Lists

Contents

The List ADT

Our first ADT is the List: an ordered collection of items of some element type E. Note that this doesn't mean that the objects are in ***sorted*** order, it just means that each object has a ***position*** in the List, starting with position zero.

Recall that when we think about an ADT, we think about both the external and internal views. The external view includes the "conceptual picture" and the set of "conceptual operations". The conceptual picture of a List is something like this:

```
item 0
item 1
item 2
. . .
item n
```

and one reasonable set of operations is:

Operation	Description
void add(E item)	add item to the end of the List

void add(int pos, E item)	add item at position pos in the List, moving the items originally in positions pos through size()-1 one place to the right to make room (error if pos is less than 0 or greater than size())
boolean contains(E item)	return **true** iff item is in the List (i.e., there is an item x in the List such that x.equals(item))
int size()	return the number of items in the List
boolean isEmpty()	return **true** iff the List is empty
E get(int pos)	return the item at position pos in the List (error if pos is less than 0 or greater than or equal to size())
E remove(int pos)	remove and return the item at position pos in the List, moving the items originally in positions pos+1 through size()-1 one place to the left to fill in the gap (error if pos is less than 0 or greater than or equal to size())

Many other operations are possible; when designing an ADT, you should try to provide enough operations to make the ADT useful in many contexts, but not so many that it gets confusing. It is not always easy to achieve this goal; it will sometimes be necessary to add operations in order for a new application to use an existing ADT.

TEST YOURSELF #1

Question 1: What other operations on Lists might be useful? Define them by writing descriptions like those in the

table above.

Question 2: Note that the second add method (the one that adds an item at a given position) can be called with a position that is equal to size(), but for the get and remove methods, the position has to be ***less*** than size(). Why?

Question 3: Another useful abstract data type is called a Map. A Map stores unique "key" values with associated information. For example, you can think of a dictionary as a Map, where the keys are the words, and the associated information is the definitions.

What are some other examples of Maps that you use?

What are the useful operations on Maps? Define them by writing descriptions like those in the table above.

Java Interfaces

The List ADT operations given in the table above describe the **public interface** of the List ADT, that is, the information the someone would need to know in order to be able to use a List (and keep in mind that the user of a List does not - and should not - need to know any details about the **private implementation**).

Now let's consider how we can create an ADT using Java. Java provides a mechanism for separating public interface from private implementation: interfaces. A **Java interface** (i.e., an interface as defined by the Java language) is very closely tied to the concept of a public interface. A Java interface is a reference type (like a class) that contains only (public) method signatures and class constants. Java interfaces are defined very similarly to the way classes are defined. A `ListADT` interface with the List ADT operations we've described would be defined as follows:

```
public interface ListADT<E> {
    void add(E item);
    void add(int pos, E item);
    boolean contains(E item);
    int size();
    boolean isEmpty();
    E get(int pos);
    E remove(int pos);
}
```

Everything contained in a Java interface is public so you do not need to include the `public` keyword in the method signatures (although you can if you want). Notice that, unlike a class, an interface does not contain any method implementations (i.e., the bodies of the methods). The method implementations are provided by a Java class that **implements** the interface. To implement an interface named `InterfaceName`, a Java class must do two things:

1. include "`implements InterfaceName`" after the name of the class at the beginning of the class's declaration, and
2. for each method signature given in the interface `InterfaceName`, define a public method with the exact same signature

For example, the following class implements the `ListADT` interface (although it is a ***trivial*** implementation):

```
public class ListImplementation<E> implements ListADT<E> {
    private E myItem;

    public ListImplementation() {
        myItem = null;
    }

    public void add(E item) {
        myItem = item;
    }

    public void add(int pos, E item) {
        myItem = item;
    }

    public boolean contains(E item) {
        return false;
    }

    public int size() {
        return 0;
    }

    public boolean isEmpty() {
        return false;
    }

    public E get(int pos) {
        return null;
    }

    public E remove(int pos) {

        return null;
    }
}
```

Lists vs. Arrays

In some ways, a List is similar to an array: both Lists and arrays are ordered collections of objects and in both cases you can add or access items at a particular position (and in both cases we consider the first position to be position zero). You can also find out how many items are in a List (using its size method), and how large an array is (using its `length` field).

The main advantage of a List compared to an array is that, whereas the size of an array is fixed when it is created (e.g., `int[] A = new int[10]` creates an array of integers of size 10 and you cannot store more than 10 integers in that array), the size of a List can change: the size increases by one each time a new item is added (using either version of the `add` method) and the size decreases by one each time an item is removed (using the `remove` method).

For example, suppose we have an `ArrayList` class that implements the `ListADT` interface. Here's some code that reads strings from a file called `data.txt` and stores them in a `ListADT` named `words`, initialized to be an `ArrayList` of `String`s:

```
ListADT<String> words = new ArrayList<String>();
File infile = new File("data.txt");
Scanner sc = new Scanner(infile);
while (sc.hasNext()) {
    String str = sc.next();
    words.add(str);
}
```

If we wanted to store the strings in an array, we'd have to know how many strings there were so that we could create an array large enough to hold all of them.

One disadvantage of a List compared to an array is that whereas you can create an array of any size and then you can fill in any element in that array, a new List always has size zero and you can never add an object at a position greater than the size. For example, the following code is fine:

```
String[] strList = new String[10];
strList[5] = "hello"; !
```

but this code will cause a runtime exception:

```
ListADT<String> strList = new ArrayList<String>();
strList.add(5, "hello");       // error! can only add at position 0
```

Another (small) disadvantage of a List compared to an array is that you can declare an array to hold any type of item, including primitive types (e.g., `int`, `char`), but a List only holds `Object`s. Thus the following causes a compile-time error:

```
List<int> numbers = new ArrayList<int>();

% javac Test.java
Test.java:9: unexpected type
found   : int
required: reference
        List<int> numbers = new ArrayList<int>();
             ^
```

Fortunately, for each primitive type, there is a corresponding "box" class (also sometimes called a **wrapper class**). For example, an `Integer` is an object that contains one `int` value. Java has a feature called **auto-boxing** that automatically converts between `int` and `Integer` as appropriate. For example,

```
ListADT<Integer> numbers = new ArrayList<Integer>();

// store 10 integer values in list numbers
for (int k = 0; k < 10; k++) {
    numbers.add(k); // short for numbers.add(new Integer(k));
}

// get the values back and put them in an array
int[] array = new int[10];
for (int k = 0; k < 10; k++) {
    array[k] = numbers.get(k); // short for array[k] = numbers.get(k).intValue();
}
```

(Older versions of Java required that your code do the boxing up of a primitive value into the corresponding wrapper class explicitly; for example, here's the code that does the same thing as the example above using explicit boxing and unboxing:

```
    ListADT<Integer> numbers = new ArrayList<Integer>();

    // store 10 integer values in list numbers
    for (int k = 0; k < 10; k++) {
        numbers.add(new Integer(k));
    }

    // get the values back and put them in an array
    int[] array = new int[10];
    for (int k = 0; k < 10; k++) {
        array[k] = numbers.get(k).intValue();
    }
```

It's useful to know this since you may encounter code written using earlier versions of Java.)

TEST YOURSELF #2

Question 1: Assume that variable words is a List containing k strings, for some integer k greater than or equal to zero. Write code that changes words to contain 2*k strings by adding a new copy of each string right after the old copy. For example, if words is like this before your code executes:

"happy", "birthday", "to", "you"

then after your code executes words should be like this:

"happy", "happy", "birthday", "birthday", "to", "to", "you", "you"

Question 2: Again, assume that variable words is a List containing zero or more strings. Write code that removes from words all copies of the string "hello". Be sure that your code works when words has more than one "hello" in a row.

solution

Implementing the List ADT

Now let's consider the "private" or "internal" part of a List ADT. We will consider two ways to implement the `ListADT` interface: using an array and using a linked list (the former is covered in this set of notes, the latter in another set of notes). Here's an outline of an implementation of the `ListADT` interface that uses an array to store the items. We will also simplify the implementation to consider only lists of `Object`s, not lists of specific classes such as `String`, `Integer`, etc. We will call our class `SimpleArrayList`. The bodies of the methods are not filled in yet.

```
public class SimpleArrayList implements ListADT<Object> {

  // *** fields ***
    private Object[] items; // the items in the List
    private int numItems;   // the number of items in the List

  // *** constructor ***
    public SimpleArrayList() { ... }

  //*** required ListADT methods ***

    // add items
    public void add(Object item) { ... }
    public void add(int pos, Object item) { ... }

    // remove items
    public Object remove(int pos) { ... }

    // get items
    public Object get (int pos) { ... }

    // other methods
    public boolean contains (Object item) { ... }
    public int size() { ... }
    public boolean isEmpty() { ... }
}
```

Note that the public methods provide the "external" view of the List ADT. Looking only at the signatures of the public methods of the `ListADT` interface (the method names, return types, and parameters), plus the descriptions of what they do (e.g., as provided in the table above), a programmer should be able to write code that uses `ListADT`s and `ArrayList`s. It is ***not*** necessary for a client of the `SimpleArrayList` class to see how the `SimpleArrayList` methods are actually implemented. The private fields and the bodies of the methods of the `SimpleArrayList` class provide the "internal" view - the actual implementation - of the List ADT.

Implementing the `add` methods

Now let's think about how to implement some of the `SimpleArrayList` methods. First we'll consider the `add` method that has just one parameter (the object to be added to the List). That method adds the object to the end of the List. Here is a conceptual picture of what `add` does:

```
BEFORE:  item 0    item 1    item 2    ...    item n


AFTER:   item 0    item 1    item 2    ...    item n   NEW ITEM
```

Now let's think about the actual code. First, note that the array that stores the items in the List (the `items` array) may be full. If it is, we'll need to:

1. get a new, larger array, and
2. copy the items from the old array to the new array.

Then we can add the new item to the end of the List.

Note that we'll also need to deal with a full array when we implement the other `add` method (the one that adds a new item in a given position). Therefore, it makes sense to implement handling that case (the two steps listed above) in a separate method, `expandArray`, that can be called from both `add` methods. Since handling a full array is part of the ***implementation*** of the `SimpleArrayList` class (it is not an operation that users of the class need to know about), the `expandArray` method should be a `private` method.

Here's the code for the first `add` method and for the `expandArray` method.

```
//*********************************************************************
// add
//
// Given: Object item
//
// Do:    Add item to the end of the List
//
// Implementation:
//   If the array is full, replace it with a new, larger array;
//   store the new item after the last item
//   and increment the count of the number of items in the List.
//*********************************************************************
public void add(Object item) {
    // if array is full, get new array of double size,
    // and copy items from old array to new array
    if (items.length == numItems) {
        expandArray();
    }

    // add new item; update numItems
```

```
        items[numItems] = item;
        numItems++;
    }


    //**********************************************************************
    // expandArray
    //
    // Do:
    //   o Get a new array of twice the current size.
    //   o Copy the items from the old array to the new array.
    //   o Make the new array be this List's "items" array.
    //**********************************************************************
    private void expandArray() {
        Object[] newArray = new Object[numItems*2];
        for (int k = 0; k < numItems; k++) {
            newArray[k] = items[k];
        }
        items = newArray;
    }
```

In general, when you write code it is a good idea to think about special cases. For example, does `add` work when the List is empty? When there is just one item? When there is more than one item? You should think through these cases (perhaps drawing pictures to illustrate what the List looks like before the call to `add` and how the call to `add` affects the List). You can then decide if the code works as is or if some modifications are needed.

Now let's think about implementing the second version of the `add` method (the one that adds an item at a specified position in the List). An important difference between this version and the one we already implemented is that for this version a bad value for the `pos` parameter is considered an error. In general, if a method detects an error that it doesn't know how to handle, it should throw an exception. (Note that exceptions should ***not*** be used for other purposes like exiting a loop.) More information about exceptions is provided in a separate set of notes.

So the first thing our `add` method should do is check whether parameter `pos` is in range and if not, throw an `IndexOutOfBoundsException`. If `pos` is OK, we must check whether the `items` array is full and if so, we must call `expandArray`. Then we must move the items in positions `pos` through `numItems` - 1 over one place to the right to make room for the new item. Finally, we can insert the new item at position `pos` and increment `numItems`. Here is the code:

```
    //**********************************************************************
    // add
    //
    // Given: int pos, Object item
```

```
//
// Do:    Add item to the List in position pos (moving items over to the right
//        to make room).
//
// Exceptions:
//        Throw IndexOutOfBoundsException if pos<0 or pos>numItems
//
// Implementation:
//   1. check for bad pos
//   2. if the array is full, replace it with a new, larger array
//   3. move items over to the right
//   4. store the new item in position pos
//   5. increment the count of the number of items in the List
// ***********************************************************************
public void add(int pos, Object item) {
    // check for bad pos and for full array
    if (pos < 0 || pos > numItems) {
        throw new IndexOutOfBoundsException();
    }
    if (items.length == numItems) {
        expandArray();
    }

    // move items over and insert new item
    for (int k = numItems; k > pos; k--) {
        items[k] = items[k-1];
    }
    items[pos] = item;
    numItems++;
}
```

TEST YOURSELF #3

Question 1: Write the `remove` and `get` methods.

solution

Implementing the constructor

Now let's think about the constructor; it should initialize the fields so that the List starts out empty. Clearly, `numItems` should be set to zero. How about the `items` field? It could be set to `null`, but that would mean another special case in the `add` methods. A better idea would be to initialize `items` to refer to an array with some initial size, perhaps specified using a class constant (i.e., a static final field), so that the initial size could be easily changed.

Below is the code for the constructor (including the declaration of the class constant for the initial size). This code uses 10 as the initial size. In practice, the appropriate initial size will probably depend on the context in which the `ArrayList` class is used. The advantage of a larger initial size is that more `add` operations can be performed before it is necessary to expand the array (which requires copying all items). The disadvantage is that if the initial array is never filled, then memory is wasted. The requirements for memory usage and runtime performance of the application that uses the `SimpleArrayList` class, as well as the expected sizes of the Lists that it uses, should be used to determine the appropriate initial size.

```
private static final int INITSIZE = 10;

//*****************************************************************
// SimpleArrayList constructor
//
// initialize the List to be empty
//*****************************************************************
public SimpleArrayList() {
    numItems = 0;
    items = new Object[INITSIZE];
}
```

The Java API and Lists

The Java API contains many interfaces, including interfaces that are very similar to many of the ADTs we will be talking about. For example, the `Java.util` package provides a `List` interface (with many more methods than the ones given in our `ListADT` interface), as well as a number of classes that implement that interface, including the `ArrayList` class and the `LinkedList` class. The `SimpleArrayList` implementation we just developed is meant to give you insight into a way that Java's `ArrayList` class might be implemented.

Iterators

What are iterators?

When a client uses an abstract data type that represents a collection of items (as the List interface does), they often need a way to **iterate** through the collection, i.e., to access each of the items in turn. Our get method can be used to support this operation. Given a List L, we can iterate through the items in the List as follows:

```
for (int k = 0; k < L.size(); k++) {
    Object ob = L.get(k);
    ... do something to ob here ...
}
```

However, a more standard way to iterate through a collection of items is to use an

`Iterator`, which is an interface defined in `java.util`. Every Java class that implements the `Iterable` interface (which includes classes that implement the subinterface `Collection`) provides an `iterator` method that returns an `Iterator` for that collection.

The way to think about an `Iterator` is that it is a finger that points to each item in the collection in turn. When an `Iterator` is first created, it is pointing to the first item; a `next` method is provided that lets you get the item pointed to (also advancing the pointer to point to the next item) and a `hasNext` method lets you ask whether you've run out of items. For example, assume that we have added an `iterator` method (that returns an `Iterator`) to our `ArrayList` class and that we have the following list of words:

```
apple    pear    banana   strawberry
```

If we create an `Iterator` for the List, we can picture it as follows, pointing to the first item in the List:

```
apple    pear    banana   strawberry

  ^
  |
```

Now if we call `next` we get back the word "apple" and the picture changes to:

```
apple    pear    banana   strawberry

           ^
           |
```

After two more calls to `next` (returning "pear" and "banana") we'll have:

```
apple    pear    banana   strawberry

                               ^
                               |
```

A call to `hasNext` now returns `true` (because there's still one more item we haven't accessed yet). A call to `next` returns "strawberry" and our picture looks like this:

```
apple    pear    banana   strawberry

                                         ^
                                         |
```

The iterator has fallen off the end of the List. Now a call to `hasNext` returns `false` and if we call `next`, we'll get a `NoSuchElementException`.

The Java interface `List<E>` has a method `iterator()` that returns an object of type `Iterator<E>`. You can use it to iterate through the elements of a list with a `while` loop like this:

```
List<String> words = new ArrayList<String>();
... // add some words to the list
Iterator<String> itr = words.iterator();
while (itr.hasNext()) {
    String nextWord = itr.next();
    ... // do something with nextWord
}
```

or with a `for` loop like this (note that the **increment** part of the loop header is empty, since the call to the iterator's `next` method inside the loop causes it to advance to the next item):

```
List<String> words = new ArrayList<String>();
... // add some words to the list
for (Iterator<String> itr = words.iterator(); itr.hasNext(); ) {
    String nextWord = itr.next();
    ... // do something with nextWord
}
```

Java also provides **extended for-loops** (sometimes also referred to as **for-each loops**), which are another way to iterate through lists. The code looks like this:

```
List<String> words = new ArrayList<String>();
... // add some words to the list
for (String nextWord : words) {
    ... // do something with nextWord
}
```

In this context, the colon '`:`' is pronounced "in", so the `for` statement would be read out loud as "for each `String nextWord` in `words`, do ...".

Implementing iterators

The easiest way to implement an iterator for the `SimpleArrayList` class is to define a new class (e.g., called `SimpleArrayListIterator`) with two fields:

1. the `List` that's being iterated over, and
2. the index of the current item (the item the "finger" is currently pointing to).

We also define a new `iterator` method for the `SimpleArrayList` class; that method calls the `SimpleArrayListIterator` class's constructor, passing the `SimpleArrayList` itself:

```
//*********************************************************************
// iterator
//
// return an iterator for this List
//*********************************************************************
public Iterator<Object> iterator() {
    return new SimpleArrayListIterator(this);
}
```

The `Iterator` interface is defined in `java.util`, so you need to include:

```
import java.util.*;  // or import java.util.Iterator;
```

at the beginning of `SimpleArrayList.java` and `SimpleArrayListIterator.java` as well as in any other file that contains code that uses `Iterator`s.

The `SimpleArrayListIterator` class is defined to implement Java's `Iterator` interface and therefore must implement three methods: `hasNext` and `next` (both discussed above) plus an optional `remove` method. If you choose not to implement the `remove` method, you still have to define it, but it should simply throw an `UnsupportedOperationException`. If you choose to implement the `remove` method, then it should remove from the `SimpleArrayList` the last (i.e., most recent) item returned by the iterator's `next` method or it should throw an `IllegalStateException` if the `next` method hasn't yet been called.

Here is code that defines the `SimpleArrayListIterator` class (note that we have chosen not to implement the `remove` operation):

```
import java.util.Iterator;
import java.util.NoSuchElementException;

public class SimpleArrayListIterator implements Iterator<Object> {

    // *** fields ***
    private SimpleArrayList list;  // the list we're iterating over
    private int curPos;            // the position of the next item

    // *** constructor ***
    public SimpleArrayListIterator(SimpleArrayList list) {
        this.list = list;
        curPos = 0;
    }

    // *** methods ***

    public boolean hasNext() {
        return curPos < list.size();
    }

    public Object next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        Object result = list.get(curPos);
        curPos++;
        return result;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Testing

One of the most important (but unfortunately most difficult) parts of programming is testing. A program that doesn't work as specified is ***not*** a good program and in some contexts can even be dangerous and/or cause significant financial loss.

There are two general approaches to testing: **black-box** testing and **white-box** testing. For black-box testing, the testers know nothing about how the code is actually implemented (or if you're doing the testing yourself, you pretend that you know nothing about it). Tests are written based on what the code is supposed to do and the tester thinks about issues like:

- testing all of the operations in the interface
- testing a wide range of input values, especially including "boundary" cases
- testing both "legal" (expected) inputs as well as unexpected ones

So if you were to do black-box testing of the `ArrayList` class, you should be sure to:

- call all of the `ArrayList` methods
- try each operation on an `ArrayList` with no items, with exactly one item, and with many items
- include calls that should cause exceptions (e.g., adding an item at position -1, removing an item at a position 1 past the end of the List, and adding an item at a position more than 1 past the end of the List).

For white-box testing, the testers have access to the code and the usual goal is to make sure that every line of code executes at least once. So for example, if you were to do white-box testing of the `ArrayList` class, you should be sure to write a test that causes the array to become full so that the `expandArray` method is tested (something a black-box tester may not test, since they don't even know that `ArrayLists` are implemented using arrays).

You will be a much better programmer if you learn to be a good tester. Therefore, testing will be an important factor in the grades you receive for the programming projects in this class.

Back

Next: Continue with Exceptions