CSC263 – Problem Set 2

Si Tong Liu(1004339628), Jing Huang(1003490705), Yifei Gao(1004152640)

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.

   Solution:

   (a) (1)The key(i) of each node is the index of the each a_n in the given sequence of number (2) psum :each node has the partial sum from 1 to i, where i is the current index(3) value: the number that is at the index i of the given sequence

   (b)

   ```
   def PARTIAL_SUM(S,m):
       if S.root is None:
           return 0        #when S is a None, sum will be 0
       if S.root.key == m:
           return S.root.psum    # psum as we defined in (a)
       elif m > S.root.key and S.right:
           return PARTIAL_SUM(S.right,m)
       elif m < S.root.key and S.left:
           return PARTIAL_SUM(S.left,m)
   ```

   explanation: for the worst case, we need to find the partial sum from 1 to the end of the given sequence. So index m = n, and m is the largest number now. Since it is a AVL tree, to find the m_{th} number, we only need to look at the right subtree. Therefore the running time is $\mathcal{O}(\log n)$.

   (c)

```
def CHANGE(S, i, y):
    if S is None:
        #if None, do nothing
    if S.root.key == i:
        S.root.value = y; # value as we defined in (a)
    elif i > S.root.key and S.right:
        CHANGE(S.right,i,y)
    elif i < S.root.key and S.left:
        CHANGE(S.left,i,y)
```

Once we run CHANGE function. Look at the input i, if i is less than S.root.key, we only need to look at the left subtree. Otherwise we only need to look at the right subtree. Therefore, the rumtime will be $\mathcal{O}(\log n)$.

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:

   For the helper function, it simply works as the Permutation: The number of ways about choosing x from y. And for the numorders function, we are using recursive method to count. To prove this:

   Path 1: the first return. If the length of input list is less than 1, there is only one way to build a BST: itself. holds

   Path 2: the second return. Since we know that the first node will always be the parent node. Then for the sublist which starts from the second element we divide them into left lists and right lists. Then by applying recursive rule, we can get the the total number of ways to build same BST is the multiplication of left and right combination.

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:

Since AVL-balanced tree, the difference of heights will not exceed 1. And during observation, the number of ways that we can generate trees are the multiplication of the bottom leaves and their parents. And every times we match one level of nodes, we should divide by two to get the number of parents in th tree.