Solutions to Exercises in Chapter 8

8.1 Some examples of lists in everyday life would include the following:

(i) A travel itinerary consisting of a list of journeys, e.g., flights, that are to be taken in the appropriate order.

(ii) A 'to-do' list of tasks to be completed, sorted in the order the tasks are to be performed.

8.2 (a) The `Stack` ADT is a special case of the `List` ADT since all of the operations required by the `Stack` ADT can be implemented using the operations of the `List` ADT (but not vice versa). In particular, the operations performed on a stack *s* can be rewritten using operations performed on a corresponding list *l* as shown in Table S8.1.

(b) The `Queue` ADT is a special case of the `List` ADT since all of the operations required by the `Queue` ADT can be implemented using the operations of the `List` ADT (but not vice versa). In particular, the operations performed on a queue *q* can be rewritten using operations performed on a corresponding list *l* as shown in Table S8.2.

Table S8.1 Implementation of the `Stack` ADT using the `List` ADT.

Stack operation	Corresponding list operation(s)
s`.addLast(`*x*`)`	*l*`.add(`*x*`)`
x `=` *s*`.removeLast()`	*x* `=` *l*`.remove(`*l*`.size() - 1)`
x `=` *s*`.getLast()`	*x* `=` *l*`.get(`*l*`.size() - 1)`

Table S8.2 Implementation of the `Queue` ADT using the `List` ADT.

Queue operation	Corresponding list operation
q`.addLast(`*x*`)`	*l*`.add(`*x*`)`
x `=` *q*`.removeFirst()`	*x* `=` *l*`.remove(0)`
x `=` *q*`.getFirst()`	*x* `=` *l*`.get(0)`

8.3 Using the `List` ADT of Program 8.2, a possible version of the `reorder` method is as follows:

```
static List reorder (List persons) {
// Assume that persons is a list of Person objects, ordered by name.
// Return a similar list of Person objects, ordered such that all
// children (aged under 18) come before all adults (aged 18 or over), but
// otherwise preserving the ordering by name.
   List children = new LinkedList();
   List adults = new LinkedList();
   Iterator iter = persons.iterator();
```

```
    while (iter.hasNext()) {
      Person p = (Person) iter.next();
      if (p.age <= 18)
        children.add(p);
      else
        adults.add(p);
    }

    // Construct the result with children before adults.
    List result = children;
    result.addAll(adults);
    return result;
  }
```

8.4 A possible solution for using an iterator to extend the simple text editor given in Program 8.1 with the methods `findFirst(`*s*`)` and `findNext()` would be as follows:

```
private Iterator textIterator;
private String searchString;
private int lineFound;

public void findFirst (String s) {
// Find the first line containing an occurrence of the given string, and
// make this the currently selected line.
  textIterator = text.iterator();
  searchString = s;
  lineFound = -1;
  findNext();
}

public void findNext () {
// Find the next line containing an occurrence of the string specified by
// findFirst, and make this the currently selected line.
  while (textIterator.hasNext()) {
    String s = (String) textIterator.next();
    lineFound++;
    if (s.indexOf(searchString) >= 0) {
      sel = lineFound;
      return;
    }
  }
}
```

8.5 Add the following operation to the `List` interface of Program 8.2:

```
public ListIterator listIterator ();
// Return a bi-directional iterator over this list.
```

Modify the `ArrayList` class of Program 8.9 as shown in Program S8.3. All operations have $O(1)$ time complexity.

Modify the `LinkedList` class of Program 8.14 as shown in Program S8.4. All operations have $O(1)$ time complexity, except `previous` which is $O(n)$.

8.6 Add the following operations to the `List` interface of Program 8.2:

```
public boolean contains (Object target);
// Return true if and only if this list contains an element equal to
// target.

public int indexOf (Object target);
// Return the index of the first element in this list that is equal to
// target, or –1 if there is no such element.
```

Implement these operations as follows in the `ArrayList` class of Program 8.9:

```
public boolean contains (Object target) {
  return (indexOf(target) >= 0);
}

public int indexOf (Object target) {
  for (int i = 0; i < length; i++) {
    Object elem = elems[i];
    if ((target == null && elem == null) ||
         (target != null && target.equals(elem)))
      return i;
  }
  return -1;
}
```

Implement these operations as follows in the `LinkedList` class of Program 8.14:

```
public boolean contains (Object target) {
  return (indexOf(target) >= 0);
}

public int indexOf (Object target) {
  SLLNode curr = first;
  while (curr != null) {
    Object elem = curr.element;
    if ((target == null && elem == null) ||
         (target != null && target.equals(elem)))
      return i;
    curr = curr.succ;
  }
  return -1;
}
```

(*Note:* These implementations allow for the possibility that `target` is null.)

8.7 Add the following operation to the `List` interface of Program 8.2:

```
public List subList (int i, int j);
// Return a new list containing all of the elements in this list with
// indices i through j–1.
```

Implement this operation as follows in the `ArrayList` class of Program 8.9:

```
public List subList (int i, int j) {
  if (i < 0 || j > length || i > j)
    throw new IndexOutOfBoundsException();
  List result = new ArrayList(j-i+1);
  for (int p = i; p < j; p++)
    result.add(elems[p]);
  return result;
}
```

Implement this operation as follows in the `LinkedList` class of Program 8.14:

```
public List subList (int i, int j) {
  if (i < 0 || j > length || i > j)
    throw new IndexOutOfBoundsException();
  List result = new LinkedList();
  SLLNode curr = get(i);
  for (int p = i; p < j; p++) {
    result.add(curr.element);
    curr = curr.succ;
  }
  return result;
}
```

8.10 Implement the auxiliary method `expand` in Program 8.9 as follows:

```
private void expand () {
// Make the elems array longer.
  Object[] newElems = new Object[elems.length*2];
  for (int i = 0; i < length; i++)
    newElems[i] = elems[i];
  elems = newElems;
}
```

8.11 The advantage of using instance variable `length` in the linked-list implementation of lists is that the size operation is $O(1)$ rather than $O(n)$. The disadvantage is that it must be updated by most of the transformer operations.

8.12 Add the following operations to the `List` interface of Program 8.2:

```
public Object getFirst ();
// Return the first element in this list, or throw a
// NoSuchElementException if this list is empty.

public Object getLast ();
// Return the last element in this list, or throw a
// NoSuchElementException if this list is empty.

public Object addFirst (Object elem);
// Add elem before the first element in this list.

public Object removeFirst ();
// Remove the first element in this list, or throw a
// NoSuchElementException if this list is empty.

public Object removeLast ();
// Remove the last element in this list, or throw a
// NoSuchElementException if this list is empty.
```

(*Note:* `addLast` would be just a synonym for the existing `add(Object)` operation.)

Implement these operations as follows in the `ArrayList` class of Program 8.9:

```
public Object getFirst () {
  if (length == 0)
    throw new NoSuchElementException();
  return elems[0];
}

public Object getLast () {
  if (length == 0)
    throw new NoSuchElementException();
  return elems[length-1];
}
```

```
public Object addFirst (Object elem) {
  add(0, elem);
}

public Object removeFirst () {
  if (length == 0)
    throw new NoSuchElementException();
  return remove(0);
}

public Object removeLast () {
  if (length == 0)
    throw new NoSuchElementException();
  return remove(length-1);
}
```

Implement these operations as follows in the `LinkedList` class of Program 8.14:

```
public Object getFirst () {
  if (length == 0)
    throw new NoSuchElementException();
  return first.element;
}

public Object getLast () {
  if (length == 0)
    throw new NoSuchElementException();
  return last.element;
}

public Object addFirst (Object elem) {
  add(0, elem);
}

public Object removeFirst () {
  if (length == 0)
    throw new NoSuchElementException();
  return remove(0);
}

public Object removeLast () {
  if (length == 0)
    throw new NoSuchElementException();
  return remove(length-1);
}
```

8.13 The `java.util.LinkedList` class uses a DLL representation because the `remove` operation has $O(1)$ time complexity, whereas an SLL representation would result in $O(n)$.

```
public class ArrayList implements List {

  …

  public ListIterator listIterator () {
    return new ArrayList.TwoWayIterator();
  }

  /////////////  Inner class  /////////////

  private class TwoWayIterator
                implements ListIterator {

  //  An ArrayList.TwoWayIterator object is a bi-directional
  //  iterator over a list represented by an ArrayList object.

    private int place;

    private TwoWayIterator () {
      place = 0;
    }

    public boolean hasNext () {
      return (place < length);
    }

    public Object next () {
      if (place >= length)
        throw new NoSuchElementException();
      return elems[place++];
    }

    public boolean hasPrevious () {
      return (place > 0);
    }

    public Object previous () {
      if (place <= 0)
        throw new NoSuchElementException();
      return elems[--place];
    }

    public int nextIndex () {
      return (place < length ? place : -1);
    }

    public int previousIndex () {
      return (place - 1);
    }

    public void remove () {
      throw new UnsupportedOperationException();
    }

    public void set (Object obj) {
      throw new UnsupportedOperationException();
    }

    public void add (Object obj) {
      throw new UnsupportedOperationException();
    }

  }
}
```

Program S5.3 `ArrayList` class modified to provide a bi-directional iterator.

```
public class LinkedList implements List {

  …

  public ListIterator listIterator () {
    return new LinkedList.TwoWayIterator();
  }

  //////////// Inner class ////////////

  private class TwoWayIterator
                implements ListIterator {

  // An LinkedList.TwoWayIterator object is a bi-directional
  // iterator over a list represented by an LinkedList object.

    private SLLNode curr, prev;
    private int place;

    private TwoWayIterator () {
      curr = first;  prev = null;  place = 0;
    }

    public boolean hasNext () {
      return (curr != null);
    }

    public Object next () {
      if (curr == null)
        throw new NoSuchElementException();
      Object nextElem = curr.element;
      prev = curr;  curr = curr.succ;  place++;
      return nextElem;
    }

    public boolean hasPrevious () {
      return (prev != null);
    }

    public Object previous () {
      if (prev == null)
        throw new NoSuchElementException();
      Object prevElem = curr.element;
      curr = prev;
      if (curr == first)
        prev = null;
      else {
        prev = first;
        while (prev.succ != curr)
          prev = prev.succ;
      }
      place--;
      return prevElem ;
    }

    public int nextIndex () {
      return (place < length ? place : -1);
    }

    public int previousIndex () {
      return (place - 1);
    }
```

Program S5.4 `LinkedList` class modified to provide a bi-directional iterator

(continued on next page).

```
        public void remove () {
          throw new UnsupportedOperationException();
        }
        public void set (Object obj) {
          throw new UnsupportedOperationException();
        }
        public void add (Object obj) {
          throw new UnsupportedOperationException();
        }
      }
    }
```

Program S5.4 `LinkedList` class modified to provide a bi-directional iterator *(continued)*.