CSC263 – Problem Set 2

Remember to write your **full name** and **student number** prominently on your submission. To avoid suspicions of plagiarism: at the beginning of your submission, **clearly state any resources (people, print, electronic) outside of your group, the course notes, and the course staff, that you consulted**.

Remember that you are required to submit your problem sets as both LaTeX `.tex` source files and `.pdf` files. There is a 10% penalty on the assignment for failing to submit both the `.tex` and `.pdf`.

Due Feb 11, 2019, 22:00; required files: ps2.pdf, ps2.tex, num_orders.py, num_trees.py

Answer each question completely, always justifying your claims and reasoning. Your solution will be graded not only on correctness, but also on clarity. Answers that are technically correct that are hard to understand will not receive full marks. Mark values for each question are contained in the [square brackets].

You may work in groups of up to THREE to complete these questions.

1. **[12]** Let $a_1, a_2, \ldots, a_n$ be a sequence of real numbers, for some $n \geq 1$. A `SUM-BOX` is an ADT that stores the sequence and supports the following operations (S is a given `SUM-BOX`):

   - `PARTIAL-SUM`(S, m): return $\sum_{i=1}^{m} a_i$, the partial sum from a_1 to a_m $(1 \leq m \leq n)$.
   - `CHANGE`(S, i, y): change the value of a_i to a real number y.

   Design a data structure that implements `SUM-BOX`, using an **augmented AVL tree**. The worst-case runtime of both `PARTIAL-SUM` and `CHANGE` must be in $\mathcal{O}(\log n)$. Describe your design by answering the following questions.

   (a) What is the key of each node in the AVL tree? What other attributes are stored in each node?

   (b) Write the pseudo-code of your `PARTIAL-SUM` operation, and explain why your code works correctly and why its worst-case running time is $\mathcal{O}(\log n)$. Let $S.root$ denote the root node of the AVL tree.

   (c) Describe in clear and concise English how your `CHANGE` operation works, and explain why it runs in $\mathcal{O}(\log n)$ time while maintaining the attributes stored in the nodes of the AVL tree.

   SOLUTIONS:

   1a) The key of each node in the AVL tree is i, where $a_1, a_2....a_n$ for 1 $<= i <= n$. The additional attribute stored in each node is the sum of its node value and the all the values in its left subtree.

   1b)

```
def PARTIAL_SUM(s,m):
  return psum(s.root,m)

def psum(node,m):
  if node.key==m:
       return node.sum
  else:
    if node.key>m:
       return psum(node.left,m)
  else
      return node.sum + psum(node.right,m)
```

   Partial_sum has a worst case runtime of O(logn) as the furthest that you will have to travel to change a value is the height of the tree, which is logn steps, due the the AVL tree property. The keys on the right will be unaccessed.

1c)

This function works by traversing the tree, either left or right depending on the i in a_i. When we reach the desired node, we replace its value with the y value in CHANGES(s,i,y). We get the difference between the old and new $a_i.value$. The difference will be the y- $a_i.value$ when you reach the target i. We then and go back to the root, then traverse back to a_i. Because in our partial sums function we always add the current node when we traverse to the right, we just need to working about the parents of the changed node. Everytime you go left, add the difference to each node's sum attribute. As this just involves traverseing the tree twice to the same node with only constant commands, this is O(logn).

Change is O(logn) because if you change a value, the maximum it has to traverse is logn steps. For example, if you have a leaf node, it will have to change the sum attribute for every node which is its parents, since it is an AVL. This results in a max of logn steps.

Programming Question

The best way to learn a data structure or an algorithm is to code it up. In each problem set, we will have a programming exercise for which you will be asked to write some code and submit it. You may also be asked to include a write-up about your code in the PDF/TEXfile that you submit. Make sure to **maintain your academic integrity** carefully, and protect your own work. The code you submit will be checked for plagiarism. It is much better to take the hit on a lower mark than risking much worse consequences by committing an academic offence.

2. [**12**] The function `num_orders` takes a list `lst` giving the insertion order of elements into an initially empty BST. For example, `[2, 1, 3]` means to insert 2, then insert 1, then insert 3. The function returns the total number of insertion orders (including `lst`) that produce the same BST that `lst` produces.

   Here is a sample call of `num_orders`:

```
>>> num_orders([2, 1, 3])
2
```

   The return value is 2 because there are 2 insertion orders, `[2, 1, 3]` and `[2, 3, 1]`, that produce the same BST as produced by `[2, 1, 3]`.

   Note that `lst` can contain duplicates. Let's agree that equal elements go into the left subtree (not the right subtree). For example, the root of the tree for the insertion sequence `[4, 4]` has 4 as its left node and an empty right subtree.

   Implement `num_orders`.

   Requirements:

   - Your code must be written in Python 3, and the filename must be `num_orders.py`.
   - We will grade only the `num_orders` function; please do not change its signature in the starter code. include as many helper functions as you wish.

   Write-up: in your `ps2.pdf/ps2.tex` files, include an explanation of how your code works. Please include a formal proof of correctness.

   SOLUTION:

   In this question we used information found here: https://stackoverflow.com/questions/17119116/how-many-ways-can-you-insert-a-series-of-values-into-a-bst-to-form-a-specific-tr

   Proof of Correctness:

   `num_orders(lst)`:
   Precondition: A list representing the insert order and values for a BST
   Postcondition: The number of distinct insertion orders that would create an identical BST.

We begin by checking the length of the input list, if the list has either 0 or 1 element, we quickly return 1. This is a valid return as the only insertion order for a 0 or 1 node binary tree has to be one.

If the list has a length of 2 or greater, we build an augmented binary tree (with functionality of having a node count) given by the insertion order and return the value of `num_orders_helper` when the tree is passed to it. This is valid as we will assume (and later prove) that `num_orders_helper` returns the correct value.

`num_orders_helper(bst)`:
Precondition: An augmented BST (possibly with no nodes) with count capability
Postcondition: The number of distinct insertion orders that will result in this tree.

path 1 (null bst):
First we check if the BST is null, and if it is we return 1. This is valid as the only way to create this null tree is 1 (not inserting anything).

path 2 (not null bst):
Notice that if we want to replicate any binary tree, we can insert items into the left sub-tree, and insert items in the right sub tree separately. But, the items in each subtree has to be inserted in the correct order to not violate the fact that parent nodes need to be inserted before their children nodes. But, we also have to account for every possible insertion sequence. For example: (insert left, insert right) and (insert right, insert left) will effectively do the same thing. We can account for this by multiplying the product of recursive calls with a value that will represent all of the possible valid orderings. This value was given to us by the link presented at the beginning of the question. The idea behind how the value is found is as follows:

Assume we have n L's and m R's where $n, m \in \mathbf{N}$ and L and R represent an insertion into the left or right sub-tree respectively. We will be inserting all the nodes, so we will have $n + m$ insertions. Normally for the permutations of $n + m$ elements would be $(n + m)!$ but because we do not differentiate between the L's and R's we need to account for this by dividing by $n!$ and $m!$ which represent the number of ways L and R could be arrange as well.

We return the product of the number of ways to insert the left subtree, the number of ways to insert the right subtree, and the number of ways to order those insertions. The recursive calls on the left and right subtree hold as the left and right children will either be a Node or null, which is accepted by our function, so we can assume they return the correct values. Therefore, this is a valid return.

The rest of the functions, are left unproven as they have been proven in previous courses.

3. **[12]** The function `num_trees` takes the total number of `nodes` and the number of `leaves`, and returns the number of **AVL-balanced** tree shapes with that many nodes and leaves.

   Here is a sample call of `num_trees`:

   ```
   >>> num_trees(5, 3)
   2
   ```

   This means that there are exactly two AVL-balanced trees that have five nodes where three of those nodes are leaves. Here are those two trees:

   Implement `num_trees`.

   Note: we're not asking you to implement any optimizations. As such, this thing really slows down when the number of nodes increases. We hope that your code can solve cases with 8 nodes or fewer in under a minute. It should of course be correct for larger numbers of nodes too, but it's OK if the time taken in these cases is prohibitive. (We're happy to talk to you about several possible optimizations if you're interested!)

   Requirements:

   - Your code must be written in Python 3, and the filename must be `num_trees.py`.

- We will grade only the `num_trees` function; please do not change its signature in the starter code. include as many helper functions as you wish.

Write-up: in your `ps2.pdf/ps2.tex` files, include an explanation of how your code works. Please include a formal proof of correctness.

SOLUTION:

In this question we used information found here: https://en.wikipedia.org/wiki/Heap%27s_algorithm

Our solution works by first creating a sequence from 1 to n, where n is the input 'nodes'. We then generate a list of all the possible permutations of 1 to n using Heap's Algorithm. We then, construct binary trees using each permutation as an insertion sequence. As this covers all insertion sequences, this produces a list of all binary trees with n nodes. From there, we add each tree if it matches our requirements: first, that it is an AVL and has the correct number of leaves, and second, that it is not a duplicate 'shape' of any tree already in our list.

Proof of Correctness:
`num_trees(nodes, leaves)` Pre: nodes, leaves (both natural numbers) Post: The number of distinctly shaped AVL's possible with nodes (as number of nodes) and leaves (as number of leaves)
path 1:
We check if the number of nodes is less than 1, if this is the case we quickly return 1. As each empty or non-empty tree can only be represented one way. So this is a valid return.
path 2:
First we call our function `get_perms()` which returns a list of all possible permutations from 1 to n. (where n is our 'nodes' input). We then construct a binary tree for each permutation. For each binary tree we check if the leaves of the tree matches the leaves request and that the Tree is an AVL. We check the AVL property by checking that each node in the tree has a balance factor within the inclusive range $[-1, 1]$. Then, another if condition checks if this tree is a 'duplicate' of any tree already in the list. We check for duplicates by confirming that each node in both trees has the exact same amount of children in each place. We insert the tree to our list of trees if both conditions are met. We then return the length of the list of correct trees.
As we have a list of all possible binary tree with 'nodes' nodes and 'leaves' leaves, that are AVL and have distinct shapes, our return value is valid.