5SENG001W - Algorithms, Week 5

Dr. Klaus Draeger

February 15, 2024

RECAP

Last week...

- We talked about the linked data structures
  - Linked lists
    - Insertion
    - Deletion
  - Trees
  - Binary search trees
    - Built-in binary search
    - Insertion, deletion
    - Optimal performance not guaranteed due to **imbalance**.

Overview of today's lecture

- Tree Properties
  - Node Level
  - Height
  - Balance
- Types of Balanced Trees
  - AVL Trees
  - Other types of Balanced Trees, e.g. B-Trees

Recap: Binary Search Trees (BST)

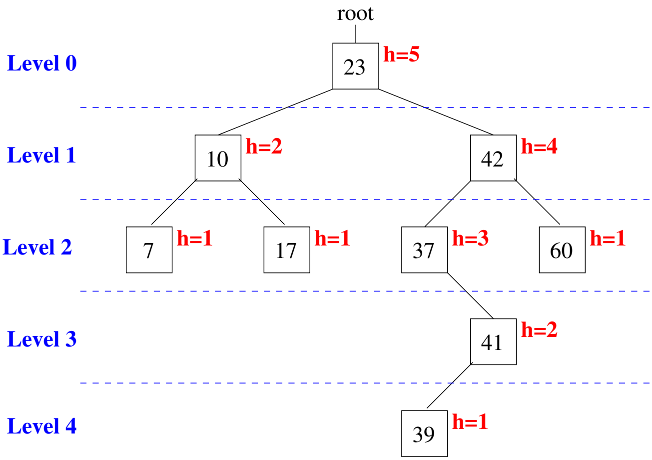
- Contains a set of **values**
  - In applications this wil often be **key-value pairs**
  - We use integers for simplicity
- Consists of **nodes**, each with
  - A data value
  - A left and right child
  - Possibly a parent (depends on application)
- Represents the ordering: for non-null children, `leftChild.data < data` and `rightChild.data > data`
- Operations: search, insert, delete, all with complexity $O(\log N)$.
  – but we are not there yet

Node and Tree Properties

Before we consider what a **balanced** tree is we need to define some properties of nodes and trees:

- The **level** of a node is its **“distance”** from the **root**:
  - This is 0 for the root
  - Otherwise it is one more than the level of the parent
  - The k-th **level of a tree** is the set of all level-k nodes.
- A **branch** of a tree consists of a leaf, its parent etc up to the root.
- The **height** of a tree is the maximum number of nodes on a branch.
- The **height of a node** n is the height of the subtree rooted at n:
  - This is 1 if n is leaf
  - Otherwise it is one more than the maximum height of n’s children

Properties Example



Quick aside: Tree traversals

- One common operation on trees is to **traverse** them
- Visit each node to output/compute/find/... something
- Often the **order of traversal** matters
- Three main ones: pre-order, in-order, post-order
  - **Pre-order**: Process the root first, then traverse the left subtree, then the right
  - **In-order**: Traverse the left sub-tree, then process the root, then traverse the right sub-tree
  - **Post-order**: Traverse the left subtree, then the right, then process the root
- So each traverses both sub-trees (left before right) and the only difference is when the root is processed

Quick aside: Tree traversals

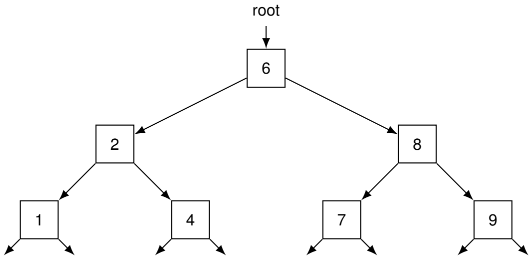
- Pre-order:
  - Useful for "top-down" computations
  - Example: Compute the levels for all nodes
- In-order:
  - Useful for "left-to-right" computations
  - Example: Printing values in a BST in increasing order
- Post-order:
  - Useful for "bottom-up" computations
  - Example: Computing node heights

```java
public class BinarySearchTree{
    /* ... */
    public int getHeight(Treenode n){
        // Base case
        if(n == null)
            return 0;
        // Post-order: process sub-trees first
        int leftHeight = getHeight(n.leftChild),
            rightHeight = getHeight(n.rightChild);
        // ...then process the root
        return 1 + Math.max(leftHeight, rightHeight);
    }
}
```

Balanced trees

Let us try to define what a **balanced** tree is:

- First idea:
  - A node is **perfectly balanced** if its left and right child have the same height
    (null pointers count as 0 height, i.e. a leaf is perfectly balanced).
  - A tree is perfectly balanced if this is true for all nodes.

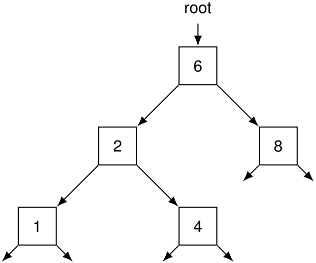


- Problem: This is only possible if all levels are complete
  Number of nodes must be one of 1, 3, 7, 15, 31, . . .
  This is too restrictive!

Balanced trees

Let us **limit** the imbalance instead:

- For each node *n*, its **balance factor** B(n) is the difference `height(n.leftChild) - height(n.rightChild)`
- A node *n* is **balanced** if B(n) is -1, 0, or 1.
- A tree is balanced if this is true for all nodes.



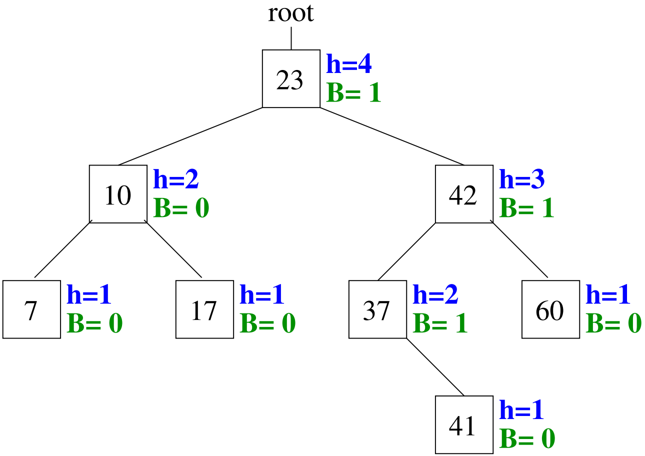
- The balance factor for the root is 1
  For all others it is 0.

Balanced trees

- Our goal was to ensure that search/insertion/deletion in a tree is in $O(\log(n))$.
- It is enough to show that a balanced tree with n nodes has height $O(\log(n))$
  Equivalently: A balanced tree of height k has $\Theta(c^k)$ nodes.
- Suppose $m(k)$ is the minimal number of nodes in a height-k balanced tree.
  - If $k = 0$, the tree is empty, so $m(0) = 0$.
  - If $k = 1$, the tree has a single node, so $m(1) = 1$.
  - Otherwise, one of the root's subtrees must have height $k - 1$ and the other height at least $k - 2$.
    So $m(k) = m(k - 1) + m(k - 2) + 1$ (for the root)
  - The first few values are $0, 1, 2, 4, 7, 13, 20, 33, \ldots$
    This is one less than the **Fibonacci numbers** which do grow exponentially.
  - An alternative argument is that $m(k) > 2 * m(k - 2)$.

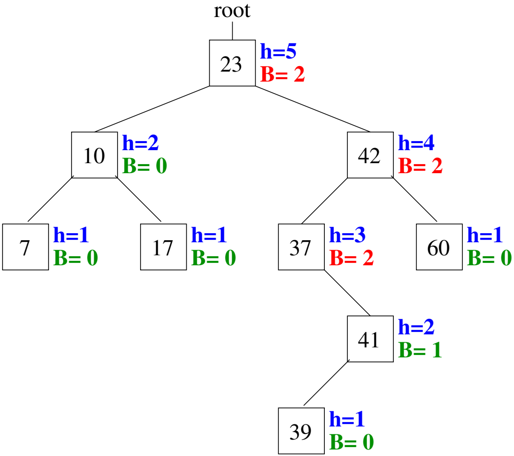
Balanced trees

- Problem: changes in a tree can break the balance



Balanced trees

- Problem: changes in a tree can break the balance

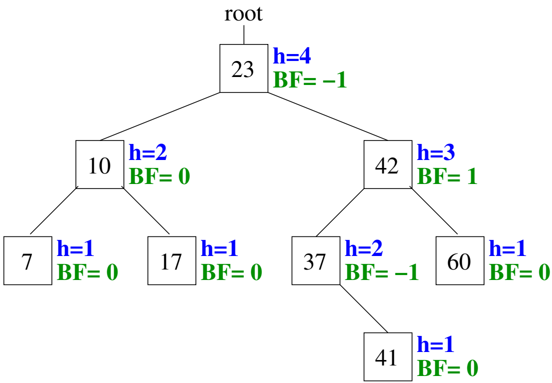


- To fix this we need to **restore** the balance
- BSTs with this addition are called **AVL trees.**

AVL trees

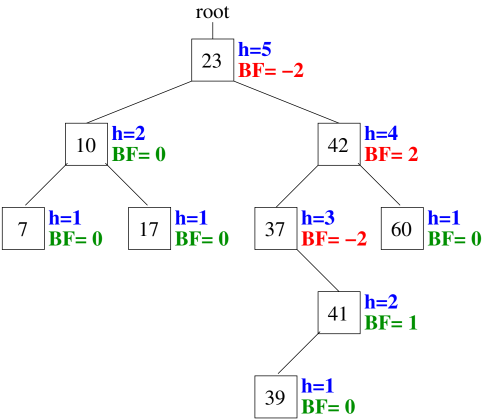
- Named after their inventors *Georgii M. **A**delson-**V**elskii* and *Evgenii M. **L**andis* (1962).
- Require additions to the BST data structure:
  - For each node, keep track of its **height** and **balance factor**
  - After each insertion or delection, **update** this structure information and **re-balance** if needed.
  - Re-balancing involves operations called **rotations**.
  - The examples will only feature insertions but it works the same way after deletions.

AVL Tree: Insertion – Balanced Tree



In this AVL tree 41 has just been inserted.
All the balance factors are still in $\{-1, 0, 1\}$.
The tree has not become unbalanced, so the insertion operation is completed.

AVL Tree: Insertion – Unbalanced Tree

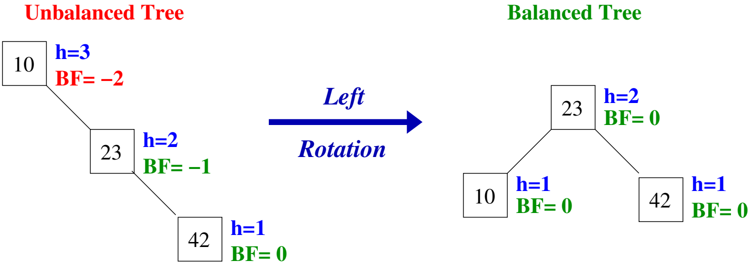


In this AVL tree 39 has just been inserted.
Some BFs are now 2 or -2: the tree has become **unbalanced** and **must be re-balanced.**

AVL Tree Operation: Re-balancing – Rotations

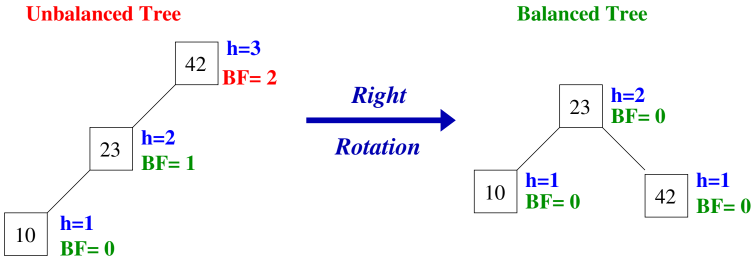
- **Rotations** are operations used to re-balance AVL trees.
- They do this by switching children and parents among two or three adjacent nodes.
- They come in 4 types:
  - Two single roations (Left and Right)
  - Two combination rotations (Left-Right and Right-Left)
- We will look at the single ones first.

AVL Tree Balance Operation: Left Rotation



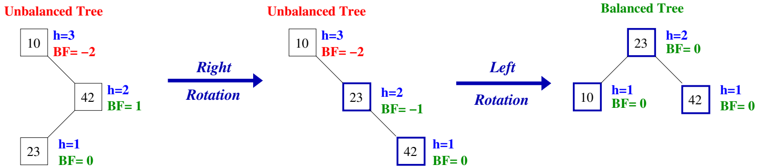
- A **left rotation** re-balances a tree by turning an unbalanced node's right child into its parent.
- This also decreases the height of the sub-tree.
- This type of rotation works if:
  - The node's balance factor is negative
  - Its right child's balance factor is also negative or 0.
    – if it is positive, a Left rotation may make the child unbalanced so use a right-left rotation instead.
- Note that we only saw a small part of the whole tree, it is worth thinking about how surrounding bits are affected.

AVL Tree Balance Operation: Right Rotation



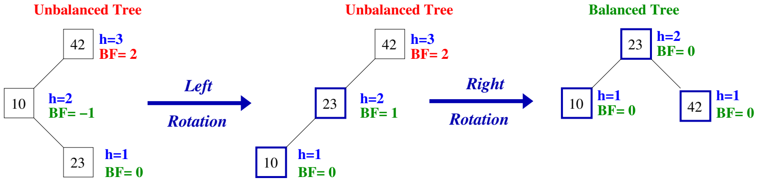
- A **right rotation** is symmetric to the left rotation.
- This time, the unbalanced node's left child becomes its parent.
- This type of rotation works if:
  - The node's balance factor is positive
  - Its right child's balance factor is also positive or 0.

AVL Tree Balance Operation: Right-Left Rotation



- If the node's balance factor is negative but the child's balance factor is positive a regular left rotation may make the child unbalanced
- Instead we
  - first do a left rotation on the child making its balance factor negative
  - then do a right rotation to re-balance.

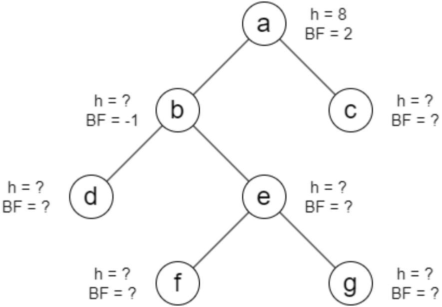
AVL Tree Balance Operation: Left-Right Rotation



- A **left-right rotation** is symmetric to the right-left rotation.
- If the node's balance factor is positive but the child's balance factor is negative we
  - first do a right rotation on the child making its balance factor positive
  - then do a left rotation to re-balance.

Left-Right Rotation: a closer look

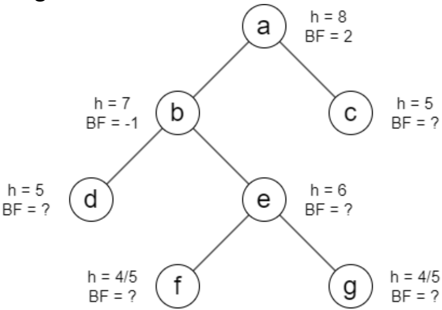
- The examples so far have treated the rotation in isolation.
- Let's look at the last one in a more general case.



- The nodes from the previous example are *a*, *b*, *e*.
- We assume:
  - The balance factors of *a* and *b* are 2 and -1
  - There are no unbalanced nodes deeper in the tree – otherwise we would re-balance these first since it may change their height and affect balance above.

Left-Right Rotation: a closer look

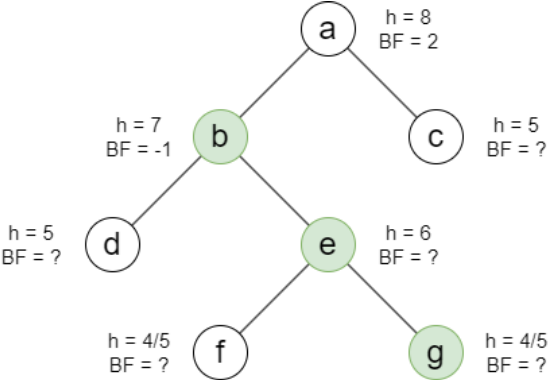
- We can use balance factors to determine some more height values:



  - $h(b)$ must be 2 more than $h(c)$ and one less than $h(a)$
  - $h(e)$ must be 1 more than $h(d)$ and one less than $h(b)$
  - one of $h(f)$ and $h(g)$ must be 1 less than $h(e)$, and the other is 1 or 2 less than $h(e)$.

Left-Right Rotation: a closer look

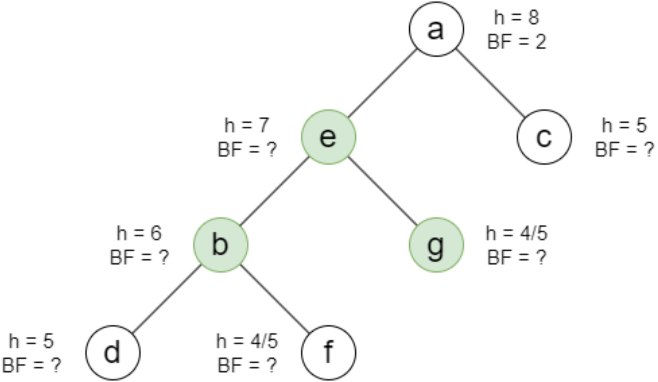
- Let's apply the left rotation (to $b/e/g$) first:



- Recomputing heights, using the fact those of c, d, f, g don't change.
- Notice what happens to f: it goes from being e's left child to b's right child

Left-Right Rotation: a closer look

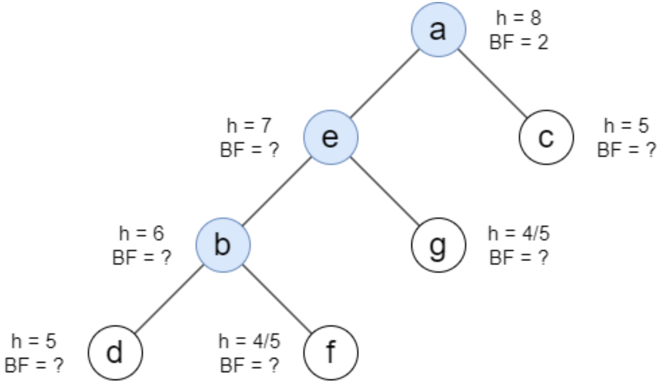
- Let's apply the left rotation (to $b/e/g$) first:



- Recomputing heights, using the fact those of c, d, f, g don't change.
- Notice what happens to f: it goes from being e's left child to b's right child

Left-Right Rotation: a closer look

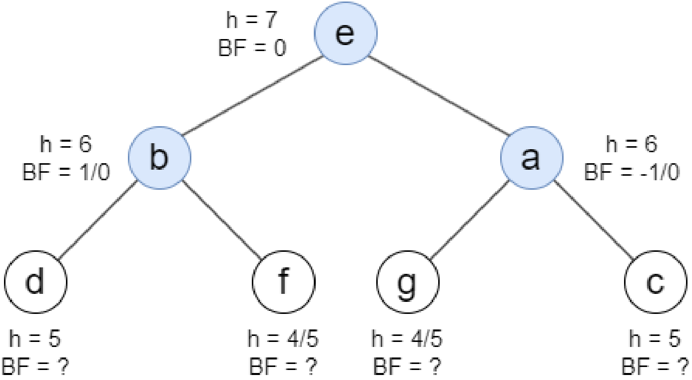
- Next is the right rotation on $a/b/e$:



- Similarly to *f* before, *g* changes parent. These details are important for implementing the rotations.
- The balance factors are now all -1/0/1 as required.

Left-Right Rotation: a closer look

- Next is the right rotation on $a/b/e$:



- Similarly to f before, g changes parent. These details are important for implementing the rotations.
- The balance factors are now all -1/0/1 as required.