\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1)

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* newNode(int data) {

struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

node->data = data;

node->left = node->right = NULL;

return node;

}

struct Node\* insert(struct Node\* node, int data) {

if (node == NULL) return newNode(data);

if (data < node->data)

node->left = insert(node->left, data);

else

node->right = insert(node->right, data);

return node;

}

void inorder(struct Node\* root) {

if (root) {

inorder(root->left);

printf("%d ", root->data);

inorder(root->right);

}

}

int main() {

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

printf("Inorder traversal: ");

inorder(root);

return 0;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2)

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* newNode(int data) {

struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

node->data = data;

node->left = node->right = NULL;

return node;

}

struct Node\* insert(struct Node\* node, int data) {

if (node == NULL) return newNode(data);

if (data < node->data)

node->left = insert(node->left, data);

else

node->right = insert(node->right, data);

return node;

}

struct Node\* minValueNode(struct Node\* node) {

struct Node\* current = node;

while (current && current->left != NULL)

current = current->left;

return current;

}

struct Node\* deleteNode(struct Node\* root, int data) {

if (root == NULL) return root;

if (data < root->data)

root->left = deleteNode(root->left, data);

else if (data > root->data)

root->right = deleteNode(root->right, data);

else {

if (root->left == NULL) {

struct Node\* temp = root->right;

free(root);

return temp;

} else if (root->right == NULL) {

struct Node\* temp = root->left;

free(root);

return temp;

}

struct Node\* temp = minValueNode(root->right);

root->data = temp->data;

root->right = deleteNode(root->right, temp->data);

}

return root;

}

void inorder(struct Node\* root) {

if (root) {

inorder(root->left);

printf("%d ", root->data);

inorder(root->right);

}

}

int main() {

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

printf("Inorder traversal before deletion: ");

inorder(root);

root = deleteNode(root, 20);

printf("\nInorder traversal after deletion of 20: ");

inorder(root);

root = deleteNode(root, 30);

printf("\nInorder traversal after deletion of 30: ");

inorder(root);

root = deleteNode(root, 50);

printf("\nInorder traversal after deletion of 50: ");

inorder(root);

return 0;

}

3)

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* newNode(int data) {

struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

node->data = data;

node->left = node->right = NULL;

return node;

}

struct Node\* insert(struct Node\* node, int data) {

if (node == NULL) return newNode(data);

if (data < node->data)

node->left = insert(node->left, data);

else

node->right = insert(node->right, data);

return node;

}

void inorder(struct Node\* root) {

if (root) {

inorder(root->left);

printf("%d ", root->data);

inorder(root->right);

}

}

void preorder(struct Node\* root) {

if (root) {

printf("%d ", root->data);

preorder(root->left);

preorder(root->right);

}

}

void postorder(struct Node\* root) {

if (root) {

postorder(root->left);

postorder(root->right);

printf("%d ", root->data);

}

}

int main() {

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

printf("Inorder traversal: ");

inorder(root);

printf("\nPreorder traversal: ");

preorder(root);

printf("\nPostorder traversal: ");

postorder(root);

return 0;

}

4)

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* newNode(int data) {

struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

node->data = data;

node->left = node->right = NULL;

return node;

}

struct Node\* insert(struct Node\* node, int data) {

if (node == NULL) return newNode(data);

if (data < node->data)

node->left = insert(node->left, data);

else

node->right = insert(node->right, data);

return node;

}

void inorderNonRecursive(struct Node\* root) {

struct Node\* stack[100];

int top = -1;

struct Node\* curr = root;

while (curr != NULL || top != -1) {

while (curr != NULL) {

stack[++top] = curr;

curr = curr->left;

}

curr = stack[top--];

printf("%d ", curr->data);

curr = curr->right;

}

}

int main() {

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

printf("Inorder traversal (non-recursive): ");

inorderNonRecursive(root);

return 0;

}

5)

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* newNode(int data) {

struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

node->data = data;

node->left = node->right = NULL;

return node;

}

struct Node\* insert(struct Node\* node, int data) {

if (node == NULL) return newNode(data);

if (data < node->data)

node->left = insert(node->left, data);

else

node->right = insert(node->right, data);

return node;

}

void preorderNonRecursive(struct Node\* root) {

struct Node\* stack[100];

int top = -1;

if (root) stack[++top] = root;

while (top != -1) {

struct Node\* curr = stack[top--];

printf("%d ", curr->data);

if (curr->right) stack[++top] = curr->right;

if (curr->left) stack[++top] = curr->left;

}

}

int main() {

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

printf("Preorder traversal (non-recursive): ");

preorderNonRecursive(root);

return 0;

}

6)

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* newNode(int data) {

struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

node->data = data;

node->left = node->right = NULL;

return node;

}

struct Node\* insert(struct Node\* node, int data) {

if (node == NULL) return newNode(data);

if (data < node->data)

node->left = insert(node->left, data);

else

node->right = insert(node->right, data);

return node;

}

void postorderNonRecursive(struct Node\* root) {

struct Node\* stack[100], \*output[100];

int top1 = -1, top2 = -1;

stack[++top1] = root;

while (top1 != -1) {

struct Node\* curr = stack[top1--];

output[++top2] = curr;

if (curr->left) stack[++top1] = curr->left;

if (curr->right) stack[++top1] = curr->right;

}

while (top2 != -1) {

printf("%d ", output[top2--]->data);

}

}

int main() {

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

printf("Postorder traversal (non-recursive): ");

postorderNonRecursive(root);

return 0;

}

7)

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* newNode(int data) {

struct Node\* node = (struct Node\*)malloc(sizeof(struct Node));

node->data = data;

node->left = node->right = NULL;

return node;

}

struct Node\* createBinaryTree(int arr[], int n) {

struct Node\* nodes[n];

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

nodes[i] = newNode(arr[i]);

}

for (int i = 0; i < n / 2; i++) {

if (2 \* i + 1 < n) nodes[i]->left = nodes[2 \* i + 1];

if (2 \* i + 2 < n) nodes[i]->right = nodes[2 \* i + 2];

}

return nodes[0];

}

void inorder(struct Node\* root) {

if (root) {

inorder(root->left);

printf("%d ", root->data);

inorder(root->right);

}

}

int main() {

int arr[] = {1, 2, 3, 4, 5};

struct Node\* root = createBinaryTree(arr, 5);

printf("Inorder traversal of the binary tree: ");

inorder(root);

return 0;

}

8)

#include <stdio.h>

#include <stdlib.h>

struct AVLNode {

int data;

struct AVLNode\* left;

struct AVLNode\* right;

int height;

};

int height(struct AVLNode\* N) {

if (N == NULL)

return 0;

return N->height;

}

int max(int a, int b) {

return (a > b) ? a : b;

}

struct AVLNode\* newAVLNode(int data) {

struct AVLNode\* node = (struct AVLNode\*)malloc(sizeof(struct AVLNode));

node->data = data;

node->left = NULL;

node->right = NULL;

node->height = 1;

return node;

}

// Right rotate

struct AVLNode\* rightRotate(struct AVLNode\* y) {

struct AVLNode\* x = y->left;

struct AVLNode\* T2 = x->right;

x->right = y;

y->left = T2;

y->height = max(height(y->left), height(y->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

return x;

}

// Left rotate

struct AVLNode\* leftRotate(struct AVLNode\* x) {

struct AVLNode\* y = x->right;

struct AVLNode\* T2 = y->left;

y->left = x;

x->right = T2;

x->height = max(height(x->left), height(x->right)) + 1;

y->height = max(height(y->left), height(y->right)) + 1;

return y;

}

int getBalance(struct AVLNode\* N) {

if (N == NULL)

return 0;

return height(N->left) - height(N->right);

}

struct AVLNode\* insert(struct AVLNode\* node, int data) {

if (node == NULL) return newAVLNode(data);

if (data < node->data)

node->left = insert(node->left, data);

else if (data > node->data)

node->right = insert(node->right, data);

else return node;

node->height = 1 + max(height(node->left), height(node->right)));

int balance = getBalance(node);

if (balance > 1 && data < node->left->data)

return rightRotate(node);

if (balance < -1 && data > node->right->data)

return leftRotate(node);

if (balance > 1 && data > node->left->data) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

if (balance < -1 && data < node->right->data) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

return node;

}

void inorder(struct AVLNode\* root) {

if (root) {

inorder(root->left);

printf("%d ", root->data);

inorder(root->right);

}

}

int main() {

struct AVLNode\* root = NULL;

root = insert(root, 30);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 10);

root = insert(root, 5);

printf("Inorder traversal of the AVL tree: ");

inorder(root);

return 0;

}

9)

#include <stdio.h>

#include <stdlib.h>

#define MIN\_DEGREE 2

typedef struct BTreeNode {

int \*keys;

int t;

struct BTreeNode \*\*C;

int n;

int leaf;

} BTreeNode;

BTreeNode\* createNode(int t, int leaf) {

BTreeNode\* newNode = (BTreeNode\*)malloc(sizeof(BTreeNode));

newNode->t = t;

newNode->leaf = leaf;

newNode->keys = (int\*)malloc((2 \* t - 1) \* sizeof(int));

newNode->C = (BTreeNode\*\*)malloc(2 \* t \* sizeof(BTreeNode\*));

newNode->n = 0;

return newNode;

}

void traverse(BTreeNode\* root) {

for (int i = 0; i < root->n; i++) {

if (!root->leaf)

traverse(root->C[i]);

printf("%d ", root->keys[i]);

}

if (!root->leaf)

traverse(root->C[root->n]);

}

int main() {

BTreeNode\* root = createNode(MIN\_DEGREE, 1);

root->keys[0] = 10;

root->n = 1;

printf("B-Tree traversal: ");

traverse(root);

return 0;

}

10)

#include <stdio.h>

int fibonacciSearch(int arr[], int size, int key) {

int fibM2 = 0;

int fibM1 = 1;

int fibM = fibM1 + fibM2;

while (fibM < size) {

fibM2 = fibM1;

fibM1 = fibM;

fibM = fibM1 + fibM2;

}

int offset = -1;

while (fibM > 1) {

int i = (offset + fibM2 < size - 1) ? offset + fibM2 : size - 1;

if (arr[i] < key) {

fibM = fibM1;

fibM1 = fibM2;

fibM2 = fibM - fibM1;

offset = i;

} else if (arr[i] > key) {

fibM = fibM2;

fibM1 -= fibM1;

fibM2 -= fibM2;

} else return i;

}

if (fibM1 && arr[offset + 1] == key) return offset + 1;

return -1;

}

int main() {

int arr[] = {10, 22, 35, 40, 45, 50, 80, 82, 85, 90, 100};

int n = sizeof(arr) / sizeof(arr[0]);

int key = 85;

int index = fibonacciSearch(arr, n, key);

if (index >= 0)

printf("Found at index: %d\n", index);

else

printf("Not found\n");

return 0;

}

11)

#include <stdio.h>

void swap(int\* a, int\* b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int partition(int arr[], int low, int high) {

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j < high; j++) {

if (arr[j] < pivot) {

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

void quickSort(int arr[], int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

int main() {

int arr[] = {10, 7, 8, 9, 1, 5};

int n = sizeof(arr) / sizeof(arr[0]);

quickSort(arr, 0, n - 1);

printf("Sorted array: ");

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

printf("%d ", arr[i]);

return 0;

}

12)

#include <stdio.h>

void merge(int arr[], int left, int mid, int right) {

int i, j, k;

int n1 = mid - left + 1;

int n2 = right - mid;

int L[n1], R[n2];

for (i = 0; i < n1; i++)

L[i] = arr[left + i];

for (j = 0; j < n2; j++)

R[j] = arr[mid + 1 + j];

i = 0; j = 0; k = left;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++; k++;

}

while (j < n2) {

arr[k] = R[j];

j++; k++;

}

}

void mergeSort(int arr[], int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

int n = sizeof(arr) / sizeof(arr[0]);

mergeSort(arr, 0, n - 1);

printf("Sorted array: ");

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

printf("%d ", arr[i]);

return 0;

}

13)

#include <stdio.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;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if (left < n && arr[left] > arr[largest])

largest = left;

if (right < n && arr[right] > arr[largest])

largest = right;

if (largest != i) {

swap(&arr[i], &arr[largest]);

heapify(arr, n, largest);

}

}

void heapSort(int arr[], int n) {

for (int i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i);

for (int i = n - 1; i >= 0; i--) {

swap(&arr[0], &arr[i]);

heapify(arr, i, 0);

}

}

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

int n = sizeof(arr) / sizeof(arr[0]);

heapSort(arr, n);

printf("Sorted array: ");

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

printf("%d ", arr[i]);

return 0;

}

14)

#include <stdio.h>

void bitonicMerge(int arr[], int low, int cnt, int dir) {

if (cnt > 1) {

int k = cnt / 2;

for (int i = low; i < low + k; i++) {

if (dir == (arr[i] > arr[i + k])) {

int temp = arr[i];

arr[i] = arr[i + k];

arr[i + k] = temp;

}

}

bitonicMerge(arr, low, k, dir);

bitonicMerge(arr, low + k, k, dir);

}

}

void bitonicSort(int arr[], int low, int cnt, int dir) {

if (cnt > 1) {

int k = cnt / 2;

bitonicSort(arr, low, k, 1); // Sort in ascending order

bitonicSort(arr, low + k, k, 0); // Sort in descending order

bitonicMerge(arr, low, cnt, dir);

}

}

int main() {

int arr[] = {12, 9, 8, 7, 6, 5};

int n = sizeof(arr) / sizeof(arr[0]);

bitonicSort(arr, 0, n, 1);

printf("Sorted array: ");

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

printf("%d ", arr[i]);

return 0;

}

15)

#include <stdio.h>

#include <stdlib.h>

#define ALPHABET\_SIZE 26

struct TrieNode {

struct TrieNode\* children[ALPHABET\_SIZE];

int isEndOfWord;

};

struct TrieNode\* getNode() {

struct TrieNode\* node = (struct TrieNode\*)malloc(sizeof(struct TrieNode));

node->isEndOfWord = 0;

for (int i = 0; i < ALPHABET\_SIZE; i++)

node->children[i] = NULL;

return node;

}

void insert(struct TrieNode\* root, const char\* key) {

struct TrieNode\* pCrawl = root;

for (int level = 0; key[level]; level++) {

int index = key[level] - 'a';

if (!pCrawl->children[index])

pCrawl->children[index] = getNode();

pCrawl = pCrawl->children[index];

}

pCrawl->isEndOfWord = 1;

}

int search(struct TrieNode\* root, const char\* key) {

struct TrieNode\* pCrawl = root;

for (int level = 0; key[level]; level++) {

int index = key[level] - 'a';

if (!pCrawl->children[index])

return 0;

pCrawl = pCrawl->children[index];

}

return (pCrawl != NULL && pCrawl->isEndOfWord);

}

int main() {

struct TrieNode\* root = getNode();

insert(root, "hello");

insert(root, "world");

printf("Searching for 'hello': %s\n", search(root, "hello") ? "Found" : "Not Found");

printf("Searching for 'world': %s\n", search(root, "world") ? "Found" : "Not Found");

printf("Searching for 'trie': %s\n", search(root, "trie") ? "Found" : "Not Found");

return 0;

}

16)

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

struct Graph {

int numVertices;

struct Node\*\* adjLists;

int\* visited;

};

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

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 dfs(struct Graph\* graph, int vertex) {

graph->visited[vertex] = 1;

printf("%d ", vertex);

struct Node\* temp = graph->adjLists[vertex];

while (temp) {

int connectedVertex = temp->data;

if (!graph->visited[connectedVertex]) {

dfs(graph, connectedVertex);

}

temp = temp->next;

}

}

int main() {

struct Graph\* graph = createGraph(5);

addEdge(graph, 0, 1);

addEdge(graph, 0, 4);

addEdge(graph, 1, 2);

addEdge(graph, 1, 3);

addEdge(graph, 1, 4);

addEdge(graph, 2, 3);

addEdge(graph, 3, 4);

printf("Depth First Search starting from vertex 0:\n");

dfs(graph, 0);

return 0;

}

17)

#include <stdio.h>

#include <stdlib.h>

struct Node {

int data;

struct Node\* next;

};

struct Graph {

int numVertices;

struct Node\*\* adjLists;

int\* visited;

};

struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = data;

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 bfs(struct Graph\* graph, int startVertex) {

int queue[100], front = -1, rear = -1;

graph->visited[startVertex] = 1;

queue[++rear] = startVertex;

while (front < rear) {

int currentVertex = queue[++front];

printf("%d ", currentVertex);

struct Node\* temp = graph->adjLists[currentVertex];

while (temp) {

int connectedVertex = temp->data;

if (!graph->visited[connectedVertex]) {

graph->visited[connectedVertex] = 1;

queue[++rear] = connectedVertex;

}

temp = temp->next;

}

}

}

int main() {

struct Graph\* graph = createGraph(5);

addEdge(graph, 0, 1);

addEdge(graph, 0, 4);

addEdge(graph, 1, 2);

addEdge(graph, 1, 3);

addEdge(graph, 1, 4);

addEdge(graph, 2, 3);

addEdge(graph, 3, 4);

printf("Breadth First Search starting from vertex 0:\n");

bfs(graph, 0);

return 0;

}

18)

#include <stdio.h>

#include <stdbool.h>

#include <limits.h>

#define V 5

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

int min = INT\_MAX, min\_index;

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

if (!mstSet[v] && key[v] < min) {

min = key[v];

min\_index = v;

}

}

return min\_index;

}

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

int parent[V];

int key[V];

bool mstSet[V];

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

key[i] = INT\_MAX;

mstSet[i] = false;

}

key[0] = 0;

parent[0] = -1;

for (int count = 0; count < V - 1; count++) {

int u = minKey(key, mstSet);

mstSet[u] = true;

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

if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {

parent[v] = u;

key[v] = graph[u][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]]);

}

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;

}