CSC263 - Week 4, Lecture 1

Cristyn Howard

Monday, January 29, 2018

Augmenting Data Structures

Think of data structures we have learned about in class as tools to solve problems. Sometimes, the data structures we have learned about will not be perfectly suited to the problem you are trying to solve. In that case, we must modify, or *augment* given data structures so suit our needs. Creative problem solving skills are required when choosing how to do this.

Dynamic Order Statistics - CLRS 14

Given: set of distinct keys S

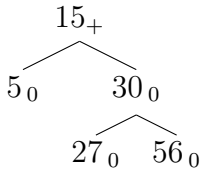
Do: INSERT, DELETE, SEARCH

new SELECT(k): return k^{th} key in sorted order of keys

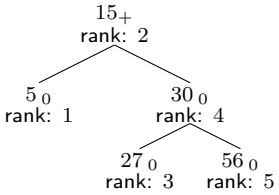
new RANK(x): return position of x in sorted order of keys

Let $S = \{5, 15, 27, 30, 56\}$.

Storing S in an AVL tree, this would allow us to perform INSERT, DELETE, and SEARCH, but it is not immediately obvious how to perform RANK and SELECT. What are some modifications we can make to an AVL tree that would allow us to implement these methods?

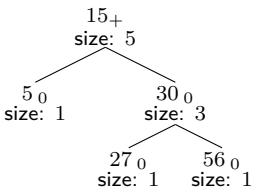


TRY: storing rank in every node.
This allows us to do RANK & SELECT easily.
However now INSERT (sometimes) requires us to update the ranking of all n nodes in the heap, so it is $O(n)$, and thus inefficient.



TRY: at each node x, store the size of the subtree rooted at x.
Property: x.size = x.left.size + x.right.size + 1

Can we use this information to implement RANK and SELECT while preserving the efficiency of INSERT, DELETE, and SEARCH?
Consider the relative ranking of the nodes contained in the subtree rooted at x.
x.rRank = x.left.size + 1, as the i nodes in x.left fill rank 1 through i.



Implementing SELECT(T, k):

- rootRank = T.root.left.size + 1;
- if rootRank = k: return T.root
- if rootRank > k: SELECT(T.root.left, k) *Look for k^{th} element in left subtree.*
- if rootRank < k: SELECT(T.root.right, k − rootRank)
- *This recursive algorithm is limited in it's number of constant time calls by the height of the tree, thus it is $O(\log n)$.*

Implementing Rank(T, x):

- *Note that the explicit code for this algorithm is available in the textbook and may be added later.*
- First, find the rank of x in the smallest tree in which it is contained.
- Then move up through ancestor nodes to the root, making the following adjustments:
  - When moving from LEFT SUBTREE to parent:
    - rank stays the same
  - When moving from RIGHT SUBTREE to parent:
    - must add nodes smaller than X in parent's left subtree to rank
- These operations happen in constant time, and the number of rank alterations that occurs for node x is limited by the height of the tree, so it is $O(\log n)$.

INSERT only requires that we update the size of trees once the AVL has finished balancing, which happens in constant time and does not impact the worst-case run time.

This augmentation is ideal because the information needed to determine the size of a tree is limited to the size of it's subtrees. This means that tree size calculations (and this RANK calculations) is easily accessible given local data.