

I began this lecture by elaborating on the singly linked list which I had defined in lecture 4. There I had concentrated on the nodes of the list and in particular how to add or remove a node from the front or back of a list. The fact that each node referenced a **Shape** object (as opposed to a **Dog** object or whatever) was not important to that discussion, since I was mainly concerned with manipulating the nodes. In this lecture, I now turn to the issue of how to represent the type of object that is referenced by each node.

To understand how singly linked lists work, you should review the slides and *you should download the code that I provided* (next to last lecture’s notes) *and you should read and run that code*. I also recommend that you try to implement a few simple singly linked list methods yourself, for example, how to add or remove the i -th element of a linked list where i is a parameter. Adding or removing an i th element is a standard operation for lists, as we will see next lecture.

Java generics

When you look at the code, you will find that it does not define a **Shape** list. The reason I did not do so is that the specifics of the **Shape** class are independent of linked lists. Instead I provided a generic **SLinkedList** class, which does not require you to commit to a specific “cargo” class such as **Shape**. Java allows you to do this by providing a *generic type*. For a discussion of generic types in Java, see

<http://download.oracle.com/javase/tutorial/java/generics/generics.html> in particular, read the first three sublinks *only* (Introduction, Generic Types, Generic Methods and Constructors). *We will be using generic types for many of the data structures in this course.*

Doubly linked lists

We next looked at doubly linked lists. These are very similar to singly linked lists, the main difference being that 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**.

One key 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 last lecture. 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 node. This took N 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
```

Dummy nodes

It is common to define doubly linked lists by including a dummy head node and a dummy tail node in the list, instead of just head and tail reference variables. The dummy nodes 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`. (See the slides and code.)

Dummy nodes make coding easier. When we add or remove a node from a doubly linked list, in general we need to modify the `next` and `prev` fields of the node we are adding or removing, and also we need to modify the `next` and `prev` fields of the nodes that are adjacent in the list, name the nodes that come before and after. If we use a dummy head or tail node, we can be sure that two adjacent nodes exist in all cases, in particular, if we are adding or removing to/from the front or back of the list, or if we are adding to an empty list. By using a dummy head and tail node, we don't have to write special code for the case that we are at the front or back of the list.

Today's lecture notes were brief and together with the slides they give only the basics. *To understand how linked lists work, you need to implement them yourselves.* I have given you a skeleton of a doubly linked list class. You should read and run that code. I also recommend that you try to implement a few linked list methods yourself, for example, how to add or remove the i -th element of a linked list where i is a parameter. For more linked list methods, see the Java API for the `LinkedList` class. This class is implemented using a doubly linked list.