LCA in C++

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

// Definition of a binary tree node
struct Node {
    int data;
    Node *left, *right;

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

// Function to find the Lowest Common Ancestor
(LCA) of two nodes
Node* getLCA(Node* root, int a, int b) {
    if (root == nullptr) {
        return nullptr;
    }
    if (root->data == a || root->data == b) {
        return root;
    }

    Node* lca1 = getLCA(root->left, a, b);
    Node* lca2 = getLCA(root->right, a, b);

    if (lca1 != nullptr && lca2 != nullptr) {
        return root;
    }
    if (lca1 != nullptr) {
        return lca1;
    } else {
        return lca2;
    }
}

// Function to create a binary tree and find LCA
int main() {
    // Hardcoded tree construction
    Node* root = new Node(6);
    root->left = new Node(3);
    root->right = new Node(8);
    root->right->left = new Node(7);
    root->right->right = new Node(9);

    // Find LCA of nodes 3 and 7
    Node* lcaNode = getLCA(root, 3, 7);
    cout << "Lowest Common Ancestor of 3 and 7 is: "
<< lcaNode->data << endl;

    // Clean up dynamically allocated memory
    delete root->right->right;
    delete root->right->left;
    delete root->left;
    delete root;
    return 0;
}
```

Tree Structure:

```
   6
  / \
 3   8
    / \
   7   9
```

You're finding the **LCA of 3 and 7**.

🔍 Dry Run of getLCA(root, 3, 7):

Function Call	Returns	Reason
getLCA(6, 3, 7)	→ 6	Found 3 in left subtree, 7 in right subtree → current is LCA
├── getLCA(3, 3, 7)	→ 3	root->data == a (found node 3)
└── getLCA(8, 3, 7)	→ 7	found 7 in left subtree, right subtree (9) doesn't contain target
├── getLCA(7, 3, 7)	→ 7	root->data == b (found node 7)
└── getLCA(9, 3, 7)	→ nullptr	no match

✅ Output:

Lowest Common Ancestor of 3 and 7 is: 6

Lowest Common Ancestor of 3 and 7 is: 6

Node at distance K in C++

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

// Definition of a binary
tree node
struct Node {
    int data;
    Node* left;
    Node* right;

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

// Function declaration
void
printNodesDown(Node*
root, int k);

// Function to print nodes
at distance k from the
given node
int
nodesAtDistanceKWithRo
otDistance(Node* root, int
node, int k) {
    if (root == nullptr) {
        return -1;
    }

    // If the current node is
the target node, print
nodes at distance k from it
    if (root->data == node) {

printNodesDown(root, k);
        return 0;
    }

    // Recursively search in
left subtree
    int leftHeight =
nodesAtDistanceKWithRo
otDistance(root->left,
node, k);
    if (leftHeight != -1) {
        // If the target node is
found in the left subtree
        if (leftHeight + 1 ==
k) {
            cout << root->data
<< endl;
        } else {
            // Print nodes at
distance k from the right
subtree
```

Binary Tree Structure:

```
          2
        /   \
      3       9
     / \       \
    4   8       2
   /           /
  4           6
```

🧠 Objective:

Print all nodes that are **exactly `k=2` distance** away from node with value `3`.

📝 Dry Run Table:

Step	Function Call	Current Node	Action	Output	Return Value
1	`nodesAtDistanceK(root=2, node=3, k=2)`	2	Call `nodesAtDistanceKWithRootDistance`		
2	`nodesAtDistanceKWithRootDistance(root=2, node=3, k=2)`	2	Not target → search left and right		
3	`nodesAtDistanceKWithRootDistance(root=3, node=3, k=2)`	3	🎯 Target found! Call `printNodesDown(3, 2)`		0
4	`printNodesDown(root=3, k=2)`	3	Go down to distance 2		
5	`printNodesDown(root=4, k=1)`	4	Recurse to left → node 4		
6	`printNodesDown(root=4, k=0)`	4 (leaf)	✅ Distance 0 → print 4	4	
7	`printNodesDown(root=8, k=1)`	8	No children		
8	Back to step 2, leftHeight = 0		Check if root (2) is at k=2? No → Call `printNodesDown(right, k-2)`		
9	`printNodesDown(root=9, k=0)`	9	✅ Distance 0 → print 9	9	
10	All done		Final output = `4, 9`		

```
printNodesDown(root-
>right, k - leftHeight - 2);
        }
        return leftHeight + 1;
    }

    // Recursively search in
right subtree
    int rightHeight =
nodesAtDistanceKWithRo
otDistance(root->right,
node, k);
    if (rightHeight != -1) {
        // If the target node is
found in the right subtree
        if (rightHeight + 1 ==
k) {
            cout << root->data
<< endl;
        } else {
            // Print nodes at
distance k from the left
subtree

printNodesDown(root-
>left, k - rightHeight - 2);
        }
        return rightHeight +
1;
    }

    // If the target node is
not found in either subtree
    return -1;
}

// Function to print nodes
at distance k from a given
node downwards
void
printNodesDown(Node*
root, int k) {
    if (root == nullptr || k
< 0) {
        return;
    }

    // If reached the
required distance, print
the node
    if (k == 0) {
        cout << root->data <<
endl;
        return;
    }

    // Recursively print
nodes at distance k in both
subtrees
    printNodesDown(root-
>left, k - 1);
```

✅ **Final Output:**

```
4
9
```

```
    printNodesDown(root->right, k - 1);
}

// Function to initiate printing nodes at distance k from a given node value
void nodesAtDistanceK(Node* root, int node, int k) {
    nodesAtDistanceKWithRootDistance(root, node, k);
}

int main() {
    // Hardcoded tree construction
    Node* root = new Node(2);
    root->left = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(8);
    root->left->left->left = new Node(4);
    root->right = new Node(9);
    root->right->right = new Node(2);
    root->right->right->left = new Node(6);

    // Call function to print nodes at distance k from node with value 3
    nodesAtDistanceK(root, 3, 2);

    // Clean up dynamically allocated memory
    delete root->right->right->left;
    delete root->right->right;
    delete root->right;
    delete root->left->left->left;
    delete root->left->left;
    delete root->left->right;
    delete root->left;
    delete root;

    return 0;
}
```

4
9

Size,Sum,Max,Min,Height in C++

```
#include <iostream>
#include <algorithm>
#include <climits>// for std::max
using namespace std;

// Definition of a binary tree node
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int data, Node* left = nullptr, Node* right =
nullptr) {
        this->data = data;
        this->left = left;
        this->right = right;
    }
};

// Function to calculate the size (number of nodes) of
the binary tree
int size(Node* node) {
    if (node == nullptr) {
        return 0;
    } else {
        return 1 + size(node->left) + size(node->right);
    }
}

// Function to calculate the sum of all nodes in the
binary tree
int sum(Node* node) {
    if (node == nullptr) {
        return 0;
    } else {
        int lsum = sum(node->left);
        int rsum = sum(node->right);
        return node->data + lsum + rsum;
    }
}

// Function to find the maximum value in the binary
tree
int max(Node* node) {
    if (node == nullptr) {
        return INT_MIN; // from <climits> for INT_MIN
    } else {
        int lmax = max(node->left);
        int rmax = max(node->right);
        return std::max(node->data, std::max(lmax,
rmax));
    }
}

// Function to calculate the height of the binary tree
int height(Node* node) {
    if (node == nullptr) {
        return -1;
    } else {
        int lh = height(node->left);
```

Binary Tree Structure:

```
        50
       /  \
     25     75
    / \      / \
  12   37   62  87
      /        \
     30          70
```

✅ Expected Outputs:

Function	Description	Output
size	Number of nodes	9
sum	Sum of all node values	448
max	Maximum value in the tree	87
height	Height of the tree (edges, not nodes)	3
display	Inorder traversal (left → root → right)	12 25 30 37 50 62 70 75 87

🧪 Let's go through function results step-by-step:

1. size(root):

- Total nodes = 9

2. sum(root):

= 50 + sum(25 subtree) + sum(75 subtree)
= 50 + (25 + 12 + 37 + 30) + (75 + 62 + 70 + 87)
= 50 + 104 + 294
= 448

3. max(root):

- Max in left subtree = max(25, 12, 37, 30) = 37
- Max in right subtree = max(75, 62, 70, 87) = 87
- Final max = max(50, 37, 87) = **87**

4. height(root):

- Longest path (e.g., 50 → 75 → 62 → 70) has 3 edges → height = **3**

5. display(root) (Inorder):

Left subtree (25): 12 25 30 37
Root: 50
Right subtree (75): 62 70 75 87
=> Full: 12 25 30 37 50 62 70 75 87

```
        int rh = height(node->right);
        return 1 + std::max(lh, rh);
    }
}

// Function to display the binary tree (inorder
traversal)
void display(Node* node) {
    if (node == nullptr) {
        return;
    }

    display(node->left);
    cout << node->data << " ";
    display(node->right);
}

int main() {
    // Hardcoded tree construction
    Node* root = new Node(50);
    root->left = new Node(25);
    root->left->left = new Node(12);
    root->left->right = new Node(37);
    root->left->right->left = new Node(30);
    root->right = new Node(75);
    root->right->left = new Node(62);
    root->right->left->right = new Node(70);
    root->right->right = new Node(87);

    // Calculating size, sum, max value, and height
    int treeSize = size(root);
    int treeSum = sum(root);
    int treeMax = max(root);
    int treeHeight = height(root);

    // Displaying results
    cout << "Size of the binary tree: " << treeSize <<
endl;
    cout << "Sum of all nodes in the binary tree: " <<
treeSum << endl;
    cout << "Maximum value in the binary tree: " <<
treeMax << endl;
    cout << "Height of the binary tree: " << treeHeight
<< endl;

    // Displaying the binary tree (inorder traversal)
    cout << "Inorder traversal of the binary tree:" <<
endl;
    display(root);
    cout << endl;

    // Clean up dynamically allocated memory
    delete root->right->left->right;
    delete root->right->left;
    delete root->right;
    delete root->left->right->left;
    delete root->left->right;
    delete root->left->left;
    delete root->left;
    delete root;

    return 0;
```

🖥 **Final Output (Console):**

```
Size of the binary tree: 9
Sum of all nodes in the binary tree: 448
Maximum value in the binary tree: 87
Height of the binary tree: 3
Inorder traversal of the binary tree:
12 25 30 37 50 62 70 75 87
```

}	
size of the binary tree: 9 Sum of all nodes in the binary tree: 448 Maximum value in the binary tree: 87 Height of the binary tree: 3 Inorder traversal of the binary tree: 12 25 30 37 50 62 70 75 87	

Tilt in C++

```
#include <iostream>
#include <cstdlib> // for abs function
using namespace std;

// Definition of a binary tree node
struct Node {
    int data;
    Node* left;
    Node* right;

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

// Function to display the binary tree (for debugging
purposes)
void display(Node* node) {
    if (node == nullptr) {
        return;
    }

    string str = "";
    str += (node->left == nullptr) ? "." : to_string(node-
>left->data);
    str += " <- " + to_string(node->data) + " -> ";
    str += (node->right == nullptr) ? "." :
to_string(node->right->data);
    cout << str << endl;

    display(node->left);
    display(node->right);
}

// Function to calculate the height of the binary tree
int height(Node* node) {
    if (node == nullptr) {
        return -1;
    }

    int lh = height(node->left);
    int rh = height(node->right);

    return max(lh, rh) + 1;
}

// Global variable to store the tilt of the entire tree
int tilt = 0;

// Function to calculate the tilt of the binary tree
int calculateTilt(Node* node) {
    if (node == nullptr) {
        return 0;
    }

    int ls = calculateTilt(node->left);
    int rs = calculateTilt(node->right);
```

Tree Structure:

```
       50
      /  \
    25    75
   / \   / \
  12  37 62  87
      /    \
    30     70
```

🧮 Dry Run with Tilt Values

Let’s go **bottom-up** and calculate each node’s tilt with its left and right subtree sums:

Node	Left Sum	Right Sum	Node Tilt = abs(L - R)
12	0	0	0
30	0	0	0
37	30	0	30
25	12	67 (37+30)	55
70	0	0	0
62	0	70	70
87	0	0	0
75	132	87	45
50	104	294	190

🔢 Total Tilt:

0 (12)
+ 0 (30)
+ 30 (37)
+ 55 (25)
+ 0 (70)
+ 70 (62)
+ 0 (87)
+ 45 (75)
+ 190 (50)
= **390**

✅ Output:

Tilt of the binary tree: 390

```
    int ltilt = abs(ls - rs);
    tilt += ltilt;

    int sum = ls + rs + node->data;
    return sum;
}

int main() {
    // Hardcoded tree construction
    Node* root = new Node(50);
    root->left = new Node(25);
    root->left->left = new Node(12);
    root->left->right = new Node(37);
    root->left->right->left = new Node(30);
    root->right = new Node(75);
    root->right->left = new Node(62);
    root->right->left->right = new Node(70);
    root->right->right = new Node(87);

    // Calculate the tilt of the tree
    calculateTilt(root);

    // Output the tilt value
    cout << "Tilt of the binary tree: " << tilt << endl;

    // Clean up dynamically allocated memory
    delete root->left->left;
    delete root->left->right->left;
    delete root->left->right;
    delete root->left;
    delete root->right->left->right;
    delete root->right->left;
    delete root->right->right;
    delete root->right;
    delete root;

    return 0;
}
```

Tilt of the binary tree: 390

All single child parent in C++	

```
#include <iostream>
#include <vector>

using namespace std;

// Definition of a Node in the Binary Tree
struct Node {
    int val;
    Node* left;
    Node* right;

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

// Function to find all nodes with exactly one child
void exactlyOneChild(Node* root, vector<int>& ans) {
    if (root == nullptr || (root->left == nullptr && root->right == nullptr)) {
        return;
    }

    if (root->left == nullptr || root->right == nullptr) {
        ans.push_back(root->val);
    }

    exactlyOneChild(root->left, ans);
    exactlyOneChild(root->right, ans);
}

// Wrapper function for exactlyOneChild
vector<int> exactlyOneChild(Node* root) {
    vector<int> res;
    exactlyOneChild(root, res);
    return res;
}
int main() {
    // Constructing the example binary tree
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->left->left = new Node(5);

    // Finding nodes with exactly one child
    vector<int> ans = exactlyOneChild(root);

    // Printing the result
    cout << "Nodes with exactly one child: ";
    for (int num : ans) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```

Tree Structure:

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

🔍 Nodes with Exactly One Child

We traverse and look for nodes that have **only one** non-null child:

Node	Left Child	Right Child	Exactly One Child?	Added to ans?
1	2	3	✕ (has both)	✕
2	4	nullptr	✓	✓ → 2
4	5	nullptr	✓	✓ → 4
5	nullptr	nullptr	✕ (no children)	✕
3	nullptr	nullptr	✕ (no children)	✕

✓ Final Output:

Nodes with exactly one child: 2 4

Nodes with exactly one child: 2 4

BottomView in C++

```
#include <iostream>
#include <map>
#include <queue>
#include <vector>
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;
    }
};

vector<int> bottomView(TreeNode* root) {
    vector<int> bottomViewNodes;
    if (!root) {
        return bottomViewNodes;
    }

    // TreeMap equivalent in C++ is std::map
    map<int, int> map;
    queue<pair<TreeNode*, int>> q;
    q.push({root, 0});

    while (!q.empty()) {
        auto front = q.front();
        q.pop();
        TreeNode* node = front.first;
        int hd = front.second;

        // Update the map with current node's value at
its horizontal distance
        map[hd] = node->val;

        // Enqueue left child with horizontal distance hd -
1
        if (node->left) {
            q.push({node->left, hd - 1});
        }

        // Enqueue right child with horizontal distance
hd + 1
        if (node->right) {
            q.push({node->right, hd + 1});
        }
    }

    // Populate bottomViewNodes with values from map
    for (const auto& pair : map) {
        bottomViewNodes.push_back(pair.second);
    }

    return bottomViewNodes;
}
```

Binary Tree Structure:

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

■ Step-by-Step Dry Run Table

We'll simulate the level order traversal using a queue storing (node, horizontal_distance) and map hd → node->val.

Step	Queue Content	Popped Node	HD	Map After Step
1	(1, 0)	1	0	{0 → 1}
2	(2, -1), (3, 1)	2	-1	{-1 → 2, 0 → 1}
3	(3, 1), (4, -2), (5, 0)	3	1	{-1 → 2, 0 → 1, 1 → 3}
4	(4, -2), (5, 0), (6, 0), (7, 2)	4	-2	{-2 → 4, -1 → 2, 0 → 1, 1 → 3}
5	(5, 0), (6, 0), (7, 2)	5	0	{-2 → 4, -1 → 2, 0 → 5, 1 → 3}
6	(6, 0), (7, 2)	6	0	{-2 → 4, -1 → 2, 0 → 6, 1 → 3}
7	(7, 2)	7	2	{-2 → 4, -1 → 2, 0 → 6, 1 → 3, 2 → 7}

🟢 Final Bottom View:

Take values from the map in order of keys (i.e., horizontal distance):

-2 → 4
-1 → 2
 0 → 6
 1 → 3
 2 → 7

▼ Output:

4 2 6 3 7

```
// Utility function to create a new node
TreeNode* newNode(int key) {
    TreeNode* node = new TreeNode(key);
    return node;
}

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

    vector<int> result = bottomView(root);

    // Print the result
    for (int value : result) {
        cout << value << " ";
    }
    cout << endl;

    // Memory cleanup (optional in this example)
    // You may need to delete nodes if not using smart
pointers
    return 0;
}
```

```
4 2 6 3 7
```

Diagonal Order in C++

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

// TreeNode structure definition
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) {
        val = x;
        left = nullptr;
        right = nullptr;
    }
};

// Function to perform diagonal order traversal of a binary tree
vector<vector<int>> diagonalOrder(TreeNode* root) {
    vector<vector<int>> ans;
    if (root == nullptr) return ans;

    queue<TreeNode*> que;
    que.push(root);

    while (!que.empty()) {
        int size = que.size();
        std::vector<int> smallAns;

        while (size--) {
            TreeNode* node = que.front();
            que.pop();

            while (node != nullptr) {
                smallAns.push_back(node->val);

                if (node->left) que.push(node->left);

                node = node->right;
            }
        }

        ans.push_back(smallAns);
    }

    return ans;
}

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->right->left = new TreeNode(6);
    root->right->right = new TreeNode(7);

    // Calling diagonalOrder function and printing
```

Tree Structure:

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

◆ **Diagonal View Intuition:**

- Diagonal lines go **from top-right to bottom-left**, i.e., every time you go to .right, you stay on the same diagonal.
- Every time you go to .left, you move to the **next diagonal**.

✅ **Dry Run Table:**

We'll simulate the queue and how the diagonal groups are formed.

Iteration	Queue (Before)	Extracted	Collected (Diagonal)	Queue (After pushing lefts)
1	[1]	1 → 3 → 7	[1, 3, 7]	[2, 6]
2	[2, 6]	2 → 5	[2, 5]	[4]
3	[4]	4	[4]	[]

◆ **Final Output:**

Diagonal Order Traversal:
1 3 7
2 5
4

💡 **Breakdown:**

- Diagonal 0 → 1 → 3 → 7
- Diagonal 1 → 2 → 5
- Diagonal 2 → 4

```
the result
    vector<vector<int>> ans =
diagonalOrder(root);

    cout << "Diagonal Order Traversal:\n";
    for (const auto level : ans) {
        for (int num : level) {
            cout << num << " ";
        }
        cout << "\n";
    }

    // Deallocating memory to avoid memory leaks
    delete root->right->right;
    delete root->right->left;
    delete root->left->right;
    delete root->left->left;
    delete root->right;
    delete root->left;
    delete root;

    return 0;
}
```

```
Diagonal Order Traversal:
1 3 7
2 5 6
4
```

Iterative tree operations in C++

```cpp
#include <iostream>
#include <queue>
#include <climits> // for INT_MIN and INT_MAX

using namespace std;

// Definition of a Node in the Binary Tree
struct Node {
    int val;
    Node* left;
    Node* right;

    Node(int x) {
        val = x;
        left = nullptr;
        right = nullptr;
    }
};

// Function to calculate the height of the tree using
BFS (level-order traversal)
int getHeight(Node* root) {
    if (root == nullptr) return 0;

    queue<Node*> q;
    q.push(root);
    int height = 0;

    while (!q.empty()) {
        int levelSize = q.size();
        height++;
        for (int i = 0; i < levelSize; i++) {
            Node* node = q.front();
            q.pop();
            if (node->left != nullptr) q.push(node->left);
            if (node->right != nullptr) q.push(node->right);
        }
    }

    return height;
}

// Function to count the number of nodes in the tree
using BFS (level-order traversal)
int getNodeCount(Node* root) {
    if (root == nullptr) return 0;

    queue<Node*> q;
    q.push(root);
    int count = 0;

    while (!q.empty()) {
        Node* node = q.front();
        q.pop();
        count++;
        if (node->left != nullptr) q.push(node->left);
        if (node->right != nullptr) q.push(node->right);
    }

    return count;
}
```

Tree Structure:

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

◆ **Function: getHeight(root)**

This uses **level-order traversal** (BFS).

Level	Nodes at Level	Height So Far
1	1	1
2	2, 3	2
3	4, 5	3

✅ **Result: 3**

◆ **Function: getNodeCount(root)**

Counts nodes using BFS:

Step	Node Processed	Count	Queue
1	1	1	2, 3
2	2	2	3, 4, 5
3	3	3	4, 5
4	4	4	5
5	5	5	

✅ **Result: 5**

◆ **Function: getMax(root)**

Finds maximum using BFS:

Step	Node Processed	Max So Far
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5 ✅

✅ **Result: 5**

◆ **Function: getMin(root)**

Finds minimum using BFS:

```
// Function to find the maximum value in the tree
using BFS (level-order traversal)
int getMax(Node* root) {
    if (root == nullptr) throw invalid_argument("Tree is
empty");

    queue<Node*> q;
    q.push(root);
    int maxValue = INT_MIN;

    while (!q.empty()) {
        Node* node = q.front();
        q.pop();
        maxValue = max(maxValue, node->val);
        if (node->left != nullptr) q.push(node->left);
        if (node->right != nullptr) q.push(node->right);
    }

    return maxValue;
}

// Function to find the minimum value in the tree
using BFS (level-order traversal)
int getMin(Node* root) {
    if (root == nullptr) throw invalid_argument("Tree is
empty");

    queue<Node*> q;
    q.push(root);
    int minValue = INT_MAX;

    while (!q.empty()) {
        Node* node = q.front();
        q.pop();
        minValue = min(minValue, node->val);
        if (node->left != nullptr) q.push(node->left);
        if (node->right != nullptr) q.push(node->right);
    }

    return minValue;
}

int main() {
    // Constructing the example binary tree
    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);

    // Using the functions to demonstrate the
functionality
    cout << "Height of the tree: " << getHeight(root) <<
endl;
    cout << "Number of nodes in the tree: " <<
getNodeCount(root) << endl;

    try {
        cout << "Maximum value in the tree: " <<
getMax(root) << endl;
        cout << "Minimum value in the tree: " <<
```

Step	Node Processed	Min So Far
1	1	1 ✅
2	2	1
3	3	1
4	4	1
5	5	1

✅ **Result: 1**

✅ **Final Output:**

Height of the tree: 3
Number of nodes in the tree: 5
Maximum value in the tree: 5
Minimum value in the tree: 1

```
getMin(root) << endl;
    } catch (const exception& e) {
        cerr << e.what() << endl;
    }

    return 0;
}
```

```
Height of the tree: 3
Number of nodes in the tree: 5
Maximum value in the tree: 5
Minimum value in the tree: 1
```

Top View in C++

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <map>
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;
    }
};

// Function to compute the top view of a binary tree
vector<int> topView(TreeNode* root) {
    vector<int> topViewNodes;
    if (!root) {
        return topViewNodes;
    }

    map<int, int> hdMap; // Horizontal Distance Map (hd -> node value)
    queue<pair<TreeNode*, int>> q; // Queue to store nodes and their horizontal distance

    q.push({root, 0}); // Start with the root node at horizontal distance 0

    while (!q.empty()) {
        TreeNode* node = q.front().first;
        int hd = q.front().second;
        q.pop();

        // If this horizontal distance is not already in the map, add the node value
        if (hdMap.find(hd) == hdMap.end()) {
            hdMap[hd] = node->val;
        }

        // Enqueue left and right children with updated horizontal distances
        if (node->left) {
            q.push({node->left, hd - 1});
        }

        if (node->right) {
            q.push({node->right, hd + 1});
        }
    }

    // Extract values from the map in order of horizontal distance
    for (const auto& pair : hdMap) {
        topViewNodes.push_back(pair.second);
    }
```

Constructed Binary Tree:

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

📘 Step-by-Step Traversal Table (Level Order with HD)

We'll perform a BFS traversal and track each node with its **Horizontal Distance (HD)** from root.

Step	Queue Content	Popped Node	HD	hdMap Before	hdMap After
1	(1, 0)	1	0	{}	{0: 1}
2	(2, -1), (3, 1)	2	-1	{0: 1}	{-1: 2, 0: 1}
3	(3, 1), (4, 0)	3	1	{-1: 2, 0: 1}	{-1: 2, 0: 1, 1: 3}
4	(4, 0), (5, 1)	4	0	already filled	(no change)
5	(5, 1), (6, 2)	5	1	already filled	(no change)
6	(6, 2)	6	2	{-1: 2, 0: 1, 1: 3}	{… , 2: 6}

🟢 Final Map (hdMap) Sorted by HD:

-1 → 2
0 → 1
1 → 3
2 → 6

✅ Output (Top View):

2 1 3 6

```
    return topViewNodes;
}

// Utility function to create a new node
TreeNode* newNode(int key) {
    TreeNode* node = new TreeNode(key);
    return node;
}

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

    // Get the top view of the binary tree
    vector<int> result = topView(root);

    // Print the top view of the binary tree
    cout << "Top view of the binary tree:" << endl;
    for (int nodeValue : result) {
        cout << nodeValue << " ";
    }
    cout << endl;

    // Clean up memory (optional in this example)
    // You may need to delete nodes if not using smart
pointers
    return 0;
}
```

```
Top view of the binary tree:
2 1 3 6
```

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++

```cpp
#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++

```
#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++

```cpp
#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**:
	    ○ Push parent's value down to children if sum of children < parent.
	2. **Postorder Phase**:
	    ○ 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

<table>
<tr><td>

```
    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;
}
```

</td><td></td></tr>
<tr><td colspan="2">The Max Path Sum for this tree is 42</td></tr>
</table>

Morris traversal in C++

```
#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

```
        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;
}
```

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

``
        }
        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;
}
``	

The path is: 1 2 5 7	