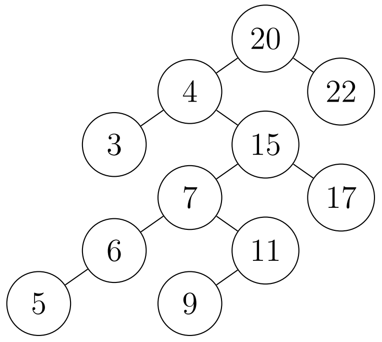
Problem Set 3

All parts are due on March 2, 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 `py.mit.edu/6.006`.

Problem 3-1. [20 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.

1. insert key 2
2. delete key 5
3. insert key 13
4. delete key 6
5. delete key 7



Solution:

```
1)                               2)
     ___________________20___         _________________20___
   __4______________       22  =>   __4____________       22  =>
 __3       _______15___           __3     _______15___
(2)      __7_____    17           2     __7_____    17
       __6   __11                      |6|  __11
       5     9                              9
3)                               4)
    ____________________20___         __________________20___
  __4_______________       22  =>   __4_____________       22  =>
__3     __________15___           __3   __________15___
2     __7_____       17           2    |7|____       17
      6   __11___                         __11___
          9   (13)                        9    13
4)
     ________________20___
   __4___________       22
 __3     ______15___
 2     _|11|__    17
       9    13
```

Rubric:

- 1 point for each operation

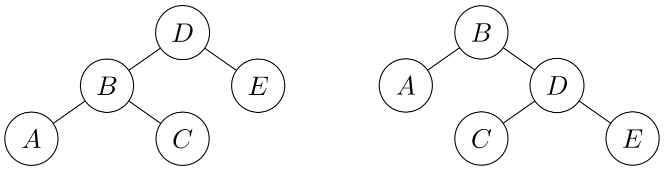
(b) [3 points] In the original tree T, list keys from nodes that are **not height balanced**, i.e. the heights of the node's left and right sub-trees differ by more than one.

Solution: Nodes that are not height balanced are 20, 4, and 15: sub-tree height differences are 4, 3, and 2 respectively.

Rubric:

- 1 point per correct node.

(c) [5 points] A **rotation** rearranges five nodes of a binary search tree between two states, maintaining the binary search tree property as shown below. Perform a sequence of at most four rotations to make the original tree T height balanced. To indicate your rotations, draw the tree after each rotation, circling or listing the keys of the five nodes participating in the rotation.



Solution:

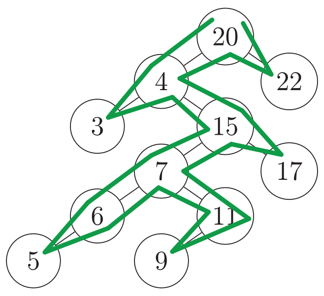
```
Rotate (6,7,11,15,17)
   ___________________20___        ___________________20___
 __4______________       22  =>  __4______               22
 3       ______(15)__            3     _(7)______
       _(7)____   (17)               _(6)    __(15)__
     _(6)  _(11)                     5     _(11)  (17)
     5     9                               9

Rotate (3,4,6,7,15)
   ___________________20___              _____________20___
 _(4)_____               22  =>    _____(7)_______       22
(3)    _(7)______                _(4)___     __(15)__
     _(6)    __(15)__           (3)  _(6)  __11    17
     5     __11    17                5     9
           9

Rotate (4,7,15,20,22)
         ____________(20)_         _____(7)_____________
   _____(7)_______      (22) =>  _(4)___        _____(20)__
 _(4)___     __(15)__            3   __6     __(15)__   (22)
 3   __6   __11    17                5     __11    17
     5     9                               9

Rotate (11,15,17,20,22)
   ______7______________           ______7________
 __4____        _____(20)__  =>  __4____     __(15)_____
 3   __6     __(15)__   (22)     3   __6   _(11)   __(20)__
     5     _(11)  (17)               5     9      (17)  (22)
           9
```

Rubric:

- 1 point for each rotation
- 1 point for all circled and/or labeled nodes

(d) [7 points] Starting at the root, find the minimum key with `min`, then repeatedly find the next key using `successor`. This process will touch every key in the tree via an **in-order traversal**. Draw the path of nodes touched by an in-order traversal on tree T. Because `min` and `successor` each take time proportional to the height h of a tree containing n keys, one can naively calculate a $O(nh)$ upper bound for in-order traversal. Show that in fact, in-order traversal requires at most $O(n)$ time.

Solution:



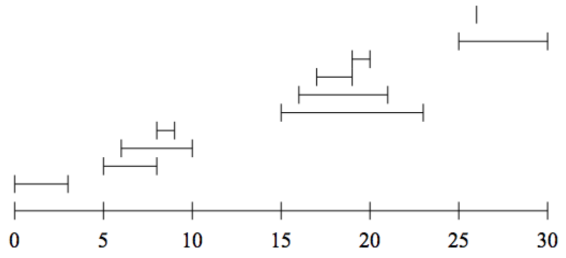
An in-order traversal walks around the tree touching each edge at most twice. Because a tree has one fewer edge than the number of nodes and the in-order traversal sequence will cross each edge at most twice, the traversal will be linear in the number of nodes (keys) in the tree.

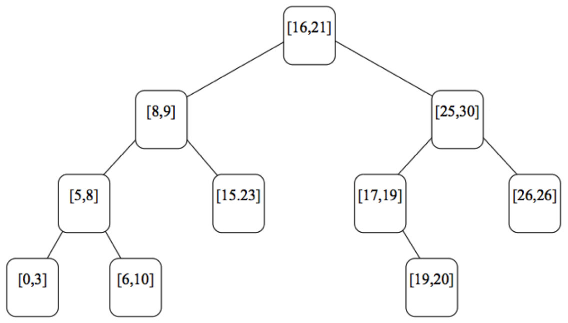
Rubric:

- 2 points for drawing of traversal
- 5 points for a $O(n)$ argument
- Partial credit may be awarded

Problem 3-2. [20 points] **Interval Search**

We would like to maintain a dynamic set of intervals that supports a query operation: a queried number q should return an interval $[a, b]$ from that set that **contains** q (i.e. $a \leq q \leq b$), or None if none of the intervals contains q. To do this, we will use an augmented BST structure to store these intervals. Each BST node will store an interval, sorting according to the left endpoint of each interval stored. Such a tree is called an **interval tree**. An example of an interval tree and its corresponding intervals are both shown below. You should verify that the interval tree is a BST.

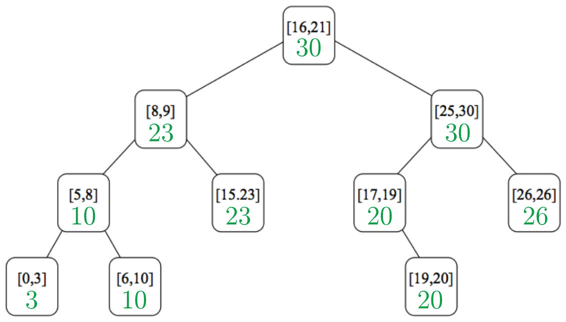




Unfortunately, using the normal BST `find` operation on a query point does not solve our problem. Searching for $q = 22$ would search to the right of the root node $[16, 21]$, while the only interval containing 22 is to the left. To aid you in finding a correct algorithm, we will augment the tree with additional information. For each node x containing interval $[a, b]$ and having children $x.left$ and $x.right$, compute and store the attribute $M(x) = \max(b, M(x.left), M(x.right))$.

(a) [10 points] For the interval tree in the figure above, compute the value of $M(x)$ for each node x in the tree. Argue that you can augment any interval tree with these values in linear time.

Solution:



By computing the value of $M(x)$ in level order (from the lowest level to the root), computing the maximum using pre-computed lower values of $M(x)$ is simply a comparison between three values, which can be done in constant time per node, leading to linear time overall. You can think of this approach as computing $M(x)$ via a bottom-up dynamic program, as we will talk about later in the term.

Rubric:

- $\lfloor k/2 \rfloor$ points for k correct values of $M(x)$
- 5 points for argument of linear time computation
- Partial credit may be awarded

(b) [10 points] Using $M(x)$, describe and give pseudocode for an algorithm `query(x,q)`, that finds an interval from the interval tree rooted at x that contains q (or None), and runs in $O(h)$ time, where h is the height of the interval tree. Prove that your algorithm is correct.

Solution:

1. We first check if q is contained in interval stored in node x – if it is we are done.
2. Next, we check if q is on left side of the left endpoint of interval in node x - if it is we recursively search in the left subtree. We argue that q cannot be contained in any interval in the right subtree because the BST is keyed using the left endpoint of the interval, and the tree satisfies the BST property.
3. If q is to the right side of the left endpoint of the interval in x, we have two cases:
   (a) If $max(left(x)) \geq q$ there must be a segment in the left subtree containing q and we recurse left. This is because the max value represents the rightmost point of *some* interval in the left subtree, and the left endpoints of all nodes in the subtree are all less than q thanks to the BST property. Therefore, there is an interval corresponding to some node in the left subtree that contains q.
   (b) If $max(left(x)) < q$ there is no segment in the left subtree containing q and we recurse right. All the intervals in the left subtree are such that the left and right endpoints are less than q. We do not know if the right subtree has an interval that contains q, but we know that the left subtree does not.

```
query(x, q):
    if x is None:
       return None
    if q contained in [leftendpoint(x), rightendpoint(x)]:
       return x
    if q < leftendpoint(x):
       return query(left(x), q)
    else
       if left(x) and M(left(x)) >= q:
          return query(left(x), q)
       else:
          return query(right(x), q)
```

Rubric:

- 10 points for a correct algorithm
- Partial credit may be awarded

Problem 3-3. [20 points] **Consulting**

Briefly describe a database for each of the following clients. Describe any operations that your database should support, and describe how your database supports them. Single operations should run in at most logarithmic time relative the number of items stored in the database (e.g. starting from an empty database, you may use logarithmic time each time you add an item).

(a) [10 points] **PriceTree:** Silliam Whatner runs a discount travel website, which maintains an inventory of available tour packages. Silliam often adds new tour packages to the site. Each package includes an itinerary, a price per person per day, and a duration measured in an integer number of nights. A client tells Silliam a desired tour length and budget per person per day, and Silliam will show them the five most expensive tour packages matching the customer's desired duration that are also within the customer's budget. If a customer decides to purchase a package, Silliam removes the package from the available inventory. Help Silliam design a database to organize the tour package information, and efficiently provide responses to customer requests.

Solution: Tell Silliam to store his tour package information in a balanced binary search tree, e.g. AVL Tree, sorted first by duration, then by price. The database must support insert, delete, and query operations. Inserting and deleting tours can be performed by normal AVL `insert` and `delete` using a duration-price tuple as a key in the tree, along with any other relevant tour information. To respond to a query, search for a key having matching duration, but price less than or equal to the customer's budget in $O(\log n)$ time. This can be done using normal BST `find` to a leaf node, which will either contain a package at or just below budget (or just above budget). Once such a leaf node is identified, call a `predecessor` (analogous to `successor`) operation four (or five) times to find the relevant cheaper packages for return. If a shorter duration tour is reached before identifying five packages (or if the left-most node is reached), then return however many were found. Since `predecessor` also runs in $O(\log n)$ time and is run only a constant number of times, the query operation takes $O(\log n)$ time.

Rubric:

- 1 point for mentioning a balanced BST
- 2 point for sorting on duration-price tuples
- 2 points for insert and delete
- 3 points for a correct algorithm for query
- 2 points for running times
- Partial credit may be awarded

(b) [10 points] **Leywand Corporation:** Ren Lipley is the leader of a group of space explorers colonizing a new planet. Their base is at the center of a large, perfectly-circular island of hospitable terrain, surrounded by an ocean of toxic liquid. Living on the coastline are thousands of alien creatures with strange movement patterns. While they can't seem to walk, they can instantaneously teleport to other locations on the coast, sometimes multiple times per day. Each day, space biologist teams journey to the coast to study and tag coastal aliens. Throughout the day, each team may ask the base to identify the twenty closest tagged aliens to their location. The base is equipped with a scanner that can detect and log the start and end locations of any tagged coastal alien when it teleports. Describe a database for the base to keep track of the tagged coastal aliens, and to support the research teams.

Solution: Have the base store the n locations of aliens in a balanced binary search tree, e.g. AVL Tree, sorted by coastline angle relative to, say, North. When an alien is tagged, insert its angular position using normal AVL `insert` in $O(\log n)$ time. When an alien teleportation log is recorded, `find` the alien in the tree, and if the alien is there, remove it from the AVL tree, change its position, and re-`insert`, also in $O(\log n)$ time. To find the k closest aliens to a queried location, `find` the location in the tree in $O(\log n)$ time, and then repeatedly find the next or previous alien in the order based on which is closer to the queried location, using `successor` and `predecessor`. Angles are cyclicly ordered, so if the minimum or maximum angle contained in the tree is reached, we modify `successor` and `predecessor` to wrap around. Because the successor and predecessor calls are consecutive, all of them together will cost no more than $O(k \log n)$. A tighter upper bound is $O(k + \log n)$, but since $k = 20$ is constant, both are logarithmic, so we will accept either bound.

Rubric:

- 1 point for mentioning a balanced BST
- 1 point for sorting on coastline angle
- 1 point for insert
- 2 points for change key
- 3 points for a correct algorithm for query
- 2 points for running times
- Partial credit may be awarded

Problem 3-4. [40 points] **Programming**

Cole the cold coder is fed up. Every day he gets up, dress appropriately for the weather from yesterday, and leaves for class. But alas! The weather is drastically different: now either too hot or too cold. Cole decides to build a system to predict today's temperature given only the temperature from the day before, using a crazy new AI technique called *wide learning* (which he hears is all the rage). Like deep learning, wide learning attempts to approximate a function $y = f(x)$ by looking at many examples, i.e., pairs (x_i, y_i) of observed inputs and outputs. Cole's wide learning algorithm will accept two inputs: yesterday's temperature x and a confidence interval w. The algorithm will return a prediction for today's temperature y^* by taking the average temperature from all example outputs y_i from the wide set of sample inputs x_i that are within w degrees of x (i.e. $|x - x_i| < w$). Cole writes a quick Python program to implement his algorithm:

```python
examples = []

def add_example(xi, yi):
    examples.append((xi, yi))

def predict(x, w):
    total, count = 0, 0
    for xi, yi in examples:
        if abs(xi - x) < w:
            total += yi
            count += 1
    return (total / count) if (count != 0) else 0
```

(a) [5 points] What is the running time of Cole's prediction function terms of the number of examples? What about the running time of his `add_example` function?

Solution: Cole's `add_example` algorithm runs in (amortized) $O(1)$ time (for now, students may state $O(1)$ without qualification), while the running time of his `predict` function is linear in the number of examples, as he evaluates each example.

Rubric:

- 2 points for `add_example` running time
- 3 points for `predict` running time

Cole is tired of waiting so long for his predictor to run. He decides to store his examples in an augmented binary search tree, hoping that adding an example and making a prediction will each be fast, requiring at most logarithmic time.

(b) [10 points] To compute averages in sub-linear time, Cole cannot afford to visit every node contained in a range. Prove that at most **one** node of a BST can have left and right sub-trees that each contain keys both inside and outside a given range. If you know that every node in a sub-tree is within range, can you think of a way to pre-compute information to store at each node to shortcut computation?

Solution: Suppose for contradiction there exist two nodes A and B for which both left and right sub-trees contain keys both inside and outside a given range. First, the key a at node A is in range or else by the BST property, no key would be in range in one of the left or right sub-trees, and similarly for the key b at node B. Second, the minimum key a^- and maximum key a^+ in A's sub-tree are both out of range, or else no key would be out of range in the left or right sub-tree respectively.

If we have A with key a where the nodes underneath A go from a^- to a^+ where a^- and a^+ are to the left and right of the given range $[l,\ r]$, with $a^- < l$, and $a^+ > r$, then the A subtree needs to contain all nodes in the entire BST that are in the range $[l,\ r]$. To see this, assume that A is not the root (the root contains all nodes) and that A is the left child of its parent, then the parent node and nodes in the right subtree of the parent will all have keys that are $> a^+ > r$, i.e., outside the $[l,\ r]$ range. If A is the right child of the parent, then the parent node and nodes in the left subtree of the parent will all have keys that are $< a^- < l$.

Since b is also in range, B must also be in A's sub-tree. But by the same argument, A must be in B's sub-tree. Both cannot be true, a contradiction, completing the proof.

Rubric:

- 10 points for a complete proof
- Partial credit may be awarded

(c) [10 points] Describe how Cole can augment his binary search tree to efficiently add examples and calculate predictions, each in logarithmic time with respect to the number of examples stored in the tree. Remember to argue correctness and demonstrate the running time of your algorithm.

Solution: At each node, we store additional information about its sub-tree to shortcut computation: sub-tree min and max to make range querying faster, the number of keys stored in the sub-tree, and the sum total of y values stored in the sub-tree. To add an example, `insert` as in normal AVL, and while maintaining height on ancestor nodes, we can maintain the additional information in the AVL `update` method by computing min, max, sum, and count from the same sub-tree properties of its children. To make a prediction, run `find` to find the highest node that is within the queried range. Then for each child, recursively find the count and sum of keys contained in its sub-tree that are in the queried range. By part (a), at most one node makes two recursive calls over the height of the tree, while all other nodes make zero or one. Thus only a logarithmic number of nodes are touched, with constant work done at each, leading to $O(\log n)$ running time.

Rubric:

- 1 point for stating augmentations
- 2 points for insertion maintence
- 4 points for prediction algorithm
- 3 points for running time

- Partial credit may be awarded

(d) [15 points] Implement `predict(x, w)` in a Python class `TemperatureLog` that extends the `AVL` class provided. You can download a code template containing some test cases from the website. Submit your code online at `py.mit.edu/6.006`.

Solution:

```python
class TemperatureLog(AVL):
    def __init__(self, key = None, parent = None):
        '''Augment AVL with additional attributes'''
        super().__init__(key, parent)
        self.Xmin, self.Xmax = 0, 0
        self.Ysum, self.Ynum = 0, 0

    def update(self):
        '''Augment AVL update() to fix any properties calculated from children'''
        super().update()
        self.Xmin = self.left.Xmin  if self.left  else self.key[0]
        self.Xmax = self.right.Xmax if self.right else self.key[0]
        self.Ynum = 1
        self.Ysum = self.key[1]
        for child in (self.left, self.right):
            if child is not None:
                self.Ynum += child.Ynum
                self.Ysum += child.Ysum

    def add_sample(self, x, y):
        '''Add a transaction to the transaction log'''
        super().insert((x, y))

    def predict(self, x, w):
        '''Return a temperature estimate from query temperature and range.'''
        if self.key is None:
            return None
        Ynum, Ysum = self.range_query(x - w, x + w)
        return Ysum / Ynum if Ynum != 0 else 0

    def range_query(self, Xmin, Xmax):
        range_Ysum, range_Ynum = 0, 0
        # add in key of root if in range
        if Xmin < self.key[0] < Xmax:
            range_Ysum = self.key[1]
            range_Ynum = 1
        for child in (self.left, self.right):
            if child is not None:
                # add subtree if completely in range
                if (Xmin < child.Xmin) and (child.Xmax < Xmax):
                    range_Ysum += child.Ysum
                    range_Ynum += child.Ynum
                # recurse on subtree if partially in range
                elif not (child.Xmax <= Xmin) and not (Xmax <= child.Xmin):
                    Ynum, Ysum = child.range_query(Xmin, Xmax)
                    range_Ysum += Ysum
                    range_Ynum += Ynum
                # otherwise out of range
        return range_Ynum, range_Ysum
```