Chatgpt full code (adv)

#include <iostream>

#include <queue>

#include <vector>

using namespace std;

/\*

Binary Tree (BT) - level-order insert, BFS search, delete by replacing

with deepest (rightmost) node. Useful when you want a general binary tree

(no ordering property).

\*/

class BinaryTree {

public:

struct Node {

int val;

Node\* left;

Node\* right;

Node(int v) : val(v), left(nullptr), right(nullptr) {}

};

private:

Node\* root;

// helper: delete whole tree (postorder)

void deleteTree(Node\* node) {

if (!node) return;

deleteTree(node->left);

deleteTree(node->right);

delete node;

}

// helper: recursive inorder (just for demonstration)

void inorderRec(Node\* node, vector<int>& out) const {

if (!node) return;

inorderRec(node->left, out);

out.push\_back(node->val);

inorderRec(node->right, out);

}

// helper: delete the deepest node given its pointer

void deleteDeepest(Node\* d\_node) {

if (!root || !d\_node) return;

queue<Node\*> q;

q.push(root);

while (!q.empty()) {

Node\* temp = q.front(); q.pop();

if (temp->left) {

if (temp->left == d\_node) {

delete temp->left;

temp->left = nullptr;

return;

} else q.push(temp->left);

}

if (temp->right) {

if (temp->right == d\_node) {

delete temp->right;

temp->right = nullptr;

return;

} else q.push(temp->right);

}

}

}

public:

BinaryTree() : root(nullptr) {}

~BinaryTree() { deleteTree(root); }

// Level-order insert: first available position (left-to-right)

void insert(int key) {

Node\* newNode = new Node(key);

if (!root) {

root = newNode;

return;

}

queue<Node\*> q;

q.push(root);

while (!q.empty()) {

Node\* temp = q.front(); q.pop();

if (!temp->left) { temp->left = newNode; return; }

else q.push(temp->left);

if (!temp->right) { temp->right = newNode; return; }

else q.push(temp->right);

}

}

// BFS search (returns true if key exists)

bool search(int key) const {

if (!root) return false;

queue<Node\*> q;

q.push(root);

while (!q.empty()) {

Node\* temp = q.front(); q.pop();

if (temp->val == key) return true;

if (temp->left) q.push(temp->left);

if (temp->right) q.push(temp->right);

}

return false;

}

// Delete a node with value 'key' by replacing its value with the deepest node

// and removing the deepest node. If multiple nodes have same value, deletes the

// first encountered in level-order.

void deleteKey(int key) {

if (!root) return;

// single-node case

if (!root->left && !root->right) {

if (root->val == key) { delete root; root = nullptr; }

return;

}

queue<Node\*> q;

q.push(root);

Node\* keyNode = nullptr;

Node\* last = nullptr;

// find node with key (first in level-order) and the last node

while (!q.empty()) {

Node\* temp = q.front(); q.pop();

if (temp->val == key) keyNode = temp;

if (temp->left) q.push(temp->left);

if (temp->right) q.push(temp->right);

last = temp;

}

if (keyNode) {

// replace found node's value with deepest node's value, then delete deepest node

keyNode->val = last->val;

deleteDeepest(last);

}

// if keyNode == nullptr, nothing to delete

}

// Traversals for demonstration:

vector<int> levelOrder() const {

vector<int> out;

if (!root) return out;

queue<Node\*> q;

q.push(root);

while (!q.empty()) {

Node\* t = q.front(); q.pop();

out.push\_back(t->val);

if (t->left) q.push(t->left);

if (t->right) q.push(t->right);

}

return out;

}

vector<int> inorder() const {

vector<int> out;

inorderRec(root, out);

return out;

}

};

/\*

Binary Search Tree (BST) - standard BST operations.

(No self-balancing here; worst-case complexity can be O(n).)

\*/

class BinarySearchTree {

public:

struct Node {

int val;

Node\* left;

Node\* right;

Node(int v) : val(v), left(nullptr), right(nullptr) {}

};

private:

Node\* root;

Node\* insertRec(Node\* node, int key) {

if (!node) return new Node(key);

if (key < node->val) node->left = insertRec(node->left, key);

else if (key > node->val) node->right = insertRec(node->right, key);

// if equal, we ignore duplicate (could choose to allow consistent side)

return node;

}

Node\* minValueNode(Node\* node) {

Node\* curr = node;

while (curr && curr->left) curr = curr->left;

return curr;

}

Node\* removeRec(Node\* node, int key) {

if (!node) return nullptr;

if (key < node->val) node->left = removeRec(node->left, key);

else if (key > node->val) node->right = removeRec(node->right, key);

else {

// node->val == key: three cases

if (!node->left) {

Node\* temp = node->right;

delete node;

return temp;

} else if (!node->right) {

Node\* temp = node->left;

delete node;

return temp;

} else {

// two children: replace with inorder successor (smallest in right subtree)

Node\* succ = minValueNode(node->right);

node->val = succ->val;

node->right = removeRec(node->right, succ->val);

}

}

return node;

}

void deleteTree(Node\* node) {

if (!node) return;

deleteTree(node->left);

deleteTree(node->right);

delete node;

}

void inorderRec(Node\* node, vector<int>& out) const {

if (!node) return;

inorderRec(node->left, out);

out.push\_back(node->val);

inorderRec(node->right, out);

}

public:

BinarySearchTree() : root(nullptr) {}

~BinarySearchTree() { deleteTree(root); }

void insert(int key) { root = insertRec(root, key); }

bool search(int key) const {

Node\* cur = root;

while (cur) {

if (key == cur->val) return true;

if (key < cur->val) cur = cur->left;

else cur = cur->right;

}

return false;

}

void remove(int key) { root = removeRec(root, key); }

vector<int> inorder() const {

vector<int> out;

inorderRec(root, out);

return out;

}

};

// small helper to print vectors neatly

void printVec(const vector<int>& v) {

cout << "[";

for (size\_t i = 0; i < v.size(); ++i) {

cout << v[i];

if (i + 1 < v.size()) cout << ", ";

}

cout << "]\n";

}

int main() {

cout << "=== Binary Tree (BT) demo ===\n";

BinaryTree bt;

for (int v : {1,2,3,4,5,6,7}) bt.insert(v); // level-order fill

cout << "Level-order before delete: "; printVec(bt.levelOrder());

cout << "Inorder (just demonstration): "; printVec(bt.inorder());

cout << "Search 5? " << (bt.search(5) ? "found" : "not found") << "\n";

cout << "Delete key = 3 (BT delete replaces with deepest node)\n";

bt.deleteKey(3);

cout << "Level-order after delete 3: "; printVec(bt.levelOrder());

cout << "\n";

cout << "=== Binary Search Tree (BST) demo ===\n";

BinarySearchTree bst;

for (int v : {50,30,20,40,70,60,80}) bst.insert(v);

cout << "Inorder before deletes (should be sorted): "; printVec(bst.inorder());

cout << "Search 40? " << (bst.search(40) ? "found" : "not found") << "\n";

cout << "Delete 20 (leaf)\n";

bst.remove(20);

cout << "Inorder: "; printVec(bst.inorder());

cout << "Delete 30 (node with one child or updated structure)\n";

bst.remove(30);

cout << "Inorder: "; printVec(bst.inorder());

cout << "Delete 50 (root with two children) -> replaced by inorder successor\n";

bst.remove(50);

cout << "Inorder after removing 50: "; printVec(bst.inorder());

return 0;

}

Node structure for trees 🡪

class Node {

public:

int data;

Node\* left;

Node\* right;

Node(int d) {

this->data = d;

this->left = nullptr;

this->right = nullptr;

}

};

Left and right are called children.

For binary trees, children <= 2

For n-ary trees, children can be more

Basic Terminologies :

* **Node** → Basic unit of a tree containing data and links to children.
* **Root** → The topmost node of the tree.
* **Parent** → A node that has children.
* **Child** → A node directly connected below a parent.
* **Leaf** → A node with no children.
* **Siblings** → Nodes having the same parent.
* **Ancestor** → Any node on the path from a node to the root.
* **Descendant** → Any node that lies below a given node.
* **Degree of Node** → Number of children a node has.
* **Degree of Tree** → Maximum degree of any node in the tree.
* **Depth of Node** → Distance (#edges) from root to that node.
* **Height of Node** → Distance (#edges) from that node to its deepest leaf.
* **Height of Tree** → Height of the root node (longest path root → leaf).
* **Level of Node** → Position from top, root at level 0 (sometimes 1).
* **Subtree** → A tree formed by any node and its descendants.
* **Binary Tree** → A tree where each node has at most 2 children.
* **Balanced Tree** → A tree where heights of subtrees differ by at most 1.

Level Order Traversal is also called BREADTH FIRST SEARCH / BFS

Chatgt

#include <bits/stdc++.h>

using namespace std;

class Node {

public:

int data;

Node\* left;

Node\* right;

Node(int d) {

data = d;

left = NULL;

right = NULL;

}

};

// Build tree recursively (taking input in preorder style)

Node\* buildTree() {

cout << "Enter data (-1 for NULL): ";

int d;

cin >> d;

if (d == -1) return NULL;

Node\* root = new Node(d);

cout << "Enter left child of " << d << endl;

root->left = buildTree();

cout << "Enter right child of " << d << endl;

root->right = buildTree();

return root;

}

// Traversals

void preorder(Node\* root) {

if (root == NULL) return;

cout << root->data << " ";

preorder(root->left);

preorder(root->right);}

void inorder(Node\* root) {

if (root == NULL) return;

inorder(root->left);

cout << root->data << " ";

inorder(root->right);}

void postorder(Node\* root) {

if (root == NULL) return;

postorder(root->left);

postorder(root->right);

cout << root->data << " ";}

void levelOrder(Node\* root) {

if (root == NULL) return;

queue<Node\*> q;

q.push(root);

while (!q.empty()) {

Node\* temp = q.front();

q.pop();

cout << temp->data << " ";

if (temp->left) q.push(temp->left);

if (temp->right) q.push(temp->right);}}

int main() {

Node\* root = buildTree();

cout << "\nPreorder: ";

preorder(root);

cout << "\nInorder: ";

inorder(root);

cout << "\nPostorder: ";

postorder(root);

cout << "\nLevel Order: ";

levelOrder(root);return 0;}

Aha nice catch 😃 you’ve landed on **level order traversal** (also called BFS traversal) of a binary tree in C++. Let me break it down line by line and explain **what this is** and **why it works**:

**🔹 What this code does**

It prints the tree **level by level** from top to bottom, left to right.  
Example: If your tree is

1

/ \

2 3

/ \

4 5

👉 Level order output will be:

1 2 3 4 5

**🔹 Why use a queue<Node\*>**

* A **queue** is FIFO (first-in-first-out).
* BFS requires us to process nodes in the order they appear level by level.
* So, we use queue<Node\*> q; where we push **addresses of nodes** (Node\*), not just data, because:
  + We need to access both left and right children.
  + If you store only int data, you lose the tree structure.

**🔹 Code explanation**

while (!q.empty()) {

Node\* temp = q.front(); // take front element

cout << temp->data << " "; // print its data

q.pop(); // remove from queue

// push left child if it exists

if (temp->left != nullptr) {

q.push(temp->left);

}

// push right child if it exists

if (temp->right != nullptr) {

q.push(temp->right);

}

}

* q.front() → looks at the first node in the queue (current node to process).
* q.pop() → removes it, since we’re done with it.
* Then we push its children (left, right) into the queue.
* This ensures the queue always contains nodes in the **correct level order**.

**🔹 Why no q.full()?**

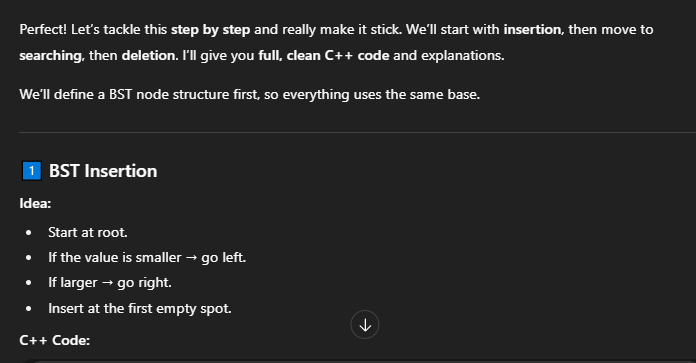
Good question! Unlike q.empty(), there’s **no q.full()** because:

* std::queue in C++ is dynamic, it grows as needed (until memory is exhausted).
* So, it technically never “fills up” under normal usage.

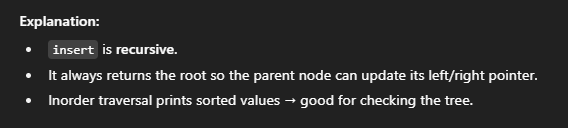
✅ So this snippet is the **core logic of level order traversal (BFS)**.

2 OCTOBER

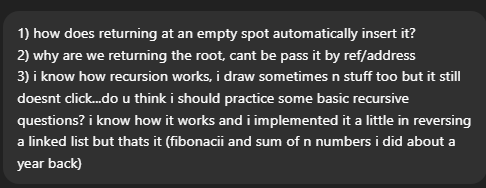
BFS AND DFS IN GRAPH, THIS IS TREES 🡪

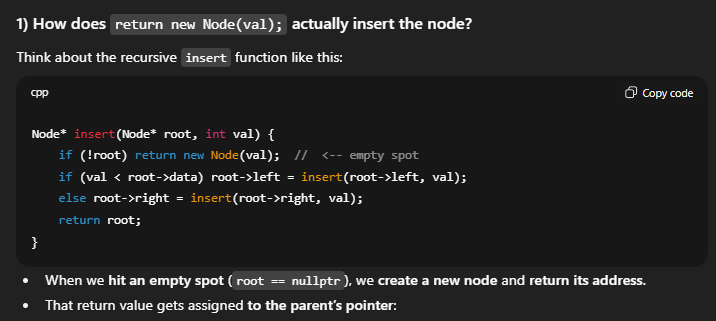


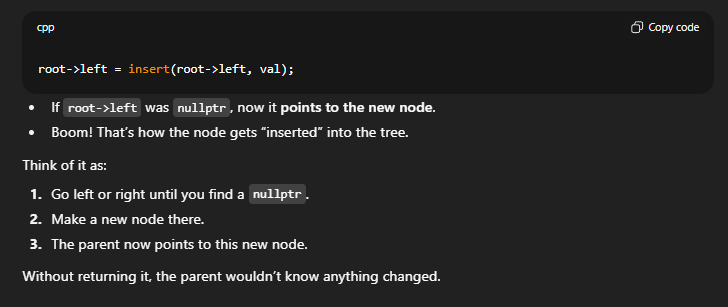
Starting with bst 🡪

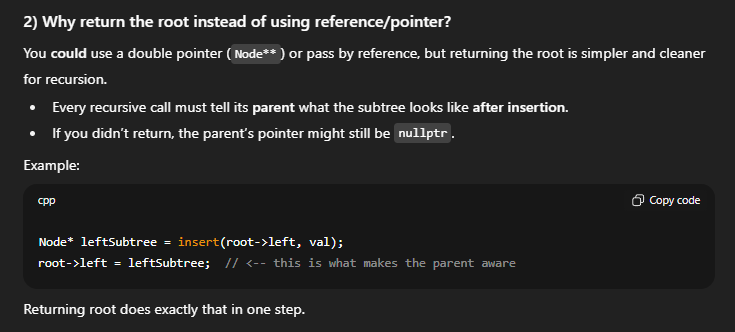


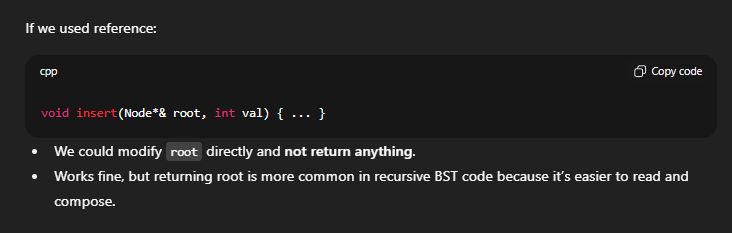
SAARE DOUBTS ----🡪

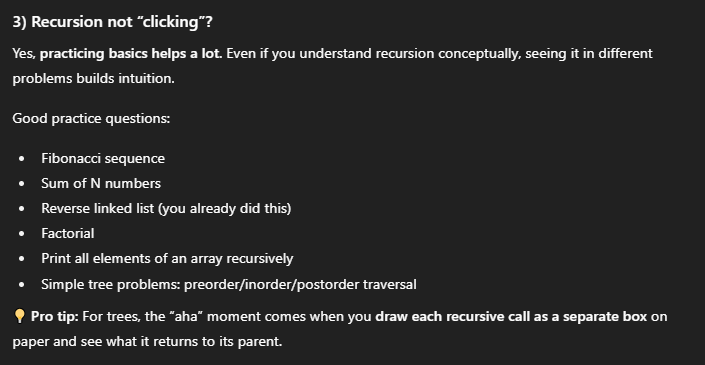


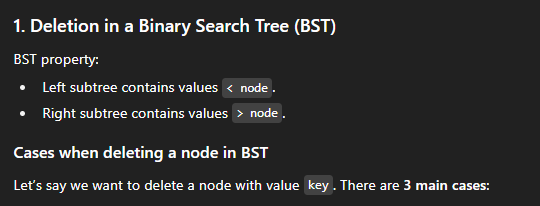


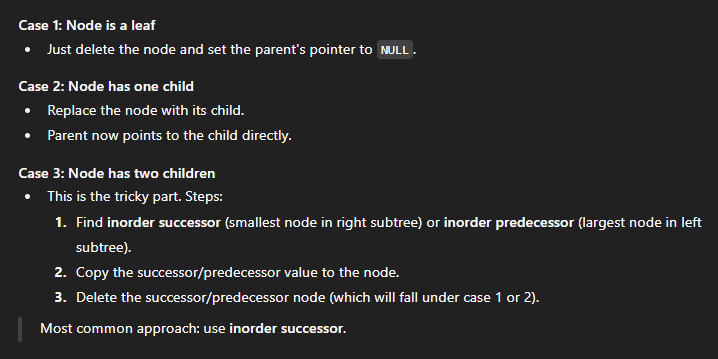


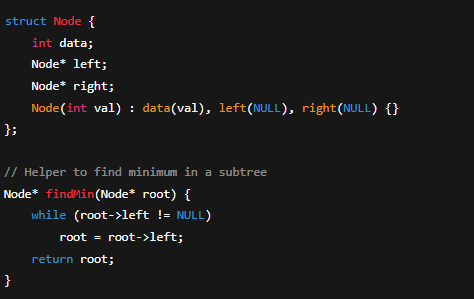


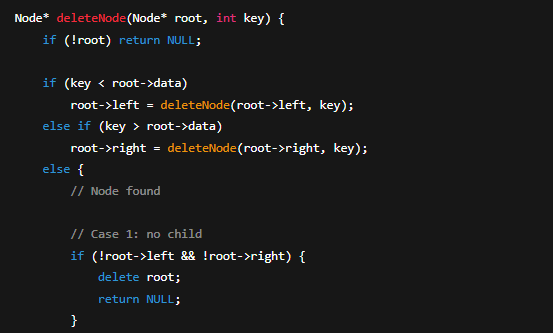


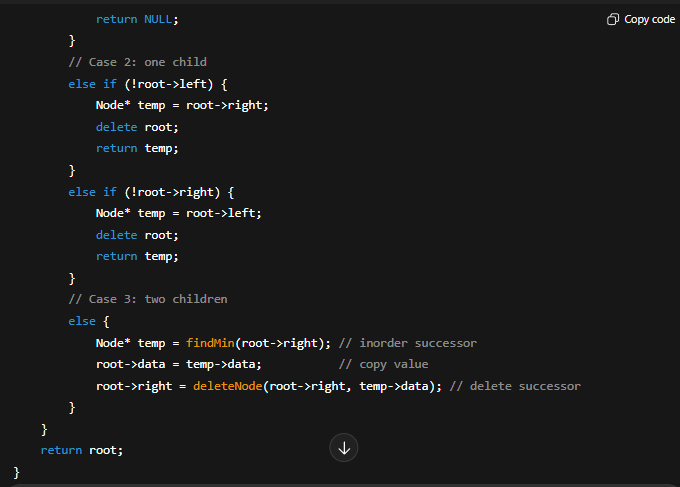


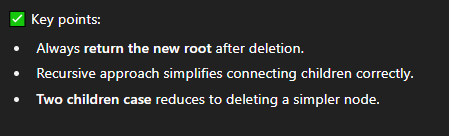




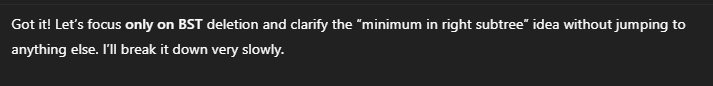


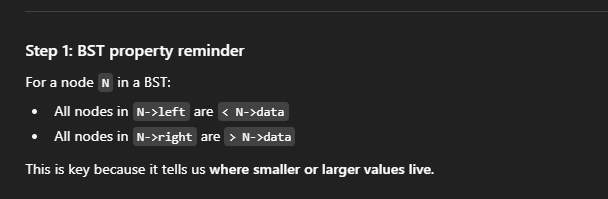




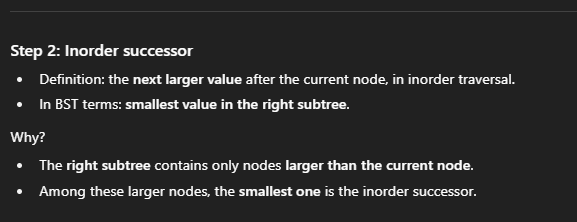


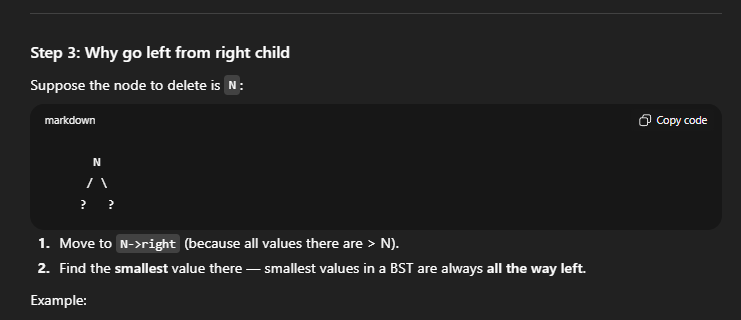
Deleting logic 🡪

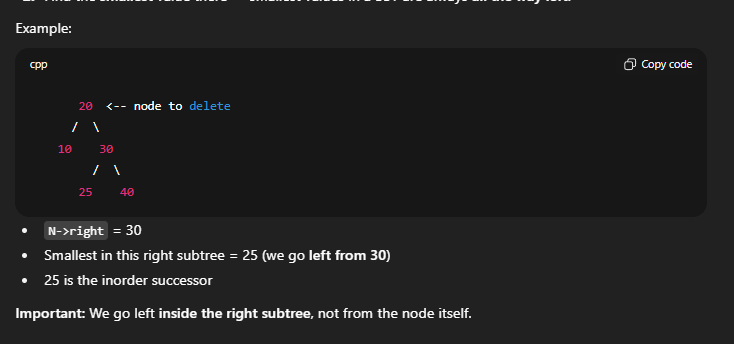


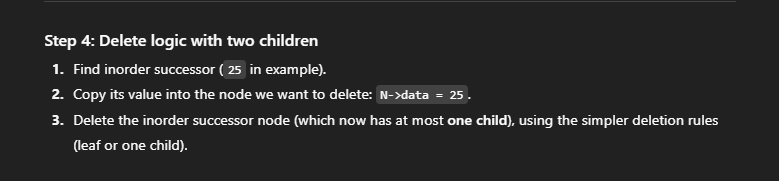


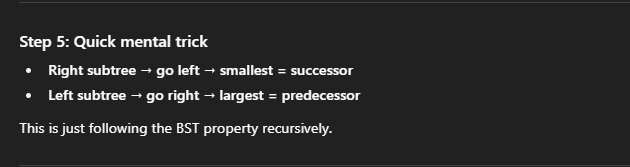
We usually go with inorder successor, or smallest in right sub tree, or smallest number larger than Node n



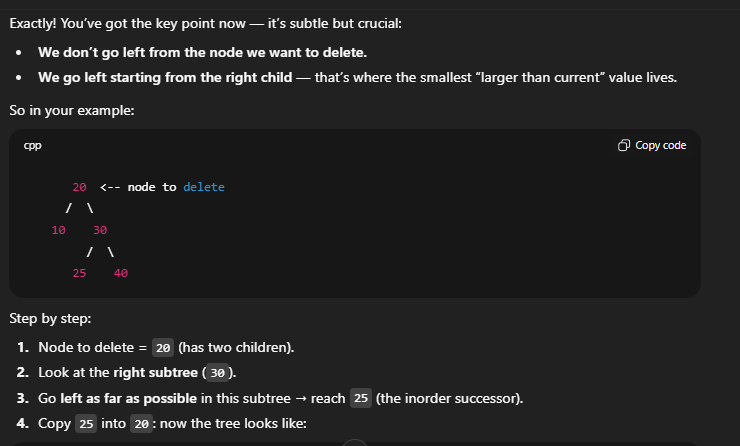


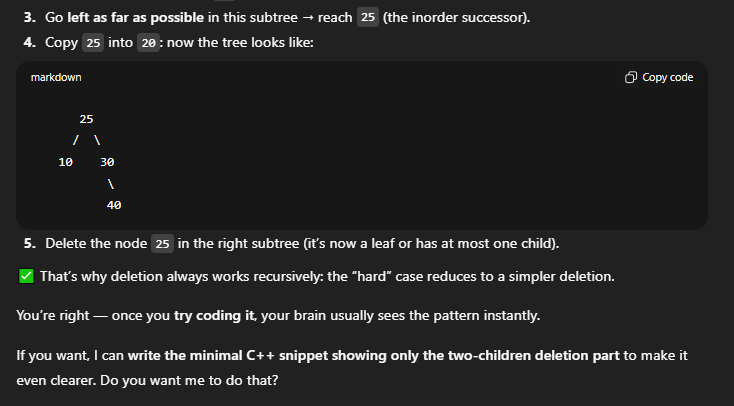




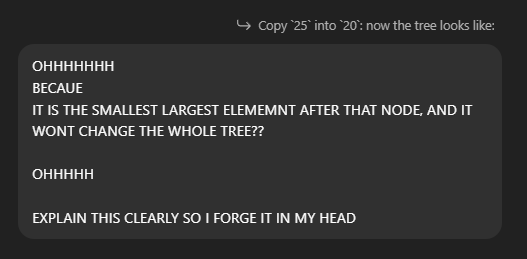


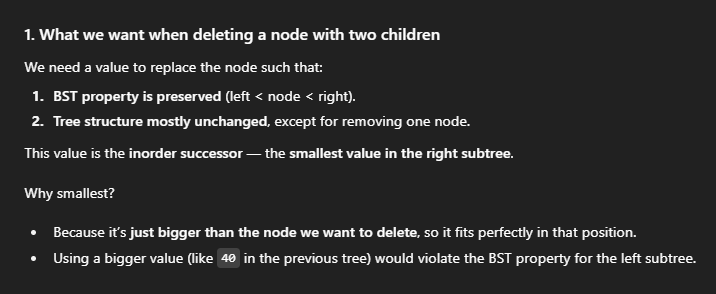
**YOU DON’T START AT THE LEFT OF THE TREE AISE HI, YOU START GOING LEFT FROM THE RIGHT SUBTREE** (ill tell u why later)





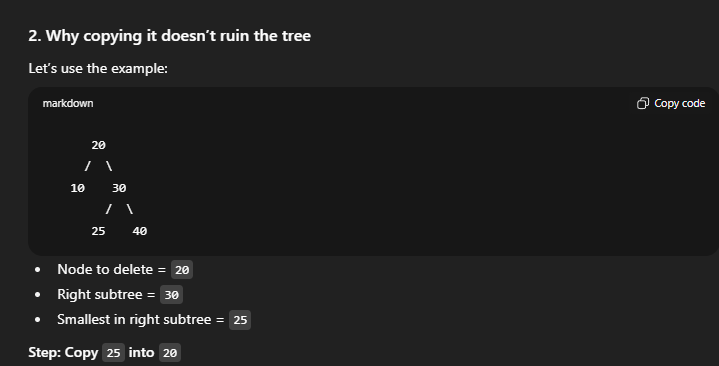
NOW UNDERSTAND **WHY** 🡪

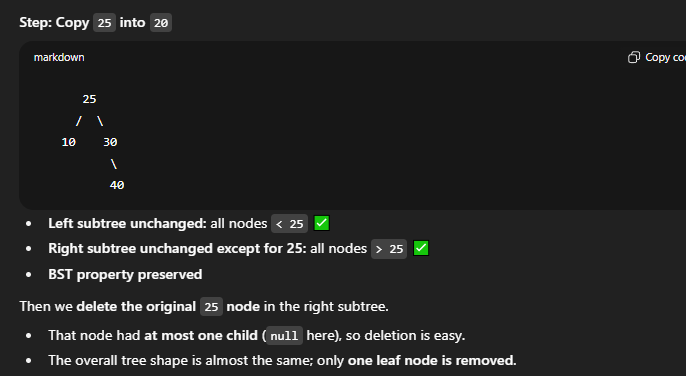


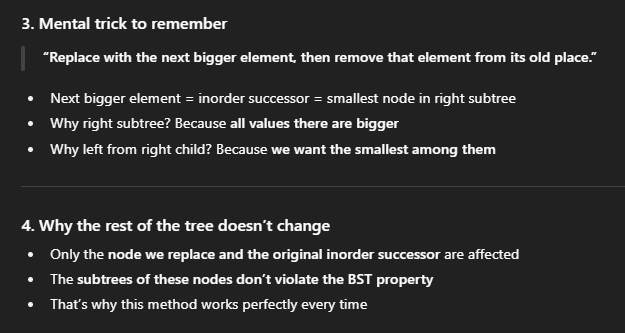


Coz anything smaller than it is the root itself and later in the left subtree.

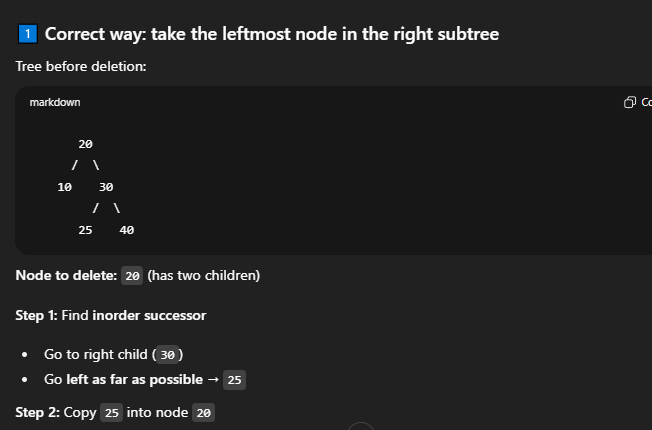
And anything bigger wont make sense cause it may alter the BST tree property(will show how later)







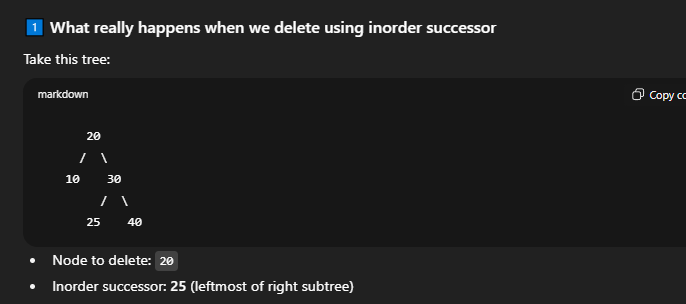
**VISUAL EXPLANATION TO MAKE IT STICK 🡪**

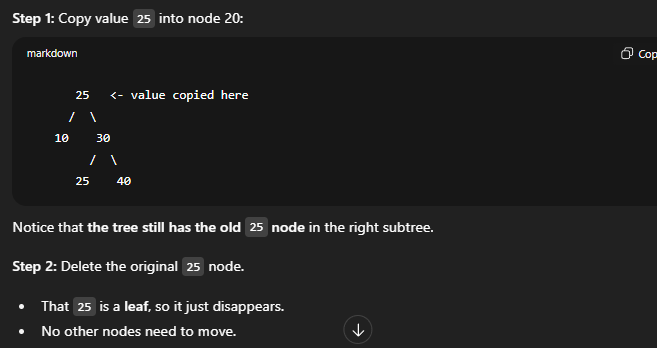
****

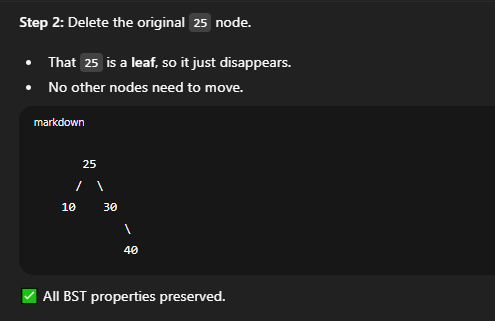
When you copy the successor onto the root, remove the og one

Why?

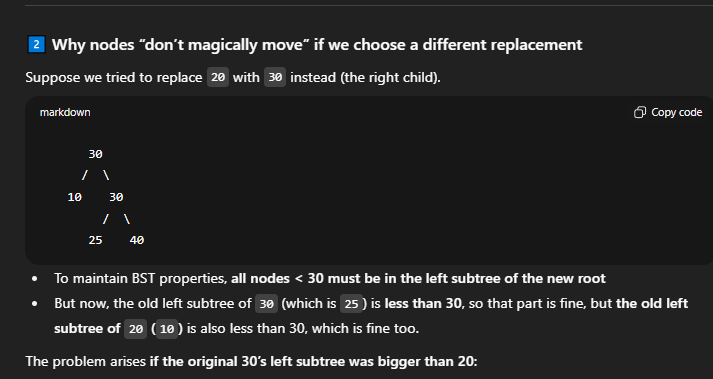
Cause you deleted the node u had to delete (here root) and replaced the empty space with a Node that wont have messed up wid the BST property(left<root<rite)

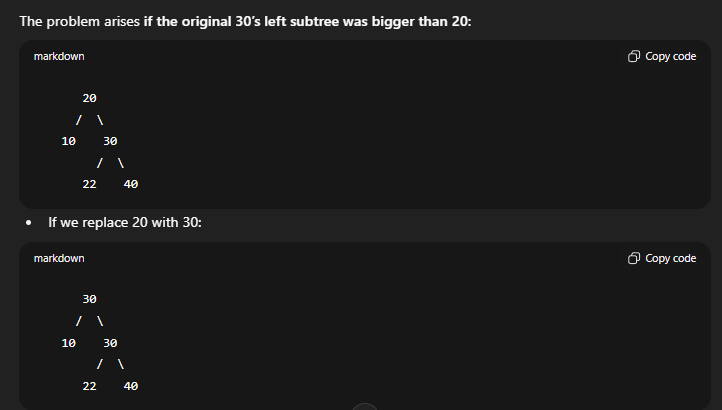
****

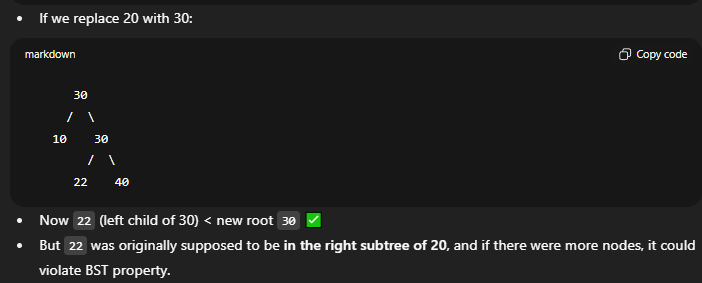
****

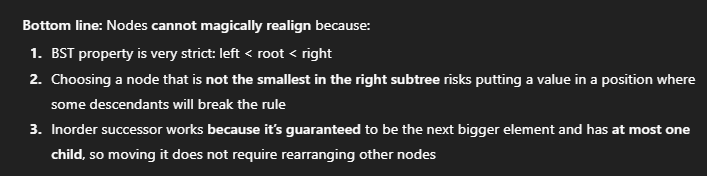
****

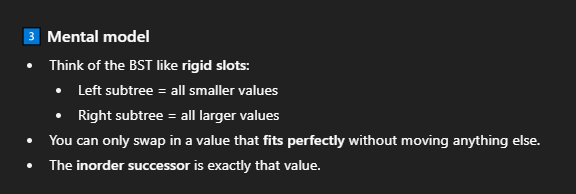
**How actually choosing the wrong one can mess up BST 🡪**

****

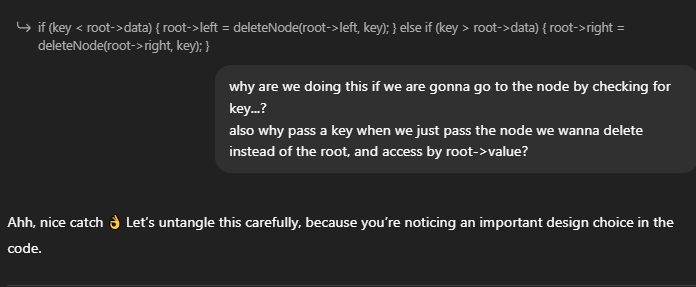
****

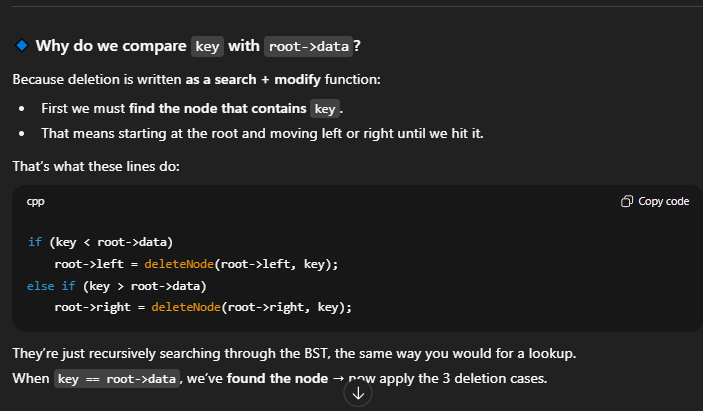
****

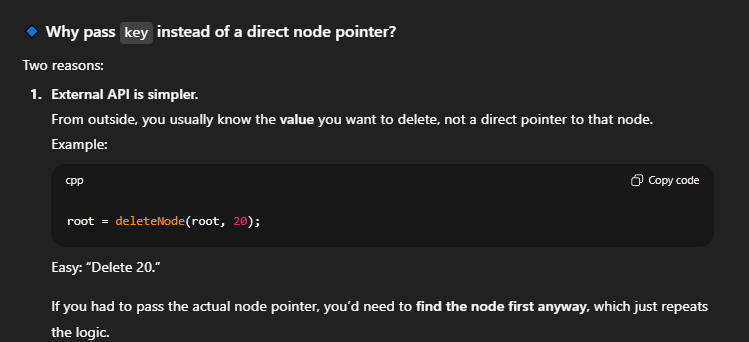
****

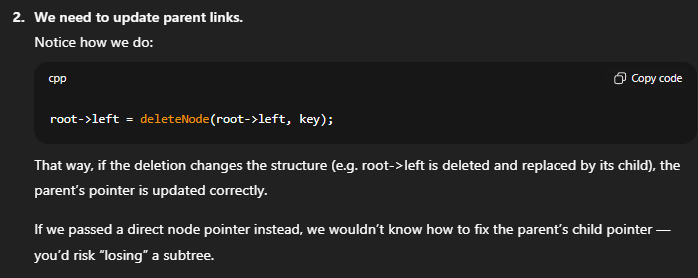
****

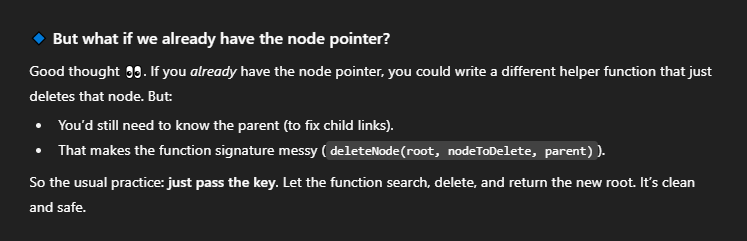
**DELETION**

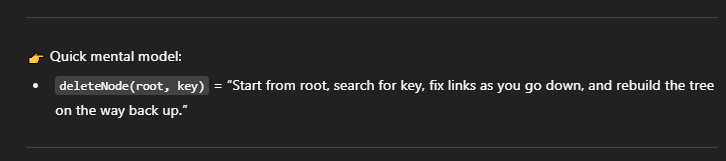
****

****

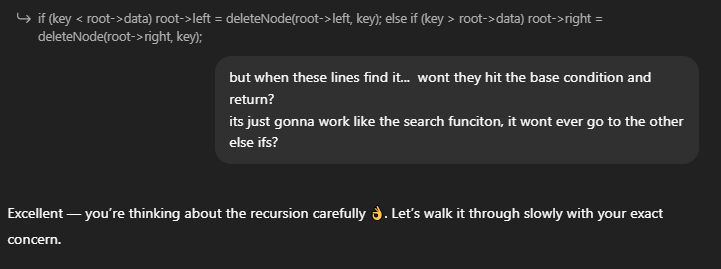
****

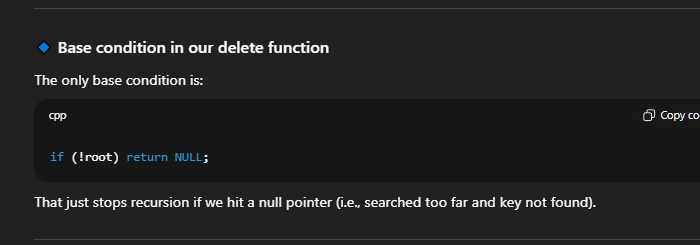
****

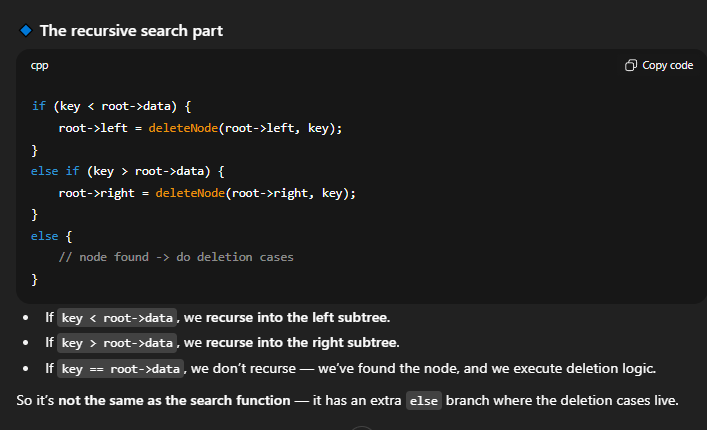
****

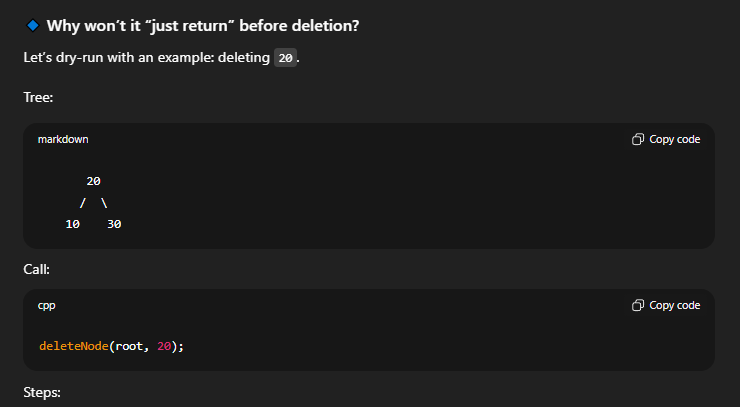
****

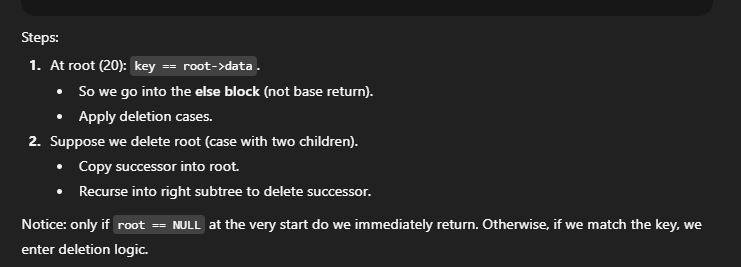
**Doubt -------🡪**

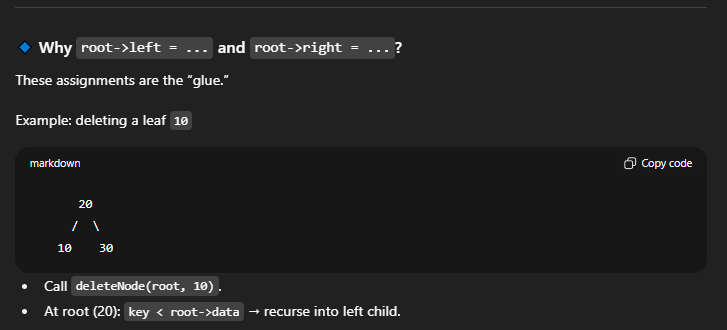
****

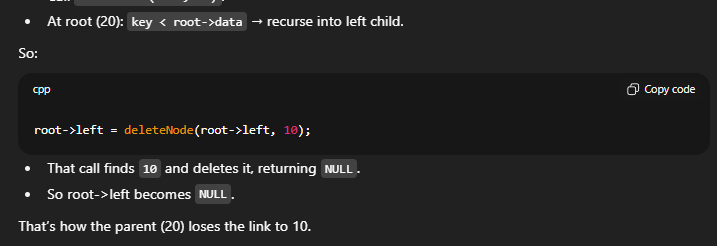
****

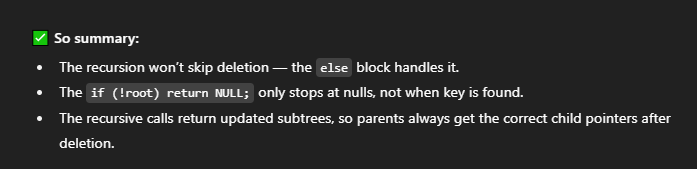
****

****

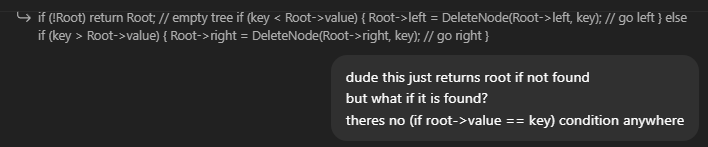
****

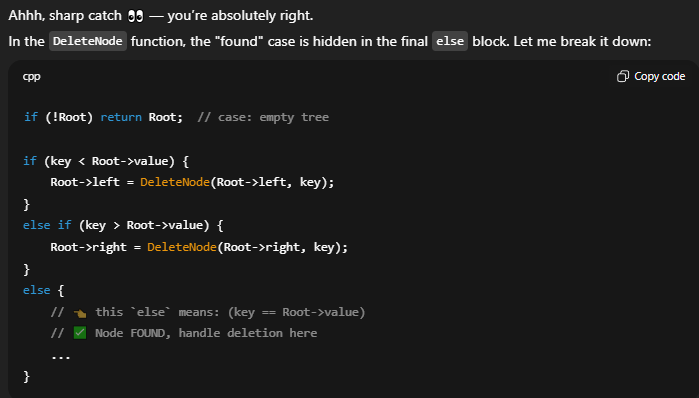
****

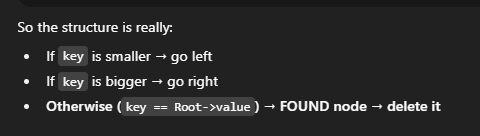
****

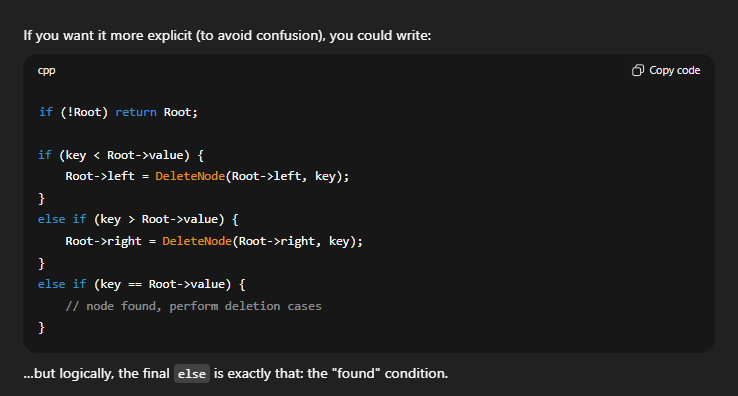
****

**AGAIN 🡪**

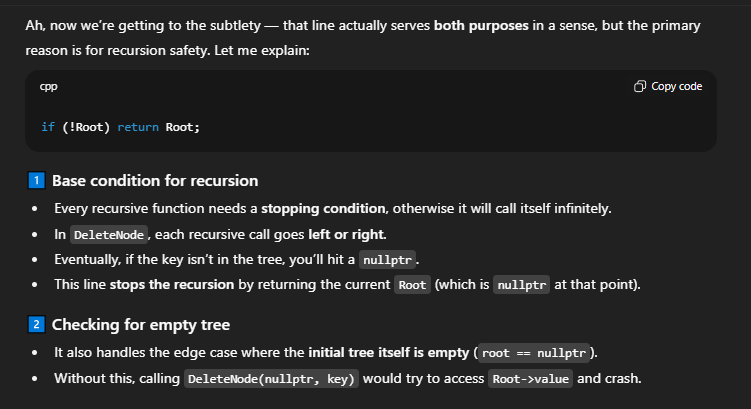
****

****

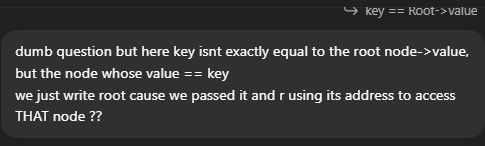
****

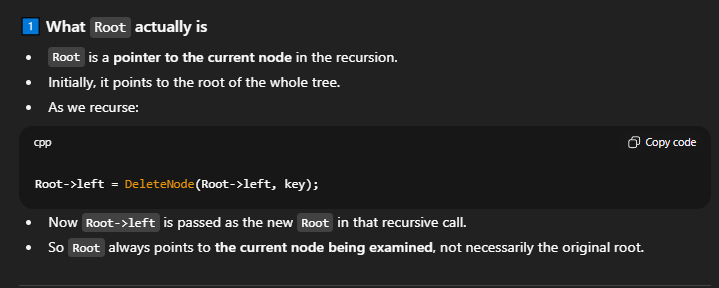
****

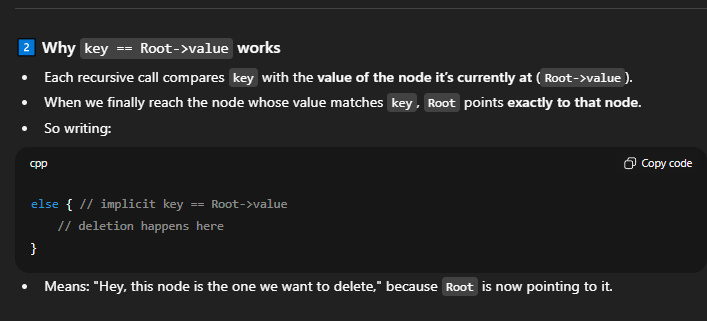
FIRST LINE --🡪

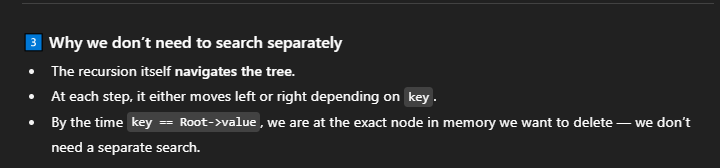


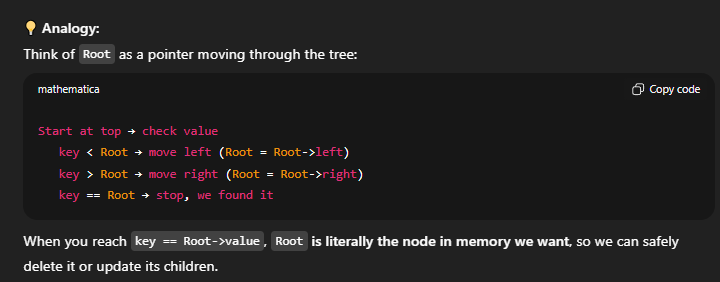
What exactly is “ROOT” ->











FINAL CODE (MINE) 🡪

#include<iostream>

using namespace std;

class Node {

public:

int value;

Node\* left;

Node\* right;

Node(int v) {

value = v;

left = nullptr;

right = nullptr;

}

};

void InOrder(Node\* Root) {

if (!Root) {

return;

}

InOrder(Root->left);

cout << Root->value << " ";

InOrder(Root->right);

}

// Inserting in a BST

Node\* Insert(Node\* Root,int newval) {

// BASE CONDITION

if (!Root) {

return new Node(newval);

}

// 2 recursive calls based on value of Root Node

// changing root node ke pointers so return new updated root node

if (Root->value < newval) {

Root->right = Insert(Root->right, newval);

}

else {

Root->left = Insert(Root->left, newval);

}

return Root;

}

// Searching in a BST

Node\* Search(Node\* Root, int key) {

// empty tree or Root->value == key

if (Root == nullptr || Root->value == key) {

return Root; // nullptr if not found, else Node pointer is returned

}

if (Root->value >= key) {

return Search(Root->left, key);

}

else {

return Search(Root->right, key);

}

}

// Same parameters as Search() function

void PrintAndSearch(Node\* Root, int key) {

Node\* Result = Search(Root, key);

if (Result) {

cout << key << " found in the tree" << endl;

}

else {

cout << key << " not found" << endl;

}

}

Node\* MinElement(Node\* Root) {

// Going to Leftmost Node

while (Root && Root->left) {

Root = Root->left;

}

return Root;

}

Node\* DeleteNode(Node\* Root, int key) {

// Empty tree and BASE CONDITION

if (!Root) {

return Root;

}

// Traversing left subtree

if (key < Root->value) {

Root->left = DeleteNode(Root->left, key);

}

// Traversing right subtree

else if (key > Root->value) {

Root->right = DeleteNode(Root->right, key);

}

// key == Root->value

else {

// Case 1 : Leaf Node

if (Root->left == nullptr && Root->right == nullptr) {

delete Root;

return nullptr;

}

// Case 2 : One child only

// Left child ONLY

if (Root->right == nullptr) {

Node\* temp = Root->left;

delete Root;

return temp;

}

// Right child ONLY

else if (Root->left == nullptr) {

Node\* temp = Root->right;

delete Root;

return temp;

}

// Case 3 : two children

// smallest element in right subtree

Node\* temp = MinElement(Root->right);

// copy that value into root (node u wanna delete)

Root->value = temp->value;

// delete original temp (inorder successor) from tree

Root->right = DeleteNode(Root->right, Root->value);

}

return Root;

}

int main() {

Node\* root = nullptr;

root = Insert(root, 50);

root = Insert(root, 30);

root = Insert(root, 70);

root = Insert(root, 20);

root = Insert(root, 40);

root = Insert(root, 60);

root = Insert(root, 80);

cout << "Inorder Traversal of BST -->" << endl;

InOrder(root); // 20 30 40 50 60 70 80

cout << endl << endl;

cout << "Search function for 40 and 100" << endl;

PrintAndSearch(root, 40);

PrintAndSearch(root, 100);

cout << endl;

cout << "Original tree --->" << endl;

InOrder(root);

cout << "\nDeleting 20,50,70" << endl;

root = DeleteNode(root, 20);

root = DeleteNode(root, 50);

root = DeleteNode(root, 70);

cout << "\nNew InOrder -->" << endl;

InOrder(root);

return 0;

}