Chapter 5, Binary search trees:

- Binary search trees; the balanced BST problem (spring-break eve; finishing Monday)
- AVL trees (Monday and **Today**)
- Traditional red-black trees (Friday)
- Left-leaning red-black trees (next week Monday)
- "Wrap-up" BST (next week Wednesday)

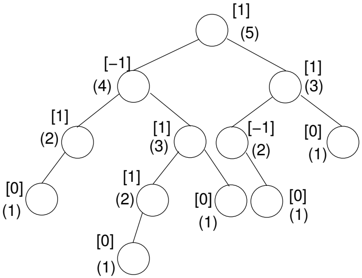
Today:

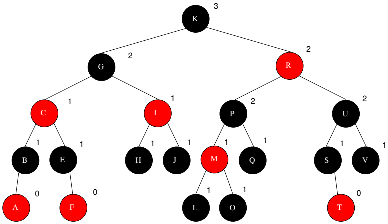
- Practice problem
- Review of balanced BST problem, AVL concepts
- Review of AVL cases
- Proof of AVL correctness
- AVL performance
- Solution to practice problem

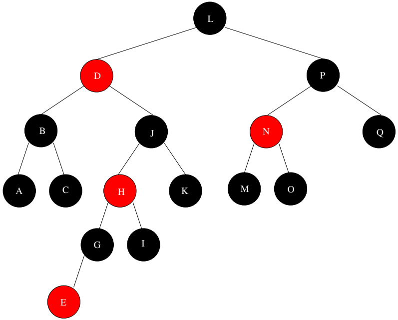
Ex 5.1. Write the method `bst2Array()`, which takes a simplified BST, represented by the root node, and returns a sorted array containing the keys. The size of the array should be the number of keys, and when given a null node, the method should return an array of size 0. Test using `BST2ATest`. Hint: You may want to write one or more recursive helper methods.

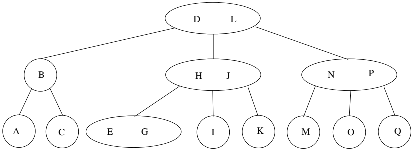
```
public class BSTNode {
    public final int key;
    public final BSTNode left, right;

    public BSTNode(int key, BSTNode left, BSTNode right) {
        this.key = key;
        this.left = left;
        this.right = right;
    }
}
```

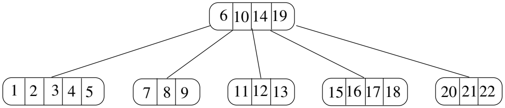
- The BST data structure supports the map ADT with $\Theta(\lg n)$ operations, as long as the tree is balanced.
- Perfect balance isn't necessary. The trees need only be "pretty balanced."
- Schemes for keeping trees have a tradeoff between time spent rebalancing vs the benefit of having the tree more balanced. Each scheme needs to ask
  - How do we define and measure "balance"?
  - What information needs to be stored for that measure?
  - How imbalanced is too imbalanced?
  - What sequence of rotations are needed to fix up the tree when it becomes too imbalanced?


[1] (5)
[−1] (4)
[1] (3)
[1] (2)
[1] (3)
[−1] (2)
[0] (1)
[0] (1)
[1] (2)
[0] (1)
[0] (1)
[0] (1)


K 3
G 2
R 2
C 1
I 1
P 2
U 2
B 1
E 1
H 1
J 1
M 1
Q 1
S 1
V 1
A 0
F 0
L 1
O 1
T 0


L
D
P
B
J
N
Q
A
C
H
K
M
O
G
I
E


D L
B
H J
N P
A
C
E G
I
K
M
O
Q


6 10 14 19
1 2 3 4 5
7 8 9
11 12 13
15 16 17 18
20 21 22

The *height* of a node (or (sub)tree) is the number of nodes on any path from that node to any leaf, inclusive.

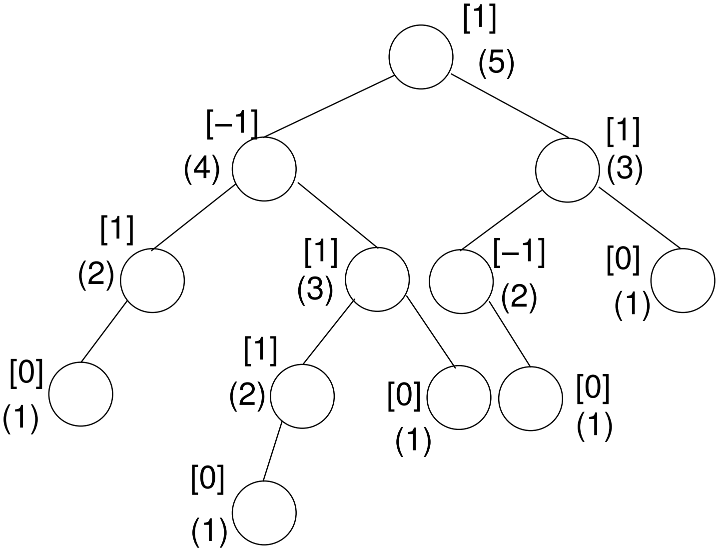
$$height(c) = \begin{cases} 0 & \text{if } c \text{ is null} \\ \max(height(c.\ell) + height(c.r)) + 1 & \text{otherwise} \end{cases}$$

The *balance* of a node is the difference between the heights of its left and right children. In an AVL tree, each node's subtrees' heights must differ by at most 1:

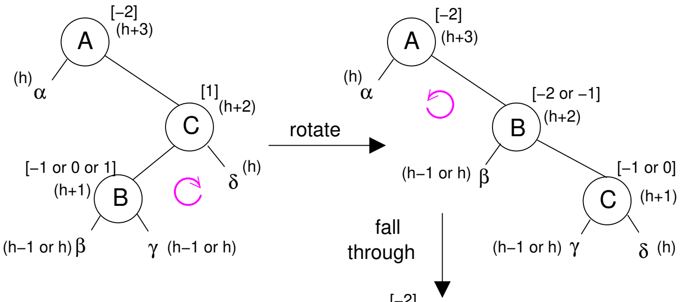
$$\forall\ x \in \text{nodes}, |height(x.\text{left}) - height(x.\text{right})| \leq 1$$

A node that has balance 1 or -1 has a *bias*. A node that (temporarily) has balance 2 or -2 is in *violation*.

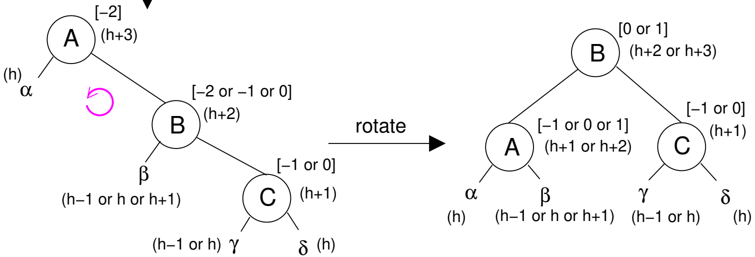
(A balance less than -2 or greater than 2 shouldn't happen even temporarily.)


[1]
(5)
[–1]
(4)
[1]
(3)
[1]
(2)
[1]
(3)
[–1]
(2)
[0]
(1)
[0]
(1)
[1]
(2)
[0]
(1)
[0]
(1)
[0]
(1)

Right–Left:



Right–Right:



Invariant 30 (Postconditions of `RealNode.put()` with `AVLBalancer`.)
Let x be the root of a subtree on which `put()` is called and y be the node returned, that is, the root of the resulting subtree. The subtree rooted at y has no violations and the height of the subtree rooted at y is equal to or one greater than the original height of the subtree rooted at x.

Proof. *Suppose `put()` is called on node x in a BST using AVL balancing which has no violations. There are three cases: x is `nully`, x is a `RealNode` containing the key being searched for, or x is a `RealNode` with a different key. We use structural induction with the first two cases as base cases.*

Base case 1. *Suppose x is `nully`, which has height 0 Then the node y returned is a new `RealNode` with `nully` as both children, height 1, and balance 0. The subtree rooted at y has no violations and height one greater than the original height of x.*

Base case 2. *Suppose x is a `RealNode` whose key is equal to the key used for this `put()`. Then the value at node x is overwritten but node x itself is returned (so $y = x$ in this case) with the tree structure unchanged.*

Inductive case. *Suppose x is a `RealNode` and, without loss of generality, the key used for this `put()` is greater than the key at x, and so `put()` is called on the right child of x. Let h_0 be the height of the tree rooted at x before this call to `put()` on the right child, and let z the root of the subtree that results from this call to `put()` on the right child. Our inductive hypothesis is that the subtree rooted at z has no violations and that its height is equal to or one greater than the height of the original right child of x.*

Let h_1 be the height of the tree rooted at x after the call to `put()` on the right child but before the call to `putFixup()` with x. Since since at most the height of its right subtree has increased by one, either $h_1 = h_0$ or $h_1 = h_0 + 1$.

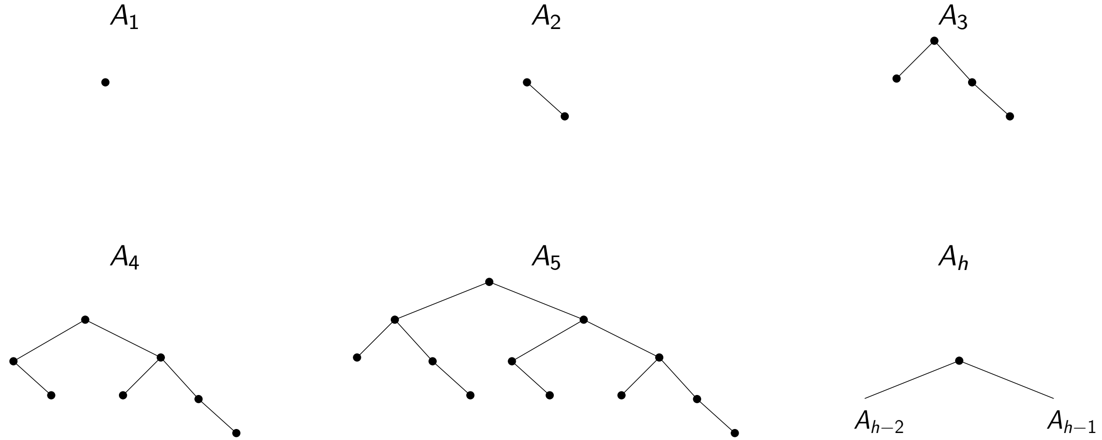
By supposition, the balance of x before the call to `put()` was no less than -1, since we supposed the tree had no violations. Since at most the height of its right subtree has increased by one, the balance of x is now no less than -2. We now have two subcases: Either the balance of x is greater than -2 or it is equal to -2.

Suppose the balance of x is greater than -2. Then the call to `putFixup()` with x returns x unchanged, which is also returned as the result of `put()` (again $y = x$), a tree with no violations and height h_1.

On the other hand, suppose the balance of x is equal to -2. Then y is a node other than x returned by `putFixup()`. Let h_2 be the height of the subtree rooted at y when `putFixup()` returns. By inspection of the right-right and right-left subcases given above, the subtree rooted at y has no violations and either $h_2 = h_1$ or $h_2 = h_1 - 1$. In either of those cases $h_2 = h_0$ or $h_2 = h_0 + 1$.

□

Let A_h be an AVL tree of height h with minimal number of nodes.



Let B_h be the number of nodes in A_h.

$$B_h = \begin{cases} 1 & \text{if } h = 1 \\ 2 & \text{if } h = 2 \\ B_{h-2} + B_{h-1} + 1 & \text{otherwise} \end{cases} \qquad B_h{+}1 = \begin{cases} 2 & \text{if } h = 1 \\ 3 & \text{if } h = 2 \\ (B_{h-2} + 1) + (B_{h-1} + 1) & \text{otherwise} \end{cases}$$

h	1	2	3	4	5	6
$B_h + 1$	2	3	5	8	13	21
B_h	1	2	4	7	12	20

$B_h + 1 = \mathrm{fib}(h+2)$. Moreover, $\mathrm{fib}(i) = [\frac{\phi^i}{\sqrt{5}}]$, and $\phi = \frac{1+\sqrt{5}}{2}$

$$\begin{aligned}
B_h + 1 &> \frac{\phi^{h+2}}{\sqrt{5}} - 1 \\
B_h + 2 &> \frac{\phi^{h+2}}{\sqrt{5}} \\
\sqrt{5}(B_h + 2) &> \phi^{h+2} \\
h + 2 &< \log_\phi(\sqrt{5}B_h) \\
h &< \log_\phi(\sqrt{5}B_h) - 2 \\
&= \log_\phi B_h + \log_\phi \sqrt{5} - 2 \\
&= \frac{1}{\lg \phi} \lg B_h + \log_\phi \sqrt{5} - 2
\end{aligned}$$

Coming up:

Do **BST rotations** *project (suggested by Wednesday, Mar 16)*

Do **AVL** *project (suggested by Monday, Mar 212)*

Due **Wed, Mar 23** *(end of day) (but spread it out)*
Read Sections 5.(4-6) [some parts carefully, some parts skim, some parts optional—see Schoology]
Do Exercise 5.14
Take quiz