6.5 Binary Search Trees

Overview of a Binary Search Tree

In Section 6.1, we provided the following recursive definition of a binary search tree:

A set of nodes T is a binary search tree if either of the following is true:

- T is empty.
- If T is not empty, its root node has two subtrees, T_L and T_R, such that T_L and T_R are binary search trees and the value in the root node of T is greater than all values in T_L and is less than all values in T_R.

Figure 6.13 shows a binary search tree that contains the words in lowercase from the nursery rhyme "The House That Jack Built." We can use the following algorithm to find an object in a binary search tree.

Recursive Algorithm for Searching a Binary Search Tree

1. **`if`** the root is **`null`**
2. The item is not in the tree; return **`null`**.
3. Compare the value of **`target`**, the item being sought, with `root.data`.
4. **`if`** they are equal
5. The target has been found, return the data at the root.

   `else if` target is less than `root.data`
6. Return the result of searching the left subtree.

   `else`
7. Return the result of searching the right subtree.

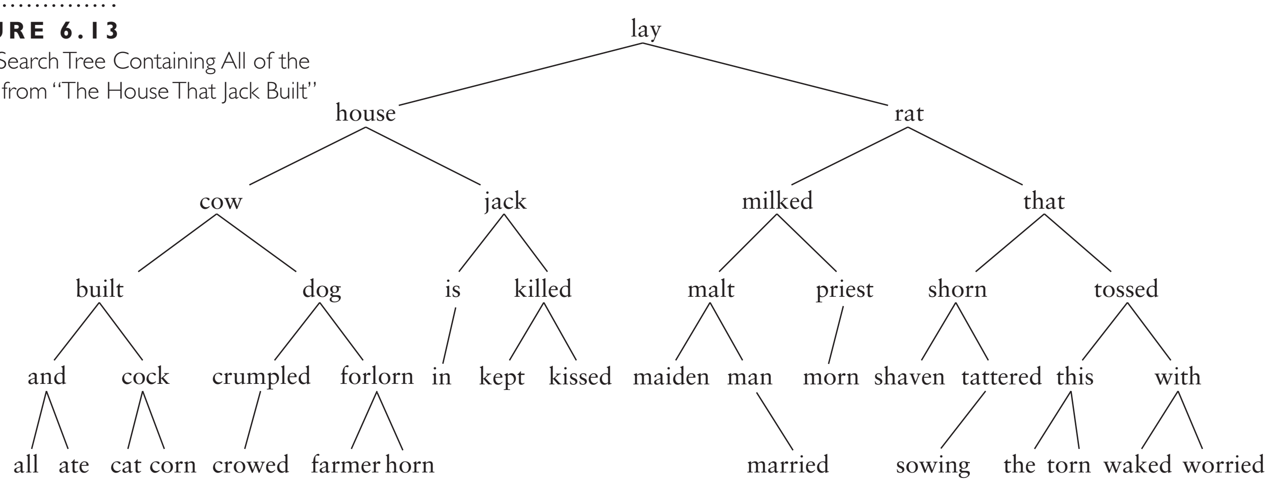


FIGURE 6.13
Binary Search Tree Containing All of the Words from "The House That Jack Built"

EXAMPLE 6.7 Suppose we wish to find *jill* in Figure 6.13. We first compare *jill* with *lay*. Because *jill* is less than *lay*, we continue the search with the left subtree and compare *jill* with house. Because *jill* is greater than *house*, we continue with the right subtree and compare *jill* with *jack*. Because *jill* is greater than *jack*, we continue with *killed* followed by *kept*. Now, *kept* has no left child, and *jill* is less than *kept*, so we conclude that *jill* is not in this binary search tree. (She's in a different nursery rhyme.) Follow the path shown in gray in Figure 6.14.

FIGURE 6.14
Looking for Jill

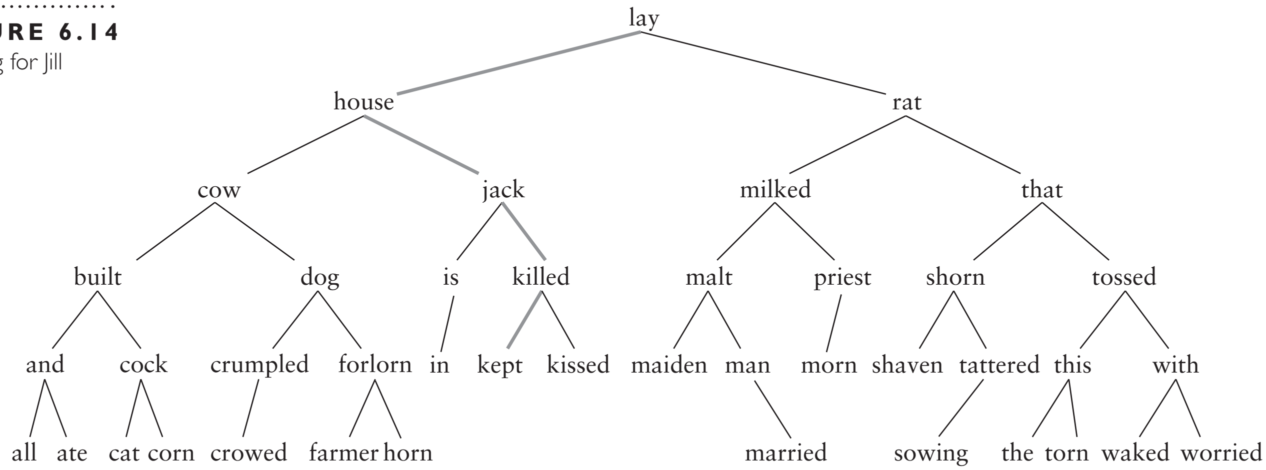
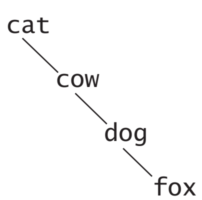


TABLE 6.3
The `SearchTree<E>` Interface

Method	Behavior
`boolean add(E item)`	Inserts `item` where it belongs in the tree. Returns **`true`** if item is inserted; **`false`** if it isn't (already in tree)
`boolean contains(E target)`	Returns **`true`** if `target` is found in the tree
`E find(E target)`	Returns a reference to the data in the node that is equal to `target`. If no such node is found, returns **`null`**
`E delete(E target)`	Removes `target` (if found) from tree and returns it; otherwise, returns **`null`**
`boolean remove(E target)`	Removes `target` (if found) from tree and returns **`true`**; otherwise, returns **`false`**



Performance

Searching the tree in Figure 6.14 is O(log n). However, if a tree is not very full, performance will be worse. The tree in the figure at left has only right subtrees, so searching it is O(n).

Interface SearchTree

As described, the binary search tree is a data structure that enables efficient insertion, search, and retrieval of information (best case is O(log n)). Table 6.3 shows a `SearchTree<E>` interface for a class that implements the binary search tree. The interface includes methods for insertion (`add`), search (`boolean contains` and `E find`), and removal (`E delete` and `boolean remove`). Next, we discuss a class `BinarySearchTree<E>` that implements this interface.

The `BinarySearchTree` Class

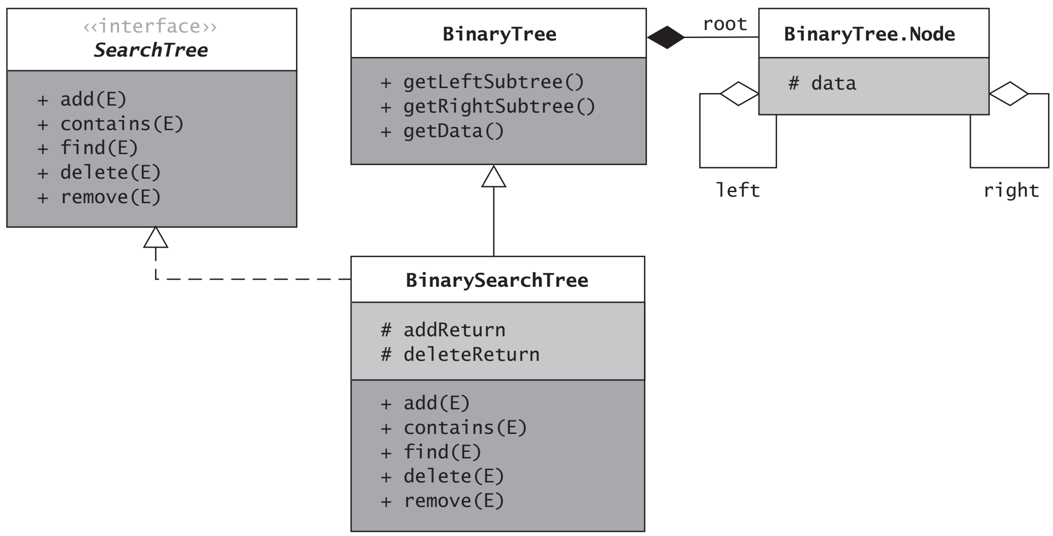
Next, we implement class `BinarySearchTree<E extends Comparable<E>>`. The type parameter specified when we create a new `BinarySearchTree` must implement the `Comparable` interface.

Table 6.4 shows the data fields declared in the class. These data fields are used to store a second result from the recursive `add` and `delete` methods that we will write for this class. Neither result can be returned directly from the recursive `add` or `delete` method because they return a reference to a tree node affected by the insertion or deletion operation. The interface for method `add` in Table 6.3 requires a **`boolean`** result (stored in `addReturn`) to indicate success

TABLE 6.4
Data Fields of Class `BinarySearchTree<E extends Comparable<E>>`

Data Field	Attribute
`protected boolean addReturn`	Stores a second return value from the recursive `add` method that indicates whether the item has been inserted
`protected E deleteReturn`	Stores a second return value from the recursive `delete` method that references the item that was stored in the tree

FIGURE 6.15
UML Diagram of `BinarySearchTree`



or failure. Similarly, the interface for `delete` requires a type `E` result (stored in `deleteReturn`) that is either the item deleted or **`null`**.

The class heading and data field declarations follow. Note that class `BinarySearchTree` extends class `BinaryTree` and implements the `SearchTree` interface (see Figure 6.15). Besides the data fields shown, class `BinarySearchTree` inherits the data field `root` from class `BinaryTree` (declared as **`protected`**) and the inner class `Node<E>`.

```
public class BinarySearchTree<E extends Comparable<E>>
             extends BinaryTree<E> implements SearchTree<E> {
    // Data Fields
    /** Return value from the public add method. */
    protected boolean addReturn;
    /** Return value from the public delete method. */
    protected E deleteReturn;
    . . .
}
```

Implementing the `find` Methods

Earlier, we showed a recursive algorithm for searching a binary search tree. Next, we show how to implement this algorithm and a nonrecursive starter method for the algorithm. Our method `find` will return a reference to the node that contains the information we are seeking.

Listing 6.4 shows the code for method `find`. The starter method calls the recursive method with the tree root and the object being sought as its parameters. If `bST` is a reference to a `BinarySearchTree`, the method call `bST.find(target)` invokes the starter method.

The recursive method first tests the local root for **`null`**. If it is **`null`**, the object is not in the tree, so **`null`** is returned. If the local root is not **`null`**, the statement

```
int compResult = target.compareTo(localRoot.data);
```

compares `target` to the data at the local root. Recall that method `compareTo` returns an **`int`** value that is negative, zero, or positive depending on whether the object (`target`) is less than, equal to, or greater than the argument (`localRoot.data`).

If the objects are equal, we return the data at the local root. If `target` is smaller, we recursively call the method `find`, passing the left subtree root as the parameter.

```
return find(localRoot.left, target);
```

Otherwise, we call `find` to search the right subtree.

```
return find(localRoot.right, target);
```

LISTING 6.4

`BinarySearchTree` `find` Method

```
/** Starter method find.
    pre: The target object must implement
         the Comparable interface.
    @param target The Comparable object being sought
    @return The object, if found, otherwise null
 */
public E find(E target) {
    return find(root, target);
}

/** Recursive find method.
    @param localRoot The local subtree's root
    @param target The object being sought
    @return The object, if found, otherwise null
 */
private E find(Node<E> localRoot, E target) {
    if (localRoot == null)
        return null;

    // Compare the target with the data field at the root.
    int compResult = target.compareTo(localRoot.data);
    if (compResult == 0)
        return localRoot.data;
    else if (compResult < 0)
        return find(localRoot.left, target);
    else
        return find(localRoot.right, target);
}
```

Insertion into a Binary Search Tree

Inserting an item into a binary search tree follows a similar algorithm as searching for the item because we are trying to find where in the tree the item would be, if it were there. In searching, a result of **`null`** is an indicator of failure; in inserting, we replace this **`null`** with a new leaf that contains the new item. If we reach a node that contains the object we are trying to insert, then we can't insert it (duplicates are not allowed), so we return **`false`** to indicate that we were unable to perform the insertion. The insertion algorithm follows.

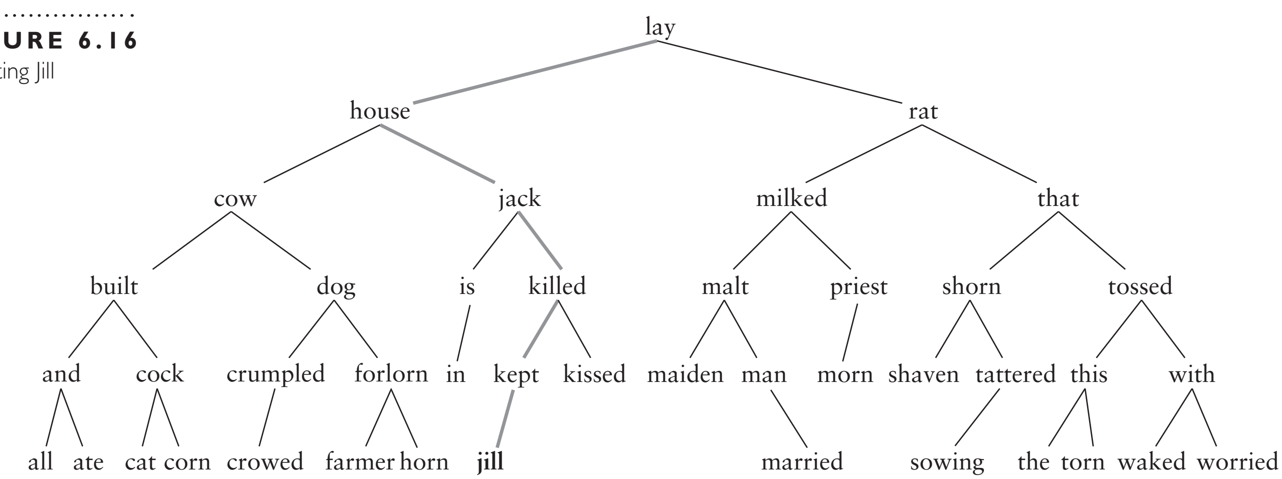
Recursive Algorithm for Insertion in a Binary Search Tree

1. `if` the root is `null`
2. Replace empty tree with a new tree with the item at the root and return **`true`**.
3. `else if` the item is equal to `root.data`
4. The item is already in the tree; return **`false`**.
5. `else if` the item is less than `root.data`
6. Recursively insert the item in the left subtree.
7. `else`
8. Recursively insert the item in the right subtree.

The algorithm returns **`true`** when the new object is inserted and **`false`** if it is a duplicate (the second stopping case). The first stopping case tests for an empty tree. If so, a new `BinarySearchTree` is created and the new item is stored in its root node (Step 2).

EXAMPLE 6.8 To insert *jill* into Figure 6.13, we would follow the steps shown in Example 6.7 except that when we reached *kept*, we would insert *jill* as the left child of the node that contains *kept* (see Figure 6.16).

FIGURE 6.16
Inserting Jill



Implementing the add Methods

Listing 6.5 shows the code for the starter and recursive `add` methods. The recursive `add` follows the algorithm presented earlier, except that the return value is the new (sub)tree that contains the inserted item. The data field `addReturn` is set to **`true`** if the item is inserted and to **`false`** if the item already exists. The starter method calls the recursive method with the root as its argument. The root is set to the value returned by the recursive method (the modified tree). The value of `addReturn` is then returned to the caller.

In the recursive method, the statements

```
addReturn = true;
return new Node<>(item);
```

execute when a **`null`** branch is reached. The first statement sets the insertion result to **`true`**; the second returns a new node containing `item` as its data.

The statements

```
addReturn = false;
return localRoot;
```

execute when `item` is reached. The first statement sets the insertion result to **`false`**; the second returns a reference to the subtree that contains `item` in its root.

If `item` is less than the root's data, the statement

```
localRoot.left = add(localRoot.left, item);
```

attempts to insert `item` in the left subtree of the local root. After returning from the call, this left subtree is set to reference the modified subtree, or the original subtree if there is no insertion. The statement

```
localRoot.right = add(localRoot.right, item);
```

affects the right subtree of `localRoot` in a similar way.

LISTING 6.5

`BinarySearchTree` `add` Methods

```
/** Starter method add.
    pre: The object to insert must implement the
         Comparable interface.
    @param item The object being inserted
    @return true if the object is inserted, false
            if the object already exists in the tree
 */
public boolean add(E item) {
    root = add(root, item);
    return addReturn;
}

/** Recursive add method.
    post: The data field addReturn is set true if the item is added to
          the tree, false if the item is already in the tree.
    @param localRoot The local root of the subtree
    @param item The object to be inserted
    @return The new local root that now contains the
            inserted item
 */
private Node<E> add(Node<E> localRoot, E item) {
    if (localRoot == null) {
        // item is not in the tree - insert it.
        addReturn = true;
        return new Node<>(item);
    } else if (item.compareTo(localRoot.data) == 0) {
        // item is equal to localRoot.data
        addReturn = false;
        return localRoot;
    } else if (item.compareTo(localRoot.data) < 0) {
        // item is less than localRoot.data
        localRoot.left = add(localRoot.left, item);
        return localRoot;
    } else {
        // item is greater than localRoot.data
        localRoot.right = add(localRoot.right, item);
        return localRoot;
    }
}
```

PROGRAM STYLE

Comment on Insertion Algorithm and add Methods

Note that as we return along the search path, the statement

```
localRoot.left = add(localRoot.left, item);
```

or

```
localRoot.right = add(localRoot.right, item);
```

resets each local root to reference the modified tree below it. You may wonder whether this is necessary. The answer is "No." In fact, it is only necessary to reset the reference from the parent of the new node to the new node; all references above the parent remain the same. We can modify the insertion algorithm to do this by checking for a leaf node before making the recursive call to `add`:

5.1. **else if** the item is less than `root.data`
5.2.   **if** the local root is a leaf node.
5.3.   Reset the left subtree to reference a new node with the item as its data.
        else
5.4.   Recursively insert the item in the left subtree.

A similar change should be made for the case where `item` is greater than the local root's data. You would also have to modify the starter `add` method to check for an empty tree and insert the new item in the root node if the tree is empty instead of calling the recursive `add` method.

One reason we did not write the algorithm this way is that we want to be able to adjust the tree if the insertion makes it unbalanced. This involves resetting one or more branches above the insertion point. We discuss how this is done in Chapter 9.

PROGRAM STYLE

Multiple Calls to compareTo

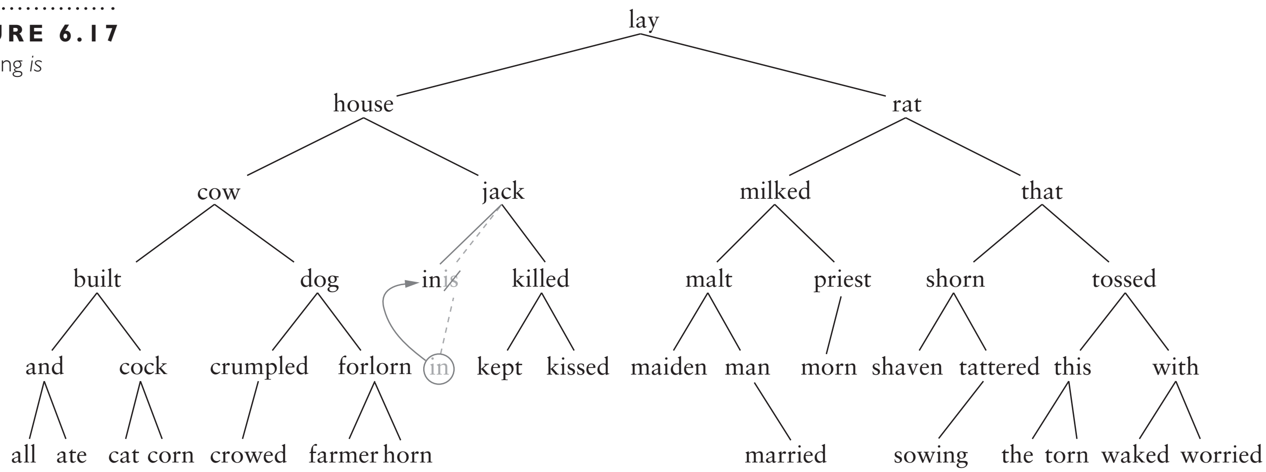
Method `add` has two calls to method `compareTo`. We wrote it this way so that the code mirrors the algorithm. However, it would be more efficient to call `compareTo` once and save the result in a local variable as we did for method `find`. Depending on the number and type of data fields being compared, the extra call to method `compareTo` could be costly.

Removal from a Binary Search Tree

Removal also follows the search algorithm except that when the item is found, it is removed. If the item is a leaf node, then its parent's reference to it is set to `null`, thereby removing the leaf node. If the item has only a left or right child, then the grandparent references the remaining child instead of the child's parent (the node we want to remove).

EXAMPLE 6.9 If we remove is from Figure 6.13, we can replace it with *in*. This is accomplished by changing the left child reference in *jack* (the grandparent) to reference *in* (see Figure 6.17).

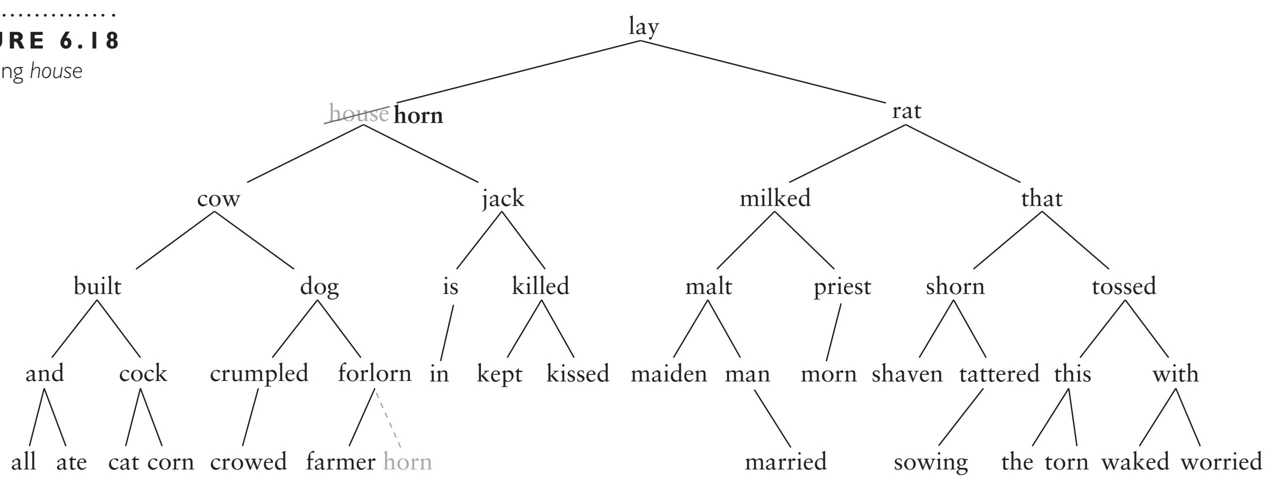
FIGURE 6.17
Removing *is*



A complication arises when the item we wish to remove has two children. In this case, we need to find a replacement parent for the children. Remember that the parent must be larger than all of the data fields in the left subtree and smaller than all of the data fields in the right subtree. If we take the largest item in the left subtree and promote it to be the parent, then all of the remaining items in the left subtree will be smaller. This item is also less than the items in the right subtree. This item is also known as the *inorder predecessor* of the item being removed. (We could use the inorder successor instead; this is discussed in the exercises.)

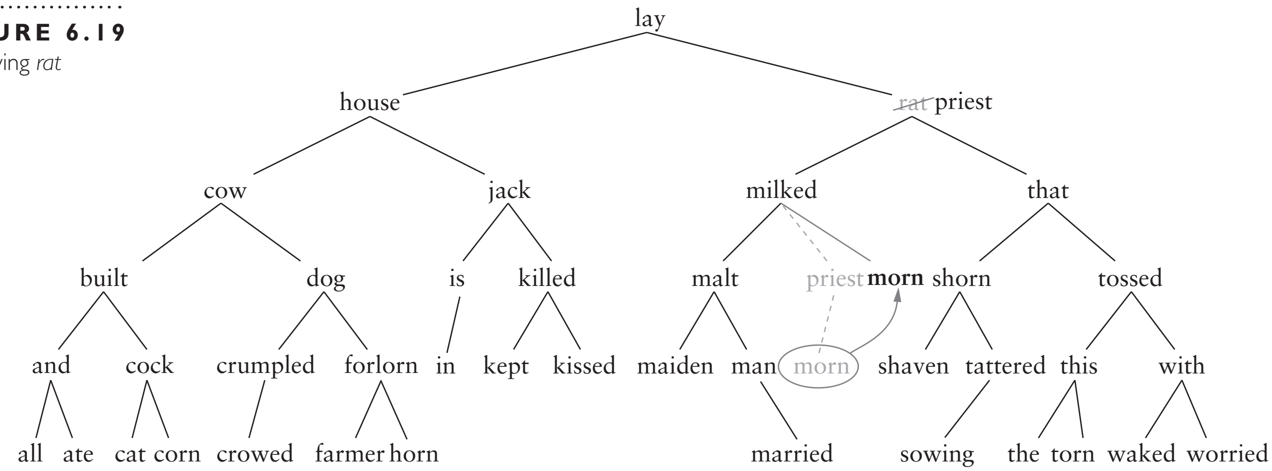
EXAMPLE 6.10 If we remove *house* from Figure 6.13, we look in the left subtree (root contains *cow*) for the largest item, *horn*. We then replace *house* with *horn* and remove the node containing *horn* (see Figure 6.18).

FIGURE 6.18
Removing *house*



EXAMPLE 6.11 If we want to remove *rat* from the tree in Figure 6.13, we would start the search for the inorder successor at *milked* and see that it has a right child, *priest*. If we now look at *priest*, we see that it does not have a right child, but it does have a left child. We would then replace *rat* with *priest* and replace the reference to *priest* in *milked* with a reference to *morn* (the left subtree of the node containing *priest*). See Figure 6.19.

FIGURE 6.19
Removing *rat*



Recursive Algorithm for Removal from a Binary Search Tree

1. `if` the root is `null`
2.   The item is not in tree – return **`null`**.
3. Compare the item to the data at the local root.
4. `if` the item is less than the data at the local root
5.   Return the result of deleting from the left subtree.
6. `else if` the item is greater than the local root
7.   Return the result of deleting from the right subtree.
8. `else` // *The item is in the local root*
9.   Store the data in the local root in `deleteReturn`.
10.   `if` the local root has no children
11.   Set the parent of the local root to reference **`null`**.
12.   `else if` the local root has one child
13.   Set the parent of the local root to reference that child.
14.   `else` // *Find the inorder predecessor*
15.   `if` the left child has no right child it is the inorder predecessor
16.   Set the parent of the local root to reference the left child.
17.   `else`
18.   Find the rightmost node in the right subtree of the left child.
19.   Copy its data into the local root's data and remove it by setting its parent to reference its left child.

Implementing the `delete` Methods

Listing 6.6 shows both the starter and the recursive `delete` methods. As with the `add` method, the recursive `delete` method returns a reference to a modified tree that, in this case, no longer contains the item. The public starter method is expected to return the item removed. Thus, the recursive method saves this value in the data field `deleteReturn` before removing it from the tree. The starter method then returns this value.

LISTING 6.6

`BinarySearchTree delete` Methods

```java
/** Starter method delete.
    post: The object is not in the tree.
    @param target The object to be deleted
    @return The object deleted from the tree
            or null if the object was not in the tree
    @throws ClassCastException if target does not implement
            Comparable
 */
public E delete(E target) {
    root = delete(root, target);
    return deleteReturn;
}
/** Recursive delete method.
    post: The item is not in the tree;
          deleteReturn is equal to the deleted item
          as it was stored in the tree or null
          if the item was not found.
    @param localRoot The root of the current subtree
    @param item The item to be deleted
    @return The modified local root that does not contain
            the item
 */
private Node<E> delete(Node<E> localRoot, E item) {
    if (localRoot == null) {
        // item is not in the tree.
        deleteReturn = null;
        return localRoot;
    }

    // Search for item to delete.
    int compResult = item.compareTo(localRoot.data);
    if (compResult < 0) {
        // item is smaller than localRoot.data.
        localRoot.left = delete(localRoot.left, item);
        return localRoot;
    } else if (compResult > 0) {
        // item is larger than localRoot.data.
        localRoot.right = delete(localRoot.right, item);
        return localRoot;
    } else {
        // item is at local root.
        deleteReturn = localRoot.data;
        if (localRoot.left == null) {
            // If there is no left child, return right child
            // which can also be null.
            return localRoot.right;
```

```
        } else if (localRoot.right == null) {
            // If there is no right child, return left child.
            return localRoot.left;
        } else {
            // Node being deleted has 2 children, replace the data
            // with inorder predecessor.
            if (localRoot.left.right == null) {
                // The left child has no right child.
                // Replace the data with the data in the
                // left child.
                localRoot.data = localRoot.left.data;
                // Replace the left child with its left child.
                localRoot.left = localRoot.left.left;
                return localRoot;
            } else {
                // Search for the inorder predecessor (ip) and
                // replace deleted node's data with ip.
                localRoot.data = findLargestChild(localRoot.left);
                return localRoot;
            }
        }
    }
}
```

For the recursive method, the two stopping cases are an empty tree and a tree whose root contains the item being removed. We first test to see whether the tree is empty (local root is **`null`**). If so, then the item sought is not in the tree. The `deleteReturn` data field is set to **`null`**, and the local root is returned to the caller.

Next, `localRoot.data` is compared to the item to be deleted. If the item to be deleted is less than `localRoot.data`, it must be in the left subtree if it is in the tree at all, so we set `localRoot.left` to the value returned by recursively calling this method.

```
localRoot.left = delete(localRoot.left, item);
```

If the item to be deleted is greater than `localRoot.data`, the statement

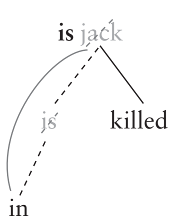
```
localRoot.right = delete(localRoot.right, item);
```

affects the right subtree of `localRoot` in a similar way.

If `localRoot.data` is the item to be deleted, we have reached the second stopping case, which begins with the lines

```
} else {
    // item is at local root.
    deleteReturn = localRoot.data;
. . .
```

The value of `localRoot.data` is saved in `deleteReturn`. If the node to be deleted has one child (or zero children), we return a reference to the only child (or **`null`**), so the parent of the deleted node will reference its only grandchild (or **`null`**).



If the node to be deleted (*jack* in the figure at left) has two children, we need to find the replacement for this node. If its left child has no right subtree, the left child (*is*) is the inorder predecessor. The first statement below

```
localRoot.data = localRoot.left.data;
// Replace the left child with its left child.
localRoot.left = localRoot.left.left;
```

copies the left child's data into the local node's data (*is* to *jack*); the second resets the local node's left branch to reference its left child's left subtree (*in*).

If the left child of the node to be deleted has a right subtree, the statement

```
localRoot.data = findLargestChild(localRoot.left);
```

calls `findLargestChild` to find the largest child and to remove it. The largest child's data is referenced by `localRoot.data`. This is illustrated in Figure 6.19. The left child *milked* of the node to be deleted (*rat*) has a right child *priest*, which is its largest child. Therefore, *priest* becomes referenced by `localRoot.data` (replacing *rat*) and *morn* (the left child of *priest*) becomes the new right child of *milked*.

Method `findLargestChild`

Method `findLargestChild` (see Listing 6.7) takes the parent of a node as its argument. It then follows the chain of rightmost children until it finds a node whose right child does not itself have a right child. This is done via tail recursion.

When a parent node is found whose right child has no right child, the right child is the inorder predecessor of the node being deleted, so the data value from the right child is saved.

```
E returnValue = parent.right.data;
parent.right = parent.right.left;
```

The right child is then removed from the tree by replacing it with its left child (if any).

LISTING 6.7
`BinarySearchTree` `findLargestChild` Method

```
/** Find the node that is the
    inorder predecessor and replace it
    with its left child (if any).
    post: The inorder predecessor is removed from the tree.
    @param parent The parent of possible inorder
                  predecessor (ip)
    @return The data in the ip
 */
private E findLargestChild(Node<E> parent) {
    // If the right child has no right child, it is
    // the inorder predecessor.
    if (parent.right.right == null) {
        E returnValue = parent.right.data;
        parent.right = parent.right.left;
        return returnValue;
    } else {
        return findLargestChild(parent.right);
    }
}
```

Testing a Binary Search Tree

To test a binary search tree, you need to verify that an inorder traversal will display the tree contents in ascending order after a series of insertions (to build the tree) and deletions are performed. You need to write a `toString` method for a `BinarySearchTree` that returns the `String` built from an inorder traversal (see Programming Exercise 3).

CASE STUDY Writing an Index for a Term Paper

Problem You would like to write an index for a term paper. The index should show each word in the paper followed by the line number on which it occurred. The words should be displayed in alphabetical order. If a word occurs on multiple lines, the line numbers should be listed in ascending order. For example, the three lines

```
a, 3
a, 13
are, 3
```

show that the word *a* occurred on lines 3 and 13 and the word *are* occurred on line 3.

Analysis A binary search tree is an ideal data structure to use for storing the index entries. We can store each word and its line number as a string in a tree node. For example, the two occurrences of the word *Java* on lines 5 and 10 could be stored as the strings `"java, 005"` and `"java, 010"`. Each word will be stored in lowercase to ensure that it appears in its proper position in the index. The leading zeros are necessary so that the string `"java, 005"` is considered less than the string `"java, 010"`. If the leading zeros were removed, this would not be the case (`"java, 5"` is greater than `"java, 10"`). After all the strings are stored in the search tree, we can display them in ascending order by performing an inorder traversal. Storing each word in a search tree is an O(log n) process where n is the number of words currently in the tree. Storing each word in an ordered list would be an O(n) process.

Design We can represent the index as an instance of the `BinarySearchTree` class just discussed or as an instance of a binary search tree provided in the Java API. The Java API provides a class `TreeSet<E>` (discussed further in Section 7.1) that uses a binary search tree as its basis. Class `TreeSet<E>` provides three of the methods in interface `SearchTree`: insertion (`add`), search (`boolean contains`), and removal (`boolean remove`). It also provides an iterator that enables inorder access to the elements of a tree. Because we are only doing tree insertion and inorder access, we will use class `TreeSet<E>`.

We will write a class `IndexGenerator` (see Table 6.5) with a `TreeSet<String>` data field. Method `buildIndex` will read each word from a data file and store it in the search tree. Method `showIndex` will display the index.

TABLE 6.5
Data Fields and Methods of Class `IndexGenerator`

Data Field	Attribute
`private TreeSet<String> index`	The search tree used to store the index
`private static final String PATTERN`	Pattern for extracting words from a line. A word is a string of one or more letters or numbers or characters
Method	**Behavior**
`public void buildIndex(Scanner scan)`	Reads each word from the file scanned by `scan` and stores it in tree `index`
`public void showIndex()`	Performs an inorder traversal of tree `index`

Implementation Listing 6.8 shows class `IndexGenerator`. In method `buildIndex`, the repetition condition for the outer `while` loop calls method `hasNextLine`, which scans the next data line into a buffer associated with `Scanner scan` or returns `null` (causing loop exit) if all lines were scanned. If the next line is scanned, the repetition condition for the inner `while` loop below

```
while ((token = scan.findInLine(PATTERN)) != null) {
    token = token.toLowerCase();
    index.add(String.format("%s, %3d", token, lineNum));
}
```

calls `Scanner` method `findInLine` to extract a token from the buffer (a sequence of letters, digits, and the apostrophe character). Next, it inserts in `index` a string consisting of the next token in lowercase followed by a comma, a space, and the current line number formatted with leading spaces so that it occupies a total of three columns. This format is prescribed by the first argument `"%s, %3d"` passed to method `String.format` (see Appendix A.5). The inner loop repeats until `findInLine` returns `null`, at which point the inner loop is exited, the buffer is emptied by the statement

```
scan.nextLine();  // Clear the scan buffer
```

and the outer loop is repeated.

LISTING 6.8

Class `IndexGenerator.java`

```
import java.io.*;
import java.util.*;

/** Class to build an index. */
public class IndexGenerator {

    // Data Fields
    /** Tree for storing the index. */
    private final TreeSet<String> index;

    /** Pattern for extracting words from a line. A word is a string of
        one or more letters or numbers or ' characters */
    private static final String PATTERN =
                "[\\p{L}\\p{N}']+";

    // Methods
    public IndexGenerator() {
        index = new TreeSet<>();
    }

    /** Reads each word in a data file and stores it in an index
        along with its line number.
        post: Lowercase form of each word with its line
              number is stored in the index.
        @param scan A Scanner object
     */
    public void buildIndex(Scanner scan) {
        int lineNum = 0; // line number

        // Keep reading lines until done.
        while (scan.hasNextLine()) {
            lineNum++;
```

```
            // Extract each token and store it in index.
            String token;
            while ((token = scan.findInLine(PATTERN)) != null) {
                token = token.toLowerCase();
                index.add(String.format("%s, %3d", token, lineNum));
            }
            scan.nextLine();  // Clear the scan buffer
        }
    }

    /** Displays the index, one word per line. */
    public void showIndex() {
        index.forEach(next -> System.out.println(next));
    }
}
```

Method `showIndex` at the end of Listing 6.8 uses the the default method `forEach` to display each line of the index. We describe the `forEach` in the next syntax box. Without the `forEach`, we could use the enhanced `for` loop below with an iterator.

```
public void showIndex() {
    // Use an iterator to access and display tree data.
    for (String next : index) {
         System.out.println(next);
    }
}
```

SYNTAX Using The Java 8 forEach statement

FORM

iterable.forEach(*lambda expression*);

EXAMPLE

`index.forEach(next -> System.out.println(next));`

INTERPRETATION

Java 8 added the default method `forEach` to the `Iterable` interface. A *default method* enables you to add new functionality to an interface while still retaining compatibility with earlier implementations that did not provide this method. The `forEach` method applies a method (represented by *`lambda expression`*) to each item of an `Iterable` object. Since the `Set` interface extends the `Iterable` interface and `TreeSet` implements `Set`, we can use the `forEach` method on the index as shown in the example above.

Testing To test class `IndexGenerator`, write a `main` method that declares new `Scanner` and `IndexGenerator<String>` objects. The `Scanner` can reference any text file stored on your hard drive. Make sure that duplicate words are handled properly (including duplicates on the same line), that words at the end of each line are stored in the index, that empty lines are processed correctly, and that the last line of the document is also part of the index.

EXERCISES FOR SECTION 6.5

SELF-CHECK

1. Show the tree that would be formed for the following data items. Exchange the first and last items in each list, and rebuild the tree that would be formed if the items were inserted in the new order.
   a. happy, depressed, manic, sad, ecstatic
   b. 45, 30, 15, 50, 60, 20, 25, 90
2. Explain how the tree shown in Figure 6.13 would be changed if you inserted *mother*. If you inserted *jane*? Does either of these insertions change the height of the tree?
3. Show or explain the effect of removing the nodes *kept, cow* from the tree in Figure 6.13.
4. In Exercise 3 above, a replacement value must be chosen for the node *cow* because it has two children. What is the relationship between the replacement word and the word *cow*? What other word in the tree could also be used as a replacement for *cow*? What is the relationship between that word and the word *cow*?
5. The algorithm for deleting a node does not explicitly test for the situation where the node being deleted has no children. Explain why this is not necessary.
6. In Step 19 of the algorithm for deleting a node, when we replace the reference to a node that we are removing with a reference to its left child, why is it not a concern that we might lose the right subtree of the node that we are removing?

PROGRAMMING

1. Write methods `contains` and `remove` for the `BinarySearchTree` class. Use methods `find` and `delete` to do the work.
2. Self-Check Exercise 4 indicates that two items can be used to replace a data item in a binary search tree. Rewrite method `delete` so that it retrieves the leftmost element in the right subtree instead. You will also need to provide a method `findSmallestChild`.
3. Write a `main` method to test a binary search tree. Write a `toString` method that returns the tree contents in ascending order (using an inorder traversal) with newline characters separating the tree elements.
4. Write a `main` method for the index generator that declares new `Scanner` and `IndexGenerator` objects. The `Scanner` can reference any text file stored on your hard drive.

6.6 Heaps and Priority Queues

In this section, we discuss a binary tree that is ordered but in a different way from a binary search tree. At each level of a heap, the value in a node is less than all values in its two subtrees. Figure 6.20 shows an example of a heap. Observe that 6 is the smallest value. Observe that each parent is smaller than its children and that each parent has two children, with the exception of node 39 at level 3 and the leaves. Furthermore, with the exception of 66, all leaves are at the lowest level. Also, 39 is the next-to-last node at level 3, and 66 is the last (rightmost) node at level 3.