[ASIDE: You should follow along with the figures in the slides.]

Doubly linked lists

The "S" in the SLinkedList class from last lecture stood for "singly", namely there was only one link from a node to another node. Today we look at "doubly linked" lists. Each node of a doubly linked list has two links rather than one, namely references to the previous node in the list and to the next node in the list. These reference variables are typically called prev and next. As with the singly linked list class, the node class is usually declared to be a private inner class. Here we define it within a DLinkedList class.

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

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

The key advantage of doubly linked lists over a singly linked list is that the former allows us to quickly access elements near the back of the list. For example, to remove the last element of a doubly linked list, one simply does the following:

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

Dummy nodes

When writing methods (or algorithms in general), one has to consider the "edge cases". For doubly linked lists, the edge cases are the first and last elements. These cases require special attention since head.prev and tail.next will be null which can cause errors in your methods if you are not careful.

To avoid such errors, it is common to define linked lists by using a "dummy" head node and a "dummy" tail node, instead of head and tail reference variables.[1] The dummy nodes are objects of type DNode just like the other nodes in the list. However, these nodes have a null element. Dummy nodes do not contribute to the size count, since the purpose of size is to indicate the number of elements in the list. See figures in slides.

[1]Dummy nodes can be defined for singly linked lists too.

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

   // constructor

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

   //  ... List methods and more
}
```

Let's now look at some DLinkedList methods. We'll start with a basic getter which gets the i-th element in the list:

```
   get( i ){
     node = getNode(i)
     return  node.element
   }
```

This method uses a helper method that I'll discuss below. Its worth having a helper method because we can re-use it for several other methods. For example, to remove the i-th node:

```
  remove( i ){
     node = getNode(i)
     node.prev.next = node.next
     node.next.prev = node.prev
     size--
  }
```

This code modifies the `next` reference of the node that comes before the i-th node, that is `node.prev`, and it modifies the `prev` reference of the node that comes after the i-th node, that is `node.next`. Because we are using dummy nodes, this mechanism works even if i = 0 or i = size-1. Without dummy nodes, `node.prev` is `null` when i = 0, and `node.next` is null when `i = size - 1`, so the above code would have an error if we didn't use dummy nodes.
[ASIDE: note that I am not bothering to set the `next` and `prev` references in the removed node to `null`.]

Here is an implementation of the `getNode(i)` method. This method would be *private* to the `DLinkedList` class.

```
getNode(i){
   node = dummyHead.next
   for (k = 0;  k < i; k++)
      node = node.next
   return node
 }
```

One can be more efficient than that, however. When index i is greater than size/2, then it would be faster to start at the tail and work backwards to the front, so one would need to traverse size/2 nodes in the worst case, rather than size nodes as above.

```
getNode(i){
   if (i < size/2){
       node = dummyHead.next
       for (k = 0;  k < i; k++)
           node = node.next
   }
   else {
       node = dummyTail.prev
       for (k = size-1;  k > i; k--)
          node = node.prev
       }
   return node
}
```

The `remove(i)` method still takes `size/2` operations in the worst case. Although this worst case is a factor of 2 smaller for doubly linked list than for singly linked lists, it still grows linearly with the size of the list. Thus we say that the `remove(i)` method for doubly linked lists still is $O(N)$ when N is the size of the size. This is the same time complexity for this method as we saw for array lists and for singly linked lists. Saving a factor of 2 by using the trick of starting from the tail of the list half the time is useful and does indeed speed things up, but only by a proportionality factor. It doesn't change the fact that in the worst case the time is takes grows linearly with the size of the list.

Java `LinkedList`

Java has a `LinkedList` class which is implemented as a doubly linked list. Like the `ArrayList` class, it uses a generic type `T`.

```
LinkedList<T>  list  =  new  LinkedList<T>();
```

The `LinkedList` class has more methods than the `ArrayList` class. In particular, the `LinkedList` class has `addFirst()` and `removeFirst()` methods. Recall removing elements from near the front of a linked list was expensive for an array list. So if you are doing this a lot in your code, then you probably don't want to be using an array list for your list. So it makes sense that that the `ArrayList` class wouldn't have such methods. But adding and removing the first elements from a list is cheap for linked lists, so that's why it makes sense to have these methods in the Java `LinkedList` class. (Of course, you could just use `remove(0)` or `add(0,e)`, but a specialized implementation `addFirst()` and `removeFirst()` might be a bit faster and the code would be easier to read – both of which are worth it if the commands are used often. In addition, the `LinkedList` class has an `addLast()` and `removeLast()` method, whereas the `ArrayList` class does not have these methods.

Suppose we add N students to the front (or back) of a linked list. Adding N students to an empty linked list takes time proportional to N since adding *each* element to the front (or back) of a linked list takes a constant amount of time, i.e. independent of the size of the list

How not to iterate through a linked list

What if we wanted to print out the elements in a list. At first glance, the following pseudocode would seem to work fine.

```
for (k = 0; k < list.size();   k ++)            //  size == N
     System.out.println( list.get(  k  ));
```

For simplicity, suppose that `get` were implemented by starting at the head and then stepping through the list, following the next reference until we get to the i-the element. (See Exercises 3 for case that the `get` method uses what we did on the previous page, namely worst case is $N/2$.) Then, with a linked list as the underlying implementation, the above for loop would require

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

steps. This is $O(N^2)$ which gets large when N is large. It is obviously inefficient since we realy only need to walk through the list and visit each element once. The problem here is that each time through the loop the `get(k)` call starts again at the beginning of the linked list and walks to the k-th element.

What alternatives to we have to using repeated `get`'s ? In Java, we can use something called an *enhanced for loop*. The syntax is:

```
  for  (E    e :   list){
         //  do something with element e
  }
```

where `list` is our variable of type `LinkedList<E>` and `e` is a variable that will reference the elements of type `E` that are in the list, from position, 0, 1, 2, ..., size-1.

Note that in Assignment 1, you will use the `LinkedList` class. At this stage of your Java abilities and knowledge, it will be easier for you to use the regular `for` loop rather than the enhanced `for loop`. So, just plug your nose when you write out the regular `for` loops such as the in the print statements above.

Space Complexity ?

We've considered how long it takes for certain list operations, using different data structures (array list, singly linked and doubly linked lists). What about the space required? Array lists use the least amount of total space since they require just one reference for each element of the list. (These 'slots' must be adjacent in memory, so that the address of the k-th slot can be computed quickly.)

Singly linked lists take twice as much space as array lists, since for each element in a singly linked lists, we require a reference to the element and a reference to the next node in the list. Doubly linked lists require three times as much space as array lists, because they require a reference to each element and also references to the previous and next node.

All three data structures require space proportional to the number of elements N in the list, however. So we would say that all require $O(N)$ space.

Miscellaneous Java stuff

Overloading, and method signatures

When you look at the Java API for the `ArrayList` or `LinkedList` class or many other Java classes you will see that there are some methods that have multiple versions. For example, in these list classes there are several versions of the `remove` method, including `remove(E element)` and `remove(int index)`. Both of these return a reference to the removed element. The first `remove` method removes the first instance of the given element in the list. The second `remove` method removes the i-th element, whatever it happens to be.

A method that has multiple versions is said *overloaded*. When a method is overloaded, there must be some way for the Java compiler to know which version of the method the programmer wants to use. Overloaded methods are distinguished by the type(s) and number of the arguments, what is called the method *signature*. The Java compiler can usually know which method the programmer wishes to use just be examining the type of the parameter passed to the method. (There are some subtleties here, which we will return to at the end of the course, when we discuss "polymorphism".) For now, you should just note that there can exist multiple versions and they are distinguished by the types of their arguments.