Lecture 11: Dynamic Arrays (ArrayLists)

11:00 AM, Feb 16, 2021

Contents

Motivating Question

What happens when an array runs out of space? How do we accommodate new elements efficiently in terms of both time and space?

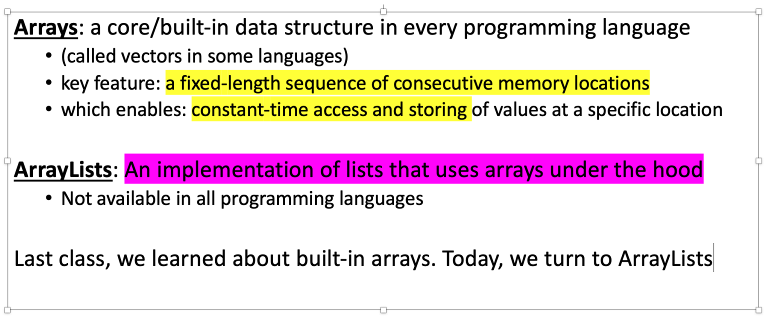
Objectives

By the end of this lecture, you will know:

- What to do when an array needs more slots than it has

1 Recap of Where we Are

Last class, we started learning about two related data structures. Here's the slide summarizing where we are:



Arrays: a core/built-in data structure in every programming language
- (called vectors in some languages)
- key feature: a fixed-length sequence of consecutive memory locations
- which enables: constant-time access and storing of values at a specific location

ArrayLists: An implementation of lists that uses arrays under the hood
- Not available in all programming languages

Last class, we learned about built-in arrays. Today, we turn to ArrayLists

2 Resizing Arrays

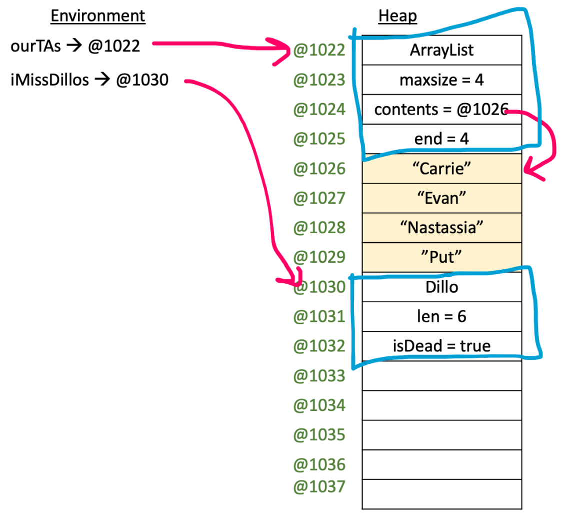
Imagine that we wanted to use an array to manage a list of the names of CS18 staff. At first, we planned only to make a list of the HTAs, so we create an array of size 4. But in the process of adding TAs to the array, we also get nostalgic and create a `DIllo`.

```
  ArrayBasedList ourTAs = new ArrayBasedList(4);
  ourTAs.addLast("Carrie");
  ourTAs.addLast("Evan");
  Dillo iMissDillos = new Dillo(6, true);
  ourTAs.addLast("Nastassia");
  ourTAs.addLast("Put");
```

As a reminder, our `addLast` method appears as follows:

```
// add given item to the end of the array
ArrayBasedList addLast(String newelt) {
   contents[end] = newelt;
   end = end + 1;
   return this;
}
```

Running the code results in the following memory contents:



Now we want to start adding one of the UTAs:

```
  ourTAs.addLast(''Joe'');
```

As written, our `addLast` method will insert `''Joe''` into the address computed as `ourTAs.contents + ourTAs.end`, which is location `@1030`. But that address is the start of the `Dillo` object. We don't want to destroy the `Dillo` when adding `''Joe''`, and indeed if we tried this sequence with regular arrays we would get an exception that we tried to access the array "out of bounds".

Upshot: the array no longer has space to add more TAs. But we are trying to build a list implementation on top of arrays. To a programmer, one can always add items to a list. So if we want to use arrays to implement lists, we have to be able to adapt to this situation.

Can we somehow put `''Joe''` in the next available location (`@1033`) and tell the existing array to look there? No, that would violate the fundamental property of arrays that all elements are in consecutive order. Without that property, we can't compute the location where a specific index into the list lies (which is what makes the `get` method constant time).

Our only option here is to make a new, longer, array with enough space for our additional TA. We'll create the longer array, copy the old array contents to the new one, then insert the new element in the additional space. This means the `addLast` method will look like the following:

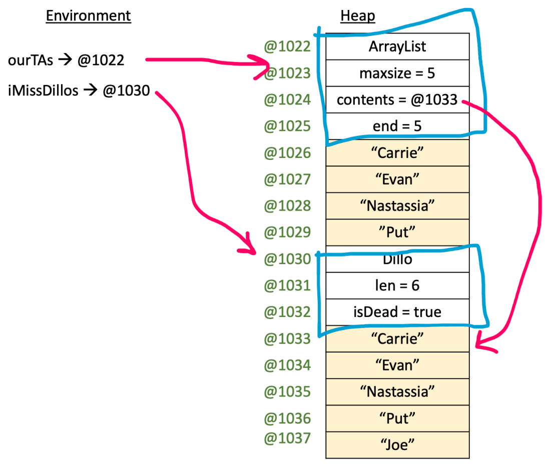
```java
ArrayBasedList addLast(String newelt) {
  if (eltCount == maxSize) { // we're out of space

    // make new larger array
    int newMaxSize = maxSize + 1;
    String[] newContents = new String[newMaxSize];

    // copy old elements over to new array
    for (int index = 0; index < maxSize; index = index + 1) {
      newContents[index] = contents[index];
    }

    // adjust maxSize and contents to match new array
    maxSize = newMaxSize;
    contents = newContents;
  }
  contents[this.end] = newelt;
  end = end + 1;
  eltCount = eltCount + 1;
  return this;
}
```

With this version of `addLast`, Java stops throwing the "index out of bounds" error. The resulting memory contents appear as follows:



Note: *This can be a really nice piece of code on which to practice working with the debugger: step through* `addLast` *and watch how the arrays copy and various fields update.*

What happens to the old array? Several of you asked whether the old array (at `@1026` just sits there being unused. It won't get used again – there's no way to get to it from the environment. But Java will eventually detect that we're no longer using it and return that memory for use by another part of the program. That process is called *garbage collection*. We'll discuss that in more detail towards the end of the course.

2.1 Running Time of `addLast` – Part 1

Our original `addLast` code was constant time: since Java can get to a location in an array in constant time, inserting an element takes constant time. What about this new version that handles resizing?

Once the array has filled up, notice that each subsequent call to `addLast` is linear time (because we have to copy the old array over to the new one). That's unfortunate. How could we avoid that?

One proposal is to just predict how much data you will eventually have, and set aside a large enough array up front. Sometimes that works. But sometimes it doesn't: we don't always know how much data we will have. And besides, that can lead to poor space usage. If I create an array with 1000 spaces (for example) but only ever use 4 of them, then we've wasted 996 memory slots (not much in practice in this specific case, but there's a general principle here).

A more practical proposal is to add space for a few elements at a time: don't just add one slot when resizing. But how many should we add? 5? 10? And what impact would our choice have on the run time?

This is where we will pick up next lecture. Specifically, we have two questions to consider:

1. How much should we grow the array by when extend it? Can we make a smart choice that saves the linear time cost?
2. And what about `addFirst`? There's never extra room at the beginning, so even if we fix `addLast`, is `addFirst` doomed to linear run time?

Tune in next time ...