9 AVL Trees Written by Irina Galata

In the previous chapter, you learned about the *O*(log *n*) performance characteristics of the binary search tree. However, you also learned that *unbalanced* trees could deteriorate the performance of the tree, all the way down to *O*(*n*).

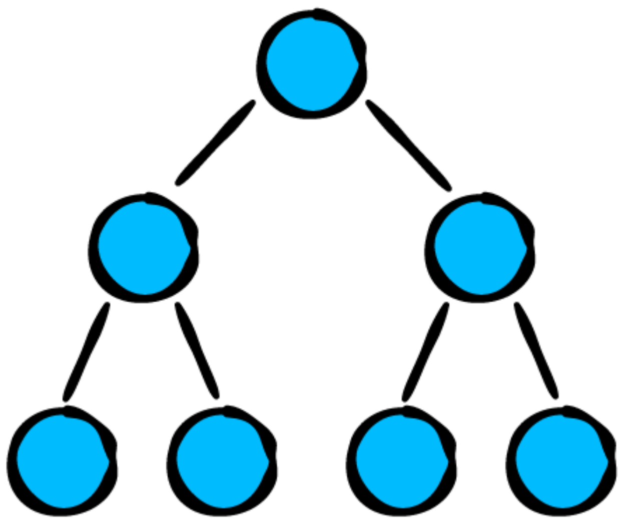
In 1962, Georgy Adelson-Velsky and Evgenii Landis came up with the first *self-balancing* binary search tree: the **AVL tree**. In this chapter, you'll dig deeper into how the balance of a binary search tree can impact performance and implement the AVL tree from scratch.

Understanding balance

A balanced tree is the key to optimizing the performance of the binary search tree. There are three main states of balance. You'll look at each one.

Perfect balance

The ideal form of a binary search tree is the **perfectly balanced** state. In technical terms, this means every level of the tree is filled with nodes from top to bottom.

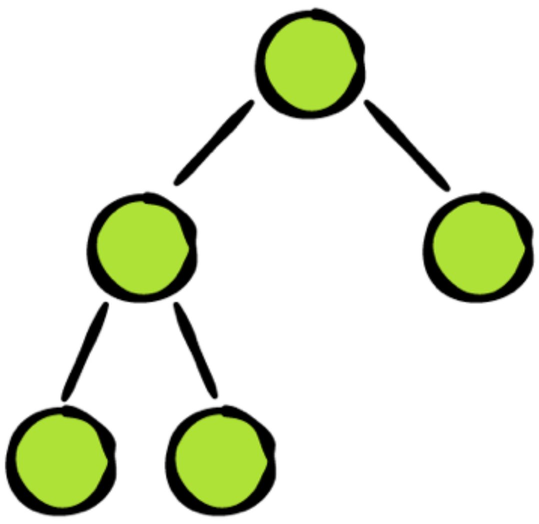


A perfectly balanced tree

Not only is the tree perfectly symmetrical, but the nodes at the bottom level are also completely filled. Note that perfect balanced trees can just have a specific number of nodes. For instance 1, 3, or 7 are possible numbers of nodes because they can fill 1, 2, or 3 levels respectively. This is the requirement for being perfectly balanced.

“Good-enough” balance

Although achieving perfect balance is ideal, it’s rarely possible because it also depends on the specific number of nodes. A tree with 2, 4, 5, or 6 cannot be perfectly balanced since the last level of the tree will not be filled.

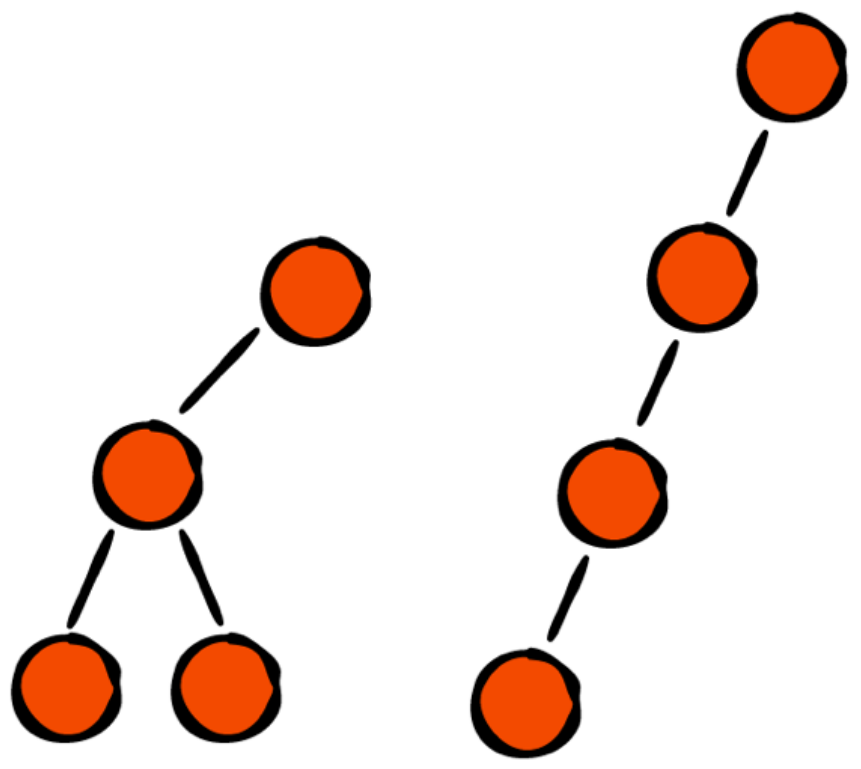


A balanced tree

Because of this, a different definition exists. A **balanced tree** must have all its levels filled, except for the bottom one. In most cases of binary trees, this is the best you can do.

Unbalanced

Finally, there’s the **unbalanced** state. Binary search trees in this state suffer from various levels of performance loss depending on the degree of imbalance.



Some unbalanced trees

Keeping the tree balanced gives the **find**, **insert**, and **remove** operations an *O*(log *n*) time complexity. AVL trees maintain balance by adjusting the structure of the tree when the tree becomes unbalanced. You'll learn how this works as you progress through the chapter.

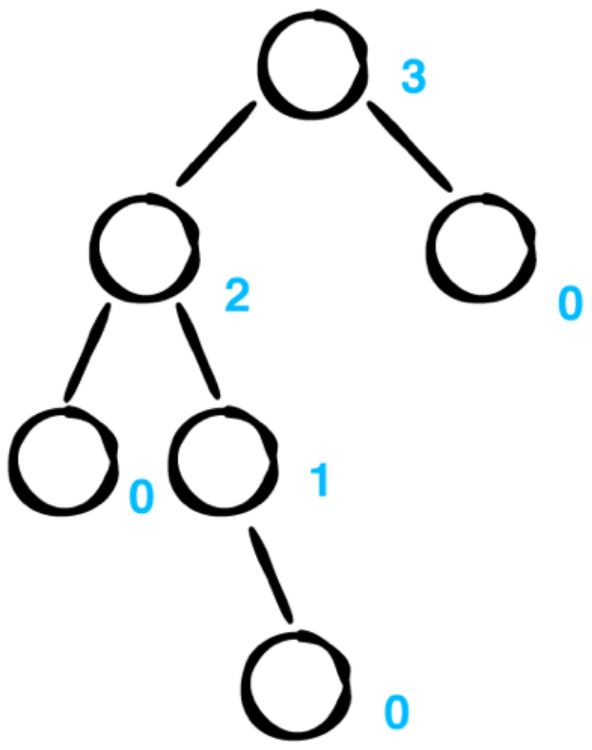
Implementation

Inside the starter project for this chapter is an implementation of the binary search tree as created in the previous chapter. The only difference is that all references to the binary search tree have been renamed to AVL tree.

Binary search trees and AVL trees share much of the same implementation; in fact, all that you'll add is the balancing component. Open the starter project to begin.

Measuring balance

To keep a binary tree balanced, you need a way to measure the balance of the tree. The AVL tree achieves this with a `height` property in each node. In tree-speak, the **height** of a node is the **longest** distance from the current node to a leaf node:



Nodes marked with heights

With the starter project for this chapter open the **AVLNode.kt** file, add the following property to the `AVLNode` class:

```
var height = 0
```

You'll use the *relative* heights of a node's children to determine whether a particular node is balanced.

The height of the left and right children of each node must differ at most by 1. This is known as the **balance factor**.

Write the following immediately below the `height` property of `AVLNode`:

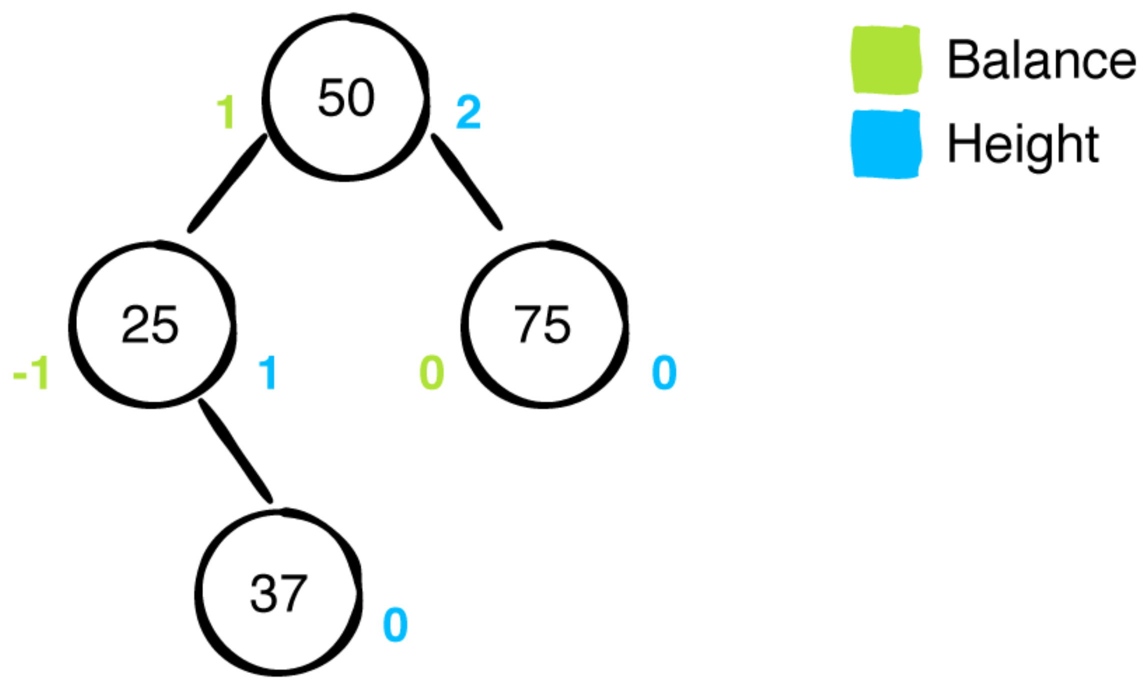
```
val leftHeight: Int
  get() = leftChild?.height ?: -1

val rightHeight: Int
  get() = rightChild?.height ?: -1

val balanceFactor: Int
  get() = leftHeight - rightHeight
```

The `balanceFactor` computes the height difference of the left and right child. If a particular child is `null`, its height is considered to be `-1`.

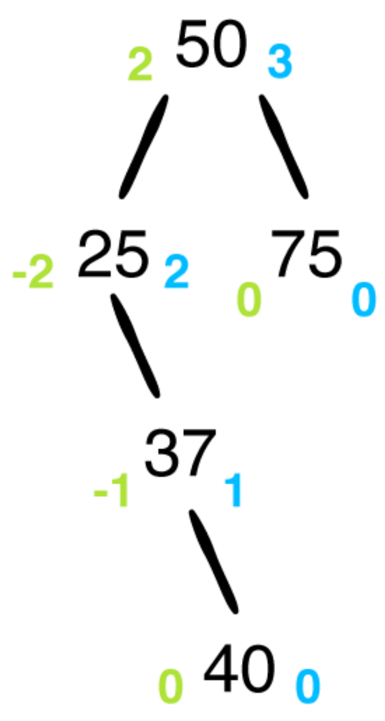
Here's an example of an AVL tree:



AVL tree with balance factors and heights

This is a balanced tree — all levels except the bottom one are filled. The blue numbers represent the `height` of each node, while the green numbers represent the `balanceFactor`.

Here's an updated diagram with **40** inserted:



Unbalanced tree

Inserting **40** into the tree turns it into an unbalanced tree. Notice how the `balanceFactor` changes. A `balanceFactor` of **2** or **-2** is an indication of an unbalanced tree.

Although more than one node may have a bad balancing factor, you only need to perform the balancing procedure on the bottom-most node containing the invalid balance factor: the node containing **25**.

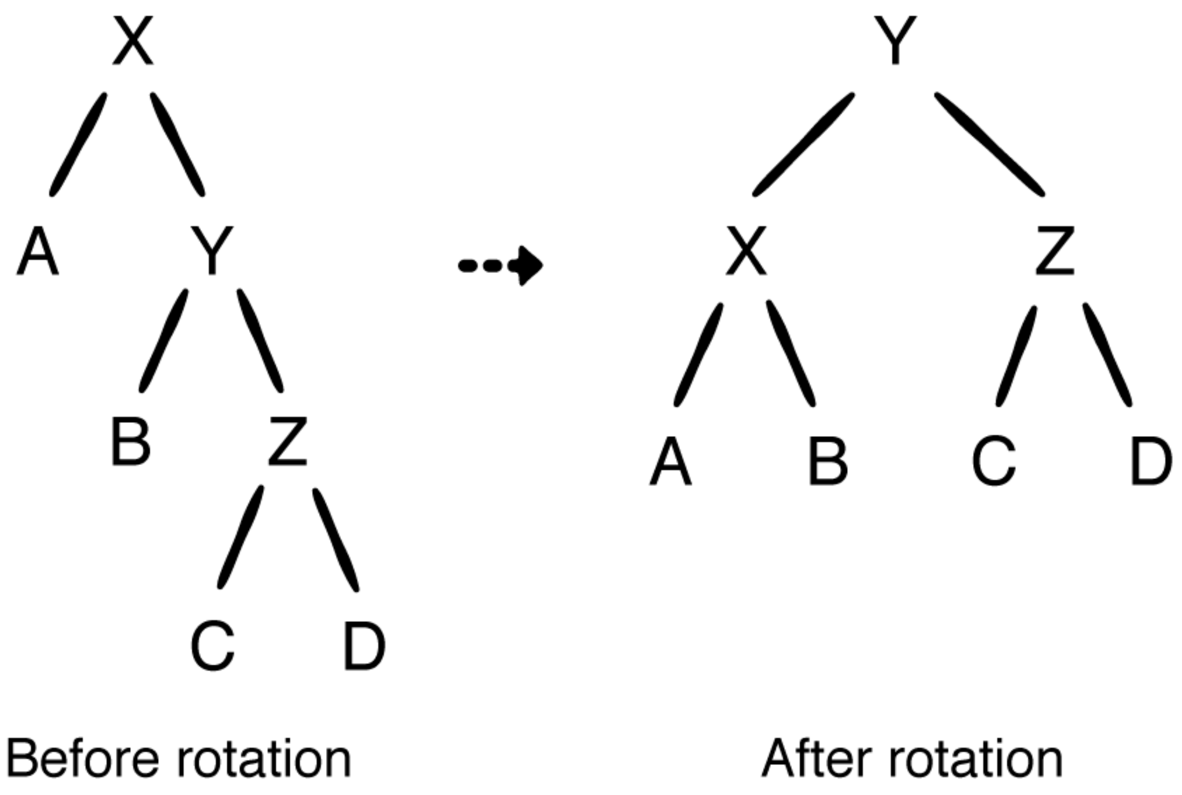
That’s where rotations come in.

Rotations

The procedures used to balance a binary search tree are known as **rotations**. There are four rotations in total, one for each way that a tree can become unbalanced. These are known as **left** rotation, **left-right** rotation, **right** rotation and **right-left** rotation.

Left rotation

You can solve the imbalance caused by inserting **40** into the tree using a **left rotation**. A generic left rotation of node **X** looks like this:



Left rotation applied on node X

Before going into specifics, there are two takeaways from this before-and-after comparison:

- In-order traversal for these nodes remains the same.
- The *depth* of the tree is reduced by one level after the rotation.

Add the following method to `AVLTree`:

```
private fun leftRotate(node: AVLNode<T>): AVLNode<T> {
  // 1
  val pivot = node.rightChild!!
  // 2
  node.rightChild = pivot.leftChild
  // 3
  pivot.leftChild = node
  // 4
  node.height = max(node.leftHeight, node.rightHeight) + 1
  pivot.height = max(pivot.leftHeight, pivot.rightHeight) + 1
  // 5
  return pivot
}
```

Here are the steps needed to perform a left rotation:

1. The right child is chosen as the **pivot**. This node replaces the rotated node as the root of the subtree (it moves up a level).

2. The node to be rotated becomes the left child of the pivot (it moves down a level). This means that the current left child of the pivot must be moved elsewhere.

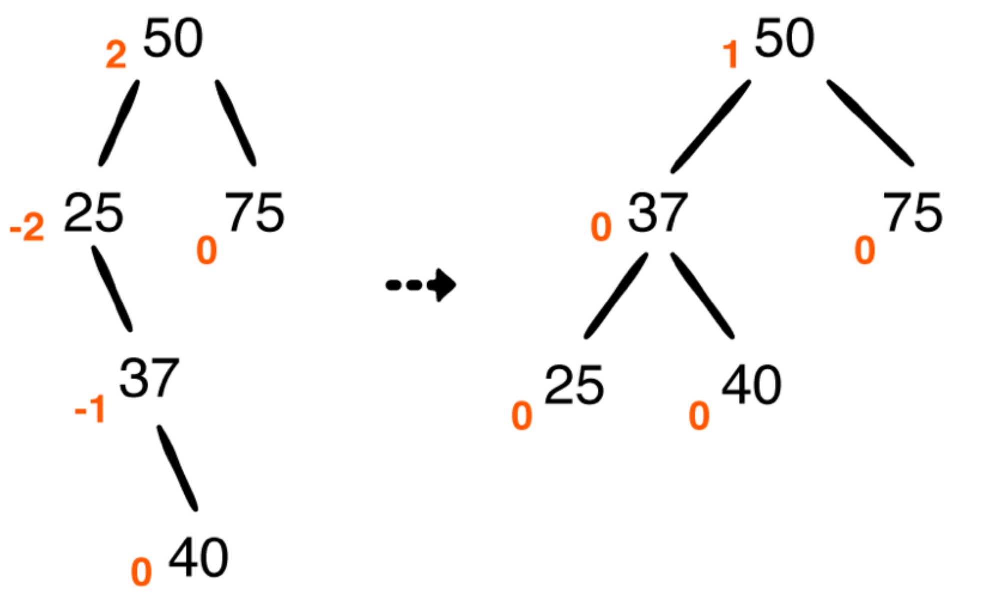
   In the generic example shown in the earlier image, this is node **b**. Because **b** is smaller than **y** but greater than **x**, it can replace **y** as the right child of **x**. So you update the rotated node's `rightChild` to the pivot's `leftChild`.

3. The pivot's `leftChild` can now be set to the rotated node.

4. You update the heights of the rotated node and the pivot.

5. Finally, you return the pivot so that it can replace the rotated node in the tree.

Here are the before-and-after effects of the left rotation of **25** from the previous example:

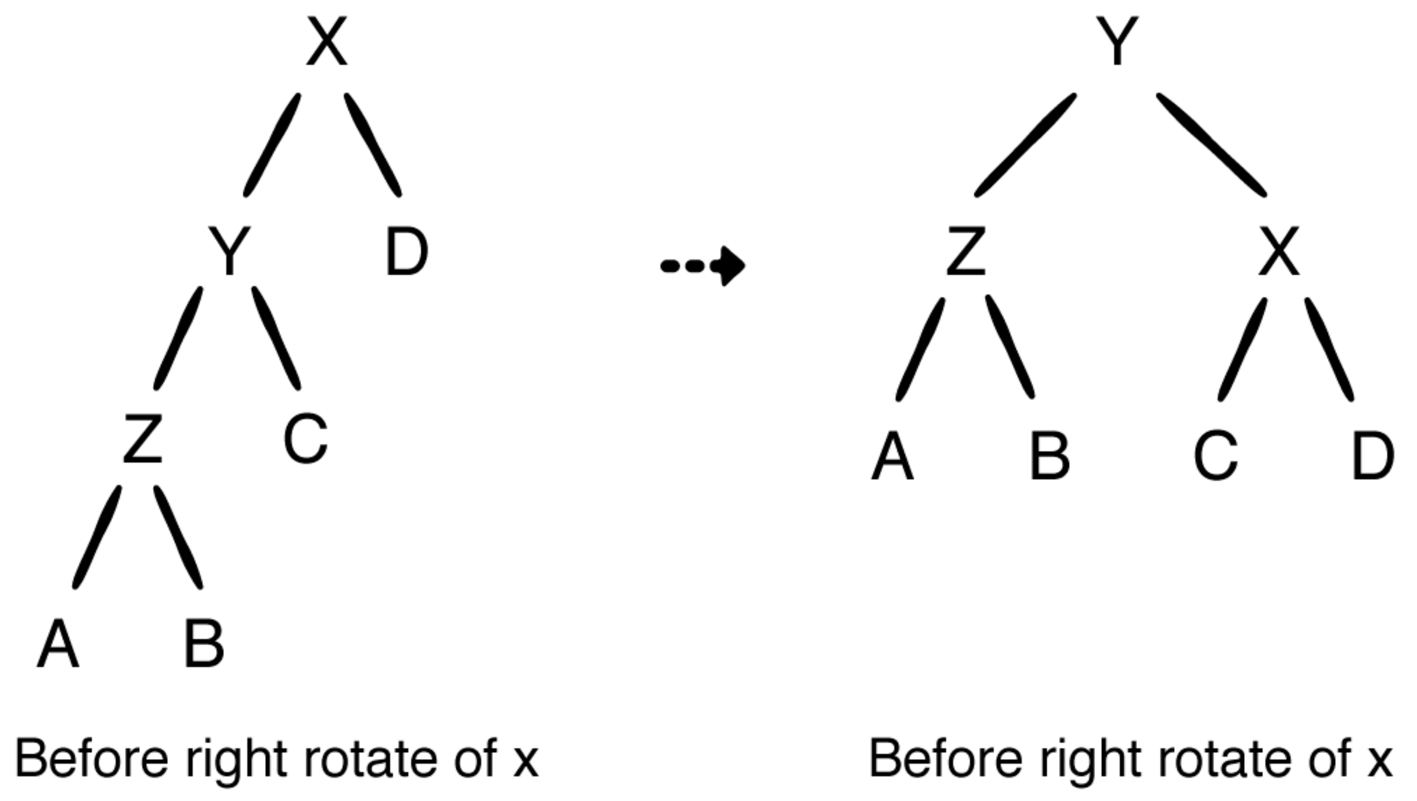


Before left rotate on 25 After left rotate on 25

Right rotation

Right rotation is the symmetrical opposite of left rotation. When a series of left children is causing an imbalance, it's time for a right rotation.

A generic right rotation of node **X** looks like this:



Right rotation applied on node X

To implement this, add the following code just after `leftRotate()`:

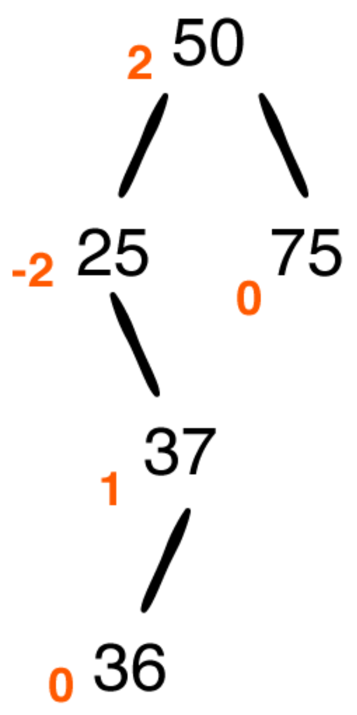
```
private fun rightRotate(node: AVLNode<T>): AVLNode<T> {
  val pivot = node.leftChild!!
  node.leftChild = pivot.rightChild
  pivot.rightChild = node
  node.height = max(node.leftHeight, node.rightHeight) + 1
  pivot.height = max(pivot.leftHeight, pivot.rightHeight) + 1
  return pivot
}
```

This is nearly identical to the implementation of `leftRotate()`, except the references to the left and right children have been swapped.

Right-left rotation

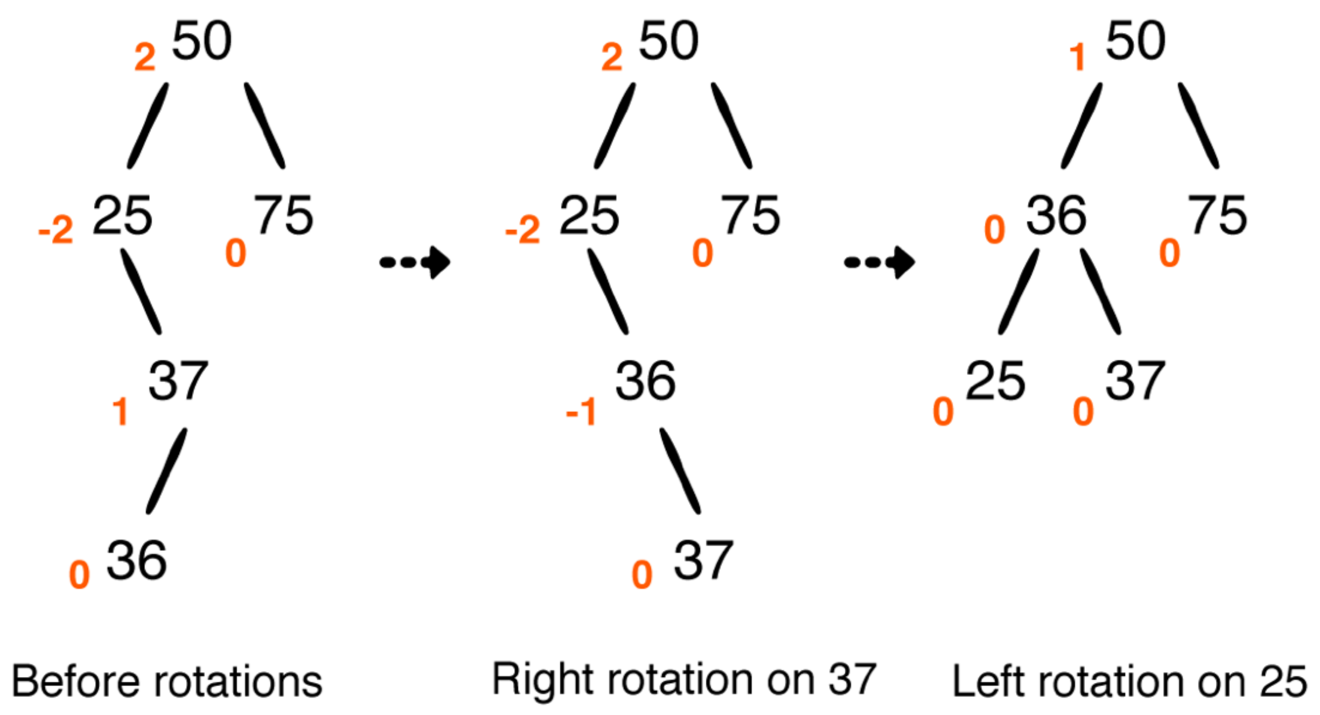
You may have noticed that the left and right rotations balance nodes that are all left children or all right children. Consider the case in which **36** is inserted into the original example tree.

The right-left rotation:



Inserted 36 as left child of 37

Doing a left rotation, in this case, won't result in a balanced tree. The way to handle cases like this is to perform a right rotation on the right child *before* doing the left rotation. Here's what the procedure looks like:



The right-left rotation

1. You apply a right rotation to **37**.
2. Now that nodes **25**, **36** and **37** are all right children, you can apply a left rotation to balance the tree.

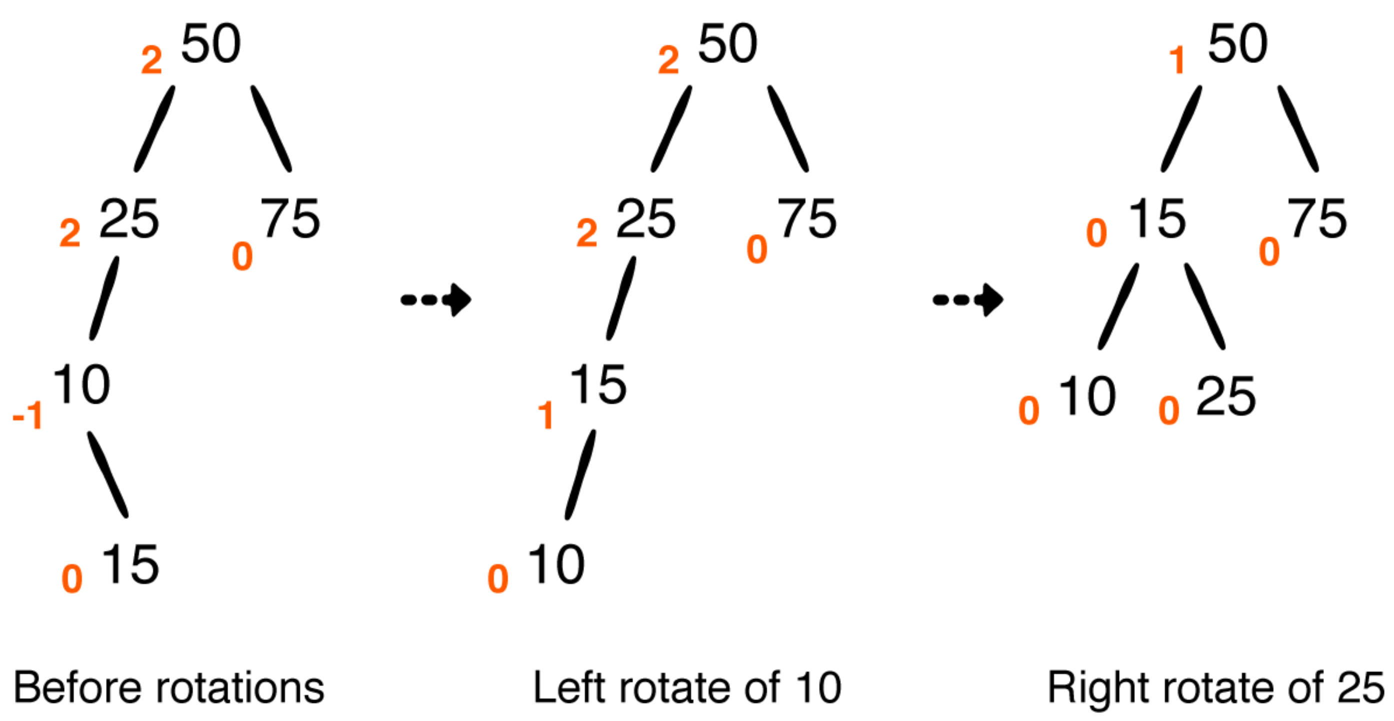
Add the following code immediately after `rightRotate()`:

```
private fun rightLeftRotate(node: AVLNode<T>): AVLNode<T> {
  val rightChild = node.rightChild ?: return node
  node.rightChild = rightRotate(rightChild)
  return leftRotate(node)
}
```

Don't worry just yet about when this is called. You'll get to that in a second. You first need to handle the last case, left-right rotation.

Left-right rotation

Left-right rotation is the symmetrical opposite of the right-left rotation. Here's an example:



The left-right rotation

1. You apply a left rotation to node **10**.
2. Now that nodes **25**, **15** and **10** are all left children, you can apply a right rotation to balance the tree.

Add the following code immediately after `rightLeftRotate()`:

```
private fun leftRightRotate(node: AVLNode<T>): AVLNode<T> {
  val leftChild = node.leftChild ?: return node
  node.leftChild = leftRotate(leftChild)
  return rightRotate(node)
}
```

That’s it for rotations. Next, you’ll figure out when to apply these rotations at the correct location.

Balance

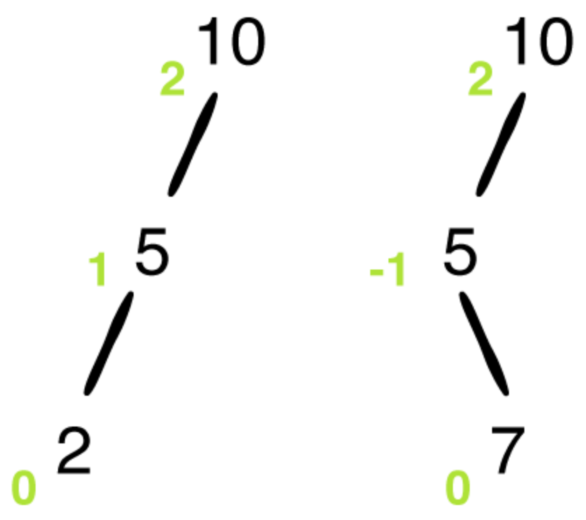
The next task is to design a method that uses `balanceFactor` to decide whether a node requires balancing or not. Write the following method below `leftRightRotate()`:

```
private fun balanced(node: AVLNode<T>): AVLNode<T> {
  return when (node.balanceFactor) {
    2 -> {
    }
    -2 -> {
    }
    else -> node
  }
}
```

There are three cases to consider.

1. A `balanceFactor` of **2** suggests that the left child is heavier (that is, contains more nodes) than the right child. This means that you want to use either right or left-right rotations.
2. A `balanceFactor` of **-2** suggests that the right child is heavier than the left child. This means that you want to use either left or right-left rotations.
3. The default case suggests that the particular node is balanced. There’s nothing to do here except to return the node.

You can use the sign of the `balanceFactor` to determine if a single or double rotation is required:



Right rotate, or left right rotate?

Update the `balanced` function to the following:

```
private fun balanced(node: AVLNode<T>): AVLNode<T> {
  return when (node.balanceFactor) {
    2 -> {
      if (node.leftChild?.balanceFactor == -1) {
        leftRightRotate(node)
      } else {
        rightRotate(node)
      }
    }
    -2 -> {
      if (node.rightChild?.balanceFactor == 1) {
        rightLeftRotate(node)
      } else {
        leftRotate(node)
      }
    }
    else -> node
  }
}
```

`balanced()` inspects the `balanceFactor` to determine the proper course of action. All that's left is to call `balanced()` at the proper spot.

Revisiting insertion

You've already done the majority of the work. The remainder is fairly straightforward. Update `insert()` to the following:

```
private fun insert(node: AVLNode<T>?, value: T): AVLNode<T>? {
  node ?: return AVLNode(value)
  if (value < node.value) {
    node.leftChild = insert(node.leftChild, value)
  } else {
    node.rightChild = insert(node.rightChild, value)
  }
  val balancedNode = balanced(node)
  balancedNode?.height = max(balancedNode?.leftHeight ?: 0, balancedNode?.r
  return balancedNode
}
```

Instead of returning the `node` directly after inserting, you pass it into `balanced()`. This ensures every node in the call stack is checked for balancing issues. You also update the node's height.

Time to test it. Go to **Main.kt** and add the following to `main()`:

```
"repeated insertions in sequence" example {
  val tree = AVLTree<Int>()

  (0..14).forEach {
    tree.insert(it)
  }

  print(tree)
}
```

You'll see the following output in the console:

```
---Example of repeated insertions in sequence---
  ┌──14
 ┌──13
 │ └──12
┌──11
│ │ ┌──10
│ └──9
│  └──8
7
│  ┌──6
│ ┌──5
│ │ └──4
└──3
 │ ┌──2
 └──1
  └──0
```

Take a moment to appreciate the uniform spread of the nodes. If the rotations weren't applied, this would have become a long, unbalanced chain of right children.

Revisiting remove

Retrofitting the `remove` operation for self-balancing is just as easy as fixing insert. In `AVLTree`, find `remove` and replace the final `return` statement with the following:

```
val balancedNode = balanced(node)
balancedNode.height = max(
    balancedNode.leftHeight,
    balancedNode.rightHeight
) + 1
return balancedNode
```

Go back to `main()` in **Main.kt** and add the following code:

```
val tree = AVLTree<Int>()
tree.insert(15)
tree.insert(10)
tree.insert(16)
tree.insert(18)
print(tree)
tree.remove(10)
print(tree)
```

You’ll see the following console output:

```
---Example of removing a value---
 ┌──18
┌──16
│ └──null
15
└──10

┌──18
16
└──15
```

Removing **10** caused a left rotation on **15**. Feel free to try out a few more test cases of your own.

Whew! The AVL tree is the culmination of your search for the ultimate binary search tree. The self-balancing property guarantees that the `insert` and `remove` operations function at optimal performance with an *O*(log *n*) time complexity.

Challenges

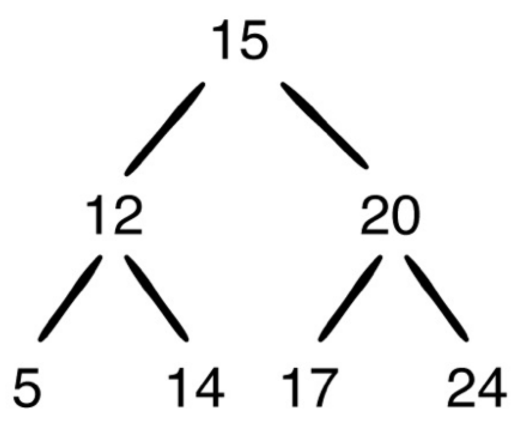
Here are three challenges that revolve around AVL trees. Solve these to make sure you understand the concepts.

Challenge 1: Count the leaves

How many **leaf** nodes are there in a perfectly balanced tree of height 3? What about a perfectly balanced tree of height `h`?

Solution 1

A perfectly balanced tree is a tree where all of the leaves are at the same level, and that level is completely filled:



Recall that a tree with only a root node has a height of zero. Thus, the tree in the example above has a height of two. You can extrapolate that a tree with a height of three would have **eight** leaf nodes.

Since each node has two children, the number of leaf nodes doubles as the height increases. Therefore, you can calculate the number of leaf nodes using a simple equation:

```
fun leafNodes(height: Int): Int {
  return 2.0.pow(height).toInt()
}
```

Challenge 2: Count the nodes

How many **nodes** are there in a perfectly balanced tree of height 3? What about a perfectly balanced tree of height `h`?

Solution 2

Since the tree is perfectly balanced, you can calculate the number of nodes in a perfectly balanced tree of height three using the following:

```
fun nodes(height: Int): Int {
  var totalNodes = 0
  (0..height).forEach { currentHeight ->
    totalNodes += 2.0.pow(currentHeight).toInt()
  }
  return totalNodes
}
```

Although this certainly gives you the correct answer, there's a faster way. If you examine the results of a sequence of height inputs, you'll realize that the total number of nodes is one less than the number of leaf nodes of the next level.

The previous solution is O(height) but here's a faster version of this in O(1):

```
fun nodes(height: Int): Int {
  return 2.0.pow(height + 1).toInt() - 1
}
```

Challenge 3: Some refactoring

Since there are many variants of binary trees, it makes sense to group shared functionality in an abstract class. The traversal methods are a good candidate.

Create a `TraversableBinaryNode` abstract class that provides a default implementation of the traversal methods so that concrete subclasses get these methods for free. Have `AVLNode` extend this class.

Solution 3

First, create the following abstract class:

```
abstract class TraversableBinaryNode<Self :
  TraversableBinaryNode<Self, T>, T>(var value: T) {

  var leftChild: Self? = null
  var rightChild: Self? = null

  fun traverseInOrder(visit: Visitor<T>) {
    leftChild?.traverseInOrder(visit)
    visit(value)
    rightChild?.traverseInOrder(visit)
  }

  fun traversePreOrder(visit: Visitor<T>) {
    visit(value)
    leftChild?.traversePreOrder(visit)
    rightChild?.traversePreOrder(visit)
  }

  fun traversePostOrder(visit: Visitor<T>) {
    leftChild?.traversePostOrder(visit)
    rightChild?.traversePostOrder(visit)
    visit(value)
  }
}
```

Finally, add the following at the bottom of `main()`:

```
"using TraversableBinaryNode" example {
  val tree = AVLTree<Int>()
  (0..14).forEach {
    tree.insert(it)
  }
  println(tree)
  tree.root?.traverseInOrder { println(it) }
}
```

You'll see the following results in the console:

```
---Example of using TraversableBinaryNode---
  ┌──14
 ┌──13
 │ └──12
```

```
┌──11
│ │ ┌──10
│ └──9
│   └──8
7
│   ┌──6
│ ┌──5
│ │ └──4
└──3
  │ ┌──2
  └──1
    └──0
```

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

Key points

- A self-balancing tree avoids performance degradation by performing a balancing procedure whenever you add or remove elements in the tree.
- AVL trees preserve balance by readjusting parts of the tree when the tree is no longer balanced.

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.