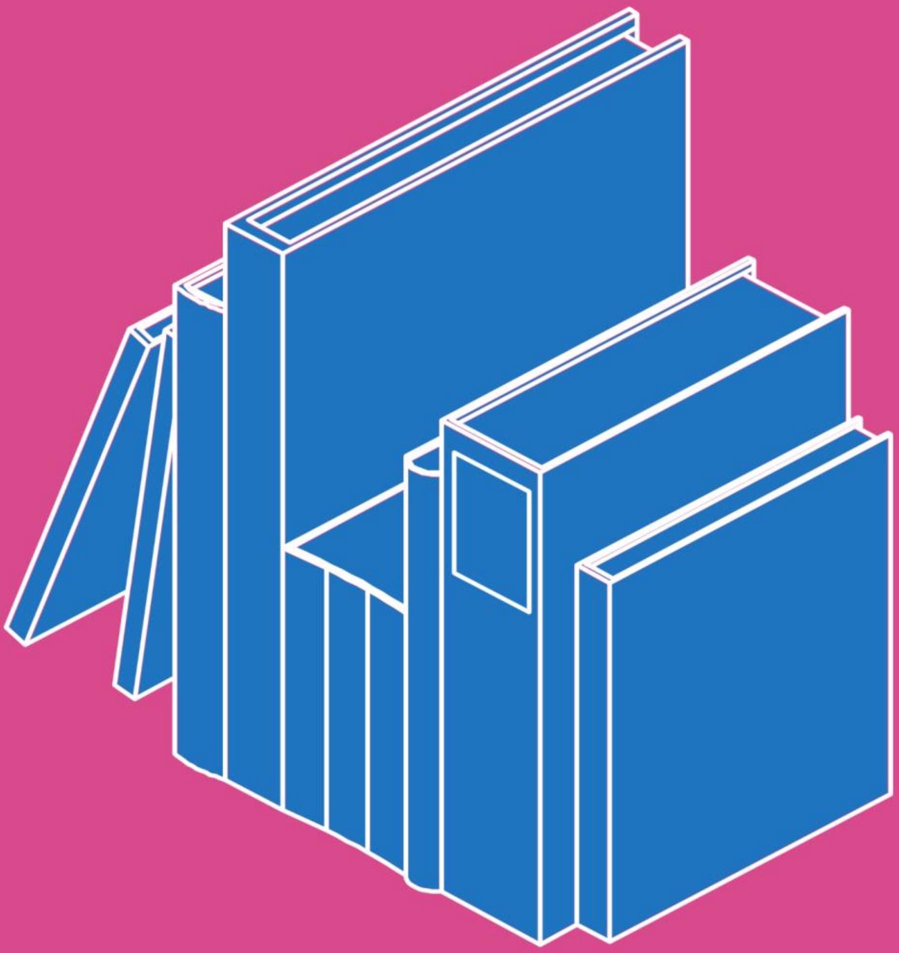
SLITHER INTO...

DATA STRUCTURES AND ALGORITHMS



THE CONCISE GUIDE TO THE BACKBONE OF THE DIGITAL WORLD THROUGH PYTHON

Contents

Who is this book for?

Data structures and the algorithms that bring them to life are the backbone of every successful application you can think of. Through their uses we have a powerful way to unlock insights that are buried in large amounts of data. This book will provide you with a strong foothold in the world of data structures and algorithms. You will see many data structures and the algorithms that underpin them and how we use them in the real world. You will see how we can use them to build things such as the queue function that is popular in media players such as Spotify that allows you to queue songs or YouTube that allows you to queue videos. You will see how the algorithms that underpin the most popular compression formats such as JPEG, Zip, GZIP and MPEG-2 work, and you will learn how to compress your own files using the algorithms you've written. You will learn some of the algorithms the are used in routing data around the internet and linking the networks that make up the internet together. You will also learn the most common algorithms used in DNA or RNA sequence pattern matching (something that is quite relevant at the time of writing).

To make it through this book you will need to understand the basics of Python which includes iteration using loops, flow control using if and else statements and Pythons built-in data types. It would also be useful to have a grasp on object-oriented programming in Python, but I give a refresher on OOP in Python at the beginning of this book. However, if you are a more experienced programmer from another language, and have experience in a few different languages, you can use this book as a reference.

Algorithmic Complexity

It will become clear throughout this book that an important aspect of algorithm design is gauging the efficiency both in terms of space (memory) and time (number of operations). This second measure is called runtime performance or time complexity. We will not be focused on space complexity in this chapter, but an identical metric is used for measuring an algorithms memory performance.

Perhaps the most obvious way to measure the runtime performance of an algorithm is to measure the amount of time it takes for the algorithm to run. This works in comparing algorithms on one machine. However, if we run those same algorithms on another machine with the exact same input, we will likely get very different results. This would be due to the differences in hardware on both machines and how the operating system is scheduling processes. For this reason, we'll need another way of measuring an algorithms performance.

We could count the number of operations it takes for an algorithm to complete however there is a problem with this. There is no way to quantify an operation. This is due to the implementations of different programming languages, different programming styles and deciding on how we count operations. Although this approach alone will not work, we can take this idea and combine it with the expectation that as the size of the input increases, the runtime will increase in a specific way. In other words, there is a mathematical relationship between the size of the input, *n*, and the time it takes for the algorithm to run.

There are multiple principles on which this metric is based. These will become clearer throughout this chapter but for now, they are:

- Worst case analysis (Make no assumptions about the input size)
- Ignore or suppress constant factors and lower order terms. This stems from the fact that at large inputs, higher order terms will dominate so we can ignore the lower order terms.
- Focus on problems with large input sizes.

Worst case analysis is very useful as it gives us a tight upper bound that our algorithm is guaranteed not to exceed. Ignoring small constant factors, and lower order terms is just about ignoring the things that, at large values of the input size, do not contribute in a large degree to the overall run time. Not only does this make our work mathematically easier but it also allows us to focus on the things that are having the most impact on performance.

Big O Notation

There are essentially three things that characterize an algorithm's runtime performance.

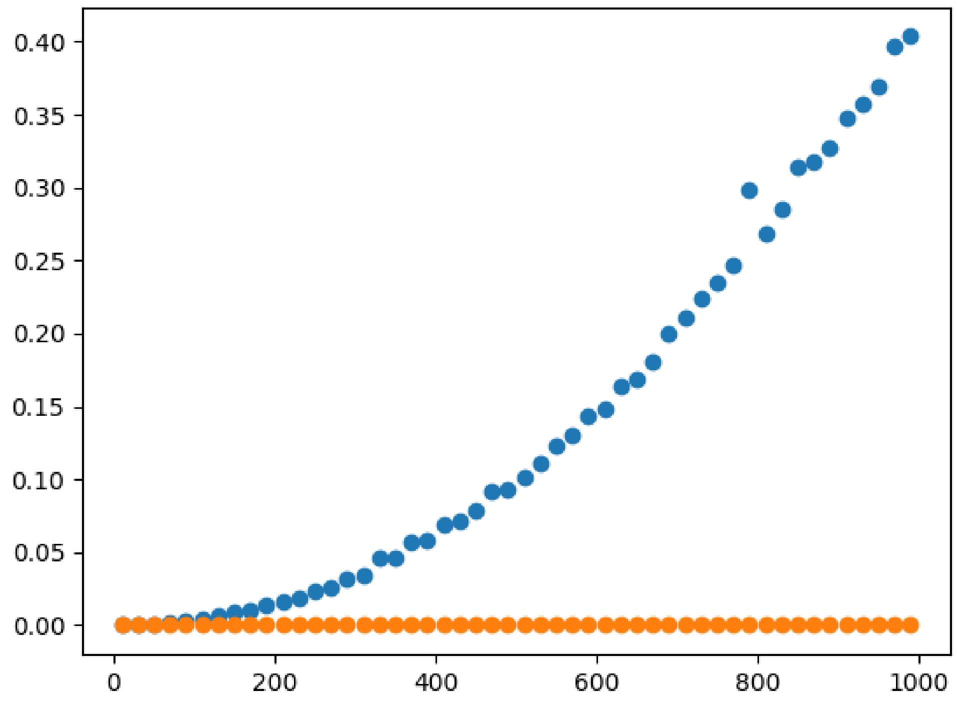
- Worse case - Use an input that gives the slowest performance.
- Best case - Use an input that gives the best performance.

- Average case - Assume the input is random.

Big Odeals with the worst case and it is the case we are usually concerned with!

When we say that both algorithms have a time complexity of *O(n^2)*, we are essentially saying that the performance of the algorithms will grow proportionally to the square of the size of the input (lists in this case).

I have taken the time to run some algorithms with different time complexities on lists of varying sizes and we can look at how the algorithm takes longer to complete as the size of the list grows. You can see this in the diagram below:



In this diagram, the y-axis represents how long it took the algorithm to finish in seconds and the x-axis shows the number of elements in the list. The blue line is for an O(n^2) algorithm and the orange is for an O(n) algorithm (Linear time algorithm). I have a fast computer, so it doesn't appear that the orange line is changing (it is, just very slowly). However, it is very clear that the O(n^2) algorithms become very slow when the size of the input becomes large.

An algorithm's time complexity is usually put into one the following categories starting with the best complexity:

- O(1)- Constant time: It takes the same time for an algorithm to complete no matter the input size
- O(log n)- Logarithmic time: Will be explained below
- O(n)- Linear time: An algorithm whose performance grows in direct proportion to the size of the input.
- O(n log n)- Linearithmic time: Will be explained below
- O(n^2)- Quadratic time: An algorithm whose performance grows in direct proportion to the square of the size of the input.
- O(2^n) - Exponential time: An algorithm whose performance doubles with each addition to the input.

How do we determine what the Big O of an algorithm is? I won't be going into the mathematics behind how we determine this, but I'll be looking at it from a programmers perspective.

Suppose we have some function *f(a)* whose complexity is O(n^2) and it is executed *n* times in a *for* loop as follows:

```
for i in range (n):
    f(...)
```

The complexity of this loop becomes O(n^3). This comes from the fact that we go around the loop *n* times and execute a function that is *O(n^3)* each time giving *O(n^3) = O(n) * O(n^2)*.

As another example, let's look at the following code example:

```
for i in range (n):
    for j in range (n ):
        # statements
```

This is a nested loop. To find out the complexity of this code, we should look at the inner most loop and find it's complexity. The running time of a loop is at most the running time of the statements inside the loop multiplied by the number of iterations. The inner loop is being iterated *n* times as indicated in the *range(n)*. The outer loop is also being iterated *n* times. The gives the code as a whole a runtime of *O(n^2)* which is a product of both loops as given by *O(n) * O(n) = O(n^2)*.

Consider the same code but this time, the inner loop is being iterated *m* times:

```
for i in range (n):
    for j in range (m):
        # statements
```

This means the complexity of the above code will be O(nm). This comes from `O(n) * O(m) = O(nm)`. Now let's add code where that statements comment is:

```
for i in range(n):
    for j in range(n):
        print (i)
        print (j)
```

We consider the print function to have a complexity of O(1) due to the fact that independent of the size of the input to it, it performs the same number of operations each time. Since both of the print functions are sequentially executed this would give us the complexity of the inner loop body to be O(2) which comes from `O(1) + O(1) = O(1 + 1) = O(2)`. However, remember the principle that we ignore or suppress constants. As 2 is a constant, we consider the complexity of the inner loop body to be just O(1). In other words, regardless of how large the variables `i` and `j` become, the inner loop body is still going to complete in the same number of operations, so we consider it constant time. Therefore, the Big O of our code is still O(n^2).

Ok, it should be pretty easy for you to gauge the complexity of a lot of code now, but you'll see algorithms in this book whose complexities are O(log n). What does this mean exactly? Without getting into the mathematics of logarithms, it essentially means that time will increase linearly as the input size increases exponentially. For example, if an algorithm takes 1 second to complete when the input size is 10, it will take 2 seconds to complete when the input size is 100, 3 seconds to complete when the input size is 1000, and so on.

We typically see this complexity class with divide and conquer algorithms such as binary search - where we halve the input size each time, or quicksort were we divide and conquer. In the case of quicksort which we will look at later, we divide a list into two parts each time and sort each part but the operation of finding where to split the array is an O(n) operation. This will give us a runtime complexity of the quicksort algorithm of `O(n Log n) = O(n) * O(Log n)`

Generally speaking, we can divide algorithms into a "good news" category and a "bad news" category. These lay out which algorithms have a good performance, and which do not.

The algorithms that would be in the "good news" category are the algorithm's whose runtime complexities are the following:

- O(1)
- O(log n)
- O(n)
- O(n log n)

The algorithms that would be in the "bad news" category are the algorithm's whose runtime complexities are as follows (and we want to avoid these where possible):

- O(n^k) - where k is >= 2

- O(2^n) - where n is the input size
- O(n!) - that is "n factorial". Factorial operations are very slow.

As an example, if we need to sort a list and we have a choice between two algorithms to use whose algorithmic complexities are O(n log n) and O(n^2) respectively, we would want to use the algorithm whose runtime complexity is O(n log n) as we know that as the input size increases that this algorithm is going far out perform the algorithm with a runtime complexity of O(n^2). This is shown in the graph above.

I've outlined here a way for us to gauge the performance of algorithms in terms of time. Throughout this book you'll encounter many code examples, and I would encourage you to try and work out their runtime complexity. It's a very useful skill to have, and if you were to go for a technical interview and were asked to write code to do something such as sorting a list, I can guarantee you that the interviewer will ask you to give the complexity of the algorithm. You may even be asked something along the lines of "Write an algorithm that will be used to sort a list of *n* integers and has a runtime complexity of O(n log n)". So, this is important stuff and should not be shrugged off. It can be difficult at times to work out the complexity of an algorithm but given some time and some experience, you will get there in the end.

Object Oriented Programming

Classes & Objects

If you're reading this book, you've met many of Pythons built-in types such as *Booleans*, *integers*, *dictionaries* etc. These are all *class types*. This means that any particular instance of a string, list, float etc., is an *object* of the *class* string, list or float. In other words, every string object, e.g., *"Hello"*, is an *instance* of the *string class*. An object is an implementation of a type.

In this chapter we are going to look at a programming paradigm called *object-oriented programming* (OOP). This paradigm is based on the concept of *objects*, which contain data, in the form of attributes and functions (called methods).

In OOP, programs are designed by making them out of objects that interact with one another. An important feature of objects is that they use their methods to access and modify their attributes. For example, if we wanted to model a *Person*, they might have an *age* attributes and a *name* attribute. Each year they get one year older so they may have a *change_age()* method to update their age.

The general syntax for invoking an object's method is *object_name.method_name(arguments)*. We have done this multiple times. For example, calling the *count()* method on a string:

```
>>> s = "I am a string object, an instance of the string class"
>>> s.count('a')
5
```

Most of the time, Python's built-in types are not enough to model data we want, so we built our own types (or classes) which model the data exactly how we need.

A good way to think of this is that a *class* is a cookie cutter, and an *instance* of that class is the cookie we cut out.

Defining New Types

Let's say we wanted to model *Time*. Python's built-in data types are not going to be enough, to easily and logically model this. Instead, we would create our own *Time* class. We define a new class using the *class* statement.

```
class Time:
    pass
```

We have now defined a new class called time. As a side note, *pass* basically means "do nothing". I have it there so our program will not crash.

Save the above code to a file called *my_time.py* and import it as follows:

```
>>> from my_time import Time
>>> t = Time()
>>> type (t)
<class 'time.Ti me'>
>>> isinstance (t, Time)
True
>>> print (t)
<time.Time object at 0x000001A7C6902F60>
```

We can see that *t* is of type *Time* and *t* is an *instance* of *Time* and we can see it is an object stored at the memory location *0x000001A7C6902F60*.

We want to represent time in *hours*, *minutes*, and *seconds*. We can therefore define a method that initializes the time of *Time* objects. We also want to define a *display()* method that will print out the time:

```
class Time:

    def set_time(time_obj, hours, minutes, seconds):
        time_obj.hours = hours
        time_obj.minutes = minutes
        time_obj.seconds = seconds

    def display(time_obj):
        print ("The time is {} : {} : {} ". format (time_obj.hours,
                                                   time_obj.minutes,
                                                   time_obj.seconds))
```

In the above *class definition*, the *set_time()* method requires four arguments. The first is the time object we want to initialize and the other three are the hours, minutes, and seconds we want to set for our time object.

As shown above, we access the hours, minutes and seconds attributes using the *period* operator e.g *time_obj.hours = hours*. This is saying, for the time object that was passed, set it's hours attribute to the hours passed as an argument.

In the *display()* method, we access the hours, minutes, and seconds using the period operator.

Now let's see how we can use this class:

```
>>> from my_time import Time
>>> t = Time()
>>> Time.set_time(t, 11, 32, 45)
>>> Time.display(t)
'The time is 11:32:45'
>>> t2 = Time()
>>> Time.set_time(t2, 22, 43, 17)
>>> Time.display(t2)
'The time is 22:43:17'
>>> Time.display(t)
'The time is 11:32:45'
```

We can see we have two different time objects. We create an object of the *Time* class by calling the class as if it were a function. Calling the class as a function will instantiate an instance of the class and return a reference to it.

The *self* variable

As seen in the previous section, we had to call methods by referring to the class in which it belongs to e.g *Time.display(t)*. However, we didn't have to do that when invoking methods on Python's built in types e.g *s.count('a')*. It turns out that this is just shorthand for *str.count(s, 'a')*.

We can adopt the same approach for our *Time* class:

```
>>> from my_time import Time
>>> t = Time()
>>> t.set_time( 11, 32, 45)
>>> t.display()
'The time is 11:32:45'
```

Note how we only have to pass three arguments to the *set_time()* method rather than the four we declared. This is because when we invoke an instance method on an object, that object is automatically passed as the first argument to the method. Therefore, we supply one less argument than the number we declared in the method definition. Python will automatically supply the missing object as the first argument on our behalf.

By convention, this first parameter, which we've been calling *time_obj*, is named *self*. This refers to the instance on which the method is acting. This makes the methods *set_time()* and *display() instance methods*

Instance methods are methods that act upon a particular instance of an object. It's first parameter is always the object upon which it will operate. By default, all a class's methods are instance methods unless we declare them as *class methods* which we will get to later.

We can therefore re-write our class as follows:

```
class Time:
    def set_time(self, hours, minutes, seconds):
        self.hours = hours
        self.minutes = minutes
        self.seconds = seconds

    def display(self):
        print("The time is {}:{}:{}".format(self.hours,
                                                            self  .minutes,
                                                            self  .seconds))
```

And we can use it as follows:

```
>>> from  my_time  import   Time
>>> t  = Time()
>>> t.set_time(   11,  32,  45)
>>> t2  = Time()
>>> t2.set_time(   22,  34,  18)
>>> t.  display()
'The time is 11:32:45'
>>> t2.display()
'The time is 22:34:18'
```

The init() method

In the previous section we looked at how to define new types by creating classes. We also had to initialize our instance objects using methods like *initialise()*. We do not have to do this with Python's built-in types such as strings.

For example, to initialize a list we could just type *my_list = [1, 2, 3]*.

We can define a special method called *__init__()* (that's double underscore). This is called a constructor. If we provide that method for our class, then it is automatically called when we create an object. We can therefore replace any initialization methods we had previously defined with *__init__()*. This allows us to create and initialize our object in one step.

Let's look at a *Point* that models a point.

```
class   Point:
    def  __init__   ( self  , x, y):
        self  .x   = x
        self  .y   = y

    def  display(   self  ):
        print   ("X coordinate:      {} \ nY coordinate:      {} " . format  ( self  .x,    self
.y))
```

We can now use our point class as follows:

```
from   my_point   import   Point
>>> p1  = Point(  3,   4)
>>> p1.display()
X coordinate:      3
Y coordinate:      4
```

As you can see, our *__init__()* method was called automatically when we created an instance of the point class.

If the *__init__()* method takes two arguments (excluding self), then we must pass two arguments, otherwise we'll get an error.

```
>>> from  my_point  import  Point
>>> p1 = Point()
Traceback (most recent call last):
   File "<stdin>" , line  1, in <module>
TypeError : __init__ () missing  2 required positi   onal arguments:  'x'  and
'y'
```

We can however, set default values for these arguments. This will allow us to initialize an instance of *Point* without passing the arguments:

```
class  Point:
      def __init__ (self , x =0, y =0):
            self .x = x
            self .y = y

      def display( self ):
            print ("X coordinate:   {} \ nY coordinate:   {} " . format (self .x,  self
.y))
```

We can now use our class as follows:

```
from  my_point  import  Point
>>> p1 = Point()
>>> p1.display()
X coordinate:   0
Y coordinate:   0
```

What happens when we define this special method and create an instance is:

- Python creates an empty instance of the *Point* class.
- This empty object is passed, along with *3* and *4* to the *__init__()* method.
- *__init__()* initializes the object with the supplied arguments
- A reference to the initialized object is created and assigned to *p1*.

Remember, special methods are never called directly!

The str() method

The *__str__()* method is another special function. We can *override*it which means we can change it's implementation and hence it's behavior.

When we call *print()* on a Python built-in such as a string or a list, we get a nicely formatted, human readable representation of the object.

Under the hood, the *list* built in type implements this *__str__()* method and hence, allows us to print our lists in human readable format.

We can *override*this method to allow us to format and print our objects as we see fit. Therefore, when we call *print()* on an object of a class we've defined, we'll no longer get something like *<time.Time object at 0x000001A7C6902F60>* printed to the terminal.

We do this by returning a formatted string. Let's look at how we might do this for the *Point*class. We could now replace our *display()* function and replace it with *__str__()*, that way we can just print our point instance.

```
cl ass  Point:
      def  __in it__  ( self  , x  =0, y  =0):
            self  .x   =  x
            self  .y   =  y

      def  __str__  ( self  ):
            return   "X coordinate:      {} \ nY coordinate:      {} " . format  ( self  .x,    sel
f .y)
```

We can now use our class as follows:

```
>>> from   my_point   import   Point
>>> p1  =  Point(  2,   3)
>>> print   (p1)
X coordinate:      2
Y coordinate:      3
```

The return value of this special method must be a string object.

The len() method

As you can probably guess, the *__len__()* method allows us *override*the default implementation of the *len()* function to define what the length of our own types are. For example with a *Point*, the length may be the distance from the origin (the point (0, 0)). However, we must return an integer. We will get an error otherwise. So, we'll have to round either up or down. Python does this in order to make the *len()* function predictable. In this example, we'll just cast our return value to an integer:

```
class   Point:
      def  __init__   ( self  , x  =0, y  =0):
            self  .x   =  x
            self  .y   =  y

      def  __str__  ( self  ):
            r eturn   "X coordinate:      {} \ nY coordinate:      {} " . format  ( self  .x,    sel
f .y)
```

```
    def __len__(self):
        distance_from_center = ((self.x) ** 2 + (self.y) ** 2) ** 0.5
        return distance_from_center
```

We can now call the *Len()* function on instances of our *Point* class.

```
>>> from my_point import Point
>>> p1 = Point( 24, 33)
>>> len (p1)
40
```

The actual distance of this point from the center is 40.8044…

It is unfortunate that we can't return a floating point number, therefore if we want real accuracy we'll have to implement a different method that we'll have to call on the instance.

Operator overloading

There are a ton of special methods in Python, I'm not going to cover them all, however, there are some special methods that are called when we use operators. For example, when we use the comparison operator == on two integers, a special method called *__eq__()* is called.

When *method overriding* is done on methods that define the behavior of *operators*, we call that: *operator overloading*

We can overload the default behaviour of the == method, so that it applies to our *Point* class. This is done as follows:

```
class Point:
    def __init__(self, x =0, y =0):
        self.x = x
        self.y = y

    def __str__(self):
        return "X coordinate: {} \ nY coordinate: {} ". format (sel f .x, sel
f .y)

    def __len__(self):
        distance_from_center = ((self.x) ** 2 + (self.y) ** 2) ** 0.5
        return int (distance_from_center)

    def __eq__(self, other):
        return ((self.x, self.y) == (other.x, other.y))
```

In this case, we overload the *__eq__()* method by comparing two tuples. If the two tuples, each containing the x and y coordinates of points, are equal, then our two points are equal. We can use this as follows:

```
>>> from mytime import Point
>>> p1 = Point( 2, 2)
>>> p2 = Point( 2, 2)
>>> p1 == p2
True
>>> p3 = Point( 2, 3)
>>> p1 == p3
False
```

We could also add two points together to get a new point. In this case we would overload the *__add()__* method. The *__add__()* method implements the functionality of the + operator, not the += operator. Therefore, we must return a new object as the + operator is not for in-place addition.

Let's look at that:

```
class Point:
    def __init__ (self , x =0, y =0):
        self .x = x
        self .y = y

    def __str__ (self ):
        return "X coordinate: {}\nY coordinate: {} ". format (self .x, sel
f .y)

    def __len__ (self ):
        distance_from_center = (( self .x) ** 2 + (self .y) ** 2) ** 0.5
        return int (distance_from_center)

    def __eq__(self , other):
        return (( self .x, self .y) == (other.x, other.y ))

    def __add__(self , other):
        new_x = self .x + other.x
        new_y = self .y + other.y
        return Point(new_x, new_y)
```

Now we can use this method as follows:

```
>>> from my_point import Point
>>> p1 = Point( 2, 3)
>>> p2 = Point( 5, 2)
>>> p3 = p1 + p2
>>> print (p3)
X coordinate: 7
Y co ordinate: 5
```

We can overload the += operator, which is for in-place addition. This way we don't need to return a reference to a new point object and assign that to a new variable *p3*, instead we update the object's (*p1* in this case) data attributes.

This is done by overloading the __*iadd()*__ method.

```
class Point:
    def __init__ (self , x =0, y =0):
        self .x = x
        self .y = y

    def __str__ (self ):
        return "X coordinate: {} \ nY coordinate: {} " . fo rmat ( self .x, sel
f .y)

    def __len__ (self ):
        distance_from_center = (( self .x) ** 2 + (self .y) ** 2) ** 0.5
        return int (distance_from_center)

    def __eq__(self , other):
        return (( self .x, self .y) == (other.x, other.y))

    def __add__(self , other):
        new_x = self .x + other.x
        new_y = self .y + other.y
        return Point(new_x, new_y)

    def __iadd__ (self , other):
        temp_point = self + other
        self .x, self .y = temp_point.x, temp_point.y
        return sel f
```

We can now use the += operator on our points

```
>>> from my_point import Point
>>> p1 = Point( 2, 2)
>>> p2 = Point( 5, 2)
>>> p1 += p2
>>> print (p1)
X coordinate: 7
Y coordinate: 4
```

Be careful when overloading in-place operators. In this case we use create a temporary point and then update the *self* instance using multiple assignment. Remember when we swapped the value in one variable for another and we had to create a temp variable? We can do that much faster by doing `x, y = y, x`. In this case `x, y` and `y, x` are tuples!

Anyway, after we update the attributes for *self*, we must return *self*.

Below are a list of other operators you can overload.

Operator	Method that can be overloaded
==	eq()
!=	ne()
<	lt()
<=	le()
>	gt()
>=	ge()
+	add()
-	sub()
//	floordiv()
/	truediv()
*	mul()
**	pow()
%	mod()
+=	iadd()
-=	isub()
*=	imul()
//=	ifloordiv()
/=	itruediv()
**=	ipow()

There are more but we won't need any more than this, in fact we won't use half of these in this book.

What are instance methods?

We have already met instance methods. They are the type of methods we have been working with in regards to object oriented programming until this point.

Instance methods are the most common type of methods in classes. They are called instance methods as they act on specific instances of a class. They can access the data attributes that are unique to that class.

For example, if we have a *Person* class, then each instance of a person may have a unique name, age etc. Instance methods have *self* as the first parameter and this allows us to go *through self* to access data attributes unique to that instance and also other methods that may reside within our class.

I'm not going to provide an example for instance methods as we've seen them time and time again!

What are class methods?

Class methods are the second type of OOP method we can have. A class method knows about its class. They cannot access data that is specific to an instance, but they can call other methods.

Class methods have limited access to methods within the class and can modify class specific details such as class variables which we will see in a moment. This allows us to modify the class's *state* which will be applied across all instances of the class. They are one of the more confusing method types. Class methods are also permanently *bound* to its class.

We invoke class methods through an instance or through a class. Unlike instance methods whose first parameter is `self`, with class methods, its first parameter is not an object, but the class itself. This first parameter is called `cls`. We use a *decorator* to mark a method as a class method. The decorator is `@classmethod`.

Let's look at an example by going back to our `Time` class. We want a function that can convert seconds to 24 hour time with hours, minutes, and seconds. This method should be bound to the class of `Time` rather than a specific instance.

```
class Time:
    def __init__(self, hours=0, minutes=0, seconds=0):
        self.hours = hours
        self.minutes = minutes
        self.seconds = seconds

    def __str__(self):
        return "The time is {:02}:{:02}:{:02}".format(self.hours,
                                                      self.minutes,
                                                      self.seconds)

   # OTHER METHODS CAN GO HERE...

    @classmethod
    def seconds_to_time(cls, s):
        minutes, seconds = divmod(s, 60)
        hours, minutes = divmod(minutes, 60)
        extra, hours = divmod(hours, 24)
        return cls(hours, minutes, seconds)
```

For this example, I have cut out other instance methods that we don't care about for now. The seconds to time class method takes the class as the first parameter and some number of seconds, `s`, as its second parameter.

A side note on the divmod() function: The *divmod()* function takes two arguments, the first is the number of seconds we want to convert to 24 hour time, the second will be divided into it (60 in this case). This returns a tuple in which the first element is the number of times the second parameter divided into the first, and the second element of the tuple is the remainder after that division. For example, if we called *divmod(180, 60)*, we'd get back a tuple of (3, 0) in which we use multiple assignment to assign the 3 to the minutes and 0 to the seconds which is what we expect as 180 seconds is 3 minutes.

Back to our class method. When we have calculated our hours, minutes and seconds, we return *cls(hours, minutes, seconds)*. *cls* will be replaced with *Time* in this case. So really we're returning a new *Time* object.

To see this in action:

```
>>> from my_time import Time
>>> t = Time.seconds_to_time(11982)
>>> print(t)
'The time is 03:19:42'
```

I also mentioned class variables earlier. Much like before, *class variables* are bound to a class rather than a specific instance. We can use a class method to check the number of times an instance of the time class has been created for example.

With convention, class variables are usually all uppercase for the variable name. Let's modify our time class above to add a class method called *COUNT* which will count the number of times an instance of the class has been created.

```
class Time:

    COUNT = 0

    def __init__(self, hours=0, minutes=0, seconds=0):
        self.hours = hours
        self.minutes = minutes
        self.seconds = seconds
        Time.COUNT += 1

    def __str__(self):
        return "The time is {:02}:{:02}:{:02}".format(self.hours,
                                                      self.minutes,
                                                      self.seconds)

   # OTHER METHODS CAN GO HERE...

    @classmethod
    def seconds_to_time(cls, s):
        minutes, seconds = divmod(s, 60)
        hours, minutes = divmod(minutes, 60)
        extra, hours = divmod(hours, 24)
        return cls(hours, minutes, seconds)
```

Note how we have a new *COUNT* variable just before our constructor and inside our constructor, after initializing all the attributes, we increase the *COUNT* class variable by 1. Now we can see how many times instance of our class has been created.

```
>>> from my_time import Time
>>> t1 = Time(23, 11, 37)
>>> t2 = Time()
>>> Time.COUNT
2
>>> t3 = Time(12, 34, 54)
>>> Time.COUNT
3
```

It may be tricky trying to spot when to use these. As a general rule of thumb, if a method can be invoked (and makes sense to) in the absence of an instance, then it seems that, that method is a good candidate for a class method.

What are static methods?

Finally, we have a third method type that is called a *static method* These are not as confusing as class methods. You can think of static methods just like ordinary functions. Python will not automatically supply any extra arguments when a static method is invoked. Unlike *self* and *cls* with instance and class methods.

Static methods are methods that are related to the class in some way, but don't need to access any class specific data and don't need an instance to be invoked. You can simply call them as pleased.

In general, static methods don't know anything about class state. They are methods that act more as utilities.

We use the *@staticmethod* decorator to specify a static method. We will add a static method to our *Time* class that will validate the time for us, checking that the hours are within the range of 0 and 23 and our minutes and seconds are within the range of 0 and 59.

```
class Time:

    COUNT = 0

    def __init__(self, hours=0, minutes=0, seconds=0):
        self.hours = hours
        self.minutes = minutes
        self.seconds = seconds
        Time.COUNT += 1
    def __str__ (self ):
        return "The time is {:02} : {:02} : {:02} " . format ( self .hours,
                                                               self .minutes,
                                                               self .seconds)

  # OTHER METHODS CAN GO HERE...

    @classmethod
    def seconds_to_time(cls, s):
        minutes, seconds = divmod(s, 60)
        hours, minutes = divmod(minutes, 60)
        extra, hours = divmod(hours, 24)
        return cls(hours, minutes, seconds)


    @staticmethod
    def validate(hours, minutes, seconds):
        return 0 <= hours <= 23 and 0 <= minutes <= 59 and 0<= seconds
<= 59
```

This static method is simply a utility that allows us to verify that times are correct. We may use this to stop a user from creating a time such as `35:88:14` as that wouldn't make any logical sense.

```
>>> from my_time import Time
>>> Time.validate( 5, 23, 44)
True
>>> Time.validate( 25, 34, 63)
False
>>> t = Time( 22, 45, 23)
>>> t.validate(t.hours, t.minutes, t.seconds)
True
```

It may also be tricky to spot when to use static methods, but again, if you find that you have a function that may be useful but it won't be applicable to any other class yet at the same time it doesn't need to be bound to a instance and it doesn't need to access or modify any class data then it is a good candidate for a static method.

Public, Private & Protected attributes

In object-oriented programming, the idea of access modifiers is important. It helps us implement the idea of Encapsulation(information hiding). Access modifiers tell compilers which other classes should have access to data attributes and methods that are defined within classes. This stops outside classes from calling methods or accessing data from within another class, or making those data attributes and methods public, in which case all other classes can call and access them.

Access modifiers are used to make code more robust and secure by limiting access to variables that don't need to be accessed by every (or maybe they do in which case they're public).

Until now we have been using *public attributes and methods*. Classes can call the methods and access the variables in other classes.

In Python, there is no strict checking for access modifiers, in fact they don't exist, but Python coders have adopted a convention to overcome this.

The following attributes are public:

```
def __init__(self, hours, minutes, seconds):
    self.hours = hours
    self.minutes = minutes
    self.seconds = seconds
```

By convention, these are accessible by anyone (by anyone I mean other classes).

To "make" these variables *private* we add a double underscore in-front of the variable name. For an attribute or method to be private, means that only the class they are contained in can call and access them. Nothing on the outside. In the Java language, if you had two classes and tried to call a private method from another class you would get an error.

```
def __init__(self, hours, minutes, seconds):
    self.__hours = hours
    self.__minutes = minutes
    self.__seconds = seconds
```

The third access modifier I'll talk about is *protected*. This is the same as private except all subclasses can also access the methods and member variables.

```
def __init__(self, hours, minutes, seconds):
    self._hours = hours
    self._minutes = minutes
    self._seconds = seconds
```

What is inheritance?

In programming we often come across objects that are quite similar or we may see that one class is a type of another class. For example, a car is a type of vehicle. Similarly, a motorbike is a type of vehicle. We can see a relationship emerging here. The type of relationship is an "is a" relationship.

In object-oriented programming we call this inheritance and inheritance helps us model an is a relationship. Inheritance is one of the pillars of object-oriented programming.

There is some important terminology around inheritance that we must clarify before we look any deeper into it. Let's go back to our example of cars and motorbikes being types of vehicles. If we look at each of these as a class, we will call the vehicle class a base class or parent class and the car and motorbike classes, derived classes, child classes or sub classes (they may be referred to differently depending on what you're reading, just know they are exactly the same thing).

We can look at inheritance in programming the same way we do in real-life where a child inherits characteristics from its parents, in addition to its own unique characteristics. In programming, a child class can inherit attributes and methods from its parent class.

Parent & child classes

We can create a parent which will act as a base from which subclasses can be derived. This allows us to create child classes through inheritance without having to rewrite the same code repeatedly.

Let's say we have *Rectangle* and *Square* classes. Many of the methods and attributes between a *Rectangle* and *Square* will be similar. For example both a rectangle and a square have an *area()* and a *perimeter()*. Up until now we may have implemented these classes as follows:

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * self.length + 2 * self.width

class Square:
    def __init__(self, length):
        self.length = length

    def area(self):
        return self.length * self.length

    def perimeter(self):
        return 4 * self.length
```

And we would have used them as follows:

```
>>> rectangle = Rectangle(2, 4)
>>> square = Square(2)
>>> square.area()
4
>>> rectangle.area()
8
```

This seems a little bit redundant however as both a square and rectangle have 4 sides each and they both have an area and a perimeter. If we look more closely at this situation, we notice that a square is a special case of a rectangle in which all sides are the same length.

We can use inheritance to reflect this relationship and reduce the amount of code we must write. In this case a *Square* is atype of *Rectangle* so it makes sense for a square to inherit from the rectangle class.

Let's look at this code again but this time I have made changes to the square class to reflect this relationship. There are a few new things going on here but I'll explain after.

```
class  Rectangle:
    def  __init__  ( self  , length, width):
        self  .length    =  leng th
        self  .width    =  width

    def  area( self  ):
        return   self  .length    *  self  .width

    def  perimeter(   self  ):
        return   2  *  self  .length    +  2  *  self  .width

class  Square(Rectangle):
    def  __init__  ( self  , length):
        super ().  __init__   (leng  th, length)
```

The first change here is when we define the name of the *Square* class. I have *class Square(Rectangle)*. This means that the *Square* inherits from the *Rectangle* class.

The next change is to the square's *__init__()* method. We can see we are calling a new *super()* method. The *super()* method in Python returns a proxy object that delegates method calls to a parent or sibling of *type*. That definition may seem a little confusing but what that means in this case is that we have used *super()* to call the *__init__()* method of the *Rectangle* class, which allows us to use it without having to re-implement it in the *Square* class. Now, a squares length is *length* and its width is also *length* as we passed length in twice to the call to the *super()*'s *__init__()* method.

Since we've inherited *Rectangle* in the *Square* class, we also have access to the *area()* and *perimeter* methods without having to redefine them. We can now use these classes as follows:

```
>>> rect   = Rectangle(  2,   3)
>>> square  = Square( 2)
>>> print   (rec  t.area())
6
>>> print   (square.area())
4
```

```
>>> print (rect.perimeter())
10
>>> print (square.perimeter())
8
```

As you can see, the *super()* method allows you to call methods of the superclass in your subclass. The main use case of this is to extend functionality of the inherited method.

With inheritance we can also override methods in our child classes. Let's add a new class called a *Cube*. The cube will inherit from Square (which inherits from Rectangle). The *Cube* class will extend the functionality of the area method to calculate the surface area of a cube. It will also have a new method called *volume* to calculate the cubes volume. We can make good use of *super()* here to help us reduce the amount of code we'll need.

Let's look at how we do this:

```
class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length, length)

class Cube(Square):
    def area(self):
        side_area = super().area()
        return 6 * side_area

    def volume(self):
        side_area = super().area()
        return side_area * self.length
```

We can now use this class as shown:

```
>>> cube = Cube(2)
>>> print (cube.area())
24
>>> print (cube.volume())
8
```

As you can see, we haven't had to override the *__init__()* method as it's inherited from *Square*. We have however overridden the *area()* method from the parent class (Square) on the *Cube* to reflect how the area of a cube is calculated. We have also used the *super()* method to help us with this.

We have also extended the functionality of cube to include a *volume()* method. Again, we have used the *super()* method to help with this.

As you can see, by using inheritance we have greatly reduced the amount of code compared to how we would have implemented these three classes previously.

I just want to go back to overriding the *__init__()* method for a second. Let's assume a *Person* class and an *Employee* class that is derived from *Person*. Everything about an employee is the same as a *Person* but the employee will also have an employee ID.

Let's look at how we would implement these:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def show_name(self):
        print("Name: " + self.name)

    def show_age(self):
        print("Age: " + str(self.age))

class Employee(Person):
    def __init__(self, name, age, employeeId):
        super().__init__(name, age)
        self.employeeId = employeeId

    def show_id(self):
        print("Employee ID: " + str(self.employeeId))
```

We can now use these as follows:

```
>>> person = Person("John", 32)
>>> emp = Employee("Simon", 25, 4532245)
>>> person.show_name()
Name: John
>>> person.show_age()
Age: 32
>>> emp.show_name()
Name: Simon
>>> emp.show_age()
Age: 25
>>> emp.show_id()
Employee ID: 4532245
```

The reason I'm showing you this is because, although I've already shown how to override methods, people get confused when overriding the *__init__()* method to also include new attributes on the derived class.

Multiple inheritance

So far, we have looked a *single inheritance*That is, child classes that inherit from a single parent class. Multiple inheritance is when a class can inherit from more than one parent class.

This allows programs to reduce redundancy, but it can also introduce a certain amount of complexity as well as ambiguity. If you plan on doing multiple inheritance in your programs, it should be done with care. Give thought to your overall program and how this might affect it. It may also affect the readability of your programs. For example:

```
class   A:
    def   method():
        # your code here

class   B:
    def   method():
        # your code here

class   C(A, B):
    pass

>>> C.method()
```

Which *method* is the *C* class referring to? The method from A or the method from B? This may make your code harder to read. In Python, when calling *method()* on C, Python first checks if C has an implementation of *method* then it checks if A has implemented *method*. It has in this case so use that. If it had not then Python would check each parent class in turn (A, B, ……) and so on until it finds the first parent class to have implemented method.

I'm not going to go too deep into multiple inheritance as it's quite self-explanatory on how to use it, so I'm only going to give one example of how it can be used:

```
class   CPU:
    def   __init__   ( self   , num_register     s):
        self   .num_registers      =  registers
    # Other methods

class   RAM:
    def   __init__   ( self   , amount):
        self   .amount   =  amount
    # Other methods

class   Computer:
    def   __init__   ( self   , num_registers, amount):
        CPU.__init__   (num_reg isters)
        RAM.__init__   (amount)
    # Other methods
```

This can be used as follows:

```
>>> comp = Computer( 16,   32)
>>> print   (comp.num_registers)
16
>>> print   (comp.amount)
32
```

Recursion

What is recursion?

Before I begin, I want to say this: writing a recursive solution is DIFFICULT when starting out. Don't expect it to work first time. Writing a recursive solution involves a whole new way of thinking and can in fact be quite off putting to newcomers but stick with it and you'll wonder how you ever found it so difficult. Within the next few chapters I have confidence you'll become comfortable with recursion as we'll get a lot of practice with the technique.

So, what exactly is recursion? A *recursive* solution is one where the solution to a problem is expressed as an operation on a *simplified* version of the *same* problem.

Now that probably didn't make a whole lot of sense and it's very tricky to explain exactly how it works so let's look at a simple example:

```
def reduce(x):
    return reduce(x - 1)
```

So, what's happening here? We have a function called reduce which takes a parameter *x* (an integer) and we want it to reduce it to '0'. Sounds simple so let's run it when we pass 5 as the parameter and see what happens:

```
>>> reduce(5)
RuntimeError: maximum recursion depth exceeded
```

An error? Well lets take a closer look at what's going on here:

reduce(5) calls *reduce(4)* which calls *reduce(3)*......Thus our initial call *reduce(5)* is the first call in an infinite number of calls to *reduce()*. Each time *reduce()* is called, Python instantiates a data structure to represent that particular call to the function. This data structure is called a *stack frame*. A stack frame occupies memory. Our program attempts to create an infinite number of stack frames which would require an infinite amount of memory which we don't have so our program crashes.

To fix this problem we need to add in something referred to as the *base case*. This is simply a check we add so that our function knows when to stop calling itself. The base case is normally the simplest version of the original problem and one we always know the answer to.

Lets fix our error by adding a base case. For this function we want it to stop when we reach 0.

```
def reduce(x):
    if x == 0:
        return 0
    return reduce(x - 1)
```

So now when we run our program we get:

```
>>> reduce ( 5)
0
```

Great it works! Let’s take another look at what is happening this time. We call reduce(5) which calls reduce(4)….which calls reduce(0). Ok stop here, we have hit our base case. Now what the function will do is return 0 to the previous call of reduce(1) which returns 0 to reduce(2)….which returns 0 to reduce(5) which returns our answer. There you go, your first recursive solution!

Let’s look at a more practical example. Let’s write a recursive solution to find N! (or N factorial):

N factorial is defined as: N! = N * (N-1) * (N-2)…… * 2 * 1.

For example, 4! = 4 * 3 * 2 * 1

Our base case for this example is N = 0 as 0! is defined as 1. So lets write our function:

```
def  factorial(n):
     if   n  ==  0:
          return   1
     return    n  *  factorial(n     - 1)
```

Okay, let’s see that in action. We will call our function and pass 4 in for n:

factorial(4) calls factorial (3) ….. which calls factorial(0) which is our base case so we return 1 back to the previous call of factorial(1) which takes our 1 and multiplies it by 1 which evaluate to 1 which passes that back to factorial(2) which takes our 1 and multiplies it by 2 which evaluates to 2 which passes that back to factorial(3) which multiplies 3 2 to get 6 which passes that back to factorial(4) which multiplies 6 4 which evaluates to 24 which is returned as our answer.

These are very simple cases and not super useful but being able to spot when a recursive solution may be implemented and then implementing that solution is a skill that will make you a better programmer. For certain problems recursion may offer an intuitive, simple, and elegant solution.

For example, you may implement a recursive solution to find the number of nodes in a tree structure, count how many leaf nodes are in a tree or even return whether a binary search tree is AVL balanced or not. These solutions tend to be quite short and elegant and take away from the long iterative approach you may have been taking.

As I have said, recursion isn’t easy but stick with it, it will click and seem so simple you’ll wonder how you ever found it so difficult.

Memoization

We have written the solution to the Fibonacci problem a couple of times throughout this book. When writing those solutions, we've used an iterative approach. We can, however, take a recursive approach. It may be difficult to recognize when a recursive solution is an option, but this is a great example of when a recursive solution is an option!

Let's look at a solution to the following problem:

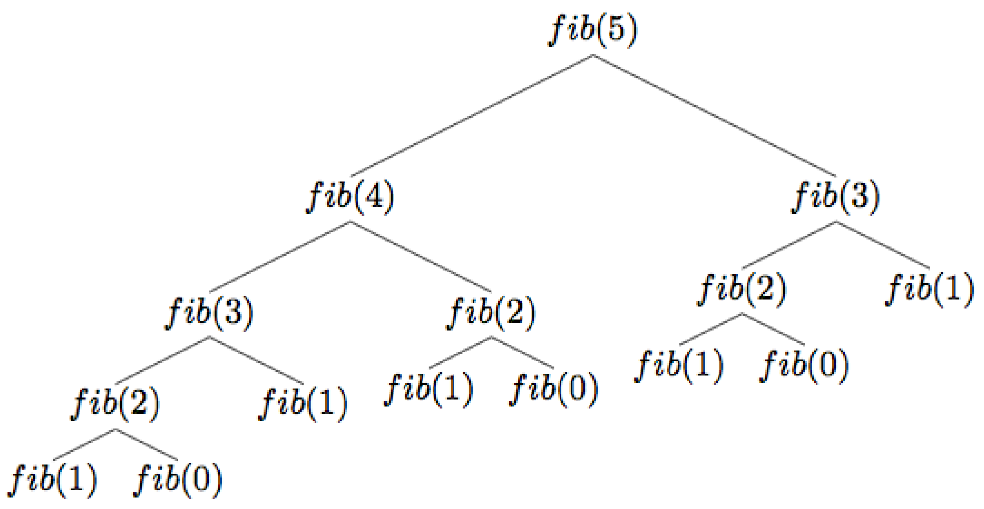
Write a program that takes an integer *n* and computes the n-th Fibonacci number:

```
def  fib(n):
    if  n <= 1:
        return  1
    else :
        return  fib(n  - 1)  + fib(n  - 2)
```

That’s much simpler! In this example we'll take fib(0) to be 1 and fib(1) to be 1 also. Therefore our base case is if n is 0 or 1 we return 1. The recursive case comes straight from the definition of the Fibonacci sequence, that is:

̈Ω̈Ωఴ سي̈Ω̈Ωఴطي ρ طى̈Ω̈Ωఴطي ς

However, you may have noticed that if you input anything greater than (roughly) 35 - 40 your program begins to really grind to a halt and take a VERY long time and anything above 45~50, well... go make a cup of coffee and come back and it MIGHT be finished. What's causing this? Let's look at a diagram of the first few recursive calls to *fib()*.



We can see that when we call *fib(5)* we, at various points in our functions execution, call *fib(1)* 5 times! and *fib(2)* 3 times! This is unnecessary. Why should we have to recompute the value for *fib(1)* over and over again?

We can reduce the number of times we need to calculate various values by using a technique called *memoization*. Memoization is an *optimization* technique used to speed up our programs by *caching* the results of expensive function calls. A *cache* is essentially a temporary storage area and we can implement a cache in our *fib()* function to store previously calculated values and just pull them out of the cache when needed rather than making unnecessary function calls.

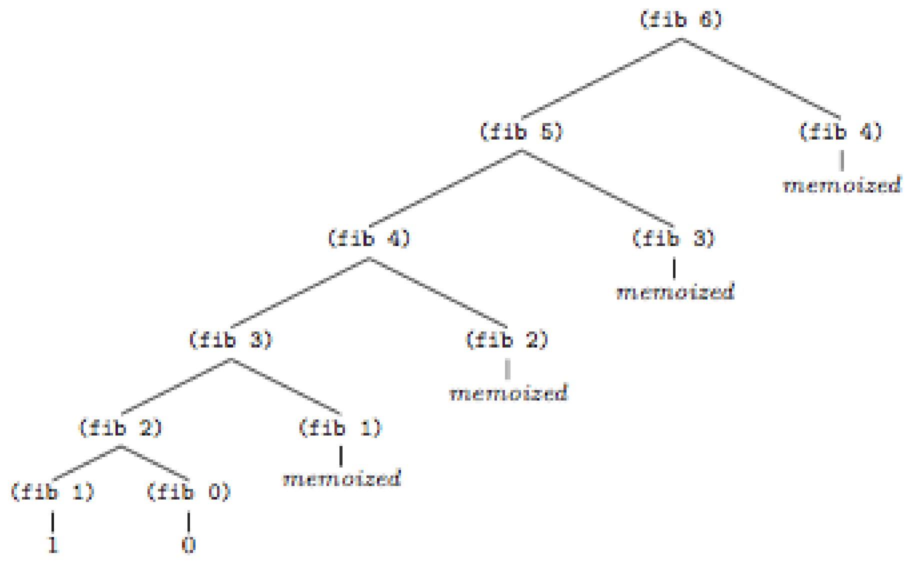
We can use a dictionary to implement a cache. This will give us O(1) for accessing and will drastically improve the performance of our function.

Let's look at how that's done:

```
def fib(n, cache=None):
    # Avoid default mutable argument trap!
    if cache == None:
        cache = {}
    # Base case
    if n <= 1:
        return 1
    # If value not in cache then calculate it and store in cache
    elif not n in cache:
        cache[n] = fib(n - 1, cache) + fib (n - 2, cache)
    # Return the value
    return cache[n]
```

We can now call *fib(900)* and get the answer straight away! By default, Python limits the recursion depth to 1000. We can override this but it's usually not a good idea!

Let's look at our tree now, this time for *fib(6)*.



We can see from the tree that there are less calls to the function even though we are a value higher. This is a significant performance increase. Even calling from *fib(5)* we have saved ourselves 6 function calls!

That's all I want to say on recursion for now. This chapter is just a brief introduction to the technique and the purpose of recursion may not seem obvious at the moment, however when we look at data structures such as Binary Search Trees, its use will become a lot more clear.

Stacks & Queues

The backbone of every piece of software you will write will consist of two main things: data and algorithms. We know that algorithms are used to manipulate the data within our software (and that manipulation should be done as efficiently as possible). For this reason, it is important to structure our data so that our algorithms can manipulate the data efficiently.

We have met various data structures already, lists and dictionaries to name a few. You also know that using lists for example, makes it very easy for us to store and manipulate sequences of data (Trust me, without this abstraction it becomes a bit of a pain).

We have also seen that data structures allow us to model systems that we see in the real world. For example, we could design and build a class that allows us to model a library. We could easily use this to build a library management system.

In this chapter we're going to look at two fundamental data structures that are used everywhere in computing. We're also going to look at some examples of how to take advantage of them.

Stacks

The stack is one of the most fundamental data structures in computing. Every time you run a program, that program takes advantage of a stack. I'll first explain how it works, then I'll explain how your computer takes advantage of it.

A stack is a linear data structure that follows a particular order in which the operations are performed. A stack is a LIFO (Last in First Out) data structure. That is, the last element entered into the stack is the first element that will leave the stack. You can think of a stack as a stack of dinner plates. Plates are stacked, one on top of the other. The last plate to be put on top of the stack will be the first plate to be removed as we cant remove the plate at the bottom (or the stack of plates would fall over and they'd all smash).

A stack (usually) has 4 main operations: *`push()`* (add an element to the top), *`pop()`* (remove an element from the top), *`top()`* (show the element at the top) and *`isEmpty()`* (is the stack empty). We can also add a length method and it is usually useful!

Here is an illustration of a stack

```
                            | ------   | <  --   Pop            | ------   |  <--   Push new
| ------   | <  --   Top                                        | ------   |        element (new
| ---  ---  |                | ------   | <  --   New top        | ------   |    top)
| ------   |    If we pop: |    ------   |  If we push           | ------   |
| ------   |                | ------   |  a new element:        | ------   |
| ------   |                | ------   |                        | ------   |
| ------   |                | ------   |                        | ------   |
| ------   |                | ------   |                        | ------   |
```

The stack is such a fundamental data structure that many computer instruction sets provide special instructions for manipulating stacks. For example, the Intel Pentium processor implements the x86 instruction set. This allows for machine language programmers to program computers at a very low level (Assembly language for example).

A very special type of stack called the "Call stack" or "Program stack", usually shortened to "The stack" is what some of the instructions in the x86 instruction set manipulate.

Let's consider what happens with the following program at a low level (For anyone with a solid understanding of computers this is a little high level but a good example).

```
    .
    .
    .
print(4)
    .
    .
    .
```

In the above Python snippet, we have some set of instructions executing in order (Like we have always seen). At some point we call the print function to output the number 4 to our terminal window. This is where the stack comes into play. Lets look at how our program might be stored in memory (At a high level)

```
Memory Address      Instruction
------------------         ----------------
0x0000001          .......      # The interal code for the print

                                                  #  function might
0x0000002          .......      # be stored at 0x0...01 to 0x0...03
.
0x0000003                   .......
      .
      .
      .
0x0000004          .......      # Some instruction before the print
.
```

```
0x0000005          print(4)     # The call to print.
0x0000006          .......     # Some instruction after print (The
                                          # next instruction to be executed).
```

.

(I'M AWARE THIS ISN'T WHAT ACTUALLY HAPPENS BUT LET'S GO WITH THIS FOR NOW).

When we run a program, it is loaded into memory as machine code (Code the computer knows how to execute). Instructions are stored sequentially. In this view, we can look at functions as "mini programs" inside our main program. That is, the code for them is stored elsewhere in memory. When we call *print(4)* we are basically saying, jump to where the code for print is and start sequentially executing instructions until the function has completed, then return to the original location and continue on executing where you left off.

Before I explain this, the CPU has many special pieces of memory within it, called registers (These are like tiny pieces of RAM, usually 32-bits or 64-bits in size). One of these registers is called the instruction pointer register often abbreviated to *IP*. It contains the memory address of the next piece of code to execute.

In this case we execute the instruction at *0x0000004* (then increase the IP), in the next instruction we call *print(4)* which is at location *0x0000005* yet the code for the print function is stored at *0x0000001* to *0x0000003*. This is where the stack comes into play. We know that when we print the number we want to continue where we left off and execute the instruction at *0x0000006*. In this case we push the return address(*0x0000006*) onto the stack, set the IP to *0x0000001* and start executing the code for the print function. When we're finished, we pop the return address off the stack (*0x0000006*) and place this in the IP. Now our program continues to execute where we left off.

I'm sure you can imagine how recursion works now at this level (essentially a series of push, push, push ...... pop, pop, pop).

Things are a lot more complex than this, but it isn't necessary for the explanation of how stacks are used at the most fundamental levels. Do not worry too much if you didn't understand all of that. It isn't necessary for this book, but it shows just how important the stack is.

Stacks have many high levels uses too, for example, *undo* mechanisms in text editors in which we keep track of all text changes in a stack. They may be used in browsers to implement a *back/forward* mechanism to go back and forward between web pages.

Let's look at the code for a stack in Python. We will implement a stack using a list

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

    def push(self, elem):
        self .stack.append(elem)

    def pop(self ):
        if len (self .stack) == 0:
            return None
        else :
            return self .stack.pop()

    def isEmpty( self ):
        if len (self .stack) == 0:
            return True
        return False

    def top( self ):
        return self .stack[ - 1]
```

This is a very simple version of a stack, but we can now use our stack as follows:

```
stack = Stack()

stack.push( 1)
stack.push( 4)
stack.push( 2)

print ( stack.top())

print (stack.pop())
print (stack.pop())
print (stack.pop())
```

We will get the following output if we run this program.

```
2 # top
2
4
1
```

When we call the *top()* method, we don't actually remove the top element, we are just told what it is. When we use *pop()* method, we do remove the top element.

Stack Applications

Matching Brackets Problem

The matching brackets problem is a classic example of where the use of stacks makes the solution to the problem very straight forward. The problem is: Given an input string consisting of opening and closing parentheses, write a python program that outputs whether the string is balanced.

For example:

Input: ([{}])
Output: Balanced

Input: (([])
Output: Unbalanced

Input: {(){[]}}
Output: Balanced

Input: [()(){]
Output: Unbalanced

One approach we can take with this problem is to use a stack. Each time we encounter an opening bracket we append that to the stack. When we encounter a closing bracket we pop from the stack and check that the element we popped is the opening bracket for the closing bracket. For example, *[* is the opening bracket for *]*, similarly, *{* is the opening bracket for *}* and so on.

Here is a possible solution. Assume our stack class from earlier exists:

```
opening = ["(", "[", "{"]
# Note that pairs is a dictionary that maps the closing bracket
# to the opening bracket
pairs = {")" : "(", "]" : "[", "}" : "{"}

def balanced(input_string):
    stack = Stack()
    for bracket in input_string:
        if bracket in opening:
            stack.push(bracket)
        elif stack.isEmpty():
            return False # Can't pop anything as stack is empty so it'
s unbalanced
        elif pairs[bracket] != stack.pop():
            return False # The brackets didn't match up so it's unbala
nced

    return not stack.isEmpty()      # If stack is empty at the end,
                                    # return True, else, False
```

We can make better use of the pairs map and make our code a bit neater but it won't be as readable so I've taken a slightly longer approach.

Now we can run our code as follows to get the desired output from the example input and output above:

```
# ABOVE CODE HERE

inputString = input()
if (balanced(inputString)):
    print("Balanced")
else:
    print("Unbalanced")
```

Queues

Queues in some sense are similar to stacks. They are a linear data structure except rather than the operation order being Last in First Out, they are FIFO (First in First Out). They behave exactly like any queue you may think of in real life. A queue at a supermarket for example, you get in line at the back and wait until you get to the front to leave.

In computing, a queue may be useful in CPU scheduling. In computers, there are many things happening at what seems to be, the same time. For example, you may have music playing as you read an article. There are many things happening here. Your music is being played, your monitor is being updated (refreshed) as you scroll through your article and you are using your mouse wheel to indicate when to scroll. In a simple sense, your CPU can only deal with one thing at a time. Your music application needs time on the CPU to output the next bit of music, as you scroll while this is going on, your mouse driver needs CPU time to tell the computer what to do, then your monitor drivers need CPU time to refresh the pixels on your screen. This all seems to be happening at the same time, except it is not (computers are just so fast it appears it is). What is actually happening in this situation is each program is quickly using the CPU, then jumping back in the queue. This is happening over and over again. A CPU scheduling algorithm deals with which program gets to go next on the CPU and may implement some variation of a queue to handle this. (A priority queue for example).

They have similar methods for adding and removing except they are *enqueue()* and *dequeue()* respectively.

I think you get the gist of how you may visualize a queue so I'm just going to jump to the code for implementing a queue. Again, we will implement a queue using a list.

```
class Queue:
    def __init__(self):
        self.q = []

    def enqueue(self, elem):
        self.q.append(elem)

    def dequeue(self):
        if len(self.q) != 0:
            return self.q.pop(0)

    def isEmpty(self):
        if len(self.q) > 0:
            return False
        return True
```

And there you go. A very simple Queue class. Notice how we pop from index 0. Recall that the *pop()* method for lists takes an optional argument which is the index of the list you want to remove an element from. If we don't supply this optional argument it defaults to -1 (the end of the list, which is what stacks do). Here we remove the element at the start of the list but add elements to the end.

Running the following program, will produce the following output

```
queue = Queue()

queue.enqueue(1)
queue.enqueue(4)
queue.enqueue(2)

print(queue.isEmpty())

print(queue.dequeue())
print(queue.dequeue())
print(queue.dequeue())

print(queue.isEmpty())
```

```
False
1
4
2
True
```

Queue Applications

The most obvious example of the uses of Queue data structures is in media players. Take the example of adding songs to a queue in an application such as Spotify. In this section we'll take a look at creating a queueing system such as the one that exists in Spotify. Firstly, we'll need a catalogue of songs to be able to add to the queue. In this example I'll represent the songs as a dictionary that contains the metadata of songs and the download links the media player will use to download the song once they've been taken out of the queue:

```
catalogue = {
    100: {
        "track_name": "Smells Like Teen Spirit",
        "album_name": "Nevermind",
        "artist_name": "Nirvana",
        "download_link": "https://www.myapp.com/track/100",
        "length": 5
    },
    101: {
        "track_name"  :  "Bohemian Rhapsody" ,
        "album_name":  "A Night At The Opera"     ,
        "artist_name"    :  "Queen",
        "download_link"    :  "https://www.myapp.com/track/101"       ,
        "length"   :  8
    },
    102: {
        " track_name"  :  "Billie Jean"      ,
        "album_name":  "Thriller"      ,
        "artist_name"    :  "Michael Jackson"    ,
        "download_link"    :  "https://www.myapp.com/track/102"       ,
        "l  ength"  :  3,
    },
    103: {
        "track_name"  :  "Imagine"  ,
        "album_name":  "Imagine"  ,
        "artist_name"    :  "John Lennon"  ,
        "download_link"    :  "https://www.myapp.com/track/103"       ,
        "length"   :  5,
    },
    104: {
        "track_name"  :  "Stairway To Heaven"    ,
        "album_name":  "Led Zeppelin IV"      ,
        " artist_name"    :  "Led Zeppelin"    ,
        "download_link"    :  "https://www.myapp.com/track/104"       ,
        "length"   :  7,
    }
}
```

Now that we have our catalogue, we can add songs to the queue based on their IDs. For the purposes of this example, we'll be implementing functions that don't really do anything such as the function to download the next song. Also, for the purposes of keeping things short we'll assume the code for the Queue class already exists. To implement the media player queue, we will use inheritance so that our TrackQueue class can inherit from the Queue class. The TrackQueue class will be used to queue track objects from our catalogue.

```
class   TrackQueue(Queue):
    def  __init__   ( self  ) :
        super (TrackQueue,   self  ).  __init__   ()
```

We made a call to *super* to allow the TrackQueue class to use methods from the Queue class. Next we'll define an *add_track()* method that will allow us to add songs to the queue.

```
class   TrackQueue(Queue):
    def  __init__   ( self  ):
        super (TrackQueue,    self  ).  __init__   ()

    def   add_track(   self   , track):
      self   .enqueue(track)
```

Next we'll implement the *cycle()* method that will download the next song, play it, and repeat. We'll also implement the *play()* method that will simulate playing a song, and the *download()* method that will simulate downloading a song. The length of the songs are defined in the metadata objects in the catalogue and they don't reflect the actual length of the songs but we're using them here so we can see our code in action.

```
import    time

class    TrackQueue(Queue):
    def   __init__   ( self   ):
        super (TrackQueue,    self  ).  __init__   ()

    def   add_track(   self   , track):
      self   .enqueue(track)

    def   cycle(   self   ):
        while    self   .q.count     >  0:
            current_track_metadata       =  self   .deque()
            current_track       =  self   .download(current_track_metadata)
            self   .play(current_track)

    def   download( self   , track_data):
        print   ( "Downloading    {}    from    {}  " . format  (track_da   ta[  "track_name"   ]
, track_data[      "download_link"     ]))
        # For the purposes of this example, we can just return the met
adata
        # but in reality we would actually be downloading the audio fi
le and returning it
        return     track_data

    def   pla y( self   , track):
        print   ( "Playing     {}    by  {}    fr  om album  {} \ n" . format  (track[    "track_n
ame"], track[      "artist_name"     ], track[      "album_name"]))
        # We sleep here to simulate the song playing
        time.sleep(track[       "length"     ])
```

We can now use our TrackQueue class as follows:

```
tq   =  TrackQueue()
# The user adds three songs to the queue
tq.add_track(catalogue[       100])
tq.add_track(catalogue[       101])
tq.add_track(catalogue[102])

# The user starts playing from the queue and songs
# should be played in the order they were added
tq.cycle()
```

Which should produce the following output:

```
Downloading Smells Like Teen Spirit from https://www.myapp.com/track/1
00
Playing Smells Like Teen Spirit by Nirvana from album Nevermind

Downloading Bohemian Rhapsody from https://www.myapp.com/track/101
Playing Bohemian Rhapsody by Queen from album A Night At The Opera

Downloading Billie Jean from https://www.myapp.com/track/102
Playing Billie Jean by Michael Jackson from album Thriller
```

Priority Queues

A priority queue is very similar to the regular queue we've just looked at. The major difference is that each element in the queue has a priority associated with it. When we remove an item from the queue, we are removing the item with the highest priority. In this chapter we will define highest priority as an element of the queue that is greater than all the other elements. For example, if our priority queue had two elements, 4 and 2, the element 4 would have highest priority and would be removed first, regardless of the order it was entered.

There are many ways to implement a priority queue, some of which are more efficient and others which are not. In this section we'll be looking at a basic implementation of a priority queue - the unsorted priority queue. This means that the queue is not ordered by priority, instead, we search for the item with the highest priority when removing an item. There are other implementations that sort the queue when adding an item such that the item with the highest priority is always at the front of the queue and other implementations that are based on specific types of Trees which are data structures, we will look at later on. However, the operations of a priority remain the same.

The items we add to a priority queue must be comparable, i.e. integers, floats, or strings. We can also add tuples. Python can compare tuples using the comparison operators. When Python compares two tuples, it compares the first elements of each tuple, if they are the same, for example the first elements are both integers and both of those integers are the same then Python will compare the second elements of both tuples and so on.

It is common practice to insert tuples into a priority queue where the first element of the tuple is an integer value for the priority and the second element is something else, perhaps an instance of a class.

I'd also like to note that in the case we are using instances of a class and that class does not override the comparison operators, we will need a third element in the tuple. This third element is usually the *id(c)* of the object. The Python built-in *id()* function returns the unique ID that Python assigned to the object when it was created. This is used kind of like a fail-safe so that we can always compare two tuples. If this is confusing to you, it will be given in examples after the code for the PriorityQueue class.

Below is the code for the class outline, the add method, and a method to check if the queue is empty for the priority queue:

```
class PriorityQueue:
    def __init__(self):
        self.queue = []

    def add(self, item):
        self.queue.append(item)

    def is_empty(self):
        return len(self.queue) == 0
```

Now we need to create the method to remove items from the queue. This works by comparing the elements of the queue to find the max:

```
class PriorityQueue:
    def __init__(self):
        self.queue = []

    def add(self, item):
        self.queue.append(item)

    def is_empty(self):
        return len(self.queue) == 0

    def remove(self):
        max = 0
        for i in range(len(self.queue)):
            if self.queue[i] > self.queue[max]:
                max = i
        item = self.queue[max]
        del self.queue[max]
        return item
```

We can now use our priority queue as follows:

```
pq = PriorityQueue()
pq.add((4, "Hello"))
pq.add((3, "this"))
pq.add((8, "is"))
pq.add(( 7, "a" ))
pq.add(( 2, "priority queue" ))

while not pq.is_empty():
    item = pq.remove()
    print (item)
```

As you can see, we've added items to the queue but the priority we assigned to those items is random (the priority is the first item in each tuple). However, the loop that removes items and prints them out gives the following:

```
(8, 'is')
(7, 'a')
(4, 'Hello')
(3, 'this')
(2, 'priority queue')
```

As you can see, the items are removed based on their priority. Back to what I mentioned earlier about needing a third parameter. Consider what happens if we have the below class:

```
class Thing:
    def __init__(self, value):
        self.value = value
```

This is just a toy class but, importantly it doesn't override the comparison operators so if we are removing items from the priority queue and two items have the same priority (and those items take the same form from the above examples), our program will crash:

```
pq = PriorityQueue()

t1 = Thing( 1)
t2 = Thing( 2)

pq.add(( 4, t1))
pq.add(( 4, t2))

while not pq.is_empty():
    item = pq.remove()
    print (item)
```

This will give the below error:

```
TypeError: '>' not supported between instances of 'Thing' and 'Thing'
```

To me and you, as a *Thing* cannot be compared to another thing (as we didn't override the comparison operators), we would consider both of these items of equal priority (as both of their priorities is 4) so we don't care which one is removed first. To account for this, we add a third parameter to the tuple as shown below:

```
pq = PriorityQueue()

t1 = Thing( 1)
t2 = Thing( 2)

pq.add(( 4, id (t1), t1))
pq.add(( 4, id (t2), t2))

while not pq.is_empty():
    item = pq.remove()
    print (item)
```

Which will give the following output:

```
(4, 2441260187504, <__main__.Thing objec        t at 0x0000023866723F70>)
(4, 2441260185872, <__main__.Thing object at 0x0000023866723910>)
```

We can see the unique IDs that Python assigned each of these objects. When comparing items for selecting which to remove, we compared the tuples at the first indices, which were the same, so Python compared the items at the second index of the tuple - which can now be compared, and we are guaranteed that one will be less than the other.

The above problem is something you will need to consider when deciding how to add elements to the priority queue, and how those items can be compared.

Linked Lists

Singly Linked Lists

A linked list is a linear data structure where each element is a separate element. Linked List objects are not stored at contiguous locations. Instead, the linked list objects are linked together using pointers.

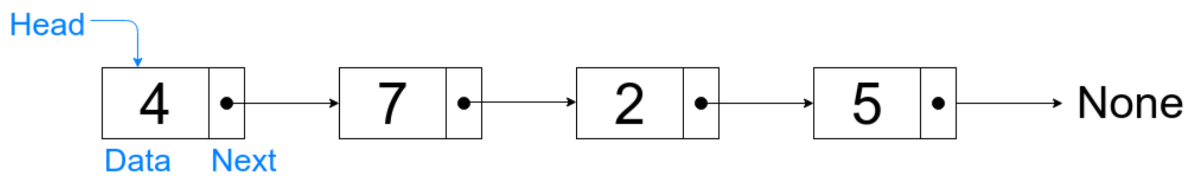
Throughout this book I may have referred to Python lists as arrays and vice versa. Python lists have a complicated implementation under the hood. This allows for lists to be of arbitrary size, grow and shrink and store many items of various types.

In many other programming languages, an array is much like a fixed size list that stores elements of a specific type. They store elements contiguously in memory. That isn't necessarily the case in Python.

Python is implemented in a language called 'C'. This language is low-level compared to Python. In the C language, Python lists are implemented with a data structure somewhat similar to a linked list. It's a lot more complicated than a linked list but a lot of the concepts and techniques we'll learn here will apply to the C Python implementation of lists.

There are many variations of linked lists but in this chapter, we're going to look at one variation called a Singly Linked List

Below is a diagram of a Singly Linked List:



This is a singly linked list as there is only a single link between the nodes in the list.

There are various elements to this so let me explain:

- Head: The head is the first element in the list

- Tail: The tail is not marked here but it’s the last element (5 in this case)

- Node: A node consists of two things; The data and a pointer to the next element (Next)

Linked Lists have various advantages over arrays (not Python lists). These include dynamic size (they can grow and shrink as needed much like python lists, whereas arrays can't do this) and ease of insertion and deletion.

When we delete something from a Python list, the list must be resized, and various other things must be done (this is done under the hood, so we don't worry about this).

There are also some disadvantages to Linked Lists compared to arrays. We can't index them, therefore if we wish to access an element we must iterate through each element, starting from the head, in order until we find the element we are searching for.

From the above diagram we can break the problem of implementing a linked list down into two things:

1. A node class: This will contain the data the node will store and a pointer to the next node.

2. Linked List class: This will contain the head node and methods we can perform on the linked list.

Let's look at the node class:

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

This is our linked list node class. It's very simple and only contains two attributes. The first is the data the node will store and a *next* attribute. The next attribute is set to None by default as it won't point to anything initially. This concept of a node is very powerful when it comes to data structures. It gives us a lot of flexibility when developing other data structures as we'll see later on.

Let's now look at the first attempt at out linked list:

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

When we initialize our LinkedList it will be empty, that is, the head will point to nothing (None). That's all we need to start implementing methods.

Adding to the Linked List

To get started, we'll take a look at the add() method. Firstly, we need to think about how we're going to go about doing this. In our linked list, we'll be adding new elements to the end of the list. Firstly, we need to check if the head is empty. If it is, then we just point the LinkedList head to a new node. Otherwise, we need to "follow" this trail of pointers to nodes until one of the node's next attribute points to None. When we find this node then we update it's next attribute to point to the element we're trying to add.

Let's look at how that's done:

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

    def   add(self, elem):
        if    self.head == None:
            self.head = Node(elem)
        else :
            curr = self.head
            while   curr.next     !=   None:
                curr = curr.next
            curr.next      = Node(elem)
```

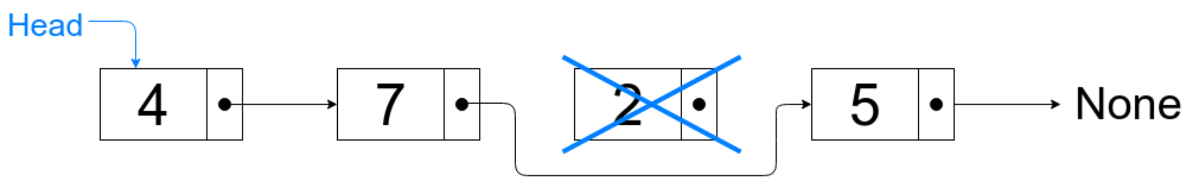
Here we check that the head of the linked list isn't empty. If it has then we assign a new `Node` to the linked list `head`. Otherwise, we create a new variable called `curr`. This keeps track of what node we're on. If we didn't do this, we would end up updating the linked list's `head` node by mistake. After we create this new variable, we loop through the elements of the list, going from node to node *through* the `next` pointer until we find a node whose `next` pointer is empty. When we find that node, we update it's `next` pointer to point to a new `Node`.

This may take a little time to wrap your head around or understand but the more you work with data structures like this or ones similar, the more sense it will make.

Deleting from the Linked List

Let's look at our deletion method. We have many options when deleting elements from a linked list just like we do with adding them. We could remove the first element, the last element or the first occurrence of an element. Since our linked list can hold many occurrences of the same data, we will remove the first occurrence of an element.

Let's look at how that may be done with the help of a diagram:



As you can see, from this diagram we want to delete the node that contains `2`. To do this we find the first node who's `next` pointer points to a `Node` that contains the value `2`. When we find this `Node` we just update it's `next` pointer to point to the same node the `Node` we want to delete points to. Essentially, we re-route the `next` pointer to bypass the element we want to delete. This is quite easy to do.

Let's look at how it's done.

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

    def delete(self, val):
        if self.head == None:
            return
        if self.head.elem == val:
            self.head = self.head.next
            r eturn val

        curr = self.head
        while curr.next != None and curr.next.elem != val:
            curr = curr.next

        if curr.next == None:
            return
        else :
            curr.next = curr.next.next
            r eturn val
```

We have to cover a couple of cases here. The first is that the list is empty in which case return nothing. The second is that the head of the linked list is the element we want to delete. If it is then we make the head of the list the second `Node` in the list (Which may be `None` but thats ok, it just means the list only had one item). Finally, if it is neither of those cases we traverse the list using linear search until we find the first occurrence of the element then we 're-route' the `next` pointer of the `Node` that points to the `Node` we want to delete, to the `Node` the `next` pointer of the `Node` we want to delete points to.

That last part might seem confusing but we're doing what you see in the diagram. It's probably best at this point if you're confused by that to draw this scenario out on paper to help clarify things.

There are a couple of other useful methods we can add here but we'll leave them for later.

Just before I finish up on linked lists, I want to talk a little bit more about the complexities of their methods and why you might use them (or not).

The run time complexity of the add method is O(n) in this case as we must iterate over each entry in the list to find the last element. The runtime complexity of the deletion method is also O(n). Although we don't always iterate over each element in the list, Big-O deals with the worst case, which is going through the entire list. There are optimizations we could make though so that the `add()` method is always O(1). In other words, it is constant.

Maintaining Efficiency

Currently, the `add()` method has a run-time complexity of O(n). We can change the implementation of the `add()` method so that it's run-time complexity is O(1). Our solution is to add the new element to the front of the linked list rather than the end.

Let's look at how this is done:

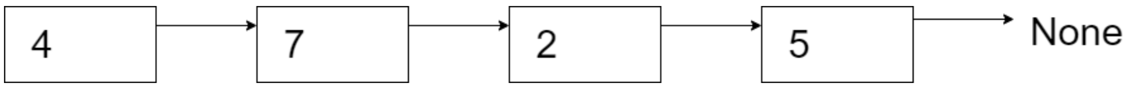
```
class   Node:
        def   __init__(self, elem):
                self.elem = elem
                self.next = None

class   LinkedList:
        def   __init__(self):
                self.head = None

        def   add(self, el      em):
                if    self.head ==      None:
                        self.head = Node(elem)
                else :
                        new_head = Node(elem)
                        new_head.next = self.head
                        self.head = new_head
```

The applications of linked lists are pretty much the same as the applications of Pythons built in List type and therefore Linked Lists in Python usually don't have a use case. This is because Python lists are already very well optimized and dynamic. However, in languages such as C or C++, Linked Lists are essential (Where dynamic arrays, like python lists, may not exist). They also pop up in technical interviews so you're best to know them. In fact, if I was hiring an engineer, I'd be very skeptical about hiring one who didn't know how to code a linked list.

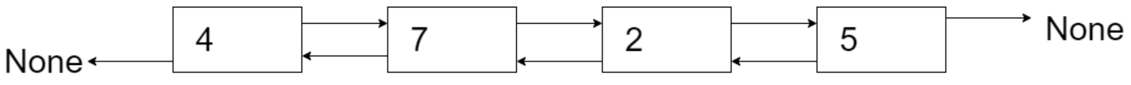
Advanced Linked Lists

Doubly Linked Lists

Now that we have a solid understanding of Singly Linked Lists and their uses, we can expand on the idea of them and take a look at Doubly Linked Lists. These are very similar to Singly Linked Lists but the difference is that there are also back links to previous nodes. To illustrate this, take a look at the basic structure of singly linked lists below:



Now take a look at the structure of a Doubly Linked List below:



We can see from the above diagram that in addition to the *next* attribute that existed on nodes in Singly Linked Lists we will also have a *previous* attribute that will point to the predecessor node. Much the same way the successor of the tail node in the Singly Linked List was *None*, the predecessor of the head node will be *None*.

The Doubly Linked List Class

Let's look at how to construct a Doubly Linked List. Firstly, let's start with the code for the Node class.

```
class Node:
    def __init__(self, elem, next=None, prev=None):
        self.elem = elem
        self.next = next
        self.prev = prev
```

In the above code, the *next* variable holds a reference to the next Node in the list while the *prev* variable holds a reference to the previous Node in the list.

Now we want to look at the class for a Doubly Linked List. This will have three attributes: the *head*, the *tail*, and an attribute called *Length* that will store the number of nodes in the list.

```
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.length = 0
```

Inserting into the Doubly Linked List

Again, appending to a doubly linked list is straight forward but is done differently to how we handled insertion into a Singly Linked List whereby we didn't track the tail node. There are two cases we need to handle. The first is when the list is empty, in which case the head and tail will be pointing to the same Node object. The second case is when the list is not empty, and we insert into the end of the list. In this case we will need to update the tail pointer to point at the new element.

Below is the code for inserting into a Doubly Linked List

```
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.length = 0

    def insert(self, elem):
        new_node = Node(elem, None, None)
        if self.head is None:
            self.head = new_node
            self.tail = self.head
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node
        self.length += 1
```

Deleting from a Doubly Linked List

When we handled deleting from a singly linked list we had to keep track of the previously visited node. The doubly linked list avoids the need for doing this due to the *prev* pointer on each Node in the list.

The Node we want to delete is identified when its data instance variable matches the *elem* variable matches the element that is passed to the method to be used in the search for the node. It's also worth noting that the deletion method will only delete the first occurrence of *elem*. We will also return *False* if the item to be deleted doesn't exist in the list and we return *True* if it did exist.

Below is the code for deleting an element from a doubly linked list:

```
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.length = 0
```

```
def delete(self, val):
    curr = self.head
    # List is empty, just return
    if curr is None:
        return False
    # Check the head
    elif curr.elem == val:
        self.head = curr.next
        self.head.prev = None
        self.length -= 1
        return True
    # Check the tail
    elif self.tail.elem == val:
        self.tail = self.tail.prev
        self.tail.next = None
        self.length -= 1
        return True
    # Search through the list
    else:
        while curr:
            if curr.elem == val:
                curr.prev.next = curr.next
                curr.next.prev = curr.prev
                self.length -= 1
                return True
            curr = curr.next
```

We check the head and tail as special cases of searching as the item is more likely to be at either one of those points in the list. A lot of applications of doubly linked lists will have items that are more likely to be deleted at either the head or tail. Take a Least Frequently Used Cache for example. This can be used to cache data that is typically stored in a database for quick retrieval. These are often implemented with a doubly linked list. It has a fixed size and if we want to add new items to it we want to remove the least frequently used item. We also arrange the items in the doubly linked list in order of how frequently we retrieve them, therefore if we want to remove, we will want to remove the item at the tail.

Doubly Linked List Applications

Doubly Linked Lists are used in browsers to implement the ability to go back or go forward to previously visited sites. In this section we'll use a doubly linked list to simulate how a browser does this.

To begin we'll create a new class called BrowserNavigation. This won't have all the functionality that standard web browsers have but it demonstrates how it may be implemented in real browsers. We'll be inheriting from the DoublyLinkedList class. Below is the outline of the new BrowserNavigation class. I also assume that the Node and DoublyLinkedList classes exist.

```
class BrowserNavigation(DoublyLinkedList):
    def __init__(self):
        super(Bro wserNavigation, self).__init__()
        self.position = self.tail
```

We have this *position* variable to keep track of where we are in the list without losing data in the list as we move through it.

To implement the forward and backward functionality we need to think about what it means to go backwards or forwards. When we open our browser and start visiting websites, we are inserting into the doubly linked list. When we go backwards, we move backwards through the list by 1 each time we move back. If we don't visit any new sites after going backwards, going forward just moves back up through the list towards the tail. However, if we move back then visit a new site, then everything from our current position in the list to the tail is lost and the tail becomes the new site we visit.

Therefore, it is easiest to code the go back functionality first. Below is the code for that.

```
class BrowserNavigation(DoublyLinkedList):
    def __init__(self):
        super(BrowserNavigation, self).__init__()
      self.position = sel f.tail

    def backward(self):
        if self.position.prev:
            self.position = self.position.prev
            print("Visiting {} ".format(self.position.elem))
        else:
            print("Nothing to go back to!")
```

Next, we need to implement the forward functionality. We do this in a similar way but going in the opposite direction:

```
clas s BrowserNavigation(DoublyLinkedList):
    def __init__(self):
        super(BrowserNavigation, self).__init__()
      self.position = self.tail

    def forward(self):
        if self.position.next:
            self.position = self.position.next
            print("Visiting {} ".format(self.position.elem))
        else:
            print("Nothing to go forward to!")
```

Now we need to implement the functionality of visiting a new website. If the list is empty, then this is straight forward, we simply add to the list the same way we do with a doubly linked list and just move the *`position`* forward. However, if we visit a new site and the *`next`* of *`position`* is not *None*, then we need to void everything after the current *`position`* and add the new site we're visiting.

```
class BrowserNavigation(DoublyLinkedList):
    def __init__(self):
        super(BrowserNavigation, self).__init__()
      self.position = self.tail

    def visit(sel f, site):
        if self.position is None:
            self.insert(site)
            self.position = self.tail
            print("Visiting {}".format(self.position.elem))
        elif self.position.next is None:
            self.insert(site)
            self.position = self.tail
            print("Visiting {}".format(self.posit ion.elem))
        else:
            # Void everything after the current position
            self.position.next = None
            # Set tail as where we are
            self.tail = self.position
            # Add new site
            self.insert(site)
            # Move the position forward to new site
            self.position = self.tail
            print("Visiting {}".format(self.position.elem))
```

Now if we can use our class as follows:

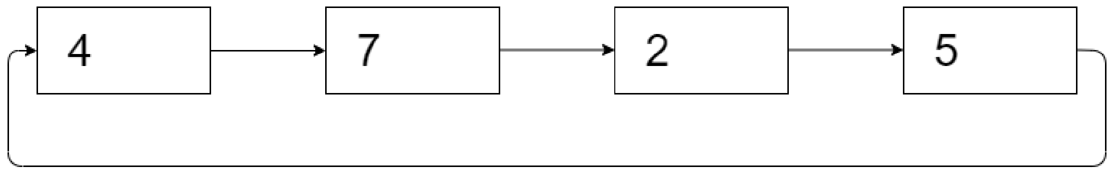
```
bn = BrowserNavigation()
bn.forward()    # Nothing
bn.backward()   # Nothing
bn.visit("www.youtube.com") # Go to Youtube
bn.visit("www.facebook.com") # Go to Facebook
bn.visit("www.instagram.com") # Go to Instagram
bn.backward()   # Back to Facebook
bn.back ward()  # Back to Youtube
bn.forward()    # Forward to Facebook
bn.backward()   # Backward to Youtube
bn.visit("www.slitherintopython.com") # Visit SlitherIntoPython, shoul
d remove facebook and instagram from list
bn.backward()   # Back to Youtube
bn.forward()    # Forward to SlitherIntoPython
bn.forward()    # Nothing
```

We get the following output:

```
Nothing to go forward to!
Nothing to go back to!
Visiting www.youtube.com
Visiting www.facebook.com
Visiting www.instagram.com
Visiting www.facebook.com
Visiting www.youtube.co     m
Visiting www.facebook.com
Visiting www.youtube.com
Visitin    g www.slitherintopython.com
Visiting www.youtube.com
Visiting www.slitherintopython.com
Nothing to go forward to!
```

Circular Linked Lists

Circular linked lists are the same as singly linked lists except the *next* pointer of the last element points to the head of the list. This is illustrated below.



In code, they vary slightly in implementation as there are some cases we need to look out for. One of which is searching. We need to be careful when searching for an element in the list as we may end up in an infinite loop if the element doesn't exist.

The Node class for circular linked lists are the same as they are for singly linked lists, so I'll omit these here. But first, let's look at how we can traverse a circular linked list. We'll start by creating the CircularLinkedList class and defining the `traverse()` method.

```
class   CircularLinkedList:
    def   __init__   ( self  , head  =None, tail     =None):
        self  .head   =  head
        self  .tail     =  tail

    def  traverse(   self  ):
        # Define the first node
        curr    =  self  .head

        # If there are more nodes, keep going
        while curr.next:
            print(curr.elem)
            curr = curr.next

            # Once we get back to the head, stop
            if curr == self.head:
                break
```

The last *if* statement in the above code checks for when we go back to the start of the list and prevents us from entering an infinite loop. In fact, *break* statements are often controversial as if not used correctly, can lead to hard-to-find bugs and are viewed by some as a crutch for bad logic but this is a perfect example of where they should be used.

Next, we'll look at inserting nodes into the list. We'll be inserting new nodes into the end of the list and this is still O(1) as we have a tail reference. You can try implementing and insertion method that adds new items to the beginning of the list - it's very similar.

```
class CircularLinkedList:
    def __init__(self, head=None, tail=None):
        self.head = head
        self.tail = tail

    def insert(self, elem):
        new_node = Node(elem)
        if self.head == None:
            self.head = new_node
            self.head.next = new_node
            self.tail = new_node

        if self.tail != None:
            self.tail.next = new_node
            new_node.next = self.head
            self.tail = new_node
```

In the above *insert()* method, the first *if* statement handles the case of an empty list. The second *if* statement inserts the node as the tail node and points its *next* pointer to the *head* node in order to keep the list circular.

Next, we'll define a deletion method. This time, we want to search the list for the element to delete, then remove it. Remember we want to stop when we reach the tail so that we avoid an infinite loop during the search part. We return *True* if it was deleted, otherwise we return *False*.

```
class CircularLinkedList:
    def __init__(self, head=None, tail=None):
        self.head = head
        self.tail = tail
    def  delete( self , val):
        # Empty list
        if  self .head  is  None:
            return  False

        # It's the first element, just update the head and tail pointe
r
        if  self .head.elem  == val:
            self .head = self .head. next
            self .tail.  next  = self .head

        # If there are more nodes, keep going
        curr  = self .head
        while  curr. next != self .head  and curr. next .elem  != val:
            curr  = curr. next

        if  curr. next == self .head:
            return  False
        else :
            curr. next  = curr. next . next
            return  val
```

The only differences between this deletion method and the deletion method for singly linked lists, is that we also need to update the tail pointer when the element being deleted is the first element in the list. The second is the condition in the *while* loop. Instead of continuing until *curr.next is None*, we continue while *curr.next != self.head*.

We can also have Circular Doubly Linked Lists. I won't be giving the code for them here but I'm sure you're well equip at this point to code them yourself, so I'd encourage you to try that yourself.

Circular Linked List Applications

I won't be going through code examples here, but I've listed some useful applications of circular linked lists below:

- Games: In turn-based games, they may be used to handle the case of when we've reached the last player and we go back to the first player. We could just reset our *curr* pointer in a singular linked list to jump back to the start but if we use a circular linked list we can just continuously go to the *next* player until some end game condition is met.

- Operating system process scheduling: Your computer probably only has a single CPU but has multiple applications running. For each of these programs to run as you'd expect them too, your computer needs to give them time on the CPU. A circular linked list can be used to hold all running applications in it and the operating system gives a fixed time slot to each application. The operating system keeps iterating over the list in a circular fashion until all the applications have finished running. There are better process scheduling algorithms today, but this data structure can be used to schedule them.

- Going back to our media player example from earlier, we could use circular linked lists to implement the 'Repeat' functionality. That is, when a playlist finishes, go back to the start. I know we used a Queue for creating our media player simulator, but we can turn our circular linked list into a queue by inserting at the head and removing from the tail. This way we have a data structure called a Circular Queue.

Trees

Binary Search Trees

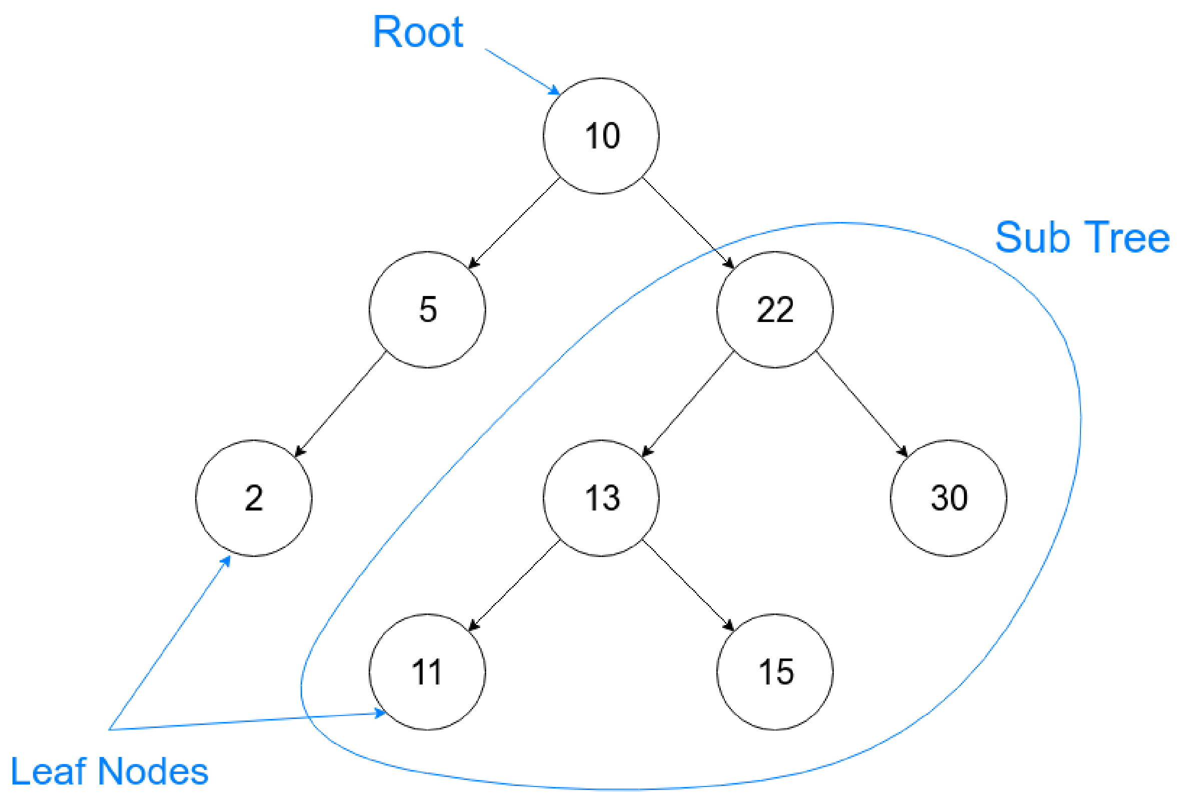
Before I start this section, I want to give you a warning. We will be using recursion quite a lot here. If recursion is not your strong point (which it probably won't be at this point), perhaps brush up on the recursion section. However, this section might be a good place to help you understand recursion. BSTs are what helped me wrap my head around recursion (well, at least that's when recursion clicked for me).

A binary search tree (BST) is somewhat different to the data structures we have met so far. It is not a linear data structure. It is a type of ordered data structure that stores items. BSTs allow for fast searches, addition and removal of items. BSTs keep their items in sorted order so that the operations that can be performed on them are fast, unlike a linked list who's add and remove operations are O(n). The add and remove operations on a BST are O(log n). Searching a BST is also O(log n). This is the average case for those three operations.

This is a similar situation to linear search vs. binary search which we looked at earlier.

In computer science, a *tree* is a widely used abstract datatype (a data type defined by its behavior not implementation) that simulates a hierarchical tree structure (much like a family tree), with a root node and subtrees of children with a parent node represented as a set of linked nodes.

A binary tree is a type of tree in which each Node in the tree has at most, two child nodes. A binary search tree is a special type of binary tree in which the elements are ordered. A diagram might help you visualize this, so here is one:



There is a lot to take in here so let me explain. The root of this tree is the node that contains the element 10. The big thing labeled *Sub Tree* is the right sub tree of the root node.

10 is the parent node to 22 and 5 and similarly, 5 is a child node of 10.

A leaf node is a node that has no children.

As you can see from this diagram, it is a binary tree as each node has at most two child nodes. It also satisfies the an important property that makes it a binary search tree. That is, that if we take any node, 10 for example, everything to the right is greater than it and everything to the left is less than it. Similarly if we look at the node 13, everything to the right is greater than 13 and everything to the left is less than 13.

What makes searching fast is this, if we wanted to check if 2 was in the tree, we start at the root node and check if 2 is less than the value at the root. In this case 2 is less than 10 so we know we don't need to bother searching the root node's right sub tree. Essentially what we've done here is cut out everything in that blue circle as the element 2 will not be in there.

Let's look at implementing a binary search tree. Again, the concept of a `Node` is important here. Our `Node` class for a binary search tree will be slightly different than what it was for a linked list. Let's take a look:

```
class   Node:
        def   __init__(self, elem):
              self.elem = elem
              self.left = None
              self.right = None
```

As you can see here, each node will have an element, a left child and a right child (either or both of which may be `None`).

Let's make our first attempt at a `BinarySearchTree` class:

```
class   BinarySearchTree:
        def   __init__(self):
              self.root =       None
```

Pretty simple so far. Basically, what we can do if we want to add, remove or search for an element is follow pointers from left sub trees or right sub trees until we find what we are looking for.

This time we can look at the search operation first. It will be a good look at how to use recursion in a practical situation. We either want to return `None` if the element is not found or return the `None` that contains what we're looking for if the element is found.

```
classBinarySearchTree:
  def __init__(self):
    self.root = None

  def search(self, value):
    return self.recursive_search(self.root, value)

  def recursive_search(self, node, value):
    if node is None or node.elem == value:
      return node
    if value < node.elem:
      return self.recursive_search(node.left, value)
    else:
      return self.recursive_search(node.right, value)
```

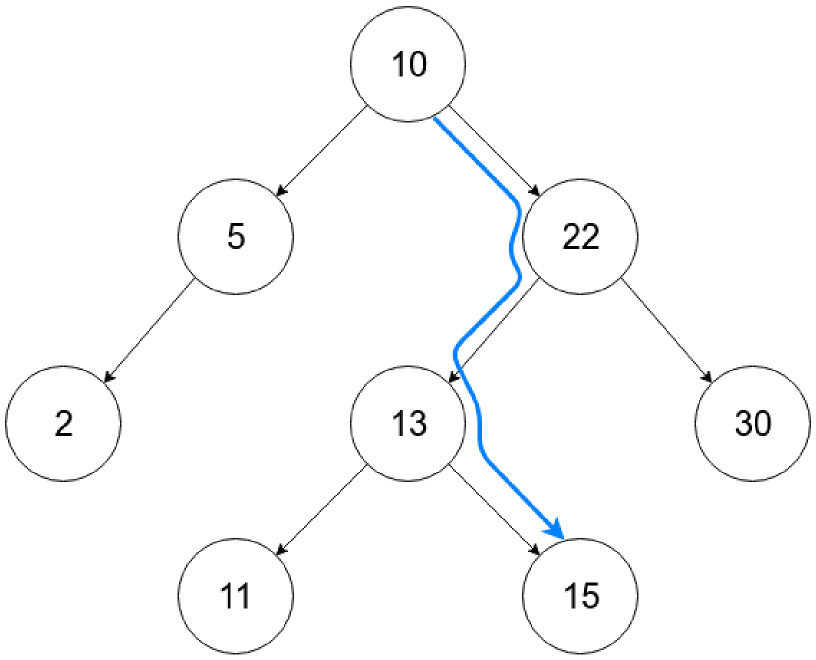
This may seem a little odd as we have two search functions but there is good reason for it. The `search` method allows us to call the `recursive_search` method and pass in the root node, that way, we now have a *copy*of the root node and not the root node itself, so we don't end up in an infinite loop when recursively searching. Again, the recursive nature of this is very difficult to explain so I'd try working through an example on paper to help clarify after the following explanation.

Inside the `recursive_search` method we have a base casewhich checks if we've reached `None` (we didn't find what we're looking for) or we've reached the `Node` that contains what we're searching for.

If we don't pass our base case, then we check if the value we're searching for is less than the value at the node we're currently at; if it is then we search the left subtree of that node by calling the `recursive_search` method and passing the current nodes `left` tree. Otherwise, the value we're searching for is greater than the value at the current node, in which case we call `recursive_search` and pass in the current nodes `right` tree.

Remember a node's left or right tree is simply a pointer to a node (we can look at this node as the root node of a sub tree).

For example, if we're searching for the value `15` in the tree from the diagram above, then the path we take looks like this:



Just to clarify, a leaf nodes `left` and `right` 'trees' are `None`.

Inserting into the BST

Next, we will look at inserting an element into the tree. What we're doing here is following pretty much the same thing we did with search except when we reach a leaf node on our path we will set it's left or right pointer to point to a new node (depending on whether or not the value is less than or greater than the value at the child node).

Here is how we will approach this:

- Base case: If the node we are at is `None` then insert the new node here.
- Recursive case:
- If the value we're inserting is less than the value of the current node then we check if left tree of the current node is `None`; If it is, then insert there, otherwise, call insert again and pass the left subtree.
- If the value we're inserting is greater than the value of the current node then we check if the right tree of the current node is `None`; If it is, then insert there, otherwise, call insert again and pass the right subtree.

Here's how that's done:

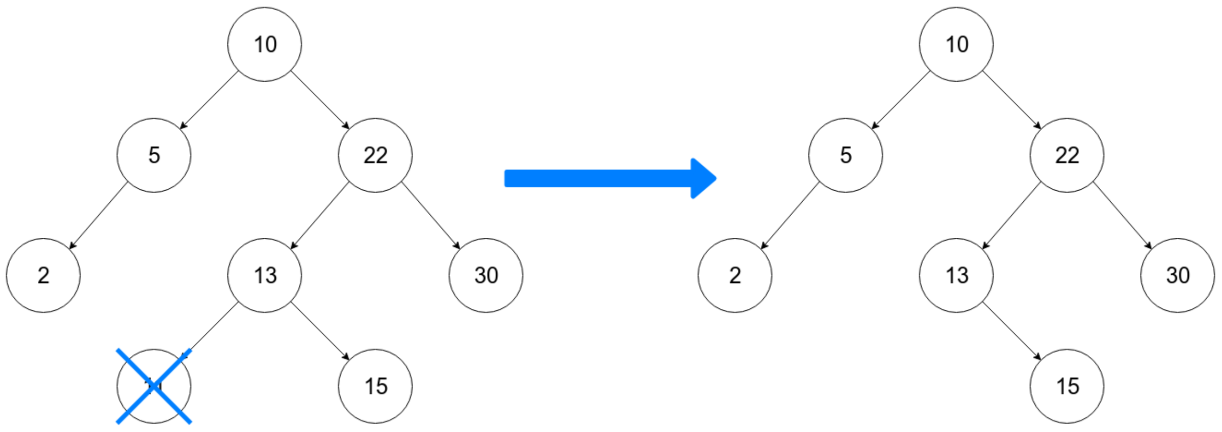
```
class   BinarySearchTree:
    def   __init__(self):
        self.root = None

    def   insert(self, val):
        if   self.root is      None:
            self.root = Node(val)
        else :
            self.recu   rsive_insert(self.root, val)

    def   recursive_insert(self, node, val):
        if   val < node.elem:
            if   node.left    is   None:
                node.left = Node(val)
            else :
                self.recursive_insert(node.left, val)
        else :
            if   node.right    is   None:
                node.right = Node(val)
            else :
                self.recursive_insert(node.right, val)
```

That's it. I hope you can see why recursion is useful here. It makes our code read better and when we're planning this out in our heads, it naturally fits into our algorithm.

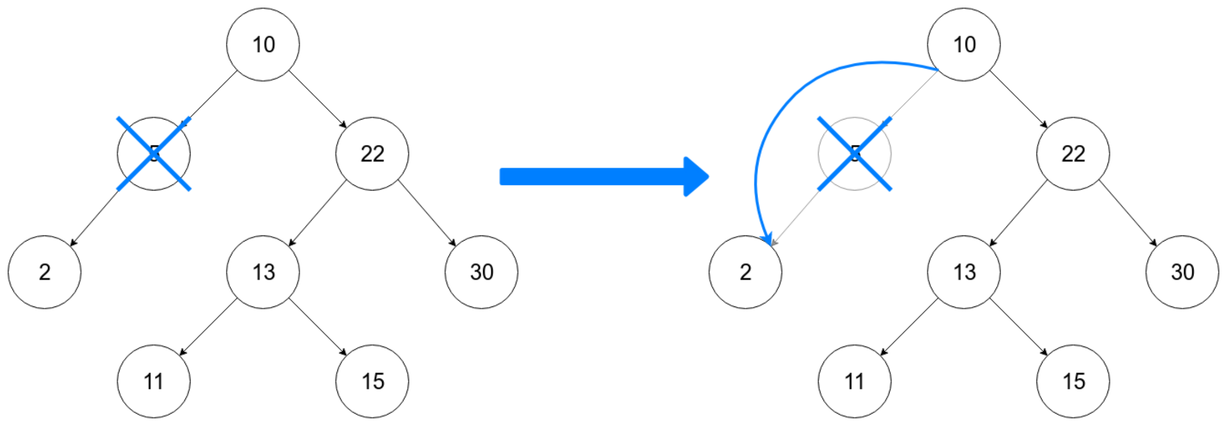
Deleting from the BST

Next, we're going to look deleting an element. This is quite a bit more difficult. With insertion, we always insert into one of the leaf nodes but if we're deleting something, then three possibilities arise. The first is the simplest case:

1. The node we're removing has no child nodes (i.e. a leaf node). In this case we can just remove the node.

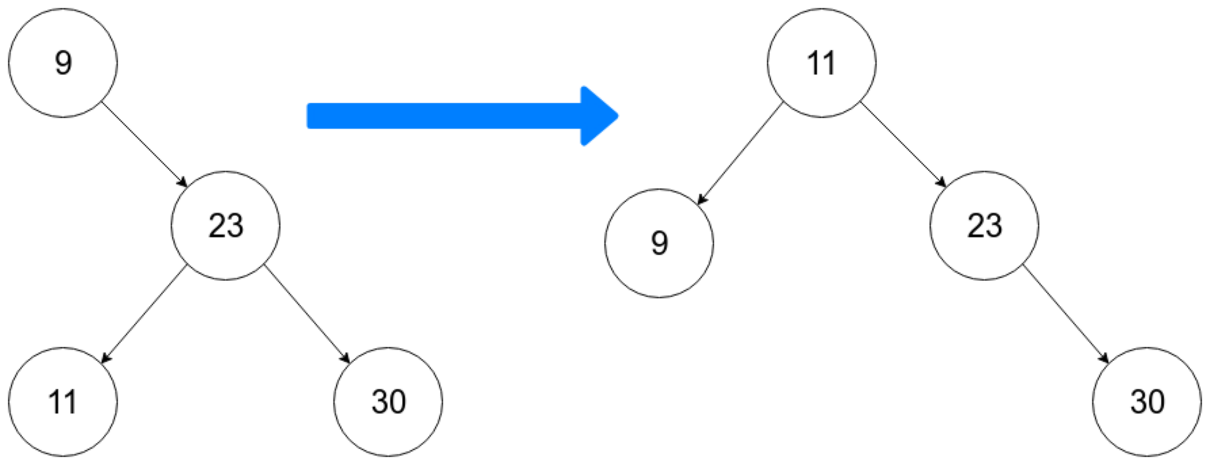


2. The next case is when we're removing a node that only has one child. Again, this is simple enough to deal with. In this case we just cut the node we're removing from the tree and link its child to its parent.



3. The final case is when we're removing a node that has two child nodes. This is the most complex case and requires some smart thinking. There is however, one useful property of BSTs that we can use to solve this problem. That is, the same set of values can be represented as different binary-search trees. Let's consider

the following set of values: *{9, 23, 11, 30}*. We can represent this in two different ways:



Both of these trees are different, yet they are both valid. What did we do here though to transform the first tree into the second?

¶ We started at the root node (9 in this case).
¶ Searched it's right subtree for the minimum element (11 in this case)
¶ Replaced 9 with 11.
¶ Hang 9 from the left subtree.

We can take this idea and apply it to removing elements with two child nodes. Here is how we'll do that:

¶ Find the node we want to delete.
¶ Find the minimum element in the right subtree.
¶ Replace the element of the node to be removed with the minimum we just found.
¶ Be careful! The right subtree now contains a duplicate.
¶ Recursively apply the removal to the right subtree

When applying the removal to the right subtree to remove the duplicate, we can be certain that the duplicate node will fall under case 1 or 2. It won't fall under case 3 (It wouldn't have been the minimum if we did).

There are two functions we'll need here. One is the remove method and the other will find the minimum node. We get a free function here (we will have a `min` operation on our BST!).

Here's how to code it, I'll also comment the code as it's a bit complex:

```
class BinarySearchTree:
    def __init__(self):
        self.root = None

    def remove(self, value):
        return self.recursive_remove(self.root, None, value)

    def recursive_remove(self, node, parent, value):
        # Helper function
        def min_node(node):
            curr = node
            while curr.left != None:
                curr = curr.left
            return curr

        # Base case (element doesn't exist in the tree)
        if node == None:
            return node

        # If element to remove is less than current node
        # then it is in the left subtree. We must set the current
        # node's left subtree equal to the result of the removal
        if value < node.elem:
            node.left = self.recursive_remove(node.left, node, value)

        # If element to remove is greater than current node
        # then it is in the right subtree. We must set the current
        # node's right subtree equal to the result of the removal
        elif value > node.elem:
            node. right = self.recursive_remove(node.right, node,
value)

        # If the element to remove is equal to the value of
        # current node, then this is the node to remove
        else :
            print (node.elem, not parent is None)
            # Not the root node
            if not parent is None:
                # Node has no children (CASE 2)
                if node.num_children() == 0:
                    if parent.left == node:
                        parent.left = None
                    els e:
                        parent.right = None
                # Node has only one child (CASE 2)
                elif    node.num_children() == 1:
                    # Get the child node
                    if   node.left !=       None:
                        chil  d = node.left
                    else :
                        child = node.right

                    # Point the parent node to the child node of
                    # the node we're removing
                    if   parent.left == node       :
                        parent.left = child
                    else :
                        parent.right = child

                # Node has two children (CASE 3)
                elif    node.num_children() == 2:
                    # Get t   he min node in the right subtree
                    #  of node to remove
                    smallest_node = min_node(node.right)

                    # Copy the smallest nodes value to the
                    #  node formerly holding
                    #  t he value we wanted to delete
                    node.elem = smallest_node.elem
                    self.recursive_remove(node, parent, value)
            # Node to delete is the root node
            else :
                # Only 1 element in the tree (t             he  root)     Ƶ
                #  set the root to None
                if   node.num_children() == 0:
                    node =   None
                # Root has one child         -   make that the root
                elif    node.num_children() ==       1:
                    if n   ode.left !=       None:
                        self.root = node.left
                    if node.right !=         None:
                        self.root = node.right
                elif    node.num_children() == 2:
                    smallest_node = min_node(node.right         )
```

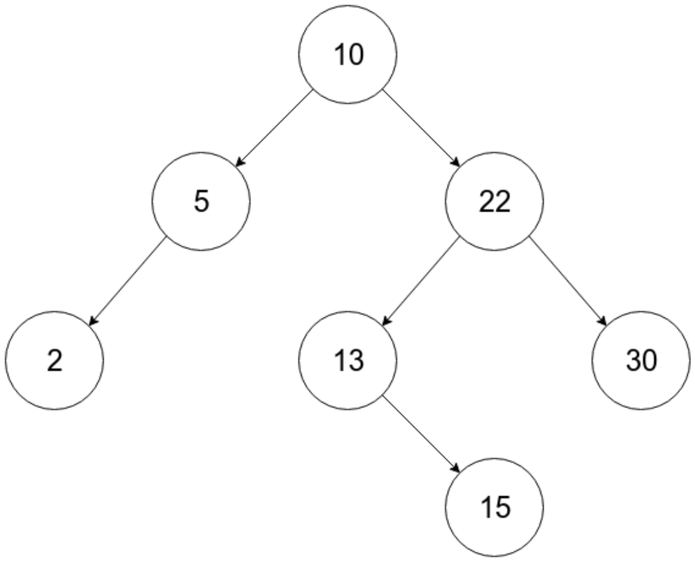
```
                        node.elem = smallest_node.elem
                        self.recursive_remove(node,    None, value)
        return  node
```

This is tough. Make sure you understand the delete operation, even if that means going through it over and over again. Use a pen and paper if you need! It is a good question for an employer to ask as it shows how someone approaches a tough problem (pointing out different scenarios, breaking the problem down, etc.).

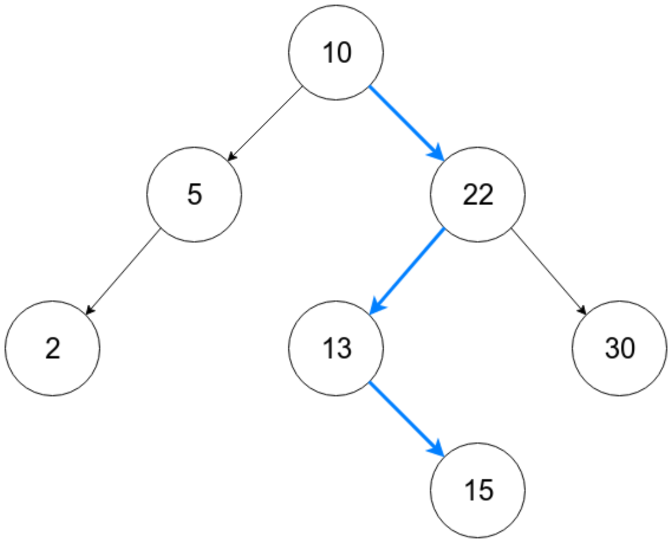
The Height of the BST

The *height* (depth) of a Tree is the number of edges on the longest path from the root node to leaf node. (An edge is basically the arrows in our diagrams that we've looked at before)

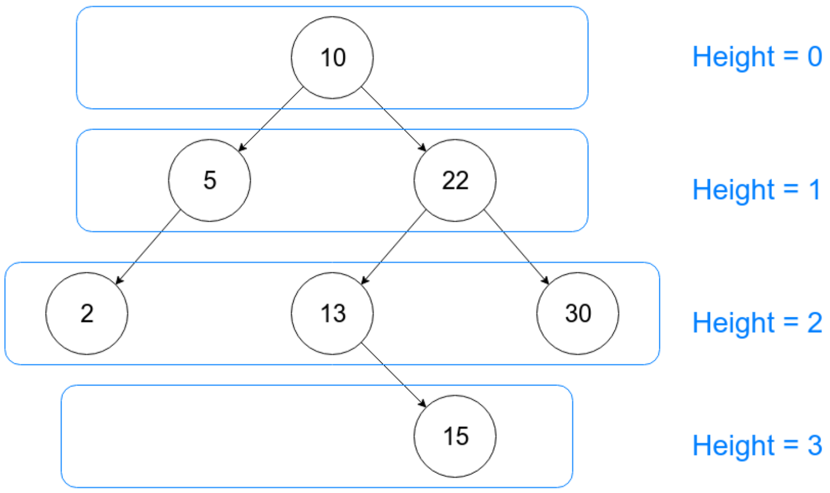
For example, the height of the following tree is 3:



The height is determined by the number of edges in the longest path, like the following:



You can also view the height of a tree as such:



Let's look at how we might implement the `height` method. I've omitted the rest of the code to do with the BST and just included the new methods needed to find the height:

```
def height(self):
    if self.root is None:
        return 0
    # We subtract 1 because the root node height is 0 not 1
    return self.recursive_height(self.root) - 1

def recursive_height(self, node):
    if node == None:
        return 0
```

```
            return    1 + max(self.recursive_height(node.left),
self.recursive_height(node.right))
```

AVL Trees

Consider what happens when we add elements to the tree in the following order: {2, 5, 7, 12, 13}. We end up with a Linked List. That means that our search and insert is no longer O(log n). It's now O(n). This isn't really that great for us.

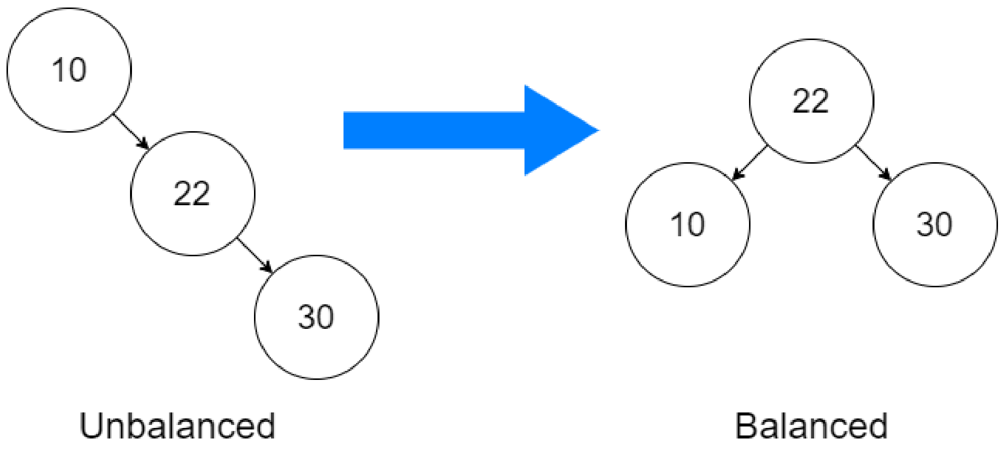
We can however, make an optimization to our binary search tree so that our search and insert are always O(log n), regardless of what order we enter the elements.

The optimization is that the difference between height of the left and right subtrees is no greater than one for all nodes. When the difference between left and right subtrees height is no greater than `1`, we say the tree is balanced. This means, if we insert or remove an element and the height between left and right subtrees becomes greater than 1 then we must rearrange the nodes. We will call this the height condition in this book, but it is also called the balance condition.

This optimization means that our binary search tree will become self-balancing.

This type of self-balancing binary search tree has a special name. It's called an AVL Tree.

This balancing operation is difficult to show in book form as a moving visual of it is the best way to understand it. If you Google 'AVL tree visualization' you should find many examples that demonstrate how it works. However, I've illustrated what we're trying to achieve below.



Consider the BST to the left. This is essentially a linked list and the height between the left and right subtrees of the root node are > 1. We should therefore balance the tree. When we perform this operation we end up with the tree on the right.

We're going to implement the tree slightly differently to how we implemented the Binary Search Tree. To begin, we can start with our Node class which is the same as the Node class used in the Binary Search Tree:

```
class Node():
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
```

Next, let's make a first attempt at creating the *AVLTree* class:

```
class AVLTree():
    def __init__(self):
        self.node = None
        self.height = -1
        self.balance = 0
```

The way this implementation works is, each AVL Tree contains a node, has a balance and has a height. The height we have seen before, but the balance attribute is new. I'll explain the purpose of this soon.

Inserting into the AVL Tree

Next, we want to insert items into the tree. Insertion works much the same way as it does in Binary Search Trees. There are three cases we want to cover:

1. The tree has nothing in it, so the item we want to insert becomes the root.
2. The item we're inserting is less than the root of the tree, so we insert into the left subtree.
3. The item we're inserting is greater than the root of the tree, so we insert into the right subtree

The main difference is, when we've inserted the new element, we must balance the tree to ensure the height of either one of the subtrees is no greater than 1 when compared to the other subtree.

The insertion method is as follows:

```
class   AVLTree():
      def   __init__(self):
            self.node =      None
            self.height =       - 1
            self.balance = 0

      def   insert(self, value):
            node = Node(value)
            # Tr  ee is empty, initialize root
            if    self.node ==      None:
                  self.node = node
                  self.node.left = A        VLTree()
                  self.node.right = AVLTree()
            # Insert into left subtree
            elif     value < self.node.value:
                  self   .node.left.insert(value)
            # Insert into right subtree
            elif     value > self.node.value:
                  self.node.right.insert(value)
            # Rebalance if needed
            self.rebalance()
```

We can see at the bottom of the above code snippet that we call a `rebalance()` method. This will handle rearranging our tree so that the height condition is met.

Rebalancing the AVL Tree

Let's take a look at how the rebalancing process works. Before we can rebalance there are two things we need to do. We need to update the height value. This will be done in an `update_heights()` method. This is recursively as follows:

```
      def   update_heights(self):
            # Height is max height of left or right
            #  subtrees + 1 for the root
            if    self.node:
                  if    self.node.left:
                          self.node.left.update_heights()
                  if    self.node.right:
                          self.node.right.update_heights()
                  self.height = 1 + max(self.node.left.height,
self.node.right.height)
            else :
                  self   .height =     - 1
```

If the tree has no elements then the height is the initialized value of -1. Otherwise the height of the tree is `1 + max(left_tree, right_tree)`. We add 1 here to account for the root node. As you can see, before we update the height for the root, we need to update the heights of the left and right subtrees. This is happening recursively.

This may be difficult to understand so I would encourage you to draw out a small Binary Search Tree and step through this method on paper on the tree you have drawn to get a better understanding of how this is working.

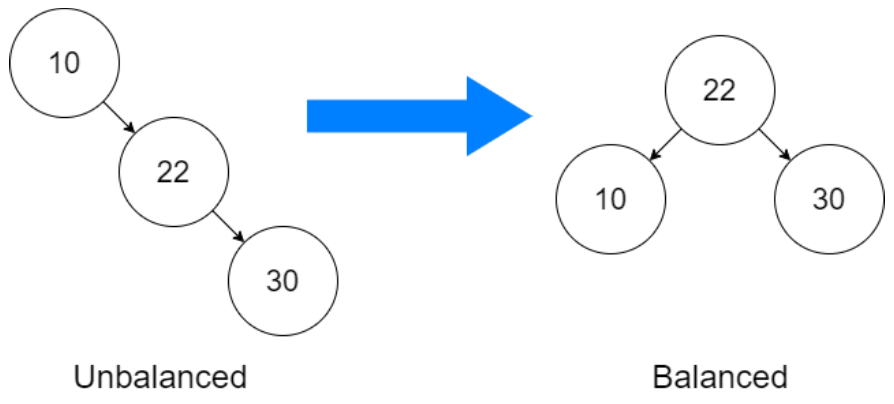
The next thing we need to do is update the balance values for the root and all subtrees. For any tree, to work out the balance factor of the tree, you follow the simple formula below:

If we have call `update_heights()` prior to calculating the balance factor, then the code for updating the balances is very straight forward. We will do this in a recursive method called `update_balances()`. The code for which is below:

```
def  update_balances(self):
        # Calculate the balance factor of the tree
        # Balance factor calculated as follows:
        #     BF = height(left_subtree)            -   height(right_subtree)
        if   self.node:
            if   self.node.left:
                  self.node.left.update_balance        s()
            if   self.node.right:
                  self.node.right.update_balances()
            self.balance = self.node.left.height               -
self.node.right.height
        else :
            self.balance = 0
```

This again may be slightly difficult to understand due to the recursive nature of the method, but I encourage you to work it out on paper so that it becomes clearer.

There are also two other things we need to do before we can write the rebalance method. We need to write two methods. One for handling a *right rotation* and the other for handling a *left rotation*. Below is an illustration of a left rotation.



The code for rotating right and lift is shown below:

```
def   rotate_right(self):
      # Set self as the subtree of the left subtree
      new_root   = self.node.left.node
      new_left_sub = new_root.right.node
      old_root = self.node

      self.node = new_root
      old_root.left.node = new_left_sub
      new_root.right.node = old_root

def   rotate_left(self):
      # Set self as        the left subtree of the right subtree
      new_root = self.node.right.node
      new_right_sub =new_root.left.node
      old_root = self.node

      self.node = new_root
      old_root.right.node = new_right_sub
      new_root.left.node = old_        root
```

Although these methods may look straight forward, you should be aware that we're manipulating the objects the variables point to and not the individual values stored in the nodes. Again, I encourage you to draw this out on paper and work through the example from the above diagram until you arrive at the solution.

Now that we have all the methods needed to rebalance the tree, we can write the `rebalance()` method. We need to know when rotations are necessary, and that information is given to us by the balance factor. If the balance is greater than 1 or less than -1, then we know we must perform a rotation as the height condition is not met.

```
    def rebalance(self):
        self.update_heights()
        self.update_balances()
        # If balance is < - 1 o r > 1 then
        #  rotations are still necessary to perform
        while self.balance < - 1 or self.balance > 1:
            # Left subtree is larger than right
            #  subtree so rotate to the left
            if self. balance > 1:
                if self.node.left.balance < 0:
                    self.node.left.rotate_left()
                    self.update_heights()
                    self.update_balances()
                self.rotate_right()
                self.updat e_heights()
                self.update_balances()

            # Right subtree larger than left subtree
            #  so rotate to the right
            if self.balance < - 1:
                if self.node.right.balance > 0:
                    self.node.righ t.rotate_righ t()
                    self.update_heights()
                    self.update_balances()
                self.rotate_left()
                self.update_heights()
                self.update_balances()
```

Now we can put everything together so that we can insert into the tree and rebalance. The complete code is below:

```
class AVLTree():
    def __init__(self):
        self.node = None
        self.height = - 1
        self.balance = 0

    def insert(self, value):
        node = Node(value)
        # Tree is empt y, initialize root
        if self.node == None:
            self.node = node
            self.node.left = AVLTree()
            self.node.right = AVLTree()
        # Insert into left subtree
        elif value < self.node.value:
            self.node.left      .insert(value)
        # Insert into right subtree
        elif value > self.node.value:
            self.node.right.insert(value)
        # Rebalance if needed
        self.rebalance()

    def rebalanc e(self):
        self.update_heights()
        self.update_balances()
        # If balance is <     - 1 or > 1 then rotations
        #  are still necessary to perform
        while self.balance <     - 1 or self.balance > 1:
            # Left subtree is larger than right
            #  subtree so rotate to the l        eft
            if self.balance > 1:
                if self.node.left.balance < 0:
                    self.node.left.rotate_left()
                    self.update_heights()
                    self.update_balances()
                self.rotate_right()
                self.update_heights()
                self.update_balances()

            # Right subtree larger than left
            #  subtree so rotate to the right
            if self.balance <     - 1:
                if self.node.right.balance        > 0:
                    self.node.right.rotate_right()
                    self.update_heights()
                    self.update_balances()
                self.rotate_left()
                self.update_heights()
                self.update_balances()

    def update_heights(self):
        # Height is max height of left or right
        #  subtrees + 1 for the root
        if self.node:
            if self.node.left:
                self.node.left.upda        te_heights()
            if    self.node.right:
                self.node.right.update_heights()
            self.height = 1 + max(self.node.left.height,
self.node.right.height)
        else :
            self.height =        - 1

    def   update_balances(self):
        # Calculate the balance factor of the tree
        #  Balance factor calculated as follows:
        #     BF = height(left_subtree)            -   height(right_subtree)
        if    self.node:
            if    self.node.left:
                self.node.left.update_balances()
            if    self.node.right:
                self   .node.right.update_balances()
            self.balance = self.node.left.height                  -
self.node.right.height
        else :
            self.balance = 0

    def   rotate_r   ight(self):
        # Set self as the subtree of the left subtree
        new_root = self.no      de.left.node
        new_left_sub = new_root.right.node
        old_root = self.node

        self.node = new_root
        old_root.left.node = new_left_sub
        new_root.right.node = old_root

    def   rotate_left(self):
        # Set self as the left s            ubtree of the right subtree
        new_root = self.node.right.node
        new_right_sub =new_root.left.node
        old_root = self.node

        self.node = new_root
        old_root.right.node = new_right_sub
        new_root.left.node = old_root
```

Deleting from the AVL Tree

Deletion from an AVL Tree is similar to that of a Binary Search Tree except we need to rebalance afterwards. The commented code for removal of a node from an AVL tree is below:

```
def remove(self, value):
    if self.node != None:
        # Found the node delete it
        if self.node.value == value:
            # Node is a leaf node - Just remove
            if not self.node.left.node and not
self.node.right.node:
                self.node = None
            # Only one subtree - the right
            # subtree - replace root with that
            elif not self.node.left.node:
                self.node = self.node.right.node
            # Only one subtree - the left
            # subtree - replace root with that
            elif not self.node.right.node:
                self.node = self.node.left.node
            else :
                # Find succe ssor as smallest
                # node in the right subtree or
                # predecessor as largest node in left subtree
                successor = self.node.right.node
                while successor and successor.left.node:
                    succes sor = successor.left.node

                if succ essor:
                    self.node.value = successor.value
                    # Delete successor from the
                    # replaced node right subtree
                    self.node.right.remove(successor.value)
        # Remove from left subtree
        elif value < self.node.value:
            self.node.left. remove(value)
        # Remove from right subtree
        elif value > self.node.value:
            self.node.right.remove(value)
        # Rebalanc e if needed
        self.rebalance()
```

The complete code for an AVL tree, including rebalancing, insertion, and deletion can be found below:

```
class   Node():
    def __init__(self, value):
        self.value = value
        self.left =        None
        self.r    ight =     None

class   AVLTree():
    def   __init__(self):
        self.node =      None
        self.height =       - 1
        self.balance = 0

    def   insert(self, value):
        node = Node(value)
        # Tree is empty, initialize root
        if    self.node == None:
            self.node = node
            self.node.le     ft = AVLTree()
            self.node.right = AVLTree()
        # Insert into left subtree
        elif     value < self.node.value:
            self.node.left.insert(value)
        # Insert into right subtr             ee
        elif     value > self.node.value:
            self.node.right.insert(value)
        # Rebalance if needed
        self.rebalance()

    def remove(self, value):
        if    self.node !=      None:
            # Found the node delete it
            if    self   .no de.value == value:
                # Node is a leaf node         -   Just remove
                if    not   self.node.left.node         and  not
self.node.right.node:
                    self.node =      None
                # Only one subtree       -   the right
                #  subtre  e  -   replace root with         that
                elif     not   self.node.left.node:
                    self.node = self.node.right.node
                # Only one subtree - the left
                # subtree - replace root with that
                elif not self.node.right.node:
                    self.node = self.node.left.node
                else:
                    # Find successor as smallest
                    # node in the right subtree or
                    # predecessor as largest node in left subtree
                    successor = self.node.right.node
                    while successor and successor.left.node:
                        successor = successor.left.node

                    if successor:
                        self.node.value = successor.value
                        # Delete successor from the
                        # replaced node right subtree
                        self.node.right.remove(successor.value)
            # Remove from left subtree
            elif value < self.node.value:
                self.node.left.remove(value)
            # Remove from right subtree
            elif value > self.node.value:
                self.node.right.remove(value)
            # Rebalance if needed
            self.rebalance()

    def rebalance(self):
        self.update_heights()
        self.update_balances()
        # If balance is < -1 or > 1 then
        # rotations are still necessary to perform
        while self.balance < -1 or self.balance > 1:
            # Left subtree is larger than right
            # subtree so rotate to the left
            if self.balance > 1:
                if self.node.left.balance < 0:
                    self.node.left.rotate_left()
                    self.update_heights()
                    self.update_balances()
                self.rotate_right()
                self.update_heights()
                self.update_balances()

            # Right subtree larger than left
            #  subtree so rotate to the right
            if    self.balance <       - 1:
                if    self.node.right.balance > 0:
                    self.node.right.rotate_right()
                    self.update_heights()
                    self.update_balances()
                self.rotate_left()
                self.update_height      s()
                self.update_balances()

    def   update_heights(self):
        # Height is max height of left
        #  or right subtrees + 1 for the root
        if    self.node:
            if    self.node.left:
                self.node.left.update_height           s()
            if    self.node.right:
                self.node.right.update_heights()
            self.height = 1 + max(self.node.left.height,
self.node.right.he       ight)
        else :
            self.height =       - 1

    def   update_balances(self):
        # Calcu  late the balance factor of the tree
        # Balance factor calculated as follows:
        #     BF = height(left_subtree)               -   height(right_subtree)
        if    self.node:
            if    self.node.left:
                self.node.left.update_balances()
            if    self.node.right:
                self.node.right.update_balances()
            self.balance = self.node.left.height                   -
self.node.right.height
        else :
            self.balance = 0

    def   rotate_right(self):
        # Set self as the subtree           of the left subtree
        new_root = self.node.left.node
        new_left_  sub = new_root.right.node
        old_root = self.node

        self.node = new_root
        old_root.left.node = new_left_sub
        new_root.right.node = old_root

    def rotat  e_left(self):
        # Set self as the left subtree of the right subtree
        new_root = self.node.right.node
        new_right_sub =new_root.left.node
        old_root = self.node

        self.node = new_root
        old_root.right.node = new_right_s           ub
        new_root.left.node = old_root
```

Hash Tables

Hash tables are probably my favorite data structure. They seem like magic and there is so much we can do with them both in terms of applications but also in terms of their underlying structure in order to fit them better to certain applications, improve them etc.

We've been looking at various forms of linked lists, and trees, all of which we must search through, or iterate over to find an element. With hash tables we do not have to do that. We get instant access to any element (this isn't strictly true all the time but we'll look at that later in the chapter) but hash tables have an average or amortized time complexity of O(1) for access, deletion, and insertion.

With Python lists, we must access elements using an index which is an integer which is easy for a computer to understand. This gives us instant access but what do we do if we're building an address book application? The index *3* doesn't mean much to us. Instead, we'd want to search by somebody's name which would be a string.

As a side note, you may hear the terms Hash Table and Hash Map. These are essentially the same thing, but different languages make certain distinctions between them such as allowing null keys or whether they're synchronized and thread-safe or not. That may not make much sense to you now but don't worry about that for now.

Hash Functions

A hash function is a function that takes a value of one type (called the *key*) and converts it into another value - usually an integer. There are a wide range of hash functions, some of which are very simple and bad and others that are considered "cryptographically secure". A cryptographically secure hashing function is a one-way function, that is, one which is practically impossible to invert. In other words, it's easy to produce the output value but almost impossible to take the output and find what the input was. We won't be looking at cryptographically secure functions in this book.

A hash function doesn't need to be cryptographically secure to be considered good. A good hash function should satisfy the two following properties as best as possible:

- Efficiently computable
- Should uniformly distribute the keys.

Notice above how I said, "as best as possible". That is because the perfect hash function doesn't exist (or at least we haven't found it yet). The perfect hash function would mean that we could map every single string imaginable to a unique hash, every single file imaginable to a unique hash, every single object to a unique hash, and every other "thing" of some type to a unique hash. All hash functions that we know of will end up having a *collision* That is, in the case of strings, we will have two different strings that will have the same hash.

A hash table takes keys and maps them to places in a list. The hash function determines where in the table where items should go. As a first attempt, let use a simple hash function and our hash table will store integers.

Our simple hash function works as follows:

hash = key % table_size

This allows us to find out the index a given key is at. For example, if we have a table of size 10, that is, it can store 10 items and we want to insert the number 345893 into the table. To do this we calculate the hash as:

hash = 345893 % 10

That means, for the number 345893, it's hash is 3. Therefore, we enter the number 345893 at index 3 in the table and when we want to see if the number 345893 is in the hash table, we pass it through the hash function which will be 3, so we check the table at index 3. That is essentially what hashing and hash tables are all about. However, we can make them a little more sophisticated.

What if we wanted to use strings as the keys so that we can do something *contact = hashtable["john"]* in order to fetch contact details for someone called John in the context of an address book? This may seem familiar now - the syntax looks just like Python Dictionary types. That's because Python dictionaries are hash maps under the hood.

In the case of strings, we'll need another hash function because we can't do something such as *hash = "john" % 10* as these are two different types. So, we'll need to convert the string to an integer.

One way we can do this is by using the Python built in *ord()* function. This function converts a single character to it's Unicode code which is an integer. Below we can see how we can convert a string to it's Unicode codes.

h	e	l	l	o		w	o	r	l	d
104	101	108	108	111	32	119	111	114	108	100

We can sum up the values of these Unicode codes to give us the hash. In the above case, this would be

104 + 101 + 108 + 108 + 111 + 32 + 119 + 111 + 114 + 108 + 100 = 1116

So, *1116* would be our hash. However, there is a problem with this approach. Consider what happens when we swap the *h* for the *w* to give the string *wello horld*.

w	e	l	l	o		h	o	r	l	d
119	101	108	108	111	32	104	111	114	108	100

If we sum these values up, we end up with the same hash of *1116*. We call this a *collision* We want a hash function that has the smallest number of collisions possible that is also efficient to compute. There has been a huge amount of work in developing hash functions and they can get quite complex, but we'll extend our simple hash function from above to try and reduce the probability of collisions occurring.

A better hash function would take the position of characters into place. There are many ways to do this such as polynomial hash codes, in which case we multiply by some non-constant number each time. There are variants of polynomial hash codes such as cyclic-shift hash codes that instead of multiplying, will carry out bit shifting operations to scramble up the bits that represent an integer. That may be a little bit confusing here, so we'll take the polynomial hash code approach.

What we want to do is take the *ord()* of each character and multiply it by some number. That number that we're multiplying should change which each new character we arrive at. Below is the function that does that:

```
def hash_code(s):
    mult = 1
    hash_val = 0
    for ch in s:
        hash_val += mult * ord(ch)
        mult += 1
    return hash_val
```

Now if we pass the strings *hello world* and *wello horld*, we should not have a collision.

```
>>> hash_code("hello world")
6736
>>> hash_code("wello horld")
6646
```

This hash function clearly functions better than our first few approaches but it's still not great. We'll still get collisions and as I've said, all hash functions produce collisions - it's simply something we can't avoid (at least we haven't discovered a way) so we need to come up with a way of dealing with them called collision handling schemes which we'll get to later but for now let's use our polynomial hash function when constructing our hash table.

Hash Tables

Fixed-size Hash Tables

Much like hash functions there are multiple approaches we can take when designing hash tables, each of which will have consequences on performance. In this book we'll talk about two types - *fixed-size hash tables* and *dynamic hash tables*. Firstly, we'll talk about fixed-size hash tables. In the previous section we look at how has tables worked - essentially an array or list and using a key which is passed through a hash function we can find the index of that list or array that we should use to store a data item at or retrieve from.

We'll create a list of size 256 and initialize all values to *None*. We'll then create methods that will allow us to store and retrieve data.

```
class HashTable:
    def __init__(self):
        self.size = 256
        self.buckets = [None for i in range(self.size)]
```

In the above code, the holders for our data items are called *buckets* this is just a Python list with 256 entries. Next, we want to create our hash function. We will use the function we created in the previous section but modify it to *%* (mod) the value with the table size so that we can create a valid index in *buckets*.

```
class HashTable:
    def __init__(self):
        self.size = 256
        self.buckets = [None for i in range(self.size)]

    def _hash(self, s):
        mult = 1
        hash_val = 0
        for ch in s:
            hash_val += mult * ord(ch)
            mult += 1
        return hash_val % self.size
```

Now we want to create methods to *put* data and *get* data. We'll start with the put method. We are assuming here that keys here are strings as our hash function must take strings in order to *ord()* the characters in them. For space, I'll be omitting functions we've already added but they are still there:

```
class HashTable:
    def __init__(self):
        self.size = 256
        self.buckets = [None for i in range(self.size)]

    def put(self, key, value):
        h = self._hash(key)
        self.buckets[h] = (key, value)
```

We are assuming here that collisions don't occur. When they do, we just replace the item at that index. However, we'll need to handle collisions in the next section but for now assume a "perfect" hash table. You may be wondering why we store a tuple of the key and value. There is good reason for this, and you'll see why we do this when we talk about dynamic hash tables.

Next, we'll create a method that allows us to retrieve data:

```
class HashTable:
    def __init__(self):
        self.size = 256
        self.buckets = [None for i in range(self.size)]

    def get(self, key):
        h = self._hash(key)
        return self.buckets[h][1] # The value is at index 1 of the tup
le
```

Let's take a look at our hash table in action. We'll be using it to map countries to their capital cities:

```
ht = HashTable()
ht.put("England", "London")
ht.put("France", "Paris")
print(ht.get("France"))
print(ht.get("England"))

# OUTPUT
(France, Paris)
(England, London)
```

We can add two extra special methods that will allow us to use *[]* when storing and retrieving data. In other words, we will be able to store data as *ht["England"] = "London"* and we can fetch data with *print(ht["England"])*. These special methods are *__setitem__* and *__getitem__*.

We can overload these to provide the functionality we want. Let's look at how that's done:

```
class HashTable:
    def __init__(self):
        self.size = 256
        self.buckets = [None for i in range(self.size)]

    def __setitem__(self, key, value):
        self.put(key, value)
    def __getitem__(self, key):
        return self.get(key)
```

Now we can use our hash table as follows:

```
ht = HashTable()
ht["England"] = "London"
ht["France"] = "Paris"
print(ht["France"])
print(ht["England"])

# OUTPUT
(France, Paris)
(England, London)
```

Dynamic Hash Tables

Now consider what happens when the hash table becomes full. If we still assume that collisions won't occur, how do we handle a full hash table? Should we just kick elements out as new ones arrive? One approach would be to resize the hash table as it becomes fuller. There are a couple of ways of handling this. The way we will solve this problem is by using something called the *load factor*. This is just a measure of how full the table is. And the formula for it is:

$$\text{load factor} = \frac{k}{n}$$

Where *n* is the total number of buckets - 256 in our case, and *k* is the is the number of buckets that are filled. We could resize when our hash table becomes 70% full (when the load factor is *0.7*). In order to do this, we have to perform an operation called re-hashing were we must re-hash all keys of all elements in the table so that we can put them in their correct locations in the newly resized hash table. To handle resizing, a good approach is to simply double the size of the hash table when resizing. This is an expensive operation, and we wouldn't want to be doing it too often, especially as the hash table grows in size. A better approach than using the load factor alone is to calculate the time it takes for a hash table to reach the load factor; we can use this to drive a decision on how much we should grow the hash table by. However, we won't be doing that here and we'll simply be using the load factor.

Each time we add a new item to the hash table, we need to calculate the new load factor. When it reaches *0.7* we must resize. To do this we create a new bucket array, rehash all the keys in the table and re-assign them to new buckets in the new bucket array then replace the old bucket array with the new one.

Firstly, let's modify our *put* method to handle calculating the load factor and potentially triggering a re-sizing:

```
class HashTable:
    LOAD_FACTOR_LIMIT = 0.7
    def __init__(self):
        self.size = 256
        self.buckets = [None for i in range(self.size)]
        self.count = 0
        self.load_factor = 0

    def put(self, key, value):
        h = self._hash(key)
        self.buckets[h] = (key, value)

        # Update counts
        self.count += 1
        self.load_factor = self.count / self.size

        # Check if resizing is needed
        if self.load_factor >= HashTable.LOAD_FACTOR_LIMIT:
            self._resize()
```

We've added a few new attributes here. We've added *count* to track the number of items in the hash table, *load_factor* to track the load factor, and we've added a call to *_resize()*. This is the next method we need to implement.

```
class HashTable:
    LOAD_FACTOR_LIMIT = 0.7

    def __init__(self):
        self.size = 256
        self.buckets = [None for i in range(self.size)]
        self.count = 0
        self.load_factor = 0

    def _resize(self):
        # Create a new buckets array thats twice the size
        new_buckets = [None for i in range(self.size * 2)]

        # Rehash all keys
        for item in self.buckets:
            if item is None:
                continue
            h = self._hash(item[0])
            new_buckets[h] = (item[0], item[1])
        self.buckets = new_buckets
        self.size = len(self.buckets)
        self.load_factor = self.count / self.size
```

We can now test our resizing method. For the sake of example, I've reduced the hash table initial size to 10. Therefore, resizing should occur when we add the 7th item to the hash table.

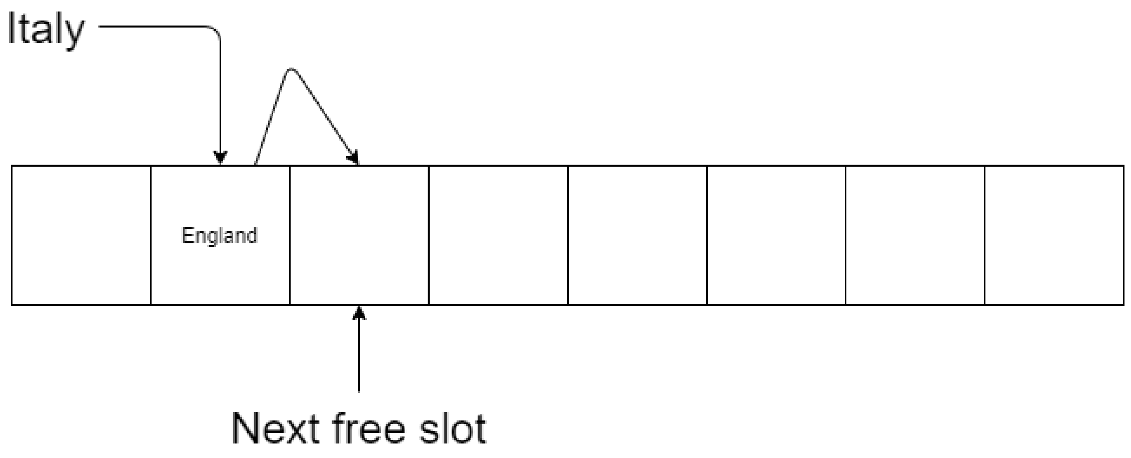
It's also very important to remember here - we've added 7 items, but we may not have 7 items in the hash table as there may have been collisions that have overwritten previous items and therefore, if you try to access some items you might get an error but we're ignoring that here and we'll handle this situation in the next section.

Collision Handling Schemes

Open Addressing

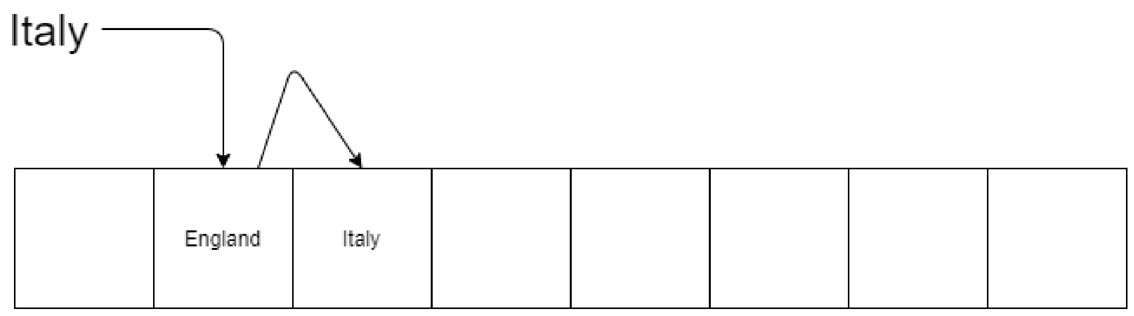
In this section we're going to take a look at various ways of handling collisions and dealing with some of the problems we experienced in the previous section. In this section we'll look at an open addressing collision handling scheme called linear probing. Open addressing is a method of collision resolution in hash tables by which collisions are resolved by probing or searching through alternate locations in the bucket array until either the target item is found or a free bucket is found which indicates there is not such key in the table.

Linear probing is one approach to this in which we move along linearly across the buckets until a free bucket is found and we store the data item there. Below is an illustration of how Linear probing works:



We can see that we tried to add `Italy` but when passed through the hash function it's hash was *1* which already contained *England* so we move along the buckets linearly in order to find the next available slot.

Below is an illustration of what happens when we perform a lookup?



When performing a lookup for *Italy*, the key is hashed, and the hash is *1*. The bucket at index *1* contains *England* so we move along to the next bucket and check if the keys match. If they do, then we've found the item we're looking for. If we don't find it then we move onto the next bucket and so on. If we haven't found the item we're looking for and we reach an empty bucket, then the item is not in the hash table.

To implement this, we need to update our *get()* and *put()* methods. We'll start with the *put()* method.

```
class HashTable:
    LOAD_FACTOR_LIMIT = 0.7

    def __init__(self):
        self.size = 256
        self.buckets = [None for i in range(self.size)]
        self.count = 0
        self.load_factor = 0

    def put(self, key, value):
        self.count += 1
        h = self._hash(key)
        # Find an empty bucket
        while self.buckets[h] is not None:
            # If item already exists, break
            if self.buckets[h][0] == key:
                self.count -= 1
                break
            # Get index of next bucket
            h = self._increment_key(h)

        # Store item
          self.buckets[h] = (key, value)

        # Check if resizing is needed
          self.load_factor = self.count / self.size
        if self.load_factor >= HashTable.LOAD_FACTOR_LIMIT:
            self._resize()

    def _increment_key(self, key):
        return (key + 1) % self.size
```

We've done a couple of new things here. Firstly, we've introduced this new *_increment_key()* function that handles incrementing the hash. We can't just increase the hash by 1 as we'd eventually run into an error when the hash goes out of bounds of the buckets array possible indexes. This function allows us to loop back around to the start. We've also added the *while* loop that searches for the next available bucket. We don't need to worry about an infinite loop here as the table will never become full as we resize when it becomes 70% full so there will be an empty bucket.

Next we'll update our *get()* method. There are a few cases we need to handle here so I'll comment the code then step through the changes afterwards:

```
class HashTable:
    LOAD_FACTOR_LIMIT = 0.7
    def __init__(self):
        self.size = 10
        self.buckets = [None for i in range(self.size)]
        self.count = 0
        self.load_factor = 0

    def get(self, key):
        h = self._hash(key)

        # Item is not in the hash table
        if self.buckets[h] is None:
            return

        if self.buckets[h][0] != key:
            # Will help us know when we've gone full circle
            original_key = h
            while self.buckets[h][0] != key:
                h = self._increment_key(h)
                # We've encountered an empty bucket, item doesnt exist
                if self.buckets[h] is None:
                    return
                # We've gone full circle, item doesn't exist
                if h == original_key:
                    return
          # Found the item, return it
        return self.buckets[h]
```

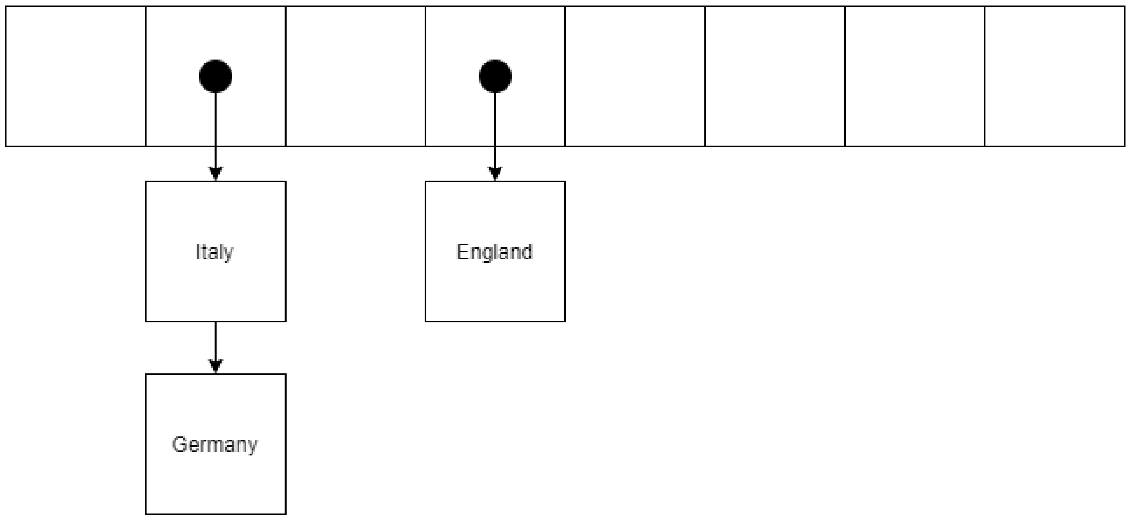
The first change is if the bucket is empty, then the item cannot exist. The second is that we need to iterate over the coming buckets and if we find an empty bucket then the item doesn't exist. We use the *original_key* variable to track if we've gone to the end of the hash table, back to the start of the buckets array and ended up back at our starting point.

Linear probing can be prone to certain issues however, the first is *clustering* That is, we tend to find clusters of items with gaps of empty buckets. This can degrade our search time and in time as the hash table fills up, the lookup and entry of data will diverge from O(1) and tend toward O(n). To prevent clusters, we can use a technique called quadratic probing I won't be covering it in detail, but instead of incrementing our key by *1* we increase it by a power of two. This will cause the data in buckets that were clustered when using linear probing to become spread out across the table.

Separate Chaining

The next collision-handling scheme we'll look at is called separate chainingThis technique works by initializing each of our buckets with an empty linked list. When inserting, we hash the key and insert the data item into the linked list at that bucket location. When searching, we hash the key and search the linked list at that location. If the item isn't in the linked list, then it doesn't exist. This approach will cause a more radical change to our hash table, so we'll start from scratch again. I will be moving through this section a bit faster as you should be familiar with the operations of both linked lists and hash tables at this point.

Below is an illustration of what our hash table looks like when using separate chaining as the collision-handling scheme.



And here is the code of how we do that. We can assume the code for the linked list exists. We won't be handling resizing here as this is to demonstrate how separate chaining works but I encourage you to try implement resizing with separate chaining yourself as an exercise.

For the following code samples, I'll be omitting functions such as *_hash*, and *__setitem__* and *__getitem__*.

```
class HashTable:
    def __init__(self):
        self.size = 10
        self.buckets = [LinkedList() for i in range(self.size)]
```

For the purpose of this example we'll limit the table size to 10.

Next we'll implement the *put()* functionality:

```
class HashTable:
    def __init__(self):
        self.size = 10
        self.buckets = [LinkedList() for i in range(self.size)]

    def put(self, key, value):
        h = self._hash(key)
        linked_list = self.buckets[h]
        if not linked_list.contains(key):
            linked_list.insert((key, value))
```

To handle putting items, we simply insert the item into the linked list at the bucket the key hashed to. We have this new *contains* method for the linked list which we didn't look at before, so I encourage you to implement that functionality yourself. Also, you may want to change the Node class for the linked list so that it contains three attributes - *key*, *value*, and *next*, this can make it much easier for you to perform searches in the context of hash tables using singly linked lists for separate chaining.

Next we'll look at the *get()* method:

```
class HashTable:
    def __init__(self):
        self.size = 10
        self.buckets = [LinkedList() for i in range(self.size)]

    def get(self, key):
        h = self._hash(key)
        linked_list = self.buckets[h]
        return linked_list.get(key)
```

Here we simply retrieve the item from the linked list at the bucket the key hashed to (we return *None* if it is not in the list).

We have only scratched the surface of hash tables in this chapter. There is so much more we can do with them in terms of performance increases, how we handle collisions etc. If you have enjoyed this section, I encourage you to take a look at other hash functions, how good hash functions were created, when to choose one over the other, different collision handling schemes etc.

Hash Table Applications

Hash tables are used just about everywhere. They're one of the most important data structures that we have. Their applications are the same as anything that Python dictionaries are used for. Below are just a few examples of where they are used.

- File systems
- Password verification
- Pattern matching
- General performance increases to applications

The list is endless. Anywhere you can think that a Python dictionary would works - a hash table would work there too.

Searching Algorithms

Linear Search

The first search algorithm we'll be looking at is linear search. It is a relatively straight forward algorithm and you have probably used it before even if you didn't know you were using it. I debated including it in this book because it's so simple, but I decided to include it for completeness.

Linear search (or sequential search) is a method to find an element in something. That something may be a string, a file or a list. The element we are searching for could be anything (a letter, a number, a word, etc.).

(Usually) we are searching for the position of the element we want. Therefore, in some cases, the element we are searching for may not exist and we need our algorithm to be able to handle that.

Linear search is commonly used in computer science and is an easy algorithm to understand and write and it's perfect for us at this stage.

In technical terms, we are searching for the position of some element Q that satisfies some property P.

In the general case, we can achieve this as follows:

```
i = 0
while i < N and not P:
    i += 1
```

What we want is the position of the element Q that satisfies some property P so we continue to search as long as that property P is not satisfied and we stop when it is.

Let's look at an example to help clarify. In this example we want to find the position of the first occurrence of the letter "W":

```
s = "Hello World"

i = 0
while i < len (s) and s[i] != "W":
    i += 1
```

What we do here is, begin at index 0 ("H") and continue through the string, checking each index until "W" is found and that is linear search.

Now we have another problem. There are two conditions in the while loop. Either "W" is found, and we exit the loop or we search the entire string and "W" is not found. How do we know which condition caused the loop to terminate?

If condition 1 (*i < len(s)*) is *True* then "W" was found as we clearly didn't reach the end of the string OR condition 1 is *False* in which case we did reach the end of the string and "W" was not found.

We can check this simply:

```
s = "Hello World"

i = 0
while i < len (s) and s[i] != "W":
    i += 1

if i < len (s):
    print ("The letter 'W' was found at position " + str (i) + " in the
string" )
else :
    print ("The letter 'W' was not found in the string" )
```

This is usually how linear search works, however, we can have more complicated versions which we will get to in the next section.

Important Note:

If you have some previous experience of programming, you may be wondering why I'm using while loops to do this? I'm doing it this way to promote computational thinking and, in the future,, we will move away from this approach.

In this section I'm going to go through some more examples of linear search using *while* loops, some of which may get a little complicated so spend some time going through them.

For our first example, let's recreate the *lstrip()* string method.

```
s = input ()

i = 0
while i < len (s) and s[i] == " " :
    i += 1

if i < len (s):
    print (s[i:])
else :
    print ("No alpha - numeric characters exist in this string" )
```

The above program will strip all leading white space characters.

Let's take a look at a more difficult example. For this problem we want the user to input a string and print out that string, removing all leading and trailing whitespace and only one white space character between each word.

```
# INPUT
"       This    is a   string       with  lots  of whitespace  "

# OUTPUT
"This is a string with lots of whitespace"
```

This is difficult to solve and can remember being given this problem when I was learning to program. To solve this problem, we'll need to nest while loops inside each other.

```
s  = input  ()
output   = ""

i  = 0
while   i  <  len (s):
    # Find a non whitespace character (start of a word)
    while   i  <  len (s)   and  s[i]     == " "  :
        i   += 1

    j  = i
    if   i  <  len (s):
        # Find the end of that word
        while   j  <  len (s)   and  s[j]     !=  " "  :
            j   += 1

        # Build up a string from the original without the excess white
space
        output   += " "    +  s[i:j]

    i  = j  + 1


# Print out the final string
print   (output[   1:])
```

I'm not going to explain this, I want you to work through it. Take a simple input string to help you walk through the program.

Binary Search

Binary search is an algorithm used to find elements in a list by repeatedly halving the amount of data to be searched, thereby reducing the time taken to find an element. There is one condition that must hold true before we can carry out a binary search - the list must be sorted

The term binary search comes from the fact that we always consider the list to have two parts and we make a guess at which half of the list the element will exist in. In the case of a list of sorted integers we start in the middle of the list. We then look at the element in the middle of the list. If the element we're searching for is less than that element, then the element must be in the first half of the list (if it exists in the list at all, but if it did that's where it must be). When we have taken a guess at which half of the list the element is a part of, we can discard the other half of the list. We then repeat this cycle until the element is found or until we can't split the list in half anymore.

Let me illustrate how binary search works:

```
2  4  9  12  34  35  77
|______________________|  # The number we were trying to guess                is in he
re

2  4  9  12          34  35  77
| low              | high
|______________________|        # We will call these two position low a
nd high

2  4  9  12  34  35  77
| low              | high
|______________________|        # The number is somewhere be              tween low a
nd high

2    4  9  12  34  35  77
| low              | high   # We're also able to calculate the inde
x for
|______________________|        # the number in the middle of low and h
igh

2  4  9  12  34  35  77
| low     | middle     | high   # We                     're also able to calcul            ate the ind
ex for
|_________|_____________|        # the number in the middle of low and
high

#
# Assume the number we're searching for is 4
#

2  4  9  12  34  35  77
| low     | middle     | high   # We're also able to calculate th                                   e

                                          # ind  ex for
|_________|_____________|        # the number in the middle of low and

                                          High
```

```
2  4  9  12  34  35  77
| low     | middle     | high   # As the list is sorted we can check i
f the
|_________|___________ _|       # number at 'middle' is <= or > 4

2  4  9  12  34  35  77
| low     | middle     | high   # In this case 4 is < 12 so we don't

                                         # n eed to
|_________|____________|        # bother searching               anything above

                                         # midd le index

2  4  9  12
| low     | high                # So we cut that portion of the list

                                         #  out
|_________|                     # and make middle the new high

2  4  9       12
| low     | high                           # Repeat the process,
|_________|                     # continuously halving the list until

                                         #  the condition
                                         # that low < high fails i.e

                                         #  low ==   high
```

So, our approach here is:

- Find the midpoint between *high* and *Low*
- Adjust *Low* or *high* depending on whether what we're searching for is less than or equal to (<=) or greater than (>) the midpoint value.

We calculate the midpoint by getting the average of low and high i.e. *(Low + high) // 2*. Notice we're doing integer division here, so we'll round down if there is a decimal place.

In the next section we'll look at implementing *binary search*

Implementing Binary Search

In the previous section we looked at how binary search works. Now we need to code it.

Roughly, only 10% of developers can code binary search properly. There is a little cop out in the middle of it, so we need to be careful so that our solution works correctly in all cases.

Below is the commented implementation of binary search:

```
def binary_search(arr, elem):
    low = 0                    # Define initial low value.
    high = len(arr)              # Define initial high value.

    while lo    w < high:           # The condition that must hold                          true.
        mid = (low + high) // 2  # Calculate the mid index.

        if arr[mid] < elem:        # Check if the value is in first

                                                # or second half.
            low = mid + 1          # Update low value i                      f

                                                # mid_val < elem (in second half).
        else:
            high = mid             # Otherwise update high value

                                                # (elem in first half).
    return low                                  # Return the position of the

                                                # element
                                                # we're searching for.
```

It is important that *mid* and *high* are never equal. This is one place where some developers mess up when coding this algorithm.

We must add *1* when updating the low value because if *Low == mid*, we will end up in an infinite loop and that is bad news! This is another place where developers mess up.

Since binary search has it's cop outs like those above, so I recommend you learn it off by heart (it can be a common interview questions).

Analysis of Binary Search

In this section I'm going to look at Binary Search's runtime complexity.

Binary Search has a runtime complexity of:

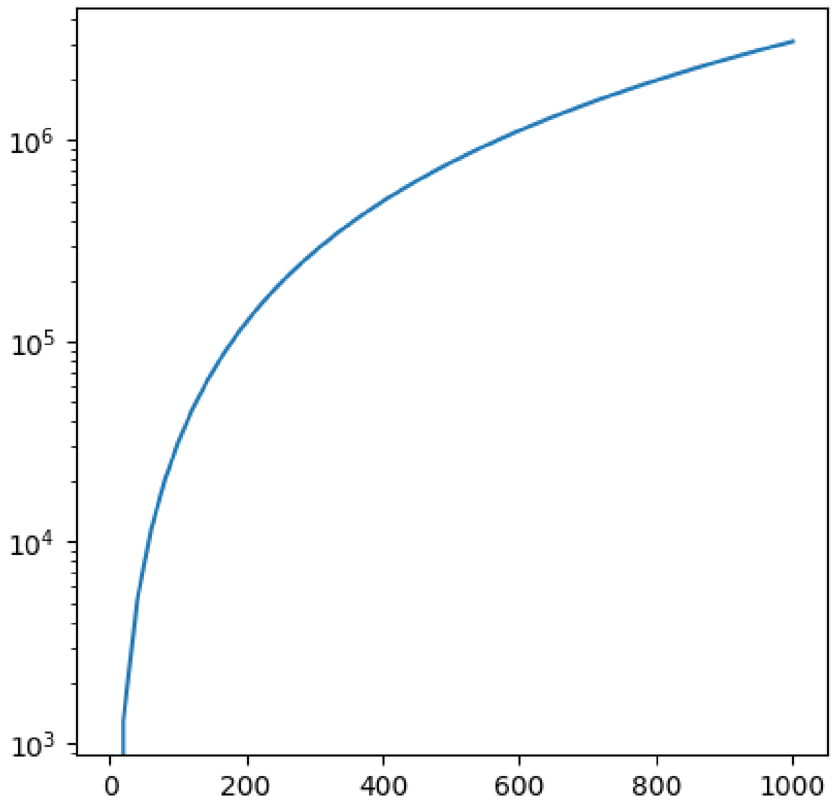
ὕ ὰ έ ῭Ω

More specifically:

ὕ ὰ῭Ὡ ὲ

It is a little trickier to tell what the runtime of an algorithm like this is. As a general rule of thumb, if we halve the input upon each iteration then it will have a logarithmic runtime (or at least, a logarithmic component to its runtime complexity).

Here is the graph of an algorithm with binary search's runtime complexity.



You can clearly see that as the input get's much larger (x-axis), the time it takes for the algorithm to complete begins to level off and increasing the input size begins to have a smaller and smaller effect on the algorithm's performance.

Binary Search's space complexity is also O(1), which is constant memory, as we don't create any copies of the list when searching.

This is considered a very efficient algorithm as it's runtime complexity is pretty close to constant in terms of the Big-O "scale" and the space complexity is constant.

Sorting Algorithms

Sorting is the process of arranging items systematically. In computer science, sorting refers to arranging items (whatever they may be) in an ordered sequence.

Sorting is a very common operation in applications and developing *efficient* algorithms to perform sorting is becoming an ever more important task as the amount of data we gather grows at an exponential rate.

Sorting is usually used to support making lookup or search efficient, to enable processing of data in a defined order possible and to make the merging of sequences efficient.

In this chapter we're going to look at many of the most common sorting algorithms.

Basic Sorting Algorithms

We're going to start off by taking a look at two simple yet important sorting algorithms. These algorithms were some of the early attempts at sorting and led to the development of more advanced algorithms that we'll look at in the next section.

Selection Sort

Selection sort is a general-purpose sorting algorithm. For selection sort to work we must assume there exists some sequence of elements that are order-able (integers, floats, etc.).

Selection sort is an *in-place* sorting algorithm which means we don't build a new list, rather, we rearrange the elements of that list.

Before we begin, we need to look at some *terminology*.

- Sorted subarray - The portion of the list that has been sorted.
- Unsorted subarray - The portion of the list that has not yet been sorted.
- *a* - The list we are sorting.

In selection sort, we break the list into two parts, the sorted subarray and the unsorted subarray and all of the elements in the sorted subarray are less than or equal to all the elements in the unsorted subarray. We also begin with the assumption that the entire list is unsorted.

Next, we search the entire list for the position of the smallest element. We must search the entire list to be certain that we have found the smallest element. If there are multiple occurrences of the smallest element, we take the position of the first one. We then move that element into the correct position of the sorted subarray. We repeat the above steps on the unsorted subarray.

This is illustrated below.

```
   6    3    9    7    2                    8
|||========================|         # FIND THE POSITION OF THE

                                              #  SMALLEST
       UNSORTED PART                # ELEMENT IN THE LIST AND SWAP

                                              #  WITH 6

--------      -----    -------------------------------------------------------

   2    3    9    7    6    8         # 2 IS NOW IN THE CORRECT

                                              #  POSTION
   |===||===================|         # FIND THE POSITION OF THE

                                              #  SMALLEST
                UNSORTED PART            # ELEMENT IN THE LIST AND SWAP

                                              #  WITH 3
                                              # 3 IS THE SMALLEST (NO SWAP)
-----------------------------                    -----    ------------------------------------

   2    3    9    7    6    8         # 3 IS NOW IN THE CORRECT

                                              #  POSITION
   |========||==============|         # FIND THE POSITION OF THE

                                              #  SMALLEST
      SORTED    UNSORTED PART          # ELEMENT IN THE LIST AND SWAP

                                              #  WITH 9
       PART

- -------------------------------------------------------------------

   2    3    6    7    9                    8         # 6 IS NOW IN THE CORRECT

                                              #  POSITION
   |============||=========|         # FIND THE POSITION OF THE

                                              #  SMALLEST
      SORTED PART    UNSORTED          # ELEMENT       IN THE LIST AND SWAP

                                              #  WITH 7
                              PART            # 7 IS THE SMALLEST (NO SWAP)

----   -----------------------------------------------------------------
```

```
   2   3   6   7   9   8                        # 7 IS NOW IN THE CORRECT
                                                #  POSITION
   |=================||=====|          # FIND THE POSITION OF THE
                                                #  SMALLEST
        SORTED PART     UNSORTED       # ELEMENT IN      THE LIST AND SWAP
                                                #  WITH 9
----------------------------------------------------------------------
   2   3        6   7   8   9         # 8 IS NOW IN THE CORRECT
                                                #  POSITION
   |=======================|||         # WE HAVE REACHED THE END OF
                                                #  THE LIST
           SORTED PART                          # THE LIST IS NOW SORTED
```

It's time to code selection sort! Let's look at the code for it below:

```
i  = 0
while   i  <  len (a):
    p  =  i
    j  =  i  +  1
    while   j  <  len (a):
        if   a[j]    <  a[p]:
            p  =  j
        j  +=  1

    tmp  =  a[p]
    a[p]   =  a[i]
    a[i]    =  tmp

    i  +=  1
```

That's it! Let's break this down as there is a lot going on here.

- The outer while loop is controlling our *sorted subarray*
- The inner while loop is searching for the next smallest element.
- The three lines above `i += 1` are swapping the smallest element into the correct position.
- The process then repeats.

A skill that all programmers have is being able to step through code and figure out how it's working. I encourage you to do the same with this algorithm (pen and paper and walking through an example)

Another good way to figure out what's going in the middle of some code is to stick in a *`print()`* statement to print out what variables hold what values at that current time. Another good way is to set break points. I'm not going to show you how to do this as it's generally a feature of the text editor you're using so look up how to set break points for your editor and how to use them! It's a skill you'd be expected to have if you worked in this field.

Insertion sort

Insertion sort is another general purpose, *in place*sorting algorithm.

To contrast how insertion sort works compared to selection sort:

- Selection Sort Select the smallest element from the unsorted subarray and append it to the sorted subarray of the list.
- Insertion Sort Take the next element in the unsorted subarray and insert it into it's correct position in the sorted subarray.

I'll illustrate this with a semi complete example:

```
2   4   5   3   9    6
|=============||==========|     # THE STATE OF THE LIST AT SOME POINT
     SORTED      UNSORTED       # DURING THE INSERTION SORT PROCESS

---------     ------------------------------                   ------------------------------
-

             This is the next element we want to sort
                           |
                           |
2   4   5   3   9    6
|=============||==========|     # WE WANT TO NOW PLACE THE 3 IN IT'S
     SORTED      UNSORTED       # CORRECT POSITION

----------------------------------------------------------------------
-
                           3
2   4   5   _   9    6
|=============||==========|     # TAKE 3 OUT OF THE LIST
     SORTED      UNS  ORTED

----------------------------------------------------------------------
-
```

```
                  3
2   4   5   _   9    6
|============||==========|     # IS 3 < 5?          YES, SO MOVE 5 UP
      SORTED      UNSORTED

----------------          ------------------------------------------------------
-

            3
2   4   _   5   9    6
|============||==========|     # IS 3 < 4? YES, SO MOVE 4 UP
      SORTED      UNSORTED

------------------------------------------------------------------                                   -----
-

      3
2   _   4   5   9    6
|============||==========|     # IS 3 < 2? NO, PLACE AFTER THE 2
      SORTED      UNSORTED

-------------------------------------------                           ---------------------------
-


2   3   4   5   9    6
|=== ==========||==========|     # 3 IS NOW IN THE CORRECT POSITION
      SORTED      UNSORTED

-----------------------------------------------------------------------
-


2   3   4           5   9    6
|=================||=====|     # MOVE ONTO THE NEXT    ELEMENT
      SORTED         UNSORTED
```

It's time to code insertion sort! Let's take a look at the code for it, I'll give the explanation as comments:

```
i  = 1          # Assume the first element (a[0]) is sorted
while  i  <  len (a):     # Same as with selection sort
    v  =  a[i]          # The value we want to insert into the correct posi
tion
    p  =  i             # The position the element should be inserted into
to
```

```
    while   p > 0  and  v  <  a[p - 1]:      # While value is < element to left
        a[p]   = a[p - 1]                    # Move the element to left up
        p  -= 1                              # Decrement p (move one position lef
t)
    a[p]   = v          # Found correct position so insert the value

    i  += 1             # Move to next element
```

Analysis of Selection sort and Insertion sort

In this section I'm going to talk about the algorithmic complexity of the two algorithms. This is a really important section. If you ever do a technical interview, in 99.9% of cases you'll be asked to write some code or an algorithm and give it's algorithmic complexity. You need to know this stuff and it isn't just for Python, this applies to any language! I'm going to explain this in the simplest way possible while still giving you a good understanding of the topic.

Selection sort and Insertion sort are what we call *quadratic sorting algorithms*

This means they both have a Big-O time complexity of:

Ǚ ὲ˙

Time complexityrefers to the time it takes for the algorithm to complete. There are generally three categories:

- Best case scenarioFor example with insertion sort, let's say our list is already sorted and we try to sort it, then we are in a best-case scenario as we don't need to move any elements around and the algorithm finishes quickly.
- Average caseThis is how the algorithm performs in terms of time on average.
- Worst case scenarioThis is how the algorithm would perform in terms of time if our list were completely scrambled and out of order.

Big Odeals with the worst case and it is the case we are usually concerned with!

There's a lot of math behind this and there's a whole topic of computer science related to it so I'm not covering it in detail here.

Anyway, when we say that both of these algorithms have a time complexity of *O(n^2)*, we are essentially saying that the performance of the algorithms will grow proportionally to the square of the size of the input (lists in this case). Another way to think of *O(n^2)*is that if we double the size of our input, it will quadruple the time taken for the algorithm to complete. These are just estimating. It doesn't mean that two algorithms with time complexities of *O(n^2)*will both take the exact same time to complete. You can look at this as a guide as to how your algorithm will perform in a general sense.

I've taken the time to run some algorithms with different time complexities on lists of varying sizes and we can look at how the algorithm takes longer to complete as the size of the list grows. You can see this in the diagram below:

FILL IN

In this diagram, the y-axis represents how long it took the algorithm to finish in seconds and the x-axis shows the number of elements in the list.

The blue line is for an O(n^2) algorithm and the orange is for an O(n) algorithm (Linear time algorithm).

I have a fast computer, so it doesn't appear that the orange line is changing (it is, just very slowly).

However, it is very clear that the O(n^2) algorithms become very slow when the size of the input becomes large.

In the real world, a list of size 1000 is small and selection sort or insertion sort wouldn't cut it. However, that doesn't mean they don't have their uses. In fact, in practice, selection sort and insertion sort outperform the faster algorithms on small lists and some of the fast sorting algorithms will actually switch to these O(n^2) algorithms when they are nearing the end of the sorting process. It turns out the whole process finishes faster when this is done (in some cases).

We also have something called space complexity and this deals with how much memory is taken up by an algorithm. Both of these algorithms have O(1) space complexity. This means they use *constant memory*. They use constant memory as they are *in-place* sorting algorithms. They don't create additional lists to assist during the sorting process.

There is usually a trade-off between time and space complexity. As you can see here, we have constant memory (This is good) but quadratic time complexity (This is bad). We could write some algorithm that is faster but takes more memory. It depends on the problem and the computing resources we have.

Quicksort

In the following sections I'm going to look at a very important sorting algorithm: *quicksort*. Quicksort is a *recursive* sorting algorithm that employs a *divide-and-conquer* strategy. The Python built-in `sorted()` function uses a modified Quicksort algorithm. Divide-and-conquer is a problem solving strategy in which we continuously break down a problem into easier, more manageable sub problems and solve them.

Since this is a divide-and-conquer algorithm we want to take a list of unsorted integers and split the problem down into two easier problems and then break each of those down…. and so on.

To achieve this, I'll first cover quicksort's core operation: partitioning. It works as follows:

```
>>> A = [6, 3, 17, 11, 4, 44, 76, 23, 12, 30]
>>> partition(A,     0, len (A) - 1)
>>> print  (A)
[6, 3, 17, 11, 4, 23, 12, 30, 76, 44]
```

So, what happened here and how does it work? We need to pick some number as our *pivot*. Our partition function takes 3 arguments, the *list*, the *first element*in the list, and the *pivot*. What we are trying to achieve here is that when we partition the list, everything to the left of the pivot is less than the *pivot* and everything to the right is greater than the *pivot*. For the first partition seen above, *30 is our pivot*We'll always take our partition to be the last element in the list. After the partition we see some elements have changed position but everything to the left of 30 is less than it and everything to the right is greater than it.

So, what does that mean for us? Well, it means that 30 is now in its *correct* position in the list ANDwe now have two easier lists to sort. All of this is done *in-place*,so we are not creating new lists.

Let's look at the code:

```
def  partition(A, p, r):
    q = j  = p
    while   j  < r:
        if   A[j]   <= A[r]:
            A[q], A[j]     = A[j], A[q]
                q += 1
        j  += 1
    A[q], A[r]     = A[r], A[q]
    return   q
```

The `return q` at the end isn't necessary for our partition but it is essential for sorting the entire list. The above code works its way across the list *A* and maintains indices *p*, *q*, *j*, *r*.

p is fixed and is the first element in the list. *r* is the *pivot* and is the last element in the list. Elements in the range `A[p:q-1]` are known to be less than or equal to the pivot and everything from `A[q-1:r-1]` are greater than the pivot. The only indices that change are *q* and *j*. At each step we compare `A[j]` with `A[r]`. If it is greater than the pivot it is in the correct position, so we increment *j* and move to the next element. If `A[j]` is less than `A[r]` we swap `A[q]` with `A[j]`. After this swap, we increment *q*, thus extending the range of elements known to be less than or equal to the pivot. We also increment *j* to move to the next element to be processed.

I encourage you to work through an example with a small list on paper to make this clearer.

Now onto the *quicksort*part. Remember it is a *recursive algorithm*so it will continuously call on `partition()` until there is nothing left to partition and all elements are in their correct positions. After the first partition we call *partition()* again, this time we call it on the list of elements to the left of the pivot and the list of elements to the right of the pivot.

Let's look at the code:

```
def  quicksort(A, p, r):
    if   r  <=  p:                    # If r <= p then our list is sorted
       return
    q  =  partition(A, p, r)          # partition incoming list
    quicksort(A, p, q      - 1)       # call quicksort again on everything to le
ft of pivot
    quicksort(A, q     +1, r)         # call quicksort again on everything to ri
ght of pivot
    return    A
```

It's that simple. All we do here is check if the index of the pivot, is less than or equal to the index of the start of our list we want to partition. If it is, we return as whatever list was passed does not need to be partitioned any further.

Otherwise, we partition the list *A*, and call *quicksort*again on the two new *sub lists*

On line 3, we return without specifying anything to return. This is because the quicksort function is a *procedure*

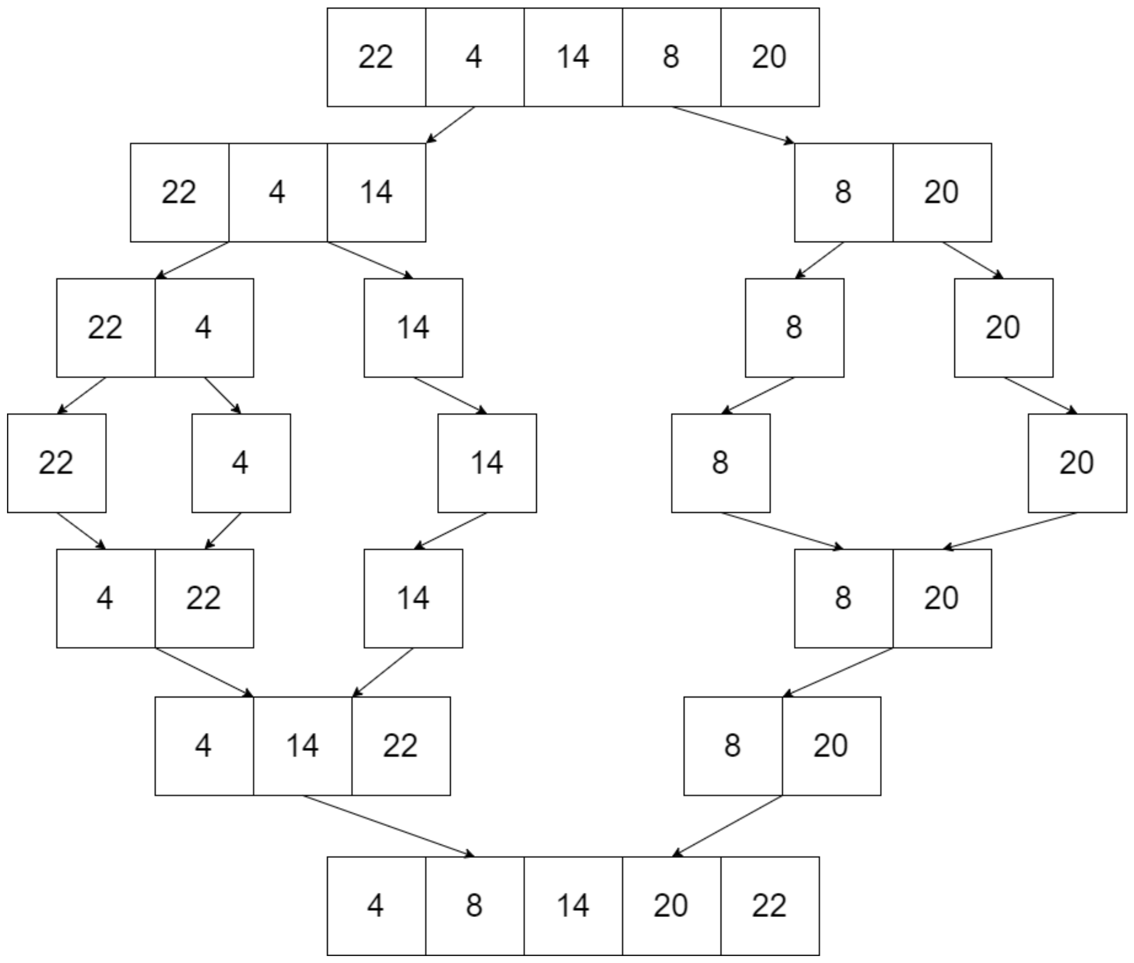
Quicksort works best on large lists that are completely scrambled. It has really bad performance on lists that are almost sorted. Or in Big-O notation, the best case (scrambled) is O(n log(n)) and in the worst case, (almost or completely ordered list) is O(n^2).

Again, I encourage you to try this on paper with a simple list. It will help clarify what is going on.

Merge Sort

The next sorting algorithm we're going to look at is the merge sort algorithm. It's similar to quicksort in the sense that it's a divide and conquer algorithm. Merge sort works by dividing an array that is to be sorted in half. It then recursively calls itself on the two halves, repeated halving these sub arrays until the number of sub arrays is equal to the number of elements in the original array. It then builds the array back up by merging the subarrays, but it merges them so that they're sorted. It repeatedly merges the sub arrays until the array is sorted.

The illustration below shows how this is done.



Let's look at the code for Merge Sort:

```
def  merge_sort(arr):
     if   len (arr)    >  1:
          # Get the mid index (where to split)
          mid  =  len (arr)    //   2

          left_arr      =  arr[:mid]
          right_arr      =  arr[mid:   ]

          merge_sort(left_arr)
          merge_sort(right_arr)

          # Merge two arrays together such that the resulting array is s
orted
          i   = j   = k  = 0
          while   i   <  len (left_arr)       and  j   <  len (right_arry)     @
```

```
if   left_arr[i]        <  right_arr[j]:
    arr[k]     =  left_arr[i      ]
    i  += 1
    else :
        arr[k]     =  right_arr[j]
        j  += 1
    k  += 1

# Handle the case when there is only a single
# array left with no other array to merge with
# in that case - tag them on
while   i  <  len (left_arry):
    arr[k]     =  left_arr[i]
    i  += 1
    k  += 1

while   j  <  len (right_arr):
    arr[k]     =  right_arr[j]
    j  += 1
    k  += 1
```

In the worst case, merge sort uses approximately 39% fewer comparisons than quicksort does in it's average case, and in terms of moves, merge sort's worst case complexity is O(n log n) - the same complexity of quicksort's best case.

On top of that, I personally find this algorithm more intuitive than quicksort and in my experience, this is usually the easier of the two for learners to understand.

Graphs and Graph Algorithms

The concept of graphs comes from the branch of mathematics called graph theory. The graphs we'll be talking about here are not to be confused with things like bar graphs or line graphs.

Graphs are used to solve several problems in computing. They don't have as much structure as the other data structures we've looked at in this book and as such, carrying out things such as traversals are more unconventional.

By the end of this chapter, you should be able to do the following:

- Understand what graphs are and the terminology surrounding them.
- Know the different types of graphs and how they're composed.
- Have a fundamental understanding of how to traverse graphs.
- Be able to find the shortest path between any two vertices in the graph.
- Be able to apply your knowledge of graphs, traversals, and shortest path algorithms to solve real-world problems.
- Understand what minimum spanning trees are and how to construct them.

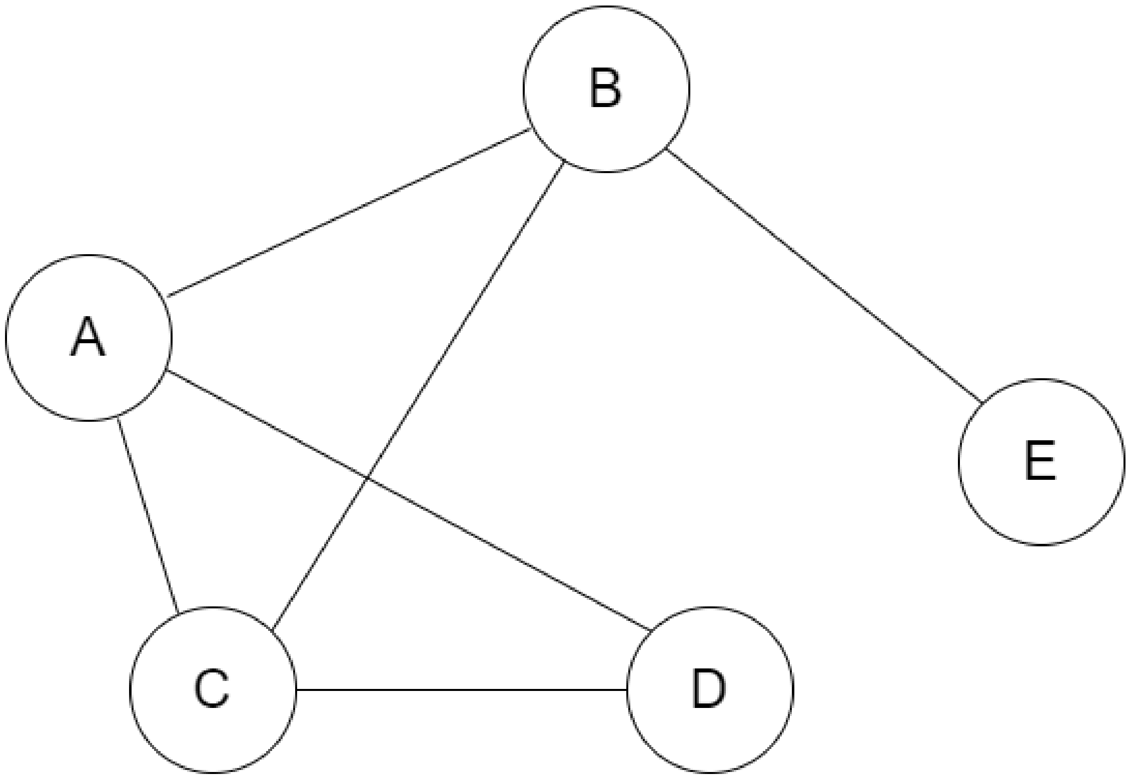
Graph Structures

I haven't gone into much mathematical detail in this book, but I think it is fitting to do so here as I believe the more detailed explanation of them will benefit a lot of you in your understanding. However, before I do that, I'll give an informal definition of what graphs are.

A graph is a set of vertices(nodes) and edgesthat form connections between vertices. That is essentially what they are - much like trees (which are in fact graphs) in the way that the nodes of the tree are connected by "lines", which we now know to be edges

Formally, a graph G is a set V of vertices and a collection E of pairs of vertices from V, called edges. A graph is a way of representing connections or relationships between pairs of objects from some set of vertices, V.

There are three main types of graphs - undirected graphs, directed graphs, and weighted graphs. There are also combinations of those which we'll come to naturally, but we'll look at the three main types in detail first. However, before we do that, look at the below diagram of a basic graph and we'll define some terminology before we continue any further:



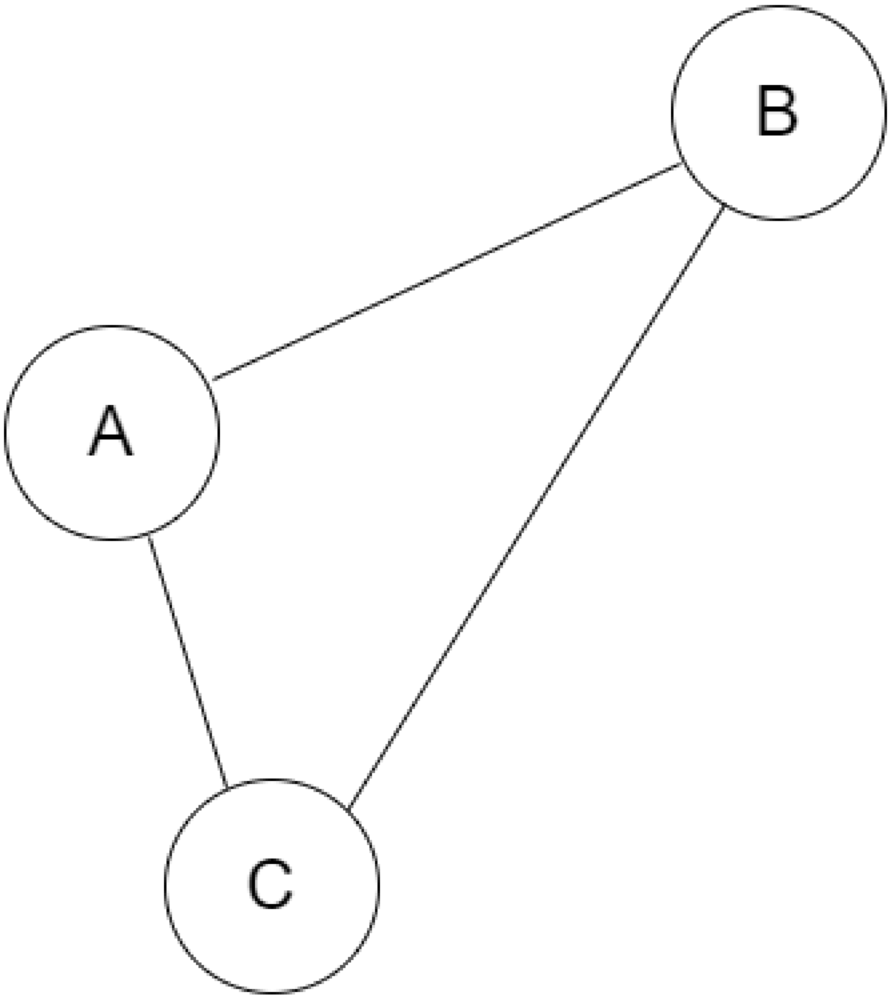
Let's go through 7 terms used in describing the above graphs:

- Vertex: A point, usually represented as a labelled circle (and sometimes a labelled dot) in a graph. The vertices in the above graph are A, B, C, D, and E.
- Edge: A connection between two vertices. From the above graph, the line connecting A and C is an edge. Edges can also connect vertices to themselves. This special case of an edge is often called a loop.
- Degree of a vertex: This is the number of vertices that are directly connected to a vertex by an edge. Above, the degree of the vertex B is 3 as A, C, and E are directly connected to it by edges.
- Adjacency: The adjacency refers to the edges between a vertex and it's neighbor. From above, vertex A is adjacent to C because there is an edge connecting them.
- Path: A sequence of vertices where each adjacent pair is connected by an edge.

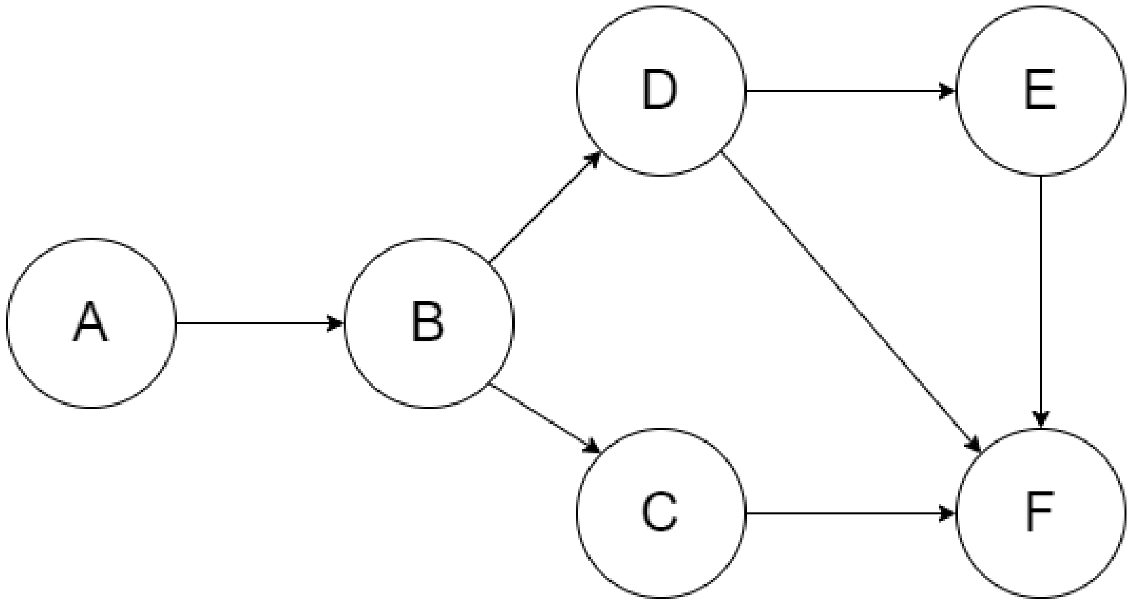
Now that we have the terminology out of the way, let's begin by looking at different types of graphs we can encounter.

Undirected and Directed Graphs

Graphs can fit into two categories, undirected or directed graphs. In an undirected graph, the edges are simply represented as lines between vertices. There is no additional information supplied about the relationship between the vertices other than the fact they are connected in some way. The graph from the previous section is an example of an undirected graph. More formally, a graph is said to be undirected if all edges are not ordered. In other words, if an edge (u, v) is not ordered.

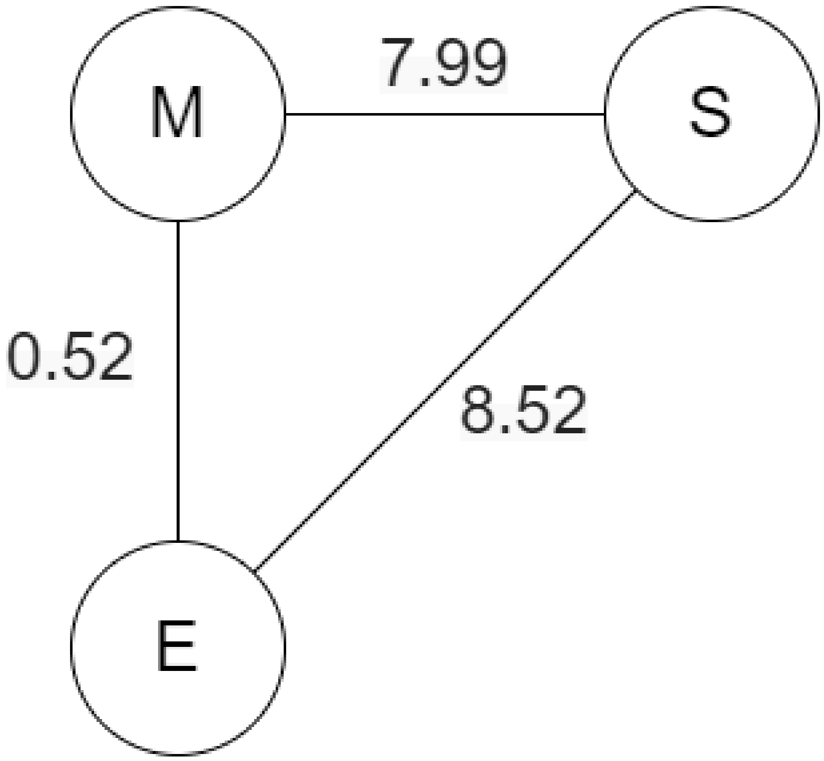


In a directed graph, the edges provide us with more information. They are represented as lines with arrows which will point in the direction the edge connects two vertices. The arrows represent the flow of direction. In the below diagram, one can only move from A to B and not B to A as we could with undirected graphs.



Weighted Graphs

A weighted graph adds even more information to the edges. This can be a numerical value (and will be throughout this book) but it can be whatever you see fit. The below diagram is an undirected weighted graph that represented the distances between Earth, Mars and Saturn. The distances are represented as numerical values between the vertices and are in astronomical units.



When describing graphs, we often represent the relationship as something like EM which in our case represents the edge connecting the vertex E and the vertex M. From the graph we can tell that EM is 8 astronomical units closer than ES.

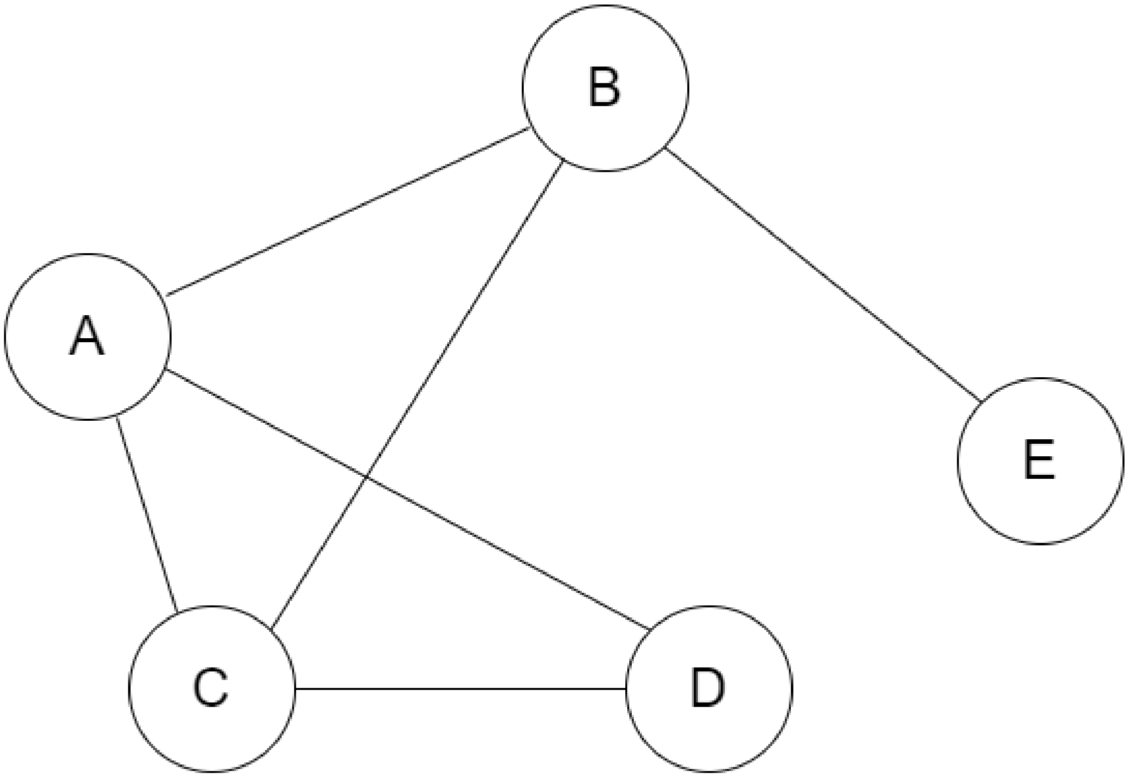
In the above diagram, ES and EMS represent two different paths. A path is simply a sequence of edges you pass through between nodes. Following these paths, we can tell that the journey from Earth to Saturn is a distance of 8.52 astronomical units, but the journey to Saturn from Earth, passing Mars on the way is 8.51 astronomical units. This tells us that we would arrive at Saturn faster if we went to Mars then Saturn rather than straight to Saturn from Earth.

Graphs in Code

We represent graphs slightly differently to how we have represented other data structures. Graphs can be represented in two main ways. One way is to use something called an adjacency matrix; the other way is an adjacency list

Adjacency Lists

A simple dictionary can be used to represent a graph. The keys of the dictionary are each of the vertices in the graph. The value at each key is a list that connects the vertex. For demonstration, take the below graph and we'll represent it as an adjacency list.



The above graph would be represented as:

A - > [B, C, D]
B - > [A, C, E]
C - > [A, B, D]
D - > [A, C]
E - > [B]

Below is how we represent this in code:

```
graph  = {}
graph[ 'A' ]  = [ 'B' ,  'C' ,  'D' ]
graph[ 'B' ]  = [ 'A' ,  'C' ,  'E' ]
graph[ 'C' ]  = [ 'A' ,  'B' ,  'D' ]
graph[ 'D' ]  = [ 'A' ,  'C' ]
graph[ 'E' ]  = [ 'B' ]
```

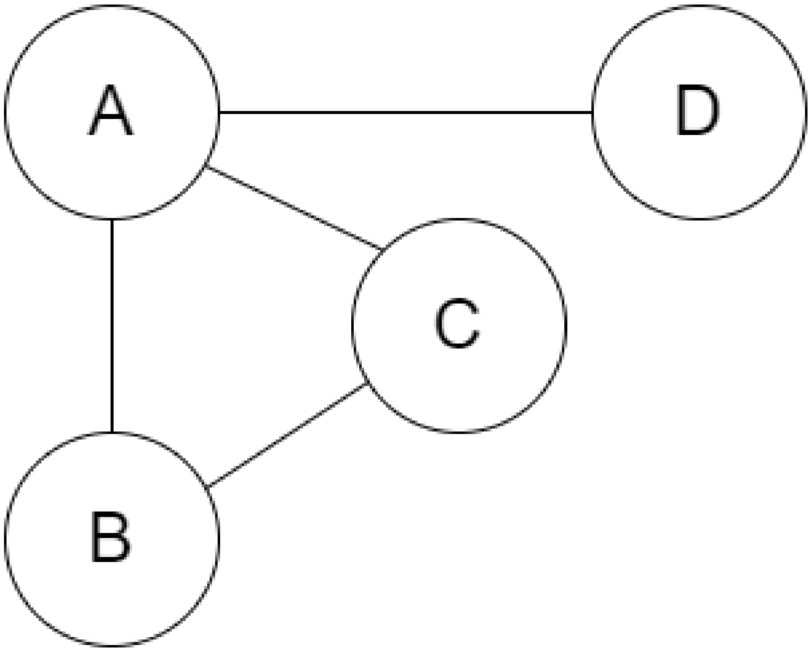
Now we can easily that vertex D has the adjacent vertices A and C whereas vertex E only has vertex B as it's neighbor.

Adjacency Matrix

An adjacency matrix is a way of representing a graph G = {V, E} as a matrix of Booleans. A matrix is a two-dimensional array. Another way of viewing this is as a table. The idea here is to represent the cells with a 1 or 0 depending on whether two vertices are connected by an edge.

The size of the matrix is V x Vwhere V is the number of vertices in the graph and the value of an entry, ö$_{i}$ $_{ċ}$ is either 1 or 0 depending on whether there is an edge from vertex i to vertex j.

Take the below graph as an example.



The adjacency matrix for the above graph would look as follows.

	A	B	C	D
A	0	1	1	1
B	1	0	1	0
C	1	1	0	0
D	1	0	0	0

There are a wide range of pros to using an adjacency matrix to represent a graph. The basic operations such as adding an edge, removing an edge, and checking whether there is an edge between two vertices are very time efficient, constant time operations.

If the graph is dense and the number of edges is large, an adjacency matrix should be the first choice. Even if the graph and the adjacency matrix is sparse, we can represent it using data structures for sparse matrices.

Perhaps the biggest advantage of representing a graph as an adjacency matrix comes from recent advances in computing hardware. The power of newer GPUs enables us to perform even expensive matrix operations in a time efficient manner.

Due to the nature of matrices, we can get important insights into the nature of the graph and the relationships between the vertices within it.

Given an adjacency list, it should be possible to create an adjacency matrix.

The adjacency list for the above graph is as follows:

```
graph  = {}
graph[ 'A ' ]  = [ 'B' ,  'C' ,  'D' ]
graph[ 'B' ]  = [ 'A' ,  'C' ]
graph[ 'C' ]  = [ 'A' ,  'B' ]
grap h[ 'D' ]  = [ 'A' ]
```

To create the adjacency matrix, we need to create a two dimensional array of V x V elements. To do this we will firstly convert the keys of our adjacency list dictionary to a list then define the number of rows and columns from this.

```
matrix_elements = sorted(graph.keys())
cols = rows = len(matrix_elements)

adjacency_matrix = [[0 for x in range(rows)] for y in range(cols)]
```

If we print this out, we'll end up with the following.

```
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
```

Now we need to populate the cells with the correct values. To help us do this we will create an edge list. This is a list of all edges in our graph.

```
edge_list = []

for vertex in matrix_elements:
    for neighbor in graph[vertex]:
        edge_list.append((vertex, neighbor))
```

Printing the edge list will give:

```
('A', 'B')
('A', 'C')
('A', 'D')
('B', 'A')
('B', 'C')
('C', 'A')
('C', 'B')
('D', 'A')
```

Now that we have our edge list, we can easily populate our adjacency matrix with the correct values:

```
for edge in edge_list:
    first_vertex_index = matrix_elements.index(edge[0])
    second_vertex_index = matrix_elements.index(edge[1])
    adjacency_matrix[first_vertex_index][second_vertex_index] = 1
```

Now if we print our adjacency matrix, we get the following:

```
[0, 1, 1, 1]
[1, 0, 1, 0]
```

```
[1, 1, 0, 0]
[1, 0, 0, 0]
```

The above adjacency matrix matches the adjacency matrix we looked at earlier.

The Graph Class

With other data structures we have looked at we had classes to represent the data structures. We can do this here to! Firstly, let us take a look at the Vertex class.

```
class Vertex:
    def __init__(self, element):
        self.element = element

    def element(self):
        return self.element

    # Allow the vertex to be a map/set key
    def __hash__(self):
        return hash(id(self))
```

You should understand most of the above code except for the special hash method. This is a method we can override to allow objects of the class to be keys of dictionaries or sets. The *hash()* function is a Python built-in function that generates a hash - similar to the hash function we created in the Hash Tables chapter. The Python built-in hash function uses a combine's ideas from both polynomial hash codes and a cyclic-shift hash codes. It works on string and tuple types.

In Python, all objects are assigned a unique ID when created. The *id()* function returns the unique ID of an object. In this case, we are getting the ID for the object itself. In total, we are hashing the ID of the object.

Now let's look at the Edge class, this is a little more complicated than the Vertex class.

```
class Edge:
    def __init__(self, u, v, element):
        self.origin = u
        self.destination = v
        self.element = element

    # Gets the vertices on either end of this edge
    def endpoints(self):
        return (self.origin, self.destination)

    # Return the vertex opposite v on this edge
    def opposite(self, v):
        return self.destination if v is self.origin else self.origin
    def element( self ):
        return self .element

    def __hash__( self ):
        return hash(( self .orgin, self .destination))
```

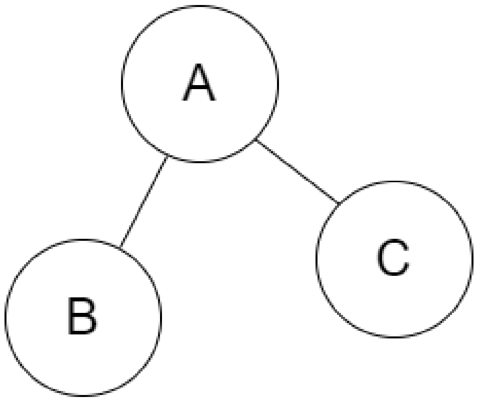
Again, the above code should be self-explanatory, however, you may be confused by the element attribute. We include this for the case our edge is a weighted edge.

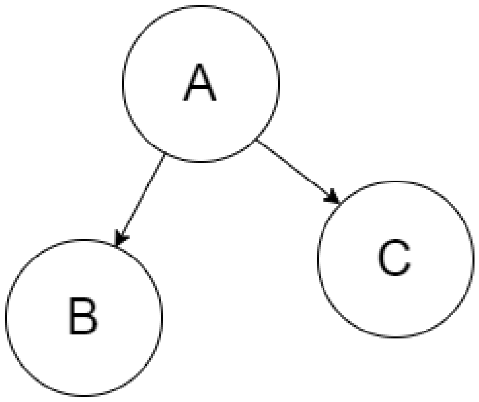
Next, we'll need to code our Graph class. This can get a little complicated as we'll also have to account for both undirected and directed graphs.

A graph will have two dictionary attributes - incoming and outgoing. In the case of directed graphs, we'll have the incoming dictionary, otherwise we'll just need the outgoing dictionary

```
class Graph:
    def __init__ ( self , directed =False ):
        self .outgoing = {}
        # If the graph is directed create a second map, otherwise use
incoming
        # as an alias for the outgoing map
        self .incoming = {} if directed else self .outgoing

    def is_directed( self ):
        # compare if objects are the same
        # if they are, then graph is undirected
        return self .incoming is not self .outgoing

    def vertex_count( self ):
        return len ( self .outgoing)

    def vertices( self ):
        return self .outgoing.keys()

    # Return all outgoing edges incident to vertex v
    def incident_edges( self , v, outgoing =True ):
        adjacent = self .outgoing if outgoing else self .incoming
        for edge in adjacent[v].values():
            yield edge

    # Insert vertex to graph
    def insert_vertex( self , elem =None):
        v = Vertex(elem)
        self .outgoing[v] = { }
        if self .is_directed():
            self .incoming[v] = {}
          return v
    # Add edge between two vertices u and v (optional weight)
    def insert_edge(self, u, v, elem=None):
        e = Edge(u, v, elem)
        self.outgoing[u][v] = e
        # Provides a backwards connection if undirected
        # as incoming points to outgoing if undirected
        self.incoming[v][u] = e

    def generate_adjacency_list(self):
        adjacency_list = {}
        for v in self.vertices():
            edges = list(self.incident_edges(v))
            adjacency_list[v.element()] = [e.opposite(v).element() for
e in edges]
        return adjacency_list

    def generate_adjacency_matrix(self):
        adjacency_list = self.generate_adjacency_list()
        matrix_elements = sorted(adjacency_list.keys())
        cols = rows = len(matrix_elements)

        edge_list = []
        for vertex in matrix_elements:
            for neighbor in adjacency_list[vertex]:
                edge_list.append((vertex, neighbor))

        adjacency_matrix = [[0 for x in range(rows)] for y in range(co
ls)]

        for edge in edge_list:
            first_vertex_index = matrix_elements.index(edge[0])
            second_vertex_index = matrix_eleme nts.index(edge[1])
            adjacency_matrix[first_vertex_index][second_vertex_index]
= 1

        return adjacency_matrix
```

There is a lot going on above so let me break down what is happening and then we'll look at how to use the class. The graph has two dictionaries. The outgoing dictionary is a dictionary whose keys are the vertices in the graph and whose values are a dictionary of the vertices each connects to whose value are the edges that connect the two vertices. Now that's a mouthful and probably still confusing so let me show the outgoing and incoming dictionaries for the below directed and undirected graphs:





The outgoing dictionary for the undirected graph on the left above is:

```
{
      "A" : {
            "B" : Edge(  "A" ,   "B" ),
            "C" : Edge(  "A" ,   "C" )
      },
      "B" : {
            "A" : Edge(  "A" ,   "B" )
      },
      "C" :   {
            "A" : Edge(  "A" ,   "C" )
      }
}
```

As this is an undirected graph, the incoming dictionary is also the same.

I should note, that if you try to print these out, you'll get a dictionary of pointers to the vertices and edges. I have replaced the pointers with the values of the vertices and Edge(x, y) for the pointers to the edges.

The outgoing dictionary for the directed graph on the left above is:

```
{
      "A" : {
            "B" : Edge(  "A" ,   "B" ),
            "C" : Edge(  "A" ,   "C" )
      },
      "B" : {}
      "C" : {}
}
```

The incoming dictionary for the directed graph is:

```
{
    "A" : {},
    "B" : {
        "A" : Edge(  "A" ,   "B" )
    },
    "C" : {
        "A" : Edge(  "A" ,   "C" )
    }
}
```

I hope this has made what's going on with these dictionaries much clearer.

The methods for checking if the graph is directed and getting the vertex count are straight forward and self-explanatory. The method for inserting a vertex should also be easily understand now that you understand how the incoming and outgoing dictionaries are structured.

The method for retrieving the incident edges for a given vector may cause some confusion so I'll explain how it works. Firstly, it's a generator function which allows us to repeatedly call *next()* on it to retrieve the next edge for a given vertex. We first get the dictionary of adjacent vertices to a given vertex. Then we yield the values which are the edges between the given vertex and each adjacent vertex.

The method for inserting an edge is quite neat as we don't need to worry about whether a graph is directed or undirect as that was handled for us in the constructor for the graph.

Next, we have a method for generating the adjacency list of the matrix. To build the list we are iterating over each of the vertices in the graph and adding the elements of each vertex to a dictionary. The values at each of those keys is a list of elements of each vertex that is on the opposite side of the edge connecting the two vertices.

As we have a method to generate the adjacency list, the method to generate the adjacency matrix is unchanged from earlier!

Let's look at how to use the class. We'll be building the graph from above:

```
graph  =  Graph()
vertex1   =  grap h.insert_vertex(      "A" )
vertex2   =  graph.insert_vertex(       "B" )
vertex3   =  graph.insert_vertex(       "C" )
graph.insert_edge(vertex1, vertex2)
graph.insert_edge(vertex1, vertex3)

print   (graph.vertex_count())
print   (graph.vertices())
print   (graph.is_directed())
print   ( next (graph.incident_edges(vertex1)).destination.element())
print (graph.generate_adjacency_list())
print (graph.generate_adjacency_matrix())
```

Which outputs:

```
# graph.vertex_count()
3

# graph.vertices()
dict_keys([ <__main__.Vertex object at 0x000002446E710340>, <__main__.Vertex object at 0x000002446E70ACA0>, <__main__.Vertex object at 0x000002446E70AC40>])

# graph.is_directed()
False

# next(graph.incident_edges(vertex1)).destination.element()
B

# graph.generate_adjacency_list()
{'A' : [ 'B' , 'C' ], 'B' : [ 'A' ] , 'C' : [ 'A' ]}

# graph.generate_adjacency_matrix()
[[ 0, 1, 1], [ 1, 0, 0], [ 1, 0, 0]]
```

To generate the directed version of the graph from above all we need to do is pass True when creating the instance of the graph:

```
graph = Graph(True)
```

Now when making calls to generating both the adjacency list and adjacency matrix we'll be returned the list and matrix having accounted for the fact the graph is directed!

Graph Traversals

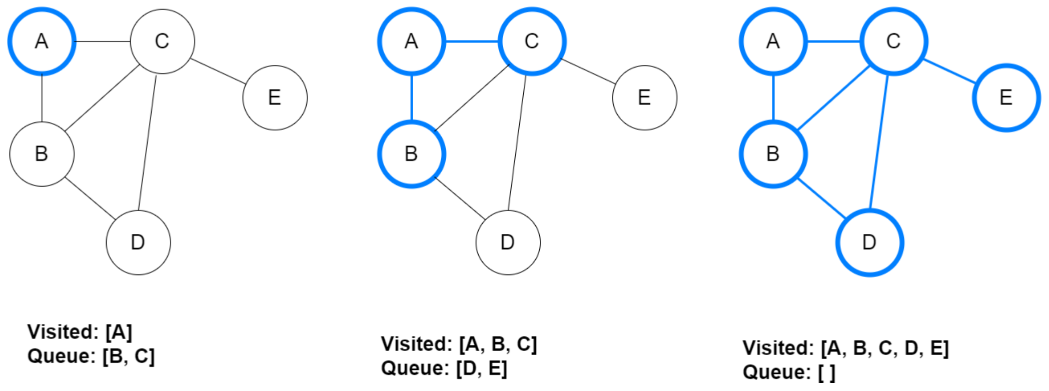
We have written algorithms to traverse many types of linked lists and trees in this book. As with all types of those data structures we have look at, they have a well-defined structure the algorithms for traversing them were relatively straight forward, both to write and understand and were quite intuitive. However, as we've seen with graphs, they don't have a well-defined structure in the sense that any "node" can be connected to any other node, this can end up forming loops as we've seen and in large graphs these loops could appear anywhere in the structure. The ability for loops to form in the graph is just one issue we'll encounter when trying to come up with a traversal algorithm.

Thankfully, there are many traversal algorithms for graphs that have been developed. In this section we'll be taking a look at two of them. A common strategy is to follow a path until a dead end is reached, then walking back up the path we've travelled until we reach a point where we can explore a path that we haven't already taken. This is called back-tracking. We can also iteratively move from one node to another in order to traverse the full graph or part of it. To remember the paths, we've taken and the nodes we've visited along the way, we normally track which vertices have been visited and which have not.

Breadth-first Search

The first traversal algorithm, or more specifically, the first search algorithm we'll look at is called breadth-first search. The algorithm works by picking a vertex in the graph as it's root. It then visits each of it's neighboring vertices, after which it explores the neighbors of those vertices and so on until it has explored all vertices, or we reach a stopping condition (finding the vertex we were searching for as an example). We keep track of the nodes we've visited along the way, so we don't revisit them or enter an infinite loop of revisiting the same nodes.

To help in our understanding let's take a look at the below diagram:



Visited: [A]
Queue: [B, C]

Visited: [A, B, C]
Queue: [D, E]

Visited: [A, B, C, D, E]
Queue: [ ]

We start by picking a vertex, A in this case and add it to the visited list. We then add the neighbors of A to the queue of nodes to visit, which are B and C here. Next, we remove the first vertex from the queue which is B, and we add that to the visited list. We then add B's neighbors to the queue. We should add the vertex A to the queue, but it's already been visited so we don't add it. Similarly, we should also add C to the queue but it's already in the queue, so we skip it, so we just add the vertex D. Next, we remove C from the queue and add it to the visited list. We then add the vertices that are neighbors of C that haven't already been visited or in the queue to be visited, which is just E in this case. Now our queue contains the vertices D and E, so we repeat this process. We remove D from the queue and add it to the visited list. As D has no neighbors that haven't been visited or are in the queue to be visited there is nothing further, we can do so we take the next vertex from the queue which is E. As there are no neighbors of E that we haven't visited or are in the queue to be visited we don't do anything. The queue is now empty, so we stop. When the queue becomes empty, we know we have traversed the graph.

In the case of searching for a specific vertex in the graph, we check the value of the vertex before we add it to the visited queue. If it is the vertex, we are searching for then we can simply stop.

To implement this algorithm, we can use the Queue you developed earlier on in the book, but for simplicity I will be using a list and using *pop(0)* to remove the vertex from the front of the list.

```
def breadth_first_search(graph, s):
    # Add starting node to queue
    queue = [s]
    visited = []
    while len(queue) > 0:
        # Take next vertex from queue
        next_vertex = queue.pop(0)
        # Add the vertex to visited list
        visited.append(next_vertex)
        # Get each edge
        for e in graph.incident_edges(next_vertex):
            # Get the vertex at the opposite end of edge
            v = e.opposite(next_vertex)
            # If it's not in visited list or the queue then add to que
ue
            if v not in visited and v not in queue:
                queue.append(v)
        print(visted, queue)
```

The above algorithm works by passing a graph and a starting vertex, s, in the graph. We then start our traversal from that point. The nice thing about the two traversal algorithms we'll look at, is that we can start the traversal from anywhere in our graph.

Although the algorithm works, it's not very useful in terms of searching for a vertex so let's modify the above algorithm to add the capability to stop when we find a vertex we're looking for. In this case, we'll be looking for the element of the vertex.

```
def breadth_first_search(graph, s, element_to_find):
    queue = [s]
    visited = []
    while len(queue) > 0:
        next_vertex = queue.pop(0)
        visited.append(next_vertex)

        # We've found the element, so return it
        if next_vertex.element() == element_to_find:
            return (next_vertex, len(visited))

        for e in graph.incident_edges(next_vertex):
            v = e.opposite(next_vertex)
            if v not in visited and v not in queue:
                queue.append(v)
    return (None, len(visited))
```

In the above modified algorithm, we're searching for an element of a vertex. If we find it, we return the vertex and the length of the visited list, otherwise we return None and the length of the visited list.

Let's look at how we can use this algorithm. Firstly, we should reconstruct the graph from the above diagrams:

```
g = Graph()
v1 = g.insert_vertex("A")
v2 = g.insert_vertex("B")
v3 = g.insert_vertex("C")
v4 = g.insert_vertex("D")
v5 = g.insert_vertex("E")

g.insert_edge(v1, v2)
g.insert_edge(v1, v3)
g.insert_edge(v2, v3)
g.insert_edge(v2, v4)
g.insert_edge(v3, v5)
g.insert_edge(v4, v3)
```

Now that we have our graph, let's search for the vertex whose element is "D" and we'll start from vertex "A".

```
(vertex, num_visited) = breadth_first_search(g, v1, "D")
if vertex is None:
    print(f'Could not find element after search {num_visited} vertices')
else:
    print(f'Found {vertex.element()} after searching {num_visited} vertices')
```

The above code will output:

```
Found D after searching 4 vertices.
```

Now, let us search for an element, "L". There are no vertices in the graph with that element so we should see the message from the else block:

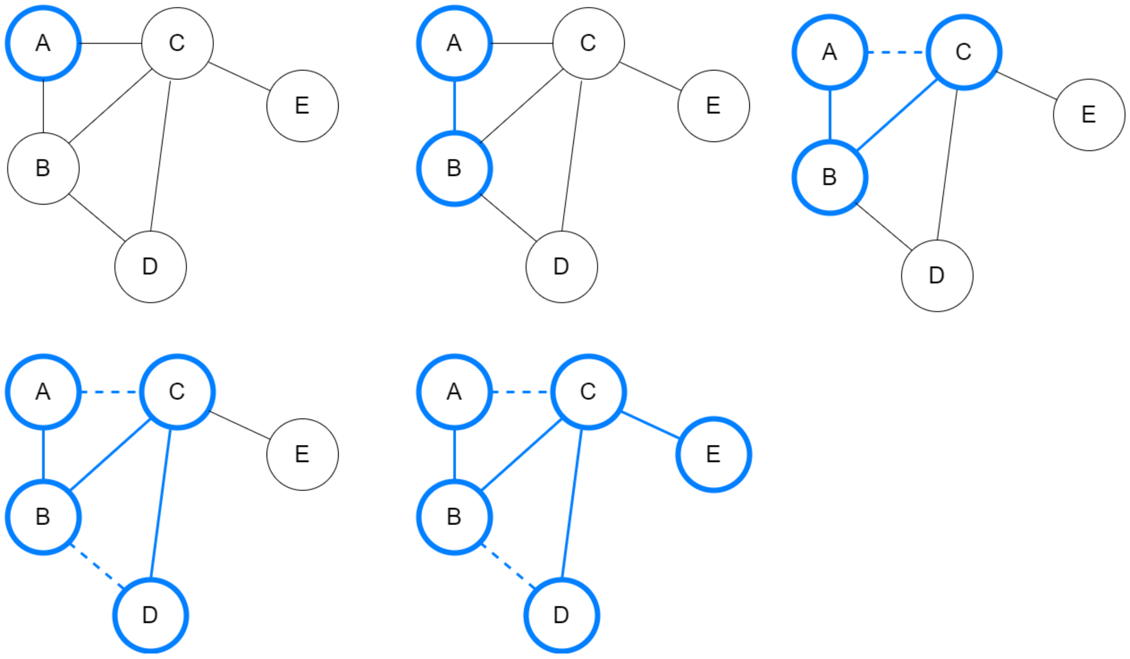
```
Could not find element after search 5 vertices.
```

Which is as we expect!

An interesting thing about the nature of breadth first search is that in un-weighted graph where all paths are equal, it will find the shortest path between any two nodes.

Depth-first Search

The next traversal algorithm we're going to look at is depth-first search. The way depth-first search works is, instead of visiting the vertices in the order we discovered them, we visited the first vertex we find, then we visit the first neighboring vertex of that vertex, and so on until we reach a dead end. When we reach a dead end we back-track until we can explore a path we haven't followed yet. This is illustrated below:



In the above illustration, we start from vertex A. We add vertex A to the visited list then we add the first neighboring vertex to the stack which is B. We then visit by and add it to the visited list. We then add the first neighboring vertex to the stack. This would be A, but we've already visited it so we skip it and add C instead. We pop from the stack which gives C so we visit C and add the first vertex to the stack which is neighboring it. This would be A but it's already been visited, so we move to B which has already been visited so we skip it too and add D instead. We then add D to the visited list and choose the next unvisited vertex for which there are none so we must backtrack to search for a path we can follow so we move back C and look for next unvisited neighboring vertex which is E so add it to the stack and visit it. We add E to the visited list and add the next unvisited node to the stack for which there are none so we move back C and add the next unvisited vertex to the stack for there are none, so we move back to B and add the next unvisited vertex to the stack for which there are none, so we move back to A and add the next unvisited vertex to the stack for which there are none. We can't backtrack any further and the stack is empty, so we are done.

We can implement this the exact same way as we did with breadth-first search, but we swap a queue for a stack.

Let's take a look at the implementation of depth-first search:

```
def depth_first_search(graph, s, element_to_find):
    stack = [s]
    visited = []
    while len(stack) > 0:
        next_vertex = stack.pop()
        visited.append(next_vertex)

        # We've found the element, so return it
        if next_vertex.element() == element_to_find:
            return (next_vertex, len(visited))

        for e in graph.incident_edges(next_vertex):
            v = e.opposite(next_vertex)
            if v not in visited and v not in stack:
                stack.append(v)
    return (None, len(visited))
```

Notice the only change we needed to make: We swapped renamed *queue* to *stack* and replaced *pop(0)* with *pop()* (which pops from the end).

When you use depth first search and search for the vertex "B" for example, you would expect to find this after visiting 2 nodes (A and B), however in my case, the algorithm visited B last. Why is this? In the above diagrams we visited the nodes in alphabetical order but remember, our graph is a dictionary which means we don't have any order as our hashes can change. When I ran this algorithm, A was visited first, then C, then E, then it back-tracked to C and visited D, then back-tracked to C and visited B. However, it still followed a depth first approach, that is, follow a path until a dead end is reached, then back-track to take another path until all possible paths are exhausted.

```
(vertex, num_visited) = depth_first_search(g, v1, "B")
if vertex is None:
    print(f'Could not find element after search {num_visited} vertices')
else:
    print(f'Found {vertex.element()} after searching {num_visited} vertices')
```

Which output:

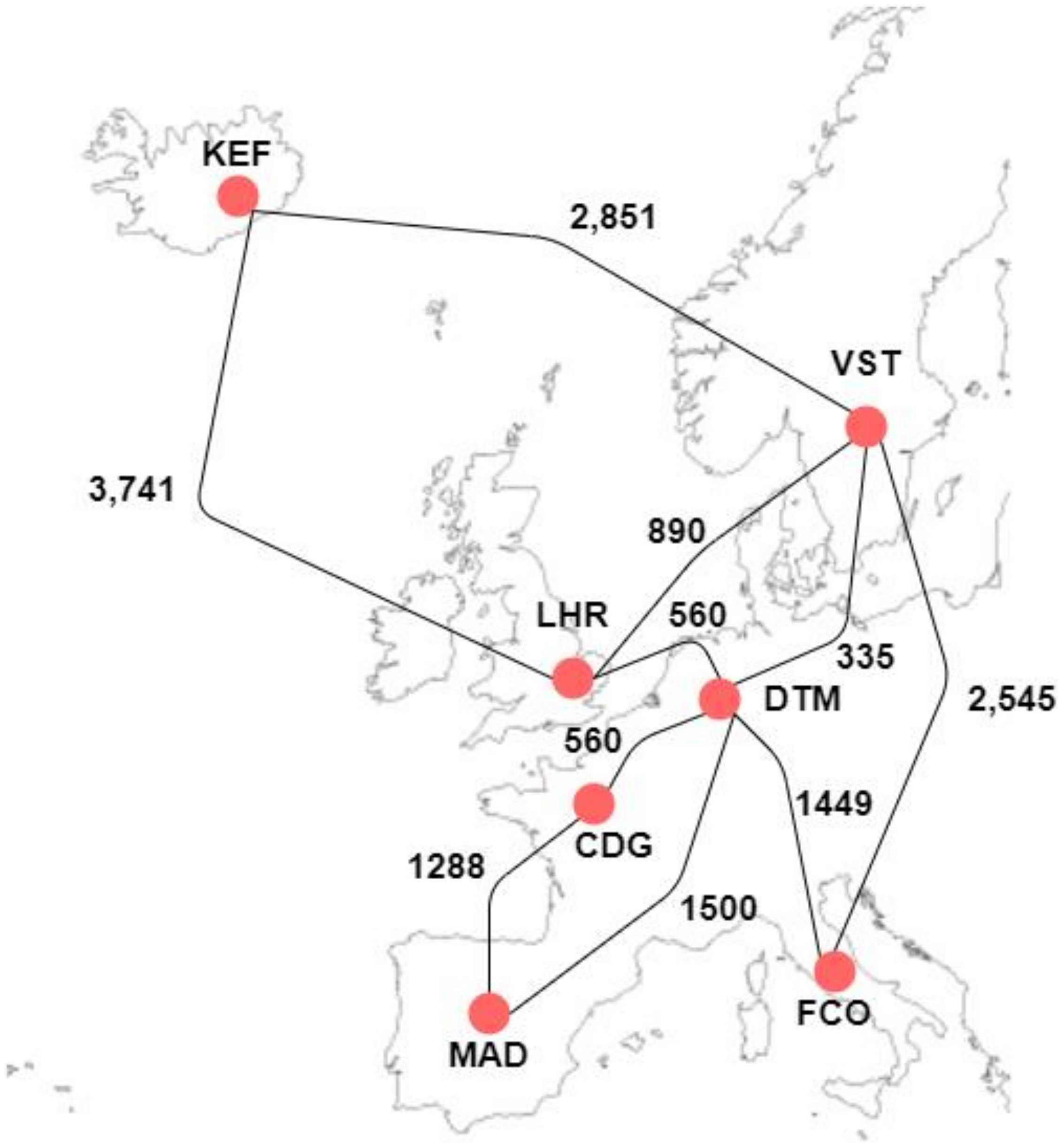
```
Found B after searching 5 vertices
```

Both of these traversal algorithms work with both directed and undirected graphs. Also, both our traversal algorithms are fit to work on our Graph class. However, the principles from both can be applied to any form of a graph so we can fit the algorithms to work on adjacency lists and adjacency matrices. As Trees are a special case of graphs, we can apply these algorithms to trees too, however as trees tend to have a better defined structure, we have better search algorithms for them.

Shortest Path

We discussed in the previous section how breadth-first search can be used to find the shortest path between any two nodes in a graph where all edges are considered equal, however, this strategy doesn't work in many situations. Consider a weighted graph where the vertices represent cities, and the weighted edges represent the distances between the cities. Likewise, we can use a graph to represent a computer network (such as the Internet), and we might be interested in finding the fastest way to route a data packet between two computers. In this case, it might not be appropriate for all edges to be equal. For example, one edge might represent a low-bandwidth connection while another might represent a high-speed fiber-optic connection.

Consider the illustration below of a partial map of Europe with some major airports and distances between them:



For the purposes of this section, it would be better to give a mathematical definition of the shortest path in a weighted graph. Let G be a weighted graph. The length (or weight) of the path is the sum of the weights of the edges of the path P. That is, if P = ((V0, V1), (V1, V2), ..., (Vn-1, Vn)), then the length of P, denoted w(P), is defined as:

ح٩ه

Ú Ũ هجسي Ú Ùׅ ȟÙׅ٨ه

ﬁﻠﻳ

The distance from any two vertices, is the length of a minimum-length path (also called the shortest path) from both vertices, if such a path exists.

Dijkstra's Algorithm

Before we look at Dijkstra's algorithm, we need to gain an understanding of another data structure. We won't be looking at this data structure in detail, instead we'll be using the built in version that comes with Python.

This new algorithm is called a Heap Queue, also known as the priority queue algorithm. A heap is a binary tree for which every parent node has a value less than or equal to any of its children.

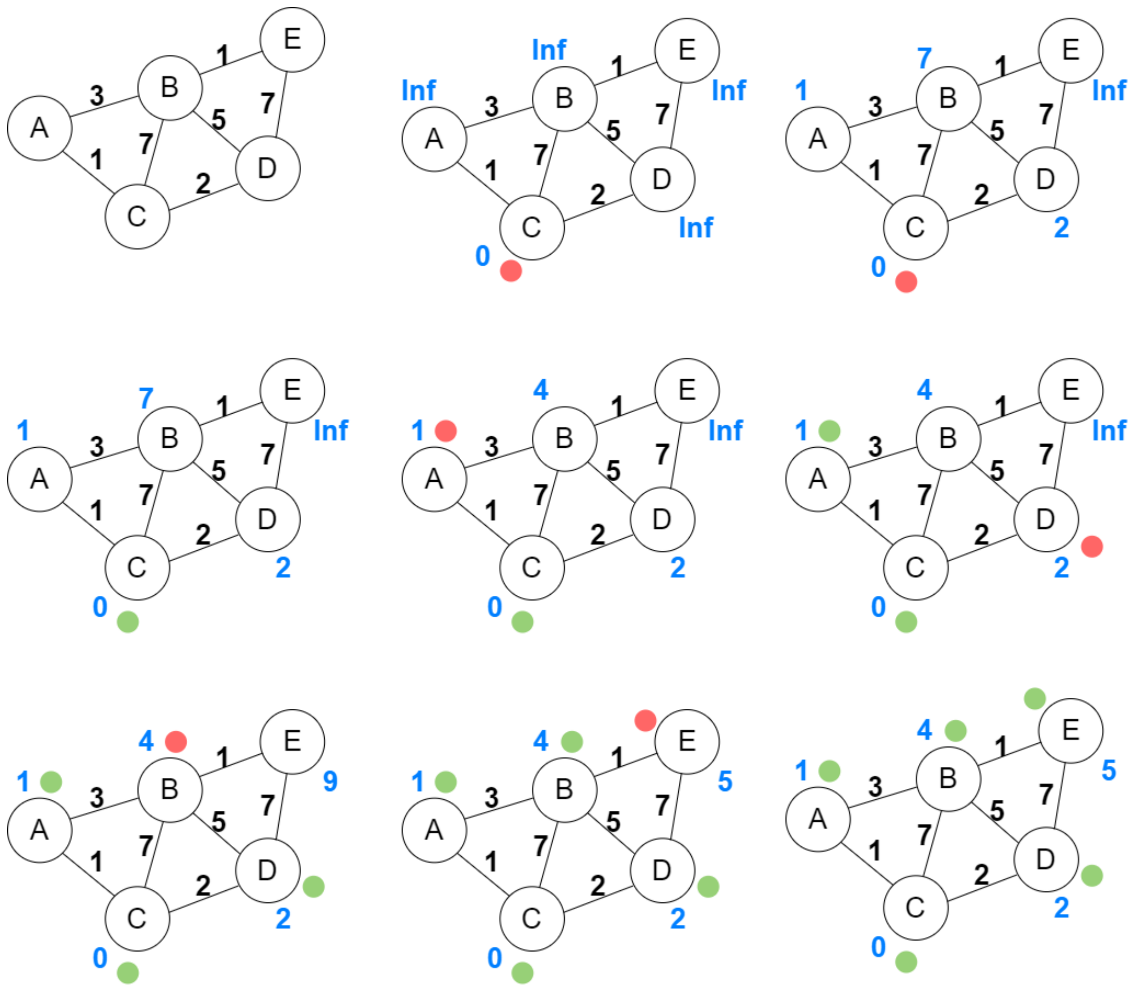
A priority queue is a data structure similar to a queue in which every element in the queue also has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with a lower priority. We can use the Python built-in *heapq* to provide a priority queue for us.

Dijkstra's Algorithm allows you to calculate the shortest path between one node and every other node in the graph. The algorithm works on weighted graphs.

Below is a description of the algorithm:

1. Mark the selected initial vertex with a current distance of 0 and the rest of the vertices have a distance of infinity.
2. Set the non-visited node with the smallest current distance as the current vertex, C
3. For each neighbor, N, of the current vertex, C:
    1. Add the current distance of C with the weight of the edge connecting C to N.
    2. If the sum of the current distance and the weight is less than or equal to the current distance of N, set it as the new current distance of N.
4. Mark the current node C as visited.
5. If there are non-visited nodes, go to step 2.

That may seem a little abstract so let's take a look at an example. There are 12 diagrams below starting from top left, moving from left to right, row by row, I'll explain what happens at each of the 12 steps.



Below is a step by step of what is happening in each diagram. Our initial vertex is C:

1. We start off in our initial state - a weighted graph.

2. We mark every vertex with it's minimum distance to vertex C (the initial vertex). For vertex C, this distance is 0. For the rest of the vertices, as we still don't know the minimum distance, it starts as infinity (Inf). The red dot indicates that this is the current vertex we're looking at.

3. Next, we check the neighbors of the current vertex (A, B, and D) in no specific order. Let's begin with B. We add the minimum distance of the current vertex (in this case, 0) with the weight of the edge that connects our current vertex with B (in this case, 7), and we obtain 0 + 7 = 7. We compare that value with the minimum distance of B (infinity). The minimum value of the two is the one that remains the minimum distance of B. (In this case, 7 is less than infinity). We repeat that process for vertices A and D. Once we have, the minimum distances for A and D are 1 and 2 respectively.

4. Now, as we have checked all the neighboring vertices of C and calculated the minimum distances for them, we can mark C as complete - this is indicated by a green circle beside the vertex.

5. We now need to pick a new current vertex. We select this from the unvisited vertices with the smallest minimum distance which is A in this case. I have now marked that vertex with a red dot to indicate it is the new current vertex. Now we repeat the algorithm. We check the neighbors of the current vertex, ignoring the visited vertices (A in this case). This means we only check B. For B, we add 1 (the minimum distance of A, our current vertex) with 3 (the weight of the edge connecting A and B) to obtain 4. We compare 4 with the minimum distance of B (which is currently 7). As 4 is less than 7, 4 becomes the new minimum distance for B.

6. We mark this the vertex A as complete and pick a new current vertex. Again, to do this we select the un-visited vertex with the smallest minimum distance which is D. We repeat the algorithm again. This time we check B, and E. For B, we obtain 2 + 5 = 7. We compare the value of B's minimum distance with 7. As 4 is less than 7 we leave the minimum distance for B as it is. For E, we obtain 2 + 9, compare it with the minimum distance of E (infinity) and set the minimum distance for E as the smallest of the two, which is 9.

7. We mark the vertex D as complete and pick a new current vertex. Again, we select the vertex from the un-visited vertices with the smallest minimum distance which is B. Now we only need to check E. We obtain 4 + 1 = 5 as a new distance to E, which is less than E's minimum distance of 9, so the minimum distance for E becomes 5. We mark B as complete and select the next vertex from the un-visited vertices with the smallest minimum distance which is E and that becomes our new current vertex.

8. As there are no un-visited neighbors to E, we can mark it as visited.

9. Now we are done as there are no more un-visited vertices. If there were we'd continue on doing what we've been doing. Now the minimum distance of each vertex represents the minimum distance from C to that vertex.

As you may have guessed, we'll be using the priority queue (*heapq*) to keep track of what vertices to visit next. The vertex with the smallest minimum distance will be at the top of the queue and the vertex with the largest minimum distance will be at the bottom.

Below is the code for Dijkstra's algorithm. I have commented the code and I'll give an explanation of what's happening with the heap below the code but read through it line by line and try gain an understanding of what's happening:

```
import    heapq

def   dijkstras(graph, src):
      distances    =  {}
      visited     =  []
```

```
    # Set initial distances for all vertices to infinity
    vertices     = g.vertices()
    for   v  in   vertice   s:
        distances[v]      = float   ( 'inf'    )

    # Set distance for src (first vertex) to 0
    distances[src]       = 0

    # create our unvisited list
    unvisited     = [(distances[v],         id (v), v)      for   v  in   vertices]
    # Convert the list to a heap
    heapq.heapify(unvisited)

    while   len (unvisited):
        # Get vertex tuple with smallest distance (highest priority)
        heap_v  =  heapq.heappop(unvis   ited)
        # Get the vertex object from the tuple
        current    = heap_v[ 2]
        # Append the vertex to the visited list
        visited.append(current)

        # For all neighbouring vertices
        for   e  in   graph.incident_edges(current):
            v  =  e.opposite(current)
            # Skip if it has been visited
            if   v  in   vis ited:
                continue

            # Calculate the new minimum distance
            new_dist   = e.element    +  distances[current]

            # Update the distance for the vertex if it's less than cur
rent minimum distance
            if   new_dist   <  di stances[v]:
                distances[v]      = new_dist

        # Destroy the heap
        while   len (unvisited):
            heapq.heappop(unvisited)

        # Rebuild the heap but only include vertices that are unvisite
d
        unvisited     = [(distances[v],         id (v), v)      for   v  in   vertices     if   v  n
ot  in   visited]
        heapq.heapify(unvisited)
    return distances
```

There is a lot going on here, but I think you should understand just about all of it, except for stuff related to the heap, so I'll explain what's going on with those pieces of code.

```
# create our unvisited list
unvisited = [(distances[v], id (v), v) for v in vertices]
# Convert the list to a heap
heapq.heapi fy(unvisited)
```

The *heapq* library provides a *heapify()* function. This will create a heap (which is just a list but is ordered according to the heap property) from the *unvisited* list in-place.

The *unvisited* list is a list of tuples. The reason this must be a list of tuples is because we want to keep pointers to the vertex objects in there along with the distance information. However, we have also included the *id()* of the vertices. We need to do this because the *heapify* function needs to compare the tuples in order to sort them correctly. The *heapify()* function will compare tuples using *<=*.

A problem arises when comparing tuples however, as, when the first items of the tuples are not unique, Python will compare the second items in the tuples. If our tuples were only *(distances[v], v)* and if comparing two tuples and the distances were the same, *heapify()* would look at the second items in the tuples it's comparing. As we didn't override the *__lte__()* method in the Vertex class, the function would fail to compare the two vertex objects and we would get an error. To stop this from happening I've included the unique *id(v)*, of each vertex. This way, if the distances are equal, we can distinguish the tuple pairs by the IDs of the vertices they represent. We have also included the pointers to the vertices in the tuple because, when we want to fetch the next item from the priority queue (the heap) we can also get the vertex.

```
# Get vertex tuple with smallest distance (highest priority)
heap_v = heapq.heappop(unvisited)
```

This line pops the highest priority item (the vertex with the smallest minimum distance) from the heap.

```
# Destroy the heap
while len (unvisited):
    heapq.heappop(unvisited)

# Rebuild the heap but only include vertices that are unvisited
unvisited = [(distan ces[v], id (v), v) for v in vertices if v not in vi
sited]
heapq.heapify(unvisited)
```

Now that we have updated the minimum distances for all neighbors of the current vertex and marked the vertex as complete, we need to rebuild the heap as each of the tuples in it contain outdated distance information.

To demonstrate the function, I have recreated the graph from the walkthrough example. This time, we've added weights to the edges:

```
graph = Graph()
a = graph.insert_vertex("A")
b = graph.insert_vertex("B")
c = gra ph.insert_vertex("C")
d = graph.insert_vertex("D")
e = graph.insert_vertex("E")

graph.insert_edge(a, b, 3)
graph.insert_edge(a, c, 1)
graph.insert_edge(b, c, 7)
graph.insert_edge(b, d, 5)
graph.insert_edge(b, e, 1)
graph.insert_edge(c, d, 2)
graph.insert_ edge(d, e, 7)

distances = dijkstras(graph, c)

for k, v in distances.items():
    print (k.element(), v)
```

We will get the following output:

```
A 1
B 4
C 0
D 2
E 5
```

Which are the distances we ended up with at the end of our walkthrough example.

You may be thinking that it's great that we can find the shortest distances between the a given vertex and all others in a graph, however we don't know the paths that were taken. We can solve this by keeping a dictionary of paths. The dictionaries structure will be the exact same as the adjacency list matrix but they keys are the vertices, and the values are a list of vertices - the shortest path.

Let's look at the revised Dijkstra's algorithm:

```
import heapq

def dijkstras(graph, src):
    distances    = {}
    paths   = {}
    visited     = []

    # Set initial distances for all vertices to infinity
    vertices     = g.vertices()
    for   v  in   vertices:
        distances[v]      = float   ( 'inf'    )

    # Set initial paths for all vertices to None
    vertices     = g.vertices()
    for   v  in   vertices   :
        paths[v]     = None

    # Set distance for src (first vertex) to 0
    distances[src]      = 0

    # Set path for src to itself
    paths[src]     = [src]

    # create our unvisited list
    unvisited     = [(distances[v],         id (v), v)      for   v  in   vertices]
    # Convert the list to a heap
    heapq.heapify(unvisited)
    while    len (unvisited):
        # Get vertex tuple with smallest distance (highest priority)
        heap_v  = heapq.heappop(unvisited)
        # Get the vertex object from the tuple
        current    = heap_v[ 2]
        # Append the vertex to the visited list
        visited.append(current)

        # For all neighbouring vertices
        for   e  in   graph.incident_edges(current):
            v  = e.opposite(current)
            # Skip if it has been visited
            if   v  in   visited:
                continue

            # Calculate the new minimum distance
            new_dist    = e.element    + distances[current]

            # Update the distance for the vertex if it's less than cur
rent minimum distance
            if   new_dist   < distances[v]:
                distances[v]      = new_dist
```

```
                    paths[v]  = paths[ current]  + [v]

        # Destroy the heap
        while  len (unvisited):
            heapq.heappop(unvisited)

        # Rebuild the heap but only include vertices that are unvisite
d
        unvisited  = [(distances[v],  id (v), v)  for  v in  vertices  if  v n
ot in  visited]
        heapq.heapify(unvisited)

    return  distances, paths
```

The change here to also return the shortest paths is quite simple. First, we initialize a paths dictionary the same way we did with the distances dictionary, except we set the initial values to *None*:

```
# Set initial paths for all vertices to None
vertices  = g.vertices()
for  v  in  vertices:
     paths[v]  = None
```

Next, we set the path for the source vertex as itself. The paths will be list of vertices taken to arrive at that vertex.

```
# Set path for src to itself
paths[src]  = [src]
```

And finally, when we are updating the distance, this is when we update the paths. To update the path to the latest shortest path (as it can change when we progress through the algorithm) we set the path for the vertex we're updating the distance for as the path taken for the current vertex plus the vertex itself.

From the walkthrough diagrams, in the case of the vertex B, it's initial path (from fig 3 of the walkthrough) would be C -> B. However, when we visit A, we find out that going to A first, then B is actually faster than going straight to B from C so we updated the distance for B to 4. We also update the path at this point as the shortest path is no longer C -> B it's the path to the current vertex (A) which is C -> A, plus B tagged onto the end of the list giving us the final path of C -> A -> B. This path is final because it is never updated again in the algorithm, but it could have been if we discovered a shorter path down the line.

I am also returning the paths in the revised function.

We can now use our function to show the shortest paths as follows:

```
graph  = Graph()
a  = graph.insert_vertex(  "A" )
```

```
b = graph.insert_vertex( "B" )
c = graph.insert_vertex( "C" )
d = graph.insert_vertex( "D" )
e = graph.insert_vertex( "E" )

graph.insert_edge(a, b, 3)
graph.insert_edge(a, c, 1)
graph.insert_edge(b, c, 7)
graph.insert_edge(b, d, 5)
graph.insert_edge(b, e, 1)
graph.in sert_edge(c, d, 2)
graph.insert_edge(d, e, 7)

distances, paths = dijkstras(graph, c)

for k, path in paths.items():
    print (f'Path for {k. element() }' )
    print (" -> " .join([p.element() for p in path]))
```

Which will give us the following outputs:

```
Path for A
C -> A

Path for B
C -> A -> B

Path for C
C

Path for D
C -> D

Path for E
C -> A -> B -> E
```

You can verify these against the walkthrough examples.

Dijkstra's algorithm has a number of real-world applications and really important ones too. It's used in network routing protocols. It's used in "Intermediate System to Intermediate System" (IS-IS) which is a routing protocol designed to move information efficiently within a computer network, a group of physically connected computers or similar devices. It accomplishes this by determining the best route for data through a packet switching network. It's also used in Open Shortest Path First (OSPF) which is a routing protocol for Internet Protocol (IP) networks (which plays a big role in linking all the networks that make up the internet).

Least-cost paths are calculated for instance to establish tracks of electricity lines or oil pipelines. Dijkstra's algorithm can be used to solve these problems. The algorithm has also been used to calculate optimal long-distance footpaths in Ethiopia and contrast them with the situation on the ground.

Data Processing

There is an abundance of multimedia information stored on computers across the world which new data being generated at an ever rapidly increasing pace. One of the most challenging types of multimedia to process is plain old textual data. Books, snapshots of the web in the form of HTML and XML, email archives, feeds from social networks such as Facebook or twitter, and customer product reviews are just some of sources of the vast wealth of data that exists on the internet. That list excludes the documents stored on everyone's computers that aren't part of the accessible internet.

In 2019, PwC, one of the world's largest professional services in tax, advisory and audit, estimated there 4.4 zettabytes in our digital universe. That's an increase of 1.7 zettabytes from a 2017 estimate. IDC predicts the world's data will grow to 175 zettabytes by 2025. Just for reference, 1 zettabyte is the same as 1,099,511,627,776 gigabytes.

With all of this information out there we must be able to sort and process it efficiently in order to make use of it. With numerical data we process it relatively easily as computers are essentially calculators but processing textual data is another challenge. In this section we'll be looking at two things. The first is pattern matching for textual data. The second is data compression.

Pattern Matching Algorithms

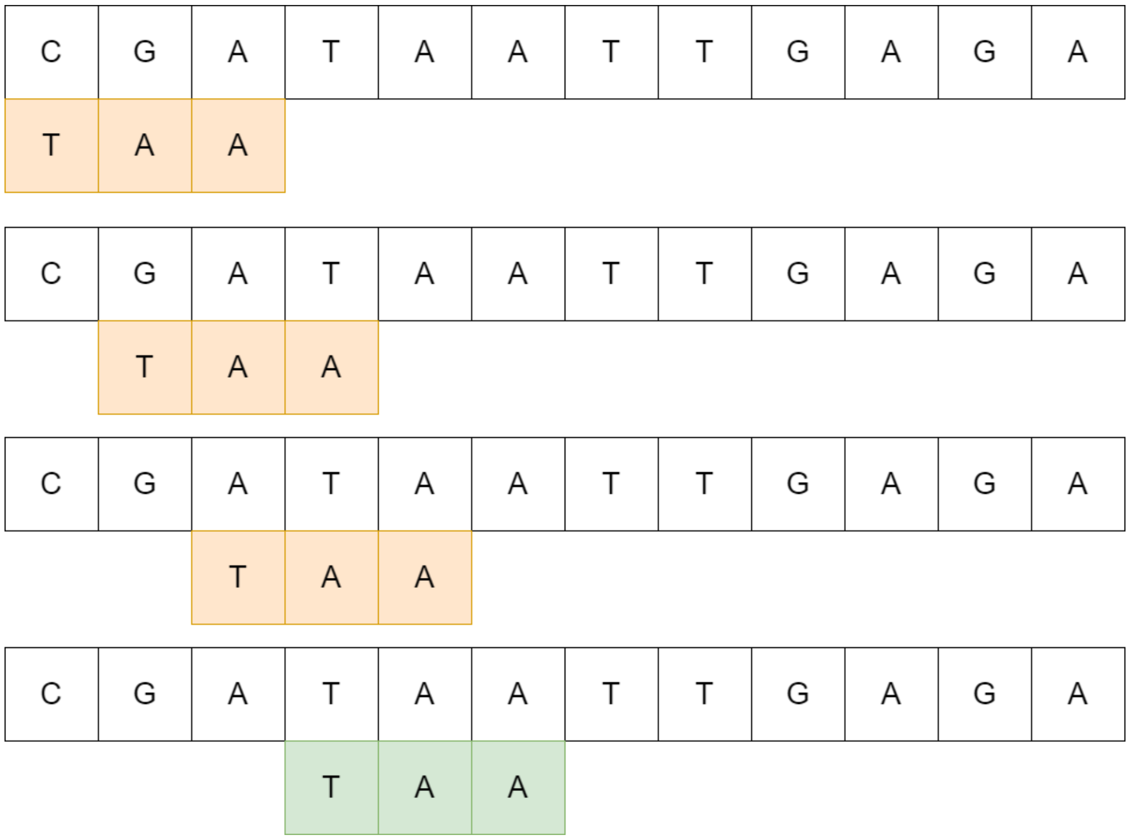
Pattern matching is a very important topic in computer science. It has been used in various applications such as information retrieval in search engines, virus scanning, DNA sequence analysis (particularly important at the time of writing), data mining, machine learning, network security, and pattern recognition just to name a few.

We will be looking at two approaches to pattern matching in text. The first is one we have looked at in passing called the Brute-Force method, the second is the Knuth-Morris-Pratt algorithm.

Brute-Force Pattern Matching

Brute-force is an algorithmic design pattern rather than a specific algorithm itself. We will be applying the design technique to the case of pattern matching in text. When applying the technique in a general situation, we typically enumerate all possible configurations of the inputs involved and pick the best of all the enumerated configurations.

To illustrate this, let's take a look at a DNA sequence in which we want to find an occurrence of a contiguous DNA sub-sequence.



In the above diagram, we are searching for the first occurrence of the subsequence "TAA" in the sequence "CGATAATTGAGA". We brute force search this by "sliding" the pattern (TAA) across the sequence until we find a match. If you remember back to the search algorithms chapter, we modified the linear search algorithm to be a brute force implementation for pattern matching in text.

We can look at another implementation of the brute force algorithm below. It works by taking two strings, T and P. T is a string and P is the pattern. The brute force algorithm works by returning the lowest index of T at which the substring P begins or -1 if T does not contain P.

```
def brute_force_pm(T, P):
    n, m = len (T), len (P)
    for i in range (n - m+1):
        k = 0
        while k < m and T[i +k] == P[k]:
            k += 1
        if k == m:
            return i
    return -1
```

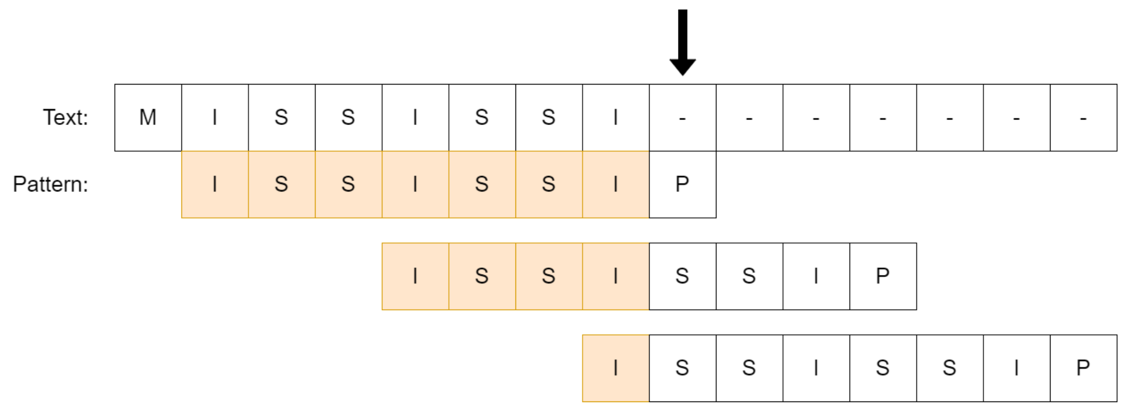
The runtime complexity of the brute-force method is O(nm). That is, the length of the input string by the length of the pattern string. This comes from the fact we have a while loop (which we go around m times) inside a for loop (which we go around n times). We can do much better than this.

The Knuth-Morris-Pratt Algorithm

There is a major inefficiency in the brute force algorithm. While performing the matching of characters between the string and the pattern, we may find several matching characters then a mismatch. In the brute force algorithm, we ignore the information gained by the successful match between possibly multiple characters when we restart with the next incremental placement of the pattern.

The Knuth-Morris-Pratt algorithm takes advantage of the information gained and achieves a runtime complexity of O(n + m) in doing so. The idea behind the KMP algorithm is to precompute self-overlaps between portions of the pattern so that when a mismatch occurs, we know the maximum amount to shift the pattern by before continuing the search (In brute-force we shifted the pattern by 1 each time).

An example of the KMP algorithm is illustrated below:



If a mismatch occurs at the position indicated by the arrow, the pattern can be shifted to the second alignment, without needing to recheck the partial match with the prefix "issi". If the mismatched character is not a "S", then the next potential alignment of the pattern can take advantage of the common "I".

To implement the Knuth-Morris-Pratt algorithm we first need to find out how much we can shift the pattern by for mismatching on any given character in the pattern. This is called the failure function. More specifically, the failure function f(k) is defined as the length of the longest prefix of P that is a suffix of P[1:k+1]. In less complicated terms, if we find a mismatch on character P[k + 1], the function f(k) tells us how many of the immediately preceding characters can be reused to restart the pattern.

The failure function for the pattern used in the example "ISSISSIP" is below:

k	0	1	2	3	4	5	6	7
P[k]	I	S	S	I	S	S	I	P
f(k)	0	0	0	1	2	3	4	0

It may seem natural to code the failure function first as the KMP algorithm relies on it, but I want to show the code for the KMP algorithm first. You'll see later why this is a better approach.

The algorithm works by comparing the characters at index j in a string T and the character at index k in our pattern string P. If the characters match, the algorithm moves on to the next characters in both T and P or returns that there was a match if it's the end of the pattern. If the comparison failed, the algorithm checks the output of the failure function (which is just a list) for a new candidate character in P (in other words the shift) or starts over with the next index in T if failing the first character of the pattern (since nothing else can be reused).

```
def  knuth_morris_pratt(T, P):
     n, m  = len (T),   len (P)

     # Handle empty string
     if   m == 0:
          return   0

     fail   = kmp_failure(P)

     j  = 0
     k  = 0
     while   j  < n:
          # Characters match
          if   T[j]   == P[k]:
               # We have a match
               if   k  == m -  1:
                    return   j  -  m + 1
               # Not the end of pattern but both characters matched
```

```
            j += 1
            k += 1
        # Mismatch, reuse suffix P[0:k]
        elif k > 0:
            k = fail[k - 1]
        # No match and nothing in pattern to reuse so go to next char
in T and restart
        else:
            j += 1
    # No match
    return -1
```

I've commented the above code to explain what is happening at each point and why we enter specific blocks. I encourage you to work through the example on a piece of paper to gain a fuller understanding.

Now we need to compute the failure function. The reason I left this until after is because the failure function is computed by carrying out roughly the same process of the KMP function but by comparing the pattern to itself with different starting indexes.

Below is the code for computing the failure function required by the KMP algorithm:

```
def kmp_failure(P):
    m = len(P)

    # Array the same length as P
    fail = [0] * m

    j = 1
    k = 0
    while j < m:
        if P[j] == P[k]:
            fail[j] = k + 1
            j += 1
            k += 1
        elif k > 0:
            k = fail[k - 1]
        else:
            j += 1
    return fail
```

The runtime complexity of O(n + m) for the KMP algorithm comes from the runtime of the main function which is O(n) and the runtime of the failure function which has a runtime complexity of O(m), hence O(n + m).

Data Compression

Data compression is the process of encoding, restructuring or otherwise modifying data in order to reduce it's size. It essentially involves re-encoding information using fewer bits than the original representation.

The main advantages of data compression are reductions in space taken up on storage hardware, data transmission time, and communication bandwidth. In terms of the internet, it's extremely important - the more we can compress data, the more data we can send between systems and the faster that data will be transmitted. This can result in significant cost savings.

Compressed files require significantly less storage capacity than uncompressed files, meaning a significant decrease in expenses for storage. A compressed file also requires less time for transfer while consuming less network bandwidth. This can also help with costs, and increase productivity.

Consider a text file that is 10 gigabytes in size that we need to send to 5,000 people and it will take 6 hours to send everyone that file. If we were able to compress this file by 50%, we would be able to send this file to all 5,000 people in roughly half the time (3 hours) as we would be able to send two compressed files over the network for every 1 uncompressed file. This is saving significant time and money! In this section we'll be looking a text compression.

Huffman Coding

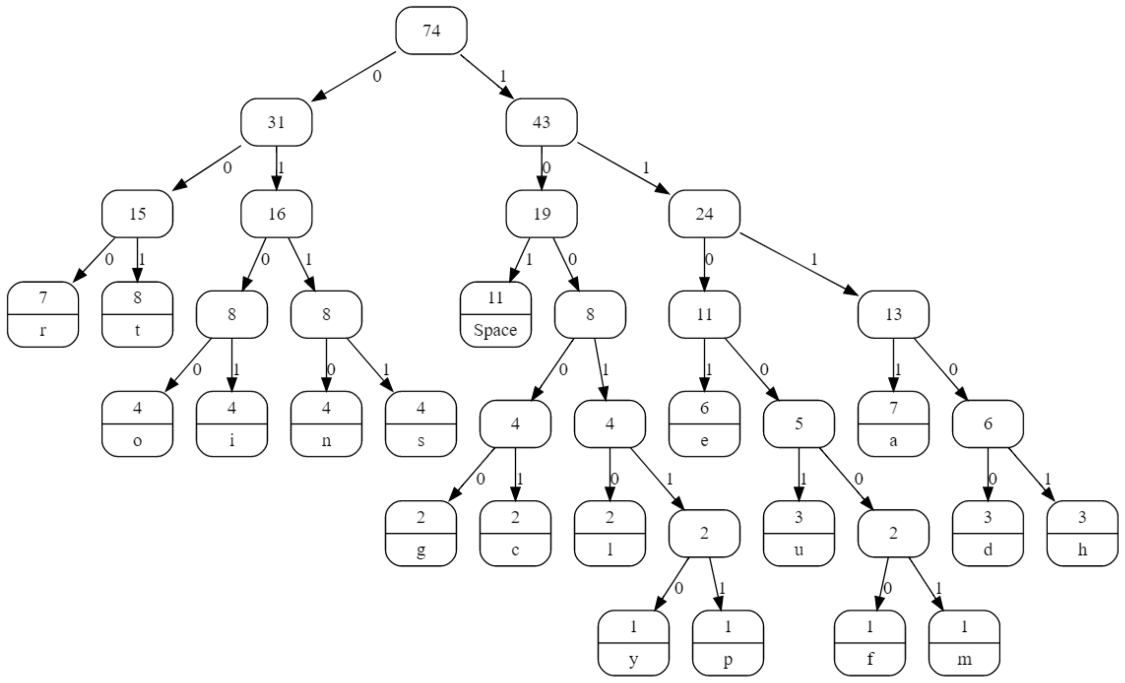
The Huffman coding algorithm is one of the most widely used data compression algorithms. It is the algorithm used in all the familiar compression formats such as GZIP, WinZip, JPEG, PNG, and MPEG-2 just to name a few. I'd like to point out that these formats may not solely use Huffman's algorithm and may be combination of multiple compression algorithms. For example, GZIP uses a combination of the LZ77 compression algorithm and Huffman coding compress files.

Standard encoding schemes such as ASCII, use fixed-length binary strings (strings of 0's and 1's) to encode characters. For example, in the ASCII encoding scheme, the character "a" is the binary string "01100001".

Huffman code compresses data over a fixed-length encoding by using short code-word strings to encode high-frequency characters and long code-word strings to encode low-frequency characters. Huffman code uses a variable-length encoding specifically optimized for a given string X over any alphabet. The optimization is based on the use of character frequencies, where we have for each character in some string X, a count function that counts the number of times each character appears in that string.

Huffman's algorithm for producing the optimal variable-length prefix code for a string X is based on the construction of a binary tree that represents the code. Each edge in T represents a bit in a code-word, with an edge to a left child representing a "0" and an edge to a right child representing a "1". Each leaf of the tree is associated with a specific character, and the code-word for that character is defined by the sequence of bits associated with the edges in the path from the root of the tree to the leaf. Each leaf has a frequency which is simply the frequency in X of the character associated with the leak. In addition, we give each internal node of the tree a frequency that is the sum of the frequencies of all leaves in the subtree.

The Huffman coding tree for the string "you are reading chapter ten of slither into data structures and algorithms" is below:



Before we write the code to generate the Huffman code tree, we need to calculate the frequencies of characters in a body of text. To do this we'll write a simple function that maps a character to the number of times it appears in the text:

```
def  frequency(text):
    freq  = {}
    for  char  in  text:
        if  char  in  freq:
            freq[char]    += 1
        else :
            freq[  char]  = 1
    return  freq
```

To build construct the Huffman code tree, we will make use of a priority queue. The reason this approach works well is because the more frequent a character, the higher it's priority will be and hence closer to the root of the tree it will be which in turn means less branches it takes to reach it from the root so it's code-word will be shorter.

But before we can do that, we need a Node class. This will be the same as a binary tree node as a Huffman tree is a binary tree:

```
class HuffmanNode:
    def __init__(self, left, right, key):
        self.left = left
        self.right = right
        self.key = key
```

Let's take a look at the code for generating the Huffman tree:

```
from queue import PriorityQueue

def huffman(text):
    # Calcualte frequency of every character
    char_frequency = frequency(text)

    # Initialize a priority queue
    # This create the order for our nodes
    pq = PriorityQueue()

    # So we don't create multiple nodes for the same character
    seen = set ()

    # Create the leaf nodes
    for char in text:
        if char not in seen:
            node = HuffmanNode(None, None, char)
            pq.put((char_frequency[char], id (node), node))
                seen.add(char)

    while pq.qsize() > 1:
        # Get two most frequent nodes
        (frequency_left_node, id_left_node, left_node) = pq.get()
        (frequency_right_node, id_right_node, right_node) = pq.get()

        # Create a new intermediate node
        node = HuffmanNode(left_node, right_node, None)

        # Put the new node in the priority queue
        pq.put(((frequency_left_node + frequency_right_node), id (node)
, node))

    # Get the root of the tree
    (root_frequency, root_id, tree_root) = pq.get()
    return tree_root
```

In the first line of the function, we generate the frequencies dictionary using the function we wrote before this. We then initialize a priority queue that we will use to generate the tree.

The tree generation works by creating nodes for all unique characters in the text and adding them to the priority queue. The nodes containing the most frequent characters will be closer to the top of the priority queue. Then, we merge the two nodes which contain the most frequent characters by creating a non-leaf node and setting the two most frequent character nodes as the left and right children. The key attribute of the internal nodes is set to *None*. Only the leaf nodes (those which represent characters in the body of text) will have their key attribute set to the character they represent. We continue this process until there is a single node left in the priority queue which represents the root node of the Huffman tree.

We are using the PriorityQueue class from the queue library. At the end of the function, we return the root node of the tree.

Now that we have our Huffman tree, we can use it to generate the Huffman codes for all the characters. To do this we will write a function that maps each character to it's equivalent Huffman code.

```
def generate_huffman_codes(node, left=True, huffman_code=""):
    # Found a leaf node - return the dictionar entry for that key
    if type(node.key) is str:
        return {(node.key): huffman_code}

    (left_child, right_child) = node.left, node.right

    codes = dict()
    codes.update(generate_huffman_codes(left_child, True, huffman_code
+ "0"))
    codes.update(generate_huffman_codes(right_child, False, huffman_co
de + "1"))
    return codes
```

The above function is a recursive function that generates the Huffman code for each character by traversing the tree from the root. The base case (*if type(node.key) is str*) will set an entry in the codes dictionary for that character when a leaf node is reached. The code is built up by adding a 1 to the *huffman_code* each time we go to a right subtree and a 0 each time we go to a left subtree.

Now that we have a dictionary mapping characters in a text to their respective Huffman code, we can write a simple function that encodes the text with the Huffman code.

```
def encode(text, codes):
    huffman_string = ""
    for char in text:
            huffman_string += codes[char] + " "
      return huffman_string.strip()
```

I will be using the following text to demonstrate Huffman coding in action:

in computer science and information theory, a huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. the process of finding or using such a code proceeds by means of huffman coding, an algorithm developed by david a. huffman while he was a sc.d. student at mit, and published in the 1952 paper "a method for the construction of minimum-redundancy codes". the output from huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file). the algorithm derives this table from the estimated probability or frequency of occurrence (weight) for each possible value of the source symbol. as in other entropy encoding methods, more common symbols are generally represented using fewer bits than less common symbols. huffman's method can be efficiently implemented, finding a code in time linear to the number of input weights if these weights are sorted however, although optimal among methods encoding symbols separately, huffman coding is not always optimal among all compression methods - it is replaced with arithmetic coding or asymmetric numeral systems if better compression ratio is required.

We can use our functions as follows. The *text* variable contains the above paragraph:

```
# The root of the huffman code tree
root = huffman(text)

# Used for encoding
codes = generate_huffman_codes(root)

encoded_text = encode(text, codes)
```

The value stored in *encoded_text* will be much longer than the text itself. However, if we were to write these codes (where each 1 and 0 are bits) to a binary file, I can guarantee you that the size of the output binary file will be a lot smaller than the size of the file that contains the example text.

Below is a sample of the *encoded_text*.

```
1000 0011 110 11101 1010 11110 111110 111111 0110 000 0010 110 0101 11
101 1000 000 0011   11101 000  110 0111 0011 10110 110 1000 0011 01001 10
10 0010 11110 0111 0110 1000 1010 0011 110 0110 11100 000 1010 0010 10
0111 10111111 110 0111 110 11100 111111 01001 01001 11110 0111 0011 11
0 11101 1010 10110 000 110 1000 0101 110 0111 110 111110 0111 0        010 011
0 10 00 11101 111111
```

We can clearly see that some characters are represented with less bits (these are the most frequent characters) than others (the less frequent characters). For example, from the Huffman codes generated for this text, a space `' '`, is represented by the Huffman code *110*, whereas the letter *x*, which is a very infrequent character is represented by the Huffman code *0100001010*. If we were using the extended ASCII encoding scheme for the text, each character would be represented by 8 bits. Here however, we have reduced the number of bits that represent a space (8) down to just three bits. However, in extended ASCII, the letter 'x' would be represented by 8 bits whereas it's represented as 10 bits in the Huffman code. This is ok though as for both these characters we have a total bit representation of 13 bits which is less than the 16 bits that would be needed in extended ASCII. Over a large text, the difference becomes even more significant.

For the sample text alone, the uncompressed version is 1,211 bytes in size (if encoded using extended ASCII), however the Huffman coded text (compressed text) is only 654 bytes in size. If our text were a large corpus and was gigabytes in size, we save a lot of space if we compressed it using the Huffman coding algorithm.

The decompress the text, we can do the reverse of what we did to encode it. Firstly, we can create an inverse codes dictionary:

```
codes_inverse = {v: k for k, v in codes.items()}
```

Using this, we can read in the Huffman coded text and use the dictionary to convert the codes back to the original text:

```
def decode(huffman_coded_text, codes_inverse):
    huffman_codes = huffman_coded_text.split(" ")
    decoded_text = ""
    for code in huffman_codes:
        decoded_text += codes_inverse[code]
    retu rn decoded_text
```

We can use all of our functions as follows:

```
# The root of the huffman code tree
root = huffman(s)

# Used for encoding
codes = generate_huffman_codes(root)

# Used for decoding
codes_inverse = {v: k for k, v in codes.items()}

# Encode the text using Huffman codes
encoded = encode(s, codes)

# Decode using the Huffman codes
decoded = decode(encoded, codes_inverse)

# Will print out the original text
print (decoded)
```

In the real world, when we compress a file using something like GZIP, some representation of the Huffman codes are stored as part of the zipped file. That way, when we send the compressed file over the internet and someone downloads it and wants to unzip it, they use the representation of the Huffman codes we sent along with the compressed file itself to do that. That representation could be the Huffman tree, or it could be the codes dictionary we generated. In the case of GZIP it is a Huffman code table - just another way of representing the codes for each character.