104. Maximum Depth of Binary Tree

Oct. 2, 2018 | 99.4K views

Average Rating: 4.53 (30 votes)

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Note: A leaf is a node with no children.

Example:

Given binary tree `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its depth = 3.

Solution

Tree definition

First of all, here is the definition of the `TreeNode` which we would use.

C++ | Java | Python — Copy

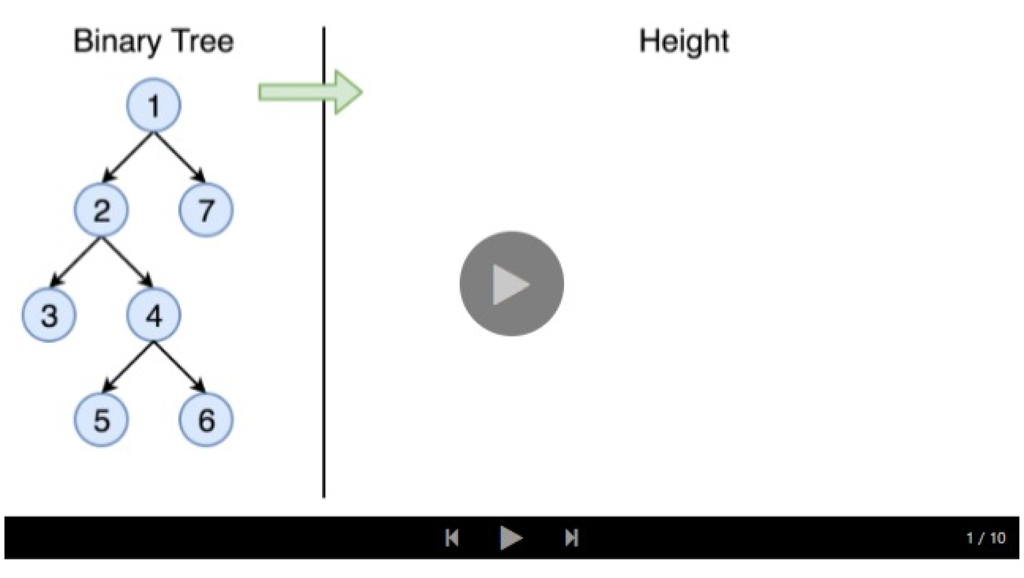
```python
class TreeNode(object):
    """ Definition of a binary tree node."""
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None
```

Approach 1: Recursion

Intuition By definition, the maximum depth of a binary tree is the maximum number of steps to reach a leaf node from the root node.

> From the definition, an intuitive idea would be to traverse the tree and record the maximum depth during the traversal.

Algorithm



One could traverse the tree either by Depth-First Search (DFS) strategy or Breadth-First Search (BFS) strategy. For this problem, either way would do. Here we demonstrate a solution that is implemented with the **DFS** strategy and **recursion**.

C++ | Java | Python — Copy

```python
class Solution:
    def maxDepth(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        if root is None:
            return 0
        else:
            left_height = self.maxDepth(root.left)
            right_height = self.maxDepth(root.right)
            return max(left_height, right_height) + 1
```

Complexity analysis

- Time complexity : we visit each node exactly once, thus the time complexity is $\mathcal{O}(N)$, where N is the number of nodes.
- Space complexity : in the worst case, the tree is completely unbalanced, *e.g.* each node has only left child node, the recursion call would occur N times (the height of the tree), therefore the storage to keep the call stack would be $\mathcal{O}(N)$. But in the best case (the tree is completely balanced), the height of the tree would be $\log(N)$. Therefore, the space complexity in this case would be $\mathcal{O}(\log(N))$.

Approach 2: Tail Recursion + BFS

One might have noticed that the above recursion solution is probably not the most optimal one in terms of the space complexity, and in the extreme case the overhead of call stack might even lead to *stack overflow*.

To address the issue, one can tweak the solution a bit to make it **tail recursion**, which is a specific form of recursion where the recursive call is the last action in the function.

The benefit of having tail recursion, is that for certain programming languages (*e.g.* `C++`) the compiler could optimize the memory allocation of call stack by reusing the same space for every recursive call, rather than creating the space for each one. As a result, one could obtain the constant space complexity $\mathcal{O}(1)$ for the overhead of the recursive calls.

Here is a sample solution. Note that the optimization of tail recursion is not supported by Python or Java.

C++ — Copy

```cpp
class Solution {

  private:
    // The queue that contains the next nodes to visit,
    //   along with the level/depth that each node is located.
    queue<pair<TreeNode*, int>> next_items;
    int max_depth = 0;

    /**
     * A tail recursion function to calculate the max depth
     *   of the binary tree.
     */
    int next_maxDepth() {

      if (next_items.size() == 0) {
        return max_depth;
      }

      auto next_item = next_items.front();
      next_items.pop();

      auto next_node = next_item.first;
      auto next_level = next_item.second + 1;

      max_depth = max(max_depth, next_level);

      // Add the nodes to visit in the following recursive calls.
```

Complexity analysis

- Time complexity : $\mathcal{O}(N)$, still we visit each node once and only once.
- Space complexity : $\mathcal{O}(2^{(\log_2 N - 1)}) = \mathcal{O}(N/2) = \mathcal{O}(N)$, *i.e.* the maximum number of nodes at the same level (the number of leaf nodes in a full binary tree), since we traverse the tree in the **BFS** manner.

As one can see, this probably is not the best example to apply the *tail recursion* technique. Because though we did gain the constant space complexity for the recursive calls, we pay the price of $\mathcal{O}(N)$ complexity to maintain the state information for recursive calls. This defeats the purpose of applying tail recursion.

However, we would like to stress on the point that tail recursion is a useful form of recursion that could eliminate the space overhead incurred by the recursive function calls.

Note: a function cannot be tail recursion if there are multiple occurrences of recursive calls in the function, even if the last action is the recursive call. Because the system has to maintain the function call stack for the sub-function calls that occur within the same function.

Approach 3: Iteration

Intuition

We could also convert the above recursion into iteration, with the help of the *stack* data structure. Similar with the behaviors of the function call stack, the stack data structure follows the pattern of FILO (First-In-Last-Out), *i.e.* the last element that is added to a stack would come out first.

With the help of the *stack* data structure, one could mimic the behaviors of function call stack that is involved in the recursion, to convert a recursive function to a function with iteration.

Algorithm

> The idea is to keep the next nodes to visit in a stack. Due to the FILO behavior of stack, one would get the order of visit same as the one in recursion.

We start from a stack which contains the root node and the corresponding depth which is `1`. Then we proceed to the iterations: pop the current node out of the stack and push the child nodes. The depth is updated at each step.

C++ | Java | Python — Copy

```python
class Solution:
    def maxDepth(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        stack = []
        if root is not None:
            stack.append((1, root))

        depth = 0
        while stack != []:
            current_depth, root = stack.pop()
            if root is not None:
                depth = max(depth, current_depth)
                stack.append((current_depth + 1, root.left))
                stack.append((current_depth + 1, root.right))

        return depth
```

Complexity analysis

- Time complexity : $\mathcal{O}(N)$.
- Space complexity : in the worst case, the tree is completely unbalanced, *e.g.* each node has only left child node, the recursion call would occur N times (the height of the tree), therefore the storage to keep the call stack would be $\mathcal{O}(N)$. But in the average case (the tree is balanced), the height of the tree would be $\log(N)$. Therefore, the space complexity in this case would be $\mathcal{O}(\log(N))$.

SanD91 ★ 536 January 31, 2019 7:47 AM

An easier approach for iterative would be to use BFS instead of any need to store depth in the stack.

```
class Solution {
    public int maxDepth(TreeNode root) {
        if (root == null) return 0;
```

Read More

33 Share Reply

SHOW 4 REPLIES

elazar ★ 18 October 3, 2018 12:36 PM

I think the second JAVA solution is BFS and not a DFS, for DFS you need to call to remoeLast

18 Share Reply

SHOW 6 REPLIES

lenchen1112 ★ 973 January 5, 2020 9:07 AM

Python 3 one line

```
class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        return max(map(self.maxDepth, [root.left, root.right])) + 1 if root else
```

Read More

7 Share Reply

jehon777 ★ 8 January 4, 2020 1:47 PM

Why is approach 3 space complexity O(N) in the worst case? if you have only left child nodes your stack would never be bigger than 1 right? (e.g. add 1, pop 1, for each iteration) please correct me if I am wrong

5 Share Reply

SHOW 5 REPLIES

vdhyani96 ★ 18 January 5, 2020 11:17 PM

Another version of approach 1 easier to understand (Python3):

```
class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        if not root:
```

Read More

3 Share Reply

ptn32 ★ 29 July 30, 2019 9:23 PM

I thought if there is just the root node the depth would be 0? Why is it 1?

3 Share Reply

SHOW 1 REPLY

aghalarov ★ 21 January 21, 2019 6:27 AM

Hi,
This code is not working in computer.How we can implement this tree which is can debug in computer not leetcode?

1 Share Reply

SHOW 1 REPLY

starlord ★ 50 December 17, 2018 3:24 AM

when the second solution run takes 5 ms and seems very slow?

1 Share Reply

SHOW 1 REPLY

pet_watchdog ★ 5 March 18, 2019 6:34 AM

Notice, C++ does not support tail call optimization (as far as I know its not in the standard) – some compilers (ex. gcc). Also, you cannot always rely on tail call optimization: gcc will not be able to perform the optimization if there are not trivially destructible objects on the stack.

0 Share Reply

SHOW 1 REPLY

sadat ★ 0 June 17, 2020 10:49 AM

Can anyone explain, how a node (and/or tree) is represented as an array here? where is that implemented?
[3,9,20,null,null,15,7]

0 Share Reply