CmpSci 187: Programming with Data Structures Spring 2015

Lecture #14, Lists: The Abstraction and Array Implementation

John Ridgway

March 24, 2015

1 Comparing Objects in Java

Comparing Objects in Java

- We are going to look at lists, one of the most general types of collections.
- Some of our lists will be sorted, requiring that we know when one element is "less than," "equal to", or "greater 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` refer to the same object.
- "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 judgments.

The `equals` Method

- Since `equals` is defined as a method in the `Object` class and every other class extends `Object`, we can always use `equals`.
- If `equals` has not been redefined by a class (or some superclass), it gives the same result as `==`.
- We've seen the `equals` method of `String`, which tests whether the two strings have the same characters in the same order.

Total Order

- A total order on a class defines a relationship between every pair x and y of instances of that class; it must be true that x is "less than or equal to" y, or vice versa.

- The $\leq$ relationship on the integers is a natural example of a total order.
- Given a way of determining a total order and of determining equality, the 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.)

Comparing Objects by Order

- Some classes have a natural total ordering, for instance `Integer`.
- We use the `Comparable<T>` interface to mark such a class. It has one method: `int compareTo(T that)`. The method returns a negative number if `this` is "less than" `that`, a positive number if it's "greater", and 0 if they're "equal".
- `compareTo` must be consistent with `equals`, i.e., `this.compareTo(that)` is zero if, and only if, `this.equals(that)`.

Clicker Question #1

Suppose that class `Dog` has a `compareTo` method as described above, and that `c`, `d`, and `w` refer to three different `Dog` objects, no two of which are `equal`. Let: `x = c.compareTo(d) + d.compareTo(w)` `y = c.compareTo(w)` Which situation is not possible?

A. `x < 0, y > 0`

B. `x > 0, y = 0`

C. `x = 0, y > 0`

D. `x < 0, y < 0`

Comparing Objects by Order

- The generic definition allows objects to be compared only to other objects in the same class. A generic class like `Sorter<T implements Comparable<T>>` can be used on any class whose objects may be compared to one another.
- The `Comparable<T>` interface should only be used when a class has a natural order.

2 Lists: Unsorted, Sorted, Indexed

Lists: Unsorted, Sorted, Indexed

- A *list* is a linear data structure, where each element except the last has a successor and each element except the first has a predecessor. Every list also has a size, the number of elements in it.
- A list is sorted if its elements are ordered consistent with the `compareTo` method of the elements — each element is "less than or equal to" its successor according to that method.

Lists: Unsorted, Sorted, Indexed

- 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; array-based implementations must resize themselves if necessary.
- 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 added to a list.
- Operations mostly report success or failure by returning a `boolean`, not by throwing an exception on failure.

More Assumptions About Lists

- Sorted lists are in non-decreasing order.
- Indexed lists have indices ranging from 0 to the size - 1, with no gaps.
- The `equals` and `compareTo` methods of the elements are always consistent with one another.

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.
- DJW define a pair of methods to use to iterate through a list: `reset()` and `getNext()`. These are really ugly and I'm going to ignore them from here on. We'll use iterators instead.

The Parent `ListInterface`

```
public interface ListInterface<T> {
  int size();
  void add(T t);
  boolean contains(T t);
  boolean remove(T t); // return false if not found
  T get(T t); // return null if t not there
  String toString();
  void reset(); // set position to beginning
  T getNext(); // advances 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).

The `IndexedListInterface`

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

Clicker Question #2

What `int` value is returned at the end of the following code fragment assuming that `AIL` implements `IndexedListInterface<Dog>` and that all the dogs exist and are distinct.

```
AIL x = new AIL();
x.add(0, cardie);
x.add(0, duncan);
x.add(1, whistle);
x.add(0, whistle);
x.set(2, whistle);
x.remove(0);
x.add(1, cardie);
x.remove(2);
return x.indexOf(whistle);
```

A. 1

B. -1

C. 2

D. 0

The `ArrayUnsortedList` Class

```
public class ArrayUnsortedList<T>
  implements UnsortedListInterface<T> {
  protected T[] list;
  protected int numElements = 0;
  protected int currentPos;// used by reset and getNext
  protected boolean found; // set by find method
  protected int location;  // set by find method

  public ArrayUnsortedList(int origCap) {
    list = (T[])new Object[origCap]; } // warning

  protected void enlarge() {
    T[] larger = (T[])new Object[2*list.length];
    for (int i = 0; i < numElements; i++) {
      larger[i] = list[i]; }
    list = larger; }
```

Methods of AUL

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++;
      }
    }
  }
```

Clicker Question #3

Which of these is not a valid postcondition of the `find` method below, given a nonempty list?

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

A. if `found==true`, then `list[loc]` contains an element equal to `target`

B. if `target` was there, `found==true`

C. if `found==false`, `loc==numElements`

D. if `found==true`, there's only one copy of `target` in the list

AUL `add` and `remove`

```
public void add(T element) {
  if (numElements == list.length)
    enlarge();
  list[numElements] = element;
  numElements += 1;
}

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

More Methods of AUL

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

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

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

Iterating Over an AUL

```
public void reset() {
  currentPos = 0;
}

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

The ArraySortedList Class

```
public class ArraySortedList<T>
  extends ArrayUnsortedList<T>
  implements ListInterface<T> {

  public ArraySortedList(int origCap) {
    super(origCap);
  }
```

Methods of ASL

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

The ASL `add` Method

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

Code for remove

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

The `ArrayIndexedList` Class

```
public class ArrayIndexedList<T>
    extends ArrayUnsortedList<T>
    implements IndexedListInterface<T>
{

  public void add(int index, T element) {
    if ((index < 0) || (index > size()))
      throw new IndexOutOfBoundsException();
    if (numElements == list.length) enlarge();
    for (int i = numElements; i > index; i -= 1)
      list[i] = list [i - 1];
    list[index] = element;
    numElements++;
  }
```

AIL `set` and `get` Methods

```
  public T set(int index, T element) {
    // if index is bad throw exception
    T hold = list[index];
    list[index] = element;
    return hold;
  }

  public T get(int index) {
    // if index is bad throw exception
    return list[index];
  }
```

AIL `indexOf` and `remove` Methods

```
  public int indexOf(T element) {
    find(element);
    if (found) return location;
    else return -1;
  }

  public T remove (int index) {
    // if index is bad throw exception
    T hold = list[index];
    for (int i = index; i < (numElements - 1); i++)
      list[i] = list[i + 1];
    list[numElements - 1] = null;
    numElements -= 1;
    return hold;
  }
```

Applications of Lists

- DJW give three sample applications of their lists, one each for unsorted, sorted, and indexed lists.
- They use the `RankCardDeck` class from Chapter 5 to simulate dealing out lots of seven-card hands for stud poker, empirically deriving the probability that a random hand will contain a pair. (They also compute the probability mathematically, which is a CMPSCI 240 problem.) They keep the hands as unsorted lists.

Applications of Lists

- They use sorted lists to store the scores of golfers — each golfer/score pair is added to the list, and at the end the list can be reported in order of score.
- They use indexed lists to assemble playlists of songs and compute their total length. The user can enter new songs with durations and get a list of the songs with the duration of each and the total time for the playlist.

The Binary Search Algorithm

- The idea of *binary search* in a sorted list is simple — we have a target range, and look at its middle element. If that element is too big or too small we refine the range, and if it is just right we report victory.
- As in the other `find` method, we use the instance variables `found` and `location`, setting the latter to the first answer we find (which may not be the first occurrence of the target). If the search fails we leave found as `false`.

Code for Binary Search

```
protected void find(T target) {
  int first = 0;
  int last = numElements - 1;
  int compareResult;
  Comparable targetElement = (Comparable) target;
  found = false; // recall this is an inst.var.
  while (first <= last) {
    location = (first + last) / 2; // rounds down
    compareResult =
      targetElement.compareTo(list[location]);
    if (compareResult == 0)
      {found = true; break;}
    else if (compareResult < 0)
      last = location - 1;
    else first = location + 1; } }
```

Recursive Binary Search

- This approach is easily made recursive with the use of a helper method that has the appropriate signature for its job "find the target if it is between this location and that".
- We can see that this method has a base case, makes progress toward that base case, and works if the recursive calls work.

Code for Recursive Binary Search

```
protected void recFind(Comparable<T> target,
                       int from, int to) {
  if (from > to)
    {found = false; return;}
  location = (from + to) / 2;
  int cR = target.compareTo(list[location]);
  if (cR == 0) found = true;
  else if (cR < 0)
    recFind(target,from,location-1);
  else recFind(target,location+1,to); }

protected void find(T target) {
  recFind((Comparable<T>)target,
          0, numElements - 1); }
```

Clicker Question #4

```
protected void recFind(Comparable<T> target,
                       int from, int to) {
  if (from > to)
    {found = false; return;}
  location = (from + to) / 2;
  int cR = target.compareTo(list[location]);
  if (cR == 0) found = true;
  else if (cR < 0)
    recFind(target,from,location-1);
  else recFind(target,location+1,to); }
```

Suppose `target` is between the elements in locations 3 and 4 in a six-element list. What is the value of `location` when we finish? (A) 2 (B) 3 (C) 4 (D) 5