[ASIDE: The lecture notes contain mostly text. Although sometimes I will insert figures, usually I will assume that you will follow along with the figures that are 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 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.

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

In addition to a node class, one defines a linked list class. (Note that in the practice code I give you in the Exercises, the above node class is an ”inner class” defined within the DLinkedList class.)

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

The key advantage of doubly linked lists over a singly linked list is that a doubly linked list allows 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 singly linked list’s removeLast() method from last lecture. To remove the last node, we needed to follow the next references from the head to the tail. This required about size steps which is inefficient. If we have a doubly linked list, then we 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
```

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\footnote{The dummy nodes are objects of type \texttt{DNode} just like the other nodes in the list. However, these nodes have a \texttt{null} element. Dummy nodes do not contribute to the \texttt{size} count, since the purpose of \texttt{size} is to indicate the number of elements in the list. See figures in slides.}. The dummy nodes are objects of type \texttt{DNode} just like the other nodes in the list. However, these nodes have a \texttt{null} element. Dummy nodes do not contribute to the \texttt{size} count, since the purpose of \texttt{size} is to indicate the number of elements in the list. See figures in slides.

class \texttt{DLinkedList\langle E\rangle}\{ 
    \texttt{DNode\langle E\rangle} \hspace{1em} \texttt{dummyHead};
    \texttt{DNode\langle E\rangle} \hspace{1em} \texttt{dummyTail};
    \texttt{int} \hspace{1em} \texttt{size};

    \:\:

    // constructor

    \texttt{DLinkedList\langle E\rangle\{\}}
    \hspace{1em} \texttt{dummyHead} = \texttt{new DNode\langle E\rangle\{\}};
    \hspace{1em} \texttt{dummyTail} = \texttt{new DNode\langle E\rangle\{\}};
    \hspace{1em} \texttt{dummyHead.next} = \texttt{dummyTail};
    \hspace{1em} \texttt{dummyTail.prev} = \texttt{dummyHead};
    \hspace{1em} \texttt{size} = 0;

    \}\n
    // ... List methods and more
\}

Let’s return to the problem of removing the i-th node. Here is one way to do it:

\texttt{remove( i )\{}
    \hspace{1em} \texttt{node} = \texttt{getNode(i)}
    \hspace{1em} \texttt{node.prev.next} = \texttt{node.next}
    \hspace{1em} \texttt{node.next.prev} = \texttt{node.prev}
    \hspace{1em} \texttt{size}--
\}\n
We have used a \texttt{getNode(i)} method which returns a reference to the ith node that we need to remove. I’ll go over an implementation of this method below, but its not the main point here. Rather, the main point is once we have a reference to the i-th node, it is easy to remove it. We just modify the \texttt{next} reference of the node that comes before i, that is \texttt{node.prev}, and we modify the \texttt{prev} reference of the node that comes after i, that is \texttt{node.next}. The key idea of dummy nodes is that this mechanism works even if i = 0 or i = size-1. Without dummy nodes, \texttt{node.prev} is \texttt{null} when i = 0, and \texttt{node.next} is \texttt{null} when i = size - 1, so the above code would crash on those cases if we didn’t use dummy nodes.
[ASIDE: note that I am ignoring the next and prev references in the removed node, i.e. I am not bothering to set them to null. These references will continue to point to nodes in the list, until the removed node is "garbage collected". (Nothing points to this removed node, so eventually the Java Virtual Machine will figure out that it is garbage.)]

Recall that the method above assumed a getNode(i) method. Here is an implementation of this method. This method would be private to the DLinkedList class.

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

One can be more efficient than that, however. If 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.

```java
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
}
```
Comparison of arrays and linked lists

We see that, for the doubly linked list, the \texttt{remove(i)} method takes \texttt{size/2} operations in the worst case. Although this worst case is a factor of 2 smaller for doubly linked list, it still grows linearly with the size of the list. Thus we say that the \texttt{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 \texttt{LinkedList} class which is implemented as a doubly linked list. Like the \texttt{ArrayList} class, it uses a generic type \( T \). 

\texttt{LinkedList}\langle T \rangle \texttt{ list } = \texttt{ new LinkedList}\langle T \rangle();

The \texttt{LinkedList} class has more methods than the \texttt{ArrayList} class. In particular, the \texttt{LinkedList} class has \texttt{addFirst()} and \texttt{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 \texttt{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 \texttt{LinkedList} class. (Of course, you could just use \texttt{remove(0)} or \texttt{add(0,e)}, but a specialized implementation \texttt{addFirst()} and \texttt{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 \texttt{LinkedList} class has an \texttt{addLast()} and \texttt{removeLast()} method, whereas the \texttt{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.

Next, what if we were to define a \texttt{display} method that prints each of the elements. At first glance, the following pseudocode would seem to work fine.

\begin{verbatim}
for (k = 0; k < list.size();  k ++) // size == N
    System.out.println( list.get( k ));
\end{verbatim}

For simplicity, suppose that \texttt{get} is implemented by starting at the head and then stepping through the list, following the next reference\footnote{See Exercises 3 for case that the \texttt{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 code would require

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

which is \( O(N^2) \). Eeeks! What went wrong? The answer is that the \texttt{get(k)} starts again at the beginning of the (linked) list and walks to the k-th element. For large lists, this is very inefficient.
What alternatives to we have to using repeated get’s ? In Java, we can use something called an enhanced for loop. The syntax is:

```java
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.

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.