C++ Code Solutions for Data Structures Problems

# 1. Binary Search Tree for Online Bookstore

\*\*Problem Statement:\*\*   
Design a system for an online bookstore to manage its catalog using binary search trees (BSTs). Allow operations like adding new books, deleting books by title or author efficiently, and updating book information. Display post-order traversal.  
  
\*\*Code:\*\*   
#include <iostream>  
#include <string>  
using namespace std;  
  
class Book {  
public:  
 string title, author;  
 int price;  
 Book \*left, \*right;  
 Book(string t, string a, int p) {  
 title = t; author = a; price = p;  
 left = right = NULL;  
 }  
};  
  
Book\* insert(Book\* root, string t, string a, int p) {  
 if (!root) return new Book(t, a, p);  
 if (t < root->title) root->left = insert(root->left, t, a, p);  
 else root->right = insert(root->right, t, a, p);  
 return root;  
}  
  
void postorder(Book\* root) {  
 if (root) {  
 postorder(root->left);  
 postorder(root->right);  
 cout << root->title << " by " << root->author << ", Rs." << root->price << endl;  
 }  
}  
  
int main() {  
 Book\* root = NULL;  
 root = insert(root, "C++", "Bjarne", 500);  
 root = insert(root, "Java", "Gosling", 400);  
 root = insert(root, "Python", "Guido", 450);  
 postorder(root);  
 return 0;  
}

# 2. Threaded Binary Tree with Insertion, Search, Preorder

\*\*Problem Statement:\*\*   
Implement a threaded binary tree with the following operations: insertion of a node, search for a node, and display the tree in preorder.  
  
\*\*Code:\*\*   
#include <iostream>  
using namespace std;  
  
class Node {  
public:  
 int data;  
 Node \*left, \*right;  
 bool lthread, rthread;  
};  
  
Node\* insert(Node\* root, int key) {  
 Node \*ptr = root, \*par = NULL;  
 while (ptr) {  
 if (key == ptr->data) return root;  
 par = ptr;  
 if (key < ptr->data) {  
 if (!ptr->lthread) ptr = ptr->left;  
 else break;  
 } else {  
 if (!ptr->rthread) ptr = ptr->right;  
 else break;  
 }  
 }  
 Node\* tmp = new Node;  
 tmp->data = key;  
 tmp->lthread = tmp->rthread = true;  
 if (!par) {  
 root = tmp;  
 tmp->left = tmp->right = NULL;  
 } else if (key < par->data) {  
 tmp->left = par->left;  
 tmp->right = par;  
 par->lthread = false;  
 par->left = tmp;  
 } else {  
 tmp->right = par->right;  
 tmp->left = par;  
 par->rthread = false;  
 par->right = tmp;  
 }  
 return root;  
}  
  
void preorder(Node\* root) {  
 Node\* ptr = root;  
 while (ptr) {  
 cout << ptr->data << " ";  
 if (!ptr->lthread)  
 ptr = ptr->left;  
 else if (!ptr->rthread)  
 ptr = ptr->right;  
 else {  
 while (ptr && ptr->rthread)  
 ptr = ptr->right;  
 if (ptr) ptr = ptr->right;  
 }  
 }  
}  
  
int main() {  
 Node\* root = NULL;  
 root = insert(root, 20);  
 root = insert(root, 10);  
 root = insert(root, 30);  
 preorder(root);  
 return 0;  
}

# 3. Height Balanced Tree (AVL) with Insert, Search, Display

\*\*Problem Statement:\*\*   
Design a height-balanced tree (AVL Tree) for handling nodes. The operations should include insertion of a node, searching for a node, and displaying the tree.  
  
\*\*Code:\*\*   
#include <iostream>  
using namespace std;  
  
class AVL {  
 public:  
 int data, height;  
 AVL \*left, \*right;  
 AVL(int value) {  
 data = value;  
 height = 1;  
 left = right = NULL;  
 }  
};  
  
int getHeight(AVL\* root) {  
 return root ? root->height : 0;  
}  
  
int getBalance(AVL\* root) {  
 return root ? getHeight(root->left) - getHeight(root->right) : 0;  
}  
  
AVL\* rightRotate(AVL\* y) {  
 AVL\* x = y->left;  
 AVL\* T2 = x->right;  
 x->right = y;  
 y->left = T2;  
 y->height = max(getHeight(y->left), getHeight(y->right)) + 1;  
 x->height = max(getHeight(x->left), getHeight(x->right)) + 1;  
 return x;  
}  
  
AVL\* leftRotate(AVL\* x) {  
 AVL\* y = x->right;  
 AVL\* T2 = y->left;  
 y->left = x;  
 x->right = T2;  
 x->height = max(getHeight(x->left), getHeight(x->right)) + 1;  
 y->height = max(getHeight(y->left), getHeight(y->right)) + 1;  
 return y;  
}  
  
AVL\* insert(AVL\* root, int key) {  
 if (!root) return new AVL(key);  
 if (key < root->data) root->left = insert(root->left, key);  
 else if (key > root->data) root->right = insert(root->right, key);  
 root->height = max(getHeight(root->left), getHeight(root->right)) + 1;  
  
 int balance = getBalance(root);  
  
 if (balance > 1 && key < root->left->data) return rightRotate(root);  
 if (balance < -1 && key > root->right->data) return leftRotate(root);  
 if (balance > 1 && key > root->left->data) {  
 root->left = leftRotate(root->left);  
 return rightRotate(root);  
 }  
 if (balance < -1 && key < root->right->data) {  
 root->right = rightRotate(root->right);  
 return leftRotate(root);  
 }  
 return root;  
}  
  
void inorder(AVL\* root) {  
 if (!root) return;  
 inorder(root->left);  
 cout << root->data << " ";  
 inorder(root->right);  
}  
  
int main() {  
 AVL\* root = NULL;  
 root = insert(root, 20);  
 root = insert(root, 10);  
 root = insert(root, 30);  
 inorder(root);  
 return 0;  
}

# 4. Depth First Traversal for User Defined Graph

\*\*Problem Statement:\*\*   
Write a program to implement Depth First Traversal (DFS) for a graph.  
  
\*\*Code:\*\*   
#include <iostream>  
#include <list>  
using namespace std;  
  
class Graph {  
 int V;  
 list<int>\* adj;  
public:  
 Graph(int V);  
 void addEdge(int v, int w);  
 void DFS(int v);  
};  
  
Graph::Graph(int V) {  
 this->V = V;  
 adj = new list<int>[V];  
}  
  
void Graph::addEdge(int v, int w) {  
 adj[v].push\_back(w);  
}  
  
void Graph::DFS(int v) {  
 bool\* visited = new bool[V];  
 for (int i = 0; i < V; i++) visited[i] = false;  
 list<int> stack;  
 stack.push\_back(v);  
 while (!stack.empty()) {  
 int node = stack.back();  
 stack.pop\_back();  
 if (!visited[node]) {  
 visited[node] = true;  
 cout << node << " ";  
 }  
 for (auto it = adj[node].begin(); it != adj[node].end(); ++it) {  
 if (!visited[\*it]) stack.push\_back(\*it);  
 }  
 }  
}  
  
int main() {  
 Graph g(5);  
 g.addEdge(0, 1);  
 g.addEdge(0, 2);  
 g.addEdge(1, 3);  
 g.addEdge(1, 4);  
 g.DFS(0);  
 return 0;  
}

**5. Breadth First Traversal (Without vectors)**

**Problem Statement:**  
Write a C++ program to implement **Breadth First Search (BFS)** traversal for a user-defined graph using an adjacency matrix and basic arrays (no vectors).

cpp

CopyEdit

#include <iostream>

using namespace std;

int graph[10][10], visited[10], queue[10];

int front = 0, rear = 0, n;

void bfs(int start) {

queue[rear++] = start;

visited[start] = 1;

while (front < rear) {

int v = queue[front++];

cout << v << " ";

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

if (graph[v][i] && !visited[i]) {

queue[rear++] = i;

visited[i] = 1;

}

}

}

}

int main() {

int edges, u, v;

cout << "Enter number of nodes and edges: ";

cin >> n >> edges;

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

cin >> u >> v;

graph[u][v] = graph[v][u] = 1;

}

bfs(0); // start from node 0

return 0;

}

**6. Transportation Network using Graph (Kruskal’s Algorithm)**

**Problem Statement:**  
Design a C++ program to represent a transportation network using graphs. The user can input cities and routes, add more cities/routes, and compute the **Minimum Spanning Tree (MST)** using **Kruskal’s Algorithm**. Use only iostream (no climits, cstring, or vector).

cpp

CopyEdit

#include <iostream>

using namespace std;

#define MAX 10

int parent[MAX], cost[MAX][MAX], n;

int find(int i) {

while (parent[i]) i = parent[i];

return i;

}

int uni(int i, int j) {

if (i != j) {

parent[j] = i;

return 1;

}

return 0;

}

int main() {

int edges, a, b, u, v, ne = 1, min, mincost = 0;

cout << "Enter number of cities: ";

cin >> n;

cout << "Enter cost matrix (0 if no route):\n";

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

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

cin >> cost[i][j];

while (ne < n) {

min = 999;

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

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

if (cost[i][j] < min && cost[i][j] != 0) {

min = cost[i][j]; a = u = i; b = v = j;

}

u = find(u); v = find(v);

if (uni(u, v)) {

cout << ne++ << ". Route " << a << " - " << b << " : " << min << "\n";

mincost += min;

}

cost[a][b] = cost[b][a] = 0;

}

cout << "Total Minimum Distance: " << mincost;

return 0;

}

**7. Optimal Route Planner using Dijkstra’s Algorithm**

**Problem Statement:**  
Create a route planning application using **Dijkstra’s algorithm** to find the shortest path between two cities. Let the user enter the start and end cities. Use only arrays and iostream.

cpp

CopyEdit

#include <iostream>

using namespace std;

#define INF 999, MAX 10

int main() {

int cost[MAX][MAX], dist[MAX], visited[MAX], n, start;

cout << "Enter number of cities: ";

cin >> n;

cout << "Enter cost matrix (0 if no path):\n";

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

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

cin >> cost[i][j];

cout << "Enter starting city: ";

cin >> start;

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

dist[i] = cost[start][i] ? cost[start][i] : 999;

visited[i] = 0;

}

dist[start] = 0;

visited[start] = 1;

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

int u = -1, min = 999;

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

if (!visited[j] && dist[j] < min)

min = dist[j], u = j;

visited[u] = 1;

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

if (!visited[v] && cost[u][v] && dist[u] + cost[u][v] < dist[v])

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

}

cout << "Shortest distances:\n";

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

cout << "To " << i << " = " << dist[i] << "\n";

return 0;

}

**8. Telephone Directory using Hash Table (Linear Probing)**

**Problem Statement:**  
Write a C++ program to create a telephone book using a hash table. Use **linear probing** for collision resolution and allow quick lookup of phone numbers by name.

cpp

CopyEdit

#include <iostream>

using namespace std;

class Entry {

public:

string name;

long phone;

};

int main() {

Entry table[10];

bool filled[10] = {false};

for (int i = 0; i < 5; i++) {

string name; long phone;

cout << "Enter name and phone: ";

cin >> name >> phone;

int index = phone % 10;

while (filled[index])

index = (index + 1) % 10;

table[index].name = name;

table[index].phone = phone;

filled[index] = true;

}

cout << "Enter phone to search: ";

long key; cin >> key;

int index = key % 10, cnt = 0;

while (cnt < 10 && filled[index] && table[index].phone != key) {

index = (index + 1) % 10;

cnt++;

}

if (filled[index] && table[index].phone == key)

cout << "Found: " << table[index].name << "\n";

else

cout << "Not found\n";

return 0;

}

**9. Set Operations (Conference Language Analysis)**

**Problem Statement:**  
You are organizing a conference. Some attendees speak **Spanish**, some speak **German**, and some speak both. Write a C++ program to find:

* Union (All who speak at least one language)
* Intersection (All who speak both)
* Difference (Spanish-only speakers)

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

int sp[10], gr[10], n1, n2;

cout << "Enter number of Spanish speakers: ";

cin >> n1;

cout << "Enter IDs:\n";

for (int i = 0; i < n1; i++) cin >> sp[i];

cout << "Enter number of German speakers: ";

cin >> n2;

cout << "Enter IDs:\n";

for (int i = 0; i < n2; i++) cin >> gr[i];

// Union

cout << "\nUnion: ";

for (int i = 0; i < n1; i++) cout << sp[i] << " ";

for (int i = 0; i < n2; i++) {

bool found = false;

for (int j = 0; j < n1; j++)

if (gr[i] == sp[j]) { found = true; break; }

if (!found) cout << gr[i] << " ";

}

// Intersection

cout << "\nIntersection: ";

for (int i = 0; i < n1; i++)

for (int j = 0; j < n2; j++)

if (sp[i] == gr[j]) cout << sp[i] << " ";

// Difference

cout << "\nSpanish only: ";

for (int i = 0; i < n1; i++) {

bool found = false;

for (int j = 0; j < n2; j++)

if (sp[i] == gr[j]) { found = true; break; }

if (!found) cout << sp[i] << " ";

}

return 0;

}

**10. Optimal Binary Search Tree (OBST) for Compiler Symbol Table**

**Problem Statement:**  
Design a C++ program to implement an **Optimal Binary Search Tree (OBST)** for storing identifiers, constants, and keywords in a compiler symbol table using given probabilities of access.

cpp

CopyEdit

#include <iostream>

using namespace std;

int main() {

int n;

float p[10], q[10], w[10][10], c[10][10];

int r[10][10];

cout << "Enter number of keys: ";

cin >> n;

cout << "Enter probabilities for keys (p): ";

for (int i = 1; i <= n; i++) cin >> p[i];

cout << "Enter probabilities for dummy keys (q): ";

for (int i = 0; i <= n; i++) cin >> q[i];

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

w[i][i] = q[i];

c[i][i] = 0;

r[i][i] = 0;

}

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

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

int j = i + l;

w[i][j] = w[i][j - 1] + p[j] + q[j];

c[i][j] = 999;

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

float t = c[i][k - 1] + c[k][j] + w[i][j];

if (t < c[i][j]) {

c[i][j] = t;

r[i][j] = k;

}

}

}

}

cout << "Minimum cost of OBST: " << c[0][n] << endl;

return 0;

}