CMPSCI 187: Programming With Data Structures

Lecture #21: Array-Based Lists
David Mix Barrington
26 October 2012

Array-Based Lists

- Comparing Objects: `equals` and `Comparable`
- Lists: Unsorted, Sorted, and Indexed
- Assumptions About Lists
- The List Interfaces
- Iterators
- The `ArrayUnsortedList` Class
- The `ArraySortedList` Class

Comparing Objects: `equals` and `Comparable`

- We now begin our study of **lists**, one of the most general types of collections. Our basic operations for lists will include testing whether a given element is already in a list (the `contains` method). Some of our lists will be sorted, which means that we have defined what it means for one element to be "less than" another. This raises questions -- what does it mean for two objects to be equal, or for one to be less than another?

- If `x` and `y` are two objects in Java, we know that `x == y` is true if and only if `x` and `y` are *two names for the same object*. By contrast, it's possible for two different primitives to have the same value, and == tells us about this.

- "Having the same value" is a concept that depends on the type of the objects. We thus define an `equals` method for any class where we need to make such judgements. Since `equals` is defined as a method in the `Object` class and every other class extends `Object`, we can always use `equals`. If it has not been redefined for that class, it gives the same result as ==.

The `Comparable` Interface

- A **total order** on a class is a definition of when one object in the class is "less than or equal to" another. Once we have defined both this concept and that of equality, definitions of "greater than", "greater than or equal to", and "less than" follow directly. (Total orders must follow some rules that you'll see in CMPSCI 250, such as **transitivity** and **trichotomy**.)

- We'd like to write programs that deal with any class that has a total order, but arbitrary objects are not guaranteed to have one. The `Comparable<T>` interface requires that a class have a method `public int compareTo (T other)`. This returns a negative number if the calling object is less than other, returns 0 if they are equal, and returns a positive number if the calling object is greater. The generic definition allows objects to be compared only to other objects in the same class.

- We could imagine different ways to order elements of the same class -- you'll see this in Project #4. A class can have only one `compareTo` method, but it's possible to define different **comparator** objects for the same class.

Lists: Unsorted, Sorted, and Indexed

- A list is a linear data structure, where each element except the last has a **successor** and each element except the last has a **predecessor**. Every list also has a size, the number of elements in it.

- A list is **sorted** if the successor and predecessor properties are consistent with the `compareTo` method of the elements -- each element is “less than or equal to” its successor according to that method. A list without this property is unsorted -- it still has an order given by the successor and predecessor properties, but that order has no meaning in terms of the elements themselves.

- A list can also be **indexed**, meaning that we can access elements directly by their position in the list, or **index**. In an indexed list, we would have methods to “return the fourth element” in the order given by successor and predecessor.

Assumptions About Lists

- DJW list a number of assumptions about their lists to simplify the treatment:
- They are unbounded -- if implemented with arrays, the arrays resize.
- Duplicate elements (where one `equals` the other) are allowed. Finding one equal element is as good as finding any other.
- A null element cannot be a member of a list.
- Operations generally report success or failure by returning a boolean, not by throwing an exception on failure (except for a bad index in an indexed list).
- Sorted lists are in increasing (more precisely, non-decreasing) order. Indexed lists have indices ranging from 0 to the size - 1, with no gaps.
- The equals and compareTo methods are consistent with each other.

The List Interfaces

- While sorted and unsorted lists differ in many respects, the names of their operations are the same and thus the same interface may be used for both.

- We often want to **iterate** through a list, processing each element in turn. If we reset the list and then call `getNext` a number of times equal to `size`, we are guaranteed to see all the elements. A precondition for `getNext` is that no add or remove operations have occurred since the last reset.

```
public interface ListInterface<T> {
   int size( );
   void add (T element);
   boolean contains (T element);
   boolean remove (T element); // return value for success
   T get (T element); // null if element not there
   String toString( );
   void reset( ); // sets current position to beginning
   T getNext( ); // advances current position, wrapping around
}
```

The Indexed List Interface

- In an indexed list, we have additional commands to add, read, or remove elements at particular positions in the list. We throw an exception if the index in our parameter is outside the range currently filled with elements.

- The add and remove methods insert or delete an element at a particular position and change the index of all higher-numbered elements to reflect that change (to make room for the new element or to fill in the gap).

```
public interface IndexedListInterface<T>
      extends ListInterface<T> {
   void add (int index, T element); // higher elements move up
   T set (int index, T element); // returns former value
   T get (int index); // exception for bad index
   int IndexOf (T element); // index of first one, -1 if none
   T remove (int index); // higher elements move down into gap
}
```

Iterators

- The Java Collections classes for lists are generally similar to those in DJW with one prominent exception. Java provides the ability to traverse lists with an **iterator** object. An iterator is an object attached to a collection that has methods that refer to the collection. Lists implement the `Iterable` interface, which requires a method `iterator` that creates an `Iterator` object.

- When an iterator object for a list is created, its "current position" is at the beginning of the list. The `next` method returns the next element of the list and advances the current position -- the remove method does the same but also takes the returned element out of the list. This can continue until the end of the list, which can be detected with the hasNext method. (And you need `hasNext` to guard against an exception from calling `next` when at the end.) Rather than reset an iterator, you create a new one.

- While list iterators usually give their elements in the list order, there is no guarantee of this for iterators -- they are required only to return each element in the list the exact number of times it occurs in the list.

The `ArrayUnsortedList` Class

- As usual, our data structure has an array of T's with constructors to create it and a method to increase its size as needed.

```
public class ArrayUnsortedList<T>
      implements UnsortedListInterface<T> {
   protected final int DEFCAP = 100;
   protected int origCap;
   protected T[ ] list;
   protected int numElements = 0, currentPos, location;
   protected boolean found; // set by find method
   public ArraySortedList(int origCap) {
      list = (T[ ]) new Object[origCap];} // ignore warning
   public ArraySortedList( ) {
      this(DEFCAP); origCap = DEFCAP;}
   protected void enlarge( )
      T[ ] larger = (T[ ]) new Object[list.length + origCap];
      for (int i = 0; i < numElements; i++)
         larger[i] = list[i];
      list = larger;}
```

Methods of ArrayUnsortedList

- Here `find` is an auxiliary method used by `remove`, `contains`, and `get`.

```
protected void find (T target) {
   location = 0;
   while (location < numElements) {
      if (list[location].equals(target)) {
         found = true; return;}
      else location++;}}

public void add (T element) {
   if (numElements == list.length) enlarge( );
   list[numElements] = element;
   numElements++;}

public boolean remove (T element) {
   find (element);
   if (found) {list[location] = list numElements - 1;}
      list[numElements - 1] = null;
      numElements--;}
   return found;}
```

More Methods of `ArrayUnsortedList`

```
public int size ( ) {return numElements;}
```

```
public boolean contains (T element) {
   find (element); return found;}

public T get (T element) {
   find (element);
   if (found) return list[location];
   else return null;}

public String toString( ) {
   String listString = "List:\n";
   for (int i = 0; i < numElements; i++)
      listString += "  " + list[i] + "\n";
   return listString;}

public void reset( ) {currentPos = 0;}

public T getNext( ) {
   T next = list[currentPos];
   if (currentPos == numElements - 1) currentPos = 0;
   else currentPos++;
   return next;}
```

The ArraySortedList Class

- The `add` method has a bunch of casts because the compiler doesn't know that the type `T` had better implement `Comparable<T>`.

```
public class ArraySortedList<T> extends ArrayUnsortedList<T>
      implements ListInterface<T> {
   //no new fields, constructors are just "super"

   public void add (T element) {
      T listElement;
      int location = 0; // local variable, same name as i.v.
      if (numElements == list.length) enlarge( );
      while (location < numElements) {
         listElement = (T) list[location];
         if (((Comparable) listElement).compareTo(element) < 0)
            location++;
         else break;}
      for (int index = numElements; index > location; index--)
         list[index] = list[index - 1];
      list[location] = element;
      numElements++;}
```

The Rest of `ArraySortedList`

- The `find`, `contains`, `get`, `toString`, `reset`, and `getNext` methods are all inherited from `ArrayUnsortedList`. Except for `toString`, they each run in O(1) time.

- The add and remove methods run in O(N) time on a list with N elements in the worst case, because of elements being moved to make room or fill a gap.

```
public boolean remove (T element) {
   find (element);
   if (found) {
      for (int i = location; i <= numElements - 1; i++)
         list[i] = list[i + 1];
      list[numElements - 1] = null;
      numElements--;}
   return found;}}
```