[See the slides for the figures to accompany these lecture notes.]

Last lecture I discussed how an array can be used to represent a list, and how various list operations can be implemented using arrays. Java has an `ArrayList` class that implements the various methods such as we discussed and uses an array as its underlying data structure. You should check out what these methods are for the `ArrayList` class in the Java API:

`https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html`

Whenever you construct an `ArrayList` object, you need to specify the type of the elements that will be stored in it. You can think of this as a parameter that you pass to the constructor. In Java, the syntax for specifying the type uses <> brackets. For example, to declare an ArrayList of objects that are of type `Shape`, use:

```
ArrayList<Shape>  shapes = new ArrayList<Shape>( );
```

If you look at the Java API, you'll see that the class is `ArrayList<E>` where `E` is called a *generic type*. We will see many examples of generic types later.

Just a few points to emphasize before we move on. First, although the `ArrayList` class implements a list using an underlying array, the user of this class does not index the elements of the array using the array syntax `a[ ]`. The user (the client) doesn't even know what is the name of the underlying array, since it is `private`. Instead the user accesses an array list element using a `get` or `set` or other methods.

Second, because the `ArrayList` class uses an array as its underlying data structure, if one uses `add` or `remove` for an element at the *front* of your list, this operation will take time proportional to `size` (the number of elements in the list) since all the other elements needs to shift position in the array by 1. This can be slow. Thus, although arrays allow you to get and set values in very little (and constant) time, they can be slow for adding and removing from near the front of the list because of this shifting property.

Singly Linked lists

We next look at another list data structure - called a linked list - that partly avoids the problem we just discussed that array lists have when adding or removing from the front. (Linked lists are not a panacea. They have their own problems, as we'll see).

With array lists, each element was referenced by a slot in an array. With linked lists, each element is referenced by a node. A linked list node is an object that contains:

- a reference to an element of a list
- a reference to the `next` node in the linked list.

In Java, we can define a linked list node class as follows:

```
class  SNode<T>{
   T            element;
   SNode<T>     next;
}
```


where T is the generic type of the object in the list, e.g. `Shape` or `Student` or some predefined Java class like `Integer`. We use the `SNode` class to define a `SLinkedList` class.

Any non-empty list has a first element and a last element. If the list has just one element, then the first and last elements are the same. A linked list thus has a first node and a last node. A linked list has variables `head` and `tail` which reference the first and last node, respectively. If there is only one node in the list, then `head` and `tail` point to the same node.

Here is a basic skeleton of an `SLinkedList` class.

```
class  SLinkedList<T>{
   SNode<T>       head;
   SNode<T>       tail;
   int            size;

   private class  SNode<T>{
      T            element;
      SNode<T>     next;
   }
}
```

We make the `SNode` class a private inner class[1] since the client of the linked list class will not ever directly manipulate the nodes. Rather the client only accesses the elements that are referenced by the nodes.

A key advantage of linked lists over array lists which I promised above is that linked lists allow you to add an element or remove an element at the front of the list in a constant amount of time. Let's look the basic algorithms for doing so. Again I will use pseudocode, rather than Java. We begin with an algorithm for adding an element to the front of a linked list.

```
addFirst( e ){                      //  add element e to front of list
   newNode = a new node
   newNode.element = e
   if (head == null){                // list is empty
      head = newNode
      tail = head
   }
   else{
       newNode.next = head
       head = newNode
   }
   size++
}
```

The order of the two instructions in the `else` block matters. If we had used the opposite order, then the `head = newNode` instruction would indeed point to the new first node. However, we would

[1]For info on inner classes (and nested classes in general), see
https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html

not remember where the old first node was. The `newNode.next = head` instruction would cause `newNode.next` to reference itself.

Also notice that we have considered the case that initial the list was empty. This special case ("edge case") will arise sometimes. Whenever you write methods, ask yourself what are the edge cases and make sure you test for them. I may omit the edges cases, sometimes intentionally, sometimes unintentionally. Don't hesitate to ask if notice one is missing.

Let's now look at an algorithm for removing the element at the front of the list. The idea is to advance the `head` variable. But there are a few other things to do too:

```
removeFirst(){
//  test for empty list omitted  (throw an exception)
   tmp = head             // remember first element, so we can return it
   head = head.next       // advance
   tmp.next = null        // not necessary but conceptually cleaner
   size = size - 1
   if (size == 0)
       tail = null        // edge case:  one element in list
   return tmp.element
}
```

Notice how we have used `tmp` here. If we had just started with (`head = head.next`), then the old first node in the list would still be pointing to the new first node in the list, even though the old first node isn't part of the list. (This might not be a problem. But it isn't clean, and sometimes these sorts of things can lead to other problems where you didn't expect them.) Also, in the code here, the method returns the element. Note how this is achieved by the tmp variable. In the slides, the method did not return the removed element.

Let's now discuss methods for adding or removing an element at the back of the list. This requires manipulating the `tail` reference. Adding a node at the tail can be done in a small number of steps.

```
addLast( e ){
   newNode = a new node
   newNode.element = e
   tail.next = newNode
   tail = tail.next
   size = size + 1
}
```

Removing the last node from a list is more complicated, however. The reason is that you need to modify the `next` reference of the node that comes *before* the tail node which you want to remove. But you have no way to directly access the node that comes before `tail`, and so you have to find this node by searching from the front of the list.

The algorithm begins by checking if the list has just one element. If it does, then the last node is the first node and this element is removed. (I do not return the element below. That code could be added.) Otherwise, it scans the list for the element that comes before the last element.


```
removeLast(){
   if (head == tail)
      head = null
      tail = null
   }
   else{
      tmp = head
      while (tmp.next != tail){
         tmp = tmp.next
      }
      tmp.next = null
      tail = tmp
   }
   size = size-1    // optional variable
}
```

This method requires about `size` steps. This is much more expensive than what we had with an array implementation, where we had a constant cost in removing the last element from a list.

Time Complexity

The table below compares the time complexity for adding/removing an element from the head/tail of an array or linked list that has size N.

	array list	singly linked list
addFirst(e)	O(N)	O(1)
removeFirst()	O(N)	O(1)
addLast(e)	O(1)	O(1)
removeLast()	O(1)	O(N)

The main problem with the singly linked list is that it is slow to remove the last element. Note that singly linked lists do just as well as array lists, except for removing an element from the end of the list. We will see one way to get around this next lecture when we discuss doubly linked lists. However, we will also see that doubly linked lists don't solve all of our problems.

How many objects are there in a singly linked list vs. array list?

As I discuss in the slides, suppose we have a linked list with `size` = 4. How many objects do we have in total? We have the four `SNode` objects. We have the four elements (objects of type `Shape, say`) that are referenced by these nodes. We also have the `SLinkedList` object which has the head and tail references. So this is 9 objects in total.

For an array list with four elements, we would have 6 objects: the four elements in the list, the arraylist object, and the underlying array object. (Yes, an array is considered to be an object in Java.)