Tree & Binary Tree

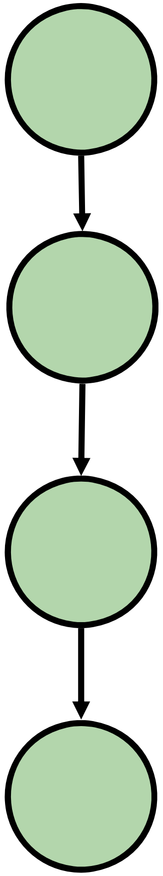
IT5003: Data Structures and Algorithms
(**AY2019/20 Semester 1**)

Lecture Overview

- **Trees**
  - Terminology
  - Definitions

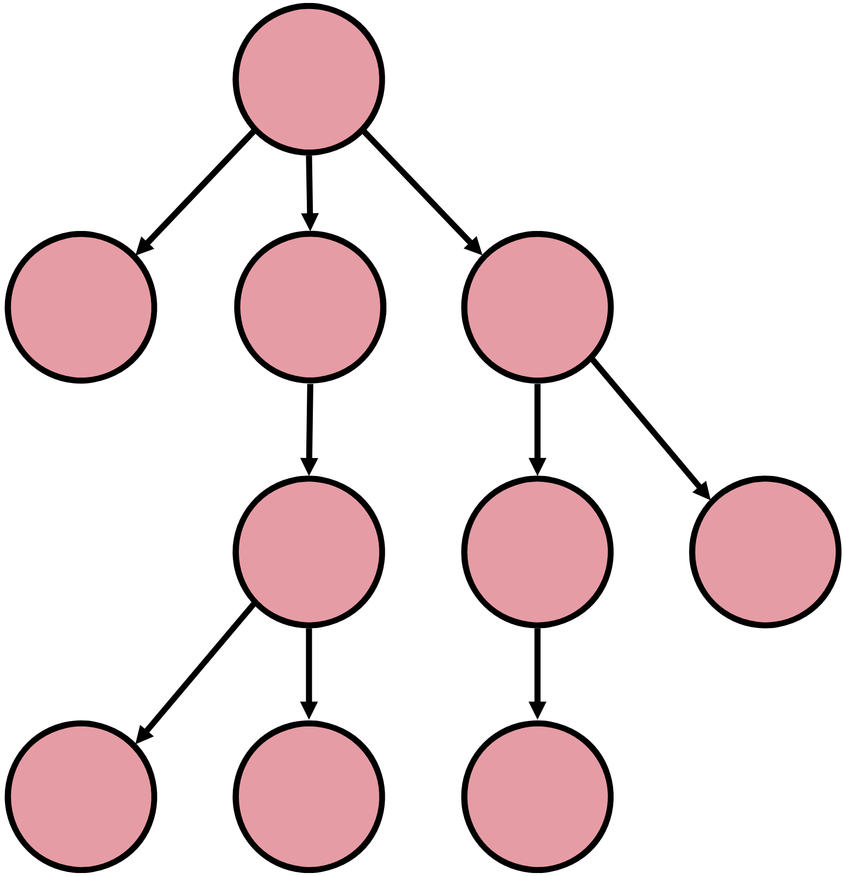
- **Binary Tree**
  - Definitions
  - Implementations
  - Major Operations and Traversals

Motivation: Why Trees?



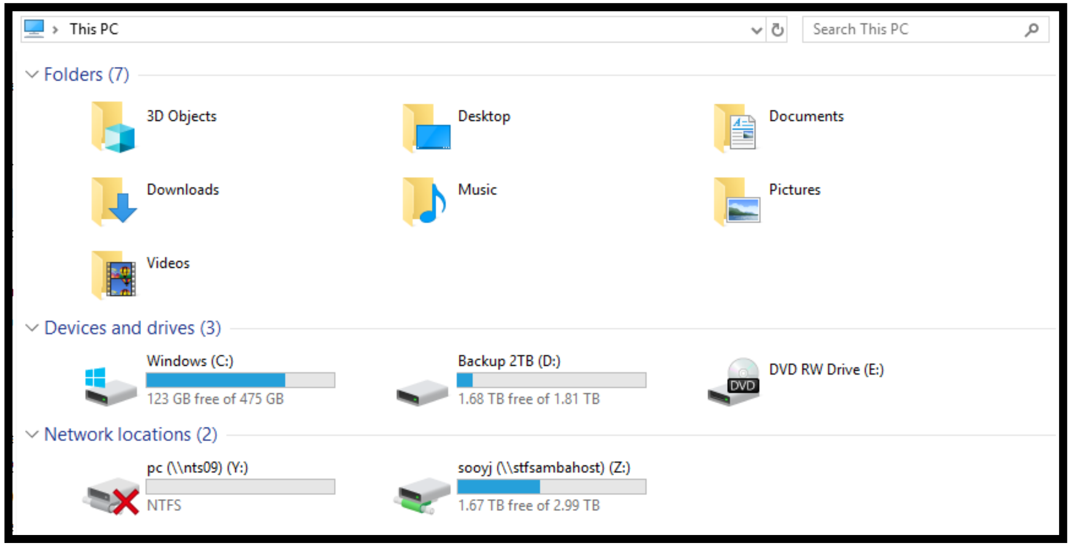
- Linked list is suitable to represent a collection of information with no hierarchical relationship
  - i.e. each node are "similar" / "equal" in stature
- With only "next" (and "previous") references, complex relationship is hard to represent in linked list

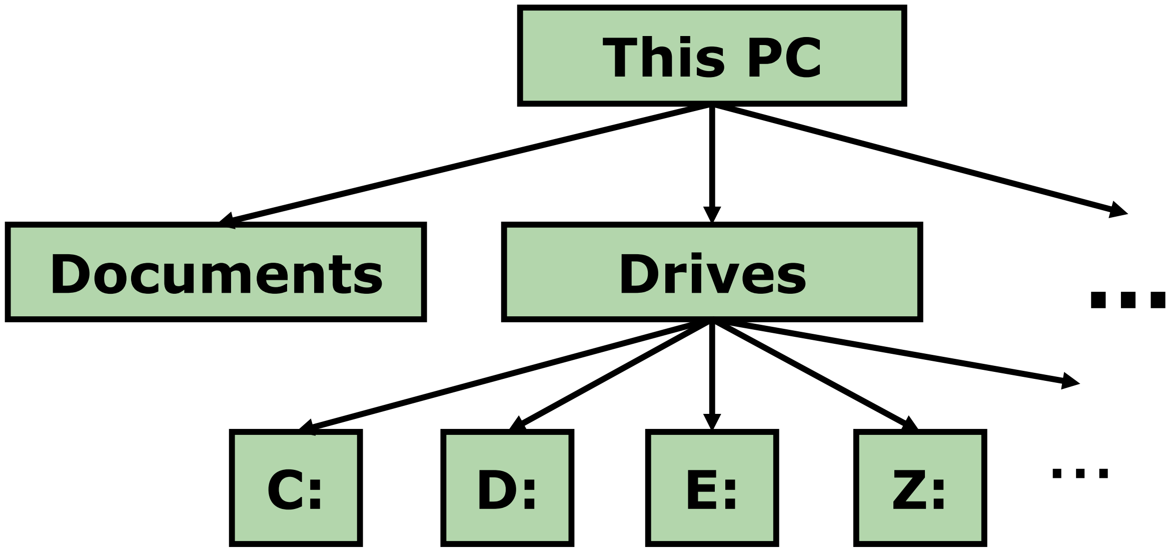
A Tree as a Data Structure



- Similar to a tree in the real world
  - Note: shown upside down
- Intuitive way to represent relationship and hierarchy
- Example:
  - Family tree, function call tree, folder and files on a disk etc

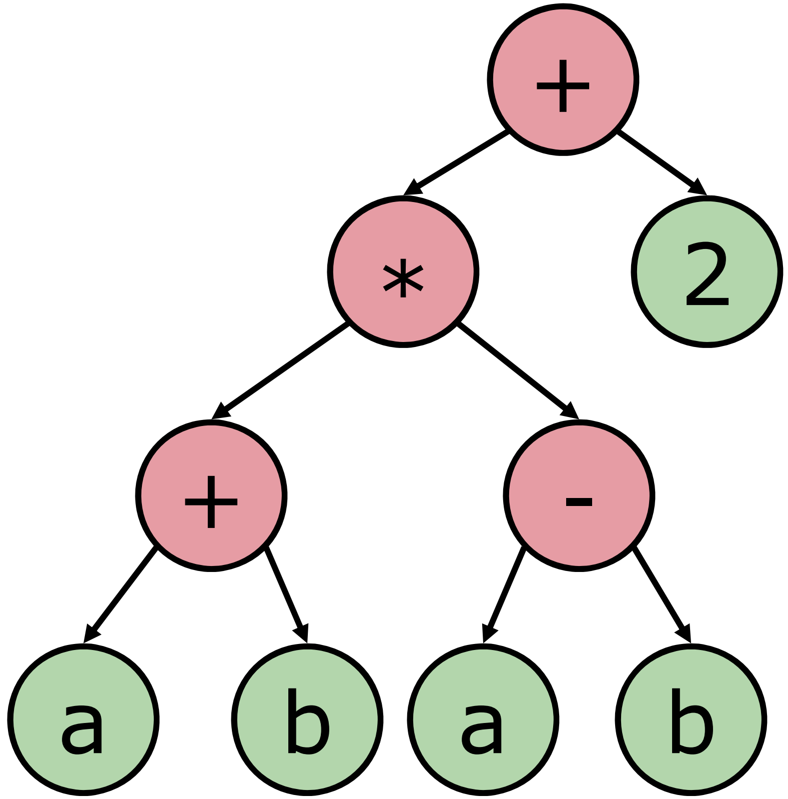
Tree Example: File Systems





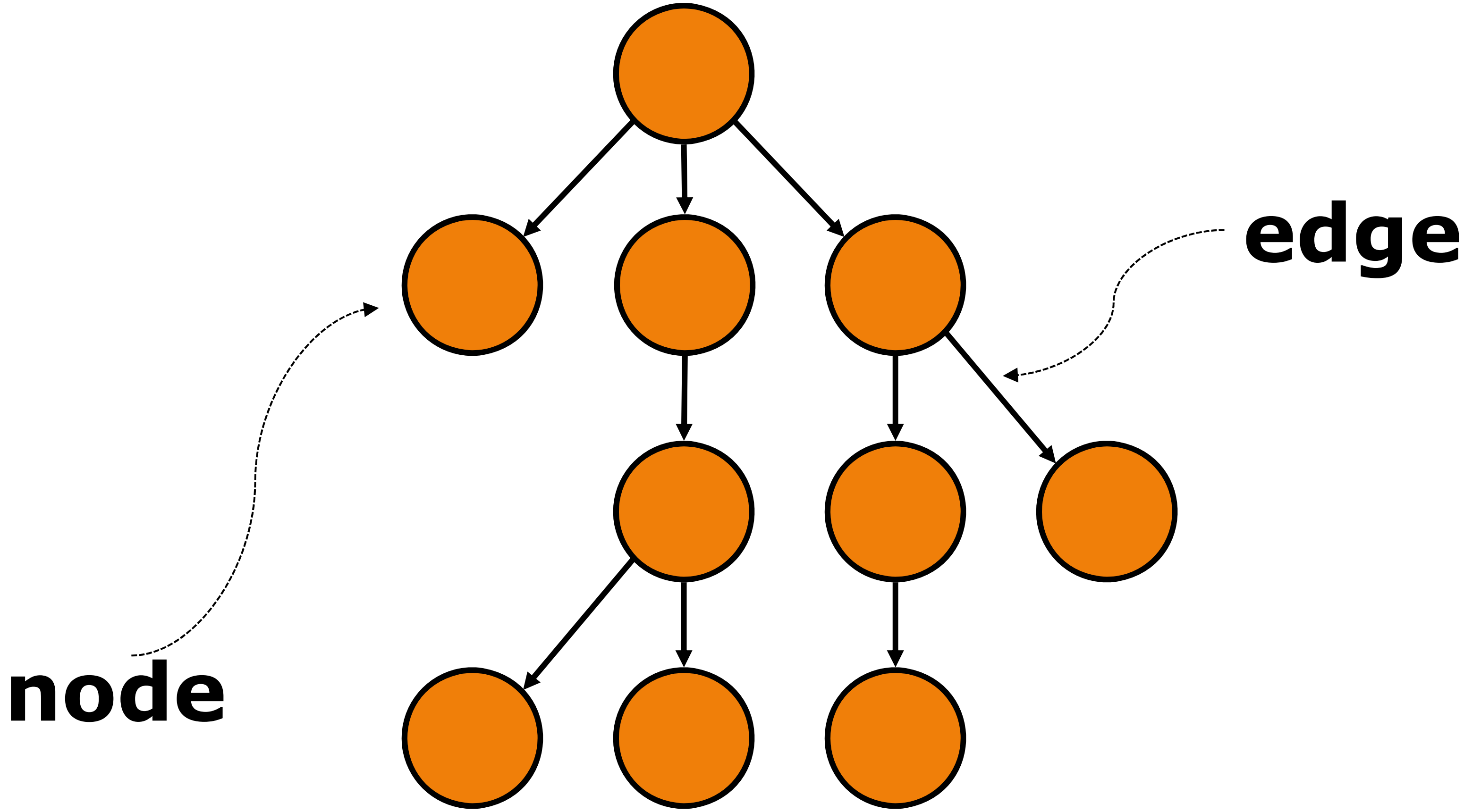
Tree Example: **Arithmetic Expression**

(a+b) * (a-b) + 2



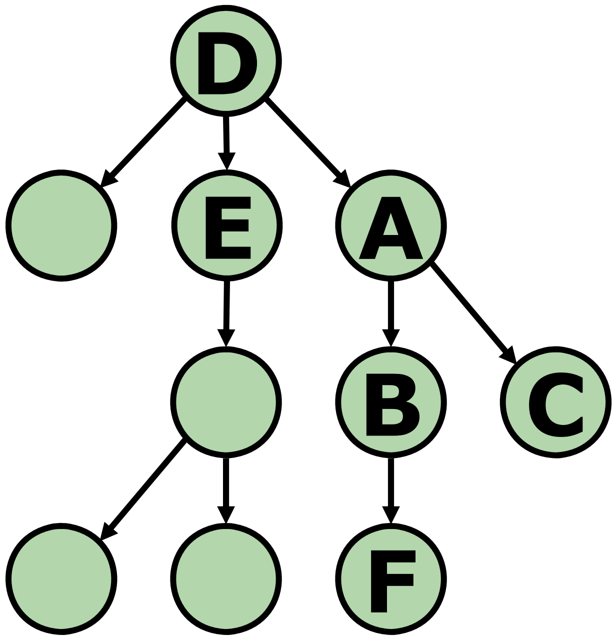
Q: How do you construct such a tree from a given arithmetic expression?

Definitions: **Edge & Node**



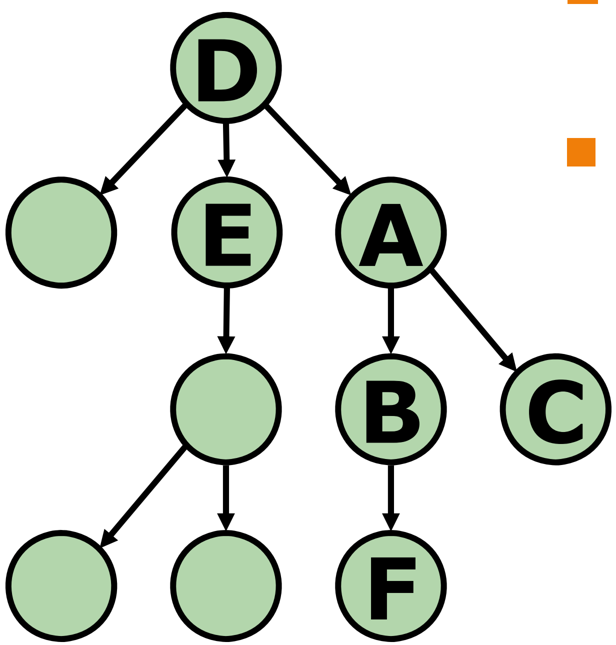
- Data objects (the circles) in a tree are called **nodes**
- Links between nodes are called **edges**

Relationship: Parent/Child/Sibling



- A is a **parent** of B and C
- B and C are **children** of A
- B and C are **siblings**
  (with the same parent A)

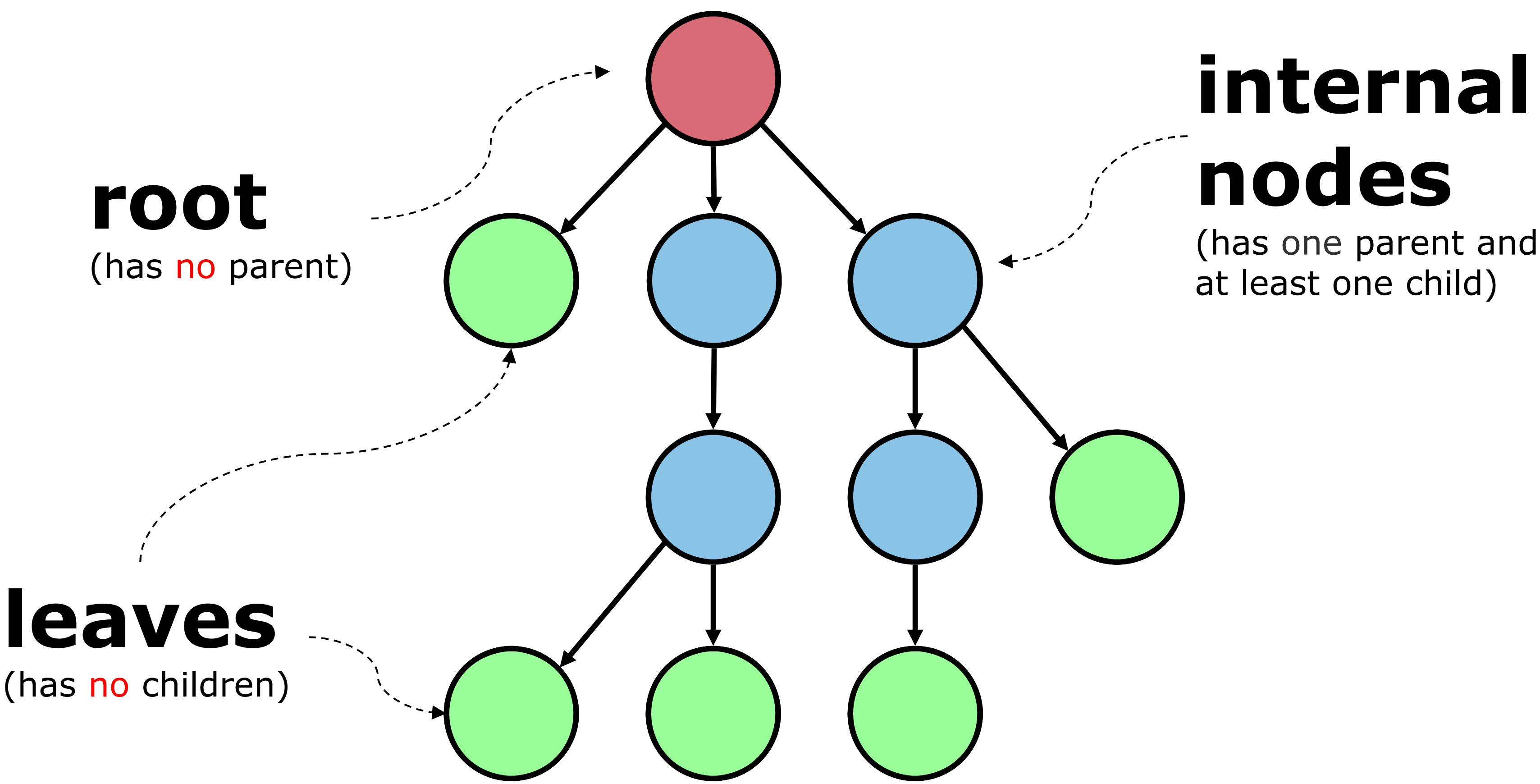
Relationship: Ancestor / Descendant



- D is an **ancestor** of B
- B is a **descendant** of A and D

- **Definition:**
  Node X is an **ancestor** of node Y if
  1) X is a parent of Y, **OR**
  2) X is a parent of some node Z and Z is an ancestor of Y

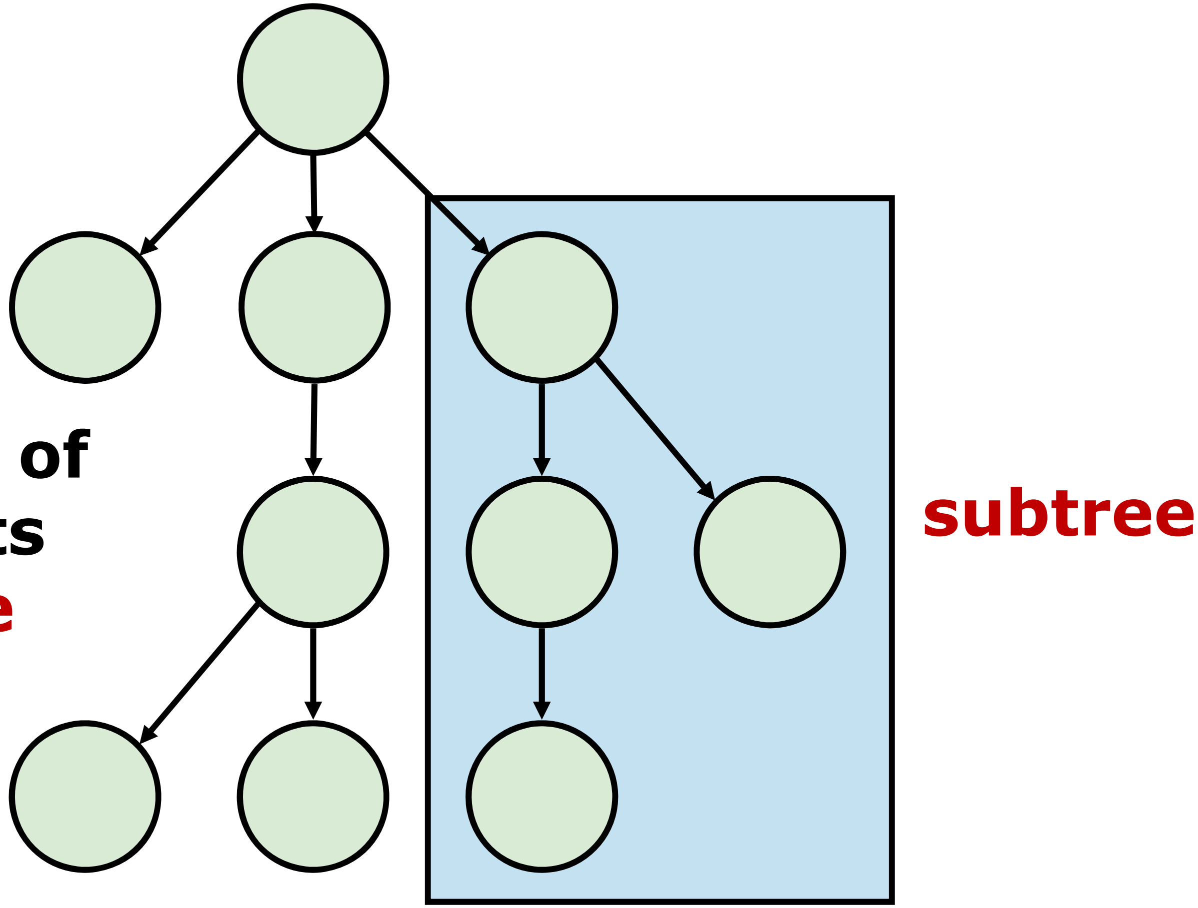
Tree Node Types



- Every node (except the root) of a tree has **one parent**
- A node with no children is a **leaf node**

Definition: Subtree

A node and all of its descendants form a subtree



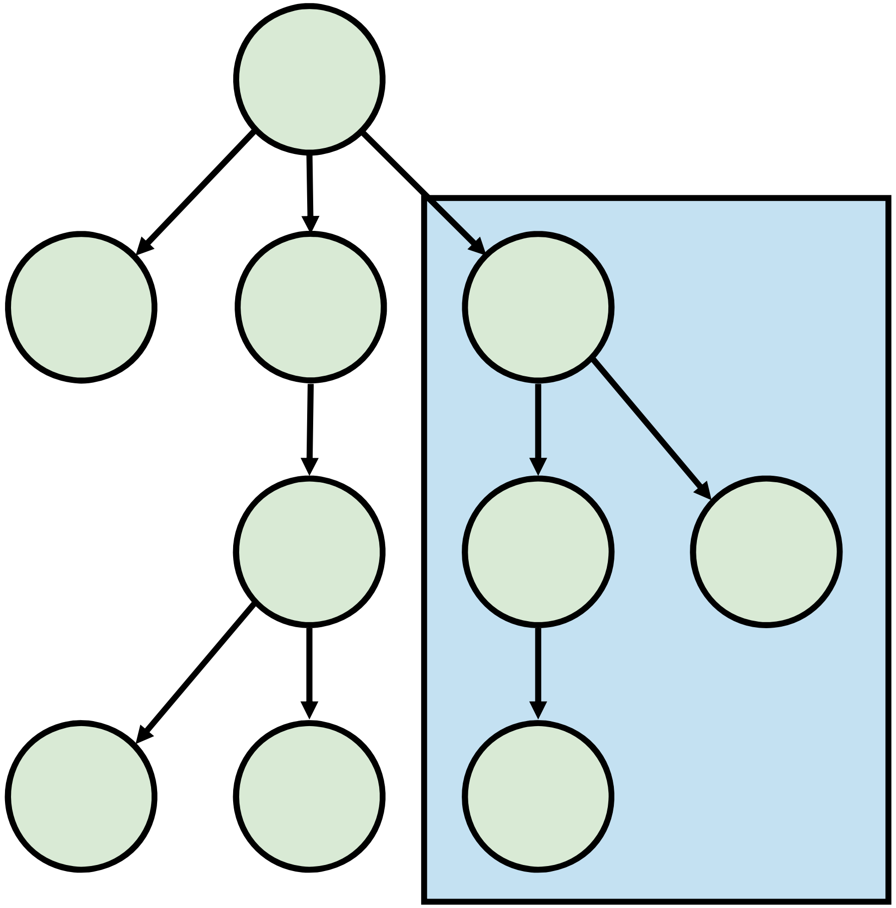
- Question: Can a leaf be a subtree?

Definition: Tree

A **tree** is either

1) nothing (i.e. **empty tree**), **OR**

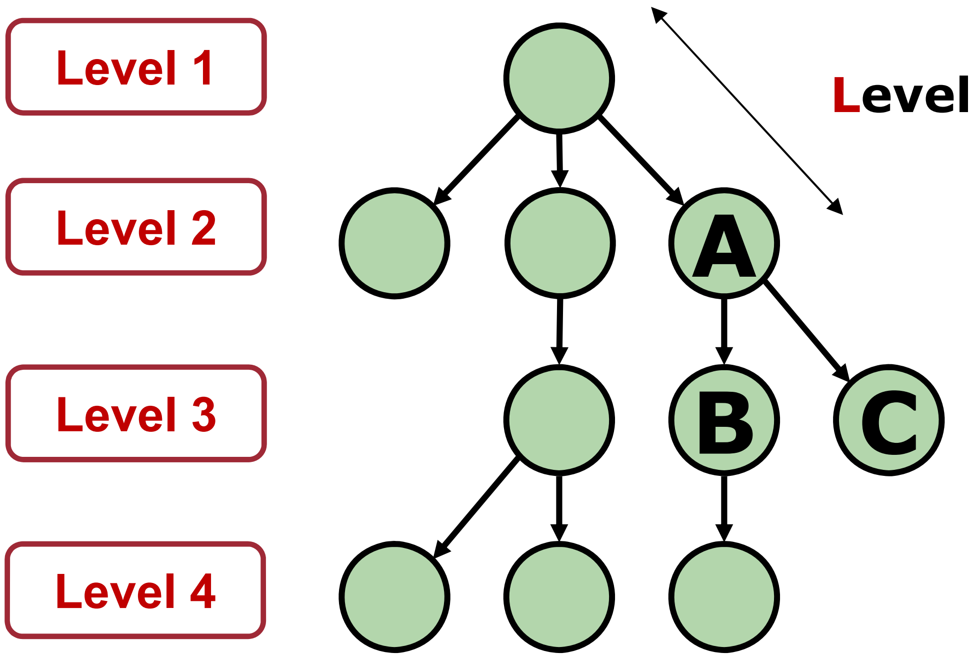
2) A node, with some set of subtrees, each of which is a **tree**...



➔ **Tree is recursive!**

Definition: **Node Level**

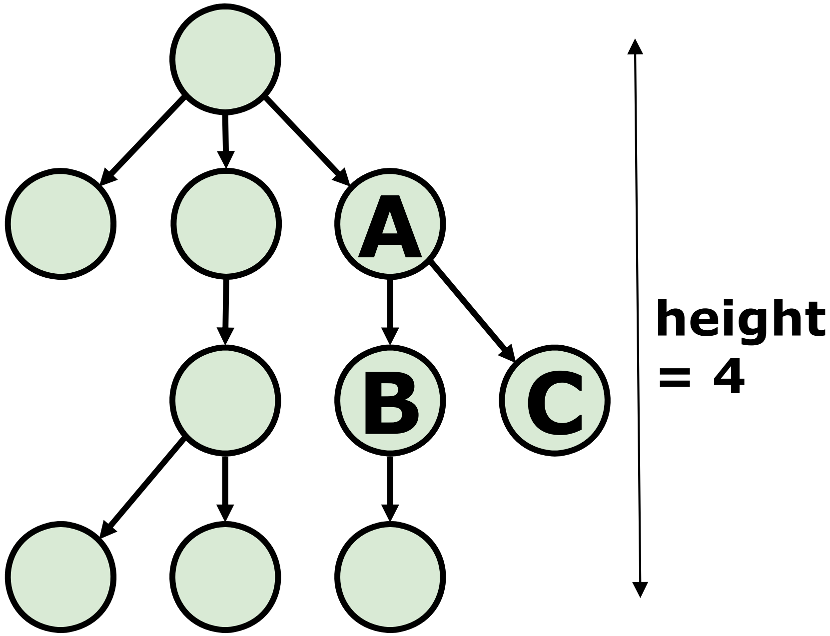
- **Number of nodes on the path from the root to the node**
  - e.g. Level of **root** is **1**
  - e.g. Level of node **A** is **2**



Definition: **Tree Height**

Tree Height

- **0** if the tree is empty, OR
- **Maximum level of the nodes** in the tree is the **height** of the tree

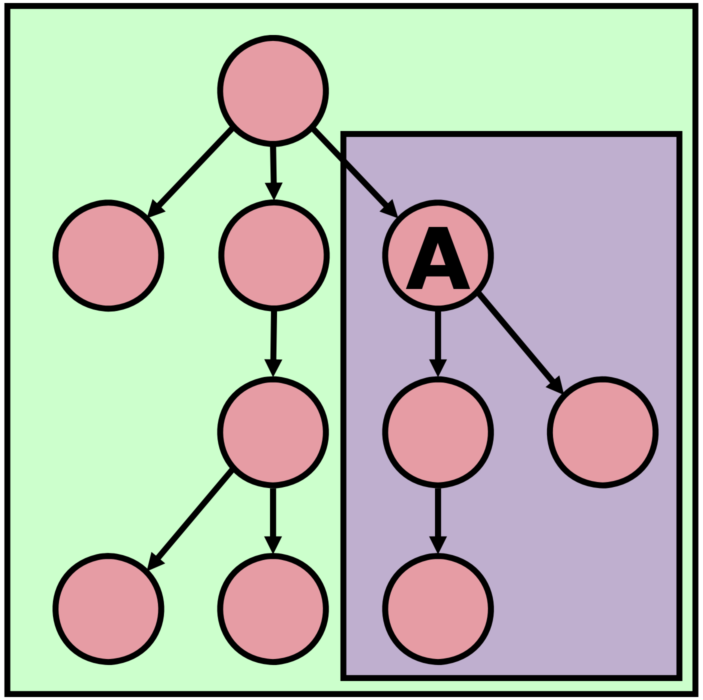


Definition: **Tree Size**

Tree Size

- Number of nodes in the tree

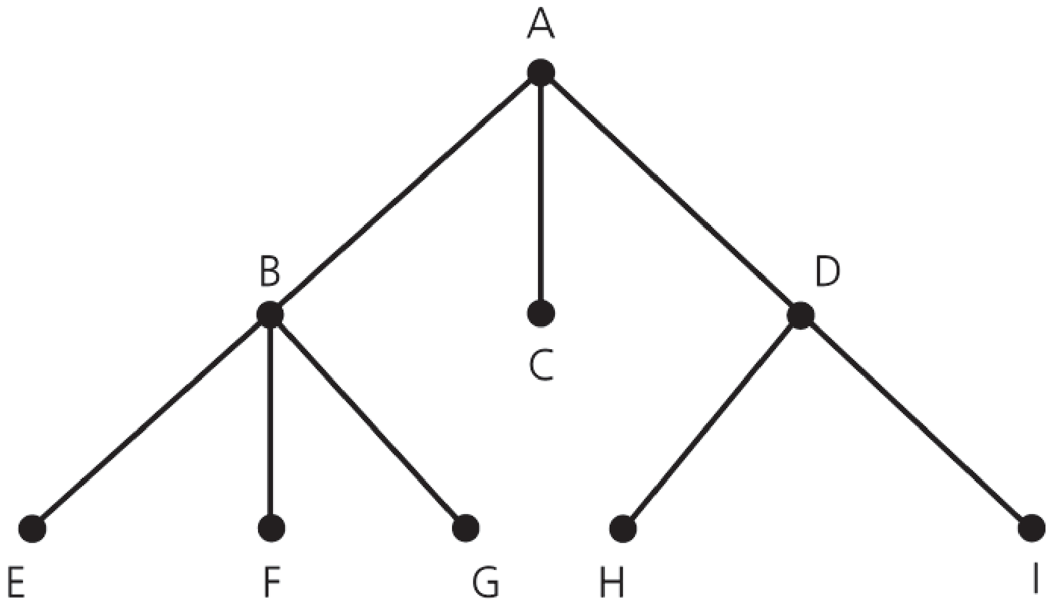
- Example:
  - The size of this tree is **10**
  - The size of the subtree rooted at **A** is **4**



Definition: N-ary Tree

An **N-ary Tree**

- Nodes in the tree can have no more than **N** children
- Subset of the general trees



This is a 3-ary (ternary) tree

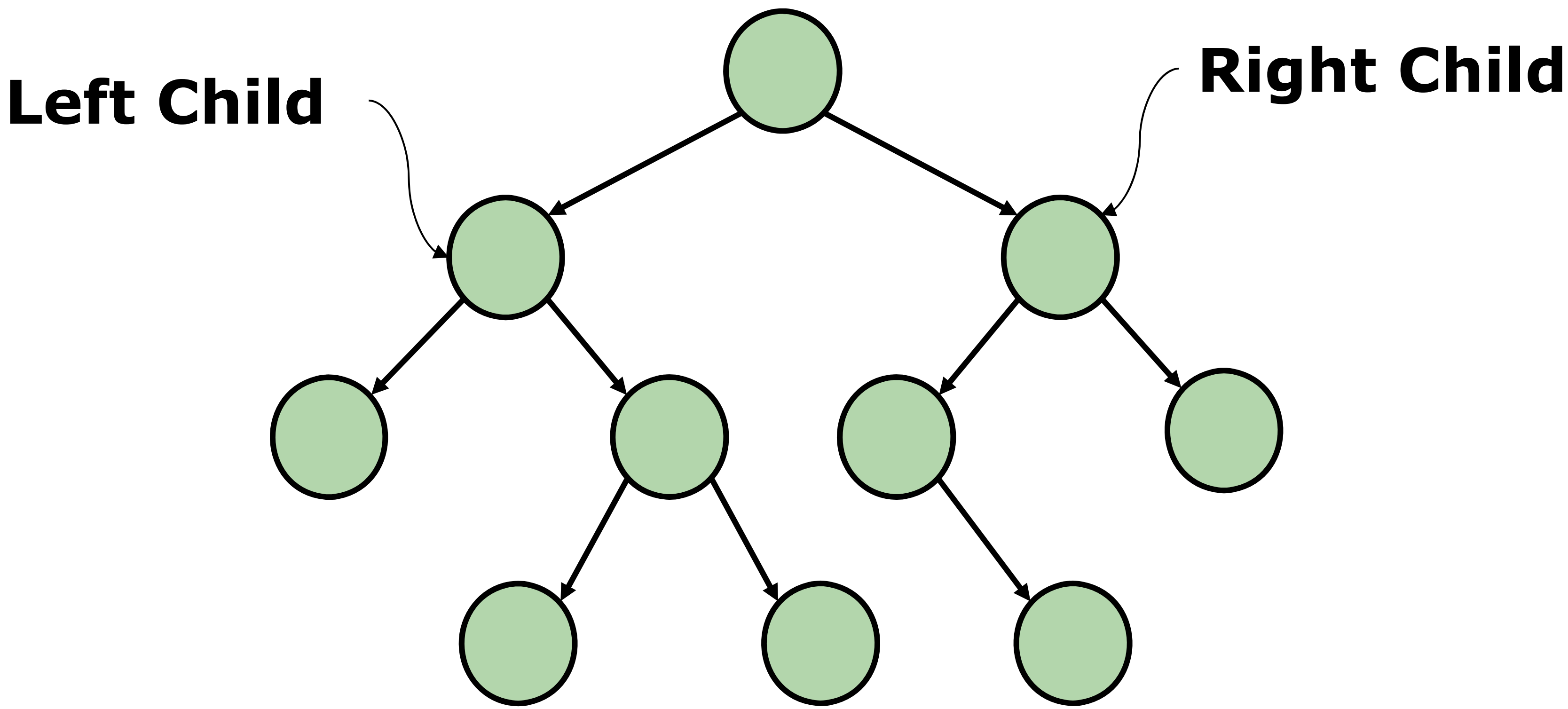
Each node has **at most 2 ordered** children

BINARY TREES

Definition: Binary Tree

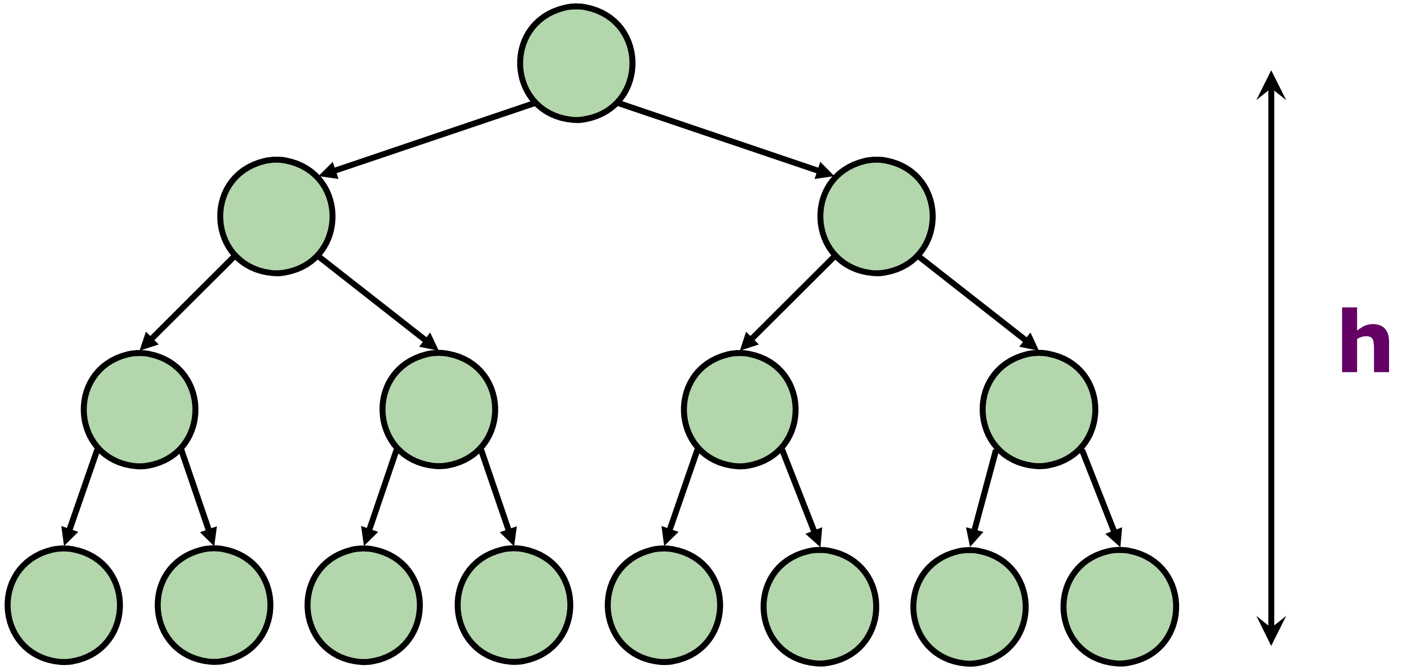
Binary Tree is

1) An empty tree **OR**
2) A node with at most 2 **ordered children**



Definition: Full Binary Tree

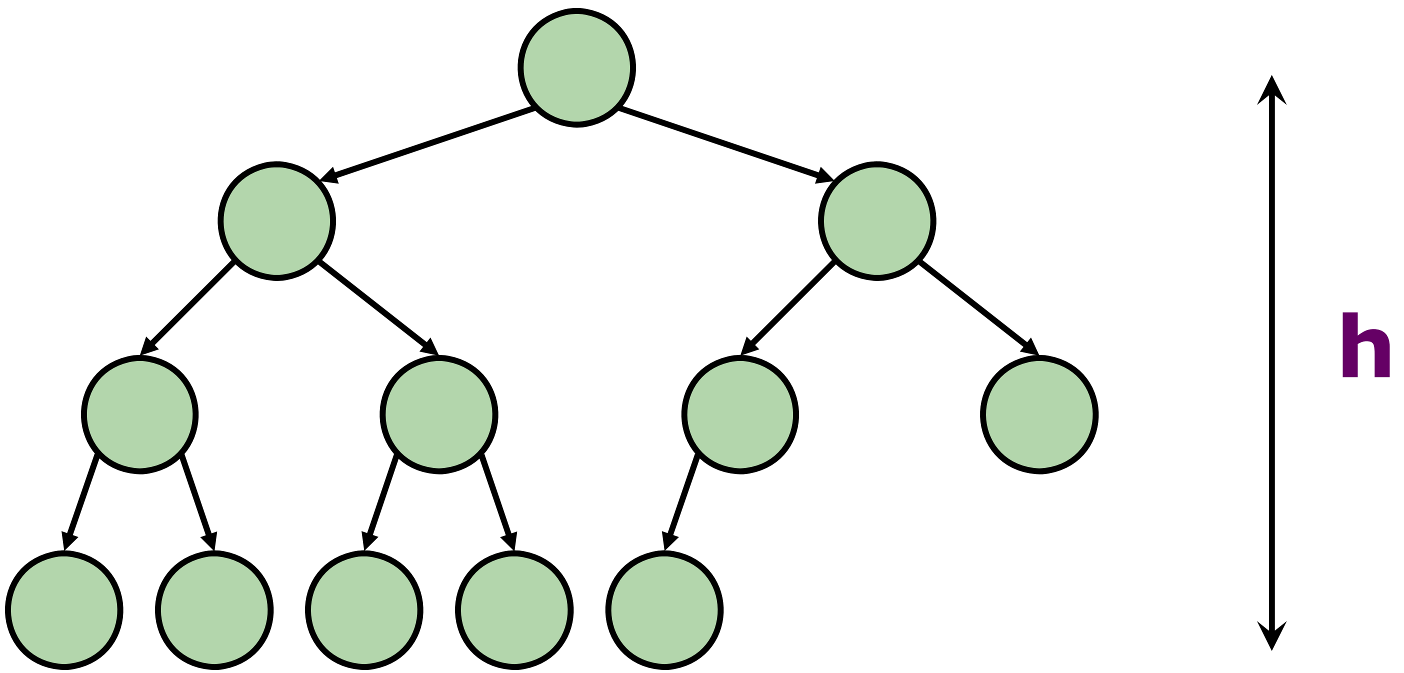
- All nodes at a **level < h** have two children
  - h is the height of the tree



Question:
How about this definition "***all nodes except the leaf nodes have 2 children***"?

Definition: Complete Binary Tree

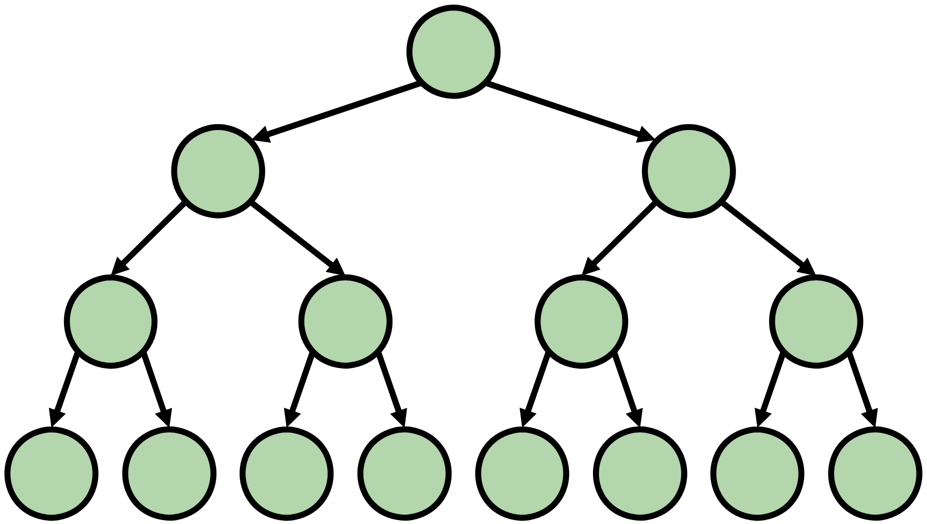
- Full down to level **h-1**
- level **h** filled in from **left to right**



Full Binary Tree Property

- Number of nodes in a full binary tree of height **h** is **2^h - 1**

  ➔ height of a full binary tree is **O(log N)**

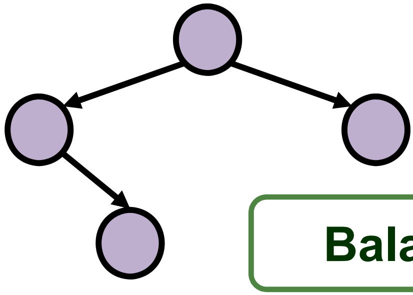


Question:
How many nodes in a complete binary tree of height **h**?

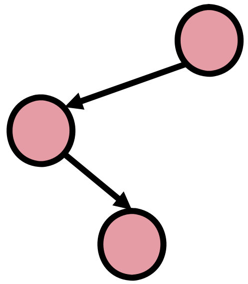
Definition: Balanced Binary Tree

Balanced Binary Tree

- For all nodes:
  - **Right subtree height** differs from the **left subtree height** by **no more than 1**



Balanced



Not Balanced

- Hence:
  - **Full** binary trees are **complete**
  - **Complete** binary trees are **balanced**

Reference/ Pointer based & Array based

REPRESENTATION

Before We Start ……

- In this section:
  - Study the low level representation of a tree:
    - How to represent a tree node? an edge? etc
  - Study high level pseudo code on important operations:
    - What are the steps to count all the tree nodes?
- Keep in mind that:
  - It is not a full blown ADT (not yet…..)
  - You can try to implement the pseudo code for different representations
    - Not necessary but good exercise

Array Based: Tree Node

- A tree node contains:
  - The **data** item
  - The **left** and **right** child node (if any)
    - Different representation mainly differs in **how to specify the left / right** child node

- **Version 1:**
  - We keep track of left and right **indices** in the array

```
class ArrayTreeNode:
    def __init__(self, item, leftIdx = -1, rightIdx = -1):
        self.item = item
        self.leftIdx = leftIdx
        self.rightIdx = rightIdx
```

Array Based v1: **Left/Right Indices**

- Given the root node is at index 2, can you reconstruct the tree using the information below?

	0	1	2	3	4	5
item	**L**	**I**	**A**	**R**	**?**	**S**
leftIdx	**-1**	**-1**	**0**	**1**	**-1**	**-1**
rightIdx	**-1**	**5**	**3**	**-1**	**-1**	**-1**

Array Based v1: **Design Considerations**

- Interesting questions:
  - How do we handle deleted nodes?
  - How do we keep track of "free" nodes?

_array

	0	1	2	3	4	5
item	1	2	3	4	5	-1
leftIdx	-1	-1	-1	-1	-1	-1
rightIdx	-1	-1	-1	-1	-1	-1

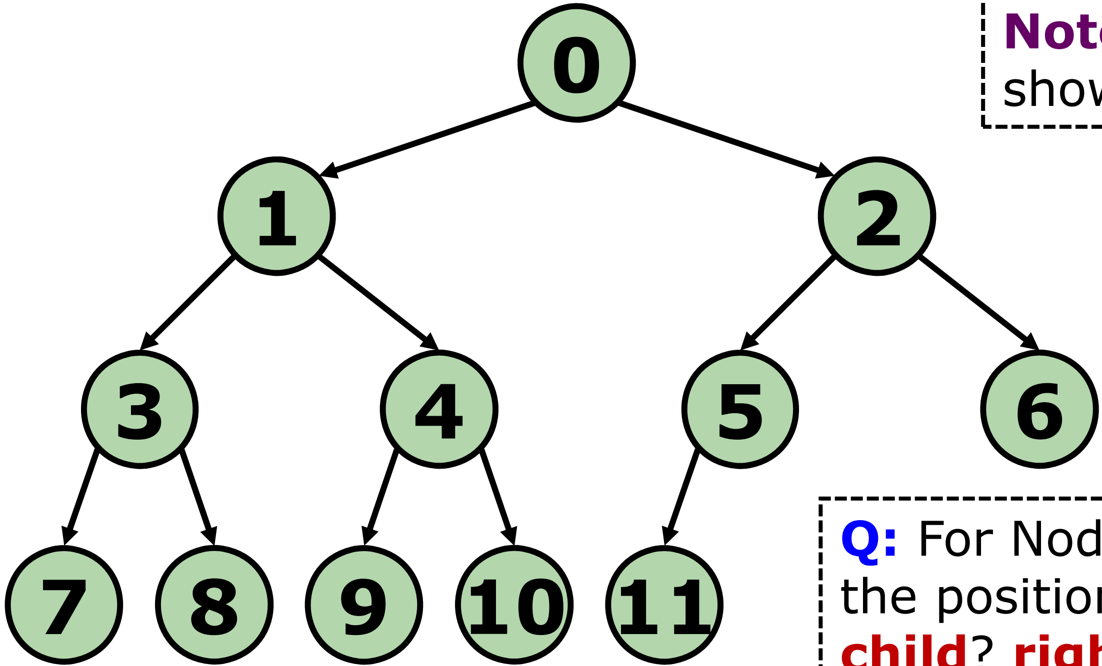
```
_root = ...
_free = 0
```

- Ideas:
  - Have a "free list" for free node
  - Reuse the "data" in a free node to maintain the list
    - e.g. "item" = 1 ➔ next free node is at index 1, etc

Array Based v2: Fixed Index

- Give **fixed and unique location** for **every node in a binary tree**
- **Idea:**
  - Generate a unique numbering layer by layer from root onwards using a **full binary tree**

Note: The unique index is shown, not the data item

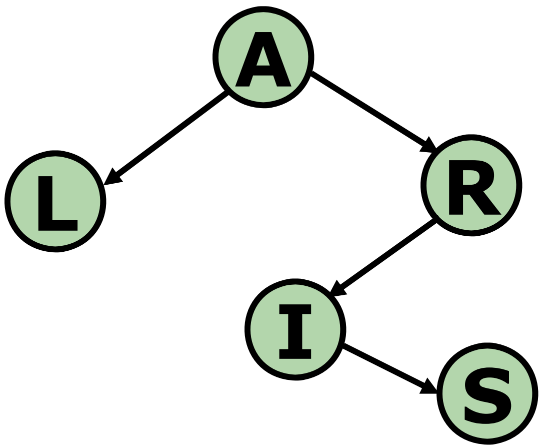


Q: For Node at index **I**, what is the position of its **parent**? **left child**? **right child**?

Array Based v2: Fixed Index

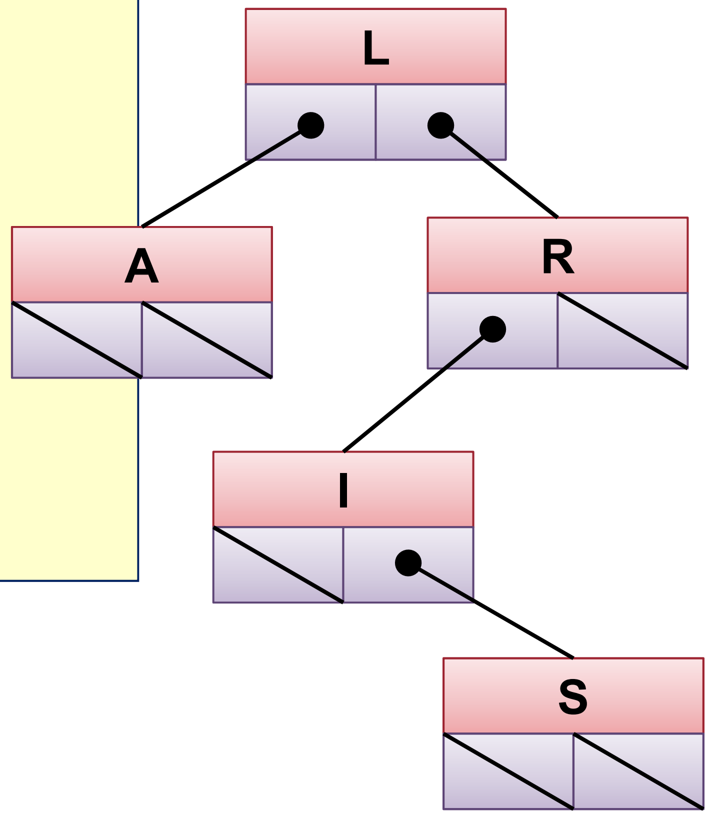
- **Pros:**
  - Each node now is very simple: Just store the item!
  - Very easy to get to parent / child nodes
- **Cons:**
  - Can have many empty nodes

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
item	A	L	R	---	---	I	---	---	---	---	---	---	S	---



Reference Based: Basic Sketch

```
class RefTreeNode:
    def __init__(self, item, \
        leftPtr = None, rightPtr = None):
        self.item = item
        self.leftPtr = leftPtr
        self.rightPtr = rightPtr


class BinaryTreeRef:
    #Constructor
    def __init__(self):
        self._root = None
        self._size = 0
```



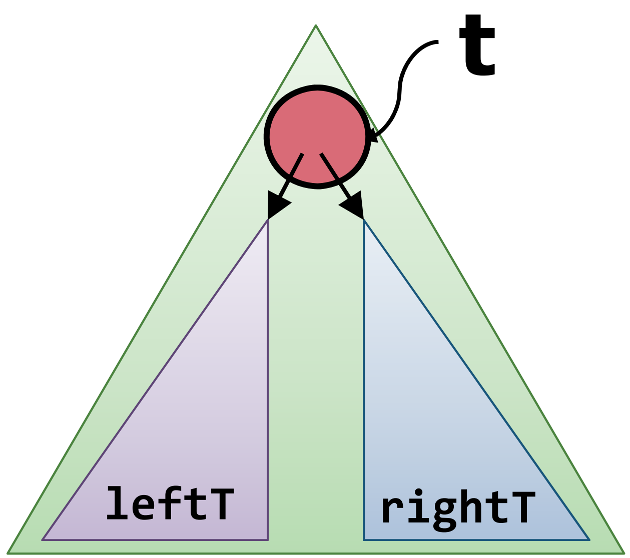
Recursion & Recursion!

MAJOR OPERATIONS

General Guideline

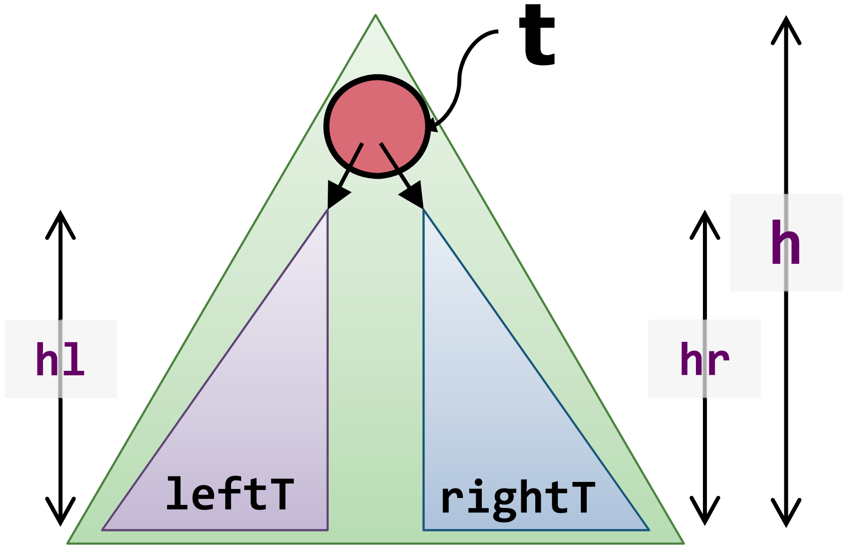
- **Most** tree operations are recursive in nature
  - Natural match to the recursive data structure
- Recursion requires:
  a. Break down to a small problem of the same type
  b. Build up on partial result
- Recursion in a Binary Tree:

```
def solve( T ):
    if T is empty:
        //base case
    else:
        solve( T➔leftT )
        solve( T➔rightT )
        build result
```



Tree Height: Pseudo-Code

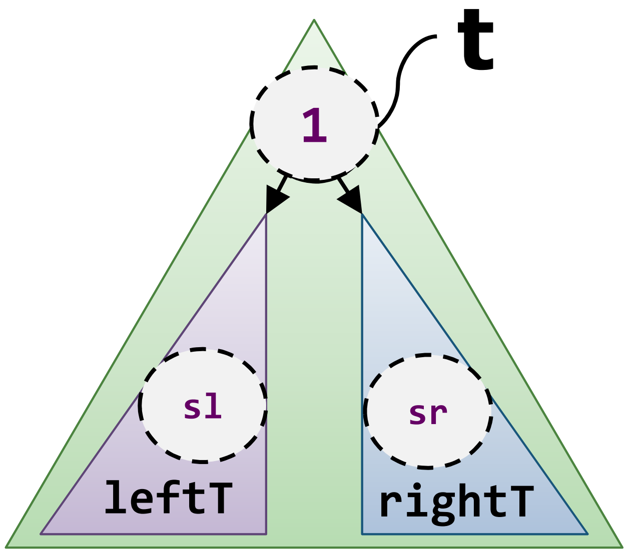
- **Maximum level** of the nodes in the tree




```
def height( T ):
    if T is empty:
        return 0
    else:
        hl = height( T→leftT )
        hr = height( T→rightT )
        return 1 + max(hl, hr)
```

Tree Size: Pseudo-Code

- **Number of nodes in the tree**




```
def height( T ):
    if T is empty:
        return 0
    else:
        sl = size( T➔leftT )
        sr = size( T➔rightT )
        return 1 + sl + sr
```

How to visit every nodes in the binary tree?

BINARY TREE TRAVERSAL

Traversing a Binary Tree

- Purpose of **binary tree traversals**:
  - Visit each node in the tree **exactly once** in a certain order **and**
  - Perform an operation for each node

- There are **four** commonly used traversals:
  a. Post-order traversal
  b. Pre-order traversal
  c. In-order traversal
  d. Level-order Traversal

Pre-, In- and Post- Order: Overview

- Given a non-empty tree **T**, there are **3 parts** that we can process:
  - The **root node**, **left subtree** and **right subtree**
- Since left subtree is prioritized before right subtree (ordered property):
  - There are **three** permutations of the process order:

```
def preOrd( T ):
    operate( T➔Item )
    preOrd( T➔leftT )
    preOrd( T➔rightT )
```

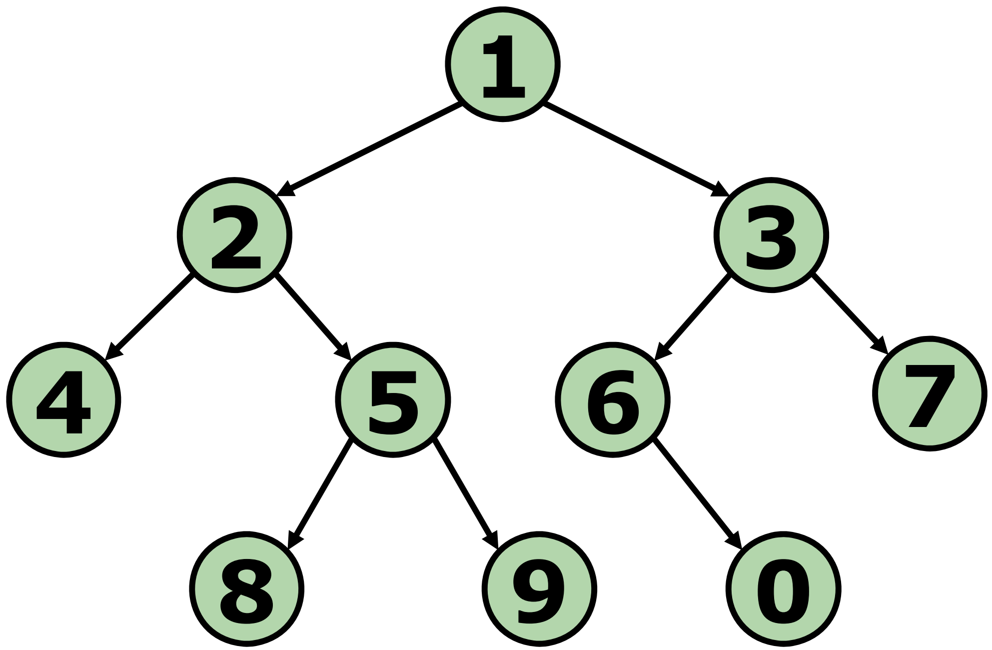
```
def inOrd( T ):
    inOrd( T➔leftT )
    operate( T➔Item )
    inOrd( T➔rightT )
```

```
def postOrd( T ):
    postOrd( T➔leftT )
    postOrd( T➔rightT )
    operate( T➔Item )
```

  - The name comes from ***when*** is the root's item processed:
    - ***pre***: Before any sub-tree, ***post***: After all sub-trees, ***in***: Between of the subtrees

Pre-Order Traversal

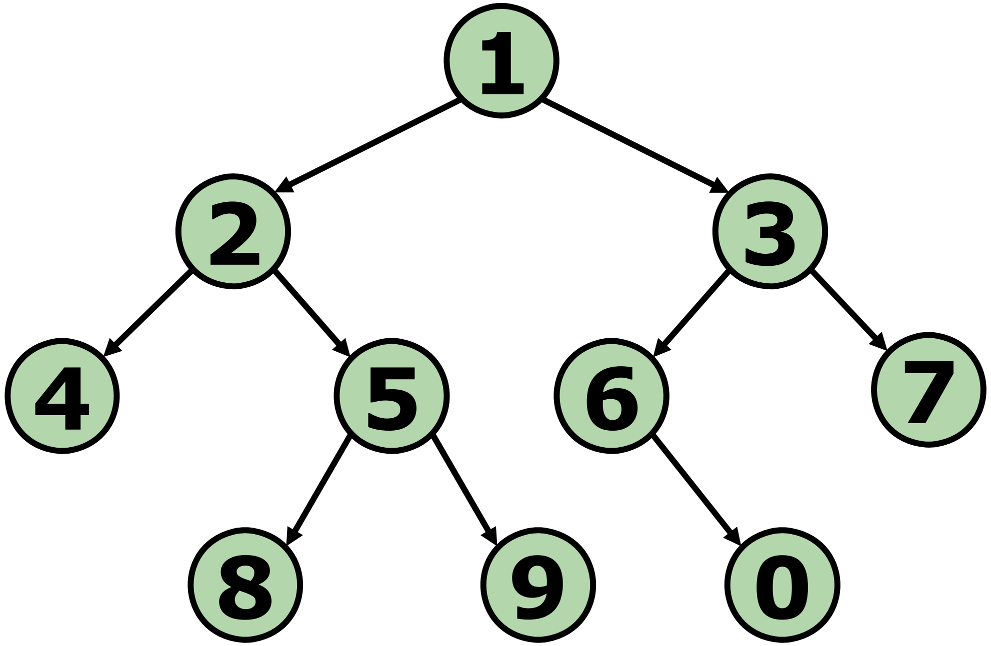
```
def preOrd( T ):
    if T is empty:
        return
    else:
        operate( T➔Item )
        preOrd( T➔leftT )
        preOrd( T➔rightT )
```



Pre-order: 1 2 4 5 8 9 3 6 0 7

In-Order Traversal

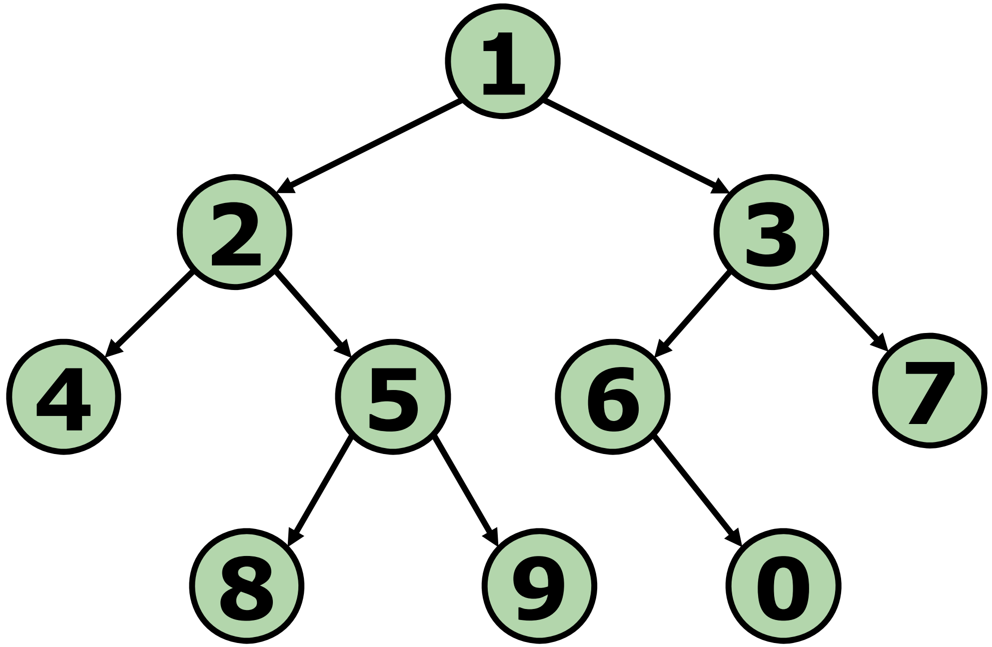
```
def inOrd( T ):
    if T is empty:
        return
    else:
        inOrd( T➔leftT )
        operate( T➔Item )
        inOrd( T➔rightT )
```



In-order: 4 2 8 5 9 1 6 0 3 7

Post-Order Traversal

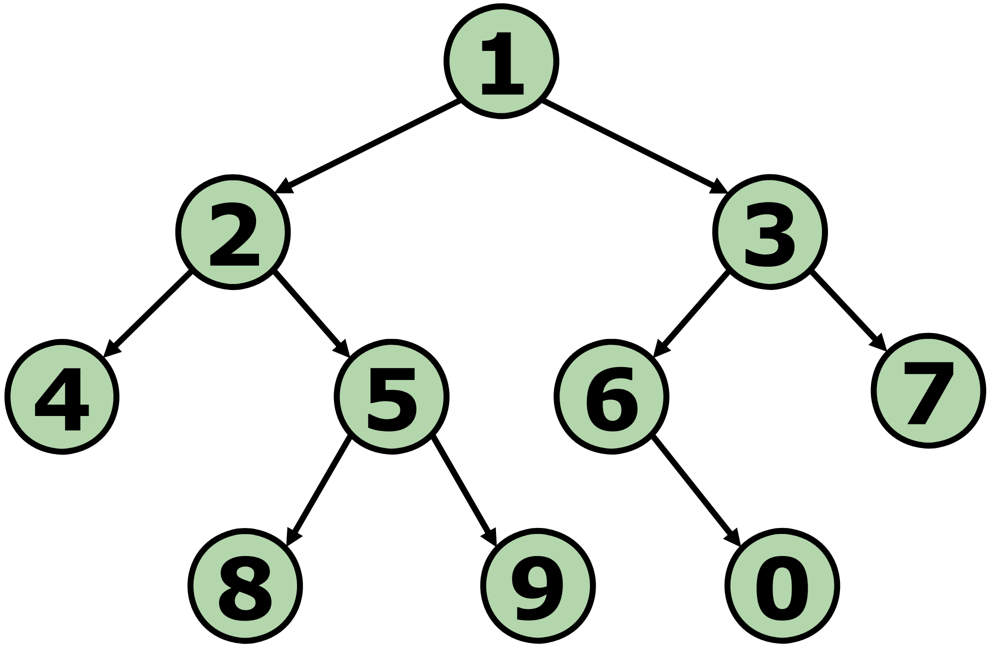
```
def postOrd( T ):
    if T is empty:
        return
    else:
        postOrd( T➔leftT )
        postOrd( T➔rightT )
        operate( T➔Item )
```



Post-order: 4 8 9 5 2 0 6 7 3 1

Level-Order Traversal

- Traverse the tree **level by level** and from **left to right**



Level-order: **1 2 3 4 5 6 7 8 9 0**

Level-Order Traversal: Pseudo code

- Use a **queue** to "remember" the child nodes as we visit a parent node
  - So that the child nodes can be visited later

```
def levelOrd( T ):
    if T is empty:
        return
    nodeQ = Queue()
    nodeQ.enqueue( T )
    while nodeQ not empty:
        cur = nodeQ.front()
        nodeQ.dequeue()

        operate( cur➔Item )

        if cur➔left not empty:
            nodeQ.enqueue( cur➔left )

        if cur➔right not empty:
            nodeQ.enqueue( cur➔right )
```

Use a queue

Non-Empty child node are queued for later processing

Binary Application

ARITHMETIC EXPRESSION TREES

Arithmetic Expression

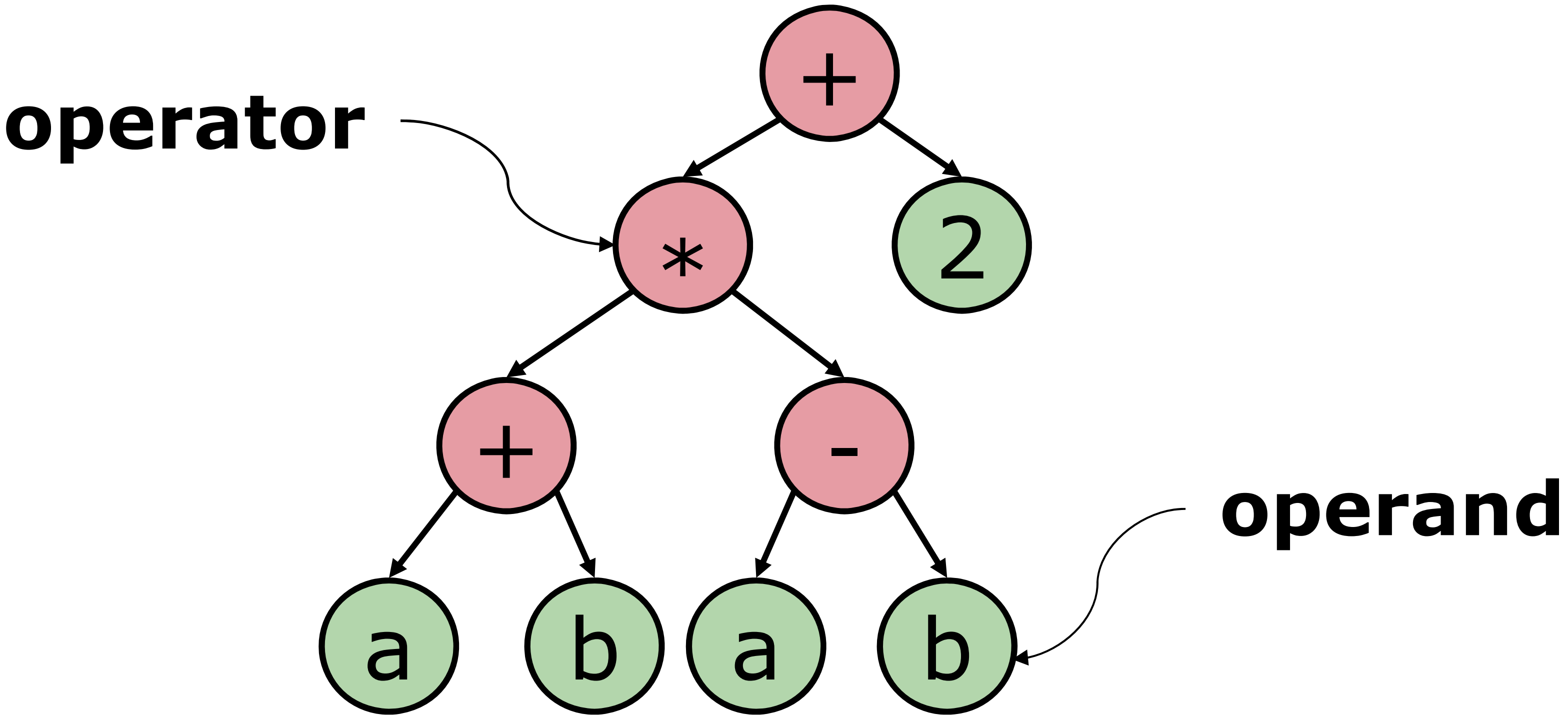
- The commonly used mathematical expression is known as **infix expression**
  - Operators like "+", "-", etc appear **in-between** of its two operands

- Infix expression is inherently **ambiguous**
  - Need precedent and brackets to show the order of operation
  - E.g. a – b – c vs a – (b – c)

- **Question:**
  - Can you guess what are **prefix** and **postfix** expressions?

Arithmetic Expressions **as Binary Tree**

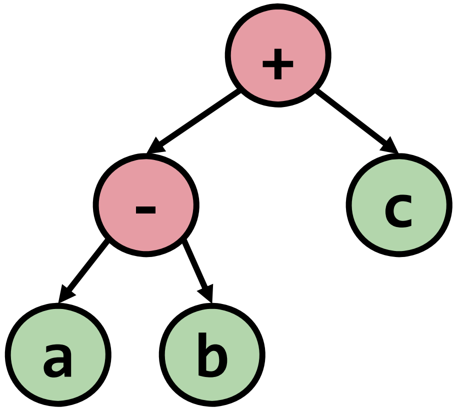
- Binary Tree provides an unambiguous representation of an expression



Leaf nodes (or **leaves**) store operands
Internal nodes and **root** store operators

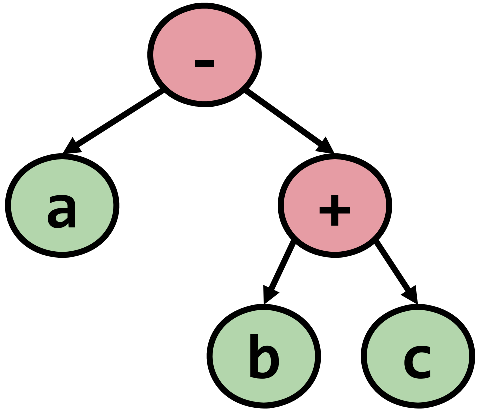
Expressions Tree: Order of evaluation

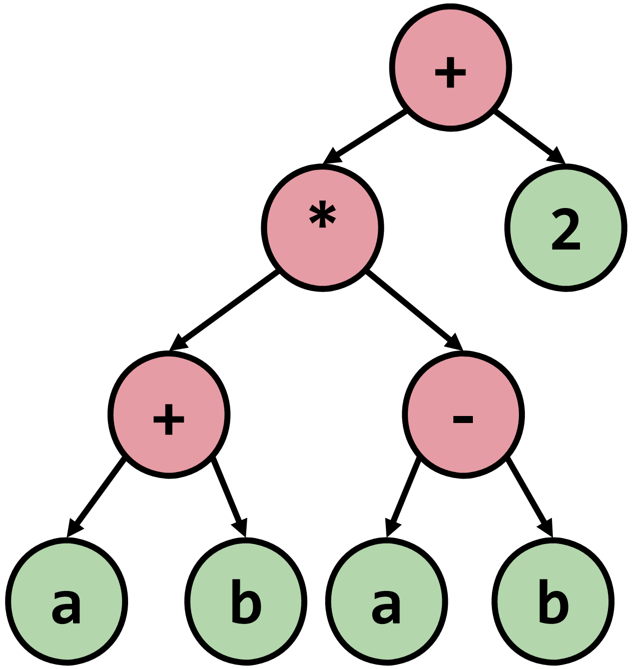
- An operator can be evaluated only if both of operands **ready**



Corresponds to
`(a – b) + c`

Corresponds to
`a – (b + c)`

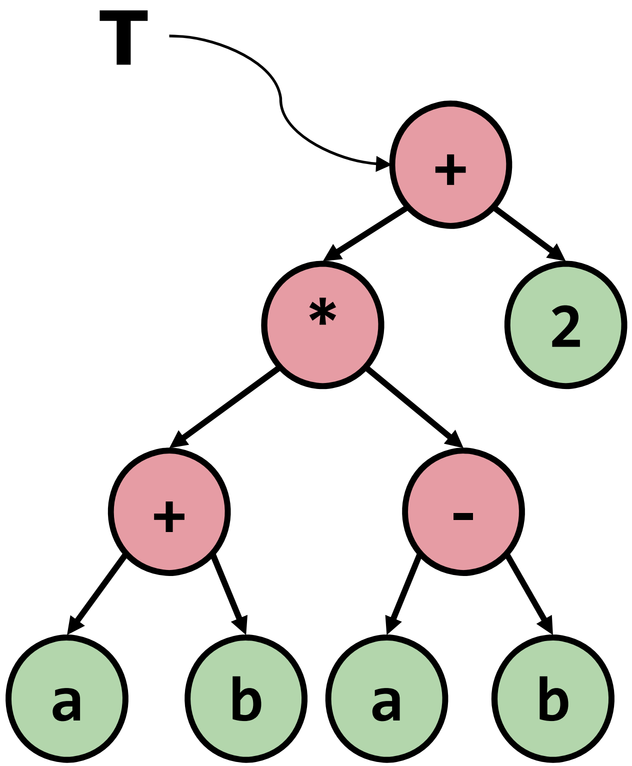




Corresponds to
???

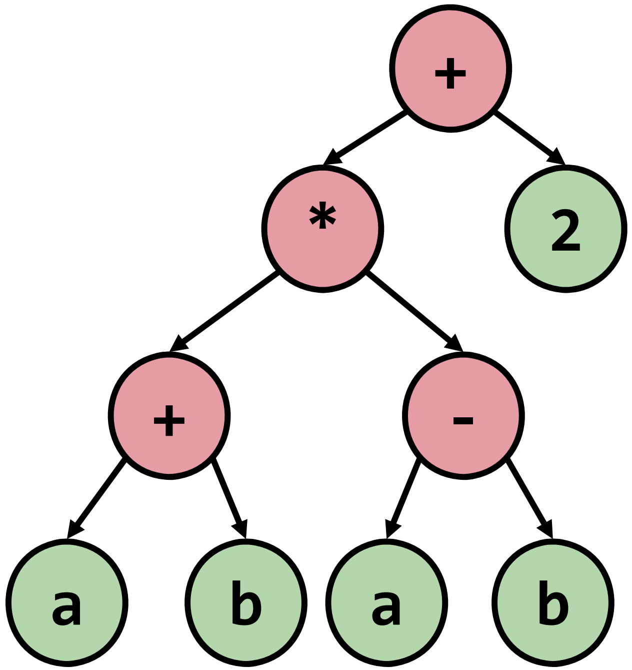
Expressions Tree: Evaluation

```
def eval( T ):
    if T is empty:
        return 0

    if T is a leaf node:
        return T➔item

    if T is a "+":
        return eval( T➔LT ) \
               + eval ( T➔RT )

    if T is a "*":
        return eval( T➔LT ) \
               * eval ( T➔RT )

    #Other operators not shown
```



Q: Do you need to consider the **priorities** of the operators?

Expression Tree: Traversal



Pre-order:
`(+ (* (+ a b) (- a b)) 2)`

In-order:
`(((a + b) * (a – b)) + 2)`

Post-order:
`(((a b +) (a b -) *) 2 +)`

- Pre-order gives **prefix expression**
- Post-order gives **postfix expression**
- Fun fact:
  - Prefix and postfix expression **do not need brackets** to eliminate ambiguity! Try it!

END