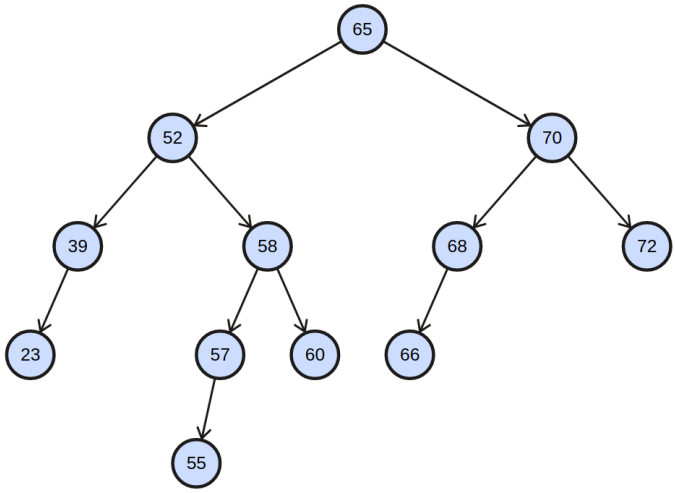
CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 6

For: February 15–February 19

1 Review

Problem 1. AVL Trees



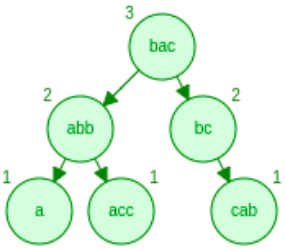
Problem 1.a. Trace the deletion of the node with the key 70.

Problem 1.b. Identify the roots of all maximally imbalanced AVL subtrees in the original tree. A maximal imbalanced tree is one with the minimum possible number of nodes given its height h.

Problem 1.c. During lectures, we've learnt that we need to store and maintain height information for each AVL tree node to determine if there is a need to rebalance the AVL tree during insertion and deletion. However, if we store height as a `int`, each tree node now requires 32 extra bits. Can you think of a way to reduce the extra space required for each node to 2 bits instead?

Problem 2. AVL vs Trie

Problem 2.a. Transform the given AVL tree with string values into a trie.



Problem 2.b. Insert "acac" to the trie from the previous question.

Problem 2.c. Discuss the trade-offs of using AVL and Trie to store strings.

2 Problems

Problem 3. 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:

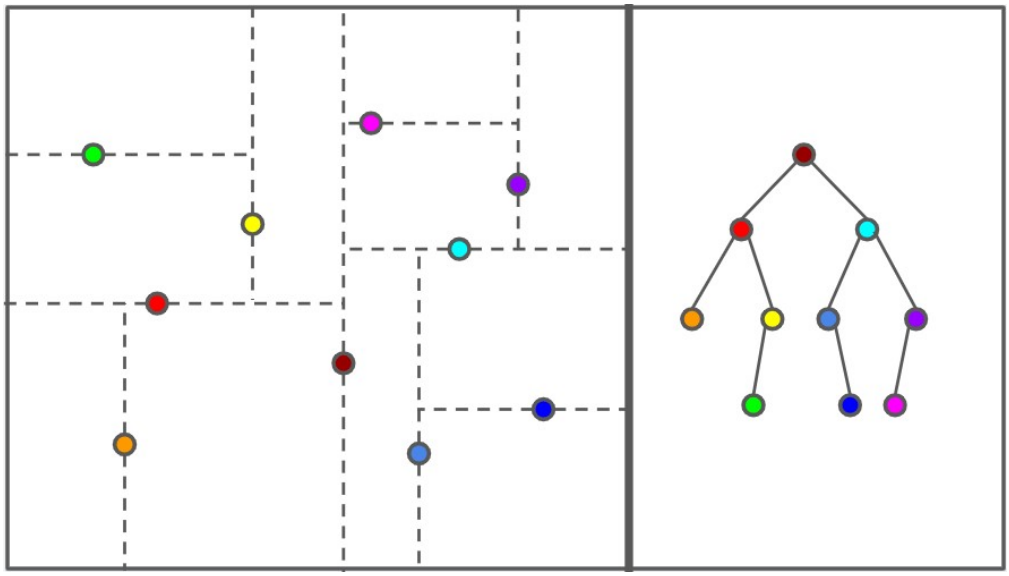


Figure 1: On the left: the points in the input. On the right: how the points are stored in the kd-tree

Problem 3.a. How do you search for a point in a kd-tree? What is the running time?

Problem 3.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 3.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 4. 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.

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

- `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 5. (Challenge) Finger Searching

It seems like it should be easier to find an element that is near an element you've 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 distance d, which refers to the difference in ranks between x and y, for example, $O(\log(d))$ would be optimal.

Note: this problem is harder compared to the rest of the tutorial.

Problem 5.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 5.b. What if you are allowed to use other kinds of data structures to implement efficient finger search? For example, a tree with all keys at the leaf level. What sort of running time do you get?