Lecture 11: Dynamic Arrays (ArrayLists)
10:00 AM, Feb 16, 2020

Contents

1 Lists via Arrays 1
   1.1 Putting the start somewhere other than 0 2
   1.2 Imagine the array as circular, rather than linear 2

2 Lists via Arrays, version 1 2

3 Implementation 3
   3.1 Common Case 7
   3.2 Edge Case 9

Credits: sections 2 and 3 were largely written by Professor Amy Greenwald

Motivating Question

How can we use arrays to implement lists, which allow insertion at either front or back and can have an arbitrary number of elements?

Objectives

By the end of this lecture, you will know:

- CS principles that help arrays efficiently behave as lists

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

- implement lists using a dynamic array

1 Lists via Arrays

Imagine that we wanted to use an array to manage a list of the names of CS18 staff. At first, we planned only to make a list of the HTAs, so we create an array of size 4. Assume that we’ve already added Evan and Carrie’s names, so we have the following picture:

```
<table>
<thead>
<tr>
<th>&quot;Evan&quot;</th>
<th>&quot;Carrie&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Now we want to run the following code:
Adding Isaac is easy, but adding Cody appears to require us to build a new array that has Cody at index 0 and everyone else shifted right by one position. Thus, adding to the front of the list seems to require linear time (in order to copy items to the new array). Is it possible to avoid this copying, and make element addition fast on either end?

### 1.1 Putting the start somewhere other than 0

If we wanted to leave room to add items at either end, maybe we should decide that the start is at an index in the middle of the array, rather than default the start position to 0. What if we decided to put the first element in the middle of the array, say at position 2? We could maintain a field `start` that stores the index of the first list element in the array.

This would work fine to add Evan and Carrie, but would then be out of space to add Isaac (but it could add Cody). Putting the start at index 1 (instead of 2) would have addressed this particular sequence of add operations, but we could have easily created a different sequence of operations that ran off either end of the array no matter where we put the start.

### 1.2 Imagine the array as circular, rather than linear

In addition to being flexible about where the start position is in the array, we could also reimagine the shape of the array. What if we instead imagined the array as being circular in shape? The we could grow either end as long as there is still room in the array. At that point, we would need to do something else to accommodate the extra elements. But we would at least have a way to use all of our allocated space regardless of where the list starts within the array.

What does “circular” mean in practice? Isn’t an array a linear structure in memory? Yes, but we can use arithmetic modulo to give the *illusion* of circularity. Modulo refers to the remainder under division: if you use modulo when doing arithmetic to compute array indices, the results effectively wrap around to their actual positions in the array. (In Java, the modulo operator is `%`).

### 2 Lists via Arrays, version 1

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 part of our original `IList` interface again, extended with a `get` method, since random access is easy using arrays:
public interface IList<T> {
    public int size();
    public void addFirst(T elt);
    public void addLast(T elt);
    public void removeFirst(T elt);
    public void removeLast(T elt);
    public boolean get(int index);
}

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 Java, get and remove methods throw a NoSuchElementException.

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).

3 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 $i$th item in the list is stored at index $\text{start} + i \mod \text{capacity}$.

Next, let’s address the question of how we might initialize start and 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 $\text{end} - \text{start} + 1$.

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

Okay, so that rules out initializing start and 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 $\text{end} - \text{start} + 1$, then if our list is empty (i.e., the number of items is 0), it follows that $\text{end} = \text{start} - 1$. So, if we initialize start to 10, then we should initialize end to 9. That’s easy enough.

But wait: the most natural way to initialize start is to 0. Then what? Well, in that case we cannot simply subtract 1 from start or we will arrive at a negative index into our array. No problem;
in this case, we can simply add capacity to start before subtracting 1. In fact, regardless of how we initialize start, we can initialize end correspondingly to start + capacity − 1, modulo capacity. (See Figure 2)

Based on this relationship, let’s revise our fourth invariant:

4. The number of items in our list is end + capacity − start + 1, modulo capacity.

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

1. The attribute end stores the index one more than where the list ends.
2. The number of items in our list is end + capacity − start, modulo capacity.

With these alternative invariants, we could initialize both start and end to 0 (or to any other non-negative integer value less than 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 start = 0 and end = 3. When start and 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 numItems, which stores the number of items in the array.

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

1. The attribute start stores the index where the list starts.
2. The attribute numItems stores the number of items in the list.
3. The ith item in the list is stored at index start + i modulo capacity.

Similarly, the start attribute contains redundant information given the end and numItems attributes. Nonetheless, we retain all three for clarity.

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

```java
class DynamicArray<T> implements List<T> { public class DynamicArray<T> implements List<T> { 
```
In Java, we are not able to declare an array to be of a generic (i.e., parameterized) type. In particular, the following declaration is illegal:

\[
T[\] \text{array} = \text{new T[1000];}
\]

Instead, we must declare an array of type \text{Object}, and then cast that array to be of type \text{T}. For example:

\[
T[\] \text{array} = (T[]) \text{new Object[1000];}
\]

But when you do this, Java warns you that you are casting, and that the type of your object will not be checked by the compiler. To suppress the warning, insert this line of code right above your method definition.

\[
@\text{SuppressWarnings("unchecked")}
\]

But remember: going around the compiler is generally bad practice. Suppress warnings with caution!

Next, we define the private helper methods \text{isEmpty} and \text{isFull} in our \text{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.

\[
\text{private boolean isEmpty() \{} \\
\text{ \text{ \text{return (this.numItems == 0); \}}} \\
\text{\} }
\]

\[
\text{private boolean isFull() \{} \\
\text{ \text{ \text{return (this.numItems == this.capacity); \}}} \\
\text{\} }
\]

At this point, it is straightforward to write \text{getFirst} and \text{getLast}:
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:

```
@Override
public T get(int index) {
    if (index < 0 || index >= this.numItems) {
        throw new IndexOutOfBoundsException();
    }
    if (isEmpty()) {
        throw new NoSuchElementException("List is empty");
    }
    return this.array[(this.start + index) % this.capacity];
}
```

### 3.1 Common Case

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:

```java
@Override
public void addFirst(T item) {
    this.start = dec(this.start);
    this.array[this.start] = item;
    this.numItems++;
}

@Override
public void addLast(T item) {
    this.end = inc(this.end);
    this.array[this.end] = item;
    this.numItems++;
}
```

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

```java
private int inc(int index) {
    return (index + 1 + this.capacity) % this.capacity;
}

private int dec(int index) {
    return (index - 1 + this.capacity) % this.capacity;
}
```

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

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.

3.2 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.

```java
@@SuppressWarnings("unchecked")
private void grow(int newCapacity) throw IndexOutOfBoundsException {
    if (newCapacity < numItems) {
        throw new IndexOutOfBoundsException
    }

    T[] newArray = (T[]) new Object[newCapacity];
    for (int i = 0; i < this.numItems; i++) {
        newArray[i] = this.array[(this.start + i) % this.capacity];
    }

    this.start = 0;
    this.end = this.numItems - 1;

    this.array = newArray;
    this.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(this.FACTOR * this.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. Tune in for part two of this lecture ...

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: [https://cs.brown.edu/courses/cs018/feedback](https://cs.brown.edu/courses/cs018/feedback)