Quiz 1 Solutions

Problem 1. True or False [21 points] (7 parts)

For each of the following questions, circle either T (True) or F (False). **Explain your choice.** (Your explanation is worth more than your choice of true or false.)

(a) T F The height of any binary search tree with n nodes is $O(\log n)$.
Explain:

Solution: False. In the best case, the height of a BST is $O(\log n)$ if it is balanced. In the worst case, however, it can be $\Theta(n)$.

(b) T F Inserting into an AVL tree with n nodes requires $\Theta(\log n)$ rotations.
Explain:

Solution: False. There were two ways you can show this.

1. There are cases where inserting into an AVL tree requires no rotations. $\Theta(\log n)$ rotations implies $\Omega(\log n)$ rotations. Since we have insertions that require no rotations, this means that inserting into an AVL tree does not require $\Omega(\log n)$ rotations and thus it does not require $\Theta(\log n)$ rotations.

2. Inserting into an AVL tree may look at $O(\log n)$ nodes, but it only needs to perform at most 2 rotations to fix the imbalance. Thus inserting into an AVL tree requires $O(1)$ rotations, which is not $\Theta(\log n)$.

Both of these were acceptable.

Common mistakes included thinking that rotations needed to be made for each node in an inserted node's ancestry line and misunderstanding the problem to think that we were asking for the runtime of insertion and not the number of rotations required.

(c) T F The depths of any two leaves in a max heap differ by at most 1.
Explain:

Solution: True. A heap is derived from an array and new levels to a heap are only added once the leaf level is already full. As a result, a heap's leaves are only found in the bottom two levels of the heap and thus the maximum difference between any two leaves' depths is 1.

A common mistake was pointing out that a heap could be arbitrarily shaped as long as the heap property (parent greater than its children in the case of a max-heap) was maintained. This heap is not a valid heap, as there would be gaps if we tried to express it in array form, heap operations would no longer have $O(\log n)$ running time, and heap sort would fail when using this heap.

Another common mistake was simply justifying this statement by saying a heap is balanced. An AVL tree is also balanced, but it does not have the property that any two leaves have depths that differ by at most 1.

(d) T F A tree with n nodes and the property that the heights of the two children of any node differ by at most 2 has $O(\log n)$ height.
Explain:

Solution: True. Using the same approach as proving AVL trees have $O(\log n)$ height, we say that n_h is the minimum number of elements in such a tree of height h.

$$\begin{aligned}
n_h &\geq 1 + n_{h-1} + n_{h-3} && (1)\\
n_h &> 2n_{h-3} && (2)\\
n_h &> 2^{h/3} && (3)\\
h &< 3\lg n_h && (4)\\
h &= O(\log n) && (5)
\end{aligned}$$

Grading was fairly tight on this one. Most of the answers that got full credit were the ones that was able to show the reduction above or something similar. Many answers did not have enough justification, though stated true statements (e.g. "if the heights of every node's two children differ by at most some constant c, the tree will have height $O(\log n)$", true but we're looking for why exactly). Some got to the right conclusion with an alternate method but had some logical flaws.

A common mistake was providing a counter example where the height was greater than $\log n$. This is not a valid counter example since that's not what $O(\log n)$ height means. $h = O(\log n)$ is comparing the asymptotic relationship between the height and the number of elements in the tree, it's not saying $h < \log n$ for all n.

(e) T F For any constants $x, y > 1$, we have $n^x = O(y^n)$.
Explain:

Solution: True. Exponential growth always dominates polynomial growth. To show more rigorously, we want to show that $\frac{n^x}{y^n} = 0$ as n goes to infinity. For large enough n, we have

$$0 < \frac{\log n}{n} < \frac{\log y}{x+1}$$

$$(x+1)\log n < n \log y$$

$$n^{x+1} < y^x$$

$$\frac{n^x}{y^n} < \frac{1}{n}$$

Since $\frac{1}{n} = 0$ as n goes to infinity, this shows that $\frac{n^x}{y^n} = 0$ for the same limit and thus $n^x = O(y^n)$. This proof was not necessary for full credit.

(f) T F Let $|U| = m^2$ and consider hashing with chaining. For any hash function $h : U \to \{1, 2, \ldots, m-1\}$, there exists a sequence of m insertions that leads to a chain of length m.
Explain:

Solution: True. Consider any h. By pigeonhole principle, there exists at least one bucket $j \in \{1, 2, \ldots, m-1\}$ in the hash array such that there is a set S of $m^2/(m-1) > m$ universe elements $i\imath U$ such that $h(i) = j$ for all $i \in S$. Inserting all elements in S creates a chain of sufficient length.

Most answers were correct. Some incorrect arguments assumed that h is a random function (using simple uniform hashing assumption) - this assumption cannot be made here because we are dealing with *any* function h. A few other answers suggested using perfect hashing to show the answer is False. This does not work: a perfect hash function is constructed for a specific set S, while in the question we are given h upfront, and want to construct a "bad" set for it.

(g) T F Five elements can always be sorted with at most seven comparisons in the comparison model.
Explain:

Solution: This question turned out to be "unfortunate". As such, we decided to award 3 points to *any* answer.

The answer to the question is True - there *is* a way to sort 5 elements using 7 comparison. Unfortunately, the answer is tricky, and *no one* got it right.

The answer is yes but in order to justify this you would need to give an actual algorithm (or give the decision tree) for sorting any 5 elements in 7 comparisons. Here is very clever way to do this[1]:

Compare A to B and C to D. Without loss of generality (WLOG), suppose $A > B$ and $C > D$. Compare A to C. WLOG, suppose $A > C$. Sort E into A-C-D. This can be done with two comparisons. Sort B into $\{E, C, D\}$. This can be done with two comparisons, for a total of seven.

On the other hand, there was a great variety of incorrect solutions to this questions. For clarity, we partition them into two categories: *incorrect* solutions yielding a *correct* answer, and *incorrect* solutions yielding an *incorrect* answer.

In the first category, the dominant argument was to compare $2^7 = 128$ (the maximum number of leaves in a decision tree of height 7) to $5! = 120$ (the number of different orderings of 5 elements), and argue that since $128 > 120$ the algorithm must exist. Unfortunately, this *only* shows that the *lower bound argument from the lecture does not apply*, not that there is an algorithm. E.g., for this argument to hold, one would have to argue that the decision tree can be always made (almost) perfectly balanced, which might not be possible in general. In particular, the optimal number of comparisons to sort a given number n of elements is not known for general values of n. A few other answers provided an algorithm, but underestimated the number of comparisons used.

In the second category, most answers compared $5 \lg 5 \approx 11.6$ to 7, and argued that since $11.6 > 7$ the algorithm cannot exist. Unfortunately, the lower bound presented in the lecture was $\lg(n!)$ not $n \lg n$. In our case this yields $\lg(120) \approx 6.9$, which is smaller than 7. Note that even though $n \lg n = \Theta(\lg(n!))$, in this question we are dealing with a *fixed* value of $n = 5$, so the asymptotic arguments cannot be used here - the constant does matter.

Other arguments assumed a *concrete* sorting algorithm and showed that it makes more than 7 comparisons. Not surprisingly, insertion sort and other slow algorithms were quite popular in this line of work. Unfortunately, this only shows that *a particular* sorting algorithm makes more than 7 comparisons, not that this is the case for *all* algorithms.

[1]http://stackoverflow.com/questions/1534748/design-an-efficient-algorithm-to-sort-5-distinct-keys-in-fewer-than-8-comparisons

All in all, the question was “unfortunate”. Apologies. On the bright side, if you read and understood the last few paragraphs, you can consider yourself well-prepared for the final.

Problem 2. Short Answer [40 points] (8 parts)

(a) What is the running time of these algorithms on a sorted list?

I. Insertion sort

Solution: The running time is $\Theta(n)$. Insertion sort iterates over the list and, for each element, swaps the element backwards until it's in the correct position in the sorted subarray. Thus, for a sorted list, each element will be swapped 0 times and each step will take $O(1)$ time for a grand total of $\Theta(n)$.
In grading, common errors were misreading that the input was a "sorted list" or misunderstanding that the insertion sort algorithm swaps backwards. Answers of $\Theta(n)$ were given full credit (no justification was needed although incorrect reasoning would incur a deduction). Answers of $O(n^2)$ were docked a couple of points since this was at least a correct answer although not a tight bound and indicated a lack of fully understanding the question. Any other answers were not given any credit.

II. Merge sort

Solution: Merge sort has the same runtime no matter what which is $O(n \log n)$. See lecture notes for a derivation of the recurrence. Almost everyone got this correct.

(b) Solve these recurrences:

I. $T(n) = 4T(n/2) + \Theta(n^2)$

Solution: This is case 2 of the master method and thus $\Theta(n^2 \log n)$.

II. $T(n) = T(4n/5) + \Theta(n)$

Solution: This is case 3 of the master method and thus $\Theta(n)$.

(c) How does the key in a node compare to the keys of its children in ...

I. ... a binary search tree?

Solution:

$$\text{node.left.key} < \text{node.key} < \text{node.right.key}$$

II. ... a max heap?

Solution:

$$\text{node.key} > \text{node.left.key}, \text{node.right.key}$$

(d) Suppose that the universe U of possible keys is $\{0, 1, \ldots, n^2 - 1\}$. For a hash table of size n, what is the greatest number of distinct keys the table can hold with each of these collision resolution strategies?

I. Chaining

Solution: With chaining, we can hold as many keys as there are in the universe which, in this case, is n^2.
Common errors were miscounting the size of the universe to be $n^2 - 1$ or saying that the table could hold an infinite amount of elements.

II. Linear probing

Solution: Linear probing can only store as many elements as the size of the table which is n.

III. Quadratic probing

Solution: Similarly to linear probing, quadratic probing can only store as many elements as the size of the table which is n.

(e) What order should we insert the elements $\{1, 2, \ldots, 7\}$ into an empty AVL tree so that we don't have to perform any rotations on it?

Solution: You should insert in the order $\{4, 2, 6, 1, 3, 5, 7\}$ to make an AVL tree. The ordering of $\{2, 6\}$ and the ordering of $\{1, 3, 5, 7\}$ do not matter. One can see the resulting binary search tree is perfectly balance therefore an AVL tree. One point is taken off if the student did not explain why the resulting BST is an AVL tree (balanced, or left and right depth differ by 0). A common mistake is that people gave an output of a max heap, which is completely different from BST.

(f) Suppose you insert three keys into a hash table with m slots. Assuming the simple uniform hashing assumption, and given that collisions are resolved by chaining, what is the probability that both slots 0 and 1 are empty?

Solution: Under SUHA, each key is independent of others and have equal probability inserting into each of the m slots. So for each key the chance of not inserting into slot 0 or 1 is $\frac{m-2}{m}$. And because each key is independent, the total probability is $(\frac{m-2}{m})^3$. A common mistake is that students think the probability of not inserting into slot 1 is independent of not inserting into slot 0, which is not true. Given a key did not end up in slot 0, the chance that it will also not end up in slot 1 is $\frac{m-2}{m-1}$.

(g) Rank the following functions by increasing order of growth; that is, find an arrangement g_1, g_2, g_3, g_4 of the functions satisfying $g_1 = O(g_2)$, $g_2 = O(g_3)$, $g_3 = O(g_4)$. (For example, the correct ordering of n^2, n^4, n, n^3 is n, n^2, n^3, n^4.)

$$f_1 = n^{\log n} \qquad f_2 = \sqrt{n} \qquad f_3 = n^{3+\sin(n)} \qquad f_4 = \log n^n$$

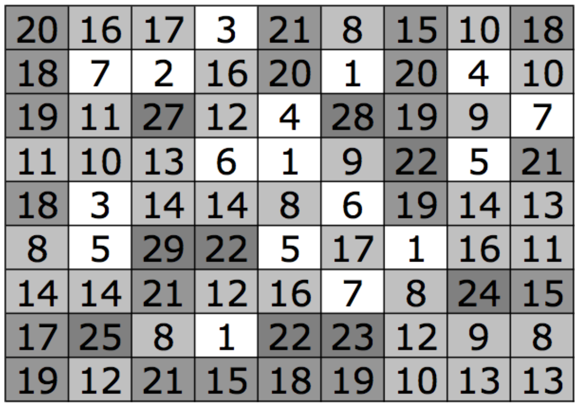
Solution: We see $\log n^n = n \log n$. $n^2 \le n^{3+\sin(n)} \le n^4$ because $\sin(n)$ is between -1 and 1 for all n. Also $n^{\log n} > n^c$ for all constant c. So the correct order is f_2, f_4, f_3, f_1. One common mistake is that students do not realize $n^{1/2} < n$.

(h) Explain why, when resolving hash-table collisions via linear probing, one cannot remove an entry from the hash table by resetting the slot to NIL.

Solution: When one tries to look up a key in the hash table, it will return NIL when it sees an empty slot and therefore stop the lookup. So if one deletes a slot by resetting the slot to NIL, it will break the chain and one may not be able to find items that were inserted after the key in the deleted slot. Full credit is given for the correct explanation or a correct example.

Problem 3. You Are The Computer [20 points] (4 parts)

(a) Find the peak that our $O(n)$-time 2D peak finding algorithm returns when used on the following matrix:



20	16	17	3	21	8	15	10	18
18	7	2	16	20	1	20	4	10
19	11	27	12	4	28	19	9	7
11	10	13	6	1	9	22	5	21
18	3	14	14	8	6	19	14	13
8	5	29	22	5	17	1	16	11
14	14	21	12	16	7	8	24	15
17	25	8	1	22	23	12	9	8
19	12	21	15	18	19	10	13	13

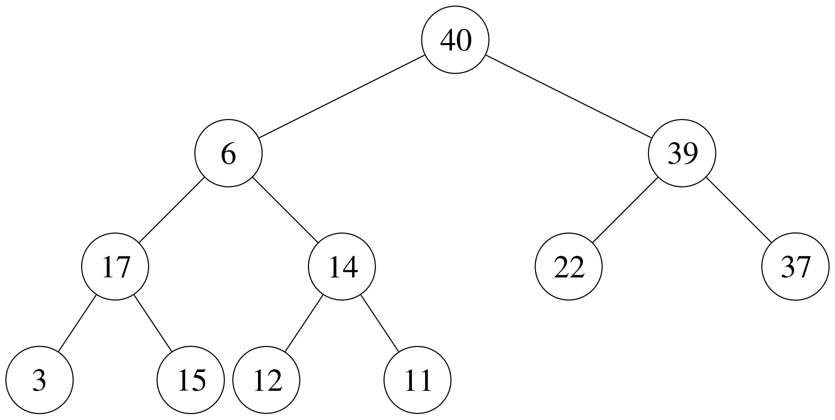
Solution: Our peak finding algorithm will find the 24 element in the bottom right as a peak. The maximum of the first window-frame is 22, in the eighth row and fifth column. It's not a peak, since 23 to its right is larger. The algorithm recurses on the bottom right 3 by 3 subgrid:

17	1	16
7	8	24
23	12	9

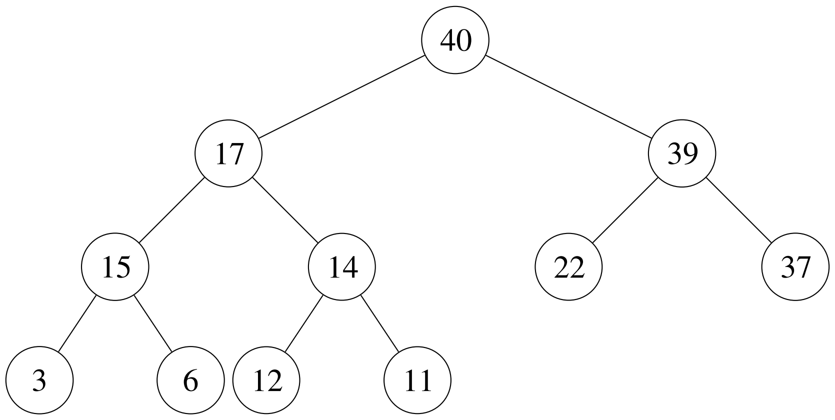
The window-frame of this subgrid is the entire subgrid. Its maximum is 24, which is a peak.

One common mistake on this problem was checking whether every element of the window-frame was a peak. Our 2D peak finding algorithm only checks whether the *maximum* on the window-frame is a peak; if it checked every element of the window-frame, then recursive calls to the peak finding algorithm might return something that is locally a peak but not globally a peak. The recitation notes have a more detailed justification.

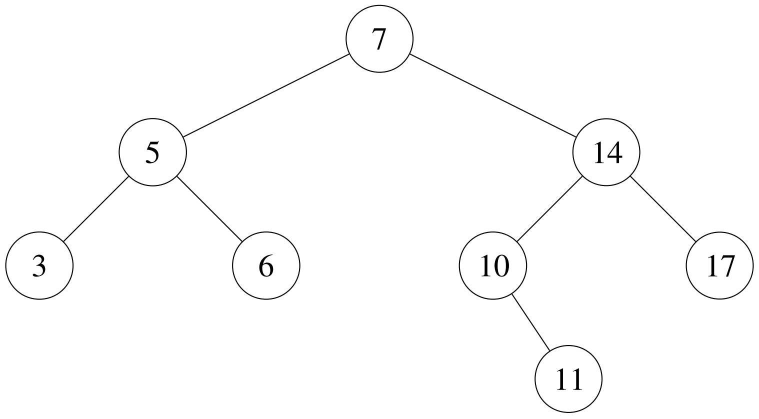
(b) What is the max-heap resulting from performing on the node storing 6?



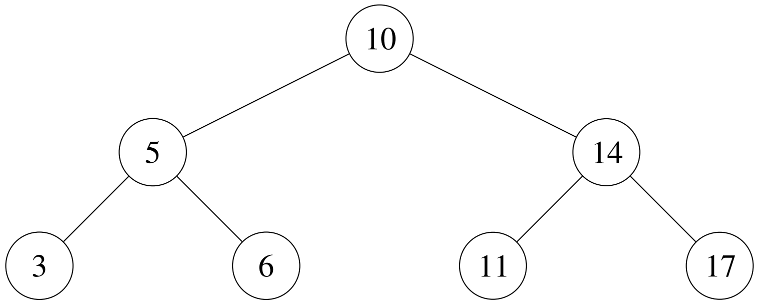
Solution: will swap 6 with its larger child until 6 reaches a position where it satisfies the max-heap property. The correct heap is:



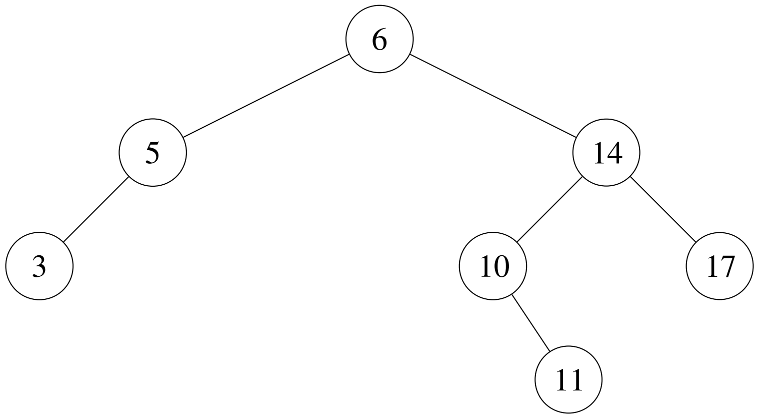
(c) What binary search tree is obtained after the root of this tree is deleted?



Solution: The correct algorithm replaces 7 with the successor (next-largest) node of 7, which is 10. The right subtree of 10 is moved to the location 10 was originally in.

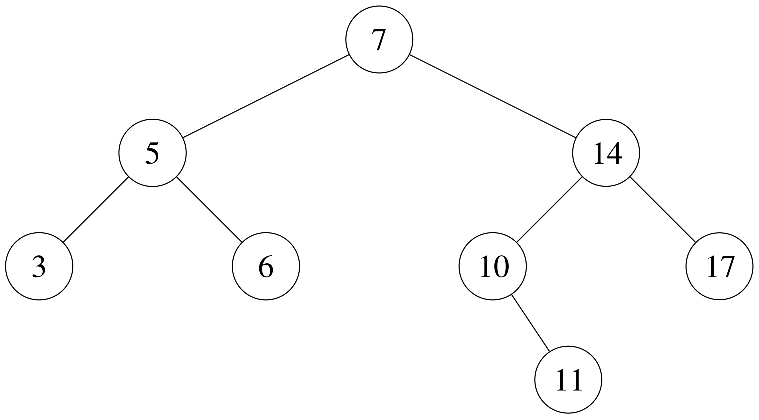


Alternatively, you can replace 7 with its predecessor 6. (If 6 had a left subtree, it would be moved to where 6 was originally.)

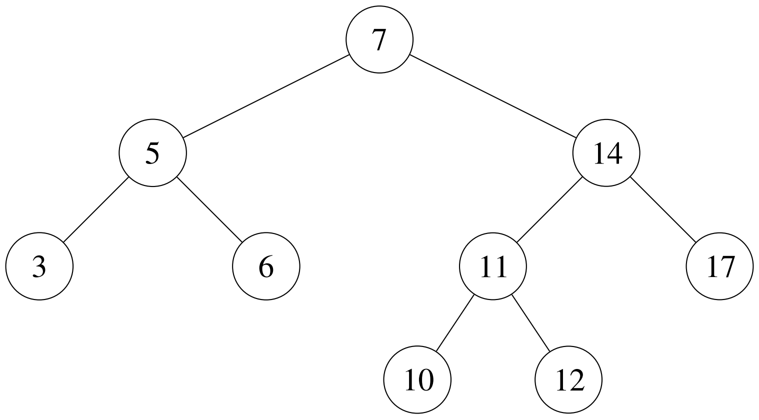


A common mistake on this problem was to use something resembling , recusively moving the larger child up to replace the deleted parent. This does not result in a valid BST.

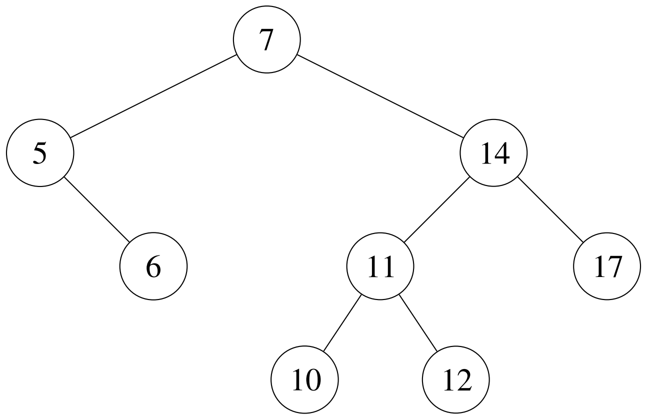
(d) What does the following AVL tree look like after we perform the operations and (in that order)?



Solution: The 12 is initially inserted as a right child of 11. This breaks the AVL property for node 10, however, since its right subtree has a depth 2 greater than its left subtree. We need to left-rotate node 10 to fix the tree.



Next, we delete node 3. After this delete, the AVL property still holds for all nodes, so we don't need to do any more rotations.



The most common mistakes on this problem were incorrect or unnecessary rotations.

Problem 4. 3D Peak Finding [10 points]

Consider a 3D matrix B of integers of size $n \times n \times n$. Define the ***neighborhood*** of an element $x = B[i][j][k]$ to consist of

$$\begin{array}{lll} B[i+1][j][k], & B[i][j+1][k], & B[i][j][k+1], \\ B[i-1][j][k], & B[i][j-1][k], & B[i][j][k-1], \end{array}$$

that is, the six elements bordering the one in question, not including diagonals. (For elements on a face we consider only five neighbors; for elements on an edge we consider only four neighbors; and for the eight corner elements we consider only three.) An element x is a ***peak*** of B if it is greater than or equal to all of its neighbors. Consider this algorithm for finding a peak in matrix B:

1. Consider the nine planes $i = 0$, $i = \frac{n}{2}$, $i = n-1$, $j = 0$, $j = \frac{n}{2}$, $j = n-1$, $k = 0$, $k = \frac{n}{2}$, $k = n-1$, which divide the matrix into eight submatrices.
2. Find the maximum element m on any of the nine planes.
3. If m is a peak, return it.
4. Otherwise, m has a larger neighbor element. Recurse into the submatrix that contains that larger neighbor element (including elements in bordering planes).

What is the running time of this 3D peak finding algorithm?

Solution: Size of original problem of finding a peak in a $n \times n \times n$ 3D matrix is $T(n)$. Time taken to divide is finding the maximum element of nine $n \times n$ planes, which takes $O(n^2)$. Size of subproblem to recurse to is $T(n/2)$, and there is only one subproblem. Recurrence is $T(n) = T(n/2) + O(n^2)$, which is $O(n^2)$ by Master Theorem.

Each of the following were awarded 2 points:

1. Correct size of the subproblem ($n/2$). A common mistake was to think that subproblem sizes were $n/8$. The reason for the confusion was that the total volume is indeed $1/8$th, as $(n/2)^3 = n^3/8$.
2. Correct number of subproblems (which is 1). A common mistake was to think that there were 8 subproblems, but in fact, we recurse on only one of the subcubes.
3. Correct time for max finding ($f(n) = O(n^2)$. A common mistake was to think that this could be done with 2D peak finding, but in fact we're looking for the maximum, not the peak, of each array, hence we need to examine all n^2 elements).
4. Correct case of the master theorem (case 3) and explanation of why this case applies.
5. Correct final runtime ($O(n^2)$). If this number appeared magically with very little or no explanation, a maximum of 4 points were awarded, depending on the strength of the explanation, as all the above points were missing.

Problem 5. Augmenting AVL Trees [15 points]

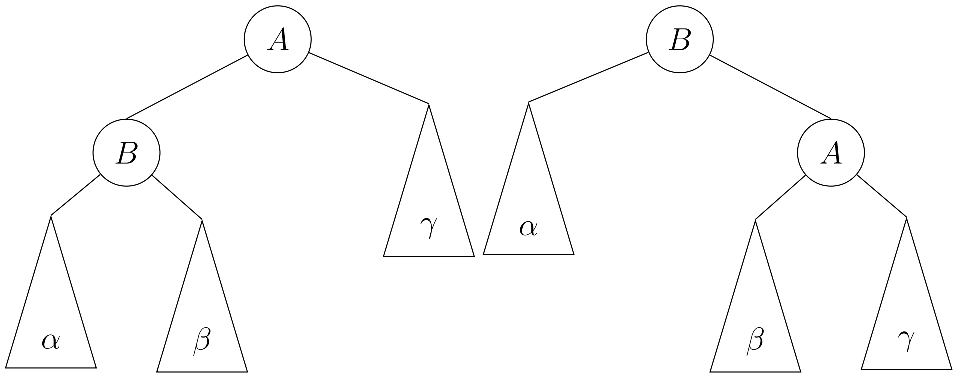
In Problem Set 1, we saw how to augment an AVL tree so that we can quickly compute the function which returns the number of items in the range from a to b in tree T. We did so by maintaining a $size$ field in every node that stored the number of nodes in the subtree rooted at that node, including the node itself.

Describe how to augment an AVL tree to quickly compute which returns the average of all the values of the nodes in the subtree rooted at node x in the tree T. Describe the augmentation and show how to update the tree in $O(1)$ time when a rotation is being performed on a tree (which might happen after an insert or delete to keep the tree balanced).

Solution: We will begin by augmenting the AVL tree by storing $size$ (defined as in PS1) and sum at each node. The sum field at a node will contain the sum of the values of the nodes in the subtree rooted at that node. Remember, as always, that the "subtree rooted at x" includes x. Some solutions didn't include x, which doesn't automatically make the solution incorrect but made it easier to make errors elsewhere. Some solutions also assumed a $size$ field without augmenting each node with one; remember that AVL trees only store $height$ (for balancing purposes). Some solutions also stored the average at each node instead of the sum; this is not necessarily incorrect, but these solutions universally frequently recomputed the sum by multiplying the average by the size.

We can update both sum and $size$ fields as we walk down the tree during insertions and deletions without increasing the asymptotic time complexity of those operations. You might also observe that, for a leaf node x, $x.size = 1$ and $x.sum = x.value$. (You were not required to discuss any of these details.)

The average function can then be computed for node x in $O(1)$ time by dividing $x.sum$ by $x.size$.



Now let's examine what happens during a rotation that transforms the tree shown on the left into the one shown on the right. A complete solution notes that only the augmented values at A and B need to be recomputed; since the membership of the subtree shown does not change (despite being rearranged), no ancestors are affected, and since none of the subtrees α, β, γ are modified, none of their augmented values are affected either.

This can be done in one of two ways; first, you can simply apply the definitions of $size$ and of sum to both A and B; second, you can account for the nodes that are entering and leaving the subtrees rooted at each of A and B.

If you chose to recompute the augmented values, you should have noted that $A.size = 1+\beta.size+\gamma.size$ and $A.sum = A.value + \beta.sum + \gamma.value$ needed to be computed before $B.size = 1 + \alpha.size + A.size$ and $B.sum = B.value + \alpha.sum + A.sum$.

If you chose the second strategy, then you should have noted that B and the nodes in α left the subtree rooted at A and that A and the nodes in γ entered the subtree rooted at B. Thus the proper update operations were $A.size = A.size-(1+\alpha.size)$, $A.sum = A.sum-(B.value+\alpha.sum)$, $B.size = B.size + 1 + \gamma.size$, and $B.sum = B.sum + A.value + \gamma.size$. (Note that neither depends on the augmented values of the other node being updated, so order does *not* matter.)

Points were given for describing augmentations, describing the implementation of the average function, discussing how the rotation operation could be completed in $O(1)$ time (e.g., by noting that only A and B needed to be updated), and describing clearly, correctly, and concisely how the augmentations described would need to be updated.

Problem 6. Word Search [14 points]

A ***word search puzzle*** consists of an $n \times n$ grid of characters and a "word bank" containing m words each length l. Thus, the total size of the input is $\Theta(n^2 + ml)$. Each of the m words in the word bank is hidden in the grid, either horizontally or vertically (not diagonally in this problem). Note that the words may be hidden in reverse, as seen in the case of SORT and DICT in the example below.

Word Search Example ($n = 8, m = 6, l = 4$)

H	T	A	Q	**A**	W	O	**T**
A	R	R	L	**L**	C	I	**C**
S	**M**	D	O	**G**	R	S	**I**
H	**A**	I	G	**O**	A	H	**D**
X	**T**	B	T	I	E	H	D
N	**H**	D	M	**H**	**E**	**A**	**P**
U	K	Y	O	O	C	C	W
X	Q	**T**	**R**	**O**	**S**	D	K

Word bank: ALGO, DICT, HASH, HEAP, MATH, SORT

Describe an algorithm that finds all the words in the word bank in the puzzle (i.e., returns the coordinates of the first letter of each word) and analyze its runtime in terms of n, m, and l. Full credit will go to algorithms with the optimal runtime.

Solution: First hash each of the m words of length l using a rolling-hash-friendly hash function, and put them into a hash table H of size $\Theta(n^2)$, say using chaining for collision resolution. Also hash the reverse of all the words and insert them into H. This part takes $O(ml)$ time, because we spend $O(l)$ time to compute the hash value of each word plus $O(1)$ time to insert it into H (assuming simple uniform hashing).

Now use a rolling hash on each of the n rows and n columns. Each row and column produces $n - l + 1$ hashes of substrings of length l, using just $O(n)$ time because the hash can be rolled in $O(1)$ time per character. Overall, because there are n rows and n columns, this work takes $O(n^2)$ time. If one of these substring hashes appears in the table H, then we need to test whether the substring and word are actually the same, and did not just happen to have the same hash value. (A common error was to skip this "double checking".) This check takes $O(l)$ time per collision (and possible actual match), but the probability that a substring–word pair hash to the same slot is $O(1/(m + n))$, and there are $O(n^2 m)$ such pairs, so the total cost for such checks is $O(n^2 ml/n^2) = O(ml)$. (Almost no one did this analysis, but it is necessary for a completely correct solution.)

The total time for this algorithm is $O(ml + n^2)$, which is optimal because we at least need to read the input to solve this problem.

Grading scheme: The most common mistake, not double-checking that hash collisions were actually matches, reduced an otherwise correct solution by 2 points. Forgetting to check the reverses of strings (or rows and columns) also reduced by 2 points. A correct but suboptimal algorithm (e.g., checking each word against each possible location) received a base score of 7. Incorrect solutions generally received a score of 0, except that mentioning "rolling hashing" could result in up to 3 points.