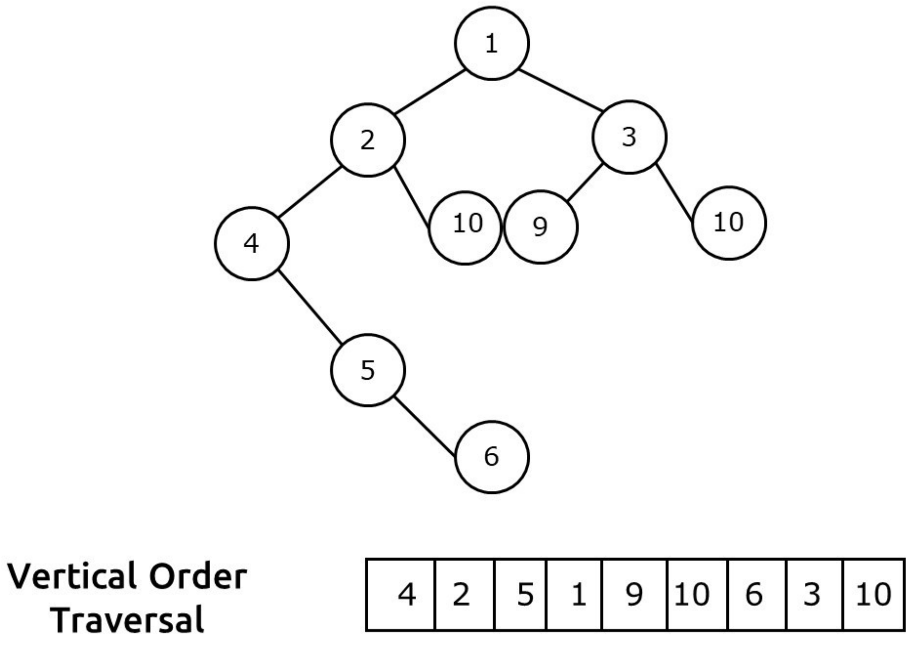
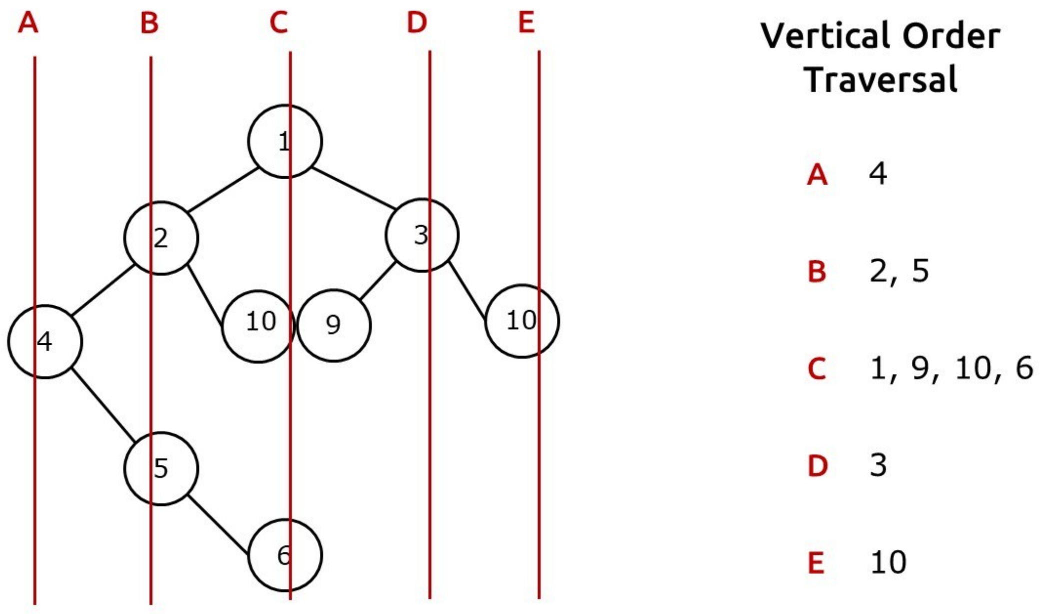
Vertical Order Traversal of Binary Tree
Problem Statement: Vertical Order Traversal Of A Binary Tree. Write a program for Vertical Order Traversal order of a Binary Tree.

Example:



Problem Description:
Vertical Order Traversal, as the name suggests is a traversal in which we divide the binary tree in verticals from left to right, and in every vertical, we print the nodes from top to bottom.



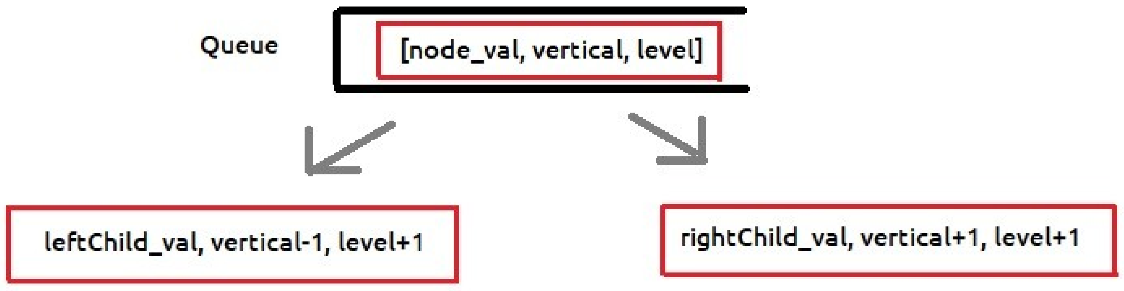
Note:

- There can be duplicate values among the nodes (here 10 is repeated twice).
- If two or more nodes are overlapping at the same position(here 10 and 9), then they will be printed in ascending order.

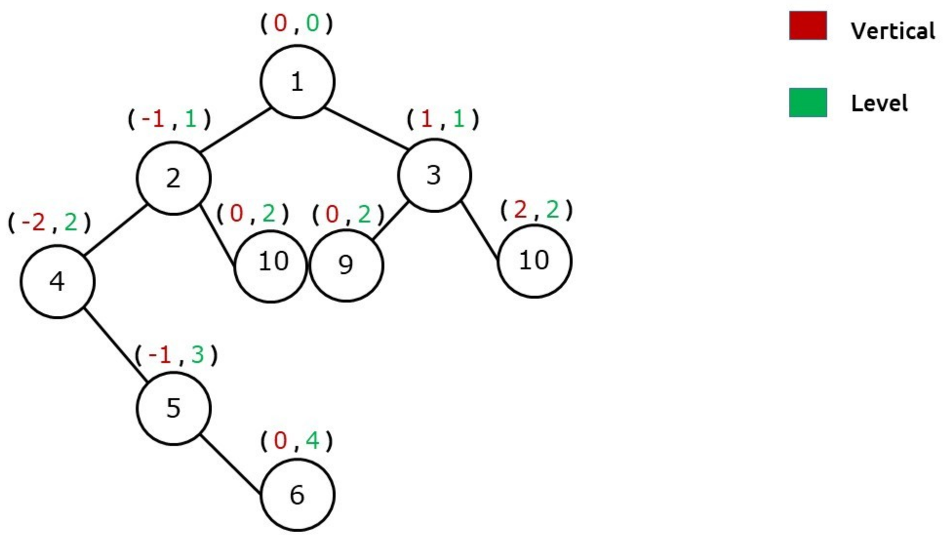
Solution :

Disclaimer*: Don't jump directly to the solution, try it out yourself first.*
Intuition: First of all, we need to assign a vertical and a level to every node. Once we assign them, we need to figure out a suitable data structure to store them. This data structure should give us the nodes with left-side vertical first and in every vertical, top-level node should appear first.
Approach:
We will perform a tree traversal and assign a vertical and level to every node. Based on this vertical and node, we store the node in our special data structure. For easy understanding, we break it into these steps:
Step-1: Assigning vertical and level to every node

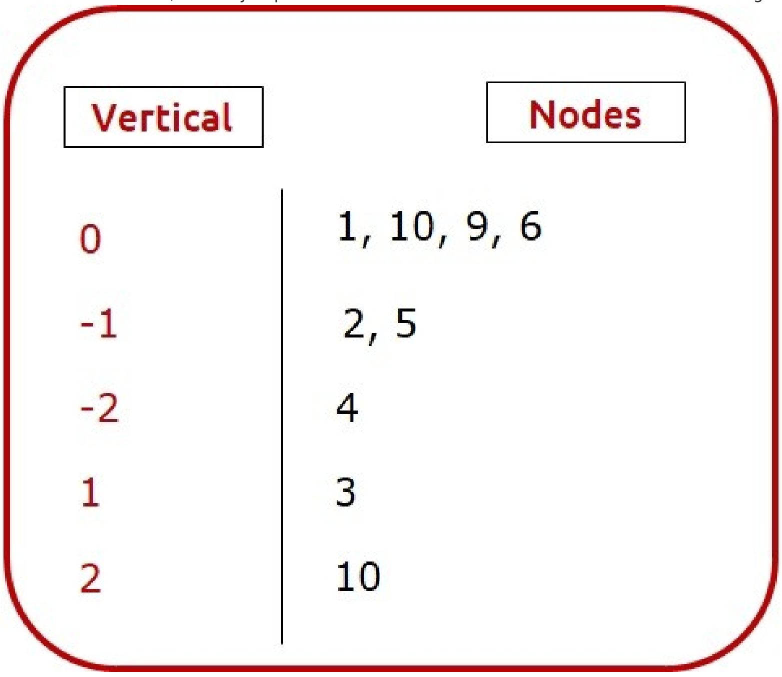
We can perform any tree traversal for this step. Here we are taking the example of level order traversal. In the initial step when we push the node to our queue, we will push two more variables to it, one for the vertical and one for the level (both initialized as zero). Now whenever we push the left child of the node, we decrease vertical by 1 and increase level by 1, whereas whenever we push the right child of the node, we increase both vertical and level by one.


Queue
[node_val, vertical, level]
leftChild_val, vertical-1, level+1
rightChild_val, vertical+1, level+1

Verticals and levels will be marked as follows:

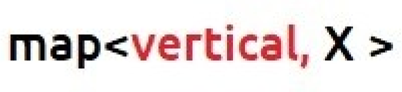


Step-2: Storing Verticals and levels to our data structure
When we are assigning the verticals and levels, it is very important to store them in a suitable data structure. After assigning we have:



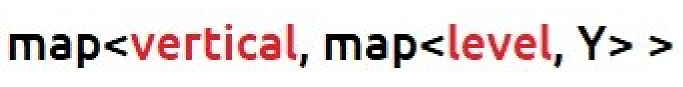
Vertical	Nodes
0	1, 10, 9, 6
-1	2, 5
-2	4
1	3
2	10

In the vertical order traversal, we need to print the nodes of the left vertical first, therefore in our example, nodes of -2 vertical will be the first to be printed. Therefore, we need a data structure that can store nodes according to their vertical value and give us the nodes of least values first. Hence we will use a **map** as it satisfies both criterias.



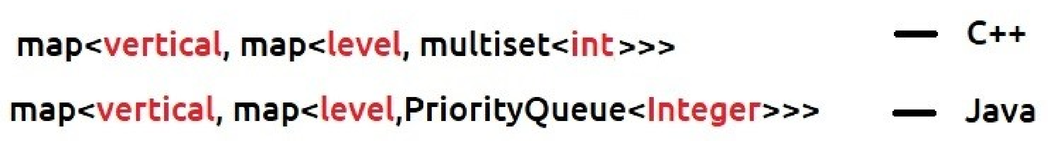
map<vertical, X >

Now, we will explore X. In a single vertical we want to get the nodes by their levels. In our example, vertical 0 has nodes of three different levels, we will print level 0 first (node 1), then level 2 (node 9 and 10) and at last level 4 (node 6). Therefore as in the case of our verticals, we will again use **map** to store nodes level-wise inside the vertical. So, our data structure will become:



map<vertical, map<level, Y> >

Now, we will explore Y. There can be a case that two or more nodes overlap at the same vertical and level, as with case of node 9 and node 10 at vertical 0 and level 2. In such a case we will print nodes with lesser value first. Therefore for every level, we need a data-structure which can store node values in a sorted order. Moreover, as duplicate values are allowed in our tree, our data structure needs to handle this well. This is acheived by using **multiset in C++**. Multiset is similar to a set which keeps elements in sorted order but also allows duplicates. **In Java**, we can use **priority queues** as it gives us the minimum value at the top.
Therefore, our final data structure will be:



map<vertical, map<level, multiset<int>>> — C++

map<vertical, map<level,PriorityQueue<Integer>>> — Java

Note: This is not the only choice for the data structure. You are free to choose any other data structure as well but the requirements remain the same. Instead of a multiset/priority queue, simple lists can also be used but then we will need to sort them in our last step.
Step 3: Printing vertical order traversal from our data structure
In the last step, we iterate over our verticals by using the data structure of step 2. In every iteration, we take a list(to store all nodes of that vertical) and again iterate over the levels. We then push the nodes of the level (stored in the multiset/priority queue) and push it to our vertical list. Once we iterate over all verticals we get our vertical order traversal.
Dry Run: In case you want to watch the dry run for this approach, please watch the video attached below.
Code:

- C++ Code
- Java Code

```
#include <bits/stdc++.h>

using namespace std;

struct node {
  int data;
  struct node * left, * right;
};

vector < vector < int >> findVertical(node * root) {
  map < int, map < int, multiset < int >>> nodes;
  queue < pair < node * , pair < int, int >>> todo;
  todo.push({
    root,
    {
      0,
      0
    }
  }); //initial vertical and level
  while (!todo.empty()) {
    auto p = todo.front();
    todo.pop();
    node * temp = p.first;

    //x -> vertical , y->level
    int x = p.second.first, y = p.second.second;
    nodes[x][y].insert(temp -> data); //inserting to multiset

    if (temp -> left) {
      todo.push({
        temp -> left,
        {
          x - 1,
          y + 1
        }
      });
    }
    if (temp -> right) {
      todo.push({
        temp -> right,
        {
          x + 1,
          y + 1
        }
      });
    }
  }
  vector < vector < int >> ans;
  for (auto p: nodes) {
    vector < int > col;
    for (auto q: p.second) {
      col.insert(col.end(), q.second.begin(), q.second.end());
    }
    ans.push_back(col);
  }
  return ans;
}

struct node * newNode(int data) {
  struct node * node = (struct node * ) malloc(sizeof(struct node));
  node -> data = data;
  node -> left = NULL;
  node -> right = NULL;

  return (node);
}

int main() {

  struct node * root = newNode(1);
  root -> left = newNode(2);
  root -> left -> left = newNode(4);
  root -> left -> right = newNode(10);
  root -> left -> left -> right = newNode(5);
  root -> left -> left -> right -> right = newNode(6);
  root -> right = newNode(3);
  root -> right -> left = newNode(9);
  root -> right -> right = newNode(10);

  vector < vector < int > > verticalTraversal;
  verticalTraversal = findVertical(root);

  cout << "The Vertical Traversal is : " << endl;
  for (auto vertical: verticalTraversal) {
    for (auto nodeVal: vertical) {
         cout << nodeVal << " ";
      }
      cout << endl;
   }
   return 0;
}
```

Output:

```
The Vertical Traversal is :

4

2 5

1 9 10 6

3

10
```

Time Complexity: O(N*logN*logN*logN)

Space Complexity: O(N)