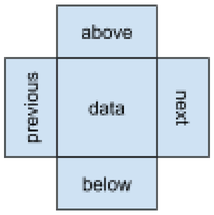
CS 1332 - Week 07

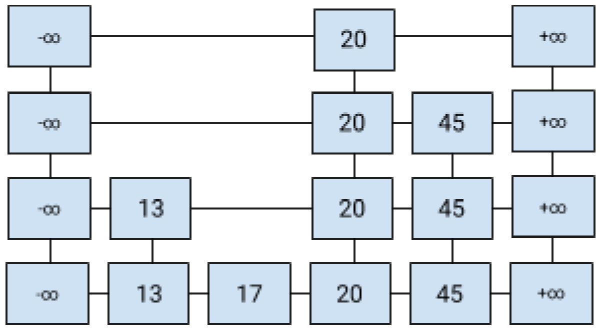
Monday

<u>Topics</u>

I. SkipLists
   A. Structure:
      1. Nodes: SkipList nodes have data and references to previous, next, above, and below nodes



      2. Levels: levels are made up of "linked lists" of nodes, all of the data can be found at the lowest level (level 0), but elements will be promoted at random to higher levels
         a) We'll use a coin flipper to determine if an element is promoted, we'll promote when heads is flipped and stop when tails is flipped:
            (1) T → data is only on level 0
            (2) HT → data is on level 0 and 1
            (3) HHT → data is on level 0, 1, and 2
         b) The probability of data being promoted one level is ½
         c) The probability of data being promoted two levels is ¼
         d) The probability of data being promoted three levels is ⅛, etc.
         e) This probability leads to log n levels
      3. Levels are bounded by negative infinity and positive infinity as the "head" and "tail" → data elements are placed in order at each level
   B. Function/Features:
      1. The goal of a SkipList is to be able to skip over data while traversing to allow us to search for elements very quickly
      2. SkipLists are unique because they employ randomization
   C. Example: add 13, 17, 20, 45 with the coin flips HT T HHHT HHT
      1. 13 (HT) is added at levels 0 and 1
      2. 17 (T) is only added at level 0
      3. 20 (HHHT) is added at levels 0, 1, 2, and 3
      4. 45 (HHT) is added at levels 0, 1, 2



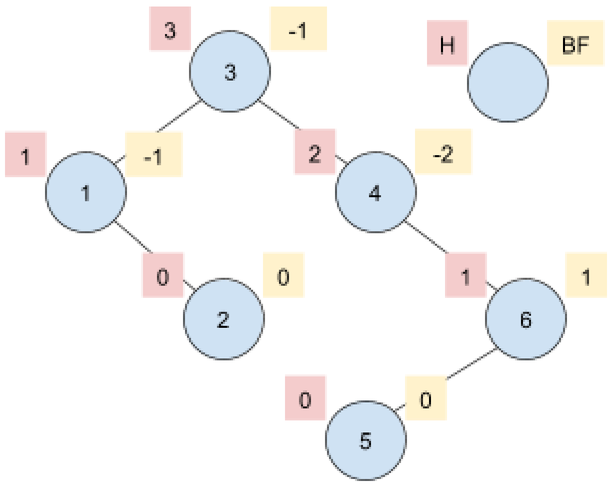
D. Searching/Adding to a SkipList
    1. Starting at the top left negative infinity node as the current node:
        a) Compare the data you want to the data to the right of current:
            (1) If the data to the right is GREATER than the data you want → set current to the node BELOW current
                (a) *When adding:* if below is null, don't update current, the data belongs to the right of current
            (2) If the data to the right is LESS than the data you want → set current to the node to the RIGHT of current
        b) Repeat until you find your data or hit null

E. Efficiencies:
    1. Best/Average case - ½ the data on level 1, ¼ the data on level 2, etc.
        a) Add/Remove/Search - O(log n)
        b) Space - O(n) (similar proof to build heap)
    2. Worst case - all of the data is on every level (the top layer is essentially a linked list and we can't skip over anything)
        a) Add/Remove/Search - O(n)
        b) Space - O(n log n) (all n data on log n levels)

Wednesday

Topics

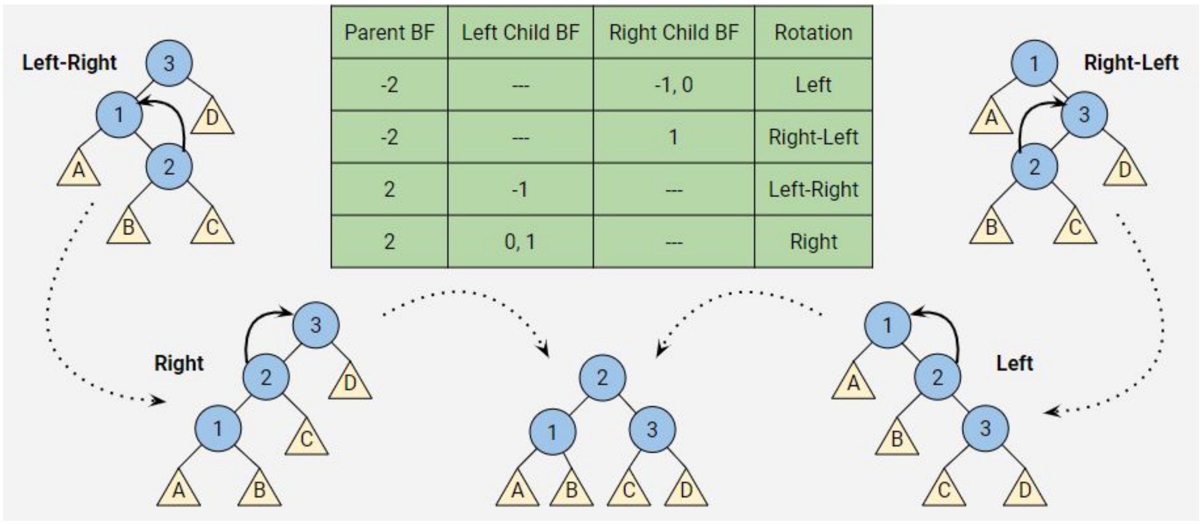
I. AVL Trees
    A. AVLs are a subset of binary search trees (BSTs)
        1. Order property is the same (left subtree < parent < right subtree)
        2. Add and remove are relatively the same
    B. AVLs self-balance to eliminate the worst case degenerate tree
        1. All AVL operations are O(log n) even in the worst case
    C. AVL nodes store the data, left child, right child, height AND balance factor
        1. Balance factor = left child height - right child height



        2. The height of a null node is -1

3. The height and balance factor of any leaf node is 0
    a) Why? when both children are null, BF = (-1) - (-1) = 0
4. If BF < -1 or BF > 1 → the node is imbalanced
5. If -1 <= BF <= 1 → the node is balanced enough
6. If BF < 0 → right heavy
7. If BF > 0 → left heavy

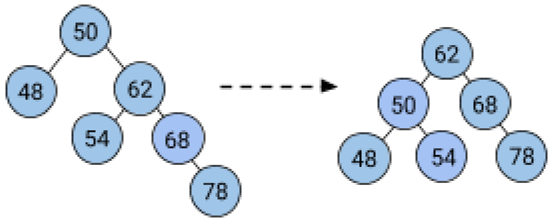
D. Using rotations to fix imbalances



Parent BF	Left Child BF	Right Child BF	Rotation
-2	---	-1, 0	Left
-2	---	1	Right-Left
2	-1	---	Left-Right
2	0, 1	---	Right

1. Left rotation (on 1 in the "Left" diagram)

```
AVLNode rotateLeft(AVLNode a) {
     AVLNode b = a.getRight();
     a.setRight(b.getLeft());
     b.setLeft(a);
     update(a); // a is now a child of b,
     update(b); // updating b relies on updating a
     return b; }
```



2. Right rotation (on 3 in the "Right" diagram)

```
AVLNode rotateRight(AVLNode a){
     AVLNode b = a.getLeft();
     a.setLeft(b.getRight());
     b.setRight(a);
     update(a); // update H & BF of node
     update(b);
     return b; // for pointer reinforcement! }
```

Friday

<u>Topics</u>

I. AVLs

A. Updating the height and balance factor of a node:

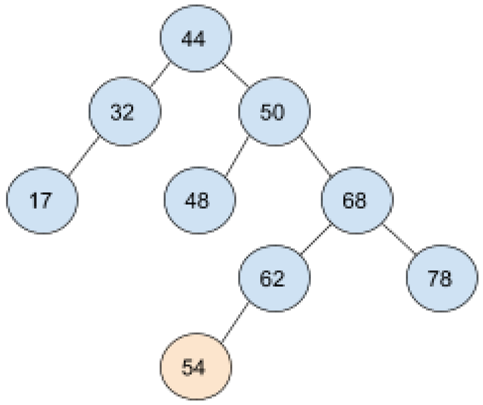
```
private void updateHBF(AVL Node<T> node) {
      node.bf = easyH(node.left) - easyH(node.right);
      node.height = 1 + Math.max(easyH(node.left),
                                 easyH(node.right));
}
private int easyH(AVLNode<T> node) {
      if (node == null) { return -1; }
      else { return node.height; }
}
```

B. Adding

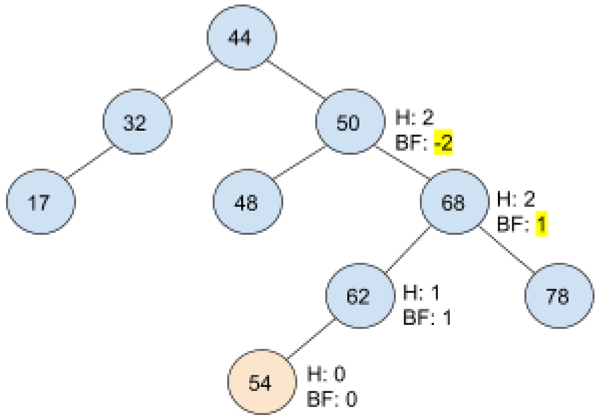
1. Add just like a BST
2. On the way back up the recursive stack (tracing back through the parents up to the root), update the heights and balance factors
3. If the new balance factor is out of balance, perform a rotation

C. Adding Example:

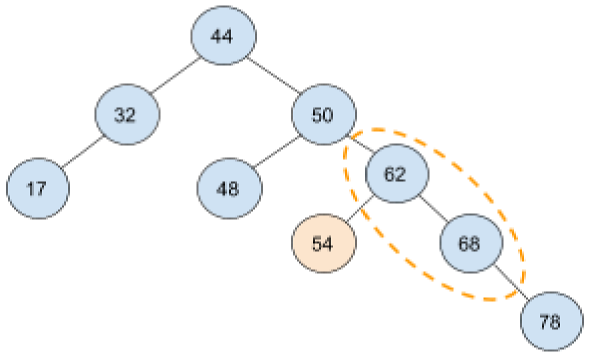
1. Add like a BST:



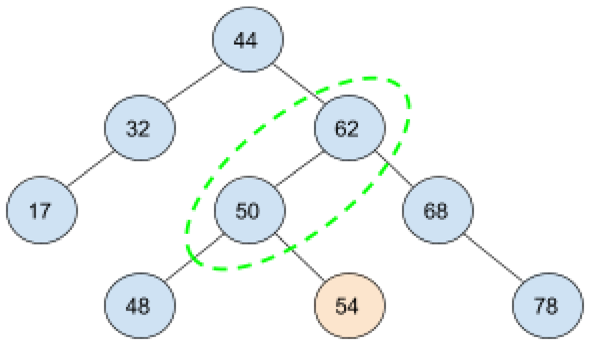
2. Update heights and balance factors:



3. Since 50’s balance factor is -2, but its right child’s balance factor is 1, we’ll need to perform a right-left rotation. First, we perform a right rotation on the right child to “straighten” out the tree:



4. Next, we perform a left rotation of 50 to complete the rotation:



D. Implementing add (put this after your regular pointer reinforcement BST add):
    1. If |bf| > 1
        a) If node is left heavy (bf > 0)
            (1) If node.left is right heavy (bf < 0)
                (a) do a left rotation on node.left
                (b) do a right rotation on node
            (2) Else
                (a) do a right rotation on node
        b) Else if node is right heavy (bf < 0)
            (1) If node.right is left heavy (bf > 0)
                (a) do a right rotation on node.right
                (b) do a left rotation on node
            (2) Else
                (a) do a left rotation on node