**Assignment: 8**

Problem Statement: Store data of students using hashing function for roll number and implement **linear probing using chaining without replacement** and chaining with replacement algorithm.

#include <stdio.h>

#include <stdlib.h> // For malloc and free

#include <string.h> // For strcpy

struct Student {

int rollNumber;

int chain; // Index of the next entry in the chain

struct Student\* next;

};

struct Student\* createStudent(int rollNumber) {

struct Student\* newStudent = (struct Student\*)malloc(sizeof(struct Student));

if (newStudent == NULL) {

printf("Memory allocation failed\n");

exit(1);

}

newStudent->rollNumber = rollNumber;

newStudent->chain = -1; // Initialize chain to -1 (no chaining)

newStudent->next = NULL;

return newStudent;

}

#define TABLE\_SIZE 10

struct HashTableWithoutReplacement {

struct Student\* table[TABLE\_SIZE];

};

void initializeHashTableWithoutReplacement(struct HashTableWithoutReplacement\* hashTable) {

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

hashTable->table[i] = NULL;

}

}

void insertWithoutReplacement(struct HashTableWithoutReplacement\* hashTable, int rollNumber) {

int index = rollNumber % TABLE\_SIZE;

if (hashTable->table[index] == NULL) {

hashTable->table[index] = createStudent(rollNumber);

} else {

// Find the next empty slot

int nextIndex = (index + 1) % TABLE\_SIZE;

while (hashTable->table[nextIndex] != NULL) {

nextIndex = (nextIndex + 1) % TABLE\_SIZE; // Linear probing

}

hashTable->table[nextIndex] = createStudent(rollNumber);

hashTable->table[index]->chain = nextIndex;

}

}

void displayWithoutReplacement(struct HashTableWithoutReplacement\* hashTable) {

printf("Hash Table (Chaining Without Replacement):\n");

printf("Index\tRoll Numbers\tChain\n");

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

printf("%d\t", i);

if (hashTable->table[i] == NULL) {

printf("Empty\t\t-1\n");

} else {

struct Student\* current = hashTable->table[i];

while (current != NULL) {

printf("%d ", current->rollNumber);

current = current->next;

}

printf("\t\t");

if (hashTable->table[i]->chain == -1) {

printf("-1");

} else {

printf("%d", hashTable->table[i]->chain);

}

printf("\n");

}

}

}

struct HashTableWithReplacement {

struct Student\* table[TABLE\_SIZE];

};

void initializeHashTableWithReplacement(struct HashTableWithReplacement\* hashTable) {

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

hashTable->table[i] = NULL;

}

}

void insertWithReplacement(struct HashTableWithReplacement\* hashTable, int rollNumber) {

int index = rollNumber % TABLE\_SIZE;

if (hashTable->table[index] == NULL) {

hashTable->table[index] = createStudent(rollNumber);

} else {

// Check if the existing key rightly belongs to the slot

if (hashTable->table[index]->rollNumber % TABLE\_SIZE == index) {

// Move the existing key to the next available slot

int nextIndex = (index + 1) % TABLE\_SIZE;

while (hashTable->table[nextIndex] != NULL) {

nextIndex = (nextIndex + 1) % TABLE\_SIZE; // Linear probing

}

hashTable->table[nextIndex] = hashTable->table[index];

hashTable->table[index] = createStudent(rollNumber);

hashTable->table[index]->chain = nextIndex;

} else {

// Find the next empty slot

int nextIndex = (index + 1) % TABLE\_SIZE;

while (hashTable->table[nextIndex] != NULL) {

nextIndex = (nextIndex + 1) % TABLE\_SIZE; // Linear probing

}

// Move the existing key to the next available slot

int currentChain = index;

while (hashTable->table[currentChain]->next != NULL) {

currentChain = hashTable->table[currentChain]->chain;

}

hashTable->table[currentChain]->next = hashTable->table[nextIndex];

hashTable->table[index] = createStudent(rollNumber);

hashTable->table[index]->chain = nextIndex;

}

}

}

void displayWithReplacement(struct HashTableWithReplacement\* hashTable) {

printf("Hash Table (Chaining With Replacement):\n");

printf("Index\tRoll Numbers\tChain\n");

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

printf("%d\t", i);

if (hashTable->table[i] == NULL) {

printf("Empty\t\t-1\n");

} else {

struct Student\* current = hashTable->table[i];

while (current != NULL) {

printf("%d ", current->rollNumber);

current = current->next;

}

printf("\t\t");

if (hashTable->table[i]->chain == -1) {

printf("-1");

} else {

printf("%d", hashTable->table[i]->chain);

}

printf("\n");

}

}

}

int main() {

int choice, numRollNumbers, rollNumber;

printf("Choose the collision resolution method:\n");

printf("1. Chaining without replacement\n");

printf("2. Chaining with replacement\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1: {

struct HashTableWithoutReplacement hashTableWithoutReplacement;

initializeHashTableWithoutReplacement(&hashTableWithoutReplacement);

printf("How many roll numbers do you want to insert? ");

scanf("%d", &numRollNumbers);

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

printf("Enter roll number %d: ", i + 1);

scanf("%d", &rollNumber);

insertWithoutReplacement(&hashTableWithoutReplacement, rollNumber);

}

displayWithoutReplacement(&hashTableWithoutReplacement);

break;

}

case 2: {

struct HashTableWithReplacement hashTableWithReplacement;

initializeHashTableWithReplacement(&hashTableWithReplacement);

printf("How many roll numbers do you want to insert? ");

scanf("%d", &numRollNumbers);

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

printf("Enter roll number %d: ", i + 1);

scanf("%d", &rollNumber);

insertWithReplacement(&hashTableWithReplacement, rollNumber);

}

displayWithReplacement(&hashTableWithReplacement);

break;

}

default:

printf("Invalid choice. Exiting program.\n");

return 1;

}

return 0;

}

**Assignment: 1**

**Problem Statement:** Construct an **expression tree from infix/postfix expression and perform recursive and non-recursive inorder, preorder** and post order traversals.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

struct TreeNode {

char data;

struct TreeNode\* left;

struct TreeNode\* right;

};

struct TreeNode\* createNode(char val) {

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

if (newNode == NULL) {

printf("Memory allocation failed\n");

exit(1);

}

newNode->data = val;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

int isOperand(char c) {

return (c >= '0' && c <= '9') || (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z');

}

struct TreeNode\* constructExpressionTreePostfix(char\* postfix) {

struct TreeNode\* stack[100]; // Assuming expression won't exceed 100 characters

int top = -1;

for (int i = 0; postfix[i]; ++i) {

char c = postfix[i];

if (isOperand(c)) {

struct TreeNode\* newNode = createNode(c);

stack[++top] = newNode;

} else {

struct TreeNode\* operand2 = stack[top--];

struct TreeNode\* operand1 = stack[top--];

struct TreeNode\* newNode = createNode(c);

newNode->left = operand1;

newNode->right = operand2;

stack[++top] = newNode;

}

}

return stack[top];

}

struct TreeNode\* constructExpressionTreePrefix(char\* prefix) {

struct TreeNode\* stack[100]; // Assuming expression won't exceed 100 characters

int top = -1;

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

char c = prefix[i];

if (isOperand(c)) {

struct TreeNode\* newNode = createNode(c);

stack[++top] = newNode;

} else {

struct TreeNode\* operand1 = stack[top--];

struct TreeNode\* operand2 = stack[top--];

struct TreeNode\* newNode = createNode(c);

newNode->left = operand1;

newNode->right = operand2;

stack[++top] = newNode;

}

}

return stack[top];

}

void recursiveInorder(struct TreeNode\* root) {

if (root) {

recursiveInorder(root->left);

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

recursiveInorder(root->right);

}

}

void recursivePreorder(struct TreeNode\* root) {

if (root) {

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

recursivePreorder(root->left);

recursivePreorder(root->right);

}

}

void recursivePostorder(struct TreeNode\* root) {

if (root) {

recursivePostorder(root->left);

recursivePostorder(root->right);

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

}

}

void nonRecursiveInorder(struct TreeNode\* root) {

struct TreeNode\* stack[100]; // Assuming expression won't exceed 100 characters

int top = -1;

while (root || top != -1) {

while (root) {

stack[++top] = root;

root = root->left;

}

root = stack[top--];

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

root = root->right;

}

}

void nonRecursivePreorder(struct TreeNode\* root) {

struct TreeNode\* stack[100]; // Assuming expression won't exceed 100 characters

int top = -1;

stack[++top] = root;

while (top != -1) {

root = stack[top--];

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

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

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

}

}

void nonRecursivePostorder(struct TreeNode\* root) {

struct TreeNode\* stack1[100]; // For storing nodes

struct TreeNode\* stack2[100]; // For reverse postorder traversal

int top1 = -1, top2 = -1;

stack1[++top1] = root;

while (top1 != -1) {

root = stack1[top1--];

stack2[++top2] = root;

if (root->left) stack1[++top1] = root->left;

if (root->right) stack1[++top1] = root->right;

}

while (top2 != -1) {

printf("%c ", stack2[top2--]->data);

}

}

void displayLevelWise(struct TreeNode\* root) {

if (!root) return;

struct TreeNode\* queue[100]; // Assuming expression won't exceed 100 characters

int front = 0, rear = 0;

queue[rear++] = root;

while (front < rear) {

int size = rear - front;

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

struct TreeNode\* current = queue[front++];

if (current) {

printf("%c ", current->data);

if (current->left) queue[rear++] = current->left;

else queue[rear++] = NULL;

if (current->right) queue[rear++] = current->right;

else queue[rear++] = NULL;

} else {

printf("\_ ");

}

}

printf("\n");

}

}

int main() {

char postfixExpression[100];

char prefixExpression[100];

printf("Enter postfix expression: ");

scanf("%s", postfixExpression);

printf("Enter prefix expression: ");

scanf("%s", prefixExpression);

// Construct expression tree from postfix expression

struct TreeNode\* postfixRoot = constructExpressionTreePostfix(postfixExpression);

// Construct expression tree from prefix expression

struct TreeNode\* prefixRoot = constructExpressionTreePrefix(prefixExpression);

// Display level-wise expression trees

printf("Level-wise Postfix Expression Tree:\n");

displayLevelWise(postfixRoot);

printf("Level-wise Prefix Expression Tree:\n");

displayLevelWise(prefixRoot);

// Recursive traversals

printf("\nRecursive Inorder (Postfix): ");

recursiveInorder(postfixRoot);

printf("\nRecursive Preorder (Postfix): ");

recursivePreorder(postfixRoot);

printf("\nRecursive Postorder (Postfix): ");

recursivePostorder(postfixRoot);

printf("\n");

printf("Recursive Inorder (Prefix): ");

recursiveInorder(prefixRoot);

printf("\nRecursive Preorder (Prefix): ");

recursivePreorder(prefixRoot);

printf("\nRecursive Postorder (Prefix): ");

recursivePostorder(prefixRoot);

printf("\n");

// Non-recursive traversals

printf("\nNon-Recursive Inorder (Postfix): ");

nonRecursiveInorder(postfixRoot);

printf("\nNon-Recursive Preorder (Postfix): ");

nonRecursivePreorder(postfixRoot);

printf("\nNon-Recursive Postorder (Postfix): ");

nonRecursivePostorder(postfixRoot);

printf("\n");

printf("Non-Recursive Inorder (Prefix): ");

nonRecursiveInorder(prefixRoot);

printf("\nNon-Recursive Preorder (Prefix): ");

nonRecursivePreorder(prefixRoot);

printf("\nNon-Recursive Postorder (Prefix): ");

nonRecursivePostorder(prefixRoot);

printf("\n");

return 0;

}

**Assignment: 2**

**Problem Statement**: **Create a binary search tree (BST**) and perform the following operations

(i.) Insert (ii.) Delete (iii.) Display level of tree (iv.) Display height of tree (v.) mirror of tree.

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <limits.h>

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

struct Node\* createNode(int data) {

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

if (newNode == NULL) {

printf("Memory allocation failed\n");

exit(1);

}

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

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

if (root == NULL) {

return createNode(data);

}

if (data < root->data) {

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

} else {

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

}

return root;

}

void inorder(struct Node\* node) {

if (node != NULL) {

inorder(node->left);

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

inorder(node->right);

}

}

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

if (root == NULL || root->data == data) {

return root;

}

if (data < root->data) {

return search(root->left, data);

} else {

return search(root->right, data);

}

}

int max(int a, int b) {

return (a > b) ? a : b;

}

int getHeight(struct Node\* node) {

if (node == NULL) {

return 0;

}

int leftHeight = getHeight(node->left);

int rightHeight = getHeight(node->right);

return 1 + max(leftHeight, rightHeight);

}

void levelOrder(struct Node\* root) {

if (root == NULL) {

return;

}

struct Node\* queue[1000]; // Assuming the tree won't have more than 1000 nodes

int front = 0, rear = 0;

queue[rear++] = root;

while (front < rear) {

int levelSize = rear - front;

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

struct Node\* node = queue[front++];

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

if (node->left) {

queue[rear++] = node->left;

}

if (node->right) {

queue[rear++] = node->right;

}

}

printf("\n");

}

}

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 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 mirror(struct Node\* node) {

if (node == NULL) {

return;

}

struct Node\* temp = node->left;

node->left = node->right;

node->right = temp;

mirror(node->left);

mirror(node->right);

}

int main() {

struct Node\* root = NULL;

root = insert(root, 60);

insert(root, 40);

insert(root, 30);

insert(root, 50);

insert(root, 80);

insert(root, 70);

insert(root, 90);

printf("Inorder traversal:\n");

inorder(root);

printf("\n");

int searchData = 40;

printf("Search for %d:\n", searchData);

struct Node\* searchedNode = search(root, searchData);

if (searchedNode) {

printf("Found: %d\n", searchedNode->data);

} else {

printf("Not found\n");

}

printf("Height of the tree: %d\n", getHeight(root));

printf("Level-order traversal:\n");

levelOrder(root);

int deleteData = 50;

printf("Deleting %d:\n", deleteData);

root = deleteNode(root, deleteData);

inorder(root);

printf("\n");

printf("Mirroring the tree:\n");

mirror(root);

inorder(root);

printf("\n");

return 0;

}

**Assignment: 3**

**Problem Statement:** Construct an **inorder threaded binary search tree. Traverse threaded** binary tree in inorder and preorder.

#include<stdio.h>

#include<stdlib.h>

#include<stdbool.h>

struct Node

{

struct Node \*left, \*right;

int info;

bool lthread;

bool rthread;

};

struct Node \*insert(struct Node \*root, int ikey)

{

struct Node \*ptr = root;

struct Node \*par = NULL;

while (ptr != NULL)

{

if (ikey == (ptr->info))

{

printf("Duplicate Key !\n");

return root;

}

par = ptr;

if (ikey < ptr->info)

{

if (ptr -> lthread == false)

ptr = ptr -> left;

else

break;

}

else

{

if (ptr->rthread == false)

ptr = ptr -> right;

else

break;

}

}

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

tmp -> info = ikey;

tmp -> lthread = true;

tmp -> rthread = true;

if (par == NULL)

{

root = tmp;

tmp -> left = NULL;

tmp -> right = NULL;

}

else if (ikey < (par -> info))

{

tmp -> left = par -> left;

tmp -> right = par;

par -> lthread = false;

par -> left = tmp;

}

else

{

tmp -> left = par;

tmp -> right = par -> right;

par -> rthread = false;

par -> right = tmp;

}

return root;

}

struct Node \*inorderSuccessor(struct Node \*ptr)

{

if (ptr -> rthread == true)

return ptr->right;

ptr = ptr -> right;

while (ptr -> lthread == false)

ptr = ptr -> left;

return ptr;

}

void inorder(struct Node \*root)

{

if (root == NULL)

printf("Tree is empty");

struct Node \*ptr = root;

while (ptr -> lthread == false)

ptr = ptr -> left;

while (ptr != NULL)

{

printf("%d ",ptr -> info);

ptr = inorderSuccessor(ptr);

}

}

void preorder(struct Node \*root)

{

if (root == NULL)

{

printf("Tree is empty\n");

return;

}

struct Node \*ptr = root;

while (ptr != NULL)

{

printf("%d ", ptr->info);

if (ptr->lthread == false)

{

ptr = ptr->left;

}

else if (ptr->rthread == false)

{

ptr = ptr->right;

}

else

{

while (ptr != NULL && ptr->rthread == true)

{

ptr = inorderSuccessor(ptr);

}

if (ptr != NULL)

{

ptr = ptr->right;

}

}

}

}

void levelOrderTraversal(struct Node \*root) {

if (root == NULL) return;

struct Node\* queue[1000]; // Assuming the tree won't have more than 1000 nodes

int front = 0, rear = 0;

queue[rear++] = root;

queue[rear++] = NULL; // Add a marker for the end of each level

while (front < rear) {

struct Node \*current = queue[front++];

// Check if current node is a level marker

if (current == NULL) {

printf("\n");

if (front < rear) queue[rear++] = NULL; // If not the last level, add a marker for the next level

} else {

printf("%d ", current->info);

// If left child exists and is not a thread, push it to the queue

if (current->left && !current->lthread)

queue[rear++] = current->left;

// If right child exists and is not a thread, push it to the queue

if (current->right && !current->rthread)

queue[rear++] = current->right;

}

}

}

int main()

{

struct Node \*root = NULL;

root = insert(root, 20);

root = insert(root, 10);

root = insert(root, 30);

root = insert(root, 5);

root = insert(root, 16);

root = insert(root, 14);

root = insert(root, 17);

root = insert(root, 13);

printf("Inorder Traversal: ");

inorder(root);

printf("\n");

printf("Preorder Traversal: ");

preorder(root);

printf("\n");

printf("Level wise display:\n");

levelOrderTraversal(root);

return 0;

}

**Assignment: 5**

Use the map of the area around the college as the graph. Identify the prominent land marks as nodes and find minimum distance to various land marks from the college as the source. Represent this graph using adjacency matrix. Find the shortest path using **Dijkstra’s algorithm.**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<limits.h>

#define MAX\_LANDMARKS 6 // Maximum number of landmarks

// Structure for the Dijkstra Graph

struct DijkstraGraph {

char landmarks[MAX\_LANDMARKS][2];

int V;

int graph[MAX\_LANDMARKS][MAX\_LANDMARKS];

};

// Function prototypes

void addEdge(struct DijkstraGraph\* graph, char landmark1[2], char landmark2[2], int distance);

int getIndex(struct DijkstraGraph\* graph, char landmark[2]);

void dijkstra(struct DijkstraGraph\* graph, char source[2]);

void printSolution(int dist[], char landmarks[MAX\_LANDMARKS][2], int V);

void printAdjacencyMatrix(struct DijkstraGraph\* graph);

int main() {

// Landmarks and distances

char landmarks[MAX\_LANDMARKS][2] = {"A", "B", "C", "D", "E", "F"};

struct DijkstraGraph graph;

graph.V = MAX\_LANDMARKS;

memcpy(graph.landmarks, landmarks, sizeof(landmarks));

// Initialize graph with 0 distance for all landmarks

for (int i = 0; i < MAX\_LANDMARKS; ++i) {

for (int j = 0; j < MAX\_LANDMARKS; ++j) {

graph.graph[i][j] = 0;

}

}

// Adding edges and distances between landmarks

addEdge(&graph, "A", "B", 7);

addEdge(&graph, "A", "C", 9);

addEdge(&graph, "A", "D", 14);

addEdge(&graph, "B", "C", 10);

addEdge(&graph, "B", "E", 15);

addEdge(&graph, "C", "D", 2);

addEdge(&graph, "C", "F", 11);

addEdge(&graph, "D", "E", 9);

addEdge(&graph, "E", "F", 6);

// Print the adjacency matrix

printAdjacencyMatrix(&graph);

// Find the shortest path distances from the college (A)

dijkstra(&graph, "A");

return 0;

}

// Function to add an edge between two landmarks with a given distance

void addEdge(struct DijkstraGraph\* graph, char landmark1[2], char landmark2[2], int distance) {

int index1 = getIndex(graph, landmark1);

int index2 = getIndex(graph, landmark2);

if (index1 != -1 && index2 != -1) {

graph->graph[index1][index2] = distance;

graph->graph[index2][index1] = distance; // Assuming undirected graph

}

}

// Function to get the index of a landmark in the graph

int getIndex(struct DijkstraGraph\* graph, char landmark[2]) {

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

if (strcmp(graph->landmarks[i], landmark) == 0)

return i;

}

return -1;

}

// Function to perform Dijkstra's algorithm

void dijkstra(struct DijkstraGraph\* graph, char source[2]) {

int src = getIndex(graph, source);

if (src == -1) {

printf("Source landmark not found!\n");

return;

}

int dist[MAX\_LANDMARKS];

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

dist[i] = INT\_MAX;

}

dist[src] = 0;

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

int u, v;

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

int minDist = INT\_MAX;

for (v = 0; v < graph->V; ++v) {

if (!visited[v] && dist[v] <= minDist) {

minDist = dist[v];

u = v;

}

}

visited[u] = 1;

for (v = 0; v < graph->V; ++v) {

if (!visited[v] && graph->graph[u][v] && dist[u] != INT\_MAX &&

dist[u] + graph->graph[u][v] < dist[v]) {

dist[v] = dist[u] + graph->graph[u][v];

}

}

}

printSolution(dist, graph->landmarks, graph->V);

}

// Function to print the shortest path distances

void printSolution(int dist[], char landmarks[MAX\_LANDMARKS][2], int V) {

printf("Shortest path distances from the college (A):\n");

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

printf("To %s: %d\n", landmarks[i], dist[i]);

}

}

// Function to print the adjacency matrix

void printAdjacencyMatrix(struct DijkstraGraph\* graph) {

printf("Adjacency Matrix representation of the graph:\n");

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

for (int j = 0; j < graph->V; ++j) {

printf("%d ", graph->graph[i][j]);

}

printf("\n");

}

}

**Assignment: 4**

# Create a graph using adjacency list representation. Perform graph traversal using using BFS and DFS

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Structure for a node in the adjacency list

struct Node {

int data;

struct Node\* next;

};

// Structure for the graph

struct Graph {

int V; // Number of vertices

struct Node\*\* adj; // Pointer to an array containing adjacency lists

};

// Function prototypes

struct Node\* createNode(int data);

struct Graph\* createGraph(int V);

void addEdge(struct Graph\* graph, int v, int w);

void BFS(struct Graph\* graph, int s);

void DFS(struct Graph\* graph, int s);

// Function to create a new node

struct Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

// Function to create a graph with V vertices

struct Graph\* createGraph(int V) {

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

graph->V = V;

graph->adj = (struct Node\*\*)malloc(V \* sizeof(struct Node\*));

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

graph->adj[i] = NULL;

return graph;

}

// Function to add an edge to the graph

void addEdge(struct Graph\* graph, int v, int w) {

// Add w to v’s list

struct Node\* newNode = createNode(w);

newNode->next = graph->adj[v];

graph->adj[v] = newNode;

}

// Function to perform Breadth First Search

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

bool\* visited = (bool\*)malloc(graph->V \* sizeof(bool));

for (int i = 0; i < graph->V; ++i)

visited[i] = false;

// Create a queue for BFS

int\* queue = (int\*)malloc(graph->V \* sizeof(int));

int front = -1, rear = -1;

// Mark the current node as visited and enqueue it

visited[s] = true;

queue[++rear] = s;

while (front != rear) {

// Dequeue a vertex from queue and print it

s = queue[++front];

printf("%d ", s);

// Get all adjacent vertices of the dequeued vertex s. If an adjacent has not been visited, then mark it visited and enqueue it

struct Node\* temp = graph->adj[s];

while (temp) {

int adjVertex = temp->data;

if (!visited[adjVertex]) {

visited[adjVertex] = true;

queue[++rear] = adjVertex;

}

temp = temp->next;

}

}

printf("\n");

free(visited);

free(queue);

}

// Function to perform Depth First Search

void DFS(struct Graph\* graph, int s) {

bool\* visited = (bool\*)malloc(graph->V \* sizeof(bool));

for (int i = 0; i < graph->V; ++i)

visited[i] = false;

// Create a stack for DFS

int\* stack = (int\*)malloc(graph->V \* sizeof(int));

int top = -1;

// Push the current source node.

stack[++top] = s;

while (top != -1) {

// Pop a vertex from stack and print it

s = stack[top--];

if (!visited[s]) {

printf("%d ", s);

visited[s] = true;

}

// Get all adjacent vertices of the popped vertex s. If an adjacent has not been visited, then push it to the stack.

struct Node\* temp = graph->adj[s];

while (temp) {

int adjVertex = temp->data;

if (!visited[adjVertex]) {

stack[++top] = adjVertex;

}

temp = temp->next;

}

}

printf("\n");

free(visited);

free(stack);

}

int main() {

// Create a graph with different vertices and edges

struct Graph\* graph = createGraph(6);

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 3);

addEdge(graph, 1, 4);

addEdge(graph, 2, 4);

addEdge(graph, 2, 5);

addEdge(graph, 3, 4);

addEdge(graph, 4, 5);

printf("Breadth First Traversal (starting from vertex 0): ");

BFS(graph, 0);

printf("Depth First Traversal (starting from vertex 0): ");

DFS(graph, 0);

// Free allocated memory for graph

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

struct Node\* current = graph->adj[i];

while (current) {

struct Node\* temp = current;

current = current->next;

free(temp);

}

}

free(graph->adj);

free(graph);

return 0;

}

# Assignment: 7

# Represent any real world graph using adjacency list /adjacency matrix. Find minimum spanning tree using Kruskal’s algorithm

#include <stdio.h>

#include <stdlib.h>

struct Edge {

int src, dest, weight;

};

struct Node {

int data;

struct Node\* next;

};

struct DisjointSets {

int \*parent, \*rank;

int n;

};

struct Graph {

int V, E;

struct Edge\* edges;

int\*\* adjMatrix;

struct Node\*\* adjList;

};

struct DisjointSets\* createDisjointSets(int n) {

struct DisjointSets\* ds = (struct DisjointSets\*)malloc(sizeof(struct DisjointSets));

ds->n = n;

ds->parent = (int\*)malloc((n+1) \* sizeof(int));

ds->rank = (int\*)malloc((n+1) \* sizeof(int));

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

ds->rank[i] = 0;

ds->parent[i] = i;

}

return ds;

}

int find(struct DisjointSets\* ds, int u) {

if (u != ds->parent[u])

ds->parent[u] = find(ds, ds->parent[u]);

return ds->parent[u];

}

void merge(struct DisjointSets\* ds, int x, int y) {

x = find(ds, x), y = find(ds, y);

if (ds->rank[x] > ds->rank[y])

ds->parent[y] = x;

else

ds->parent[x] = y;

if (ds->rank[x] == ds->rank[y])

ds->rank[y]++;

}

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(E \* sizeof(struct Edge));

graph->adjMatrix = (int\*\*)malloc(V \* sizeof(int\*));

graph->adjList = (struct Node\*\*)malloc(V \* sizeof(struct Node\*));

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

graph->adjMatrix[i] = (int\*)malloc(V \* sizeof(int));

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

graph->adjMatrix[i][j] = 0;

}

graph->adjList[i] = NULL;

}

return graph;

}

void addEdge(struct Graph\* graph, int u, int v, int w) {

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

graph->edges[u].dest = v;

graph->edges[u].weight = w;

// For adjacency matrix

graph->adjMatrix[u][v] = w;

// Include this line for undirected graph

graph->adjMatrix[v][u] = w;

// For adjacency list

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

newNode->data = v;

newNode->next = graph->adjList[u];

graph->adjList[u] = newNode;

// Include this for undirected graph

struct Node\* newNode2 = (struct Node\*)malloc(sizeof(struct Node));

newNode2->data = u;

newNode2->next = graph->adjList[v];

graph->adjList[v] = newNode2;

}

void printAdjMatrix(struct Graph\* graph) {

printf("Adjacency Matrix:\n");

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

for (int j = 0; j < graph->V; j++) {

printf("%d ", graph->adjMatrix[i][j]);

}

printf("\n");

}

}

void printAdjList(struct Graph\* graph) {

printf("Adjacency List:\n");

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

struct Node\* temp = graph->adjList[i];

printf("%d:", i);

while (temp != NULL) {

printf(" -> (%d, %d)", temp->data, graph->edges[temp->data].weight);

temp = temp->next;

}

printf("\n");

}

}

int myComp(const void\* a, const void\* b) {

struct Edge\* e1 = (struct Edge\*)a;

struct Edge\* e2 = (struct Edge\*)b;

return e1->weight - e2->weight;

}

int kruskalMST(struct Graph\* graph) {

int mst\_wt = 0;

qsort(graph->edges, graph->E, sizeof(struct Edge), myComp);

struct DisjointSets\* ds = createDisjointSets(graph->V);

for (int i = 0; i < graph->E; i++) {

int u = graph->edges[i].src;

int v = graph->edges[i].dest;

int set\_u = find(ds, u);

int set\_v = find(ds, v);

if (set\_u != set\_v) {

printf("%d - %d : %d\n", u, v, graph->edges[i].weight);

mst\_wt += graph->edges[i].weight;

merge(ds, set\_u, set\_v);

}

}

free(ds->parent);

free(ds->rank);

free(ds);

return mst\_wt;

}

int main() {

int V = 5, E = 7;

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

addEdge(g, 0, 1, 6);

addEdge(g, 0, 3, 2);

addEdge(g, 1, 2, 8);

addEdge(g, 1, 4, 5);

addEdge(g, 2, 4, 7);

addEdge(g, 3, 4, 9);

addEdge(g, 0, 2, 3);

printAdjMatrix(g);

printf("\n");

printAdjList(g);

printf("Edges of MST are:\n");

int mst\_wt = kruskalMST(g);

printf("\nWeight of MST is: %d\n", mst\_wt);

// Free allocated memory

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

free(g->adjMatrix[i]);

struct Node\* temp = g->adjList[i];

while (temp != NULL) {

struct Node\* prev = temp;

temp = temp->next;

free(prev);

}

}

free(g->adjMatrix);

free(g->adjList);

free(g->edges);

free(g);

return 0;

}

**Assignment: 6**

# A business house has several offices in different countries; they want to lease phone lines to connect them with each other and the phone company charges different rent to connect different pairs of cities. Business house want to connect all its offices with a minimum total cost. Represent using appropriate data structure. Apply Prim’s algorithm to find minimum total cost.

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <limits.h>

#define V 4 // Number of vertices

struct PrimGraph {

int graph[V][V];

};

void addEdge(struct PrimGraph\* primGraph, int u, int v, int weight) {

primGraph->graph[u][v] = weight;

primGraph->graph[v][u] = weight; // Assuming undirected graph

}

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

int min = INT\_MAX, min\_index = -1;

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

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

min = key[v];

min\_index = v;

}

}

return min\_index;

}

void printOriginalGraph(struct PrimGraph\* primGraph);

void printMST(int parent[], struct PrimGraph\* primGraph);

void primMST(struct PrimGraph\* primGraph) {

int parent[V];

int key[V];

bool mstSet[V];

int totalCost = 0;

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

parent[i] = -1;

key[i] = INT\_MAX;

mstSet[i] = false;

}

key[0] = 0; // Start from the first vertex

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

int u = minKey(key, mstSet);

mstSet[u] = true;

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

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

parent[v] = u;

key[v] = primGraph->graph[u][v];

}

}

}

// Calculate total cost

for (int i = 1; i < V; ++i) {

totalCost += primGraph->graph[i][parent[i]];

}

printOriginalGraph(primGraph);

printMST(parent, primGraph);

printf("Minimum Total Cost: %d\n", totalCost);

}

void printOriginalGraph(struct PrimGraph\* primGraph) {

printf("Original Graph (Adjacency Matrix):\n");

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

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

printf("%d ", primGraph->graph[i][j]);

}

printf("\n");

}

printf("\n");

}

void printMST(int parent[], struct PrimGraph\* primGraph) {

printf("Minimum Spanning Tree (MST) using Prim's Algorithm:\n");

printf("Edge Weight\n");

for (int i = 1; i < V; ++i) {

printf("%d - %d %d\n", parent[i], i, primGraph->graph[i][parent[i]]);

}

printf("\n");

}

int main() {

struct PrimGraph primGraph;

// Adding edges and weights (rent cost)

addEdge(&primGraph, 0, 1, 2); // Office 0 to Office 1 with cost 2

addEdge(&primGraph, 0, 2, 3); // Office 0 to Office 2 with cost 3

addEdge(&primGraph, 0, 3, 4); // Office 0 to Office 3 with cost 4

addEdge(&primGraph, 1, 2, 5); // Office 1 to Office 2 with cost 5

addEdge(&primGraph, 1, 3, 6); // Office 1 to Office 3 with cost 6

addEdge(&primGraph, 2, 3, 7); // Office 2 to Office 3 with cost 7

primMST(&primGraph);

return 0;

}

**ASSIGNMENT 9**

The internship is offered to students based on rank obtained in second year of graduation. Create

suitable non-linear data structure to identify next topper student for internship. (**Create max-heap).**

b) Sort the student data in ascending order of grades.

#include <stdio.h>

#include <stdlib.h>

// MaxHeap struct for heap operations

typedef struct MaxHeap {

int \*heapArray;

int capacity;

int currentSize;

} MaxHeap;

// Helper functions to navigate the heap

int parent(int index) { return (index - 1) / 2; }

int leftChild(int index) { return (2 \* index) + 1; }

int rightChild(int index) { return (2 \* index) + 2; }

// Heapify up function

void heapifyUp(MaxHeap \*maxHeap, int index) {

int temp = maxHeap->heapArray[index];

while (index > 0 && temp > maxHeap->heapArray[parent(index)]) {

maxHeap->heapArray[index] = maxHeap->heapArray[parent(index)];

index = parent(index);

}

maxHeap->heapArray[index] = temp;

}

// Heapify down function

void heapifyDown(MaxHeap \*maxHeap, int index) {

int temp = maxHeap->heapArray[index];

int child;

while (leftChild(index) < maxHeap->currentSize) {

child = leftChild(index);

if (child + 1 < maxHeap->currentSize && maxHeap->heapArray[child] < maxHeap->heapArray[child + 1]) {

child++;

}

if (temp >= maxHeap->heapArray[child]) {

break;

}

maxHeap->heapArray[index] = maxHeap->heapArray[child];

index = child;

}

maxHeap->heapArray[index] = temp;

}

// Function to create a new MaxHeap

MaxHeap \*createMaxHeap(int cap) {

MaxHeap \*maxHeap = (MaxHeap \*)malloc(sizeof(MaxHeap));

maxHeap->capacity = cap;

maxHeap->heapArray = (int \*)malloc(cap \* sizeof(int));

maxHeap->currentSize = 0;

return maxHeap;

}

// Function to insert into MaxHeap

void insert(MaxHeap \*maxHeap, int value) {

if (maxHeap->currentSize == maxHeap->capacity) {

printf("Heap is full. Cannot insert more elements.\n");

return;

}

maxHeap->heapArray[maxHeap->currentSize++] = value;

heapifyUp(maxHeap, maxHeap->currentSize - 1);

}

// Function to extract maximum from MaxHeap

int extractMax(MaxHeap \*maxHeap) {

if (maxHeap->currentSize <= 0) {

printf("Heap is empty. Cannot extract maximum.\n");

return -1;

}

int max = maxHeap->heapArray[0];

maxHeap->heapArray[0] = maxHeap->heapArray[maxHeap->currentSize - 1];

maxHeap->currentSize--;

heapifyDown(maxHeap, 0);

return max;

}

// Function to check if MaxHeap is empty

int isEmpty(MaxHeap \*maxHeap) {

return maxHeap->currentSize == 0;

}

// Function to free memory allocated to MaxHeap

void destroyMaxHeap(MaxHeap \*maxHeap) {

free(maxHeap->heapArray);

free(maxHeap);

}

// Function to perform heap sort

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

// Build max-heap

MaxHeap \*maxHeap = createMaxHeap(n);

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

insert(maxHeap, arr[i]);

}

// Extract maximum elements from max-heap to sort the array

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

arr[i] = extractMax(maxHeap);

}

// Free memory allocated to MaxHeap

destroyMaxHeap(maxHeap);

}

int main() {

// Student grades

int grades[] = {85, 92, 80, 95, 88};

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

// Sort the grades array

heapSort(grades, n);

printf("Sorted student grades (ascending order): ");

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

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

}

printf("\n");

// Creating a max-heap for identifying the next topper

MaxHeap \*maxHeap = createMaxHeap(n);

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

insert(maxHeap, grades[i]);

}

// Extracting the next topper (second highest grade)

int nextTopper = extractMax(maxHeap);

printf("Next topper grade for internship: %d\n", nextTopper);

// Free memory allocated to MaxHeap

destroyMaxHeap(maxHeap);

return 0;

}

**Assignment 10 in cpp**

Implement an index sequential file for any **Database** and perform all database operations on it- modify ,delete

import java.io.\*;

import java.util.\*;

class Record {

    int id;

    String name;

    public Record(int id, String name) {

        this.id = id;

        this.name = name;

    }

    public String toString() {

        return "ID: " + id + ", Name: " + name;

    }

}

class IndexSequentialFile {

    private static final int BLOCK\_SIZE = 1024; // Size of each block/page

    private RandomAccessFile dataFile;

    private Map<Integer, Long> indexMap; // Map to store record IDs and their corresponding file positions

    public IndexSequentialFile(String dataFileName) throws IOException {

        dataFile = new RandomAccessFile(dataFileName, "rw");

        indexMap = new HashMap<>();

        loadIndex();

    }

    // Load index from data file into memory

    private void loadIndex() throws IOException {

        long currentPosition = 0;

        while (currentPosition < dataFile.length()) {

            dataFile.seek(currentPosition);

            try {

                int id = dataFile.readInt();

                long position = currentPosition;

                indexMap.put(id, position);

                currentPosition += Integer.BYTES + dataFile.readUTF().getBytes().length;

            } catch (EOFException e) {

                break; // Reached end of file, exit loop

            }

        }

    }

    // Add a record to the database

    public void addRecord(Record record) throws IOException {

        long position = dataFile.length(); // Position to append record

        dataFile.seek(position);

        dataFile.writeInt(record.id);

        dataFile.writeUTF(record.name);

        indexMap.put(record.id, position);

    }

    // Retrieve a record from the database

    public Record getRecord(int id) throws IOException {

        Long position = indexMap.get(id);

        if (position == null) return null;

        dataFile.seek(position);

        int recordId = dataFile.readInt();

        String name = dataFile.readUTF();

        return new Record(recordId, name);

    }

    // Modify a record in the database

    public void modifyRecord(int id, String newName) throws IOException {

        Long position = indexMap.get(id);

        if (position == null) {

            System.out.println("Record with ID " + id + " not found.");

            return;

        }

        dataFile.seek(position + Integer.BYTES); // Move to name position skipping ID

        dataFile.writeUTF(newName); // Write new name

    }

    // Delete a record from the database

    public void deleteRecord(int id) throws IOException {

        Long position = indexMap.remove(id);

        if (position == null) {

            System.out.println("Record with ID " + id + " not found.");

            return;

        }

        // Clear data at the position to "delete" record

        dataFile.seek(position);

        dataFile.writeInt(-1); // Mark ID as -1

        dataFile.writeUTF(""); // Clear name

    }

    public void close() throws IOException {

        dataFile.close();

    }

}

public class main\_10 {

    public static void main(String[] args) {

        try {

            IndexSequentialFile database = new IndexSequentialFile("data.dat");

            Scanner scanner = new Scanner(System.in);

            while (true) {

                System.out.println("\n1. Add Record");

                System.out.println("2. Retrieve Record");

                System.out.println("3. Modify Record");

                System.out.println("4. Delete Record");

                System.out.println("5. Exit");

                System.out.print("Enter your choice: ");

                int choice = scanner.nextInt();

                scanner.nextLine(); // Consume newline character

                switch (choice) {

                    case 1:

                        System.out.print("Enter ID: ");

                        int idToAdd = scanner.nextInt();

                        scanner.nextLine(); // Consume newline character

                        System.out.print("Enter Name: ");

                        String nameToAdd = scanner.nextLine();

                        database.addRecord(new Record(idToAdd, nameToAdd));

                        System.out.println("Record added successfully.");

                        break;

                    case 2:

                        System.out.print("Enter ID to retrieve: ");

                        int idToRetrieve = scanner.nextInt();

                        Record retrievedRecord = database.getRecord(idToRetrieve);

                        if (retrievedRecord != null)

                            System.out.println("Retrieved Record: " + retrievedRecord);

                        else

                            System.out.println("Record not found.");

                        break;

                    case 3:

                        System.out.print("Enter ID to modify: ");

                        int idToModify = scanner.nextInt();

                        scanner.nextLine(); // Consume newline character

                        System.out.print("Enter new Name: ");

                        String newName = scanner.nextLine();

                        database.modifyRecord(idToModify, newName);

                        System.out.println("Record modified successfully.");

                        break;

                    case 4:

                        System.out.print("Enter ID to delete: ");

                        int idToDelete = scanner.nextInt();

                        database.deleteRecord(idToDelete);

                        System.out.println("Record deleted successfully.");

                        break;

                    case 5:

                        database.close();

                        scanner.close();

                        System.exit(0);

                    default:

                        System.out.println("Invalid choice. Please enter a number between 1 and 5.");

                }

            }

        } catch (IOException e) {

            e.printStackTrace();

        }  }}

**Assignment 11**

Implement direct access file for any **Database** and perform following operations on it i) Create

Database ii) Display Database iii) Search a record

import java.io.\*;

public class main\_11 {

    private static String DATABASE\_FILE;

    public static void main(String[] args) {

        try {

            BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

            System.out.print("Enter database file name: ");

            DATABASE\_FILE = reader.readLine();

            while (true) {

                System.out.println("\nMenu:");

                System.out.println("1. Create Database");

                System.out.println("2. Display Database");

                System.out.println("3. Add Record");

                System.out.println("4. Search Record");

                System.out.println("5. Exit");

                System.out.print("Enter your choice: ");

                int choice = Integer.parseInt(reader.readLine());

                switch (choice) {

                    case 1:

                        createDatabase();

                        break;

                    case 2:

                        displayDatabase();

                        break;

                    case 3:

                        addRecord();

                        break;

                    case 4:

                        searchRecord();

                        break;

                    case 5:

                        System.out.println("Exiting...");

                        return;

                    default:

                        System.out.println("Invalid choice! Please enter a number between 1 and 5.");

                }

            }

        } catch (IOException | NumberFormatException e) {

            e.printStackTrace();

        }

    }

    private static void createDatabase() throws IOException {

        File file = new File(DATABASE\_FILE);

        if (file.createNewFile()) {

            System.out.println("Database created successfully.");

        } else {

            System.out.println("Database already exists.");

        }

    }

    private static void displayDatabase() throws IOException {

        BufferedReader reader = new BufferedReader(new FileReader(DATABASE\_FILE));

        String line;

        System.out.println("\nDatabase Records:");

        while ((line = reader.readLine()) != null) {

            System.out.println(line);

        }

        reader.close();

    }

    private static void addRecord() throws IOException {

        BufferedWriter writer = new BufferedWriter(new FileWriter(DATABASE\_FILE, true)); // Append mode

        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        System.out.print("Enter ID: ");

        String id = reader.readLine();

        System.out.print("Enter Name: ");

        String name = reader.readLine();

        writer.write(id + "," + name);

        writer.newLine();

        writer.close();

        System.out.println("Record added to the database successfully.");

    }

    private static void searchRecord() throws IOException {

        BufferedReader reader = new BufferedReader(new FileReader(DATABASE\_FILE));

        BufferedReader userInputReader = new BufferedReader(new InputStreamReader(System.in));

        System.out.print("Enter the ID to search: ");

        String searchId = userInputReader.readLine().trim(); // trim the input

        String line;

        boolean found = false;

        while ((line = reader.readLine()) != null) {

            String[] record = line.split(",");

            if (record.length >= 2 && record[0].trim().equals(searchId)) { // trim each record ID

                System.out.println("Record found: ID=" + record[0] + ", Name=" + record[1]);

                found = true;

                break;

            }

        }

        if (!found) {

            System.out.println("Record not found.");

        }

        reader.close();

    }

}