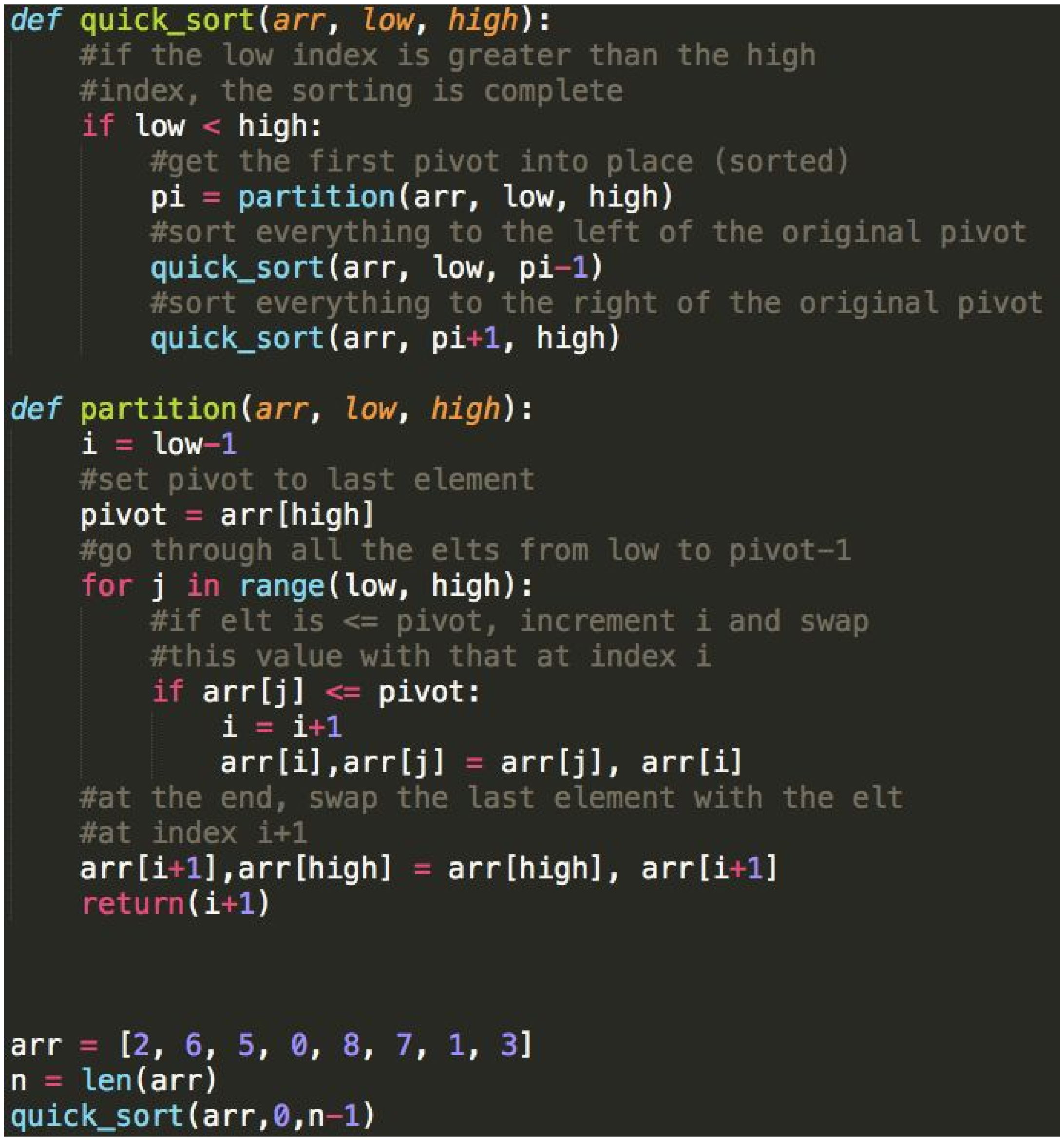
1. Sorting

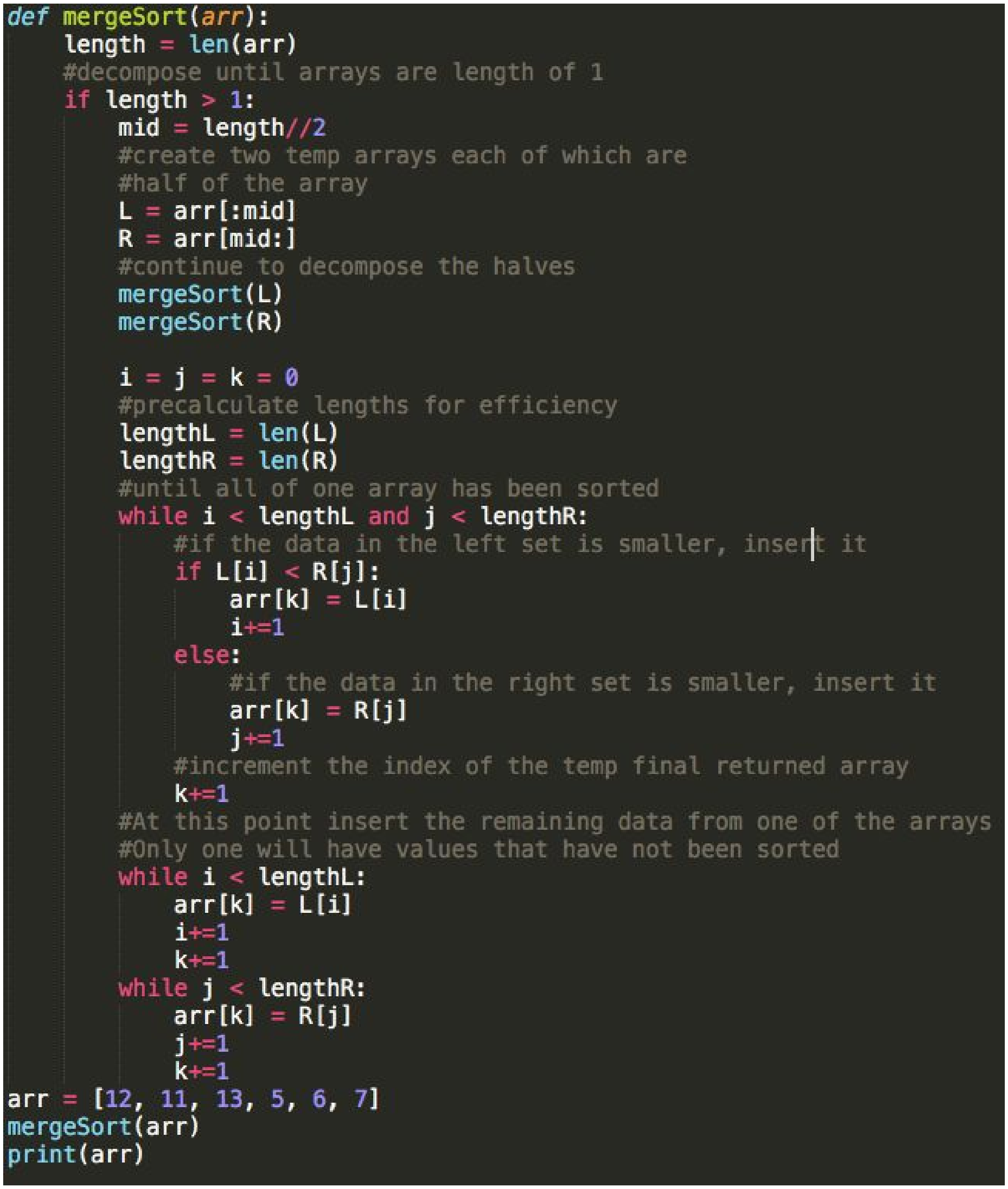
Know how to sort. Don't do bubble-sort. You should know the details of at least one n*log(n) sorting algorithm, preferably two (e.g. quicksort and merge sort). Merge sort can be highly useful in situations where quicksort is impractical, so take a look at it.

Quicksort:

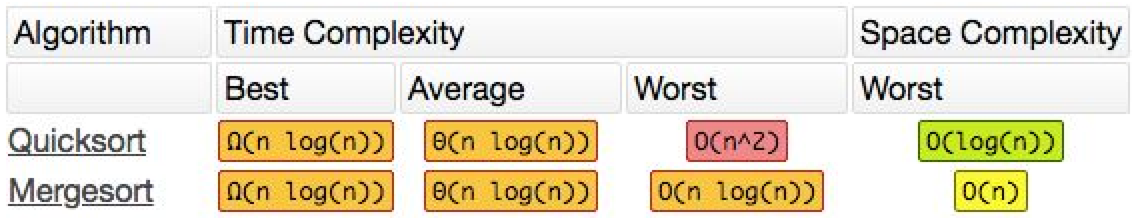


```python
def quick_sort(arr, low, high):
    #if the low index is greater than the high
    #index, the sorting is complete
    if low < high:
        #get the first pivot into place (sorted)
        pi = partition(arr, low, high)
        #sort everything to the left of the original pivot
        quick_sort(arr, low, pi-1)
        #sort everything to the right of the original pivot
        quick_sort(arr, pi+1, high)

def partition(arr, low, high):
    i = low-1
    #set pivot to last element
    pivot = arr[high]
    #go through all the elts from low to pivot-1
    for j in range(low, high):
        #if elt is <= pivot, increment i and swap
        #this value with that at index i
        if arr[j] <= pivot:
            i = i+1
            arr[i],arr[j] = arr[j], arr[i]
    #at the end, swap the last element with the elt
    #at index i+1
    arr[i+1],arr[high] = arr[high], arr[i+1]
    return(i+1)



arr = [2, 6, 5, 0, 8, 7, 1, 3]
n = len(arr)
quick_sort(arr,0,n-1)
```

Mergesort:



```python
def mergeSort(arr):
    length = len(arr)
    #decompose until arrays are length of 1
    if length > 1:
        mid = length//2
        #create two temp arrays each of which are
        #half of the array
        L = arr[:mid]
        R = arr[mid:]
        #continue to decompose the halves
        mergeSort(L)
        mergeSort(R)

        i = j = k = 0
        #precalculate lengths for efficiency
        lengthL = len(L)
        lengthR = len(R)
        #until all of one array has been sorted
        while i < lengthL and j < lengthR:
            #if the data in the left set is smaller, insert it
            if L[i] < R[j]:
                arr[k] = L[i]
                i+=1
            else:
                #if the data in the right set is smaller, insert it
                arr[k] = R[j]
                j+=1
            #increment the index of the temp final returned array
            k+=1
        #At this point insert the remaining data from one of the arrays
        #Only one will have values that have not been sorted
        while i < lengthL:
            arr[k] = L[i]
            i+=1
            k+=1
        while j < lengthR:
            arr[k] = R[j]
            j+=1
            k+=1
arr = [12, 11, 13, 5, 6, 7]
mergeSort(arr)
print(arr)
```



Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)

In practice, quicksort, despite have worse worst-case scenario run time, runs faster that mergesort.

Then why use mergesort? -- Linked Lists

It might be helpful to see why quicksort is usually faster than mergesort, since if you understand the reasons you can pretty quickly find some cases where mergesort is a clear winner. Quicksort usually is better than mergesort for two reasons:

1. Quicksort has better locality of reference than mergesort, which means that the accesses performed in quicksort are usually faster than the corresponding accesses in mergesort.
2. Quicksort uses worst-case O(log n) memory (if implemented correctly), while mergesort requires O(n) memory due to the overhead of merging.

There's one scenario, though, where these advantages disappear. Suppose you want to sort a linked list of elements. The linked list elements are scattered throughout memory, so advantage (1) disappears (there's no locality of reference). Second, linked lists can be merged with only O(1) space overhead instead of O(n) space overhead, so advantage (2) disappears. Consequently, you usually will find that mergesort is a superior algorithm for sorting linked lists, since it makes fewer total comparisons and isn't susceptible to a poor pivot choice.

2. Hash Tables

Be prepared to explain how hash tables work and be able to implement one using only arrays in your favorite language in about the space of one interview.

Hash tables are a type of data structure in which the address or the index value of the data element is generated from a hash function. That makes accessing the data faster as the index value behaves as a key for the data value. In other words Hash table stores key-value pairs but the key is generated through a hashing function.

So the search and insertion function of a data element becomes much faster as the key values themselves become the index of the array which stores the data.

In Python, the Dictionary data types represent the implementation of hash tables. The Keys in the dictionary satisfy the following requirements.

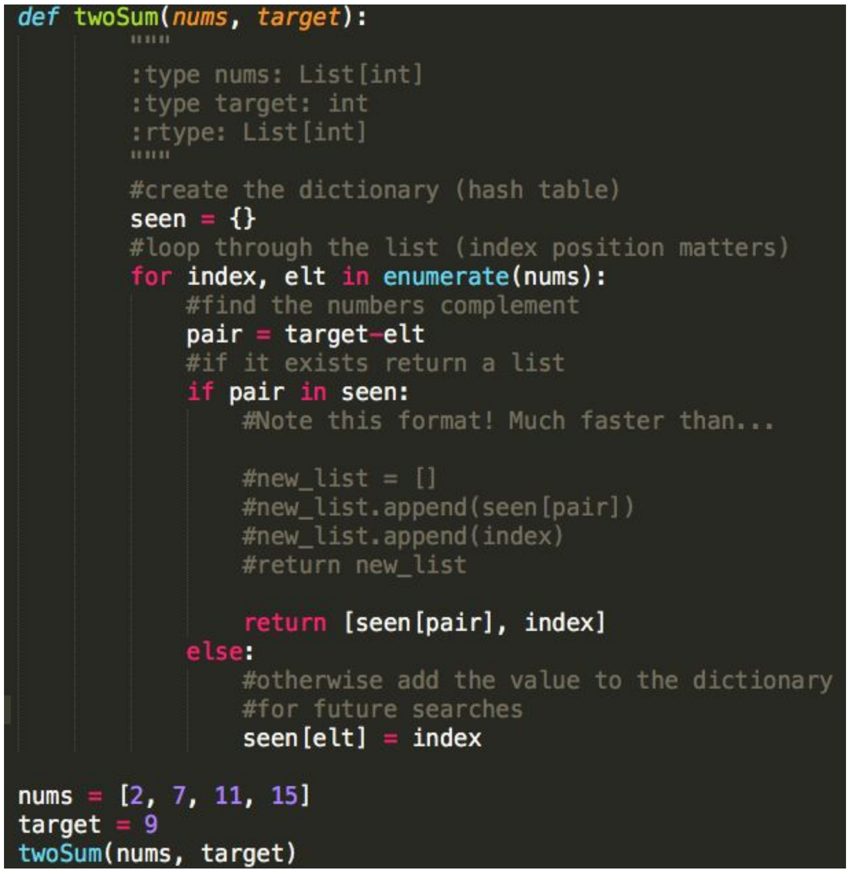
- The keys of the dictionary are hashable i.e. the are generated by hashing function which generates unique result for each unique value supplied to the hash function.
- The order of data elements in a dictionary is not fixed.

USEFUL SCENARIOS: TWO SUM

Given an array of integers, return indices of the two numbers such that they add up to a specific target.

You may assume that each input would have *exactly* one solution, and you may not use the *same* element twice.

Ex: nums = `[2, 7, 11, 15], target = 9`



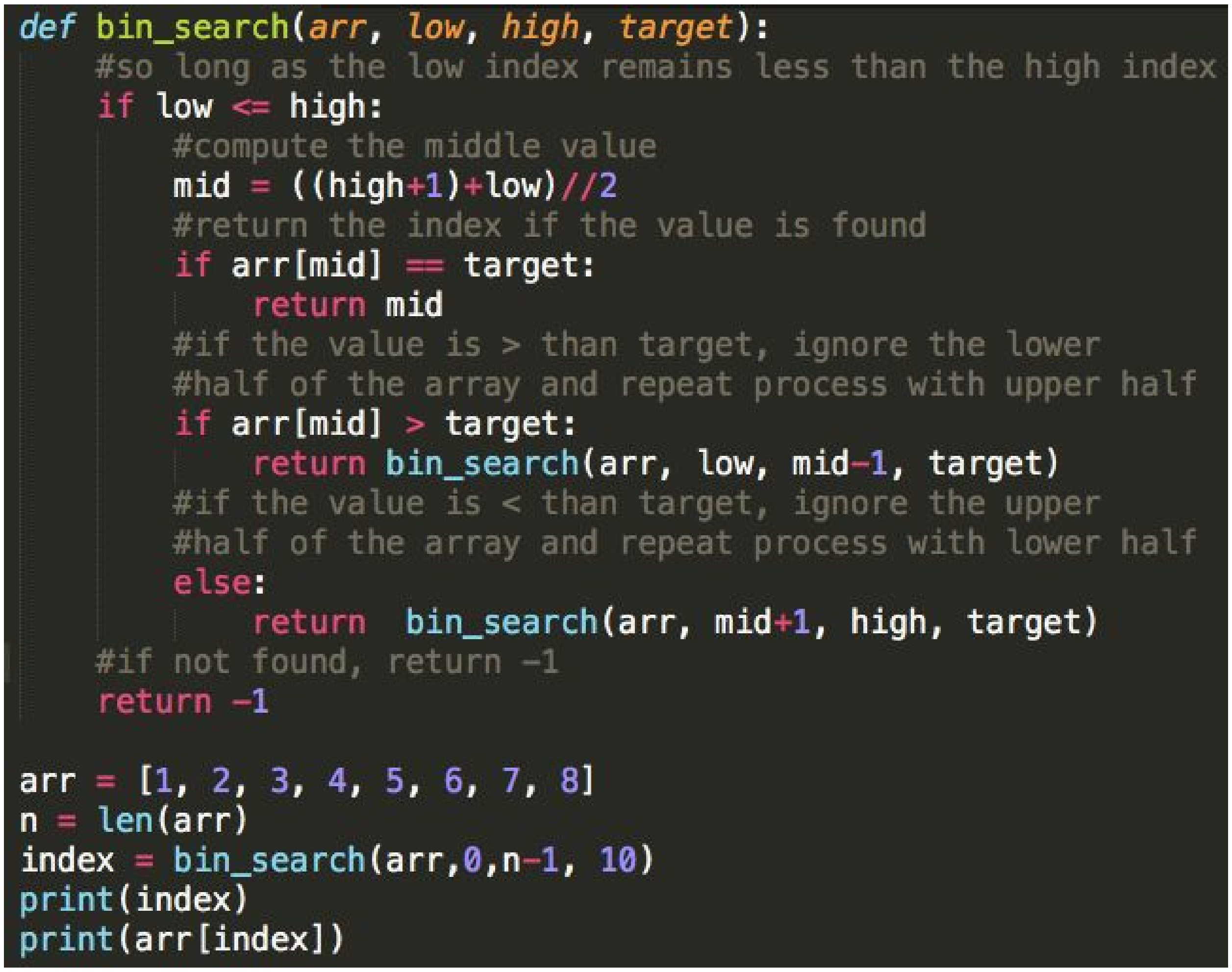
```python
def twoSum(nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[int]
        """
        #create the dictionary (hash table)
        seen = {}
        #loop through the list (index position matters)
        for index, elt in enumerate(nums):
            #find the numbers complement
            pair = target-elt
            #if it exists return a list
            if pair in seen:
                #Note this format! Much faster than...

                #new_list = []
                #new_list.append(seen[pair])
                #new_list.append(index)
                #return new_list

                return [seen[pair], index]
            else:
                #otherwise add the value to the dictionary
                #for future searches
                seen[elt] = index

nums = [2, 7, 11, 15]
target = 9
twoSum(nums, target)
```

3. Algorithms

Algorithms that are used to solve Google problems include sorting (plus searching and binary search), divide-and-conquer, dynamic programming/memoization, greediness, recursion or algorithms linked to a specific data structure.

Binary Search:

NOTE: ONLY WORKS IF ARRAY IS SORTED!



```python
def bin_search(arr, low, high, target):
    #so long as the low index remains less than the high index
    if low <= high:
        #compute the middle value
        mid = ((high+1)+low)//2
        #return the index if the value is found
        if arr[mid] == target:
            return mid
        #if the value is > than target, ignore the lower
        #half of the array and repeat process with upper half
        if arr[mid] > target:
            return bin_search(arr, low, mid-1, target)
        #if the value is < than target, ignore the upper
        #half of the array and repeat process with lower half
        else:
            return  bin_search(arr, mid+1, high, target)
    #if not found, return -1
    return -1

arr = [1, 2, 3, 4, 5, 6, 7, 8]
n = len(arr)
index = bin_search(arr,0,n-1, 10)
print(index)
print(arr[index])
```

Divide & Conquer Algorithms:

Divide and Conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps.

1. *Divide:* Break the given problem into subproblems of same type.
2. *Conquer:* Recursively solve these subproblems
3. *Combine:* Appropriately combine the answers

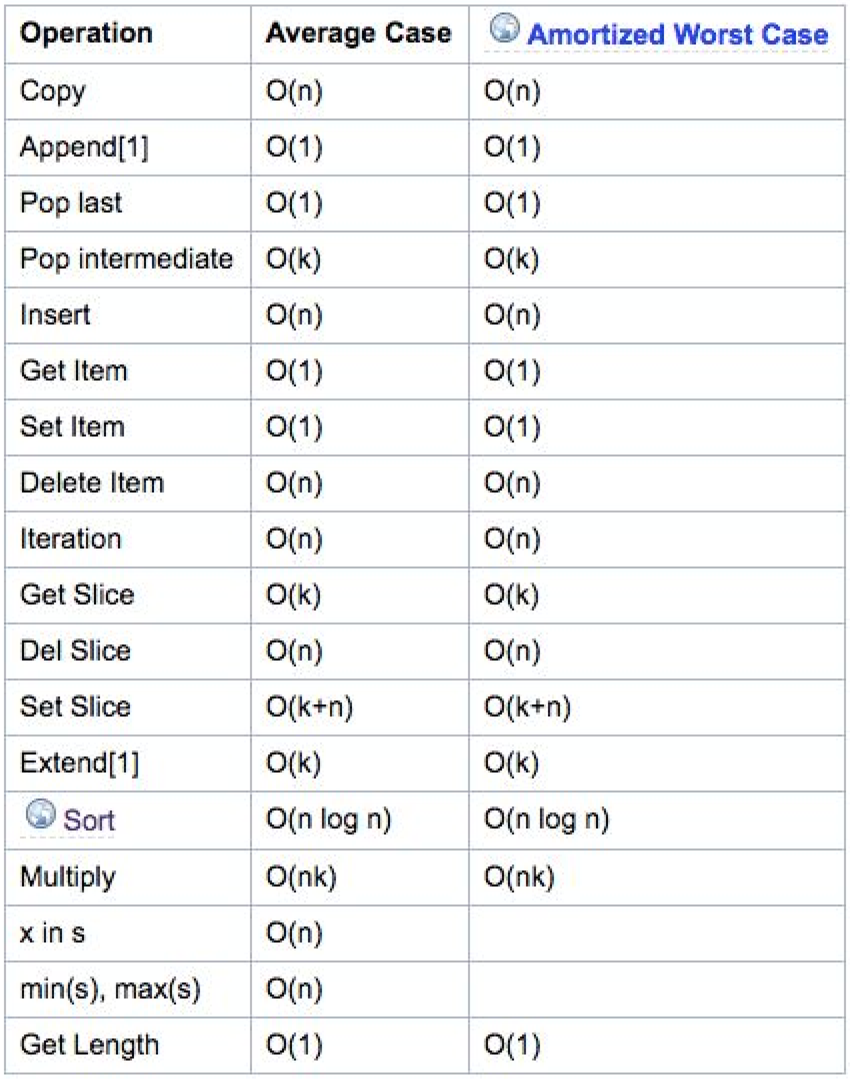
Examples include: binary search, mergesort, quicksort

3. Data Structures

You should study up on as many data structures as possible. Data structures most frequently used are arrays, linked lists, stacks, queues, hash-sets, hash-maps, hash-tables, dictionary, trees and binary trees, heaps and graphs. You should know the data structure inside out, and what algorithms tend to go along with each data structure.

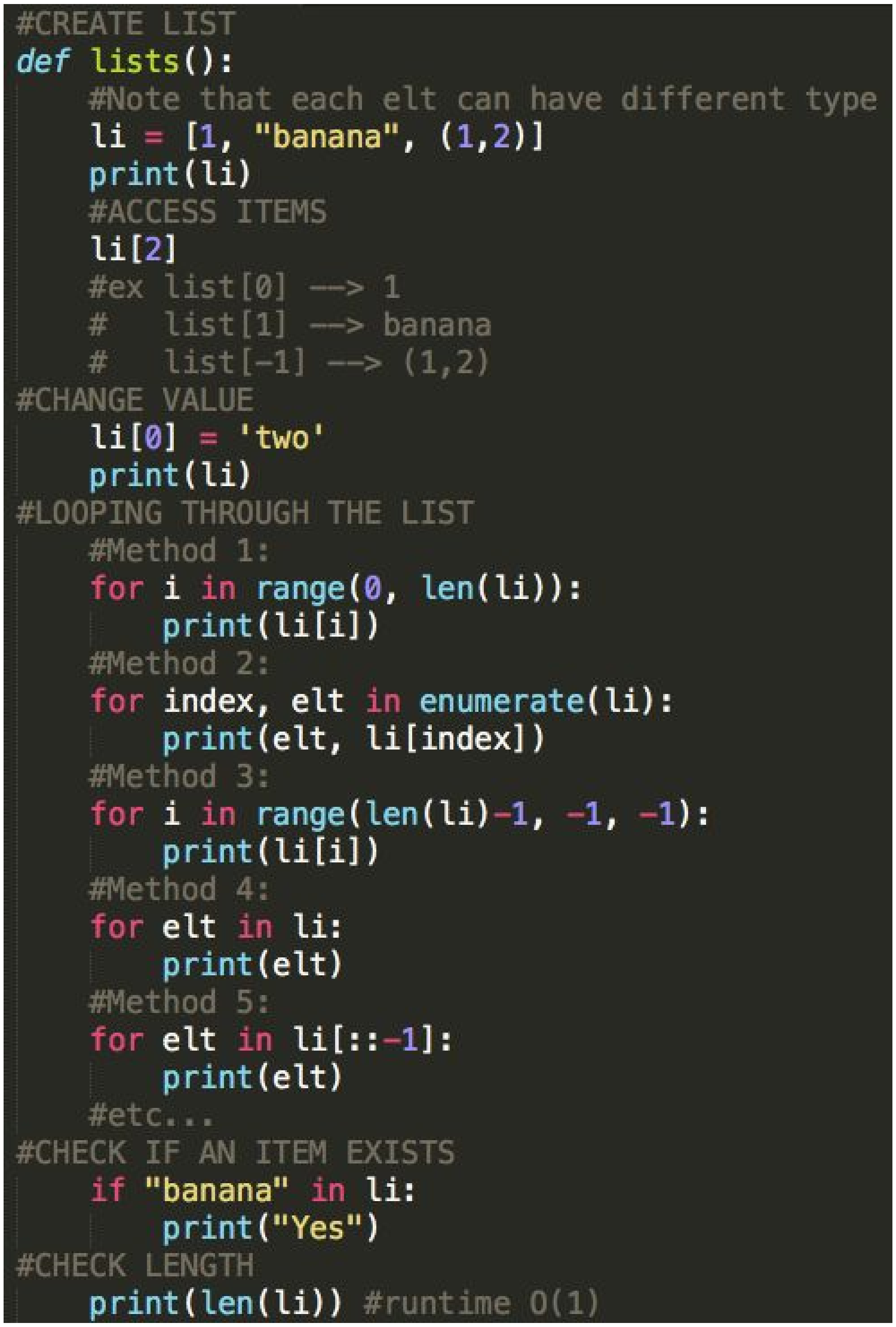
Arrays (Lists):

A list is a collection which is ordered and changeable. In Python lists are written with square brackets.

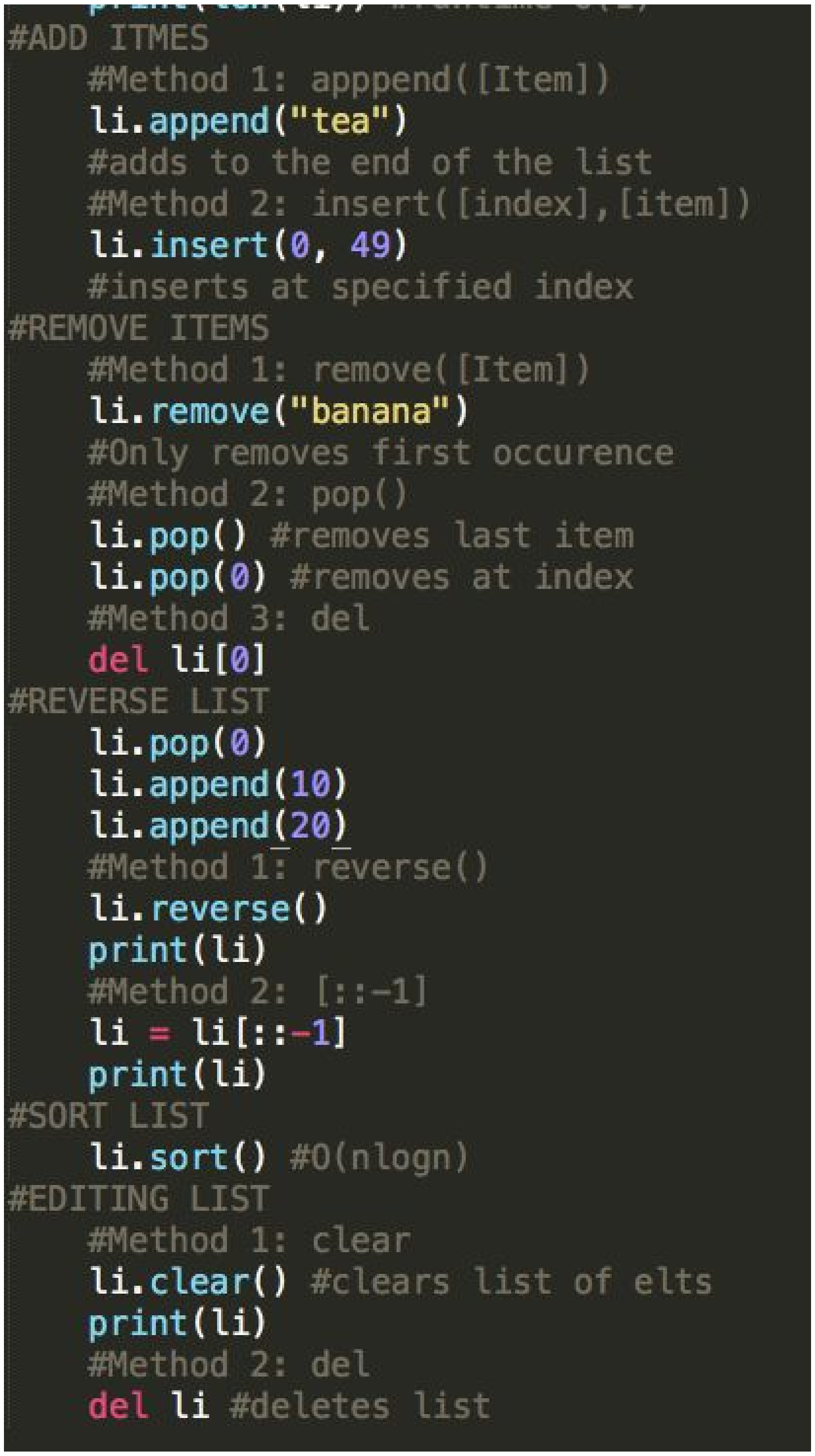


Operation	Average Case	Amortized Worst Case
Copy	O(n)	O(n)
Append[1]	O(1)	O(1)
Pop last	O(1)	O(1)
Pop intermediate	O(k)	O(k)
Insert	O(n)	O(n)
Get Item	O(1)	O(1)
Set Item	O(1)	O(1)
Delete Item	O(n)	O(n)
Iteration	O(n)	O(n)
Get Slice	O(k)	O(k)
Del Slice	O(n)	O(n)
Set Slice	O(k+n)	O(k+n)
Extend[1]	O(k)	O(k)
Sort	O(n log n)	O(n log n)
Multiply	O(nk)	O(nk)
x in s	O(n)	
min(s), max(s)	O(n)	
Get Length	O(1)	O(1)

Python time complexities for built-in list functions



```
#CREATE LIST
def lists():
    #Note that each elt can have different type
    li = [1, "banana", (1,2)]
    print(li)
    #ACCESS ITEMS
    li[2]
    #ex list[0] --> 1
    #   list[1] --> banana
    #   list[-1] --> (1,2)
#CHANGE VALUE
    li[0] = 'two'
    print(li)
#LOOPING THROUGH THE LIST
    #Method 1:
    for i in range(0, len(li)):
        print(li[i])
    #Method 2:
    for index, elt in enumerate(li):
        print(elt, li[index])
    #Method 3:
    for i in range(len(li)-1, -1, -1):
        print(li[i])
    #Method 4:
    for elt in li:
        print(elt)
    #Method 5:
    for elt in li[::-1]:
        print(elt)
    #etc...
#CHECK IF AN ITEM EXISTS
    if "banana" in li:
        print("Yes")
#CHECK LENGTH
    print(len(li)) #runtime O(1)
```

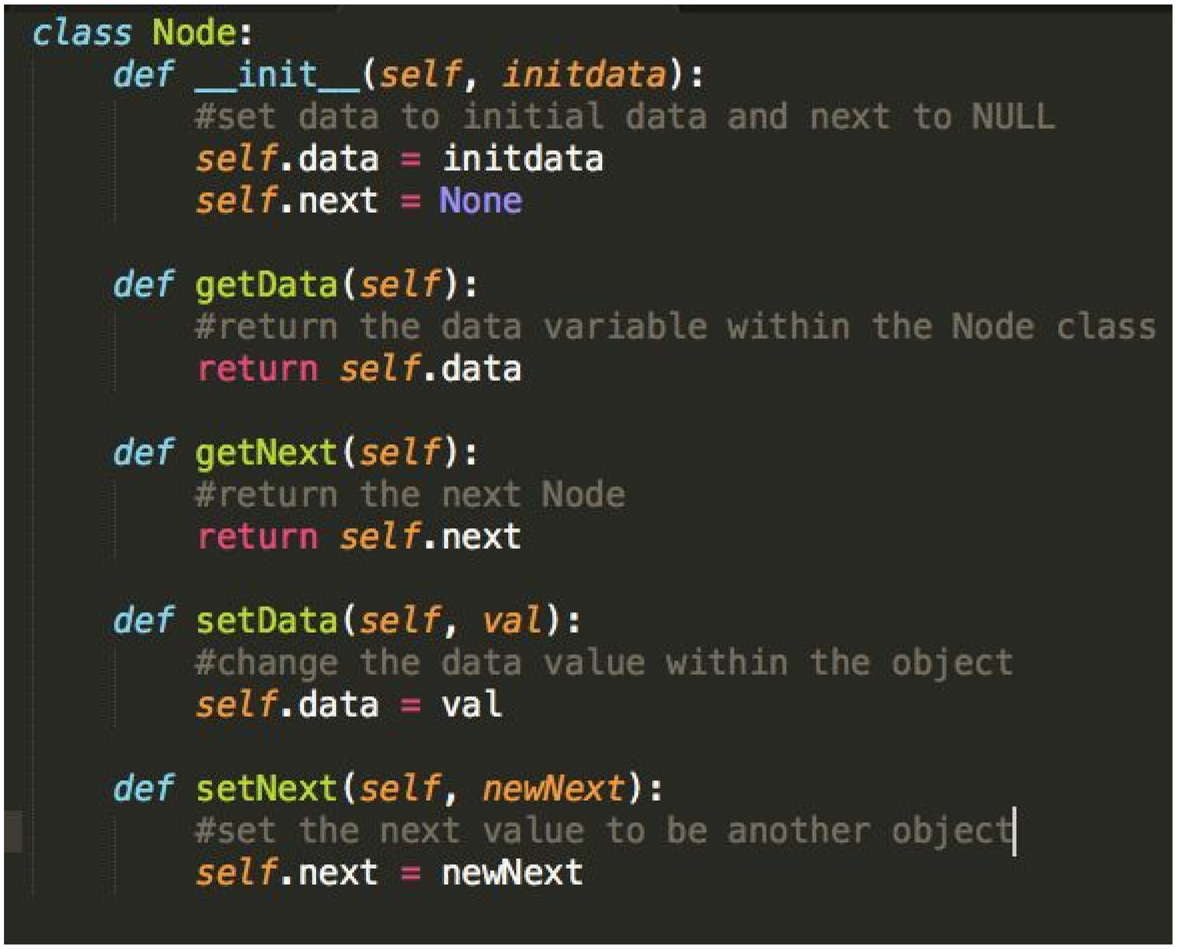


```
#ADD ITMES
    #Method 1: apppend([Item])
    li.append("tea")
    #adds to the end of the list
    #Method 2: insert([index],[item])
    li.insert(0, 49)
    #inserts at specified index
#REMOVE ITEMS
    #Method 1: remove([Item])
    li.remove("banana")
    #Only removes first occurence
    #Method 2: pop()
    li.pop() #removes last item
    li.pop(0) #removes at index
    #Method 3: del
    del li[0]
#REVERSE LIST
    li.pop(0)
    li.append(10)
    li.append(20)
    #Method 1: reverse()
    li.reverse()
    print(li)
    #Method 2: [::-1]
    li = li[::-1]
    print(li)
#SORT LIST
    li.sort() #O(nlogn)
#EDITING LIST
    #Method 1: clear
    li.clear() #clears list of elts
    print(li)
    #Method 2: del
    del li #deletes list
```

Linked Lists:

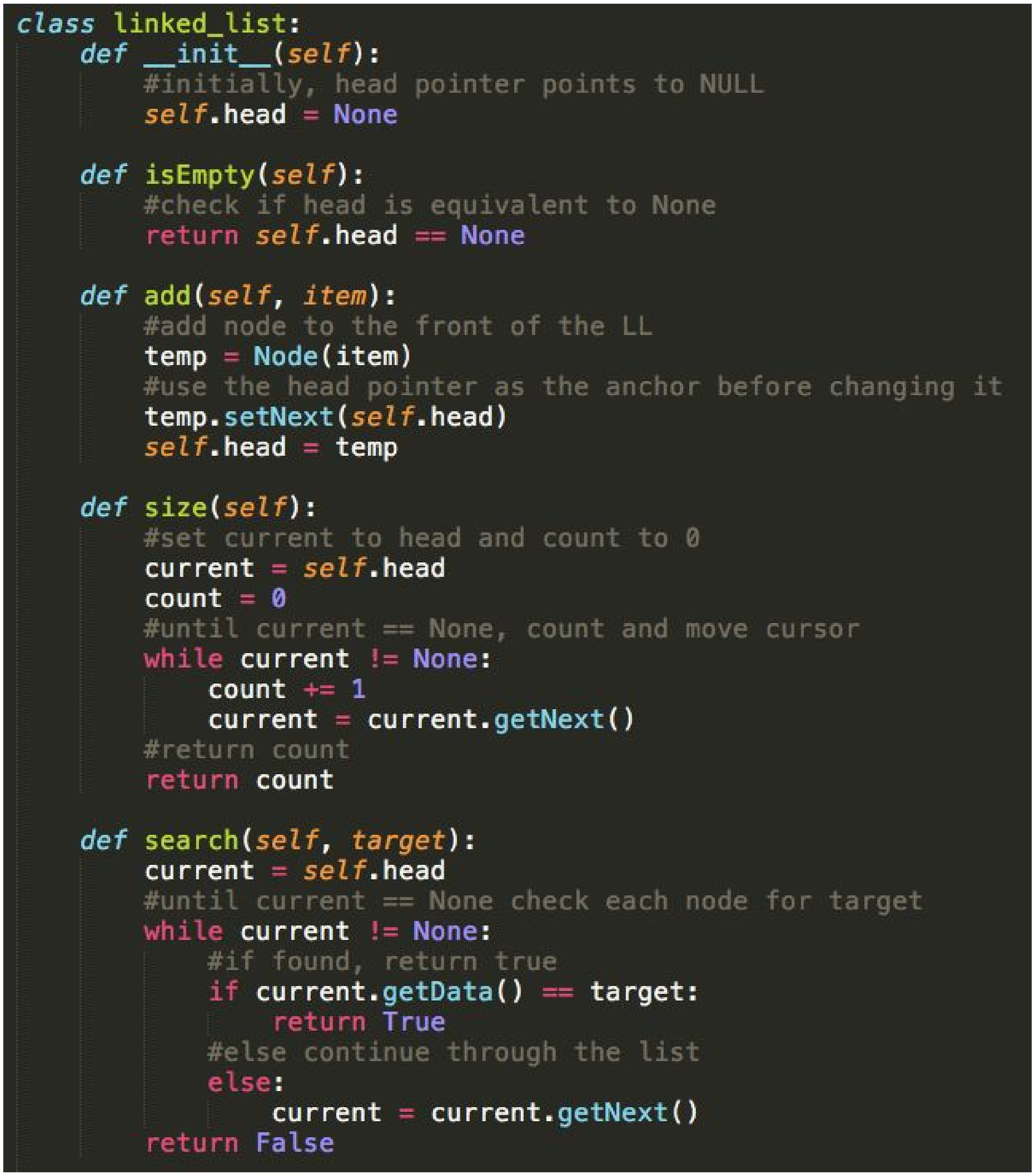
In order to implement an unordered list, we will construct what is commonly known as a linked list. Recall that we need to be sure that we can maintain the relative positioning of the items. However, there is no requirement that we maintain that positioning in contiguous memory.

The basic building block for the linked list implementation is the **node**. Each node object must hold at least two pieces of information. First, the node must contain the list item itself. We will call this the **data field** of the node. In addition, each node must hold a reference to the next node.

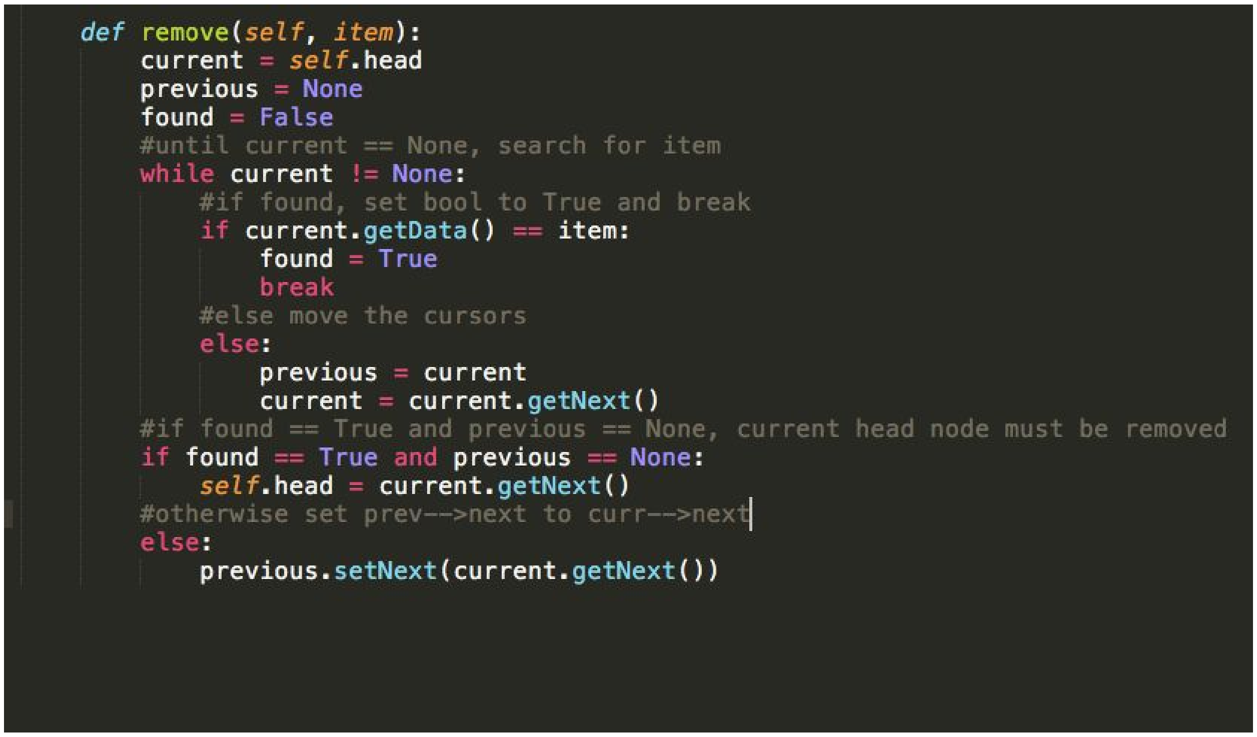


```
class Node:
    def __init__(self, initdata):
        #set data to initial data and next to NULL
        self.data = initdata
        self.next = None

    def getData(self):
        #return the data variable within the Node class
        return self.data

    def getNext(self):
        #return the next Node
        return self.next

    def setData(self, val):
        #change the data value within the object
        self.data = val

    def setNext(self, newNext):
        #set the next value to be another object
        self.next = newNext
```

The unordered list will be built from a collection of nodes, each linked to the next by explicit references. As long as we know where to find the first node (containing the first item), each item after that can be found by successively following the next links.



```python
class linked_list:
    def __init__(self):
        #initially, head pointer points to NULL
        self.head = None

    def isEmpty(self):
        #check if head is equivalent to None
        return self.head == None

    def add(self, item):
        #add node to the front of the LL
        temp = Node(item)
        #use the head pointer as the anchor before changing it
        temp.setNext(self.head)
        self.head = temp

    def size(self):
        #set current to head and count to 0
        current = self.head
        count = 0
        #until current == None, count and move cursor
        while current != None:
            count += 1
            current = current.getNext()
        #return count
        return count

    def search(self, target):
        current = self.head
        #until current == None check each node for target
        while current != None:
            #if found, return true
            if current.getData() == target:
                return True
            #else continue through the list
            else:
                current = current.getNext()
        return False
```



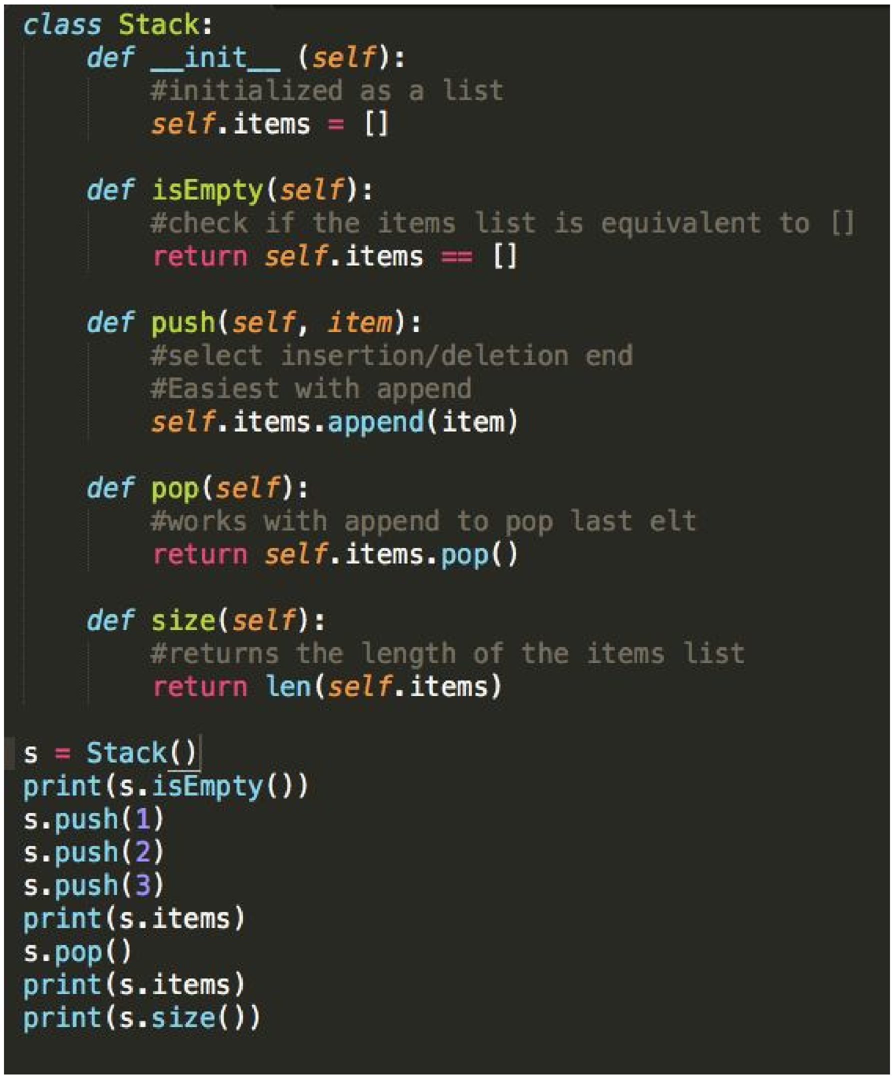
```
def remove(self, item):
    current = self.head
    previous = None
    found = False
    #until current == None, search for item
    while current != None:
        #if found, set bool to True and break
        if current.getData() == item:
            found = True
            break
        #else move the cursors
        else:
            previous = current
            current = current.getNext()
    #if found == True and previous == None, current head node must be removed
    if found == True and previous == None:
        self.head = current.getNext()
    #otherwise set prev-->next to curr-->next
    else:
        previous.setNext(current.getNext())
```

Stacks:

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

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

Src =
http://www.interactivepython.org/courselib/static/pythonds/BasicDS/ImplementingaStackinPython.html

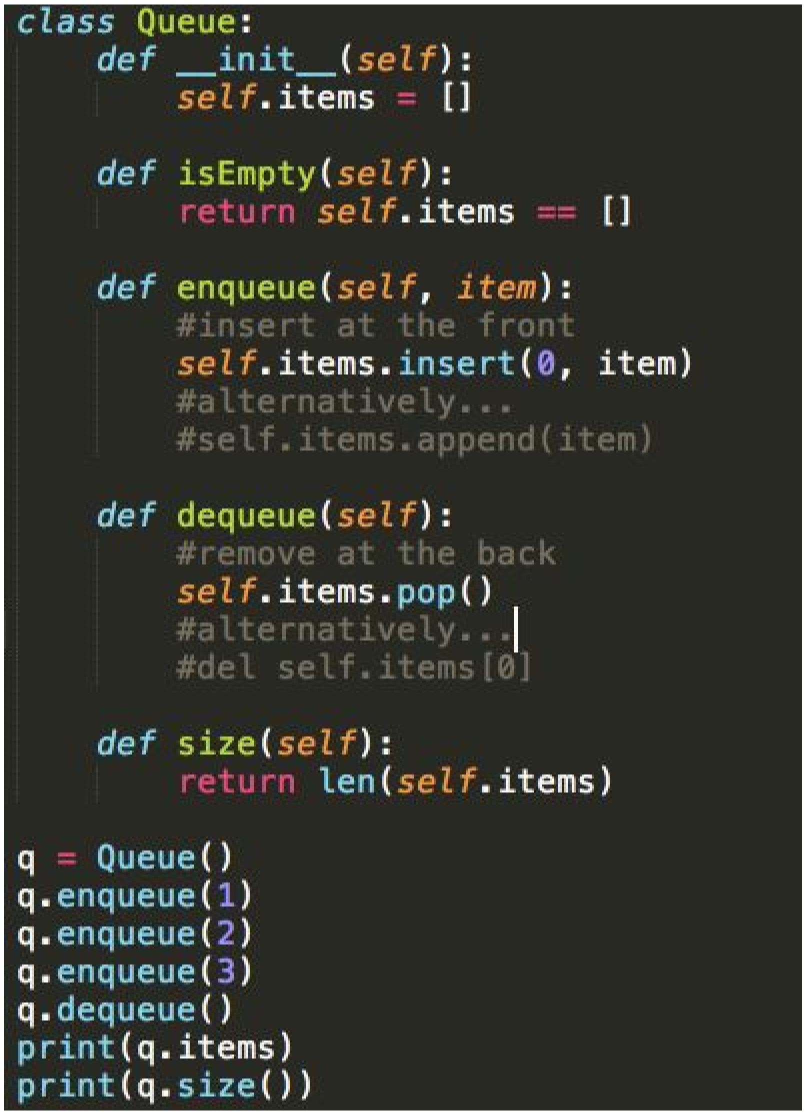


```python
class Stack:
    def __init__ (self):
        #initialized as a list
        self.items = []

    def isEmpty(self):
        #check if the items list is equivalent to []
        return self.items == []

    def push(self, item):
        #select insertion/deletion end
        #Easiest with append
        self.items.append(item)

    def pop(self):
        #works with append to pop last elt
        return self.items.pop()

    def size(self):
        #returns the length of the items list
        return len(self.items)

s = Stack()
print(s.isEmpty())
s.push(1)
s.push(2)
s.push(3)
print(s.items)
s.pop()
print(s.items)
print(s.size())
```

Queues:

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

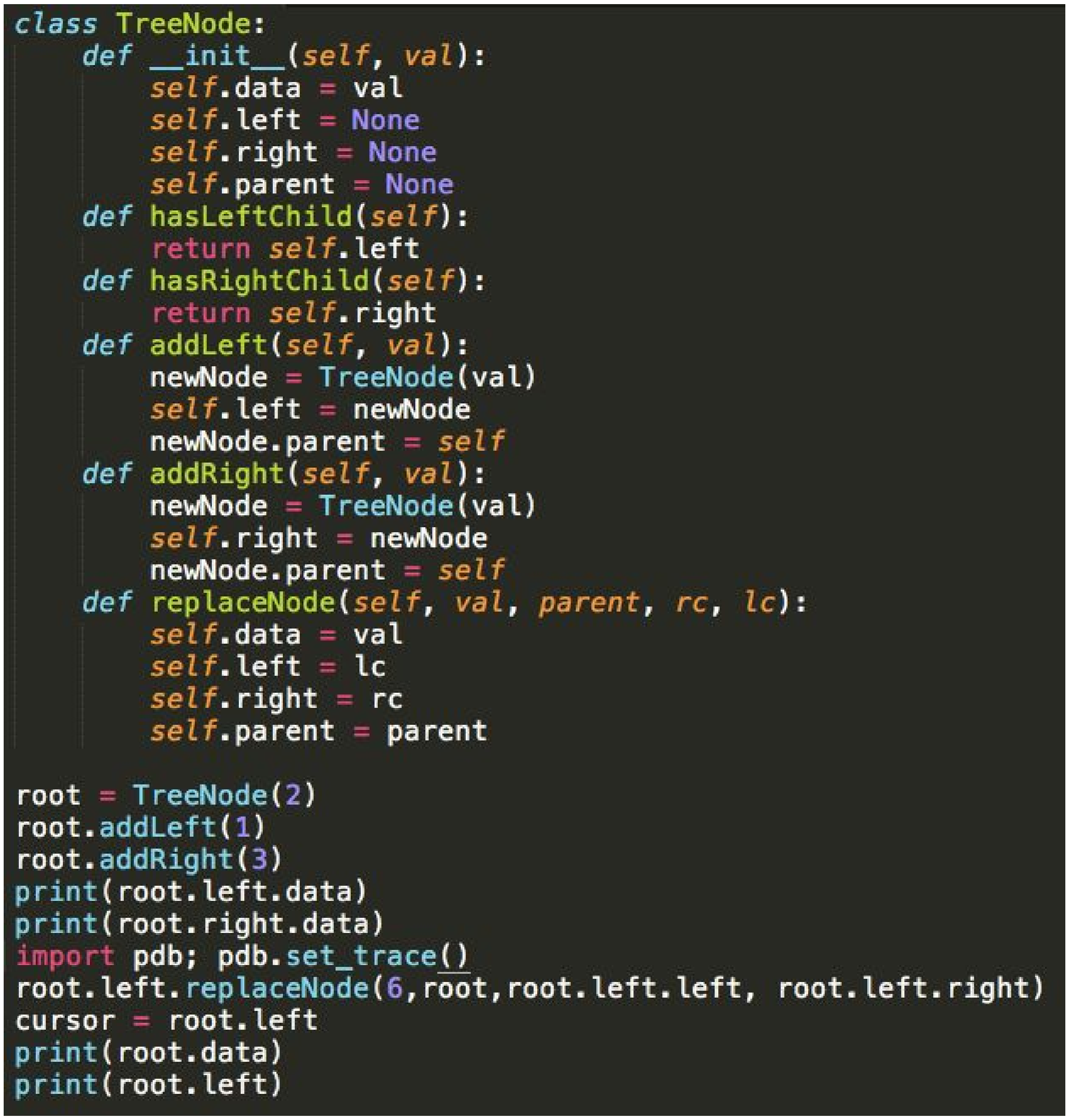
Src =
http://interactivepython.org/courselib/static/pythonds/BasicDS/ImplementingaQueueinPython.html



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

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

    def enqueue(self, item):
        #insert at the front
        self.items.insert(0, item)
        #alternatively...
        #self.items.append(item)

    def dequeue(self):
        #remove at the back
        self.items.pop()
        #alternatively...
        #del self.items[0]

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

q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
q.dequeue()
print(q.items)
print(q.size())
```

Trees/Binary Trees:

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child. Below is the class for a TreeNode. Together, TreeNodes form a binary tree. Left and right can easily be changed into a list of children, in the event the tree is not necessarily binary.

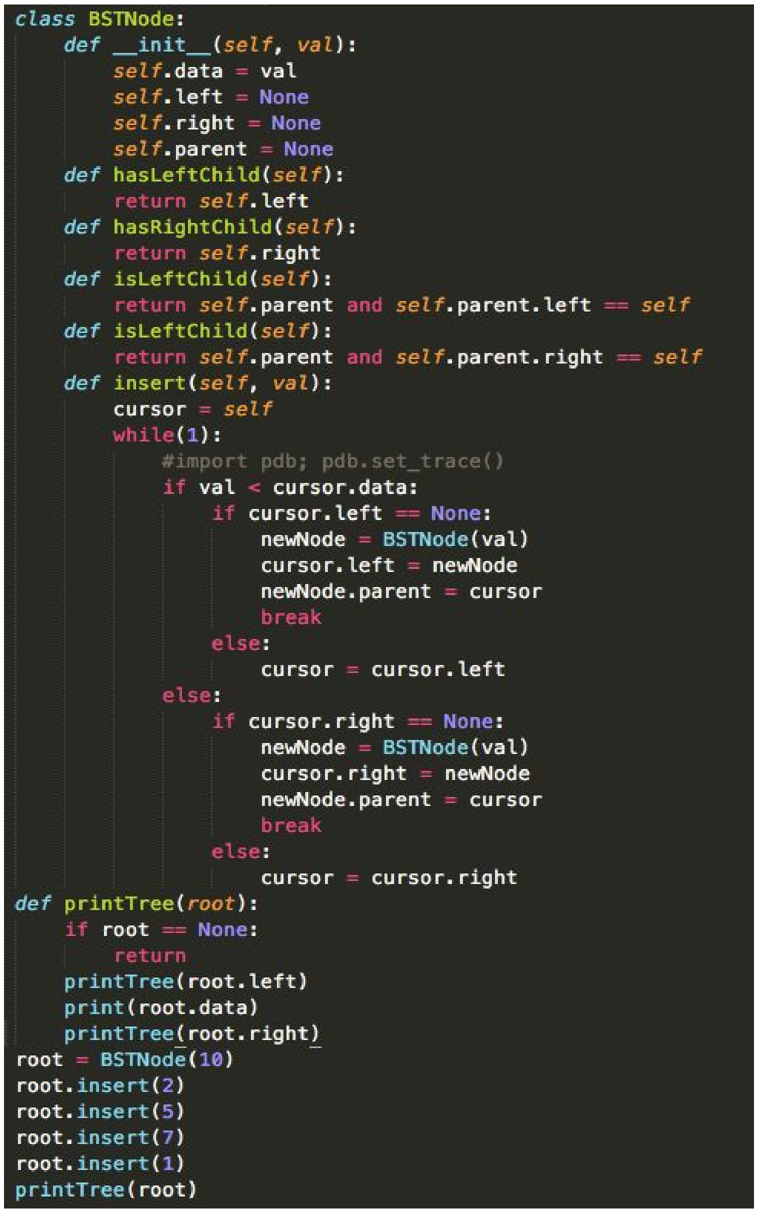


```python
class TreeNode:
    def __init__(self, val):
        self.data = val
        self.left = None
        self.right = None
        self.parent = None
    def hasLeftChild(self):
        return self.left
    def hasRightChild(self):
        return self.right
    def addLeft(self, val):
        newNode = TreeNode(val)
        self.left = newNode
        newNode.parent = self
    def addRight(self, val):
        newNode = TreeNode(val)
        self.right = newNode
        newNode.parent = self
    def replaceNode(self, val, parent, rc, lc):
        self.data = val
        self.left = lc
        self.right = rc
        self.parent = parent

root = TreeNode(2)
root.addLeft(1)
root.addRight(3)
print(root.left.data)
print(root.right.data)
import pdb; pdb.set_trace()
root.left.replaceNode(6,root,root.left.left, root.left.right)
cursor = root.left
print(root.data)
print(root.left)
```

Binary Search Tree:

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.



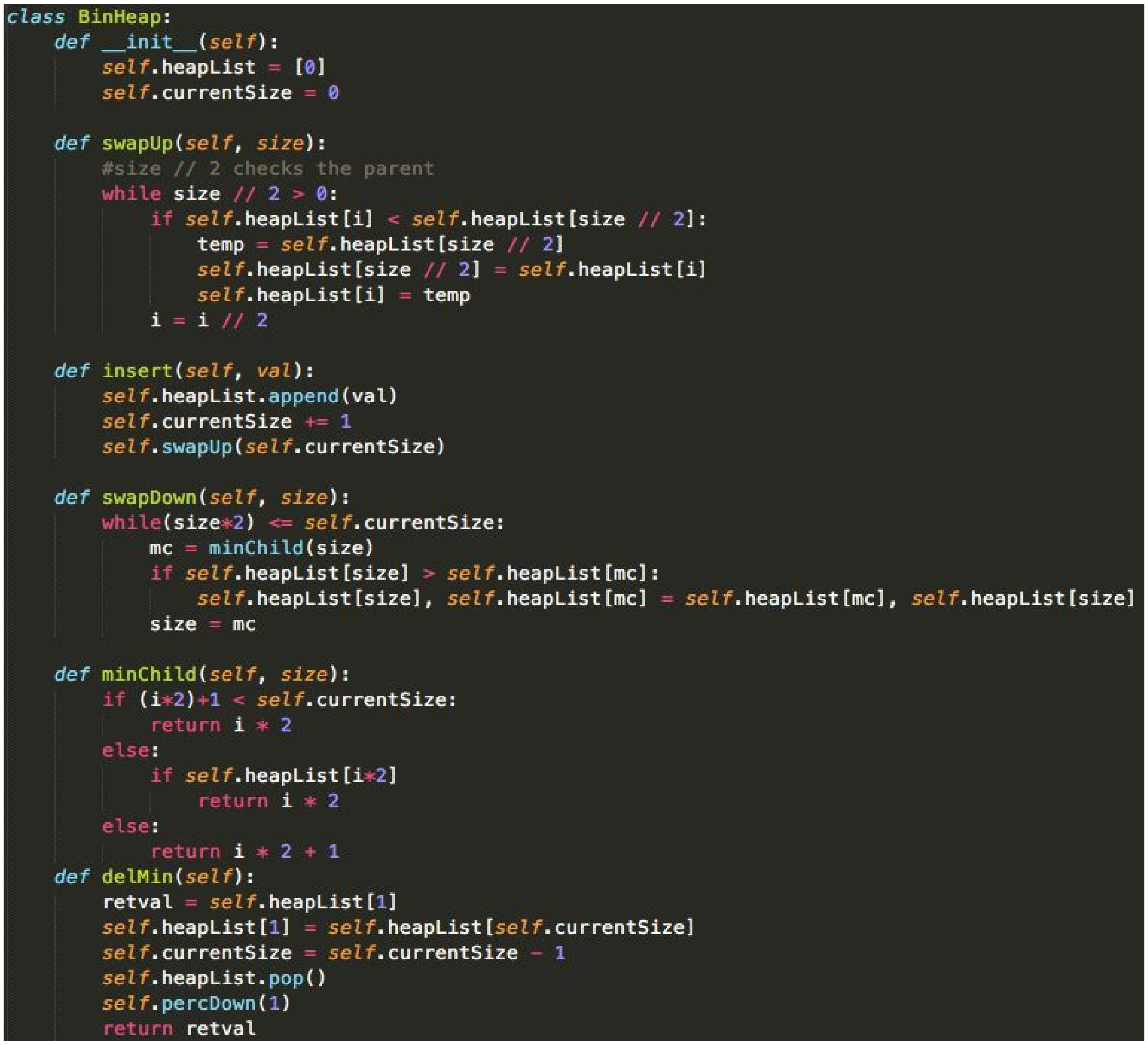
```python
class BSTNode:
    def __init__(self, val):
        self.data = val
        self.left = None
        self.right = None
        self.parent = None
    def hasLeftChild(self):
        return self.left
    def hasRightChild(self):
        return self.right
    def isLeftChild(self):
        return self.parent and self.parent.left == self
    def isLeftChild(self):
        return self.parent and self.parent.right == self
    def insert(self, val):
        cursor = self
        while(1):
            #import pdb; pdb.set_trace()
            if val < cursor.data:
                if cursor.left == None:
                    newNode = BSTNode(val)
                    cursor.left = newNode
                    newNode.parent = cursor
                    break
                else:
                    cursor = cursor.left
            else:
                if cursor.right == None:
                    newNode = BSTNode(val)
                    cursor.right = newNode
                    newNode.parent = cursor
                    break
                else:
                    cursor = cursor.right
def printTree(root):
    if root == None:
        return
    printTree(root.left)
    print(root.data)
    printTree(root.right)
root = BSTNode(10)
root.insert(2)
root.insert(5)
root.insert(7)
root.insert(1)
printTree(root)
```

Heaps:

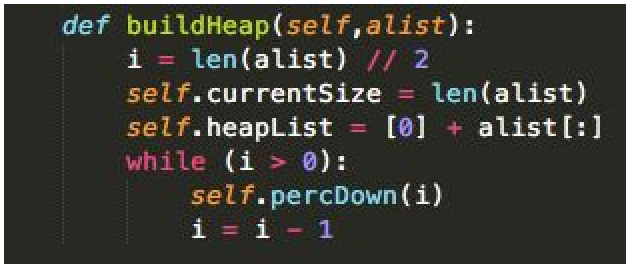
Since the entire binary heap can be represented by a single list, all the constructor will do is initialize the list and an attribute `currentSize` to keep track of the current size of the heap. Listing 1 shows the Python code for the constructor. You will notice that an empty binary heap has a single zero as the first element of `heapList` and that this zero is not used, but is there so that simple integer division can be used in later methods.

The easiest, and most efficient, way to add an item to a list is to simply append the item to the end of the list. The good news about appending is that it guarantees that we will maintain the complete tree property. The bad news about appending is that we will very likely violate the heap structure property. However, it is possible to write a method that will allow us to regain the heap structure property by comparing the newly added item with its parent. If the newly added item is less than its parent, then we can swap the item with its parent. Figure 2 shows the series of swaps needed to percolate the newly added item up to its proper position in the tree.

Src: http://interactivepython.org/courselib/static/pythonds/Trees/BinaryHeapImplementation.html



```python
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0

    def swapUp(self, size):
        #size // 2 checks the parent
        while size // 2 > 0:
            if self.heapList[i] < self.heapList[size // 2]:
                temp = self.heapList[size // 2]
                self.heapList[size // 2] = self.heapList[i]
                self.heapList[i] = temp
            i = i // 2

    def insert(self, val):
        self.heapList.append(val)
        self.currentSize += 1
        self.swapUp(self.currentSize)

    def swapDown(self, size):
        while(size*2) <= self.currentSize:
            mc = minChild(size)
            if self.heapList[size] > self.heapList[mc]:
                self.heapList[size], self.heapList[mc] = self.heapList[mc], self.heapList[size]
            size = mc

    def minChild(self, size):
        if (i*2)+1 < self.currentSize:
            return i * 2
        else:
            if self.heapList[i*2]
                return i * 2
        else:
            return i * 2 + 1
    def delMin(self):
        retval = self.heapList[1]
        self.heapList[1] = self.heapList[self.currentSize]
        self.currentSize = self.currentSize - 1
        self.heapList.pop()
        self.percDown(1)
        return retval
```

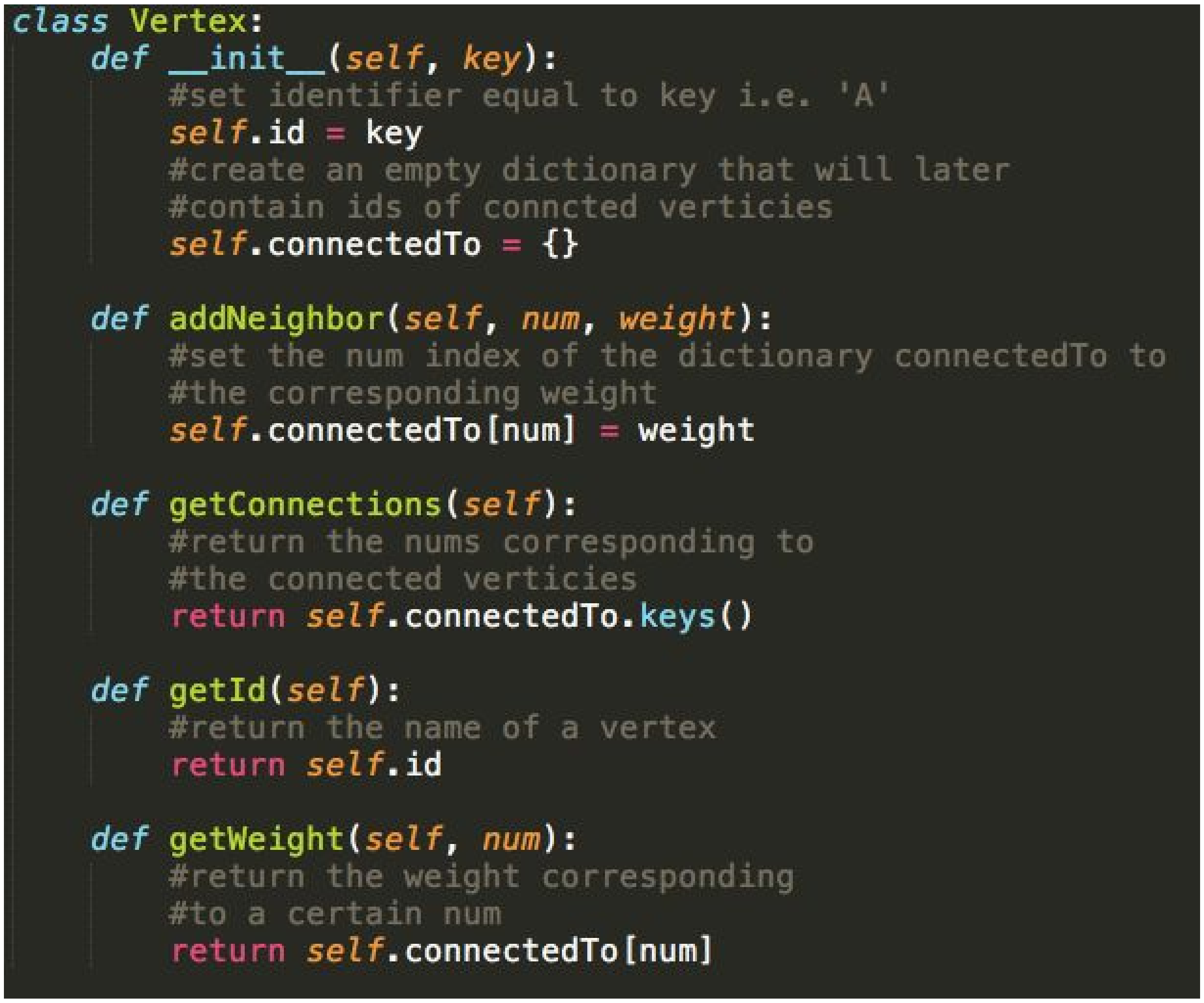


```python
    def buildHeap(self,alist):
        i = len(alist) // 2
        self.currentSize = len(alist)
        self.heapList = [0] + alist[:]
        while (i > 0):
            self.percDown(i)
            i = i - 1
```

Graphs:

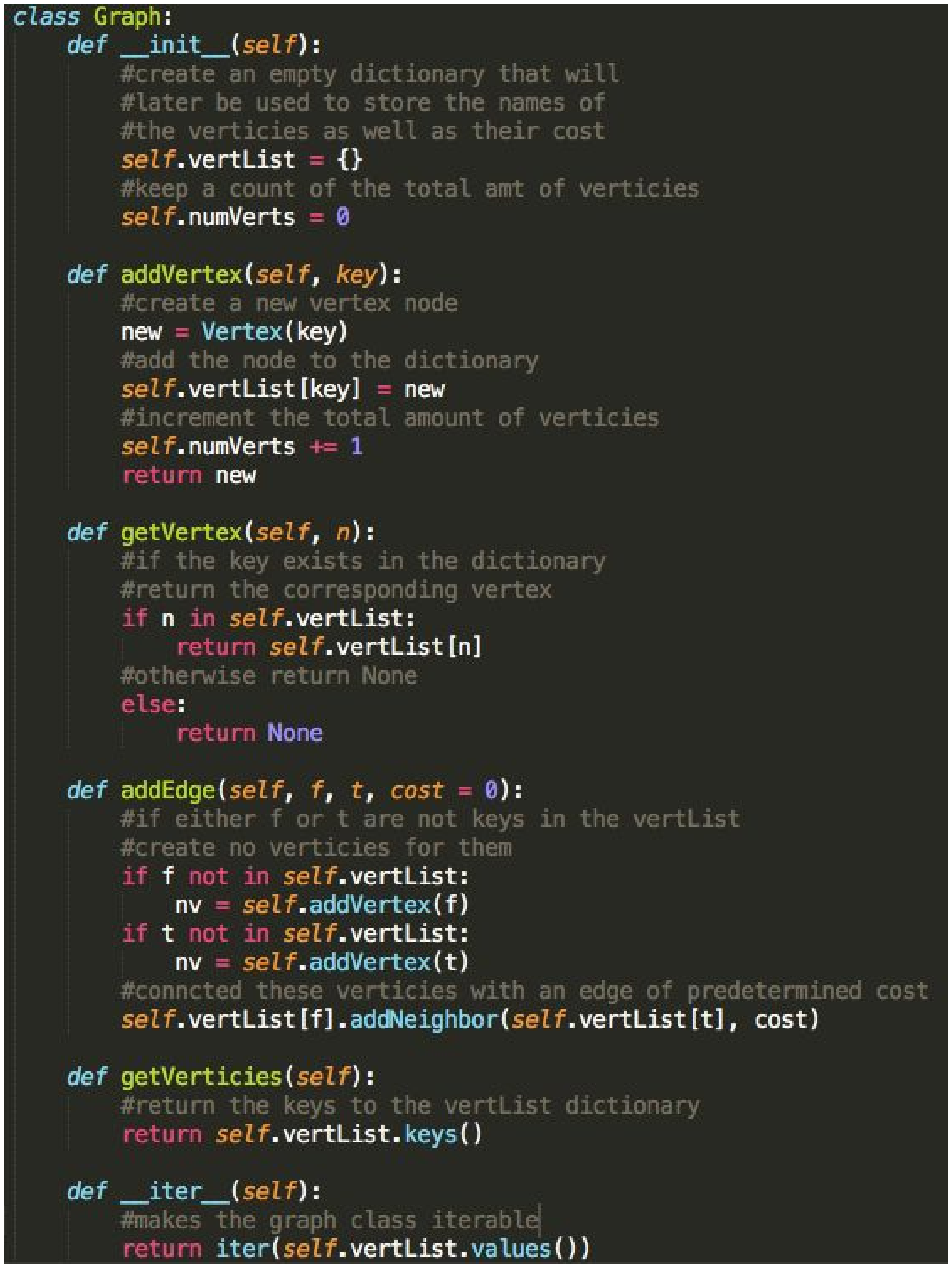
Each Vertex uses a dictionary to keep track of the vertices to which it is connected, and the weight of each edge. This dictionary is called connectedTo. The listing below shows the code for the Vertexclass. The constructor simply initializes the id, which will typically be a string, and the connectedTodictionary. The addNeighbor method is used add a connection from this vertex to another. The getConnections method returns all of the vertices in the adjacency list, as represented by the connectedTo instance variable. The getWeight method returns the weight of the edge from this vertex to the vertex passed as a parameter.

Src = http://www.interactivepython.org/courselib/static/pythonds/Graphs/Implementation.html

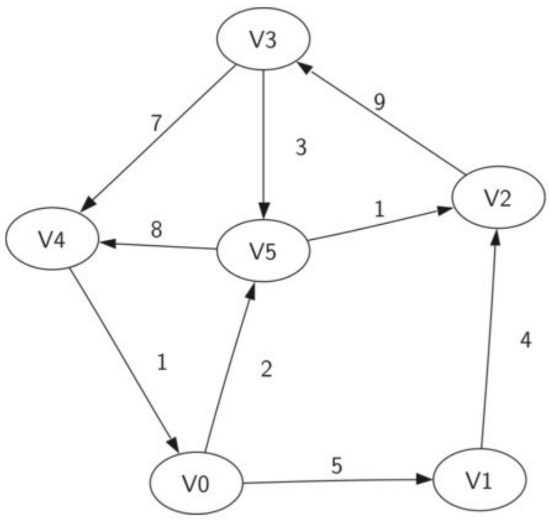


```python
class Vertex:
    def __init__(self, key):
        #set identifier equal to key i.e. 'A'
        self.id = key
        #create an empty dictionary that will later
        #contain ids of conncted verticies
        self.connectedTo = {}

    def addNeighbor(self, num, weight):
        #set the num index of the dictionary connectedTo to
        #the corresponding weight
        self.connectedTo[num] = weight

    def getConnections(self):
        #return the nums corresponding to
        #the connected verticies
        return self.connectedTo.keys()

    def getId(self):
        #return the name of a vertex
        return self.id

    def getWeight(self, num):
        #return the weight corresponding
        #to a certain num
        return self.connectedTo[num]
```

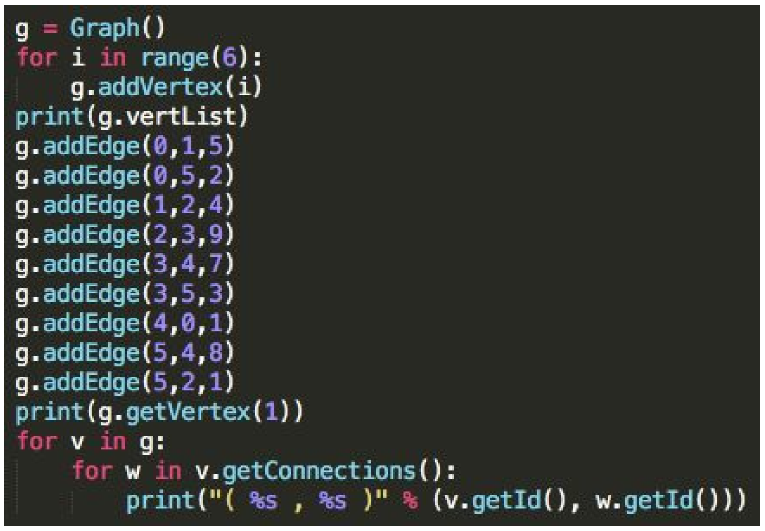
The Graph class, shown in the next listing, contains a dictionary that maps vertex names to vertex objects. Graph also provides methods for adding vertices to a graph and connecting one vertex to another. The getVertices method returns the names of all of the vertices in the graph. In addition, we have implemented the __iter__method to make it easy to iterate over all the vertex objects in a particular graph. Together, the two methods allow you to iterate over the vertices in a graph by name, or by the objects themselves.



```python
class Graph:
    def __init__(self):
        #create an empty dictionary that will
        #later be used to store the names of
        #the verticies as well as their cost
        self.vertList = {}
        #keep a count of the total amt of verticies
        self.numVerts = 0

    def addVertex(self, key):
        #create a new vertex node
        new = Vertex(key)
        #add the node to the dictionary
        self.vertList[key] = new
        #increment the total amount of verticies
        self.numVerts += 1
        return new

    def getVertex(self, n):
        #if the key exists in the dictionary
        #return the corresponding vertex
        if n in self.vertList:
            return self.vertList[n]
        #otherwise return None
        else:
            return None

    def addEdge(self, f, t, cost = 0):
        #if either f or t are not keys in the vertList
        #create no verticies for them
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
        #conncted these verticies with an edge of predetermined cost
        self.vertList[f].addNeighbor(self.vertList[t], cost)

    def getVerticies(self):
        #return the keys to the vertList dictionary
        return self.vertList.keys()

    def __iter__(self):
        #makes the graph class iterable
        return iter(self.vertList.values())
```

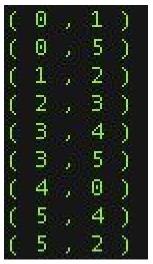


Code to create this graph:



```
g = Graph()
for i in range(6):
    g.addVertex(i)
print(g.vertList)
g.addEdge(0,1,5)
g.addEdge(0,5,2)
g.addEdge(1,2,4)
g.addEdge(2,3,9)
g.addEdge(3,4,7)
g.addEdge(3,5,3)
g.addEdge(4,0,1)
g.addEdge(5,4,8)
g.addEdge(5,2,1)
print(g.getVertex(1))
for v in g:
    for w in v.getConnections():
        print("( %s , %s )" % (v.getId(), w.getId()))
```

Output (printing of vertList & getVertex were omitted):



```
( 0 , 1 )
( 0 , 5 )
( 1 , 2 )
( 2 , 3 )
( 3 , 4 )
( 3 , 5 )
( 4 , 0 )
( 5 , 4 )
( 5 , 2 )
```