Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Instructors: Jason Ku, Muriel Medard, and Silvio Micali

September 26, 2017
Problem Set 3

Problem Set 3

All parts are due on October 3, 2017 at 11:59PM. Please write your solutions in the LaTeX and Python templates provided. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. Solutions should be submitted on the course website, and any code should be submitted for automated checking on alg.csail.mit.edu .

Problem 3-1. [20 points] **Binary Search Tree Practice**

(a) [5 points] **Traversal:** A common operation on a binary search tree is to print its elements in sorted order. This can be done using an *in-order traversal*: first find and print the minimum element of the tree (e.g. with `find_min`), and then repeatedly find and print the next element (e.g. with `find_next`). Because `find_min` and `find_next` each take $O(h)$ time, where h is the height of the tree, one can naively calculate a $O(nh)$ upper bound on the time to print all n elements using in-order traversal. Show that in fact, in-order traversal requires at most $O(n)$ time.

Solution: Notice that this traversal only visits neighboring nodes, one after another (see line 6 in `find_min` and lines 3 and 6 in `find_next` from the recitation 4 notes). Whenever an edge of the tree is traversed upwards to a node (i.e line 6 in `find_next`), the entire left subtree of the node has been traversed and will not be visited again. Therefore, each edge is traversed upwards at most once, and is traversed downwards at most once as well. So, the total running time is $O(\#edges) = O(n)$.

Rubric:

- 5 points for a correct argument that sum of work done by the traversal is $O(n)$.
- Partial credit may be awarded.

(b) [10 points] **Property Checking:** Given the root node of a binary tree, describe $O(n)$ time algorithms that evaluate whether the tree satisfies each of the following properties. Assume for this problem that the only data stored at a node is the node's key, and pointers to its parent, left child, and right child.

- **BST Property:** the key of every node is greater than or equal to every key in the node's left subtree, and less than or equal to every key in the node's right subtree.
- **AVL Property:** every node's left and right subtrees differ in height by at most one.

Solution: BST property:
1st solution: We perform a traversal of the binary tree as in part a and print all the elements in the order we encounter them. Let's denote that sequence by $\{a_i\}$. This

operation will take $O(n)$ time according to part a. Then, we check if the elements we have printed are in an increasing order. This can also be done in $O(n)$ time by reading them one by one and verifying that $a_{i+1} > a_i$.

2nd solution (Using augmentation): Let r be the value of the root and $max(\cdot), min(\cdot)$ denote the maximum and minimum element in a given subtree. We define $max(\emptyset) = -\infty$ and $min(\emptyset) = +\infty$. A binary tree has the BST property if and only if both subtrees (T_l, T_r) rooted at the left and right child of the root respectively have the BST property and the following holds: $max(T_l) \leq r \leq min(T_r)$. Using this idea, we can use the following recursive algorithm to check for the BST property:

We first do the following **preprocessing** to augment our binary tree datastructure with $self.max$ (i.e the maximum in key in the node's subtree):

1. Find $max(T_l)$ (recursive call)
2. Find $max(T_r)$ (recursive call)
3. $self.max \leftarrow \max\{max(T_l), max(T_r), r\}$

Remark: The above preprocessing takes $O(n)$ time since the algorithm does constant amount of work per node.

Algorithm (BST check):

1. Check if T_l is a BST
2. Check if T_r is a BST (recursive call)
3. Check if $max(T_l) \leq r \leq min(T_r)$
4. If all three checks pass, output "YES", otherwise output "NO".

Similarly, above algorithm takes $O(n)$ time since the it does constant amount of work per node.

Remark: Note that without the preprocessing step, step 3 would take $O(\log n)$ time, which would cause the overall running time to be $O(n \log n)$.

AVL property:

Let $AVL\_height(\cdot)$ denote the length of the longest root to leaf path in the subtree rooted at the given node. We define $height(l) = 0$ if and only if l is a leaf. The following $AVL\_height(T)$ procedure, takes a binary tree T as input and either outputs its height (in case it is an AVL tree) or outputs "NO" if is not an AVL tree. Note that, a subtree of T is not being AVL implies that T is also not AVL.

1. Find $AVL\_height(T_l)$ (recursive call)
2. Find $AVL\_height(T_r)$ (recursive call)
3. **If** $|AVL\_height(T_l) - AVL\_height(T_r)| > 1$ **then return** "NO"
4. **Return** $\max\{AVL\_height(T_l), AVL\_height(T_r)\} + 1$

Rubric:

- 5 points for a correct BST Property checking algorithm.
- 5 points for a correct AVL Property checking algorithm.
- Partial credit may be awarded.

(c) [5 points] **Make AVL:** Consider the worst case binary search tree with no branching containing the keys $\{1, 2, 3, 4, 5, 6, 7\}$, with key 7 at the root. Show how to transform it into a binary search tree satisfying the AVL Property by performing a sequence of left and/or right rotations.

Solution:

```
Right     Right     Right     Left      Right
Rotate 7  Rotate 6  Rotate 5  Rotate 5  Rotate 3

     _7  =>  _6_  =>   _5_  =>   _4_  =>   _4__  =>  __4__
    _6      _5 7      _4 6_     _3 5_     _3 _6_    _2_ _6_
   _5      _4        _3   7    _2   6_   _2  5 7    1 3 5 7
  _4      _3        _2         1     7   1
 _3      _2          1
_2       1
1
```

Rubric:

- 5 points for a sequence concluding in tree satisfying AVL.
- -1 point for each rotation problem (not a rotation, or done incorrectly.
- Any sequence of valid rotations leading to a tree satisfying the AVL property will do. The above sequece of five rotations balances the tree completely, though four rotations are necessary and sufficent.

Problem 3-2. [30 points] **Consulting**

Briefly describe systems that can help each of the following clients quickly solve their problems. By "quickly", we mean at most linear with respect to the size of outputs and at most logarithmic with respect to other problem parameters. For example, if you need to return k elements from a set of n elements, you should use at most $O(k + \log n)$ time to do so.

(a) [10 points] **Secured Shipping:** Fred runs the FredEx shipping company. Throughout the day he receives shipments to his warehouse, each shipment contained in identical regulation-sized packaging. As packages arrive, scanners read each package's ID, address, and the value of its contents. This data must be quickly entered into an inventory. Fred has hired a trucking company to pick up and deliver the packages. Each day, the trucking company sends two trucks. One is a regular truck that has very large capacity and can easily hold an entire day's inventory. The other truck is smaller, but well armored and secure. It's cargo is shared with other companies, so Fred only knows how many packages it will be able to carry when it arrives. Each day, Fred wants to ship as many of the most valuable packages from his inventory as he can inside the armored truck, and ship the remainder in the regular truck. Help Fred design his inventory so he can quickly determine which items to ship in the armored truck each day.

Solution: Tell Fred to store his inventory information in a balanced binary search tree (e.g. AVL) sorted by the value of the package. Inserting and deleting from the inventory directly carries over from analogous operations on e.g. AVL. To return a list of the most valuable k packages, simply find the package with maximum value in the tree in $O(\log n)$ time, and then repeatedly walk backward in the tree in a reverse-order traversal through the $k - 1$ previous records, returning them as you go. To show that this procedure is $O(k + \log n)$, consider the path from the root to the node containing the kth largest value v. This path has length $O(\log n)$. All nodes to the left of the path contain keys that are smaller than v while nodes to the right of the path contain keys larger than k. The path together with all the nodes to its right constitutes a binary tree on $O(k + \log n)$ nodes which, by Problem 3-1a, can be traversed in $O(k + \log n)$ time.

Rubric:

- 1 point for mentioning a balanced BST.
- 1 point for sorting the balanced BST by e.g. package value.
- 2 points for mentioning insertion into the inventory is quick.
- 3 points for a correct algorithm for query.
- 3 points for any runtime analysis (tough, so informal arguments may be accepted)
- Partial credit may be awarded.

(b) [10 points] **Cloud Computing:** Bev Jezos runs a cloud computing company that rents server time to customers around the world. She has a fixed number of servers, each with a fixed amount of bandwidth. Many times per second, some customer makes a request to the system for a fixed amount of bandwidth to be distributed evenly across a requested number of servers, for a fixed amount of time. Each server holds a reference to its record in the system and is responsible for telling the system when the server's available bandwidth changes. Help Bev design a system that will quickly allow her to either request bandwidth on available servers, or to respond that there are not enough servers available to fulfill the request.

Solution: Tell Bev to store server records in a balanced binary search tree (e.g. AVL) sorted by current available bandwidth. When a server needs to inform the system of a change in available bandwidth, the server can remove the record from the tree in $O(\log n)$, update the record in $O(1)$, and then reinsert into the tree in $O(\log n)$. To find k servers to handle a request needing bk bandwidth, it suffices to find out if the k servers with the highest available bandwidth can each accept an increase by b. To return the list of the k servers with the highest available bandwidth, find the server with the most available bandwidth in $O(\log n)$ time, and then repeatedly walk backward in the tree in a reverse-order traversal through the $k-1$ previous records, returning them as you go. A similar analysis as Problem 3-2a leads to a $O(k + \log n)$ running time.

Rubric:

- 1 point for mentioning a balanced BST.
- 1 point for sorting the balanced BST by e.g. available bandwidth.
- 3 points for a correct algorithm for updating bandwidth.
- 2 points for a correct algorithm for query.
- 3 points for correct runtime analysis (again, informal argument is OK).
- Partial credit may be awarded.

(c) [10 points] **Holiday Sales:** Holly Mark runs a website that sells cheap gifts targeted at office holiday gift giving. She wants to make it easy for shoppers to stay within budget. Each holiday season, merchants can upload price and product information for new gift items, while customers can ask for listings of a desired number of the most expensive items they can buy, below a certain price. Unfortunately, she currently stores all the product data in an unsorted array. This is great for her merchants as they can upload new products quickly; but her customers are frustrated because they have to wait a long time for product listings to load. Help Holly improve her database to quickly satisfy both merchants and customers.

Solution: Tell Holly to store the gift items in a balanced binary search tree (e.g. AVL) sorted by price. Insertion directly carries over from e.g. AVL. To return a list of the k most expensive gift items, first find the most expensive item with price less than the query, `max_under` in $O(\log n)$ time. This can be done using a modified version of `find`: if no item has the queried price, the last node in the tree visited by `find` will either be larger or smaller than the queried price. If it is less, the branching of `find` guarentees that no other item in the tree has price between that item and the queried price; if it is larger, then we `find_prev`, analogous to `find_next` also in $O(\log n)$. Then repeatedly walk backward in the tree in a reverse-order traversal through the $k - 1$ previous records, returning them as you go. A similar analysis as Problem 3-2a leads to a $O(k + \log n)$ running time.

Rubric:

- 1 point for mentioning a balanced BST.
- 1 point for sorting the balanced BST by e.g. price.
- 3 points for a correct algorithm for finding `max_under`.
- 2 points for a correct algorithm for query.
- 3 points for correct runtime analysis (again, informal OK).
- Partial credit may be awarded.

Problem 3-3. [50 points] **Transaction Log:** Barren Wuffett owns a small investment firm that trades a single volatile mutual fund. Hundreds of times a second, brokers buy and sell shares of the fund at different prices as the price dramatically fluctuates throughout the day. At any given time, Barren wants to know how many transactions were traded that day within a given price range. Help Barren design his transaction log; maybe he will share his profits with you! The transaction log must support three operations:

- `add_transaction`: adds a single transaction into the log containing the the number of shares traded at a given price. A negative price means a broker sold shares, and a zero or positive price means shares were purchased.
- `sold_in_range`: for a given inclusive price range, return the total number of shares that brokers sold at a price within the given range.
- `bought_in_range`: for a given inclusive price range, return the total number of shares that brokers bought at a price within the given range.

You want to support all of these operations in $O(\log n)$ time for a transaction log containing n transactions, so you decide to use a **single** modified AVL tree to store all the transaction information. Code implementing an AVL tree is provided in the problem set template. When describing algorithms, you may describe in terms of any of the BST or AVL operations discussed in class.

(a) [8 points] **Augmentation:** The keys in our modified AVL tree will be `Transaction` objects, not integers, containing additional information that might be useful. For example, BST nodes are often augmented to store properties of the subtree rooted at the node, such as: subtree height or skew (already in AVL); number or sum of keys in the subtree; subtree min, median, mean, max, etc. In addition to the price and number of shares for each transaction, describe any additional information you might want to store at a node. Would you need to update this information if the tree changes shape? You may want to think about the other parts of this problem before answering.

Solution:

The problem can be solved in many different ways, but probably any efficient solution will store some subtree totals. It's possible to solve this problem only by storing subtree share totals. Another choice is to augment a `Transaction` object to include the total number of shares in the subtree of the node associated with a `Transaction` object in addition to the minimum and maximum value of price contained in the subtree. Subtree totals and subtree min/max can be calculated from the respective properties of left and right children, assuming they correct; so updating them can be done at the same time that heights are updated when AVL rebalancing occurs.

Rubric:

- 4 points for mentioning subtree totals and any additional
- 4 points for correct discussion of update

- Partial credit may be awarded

(b) [3 points] **Comparison:** In order to store `Transactions` in a tree satisfing the BST Property, we need to know how to compare them. Describe how you want your modified AVL tree to order the `Transactions` it contains.

Solution: Again, multiple solutions possible. The most common two will be to sort the `Transactions` by price, or absolute value of price.

Rubric:

- 3 points for any ordering that leads to a correct algorithm.

(c) [7 points] **BST Range:** Let p be a node of a binary search tree containing key k_p, and let $[a, b]$ be an inclusive query range. If p's key is outside the query range ($k_p < a$ or $b < k_p$), it is not difficult to show that at most one of it's left or right subtrees can contain keys in the query range. If p's key is inside the query range $(a \leq k_p \leq b)$, let d be a descendant of p whose key k_d is also inside the query range; and let ℓ and r be the left and right child of d respectively. Of the subtrees rooted at ℓ and r, show that at most one of them can contain keys both inside and outside the query range.

Solution: Suppose for contradiction that both subtrees rooted at ℓ and r contain keys both inside and outside the query range. If d's key k_d is less than or equal to p's key k_p, by the BST Property r's subtree can only contain keys within the query range because every key in r's subtree has a key no larger than k_p and no smaller than k_d, and k_p and k_d are both within the range. A symmetric argument shows if $k_p \leq k_d$, then ℓ's subtree is within the query range, leading to a contradiction.

Rubric:

- 7 points for a correct proof
- Partial credit may be awarded

(d) [7 points] **Summing Shares:** Describe an algorithm to return the number of shares sold within a given inclusive price range using an augmented AVL tree. If the number of shares output by the algorithm is k, there are many algorithms that can output a correct result in $O(k + \log n)$. We are looking for a $O(\log n)$ algorithm, that does not depend on k. **Hint:** use the result from part (c) to help your analysis.

Solution: Again, multiple correct approaches. One $O(n)$ approach is a divide and conquer strategy. Recursively find the number of shares sold in the left subtree and right subtree, sum them together, and add the shares at the current node if its shares were sold. Constant work is done to combine subproblems, with one subproblem per node. However, we can do better by storing subtree totals, and minimum and maximum prices at each node, and then shortcutting any recursive calls on subtrees that contains `Transactions` with prices that either (1) fall entirely outside the query price range, or (2) fall entirely within the query price range, by simply returning zero or the stored subtree total, respectively. Consider the highest node with a key inside the query range. It may make recursive requests of both of its children. But by the result in (c), each of its descendents can make at most one recursive call. So $O(\log n)$ recursive calls may occur down the left side of the range, and $O(\log n)$ recursive calls may occur down the right side of the range, leading to a running time of $O(\log n)$.

Rubric:

- 7 points for a correct algorithm
- Partial credit may be awarded

(e) [25 points] **Implement:** Complete the `Transaction` and `TransactionLog` Python class templates provided to implement the transaction log you described above. Please submit code for this problem in a single file to alg.csail.mit.edu. ALG will import only your class implementations of `Transaction` and `TransactionLog`; do not include classes `BST` or `AVL`.

Rubric:

- This problem is graded by ALG.

```
from AVL import AVL

class Transaction():
    def __init__(self, shares, price):
        self.shares = shares
        self.price = price
        ##############################################
        # Part (a): Add any additional storage here #
        ##############################################

    def __lt__(self, other):
        "Evaluate comparison between two transactions: (self < other)"
        ##########################
        # Part (b): Implement me #
        ##########################
        return True              # return a boolean

    def __str__(self):
        return str(self.shares) + '@' + str(self.price)

class TransactionLog(AVL):
    def add_transaction(self, shares, price):
        "Adds a transaction to the transaction log"
        super().insert(Transaction(shares, price))

    def update(self):
        "Augments AVL update() to fix any properties calculated from children"
        super().update()
        ##################################################
        # Part (a): Add any additional maintenence here #
        ##################################################

    def sold_in_range(self, range_min, range_max):
        "Returns the number of shares sold within an inclusive price range"
        if self.key is None:
            return 0
        count = 0
        ##########################
        # Part (d): Implement me #
        ##########################
        return count

    def bought_in_range(self, range_min, range_max):
        "Returns the number of shares bought within an inclusive price range"
        if self.key is None:
            return 0
        count = 0
        ##########################
        # Part (d): Implement me #
        ##########################
        return count
```

Solution:

```
from AVL import AVL

class Transaction():
    def __init__(self, shares, price):
        self.shares = shares
        self.price = price
        self.submin = None
        self.submax = None
        self.subtotal = None

    def __lt__(self, other):
        return self.price < other.price

    def __str__(self):
        return str(self.shares) + '@' + str(self.price)

class TransactionLog(AVL):
    def add_transaction(self, shares, price):
        node = super().insert(Transaction(shares, price))

    def update(self):
        super().update()
        self.key.submin = self.left.key.submin  if self.left  else self.key.price
        self.key.submax = self.right.key.submax if self.right else self.key.price
        self.key.subtotal = self.key.shares
        for child in [self.left, self.right]:
            if child is not None:
                self.key.subtotal += child.key.subtotal

    def sold_in_range(self, range_min, range_max):
        return self.bought_in_range(-range_max, -range_min)

    def bought_in_range(self, range_min, range_max):
        if self.key is None:
            return 0
        count = 0
        if range_min <= self.key.price <= range_max:
            count = self.key.shares
        for child in [self.left, self.right]:
            if child is not None:
                if (range_min <= child.key.submin) and (child.key.submax <= range_max):
                    count += child.key.subtotal
                elif not (child.key.submax < range_min) and not (range_max < child.key.submin):
                    count += child.bought_in_range(range_min, range_max)
        return count
```