CS302: Data Structures using C++

Fall 2020

Final Exam Preparation - Additional Exercises: **Count Complete Tree Nodes**

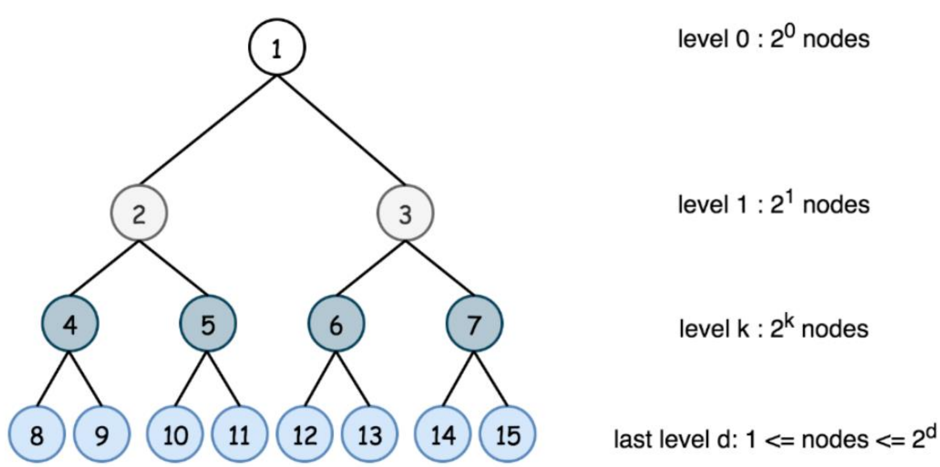
Kostas Alexis

Problem Description: Given a complete binary tree, count the number of nodes. In a complete binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between 1 and 2h nodes inclusive at the last level h.

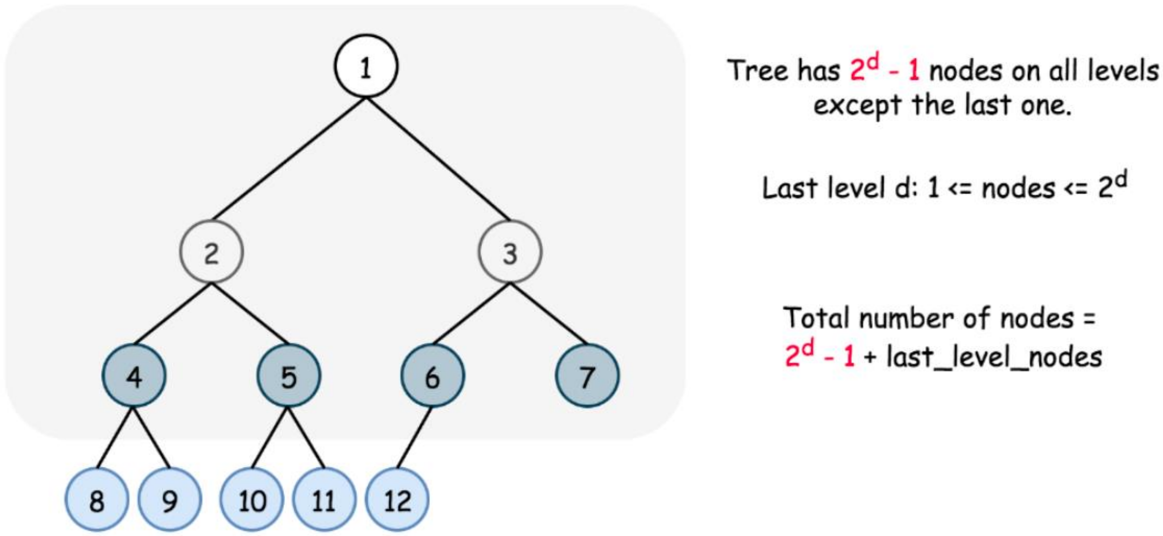
Example:

```
Input:

    1

   / \

  2   3

 / \  /

4  5 6

Output: 6
```

Solution: As we know, in a complete binary tree every level, except possibly the last is completely filled, and all nodes in the last level are as far left as possible. This in turn means that a complete tree has 2^k nodes in the k-th level if the k-th level is not the last one. The last level may not be filled completely, and hence in the last level the number of nodes could vary from 1 to 2^d, where d is the tree depth.



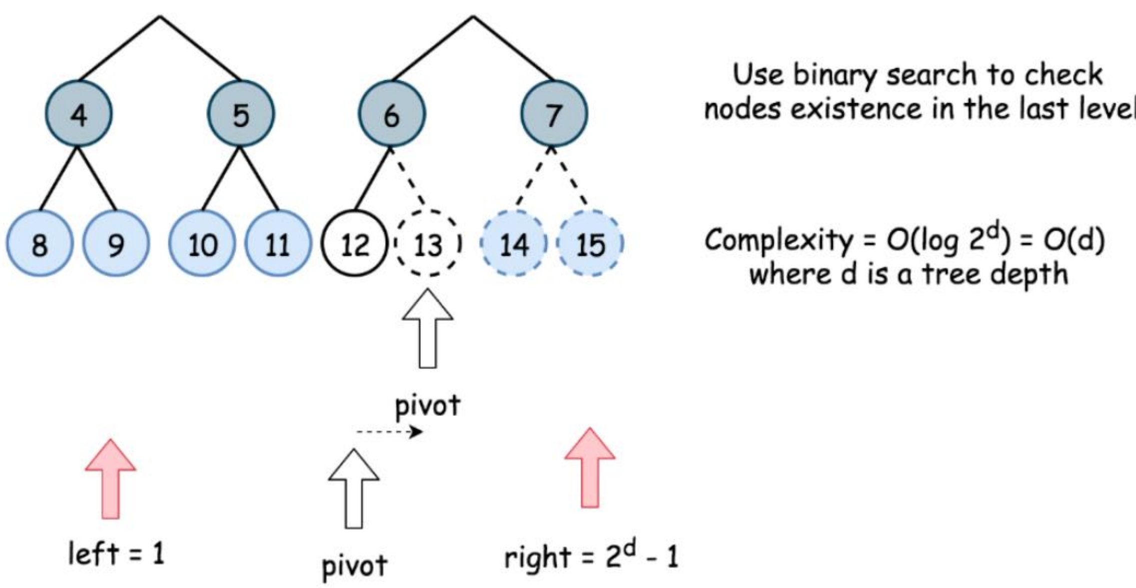
One may compute the number of nodes in all levels but the last one $\sum_{k=0}^{k=d-1} 2^k = 2^d - 1$. That reduces the problem to the simple check of how many nodes the tree has in the last level.



There are now two questions:

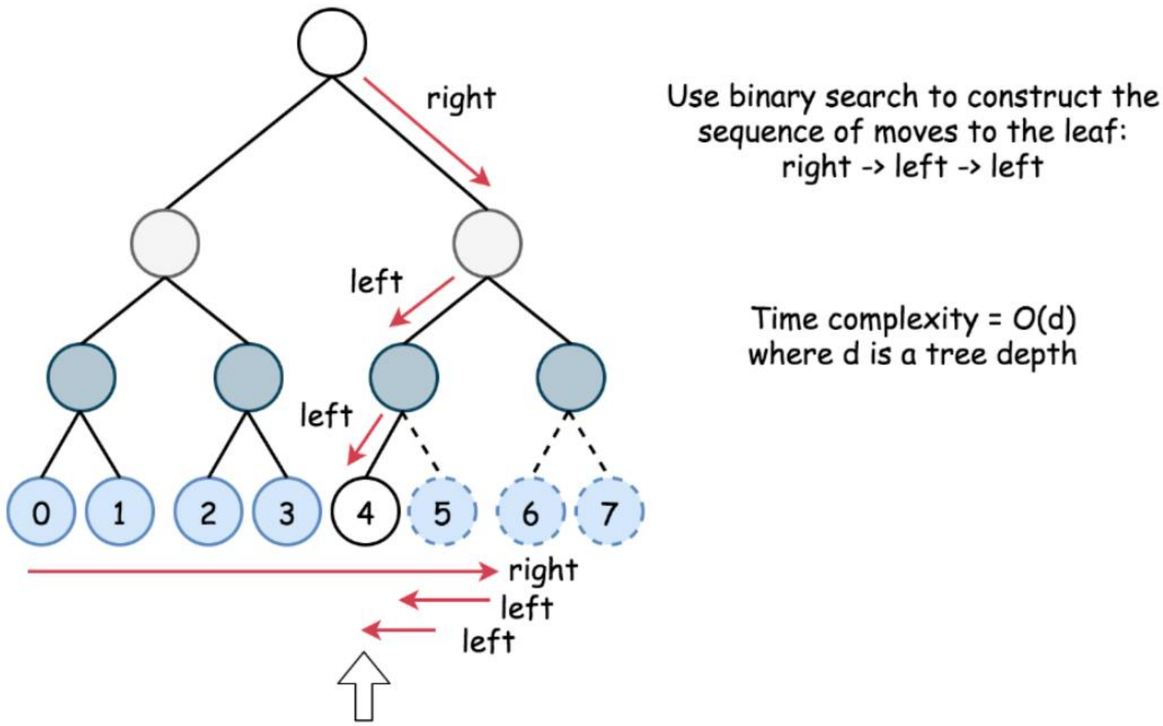
1. How many nodes in the last level have to be checked?
2. What is the best time performance for such a check?

Let's start from the first question. It is a complete tree, and hence all nodes in the last level are as far left as possible. This means that instead of checking the existence of all 2^d possible leafs, one could use binary search and check $\log(2^d) = 2$ leafs only.



Let's move to the second question, and enumerate potential nodes in the last level from 0 to $2^d - 1$. Now, the question is how to check if the node number `idx` exists. Let us use binary search again to reconstruct the sequence of moves from `root` to the `idx` node.

Then `idx` is the first half of nodes 4,5,6,7 and hence the second move is to the left. The `idx` is in the first half of nodes 4,5 and hence the next move is to the left. The time complexity for one check is $O(d)$.



Operations for (1) and (2) together result in $O(d)$ checks, with each check at a price of $O(d)$. That means that the overall time complexity would be $O(d^2)$.

The solution algorithm is as follows:

- Return 0 if the tree is empty.
- Compute the tree depth d.
- Return 1 if `d==0`.
- The number of nodes in all levels but the last is $\sum_{k=0}^{k=d-1} 2^k = 2^d - 1$. The number of nodes in the last level could vary from 1 to 2^d. Enumerate potential nodes from 0 to $2^d - 1$ and perform the binary search by the node index to check how many nodes are in the last level. Use the function `exists(idx, d, root)` to check if the node with index `idx` exists.
- Use binary search to implement `exists(idx, d, root)`.
- Return $2^d - 1$ + NUMBER_OF_NODES_IN_LAST_LEVEL

Solution File:

```
#include <math.h>
#include <iostream>
#include <queue>
#include <vector>
```

```
using namespace std;

struct TreeNode {
 int val;
 TreeNode(int v) : val(v), left(nullptr), right(nullptr) {}
 TreeNode *left;
 TreeNode *right;
};

void treeTraversal(TreeNode *root) {
 if (root != nullptr) {
   cout << root->val << ",";
   treeTraversal(root->left);
   treeTraversal(root->right);
 }
}

bool existLeaf(TreeNode *root, int d, int mid) {
 int i = 0;
 int lo = pow(2, d - 1);
 int hi = pow(2, d) - 1;
 while (i < d - 1) {
   // go left or right
   double side = ((double)mid - lo) / (hi - lo);
   if (side < 0.5) {
     root = root->left;
     hi = lo + (hi - lo) / 2;
   } else {
     root = root->right;
     lo = lo + (hi - lo) / 2 + 1;
   }
   ++i;
 }
 return (root != nullptr);
}

int countNodes(TreeNode *root) {
 if (root == nullptr) return 0;
 // depth
 TreeNode *node = root;
 int d = 0;
 while (node != nullptr) {
   ++d;
   node = node->left;
 }
 cout << "Depth: " << d << endl;
```

```
 // Binary search
 int lo = pow(2, d - 1);
 int hi = pow(2, d) - 1;
 while (lo <= hi) {
   int mid = lo + (hi - lo) / 2;
   if (existLeaf(root, d, mid))
     lo = mid + 1;
   else
     hi = mid - 1;
 }
 return (pow(2, d - 1) - 1) + (lo - pow(2, d - 1));
}

void buildCompleteTree(vector<int> &data, TreeNode* &root) {
 queue<TreeNode *> q;
 q.push(root);
 for (int i = 0; i < data.size(); ++i) {
   TreeNode *node = q.front();
   q.pop();
   node->val = data[i];
   if (i + q.size() + 1 < data.size()) {
     node->left = new TreeNode(0);
     q.push(node->left);
   }
   if (i + q.size() + 1 < data.size()) {
     node->right = new TreeNode(0);
     q.push(node->right);
   }
 }
 treeTraversal(root);
}

int main() {
 vector<int> data = {1, 2, 3, 4};
 TreeNode *root = new TreeNode(0);
 buildCompleteTree(data, root);

 int res = countNodes(root);
 cout << "Total node: " << res << endl;
 return 0;
}
```

Complexity Analysis:

- Time complexity: $O(d^2) = O(log^2 n)$, where d is the tree depth.
- Space complexity: $O(1)$