- Implement a Binary search tree (BST) library (btree.h) with operations – create, insert, postorder. Write a menu driven program that performs the above operations.

Btree.h

```
#ifndef BTREE_H
#define BTREE_H

struct Node {
    int key;
    struct Node* left;
    struct Node* right;
};

struct Node* createNode(int key);
struct Node* insert(struct Node* root, int key);
void postorderTraversal(struct Node* root);
void freeTree(struct Node* root);
-----------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include "btree.h"

struct Node* createNode(int key) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->key = key;
    newNode->left = newNode->right = NULL;
    return newNode;
}

struct Node* insert(struct Node* root, int key) {
    if (root == NULL) {
        return createNode(key);
    }
    if (key < root->key) {
        root->left = insert(root->left, key);
    } else if (key > root->key) {
        root->right = insert(root->right, key);
    }
    return root;
}

void postorderTraversal(struct Node* root) {
    if (root != NULL) {
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        printf("%d ", root->key);
    }
}

void freeTree(struct Node* root) {
    if (root != NULL) {
        freeTree(root->left);
        freeTree(root->right);
        free(root);
    }
}
#include <stdio.h>
```

```
#include <stdlib.h>
#include "btree.h"
```

```
int main() {
   struct Node* root = NULL;
   int choice, key;

   do {
      printf("\nBinary Search Tree Operations:\n");
      printf("1. Insert\n");
      printf("2. Postorder Traversal\n");
      printf("3. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch(choice) {
         case 1:
            printf("Enter key to insert: ");
            scanf("%d", &key);
            root = insert(root, key);
            break;
         case 2:
            printf("Postorder Traversal: ");
            postorderTraversal(root);
            printf("\n");
            break;
         case 3:
            printf("Exiting program.\n");
            freeTree(root);
            break;
         default:
            printf("Invalid choice! Please try again.\n");
      }
   } while (choice != 3);

   return 0;
}
```

Write a C program for the Implementation of **Prim’s Minimum spanning** tree algorithm

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

#define V 5  // Number of vertices in the graph

int minKey(int key[], int mstSet[]) {
   int min = INT_MAX, min_index;
   for (int v = 0; v < V; v++) {
      if (mstSet[v] == 0 && key[v] < min) {
         min = key[v];
         min_index = v;
      }
   }
   return min_index;
}

void printMST(int parent[], int graph[V][V]) {
   printf("Edge \tWeight\n");
   for (int i = 1; i < V; i++) {
      printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
   }
}

void primMST(int graph[V][V]) {
   int parent[V];  // Array to store constructed MST
   int key[V];     // Key values used to pick minimum weight edge in cut
   int mstSet[V];  // To represent set of vertices not yet included in MST

   for (int i = 0; i < V; i++) {
      key[i] = INT_MAX;
      mstSet[i] = 0;
   }

   key[0] = 0;
   parent[0] = -1; // First node is always the root of MST

   for (int count = 0; count < V - 1; count++) {
      int u = minKey(key, mstSet);
      mstSet[u] = 1;

      for (int v = 0; v < V; v++) {
         if (graph[u][v] && mstSet[v] == 0 && graph[u][v] < key[v]) {
            parent[v] = u;
            key[v] = graph[u][v];
         }
      }
   }

   printMST(parent, graph);
}

int main() {
```

```
    int graph[V][V] = {{0, 2, 0, 6, 0},
                       {2, 0, 3, 8, 5},
                       {0, 3, 0, 0, 7},
                       {6, 8, 0, 0, 9},
                       {0, 5, 7, 9, 0}};

    primMST(graph);

    return 0;
}
```

Write a C program that accepts the **vertices and edges of a graph** and stores it as an adjacency matrix. Display the adjacency matrix.

```
#include <stdio.h>

#define MAX_VERTICES 10

int main() {
    int vertices, edges;
    int adjacencyMatrix[MAX_VERTICES][MAX_VERTICES] = {0};

    printf("Enter the number of vertices in the graph: ");
    scanf("%d", &vertices);
    if (vertices <= 0 || vertices > MAX_VERTICES) {
        printf("Invalid number of vertices. Exiting.\n");
        return 1;
    }

    printf("Enter the number of edges in the graph: ");
    scanf("%d", &edges);
    if (edges < 0 || edges > vertices * (vertices - 1) / 2) {
        printf("Invalid number of edges. Exiting.\n");
        return 1;
    }

    printf("Enter the edges (vertex1 vertex2 weight):\n");
    for (int i = 0; i < edges; i++) {
        int vertex1, vertex2, weight;
        scanf("%d %d %d", &vertex1, &vertex2, &weight);
        if (vertex1 < 0 || vertex1 >= vertices || vertex2 < 0 || vertex2 >= vertices) {
            printf("Invalid edge. Exiting.\n");
            return 1;
        }
        adjacencyMatrix[vertex1][vertex2] = weight;
        adjacencyMatrix[vertex2][vertex1] = weight;  // For undirected graph
    }

    printf("Adjacency Matrix:\n");
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            printf("%d ", adjacencyMatrix[i][j]);
        }
        printf("\n");
```

```
        }
        return 0;
```

Topological sort

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

#define MAX_VERTICES 100

struct Node {
   int vertex;
   struct Node* next;
};

struct Graph {
   int numVertices;
   struct Node** adjLists;
   int* visited;
};

struct Node* createNode(int v) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->vertex = v;
   newNode->next = NULL;
   return newNode;
}

struct Graph* createGraph(int vertices) {
   struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
   graph->numVertices = vertices;

   graph->adjLists = (struct Node**)malloc(vertices * sizeof(struct Node*));
   graph->visited = (int*)malloc(vertices * sizeof(int));
```

```
    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }

    return graph;
}

void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;
}

void topologicalSortUtil(int v, struct Graph* graph, int visited[], struct Node* stack) {
    visited[v] = 1;

    struct Node* temp = graph->adjLists[v];
    while (temp != NULL) {
        int adjVertex = temp->vertex;
        if (!visited[adjVertex]) {
            topologicalSortUtil(adjVertex, graph, visited, stack);
        }
        temp = temp->next;
    }

    struct Node* newNode = createNode(v);
    newNode->next = stack;
    stack = newNode;
}

void topologicalSort(struct Graph* graph) {
    int* visited = (int*)malloc(graph->numVertices * sizeof(int));
    struct Node* stack = NULL;
```

```
    for (int i = 0; i < graph->numVertices; i++) {
        visited[i] = 0;
    }

    for (int i = 0; i < graph->numVertices; i++) {
        if (!visited[i]) {
            topologicalSortUtil(i, graph, visited, stack);
        }
    }

    printf("Topological Sorting: ");
    while (stack != NULL) {
        printf("%d ", stack->vertex);
        stack = stack->next;
    }
    printf("\n");

    free(visited);
}

int main() {
    int vertices, edges;
    printf("Enter the number of vertices in the graph: ");
    scanf("%d", &vertices);
    printf("Enter the number of edges in the graph: ");
    scanf("%d", &edges);

    struct Graph* graph = createGraph(vertices);

    printf("Enter the edges (src dest):\n");
    for (int i = 0; i < edges; i++) {
        int src, dest;
        scanf("%d %d", &src, &dest);
        addEdge(graph, src, dest);
    }
```

```
 topologicalSort(graph);
  return 0;
}
```

Write a C program for the implementation of **Floyd Warshall's algorithm** for finding all pairs shortest path using adjacency cost matrix.

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

#define V 4 // Number of vertices

void printSolution(int dist[][V]) {
    printf("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] == INT_MAX)
                printf("INF\t");
            else
                printf("%d\t", dist[i][j]);
        }
        printf("\n");
    }
}

void floydWarshall(int graph[][V]) {
    int dist[V][V];

    // Initialize the distance matrix
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            dist[i][j] = graph[i][j];
        }
    }
```

```
    // Apply the Floyd-Warshall algorithm
    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][k] != INT_MAX && dist[k][j] != INT_MAX && dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }

    printSolution(dist);
}

int main() {
    int graph[V][V] = {{0, 5, INT_MAX, 10},
                       {INT_MAX, 0, 3, INT_MAX},
                       {INT_MAX, INT_MAX, 0, 1},
                       {INT_MAX, INT_MAX, INT_MAX, 0}};

    floydWarshall(graph);

    return 0;
}
#include <stdio.h>
#include <stdlib.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void heapify(int arr[], int n, int i) {
    int largest = i;    // Initialize largest as root
```

```
    int left = 2 * i + 1;   // Left child
    int right = 2 * i + 2;  // Right child

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // If right child is larger than largest so far
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // If largest is not root
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);  // Recursively heapify the affected subtree
    }
}

void heapSort(int arr[], int n) {
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Extract elements from heap one by one
    for (int i = n - 1; i > 0; i--) {
        swap(&arr[0], &arr[i]);  // Move current root to end
        heapify(arr, i, 0);      // Max heapify the reduced heap
    }
}

void printArray(int arr[], int n) {
    printf("Sorted array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
```

```
}

int main() {
    int n;
    printf("Enter the number of elements to sort: ");
    scanf("%d", &n);

    int* arr = (int*)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    printf("Generating %d random elements...\n", n);
    for (int i = 0; i < n; i++)
        arr[i] = rand() % 100;  // Generate random numbers between 0 and 99

    printf("Original array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    heapSort(arr, n);

    printArray(arr, n);

    free(arr);

    return 0;
}
```

Write a program to sort n randomly generated elements using Heap sort method.

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

```c
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void heapify(int arr[], int n, int i) {
    int largest = i;    // Initialize largest as root
    int left = 2 * i + 1;   // Left child
    int right = 2 * i + 2;  // Right child

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // If right child is larger than largest so far
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // If largest is not root
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);  // Recursively heapify the affected subtree
    }
}

void heapSort(int arr[], int n) {
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Extract elements from heap one by one
    for (int i = n - 1; i > 0; i--) {
        swap(&arr[0], &arr[i]);  // Move current root to end
```

```
        heapify(arr, i, 0);      // Max heapify the reduced heap
    }
}

void printArray(int arr[], int n) {
    printf("Sorted array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int n;
    printf("Enter the number of elements to sort: ");
    scanf("%d", &n);

    int* arr = (int*)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    printf("Generating %d random elements...\n", n);
    for (int i = 0; i < n; i++)
        arr[i] = rand() % 100;  // Generate random numbers between 0 and 99

    printf("Original array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    heapSort(arr, n);

    printArray(arr, n);
```

```
    free(arr);
   return 0;
}
```

Write a C program for the Implementation of Kruskal’s Minimum spanning tree algorithm.

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

#define MAX_EDGES 20

struct Edge {
    int src, dest, weight;
};

struct Graph {
    int V, E;
    struct Edge* edge;
};

struct Graph* createGraph(int V, int E) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = V;
    graph->E = E;
    graph->edge = (struct Edge*)malloc(E * sizeof(struct Edge));
    return graph;
}

struct Subset {
    int parent;
    int rank;
};
```

```
int find(struct Subset subsets[], int i) {

    if (subsets[i].parent != i)

        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;

}


void Union(struct Subset subsets[], int x, int y) {

    int xroot = find(subsets, x);

    int yroot = find(subsets, y);


    if (subsets[xroot].rank < subsets[yroot].rank)

        subsets[xroot].parent = yroot;

    else if (subsets[xroot].rank > subsets[yroot].rank)

        subsets[yroot].parent = xroot;

    else {

        subsets[yroot].parent = xroot;

        subsets[xroot].rank++;

    }

}


int compare(const void* a, const void* b) {

    struct Edge* edge1 = (struct Edge*)a;

    struct Edge* edge2 = (struct Edge*)b;

    return edge1->weight - edge2->weight;

}


void KruskalMST(struct Graph* graph) {

    int V = graph->V;

    struct Edge result[V];

    int e = 0;

    int i = 0;


    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), compare);


    struct Subset* subsets = (struct Subset*)malloc(V * sizeof(struct Subset));
```

```
    for (int v = 0; v < V; v++) {

        subsets[v].parent = v;

        subsets[v].rank = 0;

    }


    while (e < V - 1 && i < graph->E) {

        struct Edge next_edge = graph->edge[i++];


        int x = find(subsets, next_edge.src);

        int y = find(subsets, next_edge.dest);


        if (x != y) {

            result[e++] = next_edge;

            Union(subsets, x, y);

        }

    }


    printf("Edges in the Minimum Spanning Tree:\n");

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

        printf("%d - %d : %d\n", result[i].src, result[i].dest, result[i].weight);

    }


    free(subsets);

}


int main() {

    int V = 4;  // Number of vertices

    int E = 5;  // Number of edges


    struct Graph* graph = createGraph(V, E);


    // Edge format: src, dest, weight

    graph->edge[0].src = 0;

    graph->edge[0].dest = 1;

    graph->edge[0].weight = 10;
```

```
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 15;

    graph->edge[4].src = 2;
    graph->edge[4].dest = 3;
    graph->edge[4].weight = 4;

    KruskalMST(graph);

    free(graph->edge);
    free(graph);

    return 0;
}
```

Write a C program for the implementation of **Dijkstra's shortest path algorithm** for finding shortest path from a given source vertex using adjacency cost matrix.

```c
#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#include <stdbool.h>


#define V 6  // Number of vertices


int minDistance(int dist[], bool sptSet[]) {

    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;

}


void printSolution(int dist[]) {

    printf("Vertex \t Distance from Source\n");

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

        printf("%d \t %d\n", i, dist[i]);

    }

}


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 the shortest path tree or shortest distance from src to i is finalized
```

```
    for (int i = 0; i < V; i++) {
        dist[i] = INT_MAX;
        sptSet[i] = false;
    }

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

    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = true;

        for (int v = 0; v < V; 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];
            }
        }
    }

    printSolution(dist);
}

int main() {
    int graph[V][V] = {{0, 1, 4, 0, 0, 0},
                       {1, 0, 4, 2, 7, 0},
                       {4, 4, 0, 3, 5, 0},
                       {0, 2, 3, 0, 4, 6},
                       {0, 7, 5, 4, 0, 7},
                       {0, 0, 0, 6, 7, 0}};

    int src = 0;  // Source vertex

    dijkstra(graph, src);

    return 0;
}
```