In this grammar the first and second productions have an expression composed of an expression, followed by another expression, followed by an addition or multiplication token. If we tried to write a recursive function for this grammar, the base case would not come first. The recursive case would come first and hence the function would not be written correctly since the base case must come first in a recursive function. This type of production is called a left-recursive rule. Grammars with left-recursive rules are not suitable for top-down construction of a parser. There are other ways to construct parsers that are beyond the scope of this text. You can learn more about parser construction by studying a book on compiler construction or programming language implementation.

6.5 Binary Search Trees

A binary search tree is a tree where each node has up to two children. In addition, all values in the left subtree of a node are less than the value at the root of the tree and all values in the right subtree of a node are greater than or equal to the value at the root of the tree. Finally, the left and right subtrees must also be binary search trees. This definition makes it possible to write a class where values may be inserted into the tree while maintaining the definition. The code in Sect. 6.5.1 accomplishes this.

6.5.1 The BinarySearchTree Class

```
class BinarySearchTree:
    # This is a Node class that is internal to the BinarySearchTree class.
    class __Node:
        def __init__(self,val,left=None,right=None):
            self.val = val
            self.left = left
            self.right = right

        def getVal(self):
            return self.val

        def setVal(self,newval):
            self.val = newval

        def getLeft(self):
            return self.left

        def getRight(self):
            return self.right

        def setLeft(self,newleft):
            self.left = newleft

        def setRight(self,newright):
            self.right = newright

        # This method deserves a little explanation. It does an inorder traversal
        # of the nodes of the tree yielding all the values. In this way, we get
        # the values in ascending order.
        def __iter__(self):
            if self.left != None:
                for elem in self.left:
                    yield elem

            yield self.val

            if self.right != None:
                for elem in self.right:
                    yield elem

    # Below are the methods of the BinarySearchTree class.
    def __init__(self):
        self.root = None

    def insert(self,val):

            # The __insert function is recursive and is not a passed a self parameter. It is a
            # static function (not a method of the class) but is hidden inside the insert
            # function so users of the class will not know it exists.

            def __insert(root,val):
                if root == None:
                    return BinarySearchTree.__Node(val)

                if val < root.getVal():
                    root.setLeft(__insert(root.getLeft(),val))
                else:
                    root.setRight(__insert(root.getRight(),val))

                return root

        self.root = __insert(self.root,val)

    def __iter__(self):
        if self.root != None:
            return self.root.__iter__()
        else:
            return [].__iter__()

def main():
    s = input("Enter a list of numbers: ")
    lst = s.split()

    tree = BinarySearchTree()

    for x in lst:
        tree.insert(float(x))

    for x in tree:
        print(x)

if __name__ == "__main__":
    main()
```

When the program in Sect. 6.5.1 is run with a list of values (they must have an ordering) it will print the values in ascending order. For instance, if 5 8 2 1 4 9 6 7 is entered at the keyboard, the program behaves as follows.

```
Enter a list of numbers: 5 8 2 1 4 9 6 7
1.0
2.0
4.0
5.0
6.0
7.0
8.0
9.0
```

From this example it appears that a binary search tree can produce a sorted list of values when traversed. How? Let's examine how this program behaves with this input. Initially, the tree reference points to a BinarySearchTree object where the root pointer points to *None* as shown in Fig. 6.3.

Into the tree in Fig. 6.3 we insert the 5. The *insert* method is called which immediately calls the *__insert* function on the root of the tree. The *__insert* function is given a tree, which in this case is *None* (i.e. an empty tree) and the *__insert* function returns a new tree with the value inserted. The *root* instance variable is set equal to this new tree as shown in Fig. 6.4 which is the consequence of line 62 of the code in Sect. 6.5.1. In the following figures the dashed line indicates the new reference that is assigned to point to the new node. Each time the *__insert* function is called a new tree is returned and the *root* instance variable is re-assigned on line 62. Most of the time it is re-assigned to point to the same node.

Now, the next value to be inserted is the 8. Inserting the 8 calls *__insert* on the root node containing 5. When this is done, it recursively calls *__insert* on the right subtree, which is *None* (and not pictured). The result is a new right subtree is created and the right subtree link of the node containing 5 is made to point to it as shown in Fig. 6.5 which is the consequence of line 58 in Sect. 6.5.1. Again the dashed arrows indicate the new references that are assigned during the insert. It doesn't hurt anything to reassign the references and the code works very nicely. In the recursive *__insert* we always reassign the reference on lines 56 and 58 after inserting a new value into the tree. Likewise, after inserting a new value, the *root* reference is reassigned to the new tree after inserting the new value on line 62 of the code in Sect. 6.5.1.

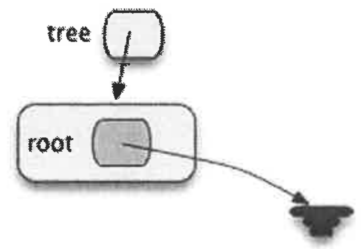

Fig. 6.3 An empty BinarySearchTree object

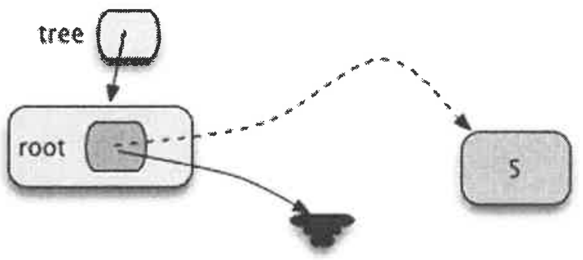

Fig. 6.4 The Tree After Inserting 5

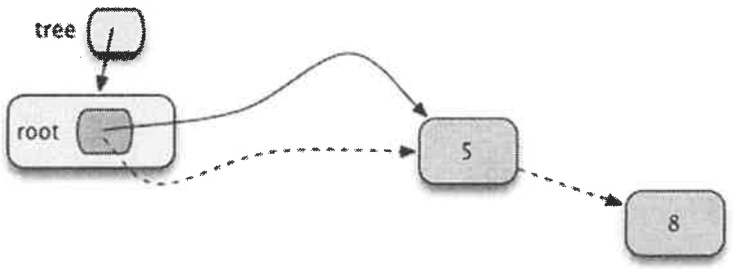

Fig. 6.5 The Tree After Inserting 8

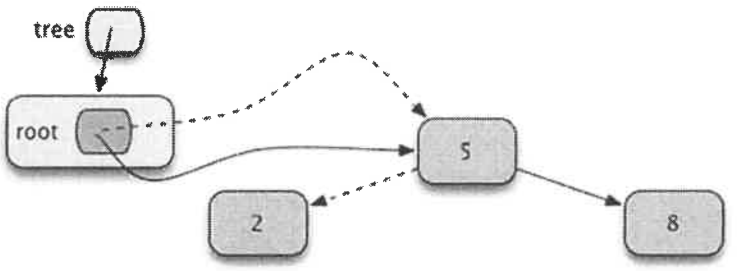

Fig. 6.6 The Tree After Inserting 2

Next, the 2 is inserted into the tree as shown in Fig. 6.6. The 8 ended up to the right of the 5 to preserve the binary search tree property. The 2 is inserted into the left subtree of the 5 because 2 is less than 5.

The 1 is inserted next and because it is less than the 5, it is inserted into the left subtree of the node containing 5. Because that subtree contains 2 the 1 is inserted into the left subtree of the node containing 2. This is depicted in Fig. 6.7.

Inserting the 4 next means the value is inserted to the left of the 5 and to the right of the 2. This preserves the binary search tree property as shown in Fig. 6.8.

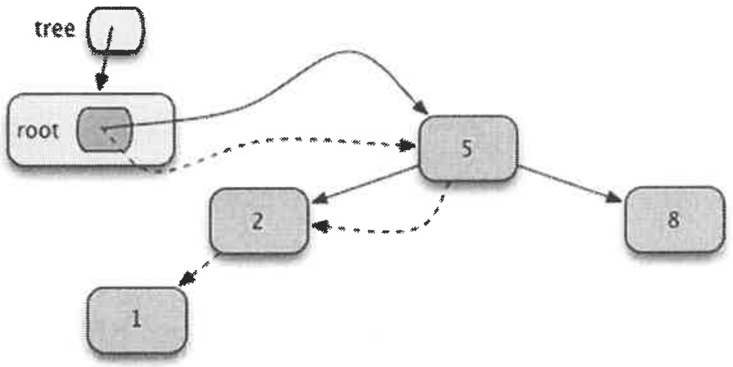

Fig. 6.7 The Tree After Inserting 1

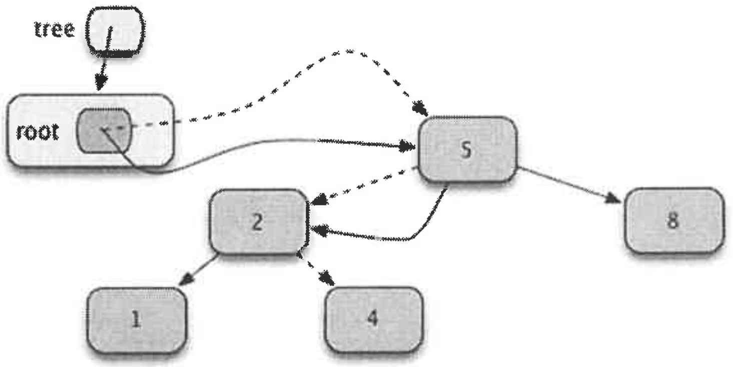

Fig. 6.8 The Tree After Inserting 4

To insert the 9 it must go to the right of all nodes inserted so far since it is greater than all nodes in the tree. This is depicted in Fig. 6.9.

The 6 goes to the right of the 5 and to the left of the 8 in Fig. 6.10.

The only place the 7 can go is to the right of the 5, left of the 8, and right of the 6 in Fig. 6.11.

The final tree is pictured in Fig. 6.12. This is a binary search tree since all nodes with subtrees have values less than the node in the left subtree and values greater than or equal to the node in the right subtree while both subtrees also conform to the binary search tree property.

The final part of the program in Sect. 6.5.1 iterates over the *tree* in the main function. This calls the *__iter__* method of the BinarySearchTree class. This *__iter__* method returns an iterator over the root's *__Node* object. The *__Node*'s *__iter__* method is interesting because it is a recursive traversal of the tree. When *for elem in self.left* is written, this calls the *__iter__* method on the left subtree. After all the elements in the left subtree are yielded, the value at the root of the tree is yielded,

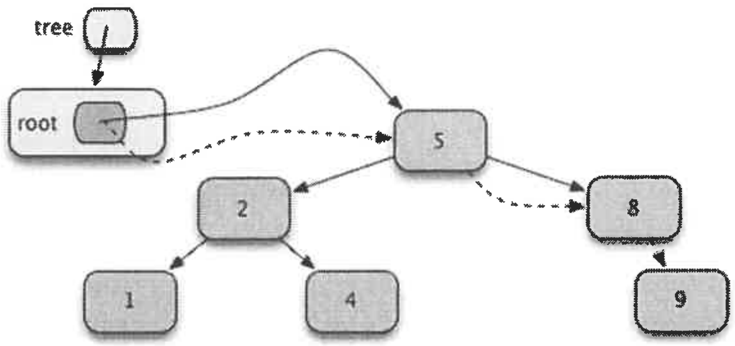

Fig. 6.9 The Tree After Inserting 9

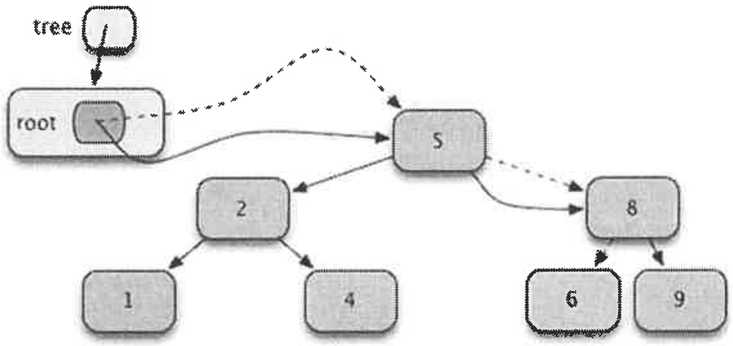

Fig. 6.10 The Tree After Inserting 6

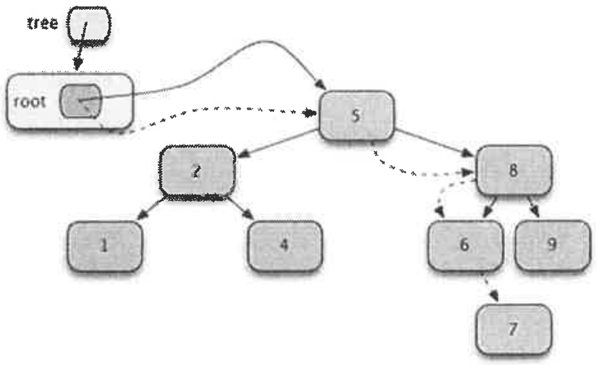

Fig. 6.11 The Tree After Inserting 7

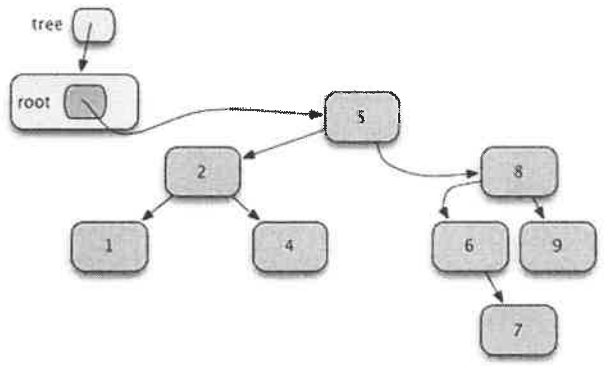

Fig. 6.12 The Final BinarySearchTree Object Contents

then the values in the right subtree are yielded by writing *for elem in self.right*. The result of this recursive function is an *inorder* traversal of the tree.

Binary search trees are of some academic interest. However, they are not used much in practice. In the average case, inserting into a binary search tree takes O(log n) time. To insert *n* items into a binary search tree would take O(n log n) time. So, in the average case we have an algorithm for sorting a sequence of ordered items. However, it takes more space than a list and the quicksort algorithm can sort a list with the same big-Oh complexity. In the worst case, binary search trees suffer from the same problem that quicksort suffers from. When the items are already sorted, both quicksort and binary search trees perform poorly. The complexity of inserting *n* items into a binary search tree becomes $O(n^2)$ in the worst case. The tree becomes a stick if the values are already sorted and essentially becomes a linked list.

There are a couple of nice properties of binary search trees that a random access list does not have. Inserting into a tree can be done in O(log n) time in the average case while inserting into a list would take O(n) time. Deleting from a binary search tree can also be done in O(log n) time in the average case. Looking up a value in a binary search tree can also be done in O(log n) time in the average case. If we have lots of insert, delete, and lookup operations for some algorithm, a tree-like structure may be useful. But, binary search trees cannot guarantee the O(log n) complexity. It turns out that there are implementations of search tree structures that can guarantee O(log n) complexity or better for inserting, deleting, and searching for values. A few examples are Splay Trees, AVL-Trees, and B-Trees which are all studied later in this text.

6.6 Search Spaces

Sometimes we have a problem that may consist of many different states. We may want to find a particular state of the problem which we'll call the *goal*. Consider Sudoku puzzles. A Sudoku puzzle has a state, reflecting how much of it we have

8. Describe a non-recursive algorithm for doing an inorder traversal of a tree. HINT: Your algorithm will need a stack to get this to work.
9. Write some code to build a tree for the infix expression $5 * 4 + 3 * 2$. Be sure to follow the precedence of operators and in your tree. You may assume the PlusNode and TimesNode classes from the chapter are already defined.
10. Provide the prefix and postfix forms of $5 * 4 + 3 * 2$.

6.9 Programming Problems

1. Write a program that asks the user to enter a prefix expression. Then, the program should print out the infix and postfix forms of that expression. Finally, it should print the result of evaluating the expression. Interacting with the program should look like this.

```
Please enter a prefix expression: + + * 4 5 6 7
The infix form is: (((4 * 5) + 6) + 7)
The postfix form is: 4 5 * 6 + 7 +
The result is: 33
```

   If the prefix expression is malformed, the program should print that the expression is malformed and it should quit. It should not try to print the infix or postfix forms of the expression in this case.
2. Write a program that reads a list of numbers from the user and lets the user insert, delete, and search for values in the tree. The program should be menu driven allowing for inserting, searching, and deleting from a binary search tree. Inserting into the tree should allow for multiple inserts as follows.

```
Binary Search Tree Program
--------------------------
Make a choice...
1. Insert into tree.
2. Delete from tree.
3. Lookup Value.
Choice? 1
insert? 5
insert? 2
insert? 8
insert? 6
insert? 7
insert? 9
insert? 4
insert? 1
insert?

Make a choice...
1. Insert into tree.
2. Delete from tree.
3. Lookup Value.
Choice? 3
Value? 8
Yes, 8 is in the tree.
```

```
Make a choice...
1. Insert into tree.
2. Delete from tree.
3. Lookup Value.
Choice? 2
Value? 5
5 has been deleted from the tree.

Make a choice...
1. Insert into tree.
2. Delete from tree.
3. Lookup Value.
Choice? 2
Value? 3
3 was not in the tree.
```

The hardest part of this program is deleting from the tree. You can write a recursive function to delete a value. In some ways, the delete from tree function is like the insert function given in the chapter. You will want to write two functions, one that is a method to call on a binary search tree to delete a value, the other would be a hidden recursive delete from tree function. The recursive function should be given a tree and a value to delete. It should return the tree after deleting the value from the tree. The recursive delete function must be handled in three cases as follows.

- **Case 1**. The value to delete is in a node that has no children. In this case, the recursive function can return an empty tree (i.e. None) because that is the tree after deleting the value from it. This would be the case if the 9 were deleted from the binary search tree in Fig. 6.12. In Fig. 6.13 the right subtree of the node containing 8 is now *None* and therefore the node containing 9 is gone from the tree.
- **Case 2**. The value to delete is in a node that has one child. In this case, the recursive function can return the child as the tree after deleting the value. This would be the case if deleting 6 from the tree in Fig. 6.13. In this case, to delete

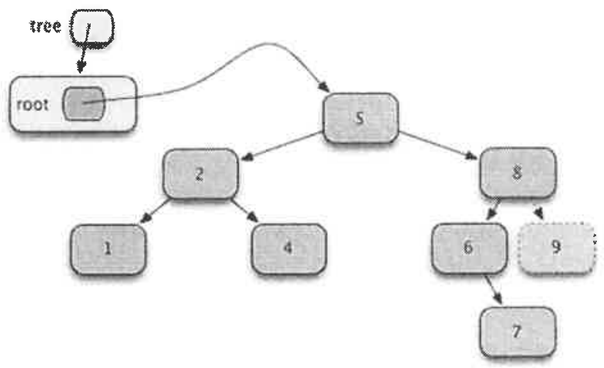

Fig. 6.13 The Tree After Deleting 9

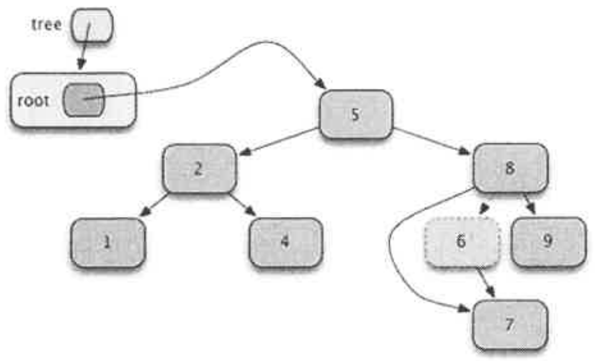

Fig. 6.14 The Tree After Deleting 6

the node containing 6 from the tree you simply return the tree for the node containing 7 so it ends up being linked to the node containing 8. In Fig. 6.14 the node containing 6 is eliminated by making the left subtree of the node containing 8 point at the right subtree of the node containing 6.

- **Case 3**. This is is hardest case to implement. When the value to delete is in a node that has two children, then to delete the node we want to use another function, call it *getRightMost*, to get the right-most value of a tree. Then you use this function to get the right-most value of the left subtree of the node to delete. Instead of deleting the node, you replace the value of the node with the right-most value of the left subtree. Then you delete the right-most value of the left subtree from the left subtree. In Fig. 6.15 the 5 is eliminated by setting the node containing 5 to 4, the right-most value of the left subtree. Then 4 is deleted from the left subtree.

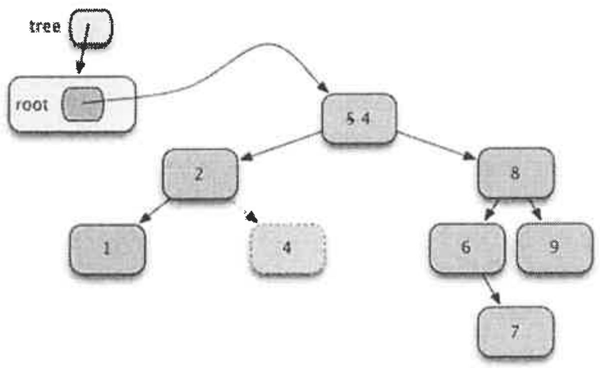

Fig. 6.15 The Tree After Deleting 5