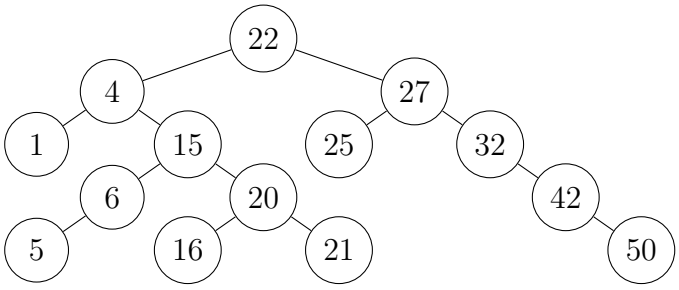
Problem Set 4

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**

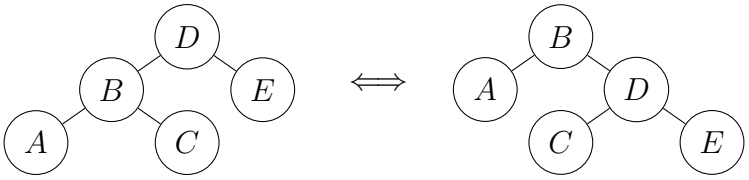
(a) [5 points] Perform the following operations in sequence on the binary search tree T below. Draw the modified tree after each operation. Note that this tree is a BST, not an AVL. You should not perform any rotations in part (a). For this problem, keys are items.

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



(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.

(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.



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.

(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.

(c) [14 points] **Eighth Mailer, *k*th 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.

[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™. 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:

```
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?

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.

(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.

(d) [5 points] Describe a recursive algorithm `p.internal_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 $[t_1, t_2]$.

(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?

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

(g) [25 points] Implement method `internal_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`.

```
class TransactionLog(AVL):
    def __init__(self, item = None, parent = None):
        "Augment AVL with additional attributes"
        super().__init__(item, parent)
        ######################################################
        # TODO: Add any additional subtree properties here #
        ######################################################

    def _update(self):
        "Augment AVL update() to fix any properties calculated from children"
        super()._update()
        ##########################################################
        # TODO: Add any maintenence of subtree properties here #
        ##########################################################

    def add_transaction(self, x):
        "Add a transaction to the transaction log"
        super().insert(x)

    def interval_revenue(self, t1, t2):
        "Return total revenue of transactions in subtree occuring in [t1,t2]"
        if self.item is None:
            return None
        ######################
        # TODO: Implement me #
        ######################
        return 0
```