**Data Structures:**

* Data structure (DS) is a way of organizing data to be used effectively
* Essential ingredients in creating fast algorithms
* Code cleaner and easier to understand
* Abstract data type provides only the interface a data structure must adhere to. For example, transportation is the interface, but different modes of transportation can be car, train, or plane

Complexity Analysis:

* **How much time?**
* **How much space does this need?**
* Big-O Notation helps quantify performance to improve efficiency
* Big-O only cares when the program becomes really big
* N = the size of the input. Complexities order smallest to largest

Constant Time: 0(1)

Logarithmic Time: 0(log(n))

Linear Time: 0(n)

Linearithmic Time: 0(nlog(n))

Quadric Time: 0(n^2)

Cubic Time:  0(n^3)

Exponential Time: 0(b^n), b > 1

Factorial Time: 0(n!)

**Static Arrays:**

* A static array is a fixed-length container containing n elements indexable from the range [0, n-1]
* Indexable means each slot/index in the array can be referenced with a number

**When Static Array used?**

1. Storing and accessing sequential data
2. Temporarily storing objects
3. Used by IO routines as buffers
4. Lookup tables and inverse lookup tables
5. Can be used to return multiple values from a function
6. Used in dynamic programming to cache answers to subproblems

**Dynamic Array:**

* Can grow and shrink in size depending on the situation

**Singly and Doubly Linked List:**

* Linked List is a sequential list of nodes that hold data which point to other nodes also containing data (last node is null indicating the list is over)
* **Singly Linked Lists nodes only point to the next node**
* **Doubly Linked Lists nodes point to the previous node and to the next node**

**When are Linked Lists Used?**

* Used in queue and stack implementations
* Great for creating circular lists
* Can easily model real-world objects such as trains
* Used in separate chaining which is present certain Hashtable implementations to deal with hashing collisions
* Often used in the implementation of adjacent lists for graphs

**Terminology:**

* **Head:** the first node in a linked list
* **Tail:** the last node in a linked list
* **Pointer:** Reference to another node
* **Node:** An object containing data and pointer(s)

**Singly Linked List:**

* **Pros**
* Uses less memory
* Simpler implementation
* **Cons**
* Cannot easily access previous elements

**Doubly Linked List:**

* **Pros**
* Can be traversed backward
* **Cons**
* Takes 2x memory

**Stacks:**

* A stack is a one-ended linear data structure which models a real-world stack by having two primary operations, namely, push and pop

**When are Stacks used:**

* Used by undoing mechanisms in text editors
* Used in compiler syntax checking for matching brackets and braces
* Can be used to model a pile of books or plates
* Used behind the scenes to support recursion by keeping track of previous function calls
* Can be used to do a Depth First Search (DFS) on a graph

**Queues:**

* Linear data structure which models real-world queues by having two primary operations, namely enqueue (adding, offering) and dequeue(polling)

**Terminology:**

* Queue front is the first thing in the queue
* Queue back is the last thing in the queue
* Adding elements to the back is called enqueueing
* Removing elements from the back is called dequeuing

**When is the queue used?**

* Any waiting line models a queue, for example, a lineup at a movie theatre
* Can be used to efficiently keep track of the x most recently added elements
* Web server request management where you want first come first serve
* Breadth-first search (BFS) graph reversal

**Priority Queues:**

* A priority queue is an Abstract Data Type (ADT) that operates similar to a normal queue except that **each element has a certain priority.** The priority of the elements in the priority queue determine the order in which elements are removed from the priority queue
* Priority queues only support **comparable data**, meaning the data inserted into the priority queue must be able to be ordered in some way either from least to greatest or greatest to least. This is so that we are able to assign relative priorities to each element

**Heap:**

* A heap is a **tree-based** DS that satisfies the **heap invariant** (also called heap property): If A is a parent node of B then A is ordered with respect to B for all nodes A, B in the heap.
* All heaps must be trees and not contain cycles
* A binary heap is a binary tree that supports heap invariants. In a binary tree, every node has exactly two children. A **complete binary tree** is a tree in which at every level, except possibly the last is completely filled and all the nodes as far left as possible. Nodes fill in from left to right

**When is Priority Queue used:**

* Anytime you need to dynamically fetch the “next best” or “next worst” element

**Turning Min PQ into Max PQ:**

Problem: Often the standard library of most programming languages only provides a min PQ which sorts by smallest elements first, but sometimes we need a Max PQ

* Since elements in a priority queue are comparable, they implement some sort of comparable interface which we can simply negate to achieve a Max heap

**Ways of Implementing a Priority Queue:**

* Priority queues are usually implemented with heaps since this gives them the best possible time complexity
* The Priority Queue (PQ) is an Abstract Data Type (ADT), hence heaps are not the only way to implement PQs. As an example, we could use an unsorted list, but this would not give the best possible time complexity

**Indexed Priority Queue:**

-   An **Indexed Priority Queue** is a traditional priority queue variant which on top of the regular PQ operations supports **quick updates and deletions** of **key-value pairs**

-   It is important to be able to **dynamically update the priority** (value) of certain people (keys)

-   The **Indexed Priority Queue (IPQ)** data structure lets us do this efficiently. The first step to using an IPQ is to assign index values to all the keys forming a bidirectional mapping

-   Create a bidirectional mapping between your N keys and the domain [0, N) using a bidirectional **hashtable**. For example:

**Key                         Key Index (ki)**

**“**Mary”                     0

**“**John”                     1

**“**Phil”                       2

**“**Caleb”                    3

**Indexed Priorit Queue (Abstract Data Type) Interface:**

-   If ʻkʻ is the key we want to update first get the keyʻs index: **ki = map[k]** then use ʻkiʻ with the IPQ

-   Some methods:

-   delete(ki), valueOf(ki), contains(ki), peekMinKeyIndex(), pollMinKeyIndex(), peekMinValue(), pollMinValue(), insert(ki, value), update(ki, value), decreaseKey(ki, value), increaseKey(ki, value)

-   To accommodate a Indexed Priority Queue, you will need a **position map and an inverse map** (inversed lookup table)

**Union Find (Disjoint Set):**

* **Union Find** is a data structure that keeps track of elements which are split into one or more disjoint sets. Two primary operations: **find and union**

**Sets:**

-   There are **no duplicates**. If you add duplicates, the computer will only process one

-   The data in sets are not in any specific order

-   Often used in computer science to find out if data is located in the set since something canʻt repeat

-   Sets often can be represented through Venn diagrams

**When and where is a Union Find used?**

* Kruskalʻs minimum spanning tree algorithm
* Grid percolation
* Network connectivity
* To find if two vertices on a graph are connected
* Least common ancestor in trees
* Image processing

**Kruskalʻs Minimum Spanning Tree:**

Given a graph G = (V,E) we want to find a **Minimum Spanning Tree** in the graph(it may not be unique). A minimum spanning tree is a subset of the edges which connect all vertices in the graph with the minimal total edge cost

**Application:**

1. Sort edges by ascending edge weight

2. Walkthrough the sorted edges and look at the two nodes the edge belongs to, if the nodes are already unified, we donʻt include this edge, otherwise we include it and unify the nodes

3. The algorithm terminates when every edge has been processed or all the vertices have been unified

**Creating Union Find:**

* To begin using Union Find, first construct a **bijection** (a mapping) between your objects and the integers in the range [0,n). This allows an array-based union find

**Find Operation:**

* To find which component a particular element belongs to find the root of that component by following the parent nodes until a self-loop is reached (a node whoʻs parent is itself)

**Union Operation:**

* To unify two elements, find which are the root nodes of each component and if the root nodes are different make one of the root nodes be the parent of the other

**Binary Trees and Binary Search Trees (BST):**

**Terminology:**

* A tree is an undirected graph which satisfies any of the following definitions:

1. An acyclic connected graph (no cycles)

2. A connected graph with N nodes and N-1 edges

3. A graph in which any two vertices are connected by exactly one path

* If there is a **rooted tree,** then we will want to have a reference to the root node of the tree
* It does not always matter which node is selected to be the root node because any node can root the tree
* A **child** is a node extending from another node. A **parent** is the inverse of this(going upwards toward the root)
* **A leaf node** is a node with no parent(they should be at the very bottom)
* A **subtree** is a tree entirely contained within another. They are usually denoted using triangles. Subtrees may consist of a single node
* A **binary tree** is a tree for which every node has at most two child nodes
* A **binary search tree** is a **binary tree** that satisfies the **BST invariant:** left subtree has smaller elements and right subtree has larger elements

**When and where are Binary Trees used?**

* Binary Search Trees (BSTs)
* Implementation of some map and set ADTs
* Red-Black Trees
* AVL Trees
* Splay Trees
* Etc..
* Used in the implementation of binary heaps
* Syntax trees(used by compiler and calculators)
* Treap - a probabilistic Data Structure (uses a randomized BST)

**Inserting elements to a BST:**

* Binary Search Tree (BST) elements must be **comparable** so that we can order them inside the tree
* When inserting an element, we want to compare its value to the value stored in the current node weʻre considering deciding on one of the following:

1. Recurse down left subtree (<case)

2. Recurse down right subtree (> case)

3. Handle finding a duplicate value (= case)

4. Create a new node (found a null leaf)

**Removing elements from BST:**

* Removing elements from a Binary Search Tree (BST) can be seen as a two-step process

1. **Find** the element we wish to remove (if it exists)

2. **Replace** the node we want to remove with its successor (if any) to maintain the BST invariant

**Finding Phase:**

* When searching out BST for a node with a particular value one of four things will happen:

1. We hit a **null node** at which point we know the value does not exist within our BST

2. Comparator value **equal to 0(**found it)

3. Comparator value **less than 0**(the value, if it exists, is in the left subtree)

4. Comparator value **greater than 0**(the value, if it exists, is in the right subtree)

**Removing Phase:**

* When removing there are 4 different cases

1. Case 1: Leaf node:

·  If the node we wish to remove is a leaf node then you can remove the node with no side effect because there are no subtrees

2. Case 2 and 3: Either the left/right child node is a subtree of the node

·   The successor of the node we are trying to remove in these cases will be the root node of the left/right subtree

·   In the case that you are removing the root node of the BST in which case its immediate child left/right becomes the new root

3. Case 4: Node to remove has both a left subtree and a right subtree

·   The successor can either be the **largest value** in the **left subtree** or the **smallest value** in the right subtree

·   This is because they both satisfy BST invariant

**Binary Search Tree Traversals:**

* These three types of traversals are naturally defined recursively:

**Preoder(node):**

**If** node == null: **return**

**print**(node.value)                                                    preorder prints before

**preorder**(node.left)                                                the recursive calls

**preorder**(node.right)

**Inorder(node): (puts values in increasing order)**

**if** node == null: **return**

**inorder**(node.left)                                                   inorder prints between

**print**(node.value)                                                    the recursive calls

**inorder**(node.right)

**Postorder**(node);

**if** node == null: **return** postorder prints after

**postorder(**node.left)                                              the recursive calls

**postorder**(node.right)

**print**(node.value)

Level order Traversal:

* Print the nodes as they appear one layer at a time
* To obtain this ordering we want to do a **Breadth-First Search** (BFS) from the root node down to the leaf nodes
* To do a BFS we will need to maintain a **Queue** of the nodes left to explore
* Begin with the root inside of the queue and finish when the queue is empty
* At each iteration, we add the left child then the right child of the current node to our queue

**Hash Tables:**

* A **Hash table** (HT) is a data structure that provides a mapping from keys to values using a technique called **hashing**
* We refer to these as **key-value pairs.** Keys must be unique, but values can be repeated
* HTs are often used to track item frequencies. For instance, counting the number of times a word appears in a given text
* The key-value pairs you can place in an HT can be of any type not just strings and numbers, but also objects! However, the keys need to be hashable

Key (name)                                     Value (fav color)

“William”        ---------------->        “green”

“Micah”             ----------------->        “purple”

“Catherine”       ---------------->         “yellow”

“Leah”              ----------------->             “purple”

**Hash Function:**

* A **hash function H**(x) is a function that maps a key ʻxʻ to a whole number in a fixed range

**Properties of Hash Functions:**

* If **H(x) = H(y)** then objects x and y **might be equal,** but if **H(x) does not equal H(y)** then x and y are **certainly not equal**
* A hash function **H(x)** must be **deterministic**
* This means that if **H(x) = y** then **H(x)** must always produce y and never another value. This is critical to the functionality of the hash function
* Important to try and make **uniform** hash functions to minimize the number of hash collisions
* A **hash collision** is when two objects x, y hash to the same value (**H(x) = H(y))**

**How does a hash table work?**

* To use hash functions efficiently, use them as a way to index into a hash table
* Some ways to deal with hash collisions are **separate chaining** and **open addressing**
* **Separate chaining:** deals with hash collisions by maintaining a data structure (usually a linked list) to hold all the different values which hashed to a particular value
* **Open addressing:** deals with hash collisions by finding another place within the hash table for the object to go by offsetting it from the position to which it was hashed to
* The **key-value pairs are stored in the table (array) itself** as opposed to a data structure like in separate chaining.
* **Load factor = items in table/size of table**
* If the position our key hashed to is occupied we try another position in the hash table by offsetting the current position subject to a **probing sequence P(X).** We keep doing this until an unoccupied slot is found
* Infinite amount of probing sequences you can make, here are a few:
* **Linear probing:** linear function
* **Quadratic probing: a** quadratic function
* **Double hashing:** secondary hash function
* **Pseudo random number generator: P(k,x) = x\*RNG(H(k),x)** where RNG is a random number generator function seeded with H(k)

**Chaos with cycles:**

* Techniques such as **linear probing**, **quadratic probing**, and **double hashing** are all subject to the issue of causing cycles which is why the **probing functions used with these methods are very specific**
* Open addressing is very specific to the hashing function and probing function used. This is not something you have to worry about (as much) if you are using separate chaining as a collision resolution method

**Linear probing:**

* To prevent from getting linear probing infinite cycles to keep x and N (table size) **relatively prime.** Two numbers are relatively prime if their **Greatest Common Denominator (GCD) is equal to one**. Hence, when **GCD**(N, x) = 1 the probing function **P(x)** will be able to generate a complete cycle and we will always be able to find an empty bucket
* Common choice for **P(x)** is **P(x) = 1x** since GCD(N, 1) = 1 no matter the choice of N(table size)

**Quadratic probing:**

* Most quadratic functions will create infinite cycles
* Three most popular approaches that will work:

1. Let **P(x) = x^2**, keep the table size a prime number > 3 and also keep a (alpha) <= 0.5

2. Let **P(x) = (x^2 + x)/2** and keep the table size a power of two

3. Let **P(x) = (-1^x)(x^2)** and keep the table size a prime N where N = 3 mod 4

**Double Hashing:**

* With second hashing function, it is similar to linear probing cycles
* To fix the issue of cycles pick the table size to be a prime number and also compute the value of delta
* Delta = H2(k) mod N
* If delta = 0 then we are guaranteed to be stuck in a cycle, so when this happens set delta = 1
* This allows 1 <= delta < N and GCD (delta, N) = 1 since N is prime
* **Probing method** which probes according to a constant multiple of another hash function, specifically:
* P(k,x) = x\*H2(k), where H2(k) is a second hash function
* H2(k) must hash the same type of keys as H1(k)
* Frequently the hash functions selected to compose H2(k) are picked from a pool of hash functions called **universal hash functions** which generally operate on one fundamental data type

**Hash Table(HT) Removing elements open addressing:**

-   Solution is to place a **unique marker** called a **tombstone** instead of null to indicate that a (k,v) pair has been deleted and the bucket should be skipped during a search

-   Sometimes tombstones get cluttered in your table so you can fix that when you lookup keys and remove tombstones

-   One optimization is to replace the earliest tombstone encountered with the value we did a lookup for. The next time we look up the key itʻll be found much faster! This is called **lazy deletion**

**Fenwick Tree(Binary Indexed Tree):**

-   A **Fenwick Tree** is a data structure that supports sum range queries as well as setting values in a static array and getting the value of the prefix sum up some index efficiently

-   Unlike a regular array, in a Fenwick tree a specific cell is responsible for other cells as well

-   The position of the **least significant bit (LSB)** determines the **range of responsibility** (always powers of 2)that cell has to the cells below itself

-   Starts 1 based whereas an array starts 0 based

-   Indexʻs are represented as a binary number

-  In a Fenwick tree we may compute the **prefix sum** up to a certain index, which ultimately lets us perform range sum queries

-   Suppose you want to find the prefix sum of [1, i] then you **start at i and cascade downwards** until you reach zero adding the value at each of the indices you encounter

**Point Updates:**

-   To update the cell at index i in the Fenwick tree of size N:

**function** add (i, x):

**while** i < n:

tree [i] = tree [i] + x

i = i + LSB (i)

-   Where **LSB** returns the value of the least significant bit

**Suffix Arrays:**

-   A **suffix** is a substring at the end of a string of characters. For our purpose suffixes are non-empty

-   **Suffixes** start from the end of the string and moved forward. Ex: **HORSE**

o   **E, SE, RSE, ORSE, HORSE**

-   A **suffix array** is an array which contains all the **sorted** suffixes of a string

-   The actual “**suffix array”** is the array of sorted indices. This provides a compressed representation of the sorted suffixes without actually needing to store the suffixes

-   The suffix array provides a space-efficient to a **suffix tree** which itself is a compressed version of a **trie**

**The longest common prefix (LCP) array:**

-   The LCP array is an array in which every index tracks how **many characters two sorted adjacent suffixes have in common**

-   Once you have all suffixes of a string (Suffix Array), now through the LCP array we are finding the longest common prefix (by first finding all the common prefixes)

**Finding Unique Substrings:**

-   All strings have **n(n+1)/2** substrings

-   Duplicate substrings can be calculated through using the **LCP**

-   Thus unique substrings is n(n+1)/2 – LCP value

**Balanced Binary Search Trees (BBSTs)**

-   A **Balanced Binary Search Tree (BBST)** is a **self-balancing** binary search tree. This type fo tree will adjust itself in order to maintain a low (logarithmic) height allowing for faster operations such as insertions and deletions

**Tree rotations:**

-   The secret ingredient to most BBST algorithms is the clever usage of a **tree invariant** and **tree rotations**

-   A tree invariant is a property/rule you impose on your tree that it must meet after every operation. To ensure that the invariant is always satisfied a series of tree rotations are normally applied

-   Tree rotations are used to balance a tree according to different cases where the tree might be right heavy or left heavy

**Inserting Elements into an AVL Tree:**

-   An **AVL tree** is one of many types of **Balanced Binary Search Tree (BBST)** which allow for logarithmic 0(log(n)) insertion, deletion, and search operations

-   The property which keeps an AVL tree balanced is called the **Balanced Factor (BF).** The balanced factor is right subtree height – left subtree height

-   H(x) is the height of node x. The height is calculated as the number of edges between x and the furthest leaf

-   AVL tree rotations require you to call the **update method**

**Removing Elements from an AVL Tree:**

-   Exact same to removing elements from Binary Search Tree (BST), just need to **augment it to suit AVL Tree**

**Augmenting BST Removal Algorithm for AVL Tree:**

-   Just need to add 2 lines of code

-   Update (node) and balance (node)