CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 4

For: Feb. 10–Feb. 4

1 Check in and PS3

Discuss questions, if you have any, with the tutor and the rest of the class, about the material and content so far.

Solution: **Notes :** Your goal here is to find out how the students are doing. One idea: ask each of them to send you (perhaps anonymously) ONE question before tutorial about something they are confused by. For questions that make sense to answer as a group, ask the students to explain the answers to each other. (For questions that are not suitable for the group, offer to answer them separately.)

2 Problems

Problem 1. Review AVL tree insertion and deletion algorithms in class. Work through an example that also covers the cases for rotations.

Problem 2. (A few puzzles with duplicates and missing elements.)
Assume you have an *unsorted* array containing $n-1$ unique elements in the range from $[1, n]$. How can you find the missing element? What if there are k missing elements? e.g. Given $[5, 2, 1, 4, 7, 3]$, we should return 6 here. (Hint: The partitioning algorithm would be quite useful here.)

Solution: The idea is to pick an element as a pivot and run partition. Now we're in a similar situation as the week before. If the pivot we pick is of value i and after we partition it is at index i, then we know all the remaining elements from 1 to $i-1$ are all also present. Else, it must end up at index $i-1$, so one element is missing, so we should recurse on that.

Problem 3. (Chicken Rice) Imagine you are the judge of a chicken rice competition. You have in front of you n plates of chicken rice. Your goal is to identify which plate of chicken rice is best, but it's only possible for you to compare when you eat the two plates in close succession.

Problem 3.a. A simple algorithm:

- Put the first plate on your table.
- Go through all the remaining plates. For each plate, taste the chicken rice on the plate, taste the chicken rice on the table, decide which is better. If the new plate is better than the one on your table, replace the plate on your table with the new plate.
- When you are done, the plate on your table is the winner!

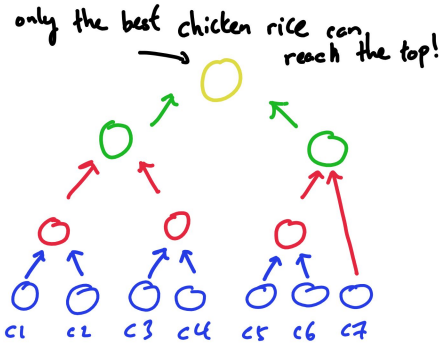
Assume each plate begins containing $n-1$ bites of chicken rice. When you are done, in the worst-case, how much chicken rice is left on the winning plate?

Solution: None! In the worst case the first plate on the table is the best plate of chicken rice already, and every comparison thereafter, you keep holding onto the same plate of chicken rice, and compare it to the remaining $n-1$ plates, so you end up taking $n-1$ bites out of the same plate.

Problem 3.b. Oh no! We want to make sure that there is as much chicken rice left on the winning plate as possible (so you can take it home and give it to all your friends). Design an algorithm to maximize the amount of remaining chicken rice on the winning plate, once you have completed the testing/tasting process. How much chicken rice is left on the winning plate? How much chicken rice have you had to consume in total? Here we assume deliciousness is transitive.

Solution: Use a tournament tree! Start with a perfectly balanced binary tree with each plate of chicken rice at a leaf. Work your way up the tree comparing plates, until you get to the root. That's the best chicken rice, and it only took $\log(n)$ bites off that plate. In total for each comparison you will only need to consume 2 bites, and in total you need to make $\leq 2n$ comparisons, so you only need to make at most $O(n)$ bites.

Below is an example with 7 plates of chicken rice. Notice that any plate of chicken rice only needs to be compared with another plate at most $\log n$ times.



Problem 3.c. Now I do not want to find the best chicken rice, but (for some perverse reason) I want to find the median chicken rice, i.e. the chicken rice that is the $\frac{n}{2}^{th}$ best, amongst the n. Again, design an algorithm to maximize the amount of remaining chicken rice on the median plate, once you have completed the testing/tasting process. How much chicken rice is left on the median plate? How much chicken rice have you had to consume in total? (If your algorithm is randomized, give your answers in expectation.)

Solution: Let's try to run QuickSelect. First, notice that in one level of QuickSelect (when the subarray is size n), almost all the plates in the subarray have one bite eaten from them; the unlucky "pivot" plate has $n-1$ bites eaten. If you choose the pivot at random, then the median plate has an expected cost of $(1/n)n + (1-1/n)1 \leq 2$. So at every level of recursion, there are at most two bites consumed from the median plate, in expectation. Since the recursion will terminate in $O(\log(n))$ levels (with high probability and in expectation), we conclude by linearity of expectation that the median plate only has $O(\log(n))$ bites consumed. In total, you have consumed $O(n)$ bites of chicken rice.

Another answer is to use an AVL tree. Inserting each plate into an AVL tree takes $O(\log n)$ comparisons. At that point, you can find the median, e.g., by doing an in-order traversal of the AVL tree. (Or, in class, we will see how to find order statistics in an AVL tree.)

In fact, using a randomized algorithm, this can be solved with only $O(\log\log(n))$ bites to the median plate, but that is much, much more complicated.

Problem 4. How much space does MergeSort as presented in class take? How can you improve this?

Solution: The basic MergeSort allocates a new array of size n to merge, when sorting n elements. So the recurrence for space looks like $S(n) = 2S(n/2) + n$, and we already know that the solution to this is $O(n \log n)$.

We can do better by using two arrays of size $n/2$ each, that we allocate in the very beginning. At each level of the recursion, to merge, we write into the extra arrays. Then before we return, we copy it back into the original array that it came in. In other words, we do not need to keep allocating space at every level of recursion.

Problem 5. (**Order Maintenance**)

The goal of the order maintenance problem is to maintain a total order over some (unspecified) objects. The data structure supports two operations:

- `InsertAfter(A, B)`: insert B immediately after A;
- `InsertBefore(A, B)`: insert B immediately before A;
- `isAfter(A, B)`: is B after A in the total order?

Notice that the insert operation adds B directly after A, while the query operation `isAfter(A, B)` asks whether B is anywhere after A in the total order. The expected complexity of each operation is $O(\log n)$, where n is the number of items in the data structure. Also, for this question, you're allowed to assume that when `isAfter(A, B)` is called, you're given the nodes corresponding to A and B.

Solution: (*Sketch*)
The data structure we use here is an AVL tree, but with an augmentation to how the insertion algorithm works. When inserting B after A, we do as follows:

- If A has no right child, then insert B as the right child of A.
- Otherwise, find the successor of A and insert B as the left child of the successor.

Either way, B now follows A in an in-order traversal of the tree. Complete the insertion by performing the rotations as described for an AVL Tree. When inserting B before A, do the same thing in reverse: either insert B as the left child (if none exists), or as the right child of the predecessor of A.
In order to determine whether A or B comes first, walk up the tree, until we find a common ancestor of A and B, i.e. when their paths "meet". We can then decide which comes first. We have to be careful, though, to compare ancestors at the same height. Thus, you need to walk all the way to the root from both A and B, storing for each step along the path (in an array), the key and whether the node was entered from the left or right. Then we can compare the two tree walks and find the common ancestor where one path entered from the left and the other from the right. (In class on Thursday, we will see an alternative wherein we can store the depth in each node, maintain it during rotations, and then use that to make this easier.)
The cost of all operations here is $O(\log n)$, since the height of an AVL tree of n nodes is at most $O(\log n)$.

Problem 6. (**Ancestor Queries**)

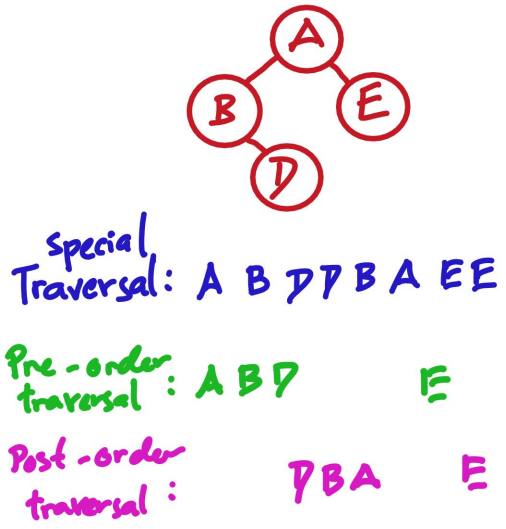
Our job now *simulate* a binary tree (not necessarily balanced). Each node has zero, one, or two children, and the tree is of height h. Unfortunately, it is not a balanced tree. We want to support the following operations:

- `InsertLeft(x, y)`: insert y as a left child of x in the binary tree.
- `InsertRight(x, y)`: insert y as a right child of x in the binary tree.
- `isAncestor(x, y)`: is x an ancestor of y in the binary search that contains them?

Solution: We define a new type of traversal in which each node in the tree appears twice: once before all of its children and once after all of its children. For a node v with children y and z, we define `traverse(v)` recursively as follows:

```
print(v) traverse(y) traverse(z) print(v)
```

For example, given the binary tree below, we have the corresponding traversals:



That is, the traversal first visits v, then both of v's children (first left, then right), and then visits v again. (Define `traverse(null)` to be empty.) If v has no children, then its traversal consists of just two elements: `v v`. Note here that each node is only going to be printed twice. Let's consider the sequence that we obtain out of this traversal. We'll call the first time a node is printed as `start(v)`, and the second time `end(v)`.

If you just focus on the the first time the nodes are printed in the traversal (removing all the second printings), then the resulting total order is exactly a pre-order traversal. Likewise, if you focus on the end nodes in the traversal (removing all the start nodes), then the resulting total order is exactly a post-order traversal.

Given two nodes v and w, we observe that v comes before w in a **pre-order** traversal of the original (unbalanced) tree if `start(v)` comes before `start(w)` in the traversal. Similarly, v comes before w in a **post-order** traversal of the original (unbalanced) tree if `end(v)` comes before `end(w)` in the traversal.

The idea is to maintain a data structure *OMtraverse* that stores this (double) "traversal" of the tree. That is, the total order in the order maintenance data structure is exactly the described traversal (where each node appears twice). To answer the `isAncestor(x,y)` query, we check whether x precedes y in the pre-order traversal and whether y precedes x in the post-order traversal. The following lemma explains why:

Lemma 1 *Node x is an ancestor of y if and only if x comes before y in a pre-order traversal and x comes after y in a post-order traversal.*

Solution: (continued)
Thus, using the OMtraverse data structure, we can determine whether x is an ancestor of y, by having to check if `start(x)` is before `start(y)` and if `end(y)` is before `end(x)`.
How about insertions? To insert a new node w as a child of x in the binary tree we are simulating, we need to insert `start(w)` and `end(w)` into OMtraverse. If w is the only child of x, then we simply insert `start(w)` after x and `end(w)` after `start(w)`. If w is the left child of x, then again we insert `start(w)` after x and `end(w)` after `start(w)`. Finally, if w is the right child of x, then we insert `end(w)` immediately before `end(x)` and we insert `start(w)` immediately before `end(w)`. The resulting ordering OMtraverse should exactly match the desired traversal of the binary tree.
So what kind of operations we need to support? Given 2 items, we need to be able to check whether one is before the other, and also, we need to be able to insert items as either directly before or directly after them. But hey! That's exactly what the data structure in the previous question supports. So we should really just be using that.
Finally, we may want to maintain some additional index structures, e.g., a tree to translate the name of a node to its location in the OMtravers data structure, and/or the name of a node to its location in the unbalanced binary tree. This tree would let us lookup x, find `start(x)` and `end(x)` in the tree, and use them to make the ancestor query.

Problem 7. (If you have time) What solutions did you find for Contest 1 (Catch the Spies)?

Solution: First, review the binary search approach. This should give a solution of cost $O(k \log n)$. Each binary search of cost $\log n$ finds one more spy.
Here's one approach for doing better:

- Choose a set of n/k students that are still unexamined and put them in set S.
- Check if there is at least one spy in S by sending all the students not in S on a mission together.
- If there is at least one spy in S, then repeatedly use binary search to find the spies in S. Each spy you find here will take $O(\log(n/k))$ time.
- Mark the students in set S as examined, and repeat with the remaining students.

Since each spy you find takes $O(\log(n/k))$ time, you will spend $O(k \log(n/k))$ time doing binary searches for spies. How many times will you query a set S and discover no spies? Well, there are only k different sets with n/k students, so this will take at most k queries. So the total number of queries is $k \log(n/k) + k$. This is optimal, since $\Omega(k \log(n/k))$ is the best you can do.