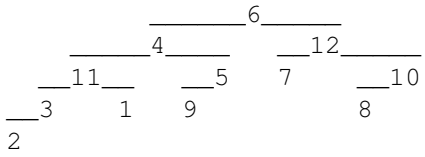
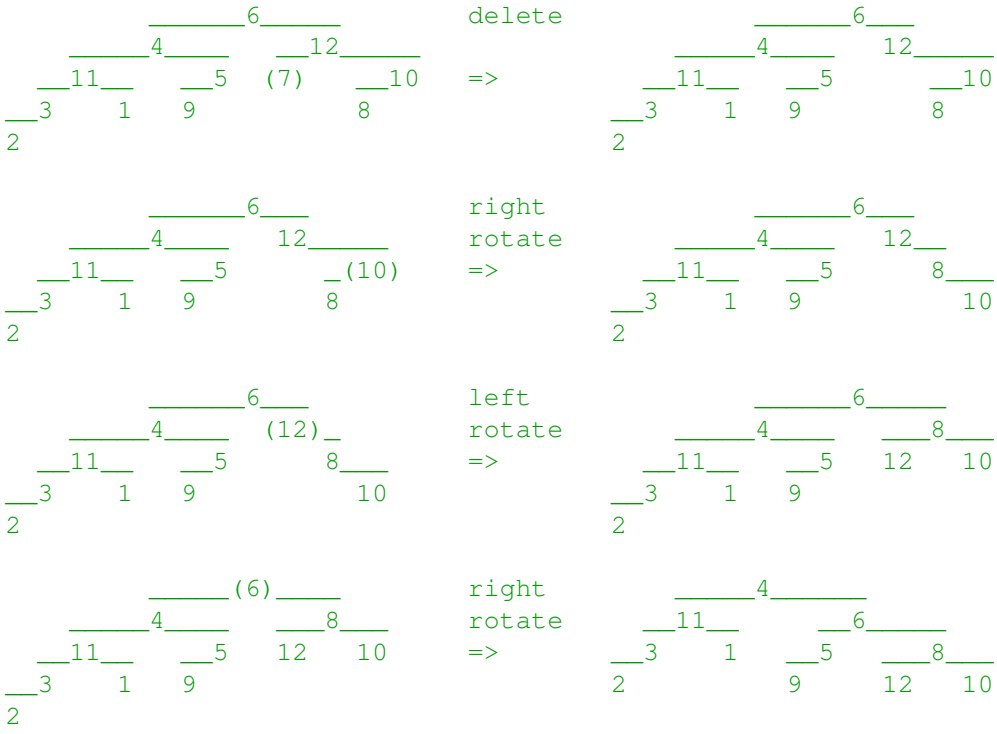
Problem Session 4

Problem 4-1. Sequence Rotations

Below is a Sequence AVL Tree `T`. Perform operation `T.delete_at(8)` and draw the tree after each rotation operation performed during the operation.



Solution:



Problem 4-2. Fick Nury

Fick Nury directs an elite group of n superheroes called the Revengers. He has heard that supervillian Sanos is making trouble on a distant planet and needs to decide whether to confront her. Fick surveys the Revengers and compiles a list of n polls, where each poll is a tuple matching a different Revenger's name with their integer opinion on the topic. Opinion $+s$ means they are for confronting Sanos with strength s, while opinion $-s$ means they are against confronting Sanos with strength s. Fick wants to generate a list containing the names of the $\log n$ Revengers having the strongest opinions (breaking ties arbitrarily), so he can meet with them to discuss. For this problem, assume that the record containing the polls is **read-only** access controlled (the material in classified), so any computation must be written to alternative memory.

(a) Describe an $O(n)$-time algorithm to generate Fick's list.

Solution: Build a maximum priority queue containing all Revengers (together with their names). The priority queue stores at most n Revengers, so should take up no more than $O(n)$ space. Key each Revenger r_i with opinion s_i on the pair $(|s_i|, i)$ to make keys unique. Then delete the maximum keyed Revenger $\log n$ times and return the results. The running time of this algorithm is $B(n) + (\log n) \cdot D$ where $B(n)$ is the time to build a priority queue of size n and D is the time for delete max. We can achieve the desired running time by using a binary max-heap as our priority queue, since build takes $O(n)$ time and deletions take amortized $O(\log n)$ time.

(b) Now suppose Fick's computer is only allowed to write to at most $O(\log n)$ space. Describe an $O(n \log \log n)$-time algorithm to generate Fick's list.

Solution: Build a minimum priority queue containing the opinions of the first $\log n$ Revengers in the list (together with their names). Key each Revenger r_i with opinion s_i on the pair $(|s_i|, i)$ to make keys unique. Then repeat the following procedure for the remaining Revengers, while maintaining the invariant that after processing Revenger i, the priority queue contains the $\log n$ Revengers from the first i having the highest absolute opinion, which is true for $i = \log n$ and would solve our problem when $i = n$: (1) delete the minimum Revenger r^* from the priority queue, (2) compare the key of r^* to that of the i^{th} Revenger r_i, and (3) insert whichever is larger into the priority queue. If $r_i < r^*$, then r_i is smaller than every Revenger in the priority queue so is not in the top $\log n$. Alternatively, if $r^* < r_i$, then r_i is larger than the k^{th} largest found so far. In either case, adding the larger to the priority queue maintains the invariant. In all cases, the priority queue stores at most $\log n$ Revengers, so should take up no more than $O(\log n)$ space. The running time of this algorithm is $B(\log n) + (n - \log n) \cdot (I + D)$ where $B(\log n)$ is the time to build a priority queue of size $\log n$ and I and D are the time for insertion and delete min respectively. Thus we can use either a Set AVL tree or a min-heap as our priority queue to achieve $O(n \log \log n)$ running time, using at most $O(\log n)$ space.

Problem 4-3. SCLR

Stormen, Ceiserson, Livest, and Rein are four academics who wrote a very popular textbook in computer science, affectionately known as SCLR. They just found k first editions in their offices, and want to auction them off online for charity. Each bidder in the auction has a unique integer bidder ID and can bid some positive integer amount for a single copy (but may increase or decrease their bid while the auction is live). Describe a database supporting the following operations, assuming n is the number of bidders in the database at the time of the operation. For each operation, state whether your running time is worst-case, expected, and/or amortized.

`new_bid(d, b)`	record a new bidder ID d with bid b in $O(\log n)$ time
`update_bid(d, b)`	update the bid of existing bidder ID d to bid b in $O(\log n)$ time
`get_revenue()`	return revenue from selling to the current k highest bidders in $O(1)$ time

Solution: First, observe that operation `update_bid` requires finding and modifying the bid of a bidder given their ID, so maintain a dictionary containing the bidders keyed by bidder ID (call it the bidder dictionary). Either a hash table or Set AVL tree will work for this purpose, where using a hash table will yield amortized expected running times for the first two operations.

We will also need to keep track of an ordering of bids stored: in particular the sum of the k highest at any given time. To do this, maintain two priority queues: a min priority queue containing the k highest bids, and a max priority queue containing the remaining $n - k$ bids. Because we need $O(\log n)$ performance on priority queue insertions or deletions, we can use either Set AVL trees or binary heaps to implement the priority queues, where using binary heaps will yield amortized running times for the first two operations.

In order to find where each bidder is in the priority queues, we store with each bidder in the bidder dictionary a cross-linking pointer to the bidder's bid in one of the priority queues (i.e., its AVL node or index in a binary heap). In what follows, we will assume a Set AVL tree is used to implement the bidder dictionary and the priority queues. In addition, maintain the sum B of all bids in the min priority queue so that we can return it in constant time.

To implement `new_bid(d, b)`, remove the smallest bid b' with bidder ID d' from the min priority queue in $O(\log n)$ time, and decrease B by its bid amount b'. Then compare b to b'. Whichever is larger is among the k highest bidders seen so far while the other is not, so insert the larger into the min priority queue in $O(\log n)$ time while increasing B by its bid; and insert the smaller into the max priority queue, also in $O(\log n)$ time. In addition, add d to the bidder dictionary pointing to its bid location in a priority queue, and update the pointer associated with bidder d', each in $O(\log n)$ time. This procedure maintains the invariants of the data structures presented, and runs in $O(\log n)$ total time in the worst case.

To implement `update_bid(d, b)`, find existing bidder d's bid location in the priority queues using the bidder dictionary in $O(\log n)$ time, and then remove its record from its priority queue, also in $O(\log n)$ time. If the min priority queue has only $k - 1$ bids, remove the max from the max priority queue and insert it into the min priority queue in $O(\log n)$ time. While doing this, recompute B in $O(1)$ time and update the cross-linked pointers in $O(\log n)$ time. Now that the bidder with ID d has been removed, we can now reinsert it as a new bidder with updated bid b and

insert using `new_bid` as above. This operation performs a constant number of calls to $O(\log n)$ time operations, so this operation runs in worst-case $O(\log n)$ time.

To implement `get_revenue()`, simply return the stored value of B in worst-case $O(1)$ time.

Problem 4-4. Receiver Roster

Coach Bell E. Check is trying to figure out which of her football receivers to put in her starting lineup. In each game, Coach Bell wants to start the receivers who have the highest **performance** (the average number of points made in the games they have played), but has been having trouble because her data is incomplete, though interns do often add or correct data from old and new games. Each receiver is identified with a unique positive integer jersey number, and each game is identified with a unique integer time. Describe a database supporting the following operations, each in **worst-case** $O(\log n)$ time, where n is the number of games in the database at the time of the operation. Assume that n is always larger than the number of receivers on the team.

`record(g, r, p)`	record p points for jersey r in game g
`clear(g, r)`	remove any record that jersey r played in game g
`ranked_receiver(k)`	return the jersey with the k^{th} highest performance

Solution: First, observe that operations require finding and modifying records for a receiver given a jersey number in $O(\log n)$ time, so maintain dictionary containing the receivers keyed by unique jersey (call it the jersey dictionary). Since we need to achieve worst-case $O(\log n)$ running time, we cannot afford the expected performance of a hash table, so we implement the dictionary using a Set AVL tree.

For each receiver, we will need to find and update their games by game ID, so for each receiver in the jersey dictionary, we will maintain a pointer to their own Set AVL tree containing that receiver's games keyed by game ID (call this a receiver's game dictionary). With each receiver's game dictionary, we will maintain the number of games they've played and the total number of points they've scored to date. We can compare the performance of two jerseys from their respective number of games and points via cross multiplication.

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

To implement `record(g, r, p)`, find player r's game dictionary D in the receiver dictionary in $O(\log n)$ time. If game g is in D, update its stored points to p in $O(\log n)$ time and update the total number of points stored with r's game dictionary in $O(1)$ time. Otherwise, insert the record of game g into D in $O(\log n)$ time, and update the number of games and total points stored in $O(1)$ time. The performance of r may have changed, so find the node corresponding to r in the performance tree, remove the receiver'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(g, r)`, find player r's game dictionary D in the receiver dictionary as before and remove g (assuming it exists). Identically to above, maintain the stored number of games and total points, and update the performance tree together in worst-case $O(\log n)$ time.

To implement `ranked_receiver(k)`, find the k^{th} 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 jersey 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 player. This recursive algorithm only walks down the tree, so it runs in worst-case $O(\log n)$ time.

Problem 4-5. Warming Weather

Gal Ore is a scientist who studies climate. As part of her research, she often needs to query the maximum temperature the earth has observed within a particular date range in history, based on a growing set of measurements which she collects and adds to frequently. Assume that temperatures and dates are integers representing values at some consistent resolution. Help Gal evaluate such range queries efficiently by implementing a database supporting the following operations.

`record_temp(t, d)`	record a measurement of temperature t on date d
`max_in_range(d1,d2)`	return max temperature observed between dates d_1 and d_2 inclusive

To solve this problem, we will store temperature mesurements in an AVL tree with binary search tree symantics keyed by date, where each node `A` stores a measurement `A.item` with a date property `A.item.key` and temperature property `A.item.temp`.

(a) To help evaluate the desired range query, we will augment each node with: `A.max_temp`, the maximum temperature stored in `A`'s subtree; and both `A.min_date` and `A.max_date`, the minimum and maximum dates stored in `A`'s subtree respectively. Describe a $O(1)$-time algorithm to compute the value of these augmentations on node `A`, assuming all other nodes in `A`'s subtree have already been correctly augmented.

Solution: Each of these augmentations can be computed in $O(1)$ time via the algorithms below:

Augmentation `A.max_temp` can be computed by taking the max of `A.item.temp`, `A.left.max_temp`, and `A.right.max_temp` (when they exist).

Augmetation `A.min_date` is `A.left.min_date` if `A` has a left child, and `A.item.key` otherwise.

Augmetation `A.max_date` is `A.right.max_date` if `A` has a right child, and `A.item.key` otherwise.

(b) A subtree **partially overlaps** an inclusive date range if the subtree contains at least one measurement that is within the range **and** at least one measurement that is outside the range. Given an inclusive date range, prove that for any binary search tree containing measurements keyed by dates, there is at most one node in the tree whose left and right subtrees both partially overlap the range.

Solution: Let's say that a node **branches** on a range if both its left and right subtrees both partially overlap the range. Then the question asks us to prove that at most one node in a binary search tree branches on a given range. First, observe that any node that branches is within the range; otherwise the range would not be continuous. Suppose for contradiction that two distinct nodes p and q branch on range (d_1, d_2). Let x be the lowest common ancestor of p and q. Since p and q are in range, then x is too since x appears between p and q in the traversal order. At least one of p and q is not x, so without loss of generality, assume p is not x and that p is in the left subtree of x. Then p and x are both in range, but the right subtree of p is between p and x in traversal order and contains keys that are not in range, a contradiction.

(c) Let `subtree_max_in_range(A, d1, d2)` be the maximum temperature of any measurement stored in node `A`'s subtree with a date between d_1 and d_2 inclusive (returning `None` if no measurements exist in the range). Assuming the tree has been augmented as in part (a), describe a **recursive** algorithm to compute the value of `subtree_max_in_range(A, d1, d2)`. If h is the height of `A`'s subtree, your algorithm should run in $O(h)$ time when `A` partially overlaps the range, and in $O(1)$ time otherwise.

Solution: Given node A and range inclusive (d_1, d_2), the maximum temperature of any measurement stored in A's subtree is either the temperature at the node itself, or it is in A's left or right subtree. Because we have augmented by the min and max dates within each node's subtree, we can evaluate whether the dates in A:

1. is disjoint from the range (`(A.max_date < d1) or (d2 < A.min_date)`),
2. fully overlaps the range (`d1 <= A.min_date and A.max_date <= d2`), or
3. otherwise partially overlaps the range,

in $O(1)$ time via a constant number of comparisons. In case (1), no measurements in the subtree are within range, so we can return `None` in $O(1)$ time. In case (2), all measurements in the subtree are within range, so we can return `A.max_temp` in $O(1)$ time, which is correct by definition of the augmentation. Lastly, in case (3), recursively compute the max temperature in range for each of A's child subtrees and return the max between them and the temperature at A (since the temperature at A must be in range). This algorithm takes at most $O(h)$ time. To see this, observe that the claim in part (b) ensures that at in all but one case (3) node, a recursive call in at least one of its child subtrees will take $O(1)$ time. So the shape of the recursive calls will be the union of at most two paths from the root to another node in the tree, touching at most $O(h)$ nodes along the way, doing at most $O(1)$ work at each.

(d) Describe a database to implement operations `record_temp(t, d)` and `max_in_range(d1, d2)`, each in **worst-case** $O(\log n)$ time, where n is the number of unique dates of measurements stored in the database at the time of the operation.

Solution: Store the measurements in a Set AVL tree keyed by date. Augment the tree as in part (a), which can be maintained at each node in $O(1)$ time from the augmenta-

tions of the node's children, without affecting the running time of the other AVL tree operations.

To implement `record_temp(t, d)`, check to see whether date d already exists in the tree. If it does, delete the measurement from the tree, and keep whichever of the two measurements has higher temperature. Then in either case, insert the new measurement into the tree in worst-case $O(\log n)$ time. This procedure maintains the invariant that the temperature stored at date d is the highest temperature recorded for that day.

To implement `max_in_range(d1, d2)`, we reduce to part (c) by simply returning `subtree_max_in_range(T.root, d1, d2)`, where `T.root` is the stored root node of the tree. This algorithm is correct by (c), and runs in worst-case $O(\log n)$ time since the height of an Set AVL tree is $O(\log n)$.

(e) Implement your database in the Python class `Temperature_DB` extending the `Set_AVL_Tree` class provided; you will only need to implement parts (a) and (c) from above. You can download a code template containing some test cases from the website. Submit your code online at `alg.mit.edu`.

Solution:

```
from Set_AVL_Tree import BST_Node, Set_AVL_Tree

class Measurement:
    def __init__(self, temp, date):
        self.key  = date
        self.temp = temp

    def __str__(self): return "%s,%s" % (self.key, self.temp)

class Temperature_DB_Node(BST_Node):
    def subtree_update(A):
        super().subtree_update()
        A.max_temp = A.item.temp
        A.min_date = A.max_date = A.item.key
        if A.left:
            A.min_date = A.left.min_date
            A.max_temp = max(A.max_temp, A.left.max_temp)
        if A.right:
            A.max_date = A.right.max_date
            A.max_temp = max(A.max_temp, A.right.max_temp)

```

```
22      def subtree_max_in_range(A, d1, d2):
23          if (A.max_date < d1) or (d2 < A.min_date):    return None
24          if (d1 <= A.min_date) and (A.max_date <= d2): return A.max_temp
25          t = None
26          if d1 <= A.item.key <= d2:
27              t = A.item.temp
28          if A.left:
29              t_left = A.left.subtree_max_in_range(d1, d2)
30              if t_left:
31                  if t:   t = max(t, t_left)
32                  else:   t = t_left
33          if A.right:
34              t_right = A.right.subtree_max_in_range(d1, d2)
35              if t_right:
36                  if t:   t = max(t, t_right)
37                  else:   t = t_right
38          return t
39
40  class Temperature_DB(Set_AVL_Tree):
41      def __init__(self):
42          super().__init__(Temperature_DB_Node)
43
44      def record_temp(self, t, d):
45          try:
46              m = self.delete(d)
47              t = max(t, m.temp)
48          except: pass
49          self.insert(Measurement(t, d))
50
51      def max_in_range(self, d1, d2):
52          return self.root.subtree_max_in_range(d1, d2)
```