Algorithms and Data Structures for Data Science
Recursion

CS 277
Brad Solomon

February 21, 2024

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Exam 1 next week

Multiple Choice / Fill in the blank exam

Covers content through Monday February 19th

See website for details

Learning Objectives

Introduce recursion in the context of trees

Explore recursion in the context of loops

Practice recursion in the context of lists

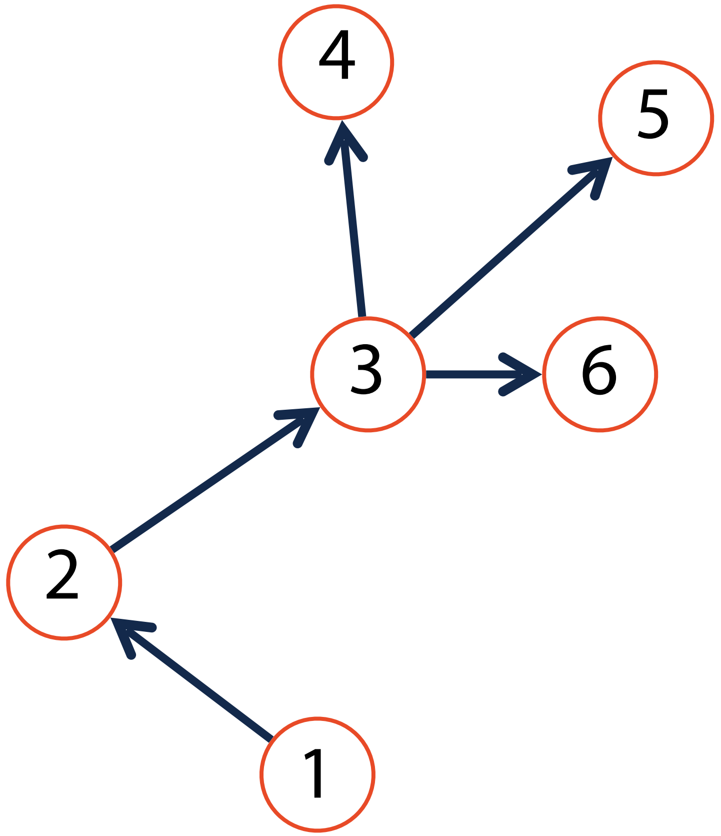
Trees

A non-linear data structure defined recursively as a collection of nodes where each node contains a value and zero or more connected nodes.

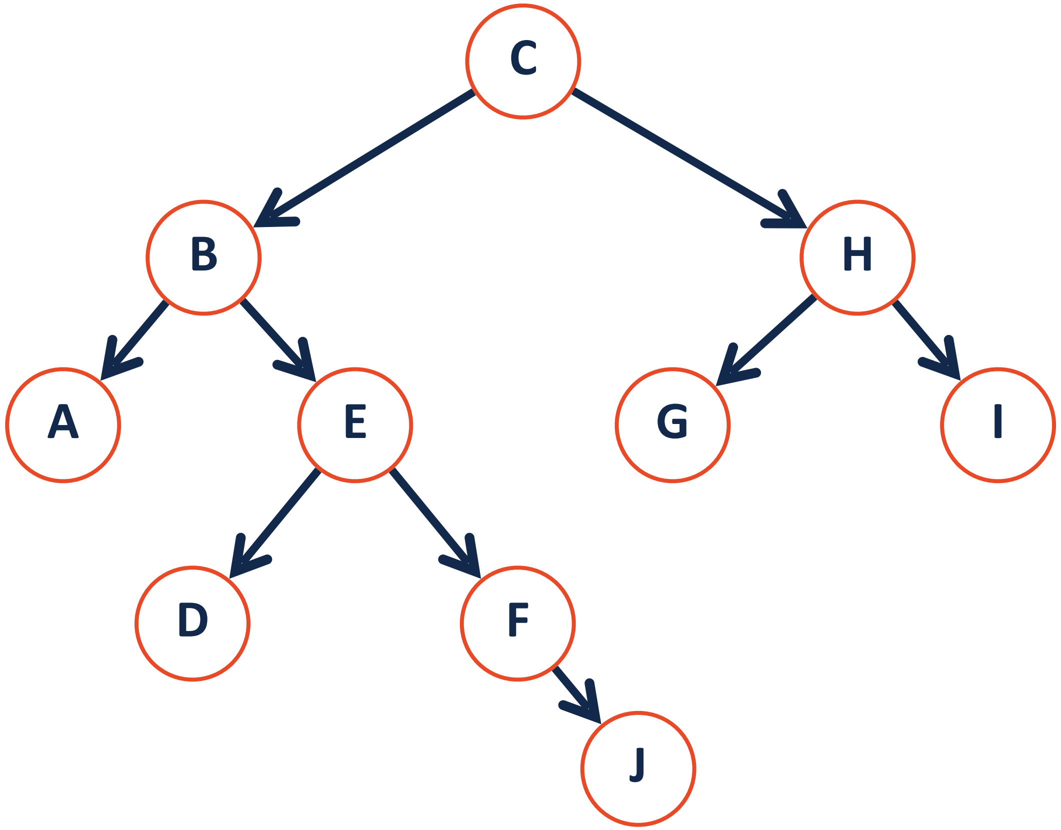
(In CS 277) a tree is also:

1) Acyclic

2) Rooted



Tree Terminology

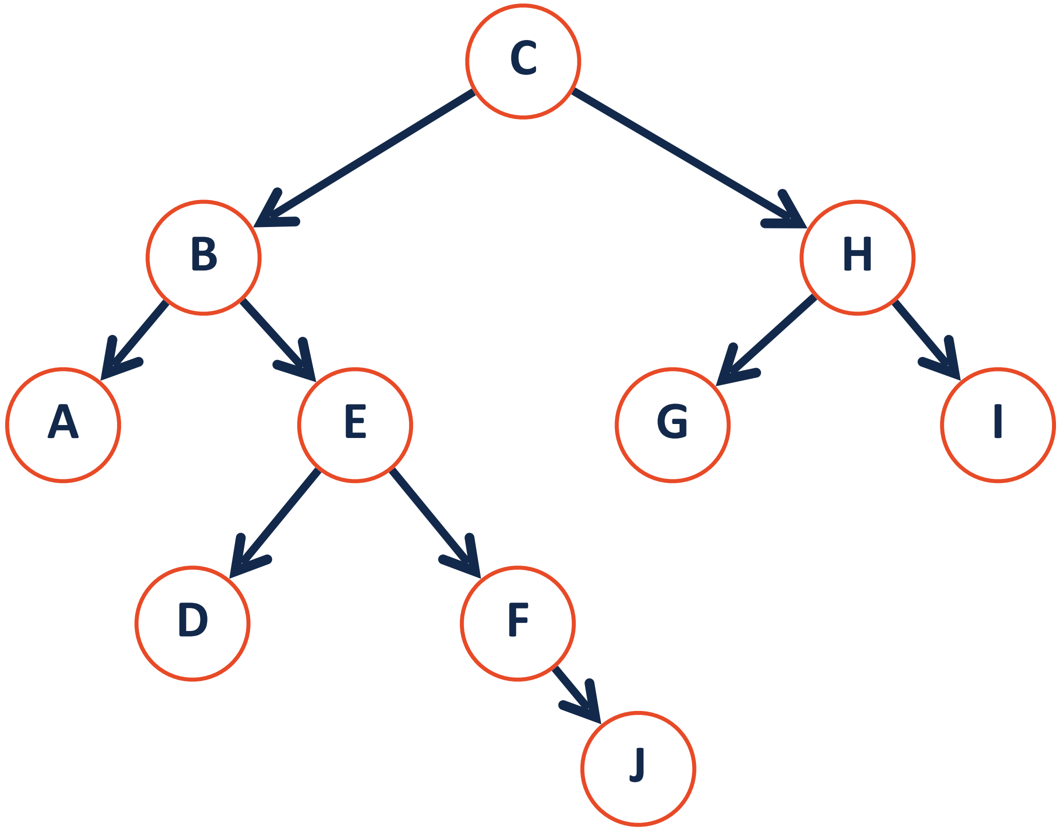


Node: The vertex of a tree

Edge: The [theoretical] connecting path between nodes

Path: A list of the edges (or nodes) traversed to go from node `start` to node `end`

Tree Terminology



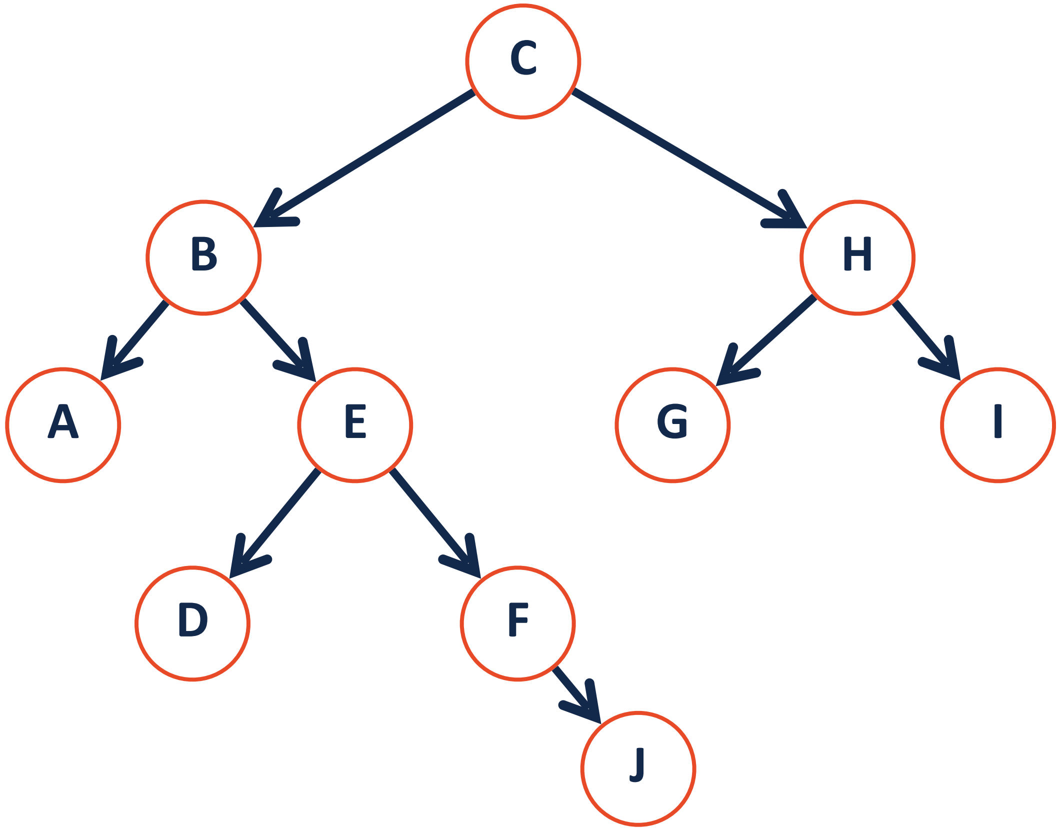
Parent: The precursor node to the current node is the 'parent'

Child: The nodes linked by the current node are it's 'children'

Neighbor: Parent or child

Degree: The number of children for a given node

Tree Terminology

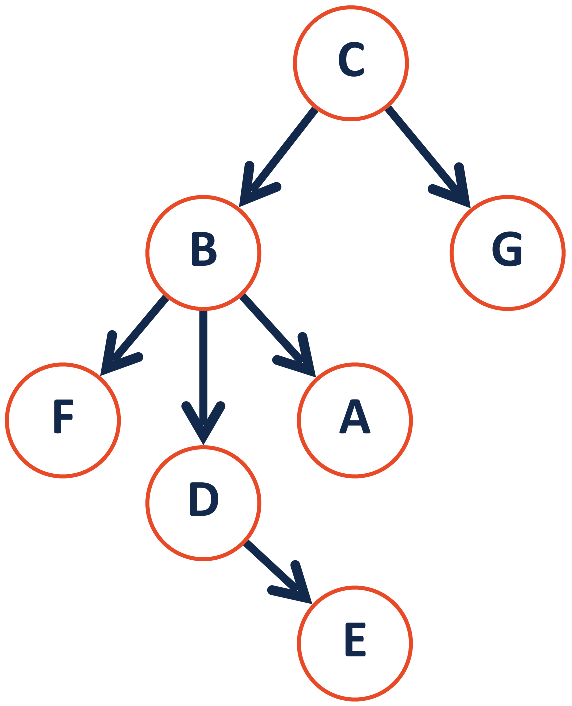


Root: The start of a tree (the only node with no parent).

Leaf: The terminating nodes of a tree (have no children)

Internal: A node with at least one child

Tree Terminology Practice



What is the longest path in the tree?

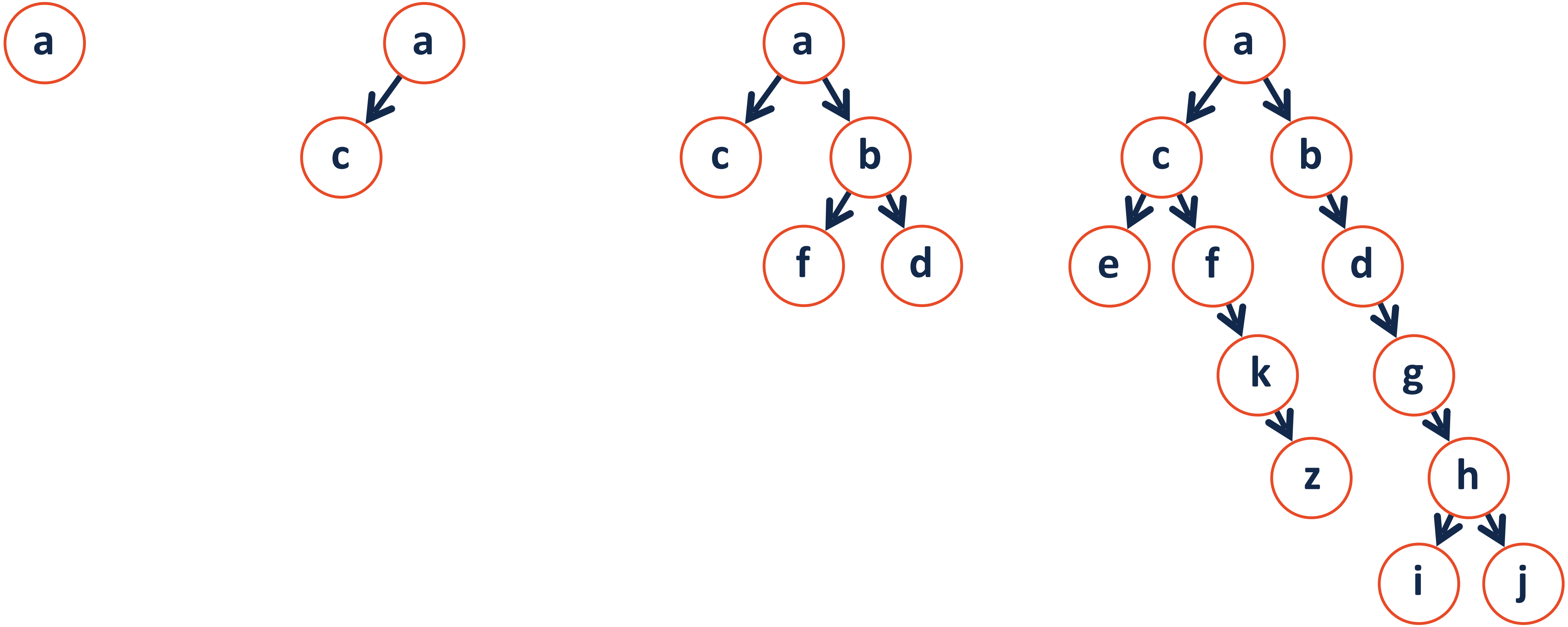
What is the neighbors of node B?

How many leaves does this tree have?

What is the largest degree in the tree?

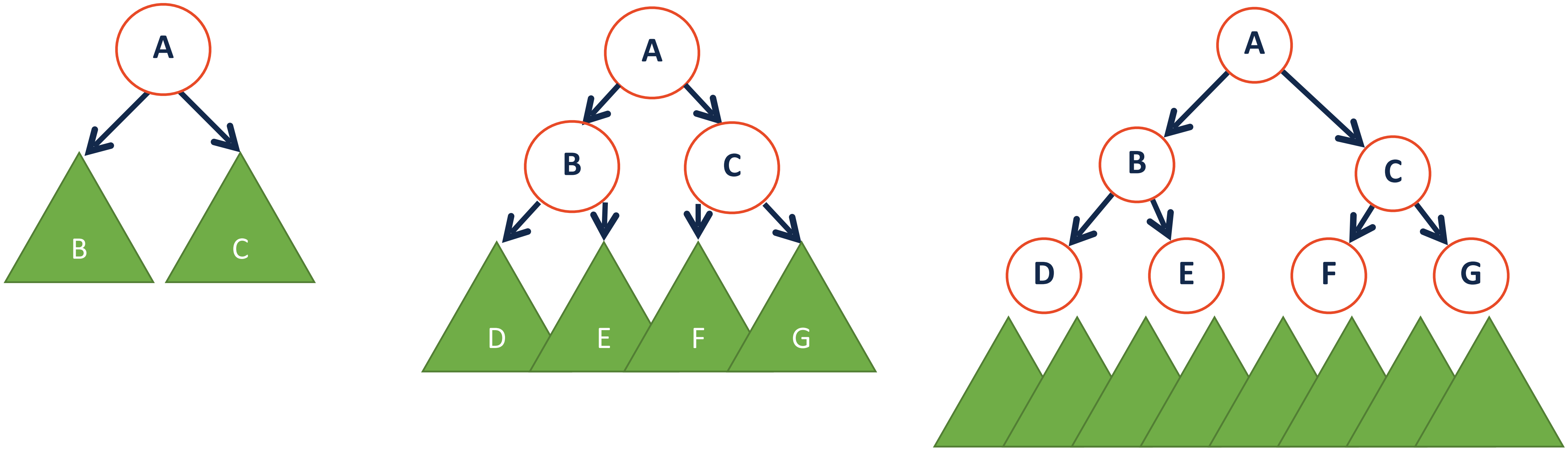
Tree Terminology

Height: the length of the longest path from the root to a leaf



Tree Height Calculation Breakdown

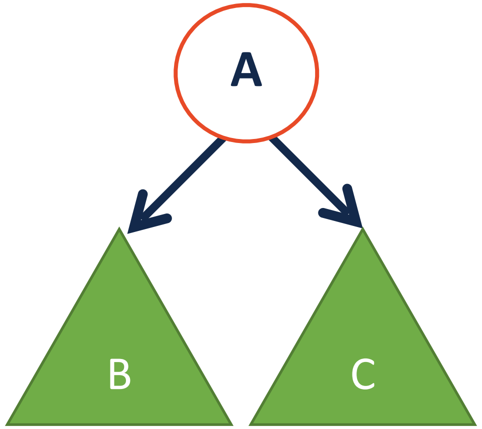
How does a *program* identify the height of a tree?



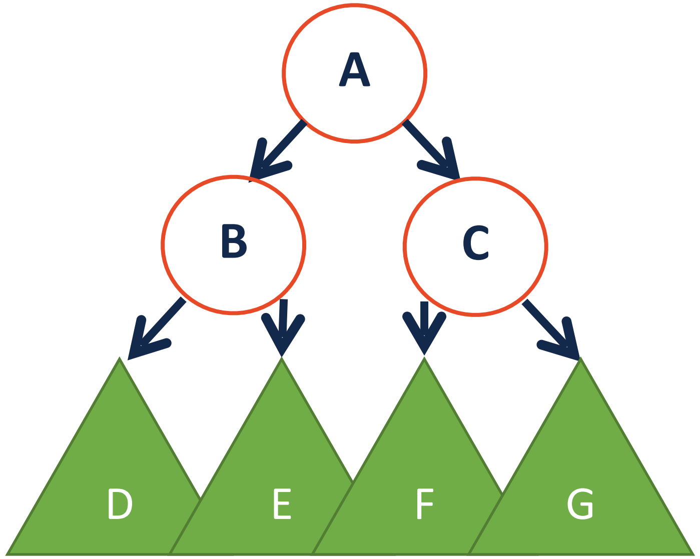
Tree Height Calculation Breakdown

How does a *program* identify the height of a tree?

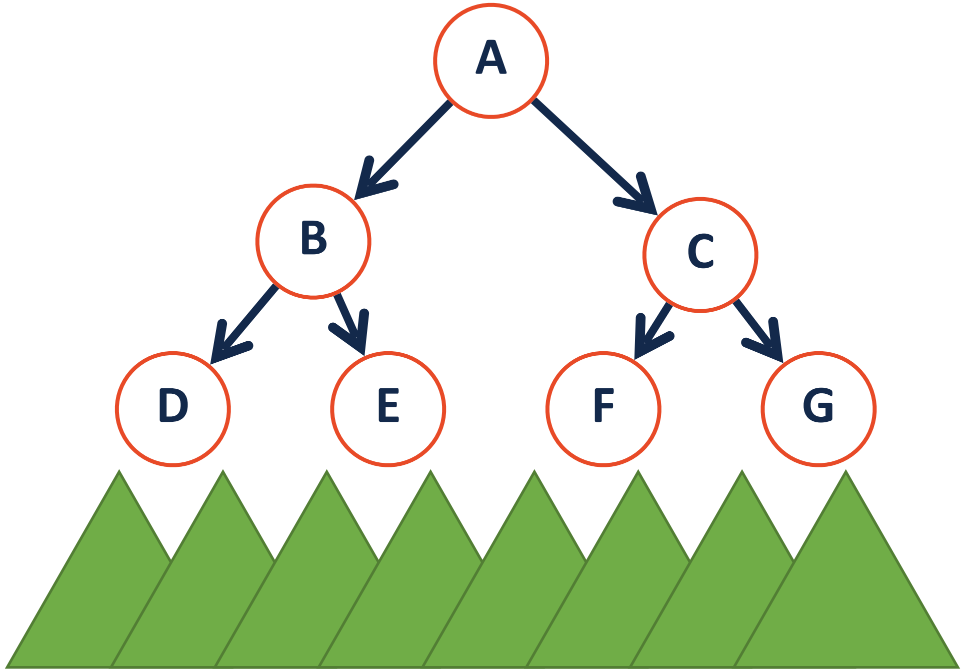
The height of my tree is 1 plus the height of my children!



To get H(A)



I need H(B) and H(C)

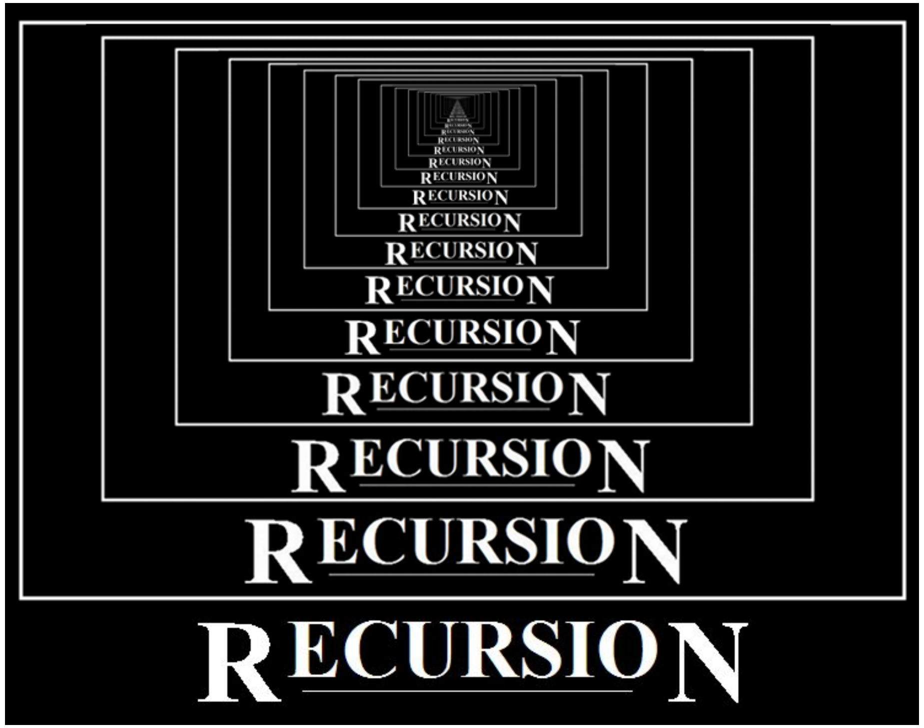


I need H(D) and H(E) and…

I need H(F) and H(G) and…

Programming Toolbox: Recursion

The process by which a function calls itself directly or indirectly is called **recursion**.



Don’t panic — we’ve already used it before!

Linked List Recursion

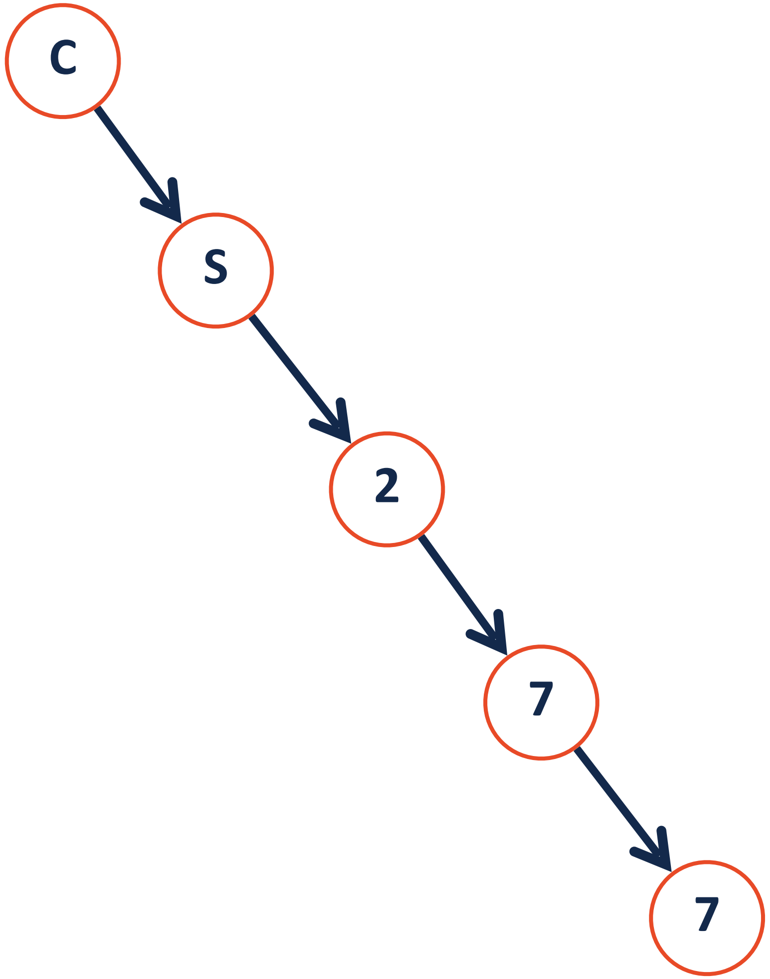
A **linked list** is a list L such that:

$$L = None$$

or

$$L = listNode(val, L_{next})$$

```
class listNode:
    def __init__(self, val, next=None):
        self.val = val
        self.next = next

```



(Binary) Tree Recursion

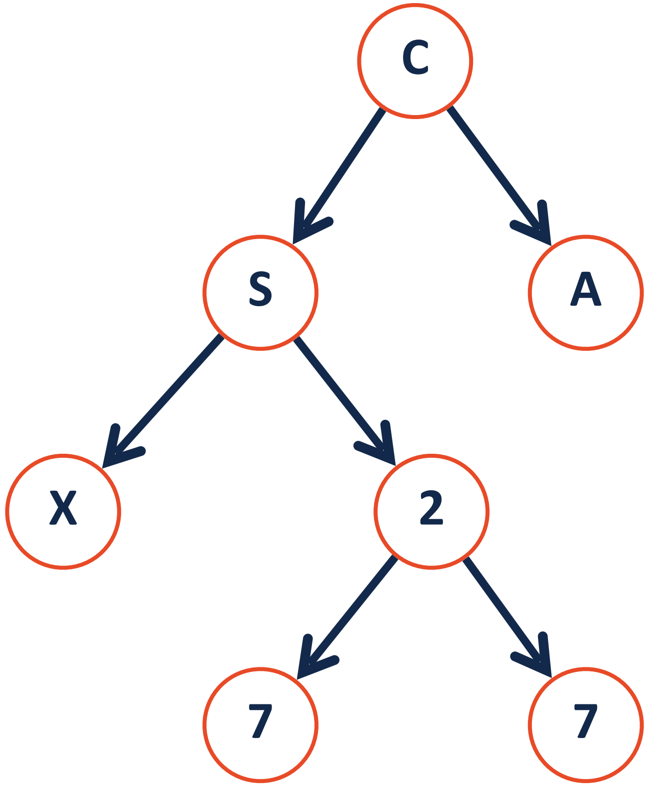
A **binary tree** is a tree T such that:

$$T = None$$

or

$$T = treeNode(val, T_L, T_R)$$

```
class treeNode:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```



Visualizing a binary tree

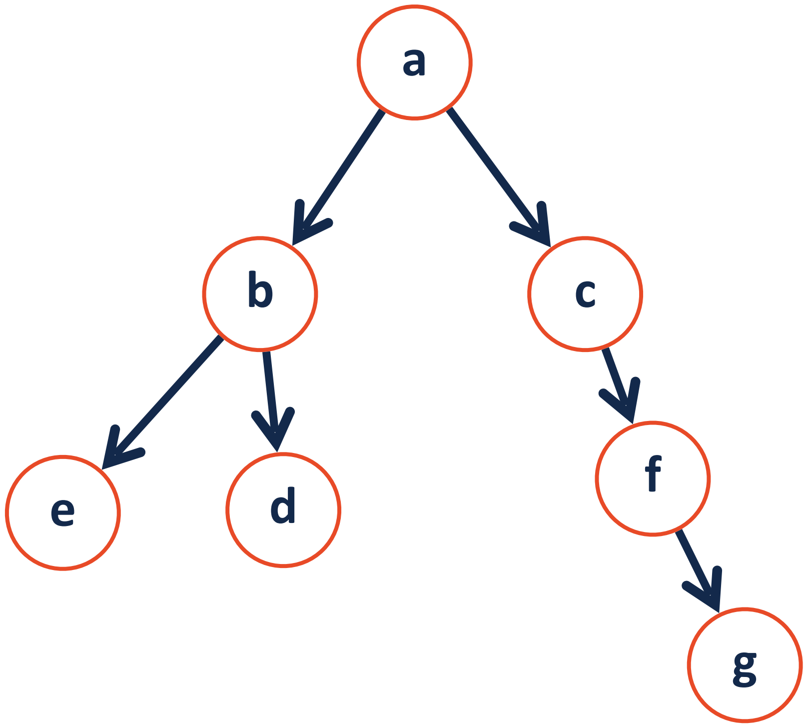
```
class treeNode:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

```
a = treeNode('a')
b = treeNode('b')
c = treeNode('c')
d = treeNode('d')
e = treeNode('e')
f = treeNode('f')
g = treeNode('g')

a.left = b
a.right= c
b.right = d
b.left = e
c.right = f
f.right = g
```

Visualizing a binary tree… recursively

```
class treeNode:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

```
a = treeNode('a')
b = treeNode('b')
c = treeNode('c')
d = treeNode('d')
e = treeNode('e')
f = treeNode('f')
g = treeNode('g')

a.left = b
a.right= c
b.right = d
b.left = e
c.right = f
f.right = g
```




```
a = treeNode('a', treeNode(‘b’,treeNode(‘e'),treeNode('d')),
treeNode(‘c', None, treeNode(‘f’, None, treeNode(‘g'))))
```

Programming Toolbox: Recursion

At its core, recursion is nothing more than another way of writing loops:

```
for i in range(n+1):
    print(i)
```

```
def recursiveFor(n):
    if n == 0:
        print(n)
        return

    recursiveFor(n-1)

    print(n)
```

Programming Toolbox: Recursion

Lets deep dive into whats actually happening here:

```
def recursiveFor(n):
    if n == 0:
        print(n)
        return

    recursiveFor(n-1)

    print(n)
```

```
def recursiveFor(n):
    if n == 0:
        print(0)
        return

    print(n)

    recursiveFor(n-1)
```

recursiveFor(2)

recursiveFor(1)

recursiveFor(0)

Programming Practice: Recursive Code

What is the following code doing?

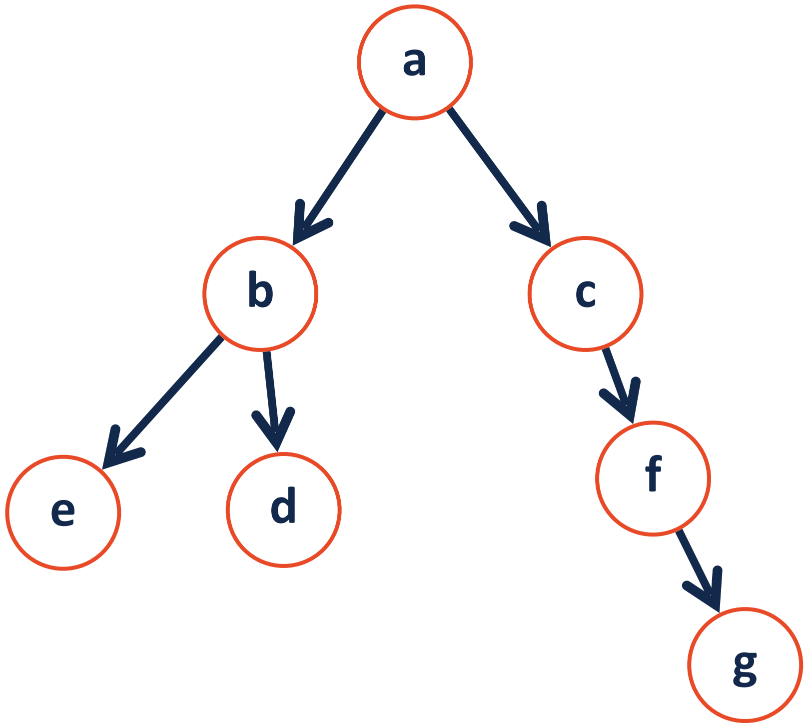
```
def recurse(i):
    if i == 0:
        return i
    return recurse(i-1)+i

```

```
def recurse(inList):

    if len(inList)==0:
        return 0

    inList.pop()

    return recurse(inList)+1

```

Programming Toolbox: Recursion

Anything that can be solved with a loop can be solved with recursion

But sometimes its easier to code up a solution recursively



I can't loop through a tree with **for** or **while**…

But I can loop through the tree using recursion!

Programming Toolbox: Recursion

When thinking recursively, break the problem into parts:

Base Case: What is the smallest sub-problem? What is the trivial solution?

Recursive Step: How can I reduce my problem to an easier one?

Combining: How can I build my solution from recursive pieces?

Recursive Tree Height

What is the height of my tree T?

Base Case: What is the smallest sub-problem? What is the trivial solution?

Recursive Step: How can I reduce my problem to an easier one?

Combining: How can I build my solution from recursive pieces?

Recursive Sum

Given a list, sum all the items in the list ***using recursion***

Base Case: What is the smallest sub-problem? What is the trivial solution?

Recursive Step: How can I reduce my problem to an easier one?

Combining: How can I build my solution from recursive pieces?

Recursive Sum

Given a list, sum all the items in the list ***using recursion***

8	4	2	6	5

Recursive findMax

Given a list, find the max item in the list ***using recursion***

Base Case:

Recursive Step:

Combining:

Recursive findMax

Given a list, find the max item in the list ***using recursion***

8	4	2	6	5

Recursive Fibonacci

Given a number *n*, return the *nth* Fibonacci number:

$$Fib(n) = Fib(n-1) + Fib(n-2)\,, \quad n > 1$$

Base Case:

Recursive Step:

Combining:

Recursive List Partitioning

Using all elements in a list, can we make two lists which have equal sums?

6	5	4	2	7

1	1	1	1	1

2	3	3	3	1

Recursive List Partitioning

How would a computer solve this problem?

6	5	4	2

Recursive List Partitioning

How would a computer solve this problem? **Compute every permutation!**

6	5	4	2

6			

5	4	2	

6	5		

4	2		

6	5	4	

2			

6	2		

5	4		

...

Recursive List Partitioning

Writing code to attempt every possible permutation is tricky with loops.

But its a great example of recursion in action!

Recursive Step: Given list L, pop() L[0] to left ***and*** right and recurse on both

Recursive List Partitioning

Recursive Step: Given list L, pop() L[0] to left ***and*** right and recurse on both

Input:

6	5	4	2

Left

Right

Recursive Calls:

5	4	2

Left

6			

Right

5	4	2

Left

Right

6			

Recursive List Partitioning

Recursive Step: Given list L, pop() L[0] to left ***and*** right and recurse on both

Base Case:

Base Case: When my input list is empty, I have tried every permutation

Recursive Step: Given list L, pop() L[0] to left ***and*** right and recurse on both

```
[4, 3, 1]                    ([], [])

[3, 1]        ([4], [])                          ([], [4])

[1]  ([3, 4], [])  ([4], [3])     ([3], [4])   ([], [3, 4])

[]

([1, 3, 4], [])  ([1, 4], [3])   ([1, 3], [4])  ([1], [3, 4])

([3, 4], [1])    ([4], [1, 3])   ([3], [1, 4])  ([], [1, 3, 4])
```

Recursive List Partitioning

Base Case: When my input list is empty, I have tried every permutation

Recursive Step: Given list L, pop() L[0] to left ***and*** right and recurse on both

Combination Step:

Lab Recursion

Recursive List Partitioning is one of the questions on Fridays lab!

In preparation for Friday, consider how you might use recursion to solve:

- Computing the factorial of a number
- Counting the sum of all digits in a number
- Checking if a string is a palindrome