Prim’s Algorithm:

```
#include <stdio.h>

#include <limits.h>

#include <stdbool.h>

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

// Function to find the vertex with the minimum key value from the set of vertices not yet included in MST

int minKey(int key[], bool mstSet[]) {

   int min = INT_MAX, min_index;

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

      if (mstSet[v] == false && key[v] < min)

         min = key[v], min_index = v;

   return min_index;

}

// Function to print the constructed MST stored in parent[]

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]]);

}


// Function that implements Prim's Algorithm

void primMST(int graph[V][V]) {

   int parent[V];  // Array to store the constructed MST

   int key[V];     // Key values to pick the minimum weight edge

   bool mstSet[V]; // To represent set of vertices included in MST


   // Initialize all keys as INFINITE

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

      key[i] = INT_MAX, mstSet[i] = false;
```

```
    // Include first vertex in 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 MST

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

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

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

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

int main() {
    // Example graph represented as an adjacency matrix
    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}
    };

    // Function call to Prim's algorithm
    primMST(graph);

    return 0;
}
```

Kruskal's Algorithm:

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

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

// Structure to represent an edge
struct Edge {
    int src, dest, weight;
};

// Structure to represent a graph
struct Graph {
    int V, E;  // V = vertices, E = edges
    struct Edge* edges;  // Array of edges
};

// Structure to represent a subset for Union-Find
struct subset {
    int parent;
```

```c
    int rank;
};

// Function to create a graph with V vertices and E edges
struct Graph* createGraph(int V, int E) {
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;
    graph->E = E;
    graph->edges = (struct Edge*) malloc(graph->E * sizeof(struct Edge));
    return graph;
}

// A utility function to find the subset of an element i (uses path compression)
int find(struct subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

// A function that does union of two sets of x and y (uses union by rank)
void Union(struct subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of the higher rank tree
    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++;
    }
}

// Compare two edges according to their weights (for qsort)
int compare(const void* a, const void* b) {
    struct Edge* a1 = (struct Edge*) a;
    struct Edge* b1 = (struct Edge*) b;
    return a1->weight > b1->weight;
}

// Function to print the MST
void printMST(struct Edge result[], int e) {
    printf("Edges in the Minimum Spanning Tree:\n");
    for (int i = 0; i < e; ++i)
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);
}

// Function to implement Kruskal's algorithm
void KruskalMST(struct Graph* graph) {
    int V = graph->V;
    struct Edge result[V];  // Array to store the resulting MST
    int e = 0;  // Index for result[]
    int i = 0;  // Index for sorted edges

    // Step 1: Sort all the edges in non-decreasing order of their weight
    qsort(graph->edges, graph->E, sizeof(graph->edges[0]), compare);

    // Allocate memory for creating V subsets
```

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

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
    while (e < V - 1 && i < graph->E) {
        // Step 2: Pick the smallest edge. Check if it forms a cycle with the MST formed so far.
        struct Edge next_edge = graph->edges[i++];

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

        // If including this edge does not cause a cycle, include it in the result
        if (x != y) {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
    }

    // Print the resulting MST
    printMST(result, e);

    // Free memory
    free(subsets);
}
```

```c
int main() {
    // Number of vertices and edges in the graph
    int V = 4;  // Vertices
    int E = 5;  // Edges
    struct Graph* graph = createGraph(V, E);

    // Adding the edges
    graph->edges[0].src = 0;
    graph->edges[0].dest = 1;
    graph->edges[0].weight = 10;

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

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

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

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

    // Call Kruskal's algorithm to find MST
    KruskalMST(graph);
```

```
    // Free memory
    free(graph->edges);
    free(graph);

    return 0;
}
```