Lecture 7, Oct 2, 2023

Augmenting Data Structures

- Sometimes we need to modify an existing data structure to perform additional operations; this is called *augmenting* the data structure
  1. Determine which additional info to store for your operations
  2. Check that this additional information can be cheaply maintained during each operation – don't accidentally change the runtime complexity of the structure's operations!
  3. Use the additional information to efficiently implement the new operations you need
- Example: Dynamic Order Statistics: maintain a dynamic set S of elements with distinct keys, supporting search, insert, delete, in addition to:
  - Select(k): find the element with rank k, i.e. the kth smallest element in S
  - Rank(x): determine the rank of the element x, i.e. its order in the set
  - We can augment an AVL tree!
    1. At each node x, we will store the size of the subtree rooted at x
       * For every other node, the size is the sum of the sizes of its two children plus one
       * For leaf nodes the size is 1, for nil nodes the size is 0
    2. This information is cheap to maintain, because to update the size of a node, we simply set it as the sum of the sizes of its children plus one
       * On insertion, increase the size of every parent node by 1, all the way up till the root, opposite on deletion
       * On a rotation, update the size of each node that underwent a rotation as the sum of the sizes of its children plus one
    3. We can use this information to efficiently implement Select(k) and Rank(k):
       * Observation: if a node has n nodes in its left subtree, then there are n nodes smaller than it in the left subtree, so it has *relative* rank $n + 1$ in its own subtree
       * For Select(k):
         - The rank of the current node is the size of the left subtree plus one
         - If k is equal to the current rank, return the current node since it has the correct rank already
         - If k is less than the current rank, recurse on the left subtree
         - If k is greater than current rank, recurse on the right subtree, but instead of k, the rank we want is now k minus the current rank
         - This has complexity $O(h) = O(\log n)$ since in the worst case it goes down the entire tree
       * For Rank(x):
         - If the current node is x, return the current rank
         - If the key at x is less than the key at the current node, recurse on the left subtree
         - If the key at x is greater than the key at the current node, recurse on the right subtree and return the result plus the current rank
         - This has complexity $O(h) = O(\log n)$ since in the worst case it goes down the entire tree
  - Focus on the modifications that you made to the original data structure, because it is assumed that you already know how the original operations work!