/*Implement c Program for minimum Heap Tree*/

```
#include <stdio.h>

// Function to heapify a subtree rooted with node i which is an index in arr[]
void minHeapify(int arr[], int n, int i) {
   int smallest = i;  // Initialize smallest as root
   int left = 2 * i + 1;  // left child
   int right = 2 * i + 2; // right child

   // If left child is smaller than root
   if (left < n && arr[left] < arr[smallest])
      smallest = left;

   // If right child is smaller than smallest so far
   if (right < n && arr[right] < arr[smallest])
      smallest = right;

   // If smallest is not root
   if (smallest != i) {
      // Swap the found smallest element with the root
      int temp = arr[i];
      arr[i] = arr[smallest];
      arr[smallest] = temp;

      // Recursively heapify the affected sub-tree
      minHeapify(arr, n, smallest);
   }
}

// Function to build a min heap from an array
void buildMinHeap(int arr[], int n) {
```

```
    // Start from the last non-leaf node and heapify all nodes in reverse order
    for (int i = n / 2 - 1; i >= 0; i--)
        minHeapify(arr, n, i);
}

// Function to print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; ++i)
        printf("%d ", arr[i]);
    printf("\n");
}

// Test the heap construction function
int main() {
    // Test with a random array
    int randomArray[] = {4, 10, 3, 5, 15};
    int n1 = sizeof(randomArray) / sizeof(randomArray[0]);

    printf("\nOriginal Random Array: ");
    printArray(randomArray, n1);

    // Build heap
    buildMinHeap(randomArray, n1);

    printf("\nMin Heap from Random Array: ");
    printArray(randomArray, n1);

    printf("\n");

    // Test with a sorted array
    int sortedArray[] = {8, 6, 5, 4, 2};
```

```c
    int n2 = sizeof(sortedArray) / sizeof(sortedArray[0]);

    printf("\nOriginal Sorted Array: ");
    printArray(sortedArray, n2);

    // Build heap
    buildMinHeap(sortedArray, n2);

    printf("\nMin Heap from Sorted Array: ");
    printArray(sortedArray, n2);

    return 0;
}
```

/*Implement c Program for DFS*/

```
#include <stdio.h>

#define MAX_VERTICES 100

// Function to perform Depth-First Search (DFS) traversal
void DFS(int graph[MAX_VERTICES][MAX_VERTICES], int visited[MAX_VERTICES], int vertices, int start) {
    printf("%d ", start); // Print the current vertex
    visited[start] = 1;    // Mark the current vertex as visited

    // Visit all adjacent vertices
    for (int i = 0; i < vertices; i++) {
        if (graph[start][i] == 1 && !visited[i]) {
            DFS(graph, visited, vertices, i);
        }
    }
}

int main() {
    int vertices, edges;

    // Input the number of vertices
    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);

    if (vertices <= 0 || vertices > MAX_VERTICES) {
        printf("Invalid number of vertices. Exiting...\n");
        return 1;
    }
```

```
int graph[MAX_VERTICES][MAX_VERTICES] = {0}; // Initialize the adjacency matrix with zeros
int visited[MAX_VERTICES] = {0};            // Initialize the visited array with zeros

// Input the number of edges
printf("Enter the number of edges: ");
scanf("%d", &edges);

if (edges < 0 || edges > vertices * (vertices - 1)) {
    printf("Invalid number of edges. Exiting...\n");
    return 1;
}

// Input edges and construct the adjacency matrix
for (int i = 0; i < edges; i++) {
    int start, end;
    printf("Enter edge %d (start end): ", i + 1);
    scanf("%d %d", &start, &end);

    // Validate input vertices
    if (start < 0 || start >= vertices || end < 0 || end >= vertices) {
        printf("Invalid vertices. Try again.\n");
        i--;
        continue;
    }

    graph[start][end] = 1;
    // For undirected graph, uncomment the following line:
    // graph[end][start] = 1;
}
```

```
    // Input the starting vertex for DFS traversal
    int startVertex;
    printf("Enter the starting vertex for DFS traversal: ");
    scanf("%d", &startVertex);

    if (startVertex < 0 || startVertex >= vertices) {
        printf("Invalid starting vertex. Exiting...\n");
        return 1;
    }

    printf("DFS Traversal Order: ");
    DFS(graph, visited, vertices, startVertex);

    return 0;
}
```

```
/*
 * C program to implement BFS using adjacency matrix
 */

#include <stdio.h>

int n, i, j, visited[10], queue[10], front = -1, rear = -1;
int adj[10][10];

void bfs(int v)
{
   for (i = 1; i <= n; i++)
      if (adj[v][i] && !visited[i])
         queue[++rear] = i;
   if (front <= rear)
   {
      visited[queue[front]] = 1;
      bfs(queue[front++]);
   }
}

void main()
{
   int v;
   printf("Enter the number of vertices: ");
   scanf("%d", &n);
   for (i = 1; i <= n; i++)
   {
      queue[i] = 0;
      visited[i] = 0;
   }
```

```
    printf("Enter graph data in matrix form:    \n");
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            scanf("%d", &adj[i][j]);
    printf("Enter the starting vertex: ");
    scanf("%d", &v);
    bfs(v);
    printf("The node which are reachable are:    \n");
    for (i = 1; i <= n; i++)
        if (visited[i])
            printf("%d\t", i);
        else
            printf("BFS is not possible. Not all nodes are reachable");
    return 0;
}
```

// C program for Dijkstra's single source shortest path

// algorithm. The program is for adjacency matrix

// representation of the graph

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

// Number of vertices in the graph
#define V 9

// A utility function to find the vertex with minimum
// distance value, from the set of vertices not yet included
// in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
   // Initialize min value
   int min = INT_MAX, min_index;

   for (int v = 0; v < V; v++)
      if (sptSet[v] == false && dist[v] <= min)
         min = dist[v], min_index = v;

   return min_index;
}

// A utility function to print the constructed distance
// array
void printSolution(int dist[])
{
   printf("Vertex \t\t Distance from Source\n");
```

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

// Function that implements Dijkstra's single source
// shortest path algorithm for a graph represented using
// adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
    int dist[V]; // The output array.  dist[i] will hold the
                 // shortest
    // distance from src to i

    bool sptSet[V]; // sptSet[i] will be true if vertex i is
                    // included in shortest
    // path tree or shortest distance from src to i is
    // finalized

    // Initialize all distances as INFINITE and stpSet[] as
    // false
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum distance vertex from the set of
        // vertices not yet processed. u is always equal to
        // src in the first iteration.
```

```
        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed
        sptSet[u] = true;

        // Update dist value of the adjacent vertices of the
        // picked vertex.
        for (int v = 0; v < V; v++)

            // Update dist[v] only if is not in sptSet,
            // there is an edge from u to v, and total
            // weight of path from src to  v through u is
            // smaller than current value of dist[v]
            if (!sptSet[v] && graph[u][v]
                && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist);
}

// driver's code
int main()
{
    /* Let us create the example graph discussed above */
    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                        { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                        { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
```

```
                    { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                    { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                    { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                    { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                    { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

    // Function call
    dijkstra(graph, 0);

    return 0;
}
```

// C Program for Floyd Warshall Algorithm

```
#include <stdio.h>

// Number of vertices in the graph
#define V 4

/* Define Infinite as a large enough
  value. This value will be used
  for vertices not connected to each other */
#define INF 99999

// A function to print the solution matrix
void printSolution(int dist[][V]);

// Solves the all-pairs shortest path
// problem using Floyd Warshall algorithm
void floydWarshall(int dist[][V])
{
   int i, j, k;

   /* Add all vertices one by one to
     the set of intermediate vertices.
     ---> Before start of an iteration, we
     have shortest distances between all
     pairs of vertices such that the shortest
     distances consider only the
     vertices in set {0, 1, 2, .. k-1} as
     intermediate vertices.
     ----> After the end of an iteration,
     vertex no. k is added to the set of
     intermediate vertices and the set
```

```
      becomes {0, 1, 2, .. k} */
    for (k = 0; k < V; k++) {
        // Pick all vertices as source one by one
        for (i = 0; i < V; i++) {
            // Pick all vertices as destination for the
            // above picked source
            for (j = 0; j < V; j++) {
                // If vertex k is on the shortest path from
                // i to j, then update the value of
                // dist[i][j]
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    // Print the shortest distance matrix
    printSolution(dist);
}

/* A utility function to print solution */
void printSolution(int dist[][V])
{
    printf(
        "The following matrix shows the shortest distances"
        " between every pair of vertices \n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
```

```
            printf("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

// driver's code
int main()
{
    /* Let us create the following weighted graph
            10
       (0)------->(3)
        |         /|\
      5 |          |
        |          | 1
       \|/         |
       (1)------->(2)
            3           */
    int graph[V][V] = { { 0, 5, INF, 10 },
                        { INF, 0, 3, INF },
                        { INF, INF, 0, 1 },
                        { INF, INF, INF, 0 } };

    // Function call
    floydWarshall(graph);
    return 0;
}
```

// C Program for Kruskal Algorithm

```
#include<stdio.h>
void main()
{
 int a,b,n,ne=1,i,j,min,cost[10] [10],mincost=0;
 printf("\n Enter The no of Vertices=");
 scanf("%d",&n);
 printf("\n Enter The adj Matrix\n");
 for(i=1;i<=n;i++)
 {
	for(j=1;j<=n;j++)
		{
			scanf("%d",&cost[i][j]);
			if(cost[i][j]==0)
				{
				cost[i][j]=999;
				}
		}
 }
 while(ne<n)
 {
	min=999;
	for(i=1;i<=n;i++)
		{
		for(j=1;j<=n;j++)
			{

			if(cost[i][j]<min)
				{
				min=cost[i][j];
```

```
                a=i;
                b=j;
                }
            }
        }
printf("edge(%d,%d)=%d\n",a,b,min);
mincost=mincost+min;

cost[a][b]=cost[b][a]=999;
ne++;
 }
printf("\nMinmum spanning Tree of wt=%d",mincost);
}
```

// C Program for Prims Algorithm

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

#define MAX_VERTICES 100

// Function to find the vertex with the minimum key value
int minKey(int key[], int mstSet[], int vertices) {
    int min = INT_MAX, minIndex;

    for (int v = 0; v < vertices; v++) {
        if (!mstSet[v] && key[v] < min) {
            min = key[v];
            minIndex = v;
        }
    }

    return minIndex;
}

// Function to print the constructed MST stored in parent[]
void printMST(int parent[], int graph[MAX_VERTICES][MAX_VERTICES], int vertices) {
    printf("Edge \tWeight\n");
    for (int i = 1; i < vertices; i++) {
        printf("%d - %d \t%d\n", parent[i], i, graph[i][parent[i]]);
    }
}

// Function to implement Prim's algorithm for a given graph
void primMST(int graph[MAX_VERTICES][MAX_VERTICES], int vertices) {
```

```c
int parent[MAX_VERTICES]; // Array to store the constructed MST
int key[MAX_VERTICES];    // Key values used to pick the minimum weight edge
int mstSet[MAX_VERTICES]; // To represent set of vertices included in MST

// Initialize all keys as INFINITE and mstSet[] as false
for (int i = 0; i < vertices; i++) {
  key[i] = INT_MAX;
  mstSet[i] = 0;
}

// Always include the first vertex in the MST
key[0] = 0; // Make key 0 so that this vertex is picked as the first vertex
parent[0] = -1; // First node is always the root of the MST

// The MST will have vertices-1 edges
for (int count = 0; count < vertices - 1; count++) {
  // Pick the minimum key vertex from the set of vertices not yet included in the MST
  int u = minKey(key, mstSet, vertices);

  // Add the picked vertex to the MST Set
  mstSet[u] = 1;

  // Update key value and parent index of the adjacent vertices
  for (int v = 0; v < vertices; v++) {
    // graph[u][v] is non-zero only for adjacent vertices of m
    // mstSet[v] is false for vertices not yet included in MST
    // Update the key only if the graph[u][v] is smaller than the key[v]
    if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {
      parent[v] = u;
      key[v] = graph[u][v];
    }
```

```c
        }
    }

    // Print the constructed MST
    printMST(parent, graph, vertices);
}

int main() {
    int vertices;

    // Input the number of vertices
    printf("Input the number of vertices: ");
    scanf("%d", &vertices);

    if (vertices <= 0 || vertices > MAX_VERTICES) {
        printf("Invalid number of vertices. Exiting...\n");
        return 1;
    }

    int graph[MAX_VERTICES][MAX_VERTICES];

    // Input the adjacency matrix representing the graph
    printf("Input the adjacency matrix for the graph:\n");
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    // Perform Prim's algorithm to find the MST
    primMST(graph, vertices);
```

```
    return 0;
}
```