CMPSCI 187: Programming With Data Structures

Lecture #22: Indexed Lists and Binary Search

David Mix Barrington

29 October 2012

Indexed Lists and Binary Search

- Review of List Interfaces
- The `ArrayIndexedList` Class
- Applications of Lists
- The Binary Search Algorithm
- Recursive Binary Search
- Analysis of Binary Search
- Linear vs. Binary Search

Review of List Interfaces

- We've seen that DJW have two interfaces for the three kinds of lists: `ListInterface` for the unsorted and sorted lists, and `IndexedListInterface` for indexed lists. `ListInterface` has the methods `size`, `add`, `contains`, `remove`, `get`, `toString`, `reset`, and `getNext`. The last two allow iteration through the list -- in the `java.util List` interface this is done by creating an `Iterator` object associated with the list.

- IndexedListInterface has variants of `add`, `remove`, and `get` that each take an index as an argument -- `add` and `remove` change the indices of elements after the target. It also has two new methods set, to return and change the value at a given index, and indexOf, to find the first occurrence of a particular value.

- We saw array implementations of unsorted and sorted lists. The former had O(1) running time for add and remove, but the latter took O(n) for these. Both were O(1) for size and the iterator methods, and O(n) for the others because of the searches involved. We'll look at faster searching of a sorted list today.

The ArrayIndexedList Class

- The code for this class is not unlike that for the Kennel class on the first midterm, except that we insist that the used slots are consolidated:

```
public class ArrayIndexedList<T> extends ArrayUnsortedList<T>
      implements IndexedListInterface<T> {
   // constructors with super
   public void add(int index, T element) {
      if ((index < 0) || (index > size( )) //throw IOOBException
      if (numElements == list.length) enlarge( );
      for (int i = numElements; i > index; i--)
         list[i] = list [i - 1];
      list[index] = element;
      numElements++;}

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

The Rest of `ArrayIndexedList`

- For some reason DJW repeat `toString` rather than inheriting it. These methods are implemented much like those of `ArraySortedList`.

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

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--;
   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.

- 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 it is too big or too small we refine the range, and if it is just right we report victory.

- Like 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`.

```
protected void find (T target) {
   int first = 0, last = numElements - 1, compareResult;
   Comparable targetElement = (Comparable) target;
   found = false; // recall this is an i.v.
   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.

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

protected void find (T target) {
   Comparable targetElement = (Comparable) target;
   found = false;
   recFind (targetElement, 0, numElements - 1);}
```

Analysis of Binary Search

- A round of binary search (whether recursive or not) either succeeds in finding the target or cuts the range to be searched in half. To search a range of N elements, therefore, costs about log n rounds in the worst case. (Remember that in computer science, **logs** are normally **base-2** and often thought of as integers -- "log x" is the smallest integer k such that $x <= 2^k$.) Each round takes at most an amount of time independent of N, so our running time is O(1) times log n or O(log n).

- The log of 1000 as defined above is 10, since $2^{10} = 1024 >= 1000$. The log of 1,000,000 is 20, therefore, and the log of 10^{15} is about 50. Modern computer operations are measured in nanoseconds, and 10^{15} nanoseconds is about two weeks. So even if N is a huge but realistic number, log N is a small one.

- **Linear search**, by contrast (the original find method, for example), takes O(N) time on a list of size N in the worst case.

Linear vs. Binary Search

- We've seen that the time for binary search is O(log n) -- it grows proportionally to the logarithm of n rather than to n itself. The larger the list, then, the greater the advantage of binary over linear search.

- The big-O hides a larger constant, though, because a step of binary search takes longer than a single comparison. For smaller lists (DJW suggest size less than 20), the simpler linear search may be faster.

- Binary search also only works on sorted lists. If our data does not come to us in sorted form, we have to spend the extra time to **sort** it. (We will look at sorting algorithms later in the course. We also need **random access** to the list to implement binary search. If is list is so long that it must be stored in a **file**, we don't have random access but only sequential access. The same is true if we may only traverse the list with an iterator.