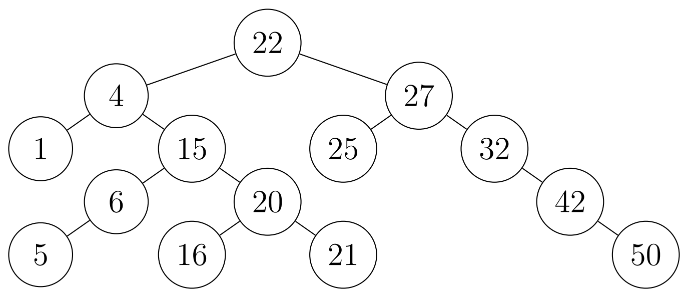
Problem Set 4

All parts are due on October 4, 2018 at 11PM. 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.mit.edu`.

Problem 4-1. [12 points] **Binary Tree Practice**

(a) [5 points] Perform the following operations in sequence on the binary search tree T below. Draw the modified tree after each operation. Note that this tree is a BST, not an AVL. You should not perform any rotations in part (a). For this problem, keys are items.

1. `insert(18)`
2. `delete(5)`
3. `insert(26)`
4. `delete(32)`
5. `delete(15)`



Solution:

```
1)    ___________22____             _____________22____
     _4____        __27__      =>  _4____          __27__
     1  _15____    25  32__        1  _15______    25  32__
       _6  __20__        42__        _6  ____20__        42__
       5   16  21          50        5   16__  21          50
                                           18

2)    _____________22____      =>    ____________22____
     _4____          __27__         _4___          __27__
     1  _15______    25  32__       1 _15______    25  32__
       _6  ____20__        42__       6  ____20__        42__
       5   16__  21          50          16__  21          50
             18                            18

3)    _____________22____      =>    ____________22______
     _4____          __27__         _4___          ____27__
     1  _15______    25  32__       1 _15______    25__  32__
       _6  ____20__        42__       6  ____20__    26    42__
       5   16__  21          50          16__  21            50
             18                            18
```

```
4)    ____________22______     =>    ____________22______
     _4___          _____27__       _4___          _____27__
     1 _15_______   25__  32__      1 _15_______   25__  42__
       6  _____20__   26    42__      6  _____20__   26    50
          16__  21            50         16__  21
            18                             18
```

```
5)    ____________22______     =>    __________22______
     _4___          _____27__       _4___        _____27__
     1 _15_______   25__  42__      1 _16____    25__  42__
       6  _____20__   26    50        6  __20__    26    50
          16__  21                       18  21
            18
```

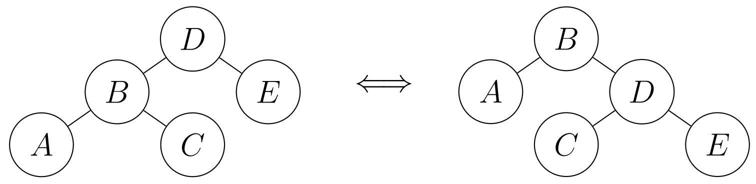
Rubric: 1 point for each operation

(b) [3 points] In the original tree T (prior to any operations), list keys from nodes that are **not height balanced**, i.e., the heights of the node's left and right subtrees differ by more than one and do not satisfy the AVL Property.

Solution: Keys: 4, 27, 32

Rubric: 1 point per correct node

(c) [4 points] A **rotation** locally re-arranges nodes of a binary search tree between two states, maintaining the binary search tree property, as shown below. Going from the left picture to the right picture is a right rotation at D. Here, we call D the **root of rotation**. Similarly, going from the right picture to the left picture is a left rotation, with B as the root of rotation. (Nodes A, C, and E may or may not exist). Perform a sequence of at most two rotations to make the original tree T height balanced, to satisfy the AVL Property. To indicate your rotations, draw the tree after each rotation, **circling the root** of each rotation.



Solution: Rotating 4 and 32 leftward restores the AVL property everywhere.

```
 _____________22____                ________22____
_(4)____        __27__           ___15____    __27____
1    _15____    25  32__    =>  _4__  __20__  25  (32)__
    _6  __20__        42__      1 _6  16  21          42__
    5   16  21          50        5                     50
```

```
        ________22____
     ___15____    __27____
=>  _4__  __20__  25  __42__
    1 _6  16  21      32  50
      5
```

Rotating 27 instead of 32 also works (not pictured).

Rubric:

- 1 point for each rotation
- 1 point for each circled root

Problem 4-2. [33 points] **Consulting**

Briefly describe a database for each of the following clients. When there are n items in the database, each supported operation should take **worst-case** $O(\log n)$ time. As always, remember to argue correctness and running time.

(a) [10 points] **Mary's Lambs:** Mary sells fleece from her flock of lambs, which are always white as snow. A higher quality fleece always sells for a higher price, and no two have the same quality. The customers in her town don't have a lot of money. When they come to Mary's store to buy fleece, they tell her the maximum price they want to pay for a fleece. Mary will then show them the 10 highest-quality fleeces from her inventory that are within the customer's budget, so that the customer may choose from among them. Design a database which maintains her inventory, allowing her to both add a fleece to her inventory, and remove from the inventory the 10 highest-quality fleeces whose price is no more than the customer's maximum price.

Solution: Mary should store the prices of her fleeces in a balanced binary search tree (e.g., an AVL tree). When the customer arrives, Mary uses `find(b)` to locate the fleece whose price exactly matches the customer's budget b, and if no such fleece exists, she uses `find_prev(b)` to locate the highest-quality fleece that is below budget. She can then call `predecessor` or `find_prev` 9 times to get the 9 highest-quality fleeces below this first one, finishing the desired query. We've performed at most eleven $O(\log n)$ operations, so this takes $O(\log n)$ time total.

Rubric:

- 8 points for a correct data structure and argument
- 2 points for runtime analysis
- Partial credit may be awarded

(b) [14 points] **Infinite Printers:** In the future, MIT's Infinite Corridor has extended to be even more infinite, with Athena clusters stationed at many locations along the **linear** hallway. Each Athena cluster has a unique integer **address** corresponding to its location along the Infinite Corridor, and contains many plasma printers which students utilize to print their futuristic homework via the online Pharos printer system (largely unchanged from its current implementation). Each printer has a **unique integer print quality** denoting how well it prints, though printers often go offline when they run out of resources, preventing students from printing to them until resources are replenished. Students want Pharos to tell them the best place to print. Design a print server which maintains the set of **working** printers, supporting the following operations: given a printer's cluster address and quality, take the printer online or offline; and given a student's location along the Infinite, return the quality of the **highest-quality working** printer, from among all working printers with **cluster address closest** to the student, keeping in mind there might be two clusters tied for closest.

Solution: First Solution (tree of pairs): Store an AVL tree of **working** printers, keyed on the pair `(address, quality)`. Keys are sorted by address first (in increasing order) and then quality (in increasing order). To take a printer online or offline, we can `insert` or `delete` the node from the tree, in $O(\log n)$ time.

To query an address a, define

`p = find_prev((a, infinity))` and `q = find_next((a, -infinity))`.

Finally, let `r = find_prev((q.address, infinity))`. Return whichever of p or r is closer, or both if they're the same distance away.

To show that this returns the correct printer(s), first observe that p is the working printer that has highest address $\leq a$ and, of those, the one with highest quality. So p is the desired printer in the backward direction. Similarly, q is the working printer with least address $\geq a$ and, of those, the one with **least** quality. This is not the most desirable printer in the forward direction, but it has the closest forward address of any printer ahead of a. Printer r is the highest-quality printer with the same address as q and is thus the most desirable printer in the forward direction. This performs up to three AVL tree operations and thus takes $O(\log n)$ time.

Second Solution (tree of trees): Store an AVL tree T whose items correspond to the clusters that have at least one working printer, keyed on address. Each cluster item stores its own AVL tree of its working printers, keyed on quality.

To insert a printer, first `find` the cluster item at the correct address (or create it if there isn't one already), and then `insert` the printer into that cluster's tree. To delete, we similarly `find` the cluster at the correct address and `delete` that printer from the tree, but in addition, if that cluster now has no working printers (its tree is empty), we `delete` that custer from T. This maintains the invariant that every cluster in T has at least one working printer.

To query an address a, we find the closest forward and backward clusters with `find` or, if there is no cluster directly at a, with `find_next` and `find_prev`. For whichever cluster is closer (or both), return the highest-quality printer from that cluster with `find_max`.

There are at most n clusters and each cluster contains at most n printers, so each AVL tree operation on any of the trees is $O(\log n)$. This gives $O(\log n)$ time overall.

(c) [14 points] **Eighth Mailer, kth Mailer:** Not to be outdone by WBST (6.042 FM) Radio's "10th caller wins" sweepstakes, Syan Recrest at WAVL (6.006 FM) Radio[1] decides to host a contest he calls "Eighth Mailer, kth Mailer". During the contest, listeners will send in postcards featuring adorbable animal pictures. On a surprise day next month, Syan will choose a random number k (between 1 and the number of postcards received) and award a trip to the Puppy Petting Zoo to the senders of the eighth postcard and the kth postcard, when ordered by **postmark**—the time and date the postcard was **sent** (you may assume postmarks are **distinct and easily comparable**). Unfortunately, the post office is very unreliable, so postcards will be delayed and will not be received in postmark order. Design a database for Syan's contest that supports two operations: recording a newly received postcard, and returning the **sender** of the kth postcard in postmark order.

Solution: Have Syan store the postcards in an AVL tree, sorted by postmark. Augment each node with the count of the number of nodes in the subtree. This augmentation can be computed from a node's children in constant time because $\text{count}(x) = \text{count}(x.\text{left}) + \text{count}(x.\text{right}) + 1$, so by updating this augmentation on every call to `update`, this value can be maintained through `insert` and `delete` operations.

Inserting postcards can be performed by normal AVL `insert` and `delete` in $O(\log n)$ time. To find the kth postcard, begin at the root, and let c be the subtree count of the root's left child. If $c \geq k$, there are more than k postcards before the root, so simply recursively query the kth postcard in the left subtree. If $k = c + 1$, then the root itself is the kth postcard, so return the root. Otherwise, recurse on the right subtree, looking for the $(k - c - 1)$th postcard in that subtree. This query process takes constant time at each level, so it runs in $O(\log n)$ time as desired.

Rubric: Parts (b) and (c):

- 11 points for a correct data structure and argument
- 3 points for runtime analysis
- Partial credit may be awarded

[1]While there are radio stations named WBST (92.1 FM) in Muncie, Indiana, and WAVL (92.1 FM) in Pittsburgh, Pennsylvania, any resemblance is purely coincidental.

Problem 4-3. [55 points] **Programming**

Jeeve Stobs, the CEO of the infamous Pineapple company, is trying to boost sales in his new product, the Pineapple Pen$^{\text{TM}}$. Jeeve has noticed that sales vary a lot depending on the time of day, and that if sales are low at a specific time of the day, they are also likely to be low at the same time the following day. In order to boost sales, Jeeve decides to start a store-wide daily sale at hours when he expect sales to be low. Jeeve decides to implement a system in which the Pineapple store can store the transactions made throughout the day, and then Jeeve can query the system to figure out how much revenue was made during a specific time interval. In Jeeve's implementation, the Pineapple storage system keeps track of all the day's transactions in an array of transaction items, where transaction `x` contains a transaction time `x.t` and the number of dollars `x.d` received during the transaction. Based on this information, Jeeve can calculate the revenue earned in a given **inclusive** time interval $[t_1, t_2]$ by scanning through all the transactions, and adding up the revenue from all the transactions made within the time interval. Here is Jeeve's implementation for his system:

```python
class TransactionLog:
    def __init__(self):
        self.transactions = []

    def add_transaction(self, x):
        self.transactions.append(x)

    def interval_revenue(self, t1, t2):
        total_revenue = 0
        for x in self.transactions:
            if t1 <= x.t <= t2:
                total_revenue += x.d
        return total_revenue
```

(a) [2 points] What is the running time of Jeeve's `add_transaction` function and `interval_revenue` function in terms of the number of transactions stored?

Solution: Jeeve's `add_transaction` algorithm runs in (amortized) $O(1)$ time, while the running time of his `interval_revenue` function is $O(n)$ when storing n transactions, because he reads through all of them.

Rubric:

- 1 point for $O(1)$ and $O(n)$
- 1 point for saying the $O(1)$ is amortized

To improve his implementation, Jeeve decides to instead store each transaction `x` in an AVL tree keyed by transaction time, where `x.key = x.t`. By the BST property, the subtree rooted at a node contains all the keys within a consecutive interval of time, between its subtree's minimum and maximum keys. In this problem, we will develop a recursive algorithm to efficiently compute the total revenue from transactions contained within a subtree whose times also fall within a query interval $[t_1, t_2]$.

(b) [5 points] First, suppose that **every** transaction within the subtree rooted at a node p occurred within the query interval $[t_1, t_2]$. Show that you can store information at p that will enable you to return the total revenue earned by all transactions within the subtree in constant time, and that this node augmentation can be maintained at all nodes during `insert` and `delete` operations without changing their asymptotic performance.

Solution: Augment each node with `node.Dsum`, the sum of dollar amounts for every node in its subtree. This augmentation can be maintained from its children in constant time with

```
node.Dsum = node.left.Dsum + node.d + node.right.Dsum
```

where terms are omitted for missing children. So by updating this augmentation at the same time that each AVL node's `height` and `skew` values are updated, the `Dsum` values can be kept up to date through insert and delete operations. With this augmentation, the total revenue in `p`'s subtree can be found in constant time by simply looking up `p.Dsum`.

Rubric:

- 2 points for a correct augmentation
- 3 points for explanation of maintaining the augmentation
- Partial credit may be awarded

(c) [5 points] Now suppose that **some but not all** transactions within the subtree rooted at p occured within the query interval $[t_1, t_2]$. Show that you can store information at p that will enable you to determine whether the subtree rooted at a child of p could contain (or could not contain) transactions within the query interval in constant time, and that this node augmentation can be maintained at all nodes during `insert` and `delete` operations without changing their asymptotic performance.

Solution: Have each node store `node.Tmin` and `node.Tmax`, the minimum and maximum `t`-value occurring in its subtree. These can be maintained from a node's children in constant time: `node.Tmin` is `node.left.Tmin`, or `node.t` if there is no left child, and `node.Tmax` is similar. So this is a valid augmentation.

When querying interval $[t_1, t_2]$ on node p, we can compare the query interval with the interval `[node.Tmin, node.Tmax]`. If these intervals do not overlap, then p's entire subtree can safely be ignored.

Rubric:

- 2 points for a correct augmentation
- 3 points for explanation of maintaining the augmentation
- Partial credit may be awarded

(d) [5 points] Describe a recursive algorithm `p.interval_revenue(t1, t2)` that uses part (c) to determine how to recurse, and part (b) as a base case to compute total revenue from transactions within `p`'s subtree occuring within query interval $[t_1, t_2]$.

Solution: If the interval `[p.Tmin, p.Tmax]` (call this "p's interval") is entirely contained in $[t_1, t_2]$, then we can simply return `p.Dsum`. Otherwise, the answer is the sum of `p.d` (or 0 if `p.t` is outside the query interval), `p.left.interval_revenue(t1, t2)`, and `p.right.interval_revenue(t1, t2).` We can skip one or both of these recursive calls if the child doesn't exist or has an interval disjoint from $[t_1, t_2]$, because that child's contribution is 0. We can also avoid a recursive call if the child's interval is fully contained in $[t_1, t_2]$, since that child's contribution is `child.Dsum`.

Rubric:

- 4 points for a correct algorithm and explanation, even if it recurses more than necessary
- 1 point for taking advantage of both of the "easy" cases where a subtree is entirely contained within $[t_1, t_2]$ or is disjoint from it
  - Note: it is fine to make a recursive call to an easy case if that recursive call returns in $O(1)$ time.
- Partial credit may be awarded

(e) [5 points] Consider calling `r.interval_revenue(t1, t2)` using your implementation from (d), where `r` is the root of the tree. Prove that, during this execution, at most one recursive call `p.interval_revenue(t1, t2)` (for only one node `p`) will make two recursive calls. What makes node `p` special with respect to the query range?

Solution: The recursive call `p.left.interval_revenue(t1,t2)` is only made if `p.left` exists and has an interval that intersects $[t_1, t_2]$ but is not entirely contained within $[t_1, t_2]$. The same is true for `p.right`. So if two recursive calls are needed at p, it must be true that t_1 is contained within `p.left`'s interval and t_2 is contained within `p.right`'s interval. Also, we must have $t_1 >$ `p.left.Tmin` and $t_2 <$ `p.right.Tmax`, so that $[t_1, t_2]$ doesn't fully contain either child's interval. Call these the "containment" and "inequality" conditions, respectively.

The containment condition shows that `p.left.Tmax` $\leq t_2$, and therefore `q.Tmax` $\leq t_2$ for any descendant q of `p.left`. But this means every node in p's left subtree fails the inequality condition! Nodes in p's right subtree fails the inequality condition for similar reasons. So no descendant of p requires two recursive calls.

Node p is special with respect to $[t_1, t_2]$ because it is the lowest node whose interval contains $[t_1, t_2]$ strictly at both ends.

Rubric:

- 5 points for a correct explanation
- Partial credit may be awarded

(f) [3 points] Argue that your algorithm runs in $O(\log n)$ time.

Solution: Since at most one node can recurse to both of its children, the set of nodes where `interval_revenue` is called forms either a path down the tree or an upside-down Y-shape, where one path splits into two paths. This set of nodes is contained in at most two root-to-leaf paths (that may initially overlap), so there are at most $2 \cdot h = O(\log n)$ such nodes. Each call to `interval_revenue` does constant work excluding the recursive call(s), so there is $O(\log n)$ work done in total.

Rubric:

- 3 points for a correct explanation
- Partial credit may be awarded

(g) [25 points] Implement method `interval_revenue(t1, t2)` in a Python class `TransactionLog` that extends the AVL class provided. Each added transaction has two attributes: `x.key`, the time of the transaction, and `x.d`, the revenue associated with the transaction. (You may assume that times and revenues are integers). You can download a code template containing some test cases from the website. Submit your code online at `alg.mit.edu`.

Solution:

```python
from AVL import AVL

class TransactionLog(AVL):
    def __init__(self, item = None, parent = None):
        super().__init__(item, parent)
        self.Tmin, self.Tmax, self.Dsum = None, None, None

    def _update(self):
        super()._update()
        self.Tmin = self.left.Tmin  if self.left  else self.item.key
        self.Tmax = self.right.Tmax if self.right else self.item.key
        self.Dsum = self.item.d
        for child in (self.left, self.right):
            if child is not None:
                self.Dsum += child.Dsum

    def add_transaction(self, x):
        super().insert(x)

    def interval_revenue(self, t1, t2):
        if self.item is None:
            return None
        revenue = self.range_query(t1, t2)
        return revenue

    def range_query(self, t1, t2):
        range_revenue = 0
        # add in key of root if in range
        if t1 <= self.item.key <= t2:
            range_revenue = self.item.d
        for child in (self.left, self.right):
            if child is not None:
                # add subtree if completely in range
                if (t1 <= child.Tmin) and (child.Tmax <= t2):
                    range_revenue += child.Dsum
                # recurse on subtree if partially in range
                elif not (child.Tmax < t1) and not (t2 < child.Tmin):
                    range_revenue += child.range_query(t1, t2)
                # otherwise out of range
        return range_revenue
```