DATA STRUCTURES & ALGORITHMS IN JAVA™

SECOND EDITION

ROBERT LAFORE

SAMS

Robert Lafore

Data Structures & Algorithms in Java

Second Edition

SAMS

800 East 96th Street, Indianapolis, Indiana 46240

LISTING 5.6 Continued

```
   }  // end class Link
////////////////////////////////////////////////////////////////
class SortedList
   {
   private Link first;                 // ref to first item on list
// -------------------------------------------------------------
   public SortedList()                 // constructor
      { first = null; }
// -------------------------------------------------------------
   public boolean isEmpty()            // true if no links
      { return (first==null); }
// -------------------------------------------------------------
   public void insert(long key)        // insert, in order
      {
      Link newLink = new Link(key);    // make new link
      Link previous = null;            // start at first
      Link current = first;
                                       // until end of list,
      while(current != null && key > current.dData)
         {                             // or key > current,
         previous = current;
         current = current.next;       // go to next item
         }
      if(previous==null)               // at beginning of list
         first = newLink;              // first --> newLink
      else                             // not at beginning
         previous.next = newLink;      // old prev --> newLink
      newLink.next = current;          // newLink --> old current
      }  // end insert()
// -------------------------------------------------------------
   public Link remove()           // return & delete first link
      {                           // (assumes non-empty list)
      Link temp = first;               // save first
      first = first.next;              // delete first
      return temp;                     // return value
      }
// -------------------------------------------------------------
   public void displayList()
      {
      System.out.print("List (first-->last): ");
      Link current = first;       // start at beginning of list
```

LISTING 5.6 Continued

```
      while(current != null)      // until end of list,
         {
         current.displayLink();   // print data
         current = current.next;  // move to next link
         }
      System.out.println("");
      }
   }  // end class SortedList
////////////////////////////////////////////////////////////////
class SortedListApp
   {
   public static void main(String[] args)
      {                            // create new list
      SortedList theSortedList = new SortedList();
      theSortedList.insert(20);    // insert 2 items
      theSortedList.insert(40);

      theSortedList.displayList(); // display list

      theSortedList.insert(10);    // insert 3 more items
      theSortedList.insert(30);
      theSortedList.insert(50);

      theSortedList.displayList(); // display list

      theSortedList.remove();      // remove an item

      theSortedList.displayList(); // display list
      }  // end main()
   }  // end class SortedListApp
////////////////////////////////////////////////////////////////
```

In `main()` we insert two items with key values 20 and 40. Then we insert three more items, with values 10, 30, and 50. These values are inserted at the beginning of the list, in the middle, and at the end, showing that the `insert()` routine correctly handles these special cases. Finally, we remove one item, to show removal is always from the front of the list. After each change, the list is displayed. Here's the output from `sortedList.java`:

```
List (first-->last): 20 40
List (first-->last): 10 20 30 40 50
List (first-->last): 20 30 40 50
```

Efficiency of Sorted Linked Lists

Insertion and deletion of arbitrary items in the sorted linked list require O(N) comparisons (N/2 on the average) because the appropriate location must be found by stepping through the list. However, the minimum value can be found, or deleted, in O(1) time because it's at the beginning of the list. If an application frequently accesses the minimum item, and fast insertion isn't critical, then a sorted linked list is an effective choice. A priority queue might be implemented by a sorted linked list, for example.

List Insertion Sort

A sorted list can be used as a fairly efficient sorting mechanism. Suppose you have an array of unsorted data items. If you take the items from the array and insert them one by one into the sorted list, they'll be placed in sorted order automatically. If you then remove them from the list and put them back in the array, the array will be sorted.

This type of sort turns out to be substantially more efficient than the more usual insertion sort within an array, described in Chapter 3, "Simple Sorting," because fewer copies are necessary. It's still an $O(N^2)$ process because inserting each item into the sorted list involves comparing a new item with an average of half the items already in the list, and there are N items to insert, resulting in about $N^2/4$ comparisons. However, each item is copied only twice: once from the array to the list and once from the list to the array. N*2 copies compares favorably with the insertion sort within an array, where there are about N^2 copies.

Listing 5.7 shows the `listInsertionSort.java` program, which starts with an array of unsorted items of type `link`, inserts them into a sorted list (using a constructor), and then removes them and places them back into the array.

LISTING 5.7 The `listInsertionSort.java` Program

```
// listInsertionSort.java
// demonstrates sorted list used for sorting
// to run this program: C>java ListInsertionSortApp
////////////////////////////////////////////////////////////////
class Link
   {
   public long dData;                  // data item
   public Link next;                   // next link in list
// -------------------------------------------------------------
   public Link(long dd)                // constructor
      { dData = dd; }
// -------------------------------------------------------------
```

LISTING 5.7 Continued

```
   }  // end class Link
////////////////////////////////////////////////////////////////
class SortedList
   {
   private Link first;            // ref to first item on list
// -------------------------------------------------------------
   public SortedList()            // constructor (no args)
      { first = null; }                    // initialize list
// -------------------------------------------------------------
   public SortedList(Link[] linkArr)  // constructor (array
      {                               // as argument)
      first = null;                        // initialize list
      for(int j=0; j<linkArr.length; j++)  // copy array
         insert( linkArr[j] );             // to list
      }
// -------------------------------------------------------------
   public void insert(Link k)     // insert (in order)
      {
      Link previous = null;            // start at first
      Link current = first;
                                       // until end of list,
      while(current != null && k.dData > current.dData)
         {                             // or key > current,
         previous = current;
         current = current.next;       // go to next item
         }
      if(previous==null)               // at beginning of list
         first = k;                    // first --> k
      else                             // not at beginning
         previous.next = k;            // old prev --> k
      k.next = current;                // k --> old current
      }  // end insert()
// -------------------------------------------------------------
   public Link remove()           // return & delete first link
      {                           // (assumes non-empty list)
      Link temp = first;               // save first
      first = first.next;              // delete first
      return temp;                     // return value
      }
// -------------------------------------------------------------
   }  // end class SortedList
```

LISTING 5.7 Continued

```
////////////////////////////////////////////////////////////////
class ListInsertionSortApp
   {
   public static void main(String[] args)
      {
      int size = 10;
                               // create array of links
      Link[] linkArray = new Link[size];

      for(int j=0; j<size; j++)  // fill array with links
         {                          // random number
         int n = (int)(java.lang.Math.random()*99);
         Link newLink = new Link(n);  // make link
         linkArray[j] = newLink;      // put in array
         }
                               // display array contents
      System.out.print("Unsorted array: ");
      for(int j=0; j<size; j++)
         System.out.print( linkArray[j].dData + " " );
      System.out.println("");
                               // create new list
                               // initialized with array
      SortedList theSortedList = new SortedList(linkArray);

      for(int j=0; j<size; j++)  // links from list to array
         linkArray[j] = theSortedList.remove();
                               // display array contents
      System.out.print("Sorted Array:   ");
      for(int j=0; j<size; j++)
         System.out.print(linkArray[j].dData + " ");
      System.out.println("");
      }  // end main()
   }  // end class ListInsertionSortApp
////////////////////////////////////////////////////////////////
```

This program displays the values in the array before the sorting operation and again afterward. Here's some sample output:

```
Unsorted array: 59 69 41 56 84 15 86 81 37 35
Sorted array:   15 35 37 41 56 59 69 81 84 86
```

The output will be different each time because the initial values are generated randomly.

A new constructor for `SortedList` takes an array of `Link` objects as an argument and inserts the entire contents of this array into the newly created list. By doing so, it helps make things easier for the client (the `main()` routine).

We've also made a change to the `insert()` routine in this program. It now accepts a `Link` object as an argument, rather than a `long`. We do this so we can store `Link` objects in the array and insert them directly into the list. In the `sortedList.java` program (Listing 5.6), it was more convenient to have the `insert()` routine create each `Link` object, using the `long` value passed as an argument.

The downside of the list insertion sort, compared with an array-based insertion sort, is that it takes somewhat more than twice as much memory: The array and linked list must be in memory at the same time. However, if you have a sorted linked list class handy, the list insertion sort is a convenient way to sort arrays that aren't too large.

Doubly Linked Lists

Let's examine another variation on the linked list: the *doubly linked* list (not to be confused with the double-ended list). What's the advantage of a doubly linked list? A potential problem with ordinary linked lists is that it's difficult to traverse backward along the list. A statement like

```
current=current.next
```

steps conveniently to the next link, but there's no corresponding way to go to the previous link. Depending on the application, this limitation could pose problems.

For example, imagine a text editor in which a linked list is used to store the text. Each text line on the screen is stored as a `String` object embedded in a link. When the editor's user moves the cursor downward on the screen, the program steps to the next link to manipulate or display the new line. But what happens if the user moves the cursor upward? In an ordinary linked list, you would need to return `current` (or its equivalent) to the start of the list and then step all the way down again to the new current link. This isn't very efficient. You want to make a single step upward.

The doubly linked list provides this capability. It allows you to traverse backward as well as forward through the list. The secret is that each link has two references to other links instead of one. The first is to the next link, as in ordinary lists. The second is to the previous link. This type of list is shown in Figure 5.13.