Problem Set 2

All parts are due March 10, 2017 at 11:59PM.

Name: Faaya Abate Fulas

Collaborators: Yingni Hatty Wang, Ebenezer Nkwate

Part A

Problem 2-1. Submit this to gradescope.

(a)
- Call *in-order-traversal* while keeping track of the last visited node.
- If the key of last visited node is greater than the key of the current node to be visited, the tree is not a BST. This is because in-order traversal returns a sorted array in ascending order.

This takes $O(n)$ time since *in-order-traversal* takes $O(n)$.

(b)
1. Initialize an empty list L and an empty BST
2. Call *in-order-traversal* on the given BST while appending the visited nodes to L. L should now be a sorted array of the given BST's elements.
3. Find the midpoint of L and insert the element at that index into the empty BST.
4. Using binary search, recursively do Step 3 for the left half and right half of L.
5. The recursion stops when the lookup range = 0

The recurrence for this algorithm should be $T(n) = 2T(n/2) + \theta(1)$ since it is reducing the size of the problem by half on each recursion and finding the element at the midpoint of the lookup range and inserting it in constant time. Solving the recurrence using the Master Theorem yields $n^{\log_2 2} = \theta(n) > f(n) = \theta(1)$. So, the runtime complexity of the algorithm is $\emptyset(n)$.

Problem 2-2.

Data Structure: Let D be an augmented AVL tree. Each node in the AVL tree has:

- key = M
- value = k
- product = None

1.UPDATE(M, k):

- Call a modified version of *INSERT* on D which checks if *k* matches the key of an existing node in the AVL tree.
- If it doesn't, insert the new node N such that:
  - If (N.left and N.right are not None):
    N.Product = N.left.product * N.value * N.right.product
  - If (N.left and N.right are None):
    N.Product = M
  - If N.left is None and N.right is not:
    N.Product = N.value * N.right.product
  - If N.right is None and N.left is not:
    N.Product = N.left.product * N.value
- If it does, update the existing node E such that:
  - E.product = E.left.product * M * E.right.product (check existence of left and right children and follow the cases in the second bullet point)
  - current = E.parent
  - While current.parent is not None:
    current.product = current.left.product * M * current.right.product (check existence of left and right children and follow the cases in the second bullet point)
    current = current.parent
- Since *UPDATE(M,k)* is a modification of *INSERT*($\emptyset(\log D)$)) where the added computation of calculating products takes constant time and, for the case where keys match, updating the product values takes $\emptyset(h_D) = \emptyset(\log D)$ time , *UPDATE*'s running time is also $\emptyset(\log D)$).

2.*COMPUTE()*:

- Since the root of D contains all the products in the left subtree multiplied by its value multiplied by all the products in the right subtree:
  - return *D.root.product*
- Therefore, COMPUTE() runs in $\emptyset(1)$

Problem 2-3. Submit this to gradescope.

(a) Part a

- Initialize empty arrays, X and Y
- In $\emptyset(2n)$ time, go through the array A and add all the values in A that fall in the range of $[0, b)$ to X and add all the values in A that are $\geq b$ in Y.
- X.size = n-k and Y.size = k
- Sort X using radix sort in $\emptyset((n-k)+b)$ time. Since $n > k$, this will take $\emptyset(n+b)$ time.

- Sort Y using merge sort in $\emptyset(k \log k)$ time.
- Since the all the elements in Y are guaranteed to be larger than the elements in X, do X.extend(Y) in $\emptyset(k)$ time. The resulting list should be the sorted array of A.
- Overall, this takes $\emptyset(n + (n + b) + (k \log k) + (k))$. This simplifies to $\emptyset(n + b + k \log k)$ since $\emptyset(2n) = \emptyset(n)$ and $n > k$.

Problem 2-4. Submit this to gradescope.

(a) Let D be made of an augumented AVL Tree(avl) and a min-heap(H) which store the items in the sorted array A. Each node in the AVL tree has:

- key = A[i]
- count = 1

Every time a new node is inserted in the AVL Tree, its count is also updated such that node.count = 1+ node.left + node.right, so that the count value in each node reflects the number of nodes beneath it, including itself.

1. COUNT-LESS-THAN(x):
   FIND(x) is a function I implemented in part B that takes $\emptyset(\log n)$ time to find the smallest node in an AVL tree that has a key greater than or equal to x.
   - node = avl.FIND(x)
   - counter = 0

   Since all the elements on the left subtree of node are guaranteed to be less than its key:
   - if (node.left is not None):
     counter += node.left.count

   Next, if node is its parent right child, then all the elements in its parents left subtree are less than its key. This holds for all the nodes above 'node' until the root is reached. So:
   - current = node
   - While current.parent is not None:
     - if current.parent.right is current:
       counter += (1+current.parent.left.count)
   - current=current.parent

   When the while loop is done, the root has been reached, so:
   - return counter

   Since the While loop in COUNT-LESS-THAN(x) walks up the height of the tree, the algorithm runs in $\emptyset(h_{avl}) = \emptyset(\log n)$ time.

2. COUNT-GREATER-THAN(x):
   - Since the number of elements in A is stored in avl.root.count: COUNT-GREATER-THAN(x)= avl.root.count - COUNT-LESS-THAN(x)
   - Therefore, the algorithm runs in $\emptyset(\log n)$ time

(b) 1. INSERT(D,x):
   - $toneChange = 10 * \frac{\log(10^{this.d/10}+1)}{\log 10}$
   - Call Insert(x) on min-heap H. Running time: $\emptyset(logn)$
   - Call Insert(x) on AVL Tree where:
     - nodeX.count = nodeX.left.count + nodeX.right + 1 (check existence of left and right children and adjust sum accordingly)
     - current = nodeX.parent
     - While current.parent is not None:
       current.count = current.left.count + current.right.count + 1 (check existence of left and right children and adjust sum accordingly)
       current = current.parent
   - The algorithm will take $\emptyset(\log n)$ since the While loop walks up the height of the tree.

2. COUNT-OUTLIERS(D,z):
   - Since the min-heap is stored as a sorted array, indexing into locations should take constant time. So, we can find Q1 and Q3 and calculate the upper bound (UB) and lower bounds(LB) for determining outliers in constant time.
   - return COUNT-LESS-THAN(LB) + COUNT-GREATER-THAN(UB)

Part B

Problem 2-5. Submit your implementation on `alg.csail.mit.edu`.