? Solutions to Self-Test Exercises

1.
```
public class DoubleNode
{
   double data;
   DoubleNode link;
   ...
```

2.
```
public class DINode
{
   double d_data;
   int i_data;
   DINode link;
```

3. The head node and the tail node.

4. The null reference is used for the link part of the final node of a linked list; it is also used for the head and tail references of a list that doesn't yet have any nodes.

5. No nodes. The head and tail are `null`.

6. A `NullPointerException` is thrown.

7. Using techniques from Section 4.2:
```
if (head == null)
    head = new IntNode(42, null);
else
    head.addNodeAfter(42);
```

8. Using techniques from Section 4.2:
```
if (head != null)
{
    if (head.getLink( ) == null)
       head = null;
    else
       head.removeNodeAfter( );
}
```

9. They cannot be implemented as ordinary methods of the `IntNode` class because they must change the head reference (making it refer to a new node).

10.
```
int i;
head = new IntNode(1, null);
tail = head;
for (i = 2; i <= 100; i++)
{
   tail.addNodeAfter(i);
   tail = tail.getLink( );
}
```

11. There are eight different nodes for the eight primitive data types (boolean, int, long, byte, short, double, float, and char). These are called `BooleanNode`, `IntNode`, and so on. There is one more class simply called `Node`, which will be discussed in Chapter 5. The data type in the `Node` class is Java's `Object` type. So there are nine different nodes in all.

    If you implement one of these nine node types, implementing another one takes little work—just change the type of the data and the type of any method parameters that refer to the data.

12. Within the `IntNode` class, you may write:
```
locate = locate.link;
```
Elsewhere, you must write:
```
locate = locate.getLink( );
```
If `locate` is already referring to the last node before the assignment statement, then the assignment will set `locate` to null.

13. The new operator is used in the method `add-NodeAfter`, `listCopy`, `listCopyWithTail`, and `listPart`.

14. It will be the null reference.

15. The `listCopyWithTail` method does exactly this by returning an array with two `IntNode` components.

16.
```
douglass =
    IntNode.listSearch(head, 42);
```


17.
```
IntNode cursor;
for (cursor = head;
     cursor != null;
     cursor = cursor.link;
   )
     System.out.print(cursor.data);
```

18.
```
public static int count42
(IntNode head)
{
   int count = 0;
   IntNode cursor;
   for (cursor = head;
        cursor != null;
        cursor = cursor.link;
       )
       {
           if (cursor.data == 42)
            count++;
       }
     return count;
}
```

19.
```
public static boolean has42
(IntNode head)
{
   IntNode cursor;
   for (cursor = head;
        cursor != null;
        cursor = cursor.link;
       )
       {
           if (cursor.data == 42)
            return true;
       }
     return false;
}
```

20.
```
public static int sum
(IntNode head)
{
   int count = 0;
   IntNode cursor;
   for (cursor = head;
        cursor != null;
        cursor = cursor.link;
       )
       {
            count += cursor.data;
       }
     return count;
}
```

21.
```
public static void tail42
(IntNode head)
{
   IntNode cursor;
   if (head = null)
      head = new IntNode(42, null);
   else
   {
       // Move cursor to tail...
       cursor = head;
       while(cursor.link != null)
            cursor = cursor.link;
       // ...and add the new node:
       cursor.addNodeAfter(42);
   }
}
```

22.
```
public static void remove42
(IntNode head)
{
   // Remove first node after the head that
   // contains a data of 42 (if there is one).
   IntNode cursor;
   if (head = null)
   {  // Empty list
      return;
   }

   // Step through each node of the list,
   // always looking to see whether the
   // next node has data of 42.
   cursor = head;
   while (cursor->link != null)
   {
      if (cursor->link->data == 42)
      {  // The next node's data is 42,
         // so remove the next node
         // and return.
         cursor->link =
            cursor->link->link;
         return;
      }
   }

}
```

23.
```
IntNode[ ] array;
array = IntNode.listPart(p. q);
x = array[0];
y = array[1];
array = IntNode.listPart(q.link, r);
y.link = array[0];
z = array[1];
```

24. These would change: add, `addAll`, remove, `union`. You could then implement a more efficient `countOccurrences` that stopped when it found a number bigger than the target (but it would still have linear worst time).

25. Use DoubleNode instead of IntNode. There are a few other changes such as changing some parameters from int to double.

26. We could write this code:

```
IntLinkedBag exercise =
   new IntLinkedBag( );
exercise.add(42);
exercise.add(8);
System.out.println
   (exercise.grab( ));
System.out.println
   (exercise.size( ));
```

27. Generally, we will choose the approach that makes the best use of the node methods. This saves us work and also reduces the chance of new errors from writing new code to do an old job. The preference would change if the other approach offered better efficiency.

28. The two lines of code we have in mind:

```
p = p.getLink( );
p = listSearch(p, d);
```

These two lines are the same as this line:

```
p = listSearch(p.getLink( ), d);
```

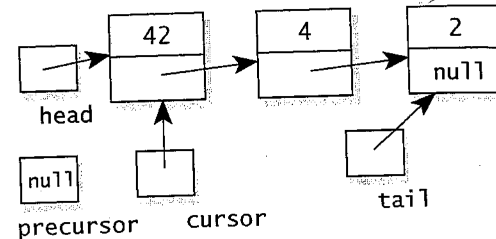
29. When the target is not in the bag, the first assignment statement to cursor will set it to null. This means that the body of the loop will not execute at all, and the method returns the answer zero.

30. Test the case where you are removing the last element from the bag.

31. `(int) (Math.random( ) * 21) - 10`

32. All the methods are constant time except for remove, grab, countOccurrences, and clone (all of which are linear); the addAll method (which is $O(n)$, where n is the size of the addend); and the union method (which is $O(m+n)$, where m and n are the sizes of the two bags).

33. The new bag and the old bag would then share the same linked list.



35. First check that the element occurs somewhere in the sequence. If it doesn't, then return with no work. If the element is in the sequence, then set the current element to be equal to this element and activate the ordinary remove method.

36. The two add methods both allocate dynamic memory, as do addAll, clone, and concatenation.

37. The clone method should use listPart, as described on page 228.

38. Arrays are quickest for random access.

39. Linked lists are quickest for additions/removals at a cursor.

40. A doubly linked list.

41. At least $O(n)$, where n is the size of the array prior to changing the size. If the new array is initialized, then there is also $O(m)$ work, where m is the size of the new array.

42.

```
public void removeTwoWay
{
   IntTwoWayNode before;
   IntTwoWayNode after;

   before = backlink;
   after forelink;

   if (before != null)
      before.forelink = after;

   if (after != null)
      after.backlink = before;

   forelink = backlink = null;
```