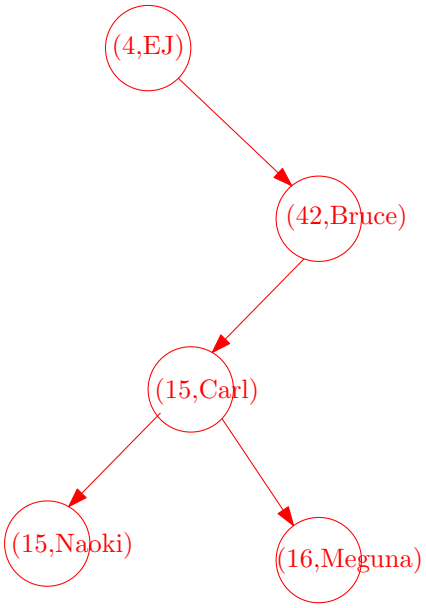
Comp 160: Algorithms **Recitation 6**

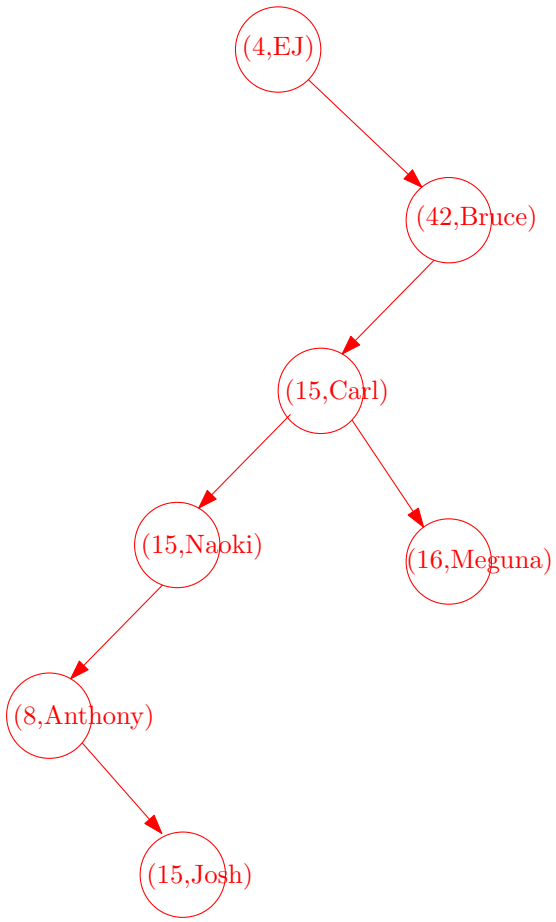
1. We want to modify a BST to allow having repeated items (say, we want to store TAs by their grade). We modify the main invariant as follows: for any item, all items in the left subtree are less than **or equal** to it, and all items in its right subtree are strictly greater than it.

   (a) As a warm-up, insert the following TAs into the modified tree: (4,EJ), (42,Bruce), (15,Carl), (15,Naoki), (16,Meguna), (8,Anthony), (4,William), (42,Kevin), (16,Radhika), (23,Ashley), (42,Cole), (23,Maria), (4,Brian), (15,Josh), (8,Robby), (8,Mert), (16,Rosa), and (23,Bruce). Draw the resulting tree

   This is a simple BST insertion practice. You should be fairly familiar now. Let's draw just a few insertions:



   Notice how the repeated values are treated. Following insertions will be all over the place, but consider after we have inserted (8, Anthony) and (15,Josh):



Notice how in particular students with the same grade are not consecutive in the tree (Many insertions are omitted for clarity).

Fun fact: these are NOT the grades that the TAs had in 160. You will have to ask them if you want to know their grades :)

(b) Devise an algorithm for retrieving **all** copies in the tree whose grade is g. Runtime must be proportional to the depth of the tree. First, describe the steps of your algorithm.

```
SearchWithDuplicates(node n, grade g)
1) If n= ∅ Return
2) If (n→grade < g) SearchWithDuplicates(n→right, g)
3) If (n→grade > g) SearchWithDuplicates(n→left, g)
4) If (n→grade = g)
        Report (n→grade, n→name)
        SearchWithDuplicates(n→left, g)
```

Note how the only difference with normal AVL search is in the last line only: in a normal BST search we stop as soon as we found the solution. In this case we still need to continue searching.

(c) Justify briefly (1-3 sentences) the correctness of your algorithm.

The core of the search is like a normal BST:

- In this modified BST, the right subtree of any node contains grades strictly larger (as usual). However, left subtree has grades smaller **or equal**.
- If n→grade $<$ g we discard nodes with strictly lower grade than our target.
- If n→grade $>$ g we discard nodes with strictly larger grade than our target.
- If n→grade $=$ g we report the node, but also check in the side that could contain other copies (alternatively, we discard nodes that are strictly larger).

(d) What is the runtime of your algorithm? Justify briefly (1-3 sentences) the runtime of your algorithm.

Runtime is $O(h)$, where h is the height of the tree: at each node we do $O(1)$ computation and then we recurse on one child. Thus, the worst possible case is that we traverse the full height of the tree. For a full proof, we would need to show that the recurrence $T(h) = T(h-1) + O(1)$ solves to $T(h) = O(h)$ (but this is not asked in the question).

(e) Is it possible to now use AVL rules (or a variation) to maintain the tree balanced? Why or why not?

This tree cannot be balanced as multiple copies of an item are placed in a tree. Consider, for example, the following tree:

When we attempt to balance it we get:

However, this violates the condition that all duplicates must be in the left subtree. Therefore as long as there are more than two duplicates of an item in a tree in cannot be balanced.

Note that further modifications of the AVL (say, allow an unbalance of up to 3 between children) would not help. The problem is that a sequence of items with the same key have only one way of being stored (forming a long unbalanced chain).

(f) Say that we consider a different invariant: instead of storing copies with the same key always to the left, we flip a coin and store it to the left with 50% probability (otherwise we store it in the right). Say that now we insert n objects in the tree.

Devise an algorithm for retrieving **all** copies of a specified item under the modified tree. What is the runtime?

The algorithm is as before, but now in step 4 we must recurse on both left and right subtrees. Runtime now increases to $\Theta(n)$ (in the worst case we must visit all nodes of the tree).

Note that this idea is not much worse, since h can also be potentially $\Theta(n)$. In short, both methods are terrible.

(g) Can you design a better way to handle duplicate keys? What is the takeaway?

Rather than having a node per duplicate key, we should store all of them together in a single node. All elements that have the same key should be in a secondary data structure that is pointed from the BST.

The type of data structure that we use to hold duplicates depends on the data itself. If we will just need to report all copies, then a simple linked list works. If you may want to do a search within elements that have the same key, then we can use any data structure that helps with search...but then why didn't we use the secondary key as part of the big key?

The takeaway is that keys should be unique. If they are not (like in the case we are indexing students by their grade)... then we should modify the key so that it is unique. In this case, an easy solution is to have the key as the pair (grade, name). When comparing two keys, you first compare their grades. In case of a tie of grades you use the name (say, alphabetical) to break the tie. This will guarantee searchers in $\Theta(\log n)$ no matter how many repeated grades you have.

2. Let $S = \{s_1, \ldots, s_n\}$ be a set containing real numbers (for simplicity, for the remainder of this exercise assume that all numbers are distinct).

(a) The *min gap* is defined as $\min_{i \neq j} |s_i - s_j|$ (in other words, the two numbers that are closest to each other). Give a very simple algorithm to compute the min gap in $\Theta(n^2)$ time. For simplicity, you can assume that numbers are stored in an array, so $S[1] = s_1$, and so on.

The brute force approach is: for all possible values $1 \leq i < j \leq n$ do: compute $|s_i - s_j|$. Report the minimum of all computed values. This algorithm needs $\Theta(n^2)$ time.

Note: min gap is undefined if S contains 0 or 1 elements.

(b) Can you use some observations to reduce the computation time to $O(n \log n)$? Look for some property that s_i and s_j would have.

The key observation is that the smallest gap is always between two numbers that would be consecutive if we sort the array. In other words, it always holds that $j = i + 1$ (after sorting).

We can prove this statement by contradiction: assume that the min gap is $|s_i - s_j|$ for some indices i, j such that $s_i < s_j$ and $i + 1 < j$. Then, we have that $s_i < s_{i+1} < s_j$. However, if this happens, we have $|s_i - s_{i+1}| < |s_i - s_j|$ which is a contradiction.

Once we have this observation the algorithm follows naturally. Sort S (using any $\Theta(n \log n)$ time algorithm). After that, examine adjacent elements to find the min gap. This can be done in $\Theta(n)$ additional time by scanning through the array and computing the gap between subsequent numbers. The smallest of the computed gaps will be the solution. Total time is $\Theta(n \log n + n) = \Theta(n)$.

(c) After you spent $\Theta(n \log n)$ time, we insert a new element into S. Can we update the min gap efficiently? What if we inserted several numbers one after each other?

Although possible, it is a bit difficult to update. Options are:

- Maintain S in a sorted array. That way, when we insert a new number we can quickly update min gap if needed (it is either the old gap or the gap between the newly inserted number and one of its neighbors). The problem is that inserting in a sorted array takes $\Theta(n)$ time
- We could instead use an unsorted array. Insertion is super fast ($O(1)$ time, we will do exact details later on the semester), but updating the min gap needs $\Theta(n)$ time (min gap after the new insertion is either (a) the old min gap or (b) created by the newly inserted element and one of its neighbors (if we sorted the array). We could store (a), but the naive way to compute (b) takes linear time (need to scan the whole array).

There are other variations but the quick summary is: using arrays to store some information that needs to be updated with insertions is a bad idea. For these kind of problems, your reflex should be to consider an augmented tree instead.

(d) Let's try solving this problem using an augmented tree. We will store all numbers in a balanced tree (such as AVL). In order to determine what to augment the tree with, we need to characterize the solution. How can a solution look like? In other words, could it be that both s_i and s_j are to the right of the root? can both be to the left of the root? Are there other possible cases?

Let r be the value stored at the root. There are essentially five cases:

- $s_i < s_j < r$. In this case, both s_i and s_j are in the left subtree.
- $r < s_i < s_j$. This is a similar situation when both are in the right subtree.
- $s_i < r < s_j$. This would correspond to s_i being in the left subtree and s_j on the right subtree. This cannot happen because r would be between them (we already argued that s_i and s_j cannot have another value of S between them).
- $r = s_i$. That is, r is the smallest number of the two that form the min gap. In this case s_j must be the smallest possible value in the right subtree.
- $r = s_j$. That is, r is the largest number of the two that form the min gap. In this case s_i must be the largest possible value in the left subtree.

(e) Using the previous answer let's choose how to augment the tree. Imagine you are at the root. You have access to the element of S that is stored there. Also, you can ask questions to your left and right children (for example, you could ask *how many elements are in your subtree?*), but you cannot look further down in the tree. What questions do you need ask to your children to compute min gap? Can we use augment the tree and store that information?

In last answer we had 4 cases:

- $s_i < s_j < r$. We could get $|s_i - s_j|$ by asking *what is the smallest possible gap you can create with the elements in your subtree?* to the left child.
- $r < s_i < s_j$. We could get $|s_i - s_j|$ by asking the same question to the right child.

- In the other cases two cases r is part of the solution. We can compute the min gap by asking the left tree *what is the largest value in your subtree?* and by asking the right tree *what is the smallest value in your subtree?*

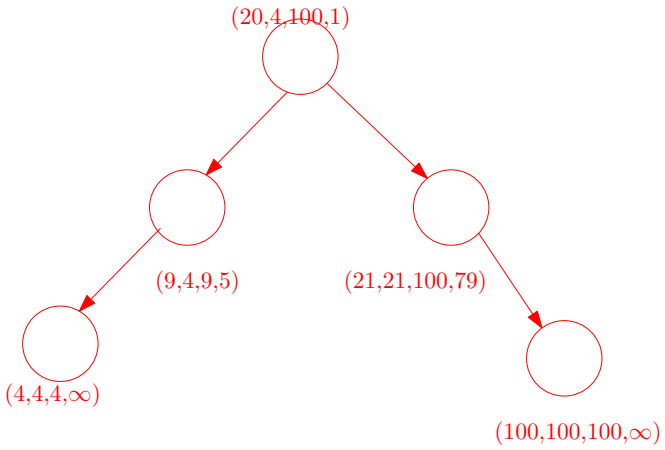
So, in addition to the usual attributes of a balanced tree (such as `value`, left child ℓ, and right child r), at each node we need to store 3 additional pieces of information: `mingap` (the smallest gap that can be created with the elements in that subtree), `maxtree` (the largest number stored in that subtree) and `mintree` (the smallest number stored in that subtree).

The computation of min gap will be the minimum of the 4 possible values. That is, if the root is n, we have:

$$\begin{aligned} mingap(n) = \min\{ & \; n \to \ell \to \texttt{mingap}, && (1)\\ & |(n \to \ell \to \texttt{maxtree}) - (n \to \texttt{value})|, && (2)\\ & |(n \to \texttt{value}) - (n \to r \to \texttt{mintree})|, && (3)\\ & n \to r \to \texttt{mingap}\} && (4) \end{aligned}$$

(f) Draw a balanced tree with 5 nodes, for each node show the augmented attributes.

See image below. For each node the four numbers represent (in order): `value, mintree, maxtree, mingap`. Notice how all leaves have `mingap`= ∞. This is because they only contain one element and thus the smallest gap is undefined.



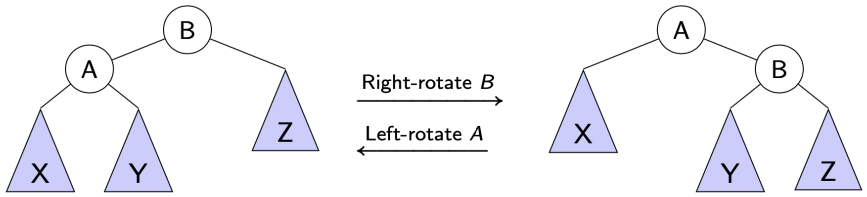
(g) Say that we have a tree augmented the tree with that information. Describe how you would handle an insertion. Make sure to describe how to update any augmented attribute.

Insertion follows the core of a regular AVL insertion: first we search for the value t to insert. Search will be unsuccessful, but that way you know where to insert the node. When you create a node that will store t, make sure to set `mintree` and `maxtree` to t (and `mingap` to ∞). This is because the node we created is a leaf of the tree.

Insertion is not over after the node has been created: we must also traverse the tree upwards towards the root checking for balance. Because the tree is augmented, in

addition to checking for balance we may have to update `mintree` and `maxtree` to t (if t is smaller/larger than whatever is currently stored). `mingap` could also change (as soon as t updates `mintree` and `maxtree`, it can have an impact in `mingap` of the parent node).

In a normal AVL insertion, the process continues until either (1) we reach the root (and thus, there is no parent to traverse to), or (2) we need to do a rotation. In the latter case we must make sure to update all augmented information:



As usual, the augmented information stored at X, Y, and Z is unchanged in neither left nor right rotation.

First let's look at how to update `mintree`: Since A always has X as its left child, A's `mintree` is unaffected in either rotation: the smallest value will be in X (or A itself if $X = \emptyset$).

B's `mintree` will change in both left and right rotations. In a right rotation we are shifting the placement of children, so we should set $B \rightarrow$`mintree` equal to $Y \rightarrow$`mintree` (if $Y \neq \emptyset$) or simply to $B \rightarrow$`value` (otherwise). A left rotation is similar but now we inserting nodes into the subtree of B (so we set B's `mintree` equal to A's `mintree`).

Updates to `maxtree` are symmetric: B's right tree does not change in neither rotation, so the value of B's `maxtree` will not change. A's right tree will change in both rotations (to the value of its right child's `maxtree`, or its own value if the right child is empty).

Once `mintree` and `maxtree` of a node and is children has been updated, we can adjust `mingap` of both A and B by using the formula described in the previous section. Note how we did not change any attribute of X, Y, or Z in the rotation.

Beware: in a normal AVL insertion when a rotation is finished we would stop (i.e., no need to continue traveling to the root because everything is ok). However, after a rotation has finished we may still have to travel upwards to adjust the augmented information: say that the newly inserted element is the one that now defines the min gap. This information has to be propagated to the root. This is a minor technicality, but worth mentioning.

Overall, a constant number of operations per node, even if a rotation is needed. We repeat this for all nodes in the path to the root, giving a $\Theta(\log n)$ time bound.

(h) Can your structure handle deletions easily? If yes, explain. If not, can you handle deletions by further extending your data structure?

Pretty much the same as an insertion, but in reverse:

When we delete a node x we must (in addition to the usual balance operations) update the augmented information. Note that nodes in the subtrees of x (if any) are not affected, so as usual we only need to check the augmented information of nodes between x and the root.

At every node x' we can recompute `mintree`, `maxtree` and `mingap` by looking at the information stored at x' and the augmented information at the children (which has already been updated). Details are almost identical to an insertion (and can be done in constant time). Beware that a deletion can generate $\Omega(\log n)$ rotations (whereas insertions can only create $O(1)$ rotations).

Additional practice questions:

3. **Challenge:** say you want to solve the min gap problem by augmenting the tree, but you are only allowed to store the subtree size at every node. How would you do so? Would this structure work for deletions?

   If you store the subtree size at every node, you can do rank-finding operations (as shown in class). Thus, we maintain one global variable `minGap` that holds the min gap value. Queries clearly take $O(1)$ time (simply return the `minGap` global variable).

   Insertion is done in the standard way of AVLs (possibly with rotations, overall needing $O(\log n)$ time) plus the maintenance explained in rank finding.

   The only item left is updating `minGap`: by inserting a new number we may only reduce the gap and only if we created a smaller gap with one of the neighbors of the newly inserted element. Thus, once inserted, we find the rank r of the inserted element and then use rank queries to find the elements with ranks $r-1$ and $r+1$ (if they exist). Compare the two new gaps with `minGap` and update if necessary. These extra operations only need $O(1)$ time, so runtime is dominated by the $\Theta(\log n)$ time needed for an insertion.

   It is trickier to handle deletions, because if a deletion destroys the current min gap, the next smallest gap need not be nearby (unlike the case of insertion). So, after deleting an element you may have to traverse the whole tree to update `minGap`.

4. Given an array of distinct numbers, define a "flip" to be a pair of numbers such that the smaller one is to the right of the larger one. For example, in the array $[3, 2, 1]$, there are three flips (3 is flipped with both 1 and 2, and 2 is flipped with 1). Give an algorithm to count the number of flips in a given array in $O(n \log n)$ time.

   Note: there are (at least) three different strategies that can be used to solving the problem. One of them uses a technique learned in the first block of the semester. Another technique uses tools explained this week, and the third one we will discuss in the future. Gotta catch them all!

5. (binary tree recursion practice) **Leetcode question 101**: Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center). Leetcode gives bonus

points if you could solve it both recursively and iteratively. We give bonus points for a good algorithm structure, and proof of correctness.

6. **Challenge**: Say that you have an AVL tree with n elements. What is the possible range of ranks that the root can have? Express your solution as a function of n (that is, can the root of an AVL tree be the smallest element, regardless of the value of n? Can it have rank $n/2$? what about other ranks?)