CmpSci 187: Programming with Data Structures
Spring 2015

Lecture #15, Lists: Linked Implementation

John Ridgway

March 26, 2015

1 Reference-Based Lists

`ListInterface` Revisited

```
import java.util.Iterator;

public interface ListInterface<T>
    extends Iterable<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();
  Iterator<T> iterator();
}
```

Making Lists With References

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

Making Lists With References

- 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.

Clicker Question #1

What exactly do we mean by "we won't bother with reference-based indexed lists"?

A. We could build such a class, but the indexed list operations would take $O(n)$ time, not $O(1)$.

B. It is not possible to implement an indexed list with references.

C. A reference-based indexed list would just be the same thing as a deque.

D. We don't need any more code to do it, because we can just implement the list interface.

Making Lists With References

- Remember that our list interfaces are the same for either an array-based or reference-based implementation.
- Both unsorted and sorted lists implement `ListInterface` because the forms of the methods are all the same.
- And remember the assumptions for lists we gave in the last lecture: duplicate elements allowed, no null entries, `boolean` return from methods, consistent `equals` and `compareTo`, etc.

Making Lists with References (continued)

- We'll create `RefUnsortedList` and `RefSortedList`.
- There's a lot of commonality between them, but neither can really extend the other.
- We'll create an `AbstractRefList` to hold the commonality, and the other two will extend it.
- Let's look at it first.

`AbstractRefList` Beginning

```
public abstract class AbstractRefList<T>
    implements ListInterface<T>
{
  protected int numElements;
  protected Node<T> head;

  public AbstractRefList() {
    numElements = 0;
    head = null;
```

```
}

public abstract void add(T element);
```

AbstractRefList Methods

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

// Returns true if this list contains an element e such that
public boolean contains (T element) {
  for (T nodeData : this) {
    if (nodeData.equals(element)) {
      return true;
    }
  }
  return false;
}
```

AbstractRefList remove Method

```
public boolean remove (T element) {
  if (head == null) { return false; }
  Node<T> prevNode = null;
  for (Node<T> node = head; node != null;
       node = node.getNext()) {
    if (node.getData().equals(element)) {
      if (prevNode==null) { head=head.getNext(); }
      else { prevNode.setNext(node.getNext()); }
      numElements -= 1;
      return true;
    }
    prevNode = node;
  }
  return false;
}
```

More AbstractRefList Methods

```
public T get(T element) {
  for (T data : this) {
    if (data.equals(element)) {
      return data;
    }
  }
  return null;
}

public Iterator<T> iterator() {
  return new LinkedNodeIterator<T>(head);
}
```

The `LinkedNodeIterator` Class

```
class LinkedNodeIterator<T>
    implements Iterator<T> {
  private Node<T> next;
  LinkedNodeIterator(Node<T> first) {
    next = first; }
  public boolean hasNext() { return next!=null; }
  public T next() {
    T t = next.getData();
    next = next.getNext();
    return t; }
  public void remove() {
    throw new UnsupportedOperationException(); }
}
```

The `RefUnsortedList` Class

```
import java.util.Iterator;

public class RefUnsortedList<T>
    extends AbstractRefList<T>
{
  public void add(T element) {
    head = new Node<T>(element, head);
    numElements += 1;
  }
}
```

Clicker Question #2

Which of these code blocks removes any duplicate copies of element `x`, leaving only one (or none if there were none to start)?

A. `while (contains(x)) remove(x); add(x);`

B. `T h; while (contains(x)) { h = get(x); remove (x); } if (h != null) add(h);`

C. `Node<T> h; while (contains(x)) h = remove(x); add(h);`

D. `while (true) { remove(x); if (!contains(x)) { add(new Node<T>(x)); break;}`
`}`

The RefSortedList Class

```
class RefSortedList<T extends Comparable<T>>
  extends AbstractRefList<T>
{
  public void add(T t) {
    Node<T> prevNode = null, node = head;
    while (node != null) {
      if (node.getData().compareTo(t)>=0)
        break;
      prevNode = node;
      node = node.getNext();
```

```
    }
    if (prevNode == null)
      head = new Node<T>(t, head);
    else
      prevNode.setNext(new Node<T>(t, node));
    numElements += 1;
  } }
```

A Variation on Generics

- `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.

2 More Lists

Doubly-Linked Lists

- The asymmetry between the forward and backward directions in a singly-linked list sometimes poses a problem.
- We can't easily remove a node from the tail of the list, for example. If we ever wanted to traverse a singly-linked list backward, we'd be in big trouble.
- A natural way to remove the asymmetry is to give each node two pointers, one forward and one backward, as shown `DNode` soon.
- Unlike DJW's version of `DLLNode`, which extends `LLNode`, our `DNode` does not extend `Node`.

The `DNode` Class

```
public class DNode<T> {
  private T data;
  private DNode<T> next, prev;
  public DNode(T data,DNode<T> next,DNode<T> prev) {
    this.data = data;
    this.next = next;
    this.prev = prev; }
  public T getData() { return data; }
  public DNode<T> getNext() { return next; }
  public DNode<T> getPrev() { return prev; }
  public void setData(T data) { this.data = data; }
  public void setNext(DNode<T> next) {
    this.next = next; }
  public void setPrev(DNode<T> prev) {
    this.prev = prev; } }
```

Doubly Linked Lists

- This doubly linked list solves the problems above, at the cost of (1) doubling the amount of memory devoted to links and (2) roughly doubling the number of pointers that have to be changed in adding or removing a node.

`DoublyLinkedList addAfter` Method

```
private void addAfter(T t, DNode<T> prev) {
  DNode<T> next;
  if (prev == null) { next = head; }
  else { next = prev.getNext(); }
  DNode<T> node = new DNode<T>(t, next, prev);
  if (prev == null) { head = node; }
  else { prev.setNext(node); }
  if (next == null) { tail = node; }
  else { next.setPrevious(prev); }
}
```

`DoublyLinkedList removeNode` Method

```
private void removeNode(DNode<T> node) {
  DNode<T> prev = node.getPrevious();
  DNode<T> next = node.getNext();
  if (prev == null) { head = next; }
  else { prev.setNext(next); }
  if (next == null) { tail = prev; }
  else { next.setPrevious(prev);
  }
}
```

Clicker Question #3

If we know that `node.getPrevious()` is `null`, which of these code fragments removes the node pointed to by `node` from the doubly-linked list headed by `head`?

A. `node.getNext( ).setPrevious(null);`

B. `node.getPrevious().setNext(null);`

C. `node.setNext(node.getNext( ).getNext( )); node.getNext().setPrevious(node);`

D. `head = head.getNext(); head.setPrevious(null);`