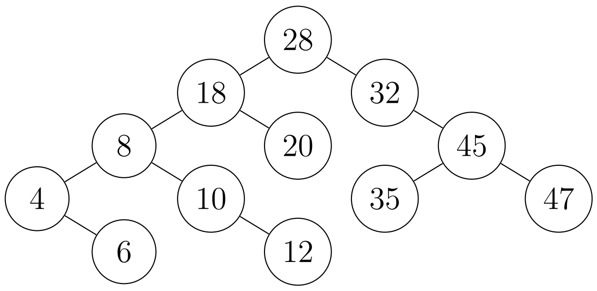
Problem Set 3

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)`



Solution:

```
insert(2)
            ______28___                           ______28___
    ________18___    32______    =>       ________18___    32______
____8___       20       ___45___      ____8___       20       ___45___
4__   10___             35    47    __4__   10___             35    47
  6      12                        (2)  6      12

delete(47)
              ______28___                           ______28___
      ________18___    32______    =>       ________18___    32______
  ____8___       20       ___45___      ____8___       20       ___45
__4__   10___             35   (47)   __4__   10___             35
2   6      12                         2   6      12

delete(10)
              ______28___                     ______28___
      ________18___    32______ =>       _____18___    32______
  ____8___       20       ___45      ____8___    20       ___45
__4__  (10)___            35       __4__   12             35
2   6      12                      2   6
```

```
insert(25)
           ______28___       =>            _________28___
      _____18___    32______          _____18___       32______
  ____8___    20       ___45      ____8___    20___       ___45
__4__   12             35       __4__   12      (25)      35
2   6                           2   6

delete(32)  // this was supposed to be delete 18 :(

           _________28___       =>            _________28______
      _____18___      (32)_____          _____18___       ___45
  ____8___    20___       ___45      ____8___    20___    35
__4__   12       25       35       __4__   12       25
2   6                              2   6
```

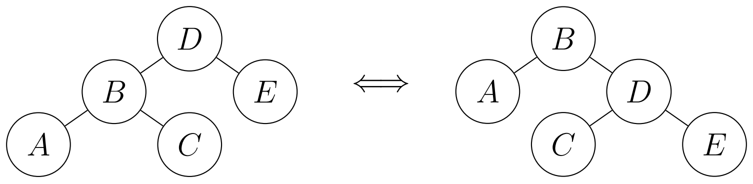
Rubric: 1 point for each correct operation

(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: Nodes containing keys 18 and 32 are not height balanced.

Rubric: 1 point for one correct node, 3 points for 2 correct nodes

(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:

```
rotate_right(18)
            ______28___                 ______________28___
    _______(18)__    32______    => ____8_________       32______
____8___       20       ___45___    4__   ______18___       ___45___
4__   10___             35    47      6   10___    20       35    47
  6      12                                  12

rotate_left(32)
    ______________28___          =>     ______________28_________
```

```
_____8___________       (32)______        _____8___________        _______45___
4__    _______18___         ____45___     4__    _______18___      32____    47
   6   10___      20        35     47        6   10___      20        35
           12                                        12
```

Rubric:

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

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)

Solution: Store advisors as (IQ, name) pairs in an AVL tree keyed by IQ. To implement `add_advisor(q, s)`, simply add pair (q, s) to the AVL tree in $O(\log n)$ time. To implement `get_advisors(k, q)`, start at the root and search for the advisor with highest IQ smaller than q in $O(\log n)$ time, and then perform a reverse in-order traversal by calling `tree_predecessor()` (analogous to `tree_successor()`) k times. By the analysis shown in R05, such an in-order traversal only takes $O(k+\log n)$ time, as desired. Correctness of these operations follow directly from the correctness of AVL tree operations.

Rubric:

- 3 points for overall description of data structure
- 2 points for description, correctness, and running time for `add_advisor`
- 4 points for description, correctness, and running time for `get_advisors`
- Partial credit may be awarded

(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

Solution: Store individuals in an AVL tree keyed by age, so node q stores an individual with age $q.age$ and power $q.power$. In addition, augment each AVL node q with the power $P(q)$ of the individuals in its subtree. The power of a subtree can be computed in $O(1)$ time from subtree powers of children: given node q, the power of the subtree rooted at q is $P(q) = P(q.left) \cdot q.power \cdot P(q.right)$, so this augmentation can be maintained during the AVL update phase without changing the running time of AVL tree operations. This augmentation is correct because the AVL tree stores the individuals sorted by age, and power multiplication is associative. To implement `recruit(a, p)`, simply insert (a, p) into the AVL tree in $O(\log n)$ time, maintaining the augmentation during update. To implement `depart(a)`, search the AVL tree for an individual with age a, and remove it if it exists in $O(\log n)$ time, again maintaining the augmentation. To implement `get_power()`, simply return the augmented subtree power stored at the root in $O(1)$ time. Correctness of these operations follow directly from the correctness of AVL tree operations.

Rubric:

- 3 points for overall description of data structure (augmentation)
- 3 points for description, correctness, and running time for `recruit`
- 3 points for description, correctness, and running time for `depart`
- 3 points for description, correctness, and running time for `get_power`
- Partial credit may be awarded

(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

Solution: We present two different solutions to this problem: one based on augmentation and one based on a cross-linked data structure.

(1) We need a data structure which allows us to first search among stations with housing availability, and then find the lowest price at a station. Store stations that contain available housing in an AVL tree keyed by station number. Link each station AVL node to the root of a local AVL tree associated with the station storing available house at that station keyed on address, where node q contains a house with address $q.a$ and price $q.p$. At each local AVL tree node, augment the node with the node $P(q)$ containing the lowest price house within its subtree. This augmentation can be computed directly in $O(1)$ time from the augmentation of children: specifically $P(q) = \arg\min_{a \in \{q, P(q.left), P(q.right)\}} a.p$, so it can be maintained during the AVL update phase without changing the running time of AVL tree operations.

To implement `list((i, a), p)`, find the station AVL node associated with station i in $O(\log n)$ time (n because we do not store empty stations). If it does not exist, this is the first house available at that station, so add a station AVL node associated with i linked to an empty local AVL tree, also in $O(\log n)$ time. Then, insert the house into station i's local AVL tree, which could take $O(\log n)$ time. To implement `change_price((i, a), p)`, search for station i in the station AVL tree in $O(\log n)$ time, and then remove the house at address a from station i's local AVL tree, change its price, and then reinsert the house in $O(\log n)$ time. Lastly, to implement `sell_perfect(i)`, we must first find the one or two closest stations nearest to i with available housing. If the station AVL tree is empty, there are no available houses, so return that no house exists. Otherwise, search for station i in the station AVL tree in $O(\log n)$ time. If station i exists, that will be our target station. Otherwise, find the smallest station number $i + k_+$ above i and largest station $i - k_-$ below i, each in $O(\log n)$ time via successor and predecessor. If $k_- \neq k_+$ our target station will be the smaller of the two, and if $k_- = k_+$, we will search both stations for an i-perfect house. Then for each of our target stations, simply follow the node in the augmentation stored at the root node of station i's local AVL tree, and remove the cheapest house found in $O(\log n)$ time. If removing a house makes a local AVL tree empty, remove that station's node from the station AVL tree in $O(\log n)$ time. Correctness of these operations follow directly from the correctness of AVL tree operations. Note that all running times are $O(\log n)$ which is also $O(\log s + \log n)$.

(2) We can also solve this problem without augmentation by cross-linking with another AVL tree. As in the previous solution, store and station AVL tree with each station's node pointing to a local AVL tree of available houses at that station, this time keyed by price (a min-heap could also be used). This data structure can directly support `list((i, a), p)` and `sell_perfect(i)` operations, but changing the price of an existing house requires us to find a house by address. If many houses at the same station have the same price, finding the house to change could take $O(n)$ time. In order to find the house we're looking for quickly, we can also maintain another lookup AVL tree of all the houses keyed by address (i, a), where each node also stores a pointer to the local AVL node from the other data structure containing that house. Then, to `change_price((i, a), p)`, search for that address in the lookup AVL tree in $O(\log n)$ time, follow the pointer to the house's node in the local AVL tree, remove the node from the tree, change it's price, and then reinsert in $O(\log n)$ time.

Rubric:

- 3 points for overall description of data structure (augmentation)
- 3 points for description, correctness, and running time for `list`
- 3 points for description, correctness, and running time for `change_price`
- 3 points for description, correctness, and running time for `sell_perfect`
- Partial credit may be awarded

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 descendants of node p are correctly augmented with these properties, show how to compute these properties for p in $O(1)$ time.

Solution: We compute these properties for p based on the properties of p's children. Then the imbalance of p's subtree is simply the sum of the imbalances of p's subtrees (zero if a subtree is empty) plus 1 if the kid at p is good, and minus 1 if the kid at p is bad. The minimum birth time in p's subtree is either the minimum birth time in p's left subtree, or the birth time at p if p has no left subtree. Similarly, the maximum birth time in p's subtree is either the maximum birth time in p's right subtree, or the birth time at p if p has no right subtree. Because the subtree properties of p's left and right children are accessible in constant time, computing each property takes $O(1)$ time.

Rubric: 1 point for each correct augmentation computation

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.

Solution: First, p's subtree fills the range exactly when $t_1 \leq$ `p.min_time` and `p.max_time` $\leq t_2$. p's subtree avoids the range exactly when $t_2 >$ `p.min_time` or `p.max_time` $< t_1$. Otherwise, p's subtree divides the range. Because p stores its min and max subtree times, we can classify the subtree in $O(1)$ time.

Rubric: 1 point for each correct classification

(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.

Solution: Because S does not divide the range, it either fills it or avoids it. If S avoids the range, then there are no kids from S within range, so its imbalance is 0. If S fills the range, then the imbalance of kids from S within range is simply the imbalance of the entire subtree, which is stored at p by our augmentation. In either case, we can return the requested result in $O(1)$ time.

Rubric:

- 3 points for a correct imbalance computation for the two cases
- (1 point if only one of the cases is correct)

(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).

Solution: First, any node that branches on a range must contain a birth time that is within the range; otherwise the range would not be continuous. Suppose for contradiction that two distinct nodes p and q branch on range (t_1, t_2), and without loss of generality, assume p appears before q in the in-order traversal. Both p and q are within the range, and since the range is continuous, p's right subtree is fills the range. But if p branches on the range, p's right subtree contains a birth time outside the range, a contradiction.

Rubric:

- 9 points for a correct proof (there are many possible approaches)
- Partial credit may be awarded

(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.

Solution: We maintain the augmented AVL tree described above, modifying the update method to maintain the augmentation as in part (a). To implement `update(b, g)`, remove any kid with birth time b from the tree. If it exists, set its goodness to g and then re-insert into the tree. If it does not exist, create a new kid from the input and insert it into the tree. AVL tree deletion and insertion each take $O(\log n)$ time, so so does this operation. Then to implement `range_imbalance(t1, t2)`, compute the imbalance of kids in the range recursively starting at the root. Given node p, define $I(p, t_1, t_2)$ to be the imbalance of the kids from p's subtree within the range (t_1, t_2). We compute $I(p, t_1, t_2)$ by first classifying it according to (b) in constant time. If p's subtree fills or avoids the range, we compute the answer in constant time based on part (c). Otherwise, we can compute $I(p, t_1, t_2)$ recursively as the sum of $I(p.left, t_1, t_2)$ and $I(p.right, t_1, t_2)$, plus 1 if the kid at p is good or minus 1 if the kid at p is bad. By part (d), at most one node in the tree will make a recursive call that will take more than constant time to evaluate. Thus when computing $I(r, t_1, t_2)$ from root r, recursive calls walk down the tree, possibly branching once, while doing constant work at each visited node. Since an AVL tree has height $O(\log n)$, this procedure takes $O(\log n)$ time.

Rubric:

- 3 points for overall description of data structure (augmentation)
- 3 points for description, correctness, and running time for `update_kid`
- 6 points for description, correctness, and running time for `range_imbalance`
- Partial credit may be awarded

(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`.

Solution:

```python
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 node_imbalance(self):
        return 1 if self.item.g == 'good' else -1

    def update(self):
        super().update()
        self.imbalance = self.node_imbalance()
        self.min_time = self.item.key
        self.max_time = self.item.key
        if self.left:
            self.imbalance += self.left.imbalance
            self.min_time = self.left.min_time
        if self.right:
            self.imbalance += self.right.imbalance
            self.max_time = self.right.max_time

    def update_kid(self, b, g):
        try:
            kid = self.delete(b)      # remove existing kid
            kid.g = g                 # update existing kid
        except:
            kid = Kid(b, g)           # make new kid
        super().insert(kid)           # insert kid

    def range_imbalance(self, t1, t2):
        if (t1 <= self.min_time) and (self.max_time <= t2): # fills
            return self.imbalance
        if (t2 < self.min_time) or (self.max_time < t1):    # avoids
            return 0
        imbalance = 0                                       # divides
        if t1 <= self.item.key <= t2:
            imbalance += self.node_imbalance()
        if self.left:
            imbalance += self.left.range_imbalance(t1, t2)
        if self.right:
            imbalance += self.right.range_imbalance(t1, t2)
        return imbalance
```