1. **Write a program in C to read n number of values in an array and display them in reverse order.**

**Input:2 5 7**

**Output :752**

#include <stdio.h>

int main() {

int n;

printf("Input the number of elements to store in the array: ");

scanf("%d", &n);

int arr[n]; // Declare the array to hold n elements

printf("Input %d number of elements in the array:\n", n);

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

printf("element - %d : ", i);

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

}

printf("The values stored into the array are:\n");

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

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

}

printf("\n");

printf("The values stored into the array in reverse are:\n");

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

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

}

printf("\n");

return 0;

}

**2. Implement a C Program for AVL tree and perform Insertion and Deletion of Nodes**

#include <stdio.h>

#include <stdlib.h>

typedef struct AVLNode {

int key;

struct AVLNode \*left;

struct AVLNode \*right;

int height;

} AVLNode;

int height(AVLNode \*node) {

if (node == NULL) return 0;

return node->height;

}

int max(int a, int b) {

return (a > b) ? a : b;

}

AVLNode\* createNode(int key) {

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

node->key = key;

node->left = NULL;

node->right = NULL;

node->height = 1; // New node is initially at height 1

return node;

}

AVLNode\* rightRotate(AVLNode \*y) {

AVLNode \*x = y->left;

AVLNode \*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;

}

AVLNode\* leftRotate(AVLNode \*x) {

AVLNode \*y = x->right;

AVLNode \*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(AVLNode \*node) {

if (node == NULL) return 0;

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

}

AVLNode\* insert(AVLNode\* node, int key) {

if (node == NULL) return createNode(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;

}

AVLNode\* minNode(AVLNode\* node) {

AVLNode\* current = node;

while (current->left != NULL)

current = current->left;

return current;

}

AVLNode\* deleteNode(AVLNode\* root, int key) {

// STEP 1: PERFORM STANDARD BST DELETE

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)) {

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

if (temp == NULL) {

temp = root;

root = NULL;

} else // One child case

\*root = \*temp; // Copy the contents of the non-empty child

free(temp);

} else {

// Node with two children: Get the inorder successor (smallest

// in the right subtree)

AVLNode\* temp = minNode(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 inOrder(AVLNode \*root) {

if (root != NULL) {

inOrder(root->left);

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

inOrder(root->right);

}

}

int main() {

AVLNode \*root = NULL;

// Inserting nodes

root = insert(root, 10);

root = insert(root, 20);

root = insert(root, 30);

root = insert(root, 15);

printf("In-order traversal of the AVL tree: ");

inOrder(root);

printf("\n");

root = deleteNode(root, 20);

printf("In-order traversal after deletion : ");

inOrder(root);

printf("\n");

return 0;

}

**3.Implement a C Program to Check for a valid String**

PROGRAM:

#include <stdio.h>

#include <ctype.h>

int isValidString(const char \*str) {

if (str == NULL || \*str == '\0') {

return 0;

}

while (\*str) {

if (!isprint((unsigned char)\*str) && !isspace((unsigned char)\*str)) {

return 0;

}

str++;

}

return 1;

}

int main() {

char str[256];

printf("Enter a string: ");

fgets(str, sizeof(str), stdin);

size\_t len = strlen(str);

if (len > 0 && str[len - 1] == '\n') {

str[len - 1] = '\0';

}

if (isValidString(str)) {

printf("The string is valid.\n");

} else {

printf("The string is invalid.\n");

}

return 0;

}

**4. Implement a C Program whether it is a Valid stack**

**Input: pushed = { 1, 2, 3, 4, 5 }, popped = { 4, 5, 3, 2, 1 }**

**Output: True**

**Following sequence can be performed:**

**push(1), push(2), push(3), push(4), pop() -> 4,**

**push(5), pop() -> 5, pop() -> 3, pop() -> 2, pop() -> 1**

**Input: pushed = { 1, 2, 3, 4, 5 }, popped = { 4, 3, 5, 1, 2 }**

**Output: False**

**1 can't be popped before 2.**

#include <stdio.h>

#include <stdbool.h>

#define MAX 100 // Maximum size of stack

// Function to check if the popped sequence is valid

bool isValidStackSequence(int pushed[], int popped[], int n) {

int stack[MAX];

int top = -1; // Initialize stack pointer

int popIndex = 0; // Index for the popped array

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

// Push current element to stack

stack[++top] = pushed[i];

// Check if the top of the stack matches the current element of popped

while (top >= 0 && stack[top] == popped[popIndex]) {

top--; // Pop the element

popIndex++; // Move to the next element in popped

}

}

// If we have processed all elements of popped, the sequence is valid

return popIndex == n;

}

int main() {

int pushed[] = {1, 2, 3, 4, 5};

int popped1[] = {4, 5, 3, 2, 1};

int popped2[] = {4, 3, 5, 1, 2};

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

if (isValidStackSequence(pushed, popped1, n)) {

printf("Output: True\n");

} else {

printf("Output: False\n");

}

if (isValidStackSequence(pushed, popped2, n)) {

printf("Output: True\n");

} else {

printf("Output: False\n");

}

return 0;

}

**5.Implement a C Program to Merge two Arrays**

#include <stdio.h>

void mergeArrays(int arr1[], int size1, int arr2[], int size2, int merged[]) {

int i, j;

for (i = 0; i < size1; i++) {

merged[i] = arr1[i];

}

for (j = 0; j < size2; j++) {

merged[i + j] = arr2[j];

}

}

void displayArray(int arr[], int size) {

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

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

}

printf("\n");

}

int main() {

int size1, size2;

printf("Enter the number of elements in the first array: ");

scanf("%d", &size1);

int arr1[size1];

printf("Enter the elements of the first array:\n");

for (int i = 0; i < size1; i++) {

printf("Element %d: ", i);

scanf("%d", &arr1[i]);

}

printf("Enter the number of elements in the second array: ");

scanf("%d", &size2);

int arr2[size2];

printf("Enter the elements of the second array:\n");

for (int i = 0; i < size2; i++) {

printf("Element %d: ", i);

scanf("%d", &arr2[i]);

}

int merged[size1 + size2];

mergeArrays(arr1, size1, arr2, size2, merged);

printf("Merged array:\n");

displayArray(merged, size1 + size2);

return 0;

}

**6. Implement a C Program for Graph to Identify shortest path**

#include <stdio.h>

#include <limits.h>

#include <stdbool.h>

#define MAX\_NODES 100

#define INF INT\_MAX

int minDistance(int dist[], bool sptSet[], int n) {

int min = INF, min\_index;

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

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

min = dist[v];

min\_index = v;

}

}

return min\_index;

}

void printPath(int parent[], int j) {

if (parent[j] == -1) {

printf("%d", j + 1);

return;

}

printPath(parent, parent[j]);

printf(" to %d", j + 1);

}

void dijkstra(int graph[MAX\_NODES][MAX\_NODES], int src, int target, int n) {

int dist[MAX\_NODES];

bool sptSet[MAX\_NODES];

int parent[MAX\_NODES];

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

dist[i] = INF;

sptSet[i] = false;

parent[i] = -1;

}

dist[src] = 0;

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

int u = minDistance(dist, sptSet, n);

sptSet[u] = true;

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

if (!sptSet[v] && graph[u][v] && dist[u] != INF && dist[u] + graph[u][v] < dist[v]) {

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

parent[v] = u;

}

}

}

printf("Shortest path from %d to %d:\n", src + 1, target + 1);

if (dist[target] == INF) {

printf("No path exists.\n");

} else {

printf("Distance: %d\n", dist[target]);

printf("Path: ");

printPath(parent, target);

printf("\n");

}

}

int main() {

int graph[MAX\_NODES][MAX\_NODES];

int n, src, target;

printf("Enter number of nodes: ");

scanf("%d", &n);

printf("Enter weight of all the paths in adjacency matrix form:\n");

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

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

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

}

}

printf("Enter the source: ");

scanf("%d", &src);

printf("Enter the target: ");

scanf("%d", &target);

src -= 1;

target -= 1;

dijkstra(graph, src, target, n);

return 0;

}

**7.Write a program in C to count the total number of duplicate elements in an array.**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

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

int freq[MAX] = {0};

int count = 0;

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

if (arr[i] >= 0 && arr[i] < MAX) {

freq[arr[i]]++;

}

}

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

if (freq[i] > 1) {

count++;

}

}

printf("Total number of duplicate elements: %d\n", count);

}

int main() {

int n;

int arr[MAX];

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

scanf("%d", &n);

printf("Enter the elements of the array:\n");

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

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

}

countDuplicates(arr, n);

return 0;

}

**8. Implement a C Program Traveling Salesman Problem to Identify shortest path**

**Given a set of cities and distances between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.**

#include <stdio.h>

#include <limits.h>

#define MAX\_CITIES 10

int n; // Number of cities

int dist[MAX\_CITIES][MAX\_CITIES]; // Distance matrix

int visited[MAX\_CITIES];

int minPath = INT\_MAX;

int finalPath[MAX\_CITIES + 1];

void tsp(int currPos, int count, int cost, int path[]) {

if (count == n && dist[currPos][0]) {

if (cost + dist[currPos][0] < minPath) {

minPath = cost + dist[currPos][0];

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

finalPath[i] = path[i];

}

finalPath[n] = 0;

}

return;

}

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

if (!visited[i] && dist[currPos][i]) {

visited[i] = 1;

path[count] = i;

tsp(i, count + 1, cost + dist[currPos][i], path);

visited[i] = 0;

}

}

}

int main() {

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

scanf("%d", &n);

printf("Enter the distance matrix:\n");

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

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

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

}

}

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

visited[i] = 0;

}

int path[MAX\_CITIES];

visited[0] = 1;

path[0] = 0;

tsp(0, 1, 0, path);

printf("Minimum cost: %d\n", minPath);

printf("Path: ");

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

printf("%d ", finalPath[i] + 1);

}

printf("\n");

return 0;

}

**9. Implement a C Program for Merging of list**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node\* next;

} Node;

// Function to create a new node

Node\* createNode(int data) {

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

newNode->data = data;

newNode->next = NULL;

return newNode;

}

// Function to print the linked list

void printList(Node\* head) {

Node\* temp = head;

while (temp != NULL) {

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

temp = temp->next;

}

printf("NULL\n");

}

// Function to merge two sorted linked lists

Node\* mergeLists(Node\* l1, Node\* l2) {

Node dummy;

Node\* tail = &dummy;

dummy.next = NULL;

while (l1 != NULL && l2 != NULL) {

if (l1->data <= l2->data) {

tail->next = l1;

l1 = l1->next;

} else {

tail->next = l2;

l2 = l2->next;

}

tail = tail->next;

}

if (l1 != NULL) {

tail->next = l1;

} else {

tail->next = l2;

}

return dummy.next;

}

int main() {

// Creating first sorted linked list: 1 -> 3 -> 5 -> NULL

Node\* l1 = createNode(1);

l1->next = createNode(3);

l1->next->next = createNode(5);

// Creating second sorted linked list: 2 -> 4 -> 6 -> NULL

Node\* l2 = createNode(2);

l2->next = createNode(4);

l2->next->next = createNode(6);

printf("First sorted list: ");

printList(l1);

printf("Second sorted list: ");

printList(l2);

Node\* mergedList = mergeLists(l1, l2);

printf("Merged sorted list: ");

printList(mergedList);

// Free memory (not shown here but should be implemented for complete program)

return 0;

}

**10. Implement a C Program for Binary search tree - search for a element, min element and Max element**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node\* left;

struct Node\* right;

} Node;

// Function to create a new node

Node\* createNode(int data) {

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

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

// Function to insert a new node into the BST

Node\* insert(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;

}

// Function to search for a value in the BST

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

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

return root;

}

if (key < root->data) {

return search(root->left, key);

} else {

return search(root->right, key);

}

}

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

Node\* findMin(Node\* root) {

while (root->left != NULL) {

root = root->left;

}

return root;

}

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

Node\* findMax(Node\* root) {

while (root->right != NULL) {

root = root->right;

}

return root;

}

// Function to print the BST in-order (for checking correctness)

void inorder(Node\* root) {

if (root != NULL) {

inorder(root->left);

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

inorder(root->right);

}

}

int main() {

Node\* root = NULL;

// Insert nodes into the BST

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

// Print the BST in-order

printf("In-order traversal of the BST:\n");

inorder(root);

printf("\n");

// Search for an element

int key;

printf("Enter the element to search: ");

scanf("%d", &key);

Node\* searchResult = search(root, key);

if (searchResult != NULL) {

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

} else {

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

}

// Find the minimum and maximum elements

Node\* minNode = findMin(root);

Node\* maxNode = findMax(root);

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

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

return 0;

}

**11. Implement a C Program Given an array of reg nos need to search for particular reg no**

#include <stdio.h>

// Function to perform linear search

int linearSearch(int arr[], int size, int key) {

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

if (arr[i] == key) {

return i; // Return the index if found

}

}

return -1; // Return -1 if not found

}

int main() {

int n, regNo, result;

// Input the number of registration numbers

printf("Enter the number of registration numbers: ");

scanf("%d", &n);

int regNos[n]; // Array to store registration numbers

// Input the registration numbers

printf("Enter the registration numbers:\n");

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

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

}

// Input the registration number to search

printf("Enter the registration number to search: ");

scanf("%d", &regNo);

// Perform linear search

result = linearSearch(regNos, n, regNo);

// Output the result

if (result != -1) {

printf("Registration number %d found at index %d.\n", regNo, result);

} else {

printf("Registration number %d not found.\n", regNo);

}

return 0;

}

**12.Implement a C Program for Haystack. There are two strings needle and haystack (or hay). You need to check if all the characters in the needle are present in haystack or not. If yes then return True (1) or False (0).**

#include <stdio.h>

#include <stdbool.h>

#define CHAR\_SET\_SIZE 256 // Size of the character set (ASCII)

// Function to count the frequency of characters in a string

void countFrequency(char \*str, int freq[]) {

for (int i = 0; str[i] != '\0'; i++) {

freq[(unsigned char)str[i]]++;

}

}

// Function to check if all characters in needle are present in haystack

bool areAllCharsPresent(char \*needle, char \*haystack) {

int needleFreq[CHAR\_SET\_SIZE] = {0};

int haystackFreq[CHAR\_SET\_SIZE] = {0};

// Count frequency of characters in needle and haystack

countFrequency(needle, needleFreq);

countFrequency(haystack, haystackFreq);

// Check if all characters in needle are present in haystack

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

if (needleFreq[i] > 0 && haystackFreq[i] < needleFreq[i]) {

return false;

}

}

return true;

}

int main() {

char needle[100], haystack[100];

// Input the needle and haystack strings

printf("Enter the needle string: ");

scanf("%s", needle);

printf("Enter the haystack string: ");

scanf("%s", haystack);

// Check if all characters in needle are present in haystack

if (areAllCharsPresent(needle, haystack)) {

printf("True\n");

} else {

printf("False\n");

}

return 0;

}

**13. Write a program in C to count the frequency of each element of an array.**

#include <stdio.h>

#define MAX\_SIZE 100

void countFrequency(int arr[], int n)

{

int visited[MAX\_SIZE] = {0}; // Array to keep track of visited elements

printf("The frequency of all elements of the array:\n");

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

if (visited[i] == 1) {

continue; // Skip the element if it has been visited

}

int count = 1; // Initialize count for the current element

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

if (arr[j] == arr[i]) {

count++;

visited[j] = 1; // Mark the element as visited

}

}

printf("%d occurs %d times\n", arr[i], count);

}

}

int main() {

int n, arr[MAX\_SIZE];

// Input the number of elements

printf("Input the number of elements to be stored in the array: ");

scanf("%d", &n);

// Input the elements

printf("Input %d elements in the array:\n", n);

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

printf("element - %d : ", i);

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

}

// Count the frequency of each element

countFrequency(arr, n);

return 0;

}

14. Implement a C Program for Given Graph convert array and print minimum edges (Prim’s Algorithm)

#include <stdio.h>

#include <limits.h>

#include <stdbool.h>

#define MAX\_NODES 100

#define INF INT\_MAX

// Function to find the vertex with the minimum key value that is not yet included in the MST

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

int min = INF, min\_index;

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

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

min = key[v];

min\_index = v;

}

}

return min\_index;

}

// Function to implement Prim's Algorithm

void primMST(int graph[MAX\_NODES][MAX\_NODES], int n) {

int parent[MAX\_NODES]; // Array to store the MST

int key[MAX\_NODES]; // Key values used to pick minimum weight edge

bool mstSet[MAX\_NODES]; // Boolean array to check if vertex is included in MST

// Initialize all keys as INFINITE

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

key[i] = INF;

mstSet[i] = false;

}

// Always include the first vertex in the MST

key[0] = 0; // Make key 0 so that this vertex is picked first

parent[0] = -1; // First node is always the root of the MST

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

// Pick the minimum key vertex from the set of vertices not yet included in MST

int u = minKey(key, mstSet, n);

mstSet[u] = true;

// Update key value and parent index of the adjacent vertices of the picked vertex

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

// Update the key only if graph[u][v] is smaller than key[v]

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

key[v] = graph[u][v];

parent[v] = u;

}

}

}

// Print the constructed MST

printf("Edge \tWeight\n");

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

printf("%d - %d \t%d \n", parent[i] + 1, i + 1, graph[i][parent[i]]);

}

}

int main() {

int graph[MAX\_NODES][MAX\_NODES];

int n;

// Input the number of nodes

printf("Enter number of nodes: ");

scanf("%d", &n);

// Input the adjacency matrix

printf("Enter weight of all the paths in adjacency matrix form:\n");

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

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

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

if (graph[i][j] == 0) {

graph[i][j] = INF; // Use INF for no edge

}

}

}

// Run Prim's Algorithm and print the MST

primMST(graph, n);

return 0;

}

**15. Write a program in C to separate odd and even integers into separate arrays.**

#include <stdio.h>

#define MAX\_SIZE 100

void separateOddEven(int arr[], int size, int oddArr[], int \*oddCount, int evenArr[], int \*evenCount) {

\*oddCount = 0;

\*evenCount = 0;

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

if (arr[i] % 2 == 0) {

evenArr[\*evenCount] = arr[i];

(\*evenCount)++;

} else {

oddArr[\*oddCount] = arr[i];

(\*oddCount)++;

}

}

}

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

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

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

}

printf("\n");

}

int main() {

int arr[MAX\_SIZE];

int oddArr[MAX\_SIZE];

int evenArr[MAX\_SIZE];

int size, oddCount, evenCount;

// Input the size of the array

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

scanf("%d", &size);

// Input the elements of the array

printf("Enter the elements:\n");

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

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

}

// Separate odd and even integers

separateOddEven(arr, size, oddArr, &oddCount, evenArr, &evenCount);

// Print the results

printf("Odd numbers:\n");

printArray(oddArr, oddCount);

printf("Even numbers:\n");

printArray(evenArr, evenCount);

return 0;

}

**16.** **Implement a C Program for Given Graph - Print valid path (BFS or DFS)**

//BINARY SEARCH TREE FOR FINDING MINIMUM AND MAXIMUM

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node\* left;

struct Node\* right;

} Node;

// Function to create a new node

Node\* createNode(int data) {

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

newNode->data = data;

newNode->left = newNode->right = NULL;

return newNode;

}

// Function to insert a new node into the BST

Node\* insert(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;

}

// Function to search for a value in the BST

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

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

return root;

}

if (key < root->data) {

return search(root->left, key);

} else {

return search(root->right, key);

}

}

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

Node\* findMin(Node\* root) {

while (root->left != NULL) {

root = root->left;

}

return root;

}

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

Node\* findMax(Node\* root) {

while (root->right != NULL) {

root = root->right;

}

return root;

}

// Function to print the BST in-order (for checking correctness)

void inorder(Node\* root) {

if (root != NULL) {

inorder(root->left);

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

inorder(root->right);

}

}

int main() {

Node\* root = NULL;

// Insert nodes into the BST

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

// Print the BST in-order

printf("In-order traversal of the BST:\n");

inorder(root);

printf("\n");

// Search for an element

int key;

printf("Enter the element to search: ");

scanf("%d", &key);

Node\* searchResult = search(root, key);

if (searchResult != NULL) {

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

} else {

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

}

// Find the minimum and maximum elements

Node\* minNode = findMin(root);

Node\* maxNode = findMax(root);

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

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

return 0;

}

**17. Implement a C Program sum of Fibonacci Series using recursion**

#include <stdio.h>

// Function to calculate the nth Fibonacci number using recursion

int fibonacci(int n) {

if (n <= 1) {

return n;

}

return fibonacci(n - 1) + fibonacci(n - 2);

}

// Function to compute the sum of the first n Fibonacci numbers

int sumFibonacci(int n) {

if (n <= 0) {

return 0;

}

return fibonacci(n - 1) + sumFibonacci(n - 1);

}

int main() {

int n;

// Input the number of terms

printf("Enter the number of Fibonacci terms to sum: ");

scanf("%d", &n);

if (n < 1) {

printf("Number of terms should be a positive integer.\n");

return 1;

}

// Compute the sum of the first n Fibonacci numbers

int sum = sumFibonacci(n);

// Output the result

printf("The sum of the first %d Fibonacci numbers is: %d\n", n, sum);

return 0;

}

**18. Implement a C Program to perform heap sort**

#include <stdio.h>

// Function to swap two elements

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to heapify a subtree rooted at index i

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

int largest = i; // Initialize largest as root

int left = 2 \* i + 1; // left = 2\*i + 1

int right = 2 \* i + 2; // right = 2\*i + 2

// Check if left child is larger than root

if (left < n && arr[left] > arr[largest]) {

largest = left;

}

// Check if right child is larger than largest so far

if (right < n && arr[right] > arr[largest]) {

largest = right;

}

// If largest is not root

if (largest != i) {

swap(&arr[i], &arr[largest]);

// Recursively heapify the affected subtree

heapify(arr, n, largest);

}

}

// Function to perform heap sort

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

// Build a max heap

for (int i = n / 2 - 1; i >= 0; i--) {

heapify(arr, n, i);

}

// Extract elements one by one from heap

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

// Move current root to end

swap(&arr[0], &arr[i]);

// Call heapify on the reduced heap

heapify(arr, i, 0);

}

}

// Function to print an array

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

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

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

}

printf("\n");

}

int main() {

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

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

printf("Original array:\n");

printArray(arr, n);

heapSort(arr, n);

printf("Sorted array:\n");

printArray(arr, n);

return 0;

}

**19. Implement a C Program for Finding factorial of a number using recursion**

#include <stdio.h>

// Recursive function to calculate factorial

int factorial(int n) {

// Base case: factorial of 0 is 1

if (n == 0) {

return 1;

}

// Recursive case: n \* factorial of (n-1)

else {

return n \* factorial(n - 1);

}

}

int main() {

int num;

// Input a number from the user

printf("Enter a number to find its factorial: ");

scanf("%d", &num);

// Validate input (factorial is not defined for negative numbers)

if (num < 0) {

printf("Factorial is not defined for negative numbers.\n");

} else {

// Calculate factorial

int result = factorial(num);

// Output the result

printf("The factorial of %d is %d.\n", num, result);

}

return 0;

}

**20. Implement a C Program to perform quick sort**

#include <stdio.h>

// Function to swap two elements

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Function to partition the array into two halves

int partition(int arr[], int low, int high) {

int pivot = arr[high]; // Choosing the last element as pivot

int i = (low - 1); // Index of the smaller element

for (int j = low; j < high; j++) {

// If current element is smaller than the pivot

if (arr[j] <= pivot) {

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

// Function to perform quick sort

void quickSort(int arr[], int low, int high) {

if (low < high) {

// Partitioning index

int pi = partition(arr, low, high);

// Recursively sort elements before and after partition

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

// Function to print an array

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

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

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

}

printf("\n");

}

int main() {

int n;

// Input the number of elements

printf("How many elements are you going to enter?: ");

scanf("%d", &n);

int arr[n];

// Input the elements of the array

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

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

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

}

// Perform quick sort

quickSort(arr, 0, n - 1);

// Print the sorted array

printf("Order of Sorted elements:\n");

printArray(arr, n);

return 0;

}