Problem Set 4

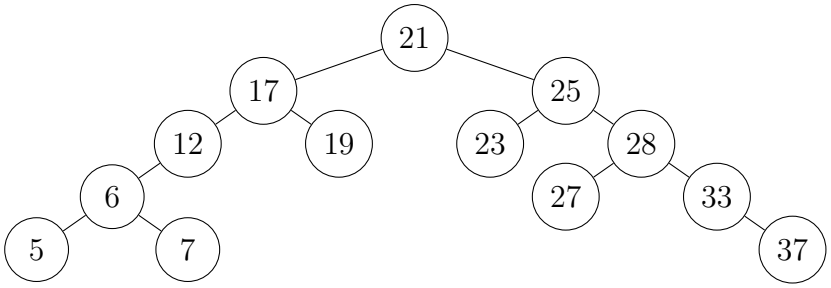
All parts are due on March 18, 2020 at 10PM. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. If you give an algorithm or a data structure, for full credit you must justify its runtime and correctness unless the problem says otherwise. Solutions should be submitted on Gradescope, and any code should be submitted for automated checking on `alg.mit.edu`.

Problem 4-2. [20 points] **Individual problem 1**

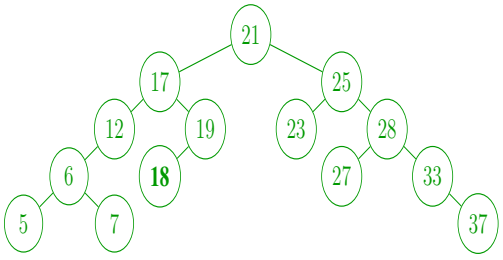
(1) BST Modifications [8 points]

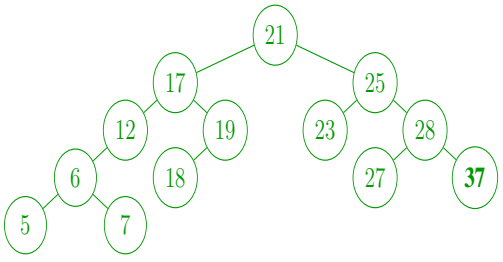
Perform the following operations (without any rotations) in sequence on the binary search tree T below. Draw the modified tree after each operation.

1. `insert(18)`
2. `delete(33)`
3. `insert(26)`
4. `delete(17)`

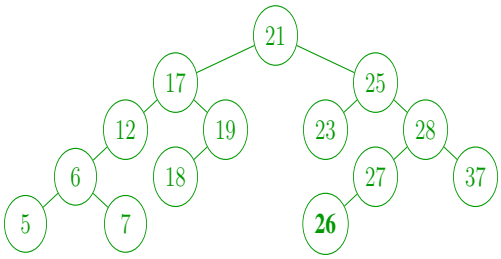


Solution:

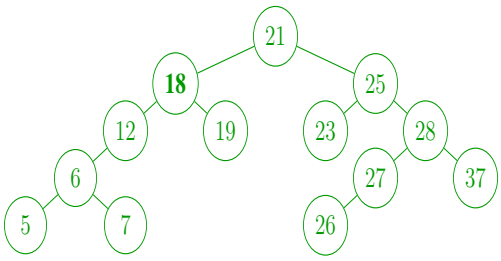
1.


2.


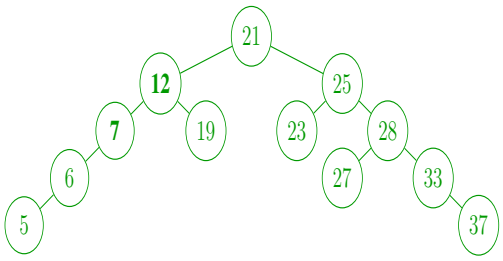
3.



4.



OR



Rubric:

- 2 points for each operation

(2) [3 points] In the original tree T (prior to any operations), list the keys from nodes that are **not height balanced**, i.e., the heights of thenode's left and right subtrees differ by more than one and do not satisfy the AVL Property.

Solution: 12, 17, 25

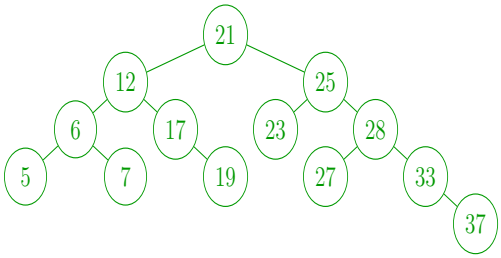
Rubric:

- 1 point for each node

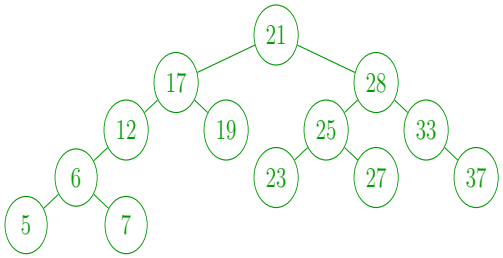
(3) [4 points] Perform a sequence of two rotations to make the original tree T height balanced, to satisfy the AVL Property. To indicate your rotations, draw the tree after each rotation.

Solution:

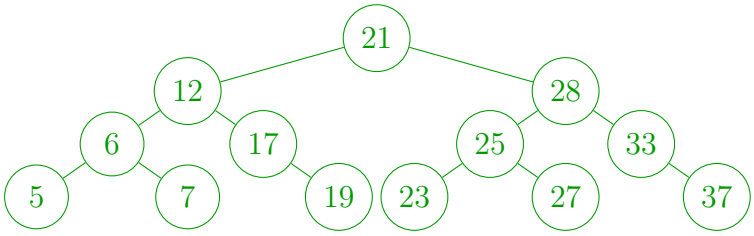
1.



OR



2.



Rubric:

- 2 points for each rotation

(4) [8 points] Given an AVL tree T, containing n distinct integer, someone accidentally deleted the value of a key stored at one of the nodes. Assume you're given a reference to the node with the unknown key, describe a **worst-case** $O(\log n)$ algorithm to calculate the number of possible replacement values you can use that still satisfy the AVL and BST properties. You should make sure that all keys are still distinct.

Solution: We note that the height requirements are already satisfied since the structure didn't change. The number of possible keys that would satisfy the BST property would be just between the successor and predecessor of the node in question. Return Successor(node) $-$ Predecessor(node) $-$ 1. If a successor or predecessor doesn't exist, we return $+\infty$. Each operation runs in $O(\log n)$ time.

Problem 4-3. [20 points] **Individual problem 2**

Mr.Vdelson-Aelsky is not happy with the fact that in a binary search tree tree only nodes with no children get a name related to real-life trees. As a garden enthusiast, he refers to nodes with 1 children as "twig nodes" and nodes with 2 children as "branch nodes". Now he wants to visualize a binary search tree by drawing twigs, branches and leaves. In order to do that, he needs to keep some important information about the structure of the tree. In particular, for every node, he wants to know about the number of each type of nodes (leaves, trigs, and branches) in the subtree rooted at this node.

(1) [3 points] What kind of augmentations would be helpful here?

Solution: We augment every node with 3 quantities: number of nodes with no children (leaves) in the subtree, number of nodes with 1 child (twigs) in the subtree, and the number of nodes with 2 children (branches) in the subtree.

Rubric:

- 1 points for each augmentation

(2) [17 points] Mr. Vdelson-Aelsky would also like to visualize dynamic operations on an AVL tree. Describe how to initialize and update the augmentations in $O(\log n)$ time after every `insert` and `delete`. Be sure to cover all the cases.

Solution:

`insert`: First, augment the new node with: 1 leaf, 0 twigs, 0 branches. If the new node's parent has 1 child after the insertion, subtract 1 from the number of leaves and add 1 to the number of twigs for every ancestor of the new node. Similarly, if the new node's parent has 2 children after the insertion, subtract 1 from the number of twigs and add 1 to the number of branches for every ancestor of the new node.

`delete`: When we delete a non-leaf node, we first recursively swap it down the tree until it becomes a leaf, and then we remove the leaf. Therefore, we only need to consider the case for removing leaves. After removing a leaf, first subtract 1 from the

number of leaves for every ancestor of the removed node. If this parent of the removed node has 1 child after the removal, subtract 1 from the number of branches and add 1 to the number of twigs for every ancestor of the removed node. If this parent of the removed node has no children after the removal, subtract 1 from the number of twigs and add 1 to the number of leaves for every ancestor of the removed node.

Since we do a constant number of updates for every ancestor of the inserted/removed node and the height of an AVL tree is $O(\log n)$, the total time complexity of updating the augmentations in the tree is $O(\log n)$.

Rubric:

- 3 points for initial augmentation of a leaf
- 5 points for updating augmentations after `insert`
- 6 points for updating augmentations after `delete`
- 3 points for justifying the time complexity
- Partial credit may be awarded

Problem 4-4. [30 points] **Jen & Berry**

Jen & Berry need your help with their ice-cream. As a professional ice-cream taste tester [1] you're organizing an *ice-cream tournament* to find out which is the best ice-cream of all time. Each ice-cream in the tournament has a positive integer ID i. The tournament works by hosting a series of matches between two ice-creams, each match is identified with some positive match ID m. People online vote for their favorite ice-cream and each ice-cream earn some p points.

We define the **performance** of an ice-cream as the average number of points made in the matches they have been in.

Describe a database supporting the following operations, each in **worst-case** $O(\log n)$ time, where n is the number of matches in the database at the time of the operation. Assume that n is always larger than the number of ice-creams.

`record(i, m, p)`	record p points for ice-cream i in match m.
`clear(i, m)`	remove any record that ice-cream i in match m.
`ranked_winner(k)`	return the ice-cream ID with the k^{th} highest performance.

Solution: First, we observe that operations require finding and modifing records for an ice-cream given an ice-cream ID in $O(\log n)$ time, so we maintain a set data-structure containing the ice-creams keyed by unique ice-cream ID. Since we need to acheive worst-case $O(\log n)$ running time, we cannot afford the average case performance of a hash table, so we implement the dictionary using an AVL tree. We call this AVL tree I.

For each ice-cream, we will need to find and update its matches by match ID, so for each ice-cream in I, we will maintain a pointer to their own Set AVL tree containing that ice-cream's

[1] `https://youtu.be/SLP9mbCuhJc`

matches keyed by match ID (call this an ice-cream's match dictionary). With each ice-cream's match dictionary, we will maintain the number of matches they've been in and the total number of points they've scored to date. We can compare the performance of two ice-creams from their respective number of matches and points via cross multiplication. Note that here we can't perform integer division, since we want to work with arbitrary precision.

Lastly, to find the k^{th} highest performing ice-cream, we maintain a separate Set AVL tree on the ice-creams keyed by performance, augmenting each node with the size of its subtree (call this the performance tree). We showed in class how to maintain subtree size in $O(1)$ time, so we can maintain this augmentation. Each node of I will store a cross-linking pointer to the node in the performance tree corresponding to that ice-cream. Since we use Set AVL trees for all data structures, Set operations run in worst-case $O(\log n)$ time.

To implement `record(i, m, p)`, find ice-cream i's match dictionary D in I in $O(\log n)$ time. If match m is in D, update its stored points to p in $O(\log n)$ time and update the total number of points stored with i's match dictionary in $O(1)$ time. Otherwise, insert the record of match m into D in $O(\log n)$ time, and update the number of matches and total points stored in $O(1)$ time. The performance of i may have changed, so find the node corresponding to i in the performance tree, remove the ice-cream's performance from the tree, update its performance, and then reinsert into the tree all in $O(\log n)$ time. This operation maintains the semantics of our data structures in worst-case $O(\log n)$ time.

To implement `clear(i, m)`, find ice-cream i's match dictionary D in I as before and remove m (assuming it exists). Identically to above, maintain the stored number of matches and total points points, and update the performance tree together in worst-case $O(\log n)$ time.

To implement `ranked_winner(k)`, find the k^{nth} highest performance in the performance tree by using the subtree size augmentation: if the size of a node's right subtree is k or larger, recursively find in the right subtree; if the size of the node's right subtree is $k-1$, then return the ice-cream stored at the current node; otherwise the size of the node's right subtree is $k' < k-1$, recurse in the node's left subtree to find its subtree's k'^{th} highest performing ice-cream. This recursive algorithm only walks down the tree, so it runs in worst-case $O(\log n)$ time.

Problem 4-5. [30 points] **Blue's Anatomy**

Geredith is an intern working at Geattle Srace Hospital, which is a pretty old hospital. Geredith's boss, the chief of surgery, would like to update the systems to make the hospital run more efficiently and assigns Geredith to the task. When patients are admitted to the emergency room, they are assigned a priority number between $[1, 100]$, which can be a float, indicating how quickly a doctor needs to come help them, with 1 being the most urgent. The chief wants Geredith to come up with a solution to keep track of the patients based on who needs to be treated first. Geredith hasn't taken an algorithms class, so she turns to you for help. In this problem, you'll design a data structure that keeps track of n patients such that querying for patients based on their priorities is efficient.

Throughout this problem, you will design a data structure which we will call `PatientDatabase` that will help Geredith compute information about the patients efficiently. **You may assume all**

priorities and patients' IDs are unique. All runtimes for this problem should be worst-case runtimes.

After each patient is admitted they not only receive a number indicating their priority but are also tagged with **one** specialty which they should be treated for. For example, a patient could come in with a priority of 45.8 and a need for cardio help, and another patient could come in with a priority of 32.98 and a need for pediatric help. Assume that Geattle Srace Hospital only has S specialties. We will treat these specialties as integers between 0 and $S-1$ in order to more easily work with them in our computation structure.

(1) [15 points] Help Geredith come up with a way to design a data structure that can support the following operations in the given times:

- `build(patients)`: build a data structure from the given list of n patients in $O(S + n \log n)$ time
- `insert(patient_ID,priority,tag)`: insert the patient with the given ID, priority, and tag into the data structure in $O(\log n)$ time
- `delete(patient_ID)`: delete the patient with the given ID in $O(\log n)$ time
- `find(patient_ID)`: find the patient's priority and tag with the given ID $O(\log n)$ time
- `find_highest_priority(tag)`: find the patient's ID with the highest priority (most urgent) that also has the given tag in $O(\log n)$ time
- `find_overall_highest_priority()`: find the patient's ID with the overall highest priority (most urgent) in $O(\log n)$ time
- `update(patient_ID, new_priority, new_tag)`: Update the given patient with the given priority to have the new input tag in $O(\log n)$ time

Solution:

- `build(patients)`: Keep an AVL tree for each of the specializations sorted by priority and augmented by name, a main AVL tree that is sorted by priority and augmented with the tag and patient ID, and a patient ID AVL tree that is sorted by patients' IDs that is augmented with the priority and tag of the patient. Keep a direct access array of size S with pointers to each of the S trees.
- `insert(patient_ID, priority, tag)`: To insert a new patient, insert the patient into the main tree, the tree that corresponds to the patient's tag, and the IDs AVL tree.
- `delete(patient_ID)`: To delete, first find the patient with the given ID in the IDs tree, and keep track of the priority and tag. Then, delete the patient from the IDs AVL tree, the specific specialty AVL tree using the tag, and the main AVL tree using the priority.
- `find(patient_ID)`: Find the patient in the IDs AVL tree and return the priority and tag associated with the node.

- `find_highest_priority(tag)`: Look in the tree corresponding to the given tag and find the item with highest priority in that tree and return the patient's name.
- `find_overall_highest_priority()`: Look in the main tree and return the ID in the node with the highest priority.
- `update(patient_ID, new_priority, new_tag)`: First find the patient in the names AVL tree and record the old priority and tag before updating the augmentations with the new priority and tag. Then, go to the AVL tree of the old specialty, delete the patient from there, and insert the patient into the new specialty's tree using the new priority and augmented with the patient's name. Finally, go into the main AVL tree, delete the patient with the old priority, and reinsert the patient with the new priority, augmented with the name and new tag.

Correctness: Clear from the algorithmic description and the correctness of AVL trees.

Time analysis: To build, we create $S + 2$ trees in $O(n \log n)$ time each. However, we know that each patient is in at most the main tree, one other specialty tree, and the names tree, so building all $S + 2$ trees will take $O(n \log n)$ time. There is an additional $O(S)$ cost to build the direct access array. Each insertion and deletion requires a lookup in three different AVL trees. We can find the AVL tree for a specific specialization in $O(1)$ time using a direct access array of size S such that index i points to the AVL for specialization i. To do either of the finds, we just have to do a find or get the max in the appropriate AVL tree. To update, we do a constant number of insertions and deletions in different AVL trees for $O(\log n)$ time. Thus, all operations but `build` take the same time as an AVL tree.

Geredith's boss is so impressed with Geredith that he wants her to help with scheduling the surgeons as well. The head of neurosurgery, Serek, has too many patients and would like an efficient way to query for the number of patients that are within a certain priority interval, to be able to effectively prioritize his time.

(2) [8 points] Serek asks for Geredith's help to come up with a data structure that can support the operation `num_patients(`p_1`,` p_2`)` where p_1 and p_2 are two priorities on **only the neurology cases**. (You do not need to worry about tags at all for this section because you can assume all patients have the same tag.) The function `num_patients(`p_1`,` p_2`,)` should determine how many patients in the data structure have a priority between p_1 and p_2. Explain how Geredith can create a data structure to implement the following operations, each in $O(\log n)$ time.

- `insert(patient_ID,priority)`: insert the patient with the given priority and tag into the data structure
- `delete(priority)`: delete the patient with the given priority
- `find(priority)`: find the patient with the given priority

- `num_patients`(p_1, p_2): Find the number of patients with priority p such that $p_1 \leq p \leq p_2$

For this part only, we do not need to be able to look up a patient by name. You data structure should also be able to be built in $O(n \log n)$ time.

Hint: You may want to consider augmenting each item with the minimum and maximum value in its subtree. Consider what happens if the given priority range includes the entire left subtree, the entire right subtree, or only part of those subtrees? Using this information, can you come up with a recursive algorithm?

Solution: Keep an AVL tree on the priority values. Insert/delete/find operations will act as normal. We also add augmentations to keep at a given `item`, `item.min` (minimum value in the item's subtree), `item.max` (maximum value in the item's subtree), and `item.size` (size of the item's subtree). To update these values based on an item's children, we have `item.min=max(item.value,item.left.min)`, `item.max=min(item,item.right.max)`, and `item.size=item.left.size+item.right.size+1`.

For `num_patients`(p_1, p_2), we implement a recursive algorithm. If $p_1 \leq$ `item.min` and `item.max` $\leq p_2$, return `item.size`. Otherwise, we keep a running total, which we call `total`. If the root is in the given range, add 1 to `total`. If the left subtree is entirely contained in the range ($p_1 \leq$ `item.left.min` $\leq$ `item.left.max` $\leq p_2$), add `item.left.size` to the total. Otherwise if the range of the left subtree contains part of the range (`item.left.min` $< p_1 \leq$ `item.left.max or item.left.min` $\leq p_2 <$`item.left.max`), recurse on the left subtree. If the right subtree is entirely contained in the range ($p_1 \leq$ `item.right.min` $\leq$ `item.right.max` $\leq p_2$), add `item.right.size` to the total. Otherwise if the range of the right subtree contains part of the range (`item.right.min` $< p_1 \leq$ `item.right.max or item.right.min` $\leq p_2 <$ `item.right.max`), recurse on the right subtree.

Correctness: Correctness of the basic AVL operations are clear from AVL correctness/explanation of our augmentations. For `num_patients`, we consider the base case when the entire tree is contained in the range. In this case, all patients in the subtree are in the range so we return the correct value. Otherwise, we figure out how many of the items in the left and right subtree are in the range via recursion and add 1 if the root is also in the range, resulting in the correct total.

Runtime: It takes constant time to update the augmentations of a given node based on its children, so as seen in class and recitation and by correctness of AVL trees we get $O(\log n)$ time for insert/delete/find and $O(n \log n)$ for build.

To analyze `num_patients`, we have to consider how often we recurse on both children of the root. We claim we only recurse on both children of a node for at most one node. Consider a node `node1` that we recurse on both the left and right children of. Say we reach a descendant of `node1` (in its left subtree) that we must recurse on both sides of. Call this `node2`. There must be an element in `node2`'s right subtree that is larger than p_2. However, this node is still in `node1`'s left subtree, and we know that

no element in `node1`'s left subtree is larger than p_2 (or else no item in `node1`'s right subtree would be in the range). Thus, this is impossible. The argument for the right subtree is symmetric.

During the recursion, we make a function call on exactly one node at each level until we reach a node where we recurse on both sides (if we ever do so). After this point, we make a function call on two nodes at each level until we reach the leaves (since we never split again). There is constant work per call and there are $O(\log n)$ calls for $O(\log n)$ overall time for this function.

(3) [10 points] Code the data structure from the previous part using the outline provided on `alg.mit.edu`. The portion of the outline given below shows all functions that you must implement (denoted by TODO).

```
1  ####################################
2  ##### YOUR CODE IN THIS SECTION
3  ####################################
4
5  # do not modify this class
6  class Patient:
7      def __init__(self, name, priority):
8          self.key, self.name =  priority, name
9
10
11 class PatientDatabase(AVL):
12     def __init__(self, item = None, parent = None):
13         # do not change this line
14         super().__init__(item, parent)
15         ########################################################
16         # TODO: initialize your augmentations #
17         ########################################################
18         raise NotImplementedError
19
20     def update(self):
21         "updates the attributes of the current element"
22         # do not change this line
23         super().update()
24         ########################################################
25         # TODO: (a) Add maintenence of subtree properties here #
26         ########################################################
27         raise NotImplementedError
28
29
30     # do not modify this function
31     def update_patient(self, name,priority):
32         "Change or update kid with name name and priority priority"
33         try:
34             patient = self.delete(priority)     # remove existing kid
35             patient.tags = tags              # update existing kid ('good' or 'bad')
36         except:
```

```
            patient = Patient(priority, tags)          # make new kid
        super().insert(patient)          # insert kid

    def num_patients(self,p1,p2):
        "return number of patients that have priority between p1 and p2"
        ###########################################################
        # TODO: implement the num_patients function here #
        ###########################################################
        raise NotImplementedError



    def item_str(self):
        return str(self.item.key) + str(self.item.tags) + str(self.subtree_tags)
```

Solution: See `alg.mit.edu` or the following solution.

```
####################################
##### YOUR CODE IN THIS SECTION
####################################

# do not modify this class
class Patient:
    def __init__(self, name, priority):
        self.key, self.name =  priority, name


class PatientDatabase(AVL):
    def __init__(self, item = None, parent = None):
        super().__init__(item, parent)
        self.size = None   # augmentation
        self.min_val, self.max_val = None, None     # augmentation

    def update(self):
        "updates the attributes of the current element"
        # do not change this line
        super().update()
        self.min_val = self.item.key
        self.max_val = self.item.key
        self.size = 1
        if self.left:
            self.size += self.left.size
            self.min_val = self.left.min_val
        if self.right:
            self.size += self.right.size
            self.max_val = self.right.max_val


    # do not modify this function
    def update_patient(self, name,priority):
```

```
34          "Change or update kid with name name and priority priority"
35          try:
36              patient = self.delete(priority)     # remove existing patient
37              patient.name = name              # update existing patient
38          except:
39              patient = Patient(name, priority)          # make new patient
40          super().insert(patient)          # insert patient
41
42
43      def num_patients(self,p1,p2):
44          "return number of patients that have priority between p1 and p2"
45          total = 0
46          # check for base case
47          if self.min_val >= p1 and self.max_val <= p2:
48              return self.size
49          if self.item.key >=p1 and self.item.key <= p2:
50              total += 1
51          # we can just always call on left/right side if part of it is in the range
52          # note that this doesn't change the runtime because the next call will be a ba
53          if self.left and self.left.max_val >= p1:
54                  total += self.left.num_patients(p1,p2)
55          if self.right and self.right.min_val <= p2:
56                  total += self.right.num_patients(p1,p2)
57          return total
58
59
60
61      def item_str(self):
62          return str(self.item.key) + str(self.item.name)
```