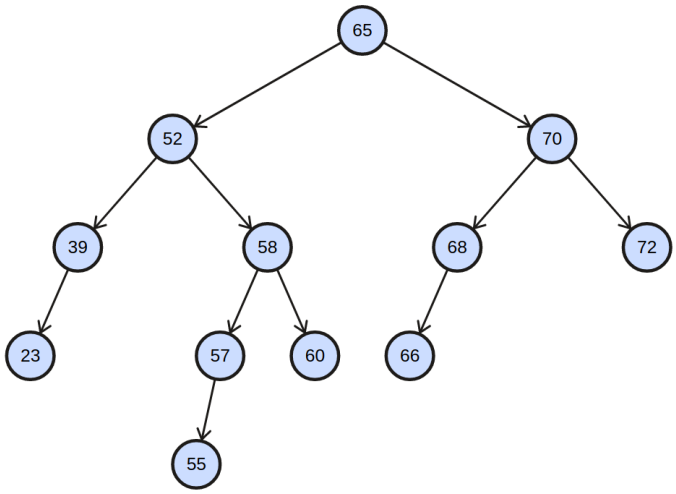
CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 6

For: February 15–February 19

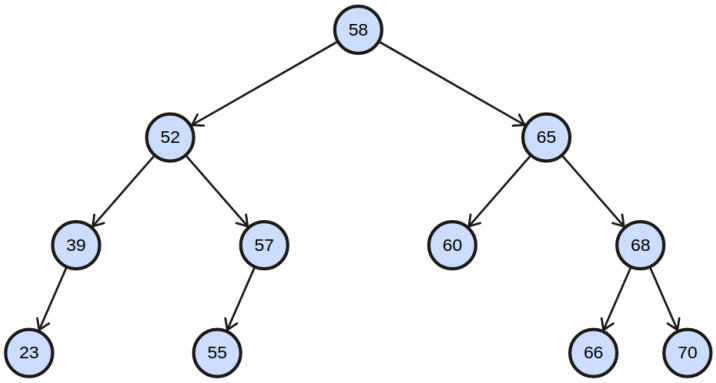
1 Review

Problem 1. AVL Trees



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

Solution: Get the successor of 70, which is 72 and copy the value over. After deleting the node, the subtree rooted at node 72 is now imbalanced. A left-left rotation is required. Then, the tree rooted at node 65 is now imbalanced. A left-right rotation is required. In total, 2 sets of rotations are performed.
The following is the final configuration of the AVL tree.



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.

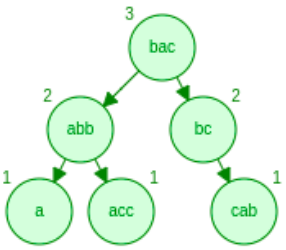
Solution: All nodes are the roots of maximally imbalanced AVL subtrees. In particular, a maximal imbalance AVL tree has all its subtrees to be maximally imbalanced. This is simply a consequence from the fact that an AVL tree with the minimum possible number of nodes with height h has two subtrees with minimum possible number of nodes with height $h-1$ and $h-2$, namely $S(h) = S(h-1) + S(h-2) + 1$.

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?

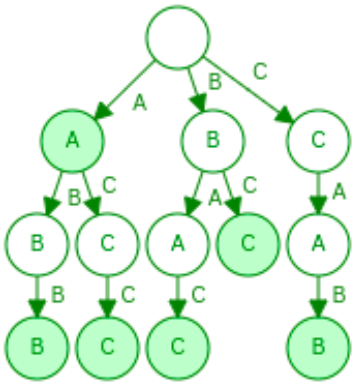
Solution: Instead of storing the height, we can store and maintain the balance factor for each node. Balance factor is equal to the difference between the left and right subtrees of a node. For AVL trees, each node can have balance factor of -1, 0 or 1, which only requires 2 bits.

Problem 2. AVL vs Trie

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

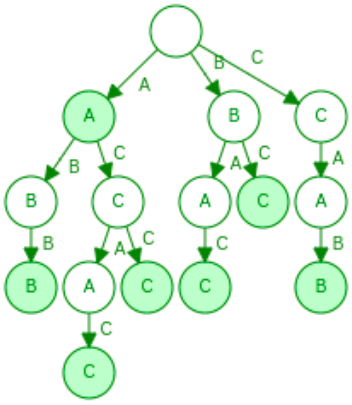


Solution: Here's the resulting trie:



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

Solution: Here's the resulting trie:



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

Solution:

- Time complexity for insert, delete and find a word with length L to a collection of N words: $O(L \log N)$ for AVL and $O(L)$ for trie.
- Space complexity is $O(total\_string\_length)$ for both AVL and trie. But, trie tends to have more overhead cost.

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:

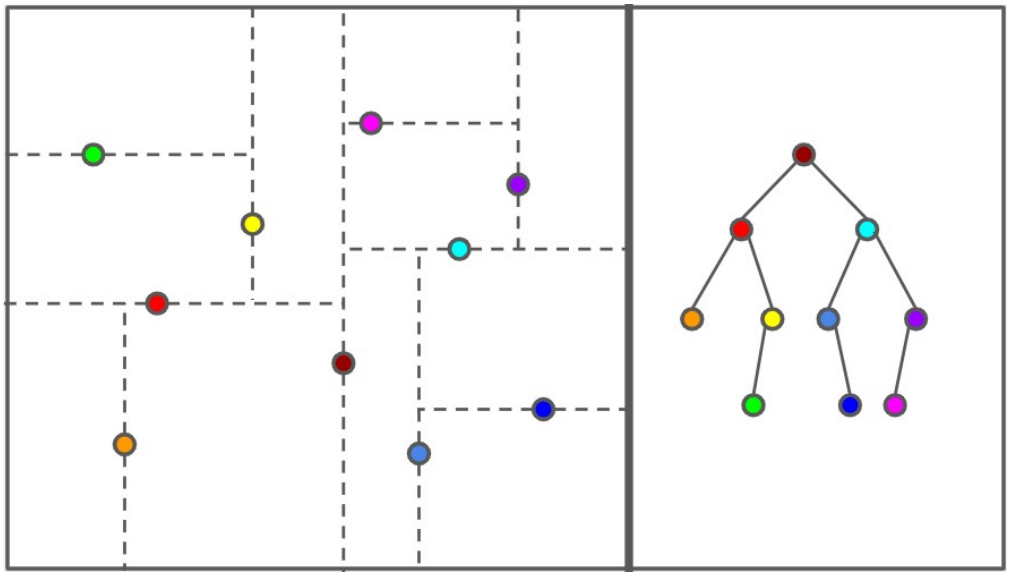


Figure 1: On the left: the points in the input. On the right: how the points are stored in the kd-tree

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

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 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?

Solution: Basic approach:
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 $T(n) = 2T(n/2) \ + O(n \log n)$.

Solution: How to do better:
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 $T(n) = 2T(n/2) \ + O(n) = O(nlogn)$.

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?

Solution: To find the minimum, if you are at a horizontal split, it 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 $O(2^{\log(n)/2})$ 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 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]

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

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. Don't forget that when you insert strings into the trie, you have to update the counts at the nodes in the trie.

- `countPrefix` just involves going down to the node and return the count stored at the node.
- For `countName`, search for the node, check if the end of word flag is set to `true`. If it is false, return 0. Otherwise, the answer is given by subtracting the count of the node by the total count of its child nodes.
- For `countBetween`, find `rank(begin, gender)` and `rank(end, gender)` respectively by traversing upwards from `begin` and `end` first. Then return `rank(end, gender) - rank(begin, gender) - countName(begin, gender)`, where `rank` is defined as follows:

```
rank(node, gender):
r ← 0
Traverse from x = node to root:
   If parent of x is a name:
      r ← r + parent.countName(gender)
   For all left siblings x' of x:
      r ← r + x'.count(gender)
return r
```

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.

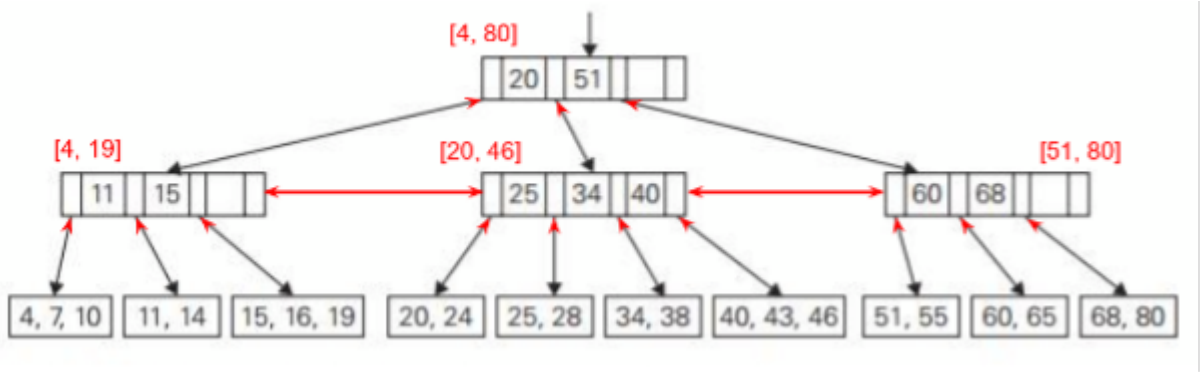
Solution: In an AVL tree, imagine that x is leaf $n/2-1$ and y is leaf $n/2+1$. The distance here is 2, but the only path connecting these two nodes is via the root. Thus the best you can do is $O(\log n)$.

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?

Solution: Use a B-Tree with the following modifications:

- Each node has a parent pointer.
- Connect all nodes on the same level, to form a doubly-linked list on each level of the tree.
- The inner nodes store search keys, the actual data is only stored at the leaves.
- We need to maintain the minimum and maximum values for the subtree rooted at each node.

The red portions below illustrate these changes (the numbers in square brackets are the minimum and maximum values for the subtree rooted at that node):



Original image obtained from:
https://www.brainkart.com/article/B-Trees-Algorithms_8040/

To perform the finger search from x to y, we first check whether y is to the left or right of x. Assume without loss of generality we are searching for $y > x$. We traverse the path from x towards the root while examining the nodes v on the path and their right neighbours u until we know that y is contained within the subtree rooted at v or u (i.e. y falls within the [min, max] range). Then, we continue by doing normal BST searches for y starting at v and/or u. The runtime for this would be $O(\log d)$ where d is the difference in ranks for x and y.

Solution: (Continued)
Note: While it may not be clear from the small B-Tree diagram above, we actually need the doubly-linked list among nodes of the same level, and cannot just rely upon moving up to a common parent before moving down again. To understand why, imagine a situation where x is the rightmost child of the root's left subtree, and y is the leftmost child of the root's right subtree. In this case, without the links, we would be travelling all the way up to the root before we can start moving down again.

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