Last class we looked at singly linked lists. We finished the lecture by examining a method for removing the first node of the list.

The next method to consider is `removeTail`. This turns out to be a bit awkward. Suppose there is more than one element in the list. If we were to remove the node that is referenced by `tail`, then after doing so `tail` should point to the previous node in the list, that is, the node that had pointed to the `tail` node just before we removed the tail node. The problem is that we have no fast way of directly indexing this previous element. The only way we could do it would be something like:

```
SNode removeTail(){
//    Insert code here that handles case that list is empty or contains
//    just one element, i.e.  (head == null) || (head.next == null)
    SNode cur = head;
    while (cur.getNext() != tail)
       cur = cur.getNext();
    tail = cur;
    SNode tmp = cur.getNext();
    tail.setNext(null);
    return tmp;
 }
```

But this means running along the whole length of the list, which could take a while. Note that a similar problem arises if we were to define a method `remove(Node node)` which removes an arbitrary node from the list.
I have put some code for singly linked lists online, next to today's lecture notes: `SinglyLinkedList.zip`.

Doubly linked lists

To avoid the problem just discussed, one typically adds a second reference variable to each node. This second reference variable points back to the previous node in the list. A list constructed from such nodes is called a *doubly linked list*, which contrasts with what we saw above which was called a *singly linked list*. Here is how we could define the `DNode`. I have not written various getter and setter methods that were used in `SNode`, but they should be there too.

```
--------------------------
|    DNode               |
|                        :
|  SomeClass  element    |
|     :                  |
|  DNode  prev           |
|  DNode  next           |
|                        |
|  DNode()               |
|      :                 |
--------------------------
```

We next define a doubly linked list class. You might think this is done in exactly the same way as with the singly linked list class, namely you have a reference variable to the first and to the last node in the list, and the `prev` variable of the first node would be `null` and the `next` variable of the last node would be `null`. This is roughly the case, except that the first node is a "dummy node" and the last node is a "dummy node." That is, if there are no objects, then we have two nodes, the dummy head node and the dummy tail node. The `prev` field for the `header` node is `null` and the `next` field for the `tailer` field is `null`.

```
--------------------------------------
|         DLinkedList                |
|                                    |
|  DNode header   //  dummy          |
|  DNode tailer   //  dummy          |
|                                    |
|  DlinkedList()                     |
|  void  remove(DNode)               |
|  void  insertAfter(DNode,DNode )   |
|            :                       |
--------------------------------------
```

The dummy nodes are mainly for coding convenience. To remove a general node, we would like to change the `next` field of the `prev` node (and the `prev` field of the `next` node). However, in order for this to make sense, these "nodes" just referred to need to exist! That is, the references cannot be `null`. Having these dummy nodes ensures that the `next` and `prev` fields of any non-dummy nodes are not `null`. [We could avoid having non-dummy nodes, but then we would need to test that the appropriate `next` or `prev` field was not null, and this could lead to headaches.]

In class, I discussed the following method and drew pictures showing how the references variables are set. These are not repeated here, as the textbook does an adequate job (see Sec. 3.3).

```
public DNode remove(DNode node){

//  maybe check (node != null) && (node != header) && (node != trailer)

   DNode  u, w;
   u = node.getPrev();   // always exists
   w = node.getNext();
   u.setNext() = w;
   w.setPrev() = u;
   node.setNext(null);
   node.setPrev(null);
   return node;
}
```