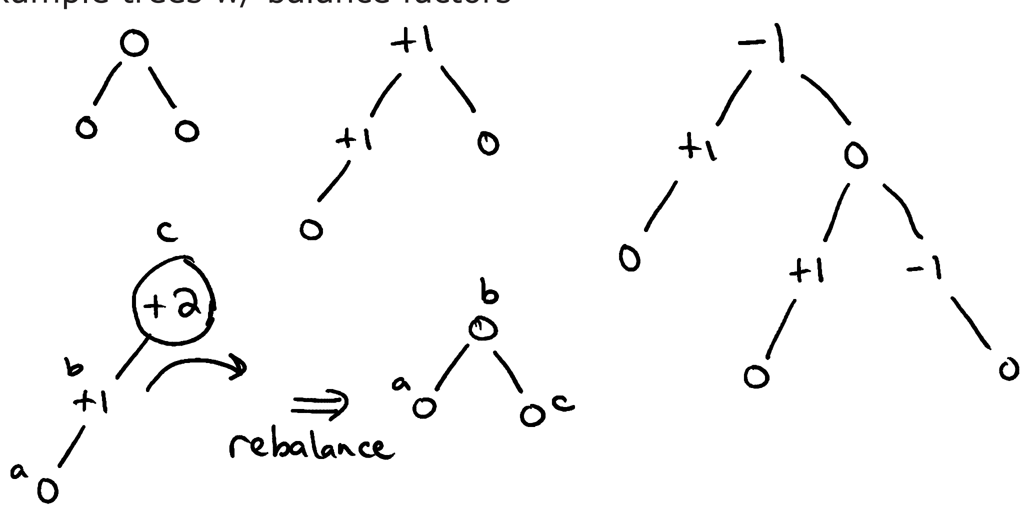
AVL Trees

Wednesday, October 31, 2007
8:02 PM

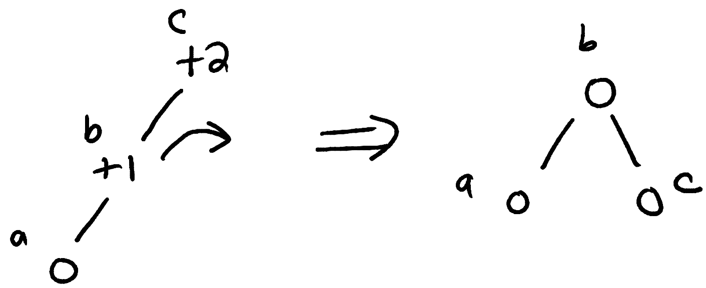
- 15.2 Tree Balancing: AVL trees
  - Order of insertion into binary search tree greatly affects balance
    - best order results in balanced tree
    - worst order results in linked list (lopsided tree)
  - AVL trees are a solution
    - named for creators, Russian mathematicians in the 1960s
      - Georgii Maksimovich Adel'son-Vel'skii
      - Evgenii Mikhailovich Landis
    - height-balanced tree
    - specialized binary search tree that has a balance factor
      - balance factor reflects the height difference of a node's subtrees
      - balance factor is calculated by taking height of left subtree and subtracting height of right subtree
      - balance factor is only allowed to be -1, 0 or 1
        - keeps height difference to at most 1
        - tree must be rebalanced when balance factor exceeds these values
  - AVL Tree ADT
    - Member variables
      - a binary search tree that maintains the balance factor
    - Basic Operations
      - use the constructor, empty(), search() and traversals from BST
      - insert an item & rebalance if needed
      - delete an item & rebalance of needed
  - AVL tree node
    - need to add a member variable for balance factor
    - so have data, balance factor and pointers to left & right children
  - Example trees w/ balance factors



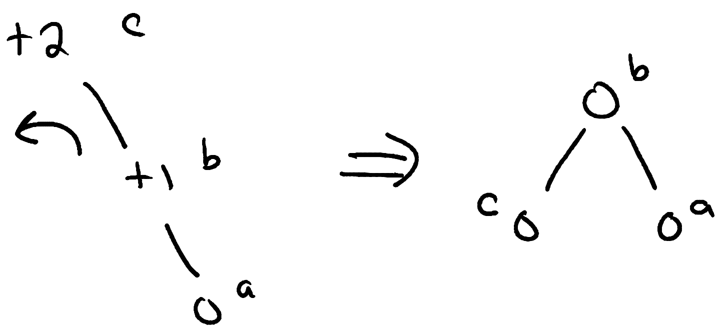
Rebalance Rotations

4 rotations to restore balance factor

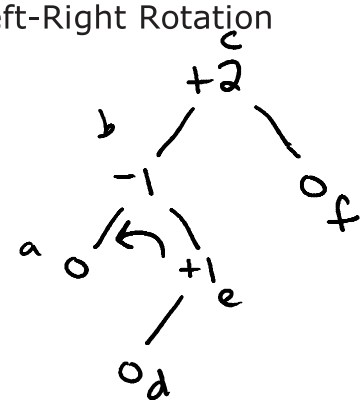
Right Rotation



Left Rotation

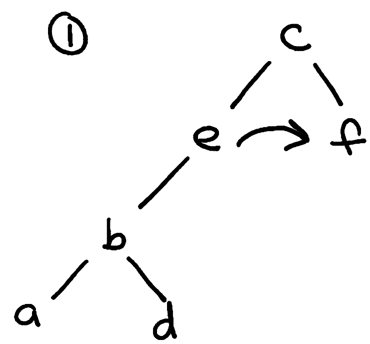


Left-Right Rotation

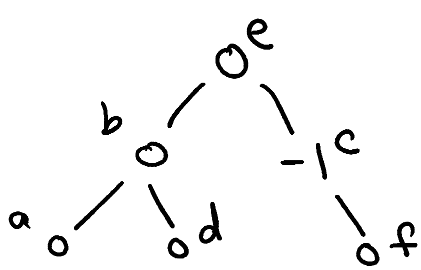


2 rotations needed

①

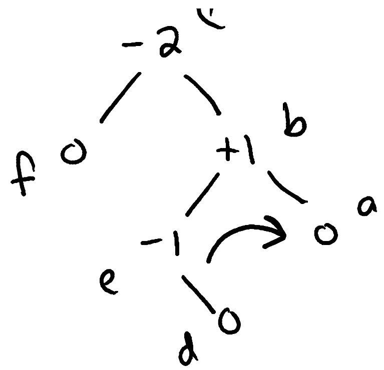


② Restores balance factor

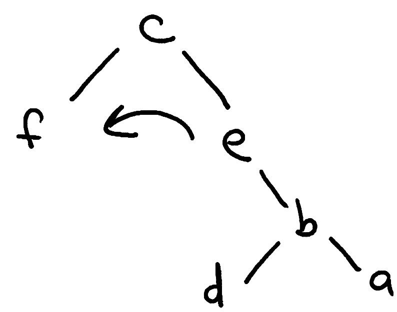


Right-Left Rotation

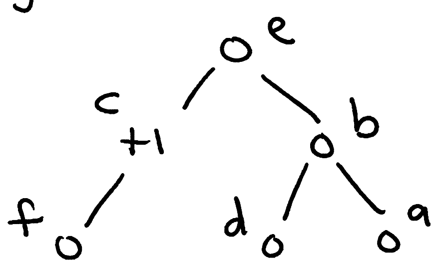
Again, 2 rotations



①

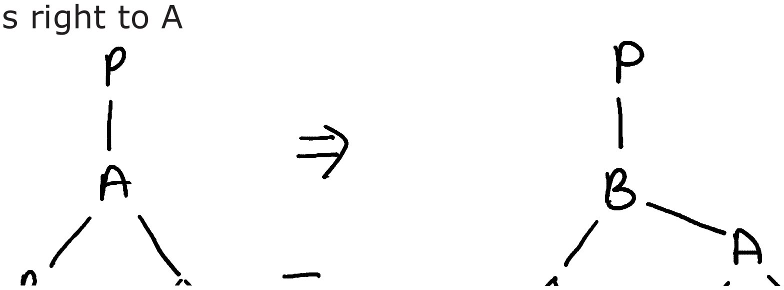


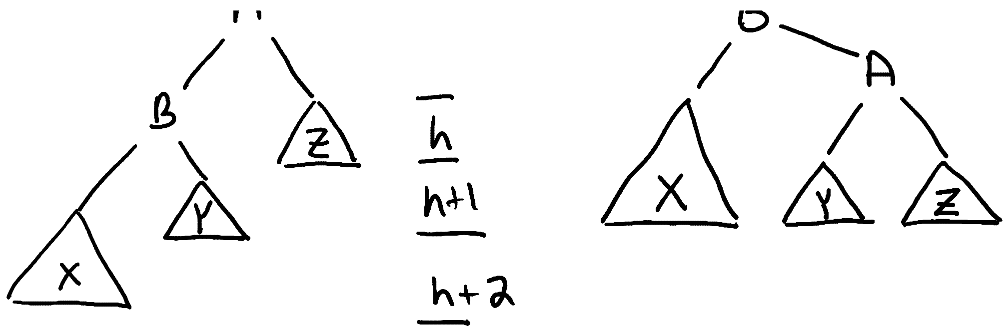
② Again, restores balance factor



Rotation Details - Insertion case
- Apply rotation when node's balance factor is +2 or -2 & is nearest ancestor to inserted node
- Cases:
  - Right rotation
    - inserted node is in left subtree of left child of unbalanced node (+2)
  - Left rotation
    - inserted node is in right subtree of right child of unbalanced node (-2)
  - Left-right rotation
    - inserted node is in right subtree of left child of unbalanced node (+2)
  - Right-left rotation
    - inserted node is in left subtree of right child of unbalanced node (-2)

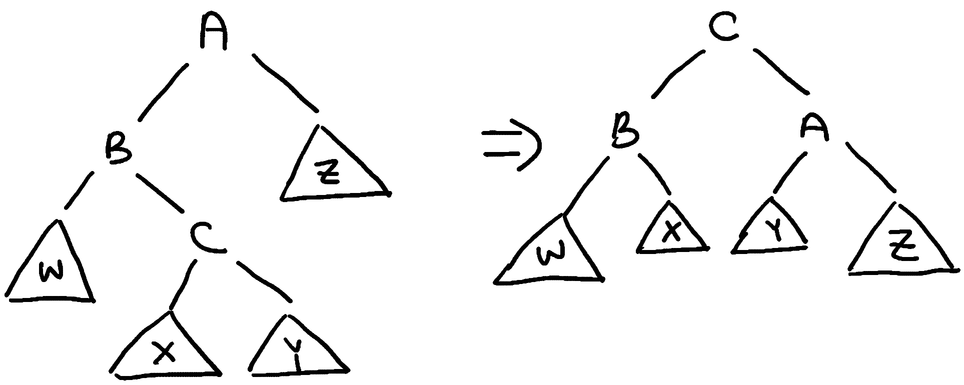
Rotation Pseudocode
- Right Rotation
  - A is unbalanced node
  - B is left child
  - set parent of B to A's parent
  - set parent of A to B
  - set A's left to B's right
    - (value in B's right is between value of A & value of B)
  - set B's right to A



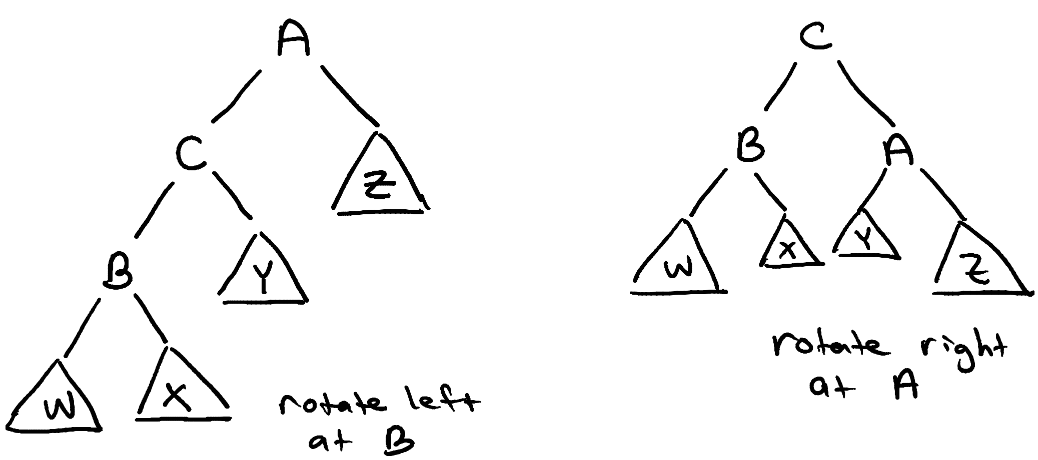


Left Rotation
- A is unbalanced node
- B is right child
- set parent of B to A's parent
- set parent of A to B
- set A's right to B's left
- set B's left to A

Left Right Rotation



via two steps



rotate left at B (node A's left child)
rotate right at node A
Alternate Method:
- set C's parent to A's parent
- set A's parent to C
- set B's parent to C
- set B's right to X (C's left)
- set A's left to Y (C's right)

- set C's left to B
- set C's right to B

Right-left rotation

- rotate right at B (node A's right child)
- rotate left at node A

Rotation on Deletion

- more difficult notations than on insertion
- can delete nodes & leaves

Runtime

- since tree is balanced, searches are O(log2n)
- overhead to rebalance
  - increases inserts delete runtime
  - studies show 45% of inserts require rotations
    - approx half are double rotations
- if searching is primary operation, fast search outweighs slower insert