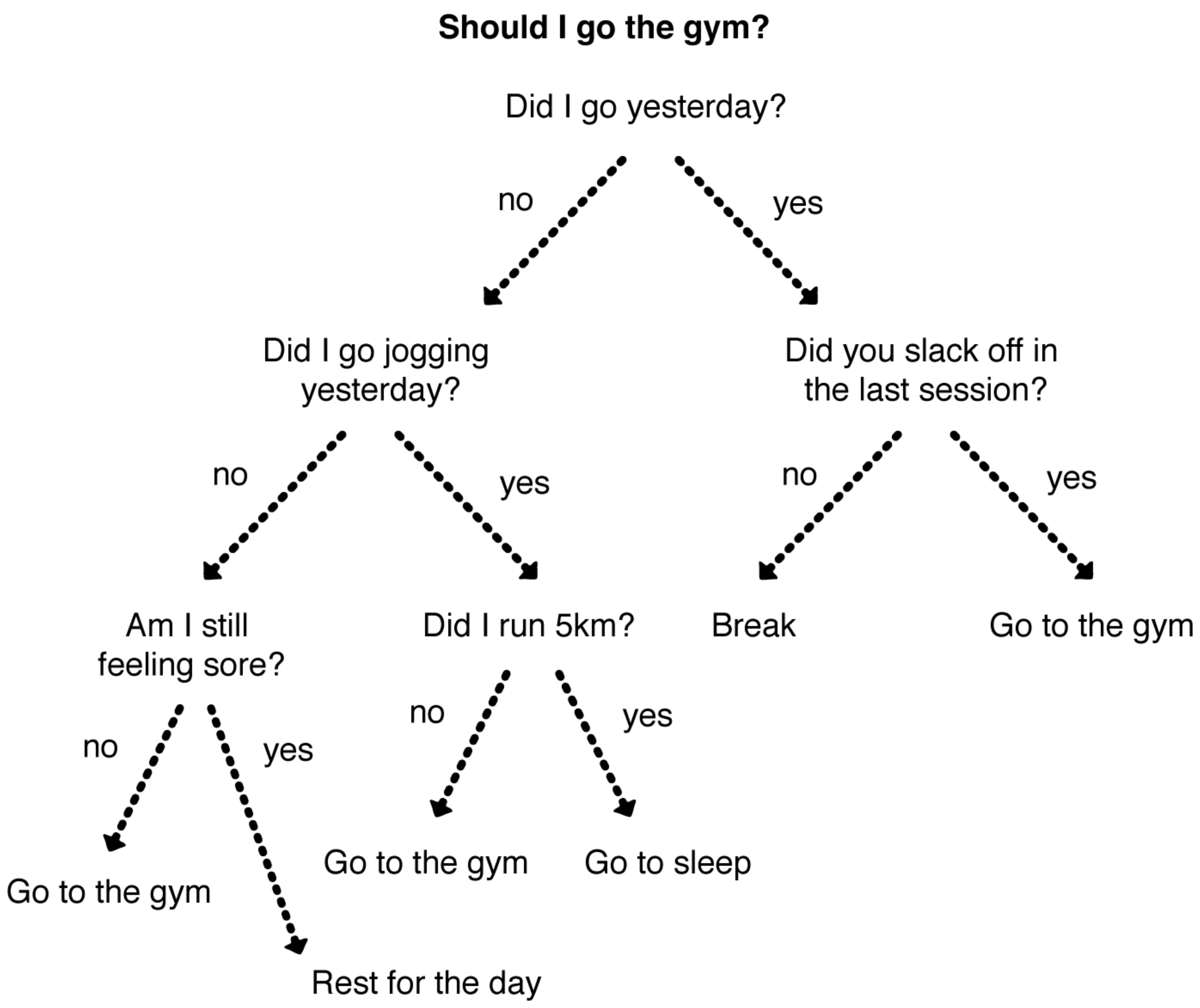
8 Binary Search Trees Written by Irina Galata

A **binary search tree**, or **BST**, is a data structure that facilitates fast lookup, insert, and removal operations. Consider the following decision tree where picking a side forfeits all of the possibilities of the other side, cutting the problem in half.



Once you make a decision and choose a branch, there's no looking back. You keep going until you make a final decision at a leaf node. Binary trees let you do the same thing. Specifically, a binary search tree imposes two rules on the binary tree you saw in the previous chapter:

- The value of a **left child** must be less than the value of its **parent**.

- Consequently, the value of a **right child** must be greater than or equal to the value of its **parent**.

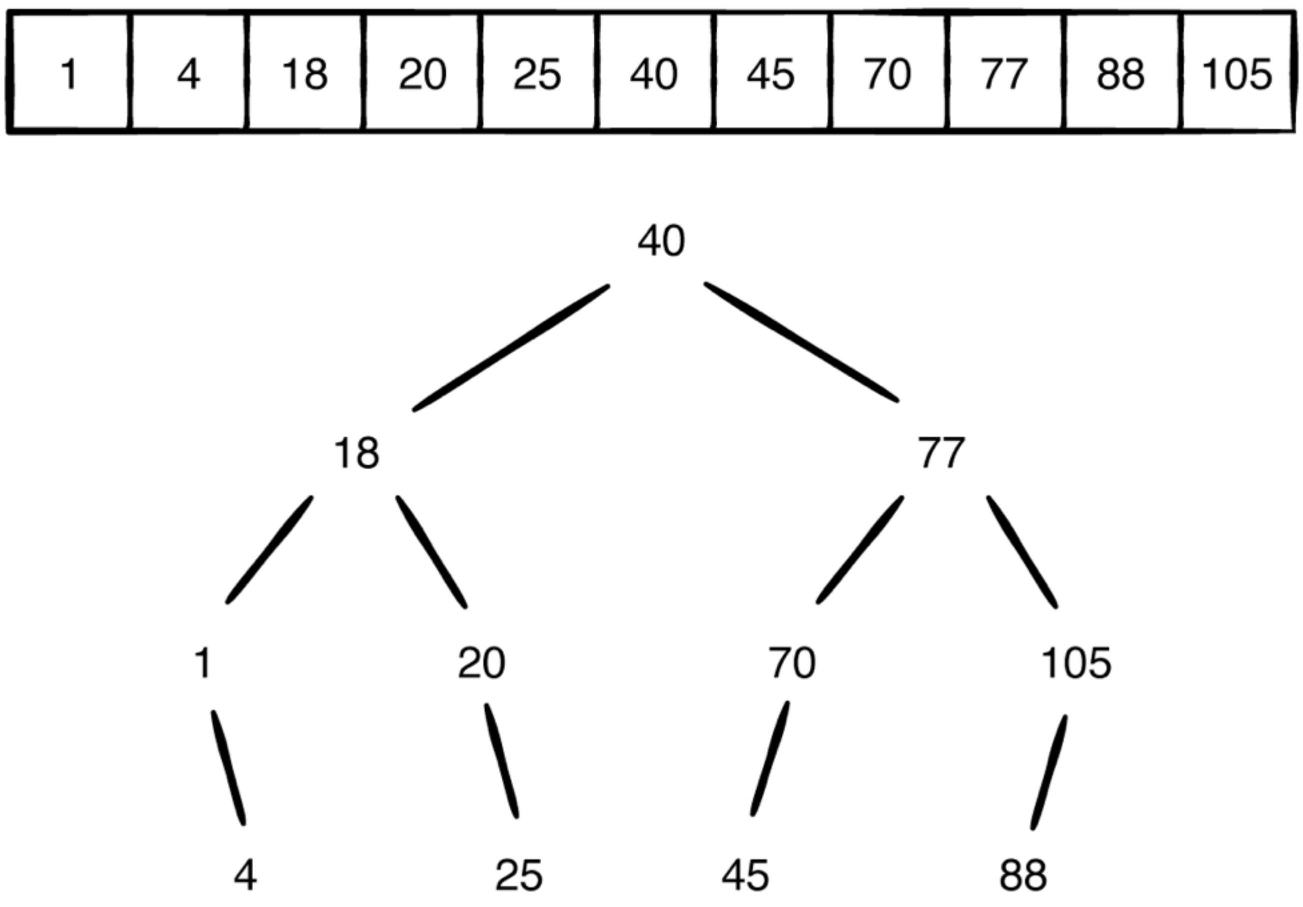
Binary search trees use this property to save you from performing unnecessary checking. As a result, lookup, insert, and removal have an average time complexity of *O*(log *n*), which is considerably faster than linear data structures such as arrays and linked lists.

In this chapter, you'll learn about the benefits of the BST relative to an array and implement the data structure from scratch.

Case study: array vs. BST

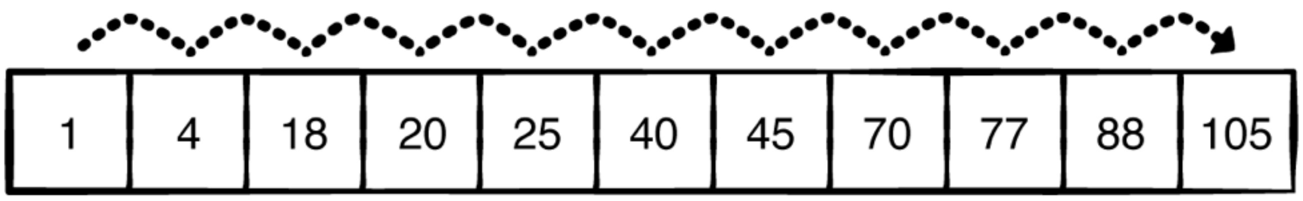
To illustrate the power of using a BST, you'll look at some common operations and compare the performance of arrays against the binary search tree.

Consider the following two collections:



Lookup

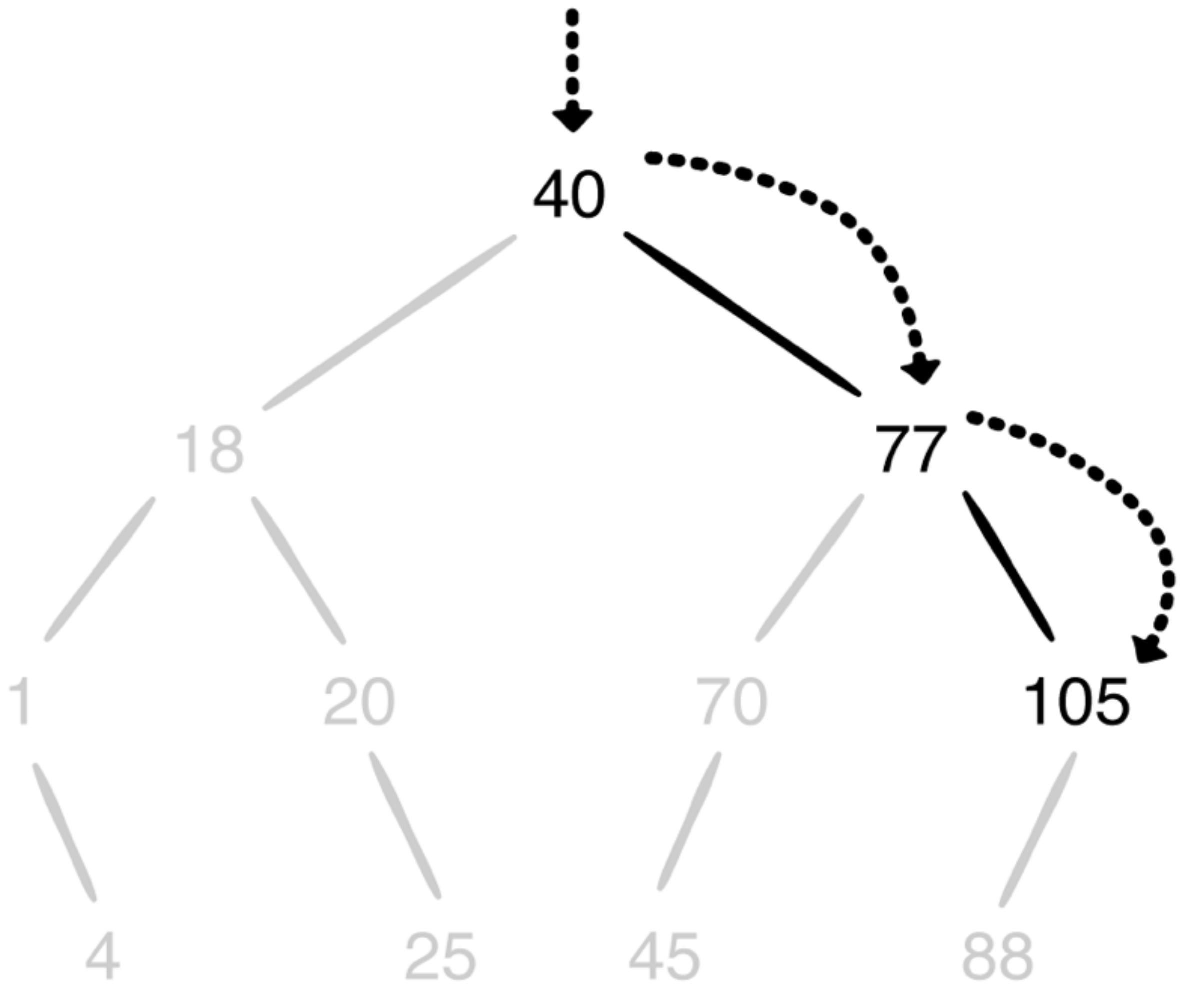
There's only one way to do element lookups for an unsorted array. You need to check every element in the array from the start.



Searching for 105

That's why `contains` is an *O*(*n*) operation.

This is not the case for binary search trees.



Searching for 105

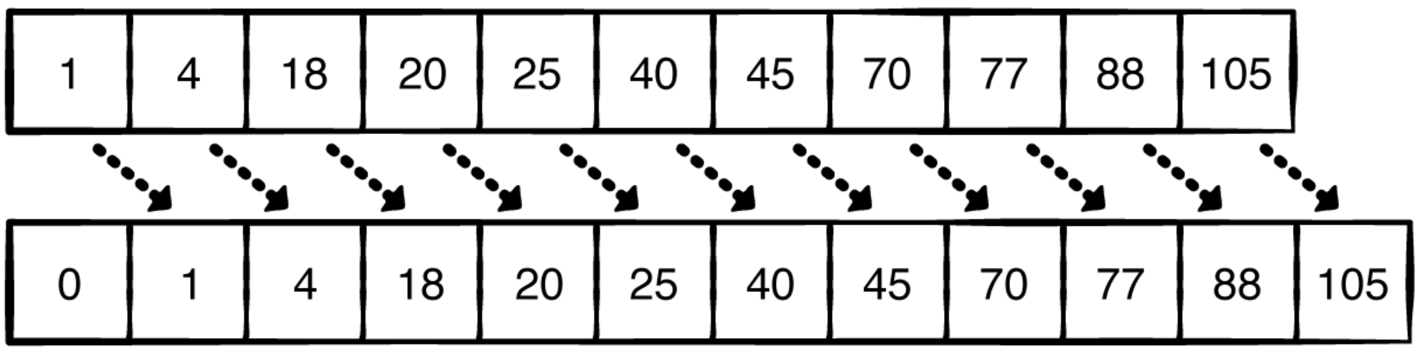
Every time the search algorithm visits a node in the BST, it can safely make these two assumptions:

- If the search value is **less than** the current value, it must be in the **left** subtree.
- If the search value is **greater than** the current value, it must be in the **right** subtree.

By leveraging the rules of the BST, you can avoid unnecessary checks and cut the search space in half every time you make a decision. That's why element lookup in a BST is an *O*(log *n*) operation.

Insertion

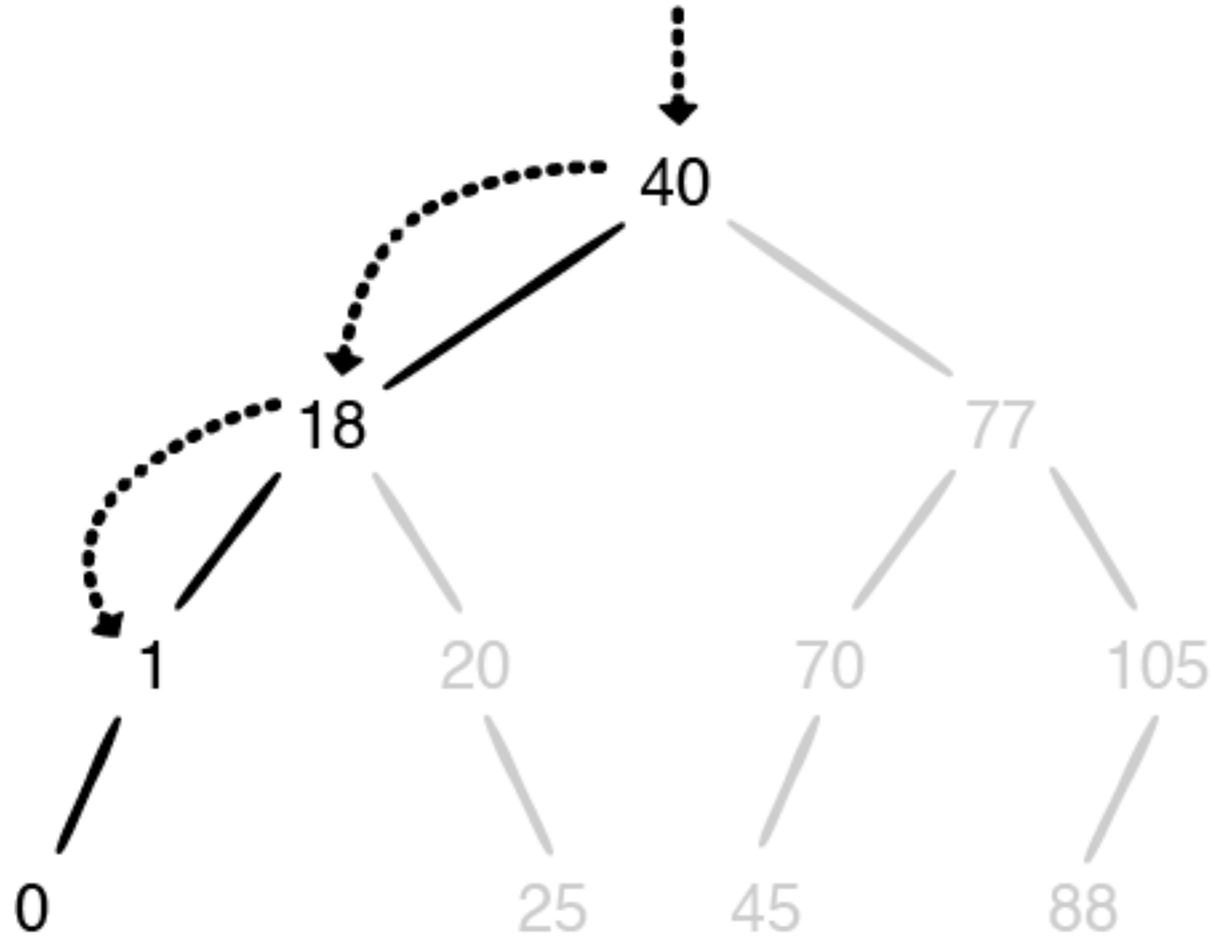
The performance benefits for the insertion operation follow a similar story. Assume you want to insert **0** into a collection.



Inserting 0 in sorted order

Inserting values into an array is like butting into an existing line: Everyone in the line behind your chosen spot needs to make space for you by shuffling back. In the above example, zero is inserted at the front of the array, causing all of the other elements to shift backward by one position. Inserting into an array has a time complexity of *O*(*n*).

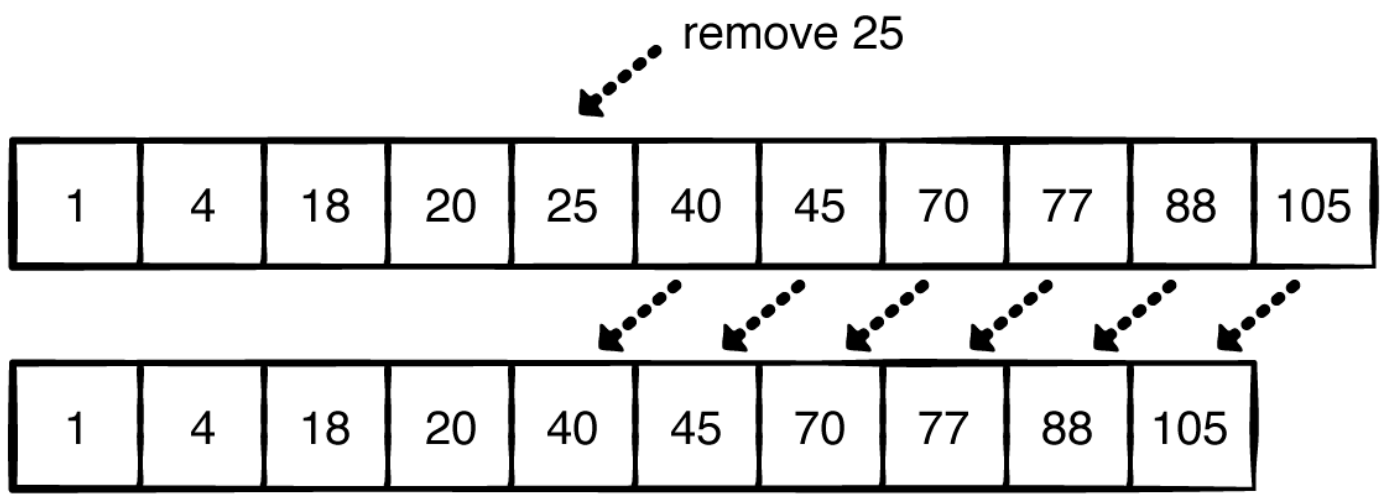
Insertion into a binary search tree is much more comforting.



By leveraging the rules for the BST, you only needed to make three hops to find the location for the insertion, and you didn't have to shuffle all of the elements around. Inserting elements in a BST is, again, an *O*(log *n*) operation.

Removal

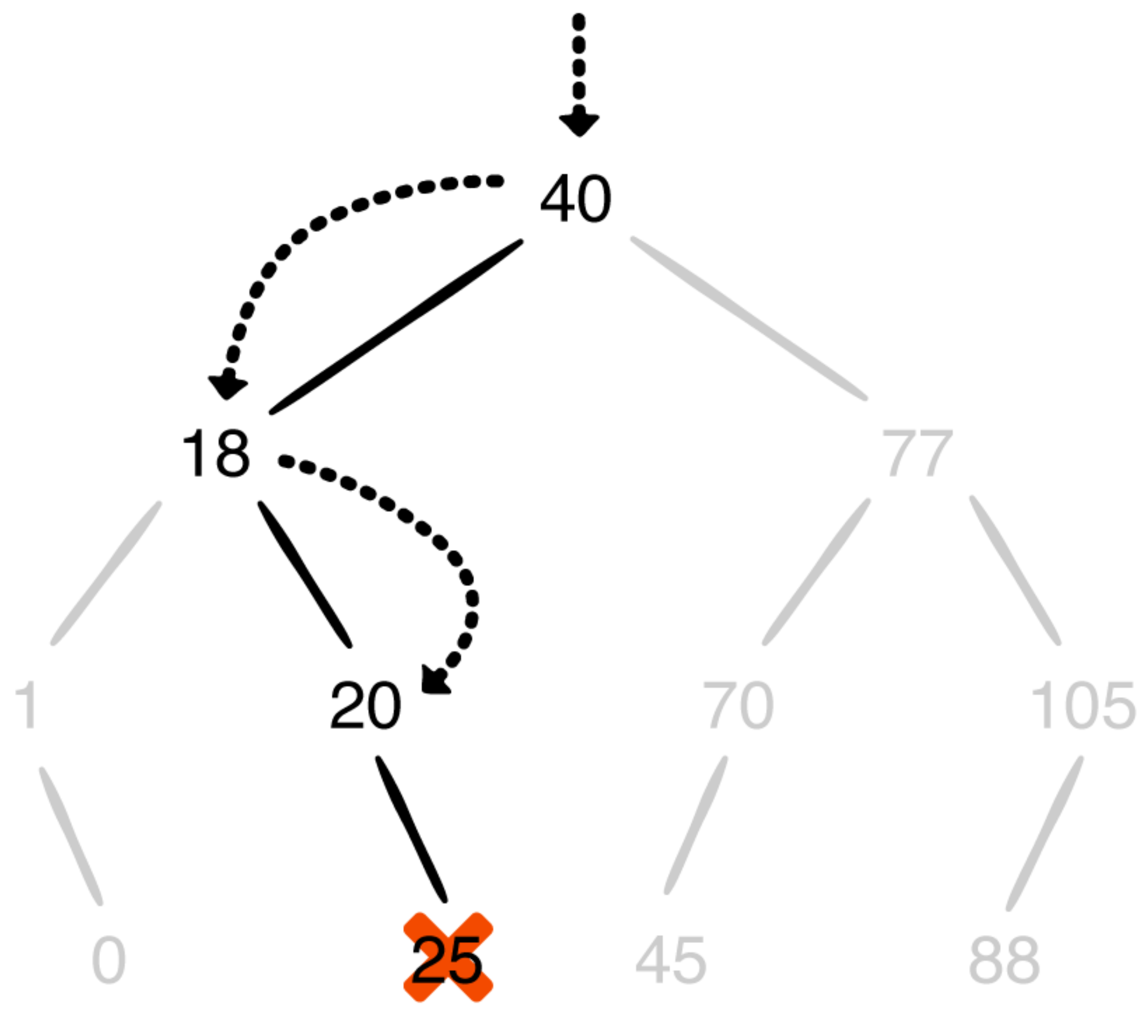
Similar to insertion, removing an element from an array also triggers a shuffling of elements.



Removing 25 from the array

This behavior also plays nicely with the line analogy. If you leave the middle of the line, everyone behind you needs to shuffle forward to take up the empty space.

Here's what removing a value from a BST looks like:



Nice and easy! There are complications to manage when the node you're removing has children, but you'll look into that later. Even with those complications, removing an element from a BST is still an *O*(log *n*) operation.

Binary search trees drastically reduce the number of steps for add, remove, and lookup operations. Now that you have a grasp of the benefits of using a binary search tree, you can move on to the actual implementation.

Implementation

Open the starter project for this chapter. In it, you'll find the `BinaryNode` class you created in the previous chapter. Create a new file named **BinarySearchTree.kt** and add the following to it:

```
class BinarySearchTree<T: Comparable<T>>() {

  var root: BinaryNode<T>? = null

  override fun toString() = root?.toString() ?: "empty tree"

}
```

By definition, binary search trees can only hold values that are `Comparable`.

Next, you'll look at the `insert` method.

Inserting elements

Following the rules of the BST, nodes of the left child must contain values less than the current node, whereas nodes of the right child must contain values greater than or equal to the current node. You'll implement `insert` while respecting these rules.

Add the following to `BinarySearchTree`:

```
fun insert(value: T) {
  root = insert(root, value)
}

private fun insert(
    node: BinaryNode<T>?,
    value: T
): BinaryNode<T> {
  // 1
  node ?: return BinaryNode(value)
  // 2
  if (value < node.value) {
    node.leftChild = insert(node.leftChild, value)
  } else {
    node.rightChild = insert(node.rightChild, value)
  }
  // 3
  return node
}
```

The first `insert` is exposed to users, while the second will be used as a private helper method:

1. This is a recursive method, so it requires a base case for terminating recursion. If the current node is `null`, you've found the insertion point and return the new `BinaryNode`.

2. This `if` statement controls which way the next `insert` call should traverse. If the new value is less than the current value, you call `insert` on the **left** child. If the new value is greater than or equal to the current value, you call `insert` on the **right** child.

3. Return the current node. This makes assignments of the form `node = insert(node, value)` possible as `insert` will either create `node` (if it was `null`) or return `node` (if it was not `null`).

Go back to `main()` and add the following at the bottom:

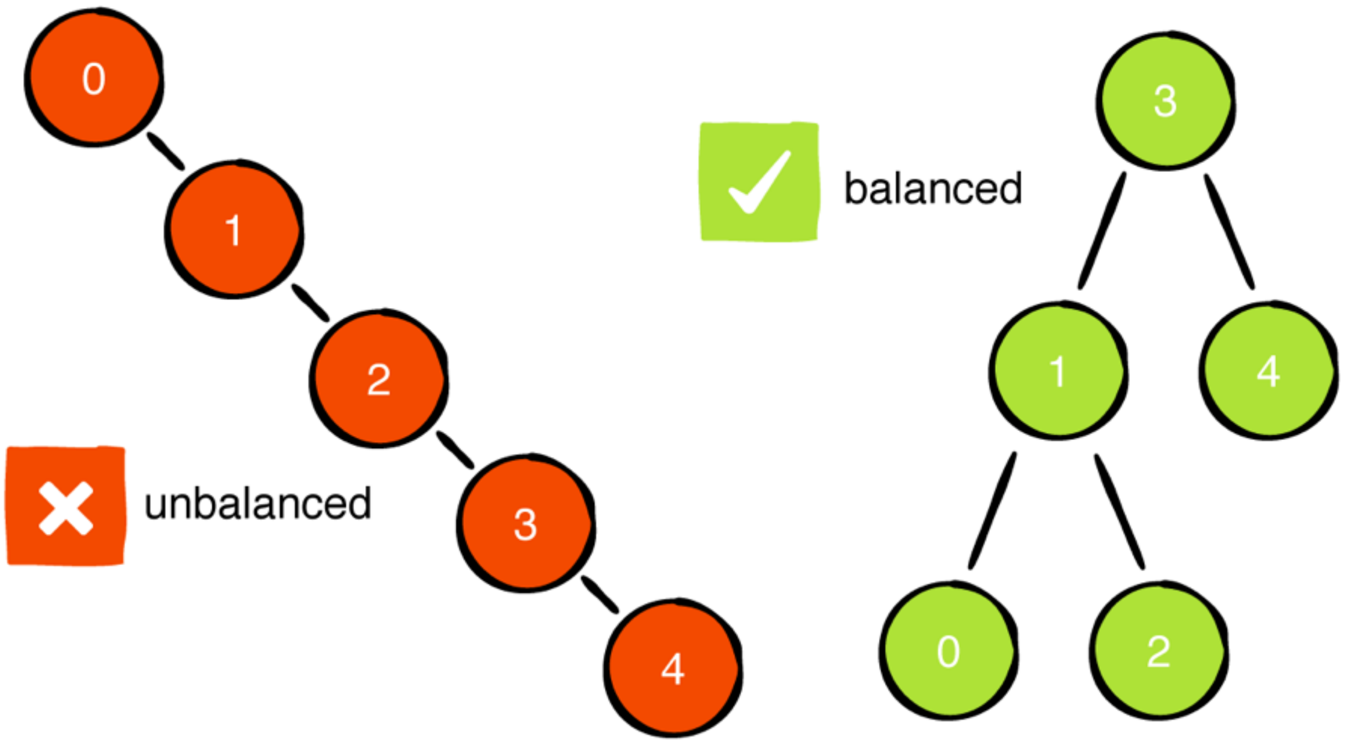
```
"building a BST" example {
  val bst = BinarySearchTree<Int>()
  (0..4).forEach {
    bst.insert(it)
  }
  println(bst)
}
```

You'll see the following output:

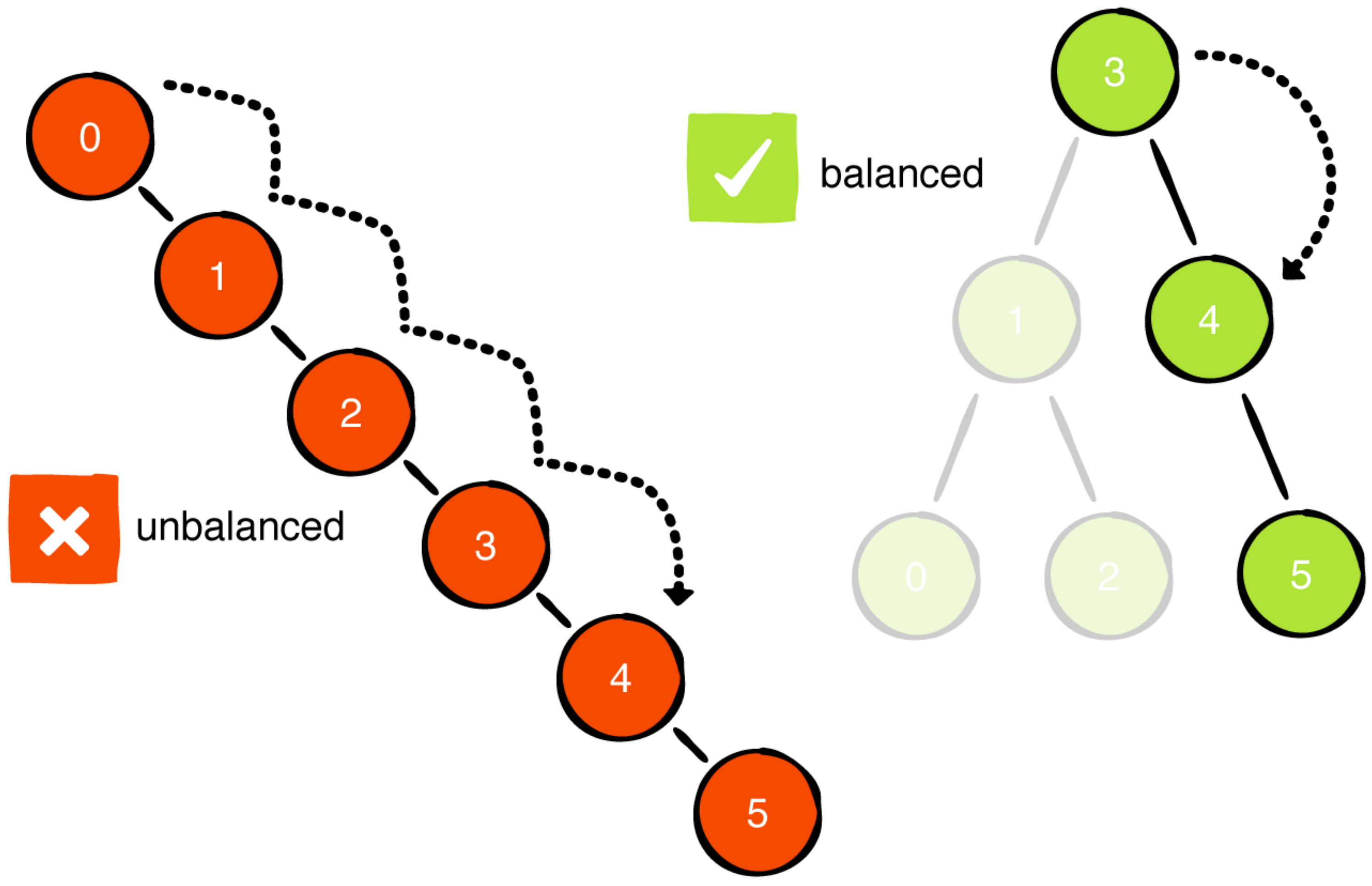
```
---Example of building a BST---
   ┌──4
  ┌──3
  │ └──null
```

```
   ┌──2
   │ └──null
 ┌──1
 │ └──null
0
└──null
```

That tree looks a bit *unbalanced*, but it does follow the rules. However, this tree layout has undesirable consequences. When working with trees, you always want to achieve a balanced format.



An unbalanced tree affects performance. If you insert **5** into the unbalanced tree you created, it becomes an *O*(*n*) operation.



You can create structures known as *self-balancing trees* that use clever techniques to maintain a balanced structure, but we'll save those details for the next chapter. For now, you'll simply build a sample tree with a bit of care to keep it from becoming unbalanced.

Add the following variable at the start of `main()`:

```
val exampleTree = BinarySearchTree<Int>().apply {
  insert(3)
  insert(1)
  insert(4)
  insert(0)
  insert(2)
  insert(5)
}
```

Then, update the previous example with this code:

```
"building a BST" example {
  println(exampleTree)
}
```

You'll see the following in the console:

```
---Example of building a BST---
 ┌──5
┌──4
│ └──null
3
│ ┌──2
└──1
 └──0
```

Much nicer!

Finding elements

Finding an element in a BST requires you to traverse through its nodes. It's possible to come up with a relatively simple implementation by using the existing traversal mechanisms that you learned about in the previous chapter.

Add the following to the bottom of `BinarySearchTree`:

```
fun contains(value: T): Boolean {
  root ?: return false

  var found = false
  root?.traverseInOrder {
    if (value == it) {
      found = true
    }
  }

  return found
}
```

Next, go back to `main()` to test this out:

```
"finding a node" example {
  if (exampleTree.contains(5)) {
    println("Found 5!")
  } else {
    println("Couldn't find 5")
  }
}
```

You'll see the following in the console:

```
---Example of finding a node---
Found 5!
```

In-order traversal has a time complexity of *O*(*n*), thus this implementation of `contains` has the same time complexity as an exhaustive search through an unsorted array. However, you can do better!

Optimizing contains

You can rely on the rules of the BST to avoid needless comparisons. Inside **BinarySearchTree.kt**, update `contains` to the following:

```
fun contains(value: T): Boolean {
  // 1
  var current = root

  // 2
  while (current != null) {
    // 3
    if (current.value == value) {
      return true
    }

    // 4
    current = if (value < current.value) {
      current.leftChild
    } else {
      current.rightChild
    }
  }

  return false
}
```

Here's how it works:

1. Start by setting `current` to the `root` node.
2. While `current` is not `null`, check the current node's value.
3. If the `value` is equal to what you're trying to find, return `true`.
4. Otherwise, decide whether you're going to check the left or right child.

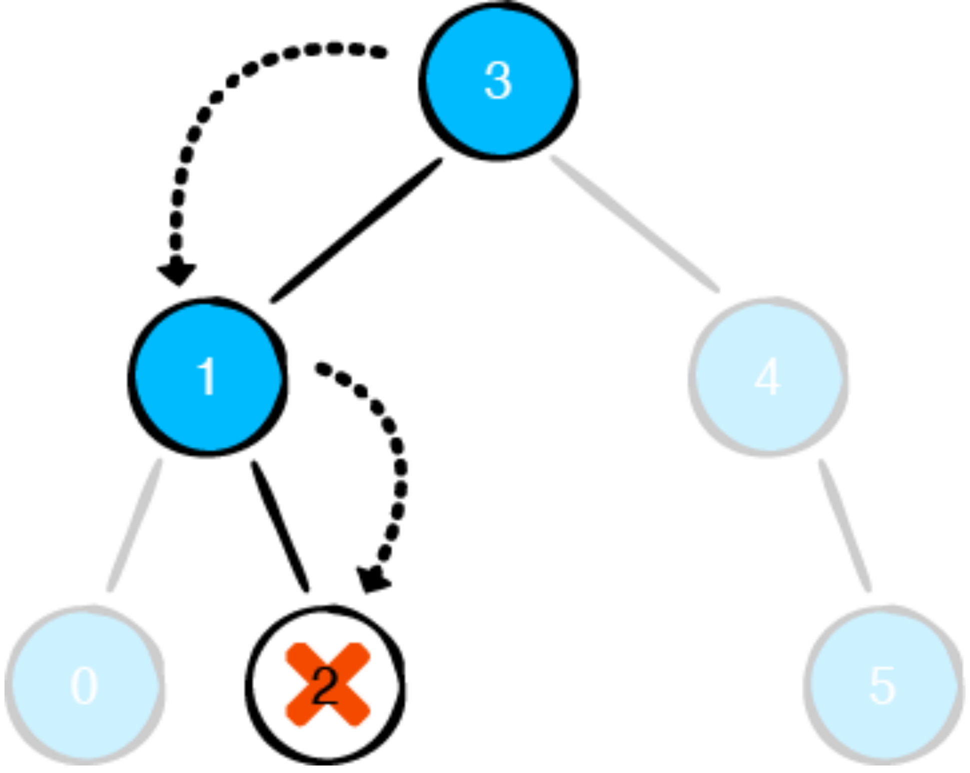
This implementation of `contains` is an *O*(log *n*) operation in a balanced binary search tree.

Removing elements

Removing elements is a little more tricky, as there are a few different scenarios you need to handle.

Case 1: Leaf node

Removing a leaf node is straightforward; simply detach the leaf node.

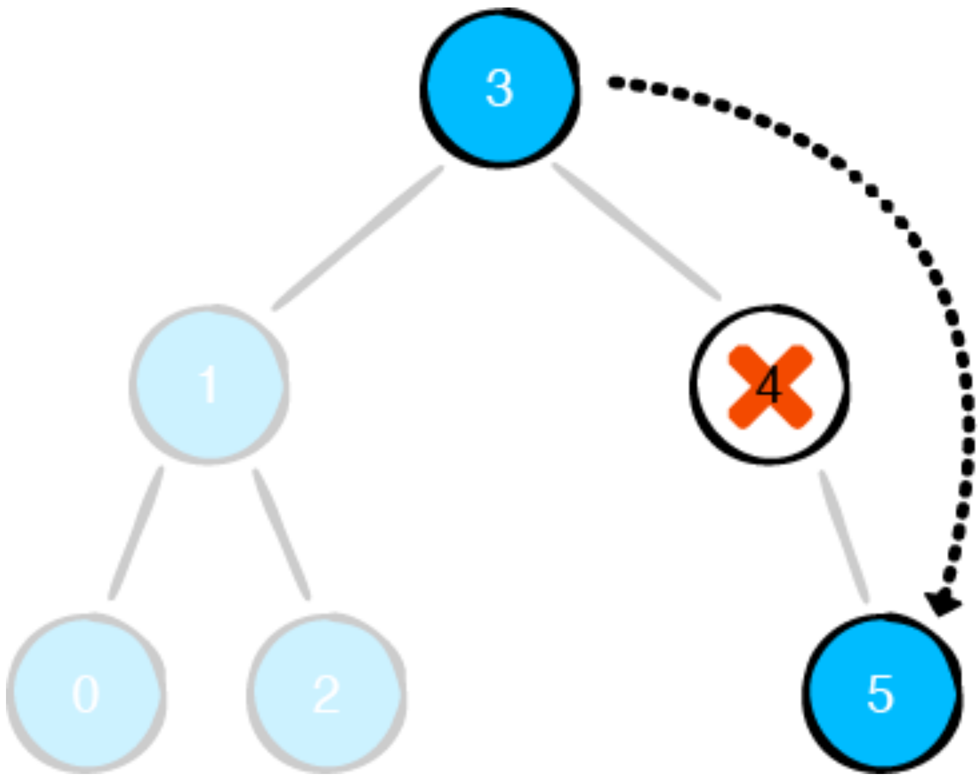


Removing 2

For non-leaf nodes, however, there are extra steps you must take.

Case 2: Nodes with one child

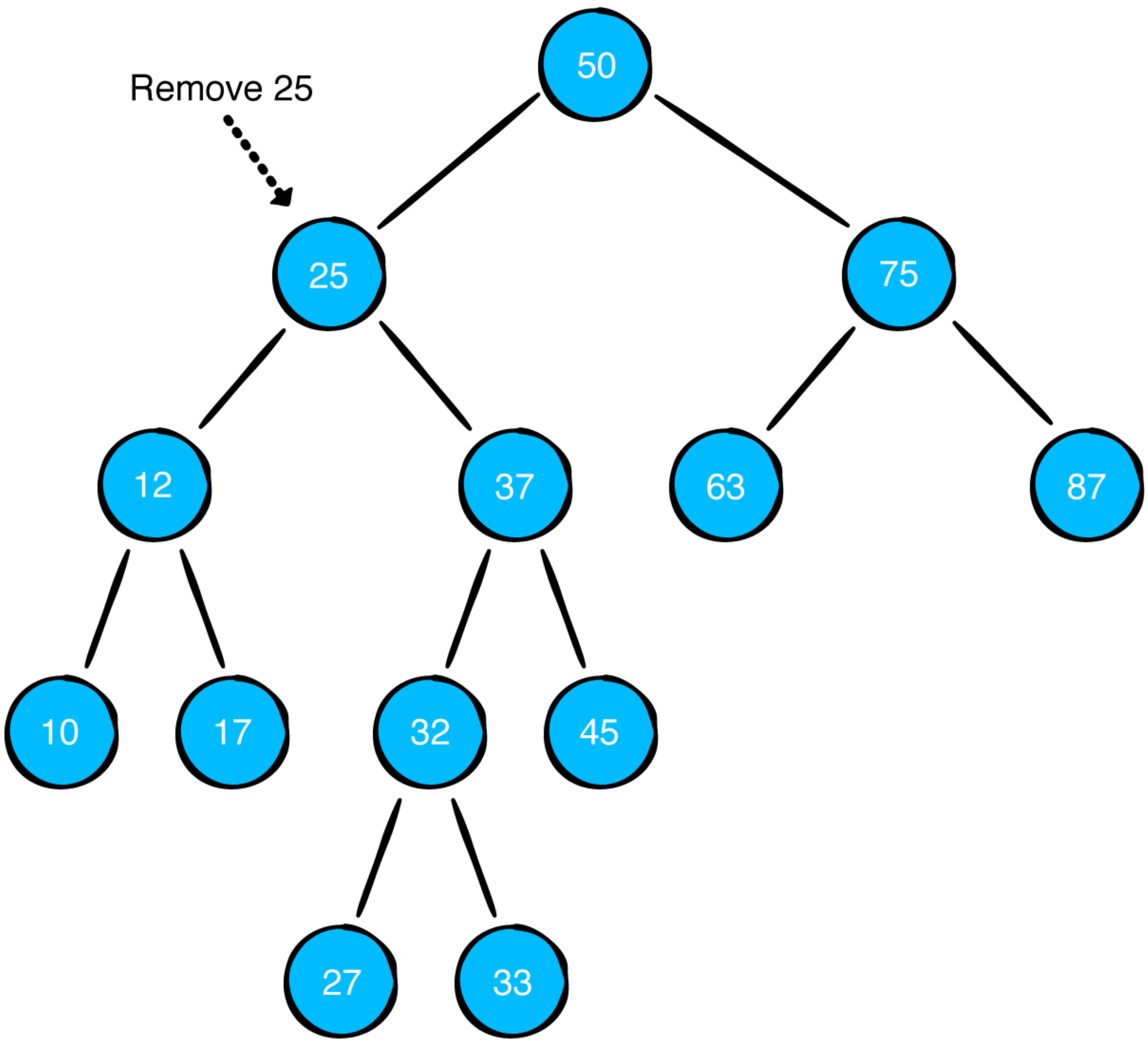
When removing nodes with one child, you need to reconnect that one child with the rest of the tree.



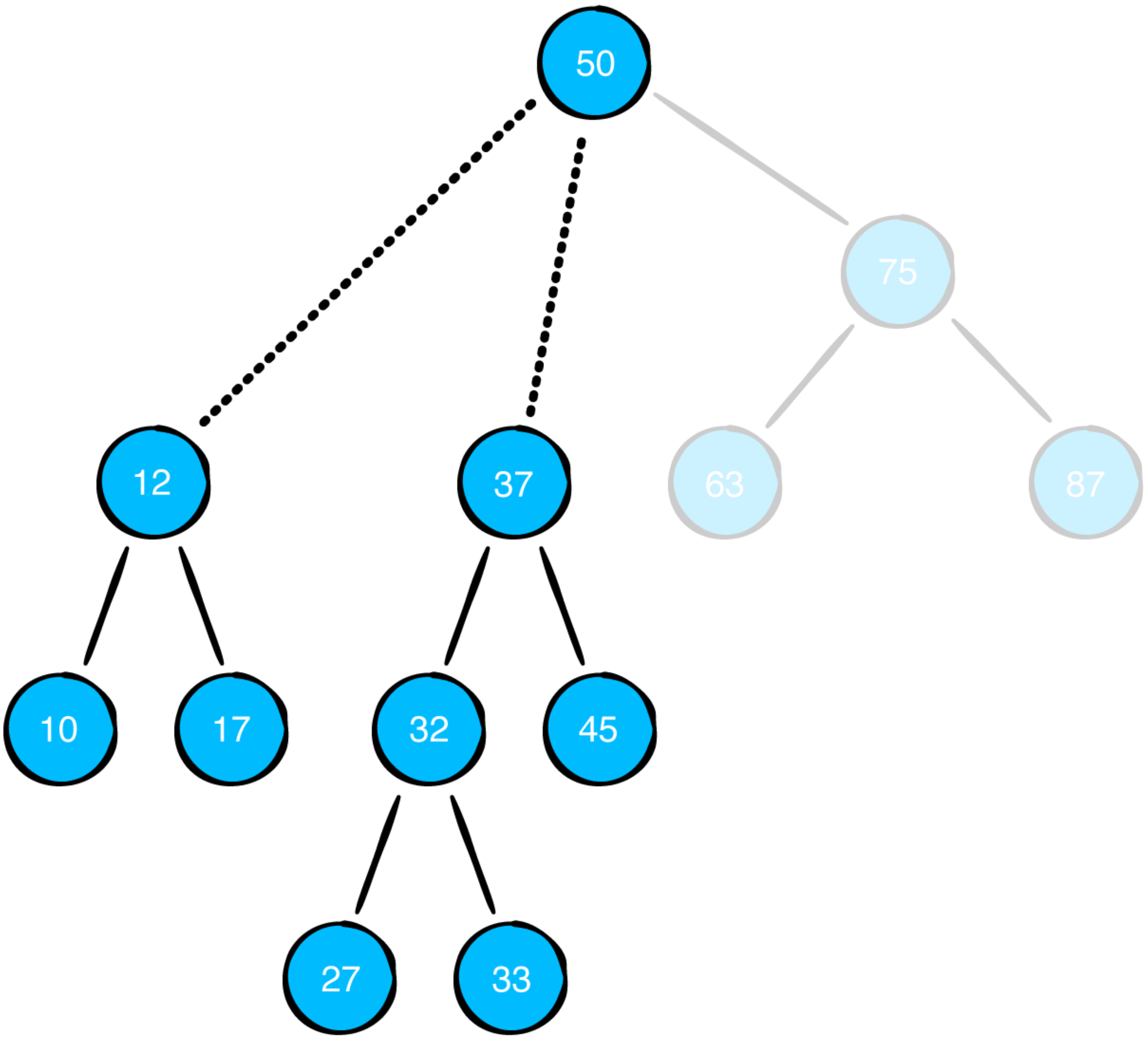
Removing 4, which has 1 child

Case 3: Nodes with two children

Nodes with two children are a bit more complicated, so a more complex example tree will serve better to illustrate how to handle this situation. Assume that you have the following tree and that you want to remove the value **25**:

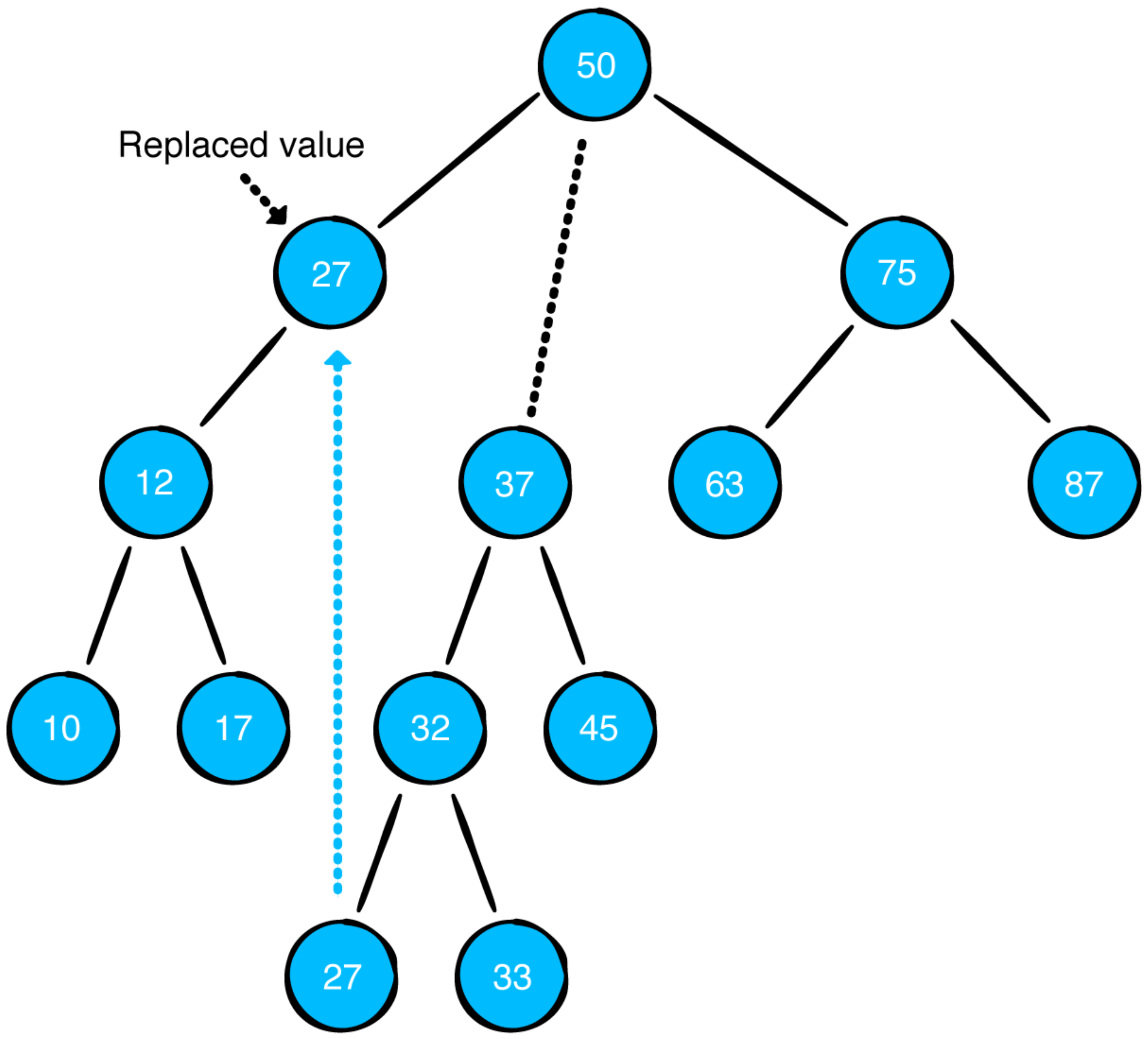


Simply deleting the node presents a dilemma.



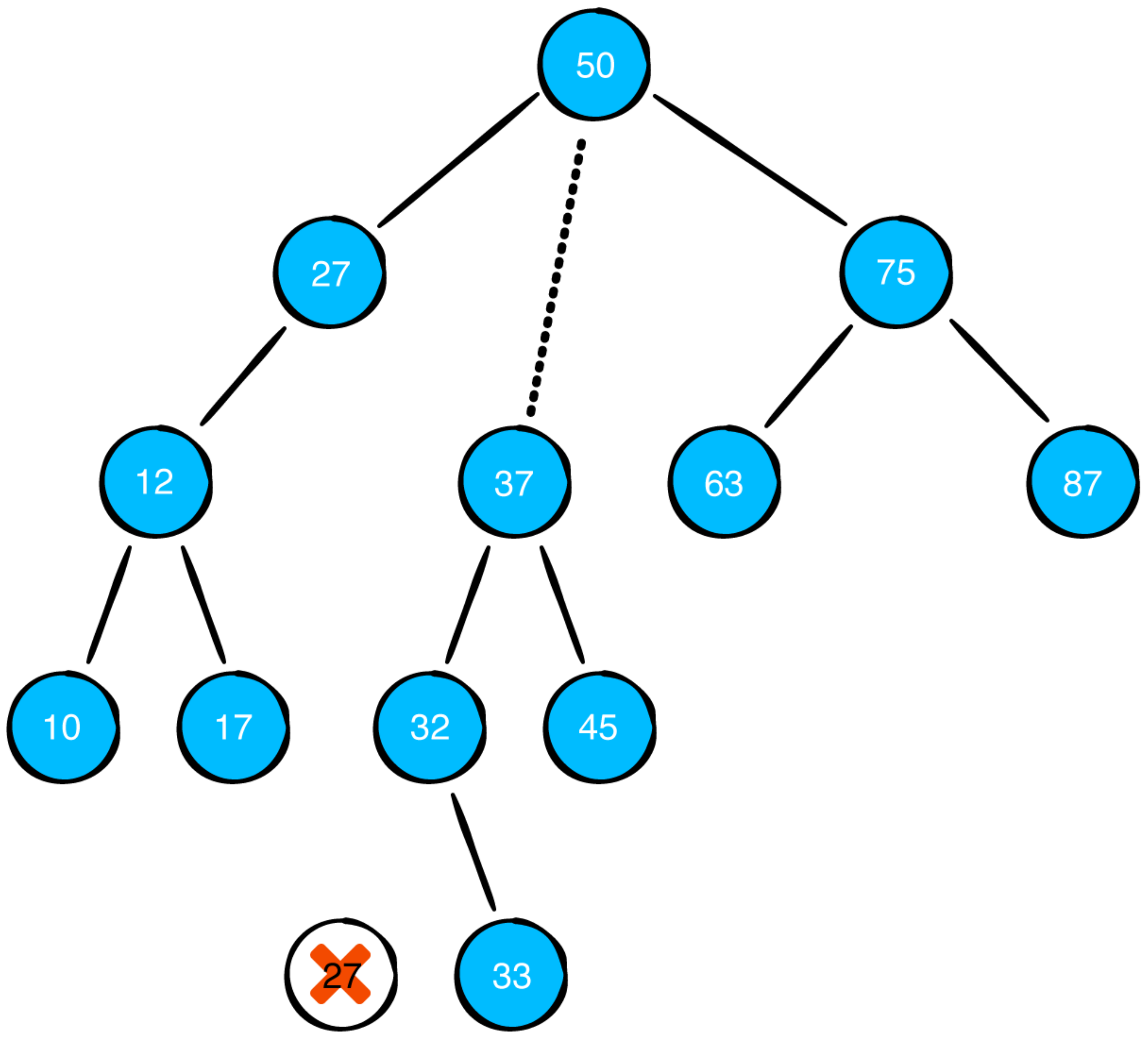
You have two child nodes (**12** and **37**) to reconnect, but the parent node only has space for one child. To solve this problem, you'll implement a clever workaround by performing a swap.

When removing a node with two children, replace the node you removed with the smallest node in its *right* subtree. Based on the rules of the BST, this is the leftmost node of the right subtree:



It’s important to note that this produces a valid binary search tree. Because the new node was the smallest node in the right subtree, all of the nodes in the right subtree will still be greater than or equal to the new node. And because the new node came from the right subtree, all of the nodes in the left subtree will be less than the new node.

After performing the swap, you can simply remove the value you copied, which is just a leaf node.



This will take care of removing nodes with two children.

Implementation

Add the following code to **BinaryNode.kt**:

```
val min: BinaryNode<T>?
  get() = leftChild?.min ?: this
```

This recursive `min` property in `BinaryNode` will help you find the minimum node in a subtree.

Open **BinarySearchTree.kt** to implement `remove`. Add the following code at the bottom of the class:

```
fun remove(value: T) {
  root = remove(root, value)
}

private fun remove(
    node: BinaryNode<T>?,
    value: T
): BinaryNode<T>? {
  node ?: return null

  when {
    value == node.value -> {
      // more to come
    }
    value < node.value -> node.leftChild = remove(node.leftChild, value)
    else -> node.rightChild = remove(node.rightChild, value)
  }
  return node
}
```

This should look familiar to you. You're using the same recursive setup with a private helper method as you did for `insert`. The different removal cases are handled in the `value == node.value` branch:

```
// 1
if (node.leftChild == null && node.rightChild == null) {
  return null
}
// 2
if (node.leftChild == null) {
  return node.rightChild
}
// 3
if (node.rightChild == null) {
  return node.leftChild
}
// 4
node.rightChild?.min?.value?.let {
  node.value = it
}
```

```
node.rightChild = remove(node.rightChild, node.value)
```

Here's what's happening:

1. In the case in which the node is a leaf node, you simply return `null`, thereby removing the current node.
2. If the node has no left child, you return `node.rightChild` to reconnect the right subtree.
3. If the node has no right child, you return `node.leftChild` to reconnect the left subtree.
4. This is the case in which the node to be removed has both a left and right child. You replace the node's value with the smallest value from the right subtree. You then call `remove` on the right child to remove this swapped value.

Go back to `main()` and test `remove` by writing the following:

```
"removing a node" example {
  println("Tree before removal:")
  println(exampleTree)
  exampleTree.remove(3)
  println("Tree after removing root:")
  println(exampleTree)
}
```

You'll see the following output in the console:

```
---Example of removing a node---
Tree before removal:
 ┌──5
┌──4
│ └──null
3
│ ┌──2
└──1
```

```
 └──0
```

```
Tree after removing root:
┌──5
4
│ ┌──2
└──1
 └──0
```

Challenges

Think you have searching of binary trees down cold? Try out these three challenges to lock down the concepts.

Challenge 1: Is it a BST?

Create a function that checks if a binary tree is a binary search tree.

Solution 1

A binary search tree is a tree where every left child is less than or equal to its parent, and every right child is greater than its parent. An algorithm that verifies whether a tree is a binary search tree involves going through all the nodes and checking for this property.

Write the following in **BinaryNode.kt** in the `BinaryNode` class:

```
val isBinarySearchTree: Boolean
  get() = isBST(this, min = null, max = null)

// 1
private fun isBST(tree: BinaryNode<T>?, min: T?, max: T?): Boolean {
  // 2
  tree ?: return true

  // 3
  if (min != null && tree.value <= min) {
    return false
  } else if (max != null && tree.value > max) {
    return false
  }

  // 4
  return isBST(tree.leftChild, min, tree.value) && isBST(tree.rightChild, t
}
```

Here's how it works:

1. `isBST` is responsible for recursively traversing through the tree and checking for the BST property. It needs to keep track of progress via a reference to a `BinaryNode` and also keep track of the `min` and `max` values to verify the BST property.
2. This is the base case. If `tree` is `null`, then there are no nodes to inspect. A `null` node is a binary search tree, so you'll return `true` in that case.
3. This is essentially a bounds check. If the current value exceeds the bounds of the `min` and `max` values, the current node does not respect the binary search tree rules.
4. This line contains the recursive calls. When traversing through the left children, the current value is passed in as the `max` value. This is because nodes in the left side cannot be greater than the parent. Vice versa, when traversing to the right, the `min` value is updated to the current value. Nodes in the right side must be greater than the parent. If any of the recursive calls evaluate `false`, the `false` value will propagate to the top.

The time complexity of this solution is ***O*(*n*)** since you need to traverse through the entire tree once. There is also a ***O*(*n*)** space cost since you're making *n* recursive calls.

Challenge 2: Equality between BSTs

Override `equals()` to check whether two binary search trees are equal.

Solution 2

Overriding `equals()` is relatively straightforward. For two binary trees to be equal, both trees must have the same elements in the same order. This is how the solution looks:

```
// 1
override fun equals(other: Any?): Boolean {
  // 2
  return if (other != null && other is BinaryNode<*>) {
    this.value == other.value &&
        this.leftChild == other.leftChild &&
        this.rightChild == other.rightChild
  } else {
    false
  }
}
```

Here's an explanation of the code:

1. `equals` recursively checks two nodes and their descendants for equality.
2. Here, you check the value of the left and right nodes for equality. You also recursively check the left children and the right children for equality.

Inside **BinaryNode.kt**, update the `BinaryNode` class declaration to make `T` type comparable:

```
class BinaryNode<T: Comparable<T>>(var value: T)
```

The time complexity of this function is ***O*(*n*)**. The space complexity of this function is ***O*(*n*)**.

Challenge 3: BSTs with the same elements?

Create a method that checks if the current tree contains all of the elements of another tree.

Solution 3

Your goal is to create a method that checks if the current tree contains all of the elements of another tree. In other words, the values in the current tree must be a superset of the values in the other tree. The solution looks like this:

```
fun contains(subtree: BinarySearchTree<T>): Boolean {
  // 1
  val set = mutableSetOf<T>()
  root?.traverseInOrder {
    set.add(it)
  }

  // 2
  var isEqual = true
  subtree.root?.traverseInOrder {
    isEqual = isEqual && set.contains(it)
  }
  return isEqual
}
```

Here's how it works:

1. Inside `contains`, you begin by inserting all of the elements of the current tree into a set.
2. `isEqual` will store the result. For every element in the subtree, you check if the value is contained in the set. If at any point `set.contains(it)` evaluates to `false`, you'll make sure `isEqual` stays `false` even if subsequent elements evaluate to `true` by assigning `isEqual && list.contains(it)` to itself.

The time complexity for this algorithm is ***O*(*n*)**. The space complexity for this algorithm is ***O*(*n*)**.

Key points

- The binary search tree is a powerful data structure for holding sorted data.
- Average performance for `insert`, `remove` and `contains` in a BST is *O*(*log n*).
- Performance will degrade to *O*(*n*) as the tree becomes unbalanced. This is undesirable, so you'll learn about a self-balancing binary search tree known as the AVL tree in the next chapter.