AlgoExpert Notes

# Data Structures and Algorithms

## Introduction

This is a little knowledge sheet that I have made to learn about the underlying background around each data – structure.

## What are Data Structures

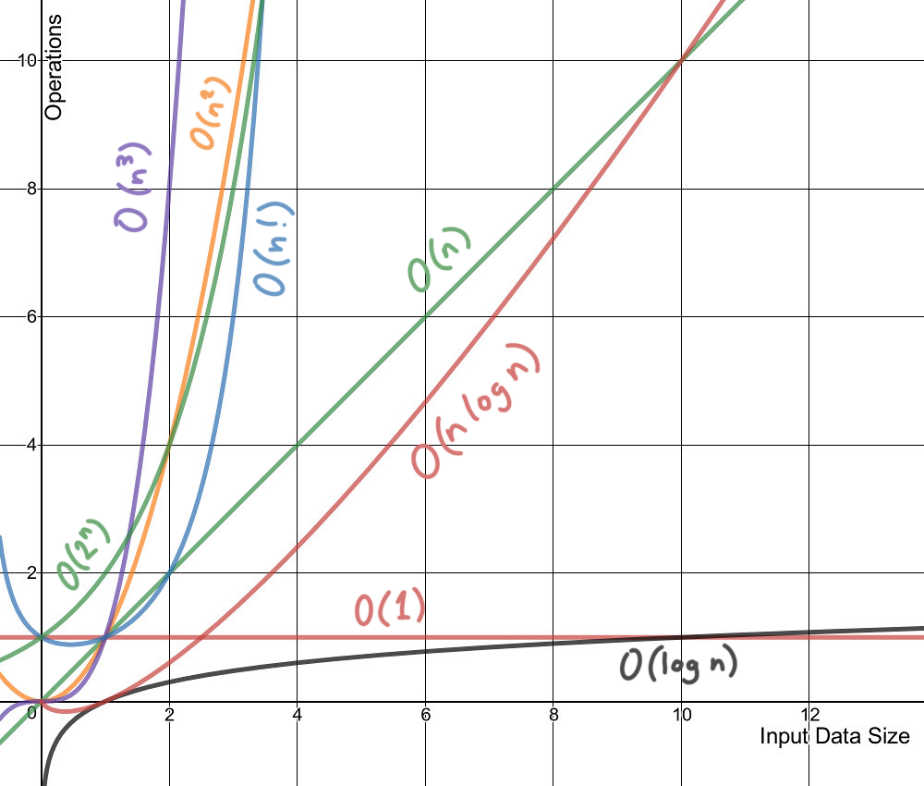
Coding is essentially manipulating data in a way in which we can accomplish something. Data structures are really just a way to organise and manage data. A **data structure** is a specialized format for organizing, processing, retrieving and storing **data**. While there are several basic and advanced **structure** types, any **data structure** is designed to arrange **data** to suit a specific purpose so that it can be accessed and worked with in appropriate ways.

## Complexity Analysis

The **complexity** of an **algorithm** is a function describing the efficiency of the **algorithm** in terms of the amount of data the **algorithm** must process. ... Time **complexity** is a function describing the amount of time an **algorithm** takes in terms of the amount of input to the **algorithm**

## Big O notation

This is how we donate the time complexity of an algorithm, below shows an example for simple algorithms and their associated big O notation.

B1 = 1 + a[0] – O(1)

B2 = sum(a) – O(n)

B3 = pair(a) – O(n2)

If we were to run all the algorithms in the same method the overall time complexity would be simplified to that of B3.

## Logarithm

What it really means….. The base is generally always assumed to be 2. (binary logarithm)

Logb(x) = y if and only if by = x

Log2(N) = y if and only if 2y = N

When you double N, you’re only increasing the increment by 1.

24 = 23 \* 2.

Algorithms that eliminate half of the input at every step tend to be logarithmic in time complexity

## Arrays

When arrays select memory they always ensure that they get back to back memory slots (every piece of memory store 8 bits, so if we had to store an array of 4 integers (32 bit ints) we would need 4 \* 4 (16 slots of memory)

* Indexing – it knows the width of one element, and the index yours specifying – O(1) Time
* Searching – O(N) Time , O(1) Space (Static)
* Insert – O(N) Time, O(1) Space (Static)

Two types of arrays: - Amortized Analysis

**Static**: fixed number of memory slots (as described above)

**Dynamic**: is an array that can change in size (arraylist), normally allocates twice the amount of space subconsciously. If you fill this, it will copy itself to another array of double the size. Example an array of two would exactually have 4 slots, if we fill this then it will become 8 slots.

* Worst Case Insertion is O(N)

## Linked Lists

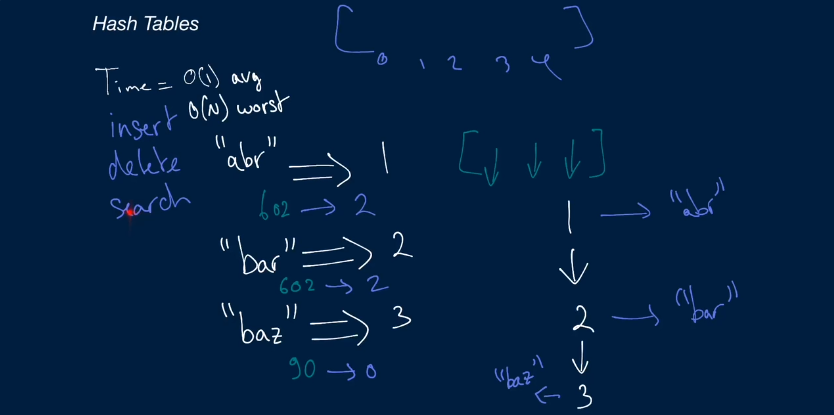
Linked Lists tend to read from left to right, they have a pointer to their next node. The last node points to null. The difference between this and arrays is how they’re stored in memory, in the case of linked lists, memory allocation is not done in a consecutive manner. They can be stored anywhere in memory and are connected via their pointers (references the next memory slots address)

* Indexing – O(i) Time as you have to move through I elements to find the one you’re looking for
* Init – O(n) ST.
* Searching O(N) time, O(1) space.
* Insert – O(1)

They can be doubly-linked, that is that each value has a pointer that points to the node before and another pointer that points to the value in front. (Head and Tail)

## Hash Tables

Most questions involve using hash-tables, they’re very important to learn! Behind the scenes when you’re inserting a values you transform the key using a hash-function into an index that fits in an underlying array. You then transform in back when retrieving it. Sometimes we have indexes that are the same value (collision) – To prevent this our array points to linked lists instead of indexes. See example below. In average however, the values are almost always separated in their own linked list.



* Insertion – O(1) Time, O(N) worst case
* Deletion – O(1) Time
* Searching – O(1) Time

If you’re underlying array does have enough space, you can implement a hash-table that resizes itself.

## Stacks and Queues

A stack is a data structure that allow inserting and removing LIFO (Last in first out). A Queue is basically the opposite of a stack FIFO (First in, first out). They’re essentially a list of elements.

**Stack:**

* Insertion / Delete – O(1) ST
  + Push (add)
  + Pop (remove)

**Queue:**

* Typically implemented with a linked list
  + Keep track of head and tail
  + Enqueue (Insertion): replace the head - O(1)
  + Dequeue (remove): remove the tail – O(1) as long as we have reference to the tail

**Searching** – O(N) Time, O(1) Space

## Strings

A string is typically stored in memory as an array of integers. The way this is done is through some sort of character encoding standard (Ascii , A -> 65, a ->97 ETC). When we’re dealing with a single character in a string it is all constant time.

Traversing O(N) Time, O(1) Space

Copy – O(N) ST

Get -O(1)

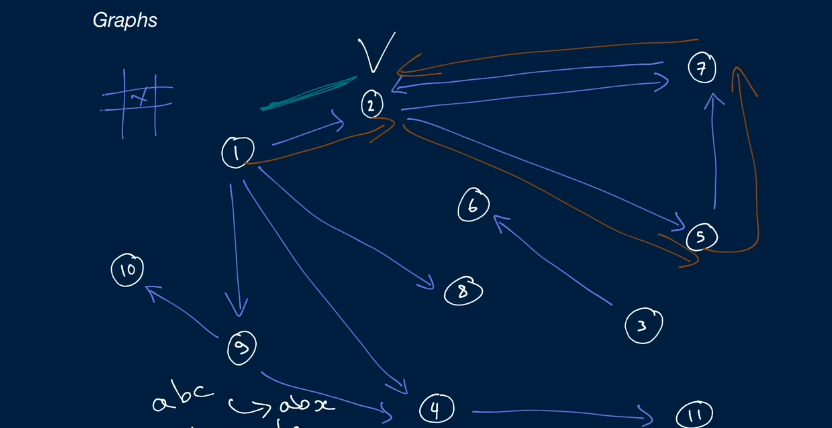
In java, String are Immutable. This means you cannot alter them after you create them, essentially you create a brand-new string every time. If you use a stringbuilder it is O(1) (imagine an array list), otherwise if you were to += on a string it would O(N). If you were adding two strings together for example, “ab” + “cd” would be O(MN).

## Graphs

A graph is a collection nodes that may or may not be connected to each other (collection of edges connections (Arrows) and vertices(nodes(vertex))). Sometimes some nodes are disconnected from each other which means the graph would be denoted as “disconnected” or not connected. Arrows imply direction, for example if you had an arrow pointing from 1 to 2, you could go to 2 from 1 but you couldn’t go from 1 to 2. Some graphs are “directed”, an undirected graph would not have these directions (just a straight-line). if a graph does not have cycles (if at any area in the graph where you have 3 or more nodes that go in an infinite loop you have a cycle) “cyclic graph” - “acyclic graph”.

* If you hit a node you’ve already visited you know you’re in a cycle
* Cyclic graphs are really important, you want to make sure that if you’re stuck In a loop you’d want to find a way of marking nodes you’ve visited (look out for them in questions)

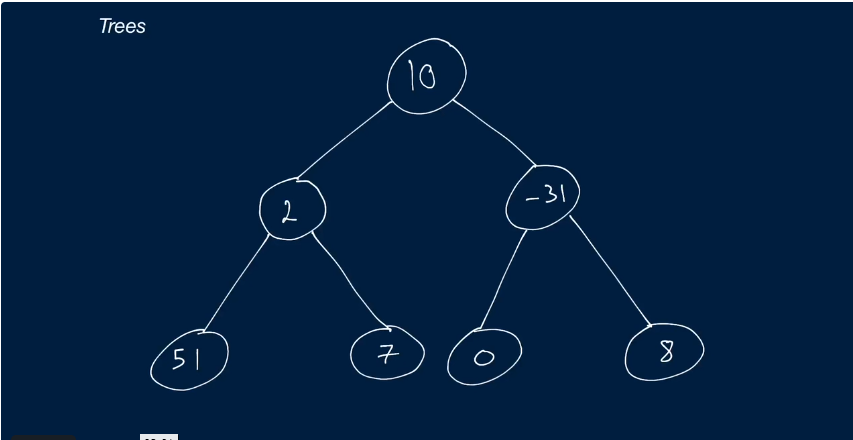
To represent a graph in code we typically use an adjacency-list. Essentially we could has a list of nodes or hash table where every key points to its relevant node and every node has a list of its edges



So the nodes 1 would have a list of (2,4,8,9). Its very easy to do, edges are essentially pointers under the hood. Space -> O(V+E). Search Methods Breadth (each level at a time), Depth first search (root, left,right). Check out the questions (Know them very well) – Time -> O(V+E).

## Trees

A tree is a type of graph, in the context of coding interviews they’re quite simple. More specifically when we talk about a tree, we’re referring to a graph structure that is rooted and has child nodes.



For example, 10 is the root node and has 2/-31 as its child nodes. (does not have cycles). K-ary trees are trees where every node has at most k -children. Other examples (Min heaps, Max heaps, Binary Trees, Binary Search Trees, Tries (tree-like data structure that stores characters in a string)) – Lots of different types of trees.

Traversing through all nodes – O(N) ST.

One subtree – O(log(N) ST

# System Design

## Client – Server Model

A client is essentially a thing that talks to computers, a server is a thing that talks to clients (vice-versa). For example, imagine a browser(client) and a algo Expert(Server). HTTP request will request information and the server will send response via the return address stored in the original request.

**Client -** A machine or process that requests data or a service from a server (A single machine or software could be both a client and a server)

**Server –** A Machine or process that provides data or service for a client, usually by listening for incoming network calls.

**IP Address** – An address given to each machine connected to the public internet

**Port –** In order for multiple programs to listen for new network connections on the same machine without colliding they pick a port to listen to. (2^16 in total)

**DNS –** Domain Name System, it describes the entities and protocols involved in the translation of a DN to IP address.

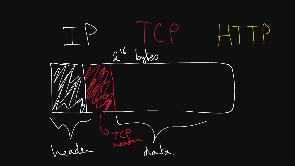
## Network Protocols

For communication between machines, this allows us to understand how a system does this.

**IP** – Stands for Internet Protocol. This network protocol outlines how almost all machine-to-machine communications should happen in the world. Other protocols like TCP, UDP and HTTP are built on top of IP. They are made up of bytes (memory allocation, 8 bits = 1 byte)

* **Header:** contains the source IP address (where its coming from) and the destination IP address (where its going), it also contains how much data the request contains the and the version of IP being used (IPv4 /IPv6 etc).
* **Data:** Can contain up to 2^16 bytes (isn’t actually that much) – in this case you would have multiple IP packets (which makes things more complicated, that is you don’t have a way of guaranteeing that all of these packets will be delivered or ordered correctly).

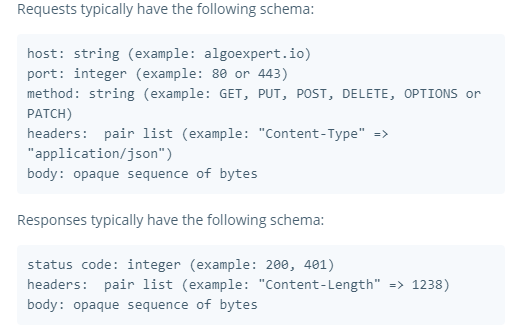
To overcome this, we then apply the TCP header which takes up more bytes in the packet. This will include information about the ordering of packets. The core idea about TCP is that when first accessing as server you will create a TCP connection (Handshake), a special TCP interaction where the two services send a few packets between each other to establish a connection. If one of the two machines don’t send anything in a given period, the connection will time out (or its ended by a server/client). However, what It lacks is a robust framework that allows developers to define meaningful and easy to use communication channels for clients and servers in a system.



This is where http comes into play, which is a protocol that was built on top of TCP which introduces a higher-level abstraction above the two previous protocols known as the request-response paradigm (see screenshot in http section). Essential for implementing business logic!

**TCP –** Network protocol built on top of the IP. Allows for ordered, reliable data delivery between machines over the public internet by creating a connection. TCP is usually implemented in the kernel, which exposes sockets to applications that they can use to stream data through an open connection.

**HTTP –** The Hypertext Transfer protocol is a very common network protocol implement on top of TCP. Clients make HTTP requests and server respond with a response.



**IP Packet –** Sometimes more broadly referred to as just a network packet, an IP packet is effectively the smallest unit used to describe data being sent over IP, asides from bytes. An IP pack consist of:

* An IP Header, which contains the source and destination IP addresses as well as other information just related to the network
* A payload, which is just the data being sent over the network

## Storage

Information storage is incredibly complex and is an extremely important concept to understand!

**Databases:**  Are programs that either use disk or memory to do 2 core things: record data and query data. In general, they are themselves servers that are ling lived and interact with the rest of your application through network calls, with protocols on top of TCP or even HTTP. Some Databases only keep records in memory, and the users of such databases are aware that those records may be lost forever if the machine or process dies.

* For the most part though, databases need persistence of those records and thus cannot use memory. This means that you must write your data to disk. Anything written to disk will remain through power loss or network partitions, so that’s what is used to keep permanent records.
* Since machines die often In large scale systems, special disk partitions or volumes are used by the database processes, and those volumes can get recovered even if the machine were to go down permanently.

**Disk:** Usually refers to either HDD(hard-disk drive) or SSD(Solid-state drive). Data written to disk will persist through power failures and general machine crashes. Disk is also referred to as non-volatile storage. SSD is far faster than HDD but also far more expensive from a financial point of view. Because of that, HDD will typically be used for data that’s rarely accessed or updated, but that’s store for a long time and SSD will be used for data that’s frequently accessed and updated.

**Memory:** Short from random access memory (RAM). Data stored in memory will be lost when the process that has written that data dies.

**Persistent Storage:** Usually refers to disk, but in general it is any form of storage that persists if the process in charge of managing it dies.

A Database is not a magical box, what it really is, is a server.

### Latency & Throughput

If you’ve ever experience lag in a video game, it was most likely due to a combination of high latency and low throughput.

**Latency:** The time It takes for a certain operation to complete in a system. Most often this measure is a time duration, like milliseconds or seconds. You should know these orders of magnitude:

* **Reading 1 MB from RAM**: 0.25ms
* **Reading 1 MB from SSD:** 1ms
* **Transfer 1 MB over 1gbps network:** 10ms
* **Reading 1MB from HDD:** 20ms
* **Inter-continental round trip:** 150ms

**Throughput:** The number of operations that a system can handle properly per time unit. For instance, the throughput of a server can often be measured in requests per second.

Depending on what context we’re talking about, latency will still apply in the same dynamic. When you’re design a system, you’ll typically want to optimise them by lowering the latency. However certain systems might not care too much about the latency (Games really do!). Imagine we have 5 clients all requesting from the same server. The throughput would be how many requests the sever can handle at a given time (per second). 50mbs means the network can handle transferring 50megabytes per second. Blindly increasing throughput doesn’t make sense, a better way to fix this system is to have multiple servers handle these requests (cost). You cannot make assumptions about latency or throughput based on the other.

### Availability

The odds of a particular server or service being up and running at any point in time, usually measured in percentages. A server that has 99% availability will be operational 99% of the time (this would be described as having two nines (measured for a given year)).

**High Availability:** Used to describe systems that have particularly high levels of availability, for examples 5 nines of more – Abbreviated to “Ha” typically.

**Nines:** Typically refers to percentages of uptime. For example, 5 nines of availability means an uptime of 99.999% of the time. List of downtimes expected:

* 99 % (two nines) : 87.7hours
* 99.9% (three): 8.8hours
* 99.99%: 52.6 mins
* 99.999%: 5.3mins

**Redundancy:** The process of replicating parts of system in an effort to make it more reliable

**SLA:** Short for “service-level agreement”, an SLA is a collection of guarantees given to a customer by a service provider. SLAs typically make guarantees on a systems availability, amongst other things SLA’s are typically made up of one more SLO’s

**SLO**: Short for “service level objective”, an SLO is a guarantee given to a customer by a service provider. Typically made on a systems availability, amongst other things.

In reality, there is an implied availability guarantee (you expect it to operational all the time). Some systems will have an explicit availability guarantee. It is difficult to ensure high levels of availability (it could come at higher costs such as higher latency or lower throughput). For example, the area in which payments are handled would have a very high availability, whereas the dashboard in which a company can monitor sales would not be as critical to ensure uptime. You as the designer, need to understand what is core to your product and what is not (critical).

You want to make sure that your system doesn’t have single points of failure (remove this by using **redundancy**). For example, if you had clients interacting with a server and a database. In this case you would introduce multiple servers. (add more machines (gets more expensive etc)). Passive redundancy is where you have multiple components at a given layer in a system and if it any point one of the dies, nothing is really going to happen. Active redundancy is where you have multiple machines where only a few handle traffic and if one of these fails the other machines will somehow know that it failed and takeover.