Q1. Write a C program to solve the n-queens problem. The value of n may be between 8 and 1000. We are interested in finding the first achieved solution only. (not all the solutions)

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

#define MAX\_N 1000

int board[MAX\_N];

bool isSafe(int row, int col, int n) {

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

if (board[i] == col ||

abs(board[i] - col) == abs(i - row)) {

return false;

}

}

return true;

}

bool solveNQueens(int row, int n) {

if (row == n) {

return true;

}

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

if (isSafe(row, col, n)) {

board[row] = col;

if (solveNQueens(row + 1, n)) {

return true;

}

}

}

return false;

}

void printSolution(int n) {

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

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

if (board[i] == j) {

printf("Q ");

} else {

printf(". ");

}

}

printf("\n");

}

}

int main() {

int n;

printf("Enter the value of N (8 to 1000): ");

scanf("%d", &n);

if (n < 8 || n > 1000) {

printf("Invalid input. N should be between 8 and 1000.\n");

return 1;

}

if (solveNQueens(0, n)) {

printf("First solution found:\n");

printSolution(n);

} else {

printf("No solution exists for N = %d.\n", n);

}

return 0;

}

Q2. Define the ADTs for Stack and Queue.

a) Implement the ADTS using Array.

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_SIZE 1000

//Stack

typedef struct {

int data[MAX\_SIZE];

int top;

} Stack;

bool isStackEmpty(Stack \*stack) {

return stack->top == -1;

}

bool isStackFull(Stack \*stack) {

return stack->top == MAX\_SIZE - 1;

}

void push(Stack \*stack, int value) {

if (isStackFull(stack)) {

printf("Stack Overflow\n");

return;

}

stack->data[++stack->top] = value;

}

int pop(Stack \*stack) {

if (isStackEmpty(stack)) {

printf("Stack Underflow\n");

return -1;

}

return stack->data[stack->top--];

}

int peek(Stack \*stack) {

if (isStackEmpty(stack)) {

printf("Stack is empty\n");

return -1;

}

return stack->data[stack->top];

}

//Queue

typedef struct {

int data[MAX\_SIZE];

int front;

int rear;

int size;

} Queue;

void initStack(Stack \*stack) {

stack->top = -1;

}

void initQueue(Queue \*queue) {

queue->front = 0;

queue->rear = -1;

queue->size = 0;

}

bool isQueueEmpty(Queue \*queue) {

return queue->size == 0;

}

bool isQueueFull(Queue \*queue) {

return queue->size == MAX\_SIZE;

}

void enqueue(Queue \*queue, int value) {

if (isQueueFull(queue)) {

printf("Queue Overflow\n");

return;

}

queue->rear = (queue->rear + 1) % MAX\_SIZE;

queue->data[queue->rear] = value;

queue->size++;

}

int dequeue(Queue \*queue) {

if (isQueueEmpty(queue)) {

printf("Queue Underflow\n");

return -1;

}

int value = queue->data[queue->front];

queue->front = (queue->front + 1) % MAX\_SIZE;

queue->size--;

return value;

}

int front(Queue \*queue) {

if (isQueueEmpty(queue)) {

printf("Queue is empty\n");

return -1;

}

return queue->data[queue->front];

}

b) Implement the ADTs using Linked List. Use the data structures for Linked List already developed in Assignment II as header files.

//Stack

typedef struct Node {

int data;

struct Node \*next;

} Node;

Node\* createNode(int value);

void deleteNode(Node \*node);

typedef struct {

Node \*top;

} LinkedStack;

void initLinkedStack(LinkedStack \*stack) {

stack->top = NULL;

}

bool isLinkedStackEmpty(LinkedStack \*stack) {

return stack->top == NULL;

}

void pushLinked(LinkedStack \*stack, int value) {

Node \*newNode = createNode(value);

newNode->next = stack->top;

stack->top = newNode;

}

int popLinked(LinkedStack \*stack) {

if (isLinkedStackEmpty(stack)) {

printf("Stack Underflow\n");

return -1;

}

Node \*temp = stack->top;

int value = temp->data;

stack->top = stack->top->next;

deleteNode(temp);

return value;

}

int peekLinked(LinkedStack \*stack) {

if (isLinkedStackEmpty(stack)) {

printf("Stack is empty\n");

return -1;

}

return stack->top->data;

}

//Queue

typedef struct {

Node \*front;

Node \*rear;

} LinkedQueue;

void initLinkedQueue(LinkedQueue \*queue) {

queue->front = queue->rear = NULL;

}

bool isLinkedQueueEmpty(LinkedQueue \*queue) {

return queue->front == NULL;

}

void enqueueLinked(LinkedQueue \*queue, int value) {

Node \*newNode = createNode(value);

if (queue->rear == NULL) {

queue->front = queue->rear = newNode;

return;

}

queue->rear->next = newNode;

queue->rear = newNode;

}

int dequeueLinked(LinkedQueue \*queue) {

if (isLinkedQueueEmpty(queue)) {

printf("Queue Underflow\n");

return -1;

}

Node \*temp = queue->front;

int value = temp->data;

queue->front = queue->front->next;

if (queue->front == NULL) {

queue->rear = NULL;

}

deleteNode(temp);

return value;

}

int frontLinked(LinkedQueue \*queue) {

if (isLinkedQueueEmpty(queue)) {

printf("Queue is empty\n");

return -1;

}

return queue->front->data;

}

c) Develop a program for converting a string containing an infix expression to a string containing the corresponding postfix expression. Both the strings are terminated by a sentinel ‘#’.

#include <ctype.h>

#include <string.h>

int precedence(char op) {

switch (op) {

case '+': case '-': return 1;

case '\*': case '/': return 2;

case '^': return 3;

default: return 0;

}

}

bool isOperator(char ch) {

return ch == '+' || ch == '-' || ch == '\*' || ch == '/' || ch == '^';

}

void infixToPostfix(char \*infix, char \*postfix) {

Stack stack;

initStack(&stack);

int k = 0; // Index for postfix string

for (int i = 0; infix[i] != '#'; i++) {

char ch = infix[i];

if (isalnum(ch)) {

postfix[k++] = ch;

} else if (ch == '(') {

push(&stack, ch);

} else if (ch == ')') {

while (!isStackEmpty(&stack) && peek(&stack) != '(') {

postfix[k++] = pop(&stack);

}

pop(&stack);

} else if (isOperator(ch)) {

while (!isStackEmpty(&stack) && precedence(peek(&stack)) >= precedence(ch)) {

postfix[k++] = pop(&stack);

}

push(&stack, ch);

}

}

while (!isStackEmpty(&stack)) {

postfix[k++] = pop(&stack);

}

postfix[k] = '#'; // Sentinel

postfix[k + 1] = '\0';

}

int main() {

char infix[MAX\_SIZE], postfix[MAX\_SIZE];

printf("Enter infix expression terminated by #: ");

scanf("%s", infix);

infixToPostfix(infix, postfix);

printf("Postfix expression: %s\n", postfix);

return 0;

}

Q3. Develop a recursive program for generating all the permutations of the letters ‘a’ to ‘f’ with repetitions taking 3, 4, 5 and 6 characters.

#include <stdio.h>

#include <string.h>

#define CHARSET "abcdef"

// Recursive function to generate permutations

void generatePermutations(char \*current, int depth, int length) {

if (depth == length) {

current[depth] = '\0';

printf("%s\n", current);

return;

}

for (int i = 0; i < strlen(CHARSET); i++) {

current[depth] = CHARSET[i];

generatePermutations(current, depth + 1, length);

}

}

int main() {

int lengths[] = {3, 4, 5, 6};

char current[7]; // Buffer to store permutations (6 characters + null terminator)

printf("Generating permutations for lengths 3, 4, 5, and 6:\n");

for (int i = 0; i < 4; i++) {

int length = lengths[i];

printf("\nPermutations of length %d:\n", length);

generatePermutations(current, 0, length);

}

return 0;

}

Q4. Define an ADT for a threaded Binary Tree. Implement the ADT including the three traversals.

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

// Definition of a node in a threaded binary tree

typedef struct Node {

int data;

struct Node \*left, \*right;

bool isLeftThread, isRightThread; // Flags to indicate threads

} Node;

// Function prototypes for the Threaded Binary Tree ADT

Node\* createNode(int data);

Node\* insert(Node \*root, int data);

void inOrderTraversal(Node \*root);

void preOrderTraversal(Node \*root);

void postOrderTraversal(Node \*root);

Node\* createNode(int data) {

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

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

newNode->isLeftThread = true; // Initially, the left pointer is a thread

newNode->isRightThread = true; // Initially, the right pointer is a thread

return newNode;

}

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

Node \*newNode = createNode(data);

if (root == NULL) {

return newNode;

}

Node \*current = root, \*parent = NULL;

while (current != NULL) {

parent = current;

if (data < current->data) {

if (!current->isLeftThread) {

current = current->left;

} else {

break;

}

} else {

if (!current->isRightThread) {

current = current->right;

} else {

break;

}

}

}

if (data < parent->data) {

newNode->left = parent->left;

newNode->right = parent;

parent->isLeftThread = false;

parent->left = newNode;

} else {

newNode->right = parent->right;

newNode->left = parent;

parent->isRightThread = false;

parent->right = newNode;

}

return root;

}

void inOrderTraversal(Node \*root) {

Node \*current = root;

// Find the leftmost node

while (current != NULL && !current->isLeftThread) {

current = current->left;

}

while (current != NULL) {

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

if (current->isRightThread) {

current = current->right;

} else {

// Otherwise, find the leftmost node in the right subtree

current = current->right;

while (current != NULL && !current->isLeftThread) {

current = current->left;

}

}

}

printf("\n");

}

void preOrderTraversal(Node \*root) {

Node \*current = root;

while (current != NULL) {

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

if (!current->isLeftThread) {

current = current->left;

} else {

// Follow the threads if no left child

while (current != NULL && current->isRightThread) {

current = current->right;

}

if (current != NULL) {

current = current->right;

}

}

}

printf("\n");

}

Q5. Define an ADT for Binary Search Tree.

a) Implement the ADT.

b) Write functions for Print-preorder, Print-inorder, and Print-postorder, so that we can check the trees after insertion and deletion operations.

#include <stdio.h>

#include <stdlib.h>

// Definition of a node in the BST

typedef struct Node {

int data;

struct Node \*left;

struct Node \*right;

} Node;

// Function prototypes

Node\* createNode(int data);

Node\* insert(Node \*root, int data);

Node\* deleteNode(Node \*root, int data);

void printPreOrder(Node \*root);

void printInOrder(Node \*root);

void printPostOrder(Node \*root);

Node\* findMin(Node \*root);

Node\* createNode(int data) {

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

newNode->data = data;

newNode->left = NULL;

newNode->right = NULL;

return newNode;

}

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

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);

}

return root;

}

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

if (root == NULL) {

return NULL;

}

if (data < root->data) {

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

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

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

} else {

// Node with only one child or no child

if (root->left == NULL) {

Node \*temp = root->right;

free(root);

return temp;

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

Node \*temp = root->left;

free(root);

return temp;

}

/\*Node with two children: Get the inorder successor (smallest in the right subtree)[approach taken here] or inorder predecessdor\*/

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

root->data = temp->data;

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

}

return root;

}

Node\* findMin(Node \*root) {

Node \*current = root;

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

current = current->left;

}

return current;

}

void printPreOrder(Node \*root) {

if (root == NULL) {

return;

}

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

printPreOrder(root->left);

printPreOrder(root->right);

}

void printInOrder(Node \*root) {

if (root == NULL) {

return;

}

printInOrder(root->left);

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

printInOrder(root->right);

}

void printPostOrder(Node \*root) {

if (root == NULL) {

return;

}

printPostOrder(root->left);

printPostOrder(root->right);

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

}

3.6 Define an ADT for Graph.

a) Implement Kruskal’s algorithm for finding the Minimun-Cost Spanning Tree of a Graph with positive edge-weights.

b) Implement Dijkstra’s algorithm for finding the shortest paths from a start node to all the other nodes in a graph with positive edge-weights.

c) Write a program to find out the Transitive Closure Matrix for a directed graph

#include <stdio.h>

#include <stdlib.h>

typedef struct Graph {

int vertices; // Number of vertices

int \*\*adjMatrix; // Adjacency matrix for edge weights

} Graph;

// Function prototypes

Graph\* createGraph(int vertices);

void addEdge(Graph \*graph, int src, int dest, int weight);

void displayGraph(Graph \*graph);

// Structure to represent an edge

typedef struct Edge {

int src, dest, weight;

} Edge;

// Structure for subset (used in union-find)

typedef struct Subset {

int parent, rank;

} Subset;

// Find the root of a subset (union-find)

int find(Subset subsets[], int i) {

if (subsets[i].parent != i) {

subsets[i].parent = find(subsets, subsets[i].parent);

}

return subsets[i].parent;

}

// Union of two subsets (union-find)

void unionSets(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++;

}

}

// Compare function for qsort (sorting edges by weight)

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

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

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

return e1->weight - e2->weight;

}

// Kruskal's Algorithm

void kruskalMST(int vertices, int edgesCount, Edge edges[]) {

// Sort edges by weight

qsort(edges, edgesCount, sizeof(Edge), compareEdges);

// Create subsets for union-find

Subset \*subsets = (Subset \*)malloc(vertices \* sizeof(Subset));

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

subsets[i].parent = i;

subsets[i].rank = 0;

}

printf("Edges in the Minimum Spanning Tree:\n");

int mstEdges = 0, i = 0;

while (mstEdges < vertices - 1 && i < edgesCount) {

Edge nextEdge = edges[i++];

int x = find(subsets, nextEdge.src);

int y = find(subsets, nextEdge.dest);

if (x != y) {

printf("(%d -- %d) Weight: %d\n", nextEdge.src, nextEdge.dest, nextEdge.weight);

unionSets(subsets, x, y);

mstEdges++;

}

}

free(subsets);

}

#define INF 1000000

void dijkstra(Graph \*graph, int start) {

int vertices = graph->vertices;

int \*dist = (int \*)malloc(vertices \* sizeof(int));

int \*visited = (int \*)malloc(vertices \* sizeof(int));

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

dist[i] = INF;

visited[i] = 0;

}

dist[start] = 0;

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

// Find the vertex with the minimum distance

int min = INF, u = -1;

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

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

min = dist[v];

u = v;

}

}

visited[u] = 1;

// Update distances of adjacent vertices

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

if (!visited[v] && graph->adjMatrix[u][v] && dist[u] != INF &&

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

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

}

}

}

printf("Vertex\tDistance from Source\n");

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

printf("%d\t%d\n", i, dist[i]);

}

free(dist);

free(visited);

}

void transitiveClosure(Graph \*graph) {

int vertices = graph->vertices;

int \*\*closure = (int \*\*)malloc(vertices \* sizeof(int \*));

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

closure[i] = (int \*)malloc(vertices \* sizeof(int));

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

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

}

}

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

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

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

closure[i][j] = closure[i][j] || (closure[i][k] && closure[k][j]);

}

}

}

printf("Transitive Closure Matrix:\n");

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

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

printf("%d ", closure[i][j]);

}

printf("\n");

}

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

free(closure[i]);

}

free(closure);

}