CS61B Summer 2006
Instructor: Erin Korber
Lecture 10, 12 July

Linked Lists

1 Why lists?

We can store a list of items as an array, but there are disadvantages to this representation. First, arrays have a fixed length that can't be changed. If we want to add items to a list, but the array is full, we have to allocate a whole new array.

Second, if we want to insert an item at the beginning or middle of an array, we have to slide half the items over one place to make room. This takes time proportional to the length of the array.

We can avoid these problems by choosing a Scheme-like representation of lists. A linked list is made up of "nodes". Each node has two components: an item, and a reference to the next node in the list. These components are analogous to Scheme's x"car" and "cdr". However, our node is an explicitly defined object.

2 List Nodes

List node is an example of a *recursive data type* - we use list nodes as part of the definition of what a list node is.

```
public class ListNode {
   int item;
   ListNode next;

   //methods here
}
```

Let's make some ListNodes.

```
  ListNode l1 = new ListNode();
  ListNode l2 = new ListNode();
  ListNode l3 = new ListNode();
  l1.item = 7;
  l2.item = 0;
```

```
l3.item = 6;
```

Now let's link them together.

```
l1.next = l2;
l2.next = l3;
```

What about the last node? Since the last node has no "next" object, we need its last reference to refer to nothing - a.k.a. `null`.

```
l3.next = null;
```

To simplify programming, we can add some constructors to the ListNode class.

```
 public ListNode(int item, ListNode next) {
    this.item = item;
    this.next = next;
 }

 public ListNode(int item) {
    this(item, null);
 }
```

These constructors allow us to emulate Scheme's "cons" operation, like so:

```
ListNode l1 = new ListNode(7, new ListNode(0, new ListNode(6)));
```

Inserting an item into the middle of a linked list is a constant-time operation, and the list can keep growing until memory runs out. The `insertAfter` ListNode method inserts a new item into the list immediately after the current one.

```
public void insertAfter(int item) {
  next = new ListNode(item, next);
}
```

3 Why not linked lists?

Linked lists do have some disadvantages from arrays. Finding the nth item of a linked list takes time proportional to n, even though it is a constant-time operation on array-based lists. The `nth` method recursively finds the nth node in a list that starts with the current node `this`.

```
public ListNode nth(int n) {
  if (n == 1) {
    return this;
  } else if ((next == null) || (n < 1)) {
    System.out.println(''There is no ''+ n +''th node!'');
    return null;
  } else {
    return next.nth(n - 1);
  }
}
```

4 Lists of Objects

For greater generality, we can change ListNodes so that each node contains not an int, but a reference to any object.

```
public class SListNode {
  public Object item;
  public SListNode next;
}
```

The “S” in “SListNode” stands for singly-linked.

5 A List Class

There are two problems with SListNodes.

1. Suppose x and y are pointers to the same shopping list. Suppose we insert a new item at the beginning of the list thus: `x = new SListNode("eggs", x);`

   y doesn’t point to the new item; y still points to the second item in x’s list. If y goes shopping for x, he’ll forget to buy eggs.

2. How do you represent an empty list? The obvious way is ”x = null”. However, Java won’t let you call any method on a null object. If you write ”x.nth(1)” when x is null, you’ll get a run-time error, even though x is declared to be an SListNode, so you would expect to get the “The list has no 1th node!” message and a null reference returned.

The solution is a separate SList class, whose job is to maintain the head (first node) of the list. We will put many of the methods that operate on lists in the SList class, rather than the SListNode class.

```
public class SList {
  private SListNode head;

  public SList() {
    head = null;
  }

  public SList(SListNode head) {
    this.head = head;
  }

  public void insertFront(Object item) {
    head = new SListNode(item, head);
  }
}
```

Now, when an item is inserted at the front of a SList, every reference to that SList can see the change.

If we wanted, we could also have the SList class keep a record of the SList's size (number of SListNodes), so that the size could be determined more quickly than if the SListNodes had to be counted.

6 Doubly linked lists

As we have seen, inserting an item at the front of a linked list is easy. Deleting from the front of a list is also easy:

```
  public void deleteFront();
    if (head != null) {
      head = head.next;
      size--;
    }
  }
```

However, inserting or deleting an item at the end of a list entails a search through the entire list, which might take a long time. One possible solution to this problem is a doubly-linked list, in which each node has references to both the previous and next node, and the list has references to its head and tail nodes.

```
public class DListNode {
    private Object item;
    private DListNode next;
    private DListNode prev;

   //methods here

}

public class Dlist {
   private DListNode head;
   private DListNode tail;

  //methods here

}
```

DLists make it possible to insert and delete items at both ends of the list without having to go through every item in the list. For example, the following code removes the tail node if there are at least two items in the DList.

```
  tail.prev.next = null;
  tail = tail.prev;
```

You'll need a special case for a DList with no items. You'll also need a special case for a DList with one item, because tail.prev.next does not exist. (Instead, head needs to be changed.) In lab today, you'll be implementing this and several other methods in the DList class.

7 Lists in the Java library

The Java library has several list classes (classes that implement the *List* interface) - the two you will see most often are `ArrayList` and `LinkedList`. You can read about these implementations in the API if you wish.

Since one of the main purposes of this class is learning to implement data structures, we are writing our own list classes rather than just using those from the library - this will be true for several of the data structures we discuss. Also, you will often find that our version of a given data structure differs from the library version in some ways - often, what is given in the library will not be exactly suited to a paricular purpose, and writing your own version will provide better usability and/or performance.

Finally, although we are using Java to write our programs in this course, the class is about data structures and algorithms, not about Java, so the techniques that you learn here are applicable to programming in any language.