CS 4: Lecture 26
Wednesday, April 26, 2006

LISTS
=====
We can store a list of ints 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, and move all the ints from the old array to the new one.

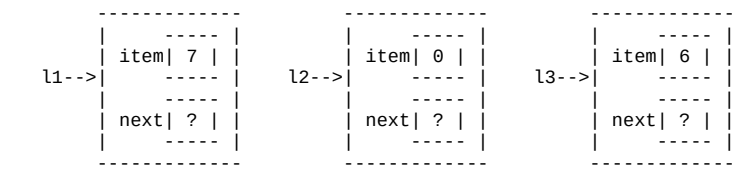
Second, if we want to insert an item at the beginning or middle of an array, we have to slide a lot of items over one place to make room. This takes time proportional to the length of the array.

LINKED LISTS (a recursive data type)
============
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.

```
  public class ListNode {          // ListNode is a recursive type
    public int item;
    public ListNode next;          // Here we're using ListNode before
  }                                //   we've finished declaring it.
```

Let's make some ListNodes.

```
  ListNode l1 = new ListNode(), l2 = new ListNode(), l3 = new ListNode();
  l1.item = 7;
  l2.item = 0;
  l3.item = 6;
```

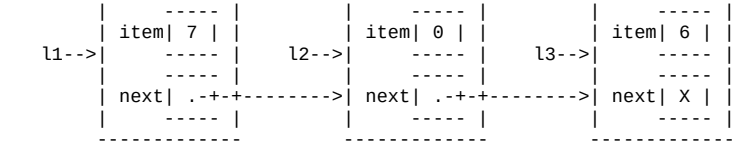


Now let's link them together.

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

What about the last node? We need a reference that doesn't reference anything. Recall that this is called "null".

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



Node Operations

To simplify programming, let's add some constructors to the ListNode class.

```
public ListNode(int i, ListNode n) {
  item = i;
  next = n;
}

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

With these constructors, we can build a list with one line of code.

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

Linked lists have several advantages over array-based lists. Inserting an item into the middle of a linked list takes just a small constant amount of time, so long as you already have a reference to the previous node (and don't have to walk through the whole list searching for it). The list can keep growing until memory runs out.

The following method inserts a new item into the list immediately after "this".

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

  l2.insertAfter(3);
```


However, linked lists aren't wholly better than 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 following method recursively finds the nth node in a list whose first item is "this".

```
  public ListNode nth(int position) {
    if (position == 1) {
      return this;
    } else if ((next == null) || (position < 1)) {
      return null;
    } else {                                                  // position > 1.
      return next.nth(position - 1);
    }
  }
```

Lists of Objects

For greater generality, let's change ListNodes so that each node contains not an int, but a reference to any Java object. In Java, we can accomplish this by declaring a reference of type Object.

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

A List Class

There are two problems with ListNodes.

(1) Suppose x and y are pointers to the same shopping list. Suppose we insert a new item at the beginning of the list thusly:

```
x = new ListNode("soap", 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 soap.

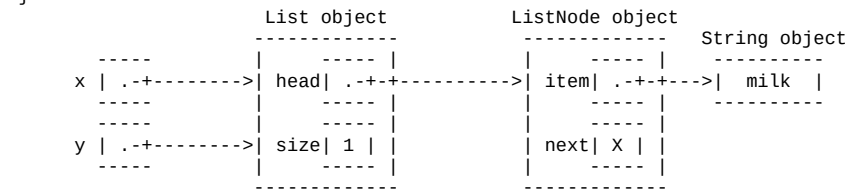
(2) How do you represent an empty list? The obvious way is "x = null". However, Java won't let you call a ListNode method--or any method--on a null object. If you write "x.nth(1)" when x is null, you'll get a run-time error, even though x is declared to be a ListNode. (This happens because Java can't do dynamic method lookup on a null reference.)

The solution is a separate List 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 List class, rather than the ListNode class.

```
public class List {
  protected ListNode head;
  protected int size;

  public List() {                    // Here's how to represent an empty list.
    head = null;
    size = 0;
  }

  public void insertFront(Object item) {
    head = new ListNode(item, head);
    size++;
  }
}
```



Now, when an item is inserted at the front of a List, every reference to that List can see the change. Another advantage of the List class is that it can keep a record of the List's size (number of ListNodes). Hence, the size can be determined more quickly than if the ListNodes had to be counted.

List Invariants

An _invariant_ is a fact about a data structure that always holds true, no matter what methods are called by external classes.

Another advantage of the List class (compared to just ListNodes) is that is enforces two useful invariants.
(1) An List's "size" variable is always correct.
(2) A list is never circularly linked; there is always a tail node whose "next" reference is null.

Both these goals are accomplished by making sure that only the methods of the List class can change the lists' internal data structures. List ensures this by two means:
(1) The fields of the List class (head and size) are declared "protected".
(2) No method of List returns an ListNode.

The first rule is necessary so that other classes can't change the fields and corrupt the List or violate invariant (1). The second rule prevents other classes changing list items, truncating a list, or creating a cycle in a list, thereby violating invariant (2).