D DIVISION

Design and Analysis of Algorithms

Design and Analysis of Algorithms

1. BST insert and delete operation :

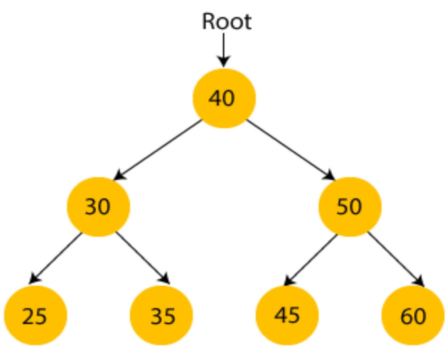
```
#include<stdio.h>
#include<stdlib.h>
struct tree
{
   int data;
   struct tree *left;
   struct tree *right;
};
typedef struct tree TREE;

TREE * insert_into_bst(TREE *, int);
void inorder(TREE *);
void preorder(TREE *);
void postorder(TREE *);
TREE * delete_from_bst(TREE *, int);

// INSERT
TREE * insert_into_bst(TREE * root, int data)
{
   TREE *newnode,*currnode,*parent;
   // Dynamically allocate the memory using malloc
   newnode=(TREE*)malloc(sizeof(TREE));
   // Check if the memory allocation was successful
   if(newnode==NULL)
   {
      printf("Memory allocation failed\n");
      return root;
   }
   // Initialize the tree node elements
   newnode->data = data;
   newnode->left = NULL;
   newnode->right = NULL;
   // When the first insertion happens which is the root node
   if(root == NULL)
   {
      root = newnode;
      printf("Root node inserted into tree\n");
      return root;
   }
   // Traverse through the desired part of the tree using currnode and parent pointers
   currnode = root;
   parent = NULL;
   while(currnode != NULL)
   {
```

```
        parent = currnode;
        if(newnode->data < currnode->data)
            currnode = currnode->left;
        else
            currnode = currnode->right;
    }
    // Attach the node at appropriate place using parent
    if(newnode->data < parent->data)
        parent->left = newnode;
    else
        parent->right = newnode;
    // print the successful insertion and return root
    printf("Node inserted successfully into the tree\n");
    return root;
}

// INORDER
void inorder(TREE *troot)
{
    if(troot != NULL)
        {
            inorder(troot->left);
            printf("%d\t",troot->data);
            inorder(troot->right);
    }
}

// DELETE
TREE * delete_from_bst(TREE * root, int data)
{
    TREE * currnode, *parent, *successor, *p;
    // Check if the tree is empty
    if(root == NULL)
    {
        printf("Tree is empty\n");
        return root;
    }
    // Traverse and reach the appropriate part of the tree
    parent = NULL;
    currnode = root;
    while (currnode != NULL && data != currnode->data)
    {
        parent = currnode;
        if(data < currnode->data)
            currnode  = currnode->left;
        else
            currnode = currnode->right;
    }
    // If the data is not present in the tree
    if(currnode == NULL)  {
```

```
        printf("Item not found\n");
        return root;
    }
    // Check and manipulate if either left subtree is absent,or right subtree is absent or both are present
    if(currnode->left == NULL)
        p = currnode->right;
    else if (currnode->right == NULL)
        p = currnode->left;
    else
    {
     // Process of finding the inorder successor
        successor = currnode->right;
        while(successor->left != NULL)
            successor = successor->left;
        successor->left = currnode->left;
        p = currnode->right;
    }
    // The case of root deletion
    if (parent == NULL) {
        free(currnode);
        return p;
    }
    if(currnode == parent ->left)
        parent->left = p;
    else
        parent->right = p;
    free(currnode);
    return root;
}

int main()
{
    TREE * root;
    root = NULL;
    int choice = 0;
    int data = 0;
    int count = 0;

    while(1)
    {
        printf("\n******** Menu *************\n");
        printf("1-Insert into BST\n");
        printf("2-Inorder Traversal\n");
        printf("3-Delete from BST\n");
        printf("Any other option to exit\n");
        printf("*\n");
        printf("Enter your choice\n");
        scanf("%d", &choice);
```

```
    switch(choice)
    {
        case 1: printf("Enter the item to insert\n");
                scanf("%d", &data);
                root = insert_into_bst(root, data);
                break;

        case 2: if(root == NULL)
                    printf("Tree is empty\n");
                else
                {
                    printf("Inorder Traversal is...\n");
                    inorder(root);
                }
                break;

        case 3: printf("Enter the item to be deleted\n");
                scanf("%d", &data);
                root = delete_from_bst(root, data);
                break;

        default: printf("Exciting Code.\n");
                 exit(0);
    }
  }
  return 0;
}
```

Output :



******** Menu *************
1-Insert into BST
2-Inorder Traversal
3-Delete from BST
Any other option to exit
*
Enter your choice
2
Tree is empty

******** Menu *************
1-Insert into BST
2-Inorder Traversal
3-Delete from BST
Any other option to exit
*
Enter your choice
1
Enter the item to insert
40
Root node inserted into tree
--------- so on ---------------
******** Menu *************
1-Insert into BST
2-Inorder Traversal
3-Delete from BST
Any other option to exit
*
Enter your choice
1
Enter the item to insert

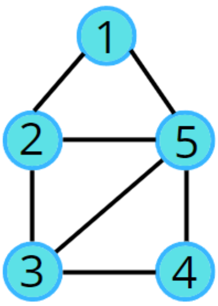
```
60
Node inserted successfully into the tree

******** Menu *************
1-Insert into BST
2-Inorder Traversal
3-Delete from BST
Any other option to exit
*
Enter your choice
2
Inorder Traversal is...
25	30	35	40	45	50	60
******** Menu *************
1-Insert into BST
2-Inorder Traversal
3-Delete from BST
Any other option to exit
*
Enter your choice
3
Enter the item to be deleted
50

******** Menu *************
1-Insert into BST
2-Inorder Traversal
3-Delete from BST
Any other option to exit
*
Enter your choice
2
Inorder Traversal is...
25	30	35	40	45	60
******** Menu *************
1-Insert into BST
2-Inorder Traversal
3-Delete from BST
Any other option to exit
*
Enter your choice
7
Exciting Code.
```

2. Breadth First Search (BFS) :

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

void bfs(int m[10][10], int v, int source) {
   int queue[20];
   int front = 0, rear = 0, u, i;
   int visited[10];

   for (i = 0; i < v; i++)
      visited[i] = 0;

   queue[rear] = source;
   visited[source] = 1;
   cout << "The BFS Traversal is... \n";

   while (front <= rear) {
      u = queue[front];
      cout << u << "\t";
      front++;

      for (i = 0; i < v; i++) {
         if (m[u][i] == 1 && visited[i] == 0) {
            visited[i] = 1;
```

```
                rear++;
                queue[rear] = i;
            }
        }
    }
}

int main() {
    int v = 5;
    int m[10][10] = {{0,1,0,0,1}, {1,0,1,0,1},{0,1,0,1,1}, {0,0,1,0,1}, {1,1,1,1,0}};

    int source;
    cout << "Enter the source vertex: ";
    cin >> source;
    bfs(m, v, source);
    return 0;
}
```

Output :



Enter the source vertex: 0

The BFS Traversal is...

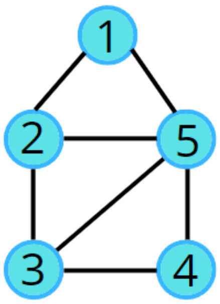
0 1 4 2 3

3. Depth First Search (DFS) :

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

int v = 5;
int m[10][10] = {{0,1,0,0,1}, {1,0,1,0,1},{0,1,0,1,1}, {0,0,1,0,1}, {1,1,1,1,0}};
int visited[10];

void dfs(int m[10][10], int v, int source) {
    visited[source] = 1;
    for (int i = 0; i < v; i++) {
        if (m[source][i] == 1 && visited[i] == 0) {
            cout << i << "\t";
            dfs(m, v, i);
        }
```

```
    }
}

int main() {
    int source;
    for (int i = 0; i < v; i++)
        visited[i] = 0;

    cout << "Enter the source vertex: ";
    cin >> source;
    cout << "The DFS Traversal is... \n";
    cout << source << "\t";
    dfs(m, v, source);
    return 0;
}
```

Output :



Enter the source vertex: 0

The DFS Traversal is...

0 1 2 3 4

4. Bubble sort :

```
#include <iostream>
using namespace std;
// Function to perform Bubble Sort
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        // Last i elements are already sorted
        for (int j = 0; j < n - i - 1; j++) {
            // Swap if the current element is greater than the next
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
```

```
    }
}
// Function to print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Original array: ";
    printArray(arr, n);
    bubbleSort(arr, n);
    cout << "Sorted array: ";
    printArray(arr, n);
    return 0;
}
```

Output :

```
Original array: 64 34 25 12 22 11 90
Sorted array: 11 12 22 25 34 64 90
```

5. Selection sort :

```
#include <iostream>
using namespace std;
// Function to perform Selection Sort
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        // Find the minimum element in the unsorted part of the array
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j; // Update the index of the smallest element
            }
        }
        // Swap the smallest element with the first element of the unsorted part
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}
// Function to print an array
void printArray(int arr[], int n) {
```

```
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Original array: ";
    printArray(arr, n);
    selectionSort(arr, n);
    cout << "Sorted array: ";
    printArray(arr, n);
    return 0;
}
```

Output :

```
Original array: 64 25 12 22 11
Sorted array: 11 12 22 25 64
```

6. Insertion sort :

```
#include <iostream>
using namespace std;
// Function to perform Insertion Sort
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i]; // The current element to be inserted in the sorted part
        int j = i - 1;
        // Move elements of the sorted part that are greater than 'key' to one position ahead
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        // Place the 'key' in its correct position
        arr[j + 1] = key;
    }
}
// Function to print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
```

```
   int arr[] = {64, 34, 25, 12, 22, 11, 90};
   int n = sizeof(arr) / sizeof(arr[0]);
   cout << "Original array: ";
   printArray(arr, n);
   insertionSort(arr, n);
   cout << "Sorted array: ";
   printArray(arr, n);
   return 0;
}
```

Output :

```
Original array: 64 34 25 12 22 11 90
Sorted array: 11 12 22 25 34 64 90
```

7. Merge sort :

```
#include <iostream>
using namespace std;
// Function to merge two subarrays
void merge(int arr[], int left, int mid, int right) {
   int n1 = mid - left + 1; // Size of the left subarray
   int n2 = right - mid;    // Size of the right subarray
   // Create temporary arrays
   int leftArr[n1], rightArr[n2];
   // Copy data to temporary arrays
   for (int i = 0; i < n1; i++)
      leftArr[i] = arr[left + i];
   for (int i = 0; i < n2; i++)
      rightArr[i] = arr[mid + 1 + i];
   // Merge the temporary arrays back into arr[left..right]
   int i = 0, j = 0, k = left;
   while (i < n1 && j < n2) {
      if (leftArr[i] <= rightArr[j]) {
         arr[k] = leftArr[i];
         i++;
      } else {
         arr[k] = rightArr[j];
         j++;
      }
      k++;
   }
   // Copy remaining elements of leftArr[], if any
   while (i < n1) {
      arr[k] = leftArr[i];
      i++;
      k++;
   }
   // Copy remaining elements of rightArr[], if any
```

```
 while (j < n2) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}
// Function to implement Merge Sort
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2; // Find the middle point
        // Recursively sort the two halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}
// Function to print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Original array: ";
    printArray(arr, n);
    mergeSort(arr, 0, n - 1);
    cout << "Sorted array: ";
    printArray(arr, n);
    return 0;
}
```

Output :

Original array: 38 27 43 3 9 82 10
Sorted array: 3 9 10 27 38 43 82

8. Quick Sort :

```
#include <iostream>
using namespace std;
// Function to partition the array
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choosing the last element as the pivot
```

```
    int i = low - 1;      // Index of the smaller element

    for (int j = low; j < high; j++) {
        // If the current element is smaller than or equal to the pivot
        if (arr[j] <= pivot) {
            i++; // Increment the index of the smaller element
            // Swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    // Swap the pivot element with the element at i+1
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return i + 1; // Return the partition index
}
// Function to perform Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Find the partition index
        int pi = partition(arr, low, high);
        // Recursively sort the elements on both sides of the partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
// Function to print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Original array: ";
    printArray(arr, n);
    quickSort(arr, 0, n - 1);
    cout << "Sorted array: ";
    printArray(arr, n);
    return 0;
}
```

Output :

Original array: 10 7 8 9 1 5

Sorted array: 1 5 7 8 9 10

9. Heap Sort :

```
#include <iostream>
using namespace std;
// Function to heapify a subtree rooted at index `i`
// `n` is the size of the heap
void heapify(int arr[], int n, int i) {
   int largest = i;       // Initialize the largest as root
   int left = 2 * i + 1;   // Left child index
   int right = 2 * i + 2;  // Right child index
   // Check if the left child is larger than the root
   if (left < n && arr[left] > arr[largest]) {
      largest = left;
   }
   // Check if the right child is larger than the largest so far
   if (right < n && arr[right] > arr[largest]) {
      largest = right;
   }
   // If the largest is not the root
   if (largest != i) {
      swap(arr[i], arr[largest]); // Swap root with the largest
      // Recursively heapify the affected subtree
      heapify(arr, n, largest);
   }
}
// Function to perform Heap Sort
void heapSort(int arr[], int n) {
   // Build a max heap
   for (int i = n / 2 - 1; i >= 0; i--) {
      heapify(arr, n, i);
   }
   // Extract elements from the heap one by one
   for (int i = n - 1; i > 0; i--) {
      // Move the current root (largest) to the end
      swap(arr[0], arr[i]);
      // Reduce the size of the heap and heapify the root
      heapify(arr, i, 0);
   }
}
// Function to print the array
void printArray(int arr[], int n) {
   for (int i = 0; i < n; i++) {
      cout << arr[i] << " ";
   }
   cout << endl;
}
```

```
int main() {
   int arr[] = {12, 11, 13, 5, 6, 7};
   int n = sizeof(arr) / sizeof(arr[0]);

   cout << "Original array: ";
   printArray(arr, n);
   heapSort(arr, n);
   cout << "Sorted array: ";
   printArray(arr, n);
   return 0;
}
```

Output :

```
Original array: 12 11 13 5 6 7
Sorted array: 5 6 7 11 12 13
```

10. Brute force String Matching :

```
#include <iostream>
#include <string>
using namespace std;
// Function to perform string matching using brute force
void bruteForceStringMatch(string text, string pattern) {
   int n = text.length();     // Length of the text
   int m = pattern.length();  // Length of the pattern
   // Loop through the text to find the pattern
   for (int i = 0; i <= n - m; i++) {
      int j;
      // Check if the pattern matches the current substring
      for (j = 0; j < m; j++) {
         if (text[i + j] != pattern[j]) {
            break;
         }
      }
      // If the pattern is found
      if (j == m) {
         cout << "Pattern found at index " << i << endl;
      }
   }
}

int main() {
   string text = "ababcabcabababd";
   string pattern = "ababd";
   cout << "Text: " << text << endl;
   cout << "Pattern: " << pattern << endl;
   bruteForceStringMatch(text, pattern);
   return 0;
```

```
}
```

Output :

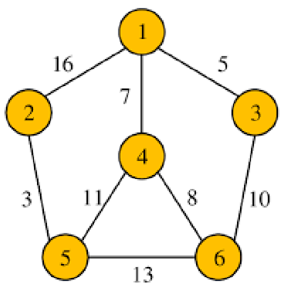
```
Text: ababcabcabababd
Pattern: ababd
Pattern found at index 10
```

11. Dijkstra’s Algorithm (Array) :

```
#include <iostream>
#include <climits> // For INT_MAX
using namespace std;
// Function to find the vertex with the minimum distance
int getMinDistanceVertex(int dist[], bool visited[], int V) {
   int min = INT_MAX, minIndex = -1;
   for (int i = 0; i < V; i++) {
      if (!visited[i] && dist[i] < min) {
         min = dist[i];
         minIndex = i;
      }
   }
   return minIndex;
}
// Dijkstra's algorithm
void dijkstra(int graph[][6], int src, int V) {
   int dist[V];   // Stores shortest distance from source
   bool visited[V] = {false}; // Tracks visited vertices
   // Initialize distances to infinity and source distance to 0
   for (int i = 0; i < V; i++){
      dist[i] = INT_MAX;
   }
   dist[src] = 0;
   // Iterate to calculate shortest paths
   for (int i = 0; i < V - 1; i++) {
      // Pick the minimum distance vertex from the set of unvisited vertices
      int u = getMinDistanceVertex(dist, visited, V);
      // Mark the picked vertex as visited
      visited[u] = true;
      // Update distances for neighbors(adjacent vertices)
      for (int v = 0; v < V; v++) {
         if (!visited[v] && graph[u][v] && dist[u] != INT_MAX
            && dist[u] + graph[u][v] < dist[v]) {
            dist[v] = dist[u] + graph[u][v];
         }
      }
   }
   // Print results
```

```
    cout<<"Using Array :" <<endl;
    cout << "Vertex\t \tDistance from Source\n";
    for (int i = 0; i < V; i++) {
        cout << "vertex\t" << i << "\t->" << "\t" << dist[i] << endl;
    }
}

int main() {
    int graph[6][6] = {
        {0, 16, 5, 7, 0, 0},
        {16, 0, 0, 0, 3, 0},
        {5, 0, 0, 0, 0, 10},
        {7, 0, 0, 0, 11, 8},
        {0, 3, 0, 11, 0, 13},
        {0, 0, 10, 8, 13, 0}
    };
    dijkstra(graph, 0, 6); // Source vertex: 0
    return 0;
}
```

Output :



Using Array :
Vertex   Distance from Source
vertex 0 -> 0
vertex 1 -> 16
vertex 2 -> 5
vertex 3 -> 7
vertex 4 -> 18
vertex 5 -> 15

12. Dijkstra's Algorithm (Heap) :

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

void dijkstra(int graph[][6], int src, int V) {
    // Min-heap to store (distance, vertex)
```

```
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    vector<int> dist(V, INT_MAX); // Distance array initialized to infinity
    dist[src] = 0; // Distance to source is 0
    pq.push({0, src}); // Push source to the queue
    while (!pq.empty()) {
        int u = pq.top().second; // Current vertex
        pq.pop();
        // Process all neighbors of u
        for (int v = 0; v < V; v++) {
            if (graph[u][v] && dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
                pq.push({dist[v], v}); // Push updated distance
            }
        }
    }
    // Print distances
    cout<<"Using Heap :" <<endl;
    cout << "Vertex\t \tDistance from Source\n";
    for (int i = 0; i < V; i++) {
        cout << "vertex\t" << i << "\t->" << "\t" << dist[i] << endl;
    }
}

int main() {
    int V = 6; // Number of vertices
    int graph[6][6] = {
        {0, 16, 5, 7, 0, 0},
        {16, 0, 0, 0, 3, 0},
        {5, 0, 0, 0, 0, 10},
        {7, 0, 0, 0, 11, 8},
        {0, 3, 0, 11, 0, 13},
        {0, 0, 10, 8, 13, 0}
    };
    dijkstra(graph, 0, V); // Source vertex is 0
    return 0;
}
```

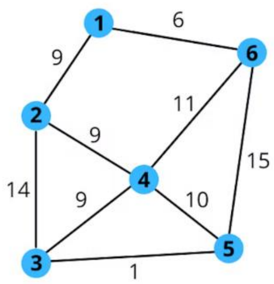
Output :

```
Using Heap :
Vertex      Distance from Source
vertex  0  -> 0
vertex  1  -> 16
vertex  2  -> 5
vertex  3  -> 7
vertex  4  -> 18
vertex  5  -> 15
```

13. Kruskal's Algorithm :

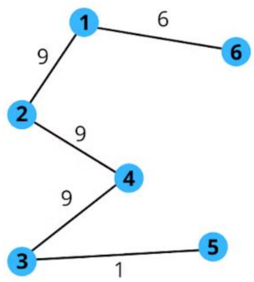
```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
// Structure to represent an edge
struct Edge {
   int src, dest, weight;
};
// Comparator function to sort edges by weight
bool compareEdges(Edge a, Edge b) {
   return a.weight < b.weight;
}
// Function to find the parent of a node (with path compression)
int findParent(int node, vector<int>& parent) {
   if (parent[node] == node)
      return node;
   return parent[node] = findParent(parent[node], parent);
}
// Function to perform union of two sets
void unionSets(int u, int v, vector<int>& parent, vector<int>& rank) {
   u = findParent(u, parent);
   v = findParent(v, parent);
   if (rank[u] < rank[v]) {
      parent[u] = v;
   } else if (rank[u] > rank[v]) {
      parent[v] = u;
   } else {
      parent[v] = u;
      rank[u]++;
   }
}
// Kruskal's Algorithm to find MST
void kruskal(int V, vector<Edge>& edges) {
   // Sort edges by weight
   sort(edges.begin(), edges.end(), compareEdges);
   // Initialize parent and rank for union-find
   vector<int> parent(V);
   vector<int> rank(V, 0);
   for (int i = 0; i < V; i++) {
      parent[i] = i; // Each node is its own parent initially
   }
   vector<Edge> mst; // Store the edges of the MST
   int mstWeight = 0; // Total weight of the MST

   for (Edge& edge : edges) {
      int u = findParent(edge.src, parent);
      int v = findParent(edge.dest, parent);
```

```
        // If u and v are in different sets, include this edge in MST
        if (u != v) {
            mst.push_back(edge);
            mstWeight += edge.weight;
            unionSets(u, v, parent, rank);
        }
    }
    // Print the MST
    cout << "Edges in the Minimum Spanning Tree:" << endl;
    for (Edge& edge : mst) {
        cout << edge.src << " -- " << edge.dest << " == " << edge.weight << endl;
    }
    cout << "Total weight of the MST: " << mstWeight << endl;
}

int main() {
    int V = 6; // Number of vertices
    vector<Edge> edges = {
        {0, 1, 9},
        {0, 5, 6},
        {1, 2, 14},
        {1, 3, 9},
        {2, 3, 9},
        {2, 4, 1},
        {3, 4, 10},
        {3, 5, 11},
        {4, 5, 15}
    };
    kruskal(V, edges);
    return 0;
}
```



Output :

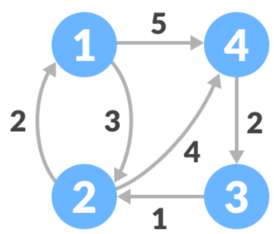
Edges in the Minimum Spanning Tree:
2 -- 4 == 1
0 -- 5 == 6
0 -- 1 == 9
1 -- 3 == 9
2 -- 3 == 9
Total weight of the MST: 34



14. Floyd's Algorithm :

```
#include <iostream>
#include <vector>
using namespace std;
const int INF = 1e9; // A large value representing infinity
```

```cpp
// Function to implement Floyd's Algorithm
void floydWarshall(vector<vector<int>>& graph, int V) {
  // Distance matrix initialized with the input graph
  vector<vector<int>> dist = graph;
  // Update the distance matrix
  for (int k = 0; k < V; k++) {      // Intermediate node
    for (int i = 0; i < V; i++) {  // Source node
      for (int j = 0; j < V; j++) { // Destination node
        if (dist[i][k] != INF && dist[k][j] != INF) {
          dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
        }
      }
    }
  }
  // Print the shortest distances between all pairs of nodes
  cout << "Shortest distances between every pair of vertices:" << endl;
  for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++) {
      if (dist[i][j] == INF)
        cout << "INF ";
      else
        cout << dist[i][j] << " ";
    }
    cout << endl;
  }
}

int main() {
  int V = 4; // Number of vertices
  vector<vector<int>> graph = {
    {0, 3, INF, 5},
    {2, 0, INF, 4},
    {INF, 1, 0, INF},
    {INF, INF, 2, 0}
  };
  floydWarshall(graph, V);
  return 0;
}
```



Output :

Shortest distances between every pair of vertices:
0 3 7 5
2 0 6 4
3 1 0 5
5 3 2 0