```c

/**
 * Implementation of Prim's algorithm to build an MST
 * Time Complexity: O(|E|.log|V|)
 *
 * Author: Mithusayel Murmu
 */

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

#define GRAPH_SZ 50
typedef enum { FALSE, TRUE } BOOL;

/** Rudimentary Graph definition */
typedef struct _Graph Graph;
typedef struct _GraphNode GraphNode;

struct _Graph {
    size_t size;
    GraphNode *nodeList[GRAPH_SZ];
    int adjList[GRAPH_SZ][GRAPH_SZ][2];     // [i][0] => adjNodeID; [i][1] => edgeWeight
};
struct _GraphNode {
    int id;                                 // Unique ID per node
    int key;                                // To be used in Prim's algo
    size_t adjSize;
    BOOL heapCutSet;
};

/** Min-Heap implementation */
#define HEAP_DTYPE GraphNode

typedef int (*compare_func)(int, int);
typedef struct _MinHeap MinHeap;

struct _MinHeap {
    size_t size;
    HEAP_DTYPE *arr[GRAPH_SZ];
    int idxMap[GRAPH_SZ];  // Internal mapping: GraphNode.id => Min-Heap.index
};

static inline void heap_swap_data(MinHeap *heap, size_t i1, size_t i2) {
    // Update internal mappings
    heap->idxMap[heap->arr[i1]->id] = i2;
    heap->idxMap[heap->arr[i2]->id] = i1;

    HEAP_DTYPE *temp;
    temp = heap->arr[i1]; heap->arr[i1] = heap->arr[i2]; heap->arr[i2] = temp;
}

void heap_init(MinHeap *heap, HEAP_DTYPE **arr, size_t size) {
    heap->size = size;
    memcpy(heap->arr, arr, size * sizeof(HEAP_DTYPE *));

    // Build internal index map
    int i;
    for (i = 0; i < heap->size; i++)
        heap->idxMap[heap->arr[i]->id] = i;
}

/**
 * Recursively min-heapify a node at index idx
 * @heap:       The heap to use
 * @idx:        Index of node to min-heapify
 */
void min_heapify(MinHeap *heap, size_t idx) {
    int lt = 2 * idx + 1;
    int rt = 2 * (idx + 1);
    int min;

    if (lt < heap->size && heap->arr[idx]->key > heap->arr[lt]->key)
        min = lt;
    else
        min = idx;
    if (rt < heap->size && heap->arr[min]->key > heap->arr[rt]->key)
        min = rt;
```

```c

    if (idx != min) {
        heap_swap_data(heap, idx, min);
        min_heapify(heap, min);
    }
}

/**
 * Builds a min-heap out of the given heap
 * @heap:       The heap to use
 * @asz:        Size of the array
 */
void build_min_heap(MinHeap *heap) {
    int i;
    for (i = heap->size / 2 - 1; i >= 0; i--)
        min_heapify(heap, i);
}

HEAP_DTYPE * heap_extract_min(MinHeap *heap) {
    if (heap->size == 0) return NULL;

    HEAP_DTYPE *node = heap->arr[0];
    heap_swap_data(heap, 0, heap->size - 1);
    heap->size--; min_heapify(heap, 0);

    return node;
}

/** Parent index */
#define PIDX(x) (((x)-1)/2)

void heap_decrease_key(MinHeap *heap, HEAP_DTYPE *node, int nkey) {
    if (nkey > node->key) return;

    node->key = nkey;
    int i = heap->idxMap[node->id];

    while (i > 0 && nkey < heap->arr[PIDX(i)]->key) {
        heap_swap_data(heap, i, PIDX(i)); i = PIDX(i);
    }
}

Graph * graph_create() {
    Graph *graph = (Graph *) malloc(sizeof(Graph));
    size_t i;
    for (i = 0; i < GRAPH_SZ; ++i)
        graph->nodeList[i] = NULL;
    graph->size = 0; return graph;
}

void graph_destroy(Graph *graph) {
    size_t i;
    for (i = 0; i < graph->size; i++) {
        free(graph->nodeList[i]);
        graph->nodeList[i] = NULL;
    }

    graph->size = 0; free(graph);
}

GraphNode * graph_create_node(const int _id) {
    GraphNode *node = (GraphNode *) malloc(sizeof(GraphNode));
    node->id = _id; node->key = 0; node->adjSize = 0;
    node->heapCutSet = FALSE; return node;
}

static void prim_init(Graph *graph, MinHeap *heap) {
    if (graph == NULL || graph->size == 0)
        return;

    size_t i;
    for (i = 0; i < graph->size; i++) {
        // Emulate positive infinity
        graph->nodeList[i]->key = INT_MAX;
        graph->nodeList[i]->heapCutSet = FALSE;
    }
    graph->nodeList[0]->key = 0;

    heap_init(heap, graph->nodeList, graph->size);
```

```c
    build_min_heap(heap);
}

void prim_print_mst(Graph *graph, MinHeap *heap, void (*callback)(HEAP_DTYPE *)) {
    prim_init(graph, heap);

    size_t i, wt;
    GraphNode *node, *adj_node;

    while (heap->size) {
        node = heap_extract_min(heap);
        node->heapCutSet = TRUE; callback(node);

        // Iterate through adjacent nodes
        for (i = 0; i < node->adjSize; i++) {
            adj_node = graph->nodeList[graph->adjList[node->id][i][0]];
            wt = graph->adjList[node->id][i][1];

            if (wt < adj_node->key && !adj_node->heapCutSet)
                heap_decrease_key(heap, adj_node, wt);
        }
    }
}

#define _scand(n) scanf("%d", &(n))
void graph_input(Graph *graph) {
    int vs, asz, vi, i, j;

    _scand(vs);                     // Number of vertices
    graph->size = vs;
    for (i = 0; i < vs; ++i) {
        _scand(asz);                // Adjacency list size
        GraphNode *node = graph_create_node(i);
        node->adjSize = asz;
        for (j = 0; j < asz; ++j) {
            _scand(vi);             // Scan adjacent node's ID
            graph->adjList[i][j][0] = vi;
            _scand(vi);             // Scan edge weight for the adjacent node
            graph->adjList[i][j][1] = vi;
        }

        graph->nodeList[i] = node;
    }
}

static void print_utility(HEAP_DTYPE *node) { printf("%d ", node->id); }

/** Driver function */
int main(int argc, char const *argv[]) {
    Graph *graph = graph_create();
    MinHeap heap;

    printf("Enter graph data:\n");
    graph_input(graph);

    printf("\nMST result:\n");
    prim_print_mst(graph, &heap, print_utility);
    printf("\n"); graph_destroy(graph);

    return 0;
}

```