CSC263 Monday Lecture Jan 29, 2018

AUGMENTING DATA STRUCTURES

Ex. Dynamic order statistics (CLRS 14)

Given: S - set of distinct keys

Do: INSERT, DELETE, SEARCH,

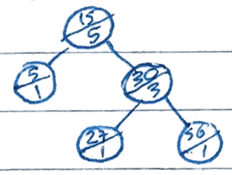
SELECT(k): return k^{th} key in sorted order of keys

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

- 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 to fit our needs. There is no one way to do this, it is a problem-solving skill.

Ex. S = {5, 15, 27, 30, 56}; SELECT(4) = "30"; RANK(15) = "2"; etc.

- Were we to store S in an AVL Tree, this would allow us to perform INSERT, DELETE, SEARCH, but it is not easy to perform RANK, SELECT;
- Try: storing rank in every node; allows us to do rank, select.
  - Now 'INSERT' requires us to update the rank of every node in the tree, so O(n). Not an efficient solution.
- Try: at each node x, store size of subtree rooted at x.



Can we use this info to implement RANK, SELECT while preserving the efficiency of INSERT, DELETE, SEARCH?

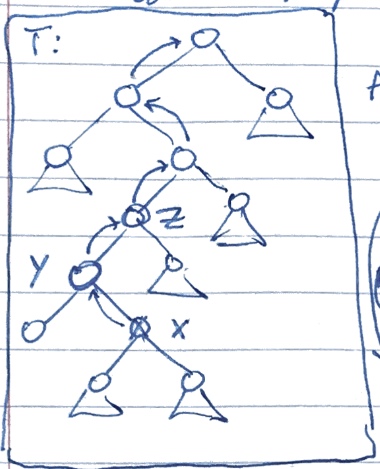
Invariant: SIZE(X) = SIZE(LEFT(X)) + SIZE(RIGHT(X)) + 1

- In the context of subtree rooted at x (ie only nodes in subtree at x); the rank of X, RR(X) = SIZE(LEFT(X)) + 1
- When asking for rank k in subtree at x (select(x, k):)

```
case (i) k < RANK(X): recurse SELECT(LEFT(X), k);
case (ii) k = RANK(X): return x;
case (iii) k > RANK(X): recurse SELECT(RIGHT(X), k - RANK(x));
```

→ O log(n)

- How do we implement RANK(T, x) on this augmented DS?



We can find the rank of X and use it to find rank

$\text{RANK}(X, x) = \text{SIZE}(\text{LEFT}(X)) + 1$

$\text{RANK}(Y, x) = \text{RANK}(X, x) + \text{RANK}(Y, y)$

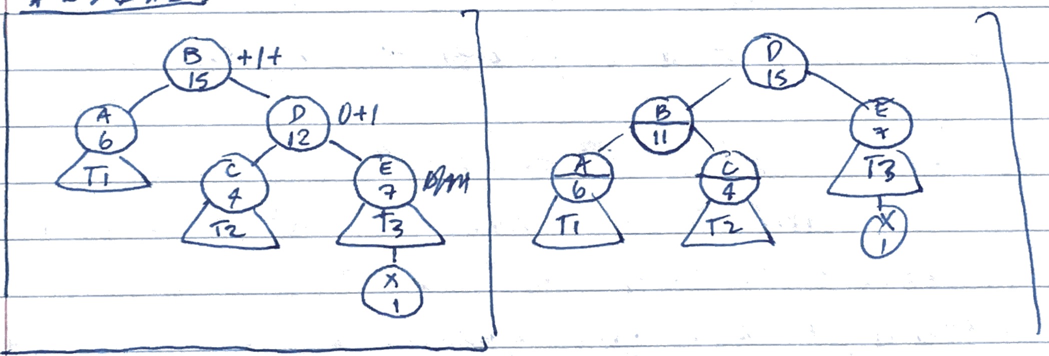
$\text{RANK}(Z, x) = \text{RANK}(Y, x)$

- when moving from left subtree to parent, rank consta
- when moving from right subtree to parent, must add nodes smaller than x in parent's left subtree.

- These operations happen in constant time at max log(n) times.
- Code for these operations is in the textbook.

- <u>INSERT</u>: increment size of all ancestor nodes, if AVL tree rotates you must update sizes after (happens in constant time).

EX ↳ - ~~DELETE~~:



- re-calculating sizes easy because only need size of children