Computer Science 136
Data Structures
Lecture #22 (November 15, 2021)

1. Questions?

2. Recall: Binary Search Trees.

   (a) An implementation of an `OrderedStructure`: `add`, `remove`, `get`, `contains`, `iterator`.

   (b) When locating values:

      i. We return first equal value found.

      ii. All values to the left of the root are smaller.

      iii. All values to the right of the root are larger.

   (c) Duplicate values are stored to left.

   (d) `removeTop` removes the top node of a (sub)tree. It has to be done with care. Several cases:

      i. If top has no left, use right as new root. Similarly if no right.

      ii. If left has no right, stick right under left, use left as root.

      iii. Otherwise, bring predecessor of root up as new root.

3. Splay Trees. A greedy approach to keeping trees balanced.

   (a) A review of rotations. (See Figure 14.4)

   (b) We splay (split) the tree "at a node," `x`. This is done by making the accessed node the root.

      i. If `x` is at the root, we're done.

      ii. If `x` is a child, perform the appropriate rotation of the tree *at the root*, bringing the node to the top.

      iii. Otherwise, `x` is at least depth 2. Find the parent (`p`) and grandparent(`g`) of the node. (Follow with Figure 14.5.)

         A. If `x` is the left child of a left child, then (1) rotate right about the `g` (raising node), and then (2) rotate right about `p` (raising `x` again). (Similar manipulations are performed if node and parent are both right children.) The opposite ordering of rotations seems to work, but does not provide the necessary performance guarantees.

         B. If `x` is the right child of a left, then (1) rotate left about parent (raising node), and then (2) rotate right about grandparent (raising `x` again). (Similar manipulations occur in mirrored circumstance.) Notice that the opposite ordering does not guarantee progress.

      iv. Repeat these various rotations until `x` becomes the root. (Note that progress is made at every step.)

      v. Notice that splaying the tree typically requires re-rooting a tree. The `splay` operation should return the ideal root.

   (c) For `add` and `contains`: splay the tree at the end of the operation.

   (d) For `remove`, we splay at the node's parent (if there is one).

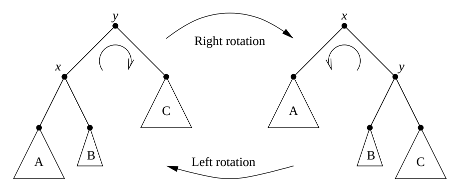

FIGURE 14.4: TREE ROTATIONS

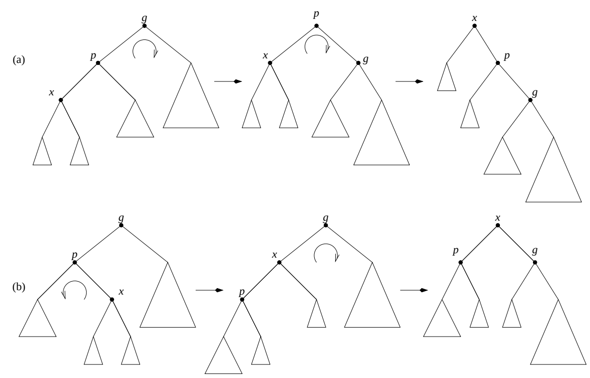

FIGURE 14.5: TREE SPLAYS

4. Red-Black Trees.

   (a) An example that manages the balance of a tree using an accounting mechanism.

   (b) Rules of engagement: All nodes are red or black (EMPTY is black).

      i. All red nodes must have two black children.

      ii. All leaves must have two black children.

      iii. All paths from a node to its leaves mention same number of black nodes.

   (c) Result: properly managed trees must not have leaf heights that differ by a factor of more than two. Therefore, tree has height $O(\log_2 n)$

   (d) When tree changes, we must spend (a little) time fixing up the colors. Details are complex, and discussed in the book.

⋆