The methods could be private, but making them protected enables a derived class to use them directly.

```
import SearchKeys.KeyedItem;

// ADT binary search tree.
//  Assumption: A tree contains at most one item with a
//              given search key at any time.

public class BinarySearchTree<T extends KeyedItem<KT>,
                              KT extends Comparable<? super KT>>
       extends BinaryTreeBasis<T> {
  // inherits isEmpty(), makeEmpty(), getRootItem(), and
  // the use of the constructors from BinaryTreeBasis

  public BinarySearchTree() {
  }  // end default constructor

  public BinarySearchTree(T rootItem) {
    super(rootItem);
  }  // end constructor

  public void setRootItem(T newItem)
              throws UnsupportedOperationException {
    throw new UnsupportedOperationException();
  }  // end setRootItem

  public void insert(T newItem) {
    root = insertItem(root, newItem);
  }  // end insert

  public T retrieve(KT searchKey) {
    return retrieveItem(root, searchKey);
  }  // end retrieve

  public void delete(KT searchKey) throws TreeException {
    root = deleteItem(root, searchKey);
  }  // end delete

  public void delete(T item) throws TreeException {
    root = deleteItem(root, item.getKey());
  }  // end delete

  protected TreeNode<T> insertItem(TreeNode<T> tNode,
                                   T newItem) {
    TreeNode<T> newSubtree;
    if (tNode == null) {
```

```
    // position of insertion found; insert after leaf
    // create a new node
    tNode = new TreeNode<T>(newItem, null, null);
    return tNode;
  }  // end if
  T nodeItem = tNode.item;

  // search for the insertion position

  if (newItem.getKey().compareTo(nodeItem.getKey()) < 0) {
    // search the left subtree
    newSubtree = insertItem(tNode.leftChild, newItem);
    tNode.leftChild = newSubtree;
    return tNode;
  }
  else { // search the right subtree
    newSubtree = insertItem(tNode.rightChild, newItem);
    tNode.rightChild = newSubtree;
    return tNode;
  }  // end if
}  // end insertItem

protected T retrieveItem(TreeNode<T> tNode,
                         KT searchKey) {
  T treeItem;
  if (tNode == null) {
    treeItem = null;
  }
  else {
    T nodeItem = tNode.item;
    if (searchKey.compareTo(nodeItem.getKey()) == 0) {
      // item is in the root of some subtree
      treeItem = tNode.item;
    }
    else if (searchKey.compareTo(nodeItem.getKey()) < 0) {
      // search the left subtree
      treeItem = retrieveItem(tNode.leftChild, searchKey);
    }
    else { // search the right subtree
      treeItem = retrieveItem(tNode.rightChild, searchKey);
    }  // end if
  }  // end if
  return treeItem;
}  // end retrieveItem

protected TreeNode<T> deleteItem(TreeNode<T> tNode,
                                 KT searchKey) {
  // Calls: deleteNode.
```

```
    TreeNode<T> newSubtree;
    if (tNode == null) {
      throw new TreeException("TreeException: Item not found");
    }
    else {
      T nodeItem = tNode.item;
      if (searchKey.compareTo(nodeItem.getKey()) == 0) {
        // item is in the root of some subtree
        tNode = deleteNode(tNode);  // delete the item
      }
      // else search for the item
      else if (searchKey.compareTo(nodeItem.getKey()) < 0) {
        // search the left subtree
        newSubtree = deleteItem(tNode.leftChild, searchKey);
        tNode.leftChild = newSubtree;
      }
      else { // search the right subtree
        newSubtree = deleteItem(tNode.rightChild, searchKey);
        tNode.rightChild = newSubtree;
      }  // end if
    }  // end if
    return tNode;
  }  // end deleteItem

  protected TreeNode<T> deleteNode(TreeNode<T> tNode) {
    // Algorithm note: There are four cases to consider:
    //   1. The tNode is a leaf.
    //   2. The tNode has no left child.
    //   3. The tNode has no right child.
    //   4. The tNode has two children.
    // Calls: findLeftmost and deleteLeftmost
    T replacementItem;

    // test for a leaf
    if ( (tNode.leftChild == null) &&
         (tNode.rightChild == null) ) {
      return null;
    }  // end if leaf

    // test for no left child
    else if (tNode.leftChild == null) {
      return tNode.rightChild;
    }  // end if no left child

    // test for no right child
    else if (tNode.rightChild == null) {
```

```
      return tNode.leftChild;
    }  // end if no right child

    // there are two children:
    // retrieve and delete the inorder successor
    else {
      replacementItem = findLeftmost(tNode.rightChild);
      tNode.item = replacementItem;
      tNode.rightChild = deleteLeftmost(tNode.rightChild);
      return tNode;
    }  // end if
  }  // end deleteNode

  protected T findLeftmost(TreeNode<T> tNode)  {
    if (tNode.leftChild == null) {
      return tNode.item;
    }
    else {
      return findLeftmost(tNode.leftChild);
    }  // end if
  }  // end findLeftmost

  protected TreeNode<T> deleteLeftmost(TreeNode<T> tNode) {
    if (tNode.leftChild == null) {
      return tNode.rightChild;
    }
    else {
      tNode.leftChild = deleteLeftmost(tNode.leftChild);
      return tNode;
    }  // end if
  }  // end deleteLeftmost

}  // end BinarySearchTree
```