Programming with Data Structures

CMPSCI 187
Spring 2016

- **Please find a seat**
  - **Try to sit close to the center (the room will be pretty full!)**
- **Turn off or silence your mobile phone**
- **Turn off your other internet-enabled devices**

Reminders and Topics

- **Project 7 is due this Friday**
  - **reuse methods**
  - **write your own tests!**
- **The second midterm is Wed, March 30, 7-9pm**

- This lecture:
  - **Binary Search**
  - **Binary Trees**

Search / Find an Element

- So far, we've learned that searching / finding an 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 if they stored in a **sorted array**, we can do a lot better, using an algorithm called **binary search**.

- To explain it, let's start with a simple game of guessing a number.

Guess-a-Number Game

- A friend picks a number between 1 to n (say n=1000), and asks you to guess that number.
- When you make a guess, she will tell you one of three things — your guess is 1) too large, or 2) too small, or 3) correct.
- How would you make your guesses in order to find out the number in the fewest number of steps?
    - 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 she 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 $\texttt{ceiling}(\log_2 1000) = 10$ 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).

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

find target=41.

Using linear search, it requires 13 steps / iterations.

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

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]

What if we are to find target=42?

Using linear search, it requires 14 steps.

Using binary search, it requires 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 =
        (Comparable)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 =
        (Comparable)target.compareTo(list[curr]);
    if (result == 0)
      return curr;
    else if (result < 0)
      upper = curr - 1;
    else
      lower = curr + 1;
  }
  return -1;
}
```

Clicker Question #1

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

What happens if the boxed line is changed to `lower = curr` instead of `curr+1`?

a) the loop may run forever.

b) it may fail to find an existing element.

c) it may throw a `NullPointerException`

d) it may throw an `Index OutofBoundException`

Clicker Question #2

```
protected int find (T target) {
  int lower=0, upper=numElements-1;
  while (lower < upper) {
    int curr=(lower+upper)/2;
    int result=(Comparable)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) the loop may run forever.

b) it may fail to find an existing element.

c) it may throw a `NullPointerException`

d) it may throw an `Index OutofBoundException`

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 a large array. 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) for sorted vs O(1) for unsorted.

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

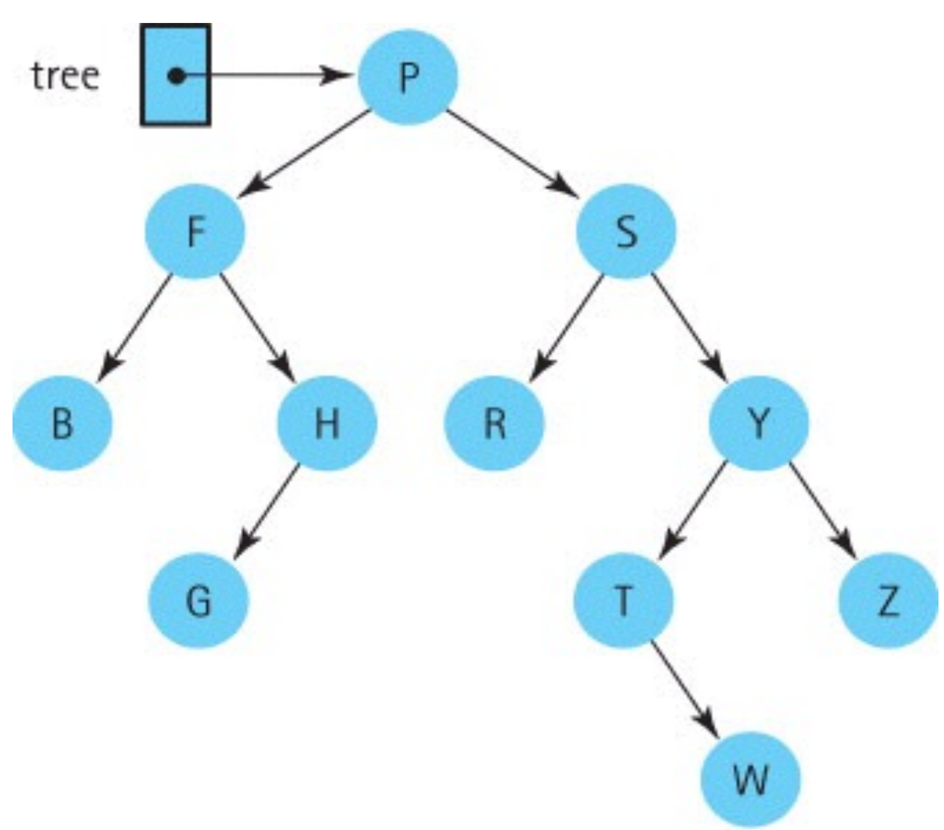
Binary Search — Recursive Version

```
protected int recFind (Comparable 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) {
  Comparable tar = (Comparable) target;
  return recFind (tar, 0, numElements - 1);
}
```

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).

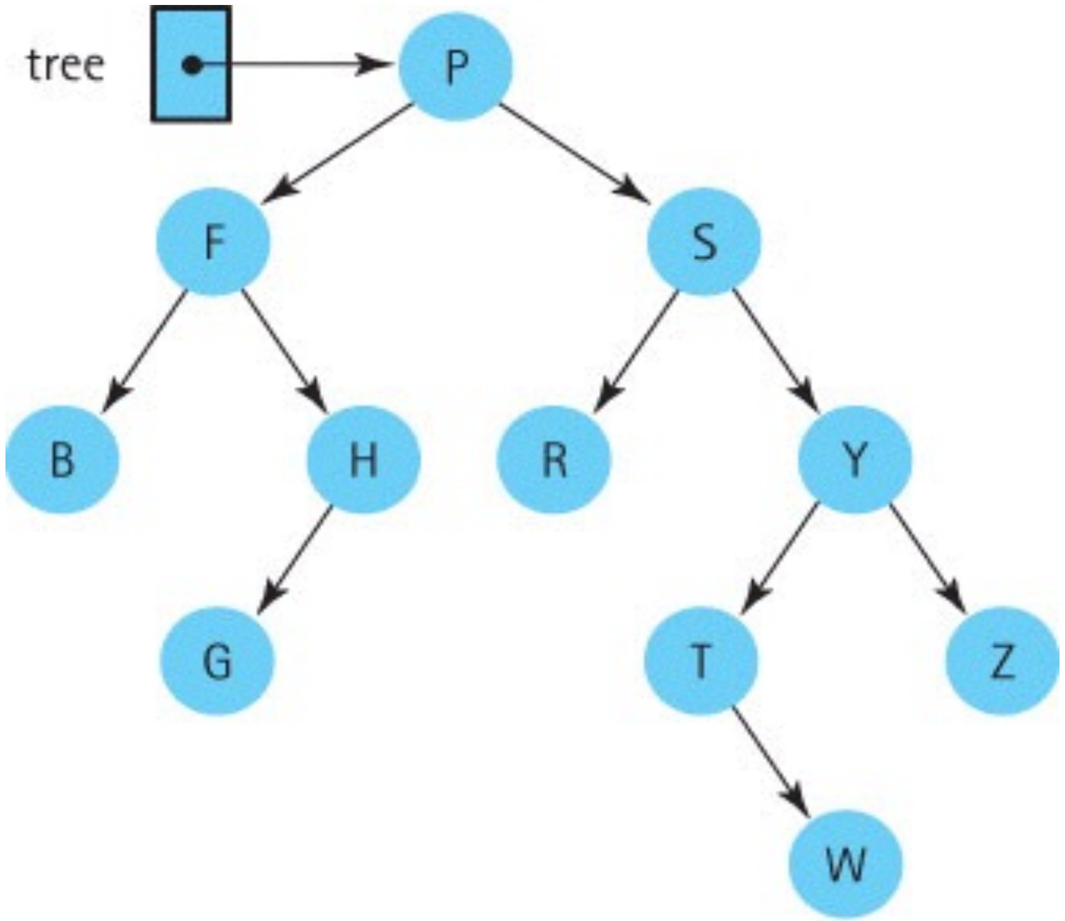
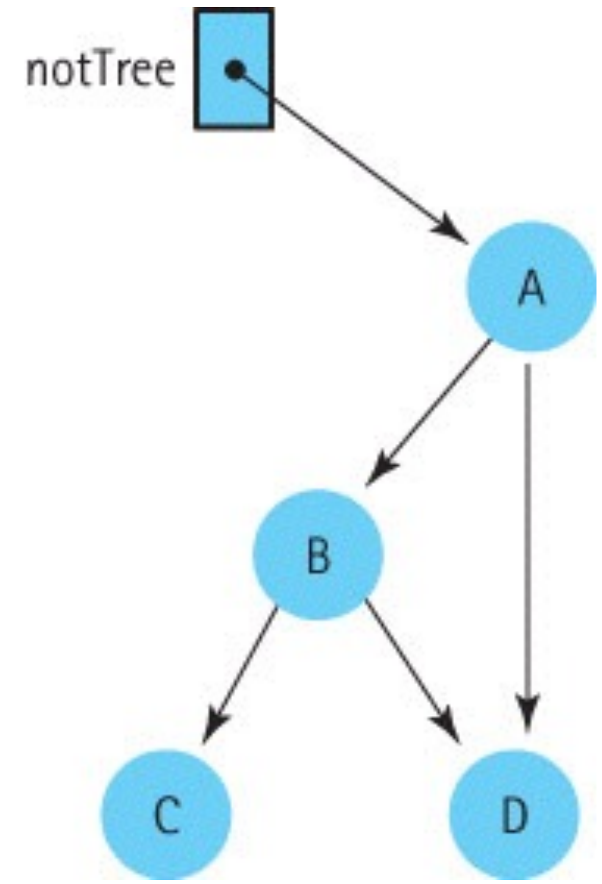


The Tree Data Structure

- 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…

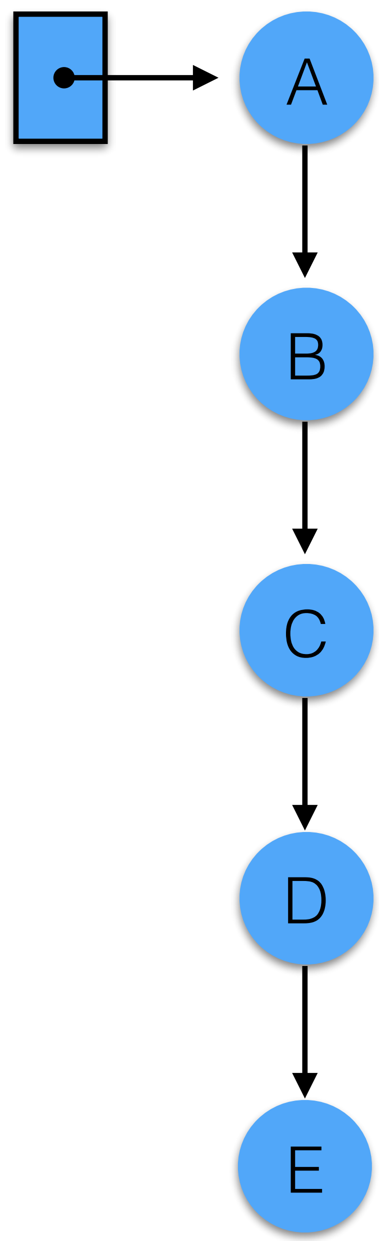
The Tree Data Structure

- Mathematically speaking, trees are **connected, acyclic graphs** (i.e. no loops).
  - There is one unique root
  - From the root to any node there is one and only one unique path.
- It's very useful for representing hierarchical structures, such as file systems, Java's 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 ?

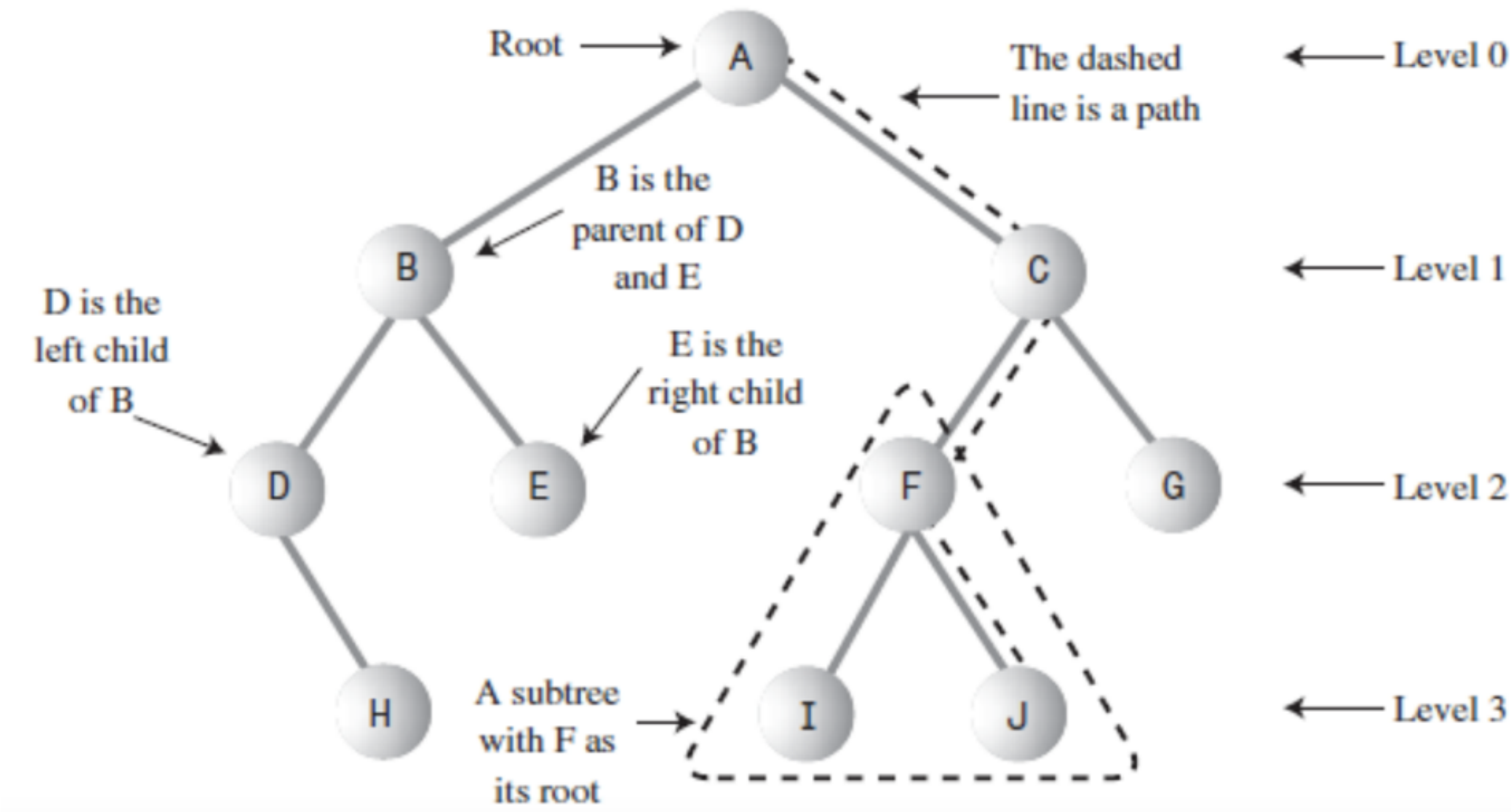


✅ • Unique root

✅ • 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. Why?
- **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**

Clicker Question #3

Remember that a node in a binary tree may have zero, one, or two children, and that the height of a tree is the length of the longest path from the root to a leaf. What are the possible sizes (number of nodes) of a binary tree of height 3?

(a) must be 15 (b) anywhere 1 to 15

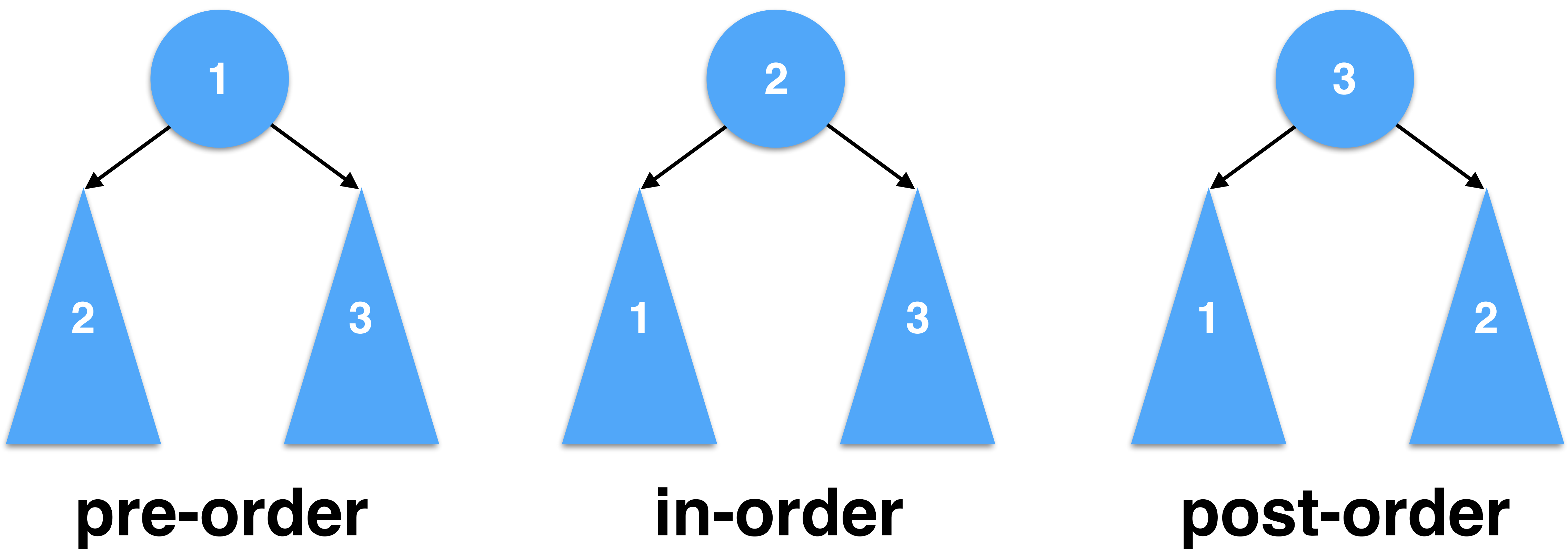
(c) anywhere 8 to 15 (d) anywhere 4 to 15

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*** in relation to its children:
  - **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:

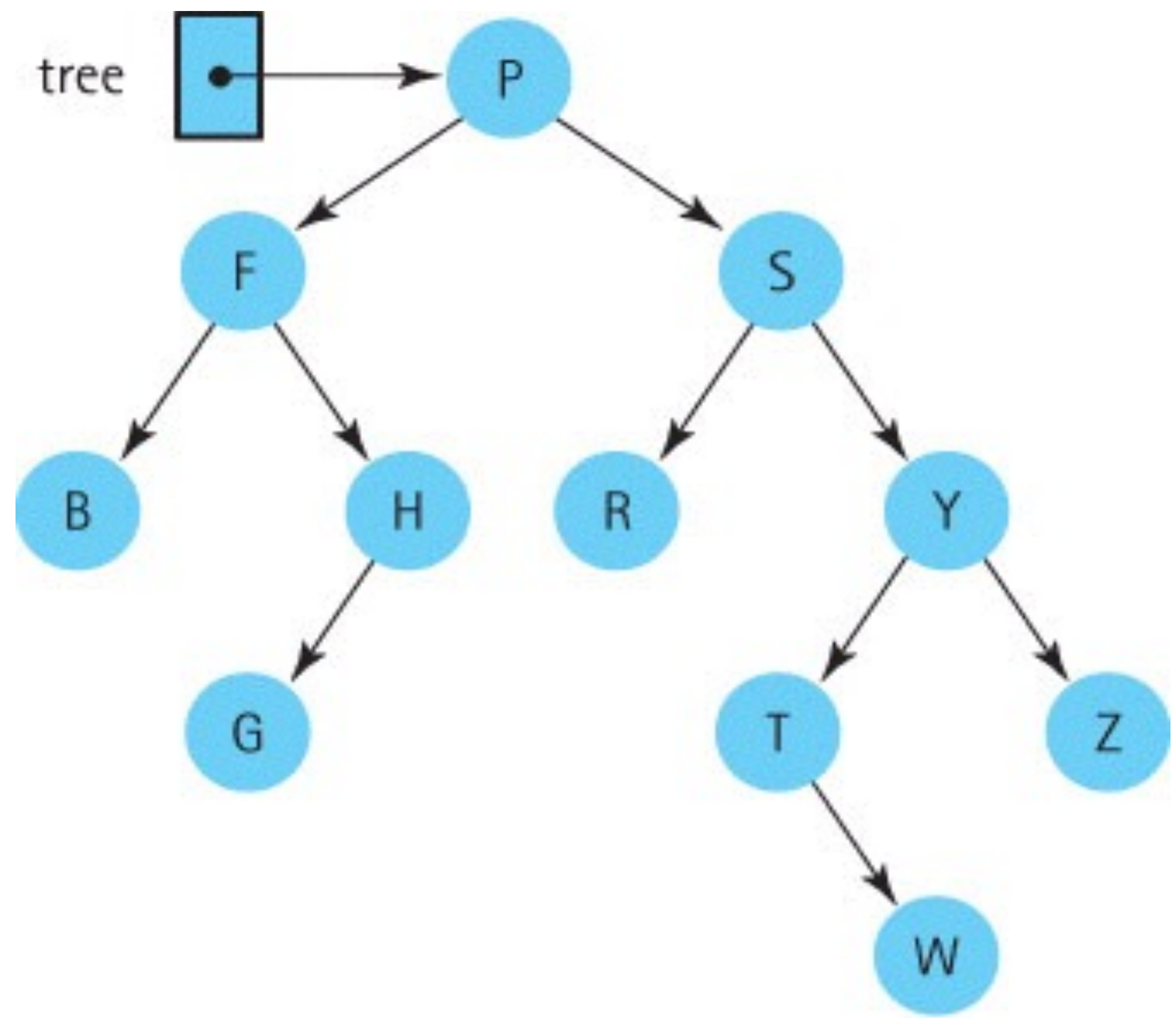


(The numbers above refer to the order of traversal.)

- 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 #4

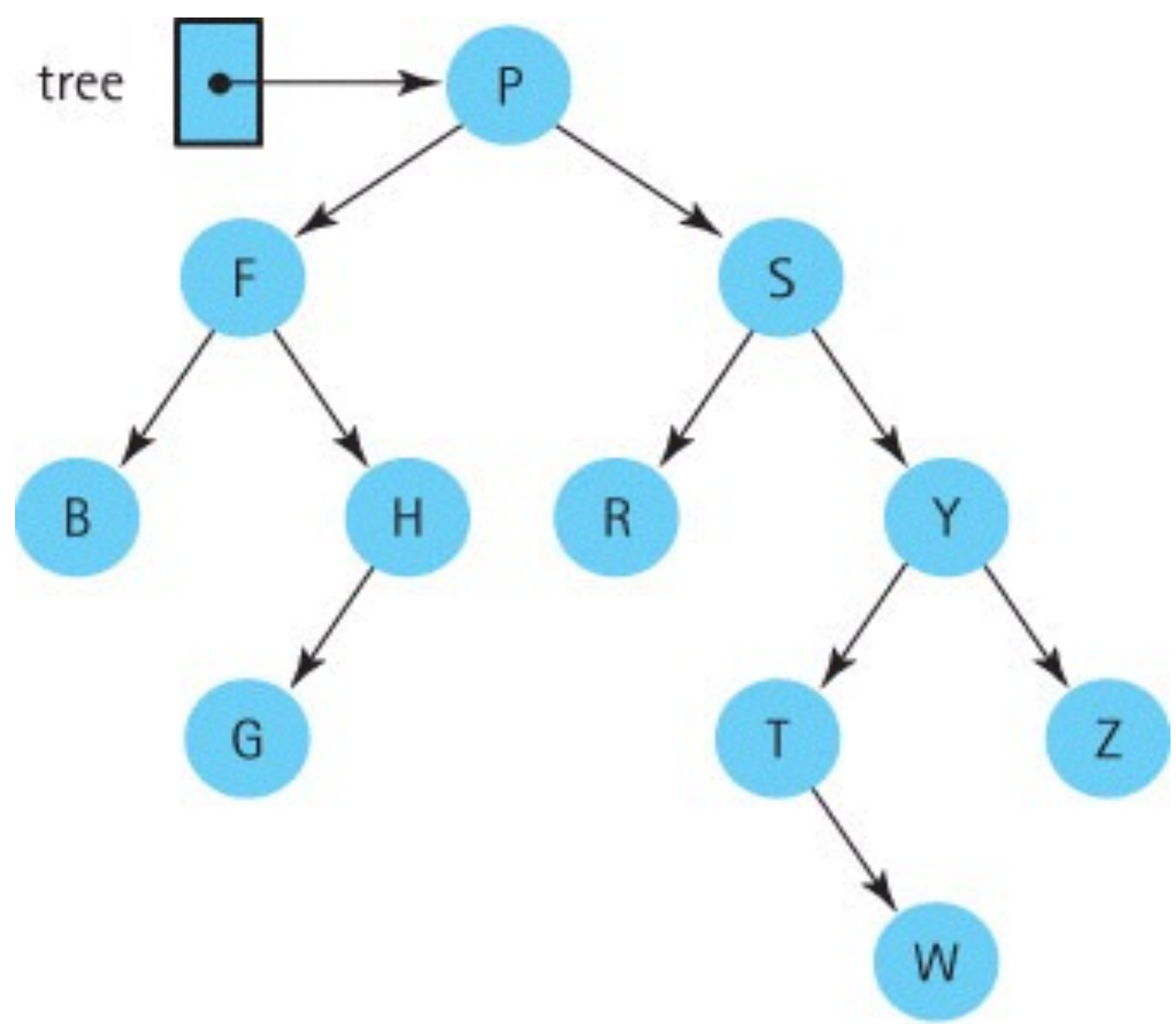
What's the post-order traversal result of this tree?

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

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

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

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



Recursive Traversals of Trees

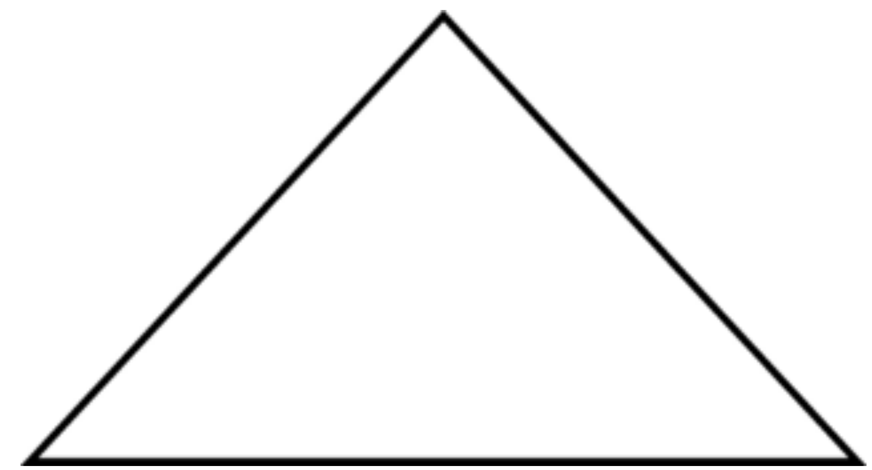
```
public void preOrder(TreeNode x) {
  if (x != null) {
    // visit by printing the value
    System.out.println(x.getInfo());
    preOrder(x.getLeft());
    preOrder(x.getRight());
  }
}
```

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.

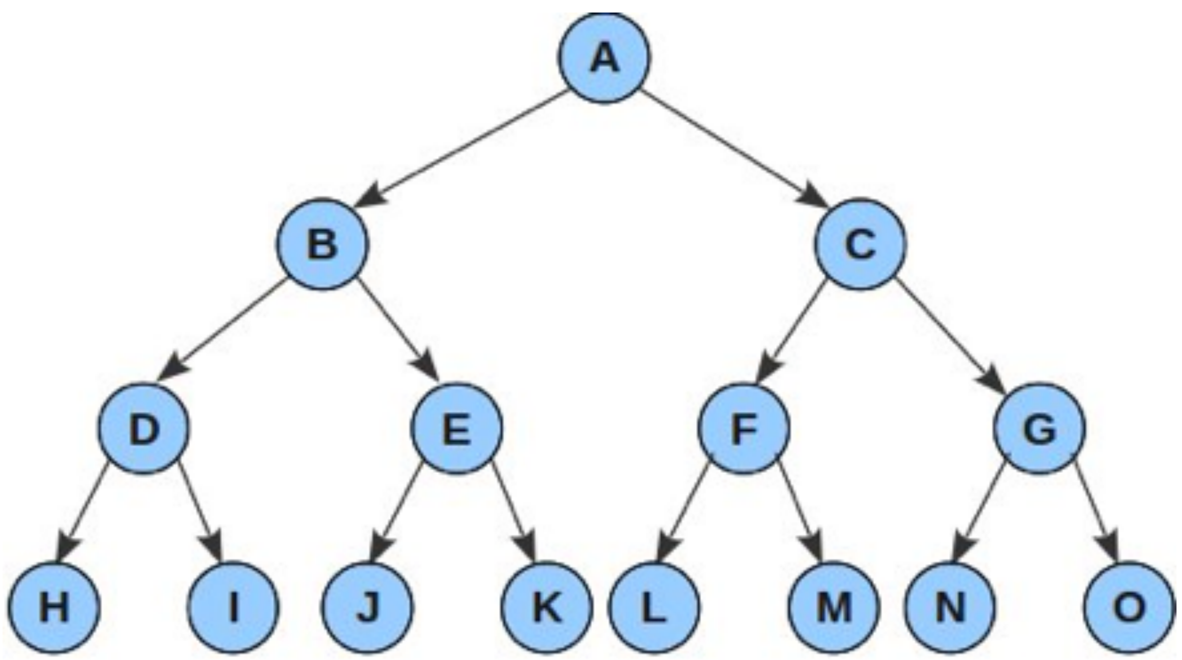
- If a full binary tree is of height **h**, how many leaf nodes does it have? How many nodes (including leaf and interior) does it have?



Full Binary Tree

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

Math of Full Binary Trees



Number of nodes at level L=

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

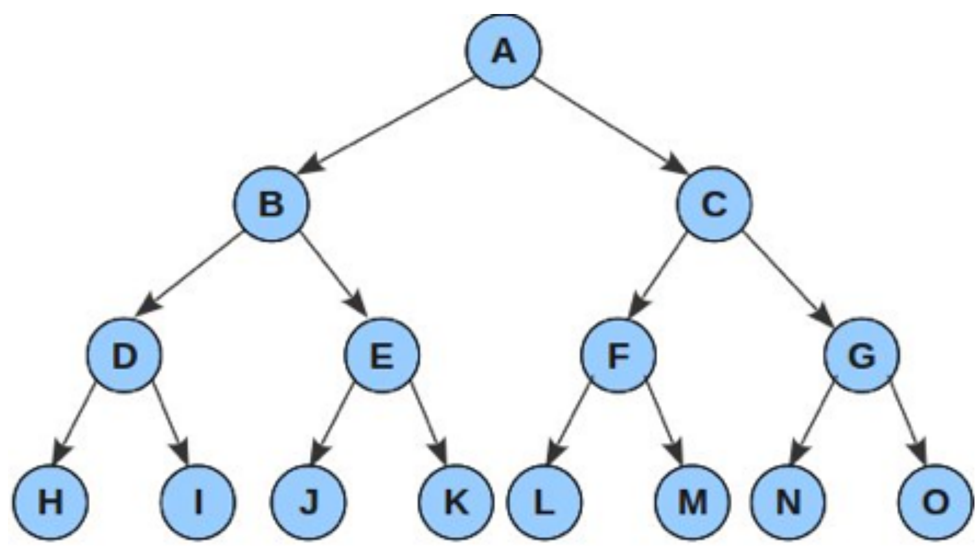
Math of Full Binary Trees

Total # nodes in a full binary tree of height **h**

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

$= 2(2^h) - 1$

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



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: $h = \log_2(N+1)-1 = O(\log N)$