Lecture 34: Data Structure Choices and Balanced BSTs
10:00 AM, Apr 20, 2018

Contents

1 A First Scenario: An Apartment Listing Site 1

2 Balancing Binary Search Trees 2
  2.1 AVL Trees ................................................................. 2

Objectives

By the end of this lecture, you will have:

- gotten some practice at choosing data structures for real-world scenarios
- learned about self-balancing BSTs

1 A First Scenario: An Apartment Listing Site

The Soccer/Football World Cup is coming to BigCity, and all the residents who aren’t into sports can’t wait to get out of town for the weekend. But they aren’t above making a little money on the side, so they start up a website where people can list their apartments for rent to spectators.

Each listing contains the rental price for the weekend, the number of beds the unit has, the street address, and an optional photo. The site also stores the email address of the person who posted each listing, but the email addresses are not displayed on the site.

When a renter visits the site, they can search on either or both of the number of beds or the price range. The site displays those listings that match the search requirements.

Listings get added to or removed from the site throughout the time that the site is up.

Questions:

1. Which operations occur frequently, and thus need an efficient run-time?
2. What data structure(s) do you propose for managing listings?
3. Justify your answer to question 2 with respect to your answer to question 1

Summarizing the discussion, we talked about using each of sorted lists, heaps/priority queues, binary search trees, and hashmaps. Priority queues would only make sense if people only ever wanted the lowest price. Sorted lists require linear time searches. Hashmaps don’t do a good job at supporting
range searches. Binary search trees, in contrast, are very good at doing range-based searches – you need log time (in the number of nodes) to find the lower and upper bounds of the search, then all intermediate nodes should be part of the answer.

Range searches are one of the reasons why binary search trees are heavily used in databases, rather than hash tables. But the log-time only happens if the trees are balanced, which isn’t something we’ve discussed maintaining algorithmically before now.

2 Balancing Binary Search Trees

At the end of the notes on Binary Search Trees, we noted that a tree with only one long branch counts as a BST, but is no better than a list with regards to searching for elements. So while BSTs are usually faster for finding elements than lists, they aren’t guaranteed to be faster than lists on this operation.

The core problem here is that BSTs don’t require the trees to be wide (as opposed to tall). If trees had to be wide, then we’d have elements to ignore as we searched the tree. We formalize the idea of "wide" by creating Balanced binary search trees (BBST).

There are several flavors of BBSTs, depending on the specific constraint that they use to achieve balance. Here, we will work with a BBST variant called AVL Trees.

2.1 AVL Trees

AVL trees augment the binary search tree invariant to require that the heights of the left and right subtrees at every node differ by at most one (“height” is the length of the longest path from the root to a leaf). For the two trees that follow, the one on the left is a BBST, but the one on the right is not (it is just a BST):

```
6
/ \ / \  
3  8 3 9  
/ \         /  
7  12  
/    
10
```

We already saw how to maintain the BST invariants, so we really just need to understand how to maintain balance. We’ll look at this by way of examples. Consider the following BST. It is not balanced, since the left subtree has height 2 and the right has height 0:
How do we balance this tree efficiently? Efficiency will come from not structurally changing more of the tree than necessary. Note that this tree is heavy on the left. This suggests that we could "rotate" the tree to the right, by making 2 the new root:

```
  2
 / \
?  4
1 .. 3
```

When we do this, 4 rotates around to the right subtree of 2. But on the left, we have the trees rooted at 1 and 3 that both used to connect to 2. The 3 tree is on the wrong side of a BST rooted at 2. So we move the 3-tree to hang off of 4 and leave 1 as the left subtree of 2:

```
  2
 / \
1  4
 / /
3
```

The 4 node must have an empty child, since it only keeps one of its subtrees after rotation. We can always hang the former right subtree of the new root (2) off the old root (4).

Even though the new tree has the same height as the original, the number of nodes in the two subtrees is closer together. This makes it more likely that we can add elements into either side of the tree without rebalancing later.

A similar rotation counterclockwise handles the case when the right subtree is taller than the left. One more example:

```
  7
 / \
5  10
 / \
8  15
 \  
  9
```

This tree is heavier on the right, so we rotate counterclockwise. This yields
which is no better than what we started with. Rotating clockwise would give us back the original tree. The problem here is that when we rotate counterclockwise, the left child of the right subtree moves to the left subtree. If that left child is larger than its sibling on the right, the tree can end up unbalanced.

The solution in this case is to first rotate the tree rooted at 10 (in the original tree) clockwise, then rotate the resulting whole tree (starting at 7) counterclockwise:

```
10
 / \
7 15
 / \
5 8
  \
   9
```

```
7
 / \
5 8
  \
   10
   / \
   9 15
```

```
------------------
8
 / \
7 10
 / / \
5 9 15
```

This gives you a feel for rebalancing, but not the details you need to understand the algorithm. Algorithmically, we

- Label every node with the difference between the height of its right child and the height of its left child. A positive difference means the tree is heavy on the right, while a negative one means the tree is heavy on the left.

- Find the lowest node (in any branch) whose difference is at least 2 (whether negative or positive):
  - If the node’s difference is positive
    * if the difference at its left child is negative, rotate the left child to the right, then rotate the node to the left.
    * else just rotate the node to the left
  - The case for a negative node difference is analogous
Note that each time you rotate a subtree, the differences in nodes higher-up might change. You update the heights and continue to rotate as needed, until you have moved up to the root of the tree.

In practice, we don’t start with a massively unbalanced tree. We start with a balanced tree, which could become unbalanced if a node is added or deleted. On each add/delete operation, we check whether the tree needs to be rebalanced. This localizes the needed changes.

Wikipedia has a good description of AVL trees. Refer to that for more details on the rotation algorithm if you are interested.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs018/feedback](http://cs.brown.edu/courses/cs018/feedback).