Chapter 14

Search Trees

Concepts:
- ▷ `BinarySearchTrees`
- ▷ Tree Sort
- ▷ Splay Trees
- ▷ Red-Black Trees

He looked to the right of him.
No caps.
He looked to the left of him.
No caps.
...
Then he looked up into the tree.
And what do you think he saw?
—Esphyr Slobodkina

STRUCTURES ARE OFTEN THE SUBJECT OF A SEARCH. We have seen, for example, that binary search is a natural and efficient algorithm for finding values within ordered, randomly accessible structures. Recall that at each point the algorithm compares the value sought with the value in the middle of the structure. If they are not equal, the algorithm performs a similar, possibly recursive search on one side or the other. The pivotal feature of the algorithm, of course, was that the underlying structure was in order. The result was that a value could be efficiently *found* in approximately logarithmic time. Unfortunately, the modifying operations—`add` and `remove`—had complexities that were determined by the linear nature of the vector.

Heaps have shown us that by relaxing our notions of order we can improve on the linear complexities of adding and removing values. These logarithmic operations, however, do not preserve the order of elements in any obviously useful manner. Still, if we were somehow able to totally order the elements of a binary tree, then an algorithm like binary search might naturally be imposed on this branching structure.

14.1 Binary Search Trees

The *binary search tree* is a binary tree whose elements are kept in order. This is easily stated as a recursive definition.

Definition 14.1 *A binary tree is a* binary search tree *if it is trivial, or if every node is simultaneously greater than or equal to each value in its left subtree, and less than or equal to each value in its right subtree.*

To see that this is a significant restriction on the structure of a binary tree, one need only note that if a maximum of n distinct values is found at the root,

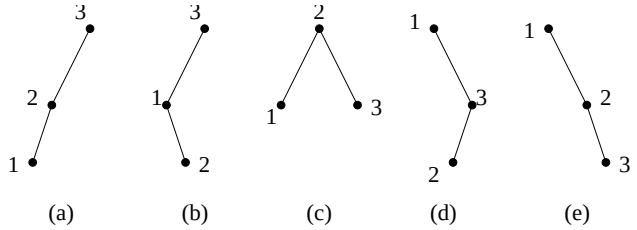
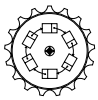


Figure 14.1 Binary search trees with three nodes.

all other values must be found in the left subtree. Figure 14.1 demonstrates the many trees that can contain even three distinct values. Thus, if one is not too picky about the value one wants to have at the root of the tree, there are still a significant number of trees from which to choose. This is important if we want to have our modifying operations run quickly: ideally we should be as nonrestrictive about the outcome as possible, in order to reduce the friction of the operation.

One important thing to watch for is that even though our definition allows duplicate values of the root node to fall on either side, our code will prefer to have them on the left. This preference is arbitrary. If we assume that values equal to the root will be found in the left subtree, but in actuality some are located in the right, then we might expect inconsistent behavior from methods that search for these values. In fact, this is not the case.

To guide access to the elements of a binary search tree, we will consider it an implementation of an `OrderedStructure`, supporting the following methods:



Binary-
SearchTree

```
public class BinarySearchTree<E extends Comparable<E>>
    extends AbstractStructure<E> implements OrderedStructure<E>
{
    public BinarySearchTree()
    // post: constructs an empty binary search tree

    public BinarySearchTree(Comparator<E> alternateOrder)
    // post: constructs an empty binary search tree

    public boolean isEmpty()
    // post: returns true iff the binary search tree is empty

    public void clear()
    // post: removes all elements from binary search tree

    public int size()
    // post: returns the number of elements in binary search tree

    public void add(E value)
```

```
        // post: adds a value to binary search tree

        public boolean contains(E value)
        // post: returns true iff val is a value found within the tree

        public E get(E value)
        // post: returns object found in tree, or null

        public E remove(E value)
        // post: removes one instance of val, if found

        public Iterator<E> iterator()
        // post: returns iterator to traverse BST
    }
```

Unlike the `BinaryTree`, the `BinarySearchTree` provides only one `iterator` method. This method provides for an in-order traversal of the tree, which, with some thought, allows access to each of the elements in order.

Maybe even with no thought!

14.2 Example: Tree Sort

Because the `BinarySearchTree` is an `OrderedStructure` it provides the natural basis for sorting. The algorithm of Section 11.2.3 will work equally well here, provided the allocation of the `OrderedStructure` is modified to construct a `BinarySearchTree`. The binary search structure, however, potentially provides significant improvements in performance. If the tree can be kept reasonably short, the cost of inserting each element is $O(\log n)$. Since n elements are ultimately added to the structure, the total cost is $O(n \log n)$.[1] As we have seen in Chapter 12, all the elements of the underlying binary tree can be visited in linear time. The resulting algorithm has a potential for $O(n \log n)$ time complexity, which rivals the performance of sorting techniques using heaps. The advantage of binary search trees is that the elements need not be removed to determine their order. To attain this performance, though, we must keep the tree as short as possible. This will require considerable attention.

14.3 Example: Associative Structures

Associative structures play an important role in making algorithms efficient. In these data structures, *values* are associated with *keys*. Typically (though not necessarily), the keys are unique and aid in the retrieval of more complete information—the value. In a `Vector`, for example, we use integers as indices to find values. In an `AssociativeVector` we can use any type of *object*. The

[1] This needs to be proved! See Problem 14.11.

SymbolTable associated with the `PostScript` lab (Section 10.5) is, essentially, an associative structure.

Associative structures are an important feature of many symbol-based systems. Here, for example, is a first approach to the construction of a general-purpose symbol table, with potentially logarithmic performance:

SymTab

```
import structure5.*;
import java.util.Iterator;
import java.util.Scanner;
public class SymTab<S extends Comparable<S>,T>
{
    protected BinarySearchTree<ComparableAssociation<S,T>> table;
    public SymTab()
    // post: constructs empty symbol table
    {
        table = new BinarySearchTree<ComparableAssociation<S,T>>();
    }

    public boolean contains(S symbol)
    // pre: symbol is non-null string
    // post: returns true iff string in table
    {
        ComparableAssociation<S,T> a =
            new ComparableAssociation<S,T>(symbol,null);
        return table.contains(a);
    }

    public void add(S symbol, T value)
    // pre: symbol non-null
    // post: adds/replaces symbol-value pair in table
    {
        ComparableAssociation<S,T> a =
            new ComparableAssociation<S,T>(symbol,value);
        if (table.contains(a)) table.remove(a);
        table.add(a);
    }

    public T get(S symbol)
    // pre: symbol non null
    // post: returns token associated with symbol
    {
        ComparableAssociation<S,T> a =
            new ComparableAssociation<S,T>(symbol,null);
        if (table.contains(a)) {
            a = table.get(a);
            return a.getValue();
        } else {
            return null;
        }
    }
```

```
    public T remove(S symbol)
    // pre: symbol non null
    // post: removes value associated with symbol and returns it
    //       if error returns null
    {
        ComparableAssociation<S,T> a =
            new ComparableAssociation<S,T>(symbol,null);
        if (table.contains(a)) {
            a = table.remove(a);
            return a.getValue();
        } else {
            return null;
        }
    }
}
```

Based on such a table, we might have a program that reads in a number of alias-name pairs terminated by the word `END`. After that point, the program prints out the fully translated aliases:

```
public static void main(String args[])
{
    SymTab<String,String> table = new SymTab<String,String>();
    Scanner s = new Scanner(System.in);
    String alias, name;
    // read in the alias-name database
    do
    {
        alias = s.next();
        if (!alias.equals("END"))
        {
            name = s.next();
            table.add(alias,name);
        }
    } while (!alias.equals("END"));
    // enter the alias translation stage
    do
    {
        name = s.next();
        while (table.contains(name))
        {
            // translate alias
            name = table.get(name);
        }
        System.out.println(name);
    } while (s.hasNext());
}
```

Given the input:

```
three 3
one unity
unity 1
pi three
END

one
two
three
pi
```

the program generates the following output:

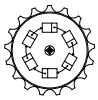
```
1
two
3
3
```

We will consider general associative structures in Chapter 15, when we discuss `Dictionaries`. We now consider the details of actually supporting the `BinarySearchTree` structure.

14.4 Implementation

In considering the implementation of a `BinarySearchTree`, it is important to remember that we are implementing an `OrderedStructure`. The methods of the `OrderedStructure` accept and return values that are to be compared with one another. By default, we assume that the data are `Comparable` and that the natural order suggested by the `NaturalComparator` is sufficient. If alternative orders are necessary, or an ordering is to be enforced on elements that do not directly implement a `compareTo` method, alternative `Comparators` may be used. Essentially the only methods that we depend upon are the compatibility of the `Comparator` and the elements of the tree.

We begin by noticing that a `BinarySearchTree` is little more than a binary tree with an imposed order. We maintain a reference to a `BinaryTree` and explicitly keep track of its size. The constructor need only initialize these two fields and suggest an ordering of the elements to implement a state consistent with an empty binary search tree:



BinarySearch-Tree

```
protected BinaryTree<E> root;

protected final BinaryTree<E> EMPTY = new BinaryTree<E>();

protected int count;
protected Comparator<E> ordering;


public BinarySearchTree()
```

```
// post: constructs an empty binary search tree
{
    this(new NaturalComparator<E>());
}

public BinarySearchTree(Comparator<E> alternateOrder)
// post: constructs an empty binary search tree
{
    root = EMPTY;
    count = 0;
    ordering = alternateOrder;
}
```

As with most implementations of `OrderedStructures`, we develop a method to find the correct location to insert the value and then use that method as the basis for implementing the public methods—`add`, `contains`, and `remove`. Our approach to the method `locate` is to have it return a reference to the location that identifies the correct point of insertion for the new value. This method, of course, makes heavy use of the `ordering`. Here is the Java code for the method:

```
protected BinaryTree<E> locate(BinaryTree<E> root, E value)
// pre: root and value are non-null
// post: returned: 1 - existing tree node with the desired value, or
//                 2 - the node to which value should be added
{
    E rootValue = root.value();
    BinaryTree<E> child;

    // found at root: done
    if (rootValue.equals(value)) return root;
    // look left if less-than, right if greater-than
    if (ordering.compare(rootValue,value) < 0)
    {
        child = root.right();
    } else {
        child = root.left();
    }
    // no child there: not in tree, return this node,
    // else keep searching
    if (child.isEmpty()) {
        return root;
    } else {
        return locate(child, value);
    }
}
```

The approach of the `locate` method parallels binary search. Comparisons are made with the `root`, which serves as a median value. If the value does not match, then the search is refocused on either the left side of the tree (among smaller values) or the right side of the tree (among larger values). In either

case, if the search is about to step off the tree, the current node is returned: if the value were added, it would be a child of the current node.

Once the `locate` method is written, the `contains` method must check to see if the node returned by `locate` actually equals the desired value:[2]

```
public boolean contains(E value)
// post: returns true iff val is a value found within the tree
{
    if (root.isEmpty()) return false;

    BinaryTree<E> possibleLocation = locate(root,value);
    return value.equals(possibleLocation.value());
}
```

It now becomes a fairly straightforward task to add a value. We simply locate the value in the tree using the `locate` function. If the value was not found, `locate` returned a node off of which a leaf with the desired value may be added. If, however, `locate` has found an equivalent value, we must insert the new value as the right child of the predecessor of the node returned by `locate`.[3]

```
public void add(E value)
// post: adds a value to binary search tree
{
    BinaryTree<E> newNode = new BinaryTree<E>(value,EMPTY,EMPTY);

    // add value to binary search tree
    // if there's no root, create value at root
    if (root.isEmpty())
    {
        root = newNode;
    } else {
        BinaryTree<E> insertLocation = locate(root,value);
        E nodeValue = insertLocation.value();
        // The location returned is the successor or predecessor
        // of the to-be-inserted value
        if (ordering.compare(nodeValue,value) < 0) {
            insertLocation.setRight(newNode);
        } else {
            if (!insertLocation.left().isEmpty()) {
                // if value is in tree, we insert just before
                predecessor(insertLocation).setRight(newNode);
            } else {
```

[2] We reemphasize at this point the importance of making sure that the `equals` method for an object is consistent with the ordering suggested by the `compare` method of the particular `Comparator`.

[3] With a little thought, it is clear to see that this is a correct location. If there are two copies of a value in a tree, the second value added is a descendant and predecessor (in an in-order traversal) of the located value. It is also easy to see that a predecessor has no right child, and that if one is added, *it* becomes the predecessor.

```
                insertLocation.setLeft(newNode);
            }
        }
    }
    count++;
}
```

Our `add` code makes use of the protected "helper" function, `predecessor`, which returns a pointer to the node that immediately precedes the indicated root:

```
protected BinaryTree<E> predecessor(BinaryTree<E> root)
{
    Assert.pre(!root.isEmpty(), "No predecessor to middle value.");
    Assert.pre(!root.left().isEmpty(), "Root has left child.");
    BinaryTree<E> result = root.left();
    while (!result.right().isEmpty()) {
        result = result.right();
    }
    return result;
}
```

A similar routine can be written for successor, and would be used if we preferred to store duplicate values in the right subtree.

We now approach the problem of removing a value from a binary search tree. Observe that if it is found, it might be an internal node. The worst case occurs when the root of a tree is involved, so let us consider that problem.

There are several cases. First (Figure 14.2a), if the root of a tree has no left child, the right subtree can be used as the resulting tree. Likewise (Figure 14.2b), if there is no right child, we simply return the left. A third case (Figure 14.2c) occurs when the left subtree has no right child. Then, the right subtree—a tree with values no smaller than the left root—is made the right subtree of the left. The left root is returned as the result. The opposite circumstance could also be true.

We are, then, left to consider trees with a left subtree that, in turn, contains a right subtree (Figure 14.3). Our approach to solving this case is to seek out the predecessor of the root and make it the new root. Note that even though the predecessor does not have a right subtree, it may have a left. This subtree can take the place of the predecessor as the right subtree of a nonroot node. (Note that this is the result that we would expect if we had recursively performed our node-removing process on the subtree rooted at the predecessor.)

Finally, here is the Java code that removes the top `BinaryTree` of a tree and returns the root of the resulting tree:

```
protected BinaryTree<E> removeTop(BinaryTree<E> topNode)
// pre: topNode contains the value we want to remove
// post: we return an binary tree rooted with the predecessor of topnode.
{
    // remove topmost BinaryTree from a binary search tree
```

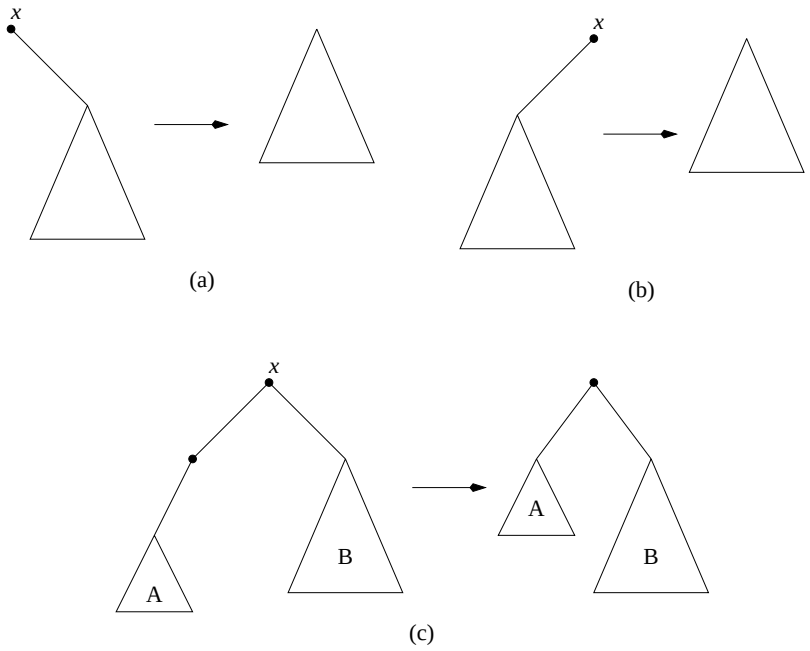


Figure 14.2 The three simple cases of removing a root value from a tree.

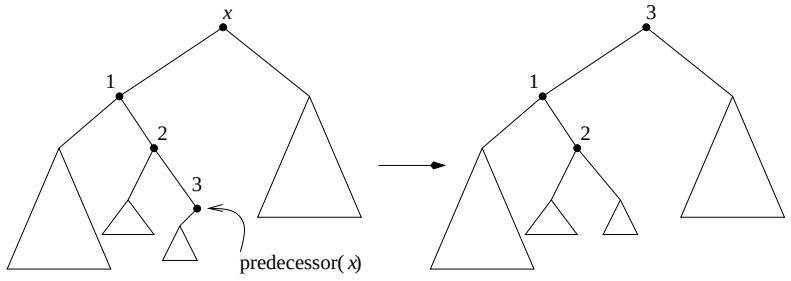


Figure 14.3 Removing the root of a tree with a rightmost left descendant.

```
        BinaryTree<E> left  = topNode.left();
        BinaryTree<E> right = topNode.right();
        // disconnect top node
        topNode.setLeft(EMPTY);
        topNode.setRight(EMPTY);
        // Case a, no left BinaryTree
        //   easy: right subtree is new tree
        if (left.isEmpty()) { return right; }
        // Case b, no right BinaryTree
        //   easy: left subtree is new tree
        if (right.isEmpty()) { return left; }
        // Case c, left node has no right subtree
        //   easy: make right subtree of left
        BinaryTree<E> predecessor = left.right();
        if (predecessor.isEmpty())
        {
            left.setRight(right);
            return left;
        }
        // General case, slide down left tree
        //   harder: successor of root becomes new root
        //           parent always points to parent of predecessor
        BinaryTree<E> parent = left;
        while (!predecessor.right().isEmpty())
        {
            parent = predecessor;
            predecessor = predecessor.right();
        }
        // Assert: predecessor is predecessor of root
        parent.setRight(predecessor.left());
        predecessor.setLeft(left);
        predecessor.setRight(right);
        return predecessor;
    }
```

With the combined efforts of the `removeTop` and `locate` methods, we can now simply locate a value in the search tree and, if found, remove it from the tree. We must be careful to update the appropriate references to rehook the modified subtree back into the overall structure.

Notice that inserting and removing elements in this manner ensures that the in-order traversal of the underlying tree delivers the values stored in the nodes in a manner that respects the necessary ordering. We use this, then, as our preferred iteration method.

```
    public Iterator<E> iterator()
    // post: returns iterator to traverse BST
    {
        return root.inorderIterator();
    }
```

The remaining methods (`size`, etc.) are implemented in a now-familiar manner.

Exercise 14.1 *One possible approach to keeping duplicate values in a binary search tree is to keep a list of the values in a single node. In such an implementation, each element of the list must appear externally as a separate node. Modify the* `BinarySearchTree` *implementation to make use of these lists of duplicate values.*

Each of the time-consuming operations of a `BinarySearchTree` has a worst-case time complexity that is proportional to the height of the tree. It is easy to see that checking for or adding a leaf, or removing a root, involves some of the most time-consuming operations. Thus, for logarithmic behavior, we must be sure that the tree remains as short as possible.

Unfortunately, we have no such assurance. In particular, one may observe what happens when values are inserted in descending order: the tree is heavily skewed to the left. If the same values are inserted in ascending order, the tree can be skewed to the right. If these values are distinct, the tree becomes, essentially, a singly linked list. Because of this behavior, we are usually better off if we shuffle the values beforehand. This causes the tree to become, on average, shorter and more balanced, and causes the expected insertion time to become $O(\log n)$.

Considering that the tree is responsible for maintaining an order among data values, it seems unreasonable to spend time shuffling values before ordering them. In Section 14.5 we find out that the process of adding and removing a node can be modified to maintain the tree in a relatively balanced state, with only a little overhead.

14.5 Splay Trees

Because the process of adding a new value to a binary search tree is *deterministic*—it produces the same result tree each time—and because inspection of the tree does not modify its structure, one is stuck with the performance of any degenerate tree constructed. What might work better would be to allow the tree to reconfigure itself when operations appear to be inefficient.

Splay: to spread outward.

The *splay tree* quickly overcomes poor performance by rearranging the tree's nodes on the fly using a simple operation called a *splay*. Instead of performing careful analysis and optimally modifying the structure whenever a node is added or removed, the splay tree simply moves the referenced node to the top of the tree. The operation has the interesting characteristic that the average depth of the ancestors of the node to be splayed is approximately halved. As with skew heaps, the performance of a splay tree's operators, when amortized over many operations, is logarithmic.

The basis for the splay operation is a pair of operations called *rotations* (see Figure 14.4). Each of these rotations replaces the root of a subtree with one of its children. A right rotation takes a left child, x, of a node y and reverses their relationship. This induces certain obvious changes in connectivity of subtrees,

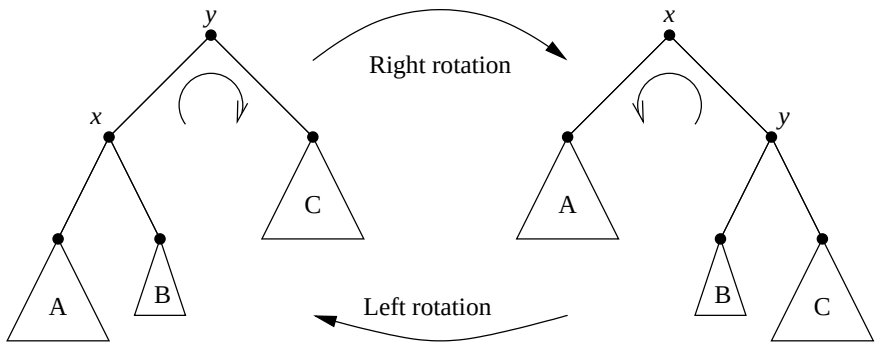
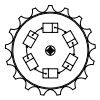


Figure 14.4 The relation between rotated subtrees.

but in all other ways, the tree remains the same. In particular, there is no structural effect on the tree above the original location of node y. A left rotation is precisely the opposite of a right rotation; these operations are inverses of each other.

The code for rotating a binary tree about a node is a method of the `BinaryTree` class. We show, here, `rotateRight`; a similar method performs a left rotation.

Finally, a right handed method!



`BinaryTree-Node`

```
protected void rotateRight()
// pre: this node has a left subtree
// post: rotates local portion of tree so left child is root
{
    BinaryTree<E> parent = parent();
    BinaryTree<E> newRoot = left();
    boolean wasChild = parent != null;
    boolean wasLeftChild = isLeftChild();

    // hook in new root (sets newRoot's parent, as well)
    setLeft(newRoot.right());

    // puts pivot below it (sets this's parent, as well)
    newRoot.setRight(this);

    if (wasChild) {
        if (wasLeftChild) parent.setLeft(newRoot);
        else              parent.setRight(newRoot);
    }
}
```

For each rotation accomplished, the nonroot node moves upward by one level. Making use of this fact, we can now develop an operation to splay a tree at a particular node. It works as follows:

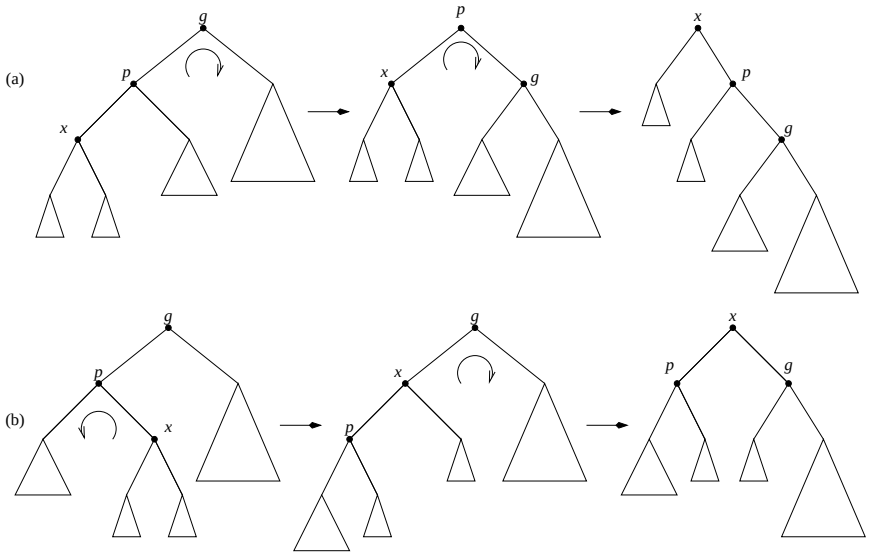


Figure 14.5 Two of the rotation pairs used in the splaying operation. The other cases are mirror images of those shown here.

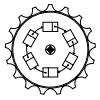
- If x is the root, we are done.
- If x is a left (or right) child of the root, rotate the tree to the right (or left) about the root. x becomes the root and we are done.
- If x is the left child of its parent p, which is, in turn, the left child of its grandparent g, rotate right about g, followed by a right rotation about p (Figure 14.5a). A symmetric pair of rotations is possible if x is a left child of a left child. After double rotation, continue splay of tree at x with this new tree.
- If x is the right child of p, which is the left child of g, we rotate left about p, then right about g (Figure 14.5b). The method is similar if x is the left child of a right child. Again, continue the splay at x in the new tree.

After the splay has been completed, the node x is located at the root of the tree. If node x were to be immediately accessed again (a strong possibility), the tree is clearly optimized to handle this situation. It is *not* the case that the tree becomes more balanced (see Figure 14.5a). Clearly, if the tree is splayed at an extremal value, the tree is likely to be extremely unbalanced. An interesting feature, however, is that the depth of the nodes on the original path from x to the root of the tree is, on average, halved. Since the average depth of these

nodes is halved, they clearly occupy locations closer to the top of the tree where they may be more efficiently accessed.

To guarantee that the splay has an effect on all operations, we simply perform each of the binary search tree operations as before, but we splay the tree at the node accessed or modified during the operation. In the case of `remove`, we splay the tree at the parent of the value removed.

14.6 Splay Tree Implementation



SplayTree

Because the splay tree supports the binary search tree interface, we extend the `BinarySearchTree` data structure. Methods written for the `SplayTree` hide or *override* existing code inherited from the `BinarySearchTree`.

```
public class SplayTree<E extends Comparable<E>>
    extends BinarySearchTree<E> implements OrderedStructure<E>
{
    public SplayTree()
    // post: construct a new splay tree

    public SplayTree(Comparator<E> alternateOrder)
    // post: construct a new splay tree

    public void add(E val)
    // post: adds a value to the binary search tree

    public boolean contains(E val)
    // post: returns true iff val is a value found within the tree

    public E get(E val)
    // post: returns object found in tree, or null

    public E remove(E val)
    // post: removes one instance of val, if found

    protected void splay(BinaryTree<E> splayedNode)

    public Iterator<E> iterator()
    // post: returns iterator that traverses tree nodes in order
}
```

As an example of how the splay operation is incorporated into the existing binary tree code, we look at the `contains` method. Here, the root is reset to the value of the node to be splayed, and the splay operation is performed on the tree. The postcondition of the splay operation guarantees that the splayed node will become the root of the tree, so the entire operation leaves the tree in the correct state.

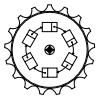
```
public boolean contains(E val)
// post: returns true iff val is a value found within the tree
{
    if (root.isEmpty()) return false;

    BinaryTree<E> possibleLocation = locate(root,val);
    if (val.equals(possibleLocation.value())) {
        splay(root = possibleLocation);
        return true;
    } else {
        return false;
    }
}
```

One difficulty with the splay operation is that it potentially modifies the structure of the tree. For example, the `contains` method—a method normally considered nondestructive—potentially changes the underlying topology of the tree. This makes it difficult to construct iterators that traverse the `SplayTree` since the user may use the value found from the iterator in a read-only operation that inadvertently modifies the structure of the splay tree. This *can* have disastrous effects on the state of the iterator. A way around this difficulty is to have the iterator keep only that state information that is necessary to help reconstruct—with help from the structure of the tree—the complete state of our traditional nonsplay iterator. In the case of the `SplayTreeIterator`, we keep track of two references: a reference to an "example" node of the tree and a reference to the current node inspected by the iterator. The example node helps recompute the root whenever the iterator is reset. To determine what nodes would have been stored in the stack in the traditional iterator—the stack of unvisited ancestors of the current node—we consider each node on the (unique) path from the root to the current node. Any node whose left child is also on the path is an element of our "virtual stack." In addition, the top of the stack maintains the current node (see Figure 14.6).

It can also wreck your day.

The constructor sets the appropriate underlying references and resets the iterator into its initial state. Because the `SplayTree` is dynamically restructuring, the root value passed to the constructor may not always be the root of the tree. Still, one can easily find the root of the current tree, given a node: follow parent pointers until one is `null`. Since the first value visited in an inorder traversal is the leftmost descendant, the reset method travels down the leftmost branch (logically pushing values on the stack) until it finds a node with no left child.



`SplayTree-Iterator`

```
protected BinaryTree<E> tree; // node of splay tree, root computed
protected final BinaryTree<E> LEAF;
protected BinaryTree<E> current; // current node
// In this iterator, the "stack" normally used is implied by
// looking back up the path from the current node.  Those nodes
// for which the path goes left are on the stack

public SplayTreeIterator(BinaryTree<E> root, BinaryTree<E> leaf)
// pre: root is the root of the tree to be traversed
```

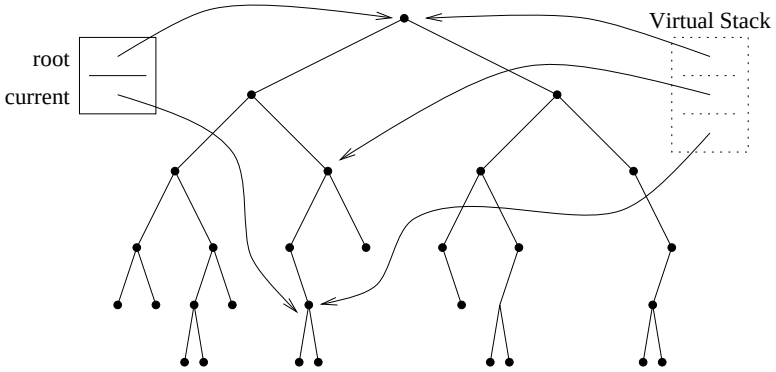


Figure 14.6 A splay tree iterator, the tree it references, and the contents of the virtual stack driving the iterator.

```
// post: constructs a new iterator to traverse splay tree
{
    tree = root;
    LEAF = leaf;
    reset();
}

public void reset()
// post: resets iterator to smallest node in tree
{
    current = tree;
    if (!current.isEmpty()) {
        current = current.root();
        while (!current.left().isEmpty()) current = current.left();
    }
}
```

The current node points to, by definition, an unvisited node that is, logically, on the top of the outstanding node stack. Therefore, the `hasNext` and `get` methods may access the current value immediately.

```
public boolean hasNext()
// post: returns true if there are unvisited nodes
{
    return !current.isEmpty();
}

public E get()
// pre: hasNext()
// post: returns current value
```

```
{
    return current.value();
}
```

All that remains is to move the iterator from one state to the next. The `next` method first checks to see if the current (just visited) element has a right child. If so, `current` is set to the leftmost descendant of the right child, effectively popping off the current node and pushing on all the nodes physically linking the current node and its successor. When no right descendant exists, the subtree rooted at the current node has been completely visited. The next node to be visited is the node under the top element of the virtual stack—the closest ancestor whose left child is also an ancestor of the current node. Here is how we accomplish this in Java:

```
public E next()
// pre: hasNext()
// post: returns current element and increments iterator
{
    E result = current.value();
    if (!current.right().isEmpty()) {
        current = current.right();
        while (!current.left().isEmpty())
        {
            current = current.left();
        }
    } else {
        // we're finished with current's subtree.  We now pop off
        // nodes until we come to the parent of a leftchild ancestor
        // of current
        boolean lefty;
        do
        {
            lefty = current.isLeftChild();
            current = current.parent();
        } while (current != null && !lefty);
        if (current == null) current = new BinaryTree<E>();
    }
    return result;
}
```

The iterator is now able to maintain its position through splay operations.

Again, the behavior of the splay tree is logarithmic when amortized over a number of operations. Any particular operation may take more time to execute, but the time is usefully spent rearranging nodes in a way that tends to make the tree shorter.

From a practical standpoint, the overhead of splaying the tree on every operation may be hard to justify if the operations performed on the tree are relatively random. On the other hand, if the access patterns tend to generate degenerate binary search trees, the splay tree can improve performance.

14.7 An Alternative: Red-Black Trees

A potential issue with both traditional binary search trees and splay trees is the fact that they potentially have bad performance if values are inserted or accessed in a particular order. Splay trees, of course, work hard to make sure that repeated accesses (which seem likely) will be efficient. Still, there is no absolute performance guarantee.

One could, of course, make sure that the values in a tree are stored in as perfectly balanced a manner as possible. In general, however, such techniques are both difficult to implement and costly in terms of per-operation execution time.

Exercise 14.2 *Describe a strategy for keeping a binary search tree as short as possible. One example might be to unload all of the values and to reinsert them in a particular order. How long does your approach take to* `add` *a value?*

Because we consider the performance of structures using big-O notation, we implicitly suggest we might be happy with performance that is within a constant of optimal. For example, we might be happy if we could keep a tree balanced within a factor of 2. One approach is to develop a structure called a *red-black tree*.

For accounting purposes only, the nodes of a red-black tree are imagined to be colored red or black. Along with these colors are several simple rules that are constantly enforced:

1. Every red node has two black children.
2. Every leaf has two black (`EMPTY` is considered black) children.
3. Every path from a node to a descendent leaf contains the same number of black nodes.

The result of constructing trees with these rules is that the height of the tree measured along two different paths cannot differ by more than a factor of 2: two red nodes may not appear contiguously, and every path must have the same number of black nodes. This would imply that the height of the tree is $O(\log_2 n)$.

Exercise 14.3 *Prove that the height of the tree with n nodes is no worse than $O(\log_2 n)$.*

Of course, the purpose of data abstraction is to be able to maintain the consistency of the structure—in this case, the red-black tree rules—as the structure is probed and modified. The methods `add` and `remove` are careful to maintain the red-black structure through at most $O(\log n)$ rotations and re-colorings of nodes. For example, if a node that is colored black is removed from the tree, it is necessary to perform rotations that either convert a red node on the path to the root to black, or reduce the *black height* (the number of black nodes from

root to leaf) of the entire tree. Similar problems can occur when we attempt to add a new node that must be colored black.

The code for red-black trees can be found online as `RedBlackTree`. While the code is too tedious to present here, it is quite elegant and leads to binary search trees with very good performance characteristics.

The implementation of the `RedBlackTree` structure in the `structure` package demonstrates another approach to packaging a binary search tree that *is* important to discuss. Like the `BinaryTree` structure, the `RedBlackTree` is defined as a recursive structure represented by a single node. The `RedBlackTree` also contains a dummy-node representation of the `EMPTY` tree. This is useful in reducing the complexity of the tests within the code, and it supports the notion that leaves have children with color, but most importantly, it allows the user to call `static` methods that are defined even for red-black trees with no nodes. This approach—coding inherently recursive structures as recursive classes—leads to *side-effect free* code. Each method has an effect on the tree at hand but does not modify any global structures. This means that the user must be very careful to record any side effects that might occur. In particular, it is important that methods that cause modifications to the structure return the "new" value of the tree. If, for example, the root of the tree was the object of a `remove`, that reference is no longer useful in maintaining contact with the tree.

To compare the approaches of the `BinarySearchTree` wrapper and the recursive `RedBlackTree`, we present here the implementation of the `SymTab` structure we investigated at the beginning of the chapter, but cast in terms of `RedBlackTrees`. Comparison of the approaches is instructive (important differences are highlighted with uppercase comments).

RBSymTab

```
import structure5.*;
import java.util.Iterator;
public class RBSymTab<S extends Comparable<S>,T>
{
    protected RedBlackTree<ComparableAssociation<S,T>> table;

    public RBSymTab()
    // post: constructs empty symbol table
    {
        table = new RedBlackTree<ComparableAssociation<S,T>>();
    }

    public boolean contains(S symbol)
    // pre: symbol is non-null string
    // post: returns true iff string in table
    {
        return table.contains(new ComparableAssociation<S,T>(symbol,null));
    }

    public void add(S symbol, T value)
    // pre: symbol non-null
    // post: adds/replaces symbol-value pair in table
```

```
        {
            ComparableAssociation<S,T> a = new ComparableAssociation<S,T>(symbol,value);
            if (table.contains(a)) table = table.remove(a);
            table = table.add(a);
        }

        public T get(S symbol)
        // pre: symbol non-null
        // post: returns token associated with symbol
        {
            ComparableAssociation<S,T> a = new ComparableAssociation<S,T>(symbol,null);
            if (table.contains(a)) {
                a = table.get(a);
                return a.getValue();
            } else {
                return null;
            }
        }

        public T remove(S symbol)
        // pre: symbol non-null
        // post: removes value associated with symbol and returns it
        //       if error returns null
        {
            ComparableAssociation<S,T> a = new ComparableAssociation<S,T>(symbol,null);
            if (table.contains(a)) {
                a = table.get(a);
                table = table.remove(a);
                return a.getValue();
            } else {
                return null;
            }
        }
    }
```

The entire definition of `RedBlackTrees` is available in the `structure` package, when $O(\log n)$ performance is desired. For more details about the structure, please see the documentation within the code.

14.8 Conclusions

A binary search tree is the product of imposing an order on the nodes of a binary tree. Each node encountered in the search for a value represents a point where a decision can be accurately made to go left or right. If the tree is short and fairly balanced, these decisions have the effect of eliminating a large portion of the remaining candidate values.

The binary search tree is, however, a product of the history of the insertion of values. Since every new value is placed at a leaf, the internal nodes are left

untouched and make the structure of the tree fairly static. The result is that poor distributions of data can cause degenerate tree structures that adversely impact the performance of the various search tree methods.

To combat the problem of unbalanced trees, various rotation-based optimizations are possible. In splay trees, rotations are used to force a recently accessed value and its ancestors closer to the root of the tree. The effect is often to shorten degenerate trees, resulting in an amortized logarithmic behavior. A remarkable feature of this implementation is that there is no space penalty: no accounting information needs to be maintained in the nodes.

Self Check Problems

Solutions to these problems begin on page 449.

14.1 What motivates the use of *binary* search trees?

14.2 Suppose values have only been added into a `BinarySearchTree`. Where is the first node added to the tree? Where is the last node added to the tree?

14.3 What is an associative structure?

14.4 Which node becomes the root after a tree is rotated left?

14.5 Is the right rotation the reverse of the left rotation?

14.6 If two values are equal (using `equals`) are they found near each other in a `BinarySearchTree`?

14.7 Why is it so difficult to construct an `Iterator` for a `SplayTree`?

14.8 What is the primary advantage of a red-black tree over a splay tree?

Problems

Solutions to the odd-numbered problems begin on page 483.

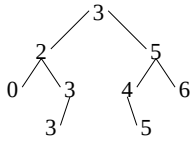
14.1 What distinguishes a binary search tree from a binary tree?

14.2 Draw all three-node integer-valued trees whose nodes are visited in the order 1-2-3 in an in-order traversal. Which trees are binary search trees?

14.3 Draw all three-node integer-valued trees whose nodes are visited in the order 1-2-3 in a preorder traversal. Which trees are binary search trees?

14.4 Draw all three-node integer-valued trees whose nodes are visited in the order 1-2-3 in a postorder traversal. Which trees are binary search trees?

14.5 Redraw the following binary search tree after the root has been removed.



14.6 Redraw the tree shown in Problem 14.5 after the leaf labeled 3 is removed.

14.7 Redraw the tree shown in Problem 14.5 after it is splayed at the leaf labeled 3.

14.8 The `locate` methods from `OrderedVectors` and `BinarySearchTrees` are very similar. They have, for example, similar best-case behaviors. Explain why their behaviors differ in the worst case.

14.9 Prove that, if values are distinct, any binary search tree can be constructed by appropriately ordering insertion operations.

14.10 In splay trees rotations are performed, possibly reversing the parent-child relationship between two equal values. It is now possible to have a root node with a right child that is equal. Explain why this will not cause problems with each of the current methods `locate`, `add`, and `remove`.

14.11 Describe the topology of a binary search tree after the values 1 through n have been inserted in order. How long does the search tree take to construct?

14.12 Describe the topology of a splay tree after the values 1 through n have been inserted in order. How long does the splay tree take to construct?

14.13 Because the `remove` method of binary search trees prefers to replace a node with its predecessor, one expects that a large number of `removes` will cause the tree to lean toward the right. Describe a scheme to avoid this problem.

14.14 Suppose n distinct values are stored in a binary tree. It is noted that the tree is a min-heap *and* a binary search tree. What does the tree look like?

14.15 As we have seen, the splay tree requires the construction of an iterator that stores a single reference to the tree, rather than an unlimited number of

references to ancestors. How does this reduction in space utilization impact the running time of the iterator?

14.16 Write an `equals` method for binary search trees. It should return `true` if both trees contain equal values.

14.17 Having answered Problem 14.16, is it possible to accurately use the same method for splay trees?

14.18 Write a `copy` method for binary search trees. The result of the `copy` should be equal to the original. Carefully argue the utility of your approach.

14.19 Prove that the expected time to perform the `next` method of the splay tree iterator is constant time.

14.9 Laboratory: Improving the BinarySearchTree

Objective. To understand it is possible to improve an implementation.

Discussion. As we have seen in the implementation of the `BinarySearchTree` class, the insertion of values is relative to the root of the tree. One of the situations that must be handled carefully is the case where more than one node can have the same key. If equal keys are allowed in the binary search tree, then we must be careful to have them inserted on one side of the root. This behavior increases the complexity of the code, and when there are many duplicate keys, it is possible that the tree's depth can be increased considerably.

Procedure. An alternative approach is to have all the nodes with similar keys stored in the same location. When the tree is constructed in this manner, then there is no need to worry about keeping similar keys together—they're *always together*.

In this lab, we will implement a `BinaryMultiTree`—a `BinarySearchTree`-like structure that stores a multiset (a set of values with potential duplicates). We are not so concerned with the set features, but we are demanding that different values are kept in sorted order in the structure. In particular, the traversal of the `BinaryMultiTree` should return the values in order.

In this implementation, a `BinaryTree` is used to keep track of a `List` of values that are equal when compared with the `compare` method of the `ordering Comparator`. From the perspective of the structure, there is no distinguishing the members of the list. Externally, the interface to the `BinaryMultiTree` is exactly the same as the `BinarySearchTree`, but the various methods work with values stored in `Lists`, as opposed to working with the values directly. For example, when we look at a value stored in a node, we find a `List`. A `getFirst` of this `List` class picks out an example that is suitable, for example, for comparison purposes.

Here are some things to think about during your implementation:

1. The `size` method does not return the number of nodes; it returns the number of values stored in all the nodes. The bookkeeping is much the same as it was before, but `size` is an upper bound on the actual size of the search tree.

2. The `add` method compares values to the heads of lists found at each node along the way. A new node is created if the value is not found in the tree; the value is inserted in a newly created `List` in the `BinaryTreeNode`. When an equal key is found, the search for a location stops, and the value is added to the `List`. A carefully considered `locate` method will help considerably here.

3. The `contains` method is quite simple: it returns `true` if the `getFirst` of any of the `Lists` produces a similar value.

4. The `get` method returns one of the matching values, if found. It should probably be the same value that would be returned if a `remove` were executed in the same situation.

5. The `iterator` method returns an `Iterator` that traverses all the values of the `BinarySearchTree`. When a list of equal values is encountered, they are all considered before a larger value is returned.

When you are finished, test your code by storing a large list of names of people, ordered only by last name (you will note that this is a common technique used by stores that keep accounts: "Smith?" "Yes!" "Are you Paul or John?"). You should be able to roughly sort the names by inserting them into a `BinaryMultiTree` and then iterating across its elements.

Thought Questions. Consider the following questions as you complete the lab:

1. Recall: What is the problem with having equal keys stored on either side of an equal-valued root?

2. Does it matter what type of `List` is used? What kinds of operations are to be efficient in this `List`?

3. What is the essential difference between implementing the tree as described and, say, just directly storing linked lists of equivalent nodes in the `BinarySearchTree`?

4. An improved version of this structure might use a `Comparator` for primary and secondary keys. The primary comparison is used to identify the correct location for the value in the `BinaryMultiTree`, and the secondary key could be used to order the keys that appear equal using the primary key. Those values that are equal using the primary key are kept within an `OrderedStructure` that keeps track of its elements using the secondary key `Comparator`.

Notes: