Lecture 8: Iterators and More Mutation

Contents

1 Traversing Lists 1
2 Motivating Iterators 2
3 Writing an Iterator 3
4 Writing Sum with an Iterator 4

Objectives

By the end of this lecture, you will know:

- How to allow iteration over a list without exposing implementation details

By the end of this lecture, you will be able to:

- Write a while loop to traverse a data structure in Java
- Implement an iterator for your LinkedList class

1 Traversing Lists

Let’s support some common methods on lists. We’ll start with size, assuming that we prefer to save the space of a size field, and instead write a method to traverse the list and compute the size each time someone asks for it. We could proceed recursively, starting from the first Node and continuing until null. However, Java also allows us to work iteratively. A while statement (also called a while loop) repeats the code inside it until its condition is false. Here is the skeleton of how we might write size iteratively:

```java
public int size() {
    Node current = start;

    while (current != null) {
        current = current.next;
    }
    return 0; // ???
}
```

When we say `Node current = start` at the beginning of the method, Java creates a new name, `current`, that refers to whatever Node object `start` points to at that moment. If the list is empty,
current will contain null and the while loop will never run. Then the method will return zero, which is what we’d expect. If the list is not empty, Java populates current with a reference to the first Node. This isn’t equal to null, so Java runs the loop body. Each time the loop body runs, current advances one step along the list, and the loop continues to run until current falls off the end of the list (i.e., is set to null).

This current = ... business looks exactly like what we’ve previously seen with creating and mutating object fields. The difference is that the name current is not a field, but a local variable. It belongs to the method, and only lasts as long as the method does. Java won’t understand if you try to talk about current outside of the size method.

There’s still a problem, though. How do we return the correct value for a non-empty list? We just add another local variable to store a count of how many Nodes get visited. Each iteration of the loop, we increase count by one:

```java
public int size() {
    Node current = start;
    int count = 0;

    while (current != null) {
        count = count + 1;
        current = current.next;
    }
    return count;
}
```

And that’s it. Every time the current reference advances, the count goes up by one, until we run out of Nodes and return how many were seen.

## 2 Motivating Iterators

Let’s try something a bit more complicated. Suppose that we want to write a method—outside the LinkedList class!—that takes in a list of integers and returns the sum of all elements in the list. You’ve done similar things before in CS 17—you’d just fold addition over the list. How can we do the same thing iteratively? We might imagine adapting the while loop we wrote for size, producing something like this:

```java
public int sum(LinkedList<Integer> lst) {
    Node current = lst.start;
    int s = 0;

    while (current != null) {
        s = s + current.item;
        current = current.next;
    }
    return s;
}
```

But it’s better not to. One of the principles of good Object-Oriented design is that external code shouldn’t need to be aware of, let alone use, internal implementation details. If we write sum
this way, we’re assuming that the list we receive is in fact a LinkedList. We’d like to write a sum method that works for any class that implements IList, no matter how it’s implemented. This means that using the next field is off the table.

Ok, but how do we iterate over the list if we can’t access the next field, or even trust that there’s an inner Node class at all? The answer is that external users of LinkedList can’t! At least not without a bit of extra help inside LinkedList.

3 Writing an Iterator

First, let’s figure out which parts of the above badly-designed sum method really belong in sum, and which in the LinkedList class. The lines where we create and add to the s variable are purpose-specific: they’re all about summing over integers. So we’ll keep them in the sum method. But what about current? It only exists to facilitate iterating over a list with this particular implementation. So we’ll have to move it into the LinkedList class somehow.

Should we make current a field of LinkedList? No! If we did that, a LinkedList could only support one iteration at a time—multiple external methods (say sum and a max method that keeps the highest value in the list) would always be looking at the same position in the list. Even more, they’d have to manually reset the field when they were done. Instead, we need to somehow provide the essence of current to many different external callers. To do so, we’ll allow any external method to obtain what’s called an iterator from a LinkedList. Each iterator keeps track of how far into the list it’s currently looked. We’ll add the iterator as another inner class within LinkedList, LinkedListIterator.

Iterators provide two methods: hasNext, which returns true if the iterator has not run out of elements to return, and next, which returns the next element and advances. Java provides a built-in Iterator interface that enforces those two methods exist. Finally, the new constructor accepts a particular LinkedList to start iterating over.

```java
class LinkedListIterator implements Iterator<T> {
    Node current;

    public LinkedListIterator(LinkedList<T> theList) {
        this.current = theList.getStart();
    }

    @Override
    public boolean hasNext() {
        return (this.current != null);
    }

    @Override
    public T next() {
        T item = this.current.item;
        this.current = this.current.next;
        return item;
    }
}
```

We’ll also had a method to the LinkedList class that manufactures an iterator for external use.
public Iterator<T> iterator() {
    return new LinkedListIterator(this);
}

Even better, we’ll modify the IList interface so that every list must provide an iterator method. (Again, Java provides a built-in interface to enforce this, called Iterable.)

public interface IList<T> extends Iterable<T> {
    public void addFirst(T newItem);
    public void addLast(T newItem);
    public T getFirst();
    public Iterator<T> iterator();
}

4 Writing Sum with an Iterator

Finally, we can write sum in proper Object-Oriented style. All our sum method has to do is obtain an iterator and accumulate the integers it returns. Even better, it no longer has to know for sure that it is being given a LinkedList—just that it is some class that implements IList<Integer>.

public int sum(IList<Integer> lst) {
    Iterator<Integer> iter = lst.iterator();
    int s = 0;

    while (iter.hasNext()) {
        s = s + iter.next();
    }
    return s;
}

Java has some syntax that makes this style of loop even easier. You can use a for loop to automatically iterate over any collection (more on this word later) that provides an iterator (i.e., implements the Iterable interface).

public int sum(IList<Integer> L) {
    int sum = 0;

    for (Integer aValue : L) {
        sum = sum + aValue;
    }
    return sum;
}

The for loop syntax creates a new local variable aValue and, for each item in L, it populates aValue with what the iterator returns and executes the body. Note that this local variable is even shorter-lived than the ones we used earlier in this lecture: the name only exists within the loop itself!
Iterators are an elegant way to expose the contents of a list to outside code without exposing implementation details. We’ll soon see several new reasons why this is important.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs018/feedback](http://cs.brown.edu/courses/cs018/feedback)