Solutions to Exercises in Chapter 6

6.3 Assuming that the stack is represented by an array, Figure S6.1 shows the contents of `symbolStack` while the phrase '`main (String[] args) { System.out.print(arg[0]); }`' is being checked.

6.4 Assuming that the stack is represented by an SLL, Figure S6.2 shows the contents of `symbolStack` while the phrase '`main (String[] args) { System .out.print(arg[0]); }`' is being checked.

6.5 Add the following accessor to the stack contract of Program 6.6:

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

Add the following to the array implementation of Program 6.8:

```
public Object get (int d) {
  if (d < 1 || d > depth)
    throw new NoSuchElementException();
  return elems[depth-d];
}
```

Add the following to the SLL implementation of Program 6.10:

```
public Object get (int d) {
  if (d < 1)
    throw new NoSuchElementException();
  SLLNode curr = top;
  for (int i = 1; i < d; i++) {
    if (curr == null)
      throw new NoSuchElementException();
    curr = curr.succ;
  }
  return curr.element;
}
```

6.6 To make the array implementation of Program 6.8 deal with an overflow by throwing an exception:

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

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

6.7 An implementation of pairs of bounded stacks is shown in Program S6.3.

6.13 To reorder a train from *input* to *output*, using *spur*:

1. For *c* = 1, …, *n*, repeat:
   1.1. Set *loc*[*c*] to *input*.
2. For *c* = 1, …, *n*, repeat:
   2.1. Let *here* be *loc*[*c*].
   2.2. If *here* is *input*:
      2.2.1. While the top car number in *input* is not *c*, repeat:
         2.2.1.1. Move car *c'* from *input* to *spur*.
         2.2.1.2. Set *loc*[*c'*] to *spur*.
      2.2.2. Move car *c* from *input* to *output*.
   2.3. If *here* is *spur*:
      2.3.1. While the top car number in *spur* is not *c*, repeat:
         2.3.1.1. Move car *c'* from *spur* to *input*.
         2.3.1.2. Set *loc*[*c'*] to *input*.
      2.3.2. Move car *c* from *spur* to *output*.
3. Terminate.

6.14 Suppose that we have *s* spurs, numbered 0, …, *s*–1. Then we can assign cars to spurs according to their car numbers. For example, we can assign car *c* to the spur numbered (*c* modulo *s*). On average, each spur will contain only about 1/*s* times as many cars as in Exercise 6.13, and the excess number of car movements will be reduced by about 1/*s*.

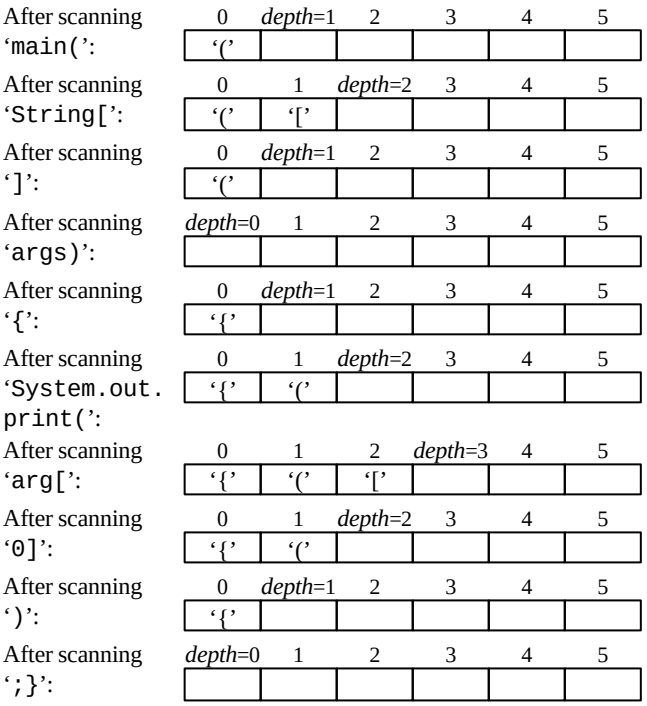

Figure S6.1 Stack contents in Algorithm 6.4 (array representation with *maxdepth* = 6).

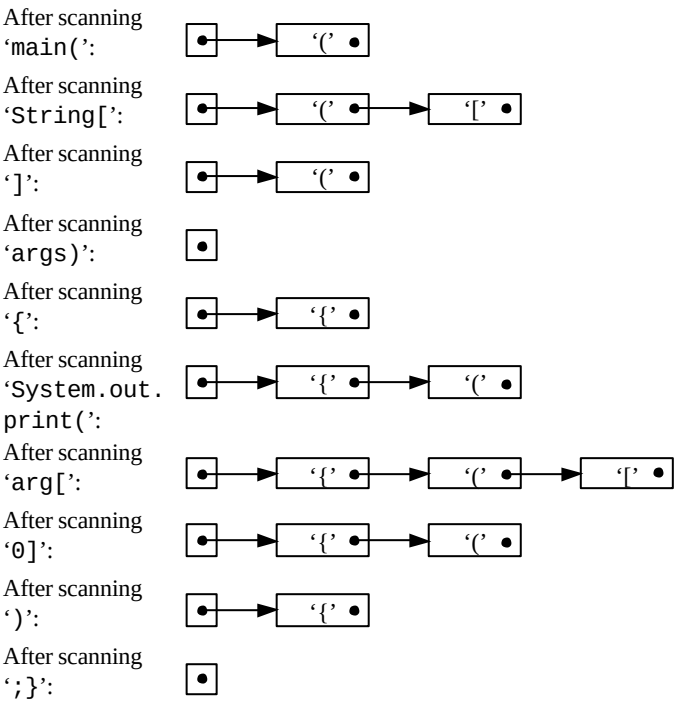

Figure S6.2 Stack contents in Algorithm 6.4 (SLL representation).

```
public class TwinStack {

  // Each TwinStack object is a pair of bounded stacks whose elements are
  // objects. The stacks are identified as LEFT and RIGHT.

  // This stack pair is represented as follows:
  // The LEFT stack's depth is held in depthL, and its elements occupy the
  // subarray elems[0…depthL–1], in bottom-to-top order.
  // The RIGHT stack's depth is held in depthR, and its elements occupy
  // the subarray elems[max–depthR…max–1], in top-to-bottom order.
  private Object[] elems;
  private int depthL, depthR;

  public static final byte LEFT = 0, RIGHT = 1;

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

  public ArrayStack (int max) {
  // Construct a stack pair, in which both stacks are initially empty, whose
  // total depth will be bounded by max.
    elems = new Object[max];
    depthL = depthR = 0;
  }

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

  public boolean isEmpty (byte id) {
  // Return true if and only if stack id in this stack pair is empty.
    switch (id) {
      case LEFT:
        return (depthL == 0);
      case RIGHT:
        return (depthR == 0);
    }
  }
```

Program S6.3 Implementation of pairs of bounded stacks *(continued on next page)*.

```
public Object getLast (byte id) {
// Return the element at the top of stack id in this stack pair. Throw a
// NoSuchElementException if that stack is empty.
   switch (id) {
     case LEFT:
       if (depthL == 0)
         throw new NoSuchElementException();
       return elems[depthL-1];
     case RIGHT:
       if (depthR == 0)
         throw new NoSuchElementException();
       return elems[elems.length-depthR];
   }
}

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

public void clear (byte id) {
// Make stack id in this stack pair empty.
   switch (id) {
     case LEFT:
       for (int i = 0; i < depthL; i++)
         elems[i] = null;
       depthL = 0;
       break;
     case RIGHT:
       for (int j = 1; j <= depthR; j++)
         elems[elems.length-j] = null;
       depthR = 0;
   }
}

public void addLast (byte id, Object elem) {
// Add elem as the top element on stack id in this stack pair.
   if (depthL + depthR == elems.length)
     expand();
   switch (id) {
     case LEFT:
       elems[depthL++] = elem;
       break;
     case RIGHT:
       elems[elems.length-(++depthR)] = elem;
   }
}
```

Program S6.3 Implementation of pairs of bounded stacks *(continued on next page)*.

```
  public Object removeLast (byte id) {
  // Remove and return the element at the top of stack id in this stack pair.
  // Throw a NoSuchElementException if that stack is empty.
    Object topElem;
    switch (id) {
      case LEFT:
        if (depthL == 0)
          throw new NoSuchElementException();
        topElem = elems[--depthL];
        elems[depthL] = null;
        break;
      case RIGHT:
        if (depthR == 0)
          throw new NoSuchElementException();
        topElem = elems[elems.length-depthR];
        elems[elems.length-(depthR--)] = null;
    }
    return topElem;
  }

  //////////// Auxiliary method ////////////

  private void expand () {
  // Make the elems array longer.
    Object[] newElems = new Object[2*elems.length];
    for (int i = 0; i < depthL; i++)
      newElems[i] = elems[i];
    for (int j = 1; j <= depthR; j++)
      newElems[newElems.length-j] =
          elems[elems.length-j];
    elems = newElems;
  }

}
```

Program S6.3 Implementation of pairs of bounded stacks *(continued)*.