**Searching**

**Linear Search**

* Time Complexity – O(n)
* Also known as Sequential Search.
* Method of finding a target value within a list.
* Sequentially checks each item in the list until a match is found or until all elements have been searched.

**Linear Search Methods**

var beasts = ['Centaur', 'Godzilla', 'Mosura', 'Minotaur', 'Hydra', 'Nessie'];

beasts.indexOf('Godzilla');

beasts.findIndex(function (item) {

    return item === 'Godzilla';

});

beasts.find(function (item) {

    return item === 'Godzilla';

})

beasts.includes('Godzilla')

**Binary Search**

* Split the list in half and see whether the item you are searching for is on the left or right side of the list. Then go the side with the item we are looking for and split that list in half. Repeat this until the item is located.
* Time Complexity – O(log n)
* Better than Linear Search
* The list is sorted already with Binary Search.
* Divide and conquer. We can discard half the items rather than one at a time by storing data in a tree data structure.
* Data is sorted as they are inserted which makes the tree sorted allowing us to use divide and conquer algorithms to search for items which is faster than an array which we need to use linear search which takes a lot longer.

**Graph & Tree Traversals**

* Breadth First Search (BFS)
* Depth First Search (DFS)
* O(n) because we must visit every node using BFS and DFS
* O(log n) searching using tree and graphs vs O(n) using lists and arrays

**Breadth First Search (BFS)**

* Start at the root node
* Move left to right on each level
* Uses additional memory because it tracks every node and its children on each level
* //              9
* //      4               20
* //   1     6       15       170
* BFS[9, 4, 20, 1, 6, 15, 170]

A picture containing clipart

Description automatically generated

Here we would have [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

**Depth First Search (DFS)**

* Go deep first starting from the left
* Lower memory requirement than BFS because it’s not necessary to store every child pointer at each level
* Implemented 3 ways

1. Inorder
2. Preorder
3. Postorder

//              9

//      4               20

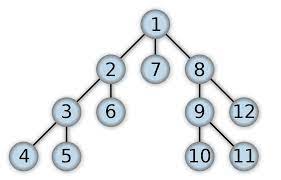
//   1     6       15       170

DFS[9, 4, 1, 6, 20, 15, 170]

InOrder - [1, 4, 6, 9, 15, 20, 170]

PreOrder - [9, 4, 1, 6, 20, 15, 170]    // very useful in recreating a tree because it is ordered

PostOrder - [1, 6, 4, 15, 170, 20, 9]



BFS vs DFS

**BFS**

* Time complexity is the same O(n) because we are traversing(visiting) every node.
* BFS has the shortest path between the starting point and any other reachable node because we start with the root node search the closest nodes first.
* BFS uses more memory because it tracks every node and its children at each level

**DFS**

* Time complexity is the same O(n) because we are traversing(visiting) every node.
* DFS is better is you know a node is at a lower level of the tree
* Uses less memory than BFS because it’s not necessary to store every child pointer at each level
* Can determine if a path exist from a source node to a target node
* Can get slow especially if the tree or graph is deep (many levels)
* Not good at finding the shortest path unlike BFS

**BFS vs DFS Interview Questions**

//If you know a solution is not far from the root of the tree:

BFS - it starts searching the closest node to the parent first

//If the tree is very deep and solutions are rare,

BFS

DFS - we use recursion in DFS which will take long time because the tree is very deep and the solutions are rare

//If the tree is very wide: - meaning there are a ton of nodes underneath each parent

DFS

BFS will need too much memory because it needs to keep track of the nodes at each level

//If solutions are frequent but located deep in the tree

DFS

//determining whether a path exists between two nodes

DFS

//Finding the shortest path

BFS