Lecture 32: DynamicArrays
10:00 AM, Apr 16, 2018

Contents

1 Dynamic Arrays 1
2 Implementation 3
   2.1 Common Case ................................................. 6
   2.2 Edge Case ..................................................... 8

Objectives

By the end of this lecture, you will know:

• what a dynamic array is

By the end of this lecture, you will be able to:

• implement a dynamic array

NOTE: The notes on MSTs from the beginning of the lecture on April 17th are under the previous class, lecture 31.

1 Dynamic Arrays

We have already implemented the IList interface once, using linked lists. Today we are going to implement this interface again, using arrays. As a refresher, here is our original IList interface again, extended with a get method, since random access is easy using arrays:

```scala
/**
 * An interface for lists.
 *
 * @param T - the type of items in the list
 */
trait IList[T] extends Iterable[T] {

  /**
   * Returns the item at the specified index.
   *
   * @param index - an item index
   * @return the item at the specified index, or None if no such item exists
   */
  def get(index: Int): Option[T] = { /* implementation */ }
}
```
Implementing lists using arrays may strike you as non-intuitive, because there is a fundamental difference between arrays and lists: an array has a fixed size, whereas the length of a list can be arbitrary. Consequently, there are two things that can go wrong:

1. First, we can try to remove or get an item from an empty array. This is called an *underflow*
error.

2. Second, we can try to add an item to an array that is already full. This is called an overflow error.

The semantics of lists preclude the possibility of removing anything from an empty list. To handle underflow errors in Scala, the get and remove methods simply return an option (i.e., None, when the list is empty).

Those same semantics impose no upper bound at all on the length of a list. Not 1000. Not 10000. Not 100000. One workaround could be to represent lists using massive arrays. But memory is a finite resource, and we cannot use it frivolously.

The fundamental problem is that a static array violates the abstraction of a list in that it cannot grow to an arbitrary size. We will solve this problem by replacing full arrays with bigger ones. That is, we’ll design a dynamic array that grows in response to overflow, instead of failing.

Before we proceed with this implementation, let’s spend a moment thinking about the pros and cons of dynamic arrays vs. linked lists. While linked lists are certainly efficient (all the basic operations take constant or linear time), accessing items of a list using array indexing is even more efficient. Hence, backing a list interface with a dynamic array rather than a linked list would be preferable when representing sequential data—for example, strings—where it is useful to be able to extract subsequences quickly.

Moreover, dynamic arrays—when appropriately sized—use less space than linked lists because nodes in a linked list have some overhead associated with them (namely, the next references). But it is not always easy to appropriately size a dynamic array. Doing so may be particularly difficult when storing a list whose size changes often (e.g., a factory’s inventory or a FIFO queue).

2 Implementation

To store a list in an array, we will allocate a chunk of memory, and then store the items of the list in a (more or less) contiguous block of that memory. Let’s call the size of this chunk of memory the array’s capacity.

As the list interface allows us to add and remove items from the front and the back of the list, it makes sense to endow lists with two attributes, start and end, that record where (i.e., at what index) the list starts and ends. Note that in an array of capacity 5, a list of capacity 2 may start at index 0 and end at index 1; or it may start at index 1 and end at index 2; or it may start at index 2 and end at index 3; or it may start at index 4 and end at index 0. When we said “more or less’ contiguous block of . . . memory” earlier, what we meant was “contiguous, modulo wrapping around as necessary.” (See Figure 1.)

Observe that our use of the start and end attributes implicitly defines three invariants:

1. The start attribute stores the index where the list starts.

2. The end attribute stores the index where the list ends.

3. The ith item in the list is stored at index start + i modulo capacity.
Next, let’s address the question of how we might initialize \texttt{start} and \texttt{end}? You might expect that we should initialize both of these variables to 0. But if we do, then our invariants imply that our initial empty list will be of length 1 even though it has no items!

That can’t be right?! Sounds like we need another invariant. Luckily, we have one. It is implied by the first two:

4. The number of items in our list is \( \texttt{end} - \texttt{start} + 1 \).

(Well, this is only half the story, actually. This invariant holds whenever \texttt{end} is greater than or equal to \texttt{start}. But if \texttt{start} is greater than \texttt{end}, then the number of items in our list is \( \texttt{end \ plus capacity} - \texttt{start} + 1 \).)

Okay, so that rules out initializing \texttt{start} and \texttt{end} to the same value. But what should we do instead? Well, let’s take a look at our three invariants. If the number of items in our list is \( \texttt{end} - \texttt{start} + 1 \), then if our list is empty (i.e., the number of items is 0), it follows that \( \texttt{end} = \texttt{start} - 1 \). So, if we initialize \texttt{start} to 10, then we should initialize \texttt{end} to 9. That’s easy enough.

But wait: the most natural way to initialize \texttt{start} is to 0. Then what? Well, in that case we cannot simply subtract 1 from \texttt{start} or we will arrive at a negative index into our array. No problem; in this case, we can simply add \texttt{capacity} to \texttt{start} before subtracting 1. In fact, regardless of how we initialize \texttt{start}, we can initialize \texttt{end} correspondingly to \( \texttt{start} + \texttt{capacity} - 1 \), modulo \texttt{capacity}. (See Figure \ref{fig:empty})

Based on this relationship, let’s revise our fourth invariant:
4. The number of items in our list is \( \text{end} + \text{capacity} - \text{start} + 1 \), modulo \( \text{capacity} \).

Alternatively, we could modify two of our four invariants as follows:

1. The attribute \( \text{end} \) stores the index one more than where the list ends.
2. The number of items in our list is \( \text{end} + \text{capacity} - \text{start} \), modulo \( \text{capacity} \).

With these alternative invariants, we could initialize both \( \text{start} \) and \( \text{end} \) to 0 (or to any other non-negative integer value less than \( \text{capacity} \); the important thing would be that both variables have the same initial value).

Although our initialization satisfies our invariant, something is still not quite right. Take, for example, the initialization \( \text{start} = 0 \) and \( \text{end} = 3 \). When \( \text{start} \) and \( \text{end} \) take on these values, we cannot be sure that the list is empty. Alternatively, it could be that the list is full! Hence, we introduce an additional attribute \( \text{numItems} \), which stores the number of items in the array.

Note that the \( \text{end} \) attribute contains redundant information, given the \( \text{start} \) and \( \text{numItems} \) attributes. With only these latter two attributes, there are only three corresponding invariants:

1. The attribute \( \text{start} \) stores the index where the list starts.
2. The attribute \( \text{numItems} \) stores the number of items in the list.
3. The \( i \)th item in the list is stored at index \( \text{start} + i \) modulo \( \text{capacity} \).

Similarly, the \( \text{start} \) attribute contains redundant information given the \( \text{end} \) and \( \text{numItems} \) attributes. Nonetheless, we retain all three for clarity.

Given these three attributes (\( \text{start} \), \( \text{end} \), and \( \text{numItems} \)), here is the beginning of our implementation of the \code{DynamicArray} class, which uses arrays to implement the \code{IList} interface.

```scala
class DynamicArray[T: ClassTag](capacity: Int) extends IList[T] {
    private var array = new Array[T](capacity)
    private var start = 0
    private var end = capacity - 1
    private var numItems = 0

    ...
}
```

In Java, you are not allowed to declare an array to be of a generic (i.e., parameterized) type. In particular, the following declaration is illegal:

```java
T[] array = new T[capacity];
```
Scala does allow such declarations. In the above code, we do precisely this:

```scala
private val array = new Array[T](capacity)
```

But in order for this to work, Scala requires the type annotation `T: ClassTag`, which you’ve surely noticed at the top of our class definition. This annotation tells the compiler that the DynamicArray class is parameterized by the type `T`, and allows us to use the type `T` in the array declaration.

Next, we override the `isEmpty` method and define the private method `isFull` in our DynamicArray class. The former will be used by the get and remove methods, and the latter by the add methods—to decide when to grow the dynamic array.

```scala
override def isEmpty: Boolean = (numItems == 0)
private def isFull: Boolean = (numItems == capacity)
```

At this point, it is straightforward to write `getFirst` and `getLast`:

```scala
override def getFirst: Option[T] = if (isEmpty) None else Some(array(start))
override def getLast: Option[T] = if (isEmpty) None else Some(array(end))
```

The `get` method is similarly simple, but not only do we have to check that the list is not empty, we also must check that the input index is within the range of valid indices: i.e., between 0 (to access the first item in the list) and `numItems` -1 (to access the last), inclusive:

```scala
def get(index: Int): Option[T] =
  if (index < 0 || index >= numItems) throw new IndexOutOfBoundsException()
  else if (isEmpty) None
  else Some(array((start + index) % capacity))
```

### 2.1 Common Case

Now, when we `addFirst` to an empty list, we can decrement `start` (modulo `capacity`) so that `start` equals `end`; or, when we `addLast` to an empty list, we can increment `end` (modulo `capacity`) so that `end` equals `start`. In either case, after adding an item to an empty list, `start` and `end` both index the same item of the array, so our list is of length 1, as expected.

Given this basic infrastructure, we now proceed to implement `addFirst` and `addLast`, **assuming the list is not already full**.

To add an item to the beginning of a dynamic array:

1. decrement the start variable (modulo `capacity`)
2. insert the item at the index of the new start variable
3. increment the number of items in the list
Analogously, to add an item to the end of a dynamic array:

1. increment the end variable (modulo capacity)
2. insert the item at the index of the new end variable
3. increment the number of items in the list

Here is the code to accomplish these tasks:

```scala
override def addFirst(item: T) {
    start = dec(start)
    array(start) = item
    numItems += 1
}

override def addLast(item: T) {
    end = inc(end)
    array(end) = item
    numItems += 1
}
```

These methods are implemented using helper methods that increment and decrement an index, wrapping around as necessary:

```scala
private def inc(index: Int): Int = (index + 1 + array.length) % array.length
private def dec(index: Int): Int = (index - 1 + array.length) % array.length
```

Further, to remove an item from the beginning of a dynamic array:

1. save the item at the start of the array
2. increment the start variable (modulo capacity)
3. decrement the number of items in the list
4. return the saved item

Analogously, to remove an item from the end of a dynamic array:

1. save the item at the end of the array
2. decrement the end variable (modulo capacity)
3. decrement the number of items in the list
4. return the saved item

As above we implement these methods, using the helper methods `inc` and `dec`:
override def removeFirst(): Option[T] = {
  if (isEmpty) return None

  val toReturn = array(start)
  start = inc(start)
  numItems -= 1

  Some(toReturn)
}

override def removeLast(): Option[T] = {
  if (isEmpty) return None

  val toReturn = array(end)
  end = dec(end)
  numItems -= 1

  Some(toReturn)
}

At this point, we have completed the basic implementation of a dynamic array, which is used in the cases when it is not necessary to grow the array at all.

### 2.2 Edge Case

But sometimes it will be necessary the grow our dynamic array; specifically, whenever it is full. Hence, we will write a private method `grow` that takes as input an integer `newCapacity` and grows the current array to one of this `newCapacity`. This method operates as follows:

1. initialize a new array of size `newCapacity`
2. copy the contents of the old array into the new array, and set the `array` attribute to refer to the new array instead of the old one
3. reset the `start` and `end` variables appropriately

Here is one possible implementation of the `grow` method. This implementation inserts items of the old array into the new array beginning at index 0.

```scala
private def grow(newCapacity: Int) {
  if (newCapacity < numItems) throw new IndexOutOfBoundsException()

  val newArray = new Array[T](newCapacity)
  for (i <- 0 until numItems) newArray(i) = array((start + i) % capacity)

  start = 0
  array = newArray
  end = numItems - 1
  capacity = newCapacity
}
```
Given this `grow` method, we can insert a single line at the start of our `addFirst` and `addLast` methods which tests whether our array is full, and if so, grows the array by some amount: e.g., by a constant `FACTOR`.

```java
if (isFull) grow(FACTOR * capacity)
```

Analogously, we can implement a `shrink` method which shrinks our dynamic array when it is nearly empty. A test for near-emptiness, or sparseness, would be added to the `removeFirst` and `removeLast` methods. In the case of sparseness, the array would shrink by some amount.

What is the run time of `grow`? Well, you have to copy the contents of an array of size `capacity`. So the run time is $O(capacity)$. And since `grow` is called by `addFirst` and `addLast`, these methods are similarly $O(capacity)$. Same for `shrink`, `removeFirst`, and `removeLast`. Ugh. That’s not good. Inserting and deleting at the start or the end of a linked list takes only constant time. What gives?

Upon further reflection, observe that we do not have to grow or shrink our array with every insertion and deletion. On the contrary, we need only grow our array when it is full, and shrink it when it is sparse. So these expensive operations are in fact few and far between. This calls for a particular type of analysis. Do you recall (from @nameSemOne) which type? If you are thinking amortized analysis, then you are correct.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs018/feedback](http://cs.brown.edu/courses/cs018/feedback)