Problem Solving with Algorithms and Data Structures

Release 3.0

Brad Miller, David Ranum

September 22, 2013

CONTENTS

CHAPTER ONE

INTRODUCTION

1.1 Objectives

- To review the ideas of computer science, programming, and problem-solving.
- To understand abstraction and the role it plays in the problem-solving process.
- To understand and implement the notion of an abstract data type.
- To review the Python programming language.

1.2 Getting Started

The way we think about programming has undergone many changes in the years since the first electronic computers required patch cables and switches to convey instructions from human to machine. As is the case with many aspects of society, changes in computing technology provide computer scientists with a growing number of tools and platforms on which to practice their craft. Advances such as faster processors, high-speed networks, and large memory capacities have created a spiral of complexity through which computer scientists must navigate. Throughout all of this rapid evolution, a number of basic principles have remained constant. The science of computing is concerned with using computers to solve problems.

You have no doubt spent considerable time learning the basics of problem-solving and hopefully feel confident in your ability to take a problem statement and develop a solution. You have also learned that writing computer programs is often hard. The complexity of large problems and the corresponding complexity of the solutions can tend to overshadow the fundamental ideas related to the problem-solving process.

This chapter emphasizes two important areas for the rest of the text. First, it reviews the framework within which computer science and the study of algorithms and data structures must fit, in particular, the reasons why we need to study these topics and how understanding these topics helps us to become better problem solvers. Second, we review the Python programming language. Although we cannot provide a detailed, exhaustive reference, we will give examples and explanations for the basic constructs and ideas that will occur throughout the remaining chapters.

1.3 What Is Computer Science?

Computer science is often difficult to define. This is probably due to the unfortunate use of the word "computer" in the name. As you are perhaps aware, computer science is not simply the study of computers. Although computers play an important supporting role as a tool in the discipline, they are just that – tools.

Computer science is the study of problems, problem-solving, and the solutions that come out of the problem-solving process. Given a problem, a computer scientist's goal is to develop an **algorithm**, a step-by-step list of instructions for solving any instance of the problem that might arise. Algorithms are finite processes that if followed will solve the problem. Algorithms are solutions.

Computer science can be thought of as the study of algorithms. However, we must be careful to include the fact that some problems may not have a solution. Although proving this statement is beyond the scope of this text, the fact that some problems cannot be solved is important for those who study computer science. We can fully define computer science, then, by including both types of problems and stating that computer science is the study of solutions to problems as well as the study of problems with no solutions.

It is also very common to include the word **computable** when describing problems and solutions. We say that a problem is computable if an algorithm exists for solving it. An alternative definition for computer science, then, is to say that computer science is the study of problems that are and that are not computable, the study of the existence and the nonexistence of algorithms. In any case, you will note that the word "computer" did not come up at all. Solutions are considered independent from the machine.

Computer science, as it pertains to the problem-solving process itself, is also the study of **abstraction**. Abstraction allows us to view the problem and solution in such a way as to separate the so-called logical and physical perspectives. The basic idea is familiar to us in a common example.

Consider the automobile that you may have driven to school or work today. As a driver, a user of the car, you have certain interactions that take place in order to utilize the car for its intended purpose. You get in, insert the key, start the car, shift, brake, accelerate, and steer in order to drive. From an abstraction point of view, we can say that you are seeing the logical perspective of the automobile. You are using the functions provided by the car designers for the purpose of transporting you from one location to another. These functions are sometimes also referred to as the **interface**.

On the other hand, the mechanic who must repair your automobile takes a very different point of view. She not only knows how to drive but must know all of the details necessary to carry out all the functions that we take for granted. She needs to understand how the engine works, how the transmission shifts gears, how temperature is controlled, and so on. This is known as the physical perspective, the details that take place "under the hood."

The same thing happens when we use computers. Most people use computers to write documents, send and receive email, surf the web, play music, store images, and play games without any knowledge of the details that take place to allow those types of applications to work. They view computers from a logical or user perspective. Computer scientists, programmers, technology support staff, and system administrators take a very different view of the computer. They

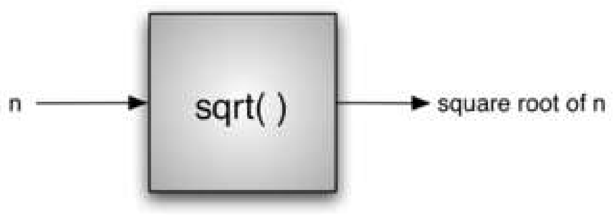


Figure 1.1: Procedural Abstraction

must know the details of how operating systems work, how network protocols are configured, and how to code various scripts that control function. They must be able to control the low-level details that a user simply assumes.

The common point for both of these examples is that the user of the abstraction, sometimes also called the client, does not need to know the details as long as the user is aware of the way the interface works. This interface is the way we as users communicate with the underlying complexities of the implementation. As another example of abstraction, consider the Python `math` module. Once we import the module, we can perform computations such as

```
>>> import math
>>> math.sqrt(16)
4.0
>>>
```

This is an example of **procedural abstraction**. We do not necessarily know how the square root is being calculated, but we know what the function is called and how to use it. If we perform the import correctly, we can assume that the function will provide us with the correct results. We know that someone implemented a solution to the square root problem but we only need to know how to use it. This is sometimes referred to as a "black box" view of a process. We simply describe the interface: the name of the function, what is needed (the parameters), and what will be returned. The details are hidden inside (see Figure 1.1).

1.3.1 What Is Programming?

Programming is the process of taking an algorithm and encoding it into a notation, a programming language, so that it can be executed by a computer. Although many programming languages and many different types of computers exist, the important first step is the need to have the solution. Without an algorithm there can be no program.

Computer science is not the study of programming. Programming, however, is an important part of what a computer scientist does. Programming is often the way that we create a representation for our solutions. Therefore, this language representation and the process of creating it becomes a fundamental part of the discipline.

Algorithms describe the solution to a problem in terms of the data needed to represent the problem instance and the set of steps necessary to produce the intended result. Programming languages must provide a notational way to represent both the process and the data. To this end, languages provide control constructs and data types.

Control constructs allow algorithmic steps to be represented in a convenient yet unambiguous way. At a minimum, algorithms require constructs that perform sequential processing, selection for decision-making, and iteration for repetitive control. As long as the language provides these basic statements, it can be used for algorithm representation.

All data items in the computer are represented as strings of binary digits. In order to give these strings meaning, we need to have **data types**. Data types provide an interpretation for this binary data so that we can think about the data in terms that make sense with respect to the problem being solved. These low-level, built-in data types (sometimes called the primitive data types) provide the building blocks for algorithm development.

For example, most programming languages provide a data type for integers. Strings of binary digits in the computer's memory can be interpreted as integers and given the typical meanings that we commonly associate with integers (e.g. 23, 654, and -19). In addition, a data type also provides a description of the operations that the data items can participate in. With integers, operations such as addition, subtraction, and multiplication are common. We have come to expect that numeric types of data can participate in these arithmetic operations.

The difficulty that often arises for us is the fact that problems and their solutions are very complex. These simple, language-provided constructs and data types, although certainly sufficient to represent complex solutions, are typically at a disadvantage as we work through the problem-solving process. We need ways to control this complexity and assist with the creation of solutions.

1.3.2 Why Study Data Structures and Abstract Data Types?

To manage the complexity of problems and the problem-solving process, computer scientists use abstractions to allow them to focus on the "big picture" without getting lost in the details. By creating models of the problem domain, we are able to utilize a better and more efficient problem-solving process. These models allow us to describe the data that our algorithms will manipulate in a much more consistent way with respect to the problem itself.

Earlier, we referred to procedural abstraction as a process that hides the details of a particular function to allow the user or client to view it at a very high level. We now turn our attention to a similar idea, that of **data abstraction**. An **abstract data type**, sometimes called an **ADT**, is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented. This means that we are concerned only with what the data is representing and not with how it will eventually be constructed. By providing this level of abstraction, we are creating an encapsulation around the data. The idea is that by encapsulating the details of the implementation, we are hiding them from the user's view. This is called **information hiding**.

Figure 1.2 shows a picture of what an abstract data type is and how it operates. The user interacts with the interface, using the operations that have been specified by the abstract data type. The abstract data type is the shell that the user interacts with. The implementation is hidden one level deeper. The user is not concerned with the details of the implementation.

The implementation of an abstract data type, often referred to as a **data structure**, will require that we provide a physical view of the data using some collection of programming constructs and primitive data types. As we discussed earlier, the separation of these two perspectives will

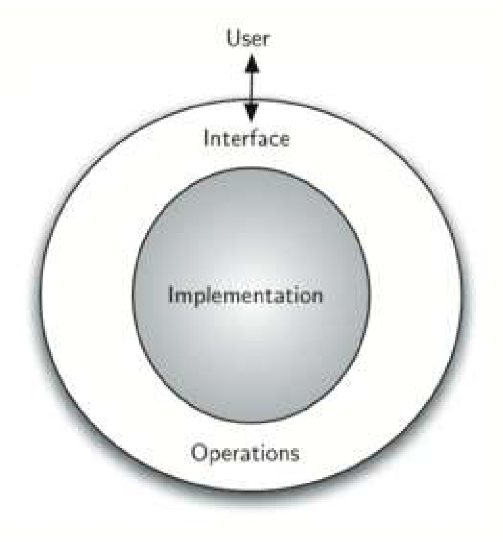


Figure 1.2: Abstract Data Type

allow us to define the complex data models for our problems without giving any indication as to the details of how the model will actually be built. This provides an **implementation-independent** view of the data. Since there will usually be many different ways to implement an abstract data type, this implementation independence allows the programmer to switch the details of the implementation without changing the way the user of the data interacts with it. The user can remain focused on the problem-solving process.

1.3.3 Why Study Algorithms?

Computer scientists learn by experience. We learn by seeing others solve problems and by solving problems by ourselves. Being exposed to different problem-solving techniques and seeing how different algorithms are designed helps us to take on the next challenging problem that we are given. By considering a number of different algorithms, we can begin to develop pattern recognition so that the next time a similar problem arises, we are better able to solve it.

Algorithms are often quite different from one another. Consider the example of `sqrt` seen earlier. It is entirely possible that there are many different ways to implement the details to compute the square root function. One algorithm may use many fewer resources than another. One algorithm might take 10 times as long to return the result as the other. We would like to have some way to compare these two solutions. Even though they both work, one is perhaps "better" than the other. We might suggest that one is more efficient or that one simply works faster or uses less memory. As we study algorithms, we can learn analysis techniques that allow us to compare and contrast solutions based solely on their own characteristics, not the characteristics of the program or computer used to implement them.

In the worst case scenario, we may have a problem that is intractable, meaning that there is no algorithm that can solve the problem in a realistic amount of time. It is important to be able to distinguish between those problems that have solutions, those that do not, and those where solutions exist but require too much time or other resources to work reasonably.

There will often be trade-offs that we will need to identify and decide upon. As computer scientists, in addition to our ability to solve problems, we will also need to know and understand solution evaluation techniques. In the end, there are often many ways to solve a problem. Finding a solution and then deciding whether it is a good one are tasks that we will do over and over again.

1.4 Review of Basic Python

In this section, we will review the programming language Python and also provide some more detailed examples of the ideas from the previous section. If you are new to Python or find that you need more information about any of the topics presented, we recommend that you consult a resource such as the Python Language Reference or a Python Tutorial. Our goal here is to reacquaint you with the language and also reinforce some of the concepts that will be central to later chapters.

Python is a modern, easy-to-learn, object-oriented programming language. It has a powerful set of built-in data types and easy-to-use control constructs. Since Python is an interpreted language, it is most easily reviewed by simply looking at and describing interactive sessions. You should recall that the interpreter displays the familiar `>>>` prompt and then evaluates the Python construct that you provide. For example,

```
>>> print("Algorithms and Data Structures")
Algorithms and Data Structures
>>>
```

shows the prompt, the `print` function, the result, and the next prompt.

1.4.1 Getting Started with Data

We stated above that Python supports the object-oriented programming paradigm. This means that Python considers data to be the focal point of the problem-solving process. In Python, as well as in any other object-oriented programming language, we define a class to be a description of what the data look like (the state) and what the data can do (the behavior). Classes are analogous to abstract data types because a user of a class only sees the state and behavior of a data item. Data items are called objects in the object-oriented paradigm. An object is an instance of a class.

Built-in Atomic Data Types

We will begin our review by considering the atomic data types. Python has two main built-in numeric classes that implement the integer and floating point data types. These Python classes are called `int` and `float`. The standard arithmetic operations, $+$, $-$, $*$, $/$, and $**$ (exponentiation), can be used with parentheses forcing the order of operations away from normal operator precedence. Other very useful operations are the remainder (modulo) operator, $\%$, and integer division, $//$. Note that when two integers are divided, the result is a floating point. The integer division operator returns the integer portion of the quotient by truncating any fractional part.

Operation Name	Operator	Explanation
less than	<	Less than operator
greater than	>	Greater than operator
less than or equal	<=	Less than or equal to operator
greater than or equal	>=	Greater than or equal to operator
equal	==	Equality operator
not equal	=!	Not equal operator
logical and	**and**	Both operands True for result to be True
logical or	**or**	Either operand True for result to be True
logical not	**not**	Negates the truth value: False becomes True, True becomes False

Table 1.1: Relational and Logical Operators

```
print(2+3*4) #14
print((2+3)*4) #20
print(2**10) #1024
print(6/3) #2.0
print(7/3) #2.33333333333
print(7//3) #2
print(7%3) #1
print(3/6) #0.5
print(3//6) #0
print(3%6) #3
print(2**100) # 1267650600228229401496703205376
```

The boolean data type, implemented as the Python `bool` class, will be quite useful for representing truth values. The possible state values for a boolean object are `True` and `False` with the standard boolean operators, `and`, `or`, and `not`.

```
>>> True
True
>>> False
False
>>> False or True
True
>>> not (False or True)
False
>>> True and True
True
```

Boolean data objects are also used as results for comparison operators such as equality (==) and greater than (>). In addition, relational operators and logical operators can be combined together to form complex logical questions. Table 1.1 shows the relational and logical operators with examples shown in the session that follows.

```
print(5 == 10)
print(10 > 5)
```

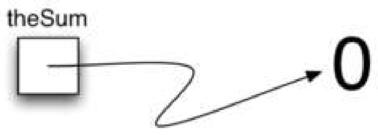


Figure 1.3: Variables Hold References to Data Objects

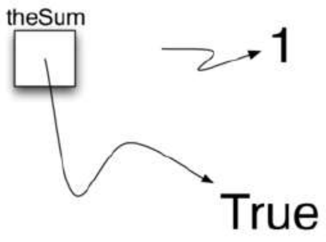


Figure 1.4: Assignment changes the Reference

```
print((5 >= 1) and (5 <= 10))
```

Identifiers are used in programming languages as names. In Python, identifiers start with a letter or an underscore (_), are case sensitive, and can be of any length. Remember that it is always a good idea to use names that convey meaning so that your program code is easier to read and understand.

A Python variable is created when a name is used for the first time on the left-hand side of an assignment statement. Assignment statements provide a way to associate a name with a value. The variable will hold a reference to a piece of data and not the data itself. Consider the following session:

```
>>> the_sum = 0
>>> the_sum
0
>>> the_sum = the_sum + 1
>>> the_sum
1
>>> the_sum = True
>>> the_sum
True
```

The assignment statement `the_sum = 0` creates a **variable** called `the_sum` and lets it hold the reference to the data object 0 (see Figure 1.3). In general, the right-hand side of the assignment statement is evaluated and a reference to the resulting data object is "assigned" to the name on the left-hand side. At this point in our example, the type of the variable is integer as that is the type of the data currently being referred to by "the_sum." If the type of the data changes (see Figure 1.4), as shown above with the boolean value True, so does the type of the variable (`the_sum` is now of the type boolean). The assignment statement changes the reference being held by the variable. This is a dynamic characteristic of Python. The same variable can refer to many different types of data.

Operation Name	Operator	Explanation
indexing	`[ ]`	Access an element of a sequence
concatenation	`+`	Combine sequences together
repetition	`*`	Concatenate a repeated number of times
membership	**`in`**	Ask whether an item is in a sequence
length	**`len`**	Ask the number of items in the sequence
slicing	`[ : ]`	Extract a part of a sequence

Table 1.2: Operations on Any Sequence in Python

Built-in Collection Data Types

In addition to the numeric and boolean classes, Python has a number of very powerful built-in collection classes. Lists, strings, and tuples are ordered collections that are very similar in general structure but have specific differences that must be understood for them to be used properly. Sets and dictionaries are unordered collections.

A list is an ordered collection of zero or more references to Python data objects. Lists are written as comma-delimited values enclosed in square brackets. The empty list is simply [ ]. Lists are heterogeneous, meaning that the data objects need not all be from the same class and the collection can be assigned to a variable as below. The following fragment shows a variety of Python data objects in a list.

```
>>> [1,3,True,6.5]
[1, 3, True, 6.5]
>>> my_list = [1,3,True,6.5]
>>> my_list
[1, 3, True, 6.5]
```

Note that when Python evaluates a list, the list itself is returned. However, in order to remember the list for later processing, its reference needs to be assigned to a variable.

Since lists are considered to be sequentially ordered, they support a number of operations that can be applied to any Python sequence. Table 1.2 reviews these operations and the following session gives examples of their use.

Note that the indices for lists (sequences) start counting with 0. The slice operation, my_list$[1 : 3]$, returns a list of items starting with the item indexed by 1 up to but not including the item indexed by 3.

Sometimes, you will want to initialize a list. This can quickly be accomplished by using repetition. For example,

```
>>> my_list = [0] * 6
>>> my_list
[0, 0, 0, 0, 0, 0]
```

One very important aside relating to the repetition operator is that the result is a repetition of references to the data objects in the sequence. This can best be seen by considering the

Method Name	Use	Explanation
`append`	`a_list.append(item)`	Adds a new item to the end of a list
`insert`	`a_list.insert(i,item)`	Inserts an item at the i^{th} position in a list
`pop`	`a_list.pop()`	Removes and returns the last item in a list
`pop`	`a_list.pop(i)`	Removes and returns the i^{th} item in a list
`sort`	`a_list.sort()`	Modifies a list to be sorted
`reverse`	`a_list.reverse()`	Modifies a list to be in reverse order
`del`	**`del`** `a_list[i]`	Deletes the item in the i^{th} position
`index`	`a_list.index(item)`	Returns the index of the first occurrence of item
`count`	`a_list.count(item)`	Returns the number of occurrences of item
`remove`	`a_list.remove(item)`	Removes the first occurrence of item

Table 1.3: Methods Provided by Lists in Python

following session:

```
my_list = [1,2,3,4]
A = [my_list]*3
print(A)
my_list[2]=45
print(A)
```

The variable **`A`** holds a collection of three references to the original list called **`my_list`**. Note that a change to one element of **`my_list`** shows up in all three occurrences in **`A`**.

Lists support a number of methods that will be used to build data structures. Table 1.3 provides a summary. Examples of their use follow.

```
my_list = [1024, 3, True, 6.5]
my_list.append(False)
print(my_list)
my_list.insert(2,4.5)
print(my_list)
print(my_list.pop())
print(my_list)
print(my_list.pop(1))
print(my_list)
my_list.pop(2)
print(my_list)
my_list.sort()
print(my_list)
my_list.reverse()
print(my_list)
print(my_list.count(6.5))
print(my_list.index(4.5))
my_list.remove(6.5)
print(my_list)
del my_list[0]
print(my_list)
```

You can see that some of the methods, such as **`pop`**, return a value and also modify the list. Others, such as **`reverse`**, simply modify the list with no return value. pop will default to the end of the list but can also remove and return a specific item. The index range starting from 0 is again used for these methods. You should also notice the familiar "dot" notation for asking an object to invoke a method. **`my_list.append(False)`** can be read as "ask the object **`my_list`** to perform its **`append`** method and send it the value **`False`**." Even simple data objects such as integers can invoke methods in this way.

```
>>> (54).__add__(21)
75
>>>
```

In this fragment we are asking the integer object **`54`** to execute its **`add`** method (called **`__add__`** in Python) and passing it **`21`** as the value to add. The result is the sum, **`75`**. Of course, we usually write this as **`54 + 21`**. We will say much more about these methods later in this section.

One common Python function that is often discussed in conjunction with lists is the **`range`** function. **`range`** produces a range object that represents a sequence of values. By using the **`list`** function, it is possible to see the value of the range object as a list. This is illustrated below.

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5,10)
range(5, 10)
>>> list(range(5,10))
[5, 6, 7, 8, 9]
>>> list(range(5,10,2))
[5, 7, 9]
>>> list(range(10,1,-1))
[10, 9, 8, 7, 6, 5, 4, 3, 2]
>>>
```

The range object represents a sequence of integers. By default, it will start with 0. If you provide more parameters, it will start and end at particular points and can even skip items. In our first example, **`range(10)`**, the sequence starts with 0 and goes up to but does not include 10. In our second example, **`range(5, 10)`** starts at 5 and goes up to but not including 10. **`range(5, 10, 2)`** performs similarly but skips by twos (again, 10 is not included).

Strings are sequential collections of zero or more letters, numbers and other symbols. We call these letters, numbers and other symbols **characters**. Literal string values are differentiated from identifiers by using quotation marks (either single or double).

```
>>> "David"
'David'
>>> my_name = "David"
>>> my_name[3]
```

Method Name	**Use**	**Explanation**
`center`	`a_string.center(w)`	Returns a string centered in a field of size w
`count`	`a_string.count(item)`	Returns the number of occurrences of item in the string
`ljust`	`a_string.ljust(w)`	Returns a string left-justified in a field of size w
`lower`	`a_string.lower()`	Returns a string in all lowercase
`rjust`	`a_string.rjust(w)`	Returns a string right-justified in a field of size w
`find`	`a_string.find(item)`	Returns the index of the first occurrence of item
`split`	`a_string.split(s_char)`	Splits a string into substrings at `s_char`

Table 1.4: Methods Provided by Strings in Python

```
'i'
>>> my_name*2
'DavidDavid'
>>> len(my_name)
5
>>>
```

Since strings are sequences, all of the sequence operations described above work as you would expect. In addition, strings have a number of methods, some of which are shown in Table 1.4. For example,

```
>>> my_name
'David'
>>> my_name.upper()
'DAVID'
>>> my_name.center(10)
'  David '
>>> my_name.find('v')
2
>>> my_name.split('v')
['Da', 'id']
>>>
```

Of these, `split` will be very useful for processing data. `split` will take a string and return a list of strings using the split character as a division point. In the example, `v` is the division point. If no division is specified, the split method looks for whitespace characters such as tab, newline and space.

A major difference between lists and strings is that lists can be modified while strings cannot. This is referred to as **mutability**. Lists are mutable; strings are immutable. For example, you can change an item in a list by using indexing and assignment. With a string that change is not allowed.

```
>>> my_list
[1, 3, True, 6.5]
>>> my_list[0]=2**10
>>> my_list
[1024, 3, True, 6.5]
>>> my_name
'David'
>>> my_name[0]='X'
Traceback (most recent call last):
  File "<pyshell#84>", line 1, in <module>
    my_name[0]='X'
TypeError: 'str' object does not support item assignment
>>>
```

Note that the error (or traceback) message displayed above is obtained on a Mac OS X machine. If you are running the above code snippet on a Windows machine, your error output will more likely be as follows.

```
>>> my_name[0]='X'
Traceback (most recent call last):
  File "<pyshell#84>", line 1, in -toplevel-
    my_name[0]='X'
TypeError: object doesn't support item assignment
>>>
```

Depending on your operating system, or version of Python, the output may slightly vary. However it will still indicate where and what the error is. You may want to experiment for yourself and get acquainted with the error message for easier and faster debugging. For the remainder of this work, we will only display the Mac OS X error messages.

Tuples are very similar to lists in that they are heterogeneous sequences of data. The difference is that a tuple is immutable, like a string. A tuple cannot be changed. Tuples are written as comma-delimited values enclosed in parentheses. As sequences, they can use any operation described above. For example,

```
>>> my_tuple = (2,True,4.96)
>>> my_tuple
(2, True, 4.96)
>>> len(my_tuple)
3
>>> my_tuple[0]
2
>>> my_tuple * 3
(2, True, 4.96, 2, True, 4.96, 2, True, 4.96)
>>> my_tuple[0:2]
(2, True)
>>>
```

However, if you try to change an item in a tuple, you will get an error. Note that the error message provides location and reason for the problem.

Operator	Use	Explanation
`in`	`x.in(set)`	Set membership
`len`	`len(set)`	Returns the cardinality (i.e. the length) of the set
`|`	`set1 | set2`	Returns a new set with all elements from both sets
`&`	`set1 & set2`	Returns a new set with only the elements common to both sets
`-`	`set1 - set2`	Returns a new set with all items from the first set not in second
`<=`	`set1 <= set2`	Asks whether all elements of the first set are in the second

Table 1.5: Operations on a Set in Python

```
>>> my_tuple[1]=False
Traceback (most recent call last):
  File "<pyshell#137>", line 1, in <module>
    my_tuple[1]=False
TypeError: 'tuple' object does not support item assignment
>>>
```

A set is an unordered collection of zero or more immutable Python data objects. Sets do not allow duplicates and are written as comma-delimited values enclosed in curly braces. The empty set is represented by **`set()`**. Sets are heterogeneous, and the collection can be assigned to a variable as below.

```
>>> {3,6,"cat",4.5,False}
{False, 4.5, 3, 6, 'cat'}
>>> my_set = {3,6,"cat",4.5,False}
>>> my_set
{False, 3, 4.5, 6, 'cat'}
>>>
```

Even though sets are not considered to be sequential, they do support a few of the familiar operations presented earlier. Table 1.5 reviews these operations and the following session gives examples of their use.

```
>>> my_set
{False, 3, 4.5, 6, 'cat'}
>>> len(my_set)
5
>>> False in my_set
True
>>> "dog" in my_set
False
>>>
```

Sets support a number of methods that should be familiar to those who have worked with them in a mathematics setting. Table 1.6 provides a summary. Examples of their use follow. Note that **`union`**, **`intersection`**, **`issubset`**, and **`difference`** all have operators that can be used as well.

Method Name	Use	Explanation
`union`	`set1.union(set2)`	Returns a new set with all elements from both sets
`intersection`	`set1.intersection(set2)`	Returns a new set with only the elements common to both sets
`difference`	`set1.difference(set2)`	Returns a new set with all items from first set not in second
`issubset`	`set1.issubset(set2)`	Asks whether all elements of one set are in the other
`add`	`set.add(item)`	Adds item to the set
`remove`	`set.remove(item)`	Removes item from the set
`pop`	`set.pop()`	Removes an arbitrary element from the set
`clear`	`set.clear()`	Removes all elements from the set

Table 1.6: Methods Provided by Sets in Python

```
>>> my_set
{False, 3, 4.5, 6, 'cat'}
>>> your_set = {99,3,100}
>>> my_set.union(your_set)
{False, 3, 4.5, 6, 99, 'cat', 100}
>>> my_set | your_set
{False, 3, 4.5, 6, 99, 'cat', 100}
>>> my_set.intersection(your_set)
{3}
>>> my_set & your_set
{3}
>>> my_set.difference(your_set)
{False, 4.5, 6, 'cat'}
>>> my_set - your_set
{False, 4.5, 6, 'cat'}
>>> {3,100}.issubset(your_set)
True
>>> {3,100} <= your_set
True
>>> my_set.add("house")
>>> my_set
{False, 3, 4.5, 6, 'house', 'cat'}
>>> my_set.remove(4.5)
>>> my_set
{False, 3, 6, 'house', 'cat'}
>>> my_set.pop()
False
>>> my_set
{3, 6, 'house', 'cat'}
>>> my_set.clear()
>>> my_set
set()
>>>
```

Operator	Use	Explanation
`[]`	`my_dict[k]`	Returns the value associated with k, otherwise its an error
`in`	`key in my_dict`	Returns True if key is in the dictionary, False otherwise
`del`	`del my_dict[key]`	Removes the entry from the dictionary

Table 1.7: Operators Provided by Dictionaries in Python

Our final Python collection is an unordered structure called a **dictionary**. Dictionaries are collections of associated pairs of items where each pair consists of a key and a value. This key-value pair is typically written as `key:value`. Dictionaries are written as comma-delimited key:value pairs enclosed in curly braces. For example,

```
>>> capitals = {'Iowa':'DesMoines','Wisconsin':'Madison'}
>>> capitals
{'Wisconsin': 'Madison', 'Iowa': 'DesMoines'}
>>>
```

We can manipulate a dictionary by accessing a value via its key or by adding another key-value pair. The syntax for access looks much like a sequence access except that instead of using the index of the item we use the key value. To add a new value is similar.

```
capitals = {'Iowa':'DesMoines','Wisconsin':'Madison'}
print(capitals['Iowa'])
capitals['Utah']='SaltLakeCity'
print(capitals)
capitals['California']='Sacramento'
print(len(capitals))
for k in capitals:
   print(capitals[k]," is the capital of ", k)
```

It is important to note that the dictionary is maintained in no particular order with respect to the keys. The first pair added `('Utah': 'SaltLakeCity')` was placed first in the dictionary and the second pair added `('California': 'Sacramento')` was placed last. The placement of a key is dependent on the idea of "hashing," which will be explained in more detail in Chapter 4. We also show the length function performing the same role as with previous collections.

Dictionaries have both methods and operators. Table 1.7 and Table 1.8 describe them, and the session shows them in action. The `keys`, `values`, and `items` methods all return objects that contain the values of interest. You can use the `list` function to convert them to lists. You will also see that there are two variations on the `get` method. If the key is not present in the dictionary, `get` will return `None`. However, a second, optional parameter can specify a return value instead.

```
>>> phone_ext={'david':1410, 'brad':1137}
>>> phone_ext
{'brad': 1137, 'david': 1410}
>>> phone_ext.keys() # Returns the keys of the dictionary phone_ext
```

Method Name	Use	Explanation
`keys`	`my_dict.keys()`	Returns the keys of the dictionary in a `dict_keys` object
`values`	`my_dict.values()`	Returns the values of the dictionary in a `dict_values` object
`items`	`my_dict.items()`	Returns the key-value pairs in a `dict_items` object
`get`	`my_dict.get(k)`	Returns the value associated with k, `None` otherwise
`get`	`my_dict.get(k,alt)`	Returns the value associated with k, *alt* otherwise

Table 1.8: Methods Provided by Dictionaries in Python

```
dict_keys(['brad', 'david'])
>>> list(phone_ext.keys())
['brad', 'david']
>>> "brad" in phone_ext
>>> True
>>> 1137 in phone_ext
>>> False          # 1137 is not a key in phone_ext
>>> phone_ext.values() # Returns the values of the dictionary
   phone_ext
dict_values([1137, 1410])
>>> list(phone_ext.values())
[1137, 1410]
>>> phone_ext.items()
dict_items([('brad', 1137), ('david', 1410)])
>>> list(phone_ext.items())
[('brad', 1137), ('david', 1410)]
>>> phone_ext.get("kent")
>>> phone_ext.get("kent","NO ENTRY")
'NO ENTRY'
>>> del phone_ext["david"]
>>> phone_ext
{'brad': 1137}
>>>
```

1.4.2 Input and Output

We often have a need to interact with users, either to get data or to provide some sort of result. Most programs today use a dialog box as a way of asking the user to provide some type of input. While Python does have a way to create dialog boxes, there is a much simpler function that we can use. Python provides us with a function that allows us to ask a user to enter some data and returns a reference to the data in the form of a string. The function is called **`input`**.

Python's input function takes a single parameter that is a string. This string is often called the **prompt** because it contains some helpful text prompting the user to enter something. For example, you might call input as follows:

```
user_name = input('Please enter your name: ')
```

Now whatever the user types after the prompt will be stored in the **user_name** variable. Using the input function, we can easily write instructions that will prompt the user to enter data and then incorporate that data into further processing. For example, in the following two statements, the first asks the user for their name and the second prints the result of some simple processing based on the string that is provided.

```
user_name = input("Please enter your name ")
print("Your name in all capitals is",user_name.upper(),
      "and has length", len(user_name))
```

It is important to note that the value returned from the **input** function will be a string representing the exact characters that were entered after the prompt. If you want this string interpreted as another type, you must provide the type conversion explicitly. In the statements below, the string that is entered by the user is converted to a float so that it can be used in further arithmetic processing.

```
user_radius = input("Please enter the radius of the circle ")
radius = float(user_radius)
diameter = 2 * radius
```

String Formatting

We have already seen that the **print** function provides a very simple way to output values from a Python program. **print** takes zero or more parameters and displays them using a single blank as the default separator. It is possible to change the separator character by setting the **sep** argument. In addition, each print ends with a newline character by default. This behavior can be changed by setting the **end** argument. These variations are shown in the following session:

```
>>> print("Hello")
Hello
>>> print("Hello","World")
Hello World
>>> print("Hello","World", sep="***")
Hello***World
>>> print("Hello","World", end="***")
Hello World***
>>> print("Hello", end="***"); print("World")
Hello***World
>>>
```

It is often useful to have more control over the look of your output. Fortunately, Python provides us with an alternative called formatted strings. A formatted string is a template in

Character	Output Format
`d,i`	Integer
`u`	Unsigned Integer
`f`	Floating point as m.ddddd
`e`	Floating point as m.dddddde+/-xx
`E`	Floating point as m.dddddE+/-xx
`g`	Use `%e` for exponents less than -4 or greater than $+5$, otherwise us `%f`
`c`	Single character
`s`	String, or any Python data object that can be converted to a string by using the **`str`** function
`%`	Insert a literal % character

Table 1.9: String Formatting Conversion Characters

which words or spaces that will remain constant are combined with placeholders for variables that will be inserted into the string. For example, the statement

```
print(name, "is", age, "years old.")
```

contains the words **`is`** and **`years old`**, but the name and the age will change depending on the variable values at the time of execution. Using a formatted string, we write the previous statement as

```
print("%s is %d years old." % (name, age))
```

This simple example illustrates a new string expression. The **`%`** operator is a string operator called the format operator. The left side of the expression holds the template or format string, and the right side holds a collection of values that will be substituted into the format string. Note that the number of values in the collection on the right side corresponds with the number of **`%`** characters in the format string. Values are taken in order, left to right from the collection and inserted into the format string.

Let's look at both sides of this formatting expression in more detail. The format string may contain one or more conversion specifications. A conversion character tells the format operator what type of value is going to be inserted into that position in the string. In the example above, the **`%s`** specifies a string, while the **`%d`** specifies an integer. Other possible type specifications include **`i, u, f, e, g, c,`** or **`%`**. Table 1.9 summarizes all of the various type specifications.

In addition to the format character, you can also include a format modifier between the % and the format character. Format modifiers may be used to left-justify or right-justify the value with a specified field width. Modifiers can also be used to specify the field width along with a number of digits after the decimal point. Table 1.10 explains these format modifiers.

The right side of the format operator is a collection of values that will be inserted into the format string. The collection will be either a tuple or a dictionary. If the collection is a tuple, the values are inserted in order of position. That is, the first element in the tuple corresponds to the first format character in the format string. If the collection is a dictionary, the values are inserted according to their keys. In this case all format characters must use the **`(name)`**

Modifier	Example	Description
number	`%20d`	Put the value in a field width of 20
-	`%-20d`	Put the value in a field 20 characters wide, left-justified
+	`%+20d`	Put the value in a field 20 characters wide, right-justified
0	`%020d`	Put the value in a field 20 characters wide, fill in with leading zeros
.	`%20.2f`	Put the value in a field 20 characters wide with 2 characters to the right of the decimal point.
`(name)`	`%(name)d`	Get the value from the supplied dictionary using `name` as the key.

Table 1.10: Additional formatting options

modifier to specify the name of the key.

```
>>> price = 24
>>> item = "banana"
>>> print("The %s costs %d cents"%(item,price))
The banana costs 24 cents
>>> print("The %+10s costs %5.2f cents"%(item,price))
The    banana costs 24.00 cents
>>> print("The %+10s costs %10.2f cents"%(item,price))
The    banana costs      24.00 cents
>>> item_dict = {"item":"banana","cost":24}
>>> print("The %(item)s costs %(cost)7.1f cents"%item_dict)
The banana costs    24.0 cents
>>>
```

In addition to format strings that use format characters and format modifiers, Python strings also include a **`format`** method that can be used in conjunction with a new **`Formatter`** class to implement complex string formatting. More about these features can be found in the Python library reference manual.

1.4.3 Control Structures

As we noted earlier, algorithms require two important control structures: iteration and selection. Both of these are supported by Python in various forms. The programmer can choose the statement that is most useful for the given circumstance.

For iteration, Python provides a standard while statement and a very powerful for statement. The while statement repeats a body of code as long as a condition is true. For example,

```
>>> counter = 1
>>> while counter <= 5:
     print("Hello, world")
     counter = counter + 1


Hello, world
Hello, world
```

```
Hello, world
Hello, world
Hello, world
>>>
```

prints out the phrase "Hello, world" five times. The condition on the `while` statement is evaluated at the start of each repetition. If the condition is `True`, the body of the statement will execute. It is easy to see the structure of a Python `while` statement due to the mandatory indentation pattern that the language enforces.

The while statement is a very general purpose iterative structure that we will use in a number of different algorithms. In many cases, a compound condition will control the iteration. A fragment such as

```
while counter <= 10 and not done:
...
```

would cause the body of the statement to be executed only in the case where both parts of the condition are satisfied. The value of the variable counter would need to be less than or equal to 10 and the value of the variable `done` would need to be `False` (`not False` is `True`) so that `True and True` results in `True`.

Even though this type of construct is very useful in a wide variety of situations, another iterative structure, the `for` statement, can be used in conjunction with many of the Python collections. The `for` statement can be used to iterate over the members of a collection, so long as the collection is a sequence. So, for example,

```
>>> for item in [1,3,6,2,5]:
     print(item)

1
3
6
2
5
>>>
```

assigns the variable `item` to be each successive value in the list $[1, 3, 6, 2, 5]$. The body of the iteration is then executed. This works for any collection that is a sequence (lists, tuples, and strings).

A common use of the `for` statement is to implement definite iteration over a range of values. The statement

```
>>> for item in range(5):
     print(item ** 2)

0
1
```

```
4
9
16
>>>
```

will perform the **print** function five times. The **range** function will return a range object representing the sequence $0, 1, 2, 3, 4$ and each value will be assigned to the variable **item**. This value is then squared and printed.

The other very useful version of this iteration structure is used to process each character of a string. The following code fragment iterates over a list of strings and for each string processes each character by appending it to a list. The result is a list of all the letters in all of the words.

```
word_list = ['cat','dog','rabbit']
letter_list = [ ]
for a_word in word_list:
   for a_letter in a_word:
      letter_list.append(a_letter)
print(letter_list)
```

Selection statements allow programmers to ask questions and then, based on the result, perform different actions. Most programming languages provide two versions of this useful construct: the **ifelse** and the **if**. A simple example of a binary selection uses the **ifelse** statement.

```
if n < 0:
   print("Sorry, value is negative")
else:
   print(math.sqrt(n))
```

In this example, the object referred to by **n** is checked to see if it is less than zero. If it is, a message is printed stating that it is negative. If it is not, the statement performs the **else** clause and computes the square root.

Selection constructs, as with any control construct, can be nested so that the result of one question helps decide whether to ask the next. For example, assume that **score** is a variable holding a reference to a score for a computer science test.

```
if score >= 90:
   print('A')
else:
   if score >= 80:
      print('B')
   else:
      if score >= 70:
         print('C')
      else:
         if score >= 60:
            print('D')
         else:
```

```
        print('F')
```

Python also has a single way selection construct, the if statement. With this statement, `if` the condition is true, an action is performed. In the case where the condition is false, processing simply continues on to the next statement after the `if`. For example, the following fragment will first check to see if the value of a variable `n` is negative. If it is, then it is modified by the absolute value function. Regardless, the next action is to compute the square root.

```
if n < 0:
  n = abs(n)
print(math.sqrt(n))
```

Returning to lists, there is an alternative method for creating a list that uses iteration and selection constructs. The is known as a **list comprehension**. A list comprehension allows you to easily create a list based on some processing or selection criteria. For example, if we would like to create a list of the first 10 perfect squares, we could use a `for` statement:

```
>>> sq_list = []
>>> for x in range(1, 11):
       sq_list.append(x * x)

>>> sq_list
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>>
```

Using a list comprehension, we can do this in one step as

```
>>> sq_list = [x * x for x in range(1, 11)]
>>> sq_list
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>>
```

The variable `x` takes on the values 1 through 10 as specified by the `for` construct. The value of `x * x` is then computed and added to the list that is being constructed. The general syntax for a list comprehension also allows a selection criteria to be added so that only certain items get added. For example,

```
>>> sq_list = [x * x for x in range(1, 11) if x % 2 != 0]
>>> sq_list
[1, 9, 25, 49, 81]
>>>
```

This list comprehension constructed a list that only contained the squares of the odd numbers in the range from 1 to 10. Any sequence that supports iteration can be used within a list comprehension to construct a new list.

```
>>>[ch.upper() for ch in 'comprehension' if ch not in 'aeiou']
```

```
['C', 'M', 'P', 'R', 'H', 'N', 'S', 'N']
>>>
```

Self Check

Test your understanding of what we have covered so far by trying the following two exercises. Use the code below, seen earlier in this subsection.

```
word_list = ['cat','dog','rabbit']
letter_list = [ ]
for a_word in word_list:
   for a_letter in a_word:
      letter_list.append(a_letter)
print(letter_list)
```

1. Modify the given code so that the final list only contains a single copy of each letter.

   ```
   # the answer is: ['c', 'a', 't', 'd', 'o', 'g', 'r', 'b', 'i']
   ```

2. Redo the given code using list comprehensions. For an extra challenge, see if you can figure out how to remove the duplicates.

   ```
   # the answer is: ['c', 'a', 't', 'd', 'o', 'g', 'r', 'a',
      'b', 'b', 'i', 't']
   ```

1.4.4 Exception Handling

There are two types of errors that typically occur when writing programs. The first, known as a syntax error, simply means that the programmer has made a mistake in the structure of a statement or expression. For example, it is incorrect to write a for statement and forget the colon.

```
>>> for i in range(10)
SyntaxError: invalid syntax
>>>
```

In this case, the Python interpreter has found that it cannot complete the processing of this instruction since it does not conform to the rules of the language. Syntax errors are usually more frequent when you are first learning a language.

The other type of error, known as a logic error, denotes a situation where the program executes but gives the wrong result. This can be due to an error in the underlying algorithm or an error in your translation of that algorithm. In some cases, logic errors lead to very bad situations such as trying to divide by zero or trying to access an item in a list where the index of the item is outside the bounds of the list. In this case, the logic error leads to a runtime error that causes the program to terminate. These types of runtime errors are typically called **exceptions**.

Most of the time, beginning programmers simply think of exceptions as fatal runtime errors that cause the end of execution. However, most programming languages provide a way to deal with these errors that will allow the programmer to have some type of intervention if they so choose. In addition, programmers can create their own exceptions if they detect a situation in the program execution that warrants it.

When an exception occurs, we say that it has been "raised." You can "handle" the exception that has been raised by using a **`try`** statement. For example, consider the following session that asks the user for an integer and then calls the square root function from the math library. If the user enters a value that is greater than or equal to 0, the print will show the square root. However, if the user enters a negative value, the square root function will report a **`ValueError`** exception.

```
>>> a_number = int(input("Please enter an integer "))
Please enter an integer -23
>>> print(math.sqrt(a_number))
Traceback (most recent call last):
  File "<pyshell#102>", line 1, in <module>
    print(math.sqrt(a_number))
ValueError: math domain error
>>>
```

We can handle this exception by calling the print function from within a try block. A corresponding **`except`** block "catches" the exception and prints a message back to the user in the event that an exception occurs. For example:

```
>>> try:
      print(math.sqrt(a_number))
    except:
      print("Bad Value for square root")
      print("Using absolute value instead")
      print(math.sqrt(abs(a_number)))


Bad Value for square root
Using absolute value instead
4.795831523312719
>>>
```

will catch the fact that an exception is raised by **`sqrt`** and will instead print the messages back to the user and use the absolute value to be sure that we are taking the square root of a non-negative number. This means that the program will not terminate but instead will continue on to the next statements.

It is also possible for a programmer to cause a runtime exception by using the **`raise`** statement. For example, instead of calling the square root function with a negative number, we could have checked the value first and then raised our own exception. The code fragment below shows the result of creating a new **`RuntimeError`** exception. Note that the program would still terminate but now the exception that caused the termination is something explicitly created by the programmer.

```
>>> if a_number < 0:
...   raise RuntimeError("You can't use a negative number")
... else:
...   print(math.sqrt(a_number))
...
Traceback (most recent call last):
  File "<pyshell#20>", line 2, in <module>
    raise RuntimeError("You can't use a negative number")
RuntimeError: You can't use a negative number
>>>
```

There are many kinds of exceptions that can be raised in addition to the `RuntimeError` shown above. See the Python reference manual for a list of all the available exception types and for how to create your own.

1.4.5 Defining Functions

The earlier example of procedural abstraction called upon a Python function called `sqrt` from the math module to compute the square root. In general, we can hide the details of any computation by defining a function. A function definition requires a name, a group of parameters, and a body. It may also explicitly return a value. For example, the simple function defined below returns the square of the value you pass into it.

```
>>> def square(n):
...   return n ** 2
...
>>> square(3)
9
>>> square(square(3))
81
>>>
```

The syntax for this function definition includes the name, `square`, and a parenthesized list of formal parameters. For this function, `n` is the only formal parameter, which suggests that `square` needs only one piece of data to do its work. The details, hidden "inside the box," simply compute the result of `n ** 2` and return it. We can invoke or call the `square` function by asking the Python environment to evaluate it, passing an actual parameter value, in this case, `3`. Note that the call to `square` returns an integer that can in turn be passed to another invocation.

We could implement our own square root function by using a well-known technique called "Newton's Method." Newton's Method for approximating square roots performs an iterative computation that converges on the correct value. The equation

$$new\_guess = \frac{1}{2} * (\frac{old\_guess + n}{old\_guess})$$

takes a value n and repeatedly guesses the square root by making each *new_guess* the *old_guess* in the subsequent iteration. The initial guess used here is $\frac{n}{2}$. Listing 1.1 shows a function definition that accepts a value n and returns the square root of n after making 20 guesses. Again, the details of Newton's Method are hidden inside the function definition and the user does not have to know anything about the implementation to use the function for its intended purpose. Listing 1.1 also shows the use of the # character as a comment marker. Any characters that follow the # on a line are ignored.

Listing 1.1: square_root Function

```
def square_root(n):
   root = n / 2 #initial guess will be 1/2 of n
   for k in range(20):
      root = (1 / 2) * (root + (n / root))

   return root

>>>square_root(9)
3.0
>>>square_root(4563)
67.549981495186216
>>>
```

Self Check

Here is a self check that really covers everything so far. You may have heard of the infinite monkey theorem? The theorem states that a monkey hitting keys at random on a typewriter keyboard for an infinite amount of time will almost surely type a given text, such as the complete works of William Shakespeare. Well, suppose we replace a monkey with a Python function. How long do you think it would take for a Python function to generate just one sentence of Shakespeare? The sentence we'll shoot for is: "methinks it is like a weasel"

You are not going to want to run this one in the browser, so fire up your favorite Python IDE. The way we will simulate this is to write a function that generates a string that is 27 characters long by choosing random letters from the 26 letters in the alphabet plus the space. We will write another function that will score each generated string by comparing the randomly generated string to the goal.

A third function will repeatedly call generate and score, then if 100% of the letters are correct we are done. If the letters are not correct then we will generate a whole new string. To make it easier to follow your program's progress this third function should print out the best string generated so far and its score every 1000 tries.

Self Check Challenge

See if you can improve upon the program in the self check by keeping letters that are correct and only modifying one character in the best string so far. This is a type of algorithm in the class of “hill climbing” algorithms, that is we only keep the result if it is better than the previous one.

1.4.6 Object-Oriented Programming in Python: Defining Classes

We stated earlier that Python is an object-oriented programming language. So far, we have used a number of built-in classes to show examples of data and control structures. One of the most powerful features in an object-oriented programming language is the ability to allow a programmer (problem solver) to create new classes that model data that is needed to solve the problem.

Remember that we use abstract data types to provide the logical description of what a data object looks like (its state) and what it can do (its methods). By building a class that implements an abstract data type, a programmer can take advantage of the abstraction process and at the same time provide the details necessary to actually use the abstraction in a program. Whenever we want to implement an abstract data type, we will do so with a new class.

A Fraction Class

A very common example to show the details of implementing a user-defined class is to construct a class to implement the abstract data type `Fraction`. We have already seen that Python provides a number of numeric classes for our use. There are times, however, that it would be most appropriate to be able to create data objects that “look like” fractions.

A fraction such as $\frac{3}{5}$ consists of two parts. The top value, known as the numerator, can be any integer. The bottom value, called the denominator, can be any integer greater than 0 (negative fractions have a negative numerator). Although it is possible to create a floating point approximation for any fraction, in this case we would like to represent the fraction as an exact value.

The operations for the `Fraction` type will allow a `Fraction` data object to behave like any other numeric value. We need to be able to add, subtract, multiply, and divide fractions. We also want to be able to show fractions using the standard “slash” form, for example $\frac{3}{5}$. In addition, all fraction methods should return results in their lowest terms so that no matter what computation is performed, we always end up with the most common form.

In Python, we define a new class by providing a name and a set of method definitions that are syntactically similar to function definitions. For this example,

```
class Fraction:

   #the methods go here
```

provides the framework for us to define the methods. The first method that all classes should provide is the constructor. The constructor defines the way in which data objects are created. To create a `Fraction` object, we will need to provide two pieces of data, the numerator and the denominator. In Python, the constructor method is always called __`init`__(two

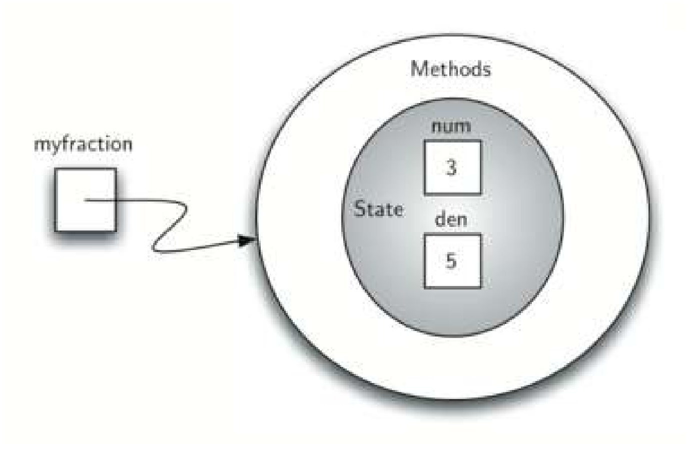


Figure 1.5: An Instance of the Fraction class

underscores before and after **init**) and is shown in Listing 1.2.

Listing 1.2: Fraction Class and its constructor

```
class Fraction:

   def __init__(self,top,bottom):

      self.num = top
      self.den = bottom
```

Notice that the formal parameter list contains three items (**self**, **top**, **bottom**). **self** is a special parameter that will always be used as a reference back to the object itself. It must always be the first formal parameter; however, it will never be given an actual parameter value upon invocation. As described earlier, fractions require two pieces of state data, the numerator and the denominator. The notation **self.num** in the constructor defines the **fraction** object to have an internal data object called **num** as part of its state. Likewise, **self.den** creates the denominator. The values of the two formal parameters are initially assigned to the state, allowing the new fraction object to know its starting value.

To create an instance of the **Fraction** class, we must invoke the constructor. This happens by using the name of the class and passing actual values for the necessary state (note that we never directly **invoke** **__init__**). For example,

```
my_fraction = Fraction(3,5)
```

creates an object called **my_fraction** representing the fraction $\frac{3}{5}$ (three-fifths). Figure 1.5 shows this object as it is now implemented.

The next thing we need to do is implement the behavior that the abstract data type requires. To begin, consider what happens when we try to print a **Fraction** object.

```
>>> my_f = Fraction(3, 5)
>>> print(my_f)
```

```
<__main__.Fraction object at 0x409b1acc>
```

The `Fraction` object, `my_f`, does not know how to respond to this request to print. The `print` function requires that the object convert itself into a string so that the string can be written to the output. The only choice `my_f` has is to show the actual reference that is stored in the variable (the address itself). This is not what we want.

There are two ways we can solve this problem. One is to define a method called `show` that will allow the `Fraction` object to print itself as a string. We can implement this method as shown in Listing 1.3. If we create a `Fraction` object as before, we can ask it to show itself, in other words, print itself in the proper format. Unfortunately, this does not work in general. In order to make printing work properly, we need to tell the `Fraction` class how to convert itself into a string. This is what the `print` function needs in order to do its job.

Listing 1.3: Show Function

```
def show(self):
     print(self.num, "/", self.den)
>>> my_f = Fraction(3, 5)
>>> my_f.show()
3 / 5
>>> print(my_f)
<__main__.Fraction object at 0x40bce9ac>
>>>
```

In Python, all classes have a set of standard methods that are provided but may not work properly. One of these, `__str__`, is the method to convert an object into a string. The default implementation for this method is to return the instance address string as we have already seen. What we need to do is provide a "better" implementation for this method. We will say that this implementation **overrides** the previous one, or that it redefines the method's behavior.

To do this, we simply define a method with the name `__str__` and give it a new implementation as shown in Listing 1.4. This definition does not need any other information except the special parameter `self`. In turn, the method will build a string representation by converting each piece of internal state data to a string and then placing a / character in between the strings using string concatenation. The resulting string will be returned any time a `Fraction` object is asked to convert itself to a string. Notice the various ways that this function is used.

Listing 1.4: Standard Method

```
def __str__ (self):
    return str(self.num) + "/" + str(self.den)

>>> my_f = Fraction(3, 5)
>>> print(my_f)
3/5
>>> print("I ate", my_f, "of the pizza")
I ate 3/5 of the pizza
>>> my_f.__str__()
```

```
'3/5'
>>> str(my_f)
'3/5'
>>>
```

We can override many other methods for our new **Fraction** class. Some of the most important of these are the basic arithmetic operations. We would like to be able to create two **Fraction** objects and then add them together using the standard "+" notation. At this point, if we try to add two fractions, we get the following:

```
>>> f1 = Fraction(1,4)
>>> f2 = Fraction(1,2)
>>> f1 + f2
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    f1 + f2
TypeError: unsupported operand type(s) for +: 'Fraction' and
    'Fraction'
>>>
```

If you look closely at the error, you see that the problem is that the "+" operator does not understand the Fraction operands. We can fix this by providing the **Fraction** class with a method that overrides the addition method. In Python, this method is called **__add__** and it requires two parameters. The first, **self**, is always needed, and the second represents the other operand in the expression. For example,

```
f1.__add__(f2)
```

would ask the **Fraction** object **f1** to add the **Fraction** object **f2** to itself. This can be written in the standard notation, **f1 + f2**.

Two fractions must have the same denominator to be added. The easiest way to make sure they have the same denominator is to simply use the product of the two denominators as a common denominator so that

$$\frac{a}{b} + \frac{c}{d} = \frac{ad}{bd} + \frac{cb}{bd} = \frac{ad + cb}{bd}$$

The implementation is shown in Listing 1.5. The addition function returns a new **Fraction** object with the numerator and denominator of the sum. We can use this method by writing a standard arithmetic expression involving fractions, assigning the result of the addition, and then printing our result.

Listing 1.5: Adding Fractions

```
def __add__(self, other_fraction):

    new_num = self.num*other_fraction.den +
       self.den*other_fraction.num
    new_den = self.den * other_fraction.den
    return Fraction(new_num, new_den)
```

```
>>> f1 = Fraction(1, 4)
>>> f2 = Fraction(1, 2)
>>> f3 = f1 + f2
>>> print(f3)
6/8
>>>
```

The addition method works as we desire, but one thing could be better. Note that $\frac{6}{8}$ is the correct result ($\frac{1}{4} + 12$) but that it is not in the "lowest terms" representation. The best representation would be $\frac{3}{4}$. In order to be sure that our results are always in the lowest terms, we need a helper function that knows how to reduce fractions. This function will need to look for the greatest common divisor, or GCD. We can then divide the numerator and the denominator by the GCD and the result will be reduced to lowest terms.

The best-known algorithm for finding a greatest common divisor is Euclid's Algorithm. Euclid's Algorithm states that the greatest common divisor of two integers m and n is n if n divides m evenly. However, if n does not divide m evenly, then the answer is the greatest common divisor of n and the remainder of m divided by n. We will simply provide an iterative implementation here. Note that this implementation of the GCD algorithm only works when the denominator is positive. This is acceptable for our fraction class because we have said that a negative fraction will be represented by a negative numerator.

```
def gcd(m, n):
   while m % n != 0:
      old_m = m
      old_n = n

      m = old_n
      n = old_m % old_n
   return n

print(gcd(20, 10))
```

Now we can use this function to help reduce any fraction. To put a fraction in lowest terms, we will divide the numerator and the denominator by their greatest common divisor. So, for the fraction $\frac{6}{8}$, the greatest common divisor is 2. Dividing the top and the bottom by 2 creates a new fraction, $\frac{3}{4}$ (see Listing 1.6).

Listing 1.6: Lowest Term Fractions

```
def __add__(self, other_fraction):
   new_num = self.num*other_fraction.den +
      self.den*other_fraction.num
   new_den = self.den * other_fraction.den
   common = gcd(new_num, new_den)
   return Fraction(new_num // common, new_den // common)
```

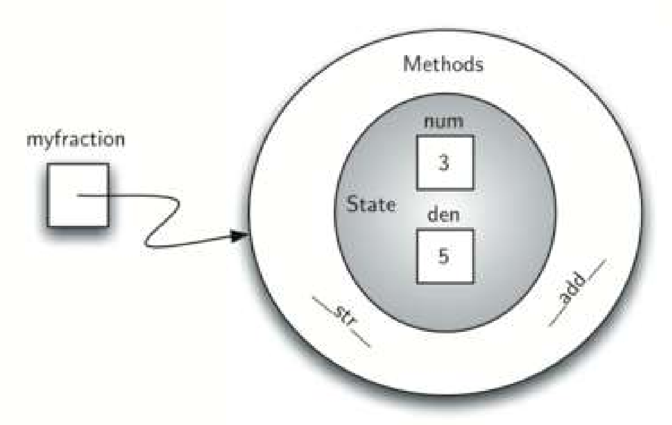


Figure 1.6: An Instance of the Fraction Class with Two Methods

```
>>> f1 = Fraction(1, 4)
>>> f2 = Fraction(1, 2)
>>> f3 = f1 + f2
>>> print(f3)
3/4
>>>
```

Our **`Fraction`** object now has two very useful methods and looks like Figure 1.6. An additional group of methods that we need to include in our example **`Fraction`** class will allow two fractions to compare themselves to one another. Assume we have two Fraction objects, **`f1`** and **`f2`**. **`f1 == f2`** will only be **`True`** if they are references to the same object. Two different objects with the same numerators and denominators would not be equal under this implementation. This is called **shallow equality** (see Figure 1.7).

We can create deep equality (see Figure 1.7) – equality by the same value, not the same reference – by overriding the **`__eq__`** method. The **`__eq__`** method is another standard method available in any class. The **`__eq__`** method compares two objects and returns **`True`** if their values are the same, False otherwise.

In the **`Fraction`** class, we can implement the **`__eq__`** method by again putting the two fractions in common terms and then comparing the numerators (see Listing 1.7). It is important to note that there are other relational operators that can be overridden. For example, the **`__le__`** method provides the less than or equal functionality.

Listing 1.7: Checking If Two Fractions are Equal

```
def __eq__(self, other):
    first_num = self.num * other.den
    second_num = other.num * self.den

    return first_num == second_num
```

The complete Fraction class, up to this point, is shown below. We leave the remaining arithmetic and relational methods as exercises.

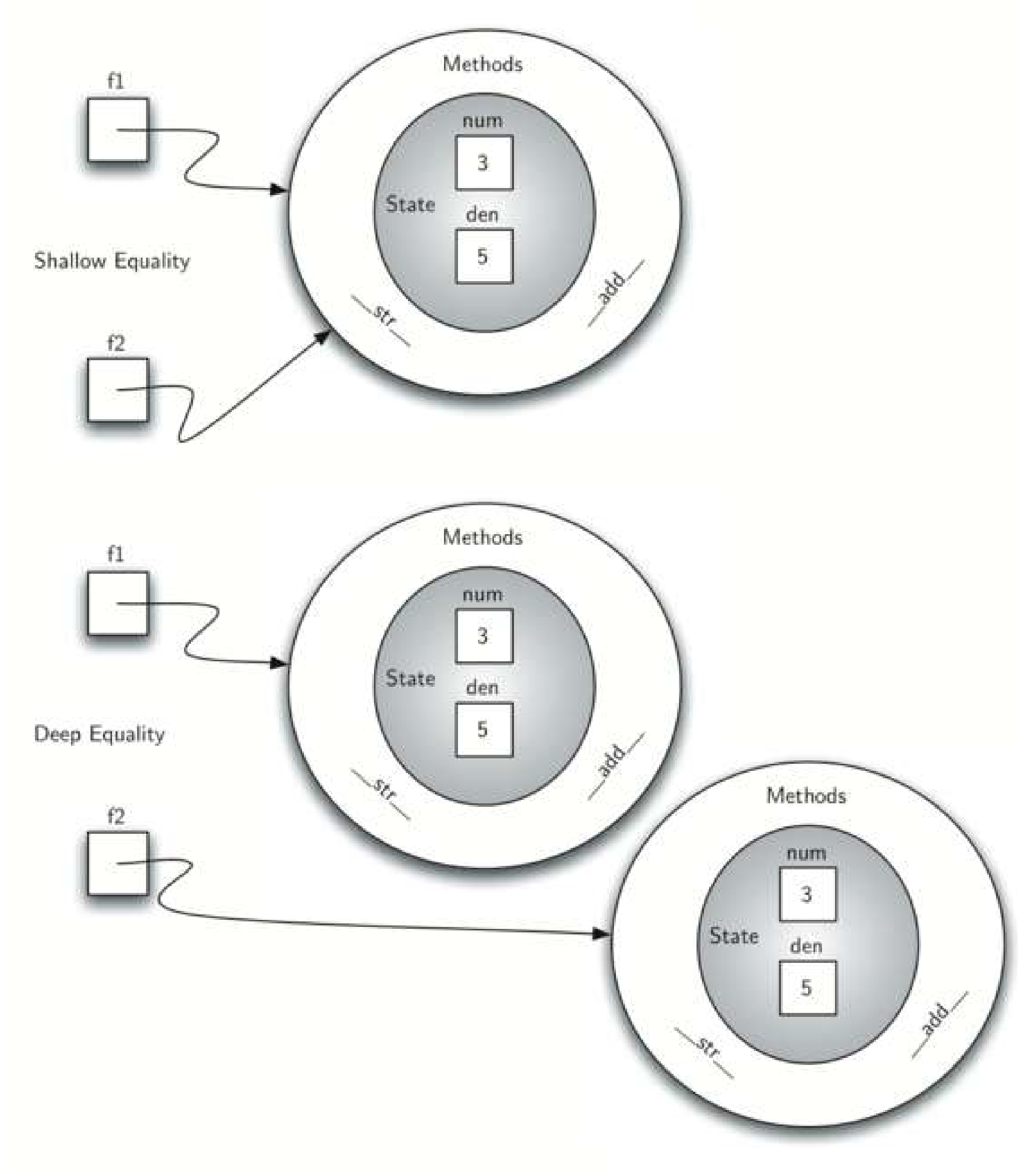


Figure 1.7: Shallow Equality Versus Deep Equality

```
# gcd function
def gcd(m, n):
   while m % n != 0:
      old_m = m
      old_n = n

      m = old_n
      n = old_m % old_n
   return n

# Fraction class
# Implements: addition and equality
# To do: multiplication, division, subtraction and comparison
   operators (< , >)
class Fraction:
    def __init__(self, top, bottom):
       self.num = top
       self.den = bottom

    def __str__(self):
       return str(self.num) + "/" + str(self.den)

    def show(self):
       print(self.num, "/", self.den)

    def __add__(self, other_fraction):
       new_num = self.num * other_fraction.den + \
                 self.den * other_fraction.num
       new_den = self.den * other_fraction.den
       common = gcd(new_num, new_den)
       return Fraction(new_num // common, new_den // common)

    def __eq__(self, other):
       first_num = self.num * other.den
       second_num = other.num * self.den

       return first_num == second_num
x = Fraction(1, 2)
y = Fraction(2, 3)
print(x + y)
print(x == y)
```

Self Check

To make sure you understand how operators are implemented in Python classes, and how to properly write methods, write some methods to implement $*$, $/$, and $-$. Also implement comparison operators $>$ and $<$.

1.5 Summary

- Computer science is the study of problem solving.
- Computer science uses abstraction as a tool for representing both processes and data.
- Abstract data types allow programmers to manage the complexity of a problem domain by hiding the details of the data.
- Python is a powerful, yet easy-to-use, object-oriented language.
- Lists, tuples, and strings are built in Python sequential collections.
- Dictionaries and sets are nonsequential collections of data.
- Classes allow programmers to implement abstract data types.
- Programmers can override standard methods as well as create new methods.
- A class constructor should always invoke the constructor of its parent before continuing on with its own data and behavior.

1.6 Key Terms

abstract data type	abstraction	algorithm
class	computable	data abstraction
data structure	data type	deep equality
dictionary	encapsulation	exception
format operator	formatted strings	implementation-independent
information hiding	list	list comprehension
method	mutability	object
procedural abstraction	programming	prompt
self	shallow equality	simulation
string	truth table	

1.7 Programming Exercises

1. Implement the simple methods `get_num` and `get_den` that will return the numerator and denominator of a fraction.
2. In many ways it would be better if all fractions were maintained in lowest terms right from the start. Modify the constructor for the `Fraction` class so that `GCD` is used to reduce fractions immediately. Notice that this means the `__add__` function no longer needs to reduce. Make the necessary modifications.
3. Implement the remaining simple arithmetic operators (`__sub__`, `__mul__`, and `__truediv__`).
4. Implement the remaining relational operators (`__gt__`, `__ge__`, `__lt__`, `__le__`, and `__ne__`)

5. Modify the constructor for the fraction class so that it checks to make sure that the numerator and denominator are both integers. If either is not an integer the constructor should raise an exception.
6. In the definition of fractions we assumed that negative fractions have a negative numerator and a positive denominator. Using a negative denominator would cause some of the relational operators to give incorrect results. In general, this is an unnecessary constraint. Modify the constructor to allow the user to pass a negative denominator so that all of the operators continue to work properly.
7. Research the `__radd__` method. How does it differ from `__add__`? When is it used? Implement `__radd__`.
8. Repeat the last question but this time consider the `__iadd__` method.
9. Research the `__repr__` method. How does it differ from `__str__`? When is it used? Implement `__repr__`.
10. Design a class to represent a playing card. Now design a class to represent a deck of cards. Using these two classes, implement a favorite card game.
11. Find a Sudoku puzzle in the local newspaper. Write a program to solve the puzzle.

CHAPTER

TWO

ALGORITHM ANALYSIS

2.1 Objectives

- To understand why algorithm analysis is important.
- To be able to use “Big-O” to describe execution time.
- To understand the “Big-O” execution time of common operations on Python lists and dictionaries.
- To understand how the implementation of Python data impacts algorithm analysis.
- To understand how to benchmark simple Python programs.

2.2 What Is Algorithm Analysis?

It is very common for beginning computer science students to compare their programs with one another. You may also have noticed that it is common for computer programs to look very similar, especially the simple ones. An interesting question often arises. When two programs solve the same problem but look different, is one program better than the other?

In order to answer this question, we need to remember that there is an important difference between a program and the underlying algorithm that the program is representing. As we stated in Chapter 1, an algorithm is a generic, step-by-step list of instructions for solving a problem. It is a method for solving any instance of the problem such that given a particular input, the algorithm produces the desired result. A program, on the other hand, is an algorithm that has been encoded into some programming language. There may be many programs for the same algorithm, depending on the programmer and the programming language being used.

To explore this difference further, consider the function shown below. This function solves a familiar problem, computing the sum of the first n integers. The algorithm uses the idea of an accumulator variable that is initialized to 0. The solution then iterates through the n integers, adding each to the accumulator.

```
def sum_of_n(n):
   the_sum = 0
   for i in range(1,n+1):
        the_sum = the_sum + i

    return the_sum

print(sum_of_n(10))
```

Now look at the function `foo` below. At first glance it may look strange, but upon further inspection you can see that this function is essentially doing the same thing as the previous one. The reason this is not obvious is poor coding. We did not use good identifier names to assist with readability, and we used an extra assignment statement during the accumulation step that was not really necessary.

```
def foo(tom):
    fred = 0
    for bill in range(1, tom+1):
       barney = bill
       fred = fred + barney

    return fred

print(foo(10))
```

The question we raised earlier asked whether one function is better than another. The answer depends on your criteria. The function `sum_of_n` is certainly better than the function `foo` if you are concerned with readability. In fact, you have probably seen many examples of this in your introductory programming course since one of the goals there is to help you write programs that are easy to read and easy to understand. In this course, however, we are also interested in characterizing the algorithm itself. (We certainly hope that you will continue to strive to write readable, understandable code.)

Algorithm analysis is concerned with comparing algorithms based upon the amount of computing resources that each algorithm uses. We want to be able to consider two algorithms and say that one is better than the other because it is more efficient in its use of those resources or perhaps because it simply uses fewer. From this perspective, the two functions above seem very similar. They both use essentially the same algorithm to solve the summation problem.

At this point, it is important to think more about what we really mean by computing resources. There are two different ways to look at this. One way is to consider the amount of space or memory an algorithm requires to solve the problem. The amount of space required by a problem solution is typically dictated by the problem instance itself. Every so often, however, there are algorithms that have very specific space requirements, and in those cases we will be very careful to explain the variations.

As an alternative to space requirements, we can analyze and compare algorithms based on the amount of time they require to execute. This measure is sometimes referred to as the "execution time" or "running time" of the algorithm. One way we can measure the execution time for the function `sum_of_n` is to do a benchmark analysis. This means that we will track the actual time required for the program to compute its result. In Python, we can benchmark a function by noting the starting time and ending time with respect to the system we are using. In the `time` module there is a function called `time` that will return the current system clock time in seconds since some arbitrary starting point. By calling this function twice, at the beginning and at the end, and then computing the difference, we can get an exact number of seconds (fractions in most cases) for execution.

```
import time

def sum_of_n_2(n):
  start = time.time()

  the_sum = 0
  for i in range(1, n+1):
    the_sum = the_sum + i

  end = time.time()

  return the_sum,end-start
```

This code shows the original **sum_of_n** function with the timing calls embedded before and after the summation. The function returns a tuple consisting of the result and the amount of time (in seconds) required for the calculation. If we perform 5 invocations of the function, each computing the sum of the first $10,000$ integers, we get the following:

```
>>>for i in range(5):
     print("Sum is %d required %10.7f seconds" % sum_of_n_2(10000))
Sum is 50005000 required 0.0018950 seconds
Sum is 50005000 required 0.0018620 seconds
Sum is 50005000 required 0.0019171 seconds
Sum is 50005000 required 0.0019162 seconds
Sum is 50005000 required 0.0019360 seconds
>>>
```

Again, the time required for each run, although longer, is very consistent, averaging about 10 times more seconds. For **n** equal to $1,000,000$ we get:

```
>>>for i in range(5):
     print("Sum is %d required %10.7f seconds" %
        sum_of_n_2(1000000))
Sum is 500000500000 required 0.1948988 seconds
Sum is 500000500000 required 0.1850290 seconds
Sum is 500000500000 required 0.1809771 seconds
Sum is 500000500000 required 0.1729250 seconds
Sum is 500000500000 required 0.1646299 seconds
>>>
```

In this case, the average again turns out to be about 10 times the previous.

Now consider the following code, which shows a different means of solving the summation problem. This function, **sum_of_n_3**, takes advantage of a closed equation $\sum_{i=0}^{n} i = \frac{(n)(n+1)}{2}$

to compute the sum of the first **n** integers without iterating.

```
def sum_of_n_3(n):
   return (n * (n + 1)) / 2

print(sum_of_n_3(10))
```

we do the same benchmark measurement for **sum_of_n_3**, using five different values for **n** ($10,000$, $100,000$, $1,000,000$, $10,000,000$, and $100,000,000$), we get the following results:

```
Sum is 50005000 required 0.00000095 seconds
Sum is 5000050000 required 0.00000191 seconds
Sum is 500000500000 required 0.00000095 seconds
Sum is 50000005000000 required 0.00000095 seconds
Sum is 5000000050000000 required 0.00000119 seconds
```

There are two important things to notice about this output. First, the times recorded above are shorter than any of the previous examples. Second, they are very consistent no matter what the value of **n**. It appears that **sum_of_n_3** is hardly impacted by the number of integers being added.

But what does this benchmark really tell us? Intuitively, we can see that the iterative solutions seem to be doing more work since some program steps are being repeated. This is likely the reason it is taking longer. Also, the time required for the iterative solution seems to increase as we increase the value of **n**. However, there is a problem. If we ran the same function on a different computer or used a different programming language, we would likely get different results. It could take even longer to perform **sum_of_n_3** if the computer were older.

We need a better way to characterize these algorithms with respect to execution time. The benchmark technique computes the actual time to execute. It does not really provide us with a useful measurement, because it is dependent on a particular machine, program, time of day, compiler, and programming language. Instead, we would like to have a characterization that is independent of the program or computer being used. This measure would then be useful for judging the algorithm alone and could be used to compare algorithms across implementations.

2.2.1 Big-O Notation

When trying to characterize an algorithm's efficiency in terms of execution time,independent of any particular program or computer, it is important to quantify the number of operations or steps that the algorithm will require. If each of these steps is considered to be a basic unit of computation, then the execution time for an algorithm can be expressed as the number of steps required to solve the problem. Deciding on an appropriate basic unit of computation can be a complicated problem and will depend on how the algorithm is implemented.

A good basic unit of computation for comparing the summation algorithms shown earlier might be to count the number of assignment statements performed to compute the sum. In the function **sum_of_n**, the number of assignment statements is 1 (the_sum$=0$) plus the value of **n** (the number of times we perform the_sum$=$the_sum$+i$). We can denote this by a function, call it

T, where $T(n) = 1 + n$. The parameter n is often referred to as the "size of the problem," and we can read this as "$T(n)$ is the time it takes to solve a problem of size n, namely $1 + n$ steps."

In the summation functions given above, it makes sense to use the number of terms in the summation to denote the size of the problem. We can then say that the sum of the first $100,000$ integers is a bigger instance of the summation problem than the sum of the first $1,000$. Because of this, it might seem reasonable that the time required to solve the larger case would be greater than for the smaller case. Our goal then is to show how the algorithm's execution time changes with respect to the size of the problem.

Computer scientists prefer to take this analysis technique one step further. It turns out that the exact number of operations is not as important as determining the most dominant part of the $T(n)$ function. In other words, as the problem gets larger, some portion of the $T(n)$ function tends to overpower the rest. This dominant term is what, in the end, is used for comparison. The **order of magnitude** function describes the part of $T(n)$ that increases the fastest as the value of n increases. Order of magnitude is often called **Big-O** notation (for "order") and written as $O(f(n))$. It provides a useful approximation to the actual number of steps in the computation. The function $f(n)$ provides a simple representation of the dominant part of the original $T(n)$.

In the above example, $T(n) = 1 + n$. As n gets large, the constant 1 will become less and less significant to the final result. If we are looking for an approximation for $T(n)$, then we can drop the 1 and simply say that the running time is $O(n)$. It is important to note that the 1 is certainly significant for $T(n)$. However, as n gets large, our approximation will be just as accurate without it.

As another example, suppose that for some algorithm, the exact number of steps is $T(n) = 5n^2 + 27n + 1005$. When n is small, say 1 or 2, the constant 1005 seems to be the dominant part of the function. However, as n gets larger, the n^2 term becomes the most important. In fact, when n is really large, the other two terms become insignificant in the role that they play in determining the final result. Again, to approximate $T(n)$ as n gets large, we can ignore the other terms and focus on $5n^2$. In addition, the coefficient 5 becomes insignificant as n gets large. We would say then that the function $T(n)$ has an order of magnitude $f(n) = n^2$, or simply that it is $O(n^2)$.

Although we do not see this in the summation example, sometimes the performance of an algorithm depends on the exact values of the data rather than simply the size of the problem. For these kinds of algorithms we need to characterize their performance in terms of best case, **worst case**, or **average case** performance. The worst case performance refers to a particular data set where the algorithm performs especially poorly. Whereas a different data set for the exact same algorithm might have extraordinarily good performance. However, in most cases the algorithm performs somewhere in between these two extremes (average case). It is important for a computer scientist to understand these distinctions so they are not misled by one particular case.

A number of very common order of magnitude functions will come up over and over as you study algorithms. These are shown in Table 2.1. In order to decide which of these functions is the dominant part of any $T(n)$ function, we must see how they compare with one another as n gets large.

Figure 2.1 shows graphs of the common functions from Table 2.1. Notice that when n is small,

f(n)	**Name**
1	Constant
$\log n$	Logarithmic
n	Linear
$n \log n$	Log Linear
n^2	Quadratic
n^3	Cubic
2^n	Exponential

Table 2.1: Common Functions for Big-O

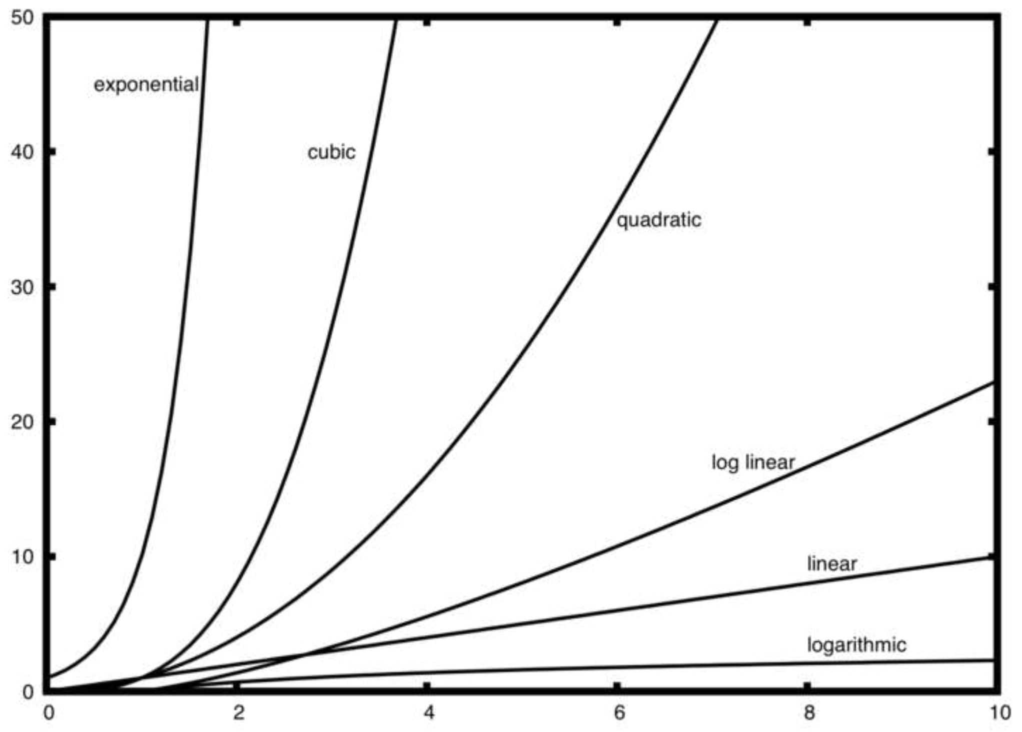


Figure 2.1: Plot of Common Big-O Functions

the functions are not very well defined with respect to one another. It is hard to tell which is dominant. However, as n grows, there is a definite relationship and it is easy to see how they compare with one another.

As a final example, suppose that we have the fragment of Python code shown below. Although this program does not really do anything, it is instructive to see how we can take actual code and analyze performance.

```
a = 5
b = 6
c = 10
for i in range(n):
   for j in range(n):
      x = i * i
      y = j * j
```

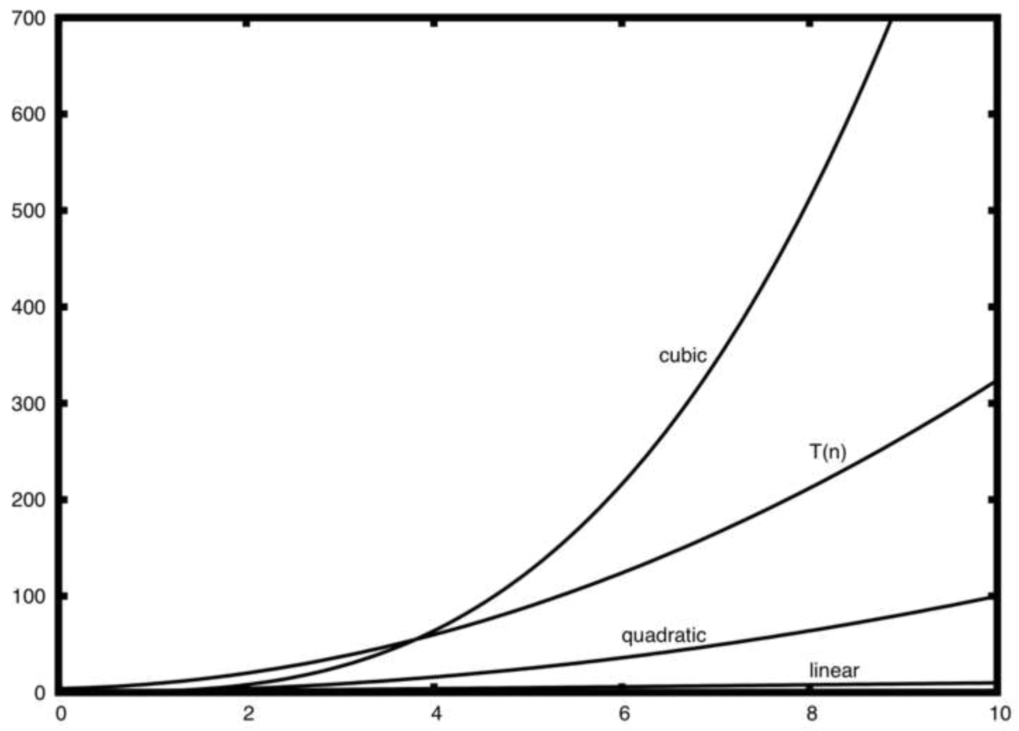


Figure 2.2: Comparing $T(n)$ with Common Big-O Functions

```
        z = i * j
for k in range(n):
    w = a * k + 45
    v = b * b
d = 33
```

The number of assignment operations is the sum of four terms. The first term is the constant 3, representing the three assignment statements at the start of the fragment. The second term is $3n^2$, since there are three statements that are performed n^2 times due to the nested iteration. The third term is $2n$, two statements iterated n times. Finally, the fourth term is the constant 1, representing the final assignment statement. This gives us $T(n) = 3 + 3n^2 + 2n + 1 = 3n^2 + 2n + 4$. By looking at the exponents, we can easily see that the n^2 term will be dominant and therefore this fragment of code is $O(n^2)$. Note that all of the other terms as well as the coefficient on the dominant term can be ignored as n grows larger.

Figure 2.2 shows a few of the common Big-O functions as they compare with the $T(n)$ function discussed above. Note that $T(n)$ is initially larger than the cubic function. However, as n grows, the cubic function quickly overtakes $T(n)$. It is easy to see that $T(n)$ then follows the quadratic function as n continues to grow.

Self Check

Write two Python functions to find the minimum number in a list. The first function should compare each number to every other number on the list. $O(n^2)$. The second function should be linear $O(n)$.

2.2.2 An Anagram Detection Example

A good example problem for showing algorithms with different orders of magnitude is the classic anagram detection problem for strings. One string is an anagram of another if the second is simply a rearrangement of the first. For example, `'heart'` and `'earth'` are anagrams. The strings `'python'` and `'typhon'` are anagrams as well. For the sake of simplicity, we will assume that the two strings in question are of equal length and that they are made up of symbols from the set of 26 lowercase alphabetic characters. Our goal is to write a boolean function that will take two strings and return whether they are anagrams.

Solution 1: Checking Off

Our first solution to the anagram problem will check to see that each character in the first string actually occurs in the second. If it is possible to "checkoff" each character, then the two strings must be anagrams. Checking off a character will be accomplished by replacing it with the special Python value `None`. However, since strings in Python are immutable, the first step in the process will be to convert the second string to a list. Each character from the first string can be checked against the characters in the list and if found, checked off by replacement.

```python
def anagram_solution1(s1,s2):
   a_list = list(s2)

   pos1 = 0
   still_ok = True

   while pos1 < len(s1) and still_ok:
      pos2 = 0
      found = False
      while pos2 < len(a_list) and not found:
         if s1[pos1] == a_list[pos2]:
            found = True
         else:
            pos2 = pos2 + 1

      if found:
         a_list[pos2] = None
      else:
         still_ok = False

      pos1 = pos1 + 1

   return still_ok

print(anagram_solution1('abcd','dcba'))
```

To analyze this algorithm, we need to note that each of the n characters in `s1` will cause an iteration through up to n characters in the list from `s2`. Each of the n positions in the list will be visited once to match a character from `s1`. The number of visits then becomes the sum of the integers from 1 to n. We stated earlier that this can be written as

$$\sum_{i=1}^{n} i = \frac{n(n+1}{2}) = \frac{1}{2}n^2 + \frac{1}{2}n$$

As n gets large, the n^2 term will dominate the n term and the $\frac{1}{2}$ can be ignored. Therefore, this solution is $O(n^2)$.

Solution 2: Sort and Compare

Another solution to the anagram problem will make use of the fact that even though `s1` and `s2` are different, they are anagrams only if they consist of exactly the same characters. So, if we begin by sorting each string alphabetically, from a to z, we will end up with the same string if the original two strings are anagrams. The code below shows this solution. Again, in Python we can use the built-in `sort` method on lists by simply converting each string to a list at the start.

```
def anagram_solution2(s1,s2):
    a_list1 = list(s1)
    a_list2 = list(s2)

    a_list1.sort()
    a_list2.sort()

    pos = 0
    matches = True

    while pos < len(s1) and matches:
        if a_list1[pos] == a_list2[pos]:
            pos = pos + 1
        else:
            matches = False

    return matches

print(anagram_solution2('abcde','edcba'))
```

At first glance you may be tempted to think that this algorithm is $O(n)$, since there is one simple iteration to compare the n characters after the sorting process. However, the two calls to the Python `sort` method are not without their own cost. As we will see in a later chapter, sorting is typically either $O(n^2)$ or $O(n \log n)$, so the sorting operations dominate the iteration. In the end, this algorithm will have the same order of magnitude as that of the sorting process.

2.2.3 Solution 3: Brute Force

A **brute force** technique for solving a problem typically tries to exhaust all possibilities. For the anagram detection problem, we can simply generate a list of all possible strings using the characters from `s1` and then see if `s2` occurs. However, there is a difficulty with this approach. When generating all possible strings from `s1`, there are n possible first characters, $n-1$ possible characters for the second position, $n-2$ for the third, and so on. The total number of candidate strings is $n*(n-1)*(n-2)*\cdots*3*2*1$, which is $n!$. Although some of the strings may be duplicates, the program cannot know this ahead of time and so it will still generate $n!$ different strings.

It turns out that $n!$ grows even faster than 2^n as n gets large. In fact, if s1 were 20 characters long, there would be $20! = 2,432,902,008,176,640,000$ possible candidate strings. If we processed one possibility every second, it would still take us $77,146,816,596$ years to go through the entire list. This is probably not going to be a good solution.

2.2.4 Solution 4: Count and Compare

Our final solution to the anagram problem takes advantage of the fact that any two anagrams will have the same number of a's, the same number of b's, the same number of c's, and so on. In order to decide whether two strings are anagrams, we will first count the number of times each character occurs. Since there are 26 possible characters, we can use a list of 26 counters, one for each possible character. Each time we see a particular character, we will increment the counter at that position. In the end, if the two lists of counters are identical, the strings must be anagrams.

```python
def anagram_solution4(s1, s2):
   c1 = [0] * 26
   c2 = [0] * 26

   for i in range(len(s1)):
      pos = ord(s1[i]) - ord('a')
      c1[pos] = c1[pos] + 1

   for i in range(len(s2)):
      pos = ord(s2[i]) - ord('a')
      c2[pos] = c2[pos] + 1

   j = 0
   still_ok = True
   while j < 26 and still_ok:
      if c1[j] == c2[j]:
         j = j + 1
      else:
         still_ok = False

   return still_ok

print(anagram_solution4('apple','pleap'))
```

Again, the solution has a number of iterations. However, unlike the first solution, none of them are nested. The first two iterations used to count the characters are both based on n. The third iteration, comparing the two lists of counts, always takes 26 steps since there are 26 possible characters in the strings. Adding it all up gives us $T(n) = 2n + 26$ steps. That is $O(n)$. We have found a linear order of magnitude algorithm for solving this problem.

Before leaving this example, we need to say something about space requirements. Although the last solution was able to run in linear time, it could only do so by using additional storage to keep the two lists of character counts. In other words, this algorithm sacrificed space in order to gain time.

This is a common occurrence. On many occasions you will need to make decisions between time and space trade-offs. In this case, the amount of extra space is not significant. However, if the underlying alphabet had millions of characters, there would be more concern. As a computer scientist, when given a choice of algorithms, it will be up to you to determine the best use of computing resources given a particular problem.

Self Check

Q-1: Given the following code fragment, what is its Big-O running time?

```
test = 0
for i in range(n):
   for j in range(n):
      test = test + i * j
```

1. $O(n)$
2. $O(n^2)$
3. $O(\log n)$
4. $O(n^3)$

Q-2: Given the following code fragment what is its Big-O running time?

```
test = 0
for i in range(n):
   test = test + 1

for j in range(n):
   test = test - 1
```

1. $O(n)$
2. $O(n^2)$
3. $O(\log n)$
4. $O(n^3)$

Q-3: Given the following code fragment what is its Big-O running time?

```
i = n
while i > 0:
   k = 2 + 2
   i = i // 2
```

1. $O(n)$
2. $O(n^2)$
3. $O(\log n)$
4. $O(n^3)$

2.3 Performance of Python Data Structures

Now that you have a general idea of Big-O notation and the differences between the different functions, our goal in this section is to tell you about the Big-O performance for the operations on Python lists and dictionaries. We will then show you some timing experiments that illustrate the costs and benefits of using certain operations on each data structure. It is important for you to understand the efficiency of these Python data structures because they are the building blocks we will use as we implement other data structures in the remainder of the book. In this section we are not going to explain why the performance is what it is. In later chapters you will see some possible implementations of both lists and dictionaries and how the performance depends on the implementation.

2.3.1 Lists

The designers of Python had many choices to make when they implemented the list data structure. Each of these choices could have an impact on how fast list operations perform. To help them make the right choices they looked at the ways that people would most commonly use the list data structure and they optimized their implementation of a list so that the most common operations were very fast. Of course they also tried to make the less common operations fast, but when a tradeoff had to be made the performance of a less common operation was often sacrificed in favor of the more common operation.

Two common operations are indexing and assigning to an index position. Both of these operations take the same amount of time no matter how large the list becomes. When an operation like this is independent of the size of the list they are $O(1)$.

Another very common programming task is to grow a list. There are two ways to create a longer list. You can use the append method or the concatenation operator. The append method is $O(1)$. However, the concatenation operator is $O(k)$ where k is the size of the list that is being concatenated. This is important for you to know because it can help you make your own programs more efficient by choosing the right tool for the job.

Let us look at four different ways we might generate a list of `n` numbers starting with 0. First we will try a `for` loop and create the list by concatenation, then we will use append rather than concatenation. Next, we will try creating the list using list comprehension and finally, and perhaps the most obvious way, using the range function wrapped by a call to the list constructor. The following code shows making our list in four different ways.

```
def test1():
    l = []
    for i in range(1000):
        l = l + [i]

def test2():
    l = []
    for i in range(1000):
        l.append(i)

def test3():
    l = [i for i in range(1000)]

def test4():
    l = list(range(1000))
```

To capture the time it takes for each of our functions to execute we will use Python's `timeit` module. The `timeit` module is designed to allow Python developers to make cross-platform timing measurements by running functions in a consistent environment and using timing mechanisms that are as similar as possible across operating systems.

To use `timeit` you create a `Timer` object whose parameters are two Python statements. The first parameter is a Python statement that you want to time; the second parameter is a statement that will run once to set up the test. The `timeit` module will then time how long it takes to execute the statement some number of times. By default `timeit` will try to run the statement one million times. When its done it returns the time as a floating point value representing the total number of seconds. However, since it executes the statement a million times you can read the result as the number of microseconds to execute the test one time. You can also pass `timeit` a named parameter called `number` that allows you to specify how many times the test statement is executed. The following session shows how long it takes to run each of our test functions 1000 times.

```
# Import the timeit module
import timeit
# Import the Timer class defined in the module
from timeit import Timer
# If the above line is excluded, you need to replace Timer with
   timeit.Timer when defining a Timer object
t1 = Timer("test1()", "from __main__ import test1")
print("concat ",t1.timeit(number=1000), "milliseconds")
t2 = Timer("test2()", "from __main__ import test2")
print("append ",t2.timeit(number=1000), "milliseconds")
t3 = Timer("test3()", "from __main__ import test3")
print("comprehension ",t3.timeit(number=1000), "milliseconds")
t4 = Timer("test4()", "from __main__ import test4")
print("list range ",t4.timeit(number=1000), "milliseconds")

concat 6.54352807999 milliseconds
append 0.306292057037 milliseconds
comprehension 0.147661924362 milliseconds
list range 0.0655000209808 milliseconds
```

In the experiment above the statement that we are timing is the function call to `test1()`, `test2()`, and so on. The setup statement may look very strange to you, so let us consider it in more detail. You are probably very familiar with the `from`, `import` statement, but this is usually used at the beginning of a Python program file. In this case the statement `from __main__ import test1` imports the function `test1` from the `__main__` namespace into the namespace that `timeit` sets up for the timing experiment. The `timeit` module does this because it wants to run the timing tests in an environment that is uncluttered by any stray variables you may have created, that may interfere with your function's performance in some unforeseen way.

From the experiment above it is clear that the append operation at 0.30 milliseconds is much faster than concatenation at 6.54 milliseconds. In the above experiment we also show the times for two additional methods for creating a list; using the list constructor with a call to `range` and a list comprehension. It is interesting to note that the list comprehension is twice as fast as a `for` loop with an `append` operation.

One final observation about this little experiment is that all of the times that you see above include some overhead for actually calling the test function, but we can assume that the function call overhead is identical in all four cases so we still get a meaningful comparison of the operations. So it would not be accurate to say that the concatenation operation takes 6.54 milliseconds but rather the concatenation test function takes 6.54 milliseconds. As an exercise you could test the time it takes to call an empty function and subtract that from the numbers above.

Now that we have seen how performance can be measured concretely you can look at Table 2.2 to see the Big-O efficiency of all the basic list operations. After thinking carefully about Table 2.2, you may be wondering about the two different times for `pop`. When `pop` is called on the end of the list it takes $O(1)$ but when pop is called on the first element in the list or anywhere in the middle it is $O(n)$. The reason for this lies in how Python chooses to implement lists. When an item is taken from the front of the list, in Python's implementation, all the other elements in the list are shifted one position closer to the beginning. This may seem silly to you now, but if you look at Table 2.2 you will see that this implementation also allows the index operation to be $O(1)$. This is a tradeoff that the Python implementors thought was a good one.

As a way of demonstrating this difference in performance let us do another experiment using the `timeit` module. Our goal is to be able to verify the performance of the `pop` operation on a list of a known size when the program pops from the end of the list, and again when the program pops from the beginning of the list. We will also want to measure this time for lists of different sizes. What we would expect to see is that the time required to pop from the end of the list will stay constant even as the list grows in size, while the time to pop from the beginning of the list will continue to increase as the list grows.

The code below shows one attempt to measure the difference between the two uses of pop. As you can see from this first example, popping from the end takes 0.0003 milliseconds, whereas popping from the beginning takes 4.82 milliseconds. For a list of two million elements this is a factor of $16,000$.

There are a couple of things to notice about this code. The first is the statement from `__main__ import $x$`. Although we did not define a function we do want to be able to use the list object x in our test. This approach allows us to time just the single `pop` statement and get the most accurate measure of the time for that single operation. Because the timer repeats

Operation	Big-O Efficiency
indexx[]	$O(1)$
index assignment	$O(1)$
append	$O(1)$
pop()	$O(1)$
pop(i)	$O(n)$
insert(i,item)	$O(n)$
del operator	$O(n)$
iteration	$O(n)$
contains (in)	$O(n)$
get slice [x:y]	$O(k)$
del slice	$O(n)$
set slice	$O(n+k)$
reverse	$O(n)$
concatenate	$O(k)$
sort	$O(n \log n)$
multiply	$O(nk)$

Table 2.2: Big-O Efficiency of Python List Operators

1000 times it is also important to point out that the list is decreasing in size by 1 each time through the loop. But since the initial list is two million elements in size we only reduce the overall size by 0.05%.

```
pop_zero = Timer("x.pop(0)",
                 "from __main__ import x")
pop_end = Timer("x.pop()",
                "from __main__ import x")

x = list(range(2000000))
pop_zero.timeit(number=1000)
4.8213560581207275

x = list(range(2000000))
pop_end.timeit(number=1000)
0.0003161430358886719
```

While our first test does show that **pop(0)** is indeed slower than **pop()**, it does not validate the claim that **pop(0)** is $O(n)$ while **pop()** is $O(1)$. To validate that claim we need to look at the performance of both calls over a range of list sizes.

```
pop_zero = Timer("x.pop(0)", "from __main__ import x")
pop_end = Timer("x.pop()", "from __main__ import x")
print("pop(0)  pop()")
for i in range(1000000,100000001,1000000):
   x = list(range(i))
   pt = pop_end.timeit(number=1000)
   x = list(range(i))
```

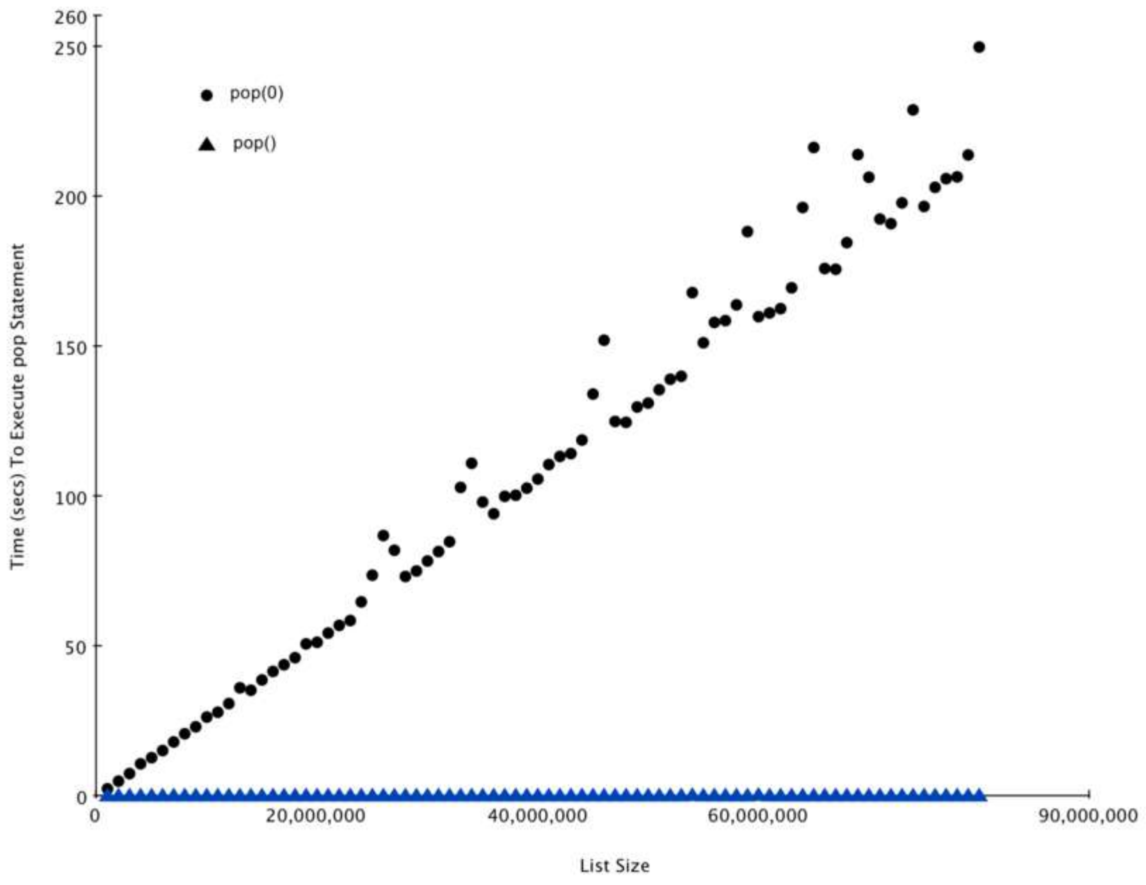


Figure 2.3: Comparing the Performance of pop and pop(0)

```
pz = pop_zero.timeit(number=1000)
print("%15.5f, %15.5f" %(pz,pt))
```

Figure 2.3 shows the results of our experiment. You can see that as the list gets longer and longer the time it takes to `pop(0)` also increases while the time for `pop` stays very flat. This is exactly what we would expect to see for a $O(n)$ and $O(1)$ algorithm.

Some sources of error in our little experiment include the fact that there are other processes running on the computer as we measure that may slow down our code, so even though we try to minimize other things happening on the computer there is bound to be some variation in time. That is why the loop runs the test one thousand times in the first place to statistically gather enough information to make the measurement reliable.

2.3.2 Dictionaries

The second major Python data structure is the dictionary. As you probably recall, dictionaries differ from lists in that you can access items in a dictionary by a key rather than a position. Later in this book you will see that there are many ways to implement a dictionary. The thing that is most important to notice right now is that the get item and set item operations on a dictionary are $O(1)$. Another important dictionary operation is the contains operation. Checking to see whether a key is in the dictionary or not is also $O(1)$. The efficiency of all dictionary operations

Operation	Big-O Efficiency
copy	$O(n)$
get item	$O(1)$
set item	$O(1)$
delete item	$O(1)$
contains (in)	$O(1)$
iteration	$O(n)$

Table 2.3: Big-O Efficiency of Python Dictionary Operations

is summarized in Table 2.3. One important side note on dictionary performance is that the efficiencies we provide in the table are for average performance. In some rare cases the contains, get item, and set item operations can degenerate into $O(n)$ performance but we will get into that in a later chapter when we talk about the different ways that a dictionary could be implemented.

For our last performance experiment we will compare the performance of the contains operation between lists and dictionaries. In the process we will confirm that the contains operator for lists is $O(n)$ and the contains operator for dictionaries is $O(1)$. The experiment we will use to compare the two is simple. We'll make a list with a range of numbers in it. Then we will pick numbers at random and check to see if the numbers are in the list. If our performance tables are correct the bigger the list the longer it should take to determine if any one number is contained in the list.

We will repeat the same experiment for a dictionary that contains numbers as the keys. In this experiment we should see that determining whether or not a number is in the dictionary is not only much faster, but the time it takes to check should remain constant even as the dictionary grows larger.

The code below implements this comparison. Notice that we are performing exactly the same operation, `number in container`. The difference is that on line 7 `x` is a list, and on line 9 `x` is a dictionary.

```
1  import timeit
2  import random
3
4  for i in range(10000,1000001,20000):
5     t = timeit.Timer("random.randrange(%d) in x"%i,
6                      "from __main__ import random,x")
7     x = list(range(i))
8     lst_time = t.timeit(number=1000)
9     x = {j:None for j in range(i)}
10    d_time = t.timeit(number=1000)
11    print("%d,%10.3f,%10.3f" % (i, lst_time, d_time))
```

Figure 2.4 summarizes the results. You can see that the dictionary is consistently faster. For the smallest list size of $10,000$ elements a dictionary is 89.4 times faster than a list. For the largest list size of $990,000$ elements the dictionary is $11,603$ times faster! You can also see that the time it takes for the contains operator on the list grows linearly with the size of the list. This verifies the assertion that the contains operator on a list is $O(n)$. It can also be seen that the

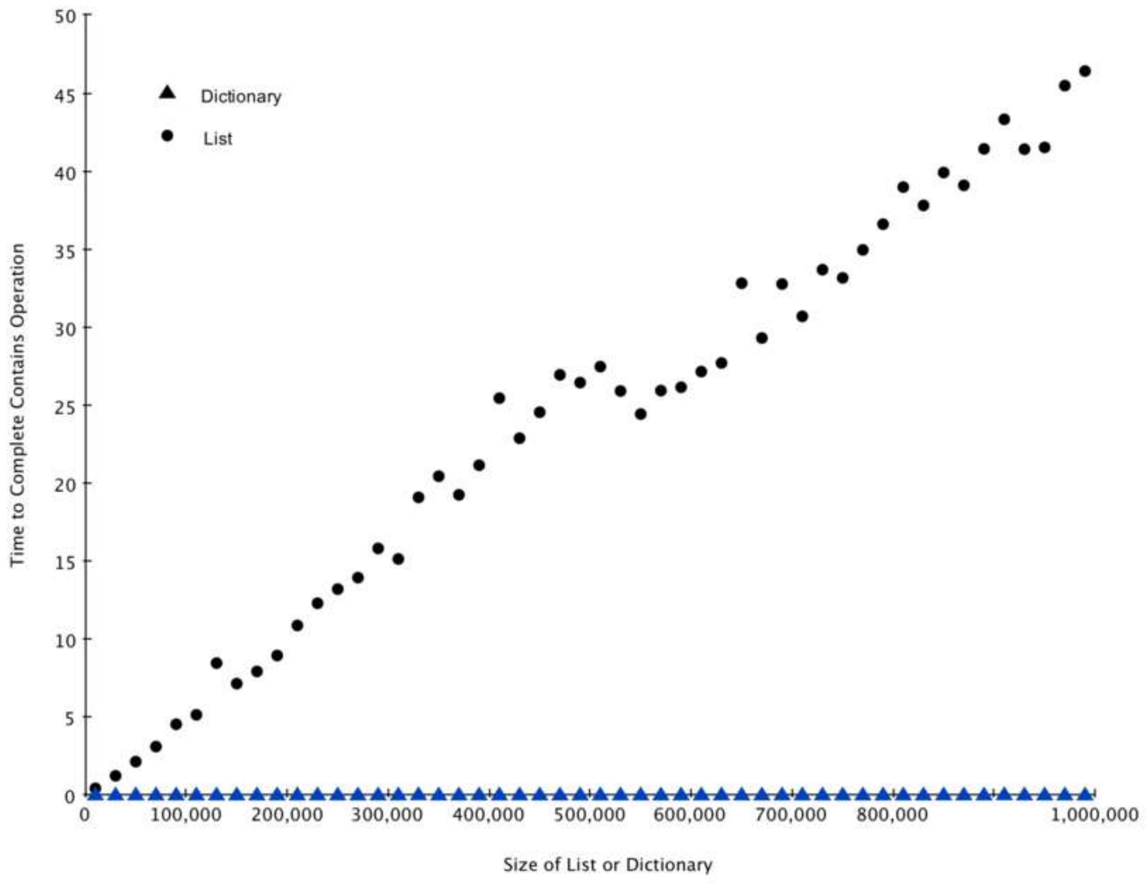


Figure 2.4: Comparing the in Operator for Python Lists and Dictionaries

time for the contains operator on a dictionary is constant even as the dictionary size grows. In fact for a dictionary size of $10,000$ the contains operation took 0.004 milliseconds and for the dictionary size of $990,000$ it also took 0.004 milliseconds.

Since Python is an evolving language, there are always changes going on behind the scenes. The latest information on the performance of Python data structures can be found on the Python website. As of this writing the Python wiki has a nice time complexity page that can be found at the Time Complexity Wiki.

Self Check

Q-4: Which of the above list operations is not $O(1)$?

1. list.pop(0)
2. list.pop()
3. list.append()
4. list[10]
5. all of the above are $O(1)$

Q-5: Which of the above dictionary operations is $O(1)$?

1. ‘x’ in my_dict
2. del my_dict[‘x’]
3. my_dict[‘x’] == 10
4. my_dict[‘x’] = my_dict[‘x’] + 1
5. all of the above are $O(1)$

2.4 Summary

- Algorithm analysis is an implementation-independent way of measuring an algorithm.
- Big-O notation allows algorithms to be classified by their dominant process with respect to the size of the problem.

2.5 Key Terms

average case	Big-O notation	brute force
checking off	exponential	linear
log linear	logarithmic	order of magnitude
quadratic	time complexity	worst case

2.6 Discussion Questions

1. Give the Big-O performance of the following code fragment:

```
for i in range(n):
   for j in range(n):
      k = 2 + 2
```

2. Give the Big-O performance of the following code fragment:

```
for i in range(n):
     k = 2 + 2
```

3. Give the Big-O performance of the following code fragment:

```
i = n
while i > 0:
   k = 2 + 2
   i = i // 2
```

4. Give the Big-O performance of the following code fragment:

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            k = 2 + 2
```

5. Give the Big-O performance of the following code fragment:

```
i = n
while i > 0:
    k = 2 + 2
    i = i // 2
```

6. Give the Big-O performance of the following code fragment:

```
for i in range(n):
    k = 2 + 2
for j in range(n):
    k = 2 + 2
for k in range(n):
    k = 2 + 2
```

2.7 Programming Exercises

1. Devise an experiment to verify that the list index operator is $O(1)$.
2. Devise an experiment to verify that get item and set item are $O(1)$ for dictionaries.
3. Devise an experiment that compares the performance of the `del` operator on lists and dictionaries.
4. Given a list of numbers in random order write a linear time algorithm to find the kth smallest number in the list. Explain why your algorithm is linear.
5. Can you improve the algorithm from the previous problem to be $O(n \log(n))$?

CHAPTER

THREE

BASIC DATA STRUCTURES

3.1 Objectives

- To understand the abstract data types stack, queue, deque, and list.
- To be able to implement the ADTs stack, queue, and deque using Python lists.
- To understand the performance of the implementations of basic linear data structures.
- To understand prefix, infix, and postfix expression formats.
- To use stacks to evaluate postfix expressions.
- To use stacks to convert expressions from infix to postfix.
- To use queues for basic timing simulations.
- To be able to recognize problem properties where stacks, queues, and deques are appropriate data structures.
- To be able to implement the abstract data type list as a linked list using the node and reference pattern.
- To be able to compare the performance of our linked list implementation with Python's list implementation.

3.2 What Are Linear Structures?

We will begin our study of data structures by considering four simple but very powerful concepts. Stacks, queues, deques, and lists are examples of data collections whose items are ordered depending on how they are added or removed. Once an item is added, it stays in that position relative to the other elements that came before and came after it. Collections such as these are often referred to as linear data structures.

Linear structures can be thought of as having two ends. Sometimes these ends are referred to as the "left" and the "right" or in some cases the "front" and the "rear." You could also call them the "top" and the "bottom." The names given to the ends are not significant. What distinguishes one linear structure from another is the way in which items are added and removed, in particular the location where these additions and removals occur. For example, a structure might allow

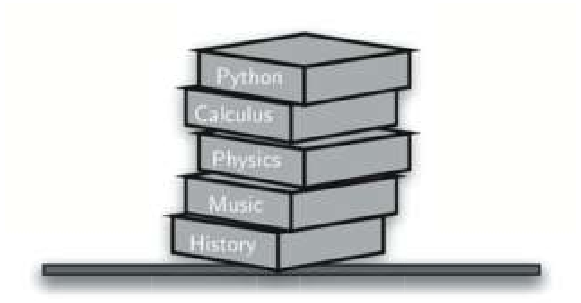


Figure 3.1: A Stack of Books

new items to be added at only one end. Some structures might allow items to be removed from either end.

These variations give rise to some of the most useful data structures in computer science. They appear in many algorithms and can be used to solve a variety of important problems.

3.3 Stacks

3.3.1 What is a Stack?

A **stack** (sometimes called a "push-down stack") is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end. This end is commonly referred to as the "top." The end opposite the top is known as the "base."

The base of the stack is significant since items stored in the stack that are closer to the base represent those that have been in the stack the longest. The most recently added item is the one that is in position to be removed first. This ordering principle is sometimes called **LIFO, last-in first-out**. It provides an ordering based on length of time in the collection. Newer items are near the top, while older items are near the base.

Many examples of stacks occur in everyday situations. Almost any cafeteria has a stack of trays or plates where you take the one at the top, uncovering a new tray or plate for the next customer in line. Imagine a stack of books on a desk (Figure 3.1). The only book whose cover is visible is the one on top. To access others in the stack, we need to remove the ones that are sitting on top of them. Figure 3.2 shows another stack. This one contains a number of primitive Python data objects.

One of the most useful ideas related to stacks comes from the simple observation of items as they are added and then removed. Assume you start out with a clean desktop. Now place books one at a time on top of each other. You are constructing a stack. Consider what happens when you begin removing books. The order that they are removed is exactly the reverse of the order that they were placed. Stacks are fundamentally important, as they can be used to reverse the order of items. The order of insertion is the reverse of the order of removal. Figure 3.3 shows the Python data object stack as it was created and then again as items are removed. Note the order of the objects.

Considering this reversal property, you can perhaps think of examples of stacks that occur as

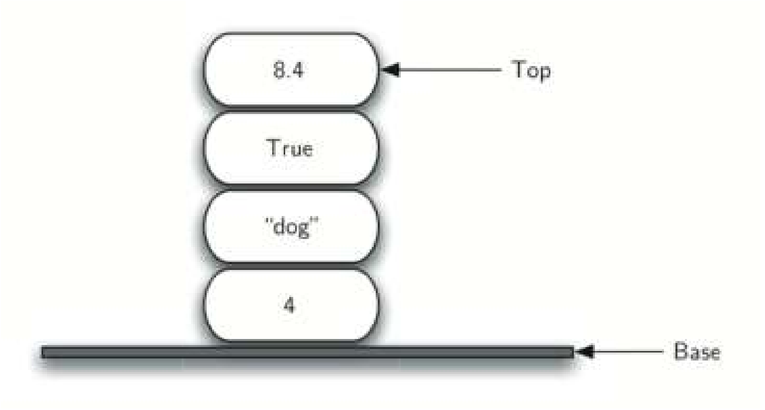


Figure 3.2: A Stack of Primitive Python Objects

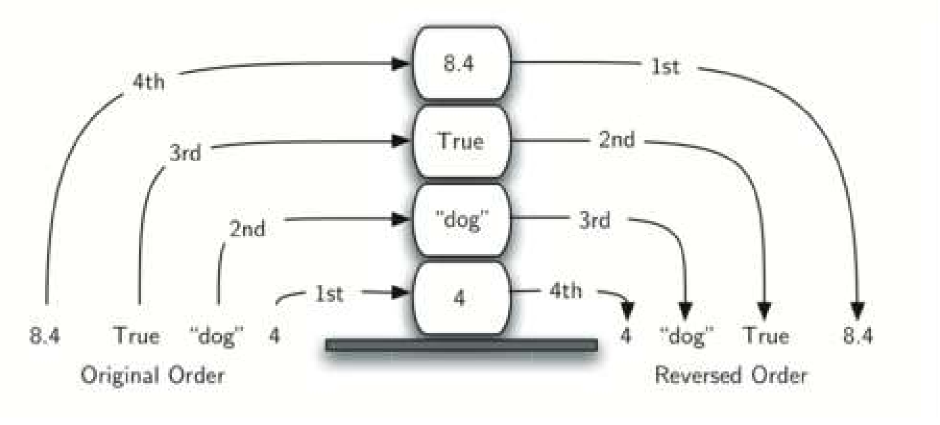


Figure 3.3: The Reversal Property of Stacks

Stack Operation	Stack Contents	Return Value
`s.is_empty()`	`[]`	`True`
`s.push(4)`	`[4]`	
`s.push('dog')`	`[4,'dog']`	
`s.peek()`	`[4,'dog']`	`'dog'`
`s.push(True)`	`[4,'dog',True]`	
`s.size()`	`[4,'dog',True]`	`3`
`s.is_empty()`	`[4,'dog',True]`	`False`
`s.push(8.4)`	`[4,'dog',True,8.4]`	
`s.pop()`	`[4,'dog',True]`	`8.4`
`s.pop()`	`[4,'dog']`	`True`
`s.size()`	`[4,'dog']`	`2`

Table 3.1: Sample Stack Operations

you use your computer. For example, every web browser has a Back button. As you navigate from web page to web page, those pages are placed on a stack (actually it is the URLs that are going on the stack). The current page that you are viewing is on the top and the first page you looked at is at the base. If you click on the Back button, you begin to move in reverse order through the pages.

3.4 The Stack Abstract Data Type

The stack abstract data type is defined by the following structure and operations. A stack is structured, as described above, as an ordered collection of items where items are added to and removed from the end called the "top." Stacks are ordered LIFO. The stack operations are given below.

- **`Stack()`** creates a new stack that is empty. It needs no parameters and returns an empty stack.
- **`push(item)`** adds a new item to the top of the stack. It needs the item and returns nothing.
- **`pop()`** removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.
- **`peek()`** returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.
- **`is_empty()`** tests to see whether the stack is empty. It needs no parameters and returns a boolean value.
- **`size()`** returns the number of items on the stack. It needs no parameters and returns an integer.

For example, if s is a stack that has been created and starts out empty, then Table 3.1 shows the results of a sequence of stack operations. Under stack contents, the top item is listed at the far right.

3.4.1 Implementing A Stack in Python

Now that we have clearly defined the stack as an abstract data type we will turn our attention to using Python to implement the stack. Recall that when we give an abstract data type a physical implementation we refer to the implementation as a data structure.

As we described in Chapter 1, in Python, as in any object-oriented programming language, the implementation of choice for an abstract data type such as a stack is the creation of a new class. The stack operations are implemented as methods. Further, to implement a stack, which is a collection of elements, it makes sense to utilize the power and simplicity of the primitive collections provided by Python. We will use a list.

Recall that the list class in Python provides an ordered collection mechanism and a set of methods. For example, if we have the list $[2, 5, 3, 6, 7, 4]$, we need only to decide which end of the list will be considered the top of the stack and which will be the base. Once that decision is made, the operations can be implemented using the list methods such as append and pop.

The following stack implementation assumes that the end of the list will hold the top element of the stack. As the stack grows (as push operations occur), new items will be added on the end of the list. pop operations will manipulate that same end.

```python
# Completed implementation of a stack ADT
class Stack:
    def __init__(self):
       self.items = []

    def is_empty(self):
       return self.items == []

    def push(self, item):
       self.items.append(item)

    def pop(self):
       return self.items.pop()

    def peek(self):
       return self.items[len(self.items)-1]

    def size(self):
       return len(self.items)
```

Remember that nothing happens when we click the run button other than the definition of the class. We must create a Stack object and then use it. shows the Stack class in action as we perform the sequence of operations from Table 3.1.

```python
s = Stack()

print(s.is_empty())
s.push(4)
s.push('dog')
```

```
print(s.peek())
s.push(True)
print(s.size())
print(s.is_empty())
s.push(8.4)
print(s.pop())
print(s.pop())
print(s.size())
```

It is important to note that we could have chosen to implement the stack using a list where the top is at the beginning instead of at the end. In this case, the previous pop and append methods would no longer work and we would have to index position 0 (the first item in the list) explicitly using pop and insert. The implementation is shown below.

```
class Stack:
  def __init__(self):
    self.items = []

  def is_empty(self):
    return self.items == []

  def push(self, item):
    self.items.insert(0, item)

  def pop(self):
    return self.items.pop(0)

  def peek(self):
    return self.items[0]

  def size(self):
    return len(self.items)

s = Stack()
s.push('hello')
s.push('true')
print(s.pop())
```

This ability to change the physical implementation of an abstract data type while maintaining the logical characteristics is an example of abstraction at work. However, even though the stack will work either way, if we consider the performance of the two implementations, there is definitely a difference. Recall that the append and pop() operations were both $O(1)$. This means that the first implementation will perform push and pop in constant time no matter how many items are on the stack. The performance of the second implementation suffers in that the insert(0) and pop(0) operations will both require $O(n)$ for a stack of size n. Clearly, even though the implementations are logically equivalent, they would have very different timings when performing benchmark testing.

3.4.2 Self Check

Given the following sequence of stack operations, what is the top item on the stack when the sequence is complete?

```
m = Stack()
m.push('x')
m.push('y')
m.pop()
m.push('z')
m.peek()
```

1. 'x'
2. 'y'
3. 'z'
4. The stack is empty

Given the following sequence of stack operations, what is the top item on the stack when the sequence is complete?

```
m = Stack()
m.push('x')
m.push('y')
m.push('z')
while not m.is_empty():
   m.pop()
   m.pop()
```

1. 'x'
2. the stack is empty
3. an error will occur
4. 'z'

Write a function rev_string(my_str) that uses a stack to reverse the characters in a string.

3.4.3 Simple Balance Parentheses

We now turn our attention to using stacks to solve real computer science problems. You have no doubt written arithmetic expressions such as

$(5 + 6) * (7 + 8)/(4 + 3)$

where parentheses are used to order the performance of operations. You may also have some experience programming in a language such as Lisp with constructs like

```
(defun square(n)
    (* n n))
```

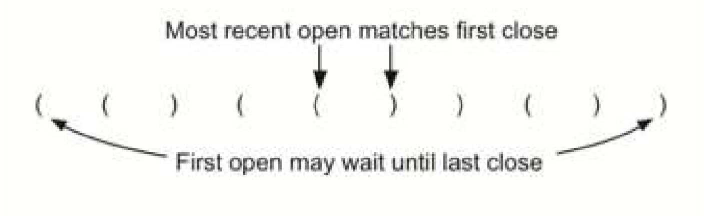

Most recent open matches first close
(( ) (( )) ( ))
First open may wait until last close

Figure 3.4: Matching Parentheses

This defines a function called square that will return the square of its argument n. Lisp is notorious for using lots and lots of parentheses.

In both of these examples, parentheses must appear in a balanced fashion. **Balanced parentheses** means that each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested. Consider the following correctly balanced strings of parentheses:

```
(()()()())

(((())))

(()((())()))
```

Compare those with the following, which are not balanced:

```
((((((())

()))

(()()(()
```

The ability to differentiate between parentheses that are correctly balanced and those that are unbalanced is an important part of recognizing many programming language structures.

The challenge then is to write an algorithm that will read a string of parentheses from left to right and decide whether the symbols are balanced. To solve this problem we need to make an important observation. As you process symbols from left to right, the most recent opening parenthesis must match the next closing symbol (see Figure 3.4). Also, the first opening symbol processed may have to wait until the very last symbol for its match. Closing symbols match opening symbols in the reverse order of their appearance; they match from the inside out. This is a clue that stacks can be used to solve the problem.

Once you agree that a stack is the appropriate data structure for keeping the parentheses, the statement of the algorithm is straightforward. Starting with an empty stack, process the parenthesis strings from left to right. If a symbol is an opening parenthesis, push it on the stack as a signal that a corresponding closing symbol needs to appear later. If, on the other hand, a symbol is a closing parenthesis, pop the stack. As long as it is possible to pop the stack to match every closing symbol, the parentheses remain balanced. If at any time there is no opening symbol on the stack to match a closing symbol, the string is not balanced properly. At the end of the string, when all symbols have been processed, the stack should be empty.

```python
 1 import Stack #import the Stack class as previously defined
 2 
 3 def par_checker(symbol_string):
 4    s = Stack()
 5    balanced = True
 6    index = 0
 7    while index < len(symbol_string) and balanced:
 8       symbol = symbol_string[index]
 9       if symbol == "(":
10          s.push(symbol)
11       else:
12          if s.is_empty():
13             balanced = False
14          else:
15             s.pop()
16 
17       index = index + 1
18 
19    if balanced and s.is_empty():
20       return True
21    else:
22       return False
23 
24 print(par_checker('((()))'))
25 print(par_checker('(()'))
```

This function, par_checker, assumes that a Stack class is available and returns a boolean result as to whether the string of parentheses is balanced. Note that the boolean variable balanced is initialized to True as there is no reason to assume otherwise at the start. If the current symbol is (, then it is pushed on the stack (lines 9–10). Note also in line 15 that pop simply removes a symbol from the stack. The returned value is not used since we know it must be an opening symbol seen earlier. At the end (lines 19–22), as long as the expression is balanced and the stack has been completely cleaned off, the string represents a correctly balanced sequence of parentheses.

3.4.4 Balanced Symbols (A General Case)

he balanced parentheses problem shown above is a specific case of a more general situation that arises in many programming languages. The general problem of balancing and nesting different kinds of opening and closing symbols occurs frequently. For example, in Python square brackets, `[` and `]`, are used for lists; curly braces, `{` and `}`, are used for dictionaries; and parentheses, `(` and `)`, are used for tuples and arithmetic expressions. It is possible to mix symbols as long as each maintains its own open and close relationship. Strings of symbols such as

```
{ { ( [ ] [ ] ) } ( ) }
```

```
[ [ { { ( ( ) ) } } ] ]

[ ] [ ] [ ] ( ) { }
```

are properly balanced in that not only does each opening symbol have a corresponding closing symbol, but the types of symbols match as well.

Compare those with the following strings that are not balanced:

```
( [ ) ]

( ( ( ) ] ) )

[ { ( ) ]
```

The simple parentheses checker from the previous section can easily be extended to handle these new types of symbols. Recall that each opening symbol is simply pushed on the stack to wait for the matching closing symbol to appear later in the sequence. When a closing symbol does appear, the only difference is that we must check to be sure that it correctly matches the type of the opening symbol on top of the stack. If the two symbols do not match, the string is not balanced. Once again, if the entire string is processed and nothing is left on the stack, the string is correctly balanced.

The Python program to implement this is shown below The only change appears in line 16 where we call a helper function, matches, to assist with symbol-matching. Each symbol that is removed from the stack must be checked to see that it matches the current closing symbol. If a mismatch occurs, the boolean variable balanced is set to False.

```
 1  import Stack # As previously defined
 2
 3  # Completed extended par_checker for: [,{,(,),},]
 4
 5  def par_checker(symbol_string):
 6     s = Stack()
 7     balanced = True
 8     index = 0
 9     while index < len(symbol_string) and balanced:
10        symbol = symbol_string[index]
11        if symbol in "([{":
12           s.push(symbol)
13        else:
14           if s.is_empty():
15              balanced = False
16           else:
17              top = s.pop()
18              if not matches(top, symbol):
19                    balanced = False
20        index = index + 1
```

```
    if balanced and s.is_empty():
        return True
    else:
        return False

def matches(open, close):
    opens = "([{"
    closes = ")]}"
    return opens.index(open) == closes.index(close)


print(par_checker('{{([][])}()}'))
print(par_checker('[{()]'))
```

These two examples show that stacks are very important data structures for the processing of language constructs in computer science. Almost any notation you can think of has some type of nested symbol that must be matched in a balanced order. There are a number of other important uses for stacks in computer science. We will continue to explore them in the next sections.

3.4.5 Converting Decimal Numbers to Binary Numbers

In your study of computer science, you have probably been exposed in one way or another to the idea of a binary number. Binary representation is important in computer science since all values stored within a computer exist as a string of binary digits, a string of 0s and 1s. Without the ability to convert back and forth between common representations and binary numbers, we would need to interact with computers in very awkward ways.

Integer values are common data items. They are used in computer programs and computation all the time. We learn about them in math class and of course represent them using the decimal number system, or base 10. The decimal number 233_10 and its corresponding binary equivalent 11101001_2 are interpreted respectively as $2*10^2+3*10^1+3*10^0$ and $1*2^7+1*2^6+1*2^5+0*2^4+1*2^3+0*2^2+0*2^1+1*2^0$

But how can we easily convert integer values into binary numbers? The answer is an algorithm called “Divide by 2” that uses a stack to keep track of the digits for the binary result.

The Divide by 2 algorithm assumes that we start with an integer greater than 0. A simple iteration then continually divides the decimal number by 2 and keeps track of the remainder. The first division by 2 gives information as to whether the value is even or odd. An even value will have a remainder of 0. It will have the digit 0 in the ones place. An odd value will have a remainder of 1 and will have the digit 1 in the ones place. We think about building our binary number as a sequence of digits; the first remainder we compute will actually be the last digit in the sequence. As shown in Figure 3.5, we again see the reversal property that signals that a stack is likely to be the appropriate data structure for solving the problem.

The Python code in below implements the Divide by 2 algorithm. The function divide_by_2 takes an argument that is a decimal number and repeatedly divides it by 2. Line 7 uses the built-in modulo operator, %, to extract the remainder and line 8 then pushes it on the stack. After the division process reaches 0, a binary string is constructed in lines 11–13. Line

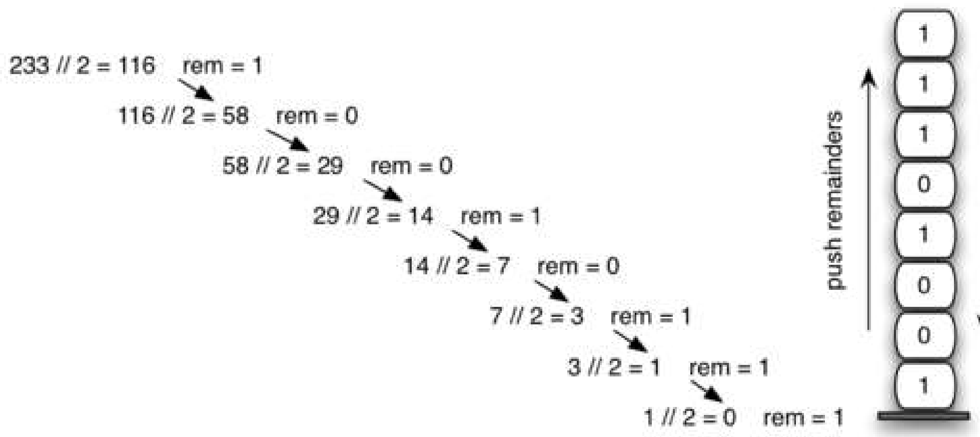


Figure 3.5: Decimal-To-Binary Conversion)

11 creates an empty string. The binary digits are popped from the stack one at a time and appended to the right-hand end of the string. The binary string is then returned.

```
1  import Stack # As previously defined
2
3  def divide_by_2(dec_number):
4     rem_stack = Stack()
5
6     while dec_number > 0:
7        rem = dec_number % 2
8        rem_stack.push(rem)
9        dec_number = dec_number // 2
10
11    bin_string = ""
12    while not rem_stack.is_empty():
13       bin_string = bin_string + str(rem_stack.pop())
14
15    return bin_string
16
17 print(divide_by_2(42))
```

he algorithm for binary conversion can easily be extended to perform the conversion for any base. In computer science it is common to use a number of different encodings. The most common of these are binary, octal (base 8), and hexadecimal (base 16).

The decimal number 233 and its corresponding octal and hexadecimal equivalents 351_8 and $E9_16$ are interpreted as $3 * 8^2 + 5 * 8^1 + 1 * 8^0$ and $14 * 16^1 + 9 * 16^0$

The function divide_by_2 can be modified to accept not only a decimal value but also a base for the intended conversion. The “Divide by 2” idea is simply replaced with a more general “Divide by base.” A new function called base_converter, shown below, takes a decimal number and any base between 2 and 16 as parameters. The remainders are still pushed onto the stack until the value being converted becomes 0. The same left-to-right string construction technique can be used with one slight change. Base 2 through base 10 numbers need a maximum of 10 digits, so the typical digit characters 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 work fine. The problem comes when we go beyond base 10. We can no longer simply use the remainders, as they are themselves represented as two-digit decimal numbers. Instead we need to create a set of digits that can be used to represent those remainders beyond 9.

```
import Stack # As previously defined

def base_converter(dec_number, base):
   digits = "0123456789ABCDEF"

   rem_stack = Stack()

   while dec_number > 0:
      rem = dec_number % base
      rem_stack.push(rem)
      dec_number = dec_number // base

   new_string = ""
   while not rem_stack.is_empty():
      new_string = new_string + digits[rem_stack.pop()]

   return new_string

print(base_converter(25, 2))
print(base_converter(25, 16))
```

A solution to this problem is to extend the digit set to include some alphabet characters. For example, hexadecimal uses the ten decimal digits along with the first six alphabet characters for the 16 digits. To implement this, a digit string is created that stores the digits in their corresponding positions. 0 is at position 0, 1 is at position 1, A is at position 10, B is at position 11, and so on. When a remainder is removed from the stack, it can be used to index into the digit string and the correct resulting digit can be appended to the answer. For example, if the remainder 13 is removed from the stack, the digit D is appended to the resulting string.

3.4.6 Self Check

1. What is the value of 25 expressed as an octal number?
2. What is the value of 256 expressed as a hexidecimal number?
3. What is the value of 26 expressed in base 26?

3.4.7 Infix, Prefix, and Postfix Expressions

When you write an arithmetic expression such as $B*C$, the form of the expression provides you with information so that you can interpret it correctly. In this case we know that the variable B is being multiplied by the variable C since the multiplication operator $*$ appears between them in the expression. This type of notation is referred to as infix since the operator is in between the two operands that it is working on.

Consider another infix example, $A + B * C$. The operators $+$ and $*$ still appear between the operands, but there is a problem. Which operands do they work on? Does the $+$ work on A and B or does the $*$ take B and C? The expression seems ambiguous.

In fact, you have been reading and writing these types of expressions for a long time and they do not cause you any problem. The reason for this is that you know something about the operators $+$ and $*$. Each operator has a **precedence** level. Operators of higher precedence are used before operators of lower precedence. The only thing that can change that order is the presence of parentheses. The precedence order for arithmetic operators places multiplication and division above addition and subtraction. If two operators of equal precedence appear, then a left-to-right ordering or associativity is used.

Let's interpret the troublesome expression $A + B * C$ using operator precedence. B and C are multiplied first, and A is then added to that result. $(A + B) * C$ would force the addition of A and B to be done first before the multiplication. In expression $A + B + C$, by precedence (via associativity), the leftmost $+$ would be done first.

Although all this may be obvious to you, remember that computers need to know exactly what operators to perform and in what order. One way to write an expression that guarantees there will be no confusion with respect to the order of operations is to create what is called a fully parenthesized expression. This type of expression uses one pair of parentheses for each operator. The parentheses dictate the order of operations; there is no ambiguity. There is also no need to remember any precedence rules.

The expression $A + B * C + D$ can be rewritten as $((A + (B * C)) + D)$ to show that the multiplication happens first, followed by the leftmost addition. $A + B + C + D$ can be written as $(((A + B) + C) + D)$ since the addition operations associate from left to right.

There are two other very important expression formats that may not seem obvious to you at first. Consider the infix expression $A + B$. What would happen if we moved the operator before the two operands? The resulting expression would be $+AB$. Likewise, we could move the operator to the end. We would get $AB+$. These look a bit strange.

These changes to the position of the operator with respect to the operands create two new expression formats, prefix and postfix. Prefix expression notation requires that all operators precede the two operands that they work on. Postfix, on the other hand, requires that its operators come after the corresponding operands. A few more examples should help to make this a bit clearer (see Table 3.2).

$A + B * C$ would be written as $+A * BC$ in prefix. The multiplication operator comes immediately before the operands B and C, denoting that $*$ has precedence over $+$. The addition operator then appears before the A and the result of the multiplication.

In postfix, the expression would be $ABC * +$. Again, the order of operations is preserved since the $*$ appears immediately after the B and the C, denoting that $*$ has precedence, with $+$ coming after. Although the operators moved and now appear either before or after their respective operands, the order of the operands stayed exactly the same relative to one another.

Now consider the infix expression $(A + B) * C$. Recall that in this case, infix requires the parentheses to force the performance of the addition before the multiplication. However, when

Infix Expression	**Prefix Expression**	**Postfix Expression**
$A + B$	$+AB$	$AB+$
$A + B * C$	$+A * BC$	$ABC * +$

Table 3.2: Examples of Infix, Prefix, and Postfix

Infix Expression	**Prefix Expression**	**Postfix Expression**
$(A + B) * C$	$* + ABC$	$AB + C*$

Table 3.3: An Expression with Parentheses

$A+B$ was written in prefix, the addition operator was simply moved before the operands, $+AB$. The result of this operation becomes the first operand for the multiplication. The multiplication operator is moved in front of the entire expression, giving us $* + ABC$. Likewise, in postfix $AB+$ forces the addition to happen first. The multiplication can be done to that result and the remaining operand C. The proper postfix expression is then $AB + C*$.

Consider these three expressions again (see Table 3.3). Something very important has happened. Where did the parentheses go? Why don't we need them in prefix and postfix? The answer is that the operators are no longer ambiguous with respect to the operands that they work on. Only infix notation requires the additional symbols. The order of operations within prefix and postfix expressions is completely determined by the position of the operator and nothing else. In many ways, this makes infix the least desirable notation to use.

Table 3.4 shows some additional examples of infix expressions and the equivalent prefix and postfix expressions. Be sure that you understand how they are equivalent in terms of the order of the operations being performed.

3.4.8 Conversion of Infix Expressions to Prefix and Postfix

So far, we have used ad hoc methods to convert between infix expressions and the equivalent prefix and postfix expression notations. As you might expect, there are algorithmic ways to perform the conversion that allow any expression of any complexity to be correctly transformed.

The first technique that we will consider uses the notion of a fully parenthesized expression that was discussed earlier. Recall that $A + B * C$ can be written as $(A + (B * C))$ to show explicitly that the multiplication has precedence over the addition. On closer observation, however, you can see that each parenthesis pair also denotes the beginning and the end of an operand pair with the corresponding operator in the middle.

Look at the right parenthesis in the subexpression $(B * C)$ above. If we were to move the

Infix Expression	**Prefix Expression**	**Postfix Expression**
$A + B * C + D$	$+ + A * BCD$	$ABC * +D+$
$(A + B) * (C + D)$	$* + AB + CD$	$AB + CD + *$
$A * B + C * D$	$+ * AB * CD$	$AB * CD * +$
$A + B + C + D$	$+ + +ABCD$	$AB + C + D+$

Table 3.4: Additional Examples of Infix, Prefix, and Postfix

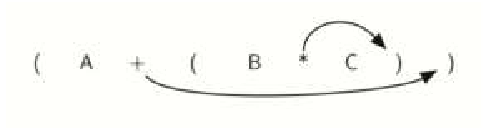


Figure 3.6: Moving Operators to the Right for Postfix Notation)

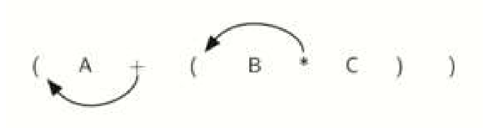


Figure 3.7: Moving Operators to the Left for Prefix Notation)

multiplication symbol to that position and remove the matching left parenthesis, giving us $BC*$, we would in effect have converted the subexpression to postfix notation. If the addition operator were also moved to its corresponding right parenthesis position and the matching left parenthesis were removed, the complete postfix expression would result (See figure 3.6)

If we do the same thing but instead of moving the symbol to the position of the right parenthesis, we move it to the left, we get prefix notation (see Figure 3.7). The position of the parenthesis pair is actually a clue to the final position of the enclosed operator.

So in order to convert an expression, no matter how complex, to either prefix or postfix notation, fully parenthesize the expression using the order of operations. Then move the enclosed operator to the position of either the left or the right parenthesis depending on whether you want prefix or postfix notation.

Here is a more complex expression: $(A + B) * C - (D - E) * (F + G)$. Figure 3.8 shows the conversion to postfix and prefix notations.

3.4.9 General Infix-to-Postfix Conversion

We need to develop an algorithm to convert any infix expression to a postfix expression. To do this we will look closer at the conversion process.

Consider once again the expression $A + B * C$. As shown above, $ABC * +$ is the postfix equivalent. We have already noted that the operands A, B, and C stay in their relative positions. It is only the operators that change position. Let us look again at the operators in the infix expression. The first operator that appears from left to right is $+$. However, in the postfix expression, $+$ is at the end since the next operator, $*$, has precedence over addition. The order

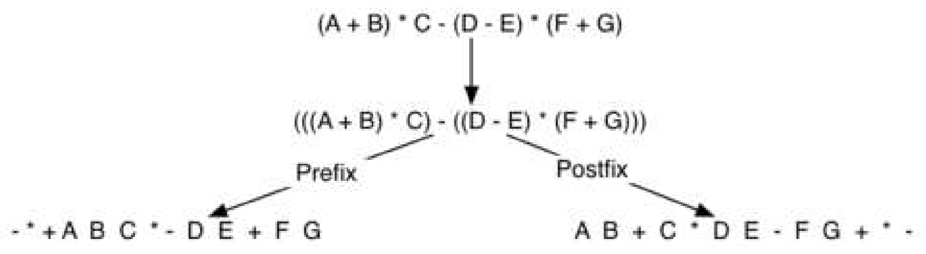

(A + B) * C - (D - E) * (F + G)
(((A + B) * C) - ((D - E) * (F + G)))
Prefix
Postfix
- * + A B C * - D E + F G
A B + C * D E - F G + * -

Figure 3.8: Converting a Complex Expression to Prefix and Postfix Notations)

of the operators in the original expression is reversed in the resulting postfix expression.

As we process the expression, the operators have to be saved somewhere since their corresponding right operands are not seen yet. Also, the order of these saved operators may need to be reversed due to their precedence. This is the case with the addition and the multiplication in this example. Since the addition operator comes before the multiplication operator and has lower precedence, it needs to appear after the multiplication operator is used. Because of this reversal of order, it makes sense to consider using a stack to keep the operators until they are needed.

What about $(A + B) * C$? Recall that $AB + C*$ is the postfix equivalent. Again, processing this infix expression from left to right, we see $+$ first. In this case, when we see $*$, $+$ has already been placed in the result expression because it has precedence over $*$ by virtue of the parentheses. We can now start to see how the conversion algorithm will work. When we see a left parenthesis, we will save it to denote that another operator of high precedence will be coming. That operator will need to wait until the corresponding right parenthesis appears to denote its position (recall the fully parenthesized technique). When that right parenthesis does appear, the operator can be popped from the stack.

As we scan the infix expression from left to right, we will use a stack to keep the operators. This will provide the reversal that we noted in the first example. The top of the stack will always be the most recently saved operator. Whenever we read a new operator, we will need to consider how that operator compares in precedence with the operators, if any, already on the stack.

Assume the infix expression is a string of tokens delimited by spaces. The operator tokens are $*$, $/$, $+$, and $-$, along with the left and right parentheses, (and). The operand tokens are the single-character identifiers A, B, C, and so on. The following steps will produce a string of tokens in postfix order.

1. Create an empty stack called op_stack for keeping operators. Create an empty list for output.
2. Convert the input infix string to a list by using the string method split.
3. Scan the token list from left to right.
    - If the token is an operand, append it to the end of the output list.
    - If the token is a left parenthesis, push it on the op_stack.
    - If the token is a right parenthesis, pop the op_stack until the corresponding left parenthesis is removed. Append each operator to the end of the output list.
    - If the token is an operator, $*$, $/$, $+$, or $-$, push it on the op_stack. However, first remove any operators already on the op_stack that have higher or equal precedence and append them to the output list.

    When the input expression has been completely processed, check the op_stack. Any operators still on the stack can be removed and appended to the end of the output list.

Figure 3.9 shows the conversion algorithm working on the expression $A * B + C * D$. Note that the first $*$ operator is removed upon seeing the $+$ operator. Also, $+$ stays on the stack when the second $*$ occurs, since multiplication has precedence over addition. At the end of the

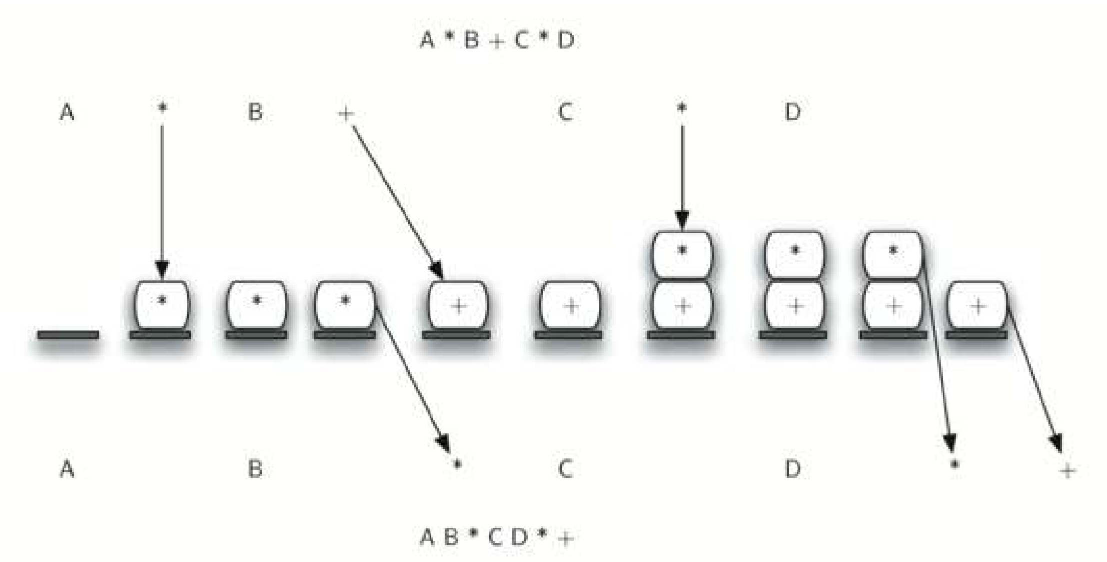


Figure 3.9: Converting $A * B + C * D$ to Postfix Notation)

infix expression the stack is popped twice, removing both operators and placing $+$ as the last operator in the postfix expression.

In order to code the algorithm in Python, we will use a dictionary called prec to hold the precedence values for the operators. This dictionary will map each operator to an integer that can be compared against the precedence levels of other operators (we have arbitrarily used the integers 3, 2, and 1). The left parenthesis will receive the lowest value possible. This way any operator that is compared against it will have higher precedence and will be placed on top of it. Line 15 defines the operands to be any upper-case character or digit. The complete conversion function is shown below.

```
1  import Stack  # As previously defined
2
3  def infix_to_postfix(infix_expr):
4     prec = {}
5     prec["*"] = 3
6     prec["/"] = 3
7     prec["+"] = 2
8     prec["-"] = 2
9     prec["("] = 1
10    op_stack = Stack()
11    postfix_list = []
12    token_list = infix_expr.split()
13
14    for token in token_list:
15       if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in
            "0123456789":
16          postfix_list.append(token)
17       elif token == '(':
18          op_stack.push(token)
19       elif token == ')':
20          top_token = op_stack.pop()
```

```
            while top_token != '(':
                postfix_list.append(top_token)
                top_token = op_stack.pop()
        else:
            while (not op_stack.is_empty()) and \
               (prec[op_stack.peek()] >= prec[token]):
                  postfix_list.append(op_stack.pop())
            op_stack.push(token)

    while not op_stack.is_empty():
        postfix_list.append(op_stack.pop())
    return " ".join(postfix_list)

print(infix_to_postfix("A * B + C * D"))
print(infix_to_postfix("( A + B ) * C - ( D - E ) * ( F + G )"))
...
```

A few more examples of execution in the Python shell are shown below.

```
>>> infix_to_postfix("( A + B ) * ( C + D )")
'A B + C D + *'
>>> infix_to_postfix("( A + B ) * C")
'A B + C *'
>>> infix_to_postfix("A + B * C")
'A B C * +'
>>>
```

3.4.10 Postfix Evaluation

As a final stack example, we will consider the evaluation of an expression that is already in postfix notation. In this case, a stack is again the data structure of choice. However, as you scan the postfix expression, it is the operands that must wait, not the operators as in the conversion algorithm above. Another way to think about the solution is that whenever an operator is seen on the input, the two most recent operands will be used in the evaluation.

To see this in more detail, consider the postfix expression $4 5 6 * +$. As you scan the expression from left to right, you first encounter the operands 4 and 5. At this point, you are still unsure what to do with them until you see the next symbol. Placing each on the stack ensures that they are available if an operator comes next.

In this case, the next symbol is another operand. So, as before, push it and check the next symbol. Now we see an operator, $*$. This means that the two most recent operands need to be used in a multiplication operation. By popping the stack twice, we can get the proper operands and then perform the multiplication (in this case getting the result 30).

We can now handle this result by placing it back on the stack so that it can be used as an operand for the later operators in the expression. When the final operator is processed, there will be only one value left on the stack. Pop and return it as the result of the expression. Figure 3.10 shows the stack contents as this entire example expression is being processed.

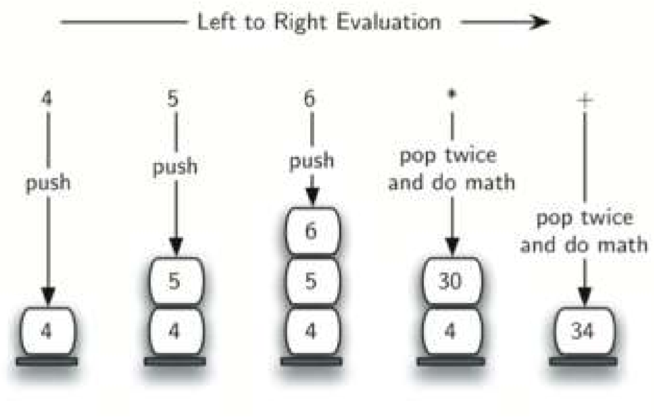


Figure 3.10: Sack Contents During Evaluation)

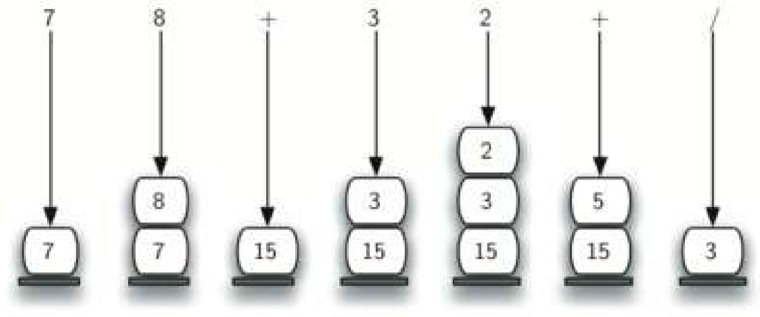


Figure 3.11: A More Complex Example of Evaluation)

Figure 3.11 shows a slightly more complex example, $78+32+/$. There are two things to note in this example. First, the stack size grows, shrinks, and then grows again as the subexpressions are evaluated. Second, the division operation needs to be handled carefully. Recall that the operands in the postfix expression are in their original order since postfix changes only the placement of operators. When the operands for the division are popped from the stack, they are reversed. Since division is *not* a commutative operator, in other words $15/5$ is not the same as $5/15$, we must be sure that the order of the operands is not switched.

Assume the postfix expression is a string of tokens delimited by spaces. The operators are $*$, $/$, $+$, and $-$, and the operands are assumed to be single-digit integer values. The output will be an integer result.

1. Create an empty stack called operand_stack.
2. Convert the string to a list by using the string method split.
3. Scan the token list from left to right.
    - If the token is an operand, convert it from a string to an integer and push the value onto the operand_stack.
    - If the token is an operator, $*$, $/$, $+$, or $-$, it will need two operands. Pop the operand_stack twice. The first pop is the second operand and the second pop is the first operand. Perform the arithmetic operation. Push the result back on the operand_stack.

4. When the input expression has been completely processed, the result is on the stack. Pop the operand_stack and return the value.

The complete function for the evaluation of postfix expressions is shown below. To assist with the arithmetic, a helper function do_math is defined that will take two operands and an operator and then perform the proper arithmetic operation.

```python
import Stack  # As previously defined

def postfix_eval(postfix_expr):
   operand_stack = Stack()
   token_list = postfix_expr.split()

   for token in token_list:
      if token in "0123456789":
         operand_stack.push(int(token))
      else:
         operand2 = operand_stack.pop()
         operand1 = operand_stack.pop()
         result = do_math(token, operand1, operand2)
         operand_stack.push(result)
   return operand_stack.pop()

def do_math(op, op1, op2):
   if op == "*":
      return op1 * op2
   elif op == "/":
      return op1 / op2
   elif op == "+":
      return op1 + op2
   else:
      return op1 - op2

print(postfix_eval('7 8 + 3 2 + /'))
```

It is important to note that in both the postfix conversion and the postfix evaluation programs we assumed that there were no errors in the input expression. Using these programs as a starting point, you can easily see how error detection and reporting can be included. We leave this as an exercise at the end of the chapter.

Self Check

1. Convert the following expression to postfix $10 + 3 * 5/(16 - 4)$.
2. Modify the infix_to_postfix function so that it can convert the following expression: $5 * 3\hat{}(4 - 2)$

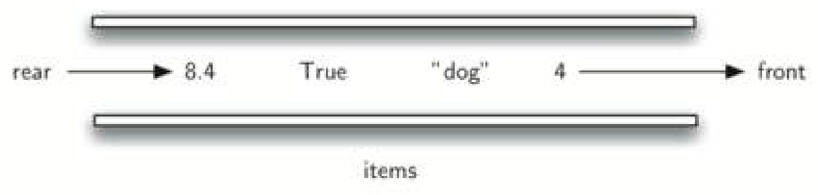


Figure 3.12: A Queue of Python Data Objects)

3.5 Queues

We now turn our attention to another linear data structure. This one is called **queue**. Like stacks, queues are relatively simple and yet can be used to solve a wide range of important problems.

3.5.1 What Is a Queue?

A queue is an ordered collection of items where the addition of new items happens at one end, called the "rear," and the removal of existing items occurs at the other end, commonly called the "front." As an element enters the queue it starts at the rear and makes its way toward the front, waiting until that time when it is the next element to be removed.

The most recently added item in the queue must wait at the end of the collection. The item that has been in the collection the longest is at the front. This ordering principle is sometimes called FIFO, first-in first-out. It is also known as "first-come first-served."

The simplest example of a queue is the typical line that we all participate in from time to time. We wait in a line for a movie, we wait in the check-out line at a grocery store, and we wait in the cafeteria line (so that we can pop the tray stack). Well-behaved lines, or queues, are very restrictive in that they have only one way in and only one way out. There is no jumping in the middle and no leaving before you have waited the necessary amount of time to get to the front. Figure 3.12 shows a simple queue of Python data objects.

Computer science also has common examples of queues. Our computer laboratory has 30 computers networked with a single printer. When students want to print, their print tasks "get in line" with all the other printing tasks that are waiting. The first task in is the next to be completed. If you are last in line, you must wait for all the other tasks to print ahead of you. We will explore this interesting example in more detail later.

In addition to printing queues, operating systems use a number of different queues to control processes within a computer. The scheduling of what gets done next is typically based on a queuing algorithm that tries to execute programs as quickly as possible and serve as many users as it can. Also, as we type, sometimes keystrokes get ahead of the characters that appear on the screen. This is due to the computer doing other work at that moment. The keystrokes are being placed in a queue-like buffer so that they can eventually be displayed on the screen in the proper order.

Queue Operation	Queue Contents	Return Value
`q.is_empty()`	`[]`	`True`
`q.enqueue(4)`	`[4]`	
`q.enqueue('dog')`	`['dog',4]`	
`q.enqueue(True)`	`[True,'dog',4]`	
`q.size()`	`[True,'dog',4]`	`3`
`q.is_empty()`	`[True,'dog',4]`	`False`
`q.enqueue(8.4)`	`[8.4,True,'dog',4]`	
`q.dequeue()`	`[8.4,True,'dog']`	`4`
`q.dequeue()`	`[8.4,True]`	`'dog'`
`q.size()`	`[8.4,True]`	`2`

Table 3.5: Example Queue Operations

3.5.2 The Queue Abstract Data Type

The queue abstract data type is defined by the following structure and operations. A queue is structured, as described above, as an ordered collection of items which are added at one end, called the "rear," and removed from the other end, called the "front." Queues maintain a FIFO ordering property. The queue operations are given below.

- Queue() creates a new queue that is empty. It needs no parameters and returns an empty queue.
- enqueue(item) adds a new item to the rear of the queue. It needs the item and returns nothing.
- dequeue() removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.
- is_empty() tests to see whether the queue is empty. It needs no parameters and returns a boolean value.
- size() returns the number of items in the queue. It needs no parameters and returns an integer.

As an example, if we assume that q is a queue that has been created and is currently empty, then Table 3.5 shows the results of a sequence of queue operations. The queue contents are shown such that the front is on the right. 4 was the first item enqueued so it is the first item returned by dequeue.

3.5.3 Implementing A Queue in Python

It is again appropriate to create a new class for the implementation of the abstract data type queue. As before, we will use the power and simplicity of the list collection to build the internal representation of the queue.

We need to decide which end of the list to use as the rear and which to use as the front. The implementation shown below assumes that the rear is at position 0 in the list. This allows us to use the insert function on lists to add new elements to the rear of the queue. The pop operation can be used to remove the front element (the last element of the list). Recall that this also means that enqueue will be $O(n)$ and dequeue will be $O(1)$.

```
# Completed implementation of a queue ADT
class Queue:
   def __init__(self):
      self.items = []

   def is_empty(self):
      return self.items == []

   def enqueue(self, item):
      self.items.insert(0,item)

   def dequeue(self):
      return self.items.pop()

   def size(self):
      return len(self.items)
```

Self Check

Suppose you have the following series of queue operations:

```
q = Queue()
q.enqueue('hello')
q.enqueue('dog')
q.enqueue(3)
q.dequeue()
```

What items are left in the queue?

1. ‘hello’, ‘dog’
2. ‘dog’, 3
3. ‘hello’, 3
4. ‘hello’, ‘dog’, 3

3.5.4 Simulation: Hot Potato

One of the typical applications for showing a queue in action is to simulate a real situation that requires data to be managed in a FIFO manner. To begin, let’s consider the children’s game Hot Potato. In this game (see Figure 3.13) children line up in a circle and pass an item from neighbour to neighbour as fast as they can. At a certain point in the game, the action is stopped and the child who has the item (the potato) is removed from the circle. Play continues until only one child is left.

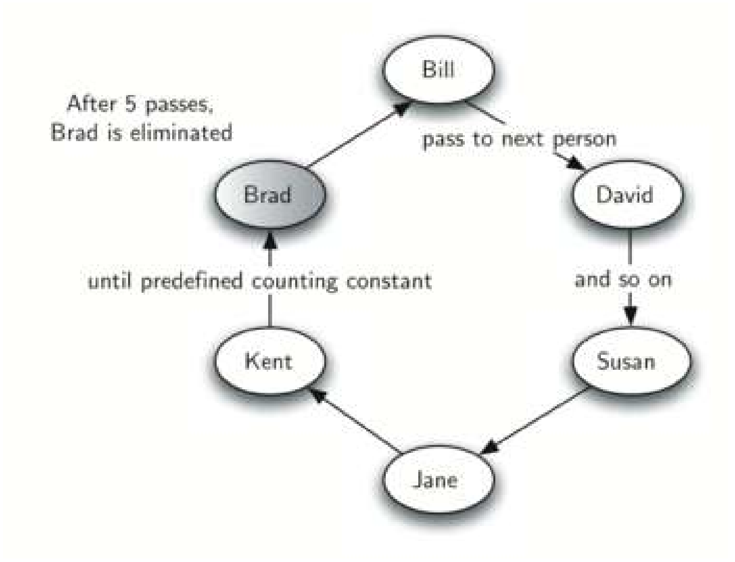


Figure 3.13: A Six Person Game of Hot Potato)

This game is a modern-day equivalent of the famous Josephus problem. Based on a legend about the famous first-century historian Flavius Josephus, the story is told that in the Jewish revolt against Rome, Josephus and 39 of his comrades held out against the Romans in a cave. With defeat imminent, they decided that they would rather die than be slaves to the Romans. They arranged themselves in a circle. One man was designated as number one, and proceeding clockwise they killed every seventh man. Josephus, according to the legend, was among other things an accomplished mathematician. He instantly figured out where he ought to sit in order to be the last to go. When the time came, instead of killing himself, he joined the Roman side. You can find many different versions of this story. Some count every third man and some allow the last man to escape on a horse. In any case, the idea is the same.

We will implement a general simulation of Hot Potato. Our program will input a list of names and a constant, call it "num" to be used for counting. It will return the name of the last person remaining after repetitive counting by num. What happens at that point is up to you.

To simulate the circle, we will use a queue (see Figure 3.14). Assume that the child holding the potato will be at the front of the queue. Upon passing the potato, the simulation will simply dequeue and then immediately enqueue that child, putting her at the end of the line. She will then wait until all the others have been at the front before it will be her turn again. After num dequeue/enqueue operations, the child at the front will be removed permanently and another cycle will begin. This process will continue until only one name remains (the size of the queue is 1).

A call to the hot_potato function using 7 as the counting constant returns Susan.

```
import Queue  # As previously defined

def hot_potato(name_list, num):
   sim_queue = Queue()
   for name in name_list:
      sim_queue.enqueue(name)
```

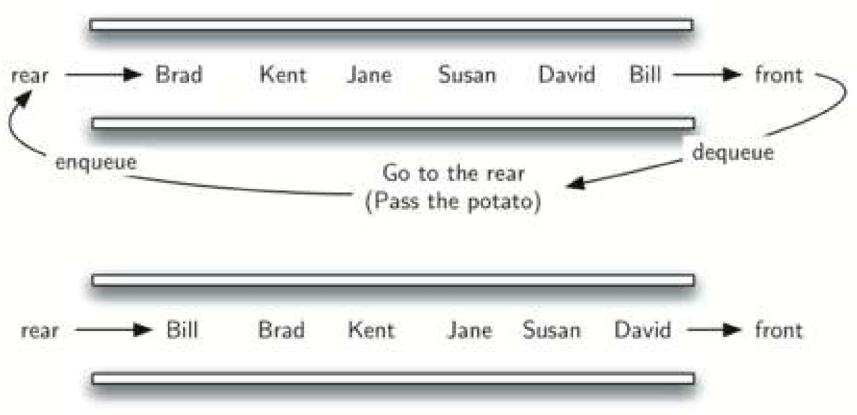


Figure 3.14: A Queue Implementation of Hot Potato)

```
    while sim_queue.size() > 1:
        for i in range(num):
            sim_queue.enqueue(sim_queue.dequeue())

        sim_queue.dequeue()

    return sim_queue.dequeue()

print(hot_potato(["Bill", "David", "Susan", "Jane", "Kent",
    "Brad"], 7))
```

Note that in this example the value of the counting constant is greater than the number of names in the list. This is not a problem since the queue acts like a circle and counting continues back at the beginning until the value is reached. Also, notice that the list is loaded into the queue such that the first name on the list will be at the front of the queue. Bill in this case is the first item in the list and therefore moves to the front of the queue. A variation of this implementation, described in the exercises, allows for a random counter.

3.5.5 Simulation: Printing Tasks

A more interesting simulation allows us to study the behavior of the printing queue described earlier in this section. Recall that as students send printing tasks to the shared printer, the tasks are placed in a queue to be processed in a first-come first-served manner. Many questions arise with this configuration. The most important of these might be whether the printer is capable of handling a certain amount of work. If it cannot, students will be waiting too long for printing and may miss their next class.

Consider the following situation in a computer science laboratory. On any average day about 10 students are working in the lab at any given hour. These students typically print up to twice during that time, and the length of these tasks ranges from 1 to 20 pages. The printer in the lab is older, capable of processing 10 pages per minute of draft quality. The printer could be switched to give better quality, but then it would produce only five pages per minute. The slower printing speed could make students wait too long. What page rate should be used?

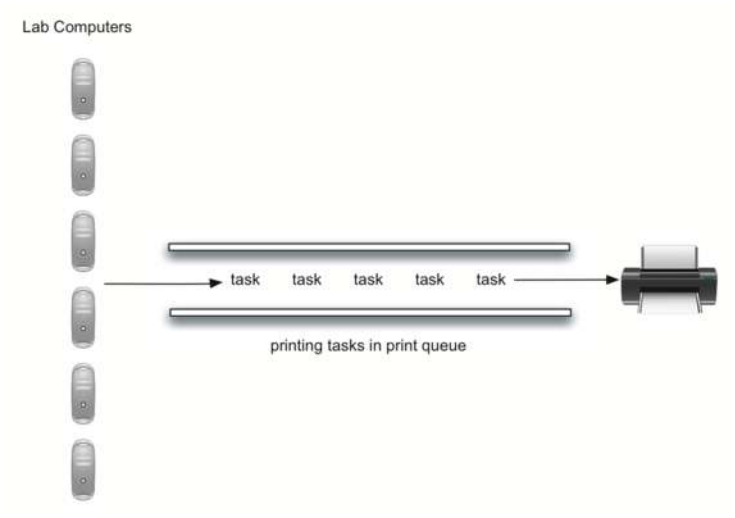


Figure 3.15: Computer Science Laboratory Printing Queue)

We could decide by building a simulation that models the laboratory. We will need to construct representations for students, printing tasks, and the printer (Figure 3.15). As students submit printing tasks, we will add them to a waiting list, a queue of print tasks attached to the printer. When the printer completes a task, it will look at the queue to see if there are any remaining tasks to process. Of interest for us is the average amount of time students will wait for their papers to be printed. This is equal to the average amount of time a task waits in the queue.

To model this situation we need to use some probabilities. For example, students may print a paper from 1 to 20 pages in length. If each length from 1 to 20 is equally likely, the actual length for a print task can be simulated by using a random number between 1 and 20 inclusive. This means that there is equal chance of any length from 1 to 20 appearing.

If there are 10 students in the lab and each prints twice, then there are 20 print tasks per hour on average. What is the chance that at any given second, a print task is going to be created? The way to answer this is to consider the ratio of tasks to time. Twenty tasks per hour means that on average there will be one task every 180 seconds:

$$\frac{20 \text{ tasks}}{1 \text{ hour}} \times \frac{1 \text{ hour}}{60 \text{ minutes}} \times \frac{1 \text{ minute}}{60 \text{ seconds}} = \frac{1 \text{ task}}{180 \text{ seconds}}$$

For every second we can simulate the chance that a print task occurs by generating a random number between 1 and 180 inclusive. If the number is 180, we say a task has been created. Note that it is possible that many tasks could be created in a row or we may wait quite a while for a task to appear. That is the nature of simulation. You want to simulate the real situation as closely as possible given that you know general parameters.

3.5.6 Main Simulation Steps

Here is the main simulation.

1. Create a queue of print tasks. Each task will be given a timestamp upon its arrival. The queue is empty to start.

2. For each second (current_second):
    - Does a new print task get created? If so, add it to the queue with the current_second as the timestamp.
    - If the printer is not busy and if a task is waiting,
        - Remove the next task from the print queue and assign it to the printer.
        - Subtract the timestamp from the current_second to compute the waiting time for that task.
        - Append the waiting time for that task to a list for later processing.
        - Based on the number of pages in the print task, figure out how much time will be required.
    - The printer now does one second of printing if necessary. It also subtracts one second from the time required for that task.
    - If the task has been completed, in other words the time required has reached zero, the printer is no longer busy.
3. After the simulation is complete, compute the average waiting time from the list of waiting times generated.

3.5.7 Python Implementation

To design this simulation we will create classes for the three real-world objects described above: Printer, Task, and PrintQueue.

The Printer class will need to track whether it has a current task. If it does, then it is busy and the amount of time needed can be computed from the number of pages in the task. The constructor will also allow the pages-per-minute setting to be initialized. The tick method decrements the internal timer and sets the printer to idle if the task is completed.

```python
class Printer:
   def __init__(self, ppm):
      self.page_rate = ppm
      self.current_task = None
      self.time_remaining = 0

   def tick(self):
      if self.current_task != None:
         self.time_remaining = self.time_remaining - 1
         if self.time_remaining <= 0:
            self.current_task = None

   def busy(self):
      if self.current_task != None:
         return True
      else:
         return False
   def start_next(self,new_task):
      self.current_task = new_task
      self.time_remaining = new_task.get_pages() * 60 /
         self.page_rate
```

The Task class will represent a single printing task. When the task is created, a random number generator will provide a length from 1 to 20 pages. We have chosen to use the randrange function from the random module.

```
>>> import random
>>> random.randrange(1, 21)
18
>>> random.randrange(1, 21)
8
>>>
```

Each task will also need to keep a timestamp to be used for computing waiting time. This timestamp will represent the time that the task was created and placed in the printer queue. The wait_time method can then be used to retrieve the amount of time spent in the queue before printing begins.

```
import random

class Task:
   def __init__(self, time):
      self.timestamp = time
      self.pages = random.randrange(1, 21)

   def get_stamp(self):
      return self.timestamp

   def get_pages(self):
      return self.pages

   def wait_time(self, current_time):
      return current_time - self.timestamp
```

The main simulation implements the algorithm described above. The print_queue object is an instance of our existing queue ADT. A boolean helper function, new_print_task, decides whether a new printing task has been created. We have again chosen to use the randrange function from the random module to return a random integer between 1 and 180. Print tasks arrive once every 180 seconds. By arbitrarily choosing 180 from the range of random integers, we can simulate this random event. The simulation function allows us to set the total time and the pages per minute for the printer.

```
import Queue  # As previously defined
import Printer  # As previously defined
import Task  # As previously defined

import random

def simulation(num_seconds, pages_per_minute):

   lab_printer = Printer(pages_per_minute)
   print_queue = Queue()
   waiting_times = []

   for current_second in range(num_seconds):

     if new_print_task():
       task = Task(current_second)
       print_queue.enqueue(task)

     if (not lab_printer.busy()) and (not print_queue.is_empty()):
       next_task = print_queue.dequeue()
       waiting_times.append(next_task.wait_time(current_second))
       lab_printer.start_next(next_task)

     lab_printer.tick()

   average_wait = sum(waiting_times) / len(waiting_times)
   print("Average Wait %6.2f secs %3d tasks remaining."
      %(average_wait, print_queue.size()))

def new_print_task():
   num = random.randrange(1, 181)
   if num == 180:
      return True
   else:
      return False

for i in range(10):
   simulation(3600, 5)
```

When we run the simulation, we should not be concerned that the results are different each time. This is due to the probabilistic nature of the random numbers. We are interested in the trends that may be occurring as the parameters to the simulation are adjusted. Here are some results.

First, we will run the simulation for a period of 60 minutes ($3,600$ seconds) using a page rate of five pages per minute. In addition, we will run 10 independent trials. Remember that because the simulation works with random numbers each run will return different results.

```
>>>for i in range(10):
      simulation(3600, 5)

Average Wait 165.38 secs 2 tasks remaining.
Average Wait 95.07 secs 1 tasks remaining.
Average Wait 65.05 secs 2 tasks remaining.
Average Wait 99.74 secs 1 tasks remaining.
Average Wait 17.27 secs 0 tasks remaining.
Average Wait 239.61 secs 5 tasks remaining.
Average Wait 75.11 secs 1 tasks remaining.
Average Wait 48.33 secs 0 tasks remaining.
Average Wait 39.31 secs 3 tasks remaining.
Average Wait 376.05 secs 1 tasks remaining.
```

After running our 10 trials we can see that the mean average wait time is 122.155 seconds. You can also see that there is a large variation in the average weight time with a minimum average of 17.27 seconds and a maximum of 239.61 seconds. You may also notice that in only two of the cases were all the tasks completed.

Now, we will adjust the page rate to 10 pages per minute, and run the 10 trials again, with a faster page rate our hope would be that more tasks would be completed in the one hour time frame.

```
>>>for i in range(10):
     simulation(3600, 10)

Average Wait 1.29 secs 0 tasks remaining.
Average Wait 7.00 secs 0 tasks remaining.
Average Wait 28.96 secs 1 tasks remaining.
Average Wait 13.55 secs 0 tasks remaining.
Average Wait 12.67 secs 0 tasks remaining.
Average Wait 6.46 secs 0 tasks remaining.
Average Wait 22.33 secs 0 tasks remaining.
Average Wait 12.39 secs 0 tasks remaining.
Average Wait 7.27 secs 0 tasks remaining.
Average Wait 18.17 secs 0 tasks remaining.
```

The code to run the simulation is as follows:

```
import Queue  # As previously defined

import random

# Completed program for the printer simulation

class Printer:
   def __init__(self, ppm):
      self.page_rate = ppm
      self.current_task = None
      self.time_remaining = 0

   def tick(self):
      if self.current_task != None:
         self.time_remaining = self.time_remaining - 1
         if self.time_remaining <= 0:
            self.current_task = None

    def busy(self):
        if self.current_task != None:
            return True
        else:
            return False

    def start_next(self, new_task):
        self.current_task = new_task
        self.time_remaining = new_task.get_pages() * 60/self.page_rate

class Task:
    def __init__(self, time):
        self.timestamp = time
        self.pages = random.randrange(1, 21)

    def get_stamp(self):
        return self.timestamp

    def get_pages(self):
        return self.pages

    def wait_time(self, current_time):
        return current_time - self.timestamp


def simulation(num_seconds, pages_per_minute):

    lab_printer = Printer(pages_per_minute)
    print_queue = Queue()
    waiting_times = []

    for current_second in range(num_seconds):

      if new_print_task():
         task = Task(current_second)
         print_queue.enqueue(task)

      if (not lab_printer.busy()) and (not print_queue.is_empty()):
        next_task = print_queue.dequeue()
        waiting_times.append(next_task.wait_time(current_second))
        lab_printer.startNext(next_task)

      lab_printer.tick()

    average_wait = sum(waiting_times) / len(waiting_times)
    print("Average Wait %6.2f secs %3d tasks remaining."
        %(average_wait, print_queue.size()))

def new_print_task():
    num = random.randrange(1, 181)
    if num == 180:
        return True
    else:
        return False

for i in range(10):
    simulation(3600, 5)
```

3.5.8 Discussion

We were trying to answer a question about whether the current printer could handle the task load if it were set to print with a better quality but slower page rate. The approach we took was to write a simulation that modeled the printing tasks as random events of various lengths and arrival times.

The output above shows that with 5 pages per minute printing, the average waiting time varied from a low of 17 seconds to a high of 376 seconds (about 6 minutes). With a faster printing rate, the low value was 1 second with a high of only 28. In addition, in 8 out of 10 runs at 5 pages per minute there were print tasks still waiting in the queue at the end of the hour.

Therefore, we are perhaps persuaded that slowing the printer down to get better quality may not be a good idea. Students cannot afford to wait that long for their papers, especially when they need to be getting on to their next class. A six-minute wait would simply be too long.

This type of simulation analysis allows us to answer many questions, commonly known as "what if" questions. All we need to do is vary the parameters used by the simulation and we can simulate any number of interesting behaviors. For example,

- What if enrolment goes up and the average number of students increases by 20?
- What if it is Saturday and students are not needing to get to class? Can they afford to wait?
- What if the size of the average print task decreases since Python is such a powerful language and programs tend to be much shorter?

These questions could all be answered by modifying the above simulation. However, it is important to remember that the simulation is only as good as the assumptions that are used to build it. Real data about the number of print tasks per hour and the number of students per hour was necessary to construct a robust simulation.

Self Check

How would you modify the printer simulation to reflect a larger number of students? Suppose that the number of students was doubled. You make need to make some reasonable assumptions about how this simulation was put together but what would you change? Modify the code. Also suppose that the length of the average print task was cut in half. Change the code to reflect that change. Finally How would you parameterize the number of students, rather than changing the code we would like to make the number of students a parameter of the simulation.

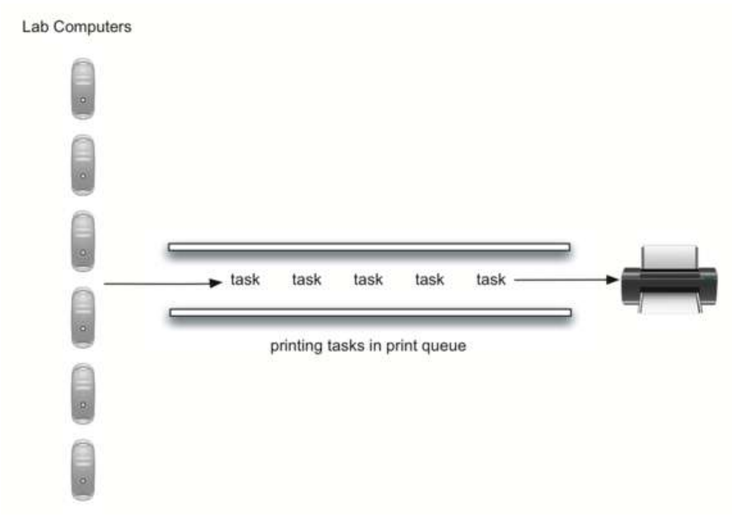


Figure 3.16: A Deque of Python Data Objects Queue)

3.6 Deques

We will conclude this introduction to basic data structures by looking at another variation on the theme of linear collections. However, unlike stack and queue, the deque (pronounced "deck") has very few restrictions. Also, be careful that you do not confuse the spelling of "deque" with the queue removal operation "dequeue."

3.6.1 What Is a Deque?

A deque, also known as a double-ended queue, is an ordered collection of items similar to the queue. It has two ends, a front and a rear, and the items remain positioned in the collection. What makes a deque different is the unrestrictive nature of adding and removing items. New items can be added at either the front or the rear. Likewise, existing items can be removed from either end. In a sense, this hybrid linear structure provides all the capabilities of stacks and queues in a single data structure. Figure 3.16 shows a deque of Python data objects.

It is important to note that even though the deque can assume many of the characteristics of stacks and queues, it does not require the LIFO and FIFO orderings that are enforced by those data structures. It is up to you to make consistent use of the addition and removal operations.

3.6.2 The Deque Abstract Data Type

The deque abstract data type is defined by the following structure and operations. A deque is structured, as described above, as an ordered collection of items where items are added and removed from either end, either front or rear. The deque operations are given below.

- Deque() creates a new deque that is empty. It needs no parameters and returns an empty deque.
- add_front(item) adds a new item to the front of the deque. It needs the item and returns nothing.

Deque Operation	Deque Contents	Return value
`d.is_empty()`	`[]`	`True`
`d.add_rear(4)`	`[4]`	
`d.add_rear('dog')`	`['dog',4,]`	
`d.add_front('cat')`	`['dog',4,'cat']`	
`d.add_front(True)`	`['dog',4,'cat',True]`	
`d.size()`	`['dog',4,'cat',True]`	`4`
`d.is_empty()`	`['dog',4,'cat',True]`	`False`
`d.add_rear(8.4)`	`[8.4,'dog',4,'cat',True]`	
`d.remove_rear()`	`['dog',4,'cat',True]`	`8.4`
`d.remove_front()`	`['dog',4,'cat']`	`True`

Table 3.6: Examples of Deque Operations

- add_rear(item) adds a new item to the rear of the deque. It needs the item and returns nothing.
- remove_front() removes the front item from the deque. It needs no parameters and returns the item. The deque is modified.
- remove_rear() removes the rear item from the deque. It needs no parameters and returns the item. The deque is modified.
- is_empty() tests to see whether the deque is empty. It needs no parameters and returns a boolean value.
- size() returns the number of items in the deque. It needs no parameters and returns an integer.

As an example, if we assume that d is a deque that has been created and is currently empty, then Table 3.6 shows the results of a sequence of deque operations. Note that the contents in front are listed on the right. It is very important to keep track of the front and the rear as you move items in and out of the collection as things can get a bit confusing.

3.6.3 Implementing a Deque in Python

As we have done in previous sections, we will create a new class for the implementation of the abstract data type deque. Again, the Python list will provide a very nice set of methods upon which to build the details of the deque. Our implementation will assume that the rear of the deque is at position 0 in the list.

```python
# Completed implementation of a deque ADT
class Deque:
   def __init__(self):
      self.items = []

   def is_empty(self):
      return self.items == []

   def add_front(self, item):
        self.items.append(item)

    def add_rear(self, item):
        self.items.insert(0,item)

    def remove_front(self):
        return self.items.pop()

    def remove_rear(self):
        return self.items.pop(0)

    def size(self):
        return len(self.items)
```

In remove_front we use the pop method to remove the last element from the list. However, in remove_rear, the pop(0) method must remove the first element of the list. Likewise, we need to use the insert method in add_rear since the append method assumes the addition of a new element to the end of the list.

You can see many similarities to Python code already described for stacks and queues. You are also likely to observe that in this implementation adding and removing items from the front is $O(1)$ whereas adding and removing from the rear is $O(n)$. This is to be expected given the common operations that appear for adding and removing items. Again, the important thing is to be certain that we know where the front and rear are assigned in the implementation.

3.6.4 Palindrome Checker

An interesting problem that can be easily solved using the deque data structure is the classic palindrome problem. A **palindrome** is a string that reads the same forward and backward, for example, *radar*, *toot*, and *madam*. We would like to construct an algorithm to input a string of characters and check whether it is a palindrome.

The solution to this problem will use a deque to store the characters of the string. We will process the string from left to right and add each character to the rear of the deque. At this point, the deque will be acting very much like an ordinary queue. However, we can now make use of the dual functionality of the deque. The front of the deque will hold the first character of the string and the rear of the deque will hold the last character (see Figure 3.17)

Since we can remove both of them directly, we can compare them and continue only if they match. If we can keep matching first and the last items, we will eventually either run out of characters or be left with a deque of size 1 depending on whether the length of the original string was even or odd. In either case, the string must be a palindrome.

```
import Deque  # As previously defined

def pal_checker(a_string):
    char_deque = Deque()

    for ch in a_string:
```

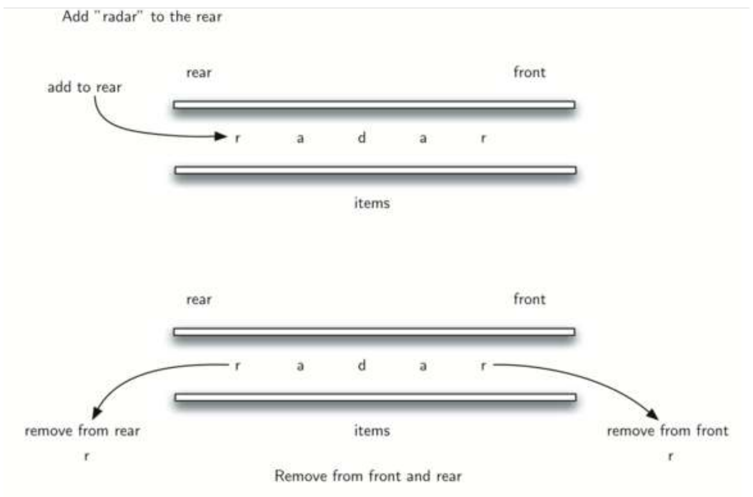


Figure 3.17: A Deque)

```
        char_deque.add_rear(ch)

    still_equal = True

    while char_deque.size() > 1 and still_equal:
        first = char_deque.remove_front()
        last = char_deque.remove_rear()
        if first != last:
            still_equal = False

    return still_equal

print(pal_checker("lsdkjfskf"))
print(pal_checker("radar"))
```

3.7 Lists

Throughout the discussion of basic data structures, we have used Python lists to implement the abstract data types presented. The list is a powerful, yet simple, collection mechanism that provides the programmer with a wide variety of operations. However, not all programming languages include a list collection. In these cases, the notion of a list must be implemented by the programmer.

A list is a collection of items where each item holds a relative position with respect to the others. More specifically, we will refer to this type of list as an unordered list. We can consider the list as having a first item, a second item, a third item, and so on. We can also refer to the beginning of the list (the first item) or the end of the list (the last item). For simplicity we will assume that lists cannot contain duplicate items.

For example, the collection of integers $54, 26, 93, 17, 77$, and 31 might represent a simple unordered list of exam scores. Note that we have written them as comma-delimited values, a common way of showing the list structure. Of course, Python would show this list as $[54, 26, 93, 17, 77, 31]$.

3.8 The Unordered List Abstract Data Type

The structure of an unordered list, as described above, is a collection of items where each item holds a relative position with respect to the others. Some possible unordered list operations are given below.

- List() creates a new list that is empty. It needs no parameters and returns an empty list.
- add(item) adds a new item to the list. It needs the item and returns nothing. Assume the item is not already in the list.
- remove(item) removes the item from the list. It needs the item and modifies the list. Assume the item is present in the list.
- search(item) searches for the item in the list. It needs the item and returns a boolean value.
- is_empty() tests to see whether the list is empty. It needs no parameters and returns a boolean value.
- size() returns the number of items in the list. It needs no parameters and returns an integer.
- append(item) adds a new item to the end of the list making it the last item in the collection. It needs the item and returns nothing. Assume the item is not already in the list.
- index(item) returns the position of item in the list. It needs the item and returns the index. Assume the item is in the list.
- insert(pos,item) adds a new item to the list at position pos. It needs the item and returns nothing. Assume the item is not already in the list and there are enough existing items to have position pos.
- pop() removes and returns the last item in the list. It needs nothing and returns an item. Assume the list has at least one item.
- pop(pos) removes and returns the item at position pos. It needs the position and returns the item. Assume the item is in the list.

3.9 Implementing an Unordered List: Linked Lists

In order to implement an unordered list, we will construct what is commonly known as a **linked list**. Recall that we need to be sure that we can maintain the relative positioning of the items. However, there is no requirement that we maintain that positioning in contiguous memory. For example, consider the collection of items shown in Figure 3.18. It appears that these values have been placed randomly. If we can maintain some explicit information in each item, namely

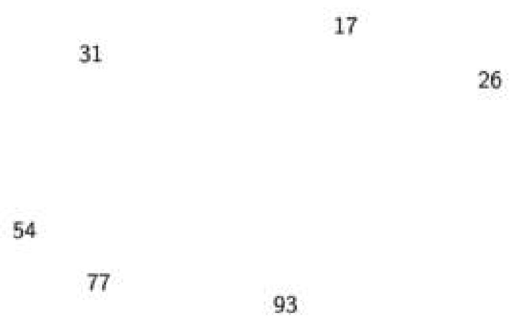


Figure 3.18: Items Not Constrained in Their Physical Placement

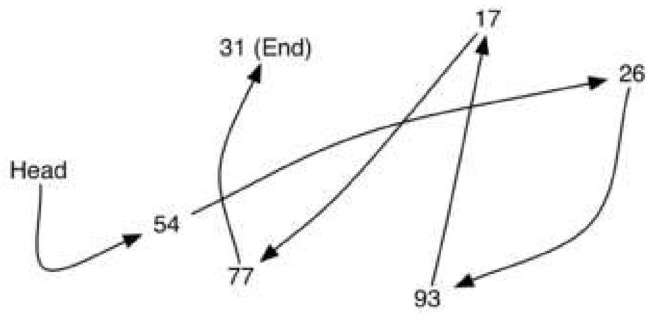


Figure 3.19: Relative Positions Maintained by Explicit Links.

the location of the next item (see Figure 3.19, then the relative position of each item can be expressed by simply following the link from one item to the next.

It is important to note that the location of the first item of the list must be explicitly specified. Once we know where the first item is, the first item can tell us where the second is, and so on. The external reference is often referred to as the head of the list. Similarly, the last item needs to know that there is no next item.

3.9.1 The Node Class

The basic building block for the linked list implementation is the **node**. Each node object must hold at least two pieces of information. First, the node must contain the list item itself. We will call this the **data field** of the node. In addition, each node must hold a reference to the next node. To construct a node, you need to supply the initial data value for the node. Evaluating the assignment statement below will yield a node object containing the value 93 (see Figure 3.20). You should note that we will typically represent a node object as shown in Figure 3.21. The Node class also includes the usual methods to access and modify the data and the next reference.

```
class Node:
   def __init__(self, init_data):
      self.data = init_data
      self.next = None

   def get_data(self):
      return self.data
```

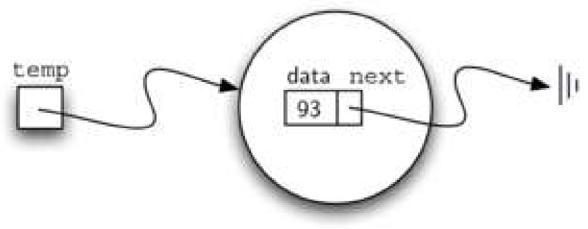


Figure 3.20: A Node Object Contains the Item and a Reference to the Next Node

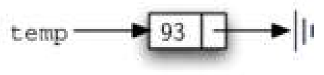


Figure 3.21: A Typical Representation for a Node.

```
    def get_next(self):
        return self.next

    def set_data(self, new_data):
        self.data = newdata

    def set_next(self, new_next):
        self.next = new_next
```

We create Node objects in the usual way.

```
>>> temp = Node(93)
>>> temp.get_data()
93
```

The special Python reference value None will play an important role in the Node class and later in the linked list itself. A reference to None will denote the fact that there is no next node. Note in the constructor that a node is initially created with next set to None. Since this is sometimes referred to as “grounding the node,” we will use the standard ground symbol to denote a reference that is referring to None. It is always a good idea to explicitly assign None to your initial next reference values.

3.9.2 The Unordered List Class

As we suggested above, the unordered list will be built from a collection of nodes, each linked to the next by explicit references. As long as we know where to find the first node (containing the first item), each item after that can be found by successively following the next links. With this in mind, the UnorderedList class must maintain a reference to the first node. The following code shows the constructor. Note that each list object will maintain a single reference to the head of the list.

```
class UnorderedList:
```

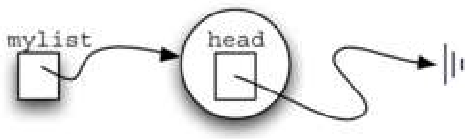


Figure 3.22: An Empty List

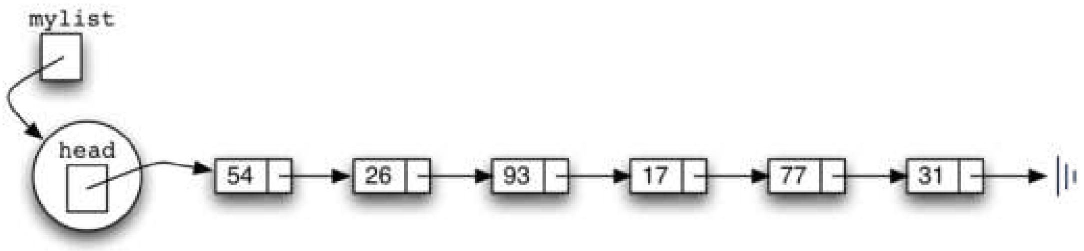


Figure 3.23: Linked List of Integers

```
    def __init__(self):
        self.head = None
```

Initially when we construct a list, there are no items. The assignment statement

```
>>> mylist = UnorderedList()
```

creates the linked list representation shown in Figure 3.22. As we discussed in the Node class, the special reference None will again be used to state that the head of the list does not refer to anything. Eventually, the example list given earlier will be represented by a linked list as shown in Figure 3.23. The head of the list refers to the first node which contains the first item of the list. In turn, that node holds a reference to the next node (the next item) and so on. It is very important to note that the list class itself does not contain any node objects. Instead it contains a single reference to only the first node in the linked structure.

The is_empty method simply checks to see if the head of the list is a reference to None. The result of the boolean expression self.head==None will only be true if there are no nodes in the linked list. Since a new list is empty, the constructor and the check for empty must be consistent with one another. This shows the advantage to using the reference None to denote the "end" of the linked structure. In Python, None can be compared to any reference. Two references are equal if they both refer to the same object. We will use this often in our remaining methods.

```
def is_empty(self):
    return self.head == None
```

So, how do we get items into our list? We need to implement the add method. However, before we can do that, we need to address the important question of where in the linked list to place the new item. Since this list is unordered, the specific location of the new item with respect to the other items already in the list is not important. The new item can go anywhere. With that in mind, it makes sense to place the new item in the easiest location possible.

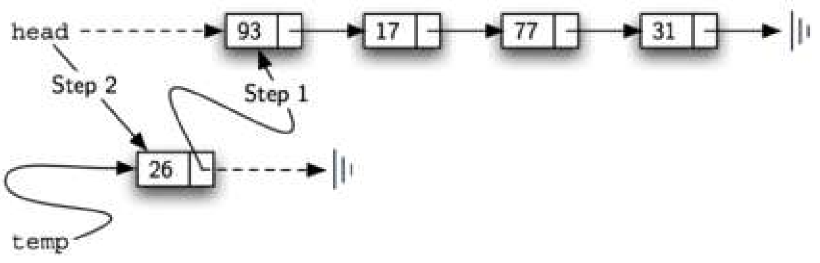


Figure 3.24: Adding a New Node is a Two-Step Process

Recall that the linked list structure provides us with only one entry point, the head of the list. All of the other nodes can only be reached by accessing the first node and then following next links. This means that the easiest place to add the new node is right at the head, or beginning, of the list. In other words, we will make the new item the first item of the list and the existing items will need to be linked to this new first item so that they follow.

The linked list shown in Figure 3.23 was built by calling the add method a number of times.

```
>>> mylist.add(31)
>>> mylist.add(77)
>>> mylist.add(17)
>>> mylist.add(93)
>>> mylist.add(26)
>>> mylist.add(54)
```

Note that since 31 is the first item added to the list, it will eventually be the last node on the linked list as every other item is added ahead of it. Also, since 54 is the last item added, it will become the data value in the first node of the linked list.

The add method is shown below. Each item of the list must reside in a node object. Line 2 creates a new node and places the item as its data. Now we must complete the process by linking the new node into the existing structure. This requires two steps as shown in Figure 3.24. Step 1 (line 3) changes the next reference of the new node to refer to the old first node of the list. Now that the rest of the list has been properly attached to the new node, we can modify the head of the list to refer to the new node. The assignment statement in line 4 sets the head of the list.

The order of the two steps described above is very important. What happens if the order of line 3 and line 4 is reversed? If the modification of the head of the list happens first, the result can be seen in Figure 3.25. Since the head was the only external reference to the list nodes, all of the original nodes are lost and can no longer be accessed.

```
def add(self, item):
   temp = Node(item)
   temp.set_next(self.head)
   self.head = temp
```

The next methods that we will implement-size, search, and remove-are all based on a technique known as linked list traversal. Traversal refers to the process of systematically visiting each

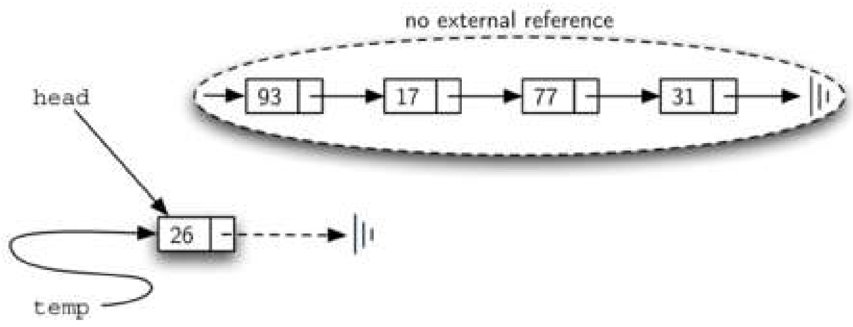


Figure 3.25: Result of Reversing the Order of the Two Steps.

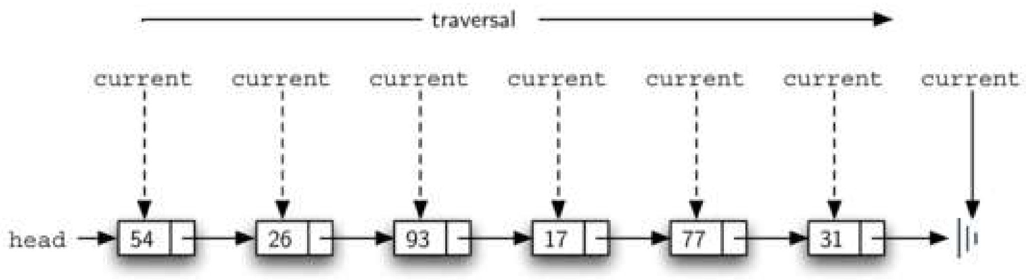


Figure 3.26: Traversing the Linked List from the Head to the End.

node. To do this we use an external reference that starts at the first node in the list. As we visit each node, we move the reference to the next node by "traversing" the next reference.

To implement the size method, we need to traverse the linked list and keep a count of the number of nodes that occurred. Below we show the Python code for counting the number of nodes in the list. The external reference is called current and is initialized to the head of the list in line 2. At the start of the process we have not seen any nodes so the count is set to 0. Lines 4–6 actually implement the traversal. As long as the current reference has not seen the end of the list (None), we move current along to the next node via the assignment statement in line 6. Again, the ability to compare a reference to None is very useful. Every time current moves to a new node, we add 1 to count. Finally, count gets returned after the iteration stops. Figure 3.26 shows this process as it proceeds down the list.

```
def size(self):
   current = self.head
   count = 0
   while current != None:
      count = count + 1
      current = current.get_next()

   return count
```

Searching for a value in a linked list implementation of an unordered list also uses the traversal technique. As we visit each node in the linked list we will ask whether the data stored there matches the item we are looking for. In this case, however, we may not have to traverse all the way to the end of the list. In fact, if we do get to the end of the list, that means that the item we are looking for must not be present. Also, if we do find the item, there is no need to continue.

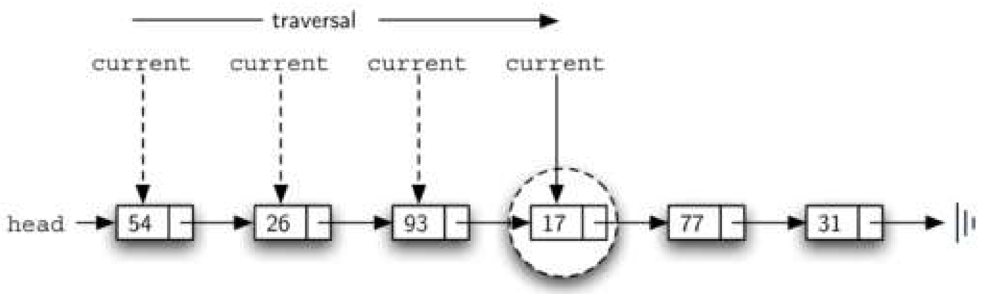


Figure 3.27: Successful Search for the Value 17

The code below shows the implementation for the search method. As in the size method, the traversal is initialized to start at the head of the list (line 2). We also use a boolean variable called found to remember whether we have located the item we are searching for. Since we have not found the item at the start of the traversal, found can be set to False (line 3). The iteration in line 4 takes into account both conditions discussed above. As long as there are more nodes to visit and we have not found the item we are looking for, we continue to check the next node. The question in line 5 asks whether the data item is present in the current node. If so, found can be set to True.

```
def search(self,item):
   current = self.head
   found = False
   while current != None and not found:
      if current.get_data() == item:
         found = True
      else:
         current = current.get_next()

   return found
```

As an example, consider invoking the search method looking for the item 17.

```
>>> mylist.search(17)
True
```

Since 17 is in the list, the traversal process needs to move only to the node containing 17. At that point, the variable found is set to True and the while condition will fail, leading to the return value seen above. This process can be seen in Figure 3.27

he remove method requires two logical steps. First, we need to traverse the list looking for the item we want to remove. Once we find the item (recall that we assume it is present), we must remove it. The first step is very similar to search. Starting with an external reference set to the head of the list, we traverse the links until we discover the item we are looking for. Since we assume that item is present, we know that the iteration will stop before current gets to None. This means that we can simply use the boolean found in the condition.

When found becomes True, current will be a reference to the node containing the item to be removed. But how do we remove it? One possibility would be to replace the value of the item with some marker that suggests that the item is no longer present. The problem with this approach is the number of nodes will no longer match the number of items. It would be much better to remove the item by removing the entire node.

In order to remove the node containing the item, we need to modify the link in the previous node so that it refers to the node that comes after current. Unfortunately, there is no way to go backward in the linked list. Since current refers to the node ahead of the node where we would like to make the change, it is too late to make the necessary modification.

The solution to this dilemma is to use two external references as we traverse down the linked list. current will behave just as it did before, marking the current location of the traverse. The new reference, which we will call previous, will always travel one node behind current. That way, when current stops at the node to be removed, previous will be referring to the proper place in the linked list for the modification.

The code below shows the complete remove method. Lines 2–3 assign initial values to the two references. Note that current starts out at the list head as in the other traversal examples. previous, however, is assumed to always travel one node behind current. For this reason, previous starts out with a value of None since there is no node before the head (see Figure 3.28). The boolean variable found will again be used to control the iteration.

In lines 6–7 we ask whether the item stored in the current node is the item we wish to remove. If so, found can be set to True. If we do not find the item, previous and current must both be moved one node ahead. Again, the order of these two statements is crucial. previous must first be moved one node ahead to the location of current. At that point, current can be moved. This process is often referred to as “inch-worming” as previous must catch up to current before current moves ahead. Figure 3.29 shows the movement of previous and current as they progress down the list looking for the node containing the value 17.

```
1  def remove(self, item):
2     current = self.head
3     previous = None
4     found = False
5     while not found:
6        if current.get_data() == item:
7           found = True
8        else:
9           previous = current
10          current = current.get_next()
11
12    if previous == None:
13       self.head = current.get_next()
14    else:
15       previous.set_next(current.get_next())
```

Once the searching step of the remove has been completed, we need to remove the node from the linked list. Figure 3.30 shows the link that must be modified. However, there is a special case that needs to be addressed. If the item to be removed happens to be the first item in the list, then current will reference the first node in the linked list. This also means that previous will be None. We said earlier that previous would be referring to the node whose next reference

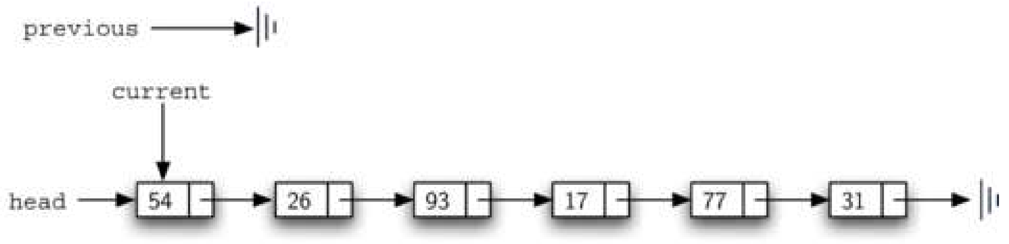


Figure 3.28: Initial Values for the previous and current references

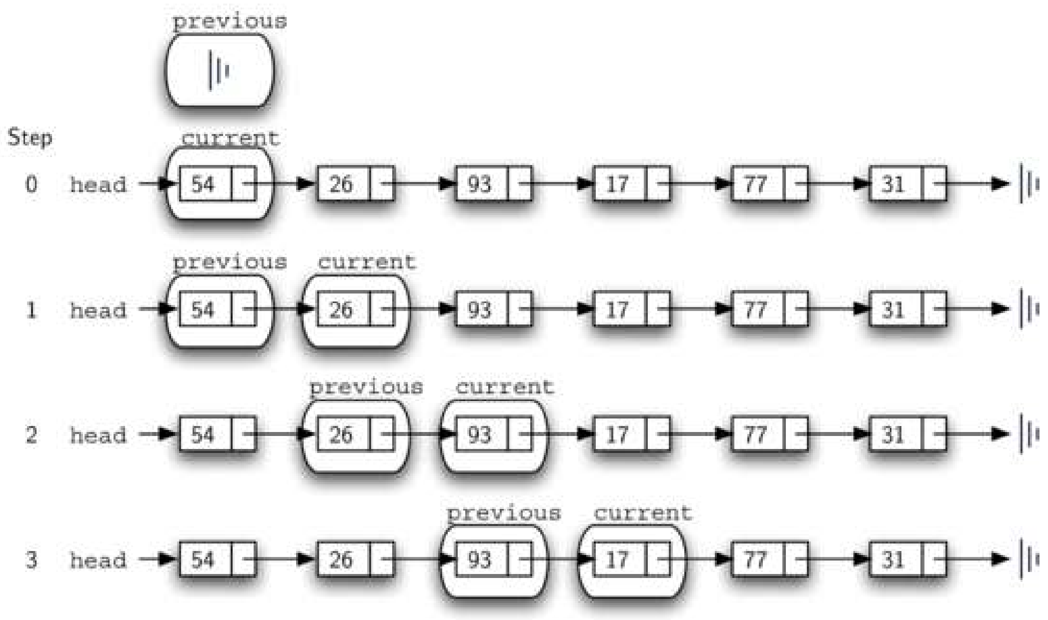


Figure 3.29: previous and current move down the list

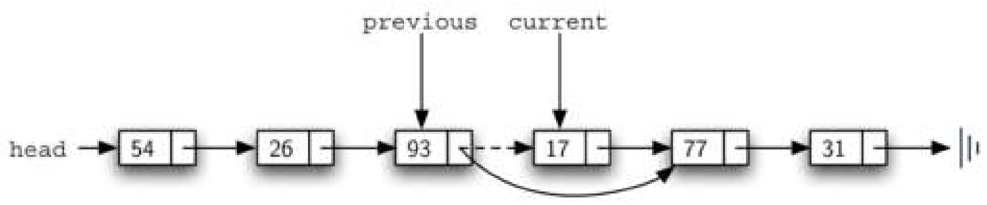


Figure 3.30: Removing an Item from the middle of the list

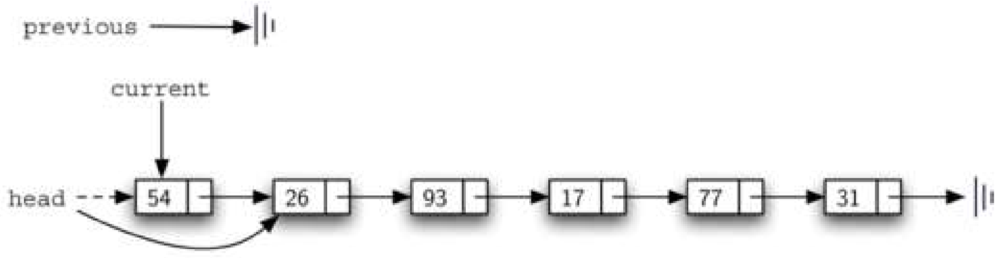


Figure 3.31: Removing the first node from the list

needs to be modified in order to complete the remove. In this case, it is not previous but rather the head of the list that needs to be changed (see Figure 3.31)

Line 12 allows us to check whether we are dealing with the special case described above. If previous did not move, it will still have the value None when the boolean found becomes True. In that case (line 13) the head of the list is modified to refer to the node after the current node, in effect removing the first node from the linked list. However, if previous is not None, the node to be removed is somewhere down the linked list structure. In this case the previous reference is providing us with the node whose next reference must be changed. Line 15 uses the set_next method from previous to accomplish the removal. Note that in both cases the destination of the reference change is current.get_next(). One question that often arises is whether the two cases shown here will also handle the situation where the item to be removed is in the last node of the linked list. We leave that for you to consider.

The remaining methods append, insert, index, and pop are left as exercises. Remember that each of these must take into account whether the change is taking place at the head of the list or someplace else. Also, insert, index, and pop require that we name the positions of the list. We will assume that position names are integers starting with 0.

Self Check

Implement the append method for UnorderedList. What is the time complexity of the method you created? It was most likely $O(n)$. If you add an instance variable to the UnorderedList class you can create an append method that is $O(1)$. Modify your append to be $O(1)$. Be careful! To really do this correctly you will need to consider a couple of special cases that may require you to make a modification to the add method as well.

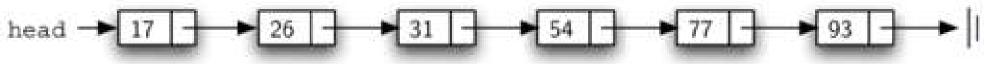


Figure 3.32: An Ordered Linked List

3.10 The Ordered List Abstract Data Type

We will now consider a type of list known as an ordered list. For example, if the list of integers shown above were an ordered list (ascending order), then it could be written as 17, 26, 31, 54, 77, and 93. Since 17 is the smallest item, it occupies the first position in the list. Likewise, since 93 is the largest, it occupies the last position.

The structure of an ordered list is a collection of items where each item holds a relative position that is based upon some underlying characteristic of the item. The ordering is typically either ascending or descending and we assume that list items have a meaningful comparison operation that is already defined. Many of the ordered list operations are the same as those of the unordered list.

- OrderedList() creates a new ordered list that is empty. It needs no parameters and returns an empty list.
- add(item) adds a new item to the list making sure that the order is preserved. It needs the item and returns nothing. Assume the item is not already in the list.
- remove(item) removes the item from the list. It needs the item and modifies the list. Assume the item is present in the list.
- search(item) searches for the item in the list. It needs the item and returns a boolean value.
- is_empty() tests to see whether the list is empty. It needs no parameters and returns a boolean value.
- size() returns the number of items in the list. It needs no parameters and returns an integer.
- index(item) returns the position of item in the list. It needs the item and returns the index. Assume the item is in the list.
- pop() removes and returns the last item in the list. It needs nothing and returns an item. Assume the list has at least one item.
- pop(pos) removes and returns the item at position pos. It needs the position and returns the item. Assume the item is in the list.

3.10.1 Implementing an Ordered List

In order to implement the ordered list, we must remember that the relative positions of the items are based on some underlying characteristic. The ordered list of integers given above (17, 26, 31, 54, 77, and 93) can be represented by a linked structure as shown in Figure 3.32. Again, the node and link structure is ideal for representing the relative positioning of the items.

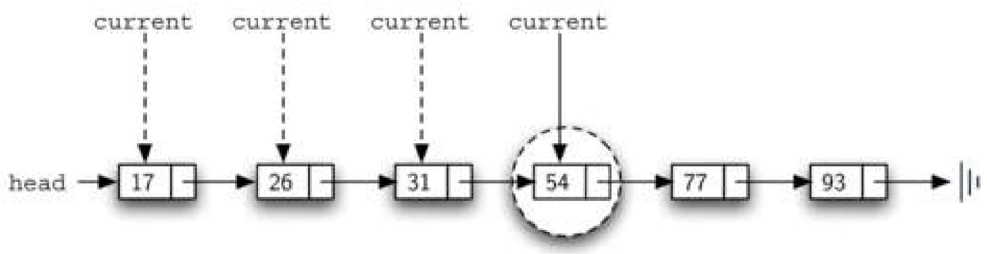


Figure 3.33: Searching an Ordered Linked List

To implement the OrderedList class, we will use the same technique as seen previously with unordered lists. Once again, an empty list will be denoted by a head reference to None.

```
class OrderedList:
   def __init__(self):
      self.head = None
```

As we consider the operations for the ordered list, we should note that the is_empty and size methods can be implemented the same as with unordered lists since they deal only with the number of nodes in the list without regard to the actual item values. Likewise, the remove method will work just fine since we still need to find the item and then link around the node to remove it. The two remaining methods, search and add, will require some modification.

The search of an unordered linked list required that we traverse the nodes one at a time until we either find the item we are looking for or run out of nodes (None). It turns out that the same approach would actually work with the ordered list and in fact in the case where we find the item it is exactly what we need. However, in the case where the item is not in the list, we can take advantage of the ordering to stop the search as soon as possible.

For example, Figure 3.33 shows the ordered linked list as a search is looking for the value 45. As we traverse, starting at the head of the list, we first compare against 17. Since 17 is not the item we are looking for, we move to the next node, in this case 26. Again, this is not what we want, so we move on to 31 and then on to 54. Now, at this point, something is different. Since 54 is not the item we are looking for, our former strategy would be to move forward. However, due to the fact that this is an ordered list, that will not be necessary. Once the value in the node becomes greater than the item we are searching for, the search can stop and return False. There is no way the item could exist further out in the linked list.

The following code shows the complete search method. It is easy to incorporate the new condition discussed above by adding another boolean variable, stop, and initializing it to False (line 4). While stop is False (not stop) we can continue to look forward in the list (line 5). If any node is ever discovered that contains data greater than the item we are looking for, we will set stop to True (lines $9 - -10$). The remaining lines are identical to the unordered list search.

```
1 def search(self, item):
2    current = self.head
3    found = False
4    stop = False
5    while current != None and not found and not stop:
```

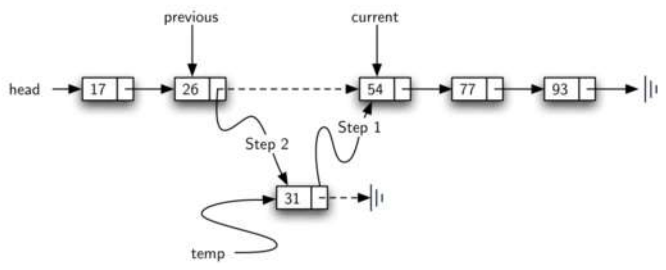


Figure 3.34: Adding an Item to an Ordered Linked List

```
      if current.get_data() == item:
         found = True
      else:
         if current.get_data() > item:
            stop = True
         else:
            current = current.get_next()

   return found
```

The most significant method modification will take place in add. Recall that for unordered lists, the add method could simply place a new node at the head of the list. It was the easiest point of access. Unfortunately, this will no longer work with ordered lists. It is now necessary that we discover the specific place where a new item belongs in the existing ordered list.

Assume we have the ordered list consisting of 17, 26, 54, 77, and 93 and we want to add the value 31. The add method must decide that the new item belongs between 26 and 54. Figure 3.34 shows the setup that we need. As we explained earlier, we need to traverse the linked list looking for the place where the new node will be added. We know we have found that place when either we run out of nodes (current becomes None) or the value of the current node becomes greater than the item we wish to add. In our example, seeing the value 54 causes us to stop.

As we saw with unordered lists, it is necessary to have an additional reference, again called previous, since current will not provide access to the node that must be modified. The below code shows the complete add method. Lines $2 - -3$ set up the two external references and lines $9 - -10$ again allow previous to follow one node behind current every time through the iteration. The condition (line 5) allows the iteration to continue as long as there are more nodes and the value in the current node is not larger than the item. In either case, when the iteration fails, we have found the location for the new node.

The remainder of the method completes the two-step process shown in Figure 3.34. Once a new node has been created for the item, the only remaining question is whether the new node will be added at the beginning of the linked list or some place in the middle. Again, previous == None (line 13) can be used to provide the answer.

```
1  def add(self, item):
2     current = self.head
3     previous = None
4     stop = False
```

```
   while current != None and not stop:
      if current.get_data() > item:
         stop = True
      else:
         previous = current
         current = current.get_next()

   temp = Node(item)
   if previous == None:
      temp.set_next(self.head)
      self.head = temp
   else:
      temp.set_next(current)
      previous.set_next(temp)
```

3.10.2 Analysis of Linked Lists

To analyze the complexity of the linked list operations, we need to consider whether they require traversal. Consider a linked list that has n nodes. The is_empty method is $O(1)$ since it requires one step to check the head reference for None. size, on the other hand, will always require n steps since there is no way to know how many nodes are in the linked list without traversing from head to end. Therefore, length is $O(n)$. Adding an item to an unordered list will always be $O(1)$ since we simply place the new node at the head of the linked list. However, search and remove, as well as add for an ordered list, all require the traversal process. Although on average they may need to traverse only half of the nodes, these methods are all $O(n)$ since in the worst case each will process every node in the list.

You may also have noticed that the performance of this implementation differs from the actual performance given earlier for Python lists. This suggests that linked lists are not the way Python lists are implemented. The actual implementation of a Python list is based on the notion of an array. We discuss this in more detail in another chapter.

3.11 Summary

- Linear data structures maintain their data in an ordered fashion.
- Stacks are simple data structures that maintain a LIFO, last-in first-out, ordering.
- The fundamental operations for a stack are push, pop, and is_empty.
- Queues are simple data structures that maintain a FIFO, first-in first-out, ordering.
- The fundamental operations for a queue are enqueue, dequeue, and is_empty.
- Prefix, infix, and postfix are all ways to write expressions.
- Stacks are very useful for designing algorithms to evaluate and translate expressions.
- Stacks can provide a reversal characteristic.

- Queues can assist in the construction of timing simulations.
- Simulations use random number generators to create a real-life situation and allow us to answer "what if" types of questions.
- Deques are data structures that allow hybrid behavior like that of stacks and queues.
- The fundamental operations for a deque are add_front, add_rear, remove_front, remove_rear, and is_empty.
- Lists are collections of items where each item holds a relative position.
- A linked list implementation maintains logical order without requiring physical storage requirements.
- Modification to the head of the linked list is a special case.

3.12 Key Terms

balanced parentheses	data field	deque
first-in first-out (FIFO)	fully parenthesized	head
infix	last-in first-out (LIFO)	linear data structure
linked list	linked list traversal	list
node	palindrome	postfix
precedence	prefix	queue
simulation	stack	

3.13 Discussion Questions

1. Convert the following values to binary using "divide by 2." Show the stack of remainders.
    - 17
    - 45
    - 96
2. Convert the following infix expressions to prefix (use full parentheses):
    - $(A + B) * (C + D) * (E + F)$
    - $A + ((B + C) * (D + E))$
    - $A * B * C * D + E + F$
3. Convert the above infix expressions to postfix (use full parentheses).
4. Convert the above infix expressions to postfix using the direct conversion algorithm. Show the stack as the conversion takes place.
5. Evaluate the following postfix expressions. Show the stack as each operand and operator is processed.

- $23 * 4+$
- $12 + 3 + 4 + 5+$
- $12345 * + * +$

6. The alternative implementation of the Queue ADT is to use a list such that the rear of the queue is at the end of the list. What would this mean for Big-O performance?
7. What is the result of carrying out both steps of the linked list add method in reverse order? What kind of reference results? What types of problems may result?
8. Explain how the linked list remove method works when the item to be removed is in the last node.
9. Explain how the remove method works when the item is in the only node in the linked list.

3.14 Programming Exercises

1. Modify the infix-to-postfix algorithm so that it can handle errors.
2. Modify the postfix evaluation algorithm so that it can handle errors.
3. Implement a direct infix evaluator that combines the functionality of infix-to-postfix conversion and the postfix evaluation algorithm. Your evaluator should process infix tokens from left to right and use two stacks, one for operators and one for operands, to perform the evaluation.
4. Turn your direct infix evaluator from the previous problem into a calculator.
5. Implement the Queue ADT, using a list such that the rear of the queue is at the end of the list.
6. Design and implement an experiment to do benchmark comparisons of the two queue implementations. What can you learn from such an experiment?
7. It is possible to implement a queue such that both enqueue and dequeue have $O(1)$ performance on average. In this case it means that most of the time enqueue and dequeue will be $O(1)$ except in one particular circumstance where dequeue will be $O(n)$.
8. Consider a real life situation. Formulate a question and then design a simulation that can help to answer it. Possible situations include:
   - Cars lined up at a car wash
   - Customers at a grocery store check-out
   - Airplanes taking off and landing on a runway
   - A bank teller

   Be sure to state any assumptions that you make and provide any probabilistic data that must be considered as part of the scenario.

9. Modify the Hot Potato simulation to allow for a randomly chosen counting value so that each pass is not predictable from the previous one.

10. Implement a radix sorting machine. A radix sort for base 10 integers is a mechanical sorting technique that utilizes a collection of bins, one main bin and 10 digit bins. Each bin acts like a queue and maintains its values in the order that they arrive. The algorithm begins by placing each number in the main bin. Then it considers each value digit by digit. The first value is removed and placed in a digit bin corresponding to the digit being considered. For example, if the ones digit is being considered, 534 is placed in digit bin 4 and 667 is placed in digit bin 7. Once all the values are placed in the corresponding digit bins, the values are collected from bin 0 to bin 9 and placed back in the main bin. The process continues with the tens digit, the hundreds, and so on. After the last digit is processed, the main bin contains the values in order.

11. Another example of the parentheses matching problem comes from hypertext markup language (HTML). In HTML, tags exist in both opening and closing forms and must be balanced to properly describe a web document. This very simple HTML document:

    ```
    <html>
      <head>
        <title>
          Example
        </title>
      </head>

      <body>
        <h1>Hello, world</h1>
      </body>
    </html>
    ```

    is intended only to show the matching and nesting structure for tags in the language. Write a program that can check an HTML document for proper opening and closing tags.

12. To implement the length method, we counted the number of nodes in the list. An alternative strategy would be to store the number of nodes in the list as an additional piece of data in the head of the list. Modify the UnorderedList class to include this information and rewrite the length method.

13. Implement the remove method so that it works correctly in the case where the item is not in the list.

14. Modify the list classes to allow duplicates. Which methods will be impacted by this change?

15. Implement the __str__ method in the UnorderedList class. What would be a good string representation for a list?

16. Implement __str__ method so that lists are displayed the Python way (with square brackets).

17. Implement the remaining operations defined in the UnorderedList ADT (append, index, pop, insert).
18. Implement a slice method for the UnorderedList class. It should take two parameters, start and stop, and return a copy of the list starting at the start position and going up to but not including the stop position.
19. Implement the remaining operations defined in the OrderedList ADT.
20. Implement a stack using linked lists.
21. Implement a queue using linked lists.
22. Implement a deque using linked lists.
23. Design and implement an experiment that will compare the performance of a Python list with a list implemented as a linked list.
24. Design and implement an experiment that will compare the performance of the Python list based stack and queue with the linked list implementation.
25. The linked list implementation given above is called a singly linked list because each node has a single reference to the next node in sequence. An alternative implementation is known as a doubly linked list. In this implementation, each node has a reference to the next node (commonly called next) as well as a reference to the preceding node (commonly called back). The head reference also contains two references, one to the first node in the linked list and one to the last. Code this implementation in Python.
26. Create an implementation of a queue that would have an average performance of $O(1)$ for enqueue and dequeue operations.

CHAPTER FOUR

RECURSION

4.1 Objectives

The goals for this chapter are as follows:

- To understand that complex problems that may otherwise be difficult to solve may have a simple recursive solution.
- To learn how to formulate programs recursively.
- To understand and apply the three laws of recursion.
- To understand recursion as a form of iteration.
- To implement the recursive formulation of a problem.
- To understand how recursion is implemented by a computer system.

4.2 What is Recursion?

Recursion is a method of solving problems that involves breaking a problem down into smaller and smaller subproblems until you get to a small enough problem that it can be solved trivially. Usually recursion involves a function calling itself. While it may not seem like much on the surface, recursion allows us to write elegant solutions to problems that may otherwise be very difficult to program.

4.2.1 Calculating the Sum of a List of Numbers

We will begin our investigation with a simple problem that you already know how to solve without using recursion. Suppose that you want to calculate the sum of a list of numbers such as: $[1, 3, 5, 7, 9]$. An iterative function that computes the sum is shown below. The function uses an accumulator variable (the_sum) to compute a running total of all the numbers in the list by starting with 0 and adding each number in the list.

```python
def list_sum(num_list):
    the_sum = 0
    for i in num_list:
        the_sum = the_sum + i
    return the_sum

print(list_sum([1,3,5,7,9]))
```

Pretend for a minute that you do not have while loops or for loops. How would you compute the sum of a list of numbers? If you were a mathematician you might start by recalling that addition is a function that is defined for two parameters, a pair of numbers. To redefine the problem from adding a list to adding pairs of numbers, we could rewrite the list as a fully parenthesized expression. Such an expression looks like this:

$$((((1 + 3) + 5) + 7) + 9)$$

We can also parenthesize the expression the other way around,

$$(1 + (3 + (5 + (7 + 9))))$$

Notice that the innermost set of parentheses, $(7 + 9)$, is a problem that we can solve without a loop or any special constructs. In fact, we can use the following sequence of simplifications to compute a final sum.

$$\text{total} = (1 + (3 + (5 + (7 + 9))))$$
$$\text{total} = (1 + (3 + (5 + 16)))$$
$$\text{total} = (1 + (3 + 21))$$
$$\text{total} = (1 + 24)$$
$$\text{total} = 25$$

How can we take this idea and turn it into a Python program? First, let's restate the the sum problem in terms of Python lists. We might say the the sum of the list num_list is the sum of the first element of the list (num_list[0]), and the sum of the numbers in the rest of the list (num_list[1 :]). To state it in a functional form: $\text{list\_sum}(\text{num\_list}) = \text{first}(\text{num\_list}) + \text{list\_sum}(\text{rest}(\text{num\_list}))$ In this equation $\text{first}(\text{num\_list})$ returns the first element of the list and $\text{rest}(\text{num\_list})$ returns a list of everything but the first element. This is easily expressed in Python as the following:

```
def list_sum(num_list):
   if len(num_list) == 1:
        return num_list[0]
   else:
        return num_list[0] + list_sum(num_list[1:])

print(list_sum([1,3,5,7,9]))
```

There are a few key ideas in this to look at. First, on line 2 we are checking to see if the list is one element long. This check is crucial and is our escape clause from the function. The sum of a list of length 1 is trivial; it is just the number in the list. Second, on line 5 our function calls

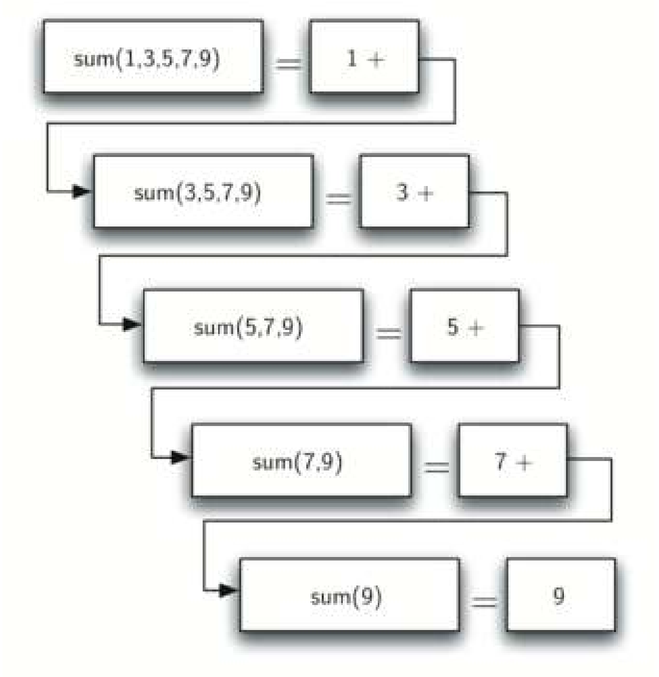


Figure 4.1: Series of Recursive Calls Adding a List of Numbers

itself! This is the reason that we call the list_sum algorithm recursive. A recursive function is a function that calls itself.

Figure 4.1 shows the series of recursive calls that are needed to sum the list $[1, 3, 5, 7, 9]$. You should think of this series of calls as a series of simplifications. Each time we make a recursive call we are solving a smaller problem, until we reach the point where the problem cannot get any smaller.

When we reach the point where the problem is as simple as it can get, we begin to piece together the solutions of each of the small problems until the initial problem is solved. Figure 4.2 shows the additions that are performed as list_sum works its way backward through the series of calls. When list_sum returns from the topmost problem, we have the solution to the whole problem.

4.2.2 The Three Laws of Recursion

Like the robots of Asimov, all recursive algorithms must obey three important laws:

1. A recursive algorithm must have a base case.
2. A recursive algorithm must change its state and move toward the base case.
3. A recursive algorithm must call itself, recursively.

Let's look at each one of these laws in more detail and see how it was used in the list_sum algorithm. First, a base case is the condition that allows the algorithm to stop recursing. A base case is typically a problem that is small enough to solve directly. In the list_sum algorithm the base case is a list of length 1.

To obey the second law, we must arrange for a change of state that moves the algorithm toward the base case. A change of state means that some data that the algorithm is using is modified.

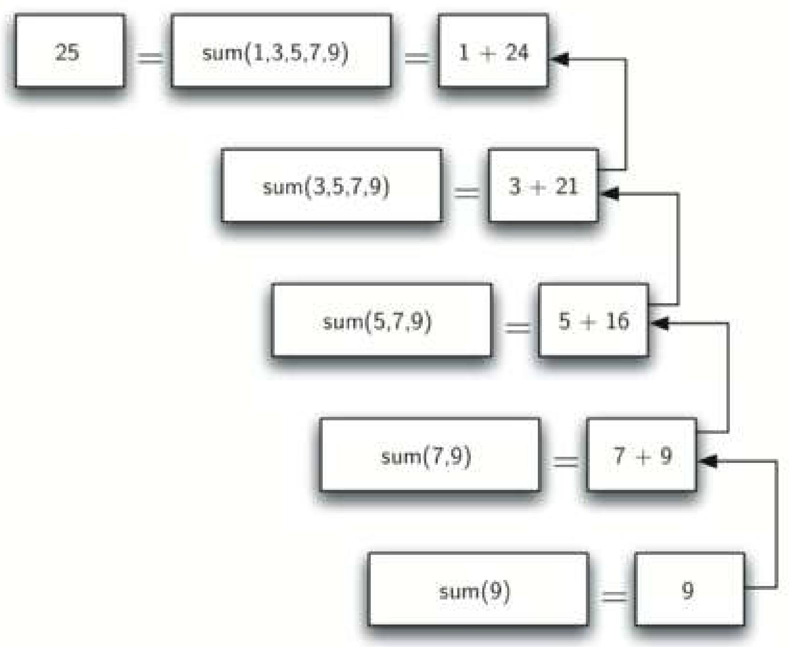


Figure 4.2: Series of Recursive Returns from Adding a List of Numbers

Usually the data that represents our problem gets smaller in some way. In the list_sum algorithm our primary data structure is a list, so we must focus our state-changing efforts on the list. Since the base case is a list of length 1, a natural progression toward the base case is to shorten the list. This is exactly what happens on line 5 of the code below when we call list_sum with a shorter list.

The final law is that the algorithm must call itself. This is the very definition of recursion. Recursion is a confusing concept to many beginning programmers. As a novice programmer, you have learned that functions are good because you can take a large problem and break it up into smaller problems. The smaller problems can be solved by writing a function to solve each problem. When we talk about recursion it may seem that we are talking ourselves in circles. We have a problem to solve with a function, but that function solves the problem by calling itself! But the logic is not circular at all; the logic of recursion is an elegant expression of solving a problem by breaking it down into a smaller and easier problems.

In the remainder of this chapter we will look at more examples of recursion. In each case we will focus on designing a solution to a problem by using the three laws of recursion.

Self Check

How many recursive calls are made when computing the sum of the list $[2, 4, 6, 8, 10]$?

1. 6
2. 5
3. 4
4. 3

Suppose you are going to write a recursive function to calculate the factorial of a number. fact(n) returns $n * n - 1 * n - 2 * \ldots$ Where the factorial of zero is defined to be 1. What would be the most appropriate base case?

1. $n == 0$
2. $n == 1$
3. $n >= 0$
4. $n <= 1$

4.2.3 Converting an Integer to a String in Any Base

Suppose you want to convert an integer to a string in some base between binary and hexadecimal. For example, convert the integer 10 to its string representation in decimal as "10," or to its string representation in binary as "1010." While there are many algorithms to solve this problem, including the algorithm discussed in the stack section, the recursive formulation of the problem is very elegant.

Let's look at a concrete example using base 10 and the number 769. Suppose we have a sequence of characters corresponding to the first 10 digits, like conv_string = "0123456789". It is easy to convert a number less than 10 to its string equivalent by looking it up in the sequence. For example, if the number is 9, then the string is conv_string[9] or "9." If we can arrange to break up the number 769 into three single-digit numbers, 7, 6, and 9, then converting it to a string is simple. A number less than 10 sounds like a good base case.

Knowing what our base is suggests that the overall algorithm will involve three components:

1. Reduce the original number to a series of single-digit numbers.
2. Convert the single digit-number to a string using a lookup.
3. Concatenate the single-digit strings together to form the final result.

The next step is to figure out how to change state and make progress toward the base case. Since we are working with an integer, let's consider what mathematical operations might reduce a number. The most likely candidates are division and subtraction. While subtraction might work, it is unclear what we should subtract from what. Integer division with remainders gives us a clear direction. Let's look at what happens if we divide a number by the base we are trying to convert to.

Using integer division to divide 769 by 10, we get 76 with a remainder of 9. This gives us two good results. First, the remainder is a number less than our base that can be converted to a string immediately by lookup. Second, we get a number that is smaller than our original and moves us toward the base case of having a single number less than our base. Now our job is to convert 76 to its string representation. Again we will use integer division plus remainder to get results of 7 and 6 respectively. Finally, we have reduced the problem to converting 7, which we can do easily since it satisfies the base case condition of $n <$base, where base$= 10$. The series of operations we have just performed is illustrated in Figure 4.3. Notice that the numbers we want to remember are in the remainder boxes along the right side of the diagram.

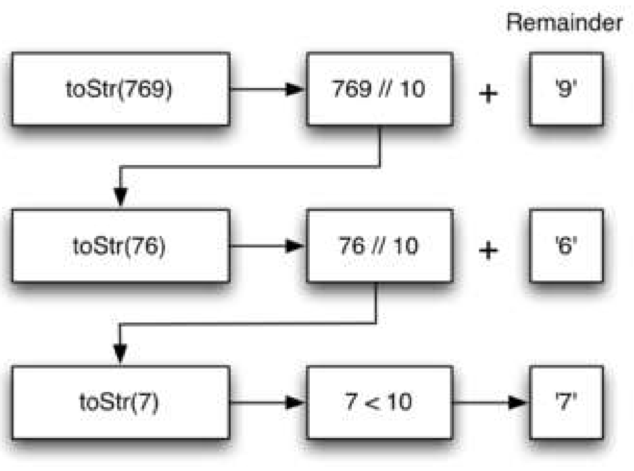


Figure 4.3: Converting an Integer to a String in Base 10

The code below shows the Python code that implements the algorithm outlined above for any base between 2 and 16.

```python
1 def to_str(n, base):
2   convert_string = "0123456789ABCDEF"
3   if n < base:
4     return convert_string[n]
5   else:
6     return to_str(n / base, base) + convert_string[n % base]
7
8 print(to_str(1453, 16))
```

Notice that in line 3 we check for the base case where n is less than the base we are converting to. When we detect the base case, we stop recursing and simply return the string from the convertString sequence. In line 6 we satisfy both the second and third laws – by making the recursive call and by reducing the problem size – using division.

Let us trace the algorithm again; this time we will convert the number 10 to its base 2 string representation ("1010").

Figure 4.4 shows that we get the results we are looking for, but it looks like the digits are in the wrong order. The algorithm works correctly because we make the recursive call first on line 6, then we add the string representation of the remainder. If we reversed returning the convertString lookup and returning the toStr call, the resulting string would be backward! But by delaying the concatenation operation until after the recursive call has returned, we get the result in the proper order. This should remind you of our discussion of stacks back in the previous chapter.

Self Check

Write a function that takes a string as a parameter and returns a new string that is the reverse of the old string.

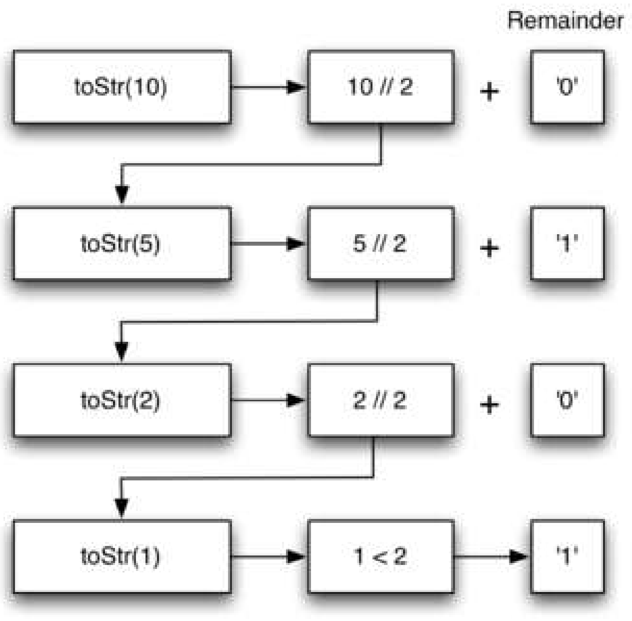


Figure 4.4: Converting an Integer to a String in Base 10

Write a function that takes a string as a parameter and returns True if the string is a palindrome, False otherwise. Remember that a string is a palindrome if it is spelled the same both forward and backward. for example: radar is a palindrome. for bonus points palindromes can also be phrases, but you need to remove the spaces and punctuation before checking. for example: madam i'm adam is a palindrome. Other fun palindromes include:

- kayak
- aibohphobia
- Live not on evil
- Reviled did I live, said I, as evil I did deliver
- Go hang a salami; I'm a lasagna hog.
- Able was I ere I saw Elba
- Kanakanak – a town in Alaska
- Wassamassaw – a town in South Dakota

4.3 Stack Frames: Implementing Recursion

Suppose that instead of concatenating the result of the recursive call to toStr with the string from convertString, we modified our algorithm to push the strings onto a stack prior to making the recursive call. The code for this modified algorithm is shown in the code below.

```
import Stack  # As previously defined
```

```
r_stack = Stack()

def to_str(n, base):
   convert_string = "0123456789ABCDEF"
   while n > 0:
      if n < base:
         r_stack.push(convert_string[n])
      else:
         r_stack.push(convert_string[n % base])
      n = n // base
   res = ""
   while not r_stack.is_empty():
      res = res + str(r_stack.pop())
   return res

print(to_str(1453, 16))
```

Each time we make a call to toStr, we push a character on the stack. Returning to the previous example we can see that after the fourth call to toStr the stack would look like Figure 4.5. Notice that now we can simply pop the characters off the stack and concatenate them into the final result, “1010.”

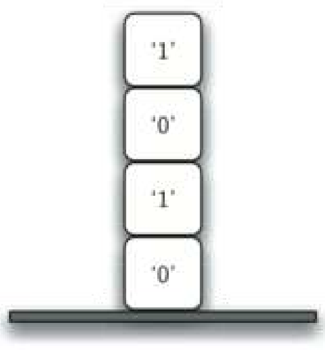


Figure 4.5: Strings Placed on the Stack During Conversion

The previous example gives us some insight into how Python implements a recursive function call. When a function is called in Python, a **stack frame** is allocated to handle the local variables of the function. When the function returns, the return value is left on top of the stack for the calling function to access. Figure 4.6 illustrates the call stack after the return statement on line 4.

Notice that the call to toStr$(2//2, 2)$ leaves a return value of “” on the stack. This return value is then used in place of the function call (toStr$(1, 2)$) in the expression “1” $+$ convertString$[2\%2]$, which will leave the string “10” on the top of the stack. In this way, the Python call stack takes the place of the stack we used explicitly earlier. In our list summing example, you can think of the return value on the stack taking the place of an accumulator variable.

The stack frames also provide a scope for the variables used by the function. Even though we are calling the same function over and over, each call creates a new scope for the variables that are local to the function.

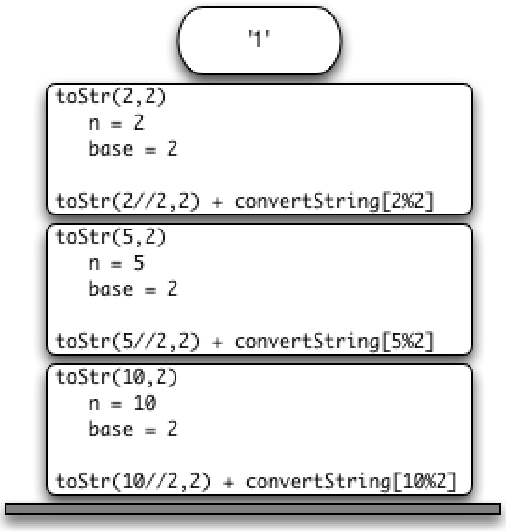

'1'
toStr(2,2)
n = 2
base = 2
toStr(2//2,2) + convertString[2%2]
toStr(5,2)
n = 5
base = 2
toStr(5//2,2) + convertString[5%2]
toStr(10,2)
n = 10
base = 2
toStr(10//2,2) + convertString[10%2]

Figure 4.6: Call Stack Generated from `toStr(10,2)`

If you keep this idea of the stack in your head, you will find it much easier to write a proper recursive function.

4.4 Visualising Recursion

In the previous section we looked at some problems that were easy to solve using recursion; however, it can still be difficult to find a mental model or a way of visualizing what is happening in a recursive function. This can make recursion difficult for people to grasp. In this section we will look at a couple of examples of using recursion to draw some interesting pictures. As you watch these pictures take shape you will get some new insight into the recursive process that may be helpful in cementing your understanding of recursion.

The tool we will use for our illustrations is Python's turtle graphics module called turtle. The turtle module is standard with all versions of Python and is very easy to use. The metaphor is quite simple. You can create a turtle and the turtle can move forward, backward, turn left, turn right, etc. The turtle can have its tail up or down. When the turtle's tail is down and the turtle moves it draws a line as it moves. To increase the artistic value of the turtle you can change the width of the tail as well as the color of the ink the tail is dipped in.

Here is a simple example to illustrate some turtle graphics basics. We will use the turtle module to draw a spiral recursively.The code below shows how it is done. After importing the turtle module we create a turtle. When the turtle is created it also creates a window for itself to draw in. Next we define the drawSpiral function. The base case for this simple function is when the length of the line we want to draw, as given by the len parameter, is reduced to zero or less. If the length of the line is longer than zero we instruct the turtle to go forward by len units and then turn right 90 degrees. The recursive step is when we call drawSpiral again with a reduced length. At the end of the code below you will notice that we call the function my_win.exitonclick(), this is a handy little method of the window that puts the turtle into a wait mode until you click inside the window, after which the program cleans up and exits.

```python
import turtle

my_turtle = turtle.Turtle()
my_win = turtle.Screen()

def draw_spiral(my_turtle, line_len):
   if lineLen > 0:
      my_turtle.forward(line_len)
      my_turtle.right(90)
      draw_spiral(my_turtle, line_len - 5)

draw_spiral(my_turtle, 100)
my_win.exitonclick()
```

That is really about all the turtle graphics you need to know in order to make some pretty impressive drawings. For our next program we are going to draw a fractal tree. Fractals come from a branch of mathematics, and have much in common with recursion. The definition of a fractal is that when you look at it the fractal has the same basic shape no matter how much you magnify it. Some examples from nature are the coastlines of continents, snowflakes, mountains, and even trees or shrubs. The fractal nature of many of these natural phenomenon makes it possible for programmers to generate very realistic looking scenery for computer generated movies. In our next example we will generate a fractal tree.

To understand how this is going to work it is helpful to think of how we might describe a tree using a fractal vocabulary. Remember that we said above that a fractal is something that looks the same at all different levels of magnification. If we translate this to trees and shrubs we might say that even a small twig has the same shape and characteristics as a whole tree. Using this idea we could say that a *tree* is a trunk, with a smaller *tree* going off to the right and another smaller *tree* going off to the left. If you think of this definition recursively it means that we will apply the recursive definition of a tree to both of the smaller left and right trees.

Lets translate this idea to some Python code. The code below shows how we can use our turtle to generate a fractal tree. Lets look at the code a bit more closely. You will see that on lines 5 and 7 we are making a recursive call. On line 5 we make the recursive call right after the turtle turns to the right by 20 degrees; this is the right tree mentioned above. Then in line 7 the turtle makes another recursive call, but this time after turning left by 40 degrees. The reason the turtle must turn left by 40 degrees is that it needs to undo the original 20 degree turn to the right and then do an additional 20 degree turn to the left in order to draw the left tree. Also notice that each time we make a recursive call to tree we subtract some amount from the branchLen parameter; this is to make sure that the recursive trees get smaller and smaller. You should also recognize the initial if statement on line 2 as a check for the base case of branchLen getting too small.

```python
def tree(branch_len, t):
   if branch_len > 5:
      t.forward(branch_len)
      t.right(20)
      tree(branch_len - 15, t)
```

```
        t.left(40)
        tree(branch_len - 10,t)
        t.right(20)
        t.backward(branch_len)
```

The complete program for this tree example is shown below. Before you run the code think about how you expect to see the tree take shape. Look at the recursive calls and think about how this tree will unfold. Will it be drawn symmetrically with the right and left halves of the tree taking shape simultaneously? Will it be drawn right side first then left side?

```
import turtle

def tree(branch_len, t):
    if branch_len > 5:
        t.forward(branch_len)
        t.right(20)
        tree(branch_len - 15, t)
        t.left(40)
        tree(branch_len - 15, t)
        t.right(20)
        t.backward(branch_len)

def main():
    t = turtle.Turtle()
    my_win = turtle.Screen()
    t.left(90)
    t.up()
    t.backward(100)
    t.down()
    t.color("green")
    tree(75, t)
    my_win.exitonclick()

main()
```

Notice how each branch point on the tree corresponds to a recursive call, and notice how the tree is drawn to the right all the way down to its shortest twig. You can see this in Figure 4.7. Now, notice how the program works its way back up the trunk until the entire right side of the tree is drawn. You can see the right half of the tree in Figure 4.8. Then the left side of the tree is drawn, but not by going as far out to the left as possible. Rather, once again the entire right side of the left tree is drawn until we finally make our way out to the smallest twig on the left.

This simple tree program is just a starting point for you, and you will notice that the tree does not look particularly realistic because nature is just not as symmetric as a computer program. The exercises at the end of the chapter will give you some ideas for how to explore some interesting options to make your tree look more realistic.

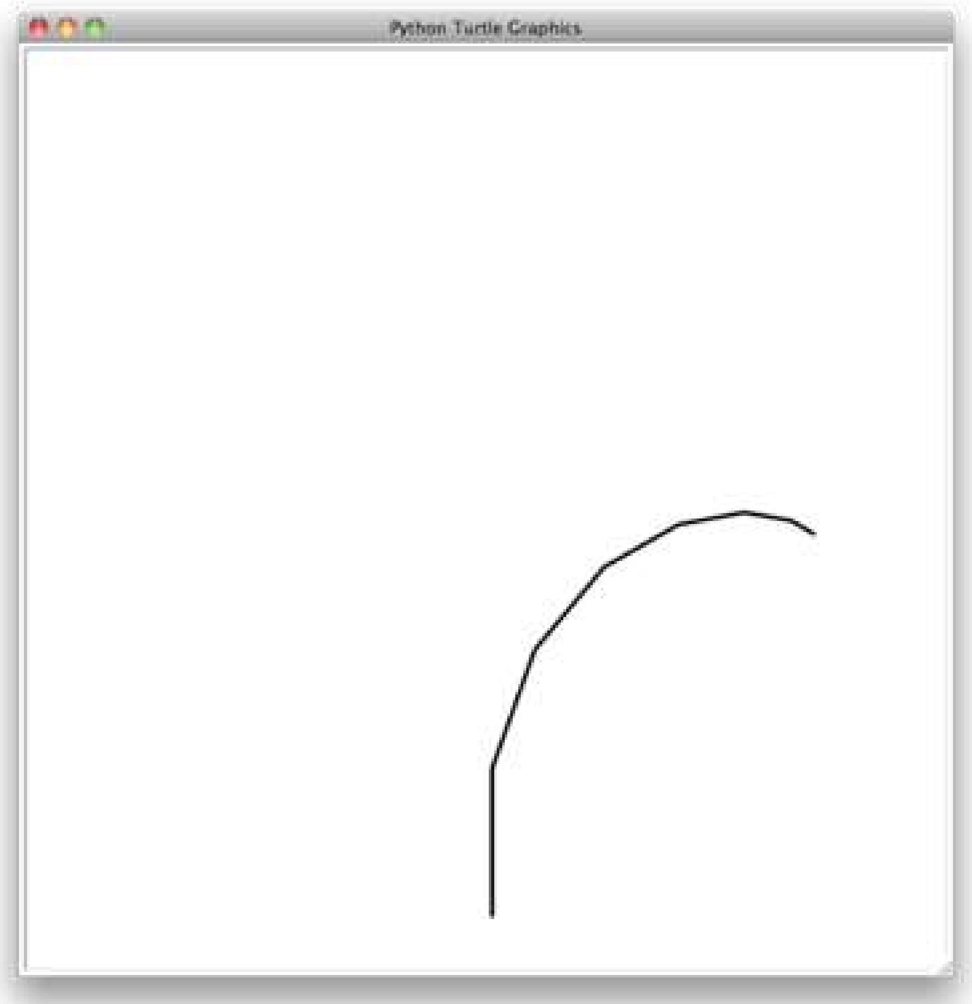


Figure 4.7: The Beginning of a Fractal Tree

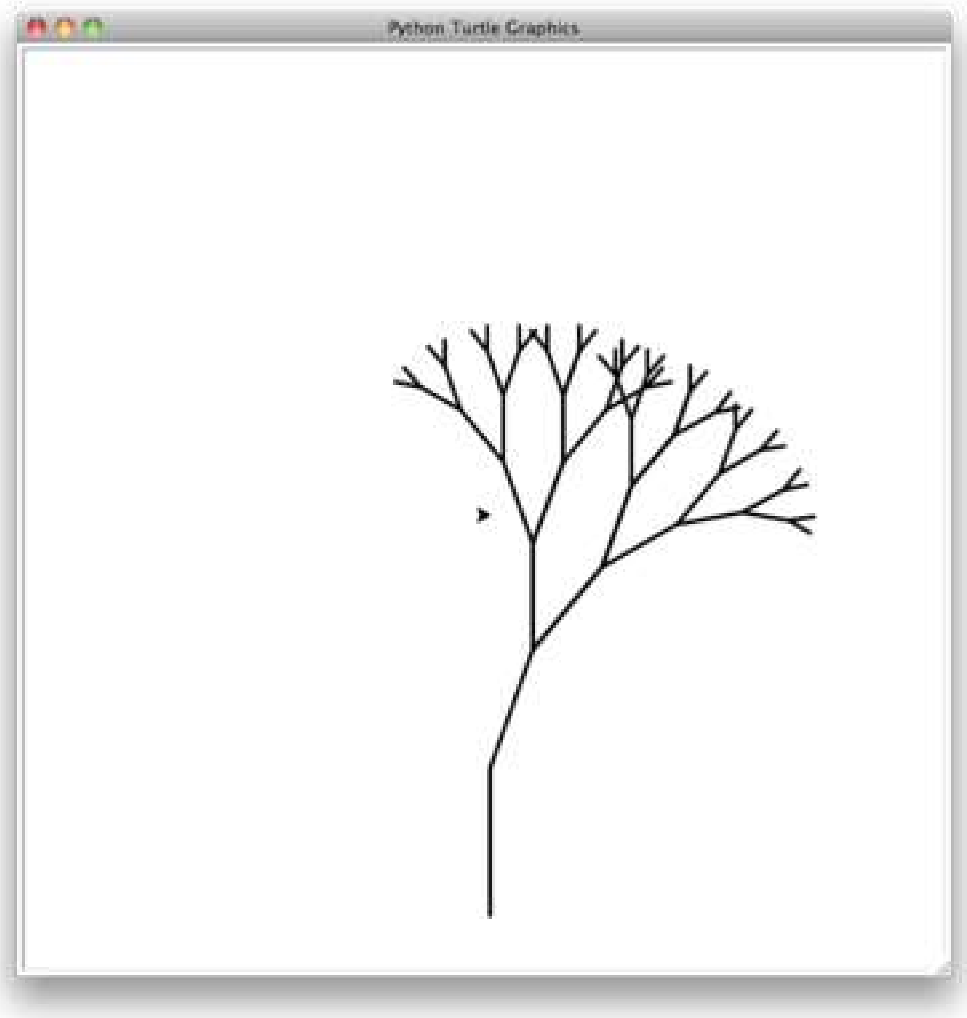


Figure 4.8: The First Half of the Tree

Self Check

Modify the recursive tree program using one or all of the following ideas:

- Modify the thickness of the branches so that as the branchLen gets smaller, the line gets thinner.
- Modify the color of the branches so that as the branchLen gets very short it is colored like a leaf.
- Modify the angle used in turning the turtle so that at each branch point the angle is selected at random in some range. For example choose the angle between 15 and 45 degrees. Play around to see what looks good.
- Modify the branchLen recursively so that instead of always subtracting the same amount you subtract a random amount in some range.

4.4.1 Sierpinski Triangle

Another fractal that exhibits the property of self-similarity is the Sierpinski triangle. An example is shown in Figure 4.9. The Sierpinski triangle illustrates a three-way recursive algorithm. The procedure for drawing a Sierpinski triangle by hand is simple. Start with a single large triangle. Divide this large triangle into four new triangles by connecting the midpoint of each side. Ignoring the middle triangle that you just created, apply the same procedure to each of the three corner triangles. Each time you create a new set of triangles, you recursively apply this procedure to the three smaller corner triangles. You can continue to apply this procedure indefinitely if you have a sharp enough pencil. Before you continue reading, you may want to try drawing the Sierpinski triangle yourself, using the method described.

Since we can continue to apply the algorithm indefinitely, what is the base case? We will see that the base case is set arbitrarily as the number of times we want to divide the triangle into pieces. Sometimes we call this number the “degree” of the fractal. Each time we make a recursive call, we subtract 1 from the degree until we reach 0. When we reach a degree of 0, we stop making recursive calls. The code that generated the Sierpinski Triangle is shown below.

```python
import turtle

def draw_triangle(points, color, my_turtle):
    my_turtle.fillcolor(color)
    my_turtle.up()
    my_turtle.goto(points[0][0],points[0][1])
    my_turtle.down()
    my_turtle.begin_fill()
    my_turtle.goto(points[1][0], points[1][1])
    my_turtle.goto(points[2][0], points[2][1])
    my_turtle.goto(points[0][0], points[0][1])
    my_turtle.end_fill()

def get_mid(p1, p2):
    return ((p1[0] + p2[0]) / 2, (p1[1] + p2[1]) / 2)
```

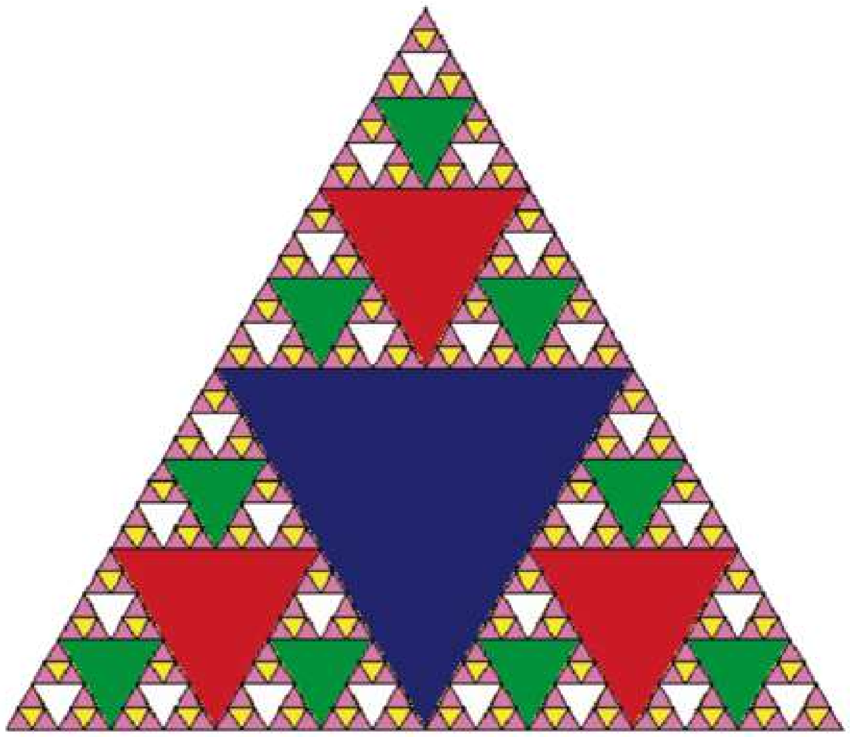


Figure 4.9: The Sierpinski Triangle

```
def sierpinski(points, degree, my_turtle):
    color_map = ['blue', 'red', 'green', 'white', 'yellow',
                 'violet', 'orange']
    draw_triangle(points, color_map[degree], my_turtle)
    if degree > 0:
        sierpinski([points[0],
                    get_mid(points[0], points[1]),
                    get_mid(points[0], points[2])],
                degree-1, my_turtle)
        sierpinski([points[1],
                    get_mid(points[0], points[1]),
                    get_mid(points[1], points[2])],
                degree-1, my_turtle)
        sierpinski([points[2],
                    get_mid(points[2], points[1]),
                    get_mid(points[0], points[2])],
                degree-1, my_turtle)

def main():
   my_turtle = turtle.Turtle()
   my_win = turtle.Screen()
   my_points = [[-100, -50], [0, 100], [100, -50]]
   sierpinski(my_points, 3, my_turtle)
   my_win.exitonclick()
```

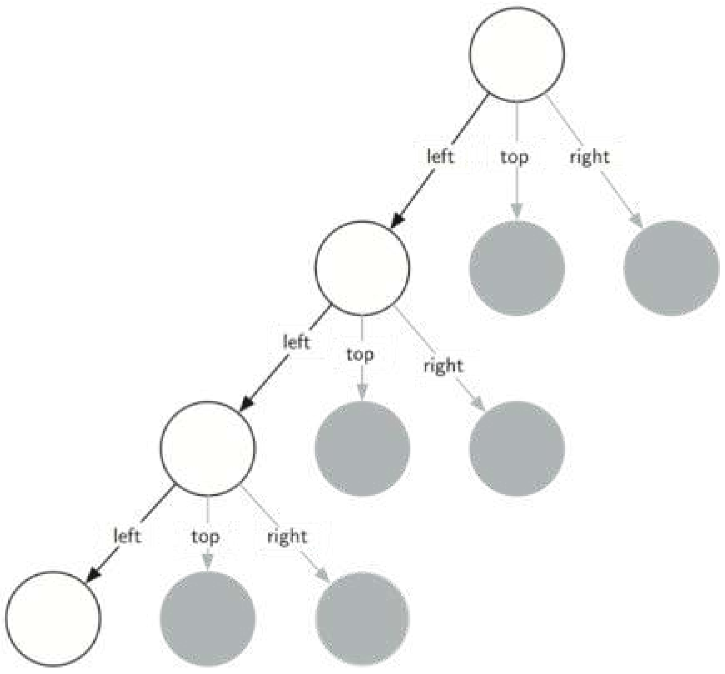


Figure 4.10: Building a Sierpinski Triangle

```
main()
```

The first thing sierpinski does is draw the outer triangle. Next, there are three recursive calls, one for each of the new corner triangles we get when we connect the midpoints. Once again we make use of the standard turtle module that comes with Python. You can learn all the details of the methods available in the turtle module by using help('turtle') from the Python prompt.

Look at the code and think about the order in which the triangles will be drawn. While the exact order of the corners depends upon how the initial set is specified, let's assume that the corners are ordered lower left, top, lower right. Because of the way the sierpinski function calls itself, sierpinski works its way to the smallest allowed triangle in the lower-left corner, and then begins to fill out the rest of the triangles working back. Then it fills in the triangles in the top corner by working toward the smallest, topmost triangle. Finally, it fills in the lower-right corner, working its way toward the smallest triangle in the lower right.

Sometimes it is helpful to think of a recursive algorithm in terms of a diagram of function calls. Figure 4.10 shows that the recursive calls are always made going to the left. The active functions are outlined in black, and the inactive function calls are in gray. The farther you go toward the bottom of Figure 4.10, the smaller the triangles. The function finishes drawing one level at a time; once it is finished with the bottom left it moves to the bottom middle, and so on.

The sierpinski function relies heavily on the getMid function. getMid takes as arguments two endpoints and returns the point halfway between them. In addition, the code has a function that draws a filled triangle using the begin_fill and end_fill turtle methods. This means that each degree of the Sierpinski triangle is drawn in a different color.

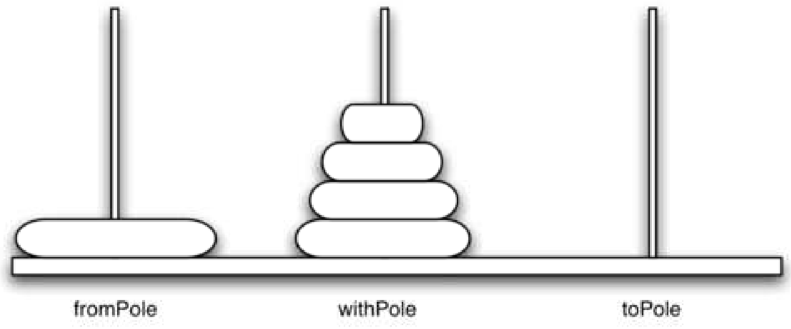


Figure 4.11: An Example Arrangement of Disks for the Tower of Hanoi

4.5 Complex Recursive Problems

In the previous sections we looked at some problems that are relatively easy to solve and some graphically interesting problems that can help us gain a mental model of what is happening in a recursive algorithm. In this section we will look at some problems that are really difficult to solve using an iterative programming style but are very elegant and easy to solve using recursion. We will finish up by looking at a deceptive problem that at first looks like it has an elegant recursive solution but in fact does not.

4.5.1 The Towers Of Hanoi

The Tower of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883. He was inspired by a legend that tells of a Hindu temple where the puzzle was presented to young priests. At the beginning of time, the priests were given three poles and a stack of 64 gold disks, each disk a little smaller than the one beneath it. Their assignment was to transfer all 64 disks from one of the three poles to another, with two important constraints. They could only move one disk at a time, and they could never place a larger disk on top of a smaller one. The priests worked very efficiently, day and night, moving one disk every second. When they finished their work, the legend said, the temple would crumble into dust and the world would vanish.

Although the legend is interesting, you need not worry about the world ending any time soon. The number of moves required to correctly move a tower of 64 disks is $264 - 1 = 18,446,744,073,709,551,615$. At a rate of one move per second, that is $584,942,417,355$ years! Clearly there is more to this puzzle than meets the eye.

Figure 4.11 shows an example of a configuration of disks in the middle of a move from the first peg to the third. Notice that, as the rules specify, the disks on each peg are stacked so that smaller disks are always on top of the larger disks. If you have not tried to solve this puzzle before, you should try it now. You do not need fancy disks and poles-a pile of books or pieces of paper will work.

How do we go about solving this problem recursively? How would you go about solving this problem at all? What is our base case? Let's think about this problem from the bottom up. Suppose you have a tower of five disks, originally on peg one. If you already knew how to move a tower of four disks to peg two, you could then easily move the bottom disk to peg three, and then move the tower of four from peg two to peg three. But what if you do not know how to move a tower of height four? Suppose that you knew how to move a tower of height three to peg three; then it would be easy to move the fourth disk to peg two and move the three from peg three on top of it. But what if you do not know how to move a tower of three? How about moving a tower of two disks to peg two and then moving the third disk to peg three, and then moving the tower of height two on top of it? But what if you still do not know how to do this? Surely you would agree that moving a single disk to peg three is easy enough, trivial you might even say. This sounds like a base case in the making.

Here is a high-level outline of how to move a tower from the starting pole, to the goal pole, using an intermediate pole:

1. Move a tower of height-1 to an intermediate pole, using the final pole.
2. Move the remaining disk to the final pole.
3. Move the tower of height-1 from the intermediate pole to the final pole using the original pole.

As long as we always obey the rule that the larger disks remain on the bottom of the stack, we can use the three steps above recursively, treating any larger disks as though they were not even there. The only thing missing from the outline above is the identification of a base case. The simplest Tower of Hanoi problem is a tower of one disk. In this case, we need move only a single disk to its final destination. A tower of one disk will be our base case. In addition, the steps outlined above move us toward the base case by reducing the height of the tower in steps 1 and 3.

```
def move_tower(height, from_pole, to_pole, with_pole):
    if height >= 1:
        move_tower(height - 1, from_pole, with_pole, to_pole)
        move_disk(from_pole, to_pole)
        move_tower(height - 1, with_pole, to_pole, from_pole)
```

Notice this code is almost identical to the English description. The key to the simplicity of the algorithm is that we make two different recursive calls, one on line 3 and a second on line 5. On line 3 we move all but the bottom disk on the initial tower to an intermediate pole. The next line simply moves the bottom disk to its final resting place. Then on line 5 we move the tower from the intermediate pole to the top of the largest disk. The base case is detected when the tower height is 0; in this case there is nothing to do, so the moveTower function simply returns. The important thing to remember about handling the base case this way is that simply returning from moveTower is what finally allows the moveDisk function to be called.

The function moveDisk, shown below, is very simple. All it does is print out that it is moving a disk from one pole to another. If you type in and run the moveTower program you can see that it gives you a very efficient solution to the puzzle.

```
def move_disk(fp,tp):
    print("moving disk from",fp,"to",tp)
```

The following program provides the entire solution for three disks.

```
def move_tower(height, from_pole, to_pole, with_pole):
   if height >= 1:
      move_tower(height - 1, from_pole, with_pole, to_pole)
      move_disk(from_pole, to_pole)
      move_tower(height - 1, with_pole, to_pole, from_pole)

def move_disk(fp,tp):
     print("moving disk from",fp,"to",tp

move_tower(3, "A", "B", "C")
```

Now that you have seen the code for both moveTower and moveDisk, you may be wondering why we do not have a data structure that explicitly keeps track of what disks are on what poles. Here is a hint: if you were going to explicitly keep track of the disks, you would probably use three Stack objects, one for each pole. The answer is that Python provides the stacks that we need implicitly through the call stack.

4.6 Exploring a Maze

In this section we will look at a problem that has relevance to the expanding world of robotics: How do you find your way out of a maze? If you have a Roomba vacuum cleaner for your dorm room (don't all college students?) you will wish that you could reprogram it using what you have learned in this section. The problem we want to solve is to help our turtle find its way out of a virtual maze. The maze problem has roots as deep as the Greek myth about Theseus who was sent into a maze to kill the minotaur. Theseus used a ball of thread to help him find his way back out again once he had finished off the beast. In our problem we will assume that our turtle is dropped down somewhere into the middle of the maze and must find its way out. Look at Figure 4.12 to get an idea of where we are going in this section.

To make it easier for us we will assume that our maze is divided up into "squares." Each square of the maze is either open or occupied by a section of wall. The turtle can only pass through the open squares of the maze. If the turtle bumps into a wall it must try a different direction. The turtle will require a systematic procedure to find its way out of the maze. Here is the procedure:

- From our starting position we will first try going North one square and then recursively try our procedure from there.
- If we are not successful by trying a Northern path as the first step then we will take a step to the South and recursively repeat our procedure.
- If South does not work then we will try a step to the West as our first step and recursively apply our procedure.
- If North, South, and West have not been successful then apply the procedure recursively from a position one step to our East.
- If none of these directions works then there is no way to get out of the maze and we fail.

Now, that sounds pretty easy, but there are a couple of details to talk about first. Suppose we take our first recursive step by going North. By following our procedure our next step would

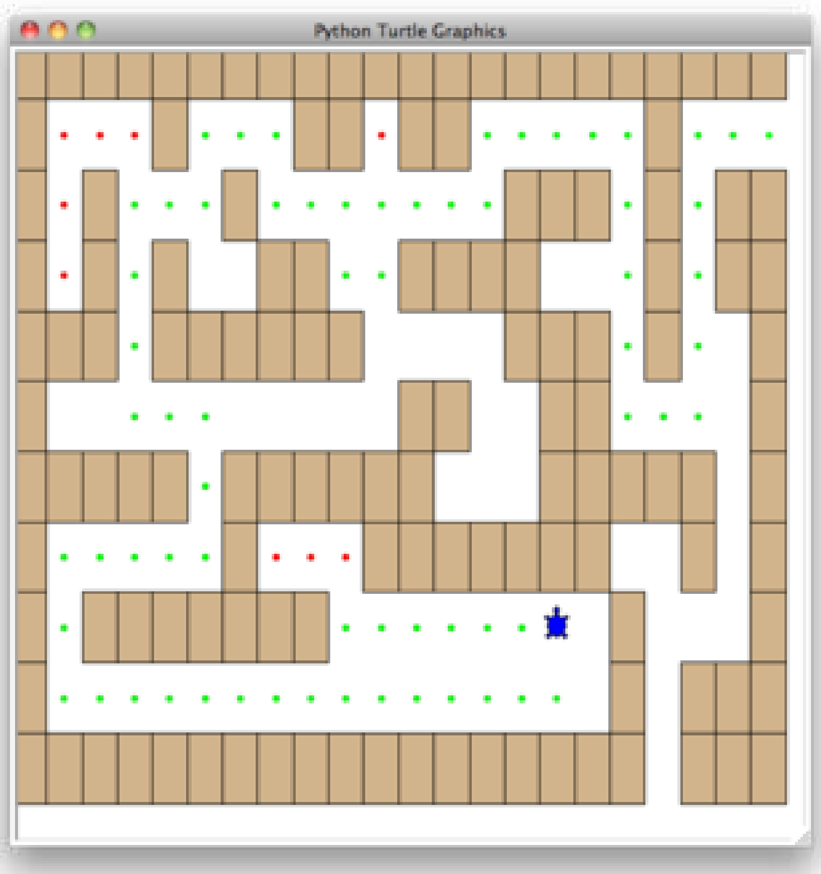


Figure 4.12: The Finished Maze Search Program

also be to the North. But if the North is blocked by a wall we must look at the next step of the procedure and try going to the South. Unfortunately that step to the south brings us right back to our original starting place. If we apply the recursive procedure from there we will just go back one step to the North and be in an infinite loop. So, we must have a strategy to remember where we have been. In this case we will assume that we have a bag of bread crumbs we can drop along our way. If we take a step in a certain direction and find that there is a bread crumb already on that square, we know that we should immediately back up and try the next direction in our procedure. As we will see when we look at the code for this algorithm, backing up is as simple as returning from a recursive function call.

As we do for all recursive algorithms let us review the base cases. Some of them you may already have guessed based on the description in the previous paragraph. In this algorithm, there are four base cases to consider:

1. The turtle has run into a wall. Since the square is occupied by a wall no further exploration can take place.
2. The turtle has found a square that has already been explored. We do not want to continue exploring from this position or we will get into a loop.
3. We have found an outside edge, not occupied by a wall. In other words we have found an exit from the maze.
4. We have explored a square unsuccessfully in all four directions.

For our program to work we will need to have a way to represent the maze. To make this even more interesting we are going to use the turtle module to draw and explore our maze so we can watch this algorithm in action. The maze object will provide the following methods for us to use in writing our search algorithm:

- `__init__` Reads in a data file representing a maze, initializes the internal representation of the maze, and finds the starting position for the turtle.
- draw_maze Draws the maze in a window on the screen.
- update_position Updates the internal representation of the maze and changes the position of the turtle in the window.
- is_exit Checks to see if the current position is an exit from the maze.

The Maze class also overloads the index operator `[]` so that our algorithm can easily access the status of any particular square.

Let's examine the code for the search function which we call searchFrom. The code is shown below. Notice that this function takes three parameters: a maze object, the starting row, and the starting column. This is important because as a recursive function the search logically starts again with each recursive call.

```
def search_from(maze, start_row, start_column):
    maze.update_position(start_row, start_column)
   # Check for base cases:
   # 1. We have run into an obstacle, return false
   if maze[start_row][start_column] == OBSTACLE :
        return False
```

```
7       # 2. We have found a square that has already been explored
8       if maze[start_row][start_column] == TRIED:
9          return False
10      # 3. Success, an outside edge not occupied by an obstacle
11      if maze.is_exit(start_row, start_column):
12         maze.update_position(start_row, start_column, PART_OF_PATH)
13         return True
14      maze.update_position(start_row, start_column, TRIED)
15
16      # Otherwise, use logical short circuiting to try each
17      # direction in turn (if needed)
18      found = search_from(maze, start_row - 1, start_column) or \
19              search_from(maze, start_row + 1, start_column) or \
20              search_from(maze, start_row, start_column - 1) or \
21              search_from(maze, start_row, start_column + 1)
22      if found:
23         maze.update_position(start_row, start_column, PART_OF_PATH)
24      else:
25         maze.update_position(start_row, start_column, DEAD_END)
26      return found
```

As you look through the algorithm you will see that the first thing the code does (line 2) is call update_position. This is simply to help you visualize the algorithm so that you can watch exactly how the turtle explores its way through the maze. Next the algorithm checks for the first three of the four base cases: Has the turtle run into a wall (line 5)? Has the turtle circled back to a square already explored (line 8)? Has the turtle found an exit (line 11)? If none of these conditions is true then we continue the search recursively.

You will notice that in the recursive step there are four recursive calls to searchFrom. It is hard to predict how many of these recursive calls will be used since they are all connected by or statements. If the first call to searchFrom returns True then none of the last three calls would be needed. You can interpret this as meaning that a step to (row-1,column) (or North if you want to think geographically) is on the path leading out of the maze. If there is not a good path leading out of the maze to the North then the next recursive call is tried, this one to the South. If South fails then try West, and finally East. If all four recursive calls return false then we have found a dead end. You should download or type in the whole program and experiment with it by changing the order of these calls.

The code for the Maze class is shown below. The `__init__` method takes the name of a file as its only parameter. This file is a text file that represents a maze by using "+" characters for walls, spaces for open squares, and the letter "S" to indicate the starting position. An example of a maze data file could look as follows:

```
++++++++++++++++++++++
+  +  ++ ++   +
+ +  +     +++ + ++
+ + + ++ ++++ + ++
+++ ++++++ +++ + +
+       ++ ++  +
+++++ ++++++ +++++ +
+    +  +++++++ + +
+ +++++++  S +  +
+            + +++
++++++++++++++++++ +++
```

The internal representation of the maze is a list of lists. Each row of the maze_list instance variable is also a list. This secondary list contains one character per square using the characters described above. For the data file shown above the internal representation looks like the following:

```
[ ['+','+','+','+',...,'+','+','+','+','+','+','+'],
  ['+',' ',' ',' ',...,' ',' ',' ','+',' ',' ',' '],
  ['+',' ','+',' ',...,'+','+',' ','+',' ','+','+'],
  ['+',' ','+',' ',...,' ',' ',' ','+',' ','+','+'],
  ['+','+','+',' ',...,'+','+',' ','+',' ',' ','+'],
  ['+',' ',' ',' ',...,'+','+',' ',' ',' ',' ','+'],
  ['+','+','+','+',...,'+','+','+','+','+',' ','+'],
  ['+',' ',' ',' ',...,'+','+',' ',' ','+',' ','+'],
  ['+',' ','+','+',...,' ',' ','+',' ',' ',' ','+'],
  ['+',' ',' ',' ',...,' ',' ','+',' ','+','+','+'],
  ['+','+','+','+',...,'+','+','+',' ','+','+','+']]
```

The update_position method, as shown below uses the same internal representation to see if the turtle has run into a wall. It also updates the internal representation with a "." or "–" to indicate that the turtle has visited a particular square or if the square is part of a dead end. In addition, the update_position method uses two helper methods, move_turtle and drop_bread_crumb, to update the view on the screen.

Finally, the is_exit method uses the current position of the turtle to test for an exit condition. An exit condition is whenever the turtle has navigated to the edge of the maze, either row zero or column zero, or the far right column or the bottom row.

```
class Maze:
   def __init__(self, maze_file_name):
      rows_in_maze = 0
      columns_in_maze = 0
      self.maze_list = []
      maze_file = open(maze_file_name,'r')
      rows_in_maze = 0
      for line in maze_file:
         row_list = []
         col = 0
         for ch in line[:-1]:
            row_list.append(ch)
            if ch == 'S':
               self.start_row = rows_in_maze
               self.start_col = col
            col = col + 1
         rows_in_maze = rows_in_maze + 1
         self.maze_list.append(row_list)
```

```
            columns_in_maze = len(row_list)

        self.rows_in_maze = rows_in_maze
        self.columns_in_maze = columns_in_maze
        self.x_translate = - columns_in_maze / 2
        self.y_translate = rows_in_maze / 2
        self.t = Turtle(shape = 'turtle')
        setup(width = 600, height = 600)
        setworldcoordinates(- (columns_in_maze - 1) / 2 - .5,
                            - (rows_in_maze - 1) / 2 - .5,
                            (columns_in_maze - 1) / 2 + .5,
                            (rows_in_maze - 1) / 2 + .5)

    def draw_maze(self):
        for y in range(self.rows_in_maze):
            for x in range(self.columns_in_maze):
                if self.maze_list[y][x] == OBSTACLE:
                    self.draw_centered_box(x + self.x_translate,
                      - y + self.y_translate, 'tan')
        self.t.color('black', 'blue')

    def draw_centered_box(self, x, y, color):
        tracer(0)
        self.t.up()
        self.t.goto(x-.5,y-.5)
        self.t.color('black',color)
        self.t.setheading(90)
        self.t.down()
        self.t.begin_fill()
        for i in range(4):
            self.t.forward(1)
            self.t.right(90)
            self.t.end_fill()
            update()
            tracer(1)

    def move_turtle(self, x, y):
        self.t.up()
        self.t.setheading(self.t.towards(x + self.x_translate,
            - y + self.y_translate))
        self.t.goto(x + self.x_translate, - y + self.y_translate)

    def drop_bread_crumb(self, color):
        self.t.dot(color)

    def update_position(self, row, col, val=None):
        if val:
            self.maze_list[row][col] = val
        self.move_turtle(col, row)

        if val == PART_OF_PATH:
```

```
      color = 'green'
    elif val == OBSTACLE:
      color = 'red'
    elif val == TRIED:
      color = 'black'
    elif val == DEAD_END:
      color = 'red'
      else:
      color = None

    if color:
      self.drop_bread_crumb(color)

  def is_exit(self, row, col):
    return (row == 0 or
        row == self.rows_in_maze - 1 or
        col == 0 or
        col == self.columns_in_maze - 1)

  def __getitem__(self, idx):
    return self.maze_list[idx]
```

The complete program is shown below. This program uses the data file maze2.txt shown below which stores the following maze:

```
++++++++++++++++++++++
+  +  ++ ++     +
    +    ++++++++++
+ +  ++ ++++ +++ ++
+ +  + + ++  +++ +
+       ++ ++ + +
+++++ + +  ++ + +
+++++ +++ + + ++ +
+       + + S+ + +
+++++ + + + + + +
++++++++++++++++++++++
```

Note that it is a much more simple example file in that the exit is very close to the starting position of the turtle.

```
# Completed maze program
# Takes maze2.txt as input

import turtle

PART_OF_PATH = 'O'
TRIED = '.'
OBSTACLE = '+'
DEAD_END = '-'
```

```
class Maze:
   def __init__(self, maze_file_name):
      rows_in_maze = 0
      columns_in_maze = 0
      self.maze_list = []
      maze_file = open(maze_file_name,'r')
      rows_in_maze = 0
      for line in maze_file:
         row_list = []
         col = 0
         for ch in line[: -1]:
            row_list.append(ch)
            if ch == 'S':
               self.start_row = rows_in_maze
               self.start_col = col
            col = col + 1
         rows_in_maze = rows_in_maze + 1
         self.maze_list.append(row_list)
         columns_in_maze = len(row_list)

      self.rows_in_maze = rows_in_maze
      self.columns_in_maze = columns_in_maze
      self.x_translate = - columns_in_maze / 2
      self.y_translate = rows_in_maze / 2
      self.t = turtle.Turtle()
      self.t.shape('turtle')
      self.wn = turtle.Screen()
      self.wn.setworldcoordinates(- (columns_in_maze - 1) / 2 - .5,
             - (rows_in_maze - 1) / 2 - .5,
             (columns_in_maze - 1) / 2 + .5,
             (rows_in_maze - 1) / 2 + .5)

   def draw_maze(self):
      self.t.speed(10)
      for y in range(self.rows_in_maze):
         for x in range(self.columns_in_maze):
            if self.maze_list[y][x] == OBSTACLE:
               self.draw_centered_box(x + self.x_translate,
                  - y + self.y_translate, 'orange')
      self.t.color('black')
      self.t.fillcolor('blue')

   def draw_centered_box(self, x, y, color):
      self.t.up()
      self.t.goto(x - .5, y - .5)
      self.t.color(color)
      self.t.fillcolor(color)
      self.t.setheading(90)
      self.t.down()
      self.t.begin_fill()
      for i in range(4):
            self.t.forward(1)
            self.t.right(90)
        self.t.end_fill()

    def move_turtle(self, x, y):
        self.t.up()
        self.t.setheading(self.t.towards(x + self.x_translate,
                    - y + self.y_translate))
        self.t.goto(x + self.x_translate, - y + self.y_translate)

    def drop_bread_crumb(self, color):
        self.t.dot(10, color)

    def update_position(self, row, col, val=None):
        if val:
            self.maze_list[row][col] = val
        self.move_turtle(col, row)

        if val == PART_OF_PATH:
            color = 'green'
        elif val == OBSTACLE:
            color = 'red'
        elif val == TRIED:
            color = 'black'
        elif val == DEAD_END:
            color = 'red'
        else:
            color = None

        if color:
            self.drop_bread_crumb(color)

    def is_exit(self, row, col):
        return (row == 0 or
                row == self.rows_in_maze - 1 or
                col == 0 or
                col == self.columns_in_maze - 1)

    def __getitem__(self,idx):
        return self.maze_list[idx]


def search_from(maze, start_row, start_column):
    # try each of four directions from this point until we find a
        way out.
    # base Case return values:
    #  1. We have run into an obstacle, return false
    maze.update_position(start_row, start_column)
    if maze[start_row][start_column] == OBSTACLE :
        return False
    #  2. We have found a square that has already been explored
    if maze[start_row][start_column] == TRIED or
       maze[start_row][start_column] == DEAD_END:
       return False
    # 3. We have found an outside edge not occupied by an obstacle
    if maze.is_exit(start_row,start_column):
       maze.update_position(start_row, start_column, PART_OF_PATH)
       return True
    maze.update_position(start_row, start_column, TRIED)
    # Otherwise, use logical short circuiting to try each direction
    # in turn (if needed)
    found = search_from(maze, start_row-1, start_column) or \
          search_from(maze, start_row+1, start_column) or \
          search_from(maze, start_row, start_column-1) or \
          search_from(maze, start_row, start_column+1)
    if found:
       maze.update_position(start_row, start_column, PART_OF_PATH)
    else:
       maze.update_position(start_row, start_column, DEAD_END)
    return found


my_maze = Maze('maze2.txt')
my_maze.draw_maze()
my_maze.update_position(my_maze.start_row, my_maze.start_col)

search_from(my_maze, my_maze.start_row, my_maze.start_col)
```

Self Check

Modify the maze search program so that the calls to searchFrom are in a different order. Watch the program run. Can you explain why the behavior is different? Can you predict what path the turtle will follow for a given change in order?

4.7 Summary

In this chapter we have looked at examples of several recursive algorithms. These algorithms were chosen to expose you to several different problems where recursion is an effective problem-solving technique. The key points to remember from this chapter are as follows:

- All recursive algorithms must have a base case.
- A recursive algorithm must change its state and make progress toward the base case.
- A recursive algorithm must call itself (recursively).
- Recursion can take the place of iteration in some cases.
- Recursive algorithms often map very naturally to a formal expression of the problem you are trying to solve.

- Recursion is not always the answer. Sometimes a recursive solution may be more computationally expensive than an alternative algorithm.content...

4.8 Key Terms

base case	decrypt	recursion
recursive call	stack frame	

4.9 Discussion Questions

1. Draw a call stack for the Tower of Hanoi problem. Assume that you start with a stack of three disks.
2. Using the recursive rules as described, draw a Sierpinski triangle using paper and pencil.

4.10 Programming Exercises

- Write a recursive function to compute the factorial of a number.
- Write a recursive function to reverse a list.
- Modify the recursive tree program using one or all of the following ideas:
  - Modify the thickness of the branches so that as the branchLen gets smaller, the line gets thinner.
  - Modify the color of the branches so that as the branchLen gets very short it is colored like a leaf.
  - Modify the angle used in turning the turtle so that at each branch point the angle is selected at random in some range. For example choose the angle between 15 and 45 degrees. Play around to see what looks good.
  - Modify the branchLen recursively so that instead of always subtracting the same amount you subtract a random amount in some range.

  If you implement all of the above ideas you will have a very realistic looking tree.
- Find or invent an algorithm for drawing a fractal mountain. Hint: One approach to this uses triangles again.
- Write a recursive function to compute the Fibonacci sequence. How does the performance of the recursive function compare to that of an iterative version?
- Implement a solution to the Tower of Hanoi using three stacks to keep track of the disks.
- Using the turtle graphics module, write a recursive program to display a Hilbert curve.
- Using the turtle graphics module, write a recursive program to display a Koch snowflake.

- Write a program to solve the following problem: You have two jugs: a 4-gallon jug and a 3-gallon jug. Neither of the jugs have markings on them. There is a pump that can be used to fill the jugs with water. How can you get exactly two gallons of water in the 4-gallon jug?
- Generalize the problem above so that the parameters to your solution include the sizes of each jug and the final amount of water to be left in the larger jug.
- Write a program that solves the following problem: Three missionaries and three cannibals come to a river and find a boat that holds two people. Everyone must get across the river to continue on the journey. However, if the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten. Find a series of crossings that will get everyone safely to the other side of the river.
- Modify the Tower of Hanoi program using turtle graphics to animate the movement of the disks. Hint: You can make multiple turtles and have them shaped like rectangles.
- Pascal's triangle is a number triangle with numbers arranged in staggered rows such that $a_nr = \frac{n!}{r!(n-r)!}$ This equation is the equation for a binomial coefficient. You can build Pascal's triangle by adding the two numbers that are diagonally above a number in the triangle. An example of Pascal's triangle is shown below.

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

  Write a program that prints out Pascal's triangle. Your program should accept a parameter that tells how many rows of the triangle to print.

CHAPTER

FIVE

SORTING AND SEARCHING

5.1 Objectives

- To be able to explain and implement sequential search and binary search.
- To be able to explain and implement selection sort, bubble sort, merge sort, quick sort, insertion sort, and shell sort.
- To understand the idea of hashing as a search technique.
- To introduce the map abstract data type.
- To implement the map abstract data type using hashing.

5.2 Searching

We will now turn our attention to some of the most common problems that arise in computing, those of searching and sorting. In this section we will study searching. We will return to sorting later in the chapter. Searching is the algorithmic process of finding a particular item in a collection of items. A search typically answers either True or False as to whether the item is present. On occasion it may be modified to return where the item is found. For our purposes here, we will simply concern ourselves with the question of membership.

In Python, there is a very easy way to ask whether an item is in a list of items. We use the in operator.

```
>>> 15 in [3,5,2,4,1]
False
>>> 3 in [3,5,2,4,1]
True
>>>
```

Even though this is easy to write, an underlying process must be carried out to answer the question. It turns out that there are many different ways to search for the item. What we are interested in here is how these algorithms work and how they compare to one another.

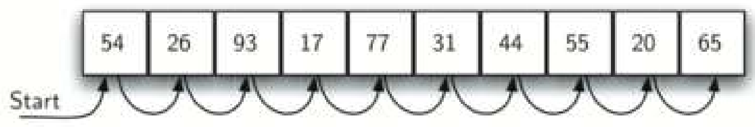


Figure 5.1: The Sequential Search of a List of Integers

5.2.1 The Sequential Search

When data items are stored in a collection such as a list, we say that they have a linear or sequential relationship. Each data item is stored in a position relative to the others. In Python lists, these relative positions are the index values of the individual items. Since these index values are ordered, it is possible for us to visit them in sequence. This process gives rise to our first searching technique, the sequential search.

Figure 5.1 shows how this search works. Starting at the first item in the list, we simply move from item to item, following the underlying sequential ordering until we either find what we are looking for or run out of items. If we run out of items, we have discovered that the item we were searching for was not present.

The Python implementation for this algorithm is shown below. The function needs the list and the item we are looking for and returns a boolean value as to whether it is present. The boolean variable found is initialized to False and is assigned the value True if we discover the item in the list.

```
def sequential_search(a_list, item):
   pos = 0
   found = False

   while pos < len(a_list) and not found:
      if a_list[pos] == item:
         found = True
       else:
          pos = pos+1

   return found

test_list = [1, 2, 32, 8, 17, 19, 42, 13, 0]
print(sequential_search(test_list, 3))
print(sequential_search(test_list, 13))
```

Analysis of Sequential Search

To analyze searching algorithms, we need to decide on a basic unit of computation. Recall that this is typically the common step that must be repeated in order to solve the problem. For searching, it makes sense to count the number of comparisons performed. Each comparison may or may not discover the item we are looking for. In addition, we make another assumption here. The list of items is not ordered in any way. The items have been placed randomly into the

Case	Best Case	Worst Case	Average Case
item is present	1	n	$\frac{n}{2}$
item is not present	n	n	n

Table 5.1: Comparisons Used in a Sequential Search of an Unordered List

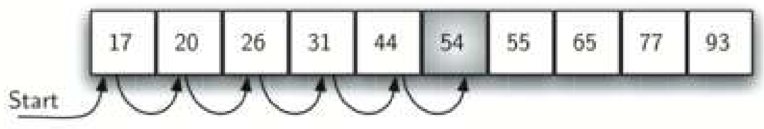


Figure 5.2: Sequential Search of an Ordered List of Integers

list. In other words, the probability that the item we are looking for is in any particular position is exactly the same for each position of the list.

If the item is not in the list, the only way to know it is to compare it against every item present. If there are n items, then the sequential search requires n comparisons to discover that the item is not there. In the case where the item is in the list, the analysis is not so straightforward. There are actually three different scenarios that can occur. In the best case we will find the item in the first place we look, at the beginning of the list. We will need only one comparison. In the worst case, we will not discover the item until the very last comparison, the nth comparison.

What about the average case? On average, we will find the item about halfway into the list; that is, we will compare against $\frac{n}{2}$ items. Recall, however, that as n gets large, the coefficients, no matter what they are, become insignificant in our approximation, so the complexity of the sequential search, is $O(n)$. Table 5.1 summarizes these results.

We assumed earlier that the items in our collection had been randomly placed so that there is no relative order between the items. What would happen to the sequential search if the items were ordered in some way? Would we be able to gain any efficiency in our search technique?

Assume that the list of items was constructed so that the items were in ascending order, from low to high. If the item we are looking for is present in the list, the chance of it being in any one of the n positions is still the same as before. We will still have the same number of comparisons to find the item. However, if the item is not present there is a slight advantage. Figure 5.2 shows this process as the algorithm looks for the item 50. Notice that items are still compared in sequence until 54. At this point, however, we know something extra. Not only is 54 not the item we are looking for, but no other elements beyond 54 can work either since the list is sorted. In this case, the algorithm does not have to continue looking through all of the items to report that the item was not found. It can stop immediately.

```
def ordered_sequential_search(a_list, item):
  pos = 0
  found = False
  stop = False
  while pos < len(a_list) and not found and not stop:
    if a_list[pos] == item:
      found = True
    else:
```

Case	**Best Case**	**Worst Case**	**Average Case**
item is present	1	n	$\frac{n}{2}$
item is not present	1	n	$\frac{n}{2}$

Table 5.2: Comparisons Used in Sequential Search of an Ordered List

```
            if a_list[pos] > item:
                stop = True
        else:
            pos = pos+1

    return found

test_list = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(ordered_sequential_search(test_list, 3))
print(ordered_sequential_search(test_list, 13))
```

Table 5.2 summarizes these results. Note that in the best case we might discover that the item is not in the list by looking at only one item. On average, we will know after looking through only $\frac{n}{2}$ items. However, this technique is still $O(n)$. In summary, a sequential search is improved by ordering the list only in the case where we do not find the item.

Self Check

Suppose you are doing a sequential search of the list $[15, 18, 2, 19, 18, 0, 8, 14, 19, 14]$. How many comparisons would you need to do in order to find the key 18?

1. 5
2. 10
3. 4
4. 2

Suppose you are doing a sequential search of the ordered list $[3, 5, 6, 8, 11, 12, 14, 15, 17, 18]$. How many comparisons would you need to do in order to find the key 13?

1. 10
2. 5
3. 7
4. 6

5.2.2 The Binary Search

It is possible to take greater advantage of the ordered list if we are clever with our comparisons. In the sequential search, when we compare against the first item, there are at most $n - 1$ more items to look through if the first item is not what we are looking for. Instead of searching the

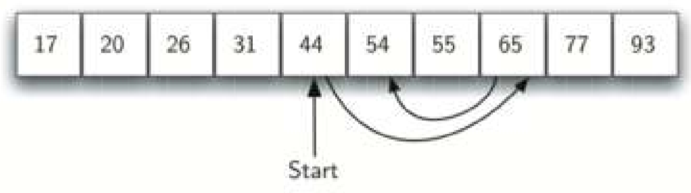


Figure 5.3: Binary Search of an Ordered List of Integers

list in sequence, a **binary search** will start by examining the middle item. If that item is the one we are searching for, we are done. If it is not the correct item, we can use the ordered nature of the list to eliminate half of the remaining items. If the item we are searching for is greater than the middle item, we know that the entire lower half of the list as well as the middle item can be eliminated from further consideration. The item, if it is in the list, must be in the upper half.

We can then repeat the process with the upper half. Start at the middle item and compare it against what we are looking for. Again, we either find it or split the list in half, therefore eliminating another large part of our possible search space. Figure 5.3 shows how this algorithm can quickly find the value 54.

```
def binary_search(a_list, item):
    first = 0
    last = len(a_list) - 1
    found = False

    while first <= last and not found:
        midpoint = (first + last) // 2
        if a_list[midpoint] == item:
            found = True
        else:
            if item < a_list[midpoint]:
                last = midpoint - 1
        else:
            first = midpoint + 1
    return found

test_list = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binary_search(test_list, 3))
print(binary_search(test_list, 13))
```

Before we move on to the analysis, we should note that this algorithm is a great example of a divide and conquer strategy. Divide and conquer means that we divide the problem into smaller pieces, solve the smaller pieces in some way, and then reassemble the whole problem to get the result. When we perform a binary search of a list, we first check the middle item. If the item we are searching for is less than the middle item, we can simply perform a binary search of the left half of the original list. Likewise, if the item is greater, we can perform a binary search of the right half. Either way, this is a recursive call to the binary search function passing a smaller list.

Comparisons	Approximate Number Of Items Left
1	$\frac{n}{2}$
2	$\frac{n}{4}$
3	$\frac{n}{8}$
...	...
i	$\frac{n}{2^i}$

Table 5.3: Tabular Analysis for a Binary Search

```
    def binary_search(a_list, item):
        if len(a_list) == 0:
            return False
    else:
        midpoint = len(a_list) // 2
    if a_list[midpoint] == item:
        return True
    else:
        if item < a_list[midpoint]:
            return binary_search(a_list[:midpoint], item)
        else:
            return binary_search(a_list[midpoint + 1:], item)

test_list = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binary_search(test_list, 3))
print(binary_search(test_list, 13))
```

Analysis of Binary Search

To analyze the binary search algorithm, we need to recall that each comparison eliminates about half of the remaining items from consideration. What is the maximum number of comparisons this algorithm will require to check the entire list? If we start with n items, about $\frac{n}{2}$ items will be left after the first comparison. After the second comparison, there will be about $\frac{n}{4}$. Then $\frac{n}{8}$, $\frac{n}{16}$, and so on. How many times can we split the list? Table 5.3 helps us to see the answer.

When we split the list enough times, we end up with a list that has just one item. Either that is the item we are looking for or it is not. Either way, we are done. The number of comparisons necessary to get to this point is i where $\frac{n}{2^i} = 1$. Solving for i gives us $i = \log n$. The maximum number of comparisons is logarithmic with respect to the number of items in the list. Therefore, the binary search is $O(\log n)$.

One additional analysis issue needs to be addressed. In the recursive solution shown above, the recursive call, `binary_search(a_list[:midpoint],item)`.

uses the slice operator to create the left half of the list that is then passed to the next invocation (similarly for the right half as well). The analysis that we did above assumed that the slice operator takes constant time. However, we know that the slice operator in Python is actually $O(k)$. This means that the binary search using slice will not perform in strict logarithmic time. Luckily this can be remedied by passing the list along with the starting and ending indices.

Even though a binary search is generally better than a sequential search, it is important to note that for small values of n, the additional cost of sorting is probably not worth it. In fact, we should always consider whether it is cost effective to take on the extra work of sorting to gain searching benefits. If we can sort once and then search many times, the cost of the sort is not so significant. However, for large lists, sorting even once can be so expensive that simply performing a sequential search from the start may be the best choice.

Self Check

Suppose you have the following sorted list $[3, 5, 6, 8, 11, 12, 14, 15, 17, 18]$ and are using the recursive binary search algorithm. Which group of numbers correctly shows the sequence of comparisons used to find the key 8.

1. $11, 5, 6, 8$
2. $12, 6, 11, 8$
3. $3, 5, 6, 8$
4. $18, 12, 6, 8$

Suppose you have the following sorted list $[3, 5, 6, 8, 11, 12, 14, 15, 17, 18]$ and are using the recursive binary search algorithm. Which group of numbers correctly shows the sequence of comparisons used to search for the key 16?

1. $11, 14, 17$
2. $18, 17, 15$
3. $14, 17, 15$
4. $12, 17, 15$

5.2.3 Hashing

In previous sections we were able to make improvements in our search algorithms by taking advantage of information about where items are stored in the collection with respect to one another. For example, by knowing that a list was ordered, we could search in logarithmic time using a binary search. In this section we will attempt to go one step further by building a data structure that can be searched in $O(1)$ time. This concept is referred to as **hashing**.

In order to do this, we will need to know even more about where the items might be when we go to look for them in the collection. If every item is where it should be, then the search can use a single comparison to discover the presence of an item. We will see, however, that this is typically not the case.

A **hash table** is a collection of items which are stored in such a way as to make it easy to find them later. Each position of the hash table, often called a **slot**, can hold an item and is named by an integer value starting at 0. For example, we will have a slot named 0, a slot named 1, a slot named 2, and so on. Initially, the hash table contains no items so every slot is empty. We can implement a hash table by using a list with each element initialized to the special Python value None. Figure 5.4 shows a hash table of size $m = 11$. In other words, there are m slots in the table, named 0 through 10.

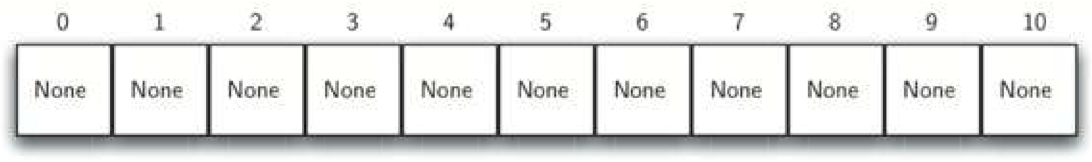


Figure 5.4: Hash Table with 11 Empty Slots

Item	Hash Value
54	10
26	4
93	5
17	6
77	0
31	9

Table 5.4: Simple Hash Function Using Remainders

The mapping between an item and the slot where that item belongs in the hash table is called the hash function. The **hash function** will take any item in the collection and return an integer in the range of slot names, between 0 and $m - 1$. Assume that we have the set of integer items $54, 26, 93, 17, 77$, and 31. Our first hash function, sometimes referred to as the "remainder method," simply takes an item and divides it by the table size, returning the remainder as its hash value $(h(\text{item}) = \text{item}\%11)$. Table 5.4 gives all of the hash values for our example items. Note that this remainder method (modulo arithmetic) will typically be present in some form in all hash functions, since the result must be in the range of slot names.

Once the hash values have been computed, we can insert each item into the hash table at the designated position as shown in Figure 5.5. Note that 6 of the 11 slots are now occupied. This is referred to as the load factor, and is commonly denoted by $\lambda = \frac{\text{number\_of\_items}}{\text{table\_size}}$. For this example, $\lambda = \frac{6}{11}$.

Now when we want to search for an item, we simply use the hash function to compute the slot name for the item and then check the hash table to see if it is present. This searching operation is $O(1)$, since a constant amount of time is required to compute the hash value and then index the hash table at that location. If everything is where it should be, we have found a constant time search algorithm.

You can probably already see that this technique is going to work only if each item maps to a unique location in the hash table. For example, if the item 44 had been the next item in our collection, it would have a hash value of 0 ($44\%11 == 0$). Since 77 also had a hash value of

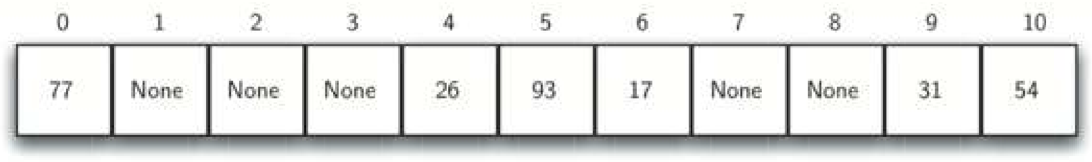


Figure 5.5: Hash Table with Six Items

0, we would have a problem. According to the hash function, two or more items would need to be in the same slot. This is referred to as a collision (it may also be called a "clash"). Clearly, collisions create a problem for the hashing technique. We will discuss them in detail later.

Hash Functions

Given a collection of items, a hash function that maps each item into a unique slot is referred to as a **perfect hash function**. If we know the items and the collection will never change, then it is possible to construct a perfect hash function (refer to the exercises for more about perfect hash functions). Unfortunately, given an arbitrary collection of items, there is no systematic way to construct a perfect hash function. Luckily, we do not need the hash function to be perfect to still gain performance efficiency.

One way to always have a perfect hash function is to increase the size of the hash table so that each possible value in the item range can be accommodated. This guarantees that each item will have a unique slot. Although this is practical for small numbers of items, it is not feasible when the number of possible items is large. For example, if the items were nine-digit Social Security numbers, this method would require almost one billion slots. If we only want to store data for a class of 25 students, we will be wasting an enormous amount of memory.

Our goal is to create a hash function that minimizes the number of collisions, is easy to compute, and evenly distributes the items in the hash table. There are a number of common ways to extend the simple remainder method. We will consider a few of them here.

The **folding method** for constructing hash functions begins by dividing the item into equal-size pieces (the last piece may not be of equal size). These pieces are then added together to give the resulting hash value. For example, if our item was the phone number 436-555-4601, we would take the digits and divide them into groups of 2 $(43, 65, 55, 46, 01)$. After the addition, $43 + 65 + 55 + 46 + 01$, we get 210. If we assume our hash table has 11 slots, then we need to perform the extra step of dividing by 11 and keeping the remainder. In this case $210\%11$ is 1, so the phone number 436-555-4601 hashes to slot 1. Some folding methods go one step further and reverse every other piece before the addition. For the above example, we get $43 + 56 + 55 + 64 + 01 = 219$ which gives $219\%11 = 10$.

Another numerical technique for constructing a hash function is called the **mid-square method**. We first square the item, and then extract some portion of the resulting digits. For example, if the item were 44, we would first compute $44^2 = 1,936$. By extracting the middle two digits, 93, and performing the remainder step, we get 5 $(93\%11)$. Table 5.5 shows items under both the remainder method and the mid-square method. You should verify that you understand how these values were computed.

We can also create hash functions for character-based items such as strings. The word "cat" can be thought of as a sequence of ordinal values.

```
>>> ord('c')
99
>>> ord('a')
97
>>> ord('t')
116
```

Item	Remainder	Mid-Square
54	10	3
26	4	7
93	5	9
17	6	8
77	0	4
31	9	6

Table 5.5: Comparisons of Remainder and Mid-Square Methods

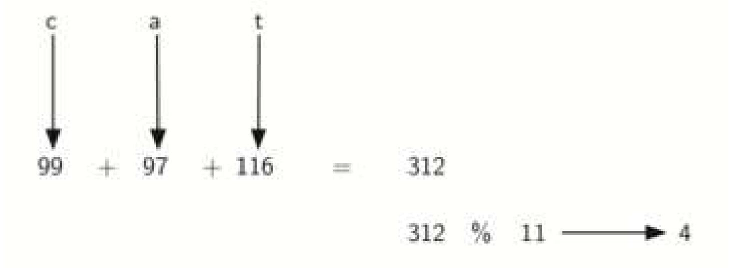


Figure 5.6: Hashing a String Using Ordinal Values

We can then take these three ordinal values, add them up, and use the remainder method to get a hash value (see Figure 5.6). The code below shows a function called **`hash`** that takes a string and a table size and returns the hash value in the range from 0 to **`table_size`**-1.

```
def hash(a_string, table_size):
   sum = 0
   for pos in range(len(a_string)):
      sum = sum + ord(a_string[pos])

   return sum % table_size
```

It is interesting to note that when using this hash function, anagrams will always be given the same hash value. To remedy this, we could use the position of the character as a weight. Figure 5.7 shows one possible way to use the positional value as a weighting factor. The modification to the **`hash`** function is left as an exercise.

You may be able to think of a number of additional ways to compute hash values for items in a collection. The important thing to remember is that the hash function has to be efficient so that it does not become the dominant part of the storage and search process. If the hash function is too complex, then it becomes more work to compute the slot name than it would be to simply do a basic sequential or binary search as described earlier. This would quickly defeat the purpose of hashing.

Collision Resolution

We now return to the problem of collisions. When two items hash to the same slot, we must have a systematic method for placing the second item in the hash table. This process is called **collision resolution**. As we stated earlier, if the hash function is perfect, collisions will never

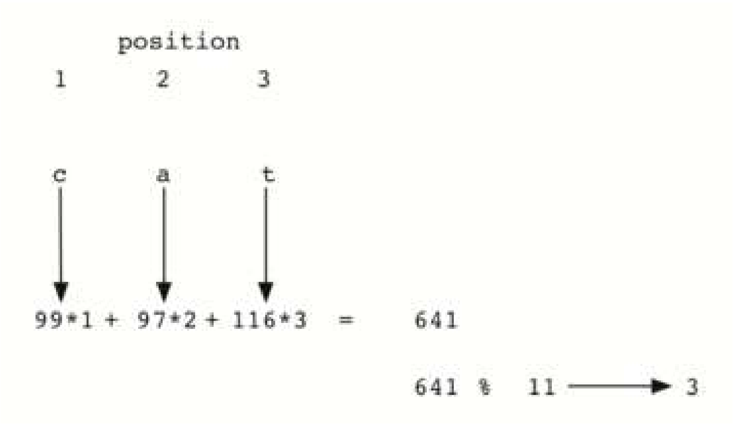


Figure 5.7: Hashing a String Using Ordinal Values with Weighting

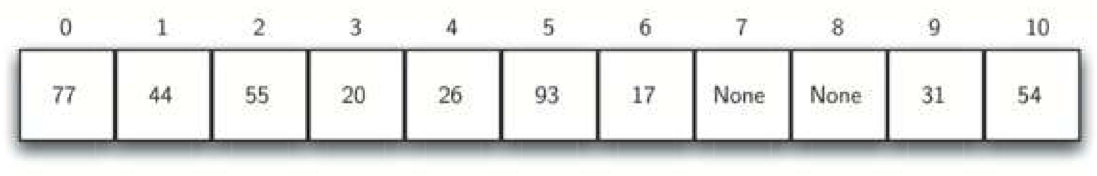


Figure 5.8: Collision Resolution with Linear Probing

occur. However, since this is often not possible, collision resolution becomes a very important part of hashing.

One method for resolving collisions looks into the hash table and tries to find another open slot to hold the item that caused the collision. A simple way to do this is to start at the original hash value position and then move in a sequential manner through the slots until we encounter the first slot that is empty. Note that we may need to go back to the first slot (circularly) to cover the entire hash table. This collision resolution process is referred to as **open addressing** in that it tries to find the next open slot or address in the hash table. By systematically visiting each slot one at a time, we are performing an open addressing technique called **linear probing**. Figure 5.8 shows an extended set of integer items under the simple remainder method hash function $(54, 26, 93, 17, 77, 31, 44, 55, 20)$. Table 5.4 above shows the hash values for the original items. Figure 5.5 shows the original contents. When we attempt to place 44 into slot 0, a collision occurs. Under linear probing, we look sequentially, slot by slot, until we find an open position. In this case, we find slot 1.

Again, 55 should go in slot 0 but must be placed in slot 2 since it is the next open position. The final value of 20 hashes to slot 9. Since slot 9 is full, we begin to do linear probing. We visit slots $10, 0, 1,$ and 2, and finally find an empty slot at position 3.

Once we have built a hash table using open addressing and linear probing, it is essential that we utilize the same methods to search for items. Assume we want to look up the item 93. When we compute the hash value, we get 5. Looking in slot 5 reveals 93, and we can return `True`. What if we are looking for 20? Now the hash value is 9, and slot 9 is currently holding 31. We cannot simply return `False` since we know that there could have been collisions. We are now forced to do a sequential search, starting at position 10, looking until either we find the item 20 or we find an empty slot.

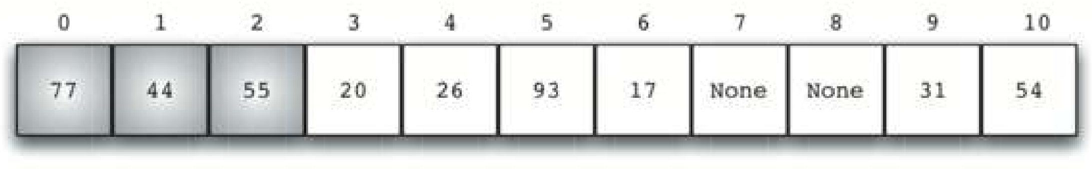


Figure 5.9: A Cluster of Items for Slot 0

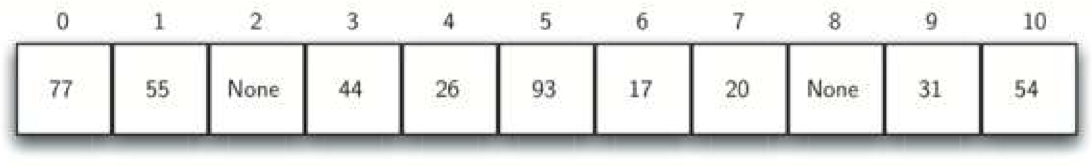


Figure 5.10: Collision Resolution Using "Plus 3"

A disadvantage to linear probing is the tendency for **clustering**; items become clustered in the table. This means that if many collisions occur at the same hash value, a number of surrounding slots will be filled by the linear probing resolution. This will have an impact on other items that are being inserted, as we saw when we tried to add the item 20 above. A cluster of values hashing to 0 had to be skipped to finally find an open position. This cluster is shown in Figure 5.9.

One way to deal with clustering is to extend the linear probing technique so that instead of looking sequentially for the next open slot, we skip slots, thereby more evenly distributing the items that have caused collisions. This will potentially reduce the clustering that occurs. Figure 5.10 shows the items when collision resolution is done with a "plus 3" probe. This means that once a collision occurs, we will look at every third slot until we find one that is empty.

The general name for this process of looking for another slot after a collision is rehashing. With simple linear probing, the rehash function is

$$\mathrm{new\_hash\_value} = \mathrm{rehash(old\_hash\_value)}$$

where

$$\mathrm{rehash(pos)} = (\mathrm{pos} + 1)\%\mathrm{size\_of\_table}.$$

The "plus 3" rehash can be defined as

$$\mathrm{rehash(pos)} = (\mathrm{pos} + 3)\%\mathrm{size\_of\_table}.$$

In general,

$$\mathrm{rehash(pos)} = (\mathrm{pos} + \mathrm{skip})\%\mathrm{sizeoftable}.$$

It is important to note that the size of the "skip" must be such that all the slots in the table will eventually be visited. Otherwise, part of the table will be unused. To ensure this, it is often suggested that the table size be a prime number. This is the reason we have been using 11 in our examples.

A variation of the linear probing idea is called **quadratic probing**. Instead of using a constant "skip" value, we use a rehash function that increments the hash value by $1, 3, 5, 7, 9$, and so on.

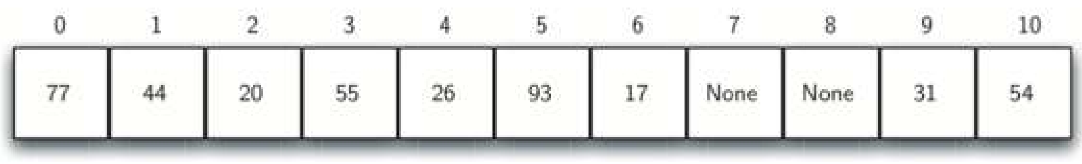


Figure 5.11: Collision Resolution with Quadratic Probing

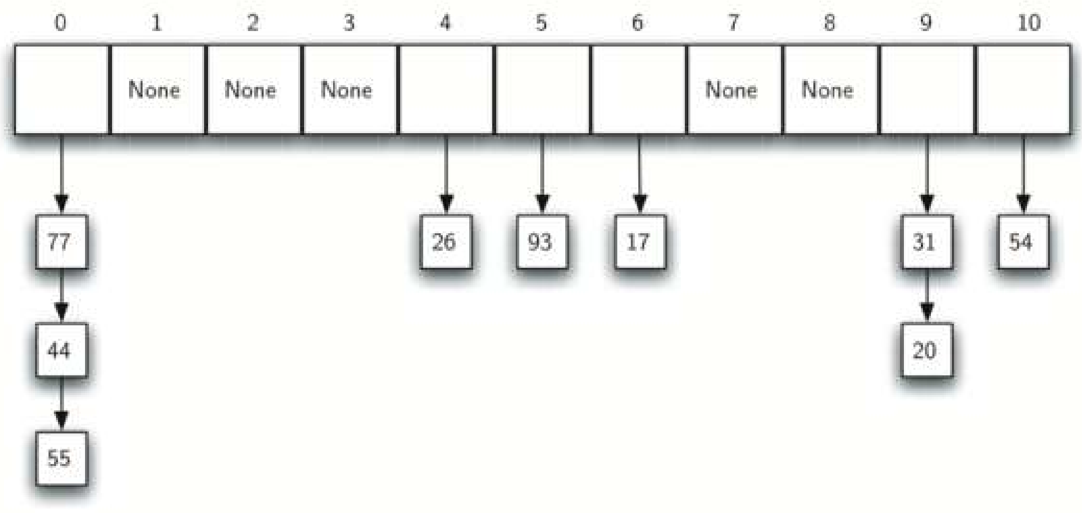


Figure 5.12: Collision Resolution with Quadratic Probing

This means that if the first hash value is h, the successive values are $h+1$, $h+4$, $h+9$, $h+16$, and so on. In other words, quadratic probing uses a skip consisting of successive perfect squares. Figure 5.11 shows our example values after they are placed using this technique.

An alternative method for handling the collision problem is to allow each slot to hold a reference to a collection (or chain) of items. **Chaining** allows many items to exist at the same location in the hash table. When collisions happen, the item is still placed in the proper slot of the hash table. As more and more items hash to the same location, the difficulty of searching for the item in the collection increases. Figure 5.12 shows the items as they are added to a hash table that uses chaining to resolve collisions.

When we want to search for an item, we use the hash function to generate the slot where it should reside. Since each slot holds a collection, we use a searching technique to decide whether the item is present. The advantage is that on the average there are likely to be many fewer items in each slot, so the search is perhaps more efficient. We will look at the analysis for hashing at the end of this section.

Self Check

In a hash table of size 13 which index positions would the following two keys map to? $27, 130$

1. $1, 10$
2. $13, 0$

3. $1, 0$
4. $2, 3$

Suppose you are given the following set of keys to insert into a hash table that holds exactly 11 values: $113, 117, 97, 100, 114, 108, 116, 105, 99$. Which of the following best demonstrates the contents of the has table after all the keys have been inserted using linear probing?

1. $100, \_\_, \_\_, 113, 114, 105, 116, 117, 97, 108, 99$
2. $99, 100, \_\_, 113, 114, \_\_, 116, 117, 105, 97, 108$
3. $100, 113, 117, 97, 14, 108, 116, 105, 99, \_\_, \_\_$
4. $117, 114, 108, 116, 105, 99, \_\_, \_\_, 97, 100, 113$

Implementing the `Map` Abstract Data Type

One of the most useful Python collections is the dictionary. Recall that a dictionary is an associative data type where you can store key-data pairs. The key is used to look up the associated data value. We often refer to this idea as a **map**.

The map abstract data type is defined as follows. The structure is an unordered collection of associations between a key and a data value. The keys in a map are all unique so that there is a one-to-one relationship between a key and a value. The operations are given below.

- `Map()` Create a new, empty map. It returns an empty map collection.
- `put(key,val)` Add a new key-value pair to the map. If the key is already in the map then replace the old value with the new value.
- `get(key)` Given a key, return the value stored in the map or None otherwise.
- `del` Delete the key-value pair from the map using a statement of the form del map[key].
- `len()` Return the number of key-value pairs stored in the map.
- `in` Return `True` for a statement of the form `key in map`, if the given key is in the map, `False` otherwise.

One of the great benefits of a dictionary is the fact that given a key, we can look up the associated data value very quickly. In order to provide this fast look up capability, we need an implementation that supports an efficient search. We could use a list with sequential or binary search but it would be even better to use a hash table as described above since looking up an item in a hash table can approach $O(1)$ performance.

Below we use two lists to create a `HashTable` class that implements the Map abstract data type. One list, called `slots`, will hold the key items and a parallel list, called `data`, will hold the data values. When we look up a key, the corresponding position in the data list will hold the associated data value. We will treat the key list as a hash table using the ideas presented earlier. Note that the initial size for the hash table has been chosen to be 11. Although this is arbitrary, it is important that the size be a prime number so that the collision resolution algorithm can be as efficient as possible.

```
class HashTable:
   def __init__(self):
      self.size = 11
      self.slots = [None] * self.size
      self.data = [None] * self.size
```

hash_function implements the simple remainder method. The collision resolution technique is linear probing with a “plus 1” rehash function. The **put** function (see Listing 5.1) assumes that there will eventually be an empty slot unless the key is already present in the **self.slots**. It computes the original hash value and if that slot is not empty, iterates the **rehash** function until an empty slot occurs. If a nonempty slot already contains the key, the old data value is replaced with the new data value.

Listing 5.1: Functions to Place Items in the Hash Table

```
1  def put(self, key, data):
2    hash_value = self.hash_function(key,len(self.slots))
3
4    if self.slots[hash_value] == None:
5      self.slots[hash_value] = key
6      self.data[hash_value] = data
7    else:
8      if self.slots[hash_value] == key:
9        self.data[hash_value] = data #replace
10     else:
11       next_slot = self.rehash(hash_value, len(self.slots))
12       while self.slots[next_slot] != None and \
13                       self.slots[next_slot] != key:
14         next_slot = self.rehash(next_slot, len(self.slots))
15
16       if self.slots[next_slot] == None:
17         self.slots[next_slot] = key
18         self.data[next_slot] = data
19       else:
20         self.data[next_slot] = data #replace
21
22 def hash_function(self, key, size):
23      return key % size
24
25 def rehash(self, old_hash, size):
26     return (old_hash + 1) % size
```

Likewise, the **get** function begins by computing the initial hash value. If the value is not in the initial slot, **rehash** is used to locate the next possible position. Notice that line 15 guarantees that the search will terminate by checking to make sure that we have not returned to the initial slot. If that happens, we have exhausted all possible slots and the item must not be present.

The final methods of the **HashTable** class provide additional dictionary functionality. We overload the `__getitem__` and `__setitem__` methods to allow access using “**[]**.” This means that once a **HashTable** has been created, the familiar index operator will be available.

We leave the remaining methods as exercises.

```
def get(self, key):
  start_slot = self.hash_function(key, len(self.slots))

  data = None
  stop = False
  found = False
  position = start_slot
  while self.slots[position] != None and \
                  not found and not stop:
     if self.slots[position] == key:
       found = True
       data = self.data[position]
     else:
       position=self.rehash(position, len(self.slots))
       if position == start_slot:
          stop = True
  return data

def __getitem__(self, key):
   return self.get(key)

def __setitem__(self, key, data):
   self.put(key, data)
```

The following session shows the **HashTable** class in action. First we will create a hash table and store some items with integer keys and string data values.

```
>>> h=HashTable()
>>> h[54]="cat"
>>> h[26]="dog"
>>> h[93]="lion"
>>> h[17]="tiger"
>>> h[77]="bird"
>>> h[31]="cow"
>>> h[44]="goat"
>>> h[55]="pig"
>>> h[20]="chicken"
>>> h.slots
[77, 44, 55, 20, 26, 93, 17, None, None, 31, 54]
>>> h.data
['bird', 'goat', 'pig', 'chicken', 'dog', 'lion',
      'tiger', None, None, 'cow', 'cat']
>>>
```

Next we will access and modify some items in the hash table. Note that the value for the key 20 is being replaced.

```
>>> h[20]
'chicken'
>>> h[17]
'tiger'
>>> h[20]='duck'
>>> h[20]
'duck'
>>> h.data
['bird', 'goat', 'pig', 'duck', 'dog', 'lion',
      'tiger', None, None, 'cow', 'cat']
>> print(h[99])
None
```

Analysis of Hashing

We stated earlier that in the best case hashing would provide a $O(1)$, constant time search technique. However, due to collisions, the number of comparisons is typically not so simple. Even though a complete analysis of hashing is beyond the scope of this text, we can state some well-known results that approximate the number of comparisons necessary to search for an item.

The most important piece of information we need to analyze the use of a hash table is the load factor, λ. Conceptually, if λ is small, then there is a lower chance of collisions, meaning that items are more likely to be in the slots where they belong. If λ is large, meaning that the table is filling up, then there are more and more collisions. This means that collision resolution is more difficult, requiring more comparisons to find an empty slot. With chaining, increased collisions means an increased number of items on each chain.

As before, we will have a result for both a successful and an unsuccessful search. For a successful search using open addressing with linear probing, the average number of comparisons is approximately $\frac{1}{2}\left(1+\frac{1}{1-\lambda}\right)$ and an unsuccessful search gives $\frac{1}{2}\left(1+\left(\frac{1}{1-\lambda}\right)^2\right)$ If we are using chaining, the average number of comparisons is $1+\frac{\lambda}{2}$ for the successful case, and simply λ comparisons if the search is unsuccessful.

5.3 Sorting

Sorting is the process of placing elements from a collection in some kind of order. For example, a list of words could be sorted alphabetically or by length. A list of cities could be sorted by population, by area, or by zip code. We have already seen a number of algorithms that were able to benefit from having a sorted list (recall the final anagram example and the binary search).

There are many, many sorting algorithms that have been developed and analyzed. This suggests that sorting is an important area of study in computer science. Sorting a large number of items can take a substantial amount of computing resources. Like searching, the efficiency of a sorting algorithm is related to the number of items being processed. For small collections, a complex sorting method may be more trouble than it is worth. The overhead may be too high.

On the other hand, for larger collections, we want to take advantage of as many improvements as possible. In this section we will discuss several sorting techniques and compare them with respect to their running time.

Before getting into specific algorithms, we should think about the operations that can be used to analyze a sorting process. First, it will be necessary to compare two values to see which is smaller (or larger). In order to sort a collection, it will be necessary to have some systematic way to compare values to see if they are out of order. The total number of comparisons will be the most common way to measure a sort procedure. Second, when values are not in the correct position with respect to one another, it may be necessary to exchange them. This exchange is a costly operation and the total number of exchanges will also be important for evaluating the overall efficiency of the algorithm.

5.3.1 Bubble Sort

The **bubble sort** makes multiple passes through a list. It compares adjacent items and exchanges those that are out of order. Each pass through the list places the next largest value in its proper place. In essence, each item “bubbles” up to the location where it belongs.

Figure 5.13 shows the first pass of a bubble sort. The shaded items are being compared to see if they are out of order. If there are n items in the list, then there are $n - 1$ pairs of items that need to be compared on the first pass. It is important to note that once the largest value in the list is part of a pair, it will continually be moved along until the pass is complete.

At the start of the second pass, the largest value is now in place. There are $n - 1$ items left to sort, meaning that there will be $n - 2$ pairs. Since each pass places the next largest value in place, the total number of passes necessary will be $n - 1$. After completing the $n - 1$ passes, the smallest item must be in the correct position with no further processing required. The code below shows the complete `bubble_sort` function. It takes the list as a parameter, and modifies it by exchanging items as necessary.

```
def bubble_sort(a_list):
    for pass_num in range(len(a_list) - 1, 0, -1):
        for i in range(pass_num):
            if a_list[i] > a_list[i + 1]:
                temp = a_list[i]
                a_list[i] = a_list[i + 1]
                a_list[i + 1] = temp

a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
bubble_sort(a_list)
print(a_list)
```

The exchange operation, sometimes called a “swap,” is slightly different in Python than in most other programming languages. Typically, swapping two elements in a list requires a temporary storage location (an additional memory location). A code fragment such as

```
temp = a_list[i]
a_list[i] = a_list[j]
```

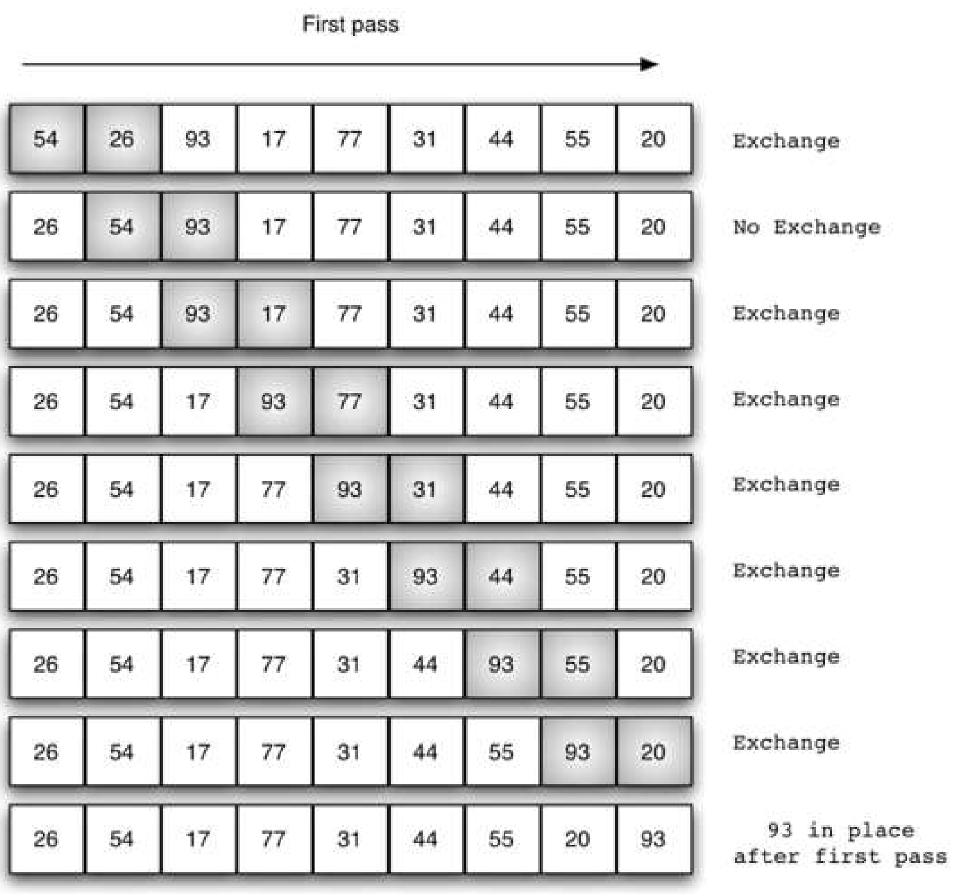

First pass
54 26 93 17 77 31 44 55 20 Exchange
26 54 93 17 77 31 44 55 20 No Exchange
26 54 93 17 77 31 44 55 20 Exchange
26 54 17 93 77 31 44 55 20 Exchange
26 54 17 77 93 31 44 55 20 Exchange
26 54 17 77 31 93 44 55 20 Exchange
26 54 17 77 31 44 93 55 20 Exchange
26 54 17 77 31 44 55 93 20 Exchange
26 54 17 77 31 44 55 20 93
93 in place
after first pass

Figure 5.13: Bubble sort: The First Pass

```
a_list[j] = temp
```

will exchange the i^{th} and j^{th} items in the list. Without the temporary storage, one of the values would be overwritten.

In Python, it is possible to perform simultaneous assignment. The statement `a, b = b, a` will result in two assignment statements being done at the same time (see Figure 5.14). Using simultaneous assignment, the exchange operation can be done in one statement.

Lines 5–7 in the `bubble_sort` function perform the exchange of the i and $(i+1)$th items using the three-step procedure described earlier. Note that we could also have used the simultaneous assignment to swap the items.

To analyze the bubble sort, we should note that regardless of how the items are arranged in the initial list, $n-1$ passes will be made to sort a list of size n. Table 5.6 shows the number of comparisons for each pass. The total number of comparisons is the sum of the first $n-1$ integers. Recall that the sum of the first n integers is $\frac{1}{2}n^2 + \frac{1}{2}n$. The sum of the first $n-1$ integers is $\frac{1}{2}n^2 + \frac{1}{2}n - n$, which is $\frac{1}{2}n^2 - \frac{1}{2}n$. This is still $O(n^2)$ comparisons. In the best case, if the list is already ordered, no exchanges will be made. However, in the worst case, every comparison will cause an exchange. On average, we exchange half of the time.

A bubble sort is often considered the most inefficient sorting method since it must exchange items before the final location is known. These “wasted” exchange operations are very costly. However, because the bubble sort makes passes through the entire unsorted portion of the list,

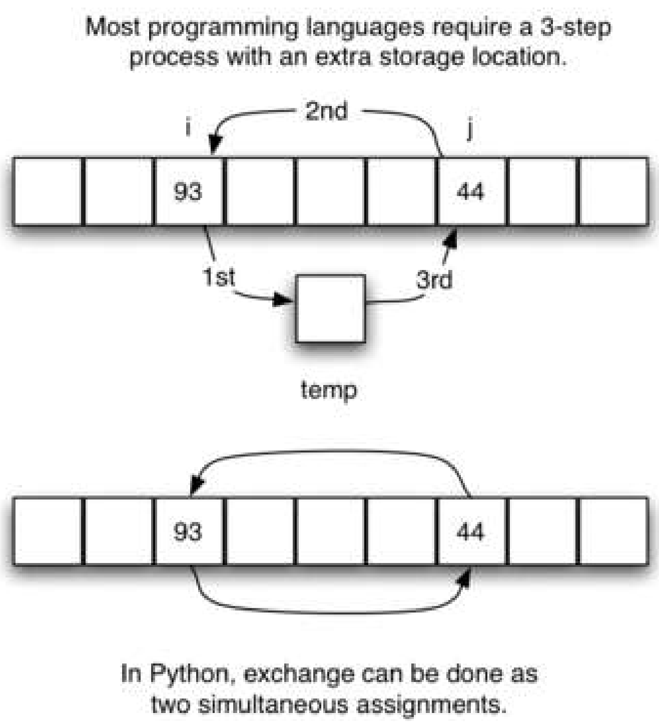

Most programming languages require a 3-step process with an extra storage location.
2nd
i
j
93
44
1st
3rd
temp
93
44
In Python, exchange can be done as two simultaneous assignments.

Figure 5.14: Exchanging Two Values in Python

Pass	**Comparisons**
1	$n-1$
2	$n-2$
3	$n-3$
...	...
$n-1$	1

Table 5.6: Comparisons for Each Pass of Bubble Sort

it has the capability to do something most sorting algorithms cannot. In particular, if during a pass there are no exchanges, then we know that the list must be sorted. A bubble sort can be modified to stop early if it finds that the list has become sorted. This means that for lists that require just a few passes, a bubble sort may have an advantage in that it will recognize the sorted list and stop.

The code below shows this modification, which is often referred to as the **short bubble**.

```
def short_bubble_sort(a_list):
   exchanges = True
   pass_num = len(a_list) - 1
   while pass_num > 0 and exchanges:
      exchanges = False
      for i in range(pass_num):
         if a_list[i] > a_list[i + 1]:
            exchanges = True
            temp = a_list[i]
            a_list[i] = a_list[i + 1]
            a_list[i + 1] = temp
```

```
        pass_num = pass_num - 1

a_list=[20, 30, 40, 90, 50, 60, 70, 80, 100, 110]
short_bubble_sort(a_list)
print(a_list)
```

Self Check

Suppose you have the following list of numbers to sort: $[19, 1, 9, 7, 3, 10, 13, 15, 8, 12]$ which list represents the partially sorted list after three complete passes of bubble sort?

1. $[1, 9, 19, 7, 3, 10, 13, 15, 8, 12]$
2. $[1, 3, 7, 9, 10, 8, 12, 13, 15, 19]$
3. $[1, 7, 3, 9, 10, 13, 8, 12, 15, 19]$
4. $[1, 9, 19, 7, 3, 10, 13, 15, 8, 12]$

5.3.2 Selection Sort

The **selection sort** improves on the bubble sort by making only one exchange for every pass through the list. In order to do this, a selection sort looks for the largest value as it makes a pass and, after completing the pass, places it in the proper location. As with a bubble sort, after the first pass, the largest item is in the correct place. After the second pass, the next largest is in place. This process continues and requires $n - 1$ passes to sort n items, since the final item must be in place after the $(n - 1)$st pass.

Figure 5.15 shows the entire sorting process. On each pass, the largest remaining item is selected and then placed in its proper location. The first pass places 93, the second pass places 77, the third places 55, and so on. The function is shown below.

```
def selection_sort(a_list):
    for fill_slot in range(len(a_list) - 1, 0, -1):
        pos_of_max = 0
        for location in range(1, fill_slot + 1):
            if a_list[location] > a_list[pos_of_max]:
                pos_of_max = location

        temp = a_list[fill_slot]
        a_list[fill_slot] = a_list[pos_of_max]
        a_list[pos_of_max] = temp

a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
selection_sort(a_list)
print(a_list)
```

You may see that the selection sort makes the same number of comparisons as the bubble sort and is therefore also $O(n^2)$. However, due to the reduction in the number of exchanges, the

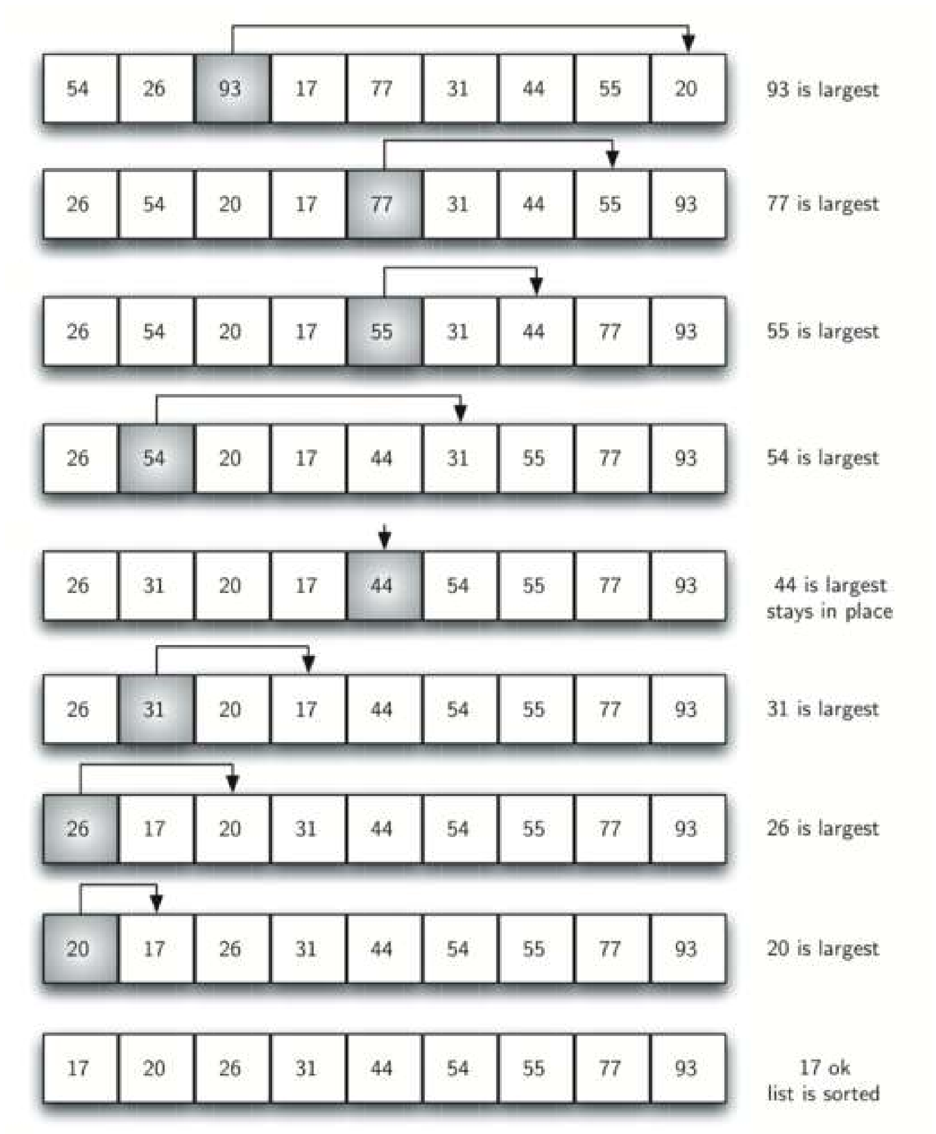


Figure 5.15: Selection Sort

selection sort typically executes faster in benchmark studies. In fact, for our list, the bubble sort makes 20 exchanges, while the selection sort makes only 8.

Self Check

Suppose you have the following list of numbers to sort: $[11, 7, 12, 14, 19, 1, 6, 18, 8, 20]$ which list represents the partially sorted list after three complete passes of selection sort?

1. $[7, 11, 12, 1, 6, 14, 8, 18, 19, 20]$
2. $[7, 11, 12, 14, 19, 1, 6, 18, 8, 20]$
3. $[11, 7, 12, 13, 1, 6, 8, 18, 19, 20]$
4. $[11, 7, 12, 14, 8, 1, 6, 18, 19, 20]$

5.3.3 The Insertion Sort

The **insertion sort**, although still $O(n^2)$, works in a slightly different way. It always maintains a sorted sublist in the lower positions of the list. Each new item is then "inserted" back into the previous sublist such that the sorted sublist is one item larger.

Figure 5.16 shows the insertion sorting process. The shaded items represent the ordered sublists as the algorithm makes each pass.

We begin by assuming that a list with one item (position 0) is already sorted. On each pass, one for each item 1 through $n - 1$, the current item is checked against those in the already sorted sublist. As we look back into the already sorted sublist, we shift those items that are greater to the right. When we reach a smaller item or the end of the sublist, the current item can be inserted.

Figure 5.17 shows the fifth pass in detail. At this point in the algorithm, a sorted sublist of five items consisting of $17, 26, 54, 77$, and 93 exists. We want to insert 31 back into the already sorted items. The first comparison against 93 causes 93 to be shifted to the right. 77 and 54 are also shifted. When the item 26 is encountered, the shifting process stops and 31 is placed in the open position. Now we have a sorted sublist of six items.

The implementation of insertion_sort shows that there are again $n - 1$ passes to sort n items. The iteration starts at position 1 and moves through position $n - 1$, as these are the items that need to be inserted back into the sorted sublists. Line 8 performs the shift operation that moves a value up one position in the list, making room behind it for the insertion. Remember that this is not a complete exchange as was performed in the previous algorithms.

The maximum number of comparisons for an insertion sort is the sum of the first $n-1$ integers. Again, this is $O(n^2)$. However, in the best case, only one comparison needs to be done on each pass. This would be the case for an already sorted list.

One note about shifting versus exchanging is also important. In general, a shift operation requires approximately a third of the processing work of an exchange since only one assignment is performed. In benchmark studies, insertion sort will show very good performance.

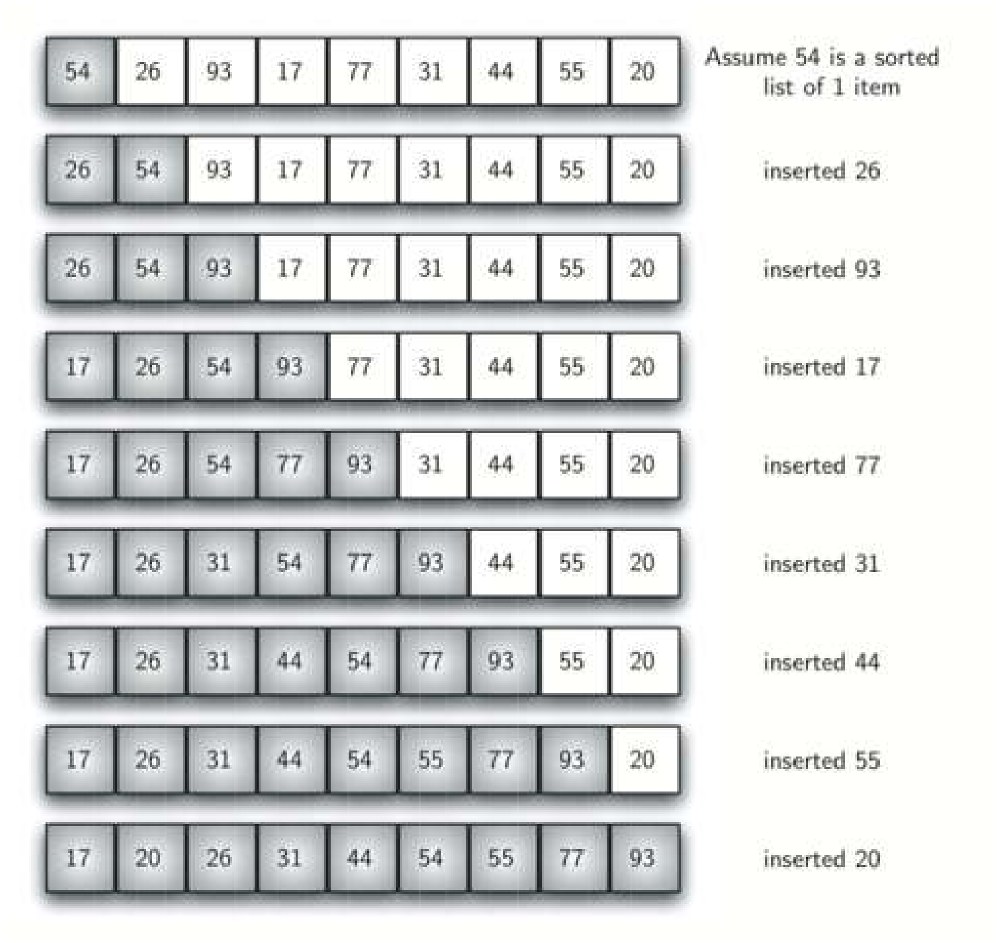
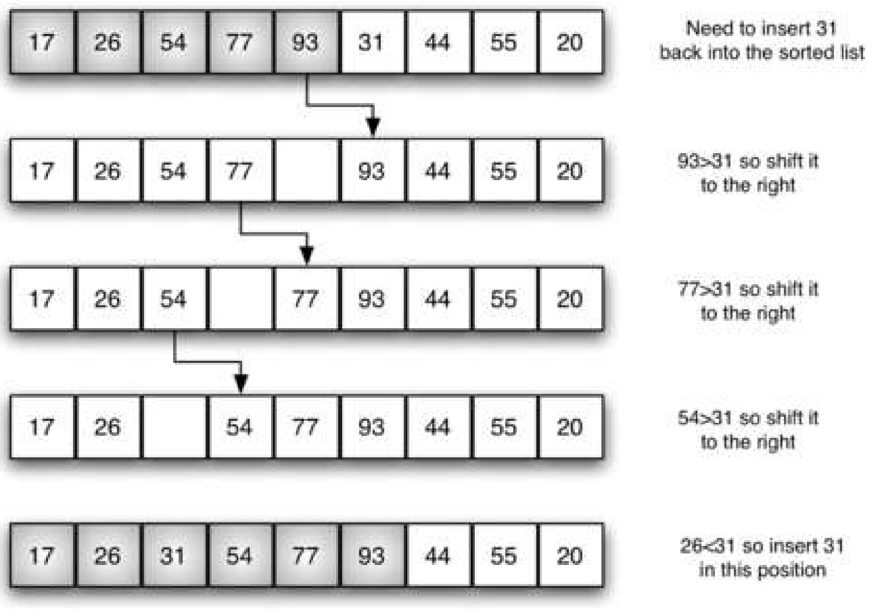


Figure 5.16: Insertion Sort

```
def insertion_sort(a_list):
   for index in range(1, len(a_list)):

     current_value = a_list[index]
     position = index

     while position > 0 and a_list[position - 1] > current_value:
        a_list[position] = a_list[position - 1]
        position = position - 1

     a_list[position] = current_value

a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
insertion_sort(a_list)
print(a_list)
```

Self Check

Suppose you have the following list of numbers to sort: $[15, 5, 4, 18, 12, 19, 14, 10, 8, 20]$ which list represents the partially sorted list after three complete passes of insertion sort?



Need to insert 31 back into the sorted list
93>31 so shift it to the right
77>31 so shift it to the right
54>31 so shift it to the right
26<31 so insert 31 in this position

Figure 5.17: Insertion Sort: Fifth Pass of the Sort

1. $[4, 5, 12, 15, 14, 10, 8, 18, 19, 20]$
2. $[15, 5, 4, 10, 12, 8, 14, 18, 19, 20]$
3. $[4, 5, 15, 18, 12, 19, 14, 10, 8, 20]$
4. $[15, 5, 4, 18, 12, 19, 14, 8, 10, 20]$

5.3.4 Shell Sort

The **shell sort**, sometimes called the “diminishing increment sort,” improves on the insertion sort by breaking the original list into a number of smaller sublists, each of which is sorted using an insertion sort. The unique way that these sublists are chosen is the key to the shell sort. Instead of breaking the list into sublists of contiguous items, the shell sort uses an increment `i`, sometimes called the **gap**, to create a sublist by choosing all items that are `i` items apart.

This can be seen in Figure 5.18. This list has nine items. If we use an increment of three, there are three sublists, each of which can be sorted by an insertion sort. After completing these sorts, we get the list shown in Figure 5.19. Although this list is not completely sorted, something very interesting has happened. By sorting the sublists, we have moved the items closer to where they actually belong.

Figure 5.20 shows a final insertion sort using an increment of one; in other words, a standard insertion sort. Note that by performing the earlier sublist sorts, we have now reduced the total number of shifting operations necessary to put the list in its final order. For this case, we need only four more shifts to complete the process.

We said earlier that the way in which the increments are chosen is the unique feature of the shell sort. The function `shell_sort` shown below uses a different set of increments. In this case, we begin with $\frac{n}{2}$ sublists. On the next pass, $\frac{n}{4}$ sublists are sorted. Eventually, a single list

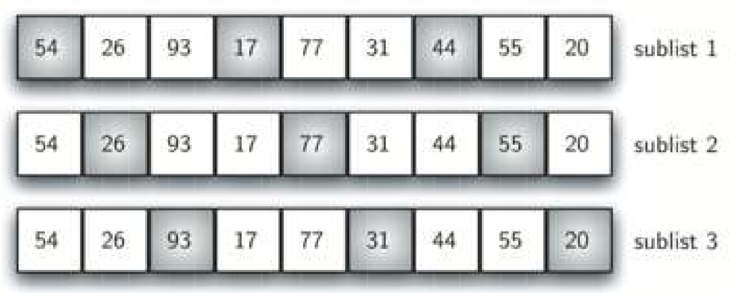


Figure 5.18: A Shell Sort with Increments of Three

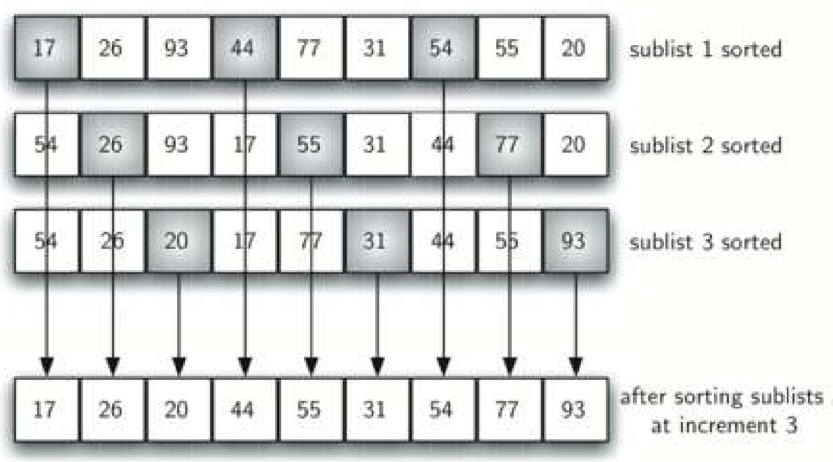


Figure 5.19: A Shell Sort after Sorting Each Sublist

is sorted with the basic insertion sort. Figure 5.21 shows the first sublists for our example using this increment.

The following invocation of the shell_sort function shows the partially sorted lists after each increment, with the final sort being an insertion sort with an increment of one.

```
def shell_sort(a_list):
   sublist_count = len(a_list) // 2
    while sublist_count > 0:
      for start_position in range(sublist_count):
         gap_insertion_sort(a_list, start_position, sublist_count)

      print("After increments of size", sublist_count, "The list is",
          a_list)

      sublist_count = sublist_count // 2

def gap_insertion_sort(a_list, start, gap):
   for i in range(start + gap, len(a_list), gap):
      current_value = a_list[i]
        position = i
```

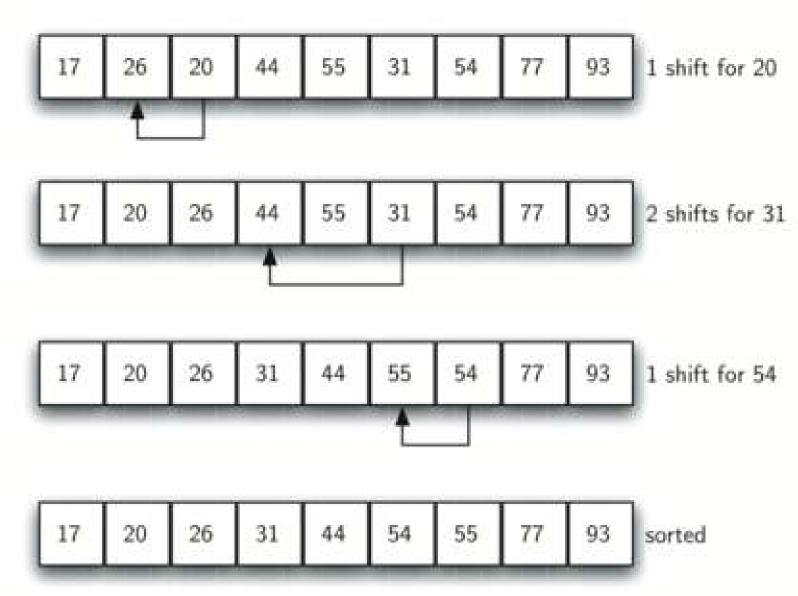


Figure 5.20: Shell Sort: A Final Insertion Sort with Increment of 1

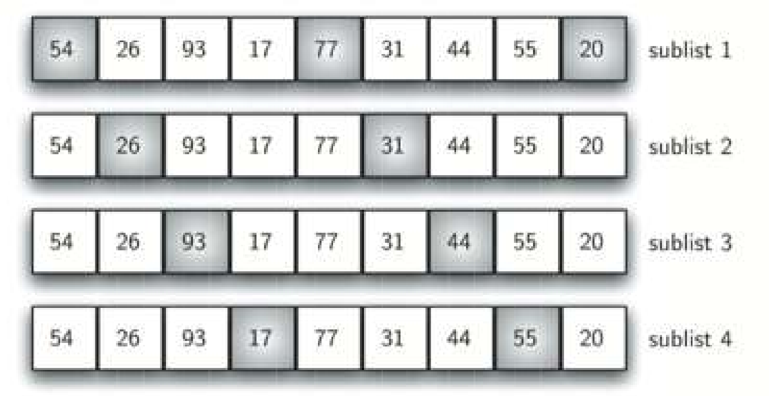


Figure 5.21: Initial Sublists for a Shell Sort

```
            while position >= gap and a_list[position - gap] >
               current_value:
             a_list[position] = a_list[position - gap]
               position = position - gap

            a_list[position] = current_value

a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
shell_sort(a_list)
print(a_list)
```

At first glance you may think that a shell sort cannot be better than an insertion sort, since it does a complete insertion sort as the last step. It turns out, however, that this final insertion sort does not need to do very many comparisons (or shifts) since the list has been pre-sorted by earlier incremental insertion sorts, as described above. In other words, each pass produces a list that is “more sorted” than the previous one. This makes the final pass very efficient.

Although a general analysis of the shell sort is well beyond the scope of this text, we can say that it tends to fall somewhere between $O(n)$ and $O(n^2)$, based on the behaviour described above. By changing the increment, for example using $2^k - 1$ $(1, 3, 7, 15, 31$, and so on), a shell sort can perform at $O(n^{\frac{3}{2}})$.

Self Check

Given the following list of numbers: $[5, 16, 20, 12, 3, 8, 9, 17, 19, 7]$ Which answer illustrates the contents of the list after all swapping is complete for a gap size of 3?

1. $[5, 3, 8, 7, 16, 19, 9, 17, 20, 12]$
2. $[3, 7, 5, 8, 9, 12, 19, 16, 20, 17]$
3. $[3, 5, 7, 8, 9, 12, 16, 17, 19, 20]$
4. $[5, 16, 20, 3, 8, 12, 9, 17, 20, 7]$

5.3.5 The Merge Sort

We now turn our attention to using a divide and conquer strategy as a way to improve the performance of sorting algorithms. The first algorithm we will study is the **merge sort**. Merge sort is a recursive algorithm that continually splits a list in half. If the list is empty or has one item, it is sorted by definition (the base case). If the list has more than one item, we split the list and recursively invoke a merge sort on both halves. Once the two halves are sorted, the fundamental operation, called a **merge**, is performed. Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list. Figure 5.22 shows our familiar example list as it is being split by `merge_sort`. Figure 5.23 shows the simple lists, now sorted, as they are merged back together.

The `merge_sort` function shown below begins by asking the base case question. If the length of the list is less than or equal to one, then we already have a sorted list and no more processing is necessary. If, on the other hand, the length is greater than one, then we use the Python `slice` operation to extract the left and right halves. It is important to note that the list may not have an even number of items. That does not matter, as the lengths will differ by at most one.

```python
def merge_sort(a_list):
   print("Splitting ", a_list)
   if len(a_list) > 1:
      mid = len(a_list) // 2
      left_half = a_list[:mid]
      right_half = a_list[mid:]

      merge_sort(left_half)
      merge_sort(right_half)

      i = 0
      j = 0
      k = 0
```

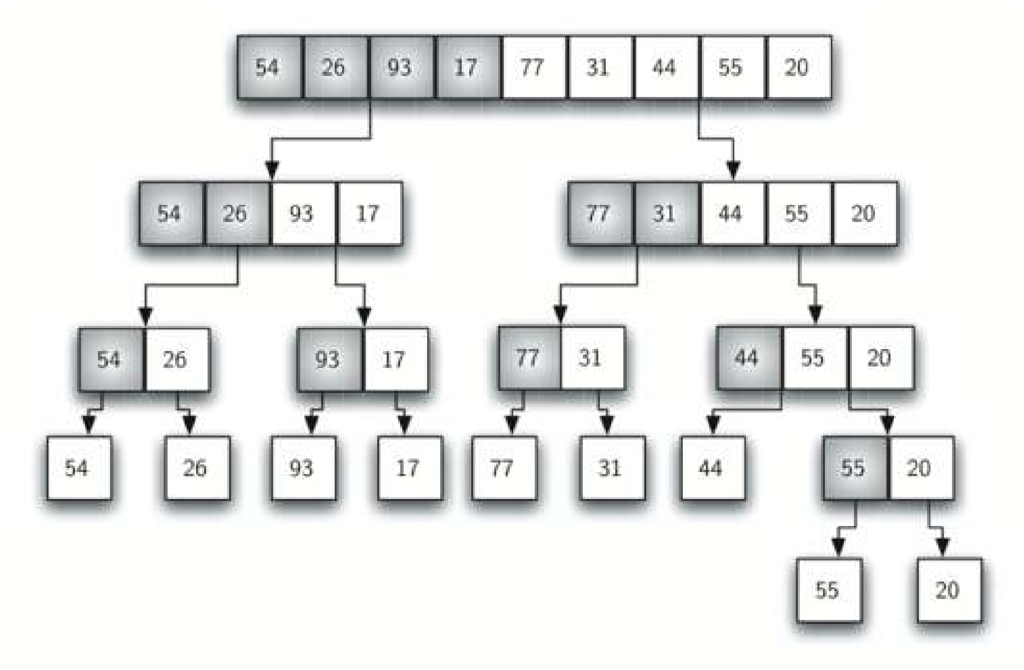


Figure 5.22: Splitting the List in a Merge Sort

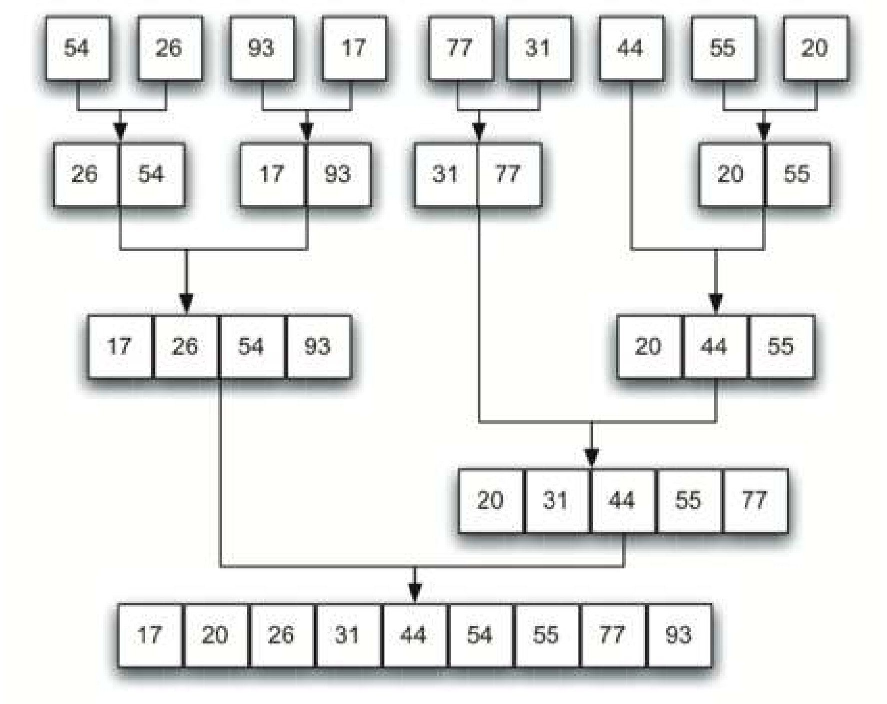


Figure 5.23: Lists as They Are Merged Together

```
14       while i < len(left_half) and j < len(right_half):
15          if left_half[i] < right_half[j]:
16             a_list[k] = left_half[i]
17             i = i + 1
18          else:
19             a_list[k] = right_half[j]
20             j = j + 1
21          k = k + 1
22
23       while i < len(left_half):
24          a_list[k] = left_half[i]
25          i = i + 1
26          k = k + 1
27
28       while j < len(right_half):
29          a_list[k] = right_half[j]
30          j = j + 1
31          k = k + 1
32    print("Merging ", a_list)
33
34 a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
35 merge_sort(a_list)
36 print(a_list)
```

Once the **merge_sort** function is invoked on the left half and the right half (lines 8–9), it is assumed they are sorted. The rest of the function (lines 11–31) is responsible for merging the two smaller sorted lists into a larger sorted list. Notice that the merge operation places the items back into the original list (**a_list**) one at a time by repeatedly taking the smallest item from the sorted lists.

The **merge_sort** function has been augmented with a **print** statement (line 2) to show the contents of the list being sorted at the start of each invocation. There is also a **print** statement (line 32) to show the merging process. The transcript shows the result of executing the function on our example list. Note that the list with 44, 55, and 20 will not divide evenly. The first split gives $[44]$ and the second gives $[55, 20]$. It is easy to see how the splitting process eventually yields a list that can be immediately merged with other sorted lists.

In order to analyze the **merge_sort** function, we need to consider the two distinct processes that make up its implementation. First, the list is split into halves. We already computed (in a binary search) that we can divide a list in half $\log n$ times where n is the length of the list. The second process is the merge. Each item in the list will eventually be processed and placed on the sorted list. So the merge operation which results in a list of size n requires n operations. The result of this analysis is that $\log n$ splits, each of which costs n for a total of $n \log n$ operations. A merge sort is an $O(n \log n)$ algorithm.

Recall that the slicing operator is $O(k)$ where k is the size of the slice. In order to guarantee that **merge_sort** will be $O(n \log n)$ we will need to remove the slice operator. Again, this is possible if we simply pass the starting and ending indices along with the list when we make the recursive call. We leave this as an exercise.

It is important to notice that the **merge_sort** function requires extra space to hold the two halves as they are extracted with the slicing operations. This additional space can be a critical factor if the list is large and can make this sort problematic when working on large data sets.

Self Check

Given the following list of numbers: $[21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40]$ which answer illustrates the list to be sorted after 3 recursive calls to mergesort?

1. $[16, 49, 39, 27, 43, 34, 46, 40]$
2. $[21, 1]$
3. $[21, 1, 26, 45]$
4. $[21]$

Given the following list of numbers: $[21, 1, 26, 45, 29, 28, 2, 9, 16, 49, 39, 27, 43, 34, 46, 40]$ which answer illustrates the first two lists to be merged?

1. $[21, 1]$ and $[26, 45]$
2. $[[1, 2, 9, 21, 26, 28, 29, 45]$ and $[16, 27, 34, 39, 40, 43, 46, 49]$
3. $[21]$ and $[1]$
4. $[9]$ and $[16]$

5.3.6 The Quick Sort

The **quick sort** uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage. As a trade-off, however, it is possible that the list may not be divided in half. When this happens, we will see that performance is diminished.

A quick sort first selects a value, which is called the **pivot value**. Although there are many different ways to choose the pivot value, we will simply use the first item in the list. The role of the pivot value is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list, commonly called the **split point**, will be used to divide the list for subsequent calls to the quick sort.

Figure 5.24 shows that 54 will serve as our first pivot value. Since we have looked at this example a few times already, we know that 54 will eventually end up in the position currently holding 31. The **partition** process will happen next. It will find the split point and at the same time move other items to the appropriate side of the list, either less than or greater than the pivot value.

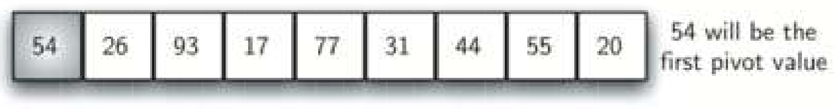


Figure 5.24: The First Pivot Value for a Quick Sort

Partitioning begins by locating two position markers – let's call them **left_mark** and **right_mark** – at the beginning and end of the remaining items in the list (positions 1 and 8 in Figure 5.25). The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while also converging on the split point. Figure 5.25 shows this process as we locate the position of 54.

We begin by incrementing **left_mark** until we locate a value that is greater than the pivot value. We then decrement **right_mark** until we find a value that is less than the pivot value. At this point we have discovered two items that are out of place with respect to the eventual split point. For our example, this occurs at 93 and 20. Now we can exchange these two items and then repeat the process again.

At the point where **right_mark** becomes less than **left_mark**, we stop. The position of **right_mark** is now the split point. The pivot value can be exchanged with the contents of the split point and the pivot value is now in place (Figure 5.26). In addition, all the items to the left of the split point are less than the pivot value, and all the items to the right of the split point are greater than the pivot value. The list can now be divided at the split point and the quick sort can be invoked recursively on the two halves.

The **quick_sort** function shown below invokes a recursive function, **quick_sort_helper**. **quick_sort_helper** begins with the same base case as the merge sort. If the length of the list is less than or equal to one, it is already sorted. If it is greater, then it can be partitioned and recursively sorted. The **partition** function implements the process described earlier.

```python
def quick_sort(a_list):
   quick_sort_helper(a_list, 0, len(a_list) - 1)

def quick_sort_helper(a_list, first, last):
   if first < last:

      split_point = partition(a_list, first, last)

      quick_sort_helper(a_list, first, split_point - 1)
      quick_sort_helper(a_list, split_point + 1, last)


def partition(a_list, first, last):
   pivot_value = a_list[first]

   left_mark = first + 1
   right_mark = last

   done = False
   while not done:

      while left_mark <= right_mark and \
            a_list[left_mark] <= pivot_value:
         left_mark = left_mark + 1

      while a_list[right_mark] >= pivot_value and \
            right_mark >= left_mark:
```

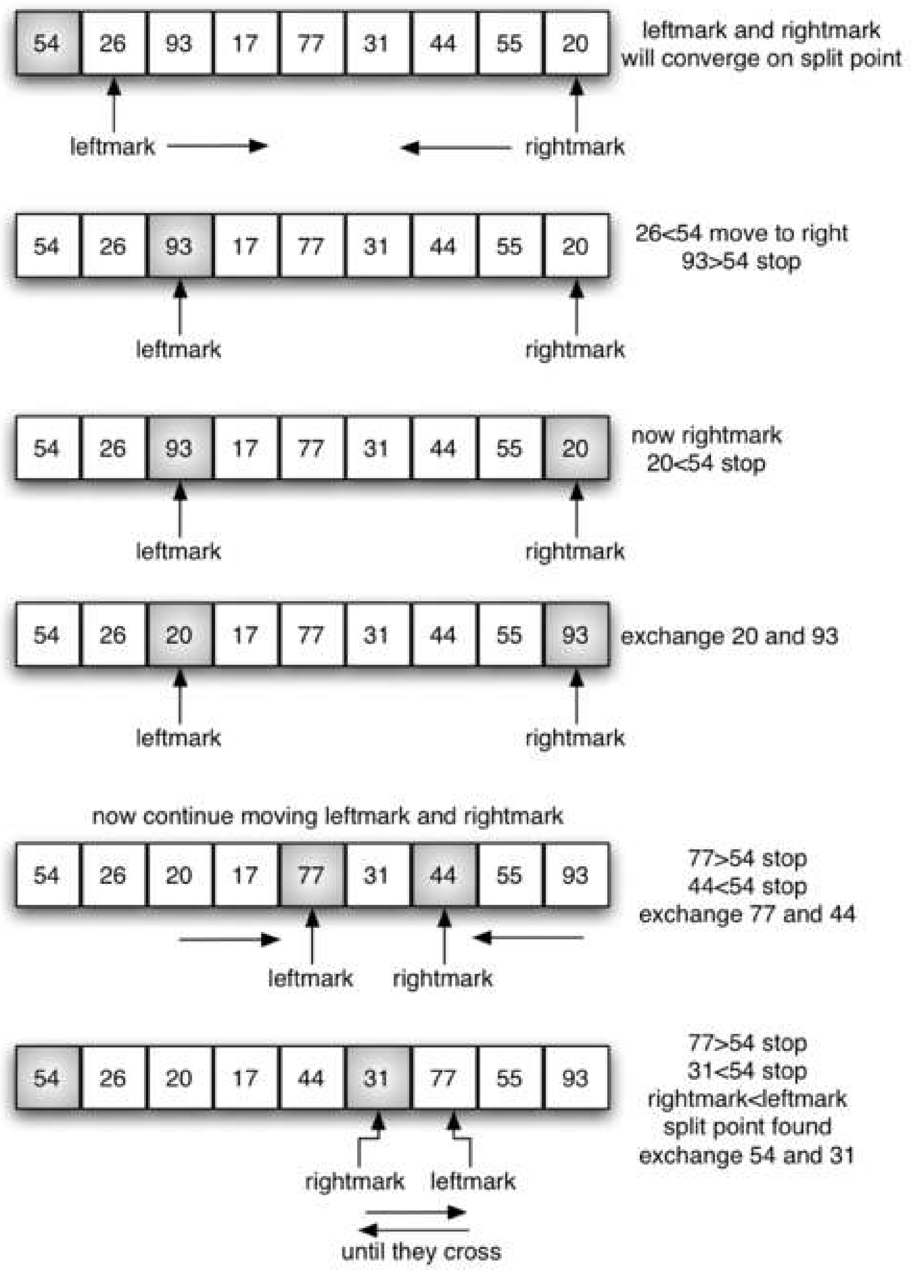


Figure 5.25: Finding the Split Point for 54

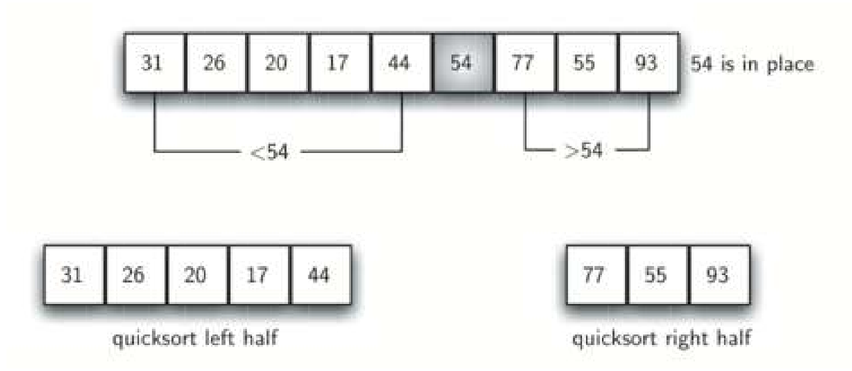


Figure 5.26: Completing the Partition Process to Find the Split Point for 54

```
                right_mark = right_mark - 1

            if right_mark < left_mark:
                done = True
            else:
                temp = a_list[left_mark]
                a_list[left_mark] = a_list[right_mark]
                a_list[right_mark] = temp

        temp = a_list[first]
        a_list[first] = a_list[right_mark]
        a_list[right_mark] = temp


        return right_mark

    a_list = [54, 26, 93, 17, 77, 31, 44, 55, 20]
    quick_sort(a_list)
    print(a_list)
```

To analyze the `quick_sort` function, note that for a list of length n, if the partition always occurs in the middle of the list, there will again be $\log n$ divisions. In order to find the split point, each of the n items needs to be checked against the pivot value. The result is $n \log n$. In addition, there is no need for additional memory as in the merge sort process.

Unfortunately, in the worst case, the split points may not be in the middle and can be very skewed to the left or the right, leaving a very uneven division. In this case, sorting a list of n items divides into sorting a list of 0 items and a list of $n - 1$ items. Then sorting a list of $n - 1$ divides into a list of size 0 and a list of size $n - 2$, and so on. The result is an $O(n^2)$ sort with all of the overhead that recursion requires.

We mentioned earlier that there are different ways to choose the pivot value. In particular, we can attempt to alleviate some of the potential for an uneven division by using a technique called **median of three**. To choose the pivot value, we will consider the first, the middle, and the last element in the list. In our example, those are 54, 77, and 20. Now pick the median value, in our case 54, and use it for the pivot value (of course, that was the pivot value we used originally). The idea is that in the case where the the first item in the list does not belong toward the middle of the list, the median of three will choose a better "middle" value. This will be particularly useful when the original list is somewhat sorted to begin with. We leave the implementation of this pivot value selection as an exercise.

Self Check

Given the following list of numbers $[14, 17, 13, 15, 19, 10, 3, 16, 9, 12]$ which answer shows the contents of the list after the second partitioning according to the quicksort algorithm?

1. $[9, 3, 10, 13, 12]$
2. $[9, 3, 10, 13, 12, 14]$
3. $[9, 3, 10, 13, 12, 14, 17, 16, 15, 19]$
4. $[9, 3, 10, 13, 12, 14, 19, 16, 15, 17]$

Given the following list of numbers $[1, 20, 11, 5, 2, 9, 16, 14, 13, 19]$ what would be the first pivot value using the median of 3 method?

1. 1
2. 9
3. 16
4. 19

Which of the following sort algorithms are guaranteed to be $O(n \log n)$ even in the worst case?

1. Shell Sort
2. Quick Sort
3. Merge Sort
4. Insertion Sort

5.4 Summary

- A sequential search is$O(n)$ for ordered and unordered lists.
- A binary search of an ordered list is $O(\log n)$ in the worst case.
- Hash tables can provide constant time searching.
- A bubble sort, a selection sort, and an insertion sort are $O(n^2)$ algorithms.
- A shell sort improves on the insertion sort by sorting incremental sublists. It falls between $O(n)$ and $O(n^2)$.
- A merge sort is $O(n \log n)$, but requires additional space for the merging process.
- A quick sort is $O(n \log n)$, but may degrade to $O(n^2)$ if the split points are not near the middle of the list. It does not require additional space

5.5 Key Terms

binary search	bubble sort	chaining
clustering	collision	collision resolution
folding method	gap	hash function
hash table	hashing	insertion sort
linear probing	load factor	map
median of three	merge	merge sort
mid-square method	open addressing	partition
perfect hash function	pivot value	quadratic probing
quick sort	rehashing	selection sort
sequential search	shell sort	short bubble
slot	split point	

5.6 Discussion Questions

1. Using the hash table performance formulas given in the chapter, compute the average number of comparisons necessary when the table is

   - 10% full
   - 25% full
   - 50% full
   - 75% full
   - 90% full
   - 99% full

   At what point do you think the hash table is too small? Explain.

2. Modify the hash function for strings to use positional weightings.

3. We used a hash function for strings that weighted the characters by position. Devise an alternative weighting scheme. What are the biases that exist with these functions?

4. Research perfect hash functions. Using a list of names (classmates, family members, etc.), generate the hash values using the perfect hash algorithm.

5. Generate a random list of integers. Show how this list is sorted by the following algorithms

   - bubble sort
   - selection sort
   - insertion sort
   - shell sort (you decide on the increments)
   - merge sort

- quick sort (you decide on the pivot value)

6. Consider the following list of integers: $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$. Show how this list is sorted by the following algorithms:
    - bubble sort
    - selection sort
    - insertion sort
    - shell sort (you decide on the increments)
    - merge sort
    - quick sort (you decide on the pivot value)
7. Consider the following list of integers: $[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]$. Show how this list is sorted by the following algorithms:
    - bubble sort
    - selection sort
    - insertion sort
    - shell sort (you decide on the increments)
    - merge sort
    - quick sort (you decide on the pivot value)
8. Consider the list of characters: $[\text{'P'}, \text{'Y'}, \text{'T'}, \text{'H'}, \text{'O'}, \text{'N'}]$. Show how this list is sorted using the following algorithms:
    - bubble sort
    - selection sort
    - insertion sort
    - shell sort (you decide on the increments)
    - merge sort
    - quick sort (you decide on the pivot value)
9. Devise alternative strategies for choosing the pivot value in quick sort. For example, pick the middle item. Re-implement the algorithm and then execute it on random data sets. Under what criteria does your new strategy perform better or worse than the strategy from this chapter?

5.7 Programming Exercises

1. Set up a random experiment to test the difference between a sequential search and a binary search on a list of integers. Use the binary search functions given in the text

(recursive and iterative). Generate a random, ordered list of integers and do a benchmark analysis for each one. What are your results? Can you explain them?

2. Implement the binary search using recursion without the slice operator. Recall that you will need to pass the list along with the starting and ending index values for the sublist. Generate a random, ordered list of integers and do a benchmark analysis.
3. Implement the **`len`** method (`__len__`) for the hash table Map ADT implementation.
4. Implement the **`in`** method (`__contains__`) for the hash table Map ADT implementation.
5. How can you delete items from a hash table that uses chaining for collision resolution? How about if open addressing is used? What are the special circumstances that must be handled? Implement the **`del`** method for the **`HashTable`** class.
6. In the hash table map implementation, the hash table size was chosen to be 101. If the table gets full, this needs to be increased. Re-implement the put method so that the table will automatically resize itself when the loading factor reaches a predetermined value (you can decide the value based on your assessment of load versus performance).
7. Implement quadratic probing as a rehash technique.
8. Using a random number generator, create a list of 500 integers. Perform a benchmark analysis using some of the sorting algorithms from this chapter. What is the difference in execution speed?
9. Implement the bubble sort using simultaneous assignment.
10. A bubble sort can be modified to "bubble" in both directions. The first pass moves "up" the list, and the second pass moves "down." This alternating pattern continues until no more passes are necessary. Implement this variation and describe under what circumstances it might be appropriate.
11. Implement the selection sort using simultaneous assignment.
12. Perform a benchmark analysis for a shell sort, using different increment sets on the same list.
13. Implement the **`merge_sort`** function without using the slice operator.
14. One way to improve the quick sort is to use an insertion sort on lists that have a small length (call it the "partition limit"). Why does this make sense? Re-implement the quick sort and use it to sort a random list of integers. Perform an analysis using different list sizes for the partition limit.
15. Implement the median-of-three method for selecting a pivot value as a modification to **`quick_sort`**. Run an experiment to compare the two techniques.

CHAPTER

SIX

TREES AND TREE ALGORITHMS

6.1 Objectives

- To understand what a tree data structure is and how it is used.
- To see how trees can be used to implement a map data structure.
- To implement trees using a list.
- To implement trees using classes and references.
- To implement trees as a recursive data structure.
- To implement a priority queue using a heap.

6.2 Examples Of Trees

Now that we have studied linear data structures like stacks and queues and have some experience with recursion, we will look at a common data structure called the **tree**. Trees are used in many areas of computer science, including operating systems, graphics, database systems, and computer networking. Tree data structures have many things in common with their botanical cousins. A tree data structure has a root, branches, and leaves. The difference between a tree in nature and a tree in computer science is that a tree data structure has its root at the top and its leaves on the bottom.

Notice that you can start at the top of the tree and follow a path made of circles and arrows all the way to the bottom. At each level of the tree we might ask ourselves a question and then follow the path that agrees with our answer. For example we might ask, “Is this animal a Chordate or an Arthropod?” If the answer is “Chordate” then we follow that path and ask, “Is this Chordate a Mammal?” If not, we are stuck (but only in this simplified example). When we are at the Mammal level we ask, “Is this Mammal a Primate or a Carnivore?” We can keep following paths until we get to the very bottom of the tree where we have the common name.

A second property of trees is that all of the children of one node are independent of the children of another node. For example, the Genus Felis has the children Domestica and Leo. The Genus Musca also has a child named Domestica, but it is a different node and is independent of the Domestica child of Felis. This means that we can change the node that is the child of Musca without affecting the child of Felis.

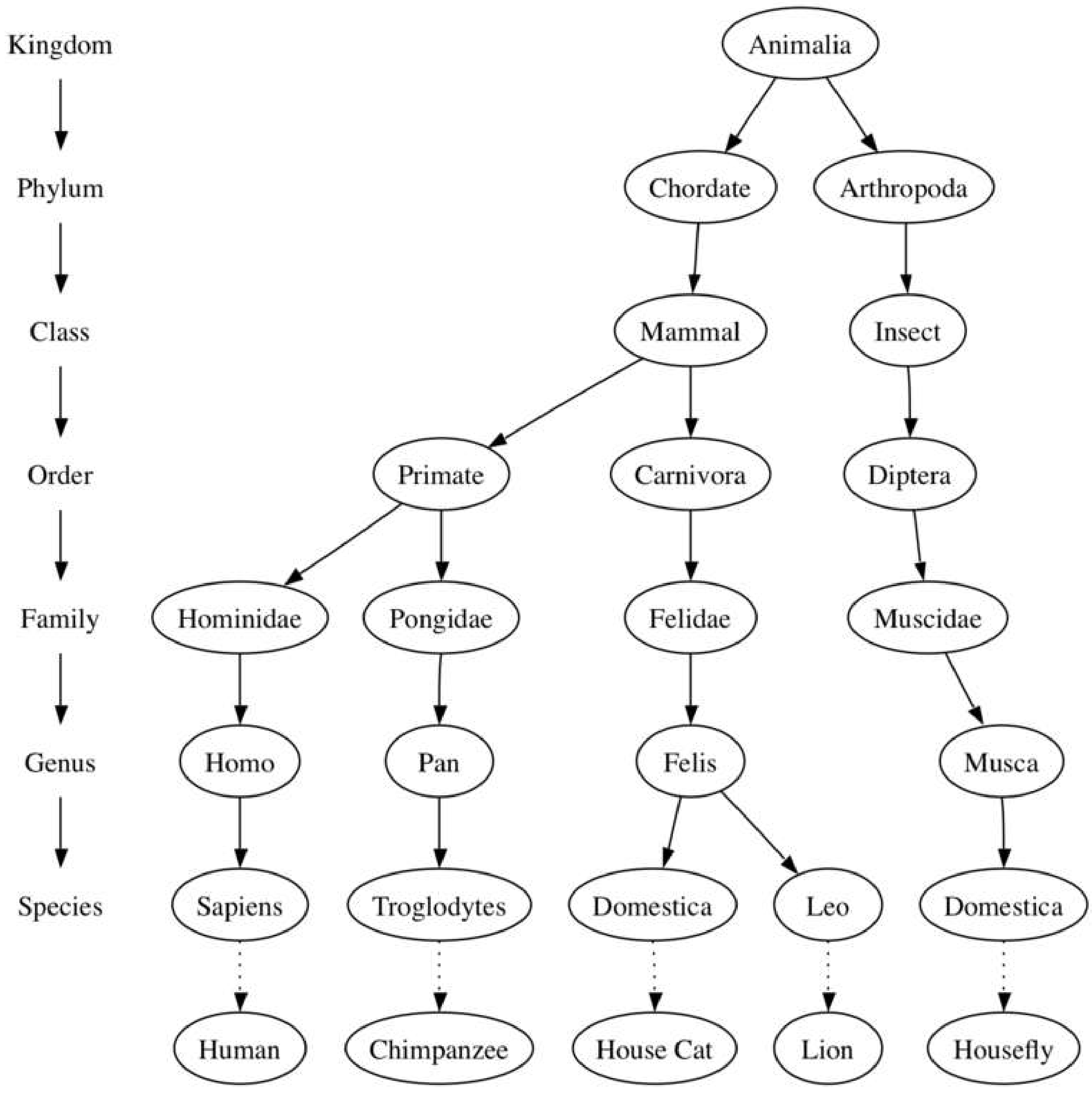


Figure 6.1: Taxonomy of Some Common Animals Shown as a Tree

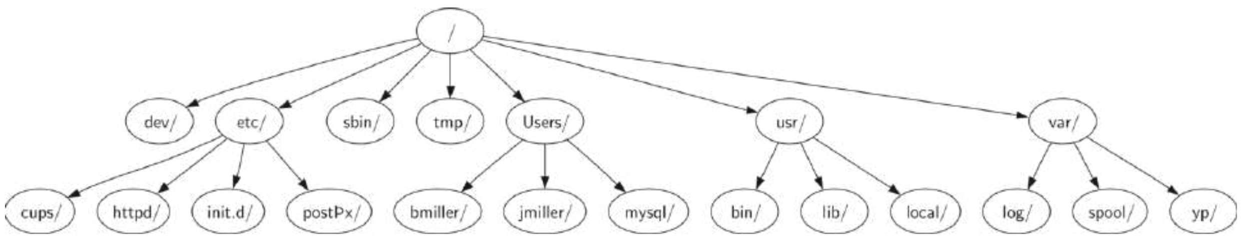


Figure 6.2: A Small Part of the Unix File System Hierarchy

A third property is that each leaf node is unique. We can specify a path from the root of the tree to a leaf that uniquely identifies each species in the animal kingdom; for example, Animalia $\rightarrow$ Chordate $\rightarrow$ Mammal $\rightarrow$ Carnivora $\rightarrow$ Felidae $\rightarrow$ Felis $\rightarrow$ Domestica.

Another example of a tree structure that you probably use every day is a file system. In a file system, directories, or folders, are structured as a tree. Figure 6.2 illustrates a small part of a Unix file system hierarchy.

The file system tree has much in common with the biological classification tree. You can follow a path from the root to any directory. That path will uniquely identify that subdirectory (and all the files in it). Another important property of trees, derived from their hierarchical nature, is that you can move entire sections of a tree (called a **subtree**) to a different position in the tree without affecting the lower levels of the hierarchy. For example, we could take the entire subtree staring with /etc/, detach etc/ from the root and reattach it under usr/. This would change the unique pathname to httpd from /etc/httpd to /usr/etc/httpd, but would not affect the contents or any children of the httpd directory.

A final example of a tree is a web page. The following is an example of a simple web page written using HTML. Figure 6.3 shows the tree that corresponds to each of the HTML tags used to create the page.

```
<html xmlns="http://www.w3.org/1999/xhtml"
     xml:lang="en" lang="en">
<head>
   <meta http-equiv="Content-Type"
        content="text/html; charset=utf-8" />
   <title>simple</title>
</head>
<body>
<h1>A simple web page</h1>
<ul>
   <li>List item one</li>
   <li>List item two</li>
</ul>
<h2><a href="http://www.cs.luther.edu">Luther CS </a><h2>
</body>
</html>
```

The HTML source code and the tree accompanying the source illustrate another hierarchy. Notice that each level of the tree corresponds to a level of nesting inside the HTML tags. The first tag in the source is `<html>` and the last is `</html>` All the rest of the tags in the page are

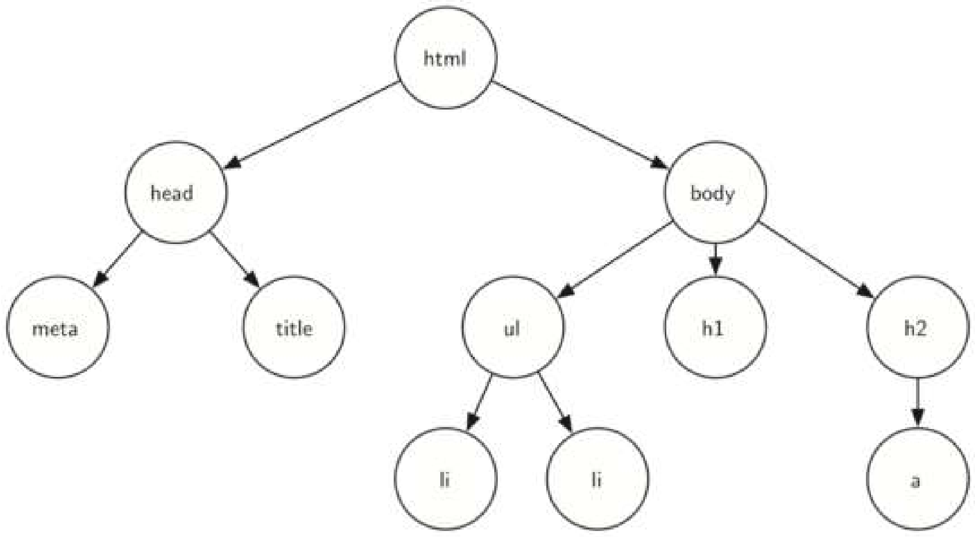


Figure 6.3: A Tree Corresponding to the Markup Elements of a Web Page

inside the pair. If you check, you will see that this nesting property is true at all levels of the tree.

6.3 Vocabulary and Definitions

6.3.1 Vocabulary

Node A node is a fundamental part of a tree. It can have a name, which we call the "key." A node may also have additional information. We call this additional information the "payload." While the payload information is not central to many tree algorithms, it is often critical in applications that make use of trees.

Edge An edge is another fundamental part of a tree. An edge connects two nodes to show that there is a relationship between them. Every node (except the root) is connected by exactly one incoming edge from another node. Each node may have several outgoing edges.

Root The root of the tree is the only node in the tree that has no incoming edges. In Figure 6.2, / is the root of the tree.

Path A path is an ordered list of nodes that are connected by edges. For example, Mammal $\rightarrow$ Carnivora $\rightarrow$ Felidae $\rightarrow$ Felis $\rightarrow$ Domestica is a path.

Children The set of nodes c that have incoming edges from the same node to are said to be the children of that node. In Figure 6.2, nodes log/, spool/, and yp/ are the children of node var/.

Parent A node is the parent of all the nodes it connects to with outgoing edges. In Figure 6.2 the node var/ is the parent of nodes log/, spool/, and yp/.

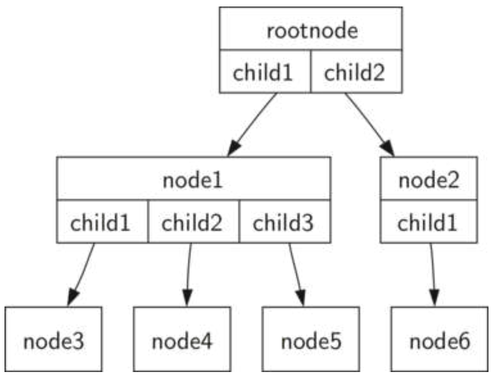


Figure 6.4: A Tree Consisting of a Set of Nodes and Edges

Sibling Nodes in the tree that are children of the same parent are said to be siblings. The nodes etc/ and usr/ are siblings in the filesystem tree.

Subtree A subtree is a set of nodes and edges comprised of a parent and all the descendants of that parent.

Leaf Node A leaf node is a node that has no children. For example, Human and Chimpanzee are leaf nodes in Figure 6.1.

Level The level of a node n is the number of edges on the path from the root node to n. For example, the level of the Felis node in Figure 6.1 is five. By definition, the level of the root node is zero.

Height The height of a tree is equal to the maximum level of any node in the tree. The height of the tree in Figure 6.2 is two.

6.3.2 Definitions

With the basic vocabulary now defined, we can move on to a formal definition of a tree. In fact, we will provide two definitions of a tree. One definition involves nodes and edges. The second definition, which will prove to be very useful, is a recursive definition.

Definition One A tree consists of a set of nodes and a set of edges that connect pairs of nodes. A tree has the following properties:

- One node of the tree is designated as the root node.
- Every node n, except the root node, is connected by an edge from exactly one other node p, where p is the parent of n.
- A unique path traverses from the root to each node.
- If each node in the tree has a maximum of two children, we say that the tree is a binary tree.

Figure 6.4 illustrates a tree that fits definition one. The arrowheads on the edges indicate the direction of the connection.

Definition Two A tree is either empty or consists of a root and zero or more subtrees, each of which is also a tree. The root of each subtree is connected to the root of the parent tree

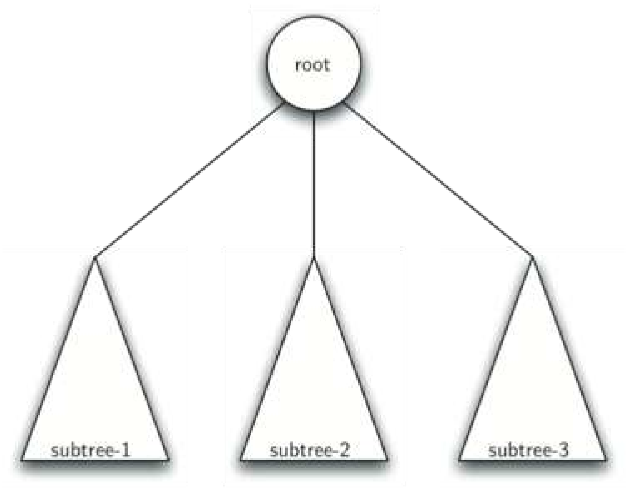


Figure 6.5: A recursive Definition of a tree

by an edge. Figure 6.5 illustrates this recursive definition of a tree. Using the recursive definition of a tree, we know that the tree in Figure 6.5 has at least four nodes, since each of the triangles representing a subtree must have a root. It may have many more nodes than that, but we do not know unless we look deeper into the tree.

6.4 Implementation

Keeping in mind the definitions from the previous section, we can use the following functions to create and manipulate a binary tree:

- `BinaryTree()` creates a new instance of a binary tree.
- `get_left_child()` returns the binary tree corresponding to the left child of the current node.
- `get_right_child()` returns the binary tree corresponding to the right child of the current node.
- `set_root_val(val)` stores the object in parameter val in the current node.
- `get_root_val()` returns the object stored in the current node.
- `insert_left(val)` creates a new binary tree and installs it as the left child of the current node.
- `insert_right(val)` creates a new binary tree and installs it as the right child of the current node.

The key decision in implementing a tree is choosing a good internal storage technique. Python allows us two very interesting possibilities, so we will examine both before choosing one. The first technique we will call “list of lists,” the second technique we will call “nodes and references.”

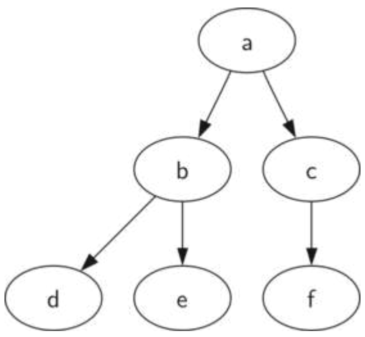


Figure 6.6: A Small Tree

6.4.1 List of Lists Representation

In a tree represented by a list of lists, we will begin with Python's list data structure and write the functions defined above. Although writing the interface as a set of operations on a list is a bit different from the other abstract data types we have implemented, it is interesting to do so because it provides us with a simple recursive data structure that we can look at and examine directly. In a list of lists tree, we will store the value of the root node as the first element of the list. The second element of the list will itself be a list that represents the left subtree. The third element of the list will be another list that represents the right subtree. To illustrate this storage technique, let's look at an example. Figure 6.6 shows a simple tree and the corresponding list implementation.

```
my_tree = ['a', #root
      ['b', #left subtree
       ['d' [], []],
       ['e' [], []] ],
      ['c', #right subtree
       ['f' [], []],
       [] ]
     ]
```

Notice that we can access subtrees of the list using standard list indexing. The root of the tree is **`my_tree[0]`**, the left subtree of the root is **`my_tree[1]`**, and the right subtree is **`my_tree[2]`**. The code below illustrates creating a simple tree using a list. Once the tree is constructed, we can access the root and the left and right subtrees. One very nice property of this list of lists approach is that the structure of a list representing a subtree adheres to the structure defined for a tree; the structure itself is recursive! A subtree that has a root value and two empty lists is a leaf node. Another nice feature of the list of lists approach is that it generalizes to a tree that has many subtrees. In the case where the tree is more than a binary tree, another subtree is just another list.

```
my_tree = ['a', ['b', ['d',[],[]], ['e',[],[]] ], ['c',
    ['f',[],[]], []] ]
print(my_tree)
print('left subtree = ', my_tree[1])
print('root = ', my_tree[0])
print('right subtree = ', my_tree[2])
```

Let us formalize this definition of the tree data structure by providing some functions that make it easy for us to use lists as trees. Note that we are not going to define a binary tree class. The functions we will write will just help us manipulate a standard list as though we are working with a tree.

```
def binary_tree(r):
   return [r, [], []]
```

The **`binary_tree`** function simply constructs a list with a root node and two empty sublists for the children. To add a left subtree to the root of a tree, we need to insert a new list into the second position of the root list. We must be careful. If the list already has something in the second position, we need to keep track of it and push it down the tree as the left child of the list we are adding. The code below shows the Python code for inserting a left child.

```
def insert_left(root, new_branch):
   t = root.pop(1)
   if len(t) > 1:
      root.insert(1, [new_branch, t, []])
   else:
      root.insert(1, [new_branch, [], []])
   return root
```

Notice that to insert a left child, we first obtain the (possibly empty) list that corresponds to the current left child. We then add the new left child, installing the old left child as the left child of the new one. This allows us to splice a new node into the tree at any position. The code for **`insert_right`** is similar to **`insert_left`** and is shown below.

```
def insert_right(root, new_branch):
   t = root.pop(2)
   if len(t) > 1:
      root.insert(2, [new_branch, [], t])
   else:
      root.insert(2, [new_branch, [], []])
   return root
```

To round out this set of tree-making functions let's write a couple of access functions for getting and setting the root value, as well as getting the left or right subtrees.

```
def get_root_val(root):
   return root[0]

def set_root_val(root,new_val):
   root[0] = new_val

def get_left_child(root):
   return root[1]
```

```python
def get_right_child(root):
    return root[2]
```

The following code exercises the tree functions we have just written. You should try it out for yourself. One of the exercises asks you to draw the tree structure resulting from this set of calls.

```python
def binary_tree(r):
    return [r, [], []]

def insert_left(root, new_branch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1, [new_branch, t, []])
    else:
        root.insert(1, [new_branch, [], []])
    return root

def insert_right(root, new_branch):
    t = root.pop(2)
    if len(t) > 1:
        root.insert(2, [new_branch, [], t])
    else:
        root.insert(2, [new_branch, [], []])
    return root

def get_root_val(root):
    return root[0]

def set_root_val(root, new_val):
    root[0] = new_val

def get_left_child(root):
    return root[1]

def get_right_child(root):
    return root[2]

r = binary_tree(3)
insert_left(r, 4)
insert_left(r, 5)
insert_right(r, 6)
insert_right(r, 7)
l = get_left_child(r)
print(l)

set_root_val(l, 9)
print(r)
insert_left(l, 11)
print(r)
print(get_right_child(get_right_child(r)))
```

Self Check

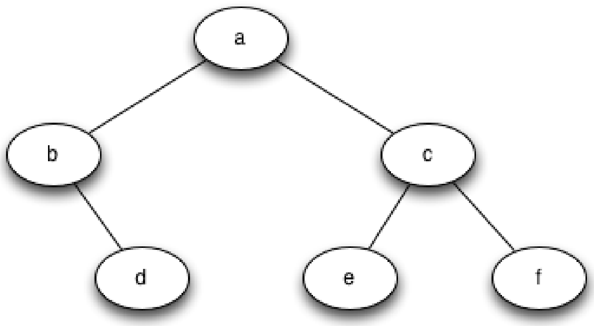
Given the following statements:

```
x = binary_tree('a')
insert_left(x,'b')
insert_right(x,'c')
insert_right(get_right_child(x), 'd')
insert_left(get_right_child(get_right_child(x)), 'e')
```

Which of the answers is the correct representation of the tree?

1. `['a', ['b', [], []], ['c', [], ['d', [], []]]]`
2. `['a', ['c', [], ['d', ['e', [], []], []]], ['b', [], []]]`
3. `['a', ['b', [], []], ['c', [], ['d', ['e', [], []], []]]]`
4. `['a', ['b', [], ['d', ['e', [], []], []]], ['c', [], []]]`

Write a function **`build_tree`** that returns a tree using the list of lists functions that looks like this:



6.4.2 Nodes and References

Our second method to represent a tree uses nodes and references. In this case we will define a class that has attributes for the root value, as well as the left and right subtrees. Since this representation more closely follows the object-oriented programming paradigm, we will continue to use this representation for the remainder of the chapter.

Using nodes and references, we might think of the tree as being structured like the one shown in 6.7.

We will start out with a simple class definition for the nodes and references approach as shown below. The important thing to remember about this representation is that the attributes **`left`** and **`right`** will become references to other instances of the **`BinaryTree`** class. For example, when we insert a new left child into the tree we create another instance of **`BinaryTree`** and

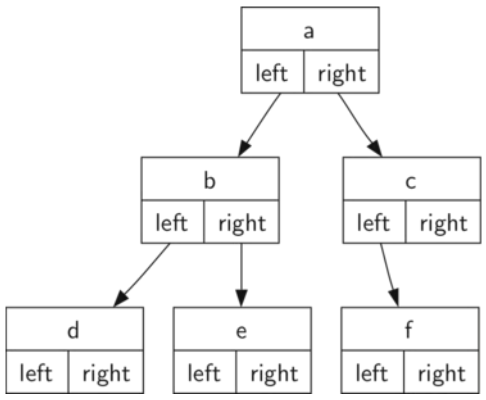


Figure 6.7: A Simple Tree Using a Nodes and References Approach

modify `self.left_child` in the root to reference the new tree.

```
class BinaryTree:
    def __init__(self, root):
        self.key = root
        self.left_child = None
        self.right_child = None
```

Notice that the constructor function expects to get some kind of object to store in the root. Just like you can store any object you like in a list, the root object of a tree can be a reference to any object. For our early examples, we will store the name of the node as the root value. Using nodes and references to represent the tree in Figure 6.7, we would create six instances of the BinaryTree class.

Next let's look at the functions we need to build the tree beyond the root node. To add a left child to the tree, we will create a new binary tree object and set the left attribute of the root to refer to this new object. The code for `insert_left` is shown below.

```
def insert_left(self,new_node):
    if self.left_child == None:
        self.left_child = BinaryTree(new_node)
    else:
        t = BinaryTree(new_node)
        t.left_child = self.left_child
        self.left_child = t
```

We must consider two cases for insertion. The first case is characterized by a node with no existing left child. When there is no left child, simply add a node to the tree. The second case is characterized by a node with an existing left child. In the second case, we insert a node and push the existing child down one level in the tree. The second case is handled by the `else` statement on line 4 of `insert_left`.

The code for `insert_right` must consider a symmetric set of cases. There will either be no right child, or we must insert the node between the root and an existing right child. The insertion code is shown below.

```
def insert_right(self,new_node):
   if self.right_child == None:
      self.right_child = BinaryTree(new_node)
   else:
      t = BinaryTree(new_node)
      t.right_child = self.right_child
      self.right_child = t
```

To round out the definition for a simple binary tree data structure, we will write accessor methods for the left and right children, as well as the root values.

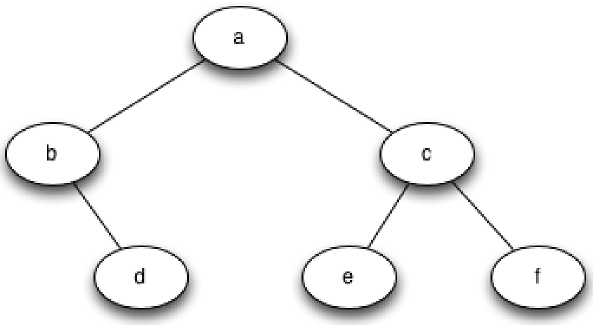
```
def get_right_child(self):
   return self.right_child

def get_left_child(self):
   return self.left_child

def set_root_val(self,obj):
   self.key = obj

def get_root_val(self):
   return self.key
```

Now that we have all the pieces to create and manipulate a binary tree, let's use them to check on the structure a bit more. Let's make a simple tree with node a as the root, and add nodes b and c as children. The code below creates the tree and looks at the some of the values stored in **`key`**, **`left`**, and **`right`**. Notice that both the left and right children of the root are themselves distinct instances of the BinaryTree class. As we said in our original recursive definition for a tree, this allows us to treat any child of a binary tree as a binary tree itself.

```
class BinaryTree:
   def __init__(self, root):
      self.key = root
      self.left_child = None
      self.right_child = None

   def insert_left(self, new_node):
      if self.left_child == None:
         self.left_child = BinaryTree(new_node)
      else:
         t = BinaryTree(new_node)
         t.left_child = self.left_child
         self.left_child = t

   def insert_right(self, new_node):
      if self.right_child == None:
         self.right_child = BinaryTree(new_node)
      else:
            t = BinaryTree(new_node)
            t.right_child = self.right_child
            self.right_child = t


    def get_right_child(self):
        return self.right_child

    def get_left_child(self):
        return self.left_child

    def set_root_val(self, obj):
        self.key = obj

    def get_root_val(self):
        return self.key


r = BinaryTree('a')
print(r.get_root_val())
print(r.get_left_child())
r.insert_left('b')
print(r.get_left_child())
print(r.get_left_child().get_root_val())
r.insert_right('c')
print(r.get_right_child())
print(r.get_right_child().get_root_val())
r.get_right_child().set_root_val('hello')
print(r.get_right_child().get_root_val())
```

Self Check

Write a function build_tree that returns a tree using the nodes and references implementation that looks like this:



6.5 Priority Queues with Binary Heaps

In earlier sections you learned about the first-in first-out data structure called a queue. One important variation of a queue is called a **priority queue**. A priority queue acts like a queue in that you dequeue an item by removing it from the front. However, in a priority queue the logical order of items inside a queue is determined by their priority. The highest priority items are at the front of the queue and the lowest priority items are at the back. Thus when you enqueue an item on a priority queue, the new item may move all the way to the front. We will see that the priority queue is a useful data structure for some of the graph algorithms we will study in the next chapter.

You can probably think of a couple of easy ways to implement a priority queue using sorting functions and lists. However, inserting into a list is $O(n)$ and sorting a list is $O(n \log n)$. We can do better. The classic way to implement a priority queue is using a data structure called a binary heap. A **binary heap** will allow us both enqueue and dequeue items in $O(\log n)$.

The binary heap is interesting to study because when we diagram the heap it looks a lot like a tree, but when we implement it we use only a single list as an internal representation. The binary heap has two common variations: the **min heap**, in which the smallest key is always at the front, and the **max heap**, in which the largest key value is always at the front. In this section we will implement the min heap. We leave a max heap implementation as an exercise.

6.5.1 Binary Heap Operations

The basic operations we will implement for our binary heap are as follows:

- `BinaryHeap()` creates a new, empty, binary heap.
- `insert(k)` adds a new item to the heap.
- `find_min()` returns the item with the minimum key value, leaving item in the heap.
- `del_min()` returns the item with the minimum key value, removing the item from the heap.
- `is_empty()` returns true if the heap is empty, false otherwise.
- `size()` returns the number of items in the heap.
- `build_heap(list)` builds a new heap from a list of keys.

The code below demonstrates the use of some of the binary heap methods. Notice that no matter the order that we add items to the heap, the smallest is removed each time. We will now turn our attention to creating an implementation for this idea.

```
>>> import BinHeap  # As defined below
>>> bh = BinHeap()
>>> bh.insert(5)
>>> bh.insert(7)
>>> bh.insert(3)
>>> bh.insert(11)
```

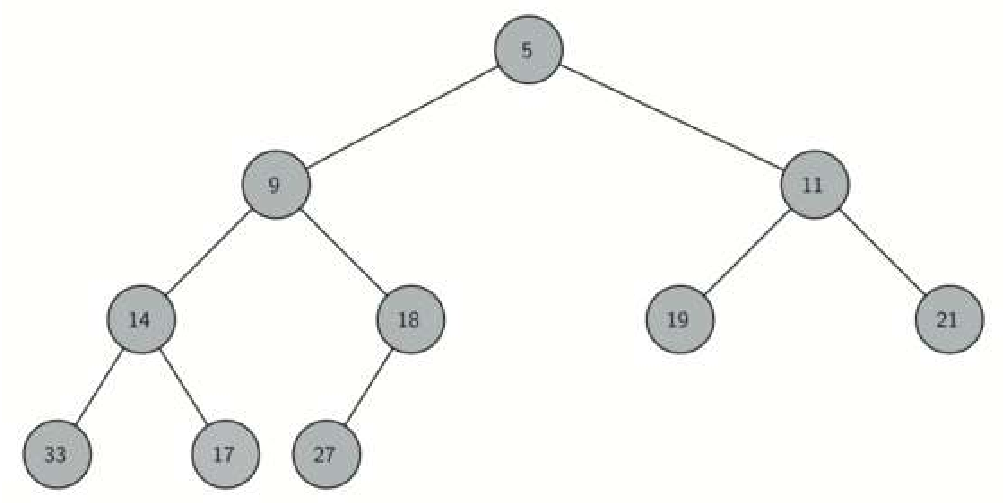


Figure 6.8: A Complete Binary Tree

```
>>> print(bh.del_min())
3
>>> print(bh.del_min())
5
>>> print(bh.del_min())
7
>>> print(bh.del_min())
11
>>>
```

6.5.2 Binary Heap Implementation

The Structure Property

In order to make our heap work efficiently, we will take advantage of the logarithmic nature of the binary tree to represent our heap. In order to guarantee logarithmic performance, we must keep our tree balanced. A balanced binary tree has roughly the same number of nodes in the left and right subtrees of the root. In our heap implementation we keep the tree balanced by creating a complete binary tree. A **complete binary tree** is a tree in which each level has all of its nodes. The exception to this is the bottom level of the tree, which we fill in from left to right. Figure 6.8 shows an example of a complete binary tree.

Another interesting property of a complete tree is that we can represent it using a single list. We do not need to use nodes and references or even lists of lists. Because the tree is complete, the left child of a parent (at position p) is the node that is found in position $2p$ in the list. Similarly, the right child of the parent is at position $2p + 1$ in the list. To find the parent of any node in the tree, we can simply use Python's integer division. Given that a node is at position n in the list, the parent is at position $n/2$. Figure 6.9 shows a complete binary tree and also gives the list representation of the tree. Note the $2p$ and $2p + 1$ relationship between parent and children. The list representation of the tree, along with the full structure property, allows us to efficiently

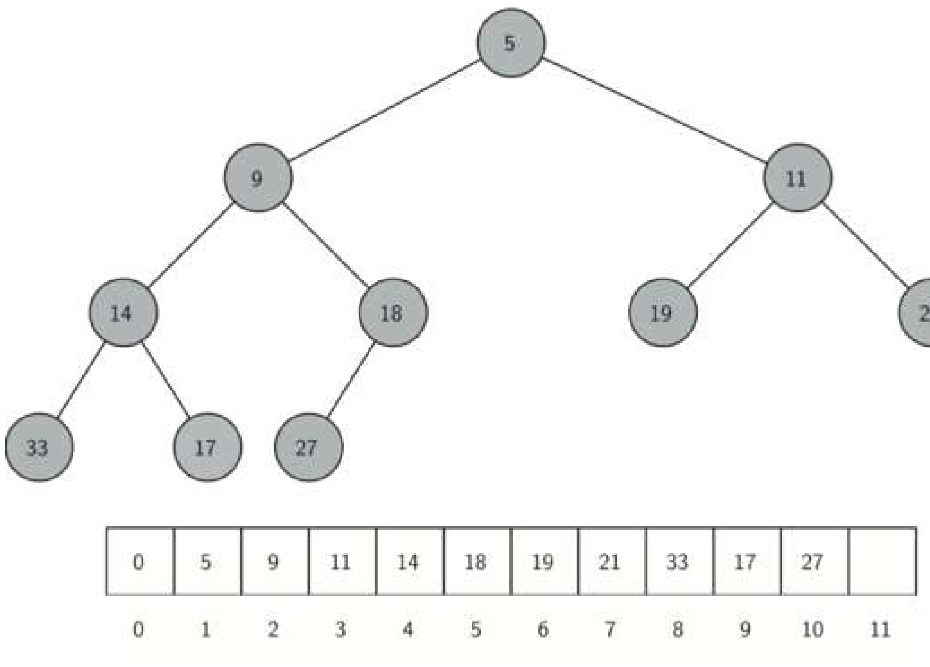


Figure 6.9: A Complete Binary Tree along with its List Representation

traverse a complete binary tree using only a few simple mathematical operations. We will see that this also leads to an efficient implementation of our binary heap.

The Heap Order Property

The method that we will use to store items in a heap relies on maintaining the heap order property. The **heap order property** is as follows: In a heap, for every node x with parent p, the key in p is smaller than or equal to the key in x. Figure 6.9 also illustrates a complete binary tree that has the heap order property.

Heap Operations

We will begin our implementation of a binary heap with the constructor. Since the entire binary heap can be represented by a single list, all the constructor will do is initialize the list and an attribute `current_size` to keep track of the current size of the heap. Below we show the Python code for the constructor. You will notice that an empty binary heap has a single zero as the first element of `heap_list` and that this zero is not used, but is there so that simple integer division can be used in later methods.

```
class BinHeap:
   def __init__(self):
      self.heap_list = [0]
      self.current_size = 0
```

The next method we will implement is `insert`. The easiest, and most efficient, way to add an item to a list is to simply append the item to the end of the list. The good news about appending is that it guarantees that we will maintain the complete tree property. The bad news about appending is that we will very likely violate the heap structure property. However, it is possible to write a method that will allow us to regain the heap structure property by comparing the newly added item with its parent. If the newly added item is less than its parent, then we can swap the item with its parent. Figure 6.10 shows the series of swaps needed to percolate the newly added item up to its proper position in the tree.

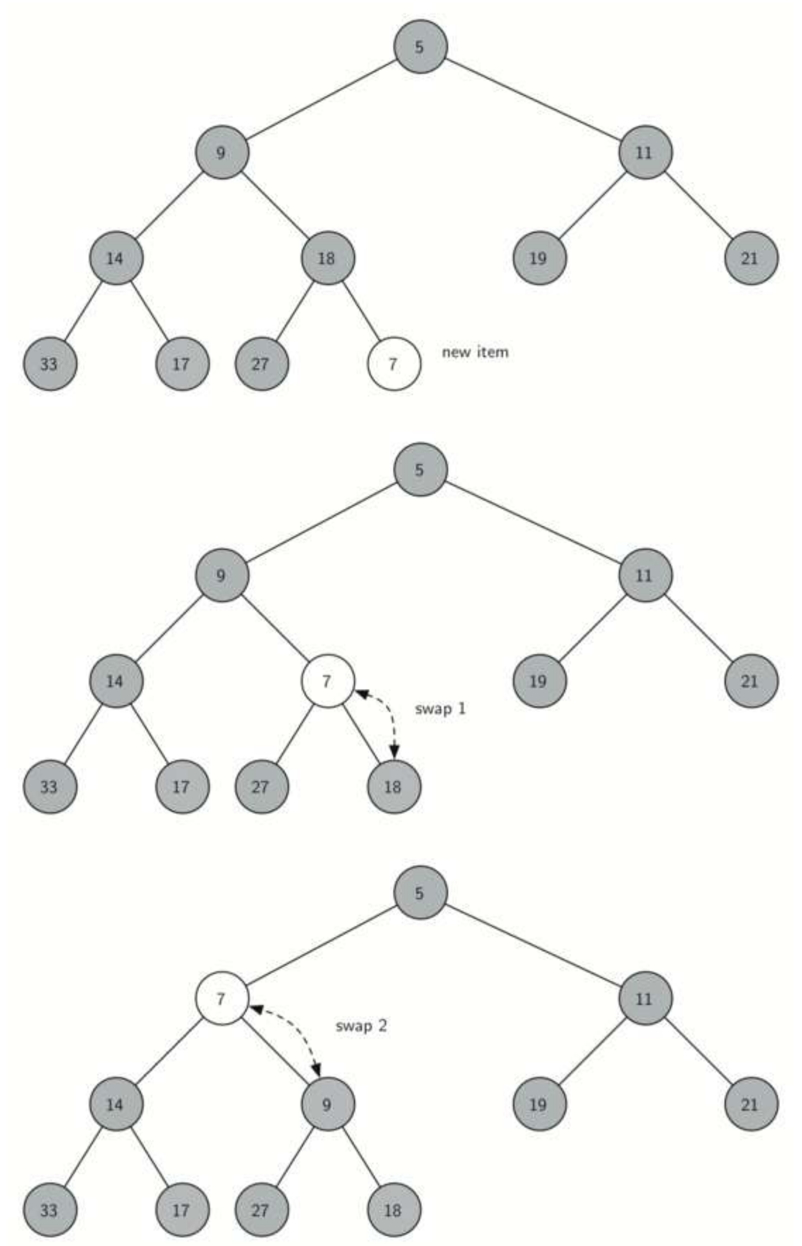


Figure 6.10: Percolate the New Node up to Its Proper Position

Notice that when we percolate an item up, we are restoring the heap property between the newly added item and the parent. We are also preserving the heap property for any siblings. Of course, if the newly added item is very small, we may still need to swap it up another level. In fact, we may need to keep swapping until we get to the top of the tree. Below we show the `perc_up` method, which percolates a new item as far up in the tree as it needs to go to maintain the heap property. Here is where our wasted element in `heap_list` is important.

Notice that we can compute the parent of any node by using simple integer division. The parent of the current node can be computed by dividing the index of the current node by 2.

```
def perc_up(self, i):
    while i // 2 > 0:
      if self.heap_list[i] < self.heap_list[i // 2]:
         tmp = self.heap_list[i // 2]
         self.heap_list[i // 2] = self.heap_list[i]
         self.heap_list[i] = tmp
      i = i // 2
```

We are now ready to write the **insert** method. Most of the work in the **insert** method is really done by **perc_up**. Once a new item is appended to the tree, perc_up takes over and positions the new item properly.

```
def insert(self, k):
    self.heap_list.append(k)
    self.current_size = self.current_size + 1
    self.perc_up(self.current_size)
```

With the **insert** method properly defined, we can now look at the **del_min** method. Since the heap property requires that the root of the tree be the smallest item in the tree, finding the minimum item is easy. The hard part of del_min is restoring full compliance with the heap structure and heap order properties after the root has been removed. We can restore our heap in two steps. First, we will restore the root item by taking the last item in the list and moving it to the root position. Moving the last item maintains our heap structure property. However, we have probably destroyed the heap order property of our binary heap. Second, we will restore the heap order property by pushing the new root node down the tree to its proper position. Figure 6.11 shows the series of swaps needed to move the new root node to its proper position in the heap.

In order to maintain the heap order property, all we need to do is swap the root with its smallest child less than the root. After the initial swap, we may repeat the swapping process with a node and its children until the node is swapped into a position on the tree where it is already less than both children. The code for percolating a node down the tree is found in the **perc_down** and **min_child** methods.

```
def perc_down(self, i):
    while (i * 2) <= self.current_size:
        mc = self.min_child(i)
        if self.heap_list[i] > self.heap_list[mc]:
            tmp = self.heap_list[i]
            self.heap_list[i] = self.heap_list[mc]
            self.heap_list[mc] = tmp
        i = mc

def min_child(self, i):
    if i * 2 + 1 > self.current_size:
        return i * 2
```

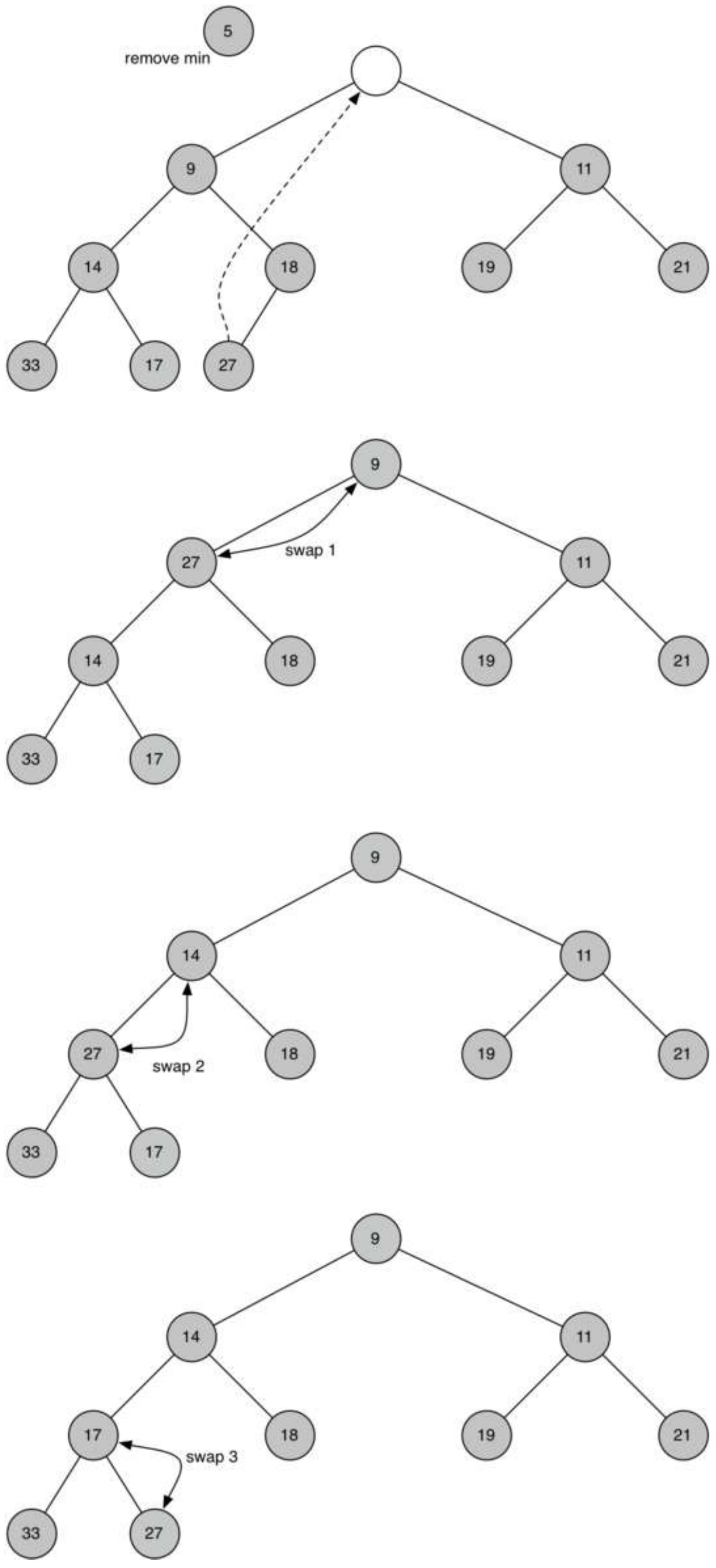


Figure 6.11: Percolating the Root Node down the Tree

```
    else:
        if self.heap_list[i * 2] < self.heap_list[i * 2 + 1]:
            return i * 2
        else:
            return i * 2 + 1
```

The code for the **del_min** operation is below. Note that once again the hard work is handled by a helper function, in this case **perc_down**.

```
def del_min(self):
    ret_val = self.heap_list[1]
    self.heap_list[1] = self.heap_list[self.current_size]
    self.current_size = self.current_size - 1
    self.heap_list.pop()
    self.perc_down(1)
    return ret_val
```

To finish our discussion of binary heaps, we will look at a method to build an entire heap from a list of keys. The first method you might think of may be like the following. Given a list of keys, you could easily build a heap by inserting each key one at a time. Since you are starting with a list of one item, the list is sorted and you could use binary search to find the right position to insert the next key at a cost of approximately $O(\log n)$ operations. However, remember that inserting an item in the middle of the list may require $O(n)$ operations to shift the rest of the list over to make room for the new key. Therefore, to insert n keys into the heap would require a total of $O(n \log n)$ operations. However, if we start with an entire list then we can build the whole heap in $O(n)$ operations. The **build_heap** function shows the code to build the entire heap.

```
def build_heap(self, a_list):
    i = len(a_list) // 2
    self.current_size = len(a_list)
    self.heap_list = [0] + a_list[:]
    while (i > 0):
        self.perc_down(i)
        i = i - 1
```

Figure 6.12 shows the swaps that the **build_heap** method makes as it moves the nodes in an initial tree of $[9, 6, 5, 2, 3]$ into their proper positions. Although we start out in the middle of the tree and work our way back toward the root, the **perc_down** method ensures that the largest child is always moved down the tree. Because the heap is a complete binary tree, any nodes past the halfway point will be leaves and therefore have no children. Notice that when $i = 1$, we are percolating down from the root of the tree, so this may require multiple swaps. As you can see in the rightmost two trees of Figure 6.12, first the 9 is moved out of the root position, but after 9 is moved down one level in the tree, **perc_down** ensures that we check the next set of children farther down in the tree to ensure that it is pushed as low as it can go. In this case it results in a second swap with 3. Now that 9 has been moved to the lowest level of the tree, no further swapping can be done. It is useful to compare the list representation of this series of swaps as shown in Figure 6.12 with the tree representation.

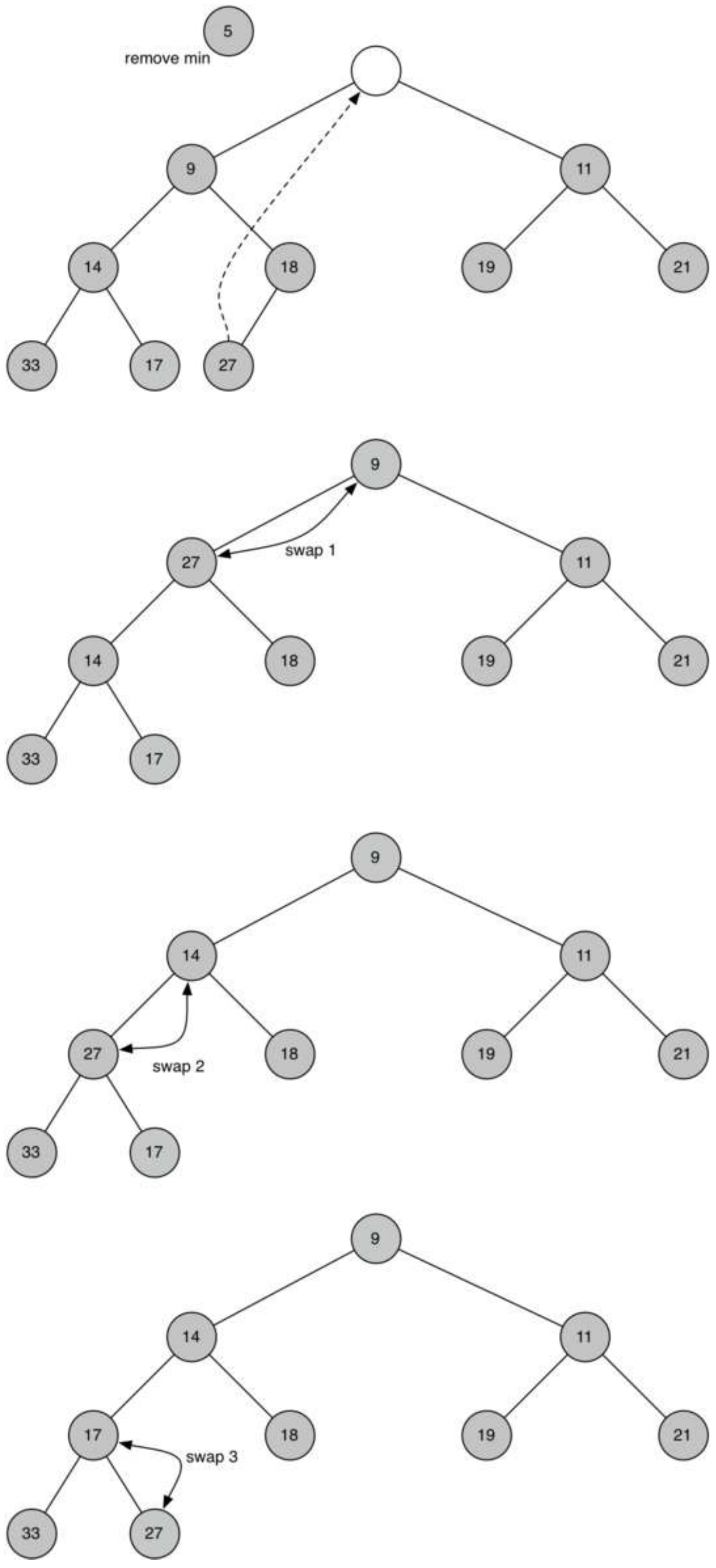


Figure 6.12: Building a Heap from the List $[9, 6, 5, 2, 3]$

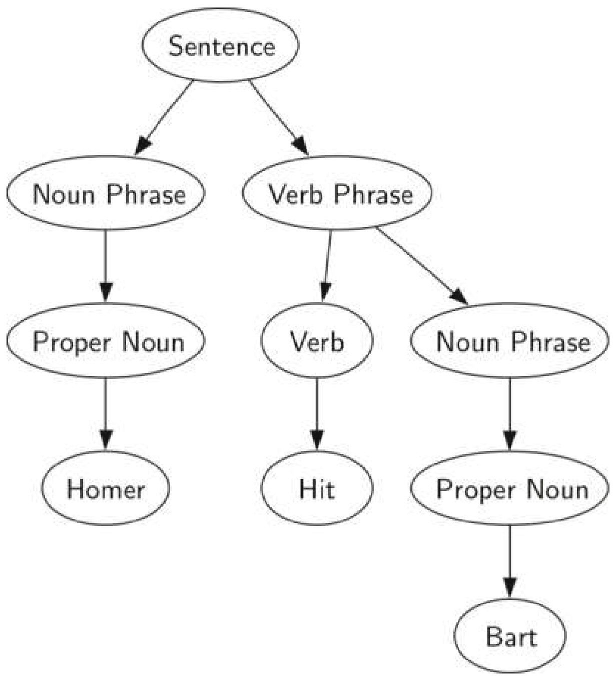


Figure 6.13: A Parse Tree for a Simple Sentence

The assertion that we can build the heap in $O(n)$ may seem a bit mysterious at first, and a proof is beyond the scope of this book. However, the key to understanding that you can build the heap in $O(n)$ is to remember that the $\log n$ factor is derived from the height of the tree. For most of the work in `build_heap`, the tree is shorter than $\log n$.

Using the fact that you can build a heap from a list in $O(n)$ time, you will construct a sorting algorithm that uses a heap and sorts a list in $O(n \log n)$ as an exercise at the end of this chapter.

6.6 Binary Tree Applications

With the implementation of our tree data structure complete, we now look at an example of how a tree can be used to solve some real problems. In this section we will look at parse trees. Parse trees can be used to represent real-world constructions like sentences or mathematical expressions.

Figure 6.13 shows the hierarchical structure of a simple sentence. Representing a sentence as a tree structure allows us to work with the individual parts of the sentence by using subtrees.

We can also represent a mathematical expression such as $((7 + 3) * (5 - 2))$ as a parse tree, as shown in Figure 6.14. We have already looked at fully parenthesized expressions, so what do we know about this expression? We know that multiplication has a higher precedence than either addition or subtraction. Because of the parentheses, we know that before we can do the multiplication we must evaluate the parenthesized addition and subtraction expressions. The hierarchy of the tree helps us understand the order of evaluation for the whole expression. Before we can evaluate the top-level multiplication, we must evaluate the addition and the subtraction in the subtrees. The addition, which is the left subtree, evaluates to 10. The subtraction, which is the right subtree, evaluates to 3. Using the hierarchical structure of trees, we can simply

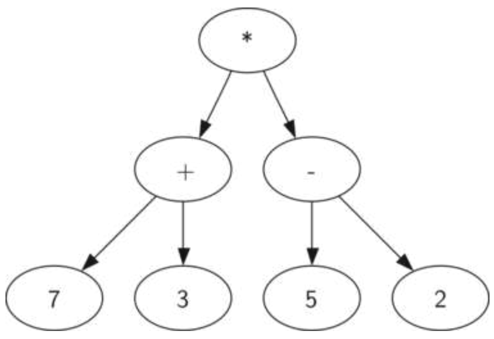


Figure 6.14: Parse Tree for $((7+3)*(5-2))$

replace an entire subtree with one node once we have evaluated the expressions in the children. Applying this replacement procedure gives us the simplified tree shown in

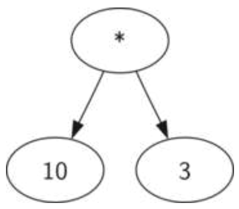


Figure 6.15: A Simplified Parse Tree for $((7+3)*(5-2))$

In the rest of this section we are going to examine parse trees in more detail. In particular we will look at

- How to build a parse tree from a fully parenthesized mathematical expression.
- How to evaluate the expression stored in a parse tree.
- How to recover the original mathematical expression from a parse tree.

The first step in building a parse tree is to break up the expression string into a list of tokens. There are four different kinds of tokens to consider: left parentheses, right parentheses, operators, and operands. We know that whenever we read a left parenthesis we are starting a new expression, and hence we should create a new tree to correspond to that expression. Conversely, whenever we read a right parenthesis, we have finished an expression. We also know that operands are going to be leaf nodes and children of their operators. Finally, we know that every operator is going to have both a left and a right child.

Using the information from above we can define four rules as follows:

1. If the current token is a '(', add a new node as the left child of the current node, and descend to the left child.
2. If the current token is in the list $[`+', `-', `/', `*']$, set the root value of the current node to the operator represented by the current token. Add a new node as the right child of the current node and descend to the right child.
3. If the current token is a number, set the root value of the current node to the number and return to the parent.
4. If the current token is a ')', go to the parent of the current node.

Before writing the Python code, let's look at an example of the rules outlined above in action. We will use the expression $(3 + (4 * 5))$. We will parse this expression into the following list of character tokens ['(', '3', '+', '(', '4', '*', '5' ,')', ')']. Initially we will start out with a parse tree that consists of an empty root node. Figure 6.16 illustrates the structure and contents of the parse tree, as each new token is processed.

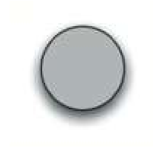
Using Figure 6.16 let's walk through the example step by step:

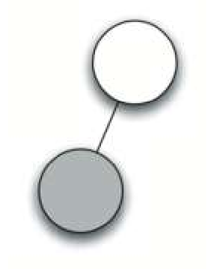
1. Create an empty tree.
2. Read (as the first token. By rule 1, create a new node as the left child of the root. Make the current node this new child.
3. Read 3 as the next token. By rule 3, set the root value of the current node to 3 and go back up the tree to the parent.
4. Read $+$ as the next token. By rule 2, set the root value of the current node to $+$ and add a new node as the right child. The new right child becomes the current node.
5. Read a (as the next token. By rule 1, create a new node as the left child of the current node. The new left child becomes the current node.
6. Read a 4 as the next token. By rule 3, set the value of the current node to 4. Make the parent of 4 the current node.
7. Read $*$ as the next token. By rule 2, set the root value of the current node to $*$ and create a new right child. The new right child becomes the current node.
8. Read 5 as the next token. By rule 3, set the root value of the current node to 5. Make the parent of 5 the current node.
9. Read) as the next token. By rule 4 we make the parent of $*$ the current node.
10. Read) as the next token. By rule 4 we make the parent of $+$ the current node. At this point there is no parent for $+$ so we are done.

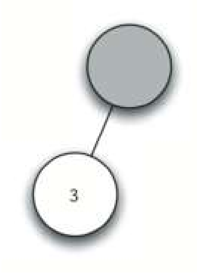
From the example above, it is clear that we need to keep track of the current node as well as the parent of the current node. The tree interface provides us with a way to get children of a node, through the `get_left_child` and `get_right_child` methods, but how can we keep track of the parent? A simple solution to keeping track of parents as we traverse the tree is to use a stack. Whenever we want to descend to a child of the current node, we first push the current node on the stack. When we want to return to the parent of the current node, we pop the parent off the stack.

Using the rules described above, along with the `Stack` and `BinaryTree` operations, we are now ready to write a Python function to create a parse tree. The code for our parse tree builder is presented below:

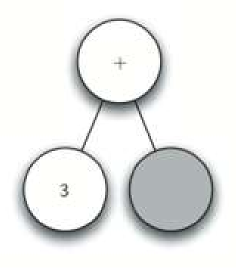
```python
def build_parse_tree(fp_exp):
    fp_list = fp_exp.split()
    p_stack = Stack()
    e_tree = BinaryTree('')
    p_stack.push(e_tree)
    current_tree = e_tree
    for i in fp_list:
```


(a)

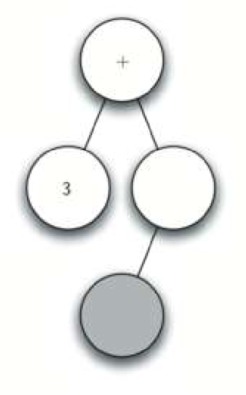

(b)



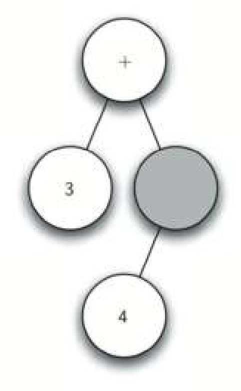
(c)



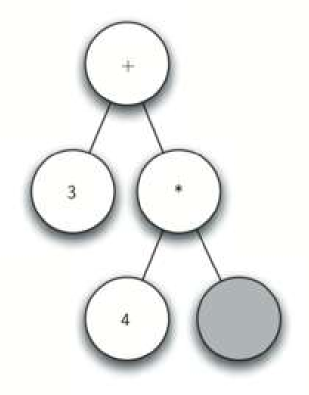
(d)



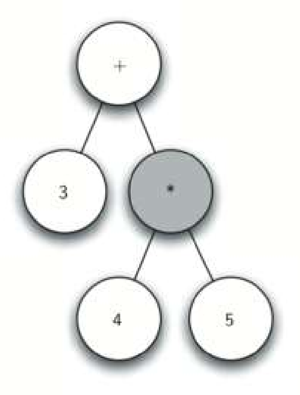
(e)



(f)



(g)



(h)

Figure 6.16: Tracing Parse Tree Construction

```
        if i == '(':
            current_tree.insert_left('')
            p_stack.push(current_tree)
            current_tree = current_tree.get_left_child()
        elif i not in ['+', '-', '*', '/', ')']:
            current_tree.set_root_val(int(i))
            parent = p_stack.pop()
            current_tree = parent
        elif i in ['+', '-', '*', '/']:
            current_tree.set_root_val(i)
            current_tree.insert_right('')
            p_stack.push(current_tree)
            current_tree = current_tree.get_right_child()
        elif i == ')':
            current_tree = p_stack.pop()
        else:
            raise ValueError
    return e_tree

pt = build_parse_tree("( ( 10 + 5 ) * 3 )")
pt.postorder() #defined and explained in the next section
```

The four rules for building a parse tree are coded as the first four clauses of the **`if`** statement on lines 11, 15, 19, and 24. In each case you can see that the code implements the rule, as described above, with a few calls to the **`BinaryTree`** or **`Stack`** methods. The only error checking we do in this function is in the else clause where we raise a **`ValueError`** exception if we get a token from the list that we do not recognize.

Now that we have built a parse tree, what can we do with it? As a first example, we will write a function to evaluate the parse tree, returning the numerical result. To write this function, we will make use of the hierarchical nature of the tree. Look back at Figure 6.14. Recall that we can replace the original tree with the simplified tree shown in Figure 6.15. This suggests that we can write an algorithm that evaluates a parse tree by recursively evaluating each subtree.

As we have done with past recursive algorithms, we will begin the design for the recursive evaluation function by identifying the base case. A natural base case for recursive algorithms that operate on trees is to check for a leaf node. In a parse tree, the leaf nodes will always be operands. Since numerical objects like integers and floating points require no further interpretation, the **`evaluate`** function can simply return the value stored in the leaf node. The recursive step that moves the function toward the base case is to call **`evaluate`** on both the left and the right children of the current node. The recursive call effectively moves us down the tree, toward a leaf node.

To put the results of the two recursive calls together, we can simply apply the operator stored in the parent node to the results returned from evaluating both children. In the example from Figure 6.15 we see that the two children of the root evaluate to themselves, namely 10 and 3. Applying the multiplication operator gives us a final result of 30.

The code for a recursive **`evaluate`** function is shown below. First, we obtain references to the left and the right children of the current node. If both the left and right children evaluate to **`None`**, then we know that the current node is really a leaf node. This check is on line 7. If the current node is not a leaf node, look up the operator in the current node and apply it to the results from recursively evaluating the left and right children.

To implement the arithmetic, we use a dictionary with the keys '$+$', '$-$', '$*$', and '$/$'. The values stored in the dictionary are functions from Python's operator module. The operator module provides us with the functional versions of many commonly used operators. When we look up an operator in the dictionary, the corresponding function object is retrieved. Since the retrieved object is a function, we can call it in the usual way `function(param1,param2)`. So the lookup `opers['+'](2,2)` is equivalent to `operator.add(2,2)`.

```
import operator
def evaluate(parse_tree):
    opers = {'+':operator.add, '-':operator.sub, '*':operator.mul,
        '/':operator.truediv}

    left = parse_tree.get_left_child()
    right = parse_tree.get_right_child()

    if left and right:
        fn = opers[parse_tree.get_root_val()]
        return fn(evaluate(left),evaluate(right))
    else:
        return parse_tree.get_root_val()
```

Finally, we will trace the evaluate function on the parse tree we created in Figure 6.16. When we first call evaluate, we pass the root of the entire tree as the parameter `parse_tree`. Then we obtain references to the left and right children to make sure they exist. The recursive call takes place on line 9. We begin by looking up the operator in the root of the tree, which is '$+$'. The '$+$' operator maps to the `operator.add` function call, which takes two parameters. As usual for a Python function call, the first thing Python does is to evaluate the parameters that are passed to the function. In this case both parameters are recursive function calls to our `evaluate` function. Using left-to-right evaluation, the first recursive call goes to the left. In the first recursive call the `evaluate` function is given the left subtree. We find that the node has no left or right children, so we are in a leaf node. When we are in a leaf node we just return the value stored in the leaf node as the result of the evaluation. In this case we return the integer 3.

At this point we have one parameter evaluated for our top-level call to `operator.add`. But we are not done yet. Continuing the left-to-right evaluation of the parameters, we now make a recursive call to evaluate the right child of the root. We find that the node has both a left and a right child so we look up the operator stored in this node, '$*$', and call this function using the left and right children as the parameters. At this point you can see that both recursive calls will be to leaf nodes, which will evaluate to the integers four and five respectively. With the two parameters evaluated, we return the result of `operator.mul(4, 5)`. At this point we have evaluated the operands for the top level '$+$' operator and all that is left to do is finish the call to `operator.add(3, 20)`. The result of the evaluation of the entire expression tree for $(3 + (4 * 5))$ is 23.

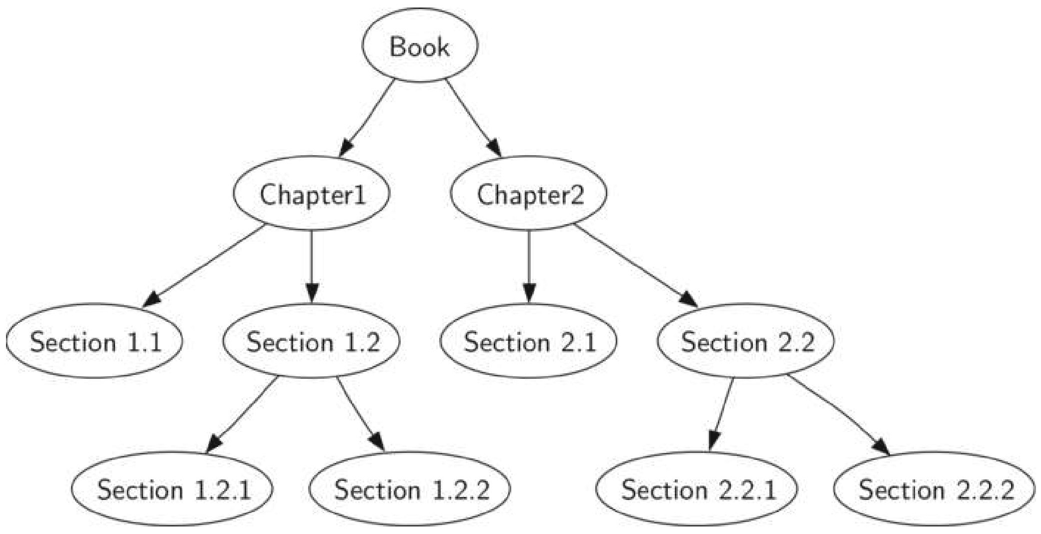


Figure 6.17: Representing a Book as a Tree

6.7 Tree Traversals

Now that we have examined the basic functionality of our tree data structure, it is time to look at some additional usage patterns for trees. These usage patterns can be divided into the three ways that we access the nodes of the tree. There are three commonly used patterns to visit all the nodes in a tree. The difference between these patterns is the order in which each node is visited. We call this visitation of the nodes a "traversal." The three traversals we will look at are called **preorder**, **inorder**, and **postorder**. Let's start out by defining these three traversals more carefully, then look at some examples where these patterns are useful.

preorder In a preorder traversal, we visit the root node first, then recursively do a preorder traversal of the left subtree, followed by a recursive preorder traversal of the right subtree.

inorder In an inorder traversal, we recursively do an inorder traversal on the left subtree, visit the root node, and finally do a recursive inorder traversal of the right subtree.

postorder In a postorder traversal, we recursively do a postorder traversal of the left subtree and the right subtree followed by a visit to the root node.

Let's look at some examples that illustrate each of these three kinds of traversals. First let's look at the preorder traversal. As an example of a tree to traverse, we will represent this book as a tree. The book is the root of the tree, and each chapter is a child of the root. Each section within a chapter is a child of the chapter, and each subsection is a child of its section, and so on. Figure 6.17 shows a limited version of a book with only two chapters. Note that the traversal algorithm works for trees with any number of children, but we will stick with binary trees for now.

Suppose that you wanted to read this book from front to back. The preorder traversal gives you exactly that ordering. Starting at the root of the tree (the Book node) we will follow the preorder traversal instructions. We recursively call `preorder` on the left child, in this case Chapter 1. We again recursively call `preorder` on the left child to get to Section 1.1. Since Section 1.1 has no children, we do not make any additional recursive calls. When we are finished with Section 1.1, we move up the tree to Chapter 1. At this point we still need to visit the right subtree of Chapter 1, which is Section 1.2. As before we visit the left subtree, which brings us to Section 1.2.1, then we visit the node for Section 1.2.2. With Section 1.2 finished, we return to Chapter 1. Then we return to the Book node and follow the same procedure for Chapter 2.

The code for writing tree traversals is surprisingly elegant, largely because the traversals are written recursively. Below we shows the Python code for a preorder traversal of a binary tree.

You may wonder, what is the best way to write an algorithm like preorder traversal? Should it be a function that simply uses a tree as a data structure, or should it be a method of the tree data structure itself? The code below shows a version of the preorder traversal written as an external function that takes a binary tree as a parameter. The external function is particularly elegant because our base case is simply to check if the tree exists. If the tree parameter is **`None`**, then the function returns without taking any action.

```
def preorder(tree):
    if tree:
        print(tree.get_root_val())
        preorder(tree.get_left_child())
        preorder(tree.get_right_child())
```

We can also implement **`preorder`** as a method of the **`BinaryTree`** class. The code for implementing preorder as an internal method is shown below. Notice what happens when we move the code from internal to external. In general, we just replace **`tree`** with **`self`**. However, we also need to modify the base case. The internal method must check for the existence of the left and the right children *before* making the recursive call to **`preorder`**.

```
def preorder(self):
    print(self.key)
    if self.left_child:
        self.left.preorder()
    if self.right_child:
        self.right.preorder()
```

Which of these two ways to implement **`preorder`** is best? The answer is that implementing **`preorder`** as an external function is probably better in this case. The reason is that you very rarely want to just traverse the tree. In most cases you are going to want to accomplish something else while using one of the basic traversal patterns. In fact, we will see in the next example that the **`postorder`** traversal pattern follows very closely with the code we wrote earlier to evaluate a parse tree. Therefore we will write the rest of the traversals as external functions.

The algorithm for the **`postorder`** traversal, shown below, is nearly identical to **`preorder`** except that we move the call to print to the end of the function.

```
def postorder(tree):
    if tree != None:
        postorder(tree.get_left_child())
        postorder(tree.get_right_child())
        print(tree.get_root_val())
```

We have already seen a common use for the postorder traversal, namely evaluating a parse tree. Look back at our `evaluate` function above. What we are doing is evaluating the left subtree, evaluating the right subtree, and combining them in the root through the function call to an operator. Assume that our binary tree is going to store only expression tree data. Let's rewrite the evaluation function, but model it even more closely on the postorder code.

```
def postorder_eval(tree):
    opers = {'+':operator.add, '-':operator.sub, '*':operator.mul,
        '/':operator.truediv}
    res1 = None
    res2 = None
    if tree:
        res1 = postorder_eval(tree.get_left_child())
        res2 = postorder_eval(tree.get_right_child())
        if res1 and res2:
            return opers[tree.get_root_val()](res1, res2)
        else:
            return tree.get_root_val()
```

Notice that the form in `postorder` is the same as the form in `postorder_eval`, except that instead of printing the key at the end of the function, we return it. This allows us to save the values returned from the recursive calls in lines 6 and 7. We then use these saved values along with the operator on line 9.

The final traversal we will look at in this section is the inorder traversal. In the inorder traversal we visit the left subtree, followed by the root, and finally the right subtree. Below we show our code for the inorder traversal. Notice that in all three of the traversal functions we are simply changing the position of the `print` statement with respect to the two recursive function calls.

```
def inorder(tree):
  if tree != None:
      inorder(tree.get_left_child())
      print(tree.get_root_val())
      inorder(tree.get_right_child())
```

If we perform a simple inorder traversal of a parse tree we get our original expression back, without any parentheses. Let's modify the basic inorder algorithm to allow us to recover the fully parenthesized version of the expression. The only modifications we will make to the basic template are as follows: print a left parenthesis before the recursive call to the left subtree, and print a right parenthesis after the recursive call to the right subtree.

```
def print_exp(tree):
  str_val = ""
  if tree:
      str_val = '(' + print_exp(tree.get_left_child())
      str_val = str_val + str(tree.get_root_val())
      str_val = str_val + print_exp(tree.get_right_child()) + ')'
  return str_val
```

Notice that the `print_exp` function as we have implemented it puts parentheses around each number. While not incorrect, the parentheses are clearly not needed. In the exercises at the end of this chapter you are asked to modify the `print_exp` function to remove this set of parentheses.

6.8 Binary Search Trees

We have already seen two different ways to get key-value pairs in a collection. Recall that these collections implement the **map** abstract data type. The two implementations of a map ADT we discussed were binary search on a list and hash tables. In this section we will study **binary search trees** as yet another way to map from a key to a value. In this case we are not interested in the exact placement of items in the tree, but we are interested in using the binary tree structure to provide for efficient searching.

6.8.1 Search Tree Operations

Before we look at the implementation, let's review the interface provided by the map ADT. You will notice that this interface is very similar to the Python dictionary.

- `Map()` Create a new, empty map.
- `put(key,val)` Add a new key-value pair to the map. If the key is already in the map then replace the old value with the new value.
- `get(key)` Given a key, return the value stored in the map or `None` otherwise.
- `del` Delete the key-value pair from the map using a statement of the form `del map[key]`.
- `len()` Return the number of key-value pairs stored in the map.
- `in` Return `True` for a statement of the form `key in map`, if the given key is in the map.

6.8.2 Search Tree Implementation

A binary search tree relies on the property that keys that are less than the parent are found in the left subtree, and keys that are greater than the parent are found in the right subtree. We will call this the **bst property**. As we implement the Map interface as described above, the bst property will guide our implementation. Figure 6.18 illustrates this property of a binary search tree, showing the keys without any associated values. Notice that the property holds for each parent and child. All of the keys in the left subtree are less than the key in the root. All of the keys in the right subtree are greater than the root.

Now that you know what a binary search tree is, we will look at how a binary search tree is constructed. The search tree in Figure 6.18 represents the nodes that exist after we have inserted the following keys in the order shown: $70, 31, 93, 94, 14, 23, 73$. Since 70 was the first key inserted into the tree, it is the root. Next, 31 is less than 70, so it becomes the left child of 70. Next, 93 is greater than 70, so it becomes the right child of 70. Now we have two levels of

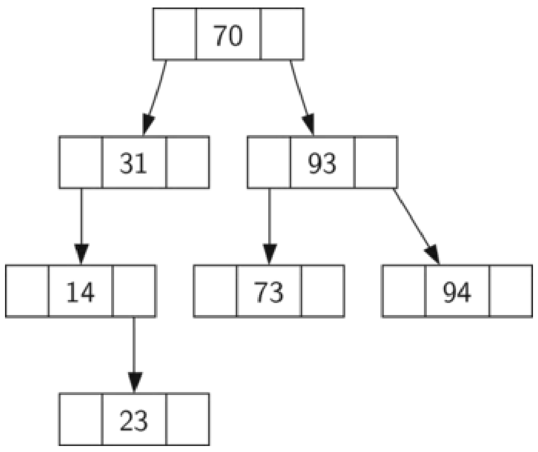


Figure 6.18: A Simple Binary Search Tree

the tree filled, so the next key is going to be the left or right child of either 31 or 93. Since 94 is greater than 70 and 93, it becomes the right child of 93. Similarly 14 is less than 70 and 31, so it becomes the left child of 31. 23 is also less than 31, so it must be in the left subtree of 31. However, it is greater than 14, so it becomes the right child of 14.

To implement the binary search tree, we will use the nodes and references approach similar to the one we used to implement the linked list, and the expression tree. However, because we must be able create and work with a binary search tree that is empty, our implementation will use two classes. The first class we will call **`BinarySearchTree`**, and the second class we will call **`TreeNode`**. The **`BinarySearchTree`** class has a reference to the **`TreeNode`** that is the root of the binary search tree. In most cases the external methods defined in the outer class simply check to see if the tree is empty. If there are nodes in the tree, the request is just passed on to a private method defined in the **`BinarySearchTree`** class that takes the root as a parameter. In the case where the tree is empty or we want to delete the key at the root of the tree, we must take special action. The code for the **`BinarySearchTree`** class constructor along with a few other miscellaneous functions is shown below.

```python
# Basic BinarySearchTree class - incomplete
class BinarySearchTree:

   def __init__(self):
      self.root = None
      self.size = 0

   def length(self):
      return self.size

   def __len__(self):
      return self.size

   def __iter__(self):
      return self.root.__iter__()
```

The **`TreeNode`** class provides many helper functions that make the work done in the **`BinarySearchTree`** class methods much easier. The constructor for a **`TreeNode`**, along with these helper functions, is shown below. As you can see in the code many of these helper functions help to classify a node according to its own position as a child, (left or right) and the kind of children the node has. The TreeNode class will also explicitly keep track of the parent as an attribute of each node. You will see why this is important when we discuss the implementation for the del operator.

```
# Completed TreeNode class
class TreeNode:
   def __init__(self, key, val, left = None, right = None, parent =
       None):
        self.key = key
        self.payload = val
        self.left_child = left
        self.right_child = right
        self.parent = parent

    def has_left_child(self):
        return self.left_child

    def has_right_child(self):
        return self.right_child

    def is_left_child(self):
        return self.parent and self.parent.left_child == self

    def is_right_child(self):
        return self.parent and self.parent.right_child == self

    def is_root(self):
        return not self.parent

    def is_leaf(self):
        return not (self.right_child or self.left_child)

    def has_any_children(self):
        return self.right_child or self.left_child

    def has_both_children(self):
        return self.right_child and self.left_child

    def replace_node_data(self, key, value, lc, rc):
        self.key = key
        self.payload = value
        self.left_child = lc
        self.right_child = rc
        if self.has_left_child():
           self.left_child.parent = self
        if self.has_right_child():
           self.right_child.parent = self
```

Another interesting aspect of the implementation of `TreeNode` is that we use Python's optional parameters. Optional parameters make it easy for us to create a `TreeNode` under several different circumstances. Sometimes we will want to construct a new `TreeNode` that already has both a **parent** and a **child**. With an existing parent and child, we can pass parent and child as parameters. At other times we will just create a `TreeNode` with the key value pair, and we will not pass any parameters for `parent` or `child`. In this case, the default values of the optional parameters are used.

Now that we have the `BinarySearchTree` shell and the `TreeNode` it is time to write the `put` method that will allow us to build our binary search tree. The `put` method is a method of the `BinarySearchTree` class. This method will check to see if the tree already has a root. If there is not a root then `put` will create a new `TreeNode` and install it as the root of the tree. If a root node is already in place then `put` calls the private, recursive, helper function _put to search the tree according to the following algorithm:

- Starting at the root of the tree, search the binary tree comparing the new key to the key in the current node. If the new key is less than the current node, search the left subtree. If the new key is greater than the current node, search the right subtree.
- When there is no left (or right) child to search, we have found the position in the tree where the new node should be installed.
- To add a node to the tree, create a new TreeNode object and insert the object at the point discovered in the previous step.

Below we show the Python code for inserting a new node in the tree. The _put function is written recursively following the steps outlined above. Notice that when a new child is inserted into the tree, the `current_node` is passed to the new tree as the parent.

One important problem with our implementation of insert is that duplicate keys are not handled properly. As our tree is implemented a duplicate key will create a new node with the same key value in the right subtree of the node having the original key. The result of this is that the node with the new key will never be found during a search. A better way to handle the insertion of a duplicate key is for the value associated with the new key to replace the old value. We leave fixing this bug as an exercise for you.

```
def put(self, key, val):
   if self.root:
      self._put(key, val, self.root)
   else:
      self.root = TreeNode(key, val)
   self.size = self.size + 1

def _put(self, key, val, current_node):
   if key < current_node.key:
      if current_node.has_left_child():
            self._put(key, val, current_node.left_child)
      else:
            current_node.left_child = TreeNode(key, val,
               parent=current_node)
   else:
      if current_node.has_right_child():
```

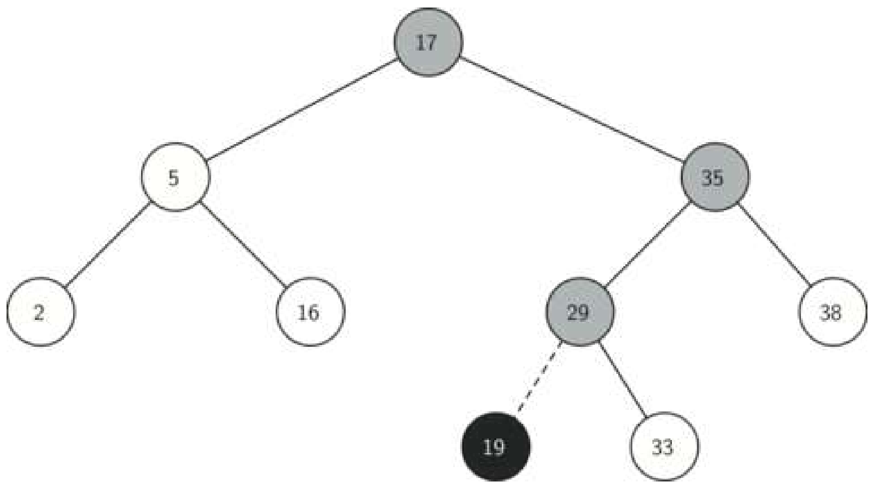


Figure 6.19: Inserting a Node with Key $= 19$

```
            self._put(key, val, current_node.right_child)
    else:
            current_node.right_child = TreeNode(key, val,
                parent=current_node)
```

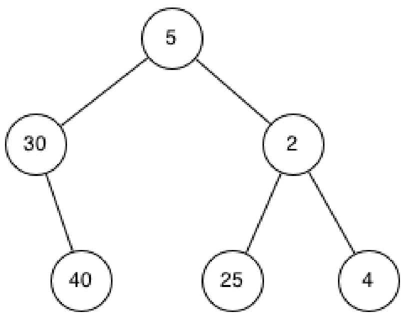
With the **put** method defined, we can easily overload the **[]** operator for assignment by having the **__setitem__** method call the put method. This allows us to write Python statements like **my_zip_tree['Plymouth'] = 55446**, just like a Python dictionary.

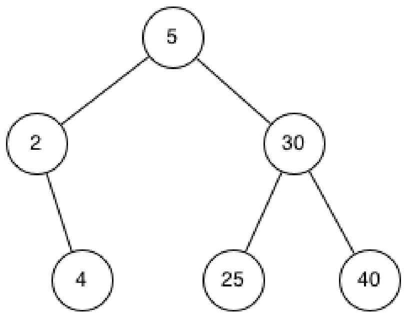
```
def __setitem__(self, k, v):
    self.put(k, v)
```

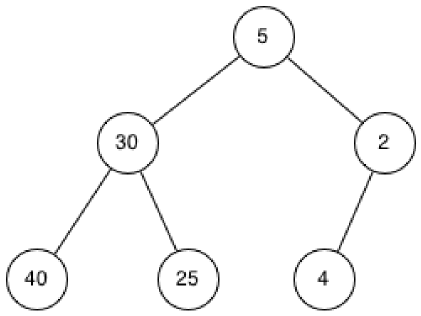
Figure 6.19 illustrates the process for inserting a new node into a binary search tree. The lightly shaded nodes indicate the nodes that were visited during the insertion process.

Self Check

Which of the trees shows a correct binary search tree given that the keys were inserted in the following order $5, 30, 2, 40, 25, 4$?

1. 

2. 

3. 

Once the tree is constructed, the next task is to implement the retrieval of a value for a given key. The `get` method is even easier than the `put` method because it simply searches the tree recursively until it gets to a non-matching leaf node or finds a matching key. When a matching key is found, the value stored in the payload of the node is returned.

Below we show the code for `get`, `_get` and `__getitem__`. The search code in the `_get` method uses the same logic for choosing the left or right child as the `_put` method. Notice that the `_get` method returns a `TreeNode` to get, this allows `_get` to be used as a flexible helper method for other `BinarySearchTree` methods that may need to make use of other data from the `TreeNode` besides the payload.

By implementing the `__getitem__` method we can write a Python statement that looks just like we are accessing a dictionary, when in fact we are using a binary search tree, for example `z = my_zip_tree['Fargo']`. As you can see, all the`__getitem__` method does is call `get`.

```
def get(self,key):
   if self.root:
      res = self._get(key, self.root)
      if res:
            return res.payload
      else:
            return None
   else:
      return None

def _get(self, key, current_node):
   if not current_node:
      return None
   elif current_node.key == key:
      return current_node
```

```
    elif key < current_node.key:
        return self._get(key, current_node.left_child)
    else:
        return self._get(key, current_node.right_child)

def __getitem__(self, key):
    return self.get(key)
```

Using **get**, we can implement the **in** operation by writing a __contains__ method for the **BinarySearchTree**. The __contains__ method will simply call **get** and return **True** if **get** returns a value, or **False** if it returns **None**. The code for __contains__ is shown below.

```
def __contains__(self, key):
    if self._get(key, self.root):
        return True
    else:
        return False
```

Recall that __contains__ overloads the in operator and allows us to write statements such as:

```
if 'Northfield' in my_zip_tree:
    print("oom ya ya")
```

Finally, we turn our attention to the most challenging method in the binary search tree, the deletion of a key. The first task is to find the node to delete by searching the tree. If the tree has more than one node we search using the _get method to find the **TreeNode** that needs to be removed. If the tree only has a single node, that means we are removing the root of the tree, but we still must check to make sure the key of the root matches the key that is to be deleted. In either case if the key is not found the **del** operator raises an error.

```
def delete(self, key):
   if self.size > 1:
      node_to_remove = self._get(key, self.root)
      if node_to_remove:
         self.remove(node_to_remove)
         self.size = self.size - 1
      else:
         raise KeyError('Error, key not in tree')
   elif self.size == 1 and self.root.key == key:
      self.root = None
      self.size = self.size - 1
   else:
      raise KeyError('Error, key not in tree')

def __delitem__(self, key):
    self.delete(key)
```

Once we've found the node containing the key we want to delete, there are three cases that we must consider:

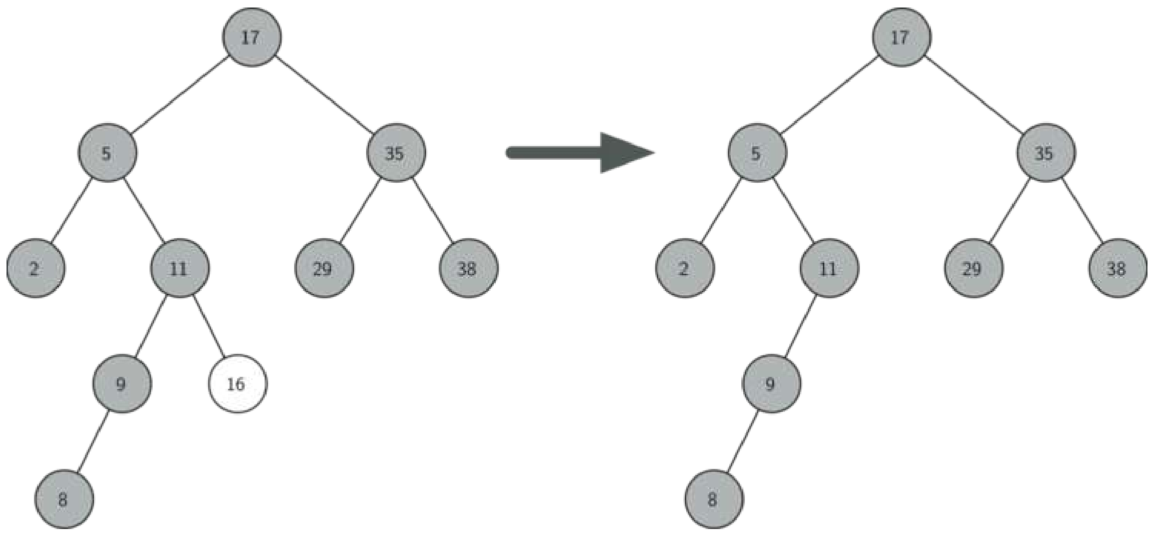


Figure 6.20: Deleting Node 16, a Node without Children

1. The node to be deleted has no children (see Figure 6.20).
2. The node to be deleted has only one child (see Figure 6.21).
3. The node to be deleted has two children (see Figure 6.22).

The first case is straightforward. If the current node has no children all we need to do is delete the node and remove the reference to this node in the parent. The code for this case is shown in here.

```
if current_node.is_leaf():
   if current_node == current_node.parent.left_child:
      current_node.parent.left_child = None
   else:
      current_node.parent.right_child = None
```

The second case is only slightly more complicated. If a node has only a single child, then we can simply promote the child to take the place of its parent. The code for this case is shown below. As you look at this code you will see that there are six cases to consider. Since the cases are symmetric with respect to either having a left or right child we will just discuss the case where the current node has a left child. The decision proceeds as follows:

1. If the current node is a left child then we only need to update the parent reference of the left child to point to the parent of the current node, and then update the left child reference of the parent to point to the current node's left child.
2. If the current node is a right child then we only need to update the parent reference of the right child to point to the parent of the current node, and then update the right child reference of the parent to point to the current node's right child.
3. If the current node has no parent, it must be the root. In this case we will just replace the **key,payload**, **left_child**, and **right_child** data by calling the **replace_node_data** method on the root.

```
else: # this node has one child
```

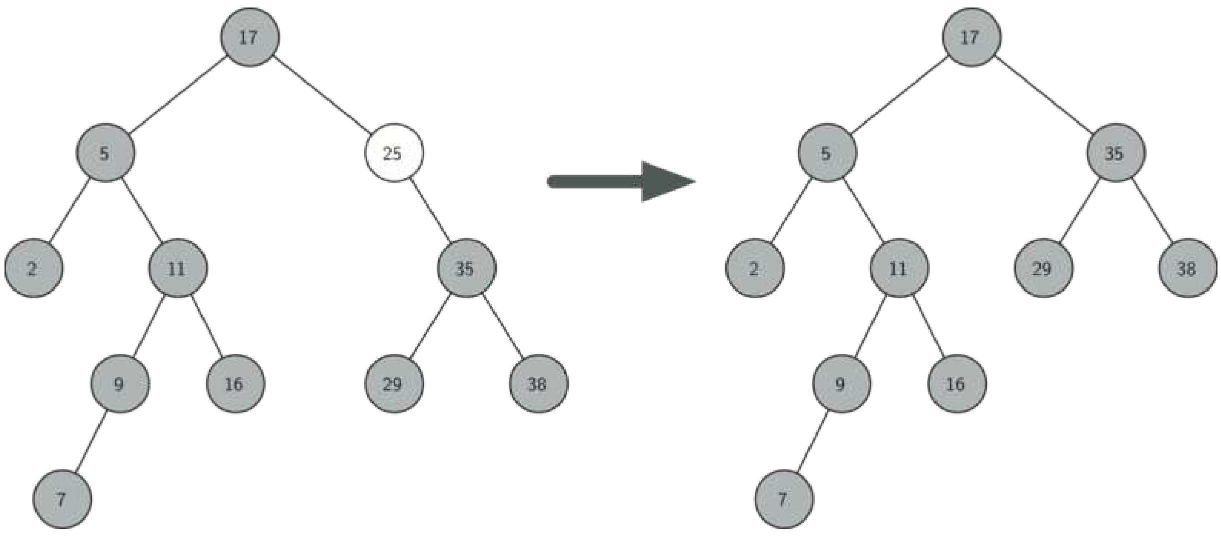


Figure 6.21: Deleting Node 25, a Node That Has a Single Child

```
if current_node.has_left_child():
   if current_node.is_left_child():
      current_node.left_child.parent = current_node.parent
      current_node.parent.left_child = current_node.left_child
   elif current_node.is_right_child():
      current_node.left_child.parent = current_node.parent
      current_node.parent.right_child = current_node.left_child
   else:
      current_node.replace_node_data(current_node.left_child.key,
                     current_node.left_child.payload,
                     current_node.left_child.left_child,
                     current_node.left_child.right_child)
else:
   if current_node.is_left_child():
      current_node.right_child.parent = current_node.parent
      current_node.parent.left_child = current_node.right_child
   elif current_node.is_right_child():
      current_node.right_child.parent = current_node.parent
      current_node.parent.right_child = current_node.right_child
   else:
      current_node.replace_node_data(current_node.right_child.key,
                     current_node.right_child.payload,
                     current_node.right_child.left_child,
                     current_node.right_child.right_child)
```

The third case is the most difficult case to handle. If a node has two children, then it is unlikely that we can simply promote one of them to take the node's place. We can, however, search the tree for a node that can be used to replace the one scheduled for deletion. What we need is a node that will preserve the binary search tree relationships for both of the existing left and right subtrees. The node that will do this is the node that has the next-largest key in the tree. We call this node the **successor**, and we will look at a way to find the successor shortly. The successor is guaranteed to have no more than one child, so we know how to remove it using the two cases for deletion that we have already implemented. Once the successor has been removed,

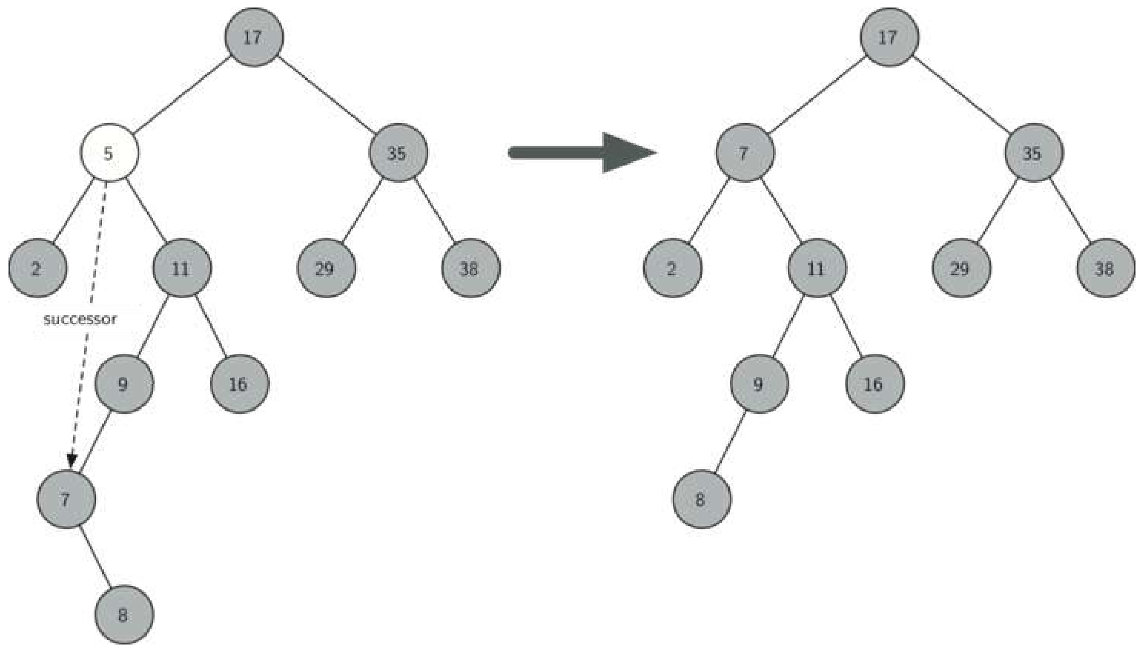


Figure 6.22: Deleting Node 5, a Node with Two Children

we simply put it in the tree in place of the node to be deleted.

The code to handle the third case is shown below. Notice that we make use of the helper methods **find_successor** and **find_min** to find the successor. To remove the successor, we make use of the method **splice_out**. The reason we use **splice_out** is that it goes directly to the node we want to splice out and makes the right changes. We could call **delete** recursively, but then we would waste time re-searching for the key node.

```
elif current_node.has_both_children(): #interior
       succ = current_node.find_successor()
       succ.splice_out()
       current_node.key = succ.key
       current_node.payload = succ.payload
```

The code to find the successor is shown below and as you can see is a method of the **TreeNode** class. This code makes use of the same properties of binary search trees that cause an inorder traversal to print out the nodes in the tree from smallest to largest. There are three cases to consider when looking for the successor:

1. If the node has a right child, then the successor is the smallest key in the right subtree.
2. If the node has no right child and is the left child of its parent, then the parent is the successor.
3. If the node is the right child of its parent, and itself has no right child, then the successor to this node is the successor of its parent, excluding this node.
4. The first condition is the only one that matters for us when deleting a node from a binary search tree. However, the find_successor method has other uses that we will explore in the exercises at the end of this chapter.

The `find_min` method is called to find the minimum key in a subtree. You should convince yourself that the minimum valued key in any binary search tree is the leftmost child of the tree. Therefore the `find_min` method simply follows the `left_child` references in each node of the subtree until it reaches a node that does not have a left child.

```
def find_successor(self):
   succ = None
   if self.has_right_child():
      succ = self.right_child.find_min()
   else:
      if self.parent:
            if self.is_left_child():
               succ = self.parent
            else:
               self.parent.right_child = None
               succ = self.parent.find_successor()
               self.parent.right_child = self
   return succ

def find_min(self):
   current = self
   while current.has_left_child():
      current = current.left_child
   return current

def splice_out(self):
   if self.is_leaf():
      if self.is_left_child():
            self.parent.left_child = None
      else:
            self.parent.right_child = None
   elif self.has_any_children():
      if self.has_left_child():
            if self.is_left_child():
              self.parent.left_child = self.left_child
            else:
              self.parent.right_child = self.left_child
            self.left_child.parent = self.parent
      else:
            if self.is_left_child():
              self.parent.left_child = self.right_child
            else:
              self.parent.right_child = self.right_child
            self.right_child.parent = self.parent
```

We need to look at one last interface method for the binary search tree. Suppose that we would like to simply iterate over all the keys in the tree in order. This is definitely something we have done with dictionaries, so why not trees? You already know how to traverse a binary tree in order, using the `inorder` traversal algorithm. However, writing an iterator requires a bit more work, since an iterator should return only one node each time the iterator is called.

Python provides us with a very powerful function to use when creating an iterator. The function is called **`yield`**. **`yield`** is similar to **`return`** in that it returns a value to the caller. However, **`yield`** also takes the additional step of freezing the state of the function so that the next time the function is called it continues executing from the exact point it left off earlier. Functions that create objects that can be iterated are called generator functions.

The code for an **`inorder`** iterator of a binary tree is shown below. Look at this code carefully; at first glance you might think that the code is not recursive. However, remember that **`__iter__`** overrides the **`for x in`** operation for iteration, so it really is recursive! Because it is recursive over **`TreeNode`** instances the **`__iter__`** method is defined in the TreeNode class.

```
def __iter__(self):
  if self:
    if self.has_left_child():
         for elem in self.left_child:
           yield elem
    yield self.key
    if self.has_right_child():
         for elem in self.right_child:
           yield elem
```

At this point you may want to download the entire file containing the full version of the **`BinarySearchTree`** and **`TreeNode`** classes.

```
# Complete BinarySearchTree implementation

class TreeNode:
   def __init__(self, key, val, left = None, right = None, parent =
      None):
      self.key = key
      self.payload = val
      self.left_child = left
      self.right_child = right
      self.parent = parent

   def has_left_child(self):
      return self.left_child

   def has_right_child(self):
      return self.right_child

   def is_left_child(self):
      return self.parent and self.parent.left_child == self

   def is_right_child(self):
      return self.parent and self.parent.right_child == self

   def is_root(self):
      return not self.parent
```

```
    def is_leaf(self):
        return not (self.right_child or self.left_child)

    def has_any_children(self):
        return self.right_child or self.left_child

    def has_both_children(self):
        return self.right_child and self.left_child

    def replace_node_data(self, key, value, lc, rc):
        self.key = key
        self.payload = value
        self.left_child = lc
        self.right_child = rc
        if self.has_left_child():
            self.left_child.parent = self
        if self.has_right_child():
            self.right_child.parent = self


class BinarySearchTree:
    def __init__(self):
        self.root = None
        self.size = 0

    def length(self):
        return self.size

    def __len__(self):
        return self.size

    def put(self, key, val):
        if self.root:
            self._put(key, val, self.root)
        else:
            self.root = TreeNode(key,val)
        self.size = self.size + 1

    def _put(self, key, val, current_node):
        if key < current_node.key:
            if current_node.has_left_child():
                    self._put(key, val, current_node.left_child)
            else:
                    current_node.left_child = TreeNode(key, val, parent =
                        current_node)
        else:
            if current_node.has_right_child():
                    self._put(key, val, current_node.right_child)
            else:
                    current_node.right_child = TreeNode(key, val, parent
                        = current_node)
```

```python
    def __setitem__(self, k, v):
        self.put(k, v)

    def get(self, key):
        if self.root:
            res = self._get(key, self.root)
            if res:
                return res.payload
            else:
                return None
        else:
            return None

    def _get(self, key, current_node):
        if not current_node:
            return None
        elif current_node.key == key:
            return current_node
        elif key < current_node.key:
            return self._get(key, current_node.left_child)
        else:
            return self._get(key, current_node.right_child)

    def __getitem__(self, key):
        return self.get(key)

    def __contains__(self, key):
        if self._get(key, self.root):
            return True
        else:
            return False

    def delete(self, key):
        if self.size > 1:
            node_to_remove = self._get(key, self.root)
            if node_to_remove:
                self.remove(node_to_remove)
                self.size = self.size - 1
            else:
                raise KeyError('Error, key not in tree')
        elif self.size == 1 and self.root.key == key:
            self.root = None
            self.size = self.size - 1
        else:
            raise KeyError('Error, key not in tree')

    def __delitem__(self, key):
        self.delete(key)

    def splice_out(self):
        if self.is_leaf():
            if self.is_left_child():
                self.parent.left_child = None
            else:
                self.parent.right_child = None
        elif self.has_any_children():
            if self.has_left_child():
                if self.is_left_child():
                    self.parent.left_child = self.left_child
                else:
                    self.parent.right_child = self.left_child
                self.left_child.parent = self.parent
            else:
                if self.is_left_child():
                    self.parent.left_child = self.right_child
                else:
                    self.parent.right_child = self.right_child
                self.right_child.parent = self.parent

    def find_successor(self):
        succ = None
        if self.has_right_child():
            succ = self.right_child.find_min()
        else:
            if self.parent:
                if self.is_left_child():
                    succ = self.parent
                else:
                    self.parent.right_child = None
                    succ = self.parent.find_successor()
                    self.parent.right_child = self
        return succ

    def find_min(self):
        current = self
        while current.has_left_child():
            current = current.left_child
        return current

    def remove(self, current_node):
        if current_node.is_leaf(): #leaf
            if current_node == current_node.parent.left_child:
                current_node.parent.left_child = None
            else:
                current_node.parent.right_child = None
        elif current_node.has_both_children(): # interior
            succ = current_node.find_successor()
            succ.splice_out()
            current_node.key = succ.key
            current_node.payload = succ.payload
        else: # this node has one child
          if current_node.has_left_child():
            if current_node.is_left_child():
                current_node.left_child.parent = current_node.parent
                current_node.parent.left_child = current_node.left_child
            elif current_node.is_right_child():
                current_node.left_child.parent = current_node.parent
                current_node.parent.right_child =
                    current_node.left_child
            else:
                current_node.replace_node_data(current_node.left_child.key,
                                  current_node.left_child.payload,
                                  current_node.left_child.left_child,
                                  current_node.left_child.right_child)
          else:
            if current_node.is_left_child():
                current_node.right_child.parent = current_node.parent
                current_node.parent.left_child =
                    current_node.right_child
            elif current_node.is_right_child():
                current_node.right_child.parent = current_node.parent
                current_node.parent.right_child =
                    current_node.right_child
            else:
                current_node.replace_node_data(current_node.right_child.key,
                                  current_node.right_child.payload,
                                  current_node.right_child.left_child,
                                  current_node.right_child.right_child)


my_tree = BinarySearchTree()
my_tree[3] = "red"
my_tree[4] = "blue"
my_tree[6] = "yellow"
my_tree[2] = "at"

print(my_tree[6])
print(my_tree[2])
```

6.8.3 Search Tree Analysis

With the implementation of a binary search tree now complete, we will do a quick analysis of the methods we have implemented. Let's first look at the `put` method. The limiting factor on its performance is the height of the binary tree. Recall from the vocabulary section that the height of a tree is the number of edges between the root and the deepest leaf node. The height is the limiting factor because when we are searching for the appropriate place to insert a node into the tree, we will need to do at most one comparison at each level of the tree.

What is the height of a binary tree likely to be? The answer to this question depends on how

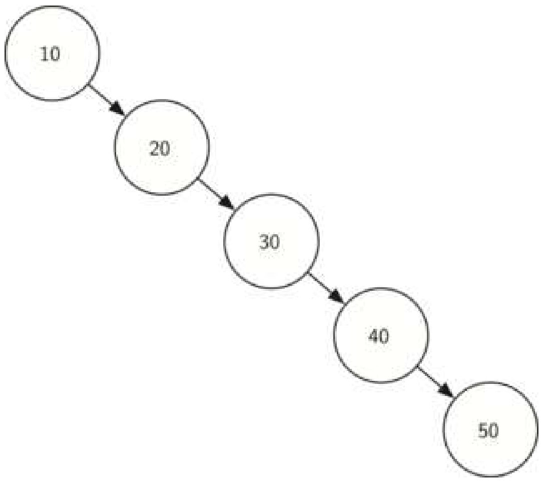


Figure 6.23: A skewed binary search tree would give poor performance

the keys are added to the tree. If the keys are added in a random order, the height of the tree is going to be around $\log 2n$ where n is the number of nodes in the tree. This is because if the keys are randomly distributed, about half of them will be less than the root and half will be greater than the root. Remember that in a binary tree there is one node at the root, two nodes in the next level, and four at the next. The number of nodes at any particular level is 2^d where d is the depth of the level. The total number of nodes in a perfectly balanced binary tree is $2^h + 1 - 1$, where h represents the height of the tree.

A perfectly balanced tree has the same number of nodes in the left subtree as the right subtree. In a balanced binary tree, the worst-case performance of **put** is $O(\log_2 n)$, where n is the number of nodes in the tree. Notice that this is the inverse relationship to the calculation in the previous paragraph. So $\log_2 n$ gives us the height of the tree, and represents the maximum number of comparisons that **put** will need to do as it searches for the proper place to insert a new node.

Unfortunately it is possible to construct a search tree that has height n simply by inserting the keys in sorted order! An example of such a tree is shown in Figure 6.23. In this case the performance of the **put** method is $O(n)$.

Now that you understand that the performance of the **put** method is limited by the height of the tree, you can probably guess that other methods, **get**, **in**, and **del**, are limited as well. Since **get** searches the tree to find the key, in the worst case the tree is searched all the way to the bottom and no key is found. At first glance **del** might seem more complicated, since it may need to search for the successor before the deletion operation can complete. But remember that the worst-case scenario to find the successor is also just the height of the tree which means that you would simply double the work. Since doubling is a constant factor it does not change worst case analysis of $O(n)$ for an unbalanced tree.

6.9 Summary

In this chapter we have looked at the tree data structure. The tree data structure enables us to write many interesting algorithms. In this chapter we have looked at algorithms that use trees to do the following:

- A binary tree for parsing and evaluating expressions.
- A binary tree for implementing the map ADT.
- A binary tree to implement a min heap.
- A min heap used to implement a priority queue.

6.10 Key Terms

AVL tree	binary heap	binary search tree
binary tree	child / children	complete binary tree
edge	heap order property	height
inorder	leaf node	level
map	min/max heap	node
parent	path	postorder
preorder	priority queue	root
rotation	sibling	successor
subtree	tree	

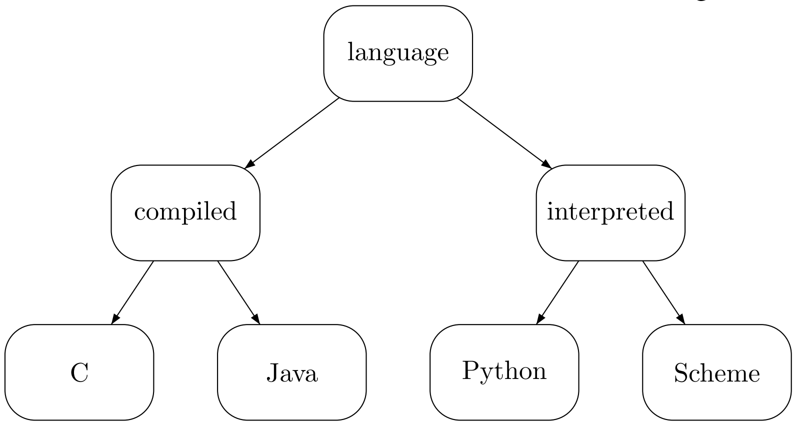
6.11 Discussion Questions

1. Draw the tree structure resulting from the following set of tree function calls:

```
>>> r = BinaryTree(3)
>>> insert_left(r,4)
[3, [4, [], []], []]
>>> insert_left(r,5)
[3, [5, [4, [], []], []], []]
>>> insert_right(r,6)
[3, [5, [4, [], []], []], [6, [], []]]
>>> insert_right(r,7)
[3, [5, [4, [], []], []], [7, [], [6, [], []]]]
>>> set_root_val(r,9)
>>> insert_left(r,11)
[9, [11, [5, [4, [], []], []], []], [7, [], [6, [], []]]]
```

2. Trace the algorithm for creating an expression tree for the expression $(4 * 8)/6 - 3$.
3. Consider the following list of integers: $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$. Show the binary search tree resulting from inserting the integers in the list.
4. Consider the following list of integers: $[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]$. Show the binary search tree resulting from inserting the integers in the list.
5. Generate a random list of integers. Show the binary heap tree resulting from inserting the integers on the list one at a time.

6. Using the list from the previous question, show the binary heap tree resulting from using the list as a parameter to the `build_heap` method. Show both the tree and list form.
7. Draw the binary search tree that results from inserting the following keys in the order given: 68, 88, 61, 89, 94, 50, 4, 76, 66, and 82.
8. Generate a random list of integers. Draw the binary search tree resulting from inserting the integers on the list.
9. Consider the following list of integers: $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$. Show the binary heap resulting from inserting the integers one at a time.
10. Consider the following list of integers: $[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]$. Show the binary heap resulting from inserting the integers one at a time.
11. Consider the two different techniques we used for implementing traversals of a binary tree. Why must we check before the call to preorder when implementing as a method, whereas we could check inside the call when implementing as a function?
12. Show the function calls needed to build the following binary tree.



6.12 Programming Exercises

1. Extend the `build_parse_tree` function to handle mathematical expressions that do not have spaces between every character.
2. Modify the `build_parse_tree` and evaluate functions to handle boolean statements (and, or, and not). Remember that "not" is a unary operator, so this will complicate your code somewhat.
3. Using the `find_successor` method, write a non-recursive inorder traversal for a binary search tree.
4. Modify the code for a binary search tree to make it threaded. Write a non-recursive inorder traversal method for the threaded binary search tree. A threaded binary tree maintains a reference from each node to its successor.
5. Modify our implementation of the binary search tree so that it handles duplicate keys properly. That is, if a key is already in the tree then the new payload should replace the old rather than add another node with the same key.

6. Create a binary heap with a limited heap size. In other words, the heap only keeps track of the `n` most important items. If the heap grows in size to more than `n` items the least important item is dropped.
7. Clean up the `print_exp` function so that it does not include an 'extra' set of parentheses around each number.
8. Using the `build_heap` method, write a sorting function that can sort a list in $O(n \log n)$ time.
9. Write a function that takes a parse tree for a mathematical expression and calculates the derivative of the expression with respect to some variable.
10. Implement a binary heap as a max heap.
11. Using the `BinaryHeap` class, implement a new class called `PriorityQueue`. Your `PriorityQueue` class should implement the constructor, plus the `enqueue` and `dequeue` methods.

CHAPTER

SEVEN

JSON

7.1 Objectives

- To establish an overview of why we use JSON
- To understand the syntax of JSON
- Demonstrate how Python objects can be serialized into JSON using the native library

7.2 What is JSON?

JSON stands for **Javascript Object Notation** and is a lightweight format used for data interchange. What do we mean by this? Well, in the age of the world wide web it is fairly common for different programs distributed across a network to wish to communicate and share data with one another. This can cause problems however as there are a large number of different programming languages in the wild. Its not hard to imagine a situation in which a program written in Python might want to send an object to another program that is written in Java, but the internal representations are going to be quite different. This can be a problem even when developing an offline project, sometimes it is more convenient to actually write different parts of a project in separate languages because they have different requirements that might be better suited to separate languages.

Well, the obvious solution is to translate the format into an intermediate data type that can be understood by both languages. JSON is one such format. Others include XML, SOAP and YAML.

7.3 The JSON Syntax

JSON syntax is based around three simple data types. The first is a Name-Value pair which in a programming language could be realised as an object, record, struct, dictionary, hash table, keyed list, or associative array. A name-value pair looks as follows:

```
"first_name" : "Jacob"
```

The name here can be any String, and is typically used to refer to the name of a property of a particular object. The value can be one of the following:

- A number
- A String
- A Boolean
- A JSON Array
- An object
- null

The second type is the JSON Object. A JSON Object is a collection of name-value pairs or Arrays encased in curly brackets.

```
{ "first_name":"Jacob" , "last_name":"Bellamy" }
```

Lastly, we have the JSON Array. A JSON Array roughly corresponds to a list in python. A JSON array is a list of values encased in square brackets. For example:

```
{"courses": ["Compsci 101", "Compsci 105", "Compsci 107"]}
```

Putting it all together, suppose we have a fairly complex Object such as a student that we wished to represent in JSON. This student has a name, id number, GPA, a list of courses they are enrolled in, and a boolean representing whether their fees are paid, and an address. a JSON representation of this object might look as follows:

```
{
  "name":"Jacob Bellamy",
  "id_number":3352976,
  "gpa":8.2,
   "courses":[
     "Compsci 101", "Compsci 105", "Phil 101", "Maths 108"
   ],
   "fees_paid":true
   "address": {
     "street_address": "1 Horse Lane",
      "city": "Auckland",
      "post_code": 0632
   }
}
```