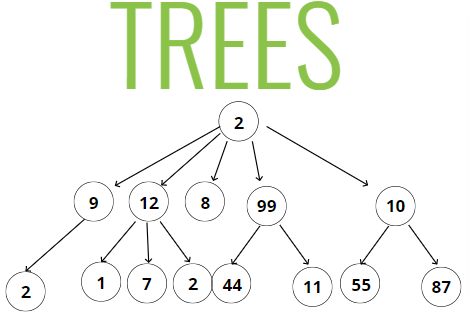
**Binary Search Trees (TREE Data Structure)**

**WHAT IS A TREE?**

A data structure that consists of nodes in a parent / child relationship.



**NOTE:**

* A node in a tree can have multiple references. For Ex: 2 node have reference of 9th, 12th, 8th, 99th & 10th Node.
* We can store any data in nodes of a tree.
* 9, 12, 8, 99& 10Node are children relative to its parent node 2.

**Difference between Linked Lists & Trees:-**

**Linked Lists:**

* Lists are Linear.
* There’ s one path

**Trees:**

* Trees are non-linear. They can have branch.
* They can have more than One pathway through a tree.

**NOTE:**

We can think of, Singly Linked List as a very special case of tree.

**Rules of a Tree:**

* There Should only one root in a Tree.
* Every node should move away from the root.
* Every node should reference down to their child node.

**TREE TERMINOLOGY**

* **Root** - The top node in a tree.
* **Child** -A node directly connected to another node when moving away from the Root.
* **Parent** - The converse notion of a child.
* **Siblings** -A group of nodes with the same parent.
* **Leaf** - A node with no children.
* **Edge** - The connection between one node and another.

**Application of Trees**

* HTML DOM
* Network Routing
* Abstract Syntax Tree
* Artificial Intelligence  
  Ex: Minimax Tree (Application: Tic Tac Toe Game). We could basically break down the logic of the game into a tree.
* Folders in Operating Systems
* Computer File Systems

**Kinds of Trees**

There’re tremendous kinds of trees across the world.

Here’s some family of Trees. Ofcourse, there are lots of further trees comes under these families as well.

* Trees
* Heaps
* Binary Tree
* Binary Search Tree (Special Case/Type of Binary Tree, which is a special type of tree)

**Conditions of Binary Trees**

* Each node can have at most 2 children.

**Binary Search Trees (BSTs):**

* BSTs are used to store data that can be compared that is sortable so often classically.
* Each parent/root node can have at most **2 children**.
* Every node to the left of a parent/root node is always **less than** the parent/root. (Everything to the left is less than the parent/root)
* Every node to the right of a parent/root node is always **greater than** the parent/root. (Everything to the right is greater than the parent/root)

Note:

* Binary Tree could be unsorted means Node Greater than parent/root might be in the left side or Node Less than parent/root might be in the right side of the tree. This is called plain Binary Tree but not Binary Search Tree.
* But Binary Search Tree is always be sorted. So that’s why, BST works for sorted data or sortable data, where there’s an order. It’s not possible to use on unsorted data.

**Simple Binary Search Tree Implementation:-**

**Information:**

* Every node in the Right Side of BST should be Greater than the Root
* Every node in the Left Side of BST should be Less than the Root.
* Every Node in the right of Parent Node should be Greater than its Parent.
* Every Node in the left of Parent Node should be Smaller than its Parent.

**Approach:**

* Every node should have value and connecting with left & Right Node.
* Create a Node (Class) with property initializes with left = null & right = null.
* Create a Root and initializes with null.

10

/ \

7 15

\

9

**Code Implementation:**

class Node{

constructor(value){

this.value = value;

this.left = null;

this.right = null;

}

}

class BinarySearchTree{

constructor(){

this.root = null;

}

}

const tree = new BinarySearchTree();

tree.root = new Node(10);  
tree.root.right = new Node(15);  
tree.root.left = new Node(7);  
tree.root.left.right = new Node(9);

tree ***//To execute BST***

**Output:**BinarySearchTree {root: Node}  
root: Node  
left: Node {value: 7, left: null, right: Node}  
right: Node {value: 15, left: null, right: null}  
value: 10

**Inserting in Binary Search Tree:-**

**Visualization:**

10

/ \

7 13

/ \ / \

5 9 11 15

insert(12):

10

/ \

7 13

/ \ / \

5 9 11 15

\

12

**Approach:**

* Create a newNode
* if there is no root
* Assign newNode to the root.
* If there is Root
* Create a current Pointer assign it on to the root node, that will traverse over the tree.
* Start a Loop
* If (required value is < than value of the Node where current Pointer is pointing on it)

-- Check If(their is no Left Node on its left Side)

-- Assign left Node of current Pointer to the newNode

-- If(Left Node is there)

-- Move/Set current pointer to that node.

-- And Keep looping till not find the leaf node

* If (required value is > than value of the Node where current Pointer is pointing on it)

-- Check If(their is no Right Node on its Right Side)

-- Assign Right Node of current Pointer to the newNode

-- If(Right Node is there)

-- Move/Set current pointer to that right node.

-- And Keep looping till not find the leaf node

**Pseudo Code:**

* Create a new node
* Starting at the root
* Check if there is a root, if not - the root now becomes that new node!
* If there is a root, check if the value of the new node is greater than or less than the value of the root
* If it is greater
* Check to see if there is a node to the right
* If there is, move to that node and repeat these steps
* If there is not, add that node as the right property
* If it is less
* Check to see if there is a node to the left
* If there is, move to that node and repeat these steps
* If there is not, add that node as the left property

**Code Implementation:**

class Node{

constructor(value){

this.value = value;

this.left = null;

this.right = null;

}

}

class BinarySearchTree{

constructor(){

this.root = null;

}

insert(value){

const newNode = new Node(value);

if(!this.root){

this.root = newNode;

return this;

}else{

let current = this.root;

while(true){

if(value<current.value){

if(!current.left){

current.left = newNode;

return this;

}else{

current = current.left;

}

}

if(value>current.value){

if(!current.right){

current.right = newNode;

return this;

}else{

current = current.right;

}

}

}

}

}

}

const tree = new BinarySearchTree();

tree.insert(10);  
tree.insert(7);  
tree.insert(5);  
tree.insert(9);  
tree.insert(13);  
tree.insert(11);  
tree.insert(15);  
tree.insert(12); **//To insert Node**

**Output:**BinarySearchTree {root: Node}  
root: Node  
left: Node {value: 7, left: Node, right: Node}  
right: Node {value: 13, left: Node, right: Node}  
value: 10

**Finding a Node in a BST**

**Approach:**

* Find Methond will accept the value of the node
* if there is no root
* return "No Root"
* Create a current Pointer assign it on to the root node.
* If(value === current.value) return current;
* Start a Loop
* If (required value is < than value of the Node where current Pointer is pointing on it)

-- Check If(their is no Left Node on its left Side)

-- return undefined;

-- If(Left Node is there)

-- Move/Set current pointer to that node.

-- IF(value === current.value) return current;

* If (required value is > than value of the Node where current Pointer is pointing on it)

-- Check If(their is no Right Node on its Right Side)

-- return undefined;

-- If(Right Node is there)

-- Move/Set current pointer to that right node.

-- IF(value === current.value) return current node;

**Pseudo Code:**

* Starting at the root
* Check if there is a root, if not - we're done searching!
* If there is a root, check if the value of the new node is the value we are looking for. If we found it, we're done!
* If not, check to see if the value is greater than or less than the value of the root
* If it is greater
* Check to see if there is a node to the right

-- If there is, move to that node and repeat these steps

-- If there is not, we're done searching!

- If it is less

* Check to see if there is a node to the left

-- If there is, move to that node and repeat these steps

-- If there is not, we're done searching!

**Code Implementation:**

class Node{

constructor(value){

this.value = value;

this.left = null;

this.right = null;

}

}

class BinarySearchTree{

constructor(){

this.root = null;

}

insert(value){

const newNode = new Node(value);

if(!this.root){

this.root = newNode;

return this;

}else{

let current = this.root;

while(true){

if(value<current.value){

if(!current.left){

current.left = newNode;

return this;

}else{

current = current.left;

}

}

if(value>current.value){

if(!current.right){

current.right = newNode;

return this;

}else{

current = current.right;

}

}

}

}

}

find(value){

if(!this.root) return "No Root";

let current = this.root;

if(value === current.value) return current;

while(true){

**//Left Side**

if(value<current.value){

if(!current.left) return undefined;

else{

current = current.left;

if(value === current.value)

return current;

}

}

**//Right Side**

if(value>current.value){

if(!current.right) return undefined;

else{

current = current.right;

if(value === current.value)

return current;

}

}

}

}

}

const tree = new BinarySearchTree();

tree.insert(10);  
tree.insert(7);  
tree.insert(5);  
tree.insert(9);  
tree.insert(13);  
tree.insert(11);  
tree.insert(15);  
tree.insert(12); **//To insert Node**

tree.find(11); **//To Find Node 11**

**Output:**Node {value: 11, left: null, right: Node}  
left: null  
right: Node {value: 12, left: null, right: null}  
value: 11

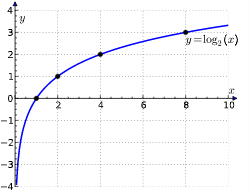
**Big O of Binary Search Tree (BST):**

Insertion - O(log n)

Searching - O(log n)

**Note**:

Double the number of nodes, we only increase the number of steps to insert/find by 1.



2x number of nodes: 1 extra step

4x number of nodes: 2 extra steps

8x number of nodes: 3 extra steps

**TREE TRAVERSAL**

This is the idea of basically given any tree, whether it’s a binary search tree or just a plain binary tree that’s unsorted or a ternary tree or some other tree that has any number of nodes. There’s no special order, just any tree.

This is going to be the universal way of traversing over any tree.

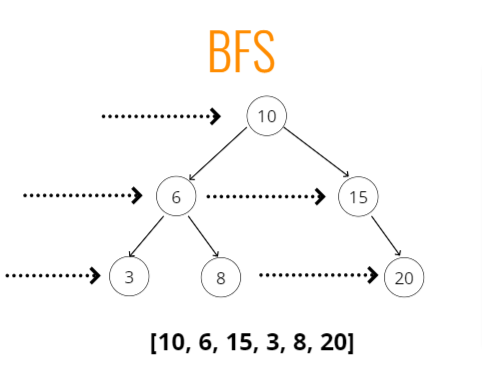
There are **2 ways** to TRAVERSING A TREE:

1. **Breadth-first Search (BFS)**
2. **Depth-first Search (DFS)**

So, these refers to the General Direction.

1. **Breadth-first Search (BFS)**

We’ve to traverse on the tree by looking every node on the same level.

****

**Approach:**

* Crate a Node Pointer pointing to Root Node
* Declare a 'data' & 'queue' variable with an empty array
* push Root Node to the queue
* Start a Loop till queue is not empty
* shift the node/element from the queue and store it in a variable called 'dequeued'. (Note:, Queue follow FIFO principle).
* push the value of the dequeued node in data array.
* IF(dequeuedNode has left Node) push it into queue
* IF(dequeuedNode has right Node) push it into queue
* return data Array. (This data will be the value of the node in Breadth-First Search order of the Tree).

**Pseudo Code:**

* Create a queue (this can be an array) and a variable to store the values of nodes visited
* Place the root node in the queue
* Loop as long as there is anything in the queue
* Dequeue a node from the queue and push the value of the node into the variable that stores the nodes
* If there is a left property on the node dequeued - add it to the queue
* If there is a right property on the node dequeued - add it to the queue
* Return the variable that stores the values

**BFS Code:**

class Node{

constructor(value){

this.value = value;

this.left = null;

this.right = null;

}

}

class BinarySearchTree{

constructor(){

this.root = null;

}

insert(value){

const newNode = new Node(value);

if(!this.root){

this.root = newNode;

return this;

}else{

let current = this.root;

while(true){

if(value === current.value) return undefined;

if(value<current.value){

if(!current.left){

current.left = newNode;

return this;

}else{

current = current.left;

}

}

if(value>current.value){

if(!current.right){

current.right = newNode;

return this;

}else{

current = current.right;

}

}

}

}

}

BFS(){

let node = this.root,

data = [],

queue = [];

queue.push(node);

while(queue.length){

let dequeued = queue.shift();

data.push(dequeued.value);

if(dequeued.left) queue.push(dequeued.left);

if(dequeued.right) queue.push(dequeued.right);

}

return data;

}

}

const tree = new BinarySearchTree();

tree.insert(10);

tree.insert(6);

tree.insert(3);

tree.insert(8);

tree.insert(15);

tree.insert(20);

tree.BFS(); //***For executing Breadth-First Search***

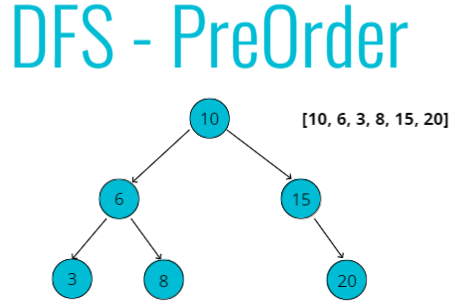
**Output:**

[10, 6, 15, 3, 8, 20]

1. **Depth-first Search**

There’re basically 3 main orders for DFS. Meaning that they pursue all nodes, they visit or traverse nodes vertically down to the end of the tree before visiting sibling nodes. We’re going to traverse from Down First or say Depth First on the Tree.

* DFS – **Pre Order**



**Visualization Implementation available in Code.**

**Approach:**

* We're going to perform DFS-Pre Order using Recursion. This is Pre-Order, We visit the node first, then left, then right.Declare a 'data' variable with an empty array.
* Declare a 'data' variable with an empty array.
* Create a Helper Function called 'Traverse'. Accept each nodes as Argument and Passing each node as Node Paramenter. Starting with Root Node.
* push value of each node into 'data' array.
* Check if(passing node has left node) recall the traverse Function with Argument (node.left Node) .
* Check if(passing node has right node) recall the traverse Function with Argument(node.right Node).
* return data Array. (This will contain the value of the node in Depth-First Search in (In-order) treaversal of the Tree).

**Pseudo Code:**

* Create a variable to store the values of nodes visited
* Store the root of the BST in a variable called current
* Write a helper function which accepts a node
* Push the value of the node to the variable that stores the values
* If the node has a left property, call the helper function with the left property on the node
* If the node has a right property, call the helper function with the right property on the node
* Invoke the helper function with the current variable
* Return the array of values

**DFS (Pre-Order) Code:**

class Node{

constructor(value){

this.value = value;

this.left = null;

this.right = null;

}

}

class BinarySearchTree{

constructor(){

this.root = null;

}

insert(value){

const newNode = new Node(value);

if(!this.root){

this.root = newNode;

return this;

}else{

let current = this.root;

while(true){

if(value === current.value) return undefined;

if(value<current.value){

if(!current.left){

current.left = newNode;

return this;

}else{

current = current.left;

}

}

if(value>current.value){

if(!current.right){

current.right = newNode;

return this;

}else{

current = current.right;

}

}

}

}

}

DfsPreOrder(){

let data = [];

function traverse(node) {

data.push(node.value);

if(node.left) traverse(node.left);

if(node.right) traverse(node.right);

}

traverse(this.root);

return data;

}

}

const tree = new BinarySearchTree();

tree.insert(10);

tree.insert(6);

tree.insert(3);

tree.insert(8);

tree.insert(15);

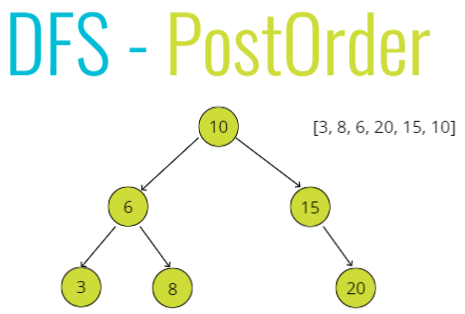
tree.insert(20);

tree.DfsPreOrder(); ***//DfsPreOrder Execution***

**Output:**[10, 6, 3, 8, 15, 20]

* DFS – **Post Order**

In post order, the route is the last thing that’s visited. For any node, we explore all children before we actually visit the node.

  
**Visualization Implementation available in Code.**

**Approach:**

* We're going to perform DFS-Post Order using Recursion. This is Pre-Order, We visit the LeafNode first, then Parent, then root.
* Post-Order Traversal is just arrangement of single line of code of Pre-Order Traversal.
* Declare a 'data' variable with an empty array
* Create a Helper Function called 'Traverse'. Accept each nodes as Argument and Passing each node as Node Paramenter. Starting with Root Node.
* Check if(passing node has left node) recall the traverse Function with Argument (node.left Node) .
* Check if(passing node has right node) recall the traverse Function with Argument(node.right Node).
* push value of each node into 'data' array.
* return data Array. (This will contain the value of the node in Depth-First Search in (In-order) treaversal of the Tree).

**Pseudo Code:**

* Create a variable to store the values of nodes visited
* Store the root of the BST in a variable called current
* Write a helper function which accepts a node
* If the node has a left property, call the helper function with the left property on the node
* If the node has a right property, call the helper function with the right property on the node
* Push the value of the node to the variable that stores the values
* Invoke the helper function with the current variable
* Return the array of values

**DFS (Post-Order) Code:**

class Node{

constructor(value){

this.value = value;

this.left = null;

this.right = null;

}

}

class BinarySearchTree{

constructor(){

this.root = null;

}

insert(value){

const newNode = new Node(value);

if(!this.root){

this.root = newNode;

return this;

}else{

let current = this.root;

while(true){

if(value === current.value) return undefined;

if(value<current.value){

if(!current.left){

current.left = newNode;

return this;

}else{

current = current.left;

}

}

if(value>current.value){

if(!current.right){

current.right = newNode;

return this;

}else{

current = current.right;

}

}

}

}

}

DfsPostOrder(){

let data = [];

function traverse(node) {

if(node.left) traverse(node.left);

if(node.right) traverse(node.right);

data.push(node.value);

}

traverse(this.root);

return data;

}

}

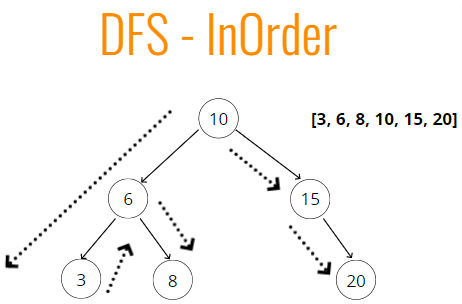
const tree = new BinarySearchTree();

tree.insert(10);  
tree.insert(6);  
tree.insert(3);  
tree.insert(8);  
tree.insert(15);  
tree.insert(20);

tree.DfsPostOrder(); ***//DFSPostOrder Execution***

**Output:**[3, 8, 6, 20, 15, 10]

* DFS – **In Order**

  
**Visualization Implementation available in Code.**

**Approach:**

* We're going to perform DFS-In Order using Recursion. This is In-Order, We visit the LeafNode first, then Parent, then child.
* In-Order Traversal is just arrangement of single line of code of Post-Order Traversal.
* Declare a 'data' variable with an empty array
* Create a Helper Function called 'Traverse'. Accept each nodes as Argument and Passing each node as Node Paramenter. Starting with Root Node.
* Check if(passing node has left node) recall the traverse Function with Argument (node.left Node) .
* push value of each node into 'data' array.
* Check if(passing node has right node) recall the traverse Function with Argument(node.right Node).
* return data Array. (This will contain the value of the node in Depth-First Search in (In-order) treaversal of the Tree).

**Pseudo Code:**

* Create a variable to store the values of nodes visited
* Store the root of the BST in a variable called current
* Write a helper function which accepts a node
* If the node has a left property, call the helper function with the left property on the node
* Push the value of the node to the variable that stores the values
* If the node has a right property, call the helper function with the right property on the node
* Invoke the helper function with the current variable
* Return the array of values

**DFS (In-Order) Code:**

class Node{

constructor(value){

this.value = value;

this.left = null;

this.right = null;

}

}

class BinarySearchTree{

constructor(){

this.root = null;

}

insert(value){

const newNode = new Node(value);

if(!this.root){

this.root = newNode;

return this;

}else{

let current = this.root;

while(true){

if(value === current.value) return undefined;

if(value<current.value){

if(!current.left){

current.left = newNode;

return this;

}else{

current = current.left;

}

}

if(value>current.value){

if(!current.right){

current.right = newNode;

return this;

}else{

current = current.right;

}

}

}

}

}

DfsInOrder(){

let data = [];

function traverse(node) {

if(node.left) traverse(node.left);

data.push(node.value);

if(node.right) traverse(node.right);

}

traverse(this.root);

return data;

}

}

const tree = new BinarySearchTree();

tree.insert(10);

tree.insert(6);

tree.insert(3);

tree.insert(8);

tree.insert(15);

tree.insert(20);

tree.DfsInOrder(); ***//DFSInOrder Execution***

**Output:**

[3, 6, 8, 10, 15, 20]

**BFS Vs DFS**(Which is Better?)

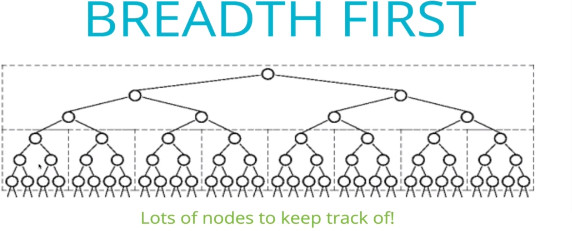
Let’s first only talk about Breadth-First Search Vs Depth First Search (Not its type).

Which is better?  
It’s totally depend upon the Tree.

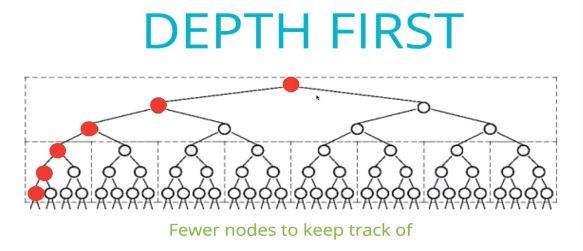
**Note:**

* Here if we’ve have only care about space Complexity Because the time complexity of BFS & DFS both has same. We’re visiting every node one time, so that’s irrelevant. It doesn’t matter if we have 100 nodes, we’re going to visit every single one breadth-first and depth-first same amount of time. Space though it really depends upon the structure of the tree.
* If it’s a really wide tree, breadth first could store or take up a lot more space for that queue.
* If it’s a really deep long tree than depth first could end up taking up more space.

1st Scenario:



If you’ve a tree that is fully fleshed out like this tree, it’s as wide as it can be. Since, in BFS we’re using queue, so the space that’s used up in this scenario could be a lot more.

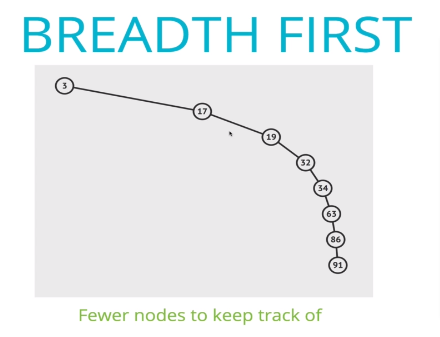


But in DFS we’re not storing all of those nodes across the tree. We only have to keep track of the nodes in a given branch a the way down to the end.

Here, in DFS we’re using recursion that gonna store recursive function in our callStack.

So in this sort of tree where it’s a lot wider than it is deep, then depth first search would use less space.

2nd Scenario:

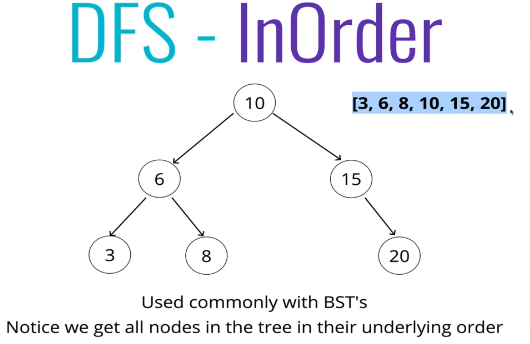


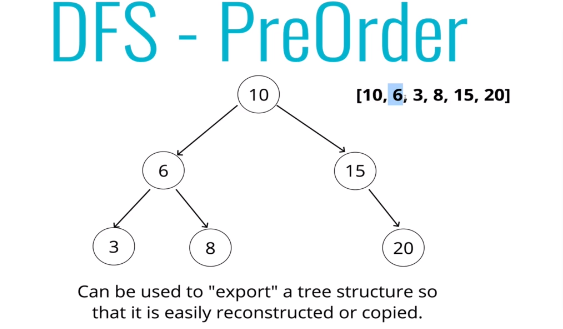
This is somewhat more depressing binary search tree, if we traverse this entire thing breadth first, the space that we take up in memory is basically nothing. Our queue never gonna have more than 1 item in it.

But if we’re traversing it Depth-First, if this tree is really, really long, then tat means a ton of levels that we have to go down and keep every other level above it in memory.

Now let’s talk about, the potential use cases of different variants of Depth First Search.  
In-Order, Pre-Order & Post-Order all are the variant of DFS.

So, why would you potentially use one over the other?





**RECAP**

* Trees are non-linear data structures that contain a root and child nodes
* Binary Trees can have values of any type, but at most two children for each parent
* Binary Search Trees are a more specific version of binary trees where every node to the left of a parent is less than it's value and every node to the right is greater
* We can search through Trees using BFS and DFS