Chapter 14 The Binary Search Tree

In Chapter 5 we discussed the binary search algorithm, which depends on a sorted vector. Although the binary search, being in O(lg(*n*)), is very efficient, inserting a new element into a vector without destroying its sorted ordering is relatively inefficient by comparison. In this chapter we will examine the binary search tree, a linked binary tree container that supports efficient searching *and* efficient insertion.

14.1 Our Current Model

Our current software design appears below in Figure 14.1.

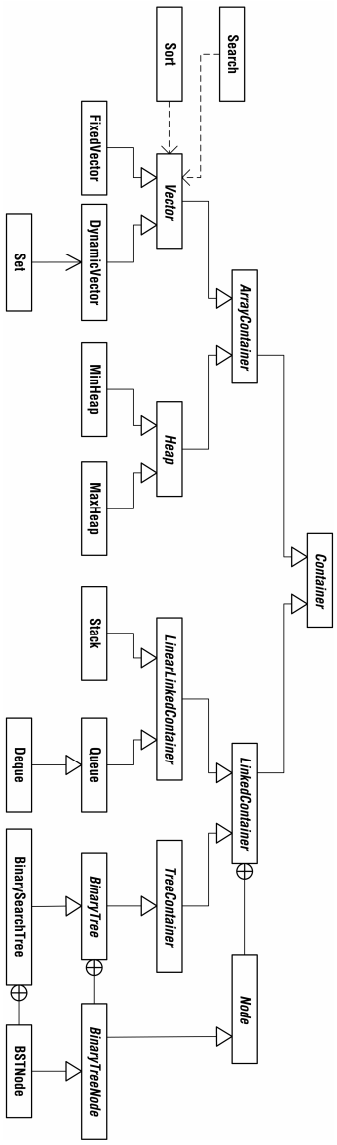


Fig. 14.1. Our current design, in which `BinarySearchTree` specializes `BinaryTree`

14.2 The Binary Search Tree ADT

A binary search tree (BST) is a *totally ordered* binary tree. The BST's total ordering does the heap's partial ordering one better; not only is there a relationship between a BST node and its children, but there is also a definite relationship between the children. In a BST, the value of a node's left child is less than the value of the node itself, and the value of a node's right child is greater than or equal to the value of the node. Consequently, the value of a node's left child is always less than the value of its right child. We will soon see how this total ordering makes the BST's fundamental operations efficient in the average case. The BST operations are shown in the following ADT specification.

```
binary search tree (BST): a totally ordered binary tree

operations:

                     clear() - Make the collection empty.
                find(element) - Search for the given element.
                     height() - What is the height of the tree?
  inorderTraverse(processor) - Perform an inorder traversal of the
                               tree, applying some operation to
                               each node.
             insert(element) - Add the given element to the
                               collection.
                   isEmpty() - Is the collection empty?
                   maximum() - Get the maximum element of the
                               collection.
                   minimum() - Get the minimum element of the
                               collection.
postorderTraverse(processor) - Perform a postorder traversal of
                               the tree, applying some operation
                               to each node.
        predecessor(element) - Get the inorder predecessor of the
                               given element.
 preorderTraverse(processor) - Perform a preorder traversal of the
                               tree, applying some operation to
                               each node.
             remove(element) - Remove the given element from the
                               collection.
                      size() - How many elements are in the
                               collection?
          successor(element) - Get the inorder successor of the
                               given element.
```

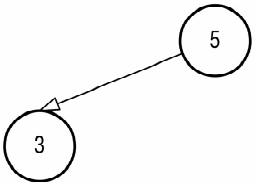
14.3 BST Operations

Now we will have a look at the `insert`, `find`, and `remove` operations of the BST.

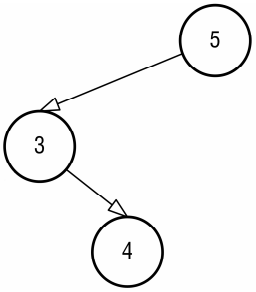
14.3.1 Insertion into a BST

Let us insert into a BST the following values, in the order given: 5, 3, 4, 8, 1, 6, 4. Since the tree is initially empty, the first value, 5, becomes the new tree's root.

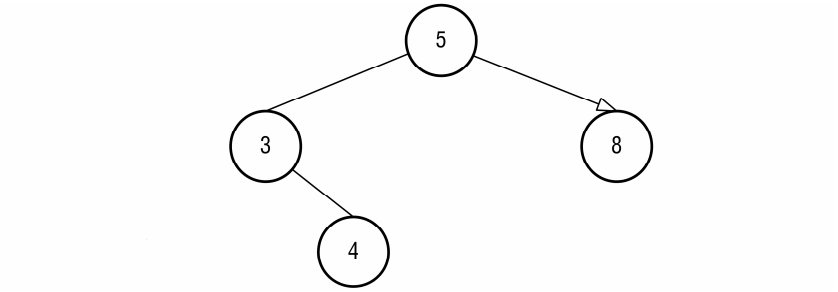
The next value, 3, is less than 5, and so the 3 becomes 5's left child.



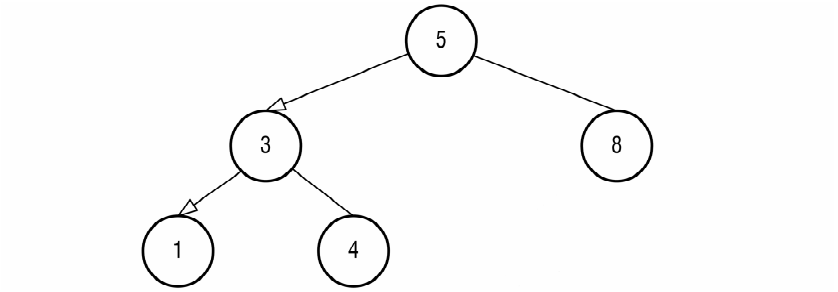
The third value, 4, is less than 5, which means that the 4 must be placed somewhere in the root's left subtree. Thus we move to 5's left child, the node containing 3. Since 4 is greater than 3 and the node containing 3 has no right child, a new node containing 4 becomes 3's right child.



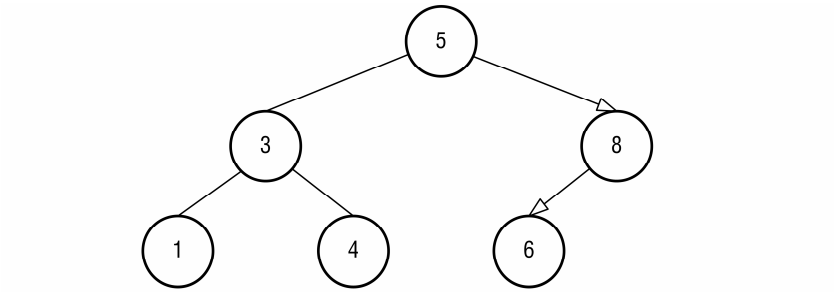
The next value, 8, is greater than the value at the root, and so the 8 must be placed somewhere in the root's right subtree. Since the root does not yet have a right child, a new node containing 8 becomes the root's right child.



The next value to be inserted is 1. We again start at the root and move left or right depending on the relationship between the value to be inserted and the value at the current node. Since 1 is less than 5, we move to 5's left child. The 1 is also less than 3, the value of the current node. That node has no left child, and so a new node containing 1 is placed as 3's left child.



The sixth value, 6, is greater than the root value, and so we move to the root's right child, 8. Since 6 is less than 8 and 8 has no left child, the 6 becomes 8's left child.



Insertion of the last value, a second 4, requires that we go left and then right, arriving at the first 4. Since 4 is greater than or equal to 4, the second 4 becomes the right child of the first. Thus we arrive at the final tree shown below.

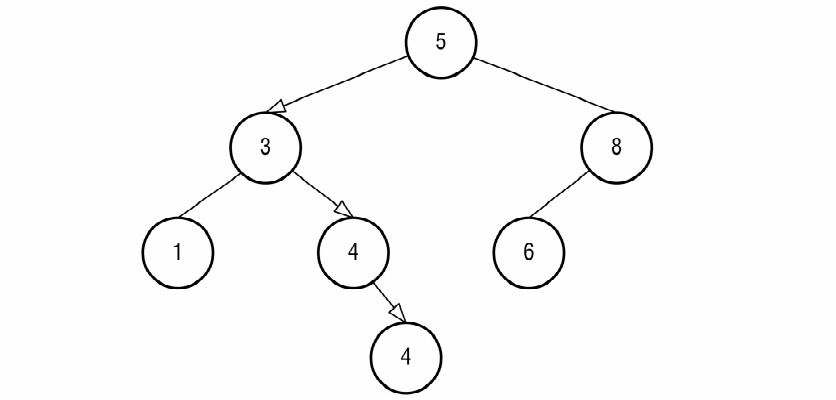


Fig. 14.2. The BST that results from the insertion of 5, 3, 4, 8, 1, 6, and 4, in that order

14.3.2 Finding an Element of a BST

Searching a BST involves the same navigation that is employed for insertion. A search begins at the root, and left or right "turns" are made until the target element is located or until the search encounters an empty child, in which case the target element is not a member of the collection.

14.3.3 Removing from a BST

Removal is by far the most complex of the BST operations, owing to the fact that a number of cases must be handled. Pseudocode for the `remove` operation appears below.

```
REMOVING AN ELEMENT OF A BST

1. Find the node to be removed. If no such node exists, terminate.

2. (1) if (the node to be removed is a leaf)

       If the target node has a parent, i.e., if the target node
       is not the root, then overwrite the parent's reference to
       the target node with null, unlinking the target node from
       the tree.

   (2) else if (the target node has no left child)

       Overwrite the target node's data portion and the links to
       its children with those of its right child, effectively
       transforming the node into its right child.

   (3) else if (the target node has no right child)

       Transform the target node into its left child.

   (4) else // the target node has two children.

       Replace the target node's data portion with that of its
       successor, which is the minimum node of the target node's
       right subtree. Then delete the successor, to which case (1)
       or case (2) must apply.
```

You should verify the correctness of this algorithm by applying it to the BST of Figure 14.2. Delete elements 1, 4, 8, and 5, restoring the tree to its original form after each deletion.

14.4 BST Implementation

The code for class `BinarySearchTree` is shown below. Most of the work is done recursively, following the pattern used in implementing class `BinaryTree`.

```
public class BinarySearchTree extends BinaryTree
{
    // A BST needs Comparable nodes because a BST is totally
    // ordered.

    public static class BSTNode extends BinaryTreeNode
        implements Comparable
    {
        public BSTNode(Comparable dat)
        {
            super(dat);
        }

        public BSTNode(Comparable dat,
                       BSTNode lc,
                       BSTNode rc,
                       BSTNode par)
        {
            super(dat, lc, rc, par);
        }

        public int compareTo(Object o)
        {
            Comparable c = (Comparable)((BSTNode)o).data;
            return ((Comparable)data).compareTo(c);
        }
    }

    // The following method is the recursive helper for find(). It
    // returns the found node or null.

    protected BSTNode findHelper(BSTNode current, BSTNode target)
    {
        if (current == null)  // If we have left the tree, then
            return null;      // the target was not found.

        // If the target node is less than the current node, then
        // turn left.

        if (target.compareTo(current) < 0)
            return findHelper((BSTNode)current.leftChild, target);

        // If the target node is greater than the current node,
        // then turn right.

        if (target.compareTo(current) > 0)
            return findHelper((BSTNode)current.rightChild,
                              target);

        // Otherwise the current node is the target node.

        return current;
    }

    public Comparable find(Comparable target)
    {
        BSTNode found = findHelper((BSTNode)root,
                                   new BSTNode(target));
        if (found == null)
            return null;
        return (Comparable)found.data;
    }
```

```
// The following method is the recursive helper for insert().

protected BSTNode insertHelper(BSTNode current,
                               BSTNode newNode)
{
    // If we have found the correct location for the new node,
    // then return the reference to the new node, linking it
    // to the tree.

    if (current == null)
    {
        numItems++;
        return newNode;
    }

    // If the new node is less than the current node, then go
    // left.

    if (newNode.compareTo(current) < 0)
    {
        current.leftChild =
            insertHelper((BSTNode)current.leftChild, newNode);
        current.leftChild.parent = current;
    }
    else  // Otherwise go right.
    {
        current.rightChild =
            insertHelper((BSTNode)current.rightChild,
                         newNode);
        current.rightChild.parent = current;
    }
    return current;
}

public void insert(Comparable newItem)
{
    root = insertHelper((BSTNode)root, new BSTNode(newItem));
}

/*
The following method is the helper for maximum(). This
method is not recursive because the maximum node of a
BST is easy to find using a loop: go right until the
rightmost node is found.
*/

protected BSTNode maxHelper(BSTNode current)
{
    if (current == null)
        return null;
    while (current.rightChild != null)
        current = (BSTNode)current.rightChild;
    return current;
}

public Comparable maximum()
{
    BSTNode max = maxHelper((BSTNode)root);
    if (max == null)
        return null;
    return (Comparable)max.data;
}
```

```
// The minimum node of a BST is the leftmost node.

protected BSTNode minHelper(BSTNode current)
{
    if (current == null)
        return null;
    while (current.leftChild != null)
        current = (BSTNode)current.leftChild;
    return current;
}

public Comparable minimum()
{
    BSTNode min = minHelper((BSTNode)root);
    if (min == null)
        return null;
    return (Comparable)min.data;
}

/*
The inorder predecessor of node n is:

    1. the maximum node of n's left subtree, provided
       that n has a left subtree.
    2. the first ancestor of n such that n is in the ances
       tor's right subtree.
*/

public Comparable predecessor(Comparable target)
{
    BSTNode found =
        findHelper((BSTNode)root, new BSTNode(target));
    if (found == null)
        return null;
    if (found.leftChild != null)
        return (Comparable)maxHelper((BSTNode)found.leftChild).data;
    BSTNode parent = (BSTNode)found.parent;
    while (parent != null && parent.compareTo(found) > 0)
        parent = (BSTNode)parent.parent;
    if (parent == null)
        return null;
    return (Comparable)parent.data;
}
```

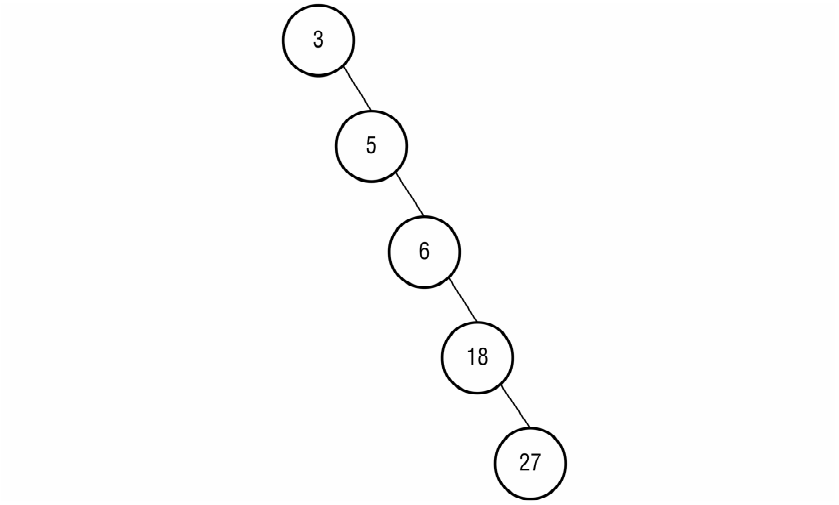
```
// Method removeHelper() implements the algorithm specified
// in 14.3.3.

protected BSTNode removeHelper(BSTNode current,
                               BSTNode target)
{
    if (current == null)
        return null;
    if (target.compareTo(current) < 0)
        current.leftChild =
            removeHelper((BSTNode)current.leftChild, target);
    else if (target.compareTo(current) > 0)
        current.rightChild =
            removeHelper((BSTNode)current.rightChild, target);
    else
    {
        if (current.isLeaf())
        {
            numItems--;
            return null;
        }
        BSTNode temp;
        if (current.leftChild == null)
        {
            temp = (BSTNode)current.rightChild;
            current.data = temp.data;
            current.leftChild = temp.leftChild;
            if (current.leftChild != null)
                current.leftChild.parent = current;
            current.rightChild = temp.rightChild;
            if (current.rightChild != null)
                current.rightChild.parent = current;
        }
        else if (current.rightChild == null)
        {
            temp = (BSTNode)current.leftChild;
            current.data = temp.data;
            current.leftChild = temp.leftChild;
            if (current.leftChild != null)
                current.leftChild.parent = current;
            current.rightChild = temp.rightChild;
            if (current.rightChild != null)
                current.rightChild.parent = current;
        }
        else
        {
            temp = (BSTNode)current.rightChild;
            if (temp.isLeaf())
            {
                current.data = temp.data;
                current.rightChild = null;
            }
            else if (temp.leftChild == null)
            {
                current.data = temp.data;
                current.rightChild = temp.rightChild;
                if (current.rightChild != null)
                    current.rightChild.parent = current;
            }
            else
            {
                while (temp.leftChild.leftChild != null)
                    temp = (BSTNode)temp.leftChild;
                current.data = temp.leftChild.data;
                removeHelper((BSTNode)temp,
                    new BSTNode((Comparable)temp.leftChild.data));
                numItems++;
            }
        }
        numItems--;
    }
    return current;
}
```

```
    public void remove(Comparable target)
    {
        removeHelper((BSTNode)root, new BSTNode(target));
    }

    /*
    The inorder successor of node n is:

        1. the minimum node of n's right subtree, provided
           that n has a right subtree.
        2. the first ancestor of n such that n is in the ances
           tor's left subtree.
    */

    public Comparable successor(Comparable target)
    {
        BSTNode found =
            findHelper((BSTNode)root, new BSTNode(target));
        if (found == null)
            return null;
        if (found.rightChild != null)
            return (Comparable)minHelper((BSTNode)found.rightChild).data;
        BSTNode parent = (BSTNode)found.parent;
        while (parent != null && parent.compareTo(found) <= 0)
            parent = (BSTNode)parent.parent;
        if (parent == null)
            return null;
        return (Comparable)parent.data;
    }
}
```

14.5 Time Analysis of the Fundamental BST Operations

A tree is said to be *balanced* if it has the least possible height. The height of a balanced binary tree with n nodes is approximately $\lg(n)$. Since a BST insertion, search, or removal may require navigation to the deepest node of the tree, the BST `insert`, `find`, and `remove` operations are all in $O(\lg(n))$ for balanced, or nearly balanced, BSTs.

In the worst case, however, operations `insert`, `find`, and `remove` are in $O(n)$, for in the worst case a BST is simply a list. Consider the BST shown below, which results from insertion of 3, 5, 6, 18, and 27, in that order.



This sorry state of affairs can be avoided by rebalancing the BST should it become unbalanced through insertions and deletions. Each of two balanced BST specializations addresses rebalancing in its own way. These specializations are the *AVL tree* and the *red-black tree*. The *splay tree* is a kind of BST that guarantees efficient *amortized*, i.e., long-run, performance, rather than guaranteeing the efficiency of any single operation.

14.6 BST Applications

The most natural application of the BST is as a dictionary. Two end-of-chapter exercises deal with this application.

Exercises

1. Add attributes and operations to the UML class diagram of Figure 14.1
2. Design and code a class called `DirectoryEntry`, an instance of which represents an entry in the white pages of a telephone directory. Code a console application that allows its user to query and alter a telephone directory that takes the form of a BST.
3. When a Java compiler encounters a non-symbol token in a Java source file, the compiler must identify the token as a Java reserved word or as a user-defined identifier. Code a console application that reads a comment-free Java source file and lists alphabetically all of the file's identifiers.

   Your application should prompt the user for the path to the desired Java source file. Store Java's reserved words in one BST and the program's identifiers in another. Use an instance of `java.util.StringTokenizer` to tokenize each line of the input file. You should shuffle the reserved words before inserting them into their BST. The code below performs a *Fisher-Yates shuffle* on a vector.

```
int i,
    j;
Random rand = new Random();
for (i = vec.size() - 1; i > 0; i--)
{
    j = rand.nextInt(i);
    Sort.swap(vec, i, j);
}
```