**Solution**

**Overview**

We are given the root of a binary tree.

Our task is to return the smallest level x such that the sum of all the values of nodes at level x is maximal.

**Approach 1: Breadth First Search**

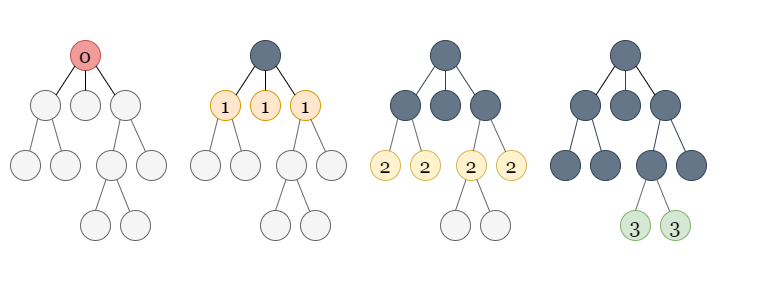
**Intuition**

The task is to compute the sum of all node values at each level to get the highest level with the maximum sum.

We can simply use a standard breadth-first search traversal because we need to analyze nodes by level.

BFS is an algorithm for traversing or searching a graph. It traverses in a level-wise manner, i.e., all the nodes at the present level (say l) are explored before moving on to the nodes at the next level (l + 1). BFS is implemented with a queue.

Here is an example with the steps:



If you are not familiar with BFS traversal, we suggest you read our [LeetCode Explore Card](https://leetcode.com/explore/featured/card/graph/620/breadth-first-search-in-graph/).

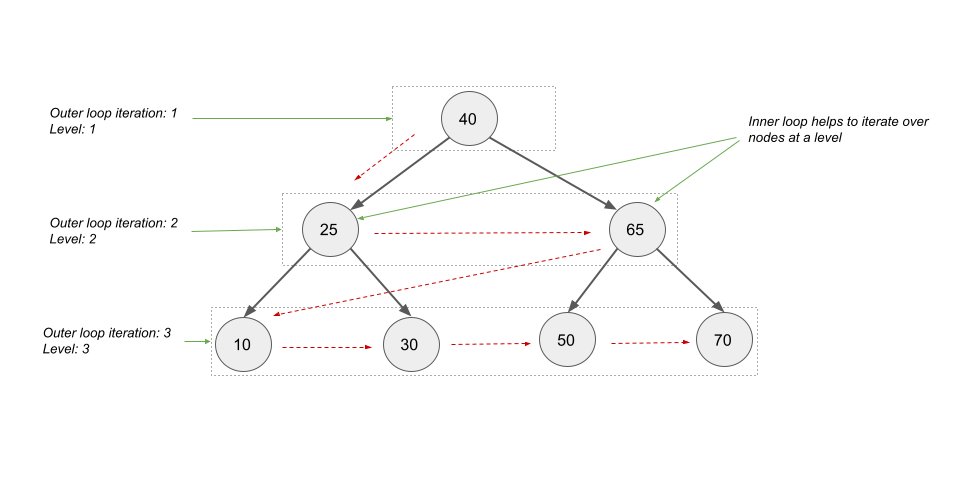
We initialize a queue of integers and an integer level = 0 to track the current level. In the queue, we push the root node.

We perform a level-wise traversal, incrementing level by 1 each time when we move to a new level. At each iteration, we remove all nodes at level, compute the sum of all node values at this level, and insert all their neighbouring nodes at level + 1.

Because we are popping all of the nodes at level and inserting all of the nodes at level + 1, the size of the queue will represent the number of nodes at the next level at the end of this iteration.

So we have two loops: the outer loop runs until the queue is empty, and the inner loop runs the number of times equal to the size of the queue to just cover the nodes at the current level. We will pop all the nodes at level, compute the sum of all the values, and insert all the nodes at level + 1 into the queue.

Here is a visual representation of how we will iterate using the loops:



To get the answer, we compare the sum of all node values at the current level to the maximum sum of values we've already seen. If the current sum of node values is greater than what we've seen before, we update our answer to level, and the current sum becomes our largest sum of values seen thus far. Since we are traversing the higher levels first, by only updating the answer when the level sum is **greater** than what we've seen before, we handle the tiebreakers automatically.

**Algorithm**

1. Create an integer variable maxSum to keep track of the maximum sum of node values at any level. We start with a large negative value.
2. Create another variable ans to store the answer to the problem.
3. Create another integer variable level to store the current level through which we are iterating. We initialize it with 0.
4. Initialize a queue q of TreeNode and push root into it.
5. Perform a BFS traversal until the queue is empty:
   * Increment level by 1 and initialize sumAtCurrentLevel = 0 to compute the sum of all values of nodes at this level.
   * Iterate through all the nodes at level using only the q.size() number of nodes. Within this inner loop, pop out all the nodes at the current level one by one, adding their values to sumAtCurrentLevel and pushing the left and right children (if they exist) into the queue.
   * Realize that after traversing all of the nodes at level, the queue only has nodes at level + 1.
   * After traversing through all the nodes at level, we check if sumAtCurrentLevel is greater than maxSum. If maxSum < sumAtCurrentLevel, update our answer variable to ans = level and set maxSum = sumAtCurrentLevel.
6. Return ans.

**Implementation**

class Solution {

public:

    int maxLevelSum(TreeNode\* root) {

        int maxSum = INT\_MIN;

        int ans = 0, level = 0;

        queue<TreeNode\*> q;

        q.push(root);

        while (!q.empty()) {

            level++;

            int sumAtCurrentLevel = 0;

            // Iterate over all the nodes in the current level.

            for (int sz = q.size(); sz > 0; --sz) {

                TreeNode\* node = q.front();

                q.pop();

                sumAtCurrentLevel += node->val;

                if (node->left != nullptr) {

                    q.push(node->left);

                }

                if (node->right != nullptr) {

                    q.push(node->right);

                }

            }

            if (maxSum < sumAtCurrentLevel) {

                maxSum = sumAtCurrentLevel;

                ans = level;

            }

        }

        return ans;

    }

};

**Complexity Analysis**

Here nn*n* is the number of nodes in the given binary tree.

* Time complexity: O(n)O(n)*O*(*n*).
  + Each queue operation in the BFS algorithm takes O(1)O(1)*O*(1) time, and a single node can only be pushed once, leading to O(n)O(n)*O*(*n*) operations for nn*n* nodes.
  + The computation of sum of all the values of nodes at a level also takes O(n)O(n)*O*(*n*) time as each node's value is used once.
* Space complexity: O(n)O(n)*O*(*n*).
  + As the BFS queue stores the nodes in level-wise manner, the maximum number of nodes in the BFS queue would equal to the most number of nodes at any level. So, the best case would be O(1)O(1)*O*(1) where all the levels have just one node.
  + The worst case would be a complete binary tree. In a complete binary tree, the last or second last level would have the most nodes (the last level can have multiple null nodes). Because we are iterating by level, the BFS queue will be most crowded when all of the nodes from the last level (or second last level) are in the queue. Assume we have a complete binary tree with height hh*h* and a fully filled last level having 2h2^h2*h* nodes. All the nodes at each level add up to 1+2+4+8+...+2h=n1 + 2 + 4 + 8 +... + 2^h = n1+2+4+8+...+2*h*=*n*. This implies that 2h+1−1=n2^{h + 1} - 1 = n2*h*+1−1=*n*, and thus 2h=(n+1)/22^h = (n + 1) / 22*h*=(*n*+1)/2. Because the last level hh*h* has 2h2^h2*h* nodes, the BFS queue will have (n+1)/2=O(n)(n + 1) / 2 = O(n)(*n*+1)/2=*O*(*n*) elements in the worst-case scenario.

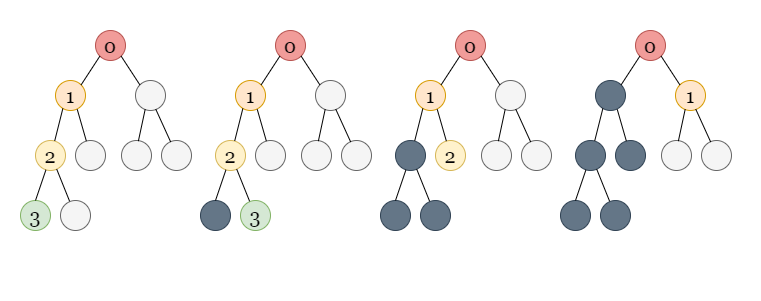
**Approach 2: Depth First Search**

**Intuition**

We can also use another traversal method, depth-first search (DFS).

In DFS, we use a recursive function to explore nodes as far as possible along each branch. Upon reaching the end of a branch, we backtrack to the next branch and continue exploring.

Once we encounter an unvisited node, we will take one of its neighbor nodes (if exists) as the next node on this branch. Recursively call the function to take the next node as the 'starting node' and solve the subproblem.



If you are new to Depth First Search, please see our [LeetCode Explore Card](https://leetcode.com/explore/featured/card/graph/619/depth-first-search-in-graph/3882/) for more information on it!

Because our task is to compute the sum of all the values of nodes at each level, we can perform a DFS traversal and pass the level of each node as an extra parameter.

We can initialize a list of integers sumOfNodesAtLevel, where sumOfNodesAtLevel[i] stores the sum of all the values of nodes at level i. Whenever we visit a node at a level, say l, we increment the index l in the list by the value of the current node. According to the problem definition, the levels should begin with 1, but to keep the list as 0-indexed, we will begin with level 0 (the root's level) and increment our answer by 1 at the end.

The question that may arise is how long this list should be.

We know that in a DFS traversal, we either move down the tree (until we can) to a node at the next level or we backtrack to a node at a lower level. As we descend the tree, if we come across a level l we haven't seen before, we add the node's value to sumOfNodesAtLevel, which places the entry at index l itself. This is due to the fact that all levels from 0 to l - 1 must have already been seen and have corresponding values in sumOfNodesAtLevel.

So, if the size of sumOfNodesAtLevel equals l, it means we've seen nodes from levels 0 to l - 1 but not any nodes at level l yet. At level l, this is the first node we see.

If the level l is smaller than the size of sumOfNodesAtLevel, it means we've seen some nodes at this level before, and we simply increment sumOfNodesAtLevel[l] by the value of the current node.

**Algorithm**

1. Create a list of integers sumOfNodesAtLevel to store the sum of all the values of nodes at a level. The value sumOfNodesAtLevel[i] stores the sum of all the values of nodes at level i (0-indexed). We would start our levels from 0 to keep the array 0-indexed and finally increment our answer by 1 to align with the problem definition of the level (levels begin with 1 as stated in the problem).
2. Perform the DFS traversal over the given binary tree. We call dfs(root, 0, sumOfNodesAtLevel) where dfs is a recursive method that takes three parameters: TreeNode node from which the traversal begins, the level of node, and sumOfNodesAtLevel. We perform the following in this method:
   * If node is null, return.
   * If the size of sumOfNodesAtLevel equals level, we haven't encountered any nodes at this level. Hence, we insert node.val in sumOfNodesAtLevel. Otherwise, if we've seen this level before, we simply perform sumOfNodesAtLevel[level] += node.val to add node.val to the corresponding level.
   * Recursively perform DFS from node.left.
   * Recursively perform DFS from node.right.
3. Create a variable maxSum to keep track of the maximum sum of node values at any level. We start with a large negative value.
4. Create another variable ans to store the answer to the problem.
5. Iterate over the sum of nodes of all the levels, i.e., iterate over sumOfNodesAtLevel and perform the following:
   * If maxSum < sumOfNodesAtLevel[i], we set maxSum = sumOfNodesAtLevel[i] and update ans to the level i + 1 (+1 is added to align with the definition of level).
6. Return ans.

**Implementation**

class Solution {

public:

    void dfs(TreeNode\* node, int level, vector<int>& sumOfNodesAtLevel) {

        if (node == nullptr) {

            return;

        }

        if (sumOfNodesAtLevel.size() == level) {

            sumOfNodesAtLevel.push\_back(node->val);

        } else {

            sumOfNodesAtLevel[level] += node->val;

        }

        dfs(node->left, level + 1, sumOfNodesAtLevel);

        dfs(node->right, level + 1, sumOfNodesAtLevel);

    }

    int maxLevelSum(TreeNode\* root) {

        vector<int> sumOfNodesAtLevel;

        dfs(root, 0, sumOfNodesAtLevel);

        int maxSum = INT\_MIN;

        int ans = 0;

        for (int i = 0; i < sumOfNodesAtLevel.size(); i++) {

            if (maxSum < sumOfNodesAtLevel[i]) {

                maxSum = sumOfNodesAtLevel[i];

                ans = i + 1;

            }

        }

        return ans;

    }

};

**Complexity Analysis**

Here nn*n* is the number of nodes in the given binary tree.

* Time complexity: O(n)O(n)*O*(*n*).
  + We traverse once over each node of the tree using DFS traversal which takes O(n)O(n)*O*(*n*) time. We also take O(1)O(1)*O*(1) time to add a node's value into sumOfNodesAtLevel for each node, which takes O(n)O(n)*O*(*n*) time for nn*n* nodes.
  + The size of sumOfNodesAtLevel is equal to the height of tree. We iterate over all the values in sumOfNodesAtLevel to get the level with maximum sum of node values. In the worst-case scenario, when the tree is a straight line, the height would be O(n)O(n)*O*(*n*), requiring O(n)O(n)*O*(*n*) time to iterate over sumOfNodesAtLevel.
* Space complexity: O(n)O(n)*O*(*n*).
  + The DFS traversal is recursive and would take some space to store the stack calls. The maximum number of active stack calls at a time would be the tree's height, which in the worst case would be O(n)O(n)*O*(*n*) when the tree is a straight line.
  + The sumOfNodesAtLevel would also take linear space in the worst case.