#include <stdio.h>

unsigned long long factorial(int n) {

unsigned long long fact = 1;

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

fact \*= i;

}

return fact;

}

int main() {

int num;

printf("Enter a number: ");

scanf("%d", &num);

if (num < 0) {

printf("Factorial of a negative number doesn't exist.\n");

} else {

printf("Factorial of %d = %llu\n", num, factorial(num));

}

return 0;

}

#avl tree

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int key;

struct Node \*left;

struct Node \*right;

int height;

} Node;

int height(Node \*node) {

if (node == NULL)

return 0;

return node->height;

}

int max(int a, int b) {

return (a > b) ? a : b;

}

Node \*newNode(int key) {

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

node->key = key;

node->left = NULL;

node->right = NULL;

node->height = 1;

return node;

}

Node \*rightRotate(Node \*y) {

Node \*x = y->left;

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;

}

Node \*leftRotate(Node \*x) {

Node \*y = x->right;

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(Node \*node) {

if (node == NULL)

return 0;

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

}

Node \*insert(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;

}

Node minValueNode(Node node) {

Node\* current = node;

while (current->left != NULL)

current = current->left;

return current;

}

Node \*deleteNode(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)) {

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

if (temp == NULL) {

temp = root;

root = NULL;

} else

\*root = \*temp;

free(temp);

} else {

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;

}

void PrintTree(Node\* root, int space) {

int count = 10;

if (root == NULL)

return;

space += count;

PrintTree(root->right, space);

printf("\n");

for (int i = count; i < space; i++)

printf(" ");

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

PrintTree(root->left, space);

}

int main() {

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("Tree structure:\n");

PrintTree(root, 0);

root = deleteNode(root, 40);

printf("\nTree structure after deletion:\n");

PrintTree(root, 0);

    return 0;

}

#valid stack

#include <stdio.h>

#include <stdlib.h>

#define MAX 100 // Maximum size of the stack

typedef struct {

int items[MAX];

int top;

} Stack;

void initStack(Stack \*s) {

s->top = -1;

}

int isFull(Stack \*s) {

return s->top == MAX - 1;

}

int isEmpty(Stack \*s) {

return s->top == -1;

}

void push(Stack \*s, int newItem) {

if (isFull(s)) {

printf("Stack is full. Cannot push %d\n", newItem);

} else {

s->items[++(s->top)] = newItem;

printf("%d pushed to stack\n", newItem);

}

}

int pop(Stack \*s) {

if (isEmpty(s)) {

printf("Stack is empty. Cannot pop\n");

return -1;

} else {

return s->items[(s->top)--];

}

}

void display(Stack \*s) {

if (isEmpty(s)) {

printf("Stack is empty\n");

} else {

printf("Stack elements: ");

for (int i = 0; i <= s->top; i++) {

printf("%d ", s->items[i]);

}

printf("\n");

}

}

int main() {

Stack s;

initStack(&s);

push(&s, 10);

push(&s, 20);

push(&s, 30);

display(&s);

printf("Popped element: %d\n", pop(&s));

printf("Popped element: %d\n", pop(&s));

display(&s);

return 0;

}

#graph\_shortest path

#include <stdio.h>

#include <limits.h>

#include <stdbool.h>

#define V 9 // Number of vertices in the graph

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[], int n) {

printf("Vertex \t\t Distance from Source\n");

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

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

}

void dijkstra(int graph[V][V], int src) {

int dist[V]; // The output array. dist[i] will hold the shortest distance from src to i

bool sptSet[V]; // sptSet[i] will be true if vertex i is included in the shortest path tree

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, V);

}

int main() {

int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},

{4, 0, 8, 0, 0, 0, 0, 11, 0},

{0, 8, 0, 7, 0, 4, 0, 0, 2},

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

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

{0, 0, 4, 14, 10, 0, 2, 0, 0},

{0, 0, 0, 0, 0, 2, 0, 1, 6},

{8, 11, 0, 0, 0, 0, 1, 0, 7},

{0, 0, 2, 0, 0, 0, 6, 7, 0}};

dijkstra(graph, 0);

return 0;

}

#travelling salesman problem

#include <stdio.h>

#include <limits.h>

#define V 4

#define INF INT\_MAX

int tsp(int graph[V][V], int dp[1 << V][V], int mask, int pos) {

if (mask == (1 << V) - 1) {

return graph[pos][0];

}

if (dp[mask][pos] != -1) {

return dp[mask][pos];

}

int ans = INF;

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

if ((mask & (1 << city)) == 0) {

int newAns = graph[pos][city] + tsp(graph, dp, mask | (1 << city), city);

if (newAns < ans) {

ans = newAns;

}

}

}

return dp[mask][pos] = ans;

}

int main() {

int graph[V][V] = {

{0, 10, 15, 20},

{10, 0, 35, 25},

{15, 35, 0, 30},

{20, 25, 30, 0}

};

int dp[1 << V][V];

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

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

dp[i][j] = -1;

}

}

int result = tsp(graph, dp, 1, 0);

printf("The minimum cost of visiting all cities is %d\n", result);

return 0;

}

#binary search tree\_search for a element ,min element and max element

#include <stdio.h>

#include <stdlib.h>

// Definition of a BST node

struct Node {

int data;

struct Node\* left;

struct Node\* right;

};

// Function to create a new BST node

struct Node\* newNode(int item) {

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

temp->data = item;

temp->left = temp->right = NULL;

return temp;

}

// Function to insert a new node with given key in BST

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

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

if (data < node->data)

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

else if (data > node->data)

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

return node;

}

// Function to search for a given key in a BST

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

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

return root;

if (root->data < key)

return search(root->right, key);

return search(root->left, key);

}

// Function to find the node with the minimum value in a BST

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

struct Node\* current = node;

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

current = current->left;

return current;

}

// Function to find the node with the maximum value in a BST

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

struct Node\* current = node;

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

current = current->right;

return current;

}

// Inorder traversal of the BST

void inorder(struct Node\* root) {

if (root != NULL) {

inorder(root->left);

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

inorder(root->right);

}

}

int main() {

struct Node\* root = NULL;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

printf("Inorder traversal of the BST: ");

inorder(root);

printf("\n");

int key = 40;

if (search(root, key) != NULL)

printf("Element %d found in the BST.\n", key);

else

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

struct Node\* minNode = findMin(root);

if (minNode != NULL)

printf("Minimum element in the BST is %d.\n", minNode->data);

struct Node\* maxNode = findMax(root);

if (maxNode != NULL)

printf("Maximum element in the BST is %d.\n", maxNode->data);

return 0;

}