Quiz 1 Solutions

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on the top of every page of this quiz booklet. Circle your recitation at the bottom of this page.
- You have 120 minutes to earn a maximum of 120 points. Do not spend too much time on any one problem. Read them all first, and attack them in the order that allows you to make the most progress.
- **You are allowed one double-sided letter-sized sheet with your own notes**. No calculators, cell phones, or other programmable or communication devices are permitted.
- Write your solutions in the space provided. If you need more space, write on the scratch pages at the end of the exam, and refer to the scratch pages in the solution space provided. Pages will be scanned and separated for grading.
- Do not waste time and paper rederiving facts that we have studied. Simply cite them.
- When writing an algorithm, a **clear** description in English will suffice. Pseudo-code is not required. But be sure to prove the required bound on running time.
- **Pay close attention to the instructions for each problem**. Depending on the problem, partial credit may be awarded for incomplete answers.

Problem	Parts	Points	Grade	Grader
0: Name	0	2		
1: Recurring Recurrences	2	10		
2: Theoretical Herpetology	2	10		
3: Theoretical Arborist	2	20		
4: Expect the Unexpected	2	6		
5: Firehose Sorting	1	10		
6: Peak Speed	1	11		
7: Resorting to Re-sorting	1	10		
8: Se7en Keys	3	26		
9: Smooth Operator	1	15		
Total		120		

Name: _____

Circle your recitation:	R01 Skanda Koppula	R02 Ludwig Schmidt	R03 Parker Zhao	R04 Gregory Hui	R05 Yonadav Shavit	R06 Atalay Ileri	R07 Anton Anastasov
	R08 Akshay Ravikumar	R09 Aradhana Sinha	R10 Ray Hua Wu	R11 Daniel Zuo	R12 Themistoklis Gouleakis	R13 Adam Hesterberg	R14 Ali Vakilian

Problem 0. [2 points] **What is Your Name?** (2 parts)

(a) [1 point] Flip back to the cover page. Write your name and circle your recitation section.

(b) [1 point] Write your name on top of each page.

Problem 1. [10 points] **Recurring Recurrences** (2 parts)

Solve the following recurrences, expressing your solution using asymptotic Θ notation:

(a) [5 points] $T(n) = 9T(\frac{n}{3}) + \Theta(n \log n)$

Solution: Because $n^{\log_3 9} = n^2$, and n^2 is asymptotically greater than $n \log n$ by more than a polylogarithmic factor, this is case 1 of the Master Theorem, and $T(n) = n^2$.

(b) [5 points] $T(n) = T(\frac{n}{2}) + \Theta(\log n)$

Solution: Because $n^{\log_2 1} = 1$, which is the same as $\log n$ to within a polylogarithmic factor (that is, $\log n$), this is case 2 of the Master Theorem, and $T(n) = \log n(\log n)$, or $\log^2 n$.

Common Mistake: Many students did not identify this to be case 2, which covers not only cases where the root and leaf functions to be compared are equal but also where they differ by only a polylogarithmic factor.

Problem 2. [10 points] **Theoretical Herpetology** (2 parts)

Analyze the **worst-case time complexity** of each of the following Python functions, expressing your answer using asymptotic Θ notation.

(a) [5 points] Assume that A is a list of n integers.

```
def double(A):
  C = []
  for x in A:
    C.append(x)
  for x in A:
    C.append(None)
  return C
```

Solution: There are two loops, each doing a $\Theta(1)$ append, so this runs in $\Theta(n)$.

Common Mistake: Several students said this function is $\Theta(n^2)$ because `append` might run in $\Theta(n)$. However, remember that `append` is $\Theta(1)$ amortized! The $\Theta(n)$ table doubling happends every once in a while, but if you average it out, n appends takes $\Theta(n)$.

(b) [5 points] Assume that A and B are both lists, each with n integers.

```
def all_a_in_b(A, B):
  for x in A:
    if x in B:
      continue
    else:
      return False
  return True
```

Solution: Note that `x in B` is an $O(n)$ operation – it scans the entirety of `B` looking for `x`. Consider the worst case where `A` and `B` are disjoint arrays – then, `x in B` always runs in $\Theta(n)$. Therefore, we have a loop of n $\Theta(n)$ searches, which means this function is $\Theta(n^2)$ in the worst case.

Problem 3. [20 points] **Theoretical Arborist** (2 parts)

You are given a rooted binary tree T (where one node r is given as the root, and each node stores a key, a left pointer, a right pointer, and a parent). You know that the nodes of T store n distinct keys, but you are not sure whether T has useful structure.

(a) [10 points] Give an $O(n)$-time algorithm that checks whether T represents a max-heap. (Your running time should be $O(n)$ for any height of the tree.)

Solution:

Here are two correct ways to solve this problem:

1. Run an in-order traversal of the given tree, and record the depth of each leaf. If the depth of the first leaf is d, then the depth of every subsequent leaf must be d or $d-1$ (for balance), and once one leaf of depth $d-1$ is encountered, all subsequent leaves must have depth $d-1$ (for left-justification). Also, at each node, check that its parent either doesn't exist or is at least as great as it, to check the heap property. This does $O(1)$ work per node, for a total time of $O(n)$.

2. Write the root in index 0 of an array. Recursively visit each of its children, writing the left child of a node (if it exists) at position i in the array into position $2i+1$ and the right child (if it exists) into position $2i+2$, and rejecting if either of its children (if they exist) are greater than the current node. If you try to write anything into the array at an index greater than $n-1$, reject. This indexing scheme numbers the nodes of the tree in increasing order by level and then by position within level, so the indices are 0 through $n-1$ if and only if the tree is left-justified and balanced. This does $O(1)$ work per node, for a total time of $O(n)$.

Common Mistakes:

The most common mistakes by far were to not realize that being a heap requires being balanced (that is, all leaves are at depth within 1 of each other) and that it requires being left-justified (that is, the unfilled level, if any, has all leaves as far left as possible).

(b) [10 points] Give an $O(n)$-time algorithm that checks whether T is a binary search tree. (Your running time should be $O(n)$ for any height of the tree.)

Solution:

Here are three correct ways to solve this problem:

1. Run an in-order traversal of the given tree, and output the keys in that order to an array. Check whether the array is sorted. (Alternatively, check that each traversed key is greater than or equal to the previous traversed key.) In class, we said that the BST property is equivalent to the in-order traversal of the tree; thus, the array will be sorted if and only if the tree is a BST. (We did not require proof of this if-and-only-if claim, but for full credit, we did require stating the claim.) This algorithm runs in $O(n)$ to do the in-order traversal and then run through the array and do $O(n)$ comparisons.

2. Recursively compute the min and max in each subtree, by first recursing in the two children, then setting the min [max] of the root to be the min [max] of the two children's min [max] and the root key. Then recursively check that each node x of the tree satisfies the BST property: x.left.max $\leq x$.key $\leq x$.right.min. This algorithm runs in $O(n)$ time because it spends $O(1)$ time at each node.

3. Recursively check the BST property, computing the required upper and lower bounds as you go. We start at the root with an upper bound of ∞ and a lower bound of $-\infty$. Upon visiting each node x, we check that x.key is between the current lower and upper bounds, then recurse into the two children of x. When recursing into x.left, we use x.key as the new upper bound; and when recursing into x.right, we use x.key as the new lower bound. By induction, this will check that all nodes in the left [right] subtree of x are less [greater] than or equal to x. This algorithm runs in $O(n)$ time because it spends $O(1)$ time at each node.

Common Mistake: The most common incorrect solution was to recursively check, for each node x, that x.left.key $\leq x \leq x$.right.key. This property is weaker than the BST property, which says that *all* keys in the left subtree of x (not just x.left) are less than or equal to x.key, and similarly for the right subtree of x. For example, the weaker property allows x.left.right.key to be larger than x.key, but that should be forbidden in a BST, because in particular it would make BST-SEARCH incorrect.

Problem 4. [6 points] **Expect the Unexpected** (2 parts)

Suppose you have a hash table of size m, storing n keys, where $m = \Theta(n)$, and collisions are resolved by chaining. Assume the simple uniform hashing assumption.

(a) [3 points] What is the **expected** running time of SEARCH(x)?

Solution: It is given than $m = \Theta(n)$, so in this hash table a search has expected runtime $\Theta(1)$.

Common Mistake: An answer of $\Theta(\frac{n}{m})$ can be simplified further because it was specified that $m = \Theta(n)$.

(b) [3 points] What is the **worst-case** running time of SEARCH(x)?

Solution: In the worst case, all keys are mapped to the same bucket, and moving through the chain could take $\Theta(n)$ time.

Problem 5. [10 points] **Firehose Sorting** (1 part)

Assume for simplicity that all MIT class numbers are of the form $x.y$ where $1 \leq x \leq 24$ and y comprises between 2 and 4 decimal digits. Give an $O(n)$-time algorithm to sort n class numbers (possibly including duplicates).

The desired order is the usual real-number ordering with a decimal point; for example: 6.004, 6.006, 6.008, 6.01, 6.02, 6.034, 6.042, 6.854, 18.03, 18.06, 18.100.

Solution: Multiply all numbers by 10000, then radix sort the resulting numbers, using counting sort on each digit, then divide all numbers by 10000. Because counting sort is stable, and the sorted entities are integers, the radix sort is correct. With a finite number of bins, radix sort takes O(n) time on a list of n integers, and multiplying and dividing all numbers by 10000 also take O(n) time, for O(n) time total.

Common Mistake: The original set of numbers cannot be radix sorted, because they are not integers. One must mention some way of transforming the numbers such that they could be radix-sorted.

Problem 6. [11 points] **Peak Speed** (1 part)

Recall that a ***peak*** (local maximum) in a 1D array $A[0\,..\,n-1]$ is an index i, $0 \leq i < n$, such that $A[i-1] \leq A[i] \geq A[i+1]$, where we imagine $A[-1] = A[n] = -\infty$.

Prove an $\Omega(\lg n)$ lower bound for finding a peak in an (unsorted) array $A[0\,..\,n-1]$, in the comparison model.

Hint: Consider arrays with only one peak.

Solution: Proving the $\Omega(\lg n)$ lower bound for finding a peak in the array A requires two steps: (1) showing that there are n possible locations for the peak, and (2) constructing a binary decision tree with n leaves, and using it to prove the lower bound.

Following the hint, consider an array A that has only one peak. The location of this peak could be any index i between 0 and $n-1$, inclusive. We do not know which index i corresponds to a peak, but we do know that there is exactly one such index i for which $A[i-1] \leq A[i] \geq A[i+1]$. Therefore, there are n possible locations for the peak.

Construct a binary decision tree with n leaves that correspond to the possible locations for the peak in the array A, i.e. there is exactly one leaf for each possible location i between 0 and $n-1$, inclusive. At every internal node of the decison tree we are making a comparison, and narrowing down the set of possible locations of the peak in A. Note that the path from the root of the decision tree down to a leaf corresponds to the execution of a peak finding algorithm. Because the decision tree is binary, its height is $\Omega(\lg n)$. Therefore, *any* algorithm for peak finding over A takes $\Omega(\lg n)$-time. This proves the lower bound. □

Common Mistake 1: The most common mistake was to use the $O(\lg n)$-time algorhtm for peak finding seen in class to prove the $\Omega(\lg n)$ lower bound. Note that a lower bound is a property of a problem, and not a property of an algorithm. It is incorrect to use an algorithm to prove a lower bound for a problem. Consider the problem of comparison-based sorting, and one particular algorithm for sorting: insertion sort. Because insertion sort takes $O(n^2)$ time, we can use the same logic to "prove" an $\Omega(n^2)$ lower bound for sorting. However, as we have seen in class, this is not the correct lower bound for comparison-based sorting.

Common Mistake 2: A common mistake was to derive a binary decision tree based on the $O(\lg n)$-time peak finding algorithm seen in class. Note that this is not correct, because a lower bound is a property of a problem, not a property of an algorithm for a given problem. The difference is quite subtle: to prove an $\Omega(\lg n)$ lower bound for finding a peak we have to show that there are instances of the peak finding problem on which *any* algorithm will take $\Omega(\lg n)$-time. If we only consider the algorithm seen in class, we are not ruling out the possibility that there could be an asymptotically faster algorithm. However, with the binary decision tree construction in the solution above, we are actually ruling out the possibility of an $o(\lg N)$-time algorithm for finding a peak in A.

Common Mistake 3: Another common mistake was to specify the particular comparisons that are going to be performed at the internal leaves of the binary decision tree. Note that in doing so, we are restricting the decision tree to only work for *some* of the possible algorithms, but not *all* of them, which is what we need. Therefore, for full credit, it is necessary to not specify any of the comparisons performed at the internal nodes of the binary decision tree.

Problem 7. [10 points] **Resorting to Re-sorting** (1 part)

After carefully sorting your array $A[0\,..\,n-1]$, teenage hackers have modified 10 values in your array. Unfortunately, you do not know which 10 values have been modified. Lacking any backups, you can't restore the original data, but you'd still like the data to be sorted.

Give an algorithm to sort A in $O(n)$ time. Your algorithm can only compare values, i.e., it must work within the comparison model.

Solution: You can assume either the items in A were initially sorted in an increasing order or a decreasing order. Here we assume that A was initially sorted in an increasing order and we aim to keep it in an increasing order even after the attack!

The idea is to use insertion sort and show that the runtime of insertion sort on this *almost sorted* array is $O(n)$. Starting from index 0 in A, for each index i, the runtime of $insert(A[i])$ is

1. At most $10 \cdot c$ if $A[i]$ is a non-hacked item (where c is a constant). It easily implied by the fact that insertion sort will never swap $A[i]$ and $A[j]$ if both are non-hacked items.
2. At most $n \cdot c$ if $A[i]$ is a hacked item.

Thus the total runtime is $(n-10)10 \cdot c + 10n \cdot c = O(n)$. Other correct arguments on bounding the runtime of insertion sort on A are acceptable, e.g.:

1. Showing that the total number of swaps performed by insertion sort on A is at most $10n$ because in each swap, at least one of the participants should be one of the hacked items.
2. Bounding the number of inversion correctly and using the fact that the runtime of insertion sort is $O(n+k)$ where k is the number of inversions.

There is also another correct way of solving this problem by *merge sort* in which you find (at most) 11 sorted sublist and merge them with another sorted list to sort the whole A.

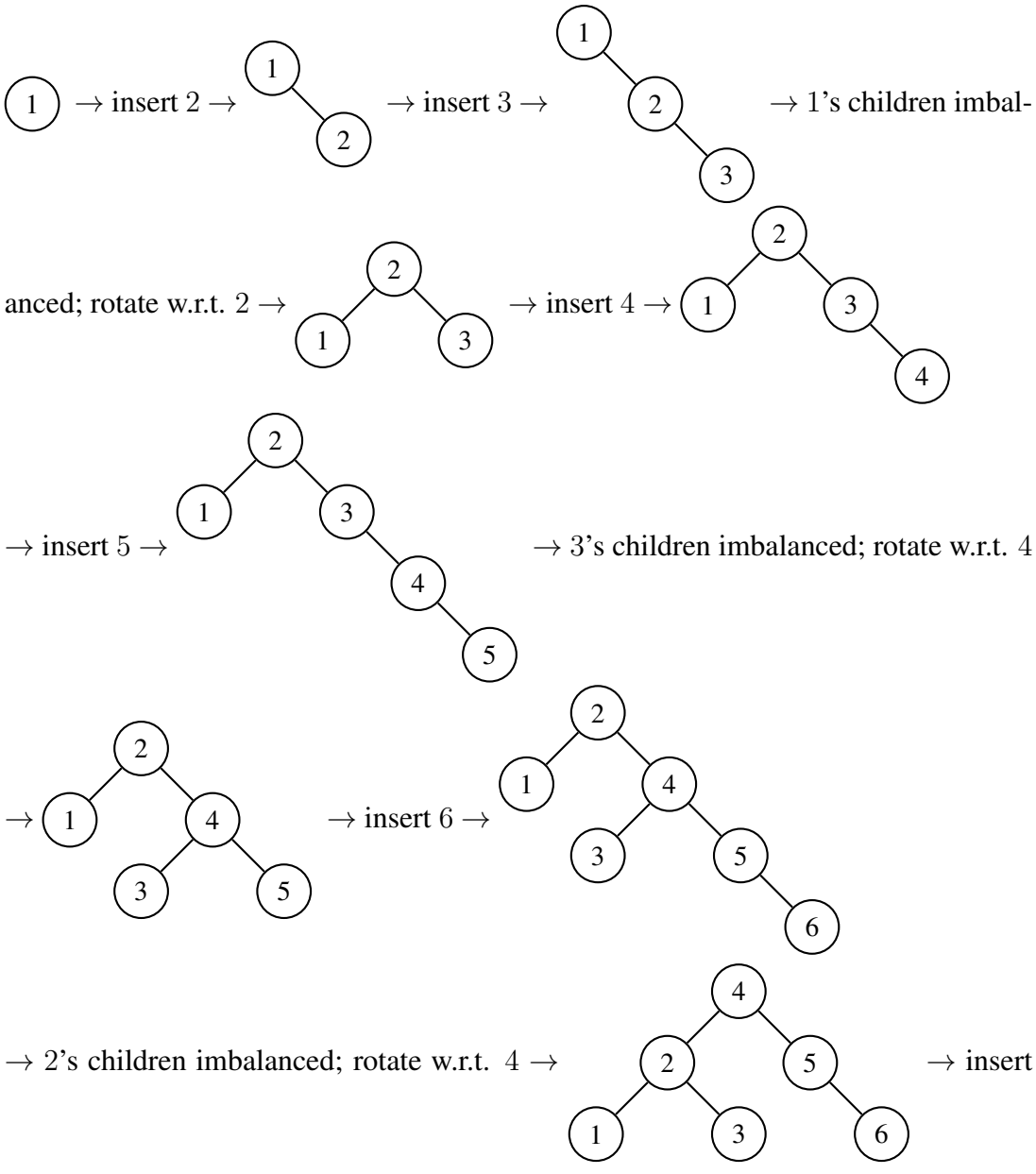
Common mistake. Note that checking whether an item is hacked or not cannot be done by just looking at its neighbor. One of the common (wrong) solutions was to iterate over the list and remove all indices i such that $A[i] \geq A[i+1]$ or $A[i] \leq A[i-1]$ (or some other similar conditions). The the claim was that the remaining items are sorted which is not correct. This approach fails if for instance all hacked items are neighbors and remains in the sorted order.Consider the following example:

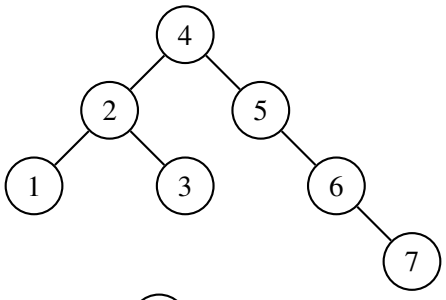
$4, 7, 9, \mathbf{100}, \mathbf{101}, \mathbf{102}, \mathbf{103}, \mathbf{104}, \mathbf{105}, \mathbf{106}, \mathbf{107}, \mathbf{108}, \mathbf{109}, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40$

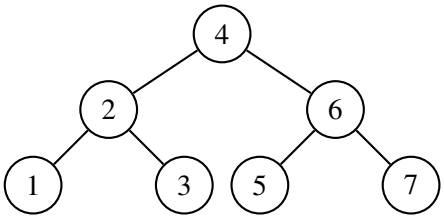
Problem 8. [26 points] **Se7en Keys** (3 parts)

(a) [10 points] If the numbers 1, 2, 3, 4, 5, 6, and 7 are inserted, in that order, into an initially empty AVL tree, how many total rotations are performed? To ensure partial credit, show your work, circling the number of rotations performed by each insertion. Draw two circles around the total number of rotations.

Solution: The total number of rotations necessary was 4:

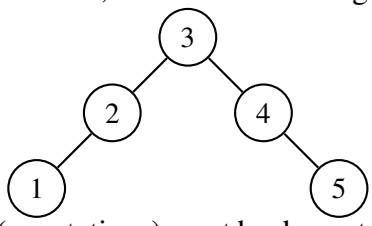


$7 \rightarrow$  $\rightarrow$ 5's children imbalanced; rotate w.r.t. $6 \rightarrow$



In total, this process required 4 left rotations.

Common Mistakes: One common mistake was to return the tree before the final rotation was completed, meaning that the 5 was left imbalanced.

Another common mistake was in the second rotation, rather than rotating around 4, the tree would be rotated around 2, leading to  , which is incorrect because after an insertion, a rotation (or rotations) must be done at the lowest imbalanced node(s) in the tree — in this case, the children of 3 are imbalanced, so we should've left-rotated around 4.

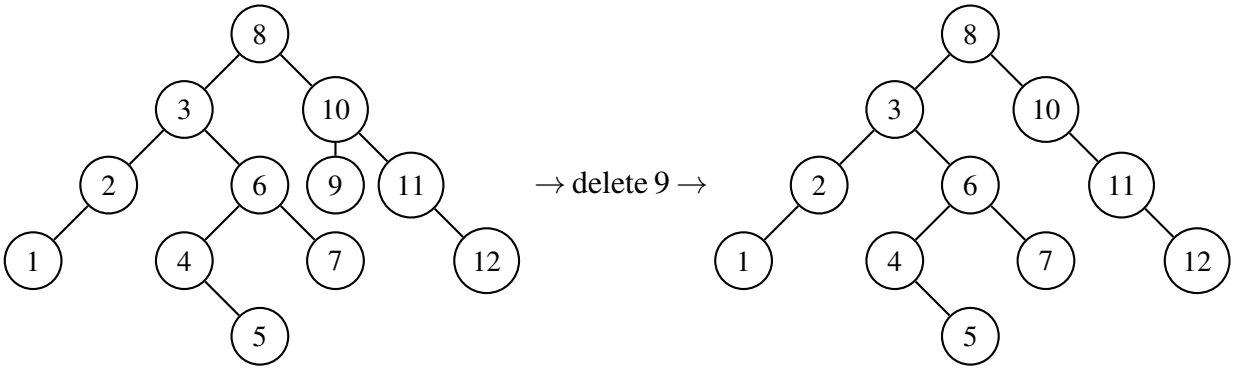
One other mistake was inserting all the nodes first, and then rotating them into a balanced tree. A core component of AVL trees is that they rotate/rebalance on every insertion, rather than all at once at some pre-specified time.

(b) [6 points] In what order can you insert the numbers 1, 2, 3, 4, 5, 6, and 7 into an initially empty AVL tree to minimize the total number of rotations performed? **Solution:**

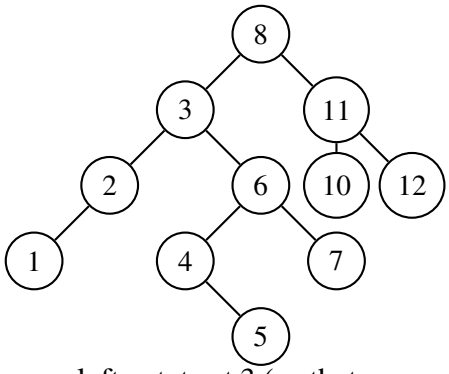
All correct sequences were of the following format: the first insertion was 4, followed by (in any order) $\{2, 6\}$, followed by (in any order) $\{1, 3, 5, 7\}$.

(c) [10 points] Give an AVL tree and an element in it such that deleting that element causes at least 3 rotations.

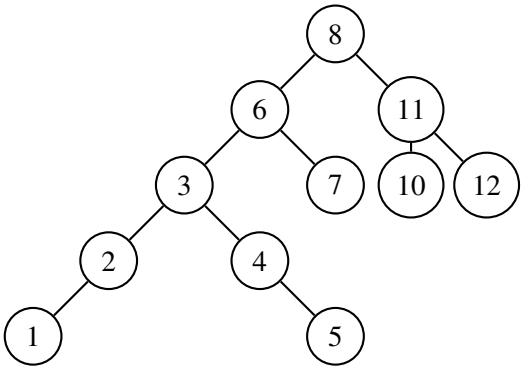
Solution: The following is one possible solution:



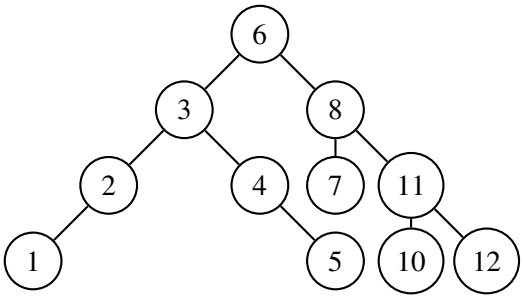
→ 10 is imbalanced, so left-rotate →



→ 8 is imbalanced, and its left child is right-heavy, so left-rotate at 3 (so that we can right-rotate at 8) →



→ right-rotate at 8 →



Another common solution is a node A of height 6 with a child of of height 5 (and as small a a subtree as possible) and a child B of height 4, which has a child of of height 3 (and as small a a subtree as possible) and a child C of height 2, which has a child

of of height 1 (and as small a a subtree as possible) and a child D of height 0. Then deleting D causes an imbalance at C; fixing that imbalance reduces the height of C to 1, causing an imbalance at B, and fixing that imbalance reduces the height of B to 3, causing an imbalance at A, which requires a total of at least 3 rotations.

Common Mistakes: One common mistake was in BST deletion. Before an AVL tree rebalances (by rotations) after a deletion, the BST deletion must be finished: if the deleted node had one child, it merges with that child; if it had two children, it swaps with its successor in the tree and then is deleted. This is a BST, not AVL, process, and takes no rotations.

Another common mistake was to forget the BST property: each node is less than everything in its left subtree, not just its left child, and greater than everything in its right subtree, not just its right child.

Another common mistake was to rotate at the wrong node. The node at which rotation happens is always the lowest imbalanced node.

Problem 9. [15 points] **Smooth Operator** (1 part)

You have an array $A[0\,..\,n-1]$ of integers with the guarantee that each value $A[i]$ is within ± 1 of the previous value $A[i-1]$, that is, $|A[i] - A[i-1]| \leq 1$ for all $0 < i < n$.

Give an $O(\log n)$-time algorithm to search in A for a given integer x with the property that $A[0] \leq x \leq A[n-1]$. Your algorithm should **always** return an index j such that $A[j] = x$.

Solution: We will use binary search on the array A. Because neighboring entries of A are either equal or consecutive integers, if $A[k] \leq A[l]$ (resp. $A[k] > A[l]$) then for every $0 \leq k \leq l \leq n-1$ the entries of A between $A[k]$ and $A[l]$ should contain all the integers in the set $\{i | A[k] \leq i \leq A[l]\}$ (resp. $\{i | A[l] \leq i \leq A[k]\}$). Thus, if $x < A[\lfloor n/2 \rfloor] (\Rightarrow A[0] < A[\lfloor n/2 \rfloor])$, then x should be in the left half of the array. if $x > A[\lfloor n/2 \rfloor] (\Rightarrow A[\lfloor n/2 \rfloor] > A[n-1])$, then x should be in the right half of the array. If $x = A[\lfloor n/2 \rfloor]$ we are done.

The running time is given by the recurrence: $T(n) = T(n/2) + O(1)$ for which case 2 of the master theorem applies and gives $T(n) = O(\log n)$.

Common mistake: Many students misunderstood the problem statement and wrote an algorithm for a slightly different problem (e.g how to find a value between $A[0]$ and $A[n-1]$).