Chapter 5, Binary search trees:

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

Today and Monday:

- Review BST basics
- BST performance and the balanced BST problem
- Introduction to the code base
- AVL tree definition
- AVL tree cases
- AVL tree performance

Coming up:

Catch up on older projects?
Do **BST rotations** *project (suggested by Wed, Mar 15)*
Do **AVL trees** *project (suggested by Mon, Mar 20)*

Due **Mon, Mar 13** *(end-of-day)*
Read Section 5.(1 & 2)
Do Exercises 5.(2 & 6)
Take quiz (BSTs)

Due **Thurs, Mar 16** *(end of day)*
Read Section 5.3
Do Exercises 5.(7 & 8)
Take quiz (AVL trees)

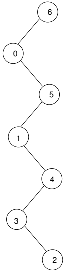
Due **Wed, Mar 22** *(end of day)—but spread it out*
Read Sections 5.(4-6)
Take quiz (red-black trees)

A **binary search tree** (BST) over some ordered key type is either

- empty, or
- a node augmented with a key k together with two children, designated *left* and *right*, such that
  - *left* is a binary search tree such that all of the keys in that tree are less than or equal to k, and
  - *right* is a binary search tree such that all of the keys in that tree are greater than or equal to k.

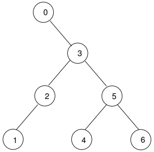
		Unsorted	Sorted
Array	Find	$\Theta(n)$	$\Theta(\lg n)$
	Insert	$\Theta(1)$ expected, $\Theta(n)$ worst	$\Theta(n)$
	Delete	$\Theta(n)$	$\Theta(n)$
Linked structure	Find	$\Theta(n)$	$\Theta(n)$
	Insert	$\Theta(1)$	$\Theta(1)$
	Delete	$\Theta(1)$	$\Theta(1)$

6, 0, 5, 1, 4, 2, 3



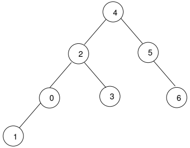
height 7
total depth 21
ANI 4

0, 3, 5, 2, 6, 1, 4



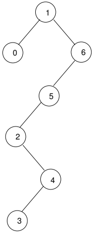
height 4
total depth 14
ANI 3

4, 2, 5, 3, 0, 1, 6



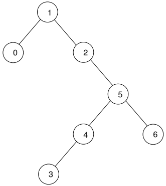
height 4
total depth 11
ANI 2.57

1, 6, 5, 2, 4, 3, 0



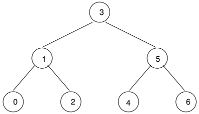
height 6
total depth 16
ANI 3.29

1, 2, 5, 4, 3, 0, 6

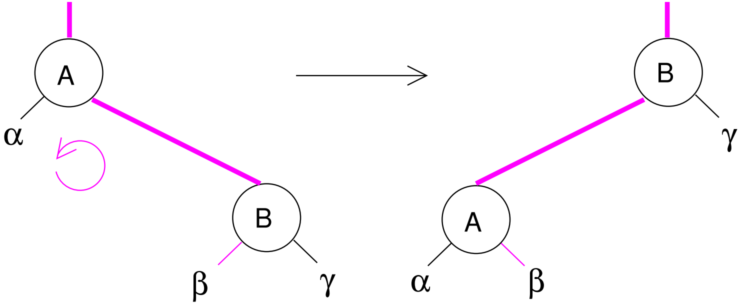


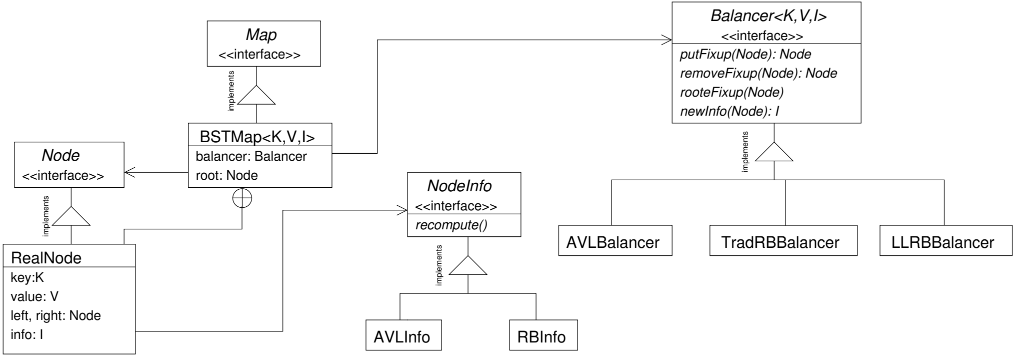
height 5
total depth 14
ANI 3

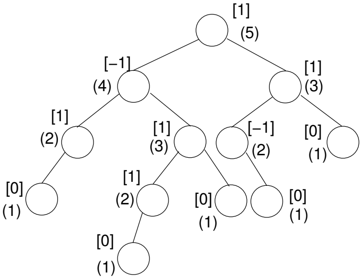
3, 1, 5, 0, 2, 4, 6



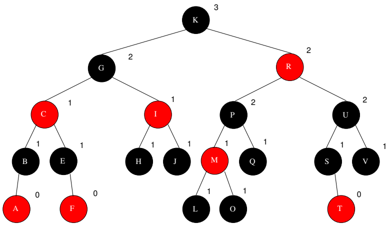
height 3
total depth 10
ANI 2.43


A
α
B
β
γ
B
γ
A
α
β

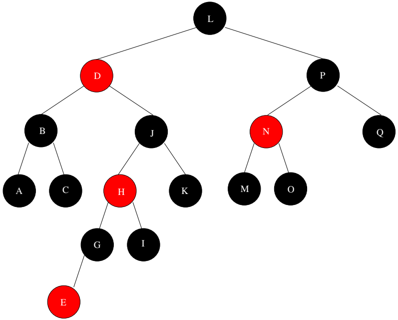

Map
<<interface>>
implements
BSTMap<K,V,I>
balancer: Balancer
root: Node
Node
<<interface>>
implements
RealNode
key:K
value: V
left, right: Node
info: I
NodeInfo
<<interface>>
recompute()
implements
AVLInfo
RBInfo
Balancer<K,V,I>
<<interface>>
putFixup(Node): Node
removeFixup(Node): Node
rooteFixup(Node)
newInfo(Node): I
implements
AVLBalancer
TradRBBalancer
LLRBBalancer



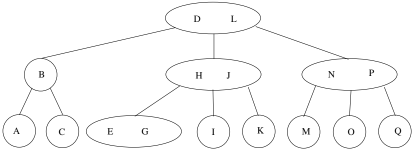
[1]
(5)
[−1]
(4)
[1]
(3)
[1]
(2)
[1]
(3)
[−1]
(2)
[0]
(1)
[0]
(1)
[1]
(2)
[0]
(1)
[0]
(1)
[0]
(1)



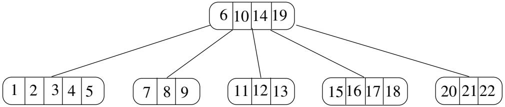
K 3
G 2
R 2
C 1
I 1
P 2
U 2
B 1
E 1
H 1
J 1
M 1
Q 1
S 1
V 1
A 0
F 0
L 1
O 1
T 0



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.

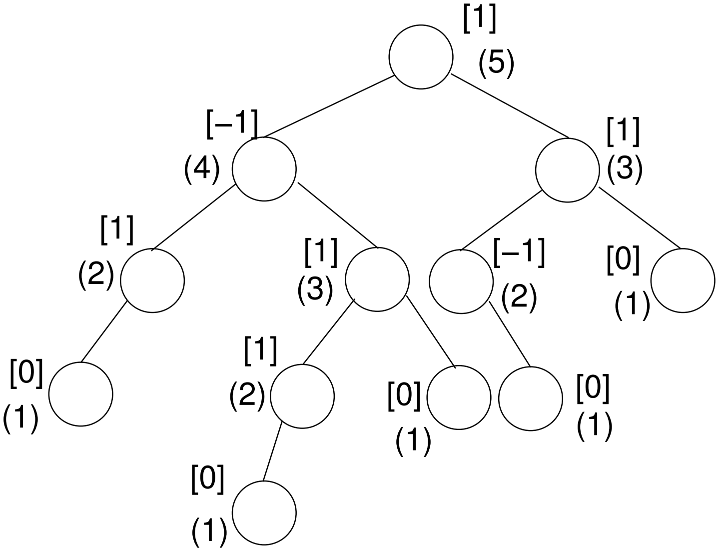
$$height(c) = \begin{cases} 0 & \text{if } c \text{ is null} \\ \max(height(c.\ell) + height(c.r)) + 1 & \text{otherwise} \end{cases}$$

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:

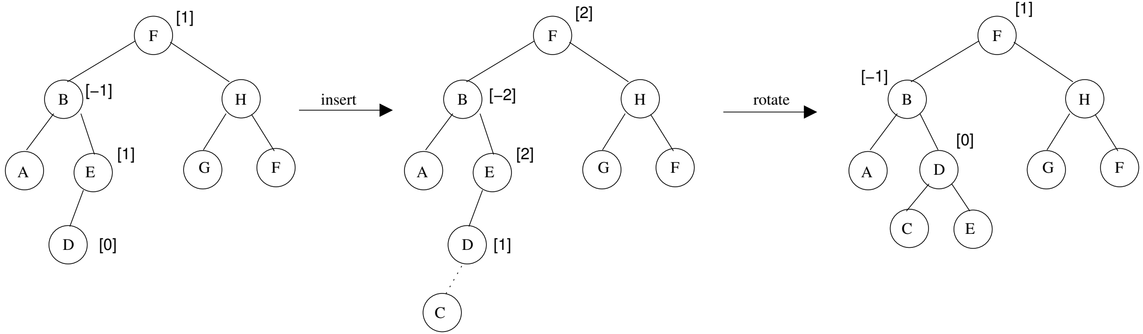
$$\forall\ x \in \text{nodes}, |height(x.\text{left}) - height(x.\text{right})| \leq 1$$

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

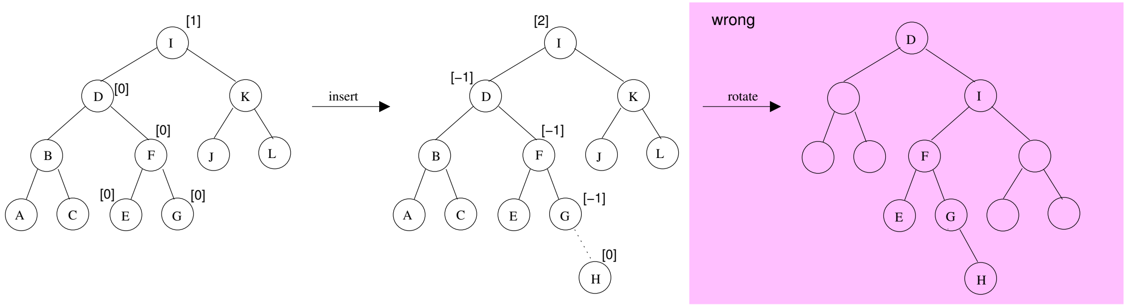
(A balance less than -2 or greater than 2 shouldn't happen even temporarily.)

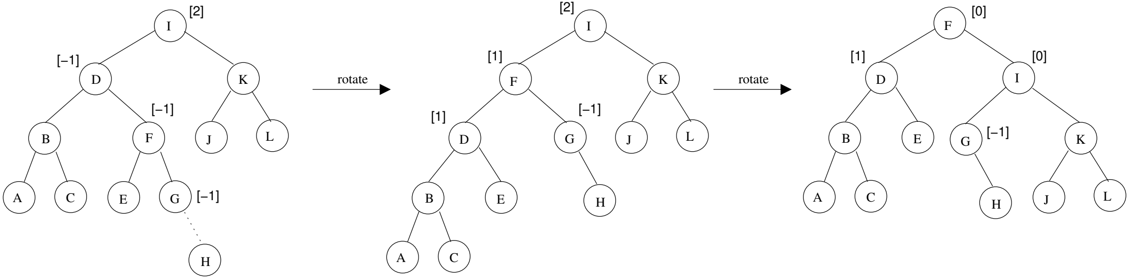


[1]
(5)
[–1]
(4)
[1]
(3)
[1]
(2)
[1]
(3)
[–1]
(2)
[0]
(1)
[0]
(1)
[1]
(2)
[0]
(1)
[0]
(1)
[0]
(1)
[0]
(1)



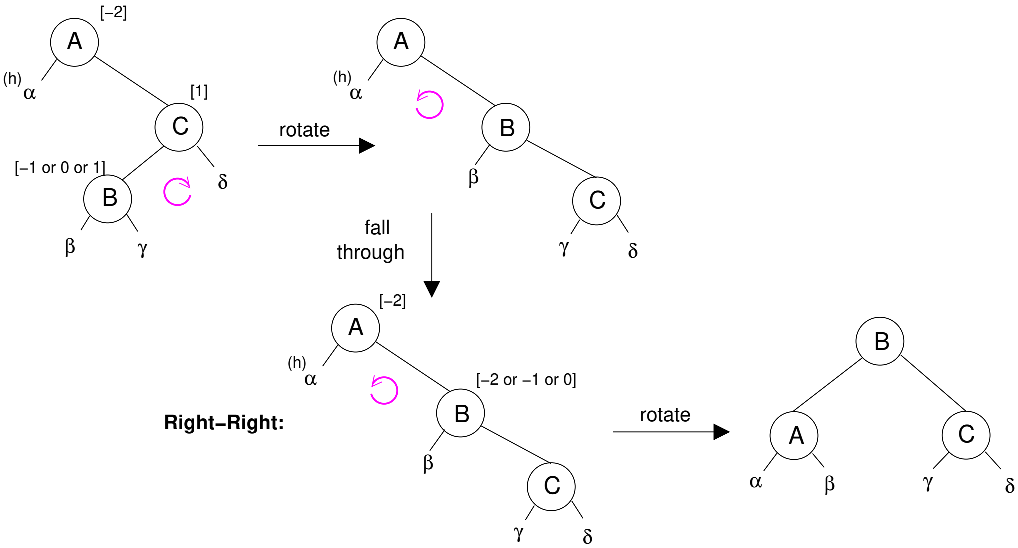
F [1]
B [−1]
H
A
E [1]
G
F
D [0]
insert
F [2]
B [−2]
H
A
E [2]
G
F
D [1]
C
rotate
F [1]
[−1] B
H
A
D [0]
G
F
C
E


[1]
I
D
[0]
K
[0]
B
F
J
L
[0]
[0]
A
C
E
G
insert
[2]
I
[−1]
D
K
[−1]
B
F
J
L
[−1]
A
C
E
G
[0]
H
rotate
wrong
D
I
F
E
G
H



[2]
I
[−1]
D
K
[−1]
B
F
J
L
A
C
E
G
[−1]
H
rotate
[2]
I
[1]
F
K
[1]
D
G
[−1]
J
L
B
E
H
A
C
rotate
F
[0]
[1]
D
I
[0]
B
E
G
[−1]
K
A
C
H
J
L

Right–Left:


[–2]
A
(h)
α
[1]
C
[–1 or 0 or 1]
B
δ
β
γ
rotate
A
(h)
α
B
β
C
γ
δ
fall
through
[–2]
A
(h)
α
[–2 or –1 or 0]
B
Right–Right:
β
C
γ
δ
rotate
B
A
C
α
β
γ
δ

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.

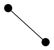
Since since at most the height of its right subtree has increased by one, either $h_1 = h_0$ or $h_1 = h_0 + 1$. By supposition, the balance of x before the call to *put()* was no less than -1, since we supposed the tree had no violations. Since at most the height of its right subtree has increased by 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.

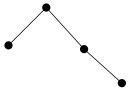
On the other hand, suppose the balance of x is equal to -2. Then y is a node other than x returned by *putFixup()*. Let h_2 be the height of the subtree rooted at y when *putFixup()* returns. By inspection of the right-right and right-left subcases given above, the subtree rooted at y has no violations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

A_1

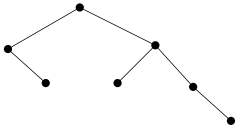
A_2



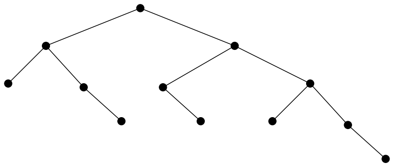
A_3



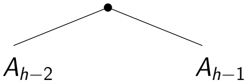
A_4



A_5



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

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