

Height in C++

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

using namespace std;

// Node class definition
class Node {
public:
    int data;
    vector<Node*> children;

    Node(int val) {
        data = val;
    }
};

// Function to construct the tree from the given array
Node* construct(vector<int>& arr) {
    Node* root = nullptr;
    stack<Node*> st;

    for (int i = 0; i < arr.size(); ++i) {
        if (arr[i] == -1) {
            st.pop();
        } else {
            Node* t = new Node(arr[i]);

            if (!st.empty()) {
                st.top()->children.push_back(t);
            } else {
                root = t;
            }

            st.push(t);
        }
    }

    return root;
}

// Function to calculate the height of the tree
int height(Node* node) {
    if (node->children.empty()) {
        return 0;
    }

    int maxChildHeight = 0;
    for (Node* child : node->children) {
        int childHeight = height(child);
        if (childHeight > maxChildHeight) {
            maxChildHeight = childHeight;
        }
    }

    return maxChildHeight + 1;
}

// Main function
int main() {
    vector<int> arr = {10, 20, -1, 30, 50, -1, 60, -1, -1, 40, -1, -1};
```

Input Array:

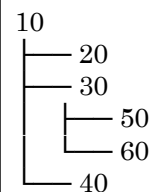
{10, 20, -1, 30, 50, -1, 60, -1, -1, 40, -1, -1}

✂ Tree Construction (construct function):

We use a **stack** to maintain the current path in the tree. When we encounter -1, we pop a node from the stack (finished with that node's children). Here's a **step-by-step construction** of the tree:

Step	arr[i]	Stack Top	Action	Tree Change
0	10	—	Create node(10), push	root = 10
1	20	10	Add 20 as child to 10, push	10 → 20
2	-1	20	Pop 20	
3	30	10	Add 30 as child to 10, push	10 → 30
4	50	30	Add 50 as child to 30, push	30 → 50
5	-1	50	Pop 50	
6	60	30	Add 60 as child to 30, push	30 → 60
7	-1	60	Pop 60	
8	-1	30	Pop 30	
9	40	10	Add 40 as child to 10, push	10 → 40
10	-1	40	Pop 40	
11	-1	10	Pop 10 (tree complete)	

✓ Final Tree:



🌲 Height Calculation:

The **height** of a tree is the number of edges in the longest path from the root to a leaf node.

We traverse each subtree and compute the max height:

- Leaf nodes like 20, 50, 60, and 40 → height = 0
- Node 30 has children 50 and 60 → height = 1
- Root 10 has children:
 - 20 → 0
 - 30 → 1

<pre>40, -1, -1}; Node* root = construct(arr); int h = height(root); cout << h << endl; return 0; }</pre>	<ul style="list-style-type: none">○ $40 \rightarrow 0$<ul style="list-style-type: none">\rightarrow max child height = 1\rightarrow root height = $1 + 1 = 2$ <p>✔ Final Height: 2</p> <p>✔ Output:</p> <p>2</p>
2	

Is Symmetric in C++																																																																												
<pre>#include <iostream> #include <vector> #include <stack> using namespace std; // Node class definition class Node { public: int data; vector<Node*> children; Node(int val) { data = val; } }; // Function to construct the tree // from the given array Node* construct(vector<int>& arr) { Node* root = nullptr; stack<Node*> st; for (int i = 0; i < arr.size(); ++i) { if (arr[i] == -1) { st.pop(); } else { Node* t = new Node(arr[i]); if (!st.empty()) { st.top()- >children.push_back(t); } else { root = t; } st.push(t); } } return root; } // Function to check if two trees // are mirrors of each other bool areMirror(Node* n1, Node* n2) { if (n1->children.size() != n2- >children.size()) { return false; } for (int i = 0; i < n1- >children.size(); ++i) { int j = n1->children.size() - 1 - i; Node* c1 = n1->children[i]; Node* c2 = n2->children[j];</pre>	<div>Tree Structure from Input</div> <div><pre>graph TD 10 --- 20 10 --- 30 10 --- 40 20 --- 50 20 --- 60 30 --- 70 30 --- 80 30 --- 90 40 --- 100 40 --- 110</pre></div> <div>Tabular Dry Run of are Mirror (node1, node2)</div> <table><tr><th>Step</th><th>node1->data</th><th>node2->data</th><th>Children Count Match</th><th>Comparing Child Pair</th><th>Recursive Call</th><th>Result</th></tr><tr><td>1</td><td>10</td><td>10</td><td>✔ Yes (3 children)</td><td>Compare 20 & 40</td><td>areMirror(20, 40)</td><td>proceeds</td></tr><tr><td>2</td><td>20</td><td>40</td><td>✔ Yes (2 children)</td><td>Compare 50 & 110</td><td>areMirror(50, 110)</td><td>✔ true</td></tr><tr><td>3</td><td>50</td><td>110</td><td>✔ Yes (0 children)</td><td>-</td><td>leaf nodes</td><td>✔ true</td></tr><tr><td>4</td><td>20</td><td>40</td><td>-</td><td>Compare 60 & 100</td><td>areMirror(60, 100)</td><td>✔ true</td></tr><tr><td>5</td><td>60</td><td>100</td><td>✔ Yes (0 children)</td><td>-</td><td>leaf nodes</td><td>✔ true</td></tr><tr><td>6</td><td>20 & 40</td><td>done</td><td>All children matched</td><td>-</td><td>return to previous</td><td>✔ true</td></tr><tr><td>7</td><td>10</td><td>10</td><td>-</td><td>Compare 30 & 30 (middle node)</td><td>areMirror(30, 30)</td><td>proceeds</td></tr><tr><td>8</td><td>30</td><td>30</td><td>✔ Yes (3 children)</td><td>Compare 70 & 90</td><td>areMirror(70, 90)</td><td>✔ true</td></tr><tr><td>9</td><td>70</td><td>90</td><td>✔ Yes (0 children)</td><td>-</td><td>leaf nodes</td><td>✔ true</td></tr></table>						Step	node1->data	node2->data	Children Count Match	Comparing Child Pair	Recursive Call	Result	1	10	10	✔ Yes (3 children)	Compare 20 & 40	areMirror(20, 40)	proceeds	2	20	40	✔ Yes (2 children)	Compare 50 & 110	areMirror(50, 110)	✔ true	3	50	110	✔ Yes (0 children)	-	leaf nodes	✔ true	4	20	40	-	Compare 60 & 100	areMirror(60, 100)	✔ true	5	60	100	✔ Yes (0 children)	-	leaf nodes	✔ true	6	20 & 40	done	All children matched	-	return to previous	✔ true	7	10	10	-	Compare 30 & 30 (middle node)	areMirror(30, 30)	proceeds	8	30	30	✔ Yes (3 children)	Compare 70 & 90	areMirror(70, 90)	✔ true	9	70	90	✔ Yes (0 children)	-	leaf nodes	✔ true
Step	node1->data	node2->data	Children Count Match	Comparing Child Pair	Recursive Call	Result																																																																						
1	10	10	✔ Yes (3 children)	Compare 20 & 40	areMirror(20, 40)	proceeds																																																																						
2	20	40	✔ Yes (2 children)	Compare 50 & 110	areMirror(50, 110)	✔ true																																																																						
3	50	110	✔ Yes (0 children)	-	leaf nodes	✔ true																																																																						
4	20	40	-	Compare 60 & 100	areMirror(60, 100)	✔ true																																																																						
5	60	100	✔ Yes (0 children)	-	leaf nodes	✔ true																																																																						
6	20 & 40	done	All children matched	-	return to previous	✔ true																																																																						
7	10	10	-	Compare 30 & 30 (middle node)	areMirror(30, 30)	proceeds																																																																						
8	30	30	✔ Yes (3 children)	Compare 70 & 90	areMirror(70, 90)	✔ true																																																																						
9	70	90	✔ Yes (0 children)	-	leaf nodes	✔ true																																																																						

<pre> if (!areMirror(c1, c2)) { return false; } } return true; } // Function to check if a tree is symmetric bool IsSymmetric(Node* node) { return areMirror(node, node); } // Main function int main() { vector<int> arr = {10, 20, 50, -1, 60, -1, -1, 30, 70, -1, 80, -1, 90, -1, -1, 40, 100, -1, 110, -1, -1, -1}; Node* root = construct(arr); bool sym = IsSymmetric(root); cout << boolalpha << sym << endl; return 0; } </pre>				children)			
	10	30	30	-	Compare 80 & 80	areMirror(80, 80)	✔ true
	11	80	80	✔ Yes (0 children)	-	leaf nodes	✔ true
	12	30	30	-	Compare 90 & 70	areMirror(90, 70)	✔ true
	13	90	70	✔ Yes (0 children)	-	leaf nodes	✔ true
	14	30 & 30	done	All children matched	-	return to previous	✔ true
	15	10	10	-	Compare 40 & 20	already compared in step 1	✔ true
	16	10 & 10	done	All pairs matched	-	final result	✔ true
<p>✔ Final Result:</p> <p>true</p>							

true

Level Order in C++

```
#include <iostream>
#include <vector>
#include <queue>
#include <stack>

using namespace std;

// Node class definition
class Node {
public:
    int data;
    vector<Node*> children;

    Node(int val) {
        data = val;
    }
};

// Function to construct the tree from the given array
Node* construct(vector<int>& arr) {
    Node* root = nullptr;
    stack<Node*> st;

    for (int i = 0; i < arr.size(); ++i) {
        if (arr[i] == -1) {
            st.pop();
        } else {
            Node* t = new Node(arr[i]);

            if (!st.empty()) {
                st.top()->children.push_back(t);
            } else {
                root = t;
            }

            st.push(t);
        }
    }

    return root;
}

// Function for level order traversal
void levelOrder(Node* node) {
    if (!node)
        return;

    queue<Node*> q;
    q.push(node);

    while (!q.empty()) {
        Node* f = q.front();
        q.pop();

        cout << f->data << " ";

        for (Node* child : f->children) {
            q.push(child);
        }
    }
}
```

Input Array:

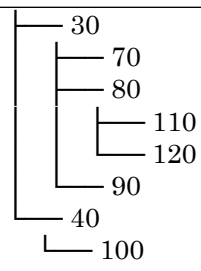
{24, 10, 20, 50, -1, 60, -1, -1, 30, 70, -1, 80, 110, -1, 120, -1, -1, 90, -1, -1, 40, 100, -1, -1, -1}

🔧 **Tree Construction Process (construct() function):**

Using a **stack**, we construct the tree as follows:

Step	arr[i]	Action	Stack Top (parent)	Node Created	Description
0	24	Create root, push to stack	—	24	Root node
1	10	Create, add to 24, push	24	10	24 → 10
2	20	Create, add to 10, push	10	20	10 → 20
3	50	Create, add to 20, push	20	50	20 → 50
4	-1	Pop 50	20	—	50 done
5	60	Create, add to 20, push	20	60	20 → 60
6	-1	Pop 60	20	—	60 done
7	-1	Pop 20	10	—	20 done
8	30	Create, add to 10, push	10	30	10 → 30
9	70	Create, add to 30, push	30	70	30 → 70
10	-1	Pop 70	30	—	70 done
11	80	Create, add to	30	80	30 → 80

<pre> cout << "." << endl; } // Main function int main() { vector<int> arr = {24, 10, 20, 50, -1, 60, -1, -1, 30, 70, -1, 80, 110, -1, 120, -1, -1, 90, -1, -1, 40, 100, -1, -1, -1}; Node* root = construct(arr); levelOrder(root); return 0; } </pre>			30, push			
	12	110	Create, add to 80, push	80	110	80 → 110
	13	-1	Pop 110	80	—	110 done
	14	120	Create, add to 80, push	80	120	80 → 120
	15	-1	Pop 120	80	—	120 done
	16	-1	Pop 80	30	—	80 done
	17	90	Create, add to 30, push	30	90	30 → 90
	18	-1	Pop 90	30	—	90 done
	19	-1	Pop 30	10	—	30 done
	20	40	Create, add to 10, push	10	40	10 → 40
	21	100	Create, add to 40, push	40	100	40 → 100
	22	-1	Pop 100	40	—	100 done
	23	-1	Pop 40	10	—	40 done
	24	-1	Pop 10	24	—	10 done
	✓ Final tree root is 24					
	♣ Tree Structure (for Visualization)					
	<pre> 24 ├── 10 │ ├── 20 │ │ ├── 50 │ │ └── 60 </pre>					



🌀 **Level Order Traversal Output**

Traverses level-by-level:

Queue Contents	Output
24	24
10	10
20, 30, 40	20
50, 60, 70, 80, 90, 100	30
—	40
—	50
—	60
—	70
110, 120	80
—	90
—	100
—	110
—	120

✔ **Final Output:**

24 10 20 30 40 50 60 70 80 90 100 110 120 .

24 10 20 30 40 50 60 70 80 90 100 110 120 .

PrePostorder Traversal in C++

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

// Node structure definition
struct Node {
    int data;
    vector<Node*> children;
};

// Function to display the tree structure
void display(Node* node) {
    cout << node->data << " -> ";
    for (Node* child : node->children) {
        cout << child->data << " , ";
    }
    cout << "." << endl;

    for (Node* child : node->children) {
        display(child);
    }
}

// Function to construct the tree from an array
Node* construct(vector<int>& arr) {
    Node* root = nullptr;
    vector<Node*> st;

    for (int i = 0; i < arr.size(); ++i) {
        if (arr[i] == -1) {
            st.pop_back();
        } else {
            Node* t = new Node();
            t->data = arr[i];

            if (!st.empty()) {
                st.back()->children.push_back(t);
            } else {
                root = t;
            }

            st.push_back(t);
        }
    }

    return root;
}

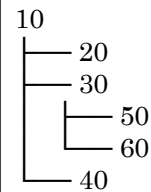
// Function to perform pre-order, post-order, and edge
printing traversals
void traversals(Node* node) {
    // Print Node Pre
    cout << "Node Pre " << node->data << endl;

    // Print Edge Pre
    for (Node* child : node->children) {
        cout << "Edge Pre " << node->data << "--" <<
child->data << endl;
        traversals(child);
    }
    cout << "Edge Post " << node->data << "--" <<
```

Input Array:

{10, 20, -1, 30, 50, -1, 60, -1, -1, 40, -1, -1}

✔ **Constructed Tree:**



🔄 Dry Run Table for traversals()

Step	Current Node	Action Type	Output
1	10	Node Pre	Node Pre 10
2	10 → 20	Edge Pre	Edge Pre 10--20
3	20	Node Pre	Node Pre 20
4	20	Node Post	Node Post 20
5	10 ← 20	Edge Post	Edge Post 10--20
6	10 → 30	Edge Pre	Edge Pre 10--30
7	30	Node Pre	Node Pre 30
8	30 → 50	Edge Pre	Edge Pre 30--50
9	50	Node Pre	Node Pre 50
10	50	Node Post	Node Post 50
11	30 ← 50	Edge Post	Edge Post 30--50
12	30 → 60	Edge Pre	Edge Pre 30--60
13	60	Node Pre	Node Pre 60
14	60	Node Post	Node Post 60
15	30 ← 60	Edge Post	Edge Post 30--60
16	30	Node Post	Node Post 30
17	10 ← 30	Edge Post	Edge Post 10--30
18	10 → 40	Edge Pre	Edge Pre 10--40
19	40	Node Pre	Node Pre 40
20	40	Node Post	Node Post 40
21	10 ← 40	Edge Post	Edge Post 10--40

<pre>child->data << endl; } // Print Node Post cout << "Node Post " << node->data << endl; } int main() { vector<int> arr = {10, 20, -1, 30, 50, -1, 60, -1, -1, 40, -1, -1}; Node* root = construct(arr); // Perform pre-order, post-order, and edge printing traversals traversals(root); // Clean up memory (not necessary in this simple example but good practice) // You would typically have a function to delete the tree return 0; }</pre>	<table><tr><td></td><td></td><td></td><td>-40</td></tr><tr><td>22</td><td>10</td><td>Node Post</td><td>Node Post 10</td></tr></table> <p>Final Output (as it would appear on console):</p> <pre>Node Pre 10 Edge Pre 10--20 Node Pre 20 Node Post 20 Edge Post 10--20 Edge Pre 10--30 Node Pre 30 Edge Pre 30--50 Node Pre 50 Node Post 50 Edge Post 30--50 Edge Pre 30--60 Node Pre 60 Node Post 60 Edge Post 30--60 Node Post 30 Edge Post 10--30 Edge Pre 10--40 Node Pre 40 Node Post 40 Edge Post 10--40 Node Post 10</pre>				-40	22	10	Node Post	Node Post 10
			-40						
22	10	Node Post	Node Post 10						
<pre>Node Pre 10 Edge Pre 10--20 Node Pre 20 Node Post 20 Edge Post 10--20 Edge Pre 10--30 Node Pre 30 Edge Pre 30--50 Node Pre 50 Node Post 50 Edge Post 30--50 Edge Pre 30--60 Node Pre 60 Node Post 60 Edge Post 30--60 Node Post 30 Edge Post 10--30 Edge Pre 10--40 Node Pre 40 Node Post 40 Edge Post 10--40 Node Post 10</pre>									

Size in C++

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

// Node structure definition
struct Node {
    int data;
    vector<Node*> children;
};

// Function to display the tree structure
void display(Node* node) {
    cout << node->data << " -> ";
    for (Node* child : node->children) {
        cout << child->data << ", ";
    }
    cout << "." << endl;

    for (Node* child : node->children) {
        display(child);
    }
}

// Function to construct the tree from array
representation
Node* construct(int arr[], int n) {
    Node* root = nullptr;
    vector<Node*> st;

    for (int i = 0; i < n; ++i) {
        if (arr[i] == -1) {
            st.pop_back();
        } else {
            Node* t = new Node();
            t->data = arr[i];

            if (!st.empty()) {
                st.back()->children.push_back(t);
            } else {
                root = t;
            }

            st.push_back(t);
        }
    }

    return root;
}

// Function to calculate the size of the tree
int size(Node* node) {
    int sz = 0;
    for (Node* child : node->children) {
        sz += size(child);
    }
    return 1 + sz;
}

int main() {
    // Static data representing the tree
    int arr[] = {10, 20, -1, 30, 50, -1, 60, -1, -1, 40, -1, -1};
```

Input Array:

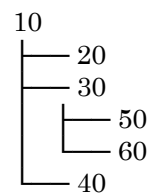
{10, 20, -1, 30, 50, -1, 60, -1, -1, 40, -1, -1}

✳ Tree Construction Dry Run

This array uses -1 to indicate the end of children for a node. We construct the tree using a vector (acting like a stack).

Step	arr[i]	Stack Top	Action	Tree Changes
0	10	—	New Node(10), push	root = 10
1	20	10	Add 20 as child to 10, push	10 → 20
2	-1	20	Pop 20	
3	30	10	Add 30 as child to 10, push	10 → 30
4	50	30	Add 50 as child to 30, push	30 → 50
5	-1	50	Pop 50	
6	60	30	Add 60 as child to 30, push	30 → 60
7	-1	60	Pop 60	
8	-1	30	Pop 30	
9	40	10	Add 40 as child to 10, push	10 → 40
10	-1	40	Pop 40	
11	-1	10	Pop 10	Done

✓ Tree Structure:



Let's apply it:

- size(20) = 1
- size(50) = 1
- size(60) = 1
- size(30) = 1 (self) + size(50) + size(60) =

<pre> int n = sizeof(arr) / sizeof(arr[0]); // Construct the tree Node* root = construct(arr, n); // Calculate the size of the tree int sz = size(root); cout << sz << endl; // Output should be 6 // Display the tree structure (optional) // display(root); return 0; } </pre>	<p> $1 + 1 + 1 = 3$ • size(40) = 1 • size(10) = 1 (self) + size(20) + size(30) + size(40) = 1 + 1 + 3 + 1 = 6 </p>
6	