Solutions to Exercises in Chapter 16

16.1 The effects of successive insertions in an AVL-tree of words are shown in Figure S16.1. The effects of successive deletions are shown in Figure S16.2.

16.2 AVL-tree insertion and rotation in cases (3) and (4) are shown in Figures S16.3 and S16.4, respectively.

16.3 AVL-tree deletion and rotation in case (1) when the height of *T3* is *h* rather than *h*+1 is shown in Figures S16.5. The rotated subtree's height decreases from *h*+3 to *h*+2.

AVL-tree deletion and rotation in cases (3) and (4) are shown in Figures S16.6 and S16.7, respectively.

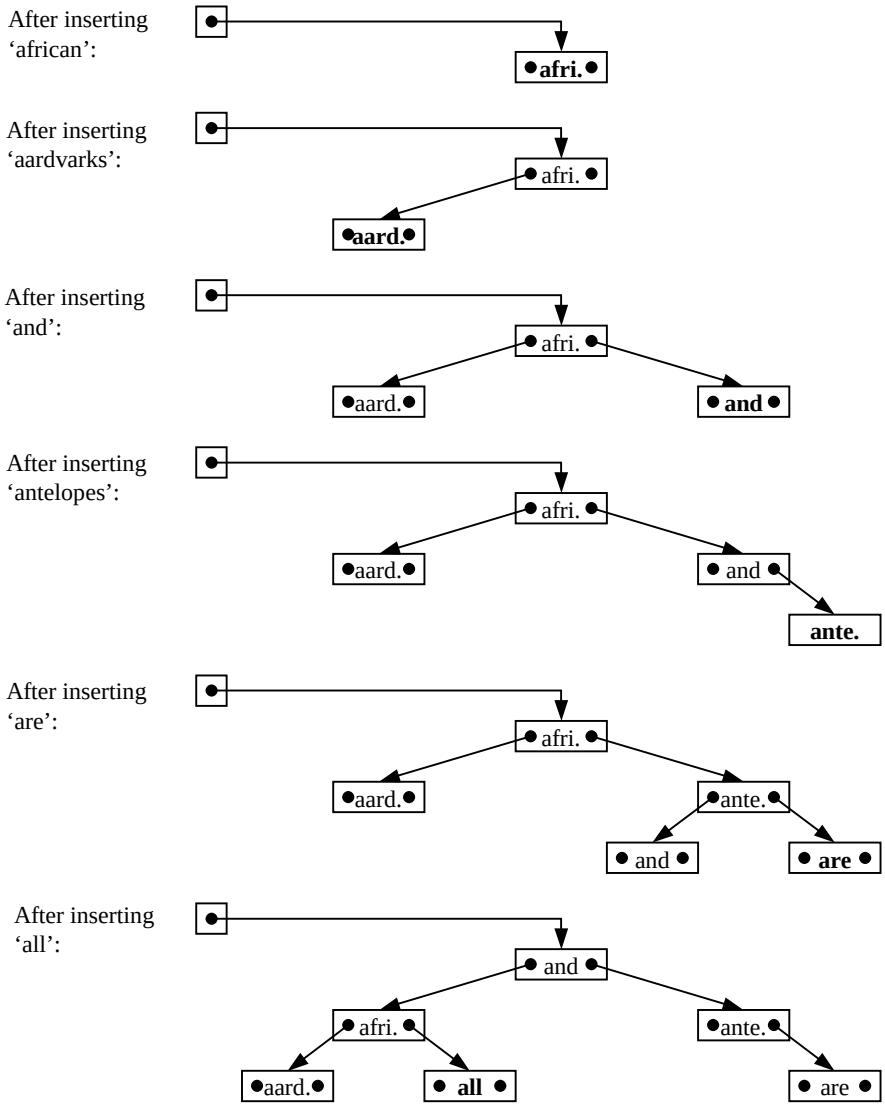

Figure S16.1 Effect of successive insertions in an AVL-tree *(continued on next page).*

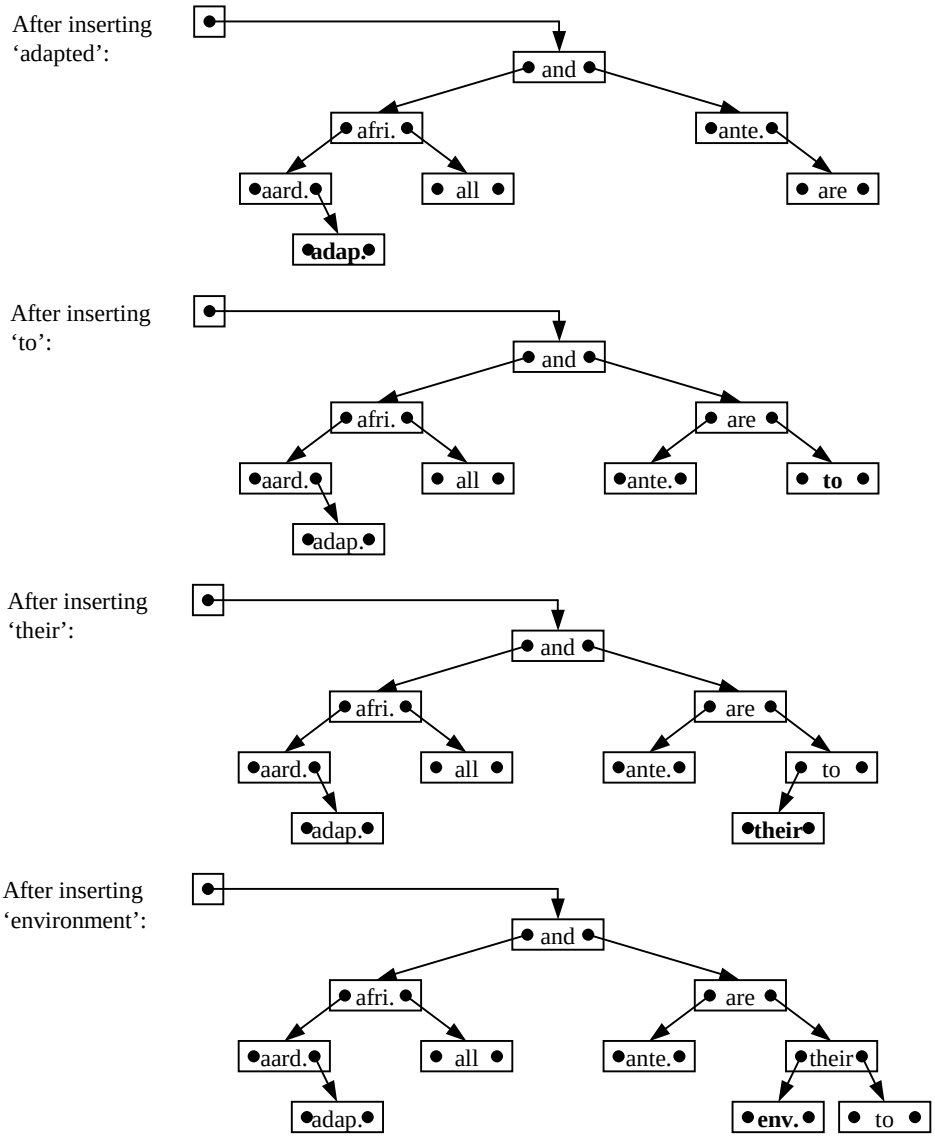

Figure S16.1 Effect of successive insertions in an AVL-tree *(continued)*.

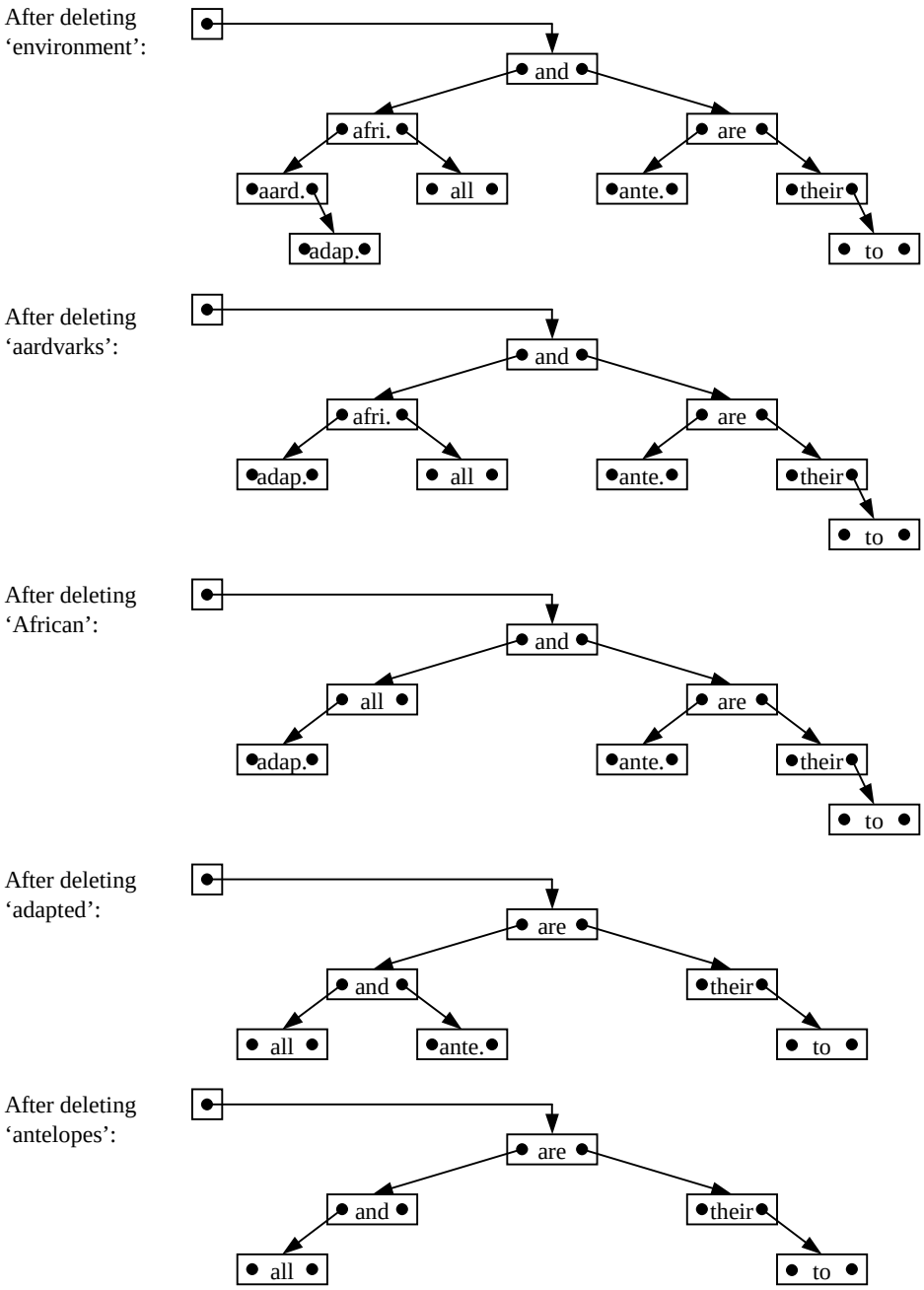

Figure S16.2 Effect of successive deletions in an AVL-tree.

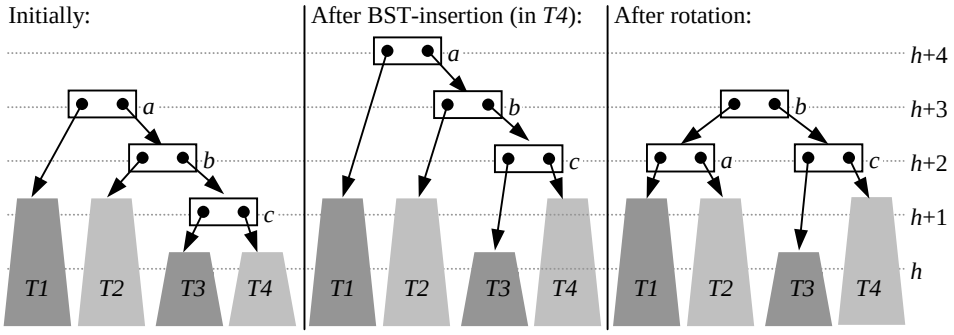

Figure S16.3 AVL-tree insertion and rotation: case (3).
(*Note:* BST-insertion in *T3* has a similar effect.)

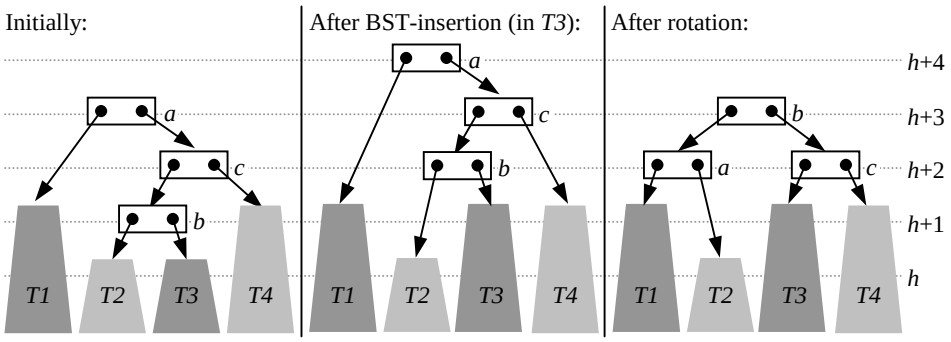

Figure S16.4 AVL-tree insertion and rotation: case (4).
(*Note:* BST-insertion in *T2* has a similar effect.)

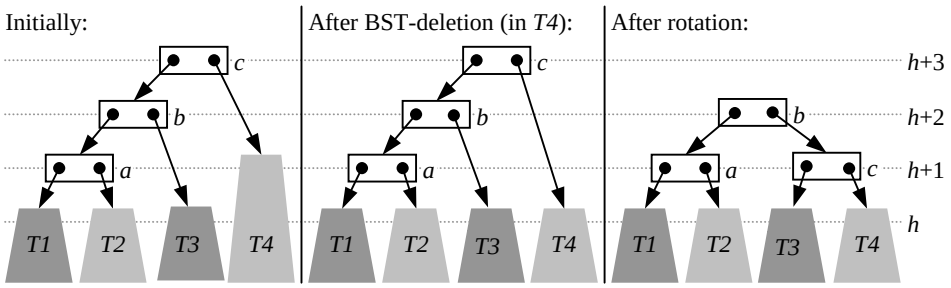

Figure S16.5 AVL-tree deletion and rotation: case (1) with *T3* having height *h*.

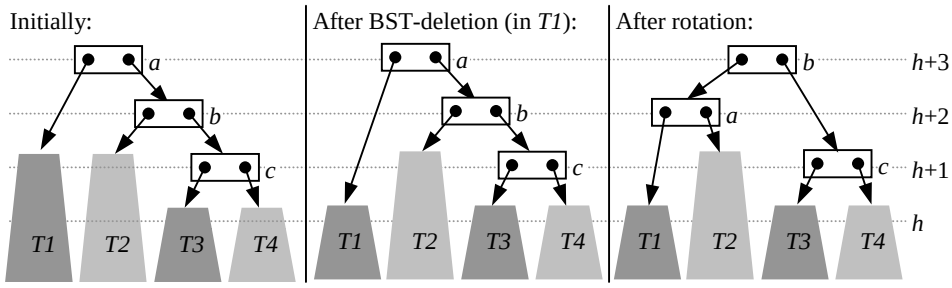

Figure S16.6 AVL-tree deletion and rotation: case (3).

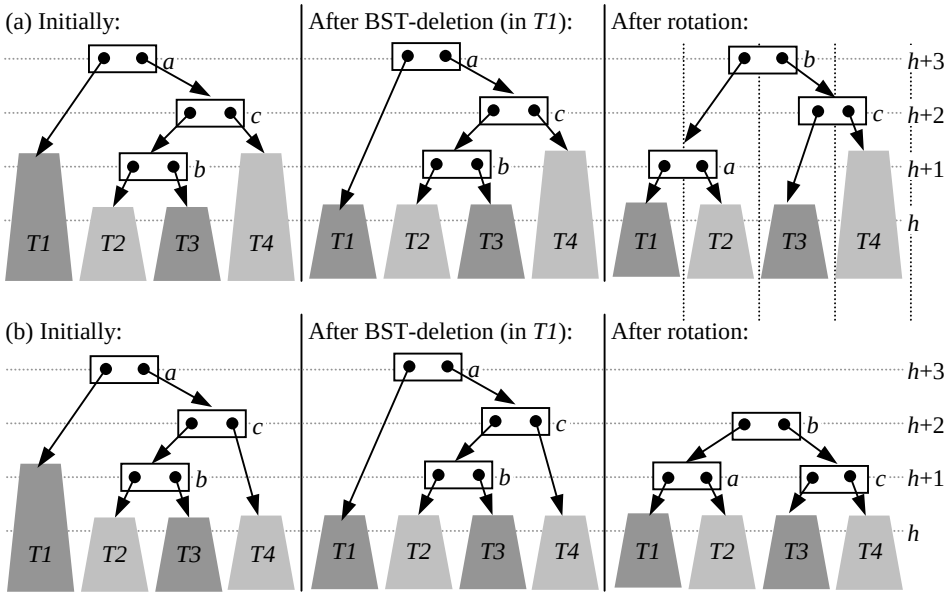

Figure S16.7 AVL-tree deletion and rotation: case (4).

16.4 To test whether a given BST is height-balanced, add the following method to the BST class:

```
public boolean isHeightBalanced () {
// Return true if and only if this BST is height-balanced.
   if (root == null)  return true;
   return BSTNode.isHeightBalanced(root);
}
```

Also add the following methods to the BSTNode class:

```
public static boolean isHeightBalanced(
              BSTNode top) {
// Return true if and only if the subtree whose topmost node is top is
// height-balanced.
   if (top == null)
      return true;
   else {
      return (isHeightBalanced(left)
           && isHeightBalanced(right))
           && Math.abs(height(left) -
                height(right)) <= 1);
}

public static int height(BSTNode top) {
// Return the height of the subtree whose topmost node is top, or –1 if
// top is null.
   if (top == null)
      return -1;
   else {
      return 1 + Math.max(height(left),
           height(right));
}
```

16.5 A Java implementation of AVL-trees is available at the book's Web site.

16.6 To make each B-tree node occupy a complete disk block, set $k = 128$.

A B-tree of $2^{21} - 1$ elements has about 2^{14} nodes, and hence occupies about 2^{14} disk blocks, provided that all nodes are fully-occupied.

Let this B-tree have depth d. From equation (16.6) we have $2^{21} - 1 = 128^{d+1} - 1$, so $2^{21} = 128^{d+1} = 2^{7(d+1)}$, so $d = 2$.

The maximum number of nodes visited (and hence the maximum number of disk transfers) is 3. The maximum number of comparisons at each node is 7, so the maximum number of comparisons altogether is 21.

The *average* numbers are nearly the same, since the vast majority of searches will terminate at the leaf nodes.

16.7 The effects of successive insertions and deletions in a 5-ary B-tree of words are shown in Figure S16.8.

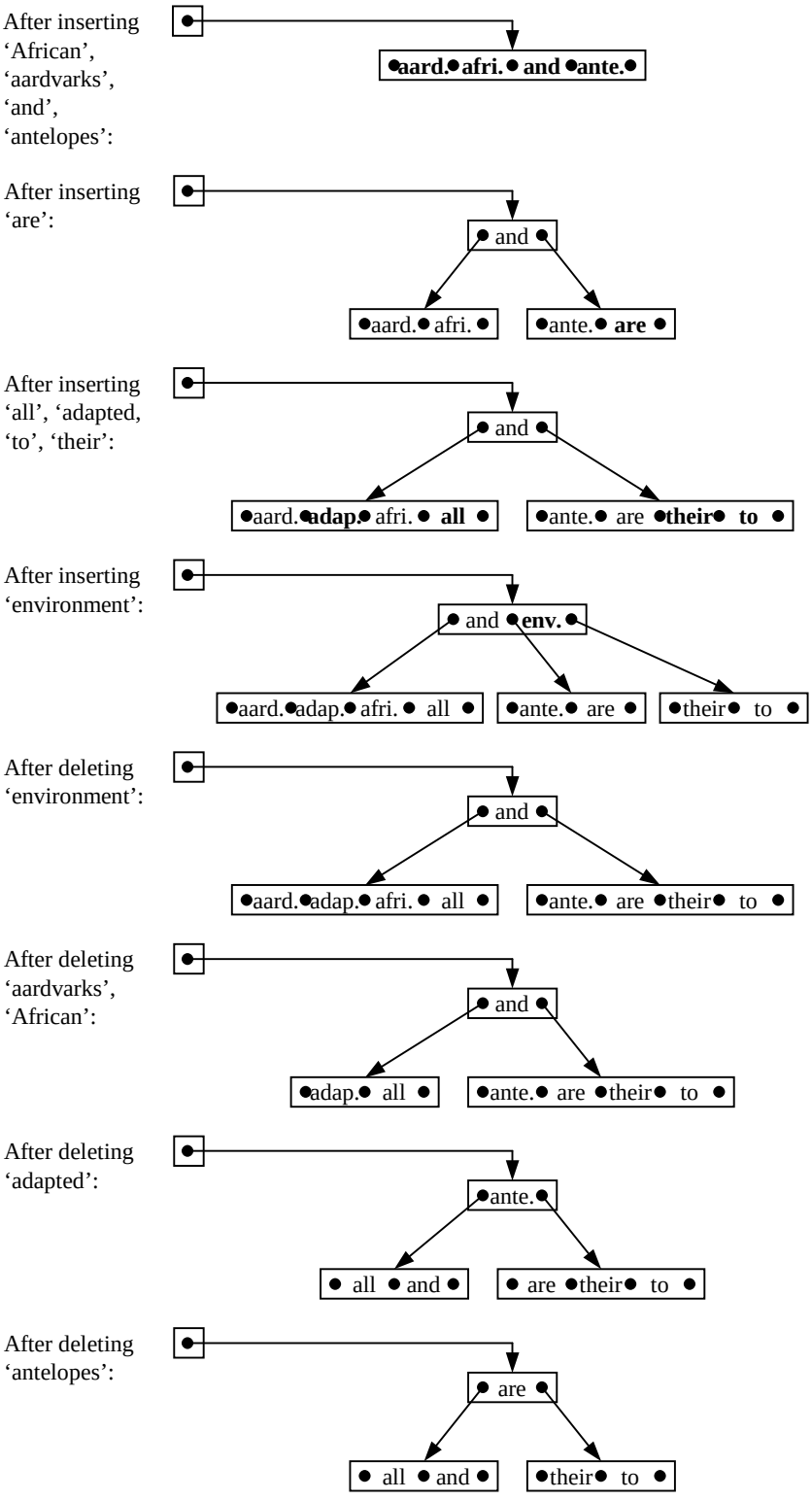

Figure S16.8 Illustration of successive insertions and deletions in a 5-ary B-tree.

16.8 A simple way to represent B-trees without wasting space in leaf nodes is to adopt the convention that the variable `childs` in a *leaf* node contains a null link (rather than a link to an array of null links). The modified implementation is outlined in Programs S16.9 and S16.10.

16.9 Methods to iterate over the elements of a B-tree are shown in Program S16.11.

16.10 A modified version of the `splitInNode` method is shown in Program S16.12.

```
public class Btree {

   // Each Btree object is a B-tree header.

   // This B-tree is represented as follows: arity is the maximum number
   // of children per node, and root is a link to its root node.

   // Each B-tree node is represented as follows: size contains its size; a
   // subarray elems[0…size–1] contains its elements;and a subarray
   // childs[0…size] contains links to its child nodes. For each element
   // elems[i], childs[i] is a link to its left child, and childs[i+1] is a
   // link to its right child. In a leaf node, childs is null.

   // Moreover, for every element x in the left subtree of element y:
   //     x.compareTo(y) < 0
   // and for every element z in the right subtree of element y:
   //     z.compareTo(y)> 0.

   private int arity;
   private Node root;

   public Btree (int k) {
   // Construct an empty B-tree of arity k.
      root = null;
      arity = k;
   }

   public Node search (Comparable target) {
   // Find which if any node of this B-tree contains an element equal to
   // target. Return a link to that node, or null if there is no such node.
      …  // unchanged
   }

   public void insert (Comparable elem) {
   // Insert element elem into this B-tree.
      if (root == null) {
         root = new Node(arity, elem);
         return;
      }
      Stack ancestors = new LinkedStack();
      Node curr = root;
      for (;;) {
         int currPos = curr.searchInNode(elem);
         if (elem.equals(curr.elems[currPos]))
            return;
         else if (curr.isLeaf())
            break;
         else {
            // Continue the search in childs[currPos], which is the left
            // child of the least element in node curr greater than elem.
            ancestors.addLast(new Integer(currPos));
            ancestors.addLast(curr);
            curr = curr.childs[currPos];
         }
      }
      curr.insertInNode(elem, currPos);
      if (curr.size == arity)  // curr has overflowed
         splitNode(curr, ancestors);
   }
```

Program S16.9 Modified Java class representing B-tree headers (changes italicized)

(continued on next page).

```
   private void splitNode (Node node,
                           Stack ancestors) {
   // Split the overflowed node in this B-tree. The stack ancestors
   // contains all ancestors of node, together with the known insertion
   // position in each of these ancestors.
      int medPos = node.size/2;
      Comparable med = node.elems[medPos];
      Node leftSib, rightSib;
      if (node.isLeaf()) {
         leftSib = new Node(arity, node.elems,
                0, medPos);
         rightSib = new Node(arity, node.elems,
                medPos+1, node.size);
      } else {
         leftSib = new Node(arity, node.elems,
              node.childs, 0, medPos);
         rightSib = new Node(arity, node.elems,
              node.childs, medPos+1, node.size);
      }
      if (node == root)
         root = new Node(arity, med, leftSib,
              rightSib);
      else {
         Node parent = (Node)ancestors.removeLast();
         int parentIns = ((Integer)
              ancestors.removeLast()).intValue();
         parent.insertInNode(med, leftSib, rightSib,
              parentIns);
         if (parent.size == arity)  // parent has overflowed
            splitNode(parent, ancestors);
      }
   }

   …  // Other Btree methods.

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

   …  // See Program S16.10.

}
```

Program S16.9 Modified Java class representing B-tree headers (changes italicized) *(continued)*.

```
private static class Node {

   // Each Btree.Node object is a B-tree node.

   private int size;
   private Comparable[] elems;
   private Node[] childs;

   private Node (int k, Comparable elem) {
   // Construct a B-tree leaf node of arity k, initially with one element, elem.
      this.elems = new Comparable[k];
      // … The array has one extra component, to allow for possible
      // overflow.
      this.size = 1;
      this.elems[0] = elem;
      this.childs = null;
   }

   private Node (int k, Comparable[] elems,
                 int l, int r) {
   // Construct a B-tree leaf node of arity k, with its elements taken from the
   // subarray elems[l…r–1].
      this.elems = new Comparable[k];
      this.size = 0;
      for (int j = l; j < r; j++) {
         this.elems[size] = elems[j];
         this.size++;
      }
      this.childs = null;
   }

   private Node (int k, Comparable elem,
                 Node left, Node right) {
   // Construct a B-tree non-leaf node of arity k, initially with one element,
   // elem, and two children, left and right.
      … // unchanged
   }

   private Node (int k, Comparable[] elems,
                 Node[] childs, int l, int r) {
   // Construct a B-tree non-leaf node of arity k, with its elements taken from
   // the subarray elems[l…r–1] and its children from the subarray
   // childs[l…r].
      … // unchanged
   }

   private boolean isLeaf () {
   // Return true if and only of this node is a leaf.
      return (childs == null);
   }

   private int searchInNode (Comparable target) {
   // Return the index of the least element in this node that is not less than
   // target.
      … // unchanged
   }
```

Program S16.10 Modified Java inner class representing B-tree nodes (changes italicized)
(continued on next page).

```
   private void insertInNode (Comparable elem,
                              int ins) {
   // Insert element elem at position ins in this leaf node.
      for (int i = node.size; i > ins; i--)
        elems[i] = elems[i-1];
      size++;
      elems[ins] = elem;
   }

   private void insertInNode (Comparable elem,
                              Node leftChild, Node rightChild,
                              int ins) {
   // Insert element elem, with children leftChild and rightChild, at
   // position ins in this non-leaf node.
      for (int i = node.size; i > ins; i--) {
        elems[i] = elems[i-1];
        childs[i+1] = childs[i];
      }
      size++;
      elems[ins] = elem;
      childs[ins] = leftChild;
      childs[ins+1] = rightChild;
   }

   …  // Other Btree.Node methods.

}
```

Program S16.10 Modified Java inner class representing B-tree nodes (changes italicized) *(continued)*.

```
public void printAscending () {
// Print, in ascending order, all elements of this B-tree.
   printAscending(root);
}

public static void printAscending (Node top) {
// Print, in ascending order, all elements of the subtree whose top node is top.
   if (top == null)  return;;
   for (int j = 0; j < top.size; j++) {
     printAscending(top.childs[j]);
     println(top.elems[j]);
   }
   printAscending(top.childs[top.size])
}

public Iterator ascendingIterator () {
// Return an iterator that will visit, in ascending order, all elements of this
// B-tree.
   return new InOrderIterator();
}
```

Program S16.11 Methods to iterate over a B-tree *(continued on next page)*.

```
////////////  Inner class  ////////////

private class InOrderIterator implements Iterator {

  // A Btree.InOrderIterator object is an iterator that will traverse,
  // in in-order, this Btree object.

  // This iterator is represented by a stack, track. Its two topmost elements
  // are (1) the node containing the element to be visited next, and (2) that
  // element's position within that node. Deeper in the stack are similar data
  // for ancestors of that node (but only those ancestors in which at least one
  // element remains to be visited).
  private Stack track;

  private InOrderIterator () {
    track = new LinkedStack();
    for (Node curr = root; curr != null;
              curr = curr.childs[0]) {
      track.addLast(new Integer(0));
      track.addLast(curr);
    }
  }

  public boolean hasNext () {
    return (! track.isEmpty());
  }

  public Object next () {
    if (track.isEmpty())
      throw new NoSuchElementException();
    Node node = (Node)track.removeLast();
    int pos = ((Integer)
         track.removeLast()).intValue();
    if (pos+1 < node.size) {
      track.addLast(new Integer(pos+1));
      track.addLast(node);
    }
    for (Node curr = node.childs[pos+1];
           curr != null; curr = curr.childs[0]) {
      track.addLast(new Integer(0));
      track.addLast(curr);
    }
    return node.elems[pos];
  }

  …  // The remove method is omitted here.

}
```

Program S16.11 Methods to iterate over a B-tree *(continued)*.

```
private void splitNode (Node node,
                        Stack ancestors) {
// Split the overflowed node in this B-tree. The stack ancestors contains
// all ancestors of node, together with the known insertion position in each of
// these ancestors.
  int medPos = node.size/2;
  Comparable med = node.elems[medPos];
  Node rightSib = new Node(arity, node.elems,
       node.childs, medPos+1, node.size);
  for (int i = medPos; i < node.size; i++) {
    node.elems[i] = null;  node.childs[i+1] = null;
  }
  node.size = medPos;
  if (node == root)
    root = new Node(arity, med, node, rightSib);
  else {
    Node parent = (Node)ancestors.removeLast();
    int parentIns = ((Integer)
         ancestors.removeLast()).intValue();
    parent.insertInNode(med, node, rightSib,
         parentIns);
    if (parent.size == arity)  // parent has overflowed
      splitNode(parent, ancestors);
  }
}
```

Program S16.12 Modified version of the `splitInNode` method in the `Btree` class (changes italicized).

16.11 A complete implementation of the `Btree` class, including the `delete` method, may be found at the book's Web site.

Note: These solutions to the B-tree exercises (except 16.8) are based on the B-tree representation of Section 16.2 in the book, *not* on the representation outlined in Programs S16.9 and S16.10.