CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 6

For: Feb. 17–Feb. 21

Below is a proposed plan for tutorial for Week 6. Week 6 is dedicated to trees, augmented trees, tries, linked lists etc.

Plan:

1. Check in on how they are doing.
2. PS4 discussion
3. Problems

1 Check in and PS4

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.)

It also might be a good idea to review solutions to PS4, both the easier and the harder version. Especially if students haven't done the harder version, you can discuss the algorithm, and then encourage them to try to write the code.

2 Problems

Problem 1. kd-Trees A kd-tree is another simple way to store geometric data in a tree. Let's think about 2-dimensional data points, i.e., points (x, y) in the plane. The basic idea behind a kd-tree is that each node represents a rectangle of the plane. A node has two children which divide the rectangle into two pieces, either vertically or horizontally.

For example, some node v in the tree may split the space vertically around the line $x = 10$: all the points with x-coordinates ≤ 10 go to the left child, and all the points with x-coordinates > 10 go to the right child.

Typically, a kd-tree will alternate splitting the space horizontally and vertically. For example, nodes at even levels split the space vertically and nodes at odd levels split the space horizontally. This helps to ensure that the data is well divided, no matter which dimension is more important.

All the points are stored at the leaves. When you have a region with only one node, instead of dividing further, simply create a leaf.

Here is an example of a kd-tree that contains 10 points:

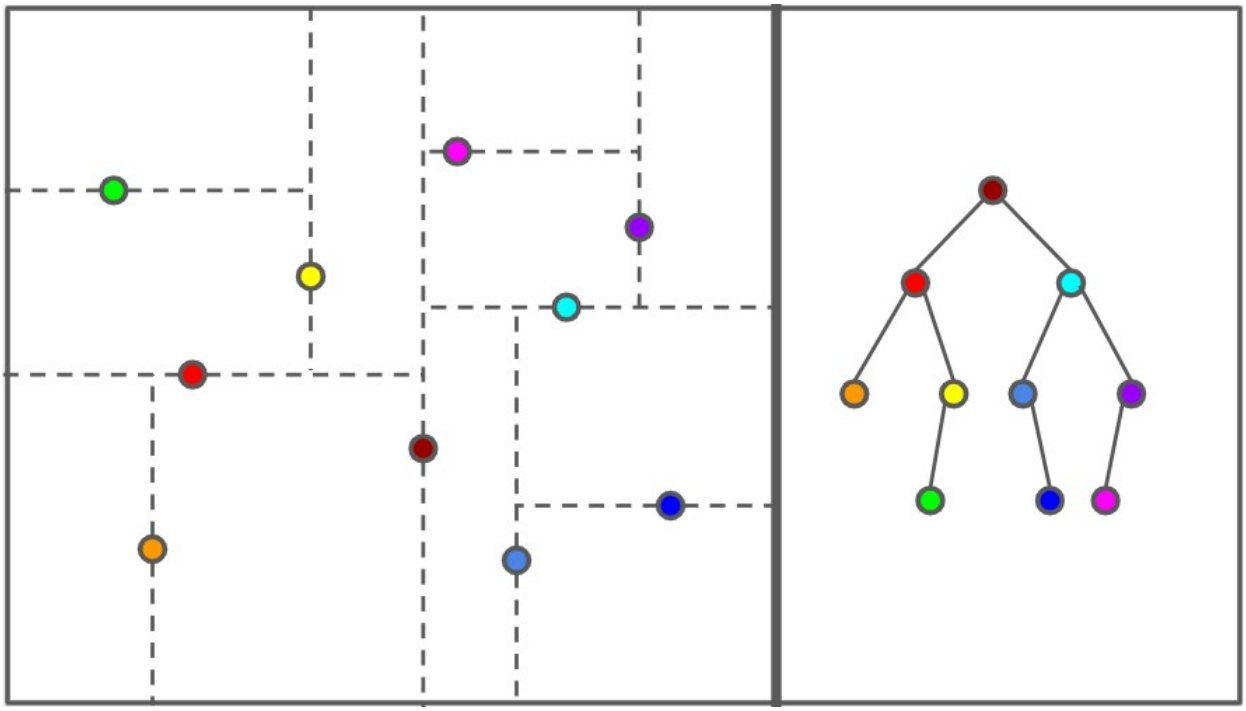


Figure 1: On the left: the points in the input. On the right: how the points are stored in the kd-tree

Problem 1.a. How do you search for a point in a kd-tree? What is the running time?

Solution: Start at the root. At each node, there is a horizontal or a vertical split. If it is a horizontal split, then compare the x-coordinate to the split value, and branch left or right. Similarly, for a vertical split. The running time is just $O(h)$, the height of the tree.

Problem 1.b. You are given an (unordered) array of points. What would be a good way to build a kd-tree? Think about what would keep the tree nicely balanced. What is the running time of the construction algorithm?

Solution: We can think of the construction recursively. At a given node, we have a set of points, and we need to split it horizontally or vertically. (We have no choice: that depends on whether it is an even or odd level.) Therefore, you might sort the data by the x or y coordinate (depending on whether it is a horizontal or vertical split), choose the median as the split value, and then partition the points among the left and right children. The running time of this is $O(n \log^2(n))$, since you spend $O(n \log n)$ at every level of the tree to do the partitioning, i.e., the recurrence is $T(n) = 2T(n/2) + O(n \log n)$. At this point, it is worth asking the students if they can think of a better solution. Something like QuickSort should come to mind! Instead of sorting at every level, we could either (1) choose a random split key, or (2) Use QuickSelect to find the Median. Then, the partitioning step is only $O(n)$, and so the total cost is $T(n) = 2T(n/2) + O(n) = O(nlogn)$. (As an aside, one can use master's theorem/substituion method to show that this imples $T(n) = O(n \log^2 n)$).

Problem 1.c. How would you find the element with the minimum (or maximum) x-coordinate in a kd-tree? How expensive can it be, if the tree is perfectly balanced?

Solution: To find the minimum, if you are at a horizontal split is easy: simply recurse on the left child. But, if you are at a vertical node, you have to recurse on both children, since the minimum could be in either the top half or the bottom half. (Write out the recursive pseudocodde.)
To find the running time, let's look at the recurrence from taking two steps of the search (one horizontal and one vertical): $T(n) = 2T(n/4) + O(1)$. At each step down the tree, the number of points divides in half, i.e., $n/2$ after one step and $n/4$ after two steps. After two steps of the search, there are two more recursive searches to do. Solving this recurrence, you get a recursion tree that is depth $\log(n)/2$, each node has cost $O(1)$, and there are $2^{(\log(n)/2+1}$ nodes in the tree, so the total cost is $O(\sqrt{n})$.
For more kd-tree fun, think about how you would search for the nearest neighbor of a point!

Problem 2. Tries(a.k.a Radix Trees) Coming up with a good name for your baby is hard. You don't want it to be too popular. You don't want it to be too rare. You don't want it to be too old. You don't want it to be too weird.[1]

Imagine you want to build a data structure to help answer these types of questions. Your data structure should support the following operations:

- `insert(name, gender, count)`: adds a name of a given gender, with a count of how many babies have that name.
- `countName(name, gender)`: returns the number of babies with that name and gender.
- `countPrefix(prefix, gender)`: returns the number of babies with that prefix of their name and gender.
- `countBetween(begin, end, gender)`: returns the number of babies with names that are lexicographically after `begin` and before `end` that have the proper gender.

In queries, the gender can be either boy, girl, or either. Ideally, the time for countPrefix should not depend on the number of names that have that prefix, but instead run in time only dependent on the length of the longest name.

Solution: The point here is to use a trie, and store in each node in the trie the count of names under that node, for each gender. (Beware the same names may be used for both genders, for example, Jordan is today both a girl and a boy name. So you risk double-counting if you are not careful. You might want to keep three categories: boy, girl, both.) Finding a prefix then just involves going down to the node in question, while counting between requires finding the beginning node and the ending node and subtracting.
Don't forget that when you insert strings into the trie, you have to update the counts at the nodes in the trie.

[1] The website https://www.babynamewizard.com/voyager let's you explore the history of baby name popularity!

Problem 3. Skip Lists A skip list is a randomized data structure that allows you to search, insert, and delete items in $O(\log n)$ time with high probability. We talked about insert and search in class. How should delete work?

Now, imagine you are a hacker and want to make my skip list run very slowly. Imagine you have sufficient access to my machine that you can insert and delete items from my skip list, but you cannot actually see the data structure. What would you do? (**Hint :** What would cause searches to be slow on a skip lists? Relate that to the runtime of every operation on skip lists.)

As a follow up: would it be different if you could see the internal state of the skip list?

Solution: The interesting point here is that when we talk about randomized data structures, it makes a big difference if the "adversary" (in this case, the hacker) gets to see the random coins. If you can see the state of the skip list, all you have to do is delete the high up items in the skip list, and you can make it very slow.
Now, even without access, if you are clever, you might run a timing attack. First you query the item and see how fast an answer you get. For high up items, you will get a faster response, which you can then delete. This type of timing attack is a powerful and sometimes non-obvious way to attack a system! (Go take a security class!)

Problem 4. Finger Searching It seems like it should be easier to find an element that is near an element youve already seen, right? That's what a **finger search** is for. Assume you have a tree of some sort. A finger search is the following query: given the node in the data structure that stores the element x, and given another element y, find the node in the data structure that stores y. Ideally, the running time should depend on the distance from x to y, for example, $\log(|x - y|)$ would be optimal.

Problem 4.a. Say that we had to implement **finger search** on a vanilla AVL tree, wtihout any further modifications, what would a very straightforward solution be? Give an example where you cannot do better than just directly searching for y, given the node x.

Solution: In an AVL tree, imagine that x is leaf $n/2 - 1$ and y is leaf $n/2 + 1$. The distance here is 2, but the only path connecting these two nodes is via the root. Thus the best you can do is $\log n$.

Problem 4.b. What if you want to implement a finger search in a skip list? How would you do it? What sort of running time do you get?

Solution: Assume you are searching for $y > x$. The search proceeds in two phases. First, climb vertically until you find a level where the next node on the list has a key $> y$. To climb the list, you go as high as you can on the current node; then advance forward one step, and repeat until you reach a point where you are stuck. Then do a normal skip list search from that point, advancing as far forward as you can forward before going down, repeating until you get to a leaf.
To analyze the running time, assume that $d = |x - y|$. By the same analysis in class, you can show that with high probability, none of these d nodes between x and y have height $> O(\log d)$. (This is because the probability of flipping $c \log d$ heads in a row is exponentially small in c.) So moving up and down will never cost more than $O(\log d)$.
We now need to bound how many steps forward you need to take. Each time you take a step foward, there is a $1/2$ probability that the next node was promoted to the next level. So you are really asking how many times do you need to flip a coin before you see $O(\log d)$ heads. We know that the expected number of flips is $O(\log d)$, but it is also true with high probability. So within $O(\log d)$ steps, the first phase completes. The same argument in reverse shows that the second phase finishes in $O(\log d)$ steps too! [a]
If you try to make an AVL support this, you can, but it might require a bit more work!

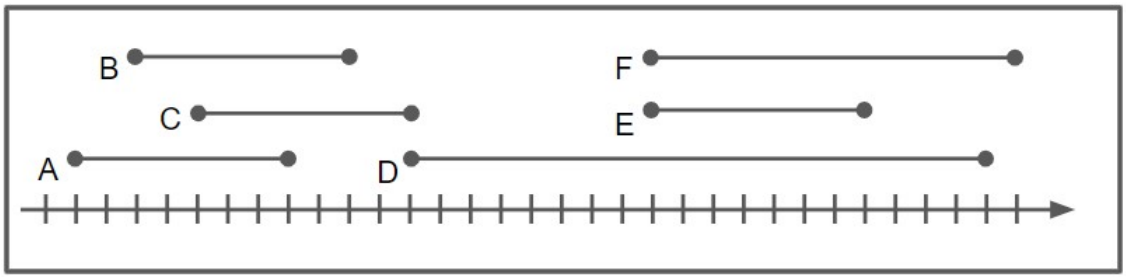
[a] See the following URL for some notes: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/assignments/ps5sol.pdf

Problem 5. Pandemic? Oh no! Turns out that the Coronavirus situation is hitting Singapore pretty hard and the university is getting pretty worried. They've implemented a strict policy where there can be no more than 50 people at an event/gathering at the same time. This is a bit of a pickle for Mr. Nodle, because he's planned for an all-day event at one of the rooms at SoC, where people may come and leave the room as they please. Thankfully, each participant has kindly indicated the start and end times that they will be there.

Problem 5.a. The first job he has, is to figure out whether at any point in time there will be more than 50 people simultaneously in the room (not including him). Given that n people who wish to participate in the event with their own start and end times, give an $O(n \log n)$ algorithm that solves this.

(As a follow up, you might also want to think about what happens if we assume that every participant's start and end times can only be for example hours of the day, rather than just arbitrary times. Can we do better?)

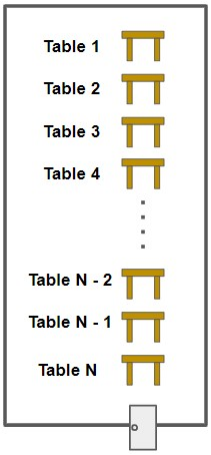
In the example below, there are only at most 3 people in the room at any point in time:



Solution: The idea here is to make two arrays, one containing all the start times and one containing all the end times. Then sort each separately. After that, initialise a counter to 0. Use one pointer at the starting of each array, and advance the pointer that is pointing to the earlier time. If the pointer on the start array is advanced, then increment the counter. Else, it must be that the pointer for the array containing the end times is being advanced, so decrement the counter by 1. What the counter is counting is really just the total number of intervals that are simultaneously open at the same time.
The solution that the algorithm should return is just the maximum possible value that the counter ever achieves. The first and last part of the algorithm run in $O(n)$ time, but since we're sorting here, say e.g. using mergesort, the total time is $O(n \log n)$.
Additionally, if we're allowed to assume that there are only 24 possible start/end times for each person, there there are two faster approaches: One is to simply use counting sort instead, which will run in $O(n)$ time. Another solution is to keep an array of size 24, with all the elements initialised to 0. For each $(start, end)$ time interval, we simply increment all the elements in $A[start..end]$ by 1. Then do one more pass to find the largest element in this final array. The entire thing takes $O(n)$ time.

Problem 5.b. Now Mr. Nodle has another concern: the university wants people to also log

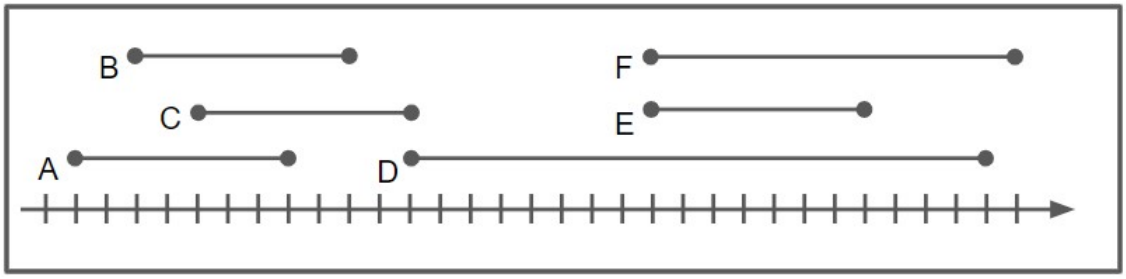
whoever has come into contact with whomever. Now the room layout is like this:



The earliest person of the day to come in sits at table 1, second earliest of the day at table 2, and so on. We're still given the list of starting and ending times that each person will be there. They also need to enter from the front, and leave through the front. So anyone they come across during the process of entering and leaving, counts as contact.

Given a list of the start and end times for each participant, output the **total number of instances of contact** in $O(n \log n)$ time. For simplicity, you may assume the input comes as an array of triples in the form of $(x, start, end)$ where x is the id of the student who is going to be present during that time interval[2]. (**Hint:** an ordered statistics tree would be very useful here.)

As an example:



There are a total of 6 instances here, namely: $(A, B), (A, C), (B, C), (C, D), (D, F), (E, F)$. Note that (D, E) here doesn't count because E arrived after D did but left before D did, so that means that D E did not come into contact (since E was sitting in front of D, so it did not have to cross D to get to his/her table). Furthermore, for example, (A, D) is not an instance of contact since A left before D entered, even though A crosses the table that D would have sat at.

[2]The problem still is solvable if the input instead was simply $(start, end)$, but let's make life a little easier.

Solution: **Note:** You might want to restrict all times to be distinct first, then leave students to discuss how to handle the equality cases.
The main idea here is that we are going to process each interval, in the order of their start times and end times again. Take note that two intervals $(s_1, e_1), (s_2, e_2)$ intersect if it comes in the form $s_1 < s_2 < e_1 < e_2$.
So what we need to do is again process the start and ending points for each interval separately. And every iteration, if we are about to "close" an interval, we need to ask how many intervals have been opened after it was open, that still remained open. What we can do is keep adding the start times into an AVL tree. Then when we need to find out how many start times have been opened after some specific start time s, we can find it as $size - rank(s)$, where $size$ is the total number of start times that are currently open, and $rank(s)$ is the number of start times that are before s.
So the algorithm looks like this:

1. Create two arrays, one to hold all the starting times, and one to hold all the ending times.
2. Again, use two pointers to keep track of the next point to process. Consider the next smallest value.
3. If the next value is a starting time, add it into an AVL tree with the starting time as the key.
4. If the next value is an ending time instead, search for the rank of its corresponding start time in the AVL tree, and then add $size - rank$ to the counter, where $size$ is the size of the AVL tree at that point in time. Then delete the corresponding start time from the AVL tree.
5. Repeat until all the elements have been processed.

Problem 6. Athletes
Problem 6.a. The Olympics are coming up soon and Mr.Govond is looking to send a strong contingent to win as many medals as possible. In this simplified scenario, every single hopeful has a strength attribute and speed attribute based on a complex set of tests Mr.Govond has run. Now, Mr Govond wants to extract the *athletes* from this set. We define an athlete as someone among all the hopefuls such that there is no one else (other than themselves) that are both stronger *and* faster than them. As an example consider an toy example where there are 4 hopefuls with the following speed and strength attributes:

- Usain Bolt, Speed 110, Strength 40
- The Mountain Speed 40, Strength 145
- Joseph Schooling Speed 70, Strength 60
- Mr. Ordinary Speed 30, Strength 30

Now Usain Bolt and The Mountain are obviously athletes since they are the fastest and the strongest among the 4 athletes presented. However, Joseph Schooling is also an athlete as well. The Mountain is stronger than him, but not faster than him and Usain Bolt is faster than him but not stronger than him. Hence, he fits the criteria of being an athlete as well. Of course Mr. Ordinary is not an athlete as everyone of the others are both faster and stronger than him. However, Mr.Govond has *a lot* of data to trawl through, and he wants to send his selected list by tomorrow. Help him come

up with an efficient algorithm such that given the strength and speed attributes of every hopeful, you can output the athletes as efficiently as possible.

Solution: There are many ways to do this, so if your students do come up with alternative methods to solve this question, do discuss it with them. One way we can do this is to sort the hopefuls by any attribute. Let's say we sort them by speed. Now, the fastest person is definitely an athlete. We shall take note of his strength and maintain a strength threshold. Subsequently, as we look at the athletes in terms of descending speed, we shall see whether the hopefuls exceed this strength threshold. If they do, they are also an athlete, because he is faster than everyone after him, and they are stronger than all the athletes that have been scanned before. Then we shall update the strength threshold to match that of the new athlete. We continue iterating through the athletes in this manner until we have reached the slowest hopeful. This will take $O(NlogN)$ time if we have N athletes, due to the sorting step. Subsequently, you are able to iterate and extract all the athletes in $O(N)$ time.

Problem 6.b. Now that Mr.Govond has selected his athletes, he sends the list to the Sports Ministry for their approval, along with the supporting data from the trials. However, a scandal breaks which suspects that data has been tampered with! Now the data from the trials is weirdly enough stored as a binary tree. Now given the original copy of the tree as well as the tree received by the Sports Ministry, can you determine whether they are the same tree? You may assume you have pointers to the children of every node,as well as parent pointers as well. You may also assume that checking equality between two tree nodes can be done in $O(1)$ time.

Solution: Again, there are multiple ways to do this.There is a very simple linear algorithm for checking whether two trees are identical. To do this, we require the following observation : any binary tree can be uniquely defined by its pre-order and in-order traversal or its post-order and in-order traversal. (Note that pre-order and post-order cannot uniquely identify binary trees. Try and get your students to come up with an example!). With this information, all we need to do is to simply do one pre-order and in-order traversal and check whether they match. Assuming that the tree node comparison can be done in $O(1)$ time, we can simply construct a pre-order and in-order traversal and compare whether they match. If they do, they are the same tree. If not, they are different trees. Some students may come up with a hashing idea (especially the Friday class). If time permits, you could talk about how such a hash would look like.