

# CSE 332: Data Abstractions AVL Trees

Richard Anderson  
Spring 2016

Адельсон-Вельский Ландис  
дерево

CSE 332 Spring 2016 1

## Announcements

- 4/11: AVL Trees
- 4/13: B-Trees, Project due
- 4/15: B-Trees
- 4/18: Hashing, Taxes due
- 4/20: Hashing
- 4/22: Sorting
- 4/25: Sorting
- 4/27: Sorting
- 4/29: Midterm

CSE 332 Spring 2016 2

## Binary Search Tree Data Structure

- Structural property
  - each node has  $\leq 2$  children
- Order property
  - all keys in left subtree smaller than root's key
  - all keys in right subtree larger than root's key
- Find / Insert
  - Compare with node value to go left or right
  - Runtime  $O(\text{height})$
- Works great, unless tree is unbalanced

CSE 332 Spring 2016 3

## Balanced binary trees

- Binary tree with guarantee on depths of leaves
- $O(\log n)$  insert and delete
- Many flavors
  - Red-black trees
  - Self-adjusting binary trees
  - 2-3 trees
  - AVL trees

CSE 332 Spring 2016 4

## AVL Trees

- Developed in 1962 by Soviet mathematicians Gregory Adelson-Velsky and Eugene Landis
- Structural property on tree guarantees depth  $O(\log n)$
- Rebalance operation to ensure property
- Practical

AN ALGORITHM FOR THE ORGANIZATION OF INFORMATION

BY G. ADELSON-VELSKY AND E. LANDIS

In the present article we discuss the organization of information contained in the text of an information-retrieval system. A theoretical question will be used for this study.

**Statement of the problem.** The information system is written in separate lines on a certain carrier. The information carriers are numbered in a given order which we designate after the others. It consists of the information elements, which in addition to different elements, is contained in the information elements. The information is organized in the order of the number of each in the list in such a way as to ensure a very large number of operations in the required time when the information with the given elements need to be used for the same information elements.

An algorithm is proposed in which both the search and the insertion are carried out in  $O(\log n)$  operations, where  $n$  is the number of information elements which have entered in a given carrier.

A part of the search of the text is not made in the original information. The information elements are arranged there in the order of their insertion. In certain parts of the text a "balanced tree" is formed. In leaves, each node of which corresponds to one of the information elements. The information base is a single tree (Figure 1), each of the nodes of which has no more than two children and no more than one right child substituted in it. These substitutions which are substituted (called rotations) in addition, are made only in the case of an operation. In particular, for the operation "insertion" (adding to the text) and "deletion" (removal from the aggregate of all the nodes, and the aggregate between others). Thus, a given node of text is a balanced tree (Figure 2), with the order of organization of the elements of the corresponding information element in its specific. We will consider the algorithm as operating from left to right.

In the case where the nodes of the information system, when it is inserted into the corresponding information element is inserted. The addition of the nodes of the information base, which are directly substituted on the left and right respectively in the given node, is carried out in the second and third algorithms. It is not, but an already substituted node or other node, then done in order to the corresponding algorithm. The final address is equal to the address that will be.

Let us call the sequence of the nodes of the tree a chain in which each previous node is directly

CSE 332 Spring 2016 5

## AVL Tree overview

- Balance condition
- Depth bound
- Rotations to rebalance the tree

CSE 332 Spring 2016 6

## The AVL Tree Data Structure

### Structural properties

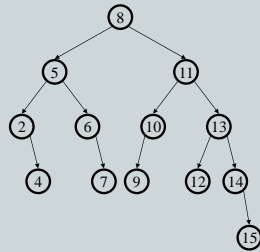
1. Binary tree property
2. Balance: left.height - right.height
3. Balance property: balance of every node is between -1 and 1

### Result:

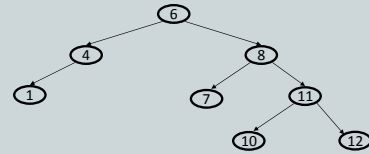
Worst-case depth is  $O(\log n)$

### Ordering property

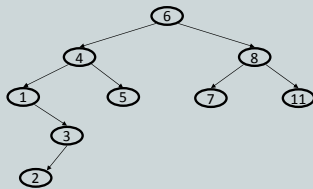
- Same as for BST



## An AVL tree?



## An AVL tree?

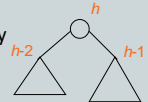


## The shallowness bound

Let  $S(h)$  = the minimum number of nodes in an AVL tree of height  $h$

- $S(h)$  grows exponentially in  $h$ , so a tree with  $n$  nodes has a logarithmic height

- Define  $S(h)$  inductively using AVL property
  - $S(-1)=0, S(0)=1, S(1)=2$
  - For  $h \geq 1, S(h) = 1 + S(h-1) + S(h-2)$



- Show this recurrence grows really fast
  - Similar to Fibonacci numbers
  - Can prove for all  $h, S(h) > \phi^h - 1$  where  $\phi$  is the golden ratio,  $(1+\sqrt{5})/2$ , about 1.62

## The Golden Ratio

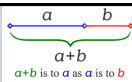
$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1.62$$

This is a special number

- Golden ratio: If  $(a+b)/a = a/b$ , then  $a = \phi b$

- We will need one special arithmetic fact about  $\phi$ :

$$\begin{aligned} \phi^2 &= ((1 + \sqrt{5})/2)^2 \\ &= (1 + 2 * \sqrt{5}/2 + 5) / 4 \\ &= (6 + 2 * \sqrt{5}) / 4 \\ &= (3 + \sqrt{5}) / 2 \\ &= 1 + (1 + \sqrt{5}) / 2 \\ &= 1 + \phi \end{aligned}$$



$$S(-1)=0, S(0)=1, S(1)=2$$

$$\text{For } h \geq 1, S(h) = 1 + S(h-1) + S(h-2)$$

## The proof

Theorem: For all  $h \geq 0, S(h) > \phi^h - 1$

Proof: By induction on  $h$

Base cases:

$$S(0) = 1 > \phi^0 - 1 = 0$$

$$S(1) = 2 > \phi^1 - 1 \approx 0.62$$

Inductive case ( $k > 1$ ):

Show  $S(k+1) > \phi^{k+1} - 1$  assuming  $S(k) > \phi^k - 1$  and  $S(k-1) > \phi^{k-1} - 1$

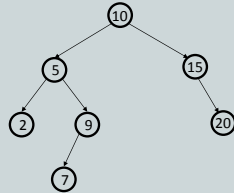
$$\begin{aligned} S(k+1) &= 1 + S(k) + S(k-1) && \text{by definition of } S \\ &> 1 + \phi^k - 1 + \phi^{k-1} - 1 && \text{by induction} \\ &= \phi^k + \phi^{k-1} - 1 \\ &= \phi^{k-1} (\phi + 1) - 1 && \text{by arithmetic (factor } \phi^{k-1} \text{)} \\ &= \phi^{k-1} \phi^2 - 1 && \text{by special property of } \phi \\ &= \phi^{k+1} - 1 \end{aligned}$$

## Good news

Proof means that if we have an AVL tree, then **find** is  $O(\log n)$   
 - Recall logarithms of different bases > 1 differ by only a constant factor

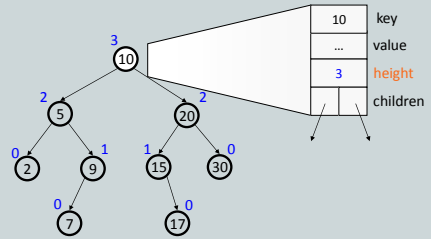
But as we insert and delete elements, we need to:

1. Track balance
2. Detect imbalance
3. Restore balance



Is this AVL tree balanced?  
 How about after insert (30)?

## An AVL Tree



Track height at all times!

## AVL tree operations

- **AVL find:**
  - Same as BST **find**
- **AVL insert:**
  - First **BST insert**, then check balance and potentially "fix" the AVL tree
  - Four different imbalance cases
- **AVL delete:**
  - The "easy way" is lazy deletion
  - Otherwise, do the deletion and then have several imbalance cases (next lecture)

## Insert: detect potential imbalance

1. Insert the new node as in a BST (a new leaf)
2. For each node on the path from the root to the new leaf, the insertion may (or may not) have changed the node's height
3. So after recursive insertion in a subtree, detect height imbalance and perform a *rotation* to restore balance at that node

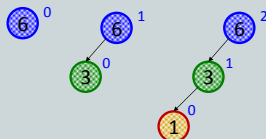
All the action is in defining the correct rotations to restore balance

Fact that an implementation can ignore:

- There must be a deepest element that is imbalanced after the insert (all descendants still balanced)
- After rebalancing this deepest node, every node is balanced
- So at most one node needs to be rebalanced

## Case #1: Example

Insert(6)  
 Insert(3)  
 Insert(1)



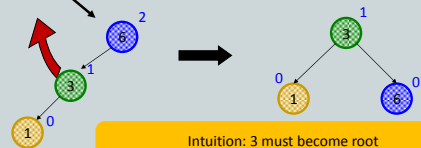
Third insertion violates balance property  
 • happens to be at the root

What is the only way to fix this?

## Fix: Apply "Single Rotation"

- **Single rotation:** The basic operation we'll use to rebalance
  - Move child of unbalanced node into parent position
  - Parent becomes the "other" child (always okay in a BST!)
  - Other subtrees move in only way BST allows (next slide)

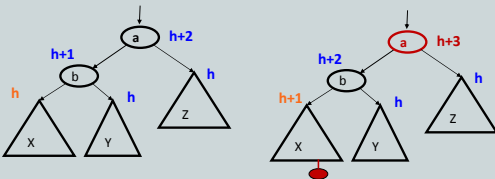
AVL Property violated here



Intuition: 3 must become root  
 new-parent-height = old-parent-height-before-insert

### The example generalized

- Node imbalanced due to insertion *somewhere* in **left-left grandchild** increasing height
  - 1 of 4 possible imbalance causes (other three coming)
- First we did the **insertion**, which would make **a** imbalanced

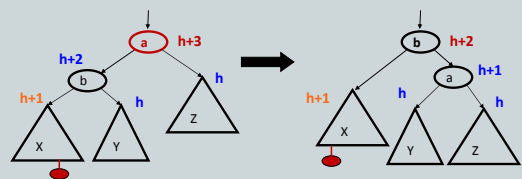


CSE 332 Spring 2016

19

### The general left-left case

- Node imbalanced due to insertion *somewhere* in **left-left grandchild**
  - 1 of 4 possible imbalance causes (other three coming)
- So we rotate at **a**, using BST facts:  $X < b < Y < a < Z$

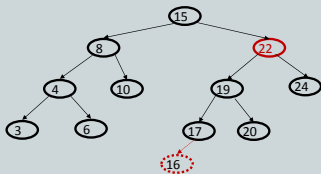


- A single rotation restores balance at the node
  - To same height as before insertion, so ancestors now balanced

CSE 332 Spring 2016

20

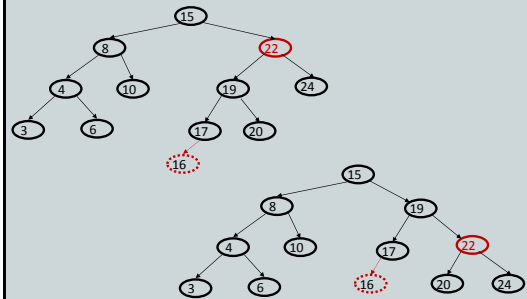
### Another example: insert (16)



CSE 332 Spring 2016

21

### Another example: insert (16)

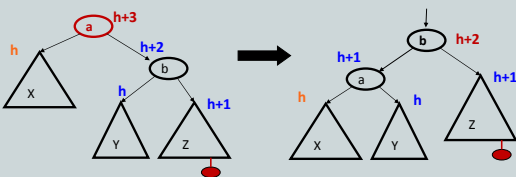


CSE 332 Spring 2016

22

### The general right-right case

- Mirror image to left-left case, so you rotate the other way
  - Exact same concept, but need different code



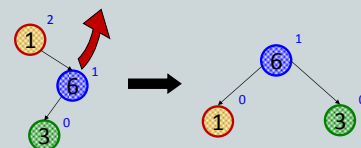
CSE 332 Spring 2016

23

### Two cases to go

Unfortunately, single rotations are not enough for insertions in the left-right subtree or the right-left subtree

Simple example: **insert(1), insert(6), insert(3)**  
 – First wrong idea: single rotation like we did for left-left



CSE 332 Spring 2016

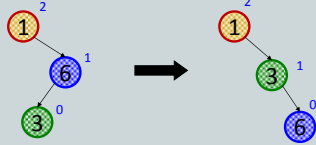
24

## Two cases to go

Unfortunately, single rotations are not enough for insertions in the left-right subtree or the right-left subtree

Simple example: `insert(1), insert(6), insert(3)`

– Second wrong idea: single rotation on the child of the unbalanced node

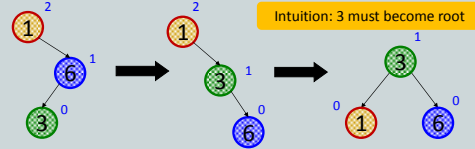


CSE 332 Spring 2016

25

## Sometimes two wrongs make a right ☺

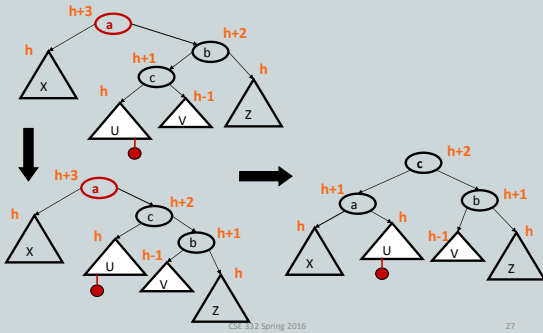
- First idea violated the BST property
- Second idea didn't fix balance
- But if we do both single rotations, starting with the second, it works! (And not just for this example.)
- Double rotation:
  1. Rotate problematic child and grandchild
  2. Then rotate between self and new child



CSE 332 Spring 2016

26

## The general right-left case

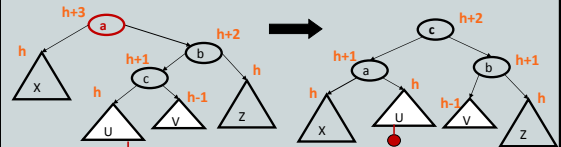


CSE 332 Spring 2016

27

## Comments

- Like in the left-left and right-right cases, the height of the subtree after rebalancing is the same as before the insert
  - No ancestor in the tree will need rebalancing
- Does not have to be implemented as two rotations; can just do:



Easier to remember than you may think:

Move c to grandparent's position

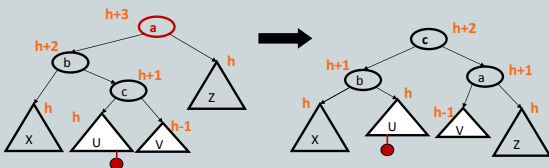
Put a, b, X, U, V, and Z in the only legal positions for a BST

CSE 332 Spring 2016

28

## The last case: left-right

- Mirror image of right-left
  - Again, no new concepts, only new code to write



CSE 332 Spring 2016

29

## Insert, summarized

- Insert as in a BST
- Check back up path for imbalance, which will be 1 of 4 cases:
  - Node's left-left grandchild is too tall
  - Node's left-right grandchild is too tall
  - Node's right-left grandchild is too tall
  - Node's right-right grandchild is too tall
- Only one case occurs because tree was balanced before insert
- After the appropriate single or double rotation, the smallest-unbalanced subtree has the same height as before the insertion
  - So all ancestors are now balanced

CSE 332 Spring 2016

30