Trees

Fall 2024
Data Structures

Trees

- Non-linear data structures
  - Hierarchical
- Before, each element followed another one.
  - Like a linked list
  - Here, we have branching out.
- Used to represent and organize data in a way that is easy to *navigate* and *search*

Use cases

- Hierarchical data
  - File systems, organizational models, etc
- Databases
  - Used for quick data retrieval
- Routing tables
  - In networking
- Sorting / Searching
- Priority Queues
  - Commonly implemented using binary heaps

Node

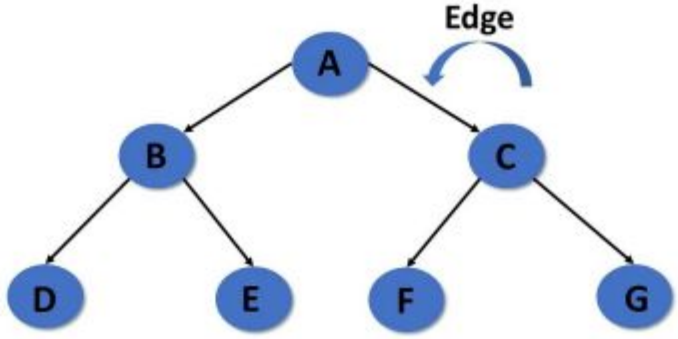
- A node is a structure that contains a key or value and pointers in its child node.
- Each node can have an arbitrary number of children.
    - Not in binary trees.

Root

- Root is the **first** node of the tree.
- It is the **initial node**
- In a tree, there can only be **one** root.

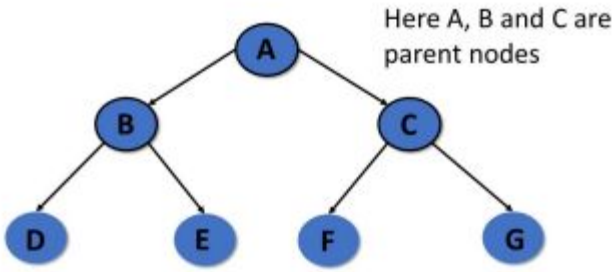
Edge

- The connecting link of any two nodes is called an **edge**
- If there are **N** number of nodes, there are **N-1** edges.



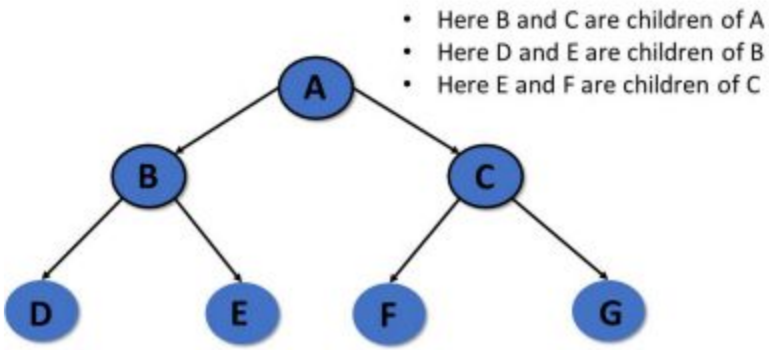
Parent

- A node which is *predecessor* of any node.
- A node with a branch from itself to any other successive node .



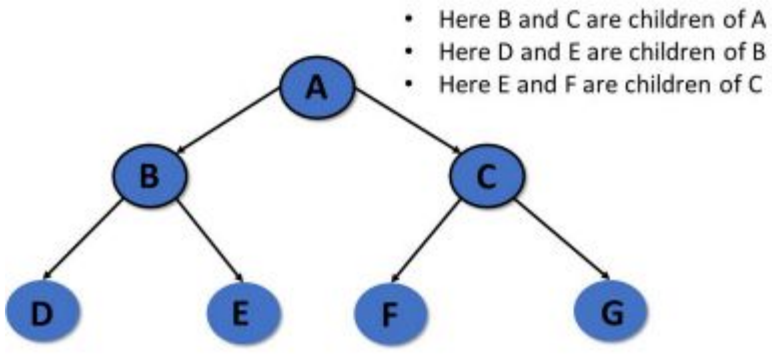
Child

- A descendant of any node is known as child node.
- Every node other than the **root** is a child node.



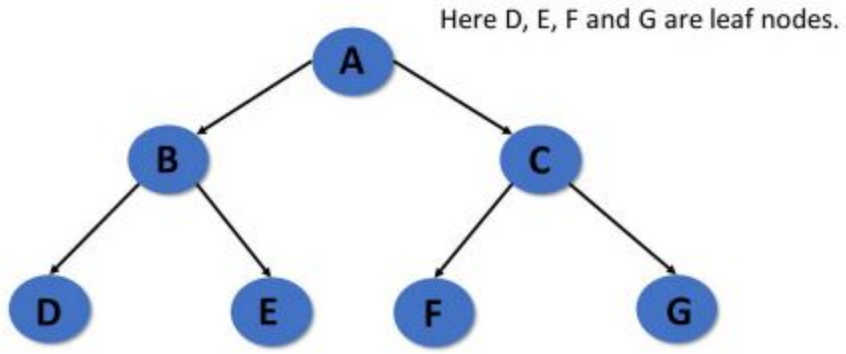
Siblings

- Nodes that belong to the same parent .



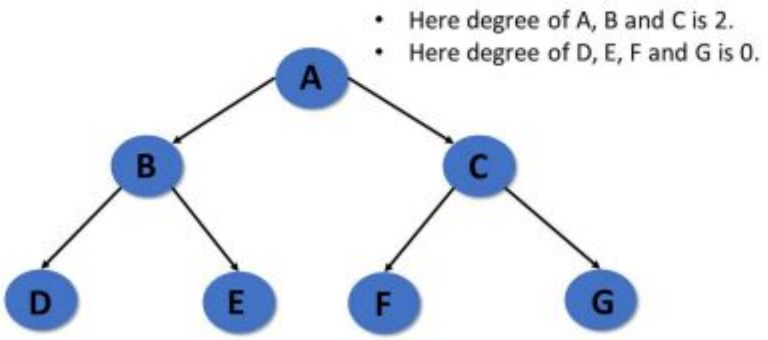
Leaf

- Nodes with no children
- Also called
  - External nodes
  - Terminal nodes



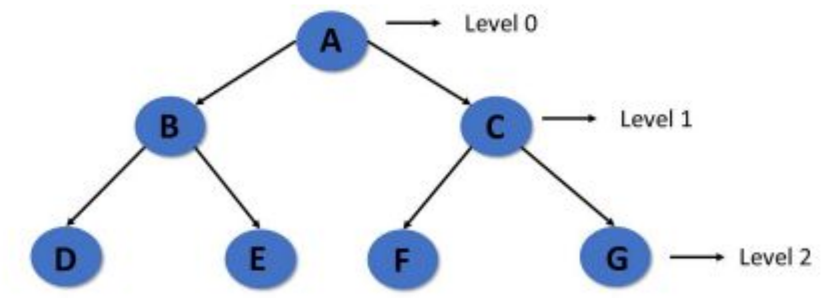
Degree

- Total number of children of a node is called the **degree**



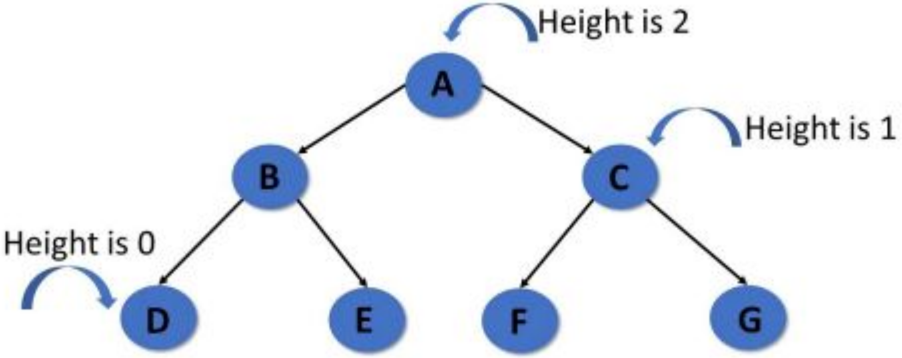
Level

- Root node is level 0
- Root's nodes children are level 1
- …

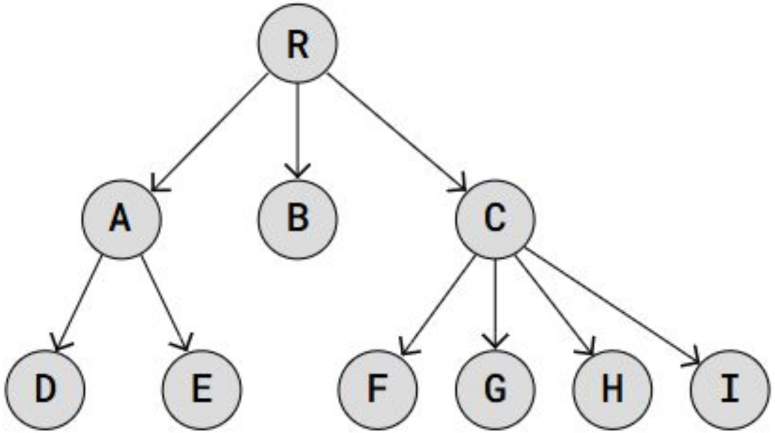


Height

- Number of edges from the leaf node to the particular node in the longest path.
- Height of tree = height of root
- Tree height of all *leaf* nodes are 0.

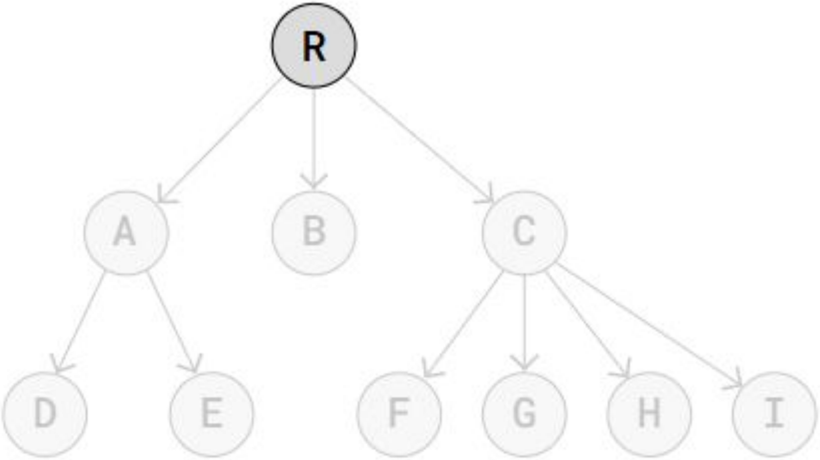


Terminology



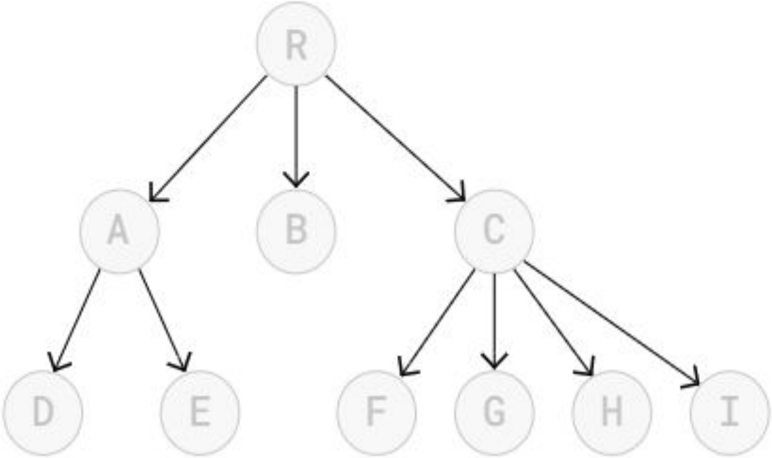
Root:
Edges:
Nodes:
Leaf Nodes:
Child Nodes:
Parent Nodes:
Tree height:
Tree size:

Terminology



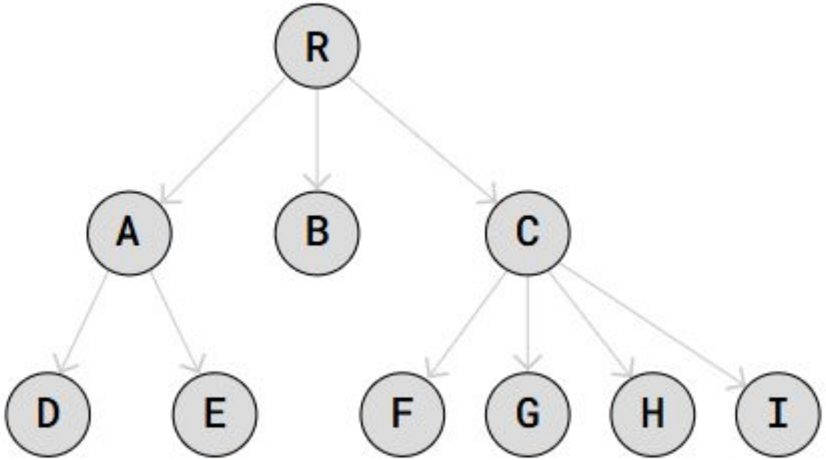
Root: R
Edges:
Nodes:
Leaf Nodes:
Child Nodes:
Parent Nodes:
Tree height:
Tree size:

Terminology



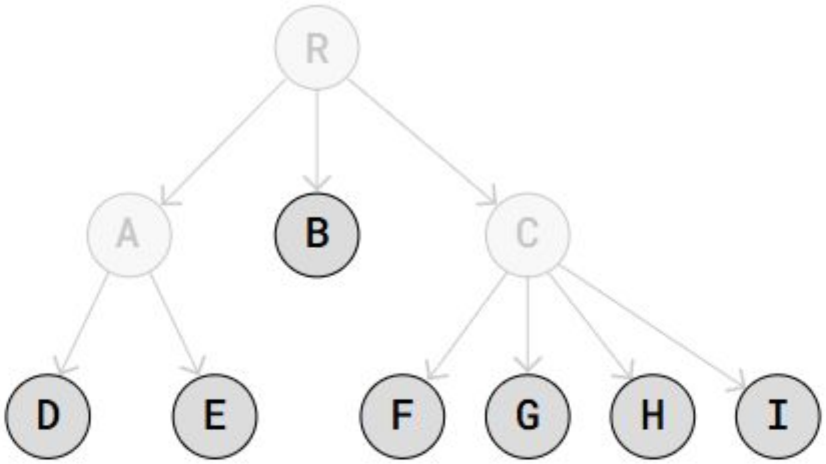
Root: R
Edges: Connections (arrows)
Nodes:
Leaf Nodes:
Child Nodes:
Parent Nodes:
Tree height:
Tree size:

Terminology



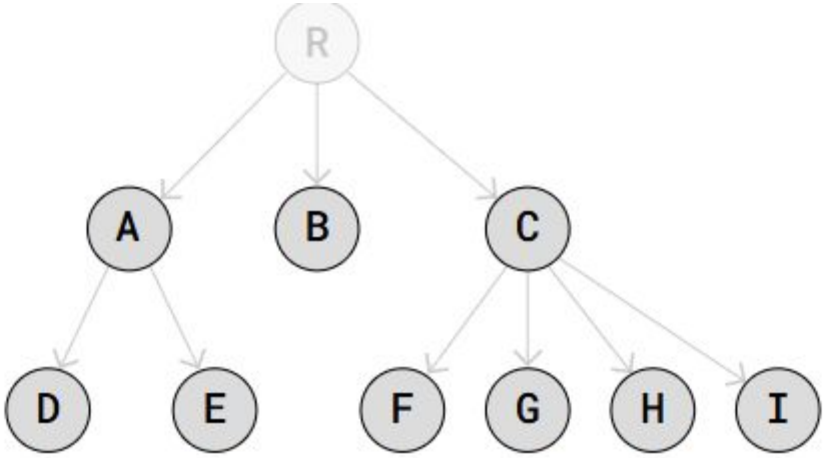
Root: R
Edges: Connections (arrows)
Nodes: {R, A, B, C, D, E, F, G,H, I}
Leaf Nodes:
Child Nodes:
Parent Nodes:
Tree height:
Tree size:

Terminology



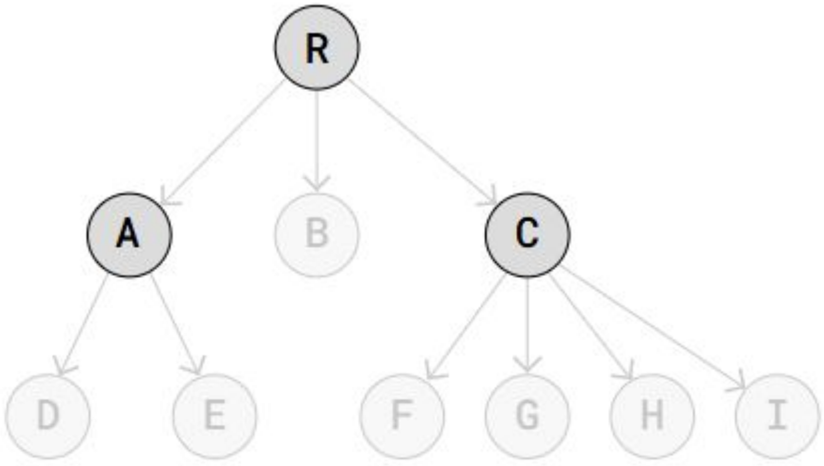
Root: R
Edges: Connections (arrows)
Nodes: {R, A, B, C, D, E, F, G,H, I}
Leaf Nodes: {B, D, E, F, G, H, I}
Child Nodes:
Parent Nodes:
Tree height:
Tree size:

Terminology



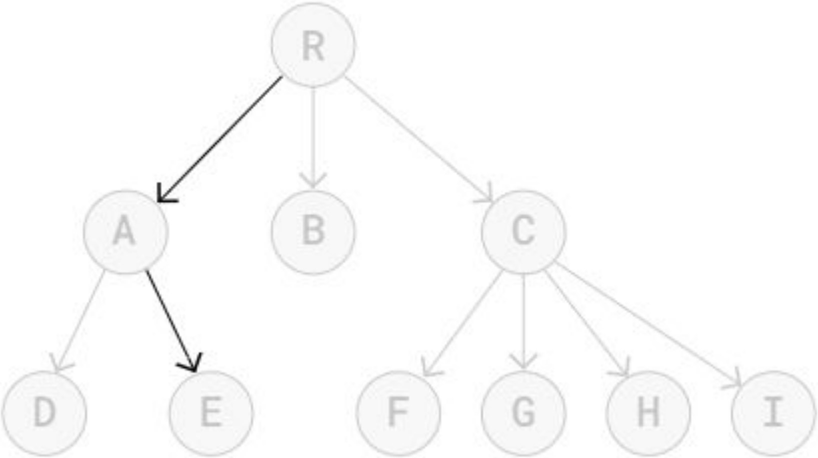
Root: R
Edges: Connections (arrows)
Nodes: {R, A, B, C, D, E, F, G,H, I}
Leaf Nodes: {B, D, E, F, G, H, I}
Child Nodes: {A, B, C, D, E, F, G, H, I}
Parent Nodes:
Tree height:
Tree size:

Terminology



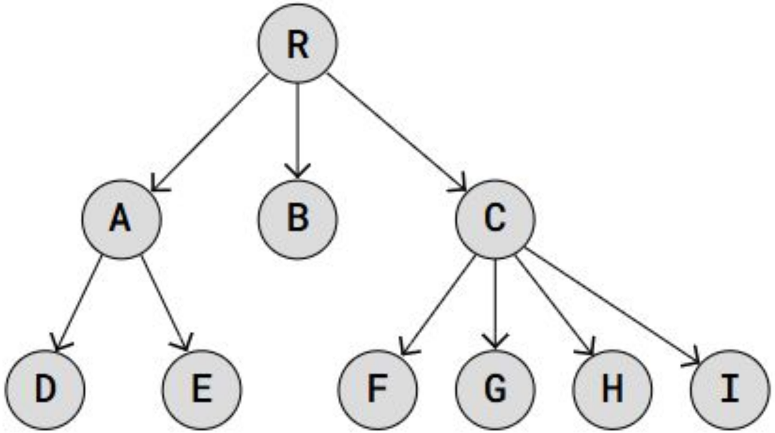
Root: R
Edges: Connections (arrows)
Nodes: {R, A, B, C, D, E, F, G,H, I}
Leaf Nodes: {B, D, E, F, G, H, I}
Child Nodes: {A, B, C, D, E, F, G, H, I}
Parent Nodes: {R, A, C}
Tree height:
Tree size:

Terminology



Root: R
Edges: Connections (arrows)
Nodes: {R, A, B, C, D, E, F, G,H, I}
Leaf Nodes: {B, D, E, F, G, H, I}
Child Nodes: {A, B, C, D, E, F, G, H, I}
Parent Nodes: {R, A, C}
Tree height: 2
Tree size:

Terminology



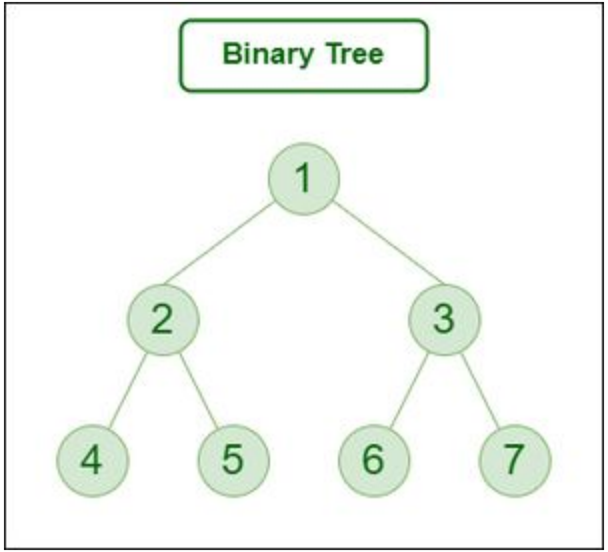
Root: R
Edges: Connections (arrows)
Nodes: {R, A, B, C, D, E, F, G,H, I}
Leaf Nodes: {B, D, E, F, G, H, I}
Child Nodes: {A, B, C, D, E, F, G, H, I}
Parent Nodes: {R, A, C}
Tree height: 2
Tree size: 10

types of trees

- There are several types of trees:
  - *Binary Trees*
  - *Binary Search Trees (BST)*
  - AVL Trees
  - B-trees
  - Tries
  - …

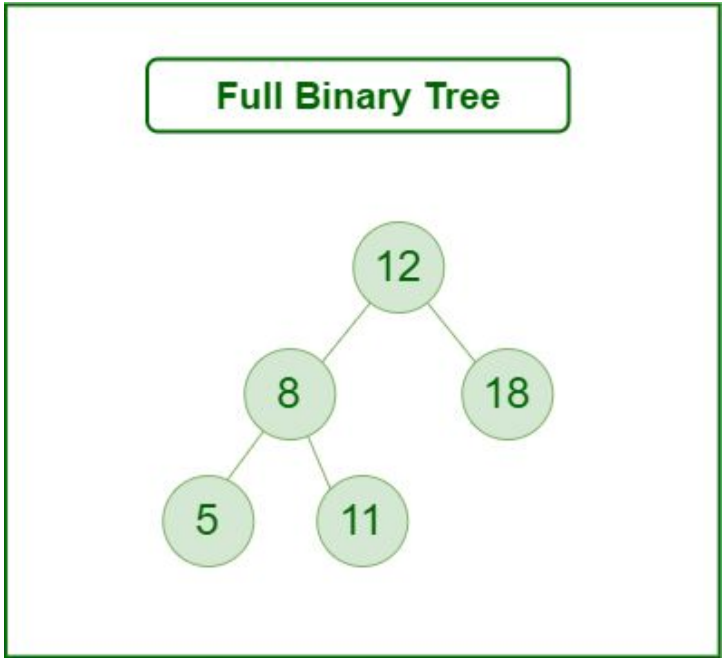
binary trees

- Different types of binary trees.
  - Based on the number of children:
    - Full Binary Tree
    - Degenerate binary tree
  - Based on the **completion of levels**
    - Complete binary tree
    - Perfect binary tree
    - Balanced binary tree



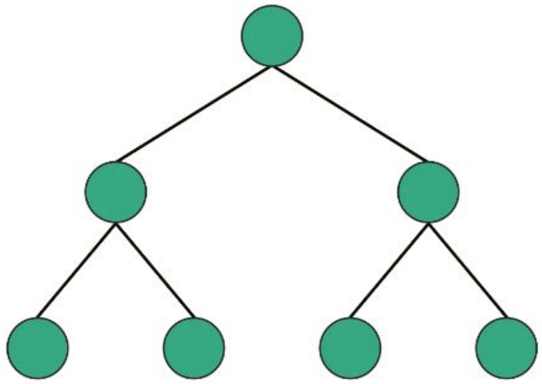
full binary tree

- maximum 2 children per node
    - left and right child
- Either 0 child and 2 children



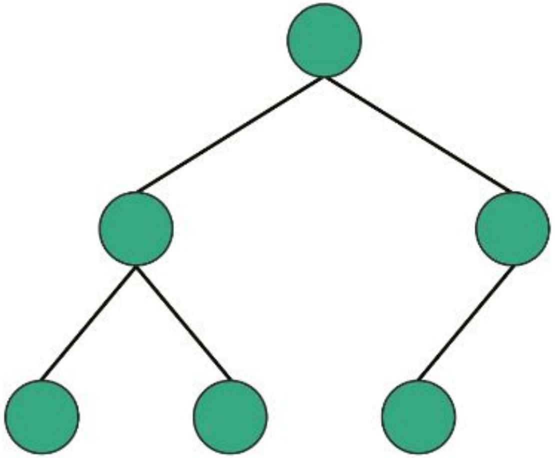
perfect binary tree

- Where all interior nodes have **two** children and **all** leaves have the same *depth* or same *level*.
- A perfect binary tree is a **full** binary tree.



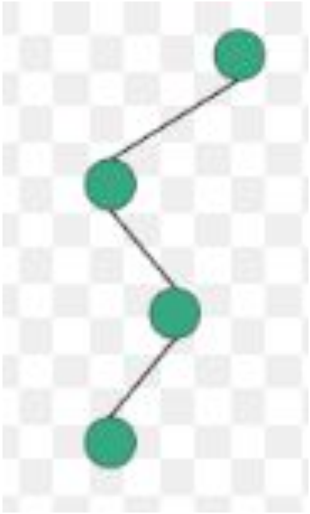
complete binary tree

- Every level except the last is **completely** filled.
- A perfect tree is always complete
- But a complete tree is **not** always perfect
- Can be efficiently represented using an array.
- If the last level is not full, it should start from the left.

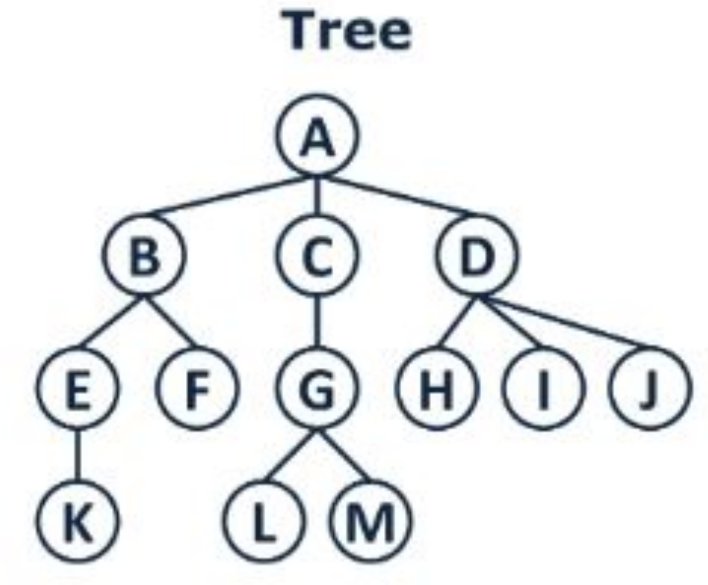


Degenerate binary tree

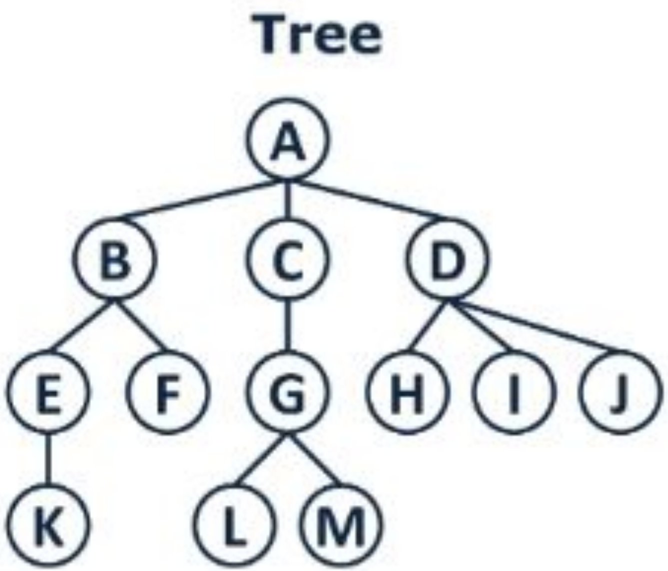
- Also known as pathological tree
- A tree where **every** parent node has only **one** child
    - Looks like a linked list.



IS THIS A BINARY TREE?



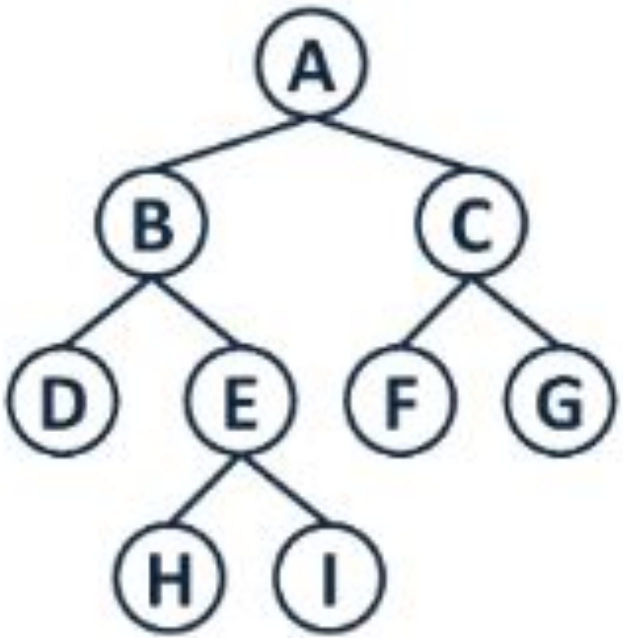
IS THIS A BINARY TREE?



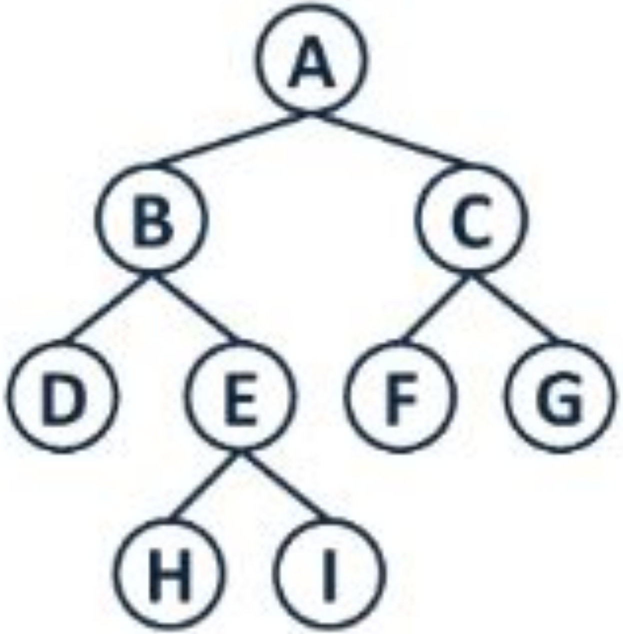
NO

CAN HAVE MAX 2 CHILDREN

IS THIS A FULL BINARY TREE?



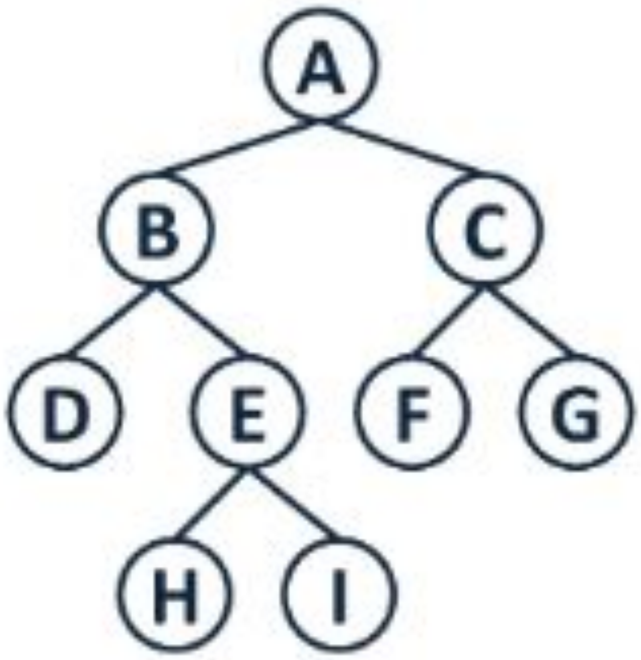
IS THIS A FULL BINARY TREE?



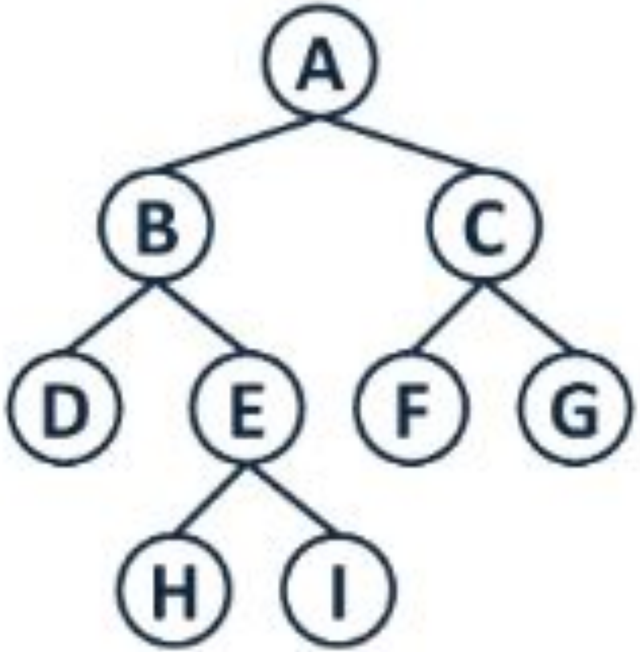
YES

NODES HAVE EITHER 0 CHILD OR 2 CHILDREN

IS THIS A PERFECT BINARY TREE?



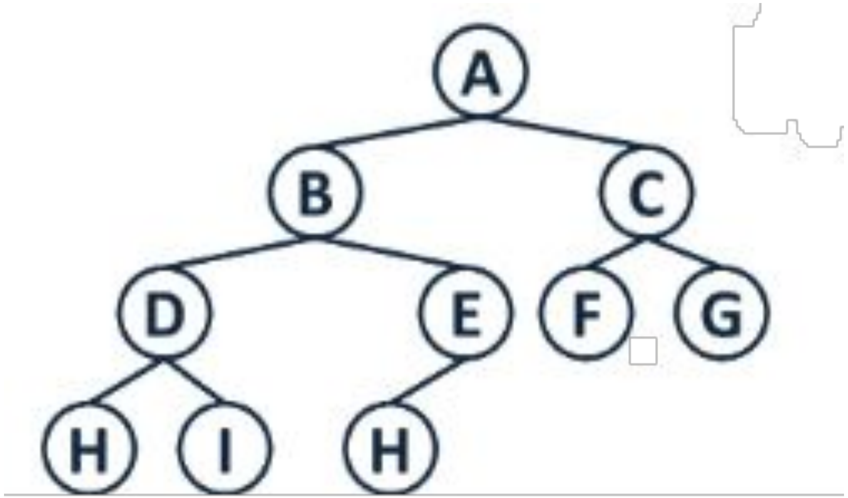
IS THIS A PERFECT BINARY TREE?



NO

D, F AND G HAS NO CHILDREN

WHAT KIND OF TREE IS THIS?

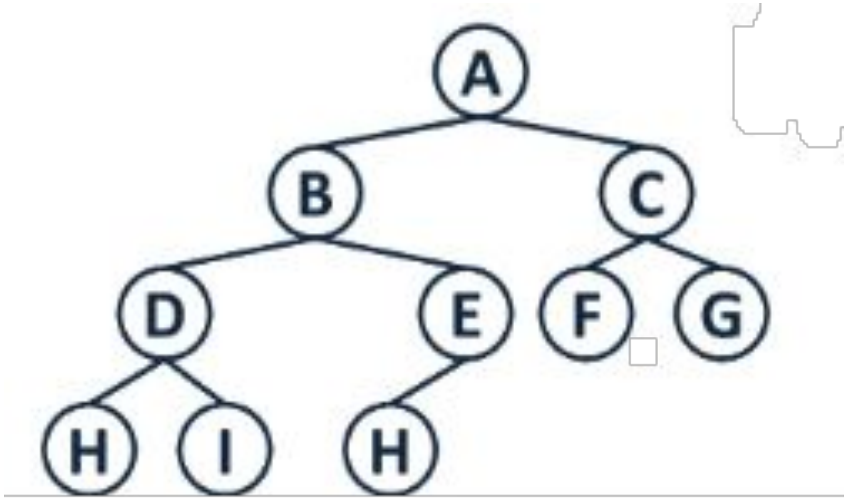


PERFECT ?

FULL ?

COMPLETE ?

WHAT KIND OF TREE IS THIS?

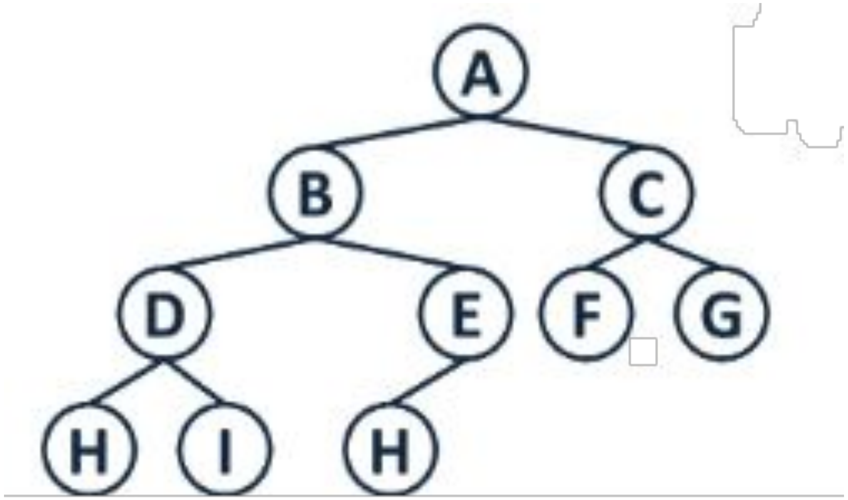


PERFECT : NO

FULL ?

COMPLETE ?

WHAT KIND OF TREE IS THIS?

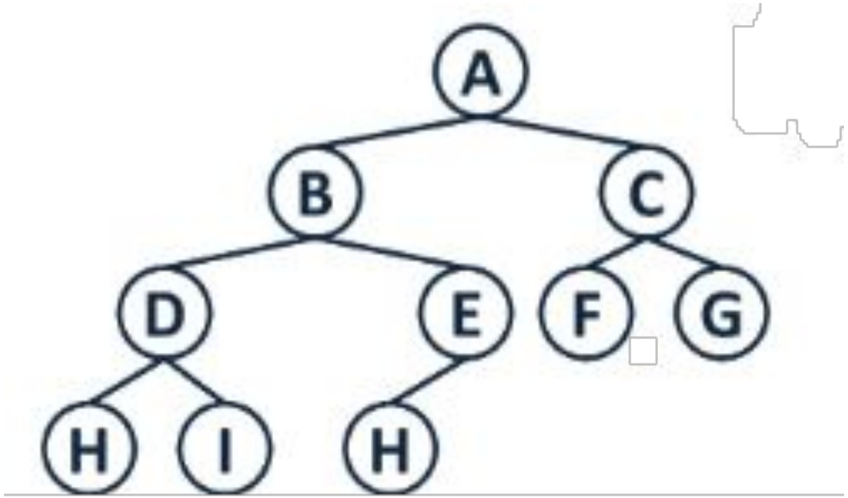


PERFECT : NO

FULL : NO

COMPLETE ?

WHAT KIND OF TREE IS THIS?



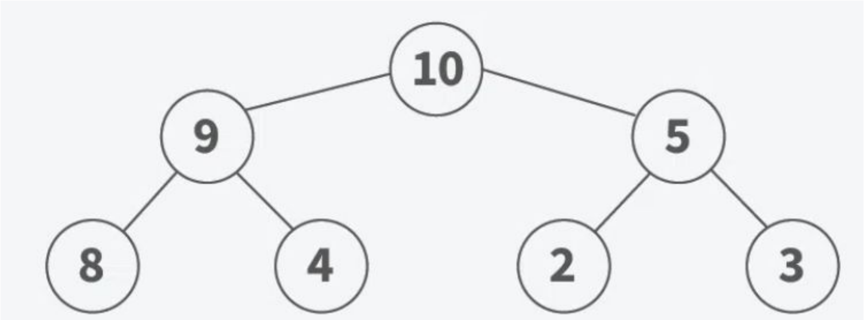
PERFECT : NO

FULL : NO

COMPLETE : YES

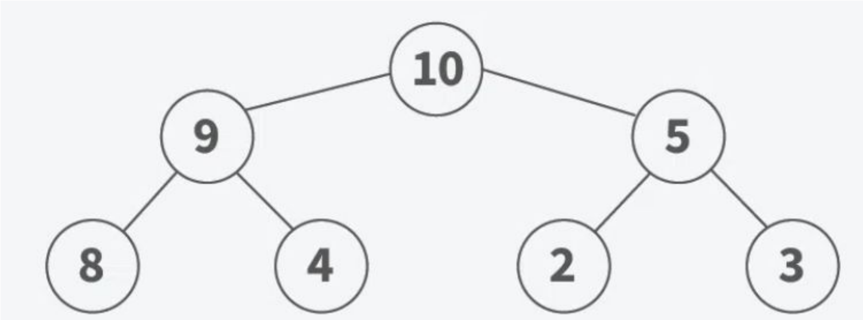
Tree representation

- A binary tree can be represented as an array.
  - Level by level
  - Give an index starting from 0 from root
    - Go from left to right for each level

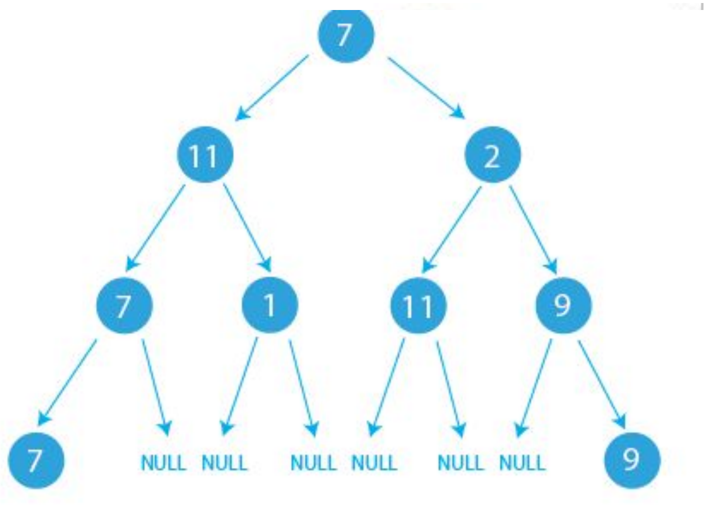


Tree representation

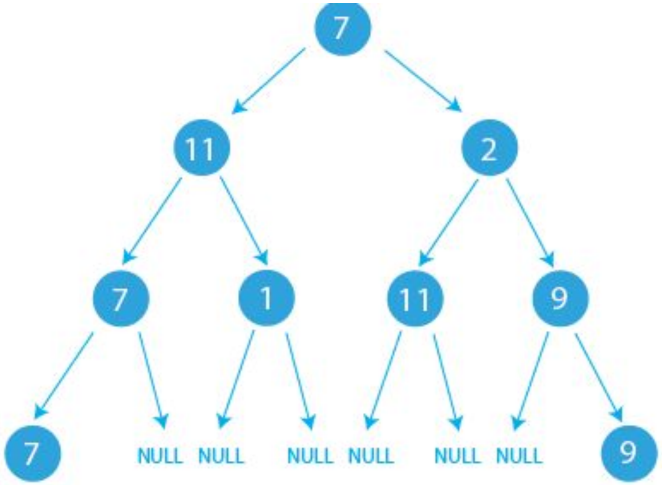
- A binary tree can be represented as an array.
  - Level by level
  - Give an index starting from 0 from root
    - Go from left to right for each level



Write this in array form



Write this in array form



Can't use *null* in Java int arrays.

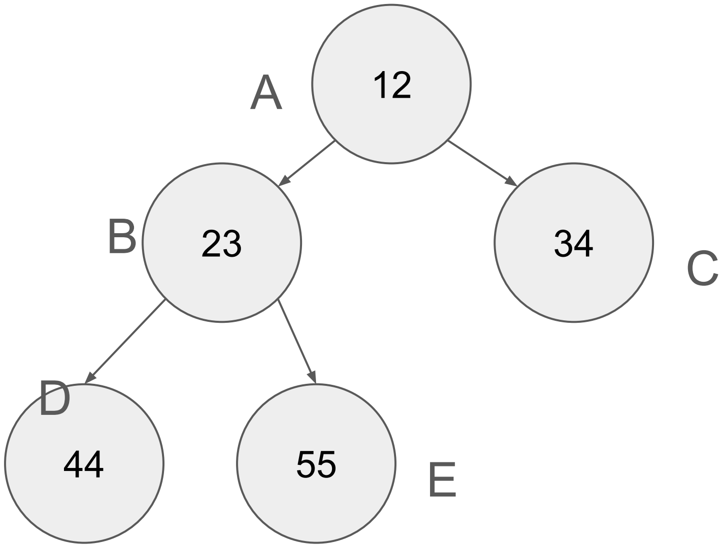
So use -1

{7,11,2,7,1,11,9,7,-1,-1,-1,-1,-1,-1,9}

Pointer (Reference) Representation

- We create a TreeNode
  - int value, Node left, Node right
- We have a root node.
  - That node has a left and right
- We can use referencing to represent a Tree.

```java
package treelife;

public class BinaryTree {
    //we need a Node class, we create it here
    static class TreeNode {
        int value;
        TreeNode left, right;

        public TreeNode(int value) {
            this.value = value;
            this.left = null;
            this.right = null;
        }
    }

    TreeNode root; //we need to have a root

    BinaryTree() {
        this.root = null;
    }

}
```



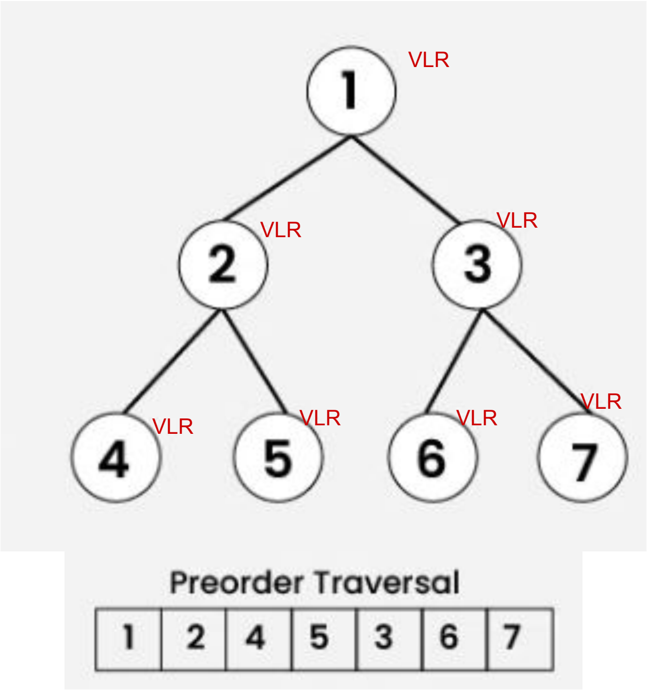

```java
public class Main {

    public static void main(String[] args) {

        BinaryTree bt = new BinaryTree();

        TreeNode A = bt.root;
        TreeNode B = A.left;
        TreeNode C = A.right;
        TreeNode D = B.left;
        TreeNode E = B.right;

        A.value = 1;
        B.value = 23;
        C.value = 34;
        D.value = 44;
        E.value = 55;
    }

}
```

tree traversals

- Breadth-First Search (BFS)
  - Level-order traversal
  - Uses **queue**
- Depth-First Search (DFS)
  - Pre-order, In-order, Post-order
  - Recursion
  - uses **Stacks**
- Applications
  - Pre-order for expression evaluation
  - In-order for bst (sorted)
  - Post-order for directory deletion, evaluation of postfix expression

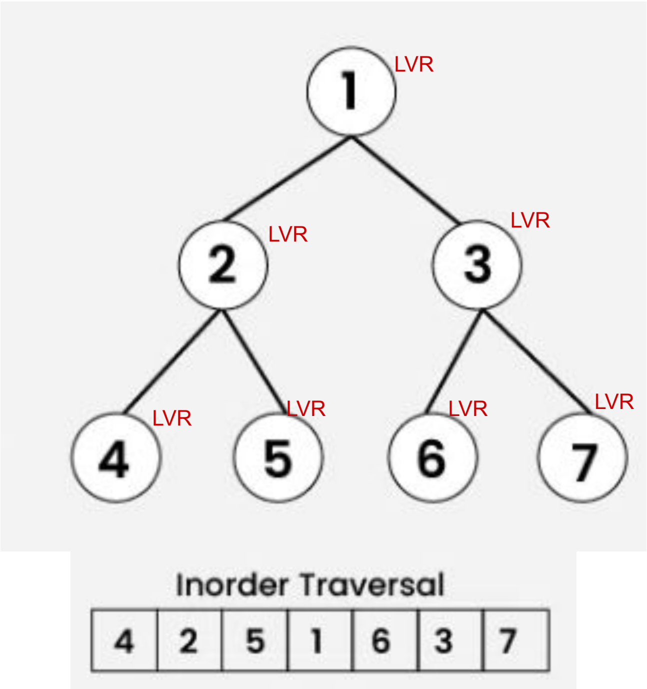
pre-order traversal

- Visit Node
- Traverse Left
- Traverse Right



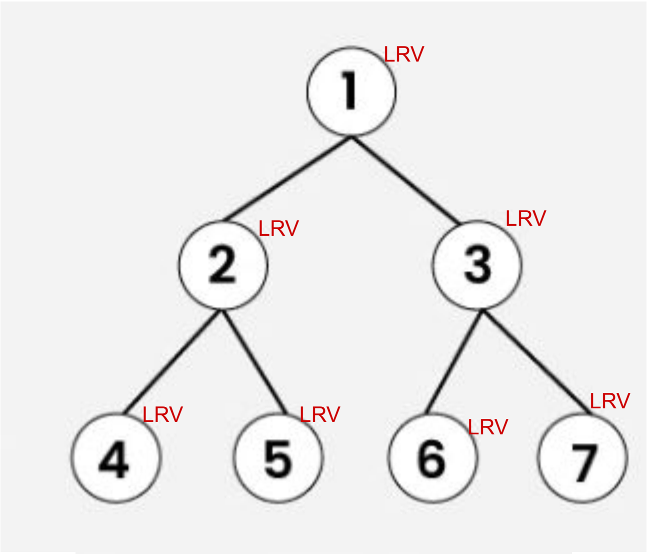
in-order traversal

- Traverse Left
- Visit Node
- Traverse Right



post-order traversal

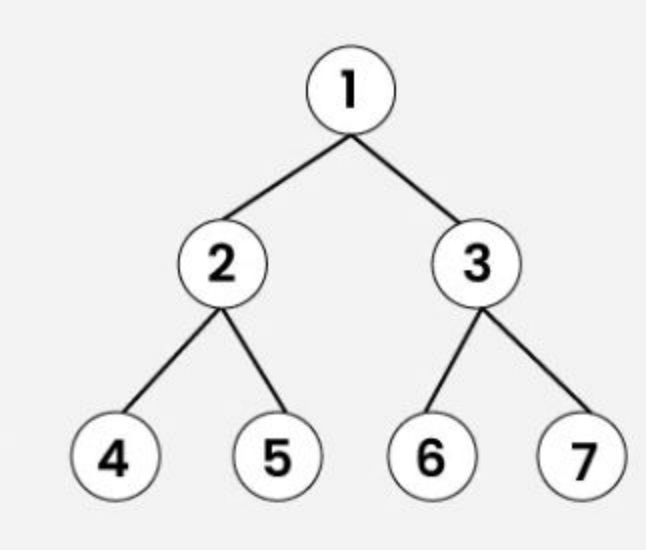
- Traverse Left
- Traverse Right
- Visit Node



4	5	2	6	7	3	1

Breadth-first traversal

- Go from left to right
- Level by level
- 1,2,3,4,5,6,7



Binary Trees (Summary)

- A tree where each node has either 0 or 2 children.
  - 2 at most.
- Can be represented with
  - Arrays (complete binary tree)
  - Pointer (reference) based -> linked structure
- Types:
  - Full binary tree
  - Complete binary tree
  - Perfect binary tree

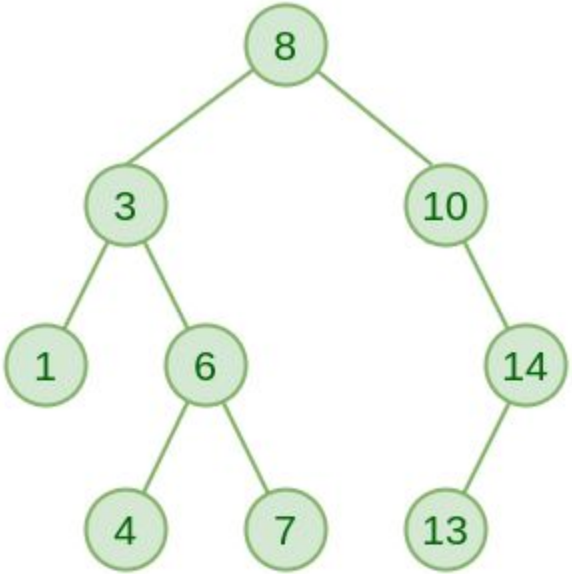
Binary Search Trees (BST)

- Special kind of binary trees
  - Left subtree < root < right subtree
- Operations
  - Search
  - Insert
  - Delete
- Applications
  - Dynamic sets
  - Searching and sorting
- Time Complexity
  - Best: O(logn)
  - Worst: O(n) - unbalanced

BST

- All BST are binary trees, but not all binary trees are BST.
  - BST is a special binary tree.
  - Maintains a specific order.

BST

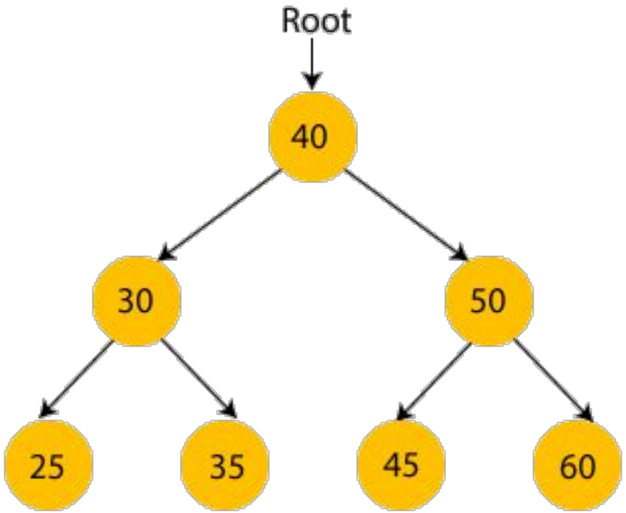


LEFT IS SMALLER THAN ROOT

ROOT IS SMALLER THAN RIGHT

THIS MUST BE TRUE FOR EVERY SUBTREE

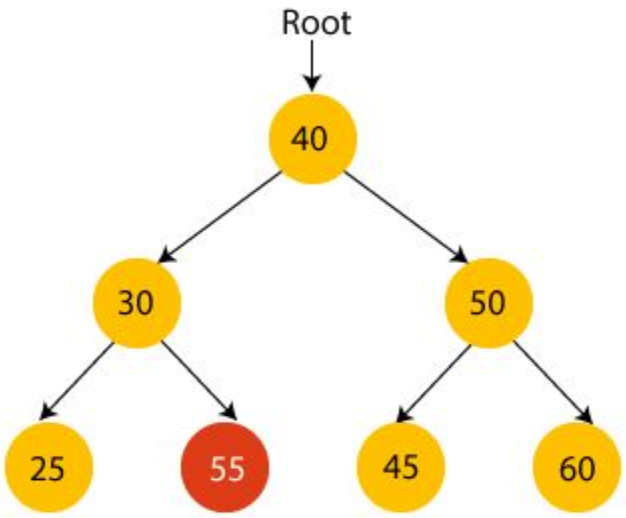
BST



Every element in the left subtree must be smaller than the root.

Every element in the right subtree **must** be larger than the root.

BST



NOT A BST

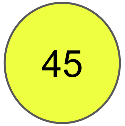
55 IS LARGER THAN 40.

CREATING A BST

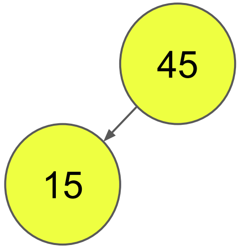
- We are going to create a BST
- 45,15,79,90,10,55,12,20,50

- We start by putting the first element as root.
- Then, we check the next element.
  - If smaller, we add it as the root of the left subtree
  - Else, we insert it as the root of right subtree.

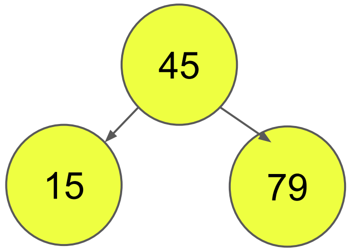
Adding the root 45,15,79,90,10,55,12,20,50



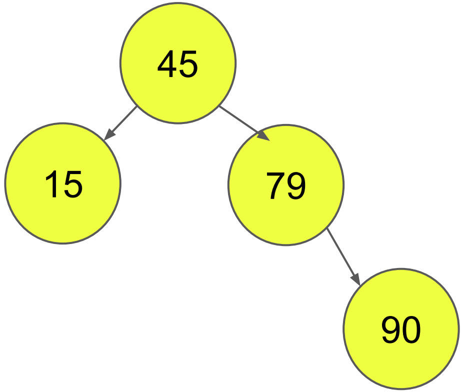
Adding 15 45,15,79,90,10,55,12,20,50



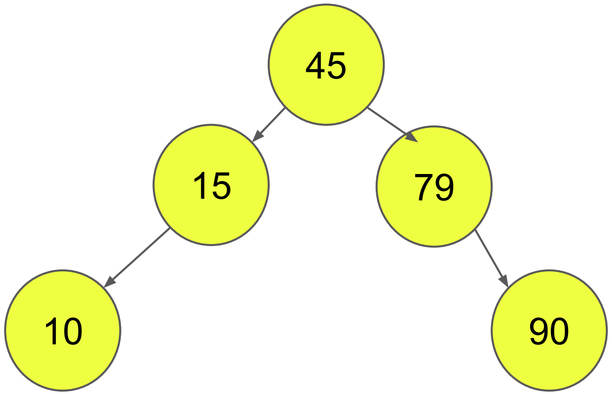
Adding 79 45,15,79,90,10,55,12,20,50



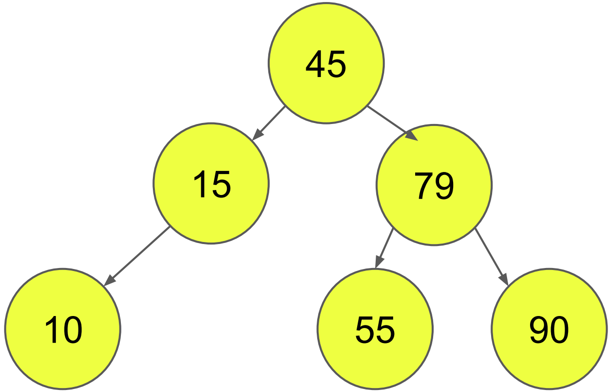
Adding 90 45,15,79,90,10,55,12,20,50



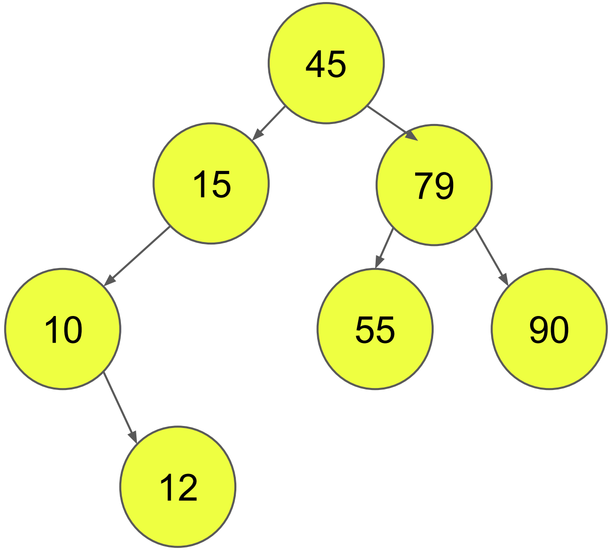
Adding 10 45,15,79,90,10,55,12,20,50



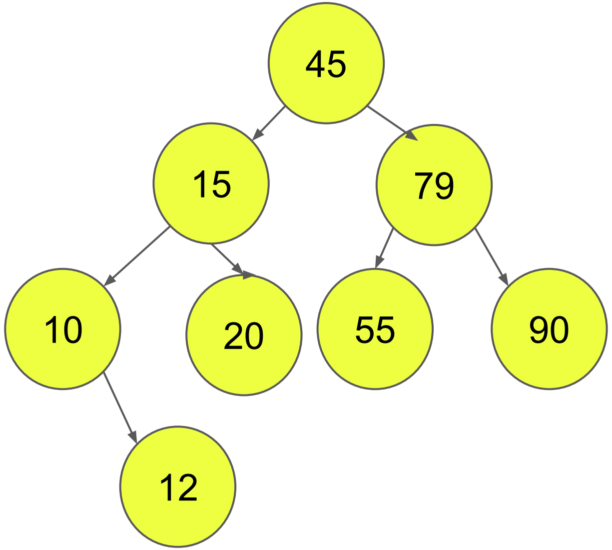
Adding 55 45,15,79,90,10,55,12,20,50



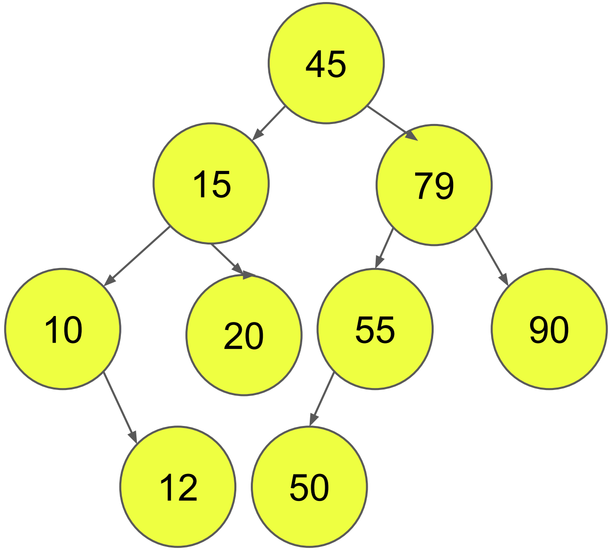
Adding 12 45,15,79,90,10,55,12,20,50



Adding 20 45,15,79,90,10,55,12,20,50



Adding 50 45,15,79,90,10,55,12,20,50

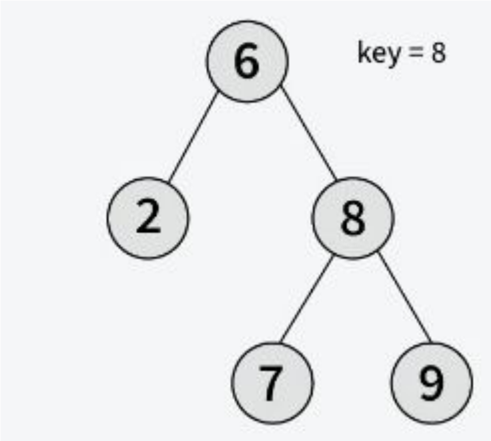


Construct a Binary tree

- 5, 10, 4, 2, 16, 7, 1, 20, 15, 3
- 8, 1, 2, 14, 12, 9, 21, 6, 10, 4
- 10, 25, 15, 6, 10, 6, 2, 15, 12, 18
  - Same elements?
  - Two options:
    - either choose where to add them
      - easier.
      - keeps the tree structure as standard BST
    - or increase the count and don't add them again
      - more compact

Operations on BST

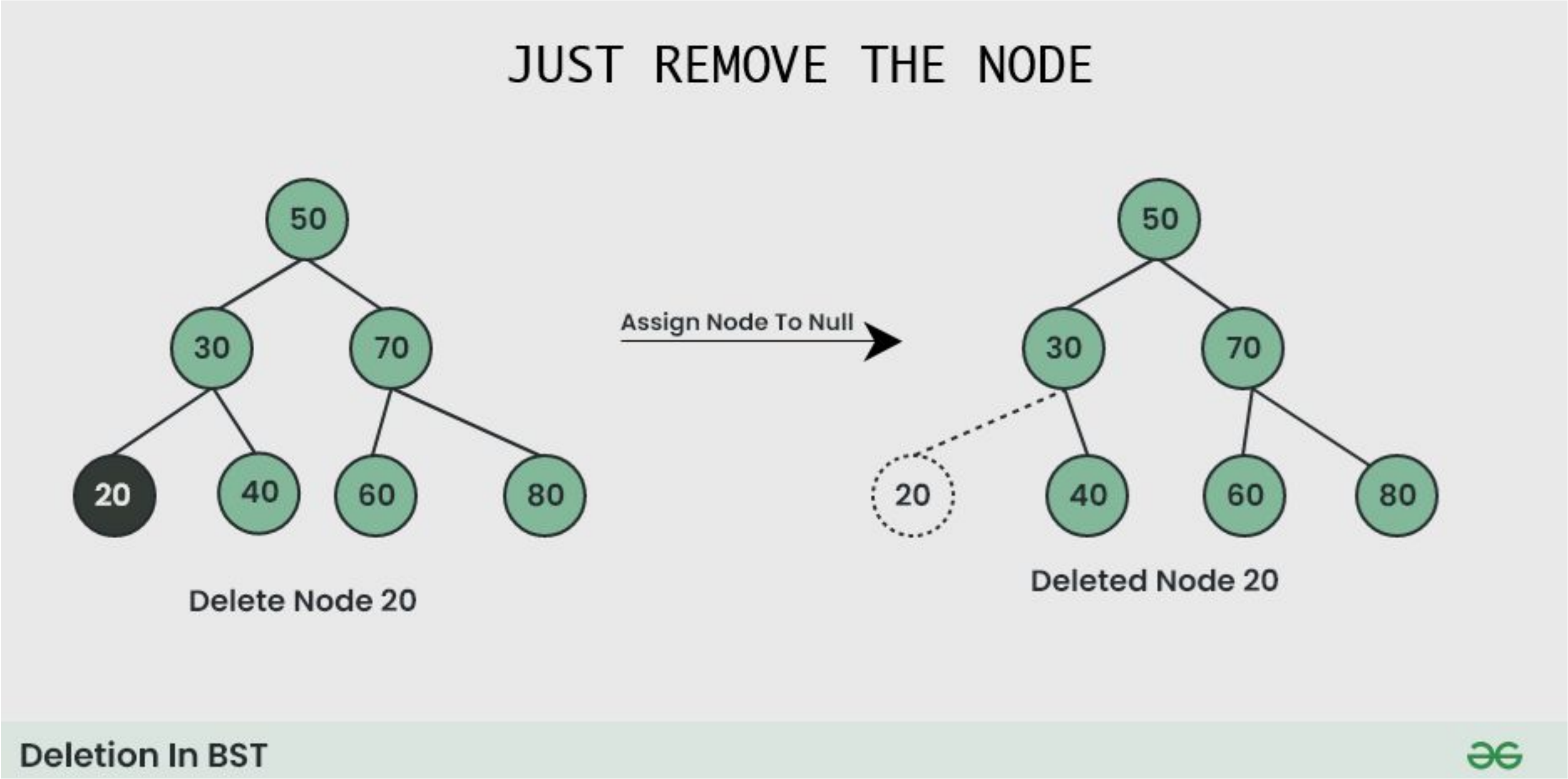
- Insertion
  - We did it before.
- Searching
  - Binary Search!
  - As we know, BST is actually sorted.
  - In-order traversal gives us the sorted result.
    - Let's try it! (LVR)
- Deletion



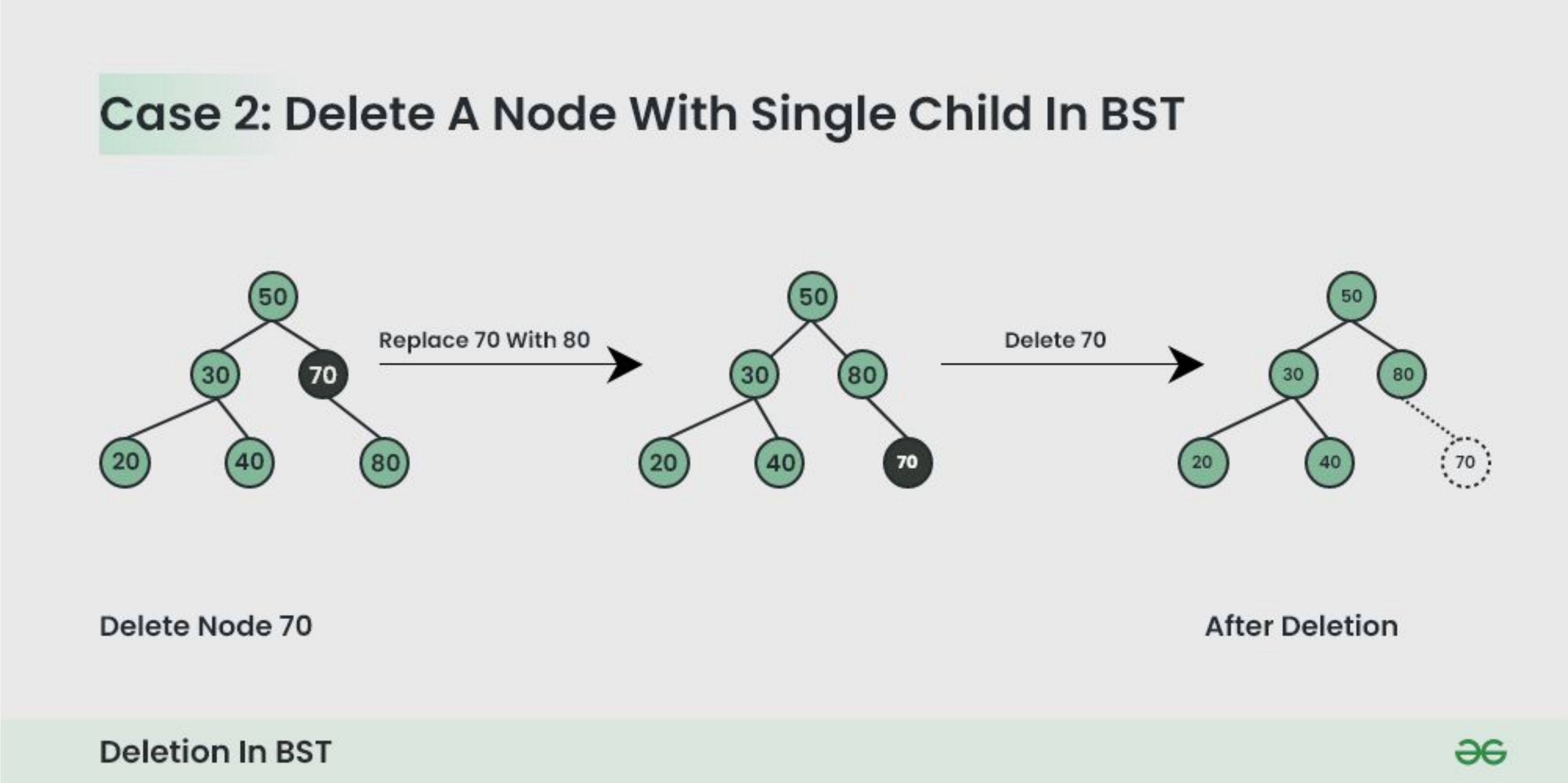
Deletion on BST

- 3 Scenarios:
  - Delete a leaf node
  - Delete a node with single child
  - Delete a node with 2 children

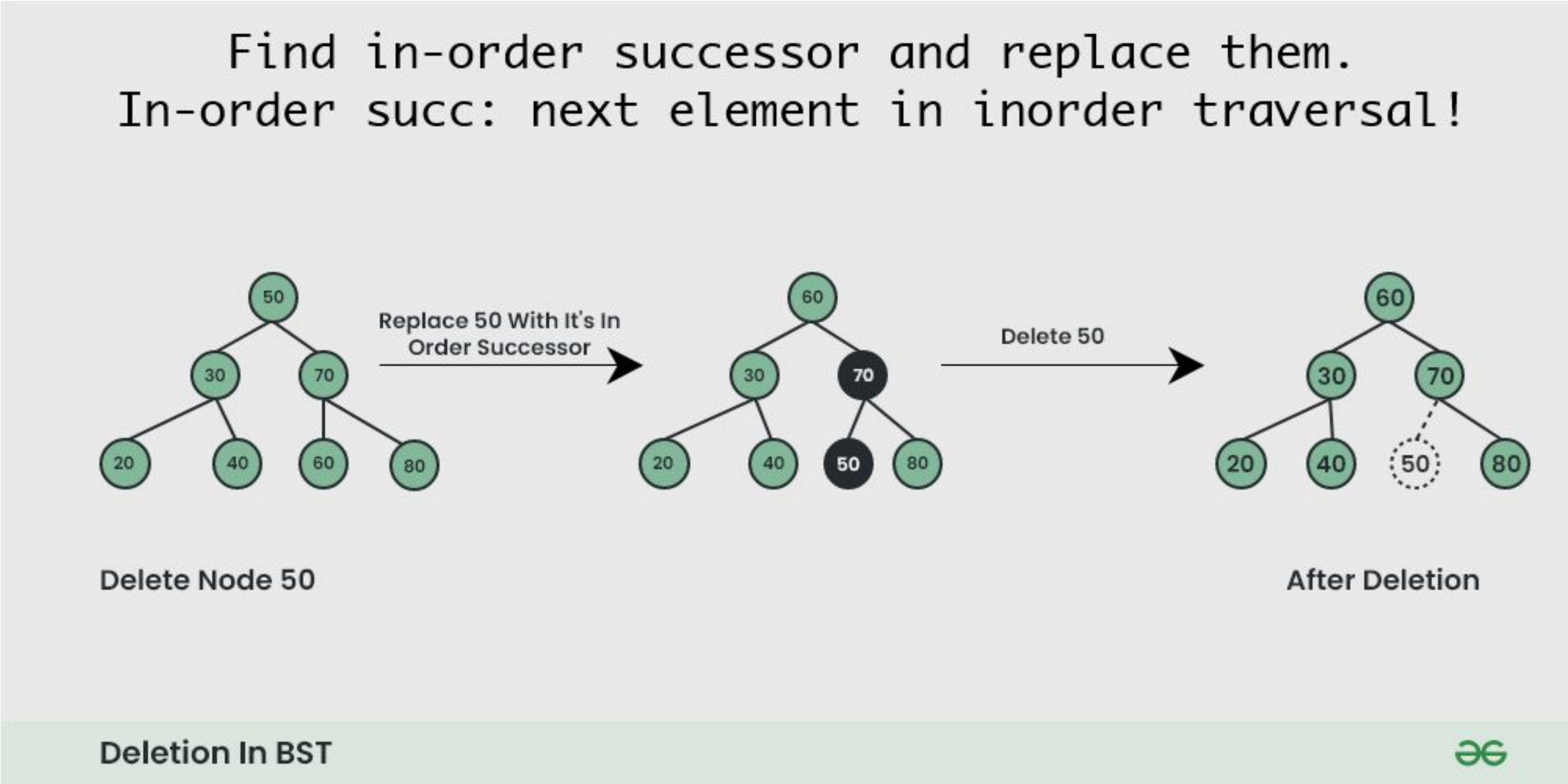
Deleting a leaf node



Deleting a node with 1 child



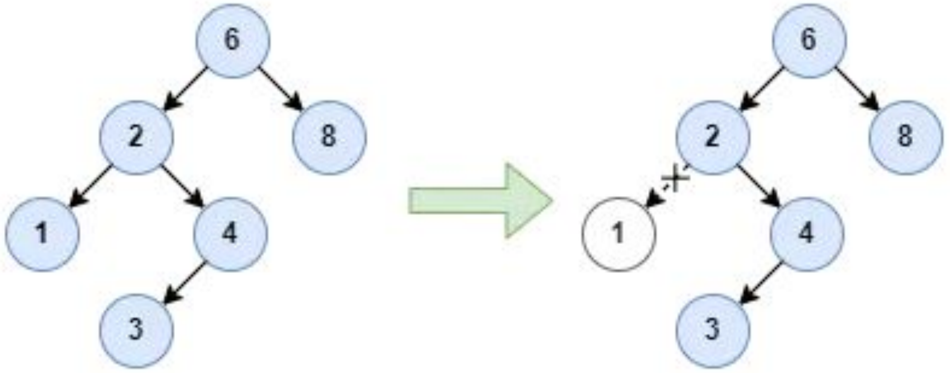
Deleting a node with 2 children



Deleting a node with 2 children

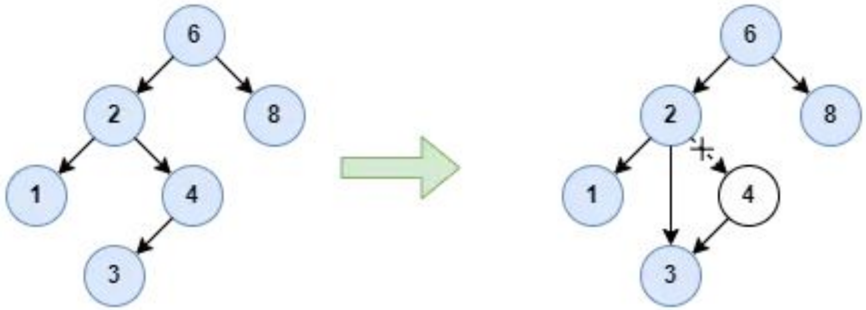
- Can either choose to use:
  - In order successor
  - In order predecessor
- In order successor
  - Node with minimum value in the right subtree
- In order predecessor
  - Node with maximum value in the left subtree
- Better performance with **successor**
  - Because it is possible predecessor also have two children.
  - Worst-case time complexity can increase to O(logn)
  - In successor, it is O(1)

Ex: Deleting a leaf node



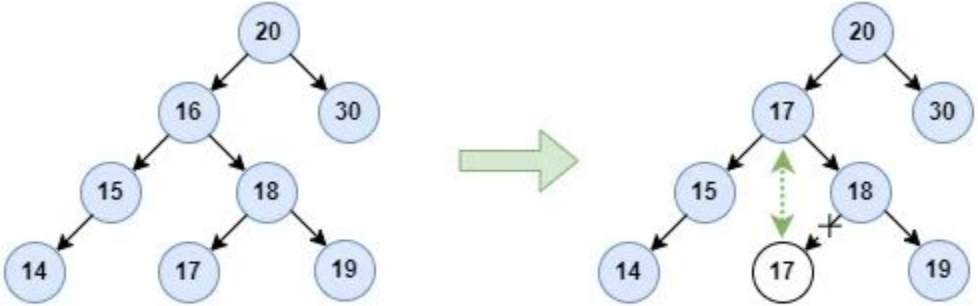
Delete leaf node

Ex: Deleting a node with 1 child



Delete a node with only one child

Ex: Deleting a node with 2 children



Traversals in BST

- In-order traversal
  - This approach results in **sorted** list.
- Pre-order traversal
  - If you want to copy your BST, use a pre-order traversal
  - Because it starts from the root and as you traverse, you can create a new BST which is the same with what you are traversing.
- Post-order traversal
  - When you want to delete your tree, use post-order traversal
  - You start from leaves and go up, getting to the root last.
  - A clean remove.

Heaps & Priority Queues

Heap

- A complete binary tree which satisfies the **heap** property
  - Max-heap or min-heap
- Applications:
  - Priority queues
  - Heap sort
- Operations
  - Insert
  - delete max-min
  - heapify
- Complexity
  - O(logn) for insert and delete

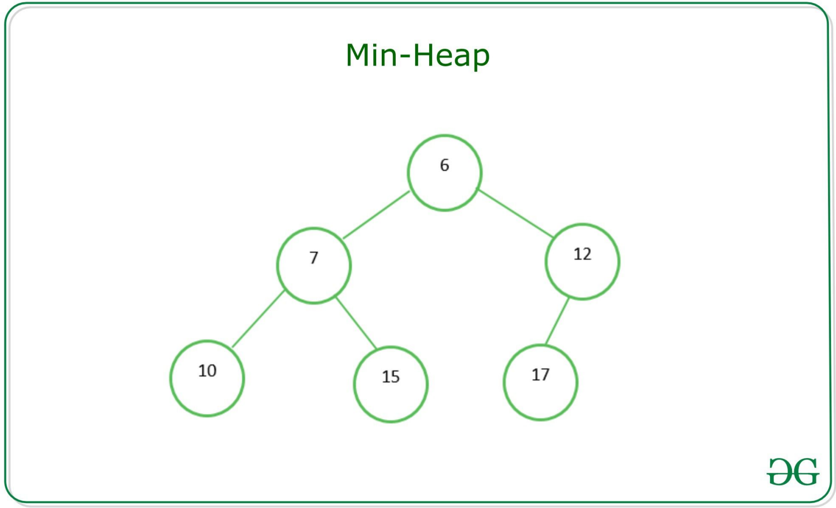
Heap

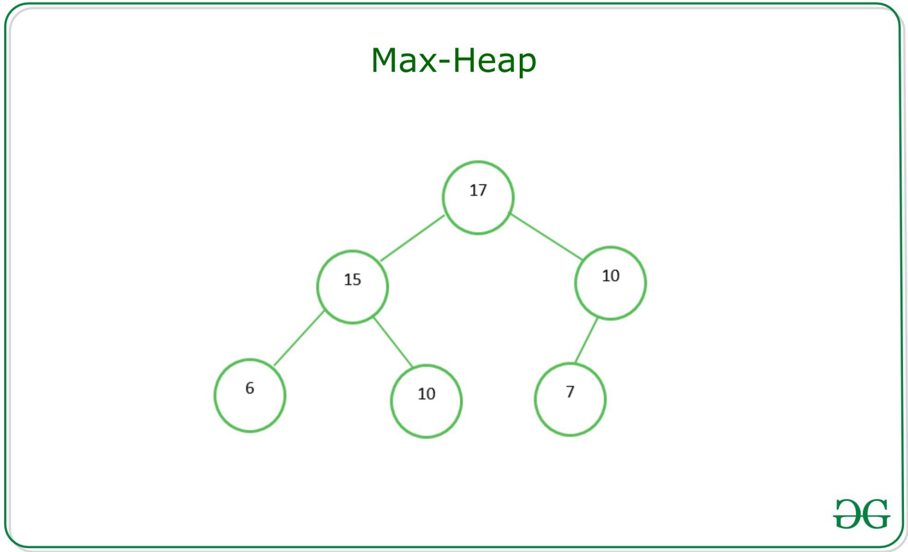
- A tree-based data structure
  - Tree is a **complete binary tree**
    - Tree is completely filled – except possible the last level which should be filled from left to right.
  - Typically represented as an array
  - Two types:
    - Min-heap
    - Max-heap
- A heap is either **min-heap** or **max-heap**
  - If a tree does not satisfy both conditions, it is **not** a heap.

Min-heap & Max-heap

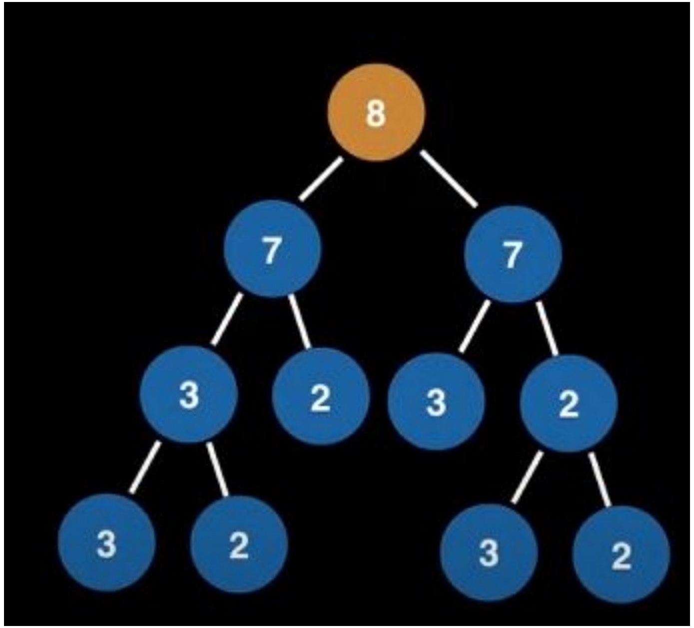
- In **min-heap**, root element must be smaller than all.
  - root is the smallest
  - Same goes for all subtrees.
- In **max-heap**, root element must be larger than all.
  - root is the largest
  - Same goes for all subtrees.

Min-heap & Max-heap

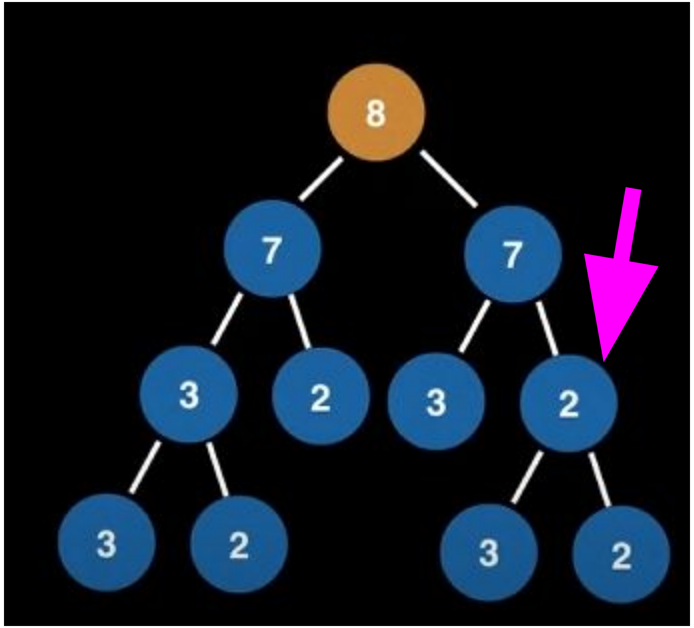




Is this a heap?



Is this a heap?



NO

Heaps

- If you want to find the smallest element or largest element
  - It is O(1)
  - Because it is at the root!
  - Good for **priority queues**
- Insertion and deletion
  - O(logn)
  - Faster than many other data structures
- Priority queues
  - Heaps are the foundation
  - Elements are processed in order of their priority
- Heap sort
  - O(nlogn)
  - Doesn’t require additional space

Priority Queue

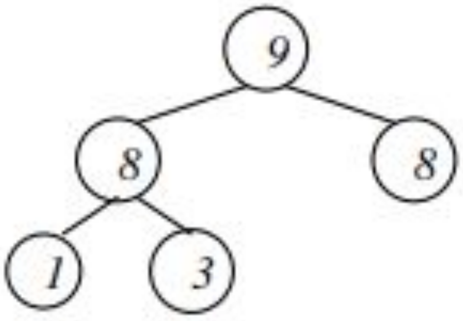
- A type of queue
    - Arranges elements based on their priority values
- Most common way to implement it is **binary heap**
- Every item has a priority
- Element with **high** priority is dequeued before an element with low priority.

- When we are representing priority queues, numbers in the nodes are representing the **priority**
  - These are **search keys** in BST.
- Good, because instead of simple FIFO, we now take out the *highest* priority.
  - Therefore, we just get the root node.

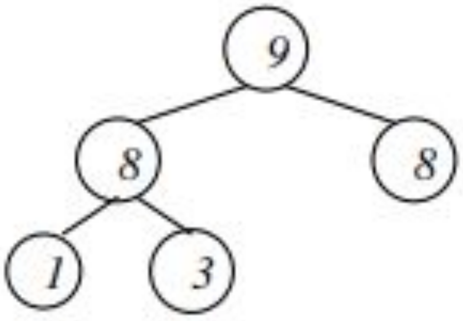
Binary heap tree

- Binary heap tree, is a **complete binary tree** which is either empty or:
  - The priority of the root is higher than (or equal to) that of its children (or vice versa for min-heap)
  - The left and right subtrees of the root are *heap trees*
- In BST, the bigger is on the right-most node.
- In BHP, the bigger element is on the **root**.

Is this a valid binary heap?

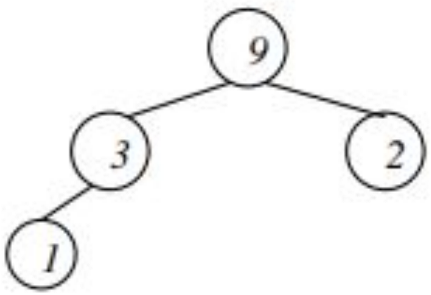


Is this a valid binary heap?



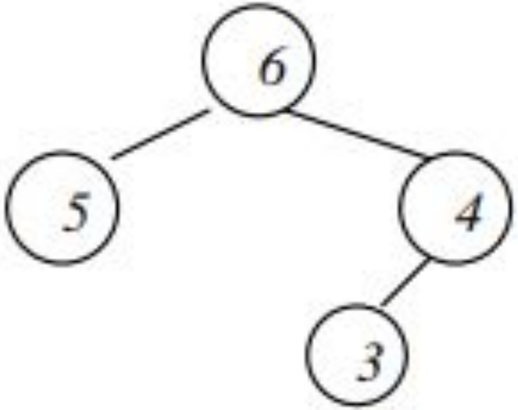
YES

Is this a valid binary heap?

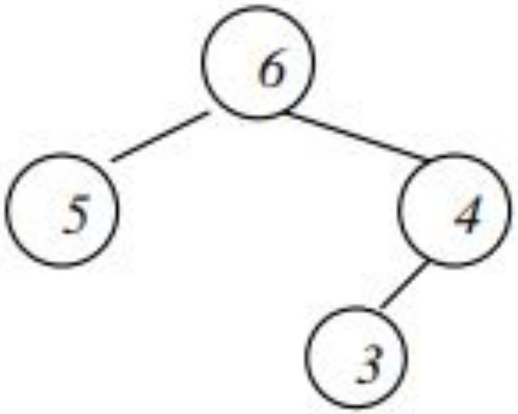


yes

Is this a valid binary heap?



Is this a valid binary heap?

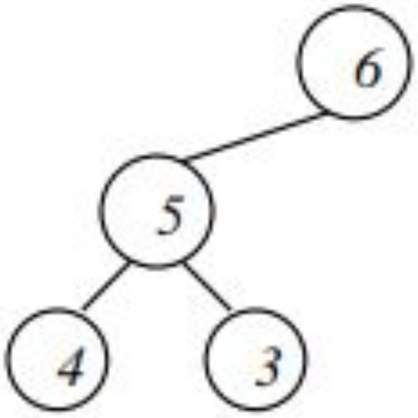


NO

NOT A COMPLETE BINARY TREE

SHOULD START FROM THE LEFT

Is this a valid binary heap?

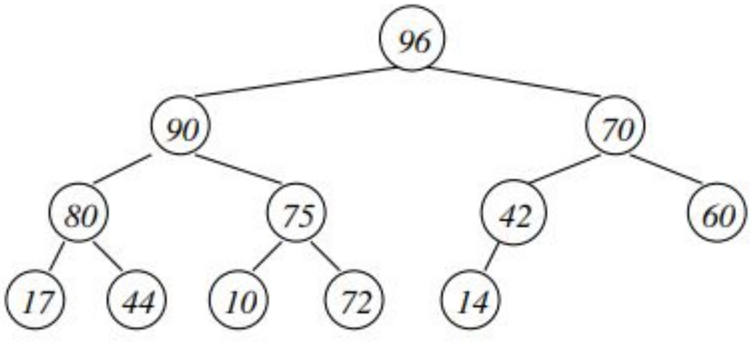


NO

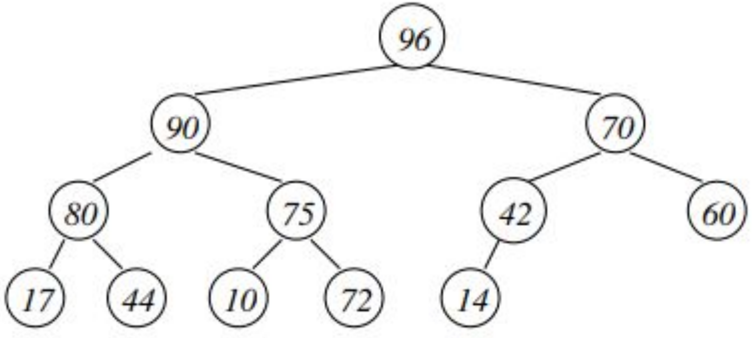
NOT A COMPLETE BINARY TREE

ALL LEVELS EXCEPT THE LAST SHOULD BE FULL

Is this a valid binary heap?

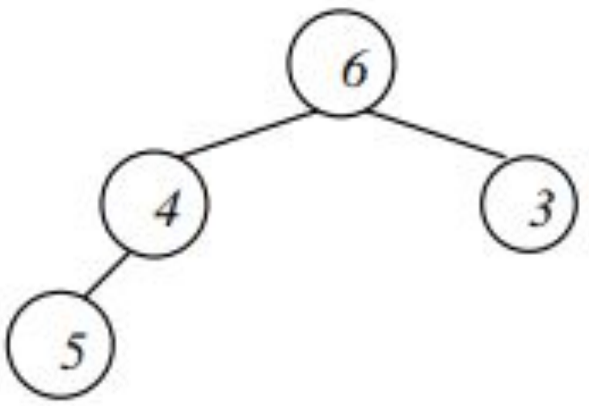


Is this a valid binary heap?



YES

Is this a valid binary heap?



NO

IS IT MIN HEAP

OR

IS IT MAX HEAP

NEITHER

heapify

- Sometimes a set of items are given to us
- We are asked to create a heap
- We will first turn the array into tree
    - Later, we are going to **heapify**