CSE 100

- AVL trees and AVL rotations
- Insert in AVL trees

Reading: Weiss Ch 4, sections 1-4
The importance of being balanced

- A binary search tree has *average*-case time cost for Find = $\Theta(\log N)$, but the probabilistic assumptions leading to that result often do not hold in practice.
- But a *balanced* binary search tree has *worst*-case time cost for Find = $\Theta(\log N)$, which is much better than $\Omega(N)$ when $N$ is large.
- So, it would be nice if our search trees were balanced. How to achieve this?
  - There are two kinds of approaches to this:
    - Deterministic methods
      - guarantee balance, but operations are somewhat complicated to implement
    - Randomized methods
      - operations are simpler to implement; balance not absolutely guaranteed, but achieved with high probability
  - We will look at AVL trees as an example of a deterministic balanced binary search tree approach.
AVL trees

• Named for its inventors Adelson-Velskii and Landis, this is the earliest balanced search tree approach [1962]
• An AVL tree is a binary search tree with this balance property:
  • For any node X in the tree, the heights of the left and right subtrees of X differ by at most 1
• From the AVL balance property, you can prove these facts:
  • The number of nodes on a path from the root to a leaf of an AVL tree with N nodes is at most $1.44 \log_2(N+2)$, compared to at most N for an ordinary BST
  • The average level of a randomly constructed AVL tree with N nodes is asymptotically very close to $\log_2 N$, which is slightly better than the approximately $2 \ln N = 2 \ln 2 \log_2 N = 1.386 \log_2 N$ for a randomly constructed ordinary BST
• The trick is to implement efficient Insert and Delete operations that ensure that the BST ordering and AVL balance properties are invariant (they are both pre- and postconditions of these operations)
• We will look at what is required for the Insert operation...
AVL rotations

• The key operation in many binary search tree balancing algorithms is the **AVL rotation**
  • Red-black trees, splay trees, treaps, and of course AVL trees all use this
• A crucial nice property of AVL rotations is that they leave the BST structural and ordering properties invariant: performing any AVL rotation on a BST will give you back a BST
• AVL rotations also move some nodes in the tree closer to the root, which can be used to improve balance, if you are careful about where to apply the rotations
• An AVL rotation is fast: constant time, requiring only following and changing a small constant number of pointers
• AVL tree insert and delete operations use AVL single rotations and also double rotations (the double rotations are just two AVL single rotations in sequence)
• We will look at these rotations, why they work, and how to implement them
AVL rotations

- AVL rotations move subtrees, while preserving BST ordering

```
   x
  /   
 y     y
   /   
  a   c
   

   y
  /   
 x     x
  / 
 a   b
   

   y
  /   
 x     x
  / 
 a   b
   
```

---

```
   "single left rotation"
   (with right child)

   "single right rotation"
   (with left child)
```
The AVL balance property

- The tree on the left has the AVL balance property.
- The tree on the right has inserted a new node using the usual Insert algorithm; this has destroyed the AVL balance property.
- Which node was inserted? And which nodes now no longer have the AVL property?
Preserving the AVL balance property

• Suppose you start with a binary search tree that has the AVL balance property.
• First, perform an insertion in the usual way: this creates a new leaf node.
• If that insertion destroyed the AVL property, the only nodes that no longer have the AVL property are on the path from the new leaf to the root (since those are the only nodes which have had any of their subtrees changed by the insertion).
• So, only nodes on that path need to be rebalanced to restore the tree’s AVL property.
• We will see that in fact, from that leaf toward the root, only the first node failing the AVL property needs to be rebalanced to restore the AVL property to the whole tree.
• One of these operations will suffice to rebalance any AVL node after an insertion:
  • single left rotation
  • single right rotation
  • double left rotation
  • double right rotation
Restoring the AVL balance property

- Here’s the basic AVL insert algorithm:
  - Perform the insertion in the usual BST way
  - Moving back up from the newly inserted leaf toward the root, note the first node that no longer has the AVL balance property. (If all still have the property, you’re done!) Call this node X.
  - There are 4 possible places the insertion happened. Depending on the case, perform the appropriate rotation to restore the balance property:
    - (1) Insertion was in the left subtree of the left child of X: single rotation with left child (a.k.a. single right rotation)
    - (2) Insertion was in the right subtree of the left child of X: double rotation with left child (a.k.a. double left-right rotation)
    - (3) Insertion was in the left subtree of the right child of X: double rotation with right child (a.k.a. double right-left rotation)
    - (4) Insertion was in the right subtree of the right child of X: single rotation with right child (a.k.a. single left rotation)
  - We will look at case (1) and (2); case (3) is symmetric with (2), and (4) is symmetric with (1)
Case 1: insertion in the left subtree of the left child of X

- The insertion occurred somewhere in the left subtree of the left child of X.
- Before the insertion, the whole tree had the AVL balance property.
- After the insertion, X is the first node on the path from the newly inserted leaf to the root of the tree that fails to have the AVL balance property.
- We can conclude that:
  - Before the insertion, the height of X’s left subtree was one greater than the height of X’s right subtree (otherwise the insertion would not have destroyed the AVL property at X).
  - Before the insertion, the height of the 2 subtrees of X’s left child were the same (otherwise the insertion would have not destroyed the AVL property at X, or would have destroyed it at X’s left child instead), and these were also the same as the height of X’s right subtree.
  - After the insertion, the height of the left subtree of X’s left child has increased by 1 (otherwise the AVL property would not have been destroyed at X).
Single rotation with left child: a picture
Single rotation to handle Case 1

- Let \( H \) be the height of the subtree rooted at \( X \) before the insertion.
- After the insertion:
  - the right subtree of \( X \) still has height \( H-2 \)
  - the right subtree of the left child of \( X \) still has height \( H-2 \)
  - the left subtree of the left child of \( X \) now has height \( H-1 \)
  - and so the height of the subtree rooted at \( X \) now has height \( H+1 \)
- The AVL property at \( X \) can be restored by a ‘single rotation with left child’:
  1. the left child of \( X \), along with the left child’s left subtree, is made the “new” \( X \)
  2. former node \( X \), along with its right subtree, is moved to be the right child of the new \( X \)
  3. the right subtree of the left child of the former node \( X \) is moved to be the left subtree of the former node \( X \)
- The height of the subtree rooted at the new \( X \) is now again \( H \), so no nodes above \( X \) need to be changed!
- This maneuver restores the AVL property, and preserves the BST ordering property.
Single rotation, after insertion: an example
Single rotations: code

/** Perform single rotation to handle AVL case 1: 
* AVL violation due to insertion in left subtree of left child. 
* @return pointer to the root of the rotated subtree 
*/
BSTNode* rotateWithLeftChild(BSTNode* X) {
    BSTNode* L = X->left;  
    X->left = L->right; 
    L->right = X;       
    return L;
}

/** Perform single rotation to handle AVL case 4: 
* AVL violation due to insertion in right subtree of right child. 
* @return pointer to the root of the rotated subtree 
*/
BSTNode* rotateWithRightChild(BSTNode* X) {
    BSTNode R = X->right; 
    X->right = R->left; 
    R->left = X;       
    return R;
}
Case 2: insertion in the right subtree of the left child of X

- The insertion occurred somewhere in the right subtree of the left child of X
- Before the insertion, the whole tree had the AVL balance property
- After the insertion, X is the first node on the path from the newly inserted leaf to the root of the tree that fails to have the AVL balance property
- We can conclude that:
  - Before the insertion, the height of X’s left subtree was one greater than the height of X’s right subtree (otherwise the insertion would not have destroyed the AVL property at X)
  - Before the insertion, the height of the 2 subtrees of X’s left child were the same (otherwise the insertion would not have destroyed the AVL property at X, or would have destroyed it at X’s left child), and these were also the same as the height of X’s right subtree
  - After the insertion, the height of the right subtree of X’s left child has increased by 1 (that’s what destroyed the AVL property at X)
Double rotation with left child: a picture
Double rotation to handle Case 2

- Let $H$ be the height of the subtree rooted at $X$ before the insertion.
- After the insertion:
  - the right subtree of $X$ still has height $H-2$
  - the left subtree of the left child of $X$ still has height $H-2$
  - the right subtree of the left child of $X$ now has height $H-1$
  - one of the subtrees of the right subtree of the left child of $X$ now has height $H-2$ (but it doesn’t matter which)
  - and so the height of the subtree rooted at $X$ now has height $H+1$
- The AVL property at $X$ can be restored by a ‘double rotation with left child’:
  - first, do a single left rotation rooted at $X$’s left child
  - then, do a single right rotation rooted at $X$
- The height of the subtree rooted at the new $X$ is now again $H$, so no nodes above $X$ need to be changed!
- This maneuver restores the AVL property, *and preserves the BST ordering property*
Double rotation, after insertion: an example
Double rotations: code

/**
 * Perform double rotation to handle AVL case 2:
 * AVL violation due to insertion in right subtree of left child.
 * @return pointer to the root of the rotated subtree
 */
BSTNode* doubleWithLeftChild(BSTNode* X) {
    X->left = rotateWithRightChild(X->left);
    return rotateWithLeftChild(X);
}
/**
 * Perform double rotation to handle AVL case 3:
 * AVL violation due to insertion in left subtree of right child.
 * @return pointer to the root of the rotated subtree
 */
BSTNode* doubleWithRightChild(BSTNode* X) {
    X->right = rotateWithLeftChild(X->right);
    return rotateWithRightChild(X);
}
Implementing AVL operations

- To implement an AVL tree, each node needs to have access to “balance” information
  - each node can have a **height** field, holding the height of the subtree rooted there
  - or, can just hold the difference in height between left and right subtrees of the node
- It is easiest to implement Insert recursively:
  - descend the tree from the root, until reaching the location of the new node
  - create the new node, with height 0, and insert it as a leaf
  - when unwinding the recursion, update height fields of nodes on the path toward the root, and check if a node now violates the AVL balance property
  - if a node does violate the property, perform the appropriate rotation (and update the height fields of affected nodes)
- Implementing a Delete operation is similar: rotations are used to ensure that the AVL balance property is invariant
Next time

- Treaps
- Find, insert, delete, split, and join in treaps
- Randomized search trees
- Randomized search tree time costs