All single child parent in C++

```cpp
#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++

```cpp
#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++

```cpp
#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:

<pre>// 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: " <<</pre>
	<table><tr><th>Step</th><th>Node Processed</th><th>Min So Far</th></tr><tr><td>1</td><td>1</td><td>1 ✅</td></tr><tr><td>2</td><td>2</td><td>1</td></tr><tr><td>3</td><td>3</td><td>1</td></tr><tr><td>4</td><td>4</td><td>1</td></tr><tr><td>5</td><td>5</td><td>1</td></tr></table>✅ **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

<pre>getMin(root) << endl;
    } catch (const exception& e) {
        cerr << e.what() << endl;
    }
 	
    return 0;
}</pre>	

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
```