

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chennai-603203.

FACULTY OF ENGINEERING ANDTECHNOLOGY

**Department of Data Science and Business Systems**

Academic Year (2022–2023)

18CSC361J – Design & Analysis of Algorithms

SEMESTER–V



### SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chennai-603203.

FACULTY OF ENGINEERING AND TECHNOLOGY

18CSC361J –Design and Analysis of Algorithms

REG.No:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| R | A | 2 | 0 | 1 | 1 | 0 | 4 | 2 | 0 | 1 | 0 | 1 | 3 | 1 |

BONAFIDE CERTIFICATE

Certified that this is the bonafide record of work done by Sahil Anwar of V semester B.Tech COMPUTER SCIENCE AND ENGINEERING during the academic year 2022-2023 in the 18CSC361J –Design and analysis of algorithms Laboratory.

Dr.Prabhu kavin B HOD

Staﬀ -Incharge

Submitted for the practical examination held on 26/11/2022 at SRM Institute of Science and Technology, Kattankulathur, Chennai-603203.

Examiner-1 Examiner-2

# Lab 1: Insertion Sort

###### AIM:- To implement the insertion sort by using C language.

**Algorithm:-**

The simple steps of achieving the insertion sort are listed as follows

**Step 1 -** If the element is the first element, assume that it is already sorted. Return 1.

**Step2 -** Pick the next element, and store it separately in a **key. Step3 -** Now, compare the **key** with all elements in the sorted array.

**Step 4 -** If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

**Step 5 -** Insert the value.

**Step 6 -** Repeat until the array is sorted.

###### Code:-

#include <math.h> #include <stdio.h>

void insertionSort(**int** arr[], **int** n)

{

**int** i, key, j;

**for** (i = 1; i < n; i++) { key = arr[i];

j = i - 1;

**while** (j >= 0 && arr[j] > key) { arr[j + 1] = arr[j];

arr[j + 1] = key;

}

**void** printArray(**int** arr[], **int** n)

{

**int** i;

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

}

**int** main()

{

**int** arr[] = { 12, 11, 13, 5, 6 };

**int** n = **sizeof**(arr) / **sizeof**(arr[0]);

insertionSort(arr, n); printArray(arr, n);

return 0;

}

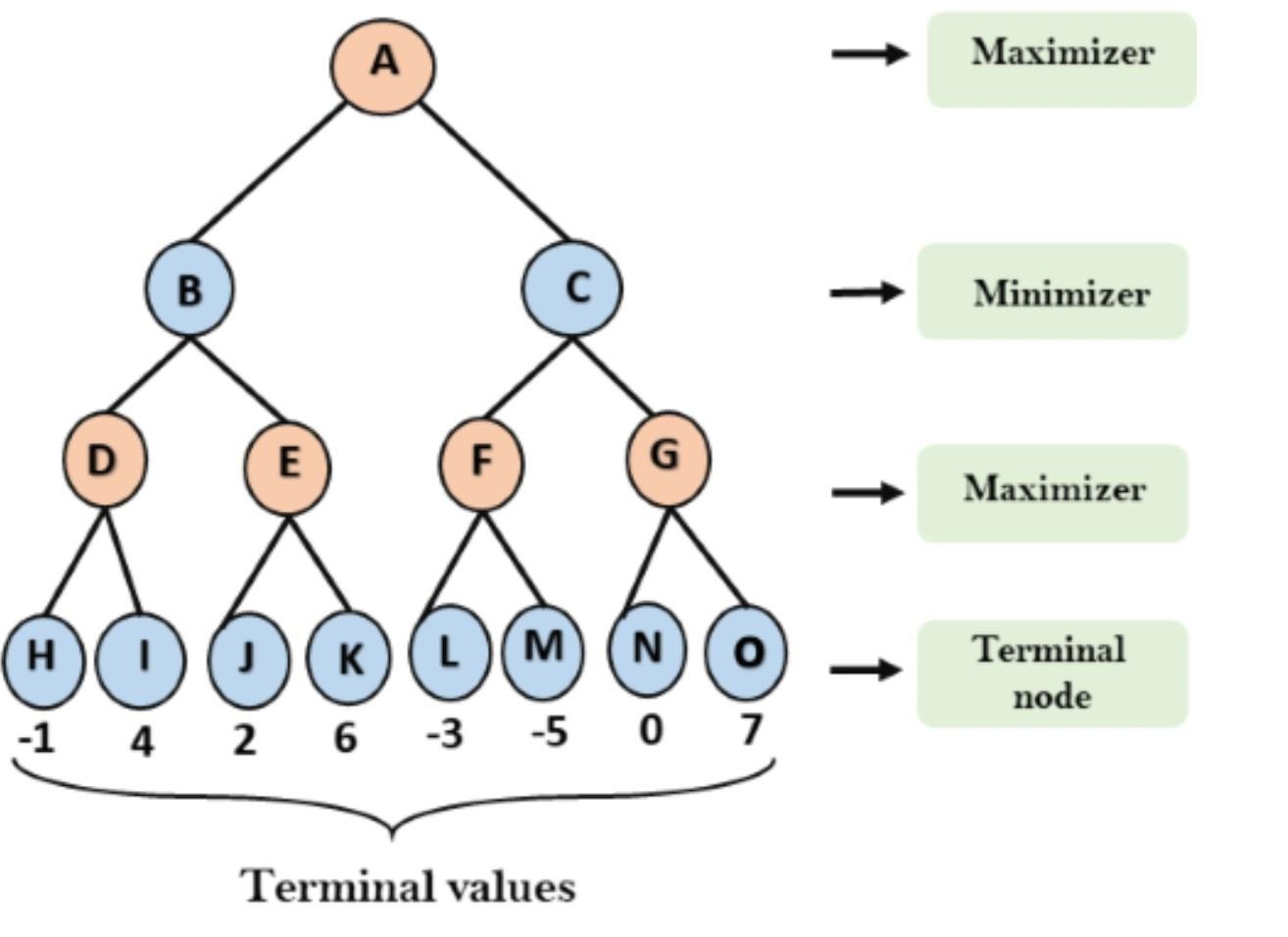
**Result: Insertion sort by using C language is successfully implemented.**

# Lab 2:Min Max Algorithm

**AIM:-** To implement the min max algorithm by using C language.

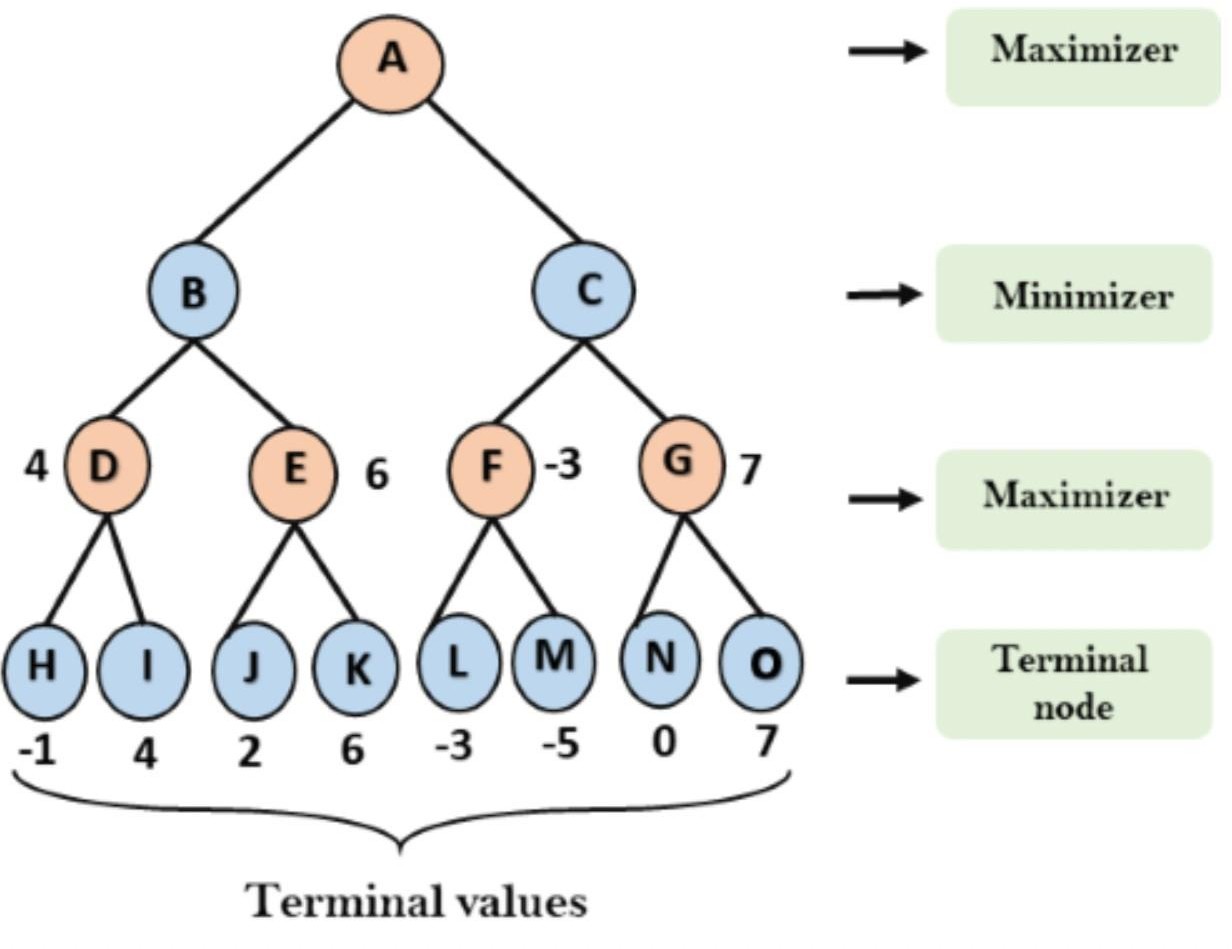
**ALGORITHM:**-

**Step-1:** In the first step, the algorithm generates the entire game- tree and apply the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimizer will take next turn which has worst-case initial value = +infinity.



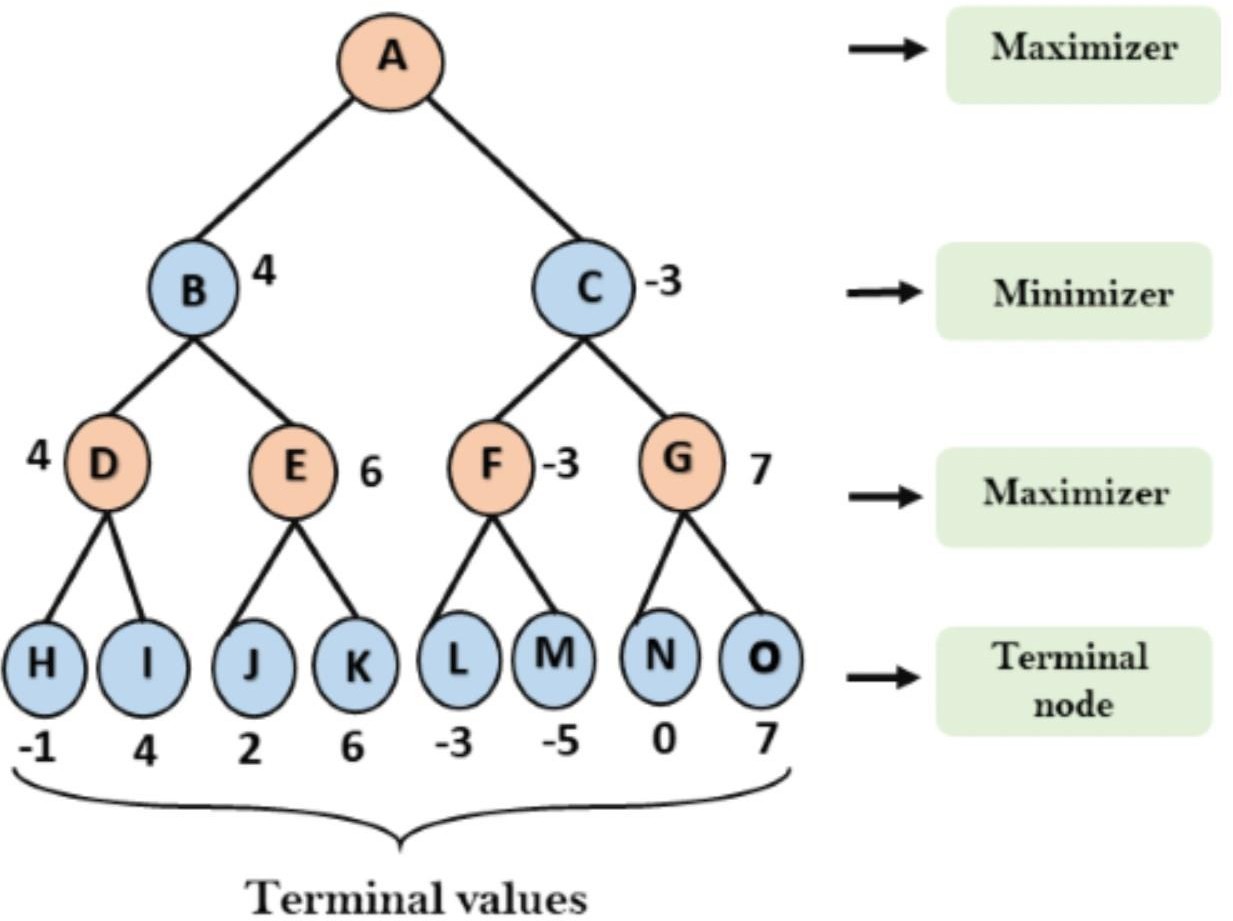
**Step-2:** Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

* For node D max(-1,- -∞) => max(-1,4)= 4
* For Node E max(2, -∞) => max(2, 6)= 6
* For Node F max(-3, -∞) => max(-3,-5) = -3
* For node G max(0, -∞) = max(0, 7) = 7



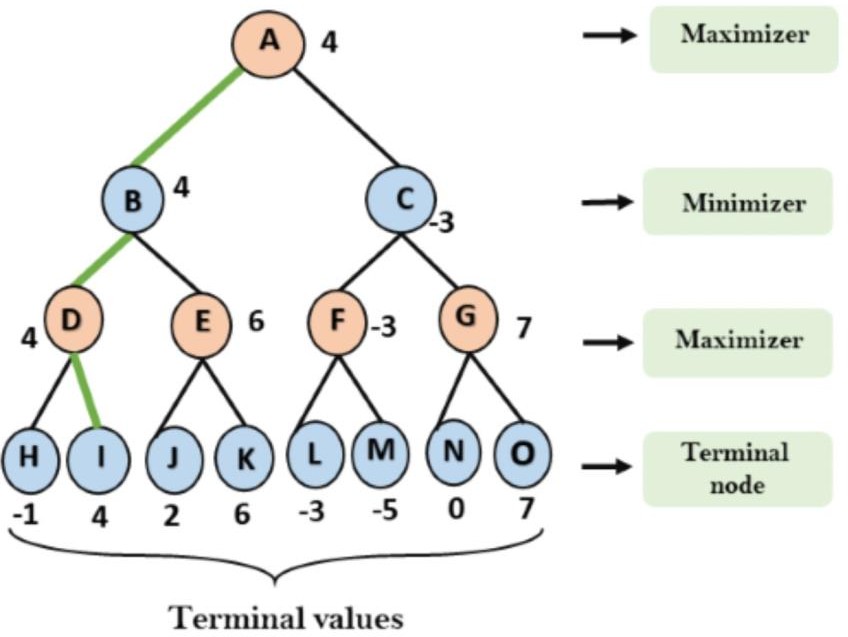
**Step-3:** In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

* For node B= min(4,6) = 4
* For node C= min (-3, 7) = -3



**Step-4:** Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

* For node A max(4, -3)= 4



#### PSUDO-CODE:-

1. function minimax(node, depth, maximizingPlayer) i s
2. **if** depth ==0 or node is a terminal node then
3. **return static** evaluation of node

4.

1. **if** MaximizingPlayer then
2. maxEva= -infinity
3. **for** each child of node **do**
4. eva= minimax(child, depth-1, **false**)
5. maxEva= max(maxEva,eva) // gives Maximum of the values
6. return maxEva
7. **else** // for Minimizer player
8. minEva= +infinity
9. **for** each child of node **do**
10. eva= minimax(child, depth-1, **true**)
11. minEva= min(minEva, eva) // gives minimum of the values
12. **return** minEva

**CODE-**

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

**int** minimax(**int** depth, **int** nodeIndex, **bool** isMax,

**int** scores[], **int** h)

{

**if** (depth == h)

**return** scores[nodeIndex];

**if** (isMax)

**return** max(minimax(depth+1, nodeIndex\*2, **false**, scores, h), minimax(depth+1, nodeIndex\*2 + 1, **false**, scores, h));

**else**

**return** min(minimax(depth+1, nodeIndex\*2, **true**, scores, h), minimax(depth+1, nodeIndex\*2 + 1, **true**, scores, h));

}

**int** log2(**int** n)

{

**return** (n==1)? 0 : 1 + log2(n/2);

}

**int** main()

{

**int** scores[] = {3, 5, 2, 9, 12, 5, 23, 23};

**int** n = **sizeof**(scores)/**sizeof**(scores[0]); **int** h = log2(n);

**int** res = minimax(0, 0, **true**, scores, h);

cout << "The optimal value is : " << res << endl; return ;

}

**OUTPUT:-**

The optimal value is : 12

**Result:-**Min Max algorithm by using C language is successfully implemented.

# Lab 3: Tower of Hanoi

**AIM:** To implement the Tower of Hanoi problem by using C++ language.

**Problem:**

Tower of Hanoi is a mathematical puzzle where we have three rods (A, B, and C) and N disks. Initially, all the disks are stacked in decreasing value of diameter i.e., the smallest disk is placed on the top and they are on rod A. The objective of the puzzle is to move the entire stack to another rod (here considered C), obeying the following simple rules:

* Only one disk can be moved at a time.
* Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
* No disk may be placed on top of a smaller disk.

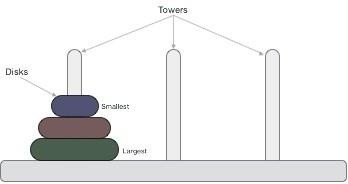
**AlGORITHM:**

To write an algorithm for Tower of Hanoi, first we need to learn how

to solve this problem with lesser amount of disks, say → 1 or 2. We mark three towers with name, source, destination and aux (only to help moving the disks). If we have only one disk, then it can easily

be moved from source to destination peg. If we have 2 disks −

* + First, we move the smaller (top) disk to aux peg.
  + Then, we move the larger (bottom) disk to destination peg.
  + And finally, we move the smaller disk from aux to destination peg.



So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (nth disk) is in one part and all other (n-1) disks are in the second part.

Our ultimate aim is to move disk **n** from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

The steps to follow are −

**Step 1 − Move n-1 disks from source to aux Step 2 − Move nth disk from source to dest Step 3 − Move n-1 disks from aux to dest**

**CODE:**

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

**void** towerOfHanoi(**int** n, **char** from\_rod, **char** to\_rod,

**char** aux\_rod)

{

**if** (n == 0) {

**return**;

}

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

cout << "Move disk " << n << " from rod " << from\_rod

<< " to rod " << to\_rod << endl;

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

}

**int** main()

{

**int** N = 3;

// A, B and C are names of rods towerOfHanoi(N, 'A', 'C', 'B'); return 0;

}

**OUTPUT:**

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

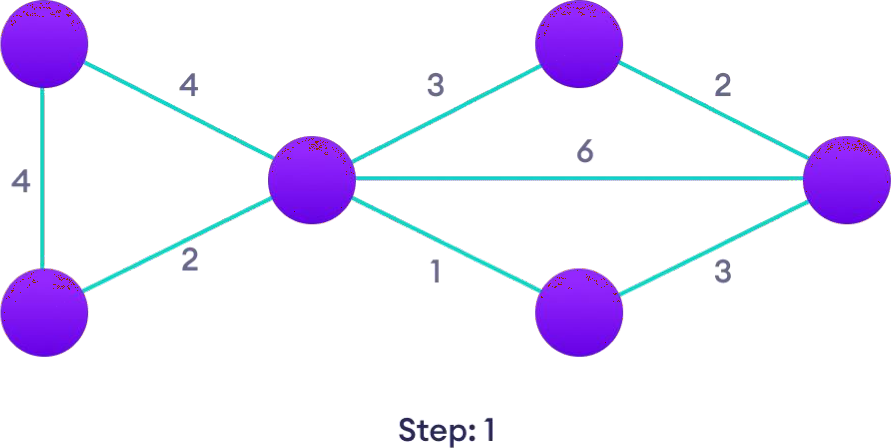
**RESULT: The Tower of Hanoi is successfully implemented using c++ language.**

**Lab 4:Dijkstra’s Shortest Path Algorithm**

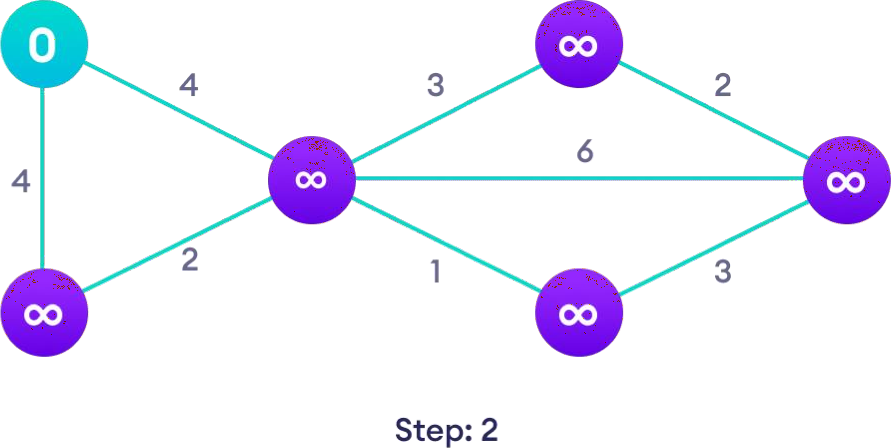
**AIM**: To implement the Dijkstra’s shortest path algorithm using c++ language.

### Algorithm:-

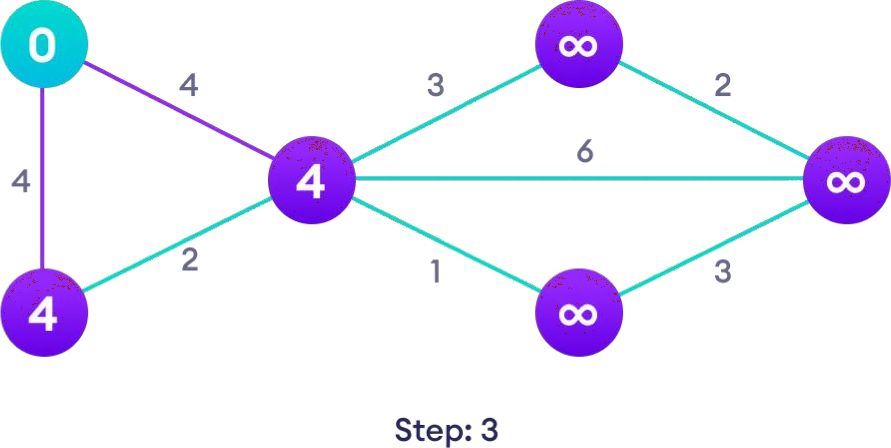
It is easier to start with an example and then think about the algorithm.



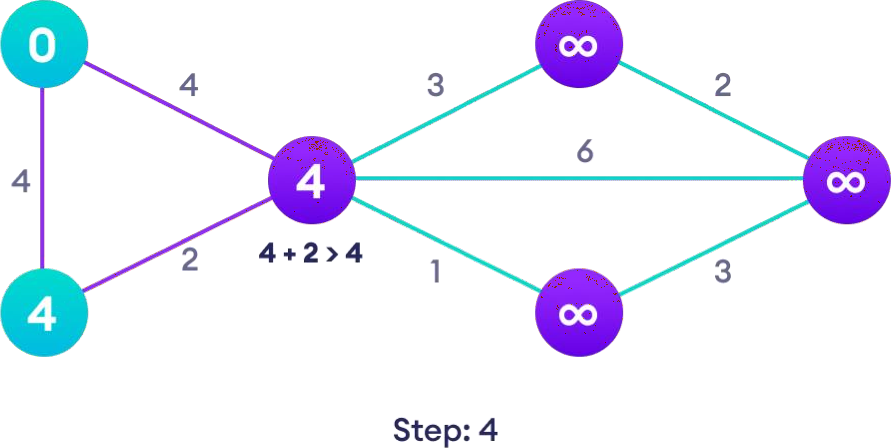
Start with a weighted graph



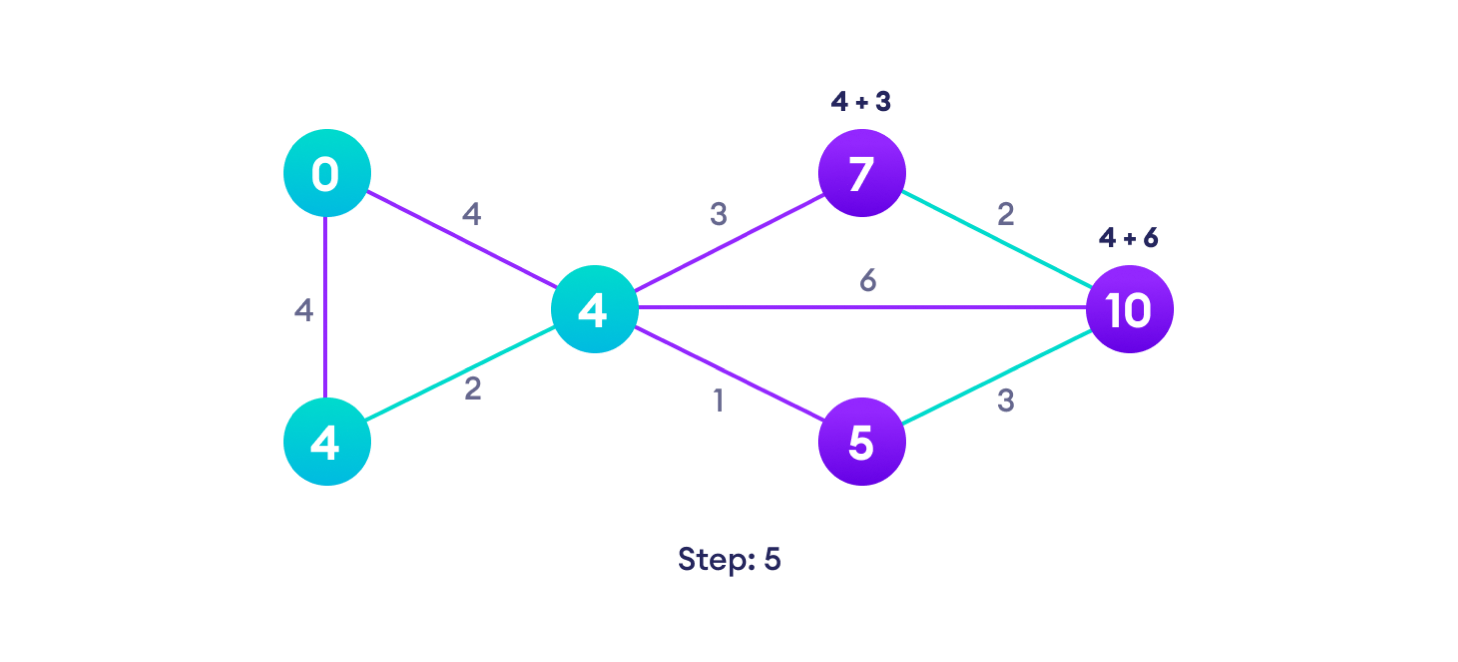
Choose a starting vertex and assign infinity path values to all other devices



Go to each vertex and update its path length

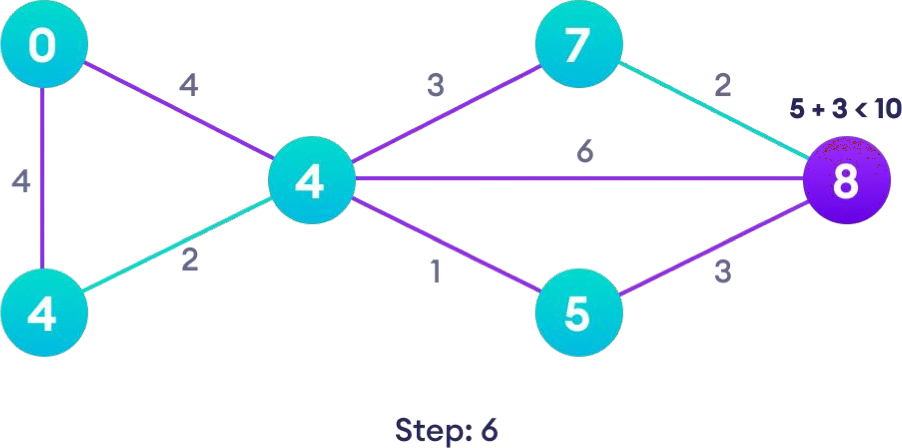


If the path length of the adjacent vertex is lesser than new path length,



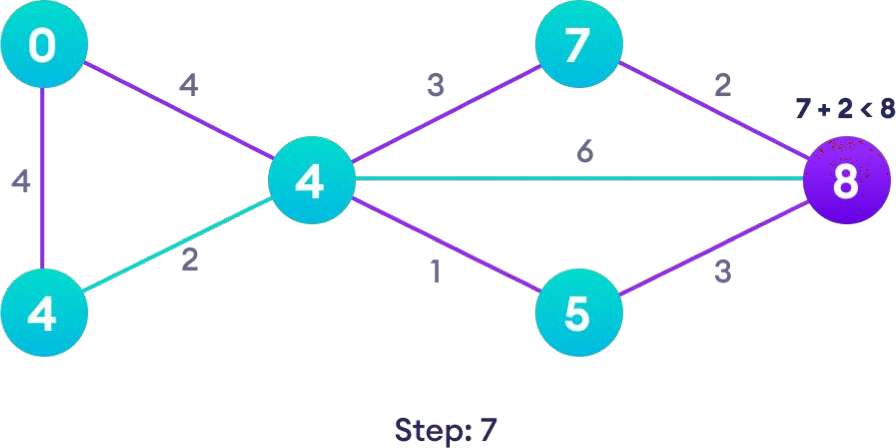
don't update it

Avoid updating path lengths of already visited vertices

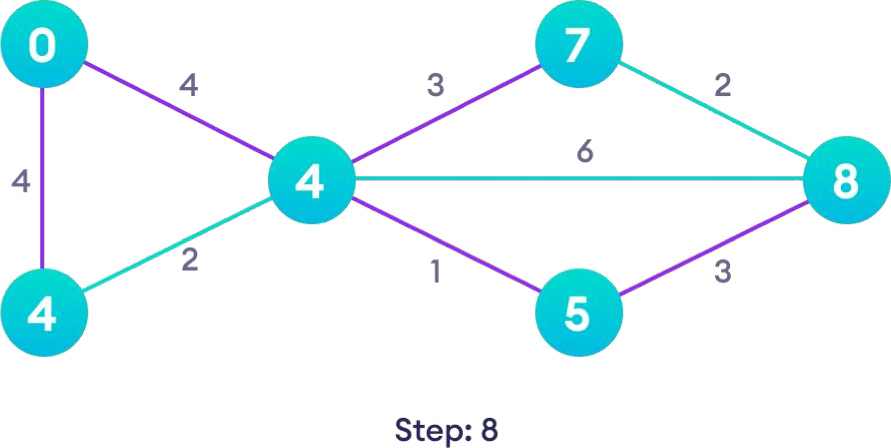


After each iteration, we pick the unvisited vertex with the least path length.

So we choose 5 before 7



Notice how the rightmost vertex has its path length updated twice



Repeat until all the vertices have been visited

#### PSEUDOCODE:

function dijkstra(G, S) for each vertex V in G

distance[V] <- infinite previous[V] <- NULL

If V != S, add V to Priority Queue Q

distance[S] <- 0

while Q IS NOT EMPTY

U <- Extract MIN from Q

for each unvisited neighbour V of U

tempDistance <- distance[U] + edge\_weight(U, V)

if tempDistance < distance[V]

distance[V] <- tempDistance

previous[V] <- U

return distance[], previous[]

##### CODE:-

#include<iostream>

#include<stdio.h>

using namespace std; #define INFINITY 9999

#define max 5

void dijkstra(int G[max][max],int n,int startnode); int main() {

int G[max][max]={{0,1,0,3,10},{1,0,5,0,0},{0,5,0,2,1},{3,0,2,0,6},

{10,0,1,6,0}}; int n=5;

int u=0; dijkstra(G,n,u);

return 0; }

void dijkstra(int G[max][max],int n,int startnode) { int cost[max] [max],distance[max],pred[max];

int visited[max],count,mindistance,nextnode,i,j; for(i=0;i<n;i++) for(j=0;j<n;j++) if(G[i][j]==0)

cost[i][j]=INFINITY; else

cost[i][j]=G[i][j]; for(i=0;i<n;i++) { distance[i]=cost[startnode][i]; pred[i]=startnode; visited[i]=0;

} distance[startnode]=0; visited[startnode]=1; count=1; while(count<n-1) {

mindistance=INFINITY; for(i=0;i<n;i++)

if(distance[i]<mindistance&&!visited[i]) { mindistance=distance[i]; nextnode=i;

} visited[nextnode]=1; for(i=0;i<n;i++)

if(!visited[i]) if(mindistance+cost[nextnode][i]<distance[i]) { distance[i]=mindistance+cost[nextnode][i];

pred[i]=nextnode; } count++; } for(i=0;i<n;i++)

if(i!=startnode) {

cout<<"\nDistance of node"<<i<<"="<<distance[i]; cout<<"\nPath="<<i; j=i;

do { j=pred[j];

cout<<"<-"<<j; }while(j!=startnode);

} }

Output

*/* rn |›/ 1gy.'GEhl›1 OK o Dis rance oT node1=1 Pa th=1< —0

Dis rance oT node2=5

## Pa th=2<—3«—0

Dis rance oT node3=3 Pa th=3<—0

Dis rance oT node4=6

Pa th=4<—2< —3< —0

# LAB 5:Knapsack Problem

**AIM:** To implement the Knapsack problem using C language.

##### Abstract:

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item or don’t pick it (0-1 property).

###### PROCEDURE:

Method 1: Recursion by Brute-Force algorithm OR Exhaustive Search.

Approach: A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

*Optimal Sub-structure*: To consider all subsets of items, there can be two cases for every item.

Case 1: The item is included in the optimal subset. Case 2: The item is not included in the optimal set.

Therefore, the maximum value that can be obtained from ‘n’ items is the max of the following two values.

Maximum value obtained by n-1 items and W weight (excluding nth item).

Value of nth item plus maximum value obtained by n-1 items and W minus the weight of the nth item (including nth item).

If the weight of ‘nth’ item is greater than ‘W’, then the nth item cannot be included and Case 1 is the only possibility.

##### Pseudo Code:

dp[N+1][W+1][F+1] // memo table, initially filled with -1 int solve(n,w,f)

{

if(n > N)return 0;

if(dp[n][w][f] != -1) return dp[n][w][f];

dp[n][w][f] = solve(n+1,w,f); //skip item

if(w + weight(n) <= W && f + isFragile(n) <=F)

dp[n][w][f] = max(dp[n][w][f], value(n) + solve(n+1, w + weight(n), f + isFragile(n)));

return dp[n][w][f]

}

print(solve(1,0,0))

**CODE:-**

#include <stdio.h>

// A utility function that returns

// maximum of two integers

int max(int a, int b) { return (a > b) ? a : b; }

// Returns the maximum value that can be

// put in a knapsack of capacity W

int knapSack(int W, int wt[], int val[], int n)

{

// Base Case

if (n == 0 || W == 0) return 0;

// If weight of the nth item is more than

// Knapsack capacity W, then this item cannot

// be included in the optimal solution if (wt[n - 1] > W)

return knapSack(W, wt, val, n - 1);

// Return the maximum of two cases:

// (1) nth item included

// (2) not included else

return max( val[n - 1]

+ knapSack(W - wt[n - 1], wt, val, n - 1),

knapSack(W, wt, val, n - 1));

}

// Driver program to test above function int main()

{

int val[] = { 60, 100, 120 };

int wt[] = { 10, 20, 30 };

int W = 50;

int n = sizeof(val) / sizeof(val[0]); printf("%d", knapSack(W, wt, val, n)); return 0;

}

**OUTPUT:-**

220

**RESULT:-**The Knapsack problem is successfully implemented.

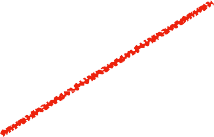
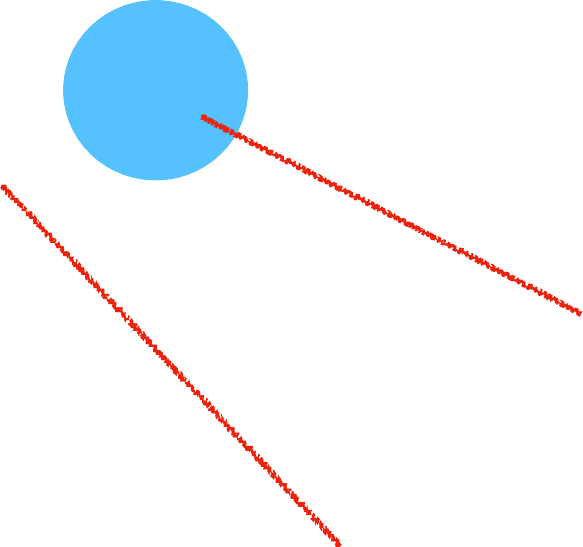
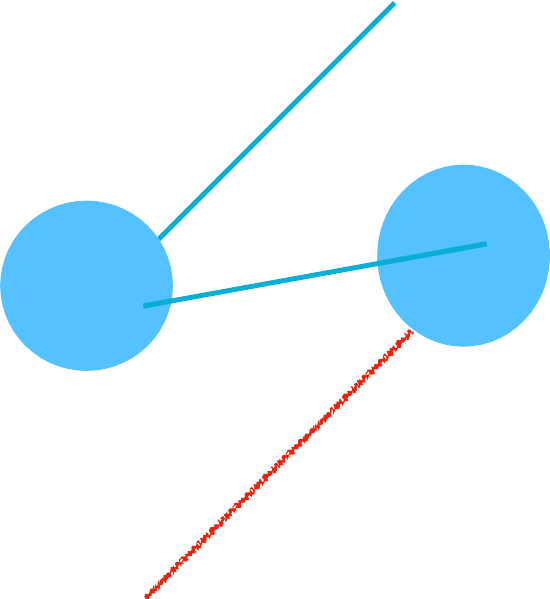
**LAB 6:Travelling Salesman Problem**

**AIM:** To implement the Travelling salesman problem using C language.

**ABSTRACT:**

There is a salesman, he has a goods to sell and has to travel through 5 cities. He need to find the optimum path to those cities; so that the cost of his travel could be reduced at min.

START



END/ TARGET

NODE

PATH

#### ALGORITHM:

C ({1}, 1) = 0

for s = 2 to n do

for all subsets S Є {1, 2, 3, … , n} of size s and containing1

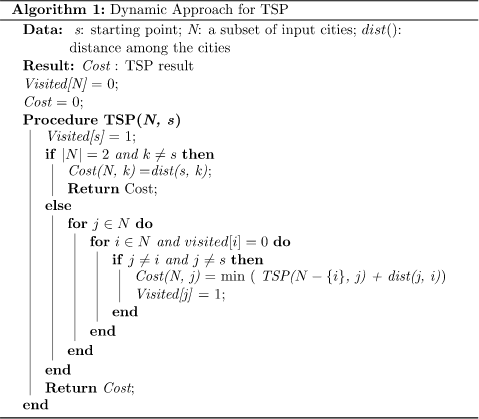
C (S, 1) = ∞

for all j Є S and j ≠ 1

C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}

Return min j C ({1, 2, 3, …, n}, j) + d(j, i)

#### PSUDO CODE:



**CODE:**

#include<stdio.h>

int ary[10][10],completed[10],n,cost=0; void takeInput()

{

int i,j;

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

printf("\nEnter the Cost Matrix\n"); for(i=0;i < n;i++)

{

printf("\nEnter Elements of Row: %d\n",i+1);

for( j=0;j < n;j++) scanf("%d",&ary[i][j]);

completed[i]=0;

}

printf("\n\nThe cost list is:"); for( i=0;i < n;i++)

{

printf("\n");

for(j=0;j < n;j++) printf("\t%d",ary[i][j]);

}

}

void mincost(int city)

{

int i,ncity;

completed[city]=1;

printf("%d--->",city+1); ncity=least(city);

if(ncity==999)

{

ncity=0; printf("%d",ncity+1); cost+=ary[city][ncity];

return;

}

mincost(ncity);

}

int least(int c)

{

int i,nc=999;

int min=999,kmin;

for(i=0;i < n;i++)

{

if((ary[c][i]!=0)&&(completed[i]==0))

if(ary[c][i]+ary[i][c] < min)

{

min=ary[i][0]+ary[c][i]; kmin=ary[c][i];

nc=i;

}

}

if(min!=999) cost+=kmin;

return nc;

}

int main()

{

takeInput();

printf("\n\nThe Path is:\n");

mincost(0); //passing 0 because starting vertex

printf("\n\nMinimum cost is %d\n ",cost); return 0;

}

#### OUTPUT:

*Enter the number of villages: 4 Enter the Cost Matrix*

*Enter Elements of Row: 1 0 4 1 3*

*Enter Elements of Row: 2 4 0 2 1*

*Enter Elements of Row: 3 1 2 0 5*

*Enter Elements of Row: 4 3 1 5 0*

*The cost list is:*

*0 4 1 3*

*4 0 2 1*

*1 2 0 5*

*3 1 5 0*

*The Path is:*

*1—>3—>2—>4—>1*

*Minimum cost is 7*

**RESULT:** The Travelling salesman problem is successfully implemented using c language.

# LAB 7: Tree Traversals

**AIM:** To implement the Tree Traversals using C language.

###### ABSTRACT:

Tree traversal means visiting each node of the tree. The tree is a non-linear data structure, and therefore its traversal is different from other linear data structures. There is only one way to visit each node/element in linear data structures, i.e. starting from the first value and traversing in a linear order.

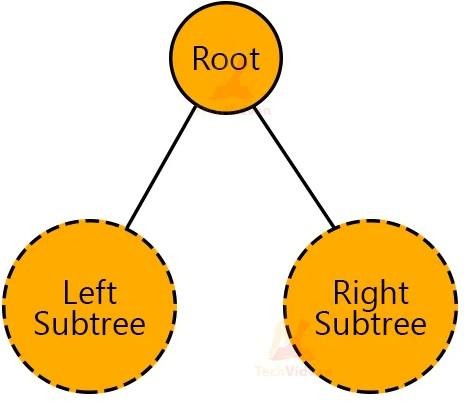
TYPES

PREORDER INORDER POSTORDER

*PREORDER:In a preorder traversal, we process/visit the root node first. Then we traverse the left subtree in a preorder manner. Finally, we visit the right subtree again in a preorder manner.*

*INORDER: In an in-order traversal, we first visit the left subtree, then the root node and then the right subtree in an inorder manner.*

*POSTORDER:Postorder traversal is a kind of traversal in which we first traverse the left subtree in a postorder manner, then traverse the right subtree in a postorder manner and at the end visit the root node.*



###### ALGORITHM:

*PREORDER:*

*Step 1: Visit the root node.*

*Step 2: Traverse left subtree recursively. Step 3: Traverse right subtree recursively.*

*INORDER:*

Step 1: Traverse left subtree recursively. Step 2: Visit the root node.

Step 3: Traverse right subtree recursively.

*POSTORDER:*

Step 1: Traverse left subtree recursively. Step 2: Traverse right subtree recursively. Step 3: Visit the root node.

###### PSEUDO-CODE:

PREORDER:

void Preorder(struct node\* ptr)

{

if(ptr != NULL)

{

printf("%d", ptr->data); Preorder(ptr->left); Preorder(ptr->right);

}

}

INORDER:

void Inorder(struct node\* ptr)

{

if(ptr != NULL)

{

Inorder(ptr->left); printf("%d", ptr->data); Inorder(ptr->right);

}

}

POSTORDER:

void Postorder(struct node\* ptr)

{

if(ptr != NULL)

{

Postorder(ptr->left); Postorder(ptr->right); printf(“%d”, ptr->data);

}

}

###### CODE:

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

struct node { int element;

struct node\* left; struct node\* right;

};

/\*To create a new node\*/

struct node\* createNode(int val)

{

struct node\* Node = (struct node\*)malloc(sizeof(struct node)); Node->element = val;

Node->left = NULL; Node->right = NULL;

return (Node);

}

/\*function to traverse the nodes of binary tree in preorder\*/ void traversePreorder(struct node\* root)

{

if (root == NULL) return;

printf(" %d ", root->element); traversePreorder(root->left); traversePreorder(root->right);

}

/\*function to traverse the nodes of binary tree in Inorder\*/ void traverseInorder(struct node\* root)

{

if (root == NULL) return;

traverseInorder(root->left); printf(" %d ", root->element); traverseInorder(root->right);

}

/\*function to traverse the nodes of binary tree in postorder\*/ void traversePostorder(struct node\* root)

{

if (root == NULL) return;

traversePostorder(root->left); traversePostorder(root->right); printf(" %d ", root->element);

}

int main()

{

struct node\* root = createNode(36); root->left = createNode(26);

root->right = createNode(46); root->left->left = createNode(21);

root->left->right = createNode(31); root->left->left->left = createNode(11);

root->left->left->right = createNode(24); root->right->left = createNode(41);

root->right->right = createNode(56);

root->right->right->left = createNode(51); root->right->right->right = createNode(66);

printf("\n The Preorder traversal of given binary tree is -\n"); traversePreorder(root);

printf("\n The Inorder traversal of given binary tree is -\n"); traverseInorder(root);

printf("\n The Postorder traversal of given binary tree is -\n"); traversePostorder(root);

return 0;

}

###### OUTPUT:

The Preorder traversal of given binary tree is - 36 26 21 11 24 31 46 41 56 51 66

The Inorder traversal of given binary tree is - 11 21 24 26 31 36 41 46 51 56 66

The Postorder traversal of given binary tree is - 11 24 21 31 26 41 51 66 56 46 36

**RESULT:** The Tree Traversal is successfully implemented.

# Lab 8: BFS and DFS

###### AIM:

To implement the “Breadth first search “ and “Depth first search” by using c++ language.

###### ABSTRACT:

Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure.

Traversal means visiting all the nodes of a graph.

###### ALGORITHM:

**BFS**

|  |  |  |
| --- | --- | --- |
| **STEP1**:SET STATUS = 1 (ready state) for each node in G |  | |
| **Step 2:** Enqueue the starting node A and set its STATUS | = | 2 |
| (waiting state) |  |  |
| **Step 3:** Repeat Steps 4 and 5 until QUEUE is empty |  |  |
| **Step 4:** Dequeue a node N. Process it and set its STATUS (processed state). | = | 3 |

**Step 5:** Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set

their STATUS = 2 (waiting state) [END OF LOOP] **Step 6:** EXIT

###### DFS

**Step 1:** SET STATUS = 1 (ready state) for each node in G

**Step 2:** Push the starting node A on the stack and set its STATUS = 2 (waiting state)

**Step 3:** Repeat Steps 4 and 5 until STACK is empty

**Step 4:** Pop the top node N. Process it and set its STATUS = 3 (processed state)

**Step 5:** Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

**Step 6:** EXIT

###### PSUDO CODE:

**BFS**

BFS (G, s) //Where G is the graph and s is the source node

let Q be queue.

Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

mark s as visited. while ( Q is not empty)

//Removing that vertex from queue,whose neighbour will be visited now

v = Q.dequeue( )

//processing all the neighbours of v for all neighbours w of v in Graph G

if w is not visited

Q.enqueue( w ) //Stores w in Q to further visit

its neighbour

mark w as visited.

###### DFS

DFS(G,v) ( v is the vertex where the search starts ) Stack S := {}; ( start with an empty stack )

for each vertex u, set visited[u] := false; push S, v;

while (S is not empty) do u := pop S;

if (not visited[u]) then visited[u] := true;

for each unvisited neighbour w of uu push S, w;

end if end while

END DFS()

**CODE**

**BFS**

import java.io.\*; import java.util.\*;

public class BFSTraversal

{

private int vertex; /\* total number number of vertices in the graph \*/

private LinkedList<Integer> adj[]; /\* adjacency list \*/ private Queue<Integer> que; /\* maintaining a queue \*/ BFSTraversal(int v)

{

vertex = v;

adj = new LinkedList[vertex]; for (int i=0; i<v; i++)

{

adj[i] = new LinkedList<>();

}

que = new LinkedList<Integer>();

}

void insertEdge(int v,int w)

{

adj[v].add(w); /\* adding an edge to the adjacency list (edges are bidirectional in this example) \*/

}

void BFS(int n)

{

boolean nodes[] = new boolean[vertex]; /\* initialize boolean array for holding the data \*/

int a = 0; nodes[n]=true;

que.add(n); /\* root node is added to the top of the queue \*/

while (que.size() != 0)

{

n = que.poll(); /\* remove the top element of the queue \*/

System.out.print(n+" "); /\* print the top element of the queue \*/

for (int i = 0; i < adj[n].size(); i++) /\* iterate through the linked list and push all neighbors into queue \*/

{

a = adj[n].get(i);

if (!nodes[a]) /\* only insert nodes into queue if they have not been explored already \*/

{

nodes[a] = true; que.add(a);

}

}

}

}

public static void main(String args[])

{

is:");

BFSTraversal graph = new BFSTraversal(10); graph.insertEdge(0, 1);

graph.insertEdge(0, 2);

graph.insertEdge(0, 3);

graph.insertEdge(1, 3);

graph.insertEdge(2, 4);

graph.insertEdge(3, 5);

graph.insertEdge(3, 6);

graph.insertEdge(4, 7);

graph.insertEdge(4, 5);

graph.insertEdge(5, 2);

graph.insertEdge(6, 5);

graph.insertEdge(7, 5);

graph.insertEdge(7, 8);

System.out.println("Breadth First Traversal for the graph graph.BFS(2); }}

**DFS**

import java.util.\*;

class DFSTraversal {

private LinkedList<Integer> adj[]; /adjacency list representation/

private boolean visited[];

/\* Creation of the graph \*/

DFSTraversal(int V) /'V' is the number of vertices in the graph/

{

adj = new LinkedList[V]; visited = new boolean[V];

for (int i = 0; i < V; i++)

adj[i] = new LinkedList<Integer>();

}

/\* Adding an edge to the graph \*/ void insertEdge(int src, int dest) { adj[src].add(dest);

}

void DFS(int vertex) {

visited[vertex] = true; /Mark the current node as visited/ System.out.print(vertex + " ");

Iterator<Integer> it = adj[vertex].listIterator(); while (it.hasNext()) {

int n = it.next(); if (!visited[n]) DFS(n);

}

}

public static void main(String args[]) { DFSTraversal graph = new DFSTraversal(8);

graph.insertEdge(0, 1);

graph.insertEdge(0, 2);

graph.insertEdge(0, 3);

graph.insertEdge(1, 3);

graph.insertEdge(2, 4);

graph.insertEdge(3, 5);

graph.insertEdge(3, 6);

graph.insertEdge(4, 7);

graph.insertEdge(4, 5);

graph.insertEdge(5, 2);

System.out.println("Depth First Traversal for the graph

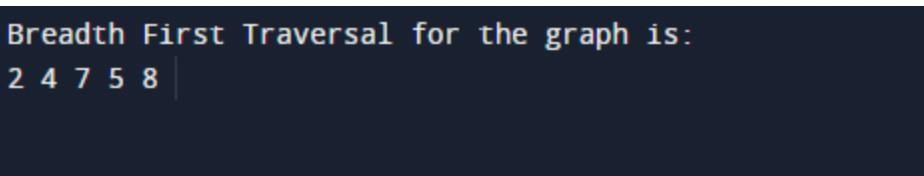
is:”);

graph.DFS(0);

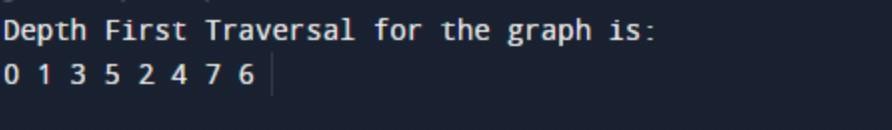
}

}

**OUTPUT:**

**BFS**

**DFS**



**RESULT:**

The BFS and DFS is successfully implemented**.**

# LAB 9: Prim’s Algorithm

**AIM:** To implement the Prim’s algorithm using C language.

**ABSTRACT:**

Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached

* First, we have to initialise an MST with the randomly chosen vertex.
* Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.
* Repeat step 2 until the minimum spanning tree is formed.

#### PSUDO CODE:

T = ∅; U = { 1 };

while (U ≠ V)

let (u, v) be the lowest cost edge such that u ∈ U and v ∈

V - U;

T = T ∪ {(u, v)} U = U ∪ {v}

#### ALGORITHM:

Step 1: Select a starting vertex

Step 2: Repeat Steps 3 and 4 until there are fringe vertices Step 3: Select an edge 'e' connecting the tree vertex and fri nge vertex that has minimum weight

Step 4: Add the selected edge and the vertex to the minimu m spanning tree T

END OF LOOP] Step 5: EXIT

**CODE:**

#include <stdio.h> #include <limits.h>

#define vertices 5 /Define the number of vertices in the graph/

/\* create minimum\_key() method for finding the vertex that has minimum key-value and that is not added in MST yet \*/

int minimum\_key(int k[], int mst[])

{

int minimum = INT\_MAX, min,i;

/iterate over all vertices to find the vertex with minimum key-value/ for (i = 0; i < vertices; i++)

if (mst[i] == 0 && k[i] < minimum ) minimum = k[i], min = i;

return min;

}

/\* create prim() method for constructing and printing the MST.

The g[vertices][vertices] is an adjacency matrix that defines the graph for MST.\*/

void prim(int g[vertices][vertices])

{

/\* create array of size equal to total number of vertices for storing the MST\*/

int parent[vertices];

/\* create k[vertices] array for selecting an edge having minimum weight\*/

int k[vertices];

int mst[vertices];

int i, count,edge,v; /Here 'v' is the vertex/ for (i = 0; i < vertices; i++)

{

k[i] = INT\_MAX;

mst[i] = 0;

}

k[0] = 0; /It select as first vertex/

parent[0] = -1; /\* set first value of parent[] array to -1 to make it root of MST\*/

for (count = 0; count < vertices-1; count++)

{

/select the vertex having minimum key and that is not added in the MST yet from the set of vertices/

edge = minimum\_key(k, mst); mst[edge] = 1;

for (v = 0; v < vertices; v++)

{

if (g[edge][v] && mst[v] == 0 && g[edge][v] < k[v])

{

parent[v] = edge, k[v] = g[edge][v];

}

}

}

/Print the constructed Minimum spanning tree/ printf("\n Edge \t Weight\n");

for (i = 1; i < vertices; i++)

printf(" %d <-> %d %d \n", parent[i], i, g[i][parent[i]]);

}

int main()

{

int g[vertices][vertices] = {{0, 0, 3, 0, 0},

{0, 0, 10, 4, 0},

{3, 10, 0, 2, 6},

{0, 4, 2, 0, 1},

{0, 0, 6, 1, 0},

};

prim(g); return 0;

}

###### OUTPUT:



**RESULT:** The Prim’s algorithm is successfully implemented.

# LAB 9: Kruskal's algorithm

**AIM:** To implement the Kruskal's algorithm by using C- Language.

**ABSTRACT:**

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

* form a tree that includes every vertex
* has the minimum sum of weights among all the trees that can be formed from the graph

**PSUDO CODE:**

KRUSKAL(G):

A = ∅

For each vertex v ∈ G.V: MAKE-SET(v)

For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):

if FIND-SET(u) ≠ FIND-SET(v):

A = A ∪ {(u, v)} UNION(u, v)

return A

#### ALGORITHM:

Step 1: Create a forest F in such a way that every vertex of the graph is a separate tree.

Step 2: Create a set E that contains all the edges of the graph. Step 3: Repeat Steps 4 and 5 while E is NOT EMPTY and F is not spanning

Step 4: Remove an edge from E with minimum weight

Step 5: IF the edge obtained in Step 4 connects two different trees, then add it to the forest F

(for combining two trees into one tree). ELSE

Discard the edge Step 6: END

#### CODE:

#include <stdio.h>

#define MAX 30

typedef struct edge { int u, v, w;

} edge;

typedef struct edge\_list { edge data[MAX];

int n;

} edge\_list; edge\_list elist;

int Graph[MAX][MAX], n; edge\_list spanlist;

void kruskalAlgo();

int find(int belongs[], int vertexno);

void applyUnion(int belongs[], int c1, int c2); void sort();

void print();

// Applying Krushkal Algo void kruskalAlgo() {

int belongs[MAX], i, j, cno1, cno2; elist.n = 0;

for (i = 1; i < n; i++) for (j = 0; j < i; j++) { if (Graph[i][j] != 0) {

elist.data[elist.n].u = i; elist.data[elist.n].v = j; elist.data[elist.n].w = Graph[i][j]; elist.n++;

}

}

sort();

for (i = 0; i < n; i++) belongs[i] = i;

spanlist.n = 0;

for (i = 0; i < elist.n; i++) {

cno1 = find(belongs, elist.data[i].u); cno2 = find(belongs, elist.data[i].v);

if (cno1 != cno2) { spanlist.data[spanlist.n] = elist.data[i]; spanlist.n = spanlist.n + 1; applyUnion(belongs, cno1, cno2);

}

}

}

int find(int belongs[], int vertexno) { return (belongs[vertexno]);

}

void applyUnion(int belongs[], int c1, int c2) { int i;

for (i = 0; i < n; i++) if (belongs[i] == c2) belongs[i] = c1;

}

// Sorting algo void sort() { int i, j;

edge temp;

for (i = 1; i < elist.n; i++)

for (j = 0; j < elist.n - 1; j++)

if (elist.data[j].w > elist.data[j + 1].w) { temp = elist.data[j];

elist.data[j] = elist.data[j + 1]; elist.data[j + 1] = temp;

}

}

// Printing the result void print() {

int i, cost = 0;

for (i = 0; i < spanlist.n; i++) {

printf("\n%d - %d : %d", spanlist.data[i].u, spanlist.data[i].v, spanlist.data[i].w);

cost = cost + spanlist.data[i].w;

}

printf("\nSpanning tree cost: %d", cost);

}

int main() {

int i, j, total\_cost; n = 6;

Graph[0][0] = 0;

Graph[0][1] = 4;

Graph[0][2] = 4;

Graph[0][3] = 0;

Graph[0][4] = 0;

Graph[0][5] = 0;

Graph[0][6] = 0;

Graph[1][0] = 4;

Graph[1][1] = 0;

Graph[1][2] = 2;

Graph[1][3] = 0;

Graph[1][4] = 0;

Graph[1][5] = 0;

Graph[1][6] = 0;

Graph[2][0] = 4;

Graph[2][1] = 2;

Graph[2][2] = 0;

Graph[2][3] = 3;

Graph[2][4] = 4;

Graph[2][5] = 0;

Graph[2][6] = 0;

Graph[3][0] = 0;

Graph[3][1] = 0;

Graph[3][2] = 3;

Graph[3][3] = 0;

Graph[3][4] = 3;

Graph[3][5] = 0;

Graph[3][6] = 0;

Graph[4][0] = 0;

Graph[4][1] = 0;

Graph[4][2] = 4;

Graph[4][3] = 3;

Graph[4][4] = 0;

Graph[4][5] = 0;

Graph[4][6] = 0;

Graph[5][0] = 0;

Graph[5][1] = 0;

Graph[5][2] = 2;

Graph[5][3] = 0;

Graph[5][4] = 3;

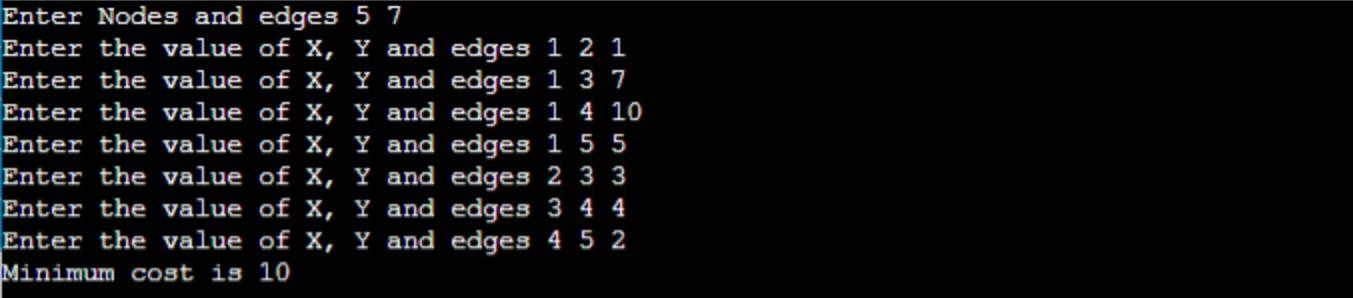
Graph[5][5] = 0;

Graph[5][6] = 0;

kruskalAlgo(); print();

}

##### OUTPUT:



**RESULT:** The Kruskal's algorithm is successfully implemented.

# LAB 10: Linear Search

**AIM:** To implement linear search using C language.

**ABSTRACT:**

Linear search is also called as sequential search algorithm. It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.

**ALGORITHM:**

Linear\_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search Step 1: set pos = -1

Step 2: set i = 1

Step 3: repeat step 4 while i <= n Step 4: if a[i] == val

set pos = i print pos

go to step 6 [end of if] set ii = i + 1 [end of loop]

Step 5: if pos = -1

print "value is not present in the array " [end of if]

Step 6: exit

**PSUDO CODE:**

procedure linear\_search (list, value)

for each item in the list if match item == value

return the item's location end if

end for

end procedure

###### CODE:

#include <stdio.h>

int linearSearch(int a[], int n, int val) {

for (int i = 0; i < n; i++)

{

if (a[i] == val) return i+1;

}

return -1;

}

int main() {

int a[] = {70, 40, 30, 11, 57, 41, 25, 14, 52}; // given array int val = 41; // value to be searched

int n = sizeof(a) / sizeof(a[0]); // size of array int res = linearSearch(a, n, val); // Store result printf("The elements of the array are - ");

for (int i = 0; i < n; i++) printf("%d ", a[i]);

printf("\nElement to be searched is - %d", val); if (res == -1)

printf("\nElement is not present in the array"); else

printf("\nElement is present at %d position of array", res); return 0;

}

###### OUTPUT



**RESULT:**The Linear search has been successfully implemented using C language.

# LAB 10:BINARY SEARCH

**AIM:** To implement the binary search using C language.

**ABSTRACT:**

Binary Search is a searching algorithm for finding an element's position in a sorted array.

In this approach, the element is always searched in the middle of a portion of an array.

Binary Search Algorithm can be implemented in two ways which are discussed below.

1. Iterative Method
2. Recursive Method

###### ALGORITHM:

**Iterative method:**

do until the pointers low and high meet each other. mid = (low + high)/2

if (x == arr[mid]) return mid

else if (x > arr[mid]) // x is on the right side low = mid + 1

else // x is on the left side high = mid - 1

###### Recursive Method:

binarySearch(arr, x, low, high) if low > high

return False else

mid = (low + high) / 2 if x == arr[mid]

return mid

else if x > arr[mid] // x is on the right side return binarySearch(arr, x, mid + 1, high)

else // x is on the right side return binarySearch(arr, x, low, mid - 1)

###### PSUDO CODE:-

Procedure binary\_search

A ← sorted array n ← size of array

x ← value to be searched

Set lowerBound = 1 Set upperBound = n

while x not found

if upperBound < lowerBound EXIT: x does not exists.

set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

if A[midPoint] < x

set lowerBound = midPoint + 1 if A[midPoint] > x

set upperBound = midPoint - 1

if A[midPoint] = x

EXIT: x found at location midPoint end while

end procedure

###### CODE:-

**Iterating method:**

#include <stdio.h>

int binarySearch(int array[], int x, int low, int high) {

// Repeat until the pointers low and high meet each other while (low <= high) {

int mid = low + (high - low) / 2;

if (array[mid] == x) return mid;

if (array[mid] < x) low = mid + 1;

else

high = mid - 1;

}

return -1;

}

int main(void) {

int array[] = {3, 4, 5, 6, 7, 8, 9};

int n = sizeof(array) / sizeof(array[0]); int x = 4;

int result = binarySearch(array, x, 0, n - 1); if (result == -1)

printf("Not found"); else

printf("Element is found at index %d", result); return 0;

}

###### Recursive Method:

#include <stdio.h>

int binarySearch(int array[], int x, int low, int high) { if (high >= low) {

int mid = low + (high - low) / 2;

// If found at mid, then return it if (array[mid] == x)

return mid;

// Search the left half if (array[mid] > x)

return binarySearch(array, x, low, mid - 1);

// Search the right half

return binarySearch(array, x, mid + 1, high);

}

return -1;

}

int main(void) {

int array[] = {3, 4, 5, 6, 7, 8, 9};

int n = sizeof(array) / sizeof(array[0]); int x = 4;

int result = binarySearch(array, x, 0, n - 1); if (result == -1)

printf("Not found"); else

printf("Element is found at index %d", result);

}

**RESULT:** The binary search has been successfully implemented.