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:

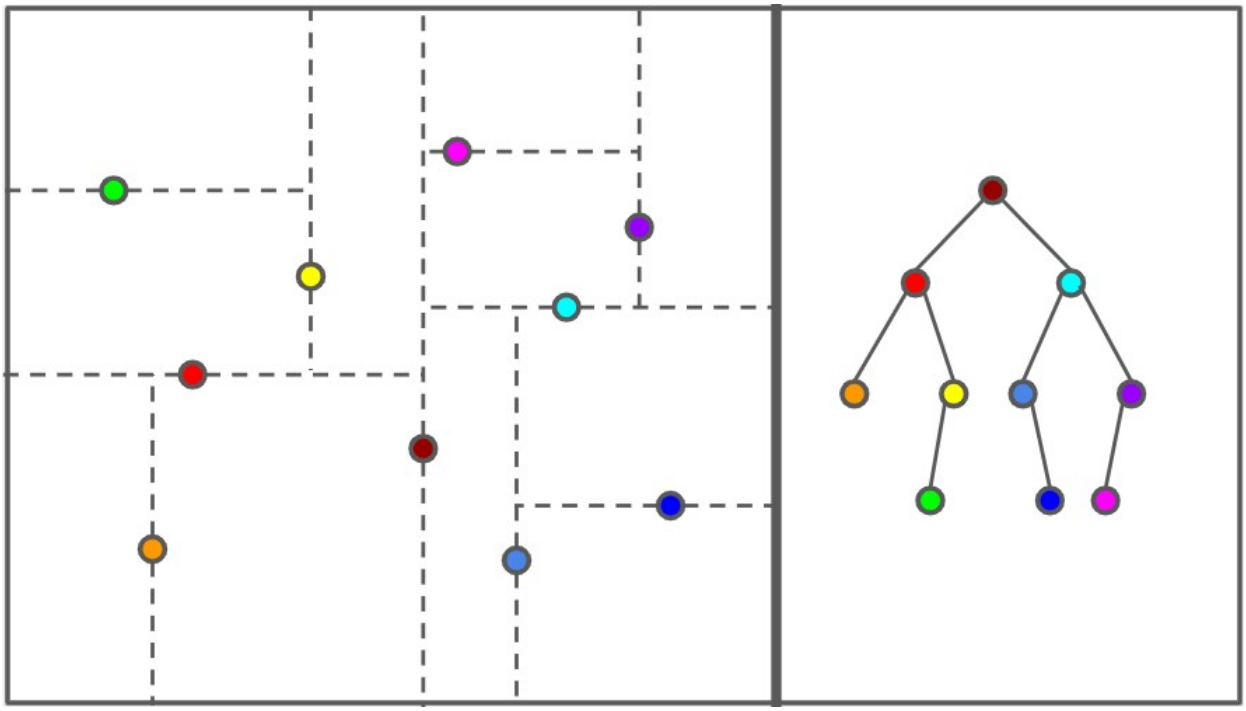


Figure 1: On the left: the points in the input. On the right: how the points are stored in the kd-tree

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

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?

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?

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:

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

- `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.

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?

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.

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?

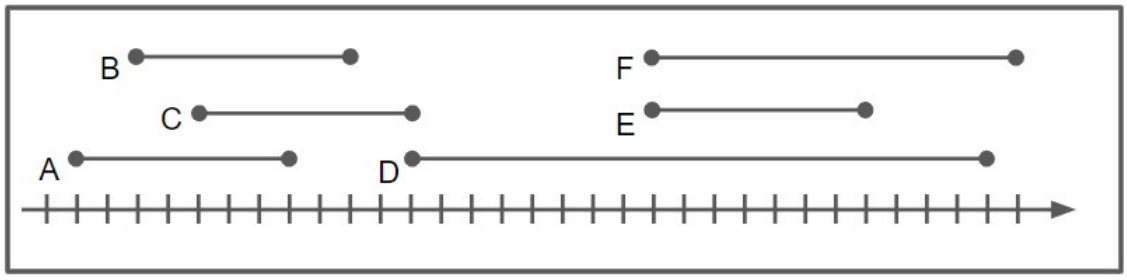
Problem 5. Pandemic? Oh no! Turns out that the Coronavirus situation is hitting Singapore pretty hard and the university is getting pretty worried. They've implemented a strict policy where there can be no more than 50 people at an event/gathering at the same time. This is a bit of a pickle

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.

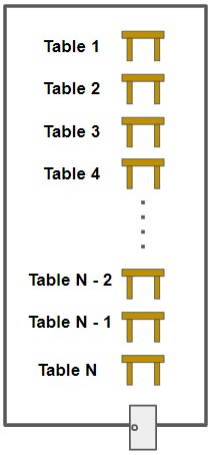
(As a follow up, you might also want to think about what happens if we assume that every participant's start and end times can only be for example hours of the day, rather than just arbitrary times. Can we do better?)

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



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

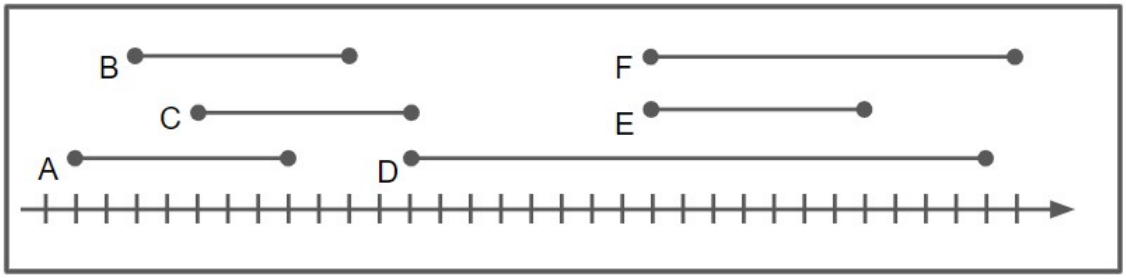
whoever has come into contact with whomever. Now the room layout is like this:



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.

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

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

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.

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.