UNIVERSITY OF
SAN FRANCISCO

Master of Science
in Analytics

Algorithms

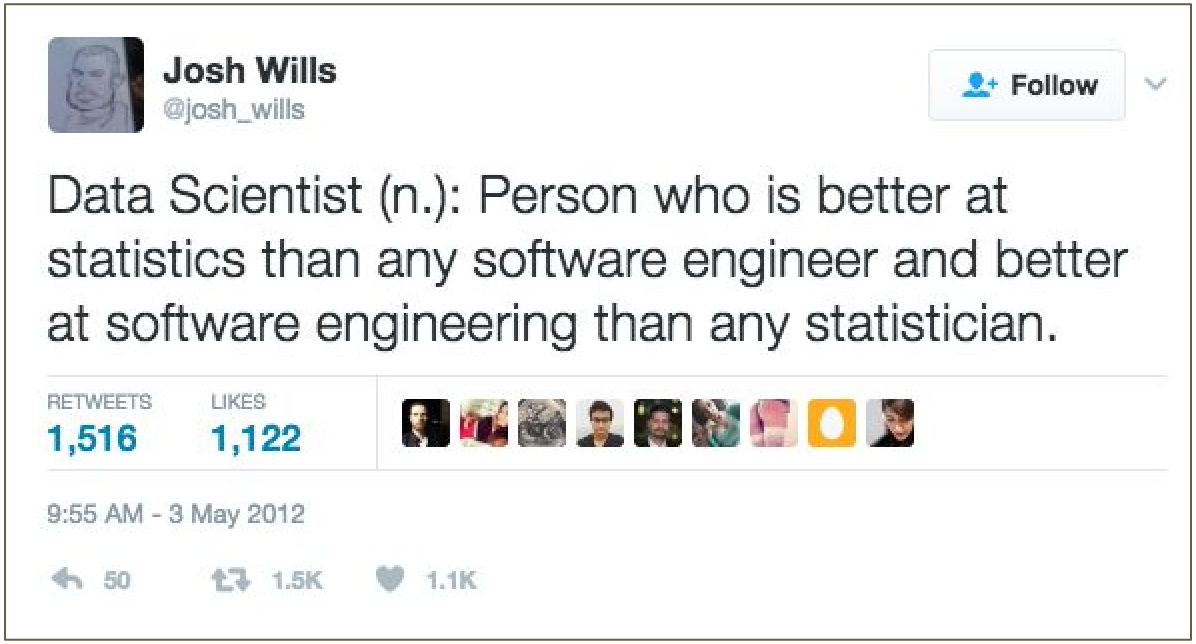
Interview Skills

Agenda

- Whiteboarding
- Algorithms (on array / arraylist instances)
  - Searching
  - Sorting
- Data structures
  - Object Oriented Programming
  - Hashtables
  - Stacks
  - Queues
  - Trees
  - Graphs

Why Study Algorithms?

- Many people interviewing for data science are computer scientists
  - Until data science comes into its own as a field, many interviewers will be software engineers, database engineers, etc.
  - Algorithms sits at the core of computer science
- Many topics from Algorithms apply to data science
  - Ideas from computational complexity apply to databases (SQL, etc.)
  - The expectation is that you can do both well



Whiteboard Interviewing

- What is whiteboarding?
  - Candidate is given a blank writing surface (chalkboard, markerboard, paper, etc.)
  - Candidate is given a technical question and expected to write a working solution
  - Common in technical interviews
  - Measures candidate's problem-solving and communication skills
- How to succeed at whiteboarding:
  - Write the question and any details you might forget (examples, guidelines, etc.)
  - Write legibly (i.e. large letters, even spacing, predictable indentation, etc.)
  - Use the space on the whiteboard efficiently — eg. leave space between lines of code — but do not exceed the size of the canvas
  - Ask questions and talk through your solution while you write
- If you don't do well:
  - Don't take it personally — highly qualified people have tried and failed — it is a completely artificial exercise
  - Try, try again (practice with mock whiteboarding interviews)

Whiteboarding — Example

“Write code to take the derivative of a polynomial”

Notice how interviewer and candidate cooperate on this question

All These Things and More...

- All algorithms and data structures are standard for a CS curriculum
  - All are available in standard CS references; all are available online
  - Some are variations
  - See Problem Solving with Algorithms and Data Structures using Python
- What good coders do
  - Understand the concept, memorise the steps
  - Avoid the urge to memorise code

Agenda

- Whiteboarding
- Algorithms (on array / arraylist instances)
    - Searching
    - Sorting
- Data structures
    - Object Oriented Programming
    - Hashtables
    - Stacks
    - Queues
    - Trees
    - Graphs

Arrays

- An array is a (memory-contiguous) series of objects sharing the same type
  - Possibly in 1 dimension (vector), 2D (matrix), or N-dimensions (tensor)
  - In Python, there are no arrays, but there is a List — i.e. ArrayList — which is an array of pointers to objects
- Access
  - Can access any element by name and offset (for example A[5])
  - First element is usually at index 0 (though there are some 1-based languages)
- Assumptions
  - Access to any part of the array can be performed in one read
  - Read operations and write operations have equal cost

Task: Search

- Task
  - Given: an array (A), a target value (target)
  - Return the index of target in A
  - Return -1 if target is not in A
- Algorithm: Linear Search
  - Iterate from the first element to the last, keeping track of the index
  - At each iteration, if the element is equal to the target, return the index
  - If no element is found at the end of the iteration, return -1
  - Used to implement “in” ala “`if letter in word`”
- Python implementation:

```
def linear_search (A, target):
    for i in range(len(A)):
        if A[i] == target:
            return i
    return -1
```

Another Search Solution

- Algorithm: Binary Search
  - Same task for search, but assume A is sorted in non-decreasing order
  - Look for the target at the middle of the array; if the middle is equal to the target, return it
  - Recalculate the range where the target may be
    - If the target is greater than the middle element, ignore the lower half of the array
    - If the target is less, ignore the upper half
  - If “upper” and “lower” cross, return -1
- Implementations:
  - There are recursive and iterative implementations
  - The recursive implementation may be more common, but the iterative implementation is more efficient

Iterative Binary Search (Python)

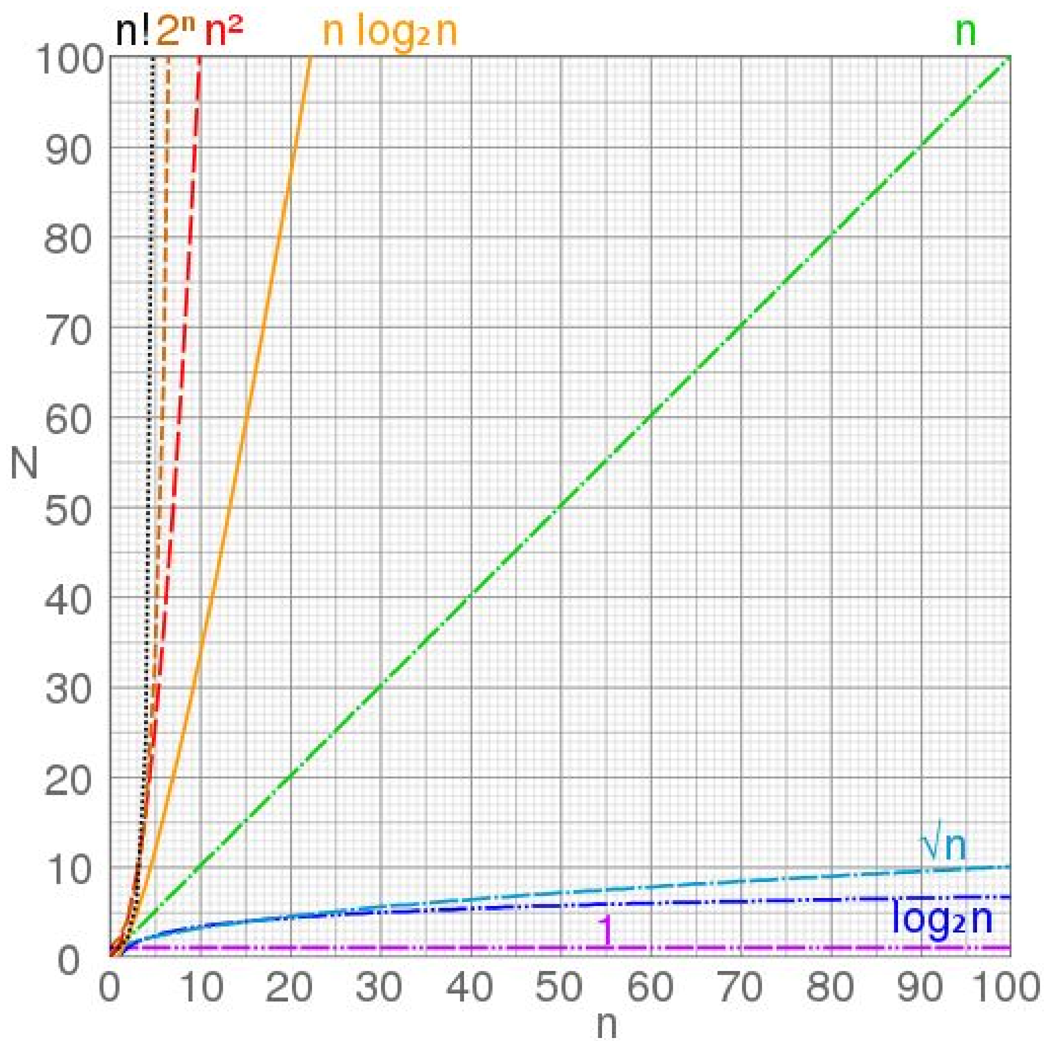
```
def binary_search (A, target):
    lower = 0
    upper = len(A) - 1
    while (lower <= upper):
        middle = (lower + upper) / 2
        if A[middle] == target:
            return middle
        else:
            if target < A[middle]:
                upper = middle - 1
            else:
                lower = middle + 1
    return -1
```

Computational Complexity

- Compare algorithms primarily by number of array reads and writes
  - Using the number of items in the array (n), the worst case for search (item not in array)
    - Linear search reads all n items
    - Binary search reads about $\log_2(n)$ items
  - Use Big-O notation
    - Formally, if f and g are two functions, we can say $f(x) = O(g(x))$ if $f(x) <= M(g(x))$ for all $x > x_0$, given a constant M
    - Informally: throw out constants, lower-order terms
- Complexity
  - Common to use "worst case" running time
  - It is also possible to use best case, average case
  - Always use "worst case" unless it is possible to prove that it is uncommon
  - Search for our search algorithms (worst case):
    - Linear Search = O(n)
    - Binary Search = O(log n)

Differences in Speed

- When the size of the array is small, speed differences are negligible
- With large values of n (thousands, millions, etc.), differences are stark



Task: Sort

- Task
  - Given: an array (A)
  - Reorder elements in A in non-decreasing order
- In-place algorithms:
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort
- See VisuAlgo for animations
- In Python, we can swap in O(1) time:

```
def swap (A, pos1, pos2):
   A[pos1], A[pos2] = A[pos2], A[pos1]
```

  - … but it's really 3 operations
  - … and it's

Selection Sort

- Algorithm:
  - Find the highest-valued item and place it in the last position
  - Eliminate last array position and fill second-to-last position with highest-valued item
  - Fill all subsequent positions similarly
- Python implementation

```
def selection_sort (A):
    for fillslot in range(len(A)-1, 0, -1):
        max = 0
        for location in range(1, fillslot+1):
            if A[location] > A[max]:
                max = location
        A[fillslot], A[max] = A[max], A[fillslot]
```

- Running time: $O(n^2)$

Bubble Sort

- Algorithm:
  - Look at each pair of items; swap if in not in order
  - Repeat n-1 times
- Python implementation

```
def bubble_sort (A):
    for j in range(len(A)-1, 0, -1):
        for i in range(j):
            if A[i] > A[i+1]:
                swap(A, i, i+1)
```

- Running time: $O(n^2)$
  - Efficiency: stop sorting when there are no swaps
  - Running time is unchanged

Insertion Sort

- Algorithm:
  - Assume part (first item) of the array is sorted
  - Insert one (the next) item from the unsorted part and shift (if necessary) to maintain sorted order
  - Repeat for all subsequent positions similarly
- Python implementation

```
def insertion_sort (A):
    for index in range(1, len(A)):
        currval = A[index]
        position = index
        while position>0 and A[position-1]>currval:
            A[position] = A[position-1]
            position -= 1
        A[position] = currval
```

- Running time: $O(n^2)$

Merge Sort — Overview

- Algorithm:
  - Recursively split A in half into subarrays until there are 0 or 1 items (which is sorted)
  - Merge elements in adjacent subarrays by selecting lowest item and placing it in the first position
  - Add remaining elements from left half or right half
- Algorithm is recursive
  - Divide-and-conquer
  - Easy split; difficult merge

Merge Sort — Split

- Algorithm:
    - Divide-and-conquer algorithm
    - Step 1: Recursively split A in half into subarrays until there are 0 or 1 items
- Python implementation

```
def merge_sort (A):
    if len(A)>1:
        mid = len(A)//2
        lefthalf = A[:mid]
        righthalf = A[mid:]

    merge_sort(lefthalf)
    merge_sort(righthalf)
```

Merge Sort — Merge

- Algorithm:
  - Step 2: merge
  - Merge elements in adjacent subarrays by selecting lowest item and placing it in the first position
- Python implementation

```
# Array is now split
i=j=k=0
while i < len(lefthalf) and j < len(righthalf):
    if lefthalf[i] < righthalf[j]:
        A[k] = lefthalf[i]
        i += 1
    else:
        A[k] = righthalf[j]
        j += 1
    k += 1
```

Merge Sort — Add Remaining

- Algorithm:
  - Step 3: add remaining elements not already merged in step 2
  - Add remaining elements from left half or right half
- Python implementation

```
# Array is now split and (mostly) merged
while i < len(lefthalf):
    A[k] = lefthalf[i]
    i += 1
    k += 1
while j < len(righthalf):
    A[k] = righthalf[j]
    j += 1
    k += 1
```

Merge Sort — Review

- Algorithm:
  - Recursively split A in half into subarrays until there are 0 or 1 items (which is sorted)
  - Merge elements in adjacent subarrays by selecting lowest item and placing it in the first position
  - Add remaining elements from left half or right half
- Running time: $O(n \log_2 n)$
  - Each step divides array exactly in half and creates a new level of recursion
  - Each level of recursion perform O(n) reads / writes

Quick Sort — Overview

- Algorithm:
  - Pick a pivot value from A
  - Partition: move all elements less than pivot to left part of (sub-)array; all elements greater than pivot to right portion
  - Recursively quick sort left part and right part
  - Difficult split; easy merge
- Python implementation

```
def quick_sort (A):
    qs(A, 0, len(A)-1)

def qs(A, first, last):
    if first<last:
        split = partition(A, first, last)
        qs(A, first, split-1)
        qs(A, split+1, last)
```

Quick Sort — Partition

```
def partition (A, first, last):
    pivot = A[first]  # Other ways to select pivot?
    left = first+1
    right = last
    done = False
    while not done:
        while left <= right and A[left] < pivot
            left += 1
        while A[right] >= pivot and right >= left:
            right -= 1
        if right < left:
            done = True
        else:
            swap(A, left, right)
    swap(A, first, right)
    return right
```

Quick Sort — Review

- Algorithm:
  - Difficult split; easy merge
  - Partition array and recursively quick sort the left and right halves
- Running time: $O(n \log_2 n)$
  - Complicated analysis because the pivot does not appear in a stable place in the array
  - Worst case: pivot is consistently on left side or right side of array: n levels of recursion, each ~ $O(n)$ reads/writes —> $O(n^2)$
  - The worst case may be rare ~ $(1/n^2)$?
  - The average case is $O(n \log n)$

Agenda

- Whiteboarding
- Algorithms (on array / arraylist instances)
  - Searching
  - Sorting
- Data structures
  - Object Oriented Programming
  - Hashtables
  - Stacks
  - Queues
  - Trees
  - Graphs

Object Oriented Programming

- OOP is a framework for building programs / applications / software
- Elements of object oriented languages:
  - **Encapsulation** (variables and functions in one place)
  - **Inheritance** (classes, special classes)
  - **Polymorphism** (a subclass can act as a member of its superclass)
- In Python (also: Java, C++, ...), you can:

Do this	... for example (in Python):
Create an object	`my_file = open("my_file.txt", "wb")`
Use an object	`my_file.write("I want to learn OOP.")` `my_file.close()`
Destroy an object	`my_file = None`

Example: person

- What makes a person?
  - Name (first names, surname)
  - DOB
  - Biography
  - [...]
- What can a person do?
  - Get married
  - Add to biography
  - Change names
  - Print to a file
  - [...]

- Example fields:
  - Guido van Rossum
  - 1956-01-31
  - Created python
- Example actions:
  - Married Kim Knapp
  - Biography additions: Working for Dropbox (2012)

Python Class: self, __init__

- The class keyword
  - Defines a class
  - Next token (person) names the class
  - All code in class indented
- Encapsulation
  - Fields and actions contained within the class
  - Actions: called “methods” (or “behaviours” or…)
- Keywords ⇒ key concepts
  - self: a particular *instance* of an object; required as first argument to each method
  - __init__: automatically called when the object is instantiated; also defines the class’ fields

```
class person:
    '''
    Comments for a person class!
    '''

    def __init__(self, name):
        '''
        First function called.
        That's 2 underscores before
        and 2 underscores after.
        '''
        self.first_names = name['f']
        self.surname = name['last']
        self.biography = []
        self.spouse = None

    def add_to_bio(self, words):
        '''
        A function to add to bio.
        '''
        self.biography.append(words)
```

Declaration of person

```
class person:

    def __init__(self, name):
        self.first_names = name['f']
        self.surname = name['last']
        self.biography = []
        self.spouse = None

    def add_to_bio(self, words):
        self.biography.append(words)

    def change_name(self, name):
        pass

    def change_spouse(self, spouse):
        pass
```

- Code needed to set up example

```
name = dict()
name['f'] = 'Guido'
name['last'] = 'van Rossum'
```

- Need an instance? Assign to a variable from a class' name

```
p = person(name)
```

- Using an instance? Use a "."

```
p.add_to_bio('2012: Dropbox')
```

- Other than the use of "self", code in methods = code in functions

Limitations & Good Practices

- Limitations of python
  - All fields (or methods) are accessible:

    ```
    p = person(name)
    p.biography = ['No'] # yuck!
    ```

  - Violation of proper encapsulation
- Good practices:
  - In each class:
    - Create accessors for fields
    - Create mutators for fields
    - Rely on methods more than on data
    - Create tiny functions
  - Between classes (collaborations):
    - Create specialty classes — eg. for I/O
    - … but be practical

```
class person:

    def __init__(self, name):
        self.name = dict()
        self.biography = []
        self.spouse = None
        self.change_name(name)

    def get_name(self, name):
        pass

    def set_name(self, name):
        pass

    def to_string(self):
        pass
```

Nomenclature

- Class — (“person”) the blueprint / template
- Object — (“guido” or “kim”) an instance of a class
- Instantiation — the act of creating an object from a class
- “On” — (apologies to English L1) the preposition used for a function of a class. For example, “the get_name function on person…”.
- Attributes
- Constructor

Hashing

- Concept
  - Implementation of dictionary (Dict) container
  - Store values in an array of (static) size; each entry with a unique position (slot)
  - 1: Convert array index to unique value by scrambling — eg. 54 = 5 + 4 ⇒ 9
  - 2: Ensure index of items being stored fit in array — eg. 54 ⇒ 9 % len(A)
- Hash function
  - Function to convert values to hash positions
  - Strings can be converted to numbers by ASCII value, etc.
  - Should be quick to calculate, result in the fewest collisions (two or more values hashing to the same location)
  - *Folding:* divide key into groups and add individual portions — eg. 415-422-5101 ⇒ [41 + 54 + 22 + 51 + 01] ⇒ 169
  - *Mid-squaring:* square the key and take middle portion — eg. 44^2 ⇒ 1936 ⇒ 93
- Resolving collisions
  - Rehashing — eg. new_hash = hash(old_hash_value)
  - Open addressing — place an element in next available slot (linear, quadratic, etc.)
  - Chaining — Each slot is a structure (list?) of items — example here

HashTable implementation

Concept — need two classes

- Many instances of hashnode class to store a record (key-value pair)
- One instance of hashtable to manage hashnode instances

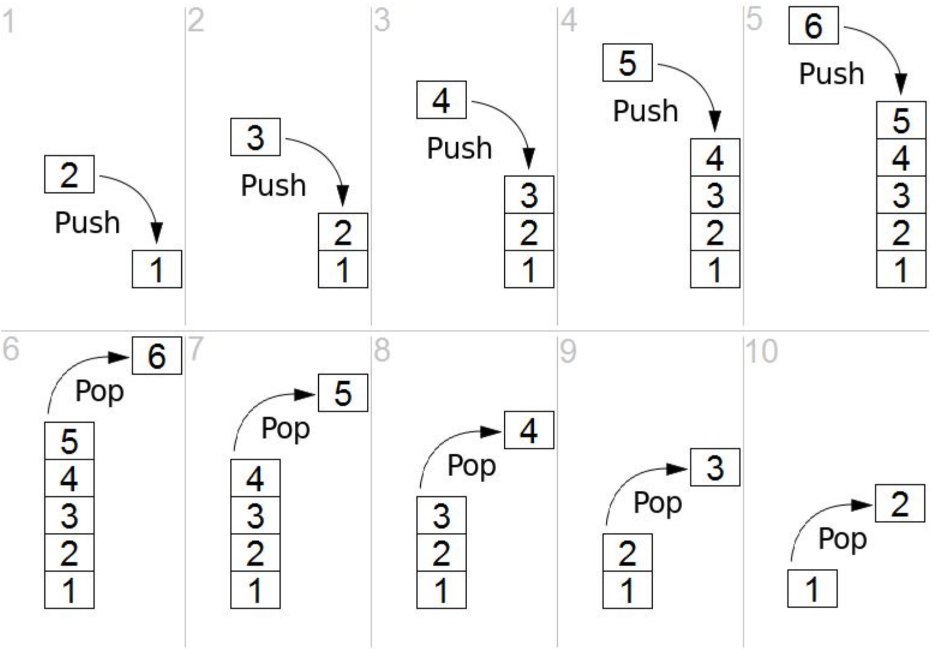
```
class hashnode:

    def __init__(self, key, value):
        self.key = key
        self.val = value
        self.next = None # This is a low-cost linked list



class hashtable:

    def __init__(self):
        self.size = 101
        self.slots = [None] * self.size
```

HashTable insert and get

```
# Equivalent to: dictionary[key] = value
# This function belongs within the hashtable class
def insert(self, key, value):

    slot = hash(key) % self.size
    node = hashnode(key, value)
    node.next = self.slots[slot]
    self.slots[slot] = node


# Equivalent to: dictionary[key]
# This function also belongs within the hashtable class
def get(self, key):

    slot = hash(key) % self.size
    if self.slots[slot] is not None:
        node = self.slots[slot]
        while node is not None:
            if node.key == key:
                return node.value
            node = node.next
```

Stacks

- Concept
  - LIFO data structure
  - Interface:
    - push(X) — places an item on (the top of) the Stack
    - pop() — removes and returns the item from the Stack
    - peek() — returns the item from (the top of) the Stack
    - empty() — True if the Stack is empty (a/k/a “is_empty()”)
- Implementation
  - Want to keep functions to O(1) time
  - In Python, common to use list



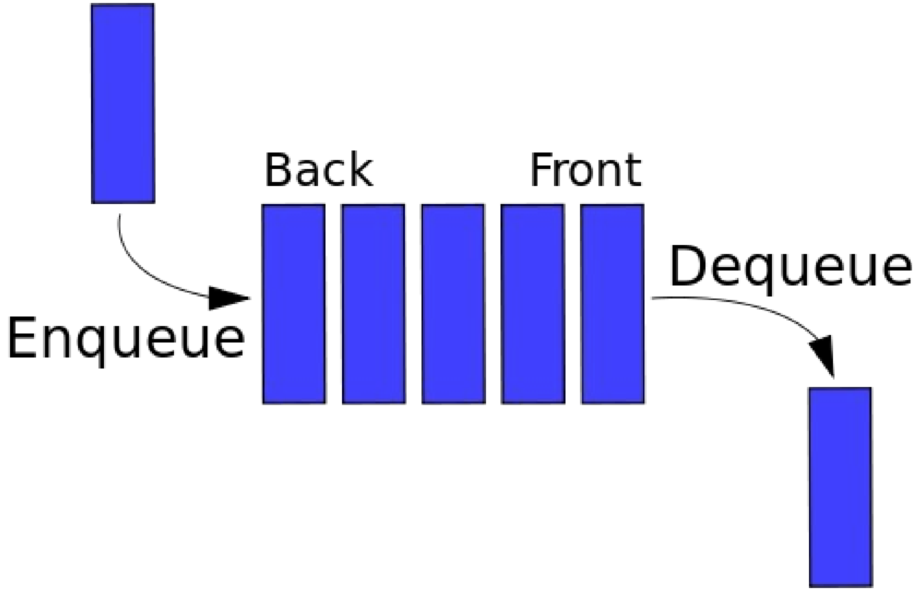
Stack Function	List Function
push(X)	`append(X)`
pop()	`pop()`
peek()	`# Write a function`
empty()	`# Use len function`

Queues

- Concept
  - FIFO data structure
  - Interface:
    - enqueue(X) — places an item on (the end of) the Queue
    - dequeue() — removes and returns item from the Queue
    - size() — returns the number of items on the Queue
  - Variation: dequeue allows insertion & removal from both front & back
- Implementation
  - Common to use a dequeue (double-ended queue), constructor:

```
queue = dequeue()
```

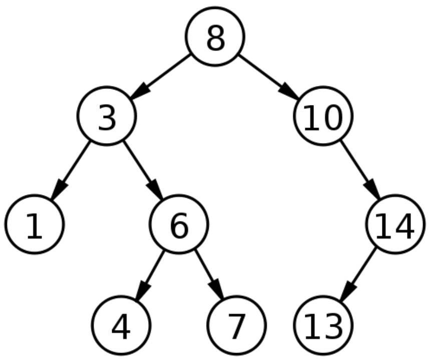
  - May also use a list



Queue Function	dequeue Function
enqueue(X)	`append(X)`
dequeue()	`popleft()`
size()	`# Use len() function`

Binary Search Trees (BST)

- Concept
  - Dynamic data structure with principles from binary search
  - Entire structure needs reference to one element of the tree (root)
  - Interface:
    - find(key) — recursive find key in BST; return associated value
    - insert(key, val) — recursive find to add {key, val} to BST
    - delete(key) — delete node with key from BST
    - traverse() — list all items in BST; 3 possible traversals
  - Visualisations usually only show keys, no associated values:



- Implementation uses two classes:
  - bst — can operate on a tree even if it's empty
  - node — does most of the work

BST — node implementation

```
class node:

    def __init__(self, key, val):
        self.key = key
        self.val = val
        self.left = None
        self.right = None

    def get(self):
        return self.val

    def set(self, val):
        self.val = val
```

BST — public implementation

```
class bst:

    def __init__(self):
        self.root = None

    def insert(self, key, val):
        if self.root is None:
            self.root = node(key, val)
        else:
            self.insert_node(self.root, key, val)

    def find(self, key):
        return self.find_node(self.root, key)

    def traverse_in_order(self):
        return self.in_order(self.root, key)
```

BST — find_node

Concept:

- Compare current node's key to target
- Recursively look left or right depending on value compared to target

```
def find_node(self, current_node, key):
    if current_node is not None:
        return False # Maybe return some other value
    elif key == current_node.key
        return current_node.val
    elif key < current_node.key:
        return self.find_node(current_node.left, key)
    else:
        return self.find_node(current_node.right, key)
```

BST — insert_node

Concept:

- Descend the tree (going left or right) until getting to an empty node
- Create new node and attach it to the tree

```
def insert_node(self, current_node, key, val):
    if key <= current_node.key:
        if current_node.left is not None:
            self.insert_node(current_node.left, key, val)
        else:
            current_node.left = node(key, val)
    elif key > current_node.key
        if current_node.right is not None:
            self.insert_node(current_node.right, key, val)
        else:
            current_node.right = node(key, val)
```

BST — traverse

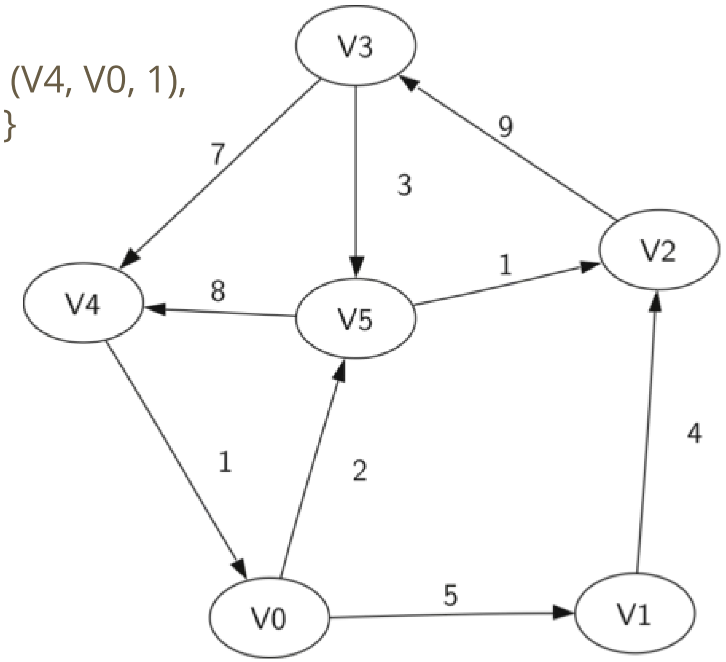
Concept: process (print) everything to the left, the current, everything right

```
# Assuming the traversal action is printing
def in_order(self, current_node):
    if current_node is None:
        return # Nothing to do
    self.in_order(current_node.left)
    print(current_node.val)
    self.in_order(current_node.right)
```

Graphs — Concept and Storage

- Vocabulary: *G = (V, E)*
  - *Vertices* / nodes; may have a key / name + additional information
  - *Edges* connect vertices; may be weighted or unweighted; directed or bidirectional
  - *Path* — a sequence of vertices connected by edges (in the correct direction)
  - *Cycle* — a path which begins and ends with the same vertex
- Example
  - V = {V1, 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) }
- May be stored as a matrix (below) or list

	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	



Graphs — One Implementation

```
class vertex:

    def __init__(self, key):
        self.id = key
        self.connected_to = {}

    def add_neighbor(self, nbr, weight=0):
        self.connected_to[nbr] = weight


class graph:

    def __init__(self):
        self.vertices = {}

    def add_vertex(self, key):
        v = vertex(key)
        self.vertices[key] = v

    def get_vertex(self, key):
        if key in vertices:
            return vertices[key]
        return None
```

```
def add_edge(self, src, tar, weight):
    if src not in self.vertices:
        self.add_vertex(src)
    if tar not in self.vertices:
        self.add_vertex(tar)
    sv = self.vertices[src]
    sv.add_neighbour([tar], weight)
```

Task: Find Shortest Path

- Task
  - Given: two vertices, (V_S and V_T) and G = (V, E)
  - Find and return a (the shortest) path starting at V_S and ending at V_T
- Algorithm: Breadth First Search
  - Visit adjacent vertices recursively, starting from V_S until V_T is found
  - Keep a queue of candidate nodes to visit, set of nodes already visited
  - Some implementations require additional methods to Vertex class
  - Running time = O(V + E)
  - Ignoring V_T, the same algorithm can create a (minimum) spanning tree from V_S

BFS in Python

```
# Implemented within graph class
def bfs(self, start):
    queue = [(start, [start])]  # Keep track of path back to start
    while queue:
        vertex, path = queue.pop(0)
        for next in self.get_vertex(vertex) - set(path):
            if next == target:
                yield path + [next]
            else:
                queue.append((next, path + [next]))
```

Resources

Time complexity — https://www.youtube.com/watch?v=8syQKTdgdzc

- See Problem Solving with Algorithms and Data Structures using Python