QUEUE USING ARRAYS

cc

}

}

}

Tree traversal

#include <stdio.h>

#include <stdlib.h>

// Define a simple structure for a binary tree node

struct TreeNode {

int data;

struct TreeNode\* left;

struct TreeNode\* right;

};

// Function to create a new node with the given data

struct TreeNode\* createNode(int data) {

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

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Function to insert a value into the binary tree

struct TreeNode\* insert(struct TreeNode\* root, int value) {

if (root == NULL) {

return createNode(value);

}

if (value < root->data) {

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

} else if (value > root->data) {

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

}

return root;

}

// Recursive Inorder Traversal

void inorderRecursive(struct TreeNode\* root) {

if (root != NULL) {

inorderRecursive(root->left);

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

inorderRecursive(root->right);

}

}

// Recursive Preorder Traversal

void preorderRecursive(struct TreeNode\* root) {

if (root != NULL) {

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

preorderRecursive(root->left);

preorderRecursive(root->right);

}

}

// Recursive Postorder Traversal

void postorderRecursive(struct TreeNode\* root) {

if (root != NULL) {

postorderRecursive(root->left);

postorderRecursive(root->right);

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

}

}

int main() {

struct TreeNode\* root = NULL;

int n, value;

printf("Enter the number of nodes in the binary tree: ");

scanf("%d", &n);

printf("Enter the values for the binary tree nodes:\n");

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

scanf("%d", &value);

root = insert(root, value);

}

// Perform recursive traversals

printf("Inorder (Recursive): ");

inorderRecursive(root);

printf("\n");

printf("Preorder (Recursive): ");

preorderRecursive(root);

printf("\n");

printf("Postorder (Recursive): ");

postorderRecursive(root);

printf("\n");

return 0;

}

8 a)BINARY TREE

#include <stdio.h>

#include <stdlib.h>

// Define a simple structure for a binary tree node

struct TreeNode {

int data;

struct TreeNode\* left;

struct TreeNode\* right;

};

// Function to create a new node with the given data

struct TreeNode\* createNode(int data) {

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

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

// Function to insert a value into the binary search tree

struct TreeNode\* insert(struct TreeNode\* root, int value) {

if (root == NULL) {

return createNode(value);

}

if (value < root->data) {

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

} else if (value > root->data) {

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

}

return root;

}

// Function to perform an inorder traversal of the binary search tree

void inorderTraversal(struct TreeNode\* root) {

if (root != NULL) {

inorderTraversal(root->left);

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

inorderTraversal(root->right);

}

}

int main() {

struct TreeNode\* root = NULL;

int n, value;

printf("Enter the number of nodes in the binary search tree: ");

scanf("%d", &n);

printf("Enter the values for the binary search tree nodes:\n");

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

scanf("%d", &value);

root = insert(root, value);

}

printf("\nInorder Traversal of Binary Search Tree: ");

inorderTraversal(root);

printf("\n");

return 0;

}

8 b)B tree

#include <stdio.h>

#include <stdlib.h>

// Define the order of the B-tree

#define ORDER 3

// Define a structure for a B-tree node

struct BTreeNode {

int n; // Number of keys in the node

int keys[2 \* ORDER - 1]; // Array to store keys

struct BTreeNode\* children[2 \* ORDER]; // Array to store child pointers

int leaf; // Flag indicating whether the node is a leaf (1 for true, 0 for false)

};

// Function to create a new B-tree node

struct BTreeNode\* createNode() {

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

newNode->n = 0;

newNode->leaf = 1; // By default, a new node is a leaf

for (int i = 0; i < 2 \* ORDER; ++i) {

newNode->keys[i] = 0;

newNode->children[i] = NULL;

}

return newNode;

}

// Function to split a child node of a B-tree node

void splitChild(struct BTreeNode\* parent, int i, struct BTreeNode\* child) {

struct BTreeNode\* newNode = createNode();

newNode->n = ORDER - 1;

for (int j = 0; j < ORDER - 1; ++j) {

newNode->keys[j] = child->keys[j + ORDER];

}

if (!child->leaf) {

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

newNode->children[j] = child->children[j + ORDER];

}

}

child->n = ORDER - 1;

for (int j = parent->n; j > i; --j) {

parent->children[j + 1] = parent->children[j];

}

parent->children[i + 1] = newNode;

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

parent->keys[j + 1] = parent->keys[j];

}

parent->keys[i] = child->keys[ORDER - 1];

parent->n++;

}

// Function to insert a key into a B-tree

void insertBTree(struct BTreeNode\*\* root, int key) {

struct BTreeNode\* rootNode = \*root;

if (rootNode == NULL) {

\*root = createNode();

rootNode = \*root;

}

if (rootNode->n == 2 \* ORDER - 1) {

struct BTreeNode\* newNode = createNode();

\*root = newNode;

newNode->children[0] = rootNode;

splitChild(newNode, 0, rootNode);

insertBTreeNonFull(newNode, key);

} else {

insertBTreeNonFull(rootNode, key);

}

}

// Function to insert a key into a non-full B-tree node

void insertBTreeNonFull(struct BTreeNode\* node, int key) {

int i = node->n - 1;

if (node->leaf) {

while (i >= 0 && key < node->keys[i]) {

node->keys[i + 1] = node->keys[i];

i--;

}

node->keys[i + 1] = key;

node->n++;

} else {

while (i >= 0 && key < node->keys[i]) {

i--;

}

i++;

if (node->children[i]->n == 2 \* ORDER - 1) {

splitChild(node, i, node->children[i]);

if (key > node->keys[i]) {

i++;

}

}

insertBTreeNonFull(node->children[i], key);

}

}

// Function to perform an inorder traversal of a B-tree

void inorderTraversalBTree(struct BTreeNode\* root) {

if (root != NULL) {

int i;

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

inorderTraversalBTree(root->children[i]);

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

}

inorderTraversalBTree(root->children[i]);

}

}

int main() {

struct BTreeNode\* root = NULL;

int n, key;

printf("Enter the number of keys to insert into the B-tree: ");

scanf("%d", &n);

printf("Enter the keys to insert into the B-tree:\n");

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

scanf("%d", &key);

insertBTree(&root, key);

}

printf("\nInorder Traversal of B-tree: ");

inorderTraversalBTree(root);

printf("\n");

return 0;

}

8 c) B+ tree

#include <stdio.h>

#include <stdlib.h>

#define ORDER 3

struct BPlusNode {

int num\_keys;

int keys[2 \* ORDER - 1];

struct BPlusNode\* children[2 \* ORDER];

struct BPlusNode\* next;

};

struct BPlusTree {

struct BPlusNode\* root;

};

struct BPlusNode\* createNode() {

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

newNode->num\_keys = 0;

newNode->next = NULL;

for (int i = 0; i < 2 \* ORDER - 1; ++i) {

newNode->keys[i] = 0;

newNode->children[i] = NULL;

}

newNode->children[2 \* ORDER - 1] = NULL;

return newNode;

}

struct BPlusTree\* createBPlusTree() {

struct BPlusTree\* newTree = (struct BPlusTree\*)malloc(sizeof(struct BPlusTree));

newTree->root = NULL;

return newTree;

}

void splitChild(struct BPlusNode\* parent, int index) {

struct BPlusNode\* newChild = createNode();

struct BPlusNode\* oldChild = parent->children[index];

newChild->num\_keys = ORDER - 1;

for (int j = 0; j < ORDER - 1; ++j) {

newChild->keys[j] = oldChild->keys[j + ORDER];

}

if (!oldChild->next) {

oldChild->next = newChild;

} else {

newChild->next = oldChild->next;

oldChild->next = newChild;

}

oldChild->num\_keys = ORDER - 1;

for (int j = parent->num\_keys; j > index; --j) {

parent->children[j + 1] = parent->children[j];

}

parent->children[index + 1] = newChild;

for (int j = parent->num\_keys - 1; j >= index; --j) {

parent->keys[j + 1] = parent->keys[j];

}

parent->keys[index] = oldChild->keys[ORDER - 1];

parent->num\_keys++;

}

void insertNonFull(struct BPlusNode\* node, int key) {

int i = node->num\_keys - 1;

if (node->children[0] == NULL) {

while (i >= 0 && key < node->keys[i]) {

node->keys[i + 1] = node->keys[i];

i--;

}

node->keys[i + 1] = key;

node->num\_keys++;

} else {

while (i >= 0 && key < node->keys[i]) {

i--;

}

i++;

if (node->children[i]->num\_keys == 2 \* ORDER - 1) {

splitChild(node, i);

if (key > node->keys[i]) {

i++;

}

}

insertNonFull(node->children[i], key);

}

}

void insert(struct BPlusTree\* tree, int key) {

struct BPlusNode\* root = tree->root;

if (!root) {

root = createNode();

tree->root = root;

}

if (root->num\_keys == 2 \* ORDER - 1) {

struct BPlusNode\* newRoot = createNode();

newRoot->children[0] = root;

tree->root = newRoot;

splitChild(newRoot, 0);

insertNonFull(newRoot, key);

} else {

insertNonFull(root, key);

}

}

struct BPlusNode\* search(struct BPlusNode\* node, int key) {

int i = 0;

while (i < node->num\_keys && key > node->keys[i]) {

i++;

}

if (i < node->num\_keys && key == node->keys[i]) {

return node;

} else if (node->children[0] == NULL) {

return NULL;

} else {

return search(node->children[i], key);

}

}

void printLeafNodes(struct BPlusNode\* root) {

struct BPlusNode\* current = root;

while (current->children[0] != NULL) {

current = current->children[0];

}

while (current != NULL) {

for (int i = 0; i < current->num\_keys; i++) {

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

}

current = current->next;

}

}

int main() {

struct BPlusTree\* bPlusTree = createBPlusTree();

int n, key;

printf("Enter the number of keys to insert into the B+ tree: ");

scanf("%d", &n);

printf("Enter the keys to insert into the B+ tree:\n");

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

scanf("%d", &key);

insert(bPlusTree, key);

}

printf("\nEnter the key to search for in the B+ tree: ");

scanf("%d", &key);

struct BPlusNode\* result = search(bPlusTree->root, key);

if (result) {

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

} else {

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

}

printf("\nLeaf Nodes of B+ Tree: ");

printLeafNodes(bPlusTree->root);

printf("\n");

return 0;

}

8 d)AVL tree

#include <stdio.h>

#include <stdlib.h>

// Define a structure for an AVL tree node

struct AVLNode {

int data;

struct AVLNode\* left;

struct AVLNode\* right;

int height;

};

// Function to calculate the height of a node

int height(struct AVLNode\* node) {

if (node == NULL) {

return 0;

}

return node->height;

}

// Function to calculate the balance factor of a node

int getBalanceFactor(struct AVLNode\* node) {

if (node == NULL) {

return 0;

}

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

}

// Function to create a new AVL tree node

struct AVLNode\* createNode(int data) {

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

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

newNode->height = 1;

return newNode;

}

// Function to perform a right rotation

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

struct AVLNode\* x = y->left;

struct AVLNode\* T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Update heights

y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));

x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));

return x;

}

// Function to perform a left rotation

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

struct AVLNode\* y = x->right;

struct AVLNode\* T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));

y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));

return y;

}

// Function to insert a key into the AVL tree

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

// Standard BST insertion

if (root == NULL) {

return createNode(data);

}

if (data < root->data) {

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

} else if (data > root->data) {

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

} else {

// Duplicate keys are not allowed in this example

return root;

}

// Update height of the current node

root->height = 1 + (height(root->left) > height(root->right) ? height(root->left) : height(root->right));

// Get the balance factor of this node to check whether it became unbalanced

int balance = getBalanceFactor(root);

// Perform rotations if necessary

// Left Left Case

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

return rightRotate(root);

}

// Right Right Case

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

return leftRotate(root);

}

// Left Right Case

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

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

return rightRotate(root);

}

// Right Left Case

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

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

return leftRotate(root);

}

return root;

}

// Function to perform an inorder traversal of the AVL tree

void inorderTraversal(struct AVLNode\* root) {

if (root != NULL) {

inorderTraversal(root->left);

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

inorderTraversal(root->right);

}

}

int main() {

struct AVLNode\* root = NULL;

int n, data;

printf("Enter the number of nodes to insert into the AVL tree: ");

scanf("%d", &n);

printf("Enter the nodes to insert into the AVL tree:\n");

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

scanf("%d", &data);

root = insert(root, data);

}

printf("\nInorder Traversal of AVL Tree: ");

inorderTraversal(root);

printf("\n");

return 0;

}

8 e)RED BLACK Tree

#include <stdio.h>

#include <stdlib.h>

// Define colors for the Red-Black Tree

#define RED 0

#define BLACK 1

// Define a structure for a Red-Black Tree node

struct RBNode {

int data;

int color;

struct RBNode\* parent;

struct RBNode\* left;

struct RBNode\* right;

};

// Define a structure for the Red-Black Tree

struct RBTree {

struct RBNode\* root;

struct RBNode\* nil; // Sentinel node representing NULL

};

// Function to create a new Red-Black Tree node

struct RBNode\* createNode(int data, struct RBNode\* nil) {

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

newNode->data = data;

newNode->color = RED; // Newly inserted node is always red

newNode->parent = nil;

newNode->left = nil;

newNode->right = nil;

return newNode;

}

// Function to create a Red-Black Tree

struct RBTree\* createRBTree() {

struct RBTree\* newTree = (struct RBTree\*)malloc(sizeof(struct RBTree));

newTree->nil = createNode(0, NULL);

newTree->nil->color = BLACK; // Set the sentinel node to black

newTree->root = newTree->nil;

return newTree;

}

// Function to perform a left rotation in the Red-Black Tree

void leftRotate(struct RBTree\* tree, struct RBNode\* x) {

struct RBNode\* y = x->right;

x->right = y->left;

if (y->left != tree->nil) {

y->left->parent = x;

}

y->parent = x->parent;

if (x->parent == tree->nil) {

tree->root = y;

} else if (x == x->parent->left) {

x->parent->left = y;

} else {

x->parent->right = y;

}

y->left = x;

x->parent = y;

}

// Function to perform a right rotation in the Red-Black Tree

void rightRotate(struct RBTree\* tree, struct RBNode\* y) {

struct RBNode\* x = y->left;

y->left = x->right;

if (x->right != tree->nil) {

x->right->parent = y;

}

x->parent = y->parent;

if (y->parent == tree->nil) {

tree->root = x;

} else if (y == y->parent->left) {

y->parent->left = x;

} else {

y->parent->right = x;

}

x->right = y;

y->parent = x;

}

// Function to fix the Red-Black Tree properties after insertion

void insertFixup(struct RBTree\* tree, struct RBNode\* z) {

while (z->parent->color == RED) {

if (z->parent == z->parent->parent->left) {

struct RBNode\* y = z->parent->parent->right;

if (y->color == RED) {

z->parent->color = BLACK;

y->color = BLACK;

z->parent->parent->color = RED;

z = z->parent->parent;

} else {

if (z == z->parent->right) {

z = z->parent;

leftRotate(tree, z);

}

z->parent->color = BLACK;

z->parent->parent->color = RED;

rightRotate(tree, z->parent->parent);

}

} else {

struct RBNode\* y = z->parent->parent->left;

if (y->color == RED) {

z->parent->color = BLACK;

y->color = BLACK;

z->parent->parent->color = RED;

z = z->parent->parent;

} else {

if (z == z->parent->left) {

z = z->parent;

rightRotate(tree, z);

}

z->parent->color = BLACK;

z->parent->parent->color = RED;

leftRotate(tree, z->parent->parent);

}

}

}

tree->root->color = BLACK;

}

// Function to insert a key into the Red-Black Tree

void insertRBTree(struct RBTree\* tree, int data) {

struct RBNode\* z = createNode(data, tree->nil);

struct RBNode\* y = tree->nil;

struct RBNode\* x = tree->root;

while (x != tree->nil) {

y = x;

if (z->data < x->data) {

x = x->left;

} else {

x = x->right;

}

}

z->parent = y;

if (y == tree->nil) {

tree->root = z;

} else if (z->data < y->data) {

y->left = z;

} else {

y->right = z;

}

insertFixup(tree, z);

}

// Function to perform an inorder traversal of the Red-Black Tree

void inorderTraversalRBTree(struct RBNode\* root, struct RBNode\* nil) {

if (root != nil) {

inorderTraversalRBTree(root->left, nil);

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

inorderTraversalRBTree(root->right, nil);

}

}

int main() {

struct RBTree\* rbTree = createRBTree();

int n, data;

printf("Enter the number of nodes to insert into the Red-Black Tree: ");

scanf("%d", &n);

printf("Enter the nodes to insert into the Red-Black Tree:\n");

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

scanf("%d", &data);

insertRBTree(rbTree, data);

}

printf("\nInorder Traversal of Red-Black Tree: ");

inorderTraversalRBTree(rbTree->root, rbTree->nil);

printf("\n");

return 0;

}

9 tree traversal

#include <stdio.h>

#include <stdlib.h>

#define MAX\_VERTICES 100

// Structure to represent a graph

struct Graph {

int vertices;

int\*\* adjacencyMatrix;

};

// Function to initialize a graph with a given number of vertices

struct Graph\* initializeGraph(int vertices) {

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

graph->vertices = vertices;

// Allocate memory for adjacency matrix

graph->adjacencyMatrix = (int\*\*)malloc(vertices \* sizeof(int\*));

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

graph->adjacencyMatrix[i] = (int\*)malloc(vertices \* sizeof(int));

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

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

}

}

return graph;

}

// Function to add an edge to the graph

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

// Assuming an undirected graph

graph->adjacencyMatrix[src][dest] = 1;

graph->adjacencyMatrix[dest][src] = 1;

}

// Function to perform Depth-First Search (DFS) traversal

void DFS(struct Graph\* graph, int vertex, int visited[]) {

printf("%d ", vertex);

visited[vertex] = 1;

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

if (graph->adjacencyMatrix[vertex][i] == 1 && !visited[i]) {

DFS(graph, i, visited);

}

}

}

// Function to perform Breadth-First Search (BFS) traversal

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

int queue[MAX\_VERTICES];

int front = -1, rear = -1;

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

queue[++rear] = start;

visited[start] = 1;

while (front != rear) {

int currentVertex = queue[++front];

printf("%d ", currentVertex);

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

if (graph->adjacencyMatrix[currentVertex][i] == 1 && !visited[i]) {

queue[++rear] = i;

visited[i] = 1;

}

}

}

}

// Function to free the memory allocated for the graph

void freeGraph(struct Graph\* graph) {

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

free(graph->adjacencyMatrix[i]);

}

free(graph->adjacencyMatrix);

free(graph);

}

int main() {

int vertices, edges;

printf("Enter the number of vertices in the graph: ");

scanf("%d", &vertices);

struct Graph\* graph = initializeGraph(vertices);

printf("Enter the number of edges in the graph: ");

scanf("%d", &edges);

printf("Enter the edges (source and destination) in the graph:\n");

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

int src, dest;

scanf("%d %d", &src, &dest);

addEdge(graph, src, dest);

}

int startVertex;

printf("Enter the starting vertex for traversal: ");

scanf("%d", &startVertex);

printf("\nDepth-First Search (DFS) Traversal: ");

int visitedDFS[MAX\_VERTICES] = {0};

DFS(graph, startVertex, visitedDFS);

printf("\n");

printf("Breadth-First Search (BFS) Traversal: ");

BFS(graph, startVertex);

printf("\n");

// Free allocated memory

freeGraph(graph);

return 0;

}

10 PATTERN MATCHING ALGORITHMS

#include <stdio.h>

#include <string.h>

// Function to compute the "bad character" heuristic for Boyer-Moore

void computeBadCharHeuristic(char\* pattern, int badCharShift[]) {

int patternLength = strlen(pattern);

for (int i = 0; i < 256; i++) {

badCharShift[i] = patternLength;

}

for (int i = 0; i < patternLength - 1; i++) {

badCharShift[(unsigned char)pattern[i]] = patternLength - 1 - i;

}

}

// Function to compute the "good suffix" heuristic for Boyer-Moore

void computeGoodSuffixHeuristic(char\* pattern, int goodSuffixShift[]) {

int patternLength = strlen(pattern);

int\* suffixArray = (int\*)malloc(patternLength \* sizeof(int));

int lastPrefixPosition = patternLength;

// Step 1: Fill the suffix array

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

if (isPrefix(pattern, i + 1)) {

lastPrefixPosition = i + 1;

}

suffixArray[patternLength - 1 - i] = lastPrefixPosition - i + patternLength - 1;

}

// Step 2: Fill the remaining suffixes

for (int i = 0; i < patternLength - 1; i++) {

int suffixLength = getSuffixLength(pattern, i);

suffixArray[suffixLength] = patternLength - 1 - i + suffixLength;

}

// Step 3: Fill the good suffix shift array

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

goodSuffixShift[i] = patternLength;

}

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

if (suffixArray[i] == i + 1) {

for (int j = 0; j < patternLength - 1 - i; j++) {

if (goodSuffixShift[j] == patternLength) {

goodSuffixShift[j] = patternLength - 1 - i;

}

}

}

}

free(suffixArray);

}

// Function to check if a string is a prefix of another string

int isPrefix(char\* pattern, int position) {

int patternLength = strlen(pattern);

for (int i = position, j = 0; i < patternLength; i++, j++) {

if (pattern[i] != pattern[j]) {

return 0;

}

}

return 1;

}

// Function to get the length of the longest suffix that is also a prefix

int getSuffixLength(char\* pattern, int position) {

int length = 0;

int patternLength = strlen(pattern);

for (int i = position, j = patternLength - 1; i >= 0 && pattern[i] == pattern[j]; i--, j--) {

length++;

}

return length;

}

// Boyer-Moore pattern matching algorithm

void boyerMoore(char\* text, char\* pattern) {

int textLength = strlen(text);

int patternLength = strlen(pattern);

int badCharShift[256];

int goodSuffixShift[patternLength];

computeBadCharHeuristic(pattern, badCharShift);

computeGoodSuffixHeuristic(pattern, goodSuffixShift);

int i = 0;

while (i <= textLength - patternLength) {

int j = patternLength - 1;

while (j >= 0 && pattern[j] == text[i + j]) {

j--;

}

if (j < 0) {

printf("Boyer-Moore: Pattern found at index %d\n", i);

i += goodSuffixShift[0];

} else {

int badCharShiftValue = badCharShift[(unsigned char)text[i + j]] - (patternLength - 1 - j);

i += (badCharShiftValue > 0) ? badCharShiftValue : goodSuffixShift[j];

}

}

}

// Knuth-Morris-Pratt pattern matching algorithm

void knuthMorrisPratt(char\* text, char\* pattern) {

int textLength = strlen(text);

int patternLength = strlen(pattern);

int lps[patternLength];

computeLPSArray(pattern, lps);

int i = 0;

int j = 0;

while (i < textLength) {

if (pattern[j] == text[i]) {

j++;

i++;

}

if (j == patternLength) {

printf("Knuth-Morris-Pratt: Pattern found at index %d\n", i - j);

j = lps[j - 1];

} else if (i < textLength && pattern[j] != text[i]) {

if (j != 0) {

j = lps[j - 1];

} else {

i++;

}

}

}

}

// Function to compute the Longest Prefix Suffix (LPS) array for KMP

void computeLPSArray(char\* pattern, int lps[]) {

int patternLength = strlen(pattern);

int len = 0;

int i = 1;

lps[0] = 0;

while (i < patternLength) {

if (pattern[i] == pattern[len]) {

len++;

lps[i] = len;

i++;

} else {

if (len != 0) {

len = lps[len - 1];

} else {

lps[i] = 0;

i++;

}

}

}

}

int main() {

char text[1000], pattern[100];

printf("Enter the text: ");

gets(text);

printf("Enter the pattern: ");

gets(pattern);

boyerMoore(text, pattern);

knuthMorrisPratt(text, pattern);

return 0;

}