Lecture 6: ArrayLists

Sam McCauley
Data Structures, Spring 2026

An ArrayList of Ints

The goal for today is to use Java classes to recoup the functionality of a "list" as you may have seen in other languages like Python. Essentially, we want this data structure to work like an array, but without this annoying restriction of a fixed size. We'll start by listing out the functionality we want. Then, we'll figure out a strategy to implement that functionality. Once we're done, we'll have an `ArrayList` class that we can interact with much like a Python list—so long as the list is only storing integers. In a couple lectures, we'll use a new Java concept, Generics, to generalize our code to be able to store any kind of data.

Java has a built-in `ArrayList` class that we will eventually be using. Today, our goal is to build a similar class from scratch to help us understand *how* this class works. Understanding how this class works will be crucial for discussing its efficiency—which in turn leads into better knowledge of when to use an `ArrayList` compared to another data structure.

List Functionality

Let's discuss some of the operations a list should support. We will be implementing all of these as methods in our `ArrayList` class. We will name our methods to match the Java `ArrayList` methods, so they may not match the name of methods you've seen in a language like Python.

First, let's make sure it matches the functionality of a normal array: we should be able to look up its size, and access and modify the entry stored at a given index of the array.

- `int size()`: returns the number of elements stored in our list
- `int get(int index)`: returns the element stored in slot `index`
- `int set(int index, int newElement)`: sets the element stored at `index` to be `newElement`. Returns the element *previously* stored at `index`.

Now, we want to be able to add new elements to the list, expanding its size. One option is to add the new element at the end of the list. Alternatively, we'll allow adding the new element at any position in the array, pushing elements down to make room. We'll also give an operation to remove an element from the list.

- `void add(int newElement)`: adds `newElement` to the end of the list.[1]
- `void add(int index, int newElement)`: adds `newElement` to slot `index` in the list, pushing all later elements in the list down by one slot.
- `int remove(int index)`: removes the element at index `index`, moving all later elements in the list up by one slot. Returns the removed element.

Finally, it is very useful to be able to search for a specific element in our list. Let's add some methods that can do that.

- `int indexOf(int element)`: returns the first index in the list that contains `element`, or -1 if there is no such index (the list does not contain the element).
- `boolean contains(int element)`: returns `true` if `element` is in the list, and `false` otherwise.

Strategy

Let's lay out a plan for what our data structure will look like before we start filling in methods.

What Data Do We Store? We need to store all of the elements in our `ArrayList`. We will do that using an array of `int`s. This will allow us to easily keep the elements in order, and access and change the element in a given slot. We will call this array `arr`.

Once we discuss array resizing, we will see that we will need to also store an integer for the number of elements stored in our list. We will call this `numElems`.

What to Do When We Run Out of Space. We need a strategy for `add()`: what do we do when we want to add a new element, and there is no space for it?

In Java, an array cannot be resized. We are left with only one option: we must *rebuild* the array, copying the elements over one by one. The only question is how big the new array should be.

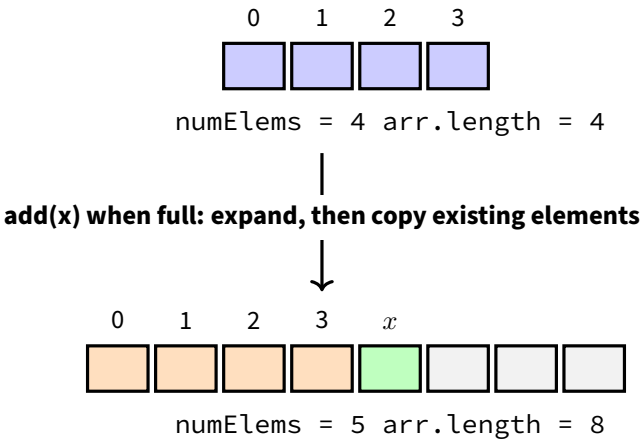
First Attempt: Grow the Array to Fit. One idea is to grow the array by 1, to handle the new element—we'll make the size of the new array one larger than the size of the old array.

There are advantages to this strategy. In particular, at all times, all slots of the array store an `int`; we are very space-efficient. Our bounds checking is also easy if every slot of the array is a legal slot to access.

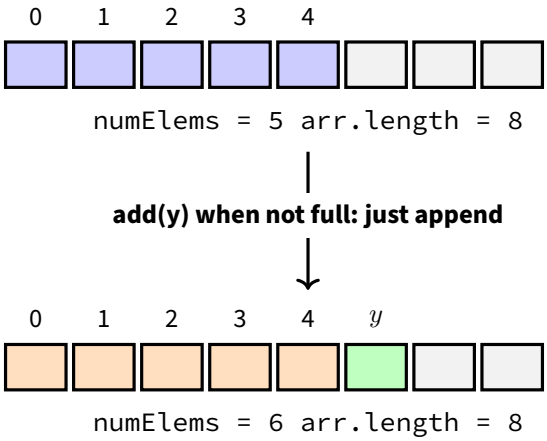
[1] In the real `ArrayList`, add returns a `boolean` value that is always `true`, for reasons that don't affect our implementation.

However, this strategy has a downside: every time we call `add()`, we need to copy the *entire array*. This is no big deal for a relatively small array of 10 elements, or even 1000 elements. But for arrays of millions or billions of elements, each `add()` will take a noticeable amount of time.

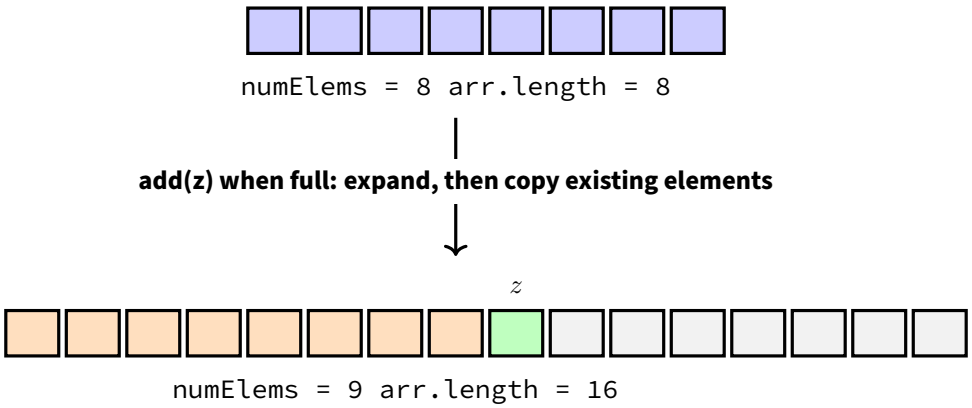
Second Attempt: Double the Size of the Array. The second idea is to grow the array much more aggressively. Every time we `add()` an element, if there is not enough room in the array, we *double* its size.



Now, we can handle many `add()` method calls without changing the length of the array, until array is full again.



When the array is again full and we `add()`, we'll double the array size again.



This strategy changes how we interact with the array. While some slots of the array store an `int`, some slots of the array are "empty"—they are only there to hold future elements if the list expands. When we implement a method like `get()`, we must be sure to only allow the user to interact with the array slots that hold inserted elements, and not the empty slots.

The advantage of this strategy is that we need to grow the array much less often. If we have an array of size 64, and the user adds a 65th element, we create an array of size 128, so we can insert 64 more elements before we grow again.

There are two disadvantages. First, as mentioned, we'll need to do a little extra bookkeeping internally; however, this will be invisible to the user. Second, this data structure does use extra space. However, it's not too bad: our array is only twice the size it needs to be after we `add()` a new element.

Comparison of the Approaches. Let's put some numbers on the two approaches mentioned so far. It seems intuitive that copying into an array 1 size larger every time is much more expensive than doubling the size of the array every time, but let's give an example to look at this quantitatively.

Consider calling `add()` 1 million times. We will compare how many elements are copied for each approach.

If we grow the array by 1 every time, we will copy 1 element, then 2, then 3, then 4, up to 999,999. The total number of elements we copy is:

$$1 + 2 + 3 + \ldots + 999,999 = 499,999,500,000.$$

So just short of 500 billion elements copied.

If we double the size of the array every time, we will copy 1 element, then 2, then 4, then 8, then 16,

up to 524,288. The total number of elements we copy is:

$$1 + 2 + 4 + 8 + \ldots + 524,288 = 1,048,575.$$

So our total number of elements copied is just over 1 million. This is 500,000 times less than the first strategy.

Being 500,000 times faster is well worth the extra space usage in this case. And for larger lists, the comparison is even more drastic. Even for small arrays, it is likely worth it: if we grow to a list of 10 elements one at a time we perform 45 copies; doubling every time uses only 15.

(Optional Topic) Repeated Doubling Analysis. Later in the class, we will analyze this "repeated doubling" strategy, and we will show that the total number of copies performed is never more than twice the number of elements in the array. This means that the cost of copying is at most 2 per operation "on average."

(Optional Topic) Repeated Doubling For Library ArrayList. The Java `ArrayList` library class actually does not do repeated doubling: it multiplies the size of the array by 1.5 each time. This is a slightly different tradeoff: the number of copies is a bit larger than repeated doubling, but the extra size is a bit smaller. Nonetheless, the above idea holds—the number of copies multiplying the size by 1.5 each time is far less than it would be adding 1 each time.

(Optional) Handling Removals. If we remove many items, our strategy becomes space inefficient. Let's say we grow to 1 million elements, but then remove all but one of them. Our algorithm would then take up far more space than is necessary.

The way to solve this is to reverse the doubling method. If we remove an item, and the number of elements is *less* than half the array slots, we copy the elements to a new array of half the size. We will not do that here, and in fact the Java library `ArrayList` does not either—though the library version has a method, `trimToSize()`, that does shrink the internal size of the array.

Filling In the Methods

We're now ready to fill in our class. We'll go piece by piece, and discuss each method we write.

First, we set up the class and fill in the instance variables.

```
1 public class ArrayListInt{
2     private int[] arr;
3     private int numElems;
```

Let's fill in the constructor as well. We'll start with `arr` having 1 element.[2]

```
public ArrayListInt(){
    arr = new int[1];
    numElems = 0;
}
```

Now let's go through our methods one by one.

`int size()` returns the number of elements stored in our list.

```
public int size() {
    return numElems;
}
```

`int get(int index)` returns the element stored at slot `index`. We can easily do this by accessing the array. However, we have to be careful: we don't want to return the "empty" entries of the array. We should give an error when an incorrect entry is accessed. (We haven't gone over how to properly handle things like errors and exceptions in Java, so let's just display an error message and quit.)

```
public int get(int index) {
    if(index < 0 || index >= numElems) {
        System.out.println("Error: Array access out of bounds at index
            " + index);
        System.exit(1);  //exit the program
    }
    return arr[index];
}
```

`int set(int index, int newElement)` sets the element stored at `index` to be `newElement`, and returns the element *previously* stored at index `i`.

```
public int set(int index, int newElement) {
    if(index < 0 || index >= numElems) {
        System.out.println("Error: Array access out of bounds at index
            " + index);
        System.exit(1);  //exit the program
    }
    int ret = arr[index];
    arr[index] = newElement;
    return ret;
}
```

Wait a minute—the error checking for `get` and `set` is exactly the same. We should factor this out into a new method. Let's make a method to check if an index is in bounds, and output that there is an error if so. This is an internal method, so we'll make it `private`. Then we can refactor `get` and `set`.

[2] The Java library version of `ArrayList` starts with 10 to avoid extra work during early rebuilds.

```
private void checkInBounds(int index) {
    if(index < 0 || index >= numElems) {
        System.out.println("Error: Array access out of bounds at index
            " + index);
        System.exit(1);  //exit the program
    }
}

public int get(int index) {
    checkInBounds(index);
    return arr[index];
}

public int set(int index, int newElement) {
    checkInBounds(index);
    int ret = arr[index];
    arr[index] = newElement;
    return ret;
}
```

Now, we want to be able to add new elements to the list, expanding its size. Remember that we'll use our strategy above: growing the array when it is not large enough. We're going to use this strategy each time a new element is added—so again, this strategy should be split off into a helper method.

We'll call this method `ensureCapacity`. It takes an `int minCapacity` as a parameter, checks to see if `arr` has capacity at least `minCapacity`; if not, it doubles the size of `arr` and copies the elements over.

```
private void ensureCapacity(int minCapacity) {
    int size = arr.length;
    while(size < minCapacity) {
        size *= 2;
    }
    int[] newArr = new int[size];
    for(int x = 0; x < numElems; x++) {
        newArr[x] = arr[x];
    }
    arr = newArr;  //set arr to "point to" newArr
}
```

A note about `ensureCapacity`: the `ArrayList` implementation in the Java library actually has this as a `public` method. This is an interesting tradeoff. Because it is `public`, programmers can set the size of the internal array themselves, possibly increasing efficiency. But in exchange, the internal code can no longer guarantee that the size of the internal array is similar to the number of elements.

Let's start writing the rest of our methods.

`void add(int newElement)` adds `newElement` to the end of the list.

```
public void add(int newElement){
    ensureCapacity(numElems+1);
    arr[numElems] = newElement;
    numElems++;
}
```

`void add(int index, int newElement)`: adds `newElement` to index `index` in the list, pushing all later elements in the list down by one slot. We'll actually do the push first, leaving room for us to insert the new element.

```
public void add(int index, int newElement){
    numElems++;
    checkInBounds(index);
    ensureCapacity(numElems);
    for(int pushIndex = numElems-1; pushIndex > index; pushIndex--) {
        arr[pushIndex] = arr[pushIndex - 1];
    }
    arr[index] = newElement;
}
```

`int remove(int index)`: removes the element at index `index`, moving all later elements in the list up by one slot. Returns the removed element.

```
public int remove(int index){
    checkInBounds(index);
    int ret = arr[index];
    for(int pushIndex = index; pushIndex < numElems-1; pushIndex++) {
        arr[pushIndex] = arr[pushIndex + 1];
    }
    numElems--;
    return ret;
}
```

Finally, let's add in our search methods. To search for an item in an array, we go one by one through the slots, checking if each slot contains the item we are searching for.

`int indexOf(int element)` returns the first index in the list that contains `element`, or -1 if there is no such index.

```
public int indexOf(int element) {
    for(int index = 0; index < numElems; index++) {
        if(arr[index] == element) {
            return index;
        }
    }
    return -1;
}
```

`boolean contains(int element)`: returns `true` if `element` is in the list, and `false` other-

wise. We already did most of the work for this—in fact, we can just call `indexOf` to complete this method.

```
1 public boolean contains(int element) {
2     return indexOf(element) != -1;
3 }
```