Tree Structures

This lesson is a general discussion on reasoning about space and time complexity of tree structures

In this lesson we won't attempt to describe various tree data-structures and their associated operations. Rather, we'll do a general discussion on how to reason and think about space and time complexity when tree structures are involved.

Tree Traversals

When traversing a tree, does the size of the tree affect how much time it takes for you to go through every node of the tree? Yes, it does. A bigger tree will take more time to go through than a smaller tree. The following recursive algorithms for tree traversals all take O(n) time since we are required to visit each node.

- pre-order traversal
- in-order traversal
- post-order traversal
- depth first search. Note the above three algorithms are types of depth first traversals
- breadth first traversal/search

For the above algorithms, the more interesting question pertains to space complexity for the above algorithms. Since one can write these algorithms recursively, it's not immediately clear what the space complexity is. A novice may think it is constant when it is not. Look at the code below:

```
class TreeNode {
    TreeNode left;
```

```
        TreeNode right;
        int val;
    }

    void recursiveDepthFirstTraversal(TreeNode root) {
        if (root == null)
            return;

        recursiveDepthFirstTraversal(root.left);
        System.out.println(root.val);
        recursiveDepthFirstTraversal(root.right);
    }
```

A naive look at the `recursiveDepthFirstTraversal` method may mislead you to think that there's no extra space being used. However, behind the scenes, the recursive calls are stored on a stack by the underlying language platform. When we implement the non-recursive version then we are better able to appreciate the space requirements. Take a look at the non-recursive version of in-order traversal below:

```
    void depthFirstTraversal(TreeNode root) {

            Stack<TreeNode> stack = new Stack<>();
            TreeNode trav = root;

            while (trav != null || !stack.isEmpty()) {

                while (trav != null) {
                    stack.push(trav);
                    trav = trav.left;
                }

                trav = stack.pop();
                System.out.println(trav.val);

                trav = trav.right;
        }
    }
```

Now it should be apparent that the algorithm relies on the use of a stack. It is either implicit in case of recursion or explicit if an iterative version is written out.

Height of Trees

In the traversal algorithm, a loose upper bound on the space complexity would be O(n) because the stack can be filled at most with every node of the tree. However, you might realize that the stack - at any time during the execution of the algorithm - could be ***filled at most with the maximum number of nodes that appear in the longest path from the root to a leaf node***. This observation allows us to put a tighter bound on the space requirements, as we know that the longest path from the root to the leaf would be the ***height of the tree***. Therefore, in the traversal algorithm above the space complexity will be:

> $$O(log_2n)$$

Height of a binary tree is given by $\log_2 n$ where n is the total number of nodes in the tree. You can think of it as given *n*, how many times do you need to divide it by 2 to get to 1? The answer is $\log_2 n$.

Trees becoming Linked Lists

One should be wary of scenarios when trees turn into linked lists. In case of a binary search tree, if it consists of all the nodes with the same values or inserts are made in an ascending order of values, then the binary search tree would turn into a linked list. Suddenly the *O(lgn)* promise for search is broken and search is now linear or *O(n) operation.*.

Graph Traversals

If asked to do graph traversal, the complexity would be O(n) where n is the total number of nodes. Visiting each node once is the least number of nodes you can visit to cover the entire graph. For a graph with cycles, you'll need to remember the visited nodes so as not to enter into an infinite cycle. The space complexity would change now. Can you think of the space complexity in case of a graph with cycles?

In the worst case, the graph is just one circle, and unless you visit the node again, you won't know you were in a cycle. In that case, you'll end up remembering all the nodes of the graph, and the space complexity will become O(n).

Binary Trees

A binary tree is a tree which has at most two children. Don't confuse a binary tree with a binary search tree. Searching in a binary tree will take O(n) time, but in a binary search tree it'll take O(lgn). However, an unbalanced binary search tree can still result in linear search.

Let's reason about the space complexity of representing a binary tree consisting of only integers. We can store the binary tree in an array or create a class like below to represent each node of the tree

```
class TreeNode {
    int val;
    TreeNode leftChild;
    TreeNode rightChild;
}
```

There’s no right answer here, and the right approach would depend on the scenario you are involved in. If it is known that the binary tree will be full, i.e. all nodes except the leaves will have two children, then it would make sense to use an array. It’ll save us the cost of the two pointer-nodes in the other approach. But if the binary tree is expected to grow then it may be better to use the second approach. Also, if the binary tree is very tall but is sparsely filled, then the array approach would waste a lot of space.

In the array approach we'll allocate space for the full binary tree, even if the number of nodes is less. Otherwise, in the second approach, the space complexity will be proportional to the size of the number of nodes. The extra pointer nodes add constant space per node so the big O space requirement will still be O(n).

Another aspect to pay attention to is that using an array to store a binary tree makes access to any node of the tree a constant time operation. However, if you choose to represent it using the TreeNode approach then accessing the leaf nodes would take time proportional to the height of the tree.