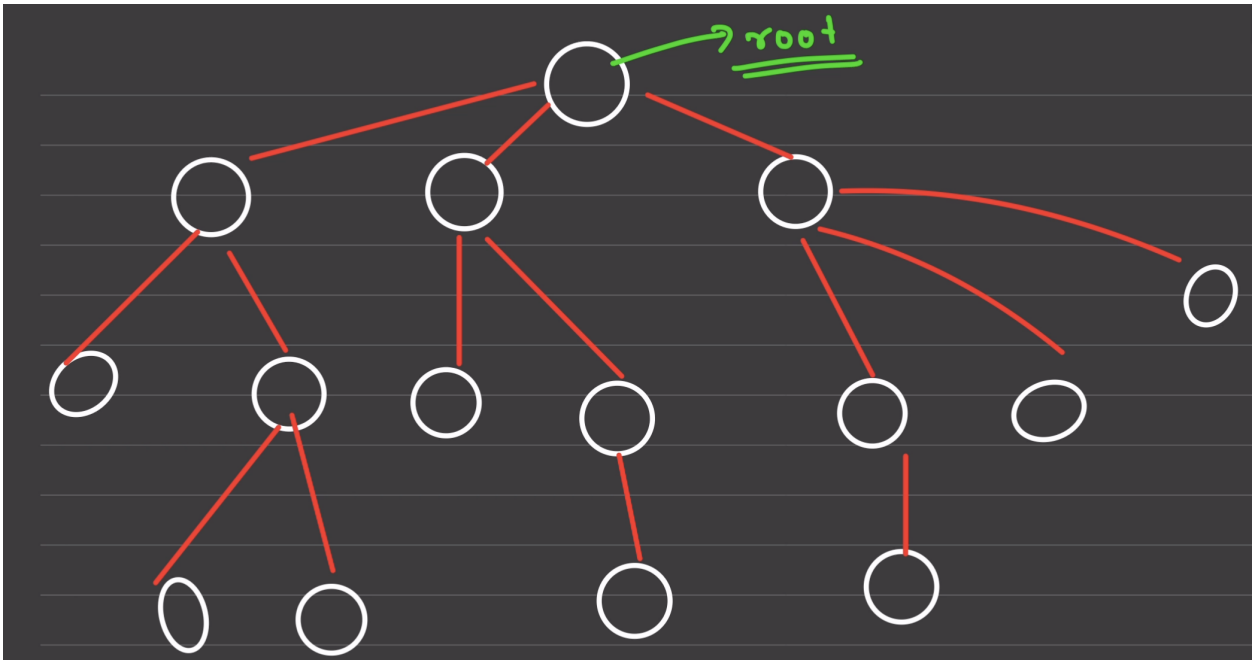
Trees

Created	@August 29, 2022 8:03 PM
Class	
Type	
Materials	
Reviewed	☐

Till now you must be already aware about Linear Data structures, like arrays, stacks, queues etc.

But in a computer, a lot of time we might have to deal with hierarchical problems, example a folder structure, a folder structure is not a linear application, it requires a sense of hierarchy, as there will be a root folder, then multiple files and folder inside it, then in each folder multiple more folders etc.

To solve these kind of situations we have a structure called as Trees.



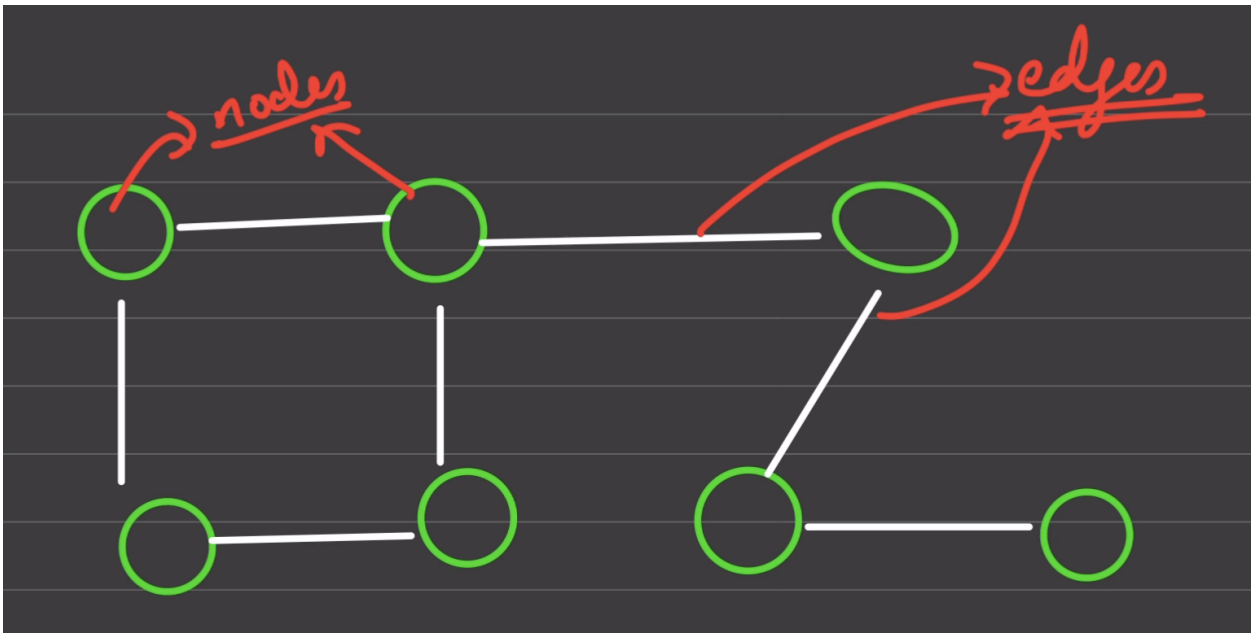
The above image is an example of a tree like structure.

How can we technically define trees ?

In order to define trees properly we need to have idea about one more data structure called as graphs. Let's briefly discuss about graphs…

What is a graph ?

Graph is a non linear data structure, that represents relationship between entities. For example, on an application like Instagram, a user can follow other user, and can be followed by some other users. So here user is a real life entity and the relationship between two users is defined by the fact that whether they follow each other or not. So this kind of relationship can be represented by graph data structure.



In a graph, entities are represented by nodes, and the relationship is represented as edges. If the relationship is one way, then edges can have directions. Whereas if it a two way relationship, then we just connect the nodes by a line.

In Google maps, we can represent places as nodes, and the roads connecting them as edges.

How is graph related to trees ?

Trees are a special type of graph, which do not have a cycle. So we can say, an acyclic graph is a tree.

A tree represents a hierarchical structure, which represents parent child relationship, where one parent can have multiple child.

Representation of Trees:

So trees can be represented by a hierarchical representation of nodes (object).

Terminologies Of Trees:

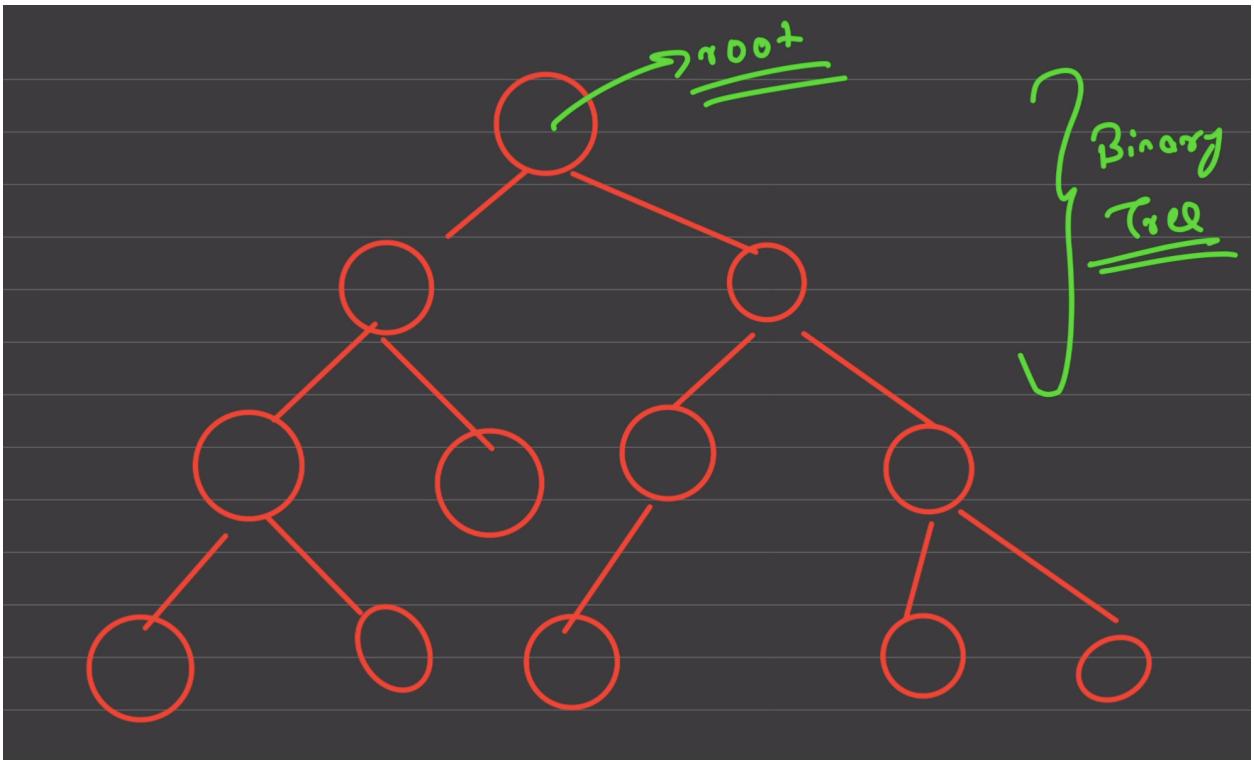
- Root: Root is a node which is not having any parent.
- Leaf: leaf is a node which is not having any children
- Binary Trees: A special type of tree in which any node can have at most 2 children.
- Ternary tree: A special type of tree in which any node can have at most 3 children.
- Generic Tree (n-ary tree): It is a general representation of tree, where a node can have as many possible children
- Subtrees: A subtree is a sub part of a tree, a tree has got many subtrees, and it’s similar to subset of a set, because in a subtree some nodes of the tree are included and some are not.

Application:

- To represent folder like structures
- In databases like MySQL, for implementing indexes (helps us to make our queries fast) we use trees
- A very famous machine learning algorithm called as decision trees is based on trees
- In browser the HTML we serve is rendered in form of a tree called as DOM Tree or Document Object Model tree.

Binary Trees

It is a special type of a tree in which a parent can have at most 2 children, that means, a parent can have 0 or 1 or 2 children but not more than that.



The above image represents a binary tree.

To represent a binary tree, we can have a collection of nodes, where each node can have the following 3 properties:

- Data
- Left Child
- Right Child

If a parent is not having any one or both the children then the left and right properties can be null.

```
class Node {
  constructor(d) {
    this.data = d;
    this.left = null;
    this.right = null;
  }
}
```

How to read trees ?

Trees are hierarchical and it is not as simple as a linked list to read. Linked lists were linear so were able to just iterate and print it, whereas trees are not that easy to read.

So there are two ways, to read a tree:

- DFS/DFT → Depth first search / Depth first traversal
- BFS/BFT → Breadth first search / Breadth first traversal

Depth first Traversal

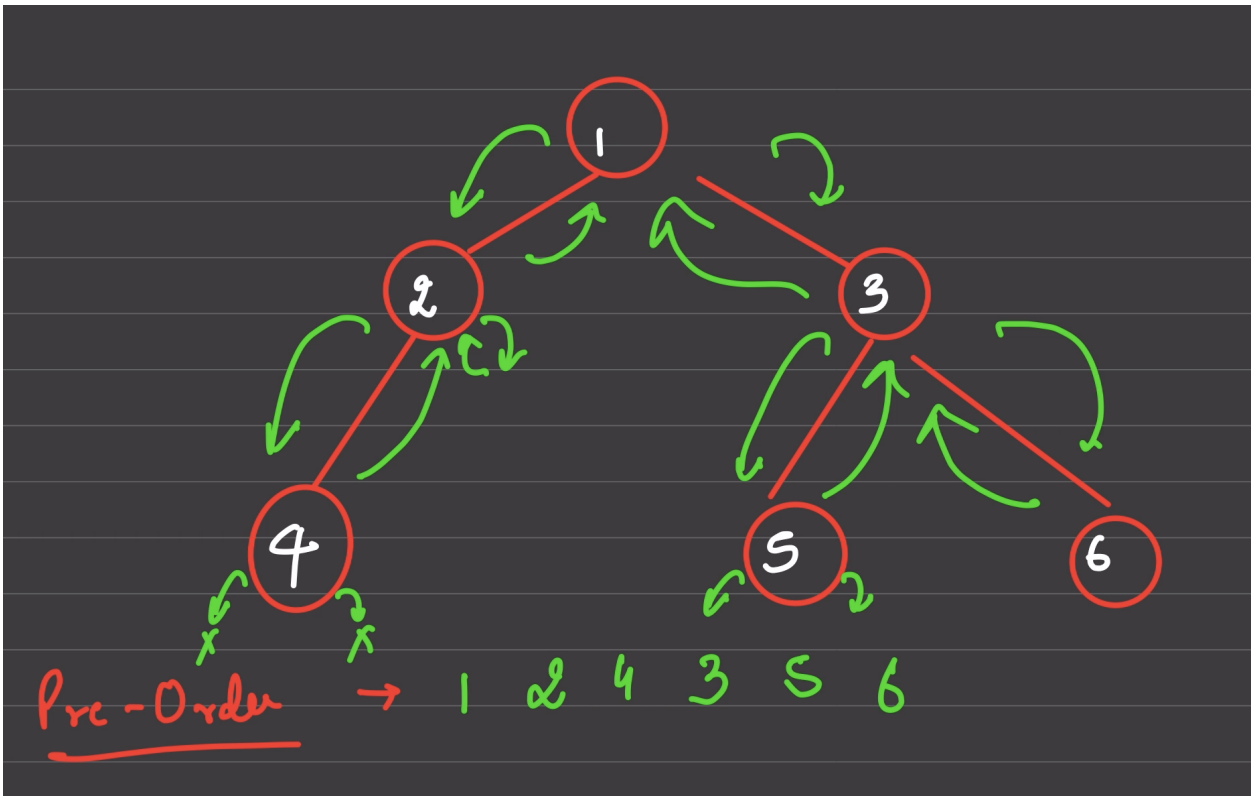
In a depth first traversal, we start reading the tree from the root node, and then we try to explore one subtree completely in depth, keeping the other subtree waiting. This is true for the underlying subtrees as well.

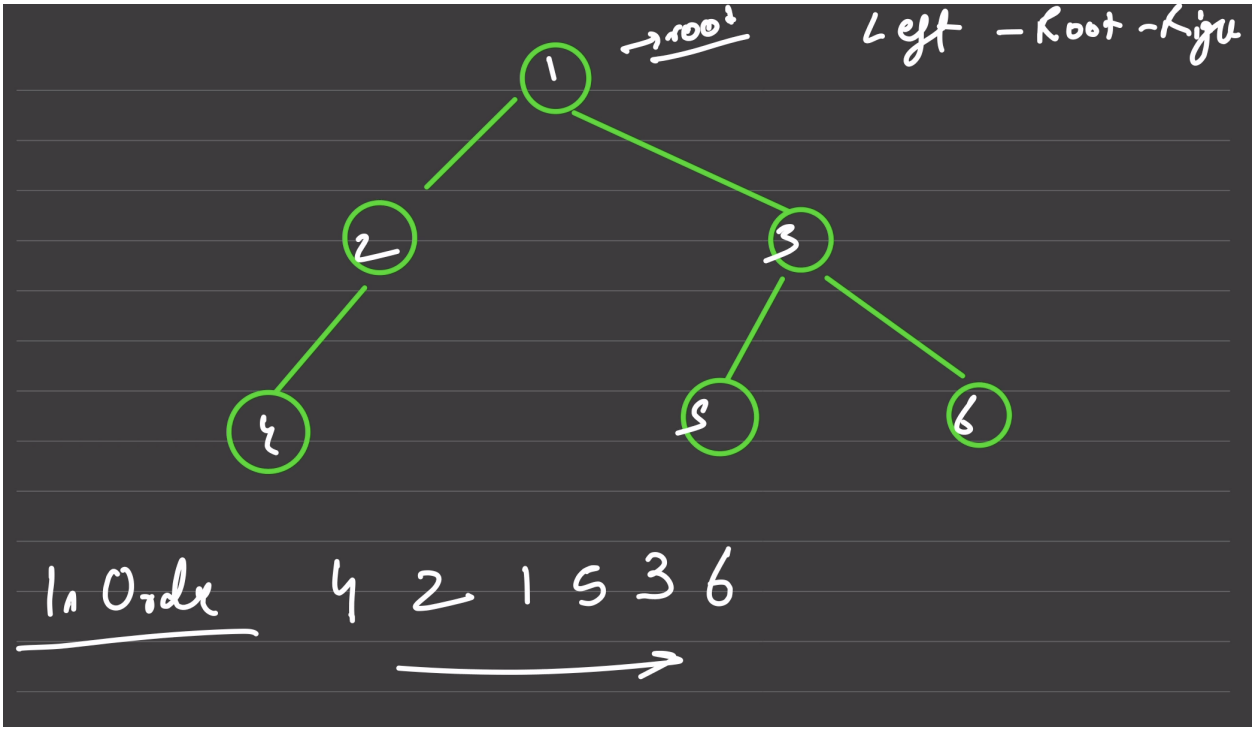
There are 3 ways to do Depth first traversal on a binary tree:

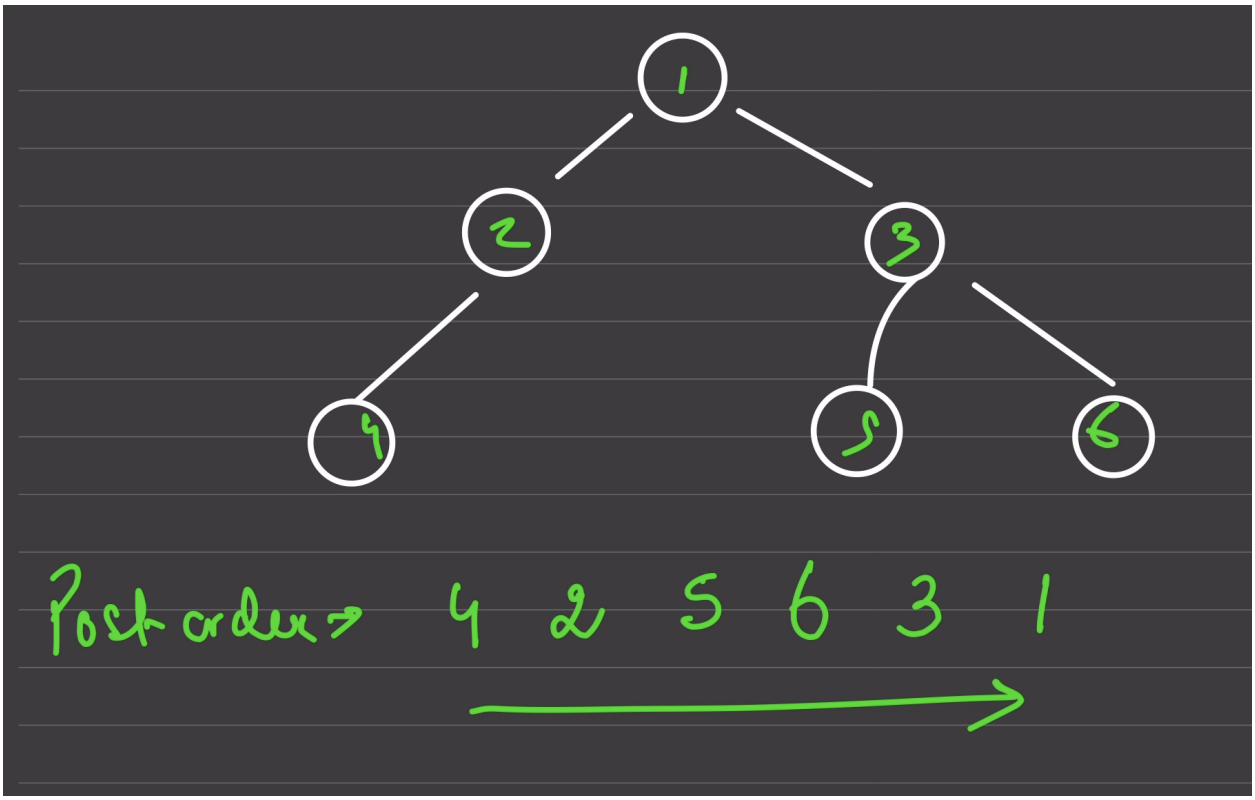
- Pre-order
- In-order
- Post-order

So one thing common in all of the above three traversals is, the left subtree is read completely before the right subtree. The only difference lies in when do we read the root ?

- Preorder: Root → left subtree → right subtree
- Inorder: left subtree → Root → right subtree
- Postorder: left subtree → right subtree → root


1
2
3
4
5
6
x
x
Pre-Order → 1 2 4 3 5 6


→root
Left - Root - Right
1
2
3
4
5
6
In Order 4 2 1 5 3 6



How to implement these ?

Pre-order:

We can implement this pre-order function recursively.

Lets assume we have a function `pre` that takes a parameter node → `pre(node)` and we say that `pre(node)` can perform pre -order on a tree.

How to implement `pre` ?

We know that for a pre order, we read the root, then the left subtree and then the right subtree

Let's divide the recursion in three parts -

- Base case → if the node is null, we return as it is.
- Self work → we know how to read data of a node so we do that
- Assumption → we assume that `pre` function will read left subtree and then right subtree correctly

```
function pre(node) {
  if(node == null) return;
  console.log(node.data);
  pre(node.left);
  pre(node.right);
}
```

Inorder

We can implement this in-order function recursively.

Lets assume we have a function `ino` that takes a parameter node → `ino(node)` and we say that `ino(node)` can perform in -order on a tree.

How to implement `ino` ?

We know that for a in order, we read the left subtree, then the root and then the right subtree

Let's divide the recursion in three parts -

- Base case → if the node is null, we return as it is.
- Self work → we know how to read data of a node so we do that
- Assumption → we assume that `ino` function will read left subtree and then right subtree correctly

```
function ino(node) {
  if(node == null) return;
  ino(node.left);
  console.log(node.data);
  ino(node.right);
}
```

Post order:

```
function post(node) {
  if(node == null) return;
  post(node.left);
  post(node.right);
```

```
  console.log(node.data);
}
```

Q: Given a root of a tree, find the maximum element of the tree. (Read the data only once)

If it would have been an array, then in order to find the max element, we would have traversed the whole array and got the answer.

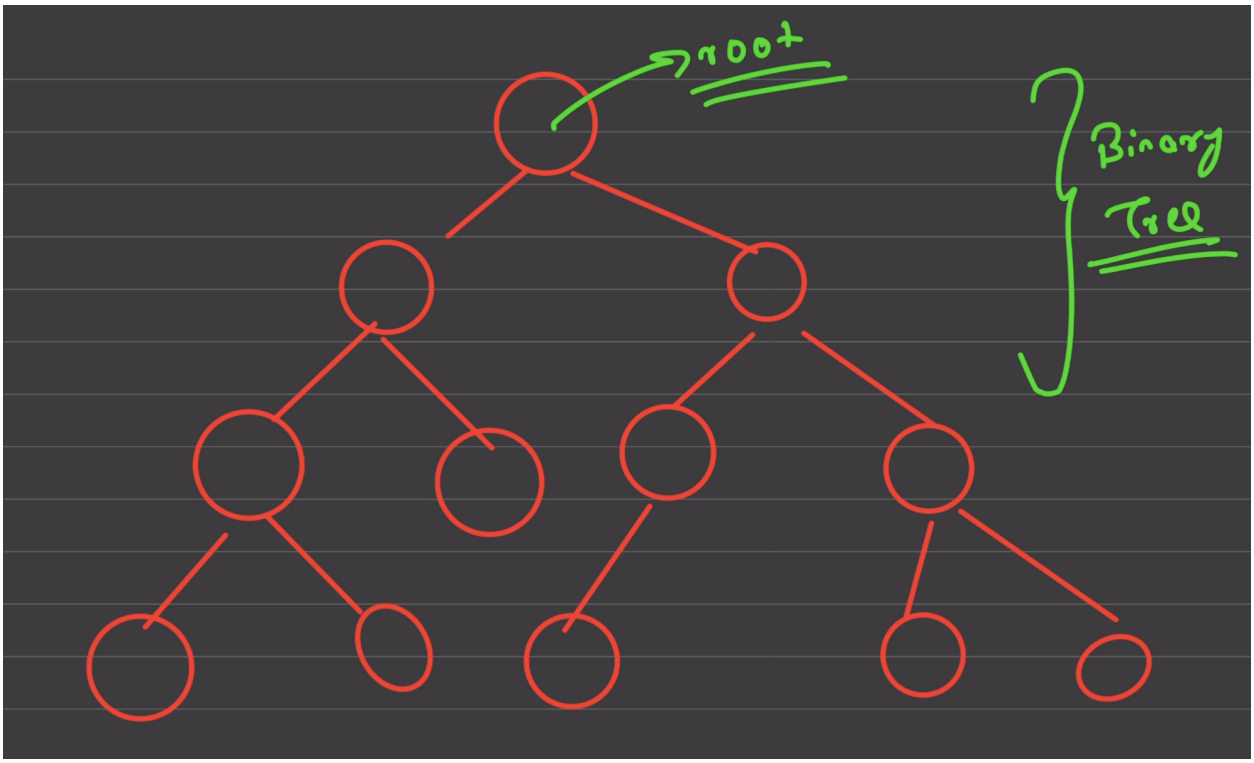
So in a similar way, how about we read the whole tree and then get our maximum, and in order to read the tree, we can use Depth first traversal.

We can maintain a global ans variable and compare value of each node of the tree and then if the value is greater than global answer, we update it.

```
let ans = -Infinity;
function getmax(root) {
  if(root == null) return;
  if(root.data > ans) ans = root.data;
  getmax(root.left);
  getmax(root.right);
}
```

Max Depth or Height Of a tree

Height of the tree is defined as the length of the longest path starting from a root node to a leaf node of the tree.



So for the any tree the height can be defined in two ways. We know that height is the longest path from root to any leaf, now sometime we count the number of nodes in the path and consider that as the length or we sometimes count the number of edges in the path and consider that as the length.

So if we consider number of nodes, then for above tree height is 4 otherwise if we consider number of edges then height is 3. If not specifically defined then we consider number of edges.

Write down a code to calculate height of a binary tree.

Solution;

What if we just got a single node, no other node apart from it, and we are considering number of edges then the height will be 0.

And we can draw one more conclusion, that if we somehow already know the height of the left subtree and the height of the right subtree, then height of the whole tree will be 1 + max(height of left subtree, height of right subtree);

Why this works, because when we take maximum from the left or right subtree height, we are actually choosing a branch from either left or right subtree which will be the

maximum length branch. And then to attach root node, to this branch we need one more edge, so that's why we add + 1 to the maximum we fetched.

We can write a function `height(node)`, which will return the height of the tree root at given `node`

Now if we assume that if we use the height function for the left and right subtree we get the correct result as the length of left and right subtree.

that means `height(node.left)` and `height(node.right`) will work correctly.

So we take max of them and then add 1 to it for considering the edge from root to the subtree.

And then we can say

`height(node) = 1 + Math.max(height(node.left), height(node.right))`

```
function height(node) {
    if(node == null) {
        return -1;
    }

  let leftHeight = height(node.left);
  let rightHeight = height(node.right);
  return 1 + Math.max(leftHeight, rightHeight);
}
```

Breadth first traversal / Level order

So we talked about the fact that we can read trees in two ways, depth first and breadth first

Breadth first traversal is actually called as level order traversal. Because in level order, instead of read a depth, we read in the breadth of the tree.

If you carefully analyse the level order traversal, then for a parent which is coming first in the level it's children will also be processed first.

```
/**
 *
 *              10
 *            /   \
 *          5      15
 *         /      / \
 *        1     24  33
 *       /
 *      0
*/
```

So in the above tree, because 5 is coming before 15 in the level, it’s children will also be processed before 15’s children and this gives us feeling of First come first serve. And for first come first serve we know that Queues can help us. So we can setup a queue, and start inserting nodes from root. Once we insert a node, we will wait for it’s turn to come in the queue for processing, once we fetch the node, we insert it’s left and right child, if present in queue, and process this tree till the time queue is not empty.

```
function levelorder(root) {
    let qu = new Queue();
    qu.enqueue(root);
    while(!qu.isEmpty()) {
        let curr = qu.getFront();
        qu.dequeue();
        console.log(curr.data);
        if(curr.left) {
            qu.enqueue(curr.left);
        }
        if(curr.right) {
            qu.enqueue(curr.right);
        }
    }
}
```

Binary Search Tree

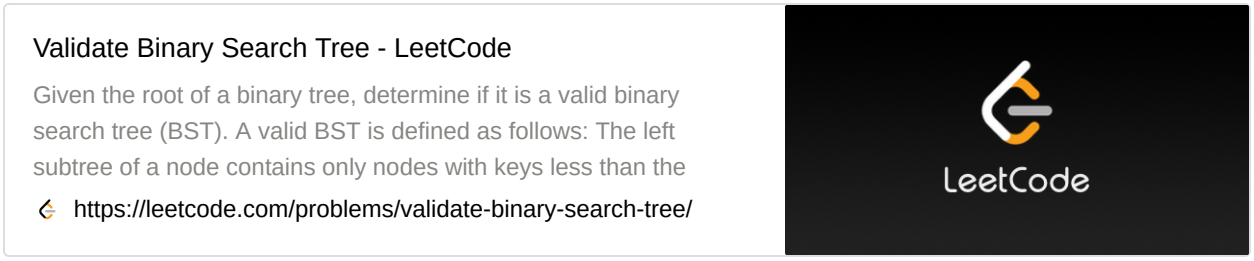
Binary search tree is a special binary tree (all the properties of binary tree it will contain), and it has a special property. In a BST, with respect to every root, every node of the left subtree of the root is less than root and every node of the right subtree is greater than root. And this definition is recursively true for all the subtrees.

```
/**
 *
 *              10
 *            /   \
 *          5      15
 *         /      / \
 *        1     12  33
 *       /
 *      0.
*/
```

If we try to get in-order traversal of a BST, it is always sorted in ascending order.

Problem:

Given a binary tree, check if it is a BST or not ?




```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {boolean}
 */
let last = -Infinity;
let ans = true;

function inorder(root) {
    if(root == null) return;
    inorder(root.left);
    // console.log(node.data);
    if(root.val <= last) {
        ans = false;
```

```
        last = root.val;
    } else if(ans != false) {
        last = root.val
    }
    inorder(root.right);
}
var isValidBST = function (root) {
    inorder(root);
    let result = ans;
    last = -Infinity;
    ans = true;
    return result;
};
```

Now for a Binary tree to be a BST, every root, should be greater than all elements of the left subtree, and less than all elements of the right subtree, and this should be true recursively for all roots.

Now, instead of checking the root with all the element of lst and rst, how about we find the maximum element of lst and minimum element of rst, and if the root is greater than max element of left subtree and root is less than min element of right subtree and this definition holds true for all subtrees also then can we say it is a bst.

```
/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */
/**
 * @param {TreeNode} root
 * @return {boolean}
 */

function checkIsBST(root) {
    if(root == null) {
        return {isBST: true, min: Infinity, max: -Infinity};
    }
    const left = checkIsBST(root.left);
    const right = checkIsBST(root.right);

    if(!left.isBST || !right.isBST) {
        return {
            isBST: false,
```

```
            min: Math.min(left.max, right.max, root.val),
            max:Math.max(left.min, right.min, root.val)
        }
    }
    if(left.isBST && right.isBST) {
        if(root.val > left.max && root.val < right.min) {
            return {
                isBST: true,
                max: Math.max(left.max, right.max, root.val),
                min: Math.min(left.min, right.min, root.val)
            }
        } else {
            return {
                isBST: false,
                min: Math.min(left.max, right.max, root.val),
                max:Math.max(left.min, right.min, root.val)
            }
        }
    }
}

var isValidBST = function(root) {
    return checkIsBST(root).isBST;
};
```

Problem:

We will be given N numbers , and we have to create a BST using them.

10, 5, 15, 1, 12, 33

We can get multiple different BSTs, but we need to create any one of them.

Solution:

We know that bst has a property that for every subtree, elements lesser than root of the subtree goes to the left subtree of root, and elements greater than root of the subtree goes to the right subtree of root.

So if we have a bunch of numbers, we can just start inserting them one by one in our tree such that we satisfy this property.

```
/**
            10
           /  \
```

```
          5   14
         /     / \
         2.   12. 45
        /     / \.
        0    11  13
       / \.      /
      -1  1     12.5

*/
```

Problem:

Remove a node with value x in a given bst.

Hint: When we remove a node, then we need to rearrange the nodes, in the subtree of the removed node.

In order to remove a node, we need to first try to search that node. Because the tree is a BST, searching will be very easy. At any point of time we can discard one complete tree while doing the searching. For example in the above the, if we want to search for 14, then we compare 14 with 10, and 14 > 10, so 14 can never be inside the left subtree. So we can discard the complete left subtree and consider the right subtree.

So once we have searched for the element, then we will try to remove it.

Now while removing we can see that there can be different situations with a node to be removed. The node can be a leaf, it can have just one child, or both children. So case to case we need to handle it.

1. If the node is leaf, then we don’t need any rearrangement, and we can just replace it with null. Example: 13
2. if the node has only one child, we can attach this child directly to the parent of the node. Example: 5, if we want to remove 5, we can attach the subtree rooted at 2, to the left of 10 which is parent of 5.
3. If node has got both the children, then we need more rearrangement, we need to replace node with a value that can follow the property of BST. So we can either replace it with biggest (rightmost) node of left subtree or smallest (leftmost) node of right subtree. Ex: 15, so we can replace 15 with 14 which is right most of lst, or with 33, which is left most of rst.

```
removeHelper(root, x) {
        /**
         * Time: O(h)
         * Space: O(h)
         */
        if(root == null) return null;
        if(root.data == x) {
            // then current node should be removed
            // Case 1: leaf node
            if(root.left == null && root.right == null) return null;

            // case 2: single child
            if(root.left == null && root.right != null) {
                // we dont have left child but have got a right child
                let temp = root.right;
                root.right = null;
                return temp;
            }
            if(root.left != null && root.right == null) {
                // we dont have right child but have got a left child
                let temp = root.left;
                root.left = null;
                return temp;
            }

            // Case 3: both the children, so we will be using rightmost of lst
            let temp = root.left;
            while(temp.right != null) temp = temp.right;
            root.data = temp.data;
            this.removeHelper(root.left,temp.data);
            return root;
        }
        if(x < root.data) {
            root.left = this.removeHelper(root.left, x); // root = 14
        } else {
            root.right = this.removeHelper(root.right, x);
        }
    }

    remove(x) {
        this.removeHelper(this.root, x);
    }
```