Lecture 12: Wrap-Around Dynamic Arrays (ArrayLists)

11:00 AM, Feb 21, 2021

Contents

Motivating Question

How efficient can we make `addLast`? Can we make `addFirst` just as efficient?

Objectives

By the end of this lecture, you will know:

- The concept of amortized analysis
- How to reposition indices within an array

1 Recap of Where we Are

Last class, we realized that we have to be able to resize the underlying array when `addLast` runs out of room. We noted that doing this makes `addLast` be worst-case linear time, so we set out to improve on that.

Note that these notes are more detailed than what we did in class: they include the detailed mathematical analysis of the running time of `addLast` as well as the code modifications needed to implement wrap-around arrays in support of `addFirst`.

2 The Cost of Resizing Arrays

So far, you've learned about worst-case running time. Here, we certainly pay linear time when we have to extend the array. But adding a new element is only constant time if the array still has space.

How do we balance that out when talking about run time?

Instead of thinking about the cost of one operation, we're going to think about the cost of multiple operations together. Some of them take linear time, some take constant time. So let's ask ourselves: how much time gets taken on average, across the multiple operations? We call this *amortized* analysis, because we are distributing the cost of the expensive operations across the cost of the cheaper ones. That makes sense here, because we allocate extra space when extending the array specifically to speed up subsequent additions of elements.

If we make the array only one space longer when extending the array, then each addition (after the initial size) takes linear time. So if we did a sequence of n calls to `addLast`, each of which took linear time, that would be a total cost of $n * O(n)$. Dividing that by the number of operations is $n * O(n)/n = O(n)$. So this strategy has `addLast` take linear time amortized.

What if we add two spaces each time we extend the array? Then we would only do a linear-time operation on every other operation. That would give us an amortized running time of $O(n)/2$ per operation. That's an improvement, but it's still linear time.

What if we doubled the array size on each extension? Assume we've done several extensions and our array is now size n. What did we pay in copying costs to get here?

$$n + n/2 + n/4 + n/8 + \ldots + 2 + 1$$

Adding this all up gets to a total cost of $2 * n$. In addition, we paid a constant amount of time to add each item to an empty space. So the total time for adding n elements is $3 * n$. If we average that out across the n operations, we see that we paid amortized constant time to add elements.

Amortized constant time doesn't erase the worst-case linear time: we will still pay that cost occasionally. The point here is that we are paying that cost to enable other operations to be cheaper. And if we distribute those costs, we see that the distributed cost is no worse than if we set aside all that space up front. From an amortized analysis perspective, the cost of `addLast` isn't bad.

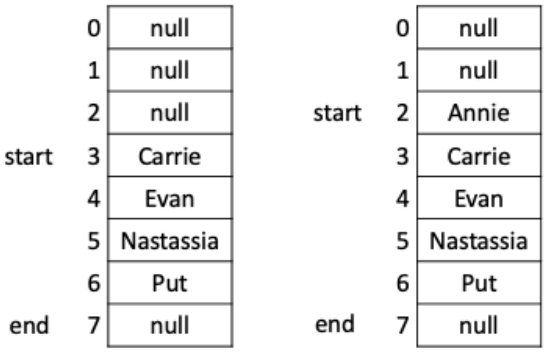
The big idea to take from amortized analysis is that it applies to a *sequence of calls to a method*, not to a single call. Worst-case time considers single calls to a method. There is also a concept known as *average-case time*, which deals with the average run-time on different inputs (but does not consider them as a sequence). You'll study average-case analysis if you go on to take CS1570 (Algorithms).

3 Providing `addFirst`

So far, our `ArrayBasedList` class is only supporting adding new items to the end of the array. What if we want to add to the front of the array as well, with an `addFirst` method. Since arrays need the items to be consecutive in memory, this suggests that adding to the front of the array requires moving all of the elements down one space, then putting the new element at the top. This is again linear time!

Maybe not. What if we left ourselves some blank space at the front of the array to add new first elements? For example, we might initially make an array of size 8 to hold our TAs, but put the first TA in at position 2. Then we'd have room to add TAs at either end! Let's try that with our HTAs. In the figure below (left), we set an array to size 8 with the start at index 3. We use `addLast` to add our four HTAs, then we use `addFirst` to add `Annie` to our list of TAs. By putting the start in

the middle of the array, we have room to do that, as shown in the center figure. Note we still have room on both ends of the array.



Let's keep going! Let's add `Joe` to the end of our array. Now what happens? Notice that the `end` marker has run off the end of the array, which means we'll need to extend the array the next time we try to `addLast` (say to add `Peter` to the array).

Or do we? Notice we still have some extra space in the top two positions of the array? Could we somehow use those by "wrapping around" the end into the top of the unused space? We sure can! We just have to adjust the end marker so that it wraps around to the unused spaces:

Let's focus on understanding this conceptually before we turn to the code. It might help to think of the top and the bottom of the array "glued together" into a cylinder (or gear) of slots: we're just rotating the slots backwards to make room for the new element. It's as if there were a phantom index 8 that actually lies in position 0 and a phantom index 9 that actually lies in position 1. In the above picture, we've used red italic numbers on the right of the array to label the conceptual positions in the list, showing how they differ from the array indices on the left.

With this approach, we can allow addition on both ends of the array, while also making sure we've used all available space before extending the array.

3.1 Implementing WrapAround Arrays

Let's look at how our code changes to allow this. Let's start with the `get` method. Recall that currently looks like:

```
String get(int position) {
  if ((position >= 0) && (position < maxSize))
    return contents[position];
  else
    throw new RuntimeException("position " + position + "out of bounds");
}
```

Someone using our code might ask for the first TA in the list (which should be Annie). Since they are asking for the first TA, they will write `ourTAs.get(0)` (remember, we count positions from 0). We know that the actual list actually starts in index 2, however. So in response to `ourTAs.get(0)`, we should return `contents[2]`. If the user asks for `ourTAs.get(1)`, we should return `contents[3]`. And so on.

Generally speaking, our code has to convert from the position the user wants to the array index where that position actually is. We do that by adjusting the position that the user has asked for to the correct index by adding the value of `start`:

```
  contents[position + start]
```

What should happen if the user calls `ourTAs.get(6)`? Our current expression would then look up `content[6 + 2]`. But there is no index 8 into our array – this would give an "out of bounds" error!

We need a way to compute the correct index while "wrapping around". A request for position 6 should retrieve the value at index 0 (since `start`) is 2), as seen in the previous picture (the italic red 6 is actually at index 0).

If you remember modular arithmetic, that's all we need here. If you are unfamiliar or rusty with this concept, it boils down to the remainder under integer division. Consider `position + start` where `position` is 6 and `start` is 2. Naively, we would ask for `contents[8]`. The max index within the array is 7 (one less than the capacity of the array). Let's divide the naive index by the size of the array (here, 8/8). The remainder in this division is 0. And that's the index of the desired element! If instead we had wanted the element in position 7, we would compute the remainder of $(7 + 2)/8$, which is 1 (the index corresponding to position 7).

The remainder under integer division comes up frequently enough in programming (in cases such as this with arrays) that language build in an operator, called *modulo* for computing this remainder.

In Java, this is written with a percent sign. Our get method therefore needs to look like:

```
String get(int position) {
  if ((position >= 0) && (position < maxSize))
    return contents[(position + start) % maxSize];
  else
    throw new RuntimeException("position " + position + "out of bounds");
}
```

Stop and Think: Now that we have modular arithmetic, do we still need the `if` statement to check whether the position is within the size of the array?

3.2 Revising `addLast`

The wrap-around adjustment that we did using modular arithmetic in `get` has to be done in any part of the code that deals with indices into the array: if we might move off the edge of the array (on either side), we have to use modulo to put our indices back within the valid indices of the array. Both `addLast` and `addFirst` do such an adjustment when they adjust the `end` and `start` fields, respectively.

Here's our updated `addLast` method:

```
ArrayBasedList addLast(String newelt) {
  if (eltCount == maxSize)
     this.resize();
  contents[end] = newelt;
  end = (end + 1) % maxSize;
  eltCount = eltCount + 1;
  return this;
}
```

We have made two changes from the earlier code: we have moved the code to resize the array into its own method (because we will need that same code in `addFirst`), and we have used modulo to adjust the updated `end` location back within the edges of the array.

3.3 Wraparound in `addFirst`

The `addFirst` method would need a similar adjustment on the `start` field. We might expect to write

```
  start = (start - 1) % maxSize;
```

If you write this, you will (probably) be surprised to get array out of bounds errors that report that `start` is -1 if it moves off the top edge of the array. This is an artifact of how Java handles division of a negative number: if `start` is 0, this code will produce -1, even though that isn't the correct answer mathematically. To fix this, we leverage the fact that for any number n, both n and $n + maxSize$ have the same remainder when divided by $maxsize$. Thus, we write:

```
  start = (start + maxSize - 1) % maxSize;
```

At this point, you might expect that `addFirst` (which we haven't provided yet) looks basically the same as `addLast`, but there's a subtlety that `addFirst` has to deal with that `addLast` does not. It has to do with how we initialize the `start` and `end` fields (both to 0).

If you want a really good exercise, think about how the initial values affect `addFirst`. Then look at the posted solution code. Ask yourself which parts of that code are there to support wrap-around, and which would be there whether or not we implement wrap-around.

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`.

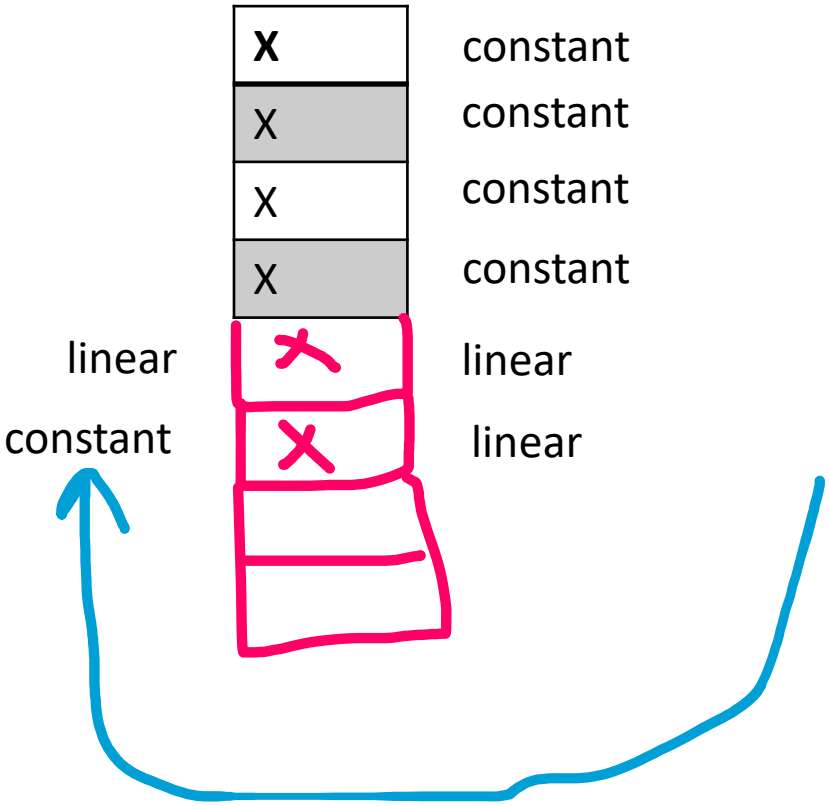
Recap

- When array runs out of space and we want to addLast, we have to:
  - create new (larger) array -- constant
  - copy over existing items from old array to new one
    - linear time
  - then add new element – constant

  Can we avoid the linear cost?

  What happens on addFirst?

Don’t pay linear cost EVERY time we addLast. Only pay when array is out of space. Worst-case running time might be overly pessimistic.



What was the cost of adding 6 elements to an array that was initially size 4?

- 4 constant + 2 linear

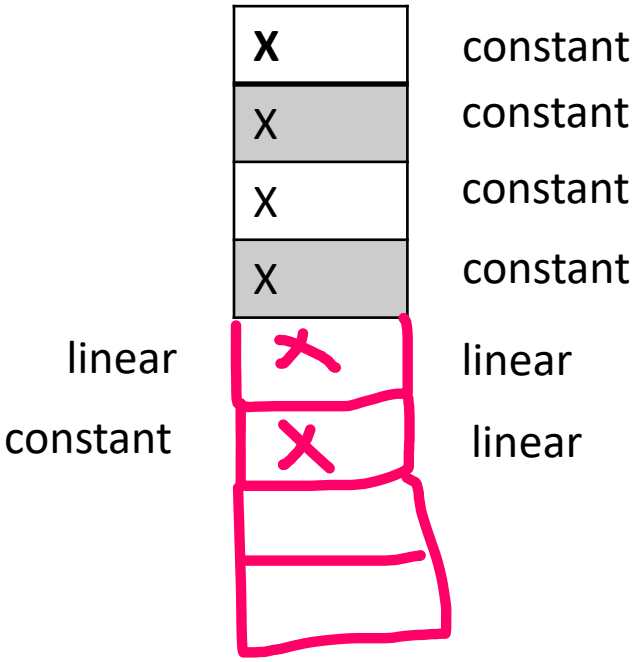
What was the AVERAGE cost of those adds?

4 + 5 + 6 = 15 / 6 – not quite linear for all

What if we added more than 1 new slot each time addLast ran out of room? Let’s add 2 spaces each time.

Now, to add n elements, we only pay linear cost (roughly) half of the time, which gives a running time of n/2 (across all adds)

Amortized run-time is the average cost across multiple uses of the same method. addLast has linear worst-case time, but amortized can be better. How much better?



- If add 2 slots per resize, amortized is n/2
- if add 10 slots, amortized is n/10
- and so on

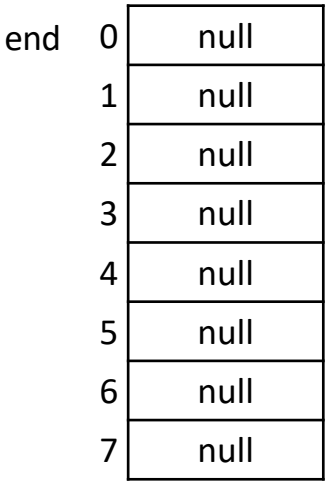
in terms of liniear, constant, etc, what time is n/10? actually, it's still linear!

- what if **DOUBLE the # of slots** each resize?

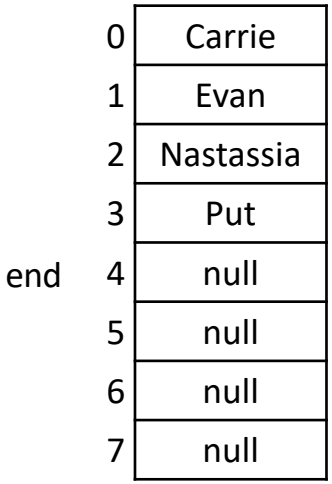
start with k elements
k constants + k + k constants + 2k

- if you sum this up and average over final array size, this is amortized CONSTANT

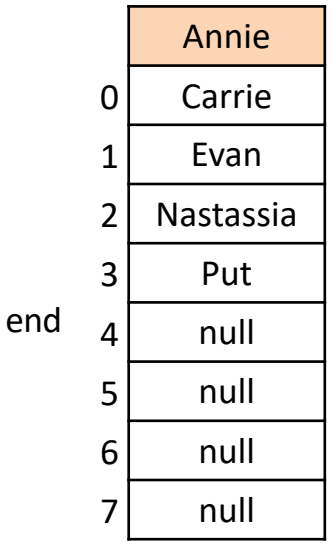
empty/initial array



after several addLast calls



try to addFirst (out of bounds)



naive addFirst shift/copy elts

Can leave empty space up front, but messes up get

what if we also tracked start?

get(i) becomes array[start + i]

now addLast("emily")

are we actually out of space?

wrapping around indices to use all available slots before resizing

Emily is in conceptual index 6 (7th element)

Under the hood, she's in index 0

get(6) → array[6+2]

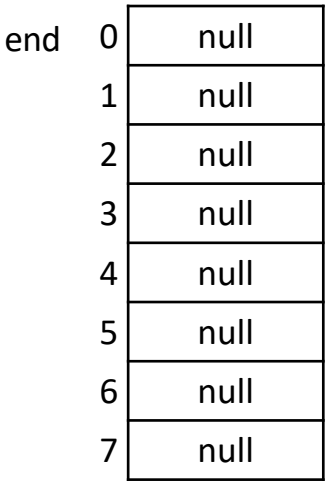
array[8] off the end

array has 8 elements

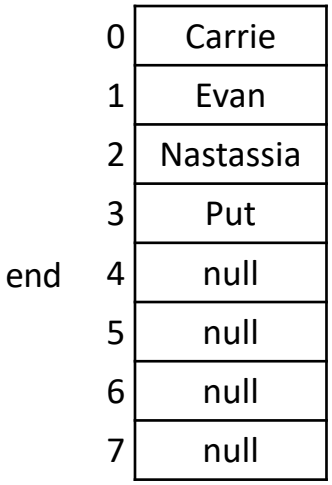
(index + start) mod size

look in array[0] for get(6)

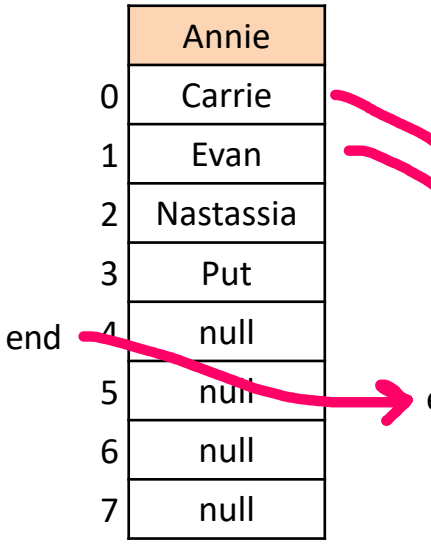
empty/initial array



after several addLast calls



try to addFirst (out of bounds)



naive addFirst shift/copy elts

Can leave empty space up front, but messes up *get*

what if we also tracked the first-elt location (as *start)*?

get(i) becomes array[start + i]

first elt of list can be stored in middle of array

wrap-around to use empty cells at front

resize retains order

```
public class LinkList implements IList {
   public int length(int elt) {
      int count = 0;
      Node current = this.start;
      while (current != null) {
         count = count + 1;
         current = current.next;
      }
      return count;
   }
}
```

```
public class LinkList implements IList
{
   public boolean contains(int elt) {
      Node current = this.start;
      while (current != null) {
         if (current.item == elt) {
            return true;
         }
         current = current.next;
      }
      return false;
}
```

Same parts are what programmer will stil write
Different or missing from while version must be under the hood in Java

We want to be able to write the for-loop versions (below) instead of the while loop ones (above)

- Which parts of the while versions remain in the for-loop versions?
- Which parts get done "automatically" by the for-loop?
- Propose names for each part that seems to be done automatically

```
public class LinkList implements IList {
   public int length(int elt) {
      int count = 0;
      for(Integer item : this) {
         count = count + 1;
      return count;
   }
}
```

```
public class LinkList implements IList {
   public boolean contains(int elt) {
      for(Integer item : this) {
         if (current.item == elt) {
            return true;
         }
      return false;
   }
}
```