Search/Insert

- Retrieval of a particular piece of information from large volumes of previously stored data
- Purpose is typically to access information within the item (not just the key)
- Recall that arrays, linked lists are worst case $O(N)$ for either searching or inserting

*Need data structure with optimal efficiency for searching and inserting*

Properties of BSTs

- Best case (balanced): about $\log N$ nodes between root and each external node/leaf
- Worst case (unbalanced): about $N$ nodes between root and each external node/leaf
- With random data:
  - trees are likely to be well-balanced on average
AVL Tree

defined for Adelson, Velskii, and Landis

- Change worst case search/insert to $O(\lg N)$
- Height Balance Property
  - for every internal node $v$ of $T$, the heights of the children of $v$ differ by at most 1
  - note recursive definition

Search (same as BST)

- To search for a key $k$, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of $k$ with the key of the current node
- If we reach a non-matching leaf, the key is not found and we return a null position
- Example: search(4, root)

Sort (same as BST)

Inorder Traversal

Algorithm inorder($T,v$)
  for left child $w$ of $v$ do
    inorder($T,w$)
  visit node $v$
  for right child $x$ of $v$ do
    inorder($T,x$)

simply stated, sorting an AVL tree is an inorder traversal of the AVL tree
Inserting into AVL Trees

- Each node records its height
- Can compute a node’s balance factor:
  \( \text{bal}(n) = \text{height}(n\text{ left}) - \text{height}(n\text{ right}) \)
- A node that is AVL-balanced:
  - \( \text{bal}(n) = 0 \): both subtrees equal
  - \( \text{bal}(n) = +1 \): left taller by one
  - \( \text{bal}(n) = -1 \): right taller by one
- \( |\text{bal}(n)| > 1 \): node is out of balance

Balance Factors

| bal(n) | | bal(n) |
|--------|----------------------|
| -2     |                      |
| -1     |                      |
| 0      |                      |
| +1     |                      |

Insertion (begins like BST)

- To perform operation `insert(k, e)`, we search for key `k`.
- Assume `k` is not already in the tree, and let `w` be the leaf reached by the search.
- We insert `k` at node `w` and expand `w` into an internal node.
- Example: `insert(5, e)`
Insertion (con’t)
- Check for ‘balance’ after insertion, where balance:
  - node v of T is balanced if 
    \[ \Delta |\text{height(children(v))}| \leq 1 \]
- If balanced after insertion, then done
- Else, rotate to re-balance

Pop Quiz
- AVL-balance the following two trees:
  (hint: think rotations)

Insertion (con’t)
Four Cases (page 431)
- single left rotation (case a)  
  – RL(a)
- single right rotation (case b)  
  – RR(c)
- double rotation (case c)  
  – RR(c)
  – RL(a)
- double rotation (case d)  
  – RL(a)
  – RR(c)
Checking and Balancing

checkAndBal(node *n)
  if bal(n) > +1
    if bal(n->left) < 0
      rotL(n->left)
    rotR(n)
  else if bal(n) < -1
    if bal(n->right) > 0
      rotR(n->right)
    rotL(n)

- Outermost if: is node out of balance?
  > +1: left too big
  < -1: right too big
- Inner ifs: do we need a double rotation? only if signs disagree

Summary: AVL Trees

- Binary Search Tree
  - worst case insertion/search is O(N)
- AVL Tree
  - worst case insertion/search is O(lg N)
  - must guarantee height balance property
- Operations
  - search: same as BST, but O(lg N)
  - sort: same as BST, with O(N)
  - insert: may have to rebalance
  - delete: may have to rebalance