Lecture 10 – ArrayLists

Review: Continuing from last lecture–memory layouts of lists

Consider the following two layouts for the list [8, 3, 6, 4]:

loc 1012	**MutableList**(start:loc1017)
loc 1013	Node(item:6, next:loc~~1014~~1016)
loc 1014	Node(item:3, next:loc1013)
loc 1015	Dillo(length:5, isDead:true)
loc 1016	Node(item:4, next:null)
loc 1017	Node(item:8, next:loc1014)
loc 1018	

get(2)
get(1)

get(3)
get(0)

What sequence of calls created this list?

addLast(6) - or addFirst
addFirst(3)
addLast(4)
addFirst(8)

loc 1012	**ConsecList**
loc 1013	8
loc 1014	3
loc 1015	6
loc 1016	4
loc 1017	
loc 1018	

get(0)
get(1)
get(2)
get(3)

These two kinds of lists organize elements differently. As such, they will need different implementations of the core list methods (get, addLast, etc). For now, consider get, which takes a position in the list and returns the element at that index.

- In the first organization (MutableList, but also applies to LinkList), the elements aren't in order, so we have to follow references until we count off to the position we asked for. This means get(index) is linear in the size of the list (since we could have asked for the last index).

- In the second (ConsecList), we know exactly where each position is stored. Better still, we can compute the address where a position lies. If we want position p, for example, it is in address

```
ConsecList-addr + 1 + p
```

This means that get is constant time in a ConsecList.

The rest of this lecture and most of next will be on how to build ConsecList.

Arrays

How do we ask Java for several consecutive memory locations (as we need to build ConsecList)? We use something called an **Array**. Arrays exist in practically every programming language.

Let's see how arrays work in general with an example, then use them to build lists. Check the comments in the code to understand how it works.

```
import java.util.Arrays;

public class Main {
   public static void main(String[] args) {
      // square brackets after a type means "array with given type"
      String[] words = new String[5]; // 5 is the # of slots
      words[2] = "on";     // store "on" in the third slot (count from 0)
      words[0] = "meet";   // store "meet" in the first slot
      System.out.println(Arrays.toString(words));
      // need the above notation to convert an array to a string
}
```

Building Lists Atop Arrays

Now we want to use arrays to organize list elements (implementing the ConsecList example from the previous page). In practice, such lists are called **ArrayLists** in Java. To avoid name clashes, we will call ours `ArrList`. We will also have this be a list of Strings, so that we reserve integers for indices rather than list contents (to avoid confusion).

Here's how to set up a basic ArrList class:

```
public class ArrList {
   String[] theArray;  // the underlying array that stores the elements

   // the constructor
   public ArrList(int initSize) {
       this.theArray = new String[initSize];
   }
}
```

If we were to call new ArrList(3), we'd get the following in memory:

loc1021	ArrList(theArray:loc1022)
loc1022	null
loc1023	null
loc1024	null

Implementing addLast

Now, let's write an addLast method for our class:

```
public class ArrList {
   String[] theArray;  // the underlying array that stores the elements

   // the constructor
   public ArrList(int initSize) {
       this.theArray = new String[initSize];
   }

   // store an item in the list
   public void addLast(String newItem) {
       this.theArray[???] = newItem;
   }
}
```

When we add a new item, where should it go? We want to put it in the next unused space. How do we figure out what that space is? Java does not keep track of that for us. We have to do it manually. Therefore, we will add a field called end to the class to track the last USED element.

```
public class ArrList {
   String[] theArray;  // the underlying array that stores the elements
   int end;            // the last USED slot in the array

}
```

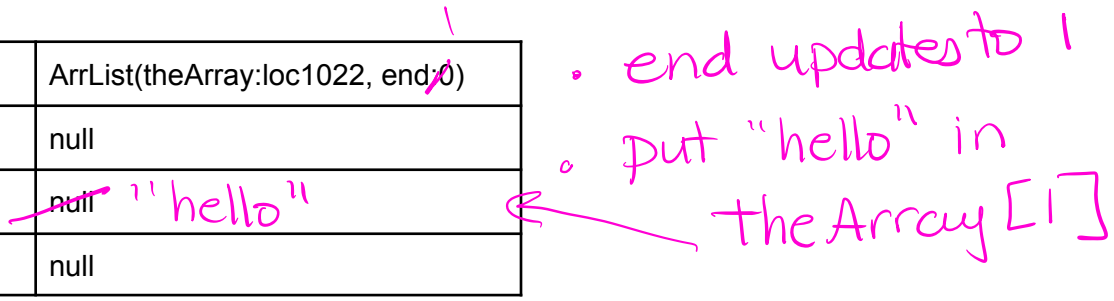
What should the constructor set `this.end` to be? Let's set it to 0 (since the type is int). Finally, in addLast, we'll increment end and put the new element in that new position, thus keeping end referring to the last used space.

```
public class ArrList {
   String[] theArray;  // the underlying array that stores the elements
   int end;            // the last USED slot in the array

   // the constructor
   public ArrList(int initSize) {
       this.end = 0;
       this.theArray = new String[initSize];
   }

   public void addLast(String newItem) {
       this.end = this.end + 1;
       this.theArray[end] = newItem;
   }
}
```

Let's test this out. Assume our ArrList in memory is called AL. Let's see what happens if we run

```
AL.addLast("hello")
```

loc1021	ArrList(theArray:loc1022, end:0)
loc1022	null
loc1023	null
loc1024	null



Whoops – "hello" ended up in the second position, not the first. Why is that? We updated end before putting the item in the array. This made sense because we said end would refer to the last USED position (and indeed this code would be fine if there were already items in the list). But if a list is empty, end refers to 0 even though there is no USED position.

The empty case is special, and we need some extra information to track it.

What we will do is add another field called `eltcount`, that will track whether the list is empty. We will only advance the end index when the list is NOT empty. If the list is empty, we don't advance `end` before inserting the element. That will leave end on the last USED space, as we intended. Instead, we want the following setup, which is what we achieve with the code on the next page

Here now is what we get after using the new addLast code:

loc1021	ArrList(theArray:loc1022, end:0)
loc1022	"hello"
loc1023	null
loc1024	null

```java
public class ArrList {
   String[] theArray;  // the underlying array that stores the elements
   int eltcount;       // how many elements are in the array
   int end;            // the last USED slot in the array
   /*
     We need eltcount to distinguish between a list with 0 and 1 elements.
     On its own, if end refers to last USED elt, a value of end=0 is ambiguous
        (does it mean list has 0 elements or 1?). Thus, we need a separate
        mechanism to track whether the list is empty. eltcount is one approach.
   */

   // the constructor
   public ArrList(int initSize) {
       this.end = 0;
       this.eltcount = 0;
       this.theArray = new String[initSize];
   }

   // this is a standard IList method (even though we haven't included the interface yet)
   public boolean isEmpty() {
       return this.eltcount == 0;
   }

   // this is the version of addLast that we had at the end of class.
   public void addLast(String newItem) {
       if (!(isEmpty())) {
           // if array were empty, end already at 0, the right spot
           this.end = this.end + 1;
       }
       this.eltcount = this.eltcount + 1;
       this.theArray[end] = newItem;
   }
}
```

the code with end and eltcount

Running out of space …

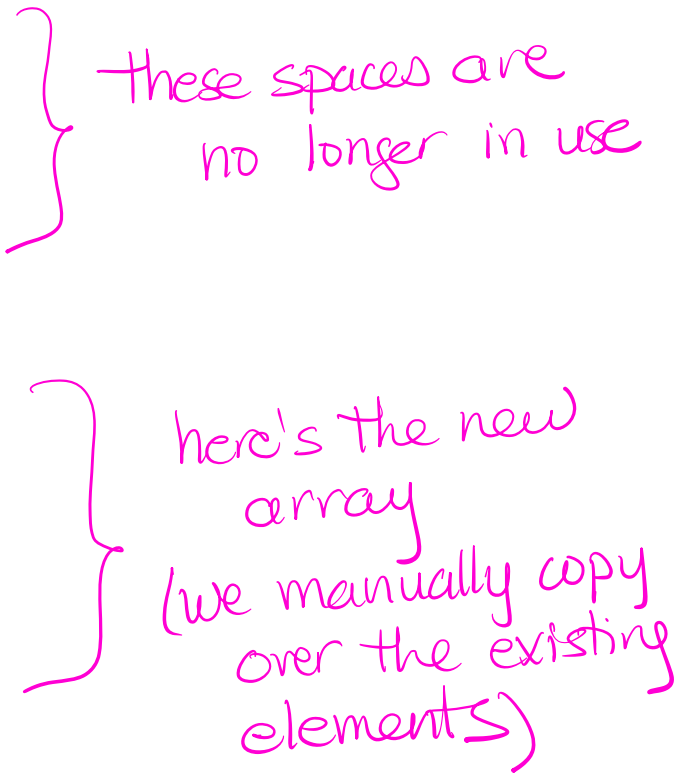
Let's jump ahead and assume we had put three strings into our list:

loc1021	ArrList(theArray:loc1022, end:2)
loc1022	"hello"
loc1023	"there"
loc1024	"brown"

What happens if we try to add another string (e.g., "bear") at the end? Java will raise an error because we would be trying to insert an item beyond the memory locations allocated to the Array. (Remember, another object might already be sitting in loc1025 if we created more objects before adding all the elements to this list).

This suggests that addLast has to check whether there is enough room in the array before storing the new item. If there is NOT enough room, the list elements need to move to a new array with enough space:

loc1021	ArrList(theArray:loc1022, end:2)
loc1022	"hello"
loc1023	"there"
loc1024	"brown"
loc1025	Dillo(5, true)
…	
loc1132	ArrList(theArray:loc1133, end:2)
loc1133	"hello"
loc1134	"there"
loc1135	"brown"
loc1136	"bear"
loc1137	



Specifically, our addLast code in ArrList needs to:

- check whether the current array still has space
- if not, create a new array with an additional slot
- copy over all of the existing array contents to the new array
- insert the new string in the next (newly added) position

The code on the next page implements this. As a good exercise, box off the code that corresponds to each of these tasks so that you see the structure.

```
public class ArrList {
   String[] theArray;  // the underlying array that stores the elements
   int eltcount;       // how many elements are in the array
   int end;            // the last USED slot in the array

   // this private method helps improve readability of later code
   private boolean isFull() {
       // theArray.length returns the CAPACITY of the array, not the # of filled slots
       return this.eltcount == this.theArray.length;
   }

   // this is a standard IList method (even though we haven't included the interface yet)
   public boolean isEmpty() {
       return this.eltcount == 0;
   }

   // A method to make a new array to store list elements. This gets
   // called from a corrected addLast (see below) when the array is out of space
   private void resize(int newSize) {
       // make the new array
       String[] newArray = new String[newSize];
       // copy items from the current theArray to newArray
       for (int index = 0; index < theArray.length; index++) {
           newArray[index] = this.theArray[index];
       }
       // change this.theArray to refer to the new, larger array
       this.theArray = newArray;
   }

   // this is the corrected addLast, which handles additions beyond
   // the array capacity by first resizing the array, then adding the new element
   public void addLast(String newItem) {
       if (this.isFull()) {
           // add capacity to the array
           this.resize(this.theArray.length + 1);
           // now that the array has room, add the item
           this.addLast(newItem);
       } else {
           if (!(this.isEmpty())) {
               this.end = this.end + 1;
           }
           // increase the element count (whether or not list was empty)
           this.eltcount = this.eltcount + 1;
           // store the new element
           this.theArray[end] = newItem;
       }
   }
```

We still have some questions to address next class, such as:

- What is the worst-case running time of addLast?
- What are the comparative running times of operations across our different types of lists
- What if we want to addFirst to an ArrList?