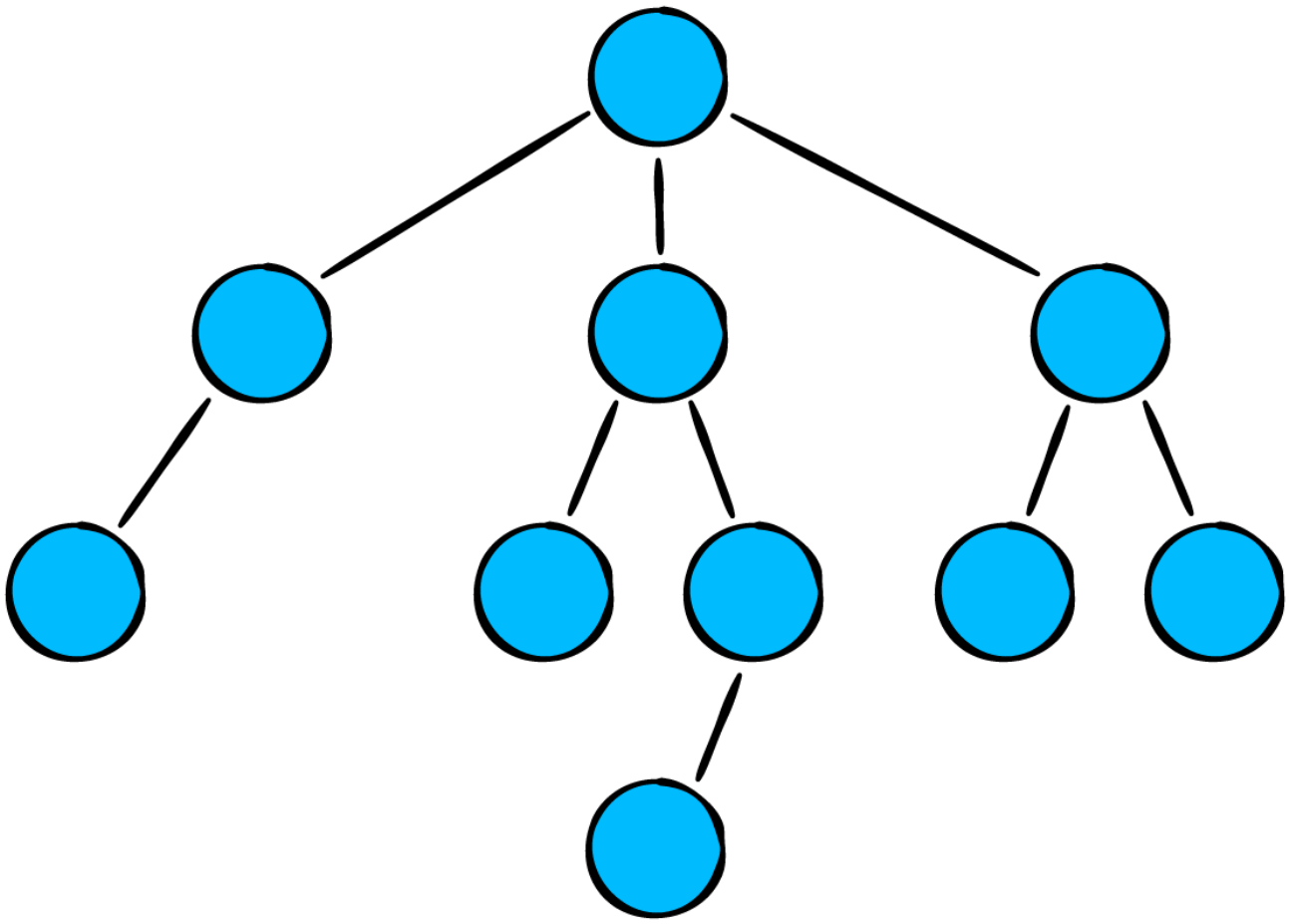
6 Trees Written by Irina Galata



A tree

The **tree** is a data structure of profound importance. It's used to tackle many recurring challenges in software development, such as:

- Representing hierarchical relationships.
- Managing sorted data.
- Facilitating fast lookup operations.

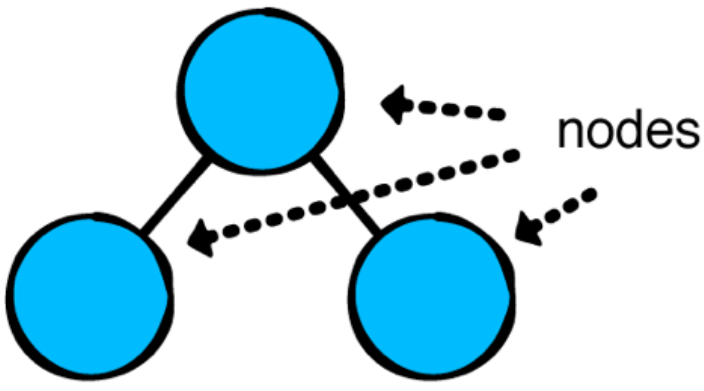
There are many types of trees, and they come in various shapes and sizes. In this chapter, you'll learn the basics of using and implementing a tree.

Terminology

There are many terms associated with trees, so it makes sense to get familiar with a few of them before starting.

Node

Like the linked list, trees are made up of **nodes**.



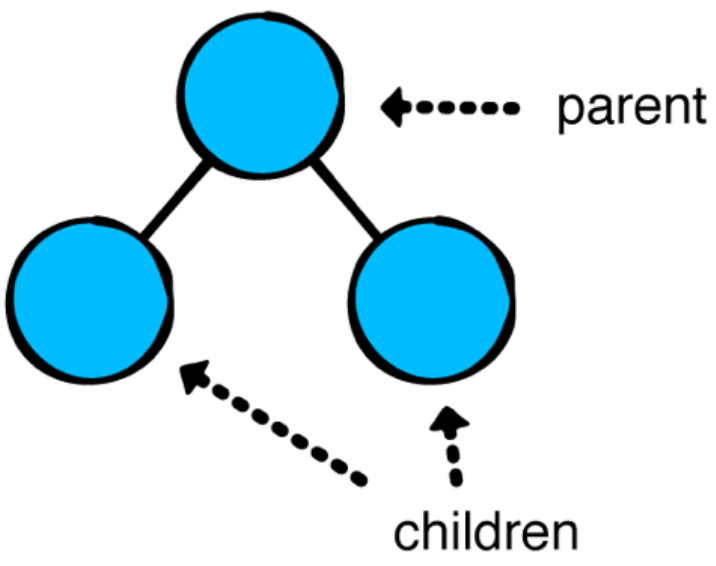
Node

Each node encapsulates some data and keeps track of its **children**.

Parent and child

Trees are viewed starting from the top and branching toward the bottom — just like a real tree, only upside-down.

Every node, except for the first one, is connected to a single node above, which is referred to as a **parent** node. The nodes directly below and connected to the parent node are known as **child** nodes. In a tree, every child has exactly one parent. That's what makes a tree, well, a tree.

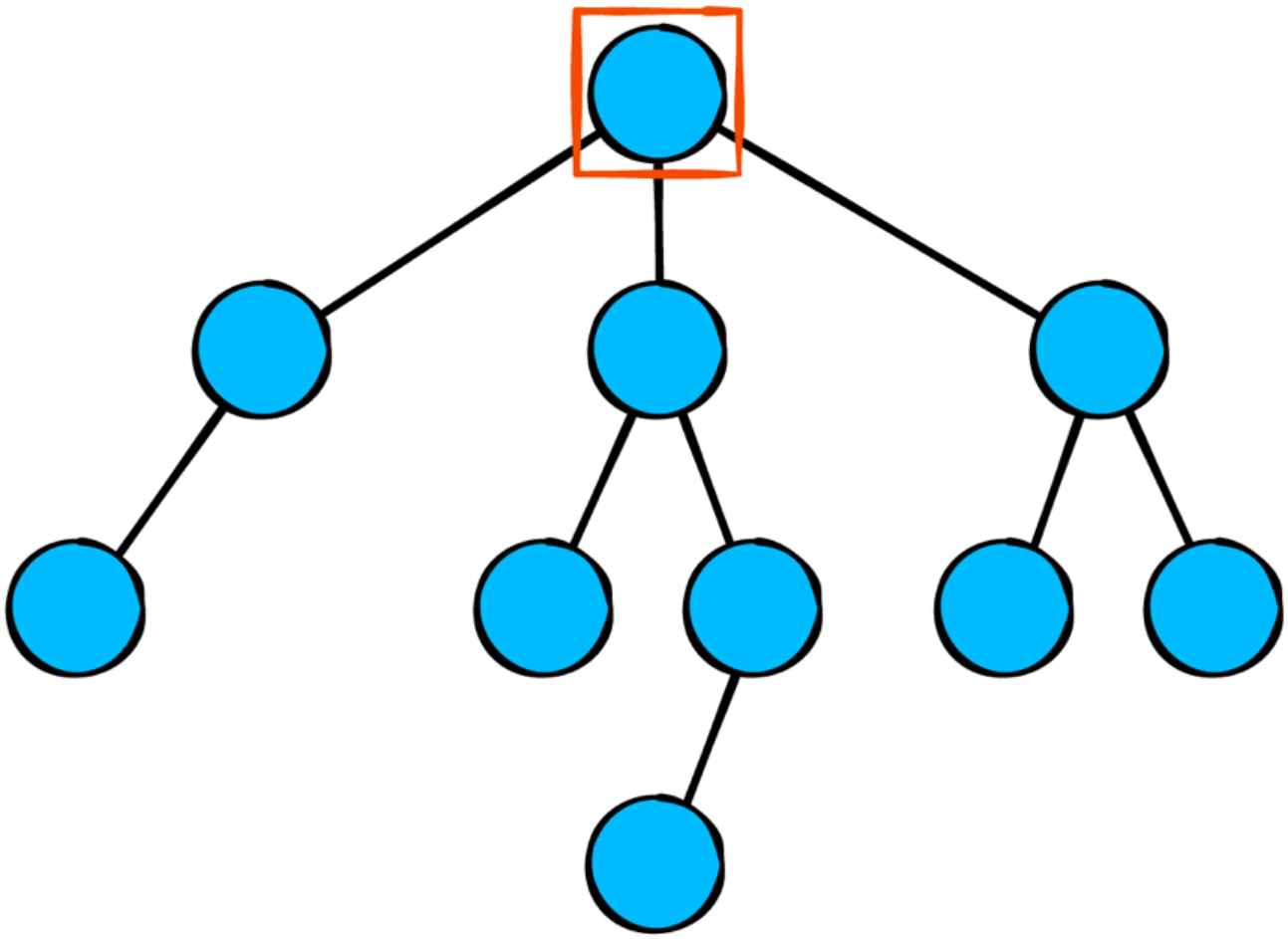


Parent and child

Root

The topmost node in the tree is called the **root** of the tree. It's the only

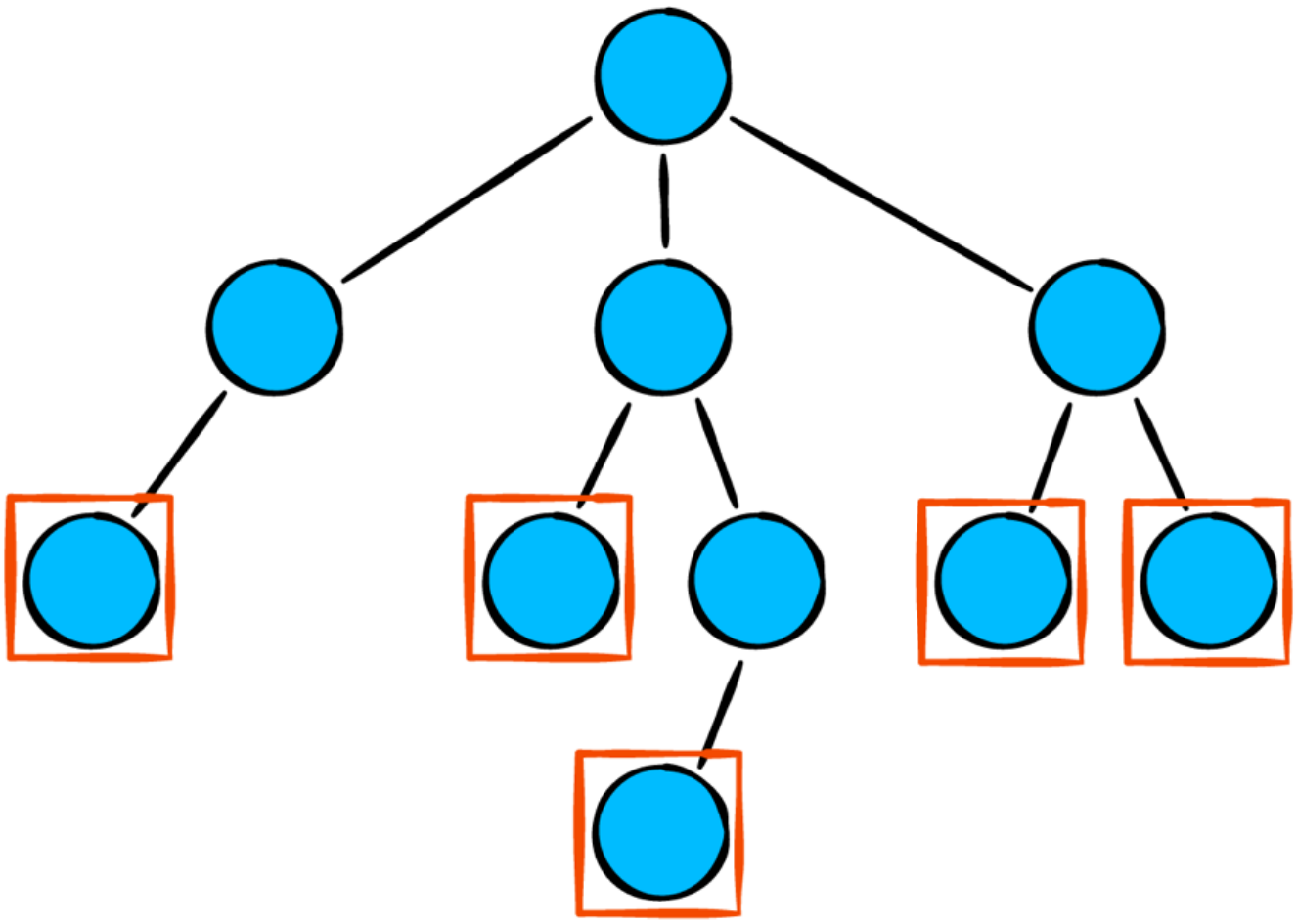
node that has no parent:



Root

Leaf

A node that has no children is called a **leaf**:



You'll run into more terms later, but this should be enough to start coding trees.

Implementation

To get started, open the starter project for this chapter.

A tree is made up of nodes, so your first task is to create a `TreeNode` class.

Create a new file named **TreeNode.kt** and add the following:

```
class TreeNode<T>(val value: T) {
  private val children: MutableList<TreeNode<T>> = mutableListOf()
}
```

Each node is responsible for a `value` and holds references to all of its children using a mutable list.

Next, add the following method inside `TreeNode`:

```
fun add(child: TreeNode<T>) = children.add(child)
```

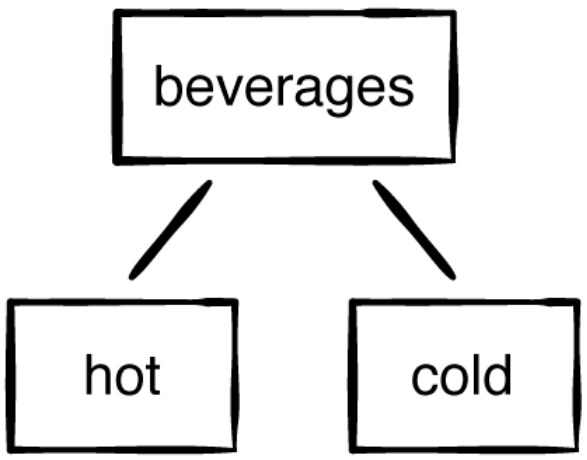
This method adds a child node to a node.

Time to give it a whirl. Go to the `main()` in the **Main.kt** file and add the following:

```
fun main() {
  val hot = TreeNode("Hot")
  val cold = TreeNode("Cold")

  val beverages = TreeNode("Beverages").run {
    add(hot)
    add(cold)
  }
```

```
}
```

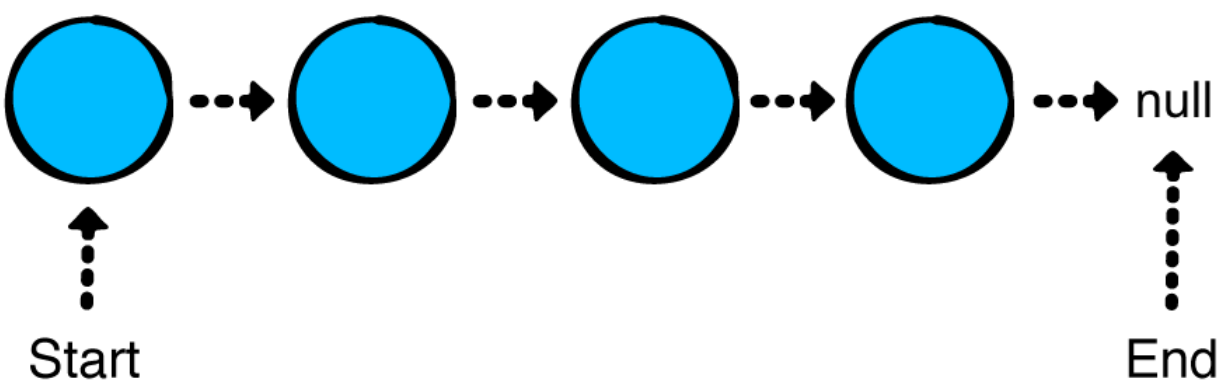
Hierarchical structures are natural candidates for tree structures. That being the case, you define three different nodes and organize them into a logical hierarchy. This arrangement corresponds to the following structure:



A small tree

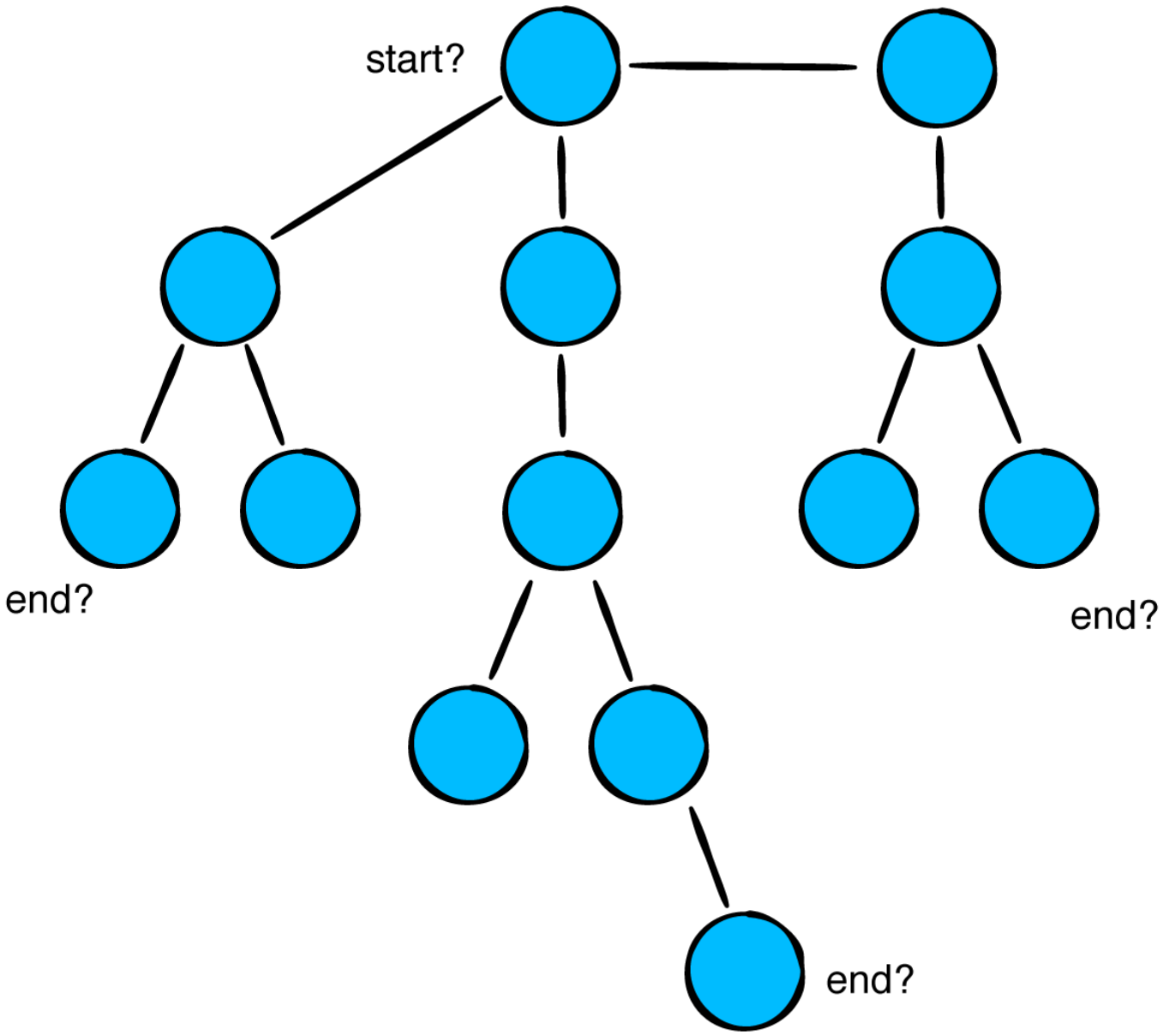
Traversal algorithms

Iterating through **linear collections** such as arrays or lists is straightforward. Linear collections have a clear start and end:



Traversing arrays or lists

Iterating through trees is a bit more complicated:



Should nodes on the left have precedence? How should the depth of a node relate to its precedence? Your traversal strategy depends on the problem you're trying to solve.

There are multiple strategies for different trees and different problems. In all of these ways you can **visit** the node and use the information into them. This is the way you add this definition into the `TreeNode.kt` file outside of the `TreeNode` class definition.

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

In the next section, you'll look at **depth-first traversal**.

Depth-first traversal

Depth-first traversal starts at the root node and explores the tree as far as possible along each branch before reaching a leaf and then backtracking.

Add the following inside `TreeNode`:

```
fun forEachDepthFirst(visit: Visitor<T>) {
  visit(this)
  children.forEach {
    it.forEachDepthFirst(visit)
  }
}
```

This simple code uses recursion to process the next node.

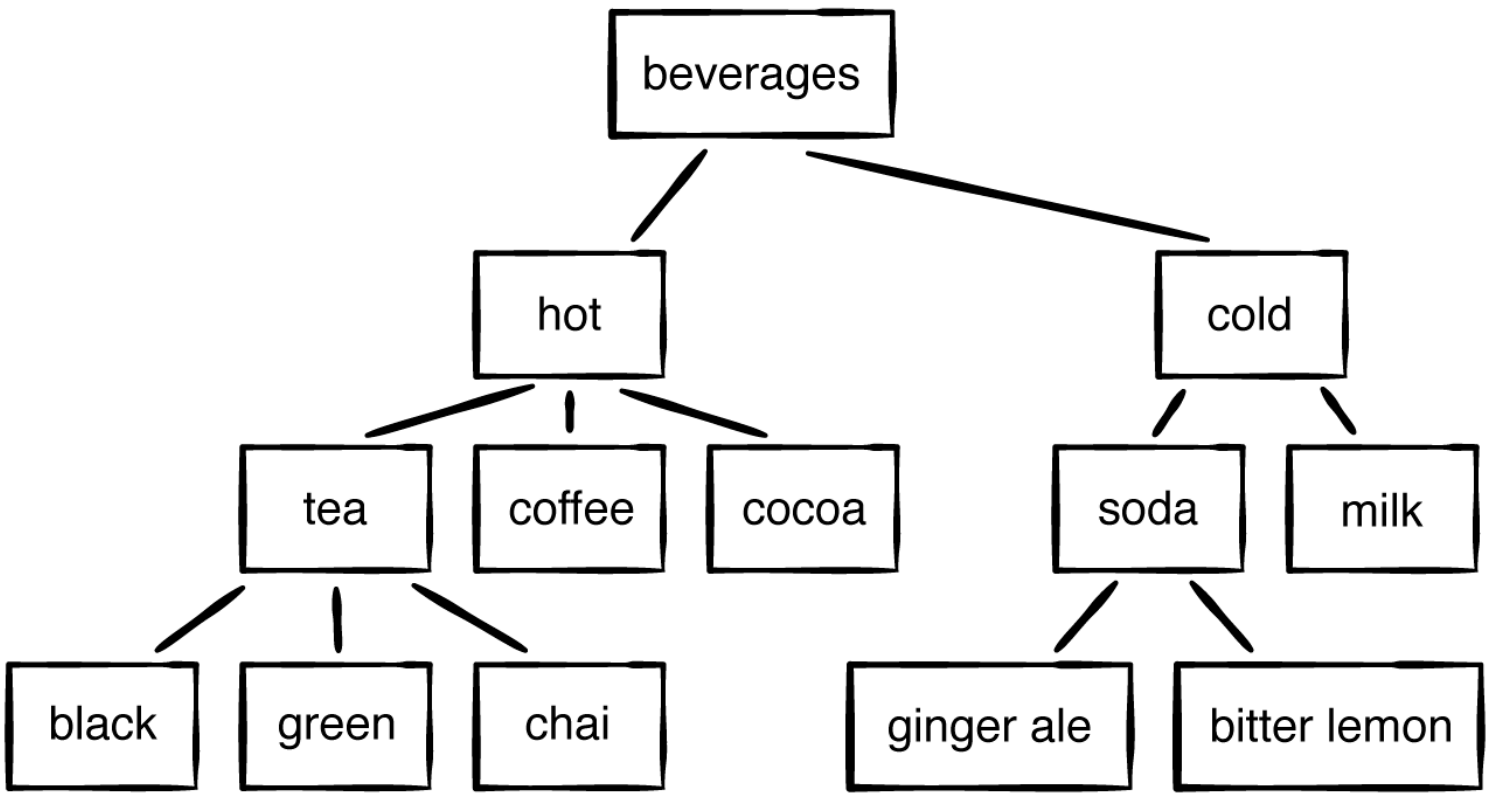
You could use your own stack if you didn't want your implementation to be recursive. However, the recursive solution is more simple and elegant to code.

To test the recursive depth-first traversal function you just wrote, it's helpful to add more nodes to the tree. Go back to **Main.kt** and add the following:

```
fun makeBeverageTree(): TreeNode<String> {
  val tree = TreeNode("Beverages")

  val hot = TreeNode("hot")
  val cold = TreeNode("cold")

  val tea = TreeNode("tea")
  val coffee = TreeNode("coffee")
  val chocolate = TreeNode("cocoa")

  val blackTea = TreeNode("black")
  val greenTea = TreeNode("green")
  val chaiTea = TreeNode("chai")

  val soda = TreeNode("soda")
  val milk = TreeNode("milk")
```

```
val gingerAle = TreeNode("ginger ale")
val bitterLemon = TreeNode("bitter lemon")

tree.add(hot)
tree.add(cold)

hot.add(tea)
hot.add(coffee)
hot.add(chocolate)

cold.add(soda)
cold.add(milk)

tea.add(blackTea)
tea.add(greenTea)
tea.add(chaiTea)

soda.add(gingerAle)
soda.add(bitterLemon)

  return tree
}
```

This function creates the following tree:



A big tree

Next, replace the code in `main()` with the following:

```
fun main() {
  val tree = makeBeverageTree()
  tree.forEachDepthFirst { println(it.value) }
}
```

This code produces the following output that illustrates the order the depth-first traversal visits each node:

```
Beverages
hot
tea
black
green
chai
coffee
cocoa
cold
soda
ginger ale
bitter lemon
milk
```

In the next section, you'll look at **level-order traversal**.

Level-order traversal

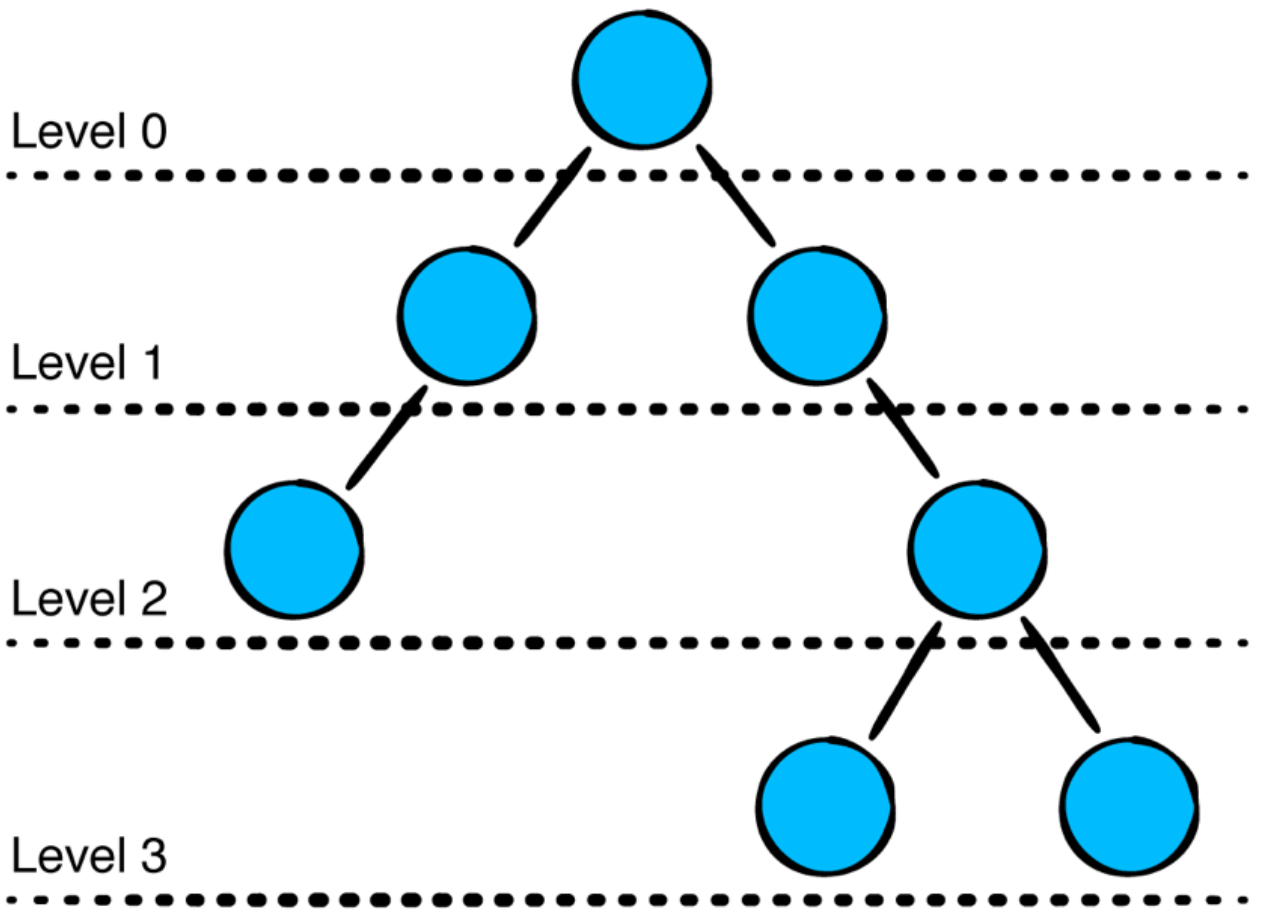
Level-order traversal is a technique that visits each node of the tree based on the depth of the nodes. Starting at the root, every node on a level is visited before going to a lower level.

Add the following inside `TreeNode`:

```
fun forEachLevelOrder(visit: Visitor<T>) {
```

```
    visit(this)
    val queue = Queue<TreeNode<T>>()
    children.forEach { queue.enqueue(it) }

    var node = queue.dequeue()
    while (node != null) {
      visit(node)
      node.children.forEach { queue.enqueue(it) }
      node = queue.dequeue()
    }
}
```

`forEachLevelOrder` visits each of the nodes in level-order:



Level-order traversal

Note how you use a queue to ensure that nodes are visited in the right level-order. You start visiting the current node and putting all its children into the queue. Then you start consuming the queue until it's empty. Every time you visit a node, you also put all its children into the queue. This ensures that all nodes at the same level are visited one after the other.

Open **Main.kt** and add the following:

```
fun main() {
  val tree = makeBeverageTree()
  tree.forEachLevelOrder { println(it.value) }
}
```

In the console, you'll see the following output:

```
beverages
hot
cold
tea
coffee
cocoa
soda
milk
black
green
chai
ginger ale
bitter lemon
```

Search

You already have a method that iterates through the nodes, so building a search algorithm won't take long.

Add the following inside `TreeNode`:

```
fun search(value: T): TreeNode<T>? {
  var result: TreeNode<T>? = null

  forEachLevelOrder {
    if (it.value == value) {
      result = it
    }
  }
```

```
  return result
}
```

To test your code, go back to `main()`. To save some time, copy the previous example and modify it to test the `search` method:

```
fun main() {
  val tree = makeBeverageTree()
  tree.search("ginger ale")?.let {
    println("Found node: ${it.value}")
  }

  tree.search("WKD Blue")?.let {
    println(it.value)
  } ?: println("Couldn't find WKD Blue")
}
```

You'll see the following console output:

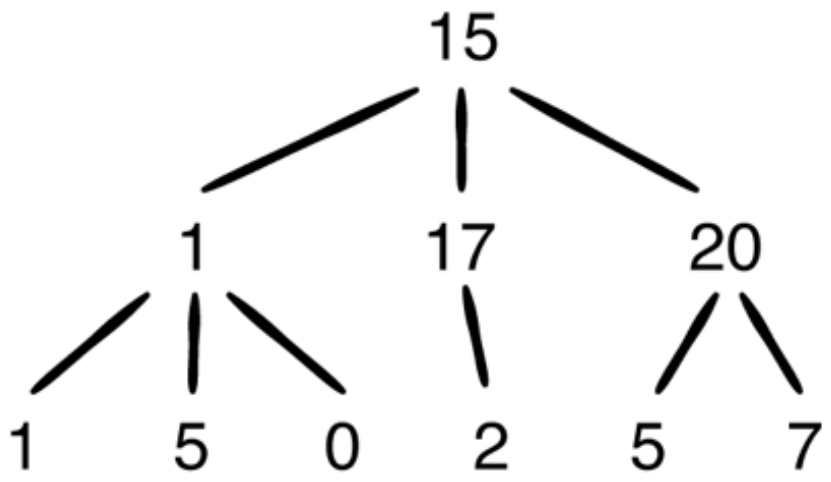
```
Found node: ginger ale
Couldn't find WKD Blue
```

Here, you used your level-order traversal algorithm. Since it visits all nodes, if there are multiple matches, the last match wins. This means that you'll get different objects back depending on what traversal you use.

Challenges

Challenge 1: Tree challenge

Print the values in a tree in an order based on their level. Nodes belonging to the same level should be printed on the same line. For example, consider the following tree:



Your algorithm should output the following in the console:

```
15
1 17 20
1 5 0 2 5 7
```

> **Hint**: Consider using a `Queue` included for you in the starter project.

Solution 1

A straightforward way to print the nodes in level-order is to leverage the level-order traversal using a `Queue` data structure. The tricky bit is determining when a new line should occur.

Here's the solution:

```
fun printEachLevel() {
  // 1
  val queue = ArrayListQueue<TreeNode<T>>()
  var nodesLeftInCurrentLevel = 0

  queue.enqueue(this)
  // 2
  while (queue.isEmpty.not()) {
    // 3
    nodesLeftInCurrentLevel = queue.count
```

```
    // 4
    while (nodesLeftInCurrentLevel > 0) {
      val node = queue.dequeue()
      node?.let {
        print("${node.value} ")
        node.children.forEach { queue.enqueue(it) }
        nodesLeftInCurrentLevel--
      } ?: break
    }

    // 5
    println()
  }
}
```

And here's how it works:

1. You begin by initializing a `Queue` data structure to facilitate the level-order traversal. You also create `nodesLeftInCurrentLevel` to keep track of the number of nodes you'll need to work on before you print a new line.
2. Your level-order traversal continues until your queue is empty.
3. Inside the first `while` loop, you begin by setting `nodesLeftInCurrentLevel` to the current elements in the queue.
4. Using another `while` loop, you dequeue the first `nodesLeftInCurrentLevel` number of elements from the queue. Every element you dequeue is printed *without* establishing a new line. You also enqueue all the children of the node.
5. At this point, you generate the new line using `println()`. In the next iteration, `nodesLeftInCurrentLevel` is updated with the count of the queue, representing the number of children from the previous iteration.

This algorithm has a time complexity of ***O*(*n*)**. Since you initialize the `Queue` data structure as an intermediary container, this algorithm also uses ***O*(*n*)** space.

Key points

- Trees share some similarities to linked lists. However, a tree node can link to infinitely many nodes, whereas linked-list nodes may only link to one other node.
- Get comfortable with the tree terminology such as a parent, child, leaf, and root. Many of these terms are common and are used to help explain other tree structures.
- Traversals, such as depth-first and level-order traversals, aren't specific to only the general type of tree. They work on other types of trees as well, although their implementation is slightly different based on how the tree is structured.
- Trees are a fundamental data structure with different implementations. Some of these will be part of the next chapters.