Balanced in C++

```cpp
#include <iostream>
#include <algorithm>
using namespace std;

// Node structure for the binary tree
struct Node {
    int key;
    Node* left;
    Node* right;

    Node(int item) {
        key = item;
        left = right = nullptr;
    }
};

// Function to calculate the height of the tree and check balance
pair<bool, int> isBalancedHelper(Node* root) {
    if (root == nullptr)
        return {true, 0};

    // Recursively get heights of left and right subtrees
    auto left = isBalancedHelper(root->left);
    auto right = isBalancedHelper(root->right);

    // If either subtree is unbalanced, the whole tree is unbalanced
    if (!left.first || !right.first)
        return {false, -1};

    // Check if the current subtree is balanced
    if (abs(left.second - right.second) > 1)
        return {false, -1};

    // Return balanced status and height of the current subtree
    return {true, max(left.second, right.second) + 1};
}

// Function to check if the binary tree is balanced
bool isBalanced(Node* root) {
    return isBalancedHelper(root).first;
}

int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    root->left->left->left = new
```

Binary Tree Structure

```
        1
       / \
      2   3
     / \
    4   5
   /
  6
```

🧮 Dry Run Table: isBalancedHelper

We'll do a **postorder traversal** (left → right → root) and track the balance and height of each subtree.

Node	Left Subtree (Balanced, Height)	Right Subtree (Balanced, Height)	Height Difference	Is Current Balanced?	Current Height
6	(true, 0)	(true, 0)	0	✅ Yes	1
4	(true, 1)	(true, 0)	1	✅ Yes	2
5	(true, 0)	(true, 0)	0	✅ Yes	1
2	(true, 2)	(true, 1)	1	✅ Yes	3
3	(true, 0)	(true, 0)	0	✅ Yes	1
1	(true, 3)	(true, 1)	**2**	❌ No	—

❌ Final Result:

- Node 1 is **not balanced** because its left and right subtrees have a height difference of **2**, which is more than 1.
- Hence, isBalanced(root) returns false.

✅ Output:

Is the tree balanced? No

`` Node(6); bool balanced = isBalanced(root); cout << "Is the tree balanced? " << (balanced ? "Yes" : "No") << endl; return 0; } ```	
Is the tree balanced? No	

Binary Tree 2 LL in C++

```
#include <iostream>
using namespace std;
class Node {
public:
    int key;
    Node* left;
    Node* right;

    Node(int value) {
        key = value;
        left = nullptr;
        right = nullptr;
    }
};

class BinTree2LL {
private:
    static Node* prev;

public:
    static void flatten(Node* root) {
        if (root == nullptr) return;

        flatten(root->right);
        flatten(root->left);

        root->right = prev;
        root->left = nullptr;
        prev = root;
    }

    static void printList(Node* root) {
        while (root->right != nullptr) {
            cout << root->key << "->";
            root = root->right;
        }
        cout << root->key;
    }
};

Node* BinTree2LL::prev = nullptr;

int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->left->left = new Node(3);
    root->left->right = new Node(4);
    root->right = new Node(5);
    root->right->right = new Node(6);
    root->right->right->left = new Node(7);

    BinTree2LL::flatten(root);
    BinTree2LL::printList(root);

    // Clean up allocated memory (not present in Java version)
    while (root != nullptr) {
        Node* temp = root;
        root = root->right;
        delete temp;
```

Original Binary Tree Structure

```
      1
     / \
    2   5
   / \   \
  3   4   6
         /
        7
```

🛠 Flattening Logic: Reverse Postorder (Right → Left → Node)

The algorithm works like this:

- Traverse the tree in **reverse postorder**.
- Use a static prev pointer to keep track of the previously processed node.
- Set the current node's right to prev, and its left to nullptr.

📊 Step-by-Step Tabular Dry Run

We will track:

- The current node being visited
- The state of prev
- Links updated

Step	Node Visited	Previous (prev)	Action	Updated Links
1	7	nullptr	Set 7.right = nullptr, 7.left = nullptr, prev = 7	7 → nullptr
2	6	7	Set 6.right = 7, 6.left = nullptr, prev = 6	6 → 7
3	5	6	Set 5.right = 6, 5.left = nullptr, prev = 5	5 → 6 → 7
4	4	5	Set 4.right = 5, 4.left = nullptr, prev = 4	4 → 5 → 6 → 7
5	3	4	Set 3.right = 4, 3.left = nullptr, prev = 3	3 → 4 → ...
6	2	3	Set 2.right = 3, 2.left = nullptr,	2 → 3 → ...

```
    }

    return 0;
}
```

			prev = 2	
7	1	2	Set 1.right = 2, 1.left = nullptr, prev = 1	1 → 2 → 3 → ...

📈 Final Flattened Linked List (Right Pointers)

1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7

All left pointers are nullptr, forming a **single right-skewed list**.

✅ Output

1->2->3->4->5->6->7

1->2->3->4->5->6->7

Boundary traversal in C++

```cpp
#include <iostream>
#include <vector>
using namespace std;

// Definition of the Node class
class Node {
public:
    int key;
    Node* left;
    Node* right;

    Node(int item) {
        key = item;
        left = right = nullptr;
    }
};

// Utility function to check if a node is a leaf node
bool isLeaf(Node* root) {
    return (root->left == nullptr && root->right == nullptr);
}

// Function to add nodes of the left boundary (excluding the leaf node itself)
void addLeftBoundary(Node* root, vector<int>& res) {
    Node* cur = root->left;
    while (cur != nullptr) {
        if (!isLeaf(cur))
            res.push_back(cur->key);
        if (cur->left != nullptr)
            cur = cur->left;
        else
            cur = cur->right;
    }
}

// Function to add nodes of the right boundary (excluding the leaf node itself)
void addRightBoundary(Node* root, vector<int>& res) {
    Node* cur = root->right;
    vector<int> tmp;
    while (cur != nullptr) {
        if (!isLeaf(cur))
            tmp.push_back(cur->key);
        if (cur->right != nullptr)
            cur = cur->right;
        else
            cur = cur->left;
    }
    for (int i = tmp.size() - 1; i >= 0; --i) {
        res.push_back(tmp[i]);
    }
}

// Function to add all leaf nodes in left-to-right order
```

Binary Tree Structure

Here's the tree again for reference:

```
         1
        / \
       2   7
      /     \
     3       8
      \     /
       4   9
      / \ / \
     5  6 10 11
```

✅ Step-by-Step Tabular Dry Run

1. 🟩 Root Node

Step	Node Visited	Is Leaf?	Action	Vector State
1	1	No	Add to result	[1]

2. 🟨 Left Boundary (excluding leaves)

Traversal path: 2 → 3 → 4 (stop before leaf nodes 5, 6)

Step	Node Visited	Is Leaf?	Action	Vector State
2	2	No	Add to result	[1, 2]
3	3	No	Add to result	[1, 2, 3]
4	4	No	Add to result	[1, 2, 3, 4]

3. 🟦 Leaf Nodes (from left to right)

Leaf nodes: 5, 6, 10, 11

Step	Node Visited	Is Leaf?	Action	Vector State
5	5	Yes	Add to result	[1, 2, 3, 4, 5]
6	6	Yes	Add to result	[1, 2, 3, 4, 5, 6]
7	10	Yes	Add to	[1, 2, 3, 4, 5, 6,

```
void addLeaves(Node* root, vector<int>& res) {
    if (isLeaf(root)) {
        res.push_back(root->key);
        return;
    }
    if (root->left != nullptr)
        addLeaves(root->left, res);
    if (root->right != nullptr)
        addLeaves(root->right, res);
}

// Function to perform boundary traversal and
return the result as vector
vector<int> printBoundary(Node* node) {
    vector<int> ans;
    if (!isLeaf(node))
        ans.push_back(node->key);
    addLeftBoundary(node, ans);
    addLeaves(node, ans);
    addRightBoundary(node, ans);
    return ans;
}

int main() {
    // Constructing the binary tree
    Node* root = new Node(1);
    root->left = new Node(2);
    root->left->left = new Node(3);
    root->left->left->right = new Node(4);
    root->left->left->right->left = new Node(5);
    root->left->left->right->right = new Node(6);
    root->right = new Node(7);
    root->right->right = new Node(8);
    root->right->right->left = new Node(9);
    root->right->right->left->left = new Node(10);
    root->right->right->left->right = new Node(11);

    // Performing boundary traversal
    vector<int> boundaryTraversal =
printBoundary(root);

    // Printing the result
    cout << "The Boundary Traversal is : ";
    for (int i = 0; i < boundaryTraversal.size(); i++)
{
        cout << boundaryTraversal[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Step	Node Visited	Is Leaf?	Action	Vector State
			result	10]
8	11	Yes	Add to result	[1, 2, 3, 4, 5, 6, 10, 11]

4. 🟥 Right Boundary (excluding leaves) — reversed

Traversal path: 7 → 8 → 9 (reverse order, ignore 10 and 11)

Step	Node Visited	Is Leaf?	Action (store in temp, then reverse)	Temporary Stack	Vector State (after reverse append)
9	7	No	Push to temp	[7]	
10	8	No	Push to temp	[7, 8]	
11	9	No	Push to temp	[7, 8, 9]	
12	--	--	Reverse and append to result		[1, 2, 3, 4, 5, 6, 10, 11, 9, 8, 7]

🎯 Final Result
Boundary Traversal: [1, 2, 3, 4, 5, 6, 10, 11, 9, 8, 7]

Boundary Traversal: [1, 2, 3, 4, 5, 6, 10, 11, 9, 8, 7]

Children Sum in C++

```
#include <iostream>
using namespace std;

// Definition of the Node class
class Node {
public:
    int key;
    Node* left;
    Node* right;

    Node(int item) {
        key = item;
        left = right = nullptr;
    }
};

// Function to reorder the binary tree based on Children Sum Property
void reorder(Node* root) {
    if (root == nullptr) return;

    int child = 0;
    if (root->left != nullptr) {
        child += root->left->key;
    }
    if (root->right != nullptr) {
        child += root->right->key;
    }

    if (child < root->key) {
        if (root->left != nullptr) root->left->key = root->key;
        else if (root->right != nullptr) root->right->key = root->key;
    }

    reorder(root->left);
    reorder(root->right);

    int tot = 0;
    if (root->left != nullptr) tot += root->left->key;
    if (root->right != nullptr) tot += root->right->key;
    if (root->left != nullptr || root->right != nullptr) root->key = tot;
}

// Function to change the tree based on Children Sum Property
void changeTree(Node* root) {
    reorder(root);
}

int main() {
    Node* root = new Node(2);
    root->left = new Node(35);
    root->left->left = new Node(2);
    root->left->right = new Node(3);
    root->right = new Node(10);
    root->right->left = new Node(5);
    root->right->right = new Node(2);
```

Initial Tree Structure

```
          2
        /   \
      35     10
     /  \   /  \
    2    3  5    2
```

🔄 Dry Run: Step-by-Step Execution

Node Visited	Children Before	Action Taken	Node Key After
2 (root)	35 + 10 = 45	Children > root → No update to children	—
35	2 + 3 = 5	Children < 35 → Set both children to 35	—
2 (left)	null	Leaf node	35
3 (right)	null	Leaf node	35
Back to 35	35 + 35 = 70	Set node key = sum of children	70
10	5 + 2 = 7	Children < 10 → Set left to 10 (since left exists)	—
5 (left)	null	Leaf node	10
2 (right)	null	Leaf node	2
Back to 10	10 + 2 = 12	Set node key = sum of children	12
Back to root	70 + 12 = 82	Set root = sum of its updated children	82

🌳 Final Tree Structure

```
          82
        /    \
      70      12
     /  \    /  \
   35   35  10    2
```

✅ Output

```
Modified Tree:
Root: 82
Left: 70, Left Left: 35, Left Right: 35
Right: 12, Right Left: 10, Right Right: 2
```

changeTree(root); // Display the modified tree cout << "Modified Tree:" << endl; cout << "Root: " << root->key << endl; cout << "Left: " << root->left->key << ", Left Left: " << root->left->left->key << ", Left Right: " << root->left->right->key << endl; cout << "Right: " << root->right->key << ", Right Left: " << root->right->left->key << ", Right Right: " << root->right->right->key << endl; return 0; }	🔁 **Summary of Key Logic in `reorder()`:** 1. **Preorder Phase**: o Push parent's value down to children if sum of children < parent. 2. **Postorder Phase**: o After children updated, update parent's value as sum of updated children.
Modified Tree: Root: 50 Left: 38, Left Left: 35, Left Right: 3 Right: 12, Right Left: 10, Right Right: 2	

Diameter in C++

```cpp
#include <iostream>
#include <algorithm> // For std::max
using namespace std;

// Definition of the Node class
class Node {
public:
    int key;
    Node* left;
    Node* right;

    Node(int item) {
        key = item;
        left = right = nullptr;
    }
};

// Function prototype for height
int height(Node* node, int* diameter);

// Function to calculate diameter of binary tree
int diameterOfBinaryTree(Node* root) {
    int diameter = 0;
    height(root, &diameter);
    return diameter;
}

// Helper function to calculate height and
update diameter
int height(Node* node, int* diameter) {
    if (node == nullptr) {
        return 0;
    }

    int leftHeight = height(node->left,
diameter);
    int rightHeight = height(node->right,
diameter);

    *diameter = max(*diameter, leftHeight +
rightHeight);

    return 1 + max(leftHeight, rightHeight);
}

int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    root->left->left->left = new Node(6);

    int dia = diameterOfBinaryTree(root);
    cout << "Diameter of the binary tree: " <<
dia << endl;

    return 0;
}
```

Tree Structure

Based on your construction, the tree looks like this:

```
      1
     / \
    2   3
   / \
  4   5
 /
6
```

🔍 What Is *Diameter*?

The **diameter** is the **length of the longest path** between any two nodes in the tree (measured by number of edges, not nodes).
This path **does not necessarily pass through the root**.

🧠 Core Logic Summary

- For each node:
  - Compute leftHeight and rightHeight.
  - Update diameter = max(diameter, leftHeight + rightHeight).
- Height is returned as 1 + max(leftHeight, rightHeight).

📋 Dry Run Table

Node	Left Height	Right Height	Local Diameter (L + R)	Max Diameter So Far	Returned Height
6	0	0	0	0	1
4	1	0	1	1	2
5	0	0	0	1	1
2	2	1	3	✅ 3	3
3	0	0	0	3	1
1	3	1	4	✅ 4	4

✅ Final Output

Diameter of the binary tree: 4

Diameter of the binary tree: 4

Identical in C++

```cpp
#include <iostream>

using namespace std;

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) {
        val = x;
        left = nullptr;
        right = nullptr;
    }
};

class Identical {
public:
    static bool isIdentical(TreeNode* node1, TreeNode* node2) {
        if (node1 == nullptr && node2 == nullptr)
            return true;
        else if (node1 == nullptr || node2 == nullptr)
            return false;

        return (node1->val == node2->val) &&
               isIdentical(node1->left, node2->left) &&
               isIdentical(node1->right, node2->right);
    }
};

int main() {
    TreeNode* root1 = new TreeNode(1);
    root1->left = new TreeNode(2);
    root1->right = new TreeNode(3);
    root1->right->left = new TreeNode(4);
    root1->right->right = new TreeNode(5);

    TreeNode* root2 = new TreeNode(1);
    root2->left = new TreeNode(2);
    root2->right = new TreeNode(3);
    root2->right->left = new TreeNode(4);
```

Tree Structures:

Tree 1:

```
   1
  / \
 2   3
    / \
   4   5
```

Tree 2:

```
   1
  / \
 2   3
    /
   4
```

🧮 Dry Run Table: isIdentical(root1, root2)

Call	node1 Val	node2 Val	Equal?	Recursive Calls	Final Result
isIdentical(1, 1)	1	1	✓	isIdentical(2, 2) && isIdentical(3, 3)	depends
└── isIdentical(2, 2)	2	2	✓	isIdentical(nullptr, nullptr)	✓
└── isIdentical(NULL, NULL)	NULL	NULL	✓		✓
└── isIdentical(NULL, NULL)	NULL	NULL	✓		✓
└── isIdentical(3, 3)	3	3	✓	isIdentical(4, 4) && isIdentical(5, NULL)	✗
└── isIdentical(4, 4)	4	4	✓	isIdentical(NULL, NULL)	✓
└── isIdentical(NULL,NULL)	NULL	NULL	✓		✓
└── isIdentical(NULL,NULL)	NULL	NULL	✓		✓
└── isIdentical(5, NULL)	5	NULL	✗		✗

✗ **Final Output:**

Two trees are non-identical

if (Identical::isIdentical(root1, root2)) cout << "Two Trees are identical" << endl; else cout << "Two trees are non-identical" << endl; return 0; }	
Two trees are non-identical	

Iterative Inorder in C++

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;
// TreeNode structure definition
struct TreeNode {
    int key;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) {
        key = x;
        left = nullptr;
        right = nullptr;
    }
};

// Function to perform iterative inorder traversal
vector<int> inOrderTrav(TreeNode* root) {
    vector<int> inOrder;
    stack<TreeNode*> s;
    TreeNode* curr = root;

    while (true) {
        if (curr != nullptr) {
            s.push(curr);
            curr = curr->left;
        } else {
            if (s.empty()) break;
            curr = s.top();
            inOrder.push_back(curr->key);
            s.pop();
            curr = curr->right;
        }
    }
    return inOrder;
}

int main() {
    // Constructing the binary tree
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->left->right->left = new TreeNode(8);
    root->right->left = new TreeNode(6);
    root->right->right = new TreeNode(7);
    root->right->right->left = new TreeNode(9);
    root->right->right->right = new TreeNode(10);

    // Perform iterative inorder traversal
    vector<int> inOrder = inOrderTrav(root);

    // Print the result
    cout << "The inorder traversal is : ";
    for (int i = 0; i < inOrder.size(); i++) {
        cout << inOrder[i] << " ";
    }
    cout << endl;
```

Tree Structure:

```
        1
      /   \
     2     3
    / \   / \
   4   5 6   7
      /     / \
     8     9  10
```

🧮 Dry Run Table

Step	Current Node (curr)	Stack (top → bottom)	Action	Output (inOrder)
1	1		Push 1, move to left	
2	2	1	Push 2, move to left	
3	4	2 → 1	Push 4, move to left	
4	nullptr	4 → 2 → 1	Pop 4, visit	4
5	nullptr (right of 4)	2 → 1	Pop 2, visit	4 2
6	5	1	Push 5, move to left	4 2
7	8	5 → 1	Push 8, move to left	4 2
8	nullptr	8 → 5 → 1	Pop 8, visit	4 2 8
9	nullptr (right of 8)	5 → 1	Pop 5, visit	4 2 8 5
10	nullptr (right of 5)	1	Pop 1, visit	4 2 8 5 1
11	3		Push 3, move to left	4 2 8 5 1
12	6	3	Push 6, move to left	4 2 8 5 1

```
    return 0;
}
```

13	nullptr	6 → 3	Pop 6, visit	4 2 8 5 1 6
14	nullptr (right of 6)	3	Pop 3, visit	4 2 8 5 1 6 3
15	7		Push 7, move to left	4 2 8 5 1 6 3
16	9	7	Push 9, move to left	4 2 8 5 1 6 3
17	nullptr	9 → 7	Pop 9, visit	4 2 8 5 1 6 3 9
18	nullptr (right of 9)	7	Pop 7, visit	4 2 8 5 1 6 3 9 7
19	10		Push 10, move to left	4 2 8 5 1 6 3 9 7
20	nullptr	10	Pop 10, visit	4 2 8 5 1 6 3 9 7 10

✅ **Final Output:**

The inorder traversal is : 4 2 8 5 1 6 3 9 7 10

The inorder traversal is : 4 2 8 5 1 6 3 9 7 10

Max path sum in C++

```cpp
#include <iostream>
#include <climits>   // For INT_MIN
#include <algorithm> // For std::max
using namespace std;
// TreeNode structure definition
struct TreeNode {
    int key;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) {
        key = x;
        left = nullptr;
        right = nullptr;
    }
};

// Helper function to calculate the maximum path sum going down from a node
int maxPathDown(TreeNode* node, int& maxValue) {
    if (node == nullptr) return 0;

    // Calculate maximum path sums from left and right subtrees
    int left = std::max(0, maxPathDown(node->left, maxValue)); // Ignore negative sums
    int right = std::max(0, maxPathDown(node->right, maxValue)); // Ignore negative sums

    // Update maxValue with the maximum path sum found so far
    maxValue = std::max(maxValue, left + right + node->key);

    // Return the maximum path sum going down from the current node
    return std::max(left, right) + node->key;
}

// Function to find the maximum path sum in a binary tree
int maxPathSum(TreeNode* root) {
    int maxValue = INT_MIN; // Initialize with minimum possible integer value
    maxPathDown(root, maxValue);
    return maxValue;
}

int main() {
    // Constructing the binary tree
    TreeNode* root = new TreeNode(-10);
    root->left = new TreeNode(9);
    root->right = new TreeNode(20);
    root->right->left = new TreeNode(15);
    root->right->right = new TreeNode(7);

    // Finding the maximum path sum in the binary tree
    int answer = maxPathSum(root);
```

Tree Structure

You built this binary tree:

```
   -10
   / \
  9   20
     / \
    15  7
```

🧠 Core Logic (Recap)

1. maxPathDown(node):
    - Gets **max sum** for any path **starting** from the current node and going **downward**.
    - Ignores negative subtrees (max(0, left/right)).
    - Updates the global maxValue if a new candidate sum left + right + node->key is higher.

📋 Dry Run Table

Node	Left Subtree	Right Subtree	Local Max (left + right + node)	Return Upward	maxValue Updated
15	0	0	15	15	✅ 15
7	0	0	7	7	❌
20	15	7	42 (=15+7+20)	35	✅ 42
9	0	0	9	9	❌
-10	9	35	34 (=9+35-10)	25	❌

🧠 So the final max path **goes through 15 → 20 → 7 = 42**

✅ Output:

The Max Path Sum for this tree is 42

`std::cout << "The Max Path Sum for this tree is " << answer << std::endl;` `// Deallocating memory` `delete root->right->right;` `delete root->right->left;` `delete root->right;` `delete root->left;` `delete root;` `return 0;` `}`	
The Max Path Sum for this tree is 42	

Morris traversal in C++

```cpp
#include <iostream>
#include <vector>
using namespace std;
// TreeNode structure definition
struct TreeNode {
    int key;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) {
        key = x;
        left = nullptr;
        right = nullptr;
    }
};

// Function to perform Morris preorder traversal
vector<int> preorderTraversal(TreeNode* root) {
    vector<int> preorder;
    TreeNode* cur = root;

    while (cur != nullptr) {
        if (cur->left == nullptr) {
            preorder.push_back(cur->key);
            cur = cur->right;
        } else {
            TreeNode* prev = cur->left;
            while (prev->right != nullptr && prev->right !
= cur) {
                prev = prev->right;
            }

            if (prev->right == nullptr) {
                prev->right = cur;
                preorder.push_back(cur->key);
                cur = cur->left;
            } else {
                prev->right = nullptr;
                cur = cur->right;
            }
        }
    }

    return preorder;
}

int main() {
    // Constructing the binary tree
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->left->right->right = new TreeNode(6);

    // Performing Morris preorder traversal
    vector<int> preorder = preorderTraversal(root);

    // Printing the result
    cout << "The Preorder Traversal is: ";
    for (int i = 0; i < preorder.size(); i++) {
```

Tree Structure

```
      1
     / \
    2   3
   / \
  4   5
       \
        6
```

🧠 Morris Preorder Key Idea

- Use the **rightmost node** in the left subtree to **thread** back to the current node.
- When revisiting via the thread, remove the link and move right.

□ Dry Run Table

We'll walk through the preorderTraversal function.

Step	cur	Action	preorder	Thread Created?
1	1	Left exists → find predecessor (5)	[1]	✅ prev->right = 1
2	2	Left exists → find predecessor (4)	[1, 2]	✅ prev->right = 2
3	4	No left child → visit, move right (nullptr)	[1, 2, 4]	❌
4	2	Thread exists → remove, move right to 5		🔄
5	5	No left child → visit, move right to 6	[1, 2, 4, 5]	❌
6	6	No left child → visit, move right (nullptr)	[1, 2, 4, 5, 6]	❌
7	1	Thread exists → remove, move right to 3		🔄
8	3	No left child → visit, move right (nullptr)	[1, 2, 4, 5, 6, 3]	❌

✅ Final Output:

The Preorder Traversal is: 1 2 4 5 6 3

<pre> cout << preorder[i] << " "; } cout << endl; // Deallocating memory delete root->left->right->right; delete root->left->right; delete root->left; delete root->right; delete root; return 0; }</pre>	
The Preorder Traversal is: 1 2 4 5 6 3	

Root 2 Node path in C++

```cpp
#include <iostream>
#include <vector>
using namespace std;
// TreeNode structure definition
struct TreeNode {
    int key;
    TreeNode* left;
    TreeNode* right;
   TreeNode(int x) {
       key = x;
       left = nullptr;
       right = nullptr;
    }
};

// Function to get the path from root to a node with
value x
bool getPath(TreeNode* root, vector<int>& arr, int x)
{
    // If root is NULL, there is no path
    if (root == nullptr)
        return false;

    // Push the node's value into 'arr'
    arr.push_back(root->key);

    // If it is the required node, return true
    if (root->key == x)
        return true;

    // Check in the left subtree and right subtree
    if (getPath(root->left, arr, x) || getPath(root-
>right, arr, x))
        return true;

    // If the required node does not lie in either subtree,
    // remove current node's value from 'arr' and return
false
    arr.pop_back();
    return false;
}

int main() {
    // Constructing the binary tree
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->left->right->left = new TreeNode(6);
    root->left->right->right = new TreeNode(7);
    root->right = new TreeNode(3);

    vector<int> arr;

    bool res = getPath(root, arr, 7);

    if (res) {
        cout << "The path is: ";
        for (int it : arr) {
            cout << it << " ";
```

Tree Structure

```
      1
     / \
    2   3
   / \
  4   5
     / \
    6   7
```

🧭 **Target: 7**

We'll step through getPath(root, arr, 7).

Step	Current Node	arr Content	Found?
1	1	[1]	❌
2	2	[1, 2]	❌
3	4	[1, 2, 4]	❌ → backtrack
4	Backtrack	[1, 2]	
5	5	[1, 2, 5]	❌
6	6	[1, 2, 5, 6]	❌ → backtrack
7	Backtrack	[1, 2, 5]	
8	7	[1, 2, 5, 7]	✅ Found!

✅ **Final Output:**

The path is: 1 2 5 7

<code> } cout << endl; } else { cout << "Node not found in the tree." << endl; } // Deallocating memory delete root->left->right->right; delete root->left->right->left; delete root->left->right; delete root->left->left; delete root->left; delete root->right; delete root; return 0; }</code>	
The path is: 1 2 5 7	