AVL[1] Trees

EECS 214

November 9, 2015

[1] Georgy **A**delson-**V**elsky and Evgenii **L**andis

Take-aways

- What is the AVL property?
- How does AVL tree insertion maintain the property?

A basic binary tree

A binary tree, describing structure but not content:

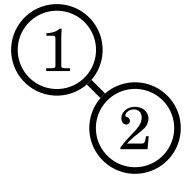
```
; An [BinTree X] is one of:
; -- (leaf)
; -- (branch [BinTree X] X [BinTree X])
(define-struct leaf [])
(define-struct branch [left element right])

; A [BST X] is a [BinTree X]
; that satisfies the BST property
```

Problem!

Binary search is $\mathcal{O}(\log n)$, right?

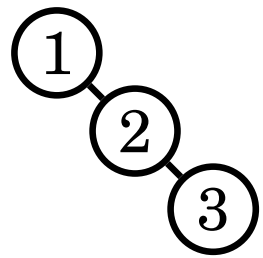
Problem!

Binary search is $\mathcal{O}(\log n)$, right?



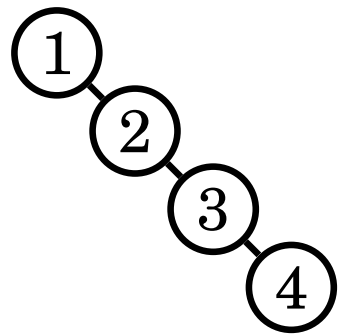
Problem!

Binary search is $\mathcal{O}(\log n)$, right?



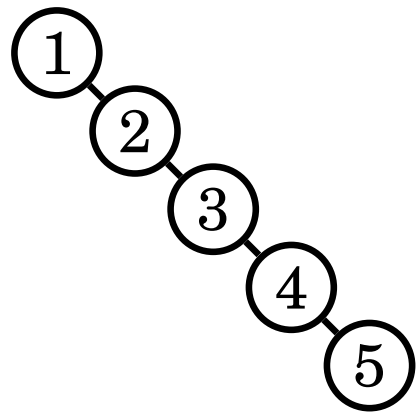
Problem!

Binary search is $\mathcal{O}(\log n)$, right?



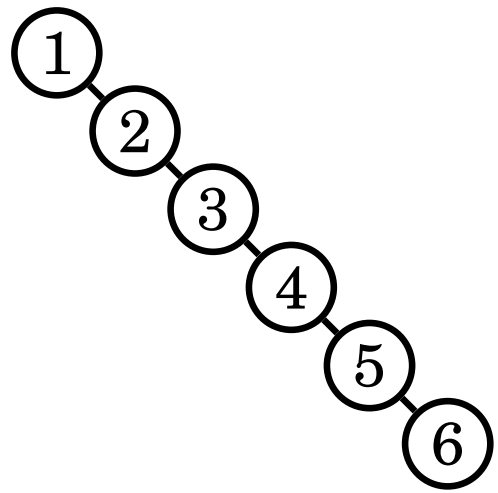
Problem!

Binary search is $\mathcal{O}(\log n)$, right?



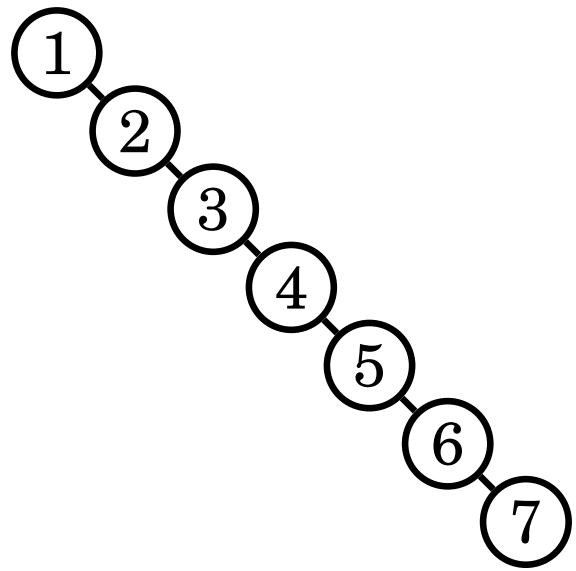
Problem!

Binary search is $\mathcal{O}(\log n)$, right?



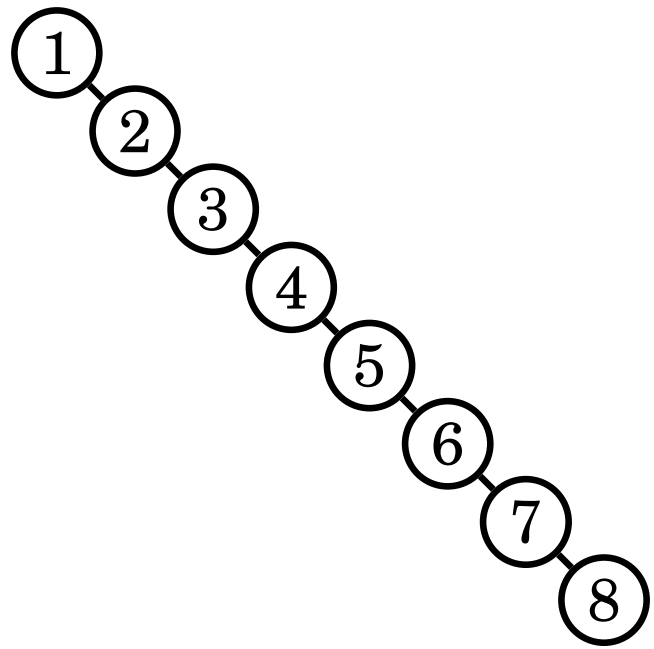
Problem!

Binary search is $\mathcal{O}(\log n)$, right?



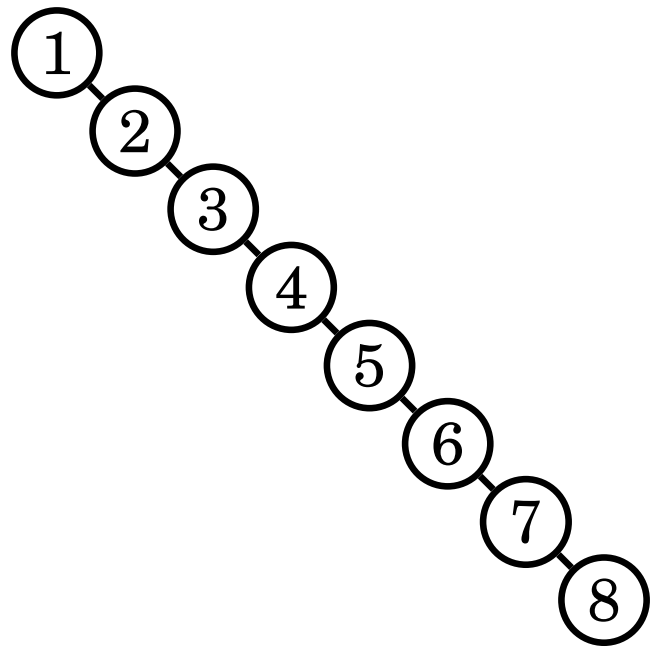
Problem!

Binary search is $\mathcal{O}(\log n)$, right?



Problem!

Binary search is $\mathcal{O}(\log n)$, right?



Only if the tree is (sufficiently) balanced.

Solution

We need some balance.

Solution

We need some balance.

There is a variety of self-balancing trees…

Red-black trees

Main idea: Every node has an extra bit marking it “red” or “black”

Local invariant: No red node has red children

Global invariant: Equal number of black nodes from root to every leaf

2-3 trees

Main idea: 2-nodes have one element and two children; 3-nodes have two elements and three children

Local invariant: All subtrees of a node have the same height

Global invariant: Every leaf is at the same depth

2-4 trees

Main idea: Like 2-3 trees, but also has 4-nodes with three elements and four children.

Local invariant: All subtrees of a node have the same height

Global invariant: Every leaf is at the same depth

B-trees

Main idea: Generalization of 2-4 trees to 2-k trees

Local invariant: Like 2-4 trees, but allow some number of missing subtrees

Global invariant: Every leaf is at the same depth

Splay trees

Main idea: Cache recently access elements near the root of the tree

Local invariant: *Need amortized analysis to talk about this*

Global invariant: Paths are *very likely* to be $\mathcal{O}(\log n)$

AVL trees

Main idea: Maintain a *balance factor* giving the difference between each node's subtrees' heights

Local invariant: Balance factor is at most 1

Global invariant: Tree is approximately height-balanced

Big theme!

We can ensure a global property by maintaining a local property

AVL tree data definition

Each branch includes a balance factor of type `B`:

```
(define-struct leaf [])
(define-struct branch [balance left element right])

; A [PreAVLTree B X] is one of:
; -- (make-leaf)
; -- (make-branch B [PreAVLTree B X]
;                  X [PreAVLTree B X])
; satisfying the BST property

; An [AVLTree X] is [PreAVLTree [-1, 1] X]
; satisfying the AVL property as well
```

Defining the AVL property

The AVL property relies on balance factors, so it requires that balance factors be accurate.

See function `accurate-balances?` in `avl.rkt`.

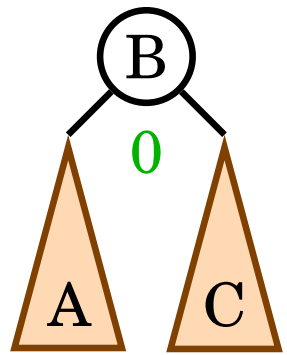
Then we require that every balance factor be `-1`, `0`, or `1`.

See function `avl-balances?` in `avl.rkt`.

```
; avl? : [PreAVLTree Integer X] -> Boolean
; Is the tree actually an AVL tree?
(define (avl? tree)
  (and (bst? tree)
       (accurate-balances? tree)
       (avl-balances? tree)))
```

Maintaining the AVL property

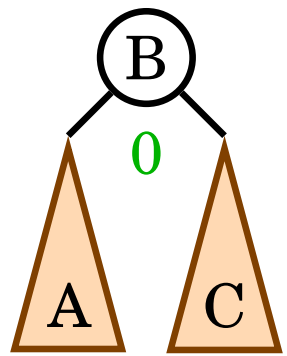
Suppose we have an AVL tree:



(Convention: triangles represent equal-height subtrees.)

Maintaining the AVL property

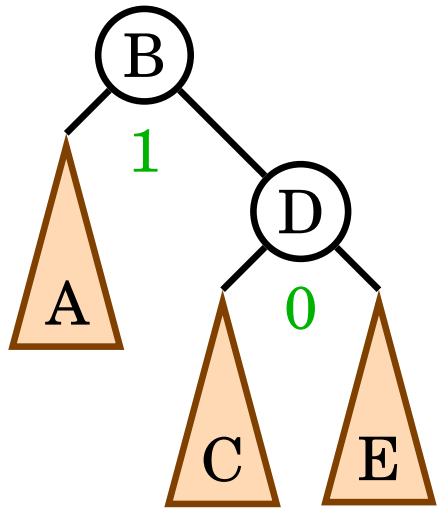
Suppose we have an AVL tree:



(Convention: triangles represent equal-height subtrees.)

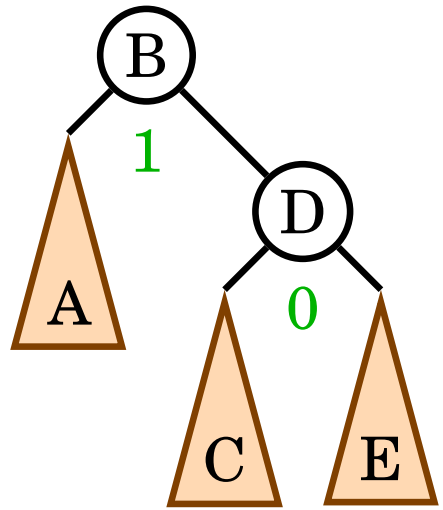
Right now the balance factor is `0`. So if we insert into A or C and that subtree grows in height, it becomes `-1` or `1`.

Maintaining the AVL property



Right now the balance factor at B is 1.
Suppose we insert into A. What happens to B's balance factor?

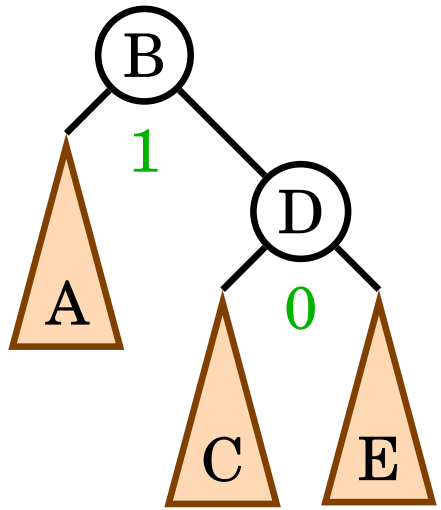
Maintaining the AVL property



Right now the balance factor at B is 1.
Suppose we insert into A. What happens to B's balance factor?

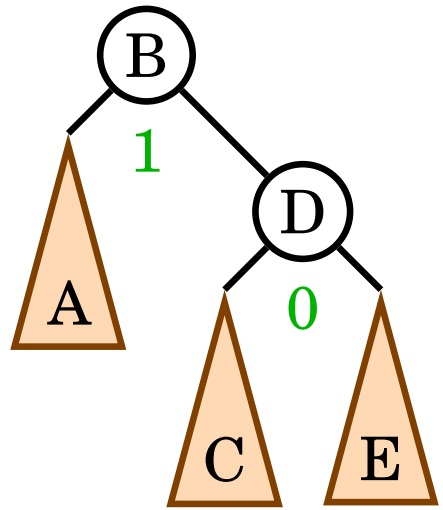
- If no change in A's height, then B's balance doesn't change
- If A's height increases, then B's balance is now `0`

Maintaining the AVL property



Right now the balance factor at B is 1.
Suppose we insert into C. What happens to B's balance factor?

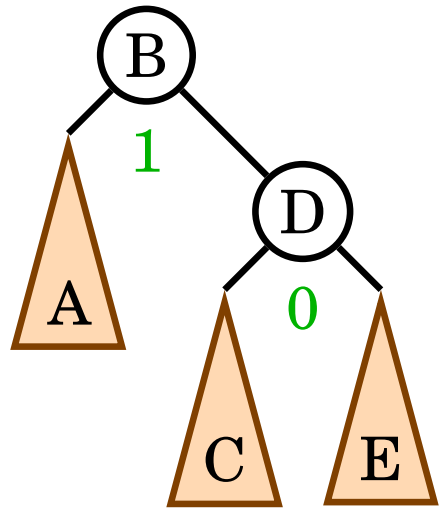
Maintaining the AVL property



Right now the balance factor at B is 1.
Suppose we insert into C. What happens to B's balance factor?

- If no change, then B's balance doesn't change
- If increase, then B's balance becomes 2

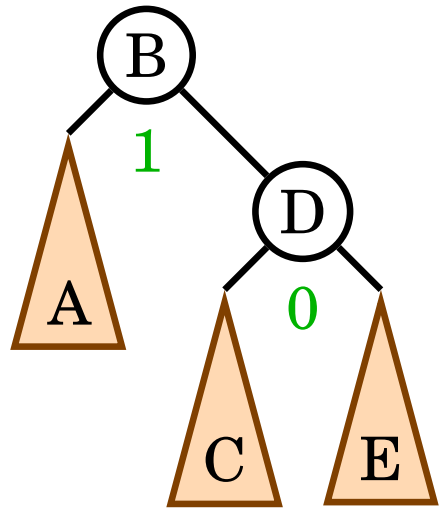
Maintaining the AVL property



Right now the balance factor at B is 1.
Suppose we insert into C. What happens to B's balance factor?

- If no change, then B's balance doesn't change
- If increase, then B's balance becomes 2—not okay!

Maintaining the AVL property

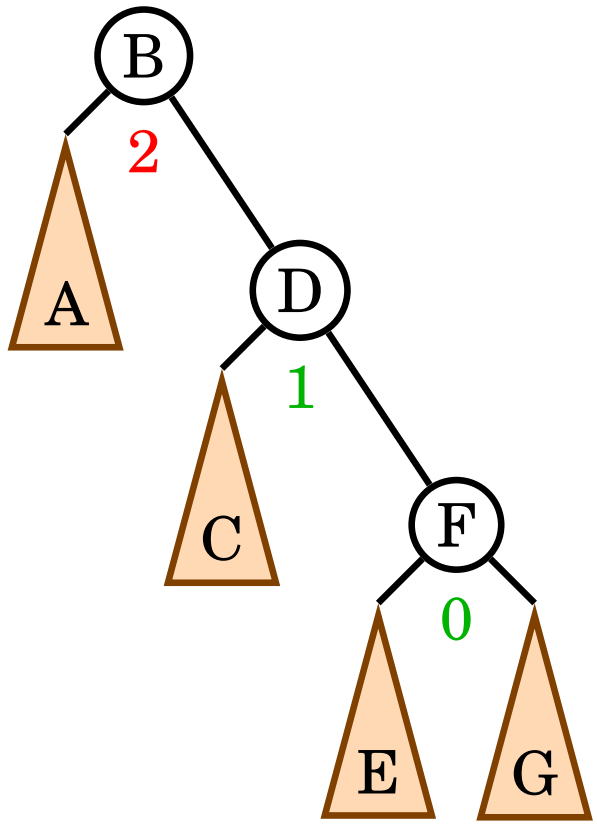


Right now the balance factor at B is 1.
Likewise, suppose we insert into E. What happens to B's balance factor?

- If no change, then B's balance doesn't change
- If increase, then B's balance becomes 2—not okay!

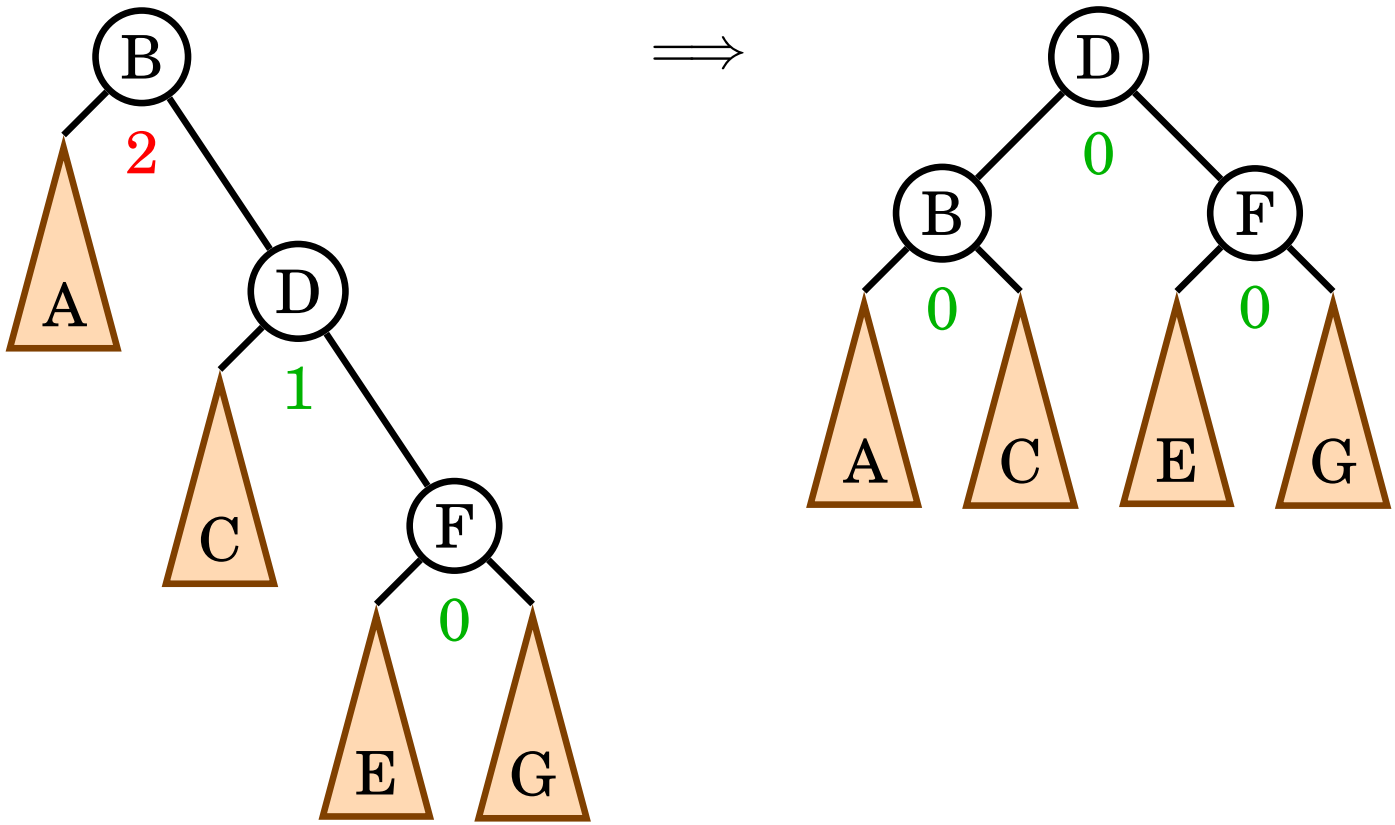
Right-right case

If the height the right-right subtree (formerly E) increases, we get a situation like this:



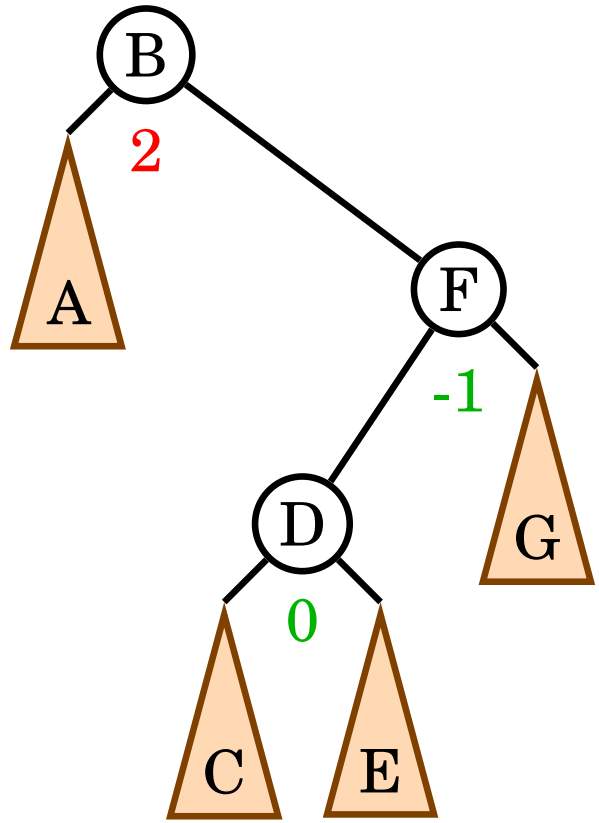
Right-right case

If the height the right-right subtree (formerly E) increases, we get a situation like this:



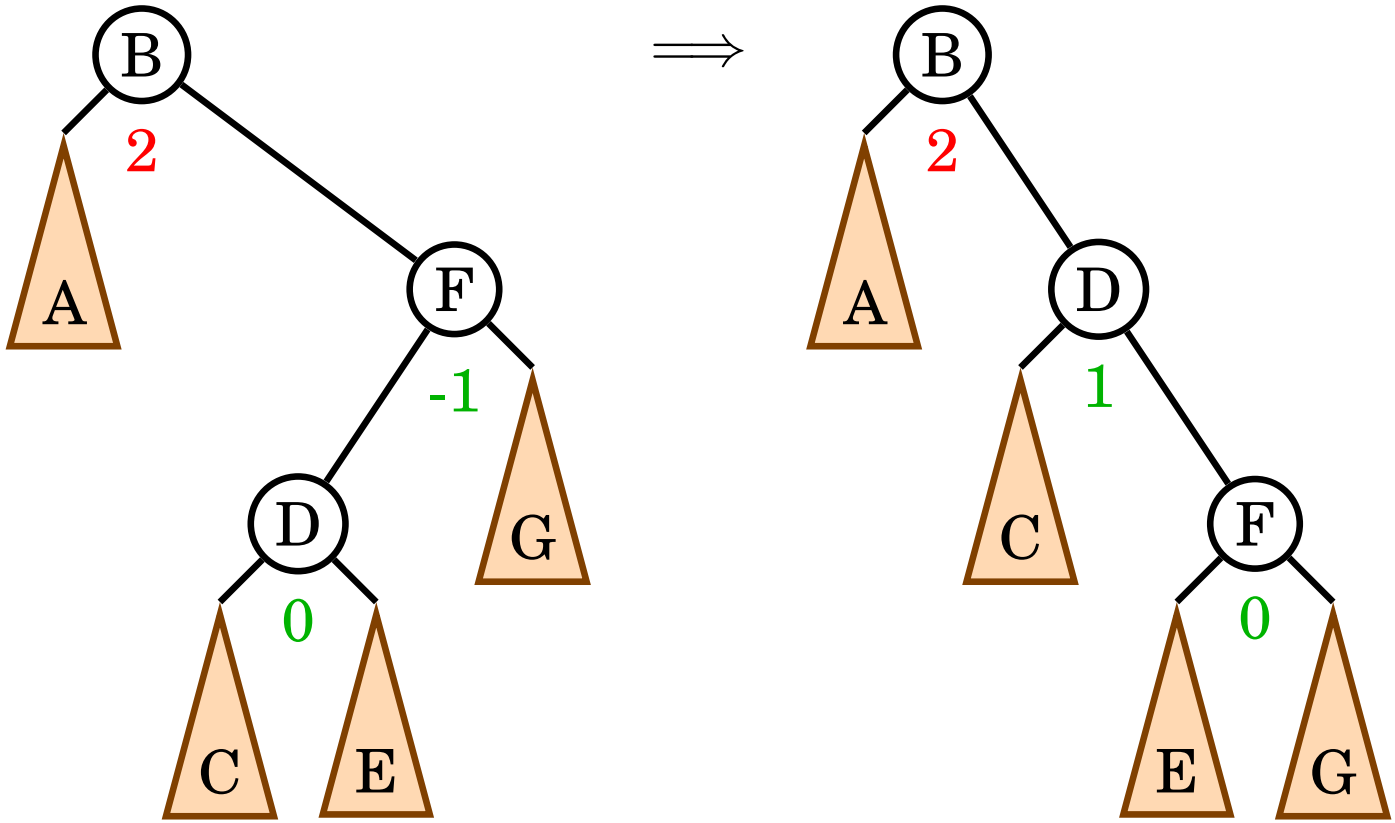
Right-left case

If the height the right-right subtree (formerly C) increases, we get a situation like this:



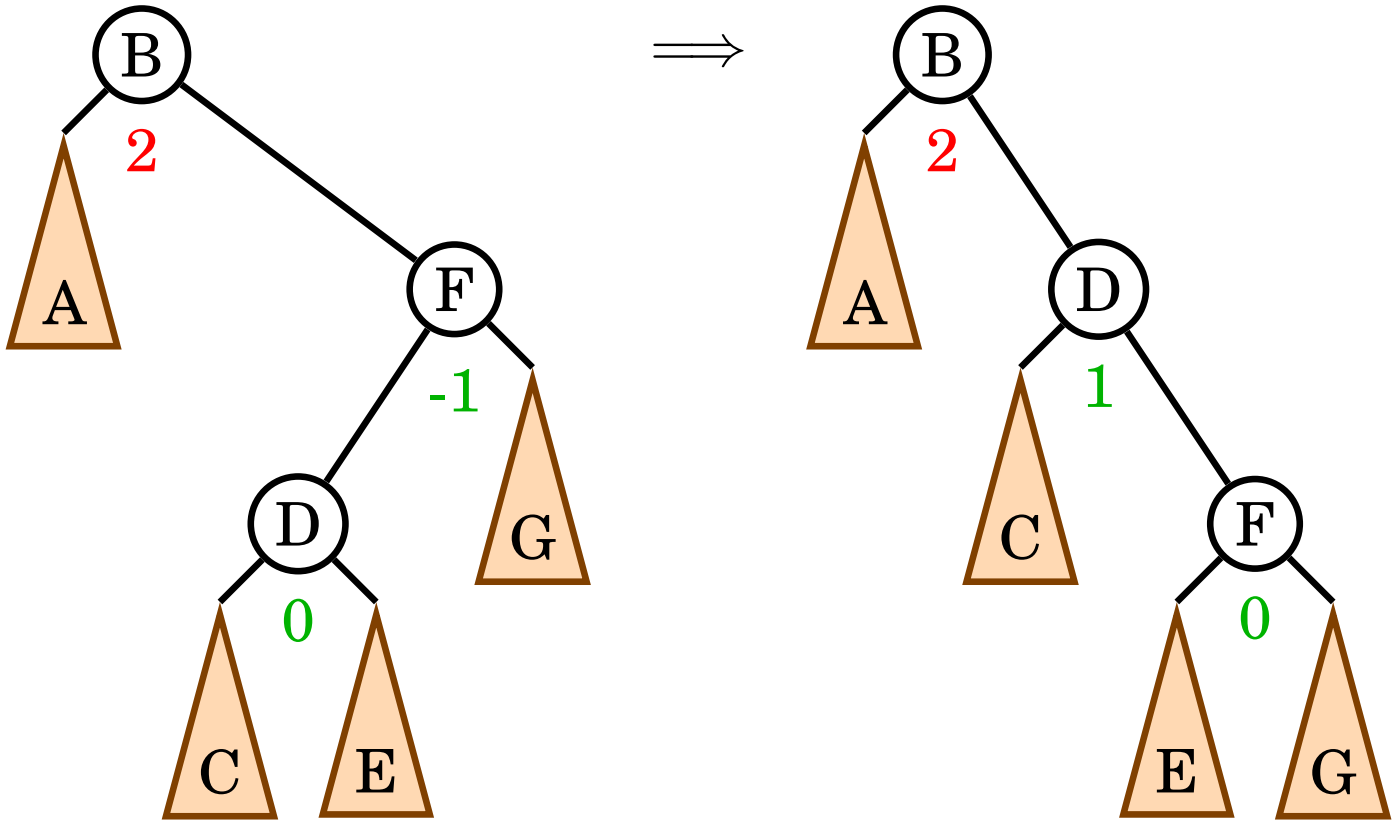
Right-left case

If the height the right-right subtree (formerly C) increases, we get a situation like this:



Right-left case

If the height the right-right subtree (formerly C) increases, we get a situation like this:



This is just the right-right case, which we know how to handle.

Take-aways

- What is the AVL property?
- How does AVL tree insertion maintain the property?