5 Queues Written by Márton Braun

We're all familiar with waiting in line. Whether you're in line to buy tickets to your favorite movie or waiting for a printer to print a file, these real-life scenarios mimic the **queue** data structure.

Queues use **FIFO** or **first in, first out** ordering, meaning the first element that was added will always be the first one removed. Queues are handy when you need to maintain the order of your elements to process later.

In this chapter, you'll learn all of the common operations of a queue, go over the various ways to implement a queue and look at the time complexity of each approach.

Common operations

First, establish an interface for queues. In the **base** package, create a file named **Queue.kt** and add the following code defining the `Queue` interface.

```
interface Queue<T : Any> {

  fun enqueue(element: T): Boolean

  fun dequeue(): T?

  val count: Int
    get

  val isEmpty: Boolean
    get() = count == 0

  fun peek(): T?
}
```

This will be your starting point. From now on, everything you implement will obey the contract of this interface, which describes the core operations for a queue.

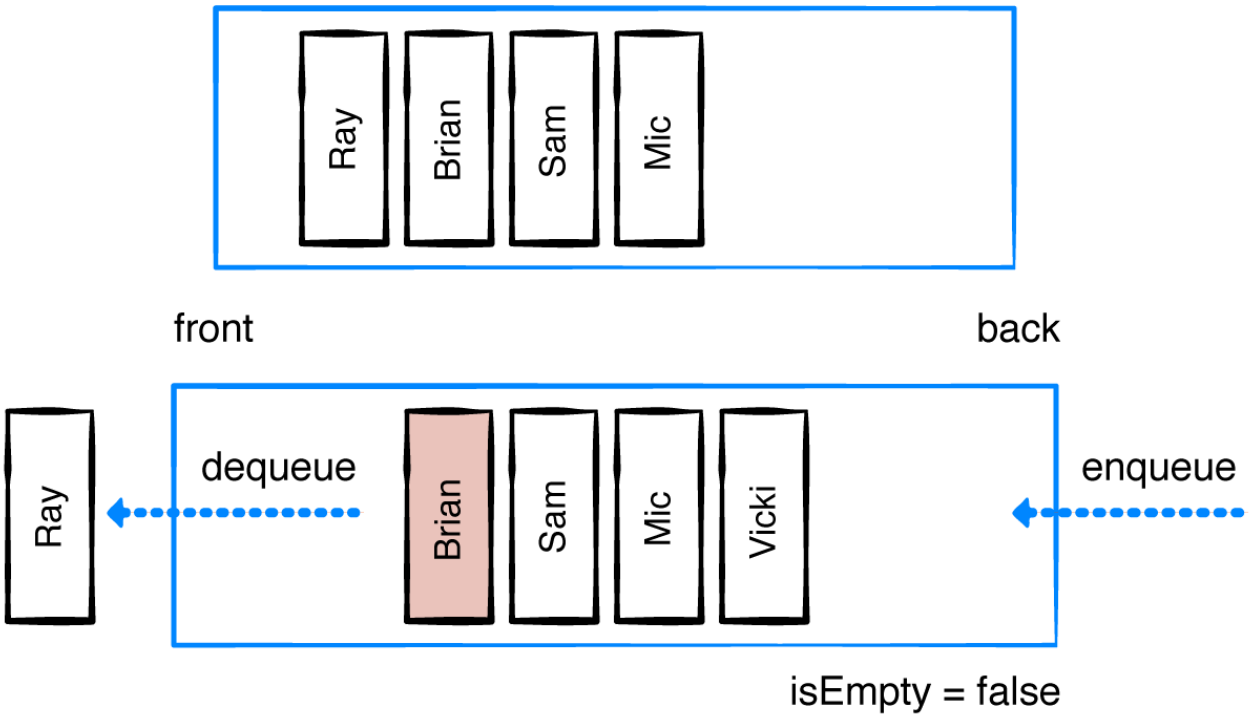
The core operations for a queue are:

- **`enqueue`**: Inserts an element at the back of the queue and returns `true` if the operation is successful.
- **`dequeue`**: Removes the element at the front of the queue and returns it.
- **`isEmpty`**: Checks if the queue is empty using the **`count`** property.
- **`peek`**: Returns the element at the front of the queue *without* removing it.

Notice that the queue only cares about removal from the front and insertion at the back. You don't need to know what the contents are in between. If you did, you'd presumably use an array instead of a Queue.

Example of a queue

The easiest way to understand how a queue works is to look at a working example. Imagine a group of people waiting in line for a movie ticket.



This queue currently holds Ray, Brian, Sam and Mic. Once Ray receives his ticket, he moves out of the line. When you call `dequeue()`, Ray is removed from the *front* of the queue.

Calling `peek()` returns Brian since he’s now at the front of the line.

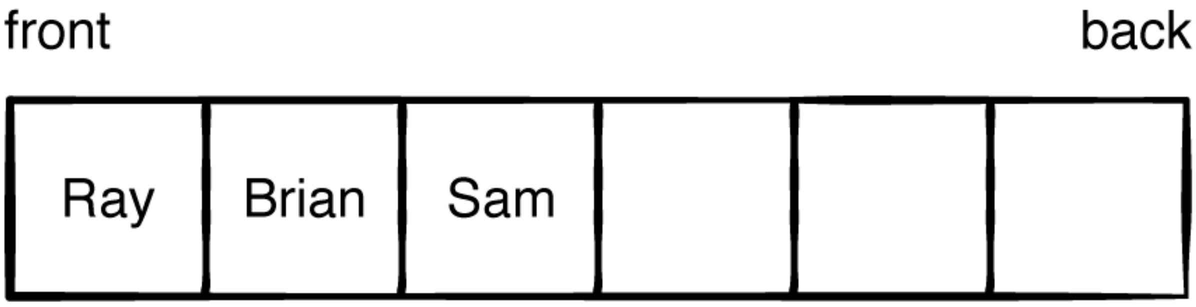
Now comes Vicki, who just joined the line to buy a ticket. When you call `enqueue("Vicki")`, Vicki gets added to the *back* of the queue.

In the following sections, you’ll learn to create a queue in four different ways:

- Using an array based list
- Using a doubly linked list
- Using a ring buffer
- Using two stacks

List-based implementation

The Kotlin standard library comes with a core set of highly optimized data structures that you can use to build higher-level abstractions. One of these is the `ArrayList`, a data structure that stores a contiguous, ordered list of elements. In this section, you’ll use an `ArrayList` to create a queue.



A simple Kotlin `ArrayList` can be used to model the queue.

Open the starter project. In the **list** package, create a file named **ArrayListQueue.kt** and add the following:

```
class ArrayListQueue<T : Any> : Queue<T> {
  private val list = arrayListOf<T>()
}
```

Here, you defined a generic `ArrayListQueue` class that implements the

`Queue` interface. Note that the interface implementation uses the same generic type `T` for the elements it stores.

Next, you'll complete the implementation of `ArrayListQueue` to fulfill the `Queue` contract.

Leveraging ArrayList

Add the following code to `ArrayListQueue`:

```
override val count: Int
  get() = list.size

override fun peek(): T? = list.getOrNull(0)
```

Using the features of `ArrayList`, you get the following for free:

1. Get the size of the queue using the same property of the list.
2. Return the element at the front of the queue, if there is any.

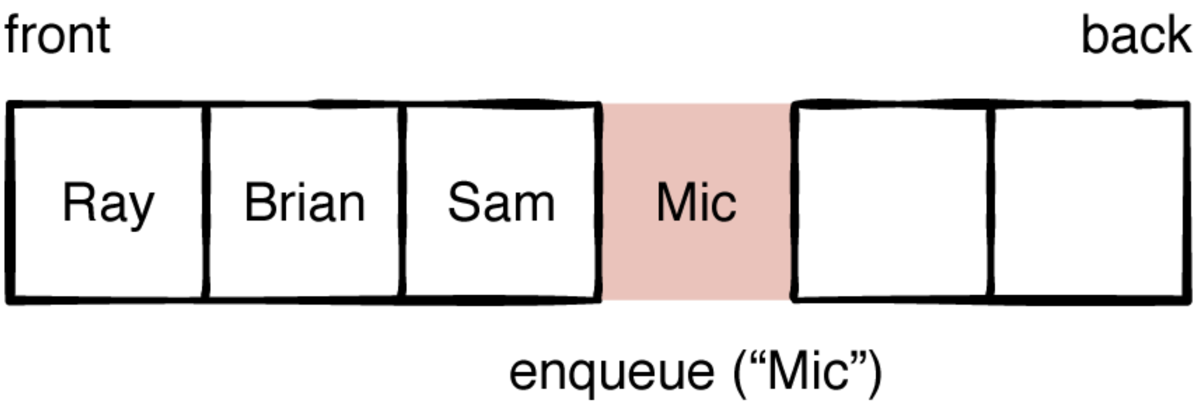
These operations are all *O*(1).

Enqueue

Adding an element to the back of the queue is easy. You simply add the element to the end of the `ArrayList`. Add the following:

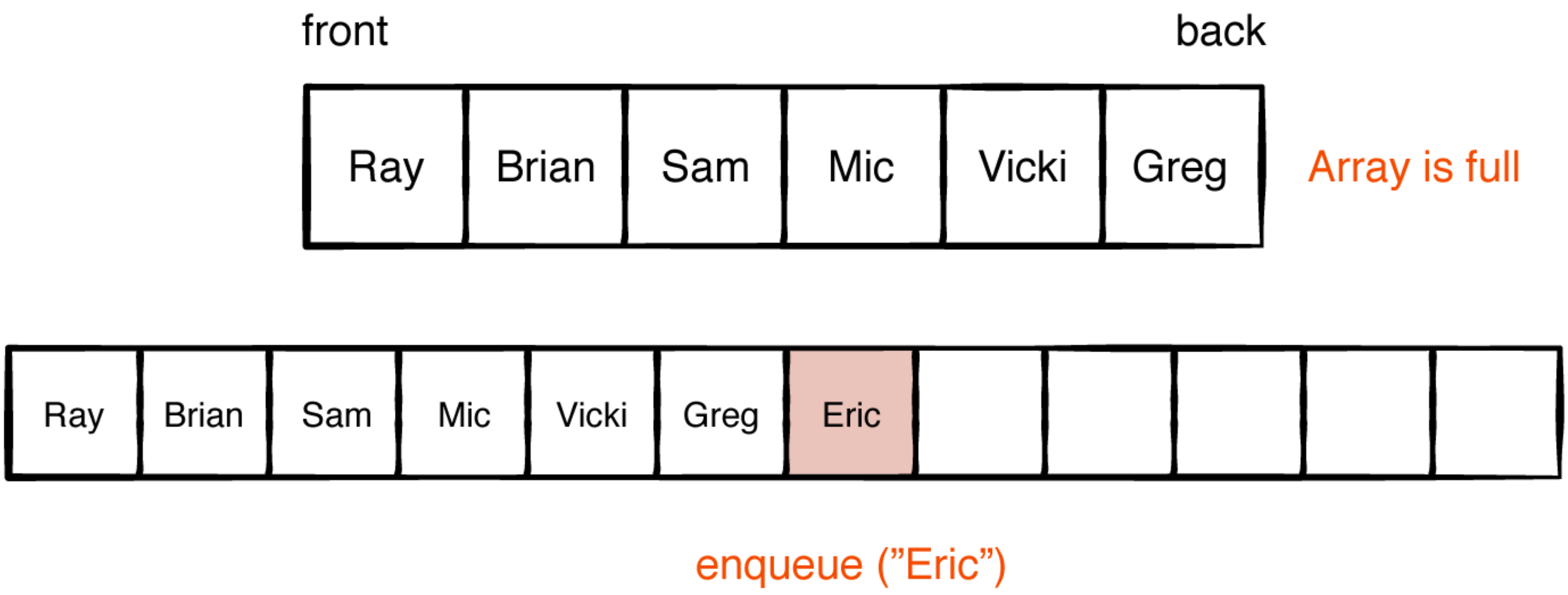
```
override fun enqueue(element: T): Boolean {
  list.add(element)
  return true
}
```

Regardless of the size of the list, enqueueing an element is an *O*(1) operation. This is because the list has empty space at the back.



In the example above, notice that once you add Mic, the list has two empty spaces.

After adding multiple elements, the internal array of the `ArrayList` will eventually be full. When you want to use more than the allocated space, the array must resize to make additional room.



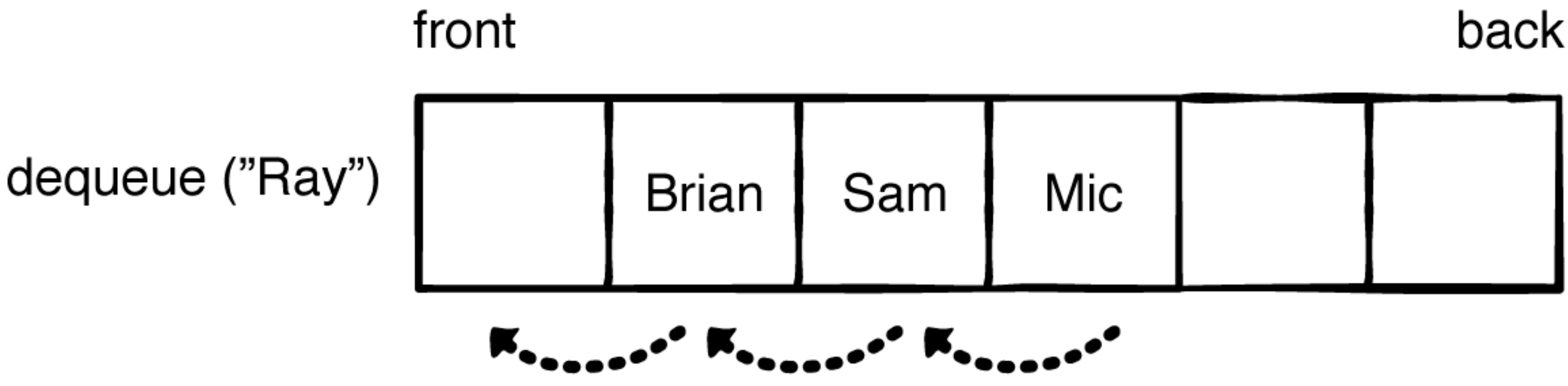
Resizing is an $O(n)$ operation. Resizing requires the list to allocate new memory and copy all existing data over to the new list. Since this doesn’t happen very often (thanks to doubling the size each time), the complexity still works out to be an amortized $O(1)$.

Dequeue

Removing an item from the front requires a bit more work. Add the following:

```
override fun dequeue(): T? =
  if (isEmpty) null else list.removeAt(0)
```

If the queue is empty, `dequeue()` simply returns `null`. If not, it removes the element from the front of the list and returns it.



Removing an element from the front of the queue is an *O*(*n*) operation. To dequeue, you remove the element from the beginning of the list. This is always a linear time operation because it requires all of the remaining elements in the list to be shifted in memory.

Debug and test

For debugging purposes, you'll have your `queue` override `toString()`. Add the following at the bottom of the class:

```
override fun toString(): String = list.toString()
```

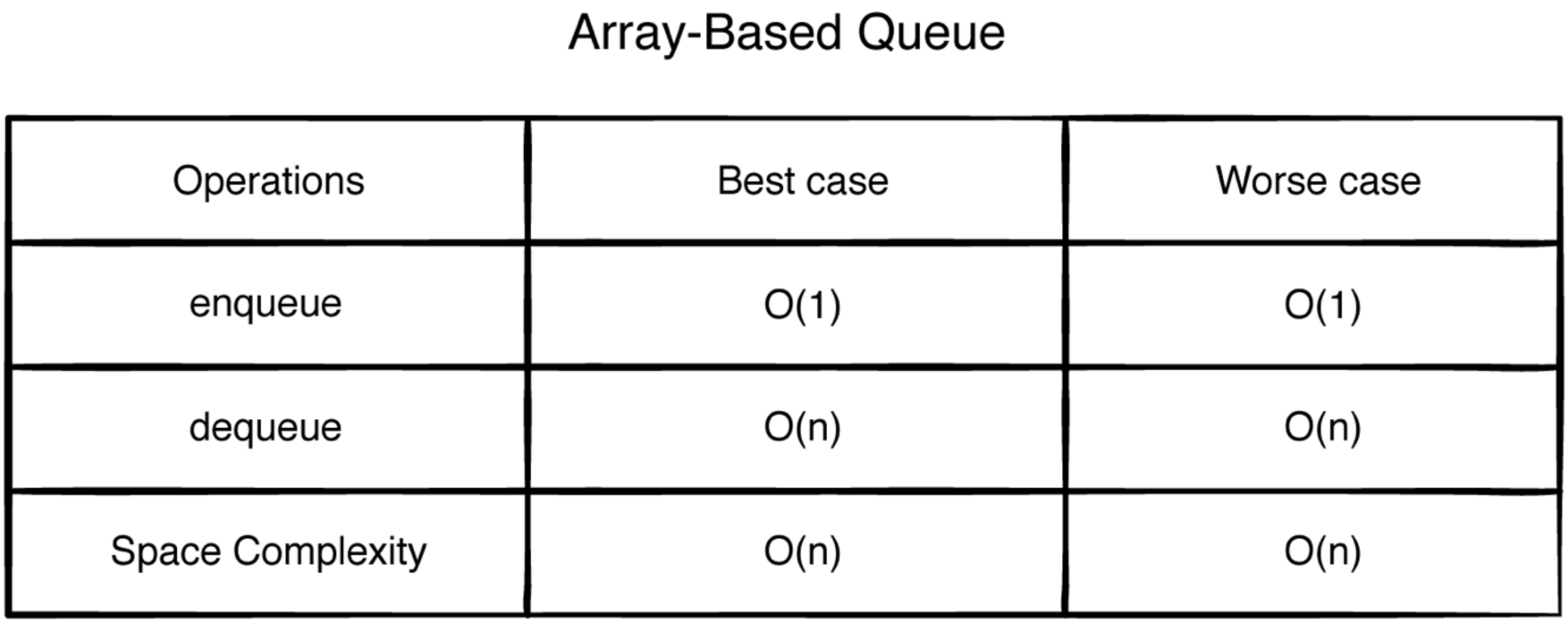
It's time to try out the queue that you just implemented. In **Main.kt**, add the following to the bottom of `main()`:

```
"Queue with ArrayList" example {
  val queue = ArrayListQueue<String>().apply {
    enqueue("Ray")
    enqueue("Brian")
    enqueue("Eric")
  }
  println(queue)
  queue.dequeue()
  println(queue)
  println("Next up: ${queue.peek()}")
}
```

This code puts Ray, Brian and Eric in the queue. It then removes Ray and peeks at Brian, but it doesn’t remove him.

Strengths and weaknesses

Here’s a summary of the algorithmic and storage complexity of the `ArrayList`-based queue implementation. Most of the operations are constant time except for `dequeue()`, which takes linear time. Storage space is also linear.



Array-Based Queue

Operations	Best case	Worse case
enqueue	O(1)	O(1)
dequeue	O(n)	O(n)
Space Complexity	O(n)	O(n)

You’ve seen how easy it is to implement a list-based queue by leveraging a Kotlin `ArrayList`. Enqueue is very fast, thanks to an *O*(1) append operation.

There are some shortcomings to the implementation. Removing an item from the front of the queue can be inefficient, as removal causes all elements to shift up by one. This makes a difference for very large queues. Once the list gets full, it has to resize and may have unused space. This could increase your memory footprint over time. Is it possible to address these shortcomings? Let's look at a linked list-based implementation and compare it to an `ArrayListQueue`.

Doubly linked list implementation

Create a new file named **LinkedListQueue.kt** in the **linkedlist** package. In this package, you'll notice a `DoublyLinkedList` class. You should already be familiar with linked lists from Chapter 3, "Linked Lists". A doubly linked list is simply a linked list in which nodes also contain a reference to the previous node.

Start by adding a generic `LinkedListQueue` to the same package, with the following content:

```
class LinkedListQueue<T : Any> : Queue<T> {
  private val list = DoublyLinkedList<T>()

  private var size = 0

  override val count: Int
    get() = size
}
```

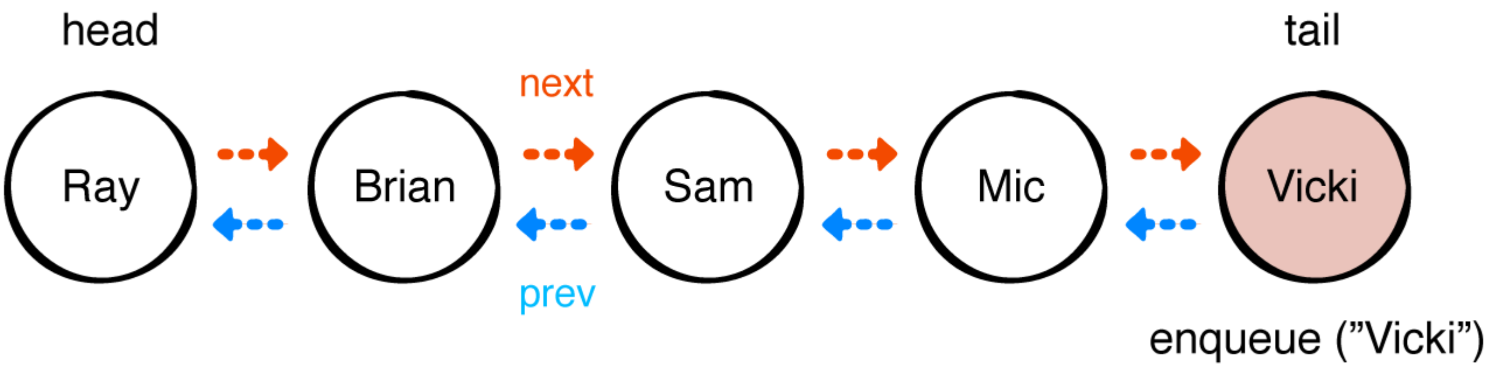
This implementation is similar to `ArrayListQueue`, but instead of an `ArrayList`, you create a `DoublyLinkedList`.

Next, you'll start implementing the `Queue` interface starting from the `count` property that the `DoublyLinkedList` doesn't provide.

Enqueue

To add an element to the back of the queue, add the following:

```
override fun enqueue(element: T): Boolean {
  list.append(element)
  size++
  return true
}
```



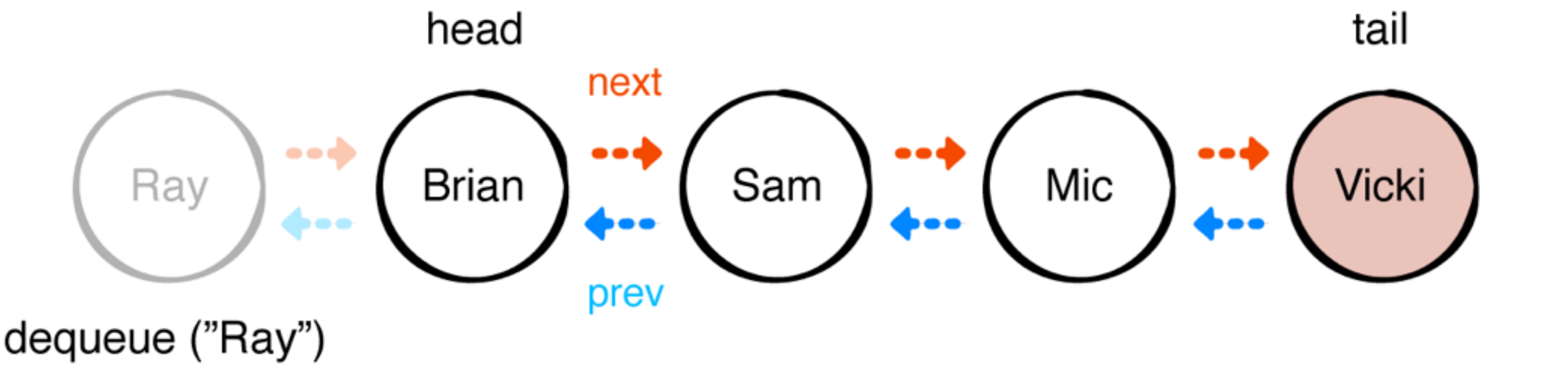
Behind the scenes, the doubly linked list will update its tail node's previous and next references to the new node. You also increment the size. This is an *O*(1) operation.

Dequeue

To remove an element from the queue, add the following:

```
override fun dequeue(): T? {
  val firstNode = list.first ?: return null
  size--
  return list.remove(firstNode)
}
```

This code checks to see if the first element of the queue exists. If it doesn't, it returns `null`. Otherwise, it removes and returns the element at the front of the queue. In this case it also decrements the size.



Removing from the front of the list is also an *O*(1) operation. Compared to the `ArrayList` implementation, you didn't have to shift elements one by one. Instead, in the diagram above, you simply update the `next` and `previous` pointers between the first two nodes of the linked list.

Checking the state of a queue

Similar to the `ArrayList` based implementation, you can implement `peek()` using the properties of the `DoublyLinkedList`. Add the following:

```
override fun peek(): T? = list.first?.value
```

Debug and test

For debugging purposes, add the following at the bottom of the class:

```
override fun toString(): String = list.toString()
```

This leverages the `DoublyLinkedList`'s existing implementation for `toString()`.

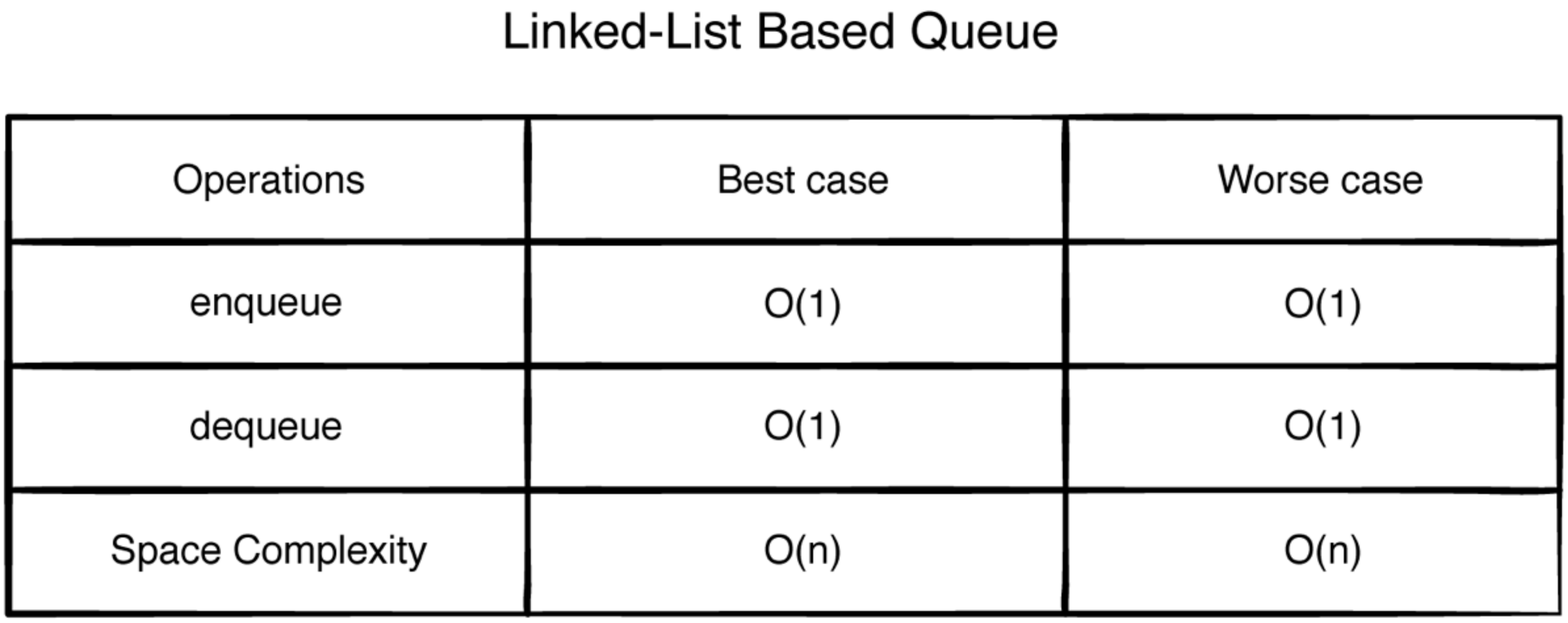
That's all there is to implementing a queue using a linked list. In **Main.kt**, you can try the following example:

```
"Queue with Doubly Linked List" example {
  val queue = LinkedListQueue<String>().apply {
    enqueue("Ray")
    enqueue("Brian")
    enqueue("Eric")
  }
  println(queue)
  queue.dequeue()
  println(queue)
  println("Next up: ${queue.peek()}")
}
```

This test code yields the same results as your `ArrayListQueue` implementation.

Strengths and weaknesses

Time to summarize the algorithmic and storage complexity of the implementation based on a doubly linked list.



Linked-List Based Queue

Operations	Best case	Worse case
enqueue	O(1)	O(1)
dequeue	O(1)	O(1)
Space Complexity	O(n)	O(n)

One of the main problems with `ArrayListQueue` is that dequeuing an item takes linear time. With the linked list implementation, you reduced it to a constant operation, *O*(1). All you needed to do was update the node's `previous` and `next` pointers.

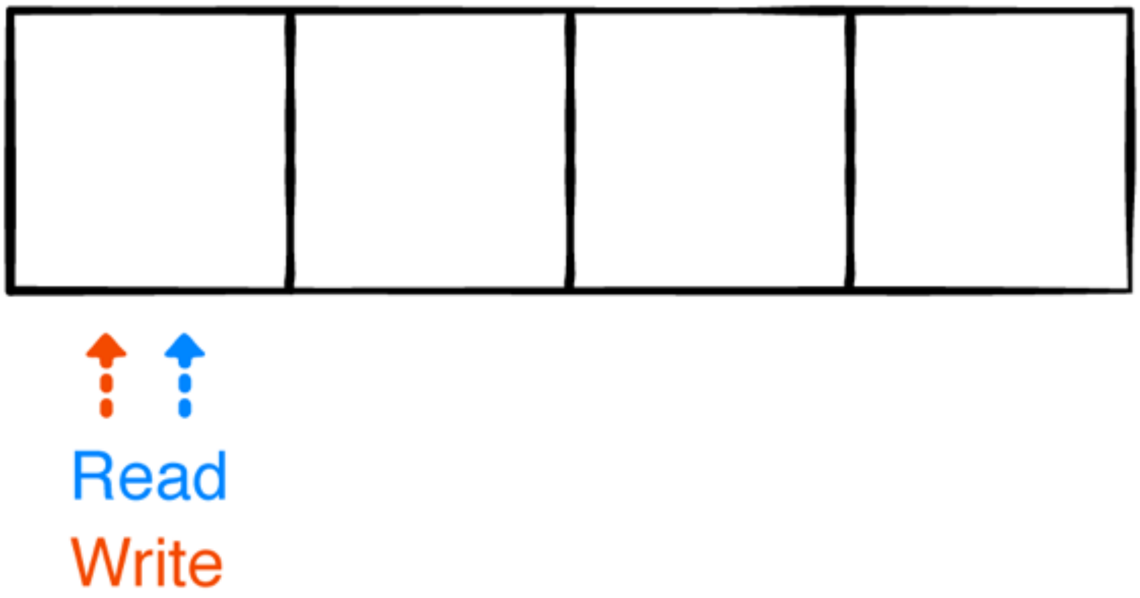
The main weakness with `LinkedListQueue` is not apparent from the table. Despite *O*(1) performance, it suffers from high overhead. Each element has to have extra storage for the forward and back reference. Moreover, every time you create a new element, it requires a relatively expensive dynamic allocation. By contrast, `ArrayListQueue` does bulk allocation, which is faster.

Can you eliminate allocation overhead and preserve *O*(1) dequeues? If you don't have to worry about your queue ever growing beyond a fixed size, you can use a different approach like the **ring buffer**. For example, you might have a game of *Monopoly* with five players. You can use a queue based on a ring buffer to keep track of whose turn is coming up next. You'll take a look at a ring buffer implementation next.

Ring buffer implementation

A ring buffer, also known as a **circular buffer**, is a fixed-size array. This data structure strategically wraps around to the beginning when there are no more items to remove at the end.

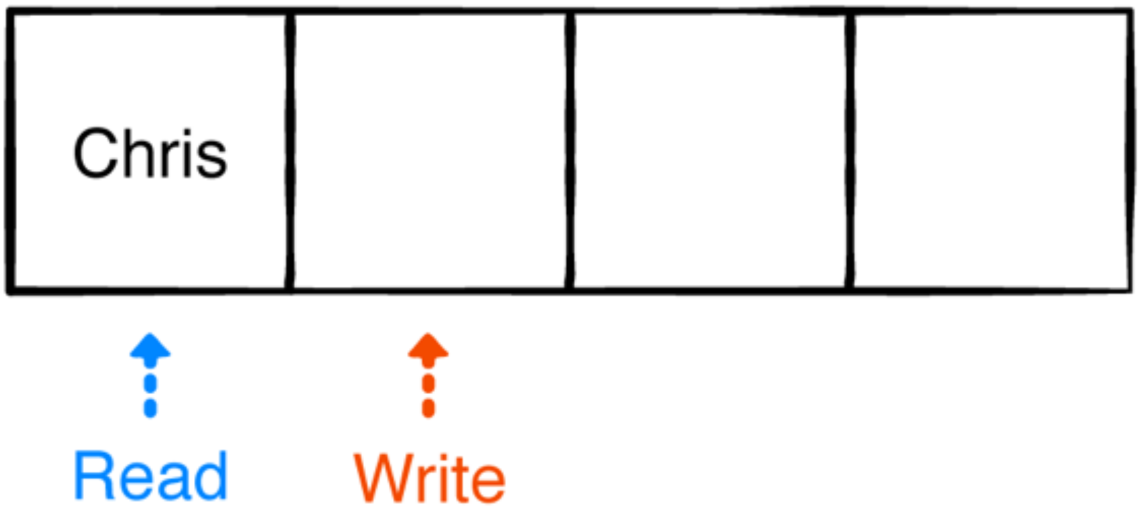
Going over a simple example of how a queue can be implemented using a ring buffer:



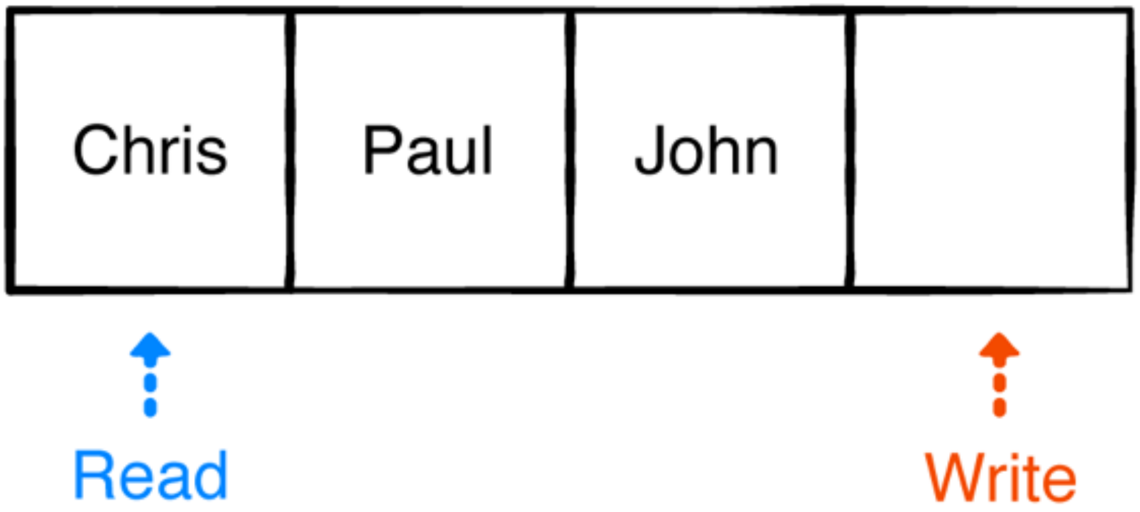
You first create a ring buffer that has a fixed size of **4**. The ring buffer has two "pointers" that keep track of two things:

1. The **read** pointer keeps track of the front of the queue.
2. The **write** pointer keeps track of the next available slot so that you can override existing elements that have already been read.

Enqueue an item:

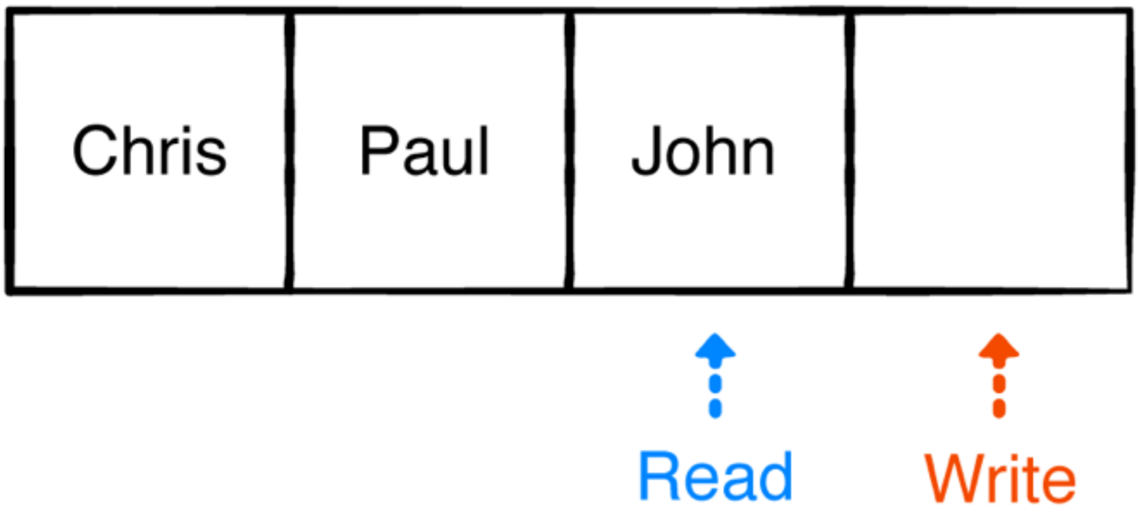


Each time that you add an item to the queue, the **write** pointer increments by one. Add a few more elements:



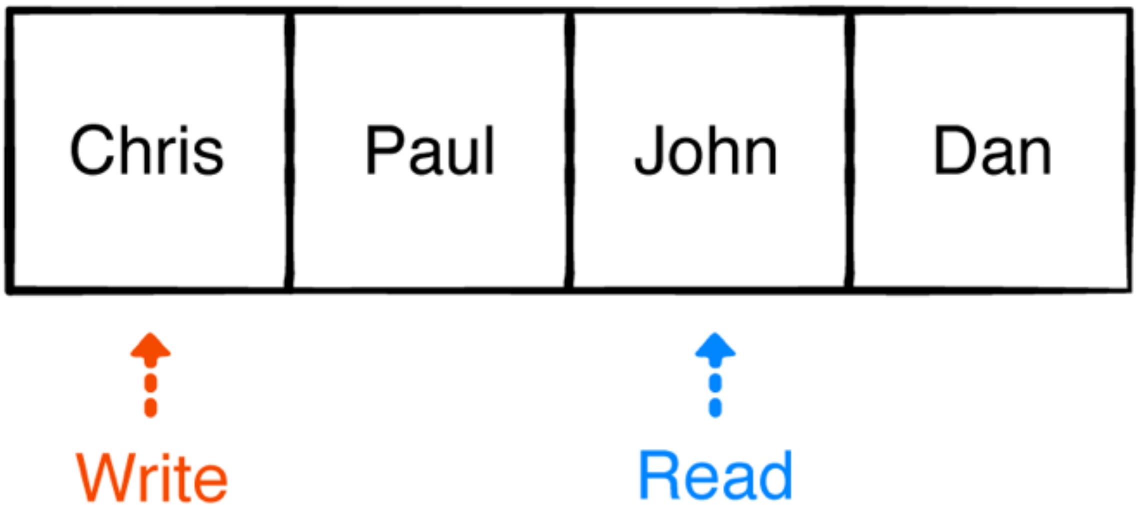
Notice that the **write** pointer moved two more spots and is ahead of the **read** pointer. This means that the queue is not empty.

Next, dequeue two items:



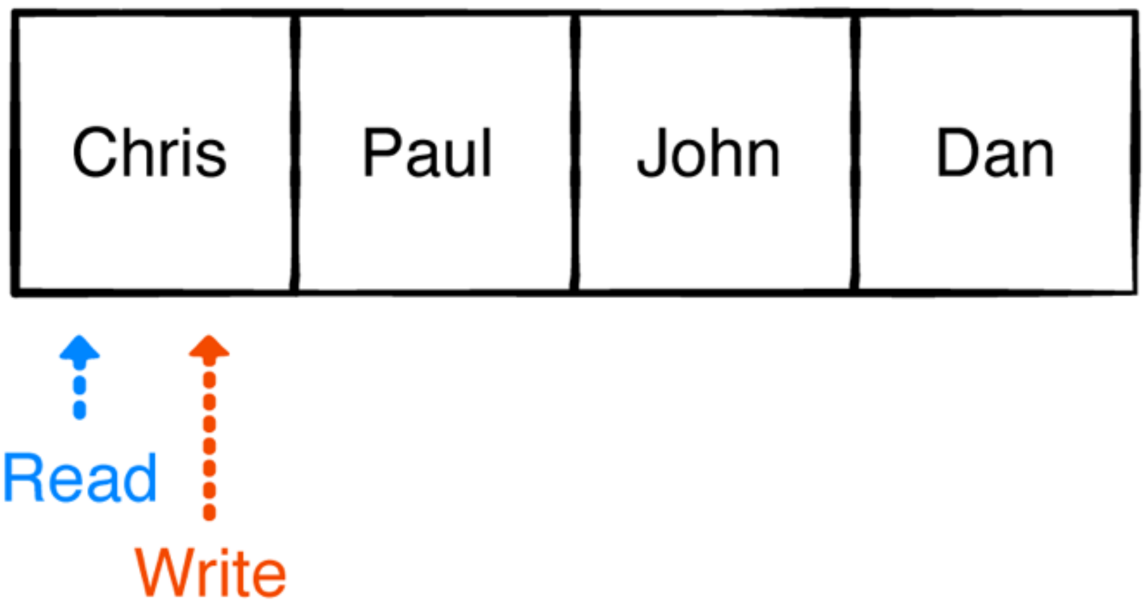
Dequeuing is the equivalent of reading a ring buffer. Notice how the **read** pointer moved twice.

Now, enqueue one more item to fill up the queue:



Since the **write** pointer reached the end, it simply wraps around to the starting index again.

Finally, dequeue the two remaining items:



The **read** pointer wraps to the beginning, as well.

As a final observation, notice that whenever the read and write pointers are at the same index, that means the queue is **empty**.

Now that you have a better understanding of how ring buffers make a queue, you're ready to implement one!

Go to the **ringbuffer** package and create a file named **RingBufferQueue.kt**. You'll notice a `RingBuffer` class inside this package, which you can look at to understand its internal mechanics.

In **RingBufferQueue.kt**, add the following:

```
class RingBufferQueue<T : Any>(size: Int) : Queue<T> {
  private val ringBuffer: RingBuffer<T> = RingBuffer(size)

  override val count: Int
    get() = ringBuffer.count

  override fun peek(): T? = ringBuffer.first
}
```

Here, you define a generic `RingBufferQueue`. Note that you must include a `size` parameter since the ring buffer has a fixed size.

To implement the `Queue` interface, you also need to implement `peek()` and the `count` property using the same from the `RingBuffer` class. Once you provide the `count` property, the `isEmpty` property is already defined in the `Queue` interface. Instead of exposing `ringBuffer`, you provide these helpers to access the front of the queue and to check if the queue is empty. Both of these are *O*(1) operations.

Enqueue

Next, add the following method at the end of the `RingBufferQueue` class:

```
override fun enqueue(element: T): Boolean =
  ringBuffer.write(element)
```

To append an element to the queue, you call `write()` on the `ringBuffer`. This increments the `write` pointer by one.

Since the queue has a fixed size, you must now return `true` or `false` to indicate whether the element has been successfully added. `enqueue()` is still an *O*(1) operation.

Dequeue

To remove an item from the front of the queue, add the following:

```
override fun dequeue(): T? =
  if (isEmpty) null else ringBuffer.read()
```

This code checks if the queue is empty and, if so, returns `null`. If not, it returns an item from the front of the buffer. Behind the scenes, the ring buffer increments the `read` pointer by one.

Debug and test

To easily see the contents of your buffer during debugging, add the following to `RingBufferQueue`:

```
override fun toString(): String = ringBuffer.toString()
```

This code creates a string representation of the queue by delegating to the underlying ring buffer.

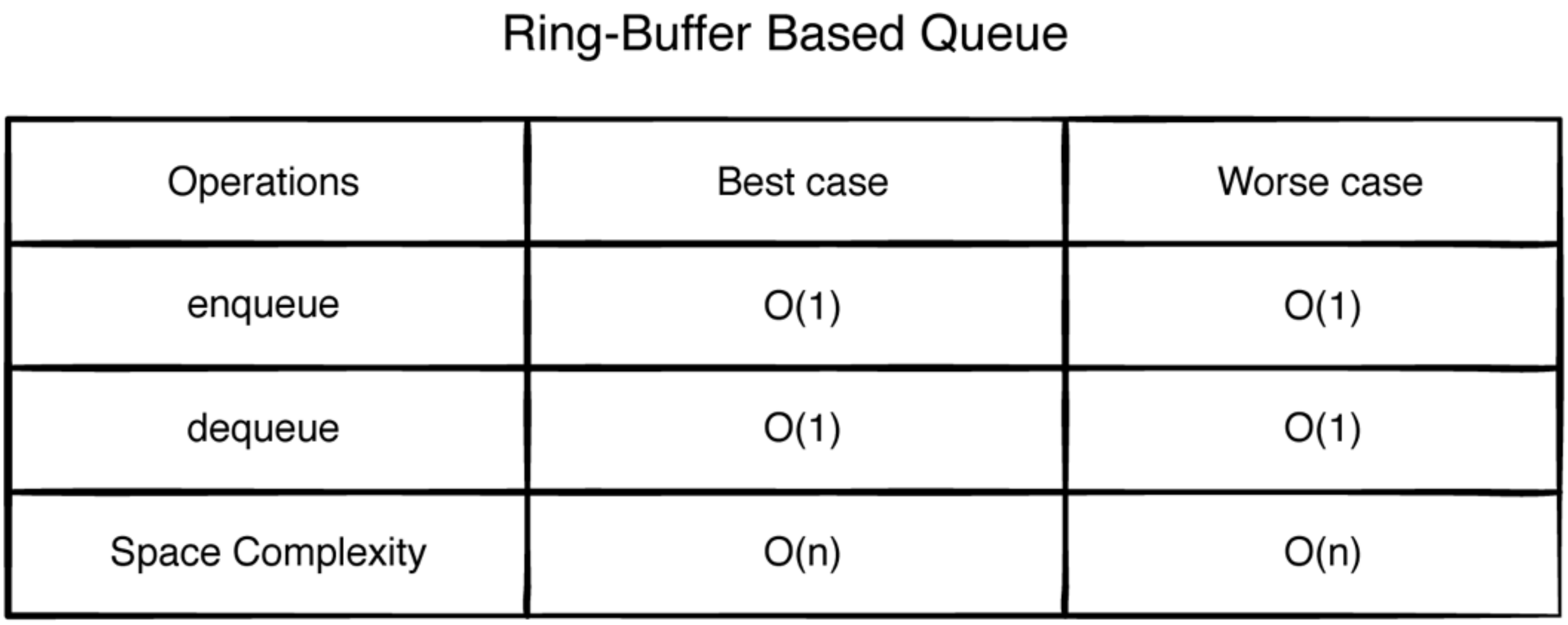
That's all there is to it. Test your ring buffer-based queue by adding the following at the bottom of **Main.kt**, inside `main()`:

```
"Queue with Ring Buffer" example {
  val queue = RingBufferQueue<String>(10).apply {
    enqueue("Ray")
    enqueue("Brian")
    enqueue("Eric")
  }
  println(queue)
  queue.dequeue()
  println(queue)
  println("Next up: ${queue.peek()}")
}
```

This test code works like the previous examples dequeuing Ray and peeking at Brian.

Strengths and weaknesses

How does the ring-buffer implementation compare? Let's look at a summary of the algorithmic and storage complexity.



Ring-Buffer Based Queue

Operations	Best case	Worse case
enqueue	O(1)	O(1)
dequeue	O(1)	O(1)
Space Complexity	O(n)	O(n)

The ring-buffer-based queue has the same time complexity for enqueue and dequeue as the linked-list implementation. The only difference is the space complexity. The ring buffer has a fixed size, which means that enqueue can fail.

So far, you've seen three implementations: an array, a doubly linked-list and a ring-buffer.

Although they appear to be eminently useful, you'll next look at a queue implemented using two stacks. You'll see how its spatial locality is far superior to the linked list. It also doesn't need a fixed size like a ring buffer.

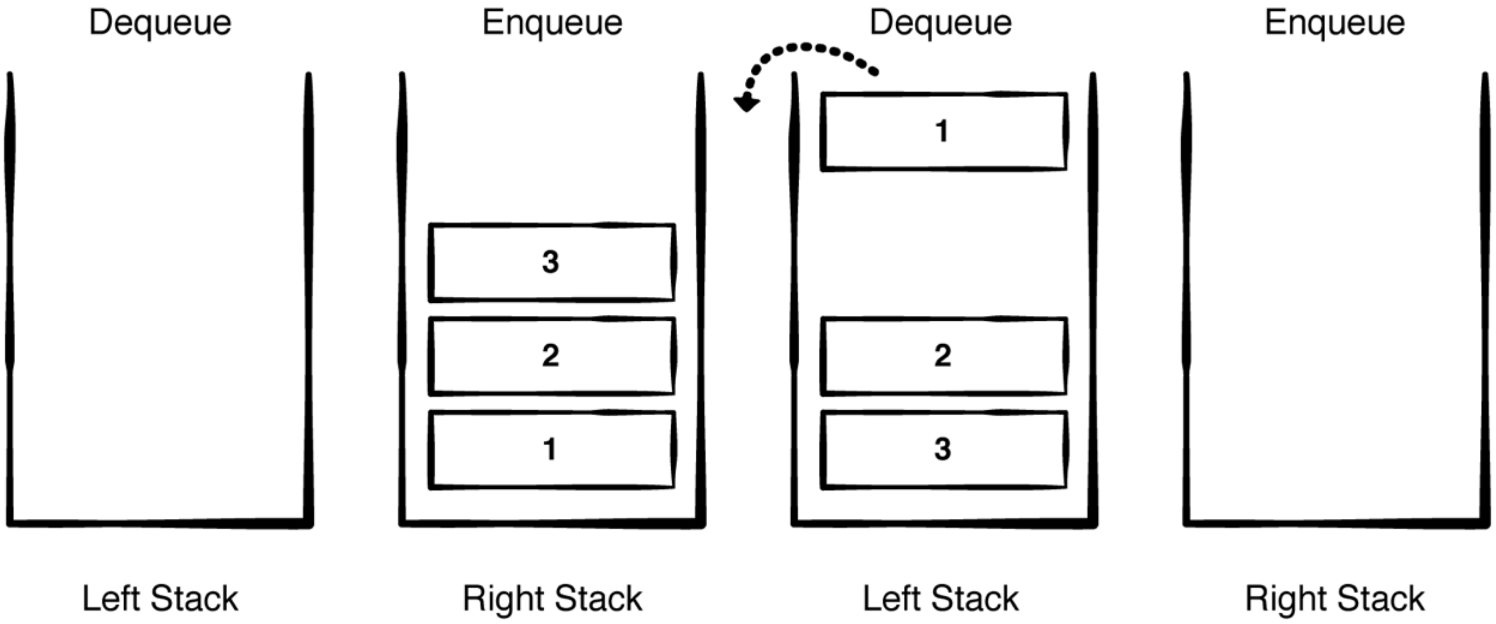
Double-stack implementation

Go to the **doublestack** package and start by adding a **StackQueue.kt** containing:

```
class StackQueue<T : Any> : Queue<T> {
  private val leftStack = StackImpl<T>()
  private val rightStack = StackImpl<T>()
}
```

The idea behind using two stacks is simple. Whenever you enqueue an element, it goes in the **right** stack.

When you need to dequeue an element, you reverse the right stack and place it in the **left** stack so that you can retrieve the elements using FIFO order.



Leveraging the stacks

Implement the common features of a queue, starting with the following:

```
override val isEmpty: Boolean
  get() = leftStack.isEmpty && rightStack.isEmpty

override val count: Int
  get() = leftStack.count + rightStack.count
```

To check if the queue is empty, simply check that both the left and right stack are empty. This means that there are no elements left to dequeue, and no new elements have been enqueued. The current count of elements in the queue is the sum of the counts in the two stacks.

As you already know, there will be a time when you need to transfer the elements from the right stack into the left stack. That needs to happen whenever the left stack is empty. Add the following helper method:

```
private fun transferElements() {
  var nextElement = rightStack.pop()
  while (nextElement != null) {
    leftStack.push(nextElement)
    nextElement = rightStack.pop()
  }
}
```

With this code, you pop elements from the right stack and push them into the left stack. You already know from the previous chapter that stacks work in a LIFO way (last in, first out). You'll get them in reversed order without any additional work.

Next, add the following:

```
override fun peek(): T? {
  if (leftStack.isEmpty) {
    transferElements()
  }
  return leftStack.peek()
}
```

You know that peeking looks at the top element. If the left stack is not empty, the element on top of this stack is at the front of the queue.

If the left stack is empty, you use `transferElements()`. That way, `leftStack.peek()` will always return the correct element or `null`. `isEmpty()` is still an *O*(1) operation, while `peek()` is *O*(*n*).

While this `peek()` implementation might seem expensive, it's amortized to *O*(1) because each element in the queue only has to be moved from the right stack to the left stack once. If you have a lot of elements in the right stack, calling `peek()` will be *O*(*n*) for just that one call when it has to move all of those elements. Any further calls will be *O*(1) again.

> **Note**: You could also make `peak()` operations precisely *O*(1) for all calls if you implemented a method in `Stack` that let you look at the very bottom of the right stack. That's where the first item of the queue is if

> they're not all in the left stack, which is what `peek()` should return in that case.

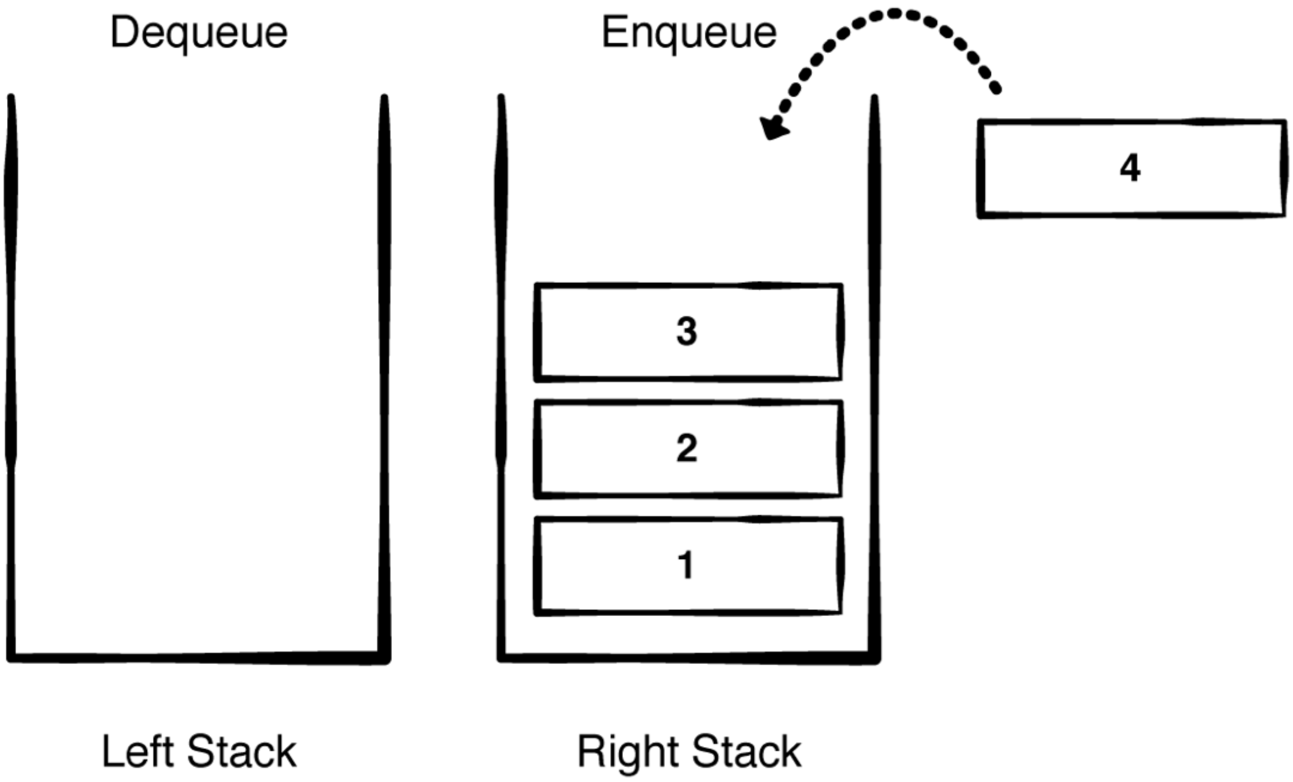
Enqueue

Next, add the method below:

```
override fun enqueue(element: T): Boolean {
  rightStack.push(element)
  return true
}
```

Recall that the **right** stack is used to enqueue elements.

Previously, from implementing `Stack`, you know that pushing an element onto it is an *O*(1) operation.



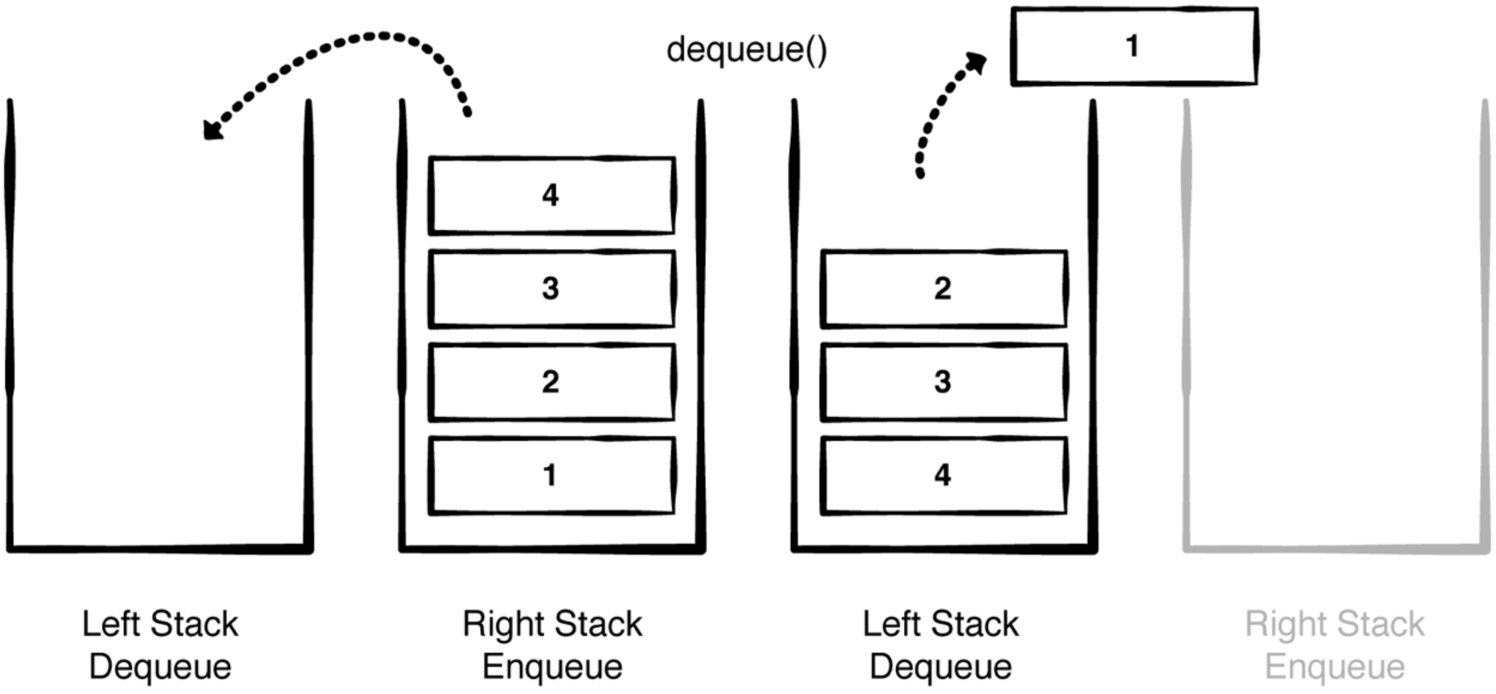
Dequeue

Removing an item from a two-stack-based implementation is as tricky as peeking. Add the following method:

```
override fun dequeue(): T? {
  if (leftStack.isEmpty) { // 1
    transferElements() // 2
  }
  return leftStack.pop() // 3
}
```

Here's how it works:

1. Check to see if the left stack is empty.
2. If the left stack is empty, you need to transfer the elements from the right stack in reversed order.



3. Remove the top element from the left stack.

Remember, you only transfer the elements in the right stack when the left stack is empty. This makes `dequeue()` an amortized *O*(1) operation, just like `peek()`.

Debug and test

To see your results, add the following to `StackQueue`:

```
override fun toString(): String {
  return "Left stack: \n$leftStack \n Right stack: \n$rightStack"
}
```

Here, you print the contents of the two stacks that represent your queue.

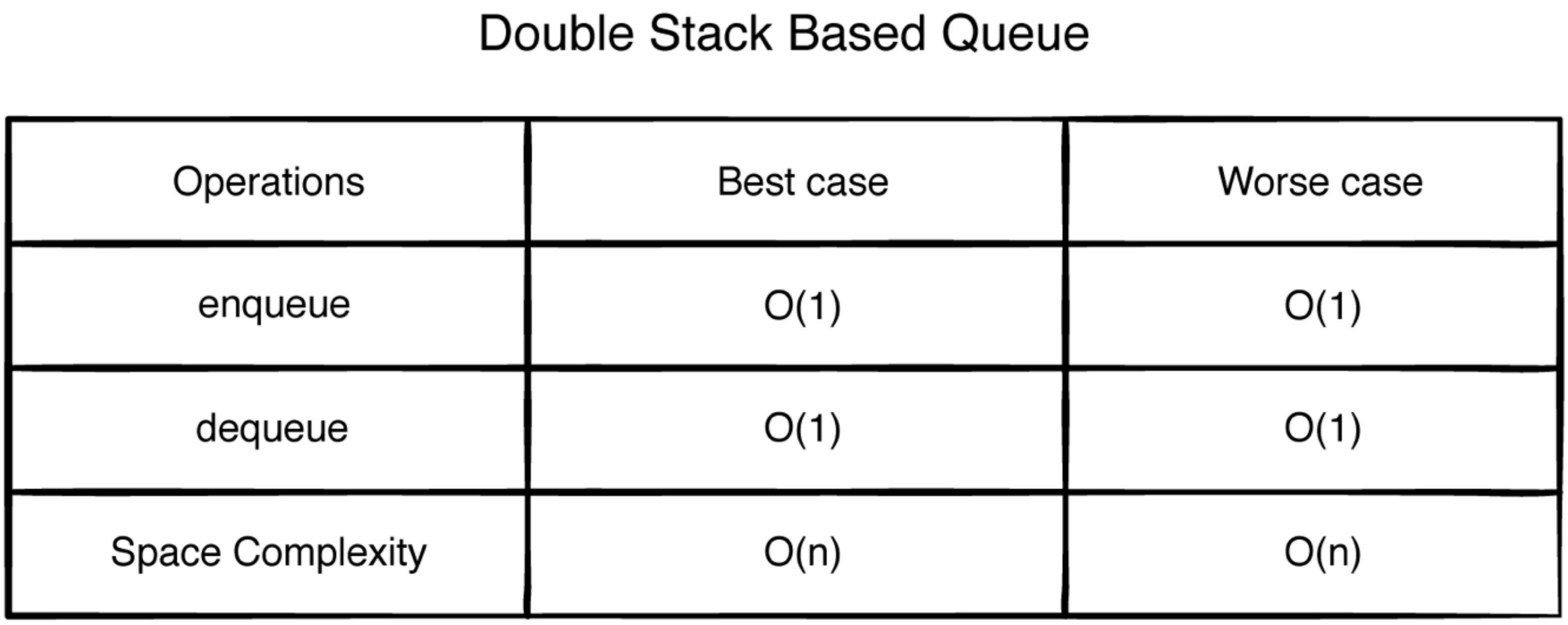
To try out the double-stack implementation, add the following to `main()`:

```
"Queue with Double Stack" example {
  val queue = StackQueue<String>().apply {
    enqueue("Ray")
    enqueue("Brian")
    enqueue("Eric")
  }
  println(queue)
  queue.dequeue()
  println(queue)
  println("Next up: ${queue.peek()}")
}
```

Similar to the previous examples, this code enqueues Ray, Brian and Eric, dequeues Ray and then peeks at Brian. Note how Eric and Brian ended up in the left stack and in reverse order as the result of the `dequeue` operation.

Strengths and weaknesses

Here's a summary of the algorithmic and storage complexity of your two-stack-based implementation.



Double Stack Based Queue

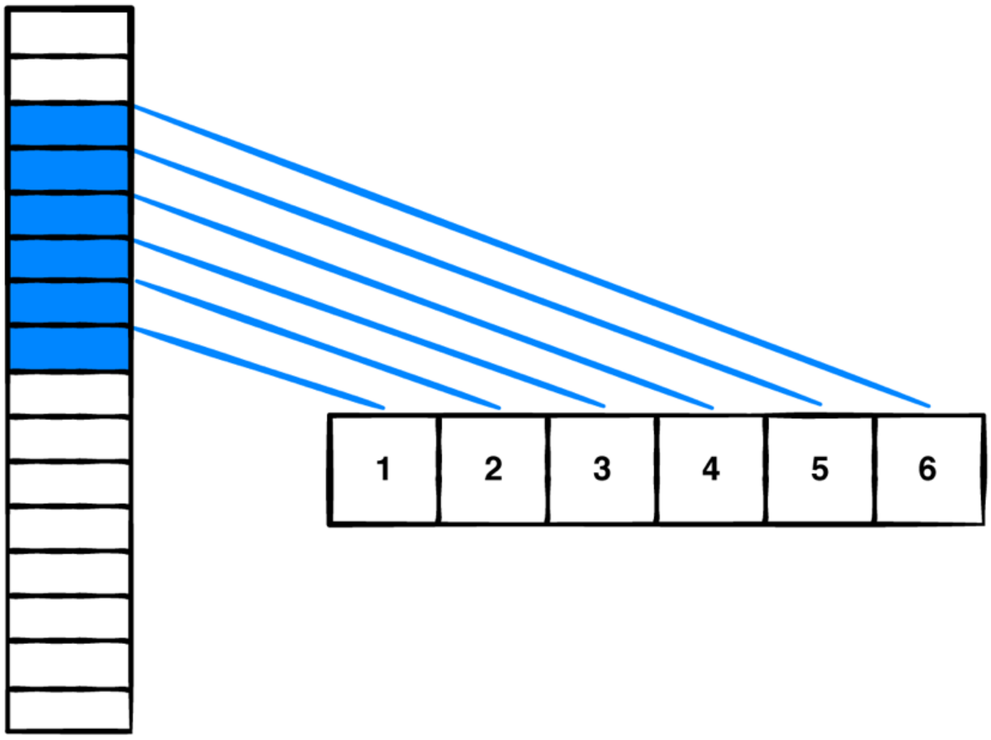
Operations	Best case	Worse case
enqueue	O(1)	O(1)
dequeue	O(1)	O(1)
Space Complexity	O(n)	O(n)

Compared to the list-based implementation, by leveraging two stacks, you were able to transform `dequeue()` into an amortized *O*(1) operation.

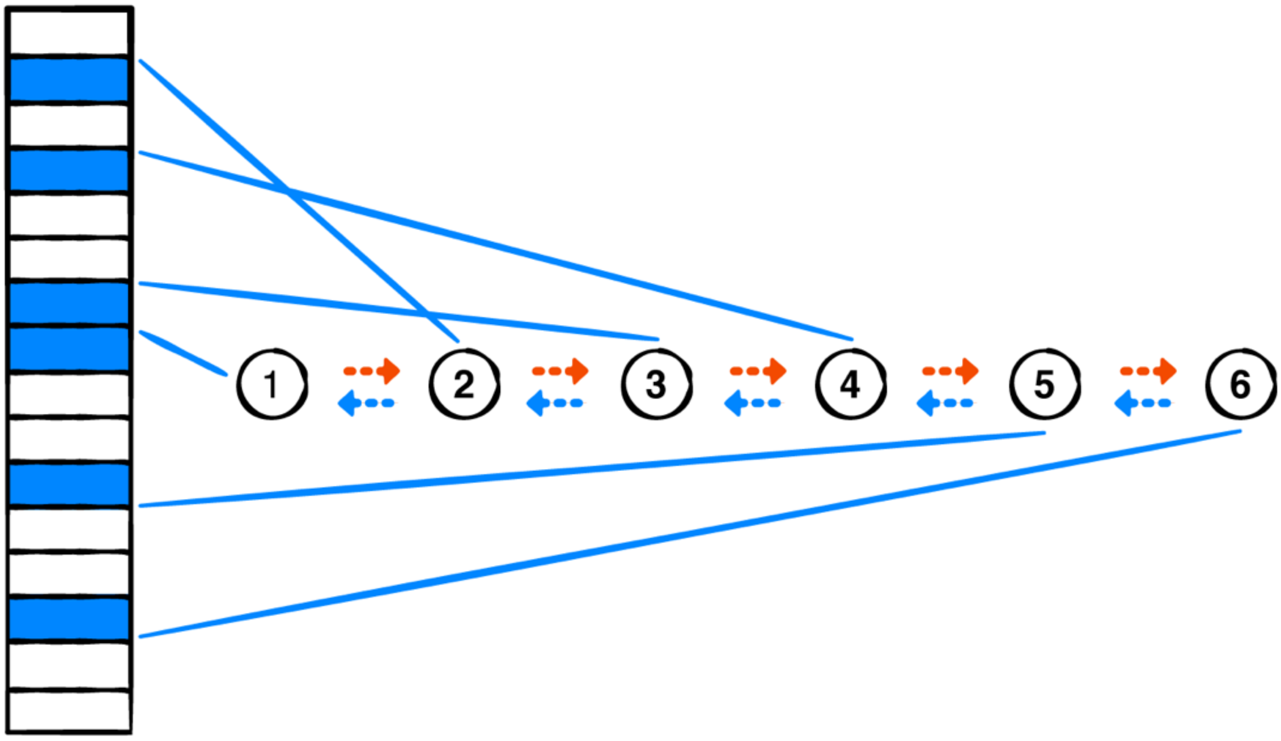
Moreover, your two-stack implementation is fully dynamic and doesn’t have the fixed size restriction that your ring-buffer-based queue implementation has.

Finally, it beats the linked list in terms of spatial locality. This is because list elements are next to each other in memory blocks. So a large number of elements will be loaded in a cache on first access.

Compare the two images on the following page; one has elements in a contiguous array, the other has elements scattered all over memory:



Elements in a contiguous array.



Elements in a linked list, scattered all over memory.

In a linked list, elements aren’t in contiguous blocks of memory. This could lead to more cache misses, which will increase access time.

Challenges

Think you have a handle on queues? In this section, you’ll explore five different problems related to queues. This serves to solidify your fundamental knowledge of data structures in general.

Challenge 1: Explain differences

Explain the difference between a stack and a queue. Provide two real-life examples for each data structure.

Solution 1

Queues have a behavior of first in, first out. What comes in first must come out first. Items in the queue are inserted from the rear and removed from the front.

Queue Examples:

1. **Line in a movie theatre**: You would hate for people to cut the line at the movie theatre when buying tickets!
2. **Printer**: Multiple people could print documents from a printer, in a similar first-come-first-serve manner.

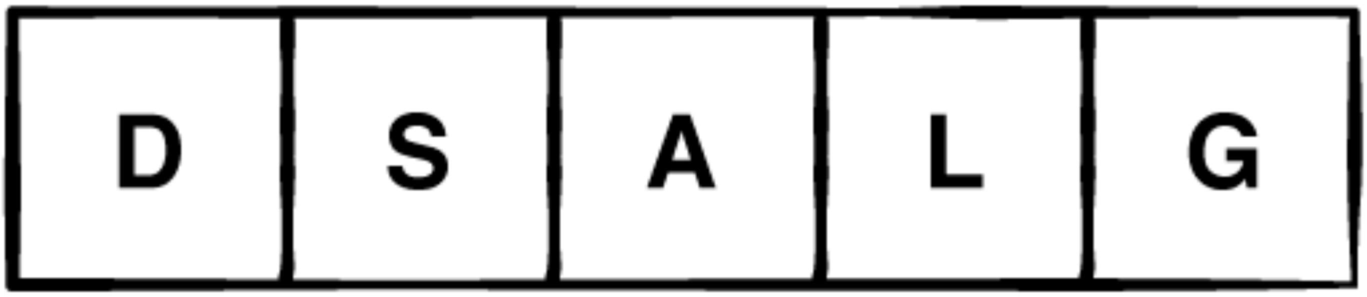
Stacks have a behavior of last-in-first-out. Items on the stack are inserted at the top and removed from the top.

Stack Examples:

1. **Stack of plates**: Placing plates on top of each other, and removing the top plate every time you use a plate. Isn't this easier than grabbing the one at the bottom?
2. **Undo functionality**: Imagine typing words on a keyboard. Most of the times, you would use undo for the last operation.

Challenge 2: What's the order?

Given the following queue:



Provide step-by-step diagrams showing how the following series of commands affects the queue:

```
enqueue("R")
enqueue("O")
dequeue()
enqueue("C")
dequeue()
dequeue()
enqueue("K")
}
```

Do this for the following queue implementations:

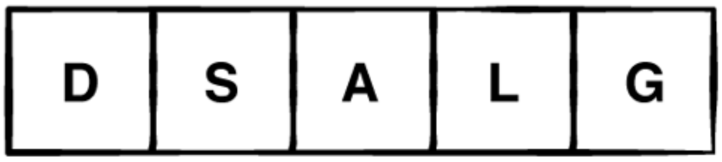
1. ArrayList
2. Linked list
3. Ring buffer
4. Double stack

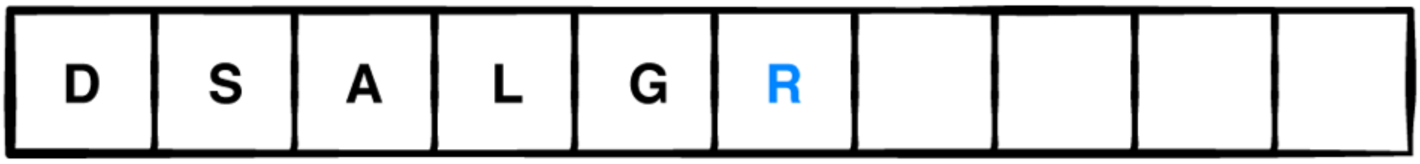
Assume that the list and ring buffer have an initial size of `5`.

Solution 2

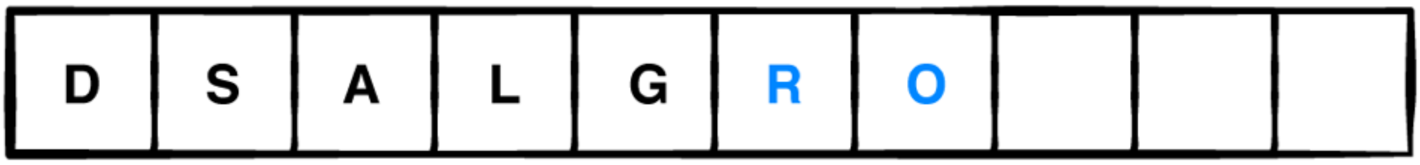
ArrayList

Keep in mind that whenever the underlying array is full, and you try to add a new element, a new array will be created with **twice** the capacity with existing elements being copied over.

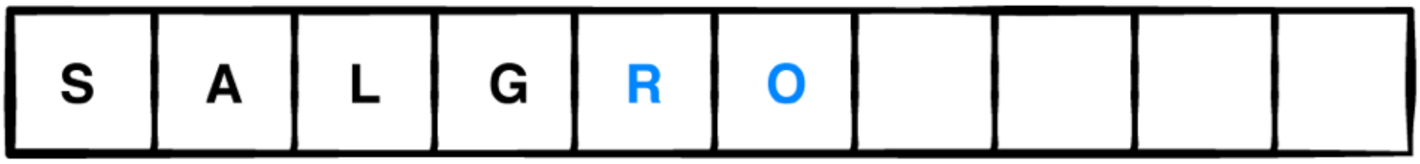




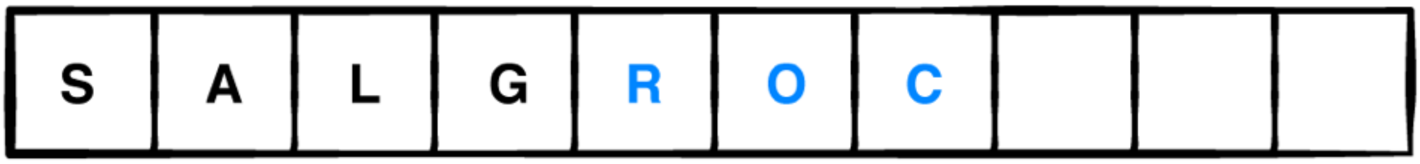
enqueue(”R”)



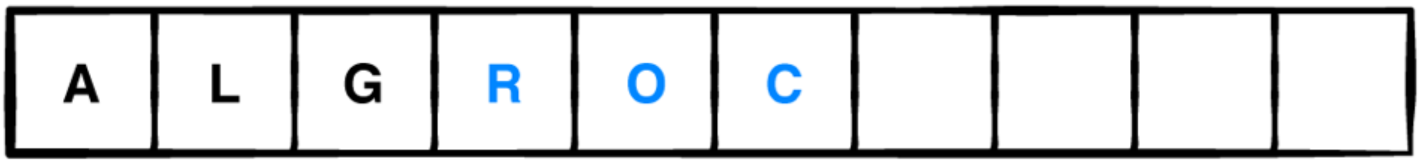
enqueue(”O”)



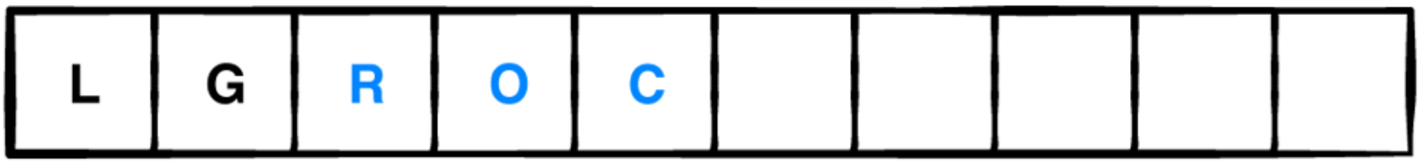
dequeue()



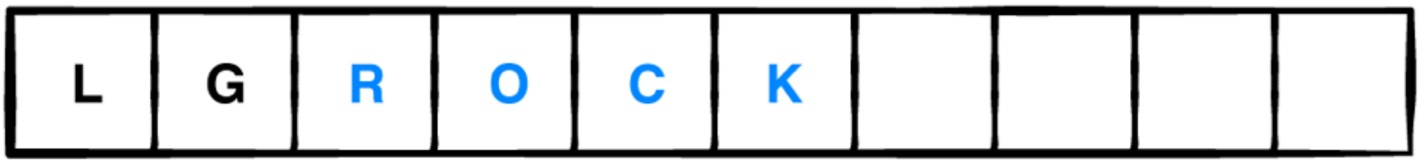
enqueue(”C”)



dequeue()

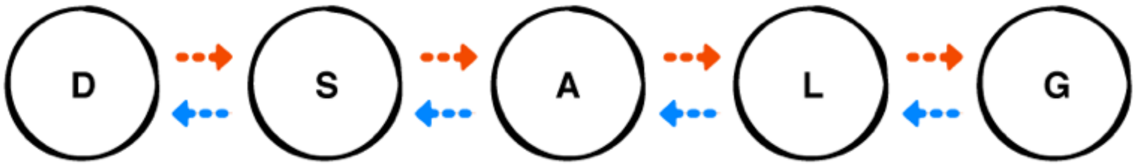


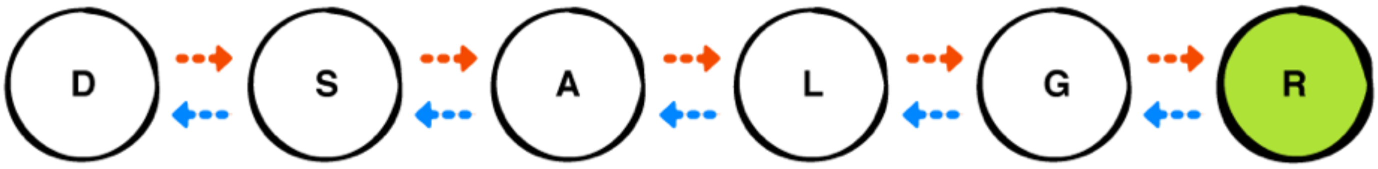
dequeue()



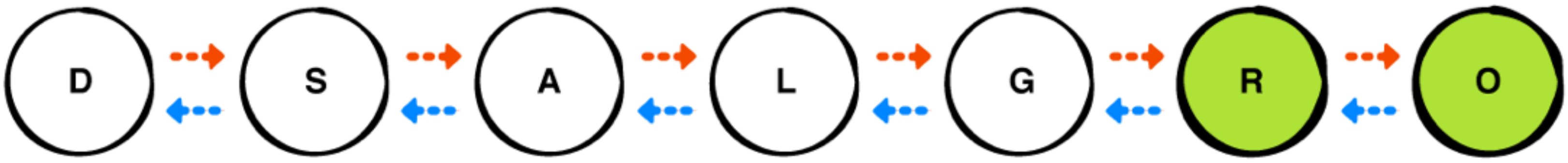
enqueue(”K”)

Linked list

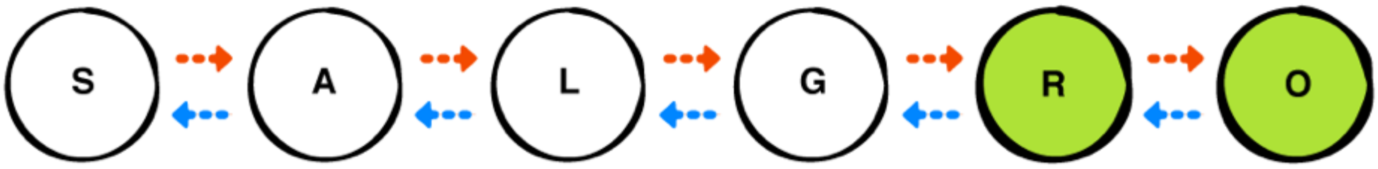




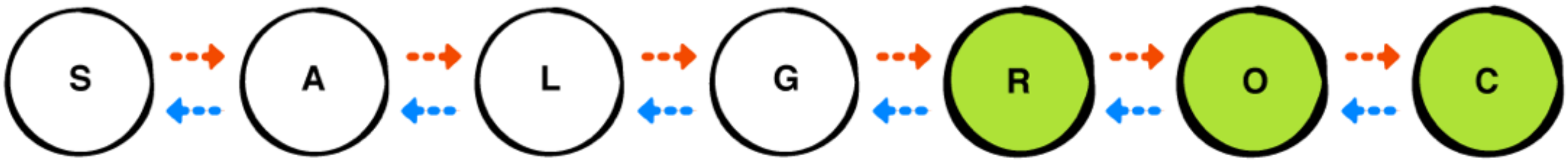
enqueue ("R")



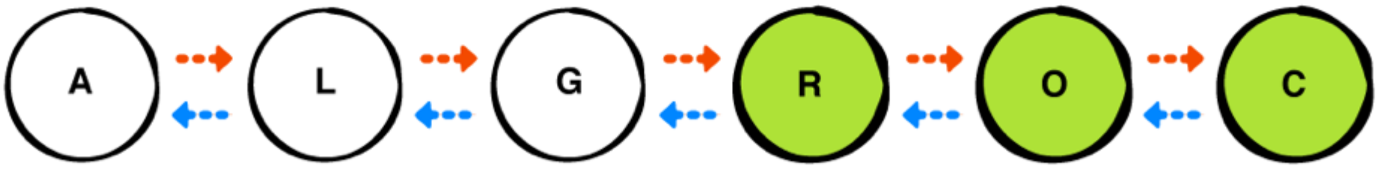
enqueue ("O")



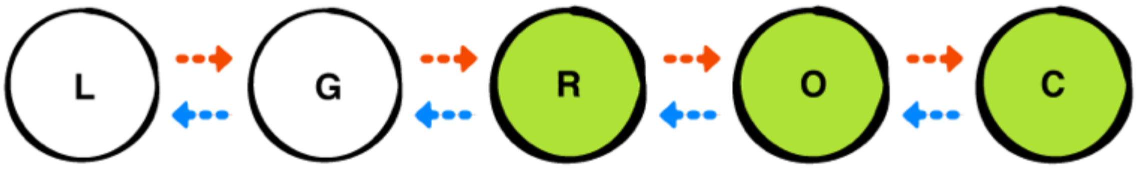
dequeue ()



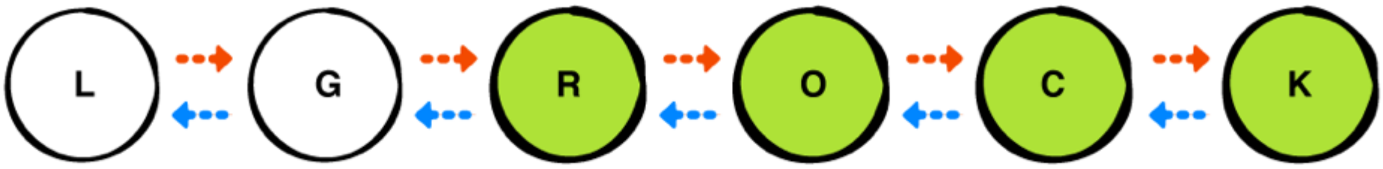
enqueue ("C")



dequeue()

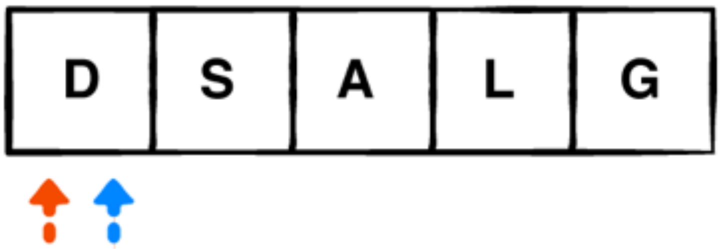


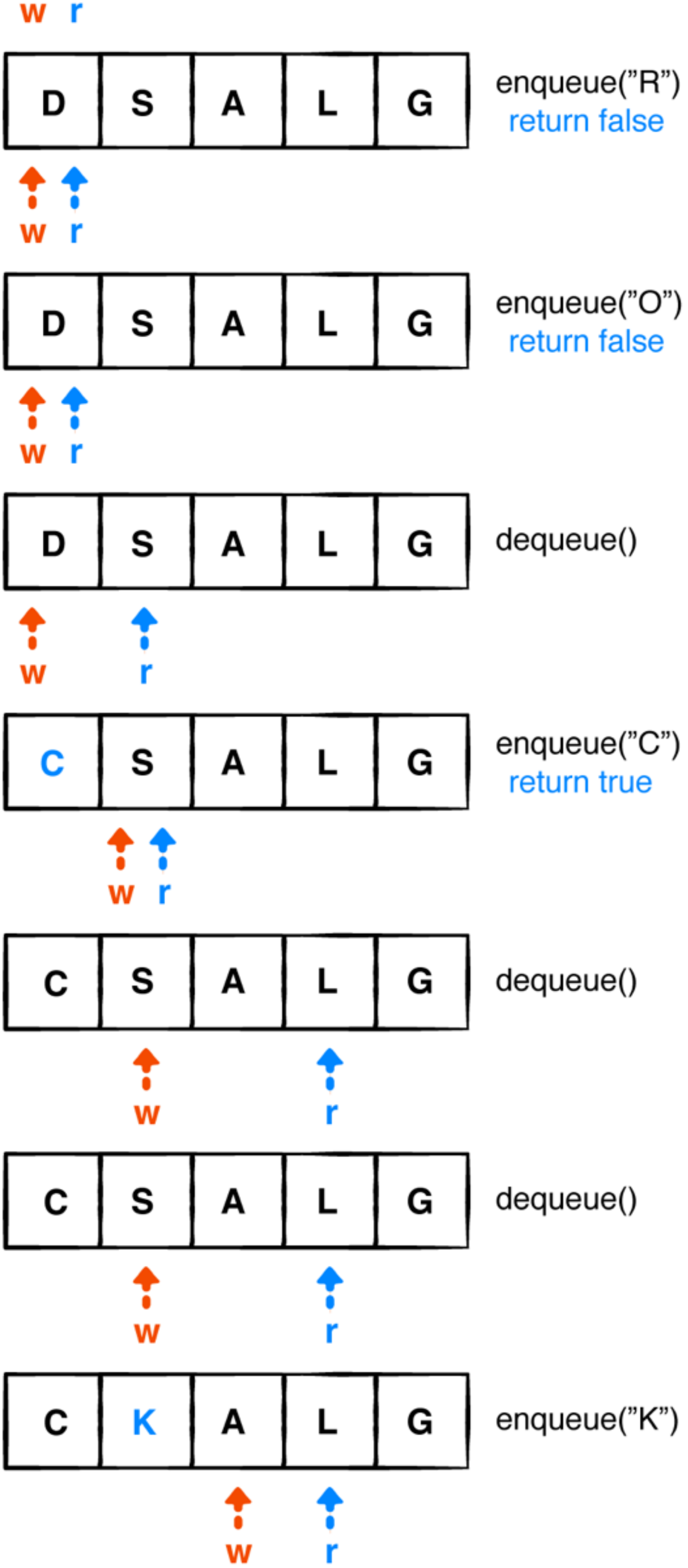
dequeue()



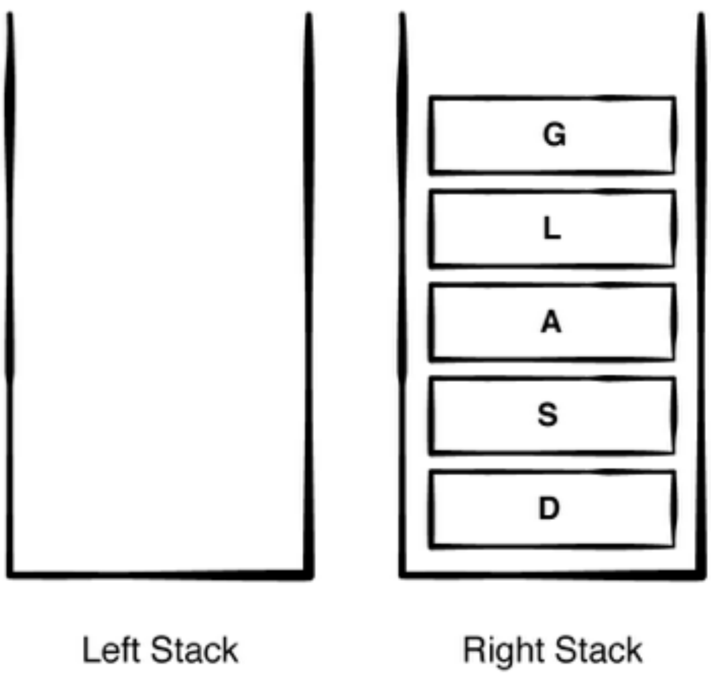
enqueue(K)

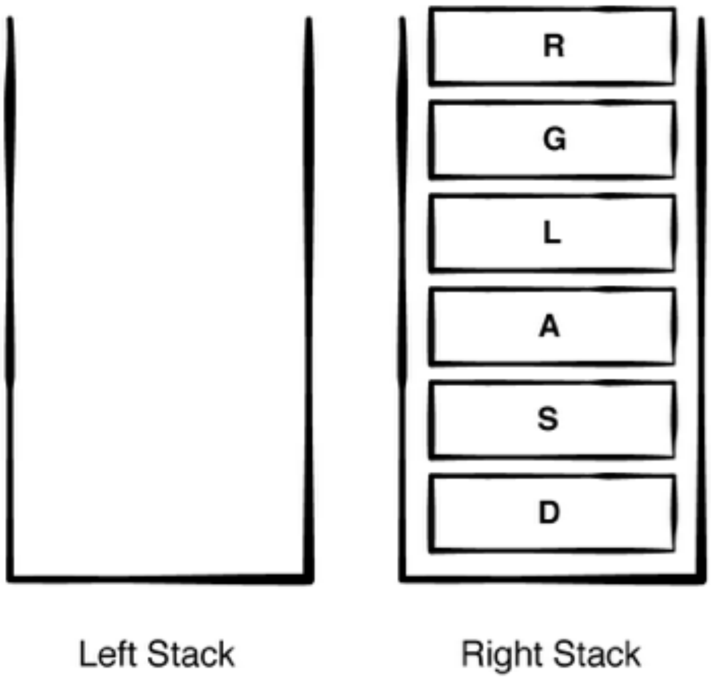
Ring buffer



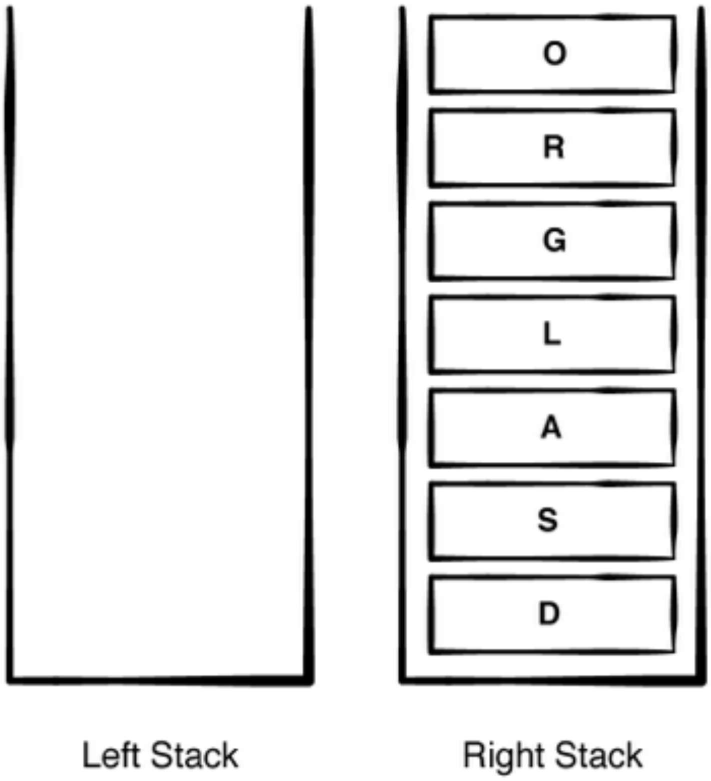


Double stack

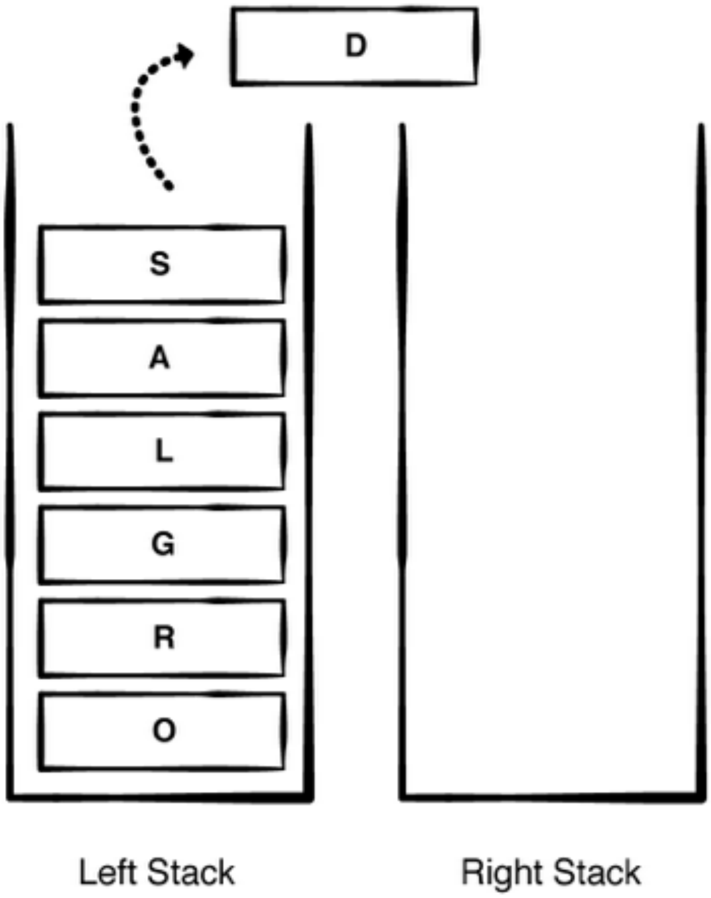




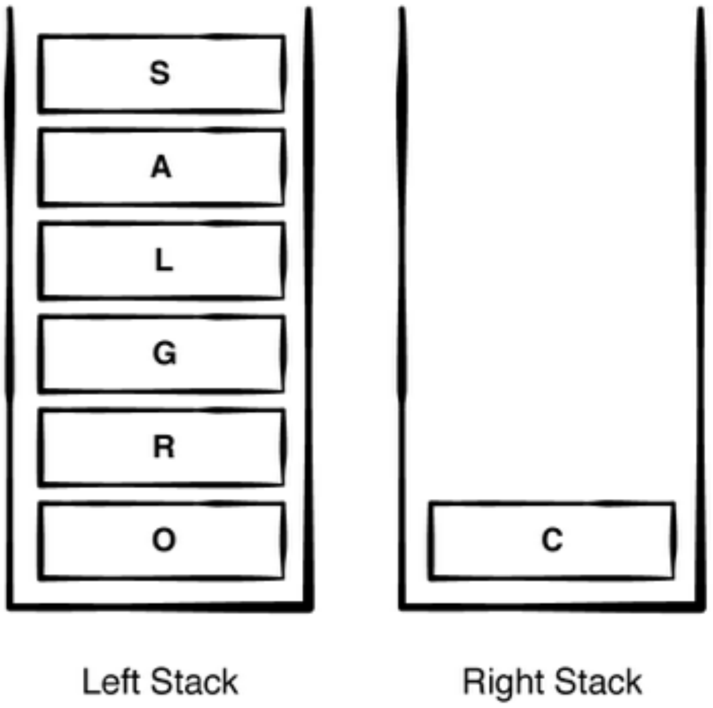
enqueue(”R”)



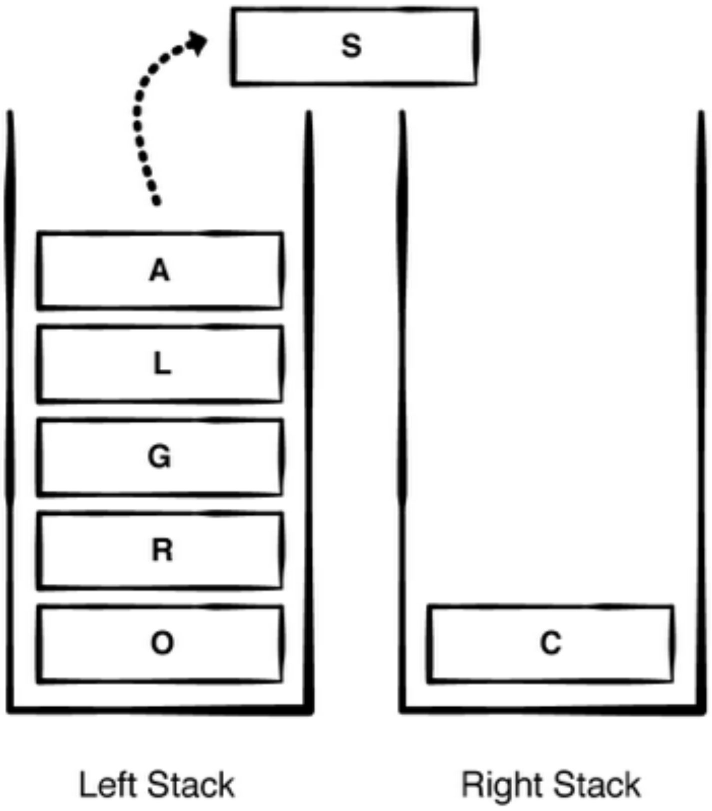
enqueue(”O”)



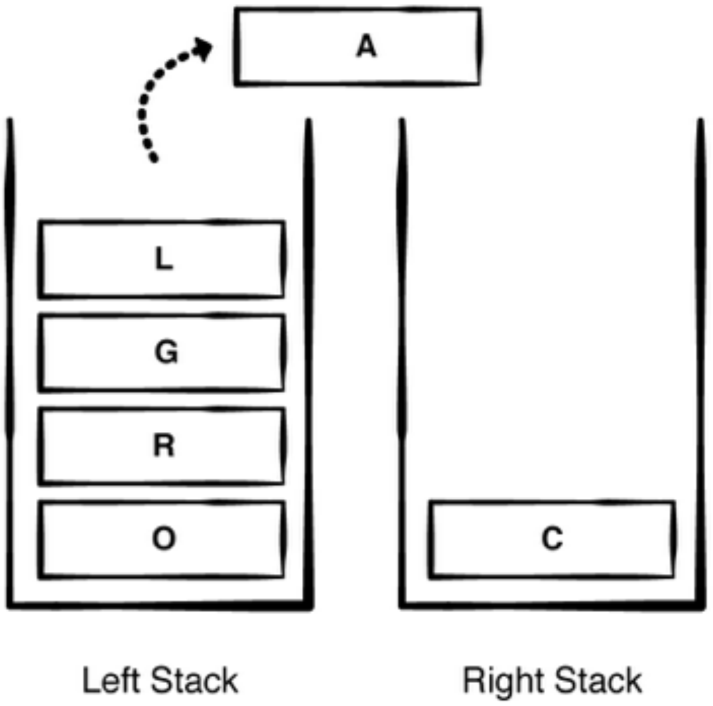
dequeue()



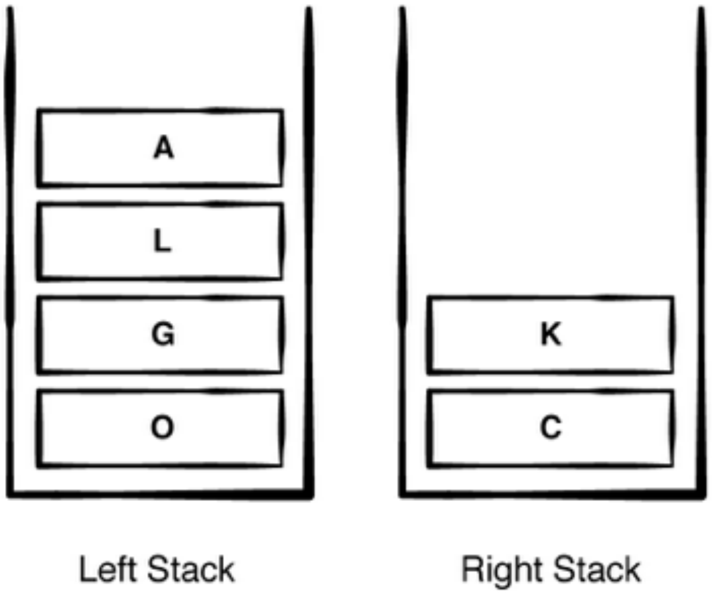
enqueue(”C”)



dequeue()



dequeue()



enqueue(”K”)

Challenge 3: Monopoly

Imagine you’re playing a game of Monopoly with your friends. The problem is that everyone always forgets whose turn it is! Create a Monopoly organizer that tells you whose turn it is. A great option is to create an extension function for `Queue` that always returns the next player. Here’s how the definition could look:

```
fun <T : Any> Queue<T>.nextPlayer(): T?
```

Solution 3

Creating a board game manager is straightforward. Your primary concern is whose turn it is. A queue data structure is a perfect choice to take care of game turns.

```
fun <T : Any> Queue<T>.nextPlayer(): T? {
  // 1
  val person = this.dequeue() ?: return null
  // 2
  this.enqueue(person)
  // 3
  return person
}
```

Here's how this works:

1. Get the next player by calling `dequeue`. If the queue is empty, return `null`, as the game has probably ended anyway.
2. `enqueue` the same person, this puts the player at the end of the queue.
3. Return the next player.

The time complexity depends on the queue implementation you select. For the array-based queue, it's overall _O(*n*) time complexity. `dequeue` takes _O(*n*) time because it has to shift the elements to the left every time you remove the first element.

Testing it out:

```
"Boardgame manager with Queue" example {
  val queue = ArrayListQueue<String>().apply {
    enqueue("Vincent")
    enqueue("Remel")
    enqueue("Lukiih")
    enqueue("Allison")
  }
  println(queue)

  println("===== boardgame =======")
  queue.nextPlayer()
  println(queue)
  queue.nextPlayer()
  println(queue)
  queue.nextPlayer()
  println(queue)
  queue.nextPlayer()
  println(queue)
}
```

Challenge 4: Reverse data

Implement a method to reverse the contents of a queue using an extension function.

> Hint: The `Stack` data structure has been included in the project.

```
fun <T : Any> Queue<T>.reverse()
```

Solution 4

A queue uses first in, first out whereas a stack uses last in, first out. You can use a stack to help reverse the contents of a queue. By inserting all of the contents of the queue into a stack, you basically reverse the order once you pop every element off the stack.

```
fun <T : Any> Queue<T>.reverse() {
  // 1
  val aux = StackImpl<T>()

  // 2
  var next = this.dequeue()
  while (next != null) {
    aux.push(next)
    next = this.dequeue()
  }

  // 3
  next = aux.pop()
  while (next != null) {
    this.enqueue(next)
    next = aux.pop()
  }
}
```

For this solution, you added an extension function for any `Queue` implementation. It works the following way:

1. Create a stack.
2. `dequeue` all of the elements in the queue onto the stack.
3. `pop` all of the elements off the stack and insert them into the queue.
4. Return your reversed queue.

The time complexity is overall *O*(*n*). You loop through the elements twice. Once for removing the elements off the queue, and once for removing the elements off the stack.

Testing it out:

```
"Reverse queue" example {
  val queue = ArrayListQueue<String>().apply {
    enqueue("1")
    enqueue("21")
    enqueue("18")
    enqueue("42")
  }
  println("before: $queue")
  queue.reverse()
  println("after: $queue")
}
```

Key points

- Queue takes a FIFO strategy, an element added first must also be removed first.
- Enqueue inserts an element to the back of the queue.
- Dequeue removes the element at the front of the queue.
- Elements in an array are laid out in contiguous memory blocks, whereas elements in a linked list are more scattered with potential for cache misses.
- A ring buffer based queue implementation is useful for queues with a fixed size.
- Compared to other data structures, leveraging two stacks improves the `dequeue()` time complexity to an amortized $O(1)$ operation.
- The double-stack implementation beats linked list in terms of spatial locality.

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.