6.006 Introduction to Algorithms
Spring 2008

Lecture 4: Balanced Binary Search Trees

Lecture Overview

- The importance of being balanced
- AVL trees
  - Definition
  - Balance
  - Insert
- Other balanced trees
- Data structures in general

Readings

CLRS Chapter 13. 1 and 13. 2 (but different approach: red-black trees)

Recall: Binary Search Trees (BSTs)

- rooted binary tree
- each node has
  - key
  - left pointer
  - right pointer
  - parent pointer

See Fig. 1

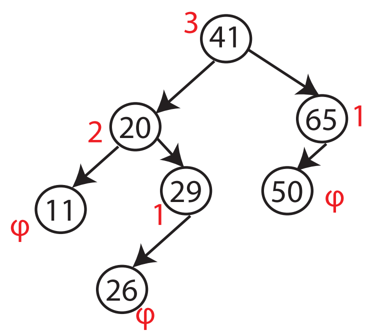

Figure 1: Heights of nodes in a BST

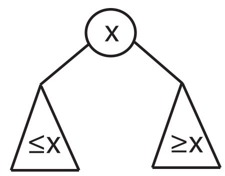

Figure 2: BST property

- BST property (see Fig. 2).
- height of node = length (♯ edges) of longest downward path to a leaf (see CLRS B.5 for details).

The Importance of Being Balanced:

- BSTs support insert, min, delete, rank, etc. in $O(h)$ time, where $h =$ height of tree (= height of root).
- h is between $\lg(n)$ and n: Fig. 3).

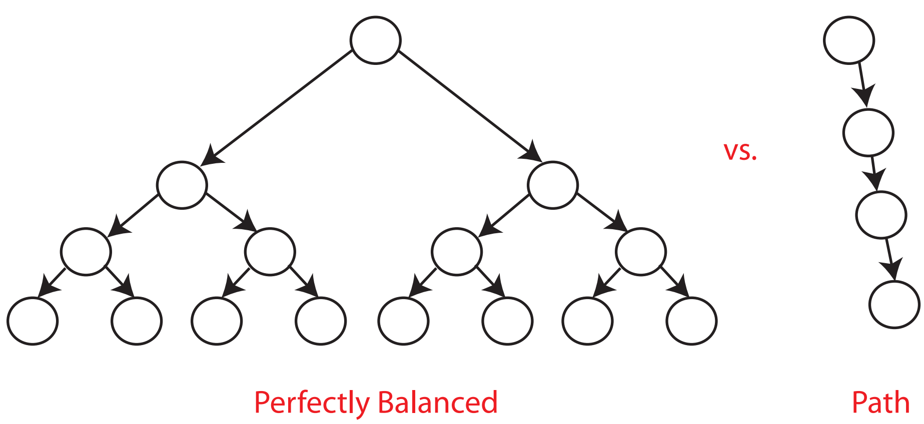

Figure 3: Balancing BSTs

- balanced BST maintains $h = O(\lg n) \Rightarrow$ all operations run in $O(\lg n)$ time.

AVL Trees:

Definition

AVL trees are self-balancing binary search trees. These trees are named after their two inventors G.M. Adel'son-Vel'skii and E.M. Landis [1]
An AVL tree is one that requires heights of left and right children of every node to differ by at most ± 1. This is illustrated in Fig. 4)

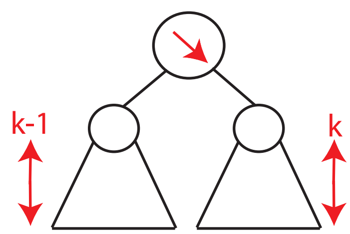

Figure 4: AVL Tree Concept

In order to implement an AVL tree, follow two critical steps:

- Treat nil tree as height -1.
- Each node stores its height. This is inherently a DATA STRUCTURE AUGMENTATION procedure, similar to augmenting subtree size. Alternatively, one can just store difference in heights.

A good animation applet for AVL trees is available at this link. To compare Binary Search Trees and AVL balancing of trees use code provided here.

[1] Original Russian article: Adelson-Velskii, G.; E. M. Landis (1962). "An algorithm for the organization of information". Proceedings of the USSR Academy of Sciences 146: 263266. (English translation by Myron J. Ricci in Soviet Math. Doklady, 3:12591263, 1962.)

Balance:

The balance is the worst when every node differs by 1.
Let $N_h =$ min ($\sharp$ nodes).

$$\begin{aligned} \Rightarrow N_h &= N_{h-1} + N_{h-2} + 1 \\ &> 2N_{h-2} \\ \Rightarrow N_h &> 2^{h/2} \\ \Longrightarrow h &< \frac{1}{2}\lg h \end{aligned}$$

Alternatively:

$$\begin{aligned} N_h &> F_n && (n^{th} \text{ Fibonacci number}) \\ \text{In fact,} N_h &= F_{n+2} - 1 && (\text{simple induction}) \\ F_h &= \frac{\phi^h}{\sqrt{5}} && (\text{rounded to nearest integer}) \\ \text{where}\, \phi &= \frac{1+\sqrt{5}}{2} \approx 1.618 && (\text{golden ratio}) \\ \Longrightarrow \quad \max h &\approx \log_\phi(n) \approx 1.440 \lg(n) \end{aligned}$$

AVL Insert:

1. insert as in simple BST
2. work your way up tree, restoring AVL property (and updating heights as you go).

Each Step:

- suppose x is lowest node violating AVL
- assume x is right-heavy (left case symmetric)
- if x's right child is right-heavy or balanced: follow steps in Fig. 5
- else follow steps in Fig. 6
- then continue up to x's grandparent, greatgrandparent ...

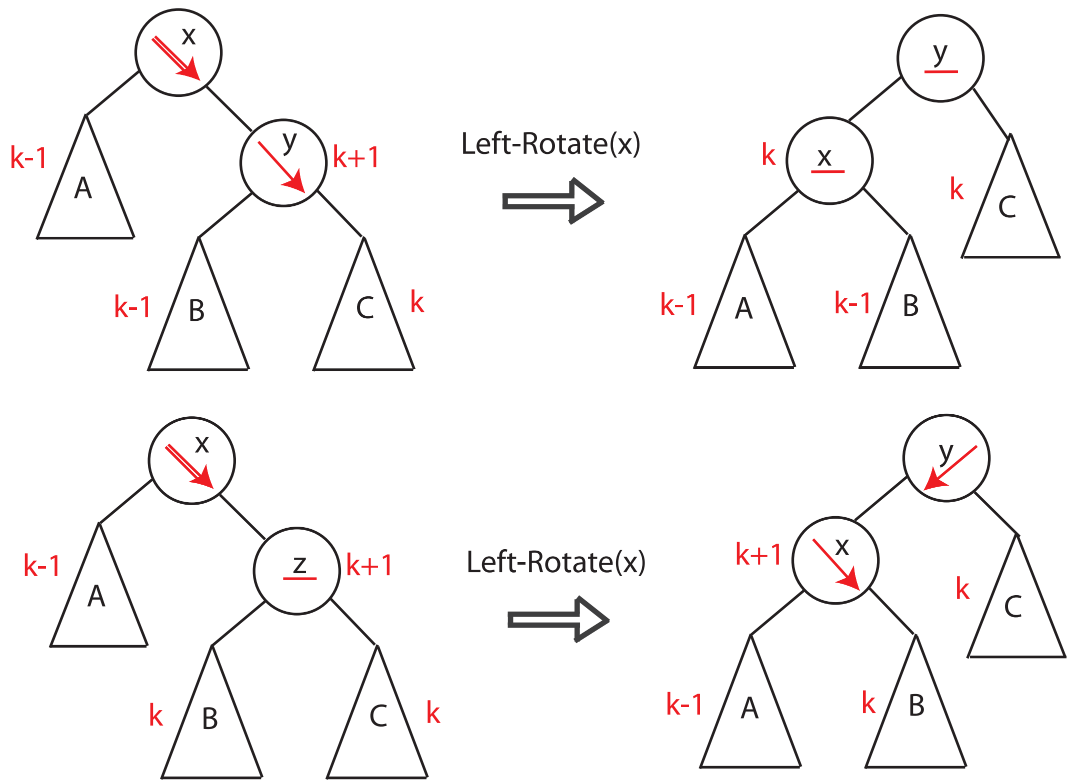

Figure 5: AVL Insert Balancing

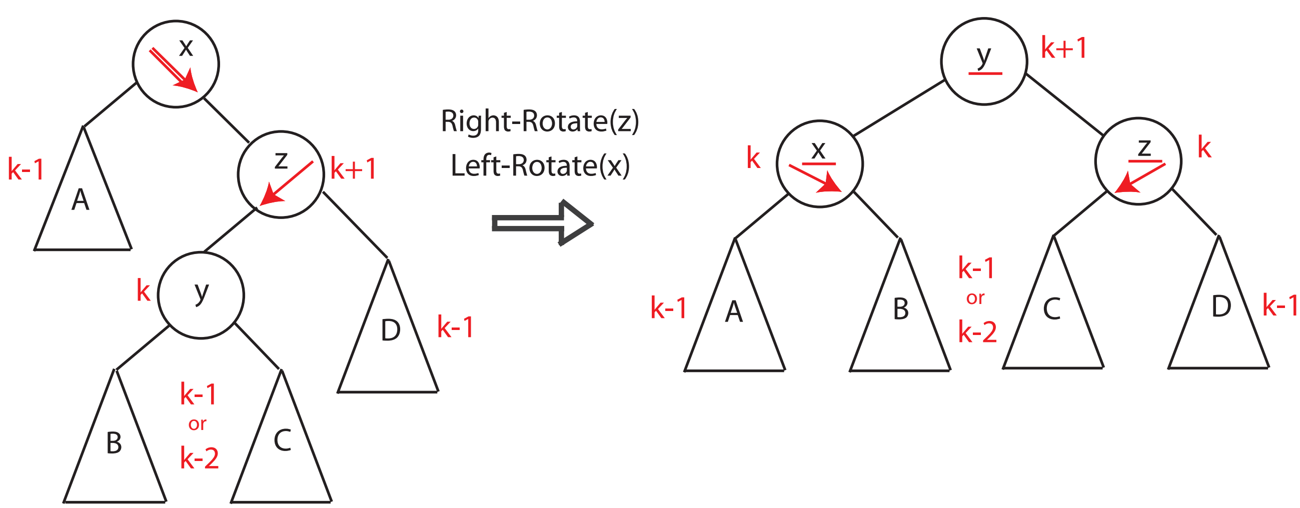

Figure 6: AVL Insert Balancing

Example: An example implementation of the AVL Insert process is illustrated in Fig. 7

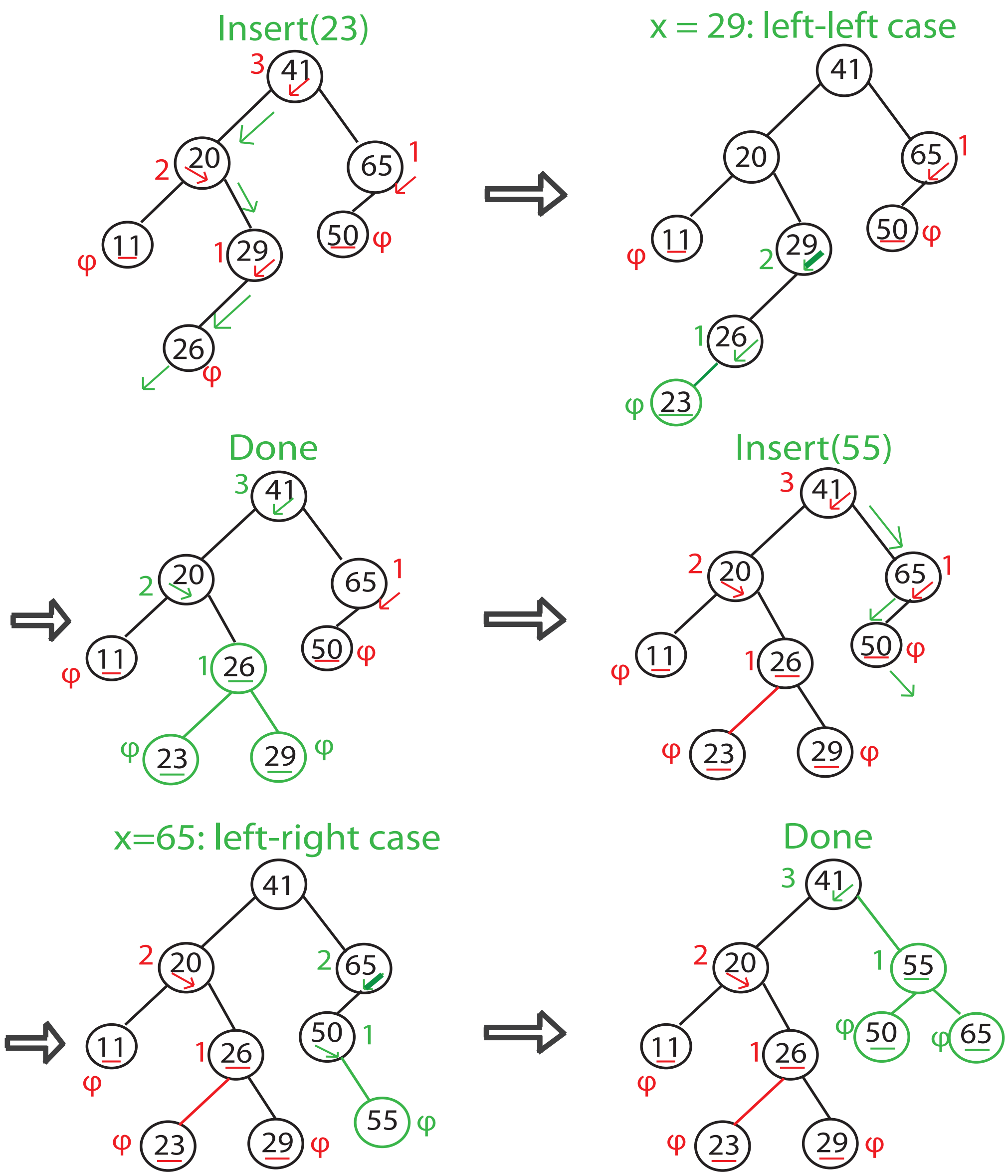

Figure 7: Illustration of AVL Tree Insert Process

Comment 1. In general, process may need several rotations before an Insert is completed.

Comment 2. Delete(-min) harder but possible.

Balanced Search Trees:

There are many balanced search trees.

AVL Trees	Adel'son-Velsii and Landis 1962
B-Trees/2-3-4 Trees	Bayer and McCreight 1972 (see CLRS 18)
BB[α] Trees	Nievergelt and Reingold 1973
Red-black Trees	CLRS Chapter 13
Splay-Trees	Sleator and Tarjan 1985
Skip Lists	Pugh 1989
Scapegoat Trees	Galperin and Rivest 1993
Treaps	Seidel and Aragon 1996

Note 1. Skip Lists and Treaps use random numbers to make decisions fast with high probability.

Note 2. Splay Trees and Scapegoat Trees are "amortized": adding up costs for several operations $\implies$ fast on average.

Splay Trees

Upon access (search or insert), move node to root by sequence of rotations and/or double-rotations (just like AVL trees). Height can be linear but still $O(\lg n)$ per operation "on average" (amortized)

Note: We will see more on amortization in a couple of lectures.

Optimality

- For BSTs, cannot do better than $O(\lg n)$ per search in worst case.
- In some cases, can do better e.g.
  - in-order traversal takes $\Theta(n)$ time for n elements.
  - put more frequent items near root

  A Conjecture: Splay trees are O(best BST) for every access pattern.
- With fancier tricks, can achieve $O(\lg \lg u)$ performance for integers $1 \cdots u$ [Van Ernde Boas; see 6.854 or 6.851 (Advanced Data Structures)]

Big Picture:

Abstract Data Type(ADT): interface spec.
e.g. *Priority Queue*:

- Q = new-empty-queue()
- Q.insert(x)
- x = Q.deletemin()

vs.

Data Structure (DS): algorithm for each op.

There are many possible DSs for one ADT. One example that we will discuss much later in the course is the "heap" priority queue.