Java generics

In our discussion of linked lists, we concentrated on how to add or remove a node from the front or back of a list. The fact that each node held a primitive type versus a reference type, or whether the reference type was a `Shape` object or `Dog` object (or whatever) was not important to that discussion. Rather, we were mainly concerned with how to manipulate the nodes in the list.

When you implement a linked list in some programming language such as Java, you do have to consider the types of elements at each node. However, you don't want to have to reimplement your linked list class for each new type (`Shape, Dog, etc`. Java allows you to write classes with a *generic type*, which addresses this issue.

Rather than declaring classes for a particular type `E` – for example, `E` being an `int` or `Shape` as we saw last lecture – we would like to define our linked list in a more general way.

```
class  SNode{
   E            element;
   SNode        next;
}

class  SLinkedList{
   SNode        head;
   SNode        tail;
   int          size;
}
```

Java allows us to do so. We can declare these classes to have a *generic* type `E`.

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

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

This way, we can write all the methods of these classes such that they can be used for any type of reference type. For example, we could have a linked list of `Shape` objects, or a linked list of `Student` objects, etc.

```
SLinkedList<Shape>    shapelist = new SLinkedList<Shape>();
SLinkedList<Student>  studentlist = new SLinkedList<Student>();
```

You should think of the class name as a parameter `E` that is passed to the constructor `SLinkedList<E>()`.

See the examples that are given in the online code.

Doubly linked lists

The "S" in the SLinkedList class did not stand for Shape, but rather it stood for "singly", namely there was only one link from a node to another node. We next look at "doubly linked" lists. Each node of a doubly linked list has two links rather than one, namely each node of a doubly linked list has a reference to the previous node in the list and to the next node in the list. These reference variables are typically called prev and next.

```
class  DNode<E>{
   E            element;
   DNode<E>        next;
   DNode<E>        prev;
}

class  DLinkedList<E>{
   DNode<E>        head;
   DNode<E>        tail;
   int             size;
}
```

One advantage of doubly linked lists is that they allow us to access elements near the back of the list without having to step all the way from the front of the list. Recall the removeLast() method from earlier. To remove the last node of a singly linked list, we needed to follow the next references from the head all the way to the node whose next node was the last/tail node. This required about size steps which is inefficient. If you have a doubly linked list, then you can remove the last element in the list in a constant number of steps, e.g.

```
tail      =  tail.prev;
tail.next = null;
size      = size-1;
```

More generally, with a doubly linked list, we can remove an arbitrary element in the list in constant time.

```
remove( node ){
   node.prev.next = node.next;
   node.next.prev = node.prev;
   size           = size-1;
}
```

(Here I am not concerning myself with the next and prev references in the removed node, i.e. if we don't set them to null then they will continue to point to nodes in the list.)

Another advantage of doubly linked lists is that, if we want to access an element, then we don't necessarily need to start at the beginning of the list. If the node containing the element is near the back of the list, then we can access the node by following the links from the back of the list. For example, suppose we wished to remove the i'th node.

```
getNode(i){
   if (i < size/2){
      tmp = head
      index = 0
      while (index < i){
         tmp = tmp.next
         index++
      }
   else{
      tmp = tail
      index = size - 1
      while (index > i){
         tmp = tmp.prev
         index--
      }
   return tmp
}
```

This algorithm takes at most `size/2` steps, instead of `size` steps.

Dummy nodes

The above method for removing a node assumes that `node.prev` and `node.next` are well defined. However, this sometimes will not be the case, for example, if the node is the first or the last in the list, respectively. Thus, to write correct code for removing an element, you need to take these special end cases into account. (Recall the `removeLast()` method on page 1.) It is easy to forget to do so.

To avoid this problem, it is common to define linked lists by by using a "dummy" head node and a "dummy" tail node, instead of head and tail reference variables. The dummy nodes[3] are nodes just like the other nodes in the list, except that they do not contain an element (e.g. a shape object). Rather the `element` field points to `null`. These nodes do not contribute to the `size` count.

```
class  DLinkedList<E>{
   DNode<E>        dummyHead;
   DNode<E>        dummyTail;
   int             size;
   }
```

[3]Dummy nodes are somewhat controversial. Some programmers regard them as an ugly "hack". Others find them elegant and useful. I personally don't have a strong opinion either way. I am teaching you about dummy nodes so that you are aware of them.

```
      DLinkedList<E>(){
         dummyHead = new DNode<E>();
         dummyTail = new DNode<E>();
         size      = 0;
      }

      //  ... lots of list methods
   }
```

Comparison of arrays and linked lists

Let's compare the time required to execute several methods, using arrays versus singly versus doubly linked lists. Let `N = size`, namely the number of elements in a list.

```
                          array        singly           doubly
                                     linked list       linked list
                         ------      ----------        ----------


   addFirst                N             1                  1
   removeFirst             N             1                  1
   addLast                 1             1                  1
   removeLast              1             N                  1
   getNode(i)              1             i            min( i, N/2 - i)
```

List as an "abstract data type" (ADT) or "interface"

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. We have seen several examples:

```
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
  :
```

We can think of these methods as an *interface*[4] to a list object, in the sense that the methods allow us to perform operations on the list. Sometimes we don't care how the list is implemented. We just want to make sure that it performs the operations as are specified in the interface.

Other times, we do care how the list is implemented, since this may affect the speed and the memory space that is used. We saw, for example, that singly linked lists use less memory than doubly linked lists but singly linked lists are slower for some operations. Our choice of using a singly linked versus doubly linked list might depend on whether this tradeoff is a consideration for our application.

Java `LinkedList`

Java has a `LinkedList` class which is implemented as a doubly linked list.

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

You should check out the Java API for the methods that this class provides, beyond those listed above.

Suppose we add n students to the front (or back) of the list. For example, the `add` method appends the element to back of the list. Adding n students to an empty list 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.

Suppose we have a list of n elements which is implemented using a doubly linked list, and we want 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.

[4]The word "interface" has a specific technical meaning in Java which we will discuss later in the course.

```
  for (j = 1; j < n;  j++)
     print( studentList.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

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

which grows like n^2. Eeeks! 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. Note:

- Even if we were to take advantage of the doubly linked list and search for the i node by starting from the front or back, whichever is closer, the time still grows with n^2.
- What alternatives to we have? In Java, we can use something called an `Iterator`. We will discuss these later in the course.

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
                   //   index >= i.
  remove(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 create a bigger underlying array[5] 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.

Copying references from a small array to a big array takes time. Let's think about how much. 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 and then fill it, we need to copy the elements from the smaller array to the bigger array and then fill the remaining elements of the bigger array. When $k = 1$, we copy 1 element from an array of size 1 to an array of size 2 and then add the second element in the 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 and then add the remaining 2 elements. In general, the number of copies and adds together is:

$$1 + 2 + 4 + 8 + \cdots + 2^k.$$

But the sum is of the form

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

where $x = 2$ and so

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

Since we have expanded k times, we have an array of size $n = 2^k$ and 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 accounts for the $2n - 1$ reference settings in total. Thus, doubling the size of a filled array and copying the references does require some work, but this extra amount of work grows as $2n$. This order of growth is not a problem; we need to set n references anyhow to add n elements.

Comparison of ArrayList and LinkedList: worst case

As a summary, let's collect the *worst case* performance of the two implementations of lists, namely the number of steps we need to take. Here I am ignoring constant factors, including the division by 2 for the doubly linked list.

[5]bigger by 50% in Java, but our arguments here assume we increase by 100% i.e. double the size of the array

	`LinkedList`	`ArrayList`
`add(element)`	1	1
`add(i,element)`	n	n
`set(i,element)`	n	1
`remove(i)`	n	n
`get(i)`	n	1
`clear()`	1	1
`isEmpty()`	1	1
`size()`	1	1

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 in the next few lectures, namely the stack and the queue.

In Java, one does not use the term ADT. Rather, there is 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.