Lecture 5, Sep 25, 2023

Balanced BSTs – AVL Trees

- We have learned previously that the runtime complexity of BST algorithms are dependent on the tree height, so in the worst case they can run in $O(n)$ time for an extremely unbalanced tree
  - In the best case the height scales as $\Theta(\log n)$; how can we achieve this?
- Note height is the number of edges in the longest path from a node to a leaf; the height of a tree is defined as the height of the root
  - A trivial tree with only a root node has a height of 0
  - We shall define the height of an empty tree as -1

Definition

The *balance factor* (BF) of a tree is the height of its right subtree minus the height of its left subtree.

In an *AVL tree*, the balance factor of every node is between -1 and +1, inclusive.

- Intuitively we would like the BF of every node to be close to 0, but we can't quite get there since the tree isn't complete
- We can get pretty close by using an AVL tree (named after Adelson-Velski-Landis)
- If the BF is less than 0, then we say it is left-heavy; if BF is greater than 0, it is right-heavy; otherwise it is balanced
- Properties of AVL trees:
  - An AVL tree of n nodes has height $\Theta(\log n)$ (less than $1.44 \log_2(n+2)$)
  - Given an AVL tree, we can do inserts and deletes while maintaining the tree's balance in $\Theta(\log n)$ time
- Practically, to store each node in memory we have a pointer to the parent and both children, as well as a key and balance factor
- For $\textsc{Search}(T, x)$ there is no difference from the regular BST search
- For $\textsc{Insert}(T, x)$, first insert as in a regular BST, and then apply a series of rotations to restore balance
  - General idea: by inserting a node, the balance factor of all its parents will potentially change by 1
    * We update the balance factors by going from the newly inserted node back to the root
      - If we're coming from a right child, we increase the BF by 1; if we're coming from a left child, we decrease it by 1
    * If now the balance factor is no longer in the range $[-1, 1]$, then we rebalance by applying rotations
  - Note if we change a balance factor to 0, it does not actually change the height of the node or any of its parents (since we just balanced it), so we can stop going up the chain
  - If we encounter a right-heavy node with a right-heavy right child, we will left-rotate
    * After a rotation, the BF of the nodes that are not involved in rotations remain the same
    * The node that we applied the rotation to will have its BF reduced by 2; its former right child will have its BF reduced by 1
    * After a rotation, the height of the tree does not change, so we can stop going up the tree to update BFs
  - If we encounter a right-heavy node with a left-heavy right child, we will first right-rotate on its right child (to make the right child right-heavy), and now left-rotate on the parent
  - In the symmetric cases of left-heavy node, left-heavy left child, or left-heavy-node, right-heavy left child, the operations are symmetric