**Algorithms & Data Structure Notes**

**By: Sang Vo**

**1. General**

TBC!

1. Heaps
2. Backtracking
3. Priority Queues
4. Graphs
5. Recursion
6. Dynamic Programming
7. Union Sets

Tips!

1. Ask questions to clarify the requirements even if you think you know them.
2. Making certain assumptions?
3. Edge cases?
4. Are they looking for optimal solutions for time and space complexity or if they prefer you first have a working solution that is readable, logical and maintainable before you go back to refactor and optimize (premature optimisation is a waste of time).
5. Do be selective when it comes to data structures.
6. Ask for hints if stuck.
7. Brute-force solution if stuck.
8. Speaking out loud and explaining thought processes.

**Algorithm:** any well-defined computational procedure that takes some value or set of values as input and produces some value or set of values as output in a finite amount of time.

* Viewed as a tool for solving a well-specified computational problem.
* Examples:
  + Sorting.
  + Searching.
  + Public-key cryptography.
  + Human Genome Project → usage of Dynamic Programming to determine similarities between DNA sequences.
  + Finding good routes on which data travels.
  + Using a search engine to quickly find pages on which particular info resides.
  + Clustering algorithm to detect cancerous tumors.
* Important to recognise NP-complete where there is no known algorithm that runs in a reasonable amount of time.
* Set of NP-complete problems has the remarkable property that if an efficient algorithm exists for any one of them, then efficient algorithms exist for all of them.
* Several NP-complete problems are similar, but not identical, to problems for which we do know of efficient algorithms → these small changes to the problem statement can cause a big change to the efficiency of the best known algorithm.
* If you can show the problem is NP-complete, you can spend your time developing an efficient approximation algorithm (good but not necessarily the best).
* Example of an NP-complete problem is the traveling-salesperson problem.

**Online algorithms:** For many important real-world examples, the input actually arrives over time, and the algorithm must decide how to proceed without knowing what data will arrive in the future. These are known as.

**Data structure:** a way to store and organise data in order to facilitate access and modifications. No single DS works well for all purposes, thus it is important to recognise the strengths and weaknesses of each especially for algorithm design.

**Efficiency:** constant variables are less significant with larger inputs as seen in insertion sort = c1n^2 and merge sort = c2nlgn. We can see that merge sort is more advantageous as the input size grows exponentially.

**Machine learning** automates the process of algorithmic design.

* However, we can look at it as a collection of algorithms just under a different name.
* The success of ML are mainly for problems for which humans do not really understand what the right algorithm is → eg. computer vision and automatic language translation.
* For algorithmic problems that humans understand well, efficient algorithms designed to solve a specific problem are typically more successful than ML approaches.
* One example of an application that requires algorithmic content at the application level is Google Maps.
  + Shortest Path Algorithm (Dijkstra's Algorithm), used for route optimization to help users find the quickest and shortest path between two locations.
  + Clustering Algorithms (DBSCAN, K-Means), groups nearby map features such as businesses, landmarks, or traffic hotspots for better user visualizations.

**2. Characterizing Running Times**

| **Notation** | **Details** |
| --- | --- |
| Big-O   * O(f(n)) | * Characterizes an **upper bound** on the asymptotic behavior of a function. * Represents the worst-case time complexity of an algorithm. * Eg. 5n² + 3n + 2 → O(n²) |
| Omega   * Ω(f(n)) | * Characterizes a l**ower bound** on the asymptotic behavior of a function. * Represents the best-case time complexity. * Eg. Insertion sort best case is if all the values are sorted → Ω(n) |
| Theta   * Θ(f(n)) | * Characterizes a **tight bound** on the asymptotic behavior of a function. * Represents both upper and lower bounds → average case. * If an algorithm is both O(f(n)) and Ω(f(n)), then it is Θ(f(n)). * Eg. Merge sort is Θ(n log n) because it always takes O(n log n) time in all cases. * People occasionally conflate O-notation with Θ-notation by mistakenly using O-notation to indicate an asymptotically tight bound. An algorithm that runs in logarithmic time **can** be faster than one that runs in quadratic time but it can also be slower. |

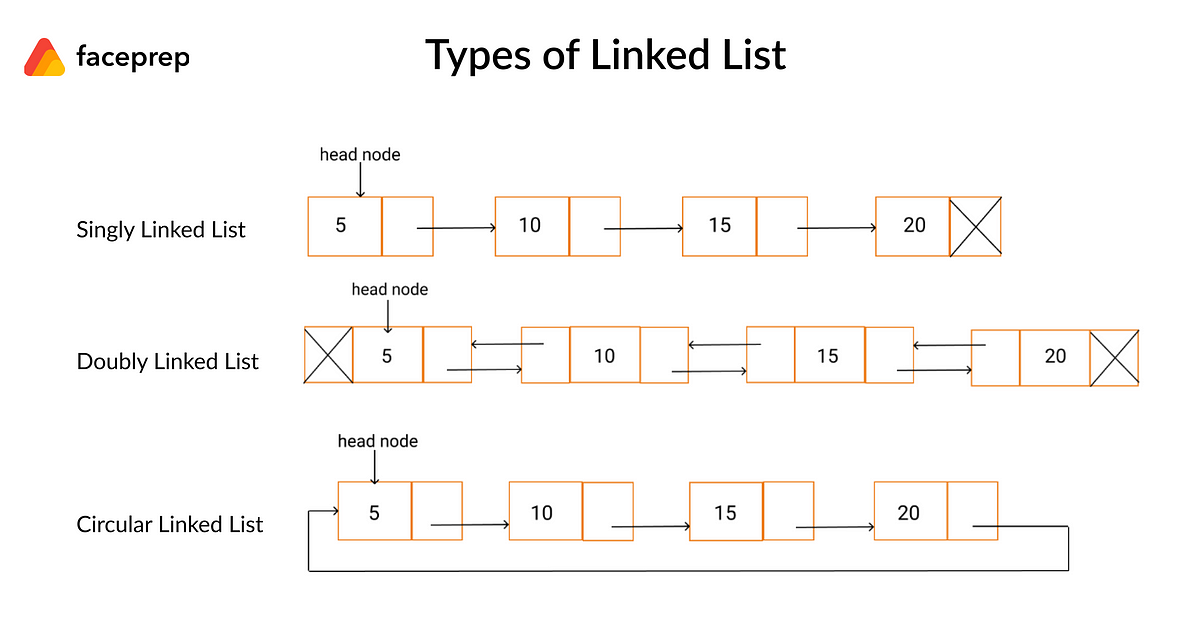
**3. Data Structures**

**Arrays**

* Stores a collection of elements, usually of the same type, in a specific order.
* Each element in an array is assigned an index, which allows you to access and manipulate elements efficiently.
* Useful commands in js include:
  + → add element to end of array
  + → remove element at end of array
  + → add element to start of array
  + → remove element at start of array
  + → quicksort on array
  + → reverse quicksort on array
  + → start is included but end **is not**
  + → start is include

**Linked List**

* Linear data structure where elements (called nodes) are connected using pointers.
* Node → contains **value** and a **pointer** to the next node on the list.
* The list starts at the **head node**.
* The **last node points to null** indicating the end of a linked list.
* Types:
  + Singly linked list → nodes point only to the next node.
  + Doubly linked list → nodes point to both the next and previous node.
  + Circular linked list → the last node links back to the head.



* Pros are efficient insertion/deletion and resizing.
* Cons are transversal for search and extra memory.

**Stacks**

* LIFO → last in first out.
* let stack = []; → initialisation of stack.
* → add an element to the top of the stack.
* → remove element at top of the stack and **PRINT OUT** removed element.
* → check if empty or size.
* \* → print out the top element of the stack.

**Queue**

* FIFO → first in first out.
* let queue = []; → initialisation of queue.
* → enqueue.
* → dequeue/remove element at the front of the queue, takes time and **PRINT OUT** removed element.
* → check if empty or size.
* \* → check the front element.

**Set**

* In js, a built-in object that stores unique values which automatically removes duplicates.
* Set objects **do not** have property but uses instead.
* look up time is .

**Maps**

* In js, built-in objects that allow you to store key-value pairs.
* Similar to objects but have some important differences and advantages:
  + Allow any type of value to be used as a key not just strings or symbols (which are the only allowed keys in plain objects).
  + Maintain order of insertion of the keys, unlike objects where the order of keys might not be guaranteed.
  + More built-in methods for easier manipulation.

.charCodeAt(i); // convert to ASCII value

let map = new Map();

map.set('name', 'John');

map.set('age', 30);

map.has('age'); // check if key exists

map.delete('age'); // remove of key-value pair

map.size; // return size

// Iterating over keys, values, and entries (key-value pairs)

for (let [key, value] of map) {

console.log(key, value);

}

// extracts only the values

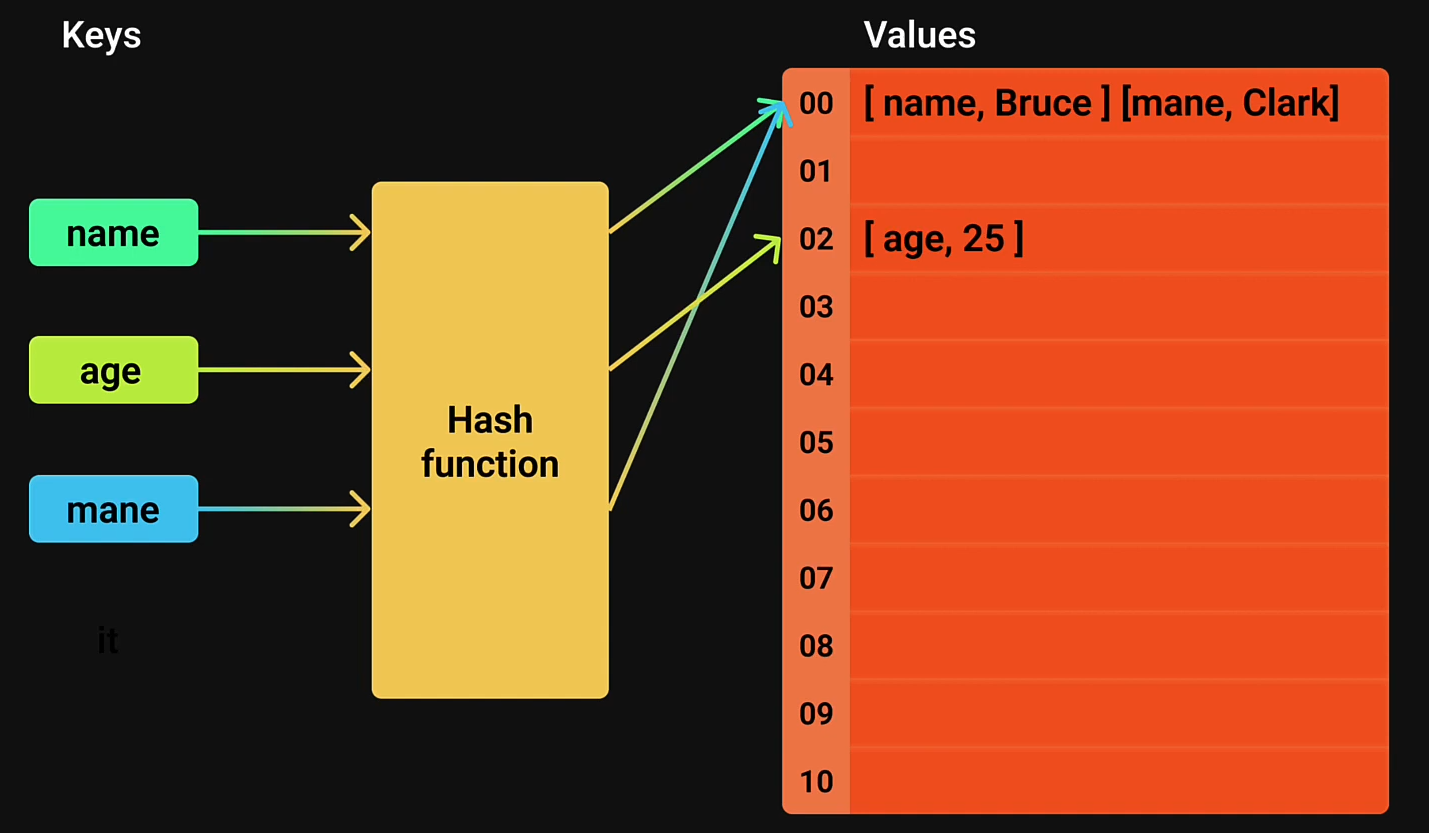
return(Object.values(map));

**Frequency Maps**

* Contains:
  + Keys → items (or characters, numbers, etc.) in your collection.
  + Values → count or frequency of each item.

**Hash Table**

* Also known as hash map, is a data structure that is used to store key-value pairs.
  + Eg. “in” → “India”, “au” → “Australia”, “fr” → “France”.
* Given a key, you can associate a value with that key for a very fast lookup.
* When constant time lookup and insertion is required → database indexing and caches.



* In js, objects behave like hash tables (or dictionaries in other languages), but they inherit properties and methods from their prototype.
  + If you use certain keys they might conflict with built-in properties. Eg. toString

**let obj = {};**

**obj["toString"] = "Hello";**

**console.log(obj.toString); // [Function: toString], not "Hello"**

* + To avoid this you can use either Object.create(null) or Maps → Map doesn’t inherit from Object.prototype, there is no risk of key conflicts.
* In js, it uses string keys but arrays work with numeric indices → use of a **hash function:** transforms a string into a numeric index that fits within the fixed array size.
  + Hash the String → convert the string into a numeric value.
  + Map the Hash to an Array Index → use **modulo** to ensure it fits within the array size.
* **Collision:** when two different keys hash to the same index in the fixed-sized array.
  + Chaining → instead of storing just one value at each index, we store multiple values using a linked list or an array.
  + Open addressing → instead of storing multiple values at one index, we find a new empty index for the colliding value.
    - Linear probing → if the spot is taken check the next slot **.**
    - Quadratic probing → jump using to reduce clustering.
    - Double hashing → use another has function for probing.
* **Bucket:** a storage container at each index of a hash table’s array → primarily used when collisions occur holding multiple key-value pairs.
* Hash Table code implementation:

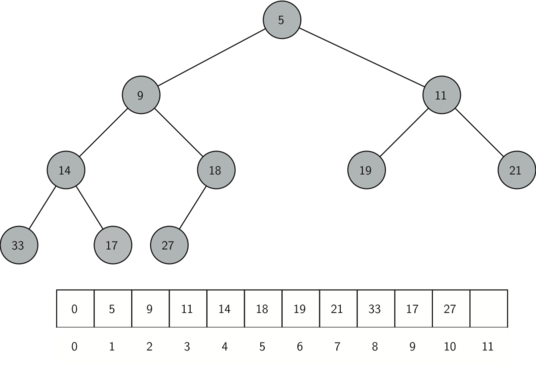
1. Constructor to allow: const table = new HashTable(int);
2. Hashing function to convert a string key to a numeric index.
3. set to store a key-value pair --> added buckets for collision cases.
4. get to retrieve a value given its key --> adjust for buckets.
5. Remove to delete a value given its key --> adjust for buckets.
6. display() results function.

**Buckets**

* Containers used to group elements based on a shared property.
* Common for sorting, hashing and frequency-based problems:
  + Faster sorting **.**
  + Hash buckets prevent collisions.
  + Find most (eg. Top K Frequent Elements) or least frequent elements efficiently.

**Heap**

* Tree-based partially ordered DS.
* Any node’s key <= parent’s key and >= child’s key.
* **Binary heap:** array object that we can view as a nearly complete binary tree that is effective for implementing a priority queue and usually implemented as an array.



* **Max-heap:** largest element is at the root and used for heapsort algorithms.
  + Max-heap property → the parent node must always be greater than or equal to its children. Ensuring the max element is always at the root and every subtree is also a max heap.
* **Min-heap:** smallest element is at the root and used to implement priority queues.
* **Binary Heap-Poll**

1. Pop the top element.
2. Move the last element to the top.
3. Heapify down: recursively compare to children, swap with bigger child.

* **Binary Heap-Push**

1. Insert element to the end.
2. Heapify up: recursively compare it to a parent,swap if parent is smaller.

* Search is → req comparing every element.
* Other operations:
  + peek → see the head of the heap
  + isEmpty
  + size
  + Merge

**Priority Queues**

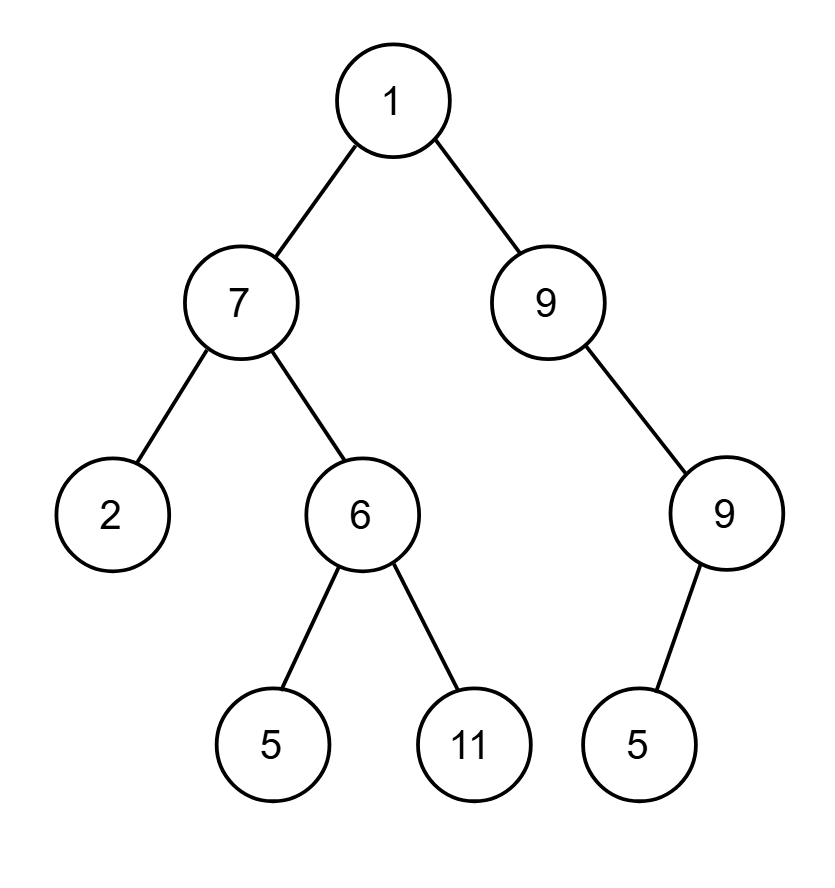
* TBC.

**Monotonic Deque**

* Elements in the deque from head to tail are in decreasing order eg. [7, 3, 2, 0].
* We use a deque (double-ended queue) to keep track of indices, not values.

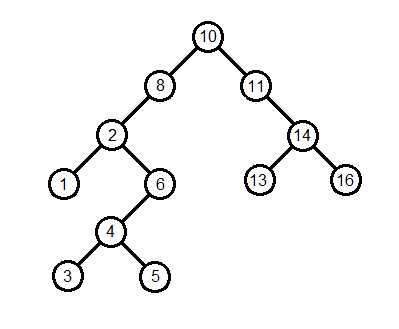
**Binary Tree**

* Non-linear data structure where each node has at most two children (left & right child).
* Top-most node = root.
* Bottom-most nodes = leaves.



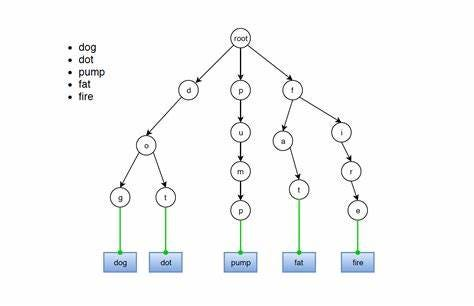
**Binary Search Tree**

* Left child < parent < right child



**Trie (Prefix-Tree)**

* Tree-like data structure used for storing strings in a way that makes prefix-based operations efficient.
* Each node represents a single character, and words are built character by character as you traverse down the tree.



* You **SHOULD NOT** use for:
  + Don’t require fast prefix search.
  + Space efficiency → Tries can use a lot of memory compared to hash tables.
  + Only need exact word lookups → A hash set (Set in JavaScript) is simpler and more space-efficient.
* Use cases:
  + Search bars and autocompletion.
  + Autocorrection (eg. grammarly).
  + Games (Boggle, Scrabble, Crossword Solvers).
  + Data compression.
  + DNA sequence matching & bioinformatics.
* Representation approaches:
  + Maps.
  + Arrays → for key to represent it childrents requires a fixed array = alphabet size.
* Operations:
  + insert
  + contains
  + remove
  + Optional → getAllWithPrefix and startsWith

**4. Sorting Algorithms**

Factors to consider:

* Input size.
* Runtime.
* Taking the worst-case running time.

| Sorting Algorithm | Best Case  Ω(f(n)) | Average Case  Θ(f(n)) | Worst Case  O(f(n)) | Space | Stable | Example |
| --- | --- | --- | --- | --- | --- | --- |
| Bubble |  |  |  |  | ✅ | Organising books on a shelf based on height. |
| Selection |  |  |  |  | ❌ | Organising books on a shelf based on height. |
| Insertion |  |  |  |  | ✅ | Reorganising a hand of playing cards. |
| Merge |  |  |  |  | ✅ | Divide-and-conquer method. |
| Quick |  |  |  |  | ❌ | Database indexing. |
| Heap |  |  |  |  | ❌ | Organising a priority queue → organise passengers boarding by priority. Used in Dijkstra’s Algorithm and task scheduling. |

**Stability:** Refers to the relative order of equal elements in the sorted output as they appeared in the input.

* Eg. Have a list of names and ages: Mariel = 27, Sang = 26, Alice = 27, Michael = 28.
* Stable → Sang = 26, Mariel = 27, Alice = 27, Michael = 28.
* Unstable → Sang = 26, Alice = 27, Mariel = 27, Michael = 28.

**Recursion:** is a programming technique where a function calls itself to solve a problem by breaking it down into smaller subproblems. Each recursive call moves closer to a **base case**, which stops further recursion.

**Divide and Conquer**

* Powerful strategy for designing asymptotically efficient algorithms.
* **Base case:** condition that stops the recursion from continuing indefinitely, it defines the simplest possible input for which the solution is already known, preventing infinite loops.
* **Recursive case:** 
  + **Divide** the problem into one or more subproblems that are smaller instances of the same problem.
  + **Conquer** the subproblem by solving them recursively.
  + **Combine** the subproblem solutions to form a solution to the original problem.
* **Recurrence** is an equation that expresses a problem in terms of smaller instances of itself. Commonly seen in recursive algorithms and is used to describe their time complexity.
* **Algorithmic recurrence:** recurrence is stated without an explicit base case, we assume that the recurrence is algorithmic → when n is smaller enough it will terminate (we do not know what n is in this case).

**Swap Methods**

* **Traditional swap:**

let temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = temp;

* **Destructuring swap:**

[arr[i], arr[minIndex]] = [arr[minIndex], arr[i]];

**Merge Sort**

* Recursive procedure that closely follows the divide-and-conquer method → it sorts a subarray starting with the entire array and recursion down to smaller and smaller subarrays.
* Array.prototype.slice() → method of array instances returns a shallow copy of a portion of an array into a new array object selected from start to end (end not included) where they represent the index of items in the array. The original array will not be modified.
  + const leftArr = arr.slice(0, mid);
  + const rightArr = arr.slice(mid);
* Array.prototype.push() & Array.prototype.shift() → push element into the array and shift removes the first element from the array and returns that element.
  + sortedArr.push(rightArr.shift());

**Heapsort**

* Like insertion sort, but unlike merge sort, heapsort sorts in place hence only a constant number of array elements outside the input array is stored at any time.
* We have two functions:
  + **heapify(arr, length, parentIndex):** used to make the array a max-heap (having a parent node must always be greater than or equal to its children. Ensuring the max element is always at the root and every subtree is also a max heap.
  + **heapSort(arr):** comprises two steps where we first turn the unsorted array → max-heap by calling upon the function previously.
    - **let parentIndex = Math.floor(length / 2 - 1);**
    - **for (let i = parentIndex; i >= 0; i--) {**

**heapify(arr, length, i);**

**}**

* + Second step once max-heap is achieved is to swap the root value with the last node then decrement the heap and have this process done recursively until we hit one value left.
    - **for (let i = length - 1; i > 0; i--) {**

**[arr[0], arr[i]] = [arr[i], arr[0]];**

**// heapify the reduced heap**

**heapify(arr, i, 0);**

**}**

**return arr;**

**}**

**Quicksort**

* Identify the pivot element in the array:
  + First element;
  + Last element;
  + Random element;
  + Median;
* Put everything that is smaller than the pivot into a ‘left’ array and everything that is greater into a ‘right’ array.
* Repeat the process for the individual ‘left’ and ‘right’ arrays till you have an array of length 1 which is sorted by definition.
* Repeatedly concatenate the left array, pivot and right array till one sorted array remains.

**5. Searching Algorithms**

**Binary Search**

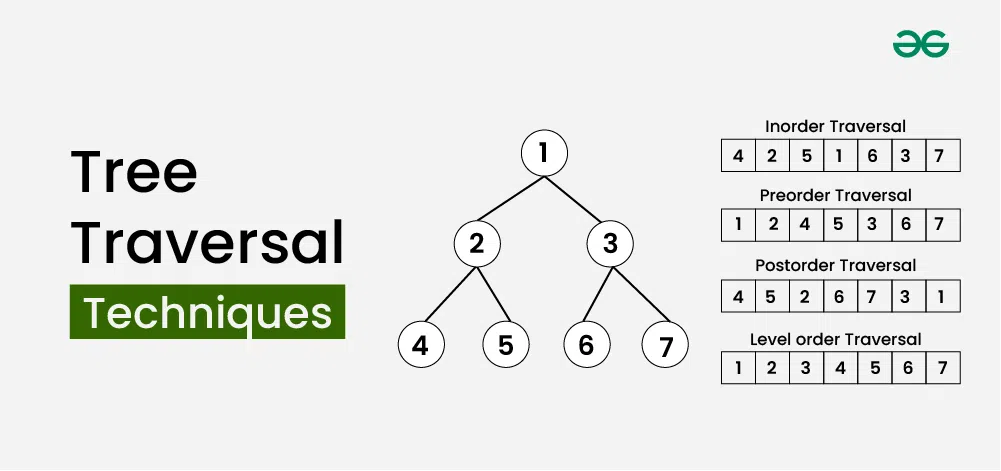
* Binary search is an algorithm with time complexity used to find the position of a target value within a **sorted array** by dividing the search interval repeatedly in half.
* Implementation via:
  + Iterative approach with a while loop.
    - Space complexity = .
    - .
  + Recursive function.
    - Space complexity = .
    - .

**Depth First Search (DFS)**

* Traversal strategy where you explore one path as deep as possible before backtracking.
* Visit the root node, visit all the does in the left subtree and visit all the nodes in the right subtree.
* JavaScript supports lexical scoping (inner functions can access variables from their parent function).
  + This is useful for efficiency as we don’t need to pass additional parameters.
  + Eg. Even though n is not passed in recursive calls, backtrack can access n from its outer scope (generateParenthesis).
* Types of DFS traversals:
  + Preorder → Root at the start.
    - Read the data of the node.
    - Visit the left subtree.
    - Visit the right subtree.
  + Inorder → Root in the middle.
    - Visit the left subtree.
    - Read the data of the node.
    - Visit the right subtree.
  + Postorder → Root at the end
    - Visit the left subtree.
    - Visit the right subtree.
    - Read the data of the node.

**Breadth First Search (BFS)**

* Explore all nodes at the present depth prior to moving on to the nodes at the next depth level → i.e. Level Order Traversal.
* Steps to create BFS:
  + Create a queue.
  + Enqueue the root node.
  + As long as a node exists in the queue:
    - Dequeue the node from the front.
    - Read the node’s value.
    - Enqueue the node’s left child if it exists.
    - Enqueue the node’s right child if it exists.



**6. Techniques**

**Two Pointers**

* Used when we know that the elements are sorted by assigned left and right pointers and incrementing them and decrementing them accordingly to achieve more efficient time complexities.

**Sliding Window**

* Technique used to optimise loops that involve checking **subarrays** of a larger dataset.
* Reusing previous calculations to reduce time complexity.
* Useful for:
  + Maximum/minimum subarray sum.
  + Longest substring with certain conditions.
  + Counting elements in a window.

**Kadane’s Algorithm**

* Used to find the maximum sum of a contiguous subarray in an array of intergers (containing **negative** values).
* You loop through the array and at each step either decide:
  + Should I continue the current subarray?
  + Should I start fresh from this element?

**Floyd’s Tortoise and Hare (Cycle Detection)**

* Used for cycle detection in linked lists via two pointers as they will converge eventually if there is a cycle.
* Once a cycle is detected, we can also check for duplicates by:
  + Resetting one pointer (i.e. slow or fast) to start from the beginning of the array/linked list.
  + The fast pointer should also be at 1x speed and index 0.

**7. Misc**

* (strict equality) when comparing both value and type.

console.log(5 === 5); // true (same value & type)

* == (loose equality) compares values after type conversion.

console.log(5 == "5"); // true (string "5" is converted to number 5)

* ~~ → cuts all fractional digits, similar to Math.floor() for positive numbers **NOT** negative numbers.
* Useful commands **strings** commands:
  + → method tests for matches in a string that returns true or false.
  + Replacing non-alphanumeric values:

**s.replace(/[^a-zA-Z0-9]/g, '').toLowerCase();**

* Useful **Math.** commands:
  + → truncates the decimal → 0
* **Constructor** is a method that runs after we instantiate an object from that class.
* **this** is required in class methods to access and modify instance properties → i.e. **this** refers to the instance of the object being created.
  + If not, a ReferenceError will occur because items and count are not globally defined → they belong to the instance of the class, not the function scope.
  + Using this.e ensures we're referring to the current instance’s data, not an undefined or external variable.
* This works because JavaScript supports lexical scoping (inner functions can access variables from their parent function).