## 1. Arrays and Lists

a. Arrays and Lists contain a sequence of values which are then stored adjacently in memory.

**Time Complexities:**

- **Access - O(1)**: accessing an element in the list requires the index
- **Search - O(n**): searching if an element exists in the list requires, in the worst case to traverse the entire list one index at a time
- **Insertion/Delete - O(n):** both require the element to be found first which is the dominant constant

**Common Questions:** Reverse a linked list, Find the middle element of a linked list, find if the list is circular

## 2. Linked Lists

a. Opposed to Lists and Arrays, linked lists do not have their order defined by their physical placement in memory. Instead each linked list element contains both the values and the address (or pointer) to the next element. So you can only traverse them sequentially and not accessible via an index like you would be able to in an array. AKA: you can only know the full list length after traversing the entire list, and the last element always has null/none as its pointer, and furthermore the linked list is defined by the pointer pointing to the first element.

**Time Complexities:**

- **Access - O(n)**: accessing an element in the list requires in the worst case to traverse the whole list
- **Search - O(n**): searching if an element exists in the list requires, in the worst case to traverse the entire list one index at a time
- **Insertion/Delete - O(1):** Inserting and deleting an element given the pointer where it needs to be inserted/deleted only requires to rearrange the pointers. **However** inserting/deleting and element at the end is **O(n)**

**Common Questions:** Reverse a linked list, Find the middle element of a linked list, find if the list is circular

## 3. HashTables

a. Hashtables are extremely useful in making efficient algorithms. In most problems you Should think of the DS. They try to map a data type to another data type creating a key,value Pair. Which can then be accessed in constant time. We love anything that is in constant Time. For each pair the key is hashed to attempt to create a unique address for the value to be Stored in memory. But one must be weary of collisions. Simply put, the hash function Generates the same address for different keys. But it is then handled by creating a linked list from them, storing them both. And then well you travers that linked list to get the value by its key

**Time Complexities:**

- **Search - O(1)**: Searching if a key exists in a hash table requires on average constant time.
- **Insert/Delete - O(1):** Still constant time because we never go through the dictionary, we simply check if it exists. If so delete/add otherwise nothing.

**Common Questions:** Find if two elements in the list sum to a target value, group anagrams, Longest substring without repeating characters

## 4. Queue

a. Or in other words think about the line at a register, **First there First Out (FIFO)**. Elements can only be accessed in the same order as they were inserted into the queue. Often used to go through trees and graphs. The common operations are **Enqueue()** and **Dequeue() --- Think Lines at a store**

**Time Complexities:**

- **Access - O(n):** Accessing and element in the queue requires (in the worst case) to Dequeue() each element one by one.
- **Search - O(n):** For the same reason above, worst case we may need to Dequeue() every element and compare it to the search target
- **Insert/Delete - O(1):** Adding and removing can always be done in constant time.

**Common Questions:** Binary Tree right side view, task scheduler, shortest subarray with sum at least K

## 5. Stack

a. Like a queue this is also a sequential data structure. However this follows the practice of **Last in First Out**, which means elements can only be accessed in reverse order as they were inserted into the stack. The common operations are **push()** and **pop(). Think of it like a stack of plates, you can't get to the bottom plate until you pop() the plate above it.**

**Time Complexities:**

- **Access - O(n):** Worst case you want to get to the bottom of the stack (ie the initial value pushed() onto it) so you will have to pop() every value on top of it.
- **Search - O(n):** For the same reason above worst case you want to get to the bottom of the stack, going thru n values
- **Insert/Delete - O(1):** Since we can only access the top of the stack there is no traversing so inserting/deleting values is constant.

**Common Questions:** Valid Parentheses, trapping rain water, exclusive time functions

## 6. Trees (Binary)

a. To keep it simple we are only going to use binary trees. If anyone but me is using this doc, it should suffice. Unless you're applying for an intense position, then why are you here? This is way too basic. But a binary tree is a data structure that maintains a hierarchical relation between its elements, **parent** and **children nodes.** Each node can have at most two childered, a right and left child.

**Root Node:** The node at the very top is calle the root. A tree is defined by the pointer pointing to this node.

**Parent Node:** Any node that has at least one node branching below it, child node

**Child Node:** The successor of a parent node is known as the child node. A node can both be a child and parent. The root is never a child

**Leaf Node:** This is a node that has no child nodes

**Subtree:** Any valid subset of the original tree, it a mini part of the tree

**Path:** The sequence of nodes along the edges of subtree is known as the path

**Traversing:** Passing through the nodes in a certain order (**Breadth/depth first search)**

**Common Questions:** Validate Binary Search Tree, BT Level order traversal, Lowest common ancestor of a BT

## 7. Graphs

a. A graph is simply a group of nodes consisting of vertices and edges. A graph **G** is a pair of sets **(V,E)** where **V** is the set of all the vertices and **E** is the set of all the edges. Opposed to trees Graphs can be cyclic, which means you can end up at your starting node if you traverse through the graph

**Common Questions:** Clone Graph, Is a graph a bipartite