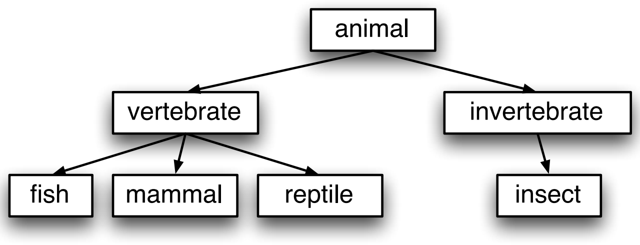

Figure 7.1: Portion of biologists' taxonomic groups

Perhaps surprisingly, it also turns out that trees are very useful in implementing plain old sequential data. In this chapter, we'll see that certain kinds of trees called *binary search trees* can be used to provide collections that allow for efficient insertion and deletion (similar to linked lists) but also allow for efficient search (similar to an ordered array). Tree-based data structures and algorithms are essential for handling large collections of data such as databases and file systems efficiently.

7.2 Tree Terminology

Computer scientists represent trees as a collection of nodes (similar to the nodes in a linked list) that are connected with *edges*. Figure 7.2 shows a tree with seven nodes, each containing an integer. The node at the very top of the diagram is called the *root*. In this tree the root contains the data value 2. A tree has exactly one root; notice that you can follow edges (the arrows) from the root to get to any other node in the tree.

Each node in a tree can have *children* connected to it via an edge. In a general tree, a node can have any number of children, but we'll only concern ourselves here with *binary trees*. In a binary tree, each node has at most two children. As you can see, the tree depicted in Figure 7.2 is a binary tree. Relationships inside the tree are described using a mixture of family and tree-like terminology. The root node has two children: the node containing 7 is its *left child*, and the one containing 6 is its *right child*. These two nodes are also said to be *siblings*. The nodes containing 8 and 4 are also siblings. The *parent* of node 5 is node 7. Node 3 is a *descendant* of node 7 and node 7 is an *ancestor* of node 3. A node that does not have any children is a *leaf* node. The *depth* of a node indicates how many edges are between it and the root node. The root node has a depth of zero. Nodes 7 and 6 have a depth of one and node 3 has a depth of three. The *height* or *depth* of a tree is the maximum depth of any node.

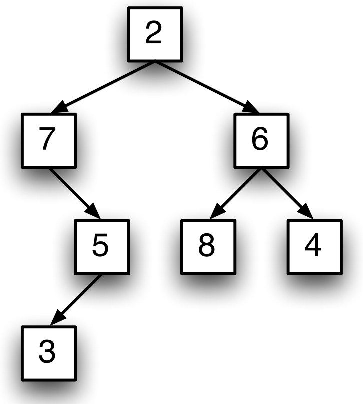

Figure 7.2: Sample binary tree

In a *full binary tree* each depth level has a node at every possible position. At the bottom level, all the nodes are leaves (i.e., all the leaves are at the same depth and every non-leaf node has two children). A *complete binary tree* has a node at every possible position except at the deepest level, and at that level, positions are filled from left to right. A complete binary tree can be created by starting with a full binary tree and adding nodes at the next level from left to right or by removing nodes at the previous level from right to left. See Figure 7.3 for examples of both.

Each node of a tree along with its descendants can be considered a *subtree*. For example, in Figure 7.2 the nodes 7, 5, and 3 can be considered a subtree of the entire tree, where node 7 is the root of the subtree. Seen in this way, a tree is naturally viewed as a recursive structure. A binary tree is either an empty tree or it consists of a root node and (possibly empty) left and right subtrees.

Just as with lists, one very useful operation on trees is traversal. Given a tree, we need a way to "walk" through the tree visiting every node in a systematic fashion. Unlike the situation with lists, there is no single, obvious way of traversing the tree. Notice that each node in the tree consists of three parts: data, left subtree, and right subtree. We have three different choices of traversal order depending on when we decide to deal with the data. If we process the data at the root and then do the left and right subtrees, we are performing a so-called *preorder traversal* because

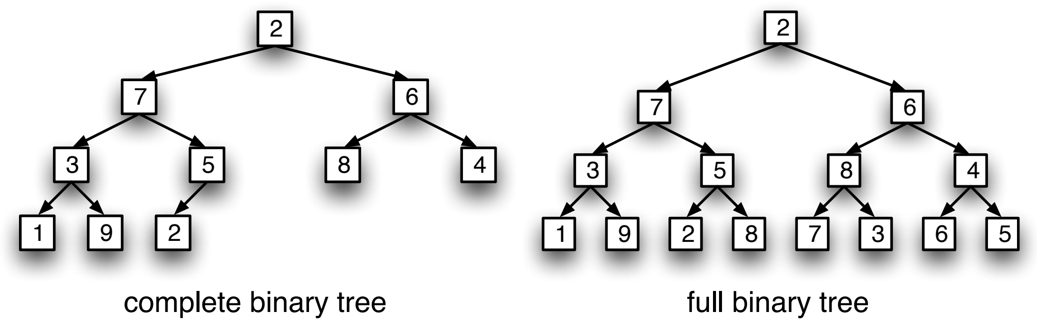

Figure 7.3: A complete binary tree on the left and a full binary tree on the right

the data at the root is considered first. A preorder traversal is easily expressed as a recursive algorithm:

```
def traverse(tree):
    if tree is not empty:
        process data at tree's root   # preorder traversal
        traverse(tree's left subtree)
        traverse(tree's right subtree)
```

Applying this algorithm to our tree from Figure 7.2 processes the nodes in the order 2, 7, 5, 3, 6, 8, 4.

Of course we can easily modify the traversal algorithm by moving where we actually process the data. An *in-order traversal* considers the root data between processing the subtrees. An in-order traversal of our sample tree yields the sequence of nodes 7, 3, 5, 2, 8, 6, 4. As you have probably guessed by now, a *postorder traversal* processes the root after the two subtrees, which gives us the ordering: 3, 5, 7, 8, 4, 6, 2.

7.3 An Example Application: Expression Trees

One important application of trees in computer science is representing the internal structure of programs. When an interpreter or compiler analyzes a program, it constructs a *parse tree* that captures the structure of the program. For example, consider a simple expression: $(2 + 3) * 4 + 5 * 6$. The form of this expression can be represented by the tree in Figure 7.4. Notice how the hierarchical structure of the tree eliminates the need for the parentheses. The basic operands of the expression end up as leaves of the tree, and the operators become internal nodes of the tree. Lower level operations in the tree have to be performed before their results are available for higher level expressions. It is clear that the addition of $2 + 3$ must be the first operation because it appears at the lowest level of the tree.

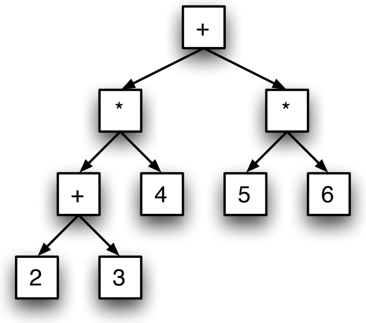

Figure 7.4: Tree representation of a mathematical expression

Given the tree structure for an expression, we can do a number of interesting things. A compiler would traverse this structure to produce a sequence of machine instructions that carry out the computation. An interpreter might use this structure to evaluate the expression. Each node is evaluated by taking the values of the two children and applying the operation. If one or both of the children is itself an operator, it will have to be evaluated first. A simple postorder traversal of the tree suffices to evaluate the expression.

```
def evaluateTree(tree):
    if tree’s root is an operand:
        return root data
    else:   # root contains an operator
        leftValue = evaluateTree(tree’s left subtree)
        rightValue = evaluateTree(tree’s right subtree)
        result = apply operator at root to leftValue and rightValue
        return result
```

If you think about it carefully, you’ll see that this is basically a recursive algorithm for evaluating the postfix version of an expression. Simply walking the expression tree in a postorder fashion yields the expression `2 3 + 4 * 5 6 * +`, which is just the postfix form of our original expression. In Chapter 5, we used a stack algorithm to evaluate postfix expressions. Here, we are implicitly using the computer's run-time stack via recursion to accomplish the same task. By the way, you can get the prefix and infix versions of the expression by doing the appropriate traversal. Isn't it fascinating how this all weaves together?

7.4 Tree Representations

Now that you've gotten a taste of what trees can do, it's time to consider some possible concrete representations for our trees. One straightforward and obvious way to build trees is to use a linked representation. Just as we did for linked lists, we can create a class to represent the nodes of our trees. Each node will have an instance variable to hold a reference to the data of the node and also variables for references to the left and right children. We'll use the `None` object for representing empty subtrees. Here's a Python class:

```
# TreeNode.py
class TreeNode(object):

    def __init__(self, data = None, left=None, right=None):

        """creates a tree node with specified data and references to left
        and right children"""

        self.item = data
        self.left = left
        self.right = right
```

Using our `TreeNode` we can easily create linked structures that directly mirror the binary tree diagrams that you've seen so far. For example, here's some code that builds a simple tree with three nodes:

```
left = TreeNode(1)
right = TreeNode(3)
root = TreeNode(2, left, right)
```

We could do the same thing with a single line of code by simply composing the calls to the `TreeNode` constructor.

```
root = TreeNode(2, TreeNode(1), TreeNode(3))
```

We can follow this approach even farther to create arbitrarily complex tree structures from our nodes. Here's some code that creates a structure similar to that of Figure 7.2.

```
root = TreeNode(2,
          TreeNode(7,
              None,
              TreeNode(5,
                  TreeNode(3),
                  None
              )
           )
           TreeNode(6,
               TreeNode(8),
               TreeNode(4)
           )
       )
```

We have used indentation to help make the layout of the expression match the structure of the tree. Notice, for example, that the root (2) has two subtrees indented under it (7 and 6). If you don't see it as a tree, try turning your head sideways.

Of course, we will not generally want to directly manipulate `TreeNode`s to build complicated structures like this. Instead, we will create a higher level container class that encapsulates the details of tree building and provides a convenient API for manipulating the tree. The exact design of the container class will depend on what we are trying to accomplish with the tree. We'll see an example of this in the next section.

We should mention that the linked representation, while obvious, is not the only possible implementation of a binary tree. In some cases, it is convenient to use an array/list-based approach. Instead of storing explicit links to children, we can maintain the relationships implicitly through positions in the array.

In the array approach, we assume that we always have a complete tree and pack the nodes into the array level by level. So, the first cell in the array stores the root, the next two positions store the root's children, the next four store the grandchildren, and so on. Following this approach, the node at position `i` always has its left child located at position `2*i+1` and its right child at position `2*i+2`. The parent of node `i` is in position `(i-1)//2`. Notice that it's crucial for these formulas that every node always has two children. You will need some special marker value (e.g., `None`) to indicate empty nodes. The array representation for the sample binary tree in Figure 7.2 is: `[2, 7, 6, None, 5, 8, 4, None, None, 3]`. If you want to simplify the calculations a bit, you can leave the first position in the array (index 0) empty and put the root at index 1. With this implementation, the left child is in position `2*i` and the right child is in position `2*i+1`, while the parent is found in position `i//2`.

The array-based tree implementation has the advantage that it does not use memory to store explicit child links. However, it does require us to waste cells for empty nodes. If the tree is sparsely filled, there will be a large number of `None` entries and the array/list implementation does not make efficient use of memory. In these cases, the linked implementation is more appropriate.

7.5 An Application: A Binary Search Tree

In this section, we're going to exercise our tree implementation techniques by building another container class for ordered sequences. Back in section 4.7 we discussed the trade-offs between linked and array-based implementations of sequences. While linked lists offer efficient insertions and deletions (since items don't have to be shifted around), they don't allow for efficient searching. A sorted array allows for efficient searching (via the binary search algorithm) but requires $\Theta(n)$ time for insertions and deletions. Using a special kind of tree, a binary search tree, we can combine the best of both worlds.

7.5.1 The Binary Search Property

A *binary search tree* is just a binary tree with an extra property that holds for every node in the tree: the values in the left subtree are less than the value at the node and the values in the right subtree are greater than the value at the node. Figure 7.5 shows a sample binary search tree.

It is usually very efficient to search for an item in a binary search tree. We start at the root of the tree and examine the data value of that node. If the root value is the one we are searching for, then we're done. If the value we are searching for is less than the value at the root, we know that if the value is in the tree, it must be in the left subtree. Similarly, if the value we are searching for is larger than the value at the root, it is in the right subtree. We can continue the search process to the appropriate subtree and apply the same rules until we find the item or reach a node that has an empty subtree where the value would be located. If the tree is reasonably well "balanced," then at each node we are essentially cutting the number of items that we have to compare against in half. That is, we are performing a binary search, which is why this is called a binary search tree.

7.5.2 Implementing A Binary Search Tree

Following good design principles, we will write a `BST` class that encapsulates all the details of the binary search tree and provides an easy-to-use interface. Our tree

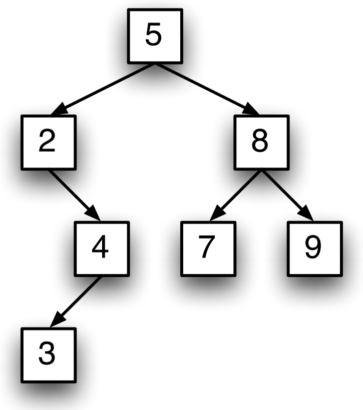

Figure 7.5: Sample binary search tree

will maintain a set of items and allow us to add, remove, and search for specific values. We're going to use a linked representation for practice with references, but you could easily convert this to the array-based implementation discussed above. A `BST` object will contain a reference to a `TreeNode` that is the root node of a binary search tree. Initially, the tree will be empty, so the reference will be to `None`. Here's our class constructor.

```
# BST.py
from TreeNode import TreeNode

class BST(object):

    def __init__(self):

        """create empty binary search tree
        post: empty tree created"""

        self.root = None
```

Now let's tackle adding items to our binary search tree. It's pretty easy to grow a tree one leaf at a time. A key point to realize is that given an existing binary search tree, there is only one location where a newly inserted item can go. Let's consider an example. Suppose we want to insert 5 into the binary search tree shown in Figure 7.6. Starting at the root node 6, we see that 5 must go in the left subtree.

The root of that tree has the value 2 so we proceed to its right subtree. The root of that subtree has the value 4 so we would proceed to its right subtree, but the right subtree is empty. The 5 is then inserted as a new leaf at that point.

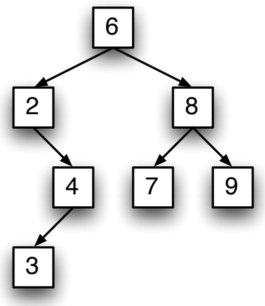

Figure 7.6: Example for inserting into a binary search tree

We can implement this basic insertion algorithm using either an iterative or a recursive approach. Either way, we start at the top of the tree and work our way down going left or right as needed until we find the spot where the new item will go. As is typical with algorithms on linked structures, we need to take some care with the special case when the structure is empty, since that requires us to change the `root` instance variable. Here's a version of the algorithm that uses a loop to march down the tree.

```
    def insert(self, item):

        """insert item into binary search tree
        pre: item is not in self
        post: item has been added to self"""

        if self.root is None:   # handle empty tree case
            self.root = TreeNode(item)
            else:
                # start at root
                node = self.root
                # loop to find the correct spot (break to exit)
                while True:
                    if item == node.item:
                        raise ValueError("Inserting duplicate item")

                    if item < node.item:           # item goes in left subtree
                        if node.left is not None:  # follow existing subtree
                            node = node.left
                        else:                      # empty subtree, insert here
                            node.left = TreeNode(item)
                            break
                    else:                          # item goes in right subtree
                        if node.right is not None: # follow existing subtree
                            node = node.right
                        else:                      # empty subtree, insert here
                            node.right = TreeNode(item)
                            break
```

This code looks rather complicated with its nested decision structures, but you should not have too much trouble following it. Notice the precondition that the item is not already in the tree. A plain binary search tree does not allow multiple copies of a value, so we check for this condition and raise an exception if an equivalent item is already in the tree. This design could easily be extended to allow multiple values by keeping a count in each node of the number of times that value has been added.

With the algorithm fresh in your mind, let's also consider how we might tackle this problem recursively. We said above that trees are a naturally recursive data structure, but our `BST` class is not really recursively structured. It is the interlinked structure of tree nodes themselves that is recursive. We can think of any node in the tree as being the root of a subtree that itself contains two smaller subtrees. A value of `None`, of course, indicates a subtree that is empty. With this insight, it's very easy to cast our insertion algorithm as a recursive method that operates on subtrees. We'll write this as a recursive helper method that is called to perform the insertion. Using this design, the `insert` method itself is trivial.

```
    def insert_rec(self, item):

        """insert item into binary search tree
        pre: item is not in self
        post: item has been added to self"""

        self.root = self._subtreeInsert(self.root, item)
```

It's important to clearly understand what `_subtreeInsert` is up to. Notice that it takes a node as the root of a subtree into which `item` must be inserted. Initially, this is the entire tree structure (`self.root`). The `_subtreeInsert` both performs the insertion and returns the node that is the root of the resulting (sub)tree. This approach makes sure that our `insert` will work even for an initially empty tree. For that case, `self.root` will start out as `None` (indicating an empty tree) and `_subtreeInsert` will return a proper `TreeNode` containing `item` that becomes the new root of the tree.

Now let's write the recursive helper function `_subtreeInsert`. The parameter to the function gives us the root of a tree structure that the item is being inserted into, and it must return the root of the resulting tree. The algorithm is very simple. If this (sub)tree is empty, we just hand back a `TreeNode` for the item, and we're done. If the tree is not empty, we modify it by recursively adding the item to either the left or right subtree, as appropriate, and return the original root of the tree as the root of the new tree (since that didn't change). Here's the code that gets the job done.

```
    def _subtreeInsert(self, root, item):

        if root is None:          # inserting into empty tree
            return TreeNode(item) # the item becomes the new tree root

        if item == root.item:
            raise ValueError("Inserting duplicate item")

        if item < root.item:                              # modify left subtree
            root.left = self._subtreeInsert(root.left, item)
        else:                                             # modify right subtree
            root.right = self._subtreeInsert(root.right, item)

        return root # original root is root of modified tree
```

So far we can create and add items to our `BST` objects, now let's work on a method to find items in the tree. We've already discussed the basic searching algorithm. It is easily implemented with a loop that walks down the tree from the root until either the item is found or we reach the bottom of the tree.

```
    def find(self, item):

        """ Search for item in BST
            post: Returns item from BST if found, None otherwise"""

        node = self.root
        while node is not None and not(node.item == item):
            if item < node.item:
                node = node.left
            else:
                node = node.right

        if node is None:
            return None
        else:
            return node.item
```

You might wonder why this method returns the item from the tree instead of just returning a Boolean value to indicate the item was found. For simplicity, our illustrations so far have used numbers, but we could store arbitrary objects in a binary search tree. The only requirement is that the objects be comparable. In general two objects might be `==` but not necessarily identical. Later on we'll see how we can exploit this property to turn our `BST` into a dictionary-like object.

For completeness, we should also add a method to our `BST` class for removing items. Removing a specific item from a binary search tree is a bit tricky. There are a number of cases that we need to consider. Let's start with the easy one. If the node to be deleted is a leaf, we can simply drop it off the tree by setting the reference in its parent node to `None`. But what if the node to delete has children? If the victim node has only a single child, our job is still straightforward. We can simply set the parent reference that used to point to the victim to point to its child instead. Figure 7.7 illustrates the situation where the left child of the victim is promoted to be the left child of the victim's parent. You might want to look at other single-child cases (there are three more) to convince yourself that this always works.

That leaves us with the problem of what to do when the victim node has two children. We can't just promote either child to take the victim's place, because that would leave the other one hanging unconnected. The solution to this dilemma is to simply leave the node in place, as we need it to maintain the structure of the tree. Instead of removing the node, we can replace its contents. We just need to find an easily deletable node whose value can be transferred into the target node while maintaining the tree's binary search property.

Consider the tree on the left side of Figure 7.8. Suppose we want to delete the 6 from this tree. What value in the tree could take its place? We could place

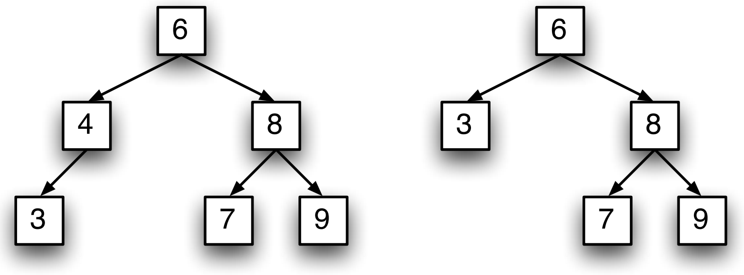

Figure 7.7: Deleting 4 from the binary search tree

either 5 or 7 in this node and the search property would be maintained. In general, it's correct to replace the victim's item with either its immediate predecessor or its immediate successor, since those values are guaranteed to stand in the same relation to the rest of the nodes in the tree. Let's say we decide to use the predecessor. We just place this value into the victim node and delete the predecessor node from the tree. Doing this gives us the tree pictured on the right side of Figure 7.8.

You might be a little concerned at this point about how we are going to delete the predecessor node. Couldn't this be just as hard as deleting the original victim? Thankfully, this is not the case. The predecessor value will always be the largest value in the victim's left subtree. Of course, to find the largest node in a binary search tree, we just march down the tree always choosing to follow links on the right. We stop when we run out of right links to follow. That means the predecessor node must have an empty right subtree, and we can always delete it by simply promoting its left subtree.

We'll again implement this algorithm using recursion on subtrees. Our top-level method just consists of a call to the recursive helper.

```
    def delete(self, item):

        """remove item from binary search tree
        post: item is removed from the tree"""

        self.root = self._subtreeDelete(self.root, item)
```

The `_subtreeDelete` method implements the heart of the deletion algorithm. It must return the root node of the subtree from which the item is removed.

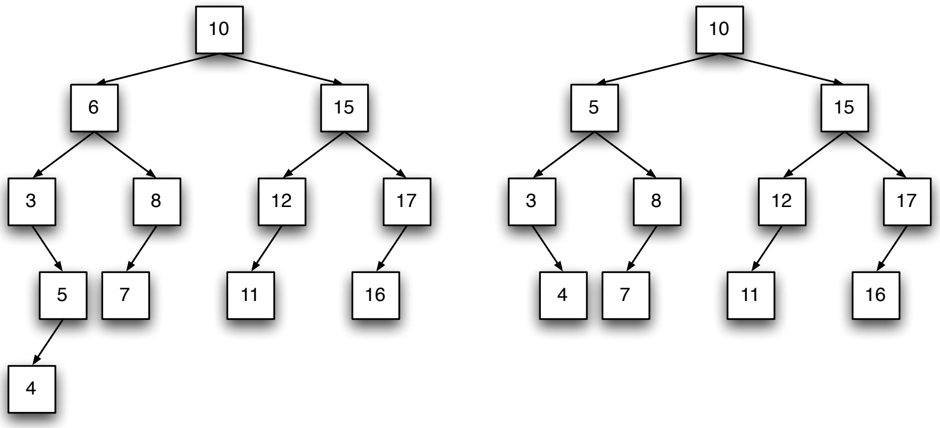

Figure 7.8: Deleting 6 from the binary search tree

```
    def _subtreeDelete(self, root, item):
        if root is None:   # Empty tree, nothing to do
           return None
        if item < root.item:                             # modify left
            root.left = self._subtreeDelete(root.left, item)
        elif item > root.item:                           # modify right
            root.right = self._subtreeDelete(root.right, item)
        else:                                            # delete root
            if root.left is None:                        # promote right subtree
                root =  root.right
            elif root.right is None:                     # promote left subtree
                root = root.left
            else:
                # overwrite root with max of left subtree
                root.item, root.left = self._subtreeDelMax(root.left)
        return root
```

As you get the hang of trees as recursive structures, this code should not be too hard to follow. If the item to delete is in the left or right subtrees, we call `_subtreeDelete` recursively to produce the modified subtree. When the root is the node to be deleted, we handle the three possible cases: promoting the right subtree, promoting the left subtree, or replacing the item with its predecessor. That last case is actually handled by another recursive method `_subtreeDelMax`. This method finds the maximum value of a tree and also deletes the node containing that value. It looks like this.

```
    def _subtreeDelMax(self, root):

        if root.right is None:           # root is the max
            return root.item, root.left  # return max and promote left subtree
        else:
            # max is in right subtree, recursively find and delete it
            maxVal, root.right = self._subtreeDelMax(root.right)
            return maxVal, root
```

7.5.3 Traversing a BST

At this point we have a useful abstraction of a set of items. We can add items to the set, find them, and delete them. All that's really missing at this point is some easy way to iterate over the collection. Given the organization of the binary search tree, an in-order traversal is particularly nice, as it produces the items in sorted order. But users of our BST class should not have to know the internal details of the data structure in order to write their own traversal algorithms. There are a number of possible ways to accomplish this.

One approach would be to simply write a traversal algorithm that assembles the items from the tree into some sequential form, say a list or a queue. We can easily write a recursive in-order traversal algorithm to produce a Python list. Here's the code to add an `asList` method to our BST class.

```
    def asList(self):

        """gets item in in-order traversal order
        post: returns list of items in tree in orders"""

        items = []
        self._subtreeAddItems(self.root, items)
        return items
```

```
    def _subtreeAddItems(self, root, itemList):

        if root is not None:
            self._subtreeAddItems(root.left, itemList)
            itemList.append(root.item)
            self._subtreeAddItems(root.right, itemList)
```

Here the helper function `_subtreeAddItems` does a basic in-order traversal of the tree where the processing of an item just requires appending it to `itemList`. You should compare this code with the generic traversal algorithm from section 7.2. Our `asList` method just creates an initial list and calls `_subtreeAddItems` to populate the list. With the addition of this method, we can easily convert a BST into a sorted list. Of course that also means we could loop over all the items in the collection. For example, we could print out the contents of our BST in order like this:

```
for item in myBST.asList():
    print item
```

The only real problem with this approach is that it produces a list that is just as large as the original collection. If the collection is huge and we are just looking for a way to loop over all of the items, producing another collection of the same size is not necessarily a good idea.

Another idea is to use a design pattern sometimes called the *visitor pattern*. The idea of this pattern is that the container provides a method that traverses the data structure and performs some client-requested function on each node. In Python, we can implement this pattern via a method that takes an arbitrary function as a parameter and applies that function to every node in the tree. We again use a recursive helper method to actually perform the traversal.

```
    def visit(self, f):

        """perform an in-order traversal of the tree
        post: calls f with each TreeNode item in an in-order traversal
        order"""

        self._inorderVisit(self.root, f)

    def _inorderVisit(self, root, f):
        if root is not None:
            self._inorderVisit(root.left, f)
            f(root.item)
            self._inorderVisit(root.right, f)
```

Notice that throughout this code, f represents some arbitrary function that the client wants applied to each item in the BST. The function is applied via the line f(root.item). Again, this is just a variation on our generic recursive-traversal algorithm.

In order to use the visit method, we just need to construct a suitable function to apply to each item. For example, if we want to print out the contents of the BST in order again, we can now do it by visiting.

```
def prnt(item):
    print item

...
myBST.visit(prnt)
```

The main thing to note here is that in the call to `visit` there are no parentheses on `prnt`. We put the parentheses on when we call a function. Here we are not actually calling the function, but rather passing the function object itself along to the `visit` method that will actually do the calling.

The visitor pattern provides a nice way for clients to perform a traversal of a container without looking through the abstraction barrier. But it is sometimes cumbersome to code an appropriate function to do the processing, and the resulting code is not very Pythonic. As with our other containers, the ideal solution in Python is to define an iterator for our `BST` using the Python generator mechanism. The basic idea is that we will just code a generic in-order traversal that `yield`s the items in the tree one at a time. By now, you should have a pretty good idea what the code will look like.

```
    def __iter__(self):

        """in-order iterator for binary search tree"""

        return self._inorderGen(self.root)

    def _inorderGen(self, root):

        if root is not None:
            # yield all the items in the left subtree
            for item in self._inorderGen(root.left):
                yield item
            yield root.item
            # yield all the items from the right subtree
            for item in self._inorderGen(root.right):
                yield item
```

The only new wrinkle in this code is the form of the recursive generator function. Remember, when you call a generator you do not get an item, rather you get an iterator object that provides items on demand. In order to actually produce the items from the left subtree, for example, we have to loop through the iterator provided by `self._inorderGen(root.left)` and yield each item.

Now we have a very convenient way of iterating through our `BST` container. Our code for printing the items in sorted order couldn't be simpler:

```
for item in myBST:
    print item
```

By the way, now that we have an iterator for the BST class, we really don't need a separate asList method. Python can produce a list of the items from a BST using the iterator via list(myBST). Being able to create a list of the items in a BST is particularly handy in writing unit tests for the BST class, as it provides an easy way to check the contents of a tree in assertions. Of course, getting a sorted list out of the BST does not guarantee that the tree has the correct form. For that, it might be helpful to have another traversal method (either pre- or postorder) as well. It's possible to deduce the true structure of a binary tree by examining two different traversals, so if both traversals come out right, you know the tree is structured the way you expect it to be.

7.5.4 A Run-time Analysis of BST Algorithms

In the introduction to this section, we suggested that a binary search tree can maintain an ordered collection quite efficiently. We've shown how a binary search tree gives us an ordered collection, but we haven't yet examined the run-time efficiency of the operations in any detail. Since many of the tree algorithms are written recursively, the analysis might seem daunting, but it's actually pretty easy if we just consider what's going on in the underlying structure.

Let's start by considering the operations that traverse the tree. Since the work we have to do at each node is constant, the time to do a traversal is just proportional to the number of nodes in the tree, which is just the number of items in the collection. That makes those operations $\Theta(n)$ where n is the size of the collection.

For the algorithms that examine only part of the tree (e.g., searching, inserting, and deleting) our analysis depends on the shape of the tree. The worst case for all of these methods requires walking a path from the root of the tree down to its "bottom." Clearly the number of steps required to do this will be proportional to the height of the tree. So the interesting question becomes how high is the tree? Of course that depends on the exact shape of the tree. Consider the tree that results from inserting a set of numbers in sorted order. The tree will end up being a linked list as each node is added as a right child of the previous number. For this tree with n elements, an insertion takes n steps to get to the bottom of the tree.

However, if the data in a tree is well distributed, then we expect that about half the items in any given subtree lie to the left and about half lie to the right. We call this a "balanced" tree. A relatively well-balanced tree will have an approximate height of $\log_2 n$. In this case, operations that have to find a particular spot in the tree will have $\Theta(\log n)$ behavior. Fortunately, if data is inserted into the tree in a random fashion, then at each node an item is equally likely to go into the left or right subtree as we work our way down from the root. On average, the result will be a nicely balanced tree.

In practice then, a binary search tree will offer very good performance, provided some care is taken in how the data is inserted and deleted. For the paranoid, there are well-known techniques (covered in section 13.3) for implementing insertion and deletion operations that guarantee the preservation of (approximately) balanced trees.

7.6 Implementing a Mapping with BST (Optional)

The BST object outlined in the previous section implements something akin to an ordered set. We can insert items, delete items, check membership, and get the items out in sorted order. Often trees are used in more database-like applications where we don't want to just ask if a particular item is in a set, but where we want to look up an item that has some particular characteristic. As a simple example, we might be maintaining a club membership list. Of course we need to be able to add and delete members from the club, but we also need something more. We need a way to bring up the record for a particular member of the club, for example to get their telephone number.

In this section we're going to take a look at how we might extend the usefulness of our binary search tree by using it to implement a general mapping similar to that provided by Python dictionaries. In our membership list example, we might use a special "key" value constructed from a member's name as a way to look up his data record. Assuming we have an appropriate membershipList object, we might get a phone number by doing something along these lines:

```
...
info = membershipList["VanRossum, Guido"]
print info.home_phone
```

Here our membershipList is an object that maps from a member's name to the corresponding record of his information. We could just use a Python dictionary for this task, but a dictionary is an unordered mapping, and we'd also like to be able to efficiently output our (huge!) membership list in sorted order.

One way to approach this problem would be rework the BST class so that all the methods take an extra parameter for the key and maintain a tree of key-value pairs. However, that's a lot more work than we really need to do. We can get a similar effect simply by using the existing BST implementation and building a wrapper around it to implement a general mapping interface. That way, we can get the advantages of a tree-based mapping object without having to modify or duplicate the BST class. Whenever possible, it's better to extend existing code than to duplicate or modify.

So how do we turn our BST from a set into a mapping? The key is to exploit the existing ordering and lookup functions of the BST class. Our existing class can be used to store any objects that are comparable. We will store items as key-value combinations, but the trick is that these items will be ordered according to just their keys. The first step is to create a new class to represent these key-value items. Let's call this combination item a KeyPair. In order to make our KeyPairs comparable, we just implement some comparison operations.

```
# KeyPair.py
class KeyPair(object):

    def __init__(self, key, value=None):
        self.key = key
        self.value = value

    def __eq__(self, other):
        return self.key == other.key

    def __lt__(self, other):
        return self.key < other.key

    def __gt__(self, other):
        return self.key > other.key
```

We have implemented only three of the six comparison operators, because all of the routines in BST use only these. For safety sake, we probably should implement the other three comparisons, just in case the BST code changes in the future. We leave this as an exercise.

Armed with this KeyPair class, we can now define a dictionary-like mapping based on BST. Here's the constructor for our class.

```
# TreeMap.py
from BST import BST
from KeyPair import KeyPair

class TreeMap(object):

    def __init__(self, items=()):
        self.items = BST()
        for key,value in items:
            self.items.insert(KeyPair(key,value))
```

We use the instance variable `items` to keep track of a `BST` that will store our `KeyPair` items. Just as a Python dictionary can be initialized with a sequence of pairs, we allow our `TreeMap` constructor to accept a sequence of pairs. We just need to loop through the pairs and call the `BST insert` operation to populate our tree. Of course, `insert` will keep the underlying binary search tree ordered according to the key values, since that's how `KeyPair`s compare to each other.

Once a `KeyPair` is in our `BST` we need to be able to retrieve it again by its key value. We can do this using the `find` operation from `BST`. The parameter we supply to the `find` operation will be a new `KeyPair` that is equivalent to (has the same key as) the one we are looking up. A line of code like this does the trick.

```
        result = self.items.find(KeyPair(key))
```

Remember that the `find` operation searches the binary search tree for an item that is `==` to the target. In this case `KeyPair(key)` "matches" the pair in the `BST` that has the same key, and it returns this matching `KeyPair`. Our partial record with just the key filled in is sufficient to retrieve the actual record for that key.

To make our `TreeMap` class work like a Python dictionary, we implement the usual Python hooks for indexing: `__getitem__` and `__setitem__`.

```
    def __getitem__(self, key):
        result = self.items.find(KeyPair(key))
        if result is None:
            raise KeyError()
        else:
            return result.value
```

```
    def __setitem__(self, key, item):
        partial = KeyPair(key)
        actual = self.items.find(partial)
        if actual is None:
            # no pair yet for this key, add one
            actual = partial
            self.items.insert(actual)

        actual.value = item
```

Each of these methods does just a little bit of extra work to handle cases when the given key is not yet in the dictionary. The `__getitem__` method just raises a `KeyError` exception in this case. When `__setitem__` is passed a new key, it needs to insert a `KeyPair` for it into the `BST`. Since we already created the new `KeyPair` `partial` to do the initial search, it's a simple matter to use it for the new entry.