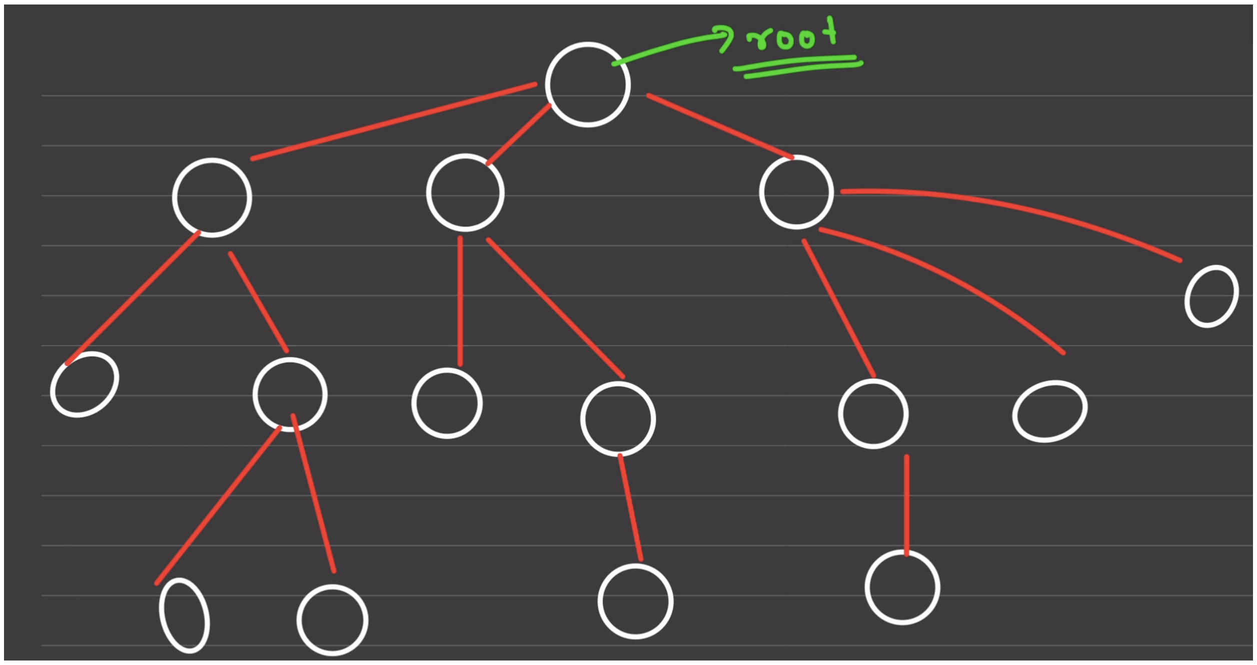
Trees

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.



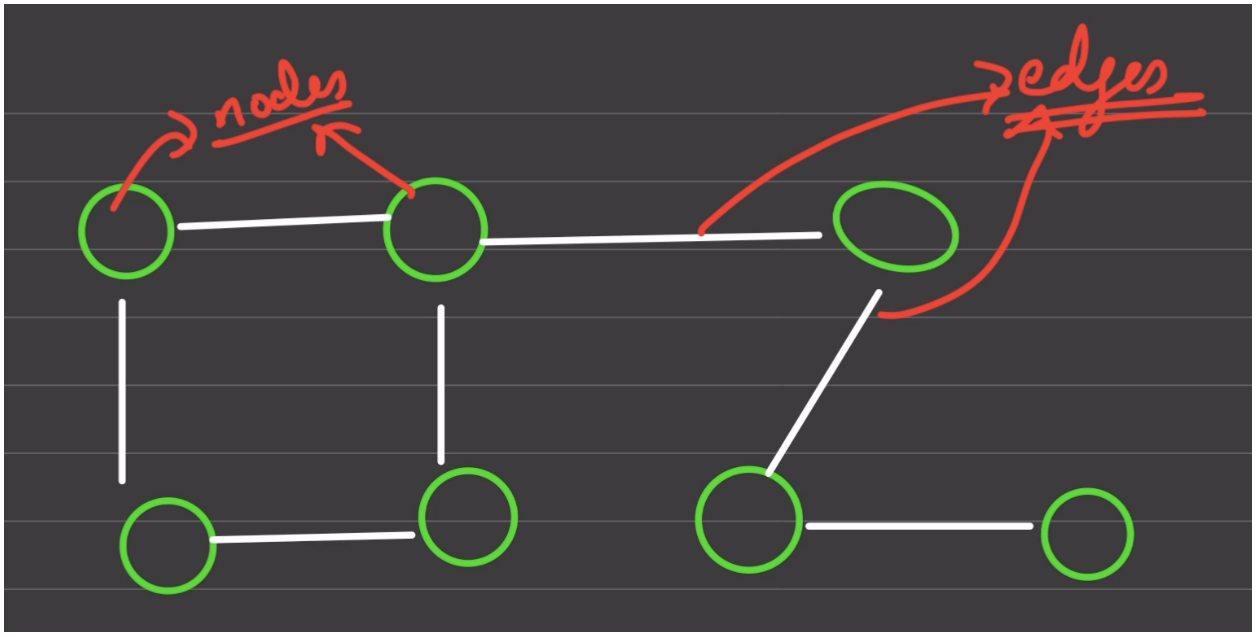
The above image is an example of a tree like structure.

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:

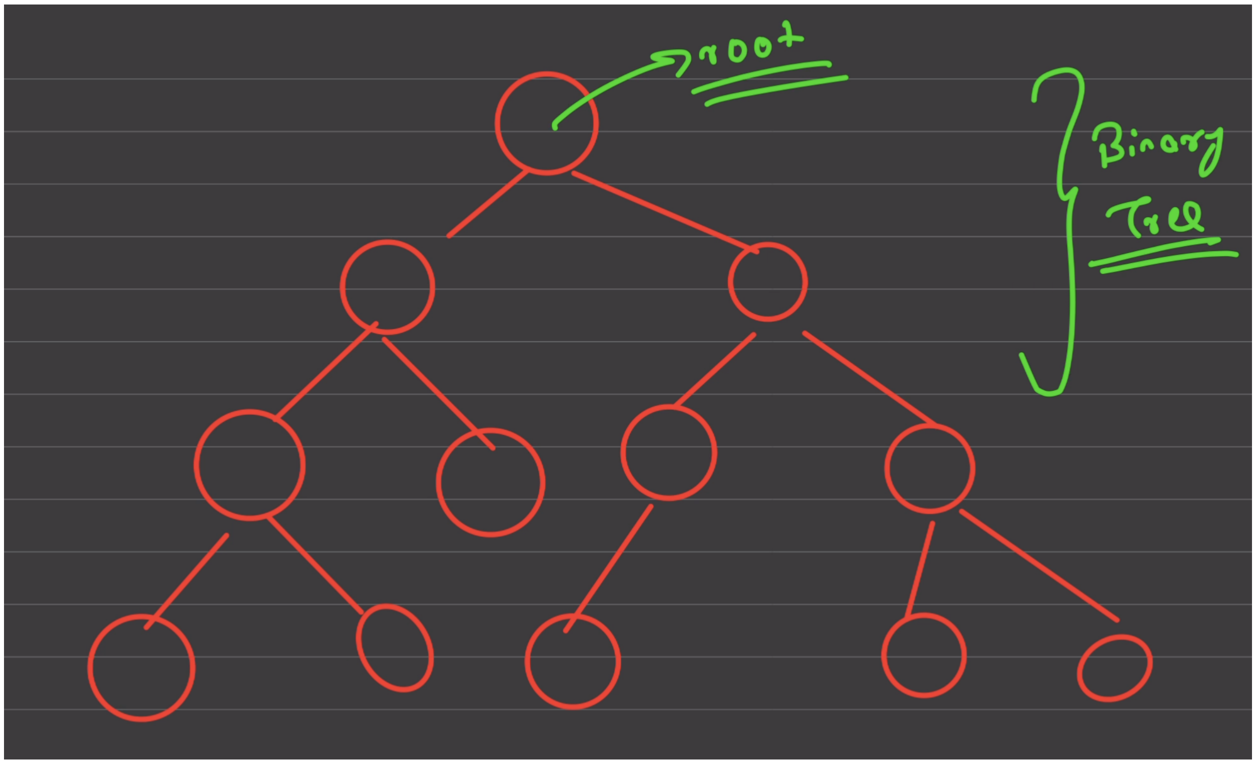
- Root: Root is a node which is not having any parent.
- Leaf: leaf is a node which is not having any children
- Binary Trees: A special type of tree in which any node can have at most 2 children.
- Ternary tree: A special type of tree in which any node can have at most 3 children.
- Generic Tree (n-ary 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

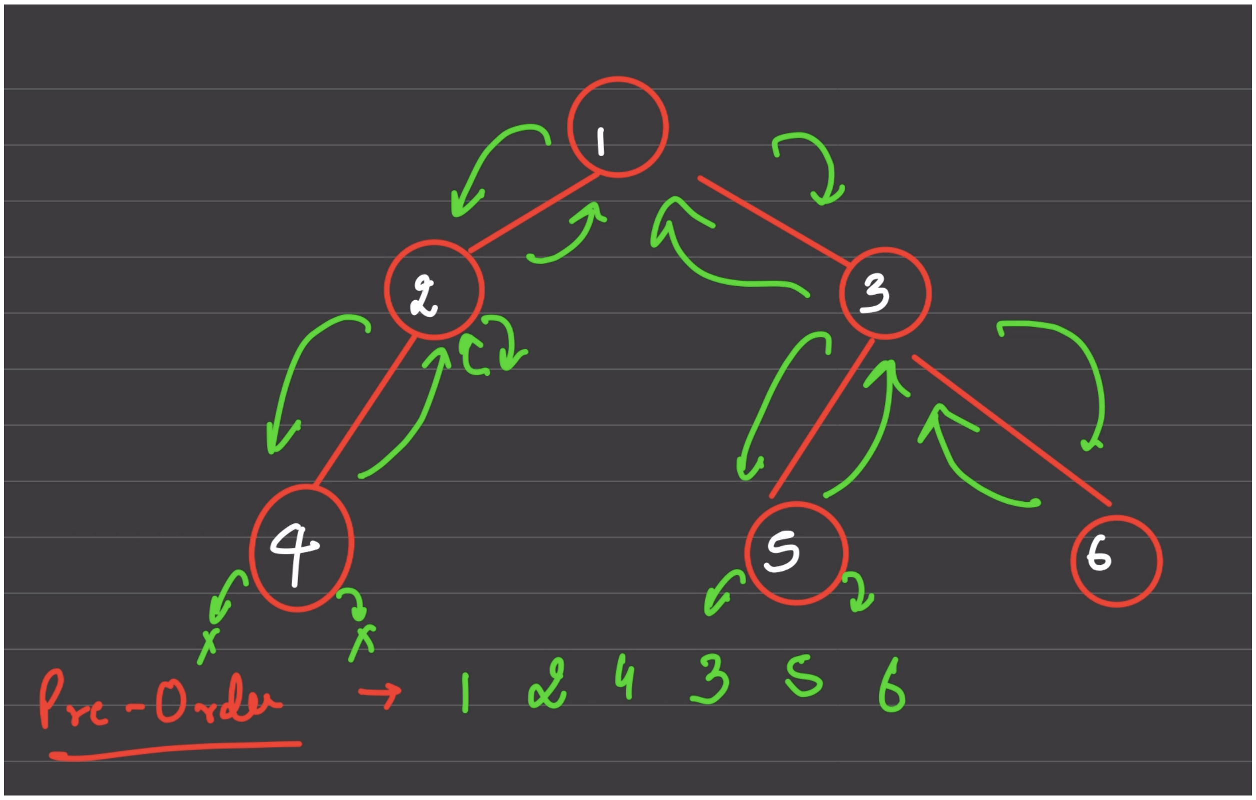
In a depth first traversal, we start reading the tree from the root node, and then we try to explore one subtree completely in depth, keeping the other subtree waiting. This is true for the underlying subtrees as well.

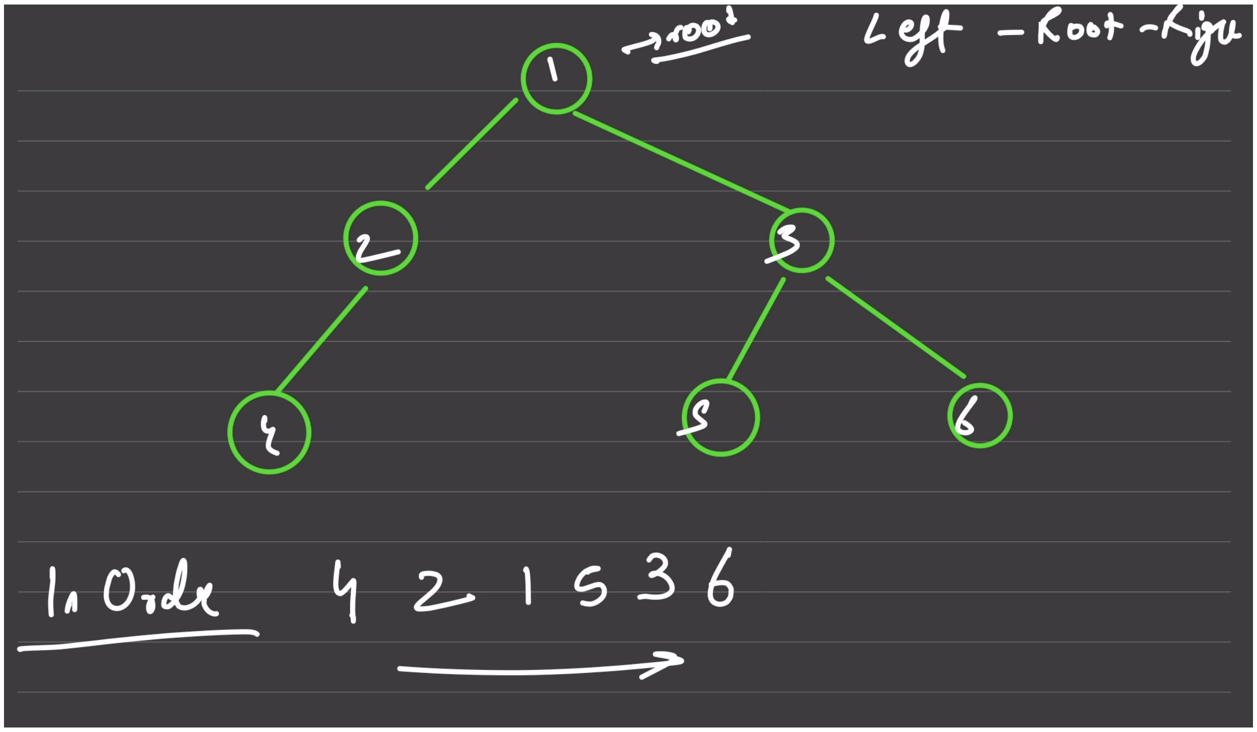
There are 3 ways to do Depth first traversal on a binary tree:

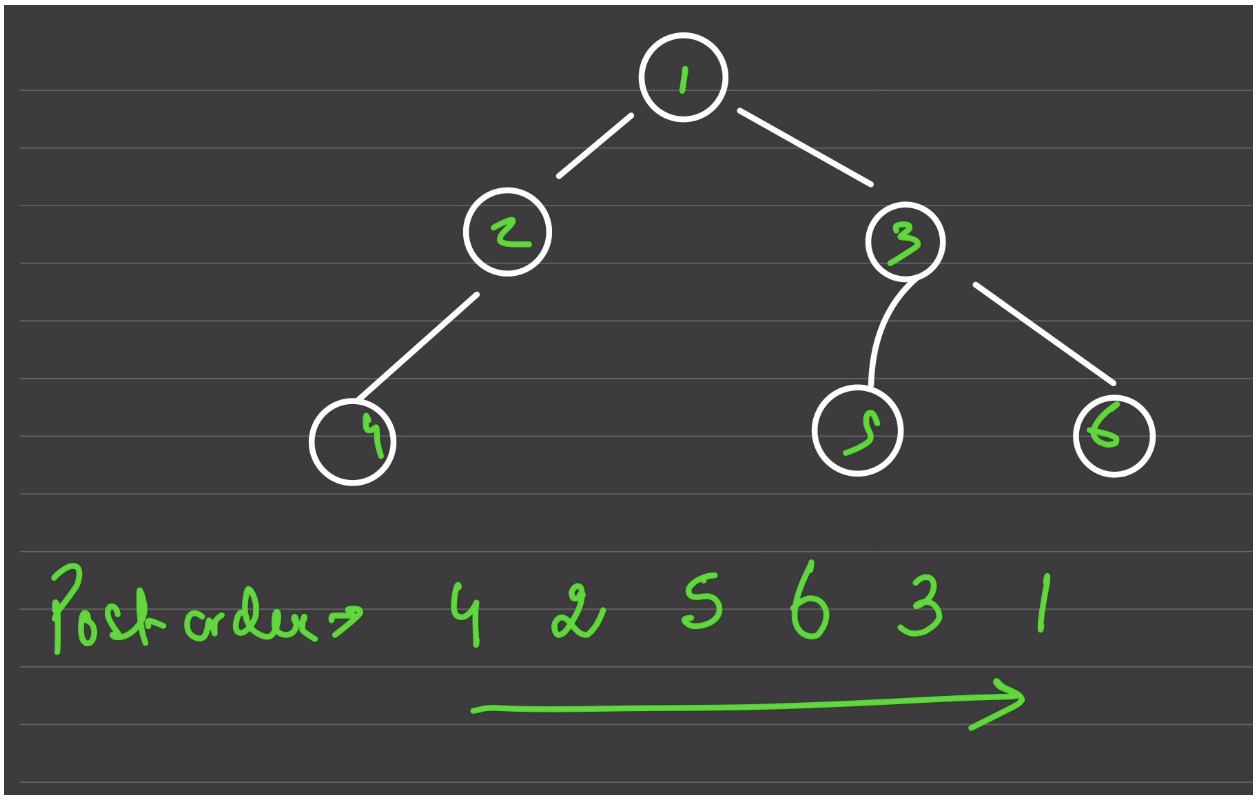
- Pre-order
- In-order
- Post-order

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);
}
```