Balanced Binary Search Trees

a.k.a AVL Trees

CS 3358
Summer II 2013

Jill Seaman

Section 19.4

Binary Search Trees

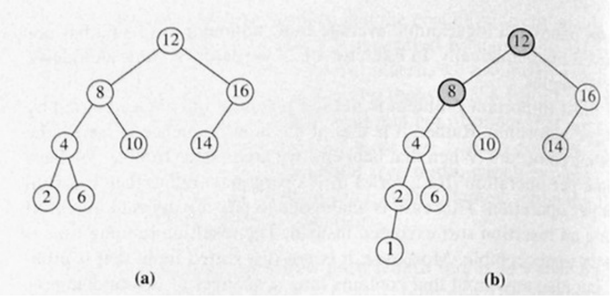
- Problem:
  - when the nodes get too deep in the tree, operations take longer than O(log N)
  - this happens when the tree is taller than it is wide
- Solution:
  - keep the trees balanced so their height remains (O(log N)).

AVL Tree:

- **AVL Tree**:
  - A BST with the added property that for **each** node in the tree, the height of the left and right subtrees differ by at most 1
- **Note**: the height of an empty subtree is -1
- The balance information (height of left subtree - height of right subtree) can be computed and stored at each node.
  - this value must be -1, 0 or 1 for each node

AVL Tree:

example



unbalanced nodes are darkened

- (a) is an AVL tree
- (b) after inserting 1, it is not an AVL tree
- What if you insert 13? does it become balanced?

AVL Trees

- Searching is O(log n) for AVL trees
  - the height is O(log n)
- insert and remove are complicated
  - they may put the tree out of balance
  - must re-balance the tree before operation is complete.

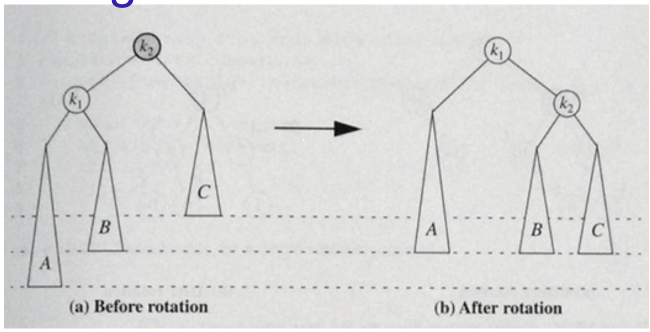
Rebalancing

- After insertion, only the nodes on the path from insertion to root might have their balances altered.
- We fix the balance of the first (deepest) node on the path to the root, and this rebalances the entire tree.
- Balance is restored by a **tree rotation**.
- A single rotation switches the roles of the parent and child while maintaining the search order (BST property).

Rebalancing

- If node X is balanced, then as a result of an insert, X becomes unbalanced, we have only the following possibilities for where the insert happened:
  - 1. into the left subtree of the left child of X. (left-left)
  - 2. into the right subtree of the left child of X. (left-right)
  - 3. into the left subtree of the right child of X. (right-left)
  - 4. into the right subtree of the right child of X. (right-right)
- 1 and 4 are mirror images of each other.
- 2 and 3 are mirror images of each other.

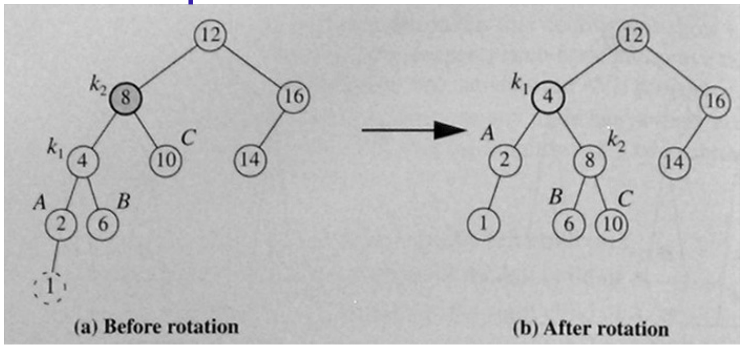
Single Rotation for case 1



(a) Before rotation (b) After rotation

- k2 is now unbalanced, after insert into A
- A < value(k1) < B < value(k2) < C
- make k2 the right child of k1.
- make B the left subtree of k2

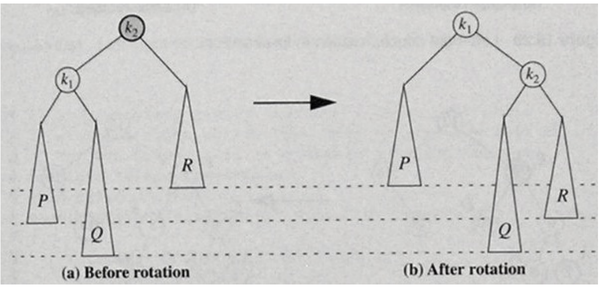
Example: Rotation for case 1



(a) Before rotation (b) After rotation

- k2 is unbalanced, after insert of value 1
- make k2 the right child of k1.
- make B the left subtree of k2

Single rotation does not fix case 2

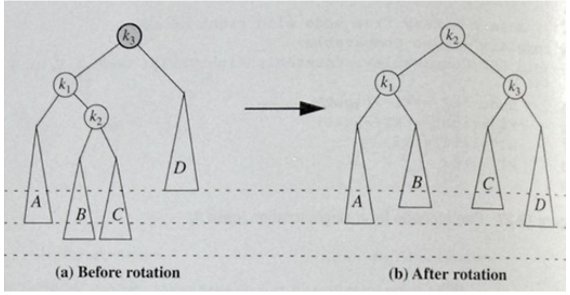


(a) Before rotation (b) After rotation

- k2 is unbalanced, after insert into Q
- P < value(k1) < Q < value(k2) < R
- Problem: still unbalanced after single rotation!

Double Rotation

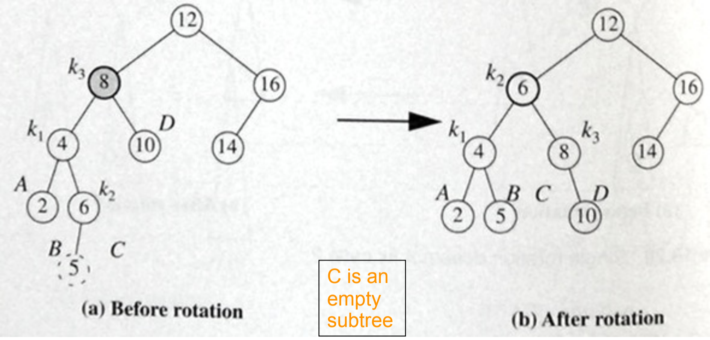
Same case as previous slide, split Q into B and C



(a) Before rotation (b) After rotation

- k3 is unbalanced, after insert into B or C
- A < value(k1) < B < value(k2) < C < value(k3) < D
- make k1 the left child of k2, B becomes right child of k1.
- make k3 the right child of k2, C becomes left child of k3

Example: Rotation for case 2



(a) Before rotation

C is an empty subtree

(b) After rotation

- k3 is unbalanced, after insert of value 5
- make k1 the left child of k2, B becomes right child of k1.
- make k3 the right child of k2, C becomes left child of k3