Data Structure

- Big O
    - Always consider the efficiency/ complexity trade-off in terms of time and space
    - The notation “O()”, e.g. O(n) - big O of n
    - Formally we consider the worse case as the upper bound (there are the worse case, average case, and best case)

- List-based collections
    - Collections don’t have order, but collections can have different objects
    - Lists
        - Lists have order and can have different objects
    - Arrays
        - Arrays are the implementation of lists
        - Arrays have indices
        - Inserting in an array is not efficient - O(n)
    - Linked-lists
        - Have order but no indices - by its links that indicating the next item
        - Adding and removing items are very easy
        - Doubly linked-list - has a link indicating the previous one and a link that indicating the next item
    - Stacks (LIFO - last in first out)
        - Easily add items on top and see the item on the top
        - Good to use them when you only care about the most recently added item
        - “Push” - insert, “Pop” - remove
            - In python - “push” append(); “pop” pop()
    - Queues (FIFO - first in first out)
        - Enqueue - add one
        - Dequeue - delete one
        - Peek - show the oldest one
        - Deque - a double-ended queue

- Searching & sorting
    - Binary search
        - Searching from the middle O(log(n)) which is awesome
        - Elements need to be in the increasing order
    - Recursion
    - Sorting
        - Naive approach
            - Bubble/Sinking sort (in-place sorting): time - O(n^2), space - O(1)
            - Merge sort (divide & conquer): time - O(log(n)), space - O(n)
            - Quick search** (pivot from the last element):
                - time - worse case O(n^2), avg case O(nlog(n)), space - O(1)

- Maps & hashing
    - Maps (also called dictionary in Python): key and value, don't allow repeated elements
        - Arrays - list-based data structure; Maps - set-based data structure
    - Hashing functions***: doing lookups in constant time!! (while in above concepts we need to do a linear time search)
        - Values are converted to hash values (the index of an array) through hashing functions
            - E.g. taking the last few digits of a serial number - %n as hashing function - getting the remainder as the index to do constant time search
            - Collisions: two values happen to be the same hashing value
                - Two solutions here need to be balanced (trade-off solutions)
                - One is to change the whole function or change the value in the function
                    - Time: O(1), Space: largely increased :(
                - Two is to keep the hash function but change the array structure - buckets that stores collections (or hash functions in hash function)
                    - Time: O(n), Space: not increasing :)
                    - Use the load factor to calculate how "full" the bucket is. Load factor -> [0,1] -> 1 means fully used
    - Hash maps (a Python dictionary is a hash map)
        - <key, value> → hash value with <k, v> through hash function on KEY
        - Always consider using a hash function when dealing with data structures because it's so efficient
    - String keys: using hash functions by converting the letters through ASCII values
        - Python function, ord() - get the ASCII value for the letter
        - chr() - get the letter associated with an ASCII value

- Trees (an extension of a linked-list) (connected without cycles)
    - Terms:
        - Root, nodes, leaves, edges (the connections), path (a group of edges)
        - Parent: a node at a lower level; child: a node connected to a parent at a higher level
        - External node: leaves; internal node: parents
        - The height: from low to high; the depth: from high to low (both begin from 0)
    - Tree traversal
        - DFS (depth-first search)
        - BFS (breadth-first search) (through level order)
    - Binary tree (a tree with no more than two leaves per node)
        - Seach: O(n), delete: O(n); insert: O(log(n))
        - BST (Binary search tree): O(log(n))
        - Unbalanced BST: O(n)
    - Heaps (all the tree values are sorted desc/asc)
        - Seach: O(n); insert/delete: O(log(n))
        - Max heaps O(1); min heaps

- Graphs/ networks (a data structure designed to show relationships between objects)
    - Both vertex and edges contain data (edges often contain data about the strength of a connection)
    - Directed graph: a graph where the edges have a sense of direction
    - Cycles can exist in a graph but they are dangerous for loops
    - Connectivity tells which graph has stronger connection
    - Graph representations
        - Edge list [[2,3], [1,5], [5, 0], [3,9]] 2D list
        - Adjacency list (shows the adjacency node of the indicated index): [[1], [2, 3, 5], [5, 8]]
        - Adjacency matrix (shows the members of edges linked to the specific index)
    - Graph traversal
        - DFS can start from any vertex, O(|E|+|V|) -- seen and store in a stack
        - BFS O(|E|+|V|) -- seen and store in a queue
    - Eulerian Paths

Notes: Jiaying Wu, Feb 20, 2019
Source: Algorithms & Data Structure for Technical Interviewing (Python)
https://classroom.udacity.com/courses/ud513/lessons/7174469398/concepts/71212749920923