Solutions to Exercises in Chapter 10

10.1 The effects of successively additions followed by deletions in a BST of chemical elements are shown in Figure S10.1.

10.2 Implementation of Algorithm 10.45 as a recursive Java method:

```
public static BSTNode search (BSTNode top,
                Comparable target);
// Find which if any node of the subtree whose topmost node is top
// contains an element equal to target. Return a link to that node (or
// null if there is none).
  if (top == null)
    return null;
  else {
    int comp = target.compareTo(top.element);
    if (comp == 0)
      return top;
    else if (comp < 0)
      return search(top.left, target);
    else // comp > 0
      return search(top.right, target);
  }
}
```

10.3 Implementation of Algorithm 10.46 as a recursive Java method:

```
public static BSTNode insert (BSTNode top,
                Comparable elem);
// Insert the element elem in the subtree whose topmost node is top.
// Return a link to the modified subtree.
  if (top == null) {
    return new BSTNode(elem, null, null);
  } else {
    int comp = elem.compareTo(top.element);
    if (comp < 0)
      top.left = insert(top.left, elem);
    else if (comp > 0)
      top.right = insert(top.right, elem);
    }
    return top;
  }
}
```

10.4 To delete the element *elem* from the subtree whose topmost node is *top* (recursive version):

1. If *top* is null:
   1.1. Terminate with answer *top*.
2. If *top* is not null:
   2.1. If *elem* is equal to node *top*'s element:
      2.1.1. Delete the topmost element in the subtree whose topmost node is node *top*, and let *del* be a link to the modified subtree.
      2.1.2. Terminate with answer *del*.
   2.2. Otherwise, if *elem* is less than node *top*'s element:
      2.2.1. Delete *elem* from the subtree whose topmost node is node *top*'s left child, updating *top*'s left child accordingly.
      2.2.2. Terminate with answer *top*.
   2.3. Otherwise, if *elem* is greater than node *top*'s element:
      2.3.1. Deleting *elem* from the subtree whose topmost node is node *top*'s right child, updating *top*'s right child accordingly.
      2.3.2. Terminate with answer *top*.

Implementation of this algorithm as a Java method:

```
public static BSTNode delete (BSTNode top,
                Comparable elem);
// Delete the element elem from the subtree whose topmost node is
// top. Return a link to the modified subtree.
   if (top == null) {
      return top;
   } else {
      int comp = elem.compareTo(top.element);
      if (comp == 0)
         return top.deleteTopmost();
      else if (comp < 0)
         top.left = delete(top.left, elem);
      else // comp > 0
         top.right = delete(top.right, elem);
      }
      return top;
   }
}
```

10.5 Java methods to return the depth of a given BST, and to return an element given its index:

```
public static int depth (BSTNode top) {
// Return the depth of the BST with root node top.
   if (top == null)
         return -1;
   else
      return 1 + Math.max(depth(top.left),
            depth(top.right));
}
```

```
public static Object get (BSTNode top, int p) {
// Return the element in the p'th node from the left of the BST with
// root node top. Throw an IndexOutOfBoundsException if
// there is no such element.
   if (top == null)
      throw new IndexOutOfBoundsException();
   else {
      int leftSize = size(top.left);
      if (p == leftSize)
         return p.element;
      else if (p < leftSize)
         return get(top.left, p);
      else // p > leftSize
         return get(top.right, p - leftSize - 1);
   }
}
```

10.6 A pre-order traversal of the BST of Figure 10.5(b) produces the following output:

cat, pig, fox, dog, lion, tiger, rat

Inserting these words in this sequence, one by one into an initially empty BST does reproduce the original BST.

10.7 If the BST save algorithm traverses the BST in *in-order*, the elements are written in ascending order to the file. If this file is then restored using the BST restore algorithm, the result will be an extremely ill-balanced BST, with the root node containing the least element and every other node being the right child of its parent.

10.8 To implement the BST save and restore algorithms, add the following methods to the `BST` class:

```
public static void save (
                ObjectOutputStream file) {
// Save this BST to the file file.
   BSTNode.save(file, root);
}

public static void load (
                ObjectInputStream file) {
// Restore this BST from the file file.
   root = null;
   for (;;) {
      Comparable elem =
            (Comparable) file.readObject();
      if (elem == null)  break;  // end of file
      insert(elem);
   }
}
```

Also add the following method to the `BSTNode` class:

```
public static void save (ObjectOutputStream file,
                BSTNode top) {
// Save the contents of the subtree whose topmost node is top to the
// file file.
   if (top != null)
      file.writeObject(top.element);
      save(file, top.left);
      save(file, top.right);
   }
}
```

(*Note:* These methods assume that the BST elements are *serializable* objects. Arbitrary serializable objects can be written to an `ObjectOutputStream` using the `writeObject` method, and can be read from an `ObjectInputStream` using the `readObject` method.)

10.9 If the set implementation of Program 10.41 is modified to store the set's cardinality in an instance variable `card`, the `size` operation will have time complexity $O(1)$ rather than $O(n)$, and this will in turn speed up the `equals` operation. The `insert` and `delete` auxiliary methods must be modified to increase or decrease `card`; this does not affect their time complexity.

10.10 To test whether a given BST is well-balanced, add the following method (which uses the `depth` method from Exercise 10.5) to the `BST` class:

```
public boolean isBalanced () {
// Return true if and only if this BST is balanced.
   if (root == null)
      return true;
   int depth = BSTNode.depth(root);
   return BSTNode.isBalanced(root, depth, 0);
}
```

Also add the following method to the `BSTNode` class:

```
public static boolean isBalanced(BSTNode top,
            int treeDepth, int nodeDepth) {
// Return true if and only if the subtree whose topmost node is top is
// balanced, where treeDepth is the depth of the entire BST and
// nodeDepth is the depth of the node top.
   if (top == null)
      return (nodeDepth <= treeDepth);
   else
      return isBalanced(top.left, treeDepth,
            nodeDepth+1) && isBalanced(top.right,
            treeDepth, nodeDepth+1);
}
```

10.11 To balance a given BST, add the following method to the `BST` class. This method first uses the `isBalanced` method from Exercise 10.10 to test whether the BST is ill-balanced. If so, it copies all its elements to an auxiliary array, makes the BST empty, and then reinserts the elements in a suitable order to produce a balanced BST.

```
public void balance () {
// Balance this BST if it is ill-balanced.
   if (! isBalanced()) {
      int size = BSTNode.size(root);
      Comparable elems = new Comparable[size];
      BSTNode.fillArray(root, elems, 0);
      root = null;
      insertAll(elems, 0, size-1);
   }
}
```

```
private void insertAll (Comparable[] elems,
                int left, int right) {
// Insert the elements of the sorted subarray elems[left…right]
// into this BST in such a way that this BST remains balanced.
  if (left <= right) {
    int mid = (left + right)/2;
    insert(elems[mid]);
    insertAll(elems, left, mid-1);
    insertAll(elems, mid+1, right);
  }
}
```

Also add the following method to the `BSTNode` class:

```
public static int fillArray (BSTNode top,
                Comparable[] elems, int i) {
// Fill the subarray elems[i…] with the sorted elements of the subtree
// whose topmost node is top. Return the index of the first unused
// component of the subarray.
  if (top == null)
    return i;
  else {
    int j = fillArray(top.left, elems, i);
    elems[j] = top.element;
    return fillArray(top.right, elems, j+1);
  }
}
```

10.12 The array representation of BSTs, with search, insertion, and deletion methods, is shown in Program S10.2.

When the BST is ill-balanced, one half of the array is much more sparsely occupied than the other half, thus harming the space efficiency.

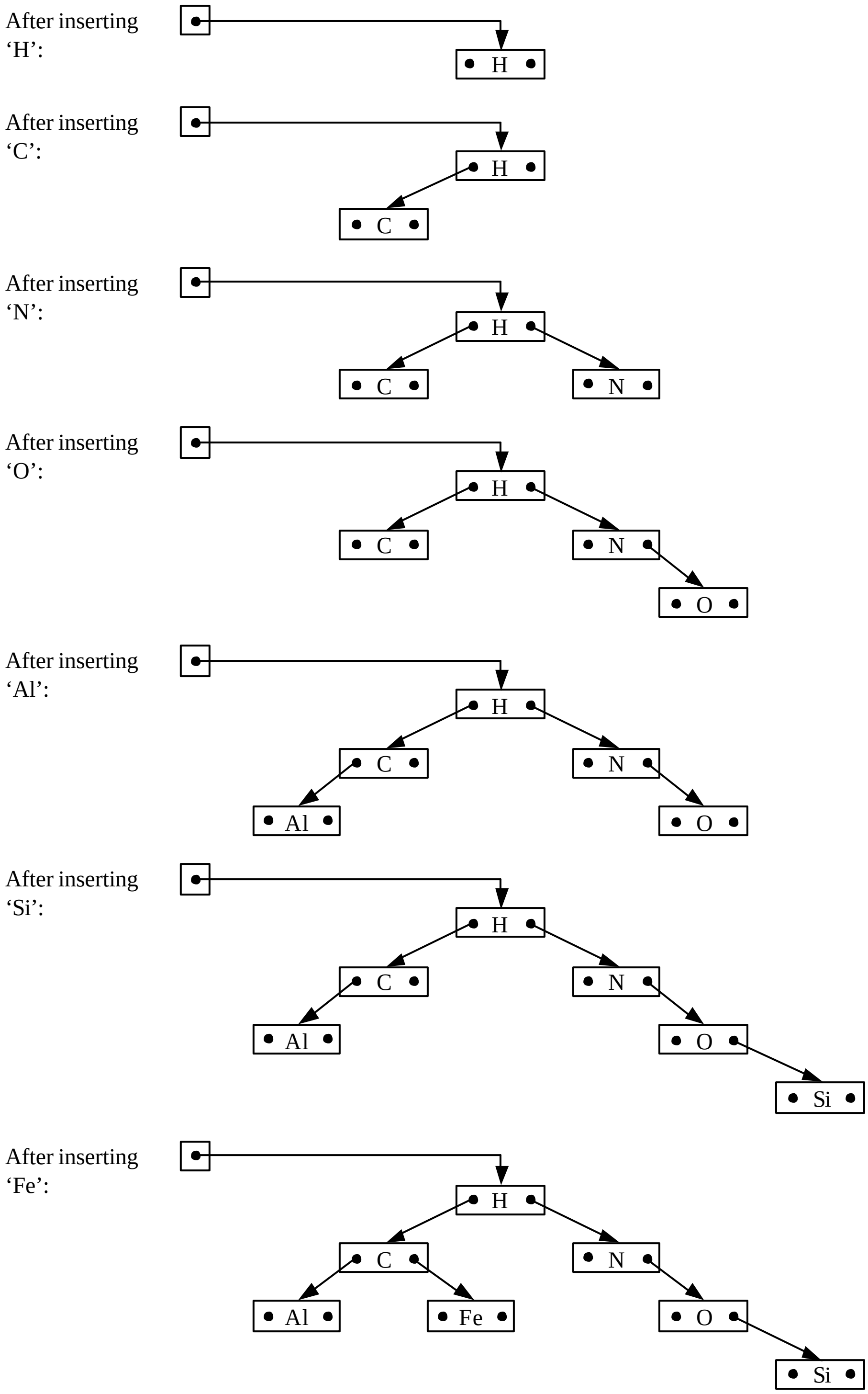
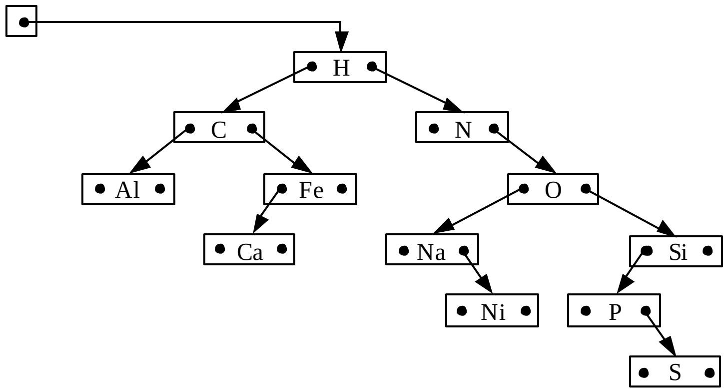
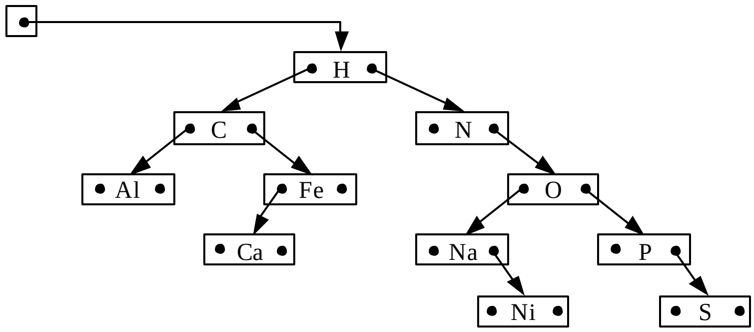


Figure S10.1 Effect of successive insertions and deletions in a BST *(continued on next page)*.

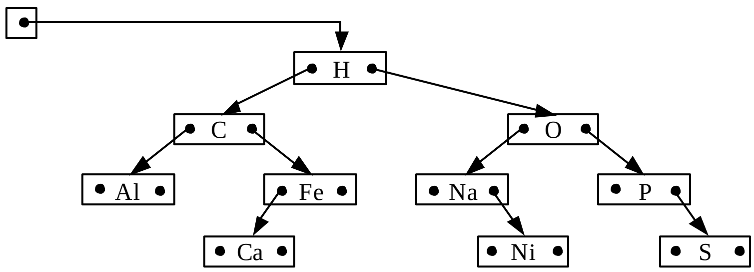
After inserting
‘Na’, ‘P’, ‘S’,
‘Ni’, and ‘Ca’:



After deleting
‘Si’:



After deleting
‘N’:



After deleting
‘O’:

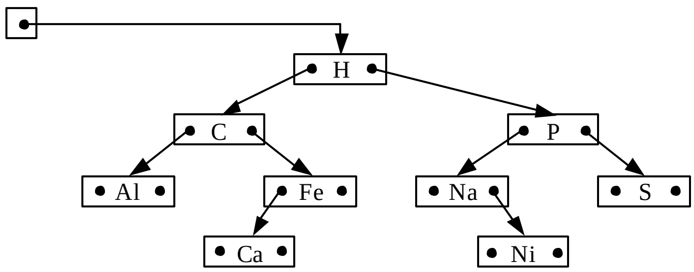


Figure S10.1 Effect of successive insertions and deletions in a BST *(continued)*.

```
public class BST {

   private Comparable[] nodes;
   private static final int ROOT = 0, NONE = -1;

   private int parent (int p) {
   // Return the position of the parent of the node at position p.
   // Return NONE if the node at position p is the root node.
      return (p == ROOT ? NONE : (p-1)/2);
   }

   private int left (int p) {
   // Return the position of the left child of the node at position p.
      return 2*p + 1;
   }

   private int right (int p) {
   // Return the position of the right child of the node at position p.
      return 2*p + 2;
   }

   private boolean exists (int p) {
   // Return true if and only if there exists a node at position p.
      return (p < nodes.length
            && nodes[p] != null);
   }

   private void moveSubtree (int p1, int p2) {
   // Move the subtree whose topmost node is at position p1 such
   // that its topmost node is at position p2.
      if (exists(p1)) {
         nodes[p1] = nodes[p2];  nodes[p2] = null;
         moveSubtree(left(p1), left(p2));
         moveSubtree(right(p1), right(p2));
      }
   }

   public int search (Comparable target) {
   // Find which if any node of this BST contains an element equal to
   // target. Return the position of that node (or NONE if there is no
   // such element).
      int direction = 0;  // = 0 for here, < 0 for left,
                          // > 0 for right
      int curr = ROOT;
      for (;;) {
         if (! exists(curr))
            return NONE;
         direction = target.compareTo(nodes[curr]);
         if (direction == 0)
            return curr;
         else if (direction < 0)
            curr = left(curr);
         else  // direction > 0
            curr = right(curr);
      }
   }
```

Program S10.2 Implementation of BSTs using arrays *(continued on next page)*.

```
public void insert (Comparable elem) {
// Insert the element elem into this BST.
  int direction = 0;  // = 0 for here, < 0 for left,
                      // > 0 for right
  int curr = ROOT;
  for (;;) {
    if (curr >= nodes.length)  expand();
    if (! exists(curr)) {
      nodes[curr] = elem;
      return;
    }
    direction = elem.compareTo(nodes[curr]);
    if (direction == 0)
      return;
    if (direction < 0)
      curr = left(curr);
    else  // direction > 0
      curr = right(curr);
  }
}

private void delete (Comparable elem) {
// Delete the element elem in this BST.
  int direction = 0;  // = 0 for here, < 0 for left,
                      // > 0 for right
  int curr = ROOT;
  for (;;) {
    if (! exists(curr))
      return;
    direction = elem.compareTo(nodes[curr]);
    if (direction == 0) {
      deleteTopmost(curr);
      return;
    } else if (direction < 0)
      curr = left(curr);
    else  // direction > 0
      curr = right(curr);
  }
}

private void deleteTopmost (int top) {
// Delete the topmost element in the subtree whose topmost node is
// the node top.
  int left = left(top), right = right(top);
  if (! exists(left)) {
    moveSubtree(right, top);
  } else if (! exists(right)) {
    moveSubtree(left, top);
  } else {  // this node has both a left child and a right child
    nodes[top] = getLeftmost(right);
    deleteLeftmost(right);
  }
}
```

Program S10.2 Implementation of BSTs using arrays *(continued on next page)*.

```
    private void deleteLeftmost (int top) {
    // Delete the leftmost element in the (nonempty) subtree whose
    // topmost  node is the node top.
        int left = left(top);
        if (! exists(left))
            moveSubtree(right(top), top);
        else {
            int curr = left;
            while (! exists(left(curr))) {
                curr = left(curr);
            }
            int parent = parent(curr);
            moveSubtree(right(curr), left(parent));
        }
    }

    private Comparable getLeftmost (int top) {
    // Return the leftmost element in the (nonempty) subtree whose
    // topmost node is the node top.
        int curr = top, left = left(top);
        while (exists(left)) {
            curr = left;  left = left(curr);
        }
        return nodes[curr];
    }

}
```

Program S10.2 Implementation of BSTs using arrays *(continued)*.