

List as an abstract data type (ADT)

We have considered three data structures for representing a list of elements: an array, a singly linked list, and a doubly linked list. Regardless of which of these data structure we use, operations such as adding or removing from the front or back of the list, or removing the i -th element of the list, or adding an element e before the i -th element of the list, etc, are all well defined. We can say what these operations do, without saying how the list is implemented. In this sense, a list is an *abstract data type* (sometimes called an ADT), namely a particular set of things and a particular set of methods that can be applied on/by/with these things.

Here are some methods that are typical for lists:

```
add(element)    // Adds element to the end of the list
add(i,element) // Inserts element into the i-th position
                // (and increments the indices of elements that were
                // previously at index i or up)
set(i,element) // Replaces the element in the i-th position
remove(i)      // Removes the i-th element from list
get(i)         // Returns the i-th element (but doesn't alter list)
clear()        // Empties list.
isEmpty()      // Returns true if empty, false if not empty.
size()         // Returns number of elements in the list
```

Java ArrayList

You have seen in COMP 202 that Java allows you to declare arrays, and these arrays can have either primitive or reference type, e.g.

```
int[]    intArray = new int[desiredSize];
:
Student[] studentArray = new Student[numStudents];
```

If you are using an array to maintain a list which can vary over time, then you will want to have methods for adding and removing elements such as we have been discussing. You don't want to write these list manipulation methods yourself. For this, Java has an `ArrayList` class. This class has a generic type, so you can define:

```
ArrayList<Student> students = new ArrayList<Student>(numStudents);
```

The `ArrayList` class implements a list using an underlying array, but you do not index the elements of the array using the `[]` syntax that you are used to. Instead you access an array element using a `get` or `set` method (see below). Some of the `ArrayList` methods were listed above. Here I list some of them again, and specify what happens in the underlying array implementation.

```
:
add(element); // Expands the underlying array, if it is full.
add(i,element); // Inserts a new element into the i-th position,
                // shifting up the positions of all elements with
```

```

remove(i)      // index >= i.
                // Removes the element at the i-th position,
                // shifting down the positions of all elements with
                // index > i.
a.clear();     // Makes a new (small) array with no elements or
                // alternatively sets all big array elements to null
a.size();      // returns number of elements in the list (NOT the
                // length of the underlying array)

```

It is important to realize that when you use the `ArrayList` class to represent a list, and you **add** or **remove** an elements to/from the *front* of your list, this operation will take time proportional to **size** (the number of elements in the list) since the index of every other element in the list changes. i.e. if you are removing then elements need to be shifted down and if you are adding then elements need to be shifted up.

Another important property of an *ArrayList* is that the underlying array has a certain length i.e. capacity. What happens if the underlying array is full and you try to add another element. In this case, the **add** method in Java's `ArrayList` class will creates a bigger underlying array¹ and it will *copy* all references from the existing array to this new underlying array. It will then add the reference to the new element. The old array will become garbage, and the new bigger array will be used instead. The user of the class will never see this though.

This hidden work of copying references from a small array to a big array takes time. How much time? How many steps are involved? Suppose you start with an empty list and you add n items. To keep the math simple, suppose that whenever you try to add to a full array, you first double the size of the array (i.e. expanding by 100%). If you double the size k times (starting with size 1), then you have an underlying array with 2^k elements. So let's say $n = 2^k$.

How does the amount of work that we need to do depend on n ? When we double the size of the underlying array, we then copy the elements from the small array to the big array. When $k = 1$, we just copy 1 element from an array of size 1 to an array of size 2. When $k = 2$ and we expand from the array of size 2 to the array of size 4, we copy 2 elements. In general, the number of copies we need to do is:

$$1 + 2 + 4 + 8 + \dots + 2^{k-1}.$$

But the sum is of the form

$$1 + x + x^2 + x^3 + \dots + x^{k-1} = \frac{x^k - 1}{x - 1}$$

where $x = 2$ and so

$$1 + 2 + 4 + 8 + \dots + 2^{k-1} = 2^k - 1.$$

Since we have expanded k times, we have an array of size 2^k . If we keep adding to fill that array (but no more) then we will have $n = 2^k$ elements in the array.

Thus, in adding n elements to an empty array list, we need to set n references (to set a reference to each element for the first time) and we also need to set $n - 1$ references (for the copying that needs to be done when we expand the underlying arrays k times). This leads to $2n - 1$ reference settings in total. Thus, doubling the size of a filled array and copying the references does require some work, and this extra amount of work grows with n .

¹bigger by 50% in Java

Java LinkedList

Suppose you will be inserting and deleting a lot at the beginning of the list. In this case, if you care about efficiency then you probably want to use a linked list as the underlying data structure, as opposed to an array. Java has a `LinkedList` class which is implemented as a doubly linked list.

```
LinkedList<Student> studentList = new LinkedList<Student>();
```

and then do the usual things with these lists:

```
studentList.add(new Student("Albert Einstein"));
:
studentList.add(1, new Student("Emmy Noether"));
```

If we add n students to the front (or back) of the list, then this takes about cn steps, where c is the number of steps required to set the `prev` and `next` references in the nodes of the underlying data structure, e.g. `dummyHeader`'s `next` reference and the `prev` reference of the node `dummyHeader.next`, as well as the `next` and `prev` references of the node we are adding i.e. c is 4.

Another example: Suppose we have a list of n elements which is implemented using a linked list, and suppose we wanted to print out the elements. What if we were to define a `display` method that prints each of the elements. At first glance, the following pseudocode would seem to work fine.

```
for (j = 1; j < n; j++)
    print( myList.get(j) )           // or the equivalent Java statement
```

For simplicity, suppose that `get` is implemented by starting at the head and then stepping through the list, following the next reference. Then, with a linked list as the underlying implementation, the above code would require

$$1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$$

which grows like n^2 (bad!) What went wrong? The j^{th} `get` starts again at the beginning of the (linked) list and walks to the j^{th} element, which is very inefficient.

What alternatives do we have? In Java, we can use something called an `Iterator`. We will discuss these later in the course.

Comparison of ArrayList and LinkedList: worst case

Let's look at the *worst case* performance of the two implementations of lists, namely the number of steps we need to take. Here I ignore constants c , that is, I don't distinguish $n/2$ from n or $2n$ and I don't distinguish 1 from say 2 or 3. (You will understand this better when we get to "Big O" in early February.)

	LinkedList	ArrayList
<code>add(element)</code>	1	1
<code>add(i,element)</code>	n	n
<code>set(i,element)</code>	n	1
<code>remove(i)</code>	n	n

<code>get(i)</code>	<code>n</code>	<code>1</code>
<code>clear()</code>	<code>1</code>	<code>1</code>
<code>isEmpty()</code>	<code>1</code>	<code>1</code>
<code>size()</code>	<code>1</code>	<code>1</code>

[ASIDE: I mentioned in class that for a doubly linked list, you really only need $n/2$ steps to access a position, in the worst case. If the item you want to access is in the front half of the list ($index < size/2$), then you search for it starting the head; if the item is in the back half of the list then you search for it starting from the tail.]

ADT's in Java: the interface

At the beginning of the lecture, we discussed what a list was in an abstract sense: a set of things and a certain set of methods (operations) that are applied to these things. Stated in this general way, a *list* is an abstract data type (ADT). We will see two more ADT's next lecture, namely the stack and the queue.

In Java, one does not use the term ADT. Rather, there an entity called an **interface**. An interface is set of methods, defined formally by a return type, a method name, and method argument types in a particular order (together called the *signature* of the method. An interface does *not* and **cannot** include an implementation of the methods. Rather, one needs to define a class that implements the interface. In Java, for example, there is a `List` interface (look it up in the Java API), and this interface is implemented by the `ArrayList` and `LinkedList` classes. We will see more examples of interfaces throughout the course.