Computer Science Foundation Outline

Lecture 1: Linear Data Structures

- Implement a singly linked list and discuss the worst-case time and space complexity of iterating over the list, prepending and appending an item to it.
- Explain how arrays are different from linked lists and how they are used to implement numpy arrays and Python lists.
- Compare the system performance of numpy arrays and built-in Python lists for iteration, prepending and appending a single element, outlining potential causes of the performance differences.
- Implement an algorithm that solves uses either a singly linked list, Python list or a numpy array to solve a logic puzzle, factoring in the relative performance of the data structures in the context of the problem.

Lecture 2: Stacks and Queues

- Implement a stack and discuss the time and space complexity order of its operations.
- Implement a simple queue and discuss the time and space complexity order of its operations..
- Implement a priority queue and discuss the time and space complexity order of its operations.
- Implement an algorithm that uses either a stack or a priority queue for a scheduler, factoring in the relative performance and flexibility of the data structures in selecting the algorithm.

Lecture 3: Strings

- Use ASCII to map characters to integers, noting the necessity for and examples of Unicode standards and systems that implement them.
- Use at least one Unicode standard to specify the representation of non-Latin letters and emotion icons in terms of Unicode code points
- Compare and justify the system performance of concatenating strings using the '+' operator and using the string builder pattern in Python with the join method
- Use the string builder pattern to efficiently construct long bodies of text

Lecture 4: Hash Tables

- Explain the general procedure followed by hash functions in terms of hash codes and buckets.
- Explain how hash collisions occur, affect the time complexity order of lookups and are resolved in a hash table.

- Implement a hash table using Python's built-in hash function and numpy arrays, and use resulting the hash table implement an in-memory cache for data stored for a file stored on-disk.
- Compare, in terms of features and performance, the bespoke hash table implementation to that of Python dictionaries.

Lecture 5: Trees

- Implement a binary tree and functions to insert and verify if an element exists in a tree and discuss the time and space complexity order of the functions.
- Implement a rose tree and functions to insert and verify if an element exists in a tree and discuss the time and space complexity order of the functions.
- Use a binary tree to implement a heap and use a stack to support an iterative algorithm that inserts and retrieves values, discussing the time and space complexity of the iterative algorithm.
- Compare the time and space complexity of the iterative traversal of the heap to a recursive solution, noting the absence of the stack in the recursive solution.

Lecture 6: Graphs

- Describe different types of graphs in terms of direction, cycles and weights and discuss the implications for various applications.
- Calculate the degrees of a node and the path length between two nodes in a graph.
- Implement Prim's minimum spanning tree algorithm for an unweighted undirected graph and discuss what it makes it a greedy algorithm.
- Simulate a social network that supports addition of new unique members and reporting on the total of number of members and the shortest number of connections between two members.

Lecture 7: The Stack and the Heap

- Explain how the stack and heap are used to support memory allocation in a finite memory space.
- Outline the differences between values and referenced data, and the role they play in decision to allocate on either the stack or the heap.
- Explain how a stack overflow occurs with recursion in Python and how it can be mitigated by increasing the max recursion depth and using an iterative equivalent, while noting the absence of tail recursion in Python
- Simulate the memory model using a stack and a heap allowing for instructions to allocate values and referenced data on the stack and heap.