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 = nodesAtDistanceKWithRootDistance(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++

```cpp
#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++

```cpp
#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