Data Structures and Algorithms

algorithm is simply a procedure or formula for solving problems

some problems have names as well as some procedures being common enough that the algorithm

associated with it also has a name

**Big-O notation**

Big-O notation describes how quickly runtime will grow relative to the input as the input get arbitrarily large

Different types of Big-O functions

1 constant

log(n) logartithmic

n linear

nlog(n) Log linear

n^2 Quadratic

n^3 cubic

2^n Exponential

**Constant:**

When using constant type function of Big –O , for any number of input the value returned by the function is a single value eg below

**def** func\_constant(values):

*'''*

*Prints first item in a list of values.*

*'''*

print values[0]

func\_constant([1,2,3])

Ouput 1 , because we are accessing the first value in the list , whatever may be the length of the since we are accessing the first element in the list the values will be always constant

Note how this function is constant because regardless of the list size, the function will only ever take a constant step size, in this case 1, printing the first value from a list. so we can see here that an input list of 100 values will print just 1 item, a list of 10,000 values will print just 1 item, and a list of n values will print just 1 item!

**Linear**

When using linear type of function , for n number of inputs the function runs n times , for all the values of n.

**def** func\_lin(lst):

*'''*

*Takes in list and prints out all values*

*'''*

**for** val **in** lst:

print val

func\_lin([1,2,3])

Output: 1

2

3

This function runs in O(n) (linear time). This means that the number of operations taking place scales linearly with n, so we can see here that an input list of 100 values will print 100 times, a list of 10,000 values will print 10,000 times, and a list of n values will print n times.

**Quadratic:**

While using quadratic the values in the list are in order of n \* n i.e , we need to perform two looping in order to get the get the values twice .

**def** func\_quad(lst):

*'''*

*Prints pairs for every item in list.*

*'''*

**for** item\_1 **in** lst:

**for** item\_2 **in** lst:

print item\_1,item\_2

lst = [0, 1, 2, 3]

func\_quad(lst)

Output: 0 1 in the next line 0 1 in the next line 0 2 in the next line 0 3 and so on upto 3 3.

Note how we now have two loops, one nested inside another. This means that for a list of n items, we will have to perform n operations for every item in the list! This means in total, we will perform n times n assignments, or n^2. So a list of 10 items will have 10^2, or 100 operations. You can see how dangerous this can get for very large inputs! This is why Big-O is so important to be aware of!

**Calculating Scale of Big-O**

In this section we will discuss how insignificant terms drop out of Big-O notation.

When it comes to Big O notation we only care about the most significant terms, remember as the input grows larger only the fastest growing terms will matter. If you've taken a calculus class before, this will reminf you of taking limits towards infinity. Let's see an example of how to drop constants:

**def** print\_once(lst):

*'''*

*Prints all items once*

*'''*

**for** val **in** lst:

print val

print\_once(lst)

0

1

2

3

The print\_once() function is O(n) since it will scale linearly with the input. What about the next example?

**def** print\_3(lst):

*'''*

*Prints all items three times*

*'''*

**for** val **in** lst:

print val

**for** val **in** lst:

print val

**for** val **in** lst:

print val

print\_3(lst)

0

1

2

3

0

1

2

3

0

1

2

3

We can see that the first function will print O(n) items and the second will print O(3n) items. However for n going to inifinity the constant can be dropped, since it will not have a large effect, so both functions are O(n).

Let's see a more complex example of this:

**def** comp(lst):

*'''*

*This function prints the first item O(1)*

*Then is prints the first 1/2 of the list O(n/2)*

*Then prints a string 10 times O(10)*

*'''*

print lst[0]

midpoint = len(lst)/2

**for** val **in** lst[:midpoint]:

print val

**for** x **in** range(10):

print 'number'

lst = [1,2,3,4,5,6,7,8,9,10]

comp(lst)

1

1

2

3

4

5

number

number

number

number

number

number

number

number

number

number

So let's break down the operations here. We can combine each operation to get the total Big-O of the function:

O(1+n/2+10)O(1+n/2+10)

We can see that as n grows larger the 1 and 10 terms become insignificant and the 1/2 term multiplied against n will also not have much of an effect as n goes towards infinity. This means the function is simply O(n)!

**Worst Case vs Best Case**

Many times we are only concerned with the worst possible case of an algorithm, but in an interview setting its important to keep in mind that worst case and best case scenarios may be completely different Big-O times. For example, consider the following function:

**def** matcher(lst,match):

*'''*

*Given a list lst, return a boolean indicating if match item is in the list*

*'''*

**for** item **in** lst:

**if** item == match:

**return** **True**

**return** **False**

lst= [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

matcher(lst,1)

True

matcher(lst,11)

False

Note that in the first scenario, the best case was actually O(1), since the match was found at the first element. In the case where there is no match, every element must be checked, this results in a worst case time of O(n). Later on we will also discuss average case time.

Finally let's introduce the concept of space complexity.

**Space Complexity**

Many times we are also concerned with how much memory/space an algorithm uses. The notation of space complexity is the same, but instead of checking the time of operations, we check the size of the allocation of memory.

Let's see a few examples:

**def** printer(n=10):

*'''*

*Prints "hello world!" n times*

*'''*

**for** x **in** range(n):

print 'Hello World!'

printer()

Hello World!

Hello World!

Hello World!

Hello World!

Hello World!

Hello World!

Hello World!

Hello World!

Hello World!

Hello World!

Note how we only assign the 'hello world!' variable once, not every time we print. So the algorithm has O(1) **space** complexity and an O(n) **time** complexity.

Let's see an example of O(n) **space** complexity:

**def** create\_list(n):

new\_list = []

**for** num **in** range(n):

new\_list.append('new')

**return** new\_list

print create\_list(5)

['new', 'new', 'new', 'new', 'new']

Note how the size of the new\_list object scales with the input **n**, this shows that it is an O(n) algorithm with regards to **space** complexity.

Thats it for this lecture, before continuing on, make sure to complete the homework assignment below:

Dynamic Arrays:

The dynamic array is one dynamically doubles the size when the size of the array increases. We can construct the dynamic array, but it uses order O(n) methodology.

Eg of creating a dynamic array

import ctypes

import sys

class DynamicArray(object):

def \_\_init\_\_(self):

self.n =0

self.capacity = 1

self.A = self.make\_array(self.capacity)

def \_\_len\_\_(self):

return self.n

def \_\_getitem\_\_(self, k):

if not 0 <= k < self.n:

return IndexError('k is out of bounds!')

return self.A[k]

def append(self, ele):

if self.n == self.capacity:

self.\_resize(2 \* self.capacity)

""" The new capacity variable which is passed in the resize and makenew array method is the the value

passed in the resize function in append."""

self.A[self.n] = ele

self.n +=1

def \_resize(self, new\_cap):

B = self.make\_array(new\_cap)

for k in range(self.n):

B[k] =self.A[k]

self.A = B

self.capacity = new\_cap

def make\_array(self, new\_cap):

return (new\_cap \* ctypes.py\_object)()

arr = DynamicArray()

arr.append(1)

print(len(arr)) , Ouptut: we can get add elements to the array using the append function we can get the length and the index value of the array.

Amortization

By using this algorithm pattern we can show that performing a sequence of such append operations on a dynamic array is actually quite efficient by using with the Amortization we can optimize the dynamic array to O(1)

Stacks Deques and Queue:

# Implementation of Stack

## Stack Attributes and Methods

Before we implement our own Stack class, let's review the properties and methods of a Stack.

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

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

# Queues Overview

In this lecture we will get an overview of what a Queue is, in the next lecture we will implement our own Queue class.

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

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

The simplest example of a queue is the typical line that we all participate in from time to time. We wait in a line for a movie, we wait in the check-out line at a grocery store, and we wait in the cafeteria line. The first person in that line is also the first person to get serviced/helped.

Note how we have two terms here, **Enqueue** and **Dequeue**. The enqueue term describes when we add a new item to the rear of the queue. The dequeue term describes removing the front item from the queue.

# Deques Overview

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

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

Singly Linked Lists

A singly linked list, in its simplest form, is a collection of nodes that collectively form a linear sequence.

Each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list.

Doubly Linked Lists

In a doubly linked list, we define a linked list in which each node keeps an explicit reference to the node before it and a reference to the node after it.

These lists allow a greater variety of O(1)-time update operations, including insertions and deletions.

We continue to use the term “next” for the reference to the node that follows another.

We have a new term “prev” for the reference to the node that precedes it.

## Recursion

There are two main instances of recursion. The first is when recursion is used as a technique in which a function makes one or more calls to itself. The second is when a data structure uses smaller instances of the exact same type of data structure when it represents itself. Both of these instances are use cases of recursion.

Recursion actually occurs in the real world, such as fractal patterns seen in plants!

## Trees

## In a list of lists tree, we will store the value of the root node as the first element of the list.

## The second element of the list will itself be a list that represents the left subtree.

## The third element of the list will be another list that represents the right subtree.

## ../_images/smalltree.png

## 

## Nodes and References implementation

## In this case we will define a class that has attributes for the root value, as well as the left and right subtrees.

## Since this representation more closely follows the object-oriented programming paradigm, we will continue to use this representation for the remainder of this section

## image

**Tree Reversal**

There are three commonly used patterns to visit all the nodes in a tree.

The difference between these patterns is the order in which each node is visited (a “traversal”)

The three traversals we will look at are called preorder, inorder, and postorder

**PreOrder**

* In a preorder traversal, we visit the root node first, then recursively do a preorder traversal of the left subtree, followed by a recursive preorder traversal of the right subtree.

**Inorder**

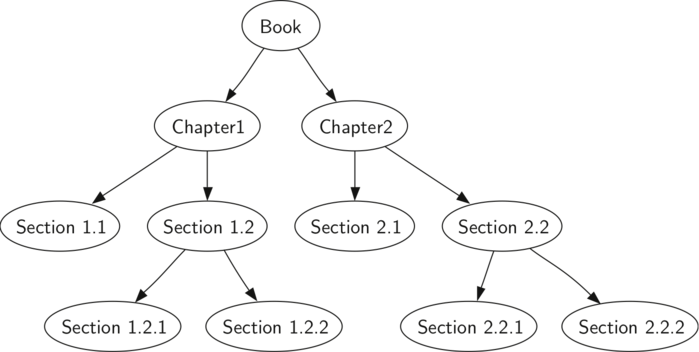
* In an inorder traversal, we recursively do an inorder traversal on the left subtree, visit the root node, and finally do a recursive inorder traversal of the right subtree.

**Postorder**

* In a postorder traversal, we recursively do a postorder traversal of the left subtree and the right subtree followed by a visit to the root node

**Example**

* **As an example of a tree to traverse, we will represent this book as a tree. The book is the root of the tree, and each chapter is a child of the root.**

****

Suppose that you wanted to read this book from front to back.

The preorder traversal gives you exactly that ordering.

Starting at the root of the tree (the Book node) we will follow the preorder traversal instructions.

We recursively call preorder on the left child, in this case Chapter1.

We again recursively call **preorder** on the left child to get to Section 1.1.

Since Section 1.1 has no children, we do not make any additional recursive calls.

When we are finished with Section 1.1, we move up the tree to Chapter 1.

At this point we still need to visit the right subtree of Chapter 1, which is Section 1.2.

As before we visit the left subtree, which brings us to Section 1.2.1, then we visit the node for Section 1.2.2.

With Section 1.2 finished, we return to Chapter 1.

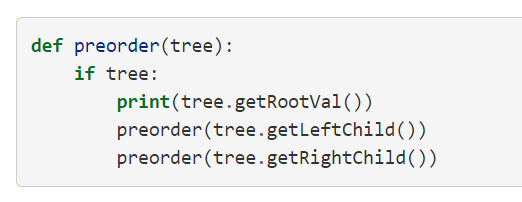
Then we return to the Book node and follow the same procedure for Chapter 2.

**Preorder – Recursive Implementation**

Base case is simply to check if the tree exists.

If the tree parameter is None, then the function returns without taking any action

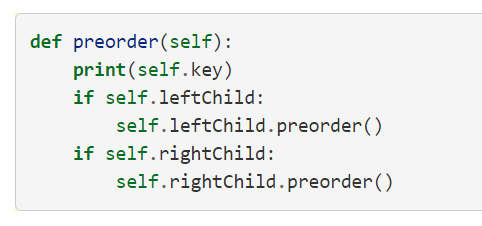
.



**Preorder – Method Implementation**

**We can also implement preorder as a method of the BinaryTree class.**

**The internal method must check for the existence of the left and the right children before making the recursive call to preorder.**

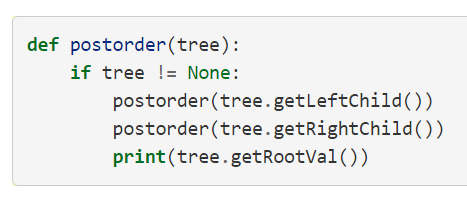
****

**Preorder – Best Implementation**

* Implementing preorder as an external function is probably better in this case.
* The reason is that you very rarely want to just traverse the tree.
* In most cases you are going to want to accomplish something else while using one of the basic traversal patterns.
* We will write the rest of the traversals as external functions.

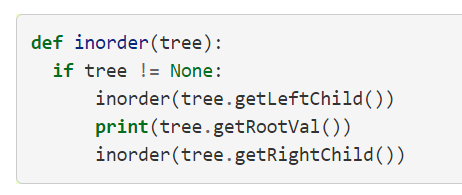
**Postorder**

* The algorithm for the postorder traversal is nearly identical to preorder except that we move the call to print to the end of the function.



**Inorder**

* **In the inorder traversal we visit the left subtree, followed by the root, and finally the right subtree.**
* **Notice that in all three of the traversal functions we are simply changing the position of the print statement with respect to the two recursive function calls.**

****

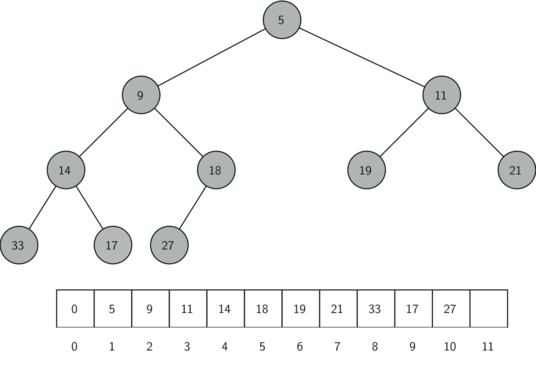
**Priority Queues with Binary Heaps**

* One important variation of a queue is called a priority queue.
* A priority queue acts like a queue in that you dequeue an item by removing it from the front.
* However, in a priority queue the logical order of items inside a queue is determined by their priority.
* The highest priority items are at the front of the queue and the lowest priority items are at the back.
* When you enqueue an item on a priority queue, the new item may move all the way to the front.
* The classic way to implement a priority queue is using a data structure called a **binary heap**.
* A binary heap will allow us both enqueue and dequeue items in O(log**n**).
* The binary heap has two common variations: the **min heap**, in which the smallest key is always at the front, and the **max heap**, in which the largest key value is always at the front.
* In this section we will implement the min heap.

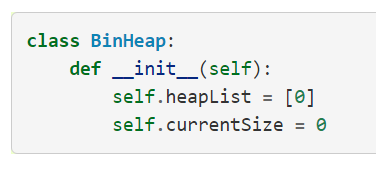
**Binary Heap implementation**

* **In order to make our heap work efficiently, we will take advantage of the logarithmic nature of the binary tree to represent our heap.**
* **In order to guarantee logarithmic performance, we must keep our tree balanced.**
* **A balanced binary tree has roughly the same number of nodes in the left and right subtrees of the root.**
* **In our heap implementation we keep the tree balanced by creating a complete binary tree.**
* **A complete binary tree is a tree in which each level has all of its nodes.**

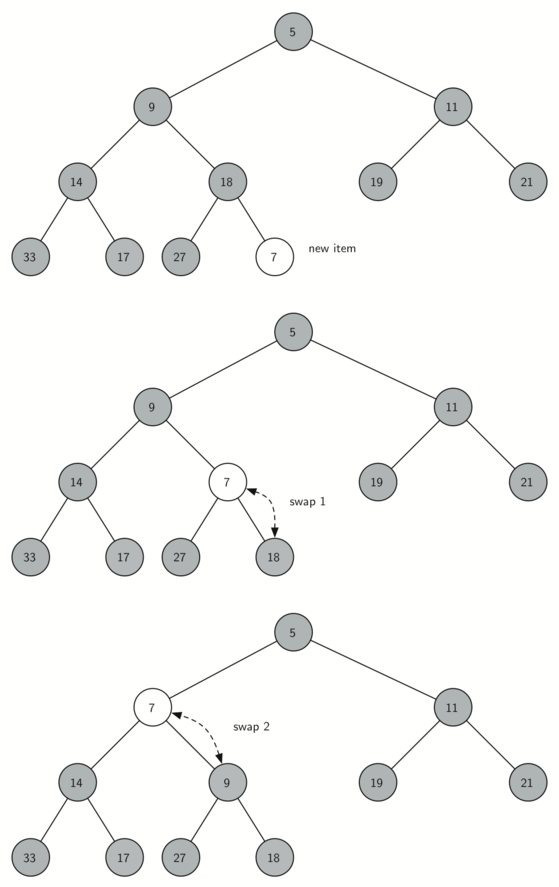
**Start off with our list representation code**

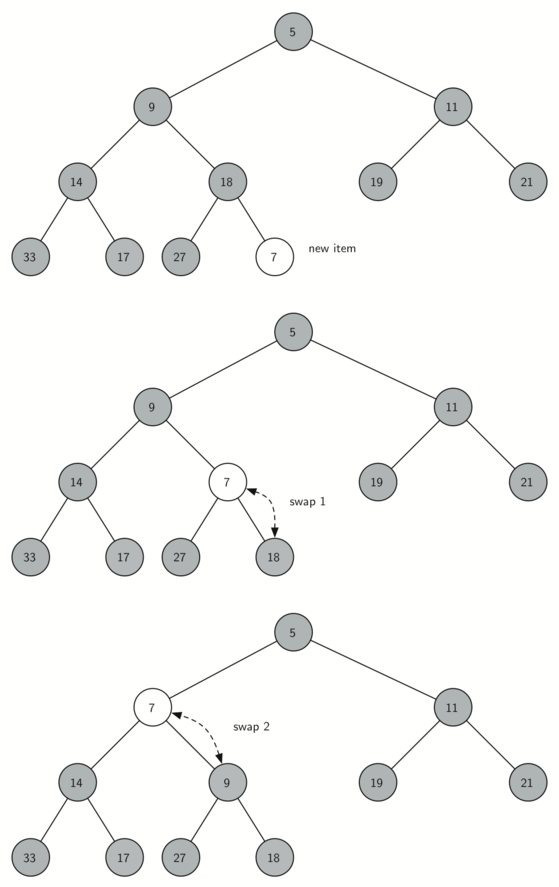


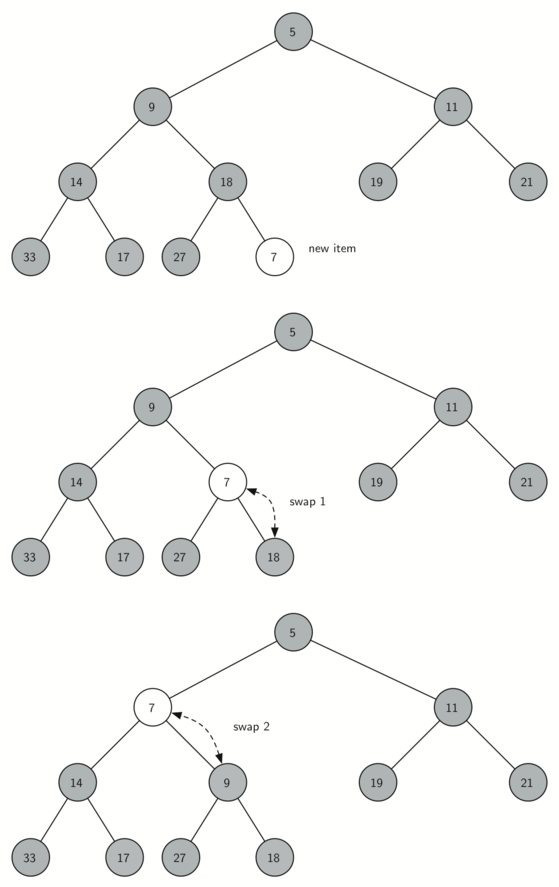
**Start off with our list representation code**



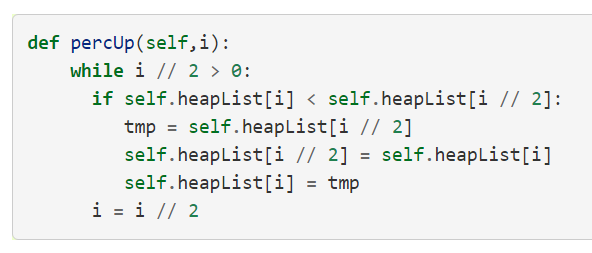
* The next method we will implement is insert. The easiest, and most efficient, way to add an item to a list is to simply append the item to the end of the list.
* The good news about appending is that it guarantees that we will maintain the complete tree property.
* The bad news about appending is that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

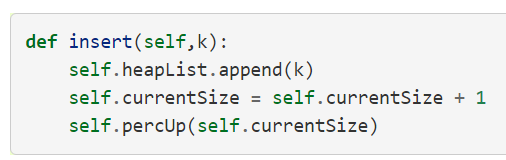




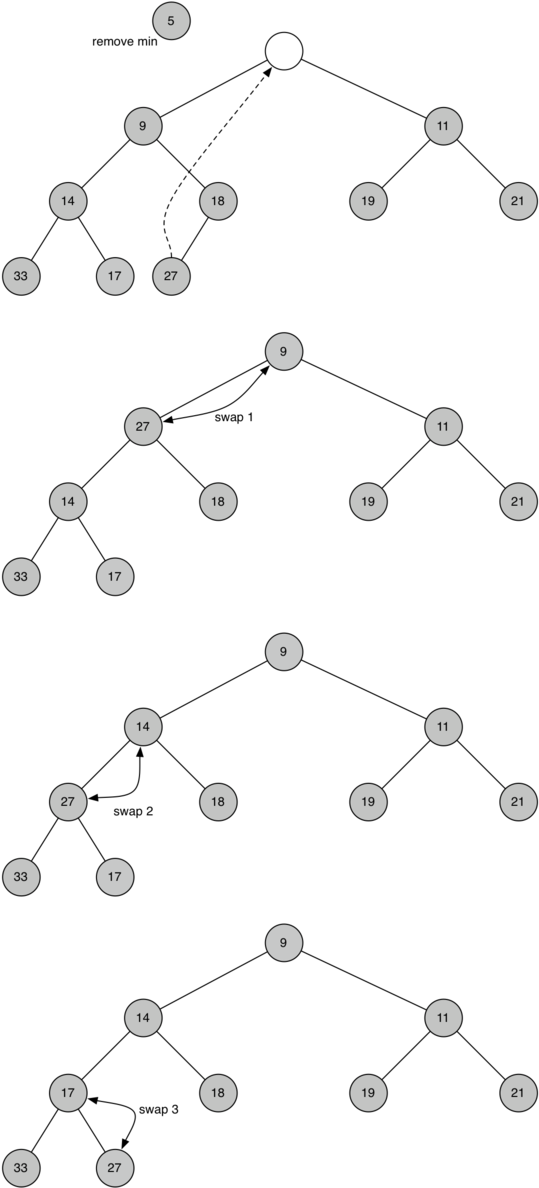


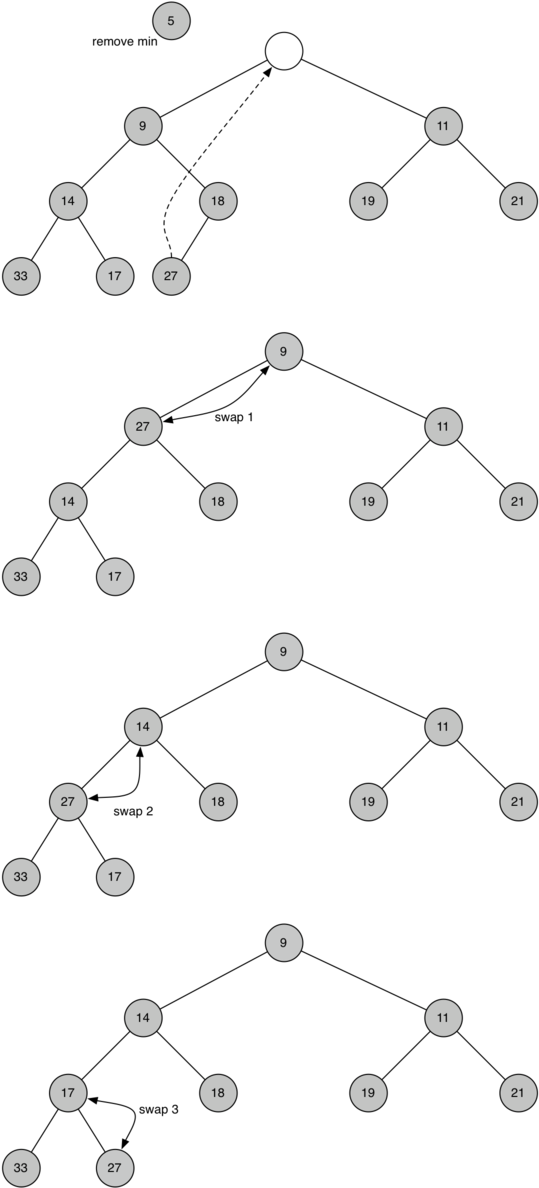
* Notice that when we percolate an item up, we are restoring the heap property between the newly added item and the parent.
* We are also preserving the heap property for any siblings.
* Of course, if the newly added item is very small, we may still need to swap it up another level.
* In fact, we may need to keep swapping until we get to the top of the tree.

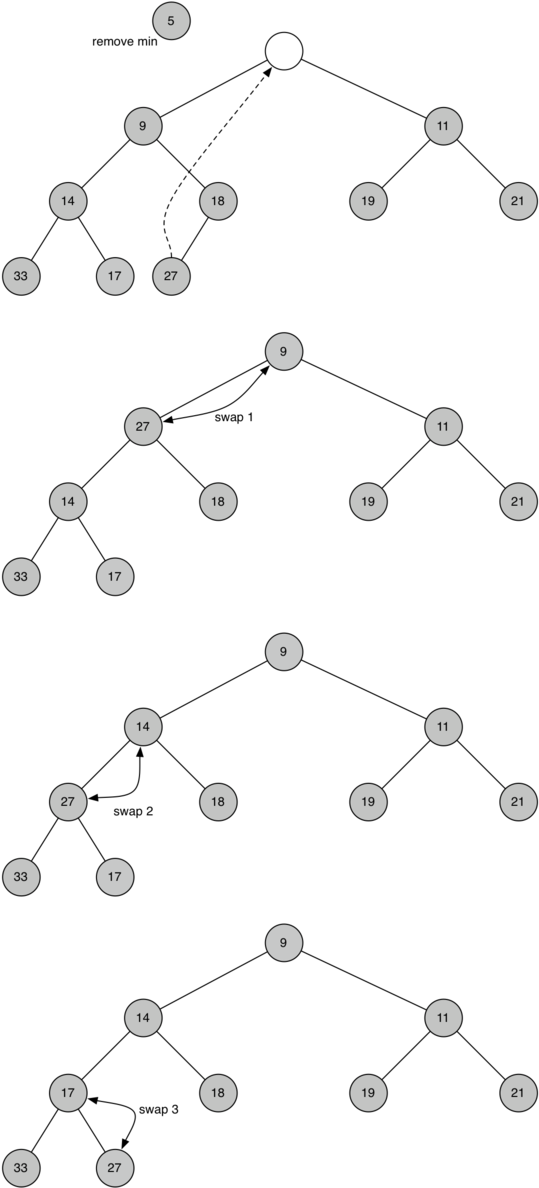


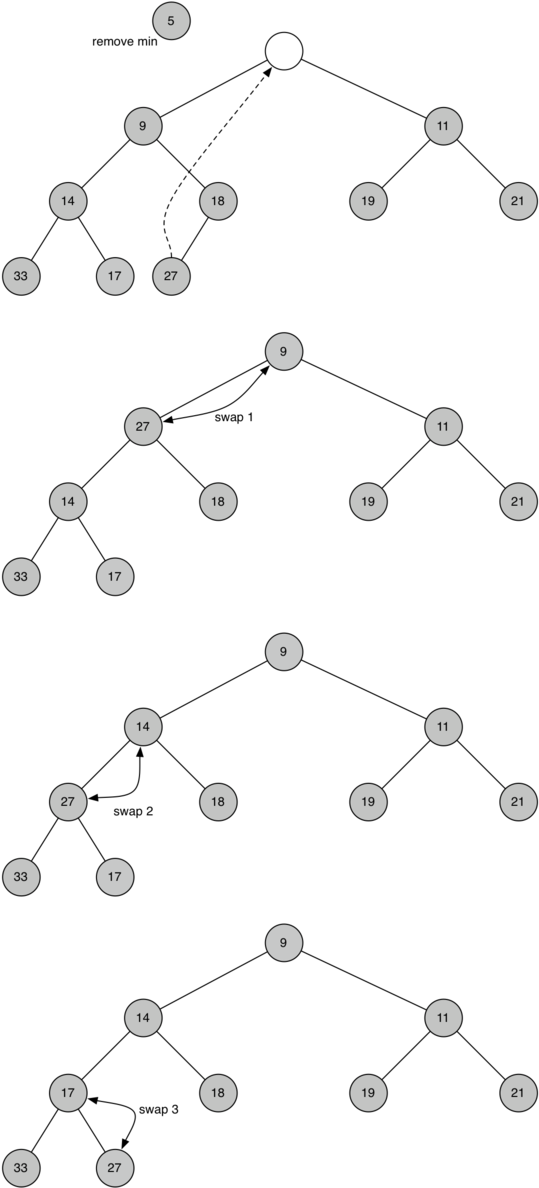
****

* With the insert method properly defined, we can now look at the delMin method.
* Since the heap property requires that the root of the tree be the smallest item in the tree, finding the minimum item is easy.
* The hard part of delMin is restoring full compliance with the heap structure and heap order properties after the root has been removed.
* We can restore our heap in two steps.
* First, we will restore the root item by taking the last item in the list and moving it to the root position.
* Moving the last item maintains our heap structure property.
* However, we have probably destroyed the heap order property of our binary heap.
* Second, we will restore the heap order property by pushing the new root node down the tree to its proper position.
* Series of swaps needed to move the new root node to its proper position in the heap

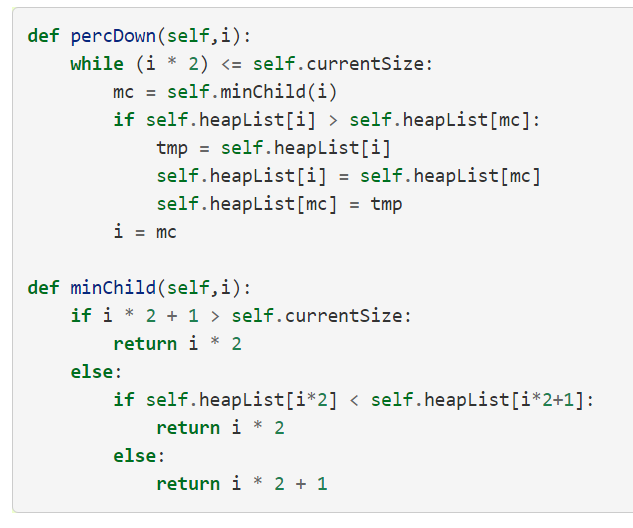




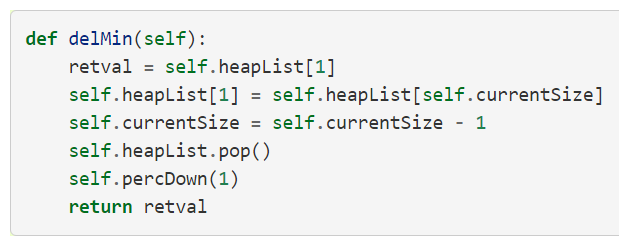
****

****

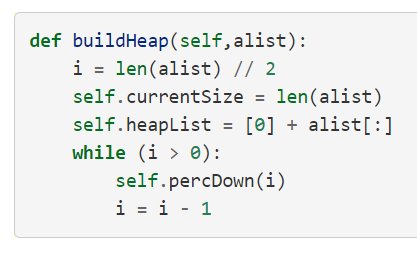
* In order to maintain the heap order property, all we need to do is swap the root with its smallest child less than the root.
* After the initial swap, we may repeat the swapping process with a node and its children until the node is swapped into a position on the tree where it is already less than both children.
* Code for percolating a node down the tree is found in the **percDown** and **minChild**

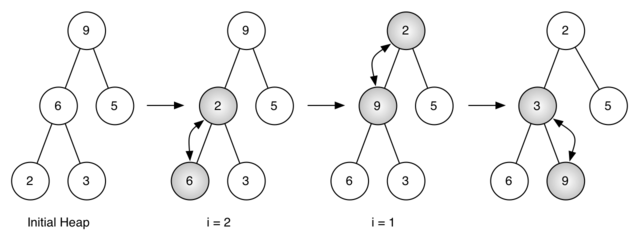


Code for **delMin**



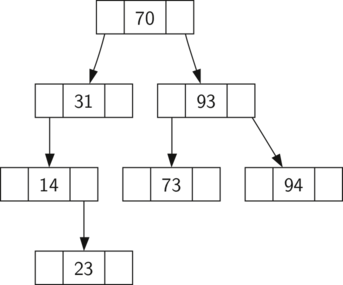
* To finish our discussion of binary heaps, we will look at a method to build an entire heap from a list of keys.
* The first method you might think of may be like the following.
* Given a list of keys, you could easily build a heap by inserting each key one at a time.
* Since you are starting with a list of one item, the list is sorted and you could use binary search to find the right position to insert the next key at a cost of approximately O(log⁡n) operations.
* However, remember that inserting an item in the middle of the list may require **O(n)** operations to shift the rest of the list over to make room for the new key.
* Therefore, to insert n keys into the heap would require a total of **O(nlog⁡n)**operations.
* However, if we start with an entire list then we can build the whole heap in **O(n)** operations.



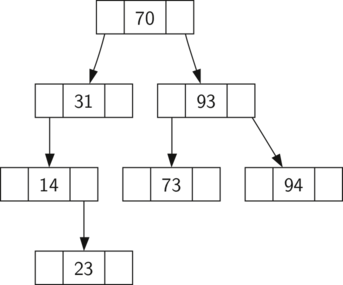


**Implementation of Binary search trees**

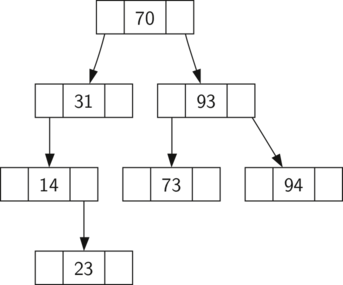
* A binary search tree relies on the property that keys that are less than the parent are found in the left subtree, and keys that are greater than the parent are found in the right subtree.
* We will call this the bst property.
* As we implement the Map interface as described above, the bst property will guide our implementation.

****

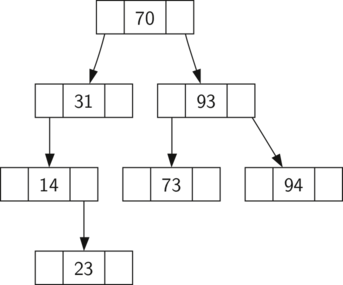
* Notice that the property holds for each parent and child.
* All of the keys in the left subtree are less than the key in the root.
* All of the keys in the right subtree are greater than the root.
* Now that you know what a binary search tree is, we will look at how a binary search tree is constructed.
* The search tree in the figure represents the nodes that exist after we have inserted the following keys in the order shown: 70,31,93,94,14,23,73



* Since 70 was the first key inserted into the tree, it is the root.
* Next, 31 is less than 70, so it becomes the left child of 70.
* Next, 93 is greater than 70, so it becomes the right child of 70.
* Now we have two levels of the tree filled, so the next key is going to be the left or right child of either 31 or 93.



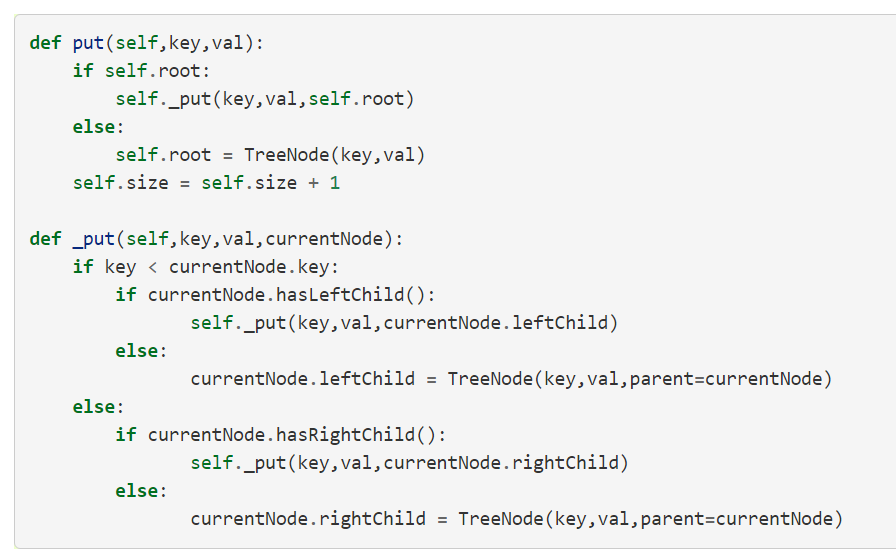
* Since 94 is greater than 70 and 93, it becomes the right child of 93.
* Similarly 14 is less than 70 and 31, so it becomes the left child of 31. 23 is also less than 31, so it must be in the left subtree of 31.
* However, it is greater than 14, so it becomes the right child of 14.

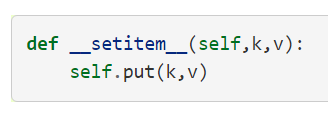
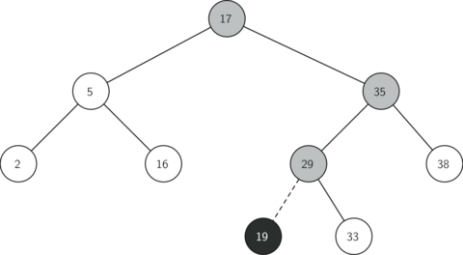


* To implement the binary search tree, we will use the nodes and references approach similar to the one we used to implement the linked list, and the expression tree.
* However, because we must be able create and work with a binary search tree that is empty, our implementation will use two classes.
* The first class we will call **BinarySearchTree**, and the second class we will call **TreeNode**.
* The **BinarySearchTree** class has a reference to the **TreeNode** that is the root of the binary search tree.
* In most cases the external methods defined in the outer class simply check to see if the tree is empty.
* If there are nodes in the tree, the request is just passed on to a private method defined in the **BinarySearchTree** class that takes the root as a parameter.
* In the case where the tree is empty or we want to delete the key at the root of the tree, we must take special action.

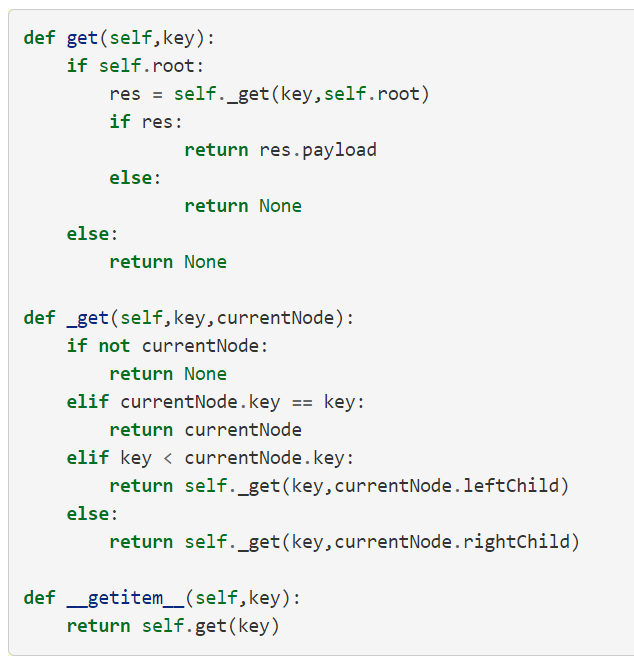


* Let’s jump to the Notebook to check out the full **TreeNode** class!
* Now that we have the **BinarySearchTree** shell and the **TreeNode** it is time to write the put method that will allow us to build our binary search tree.
* The put method is a method of the **BinarySearchTree** class.
* Now that we have the **BinarySearchTree** shell and the **TreeNode** it is time to write the put method that will allow us to build our binary search tree.
* The put method is a method of the **BinarySearchTree** class.
* If a root node is already in place then put calls the private, recursive, helper function**put** to search the tree according to the following algorithm…
* Starting at the root of the tree, search the binary tree comparing the new key to the key in the current node.
* If the new key is less than the current node, search the left subtree. If the new key is greater than the current node, search the right subtree.
* When there is no left (or right) child to search, we have found the position in the tree where the new node should be installed.
* To add a node to the tree, create a new TreeNode object and insert the object at the point discovered in the previous step.

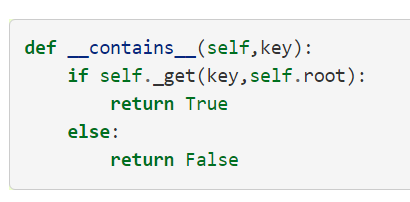


* Once the tree is constructed, the next task is to implement the retrieval of a value for a given key.
* The get method is even easier than the put method because it simply searches the tree recursively until it gets to a non-matching leaf node or finds a matching key.
* When a matching key is found, the value stored in the payload of the node is returned.



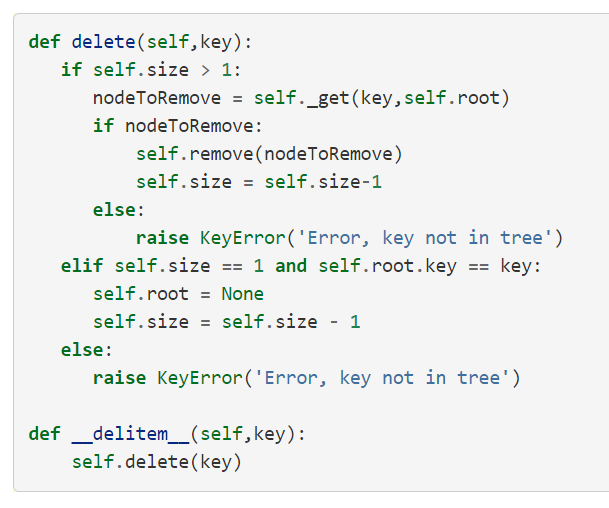
* Using get, we can implement the in operation by writing a **\_\_contains\_\_** method for the **BinarySearchTree**.
* The **\_\_contains\_\_** method will simply call get and return True if get returns a value, or False if it returns None.



* Finally, we turn our attention to the most challenging method in the binary search tree, the deletion of a key.
* The first task is to find the node to delete by searching the tree.
* If the tree has more than one node we search using the **\_get** method to find the **TreeNode** that needs to be removed.

**Deleting Nodes**

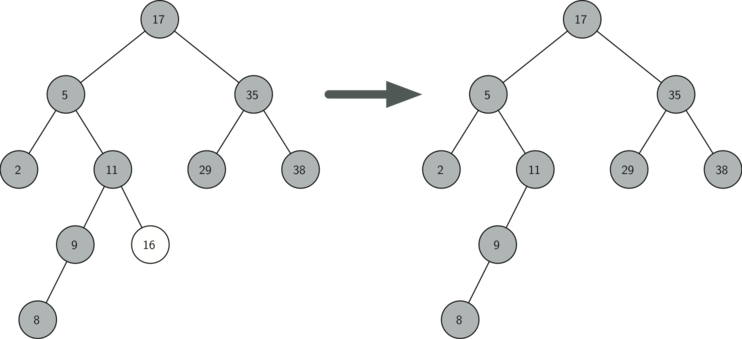
* Finally, we turn our attention to the most challenging method in the binary search tree, the deletion of a key.
* The first task is to find the node to delete by searching the tree.
* If the tree has more than one node we search using the \_get method to find the TreeNode that needs to be removed.
* If the tree only has a single node, that means we are removing the root of the tree, but we still must check to make sure the key of the root matches the key that is to be deleted.
* In either case if the key is not found the del operator raises an error.



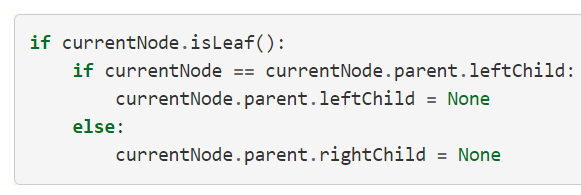
* Once we’ve found the node containing the key we want to delete, there are three cases that we must consider:

**Deleting Nodes Case 1**

* The node to be deleted has no children

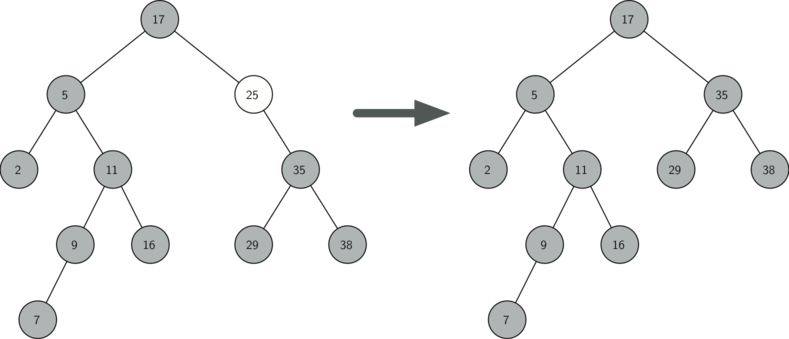


* The first case is straightforward
* If the current node has no children all we need to do is delete the node and remove the reference to this node in the parent.



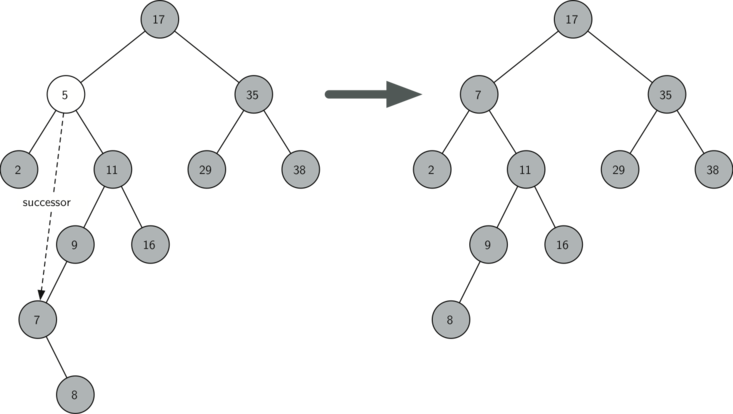
**Deleting Nodes Case 2**

* The node to be deleted has only one child

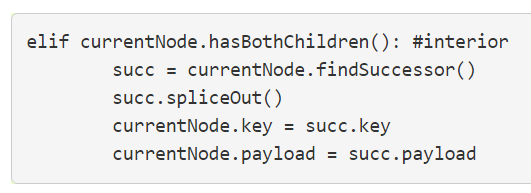
****

* The second case is only slightly more complicated.
* If a node has only a single child, then we can simply promote the child to take the place of its parent.
* Let’s jump to the code to discuss this second case!

**Deleting Nodes Case 3**

****

* The third case is the most difficult case to handle
* If a node has two children, then it is unlikely that we can simply promote one of them to take the node’s place.
* We can, however, search the tree for a node that can be used to replace the one scheduled for deletion.
* What we need is a node that will preserve the binary search tree relationships for both of the existing left and right subtrees.
* The node that will do this is the node that has the next-largest key in the tree. We call this node the **successor**, and we will look at a way to find the successor shortly.
* The successor is guaranteed to have no more than one child, so we know how to remove it using the two cases for deletion that we have already implemented.
* Once the successor has been removed, we simply put it in the tree in place of the node to be deleted.
* Notice that we make use of the helper methods **findSuccessor** and **findMin** to find the successor.
* To remove the successor, we make use of the method **spliceOut**.
* The reason we use **spliceOut** is that it goes directly to the node we want to splice out and makes the right changes.

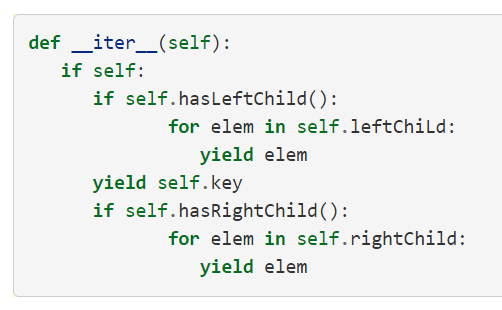


**findSuccessor**

* Let’s jump to the notebook again to discuss the findSuccessor method**.**

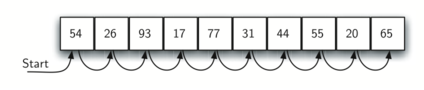
One last method!

* We need to look at one last interface method for the binary search tree.
* Suppose that we would like to simply iterate over all the keys in the tree in order.
* This is definitely something we have done with dictionaries, so why not trees?
* You already know how to traverse a binary tree in order, using the inorder traversal algorithm.
* However, writing an iterator requires a bit more work, since an iterator should return only one node each time the iterator is called.
* The \_\_iter\_\_ method

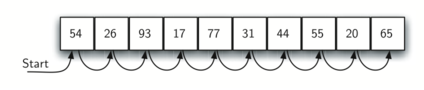


**Sequential Search**

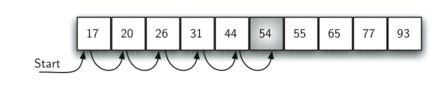
* Basic searching technique, sequentially go through the data structure, comparing elements as you go along.
* For example, on an unordered list searching for the element 50:

****

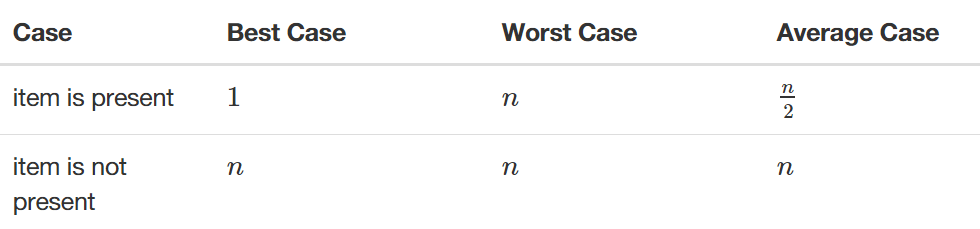
* 50 was not present, but we still had to check every element in the array.
* But what if it was ordered?



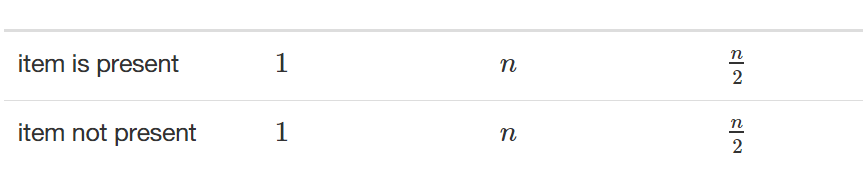
* If the list is ordered, we know we only have search until we reach an element which is a match or we reach an element which is greater than our search target.
* For example, searching for 50, we can stop here at 54.

****

**Unordered List Analysis**

****

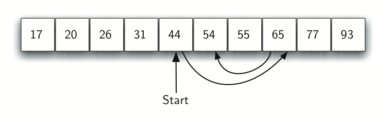
**Ordered List Analysis**

****

* Let’s do some basic implementations of Sequential Search!
* We’ll do both Ordered and Unordered implementations!

**Binary Search**

* We can take greater advantage of the ordered list!
* Instead of searching the list in sequence, a binary search will start by examining the middle item.
* A **binary search** will start by examining the middle item.
* If that item is the one we are searching for, we are done.
* If the item we are searching for is greater than the middle item, we know that the entire lower half of the list as well as the middle item can be eliminated from further consideration.
* The item, if it is in the list, must be in the upper half.
* We can then repeat the process with the upper half. Start at the middle item and compare it against what we are looking for.
* Again, we either find it or split the list in half, therefore eliminating another large part of our possible search space.
* For example, searching for **54** again:

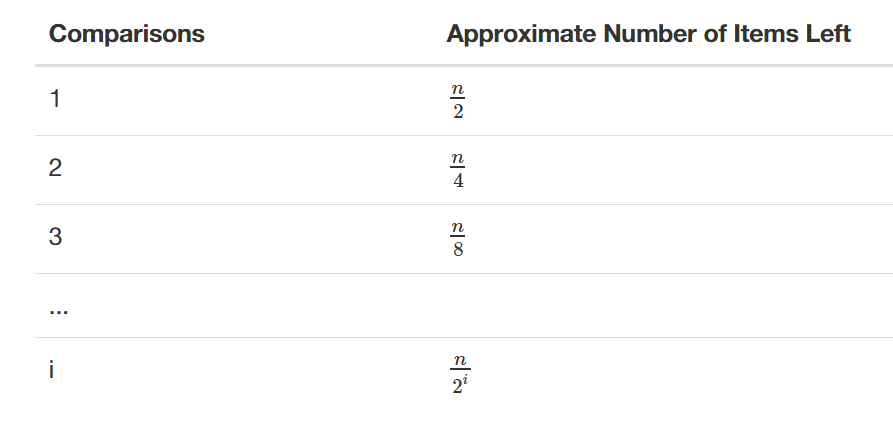


**Divide and Conquer**

* Binary search uses Divide and Conquer!
* We divide the problem into smaller pieces, solve the smaller pieces in some way, and then reassemble the whole problem to get the result.

**Binary Search Analysis**

* Each comparison eliminates about half of the remaining items from consideration.
* What is the maximum number of comparisons this algorithm will require to check the entire list?

****

**Hashing**

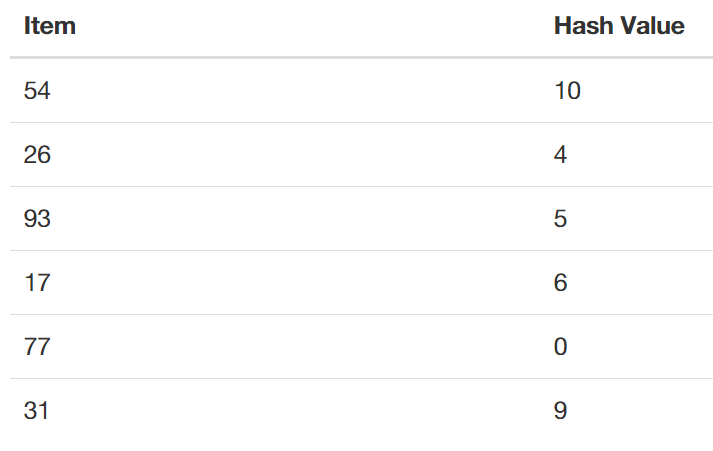
* We’ve seen how to improve search by knowing about structures beforehand.
* We can build a data structure that can be searched in O(1) time.
* This concept is referred to as hashing.
* A hash table is a collection of items which are stored in such a way as to make it easy to find them later.
* Each position of the hash table, slots, can hold an item and is named by an integer value starting at 0.
* For example, we will have a slot named 0, a slot named 1, a slot named 2, and so on.
* Initially, the hash table contains no items so every slot is empty.
* We can implement a hash table by using a list with each element initialized to the special Python value None.
* Here is an empty hash table with size m=11

****

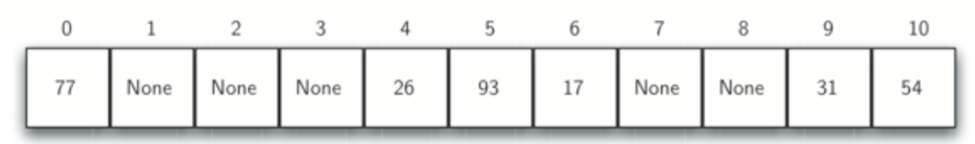
* The mapping between an item and the slot where that item belongs in the hash table is called the hash function.
* The hash function will take any item in the collection and return an integer in the range of slot names, between 0 and *m*-1.
* So how should we use hash functions to map items to slots?

**Hash Function – Remainder Method**

* One hash function we can use is the remainder method.
* When presented with an item, the hash function is the item divided by the table size, this is then its slot number.
* Let’s see an example!
* Assume that we have the set of integer items 54, 26, 93, 17, 77, and 31.
* We’ve preassigned an empty hash table of m=11
* Our remainder hash function then is: **h(item)=item%11**
* Let’s see the results as a table



* We’re now ready to occupy 6 out of the 11 slots.
* This is referred to as the **load factor**, and is commonly denoted by λ=.
* For this example, λ=6/11.
* Our hash table has now been loaded.



* When we want to search for an item, we simply use the hash function to compute the slot name for the item and then check the hash table to see if it is present.
* This searching operation is O(1), since a constant amount of time is required to compute the hash value and then index the hash table at that location.
* You might be thinking, what if you have two items that would result in the same location?
* For example 44%11 and 77%11 are the same.
* This is known as a **collision** (also known as a clash).
* We’ll learn how to deal with them later on.
* Let’s learn about hash functions in general!

**Hash Functions**

* A hash function that maps each item into a unique slot is referred to as a perfect hash function.
* Our goal is to create a hash function that minimizes the number of collisions, is easy to compute, and evenly distributes the items in the hash table.
* Let’s discuss a few techniques for this!

**Hash Functions – Folding Method**

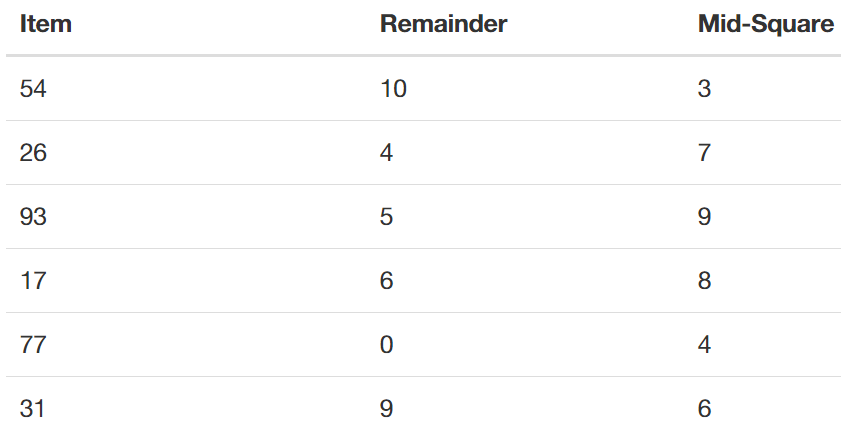
* The folding method for constructing hash functions begins by dividing the item into equal-size pieces (the last piece may not be of equal size).
* These pieces are then added together to give the resulting hash value.
* If our item was the phone number 436-555-4601
* We would take the digits and divide them into groups of 2 (43,65,55,46,01).
* After the addition, 43+65+55+46+01, we get 210.
* If we assume our hash table has 11 slots, then we need to perform the extra step of dividing by 11 and keeping the remainder.
* 210 % 11 is 1, so the phone number 436-555-4601 hashes to slot 1.

**Hash Functions – Mid Square Method**

* For the mid-square method we first square the item, and then extract some portion of the resulting digits.
* For example, if the item were 44, we would first compute 442=1,936.
* By extracting the middle two digits, 93, and performing the remainder step, we get 93%11 =5

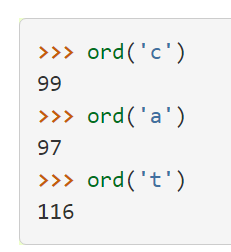
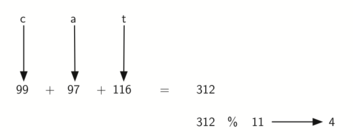
**Hash Functions**

* Comparison Table

****

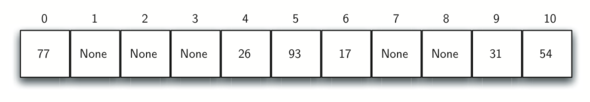
**Non-integer elements**

* We can also create hash functions for character-based items such as strings.
* The word “cat” can be thought of as a sequence of ordinal values.

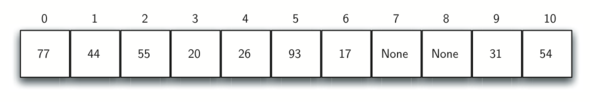
 

**Collision Resolution**

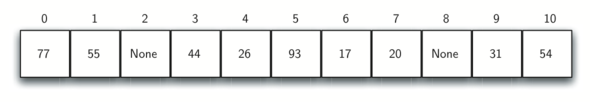
* One method for resolving collisions looks into the hash table and tries to find another open slot to hold the item that caused the collision.
* We could start at the original hash value position and then move in a sequential manner through the slots until we encounter the first slot that is empty.
* This collision resolution process is referred to as open addressing in that it tries to find the next open slot or address in the hash table.
* By systematically visiting each slot one at a time, we are performing an open addressing technique called **linear probing**.
* Consider the following table:
* What if we had to add 44,55, and 20?



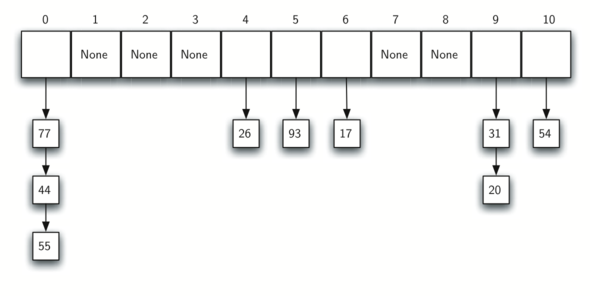
* With linear probing we keep moving down until we find an empty slot!



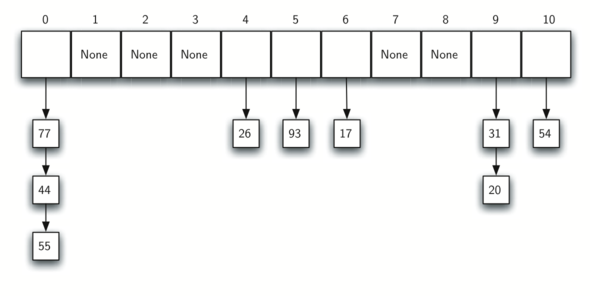
* One way to deal with clustering is to skip slots, thereby more evenly distributing the items that have caused collisions.



* The general name for this process of looking for another slot after a collision is **rehashing**.
* A variation of the linear probing idea is called **quadratic probing**.
* Instead of using a constant “skip” value, we use a rehash function that increments the hash value by 1, 3, 5, 7, 9, and so on.
* This means that if the first hash value is *h*, the successive values are h+1, h+4, h+9, h+16, and so on.
* An alternative method for handling the collision problem is to allow each slot to hold a reference to a collection (or chain) of items.
* Chaining allows many items to exist at the same location in the hash table.
* When collisions happen, the item is still placed in the proper slot of the hash table**.**

****

* As more and more items hash to the same location, the difficulty of searching for the item in the collection increases.



**Sorting**

**Bubble Sort**

* The bubble sort makes multiple passes through a list.
* It compares adjacent items and exchanges those that are out of order.
* Each pass through the list places the next largest value in its proper place.
* Each item “bubbles” up to the location where it belongs.



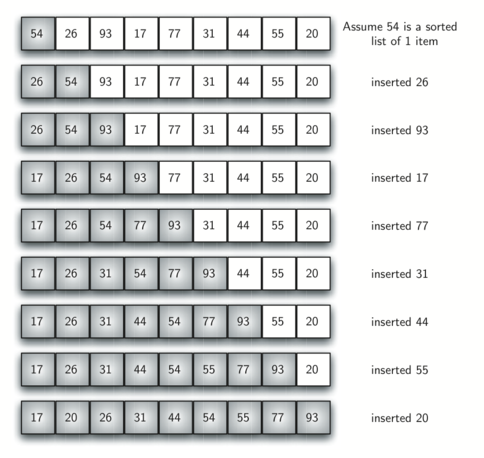
**Selection Sort**

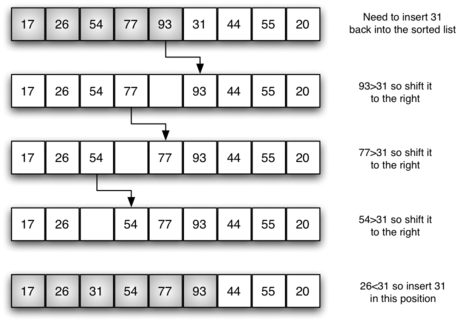
* The bubble sort makes multiple passes through a list.
* It compares adjacent items and exchanges those that are out of order.
* Each pass through the list places the next largest value in its proper place.
* Each item “bubbles” up to the location where it belongs**.**

****

**insertion Sort**

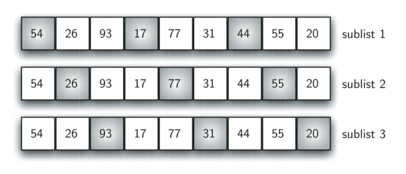
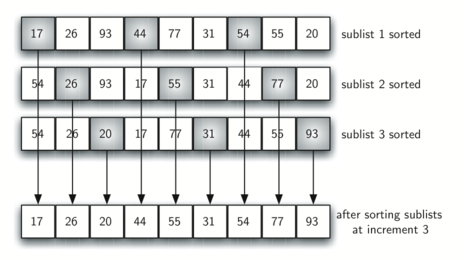
* We begin by assuming that a list with one item (position 0) is already sorted.
* On each pass, one for each item 1 through n−1, the current item is checked against those in the already sorted sublist.
* As we look back into the already sorted sublist, we shift those items that are greater to the right.
* When we reach a smaller item or the end of the sublist, the current item can be inserted.

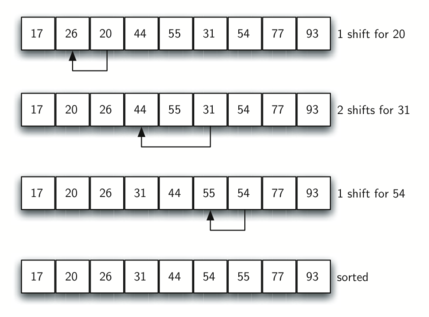
****

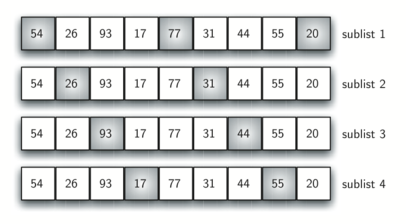
****

**Shell Sort**

* The shell sort improves on the insertion sort by breaking the original list into a number of smaller sublists,
* The unique way that these sublists are chosen is the key to the shell sort.
* Instead of breaking the list into sublists of contiguous items, the shell sort uses an increment ”i”to create a sublist by choosing all items that are ”i” items apart.

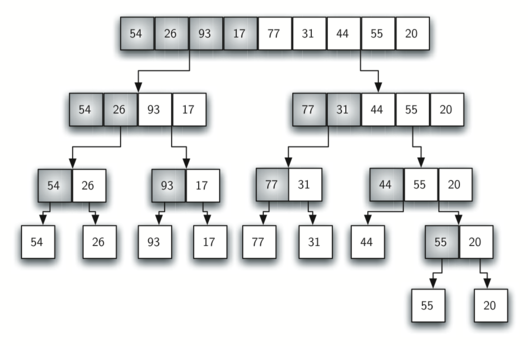
 ****

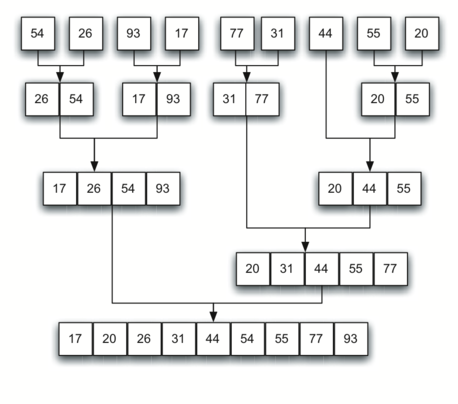




**Merge Sort**

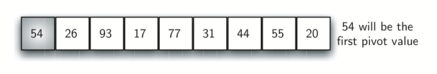
* Merge sort is a recursive algorithm that continually splits a list in half.
* If the list is empty or has one item, it is sorted by definition (the base case).
* If the list has more than one item, we split the list and recursively invoke a merge sort on both halves. Once the two halves are sorted, the fundamental operation, called a **merge**, is performed.
* Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list.



****

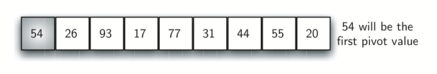
**Quick Sort**

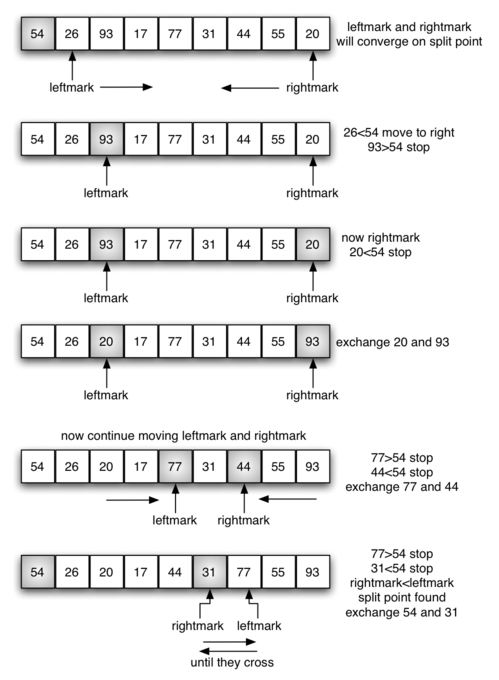
* The quick sort uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage.
* As a trade-off, however, it is possible that the list may not be divided in half.
* When this happens, we will see that performance is diminished.
* A quick sort first selects a value, which is called the **pivot value**.
* The role of the pivot value is to assist with splitting the list.
* The actual position where the pivot value belongs in the final sorted list, commonly called the **split point**, will be used to divide the list for subsequent calls to the quick sort.
* 54 is out first pivot value.

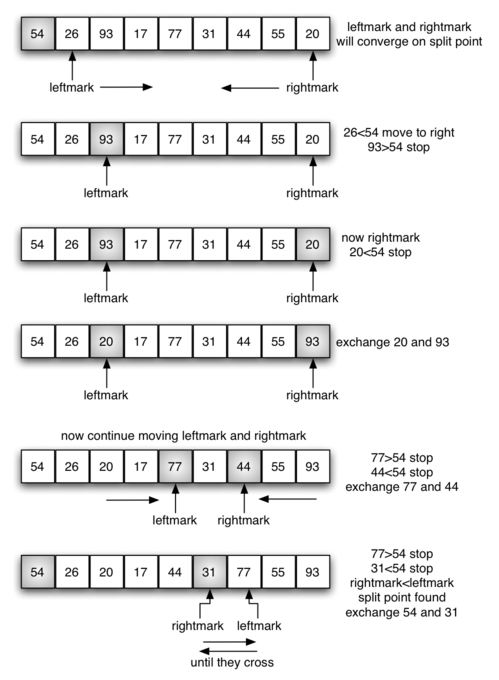


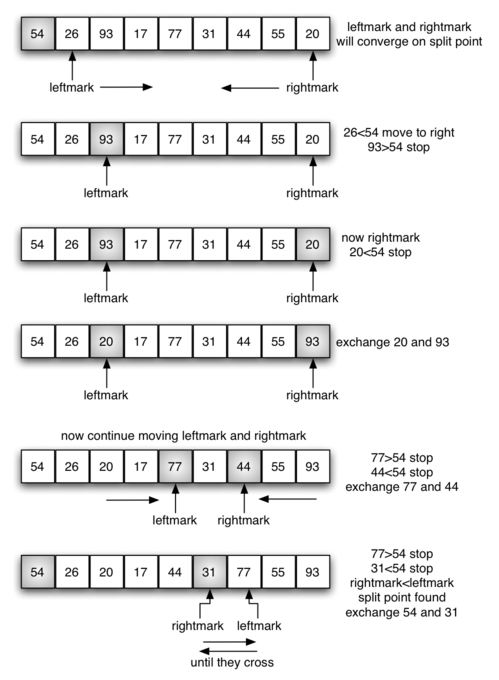
* The **partition** process will happen next.

It will find the split point and at the same time move other items to the appropriate side of the list, either less than or greater than the pivot value



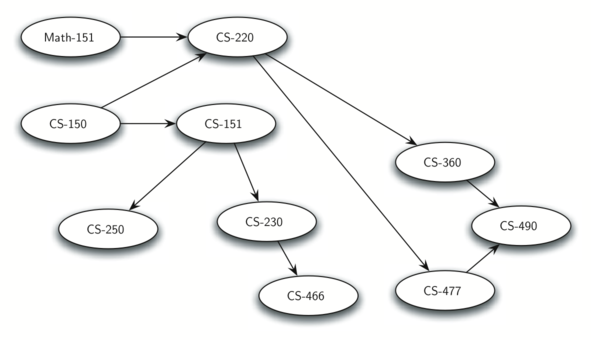
****

****

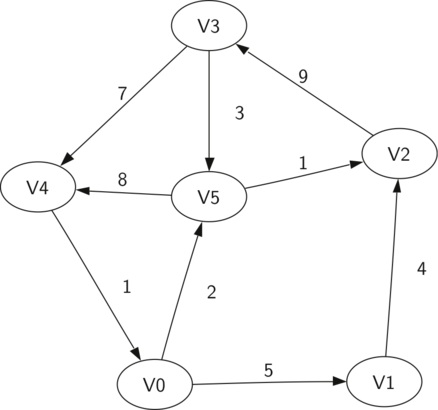


Graphs

* Graphs are a more general structure than trees we can think of a tree as a special kind of graph.
* Graphs can be used to represent many real-world things such as systems of roads, airline flights from city to city, how the Internet is connected, etc.
* Once we have a good representation for a problem, we can use some standard graph algorithms to solve what otherwise might seem to be a very difficult problem.
* Computers can operate well with information presented as a graph.
* An example graph may be the course requirements for a computer science major



* Now that we have looked at some examples of graphs, we will more formally define a graph and its components.
* We already know some of these terms from our discussion of trees. A **vertex** (also called a “**node**”) is a fundamental part of a graph.
* It can have a name, which we will call the “key.”
* A vertex may also have additional information.
* We will call this additional information the “payload.”
* An edge connects two vertices to show that there is a relationship between them.
* Edges may be one-way or two-way.
* If the edges in a graph are all one-way, we say that the graph is a **directed graph**, or a **digraph**.
* The class prerequisites graph shown previously is clearly a digraph since you must take some classes before others.
* Edges may be weighted to show that there is a cost to go from one vertex to another.
* For example in a graph of roads that connect one city to another, the weight on the edge might represent the distance between the two cities.
* A graph can be represented by **G** where **G=(V,E)**
* For the graph **G**, **V** is a set of vertices and **E** is a set of edges.
* Each edge is a tuple **(v,w)** where **w,v∈V**
* We can add a third component to the edge tuple to represent a weight.
* A subgraph **s** is a set of edges **e** and vertices **v** such that **e⊂E** and **v⊂V**



**V={V0,V1,V2,V3,V4,V5}**

**E={(v0,v1,5),(v1,v2,4),**

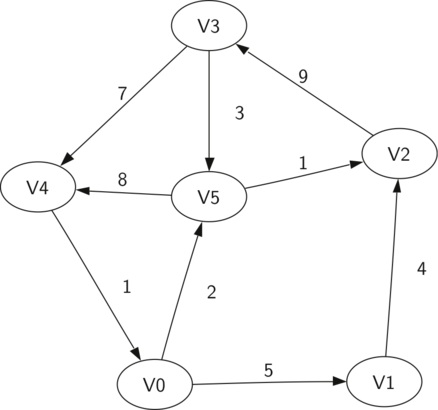
**(v2,v3,9),(v3,v4,7),**

**(v4,v0,1),(v0,v5,2),**

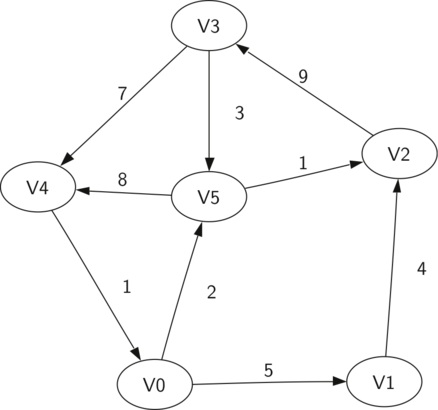
**(v5,v4,8),(v3,v5,3),**

**(v5,v2,1)}**

* A path in a graph is a sequence of vertices that are connected by edges.
* Formally we would define a path as w1,w2,...,wn  such that (wi,wi+1)∈E for all 1≤i≤n−1
* The unweighted path length is the number of edges in the path, specifically n−1.
* The weighted path length is the sum of the weights of all the edges in the path.
* The path from V3 to V1 is the sequence of vertices (V3,V4,V0,V1)
* The edges are {(v3,v4,7),(v4,v0,1),(v0,v1,5)}

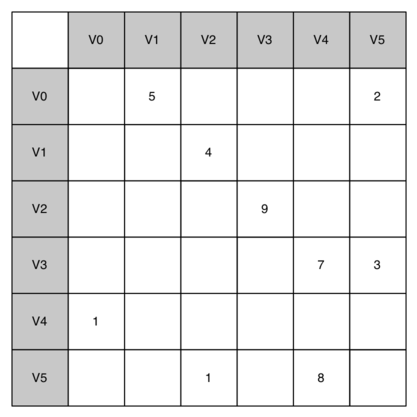


* A cycle in a directed graph is a path that starts and ends at the same vertex.
* A graph with no cycles is called an acyclic graph.
* A directed graph with no cycles is called a directed acyclic graph or a DAG.
* We will see that we can solve several important problems if the problem can be represented as a DAG.
* The path (V5,V2,V3,V5)  is a cycle.

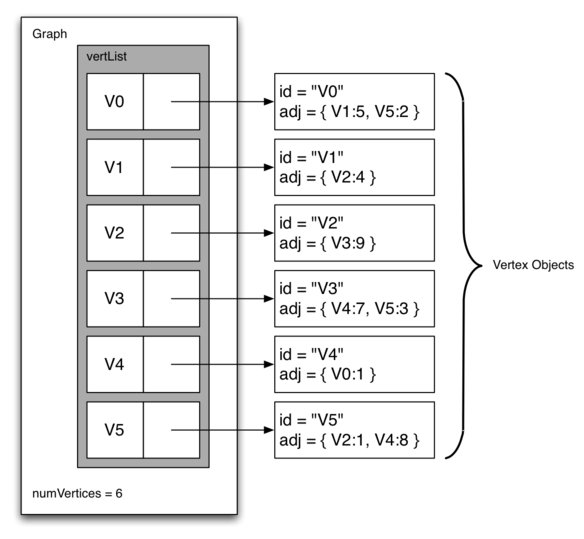
****

**Adjacency matrix and list**

* One of the easiest ways to implement a graph is to use a two-dimensional matrix.
* In this matrix implementation, each of the rows and columns represent a vertex in the graph.
* The value that is stored in the cell at the intersection of row v and column w indicates if there is an edge from vertex v to vertex w.
* When two vertices are connected by an edge, we say that they are adjacent.

****

* The advantage of the adjacency matrix is that it is simple, and for small graphs it is easy to see which nodes are connected to other nodes.
* However, notice that most of the cells in the matrix are empty.
* Because most of the cells are empty we say that this matrix is “sparse.”
* A matrix is not a very efficient way to store sparse data.
* The adjacency matrix is a good implementation for a graph when the number of edges is large.
* Since there is one row and one column for every vertex in the graph, the number of edges required to fill the matrix is **|V|2**.
* A matrix is full when every vertex is connected to every other vertex.
* A more space-efficient way to implement a sparsely connected graph is to use an adjacency list.
* In an adjacency list implementation we keep a master list of all the vertices in the Graph object and then each vertex object in the graph maintains a list of the other vertices that it is connected to.
* In our implementation of the Vertex class we will use a dictionary rather than a list where the dictionary keys are the vertices, and the values are the weights.



* The advantage of the adjacency list implementation is that it allows us to compactly represent a sparse graph.
* The adjacency list also allows us to easily find all the links that are directly connected to a particular vertex.

**Word Ladder Problem**

* Consider the following puzzle called a word ladder.
* Transform the word “FOOL” into the word “SAGE”.
* In a word ladder puzzle you must make the change occur gradually by changing one letter at a time.
* At each step you must transform one word into another word, you are not allowed to transform a word into a non-word.

FOOL

POOL

POLL

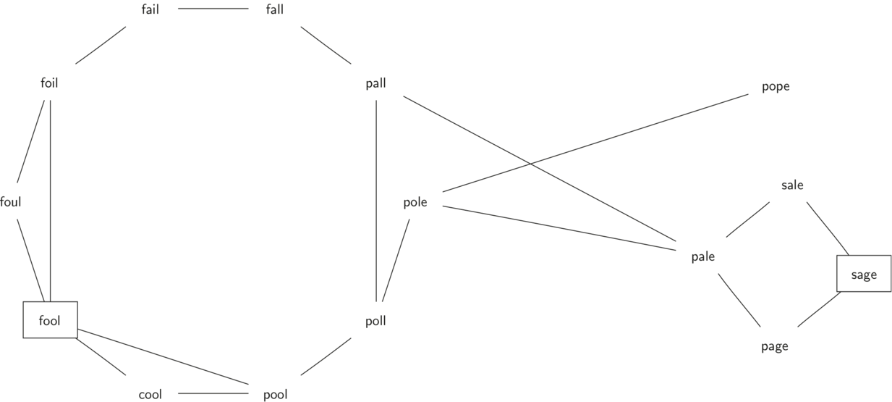
POLE

PALE

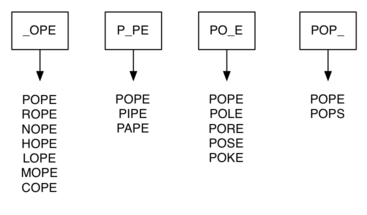
SALE

SAGE

* We can solve this problem using a graph algorithm.
  + Represent the relationships between the words as a graph.
  + Use the graph algorithm known as breadth first search to find an efficient path from the starting word to the ending word.
* Figure out how to turn a large collection of words into a graph.
* What we would like is to have an edge from one word to another if the two words are only different by a single letter.
* Then any path from one word to another is a solution to the word ladder puzzle.



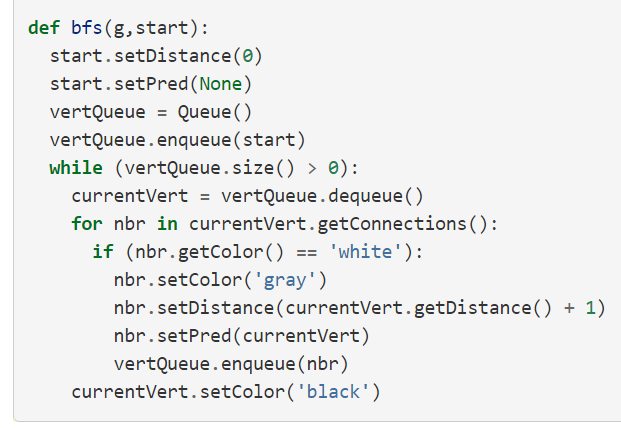
* Suppose that we have a huge number of buckets, each of them with a four-letter word on the outside, except that one of the letters in the label has been replaced by an underscore.

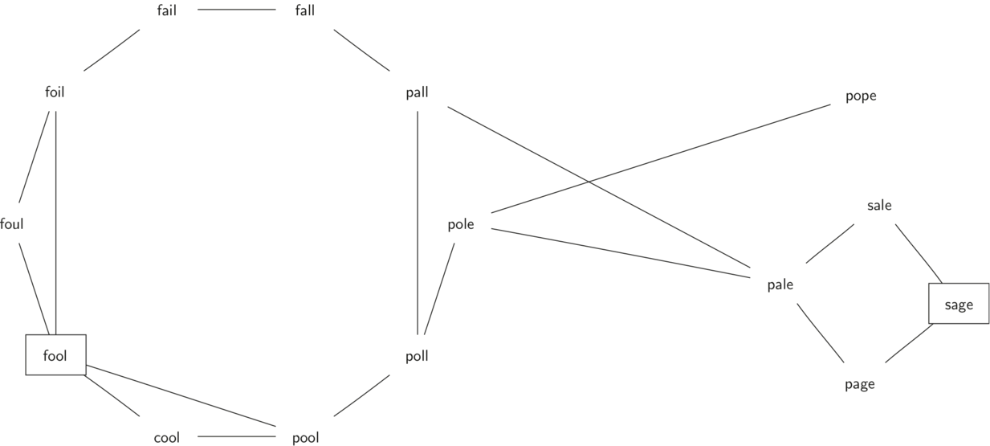


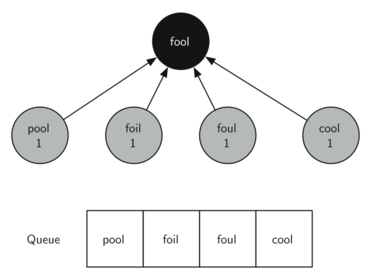
* We can implement the scheme we have just described by using a dictionary.
* The labels on the buckets we have just described are the keys in our dictionary.
* The value stored for that key is a list of words.
* Once we have the dictionary built we can create the graph.
* We start our graph by creating a vertex for each word in the graph.
* Then we create edges between all the vertices we find for words found under the same key in the dictionary.

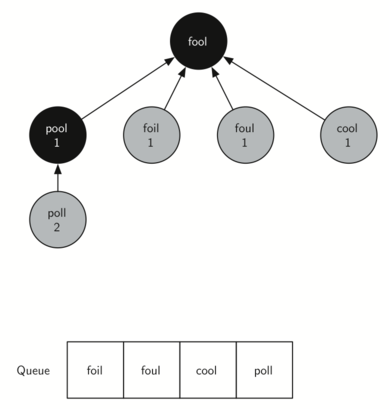
**Breadth First Search**

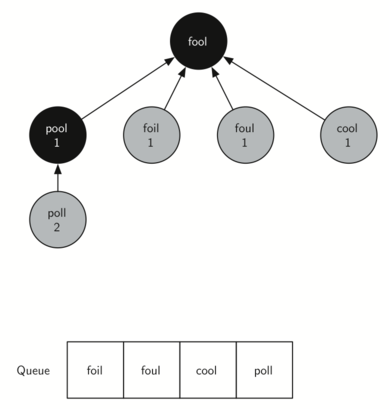
* How can we find the shortest solution to the word ladder problem?
* The graph algorithm we are going to use is called the “breadth first search” algorithm.
* Breadth first search (BFS) is one of the easiest algorithms for searching a graph.
* It also serves as a prototype for several other important graph algorithms that we will study later.
* MIT Algorithms and Data Structure Course!
* <https://www.youtube.com/watch?v=s-CYnVz-uh4>
* Given a graph **G** and a starting vertex **s**, a breadth first search proceeds by exploring edges in the graph to find all the vertices in **G** for which there is a path from **s**.
* The remarkable thing about a breadth first search is that it finds *all* the vertices that are a distance **k** from **s** before it finds *any* vertices that are a distance **k+1**.
* One good way to visualize what the breadth first search algorithm does is to imagine that it is building a tree, one level of the tree at a time.
* A breadth first search adds all children of the starting vertex before it begins to discover any of the grandchildren.
* To keep track of its progress, BFS colors each of the vertices white, gray, or black.
* All the vertices are initialized to white when they are constructed.
* A white vertex is an undiscovered vertex.
* To keep track of its progress, BFS colors each of the vertices white, gray, or black.
* All the vertices are initialized to white when they are constructed.
* A white vertex is an undiscovered vertex.
* When a vertex is initially discovered it is colored gray, and when BFS has completely explored a vertex it is colored black.
* This means that once a vertex is colored black, it has no white vertices adjacent to it.
* A gray node, on the other hand, may have some white vertices adjacent to it, indicating that there are still additional vertices to explore.
* BFS begins at the starting vertex s and colors start gray to show that it is currently being explored.
* Two other values, the distance and the predecessor, are initialized to 0 and None respectively for the starting vertex.
* Finally, start is placed on a Queue.
* The next step is to begin to systematically explore vertices at the front of the queue.
* We explore each new node at the front of the queue by iterating over its adjacency list. As each node on the adjacency list is examined its color is checked.
* If it is white, the vertex is unexplored, and four things happen:
* If it is white, the vertex is unexplored, and four things happen:
  + The new, unexplored vertex nbr, is colored gray.
  + The predecessor of nbr is set to the current node currentVert
  + The distance to nbr is set to the distance to currentVert + 1
  + nbr is added to the end of a queue. Adding nbr to the end of the queue effectively schedules this node for further exploration, but not until all the other vertices on the adjacency list of currentVert have been explored.

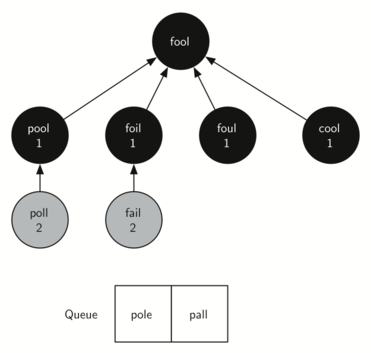


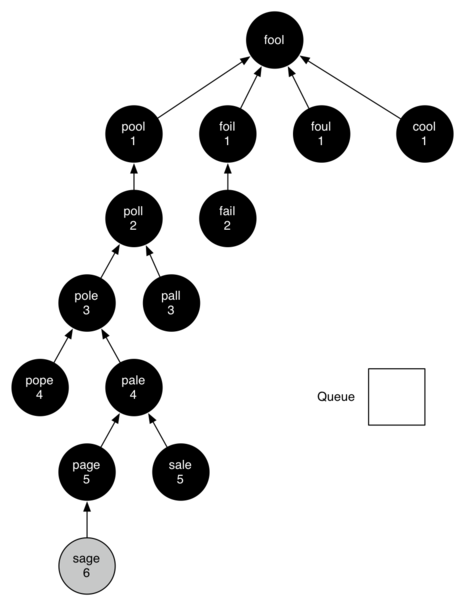


****

****



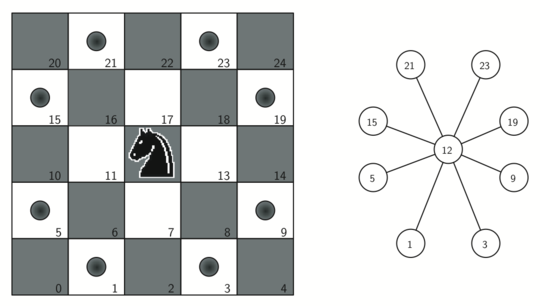
****

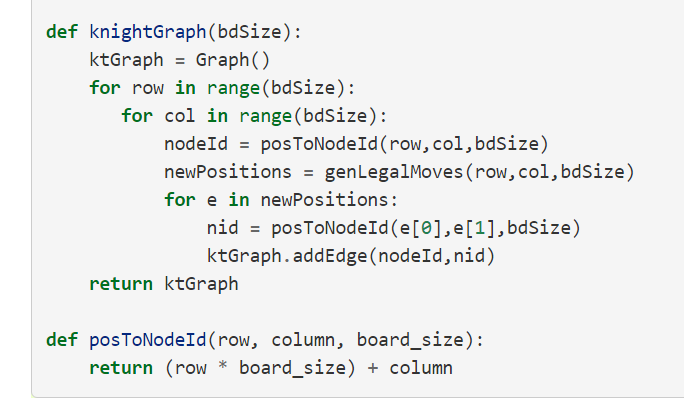
****

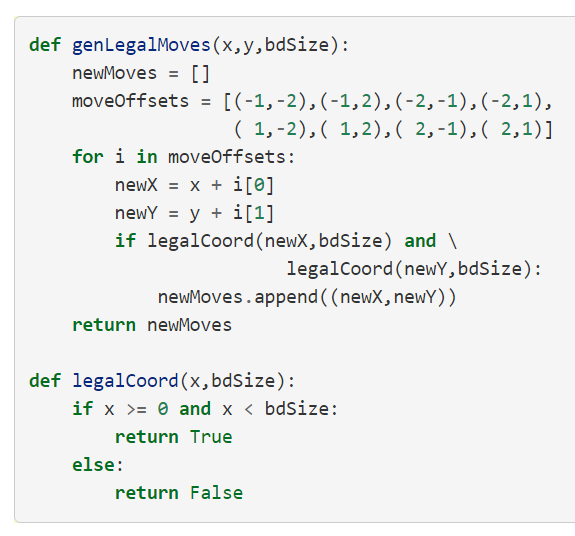
* The amazing thing about the breadth first search solution is that we have not only solved the FOOL–SAGE problem we started out with, but we have solved many other problems along the way.
* We can start at any vertex in the breadth first search tree and follow the predecessor arrows back to the root to find the shortest word ladder from any word back to fool.

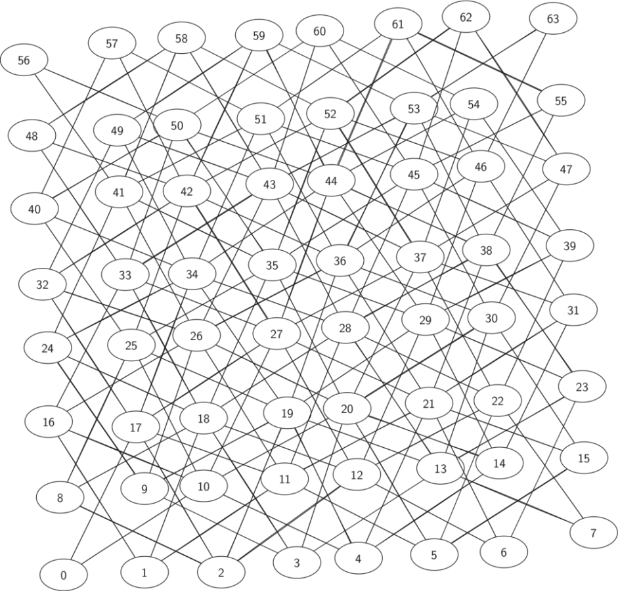
**Knight’s Tour**

* The knight’s tour puzzle is played on a chess board with a single chess piece, the knight.
* The object of the puzzle is to find a sequence of moves that allow the knight to visit every square on the board exactly once.
* We will solve the problem using two main steps:
* Represent the legal moves of a knight on a chessboard as a graph.
* Use a graph algorithm to find a path of length **rows×columns−1** where every vertex on the graph is visited exactly once.
* We will solve the problem using two main steps:
* Represent the legal moves of a knight on a chessboard as a graph.
* Use a graph algorithm to find a path of length **rows×columns−1** where every vertex on the graph is visited exactly once.
* To represent the knight’s tour problem as a graph we will use the following two ideas:
* Each square on the chessboard can be represented as a node in the graph.
* Each legal move by the knight can be represented as an edge in the graph.

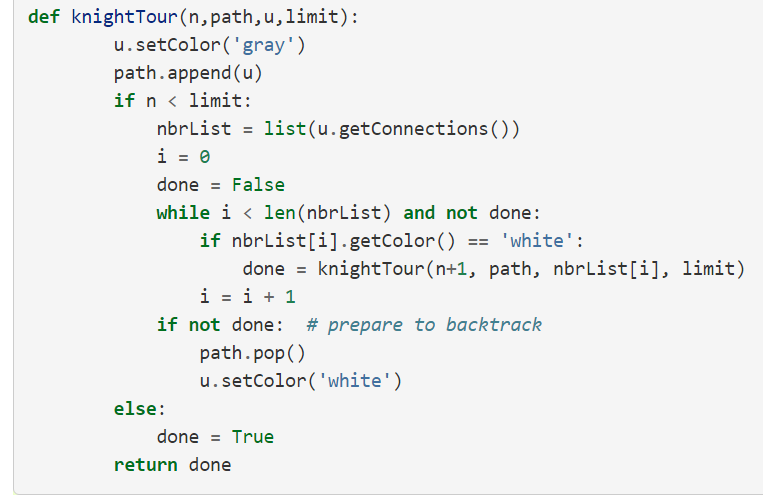




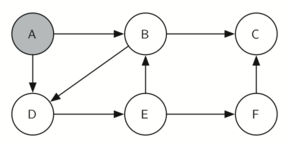
****

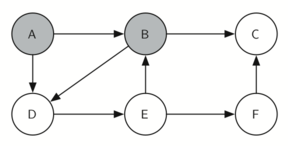
****

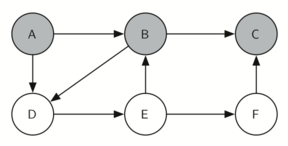
* The search algorithm we will use to solve the knight’s tour problem is called depth first search (DFS).
* Whereas the breadth first search algorithm discussed in the previous section builds a search tree one level at a time, a depth first search creates a search tree by exploring one branch of the tree as deeply as possible.
* In this section we will look at two algorithms that implement a depth first search.
* The first algorithm we will look at directly solves the knight’s tour problem by explicitly forbidding a node to be visited more than once.
* The second implementation is more general, but allows nodes to be visited more than once as the tree is constructed. (This will be the general DFS discussed later)
* The depth first exploration of the graph is exactly what we need in order to find a path that has exactly 63 edges.
* We will see that when the depth first search algorithm finds a dead end (a place in the graph where there are no more moves possible) it backs up the tree to the next deepest vertex that allows it to make a legal move.
* The **knightTour** function takes four parameters:
* **n**, the current depth in the search tree
* **path**, a list of vertices visited up to this point
* **u**, the vertex in the graph we wish to explore
* **limit** the number of nodes in the path.
* The **knightTour** function is recursive.

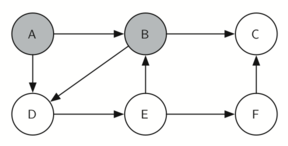


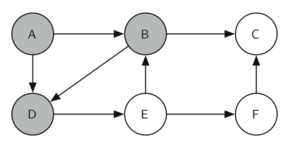
* When the knightTour function is called, it first checks the base case condition.
* If we have a path that contains 64 vertices, we return from knightTour with a status of True, indicating that we have found a successful tour.
* If the path is not long enough we continue to explore one level deeper by choosing a new vertex to explore and calling knightTour recursively for that vertex.
* DFS also uses colors to keep track of which vertices in the graph have been visited.
* Unvisited vertices are colored white, and visited vertices are colored gray.
* If all neighbors of a particular vertex have been explored and we have not yet reached our goal length of 64 vertices, we have reached a dead end.
* When we reach a dead end we must backtrack
* Backtracking happens when we return from **knightTour** with a status of **False**.
* In the breadth first search we used a queue to keep track of which vertex to visit next.
* Since depth first search is recursive, we are implicitly using a stack to help us with our backtracking.
* When we return from a call to **knightTour** with a status of **False** we remain inside the while loop and look at the next vertex in **nbrList**.

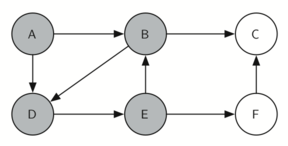


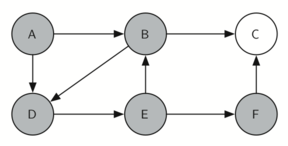


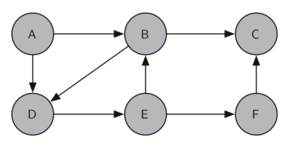
\ 

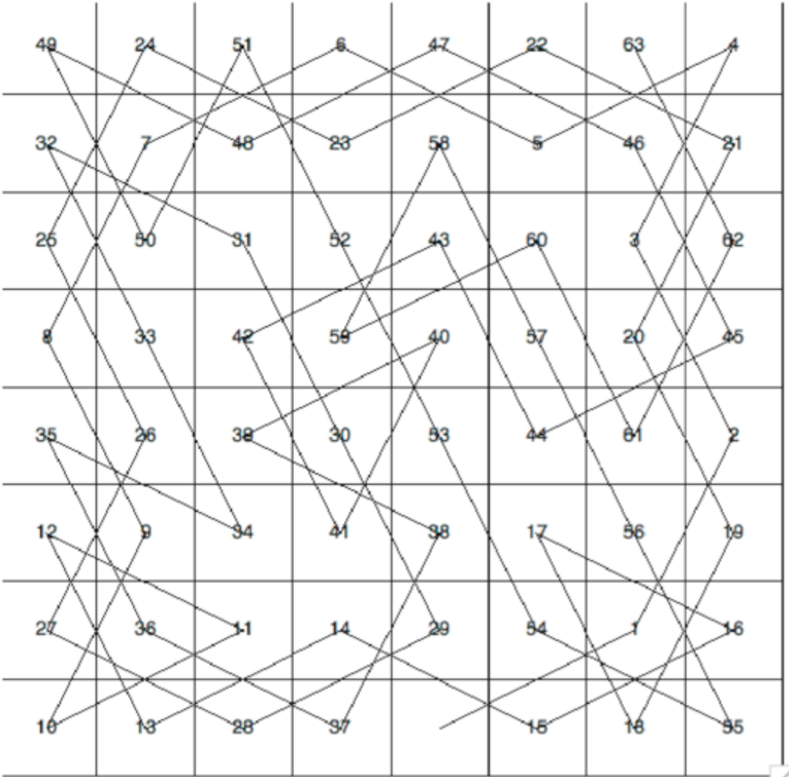






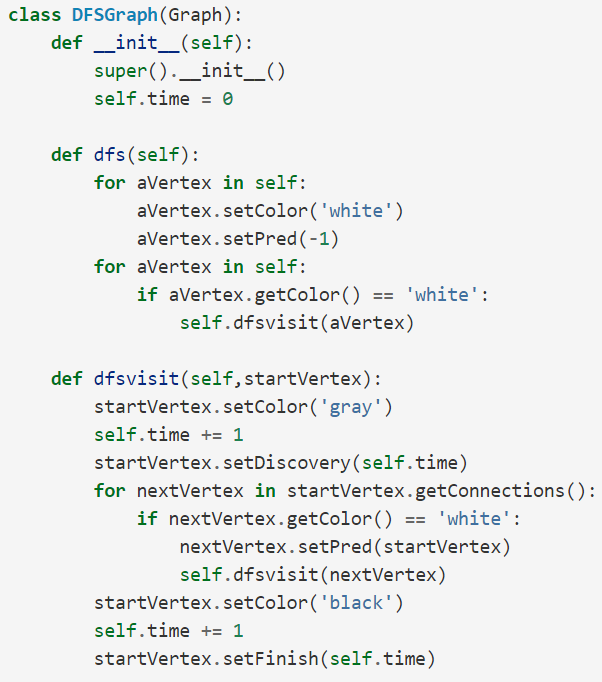
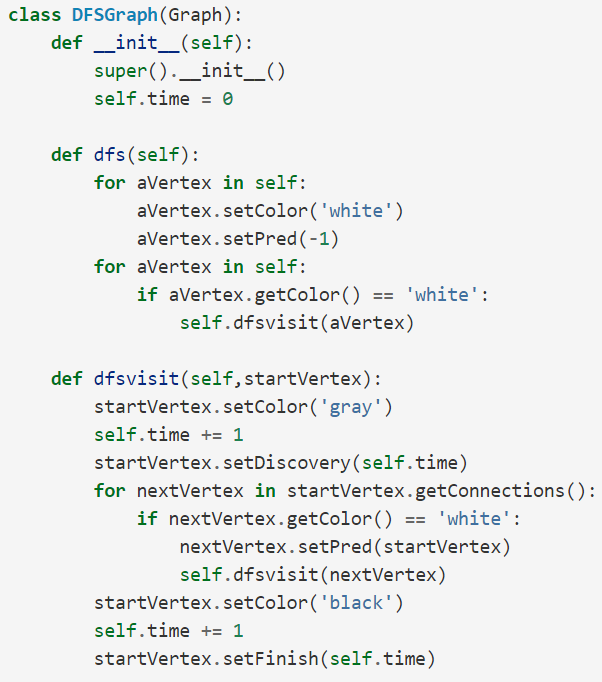
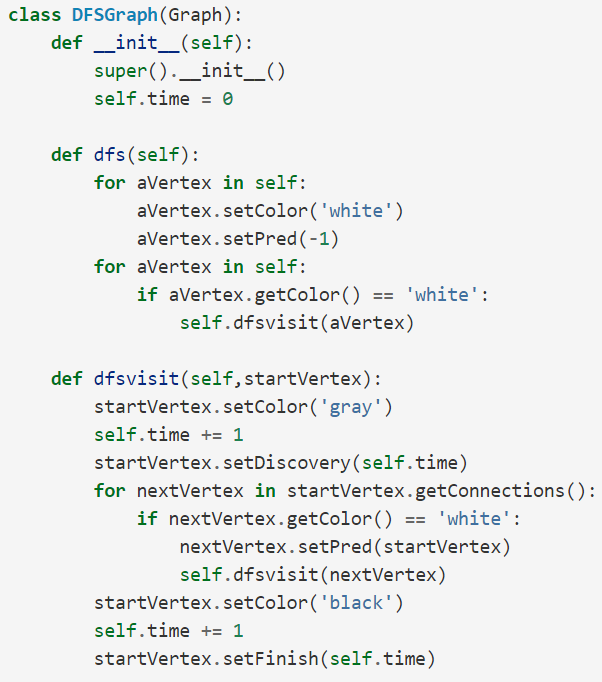
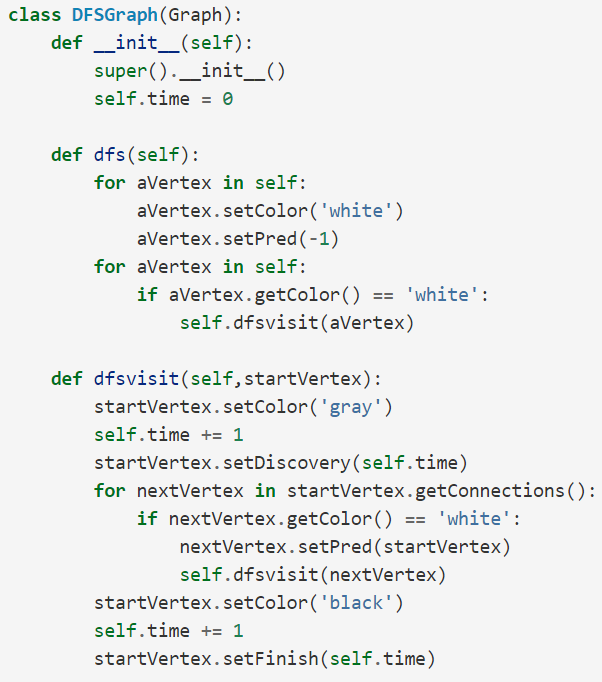


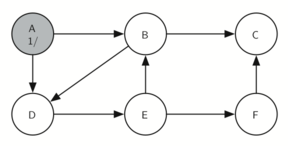


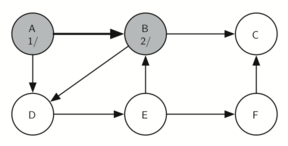


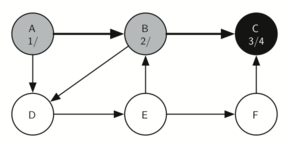
**Depth First Search**

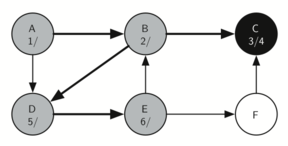
* The knight’s tour is a special case of a depth first search where the goal is to create the deepest depth first tree, without any branches.
* The more general depth first search is actually easier.
* Its goal is to search as deeply as possible, connecting as many nodes in the graph as possible and branching where necessary**.**
* It is even possible that a depth first search will create more than one tree.
* When the depth first search algorithm creates a group of trees we call this a depth first forest.
* As with the breadth first search our depth first search makes use of predecessor links to construct the tree.
* As with the breadth first search our depth first search makes use of predecessor links to construct the tree.
* In addition, the depth first search will make use of two additional instance variables in the Vertex class.
* The new instance variables are the discovery and finish times.
* The discovery time tracks the number of steps in the algorithm before a vertex is first encountered.
* The finish time is the number of steps in the algorithm before a vertex is colored black

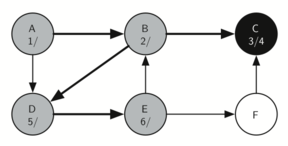


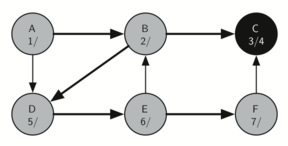


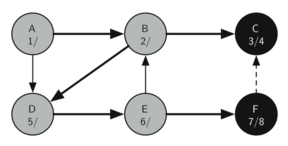


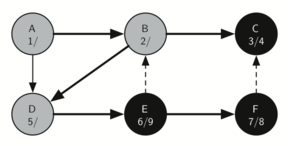


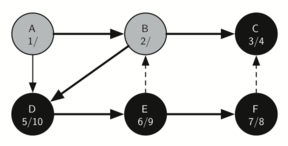


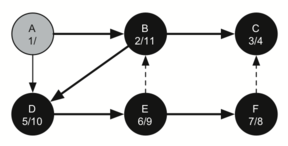


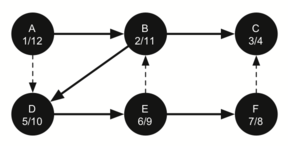












* The starting and finishing times for each node display a property called the **parenthesis property**.
* This property means that all the children of a particular node in the depth first tree have a later discovery time and an earlier finish time than their parent.

