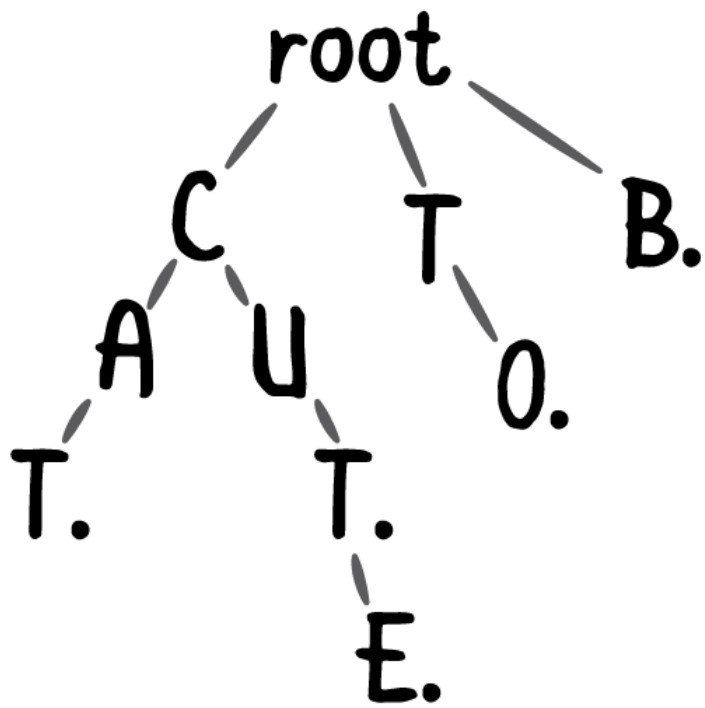
10 Tries Written by Irina Galata

The **trie** (pronounced *try*) is a tree that specializes in storing data that can be represented as a collection, such as English words:



A trie containing the words CAT, CUT, CUTE, TO, and B

Each character in a string is mapped to a node. The last node in each string is marked as a terminating node (a dot in the image above). The benefits of a trie are best illustrated by looking at it in the context of prefix matching.

In this chapter, you'll first compare the performance of the trie to the array. You'll then implement the trie from scratch.

Example

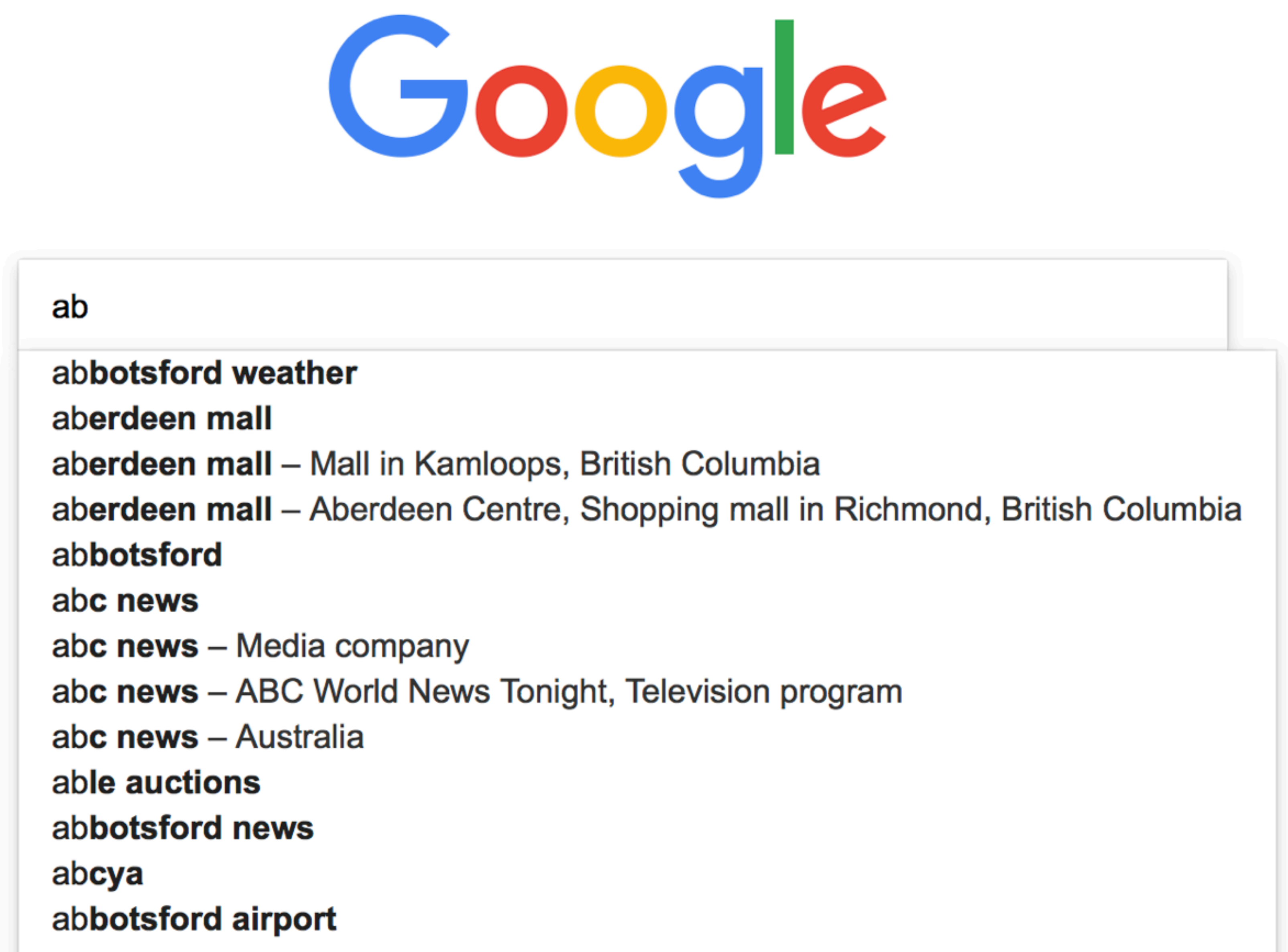
You are given a collection of strings. How would you build a component that handles prefix matching? Here's one way:

```
class EnglishDictionary {

  private val words: ArrayList<String> = ...
```

```
  fun words(prefix: String) = words.filter { it.startsWith(prefix) }

}
```

`words()` goes through the collection of strings and returns the strings that match the prefix.

If the number of elements in the `words` array is small, this is a reasonable strategy. But if you're dealing with more than a few thousand words, the time it takes to go through the `words` array will be unacceptable. The time complexity of `words()` is $O(k*n)$, where k is the longest string in the collection, and n is the number of words you need to check.



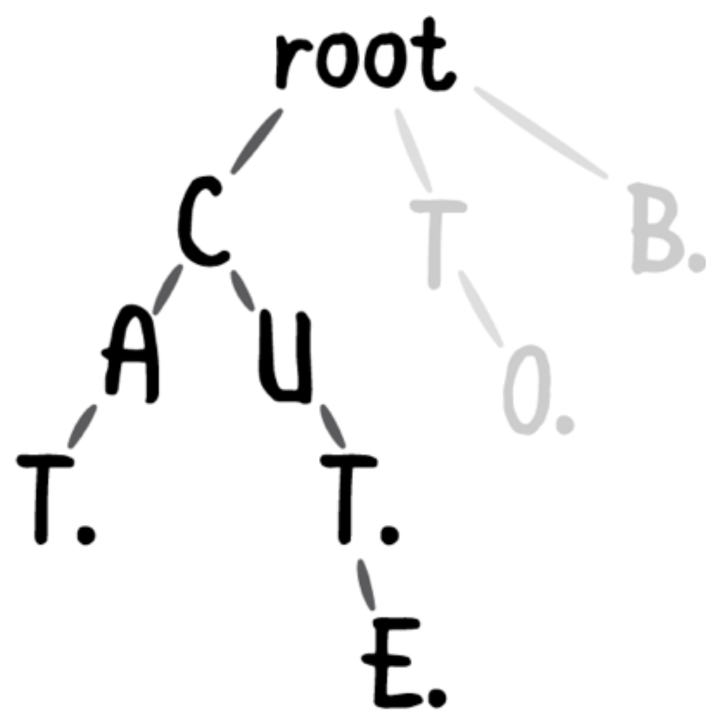
Imagine the number of words Google needs to parse

The trie data structure has excellent performance characteristics for this type of problem; like a tree with nodes that support multiple children, each node can represent a single character.

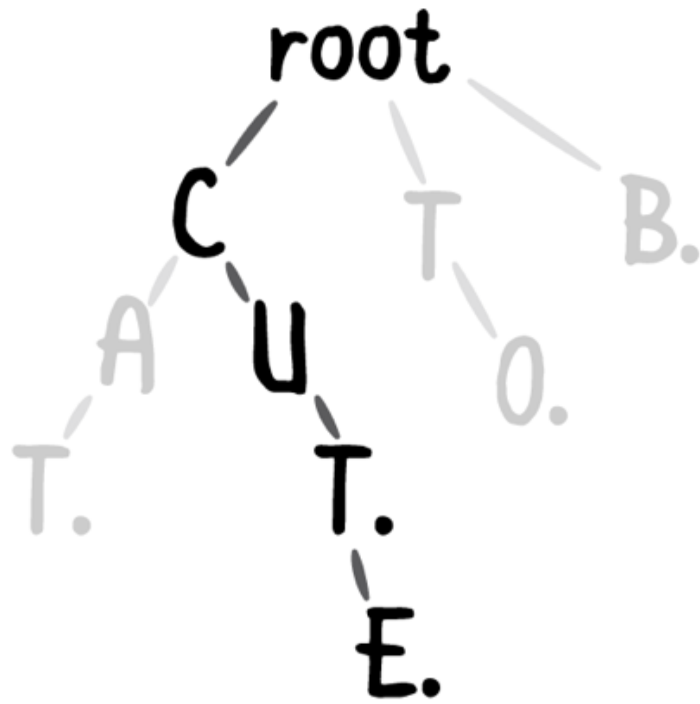
You form a word by tracing the collection of characters from the root to a node with a special indicator — a terminator — represented by a black dot. An interesting characteristic of the trie is that multiple words can share the same characters.

To illustrate the performance benefits of the trie, consider the following example in which you need to find the words with the prefix CU.

First, you travel to the node containing C. This quickly excludes other branches of the trie from the search operation:

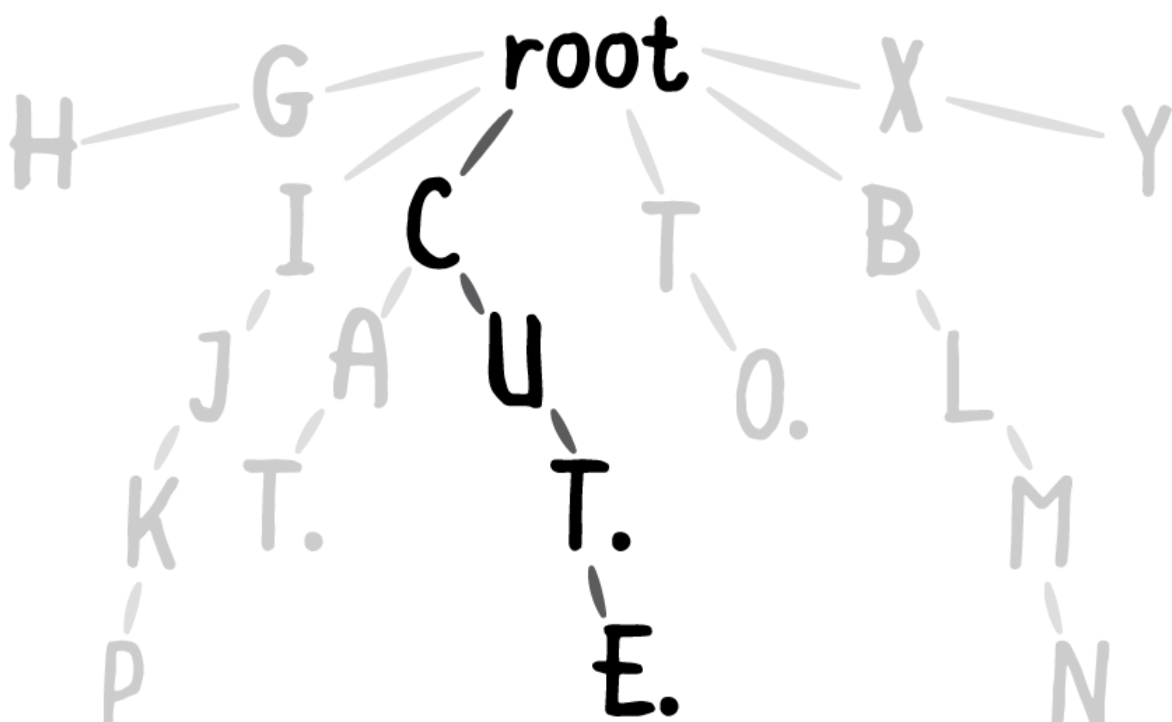


Next, you need to find the words that have the next letter, U. You traverse to the U node:



Since that's the end of your prefix, the trie returns all collections formed by the chain of nodes from the `U` node. In this case, the words `CUT` and `CUTE` are returned. Imagine if this trie contained hundreds of thousands of words.

The number of comparisons you can avoid by employing a trie is substantial.



Implementation

Open up the starter project for this chapter.

TrieNode

You'll begin by creating the node for the trie. Create a new file named **TrieNode.kt**. Add the following to the file:

```
class TrieNode<Key: Any>(var key: Key?, var parent: TrieNode<Key>?) {

  val children: HashMap<Key, TrieNode<Key>> = HashMap()

  var isTerminating = false

}
```

This interface is slightly different compared to the other nodes you've encountered:

1. `key` holds the data for the node. This is optional because the root node of the trie has no key.
2. A `TrieNode` holds a reference to its parent. This reference simplifies `remove()` later on.
3. In binary search trees, nodes have a left and right child. In a trie, a node needs to hold multiple different elements. You've declared a `children` map to help with that.
4. As discussed earlier, `isTerminating` acts as an indicator for the end of a collection.

Trie

Next, you'll create the trie itself, which will manage the nodes. Create a new file named **Trie.kt**. Add the following to the file:

```
class Trie<Key: Any> {

  private val root = TrieNode<Key>(key = null, parent = null)

}
```

The `Trie` class can store collections containing `Key`s.

Next, you'll implement four operations for the trie: `insert`, `contains`, `remove` and a prefix match.

Insert

Tries work with lists of the `Key` type. The trie takes the list and represents it as a series of nodes in which each node maps to an element in the list.

Add the following method to `Trie`:

```
fun insert(list: List<Key>) {
  // 1
  var current = root

  // 2
  list.forEach { element ->
    val child = current.children[element] ?: TrieNode(element, current)
    current.children[element] = child
    current = child
  }

  // 3
  current.isTerminating = true
}
```

Here's what's going on:

1. `current` keeps track of your traversal progress, which starts with the

root node.

2. A trie stores each element of a list in separate nodes. For each element of the list, you first check if the node currently exists in the `children` map. If it doesn't, you create a new node. During each loop, you move `current` to the next node.
3. After iterating through the `for` loop, `current` should be referencing the node representing the end of the list. You mark that node as the terminating node.

The time complexity for this algorithm is $O(k)$, where k is the number of elements in the list you're trying to insert. This is because you need to traverse through or create each node that represents each element of the new list.

Contains

`contains` is similar to `insert`. Add the following method to `Trie`:

```
fun contains(list: List<Key>): Boolean {
  var current = root

  list.forEach { element ->
    val child = current.children[element] ?: return false
    current = child
  }

  return current.isTerminating
}
```

Here, you traverse the trie in a way similar to `insert`. You check every element of the list to see if it's in the tree. When you reach the last element of the list, it must be a terminating element. If not, the list wasn't added to the tree and what you've found is merely a subset of a larger list.

The time complexity of `contains` is $O(k)$, where k is the number of elements in the list that you're looking for. This is because you need to traverse through *k* nodes to find out whether or not the list is in the trie.

To test `insert` and `contains`, navigate to `main()` and add the following code:

```
"insert and contains" example {
  val trie = Trie<Char>()
  trie.insert("cute".toList())
  if (trie.contains("cute".toList())) {
    println("cute is in the trie")
  }
}
```

`String` is not a collection type in Kotlin, but you can easily convert it to a list of characters using the `toList` extension.

After running `main()`, you'll see the following console output:

```
---Example of insert and contains---
cute is in the trie
```

You can make storing `String`s in a trie more convenient by adding some extensions. Create a file named **Extensions.kt**, and add the following:

```
fun Trie<Char>.insert(string: String) {
  insert(string.toList())
}

fun Trie<Char>.contains(string: String): Boolean {
  return contains(string.toList())
}
```

These extension functions are only applicable to tries that store lists of characters. They hide the extra `toList()` calls you need to pass in a `String`, allowing you to simplify the previous code example to this:

```
"insert and contains" example {
  val trie = Trie<Char>()
  trie.insert("cute")
  if (trie.contains("cute")) {
    println("cute is in the trie")
  }
}
```

Remove

Removing a node in the trie is a bit more tricky. You need to be particularly careful when removing each node since nodes can be shared between multiple different collections. Write the following method immediately below `contains`:

```
fun remove(list: List<Key>) {
  // 1
  var current = root

  list.forEach { element ->
    val child = current.children[element] ?: return
    current = child
  }

  if (!current.isTerminating) return

  // 2
  current.isTerminating = false

  // 3
  val parent = current.parent
  while (parent != null && current.children.isEmpty() && !current.isTermina
    parent.children.remove(current.key)
    current = parent
  }
}
```

Here's how it works:

1. This part should look familiar, as it's basically the implementation of `contains`. You use it here to check if the collection is part of the trie and to point `current` to the last node of the collection.

2. You set `isTerminating` to `false` so that the current node can be removed by the loop in the next step.

3. This is the tricky part. Since nodes can be shared, you don't want to carelessly remove elements that belong to another collection. If there are no other children in the current node, it means that other collections do not depend on the current node.

   You also check to see if the current node is a terminating node. If it is, then it belongs to another collection. As long as `current` satisfies these conditions, you continually backtrack through the `parent` property and remove the nodes.

The time complexity of this algorithm is $O(k)$, where k represents the number of elements of the collection that you're trying to remove.

Sticking to strings, it's time to add another extension in **Extensions.kt**:

```
fun Trie<Char>.remove(string: String) {
  remove(string.toList())
}
```

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

```
"remove" example {
  val trie = Trie<Char>()

  trie.insert("cut")
  trie.insert("cute")
```

```
  println("\n*** Before removing ***")
  assert(trie.contains("cut"))
  println("\"cut\" is in the trie")
  assert(trie.contains("cute"))
  println("\"cute\" is in the trie")

  println("\n*** After removing cut ***")
  trie.remove("cut")
  assert(!trie.contains("cut"))
  assert(trie.contains("cute"))
  println("\"cute\" is still in the trie")
}
```

You'll see the following output in the console:

```
---Example of: remove---

*** Before removing ***
"cut" is in the trie
"cute" is in the trie

*** After removing cut ***
"cute" is still in the trie
```

Prefix matching

The most iconic algorithm for the trie is the prefix-matching algorithm. Write the following at the bottom of `Trie`:

```
fun collections(prefix: List<Key>): List<List<Key>> {
  // 1
  var current = root

  prefix.forEach { element ->
    val child = current.children[element] ?: return emptyList()
    current = child
  }
```

```
  // 2
  return collections(prefix, current)
}
```

Here’s how it works:

1. You start by verifying that the trie contains the prefix. If not, you return an empty list.
2. After you’ve found the node that marks the end of the prefix, you call a recursive helper method to find all of the sequences after the `current` node.

Next, add the code for the helper method:

```
private fun collections(prefix: List<Key>, node: TrieNode<Key>?): List<List
  // 1
  val results = mutableListOf<List<Key>>()

  if (node?.isTerminating == true) {
    results.add(prefix)
  }

  // 2
  node?.children?.forEach { (key, node) ->
    results.addAll(collections(prefix + key, node))
  }

  return results
}
```

This code works like so:

1. You create a `MutableList` to hold the results. If the current node is a terminating node, you add the corresponding prefix to the results.
2. Next, you need to check the current node’s children. For every child node, you recursively call `collections()` to seek out other

terminating nodes.

`collection()` has a time complexity of *O*(*k***m*), where *k* represents the longest collection matching the prefix and *m* represents the number of collections that match the prefix.

Recall that arrays have a time complexity of *O*(*k***n*), where *n* is the number of elements in the collection.

For large sets of data in which each collection is uniformly distributed, tries have far better performance as compared to using arrays for prefix matching.

Time to take the method for a spin. Add a handy extension first, in **Extensions.kt**:

```
fun Trie<Char>.collections(prefix: String): List<String> {
  return collections(prefix.toList()).map { it.joinToString(separator = "") }
}
```

This extension maps the input string into a list of characters, and then maps the lists in the result of the `collections()` call back to strings. Neat!

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

```
"prefix matching" example {
  val trie = Trie<Char>().apply {
    insert("car")
    insert("card")
    insert("care")
    insert("cared")
    insert("cars")
    insert("carbs")
    insert("carapace")
    insert("cargo")
  }
```

```
  println("\nCollections starting with \"car\"")
  val prefixedWithCar = trie.collections("car")
  println(prefixedWithCar)

  println("\nCollections starting with \"care\"")
  val prefixedWithCare = trie.collections("care")
  println(prefixedWithCare)
}
```

You'll see the following output in the console:

```
---Example of prefix matching---

Collections starting with "car"
[car, carapace, carbs, cars, card, care, cared, cargo]

Collections starting with "care"
[care, cared]
```

Challenges

Challenge 1: Adding more features

The current implementation of the trie is missing some notable operations. Your task for this challenge is to augment the current implementation of the trie by adding the following:

1. A `lists` property that returns all of the lists in the trie.

2. A `count` property that tells you how many lists are currently in the trie.

3. An `isEmpty` property that returns `true` if the trie is empty, `false` otherwise.

Solution 1

For this solution, you'll implement `lists` as a computed property. It'll be backed by a private property named `storedLists`.

Inside **Trie.kt**, add the following new properties:

```
private val storedLists: MutableSet<List<Key>> = mutableSetOf()

val lists: List<List<Key>>
  get() = storedLists.toList()
```

`storedLists` is a set of the lists currently contained by the trie. Reading the `lists` property returns a list of these tries, which is created from the privately maintained set.

Next, inside `insert()`, find the line `current.isTerminating = true` and add the following immediately below it:

```
storedLists.add(list)
```

In `remove()`, find the line `current.isTerminating = false` and add the following immediately above that line:

```
storedLists.remove(list)
```

Adding the `count` and `isEmpty` properties is straightforward now that you're keeping track of the lists:

```
val count: Int
  get() = storedLists.count()

val isEmpty: Boolean
  get() = storedLists.isEmpty()
```

Key points

- Tries provide great performance metrics in regards to prefix matching.
- Tries are relatively memory efficient since individual nodes can be shared between many different values. For example, “car”, “carbs”, and “care” can share the first three letters of the word.

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.