Junwen Shen (1004299190), Jianhao Tian (1001354465), Mengning Yang(1002437552)

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?
   Answer:
   The key of each node in the AVL tree is a value pair like (i, j) which contains two index indicating that the current node stores the partial sum from a_i to a_j.
   Some attributes that stored in each node of the AVL tree are:
   1) the key, (i,j)
   2) partial sum, which stores the sum of a sequence of real numbers from a_i to a_j. if $i == j$, then it must be a leaf node where it only contains sum of one real number, namely, itself.
   3) the left child and the right child of the parent node.
   In our AVL tree, we store all our values from $a_1, a_2, \ldots, a_n$ in the leaf nodes, the first half, from element 1 to element $\frac{1+n}{2}$ are stored in the left sub-tree, the second half, from element $\frac{1+n}{2}$ to element n are stored in the right sub-tree, and all other nodes stores the partial sum of it's children.

   (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.
   Answer:

```
def partial_sum(S, m):
                if the key (i, j) where i == j:
                    return S.partial_sum
                elif i+j//2 < m:
                    return S.left.sub_sum + partial_sum(S.right, m)
                else:
                    return partial_sum(S.left, m)
```

   if the key (i, j) where i == j, which means we have reached the leafs, so we return it's attribute partial sum, namely, the value itself.
   else if $\frac{i+j}{2} < m$, which means the partial sum we are looking for is stored in the right sub-tree, so we keep looking in the right sub-tree and add the partial sum of the left sub tree to it.
   else, the partial sum we are looking for is stored in the left sub-tree, we simply keeps searching in the left sub-tree until the correct value is found.
   It's worst running time is $\mathcal{O}(\log n)$ because it breaks the problem size into half each time and it only searches into one path of the 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.

Answer:
Because all the values from $a_1, a_2, \ldots, a_n$ are stored in the leaves. If we want to change some values, just traverse to the leaf and change it. This operation takes $\mathcal{O}(\log n)$ time because before searching for the value, we compare it with current node's key pair $\frac{i+j}{2}$, if i is greater than $\frac{i+j}{2}$, call this function with current node's right sub-tree, because a_i must be in S's right sub-tree. Similarly, if i is less than $\frac{i+j}{2}$, call this function with current node's left sub-tree. Each recursive call the size we are searching shrinks in half. Changing the value takes constant time, and to update the partial sum, we also only need to update one path of nodes (ancestors of the node we changed) which the value we changed makes up their partial sum.
The following is our pseudo-code for this question:

```
def change(S, i, y):
                    if S.key[0] == S.key[1] == i: #note: the key is a pair (i, j)
                        S.partial_sum = y #the partial sum here is the value itself
                        return
                    if (S.key[0]+S.key[1])//2 < i:
                        change(S.left, i, y)
                        S.partial_sum = S.left.partial_sum + S.right.partial_sum
                    else:
                        change(S.right, i, y)
                        S.partial_sum = S.left.partial_sum + S.right.partial_sum
```

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 Code:

```
class Node():

      def __init__(self, data, left = None, right = None):
            self.data =  data
            self.left = left
            self.right = right

      def insert(self, data):
            if self.data < data:
                  if self.right == None:
                        self.right = Node(data)
                  else:
                        self.right.insert(data)
            elif self.data > data:
                  if self.left == None:
                        self.left = Node(data)
                  else:
                        self.left.insert(data)
            else:
                  if self.left == None:
                        self.left = Node(data)
                  elif self.left.data != data and self.right == None:
                        self.right = Node(data)
                  elif self.left.data == data and self.right == None:
                        self.left.insert(data)
                  elif self.left.data != data and self.right != None and self.right.data
                        self.right.insert(data)


      def is_leaf(self):
            if self.left == self.right == None:
                  return True
            return False





      def get_node_number(self):
            if self.left == None and self.right != None:
                  return 1 + self.right.get_node_number()
            if self.left != None and self.right == None:
                  return 1 + self.left.get_node_number()
            if self.left == self.right == None:
                  return 1;
            else:
                  return 1 + self.right.get_node_number() + self.left.get_node_number()



def generate_tree(lst):
      a = Node(lst[0])
      i = 0
```

```
    while i + 1 < len(lst):
        a.insert(lst[i + 1])
        i += 1
    return a


def get_list_combi(m, n):
    '''
    return the number of combination of two list with length m and n, without
    disrupt the order of each list
    '''
    if m == 1:
        return n + 1
    if n == 1:
        return m + 1
    else:
        return get_list_combi(m - 1, n) + get_list_combi(m, n - 1)

def get_orders(node):
    if node.is_leaf():
        return 1
    elif node.left == None and node.right != None:
        return get_orders(node.right)
    elif node.right == None and node.left != None:
        return get_orders(node.left)
    elif node.left.is_leaf() and node.right.is_leaf():
        return 2
    left_node_num = node.left.get_node_number()
    right_node_num = node.right.get_node_number()
    return get_orders(node.left) * get_orders(node.right) * get_list_combi(left_node_n


def num_orders(lst):
    bst_tree = generate_tree(lst)
    return get_orders(bst_tree)
```

Proof of Correctness:
1. For function `generate_tree`():

This function takes a list and generate a BST by the given list, loop invariant: i + 1 < len(lst), and for each literation, the loop invariant does not change. The function will generate len(lst) nodes according to lst's order.

2. For function `get_list_combi`():

This function takes two ints, which represents the length of two lists, and returns the number of combinations of two lists, without changing the inner order of each list. For the base case, when the length of first list is 1, then the combination of these two lists will be the number of blanks between the elements in the second list, which is n + 1. When the length of second list is 1, then the combination of these two lists will be the number of blanks between elements in the first list, which is m + 1.The base case is correct.
For the recursive step, first, choose the first element of first list to be the fron, then the combinations between the reset of the list with length (m - 1) and the second list with length n will be `get_list_combi`(m - 1, n). similarly, choose the first element of the other list to be the front, then the combinations between the two lists will be `get_list_combi`(m, n - 1). And the total number of combinations between two lists will either be the first element in the first list as the front or the first element in the second list as the front, which will be `get_list_combi`(m - 1, n) + `get_list_combi`(m, n - 1).

3. For function `get_orders`():

This function takes a BST Node, which generated by `generate_tree`() and returns the the total number of insertion orders that produce the same BST that a list can produces. For the base case, when the node is a single node, the function returns 1, which is correct. When the node does not has a left child and has a right child, the function will returns the number of combinations of the right child, similarly, the function returns the number of combinations of the left child when the node dose not has a right child and has a left child. When the node has two leaf child, the function will return 2.
For recursive step, assume the number of different list given by the left child is m, the number of different list given by the right child is n, then the total number of lists given by the root node will be m * n * (combinations between list given by the left child and list given by the right child), which will be m * n * `get_list_combi`(len(left list), len(right list)), where len(left list) can be represented by the number of nodes of left child and len(right list) can be represented by the number of nodes of right child. Therefor the function is correct.

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:
Overall, Our function first takes in a list with [total number of nodes] elements, then do a permutation of the list to get all different combinations of the list. Then we make a balanced AVL tree based on each list, and checks if number of leaves is equal to the number specified by the user, remove the trees that does not satisfy the requirement. Lastly, remove the AVL tree that has the same structures. Return the number of remaining trees.