Chapter 5, Binary search trees:

- Binary search trees; the balanced BST problem (spring-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
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 \( k \).

<table>
<thead>
<tr>
<th></th>
<th>Unsorted</th>
<th>Sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Array</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Find</td>
<td>( \Theta(n) )</td>
<td>( \Theta(lg\ n) )</td>
</tr>
<tr>
<td>Insert</td>
<td>( \Theta(1) ) expected, ( \Theta(n) ) worst</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>Delete</td>
<td>( \Theta(n) )</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td><strong>Linked structure</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Find</td>
<td>( \Theta(n) )</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>Insert</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>Delete</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
</tbody>
</table>
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.)
```
A  E  F
B  H  C
D  G  F

rotateinsert
```

```
A  E  F
B  H  C
D  G  F
```

```
A  E  F
B  H  C
D  G  F
```

```
A  E  F
B  H  C
D  G  F
```

```
A  E  F
B  H  C
D  G  F
```

```
A  E  F
B  H  C
D  G  F
```

```
A  E  F
B  H  C
D  G  F
```

```
A  E  F
B  H  C
D  G  F
```

```
A  E  F
B  H  C
D  G  F
```

```
A  E  F
B  H  C
D  G  F
```

```
A  E  F
B  H  C
D  G  F
```

```
A  E  F
B  H  C
D  G  F
```

```
A  E  F
B  H  C
D  G  F
```

```
A  E  F
B  H  C
D  G  F
```

```
A  E  F
B  H  C
D  G  F
```

```
A  E  F
B  H  C
D  G  F
```

```
A  E  F
B  H  C
D  G  F
```

```
A  E  F
B  H  C
D  G  F
```

```
A  E  F
B  H  C
D  G  F
```
The diagram illustrates a binary search tree (BST) and the process of inserting and rotating nodes. Initially, the tree is balanced with nodes A, B, C, D, and I. When node E is inserted as node K's child, the tree becomes unbalanced. To correct this, a right rotation is performed at node K, ensuring the tree remains a valid BST.

The process can be summarized as follows:

1. Initial balanced BST with nodes A, B, C, D, and I.
2. Insertion of node E (as node K's child) causes the tree to become unbalanced.
3. Right rotation at node K to balance the tree.

The diagrams on the right highlight the incorrect and correct steps involved in the insertion and rotation process.
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 null, $x$ 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 $\text{put()}$ on the right child but before the call to $\text{putFixup()}$ 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 $\text{put()}$ 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 $\text{putFixup()}$ with $x$ returns $x$ unchanged, which is also returned as the result of $\text{put()}$ (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 $\text{putFixup()}$. Let $h_2$ be the height of the subtree rooted at $y$ when $\text{putFixup()}$ 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 $h_2 = h_0 + 1$. □
Coming up:

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

Due **Tues, Mar 15** (end of day)
Read Section 5.3
Do Exercises 5.(7 & 8)
Take quiz

*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
\[ \begin{align*}
B_h &= \begin{cases} 
1 & \text{if } h = 1 \\
2 & \text{if } h = 2 \\
B_{h-2} + B_{h-1} + 1 & \text{otherwise}
\end{cases} \\
B_{h+1} &= \begin{cases} 
2 & \text{if } h = 1 \\
3 & \text{if } h = 2 \\
(B_{h-2} + 1) + (B_{h-1} + 1) & \text{otherwise}
\end{cases}
\end{align*} \]
\[ B_h + 1 > \frac{\phi^{h+2}}{\sqrt{5}} - 1 \]

\[ B_h + 2 > \frac{\phi^{h+2}}{\sqrt{5}} \]

\[ \sqrt{5}(B_h + 2) > \phi^{h+2} \]

\[ h + 2 < \log_\phi(\sqrt{5}B_h) \]

\[ h < \log_\phi(\sqrt{5}B_h) - 2 \]

\[ = \log_\phi B_h + \log_\phi \sqrt{5} - 2 \]

\[ = \frac{1}{\lg \phi} \lg B_h + \log_\phi \sqrt{5} - 2 \]