Tree Traversal-Breadth First, Depth First.		
7.		Program to convert graphs into spanning tree: -Prim, Kruskal.		
8.		Program to find shortest path using Dijkstra's Algorithm.		
9.		Program to find All pair shortest path.		
10.		Program to implement Topological Sort.		
11.		Program to implement Maximum flow algorithm.		
12.		Implementation of Red Black tree.		
13.		Program to find Longest Common subsequence using heuristic technique.		
14.		Program to find Longest Common subsequence using Brute-Force technique.		
15.		Program to find Longest Common subsequence using Knuth Morris Pratt technique.		

1. Implementation of Dictionary ADT-Log file- a) Insertion b) Deletion c) Updating d) Searching e) Display.

```
#include <iostream>
#include <fstream>
#include <string>
#include <unordered_map>

using namespace std;
class Dictionary
{
public:

    Dictionary(const string &filename) : filename_(filename)
    {
        log_.open(filename, ios::app);
    }
     void insert(const string &key, const string &value)
    {
        log_ << "i " << key << " " << value << endl;
        entries_[key] = value;
    }
    void remove(const string &key)
    {
        log_ << "d " << key << endl;
        entries_.erase(key);
    }
    void update(const string &key, const string &value)
    {
        log_ << "u " << key << " " << value << endl;
        entries_[key] = value;
    }
     bool search(const string &key) const
    {
        return entries_.count(key) > 0;
    }
    void display() const
    {
        for (const auto &entry : entries_)
        {
            cout << entry.first << ": " << entry.second << endl;
        }
    }

private:

    string filename_;
```

```
    ofstream log_;
    unordered_map<string, string> entries_;
};

int main()
{
    Dictionary dict("dictionary.log");

    dict.insert("apple", "a fruit");
    dict.insert("banana", "another fruit");
    dict.insert("car", "a vehicle");

    dict.display();

    dict.remove("apple");

    dict.display();

    dict.update("car", "a mode of transportation");

    dict.display();

    cout << dict.search("apple") << endl;
    cout << dict.search("banana") << endl;
    cout << dict.search("car") << endl;

    return 0;
}
```

OUTPUT :

```
car: a vehicle
banana: another fruit
apple: a fruit
car: a vehicle
banana: another fruit
car: a mode of transportation
banana: another fruit
0
1
1
```

2: Implementation of Look-up table-

a) Insertion b) Deletion c) Updating d) Searching e) Display.

```
#include <bits/stdc++.h>
using namespace std;

class LookupLTABLE
{
public:
void insert(const string &key, const string &value)
{
   LTABLE_[key] = value;
}
void remove(const string &key)
{
   LTABLE_.erase(key);
}
void update(const string &key, const string &value)
{
    LTABLE_[key] = value;
}
string search(const string &key)
{
if (LTABLE_.count(key) == 0)
{
return "";
}
return LTABLE_[key];
}

void display()
{
  for (const auto &entry : LTABLE_)
{
cout << entry.first << ": " << entry.second << endl;
}
}

private:
    map<string, string> LTABLE_;
};

int main()
{
LookupLTABLE LTABLE;
LTABLE.insert("DOG", "ANIMAL");
LTABLE.insert("YAMAHA", "BIKE");
LTABLE.insert("TAJ MAHAL", "MONUMENT");
LTABLE.display();
cout << LTABLE.search("DOG") << endl;
```

```
LTABLE.update("YAMAHA", "VEHICLE");
LTABLE.remove("DOG");
LTABLE.display();

}
```

OUTPUT :

```
DOG: ANIMAL
TAJ MAHAL: MONUMENT
YAMAHA: BIKE
ANIMAL
TAJ MAHAL: MONUMENT
YAMAHA: VEHICLE
```

3: Program to implement different applications using Divide and Conquer and Computation of time complexity-Binary Search, Ternary Search, Merge-sort, Quick-sort.

a) **BINARY**

```
#include<iostream>
using namespace std;
int binary(int arr[],int l,int h,int key)
{ int mid;
     while(l<=h)
     {
        mid=(l+h)/2;
        if(arr[mid]==key)
        {
            return mid;
        }
        else if(arr[mid]>key)
        {
            h=mid-1;
        }
        else{
            l=mid+1;
        }
     }
     return -1;
}
int main()
{
    int arr[]={1,2,3,4,25,45,63};
    int len=sizeof(arr)/sizeof(arr[0]);
    cout<<binary(arr,0,len-1,63);
}
```

Output:

```
if ($?) { .\binaryseacrh }
6
```

TERNARY

```
#include<iostream>
#include<vector>
using namespace std;
 int  ternary(vector<int>arr,int start, int end,int key)
 {
    int  mid1;
    int  mid2;
       while (start<=end)
       {
          mid1=start +(end-start)/3;
          mid2= end -(end-start)/3;
          if(arr[mid1]==key)
          {
            return mid1;
          }
          else if(arr[mid2]==key)
          {
            return mid2;
          }
          else if(arr[mid1]>key)
          {
               end=mid1-1;
          }
          else if(arr[mid2]<key)
          {
            start=mid2+1;
          }
          else
          {
             start=mid1+1;
             end=mid2-1;
          }

       } return -1;}
 int main()
 {
    vector<int>arr={1,2,3,4,5,87,110};
    cout<<"Index of 87 ="<<ternary(arr,0,arr.size()-1,87);

 }
```

OUTPUT::

```
 .\ternary }
Index of 87 =5
```

MERGE SORT

```
#include <iostream>
#include <vector>
using namespace std;
void merge(vector<int> &arr, int s, int e)
{

        int mid = s + (e - s) / 2;
        int len1 = mid - s + 1;
        int len2 = e - mid;
        int *first = new int[len1];
        int *second = new int[len2];
        int mainArrIndex = s;
        for (int i = 0; i < len1; i++)
        {
                first[i] = arr[mainArrIndex++];
        }
        mainArrIndex = mid + 1;
        for (int i = 0; i < len2; i++)
        {
                second[i] = arr[mainArrIndex++];
        }
        //     merge the array
        int index1 = 0;
        int index2 = 0;
        mainArrIndex = s;
        while (index1 < len1 && index2 < len2)
        {
                if (first[index1] < second[index2])
                {
                        arr[mainArrIndex++] = first[index1++];
                }
                else
                {
                        arr[mainArrIndex++] = second[index2++];
                }
        }
        while (index1 < len1)
        {
                arr[mainArrIndex++] = first[index1++];
        }
        while (index2 < len2)
        {
                arr[mainArrIndex++] = second[index2++];
        }
        delete[]first;
        delete[]second;
```

```
}
void solve(vector<int> &arr, int s, int e)
{
        if (s >= e)
        {
                return;
        }
        int mid = s + (e - s) / 2;
        //     sort left side
        solve(arr, s, mid);
        //     sort right side
        solve(arr, mid + 1, e);
        merge(arr, s, e);
}

void mergeSort(vector<int> &arr, int n)
{
        int s = 0;
        int e = n - 1;
        solve(arr, s, e);
}

int main ()
{
        vector<int>arr={ 1,2,3,5,6,736,88,99};
        mergeSort(arr,8);
        for(int i=0;i<arr.size();i++)
        {
                cout<<arr[i];
        }
}
```

OUTPUT::

```
nerFile } ; if ($?) { .\tempCodeRunnerFile }
123568899736
```

}

QUICK SORT

```
#include<iostream>
using namespace std;
int partion(int arr[],int start,int end)
{
    int pivot=arr[start];
    int count=0;
     for(int i=start+1;i<=end;i++)
     { if(arr[i]<pivot)
        {count++;}
     }
     int pivotindex=start+count;
     swap(arr[pivotindex],arr[start]);
     int i=0,j=end;
     while(i<pivotindex && j>pivotindex)
     {
        while(arr[i]<arr[pivotindex])
        {
         i++;
        }
         while(arr[j]>arr[pivotindex])
        {
         j--;
        }
        if(i<pivotindex && j>pivotindex)
        {
            swap(arr[i++],arr[j--]);
        }
     }
     return pivotindex;

}
void  quick(int arr[],int start, int end)
{
    if(start>=end)
    {
        return;
    }
    int p=partion(arr,start,end);
    quick(arr,start,p);
    quick(arr,p+1,end);
}
int main()
{
    int arr[]={1,3,5,4};
    int len=sizeof(arr)/sizeof(arr[0]);
```

```
    quick(arr,0,len-1);
    for(int i=0;i<len;i++)
    {
        cout<<arr[i]<<" ";
    }
}
```

```
nnerFile } ; if ($?) { .\tempCodeRunnerFile }
1 3 4 5
```

4. Program to implement randomized quick-sort and compute its complexity.

```
#include <bits/stdc++.h>
#define MAX 100

using namespace std;

int Partition(int a[], int low, int high)
{
   int pivot, index, i;
   index = low;
   pivot = high;
   for (i = low; i < high; i++)
   {
      if (a[i] < a[pivot])
      {
         swap(a[i], a[index]);
         index++;
      }
   }
   swap(a[pivot], a[index]);
   return index;
}
void random_shuffle(int arr[], int n)
{
   srand(time(NULL));
   for (int i = n - 1; i > 0; i--)
   {
      int j = rand() % (i + 1);
      int temp = arr[i];
      arr[i] = arr[j];
      arr[j] = temp;
   }
}
int RandomPivotPartition(int a[], int low, int high)
{
   int pvt, n, temp;
   n = rand();
   pvt = low + n % (high - low + 1);
   swap(a[high], a[pvt]);
   return Partition(a, low, high);
}
void quick_sort(int arr[], int p, int q)
{
   int pindex;
   if (p < q)
   {
      pindex = RandomPivotPartition(arr, p, q);
```

```
        quick_sort(arr, p, pindex - 1);
        quick_sort(arr, pindex + 1, q);
    }
}
int main()
{
    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
        arr[i] = i + 1;
    random_shuffle(arr, n);
    quick_sort(arr, 0, n - 1);
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}
```

Time complexity – O(nlogn)

```
Sorted array :
[1, 5, 7, 8, 9, 10]
```

5. Program to implement Count-Sort and compute its complexity.

```
#include<iostream>
using namespace std;
void countsort(int arr[],int n)
{
    int max1=arr[0];
    for(int i=0;i<n;i++)
    {
        max1=max(max1,arr[i]);
    }
    int countarray[10]={0};
    for(int i=0;i<n;i++)
    {
        countarray[arr[i]]++;
    }
    for(int i=1;i<=max1;i++)
    {
        countarray[i]=countarray[i-1]+countarray[i];
    }
    int out[n];
    for(int i=n-1;i>=0;i--)
    { countarray[arr[i]]--;
        out[countarray[arr[i]]]=arr[i];
    }
    for(int i=0;i<n;i++)
    {
        arr[i]=out[i];
    }

}
int main()
{
    int arr[]={1,3,2,3,4,1,6,4,3};
    countsort(arr,9);
    for(int i=0;i<9;i++)
    {
        cout<<arr[i]<<" ";
    }
}
```

```
?) { .\countsort }
1 1 2 3 3 3 4 4 6
```

6. Program for Tree Traversal-Breadth First, Depth First.

Breadth first search

```
#include <iostream>
#include<queue>
using namespace std;
 struct node
 {
      int data;
         node *left ,*right;
         node(int key)
         {
                 data=key;
                 left=NULL;right=NULL;
         }
};
 void bfs(node * root )
 {
         if(root==NULL)
         {
            return;
         }
         queue<node*>q;
      q.push(root);
         while(q.empty()==false)
         {
         node * curr=q.front();
            q.pop();
            cout<<curr->data;
            if(curr->left)
            {
                q.push(curr->left);
            }
         if(curr->right)
            {
                q.push(curr->right);
            }
         }
          }
 int main(){
        struct node *root=new node(1);
        root->left=new node(2);
        root->right= new node(3);
        root->left->left=new node(4);
        root->right->right=new node (5);
     bfs(root);
  return 0;}
```

```
e }
12345
```

a) Depth First search

```
#include <iostream>
using namespace std;
struct Node
{
  int data;
  struct Node* left, *right;
  Node(int data)
  {
    this->data = data;
    left = right = NULL;
  }
};
void printPostorder(struct Node* node)
{
  if (node == NULL)
    return;
  printPostorder(node->left);
  printPostorder(node->right);
  cout << node->data << " ";
}
void printInorder(struct Node* node)
{
  if (node == NULL)
    return;
  printInorder(node->left);
  cout << node->data << " ";
  printInorder(node->right);
}
void printPreorder(struct Node* node)
{
  if (node == NULL)
    return;
  cout << node->data << " ";
  printPreorder(node->left);
  printPreorder(node->right);
}
int main()
{
  struct Node *root = new Node(1);
  root->left = new Node(2);
  root->right = new Node(3);
  root->left->left = new Node(4);
```

```
  root->left->right = new Node(5);

  cout << "\nPreorder traversal of binary tree is \n";
  printPreorder(root);

  cout << "\nInorder traversal of binary tree is \n";
  printInorder(root);

  cout << "\nPostorder traversal of binary tree is \n";
  printPostorder(root);

  return 0;
}
```

```
Preorder traversal of binary tree is
1 2 4 5 3
Inorder traversal of binary tree is
4 2 5 1 3
Postorder traversal of binary tree is
4 5 2 3 1
```

7. Program to convert graphs into spanning tree: Prims, Kruskal.

Prims Algorithm

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
   int N = 5, m = 6;
   vector<pair<int, int>> adj[N];

   adj[0].push_back({1, 2});
   adj[0].push_back({3, 6});
   adj[1].push_back({0, 2});
   adj[1].push_back({2, 3});
   adj[1].push_back({3, 8});
   adj[1].push_back({4, 5});
   adj[2].push_back({1, 3});
   adj[2].push_back({4, 7});
   adj[3].push_back({0, 6});
   adj[3].push_back({1, 8});
   adj[4].push_back({1, 5});
   adj[4].push_back({2, 7});

   int parent[N];
   int key[N];
   bool mstSet[N];

   for (int i = 0; i < N; i++)
      key[i] = INT_MAX, mstSet[i] = false;
   key[0] = 0;
   parent[0] = -1;
   int ansWeight = 0;
   for (int count = 0; count < N - 1; count++)
   {

      int mini = INT_MAX, u;

      for (int v = 0; v < N; v++)
      {
         if (mstSet[v] == false && key[v] < mini)
            mini = key[v], u = v;
      }
      mstSet[u] = true;

      for (auto it : adj[u])
      {
         int v = it.first;
```

```
            int weight = it.second;
            if (mstSet[v] == false && weight < key[v])
                parent[v] = u, key[v] = weight;
        }
    }

    for (int i = 1; i < N; i++)
        cout << parent[i] << " - " << i << " \n";
    return 0;
}
```

OUTPUT

```
0 - 1
1 - 2
0 - 3
1 - 4
```

Kruksals Algorithm

```
#include <bits/stdc++.h>
using namespace std;
```

```
struct node
{
   int u;
   int v;
   int wt;
   node(int first, int second, int weight)
   {
      u = first;
      v = second;
      wt = weight;
   }
};

bool comp(node a, node b)
{
   return a.wt < b.wt;
}

int findPar(int u, vector<int> &parent)
{
   if (u == parent[u])
      return u;
   return parent[u] = findPar(parent[u], parent);
}

void unionn(int u, int v, vector<int> &parent, vector<int> &rank)
{
   u = findPar(u, parent);
   v = findPar(v, parent);
   if (rank[u] < rank[v])
   {
      parent[u] = v;
   }
   else if (rank[v] < rank[u])
   {
      parent[v] = u;
   }
   else
   {
      parent[v] = u;
      rank[u]++;
   }
}
int main()
{
   int N = 5, m = 6;
   vector<node> edges;
   edges.push_back(node(0, 1, 2));
   edges.push_back(node(0, 3, 6));
```

```
    edges.push_back(node(1, 0, 2));
    edges.push_back(node(1, 2, 3));
    edges.push_back(node(1, 3, 8));
    edges.push_back(node(1, 4, 5));
    edges.push_back(node(2, 1, 3));
    edges.push_back(node(2, 4, 7));
    edges.push_back(node(3, 0, 6));
    edges.push_back(node(3, 1, 8));
    edges.push_back(node(4, 1, 5));
    edges.push_back(node(4, 2, 7));
    sort(edges.begin(), edges.end(), comp);

    vector<int> parent(N);
    for (int i = 0; i < N; i++)
        parent[i] = i;
    vector<int> rank(N, 0);

    int cost = 0;
    vector<pair<int, int>> mst;
    for (auto it : edges)
    {
        if (findPar(it.v, parent) != findPar(it.u, parent))
        {
            cost += it.wt;
            mst.push_back({it.u, it.v});
            unionn(it.u, it.v, parent, rank);
        }
    }
    cout << cost << endl;
    for (auto it : mst)
        cout << it.first << " - " << it.second << endl;
    return 0;
}
```

OUTPUT:

```
16
0 - 1
1 - 2
1 - 4
0 - 3
```

8. Program to find shortest path using Dijkstra's Algorithm.

```
#include <bits/stdc++.h>
using namespace std;
#define INF 0x3f3f3f3f
typedef pair<int, int> iPair;
class Graph {
	int V;
	list<pair<int, int> >* adj;
public:
	Graph(int V);
	void addEdge(int u, int v, int w);
	void shortestPath(int s);
};

Graph::Graph(int V)
{
	this->V = V;
	adj = new list<iPair>[V];
}

void Graph::addEdge(int u, int v, int w)
{
	adj[u].push_back(make_pair(v, w));
	adj[v].push_back(make_pair(u, w));
}
void Graph::shortestPath(int src)
{

	priority_queue<iPair, vector<iPair>, greater<iPair> >
		pq;
	vector<int> dist(V, INF);
	pq.push(make_pair(0, src));
	dist[src] = 0;


	while (!pq.empty()) {

		int u = pq.top().second;
		pq.pop();
		list<pair<int, int> >::iterator i;
		for (i = adj[u].begin(); i != adj[u].end(); ++i) {

			int v = (*i).first;
			int weight = (*i).second;

			if (dist[v] > dist[u] + weight) {
				dist[v] = dist[u] + weight;
```

```
                    pq.push(make_pair(dist[v], v));
                }
            }
        }

        printf("Vertex Distance from Source\n");
        for (int i = 0; i < V; ++i)
            printf("%d \t\t %d\n", i, dist[i]);
}

int main()
{
        int V = 9;
        Graph g(V);
        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);

        g.shortestPath(0);

        return 0;
}
```

OUTPUT:

```
Vertex Distance from Source
0          0
1          4
2          12
3          19
4          21
5          11
6          9
7          8
8          14
```

9. Program to find All pair shortest path

```
#include <iostream>
#include <iomanip>
#define N 7
#define INF 999
using namespace std;
// Cost matrix of the graph
int costMat[N][N] = {
   {0, 2, 6, INF, INF, INF, INF},
   {3, 1, 2, 1, 9, INF, INF},
   {6, 2, 0, 5, 4, 2, 2},
   {INF, 1, 1, 0, 2, INF, 4},
   {INF, INF, 4, 5, 0, 2, 1},
   {INF, INF, 2, INF, 2, 0, 1},
   {INF, INF, INF, 4, 1, 1, 0}};
void floydWarshal()
{
   int cost[N][N]; // defind to store shortest distance from any N to any N
   for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
         cost[i][j] = costMat[i][j]; // copy costMatrix to new matrix
   for (int k = 0; k < N; k++)
   {
      for (int i = 0; i < N; i++)
         for (int j = 0; j < N; j++)
            if (cost[i][k] + cost[k][j] < cost[i][j])
               cost[i][j] = cost[i][k] + cost[k][j];
   }
   cout << "The matrix:" << endl;
   for (int i = 0; i < N; i++)
   {
      for (int j = 0; j < N; j++)
         cout << setw(3) << cost[i][j];
      cout << endl;
   }
}
int main()
{
   floydWarshal();
}
```

OUTPUT:

```
The matrix:
  0  2  4  3  5  6  6
  3  1  2  1  3  4  4
  5  2  0  3  3  2  2
  4  1  1  0  2  3  3
  9  6  4  5  0  2  1
  7  4  2  5  2  0  1
  8  5  3  4  1  1  0
```

10: Implement Topological Sort

```
#include <bits/stdc++.h>
using namespace std;
vector<int> topoSort(int n, vector<int> adj[])
{
   vector<int> indegree(n, 0);
   for (int i = 0; i < n; i++)
   {
      for (auto it : adj[i])
      {
         indegree[it]++;
      }
   }
   queue<int> q;
   vector<int> ans;
   for (int i = 0; i < n; i++)
   {
      if (indegree[i] == 0)
         q.push(i);
   }
   while (!q.empty())
   {
      int currNode = q.front();
      q.pop();
      ans.push_back(currNode);
      for (auto it : adj[currNode])
      {
         indegree[it]--;
         if (indegree[it] == 0)
            q.push(it);
      }
   }
   return ans;
   // code here
}
int main()
{
   int n;
   cin >> n;
   vector<int> adj[n];
   int e;
   cin >> e;
   for (int i = 0; i < e; i++)
   {
      int u, v;
      cin >> u >> v;
```

```
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    vector<int> temp = topoSort(n, adj);
    for (auto it : temp)
        cout << it << " ";
    cout << endl;
}
```

OUTPUT:

For Input:

```
3 4
3 0
1 0
2 0
```

Your Output:

```
1 2 3 0
```

11. Program to implement Maximum flow algorithm.

```
#include <bits/stdc++.h>
using namespace std;

// Number of vertices in given graph
#define V 6

bool bfs(int rGraph[V][V], int s, int t, int parent[])
{
   bool visited[V] = {};

   queue<int> q;
   q.push(s);
   visited[s] = true;
   parent[s] = -1;

   while (!q.empty())
   {
      int u = q.front();
      q.pop();

      for (int v = 0; v < V; v++)
      {
         if (visited[v] == false && rGraph[u][v] > 0)
         {
            if (v == t)
            {
               parent[v] = u;
               return true;
            }
            q.push(v);
            parent[v] = u;
            visited[v] = true;
         }
      }
   }
   return false;
}

int fordFulkerson(int graph[V][V], int s, int t)
{
   int u, v;

   int rGraph[V][V];
   for (u = 0; u < V; u++)
      for (v = 0; v < V; v++)
         rGraph[u][v] = graph[u][v];
```

```
    int parent[V];

    int max_flow = 0;
    while (bfs(rGraph, s, t, parent))
    {

        int path_flow = INT_MAX;
        for (v = t; v != s; v = parent[v])
        {
            u = parent[v];
            path_flow = min(path_flow, rGraph[u][v]);
        }

        for (v = t; v != s; v = parent[v])
        {
            u = parent[v];
            rGraph[u][v] -= path_flow;
            rGraph[v][u] += path_flow;
        }

        max_flow += path_flow;
    }
    return max_flow;
}
int main()
{
    int src = 0, sink = 4;
    int graph[V][V] = {{0, 20, 13, 0, 0, 0},
                       {0, 0, 10, 12, 0, 0},
                       {0, 4, 0, 4, 14, 0},
                       {0, 0, 9, 0, 0, 20},
                       {0, 0, 0, 7, 6, 4},
                       {0, 1, 0, 3, 0, 0}};

    cout << "The maximum possible flow is "
         << fordFulkerson(graph, src, );
}
```

OUTPUT:

```
The source vertex is 0
The sink vertex is 4
The maximum possible flow is 14
```

12. Implementation of Red Black tree.

```
#include <iostream>

using namespace std;

struct node
{
   int key;
   node *parent;
   char color;
   node *left;
   node *right;
};
class RBtree
{
   node *root;
   node *q;

public:
   RBtree()
   {
      q = NULL;
      root = NULL;
   }
   void insert();
   void insertfix(node *);
   void leftrotate(node *);
   void rightrotate(node *);
   void del();
   node *successor(node *);
   void delfix(node *);
   void disp();
   void display(node *);
   void search();
};
void RBtree::insert()
{
   int z, i = 0;
   cout << "\nEnter key of the node to be inserted: ";
   cin >> z;
   node *p, *q;
   node *t = new node;
   t->key = z;
   t->left = NULL;
   t->right = NULL;
   t->color = 'r';
   p = root;
   q = NULL;
```

```
    if (root == NULL)
    {
        root = t;
        t->parent = NULL;
    }
    else
    {
        while (p != NULL)
        {
            q = p;
            if (p->key < t->key)
                p = p->right;
            else
                p = p->left;
        }
        t->parent = q;
        if (q->key < t->key)
            q->right = t;
        else
            q->left = t;
    }
    insertfix(t);
}
void RBtree::insertfix(node *t)
{
    node *u;
    if (root == t)
    {
        t->color = 'b';
        return;
    }
    while (t->parent != NULL && t->parent->color == 'r')
    {
        node *g = t->parent->parent;
        if (g->left == t->parent)
        {
            if (g->right != NULL)
            {
                u = g->right;
                if (u->color == 'r')
                {
                    t->parent->color = 'b';
                    u->color = 'b';
                    g->color = 'r';
                    t = g;
                }
            }
            else
            {
```

```
            if (t->parent->right == t)
            {
                t = t->parent;
                leftrotate(t);
            }
            t->parent->color = 'b';
            g->color = 'r';
            rightrotate(g);
        }
    }
    else
    {
        if (g->left != NULL)
        {
            u = g->left;
            if (u->color == 'r')
            {
                t->parent->color = 'b';
                u->color = 'b';
                g->color = 'r';
                t = g;
            }
        }
        else
        {
            if (t->parent->left == t)
            {
                t = t->parent;
                rightrotate(t);
            }
            t->parent->color = 'b';
            g->color = 'r';
            leftrotate(g);
        }
    }
    root->color = 'b';
  }
}

void RBtree::del()
{
  if (root == NULL)
  {
    cout << "\nEmpty Tree.";
    return;
  }
  int x;
  cout << "\nEnter the key of the node to be deleted: ";
  cin >> x;
```

```
node *p;
p = root;
node *y = NULL;
node *q = NULL;
int found = 0;
while (p != NULL && found == 0)
{
   if (p->key == x)
      found = 1;
   if (found == 0)
   {
      if (p->key < x)
         p = p->right;
      else
         p = p->left;
   }
}
if (found == 0)
{
   cout << "\nElement Not Found.";
   return;
}
else
{
   cout << "\nDeleted Element: " << p->key;
   cout << "\nColour: ";
   if (p->color == 'b')
      cout << "Black\n";
   else
      cout << "Red\n";

   if (p->parent != NULL)
      cout << "\nParent: " << p->parent->key;
   else
      cout << "\nThere is no parent of the node.  ";
   if (p->right != NULL)
      cout << "\nRight Child: " << p->right->key;
   else
      cout << "\nThere is no right child of the node.  ";
   if (p->left != NULL)
      cout << "\nLeft Child: " << p->left->key;
   else
      cout << "\nThere is no left child of the node.  ";
   cout << "\nNode Deleted.";
   if (p->left == NULL || p->right == NULL)
      y = p;
   else
      y = successor(p);
   if (y->left != NULL)
```

```
        q = y->left;
      else
      {
        if (y->right != NULL)
          q = y->right;
        else
          q = NULL;
      }
      if (q != NULL)
        q->parent = y->parent;
      if (y->parent == NULL)
        root = q;
      else
      {
        if (y == y->parent->left)
          y->parent->left = q;
        else
          y->parent->right = q;
      }
      if (y != p)
      {
        p->color = y->color;
        p->key = y->key;
      }
      if (y->color == 'b')
        delfix(q);
    }
}

void RBtree::delfix(node *p)
{
    node *s;
    while (p != root && p->color == 'b')
    {
      if (p->parent->left == p)
      {
        s = p->parent->right;
        if (s->color == 'r')
        {
          s->color = 'b';
          p->parent->color = 'r';
          leftrotate(p->parent);
          s = p->parent->right;
        }
        if (s->right->color == 'b' && s->left->color == 'b')
        {
          s->color = 'r';
          p = p->parent;
        }
```

```
        else
        {
            if (s->right->color == 'b')
            {
                s->left->color == 'b';
                s->color = 'r';
                rightrotate(s);
                s = p->parent->right;
            }
            s->color = p->parent->color;
            p->parent->color = 'b';
            s->right->color = 'b';
            leftrotate(p->parent);
            p = root;
        }
    }
    else
    {
        s = p->parent->left;
        if (s->color == 'r')
        {
            s->color = 'b';
            p->parent->color = 'r';
            rightrotate(p->parent);
            s = p->parent->left;
        }
        if (s->left->color == 'b' && s->right->color == 'b')
        {
            s->color = 'r';
            p = p->parent;
        }
        else
        {
            if (s->left->color == 'b')
            {
                s->right->color = 'b';
                s->color = 'r';
                leftrotate(s);
                s = p->parent->left;
            }
            s->color = p->parent->color;
            p->parent->color = 'b';
            s->left->color = 'b';
            rightrotate(p->parent);
            p = root;
        }
    }
    p->color = 'b';
    root->color = 'b';
```

```
    }
}

void RBtree::leftrotate(node *p)
{
    if (p->right == NULL)
        return;
    else
    {
        node *y = p->right;
        if (y->left != NULL)
        {
            p->right = y->left;
            y->left->parent = p;
        }
        else
            p->right = NULL;
        if (p->parent != NULL)
            y->parent = p->parent;
        if (p->parent == NULL)
            root = y;
        else
        {
            if (p == p->parent->left)
                p->parent->left = y;
            else
                p->parent->right = y;
        }
        y->left = p;
        p->parent = y;
    }
}
void RBtree::rightrotate(node *p)
{
    if (p->left == NULL)
        return;
    else
    {
        node *y = p->left;
        if (y->right != NULL)
        {
            p->left = y->right;
            y->right->parent = p;
        }
        else
            p->left = NULL;
        if (p->parent != NULL)
            y->parent = p->parent;
        if (p->parent == NULL)
```

```
            root = y;
        else
        {
            if (p == p->parent->left)
                p->parent->left = y;
            else
                p->parent->right = y;
        }
        y->right = p;
        p->parent = y;
    }
}

node *RBtree::successor(node *p)
{
    node *y = NULL;
    if (p->left != NULL)
    {
        y = p->left;
        while (y->right != NULL)
            y = y->right;
    }
    else
    {
        y = p->right;
        while (y->left != NULL)
            y = y->left;
    }
    return y;
}

void RBtree::disp()
{
    display(root);
}
void RBtree::display(node *p)
{
    if (root == NULL)
    {
        cout << "\nEmpty Tree.";
        return;
    }
    if (p != NULL)
    {
        cout << "\n\t NODE: ";
        cout << "\n Key: " << p->key;
        cout << "\n Colour: ";
        if (p->color == 'b')
            cout << "Black";
```

```
        else
            cout << "Red";
        if (p->parent != NULL)
            cout << "\n Parent: " << p->parent->key;
        else
            cout << "\n There is no parent of the node.  ";
        if (p->right != NULL)
            cout << "\n Right Child: " << p->right->key;
        else
            cout << "\n There is no right child of the node.  ";
        if (p->left != NULL)
            cout << "\n Left Child: " << p->left->key;
        else
            cout << "\n There is no left child of the node.  ";
        cout << endl;
        if (p->left)
        {
            cout << "\n\nLeft:\n";
            display(p->left);
        }
        /*else
         cout<<"\nNo Left Child.\n";*/
        if (p->right)
        {
            cout << "\n\nRight:\n";
            display(p->right);
        }
        /*else
         cout<<"\nNo Right Child.\n"*/
    }
}
void RBtree::search()
{
    if (root == NULL)
    {
        cout << "\nEmpty Tree\n";
        return;
    }
    int x;
    cout << "\n Enter key of the node to be searched: ";
    cin >> x;
    node *p = root;
    int found = 0;
    while (p != NULL && found == 0)
    {
        if (p->key == x)
            found = 1;
        if (found == 0)
        {
```

```
            if (p->key < x)
                p = p->right;
            else
                p = p->left;
        }
    }
    if (found == 0)
        cout << "\nElement Not Found.";
    else
    {
        cout << "\n\t FOUND NODE: ";
        cout << "\n Key: " << p->key;
        cout << "\n Colour: ";
        if (p->color == 'b')
            cout << "Black";
        else
            cout << "Red";
        if (p->parent != NULL)
            cout << "\n Parent: " << p->parent->key;
        else
            cout << "\n There is no parent of the node.  ";
        if (p->right != NULL)
            cout << "\n Right Child: " << p->right->key;
        else
            cout << "\n There is no right child of the node.  ";
        if (p->left != NULL)
            cout << "\n Left Child: " << p->left->key;
        else
            cout << "\n There is no left child of the node.  ";
        cout << endl;
    }
}
int main()
{
    int ch, y = 0;
    RBtree obj;
    do
    {
        cout << "\n\t RED BLACK TREE ";
        cout << "\n 1. Insert in the tree ";
        cout << "\n 2. Delete a node from the tree";
        cout << "\n 3. Search for an element in the tree";
        cout << "\n 4. Display the tree ";
        cout << "\n 5. Exit ";
        cout << "\nEnter Your Choice: ";
        cin >> ch;
        switch (ch)
        {
        case 1:
```

```
        obj.insert();
        cout << "\nNode Inserted.\n";
        break;
      case 2:
        obj.del();
        break;
      case 3:
        obj.search();
        break;
      case 4:
        obj.disp();
        break;
      case 5:
        y = 1;
        break;
      default:
        cout << "\nEnter a Valid Choice.";
      }
      cout << endl;

    } while (y != 1);
    return 1;
}
```

OUTPUT:

```
RED BLACK TREE
 1. Insert in the tree
 2. Delete a node from the tree
 3. Search for an element in the tree
 4. Display the tree
 5. Exit
Enter Your Choice: 1
Enter key of the node to be inserted: 5
Node Inserted.


     RED BLACK TREE
 1. Insert in the tree
 2. Delete a node from the tree
 3. Search for an element in the tree
 4. Display the tree
 5. Exit
Enter Your Choice: 4
```

13:Program to find Longest Common subsequence using heuristic technique.

```
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;

string LCS(string X, string Y)
{
   int m = X.length();
   int n = Y.length();

   int L[m + 1][n + 1];

   for (int i = 0; i <= m; i++)
   {
      for (int j = 0; j <= n; j++)
      {
         if (i == 0 || j == 0)
            L[i][j] = 0;

         else if (X[i - 1] == Y[j - 1])
            L[i][j] = L[i - 1][j - 1] + 1;

         else
            L[i][j] = max(L[i - 1][j], L[i][j - 1]);
      }
   }

   int index = L[m][n];

   string lcs(index + 1, ' ');

   int i = m, j = n;
   while (i > 0 && j > 0)
   {
      if (X[i - 1] == Y[j - 1])
      {
         lcs[index - 1] = X[i - 1]; // put current character in result
         i--;
         j--;
         index--;        }

      else if (L[i - 1][j] > L[i][j - 1])
         i--;
      else
         j--;
```

```
    }

    return lcs;
}

int main()
{
    string X, Y;

    cout << "Enter first string: ";
    cin >> X;
    cout << "Enter second string: ";
    cin >> Y;
    string lcs = LCS(X, Y);
    cout << "The longest common subsequence is: " << lcs << endl;

    return 0;
}
```

OUTPUT:

```
Enter first string: structure
Enter second string: struct
The longest common subsequence is: struct
```

Q14. Program to find Longest Common subsequence using Brute-Force technique.

```
#include <bits/stdc++.h>
  using namespace std;
  int main()
  {

  int t;cin>>t;
  while(t--)
  {
  string s1,s2;
  cin>>s1>>s2;
  int dp[s1.length()+1][s2.length()+1];
  int n=s1.length(),m=s2.length();
  for(int i=0;i<=n;i++)
  {
   for(int j=0;j<=m;j++)
   if(i==0||j==0)
   dp[i][j]=0;
   else
   {
    if(s1[i-1]==s2[j-1])
    dp[i][j]=dp[i-1][j-1]+1;
    else
    dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
   }
  }
  cout<<dp[n][m]<<"\n";
  }
   return 0;
  }
```

```
6
abcabcabc
abc
3
```

15. Program to find the Longest Common subsequence using Knuth Morris Pratt technique.

```
#include <bits/stdc++.h>
void computeLPSArray(char* pat, int M, int* lps);

// Prints occurrences of txt[] in pat[]
void KMPSearch(char* pat, char* txt)
{
	int M = strlen(pat);
	int N = strlen(txt);

	// create lps[] that will hold the longest prefix suffix
	// values for pattern
	int lps[M];

	// Preprocess the pattern (calculate lps[] array)
	computeLPSArray(pat, M, lps);

	int i = 0; // index for txt[]
	int j = 0; // index for pat[]
	while ((N - i) >= (M - j)) {
		if (pat[j] == txt[i]) {
			j++;
			i++;
		}

		if (j == M) {
			printf("Found pattern at index %d ", i - j);
			j = lps[j - 1];
		}

		// mismatch after j matches
		else if (i < N && pat[j] != txt[i]) {
			// Do not match lps[0..lps[j-1]] characters,
			// they will match anyway
			if (j != 0)
				j = lps[j - 1];
			else
				i = i + 1;
		}
	}
}

// Fills lps[] for given pattern pat[0..M-1]
void computeLPSArray(char* pat, int M, int* lps)
```

```
{
    // length of the previous longest prefix suffix
    int len = 0;

    lps[0] = 0; // lps[0] is always 0

    // the loop calculates lps[i] for i = 1 to M-1
    int i = 1;
    while (i < M) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        }
        else // (pat[i] != pat[len])
        {

            if (len != 0) {
                len = lps[len - 1];

                // Also, note that we do not increment
                // i here
            }
            else // if (len == 0)
            {
                lps[i] = 0;
                i++;
            }
        }
    }

int main()
{
    char txt[] = "ABABDABACDABABCABAB";
    char pat[] = "ABABCABAB";
    KMPSearch(pat, txt);
    return 0;
}
```

OUTPUT:

```
Found pattern at index 10
```