AVL Trees: Augmentations and Range Queries

Neena Dugar
ndugar@mit.edu
6.006 Spring 2019

1 Example Problem

Pary Moppins runs a very large daycare, where children join and leave regularly, but there are at most n children at any given time. Every child has age a, and an integer badness b, which Pary Moppins wishes to store in a database. Age is stored as an integer number of seconds after some reference time you can assume the ages are unique.

Describe a database which supports the following operations in $O(\log n)$ time:

- `add_child`(a, b): add a child with age a and badness b
- `remove_child`(a): remove the child with age a
- `badness`(a_{max}): return the sum total badness for children younger than a_{max}

2 Solution with Commentary

This problem is very similar to the Rainy Research question from Quiz 1 we will use an augmented AVL tree and a range query to solve this problem!

2.1 AVL with Augmentation

Create an AVL tree keyed by age, storing badness, augmented with subtree sums of badness. That is, at any given node n store the sum of the badness in the subtree rooted n as follows:

$$\texttt{n.sum\_b} = \texttt{n.left.sum\_b} + \texttt{n.right.sum\_b} + \texttt{n.b}$$

Since the subtree sum badness of n is only dependent on that of its children, we can maintain this in $O(1)$ time. *See Lecture 6 notes for details on maintaining augmentations.*

The `add_child` and `remove_child` functions are implemented by simply adding and removing nodes from the AVL tree in $O(\log n)$ time.

2.2 Range Query

To compute `badness`(a_{max}), we can use a one sided AVL range query, which requires defining a recursive function. We define `total_b`(n, a_{max}): the total badness for children younger than a_{max} in the subtree rooted at n. Consider two possible cases:

Case 1: `n.age` $< a_{max}$

- All items in the **left subtree** are smaller than $n.age$ (through the BST property), so if $n.age < a_{max}$ then they are also smaller than a_{max}. Therefore all items in this subtree must be included in our badness sum, i.e. `n.left.sum_b`
- The child at n is by definition of the case younger than a_{max}, so their badness should be included.
- Some portion of the **right subtree** may be smaller than a_{max} (but not necessarily the entire subtree), so we recurse on the right: `total_b`($n.right, a_{max}$)

All this gives, if `n.age` $< a_{max}$, `total_b`(n, a_{max}) = `n.left.sum_b`+`n.b`+`total_b`(`n.right`, a_{max}).

Case 2: $a_{max} \leq$ `n.age`

- Some portion of the **left subtree** may be smaller than a_{max} (but not necessarily the entire subtree), so we recurse on the left: `total_b`($n.left, a_{max}$)
- The child at n is by definition of the case at least as old as a_{max}, so their badness should not be included.
- All items in the **right subtree** are older than $n.age$ (through the BST property), so if $n.age \geq a_{max}$ then they are also older than a_{max}. Therefore all items in this subtree must not be included in our badness sum.

This gives, if `n.age` $\geq a_{max}$, `total_b`(n, a_{max}) = `total_b`(`n.left`, a_{max})

Putting it Together

If any of these cases are confusing, it may be helpful to draw out an example and try examining each case before moving on.

This gives the following recursion:

$$\texttt{total\_b}(n, a_{max}) = \begin{cases} \texttt{n.left.sum\_b} + \texttt{n.b} + \texttt{total\_b}(\texttt{n.right}, a_{max}) & \text{if } \texttt{n.age} < a_{max} \\ \texttt{total\_b}(\texttt{n.left}, a_{max}) & \text{otherwise} \end{cases}$$

Thus, `badness`(a_{max}) is simply `total_b`($root, a_{max}$). The runtime is $O(\log n)$ since in each case we recurse left **or** right (but never both), doing constant work at every call. This results in $\log n$ calls, each with $O(1)$ work, so $O(\log n)$.

3 Two Sided Range Queries

This idea can be extended to two sided range queries. For more details, see the solutions to the Holiday Helper problem on Problem Set 3. The recursive idea remains, but we need to store additional information at each node, and proving the runtime becomes more tricky (since there is a case where we recurse both left and right).