Midterm Solutions Discussion

Problem 6:

The goal of this problem was to look at augmenting a data structure, and in particular, to leverage the ability of a tree to efficiently summarize the data in a subtree.

There were essentially four operations to support: insert user, add/delete a skill, search for a user, and search for a super user. If you do not care about implementing searchSuper efficiently, then you actually do not need to do almost anything. You can solve the problem using any dictionary, where the key is the user name, and the value is an array of skills. (Your solution to that problem could have been basically two sentences.)

So the entire challenge here was to ensure that all the operations, including searchSuper, were efficient. By efficient, here, we mean O(log n) or better. (A linear time O(n) solution for searchSuper is so much slower that it overwhelms any other benefits.)

Note that this is typical of data structure problems. If you need 4 operations, it is often easy to get any subset of 3 of them to be efficient, but the algorithmic challenge is to make them all efficient!

So as a grading note, a solution that had the potential to implement all operations efficiently could achieve 16 points. But a solution that fundamentally could not implement all operations efficiently could achieve at most 8 points. So for example, an unaugmented AVL tree or a standard hash table would get at most 8 points.

Some general issues:

- The goal of the question was to explain the augmentation. It explicitly said you did not need to repeat the algorithms from class. And yet almost every solution spent a lot of space explaining how, say, a binary search tree worked. You did not need to tell me how to search a binary search tree, or how to implement chaining.
- At the same time, solutions were often very vague about the parts that were really different. For example, comments like, "the max variable is updated accordingly." According to what? Or that something is updated "automatically." There is no automatic: you are supposed to give the algorithm! Or "the tree is updated as needed." When is it needed? Which nodes? What sort of update? And the brief comment was often stuffed into a small corner as a footnote to a long description of how to search a tree.

  I really wanted to see at least a clear specification of which nodes were being updated, when they were being udpated, and how exactly to determine the new value. (There were lots of bugs for the deleteSkill case in that!)
- So the moral is for a question *like this one*, spend less time repeating what I said in class and be much more detailed/careful about your solution, i.e., what differs from the basic structure.
- One part asked for a picture, and many pictures were good. But some solutions gave pictures that didn't explain much, e.g., a tree with one node, or a tree with no example labels or values, etc. The goal of the picture is to make your idea easier to understand.
- An array and an arraylist are not the same thing. An array is a relatively abstract concept, while an arraylist is a specific Java class. When describing algorithms, try to use abstract types (e.g., array) rather than Java classes (e.g., arraylists). The reason for this is that there is lots of functionality hidden inside an arraylist that might obscure what is really happening. (Thing of the cost of appending to a string in Java.). The exception to this rule, of course, is if you are actually giving Java code for your solution. But if you are giving pseudocode (or an English paragraph), don't use Java.
- Many people seemed to give running times of O(k log(n)) instead of O(k + log(n)). If you think the running time is O(k log(n)), that means that you are spending k time on every node on the search path on the tree! That is rarely what you mean. (Usually, you are only spending k time on the node for the user in question).
- When giving running times, be aware of what the running time is a function of. Is this a function of n or k or something else altogether? Is it expected or with high probability or worst-case? And what is the question asking for? Be careful to answer what the question asks.

A quick overview of some solutions that I saw:

- *Augmented AVL tree summarizing subtrees*: The most common correct answer stored in every node some summary, whether the number of super-virus-fighters in the subtree, or a boolean indicating if there were any super-virus-fighters in the subtree, or even the name of a single super-virus-fighter in the subtree.

  The most common mistakes were not maintaining that augmented data. For example, a solution might forget to update the augmented data on addSkill/deleteSkill, or might not explain sufficiently clearly how the augmented data should be maintained.

Or a solution might forget to maintain the augmented data on a rotation. On an insert, rotations occur, and you have to update two nodes involved in each rotation. These nodes are NOT necessarily on a leaf-to-root path (though one of them is) from the inserted node, so it is not sufficient just to update the path.

Some solutions had a reasonable augmentation, but then didn't seem to understand how to implement searchSuper, recursing down both sides of a node. (To get efficient performance, it is critical to only recurse down one child so as not to explore the entire tree.) Sometimes, it simply wasn't clear exactly what was meant!

- *Augmented trie*: Some solutions used a trie instead of an AVL tree. If string comparisons were proportional to the length of the string, that would be a good idea. As is, we assumed that string comparisons had cost O(1), so there was no real advantage of using a trie.

  One possible issue was that the costs were O(L), where in the question we asked for a solution that depended on n (the number of users) and k (the number of superpowers). So a function of L was not really what we asked for. (I still counted it as correct, but it was iffy.)

  Still, a trie could be a completely correct solution if you augmented every node, as before, in a way that summarized the rest of the tree, and assumed that the number of possible characters in a name was constant (e.g., 26). Few trie solutions got this completely right, though. For example, many only augmented the nodes represented users, rather than the intermediate nodes, and many had linear time searchSuper.

- *Non-augmented AVL tree*: As discussed above, some people had an AVL tree, but no "summary" information. This led to linear time searchSuper. And with the specified augmentation, there was really no way to do better.

- *Failed augmentations for an AVL tree*: One mistake was to have data that cannot be maintained efficiently: some people tried to maintain a list in every node of ALL the super-virus-fighters in the subtree. These solutions understood the idea of trying to use a tree to summarize, but unfortunately there is no efficient way to maintain that much data in every node.

  For example, if the list of super-virus-fighters is stored as an unsorted array, then deleteSkill has to search in the unsorted list to find the user to remove the downgraded name, and that search could be Theta(n) time. And when updating nodes after a rotation, you might will have to merge to lists which (if they are arrays) will take Theta(n) time. If they are sorted, or if they are linked lists, there are similar problems.

  Another example of an improper augmentation was to store the total number of special skills in a subtree, not the total number of super-virus-fighters. In general, this information is not enough, because you cannot distinguish between a subtree where everyone has one skill (but there are no super virus fighers) and a subtree where half the users are super virus fighers and half have no skills. (If you simply recursed on the side with more skills that certainly doesn'twork.). Some tried to compare this sum to the total number of nodes in the subtree, but that also doesn't work because the distribution matters. (It does work if you ALSO maintain the number of users with at least one skill.)

- *Hash tables*: a basic hash table inevitably led to a Theta(n) searchSuper, because there is no way to quickly find the super virus fighers.

  To fix this, some solutions tried to create a special entry in the hash table that would hold all the super virus fighters. Unfortunately, this was hard to maintain: on deleteSkill you had to search the list for the user who was being downgraded. Some solutions just stored one solution in the list, and then on deleteSkill you had to find a new super user to populate the special cell. There were a variety of variants of this, none of which really worked. (Note, some of these solutions also showed up in the non-augmented AVL tree, where a list of all super virus fighers was stored at the root, say.)

  There was (at least) one hash table based solution (that I remember) that were correct. It threaded a linked list through the super-virus-fighters: each super-virus-fighter had a pointer to the next and previous super-virus-figher in the chain, wherever it was stored in the hash table. You can find a user via the hash table, and you can upgrade/downgrade users via the linked list.

  One additional issue about hash-based solutions. Many people ignored collisions altogether. And even solutions that were aware of collisions often described the running time as O(1). That's basically true, but the problem specifically called for WORST-CASE running time, and the worst-case for searching a hash table with chaining is Theta(n). That's a bit pedantic, but in general, you are advised to think about what the question is asking for. And if the question wants you to optimize for worst-case performance, then an AVL tree is a better solution.

  That said, we usually do think about a hash table as being O(1), but I always try to be clear (especially an algorithms context) that I mean O(1) expected time under (whatever) assumption on the hash function.

Problem 7:

Q7a. prefixMax(A) function

Less efficient solutions: - [ ] O(N2) solution with two nested loops.

Common mistakes: - [ ] Overwrite A[] (i.e. didn’t produce the result in a separate M[]) - [ ] Calculate max instead of a prefixMax (i.e. finding max value among A[] values).

Q7b. findFirst(A, key)

Less efficient solutions: - [ ] O(N) linear search. - [ ] Standard binary search, then switch to linear search to handle duplicate keys (worst case still O(N), e.g. all keys are the same).

Common mistakes: - [ ] [Minor] Did not handle the possibility of "! found" as it is possible to have all values in M[] < than key. - [ ] [Minor] Use specialized checks that are error-prone and situational (e.g. if (M[mid] >= key && M[mid-1] < key), …).

Q7c. Maximum recover period.

Less efficient solution: - [ ] Brute force: generate all possible interval (i, j) and check for A[i] > A[j], keep the maximum "j-i".

Common mistakes: - [ ] Look for stretch of repeated number in the prefixMax array M[]. The prefixMax array doesn’t have enough information to determine the extend of recovery period. Sample counter example: A[] = {5, 7, 3, 11, 2}, M = {5, 7, 7, 11, 11}. Using only M[], there is no way to detect the recovery period from A[0], i.e. 5 to A[4], i.e. 2.

Q7d. Describe algorithm, show correctness and give example.

Marking decision: Need to touch on all 3 aspects. Most give example and description of the algorithm but didn’t try to show why it is the right way.

Q7e. Complexity

Marking decision: - [ ] Only reward mark if Q7c is correct. - [ ] Ties to your Q7a, Q7b and Q7c. i.e. the complexity must correctly reflect your own implementation, not the "best" complexity.