PRIM ALGO

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

typedef struct node
{
  int vertex;
  int weight;
  struct node *next;
} node;

typedef struct Graph
{
  int numVertices;
  struct node **adjLists;
} Graph;

typedef struct MinHeapNode
{
  int v;
  int edgeWeight;
} MinHeapNode;

typedef struct MinHeap
{
  int size;
  int capacity;
  int *pos;
  struct MinHeapNode **array;
} MinHeap;

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

node *createNode(int v, int weight)
{
  node *newNode = (node *)malloc(sizeof(node));
  newNode->vertex = v;
  newNode->weight = weight;
  newNode->next = NULL;
  return newNode;
}

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

  graph->adjLists = (node **)malloc(vertices * sizeof(node *));

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

```
  return graph;
}

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

  newNode = createNode(src, weight);
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}

MinHeapNode *newMinHeapNode(int v, int key)
{
  MinHeapNode *minHeapNode = (MinHeapNode *)malloc(sizeof(MinHeapNode));
  minHeapNode->v = v;
  minHeapNode->edgeWeight = key;
  return minHeapNode;
}

MinHeap *createMinHeap(int capacity)
{
  MinHeap *minHeap = (MinHeap *)malloc(sizeof(MinHeap));
  minHeap->pos = (int *)malloc(capacity * sizeof(int));
  minHeap->size = 0;
  minHeap->capacity = capacity;
  minHeap->array = (MinHeapNode **)malloc(capacity * sizeof(MinHeapNode *));
  return minHeap;
}

void swapMinHeapNode(MinHeapNode **a, MinHeapNode **b)
{
  MinHeapNode *t = *a;
  *a = *b;
  *b = t;
}

void minHeapify(MinHeap *minHeap, int idx)
{
  int smallest, left, right;
  smallest = idx;
  left = 2 * idx + 1;
  right = 2 * idx + 2;

  if (left < minHeap->size && minHeap->array[left]->edgeWeight < minHeap-
>array[smallest]->edgeWeight)
    smallest = left;

  if (right < minHeap->size && minHeap->array[right]->edgeWeight < minHeap-
>array[smallest]->edgeWeight)
    smallest = right;

  if (smallest != idx)
  {
    MinHeapNode *smallestNode = minHeap->array[smallest];
```

```
    MinHeapNode *idxNode = minHeap->array[idx];

    minHeap->pos[smallestNode->v] = idx;
    minHeap->pos[idxNode->v] = smallest;

    swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

    minHeapify(minHeap, smallest);
  }
}

int isEmpty(MinHeap *minHeap)
{
  return minHeap->size == 0;
}

MinHeapNode *extractMin(MinHeap *minHeap)
{
  if (isEmpty(minHeap))
    return NULL;

  MinHeapNode *root = minHeap->array[0];

  MinHeapNode *lastNode = minHeap->array[minHeap->size - 1];
  minHeap->array[0] = lastNode;

  minHeap->pos[root->v] = minHeap->size - 1;
  minHeap->pos[lastNode->v] = 0;

  --minHeap->size;
  minHeapify(minHeap, 0);

  return root;
}

void decreaseKey(MinHeap *minHeap, int v, int key)
{
  int i = minHeap->pos[v];

  minHeap->array[i]->edgeWeight = key;

  while (i && minHeap->array[i]->edgeWeight < minHeap->array[(i - 1) / 2]-
>edgeWeight)
  {
    minHeap->pos[minHeap->array[i]->v] = (i - 1) / 2;
    minHeap->pos[minHeap->array[(i - 1) / 2]->v] = i;
    swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

    i = (i - 1) / 2;
  }
}

int isInMinHeap(MinHeap *minHeap, int v)
{
  if (minHeap->pos[v] < minHeap->size)
    return 1;
  return 0;
}
```

```c
void printGraph(int parent[], int n, int key[])
{
  printf("Edge   Weight\n");
  int minWeight = 0;
  for (int i = 1; i < n; i++)
    printf("%c - %c    %d \n", ((char)parent[i] + 65), ((char)i + 65), key[i]);
  for (int i = 1; i < n; i++)
    minWeight += key[i];
  printf("Minimum Weight: %d\n", minWeight);
}

void PrimMST(Graph *graph)
{
  int V = graph->numVertices;
  int *parent = (int *)malloc(V * sizeof(int));
  int *key = (int *)malloc(V * sizeof(int));
  MinHeap *minHeap = createMinHeap(V);
  int v;

  for (v = 1; v < V; ++v)
  {
    parent[v] = -1;
    key[v] = INT_MAX;
    minHeap->array[v] = newMinHeapNode(v, key[v]);
    minHeap->pos[v] = v;
  }

  key[0] = 0;
  minHeap->array[0] = newMinHeapNode(0, key[0]);
  minHeap->pos[0] = 0;
  minHeap->size = V;

  while (!isEmpty(minHeap))
  {
    MinHeapNode *minHeapNode = extractMin(minHeap);
    int u = minHeapNode->v;

    node *temp = graph->adjLists[u];
    while (temp != NULL)
    {
      int v = temp->vertex;

      if (isInMinHeap(minHeap, v) && temp->weight < key[v])
      {
        key[v] = temp->weight;
        parent[v] = u;
        decreaseKey(minHeap, v, key[v]);
      }
      temp = temp->next;
    }
  }

  printGraph(parent, V, key);
}

int main()
{
```

```
  int V = 9;
  Graph *graph = createGraph(V);
  FILE *file = fopen("graph.txt", "r");
  int src, dest, weight;
  while (fscanf(file, "%d %d %d", &src, &dest, &weight) != EOF)
  {
    addEdge(graph, src, dest, weight);
  }
  fclose(file);
  PrimMST(graph);
  return 0;
}
```

GRAPH.TXT(Start the vertices from 0 where 0->A, 1->B.....)

0 1 4

0 2 8

1 2 11

1 3 8

2 4 7

2 5 1

3 4 2

3 7 4

3 6 7

4 5 6

5 7 2

6 7 14

6 8 9

7 8 10

OUTPUT:

Edge Weight

A - B 4

A - C 8

H - D 4

D - E 2

C - F 1

D - G 7

F - H 2

G - I 9

Minimum Weight: 37

SKIP LIST

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

typedef struct node
{
  int val;
  struct node *up, *down, *right, *left;
} node;

node *createNode(int data)
{
  node *newNode = (node *)malloc(sizeof(node));
  newNode->val = data;
  newNode->left = newNode->right = newNode->up = newNode->down = NULL;
  return newNode;
}

node *createSkipList()
{
  node *rightBound = createNode(INT_MAX);
  node *leftBound = createNode(INT_MIN);
  rightBound->left = leftBound;
  leftBound->right = rightBound;
  return leftBound;
}

node *getLevel(node *ele)
{
  while (ele->left)
    ele = ele->left;
  return ele;
}

node *getTopLevel(node *curLevel)
{
  while (curLevel->up)
    curLevel = curLevel->up;
  return curLevel;
}

void addNewLevel(node *curLevel)
{
  node *newLevel = createSkipList();
  newLevel->down = curLevel;
  curLevel->up = newLevel;
  node *curLevelLast = curLevel;
  while (curLevelLast->right)
    curLevelLast = curLevelLast->right;
```

```
  newLevel->right->down = curLevelLast;
  curLevelLast->up = newLevel->right;
}

node *find(node *skipList, int data)
{
  node *level = skipList, *preTemp = NULL, *temp = NULL;
  while (level)
  {
    node *temp = level;
    while (temp)
    {
      int y = temp->right->val;
      if (data == y)
        return temp->right;
      else if (data > y)
        temp = temp->right;
      else
      {
        preTemp = temp->right;
        level = level->down;
        break;
      }
    }
  }
  return preTemp;
}

node *findInSameLevel(node *curLevel, int data)
{
  int y = curLevel->right->val;
  while (y < data)
  {
    curLevel = curLevel->right;
    y = curLevel->right->val;
  }
  return curLevel->right;
}

void insertx(node **skipList, int data)
{
  node *found = find(*skipList, data);
  if (found->val != data)
  {
    node *newNode = createNode(data);
    node *left = found->left;
    left->right = newNode;
    found->left = newNode;
    newNode->left = left;
    newNode->right = found;

    node *curLevel = getLevel(found);
    if (!curLevel->up)
      addNewLevel(curLevel);
    int toss = rand() % 2;
    while (toss)
    {
      curLevel = curLevel->up;
```

```
      found = findInSameLevel(curLevel, data);
      node *curNewNode = createNode(data);
      left = found->left;
      left->right = curNewNode;
      found->left = curNewNode;
      curNewNode->left = left;
      curNewNode->right = found;
      curNewNode->down = newNode;
      newNode->up = curNewNode;
      newNode = curNewNode;
      if (!curLevel->up)
        addNewLevel(curLevel);
      toss = rand() % 2;
    }
    *skipList = getTopLevel(curLevel);
  }
}

void deleteTopLevel(node **skipList)
{
  node *temp = *skipList;
  *skipList = temp->down;
  free(temp->right);
  free(temp);
  (*skipList)->up = (*skipList)->right->up = NULL;
}

void deletex(node **skipList, int data)
{
  node *found = find(*skipList, data);
  if (found->val == data)
  {
    node *curLevel = getLevel(found);
    node *nextEle;
    while (found)
    {
      node *nextEle = found->down;
      node *left = found->left;
      node *right = found->right;
      left->right = right;
      right->left = left;
      free(found);
      found = nextEle;
      if (left->val == INT_MIN && right->val == INT_MAX)
        deleteTopLevel(skipList);
    }
    printf("%d deleted\n", data);
  }
  else
    printf("Element not present\n");
}

void printSkipList(node *skipList)
{
  node *level = skipList;
  while (level)
  {
    node *temp = level;
```

```
    while (temp)
    {
      int val = temp->val;
      if (val == INT_MIN)
        printf("-INF  ");
      else if (val == INT_MAX)
        printf("INF\n");
      else
        printf("%d  ", val);
      temp = temp->right;
    }
    level = level->down;
  }
}

void deleteNodes(node *head)
{
  if (!head)
    return;
  deleteNodes(head->right);
  free(head);
}

void deleteLevels(node *skipListLevel)
{
  if (!skipListLevel)
    return;
  deleteLevels(skipListLevel->down);
  deleteNodes(skipListLevel);
}

int isEmpty(node *skipList)
{
  return (skipList->right->val == INT_MAX && skipList->right->left->val == INT_MIN);
}

int main()
{
  srand(time(NULL));
  int data;
  node *skipList = createSkipList();
  printf("Initially\n");
  printSkipList(skipList);
  while (1)
  {
    int ch;
    printf("Menu:\n");
    printf("1. Insert\n");
    printf("2. Delete\n");
    printf("3. Print\n");
    printf("4. Find\n");
    printf("5. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &ch);
    switch (ch)
    {
    case 1:
      printf("Enter data: ");
```

```
      scanf("%d", &data);
      insertx(&skipList, data);
      break;
    case 2:
      if (!isEmpty(skipList))
      {
        printf("Enter data to delete: ");
        scanf("%d", &data);
        deletex(&skipList, data);
      }
      else
      {
        printf("Skip list is empty\n");
      }
      break;
    case 3:
      printf("Skip list upto now: \n");
      printSkipList(skipList);
      break;
    case 4:
      if (isEmpty(skipList))
      {
        printf("Skip list is empty\n");
        break;
      }
      printf("Enter the data to find: \n");
      scanf("%d", &data);
      node *found = find(skipList, data);
      if (!found)
        printf("Skip list does not exist\n");
      else if (found->val != data)
        printf("data does not exist\n");
      else
        printf("Data exist\n");
      break;
    case 5:
      printf("Exiting...\n");
      deleteLevels(skipList);
      exit(1);
    default:
      printf("ERROR: Entering the choice\n");
    }
  }
  return 0;
}
```