Chapter 5, Binary search trees:

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

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 Mon, Oct 24)
* Do **AVL trees** project (suggested by Fri, Oct 28)

**Due Fri, Oct 21** (class time)
Read Section 5.(1 & 2)
Do Exercises 5.(2 & 6)
Take quiz

**Due Tues, Oct 25** (end of day)
Read Section 5.3
Do Exercises 5.(7 & 8)
Take quiz

**Due Monday, Oct 31** (end of day)—but spread it out
Read Sections 5.(4-6)
Take quiz
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>Find</strong></td>
<td>$\Theta(n)$</td>
<td>$\Theta(\lg n)$</td>
</tr>
<tr>
<td><strong>Array</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Insert</strong></td>
<td>$\Theta(1)$ expected, $\Theta(n)$ worst</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td><strong>Delete</strong></td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td><strong>Linked structure</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Find</strong></td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td><strong>Insert</strong></td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td><strong>Delete</strong></td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>
Indicate the worst-case and best-case running times for a `get()` operation on a map implemented by each of the following data structures.

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Worst case</th>
<th>Best case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>LinkedList</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>BST, worst-case structure</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>BST, best-case structure</td>
<td>$\Theta(\lg n)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>
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
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:

\[
\forall x \in \text{nodes}, \ |height(x.\ell) - height(x.r)| \leq 1
\]

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.)
The image contains a tree structure with nodes labeled with values from a list. The tree is rooted at the top node labeled with [1] (5). The tree branches down with nodes labeled [1] (4), [-1] (3), and [0] (2) at various levels. Each node is connected by lines indicating the parent-child relationship. The values within the nodes are enclosed in brackets, and the numbers in parentheses indicate the level of each node in the tree.
The diagram shows a tree with nodes labeled from A to I, and some nodes have values [0] or [1]. The diagram illustrates the process of inserting and rotating the tree. The left part of the diagram shows the original tree, the middle part shows the tree after an insertion, and the right part shows the tree after a rotation and highlighting the 'wrong' subtree. The values [0] and [1] are annotations for the nodes.
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 \texttt{null}, which has height 0. Then the node \( y \) returned is a new \texttt{RealNode} with \texttt{null} 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 \texttt{RealNode} whose key is equal to the key used for this \texttt{put()} method. 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 \texttt{RealNode} and, without loss of generality, the key used for this \texttt{put()} method is greater than the key at \( x \), and so \texttt{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 \texttt{put()} on the right child, and let \( z \) the root of the subtree that results from this call to \texttt{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 $h_2 = h_0 + 1$. □
\[
B_h = \begin{cases} 
1 & \text{if } h = 1 \\
2 & \text{if } h = 2 \\
B_{h-2} + B_{h-1} + 1 & \text{otherwise}
\end{cases}
\]

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

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

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

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

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

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

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