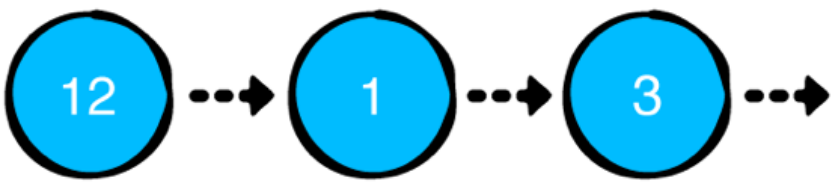
3 Linked List Written by Márton Braun

A linked list is a collection of values arranged in a linear, unidirectional sequence. A linked list has several theoretical advantages over contiguous storage options such as the Kotlin `Array` or `ArrayList`:

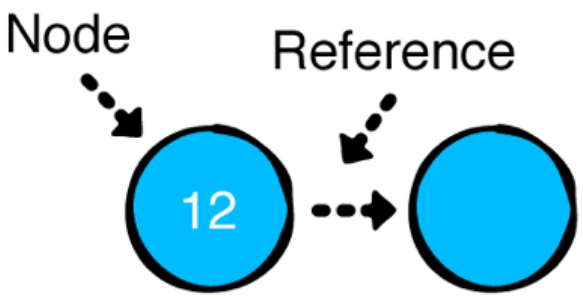
- Constant time insertion and removal from the front of the list.
- Reliable performance characteristics.



A linked list

As the diagram suggests, a linked list is a chain of **nodes**. Nodes have two responsibilities:

1. Hold a value.
2. Hold a reference to the next node. The absence of a reference to the next node, `null`, marks the end of the list.



A node holding the value 12

In this chapter, you'll implement a linked list and learn about the common operations associated with it. You'll also learn about the time complexity of each operation. Open the starter project for this chapter so that you can dive right into the code.

Node

Create a new Kotlin file in **src** and name it **Node.kt**. Add the following to the file:

```
data class Node<T : Any>(var value: T, var next: Node<T>? = null) {
  override fun toString(): String {
    return if (next != null) {
      "$value -> ${next.toString()}"
    } else {
      "$value"
    }
  }
}
```

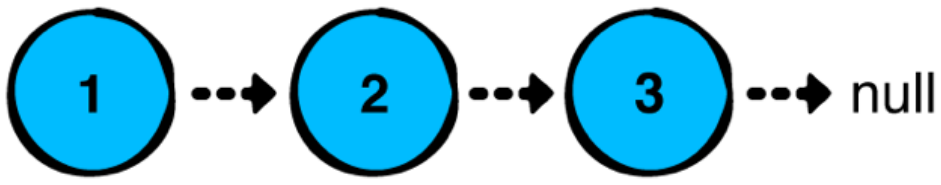
> **Note:** Using `T : Any` to set an upper bound for the type parameter ensures that `T` will always be a non-nullable type.

Navigate to the **Main.kt** file and add the following inside `main()`:

```
fun main() {
  "creating and linking nodes" example {
    val node1 = Node(value = 1)
    val node2 = Node(value = 2)
    val node3 = Node(value = 3)

    node1.next = node2
    node2.next = node3

    println(node1)
  }
}
```

> **Note**: `example` is a helper function used throughout the book to format console output nicely. You can find it in the **Utils.kt** file of each chapter's starter project.

You've just created three nodes and connected them:



A linked list containing values 1, 2, and 3

Once you run `Main.kt`, you'll see the following output in the console:

```
---Example of creating and linking nodes---
1 -> 2 -> 3
```

As far as practicality goes, this method of building lists is far from ideal. You can easily see that building long lists in this way is impractical. A common way to alleviate this problem is to build a `LinkedList` that manages the `Node` objects. You'll do just that!

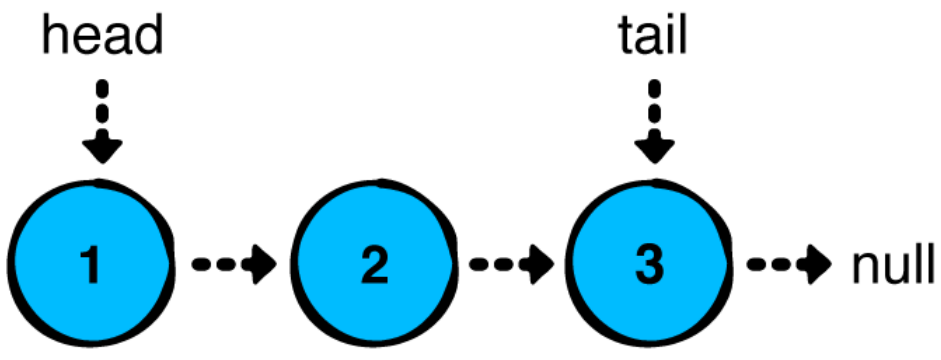
LinkedList

In **src**, create a new file and name it **LinkedList.kt**. Add the following to the file:

```
class LinkedList<T : Any> {

  private var head: Node<T>? = null
  private var tail: Node<T>? = null
  private var size = 0

  fun isEmpty(): Boolean = size == 0

  override fun toString(): String {
    if (isEmpty()) {
      return "Empty list"
    } else {
      return head.toString()
```

```
    }
  }
}
```

A linked list has the concept of a **head** and **tail**, which refers to the first and last nodes of the list respectively:



The head and tail of the list

You'll also keep track of the size of the linked list in the `size` property. This might not seem useful yet, but it will come in handy later.

Adding values to the list

Next, you're going to provide an interface to manage the `Node` objects. You'll first take care of adding values. There are three ways to add values to a linked list, each having their own unique performance characteristics:

1. **`push`**: Adds a value at the front of the list.
2. **`append`**: Adds a value at the end of the list.
3. **`insert`**: Adds a value after a particular node of the list.

You'll implement each of these in turn and analyze their performance characteristics.

Push operations

Adding a value at the front of the list is known as a `push` operation. This is also known as **head-first insertion**. The code for it is deliciously simple.

Add the following method to `LinkedList`:

```
fun push(value: T) {
  head = Node(value = value, next = head)
  if (tail == null) {
    tail = head
  }
  size++
}
```

You create a new `Node` which holds the new value, and points to the node that was previously the `head` of the list.

In the case in which you're pushing into an empty list, the new node is both the `head` and `tail` of the list. Since the list now has a new node, you increment the value of `size`.

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

```
"push" example {
  val list = LinkedList<Int>()
  list.push(3)
  list.push(2)
  list.push(1)

  println(list)
}
```

Your console output will show this:

```
---Example of push---
1 -> 2 -> 3
```

This is pretty cool, but you can do even better. You'll use the **fluent interface** pattern to chain multiple `push` calls. Go back to `push()` and add

`LinkedList<T>` as its return type. Then, wrap the code of the method in an `apply` call to return the list that you've just pushed an element into.

The method will now look like this:

```
fun push(value: T): LinkedList<T> = apply {
  head = Node(value = value, next = head)
  if (tail == null) {
    tail = head
  }
  size++
}
```

In `main()`, you can now rewrite the previous example, making use of `push()`'s return value:

```
"fluent interface push" example {
  val list = LinkedList<Int>()
  list.push(3).push(2).push(1)
  println(list)
}
```

That's more like it! Now that you can add multiple elements to the start of the list with ease.

Append operations

The next operation you'll look at is `append`. This adds a value at the end of the list, which is known as **tail-end insertion**.

In **LinkedList.kt**, add the following code just below `push()`:

```
fun append(value: T) {
  // 1
  if (isEmpty()) {
    push(value)
```

```
    return
  }
  // 2
  val newNode = Node(value = value)
  tail!!.next = newNode

  // 3
  tail = newNode
  size++
}
```

This code is relatively straightforward:

1. Like before, if the list is empty, you'll need to update both `head` and `tail` to the new node. Since `append` on an empty list is functionally identical to `push`, you invoke `push` to do the work for you.
2. In all other cases, you create a new node *after* the current `tail` node. `tail` will never be `null` here because you've already handled the case where the list is empty in the `if` statement.
3. Since this is tail-end insertion, your new node is also the tail of the list. You also have to increment `size` since a new value was added to the list.

Go back to **Main.kt** and write the following at the bottom of `main()`:

```
"append" example {
  val list = LinkedList<Int>()
  list.append(1)
  list.append(2)
  list.append(3)

  println(list)
}
```

You'll see the following output in the console:

```
---Example of append---
1 -> 2 -> 3
```

You can use the trick you learned for `push()` to get a fluid interface here too. It's up to you if you've liked it or not, but imagine how you could chain pushes and appends in a world of endless possibilities! Or just have some fun with it. :]

Insert operations

The third and final operation for adding values is `insert`. This operation inserts a value at a particular place in the list and requires two steps:

1. Finding a particular node in the list.
2. Inserting the new node after that node.

First, you'll implement the code to find the node where you want to insert your value.

In **LinkedList.kt**, add the following code just below `append`:

```
fun nodeAt(index: Int): Node<T>? {
  // 1
  var currentNode = head
  var currentIndex = 0

  // 2
  while (currentNode != null && currentIndex < index) {
    currentNode = currentNode.next
    currentIndex++
  }

  return currentNode
}
```

`nodeAt()` tries to retrieve a node in the list based on the given index. Since

you can only access the nodes of the list from the head node, you'll have to make iterative traversals. Here's the play-by-play:

1. You create a new reference to `head` and keep track of the current number of steps taken in the list.

2. Using a `while` loop, you move the reference down the list until you reach the desired index. Empty lists or out-of-bounds indexes will result in a `null` return value.

Now, you need to insert the new node.

Add the following method just below `nodeAt()`:

```
fun insert(value: T, afterNode: Node<T>): Node<T> {
  // 1
  if (tail == afterNode) {
    append(value)
    return tail!!
  }
  // 2
  val newNode = Node(value = value, next = afterNode.next)
  // 3
  afterNode.next = newNode
  size++
  return newNode
}
```

Here's what you've done:

1. In the case where this method is called with the `tail` node, you call the functionally equivalent `append` method. This takes care of updating `tail`.
2. Otherwise, you create a new node and link its `next` property to the next node of the list.
3. You reassign the `next` value of the specified node to link it to the new node that you just created.

To test things, go back to **Main.kt** and add the following to the bottom of `main()`:

```
"inserting at a particular index" example {
  val list = LinkedList<Int>()
  list.push(3)
  list.push(2)
  list.push(1)

  println("Before inserting: $list")
  var middleNode = list.nodeAt(1)!!
  for (i in 1..3) {
    middleNode = list.insert(-1 * i, middleNode)
  }
  println("After inserting: $list")
}
```

You'll see the following output:

```
---Example of inserting at a particular index---
Before inserting: 1 -> 2 -> 3
After inserting: 1 -> 2 -> -1 -> -2 -> -3 -> 3
```

Performance analysis

Whew! You made good progress so far. To recap, you've implemented the three operations that add values to a linked list and a method to find a node at a particular index.

	push	append	insert	nodeAt
Behaviour	insert at head	insert at tail	insert after a node	returns a node at given index
Time complexity	O(1)	O(1)	O(1)	O(i), where i is the given index

Next, you'll focus on the opposite action: removal operations.

Removing values from the list

There are three primary operations for removing nodes:

1. **`pop`**: Removes the value at the front of the list.
2. **`removeLast`**: Removes the value at the end of the list.
3. **`removeAfter`**: Removes a value after a particular node of the list.

You'll implement all three and analyze their performance characteristics.

Pop operations

Removing a value at the front of the list is often referred to as **pop**. This operation is almost as simple as `push()`, so dive right in.

Add the following method to `LinkedList`:

```
fun pop(): T? {
  if (isEmpty()) return null

  val result = head?.value
  head = head?.next
  size--
  if (isEmpty()) {
```

```
    tail = null
  }

  return result
}
```

`pop()` returns the value that was removed from the list. This value is nullable since it's possible that the list is empty.

By moving the `head` down a node, you've effectively removed the first node of the list. The garbage collector will remove the old node from memory once the method finishes since there will be no more references attached to it. If the list becomes empty, you set `tail` to `null` as well.

To test, go to **Main.kt** and add the following code at the bottom inside `main()`:

```
"pop" example {
  val list = LinkedList<Int>()
  list.push(3)
  list.push(2)
  list.push(1)

  println("Before popping list: $list")
  val poppedValue = list.pop()
  println("After popping list: $list")
  println("Popped value: $poppedValue")
}
```

You'll see the following output:

```
---Example of pop---
Before popping list: 1 -> 2 -> 3
After popping list: 2 -> 3
Popped value: 1
```

Remove last operations

Removing the last node of the list is somewhat inconvenient.

Although you have a reference to the `tail` node, you can't chop it off without having a reference to the node before it. Thus, you need to traverse the whole list to find the node before the last.

Add the following code just below `pop()`:

```
fun removeLast(): T? {
  // 1
  val head = head ?: return null
  // 2
  if (head.next == null) return pop()
  // 3
  size--

  // 4
  var prev = head
  var current = head

  var next = current.next
  while (next != null) {
    prev = current
    current = next
    next = current.next
  }
  // 5
  prev.next = null
  tail = prev
  return current.value
}
```

Here's what's happening:

1. If `head` is `null`, there's nothing to remove, so you return `null`.
2. If the list only consists of one node, `removeLast` is functionally

equivalent to `pop`. Since `pop` will handle updating the `head` and `tail` references, you can delegate this work to the `pop` function.

3. At this point, you know that you'll be removing a node, so you update the size of the list accordingly.
4. You keep searching for the next node until `current.next` is `null`. This signifies that `current` is the last node of the list.
5. Since `current` is the last node, you disconnect it using the `prev.next` reference. You also make sure to update the `tail` reference.

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

```
"removing the last node" example {
  val list = LinkedList<Int>()
  list.push(3)
  list.push(2)
  list.push(1)

  println("Before removing last node: $list")
  val removedValue = list.removeLast()

  println("After removing last node: $list")
  println("Removed value: $removedValue")
}
```

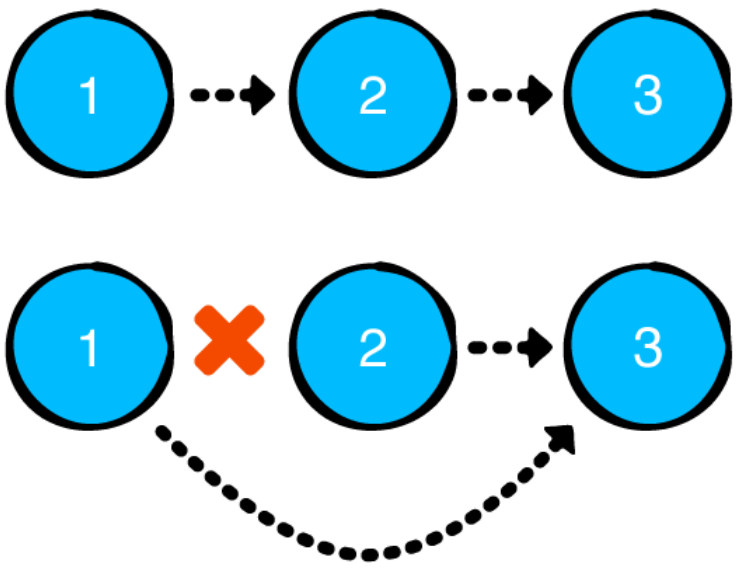
You'll see the following at the bottom of the console:

```
---Example of removing the last node---
Before removing last node: 1 -> 2 -> 3
After removing last node: 1 -> 2
Removed value: 3
```

`removeLast()` requires you to traverse down the list. This makes for an *O(n)* operation, which is relatively expensive.

Remove after operations

The final remove operation is removing a node at a particular point in the list. This is achieved much like `insert()`. You'll first find the node immediately before the node you wish to remove and then unlink it.



Removing the middle node

Navigate back to **LinkedList.kt** and add the following method below `removeLast()`:

```
fun removeAfter(node: Node<T>): T? {
  val result = node.next?.value

  if (node.next == tail) {
    tail = node
  }

  if (node.next != null) {
    size--
  }

  node.next = node.next?.next
  return result
}
```

Special care needs to be taken if the removed node is the tail node since the `tail` reference will need to be updated.

Now, add the following example to `main()` to test `removeAfter()`. You know the drill:

```
"removing a node after a particular node" example {
  val list = LinkedList<Int>()
  list.push(3)
  list.push(2)
  list.push(1)

  println("Before removing at particular index: $list")
  val index = 1
  val node = list.nodeAt(index - 1)!!
  val removedValue = list.removeAfter(node)

  println("After removing at index $index: $list")
  println("Removed value: $removedValue")
}
```

You'll see the following output in the console:

```
---Example of removing a node after a particular node---
Before removing at particular index: 1 -> 2 -> 3
After removing at index 1: 1 -> 3
Removed value: 2
```

Try adding more elements and play around with the value of the index. Similar to `insert()`, the time complexity of this operation is *O*(1), but it requires you to have a reference to a particular node beforehand.

Performance analysis

You've hit another checkpoint. To recap, you've implemented the three operations that remove values from a linked list:

	pop	removeLast	removeAfter
Behaviour	remove at head	remove at tail	remove the immediate next node
Time complexity	O(1)	O(n)	O(1)

At this point, you've defined an interface for a linked list that most programmers around the world can relate to. However, there's work to be done to adorn the Kotlin semantics. In the next half of the chapter, you'll focus on making the interface better by bringing it closer to idiomatic Kotlin.

Kotlin collection interfaces

The Kotlin Standard Library has a set of interfaces that help define what's expected of a particular type. Each of these interfaces provides certain guarantees on characteristics and performance. Of these many interfaces, you'll take a look at four important *collection interfaces*.

Here's what each interface represents:

- **Tier 1, Iterable**: An iterable type provides sequential access to its elements via an `Iterator`.
- **Tier 2, Collection**: A collection is an iterable that provides additional functionality, allowing you to check if the collection contains a particular element or a collection of elements.
- **Tier 3, MutableIterable**: An iterable that provides a `MutableIterator`, which allows both accessing the items and removing them.

- **Tier 4, MutableCollection**: A collection that also provides methods to alter its contained items. For example, you can **add** and **remove** elements, and even **clear** the entire collection.

There's more to be said for each of these interfaces. You'll learn more when you need to conform to them.

A linked list can get to the tier 4 of the collection interfaces. Since a linked list is a chain of nodes, adopting the `Iterable` interface makes sense. And because you've already implemented adding elements and removing them, it's pretty clear we can go all the way to the `MutableCollection` interface.

Becoming a Kotlin mutable collection

In this section, you'll look into implementing the `MutableCollection` interface. A mutable collection type is a finite sequence and provides nondestructive sequential access but also allows you to modify the collection.

Iterating through elements

Reaching Tier 1 means implementing `Iterable` in the `LinkedList`. To make things easier, first make reading the `size` available outside the class. Modify the `size` property in **LinkedList.kt**, so that the property itself is public, but its setter remains private:

```
var size = 0
  private set
```

Then, add the `Iterable` interface to the class definition. The definition will now look like:

```
class LinkedList<T : Any> : Iterable<T> {
  ...
```

```
}
```

This means that you're now required to add the following method to fulfill the `Iterable` interface:

```
override fun iterator(): Iterator<T> {
  return LinkedListIterator(this)
}
```

Right now, the compiler complains because it doesn't know what a `LinkedListIterator` is. Create a class named `LinkedListIterator` and make it implement the `Iterator` interface:

```
class LinkedListIterator<T : Any> : Iterator<T> {
  override fun next(): T {
    TODO("not implemented")
  }
  override fun hasNext(): Boolean {
    TODO("not implemented")
  }
}
```

To iterate through the linked list, you need to have a reference to the list. Add this parameter to the constructor:

```
class LinkedListIterator<T>(
  private val list: LinkedList<T>
) : Iterator<T> {
  ...
}
```

You can now start implementing the required methods, starting with the easier one, `hasNext()`. This method indicates whether your `Iterable` still has values to read. You'll need to keep track of the position that the

iterator has in the collection, so create an `index` property inside the class:

```
private var index = 0
```

Then, you can easily check if the position you're at is less than the total number of nodes:

```
override fun hasNext(): Boolean {
  return index < list.size
}
```

The `next()` method reads the values of your nodes in order, and you can use another property to help you out with its implementation. You'll want to keep track of the last node, so you can easily find the next one:

```
private var lastNode: Node<T>? = null
```

The `next()` function looks like:

```
override fun next(): T {
  // 1
  if (index >= list.size) throw IndexOutOfBoundsException()
  // 2
  lastNode = if (index == 0) {
    list.nodeAt(0)
  } else {
    lastNode?.next
  }
  // 3
  index++
  return lastNode!!.value
}
```

Here are the crucial bits of this function, step-by-step:

1. You check that there are still elements to return. If there aren't, you throw an exception. This should never be the case if clients use the `Iterator` correctly, always checking with `hasNext()` before trying to read a value from it with `next()`.
2. If this is the first iteration, there is no `lastNode` set, so you take the first node of the list. After the first iteration, you can get the `next` property of the last node to find the next one.
3. Since the `lastNode` property was updated, you need to update the `index` too. You've now gone through one more iteration, so the index increments.

Now that you've implemented `Iterator`, you can do some really cool things. For example, you can iterate your linked list with a regular Kotlin `for` loop and print the double of each element in a list.

Add this to `main()`:

```
"printing doubles" example {
  val list = LinkedList<Int>()
  list.push(3)
  list.push(2)
  list.push(1)
  println(list)

  for (item in list) {
    println("Double: ${item * 2}")
  }
}
```

The output you'll get is:

```
---Example of printing doubles---
1 -> 2 -> 3
Double: 2
Double: 4
Double: 6
```

Cool, huh? A `for` loop helps you a lot but there are still things to implement if you want to have a fully featured `MutableCollection`.

Becoming a collection

Being a `Collection` requires more than just being an `Iterable` class. Change the definition of your `LinkedList` to implement `Collection`:

```
class LinkedList<T : Any> : Iterable<T>, Collection<T>  {
  ...
}
```

Of course, you could now remove `Iterable` because a `Collection` is an `Iterable` anyway. You may also leave it there so that you can see the progress you're making.

The compiler will start complaining about the methods you need to implement. Here are some quick wins: You already have `isEmpty()` and `size`, so you can add the `override` keyword in front of them.

```
override var size = 0
  private set

override fun isEmpty(): Boolean {
  return size == 0
}
```

You still need to implement two more methods, but the good news is that you can use one to implement the other easily:

```
override fun contains(element: T): Boolean {
  TODO("not implemented")
}
```

```
override fun containsAll(elements: Collection<T>): Boolean {
  TODO("not implemented")
}
```

Since you can now iterate through the list with `for`, the implementation of `contains` is straightforward:

```
override fun contains(element: T): Boolean {
  for (item in this) {
    if (item == element) return true
  }
  return false
}
```

This method iterates through all elements of the list if needed, so it has a complexity of *O*(*n*).

The second method is similar; it just works with a collection of elements.

```
override fun containsAll(elements: Collection<T>): Boolean {
  for (searched in elements) {
    if (!contains(searched)) return false
  }
  return true
}
```

As you'd probably guess, this is an inefficient method, it's *O*(*n^2*). But if the `Collection` interface requires it, you need to provide it.

Mutating while iterating

To get to the 3rd tier, you need to make `LinkedList` a `MutableIterable`. If you add this interface to the list of interfaces that `LinkedList` implements, you'll see the compiler complain again because `iterator()` doesn't return a `MutableIterator`. This time, start the other way around. Make

`LinkedListIterator` implement the `MutableIterator<T>` interface:

```
class LinkedListIterator<T>(
  private val list: LinkedList<T>
) : Iterator<T>, MutableIterator<T> {
  ...
}
```

Again, `MutableIterator` is a broader interface than `Iterator`, so you can remove `Iterator` from the list of implemented interfaces.

You'll need to add the `remove()` method to comply with the new interface you've added:

```
override fun remove() {
  // 1
  if (index == 1) {
    list.pop()
  } else {
    // 2
    val prevNode = list.nodeAt(index - 2) ?: return
    // 3
    list.removeAfter(prevNode)
    lastNode = prevNode
  }
  index--
}
```

Here's a breakdown of how this code uses the methods `LinkedList` already has:

1. The simplest case is when you're at the beginning of the list. Using `pop()` will do the trick.
2. If the iterator is somewhere inside the list, it needs to find the previous node. That's the only way to remove items from a linked list.
3. The iterator needs to step back so that `next()` returns the correct

method the next time. This means reassigning the `lastNode` and decreasing the index.

Now, go to **LinkedList.kt** and add `MutableIterable` to the class definition:

```
class LinkedList<T : Any>: Iterable<T>, Collection<T>, MutableIterable<T>
  ...
}
```

Modify `iterator()` to return a `MutableIterator`, which `LinkedListIterator` now implements:

```
override fun iterator(): MutableIterator<T> {
  return LinkedListIterator(this)
}
```

That's it, your linked list is now a proper `MutableIterable`.

Final step: Mutable collection

You've already completed the hardest steps, so this last step shouldn't be too bad. First, add the `MutableCollection` interface to the definition of `LinkedList`:

```
class LinkedList<T : Any>: Iterable<T>, Collection<T>, MutableIterable<T>,
  ...
}
```

This will make you add six more methods:

```
override fun add(element: T): Boolean {
  TODO("not implemented")
}

override fun addAll(elements: Collection<T>): Boolean {
```

```
  TODO("not implemented")
}

override fun clear() {
  TODO("not implemented")
}

override fun remove(element: T): Boolean {
  TODO("not implemented")
}

override fun removeAll(elements: Collection<T>): Boolean {
  TODO("not implemented")
}

override fun retainAll(elements: Collection<T>): Boolean {
  TODO("not implemented")
}
```

Three of them are relatively simple to implement. `add()`, `addAll()` and `clear()` are almost one-liners:

```
override fun add(element: T): Boolean {
  append(element)
  return true
}

override fun addAll(elements: Collection<T>): Boolean {
  for (element in elements) {
    append(element)
  }
  return true
}

override fun clear() {
  head = null
  tail = null
  size = 0
}
```

Since the `LinkedList` doesn't have a fixed size, `add()` and `addAll()` are always successful and need to return `true`. Performing `clear()` on a linked list is as easy as dropping all its node references and resetting the size to 0.

For the removal of elements, you'll use a different approach for iterating through your `MutableIterable` linked list. This way, you can benefit from your `MutableIterator`:

```
override fun remove(element: T): Boolean {
  // 1
  val iterator = iterator()
  // 2
  while (iterator.hasNext()) {
    val item = iterator.next()
    // 3
    if (item == element) {
      iterator.remove()
      return true
    }
  }
  // 4
  return false
}
```

This method is a little complex, so here's the step-by-step walkthrough:

1. Get an iterator that will help you iterate through the collection.
2. Create a loop that checks if there are any elements left, and gets the next one.
3. Check if the current element is the one you're looking for, and if it is, remove it.
4. Return a boolean that signals if an element has been removed.

With `removeAll()`, you can make use of `remove()`:

```
override fun removeAll(elements: Collection<T>): Boolean {
  var result = false
  for (item in elements) {
    result = remove(item) || result
  }
  return result
}
```

The return value of `removeAll` is `true` if any elements were removed.

The last method to implement is `retainAll()`, which should remove any elements in the list besides the ones passed in as the parameter. You'll need to approach this the other way around. Iterate through your list once and remove any element that is not in the parameter. Luckily, the parameter of `retainAll` is also a collection, so you can use all of the methods you implemented yourself, like `contains`:

```
override fun retainAll(elements: Collection<T>): Boolean {
  var result = false
  val iterator = this.iterator()
  while (iterator.hasNext()) {
    val item = iterator.next()
    if (!elements.contains(item)) {
      iterator.remove()
      result = true
    }
  }
  return result
}
```

Congrats! You finished the implementation, so it's time for some testing. Go back into **Main.kt** and add these at the end of `main()`:

```
"removing elements" example {
  val list: MutableCollection<Int> = LinkedList()
  list.add(3)
  list.add(2)
```

```
  list.add(1)

  println(list)
  list.remove(1)
  println(list)
}

"retaining elements" example {
  val list: MutableCollection<Int> = LinkedList()
  list.add(3)
  list.add(2)
  list.add(1)
  list.add(4)
  list.add(5)

  println(list)
  list.retainAll(listOf(3, 4, 5))
  println(list)
}

"remove all elements" example {
  val list: MutableCollection<Int> = LinkedList()
  list.add(3)
  list.add(2)
  list.add(1)
  list.add(4)
  list.add(5)

  println(list)
  list.removeAll(listOf(3, 4, 5))
  println(list)
}
```

As you'll see momentarily, your first challenge in this chapter is to check the output of each example to make sure it's correct. You'll also see the rest of the challenges after a quick summary.

Challenges

In these challenges, you'll work through five common scenarios for the linked list. These problems are relatively easy compared to most challenges, and they will serve to solidify your knowledge of data structures. You'll find the solutions to the challenges at the end of this chapter.

Challenge 1: Reverse a linked list

Create an extension function that prints out the elements of a linked list in reverse order. Given a linked list, print the nodes in reverse order. For example:

```
1 -> 2 -> 3

// should print out the following:
3
2
1
```

Solution 1

A straightforward way to solve this problem is to use recursion. Since recursion allows you to build a call stack, you need to call the `print` statements as the call stack unwinds.

Your first task is to define an extension function for `LinkedList`. Add the following helper function to your solution file:

```
fun <T : Any> LinkedList<T>.printInReverse() {
  this.nodeAt(0)?.printInReverse()
}
```

This function forwards the call to the recursive function that traverses the list, node by node. To traverse the list, add this extension function for `Node`:

```
fun <T : Any> Node<T>.printInReverse() {
  this.next?.printInReverse()
}
```

As you'd expect, this function calls itself on the next node. The terminating condition is somewhat hidden in the null-safety operator. If the value of `next` is `null`, the function stops because there's no next node on which to call `printInReverse()`. You're almost done; the next step is printing the nodes.

Printing

Where you add the `print` statement will determine whether you print the list in reverse order or not. Update the function to the following:

```
fun <T : Any> Node<T>.printInReverse() {
  this.next?.printInReverse()
  // 1
  if (this.next != null) {
    print(" <- ")
  }
  // 2
  print(this.value.toString())
}
```

Any code that comes after the recursive call is called only after the base case triggers (i.e., after the recursive function hits the end of the list).

1. First, you check if you've reached the end of the list. That's the beginning of the reverse printing, and you'll not add an arrow there. The arrows start with the second element of the reverse output. This is just for pretty formatting.

2. As the recursive statements unravel, the node data gets printed.

Test it out!

Write the following at the bottom of `main()`:

```
"print in reverse" example {
  val list = LinkedList<Int>()
  list.add(3)
  list.add(2)
  list.add(1)
  list.add(4)
  list.add(5)

  println(list)
  list.printInReverse()
}
```

You'll see the following output:

```
---Example of print in reverse---
3 -> 2 -> 1 -> 4 -> 5
5 <- 4 <- 1 <- 2 <- 3
```

The time complexity of this algorithm is ***O*(*n*)** since you have to traverse each node of the list.

Challenge 2: The item in the middle

Given a linked list, find the middle node of the list. For example:

```
1 -> 2 -> 3 -> 4
// middle is 3

1 -> 2 -> 3
// middle is 2
```

Solution 2

One solution is to have two references traverse down the nodes of the list

where one is twice as fast as the other. Once the faster reference reaches the end, the slower reference will be in the middle. Write the following function:

```
fun <T : Any> LinkedList<T>.getMiddle(): Node<T>? {
  var slow = this.nodeAt(0)
  var fast = this.nodeAt(0)

  while (fast != null) {
    fast = fast.next
    if (fast != null) {
      fast = fast.next
      slow = slow?.next
    }
  }

  return slow
}
```

In the `while` declaration, you bind the next node to `fast`. If there's a next node, you update `fast` to the next node of `fast`, effectively stepping down the list twice. `slow` is updated only once. This is also known as the **runner technique**.

Try it out!

Write the following at the bottom of **Main.kt**:

```
"print middle" example {
  val list = LinkedList<Int>()
  list.add(3)
  list.add(2)
  list.add(1)
  list.add(4)
  list.add(5)

  println(list)
```

```
  println(list.getMiddle()?.value)
}
```

You'll see the following output:

```
---Example of print middle---
3 -> 2 -> 1 -> 4 -> 5
1
```

The time complexity of this algorithm is ***O*(*n*)** since you traversed the list in a single pass. The **runner technique** helps solve a variety of problems associated with the linked list.

Challenge 3: Reverse a linked list

To reverse a list is to manipulate the nodes so that the nodes of the list are linked in the opposite direction. For example:

```
// before
1 -> 2 -> 3

// after
3 -> 2 -> 1
```

Solution 3

To reverse a linked list, you need to visit each node and update the `next` reference to point in the other direction. This can be a tricky task since you'll need to manage multiple references to multiple nodes. To do this, you would also need access to the `head` and `tail` of your linked list. Since you're implementing an extension function, you won't have access to these variables. Luckily, there's a simpler solution that has a small drawback discussed later.

You can easily reverse a list by using a recursive function that goes to the

end of the list and then starts copying the nodes when it returns, into a new linked list. Here's how this function would look like:

```
private fun <T : Any> addInReverse(list: LinkedList<T>, node: Node<T>) {
  // 1
  val next = node.next
  if (next != null) {
    // 2
    addInReverse(list, next)
  }
  // 3
  list.append(node.value)
}
```

The following explains how this function works:

1. Get the next node of the list, starting from the one you've received as a parameter.
2. If there's a following node, recursively call the same function; however, now the starting node is the one after the current node.
3. When you reach the end, start adding the nodes in the reverse order.

***O*(*n*)** time complexity, short and sweet! The only drawback is that you need a new list, which means that the space complexity is also ***O*(*n*)**.

To use this helper function conveniently on a `LinkedList`, add this extension function:

```
fun <T : Any> LinkedList<T>.reversed(): LinkedList<T> {
  val result = LinkedList<T>()
  val head = this.nodeAt(0)
  if (head != null) {
    addInReverse(result, head)
  }
  return result
}
```

This extension creates a new `LinkedList` and fills it with nodes by calling `addInReverse()`, passing in the first node of the current list.

Try it out!

Test `reversed()` by writing the following at the bottom of `main()`:

```
"reverse list" example {
  val list = LinkedList<Int>()
  list.add(3)
  list.add(2)
  list.add(1)
  list.add(4)
  list.add(5)

  println("Original: $list")
  println("Reversed: ${list.reversed()}")
}
```

You'll see the following output:

```
---Example of reverse list---
Original: 3 -> 2 -> 1 -> 4 -> 5
Reversed: 5 -> 4 -> 1 -> 2 -> 3
```

Challenge 4: Merging two linked lists

Create a function that takes two sorted linked lists and merges them into a single sorted linked list.

Your goal is to return a new linked list that contains all the nodes from two lists in sorted order. You may assume they are both lists are sorted in ascending order. For example:

```
// list1
```

```
1 -> 4 -> 10 -> 11

// list2
-1 -> 2 -> 3 -> 6

// merged list
-1 -> 1 -> 2 -> 3 -> 4 -> 6 -> 10 -> 11
```

Since you need to compare values for this challenge, you can assume that `T : Comparable<T>`, meaning that the type parameter of the lists in question implements the `Comparable` interface.

Solution 4

The solution to this problem is to continuously pluck nodes from the two sorted lists and add them to a new list. Since the two lists are sorted, you can compare the `next` node of both lists to see which one should be the next one to add to the new list.

Setting up

You'll begin by checking the cases where one or both of the lists are empty. Create the following `mergeSorted` extension function:

```
fun <T : Comparable<T>> LinkedList<T>.mergeSorted(
  otherList: LinkedList<T>
): LinkedList<T> {
  if (this.isEmpty()) return otherList
  if (otherList.isEmpty()) return this

  val result = LinkedList<T>()

  return result
}
```

If one is empty, you return the other. You also introduce a new reference to hold a new `LinkedList`. The strategy is to merge the nodes in `this` and

`otherList` into `result` in sorted order.

Next, you need to write a helper function that adds the current node to the result list and returns the next node. You'll use this function in your algorithm multiple times, so it's useful to have it extracted:

```
private fun <T : Comparable<T>> append(
  result: LinkedList<T>,
  node: Node<T>
): Node<T>? {
  result.append(node.value)
  return node.next
}
```

Merging

Add the following to `mergeSorted` immediately below the declaration for `result` and right above `return result`:

```
// 1
var left = nodeAt(0)
var right = otherList.nodeAt(0)
// 2
while (left != null && right != null) {
  // 3
  if (left.value < right.value) {
    left = append(result, left)
  } else {
    right = append(result, right)
  }
}
```

Here's how it works:

1. You start with the first node of each list.
2. The `while` loop continues until one of the lists reaches its end.
3. You compare the first nodes `left` and `right` to append to the `result`.

Since this loop depends on both `left` and `right`, it will terminate even if there are nodes left in either list.

Finally, add the following below the newly added code, and above `return result`, to handle the remaining nodes:

```
while (left != null) {
  left = append(result, left)
}

while (right != null) {
  right = append(result, right)
}
```

Try it out!

Write the following at the bottom of `main()`:

```
"merge lists" example {
  val list = LinkedList<Int>()
  list.add(1)
  list.add(2)
  list.add(3)
  list.add(4)
  list.add(5)

  val other = LinkedList<Int>()
  other.add(-1)
  other.add(0)
  other.add(2)
  other.add(2)
  other.add(7)

  println("Left: $list")
  println("Right: $other")
  println("Merged: ${list.mergeSorted(other)}")
}
```

You'll see the following output:

```
---Example of merge lists---
Left: 1 -> 2 -> 3 -> 4 -> 5
Right: -1 -> 0 -> 2 -> 2 -> 7
Merged: -1 -> 0 -> 1 -> 2 -> 2 -> 2 -> 3 -> 4 -> 5 -> 7
```

This algorithm has a time complexity of ***O*(*m* + *n*)**, where ***m*** is the # of nodes in the first list, and ***n*** is the # of nodes in the second list.

Key points

- Linked lists are linear and unidirectional. As soon as you move a reference from one node to another, you can't go back.
- Linked lists have a *O*(*1*) time complexity for head-first insertions. Arrays have *O*(*n*) time complexity for head-first insertions.
- Conforming to Kotlin collection interfaces, such as `Iterable` and `Collection`, offers a host of helpful methods for a reasonably small amount of requirements.

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.