

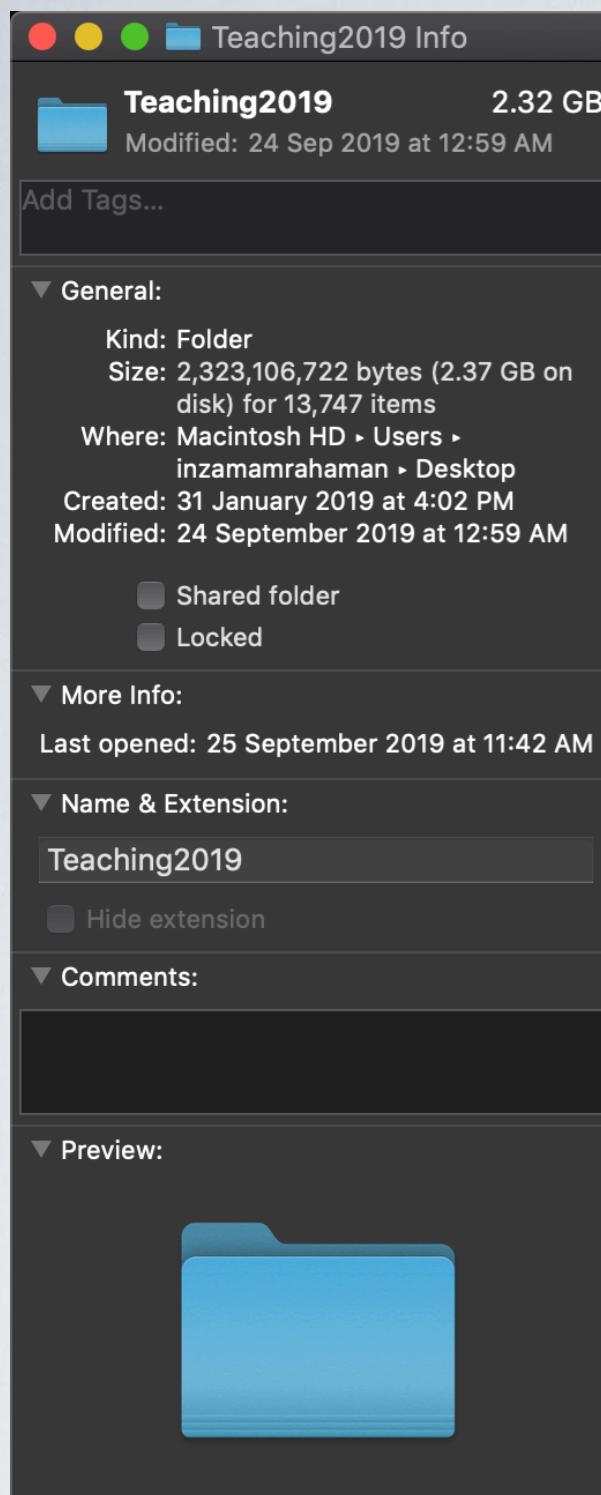
Trees and Binary Trees

COMP2611: Data Structures

Sept 2019

Outline

- ▶ Trees
 - ▶ Basic definitions and properties
 - ▶ Notion of a traversal on a tree
 - ▶ BFS (Breadth-first traversal)
 - ▶ DFS (Depth-first traversal)
- ▶ Binary Trees
 - ▶ Pre-order, in-order, post-order
 - ▶ Properties and how to prove them



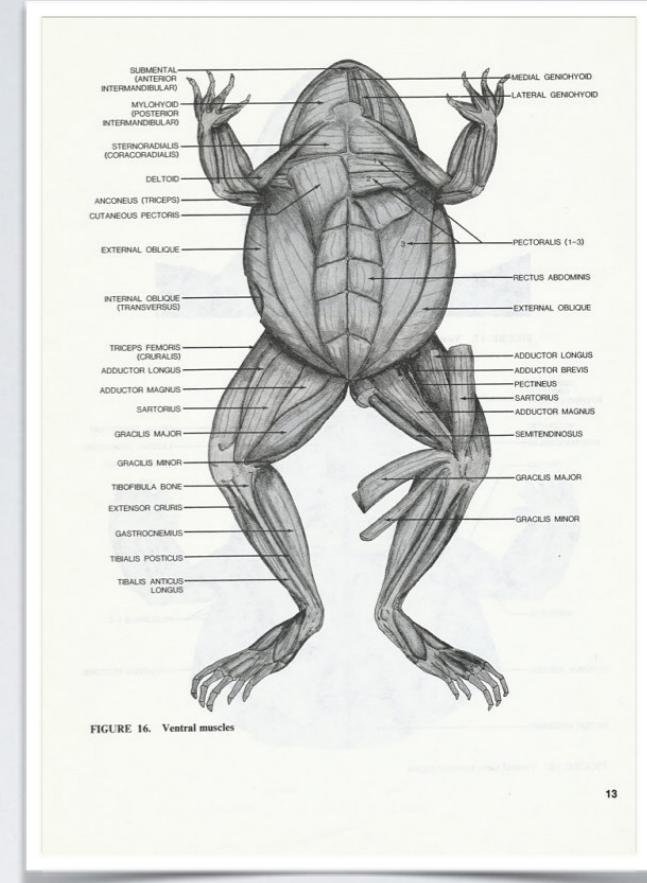
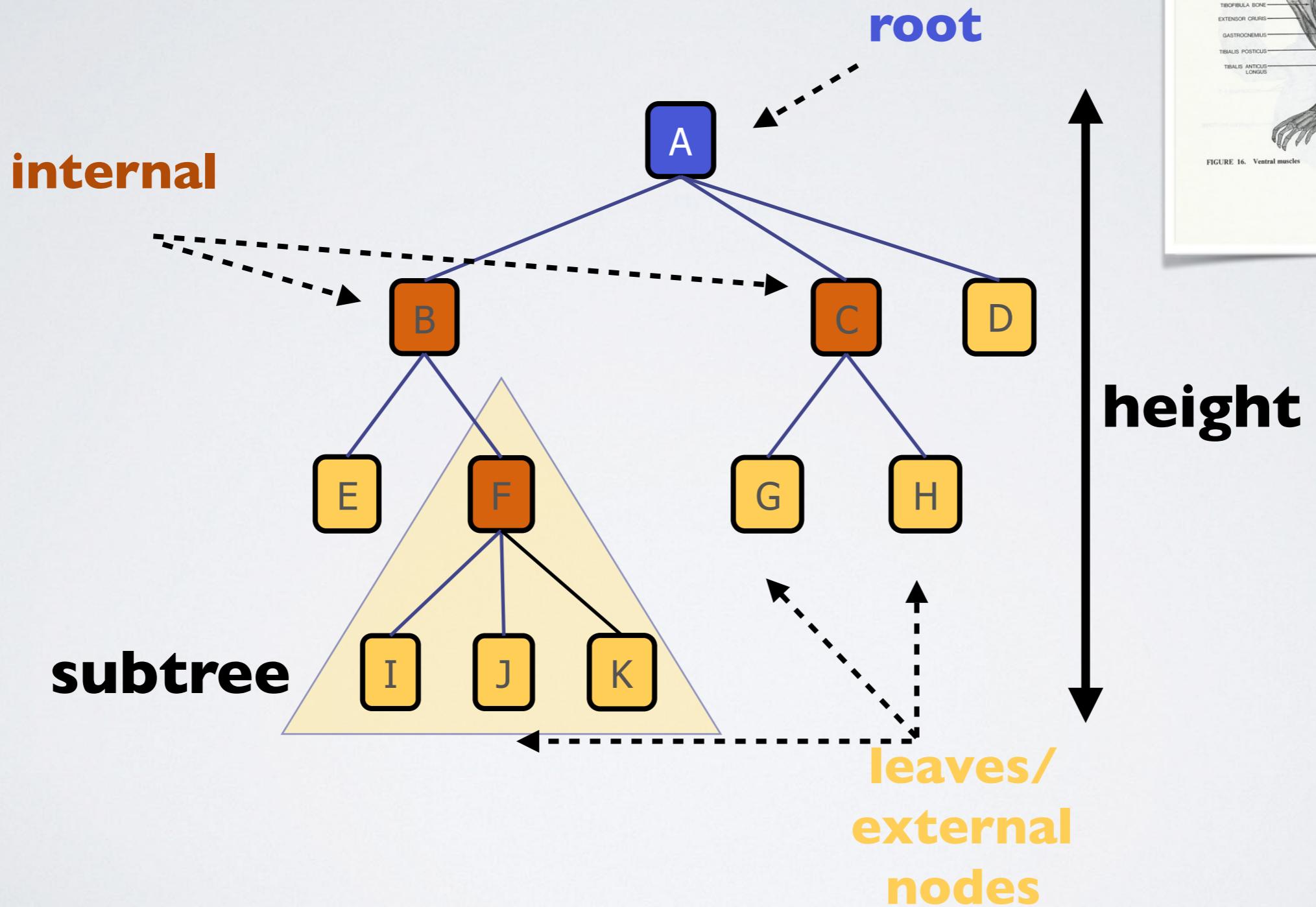
► How does OS calculate size of
directories?

What is a Tree?

- ▶ A lot of data and information can be organised hierarchically
- ▶ Such data can be stored in a tree
- ▶ Tree consists of
 - ▶ nodes with parent/child relationship
 - ▶ formally, a connected acyclic graph (DAG) where every node is connected to every other node by a single path (definition will become clearer towards end of course)
- ▶ Examples
 - ▶ Files/folders (Windows, MacOSX, ...)
 - ▶ Structure of Source code in compiler or parser
 - ▶ Decision Trees



Tree “Anatomy”



Tree Terminology

- ▶ **Root:** node without a parent (A)
- ▶ **Internal node:** node with at least one child (A, B, C, F)
- ▶ **Leaf (external node):** node without children (E, I, J, K, G, H, D)
- ▶ **Parent node:** node immediately above a given node (parent of C is A)
- ▶ **Child node:** node(s) immediately below a given node (children of C are G and H)
- ▶ **Ancestors of a node:**
 - ▶ parent, grandparent, grand-grandparent, etc. (ancestors of G are C, A)
- ▶ **Descendant of a node:** child, grandchild, grand-grandchild, etc.
- ▶ **Depth of a node:** number of ancestors (I has depth 3)
- ▶ **Height of a tree:**
 - ▶ maximum depth of any node (tree with just a root has height 0, this tree has height 3)
- ▶ **Subtree:** tree consisting of a node and its descendants

Example Tree ADT



- ▶ Tree methods:
 - ▶ int **size()**: returns the number of nodes
 - ▶ boolean **isEmpty()**: returns true if the tree is empty
 - ▶ Node **root()**: returns the root of the tree
- ▶ Node methods:
 - ▶ Node **parent()**: returns the parent of the node
 - ▶ Node[] **children()**: returns the children of the node
 - ▶ boolean **isInternal()**: returns true if the node has children
 - ▶ boolean **isExternal()**: returns true if the node is a leaf
 - ▶ boolean **isRoot()**: returns true if the node is the root
 - ▶ int **numDescendents()**: returns the number of descendants of this node

Implementation

- ▶ Usually implemented similarly to linked lists
- ▶ Some special trees can also be implemented using arrays

Implementation (Python 3.x)

```
class TreeNode:  
    def __init__(self, data, parent=None, children=[]):  
        self.data = data  
        self.parent = parent  
        self.children = children  
  
    def is_root(self):  
        return self.parent is None  
  
    def is_internal(self):  
        return len(self.children) > 0  
  
    def is_external(self):  
        return not self.is_internal()  
  
    def get_children(self):  
        return self.children  
  
    def add_child(self, child):  
        child.parent = self  
        self.children.append(child)  
  
    def get_num_descendants(self):  
        count = 0  
        for child in self.children:  
            count += 1  
            count += child.get_num_descendants()  
        return count
```

Implementation (Python 3.x)

```
class Tree:  
    def __init__(self, root=None):  
        self.root = root  
  
    def is_empty(self):  
        return self.root is None  
  
    def size(self):  
        if self.root is not None:  
            count = 1 + self.root.get_num_descendants()  
        return count  
    return 0
```

General Trees

- ▶ General Tree structures are more difficult to analyse
 - ▶ Will look at better ways of implementing and analysis when doing Graphs towards the end of the course
- ▶ Will focus on special case called binary trees
 - ▶ But note that some (though not all) results generalise
 - ▶ In particular with traversals

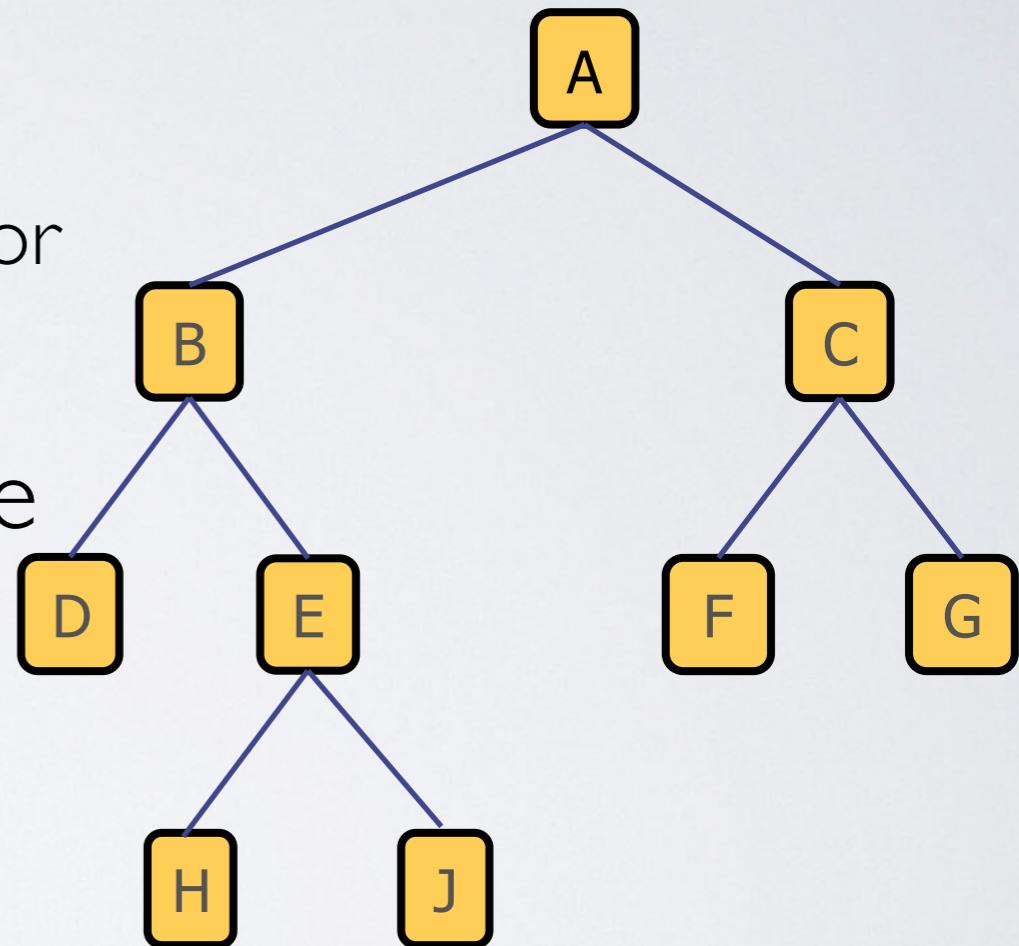
Binary Trees

- ▶ Internal nodes have at most 2 children

- ▶ left and right
- ▶ if only 1 child, still need to specify if left or right

- ▶ Recursive definition of a Binary Tree

- ▶ a single node
- ▶ or a root node with at most 2 children
 - ▶ each of which is a binary tree



Binary Tree ADT

- ▶ In addition to Tree methods *binary* trees have:
 - ▶ Node **left()**: returns the left child if it exists, else NIL
 - ▶ Node **right()**: returns the right child if it exists, else NIL
 - ▶ Node **hasLeft()**: returns TRUE if node has left child
 - ▶ Node **hasRight()**: returns TRUE if node has right child



How to implement?

- ▶ Can use linked structure
 - ▶ Pointers or references to left child, right child, or parent
- ▶ Can use array
 - ▶ Start storing nodes at index 1
 - ▶ Node at index i has left child at $i \times 2$ and $i \times 2 + 1$
 - ▶ Node at index i has parent $\lfloor i / 2 \rfloor$
- ▶ Most analysis applied to both implementations
- ▶ Should be able to implement either one!

Trade offs

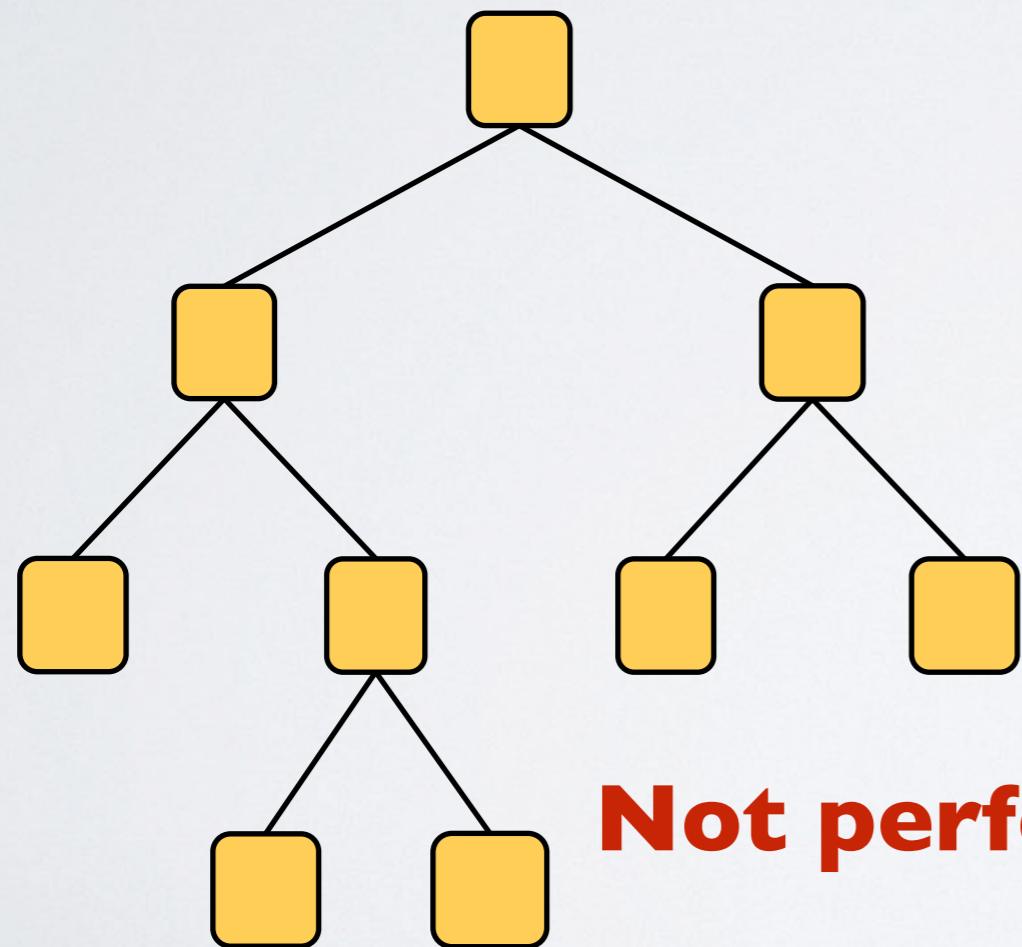
- ▶ Linked structures require dynamic memory allocation
 - ▶ Not a good idea for most embedded systems....
- ▶ Array access is quicker
 - ▶ Exploits cache locality
- ▶ Linked structure matches intuition more; hence “easier” to implement
 - ▶ Tip remember the formulae for left children, right children, and parents!
 - ▶ Then just “swap” out access through pointers

Implementation of Linked (Python 3.x)

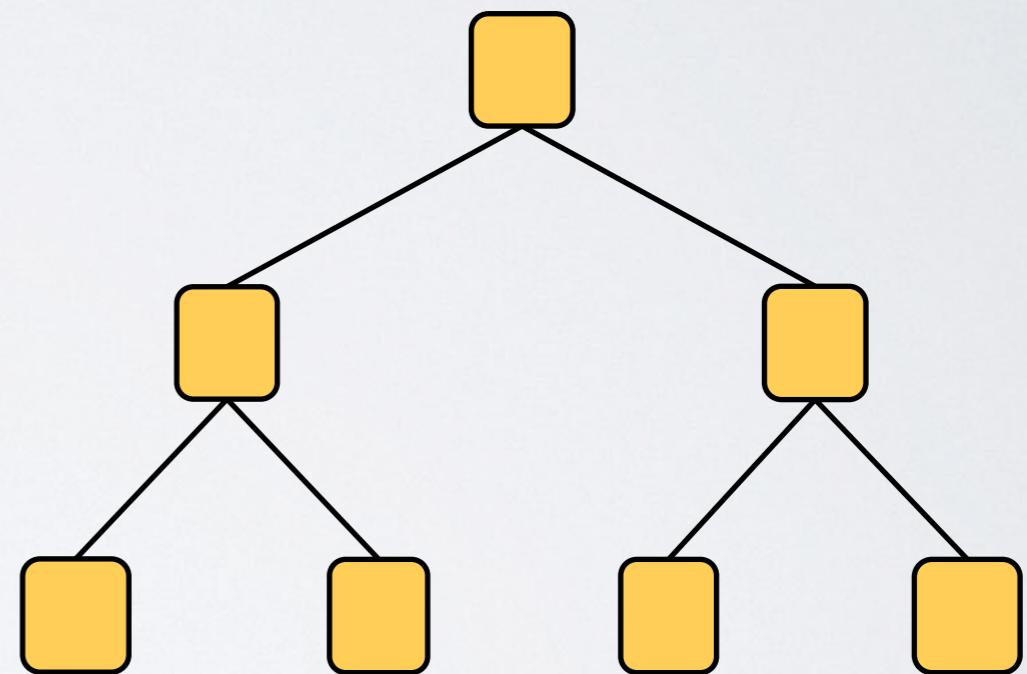
```
class TreeNode:  
    def __init__(self, data, parent=None, left=None, right=None):  
        self.data = data  
        self.parent = parent  
        # direct access to modify children  
        self.left = left  
        self.right = right  
  
    def is_root(self):  
        return self.parent is None  
  
    def has_left(self):  
        return self.left is not None  
  
    def has_right(self):  
        return self.right is not None  
  
    def is_internal(self):  
        return self.left is None and self.right is None  
  
    def is_external(self):  
        return not self.is_internal()  
  
    def get_num_descendants(self):  
        count = 0  
        if self.has_left():  
            count += 1 + left.get_num_descendants()  
        if self.has_right():  
            count += 1 + right.get_num_descendants()  
        return count
```

Perfection

- ▶ A binary tree is **perfect** if
 - ▶ every level is completely full



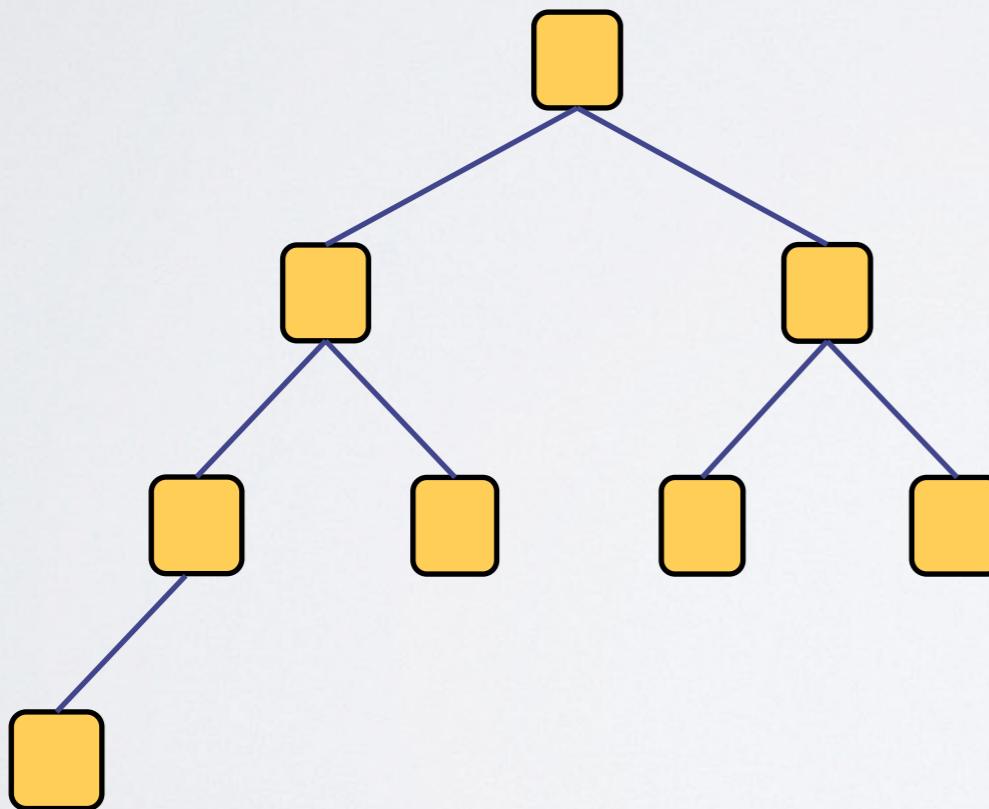
Not perfect



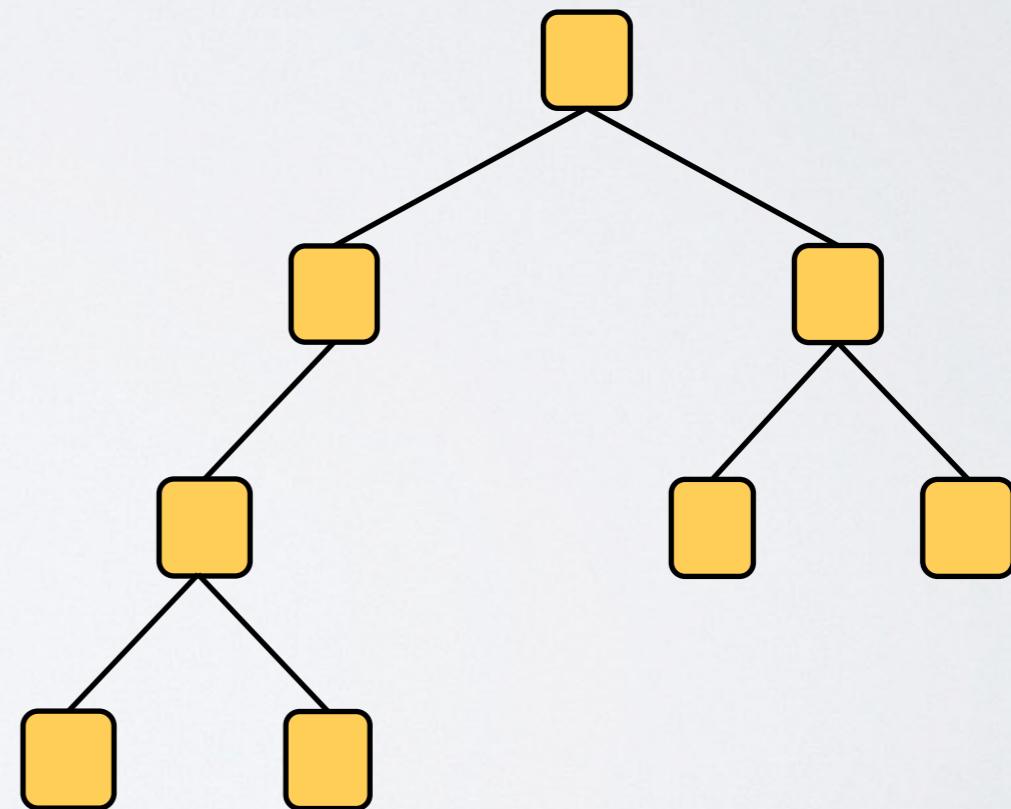
Perfect!

Completeness

- A binary tree is **left-complete** if
 - every level is completely full, possibly excluding the lowest level
 - all nodes are as far left as possible



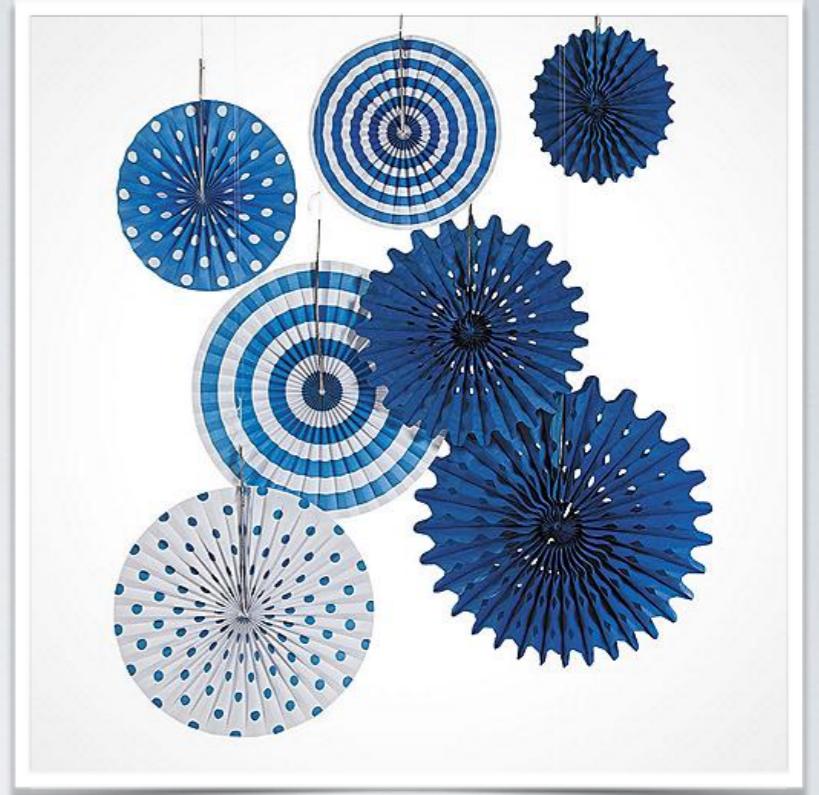
**Left-
complete!**



**Not left-
complete**

Aside: Decorations

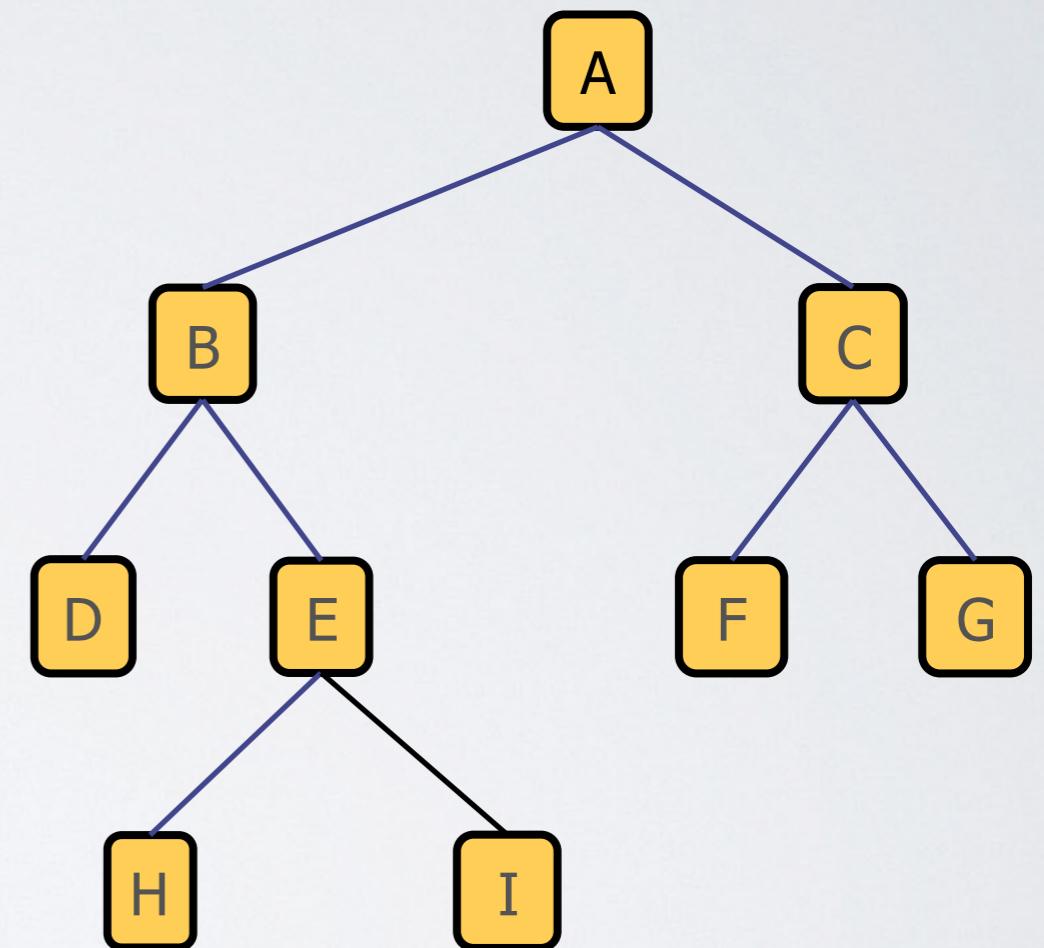
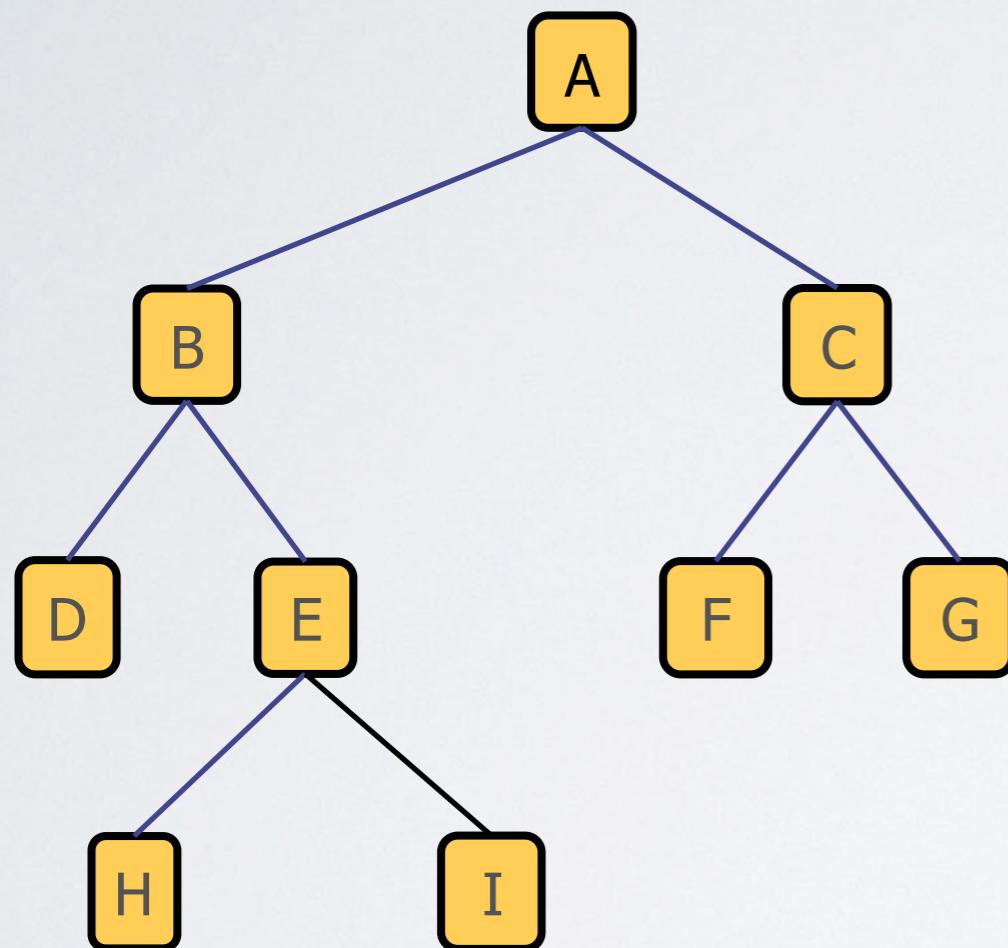
- ▶ Decorating a node
 - ▶ associating a value to it
- ▶ Two approaches
 - ▶ Add new attribute to each node
 - ▶ ex: `node.numDescendants = 5`
 - ▶ Maintain dictionary that maps nodes to decoration
 - ▶ do this if you can't modify tree
 - ▶ ex: `descendantDict[node] = 5`



Tree Traversals

- ▶ How would you enumerate every item in an array?
 - ▶ use a for loop from $i = 1$ to n and read $\text{arr}[i]$
- ▶ How would you enumerate every item in a (linked) Tree?
 - ▶ not obvious...
 - ▶ because Trees don't have an "obvious" order like arrays
- ▶ Tree traversal
 - ▶ algorithm that visits every node of a tree
- ▶ Many possible tree traversals
 - ▶ each kind of traversal visits nodes in different order

Breadth- vs. Depth-First Traversals

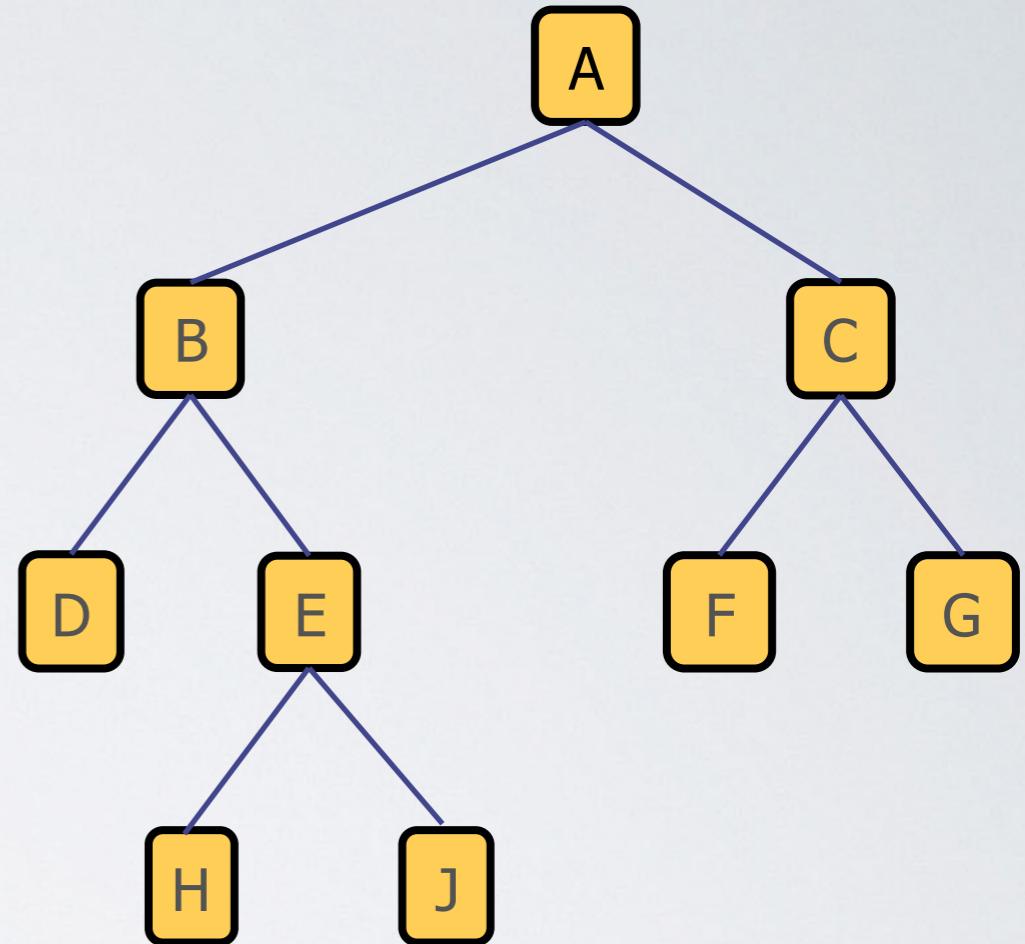


Traversals

- ▶ Can't use for loop
 - ▶ Don't know number of items apriori
 - ▶ Might need to loop for an undefined number of iterations
 - ▶ Might need to adjust direction every iteration
 - ▶ Would effect number of iterations
- ▶ Should use while loop instead

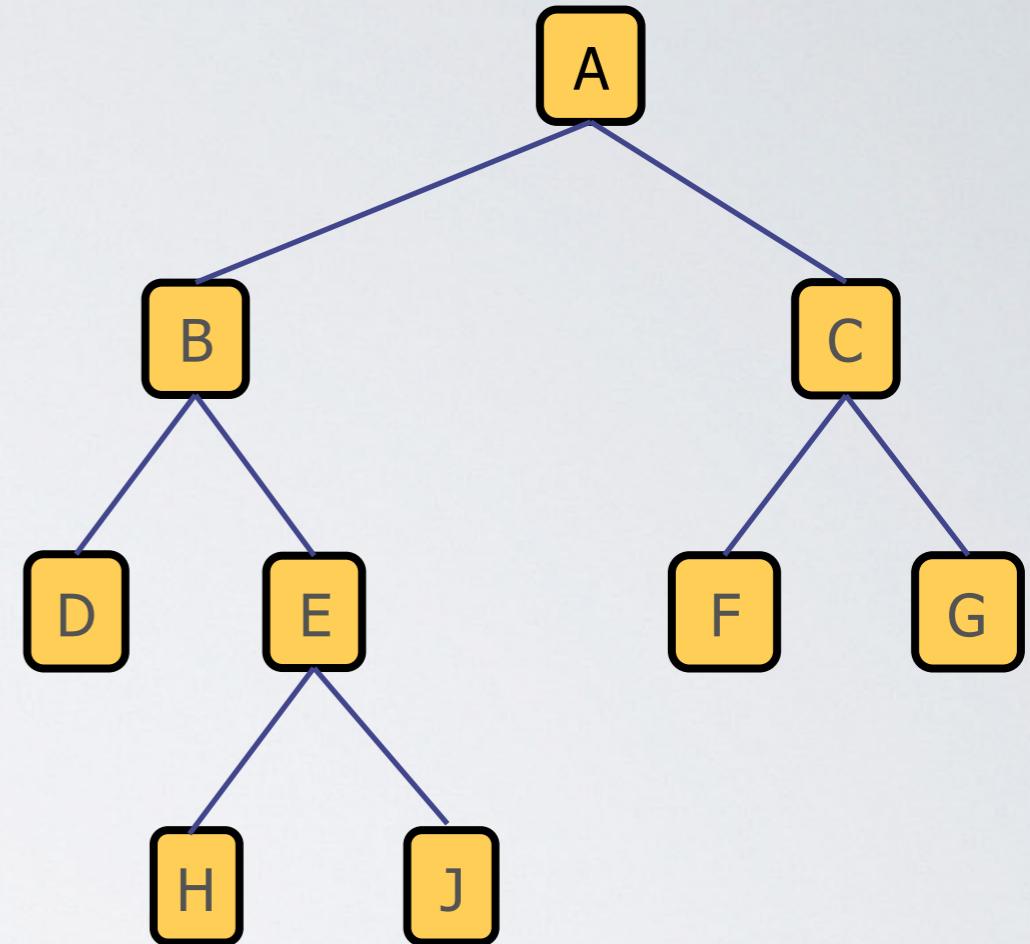
Traversal Strategy

```
function traversal(root):  
    Store root in S  
    while S is not empty  
        get node from S  
        do something with node  
        store children in S
```



Traversal Strategy

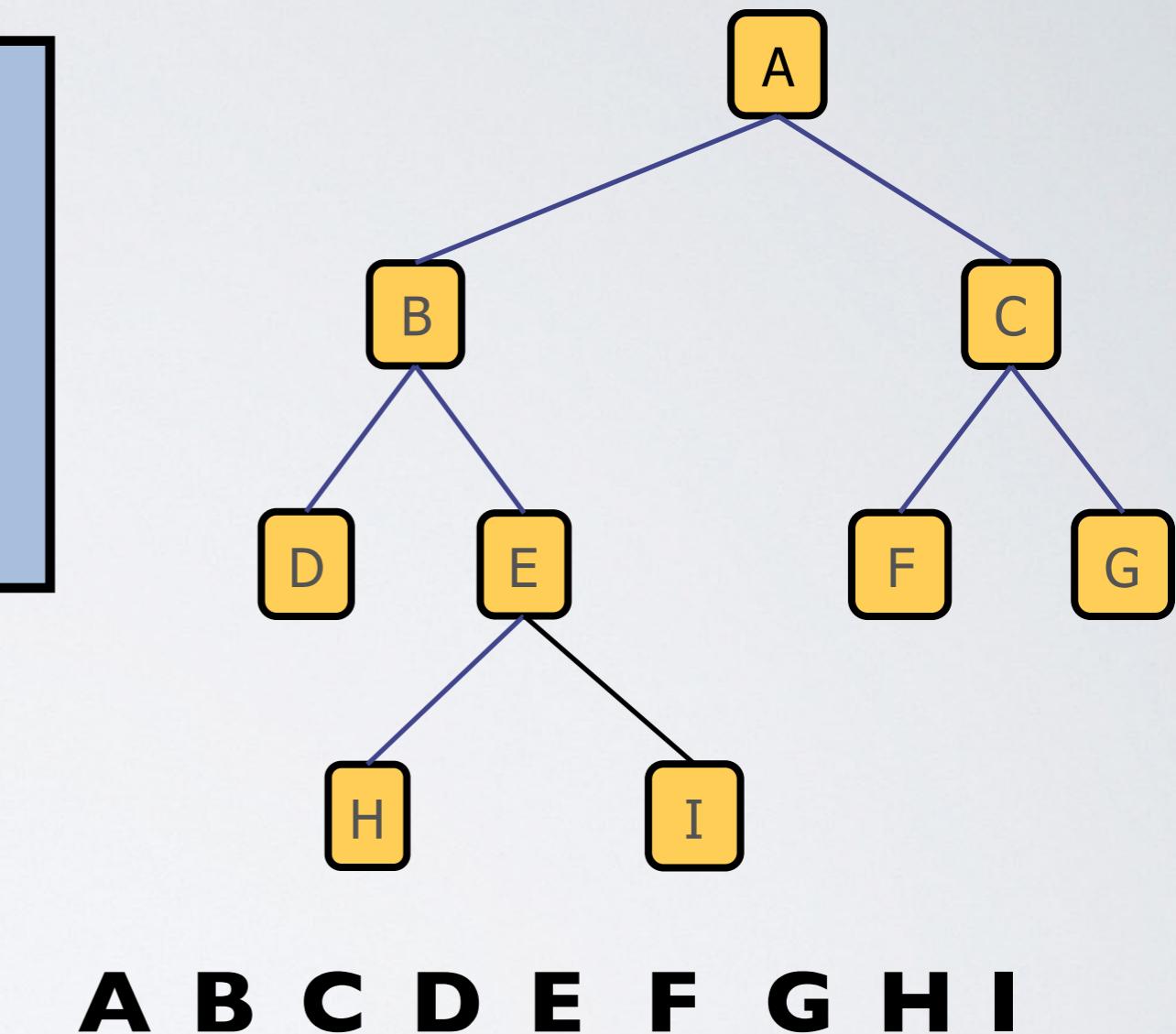
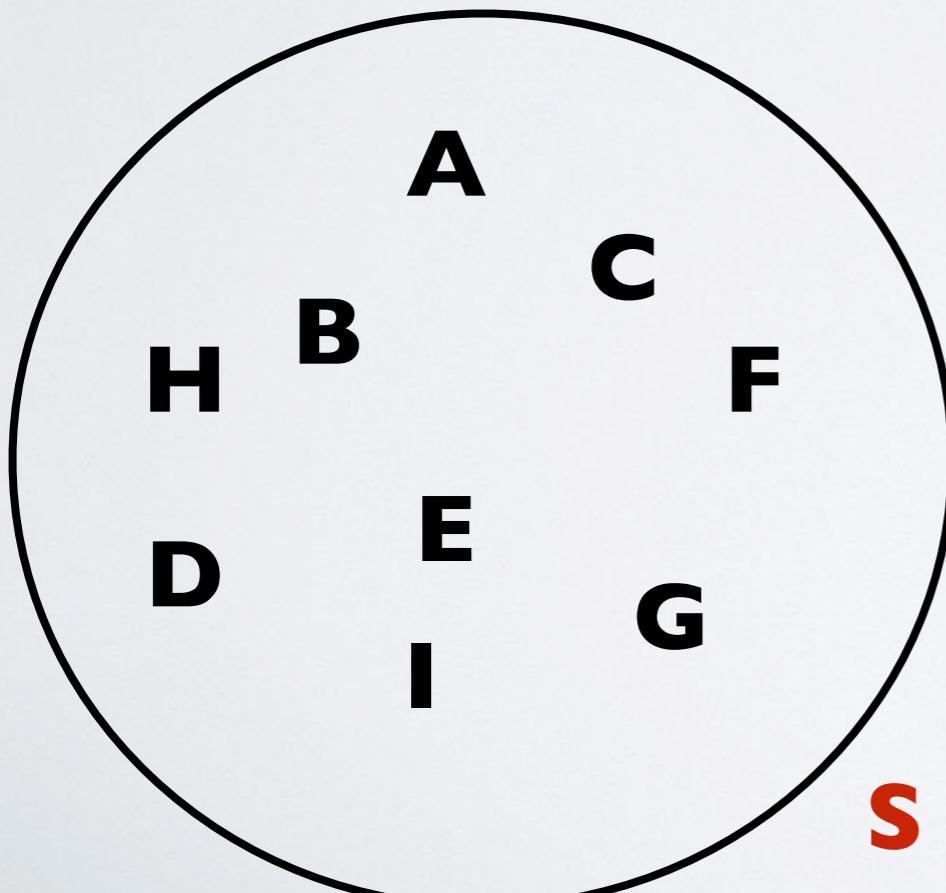
```
function traversal(root):  
    Store root in S  
    while S is not empty  
        get node from S  
        do something with node  
        store children in S
```



- ▶ What is **S** exactly?
 - ▶ A place we store nodes until we can process them
- ▶ Which node of **S** should we process next?
 - ▶ the first? the last?

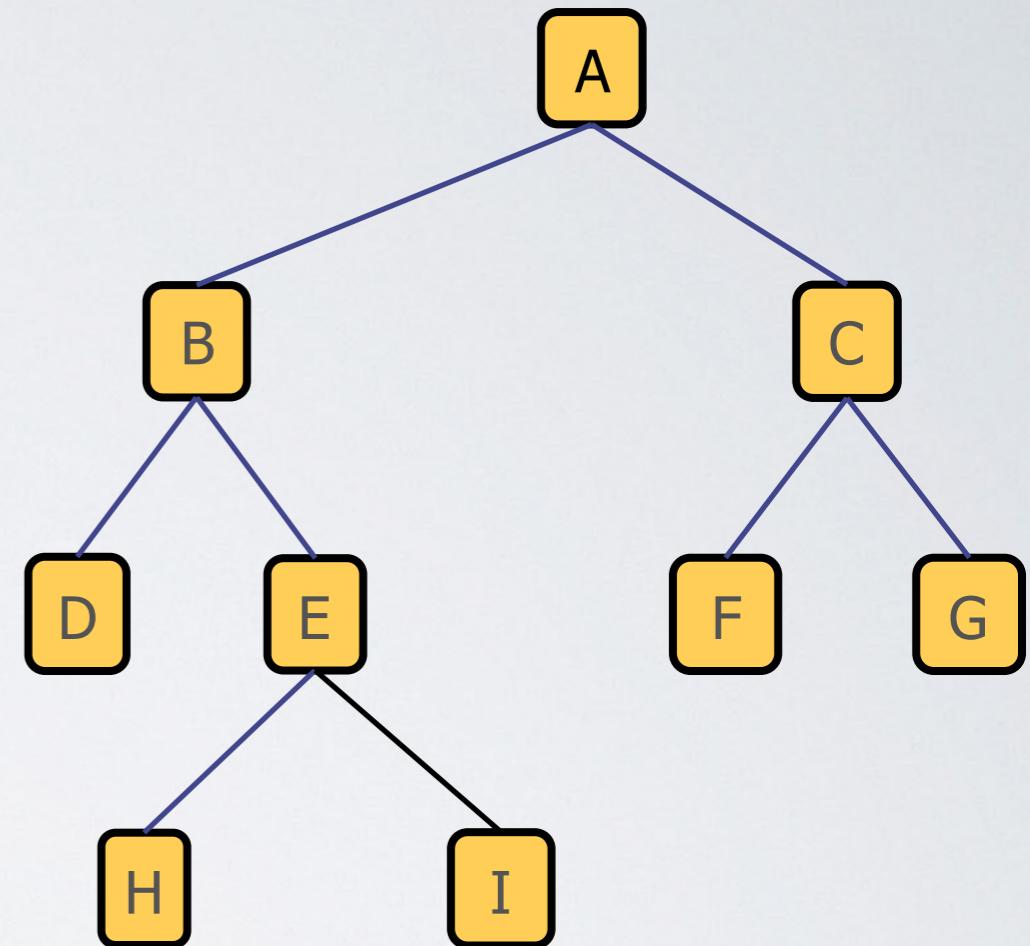
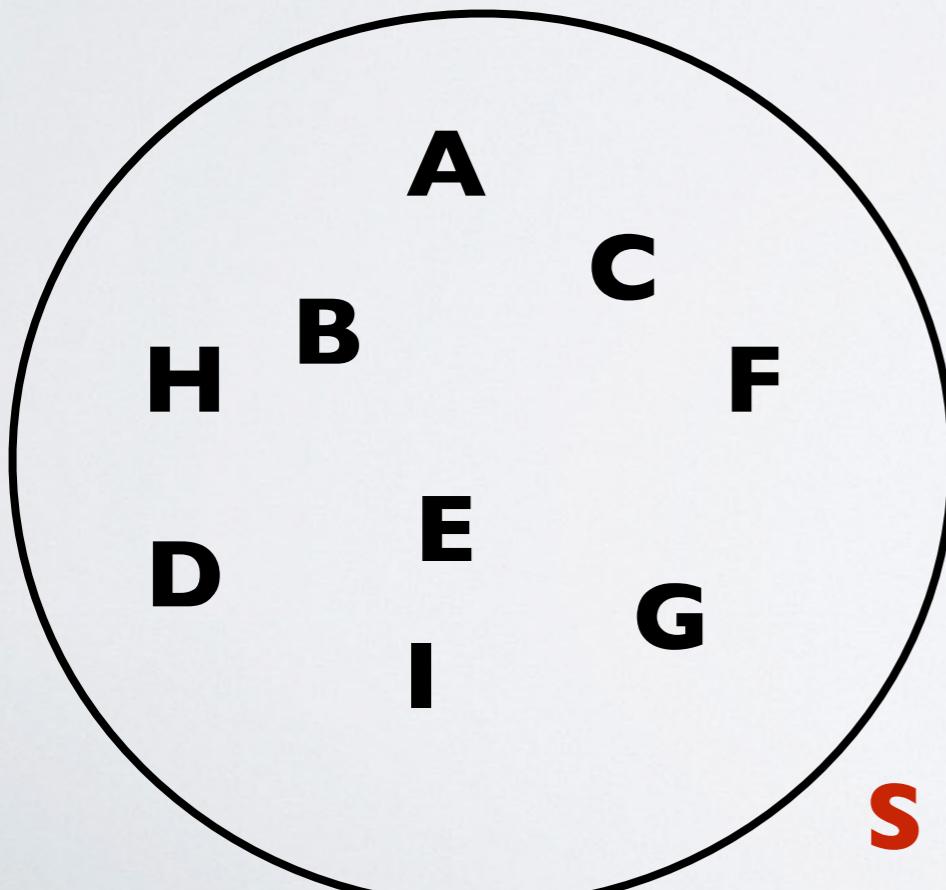
Traversal Strategy — Grab Oldest Node

```
function traversal(root):  
    Store root in S  
    while S is not empty  
        get node from S  
        do something with node  
        store children in S
```



Traversal Strategy — Grab Oldest Node

```
function traversal(root):  
    Store root in S  
    while S is not empty  
        get node from S  
        do something with node  
        store children in S
```



Does S remind you of something?

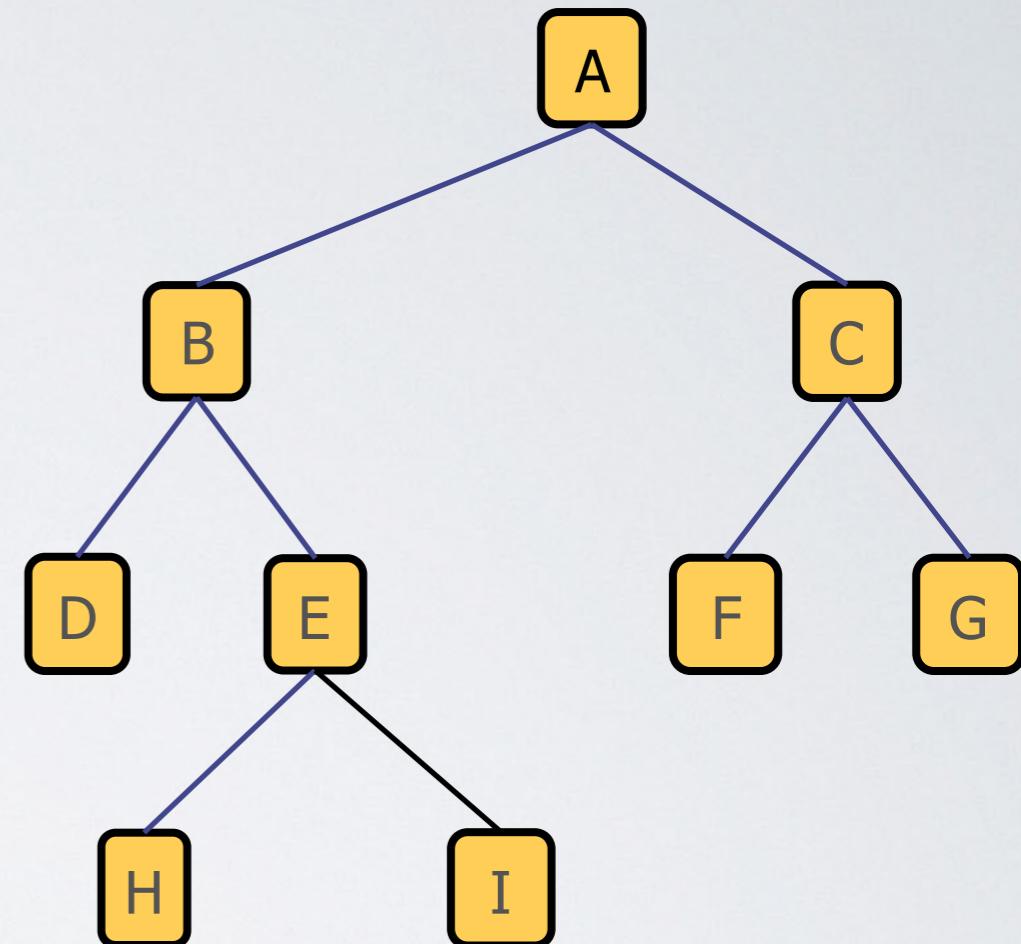
Traversal Strategy — Grab Oldest Node

- ▶ If we grab the oldest node in **S**
 - ▶ we're doing FIFO...
 - ▶ so **S** is a queue!
 - ▶ Traversal w/ Queue gives breadth-first traversal
- ▶ Why?
 - ▶ Queue guarantees a node is processed before its children
 - ▶ Children can be inserted in any order

```
function bft(root):  
    Q = new Queue()  
    enqueue root  
    while Q is not empty  
        node = Q.dequeue()  
        visit(node)  
        enqueue node's left &  
        right children
```

Breadth-First Traversal

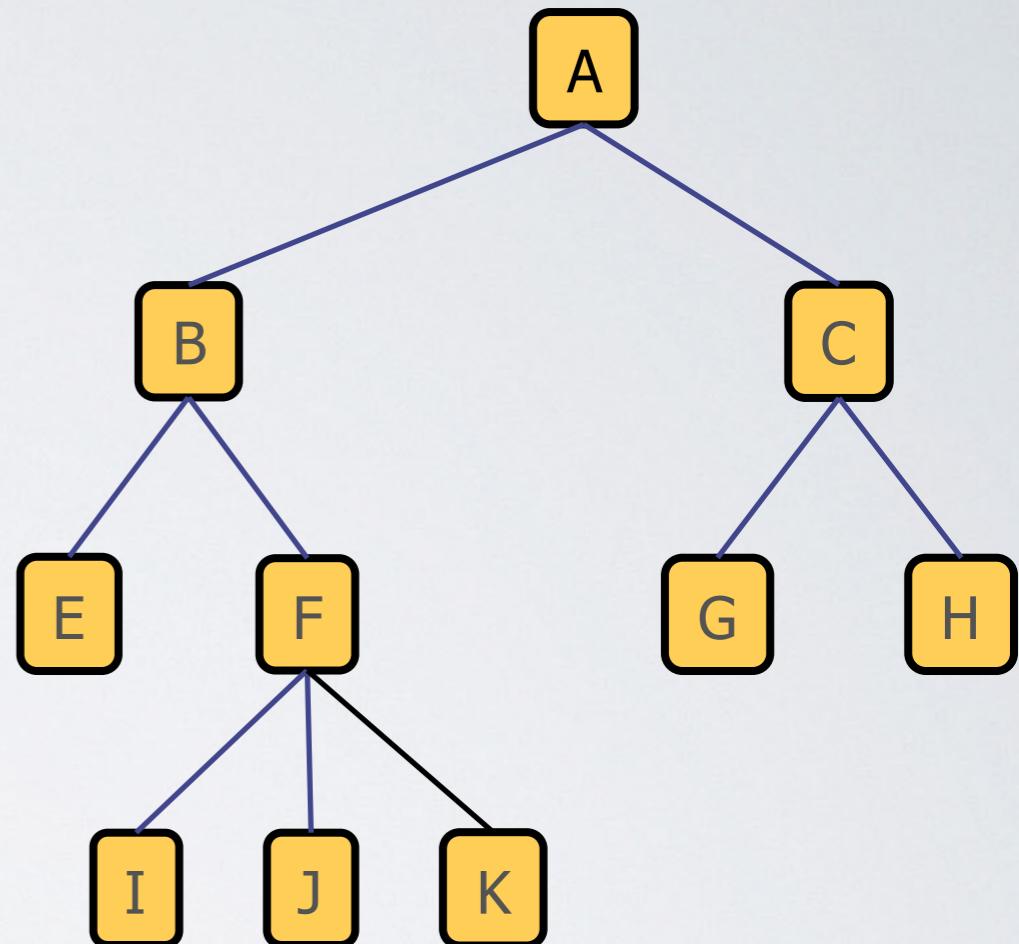
- ▶ Start at root
 - ▶ Visit both of its children first,
 - ▶ Then all of its grandchildren,
 - ▶ Then great-grandchildren
 - ▶ etc...
- ▶ Also known as
 - ▶ level-order traversal



A B C D E F G H I

Depth-First Traversal

- ▶ What if we grab youngest node in **S**?
 - ▶ we're doing LIFO...
 - ▶ so **S** is a stack!
 - ▶ Traversal w/ Stack gives us...
- ▶ Depth-first search
 - ▶ start from root
 - ▶ traverse each branch before backtracking
 - ▶ can produce different orders



A C H G B F K J I E
A B E F I J K C G H

Depth-First Traversal

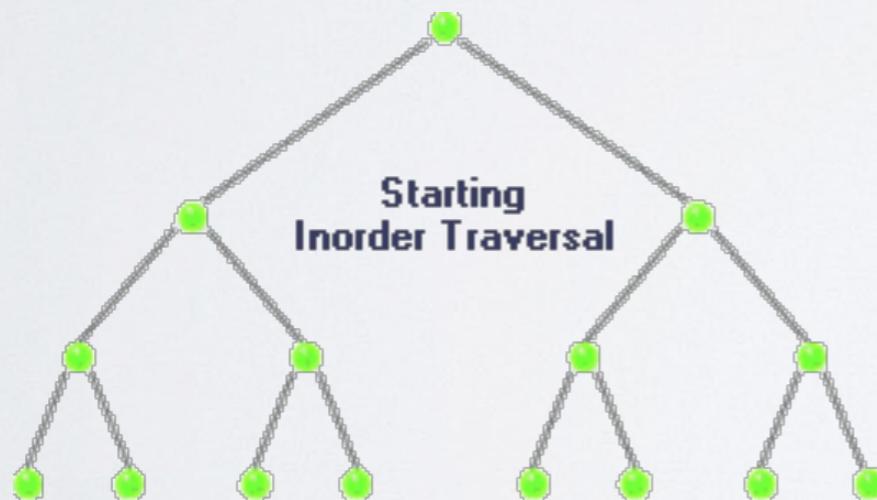
```
function dft(root):
    S = new Stack()
    push root
    while S is not empty
        node = S.pop()
        visit(node)
        push node's left & right children
```

- ▶ Why does Stack give DFT?
 - ▶ Stack guarantees entire branch will be visited before visiting another branch
- ▶ Children can be pushed on stack in any order

Recursive Depth-First Traversal

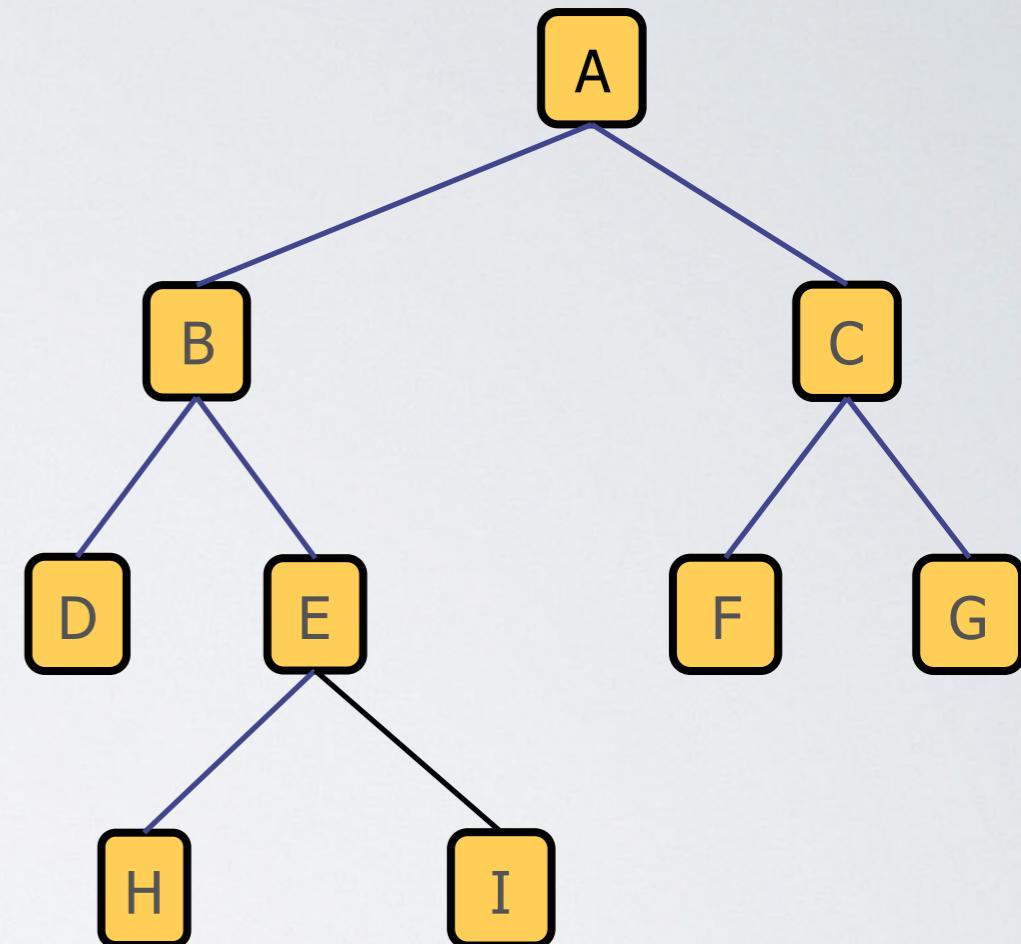
- ▶ DFT can be implemented recursively
- ▶ With recursion we can have 3 different orders
 - ▶ **pre-order:** visits node before visiting left and right children
 - ▶ V-L-R
 - ▶ **post-order:** visits each child before visiting node
 - ▶ L-R-V
 - ▶ **in-order:** visits left child, node and then right child
 - ▶ L-V-R
 - ▶ Notice that left child is always visited before right child

Depth-First Visualizations



Pre-order Traversal

```
function preorder(node):  
    visit(node)  
    if node has left child  
        preorder(node.left)  
    if node has right child  
        preorder(node.right)
```

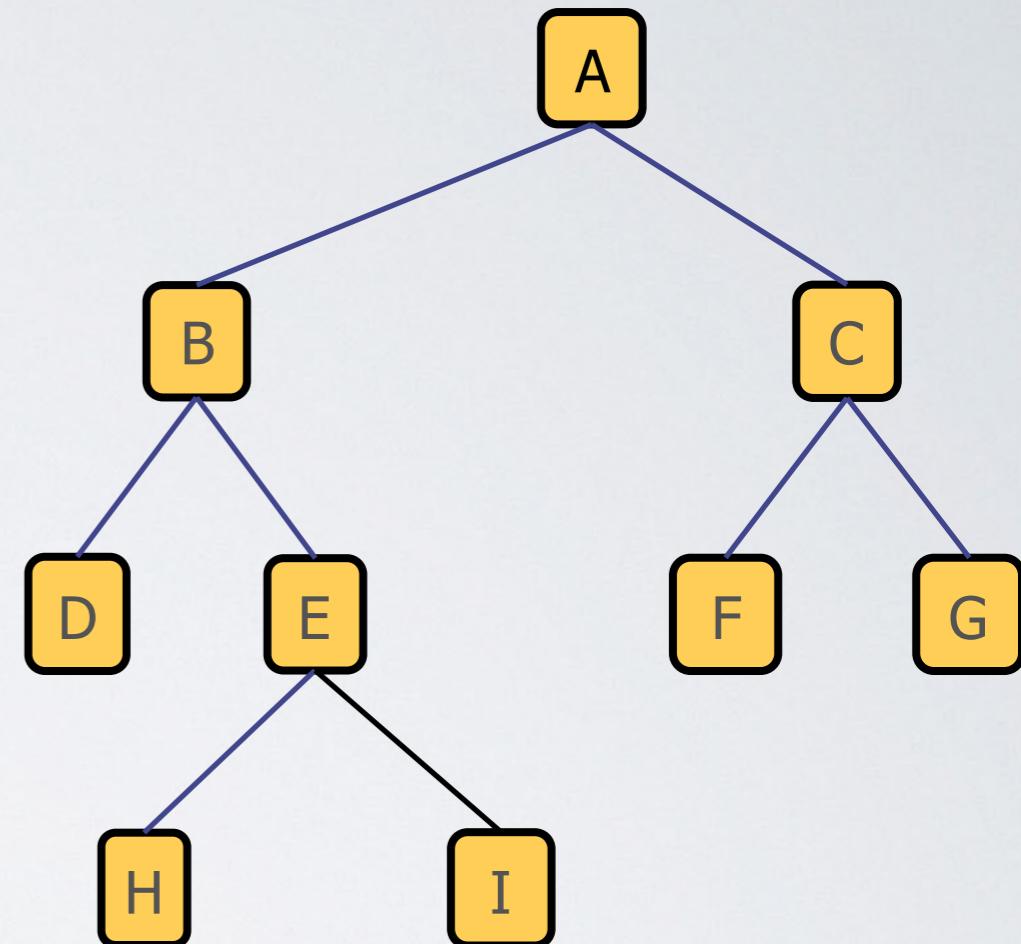


A B D E H I C F G

Note: like iterative DFT

Post-order Traversal

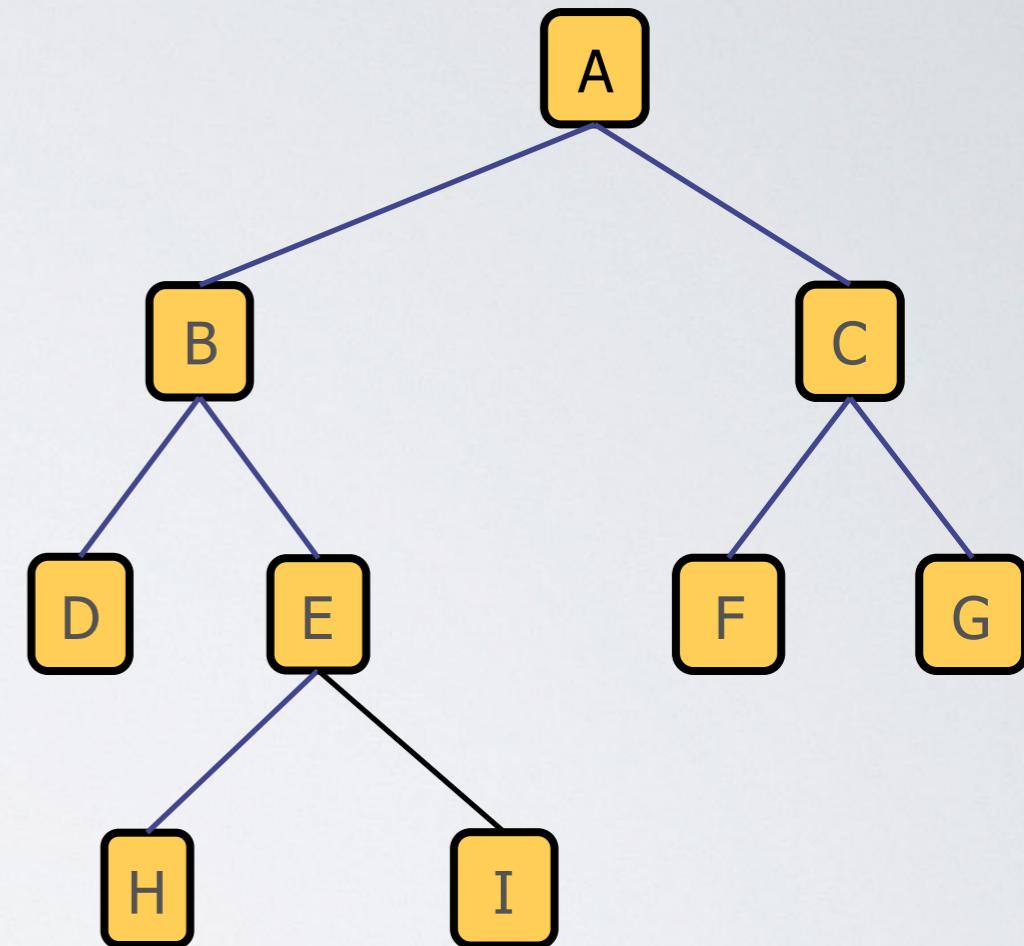
```
function postorder(node):  
    if node has left child  
        postorder(node.left)  
    if node has right child  
        postorder(node.right)  
    visit(node)
```



D H I E B F G C A

In-order Traversal

```
function inorder(node):  
    if node has left child  
        inorder(node.left)  
    visit(node)  
    if node has right child  
        inorder(node.right)
```



D B H E I A F C G

When to Use What Traversal?

- ▶ How do you know which traversal to use?
- ▶ Sometimes it doesn't matter
- ▶ Often one traversal makes solving problem easier

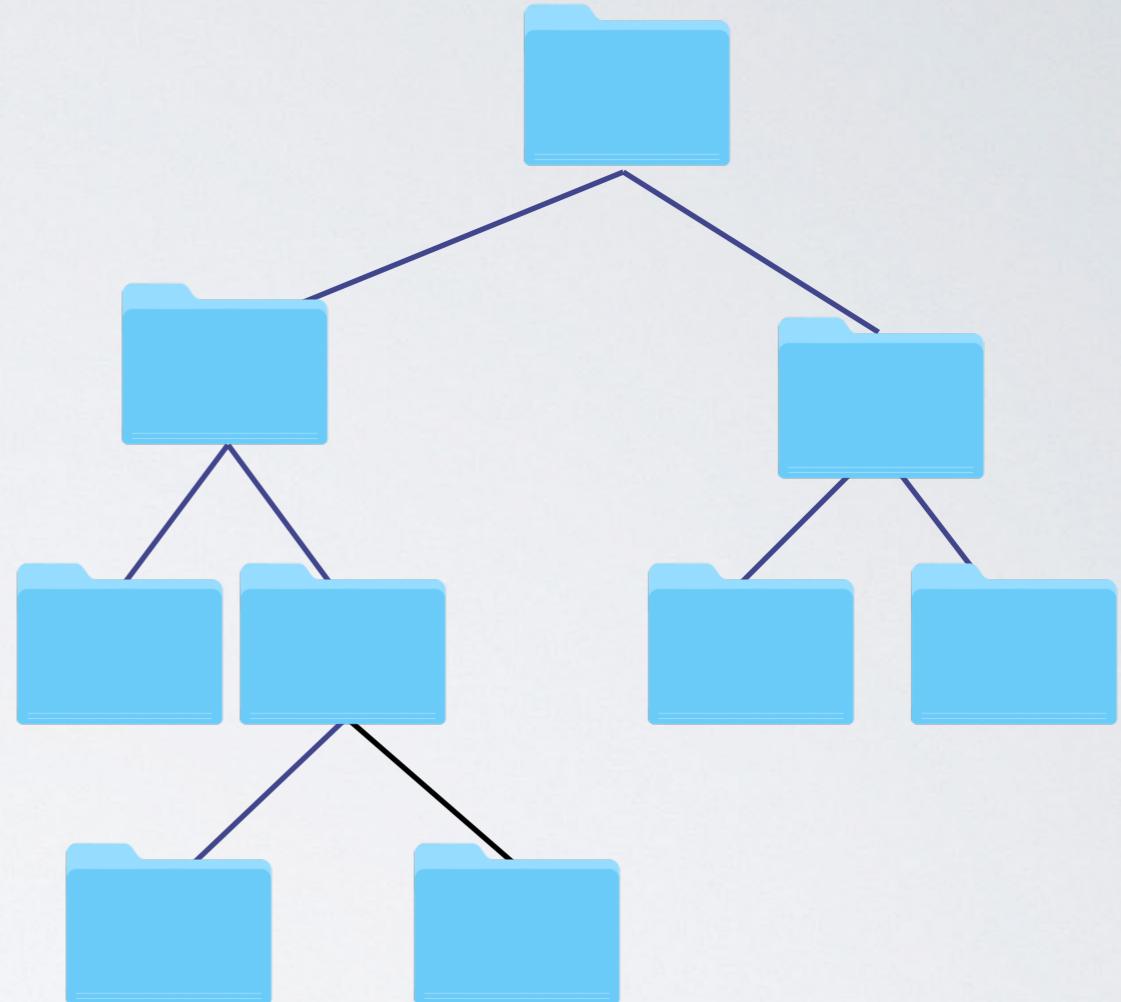
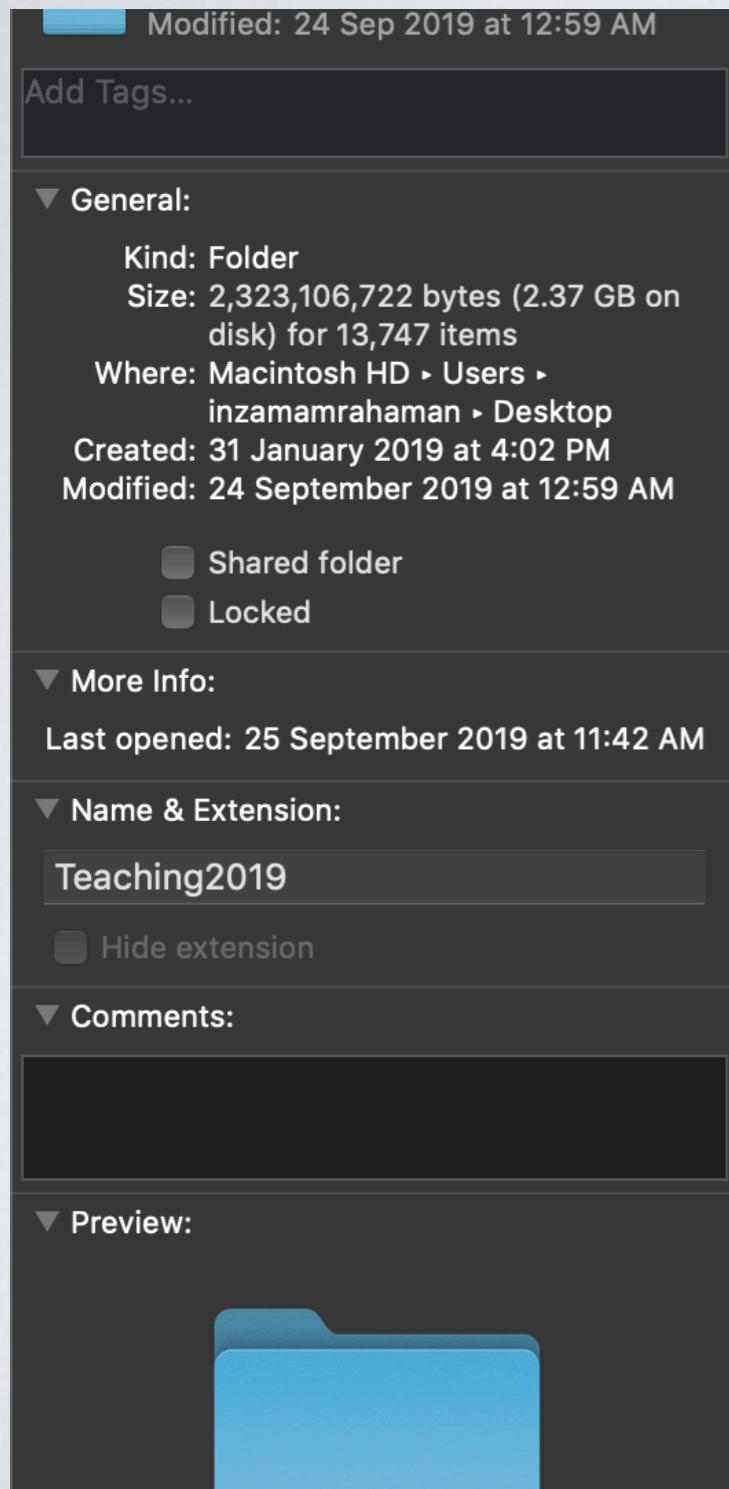
Tree Traversal Problem

- ▶ Decorating with number of descendants?
- ▶ **Post-order**
 - ▶ visits both children before node
 - ▶ easy to calculate # of descendants if you know # of descendants of both children
 - ▶ try writing pseudo-code for this based on code on the Github repo

Tree Traversal Problem

- ▶ Testing if tree is perfect
- ▶ **Breadth-first**
 - ▶ traverses tree level by level
 - ▶ keep track of how many nodes at level
 - ▶ each level should have twice as many as previous level
 - ▶ should try writing pseudocode for this

Tree Traversal Problem

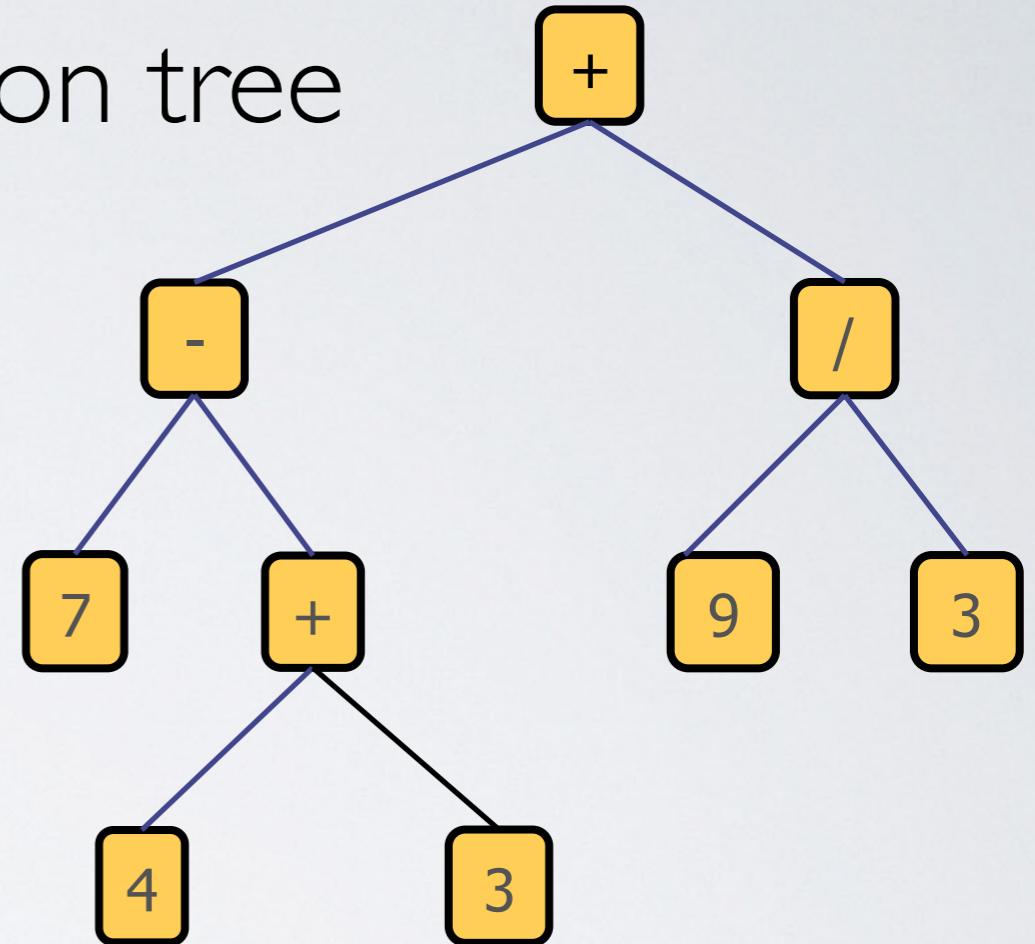


- ▶ Best traversal?
- ▶ **post-order:** need to know size of subfolders before you can compute size of a folder

Tree Traversals Problems

- ▶ Evaluate arithmetic expression tree

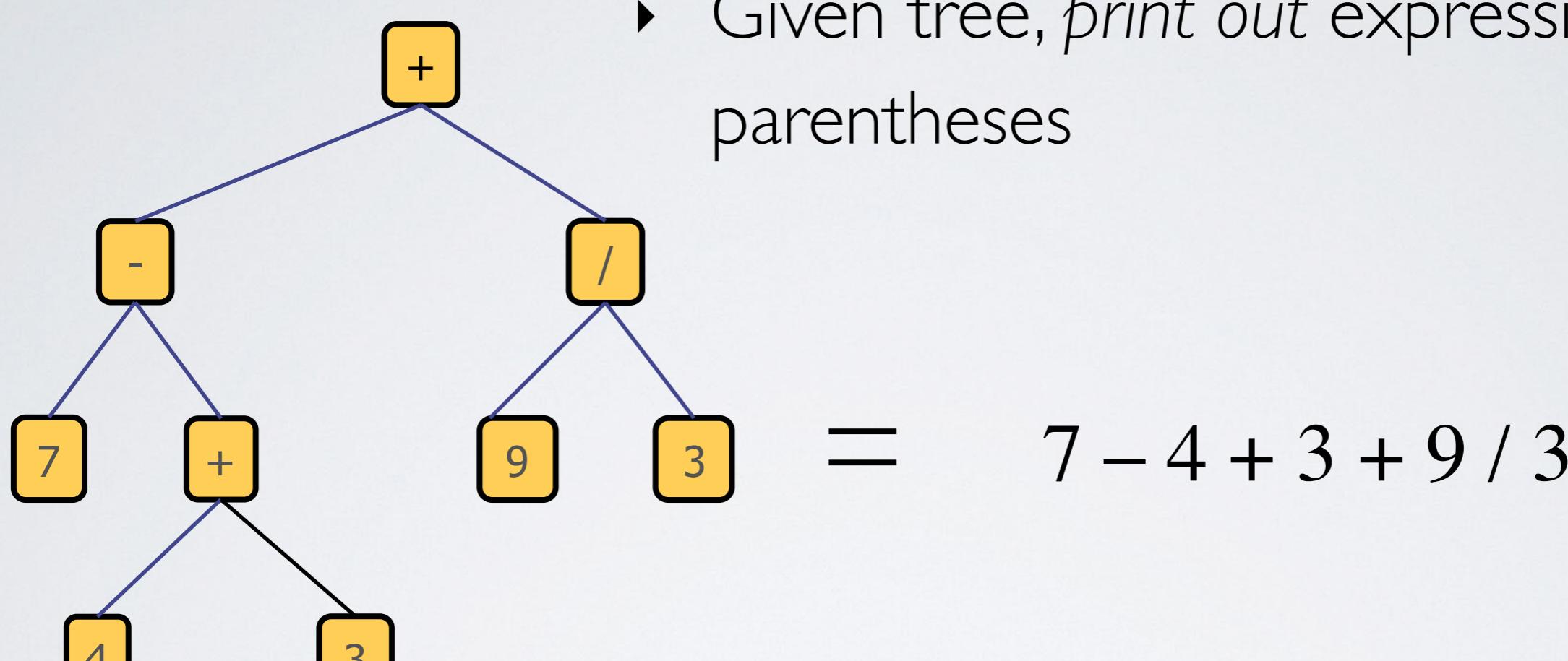
$$(7 - (4 + 3)) + (9 / 3) =$$



- ▶ Best traversal?

- ▶ **post-order:** to evaluate operation, you first need to evaluate sub-expression on each side
- ▶ What should you do when you get to a leaf?

Tree Traversals Problems

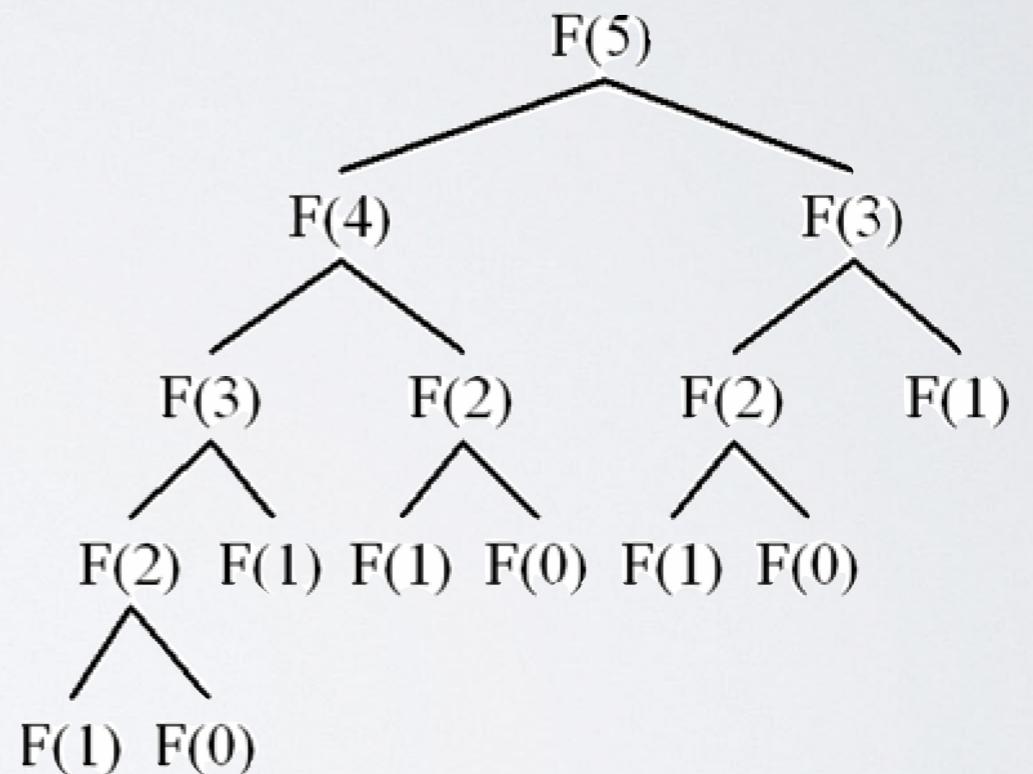


- ▶ Given tree, print out expression w/o parentheses
- ▶ Best traversal?
- ▶ **in-order:** gives nodes from left to right

Analyzing Binary Trees

- ▶ Many things can be modeled as binary trees
 - ▶ ex: Fibonacci recursive tree

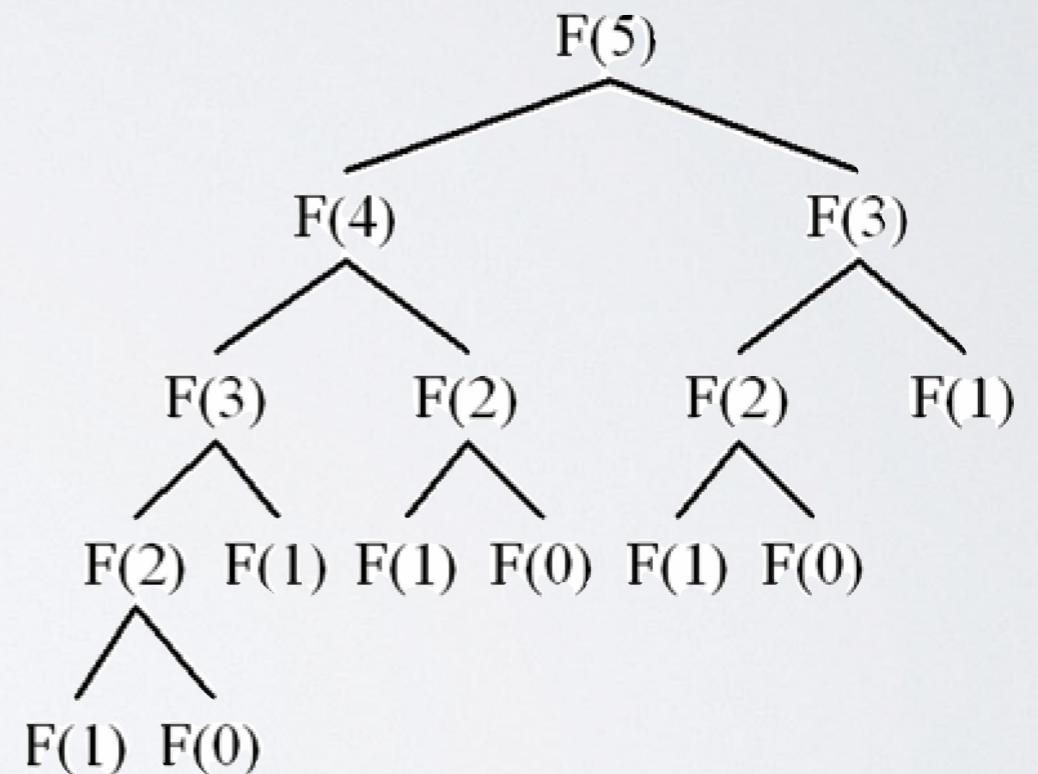
$$F(n) = F(n - 1) + F(n - 2)$$



Analyzing Binary Trees

- ▶ Many things can be modeled as binary trees
 - ▶ ex: Fibonacci recursive tree

$$F(n) = F(n - 1) + F(n - 2)$$



Analyzing Perfect Binary Trees

- ▶ Number of nodes in perfect binary tree of height h :
 - ▶ $2^{h+1} - 1$
- ▶ Height of a perfect binary tree with n nodes:
 - ▶ $\log_2(n+1) - 1$
- ▶ Number of leaves in perfect binary tree of height h :
 - ▶ 2^h
- ▶ Number of nodes in perfect binary tree with L leaves:
 - ▶ $2L - 1$

Induction on Perfect Binary Trees

- ▶ Can use (structural) induction to prove things about PBTs
- ▶ Using recursive definition of perfect binary trees
- ▶ Tree T is a perfect binary tree if
 - ▶ it has only one node
 - ▶ has root with left and right subtrees which are both perfect binary trees of same height
 - ▶ (if subtrees have height h , then T has height $h+1$)

Induction on non-Perfect Binary Trees

- ▶ Can use same techniques for balanced, though not perfect, binary trees
- ▶ However, instead of exact values, we often get bounds.
 - ▶ Property **x** of binary tree with **n** nodes is no less than or equal to ...

Example Inductive Proof on PBTs

- ▶ Prove $P(n)$:
 - ▶ number of nodes in a perfect binary tree of height n is $f(n) = 2^{n+1} - 1$
- ▶ Base case $P(0)$:
 - ▶ number of nodes in perfect binary tree of height 0 is 1 (by definition)
 - ▶ $f(0) = 2^{0+1} - 1 = 2 - 1 = 1$
- ▶ Inductive hypothesis:
 - ▶ assume $P(k)$ is true (for some $k \geq 0$)
 - ▶ in words: the number of nodes in perfect binary tree of height k is $f(k) = 2^{k+1} - 1$

Example Inductive Proof on PBTs

- ▶ Then prove that $P(k+1)$ is true:
 - ▶ Let T be any perfect binary tree of height $k+1$
 - ▶ By definition, T consists of root with two subtrees, L and R , which are both perfect binary trees of height k
 - ▶ By inductive hypothesis, L and R both have $2^{k+1}-1$ nodes
 - ▶ So total number of nodes in T is:
 - ▶ $2 * (2^{k+1}-1) + 1 = 2^{k+2}-2+1 = 2^{(k+1)+1}-1$
- ▶ Since we've proved
 - ▶ $P(0)$ is true
 - ▶ $P(k)$ implies $P(k+1)$ (for any $k \geq 0$)
 - ▶ It follows by induction that $P(n)$ is true for all $n \geq 0$

Example Proof on Perfect Binary Trees

- ▶ Sometimes, you don't need induction
- ▶ Consider that the height of a perfect binary tree with n nodes is $\log_2(n+1) - 1$
- ▶ Can you think of a way to prove this without induction?
- ▶ Hint: Add up all the number of nodes on each level of the tree to get the total number of nodes

References

- ▶ Some slides pulled from Brown's CS16
- ▶ Sedgewick