Computer Science 523
Advanced Programming
The College of Saint Rose
Summer 2014

Topic Notes: Linked Lists and Iterators

Linked Lists

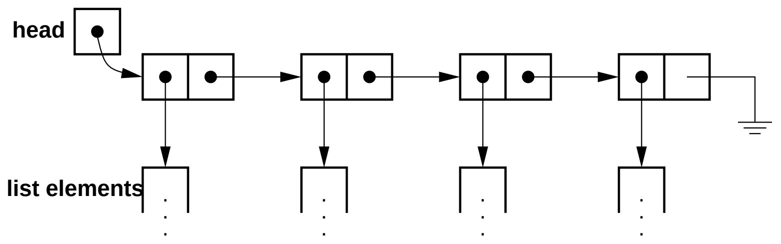
Just as we saw that other types of structures could be made generic, the same is true of the "ratio list" recursive structure we recently examined. It is an example of a fundamental data structure that is useful in many contexts called a *singly linked list*.

The name singly linked list might be self-explanatory, but let's say a little more about it. The reason for calling this data structure a list is because it allows us to store a list of items. We call it singly linked because the way it is represented allows us to traverse it in one direction only. To "walk through" the list we saw always considered `first` first; then we stepped through the list until we reached the end. We never moved backward through the list.

When we build a general-purpose data structure to hold a collection of items, there are certain types of functionality we want them to have. We need to be able to:

- add a new item to the collection
- remove an item from the collection
- tell whether the collection is empty
- get at the individual items in the collection

We can think of a general-purpose linked list:



See Example: SimpleLinkedList

The list structure is a reference to the first *list node*.

The structure that makes up a list node has two fields:

1. `value`: the list *element*, or value, which is stored at that list node's position in the list.
2. `next`: a pointer to the next list node, or `null` for the last node.

So the data fields for a very basic linked structure could look like this:

```
class SimpleListNode<E> {

  protected E value;
  protected SimpleListNode<E> next;
}

public class SimpleLinkedList<E> {

  protected SimpleListNode<E> head;
}
```

A few things to note right away about these:

- The `public` qualifier is not specified in the `SimpleListNode<E>` class definition, since we aren't allowing regular users to create one of these. Users can only a `SimpleLinkedList<E>`, and the methods of that class will in turn create the `SimpleListNode<E>` objects.
- The `SimpleListNode<E>` is a recursive data structure.

So if we want to create one of these, it's very easy. We just construct a `SimpleLinkedList<E>` and set its `head` to `null`.

```
  public SimpleLinkedList() {
    head = null;
  }
```

We then have a list with no elements.

How about adding an element? This involves two steps:

1. construct a new list node for the element (a node whose `value` contains the element we would like to store in our list)
2. insert the new list node into the list

Let's think about what this will mean. When we add our first element, let's call it A, we want this list to go from just an empty `head` reference, to a node pointed at by `head` which has A as its `value` and `null` as its `next`.

Now, we add another element, say B. We have two choices. We can add it either before or after A in the list.

Now, we add another element, C. We have three choices: beginning, middle, or end. In general, we can add at *position* 0, 1, or 2.

So let's see how we can support these with Java code.

Construction of the new list node is easy, once we know what to set its `next` pointer to. Here's a constructor:

```
    public SimpleListNode(E value, SimpleListNode<E> next) {
        this.value = value;
        this.next = next;
    }
```

We will soon see the need to be able to set and retrieve the element `value` from a list node and its `next` pointer. We'll call the accessors `value()` and `next()`, and the mutators `setValue()` and `setNext()`.

If our linked list is going to be as general as the `ArrayList` structure we have already been using, we will need to allow additions to any position in our list. Thus, we will develop a general `add` method that deals with all three of the cases described above: the start of the list, somewhere in the middle, and at the end.

We'll need to provide our `add` method with an index and an object to become our list element:

```
    public void add(int pos, E obj) {...}
```

Each step of the way, we need to provide some error checking. In particular, we can only add at item at position i if its addition will result in a list containing at least $i + 1$ elements (we will designate positions starting at 0, as is done with arrays and `ArrayList`s). However, our list as we are developing it initially does not have a direct way to count the number of elements it contains, so this check will be done in a few locations in the code. If at some point we determine that the position is invalid, the method will throw an `IndexOutOfBoundsException`, just like an array or `ArrayList`.

```
        if (pos < 0) {
            throw new IndexOutOfBoundsException("Attempt to add at negative positic
        }
```

Next, we check if there's an empty list. If so, `pos` should be 0. If not, we throw an exception.

Adding at position 0 is easy. We construct a new `SimpleListNode` containing our element, using the old `head` as its `next`. Note that this works for an empty list, in which case `head` is `null`, or the case when there is already a previous `head`, in which case the old `head` becomes the second node in the list.

```
        if (pos == 0) {
            head = new SimpleListNode<E>(obj, head);
            return;
        }
```

It gets more complicated if we want to insert in the middle or at the end (`pos` != 0). We need to search for the item after which we want to insert, then do the insertion.

```
        int i = 0;
        SimpleListNode<E> finger = head;
        while (i < pos-1) {
            i++;
            finger = finger.next();
            if (finger == null) {
                throw new IndexOutOfBoundsException("Attempt to add at position " +
            }
        }
        finger.setNext(new SimpleListNode<E>(obj, finger.next()));
```

There is also a “default” `add` method that doesn’t take a position parameter. In this case, we add at the start of the list. This is done quite simply by calling the more general `add` method already provided, passing a `pos` of 0.

Now that we can build up our lists, let’s consider a few accessors. First, `get`. Again, we’ll allow users to `get` the element at any position.

```
    public E get(int pos) {

        if (pos < 0) {
            throw new IndexOutOfBoundsException("Attempt to get from a negative pos
        }

        SimpleListNode<E> finger = head;
        int i = 0;

        if (head == null) {
            throw new IndexOutOfBoundsException("Attempt to get from an empty list'
        }

        while (i < pos) {
            i++;
            finger = finger.next();
            if (finger == null) {
                throw new IndexOutOfBoundsException("Attempt to get element " + pos
            }
        }
        return finger.value();
    }
```

We can write a `set` method almost identical to this, except that instead of returning the value at the desired position, we just set it and return the old value.

So now about `contains`? We need to search through looking for the element until we find it or find the end of the list.

The basic structure is the same as `get`. We have a "finger" tracking our progress through the list. Here, we never encounter an error condition and throw and exception – we always return `true` or `false`.

```
    public boolean contains(E obj) {

        // easy when the list is empty
        if (head == null) return false;

        // otherwise look for it
        SimpleListNode<E> finger = head;
        while (finger != null) {
            if (finger.value().equals(obj)) return true;
            finger = finger.next();
        }
        return false;
    }
```

Let's do an easy one: `size()`.

```
    public int size() {
        SimpleListNode<E> finger = head;
        int count = 0;

        // count up the number of list nodes until we get a null next
        while (finger != null) {
            count++;
            finger = finger.next();
        }

        return count;
    }
```

That was easy, but quite inefficient. We could alternately keep a count of the number of elements in the list and return that immediately, but that value would be extra memory required for every list we allocate, and we would have to update that count in all methods that modify the list.

Now, let's consider a harder one: `remove()`. We could remove items by value or by index. We'll just implement by index.

There are a number of cases to consider:

1. remove the only item from a list
2. remove the first item in a list from a list with at least two elements
3. remove the last item in a list from a list with at least two elements
4. remove an item from the middle of a list from a list with at least two elements

```
    public E remove(int pos) {
```

First, we make sure we're not removing from a negative position or from an empty list, and throw exceptions if so.

Next, we can take care of the first item case.

```
        if (pos == 0) {
            E retval = head.value();
            head = head.next();
            return retval;
        }
```

In other cases, we need to find the node containing the item we want to remove and adjust some references to point "around" the node we are removing.

So we need to have our "finger" on the element *before* the one we want to remove, since that's the one whose `next` pointer will need to be adjusted.

```
        // remove an item at a non-first position
        SimpleListNode<E> finger = head;
        int count = 0;
        // find the item before the one we want to remove
        while (count < pos-1) {
            count++;
            finger = finger.next();
            if (finger == null) {
                throw new IndexOutOfBoundsException("Attempt to remove element at i
            }
        }
        // finger is pointing to item pos-1
        // make sure there is something at pos
        if (finger.next() == null) {
            throw new IndexOutOfBoundsException("Attempt to remove element at inde
        }
        E retval = finger.next().value();
        finger.setNext(finger.next().next());
        return retval;
```

Removing everything is very simple.

```
    public void clear() {
        head = null;
    }
```

What about all those list nodes? We still have references to them! Not to worry, Java's garbage collector will clean them up.

However, not all languages are garbage collected like Java. In C or C++, you need to be careful to `free` (in C) or `delete` (in C++) all of the objects you no longer need.

Let's consider the complexity of our operations.

- `add(0)` : $O(1)$
- `add(i)` : $O(i)$
- `add(n)` : $O(n)$
- `get/set(0)` : $O(1)$
- `get/set(i)` : $O(i)$
- `get/set(n-1)` : $O(n)$
- `remove(0)` : $O(1)$
- `remove(i)` : $O(i)$
- `remove(n-1)` : $O(n)$
- `get` all values in sequence : $O(n^2)$ (hey, we need an `Iterator`! see below)
- `size()` : $O(n)$ (hey, we can do better if we remember this)

How do these compare to similar operations on `ArrayList`s?

- adding at the front is easier (`ArrayList` is $O(n)$ since all items need to be shifted up).

- adding at the end is harder (`ArrayList` is $O(1)$ except when the `ArrayList`'s internal array structure is full, in which case it is $O(n)$).
- adding in the middle, it depends where.
- the cost is consistent, though, since there is no reallocation and copying to grow the structure.
- removing at the front is easier (`ArrayList` is $O(n)$, since all items need to be shifted down).
- removing at the end is harder (`ArrayList` is $O(1)$).
- removing in the middle is similar.
- getting/setting an arbitrary value is harder (`ArrayList` is $O(1)$).

What about space usage?

- there are no empty slots like we have in `ArrayList`s
- but there's an extra reference for each object stored! That's $O(n)$ space overhead.

We still have a couple of problems with this implementation that we'd like to address. First, the $O(n^2)$ traversal is no good – we need an `Iterator`...

Iterators

How do we "visit" each item in a collection? With a `ArrayList`, or an array, it's easy. We can write a `for` loop:

```
public <T> void traverse(ArrayList<T> v) {
  int i;

  for (i=0; i<v.size(); i++) {
    T visitme = v.get(i);
    // do something with visitme
  }
}
```

But imagine if someone has changed the implementation of `ArrayList`. It no longer has an array, but a linked structure, like our `SimpleLinkedList`.

To get access to the n^{th} element, we need to visit the first $n-1$ elements. If our `ArrayList` contained one of these linked structures instead of an array, our traverse method suddenly becomes very inefficient.

This is not good. What is the complexity of `get()`? In order to get the item at position i, we have to start at the beginning and we have to follow links until we find the right element.

What we want to do is to use the previous value returned, and take the one pointed to by the list element we just used to get that previous value. But how? We don't have that information!

We often need a way of cycling through all of the elements of a data structure. We will use another object type, one which knows the details of the structure we are traversing and which can remember the status of the progress through our traversal. These are called *iterators* and Java provides exactly what we need: the `java.util.Iterator<E>` interface.

A data structure can create an object that implements the `Iterator` interface, which can be used to cycle through the elements. For example, built-in Java class `ArrayList` has method:

```
public Iterator<E> iterator()
```

that we can print out the elements of `ArrayList<E> v` as follows:

```
for (Iterator<E> iter=v.iterator(); iter.hasNext(); )
    System.out.println(iter.next());
```

Or in Java 5 and up, if our class implements the `Iterable` `interface` (which simply requires the method `iterator`) we can use a "for each" loop:

```
for (E item: v) {
  System.out.println(item);
}
```

Important Notes:

- Never change the state of a data structure with an active works, or you may end up in an infinite loop!
- There is also a `remove()` methos in Java's `Iterator` interface, but we will ignore that for now, as not all iterators provide it.
- `Iterator`s guarantee a predictable and consistent order of the elements returned.

Remember, an iterator must remember some state about the collection it's visiting. With an `ArrayList` iterator, we just needed to remember the index of the next item to be returned. Remembering the index doesn't help us with linked lists. We need to remember something about the internals of the list to make this work. The most useful thing to remember here is the list node – that "finger" we used in most of the methods we've looked at.

Again, it's not a public class, since no one except our `SimpleLinkedList` is allowed to construct one.

We need to have data to support the regular iterator operations, plus be able to reset the iterator, so we need to have our iterator remember the head of the list and the "finger":

```
    protected SimpleListNode<E> current;
    protected SimpleListNode<E> head;
```

and to construct one, we need to have the head of the list passed in:

```
    public SimpleListIterator(SimpleListNode<E> t) {
        head = t;
        current = head;
    }
```

So the `current` pointer always points to the next node whose value has *not yet been returned*. From this, we can construct the remaining methods:

```
    public boolean hasNext() {
        return current != null;
    }

    public E next() {
        E temp = current.value();
        current = current.next();
        return temp;
    }
```

And in the `SimpleLinkedList` class, we have a method to create one:

```
    public Iterator<E> iterator() {
        return new SimpleListIterator<E>(head);
    }
```

We also make our `SimpleLinkedList` implement `Iterable` so we can use it in "for each" loops.

In subsequent courses, you will see variations on linked lists (including doubly linked lists, which have references between adjacent list nodes in both directions), and more carefully consider the differences among arrays, `ArrayList`s, different types of linked lists, and more advanced data structures. While many of these structures provide the same operations and can often be used interchangeably, it is very important to know which will be most efficient for a given problem, both in terms of memory and computational costs. So stay tuned!

Lists in the Java API

Java's builtin API provides a variety of generic classes that perform list-like functionality. Many of these can be used interchangeably because they implement the `java.util.List<E>` interface.

The `List` interface defines a set of common operations that a number of API classes provide. Programmers can also write their own classes that satisfy the `List` interface. As long as a user of a class that implements the interface restricts his or her usage to the methods specified in the interface, different implementations can be swapped in with the only change being in the construction of the object that implements `List`. Consider this example:

See Example: ListInterfaceDemo

We create lists of three of the types that implement the `List` interface and then use them in various ways. The main thing to notice here is that the name of the actual type of the list being used (in this case, `ArrayList`, `Vector`, and `LinkedList`) appears only when constructing each list. We could change any of those to any other type that implements the `List` interface, and the remaining code would continue to work unchanged.

Recall that different underlying structures do different things more or less efficiently than others, so depending on which operations we expect to use, it might make more sense to use one structure over another. But by writing as much of our code using only the methods provided by an interface common to the options, we can switch among actual implementations later with minimal effort!