We begin with the simpler linked-list implementation of stacks and queues (section 12.7) before going into the richer set of operations in section 12.4. Here are the traits they will implement:

```scala
trait Stack[A] {
  def peek: A
  def pop(): A
  def push(item: A): Unit
  def isEmpty: Boolean
  def size: Int  //do we promise O(1) time for this too?
}

trait Queue[A] {
  def peek: A
  def pop(): A    //preferable to "dequeue" IMPHO
  def enqueue(item: A): Unit
  def isEmpty: Boolean
  def size: Int    //do we promise O(1) time for this too?
}

trait Staque[A] extends Stack[A] with Queue[A]
```

Per Scala conventions, `pop()` has parens to indicate its doing a side effect, while `peek` does not. We will implement these in terms of the operations previously conceptualized as `pushFront`, `pushRear`, and `popFront` (and `popRear` when we get to double-ended queues and doubly linked lists), but the implementation concepts will be hidden from the outside world. [In the array-based implementations, we use different ends of the array for "pop()" between `Stack` and `Queue`, but clients need not know or care.] Compared to the text, I've included `size` into the trait, to highlight the question of whether it should be "contracted" as running in $O(1)$ time.

**Singly Linked Lists**

Linked lists are made out of `nodes`. Each node holds one data item and one or two links to neighboring nodes. It is standard to represent the nodes via a `private inner class` `Node`. In Scala this can be a single line of code.

```scala
class ListStack[A] extends Stack[A] {

  /** Inner class. Can see the generic type A. One line of code is enough! */
  ```
private class Node(var item: A, var next: Node)

If we wrote `val item: A`, then the data would be immutable at its own top level. This would only go part way to creating a truly immutable list data structure (the text goes thru many more machinations in section 12.6). If we wrote `val next: Node` then we would not be able to change any links once a node is created. See where and whether we need to change a `next` link in what follows.

The `ListStack[A]` class itself has no class arguments. The body of the class is its own constructor. This may feel weirdest coming over from Java or C++, maybe a little less weird from Python or JavaScript. Java allows a little more freedom than C++ to initialize class fields directly, but variable fields must be set by constructors. In Scala you "just do it"—the class fields are also initializations executed in the body of the one primary constructor:

```scala
private var head: Node = null  //INV: On legal first item unless list is empty.
private var _size = 0
```

The second field follows the convention of using an underscore for a private settable field, which the non-underscore name `size` (as specified already in the trait) for its getter. Now—not being quite as simple as the text—we continue the implementation privately.

```scala
private def peekFront: A = { return head.item }
```

In the public section we will do `def peek = peekFront` to implement the method by the name required in the trait. But let's stop to ponder the question: what if (as initially) `head` is `null`? Then we will get a run-time error. Literally, we will get a `java.lang.NullPointerException` and might have to hunt to see where and why. We can localize the error and give a more specific syntax for catching it by using Scala's native assertions facility:

```scala
private def peekFront: A = { assert(!isEmpty, "Peek at empty stack."); return head.item }
```

Although Scala executes field commands sequentially, methods can be declared in any order—so the `isEmpty` test can be defined later. We can use a similar test for `pop()`:

```scala
private def popFront(): A = {
  assert(!isEmpty, "Pop from empty stack.")
  val ret = head.item
  head = head.next   //Scala does not require manual deallocation like in C++
  _size -= 1         //extra line adds to the principal constant of the "O(1)"
  return ret
}
```

Pushing a new element, however, does not need the test `isFull` used by the array implementation. We
again need just one line of code:

```scala
private def pushFront(item: A) = { head = new Node(item,head); _size += 1 }
```

OK, we used 2 lines because we also update the _size field on the fly, so that it fulfills its obvious contract, INV: _size always has the correct number of real data points in the container. Doing so allows meeting every call to the public size method in $O(1)$ time. Let’s discuss this further after seeing the rest of the class:

```scala
//public section
def peek = peekFront
def pop() = popFront()
def push(item: A) = pushFront(item)
def isEmpty: Boolean = (_size == 0) // (head == null)
def size = _size

//this one requires transversal
override def toString = {
  var ret: List[A] = Nil
  var rover = head
  while (rover != null) {
    ret ::= rover.item
    rover = rover.next
  }
  ret.reverse.toString
}
``` //end of class ListStack[A]

Reversing the output of toString gives the oldest element first. Note that this is the reverse of the List[A] class, and whether it is $O(1)$ time by itself is moot because we are outputting $n$ units of stuff anyway. It is an example of traversing the class by a "pointer" (here, a reference). We will later wrap the "rover" into an associated Iterator object.

Now back to the $O(1)$ time issue for size. The cost of this privilege is incremental: every time we push or pop, we have to execute 1 more line of code to increment or decrement _size. Sometimes this needs no extra time because it slots into the bandwidth of the instruction pipeline. But it is more work regardless. If we don’t do that, then we would have to implement size by the same kind of transversal as toString, counting elements one-by-one. This would take $\Theta(n)$ time, i.e., "$O(n)$ time best possible." The Scala library generally does not promise size or length computation in $O(1)$ time.
And back to an earlier question: Did we ever update a node's next field? Above, no. Prepending and pulling items off the front of the list needs no splicing of internal links. But appending, which we need for the Queue[A] implementation, does. We can see why already from the code snippets shown for linked lists at https://visualgo.net/en/list ...

We will introduce a sentinel end node, not going as far as the text does on page 421, as a precursor to later ideas about iteration. The following implements Stack and Queue together as "Staque", using the fact that 3 of the 4 double-ended queue operations are convenient with singly linked lists:

```scala
/** "Staque" with operations pushFront→push, pushRear→enqueue, popFront→pop.
   INV: (head is real node, penult.next == end), UNLESS list is empty
   The "UNLESS" needs a kludgey test on every push, but modern CPUs punish less for it.
*/
  private class Node(var item: A, var next: Node) {
    private val end = new Node(null.asInstanceOf[A], null)   //note special feature
  }
  private var head = end       //INV: head.item is real data unless list is empty
  private var penult = end
  private var _size = 0

  private def peekFront: A = {
    assert(!isEmpty, "Peek at empty Staque.")
    return head.item
  }
  private def popFront(): A = {
    assert(!isEmpty, "Pop from empty Staque.")
    val ret = head.item
    head = head.next   //Scala does not require manual deallocation like in C++
    _size -= 1         //extra line adds to the principal constant of the "O(1)"
    return ret
  }
  private def pushFront(item: A) = {
    if (isEmpty) {
      head = new Node(item, end)
      penult = head
    } else {
      head = new Node(item, head)  //leaves penult situated OK
    }
    _size += 1
  }
  private def pushRear(item: A) = {
```

```scala
```
if (isEmpty) {  //same code as for pushFront in this case
    head = new Node(item, end)
    penult = head
} else {
    penult.next = new Node(item, penult.next)  //need to update penult too
    penult = penult.next
}
_size += 1
}

//public section
def peek = peekFront
def pop() = popFront()
def push(item: A) = pushFront(item)
def enqueue(item: A) = pushRear(item)
def isEmpty: Boolean = (_size == 0) // (head == null)
def size = _size
    //toString is as before---duplicated code.
}

For pushRear we had to reassign the next field, so it could not be val. The syntax for inserting in a new node anywhere in the list is in fact the same for splicing it in front of the sentinel.