CS and Data Structures Assessment 1

Concept Review

Runtime Concepts

- Big O Notation
- Why it's important to think about runtime
  - Interviews (be able to describe the runtime of your own solutions)
  - A language to describe algorithm efficiency, and compare alternate solutions
- Runtimes to know
  - Linear, or O(n): as size of work increases, length of time the algorithm takes increases linearly
  - Constant, or O(1): even when size of work increases, length of time is always the same
  - Logarithmic: binary searches (looking up a word in a dictionary, for example); the work is divided in half with each iteration of the algorithm
    - O(log n)
    - O(n log n) (do something with ***O(log n) n*** times)
  - Quadratic, or O(n^2): nested for-loops, typically
  - Exponential: password cracking algorithms, for example
    - O(2^n)
  - Factorial, or O(n!)

Stacks and Queues

- Concepts, not implementations
  - Stacks and queues are ways to manage and process information
  - You can build a stack or queue out of different implementations (lists, linked lists, etc)
- Stack: LIFO
- Queue: FIFO

Linked Lists

- A data structure for storing an iterable collection

- Can be used similarly to a regular Python list ("array")
- But have performance differences

Trees

- Trees: storing hierarchical data (org charts, taxonomies)
- Binary Search Trees: searching a binary tree using a "rule"
- Navigating trees using BFS and DFS

Part 1: Discussion Questions

Runtime

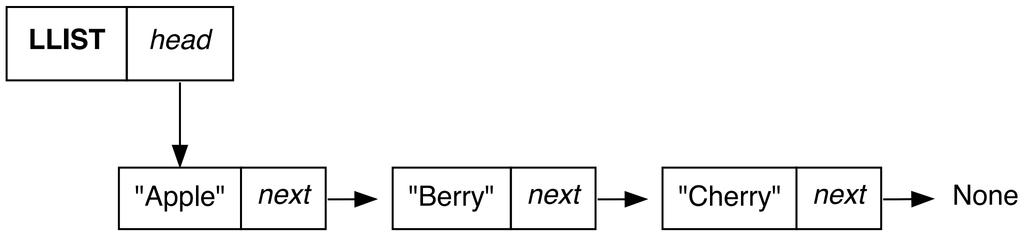
1. When calculating the Big O notation for a particular algorithm, it's necessary to consider the length of time it takes for the algorithm to run as the algorithm's workload approaches infinity. You can think of the workload as the number of tasks required to complete a job. What determines the **workload** of figuring out whether your box of animal crackers contains an elephant?
2. Order the following runtimes in ascending order by efficiency as ***n*** approaches infinity:
   - O(log n),
   - O(n^2)
   - O(n log n)
   - O(n)
   - O(2^n)
   - O(1)

Stacks and Queues

1. In the following cases, would a stack or queue be a more appropriate data structure?
   1. The process of loading and unloading pallets onto a flatbed truck
   2. Putting bottle caps on bottles of beer as they roll down an assembly line
   3. Calculating the solution to this mathematical expression: `2 + (7 * 4) - (3 / 2)`
2. Describe two more situations where a queue would be an appropriate data structure.
3. Describe two more situations where a stack would be an appropriate data structure.

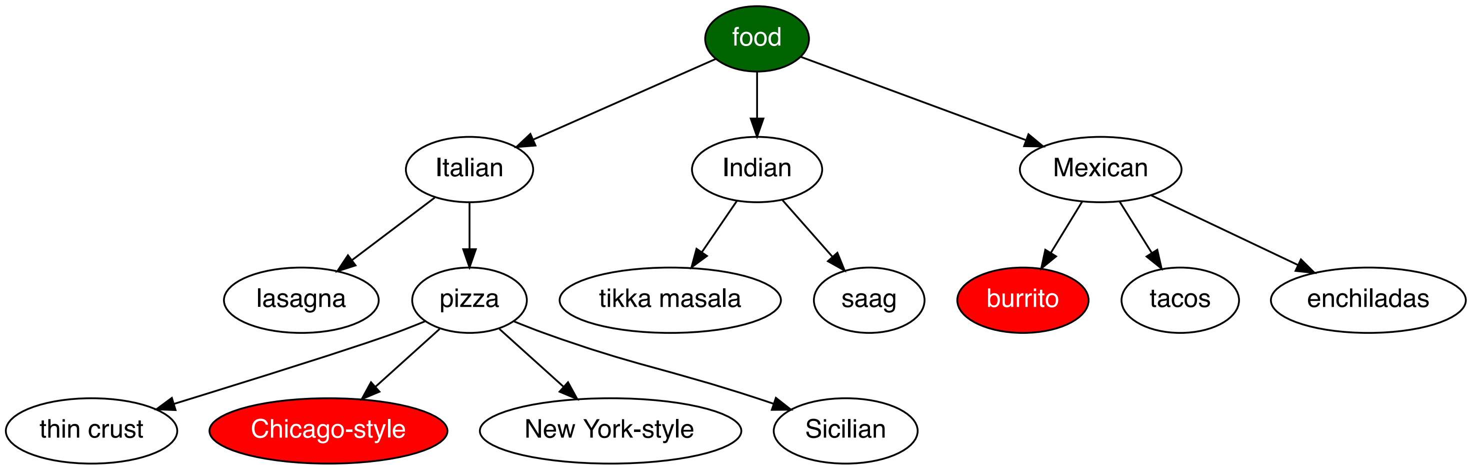
Linked Lists

1. Given the linked list below, which are the nodes? What is the data for each node? Where is the head? Where is the tail? (Please be as specific as possible — exactly which parts of the diagram correspond to each part? Arrows? Boxes? Text?)



2. What’s the difference between doubly- and singly-linked lists?
3. Why is it faster to append to a linked list if we keep track of the ***tail*** as an attribute?

Trees



1. Given the tree above, in what order would a Breadth First Search (BFS) algorithm visit each node until finding ***burrito*** (starting at ***food***)? Just list the order of nodes visited; no need to recreate the state of the algorithm data in your answer.
2. Given the tree above, in what order would a Depth First Search (DFS) algorithm visit each node until finding ***Chicago-style*** (starting at ***food***)? Just list the order of nodes visited; no need to recreate the state of the algorithm data in your answer.
3. How is a binary search tree different from other trees?

Part 2: Skills Assessment

Runtime

In the file called ***runtime.py***, there are 5 functions.

1. What is the runtime for ***string_compare***? Add your answer to the end of the function's docstring.
2. What is the runtime for ***has_exotic_animals***? Add your answer to the end of the function's docstring.
3. What is the runtime for ***sum_zero_1***? Add your answer to the end of the function's docstring.
4. What is the runtime for ***sum_zero_2***? Add your answer to the end of the function's docstring.
5. What is the runtime for ***sum_zero_3***? Add your answer to the end of the function's docstring.

Stacks

In the file called ***stack.py***, there is a class definition, along with some methods for a ***Stack*** class.

6. Finish the method called ***length***.
7. Finish the method called ***empty***.
8. Finish the method called ***is_empty***.

Queues

In the file called ***queue.py***, there is a class definition, along with some methods for a ***Queue*** class.

9. Finish the method called ***enqueue***
10. Finish the method called ***peek***.

Linked Lists

In the file called ***linkedlist.py*** there is a class definition for a ***Node*** in a linked list, as well as a class definition for a ***LinkedList***. There are methods already defined for ***remove_node_by_index***, ***find_node*** (by data), and ***add_node***.

11. Finish the method called ***print_list***.
12. Finish the method called ***get_node by index***.

Trees

In the file called ***tree.py*** there is a class definition for a ***Tree***, which we might use for keeping track of an organizational chart at a company. There is also a class definition for a ***Node***, which would represent an employee. Each node has an attribute called ***children***, which is a list of other instances of nodes. In the real world, a node's ***children*** would represent the people whom that node/person supervises.

13. In the ***Node*** class, finish the method called ***get_num_children***. This function could be used to count the number of people whom person supervises.

14. In the ***Tree*** class, finish the method called ***breadth_first_search***. Hover over the section below if you need a hint.

Hint

The ***depth_first_search*** function uses a stack to keep track of the nodes that need to be visited. Notice it doesn't actually have to use an instance of the ***Stack*** class, here. Simply initialize an empty list of nodes to visit, and when deciding which node to visit next, take from the *end* of the list of nodes to visit, instead of the beginning. This is a list acting like a stack, but it's a stack nonetheless!

Why does this work? It will guarantee that the nodes that are most recently added to the list of nodes to visit will be visited first. First the root's children will get added, and directly after that, some of the root's grandchildren (the ones belonging to the first child), and then directly after that, some of the root's great grandchildren (the ones belonging to the first child's first child), and so on down the line.

In order to implement breadth first search, you'll need to use a queue to keep track of which nodes to visit. This means that you'll add to the end, but *take from the beginning*.