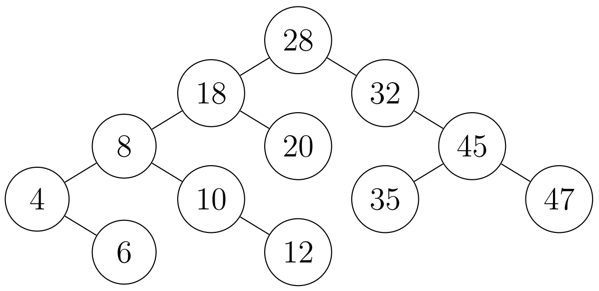
Problem Set 3

All parts are due on March 1, 2019 at 6PM. 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 3-1. [12 points] **Binary Tree Practice**

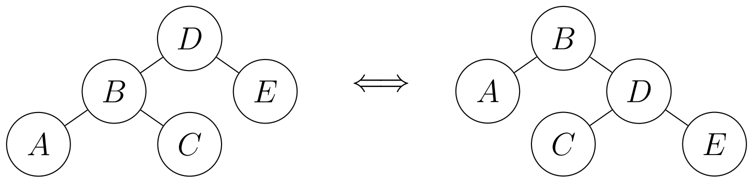
(a) [5 points] Perform the following operations in sequence on the binary search tree T below. Draw the modified tree after each operation. Note that this tree is a BST, not an AVL. You should not perform any rotations in part (a). For this problem, keys are items.

1. `insert(2)`
2. `delete(47)`
3. `delete(10)`
4. `insert(25)`
5. `delete(32)`



(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 3-2. [33 points] **Consulting**

Briefly describe a database to help each of the following clients, remembering to argue correctness and running time for each. In each problem, n represents the number of things stored in the database at the time of a database operation.

(a) [9 points] **Cilison Valley:** Hichard Rendricks has a start-up idea. While a good CEO wants to be surrounded by really smart advisors, the CEOs in Cilison Valley want the smartest group of advisors dumber than they are. That way, they will always be right! Smartness is evaluated by a positive integer IQ, where two people may have the same IQ. Hichard needs a database to keep track of potential advisors that will allow him to quickly find the k smartest advisors no smarter than a given Cilison Valley CEO. Design a database supporting the following operations:

- `add_advisor(q, s)`: add advisor with IQ q and name s in $O(\log n)$ time
- `get_advisors(k, q)`: return the names of k advisors having the highest IQs strictly less than q in $O(k + \log n)$ time (or return that no such k exist)

(b) [12 points] **The $\times$-People:** Professor-$\times$ is the leader of the $\times$-People, a team of super heroes. She is interested in finding the **power** of the $\times$-People at any point in time. Power is higher dimensional than can be expressed by any scalar value, so Professor-$\times$ represents the power of an individual using a 2×2 matrix of integers. Professor-$\times$ often observes: "We are the $\times$-People; together, our power multiplies!" Specifically, the power of the $\times$-People is the **product** of the powers of its members when ordered **increasing by age**. You may assume ages are given in integer seconds, and no two $\times$-People have the same age. Recall that 2×2 matrix multiplication can be computed in $O(1)$ time and is associative but **not commutative**, so given three 2×2 matricies A, B, and C, the statement $(A \times B) \times C = A \times (B \times C)$ is true but, generally, $A \times B \neq B \times A$. While individuals may periodically leave or join the $\times$-People, Professor-$\times$ wants to be able to very quickly query the power of the $\times$-People at any point in time. Describe a database supporting the following operations:

- `recruit(a,p)`: add a team member with age a and power p in $O(\log n)$ time
- `depart(a)`: remove a team member with age a in $O(\log n)$ time
- `get_power()`: return the power of the current $\times$-People in $O(1)$ time

(c) [12 points] **Perfect Placements:** Warty Malsh is the mayor of Stobon, a city built up along a single subway line having s stations identified by unique integers. Station identifiers increase from one end of the subway line to the other. Every house and office in Stobon is located near one of the subway stations. The address for each house is a tuple (i, a), where i is the identifier of the station near it, with a a unique integer. Each house also has a purchase price in integer cents (two houses may have the same price). Warty wants to make it easy for citizens to find the perfect place to live in the city. A house is ***i*-perfect** if it is a cheapest house, from among available houses that are the fewest subway stations away from station i. More formally, define

$H_k(i)$ to be the set of houses currently for sale at either station $i-k$ or station $i+k$. If there are houses available for purchase at station i, i.e., $|H_0(i)| > 0$, then an i-perfect house will be any cheapest house from $H_0(i)$. If there are no houses currently for sale at station i, i.e., $|H_0(i)| = 0$, then an i-perfect house will be a cheapest house from $H_k(i)$, for the smallest value of k for which $H_k(i)$ is nonempty. Describe a database supporting the following operations, each in $O(\log s + \log n)$ time:

- `list((i, a), p)`: put the house located at address (i, a) up for sale at price p
- `change_price((i, a), p)`: set the price of the house at address (i, a) to p
- `sell_perfect(i)`: remove any i-perfect house from the market, if one exists

Problem 3-3. [55 points] **Holiday Helper**

Throughout the year, Mrs. Klaus keeps track of all the kids in the world and whether they have been bad or good. At any point in time, the goodness $g \in \{$`'good'`,`'bad'`$\}$ of a kid may change based on the kid's behavior. Define the **imbalance** of a set of kids to be the number of good kids in the set minus the number of bad kids in the set. At any point in time, Mrs. Klaus wants to know the imbalance of all kids within a given age range. Each kid has a birth time represented by a (possibly negative) integer number of seconds after a reference time t (you may assume that each kid's birth time is unique). Given a time range (t_1, t_2), a kid is **within the range** if their birth time is between times t_1 and t_2 inclusive. In this problem, you will help Mrs. Klaus by designing a database supporting the following operations:

- `update_kid(b, g)`: change the goodness of the kid having birth time b to g (or add the kid to the database if birth time b is not already in the database)
- `range_imbalance(t1, t2)`: return the imbalance of all kids within range (t_1, t_2)

To solve this problem, we will represent each kid as an item `x` having a birth time key `x.key` and goodness property `x.g`, and then store the items in an augmented AVL tree, keyed by birth time. First, we augment each AVL node with three subtree properties:

- `node.imbalance`: the imbalance of all kids in `node`'s subtree
- `node.min_time`: the minimum birth time of any kid in `node`'s subtree
- `node.max_time`: the maximum birth time of any kid in `node`'s subtree

(a) [3 points] Assuming all descendents of node p are correctly augmented with these properties, show how to compute these properties for p in $O(1)$ time.

Let S be the subtree of node p from an AVL tree augmented as in (a). Given range (t_1, t_2), either:

- all kids in S are within the range (S **fills** the range),
- kids in S are either all strictly above or all strictly below the range (S **avoids** the range), or
- S neither fills nor avoids the range (S **divides** the range).

(b) [3 points] Given node p, show how to classify whether p's subtree fills, avoids, or divides range (t_1, t_2) in $O(1)$ time.

(c) [3 points] Given node p whose subtree S does not divide range (t_1, t_2), show how to compute the imbalance of kids from S who are within the range in $O(1)$ time.

(d) [9 points] A node in this augmented AVL tree **branches** on range (t_1, t_2) if it has a left child p and a right child q, and the subtrees rooted at p and q both divide the range. Prove that at most one node in the AVL tree branches on range (t_1, t_2).

(e) [12 points] Describe how to use this augmented AVL tree to implement a database that supports both requested operations, each in $O(\log n)$ time, where n is the number of kids stored in the AVL tree at the time of the operation.

(f) [25 points] Implement your database in the Python class `HolidayHelper` extending the `AVL` class provided. You should modify `AVL` `update()` and implement `range_imbalance(t1, t2)`; `update_kid(b, g)` has been implemented for you. You can download a code template containing some test cases from the website. Submit your code online at `alg.mit.edu`.

```python
class Kid:
    def __init__(self, b, g):
        self.key, self.g = b, g

class HolidayHelper(AVL):
    def __init__(self, item = None, parent = None):
        super().__init__(item, parent)
        self.imbalance = 0                              # augmentation
        self.min_time, self.max_time = None, None       # augmentation

    def update(self):
        super().update()
        ########################################################
        # TODO: (a) Add maintenence of subtree properties here #
        ########################################################

    def update_kid(self, b, g):
        "Change or update kid with birth time b and goodness g"
        try:
            kid = self.delete(b)     # remove existing kid
            kid.g = g                # update existing kid ('good' or 'bad')
        except:
            kid = Kid(b, g)          # make new kid
        super().insert(kid)          # insert kid

    def range_imbalance(self, t1, t2):
        "Return imbalance of kids in subtree within (t1, t2)"
        ######################
        # TODO: Implement me #
        ######################
        return 0
```