Reflection Essay - Assignment 4

Prasann Viswanathan

Question 1: Ghastly Checkpoints

i) I have some prior experience working with graphs (part of tech team recruitment) and a tree is essentially a graph (V, E) which is connected and acyclic. Since the question involved a tree with an arbitrary number of edges, unlike a BST, it was easier to implement it as an adjacency list graph. Then was the matter of finding the shortest path to a leaf.
Algorithm:

- Suppose we have a tree, and we start off at node 1.
- Mark all leaves, which are identified by the fact that they have exactly one connected edge
- Mark each Node as visited if you enter it.
- From a node, recursively search for the closest leaf to that Node, provided the connected node is unvisited.
- Return the minimum distance of the leaf from the node.

Correctness:

- The distance of an actual leaf node is 0 which handles the boundary condition for recursion.
- As we start from Node 1, We essentially do a pre order traversal only down the tree, as no Nodes are revisited once they are visited even once.
- In turn, we visit each Node exactly once, and keep track of the closest leaf to it.
- As we are checking for all Nodes, we surely check all leaves, and thus have all the path lengths. Thus our algorithm always returns the shortest path.

ii) If there are N nodes in our tree, the algorithm runs in O(N)
(95% my work, 5% reference)
Ref: https://www.geeksforgeeks.org/graph-and-its-representations/

Question 2: Quad Nodes

1) My method involves doing a post-order traversal of the trees, which would imply we traverse a branch till we reach the bottom, after which we operate on the nodes as the stack is released.
Each node can store additional information fields so we store the number of left descendants and the number of right descendants for each node. Leaves have 0 left and right descendants. Also, a boolean flag is_quad_desc (True if descendants are quad, false otherwise) and is_quad is needed. .
Then we do a post order traversal simultaneously updating left descendants and right descendants recursively and if both left child and right child of a node have their is_quad as true, we set is_quad of node as true as well.

Pseudo Code:

Node{ Key, Left_desc, Right_desc, bool is_quad_desc, is_quad, Node* left, right}

Pre_order(Node){

```
        if(NULL)
                Left desc, right desc = 0
                Is_quad = true;
                Is_quad_desc = true;
                Return
        else
                Pre_order(left)
                Pre_order(right)
                Left desc = 1+left->left_desc+left->right_desc
                right desc = …
                is _quad = (abs(left_desc-right_desc)<=4)
                Is_quad_desc = left->is_quad_desc && right->is_quad
}

Insert(...){ … }

delete(...){ … }
```

Although this is rough, the idea should be clear.

2) The method is correct as we are using recursion to build member variables of each node AND we have accounted for the base case.
By induction, the base case is done and the subsequent cases depend solely on the correctness of the previous case. Some worked out pen and paper examples did correctly so the update equation is also correct.

3) Since each node is visited exactly once, if we have N nodes, the algorithm is O(N) - Linear as desired

For the given example -
The following will be the updates in sequence:

4 -> quad, quad_desc (abbreviated as Q and QD now, -Q and -QD for false)
10 -> Q, QD
9 -> Q, QD
8 -> Q, QD
6 -> Q, QD
7 -> Q, QD
5 -> Q, QD
2 -> -Q, QD
3 -> Q, QD
1 -> -Q, QD

An in order traversal of the same would give nodes 1 and 2 as the required nodes.

Question 3: Mutant Virus
We need to implement a dictionary via a BST such that the keys are the index and of the character in the word (BAD, B is indexed 1, A - 2, D - 3) and the value is the character. Now each correct mutant is a substring of the in order traversal of the BST, however this will result in some repetitions.

To count the number of mutants we do an inorder traversal of the tree to get the characters in order, then a Dynamic Programming approach can be used to return number of distinct subsequences in O(n)

Ref: https://www.geeksforgeeks.org/count-distinct-subsequences/

In summary - We use the BST as a dictionary with insert having the 'quaternion' comparison function in place of regular ordering via < and >.
Thus the BST is filled with entries in order of their occurrence in the word

Finally, the word is obtained via an in-order traversal and we find subsequences that are unique using a dynamic programming approach.

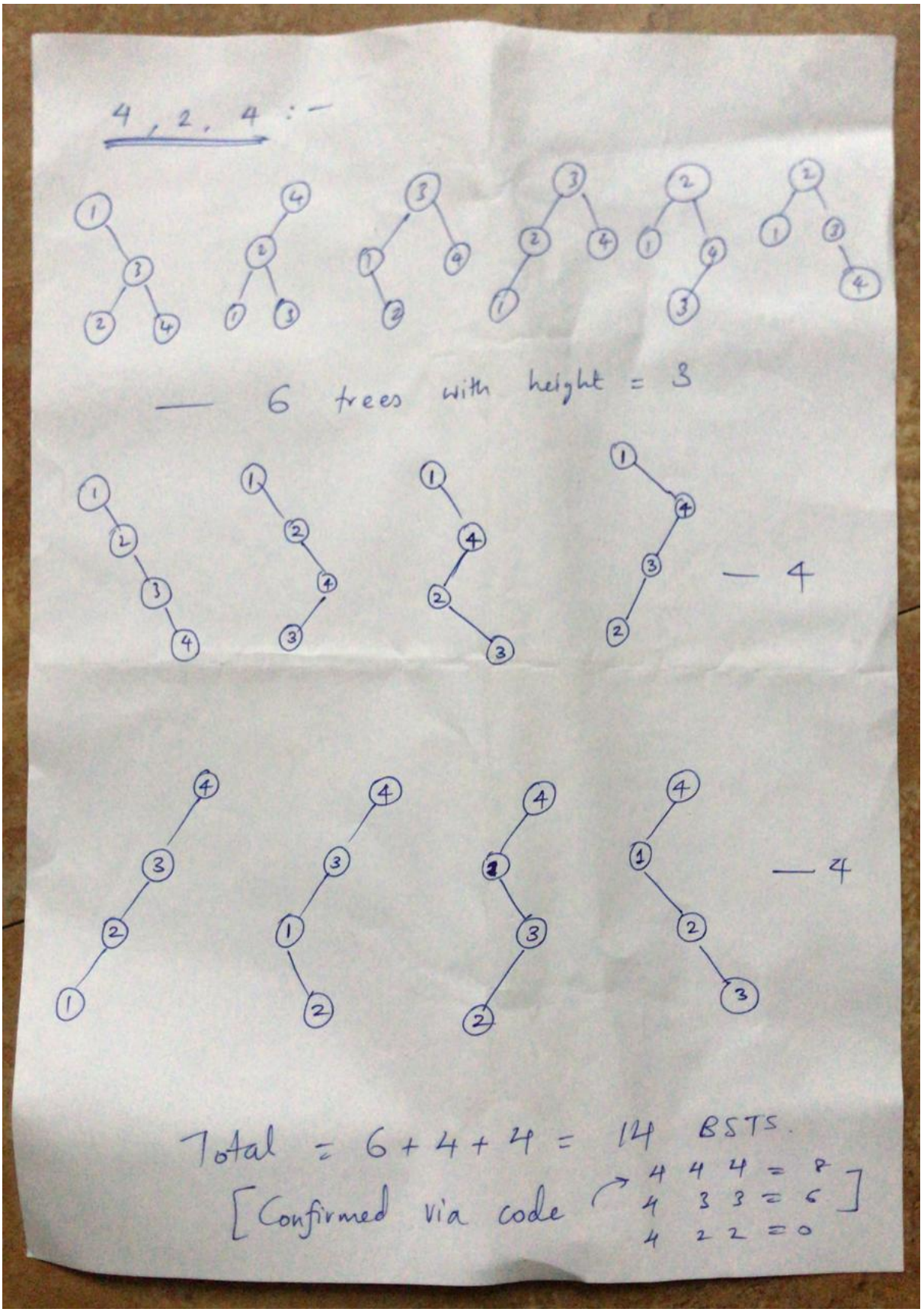
The dp update is 2*dp of n-1 th entry - repeated subsequences.

Question 4: In Summary
Each node has 4 components only: Key (x), Value f(x), left (Node to left), right (Node to right). These are standard for a BST and uphold the O(1) space condition.
Because the question expected changes in the BST structure as per class, I initially implemented AVL trees - which would self adjust height using rotations, thus keeping the tree balanced and essentially maintaining O(logn) complexity for each operation.
However, this gave segmentation faults that I wasn't able to overcome, forcing me to try passing some cases using the basic algorithm as described in class.
Insert remained the same as described in class. Care had to be taken to accommodate the problem statement by updating the value in case a key was repeated.
Delete also remains the same as described. I used the least successor approach as described in class.
The functions to implement on our own were Add and Print f(x).
Print simply involved traversal of the tree till we found a matching key and reporting it's value.
Add was done in-order because x1 - x2 is essentially a range within which we can update efficiently if we check keys in ascending order from x1 to x2 (BSTs are printed in ascending order via in-order traversal - from class).
This basic algorithm from class, with little to no changes passed all test cases and insert, delete, print all remain O(height) while Add is O(height+(No. Of keys within [x1, x2]).

Note: The references make up almost the entire code, but I didn't copy paste and understood and implemented. Also I spent 2 hours debugging for this Question. For this reason I call this 50% my work and 50% reference.
Ref: https://www.geeksforgeeks.org/avl-tree-set-1-insertion
https://www.geeksforgeeks.org/avl-tree-set-2-deletion/

Question 5: Keep Counting

1)



2) We start by calling no_of_trees(4, 4) which will return the number of trees with max height of 4 and 4 nodes.
These in turn call no_of_trees(3, 0), no_of_trees(3, 1), …, no_of_trees(3, 3).
And these call backwards as well till we reach the base cases which help build our table of values.

```
1   1   0   0   0
1   1   2   -1  -1
1   1   2   5   -1
1   -1  -1  -1  14
1   -1  -1  -1  -1
```

These come from the formula in reference, which is also intuitive and comes recursively, like catalan numbers.
Which in turn shows that the number of trees with upto height of 4 is 14 and since the entry corresponding to height of 1 and 4 nodes is 0, we know that all 14 trees have heights between 2 and 4.

3) The logic of the algorithm is that we fix a root value and find the number of subtrees on the left side and right side and the product of these is the number of trees with that root. This is done by a recursion with memoization approach.

4) Time complexity is O(n*r) which is very less considering n, r<=50
Space complexity is also O(n*r) to store the values which is also as good as fixed space because we can declare a 2D array of dimensions 50*50 which would work for all cases such that n, r <= 50

https://math.stackexchange.com/questions/1388564/number-of-binary-search-trees-on-n-nodes-of-height-up-to-h
(Most credit goes to the formula, but the memoization trick is mine)