

Today's lecture notes are briefer than usual: the slides and the posted Java code should be sufficient. I am expecting you to run and play with that code to make sure you understand how it works.

In the next part of the course, we consider data structures than maintain an ordering on a set of objects. By “ordering”, I just mean that we can talk about a first object, a second object, etc. Such a set of ordered objects is often called a *list*.

## Arrays as lists

A natural data structure for representing a list is an array. Arrays allow us to access any of the objects in the list by providing the position (index) of the object. There are problems with arrays, however. If you want to insert a new object into the front of the list then, if you use an array, you have to make room for that object in the array. For example, if you want to insert a new object at the front of the list (into array slot 0), then you need to move all the elements ahead one position ( $i \rightarrow i + 1$ ).

```
for (i = N-1 downto 0)
  a[i+1] <-- a[i]
a[0] <-- new element
```

Similarly, if you want to remove the first element, then you need to move all elements back one position ( $i \rightarrow i - 1$ ).

```
for (i = 1 to N-1)
  a[i-1] <-- a[i]
```

In either case,  $N$  operations are done. This is obviously inefficient if we are doing a lot of insertions and deletions from the list, and if many of these insertions and deletions are at the front of the list.

## Singly Linked lists

In this lecture I discussed an alternative data structure for representing a list. Define a list *node* which contains:

- a reference to an object<sup>1</sup>
- a reference to the *next* node in the list.

In Java, we would have a node class defined as follows:

```
class SNode{
  Type      element
  SNode     next
}
```

---

<sup>1</sup>Alternatively, we might have a list of numbers in which case each node could contain the number (rather than a reference to it).

where `Type` is the type of the objects that we are trying to organize.

To define a *linked list*, we define another class, say:

```
class SLinkedList{
    SNode      head
    SNode      tail
}
```

The field `head` points to the first node in the list and the field `tail` points to the last node in the list. If there is only one node in the list, then `head` and `tail` would point to the same node.

In the lecture I discussed four methods for manipulating a linked list:

- `addFirst(SNode)`
- `addLast(SNode)`
- `removeFirst()`
- `removeLast()`

I sketched out why the first three of these only require a small number of steps, and also showed that the `removeLast()` method requires about  $N$  steps, if the list has  $N$  elements.

*Please see the lecture slides and Java code for the details. The code contains other methods that you should familiarize yourselves with as well.*