CMPSCI 187: Programming With Data Structures

Lecture #23: Linked Lists
David Mix Barrington
31 October 2012

Linked Lists

- Implementing Lists With References
- The `find` Method
- The `RefUnsortedList` Class
- Methods of `RefUnsortedList`
- The `RefSortedList` Class
- A Variation of Generics
- Methods of `RefSortedList`

Implementing Lists With References

- We’ve seen array-based and link-based implementations of StringLogs, stacks, and queues. Lists can also be implemented with arrays or with **references** (the word DJW are using for links here). Unlike the other classes, lists may require us to add or remove elements from the middle of the linked structure, which is more complicated.

- A reference-based list may be either unsorted or sorted, but we don’t bother with *indexed* reference-based lists. This is because there is really no good way to get at the i’th element of a linked structure, for an arbitrary index i. We might manage this with additional long-range links beyond the basic ones, but we don’t need to consider this now.

- Remember that our list interfaces are the same for either an array-based or reference-based implementation, and that both unsorted and sorted lists implement ListInterface because the forms of the methods are all the same.

The `RefUnsortedList` Class

- As before we use `LLNode<T>` objects to hold our elements. There is no sentinel node, so an empty list has `list == null`.

- As with the array-based lists, we will have an auxiliary find method that will set the boolean `found` true if it finds its target, and set `position` where it is. But this time `position` is a node, as is the node `previous` that comes immediately before `position`.

```
public class RefUnsortedList<T> implements ListInterface<T> {
   protected int numElements;
   protected LLNode<T> currentPos; // for iterator
   protected boolean found; //set by find
   protected LLNode<T> location; // set by find
   protected LLNode<T> previous; // set by find
   protected LLNode<T> list; // head of list

   public RefUnsortedList( ) {
      numElements= 0;
      list = currentPos = null;}  // no sentinel
```

The `find` Method

- Just as in the array-based implementation, the `find` method conducts a linear search of the list, reporting the results of the search through side effects on the instance variables `found` and `position`. If it finds the target, `found` is set `true` and `position` and `previous` point to where it is and just before that. If it is not found, `position` is `null` and `found` is false -- `previous` is still set to the last node but this shouldn't cause a problem.

```
protected void find (T target) {
   location = list;
   found = false;
   while (location != null) {
      if (location.getInfo( ).equals(target) {
         found = true;
         return;}
      else {previous = location;
         location = location.getLink;}}}
```

Other Methods of `RefUnsortedList`

- Many of these methods are almost identical to the array-based ones.

```
public void add (T element) {
   LLNode<T> newNode = new LLNode<T>(element);
   newNode.setLink(list);
   list = newNode;
   numElements++;}

public int size( ) {return numElements;}

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

public boolean remove (T element) {
   find(element);
   if (found) {
      if (list == location) list = list.getLink;
      else previous.setLink(location.getLink( ));
      numElements--;}
   return found;}
```

Still More Methods of `RefUnsortedList`

```
public T get (T element) {
   find(element);
   if (found) return location.getInfo( );
   else return null;}

public String toString( ) {
   LLNode<T> currNode = list;
   String listString = “List:\n”);
   while (currNode != null) {
      listString += “  “ + currNode.getInfor( ) + “\n”;
      currNode = currNode.getLink( );}
   return listString;}

public void reset( ) {currentPos = list;}

public T getNext( ) {
   T next = currentPos.getInfo( );
   if (currentPos.getLink == null) currentPos = list;
   else currentPos = currentPos.getLink( );
   return next;}
```

The `RefSortedList` Class

- If we have our sorted list class extend the unsorted list class, what do we need to change? Actually not very much. The `contains`, `remove`, and `get` methods all depend on the `find` method for their behavior. What they do after the element is found does not change at all.

- And for that matter, the `find` method of `RefUnsortedList` works perfectly well on sorted lists. The one thing we could change is that in an unsuccessful search, we could give up once we find an element in the list greater than the target, since we couldn't have the target come after that point. This is no savings at all in the worst case (the method is still O(n)) but will save us some time in some cases.

- The `add` method, however, definitely needs to be overridden.

```
public class RefSortedList<T extends Comparable<T>>
      extends RefUnsortedList<T> implements ListInterface<T> {
   public RefSortedList( ) {super( );}
```

A Variation of Generics

- Before we look at add, though, we should notice something unusual in the first line of this class' header. `RefSortedList` is a generic class, but we can only define `RefSortedList<T>` in the cases where `T` implements the interface `Comparable<T>`, meaning that it has a `compareTo` method that takes a parameter of type `T`.

- We can add an `extends` clause to the type variable in the declaration of a generic class, which makes it only valid when the promise of that clause is fulfilled. We say extends rather than implements because `T` could be anywhere below `Comparable<T>` in the inheritance hierarchy, not just directly under it.

- In `ArraySortedList` we didn't bother with this because of the constructor business -- we relied on a cast to check that `T` implemented `Comparable<T>`.

```
public class RefSortedList<T extends Comparable<T>>
```

Methods of `RefSortedList`

- To add to a sorted list, we skip past any and all elements that are less than the new element, until we reach the end or find the right element to put the new element in front of. Inserting a new first node is a special case.

```
public void add (T element) {
   LLNode<T> prevLoc, location; // location is local, not instance
   T listElement;
   location = list; prevLoc = null;
   while (location != null) {
      listElement = location.getInfo( );
      if (listElement.compareTo(element) < 0) {
         prevLoc = location; location = location.getLink( );}
      else break;}
   LLNode<T> newNode = new LLNode<T>(element);
   if (prevLoc == null) {newNode.setLink(list); list = newNode;}
   else {newNode.setLink(location);
      prevLoc.setLink(newNode);}
   numElements++;}
```