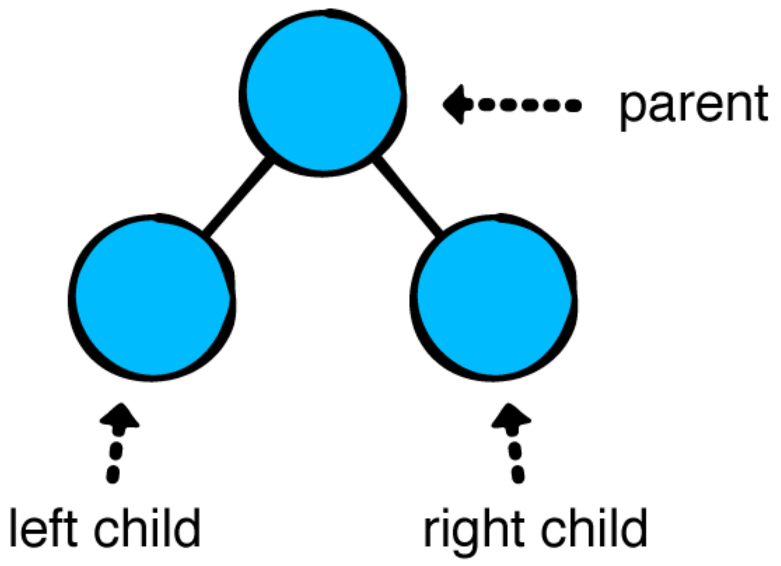
7 Binary Trees Written by Irina Galata

In the previous chapter, you looked at a basic tree in which each node can have many children. A **binary tree** is a tree in which each node has at most **two** children, often referred to as the **left** and **right** children:



Binary Tree

Binary trees serve as the basis for many tree structures and algorithms. In this chapter, you'll build a binary tree and learn about the three most important tree traversal algorithms.

Implementation

Open the starter project for this chapter. Create a new file and name it **BinaryNode.kt**. You also define the `Visitor<T>` typealias. Add the following inside this file:

```
typealias Visitor<T> = (T) -> Unit

class BinaryNode<T: Any>(var value: T) {
  var leftChild: BinaryNode<T>? = null
  var rightChild: BinaryNode<T>? = null
}
```

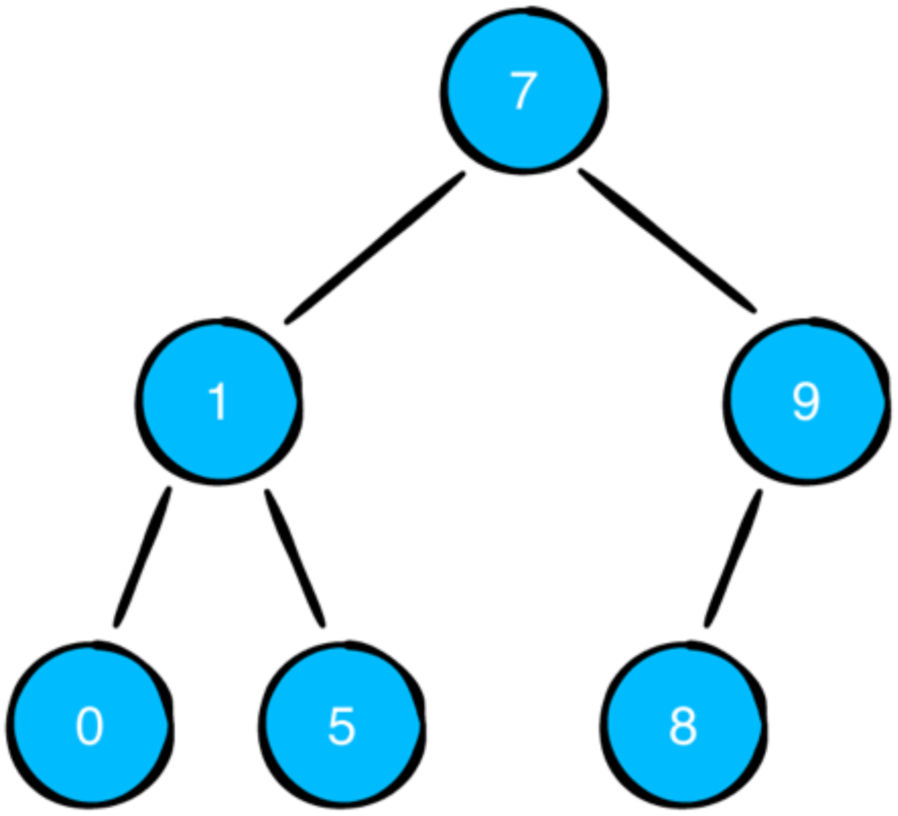
In `main()` in the **Main.kt** file, add the following:

```
fun main() {
  val zero = BinaryNode(0)
  val one = BinaryNode(1)
  val five = BinaryNode(5)
  val seven = BinaryNode(7)
  val eight = BinaryNode(8)
  val nine = BinaryNode(9)

  seven.leftChild = one
  one.leftChild = zero
  one.rightChild = five
  seven.rightChild = nine
  nine.leftChild = eight

  val tree = seven
}
```

This defines the following tree by executing the closure:



Example Binary Tree

Building a diagram

Building a mental model of a data structure can be quite helpful in learning how it works. To that end, you'll implement a reusable algorithm that helps visualize a binary tree in the console.

> **Note**: This algorithm is based on an implementation by Károly Lőrentey in his book *Optimizing Collections*, available from https://www.objc.io/books/optimizing-collections/.

Add the following to the bottom of **BinaryNode.kt**:

```
override fun toString() = diagram(this)

private fun diagram(node: BinaryNode<T>?,
                    top: String = "",
                    root: String = "",
                    bottom: String = ""): String {
  return node?.let {
    if (node.leftChild == null && node.rightChild == null) {
      "$root${node.value}\n"
    } else {
      diagram(node.rightChild, "$top ", "$top┌──", "$top│ ") +
          root + "${node.value}\n" + diagram(node.leftChild, "$bottom│ ", "
    }
  } ?: "${root}null\n"
}
```

This method recursively creates a string representing the binary tree.

To try it out, open **main.kt** and add the following:

```
println(tree)
```

You'll see the following console output:

```
  ┌──null
┌──9
│ └──8
7
│ ┌──5
└──1
  └──0
```

You'll use this diagram for other binary trees in this book.

Traversal algorithms

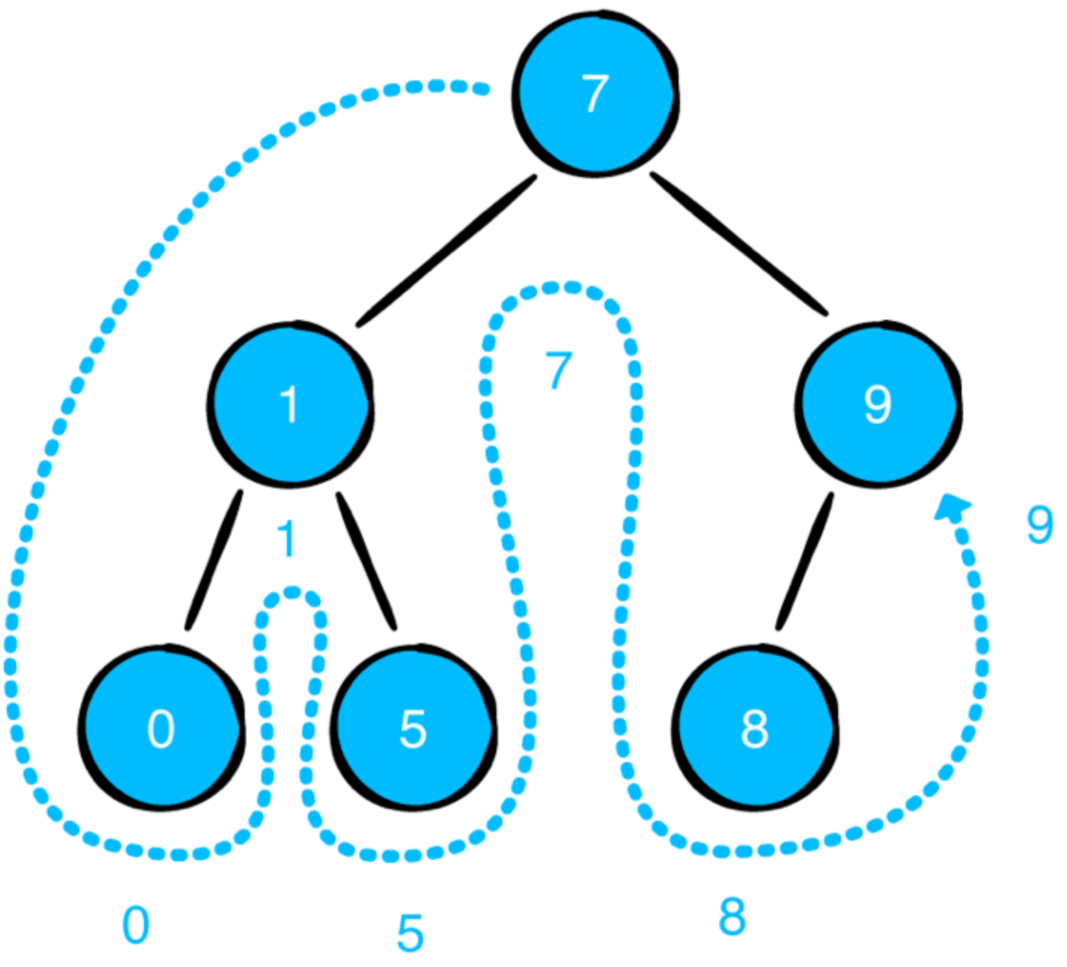
Previously, you looked at a level-order traversal of a tree. With a few tweaks, you can make this algorithm work for binary trees as well. However, instead of re-implementing level-order traversal, you'll look at three traversal algorithms for binary trees: **in-order**, **pre-order** and **post-order** traversals.

In-order traversal

In-order traversal visits the nodes of a binary tree in the following order, starting from the root node:

- If the current node has a left child, recursively visit this child first.
- Then visit the node itself.
- If the current node has a right child, recursively visit this child.

Here's what an in-order traversal looks like for your example tree:



0, 1, 5, 7, 8, 9

You may have noticed that this prints the example tree in ascending order. If the tree nodes are structured in a certain way, in-order traversal visits them in ascending order. You'll learn more about binary search trees in the next chapter.

Open **BinaryNode.kt** and add the following code to the bottom of the file:

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

Following the rules laid out above, you first traverse to the left-most node before visiting the value. You then traverse to the right-most node.

To test this, go to `main()`, and add the following at the bottom:

```
tree.traverseInOrder { println(it) }
```

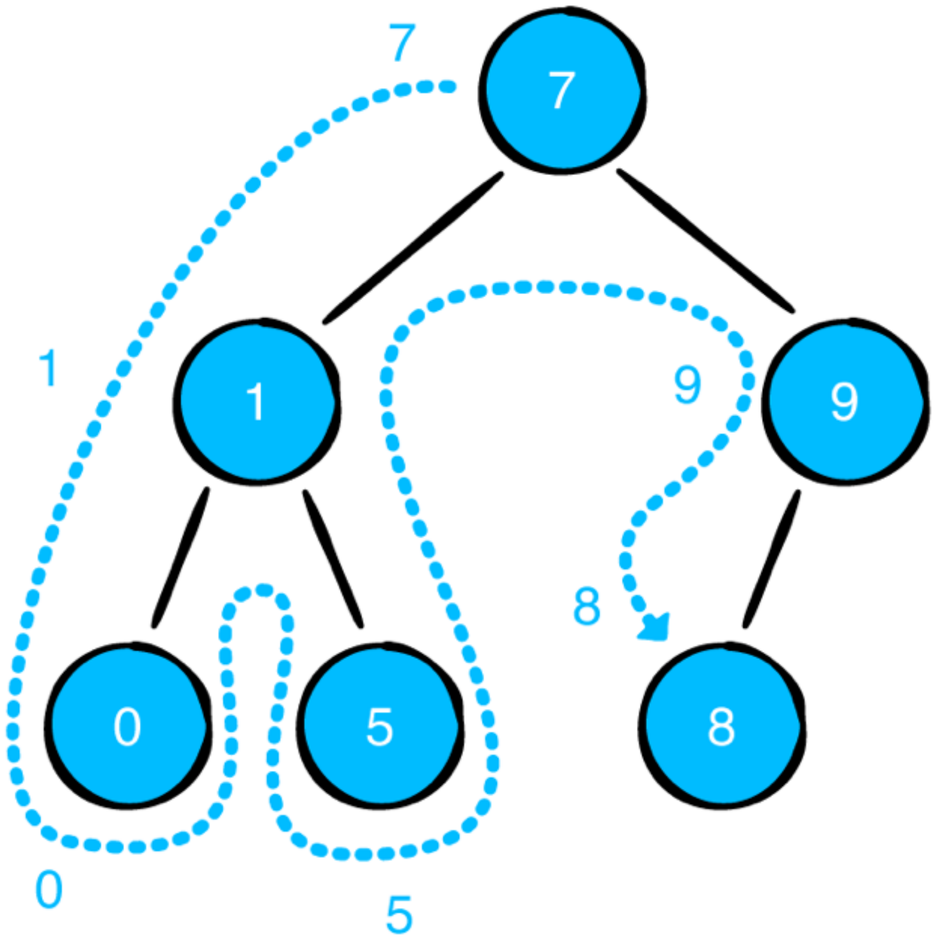
You should see the following in the console:

```
0
1
5
7
8
9
```

Pre-order traversal

Pre-order traversal visits the nodes of a binary tree in the following order:

- Visits the current node first.
- Recursively visits the left and right child.



Pre-order traversal

Write the following immediately below the in-order traversal method:

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

Test it out with the following code in the `main` method:

```
tree.traversePreOrder { println(it) }
```

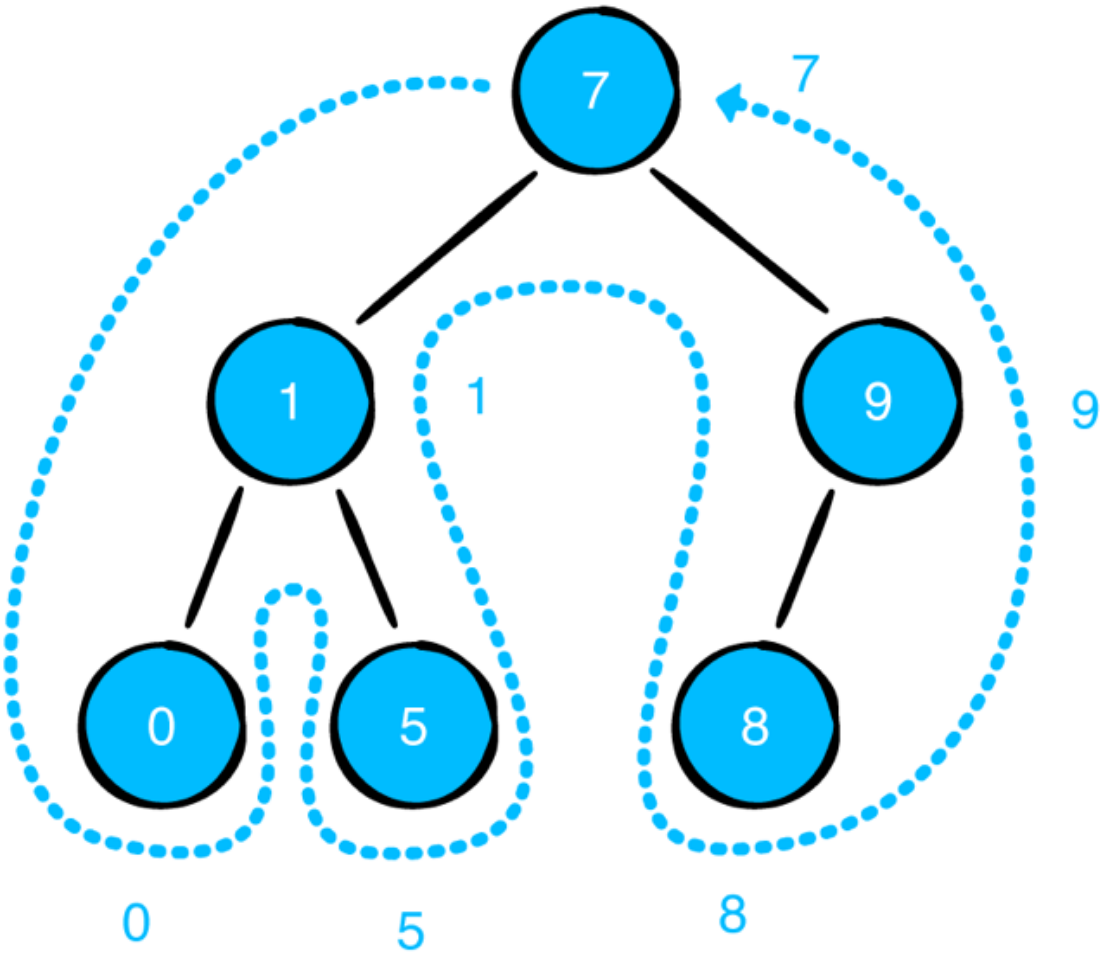
You'll see the following output in the console:

```
7
1
0
5
9
8
```

Post-order traversal

Post-order traversal always visits the nodes of a binary tree in the following order:

- Recursively visits the left and right child.
- Only visits the current node after the left and right child have been visited recursively.



Post-order traversal

In other words, given any node, you'll visit its children before visiting itself. An interesting consequence of this is that the root node is always visited last.

Inside **BinaryNode.kt**, add the following below `traversePreOrder`:

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

Navigate back to `main()` and add the following to try it out:

```
tree.traversePostOrder { println(it) }
```

You'll see the following in the console:

```
0
5
1
8
9
7
```

Each one of these traversal algorithms has both a time and space complexity of $O(n)$.

While this version of the binary tree isn't too exciting, you saw that you can use in-order traversal to visit the nodes in ascending order. Binary trees can enforce this behavior by adhering to some rules during insertion.

In the next chapter, you'll look at a binary tree with stricter semantics: the **binary search tree**.

Challenges

Binary trees are a surprisingly popular topic in algorithm interviews. Questions on the binary tree not only require a good foundation of how traversals work, but can also test your understanding of recursive backtracking. The challenges presented here offer an opportunity to put into practice what you've learned so far.

Open the starter project to begin these challenges.

Challenge 1: The height of the tree

Given a binary tree, find the height of the tree. The height of the binary tree is determined by the distance between the root and the furthest leaf. The height of a binary tree with a single node is zero since the single node is both the root and the furthest leaf.

Solution 1

A recursive approach for finding the height of a binary tree is as follows:

```
fun height(node: BinaryNode<T>? = this): Int {
  return node?.let { 1 + max(height(node.leftChild),
    height(node.rightChild)) } ?: -1
}
```

You recursively call the height function. For every node you visit, you add one to the height of the highest child. If the node is `null`, you return `-1`.

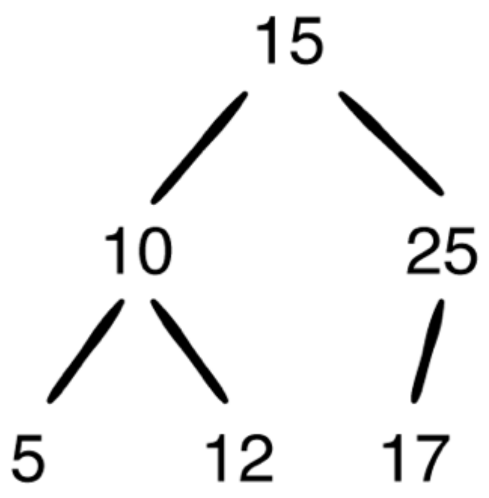
This algorithm has a time complexity of ***O*(*n*)** since you need to traverse through all of the nodes. This algorithm incurs a space cost of ***O*(*n*)** since you need to make the same *n* recursive calls to the call stack.

Challenge 2: Serialization of a Binary Tree

A common task in software development is serializing an object into another data type. This process is known as **serialization**, and it allows custom types to be used in systems that only support a closed set of data types.

An example of serialization is JSON. Your task is to devise a way to serialize a binary tree into a list, and a way to deserialize the list back into the same binary tree.

To clarify this problem, consider the following binary tree:



A particular algorithm may output the serialization as `[15, 10, 5, null,`

`null, 12, null, null, 25, 17, null, null, null]`. The deserialization process should transform the list back into the same binary tree. Note that there are many ways to perform serialization. You may choose any way you wish.

Solution 2

There are many ways to serialize or deserialize a binary tree. Your first task when encountering this question is to decide on the traversal strategy.

For this solution, you'll explore how to solve this challenge by choosing the **pre-order** traversal strategy.

Traversal

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

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

This is the pre-order traversal function. As the code suggests, pre-order traversal traverses each node and visit the node before traversing the children.

It's critical to point out that you'll need to also visit the `null` nodes since it's important to record those for serialization and deserialization.

As with all traversal functions, this algorithm goes through every element of the tree once, so it has a time complexity of ***O*(*n*)**.

Serialization

For serialization, you traverse the tree and store the values into an array.

The elements of the array have type `T?` since you need to keep track of the `null` nodes. Add the following code to **BinaryNode.kt**:

```
fun serialize(node: BinaryNode<T> = this): MutableList<T?> {
  val list = mutableListOf<T?>()
  node.traversePreOrderWithNull { list.add(it) }
  return list
}
```

`serialize` returns a new array containing the values of the tree in pre-order.

The time complexity of the serialization step is ***O*(*n*)**. Because you're creating a new list, this also incurs an ***O*(*n*)** space cost.

Deserialization

In the serialization process, you performed a pre-order traversal and assembled the values into an array. The deserialization process is to take each value of the array and reassemble it back to the tree.

Your goal is to iterate through the array and reassemble the tree in pre-order format. Write the following at the bottom of your **Main.kt**:

```
fun deserialize(list: MutableList<T?>): BinaryNode<T?>? {
  // 1
  val rootValue = list.removeAt(list.size - 1) ?: return null

  // 2
  val root = BinaryNode<T?>(rootValue)

  root.leftChild = deserialize(list)
  root.rightChild = deserialize(list)

  return root
}
```

Here's how the code works:

1. This is the base case. If `removeAt` returns `null`, there are no more elements in the array, thus you'll end recursion here.
2. You reassemble the tree by creating a node from the current value and recursively calling `deserialize` to assign nodes to the left and right children. Notice this is very similar to the pre-order traversal, except, in this case, you're building nodes rather than extracting their values.

Your algorithm is now ready for testing. Write the following at the bottom of `main()`:

```
println(tree)
val array = tree.serialize()
println(tree.deserialize(array))
```

You'll see the following in your console:

```
┌──null
┌──9
│ └──8
7
│ ┌──5
└──1
└──0

┌──null
┌──9
│ └──8
7
│ ┌──5
└──1
└──0
```

Your deserialized tree mirrors the sample tree in the provided code. This is the behavior you want.

However, as mention earlier, the time complexity of this function isn’t desirable. Because you’re calling `removeAt` as many times as there are elements in the array, this algorithm has an ***O*(*n*^{2})** time complexity. There’s an easy way to remedy that.

Write the following function just after the `deserialize` function you created earlier:

```
fun deserializeOptimized(list: MutableList<T?>): BinaryNode<T>? {
  return deserialize(list.asReversed())
}
```

This is a function that first reverses the array before calling the previous `deserialize` function. In the other `deserialize` function, find the `removeAt(0)` call and change it to `list.removeAt(list.size - 1)`:

```
val rootValue = list.removeAt(list.size - 1) ?: return null
```

This small change has a big effect on performance. `removeAt(0)` is an ***O*(*n*)** operation because, after every removal, every element after the removed element must shift left to take up the missing space. In contrast, `list.removeAt(list.size - 1)` is an ***O*(1)** operation.

Finally, find and update the call of `deserialize` to use the new function that reverses the array:

```
println(tree.deserializeOptimized(array))
```

You’ll see the same tree before and after the deserialization process. The time complexity for this solution is now ***O*(*n*)** because you created a new reversed list and chose a recursive solution.

Key points

- The binary tree is the foundation to some of the most important tree structures. The binary search tree and AVL tree are binary trees that impose restrictions on the insertion/deletion behaviors.
- In-order, pre-order and post-order traversals aren't just important only for the binary tree; if you're processing data in any tree, you'll interface with these traversals regularly.