Lists

In the next part of the course, we are going to look at ways of representing and describing lists. We are all familiar with the concept of a list. You have a TODO list, a grocery list, etc. A list is different from a "set": the term "list" implies that there is a positional ordering. It is meaningful to talk about the first element, the second element, the i-th element of the list, for example.

There are many operations that are commonly performed on a list. Examples are getting or setting the i-th element of the list, or adding or removing the i-th element of the list, or adding (inserting) an element e in the i-th position of the list.

get(i) // Returns the i-th element (but doesn’t remove it)
set(i,e) // Replaces the i-th element with e
add(i,e) // Inserts element e into the i-th position
add(e) // Inserts element e (e.g. at the end of the list)
remove(i) // Removes the i-th element from list
remove(e) // Removes element e from the list (if it is there)
clear() // Empties the list.
isEmpty() // Returns true if empty, false if not empty.
size() // Returns number of elements in the list

In the next several lectures, we will look at some ways of implementing lists.

Arrays

In COMP 202, you should have learned about "arrays". You presumably solved problems such as finding the largest element in the array (assuming the elements were comparable, e.g. numbers), or checking if if there are any duplicate elements in the array. Here I will briefly review some concepts about arrays, in particular, arrays in Java.

An array has a capacity or "length", which is the maximum number N of things can be stored in the array. In Java, we can have arrays of primitive elements e.g.

```java
int[] myInts = new int[15];
```

which creates 15 slots that can store numbers of type int, and each is initialized to value 0. We can also define an array of objects of some class, say Shape.

```java
Shape[] shapes = new Shape[428];
```

This array can reference up to 428 Shape objects. At the time we construct the array, there will be no Shape objects, and each slot of the array will hold a reference value of null. I will say more below about what this actually means.
We are free to assign values to arbitrary slots of these arrays. We can write

```java
myInts[12] = -3;
shape[239] = new Shape("triangle", "blue");
```

and when we do this we think we have one element in the array. In fact, the array `myInts` doesn’t represent how many elements were written into it. The array is initialized to have value 0 everywhere, and we can change the values, but it would be wrong to think of the 0 values as empty.

It does make sense to think of the array of `Shape` array as having one object. However, note that we wouldn’t think of this array as being a *list* that contains one object, since that object is not at the 0 slot of the array.

**Arrays have constant time access**

Before I discuss how to use an array to represent a list, I would like to point out a central properties of arrays: the time it takes to access an element in an array does not depend on the length (number of slots) in the array. This is so, whether we are writing to a slot in an array,

```java
a[i] = ....
```

or reading from a slot in an array

```java
... = ... a[i] ....
```

You will understand this property better once you have taken COMP 206 and especially COMP 273, but the basic idea can be explained now. An array has a location (called an address) in memory which is just a number (like an apartment number, or a street number, or a phone number). This address specifies where `a[0]` is. To find out where `a[k]` is, you just add the address of `a[0]` to `k` times the size of each slot. This assumes that each slot in the array has constant size, which is indeed true. For example, an array of `int` will have slots that are the size of one `int`, namely 4 bytes. An array of objects will have slots that are each the size of a reference (called a pointer in the C programming language). In Java, the size of a reference is determined by how the ”Java Virtual Machine” for the computer is implemented, but no matter what that size is, it is constant for all slots.

**Using an array to represent a list**

Consider the list operations that I gave at the beginning of the lecture. Let’s suppose we have an array `a[ ]`. (For now I won’t specify the type because it is not important.) We want to use this array to represent a list. To do so, let’s say we have a list of *size* elements. The main idea is that we will keep the elements at positions 0 to *size*-1 in the array, so element `i` in the list is at position `i` in the array. Let’s look at the operations I described above and sketch out algorithms for them. Here we will not worry about syntax so much, and instead just write it the algorithm at the pseudocode.

The array length is `a.length` which refers to the number of slots. It is also called the *capacity* of the array. I emphasize that this is usually different from the *size*. (Again, *size* is only meaningful if the array is representing a list.)

Let’s first look at the simple operations of accessing an element in an array list by a read (get) or write (set).
get(i)
To get the i-th element in a list, we can simply do the following:

```java
if (i >= 0) & (i < size)
    return a[i]
```

Note that we are testing that the index \( i \) makes sense in terms of the list definition, and that \( size \) itself is well defined. To set the value at position \( i \) in the list, we can do this:

set(i, e)

```java
if (i >= 0) & (i < size)
    a[i] = e
```

Note that we are replacing an existing value here. If there was a previous value in that slot, it is now lost. We could alternatively return this previous element, for example,

```java
tmp = a[i]
if (i >= 0) & (i < size)
    a[i] = e
    return tmp
```

[ASIDE: in a Java’s List, the previous element is returned. No, I don’t expect you to understand what I mean by a Java List now. I just want to note it for the record.]

add(i, e)

Here we want to ”add” an element \( e \) to \( i \)-th position in the list. Rather than replacing the element at that position, which we did with a set operation, we displace (shift) the elements that are currently at index \( i, i+1, ..., size - 1 \). Here we can assume that \( i \leq size \), so if we want to add to the end of the list then we would add at slot \( size \). Otherwise, if \( i > size \), we would leave a gap in the array.

```java
if ((i <= size) & (size < length)){
    for (j = size; j > i; j--)
        a[j] = a[j-1] // shift to bigger index
    a[i] = e // insert into now empty slot
    size = size + 1
}
```

What happens when we want to add an element to a full array?

What happens if the first condition of the add ’method’ fails because the array is full, i.e. \( size >= length \) ? In this case, then we need to make a new and bigger array. Here is an algorithm for doing so.
// expand array if full
if (size == length){
    b = new array with 2 * length slots
    for (int j=0; j < length; j++)
        b[j] = a[j]
    a = b
    // Now the array is big enough, so we can use the code I presented above.
}

In the lecture, one of you suggested that this method is inefficient since adding the element could be done while copying. Here is how that would go:

if (size == length){
    b = new array with 2 * length slots
    for (int j=0; j < i; j++)
        b[j] = a[j]
    b[i] = e
    for (int j = i; j < size; j++)
        b[j+1] = a[j]  // elements must be shifted
    size = size + 1
    a = b
}

Adding n elements to an empty array

Suppose we are representing a list, and we start with an array of length 1. (We wouldn’t do this, but it makes the math easier to start at 1.) Now suppose we add n elements, calling add() n times.

    for i = 1 to n
        add( new element )

How expensive will this be? In particular, how many times will we fill up a smaller array and have to double its size? How many copies from small to large array do we need to do? It requires just a bit of math to answer this question, and it is math we will see a lot in this course. So let’s do it!

If you double the size k times (starting with size 1), then you end up with array with $2^k$ slots. So let’s say $N = 2^k$, i.e. $k = \log_2 N$. When we double the size of the underlying array and then fill it, we need to copy the elements from the smaller array to the lower half of the bigger array (and then fill the upper half of the bigger array with new elements). When $k = 1$, we copy 1 element from an array of size 1 to an array of size 2 (and then add the new element in the array of size 2). When $k = 2$ and we create a new array of size 4, and then copy 2 elements (and then eventually add the 2 new elements). The number of copies from smaller to larger arrays is:

$$1 + 2 + 4 + 8 + \ldots + 2^{k-1}$$
and note that the last term is $2^{k-1}$ since we are copying into the lower half of an array of size $2^k$.

As I showed last lecture (see the updated lecture notes),

$$1 + 2 + 4 + 8 + \cdots + 2^{k-1} = 2^k - 1 = N - 1$$

So using the doubling array length scheme, the amount of work you need to do to add $N$ items to an array is about twice what you would need to do if the array was already big enough. The advantage to using the doubling scheme, in general, is that you don’t need to know in advance how big the array needs to be!

I did not quite finish the material in the slides during the lecture, namely I didn’t present the remove method. Rather than presenting it at the start of next lecture, I will instead present it here and let you think it through on your own.

**remove**(i)

To remove an element, we again shift all elements by one position, but now we shift down by 1 rather than up by 1. The for loops goes forward from slot $i$ to $size - 2$, rather than backward from $size$ to $i + 1$ as in the add method on page 3.

[ASIDE: As you will discover for yourself when you write such algorithms and code them up, it is easy to make "off-by-one" errors. When testing your code, be sure to test the "edge" cases.]

```c
if ((i >= 0) & (i < size)){
    tmp = a[i] // save it for later
    for (k = i; k < size-1; k++){
        a[k] = a[k+1] // copy back by one position
    }
    size = size - 1
    a[size] = null // optional, but perhaps cleaner
    return tmp // optional
}
```

Finally, note that if the array has many elements in it, and if you add a new element to position 0 or you remove the element at position 0, then you need to do a lot of shifts. This seems inefficient. On the other hand, adding or removing from the other end of the list (near $i == size$) is fast, since few operations are necessary. We will return to this point in the next few lectures when we compare arrays to linked lists.