Important implementation from the textbook:

11.1.3 Java Implementation

In Code Fragments 11.3 through 11.6 we define a TreeMap class that implements the sorted map ADT while using a binary search tree for storage. The TreeMap class is declared as a child of the AbstractSortedMap base class, thereby inheriting support for performing comparisons based upon a given (or default) Comparator, a nested MapEntry class for storing key-value pairs, and concrete implementations of methods keySet and values based upon the entrySet method, which we will provide. (See Figure 10.2 on page 406 for an overview of our entire map hierarchy.)

For representing the tree structure, our TreeMap class maintains an instance of a subclass of the LinkedBinaryTree class from Section 8.3.1. In this implementation, we choose to represent the search tree as a *proper* binary tree, with explicit leaf nodes in the binary tree as sentinels, and map entries stored only at internal nodes. (We leave the task of a more space-efficient implementation to Exercise P-11.55.)

The TreeSearch algorithm of Code Fragment 11.1 is implemented as a private recursive method, treeSearch(p, k). That method either returns a position with an entry equal to key k, or else the last position that is visited on the search path. The method is not only used for all of the primary map operations, get(k), put(k, v), and remove(k), but for most of the sorted map methods, as the final internal position visited during an unsuccessful search has either the greatest key less than k or the least key greater than k.

Finally, we note that our TreeMap class is designed so that it can be subclassed to implement various forms of *balanced* search trees. We discuss the balancing framework more thoroughly in Section 11.2, but there are two aspects of the design that impact the code presented in this section. First, our tree member is technically declared as an instance of a BalanceableBinaryTree class, which is a specialization of the LinkedBinaryTree class; however, we rely only on the inherited behaviors in this section. Second, our code is peppered with calls to presumed methods named rebalanceAccess, rebalanceInsert, and rebalanceDelete; these methods do not do anything in this class, but they serve as ***hooks*** that can later be customized.

We conclude with a brief guide to the organization of our code.

Code Fragment 11.3: Beginning of TreeMap class, including constructors, size method, and expandExternal and treeSearch utilities.

Code Fragment 11.4: Map operations get(k), put(k, v), and remove(k).

Code Fragment 11.5: Sorted map ADT methods lastEntry( ), floorEntry(k), and lowerEntry(k), and protected utility treeMax. Symmetric methods firstEntry( ), ceilingEntry(k), higherEntry(k), and treeMin are provided online.

Code Fragment 11.6: Support for producing an iteration of all entries (method entrySet of the map ADT), or of a selected range of entries (method subMap(k_1, k_2) of the sorted map ADT).

Important implementation from the textbook:


```
1  /** An implementation of a sorted map using a binary search tree. */
2  public class TreeMap<K,V> extends AbstractSortedMap<K,V> {
3    // To represent the underlying tree structure, we use a specialized subclass of the
4    // LinkedBinaryTree class that we name BalanceableBinaryTree (see Section 11.2).
5    protected BalanceableBinaryTree<K,V> tree = new BalanceableBinaryTree<>();
6
7    /** Constructs an empty map using the natural ordering of keys. */
8    public TreeMap() {
9      super();                                  // the AbstractSortedMap constructor
10     tree.addRoot(null);                       // create a sentinel leaf as root
11   }
12   /** Constructs an empty map using the given comparator to order keys. */
13   public TreeMap(Comparator<K> comp) {
14     super(comp);                              // the AbstractSortedMap constructor
15     tree.addRoot(null);                       // create a sentinel leaf as root
16   }
17   /** Returns the number of entries in the map. */
18   public int size() {
19     return (tree.size() - 1) / 2;             // only internal nodes have entries
20   }
21   /** Utility used when inserting a new entry at a leaf of the tree */
22   private void expandExternal(Position<Entry<K,V>> p, Entry<K,V> entry) {
23     tree.set(p, entry);                       // store new entry at p
24     tree.addLeft(p, null);                    // add new sentinel leaves as children
25     tree.addRight(p, null);
26   }
27
28   // Omitted from this code fragment, but included in the online version of the code,
29   // are a series of protected methods that provide notational shorthands to wrap
30   // operations on the underlying linked binary tree. For example, we support the
31   // protected syntax root() as shorthand for tree.root() with the following utility:
32   protected Position<Entry<K,V>> root() { return tree.root(); }
33
34   /** Returns the position in p's subtree having given key (or else the terminal leaf).*/
35   private Position<Entry<K,V>> treeSearch(Position<Entry<K,V>> p, K key) {
36     if (isExternal(p))
37       return p;                               // key not found; return the final leaf
38     int comp = compare(key, p.getElement());
39     if (comp == 0)
40       return p;                               // key found; return its position
41     else if (comp < 0)
42       return treeSearch(left(p), key);        // search left subtree
43     else
44       return treeSearch(right(p), key);       // search right subtree
45   }
```

Code Fragment 11.3: Beginning of a TreeMap class based on a binary search tree.

Important implementation from the textbook:


```
46   /** Returns the value associated with the specified key (or else null). */
47   public V get(K key) throws IllegalArgumentException {
48     checkKey(key);                              // may throw IllegalArgumentException
49     Position<Entry<K,V>> p = treeSearch(root(), key);
50     rebalanceAccess(p);                         // hook for balanced tree subclasses
51     if (isExternal(p)) return null;             // unsuccessful search
52     return p.getElement().getValue();           // match found
53   }
54   /** Associates the given value with the given key, returning any overridden value.*/
55   public V put(K key, V value) throws IllegalArgumentException {
56     checkKey(key);                              // may throw IllegalArgumentException
57     Entry<K,V> newEntry = new MapEntry<>(key, value);
58     Position<Entry<K,V>> p = treeSearch(root(), key);
59     if (isExternal(p)) {                        // key is new
Insertion in a tree map:
60       expandExternal(p, newEntry);
61       rebalanceInsert(p);                       // hook for balanced tree subclasses
62       return null;
63     } else {                                    // replacing existing key
64       V old = p.getElement().getValue();
65       set(p, newEntry);
66       rebalanceAccess(p);                       // hook for balanced tree subclasses
67       return old;
68     }
69   }
70   /** Removes the entry having key k (if any) and returns its associated value. */
71   public V remove(K key) throws IllegalArgumentException {
72     checkKey(key);                              // may throw IllegalArgumentException
73     Position<Entry<K,V>> p = treeSearch(root(), key);
Deletion in a tree map:
74     if (isExternal(p)) {                        // key not found
75       rebalanceAccess(p);                       // hook for balanced tree subclasses
76       return null;
77     } else {
78       V old = p.getElement().getValue();
79       if (isInternal(left(p)) && isInternal(right(p))) { // both children are internal
80         Position<Entry<K,V>> replacement = treeMax(left(p));
81         set(p, replacement.getElement());
82         p = replacement;
83       } // now p has at most one child that is an internal node
84       Position<Entry<K,V>> leaf = (isExternal(left(p)) ? left(p) : right(p));
85       Position<Entry<K,V>> sib = sibling(leaf);
86       remove(leaf);
87       remove(p);                                // sib is promoted in p's place
88       rebalanceDelete(sib);                     // hook for balanced tree subclasses
89       return old;
90     }
91   }
```

Code Fragment 11.4: Primary map operations for the TreeMap class.

Important implementation from the textbook:


```
92   /** Returns the position with the maximum key in subtree rooted at Position p. */
93   protected Position<Entry<K,V>> treeMax(Position<Entry<K,V>> p) {
94     Position<Entry<K,V>> walk = p;
95     while (isInternal(walk))
96       walk = right(walk);
97     return parent(walk);                    // we want the parent of the leaf
98   }
99   /** Returns the entry having the greatest key (or null if map is empty). */
100  public Entry<K,V> lastEntry( ) {
101    if (isEmpty( )) return null;
102    return treeMax(root( )).getElement( );
103  }
104  /** Returns the entry with greatest key less than or equal to given key (if any). */
105  public Entry<K,V> floorEntry(K key) throws IllegalArgumentException {
106    checkKey(key);                          // may throw IllegalArgumentException
107    Position<Entry<K,V>> p = treeSearch(root( ), key);
108    if (isInternal(p)) return p.getElement( );  // exact match
109    while (!isRoot(p)) {
110      if (p == right(parent(p)))
111        return parent(p).getElement( );     // parent has next lesser key
112      else
113        p = parent(p);
114    }
115    return null;                            // no such floor exists
116  }
117  /** Returns the entry with greatest key strictly less than given key (if any). */
118  public Entry<K,V> lowerEntry(K key) throws IllegalArgumentException {
119    checkKey(key);                          // may throw IllegalArgumentException
120    Position<Entry<K,V>> p = treeSearch(root( ), key);
121    if (isInternal(p) && isInternal(left(p)))
122      return treeMax(left(p)).getElement( );  // this is the predecessor to p
123    // otherwise, we had failed search, or match with no left child
124    while (!isRoot(p)) {
125      if (p == right(parent(p)))
126        return parent(p).getElement( );     // parent has next lesser key
127      else
128        p = parent(p);
129    }
130    return null;                            // no such lesser key exists
131  }
```

Code Fragment 11.5: A sample of the sorted map operations for the TreeMap class. The symmetrical utility, treeMin, and public methods firstEntry, ceilingEntry, and higherEntry are available online.

Important implementation from the textbook:


```
132   /** Returns an iterable collection of all key-value entries of the map. */
133   public Iterable<Entry<K,V>> entrySet( ) {
134     ArrayList<Entry<K,V>> buffer = new ArrayList<>(size( ));
135     for (Position<Entry<K,V>> p : tree.inorder( ))
136       if (isInternal(p)) buffer.add(p.getElement( ));
137     return buffer;
138   }
139   /** Returns an iterable of entries with keys in range [fromKey, toKey). */
140   public Iterable<Entry<K,V>> subMap(K fromKey, K toKey) {
141     ArrayList<Entry<K,V>> buffer = new ArrayList<>(size( ));
142     if (compare(fromKey, toKey) < 0)          // ensure that fromKey < toKey
143       subMapRecurse(fromKey, toKey, root( ), buffer);
144     return buffer;
145   }
146   private void subMapRecurse(K fromKey, K toKey, Position<Entry<K,V>> p,
147                              ArrayList<Entry<K,V>> buffer) {
148     if (isInternal(p))
149       if (compare(p.getElement( ), fromKey) < 0)
150         // p's key is less than fromKey, so any relevant entries are to the right
151         subMapRecurse(fromKey, toKey, right(p), buffer);
152       else {
153         subMapRecurse(fromKey, toKey, left(p), buffer); // first consider left subtree
154         if (compare(p.getElement( ), toKey) < 0) {      // p is within range
155           buffer.add(p.getElement( ));                  // so add it to buffer, and consider
156           subMapRecurse(fromKey, toKey, right(p), buffer); // right subtree as well
157         }
158       }
159   }
```

Code Fragment 11.6: TreeMap operations supporting iteration of the entire map, or a portion of the map with a given key range.

11.2.1 Java Framework for Balancing Search Trees

Our TreeMap class (introduced in Section 11.1.3) is a fully functional map implementation. However, the running time for its operations depend on the height of the tree, and in the worst-case, that height may be $O(n)$ for a map with n entries. Therefore, we have intentionally designed the TreeMap class in a way that allows it to be easily extended to provide more advanced tree-balancing strategies. In later sections of this chapter, we will implement subclasses AVLTreeMap, SplayTreeMap, and RBTreeMap. In this section, we describe three important forms of support that the TreeMap class offers these subclasses.

Hooks for Rebalancing Operations

Our implementation of the basic map operations in Section 11.1.3 includes strategic calls to three nonpublic methods that serve as ***hooks*** for rebalancing algorithms:

- A call to rebalanceInsert(p) is made from within the put method, after a new node is added to the tree at position p (line 61 of Code Fragment 11.4).
- A call to rebalanceDelete(p) is made from within the remove method, after a node is deleted from the tree (line 88 of Code Fragment 11.4); position p identifies the child of the removed node that was promoted in its place.
- A call to rebalanceAccess(p) is made by any call to get, put, or remove that does *not* result in a structural change. Position p, which could be internal or external, represents the deepest node of the tree that was accessed during the operation. This hook is specifically used by the ***splay tree*** structure (see Section 11.4) to restructure a tree so that more frequently accessed nodes are brought closer to the root.

Within our TreeMap class, we provide the trivial declarations of these three methods, having bodies that do nothing, as shown in Code Fragment 11.8. A subclass of TreeMap may override any of these methods to implement a nontrivial action to rebalance a tree. This is another example of the ***template method design pattern***, as originally discussed in Section 2.3.3.

```
protected void rebalanceInsert(Position<Entry<K,V>> p) { }
protected void rebalanceDelete(Position<Entry<K,V>> p) { }
protected void rebalanceAccess(Position<Entry<K,V>> p) { }
```

Code Fragment 11.8: Trivial definitions of TreeMap methods that serve as hooks for our rebalancing framework. These methods may be overridden by subclasses in order to perform appropriate rebalancing operations.

Important implementation details extracted from the textbook:

Protected Methods for Rotating and Restructuring

To support common restructuring operations, our TreeMap class relies on storing the tree as an instance of a new nested class, BalanceableBinaryTree (shown in Code Fragments 11.9 and 11.10). That class is a specialization of the original LinkedBinaryTree class from Section 8.3.1. This new class provides protected utility methods rotate and restructure that, respectively, implement a single rotation and a trinode restructuring (described at the beginning of Section 11.2). Although these methods are not invoked by the standard TreeMap operations, their inclusion supports greater code reuse, as they are available to all balanced-tree subclasses.

These methods are implemented in Code Fragment 11.10. To simplify the code, we define an additional relink utility that properly links parent and child nodes to each other. The focus of the rotate method then becomes redefining the relationship between the parent and child, relinking a rotated node directly to its original grandparent, and shifting the "middle" subtree (that labeled as T_2 in Figure 11.8) between the rotated nodes.

For the trinode restructuring, we determine whether to perform a single or double rotation, as originally described in Figure 11.9. The four cases in that figure demonstrate a downward path z to y to x that are respectively right-right, left-left, right-left, and left-right. The first two patterns, with matching orientation, warrant a single rotation moving y upward, while the last two patterns, with opposite orientations, warrant a double rotation moving x upward.

Specialized Nodes with an Auxiliary Data Member

Many tree-balancing strategies require that some form of auxiliary "balancing" information be stored at nodes of a tree. To ease the burden on the balanced-tree subclasses, we choose to add an auxiliary integer value to every node within the BalanceableSearchTree class. This is accomplished by defining a new BSTNode class, which itself inherits from the nested LinkedBinaryTree.Node class. The new class declares the auxiliary variable, and provides methods for getting and setting its value.

We draw attention to an important subtlety in our design, including that of the original LinkedBinaryTree subclass. Whenever a low-level operation on an underlying linked tree requires a new node, we must ensure that the correct type of node is created. That is, for our balanceable tree, we need each node to be a BTNode, which includes the auxiliary field. However, the creation of nodes occurs within low-level operations, such as addLeft and addRight, that reside in the original LinkedBinaryTree class.

Important implementation details extracted from the textbook:

We rely on a technique known as the ***factory method design pattern***. The LinkedBinaryTree class includes a protected method, createNode (originally given at lines 30–33 of Code Fragment 8.8), that is responsible for instantiating a new node of the appropriate type. The rest of the code in that class makes sure to always use the createNode method when a new node is needed.

In the LinkedBinaryTree class, the createNode method returns a simple Node instance. In our new BalanceableBinaryTree class, we override the createNode method (see lines 22–27 in Code Fragment 11.9), so that a new instance of the BSTNode class is returned. In this way, we effectively change the behavior of the low-level operations in the LinkedBinaryTree class so that it uses instances of our specialized node class, and therefore, that every node in our balanced trees includes support for the new auxiliary field.

```
1  /** A specialized version of LinkedBinaryTree with support for balancing. */
2  protected static class BalanceableBinaryTree<K,V>
3                          extends LinkedBinaryTree<Entry<K,V>> {
4    //-------------- nested BSTNode class --------------
5    // this extends the inherited LinkedBinaryTree.Node class
6    protected static class BSTNode<E> extends Node<E> {
7      int aux=0;
8      BSTNode(E e, Node<E> parent, Node<E> leftChild, Node<E> rightChild) {
9        super(e, parent, leftChild, rightChild);
10     }
11     public int getAux( ) { return aux; }
12     public void setAux(int value) { aux = value; }
13   } //--------- end of nested BSTNode class ---------
14
15   // positional-based methods related to aux field
16   public int getAux(Position<Entry<K,V>> p) {
17     return ((BSTNode<Entry<K,V>>) p).getAux( );
18   }
19   public void setAux(Position<Entry<K,V>> p, int value) {
20     ((BSTNode<Entry<K,V>>) p).setAux(value);
21   }
22   // Override node factory function to produce a BSTNode (rather than a Node)
23   protected
24   Node<Entry<K,V>> createNode(Entry<K,V> e, Node<Entry<K,V>> parent,
25                               Node<Entry<K,V>> left, Node<Entry<K,V>> right) {
26     return new BSTNode<>(e, parent, left, right);
27   }
```

Code Fragment 11.9: The BalanceableBinaryTree class, which is nested within the TreeMap class definition. (Continues in Code Fragment 11.10.)

Important implementation details extracted from the textbook:


```
28    /** Relinks a parent node with its oriented child node. */
29    private void relink(Node<Entry<K,V>> parent, Node<Entry<K,V>> child,
30                        boolean makeLeftChild) {
31      child.setParent(parent);
32      if (makeLeftChild)
33        parent.setLeft(child);
34      else
35        parent.setRight(child);
36    }
37    /** Rotates Position p above its parent. */
38    public void rotate(Position<Entry<K,V>> p) {
39      Node<Entry<K,V>> x = validate(p);
40      Node<Entry<K,V>> y = x.getParent( );          // we assume this exists
41      Node<Entry<K,V>> z = y.getParent( );          // grandparent (possibly null)
42      if (z == null) {
43        root = x;                                   // x becomes root of the tree
44        x.setParent(null);
45      } else
46        relink(z, x, y == z.getLeft( ));            // x becomes direct child of z
47      // now rotate x and y, including transfer of middle subtree
48      if (x == y.getLeft( )) {
49        relink(y, x.getRight( ), true);             // x's right child becomes y's left
50        relink(x, y, false);                        // y becomes x's right child
51      } else {
52        relink(y, x.getLeft( ), false);             // x's left child becomes y's right
53        relink(x, y, true);                         // y becomes left child of x
54      }
55    }
56    /** Performs a trinode restructuring of Position x with its parent/grandparent. */
57    public Position<Entry<K,V>> restructure(Position<Entry<K,V>> x) {
58      Position<Entry<K,V>> y = parent(x);
59      Position<Entry<K,V>> z = parent(y);
60      if ((x == right(y)) == (y == right(z))) {     // matching alignments
61        rotate(y);                                  // single rotation (of y)
62        return y;                                   // y is new subtree root
63      } else {                                      // opposite alignments
64        rotate(x);                                  // double rotation (of x)
65        rotate(x);
66        return x;                                   // x is new subtree root
67      }
68    }
69  }
```

Code Fragment 11.10: The BalanceableBinaryTree class, which is nested within the TreeMap class definition (continued from Code Fragment 11.9).

11.3.2 Java Implementation

A complete implementation of an AVLTreeMap class is provided in Code Fragments 11.11 and 11.12. It inherits from the standard TreeMap class and relies on the balancing framework described in Section 11.2.1. We highlight two important aspects of our implementation. First, the AVLTreeMap uses the node's auxiliary balancing variable to store the height of the subtree rooted at that node, with leaves having a balance factor of 0 by default. We also provide several utilities involving heights of nodes (see Code Fragment 11.11).

To implement the core logic of the AVL balancing strategy, we define a utility, named rebalance, that suffices to restore the height-balance property after an insertion or a deletion (see Code Fragment 11.11). Although the inherited behaviors for insertion and deletion are quite different, the necessary post-processing for an AVL tree can be unified. In both cases, we trace an upward path from the position p at which the change took place, recalculating the height of each position based on the (updated) heights of its children. We perform a trinode restructuring operation if an imbalanced position is reached. The upward march from p continues until we reach an ancestor with height that was unchanged by the map operation, or with height that was restored to its previous value by a trinode restructuring operation, or until reaching the root of the tree (in which case the overall height of the tree has increased by one). To easily detect the stopping condition, we record the "old" height of a position, as it existed before the insertion or deletion operation begin, and compare that to the newly calculated height after a possible restructuring.

```
1  /** An implementation of a sorted map using an AVL tree. */
2  public class AVLTreeMap<K,V> extends TreeMap<K,V> {
3    /** Constructs an empty map using the natural ordering of keys. */
4    public AVLTreeMap() { super(); }
5    /** Constructs an empty map using the given comparator to order keys. */
6    public AVLTreeMap(Comparator<K> comp) { super(comp); }
7    /** Returns the height of the given tree position. */
8    protected int height(Position<Entry<K,V>> p) {
9      return tree.getAux(p);
10   }
11   /** Recomputes the height of the given position based on its children's heights. */
12   protected void recomputeHeight(Position<Entry<K,V>> p) {
13     tree.setAux(p, 1 + Math.max(height(left(p)), height(right(p))));
14   }
15   /** Returns whether a position has balance factor between −1 and 1 inclusive. */
16   protected boolean isBalanced(Position<Entry<K,V>> p) {
17     return Math.abs(height(left(p)) − height(right(p))) <= 1;
18   }
```

Code Fragment 11.11: AVLTreeMap class. (Continues in Code Fragment 11.12.)

```
19    /** Returns a child of p with height no smaller than that of the other child. */
20    protected Position<Entry<K,V>> tallerChild(Position<Entry<K,V>> p) {
21      if (height(left(p)) > height(right(p))) return left(p);          // clear winner
22      if (height(left(p)) < height(right(p))) return right(p);         // clear winner
23      // equal height children; break tie while matching parent's orientation
24      if (isRoot(p)) return left(p);                                   // choice is irrelevant
25      if (p == left(parent(p))) return left(p);                        // return aligned child
26      else return right(p);
27    }
28    /**
29     * Utility used to rebalance after an insert or removal operation. This traverses the
30     * path upward from p, performing a trinode restructuring when imbalance is found,
31     * continuing until balance is restored.
32     */
33    protected void rebalance(Position<Entry<K,V>> p) {
34      int oldHeight, newHeight;
35      do {
36        oldHeight = height(p);                                         // not yet recalculated if internal
37        if (!isBalanced(p)) {                                          // imbalance detected
38          // perform trinode restructuring, setting p to resulting root,
39          // and recompute new local heights after the restructuring
40          p = restructure(tallerChild(tallerChild(p)));
41          recomputeHeight(left(p));
42          recomputeHeight(right(p));
43        }
44        recomputeHeight(p);
45        newHeight = height(p);
46        p = parent(p);
47      } while (oldHeight != newHeight && p != null);
48    }
49    /** Overrides the TreeMap rebalancing hook that is called after an insertion. */
50    protected void rebalanceInsert(Position<Entry<K,V>> p) {
51      rebalance(p);
52    }
53    /** Overrides the TreeMap rebalancing hook that is called after a deletion. */
54    protected void rebalanceDelete(Position<Entry<K,V>> p) {
55      if (!isRoot(p))
56        rebalance(parent(p));
57    }
58  }
```

Code Fragment 11.12: AVLTreeMap class (continued from Code Fragment 11.11).