CS483-11 Transform-and-Conquer

Instructor: Fei Li

Room 443 ST II

Office hours: Tue. & Thur. 1:30pm - 2:30pm or by appointments

lifei@cs.gmu.edu with subject: CS483

http://www.cs.gmu.edu/∼ lifei/teaching/cs483_fall07/

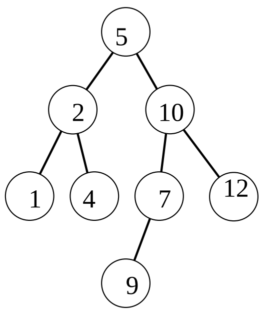
Based on *Introduction to the Design and Analysis of Algorithms* by Anany Levitin and Professor Jyh-Ming Lien's notes.

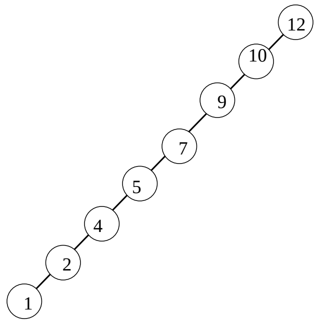
Outline

- Transform and Conquer Techniques (which allow us to handle **dynamic** data / information)
  - Binary search tree
  - AVL tree via **rotations** (or red-black tree or splay tree)
  - $2-3$ tree (or $2-3-4$ tree or B tree)
  - Heap

Binary Search Tree

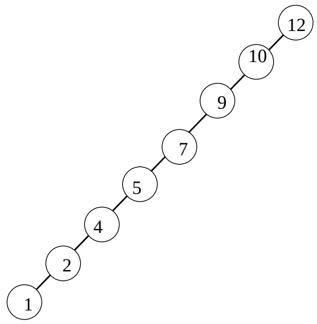
- Binary search tree is a **binary tree** each of whose nodes n has the following properties:
  - All values in the left sub-tree are smaller than the value of n
  - All values in the right sub-tree are larger than the value of n





Binary Search Tree

- What's the advantage of a binary search tree over an array or a list?
  - Efficient related searching and sorting algorithms
  - Inorder traversal produces sorted list
- We can search and **dynamically** insert a value and delete a node from binary search tree.
- Unfortunately, the worst case of these operation can have time complexity: $O(n)$, when the tree becomes a list



Balanced Search Trees

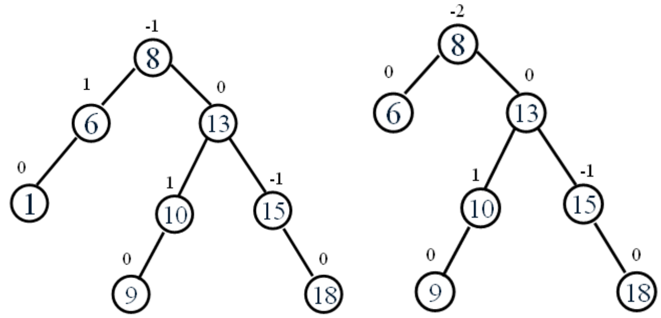
- Attractiveness of binary search tree is blurred by the bad (linear) worst-case efficiency.
- Two ways to solve this:
  - To rebalance binary search tree when a new insertion makes the tree “too unbalanced”
    - AVL trees
    - red-black trees
  - To allow more than one key per node of a search tree
    - $2-3$ trees
    - $2-3-4$ trees
    - B-trees

Outline

- Transform and Conquer Techniques (which allow us to handle **dynamic** data / information)
  - Binary search tree
  - AVL tree via **rotations** (or red-black tree or splay tree)
  - $2-3$ tree (or $2-3-4$ tree or B tree)
  - Heap

AVL Tree

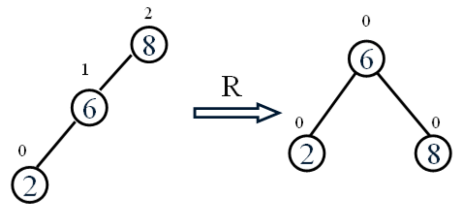
- AVL tree (named after G.M. Adelson-Velsky and E.M. Landis) is always a **balanced** binary search tree.
- **Balance factor** of a node n: the difference between the heights of n's left and right sub-tree.
- The **balance factor** of every node in an AVL tree must be either -1, 0, or $+1$. The height of an empty tree defined as -1.



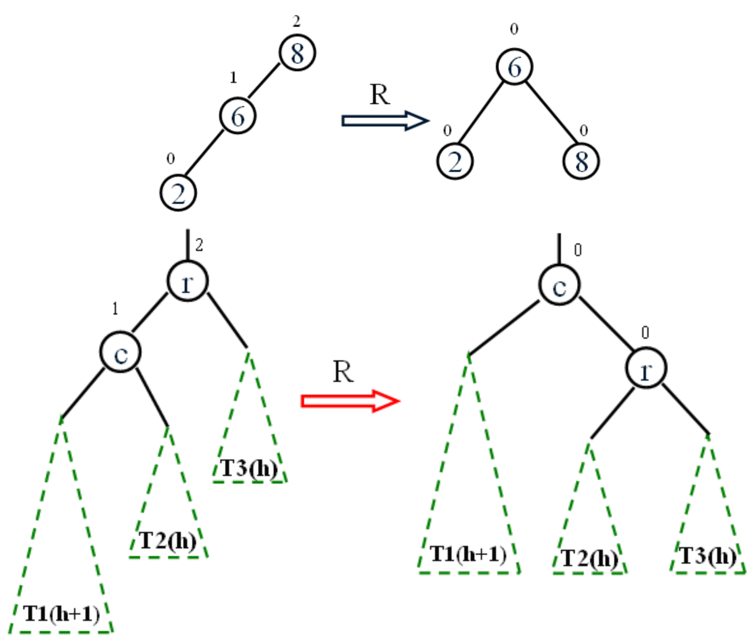
AVL Tree

- AVL tree rotates nodes to maintain the balance after a node is added or removed from the tree.
- Rotation is performed for a subtree rooted at the lowest unbalanced node.
- There are four types of rotations: L-rotation, R-rotation (single rotations), LR-rotation, RL-rotation (double rotations)

Case 1: R rotation

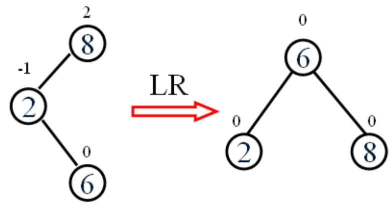


Case 1: R rotation

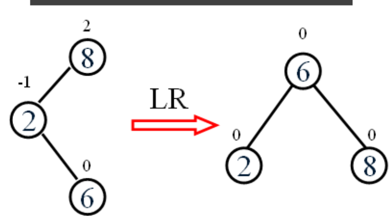


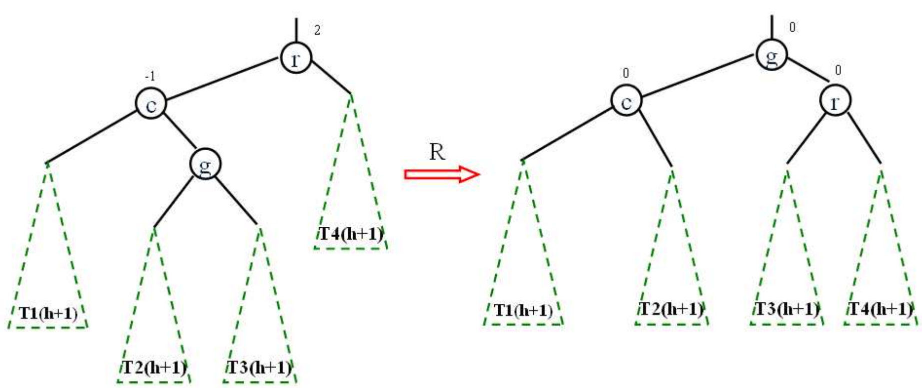
Case 2: L rotation

Case 3: LR rotation



Case 3: LR rotation

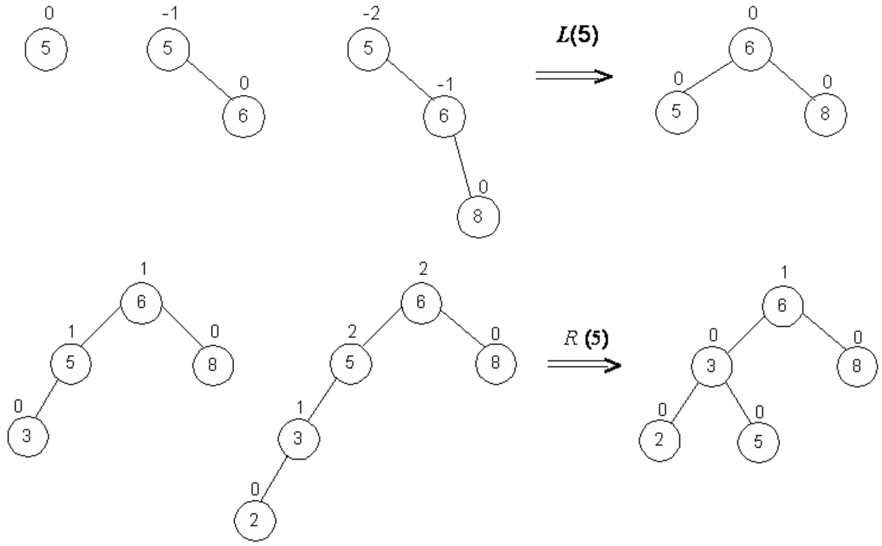




Case 4: RL rotation

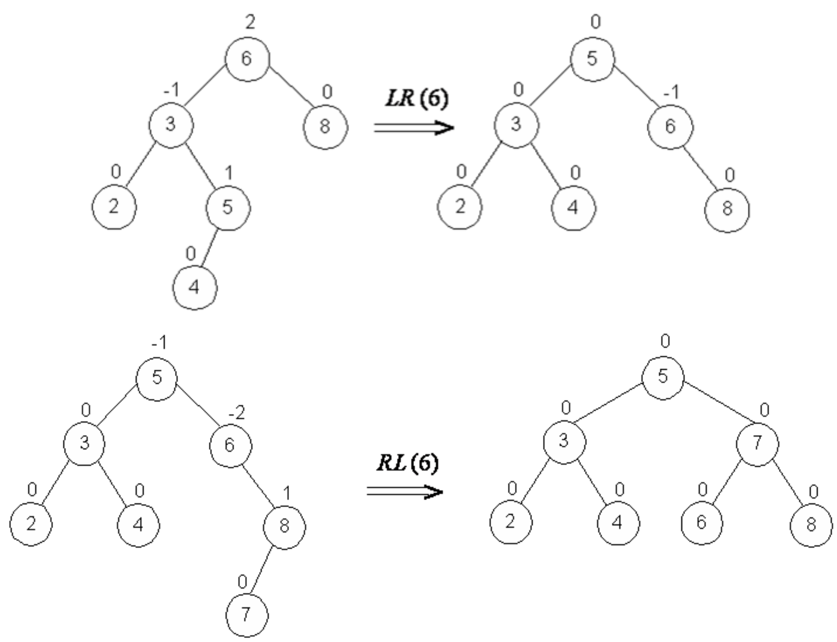
AVL Tree Construction - Example

Build an AVL tree from the list $\{5, 6, 8, 3, 2, 4, 7\}$



AVL Tree Construction - Example

Build an AVL tree from the list $\{5, 6, 8, 3, 2, 4, 7\}$



AVL Tree Analysis

- What is the height h of any AVL tree with n nodes?
  - $h \leq 1.4404 \log_2(n+2) - 1.3277$
  - Average height: $1.01 \log_2 n + 0.1$ for large n (found empirically)

AVL Tree Analysis

- What is the height h of any AVL tree with n nodes?
  - $h \leq 1.4404 \log_2(n+2) - 1.3277$
  - Average height: $1.01 \log_2 n + 0.1$ for large n (found empirically)
- What is the time complexity of the R-, L-, LR-, or RL-rotations?
- What is the time complexity of search/insert/delete of AVL tree?

AVL Tree Analysis

- What is the height h of any AVL tree with n nodes?
  - $h \leq 1.4404 \log_2(n+2) - 1.3277$
  - Average height: $1.01 \log_2 n + 0.1$ for large n (found empirically)
- What is the time complexity of the R-, L-, LR-, or RL-rotations?

  $O(1)$
- What is the time complexity of search/insert/delete of AVL tree?

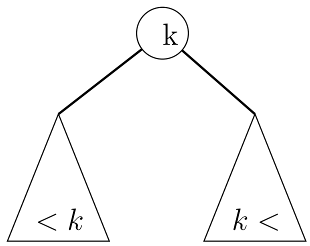
  $O(\log n)$
- Disadvantages:
  - Frequent rotations
  - Complexity

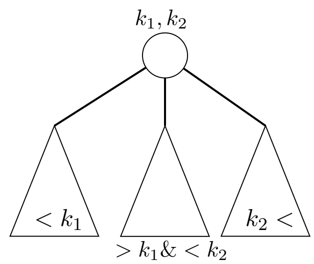
Outline

- Transform and Conquer Techniques (which allow us to handle **dynamic** data / information)
  - Binary search tree
  - AVL tree via **rotations** (or red-black tree or splay tree)
  - $2-3$ tree (or $2-3-4$ tree or B tree)
  - Heap

$2-3$ Tree

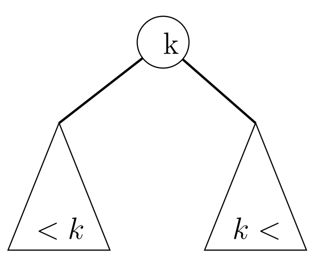
- A $2-3$ tree is a search tree in which
  - Each node can have two or three kids (with one or two keys);
  - Height-balanced (all leaves of the tree are at the same level).





2-3 Tree

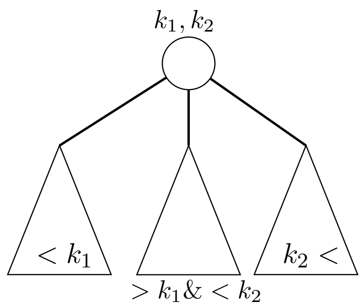
- For a node with two kids, the node has one key. All the keys in the left (resp., right) sub-tree are smaller (resp., larger) than the key of the node.



$2-3$ Tree

- For a node with three kids, the node has two keys. All the keys in the left (resp., right) sub-tree are smaller (resp., larger) than the first (resp., second) keys of the node.

  The keys in the middle tree have values between the first and second key.



Building a $2-3$ Tree

- Iteratively insert the values in to the tree
  - If the tree is empty, create a node with the value
  - Otherwise, search a leaf n that v can be put into and put v to n:
    1. If the size of n is three, make the node into to a 2-node sub-tree T.
    2. Insert the root of T into n's parent node.
    3. Repeat step 1 and 2 for n's parent
- Example: Build a $2-3$ tree from the list $\{9, 5, 8, 3, 2, 4, 7\}$

  Answer is in the book.

Analyzing a $2-3$ Tree

- What is the height of a $2-3$ Tree with n nodes?

  $\log_3(n+1) - 1 \leq h \leq \log_2(n+1) - 1$

- What is the time complexity of each insertion, search, and delete?

  $O(\log n)$