Trees (cont.)

- Basic concepts (Section 4.1)
- Binary trees (Section 4.2)
- Binary search trees (Section 4.3)
- **Balanced trees**:
  - AVL trees (Section 4.4)
  - Splay trees (Section 4.5)
  - B-trees (section 4.7)

Approaches to balancing a BST

- Optimization:
  - Make sure the tree is balanced after every insertion or deletion. Fix it (through rotations) if the balance is broken.
  - e.g., AVL tree.
- Amortization:
  - Do some work now so it can save time later.
  - e.g. splay tree -- every time a node is inserted or searched, bring it to the root (through rotations).
- Randomization:
  - For each node inserted (as a leaf), randomly decide whether to keep it there or bring it to the root (through rotations).
  - e.g., randomized BST.
  - The effect is as if the order of inserted keys were randomized, thus achieving the logarithmic run-time with high probability.

AVL trees

The name AVL came from the names of the inventors -- Adelson, Velskii and Landis.

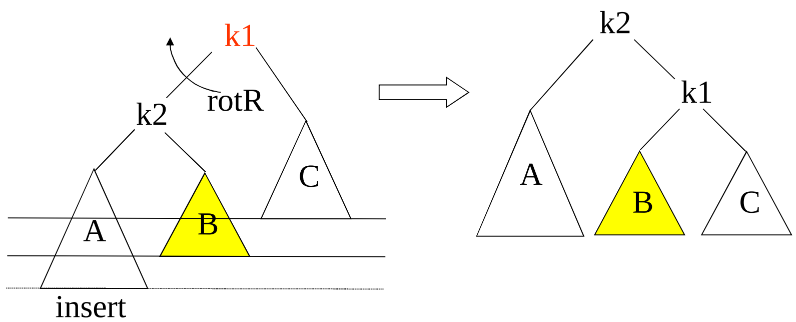
- Property: The heights of the two subtrees of each node differ by at most one.
  - Reminder: the height of a tree is defined as the path length from the root to a farthest leaf.
- An insertion causes either of the subtrees of some node to grow in height by one. If the property is violated as a result, then rotation (either single or double) brings the node back to balance.
- The worst case run-time of a search/insert/remove operation is O(logN).
  - Note: not O(N) as in the BST.

Node rotations in the AVL tree

- If the property is violated as a result of
  - (zig-zig) inserting into the left subtree of the left child, then rotate right (rotR) -- single rotation
  - (zig-zig) inserting into the right subtree of the right child, then rotate left (rotL) -- single rotation
  - (zig-zag) insert into the left subtree of the right child, then rotate right and rotate left bottom up (rotR;rotL) -- double rotation
  - (zig-zag) inserting into the right subtree of the left child, then rotate left and rotate right bottom up (rotL;rotR) -- double rotation

Node rotations in the AVL tree (example)

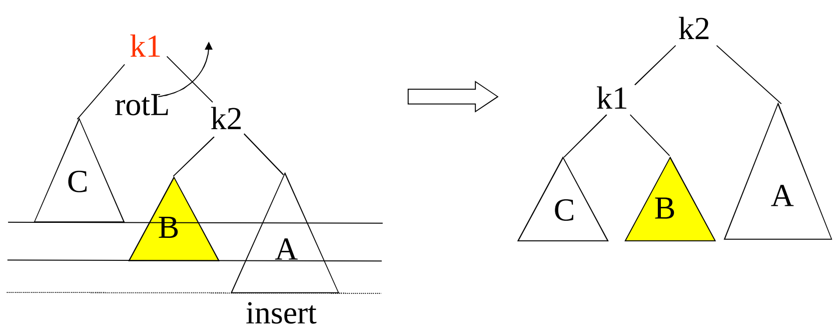
Zig-zig Single rotation right



Assume k1 became unbalanced as a result of the insertion.

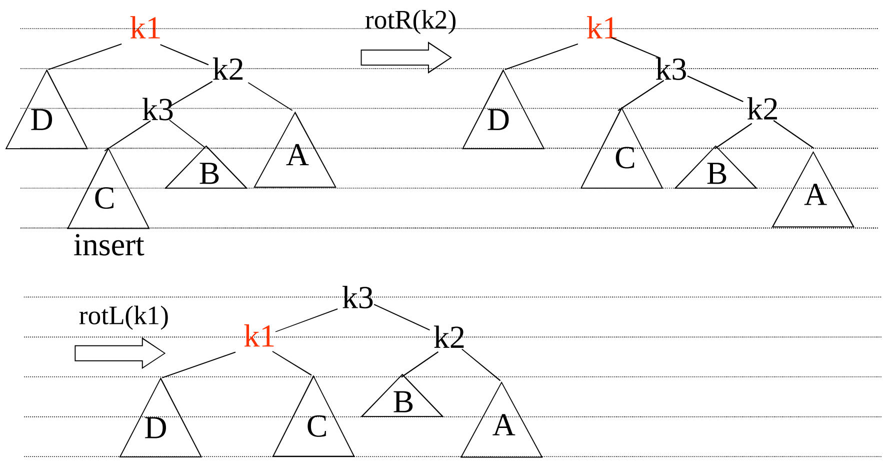
Then, subtree B becomes the left subtree of k1 as a result of rotR.

Zig-zig Single rotation left



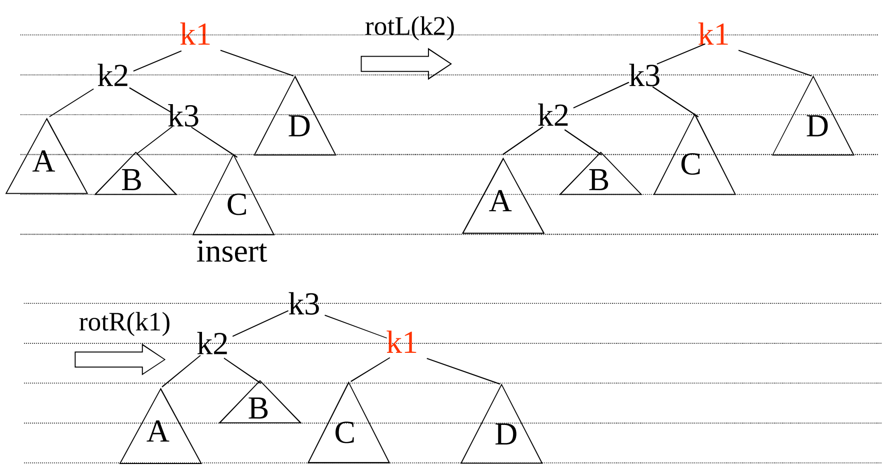
This case is symmetric to the previous zig-zig case.

Zig-zag Double rotation right & left



Side note: Insertion may be into B or C. It does not matter.

Zig-zag Double rotation left & right



This case is symmetric to the previous zig-zag case.

Exercise: AVL tree construction

- Construct an AVL tree by inserting nodes with the following keys: C, B, E, F, J, D, G, and H.
  - By convention, the height of an empty tree (i.e., null link) is -1.

(See the answer key.)

Adapted from Weiss

```
/** Insert a node with key x into a tree with root t. */
Node insert(Item x, Node t) {
     if(t == NULL) t = new Node(x, NULL, NULL);
     else if(x < t.element) {
         insert(x, t.lchild);
         if (height(t.lchild) - height(t.rchild) == 2)
             if(x < t.lchild.element)
                // x has been inserted into the left subtree of t.lchild,
                // so zig-zig.
                singleRotateWithLeftChild(t);  // rotR(t)
             else
                // x has been inserted into the right subtree of t.lchild,
                // so zig-zag.
                doubleRotateWithLeftChild(t); // rotL(t.lchild) & rotR(t)
     } else if(x > t.element) {
         insert(x, t.rchild);
         if(height(t.rchild) - height(t.lchild) == 2)
             if(x > t.rchild.element)
                 // x has been inserted into the right subtree of t.rchild,
                 // so zig-zig.
                 singleRotateWithRightChild(t);  // rotL(t)
             else
                 // x has been inserted into the left subtree of t.rchild,
                 // so zig-zag.
                 doubleRotateWithRightChild(t);  // rotR(t.rchild) & rotL(t)
      }  else ; // Duplicate; do nothing
      t.height = max(height(t.lchild), height(t.rchild)) + 1;
 }
```

Running time of AVL tree operations

- Worst case run-time for a single node search, insertion, and deletion is O(logN).

Sketch of proof: It suffices to prove that, for an AVL tree of N nodes with height h, h = O(logN). Let S(h) be the minimum possible number of nodes in an AVL tree of height h. Then,

$S(h) = S(h-1) + S(h-2) + 1$ for $h \geq 2$; $S(1)=2$; $S(0)=1$. (Why?)

  - (Why?) Heights of the two subtrees satisfying the AVL property can be either h-1 and h-1 or h-1 and h-2. The number of nodes is smaller in the latter case.

Note the similarity between S(h) and Fib(h).

Knowing that $Fib(h) \approx \phi^h/\sqrt{5}$ (golden ratio $\phi = 1.61803...$), we can derive $S(h) \approx \phi^h/\sqrt{5}$. (A formal proof of this is omitted.)

$S(h) \leq N$ by the definition of S(h). So, $h \leq \log_\phi(\sqrt{5} \cdot N) = O(\log N)$.