Python for Data Structures, Algorithms

Course: Udemy

Introduction to Algorithm Analysis and Big O

In this lecture we will discuss how to analyze Algorithms and why it is important to do so!

Why analyze algorithms?

An **algorithm** is simply a procedure or formula for solving a problem. Some problems are famous enough that the algorithms have names, as well as some procdures being common enough that the algorithm associated with it also has a name. So now we have a good question we need to answer:

In [1]:
```
# First function (Note the use of xrange since this is in Python 2)
def sum1(n):
    '''
    Take an input of n and return the sum of the numbers from 0 to n
    '''
    final_sum = 0
    for x in range(n+1):
        final_sum += x

    return final_sum

sum1(10)
```

Out[1]: 55

In [2]:
```python
def sum2(n):
    """
    Take an input of n and return the sum of the numbers from 0 to n
    """
    return (n*(n+1))/2

sum2(10)
```

Out[2]: 55.0

You'll notice both functions have the same result, but completely different algorithms. You'll note that the first function iteratively adds the numbers, while the second function makes use of:

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

So how can we objectively compare the algorithms? We could compare the amount of space they take in memory or we could also compare how much time it takes each function to run. We can use the built in **%timeit** magic function in jupyter to compare the time of the functions. The %timeit magic in Jupyter Notebooks will repeat the loop iteration a certain number of times and take the best result. Check out the link for the documentation.

Let's go ahead and compare the time it took to run the functions:

In [3]:
```python
%timeit sum1(100)
```

```
4.64 µs ± 246 ns per loop (mean ± std. dev. of 7 runs, 100000 loops eac
h)
```

In [4]:
```python
%timeit sum2(100)
```

```
143 ns ± 4.4 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops ea
ch)
```

We can see that the second function is much more efficient! Running at a much faster rate than the first. However, we can not use "time to run" as an objective measurement, because that will depend on the speed of the computer itself and hardware capabilities. So we will need to use another method, **Big-O**!

The original **sum1** function will create an assignment **n+1** times, we can see this from the range based function. This means it will assign the final_sum variable n+1 times. We can then say that for a problem of n size (in this case just a number n) this function will take 1+n steps.

This **n** notation allows us to compare solutions and algorithms relative to the size of the problem, since sum1(10) and sum1(100000) would take very different times to run but be using the same algorithm. We can also note that as n grows very large, the **+1** won't have much effect. So let's begin discussing how to build a syntax for this notation.

Now we will discuss how we can formalize this notation and idea.

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

Let's examine some of these points more closely:

- Remember, we want to compare how quickly runtime will grows, not compare exact runtimes, since those can vary depending on hardware.
- Since we want to compare for a variety of input sizes, we are only concerned with runtime grow *relative* to the input. This is why we use **n** for notation.
- As n gets arbitrarily large we only worry about terms that will grow the fastest as n gets large, to this point, Big-O analysis is also known as **asymptotic analysis**

As for syntax sum1() can be said to be **O(n)** since its runtime grows linearly with the input size. In the next lecture we will go over more specific examples of various O() types and examples. To conclude this lecture we will show the potential for vast difference in runtimes of Big-O functions.

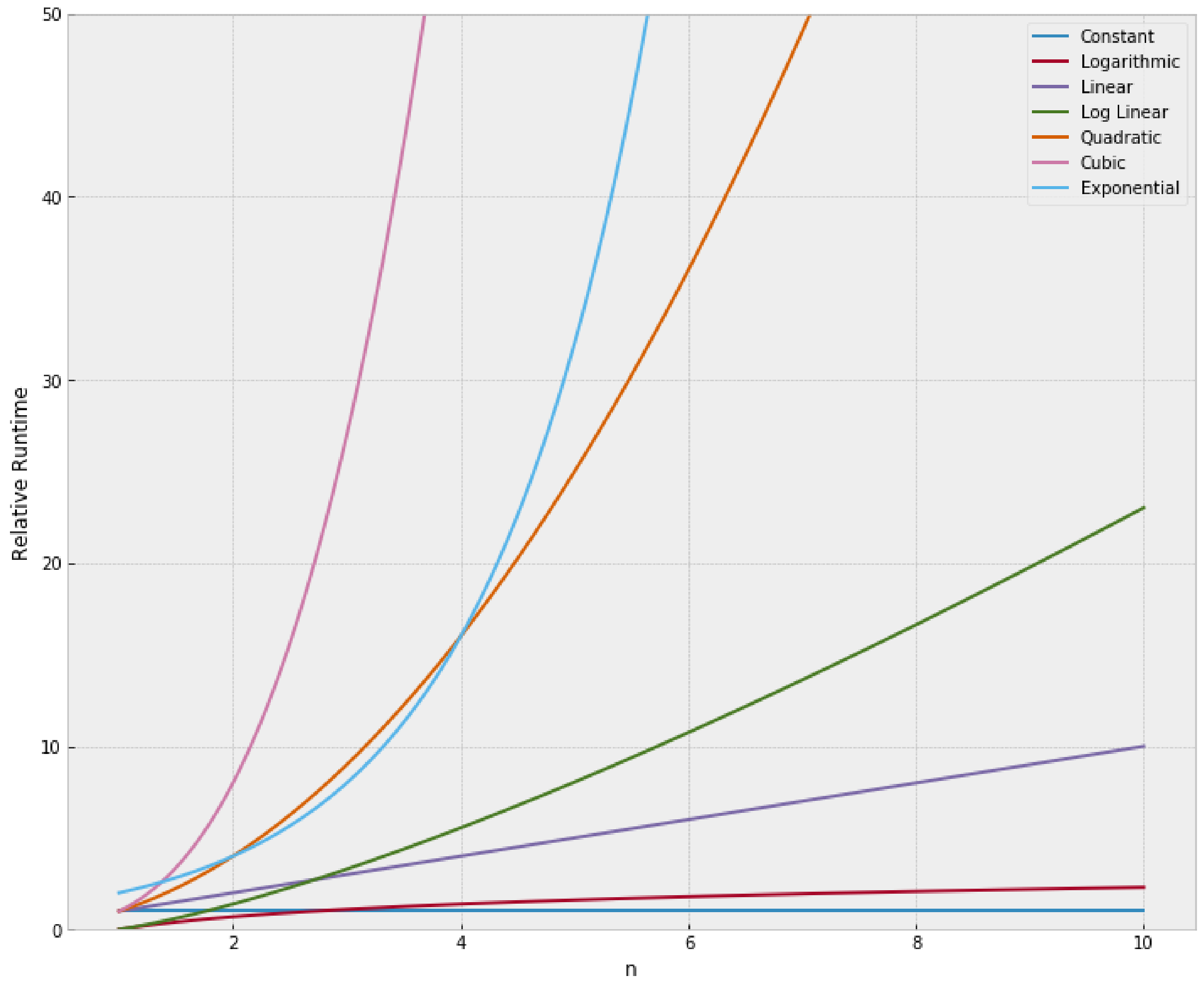
Runtimes of Common Big-O Functions

Here is a table of common Big-O functions:

Big-O	Name
1	Constant
log(n)	Logarithmic
n	Linear
nlog(n)	Log Linear
n^2	Quadratic
n^3	Cubic
2^n	Exponential

In [5]:
```
from math import log
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('bmh')

# Set up runtime comparisons
n = np.linspace(1,10,1000)
labels = ['Constant','Logarithmic','Linear','Log Linear','Quadratic','C
ubic','Exponential']
big_o = [np.ones(n.shape),np.log(n),n,n*np.log(n),n**2,n**3,2**n]

# Plot setup
plt.figure(figsize=(12,10))
plt.ylim(0,50)

for i in range(len(big_o)):
    plt.plot(n,big_o[i],label = labels[i])


plt.legend(loc=0)
plt.ylabel('Relative Runtime')
plt.xlabel('n')
```

```
Out[5]: Text(0.5, 0, 'n')
```



Stack Overflow

- Big-O Notation Explained
- Big-O Examples Explained

```
In [6]: # InterviewBit notes
```

Array Sequences

- Introduction to array
- Low level arrays
- Dynamic array and Amortizing

Term array sequence is general, python has three main sequence classes:

- list: [1,2,3]
- Tuple: (1,2,3)
- String: '123' All support indexing (e.g t[0] =1)

Low level computer architecture

- Memory of a computer stored in bits
- Typical unit in bytes, which is 8 bits
- Computers typically use a memory address.
- Each bytes associated with unquie address. Byte #2144 verses Bytes #2147

Arrays

- Representation of computer memory
- Individual bytes with consecutive addresses
- Computer hardware is designed, in theory, so that an byte of the main memorycan be efficiently accessed
- Computer's main memory performs an random access memory (RAM)
- Just as easy to retrieve byte #8675309 as it is to retrieve byte #309
- Individual bytes of memory can be stored or retrieved in O(1) time.
- Programming language keeps track of the association between an identifier and the memory address.
- May want video game to keep track of the top ten scores for that game.
- Prefer to use a single name for the group.

- Use index numbers to refer to the high scores in the group.
- A group of related variable can be stored on after another in a contiguous portion of the computer's meomory.
- We will denote such a representation as an array.
- A text string is stored as an ordered sequence of individual characters.
- Python internally represents each unicode character with 16bits (2Bytes).
- Six character string, such as 'SAMPLE' would be stored in 12 consecutive bytes of memory.
- This is an array of six characters.
- Each location within an array as an cell.
- Integer index to describe its location.
- Each cell of an array uses the same number of bytes.
- Allows any cell to be accessed in constant time.
- Appropriate memory address can be computed using the calculation, start + (sellsize)(index)
- Higher level abstraction
- Basic abstraction for real world discussion

Referential Arrays

- Imagine 100 student names with ID numbers. Each cell of the array needs to have the some number of bytes. How can we avoid having to have series of names? We can use an array of object References.
- A single list intance may include multiple reference to the same object as element of the list. Single object can be element of two or more lists. When computing the slice of a list, the result is a new list instance. New list has reference to the same elements that are in the original list.
- When computing the slice of a list, the is a new list instance. New list has reference to the same elements that are in the orginal list. temp = primes[3:6],
- Copying Arrays backup = list(primes)
- This produces a ne list that is a shallow copy in that it refrences the same elements as in the first list. If the contents of the list were mutable type, a deep copy, meaning a new list with new elements, can be produced bt using the deepcopy function from the copy module.
- counter = [0]*8, all eight cell reference the same object,
- counter[2] +=1, Does not technically change the value of the existing integer instance. This computers a new integer. prime.extend(extra)

Dynamic Array

- Don't need to specify how large array is beforhand. A list intance often has greater capacity than current length. If element keep getting appended, eventually this extra space runs out. Let's show an example of this extra 'room' with a live code demonstration.

In [7]:
```
import sys
n =10
data = []
for i in range(n):
    a = len(data)
    b = sys.getsizeof(data)
    print('length: {0:3d}; size in bytes: {1:4d} '.format(a,b))
    data.append(n)
```

```
length:   0; size in bytes:   72
length:   1; size in bytes:  104
length:   2; size in bytes:  104
length:   3; size in bytes:  104
length:   4; size in bytes:  104
length:   5; size in bytes:  136
length:   6; size in bytes:  136
length:   7; size in bytes:  136
length:   8; size in bytes:  136
length:   9; size in bytes:  200
```

Dynamic array implementation

- The key is to provide means to grow the array A that stores the element of a list. We can't actually grow that array, its capacity is fixed. If an element is appended to a list at a time when the underlying array is full, we will need to perform the following steps. Allocate a new array B with larger capacity. Set B[i] =A[i], for i = 0,1,...n-1, where n denotes current number of items. Set A = B, that is, we hence forth use B as the array supporting the list. Insert the new element in the new arry.

a) create new array B; b) store element af A in B,c) reassign reference A to the new array.

How large of a new array to create? A commonly used rule is for the new array to have twice the capacity of the existing array that has been filled. We will mathematic reasoning behind this later.

In [8]:
```
import ctypes

class DynamicArray(object):
    '''
    DYNAMIC ARRAY CLASS (Similar to Python List)
    '''

    def __init__(self):
        self.n = 0 # Count actual elements (Default is 0)
        self.capacity = 1 # Default Capacity
        self.A = self.make_array(self.capacity)

    def __len__(self):
        """
        Return number of elements sorted in array
        """
        return self.n

    def __getitem__(self,k):
        """
        Return element at index k
        """
        if not 0 <= k <self.n:
            return IndexError('K is out of bounds!') # Check it k index
 is in bounds of array

        return self.A[k] #Retrieve from array at index k

    def append(self, ele):
        """
        Add element to end of the array
        """
        if self.n == self.capacity:
            self._resize(2*self.capacity) #Double capacity if not enoug
```

```
h room

        self.A[self.n] = ele #Set self.n index to element
        self.n += 1

    def _resize(self,new_cap):
        """
        Resize internal array to capacity new_cap
        """

        B = self.make_array(new_cap) # New bigger array

        for k in range(self.n): # Reference all existing values
            B[k] = self.A[k]

        self.A = B # Call A the new bigger array
        self.capacity = new_cap # Reset the capacity

    def make_array(self,new_cap):
        """
        Returns a new array with new_cap capacity
        """
        return (new_cap * ctypes.py_object)()
```

In [9]:

```
# Instantiate
arr = DynamicArray()
# Append new element
arr.append(1)
# Check length
len(arr)
```

Out[9]:

```
1
```

In [10]:

```
# Append new element
arr.append(2)
# Check length
len(arr)
```

Out[10]:

```
2
```

Amortized Analysis

The strategy of replacing an array with a new, larger array might at first seem slow, A single append operation may require O(n) time to perform. Our new array allows us to add n new elements before the array must be replaced again. Using an algorithmic design pattern called amortization, we can show that performing a sequence of such append operations on a dynamic array is actually quite efficient.

Amortized analysis

1. Allocate memory for a larger array of size, typically twice the old array.
2. Copy the contents of old array to new array.
3. Free the old array.

The amortized cost per operation for a sequence of n operations is the total cost of the operations divided by n.

Anagram Check: Problem

Given two strings, check to see if they are anagrams. An anagram is when the two strings can be written using the exact same letters (so you can just rearrange the letters to get a different phrase or word). For example:

```
"public relations" is an anagram of "crap built on lies."
"clint eastwood" is an anagram of "old west action"
```

Note: Ignore spaces and capitalization. So "d go" is an anagram of "God" and "dog" and "o d g".

In [11]:
```
# 242. Valid Anagram(https://leetcode.com/problems/valid-anagram/descri
ption/)
def anagram(s, t):
    """
    :type s: str
    :type t: str
    :rtype: bool
    """
    s=s.replace(' ','').lower()
    t=t.replace(' ','').lower()

    if(len(s)!=len(t)):
        return False
    counter = {}
    for letter in s:
        if letter in counter:
            counter[letter] += 1
        else:
            counter[letter] = 1

    for letter in t:
        if letter in counter:
            counter[letter] -= 1
        else:
            return False

    for k in counter:
        if counter[k]!=0:
            return False

    return True
```

In [12]:
```
anagram('dog','god')
```

Out[12]:
```
True
```

Array Pair Sum

Given an integer array, output all the ***unique*** pairs that sum up to a specific value **k**. So the input:

```
pair_sum([1,3,2,2],4)
```

would return **2** pairs:

```
(1,3)
(2,2)
```

NOTE: FOR TESTING PURPOSES CHANGE YOUR FUNCTION SO IT OUTPUTS THE NUMBER OF PAIRS

In [13]:
```python
def pair_sum(arr,total):

    length = len(arr)
    # Edge check
    if length %2 != 0:
        return False
    i =0
    while i < length:
        if arr[i] + arr[i+1] != total:
            return False
        i +=2
    else:
        return True
```

In [14]:
```python
pair_sum([10,2,6,6,8,14],12)
```

Out[14]:
```
False
```

In [15]:
```python
def pair_sum(arr,k):

    if len(arr) <2:
        return

    #sets of tracking
    seen = set()
    output = set()

    for num in arr:
        target = k-num

        if target not in seen:
            seen.add(num)
        else:
            output.add((min(num,target),max(num,target)))

    #return len(output)
    print( '\n'.join(map(str,list(output))))
```

In [16]:
```
pair_sum([10,2,6,6,8,14],12)
```

```
(6, 6)
(2, 10)
```

Find missing element

In [17]:
```
# My solution
def finder(arr1,arr2):
    missing = []
    for i in arr1:
        if i not in arr2:
            missing.append(i)
    return missing
```

In [18]:
```
finder([1,2,3,4,5],[2,1,3])
```

Out[18]:
```
[4, 5]
```

Largest Continuous Sum

Given an array of integers (positive and negative) find the largest continuous sum.

In [19]:
```python
def large_cont_sum(arr):
    if len(arr) == 0:
        return 0

    max_num = sum1 = arr[0]# max=sum=arr[0] bug: TypeError: 'int' objec
t is not callable. (Do not use the keyword!)

    for n in arr[1:]:
        sum1 = max(sum1+n, n)
        max_num = max(sum1, max_num)
    return max_num
```

In [20]:
```python
large_cont_sum([1,2,-1,3,4,10,10,-10,-1])
```

Out[20]:
```
29
```

Sentence Reversal

Given a string of words, reverse all the words. For example: Given:

```
'This is the best'
```

Return:

```
'best the is This'
```

As part of this exercise you should remove all leading and trailing whitespace. So that inputs such as:

```
'  space here'  and 'space here      '
```

both become:

```
'here space'
```

In [21]:
```
def rev_word(s):

    words = []
    length = len(s)
    space = [' ']

    i= 0
    while i < length:
        if s[i] not in space:
            word_start =i

            while i < length and s[i] not in space:

                i+=1
            words.append(s[word_start:i])

        i +=1

    return words

def final_output(words):
    length = len(words)
    s = ''

    i = length
    while i != 0:
        s =s + words[i-1] + " "
        i -= 1
    return s
```

In [22]:
```
words = rev_word('Hi sudhir kumar')
final_output(words)
```

Out[22]:
```
'kumar sudhir Hi '
```

String Compression

Given a string in the form 'AAAABBBBCCCCCDDEEEE' compress it to become 'A4B4C5D2E4'. For this problem, you can falsely "compress" strings of single or double letters. For instance, it is okay for 'AAB' to return 'A2B1' even though this technically takes more space.

The function should also be case sensitive, so that a string 'AAAaaa' returns 'A3a3'.

In [23]:
```python
def compress(s):
    r = ""
    l = len(s)

    if l==0:
        return ''
    if l==1:
        return s+'1'

    last = s[0]
    cnt = 1
    i=1

    while i<l:
        if s[i]==s[i-1]:
            cnt += 1
        else:
            r=r+s[i-1]+str(cnt)
            cnt=1
        i+=1
    r = r+s[i-1]+str(cnt)
    return r
```

In [24]:
```python
compress('AAAAABBBBCCCC')
```

Out[24]:
```
'A5B4C4'
```

Unique Characters in String

Given a string,determine if it is compreised of all unique characters. For example, the string 'abcde' has all unique characters and should return True. The string 'aabcde' contains duplicate characters and should return false.

In [25]:
```
def uni_char(s):
    u = set()
    for c in s:
        if c in u:
            return False
        else:
            u.add(c)
    return True
```

In [26]:
```
# another 1-line solution
def uni_char2(s):
    return len(set(s)) == len(s)
```

Stacks Overview

A stack is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end. This end is commonly referred to as the “top.” The end opposite the top is known as the “base.”

The base of the stack is significant since items stored in the stack that are closer to the base represent those that have been in the stack the longest. The most recently added item is the one that is in position to be removed first.

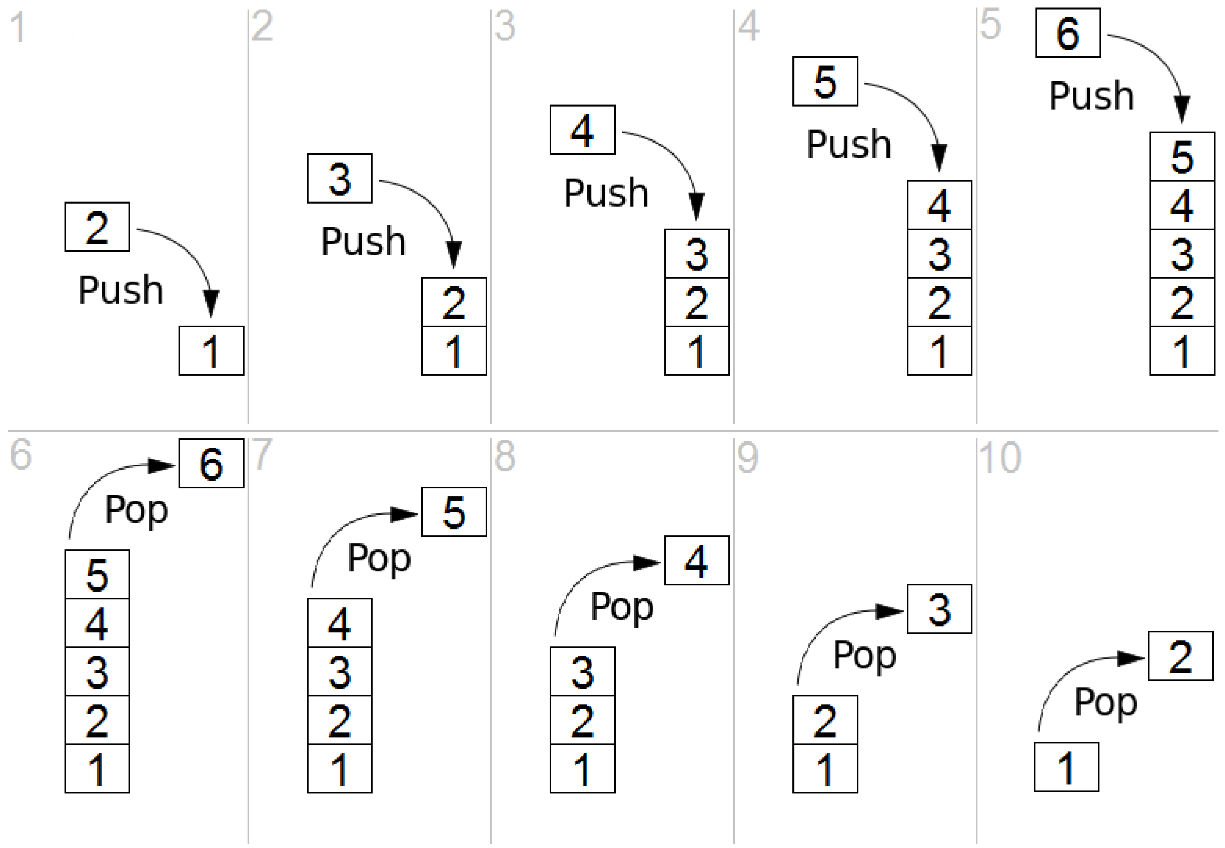
This ordering principle is sometimes called LIFO, last-in first-out. It provides an ordering based on length of time in the collection. Newer items are near the top, while older items are near the base.

For example, consider the figure below:

In [27]:
```
from IPython.display import Image
url ='https://upload.wikimedia.org/wikipedia/commons/b/b4/Lifo_stack.pn
g'
```

```
Image(url)
```

Out[27]:



Note how the first items "pushed" to the stack begin at the base, and as items are "popped" out. Stacks are fundamentally important, as they can be used to reverse the order of items. The order of insertion is the reverse of the order of removal.

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

Extra Resources:

Wikipedia Page on Stacks

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.

In [28]:
```
class Stack:


    def __init__(self):
        self.items = []

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

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

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

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

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

In [29]:
```
s = Stack()
print(s.isEmpty())
```

```
True
```

In [30]:
```
s.push(1)
s.push('two')
s.peek()
```

Out[30]:
```
'two'
```

In [31]:
```
s.push(True)
s.size()
```

Out[31]:
```
3
```

In [32]:
```
s.isEmpty()
```

Out[32]:
```
False
```

In [33]:
```
s.pop()
```

```
Out[33]: True
```

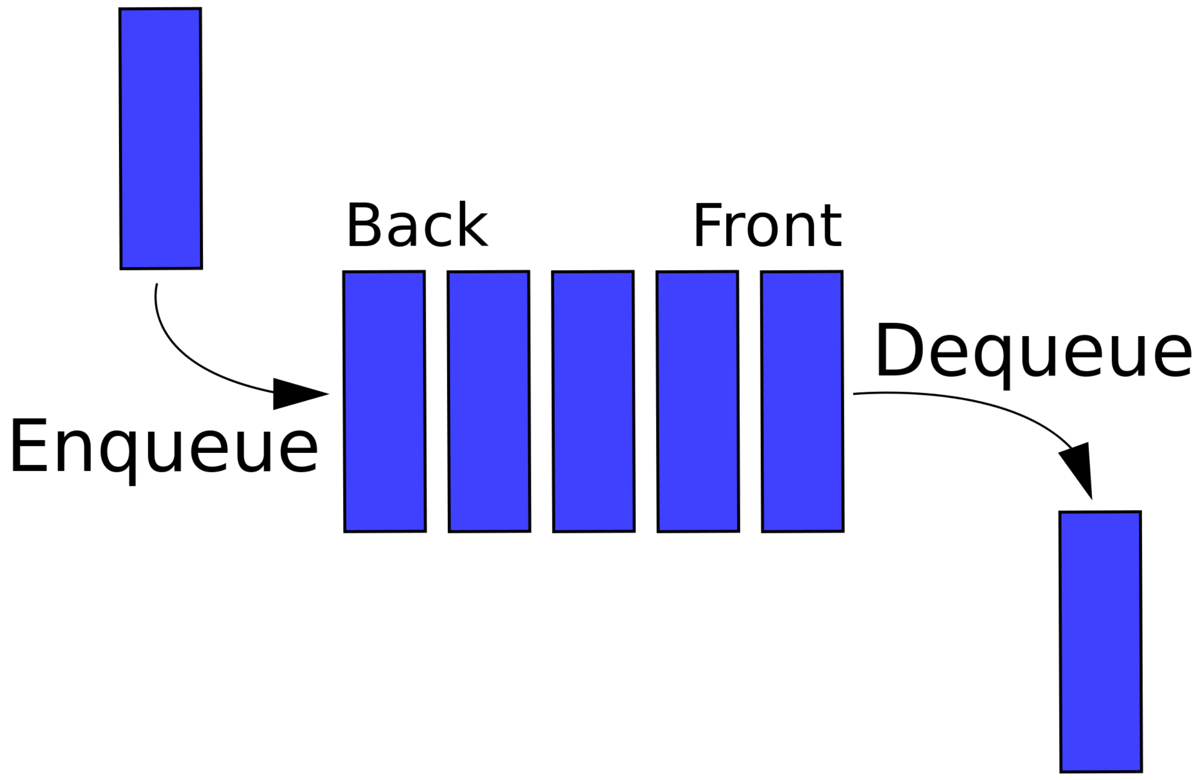
Queues Overview

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.

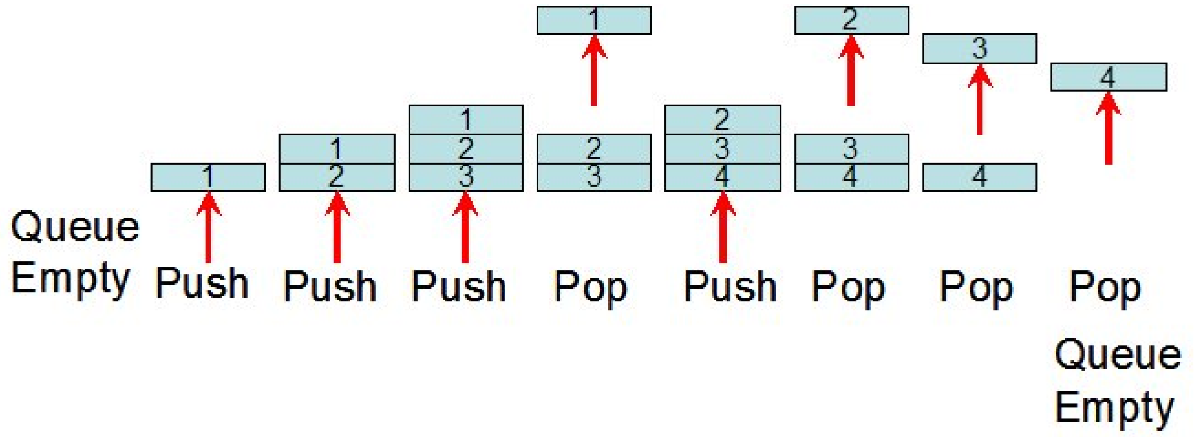
Let's see a diagram which shows this and compares it to the Stack Data Structure:



In [34]:
```
#from IPython.display import Image
#url = 'https://netmatze.files.wordpress.com/2014/08/queue.png'
#Image(url)
```

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.

Let's take a look at how pop and push methods would work with a Queue (versus that of a Stack):



In [35]:
```
#url2 = 'http://www.csit.parkland.edu/~mbrandyberry/CS2Java/Lessons/Sta
ck_Queue/images/QueuePushPop.jpg'
#Image(url2)
```

You should now have a basic understanding of Queues and the FIFO principal for them.

Implementation of Queue

Queue Methods and Attributes

Before we begin implementing our own queue, let's review the attribute and methods it will have:

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

- size() returns the number of items in the queue. It needs no parameters and returns an integer.

```
In [36]: class Queue:
             def __init__(self):
                 self.items = []

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

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

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

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

```
In [37]: q = Queue()
         q.size()

Out[37]: 0
```

```
In [38]: q.isEmpty()

Out[38]: True
```

```
In [39]: q.enqueue(1)
```

```
In [40]: q.dequeue()

Out[40]: 1
```

Deques Overview

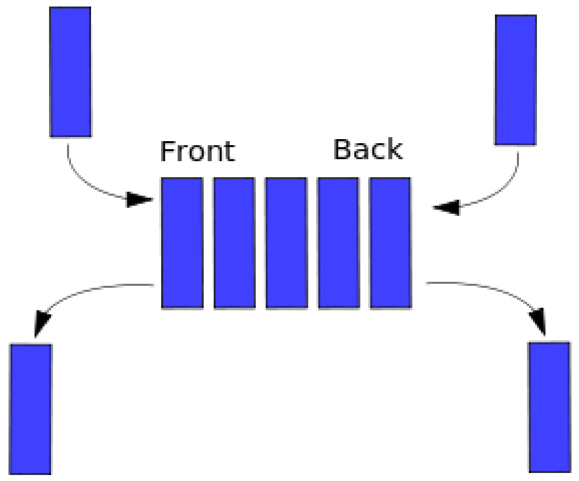
A deque, also known as a double-ended queue, is an ordered collection of items similar to the queue. It has two ends, a front and a rear, and the items remain positioned in the collection. What makes a deque different is the unrestrictive nature of adding and removing items. New items can be added at 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.

Let's see an Image to visualize the Deque Data Structure:

In [41]:
```
from IPython.display import Image
Image('http://www.codeproject.com/KB/recipes/669131/deque.png')
```

Out[41]:



Note how we can both add and remove from the front and the back of the Deque.

Implementation of Deque

Methods and Attributes

- Deque() creates a new deque that is empty. It needs no parameters and returns an empty deque.
- addFront(item) adds a new item to the front of the deque. It needs the item and returns nothing.
- addRear(item) adds a new item to the rear of the deque. It needs the item and returns nothing.
- removeFront() removes the front item from the deque. It needs no parameters and returns the item. The deque is modified.
- removeRear() removes the rear item from the deque. It needs no parameters and returns the item. The deque is modified.
- isEmpty() tests to see whether the deque is empty. It needs no parameters and returns a boolean value.
- size() returns the number of items in the deque. It needs no parameters and returns an integer.

In [42]:
```
class Deque:
    def __init__(self):
        self.items = []

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

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

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

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

    def removeRear(self):
        return self.items.pop(0)
    def size(self):
        return len(self.items)
```

In [43]:
```
d = Deque()
d.addFront('hello')
d.addRear('world')
d.size()
```

Out[43]:
```
2
```

In [44]:
```
print( d.removeFront() + ' ' +  d.removeRear())
```

```
hello world
```

In [45]:
```
d.size()
```

Out[45]:
```
0
```

Balanced Parentheses Check - SOLUTION

Problem Statement

Given a string of opening and closing parentheses, check whether it's balanced. We have 3 types of parentheses: round brackets: (), square brackets: [], and curly brackets: {}. Assume that the string doesn't contain any other character than these, no spaces words or numbers. As a reminder, balanced parentheses require every opening parenthesis to be closed in the reverse order opened. For example '([])' is balanced but '([)]' is not.

Solution

This is a very common interview question and is one of the main ways to check your knowledge of using Stacks! We will start our solution logic as such:

First we will scan the string from left to right, and every time we see an opening parenthesis we push it to a stack, because we want the last opening parenthesis to be closed first. (Remember the FILO structure of a stack!)

Then, when we see a closing parenthesis we check whether the last opened one is the corresponding closing match, by popping an element from the stack. If it’s a valid match, then we proceed forward, if not return false.

Or if the stack is empty we also return false, because there’s no opening parenthesis associated with this closing one. In the end, we also check whether the stack is empty. If so, we return true, otherwise return false because there were some opened parenthesis that were not closed.

Here's an example solution:

In [46]:
```
def balance_check(s):

    # Check is even number of brackets
    if len(s)%2 != 0:
        return False

    # Set of opening brackets
    opening = set('([{')

    # Matching Pairs
    matches = set([ ('(',')'), ('[',']'), ('{','}') ])

    # Use a list as a "Stack"
    stack = []

    # Check every parenthesis in string
    for paren in s:

        # If its an opening, append it to list
        if paren in opening:
            stack.append(paren)

        else:

            # Check that there are parentheses in Stack
            if len(stack) == 0:
                return False
            # Check the last open parenthesis
            last_open = stack.pop()

            # Check if it has a closing match
            if (last_open,paren) not in matches:
                return False

    return len(stack) == 0
```

In [47]:
```
balance_check('[]')
```

Out[47]:
```
True
```

In [48]:
```
balance_check('[](){([[[]]])}')
```

Out[48]:
```
True
```

In [49]:
```
balance_check('()(){]}')
```

Out[49]:
```
False
```

Implement a Queue - Using Two Stacks - SOLUTION

Given the Stack class below, implement a Queue class using **two** stacks! Note, this is a "classic" interview problem. Use a Python list data structure as your Stack.

In [50]:
```
stack1 = []
stack2 = []
```

Solution

The key insight is that a stack reverses order (while a queue doesn't). A sequence of elements pushed on a stack comes back in reversed order when popped. Consequently, two stacks chained together will return elements in the same order, since reversed order reversed again is original order.

We use an in-stack that we fill when an element is enqueued and the dequeue operation takes elements from an out-stack. If the out-stack is empty we pop all elements from the in-stack and push them onto the out-stack.

In [51]:
```
class Queue2Stacks(object):

    def __init__(self):

        # Two Stacks
        self.instack = []
        self.outstack = []

    def enqueue(self,element):

        # Add an enqueue with the "IN" stack
        self.instack.append(element)

    def dequeue(self):
        if not self.outstack:
            while self.instack:
                # Add the elements to the outstack to reverse the order
 when called
                self.outstack.append(self.instack.pop())
        return self.outstack.pop()
```

Test Your Solution

You should be able to tell with your current knowledge of Stacks and Queues if this is working as it should. For example, the following should print as such:

In [52]:
```
"""
RUN THIS CELL TO CHECK THAT YOUR SOLUTION OUTPUT MAKES SENSE AND BEHAVE
S AS A QUEUE
"""
q = Queue2Stacks()
```

```
for i in range(5):
    q.enqueue(i)

for i in range(5):
    print( q.dequeue())
```

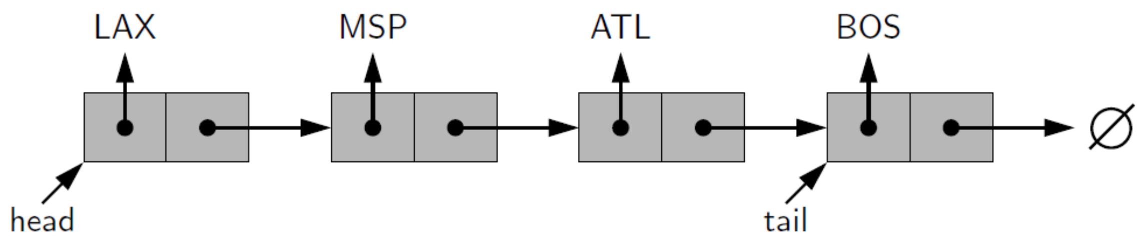
```
0
1
2
3
4
```

Singly Linked List

- Singly Linked List Overview
- Adding Elements to List
- Removing Elements from List

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 ext node of the list.

The list instance maintains a member named head that identifies the first node of the list.In some applications another member named tail that identifies the last node of the list. The first and last node of a linked list are known as the head and tail of the list.We can identify the tail as the node having None as its next reference. This process is commonly known as traversing the linked list. Because the next reference of a node can be viewed as a link or pointer to another node, the process of traversing a list is also known as link hopping or pointer hopping. Each node is represented as a unique object, with that instance storing a reference to its element and a reference to the next node (or None)



Singly Linked List Implementation

In this lecture we will implement a basic Singly Linked List.

Remember, in a singly linked list, we have an ordered list of items as individual Nodes that have pointers to other Nodes.

In [53]:
```
class Node(object):

    def __init__(self,value):

        self.value = value
        self.nextnode = None
```

Now we can build out Linked List with the collection of nodes:

In [54]:
```
a = Node(1)
b = Node(2)
c = Node(3)
```

In [55]:
```
a.nextnode = b
```

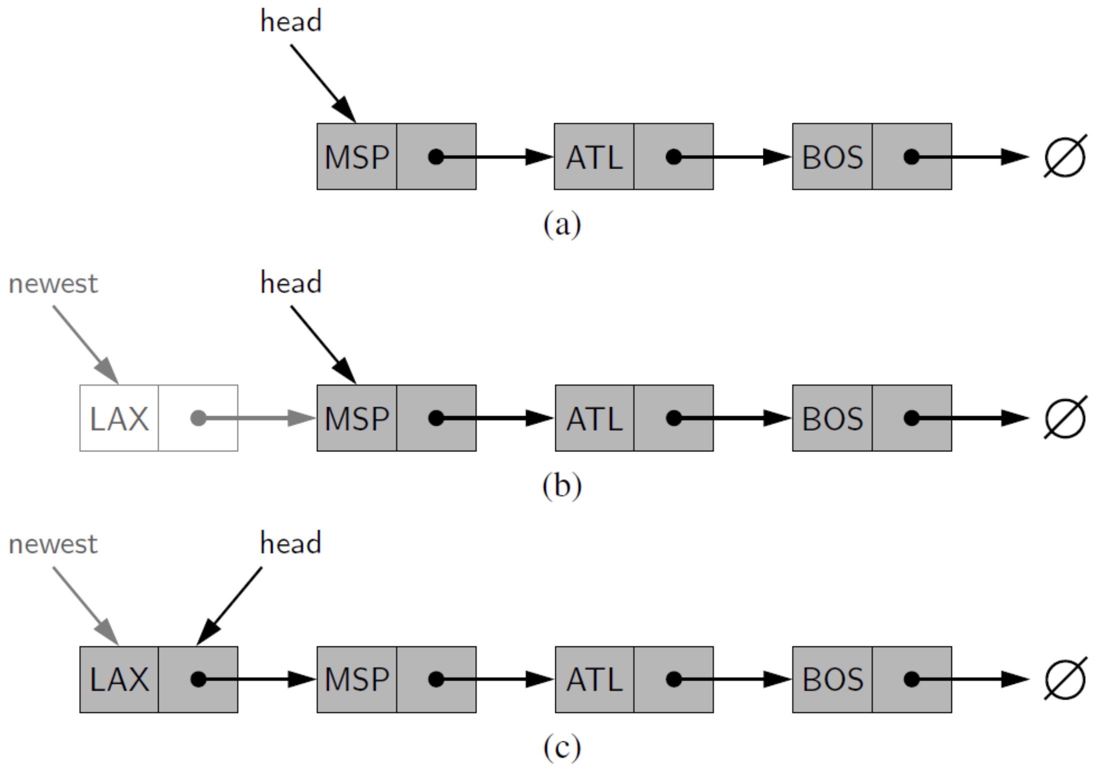
In [56]:
```
b.nextnode = c
```

Inserting an Element at the Head of a Singly Linked

List

An important property of a linked list is that it does not have a predetermined fixed size. It uses space proportionally to its current number of elements. To insert a new element at the head of the list:

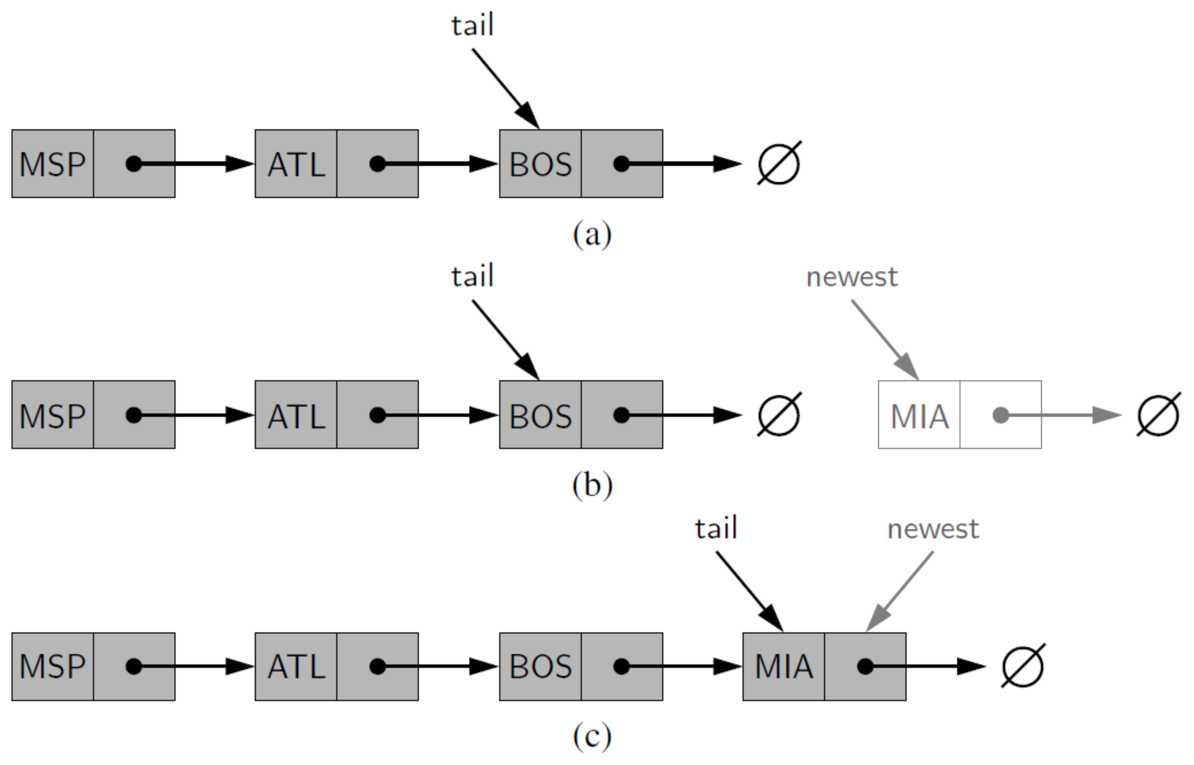
- We create a new node
- Set its element to the new element
- set its next link to refer to the current head then set the list's head to point to the new node.



Inserting an Element at the Tail of a Singly Linked List

We can also easily insert an element at the tail of the list, provided we keep a reference to the tail node

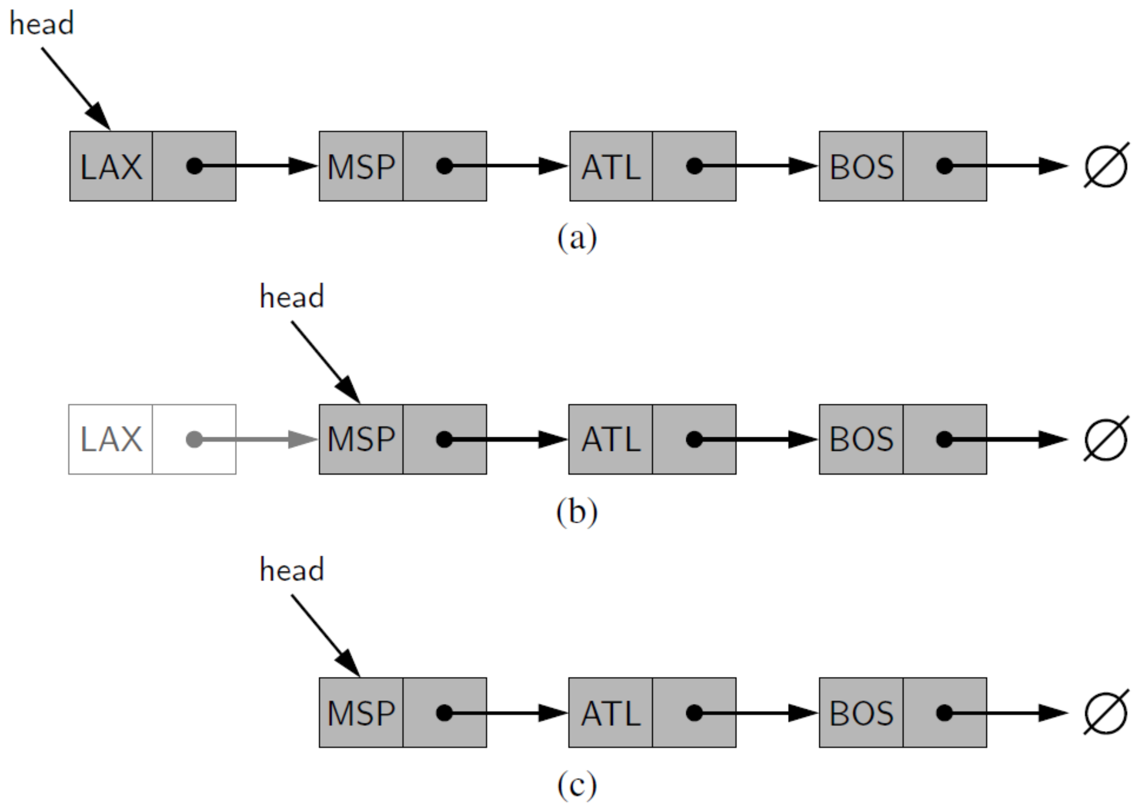
- Create a new node
- Assign its next reference to None
- Set the next reference of the tail to point to this new node
- Then update the tail reference itself to this new node.



Removing an Element from a Singly Linked List

Removing an element from the head of a singly linked list is essentially the reverse operation of inserting a new element at the head.

- We cannot easily delete the last node of a singly linked list.
- Even if we maintain a tail reference directly to the last node of the list, we must be able to access the node before the last node in order to remove the last node.
- But we cannot reach the node before the tail by following next links from the tail.
- If we want to support such an operation efficiently, we will need to make our list doubly linked.



In a Linked List the first node is called the **head** and the last node is called the **tail**. Let's discuss the pros and cons of Linked Lists:

Pros

- Linked Lists have constant-time insertions and deletions in any position, in comparison, arrays require O(n) time to do the same thing.
- Linked lists can continue to expand without having to specify their size ahead of time (remember our lectures on Array sizing form the Array Sequence section of the course!)

Cons

- To access an element in a linked list, you need to take O(k) time to go from the head of the list to the kth element. In contrast, arrays have constant time operations to access elements in an array.

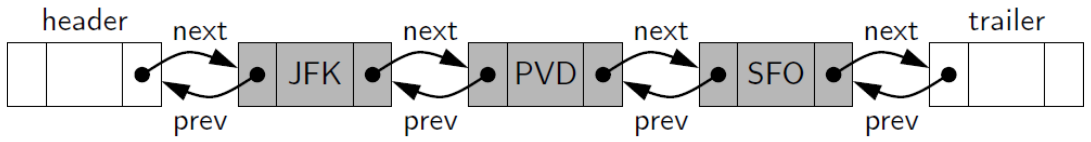
Doubly Linked List Implementation

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.

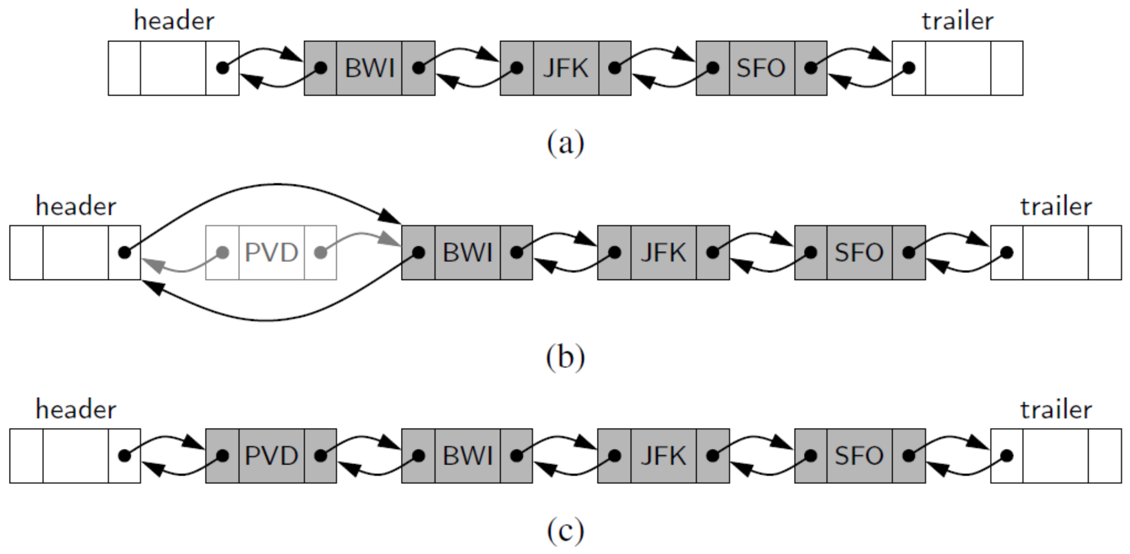
We add special nodes at both ends of the list.

- a header node at the beginning of the list
- a trailer node at the end of the list.
- These “dummy” nodes are known as sentinels (or guards)



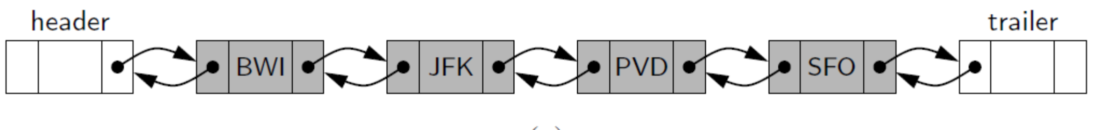
Inserting and Deleting with a Doubly Linked List

- Every insertion into our doubly linked list representation will take place between a pair of existing nodes
- When a new element is inserted at the front of the sequence, we will simply add the new node between the header and the node that is currently after the header.

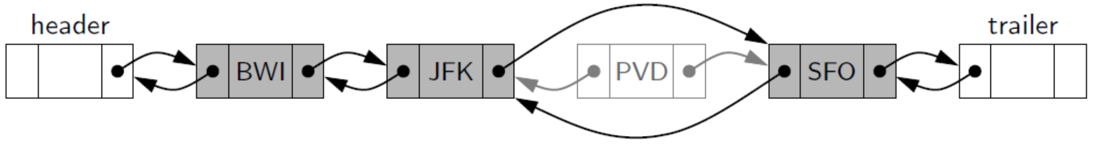


Deletion of a Node

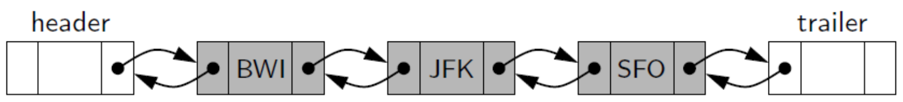
- The two neighbors of the node to be deleted are linked directly to each other
- As a result, that node will no longer be considered part of the list and it can be reclaimed by the system.
- Because of sentinels, the same implementation can be used when deleting the first or the last element of a sequence.



(a)



(b)



(c)

In [57]:
```
class DoublyLinkedListNode(object):

    def __init__(self,value):

        self.value = value
        self.next_node = None
        self.prev_node = None
```

Now that we have our node that can reference next *and* previous values, let's begin to build out our linked list!

In [58]:
```
a = DoublyLinkedListNode(1)
b = DoublyLinkedListNode(2)
c = DoublyLinkedListNode(3)
```

In [59]:
```
# Setting b after a
b.prev_node = a
a.next_node = b
```

In [60]:
```python
# Setting c after a
b.next_node = c
c.prev_node = b
```

Having a Doubly Linked list allows us to go though our Linked List forwards **and** backwards.

Linked List Nth to Last Node - SOLUTION

Write a function that takes a head node and an integer value **n** and then returns the nth to last node in the linked list. For example, given:

Solution

One approach to this problem is this:

Imagine you have a bunch of nodes and a "block" which is n-nodes wide. We could walk this "block" all the way down the list, and once the front of the block reached the end, then the other end of the block would be a the Nth node!

So to implement this "block" we would just have two pointers a left and right pair of pointers. Let's mark out the steps we will need to take:

- Walk one pointer **n** nodes from the head, this will be the right_point
- Put the other pointer at the head, this will be the left_point
- Walk/traverse the block (both pointers) towards the tail, one node at a time, keeping a distance **n** between them.
- Once the right_point has hit the tail, we know that the left point is at the target.

Let's see the code for this!

In [61]:
```python
def nth_to_last_node(n, head):

    left_pointer  = head
    right_pointer = head
    # Set right pointer at n nodes away from head
    for i in range(n-1):

        # Check for edge case of not having enough nodes!
        if not right_pointer.nextnode:
            raise LookupError('Error: n is larger than the linked lis
t.')

        # Otherwise, we can set the block
        right_pointer = right_pointer.nextnode

    # Move the block down the linked list
    while right_pointer.nextnode:
        left_pointer  = left_pointer.nextnode
        right_pointer = right_pointer.nextnode

    # Now return left pointer, its at the nth to last element!
    return left_pointer
```

In [62]:

```
a = Node(1)
b = Node(2)
c = Node(3)
d = Node(4)
e = Node(5)

a.nextnode = b
b.nextnode = c
c.nextnode = d
d.nextnode = e

# This would return the node d with a value of 4, because its the 2nd t
o last node.
target_node = nth_to_last_node(2, a)
```

In [63]:

```
target_node.value
```

Out[63]:

```
4
```

Linked List Reversal - SOLUTION

Write a function to reverse a Linked List in place. The function will take in the head of the list as input and return the new head of the list.

You are given the example Linked List Node class:

Solution

Since we want to do this in place we want to make the funciton operate in O(1) space, meaning we don't want to create a new list, so we will simply use the current nodes! Time wise, we can perform the reversal in O(n) time.

We can reverse the list by changing the next pointer of each node. Each node's next pointer should point to the previous node.

In one pass from head to tail of our input list, we will point each node's next pointer to the previous element.

Make sure to copy current.next_node into next_node **before** setting current.next_node to previous. Let's see this solution coded out:

In [64]:
```
def reverse(head):

    # Set up current,previous, and next nodes
    current = head
    previous = None
    nextnode = None

    # until we have gone through to the end of the list
    while current:

        # Make sure to copy the current nodes next node to a variable n
ext_node
        # Before overwriting as the previous node for reversal
        nextnode = current.nextnode
        # Reverse the pointer ot the next_node
        current.nextnode = previous

        # Go one forward in the list
        previous = current
        current = nextnode

    return previous
```

In [65]:
```
# Create a list of 4 nodes
a = Node(1)
b = Node(2)
c = Node(3)
d = Node(4)

# Set up order a,b,c,d with values 1,2,3,4
a.nextnode = b
b.nextnode = c
c.nextnode = d
```

Now let's check the values of the nodes coming after a, b and c:

In [66]:
```
print(a.nextnode.value)
print(b.nextnode.value)
print(c.nextnode.value)
```

```
2
3
4
```

In [67]:
```
#d.nextnode.value
```

So far so good. Note how there is no value proceeding the last node, this makes sense! Now let's reverse the linked list, we should see the opposite order of values!

In [68]:
```
reverse(a)
```

Out[68]:
```
<__main__.Node at 0x7f82b142aad0>
```

In [69]:
```
print(d.nextnode.value)
print(c.nextnode.value)
print(b.nextnode.value)
```

```
3
2
1
```

In [70]:
```
#a.nextnode.value
```

Singly Linked List Cycle Check

Given a singly linked list, write a function which takes in the first node in a singly linked list and returns a boolean indicating if the linked list contains a "cycle".

A cycle is when a node's next point actually points back to a previous node in the list. This is also sometimes known as a circularly linked list.

In [71]:
```
class Node(object):

    def __init__(self,value):

        self.value = value
        self.nextnode = None
```

Solution

To solve this problem we will have two markers traversing through the list. **marker1** and **marker2**. We will have both makers begin at the first node of the list and traverse through the linked list. However the second marker, marker2, will move two nodes ahead for every one node that marker1 moves.

By this logic we can imagine that the markers are "racing" through the linked list, with marker2 moving faster. If the linked list has a cylce and is circularly connected we will have the analogy of a track, in this case the marker2 will eventually be "lapping" the marker1 and they will equal each other.

If the linked list has no cycle, then marker2 should be able to continue on until the very end, never equaling the first marker.

Let's see this logic coded out:

In [72]:
```
def cycle_check(node):

    # Begin both markers at the first node
    marker1 = node
    marker2 = node

    # Go until end of list
    while marker2 != None and marker2.nextnode != None:

        # Note
        marker1 = marker1.nextnode
        marker2 = marker2.nextnode.nextnode

        # Check if the markers have matched
        if marker2 == marker1:
            return True

    # Case where marker ahead reaches the end of the list
    return False
```

In [73]:
```
"""
RUN THIS CELL TO TEST YOUR SOLUTION
"""
from nose.tools import assert_equal

# CREATE CYCLE LIST
a = Node(1)
b = Node(2)
```

```
c = Node(3)

a.nextnode = b
b.nextnode = c
c.nextnode = a # Cycle Here!


# CREATE NON CYCLE LIST
x = Node(1)
y = Node(2)
z = Node(3)

x.nextnode = y
y.nextnode = z


#############
class TestCycleCheck(object):

    def test(self,sol):
        assert_equal(sol(a),True)
        assert_equal(sol(x),False)

        print("ALL TEST CASES PASSED")

# Run Tests

t = TestCycleCheck()
t.test(cycle_check)
```

```
ALL TEST CASES PASSED
```

Introduction to Recursion

What is 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!

Why use Recursion?

Recursion provides a powerful alternative for performing repetitions of tasks in which a loop is not ideal. Most modern programming languages support recursion and recursion serves as a great tool for building out particular data structures.

We'll start this introduction with an example of recursion- a factorial function.

Factorial Example

In this part of the lecture we will explain recursion through an example excercise of creating the factorial function. The factorial function is denoted with an exclamation point and is defined as the product of the integers from 1 to *n*. Formally, we can state this as:

$$n! = n \cdot (n-1) \cdot (n-2) \ldots 3 \cdot 2 \cdot 1$$

Note, **if n = 0, then n! = 1**. This is important to take into account, because it will serve as our *base case*.

Take this example:

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24.$$

So how can we state this in a recursive manner? This is where the concept of **base case** comes in.

Base case is a key part of understanding recursion, especially when it comes to having to solve interview problems dealing with recursion. Let's rewrite the above equation of 4! so it looks like this:

$$4! = 4 \cdot (3 \cdot 2 \cdot 1) = 24$$

Notice that this is the same as:

$$4! = 4 \cdot 3! = 24$$

Meaning we can rewrite the formal recursion definition in terms of recursion like so:

$$n! = n \cdot (n-1)!$$

Note, **if n = 0, then n! = 1**. This means the **base case** occurs once n=0, the *recursive cases* are defined in the equation above. Whenever you are trying to develop a recursive solution it is very important to think about the base case, as your solution will need to return the base case once all the recursive cases have been worked through. Let's look at how we can create the factorial function in Python:

In [74]:
```
def fact(n):
    '''
    Returns factorial of n (n!).
    Note use of recursion
    '''
    # BASE CASE!
    if n == 0:
        return 1

    # Recursion!
    else:
        return n * fact(n-1)
```

Let's see it in action!

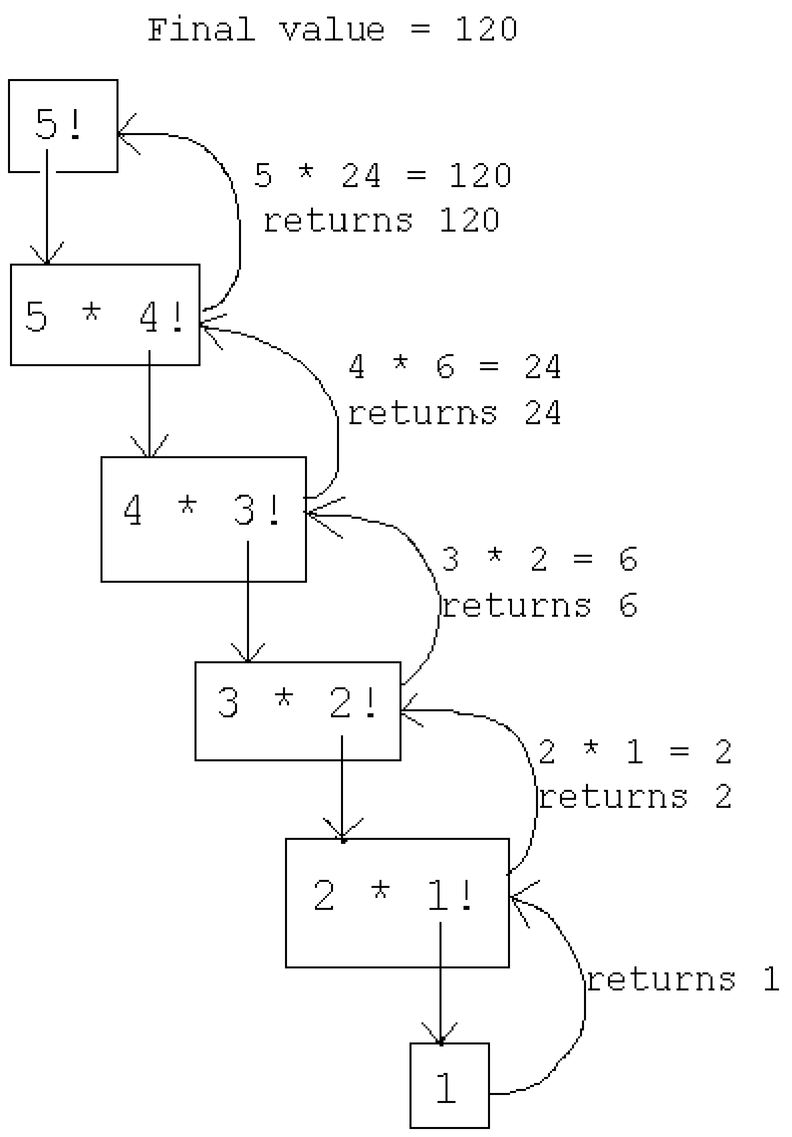
In [75]:
```
fact(5)
```

Out[75]:
```
120
```

Note how we had an if statement to check if a base case occured. Without it this function would not have successfully completed running. We can visualize the recursion with the following figure:

In [76]:
```
from IPython.display import Image
from IPython.core.display import HTML
Image(url= 'http://faculty.cs.niu.edu/~freedman/241/241notes/recur.gif'
)
```

Out[76]:



We can follow this flow chart from the top, reaching the base case, and then working our way back up.

Conclusion

Recursion is a powerful tool, but it can be a tricky concept to implement. In the next lectures we will go over a few more example problems for recursion. Then afterwards you'll be faced with some real interview questions involving recursion!

Memoization

In this lecture we will discuss memoization and dynamic programming. For your homework assignment, read the Wikipedia article on Memoization, before continuing on with this lecture!

Memoization effectively refers to remembering ("memoization" -> "memorandum" -> to be remembered) results of method calls based on the method inputs and then returning the remembered result rather than computing the result again. You can think of it as a cache for method results. We'll use this in some of the interview problems as improved versions of a purely recursive solution.

A simple example for computing factorials using memoization in Python would be something like this:

In [77]:

```python
# Create cache for known results
factorial_memo = {}

def factorial(k):

    if k < 2:
        return 1

    if not k in factorial_memo:
        factorial_memo[k] = k * factorial(k-1)

    return factorial_memo[k]
```

In [78]:

```python
factorial(4)
```

Out[78]: 24

Note how we are now using a dictionary to store previous results of the factorial function! We are now able to increase the efficiency of this function by remembering old results!

Keep this in mind when working on the Coin Change Problem and the Fibonacci Sequence Problem.

We can also encapsulate the memoization process into a class:

In [79]:
```python
class Memoize:
    def __init__(self, f):
        self.f = f
        self.memo = {}
    def __call__(self, *args):
        if not args in self.memo:
            self.memo[args] = self.f(*args)
        return self.memo[args]
```

Then all we would have to do is:

In [80]:
```python
def factorial(k):

    if k < 2:
        return 1

    return k * factorial(k - 1)

factorial = Memoize(factorial)
```

Try comparing the run times of the memoization versions of functions versus the normal recursive solutions!

Recursion Homework Problems - Solutions

Problem 1

Write a recursive function which takes an integer and computes the cumulative sum of 0 to that integer

For example, if n=4 , return 4+3+2+1+0, which is 10.

This problem is very similar to the factorial problem presented during the introduction to recursion. Remember, always think of what the base case will look like. In this case, we have a base case of n =0 (Note, you could have also designed the cut off to be 1).

In this case, we have: n + (n-1) + (n-2) + ... + 0

```
In [81]: def rec_sum(n):

             # Base Case
             if n == 0:
                 return 0

             # Recursion
             else:
                 return n + rec_sum(n-1)
```

```
In [82]: rec_sum(4)

Out[82]: 10
```

Problem 2

Given an integer, create a function which returns the sum of all the individual digits in that integer. For example: if n = 4321, return 4+3+2+1

```
In [85]: def sum_func(n):
             if n ==0:
                 return n
             else:
```

```
        m = n%10
        n = n//10
        return m+ sum_func(n)
```

In [86]:
```
sum_func(4321)
```

Out[86]:
```
10
```

Problem 3

Note, this is a more advanced problem than the previous two! It aso has a lot of variation possibilities and we're ignoring strict requirements here.

Create a function called word_split() which takes in a string **phrase** and a set **list_of_words**. The function will then determine if it is possible to split the string in a way in which words can be made from the list of words. You can assume the phrase will only contain words found in the dictionary if it is completely splittable.

In [87]:
```
def word_split(phrase,list_of_words, output = None):
    '''
    Note: This is a very "python-y" solution.
    '''

    # Checks to see if any output has been initiated.
    # If you default output=[], it would be overwritten for every recur
sion!
    if output is None:
        output = []

    # For every word in list
    for word in list_of_words:

        # If the current phrase begins with the word, we have a split p
oint!
        if phrase.startswith(word):

            # Add the word to the output
            output.append(word)

            # Recursively call the split function on the remaining port
ion of the phrase--- phrase[len(word):]
            # Remember to pass along the output and list of words
            return word_split(phrase[len(word):],list_of_words,output)

    # Finally return output if no phrase.startswith(word) returns True
    return output
```

In [88]:
```
word_split('themanran',['the','ran','man'])
```

Out[88]:
```
['the', 'man', 'ran']
```

In [89]:
```
word_split('ilovedogsJohn',['i','am','a','dogs','lover','love','John'])
```

Out[89]:
```
['i', 'love', 'dogs', 'John']
```

In [90]:
```
word_split('themanran',['clown','ran','man'])
```

Out[90]:
```
[]
```

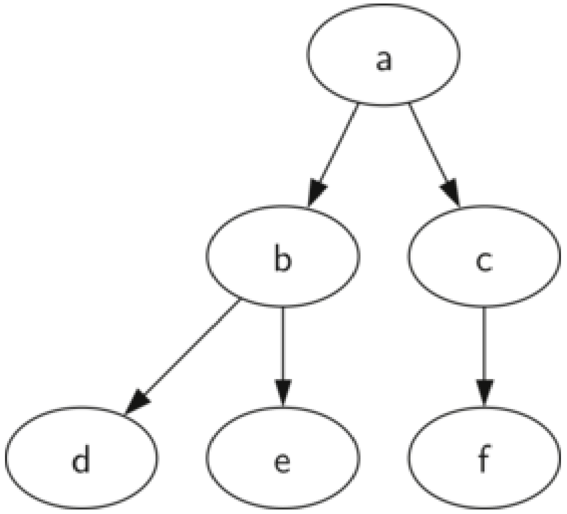
Conclusion

Alright, so now that we've seen a few examples, let's dive in to the interview practice problems!

Tree

- 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.

```
g = {'A':set(['B','C']),
     'B':set(['A','D','E']),
     'C':set(['A','F']),
     'D':set(['B']),
     'E':set(['B','F']),
     'F':set(['C','E'])
     }
```



Tree Representation Implementation (Lists)

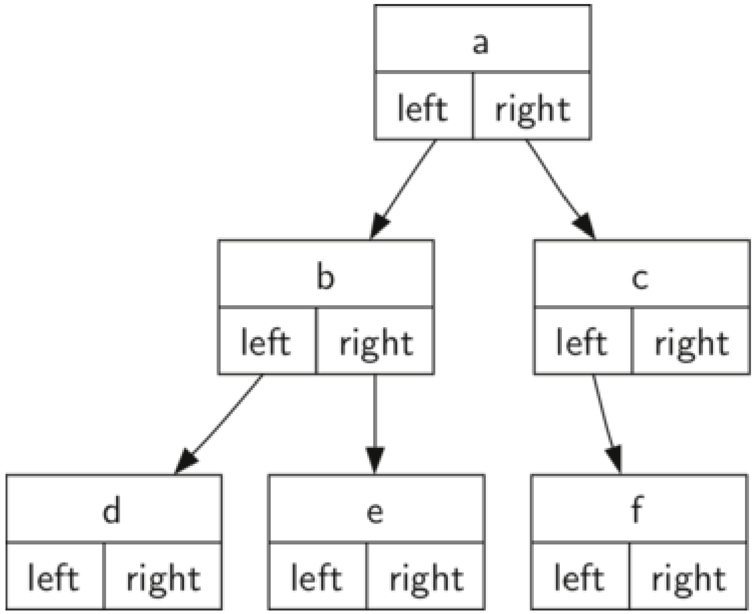
Below is a representation of a Tree using a list of lists. Refer to the video lecture for an explanation and a live coding demonstration!

In [91]:
```
def BinaryTree(r):
    return [r, [], []]

def insertLeft(root,newBranch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch, [], []])
    return root
```

```
def insertRight(root,newBranch):
    t = root.pop(2)
    if len(t) > 1:
        root.insert(2,[newBranch,[],t])
    else:
        root.insert(2,[newBranch,[],[]])
    return root

def getRootVal(root):
    return root[0]

def setRootVal(root,newVal):
    root[0] = newVal

def getLeftChild(root):
    return root[1]

def getRightChild(root):
    return root[2]
```

Nodes and References Implementation of a Tree

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.



In [92]:
```python
class BinaryTree(object):
    def __init__(self,rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None

    def insertLeft(self,newNode):
        if self.leftChild == None:
            self.leftChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.leftChild = self.leftChild
            self.leftChild = t

    def insertRight(self,newNode):
        if self.rightChild == None:
            self.rightChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.rightChild = self.rightChild
            self.rightChild = t


    def getRightChild(self):
        return self.rightChild

    def getLeftChild(self):
        return self.leftChild

    def setRootVal(self,obj):
        self.key = obj

    def getRootVal(self):
        return self.key
```

We can see some examples of creating a tree and assigning children. Note that some outputs are Trees themselves!

In [93]:

```
from __future__ import print_function

r = BinaryTree('a')
print(r.getRootVal())
print(r.getLeftChild())
r.insertLeft('b')
print(r.getLeftChild())
print(r.getLeftChild().getRootVal())
r.insertRight('c')
print(r.getRightChild())
print(r.getRightChild().getRootVal())
r.getRightChild().setRootVal('hello')
print(r.getRightChild().getRootVal())
```

```
a
None
<__main__.BinaryTree object at 0x7f8283e0f490>
b
<__main__.BinaryTree object at 0x7f8283e0f710>
```

```
c
hello
```

Tree Traversals

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.

```
In [98]: r = BinaryTree('a')
```

```
In [99]: r.getRootVal()
```

```
Out[99]: 'a'
```

```
In [100]: r.insertLeft('b')
          r.insertRight('c')
          r.insertLeft('d')
          r.insertRight('e')
```

```
In [101]: r.getRightChild().getRootVal()
```

```
Out[101]: 'e'
```

```
In [102]: r.getLeftChild().getRootVal()
```

```
Out[102]: 'd'
```

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.

```
In [50]: def preorder(tree):
    if tree != None:
        print(tree.getRootVal())
        preorder(tree.getLeftChild())
        preorder(tree.getRightChild())
```

In [51]:
```
preorder(r)
```

```
a
d
b
e
c
```

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.

In [52]:
```
def inorder(tree):
    if tree != None:
        inorder(tree.getLeftChild())
        print(tree.getRootVal())
        inorder(tree.getRightChild())
```

In [53]:
```
inorder(r)
```

```
b
d
a
e
c
```

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.

In [48]:
```
def postorder(tree):
    if tree != None:
        postorder(tree.getLeftChild())
        postorder(tree.getRightChild())
        print(tree.getRootVal())
```

In [49]:
```
postorder(r)
```

```
b
d
c
e
a
```

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.

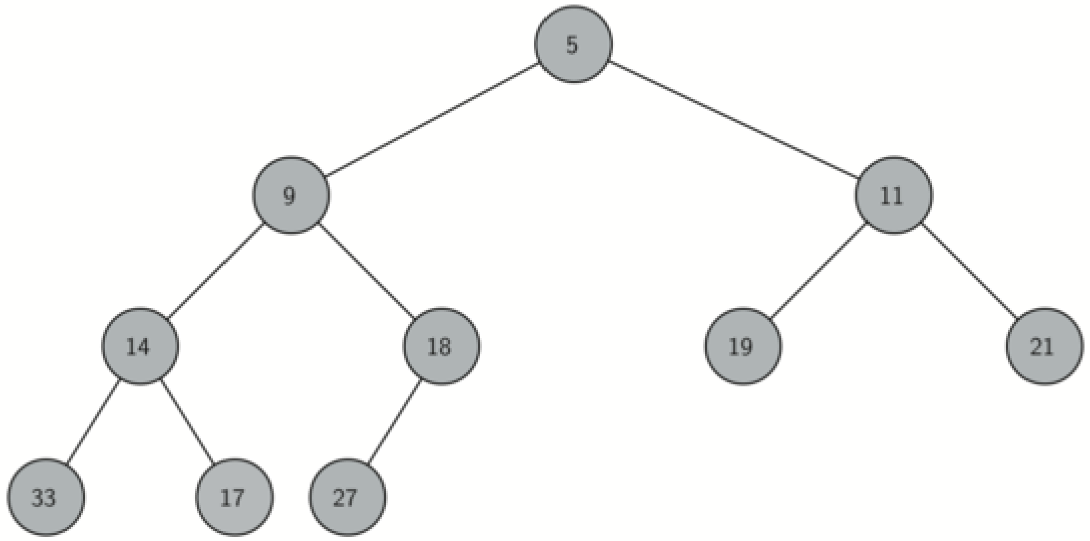
Binary Heaps

- 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(logn).

- 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.

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.

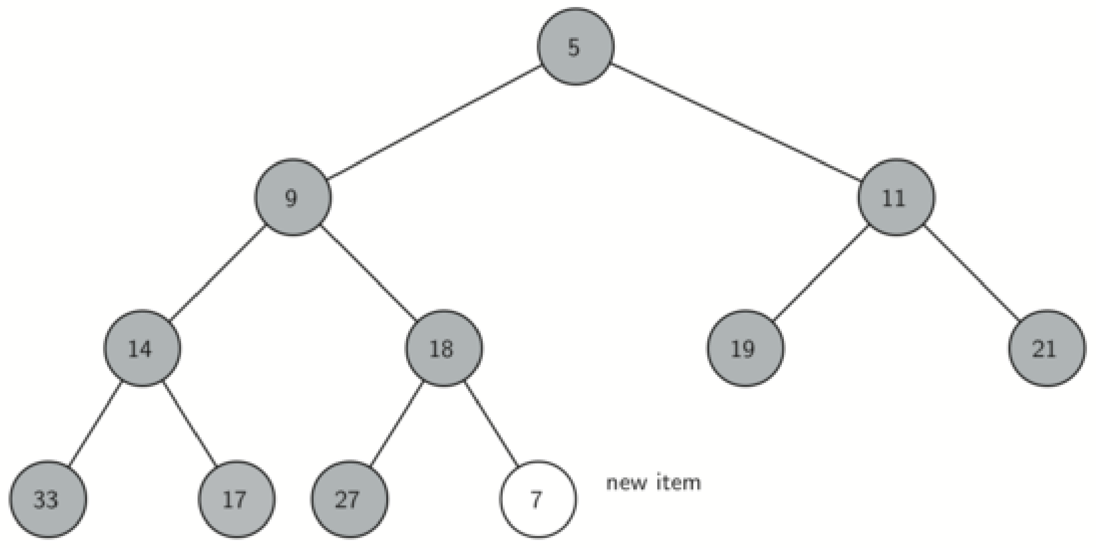


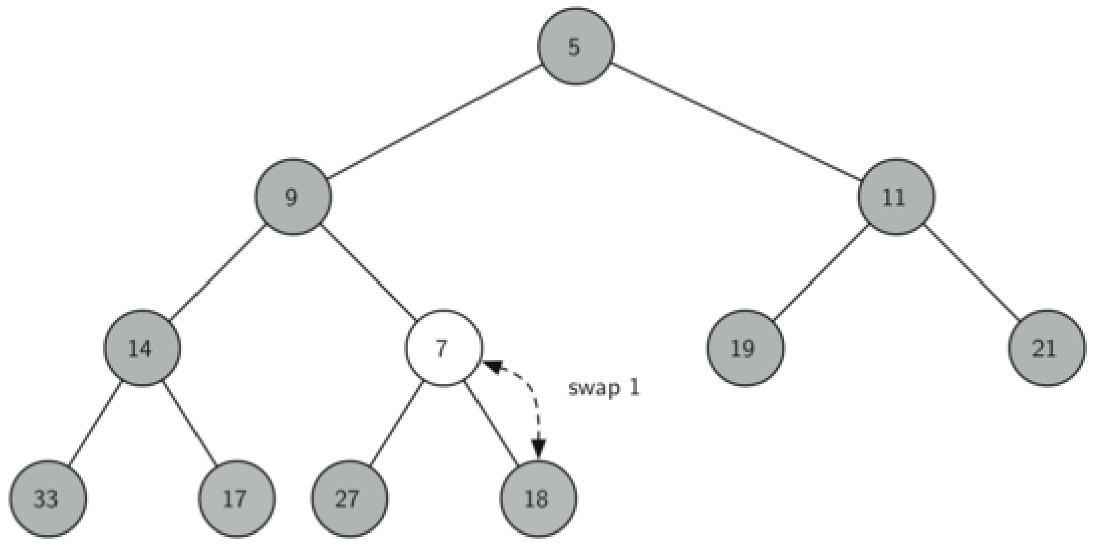
`__init__`

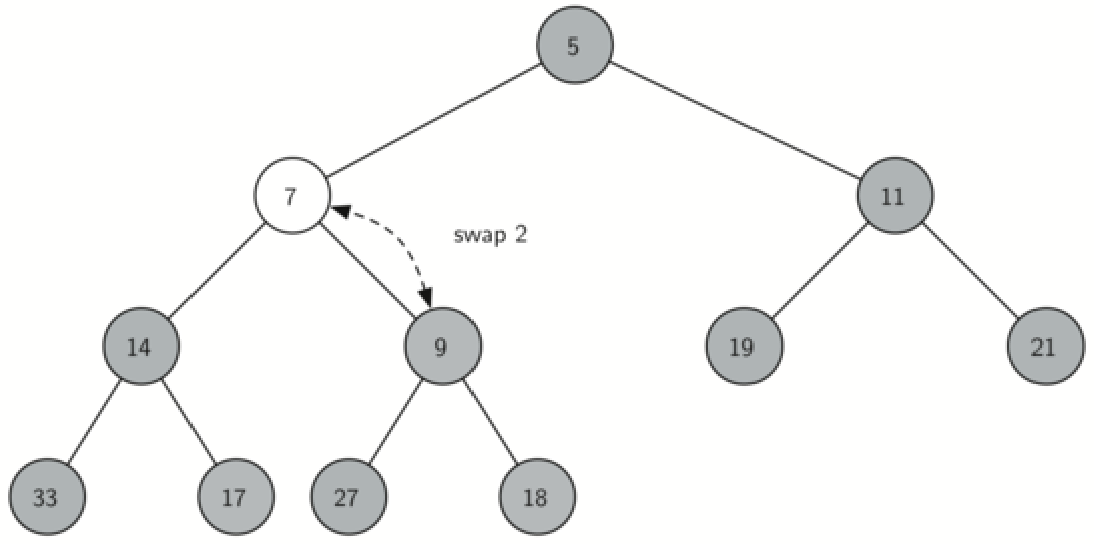
Start off with our list representation code

`insert`

- 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!




5
9
11
14
7
19
21
33
17
27
18
swap 1


5
7
11
14
9
19
21
33
17
27
18
swap 2

- 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.

Methods for insertion

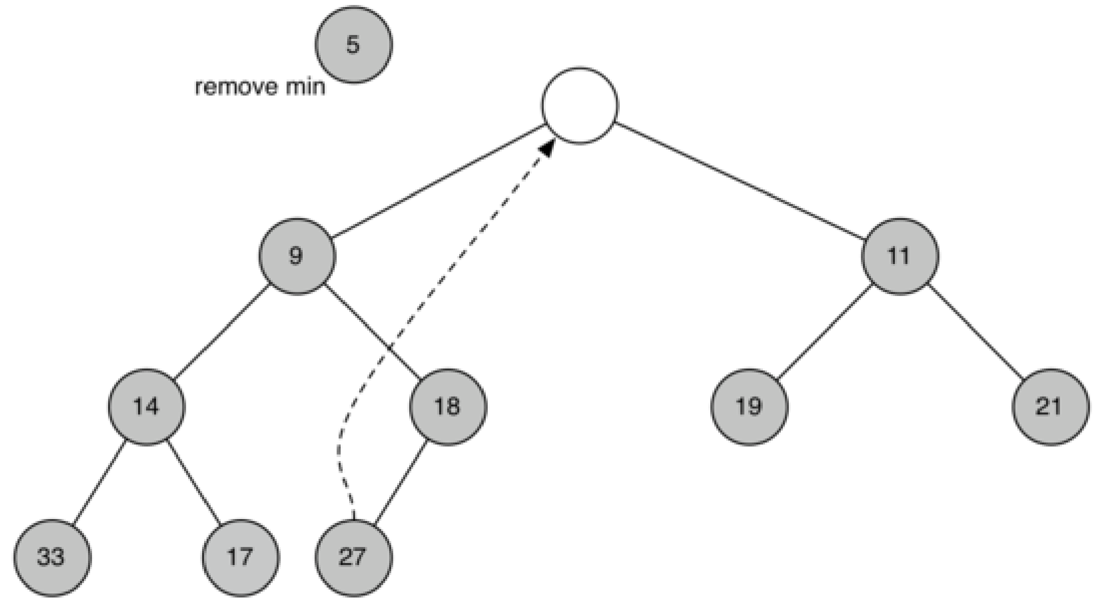
`insert` `PerUp`

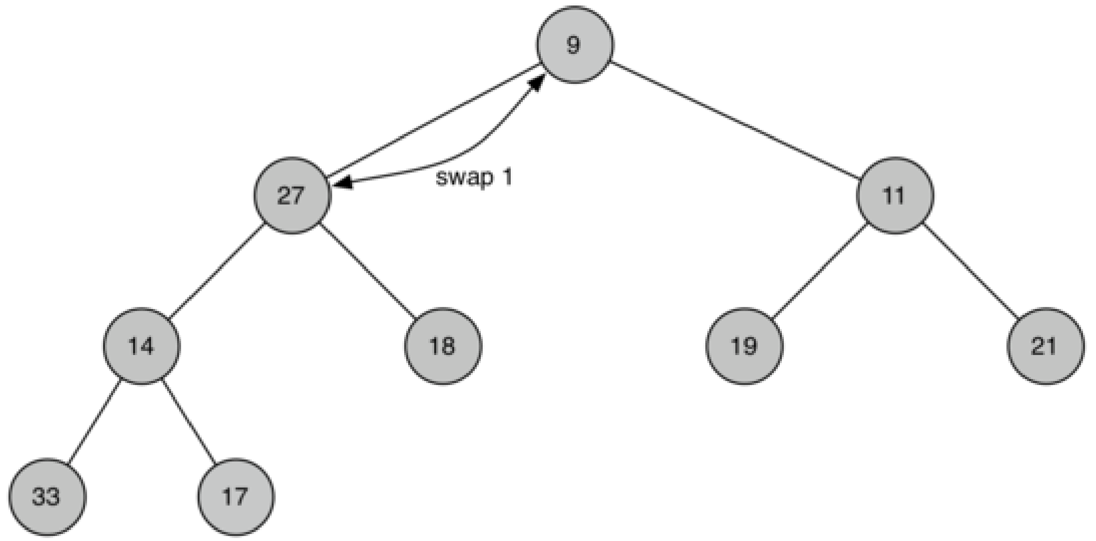
`delMin` `PerDown`

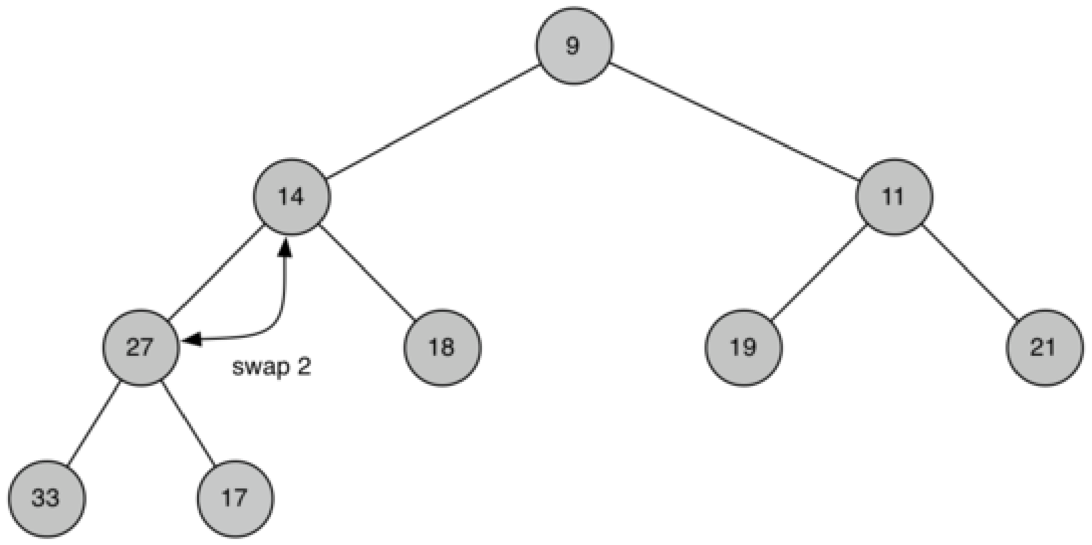
- Since the heap property requires that the root of the tree be the smallest item in the tree, finding the minimum item is easy.
- The hard part of delMin is restoring full compliance with the heap structure and heap order properties after the root has been removed.

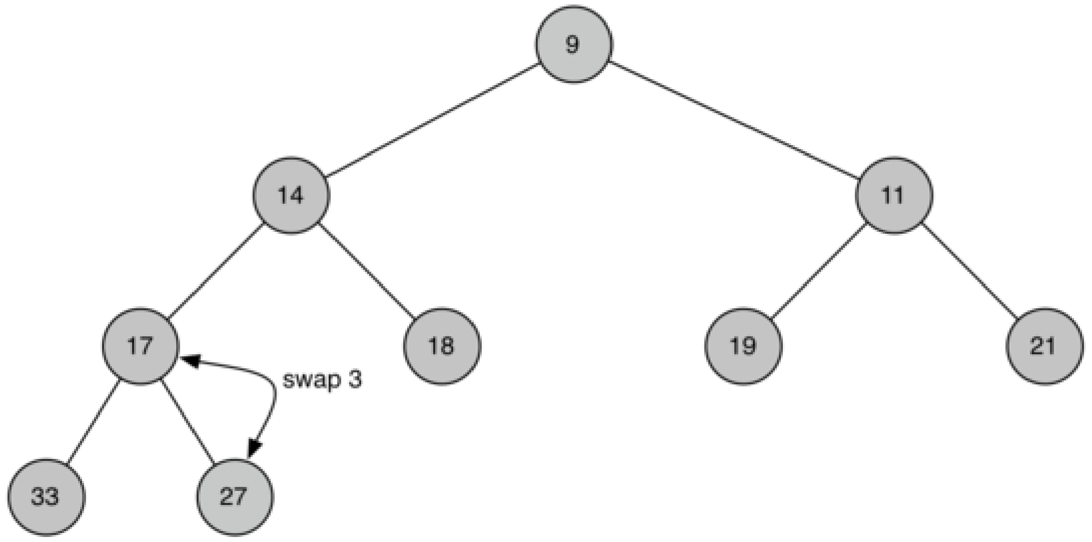
We can restore our heap in two steps.

- 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.


5
remove min
9
11
14
18
19
21
33
17
27


9
27
swap 1
11
14
18
19
21
33
17


9
14
11
27
18
19
21
33
17
swap 2

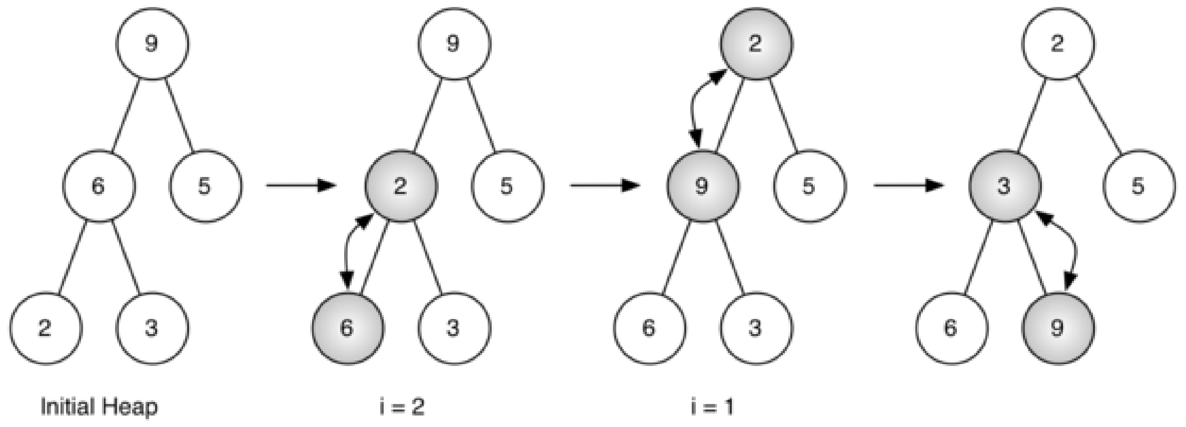

9
14
11
17
18
19
21
33
27
swap 3

- 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.

`buildHeap`

- 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(logn) 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(nlogn) operations.
- However, if we start with an entire list then we can build the whole heap in O(n) operations.

Heap from list of [9,6,5,2,3]



Binary Heap Operations

The basic operations we will implement for our binary heap are as follows:

- BinaryHeap() creates a new, empty, binary heap.
- insert(k) adds a new item to the heap.
- findMin() returns the item with the minimum key value, leaving item in the heap.
- delMin() returns the item with the minimum key value, removing the item from the heap.
- isEmpty() returns true if the heap is empty, false otherwise.
- size() returns the number of items in the heap.
- buildHeap(list) builds a new heap from a list of keys.

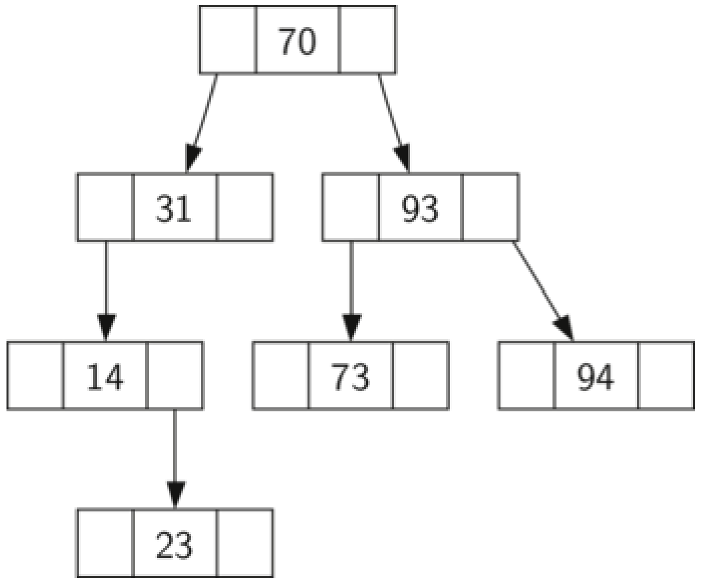
In [1]:
```
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0


    def percUp(self,i):

        while i // 2 > 0:

            if self.heapList[i] < self.heapList[i // 2]:


                tmp = self.heapList[i // 2]
                self.heapList[i // 2] = self.heapList[i]
                self.heapList[i] = tmp
            i = i // 2

    def insert(self,k):

        self.heapList.append(k)
        self.currentSize = self.currentSize + 1
        self.percUp(self.currentSize)

    def percDown(self,i):

        while (i * 2) <= self.currentSize:

            mc = self.minChild(i)
            if self.heapList[i] > self.heapList[mc]:

                tmp = self.heapList[i]
                self.heapList[i] = self.heapList[mc]
                self.heapList[mc] = tmp
            i = mc

    def minChild(self,i):

        if i * 2 + 1 > self.currentSize:

            return i * 2
        else:

            if self.heapList[i*2] < self.heapList[i*2+1]:
                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

    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
```

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.



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.

In [1]:
```
class TreeNode:

    def __init__(self,key,val,left=None,right=None,parent=None):
        self.key = key
        self.payload = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent

    def hasLeftChild(self):
        return self.leftChild
    def hasRightChild(self):
        return self.rightChild

    def isLeftChild(self):
        return self.parent and self.parent.leftChild == self

    def isRightChild(self):
        return self.parent and self.parent.rightChild == self

    def isRoot(self):
        return not self.parent

    def isLeaf(self):
        return not (self.rightChild or self.leftChild)

    def hasAnyChildren(self):
        return self.rightChild or self.leftChild

    def hasBothChildren(self):
        return self.rightChild and self.leftChild

    def replaceNodeData(self,key,value,lc,rc):
        self.key = key
        self.payload = value
        self.leftChild = lc
        self.rightChild = rc
        if self.hasLeftChild():
            self.leftChild.parent = self
        if self.hasRightChild():
            self.rightChild.parent = self
```

In [2]:
```
class BinarySearchTree:

    def __init__(self):
        self.root = None
        self.size = 0

    def length(self):
        return self.size

    def __len__(self):
        return self.size

    def put(self,key,val):
        if self.root:
            self._put(key,val,self.root)
        else:
            self.root = TreeNode(key,val)
        self.size = self.size + 1

    def _put(self,key,val,currentNode):
        if key < currentNode.key:
            if currentNode.hasLeftChild():
                   self._put(key,val,currentNode.leftChild)
            else:
                   currentNode.leftChild = TreeNode(key,val,parent=curr
entNode)
        else:
            if currentNode.hasRightChild():
                   self._put(key,val,currentNode.rightChild)
            else:
                   currentNode.rightChild = TreeNode(key,val,parent=cur
rentNode)

    def __setitem__(self,k,v):
        self.put(k,v)

    def get(self,key):
        if self.root:
            res = self._get(key,self.root)
            if res:

                return res.payload
            else:
                return None
        else:
            return None
```

```
def _get(self,key,currentNode):

    if not currentNode:
        return None
    elif currentNode.key == key:
        return currentNode
    elif key < currentNode.key:
        return self._get(key,currentNode.leftChild)
    else:
        return self._get(key,currentNode.rightChild)

def __getitem__(self,key):
    return self.get(key)

def __contains__(self,key):
    if self._get(key,self.root):
        return True
    else:
        return False

def delete(self,key):

    if self.size > 1:

        nodeToRemove = self._get(key,self.root)
        if nodeToRemove:
            self.remove(nodeToRemove)
            self.size = self.size-1
        else:
            raise KeyError('Error, key not in tree')
    elif self.size == 1 and self.root.key == key:
        self.root = None
        self.size = self.size - 1
    else:
        raise KeyError('Error, key not in tree')

def __delitem__(self,key):
    self.delete(key)

def spliceOut(self):
    if self.isLeaf():
        if self.isLeftChild():

            self.parent.leftChild = None
        else:
            self.parent.rightChild = None
    elif self.hasAnyChildren():
        if self.hasLeftChild():

            if self.isLeftChild():

                self.parent.leftChild = self.leftChild
            else:

                self.parent.rightChild = self.leftChild
                self.leftChild.parent = self.parent
    else:

        if self.isLeftChild():

            self.parent.leftChild = self.rightChild
        else:
            self.parent.rightChild = self.rightChild
            self.rightChild.parent = self.parent

def findSuccessor(self):

    succ = None
    if self.hasRightChild():
        succ = self.rightChild.findMin()
    else:
        if self.parent:

            if self.isLeftChild():

                succ = self.parent
            else:
                self.parent.rightChild = None
                succ = self.parent.findSuccessor()
                self.parent.rightChild = self
    return succ

def findMin(self):

    current = self
    while current.hasLeftChild():
        current = current.leftChild
    return current

def remove(self,currentNode):

    if currentNode.isLeaf(): #leaf
        if currentNode == currentNode.parent.leftChild:
            currentNode.parent.leftChild = None
        else:
            currentNode.parent.rightChild = None
    elif currentNode.hasBothChildren(): #interior

        succ = currentNode.findSuccessor()
        succ.spliceOut()
        currentNode.key = succ.key
        currentNode.payload = succ.payload

    else: # this node has one child
        if currentNode.hasLeftChild():
            if currentNode.isLeftChild():
                currentNode.leftChild.parent = currentNode.parent
                currentNode.parent.leftChild = currentNode.leftChil
d
            elif currentNode.isRightChild():
                currentNode.leftChild.parent = currentNode.parent
                currentNode.parent.rightChild = currentNode.leftChi
ld
            else:
                currentNode.replaceNodeData(currentNode.leftChild.k
ey,
                                            currentNode.leftChild.payload,
                                            currentNode.leftChild.leftChild,
                                            currentNode.leftChild.rightChild)
        else:

            if currentNode.isLeftChild():
                currentNode.rightChild.parent = currentNode.parent
                currentNode.parent.leftChild = currentNode.rightChi
ld
            elif currentNode.isRightChild():
                currentNode.rightChild.parent = currentNode.parent
                currentNode.parent.rightChild = currentNode.rightCh
ild
            else:
                currentNode.replaceNodeData(currentNode.rightChild.
key,
                                            currentNode.rightChild.payload,
                                            currentNode.rightChild.leftChild,
                                            currentNode.rightChild.rightChild)
```

In [6]:
```
mytree = BinarySearchTree()
mytree[3]="red"
mytree[4]="blue"
mytree[6]="yellow"
mytree[2]="at"

print(mytree[6])
print(mytree[2])
```

```
yellow
at
```

Binary Search Tree Check

Given a binary tree, check whether it’s a binary search tree or not.

Again, no solution cell, just worry about your code making sense logically. Hint: Think about tree traversals.

Solution

Here is a simple solution- If a tree is a binary search tree, then traversing the tree inorder should lead to sorted order of the values in the tree. So, we can perform an inorder traversal and check whether the node values are sorted or not.

In [7]:
```
tree_vals = []

def inorder(tree):
    if tree != None:
        inorder(tree.getLeftChild())
        tree_vals.append(tree.getRootVal())
        inorder(tree.getRightChild())

def sort_check(tree_vals):
    return tree_vals == sorted(tree_vals)

inorder(tree)
sort_check(tree_vals)
```

```
True
```

Another classic solution is to keep track of the minimum and maximum values a node can take. And at each node we will check whether its value is between the min and max values it’s allowed to take. The root can take any value between negative infinity and positive infinity. At any node, its left child should be smaller than or equal than its own value, and similarly the right child should be larger than or equal to. So during recursion, we send the current value as the new max to our left child and send the min as it is without changing. And to the right child, we send the current value as the new min and send the max without changing.

In [8]:
```
class Node:
    def __init__(self, k, val):
        self.key = k
        self.value = val
        self.left = None
        self.right = None

def tree_max(node):
    if not node:
        return float("-inf")
    maxleft  = tree_max(node.left)
    maxright = tree_max(node.right)
    return max(node.key, maxleft, maxright)

def tree_min(node):
    if not node:
        return float("inf")
    minleft  = tree_min(node.left)
    minright = tree_min(node.right)
    return min(node.key, minleft, minright)

def verify(node):
    if not node:
        return True
    if (tree_max(node.left) <= node.key <= tree_min(node.right) and
        verify(node.left) and verify(node.right)):
        return True
    else:
        return False

root= Node(10, "Hello")
root.left = Node(5, "Five")
root.right= Node(30, "Thirty")

print(verify(root)) # prints True, since this tree is valid

root = Node(10, "Ten")
root.right = Node(20, "Twenty")
root.left = Node(5, "Five")
root.left.right = Node(15, "Fifteen")

print(verify(root)) # prints False, since 15 is to the left of 10
```

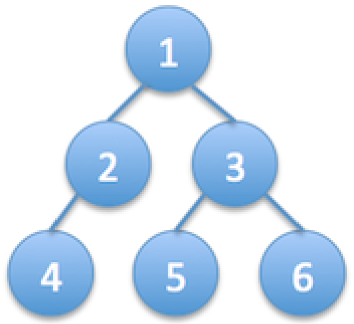
```
True
False
```

This is a classic interview problem, so feel free to just Google search "Validate BST" for more information on this problem!

```
In [ ]:
```

Tree Level Order Print

Given a binary tree of integers, print it in level order. The output will contain space between the numbers in the same level, and new line between different levels. For example, if the tree is:



The output should be:

```
1
2 3
4 5 6
```

Solution

It won’t be practical to solve this problem using recursion, because recursion is similar to depth first search, but what we need here is breadth first search. So we will use a queue as we did previously in breadth first search. First, we’ll push the root node into the queue. Then we start a while loop with the condition queue not being empty. Then, at each iteration we pop a node from the beginning of the queue and push its children to the end of the queue. Once we pop a node we print its value and space.

To print the new line in correct place we should count the number of nodes at each level. We will have 2 counts, namely current level count and next level count. Current level count indicates how many nodes should be printed at this level before printing a new line. We decrement it every time we pop an element from the queue and print it. Once the current level count reaches zero we print a new line. Next level count contains the number of nodes in the next level, which will become the current level count after printing a new line. We count the number of nodes in the next level by counting the number of children of the nodes in the current level. Understanding the code is easier than its explanation:

In [1]:
```
class Node:
    def __init__(self, val=None):
        self.left = None
        self.right = None
        self.val =  val
```

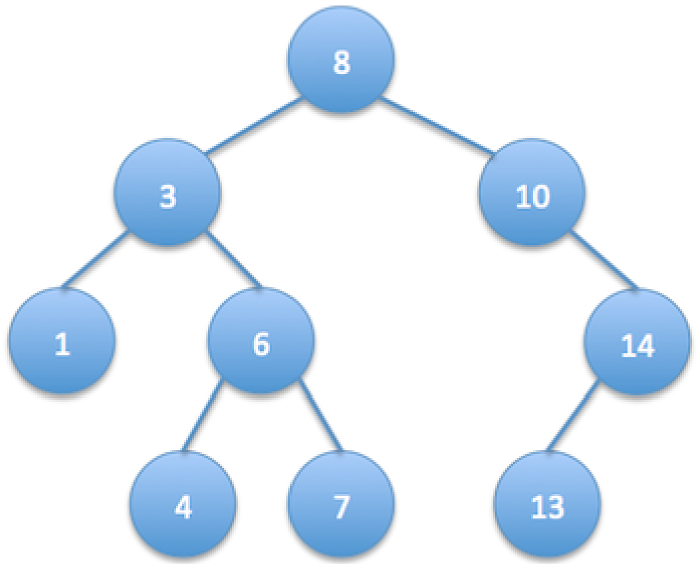
In [3]:
```
def levelOrderPrint(tree):
    if not tree:
        return
    nodes=collections.deque([tree])
    currentCount, nextCount = 1, 0
    while len(nodes)!=0:
        currentNode=nodes.popleft()
        currentCount-=1
        print(currentNode.val)
        if currentNode.left:
            nodes.append(currentNode.left)
            nextCount+=1
        if currentNode.right:
            nodes.append(currentNode.right)
            nextCount+=1
        if currentCount==0:
            #finished printing current level
            print('\n')
            currentCount, nextCount = nextCount, currentCount
```

The time complexity of this solution is O(N), which is the number of nodes in the tree, so it’s optimal. Because we should visit each node at least once. The space complexity depends on maximum size of the queue at any point, which is the most number of nodes at one level. The worst case occurs when the tree is a complete binary tree, which means each level is completely filled with maximum number of nodes possible. In this case, the most number of nodes appear at the last level, which is (N+1)/2 where N is the total number of nodes. So the space complexity is also O(N). Which is also optimal while using a queue.

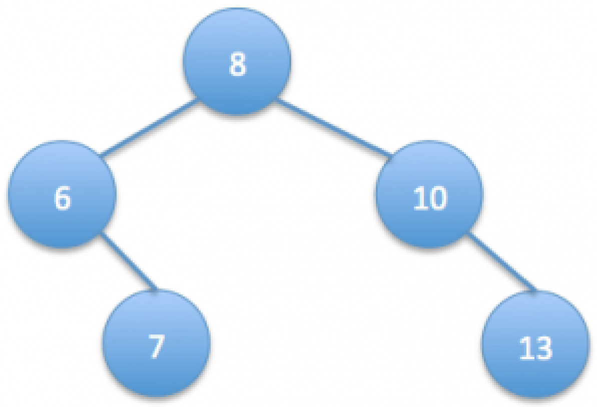
Again, this is a very common tree interview question!

Trim a Binary Search Tree

Given the root of a binary search tree and 2 numbers min and max, trim the tree such that all the numbers in the new tree are between min and max (inclusive). The resulting tree should still be a valid binary search tree. So, if we get this tree as input:



and we’re given **min value as 5** and **max value as 13**, then the resulting binary search tree should be:



We should remove all the nodes whose value is not between min and max.

Solution

We can do this by performing a post-order traversal of the tree. We first process the left children, then right children, and finally the node itself. So we form the new tree bottom up, starting from the leaves towards the root. As a result while processing the node itself, both its left and right subtrees are valid trimmed binary search trees (may be NULL as well).

At each node we’ll return a reference based on its value, which will then be assigned to its parent’s left or right child pointer, depending on whether the current node is left or right child of the parent. If current node’s value is between min and max (min<=node<=max) then there’s no action need to be taken, so we return the reference to the node itself. If current node’s value is less than min, then we return the reference to its right subtree, and discard the left subtree. Because if a node’s value is less than min, then its left children are definitely less than min since this is a binary search tree. But its right children may or may not be less than min we can’t be sure, so we return the reference to it. Since we’re performing bottom-up post-order traversal, its right subtree is already a trimmed valid binary search tree (possibly NULL), and left subtree is definitely NULL because those nodes were surely less than min and they were eliminated during the post-order traversal. Remember that in post-order traversal we first process all the children of a node, and then finally the node itself.

Similar situation occurs when node’s value is greater than max, we now return the reference to its left subtree. Because if a node’s value is greater than max, then its right children are definitely greater than max. But its left children may or may not be greater than max. So we discard the right subtree and return the reference to the already valid left subtree. The code is easier to understand:

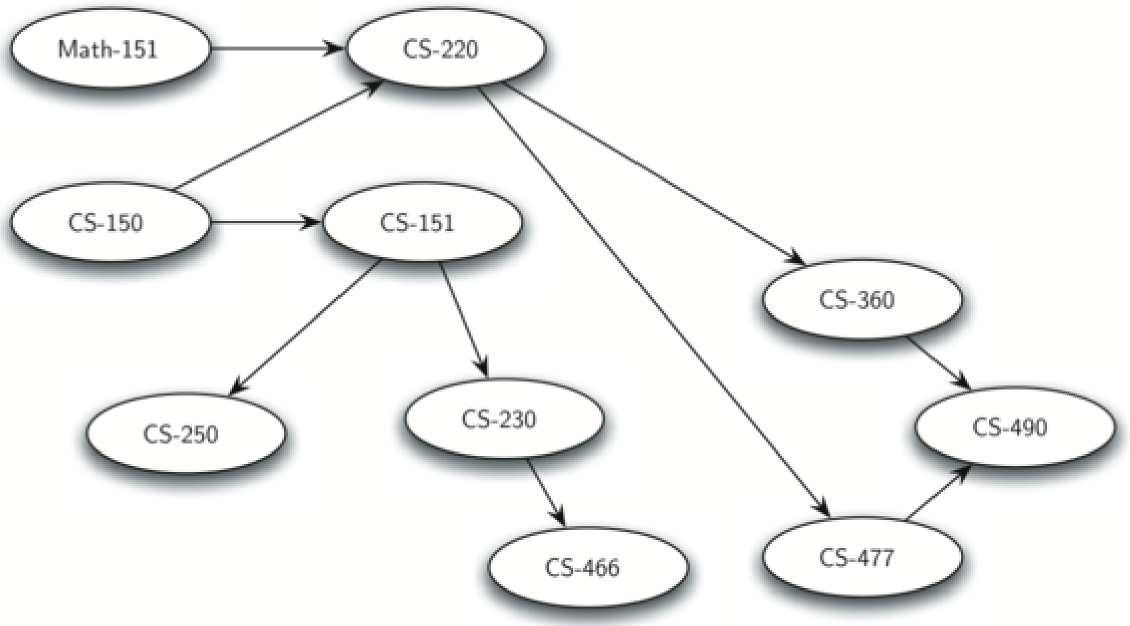
In [4]:
```
def trimBST(tree, minVal, maxVal):

    if not tree:
        return

    tree.left=trimBST(tree.left, minVal, maxVal)
    tree.right=trimBST(tree.right, minVal, maxVal)

    if minVal<=tree.val<=maxVal:
        return tree

    if tree.val<minVal:
        return tree.right

    if tree.val>maxVal:
        return tree.left
```

The complexity of this algorithm is O(N), where N is the number of nodes in the tree. Because we basically perform a post-order traversal of the tree, visiting each and every node one. This is optimal because we should visit every node at least once. This is a very elegant question that demonstrates the effectiveness of recursion in trees.

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



Vertex (Nodes)

- 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.”

Edge

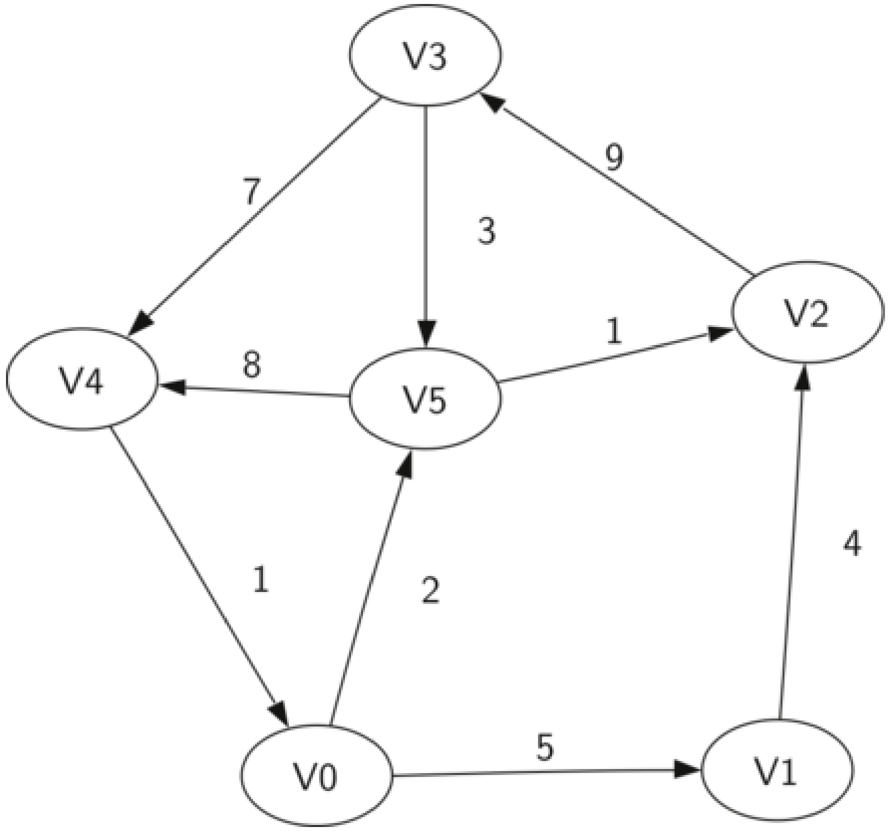
- An edge connects two vertices to show that there is a relationship between them. Edges may be one-way or two-way. If the edges in a graph are all one-way, we say that the graph is a directed graph, or a digraph.

Weight

- 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.

Formal Definition of a Graph

- 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)}

Path

A path in a graph is a sequence of vertices that are connected by edges. Formally we would define a path as $w1, w2, \ldots, wn$ such that $(wi, wi + 1) \in E$ for all $1 \leq i \leq n - 1$. 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)}

Cycle

- 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. The path (V5,V2,V3,V5) is a cycle.

Adjacency matrix and list

Learn how to represent graphs

- Adjacency Matrix
- Adjacency List

Adjacency Matrix

- 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.

Adjacency List

- 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.

Implementation of a Graph as an Adjacency List

Using dictionaries, it is easy to implement the adjacency list in Python. In our implementation of the Graph abstract data type we will create two classes: **Graph**, which holds the master list of vertices, and **Vertex**, which will represent each vertex in the graph.

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 constructor simply initializes the id, which will typically be a string, and the **connectedTo** dictionary. 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.

In [3]:
```
class Vertex:
    def __init__(self,key):
        self.id = key
        self.connectedTo = {}

    def addNeighbor(self,nbr,weight=0):
        self.connectedTo[nbr] = weight
    def __str__(self):
        return str(self.id) + ' connectedTo: ' + str([x.id for x in sel
f.connectedTo])

    def getConnections(self):
        return self.connectedTo.keys()

    def getId(self):
        return self.id

    def getWeight(self,nbr):
        return self.connectedTo[nbr]
```

In order to implement a Graph as an Adjacency List what we need to do is define the methods our Adjacency List object will have:

- **Graph()** creates a new, empty graph.
- **addVertex(vert)** adds an instance of Vertex to the graph.
- **addEdge(fromVert, toVert)** Adds a new, directed edge to the graph that connects two vertices.
- **addEdge(fromVert, toVert, weight)** Adds a new, weighted, directed edge to the graph that connects two vertices.
- **getVertex(vertKey)** finds the vertex in the graph named vertKey.
- **getVertices()** returns the list of all vertices in the graph.
- **in** returns True for a statement of the form vertex in graph, if the given vertex is in the graph, False otherwise.

In [4]:
```
class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self,key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
```

```
            self.vertList[key] = newVertex
            return newVertex

        def getVertex(self,n):
            if n in self.vertList:
                return self.vertList[n]
            else:
                return None

        def __contains__(self,n):
            return n in self.vertList

        def addEdge(self,f,t,cost=0):
            if f not in self.vertList:
                nv = self.addVertex(f)
            if t not in self.vertList:
                nv = self.addVertex(t)
            self.vertList[f].addNeighbor(self.vertList[t], cost)

        def getVertices(self):
            return self.vertList.keys()

        def __iter__(self):
            return iter(self.vertList.values())
```

Let's see a simple example of how to use this:

In [5]:
```
g = Graph()
for i in range(6):
    g.addVertex(i)
```

In [6]:
```
g.vertList
```

Out[6]:
```
{0: <__main__.Vertex instance at 0x10476b680>,
 1: <__main__.Vertex instance at 0x104cce5f0>,
 2: <__main__.Vertex instance at 0x10395d950>,
 3: <__main__.Vertex instance at 0x1039c00e0>,
 4: <__main__.Vertex instance at 0x1039c4e60>,
 5: <__main__.Vertex instance at 0x1039c45f0>}
```

In [7]:
```
g.addEdge(0,1,2)
```

In [12]:
```
for vertex in g:
    print vertex
    print vertex.getConnections()
    print '\n'
```

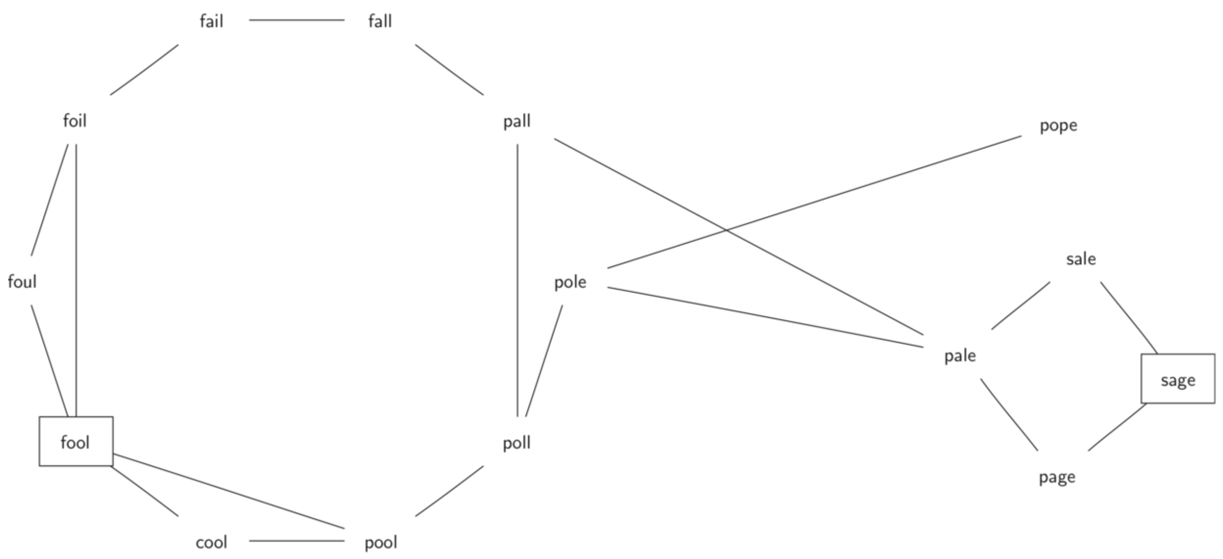
```
0 connectedTo: [1]
[<__main__.Vertex instance at 0x104cce5f0>]


1 connectedTo: []
[]


2 connectedTo: []
[]


3 connectedTo: []
[]


4 connectedTo: []
[]


5 connectedTo: []
[]
```

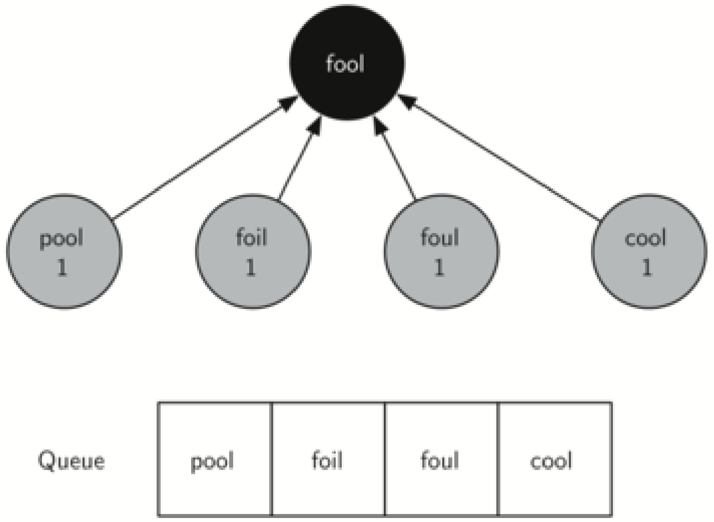
Breadth First Search

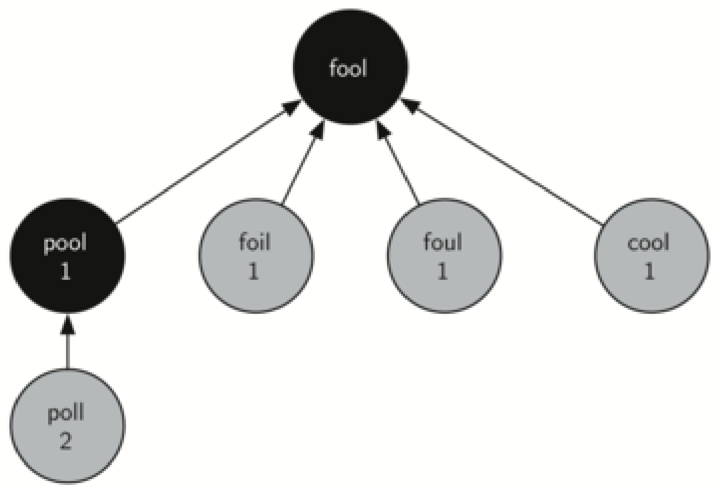
- Breadth first search (BFS) is one of the easiest algorithms for searching a graph. It also serves as a prototype for several other important graph algorithms that we will study later.
- Given a graph G and a starting vertex s, a breadth first search proceeds by exploring edges in the graph to find 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.

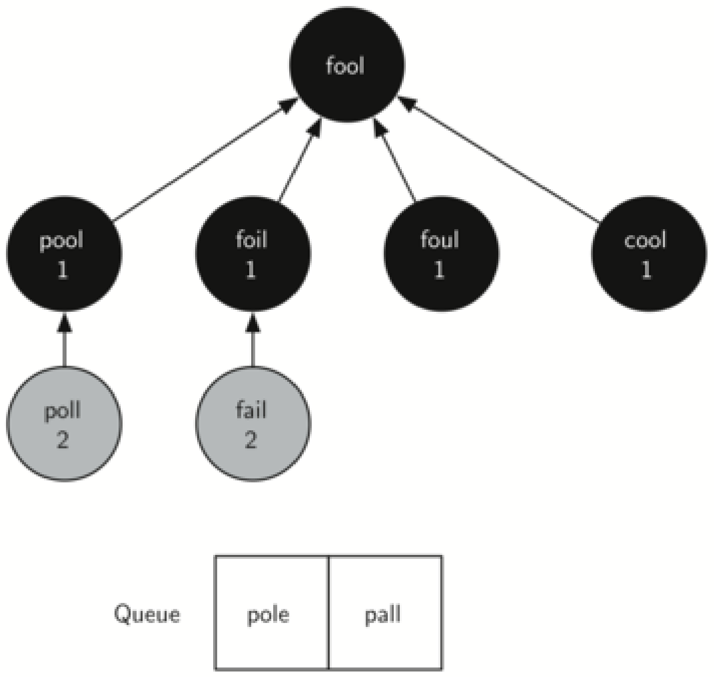
- To keep track of its progress, BFS colors each of the vertices white, gray, or black.
- All the vertices are initialized to white when they are constructed.
- A white vertex is an undiscovered vertex.
- When a vertex is initially discovered it is colored gray, and when BFS has completely explored a 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

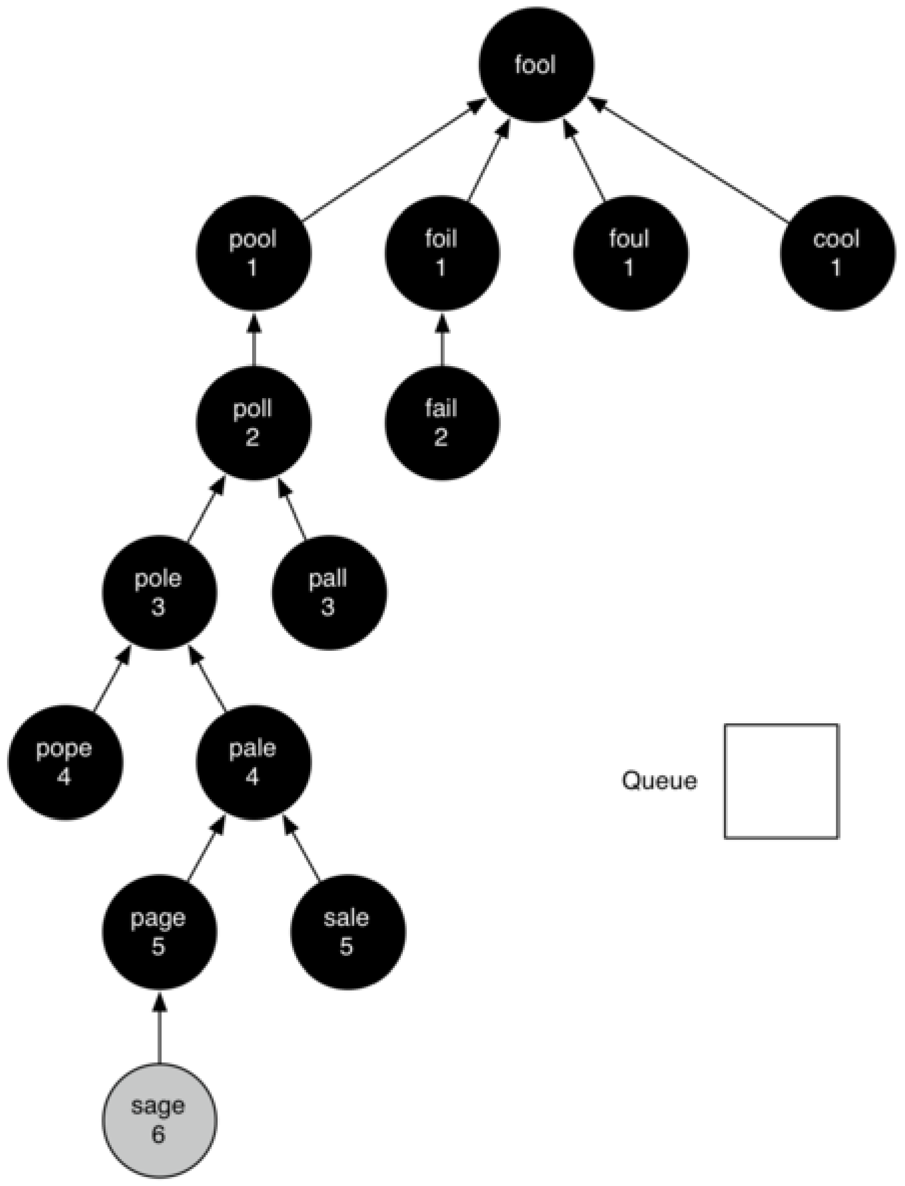
adjacency list of currentVert have been explored.






fool
pool 1
foil 1
foul 1
cool 1
poll 2


fool
pool 1
foil 1
foul 1
cool 1
poll 2
fail 2
Queue
pole
pall



- 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.

Implementation of Breadth First Search

An alternative algorithm called Breath-First search provides us with the ability to return the same results as DFS but with the added guarantee to return the shortest-path first. This algorithm is a little more tricky to implement in a recursive manner instead using the queue data-structure, as such I will only being documenting the iterative approach. The actions performed per each explored vertex are the same as the depth-first implementation, however, replacing the stack with a queue will instead explore the breadth of a vertex depth before moving on. This behavior guarantees that the first path located is one of the shortest-paths present, based on number of edges being the cost factor.

We'll assume our Graph is in the form:

In [1]:
```
graph = {'A': set(['B', 'C']),
         'B': set(['A', 'D', 'E']),
         'C': set(['A', 'F']),
         'D': set(['B']),
         'E': set(['B', 'F']),
         'F': set(['C', 'E'])}
```

Connected Component

Similar to the iterative DFS implementation the only alteration required is to remove the next item from the beginning of the list structure instead of the stacks last.

In [2]:
```
def bfs(graph, start):
    visited, queue = set(), [start]
    while queue:
        vertex = queue.pop(0)
        if vertex not in visited:
            visited.add(vertex)
            queue.extend(graph[vertex] - visited)
    return visited

bfs(graph, 'A')
```

Out[2]:
```
{'A', 'B', 'C', 'D', 'E', 'F'}
```

Paths

This implementation can again be altered slightly to instead return all possible paths between two vertices, the first of which being one of the shortest such path.

In [2]:
```
def bfs_paths(graph, start, goal):
    queue = [(start, [start])]
    while queue:
        (vertex, path) = queue.pop(0)
        for next in graph[vertex] - set(path):
            if next == goal:
                yield path + [next]
            else:
                queue.append((next, path + [next]))

list(bfs_paths(graph, 'A', 'F'))
```

Out[2]:
```
[['A', 'C', 'F'], ['A', 'B', 'E', 'F']]
```

Knowing that the shortest path will be returned first from the BFS path generator method we can create a useful method which simply returns the shortest path found or ‘None’ if no path exists. As we are using a generator this in theory should provide similar performance results as just breaking out and returning the first matching path in the BFS implementation.

In [3]:
```
def shortest_path(graph, start, goal):
    try:
        return next(bfs_paths(graph, start, goal))
    except StopIteration:
        return None

shortest_path(graph, 'A', 'F')
```

```
Out[3]: ['A', 'C', 'F']
```

Resources

- Depth-and Breadth-First Search
- Connected component)
- Adjacency matrix
- Adjacency list
- Python Gotcha: Default arguments and mutable data structures
- Generators

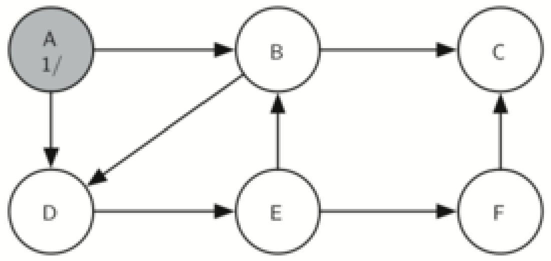
Depth First Search

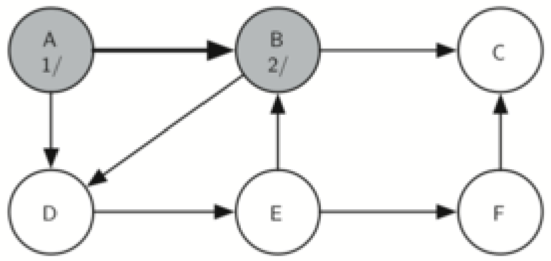
DFS

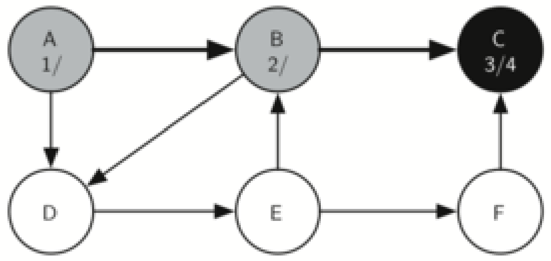
- The knight's tour is a special case of a depth first search where the goal is to create the deepest depth first tree, without any branches.
- The more general depth first search is actually easier.
- Its goal is to search as deeply as possible, connecting as many nodes in the graph as possible and 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.

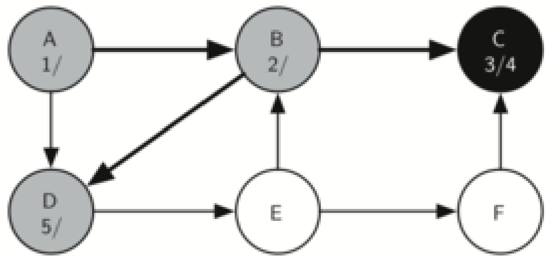
- The new instance variables are the discovery and finish times.
- The discovery time tracks the number of steps in the algorithm before a vertex is first encountered.
- The finish time is the number of steps in the algorithm before a vertex is colored black

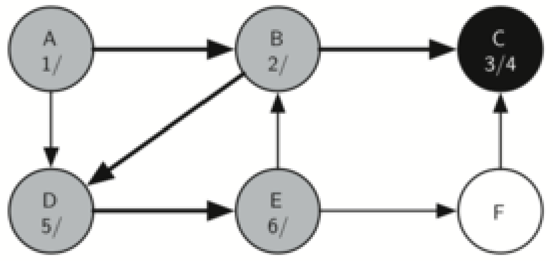
Constructing the Depth First Search Tree

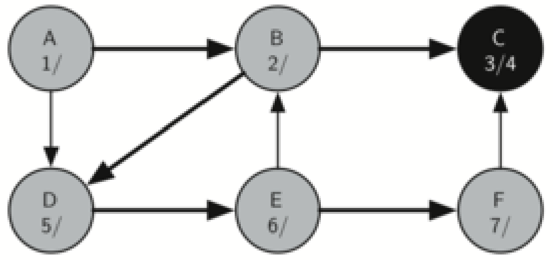


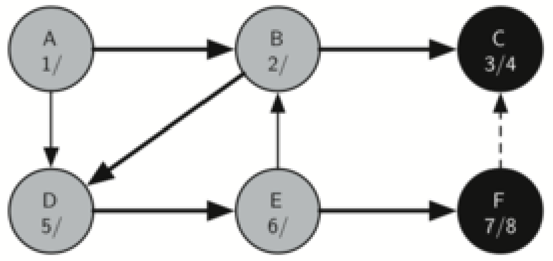


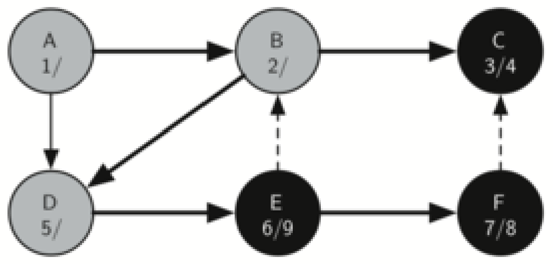


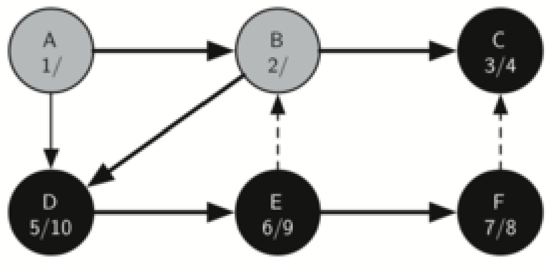

A
1/
B
2/
C
3/4
D
5/
E
F

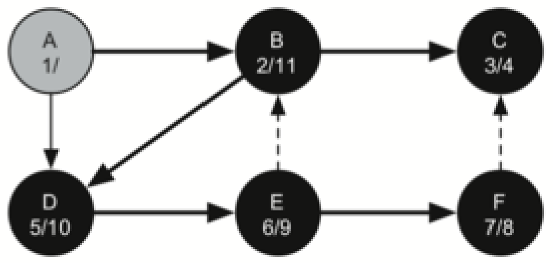

A
1/
B
2/
C
3/4
D
5/
E
6/
F

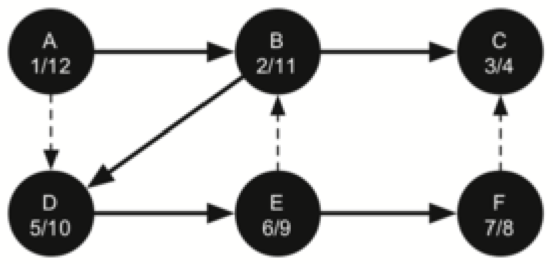

A
1/
B
2/
C
3/4
D
5/
E
6/
F
7/


A
1/
B
2/
C
3/4
D
5/
E
6/
F
7/8


A 1/
B 2/
C 3/4
D 5/
E 6/9
F 7/8


A 1/
B 2/
C 3/4
D 5/10
E 6/9
F 7/8


A 1/
B 2/11
C 3/4
D 5/10
E 6/9
F 7/8


A 1/12
B 2/11
C 3/4
D 5/10
E 6/9
F 7/8

- 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.

Implementation of Depth-First Search

This algorithm we will be discussing is Depth-First search which as the name hints at, explores possible vertices (from a supplied root) down each branch before backtracking. This property allows the algorithm to be implemented succinctly in both iterative and recursive forms. Below is a listing of the actions performed upon each visit to a node.

- Mark the current vertex as being visited.
- Explore each adjacent vertex that is not included in the visited set.

We will assume a simplified version of a graph in the following form:

In [7]:
```
graph = {'A': set(['B', 'C']),
         'B': set(['A', 'D', 'E']),
         'C': set(['A', 'F']),
         'D': set(['B']),
         'E': set(['B', 'F']),
         'F': set(['C', 'E'])}
```

Connected Component

The implementation below uses the stack data-structure to build-up and return a set of vertices that are accessible within the subjects connected component. Using Python's overloading of the subtraction operator to remove items from a set, we are able to add only the unvisited adjacent vertices.

In [15]:
```
def dfs(graph, start):
    visited, stack = set(), [start]
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            stack.extend(graph[vertex] - visited)
    return visited

dfs(graph, 'A')
```

Out[15]:
```
{'A', 'B', 'C', 'D', 'E', 'F'}
```

The second implementation provides the same functionality as the first, however, this time we are using the more succinct recursive form. Due to a common Python gotcha with default parameter values being created only once, we are required to create a new visited set on each user invocation. Another Python language detail is that function variables are passed by reference, resulting in the visited mutable set not having to reassigned upon each recursive call.

In [12]:
```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    for nxt in graph[start] - visited:
        dfs(graph, nxt, visited)
    return visited

dfs(graph, 'A')
```

Out[12]:
```
{'A', 'B', 'C', 'D', 'E', 'F'}
```

Paths

We are able to tweak both of the previous implementations to return all possible paths between a start and goal vertex. The implementation below uses the stack data-structure again to iteratively solve the problem, yielding each possible path when we locate the goal. Using a generator allows the user to only compute the desired amount of alternative paths.

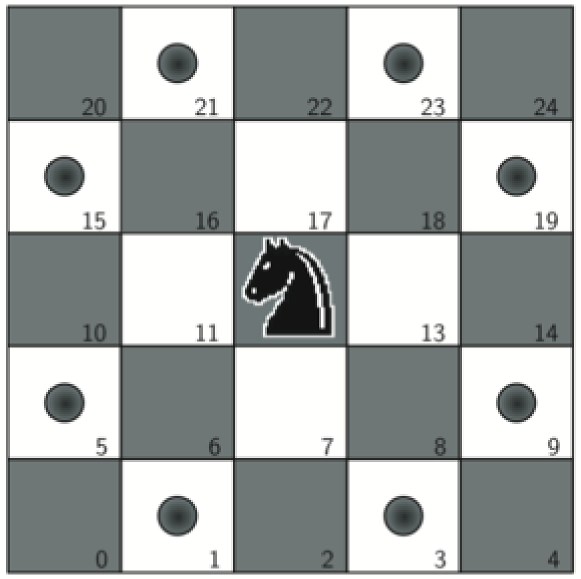
In [11]:
```
def dfs_paths(graph, start, goal):
    stack = [(start, [start])]
    while stack:
        (vertex, path) = stack.pop()
        for nxt in graph[vertex] - set(path):
            if nxt == goal:
                yield path + [nxt]
            else:
                stack.append((nxt, path + [nxt]))

list(dfs_paths(graph, 'A', 'F'))
```

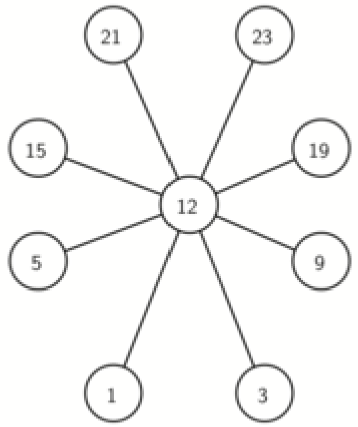
Out[11]:
```
[['A', 'B', 'E', 'F'], ['A', 'C', 'F']]
```

Knight's Tour Code

- 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 $rows \times columns - 1$ 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.
- 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.
- 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.





In [1]:
```
def knightGraph(bdSize):
    ktGraph = Graph()
    for row in range(bdSize):
        for col in range(bdSize):
            nodeId = posToNodeId(row,col,bdSize)
            newPositions = genLegalMoves(row,col,bdSize)
            for e in newPositions:
                nid = posToNodeId(e[0],e[1],bdSize)
                ktGraph.addEdge(nodeId,nid)
    return ktGraph

def posToNodeId(row, column, board_size):
    return (row * board_size) + column
```

In [2]:

```
def genLegalMoves(x,y,bdSize):
    newMoves = []
    moveOffsets = [(-1,-2),(-1,2),(-2,-1),(-2,1),
                   ( 1,-2),( 1,2),( 2,-1),( 2,1)]
    for i in moveOffsets:
        newX = x + i[0]
        newY = y + i[1]
        if legalCoord(newX,bdSize) and \
                        legalCoord(newY,bdSize):
            newMoves.append((newX,newY))
    return newMoves

def legalCoord(x,bdSize):
    if x >= 0 and x < bdSize:
        return True
    else:
        return False
```

In [3]:

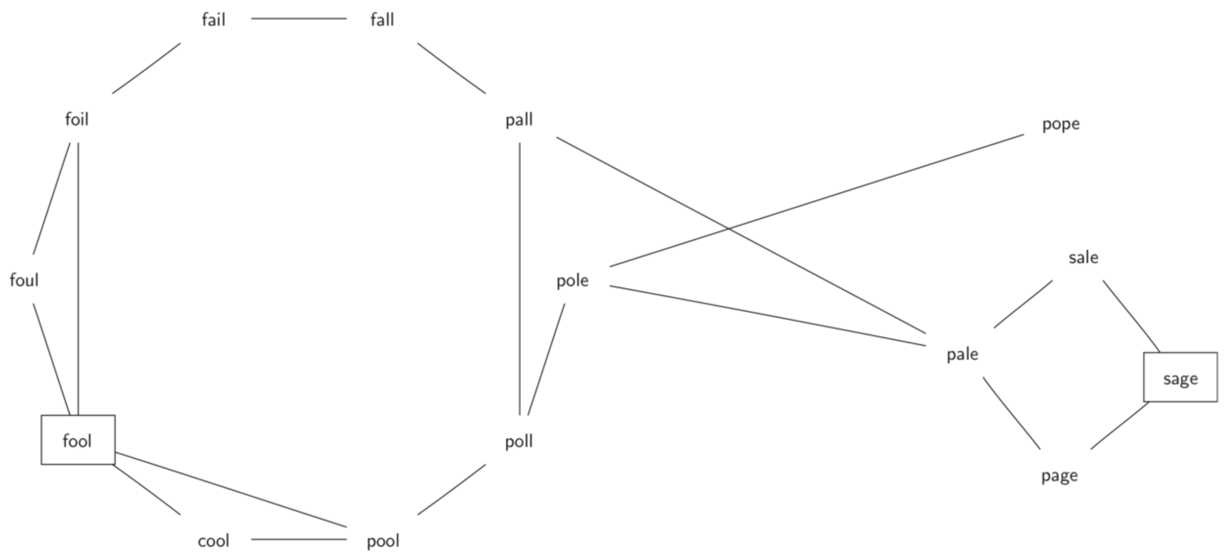
```
def knightTour(n,path,u,limit):
        u.setColor('gray')
        path.append(u)
        if n < limit:
            nbrList = list(u.getConnections())
            i = 0
            done = False
            while i < len(nbrList) and not done:
                if nbrList[i].getColor() == 'white':
                    done = knightTour(n+1, path, nbrList[i], limit)
                i = i + 1
            if not done:  # prepare to backtrack
                path.pop()
                u.setColor('white')
        else:
            done = True
        return done
```

Word Ladder Example Code

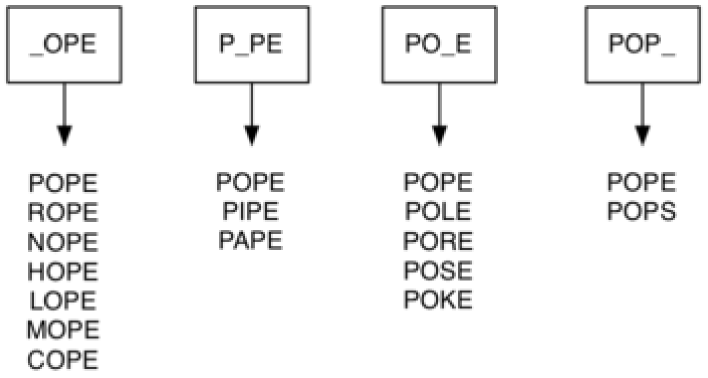
Consider the following puzzle called a word ladder. Transform the word “FOOL” into the word “SAGE”.In a word ladder puzzle you must make the change occur gradually by changing one letter at a time.At each step you must transform one word into another word, you are not allowed to transform a word into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



In [1]:
```
class Vertex:
    def __init__(self,key):
        self.id = key
        self.connectedTo = {}
    def addNeighbor(self,nbr,weight=0):
        self.connectedTo[nbr] = weight

    def __str__(self):
        return str(self.id) + ' connectedTo: ' + str([x.id for x in sel
f.connectedTo])

    def getConnections(self):
        return self.connectedTo.keys()

    def getId(self):
        return self.id

    def getWeight(self,nbr):
        return self.connectedTo[nbr]
```

In [2]:
```
class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self,key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex

    def getVertex(self,n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None

    def __contains__(self,n):
        return n in self.vertList

    def addEdge(self,f,t,cost=0):
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t], cost)

    def getVertices(self):
        return self.vertList.keys()

    def __iter__(self):
        return iter(self.vertList.values())
```

Code for buildGraph function:

```
In [19]: def buildGraph(wordFile):
             d = {}
             g = Graph()

             wfile = open(wordFile,'r')
             # create buckets of words that differ by one letter
             for line in wfile:
                 print line
                 word = line[:-1]
                 print word
                 for i in range(len(word)):
                     bucket = word[:i] + '_' + word[i+1:]
                     if bucket in d:
                         d[bucket].append(word)
                     else:
                         d[bucket] = [word]
             # add vertices and edges for words in the same bucket
             for bucket in d.keys():
                 for word1 in d[bucket]:
                     for word2 in d[bucket]:
                         if word1 != word2:
                             g.addEdge(word1,word2)
             return g
```

```
In [ ]:
```

Extra Resource: MIT Algorithms and Data Structure Course! https://www.youtube.com/watch?v=s-CYnVz-uh4