Solutions to Exercises 5

5A In principle, the solution to the 'millenium bug' was simply to change the date representation to allow *four* digits for the year number.

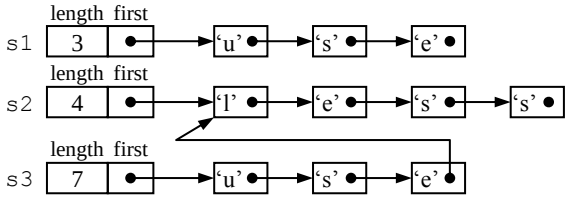
In practice, the problem was enormous. In thousands of programs totaling billions of lines of code, it was necessary to locate and modify every declaration defining the date representation, and every statement whose effect depended on the date representation. Thousands of files and databases containing date values had to be translated to the new date representation.

The problem was compounded by the fact that many of the programs were poorly structured and undocumented. If ADTs had been used, the change to each program would have been localized in a single place.

5C The following table shows the time complexity of each operation (where n, n_1, and n_2 are the string lengths):

Operation	Time complexity (array representation)	Time complexity (SLL representation)
`length`	$O(1)$	$O(1)$
`charAt`	$O(1)$	$O(n)$
`equals`	$O(\min(n_1, n_2))$	$O(\min(n_1, n_2))$
`compareTo`	$O(\min(n_1, n_2))$	$O(\min(n_1, n_2))$
`substring`	$O(n_2)$	$O(n_1)$
`concat`	$O(n_1 + n_2)$	$O(n_1)$

The diagram below shows the effect of `s3 = s1.concat(s2)` when `String` objects are represented by SLLs. The operation makes a copy of `s1`'s SLL, then links the resulting SLL's last node to `s2`'s first node. Thus the resulting SLL shares nodes with `s2`'s SLL, rather than containing copies of these nodes. The operation makes only n_1 copies, so its time requirement does not depend on n_2. (*Note:* This implementation of `concat` is possible only because `String` objects are immutable.)



5D Here is a possible contract for a `Text` ADT:

```
public class Text {

   // Each Text value is a sequence of characters subdivided into lines. The
   // characters and line-terminators have consecutive position numbers, starting at 0.

   private ...;

   ////////////// Constructors ////////////

   public Text ();
   // Construct an empty text.

   ////////////// Accessors ////////////

   public void display ();
   // Display this text on the screen.

   public void save (String filename);
   // Save this text to the named file.

   ////////////// Transformers ////////////

   public void load (String filename);
   // Load the contents of the named file into this text.

   public void insert (int p, char c);
   // Insert c just before the character at position p in this text.

   public void insert (int p, Text that);
   // Insert that just before the character at position p in this text.

   public void delete (int p);
   // Delete the character at position p in this text.

   public void delete (int p1, int p2);
   // Delete all characters at positions p1 through p2 in this text.

   public Text copy (int p1, int p2);
   // Return a copy of all characters at positions p1 through p2 in this text.

}
```

Cut would be achieved by:

```
clipboard = text.copy(start, finish);
text.delete(start, finish);
```

Paste would be achieved by:

```
text.insert(cursor, clipboard);
```

The simplest representation for a text would be a long array or SLL of characters, where each line-terminator is represented by a suitable control character (such as CR or LF).

5E Here is a possible contract for a `TimeOfDay` ADT:

```
public class TimeOfDay {
   // Each TimeOfDay value is a time-of-day, accurate to 1 second.
   private ...;
   ////////////// Constructors ////////////
   public TimeOfDay (int s);
   // Construct a time-of-day that is s seconds since midnight.
   public TimeOfDay (int h, int m, int s);
   // Construct a time-of-day that is h hours, m minutes, and s seconds since midnight.
   ////////////// Accessors ////////////
   public int getHour ();
   // Return the hours part of this time-of-day.
   public int getMinute ();
   // Return the minutes part of this time-of-day.
   public int getSecond ();
   // Return the seconds part of this time-of-day.
   public int minus (TimeOfDay that);
   // Return the number of seconds between this time-of-day and that.
   ////////////// Transformer ////////////
   public TimeOfDay plus (int s);
   // Return this time-of-day advanced by s seconds (or retarded if s < 0).
}
```

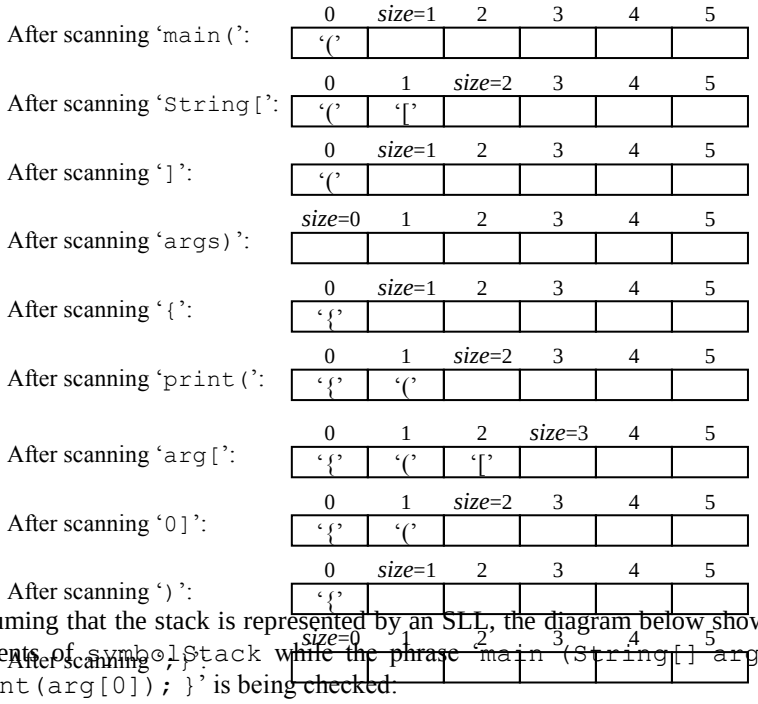
A time-of-day could be represented by a triple (hours, minutes, seconds), or by the number of seconds since midnight. The latter representation would make the `plus` and `minus` operations easier to implement.

5F Here is a possible contract for a `Course` ADT:

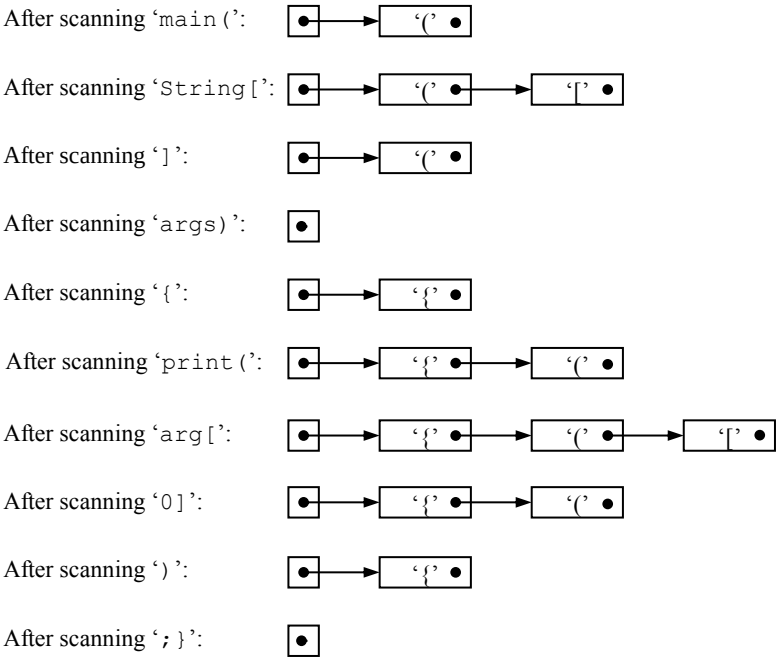
```
public class Course {
    // Each Course value is a course description, consisting of a course-code, title,
    // names one or more instructors, and names of zero or more teaching assistants.
    // All fields can be updated except the course-code.

    private ...;

    /////////////  Constructor  ////////////

    public Course (String code, String title,
                   String[] instructors);
    // Construct a course description with the given details but no teaching assistants.

    /////////////  Accessors  ////////////

    public String getCode ();
    // Return this course's course-code.

    public String getTitle ();
    // Return this course's title.

    public String[] getInstructors ();
    // Return the names of this course's instructors.

    public String[] getAssistants ();
    // Return the names of this course's teaching assistants.

    /////////////  Transformers  ////////////

    public void changeTitle (String newTitle);
    // Change this course's title to newTitle.

    public void addInstructor (String name);
    // Add an instructor with name to this course.

    public void removeInstructor (String name);
    // Remove the instructor with name from this course.

    public void addAssistant (String name);
    // Add a teaching assistant with name to this course.

    public void removeAssistant (String name);
    // Remove the teaching assistant with name from this course.
}
```

Solutions to Exercises 6

6B Assuming that the stack is represented by an array, the diagram below shows the contents of `symbolStack` while the phrase '`main (String[] args) { print(arg[0]); }`' is being checked:



6C Assuming that the stack is represented by an SLL, the diagram below shows the contents of `symbolStack` while the phrase '`main (String[] args) { print(arg[0]); }`' is being checked:



6D Add the following accessor to the homogeneous stack contract:

```
public E get (int d);
// Return the element at depth d in this stack, counting the topmost element
// as having depth 1. Throw an exception if d < 1 or d > stack size.
```

Add the following to the array implementation:

```
public E get (int d) {
   if (d < 1 || d > size) throw …;
   return elems[size-d];
}
```

Add the following to the SLL implementation:

```
public E get (int d) {
   if (d < 1) throw …;
   Node curr = top;
   for (int i = 1; i < d; i++) {
      if (curr == null) throw …;
      curr = curr.succ;
   }
   return curr.element;
}
```

6E To make the array implementation deal with an overflow by throwing an exception:

```
public void push (E it) throws StackException {
// Add it as the top element on this stack.
// Throw a StackException if there is no room.
   if (size == elems.length)
      throw new StackException();
   elems[size++] = it;
}
```

This assumes that `StackException` is a subclass of `Exception`.

To make the array implementation deal with an overflow by substituting a longer array:

```
public void push (E it) throws StackException {
// Add it as the top element on this stack.
// Throw a StackException if there is no room.
   if (size == elems.length) {
      E[] newElems = (E[]) new Object [2*size];
      System.arraycopy(elems, 0, newElems, 0, size);
      elems = newElems;
   }
   elems[size++] = it;
}
```

Solutions to Exercises 7

7A Make the `add` method call the following auxiliary method to expand the cyclic array:

```
private void expand () {
// Make the elems array twice as long.
   E[] newElems =
      (E[]) new Object[2*elems.length];
   int j = front;
   for (int i = 0; i < size; i++) {
      newElems[i] = elems[j++];
      if (j == elems.length)  j = 0;
   }
   elems = newElems;
   front = 0;  rear = size;
}
```

7B We could drop instance variable `rear`, since its value can be computed from `front` and `size` whenever required. Modify the `addLast` operation as follows:

```
public void addLast (E it) {
// Add it as the rear element of this queue.
   if (size == elems.length)  expand();
   int rear = (front + size) % elems.length;
   elems[rear] = it;
   size++;
}
```

Also, remove all occurrences of `rear` in other operations.

(*Note:* We could similarly drop instance variable `front`. But we cannot drop instance variable `size`, since it would be impossible to tell whether the queue is empty or full when `front` and `rear` are equal.)

7C It would be pointless to implement the queue ADT using a DLL, since none of the operations needs to access any node's predecessor.

7D A UNIX pipe connecting process P_1 to process P_2 can be implemented by a queue of bytes, q. Initially q is empty. Whenever P_1 writes a byte to the pipe, that byte is added to the rear of q. Whenever P_2 reads a byte from the pipe, that byte is removed from the rear of q.

(*Note:* Since processes P_1 and P_2 are concurrent, we must *synchronize* the queue operations, i.e., ensure that only one operation is called at a time. If the `addLast` and `removeFirst` operations were called at the same time, the instance variables representing the queue would be left in an unpredictable state.)

7E The keyboard driver can communicate with the application program via a queue whose elements are characters. The driver adds characters to the rear of the queue, and the application removes them from the front.

(*Note:* As in Exercise 7D, we must synchronize the queue operations.)

To handle a keyboard (ignoring all control characters):

1. Make character queue *q* empty.
2. Repeat indefinitely:
   2.1. Accept a character *char* from the keyboard.
   2.2. If *char* is a graphic character:
      2.2.1. Echo *char* to the screen.
      2.2.2. Add *char* to the rear of *q*.
   2.3. Else, if *char* is a control character:
      2.3.1. Do nothing.

To handle a keyboard (ignoring all control characters other than DELETE):

1. Make character queue *q* empty.
2. Repeat indefinitely:
   2.1. Accept a character *char* from the keyboard.
   2.2. If *char* is a graphic character:
      2.2.1. Echo *char* to the screen.
      2.2.2. Add *char* to the rear of *q*.
   2.3. Else, if *char* is DELETE:
      2.3.1. Backspace the screen cursor, blanking out the character there.
      2.3.2. Remove the rearmost character of *q*.
   2.4. Else, if *char* is a control character other than DELETE:
      2.4.1. Do nothing.

(*Note:* Step 2.3.2 removes the *rearmost* (last) element of the queue. But that is not an operation of the standard queue ADT, so we must instead use a special kind of queue, namely the *double-ended queue* of Exercise 7F.)

7F A contract for a deque ADT is shown below.

A DLL implementation of deques is also outlined below. With this implementation, all deque operations have time complexity $O(1)$.

```
public interface Deque<E> {
   // Each Deque object is a homogeneous double-ended queue (dequeue)
   // whose elements are of type E.
   //////////// Accessors ////////////
   public boolean isEmpty ();
   // Return true if and only if this deque is empty.
   public int size ();
   // Return this deque's size.
   public E getFirst ();
   // Return the element at the front of this deque. Throw an exception
   // if this deque is empty.
   public E getLast ();
   // Return the element at the rear of this deque. Throw an exception
   // if this deque is empty.
   //////////// Transformers ////////////
   public void clear ();
   // Make this deque empty.
   public void addFirst (E it);
   // Add it as the front element of this deque.
   public void addLast (E it);
   // Add it as the rear element of this deque.
   public E removeFirst ();
   // Remove and return the front element from this deque. Throw an
   // exception if this deque is empty.
   public E removeLast ();
   // Remove and return the rear element from this deque. Throw an
   // exception if this deque is empty.
}

public class LinkedDeque implements Deque {
   // Each LinkedDeque object is a deque (double-ended queue) whose
   // elements are objects.
   // This deque is represented as follows: its size is held in size;
   // front and rear are links to the first and last nodes of a DLL
   // containing its elements.
   private Node front, rear;
   private int size;
   //////////// Inner class ////////////
   …
   private class Node {
      public E element;
      public Node pred, succ;
      …
   }
   //////////// Constructor ////////////
   public LinkedDeque () {
   // Construct a deque, initially empty.
      front = rear = null;
      size = 0;
   }
   //////////// Accessors ////////////
```

```
  …
  public E getLast () {
  // Return the element at the rear of this deque. Throw an exception
  // if this deque is empty.
    if (rear == null) throw …;
    return rear.element;
  }
  /////////////  Transformers  /////////////
  …
  public void addFirst (E it) {
  // Add it as the front element of this deque.
    Node newest = new Node(it, null, null);
    if (front != null)
      front.pred = newest;
    else
      rear = newest;
    front = newest;
    size++;
  }
  public E removeLast () {
  // Remove and return the rear element of this deque. Throw an exception
  // if this deque is empty.
    if (rear == null) throw …;
    Object rearElem = rear.element;
    rear = rear.pred;
    if (rear == null)  front = null;
    size--;
    return rearElem;
  }
}
```

Solutions to Exercises 8

8A (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 follows:

Stack operation	Corresponding list operation(s)
s`.push(`*x*`)`	*l*`.add(`*x*`)`
x `=` *s*`.pop()`	*x* `=` *l*`.remove(`*l*`.size() - 1)`
x `=` *s*`.top()`	*x* `=` *l*`.get(`*l*`.size() - 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 follows:

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)`

8B Using the `List` ADT, a possible version of the `reorder` method is as follows:

```
static List<Person> reorder (
                List<Person> 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<Person> result =
        new LinkedList<Person>();
   List<Person> adults =
        new LinkedList<Person>();
   Iterator<Person> persons = persons.iterator();
   while (persons.hasNext()) {
      Person p = persons.next();
      if (p.age <= 18)
         result.add(p);
      else
         adults.add(p);
   }
   // Construct the result with children before adults.
   result.addAll(adults);
   return result;
}
```

8C A possible solution for using an iterator to extend the simple text editor with the methods `findFirst(s)` and `findNext()` would be as follows:

```
private Iterator<String> lines;
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.
   lines = 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 (lines.hasNext()) {
      String s = lines.next();
      lineFound++;
      if (s.indexOf(searchString) >= 0) {
         sel = lineFound;
         return;
      }
   }
}
```

8D Add the following operations to the `List<E>` interface:

```
public boolean contains (E target);
// Return true if and only if this list contains an element equal to
// target.
public int indexOf (E 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<E>` class:

```
public boolean contains (E target) {
  return (indexOf(target) >= 0);
}
public int indexOf (E target) {
  for (int i = 0; i < size; i++) {
    E 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<E>` class:

```
public boolean contains (E target) {
  return (indexOf(target) >= 0);
}
public int indexOf (E target) {
  Node curr = first;
  while (curr != null) {
    E 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.)

8E Add the following operation to the `List<E>` interface:

```
public List<E> 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<E>` class:

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

Implement this operation as follows in the `LinkedList<E>` class:

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

8F Implement an auxiliary method `expand` as follows:

```
private void expand () {
// Make the elems array twice as long.
   E[] newElems =
      (E[]) new Object[2*elems.length];
   System.arraycopy(elems, 0, newElems, 0, size);
   elems = newElems;
}
```

8H Add the following operations to the `List<E>` interface:

```
public E getFirst ();
// Return the first element in this list, or throw an exception
// if this list is empty.
public E getLast ();
// Return the last element in this list, or throw an exception
// if this list is empty.
public void addFirst (E it);
// Add it before the first element in this list.
public E removeFirst ();
// Remove the first element in this list, or throw an exception
// if this list is empty.
public E removeLast ();
// Remove the last element in this list, or throw an exception
// if this list is empty.
```

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

Implement these operations as follows in the `ArrayList<E>` class:

```
public E getFirst () {
   if (size == 0) throw …;
   return elems[0];
}
```

```
public E getLast () {
  if (size == 0) throw …;
  return elems[size-1];
}
public void addFirst (E it) {
  add(0, it);
}
public E removeFirst () {
  if (size == 0) throw …;
  return remove(0);
}
public E removeLast () {
  if (size == 0) throw …;
  return remove(size-1);
}
```

Implement these operations as follows in the `LinkedList<E>` class:

```
public E getFirst () {
  if (size == 0) throw …;
  return first.element;
}
public E getLast () {
  if (size == 0) throw …;
  return last.element;
}
public E addFirst (E it) {
  add(0, it);
}
public E removeFirst () {
  if (size == 0) throw …;
  return remove(0);
}
public E removeLast () {
  if (size == 0) throw …;
  return remove(size-1);
}
```

8J 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 $O(n)$.