Trees

6

When we see a tree in our everyday lives the roots are generally in the ground and the leaves are up in the air. The branches of a tree spread out from the roots in a more or less organized fashion. The word *tree* is used in Computer Science when talking about a way data may be organized. Trees have some similarities to the linked list organization found in Chap. 4. In a tree there are nodes which have links to other nodes. In a linked list each node has one link, to the next node in the list. In a tree each node may have two or more links to other nodes. A tree is not a sequential data structure. It is organized like a tree, except the root is at the top of tree data structures and the leaves are at the bottom. A tree in computer science is usually drawn inverted when compared to the trees we see in nature. There are many uses for trees in computer science. Sometimes they show the structure of a bunch of function calls as we saw when examining the Fibonacci function as depicted in Fig. 6.1.

Figure 6.1 depicts a call tree of the fib function for computing fib(5). Unlike real trees it has a *root* (at the top) and *leaves* at the bottom. There are relationships between the nodes in this tree. The fib(5) call has a left *sub-tree* and a right sub-tree. The fib(4) node is a *child* of the fib(5) node. The fib(4) node is a *sibling* to the fib(3) node to the right of it. A *leaf* node is a node with no children. The leaf nodes in Fig. 6.1 represent calls to the *fib* function which matched the base cases of the function.

In this chapter we'll explore trees and when it makes sense to build and or use a tree in a program. Not every program will need a tree data structure. Nevertheless, trees are used in many types of programs. A knowledge of them is not only a necessity, proper use of them can greatly simplify some types of programs.

6.1 Chapter Goals

This chapter introduces trees and some algorithms that use trees. By the end of the chapter you should be able to answer these questions.

K. D. Lee and S. Hubbard, *Data Structures and Algorithms with Python*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-031-42209-6_6

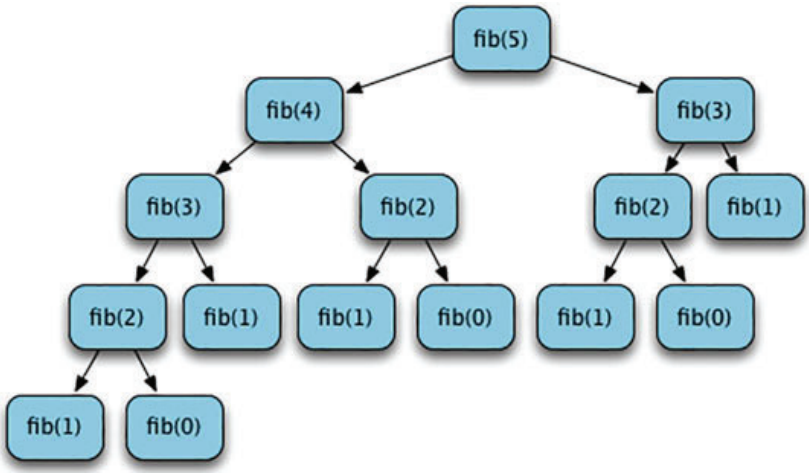

Fig. 6.1 The Call Tree for Computing fib(5)

- How are trees constructed?
- How can we traverse a tree?
- How are expressions and trees related?
- What is a binary search tree?
- Under what conditions is a binary search tree useful?
- What is depth first search and how does it relate to trees and search problems?
- What are the three types of tree traversals we can do on binary trees?
- What is a grammar and what can we do with a grammar?

Read on to discover trees and their uses in Computer Science.

6.2 Abstract Syntax Trees and Expressions

Trees have many applications in Computer Science. They are used in many different types of algorithms. For instance, every Python program you write is converted to a tree, at least for a little while, before it is executed by the Python interpreter. Internally, a Python program is converted to a tree-like structure called an *Abstract Syntax Tree*, often abbreviated *AST*, before it is executed. We can build our own abstract syntax trees for expressions so we can see how a tree might be evaluated and why we would want to evaluate a tree.

In Chap. 4 linked lists were presented as a way of organizing a list. Trees may be stored using a similar kind of structure. If a node in a tree has two children, then that node would have two links to its children as opposed to a linked list which has one link to the next node in the sequence.

Consider the expression (5 + 4) * 6 + 3. We can construct an abstract syntax tree for this expression as shown in Fig. 6.2. Since the + operation is the last operation performed when evaluating this function, the + node will be at the root of the tree.

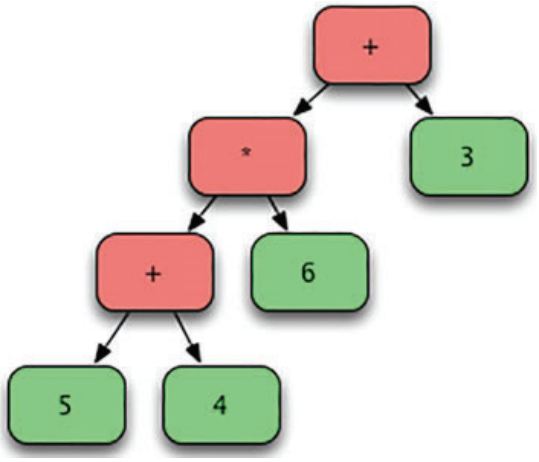

Fig. 6.2 The AST for (5 + 4) * 6 + 3

It has two subtrees, the expression to the left of the + and then 3 to the right of the +. Similarly, nodes for the other operators and operands can be constructed to yield the tree shown in Fig. 6.2.

To represent this in the computer, we could define one class for each type of node. We'll define a TimesNode, a PlusNode, and a NumNode class. So we can evaluate the abstract syntax tree, each node in the tree will have one eval method defined on it. The code in Listing 6.1 defines these classes, the eval methods, and a main function that builds the example tree in Fig. 6.2.

```
1  class TimesNode:
2      def __init__(self, left, right):
3          self.left = left
4          self.right = right
5
6      def eval(self):
7          return self.left.eval() * self.right.eval()
8
9  class PlusNode:
10     def __init__(self, left, right):
11         self.left = left
12         self.right = right
13
14     def eval(self):
15         return self.left.eval() + self.right.eval()
16
17 class NumNode:
18     def __init__(self, num):
19         self.num = num
20
21     def eval(self):
22         return self.num
23
24 def main():
25     x = NumNode(5)
26     y = NumNode(4)
27     p = PlusNode(x,y)
28     t = TimesNode(p, NumNode(6))
29     root = PlusNode(t, NumNode(3))
30
```

```
31      print(root.eval())
32
33  if __name__ == "__main__":
34      main()
```

Listing 6.1 Constructing ASTs

In Listing 6.1 the tree is built from the bottom (i.e. the leaves) up to the root. The code above contains an *eval* function for each node. Calling *eval* on the root node will recursively call *eval* on every node in the tree, causing the result, 57, to be printed to the screen.

Once an AST is built, evaluating such a tree is accomplished by doing a recursive traversal of the tree. The *eval* methods together are the recursive function in this example. We say that the *eval* methods are *mutually recursive* since all the *eval* methods together form the recursive function.

6.3 Prefix and Postfix Expressions

Expressions, as we normally write them, are said to be in infix form. An *infix expression* is an expression written with the binary operators in between their operands. Expressions can be written in other forms though. Another form for expressions is postfix. In a *postfix expression* the binary operators are written after their operands. The infix expression (5 + 4) * 6 + 3 can be written in postfix form as 5 4 + 6 * 3 +. Postfix expressions are well-suited for evaluation with a stack. When we come to an operand we push the value on the stack. When we come to an operator, we pop the operands from the stack, do the operation, and push the result. Evaluating expressions in this manner is quite easy for humans to do with a little practice. Hewlett-Packard has designed many calculators that use this postfix evaluation method. In fact, in the early years of computing, Hewlett-Packard manufactured a whole line of computers that used a stack to evaluate expressions in the same way. The HP 2000 was one such computer. In more recent times many virtual machines are implemented as stack machines including the Java Virtual Machine, or JVM, and the Python virtual machine.

As another example of a tree traversal, consider writing a method that returns a string representation of an expression. The string is built as the result of a traversal of the abstract syntax tree. To get a string representing an infix version of the expression, you perform an *inorder traversal* of the AST. To get a postfix expression you would do a *postfix* traversal of the tree. The *inorder* methods in Listing 6.2 perform an inorder traversal of an AST.

```
1  class TimesNode:
2      def __init__(self, left, right):
3          self.left = left
4          self.right = right
5
6      def eval(self):
7          return self.left.eval() * self.right.eval()
```

```
8
9       def inorder(self):
10          return "(" + self.left.inorder() + " * " + self.right.inorder() + ")"
11
12  class PlusNode:
13      def __init__(self, left, right):
14          self.left = left
15          self.right = right
16
17      def eval(self):
18          return self.left.eval() + self.right.eval()
19
20      def inorder(self):
21          return "(" + self.left.inorder() + " + " + self.right.inorder() + ")"
22
23  class NumNode:
24      def __init__(self, num):
25          self.num = num
26
27      def eval(self):
28          return self.num
29
30      def inorder(self):
31          return str(self.num)
```

Listing 6.2 AST Tree Traversal

The *inorder* methods in Listing 6.2 provide for an inorder traversal because each binary operator is added to the string *in between* the two operands. To do a postorder traversal of the tree we would write a *postorder* method that would add each binary operator to the string *after* postorder traversing the two operands. Note that because of the way a postorder traversal is written, parentheses are never needed in postfix expressions.

One other traversal is possible, called a *preorder traversal*. In a preorder traversal, each binary operator is added to the string before its two operands. Given the infix expression (5 + 4) * 6 + 3 the prefix equivalent is + * + 5 4 6 3. Again, because of the way a prefix expression is written, parentheses are never needed in prefix expressions.

6.4 Parsing Prefix Expressions

Abstract syntax trees are almost never constructed by hand. They are often built automatically by an *interpreter* or a *compiler*. When a Python program is executed the Python interpreter scans it and builds an abstract syntax tree of the program. This part of the Python interpreter is called a parser. A *parser* is a program, or part of a program, that reads a file and automatically builds an abstract syntax tree of the expression (i.e. a source program), and reports a syntax error if the program or expression is not properly formed. The exact details of how this is accomplished is beyond the scope of this text. However, for some simple expressions, like prefix expressions, it is relatively easy to build a parser ourselves.

In middle school we learned when checking to see if a sentence is *properly formed* we should use the English grammar. A grammar is a set of rules that dictate how a sentence in a language can be put together. In Computer Science we have many different languages and each language has its own grammar. Prefix expressions make up a language. We call them the language of prefix expressions and they have their own grammar, called a context-free grammar. A context-free grammar for prefix expressions is given in Sect. 6.4.1.

6.4.1 The Prefix Expression Grammar

$G = (\mathscr{N}, \mathscr{T}, \mathscr{P}, E)$ where
$\mathscr{N} = \{E\}$
$\mathscr{T} = \{identifier, number, +, *\}$
$\mathscr{P}$ is defined by the set of productions

$$E \rightarrow + \; E \; E \mid * \; E \; E \mid number$$

A grammar, G, consists of three sets: a set of non-terminals symbols denoted by N, a set of terminals or tokens called T, and a set, P, of productions. One of the non-terminals is designated the start symbol of the grammar. For this grammar, the special symbol E is the start symbol and only non-terminal of the grammar. The symbol E stands for any prefix expression. In this grammar there are three *productions* that provide the rules for how prefix expressions can be constructed. The productions state that any prefix expression *is composed of* (you can read $\rightarrow$ as *is composed of*) a plus sign followed by two prefix expressions, a multiplication symbol followed by two prefix expressions, or just a number. The grammar is recursive so every time you see E in the grammar, it can be replaced by another prefix expression. This grammar is very easy to convert to a function that given a queue of tokens will build an abstract syntax tree of a prefix expression. A function, like the E function in Listing 6.3, that reads tokens and returns an abstract syntax tree is called a *parser*. Since the grammar is recursive, the parsing function is recursive as well. It has a base case first, followed by the recursive cases. The code in Listing 6.3 provides that function.

```
1  import queue
2
3  def E(q):
4      if q.isEmpty():
5          raise ValueError("Invalid Prefix Expression")
6
7      token = q.dequeue()
8
9      if token == "+":
10         return PlusNode(E(q),E(q))
11
12     if token == "*":
13         return TimesNode(E(q),E(q))
14
15     return NumNode(float(token))
```

```
16
17 def main():
18     x = input("Please enter a prefix expression: ")
19
20     lst = x.split()
21     q = queue.Queue()
22
23     for token in lst:
24         q.enqueue(token)
25
26     root = E(q)
27
28     print(root.eval())
29     print(root.inorder())
30
31 if __name__ == "__main__":
32     main()
```

Listing 6.3 A Prefix Expression Parser

In Listing 6.3 the parameter q is a queue of the tokens read from the file or string. Code to call this function is provided in the *main* function of Listing 6.3. The *main* function gets a string from the user and enqueues all the tokens in the string (tokens must be separated by spaces) on a queue of tokens. Then the queue is passed to the function E. This function is based on the grammar given above. The function looks at the next token and decides which rule to apply. Each call to the E function returns an abstract syntax tree. Calling E from the *main* function results in parsing the prefix expression and building its corresponding tree. This example gives you a little insight into how Python reads a program and constructs an abstract syntax tree for it. A Python program is parsed according to a grammar and an abstract syntax tree is constructed from the program. The Python interpreter then interprets the program by traversing the tree.

This parser in Listing 6.3 is called a top-down parser. Not all parsers are constructed this way. The prefix grammar presented in this text is a grammar where the top-down parser construction will work. In particular, a grammar cannot have any left-recursive rules if we are to create a top-down parser for it. Left recursive rules occur in the postfix grammar given in Sect. 6.4.2.

6.4.2 The Postfix Expression Grammar

$G = (\mathscr{N}, \mathscr{T}, \mathscr{P}, E)$ where
$\mathscr{N} = \{E\}$
$\mathscr{T} = \{identifier, number, +, *\}$
$\mathscr{P}$ is defined by the set of productions

$$E \rightarrow E\ E + \mid E\ E * \mid number$$

In this grammar the first and second productions have an expression composed of an expression, followed by another expression, followed by an addition or mul-

tiplication 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 Listing 6.4 accomplishes this.

```
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()
```

Listing 6.4 The BinarySearchTree Class

When the program in Listing 6.4 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 Listing 6.4. 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 Listing 6.4. 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 Listing 6.4.

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.

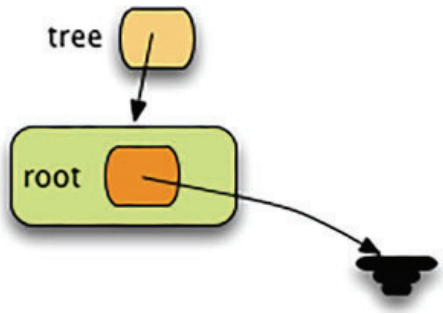

Fig. 6.3 An empty BinarySearchTree object

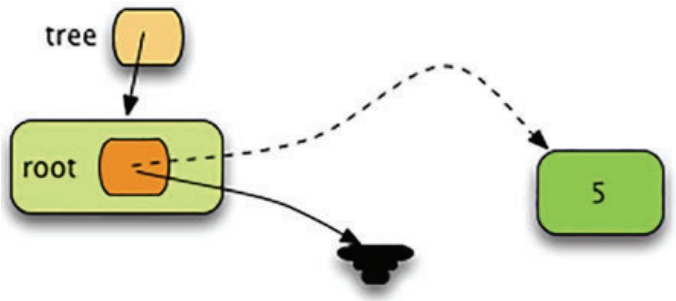

Fig. 6.4 The Tree After Inserting 5

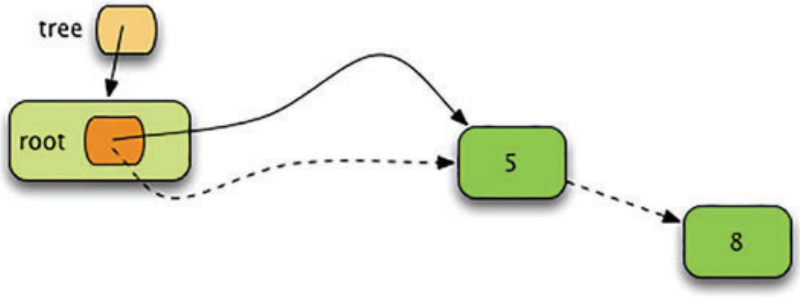

Fig. 6.5 The Tree After Inserting 8

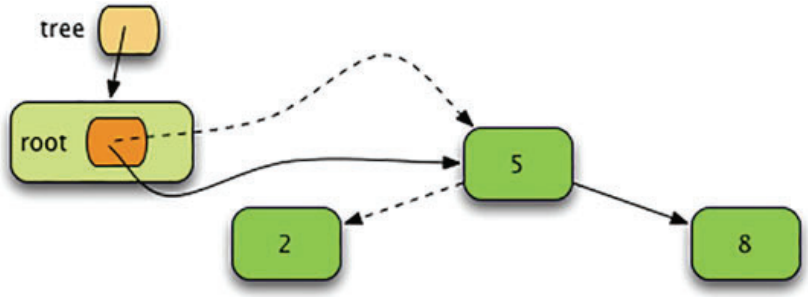

Fig. 6.6 The Tree After Inserting 2

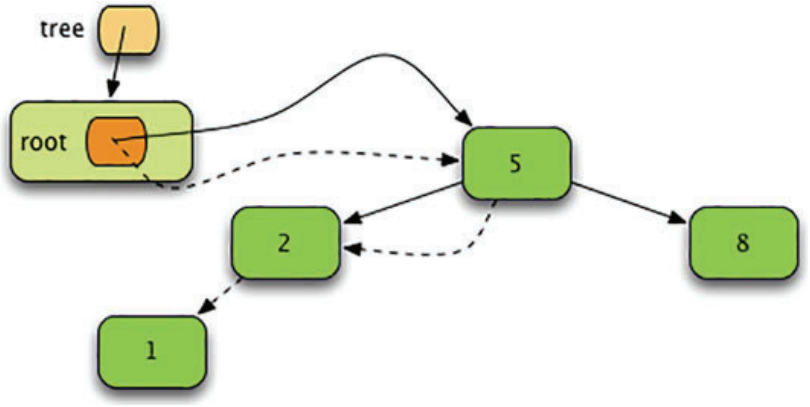

Fig. 6.7 The Tree After Inserting 1

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.

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.