

Fundamental Algorithms

Chapter 6: AVL Trees

Michael Bader

Winter 2013/14



Part I

AVL Trees

(Adelson-Velsky and Landis, 1962)

Binary Search Trees – Summary

Complexity of Searching:

- worst-case complexity depends on height of the search trees
- $O(\log n)$ for balanced trees

Inserting and Deleting:

- insertion and deletion might change balance of trees
- question: how expensive is re-balancing?

Test: Inserting/Deleting into a (fully) balanced tree
⇒ strict balancing (uniform depth for all leaves) too strict

Height Balance

Definition (height balance)

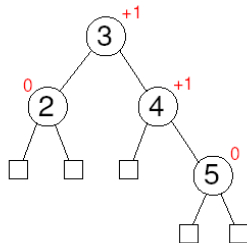
Let $h(x)$ be the height of a binary tree x .

Then, the height balance $b(x.key)$ of a node $x.key$ is defined as

$$b(x.key) := h(x.rightChild) - h(x.leftChild)$$

i.e. the difference of the heights of the two subtrees of $x.key$.

Example:



height balance

AVL Trees

Definition (AVL tree)

A binary search tree x is called an AVL tree, if:

1. $b(x.key) \in \{-1, 0, 1\}$, and
 2. $x.leftChild$ and $x.rightChild$ are both AVL trees.
- = the height balance of every node must be -1, 0, or 1

Number of Nodes in an AVL Tree

Crucial Question for Complexity:

What is the maximal and minimal number of nodes that can be stored in an AVL tree of given height h ?

Maximal number:

- a full binary tree has a height balance of 0 for every node
- thus: full binary trees are AVL trees
- number of nodes: $2^h - 1$ nodes.

Minimal Number of Nodes in an AVL Tree

Minimal number:

A “minimal” AVL tree of height h consists of

- a root node
- one subtree that is a minimal AVL tree of height $h - 1$
- one subtree that is a minimal AVL tree of height $h - 2$

⇒ leads to recurrence:

$$N_{\min\text{AVL}}(h) = 1 + N_{\min\text{AVL}}(h - 1) + N_{\min\text{AVL}}(h - 2)$$

In addition, we know that

- a minimal AVL tree of height 1 has 1 node: $N_{\min\text{AVL}}(1) = 1$
- a minimal AVL tree of height 2 has 2 nodes: $N_{\min\text{AVL}}(2) = 2$

Minimal Number of Nodes in an AVL Tree (2)

Solve recurrence:

$$N_{\min\text{AVL}}(h) = 1 + N_{\min\text{AVL}}(h-1) + N_{\min\text{AVL}}(h-2)$$

$$N_{\min\text{AVL}}(1) = 1$$

$$N_{\min\text{AVL}}(2) = 2$$

Compare with Fibonacci numbers: $f_n = f_{n-1} + f_{n-2}$, $f_0 = f_1 = 1$

| | | | | | | | | |
|-------------------------|---|---|---|---|----|----|----|----|
| h | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| f_h | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 |
| $N_{\min\text{AVL}}(h)$ | 1 | 2 | 4 | 7 | 12 | 20 | 33 | 54 |

Claim: $N_{\min\text{AVL}}(h) = f_{h+1} - 1$ (proof by induction)

Minimal Number of Nodes in an AVL Tree (2)

Minimum number of nodes: $N_{\min\text{AVL}}(h) = f_{h+1} - 1$

- inequality for Fibonacci numbers: $2^{\lfloor n/2 \rfloor} \leq f_n \leq 2^n$

$$\Rightarrow 2^{\lfloor (h+1)/2 \rfloor} - 1 \leq N_{\min\text{AVL}}(h) \leq 2^{h+1} - 1$$

- thus: an AVL tree that contains n nodes will be of height $\Theta(\log n)$

Corollaries:

- Searching in an AVL tree has a time complexity of $\Theta(\log n)$
- Inserting, or deleting a single element in an AVL tree has a time complexity of $\Theta(\log n)$
- BUT: standard inserting/deleting will probably destroy the AVL property.

Part II

Algorithms on AVL Trees

Inserting and Deleting on AVL Trees

General Concept:

- insert/delete via standard algorithms
- after insert/delete: load balance $b(\text{node})$ might be changed to $+2$ or -2 for certain nodes
- re-balance load after each step

Requirements:

- re-balancing must have $O(\log n)$ worst-case complexity

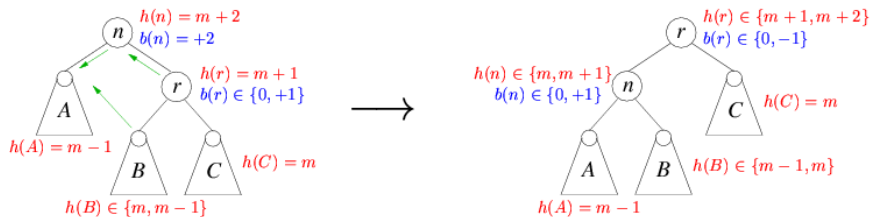
Solution:

- apply certain “rotation” operations

Left Rotation

Situation:

- height balance of the node is +2 (or larger), and
- height balance of the right subtree is 0, or +1



- can reduce height of the subtree and require further rotations

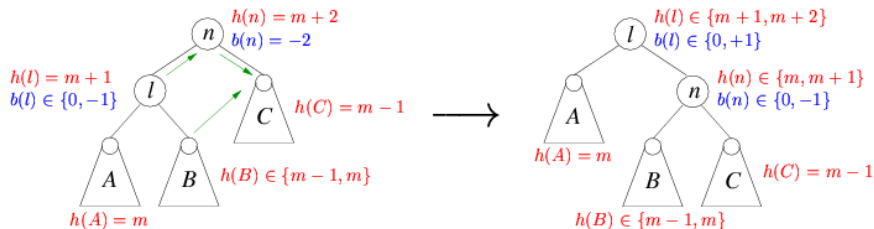
Left Rotation – Implementation

```
LeftRotAVL (val x:BinTree) {  
    x := ( x.rightChild.key,  
          (x.key, x.leftChild , x.rightChild.leftChild),  
          x.rightChild.rightChild  
        );  
}
```

Right Rotation

Situation:

- height balance of the node is -2 (or smaller), and
- height balance of the left subtree is 0 , or -1



- can reduce height of the subtree and require further rotations

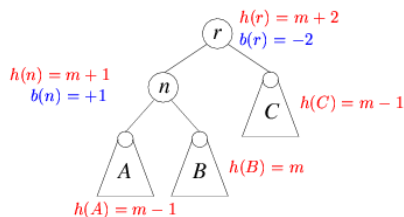
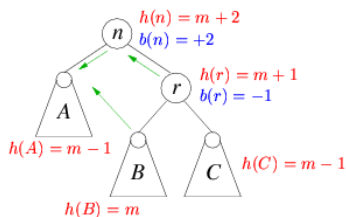
Right Rotation – Implementation

```
RightRotAVL (val x:BinTree) {  
    x := ( x.leftChild.key,  
          x.leftChild.leftChild,  
          (x.key, x.leftChild.rightChild, x.rightChild)  
        );  
}
```

Right-Left Rotation

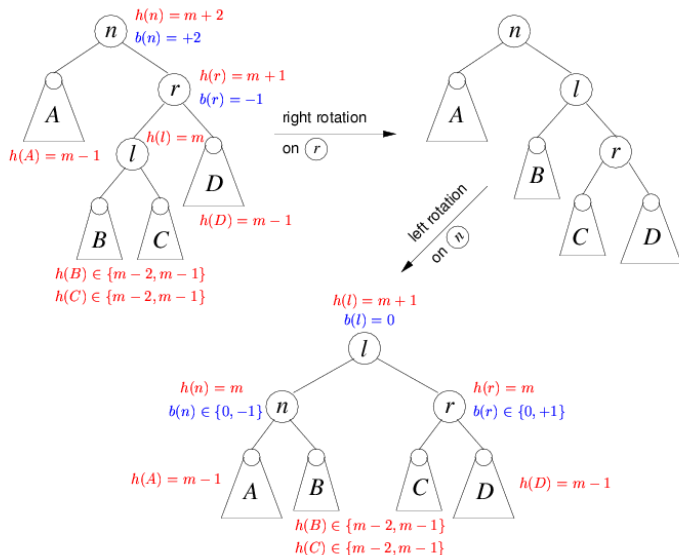
Situation:

- height balance of the node is $+2$, and
- height balance of the right subtree is -1
 \Rightarrow left rotation successful?



- left rotation is not sufficient to restore the AVL property

Solution: Two Successive Rotations



Right-Left Rotation – Implementation

```
RightRotLeftRot (val x:BinTree) {  
    RightRotAVL(x.rightChild);  
    LeftRotAVL(x);  
}
```

In a single procedure:

```
RightLeftRotAVL (val x:BinTree) {  
    x := ( x.rightChild.leftChild.key,  
          ( x.key,  
            x.leftChild,  
            x.rightChild.leftChild.leftChild ),  
          ( x.rightChild.key,  
            x.rightChild.leftChild.rightChild,  
            x.rightChild.rightChild )  
    );  
}
```

Left-Right Rotation

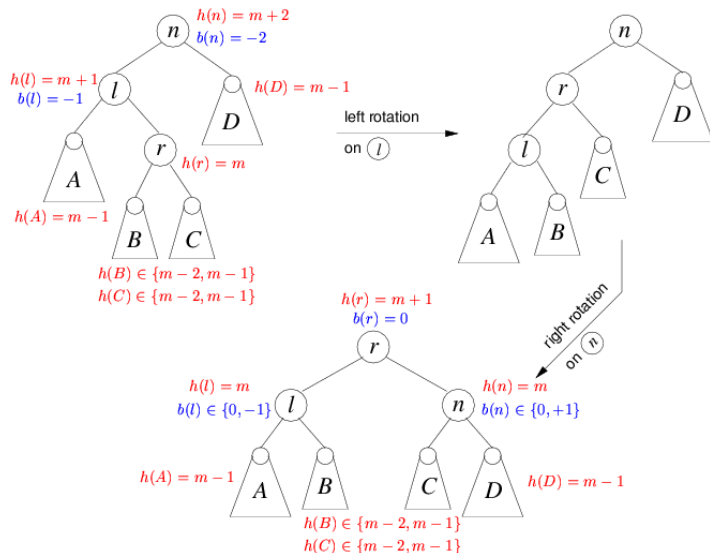
Situation:

- height balance of the node is -2 , and
- height balance of the right subtree is $+1$

Analogous to Right-Left Rotation:

- again, right rotation is not sufficient to restore the AVL property
- right rotation followed by a left rotation

Left-Right Rotation – Scheme



Left-Right Rotation – Implementation

```
LeftRotRightRot(val x: BinTree) {  
    LeftRotAVL(x.leftChild);  
    RightRotAVL(x);  
}
```

In a single procedure:

```
LeftRightRotAVL(val x: BinTree) {  
    x := ( x.leftChild.rightChild.key,  
          ( x.leftChild.key,  
            x.leftChild.leftChild,  
            x.leftChild.rightChild.leftChild ),  
          ( x.key,  
            x.leftChild.rightChild.rightChild,  
            x.rightChild )  
          );  
}
```

Effect of Rotation Operations

Observations:

- LeftRightRot and RightLeftRot always reduce the height of the (sub-)tree by 1.
- LeftRotAVL and RightRotAVL reduce the height by at most 1 (might keep it unchanged)
- thus: AVL property can be violated in a parent node!

After inserting one node into an AVL tree:

- at most one rotation required to restore AVL property

Effect of Rotation Operations (2)

After inserting one node into an AVL tree:

- at most one rotation required to restore AVL property

After deleting one node of an AVL tree:

- up to $\log n$ rotations can be required to restore AVL property

For insertion and deletion:

- imbalance might occur on a much higher level
- thus: AVL property has to be checked in the entire branch of the tree (up to the root)

Corollary:

Time complexity for **deleting**, **inserting**, and **searching** in an AVL tree is $O(\log n)$