Tree Balance

Seth Long

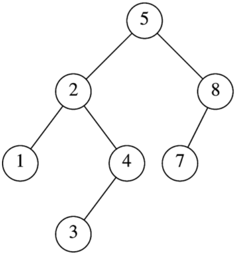
February 8, 2010

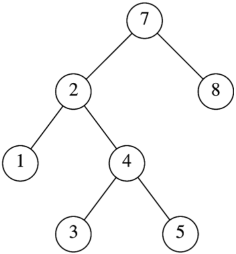
Balanced BSTs

- AVL Trees
  - Height of left and right subtrees at every node differ by at most 1
  - Maintained via rotations
  - Depth always $O(\log_2 N)$
  - Named after Adelson-Velskii and Landis (in 1962)
- Splay Trees
  - After a node is accessed, it moves to the root
  - Average depth per operation is $O(\log_2 N)$

AVL Trees

- Minimum nodes in an AVL tree of height h:
  - S(h) = S(h-1) + S(h-2) + 1
  - Kinda like Fibonacci, but not quite
- AVL trees?



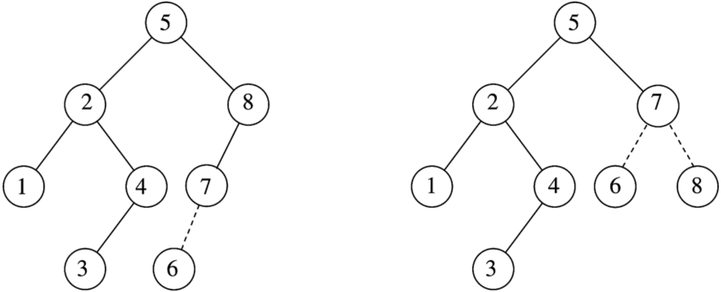


Remove

- Lazy Deletion!
  - Removed nodes are marked as deleted, but NOT removed
  - If same object is re-inserted, these are undeleted
  - Does not affect $O(\log_2 N)$ height as long as deleted nodes are not in the majority
  - If too many, remove all and re-balance

Insert

- Can break balance
- Can fix via rotation. Example inserting 6:

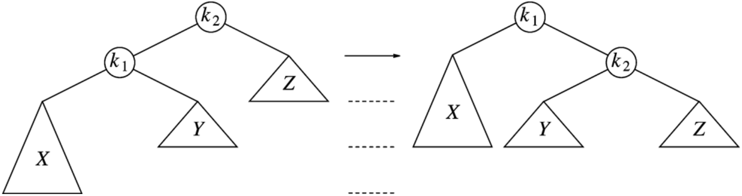


Insert Cont.

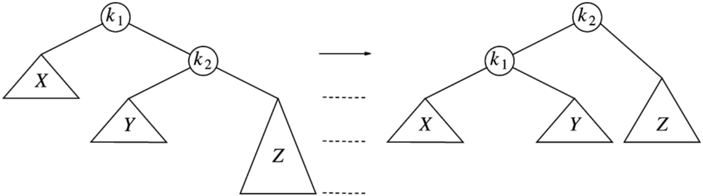
- Only nodes along path to insertion have balance altered.
- Fix violations along path back to root
- Two types of rotation: Single and Double
- Single was on previous slide
- Double involves moving a node up two levels
- Given an unbalanced node, re-balance can be required because of insertion int:
    1. left subtree of the left child
    2. right subtree of left child
    3. left subtree of right child
    4. right subtree of right child
- Cases 1 and 4 require single rotation
- Cases 2 and 3 require double

Case 1: Single rotation right

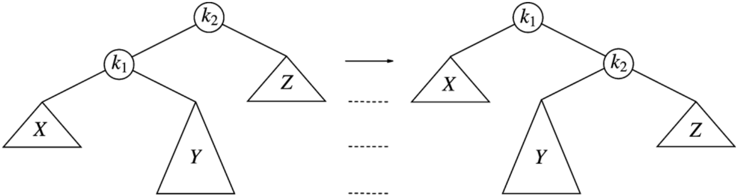
k_2 **is unbalanced**



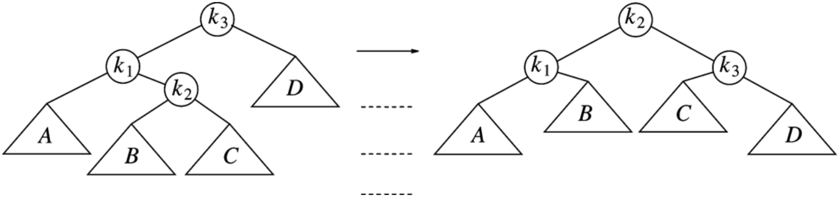
Case 4 example



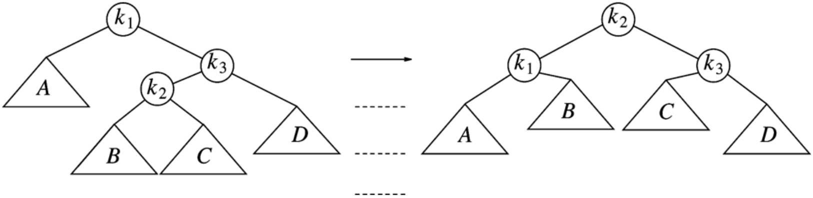
Case 2: Single Rotation Fails



Case 2: Left-Right Double rotation



Case 3: Right-Left Double rotation



Preliminaries

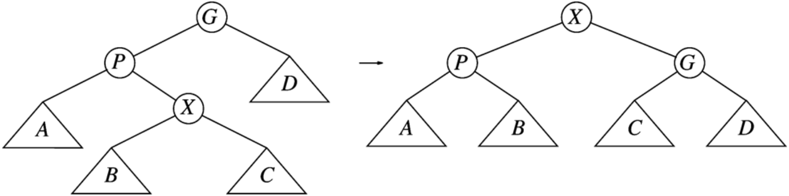
- Accessed nodes are pushed to root via AVL rotations
- Any M consecutive operations take at most $O(M \log_2 N)$ time
- Cost per operation is on average $O(\log_2 N)$
- Some operations take $O(n)$ time
- Does not require maintaining height or balance information!

Solution 1

- Perform single rotations with accessed/new node and parent until accessed/new node is the root
- Problem:
  - Pushes current root node deep into tree
  - In general, can result in $O(M * N)$ time for M operations
  - Example: Insert 1, 2, 3, ..., N
  - Then access 1
  - ...and then n, and then 1...

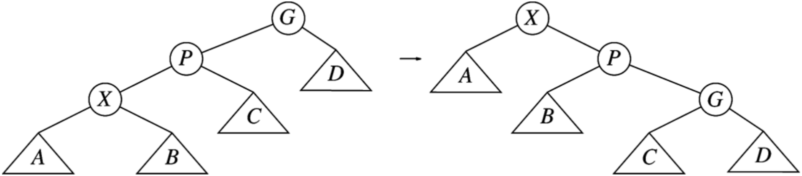
Solution 2

- Still rotate on path from new/accessed node to root
- But, use more selective rotations.
- Still swap with root if root is parent of new/accessed node
- Use double rotation in this situation:



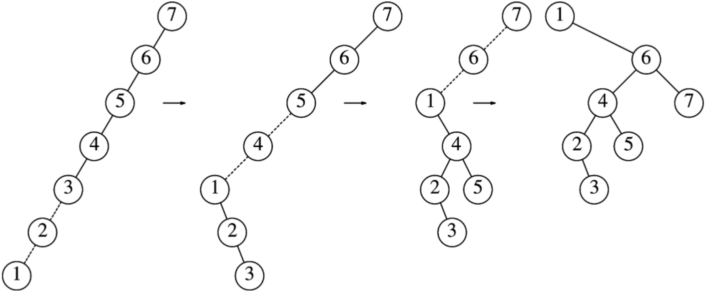
Zig-Zag

- If node X is left child of parent, which is left child of grandparent
- Do double rotation like this:



Previous "bad" example

- The tree from inserting 1...7, when 1 is accessed, given the new rotation methods:



Removal from Splay Trees

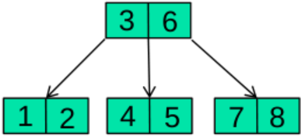
- Access node to be removed (moves it to the root)
- Remove node, leaving subtrees T_L and T_R
- Access largest element in T_L
  - Note that this does not have a right child
- Make T_R the right child of T_L

Why a B Tree?

- Many databases are very large! Some examples:
  - Google
  - Amazon and other online marketers
  - Netflix (user ratings)
  - Filesystems
- Google might have 33 trillion items. Access time for BST:
  - $h = \log_2 33 * 10^{12} = 44.9$
  - Assume 120 disk accesses per second (8.3 millisecond seek time)
  - Each search takes .37 seconds, assuming exclusive use of storage

Reducing Disk Accesses

- Use a 3-way search tree
- Each node stores 2 keys, has at most 3 children
- Each level has $2l^3$ nodes, where l is the height of the level
- Like this:



M-ary trees

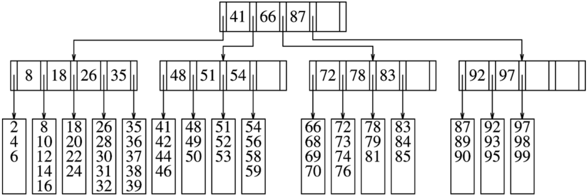
- Each node access gets M-1 keys and M children
- Choose M so that one node is stored in one disk page
  - Yes, this is dependant on how hard drives work.
- Height of tree: $\log_M N$
- Example: Assume 8192 byte page, 32 **bytes** per key, 4 bytes per pointer.
- $32(M-1) + 4M = 8192$
- Solving the above, M $= 228$
- Google example again: $\log_{228} 33 * 10^{12} = 5.7$ disk accesses
- Using values from before, 0.047 seconds per query

B Trees

- M-ary tree where:
  - Data items are stored at the leaves
  - Non-leaf nodes store up to M-1 keys
    - Key i represents the smallest key in subtree i+1
    - Basically, no data is stored in non-leaf nodes
  - Root node is either a leaf, or has between 2 and M children
  - Non-leaf non-root nodes have between $\lceil \frac{M}{2} \rceil$ and M children
  - All leaves are at the same depth and have between $\lceil \frac{L}{2} \rceil$ and L data items
- Requiring at least half full nodes avoids degenerating into binary tree
- Example of choosing L:
  - Assume a data element requires 256 bytes
  - Leaf node capacity of 8192 bytes implies L=32
  - Each node has between 16 and 32 elements

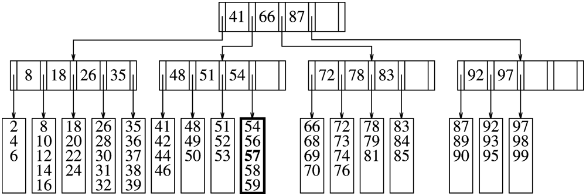
B Tree

- B tree of order 5 (M = 5)
  - Node has 2-4 keys and 3-5 children
  - Leaves have 3-5 data elements



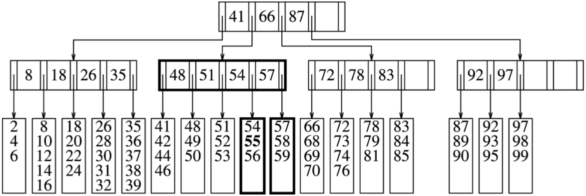
Insertion into Non-Full Leaf

- Insert 57 into previous order 5 tree



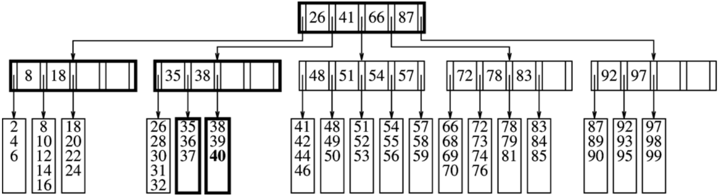
Insertion into full leaf with non-full parent

- Split leaf and promote middle element to parent
- Example: Insert 55 into previous example



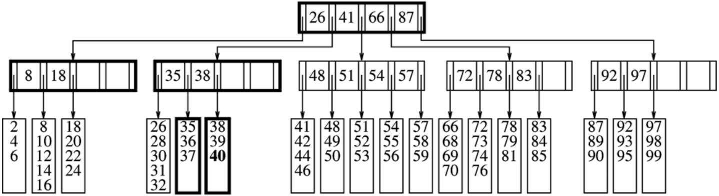
Insertion into full leaf with full parent

- Split parent, promote parent's middle element to grandparent
- Continue until non-full parent or split root
- Example: Insert 40 into previous example. Then 43 and 45?



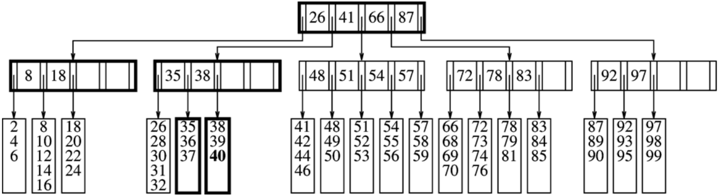
Leaf node not at minimum

- Easy case: Just delete it!
- Example: Remove 16 from previous example



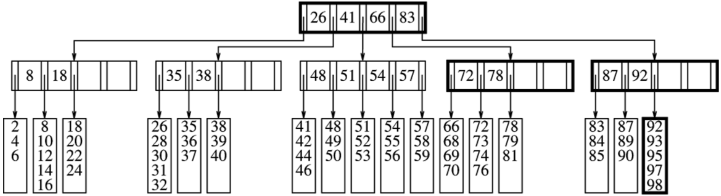
Leaf node at minimum, but not neighbor

- Adopt an element from the neighbor
- Example: Remove 6 from previous example



Further borrowing from the neighbors

- Merge with neighbor, borrow at higher level
- Go as far up the tree as needed
- Example: Remove 99 from previous example



Summary of B Trees

- Optimized for large numbers of items and secondary storage
- Works on:
  - Hard drives
  - Network storage
  - Clusters
  - Any high-latency storage
- M-ary tree with height $\log_M N$
- Used for many real databases, and ReiserFS