CMPSCI 187: Programming With Data Structures

Lecture #19: Implementing Lists With Arrays
Guest Lecturer: Prof. Brian Levine
24 October 2011

Implementing Lists With Arrays

- Ordinary or Circular Arrays?
- L&C's `ArrayList` Class
- The `find`, `remove`, and `contains` Methods
- Building an Iterator
- Adding to an Ordered List
- The `addAfter` Method for an Unordered List

Ordinary or Circular Arrays?

- A stack was easy to implement with an array because changes to the data occurred only at one end of the sequence of entries. We could fix one end at index 0, and let the other end move with pushes and pops.

- With a queue, dropout stack, or deque, we needed to let both ends of the sequence move in order to allow additions and removals at each end. A circular array handled this pretty cleanly, taking O(1) time except for resizing.

- With a list, we need to be able to insert or remove elements anywhere in the sequence, which is impossible without shifting many elements in the worst case. (In the next discussion we'll look at keeping some empty locations in reserve throughout the array, but this method still can't prevent massive shifting (or resizing) in the worst case. Since circular arrays give us no advantage, we will implement lists with ordinary arrays and sometimes need O(n) time for a basic operation.

L&C's `ArrayList` Class

- This is *not* the `ArrayList` class from `java.util` that was used in CMPSCI 121, but a new class written by L&C as their basic array implementation of a list. We'll give implementations of the methods common to all lists, then include the different versions of the add methods.

```
public class ArrayList<T> implements ListADT<T>, Iterable<T> {
   protected final int DEFAULT_CAPACITY = 100;
   private final int NOT_FOUND = -1;
   protected int rear;
   protected T[ ] list;
   public ArrayList ( ) {
      rear = 0; list = (T[ ]) new Object [DEFAULT_CAPACITY];}
   public ArrayList (int capacity) {
      rear = 0; list = (T[ ]) new Object [capacity];}
```

The `find`, `remove`, and `contains` Methods

- Here `find` is a helper method used in the other two public methods

```
private int find (T target) {
   int scan = 0; result = NOT_FOUND;
   boolean found = false;
   if (!isEmpty( )) while (!found && (scan < rear))
      if (target.equals(list[scan]) found = true else scan++;
   if (found) result = scan;
   return result;}

public boolean contains (T target) {
   return (find (target) != NOT_FOUND);}

public T remove (T element) throws ElementNotFoundException {
   T result; int index = find (element);
   if (index == NOT_FOUND) throw new ElementNotFoundException( );
   result = list[index]; rear--;
   for (int scan = index; scan < rear; scan++)
      list[scan] = list[scan + 1];
   list[rear] = null; return result;}
```

Building an Iterator

- Remember that `ListADT` has a method `iterator` that returns an `Iterator` object attached to the list, that supports `hasNext` and `next`. We define a class of iterators that will work for any array objects. Note that the iterator delivers the elements in their array order, though it doesn't have to.

```
public Iterator<T> iterator( ) {
   return new ArrayIterator<T> (list, rear);}

public class ArrayIterator<T> implements Iterator<T> {
   private int count, current; private T[ ] items;
   public ArrayIterator (T[ ] collection, int size) {
      items = collection; count = size; current = 0;}
   public boolean hasNext( ) {
      return (current < count);}
   public T next( ) throws NoSuchElementException {
      if (!hasNext( )) throw new NoSuchElementException( );}
      current++;
      return items[current - 1];}}
```

Adding to an Ordered List

- We find the place where the new element must go, shift any later elements to make room for it, and put it in.

- We cast the element to a `Comparable<T>`, getting a `ClassCastException` if its true class does not implement `Comparable<T>.` Normally we design our class `T` to implement that interface.

```
public void add (T element) {
   if (size( ) == list.length) expandCapacity( );
   Comparable<T> temp = (Comparable<T>) element;
   int scan = 0;
   while (scan < rear && temp.compareTo (list[scan]) > 0)
      scan++;
   for (int scan2 = rear; scan2 > scan; scan2--)
      list[scan2] = list[scan2 - 1];
   list[scan] = element;
   rear++;}
```

The `addAfter` Method for an Unordered List

- The `addtoFront` and `addToRear` methods of `ArrayList` are the same as those for deques. We need one more method `addAfter` that inserts its first parameter after its second, if the second one is in the list.

```
public void addAfter (T element, T target)
             throws ElementNotFoundException {
   if (size( ) == list.length) expandCapacity( );
   int scan = 0;
   while (scan < rear && !target.equals (list[scan]))
      scan++;
   if (scan == rear) throw new ElementNotFoundException( );
   scan++;
   for (int scan2 = rear; scan2 > scan; scan2--)
      list[scan2] = list[scan2 - 1];
   list[scan] = element;
   rear++;}
```