Search / Find an Element

- So far, we've learned that searching / finding a specific element in a list of `N` elements requires `O(N)` time, whether the list is stored as an array or a linked structure.

- Turns out that in the case of a **sorted array**, we can do a lot better, using an algorithm called **binary search**.

- To explain it, let's start with a simple number guessing game (this is NOT the same number guess game from Projects 2 and 3!)

Guess-a-Number Game

- The host picks a number between 1 to n (say n=1000), and asks you as the player to guess that number.
- When you make a guess, the host will tell you one of three things — your guess is 1) too large, or 2) too small, or 3) correct.
- How would you take your guesses in order to find the correct number in the fewest possible guesses?
    - Obviously if you are lucky, the first number you guess is correct. But in general you are not that lucky.

Guess-a-Number Game

- Start with the number in the middle, in our case, (1+1000) / 2 = 500. If the host says 500 is:
  - **Too large** — you know the correct number must be between 1 to 499. The next guess would be (1+499) / 2 = 250.
  - **Too small** — you know the correct number must be between 501 to 1000. The next guess would be (501+1000) / 2 = 750.
  - **Correct** — great!
- How many guesses do you have to make in the worst case?

Guess-a-Number Game

- Each guess successively **halves** the range of possible values. Eventually (in the worst case) the range narrows down to only one number, and that must be the answer.

- Even in the worst case, this will take no more than `ceiling(log`$_2$`1000) = 10` steps.

- In general, this is a logarithmic time `O(log N)`, which is enormously better than a linear time algorithm `O(N)` for a sufficiently large N.

Binary Search

Problem Statement: given a **sorted array** of elements and a target element, find if the target exists in the array and return its index (or -1 if it doesn't exist). Example:

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

Idx: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Task: find target=41.

Using linear search, it requires 13 steps.

Show how binary search works. How many steps?

Hint (u+l)/2 finds the middle, but don't include the middle when you search again

Find target=31

Index:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	5	7	11	13	17	**19**	23	29	31	37	41	43	47

Find target=31

Index:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	5	7	11	13	17	**19**	23	29	31	37	41	43	47
2	3	5	7	11	13	17	19	23	29	31	**37**	41	43	47

Find target=31

Index:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	5	7	11	13	17	**19**	23	29	31	37	41	43	47
2	3	5	7	11	13	17	19	23	29	31	**37**	41	43	47
2	3	5	7	11	13	17	19	23	**29**	31	37	41	43	47

Find target=31

Index:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	5	7	11	13	17	**19**	23	29	31	37	41	43	47
2	3	5	7	11	13	17	19	23	29	31	**37**	41	43	47
2	3	5	7	11	13	17	19	23	**29**	31	37	41	43	47
2	3	5	7	11	13	17	19	23	29	**31**	37	41	43	47

Binary Search

Problem Statement: given a **sorted array** of elements and a target element, find if the target exists in the array and return its index (or -1 if it doesn't exist). Example:

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

Idx: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

What if we are to find target=42 (non-existent)?

Using linear search, it requires 15 steps.

Using binary search, it requires only 4 steps.

Binary Search

```
protected int find (T target) {
  int lower = 0, upper = numElements-1;
  while (lower <= upper) {
    int curr = (lower + upper) / 2; // rounds down
    int result = target.compareTo(list[curr]);
    if (result == 0)
      return curr;
    else if (result < 0)
      upper = curr - 1;
    else
      lower = curr + 1;
  }
  return -1;
}
```

Binary Search

```
protected int find (T target) {
  int lower = 0, upper = numElements-1;
  while (lower <= upper) {
    int curr = (lower + upper) / 2; // rounds down
    int result = target.compareTo(list[curr]);
    if (result == 0)
      return curr;
    else if (result < 0)
      upper = curr - 1;
    else
      lower = curr + 1;
  }
  return -1;
}
```

The highlighted lines are important parts of binary search that are easy to make mistakes on.

Clicker Question #1

```
protected int find (T target) {
  int lower=0, upper=numElements-1;
  while (lower <= upper) {
    int curr=(lower+upper)/2;
    int result=target.
            compareTo(list[curr]);
    if (result == 0)
      return curr;
    else if (result < 0)
      upper = curr;
    else
      lower = curr+1;
  }
  return -1;
}
```

What happens if the marked line is changed to `upper = curr` instead of `curr-1`?

a) When element is found, the returned index may be wrong.
b) it may throw a `NullPointerException`
c) it may fail to find an existing element.
d) the loop may run forever.
e) it may throw an `Index OutofBoundException`

Another One

Clicker Question #2

```
protected int find (T target) {
  int lower=0, upper=numElements-1;
  while (lower < upper) {
    int curr=(lower+upper)/2;
    int result=target.
            compareTo(list[curr]);
    if (result == 0)
      return curr;
    else if (result < 0)
      upper = curr - 1;
    else
      lower = curr + 1;
  }
  return -1;
}
```

What happens if the <= in the while loop condition is changed to <?

a) When element is found, the returned index may be wrong.
b) the loop may run forever.
c) it may fail to find an existing element.
d) it may throw a `NullPointerException`
e) it may throw an `Index OutofBoundException`

Binary Search — Recursive Version

```
protected int recFind(T target,
                      int lower, int upper) {
  if (lower > upper)
     return -1;
  int curr = (lower + upper) / 2;
  int result = target.compareTo (list[curr]);
  if (result == 0)
     return curr;
  else if (result < 0)
    return recFind (target, lower, curr - 1);
  else return recFind (target, curr + 1, upper);
}

protected int find (T target) {
  return recFind (target, 0, numElements-1);
}
```

Binary Search

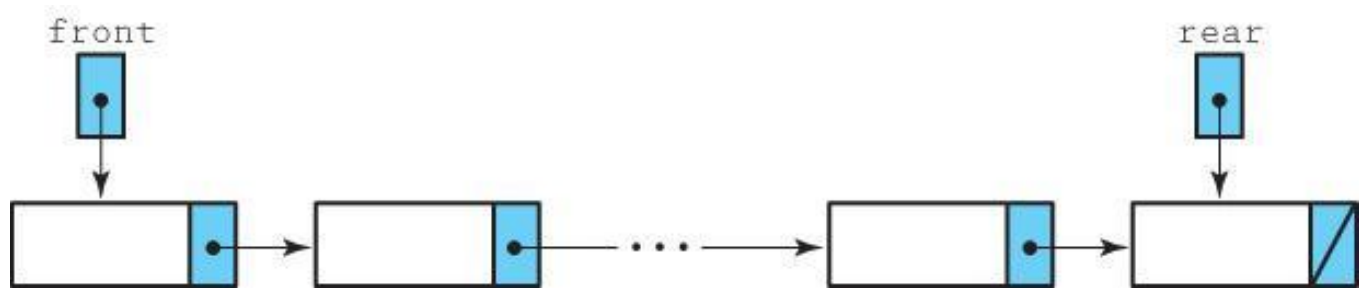
- For a sorted array with N elements, binary search is guaranteed to finish within `O(log N)` time. This is a big win for large arrays. For example, how big is the difference for N=1,000 or even 1,000,000?
- Is there any downside? What's the tradeoff?

The array must be sorted. So insertion is more expensive: `O(N)` (compared to `O(1)` for unsorted).

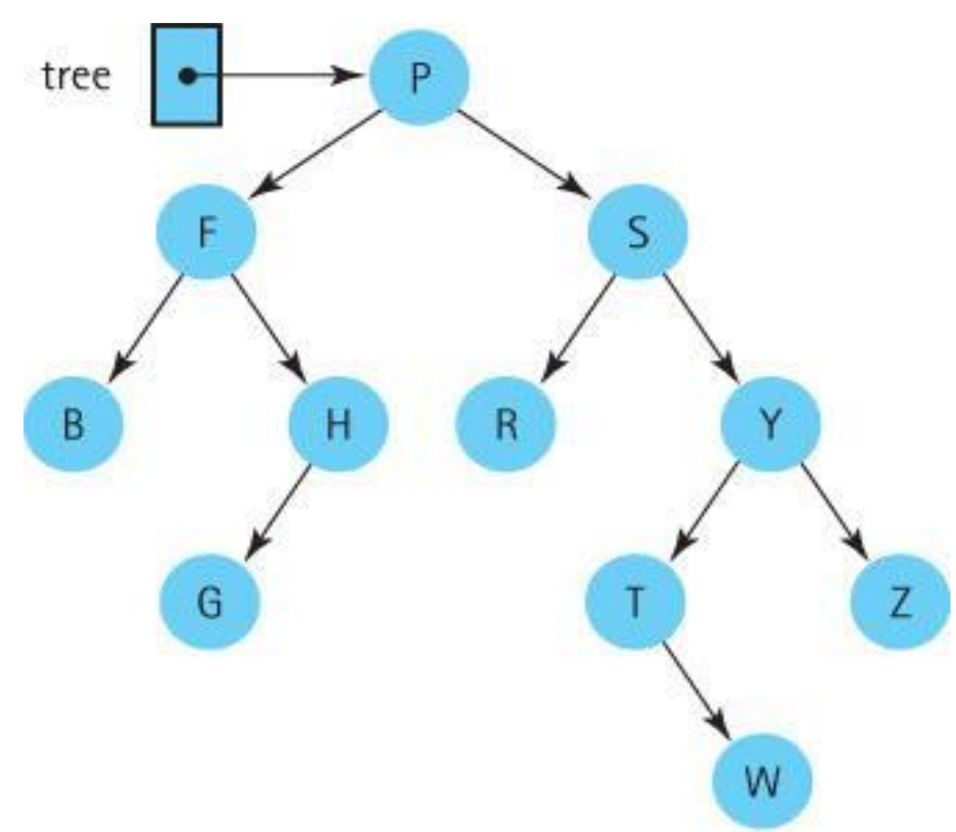
It does not work on a linked structure as there is no simple way to index a linked element in `O(1)` time.

The Tree Data Structure

- A **linked list** is a linear structure in which each element has one “successor”.



- A **tree** is a more generalized structure in which which each element may have many “successors” (i.e. children).



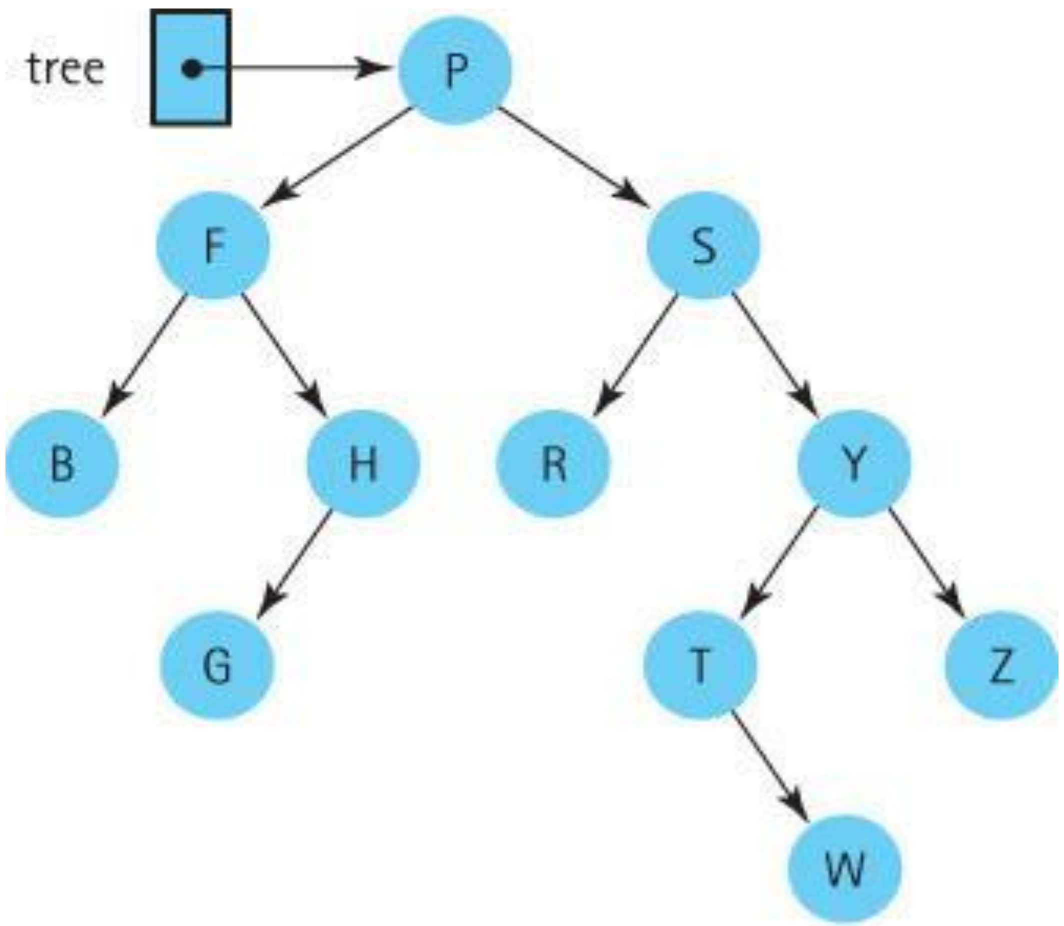
The Tree Data Structure

- A tree has a top node (**root node**), followed by its children, and the children of children…
- It actually looks like reversed from real trees…

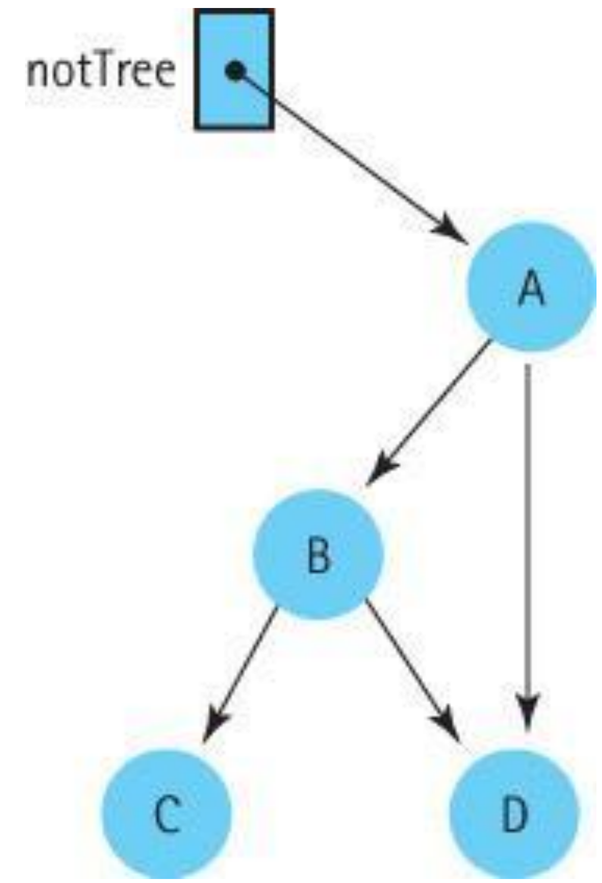
The Tree Data Structure

- Mathematically speaking, trees are **connected, acyclic graphs** (i.e. no loops).
    - There is one unique root
    - From root to any node there is **one and only one** path.
- It's very useful for representing hierarchical structures, such as file systems, Java's classes and inheritance relationships between classes.
- Here we will focus on **binary** trees, where each node has **at most two children**.

Tree

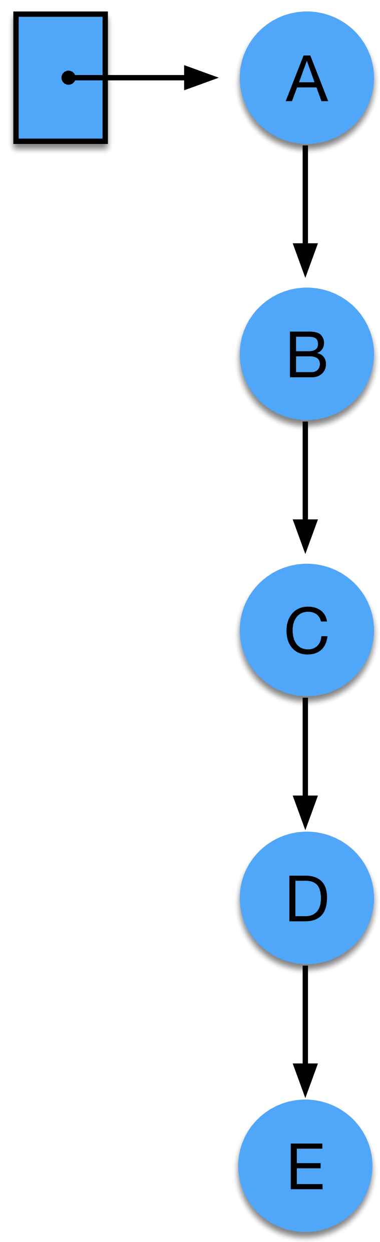


Not-tree



Tree		Not-tree
✓	• Unique root	✓
✓	• Unique path from root to any node.	✗

Is a Linked List a Tree ?

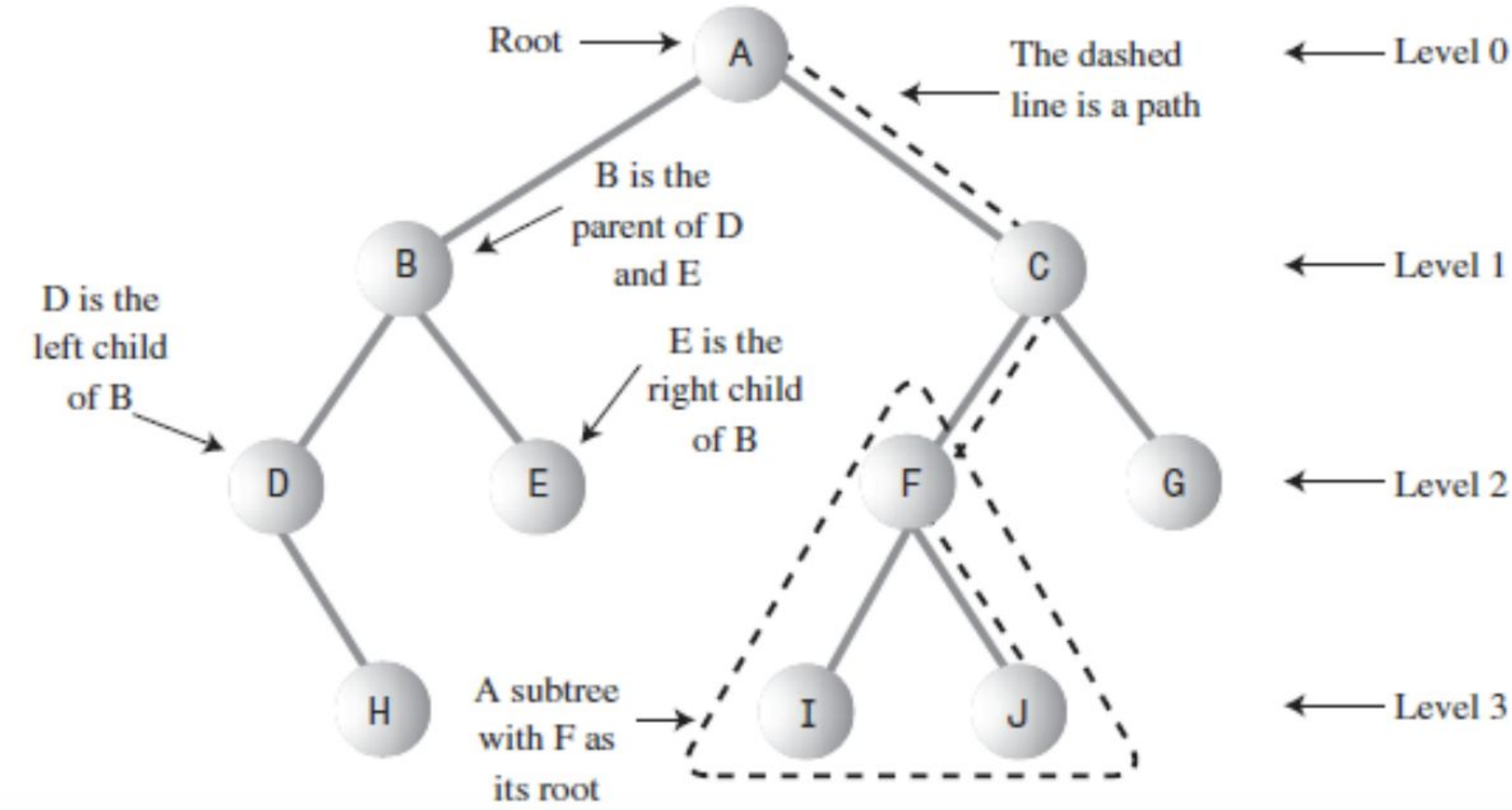


✓ • Unique root

✓ • Unique path from root to any node.

Yes, it's a tree!

Tree Terminology



Tree Terminology

- **Root**: the starting node at the top. There is only one root.
- **Parent (predecessor)**: the node that points to the current node. Any node, except the root, has 1 and only 1 parent.
- **Child (successor)**: nodes pointed to by the current node. For a binary tree, we say left child and right child.
- **Leaf**: a node with no children. There may be many leaves in a tree. Note that the root may be a leaf! How?
- **Interior node**: non-leaf node. An interior node has at least one child.

Tree Terminology

- **Path**: the sequence of nodes visited by traveling from the root to a particular node.
  - Each path is unique (due to tree being acyclic)
- **Ancestor**: any node on the path from the root to the current node.
- **Descendant**: any node whose path from the root contains the current node.
- **Subtree**: any node may be considered the root of a subtree, which consists of all descendants of this node.

More Tree Terminology

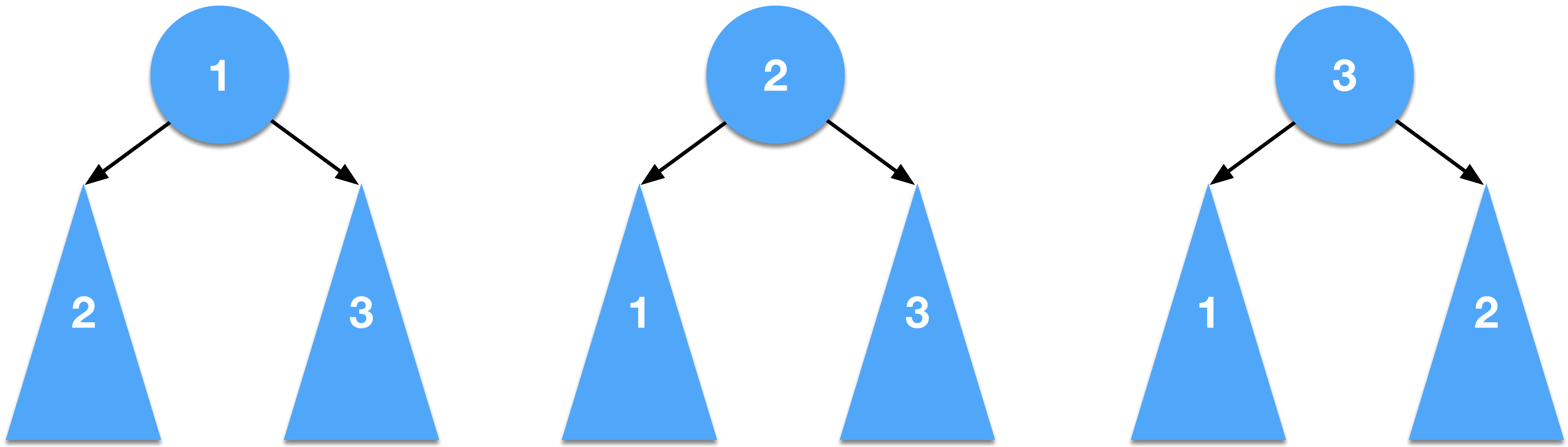
- **Level**: the path length from the root to the current node.
  - Go back 3 slides to check the example.
  - Recall that each path is unique, hence level is unique.
  - Root is at level 0.
- **Height**: the maximum level in a tree.
  - For a reasonably balanced tree with N nodes, the height is `O(log N)`. This will become obvious later.
  - What's the maximum possible height of a tree of N nodes?   **—> N-1**

Traversing a Binary Tree

- Traversing means visiting all nodes in the tree in a specific order. While the traversal order is obvious for a linked list, for trees there are 3 common methods, distinguished by the ***order in which the current node is visited*** during the recursive traversal:
  - **Pre-order traversal**: visit the ***current*** node, visit the left subtree, then visit the right subtree.
  - **In-order traversal**: visit the left subtree, visit the ***current*** node, then visit the right subtree.
  - **Post-order traversal**: visit the left subtree, visit the right subtree, then visit the ***current*** node.

Traversing a Binary Tree

- Comparing the tree traversal methods:



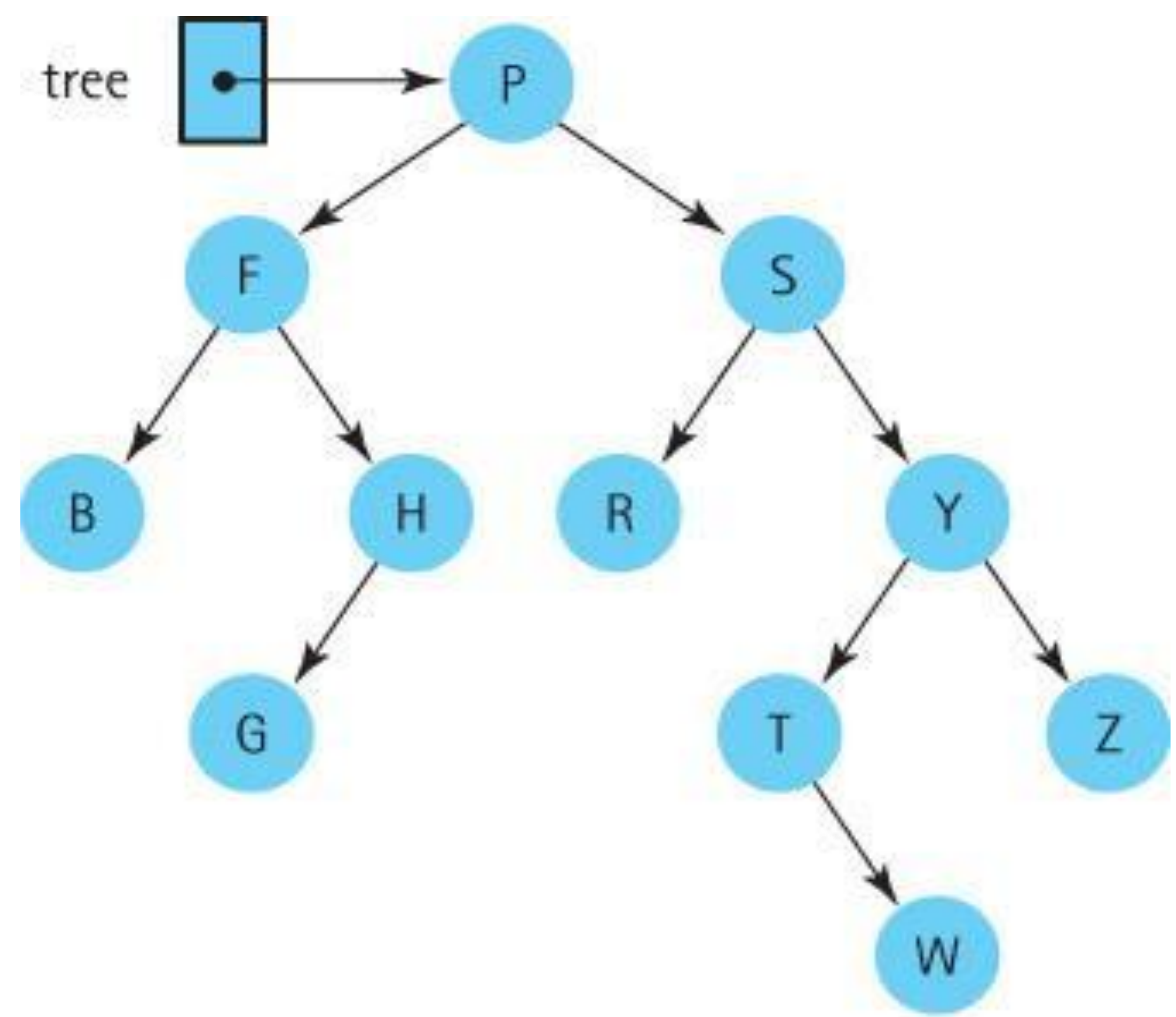
pre-order **in-order** **post-order**

(The numbers above refer to the order of traversal.)

- The subtrees are traversed **recursively**!

Tree Traversal Examples

- Pre-Order:
  - P F B H G S R Y T W Z
- In-Order:
  - B F G H P R S T W Y Z
- Post-Order:
  - ?



Clicker Question #3

What’s the post-order traversal result of this tree?

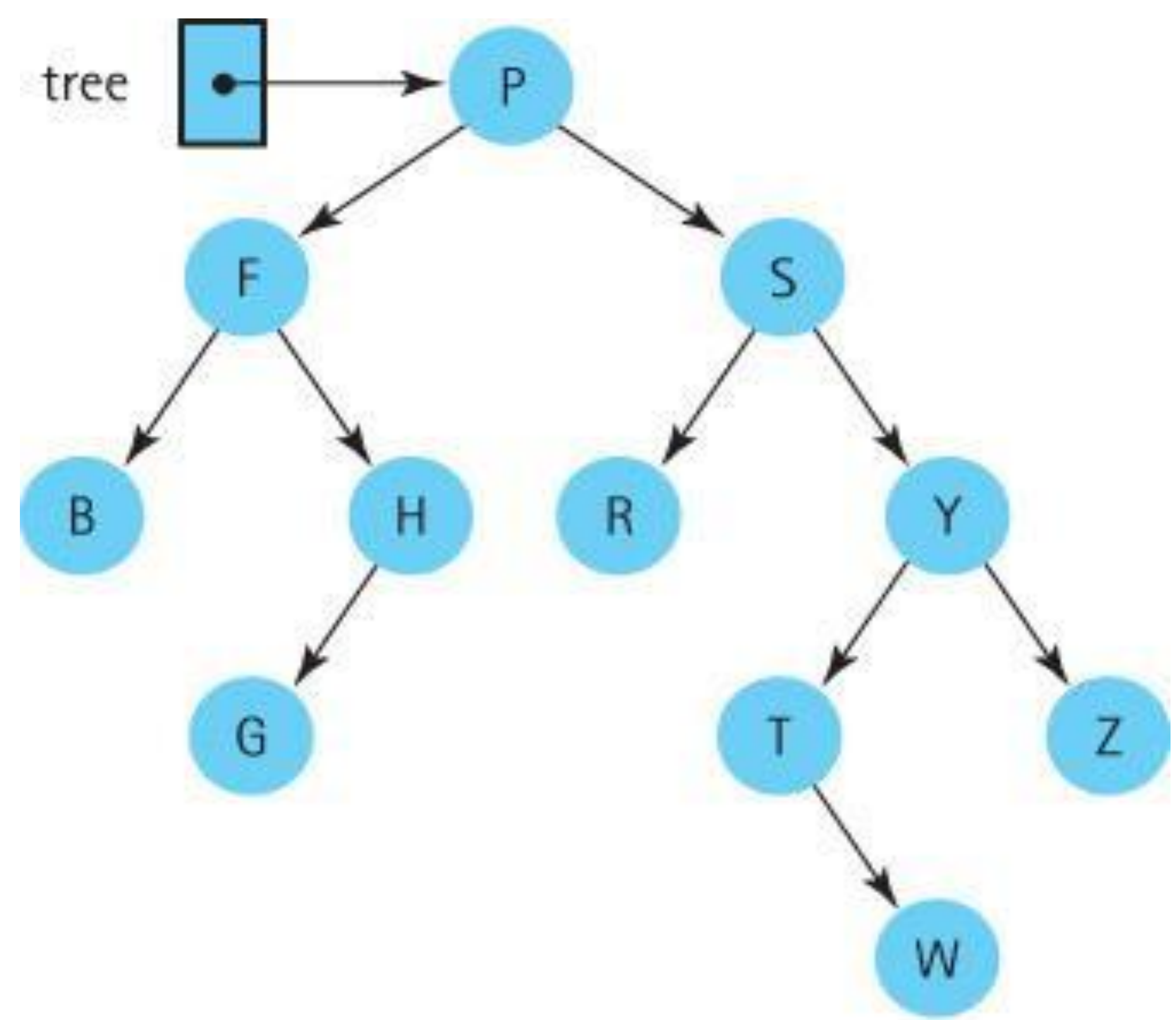
(a) B G H F P R S W T Z Y

(b) B H G F W T Z Y R S P

(c) F S P B H G R Y T W Z

(d) F B G H R W T Z Y S P

(e) B G H F R W T Z Y S P



Recursive Traversals of Trees

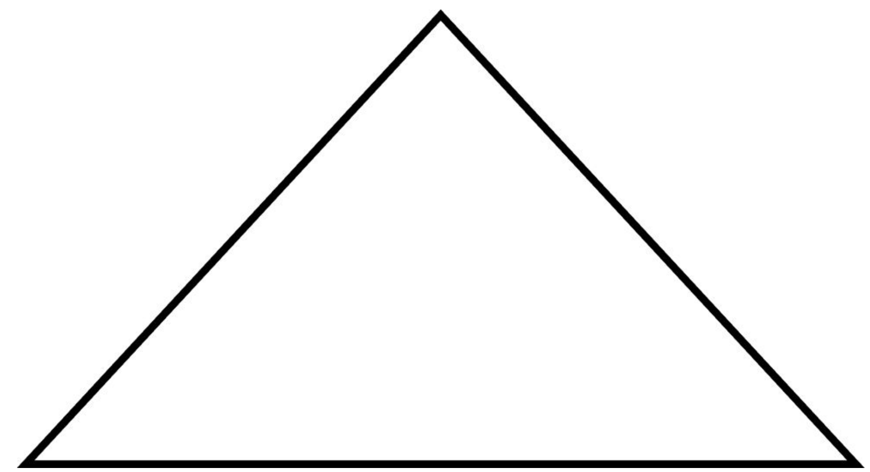
```
public void preOrder(TreeNode x) {
  if (x != null) {
    // visit by printing the value
   System.out.println(x.getInfo());
   preOrder(x.getLeft());
   preOrder(x.getRight());
  }
}
```

How are in-order and post-order traversals different?

More Terminology

- **Full Binary Tree**: A binary tree in which all of the leaves are on the same level and every non-leaf node has two children.

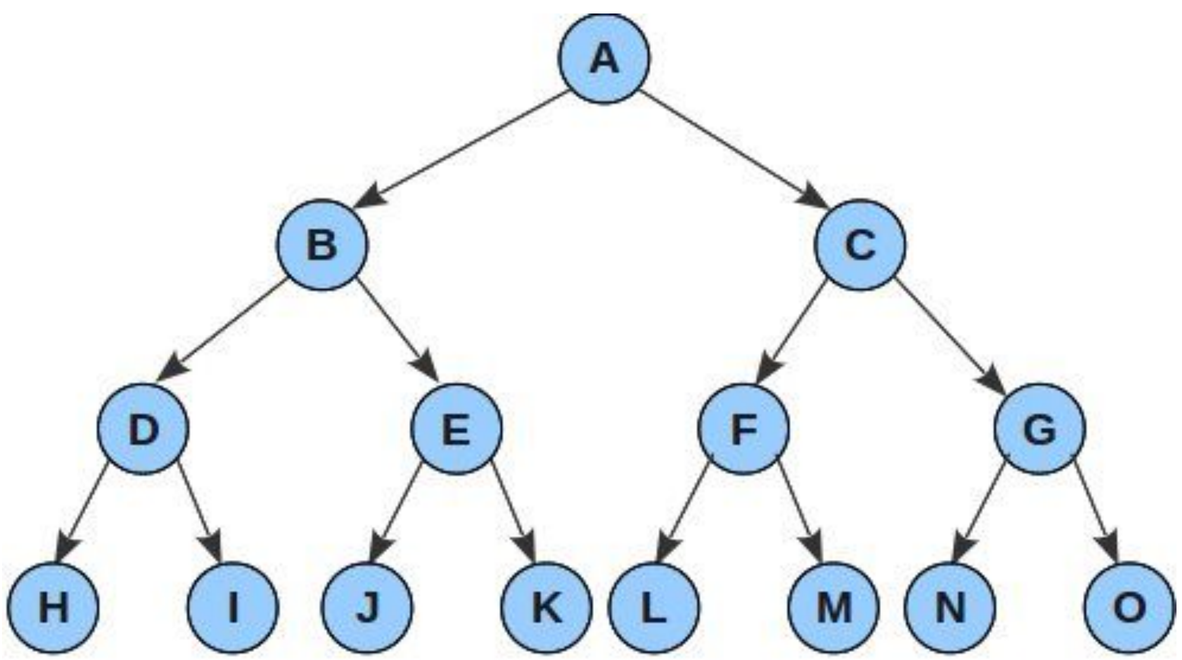
- If a full binary tree is of height h, how many leaf nodes does it have? How many nodes (including leaf and interior) does it have?



Full Binary Tree

  - Work on a few examples and you will find out.

Math of Full Binary Trees



level L	Number nodes at level L
0	1
1	2
2	4
3	8
…	…
h	$\mathbf{2}^{h}$

Number of nodes at level L=

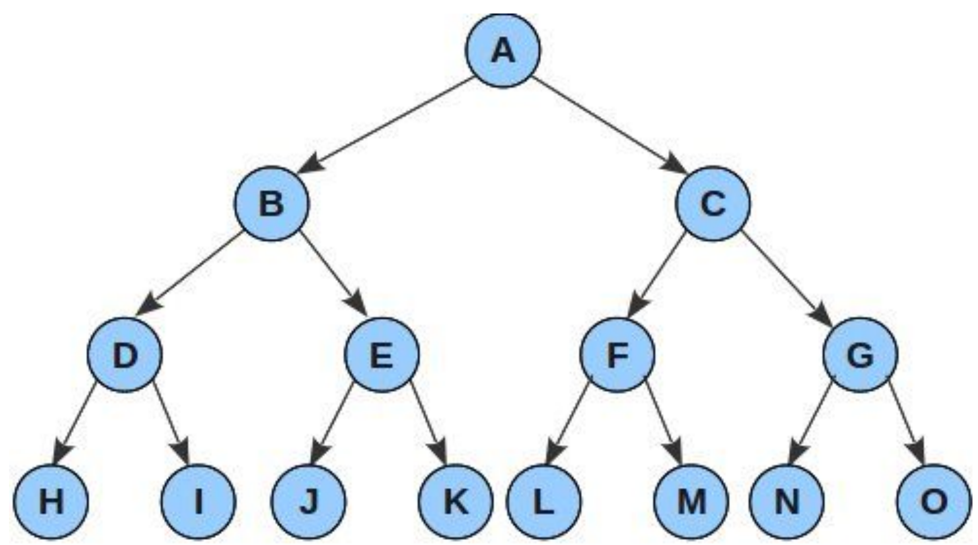
Math of Full Binary Trees

Total # nodes in a full binary tree of height h

$$N = 2^0 + 2^1 + 2^2 + \ldots + 2^h$$

$$= 2(2^h) - 1$$

$$= 2^{(h+1)} - 1$$



1	level 0
2	level 1
4	level 2
8	level 3
15	Total

Conversely, the height of a full binary tree with N nodes is: $h = \log_2(N+1)-1 = O(\log N)$