CPSC 331 HW3_Written

Ali Akbari

TOTAL POINTS

13.5 / 15

QUESTION 1

Properties of binary trees 5 pts

1.1 n nodes has n+1 null links 1.5 / 2

- **0 pts** Correct

✓ **- 0.5 pts** **Click here to replace this description.**

- **1 pts** Click here to replace this description.
- **2 pts** Click here to replace this description.

You should start with a tree that has k+1 nodes and then remove a node from it not the otherway around. since you are trying to prove that any tree with k+1 nodes has k+2 null link

1.2 number of leaves 3 / 3

✓ **- 0 pts** **Correct**

- **0.5 pts** Base Case, minor
- **1 pts** Induction step, adding a node to hypothesis instead of removing a node from the claim.
- **1 pts** Base case Major
- **0.5 pts** Inductive step minor, not covering all possibilities of change in full node or node count
- **0.5 pts** Using induction incorrectly
- **0.5 pts** base case might be a chian
- **3 pts** Incorrect, or no asnwer
- **0.5 pts** Base case not clear, more of why that is the case is needed
- **1 pts** major issue
- **0.5 pts** Small attempt, formal proof is required
- **0.5 pts** unclear inductive step, more details are required

QUESTION 2

Is BST algorithm and analysis 5 pts

2.1 Algorithm 2.5 / 3

- **0 pts** Correct
- **1 pts** Root null check missing
- **1 pts** Root children value check missing
- **1 pts** Missing recursive call
- **1 pts** Input should only take a tree, could be fixed using an interface or helper function,
- **0.5 pts** Minor issue

✓ **- 0.5 pts** **Global variables(use interface instead)**

- **1 pts** Missing check for null children
- **3 pts** Incorrect/ no answer
- **2 pts** wrong format, code or pseudo code is needed
- **1 pts** Major issue
- **0 pts** Click here to replace this description.
- **1 pts** local variable always reset at the begining
- **0 pts** Click here to replace this description.

2.2 Linearity 1.5 / 2

- **0 pts** Correct
- **1 pts** Wrong recurrence, in case of a perfect tree T(n)= 2T(n/2)+1
- **1 pts** Wrong end result
- **0.5 pts** Minor issue, recurrence missing +1 (it shows there exist a constant time operation even though it wont affect the final result)

✓ **- 0.5 pts** **Minor issue, you should state if you have assumed the tree is perfect**

- **1 pts** Unclear steps
- **2 pts** Incorrect/ no answer/ no justification
- **0.5 pts** good attempt, more formal justification is needed
- **1 pts** Recurrence does not match the algorithm
- **0.5 pts** unfinished math
- **0.5 pts** Master theorem details missing
- **1 pts** Good attempt but a more formal justification is needed

- **0.5 pts** Missing steps

QUESTION 3

BST insertion 5 pts

3.1 Worst case 2 / 2

✓ **- 0 pts Correct**

- **0.5 pts** Clearly Identifying the worst case(inserting a sorted sequence, an example is not enough)
- **0.5 pts** Minor issue, connecting BST to SLL is unclear
- **0.5 pts** wrong result/not mentioning the result ,n^2 is the worst case
- **0.5 pts** No mathematical modeling of worst case (summation)
- **2 pts** No asnwer / Incorrect
- **0.5 pts** wrong summation / recurrence

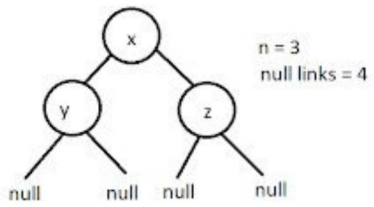
3.2 Best case 3 / 3

✓ **- 0 pts Correct**

- **1 pts** Not mentioning that the tree should remain balanced
- **0.5 pts** Clearly mentioning that the BST should remain "balanced" or complete/perfect after all insertions
- **1 pts** Wrong end result/ or no result, (nlogn is the best case)
- **1 pts** No mathematical modeling (Summation)
- **0 pts** Best case can be describe more precisely, refer to the solutions
- **0.5 pts** Inaccuracy in calculations
- **0.5 pts** unclear calculations
- **0.5 pts** wrong mathematical modeling
- **0.5 pts** In a balanced tree all leaves are necessarily not on the same level
- + **0.5 pts** Good attempt
- **0.5 pts** Not mentioning the final result, nlogn
- **3 pts** No answer

CPSC 331 Assignment #3 Written
Ali Akbari
30010402

1. **Prove the following:**
    a. **A binary tree with n nodes has n + 1 null links**



It can easily be shown for a perfect binary tree, like from the above image this property holds, but to prove we must use induction as binary trees are not always perfect trees.

Base Case:
Consider a binary tree with one node, it has two null links as it has no left or right children.
$For\ n\ = 1\ there\ is\ n_{links} = 2.\ So\ the\ base\ case\ holds.$

Induction:
Consider a binary tree with n nodes that has n + 1 null links, and suppose we add/insert one more node. Adding a node also adds two more null links and removes one null link from the node it was added to. A node can have either two null links(leaf node) or one null link, so a node can only be added to either a leaf node or a node with only one child, but both have similar outcomes. Suppose the node is added to a leaf node, as stated before two null links have to be added and one null link is removed, this is the same for adding to a node with one null link.
So for $n\ +\ 1$ nodes, there are $n\ +\ 1\ -\ 1\ +\ 2$ null links.

Thus for $n+1$ nodes, there are $n\ +\ 2$ null links.

    b. **In a non-empty binary tree, the number of full nodes plus one is equal to the number of leaves. (A full node is a node that has two children.)**

Take n to be the total number of nodes in a binary tree. In an unspecified non-empty binary tree, there could be full nodes (denoted as T(two children)) as well as nodes with 1 child, (denoted as O (one child)). Full nodes have no null links while leaf nodes have two and the rest (O) have one null link.

1.1 n nodes has n+1 null links 1.5 / 2

- **0 pts** Correct

✓ - **0.5 pts** **Click here to replace this description.**

- **1 pts** Click here to replace this description.
- **2 pts** Click here to replace this description.

You should start with a tree that has k+1 nodes and then remove a node from it not the otherway around. since you are trying to prove that any tree with k+1 nodes has k+2 null link

Let L equal to the number of Leaves.

From the above information we get $Total\ \#\ Leave\ =\ L\ =n\ -\ T\ +\ O$

Since each node has two links leading out, the total number of links would be 2 times the number of nodes so $2n$.
To prove the proposition $T\ +\ 1\ =\ L$ from the question we have to use the relation of null links to the number of nodes from question one and the above information, the following equation could be built:

$Total\ links\ -\ links\ from\ full\ nodes\ and\ half\ nodes\ =\ null\ links$

$2n\ -\ (2T\ +\ O)\ =\ n\ +\ 1$

$2n\ -\ Total\ number\ of\ links$

$(2T\ +\ O)\ -Amount\ of\ non\ null\ links,\ 2\ *\ T\ for\ full\ nodes,\ 1\ *\ O\ for\ nodes\ with\ 1\ null\ link$

$n\ +\ 1\ -\ Total\ null\ links$

And by using the first equation and substitution,

$equation\ \#1\ :L\ =n\ -\ T\ -\ O$

$equaion\ \#2\ :\ 2n\ -\ (2T\ +\ (n\ -\ T\ -\ L))\ =\ n\ +\ 1$

$2n\ -\ 2T\ -\ n\ +\ T\ +\ L\ =\ n\ +\ 1$

$n\ -\ T\ +\ L\ =\ n\ +\ 1$

$-\ T\ +\ L\ =\ +\ 1$

$L\ =\ T\ +\ 1$

$T\ +\ 1\ =\ L$

Thus it holds that full nodes plus one is equal to the number of leaves in a nonempty binary tree.

2. **Design a linear-time algorithm to test whether a binary tree is a binary search tree. Provide pseudocode or Java code for your algorithm and justify that your algorithm is linear in the number of nodes.**

   One way to check if a binary tree is a binary search tree is by checking for the BST properties. All nodes and their left subtree contain children with smaller value/data and the right subtree contains children with larger value/data. Since it is not stated what type of data is contained in the tree, we must use compareTo function to check the data hierarchy between parent's data and children's data. This algorithm could be done recursively like traversing a binary tree. To see if the algorithm is linear we need to check its complexity, which is similar to the complexity of traversing a binary tree.

1.2 number of leaves 3 / 3

✓ **- 0 pts Correct**

- 0.5 pts Base Case, minor

- 1 pts Induction step, adding a node to hypothesis instead of removing a node from the claim.

- 1 pts Base case Major

- 0.5 pts Inductive step minor, not covering all possibilities of change in full node or node count

- 0.5 pts Using induction incorrectly

- 0.5 pts base case might be a chian

- 3 pts Incorrect, or no asnwer

- 0.5 pts Base case not clear, more of why that is the case is needed

- 1 pts major issue

+ **0.5 pts** Small attempt, formal proof is required

- 0.5 pts unclear inductive step, more details are required

Let L equal to the number of Leaves.

From the above information we get $Total\ \#\ Leave\ =\ L\ =n\ -\ T\ +\ O$

Since each node has two links leading out, the total number of links would be 2 times the number of nodes so $2n$.
To prove the proposition $T\ +\ 1\ =\ L$ from the question we have to use the relation of null links to the number of nodes from question one and the above information, the following equation could be built:

$Total\ links\ -\ links\ from\ full\ nodes\ and\ half\ nodes\ =\ null\ links$

$2n\ -\ (2T\ +\ O)\ =\ n\ +\ 1$

$2n\ -\ Total\ number\ of\ links$

$(2T\ +\ O)\ -Amount\ of\ non\ null\ links,\ 2\ *\ T\ for\ full\ nodes,\ 1\ *\ O\ for\ nodes\ with\ 1\ null\ link$

$n\ +\ 1\ -\ Total\ null\ links$

And by using the first equation and substitution,

$equation\ \#1\ :L\ =n\ -\ T\ -\ O$

$equaion\ \#2\ :\ 2n\ -\ (2T\ +\ (n\ -\ T\ -\ L))\ =\ n\ +\ 1$

$2n\ -\ 2T\ -\ n\ +\ T\ +\ L\ =\ n\ +\ 1$

$n\ -\ T\ +\ L\ =\ n\ +\ 1$

$-\ T\ +\ L\ =\ +\ 1$

$L\ =\ T\ +\ 1$

$T\ +\ 1\ =\ L$

Thus it holds that full nodes plus one is equal to the number of leaves in a nonempty binary tree.

2. **Design a linear-time algorithm to test whether a binary tree is a binary search tree. Provide pseudocode or Java code for your algorithm and justify that your algorithm is linear in the number of nodes.**

   One way to check if a binary tree is a binary search tree is by checking for the BST properties. All nodes and their left subtree contain children with smaller value/data and the right subtree contains children with larger value/data. Since it is not stated what type of data is contained in the tree, we must use compareTo function to check the data hierarchy between parent's data and children's data. This algorithm could be done recursively like traversing a binary tree. To see if the algorithm is linear we need to check its complexity, which is similar to the complexity of traversing a binary tree.

Pseudocode/ Algorithm layout from Geeksforgeeks:

```
Algorithm Inorder(tree)
   1. Traverse the left subtree, i.e., call Inorder(left-subtree)
   2. Visit the root.
   3. Traverse the right subtree, i.e., call Inorder(right-subtree)
```

Java code:

```
//Implemented algorithm from geeksforgeeks website for in order traversal
//Using codeblocks for format
//Main method
public class main{
    public static void main(String[] args) {
        //Initializing  binary tree nodes
        BinaryTreeNode aNode;
        BinaryTreeNode aNodePrevious = null;
        checkIfBST(aNode);
       }
}

public boolean checkIfBST (BSTNode aNode){
       return checker(aNode);
}
public boolean checker (BinaryTreeNode aNode) < T extends Comparable <? Super T>>{
      if (aNode != null){
             if (!checker(aNode.left)){
                 Return false;
             }

             if (aNodePrevious != null && node.el.compareTo(aNodePrevious.el) <= 0){
                 Return false;
             }
             aNodePrevious = aNode;
             return checker(aNode.right)
       }
       return true;
}
```

Let n be the number of nodes.

Let $T(n)$ be the number of node traversal and comparison recursive relation function.

Intuitively and from the code, we get the following:

$T(1) = 1$, traverses to only the root node.

2.1 Algorithm 2.5 / 3

- **0 pts** Correct
- **1 pts** Root null check missing
- **1 pts** Root children value check missing
- **1 pts** Missing recursive call
- **1 pts** Input should only take a tree, could be fixed using an interface or helper function,
- **0.5 pts** Minor issue

✓ **- 0.5 pts Global variables(use interface instead)**

- **1 pts** Missing check for null children
- **3 pts** Incorrect/ no answer
- **2 pts** wrong format, code or pseudo code is needed
- **1 pts** Major issue
- **0 pts** Click here to replace this description.
- **1 pts** local variable always reset at the begining
- **0 pts** Click here to replace this description.

Based on the inorder method and the above pseudocode, we first traverse the left subtree, we then visit the node. Then we traverse the right subtree. We are dividing the number of nodes by two in each recursive call (the left or right subtree). We do this two times once for the left side of the tree and once for the right side of the tree. So we are visiting basically all nodes. Also, there is a constant $O(1)$ time cost of the comparison in the third if statement in java code. We get the following recursive relation base on the above information,

$$T(n) = 2T(\tfrac{n}{2}) + 1$$

And we can solve this by iterative substitution.

$T(n) = 2T(\frac{n}{2}) + 1$ **Iteration #1**

$= 2(2T(\frac{\frac{n}{2}}{2}) + 1) + 1$

$= 4T(\frac{n}{4}) + 2$ **Iteration #2**

$= 4(2T(\frac{\frac{n}{2}}{4}) + 1) + 2$

$= 8T(\frac{n}{8}) + 3$ **Iteration #3**

.
.
.

So if this continues for k iterations we get the following:

$$= 2^k T(\tfrac{n}{2^k}) + \sum_{i=0}^{k} i$$

Since we know T(1), then we know that $(\frac{n}{2^k})$ decreases to at least 1 from the base case. So,

$(\frac{n}{2^k}) = 1$

$n = 2^k$, $k = log_2 n$

So,

$$T(n) = 2^{log_2 n} T(1) + \sum_{i=0}^{log_2 n} i$$

$$T(n) = n * (1) + \sum_{i=0}^{log_2 n} i$$

$$T(n) = n + log_2 n$$

With a limit test, we can see that n grows faster than $n + log_2 n$.

$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{n}{log_2 n}$, *top grows faster then below so*

$\lim_{n \to \infty} \frac{\infty}{log_2 \infty}$, $f(n) = \omega\, g(n)$

For a tight complexity, we get $T(n) = \Theta(n)$.

2.2 Linearity 1.5 / 2

- **0 pts** Correct
- **1 pts** Wrong recurrence, in case of a perfect tree T(n)= 2T(n/2)+1
- **1 pts** Wrong end result
- **0.5 pts** Minor issue, recurrence missing +1 (it shows there exist a constant time operation even though it wont affect the final result)

✓ **- 0.5 pts Minor issue, you should state if you have assumed the tree is perfect**

- **1 pts** Unclear steps
- **2 pts** Incorrect/ no answer/ no justification
- **+ 0.5 pts** good attempt, more formal justification is needed
- **1 pts** Recurrence does not match the algorithm
- **0.5 pts** unfinished math
- **0.5 pts** Master theorem details missing
- **+ 1 pts** Good attempt but a more formal justification is needed
- **0.5 pts** Missing steps

3. **Analyze the worst case and best case running times of inserting n elements into an initially empty binary search tree. For each case, provide details of your analysis starting from counting relevant operations to an accurate asymptotic characterization.**

   a. **Worst Case**

   The worst case of inserting n elements into an initially empty binary search tree is when the elements to be inserted are sorted, so it is either max to min or min to max elements that are inserted, so for inserting you are inserting in a general direction, right or left, which would resemble a list. i.e only adding to the right/left child of the previous node. Inserting and making an unbalanced binary tree.

   To insert into the list you have to traverse the list, initially for the first element it takes no traversal, but suppose you insert an element n, you would have to traverse i times into that list plus the traversal done for n-1, n-2.. and so on.
   The worst-case complexity for traversing the tree for a node can be shown as a summation. From the lectures, we know that inserting a node into a binary search tree has a time complexity of O(n), this is also shown in the summation below.

   $\sum_{i=0}^{n-1} = i$, 0, 1, 2, 3, 4... n-1

   I.e no traverses for the first node, one traversal for the second node, and so on.
   So we have n nodes and know that for a single insertion it will take O(n-1) order of time for traversal. $n(n-1) = n^2 - n$
   This yields a time complexity of $\Theta = n^2$.

   b. **Best Case**

   The cost of inserting n elements/nodes is dependent on the shape/height of the binary search tree. In the worst-case above, the shape of the tree is similar to a list. For the best case, it is when we insert n elements/nodes to make a balanced or close to perfect binary tree. The analysis can be done by using the internal path length.
   From the binary trees lecture slides,
   the internal path length is the sum of all path lengths of all the nodes.
   We get the following:
   For $T(1) = 0$
   $T(n) = 2T(\frac{n-1}{2}) + (n-1)$
   $2T(\frac{n-1}{2}) = Internal\ path\ length\ of\ a\ subtree$
   $(n-1) = path\ from\ all\ nodes\ to\ the\ root$

3.1 Worst case 2 / 2

✓ **- 0 pts** **Correct**

- 0.5 pts Clearly Identifying the worst case(inserting a sorted sequence, an example is not enough)

- 0.5 pts Minor issue, connecting BST to SLL is unclear

- 0.5 pts wrong result/not mentioning the result ,n^2 is the worst case

- 0.5 pts No mathematical modeling of worst case (summation)

- 2 pts No asnwer / Incorrect

- 0.5 pts wrong summation / recurrence

3. **Analyze the worst case and best case running times of inserting n elements into an initially empty binary search tree. For each case, provide details of your analysis starting from counting relevant operations to an accurate asymptotic characterization.**

   a. **Worst Case**

   The worst case of inserting n elements into an initially empty binary search tree is when the elements to be inserted are sorted, so it is either max to min or min to max elements that are inserted, so for inserting you are inserting in a general direction, right or left, which would resemble a list. i.e only adding to the right/left child of the previous node. Inserting and making an unbalanced binary tree.

   To insert into the list you have to traverse the list, initially for the first element it takes no traversal, but suppose you insert an element n, you would have to traverse i times into that list plus the traversal done for n-1, n-2.. and so on.
   The worst-case complexity for traversing the tree for a node can be shown as a summation. From the lectures, we know that inserting a node into a binary search tree has a time complexity of O(n), this is also shown in the summation below.

   $\sum_{i=0}^{n-1} = i$, 0, 1, 2, 3, 4... n-1

   I.e no traverses for the first node, one traversal for the second node, and so on.
   So we have n nodes and know that for a single insertion it will take O(n-1) order of time for traversal. $n(n-1) = n^2 - n$
   This yields a time complexity of $\Theta = n^2$.

   b. **Best Case**

   The cost of inserting n elements/nodes is dependent on the shape/height of the binary search tree. In the worst-case above, the shape of the tree is similar to a list. For the best case, it is when we insert n elements/nodes to make a balanced or close to perfect binary tree. The analysis can be done by using the internal path length.
   From the binary trees lecture slides,
   the internal path length is the sum of all path lengths of all the nodes.
   We get the following:
   For $T(1) = 0$
   $T(n) = 2T(\frac{n-1}{2}) + (n-1)$
   $2T(\frac{n-1}{2}) = Internal\ path\ length\ of\ a\ subtree$
   $(n-1) = path\ from\ all\ nodes\ to\ the\ root$

By using iterative substitution

$T(n) = 2T(\frac{n-1}{2}) + (n-1)$ **Iteration #1**

$T(\frac{n-1}{2}) = 2T(\frac{(\frac{n-1}{2})-1}{2}) + (\frac{n-1}{2} - 1)$

$T(n) = 2(2T(\frac{\frac{n-1}{2}-1}{2}) + (\frac{n-1}{2} - 1)) + (n-1)$

$T(n) = 4T(\frac{n-3}{4}) + (n-3) + (n-1)$ **Iteration #2**

$T(\frac{n-3}{4}) = 2T(\frac{\frac{n-3}{4}-1}{2}) + (\frac{n-3}{4} - 1)$

$T(n) = 4(2T(\frac{n-7}{8}) + \frac{(n-7)}{4}) n - 3 + n - 1$

$T(n) = 8T(\frac{n-7}{8}) + n - 7 + n - 3 + n - 1$ **Iteration #3**

.

.

.

So if this continues for k iterations we get the following:

$= 2^k T(\frac{n-2^k+1}{2^k}) + kn - \sum_{i=1}^{k} 2^i - 1$

We get this after simplification of sum using the sum of polynomial rule:

$= 2^k T(\frac{n+1-2^k}{2^k}) + k(n+1) - 2^{k+1} + 2$

The base case of T(1) = 0 is reached when $(\frac{n+1-2^k}{2^k}) = 1.$

$(n+1-2^k) = 2^k$

$n+1 = 4^k$

$k = log_2(n+1) - 1$

Subsituting this back into the simplified formula

$T(n) = (n+1)log_2(n+1) - 2n$

$T(n) = nlog_2(n+1) - 2n + log_2(n+)$

The biggest term is $nlog_2(n+1)$ which in order of $nlog_2 n$

So the best case we need $\Theta(nlog_2 n)$ traversals.

Reference:

https://www.geeksforgeeks.org/a-program-to-check-if-a-binary-tree-is-bst-or-not/

https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/

3.2 Best case 3 / 3

✓ **- 0 pts Correct**

- 1 pts Not mentioning that the tree should remain balanced

- 0.5 pts Clearly mentioning that the BST should remain "balanced" or complete/perfect after all insertions

- 1 pts Wrong end result/ or no result, (nlogn is the best case)

- 1 pts No mathematical modeling (Summation)

- 0 pts Best case can be describe more precisely, refer to the solutions

- 0.5 pts Inaccuracy in calculations

- 0.5 pts unclear calculations

- 0.5 pts wrong mathematical modeling

- 0.5 pts In a balanced tree all leaves are necessarily not on the same level

+ **0.5 pts** Good attempt

- 0.5 pts Not mentioning the final result, nlogn

- 3 pts No answer