



ChatGPT

Explore GPTs

Yesterday

Top View Binary Tree

...

Village Population Literacy Calcul

Autorickshaw Fare Converter

Fixing Tree Construction Code

Morris Preorder Traversal Iterativ

Password Length Slider Update

React Password Generator Code

Password Generator Functionality

useCallback in React

New chat

Password Generator Component

Left View of Binary Tree



The "Top View of a Binary Tree" refers to the nodes visible when the tree is viewed from the top. To solve the problem in C++, you can follow these steps:

1. **Horizontal Distance (HD):** Each node in the tree has a horizontal distance from the root. The root has an HD of 0. The HD of a left child is HD of the parent minus 1, and the HD of a right child is HD of the parent plus 1.
2. **Use a Map:** Maintain a map where the key is the HD and the value is the first node at that HD (as you go level-wise).
3. **Level Order Traversal:** Use level-order traversal (BFS) to ensure that you get the first node for each horizontal distance.

C++ Code to Implement Top View of a Binary Tree:

cpp Copy code

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

// Define the structure of a tree node
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) {
        data = val;
        left = right = NULL;
    }
};

// Function to print the top view of the binary tree
void topView(Node* root) {
    if (root == NULL)
        return;

    // Create a map to store the first node at each horizontal distance
    map<int, int> topNodeMap;

    // Queue for level-order traversal; stores pairs of node and its horizontal distance
    queue<pair<Node*, int>> q;

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

    while (!q.empty()) {
        pair<Node*, int> curr = q.front();
        q.pop();

        Node* node = curr.first;
        int hd = curr.second;

        // If the horizontal distance is not already present in the map, insert it
        if (topNodeMap.find(hd) == topNodeMap.end()) {
            topNodeMap[hd] = node->data;
        }

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

    // Print the top view from the map
    for (auto it : topNodeMap) {
        cout << it.second << " ";
    }
}

int main() {
    // 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);    ↓
root->right->left = new Node(6);
root->right->right = new Node(7);

cout << "Top view of the binary tree is: ";
topView(root);

return 0;
}

```

Explanation:

- The `topView` function uses a `map` to store the first node encountered at each horizontal distance (HD).
- The `queue` stores pairs of nodes and their corresponding HDs during the level-order traversal.
- The map ensures that we only consider the first node at each HD, which corresponds to the top view.
- Finally, the map is printed in increasing order of the HD to display the top view from left to right.

Example Output:

For the binary tree:

```

markdown
Copy code

1
/
2 3
/ \ /
4 5 6 7

```

The top view will be:

```

Copy code

4 2 1 3 7

```

[explain the step by step of this code](#)



Let's go through the code step by step to understand how the "Top View of a Binary Tree" is implemented in C++.

Step-by-Step Breakdown:

1. Node Structure Definition

```

cpp
Copy code

struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) {
        data = val;
        left = right = NULL;
    }
};

```

- Purpose: This defines the structure of a binary tree node. Each node contains:

- An integer `data` that stores the node's value.
- Two pointers, `left` and `right`, pointing to the left and right children of the node, respectively.
- The constructor initializes the node with a value `val` and sets both child pointers to `NULL` (no children yet).

2. The `topView` Function

```

cpp
Copy code

void topView(Node* root) {
    if (root == NULL)
        return;
}

```

- **Purpose:** The `topView` function prints the top view of the binary tree.
 - It first checks if the tree is empty (`root == NULL`). If true, it simply returns without doing anything.

3. Map to Store Top View Nodes

cpp Copy code

```
map<int, int> topNodeMap;
```

- **Purpose:** A `map` is used to store the first node encountered at each **horizontal distance (HD)**.
 - **Key:** The horizontal distance (HD) of the node.
 - **Value:** The data value of the node.
 - A `map` in C++ is ordered by default, which ensures that when you print the keys (HDs), they will be in ascending order.

4. Queue for Level Order Traversal

cpp Copy code

```
queue<pair<Node*, int>> q;
q.push(make_pair(root, 0));
```

- **Purpose:** A `queue` is used to perform a level-order traversal (BFS - Breadth-First Search) of the binary tree.
 - The queue stores pairs, where each pair consists of:
 - A node pointer (`Node*`) for the current node.
 - An integer representing the horizontal distance (`HD`) of that node from the root.
 - We initialize the queue by pushing the root node with a horizontal distance of 0.

5. Processing Nodes Level by Level

cpp Copy code

```
while (!q.empty()) {
    pair<Node*, int> curr = q.front();
    q.pop();
```

- **Purpose:** The loop continues as long as there are nodes in the queue.
 - `q.front()` gets the first node in the `queue` (current node and its HD).
 - `q.pop()` removes the node from the queue after processing it.

6. Extract Current Node and Horizontal Distance

cpp Copy code

```
Node* node = curr.first;
int hd = curr.second;
```

- **Purpose:** Extract the node pointer (`node`) and its corresponding horizontal distance (`hd`) from the current pair (`curr`).

7. Add the First Node at Each Horizontal Distance

cpp Copy code

```
if (topNodeMap.find(hd) == topNodeMap.end()) {
    topNodeMap[hd] = node->data;
}
```

- **Purpose:** The `topNodeMap` is updated only if the current horizontal distance (`hd`) is not already in the map.
 - This ensures that only the **first node** seen at a particular horizontal distance (from top view) is added to the map. Any subsequent nodes at the same HD (which are hidden from the top view) are ignored.

8. Traverse Left and Right Subtrees

cpp Copy code

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

- **Purpose:** The left and right children of the current node are added to the queue for further processing.

- The horizontal distance for the left child is $hd - 1$ (since it's left of the current node).
- The horizontal distance for the right child is $hd + 1$ (since it's right of the current node).
- These child nodes will be processed in subsequent iterations of the loop.

9. Printing the Top View

cpp

Copy code

```
for (auto it : topNodeMap) {
    cout << it.second << " ";
}
```

- **Purpose:** After all nodes have been processed, the `topNodeMap` will contain the first node encountered at each horizontal distance.

- The `for` loop iterates over the map and prints the values (`it.second`) in increasing order of horizontal distances (`it.first`), thereby displaying the top view of the tree.

10. Main Function and Example Tree

cpp

Copy code

```
int main() {
    // 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);
    root->right->left = new Node(6);
    root->right->right = new Node(7);

    cout << "Top view of the binary tree is: ";
    topView(root);

    return 0;
}
```

- **Purpose:** The `main()` function creates an example binary tree and calls the `topView()` function to print the top view.

- In this tree, the root is 1, with left child 2 and right child 3, respectively, and further children as shown in the structure.

Example Binary Tree:

markdown

Copy code

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

How the Top View is Calculated:

1. The root node 1 is at HD = 0.
2. The left child 2 is at HD = -1, and the right child 3 is at HD = +1.
3. The left child of 2, node 4, is at HD = -2, and the right child of 3, node 7, is at HD = +2.
4. Nodes 5 and 6 are at HD = 0 and HD = 0, but since node 1 was the first node encountered at HD = 0, they are not part of the top view.

Top View Output:

The top view nodes are 4 2 1 3 7, which are printed in order of increasing horizontal distance (from left to right).

⬇️ ⏪ ⏴ ⏵ ⏴ ⏵

explain me the iteration steps



Let's walk through the iteration steps of the top view algorithm using the example binary tree:

markdown

Copy code

1



We will process the tree level by level (using \downarrow and we'll keep track of the horizontal distance (HD) of each node from the root.

Step 1: Initialization

1. **Map** (`topNodeMap`): This stores the first node encountered at each horizontal distance (HD).
 2. **Queue** (`q`): This helps in doing level-order traversal (BFS). The queue holds pairs of nodes and their horizontal distances.
 3. **Initial state:**
 - The root node `1` is inserted into the queue with HD = 0.
 - Queue: `[(1, 0)]` (where `1` is the node, and `0` is the HD).
 - `topNodeMap` is empty.
-

Step 2: First Iteration

- **Current queue:** `[(1, 0)]` \downarrow
 - Dequeue node `1` (HD = 0).
 - Since HD = 0 is not in the map, add node `1` to `topNodeMap` with HD = 0.
 - `topNodeMap`: `{0: 1}`
 - Enqueue its left child `2` (HD = -1) and right child `3` (HD = +1).
 - Queue: `[(2, -1), (3, +1)]`
-

Step 3: Second Iteration

- **Current queue:** `[(2, -1), (3, +1)]`
 - Dequeue node `2` (HD = -1).
 - Since HD = -1 is not in the map, add node `2` to `topNodeMap` with HD = -1.
 - `topNodeMap`: `{-1: 2, 0: 1}`
 - Enqueue its left child `4` (HD = -2) and right child `5` (HD = 0).
 - Queue: `[(3, +1), (4, -2), (5, 0)]` \downarrow
-

Step 4: Third Iteration

- **Current queue:** `[(3, +1), (4, -2), (5, 0)]`
 - Dequeue node `3` (HD = +1).
 - Since HD = +1 is not in the map, add node `3` to `topNodeMap` with HD = +1.
 - `topNodeMap`: `{-1: 2, 0: 1, +1: 3}`
 - Enqueue its left child `6` (HD = 0) and right child `7` (HD = +2).
 - Queue: `[(4, -2), (5, 0), (6, 0), (7, +2)]`
-

Step 5: Fourth Iteration

- **Current queue:** `[(4, -2), (5, 0), (6, 0), (7, +2)]` \downarrow
 - Dequeue node `4` (HD = -2).
 - Since HD = -2 is not in the map, add node `4` to `topNodeMap` with HD = -2.
 - `topNodeMap`: `{-2: 4, -1: 2, 0: 1, +1: 3}`
 - Node `4` has no children, so we do not enqueue anything.
 - Queue: `[(5, 0), (6, 0), (7, +2)]`
-

Step 6: Fifth Iteration

—Stephanie

- **Current queue:** `[(5, 0), (6, 0), (7, +2)]`
 - Dequeue node `5` (HD = 0).
 - Since HD = 0 is already in the map (we have already added node `1` at HD = 0), we ignore node `5` for the top view.
 - `topNodeMap : {-2: 4, -1: 2, 0: 1, +1: 3}`
 - Node `5` has no children, so we do not enqueue anything.

Step 7: Sixth Iteration

- **Current queue:** `[(6, 0), (7, +2)]`
 - Dequeue node `6` (HD = 0).
 - Since HD = 0 is already in the map (we already have node `1` at HD = 0), we ignore node `6` for the top view.
 - `topNodeMap: {-2: 4, -1: 2, 0: 1, +1: 3}`
 - Node `6` has no children, so we do not enqueue anything.
 - Queue: `[(7, +2)]`

Step 8: Seventh Iteration

- **Current queue:** `[(7, +2)]`
 - Dequeue node `7` (HD = `+2`).
 - Since HD = `+2` is not in the map, add node `7` to `topNodeMap` with HD = `+2`.
 - `topNodeMap` : `{-2: 4, -1: 2, 0: 1, +1: 3, +2: 7}`
 - Node `7` has no children, so we do not enqueue anything.
 - Queue: `[]` (now empty)

Final Step: Printing the Top View

After processing all nodes, the `topNodeMap` contains:

cpp

 Copy code

{-2: 4, -1: 2, 0: 1, +1: 3, +2: 7}

- The keys (horizontal distances) are sorted automatically in the `map`.
 - The final top view is obtained by printing the values in the `map` from the smallest HD to the largest HD:
 - **Top View:** 4 2 1 3 7

Conclusion:

- The algorithm goes level by level, and for each node, it checks if its horizontal distance is already in the map.
 - If the horizontal distance is not in the map, the node is part of the top view.
 - This process ensures that we only include the first node encountered at each horizontal distance, which gives us the top view of the binary tree.