1. Write a C program to implement hashing using Linear Probing method

struct HashTable {

struct HashEntry \*table;

int size;

};

struct HashTable \*initHashTable(int size) {

struct HashTable \*ht = (struct HashTable \*)malloc(sizeof(struct HashTable));

ht->size = size;

ht->table = (struct HashEntry \*)malloc(size \* sizeof(struct HashEntry));

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

ht->table[i].key = -1; // Key of -1 indicates empty slot

}

return ht;

}

int hashFunction(int key, int size) {

return key % size;

}

void insert(struct HashTable \*ht, int key, int data) {

int index = hashFunction(key, ht->size);

while (ht->table[index].key != -1) {

index = (index + 1) % ht->size;

}

ht->table[index].key = key;

ht->table[index].data = data;

}

int search(struct HashTable \*ht, int key) {

int index = hashFunction(key, ht->size);

int originalIndex = index;

while (ht->table[index].key != key) {

index = (index + 1) % ht->size;

if (index == originalIndex)

return -1;

}

return ht->table[index].data;

}

void display(struct HashTable \*ht) {

printf("\nHash Table:\n");

printf("-----------\n");

printf("Index\tKey\tData\n");

for (int i = 0; i < ht->size; ++i) {

printf("%d\t%d\t%d\n", i, ht->table[i].key, ht->table[i].data);

}

printf("\n");

}

int main() {

struct HashTable \*ht = initHashTable(SIZE);

insert(ht, 12, 123);

insert(ht, 25, 456);

insert(ht, 10, 789);

insert(ht, 4, 321);

display(ht);

int key\_to\_search = 25;

int result = search(ht, key\_to\_search);

if (result != -1) {

printf("Key %d found with value: %d\n", key\_to\_search, result);

} else {

printf("Key %d not found in hash table\n", key\_to\_search);

}

free(ht->table);

free(ht);

return 0;

}

1. Write a C program to arrange a series of numbers using Insertion . without

#include <stdio.h>

#define MAX\_SIZE 100

int main() {

int arr[MAX\_SIZE];

int n, i, j, temp;

printf("Enter the number of elements: ");

scanf("%d", &n);

printf("Enter %d elements:\n", n);

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

scanf("%d", &arr[i]);

}

for (i = 1; i < n; i++) {

temp = arr[i];

j = i - 1;

while (j >= 0 && arr[j] > temp) {

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

j--;

}

arr[j + 1] = temp;

}

printf("\nSorted array in ascending order:\n");

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

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

}

printf("\n");

return 0;

}

1. Write a C program to arrange a series of numbers using Merge Sort

#include <stdio.h>

#define MAX\_SIZE 100

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[MAX\_SIZE];

int n, i;

printf("Enter the number of elements: ");

scanf("%d", &n);

printf("Enter %d elements:\n", n);

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

scanf("%d", &arr[i]);

}

mergeSort(arr, 0, n - 1);

printf("\nSorted array in ascending order:\n");

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

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

}printf("\n");

return 0;

}

1. Write a C program to arrange a series of numbers using Quick Sort

#include <stdio.h>

#define MAX\_SIZE 100

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 - 1; 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[MAX\_SIZE];

int n, i;

printf("Enter the number of elements: ");

scanf("%d", &n);

printf("Enter %d elements:\n", n);

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

scanf("%d", &arr[i]);

}

quickSort(arr, 0, n - 1);

printf("\nSorted array in ascending order:\n");

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

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

}

printf("\n");

return 0;

}

1. Write a C program to implement Heap sort.

#include <stdio.h>

void heapify(int arr[], int n, int i);

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

void printArray(int arr[], int n);

void heapify(int arr[], int n, int i) {

int largest = i;

int l = 2 \* i + 1;

int r = 2 \* i + 2;

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

largest = l;

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

largest = r;

if (largest != i) {

int temp = arr[i];

arr[i] = arr[largest];

arr[largest] = temp;

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--) {

int temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

heapify(arr, i, 0);

}

}

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

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

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

printf("\n");}

int main() {

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

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

printf("Original array: \n");

printArray(arr, n);

heapSort(arr, n);

printf("Sorted array: \n");

printArray(arr, n);

}

1. Write a C program to perform the following operations:

a) Insert an element into a AVL tree.

b) Delete an element from a AVL tree.

c) Search for a key element in a AVL tree.

int height(struct Node \*N) {

if (N == NULL)

return 0;

return N->height;}

int max(int a, int b) {

return (a > b) ? a : b;

}

struct Node \*newNode(int key) {

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

node->key = key;

node->left = NULL;

node->right = NULL;

node->height = 1;

return (node);

}

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

struct Node \*x = y->left;

struct Node \*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;

}

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

struct Node \*y = x->right;

struct Node \*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 Node \*N) {

if (N == NULL)

return 0;

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

}

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

if (node == NULL)

return newNode(key);

if (key < node->key)

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

else if (key > node->key)

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

else

return node;

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

int balance = getBalance(node);

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

return rightRotate(node);

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

return leftRotate(node);

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

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

return rightRotate(node);

}

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

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

return leftRotate(node);

}

return node;

}

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

struct Node \*current = node;

while (current->left != NULL)

current = current->left;

return current;

}

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

if (root == NULL)

return root;

if (key < root->key)

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

else if (key > root->key)

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

else {

if ((root->left == NULL) || (root->right == NULL)) {

struct Node \*temp = root->left ? root->left : root->right;

if (temp == NULL) {

temp = root;

root = NULL;

} else

\*root = \*temp;

free(temp);

} else {

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

root->key = temp->key;

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

}

}

if (root == NULL)

return root;

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

int balance = getBalance(root);

if (balance > 1 && getBalance(root->left) >= 0)

return rightRotate(root);

if (balance > 1 && getBalance(root->left) < 0) {

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

return rightRotate(root);

}

if (balance < -1 && getBalance(root->right) <= 0)

return leftRotate(root);

if (balance < -1 && getBalance(root->right) > 0) {

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

return leftRotate(root);

}

return root;

}

struct Node \*search(struct Node \*root, int key) {

if (root == NULL || root->key == key)

return root;

if (key < root->key)

return search(root->left, key);

return search(root->right, key);

}

void inorder(struct Node \*root) {

if (root != NULL) {

inorder(root->left);

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

inorder(root->right);

}

}

int main() {

struct Node \*root = NULL;

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 30);

root = insert(root, 40);

root = insert(root, 50);

root = insert(root, 25);

printf("Inorder traversal of the AVL tree after insertion:\n");

inorder(root);

printf("\n");

root = deleteNode(root, 30);

printf("Inorder traversal of the AVL tree after deletion:\n");

inorder(root);

printf("\n");

int key = 40;

struct Node \*result = search(root, key);

if (result != NULL)

printf("%d found in the AVL tree.\n", key);

else

printf("%d not found in the AVL tree.\n", key);

return 0;

}

1. Write a C program for Graph traversal using Breadth First Search

struct AdjList {

struct AdjListNode\* head;

};

struct Graph {

int V;

struct AdjList\* array;

};

struct AdjListNode\* newAdjListNode(int dest) {

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

newNode->dest = dest;

newNode->next = NULL;

return newNode;

}

struct Graph\* createGraph(int V) {

struct Graph\* graph = (struct Graph\*) malloc(sizeof(struct Graph));

graph->V = V;

graph->array = (struct AdjList\*) malloc(V \* sizeof(struct AdjList));

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

graph->array[i].head = NULL;

return graph;

}

void addEdge(struct Graph\* graph, int src, int dest) {

struct AdjListNode\* newNode = newAdjListNode(dest);

newNode->next = graph->array[src].head;

graph->array[src].head = newNode;

newNode = newAdjListNode(src);

newNode->next = graph->array[dest].head;

graph->array[dest].head = newNode;

}

void BFS(struct Graph\* graph, int startVertex) {

int visited[MAX\_SIZE] = {0};

int queue[MAX\_SIZE];

int front = 0, rear = 0;

visited[startVertex] = 1;

queue[rear++] = startVertex;

while (front < rear) {

int current = queue[front++];

printf("%d ", current);

struct AdjListNode\* temp = graph->array[current].head;

while (temp != NULL) {

int adjVertex = temp->dest;

if (!visited[adjVertex]) {

visited[adjVertex] = 1;

queue[rear++] = adjVertex;

}

temp = temp->next;

}

}

}

int main() {

int V = 5;

struct Graph\* graph = createGraph(V);

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);

BFS(graph, 0);

printf("\n");

return 0;

}

23 .Write a C program for Graph traversal using Depth First Search

struct AdjList {

struct AdjListNode\* head;

};

struct Graph {

int V;

struct AdjList\* array;

};

struct AdjListNode\* newAdjListNode(int dest) {

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

newNode->dest = dest;

newNode->next = NULL;

return newNode;

}

struct Graph\* createGraph(int V) {

struct Graph\* graph = (struct Graph\*) malloc(sizeof(struct Graph));

graph->V = V;

graph->array = (struct AdjList\*) malloc(V \* sizeof(struct AdjList));

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

graph->array[i].head = NULL;

return graph;

}

void addEdge(struct Graph\* graph, int src, int dest) {

struct AdjListNode\* newNode = newAdjListNode(dest);

newNode->next = graph->array[src].head;

graph->array[src].head = newNode;

newNode = newAdjListNode(src);

newNode->next = graph->array[dest].head;

graph->array[dest].head = newNode;

}

void BFS(struct Graph\* graph, int startVertex) {

int visited[MAX\_SIZE] = {0};

int queue[MAX\_SIZE];

int front = 0, rear = 0;

visited[startVertex] = 1;

queue[rear++] = startVertex;

while (front < rear) {

int current = queue[front++];

printf("%d ", current);

struct AdjListNode\* temp = graph->array[current].head;

while (temp != NULL) {

int adjVertex = temp->dest;

if (!visited[adjVertex]) {

visited[adjVertex] = 1;

queue[rear++] = adjVertex;

}

temp = temp->next;

}

}

}

int main() {

int V = 5;

struct Graph\* graph = createGraph(V);

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);

BFS(graph, 0);

printf("\n");

return 0;

}

1. 24. Implementation of Shortest Path Algorithms using Dijkstra’s Algorithm

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];

bool sptSet[V];

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

dist[i] = INT\_MAX;

sptSet[i] = false;

}

dist[src] = 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, 4, 0, 0, 0, 0},

{4, 0, 8, 0, 0, 0},

{0, 8, 0, 7, 0, 4},

{0, 0, 7, 0, 9, 14},

{0, 0, 0, 9, 0, 10},

{0, 0, 4, 14, 10, 0}

};

dijkstra(graph, 0);

return 0;

}

25.Implementation of Minimum Spanning Tree using Prim’s Algorithm.

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;

}

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];

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] == false && 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;

}

26.Implementation of Minimum Spanning Tree using Kruskal Algorithm

struct Edge {

int src, dest, weight;

};

struct Graph {

int V, E;

struct Edge edges[MAX\_EDGES];

};

struct Subset {

int parent;

int rank;

};

struct Graph\* createGraph(int V, int E) {

struct Graph\* graph = (struct Graph\*) malloc(sizeof(struct Graph));

graph->V = V;

graph->E = E;

return graph;

}

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->edges, graph->E, sizeof(graph->edges[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->edges[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("Edge \tWeight\n");

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

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

free(subsets);

}

int main() {

int V = 4; // Number of vertices in the graph

int E = 5; // Number of edges in the graph

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

// Edge 0-1

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

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

graph->edges[0].weight = 10;

// Edge 0-2

graph->edges[1].src = 0;

graph->edges[1].dest = 2;

graph->edges[1].weight = 6;

// Edge 0-3

graph->edges[2].src = 0;

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

graph->edges[2].weight = 5;

// Edge 1-3

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

graph->edges[3].dest = 3;

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

// Edge 2-3

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

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

graph->edges[4].weight = 4;

KruskalMST(graph);

return 0;

}