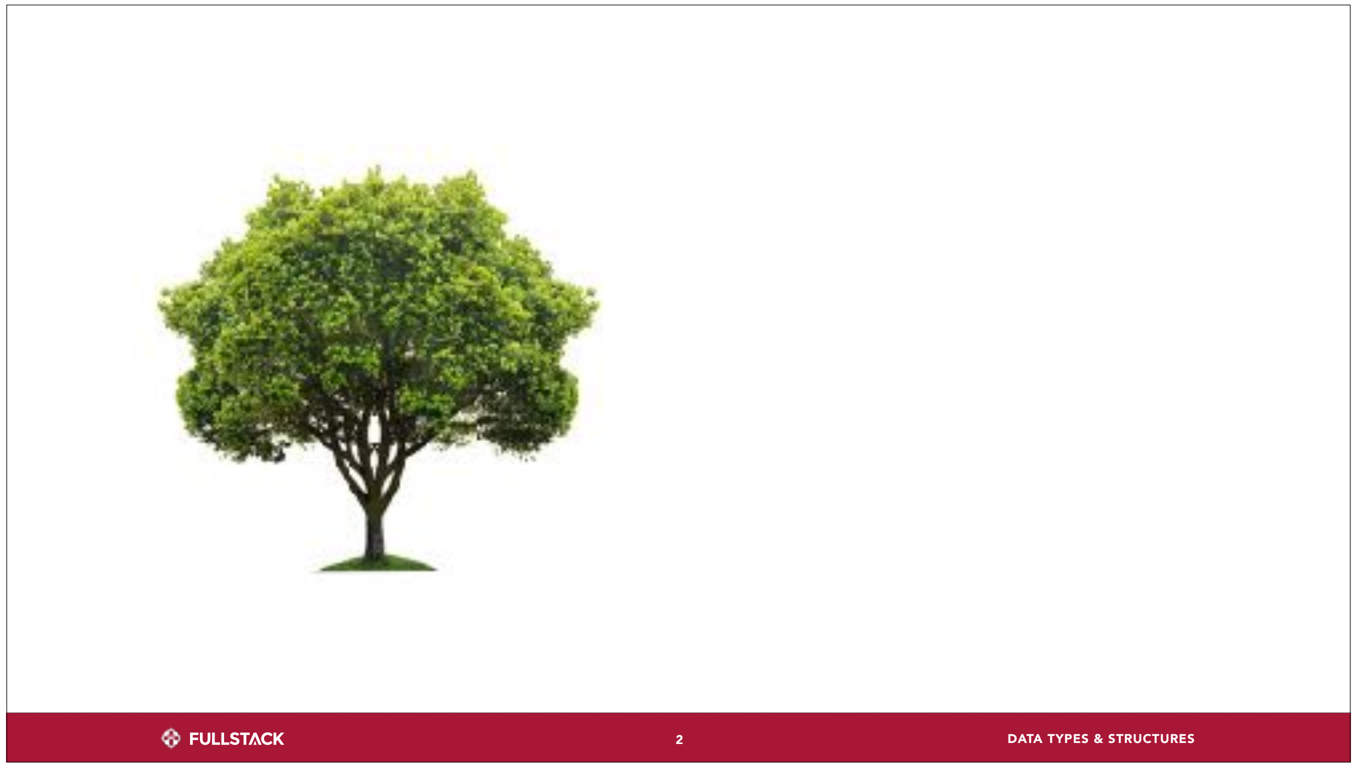
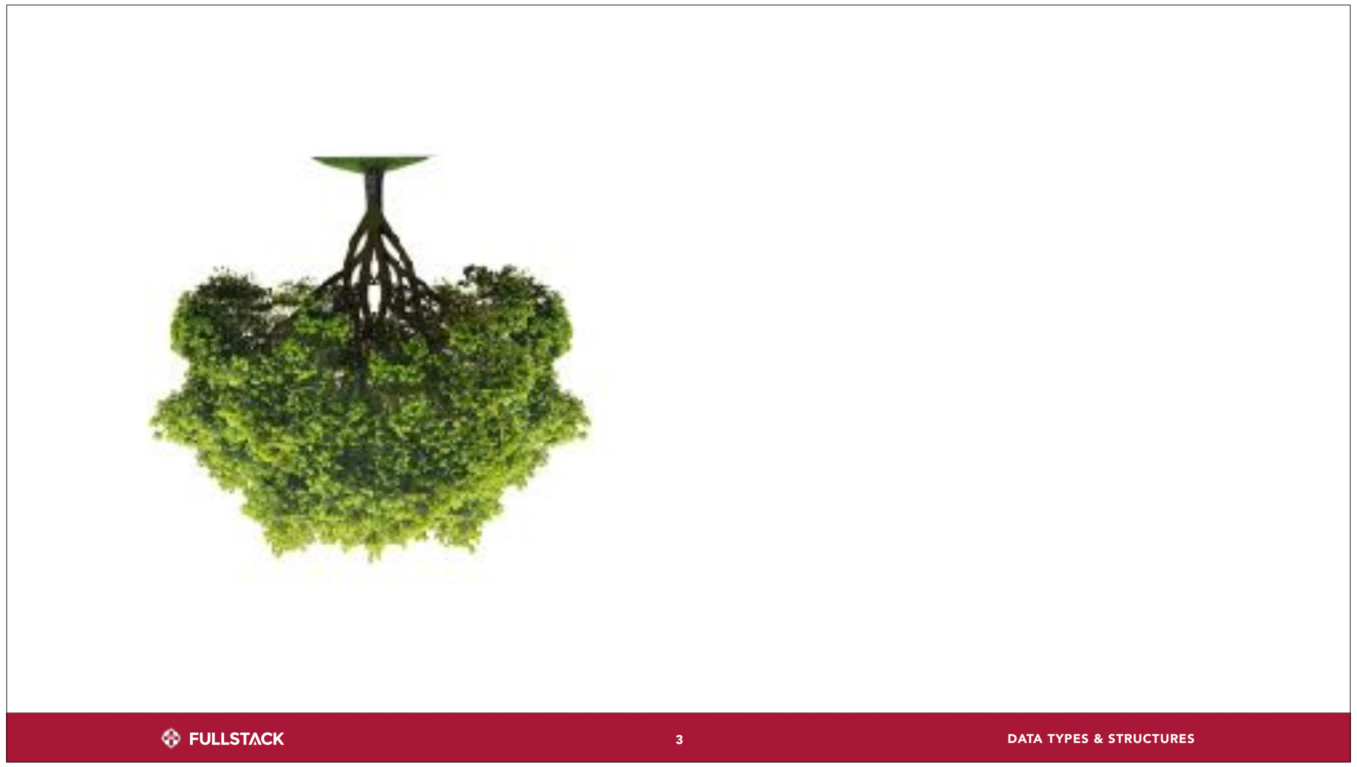
Trees

Trees are an incredibly important data structure in computer science. A vast number of useful data structures — binary search trees, red-black trees, B-trees, AVL trees, tries, heaps, and others — are variations on the concept of a tree.



Computer scientists tend to look at things from novel perspectives. For example, most computer scientists…



…think of trees with the root at the top and the leaves at the bottom.

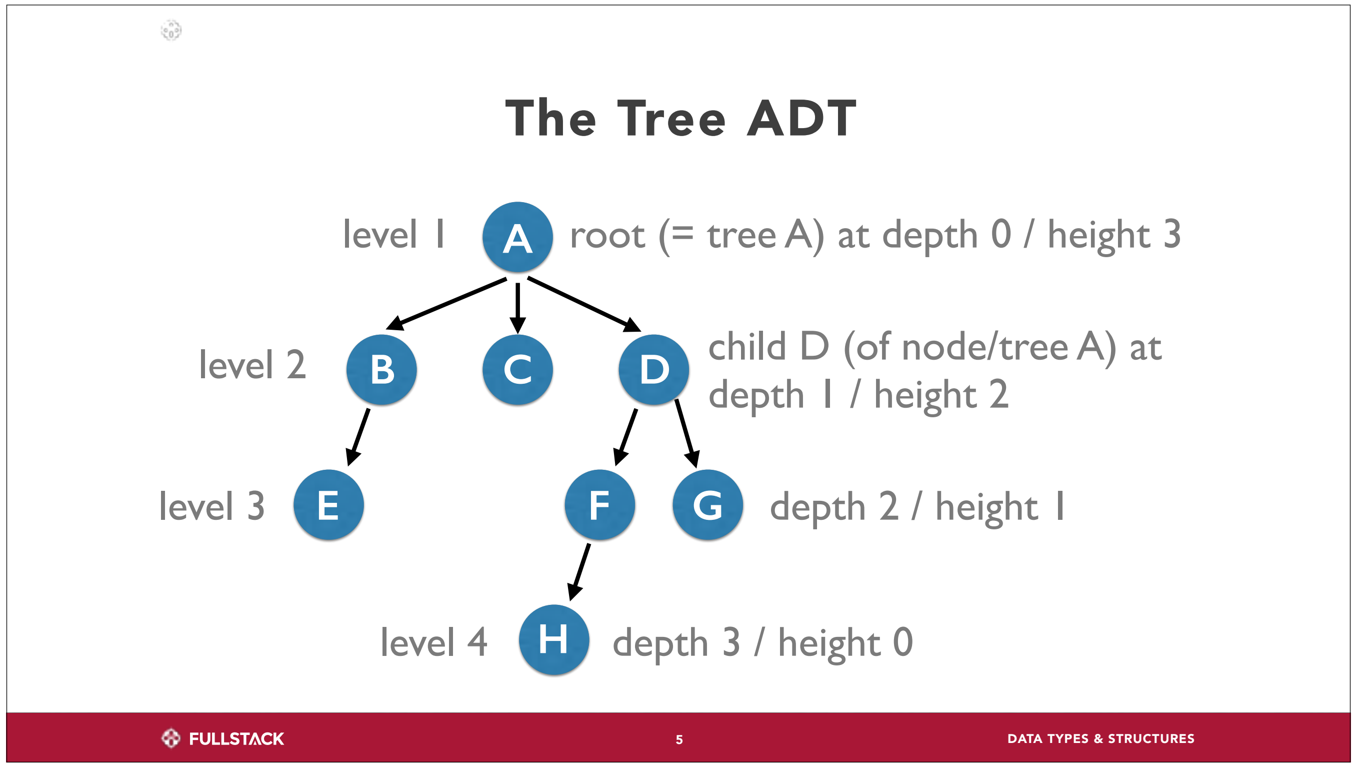
The Tree ADT

$$\begin{tikzcd}
 & & A \arrow[dl] \arrow[d] \arrow[dr] & & \\
 & B \arrow[dl] & C & D \arrow[d] \arrow[dr] & \\
E & & & F \arrow[d] & G \\
 & & & H &
\end{tikzcd}$$

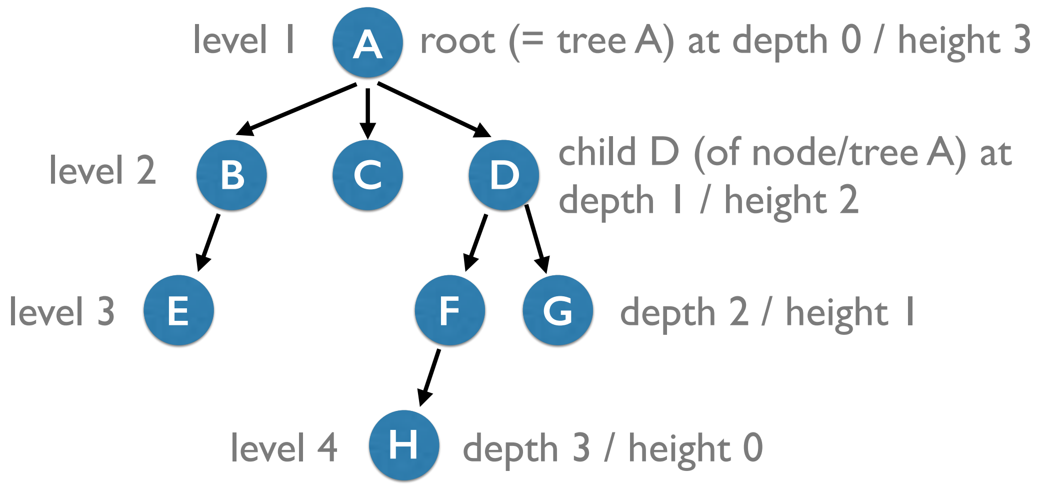
- Nodes contain value(s)
- A primary "root" node
- Children are subtrees (recursive!)
- No duplicated children (cycles); trees can *branch* but never *converge*
- Final nodes called "leaves"
- Height of tree = longest path to leaf
- Level = 1 + number of jumps to root
- Depth = inverse of height

A tree ADT consists of Nodes, which contain values (very similar to a linked list). We have a primary root node at the very top, and the child nodes of that root node are called “branches”. The final node on any branch is, of course, called a leaf. One of the things that distinguishes a tree from more generalized data structures called graphs is that there no cycles - that is, our branches never converge (in this case, C would never point to F or anything like that).

Trees are also very flexible in the operations that can be performed on them - you can add children to the tree, look for elements, remove elements, etc. Usually, each parent has references to children, but it is rare that children have a reference to their parent.

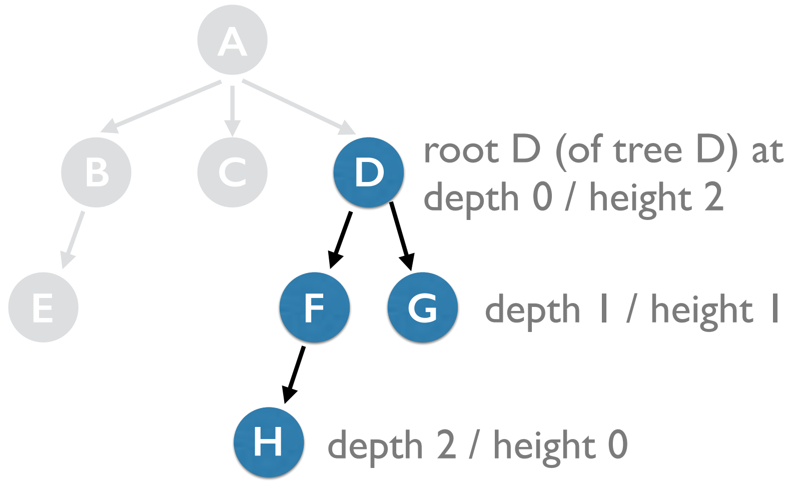


The Tree ADT

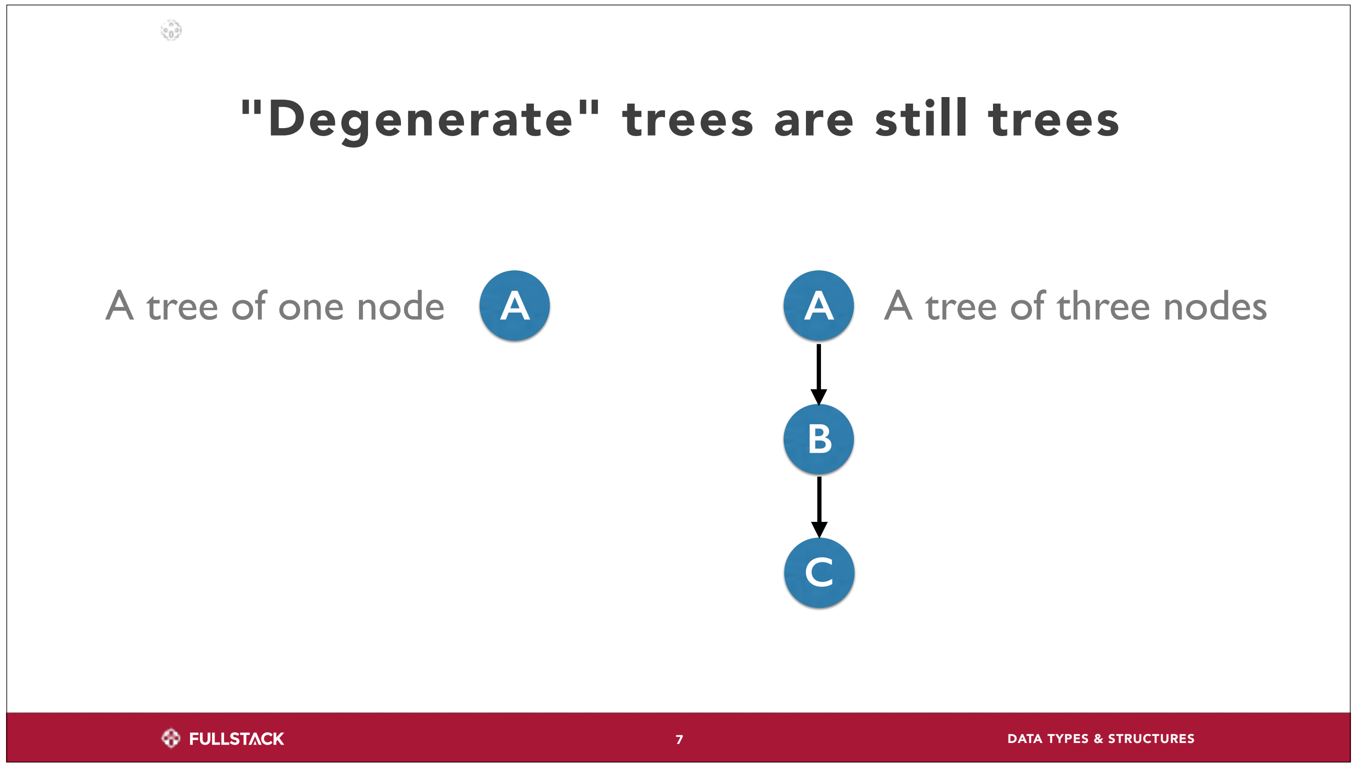

level 1
A
root (= tree A) at depth 0 / height 3
level 2
B
C
D
child D (of node/tree A) at depth 1 / height 2
level 3
E
F
G
depth 2 / height 1
level 4
H
depth 3 / height 0

Trees have a lot of confusing terminology related to level / height / depth, but it's not really important. The crucial stuff is understanding that every tree is defined by its root node, and all other nodes are children or further descendants.

Every node is the root of a tree.
You might even say a node *represents* a tree.



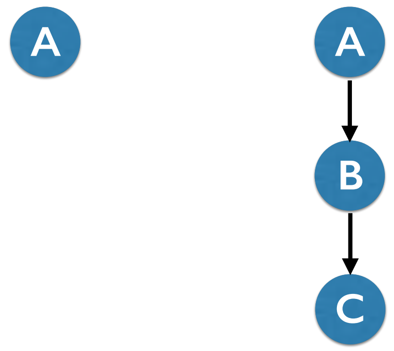
Trees are highly recursive structures! In fact one often considers "tree" and "node" as synonymous because every node is the root of a subtree. Side note: the height of a node (distance to farthest leaf) is invariant, but the depth is relative to which node you select as the root.



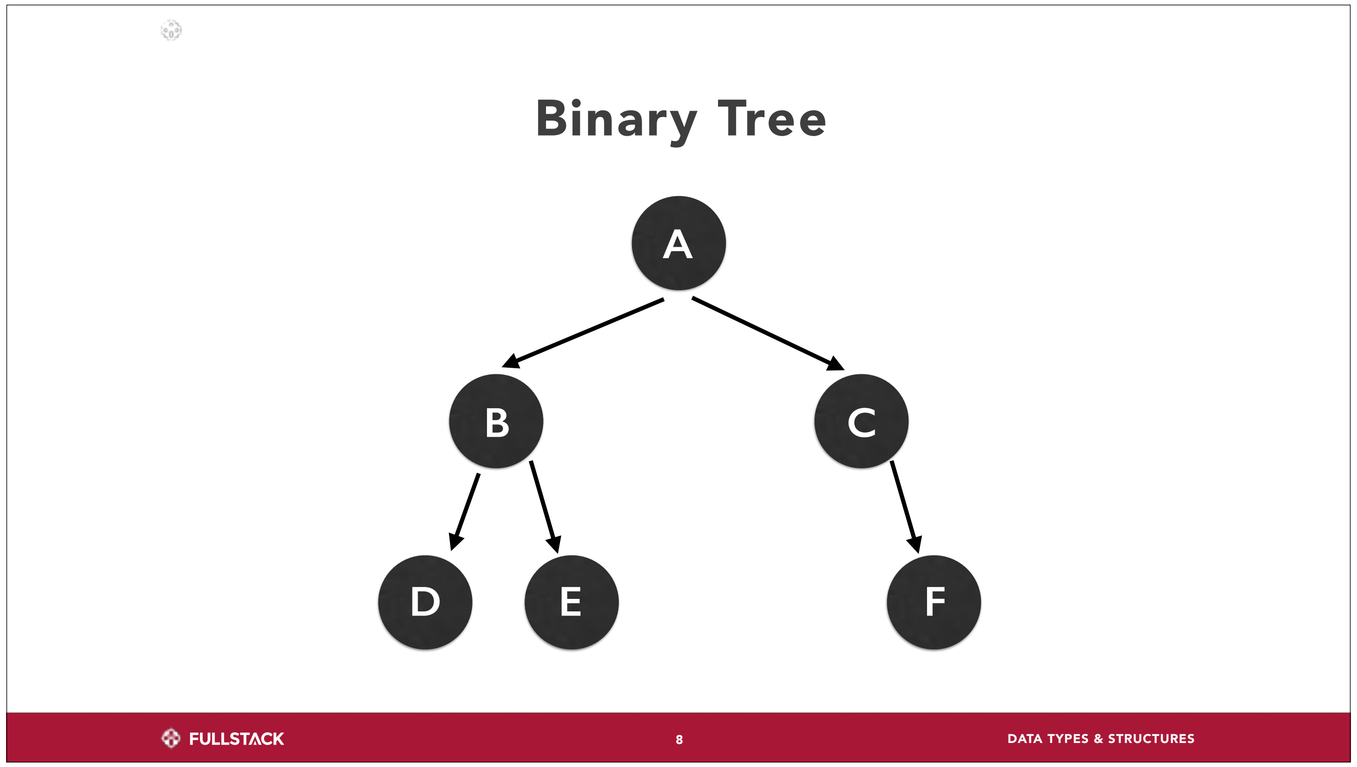
"Degenerate" trees are still trees

A tree of one node

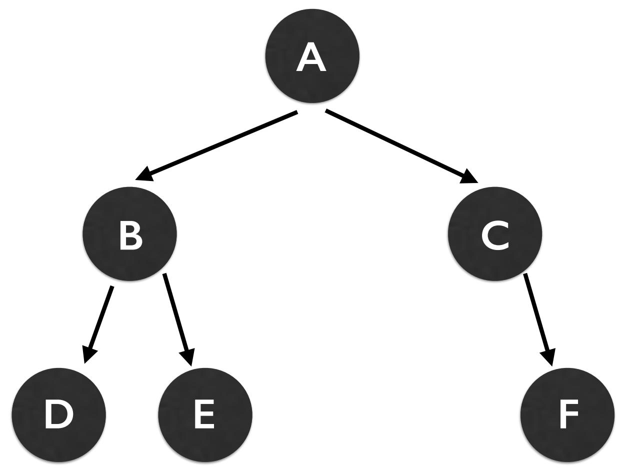
A tree of three nodes



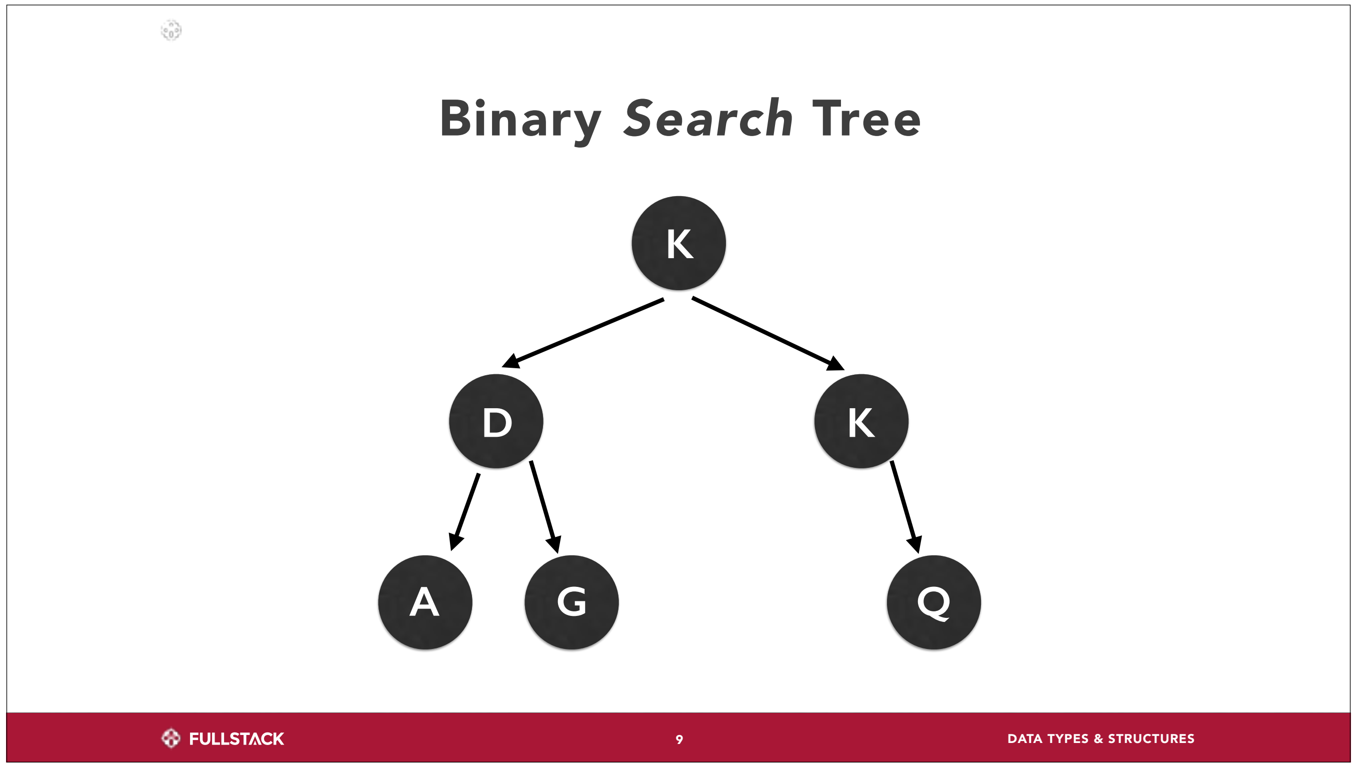
Note that the definition of a tree doesn't say that nodes MUST have more than one child — or even any children at all. A node all by its lonesome is technically still a tree.



Binary Tree



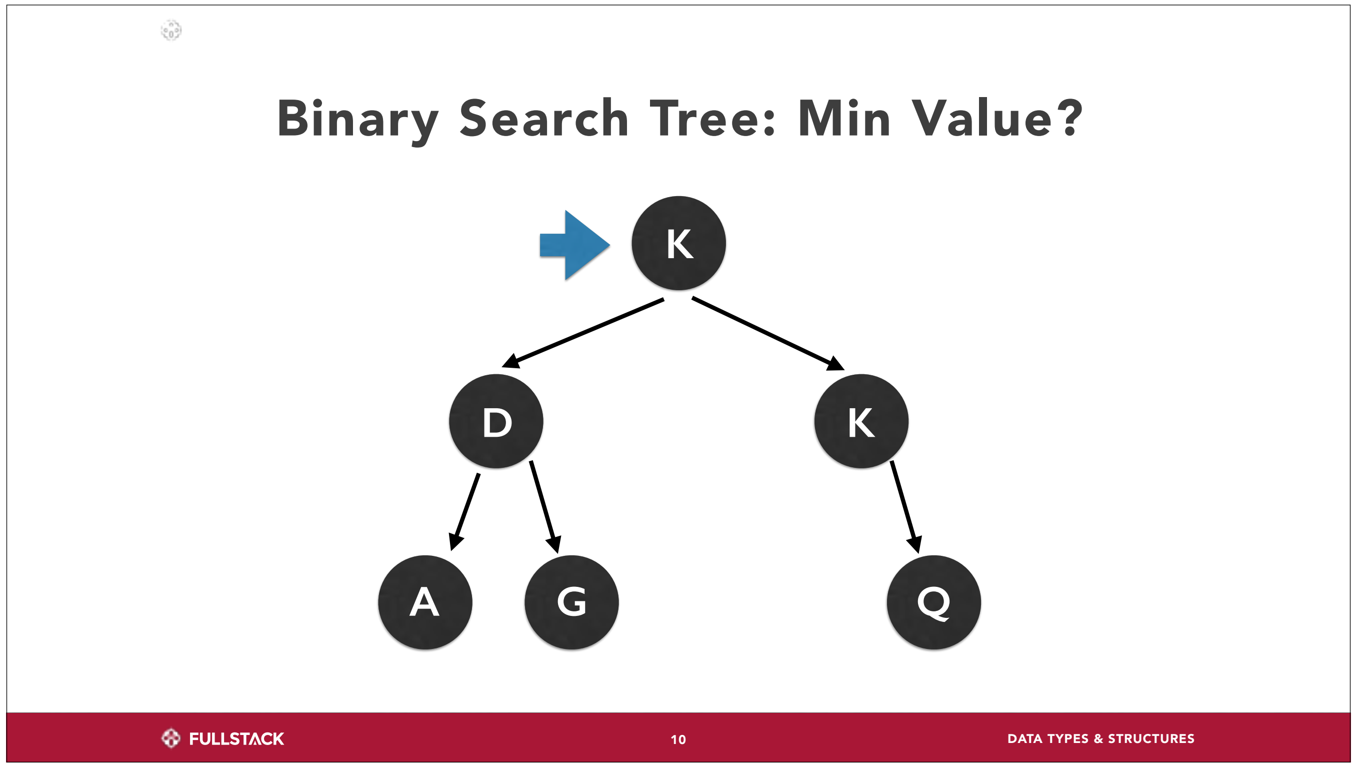
Binary trees are those for which every node has at most two children, typically called the *left* and *right* child with respect to its parent.



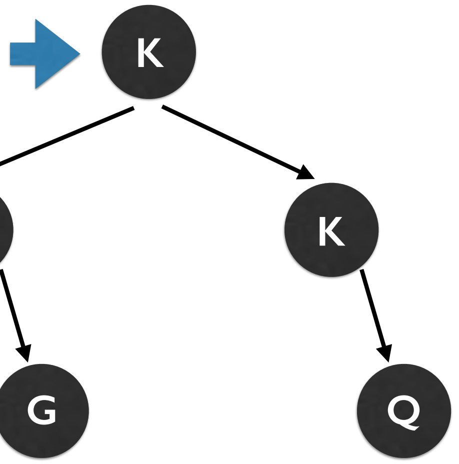
Binary *Search* Tree

$$\begin{tikzcd}
 & K \arrow[dl] \arrow[dr] & \\
D \arrow[d] \arrow[d, shift left] & & K \arrow[d] \\
A \quad G & & Q
\end{tikzcd}$$

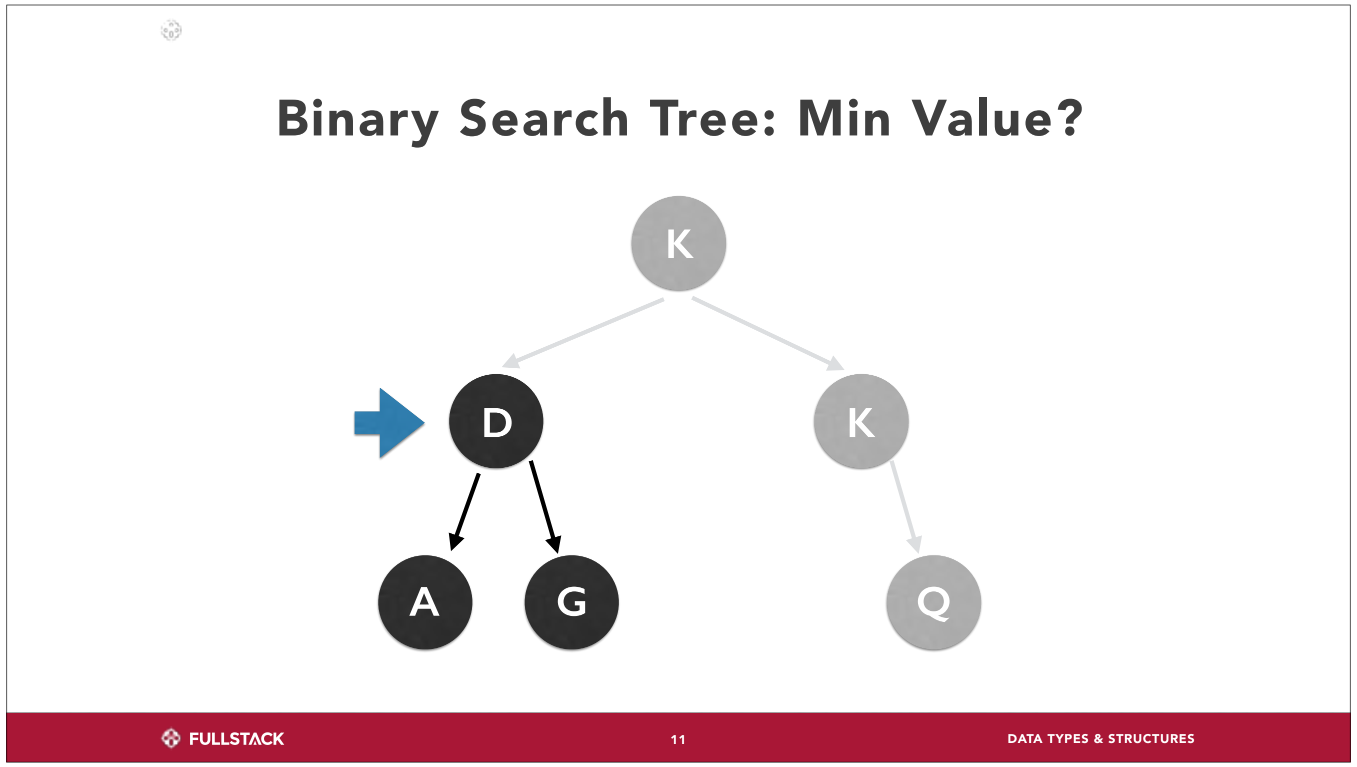
A binary SEARCH tree (BST) is one for which every node satisfies an *ordering*, i.e. all left descendants < node < all right descendants. Ties (equal values) can either go left or right so long as one consistent rule is chosen for the whole tree. Why is ordering useful?



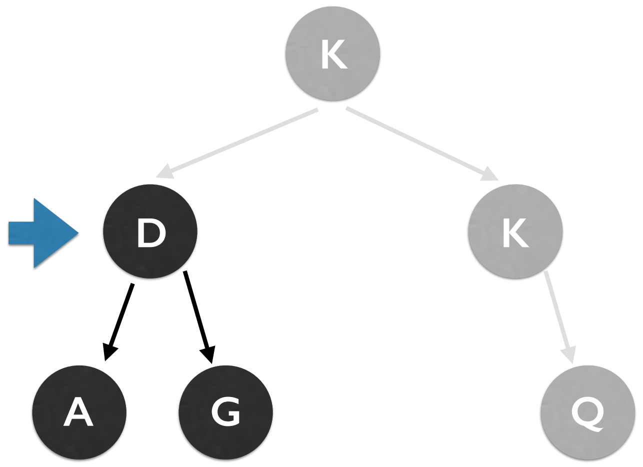
Binary Search Tree: Min Value?



Let's see a BST in action, by SEARCHing for the minimum (leftmost) value. With a tree, we always start at the root (usually because that's the node we have a reference to).

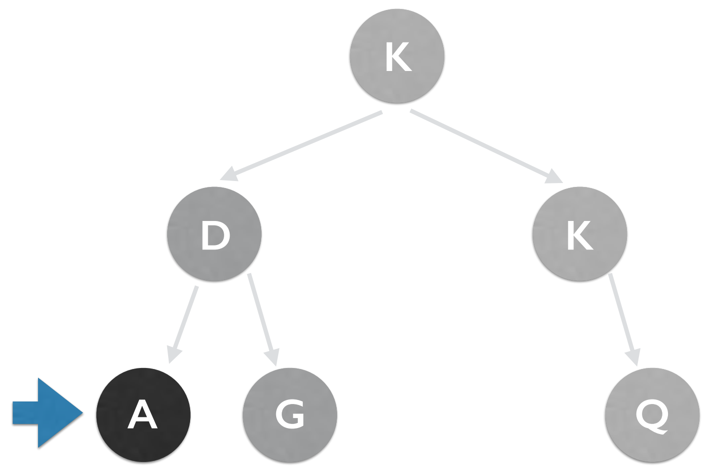


Binary Search Tree: Min Value?

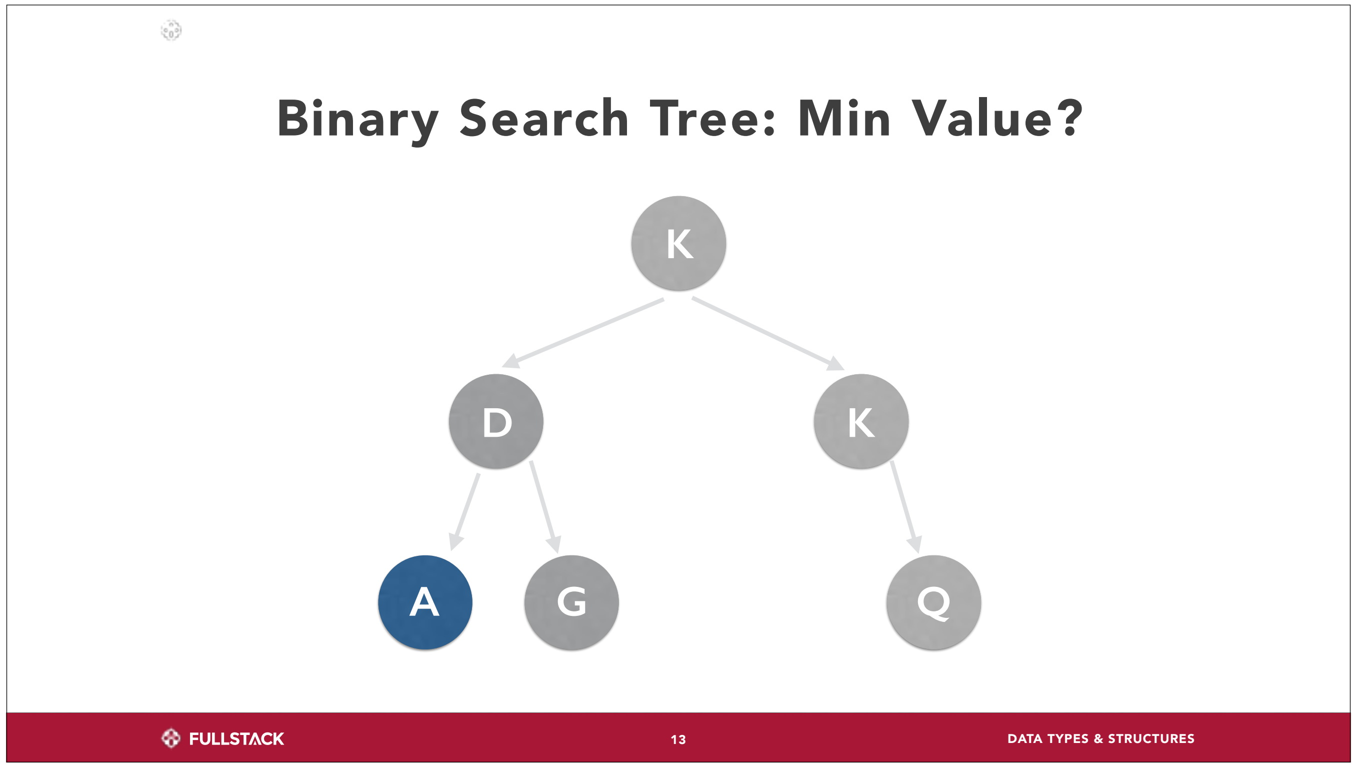


Since we want the minimum (leftmost) value, we go down to the left child. We just threw away (on average) half of the remaining tree; if our tree had 100 nodes, we just eliminated an average of 50 nodes, without even having to look for them!

Binary Search Tree: Min Value?



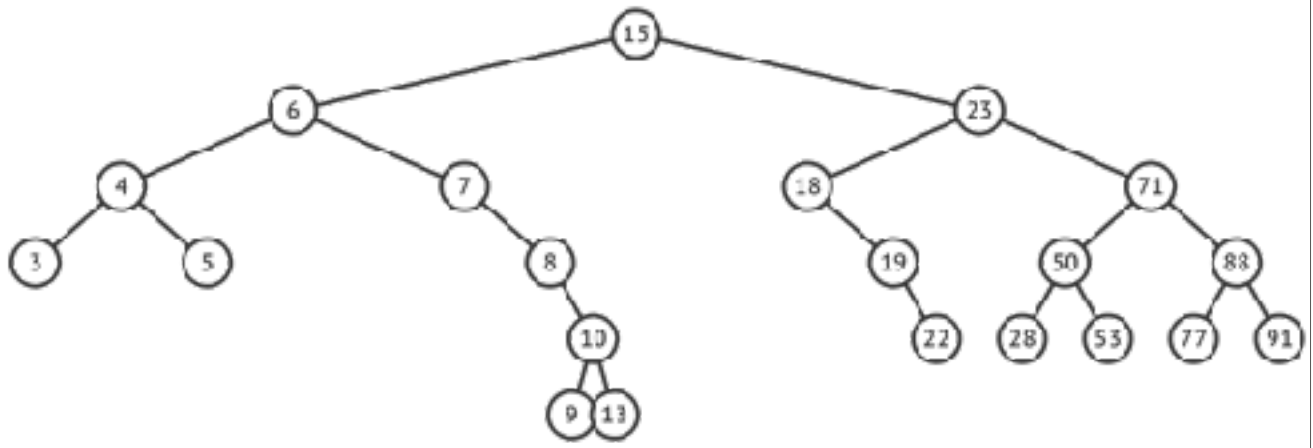
Again we move to the left child, again eliminating half of the remaining tree. We are eliminating nodes pretty quickly!



Binary Search Tree: Min Value?

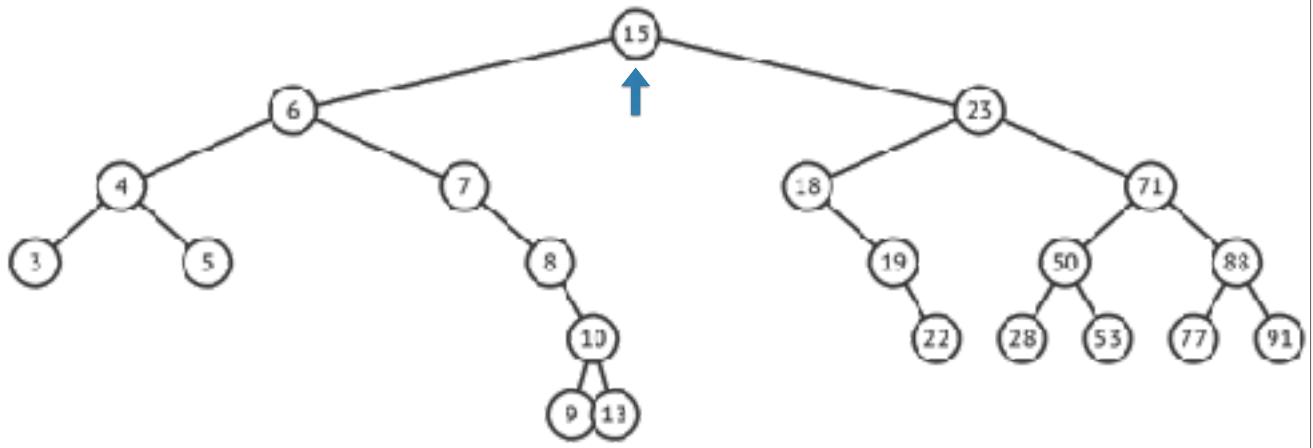
Since this node has no left child, we know it must be the minimum. We just found the minimum value in a tree of five nodes using only two jumps. This is the terrific power of BSTs — in a tree of N nodes, it only takes $\log_2(n)$ checks to find a particular value.

Does this BST contain the value 28?



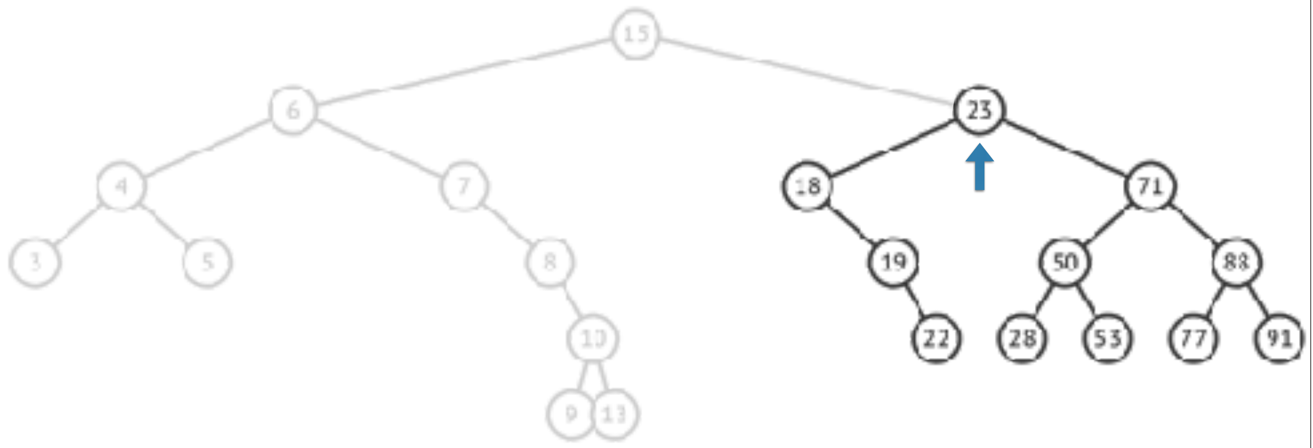
Here is a tree with 23 nodes. Does it contain the value 28? (Yes.) How many jumps did we need to determine that? (just 4).

Does this BST contain the value 28?



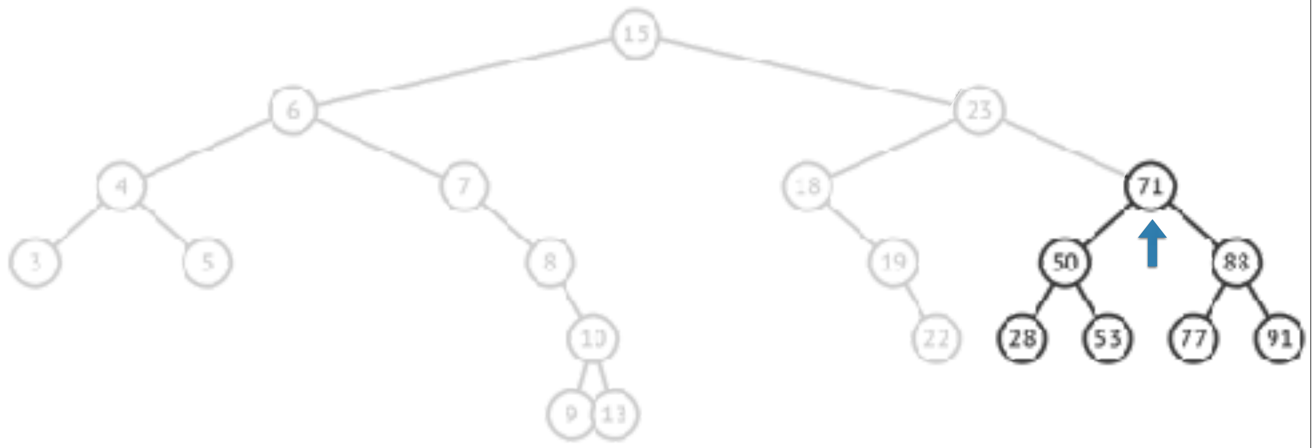
Here is a tree with 23 nodes. Does it contain the value 28? (Yes.) How many jumps did we need to determine that? (just 4).

Does this BST contain the value 28?



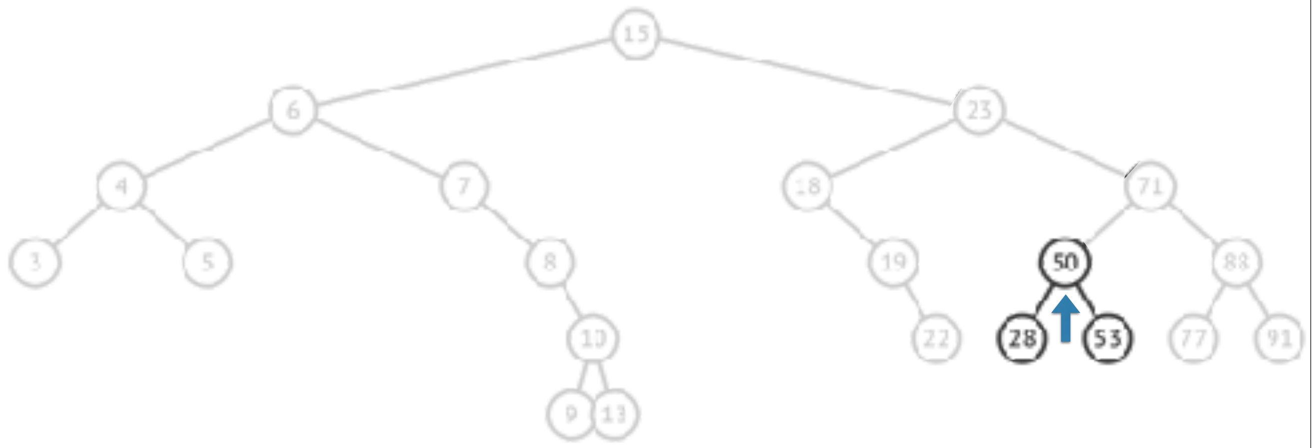
Here is a tree with 23 nodes. Does it contain the value 28? (Yes.) How many jumps did we need to determine that? (just 4).

Does this BST contain the value 28?

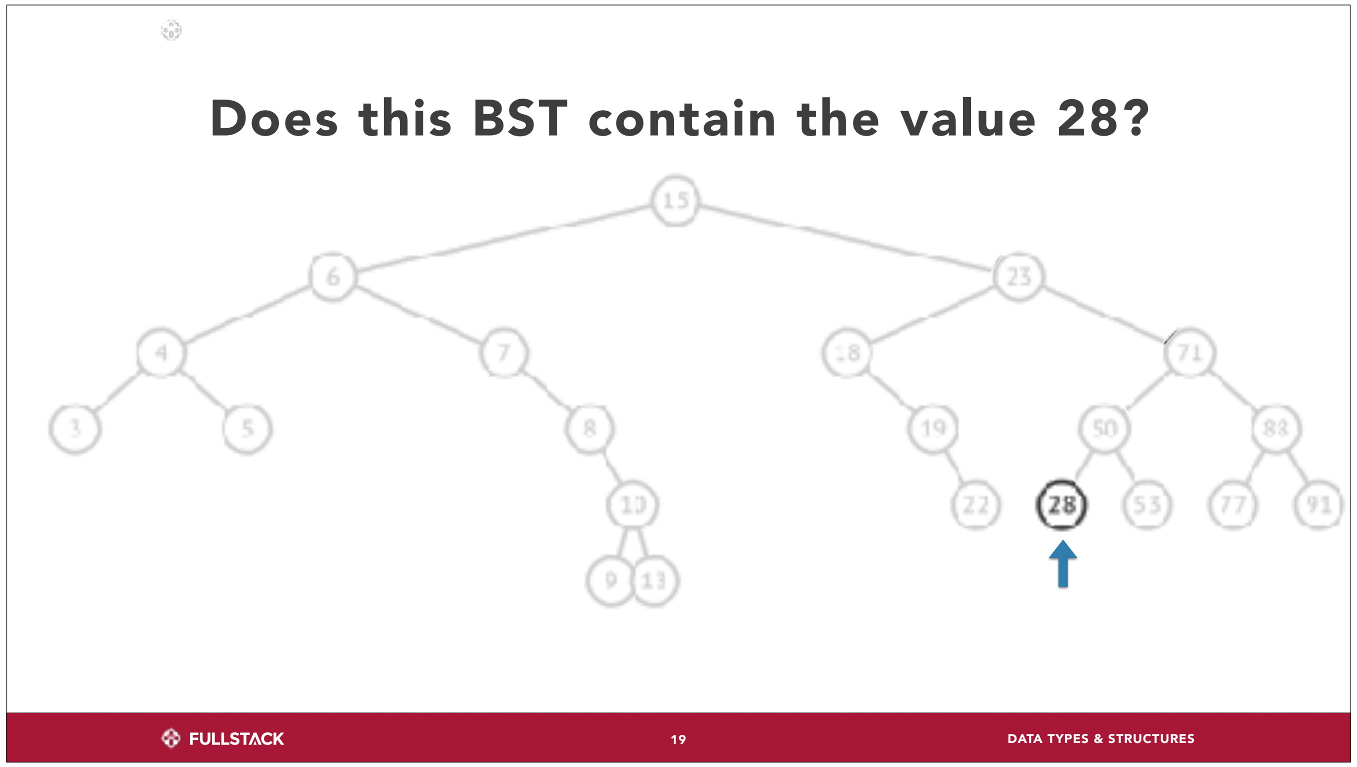


Here is a tree with 23 nodes. Does it contain the value 28? (Yes.) How many jumps did we need to determine that? (just 4).

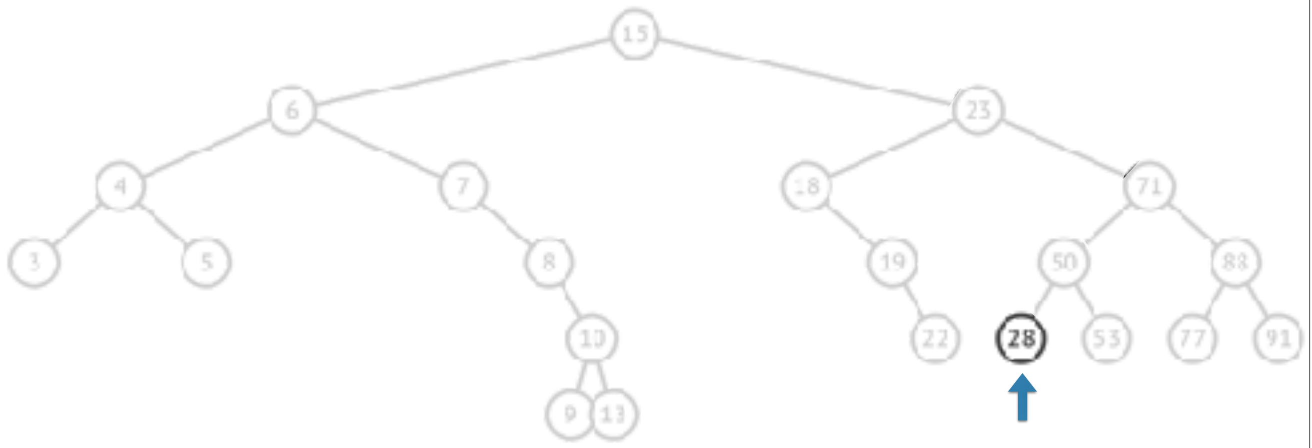
Does this BST contain the value 28?



Here is a tree with 23 nodes. Does it contain the value 28? (Yes.) How many jumps did we need to determine that? (just 4).



Does this BST contain the value 28?



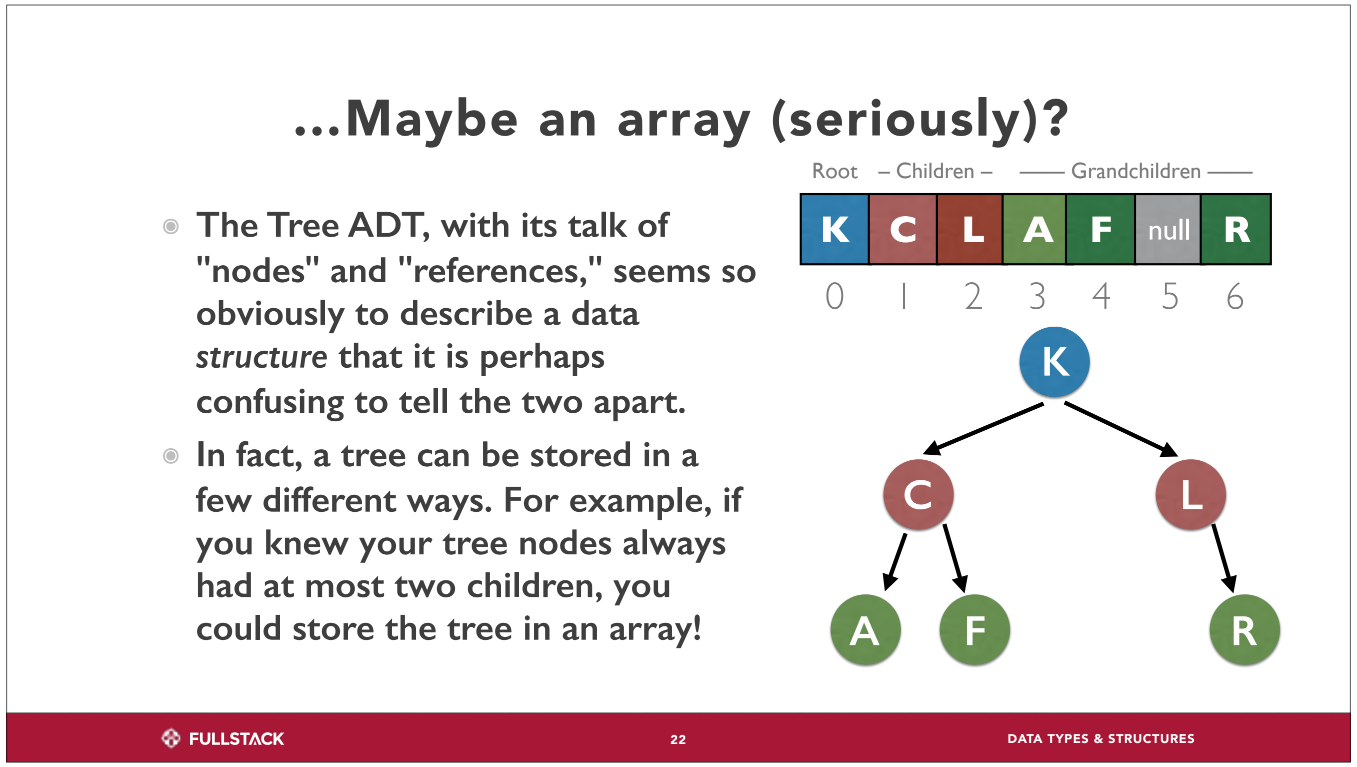
Here is a tree with 23 nodes. Does it contain the value 28? (Yes.) How many jumps did we need to determine that? (just 4).

BST ADT

- **Root node satisfies ordering principle**
  - Left descendants < root value <= right descendants
- **Both children are BSTs (recursive definition)**
- Operations
  - **Insert** new values, *respecting the ordering principle*
  - **Find** existing values (takes advantage of ordering)
  - **Delete** values (tricky, skipped in workshop)

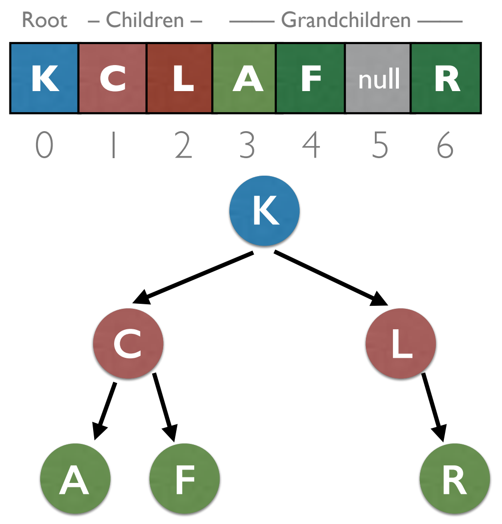
How to implement this ADT?

(ask audience)

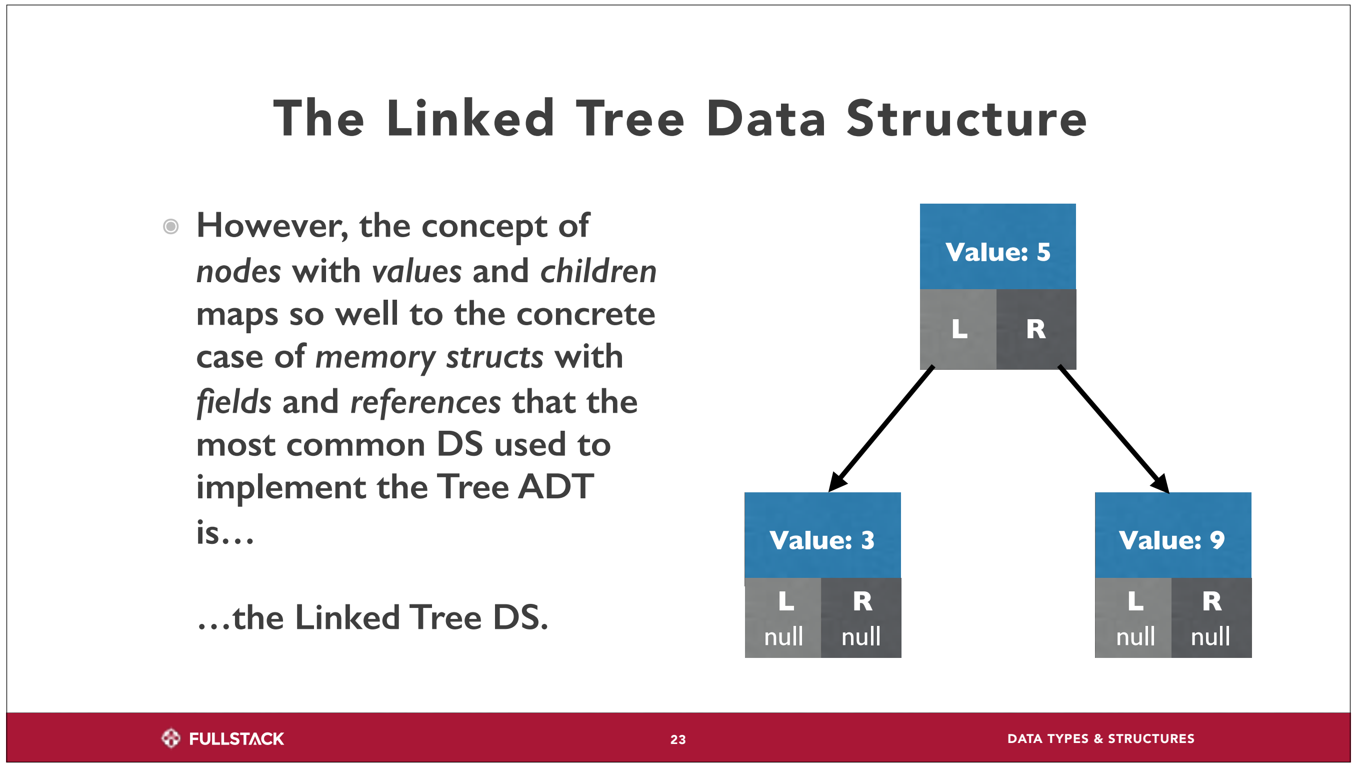


…Maybe an array (seriously)?

- The Tree ADT, with its talk of "nodes" and "references," seems so obviously to describe a data *structure* that it is perhaps confusing to tell the two apart.
- In fact, a tree can be stored in a few different ways. For example, if you knew your tree nodes always had at most two children, you could store the tree in an array!



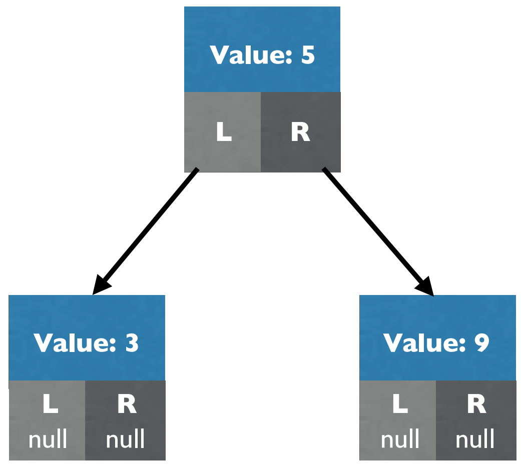
Storing a tree in an array is not as silly as it might seem. In fact, this is how a heap works, and it results in a number of surprising advantages with respect to memory, sorting, etc. You can navigate to any node in the tree using some basic math to determine which array index corresponds to which "position" in the tree. For example, a given node's left child is always located at (node index * 2) + 1 and the right child is at (node index * 2) + 2.



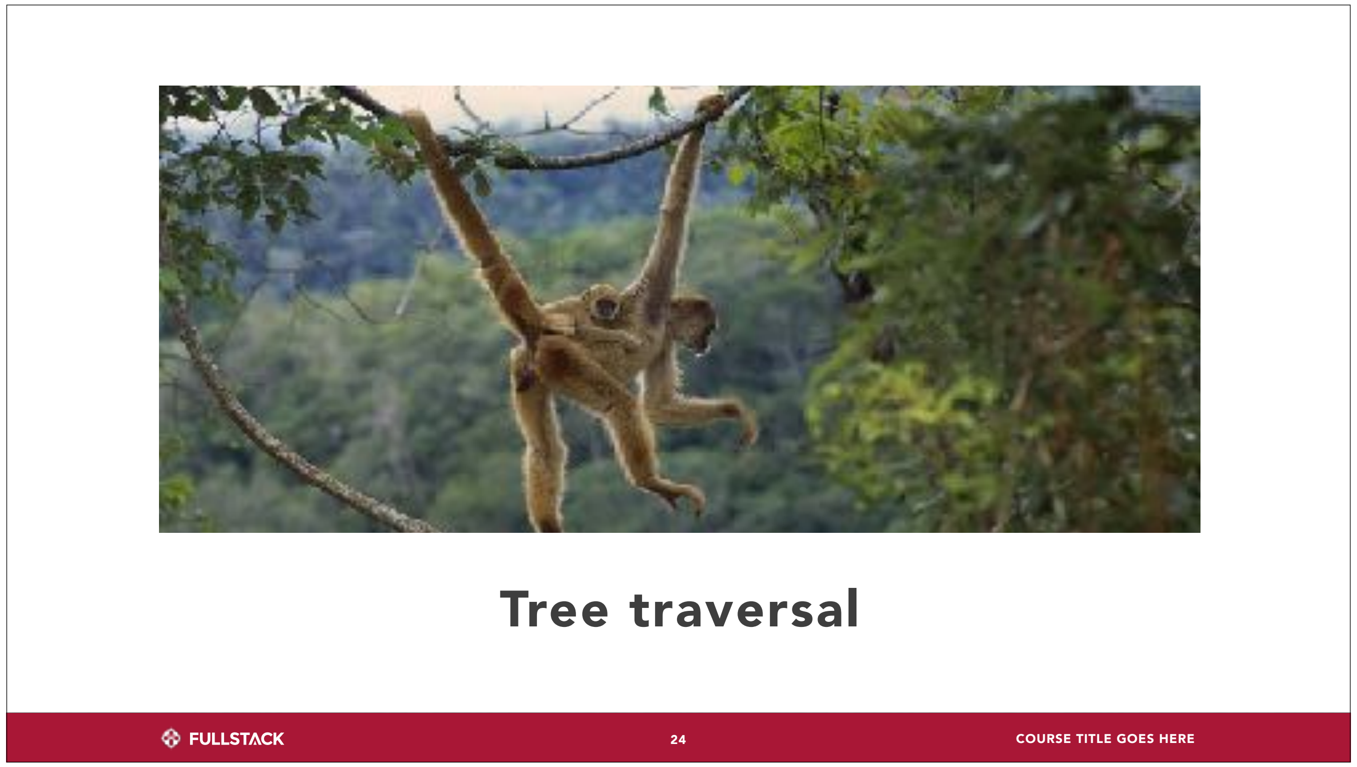
The Linked Tree Data Structure

- However, the concept of *nodes* with *values* and *children* maps so well to the concrete case of *memory structs* with *fields* and *references* that the most common DS used to implement the Tree ADT is…

  …the Linked Tree DS.

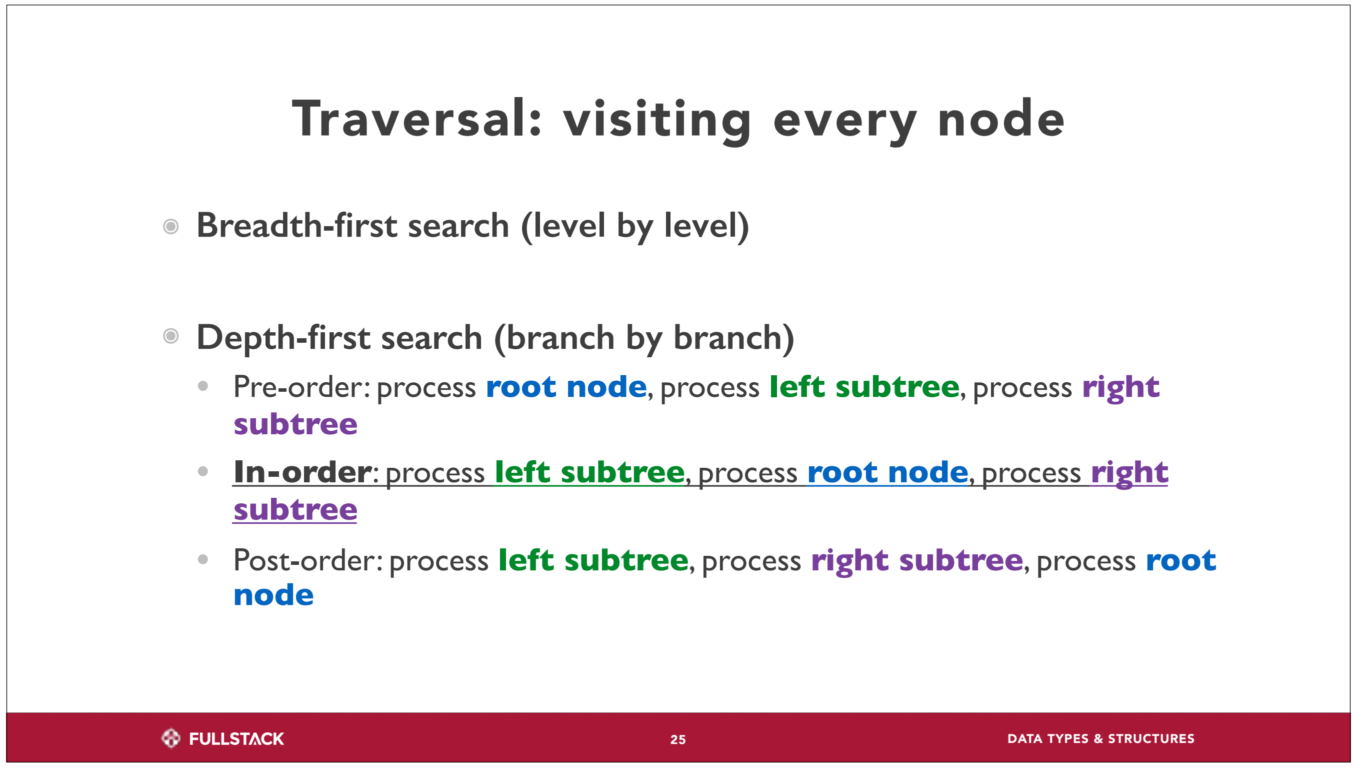


Just like linked lists, each node will be an object with pointers to other objects. This isn't the first time we've seen an ADT with the same name as a closely related DS — for example, the Linked List DS is often used to implement the List ADT (linear collection of elements you can add, delete, insert, etc.).



Tree traversal

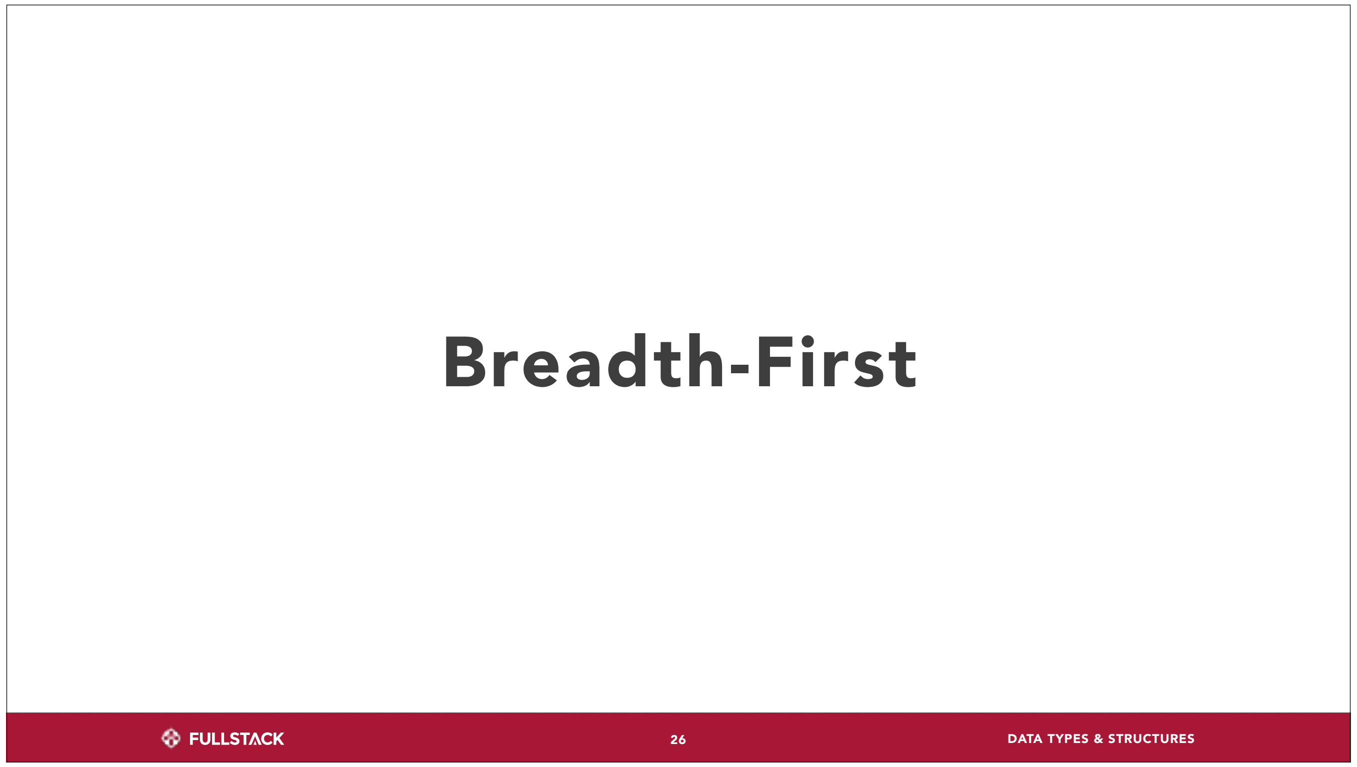
It is frequently necessary to *traverse* a tree — visit all of its nodes systematically, either to find a particular node (in the case of non-search trees) or to process nodes (e.g. print all values).



Traversal: visiting every node

- **Breadth-first search (level by level)**
- **Depth-first search (branch by branch)**
  - Pre-order: process **root node**, process **left subtree**, process **right subtree**
  - **In-order**: process **left subtree**, process **root node**, process **right subtree**
  - Post-order: process **left subtree**, process **right subtree**, process **root node**

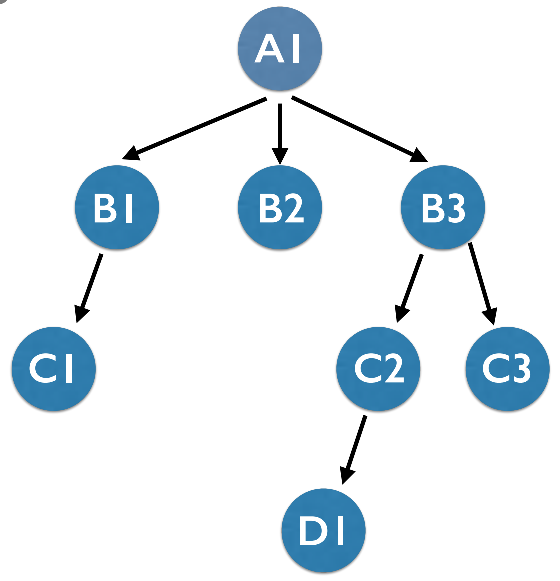
There are two main categories of tree traversal; depth-first and breadth-first. In addition, ordered trees like BSTs have several flavors of depth-first traversal. In general, the two most useful strategies for trees depth-first in-order (for BSTs), and breadth-first (for trees where level is meaningful).



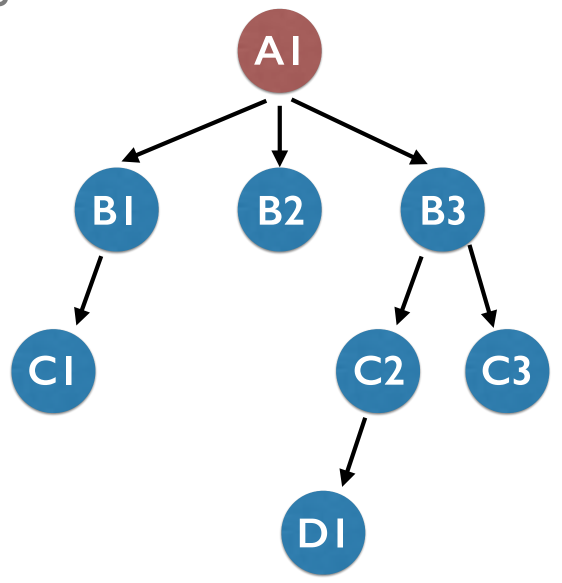
Breadth-First

Breadth-first is most useful when your tree levels have some kind of meaning, which doesn’t happen very often with BSTs, or when your tree is extremely deep but you suspect the node you want is close to the root. Think: org chart, taxonomy, anything hierarchical, shortest path out of a maze, degrees of Kevin Bacon, possible chess moves, etc.

BFS

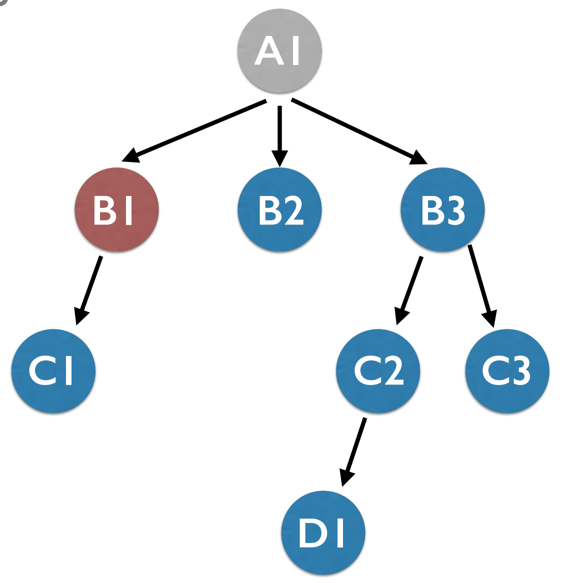


BFS



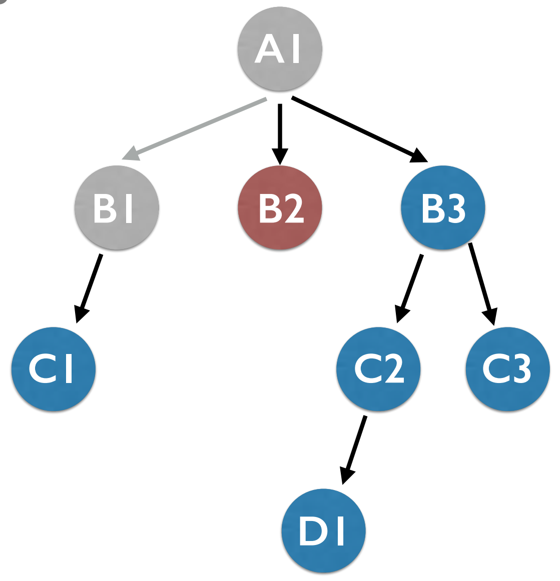
A1

BFS



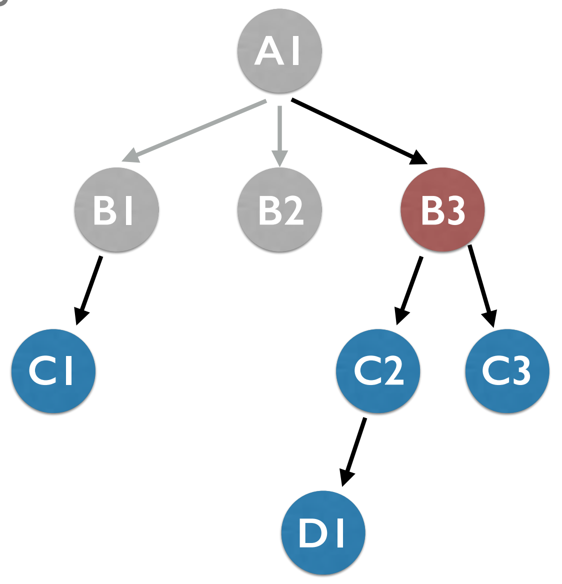
A1, B1

BFS



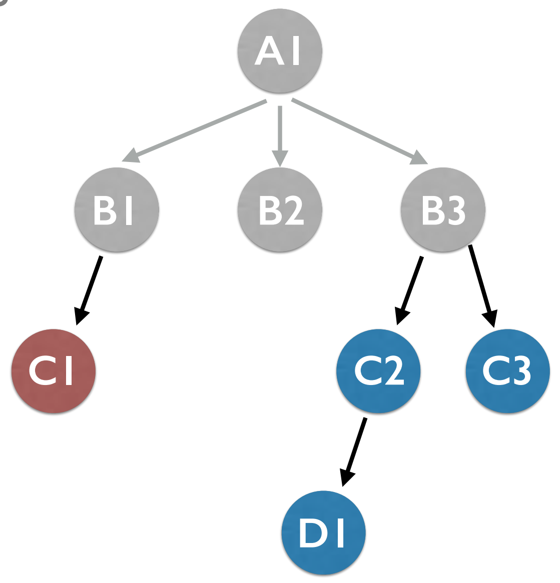
A1, B1, B2

BFS



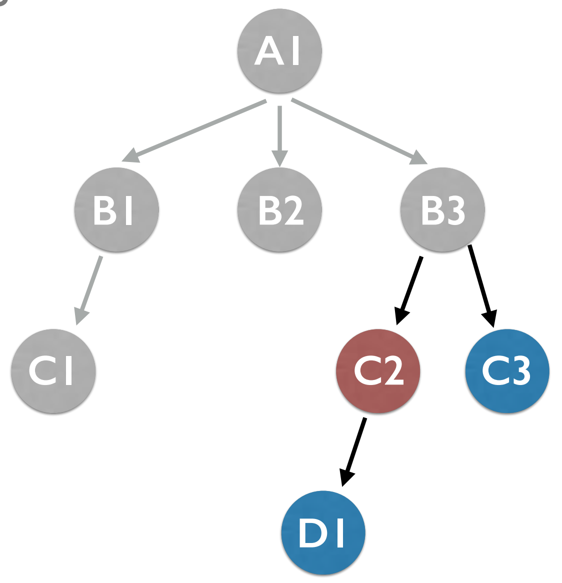
A1, B1, B2, B3

BFS



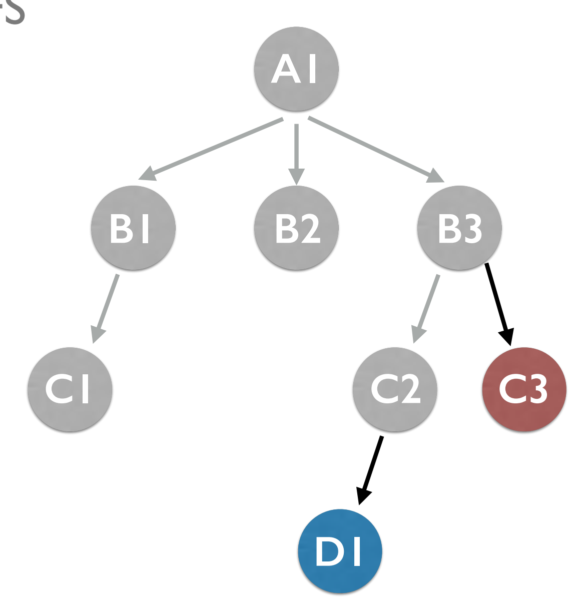
A1, B1, B2, B3, C1

BFS



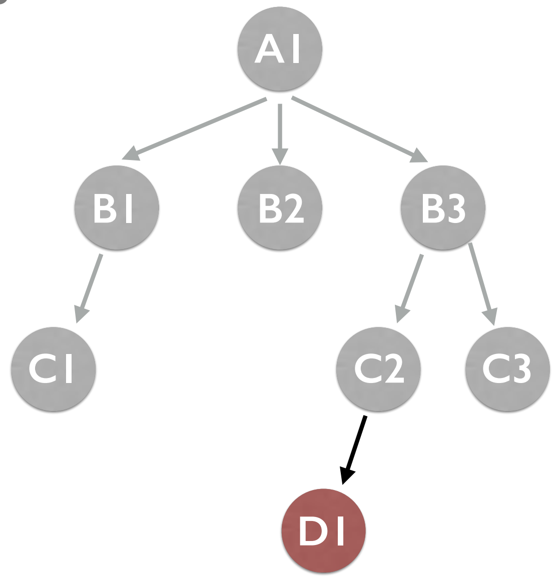
A1, B1, B2, B3, C1, C2

BFS



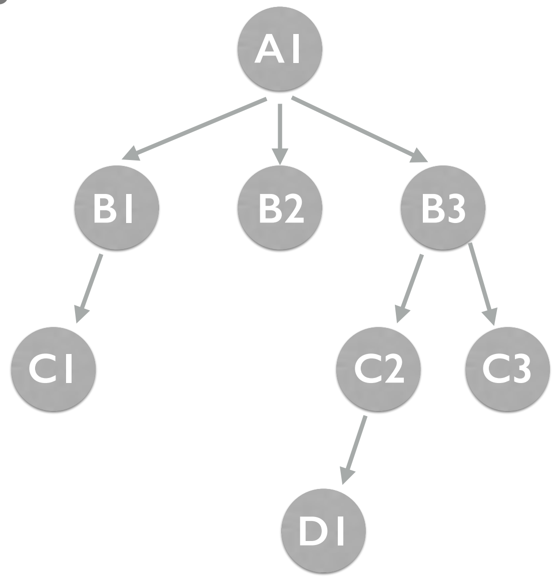
A1, B1, B2, B3, C1, C2, C3

BFS



A1, B1, B2, B3, C1, C2, C3, D1

BFS



A1, B1, B2, B3, C1, C2, C3, D1

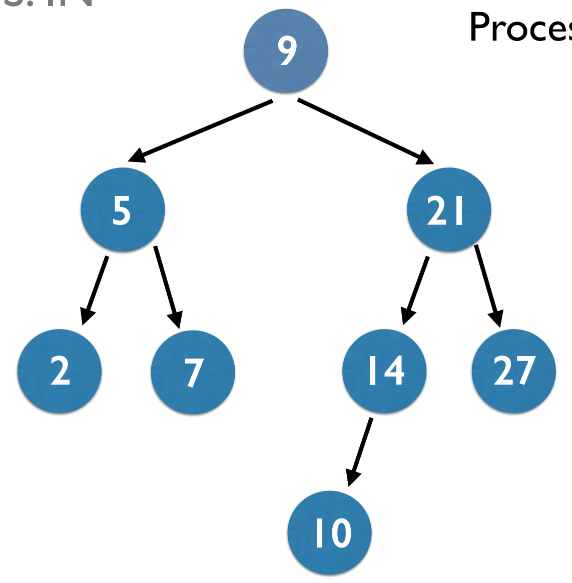
- **Looks easy… but with a tree data structure it actually requires an elegant trick to do correctly.**
- **No full solution here, but a BIG hint: you'll need a queue!**

Depth First: In-Order

In-order traversal is the most generally useful DFS strategy for BSTs. First the left subtree (lower values) is processed; then the current node value is processed; then the right subtree (higher values) is processed. In the end, all values are processed *in order*.

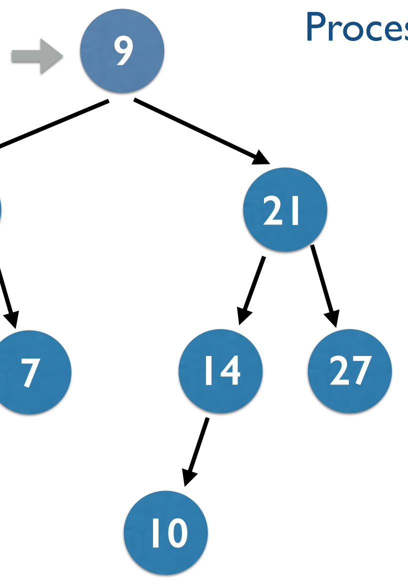
DFS: IN

Process left · Process root · Process right



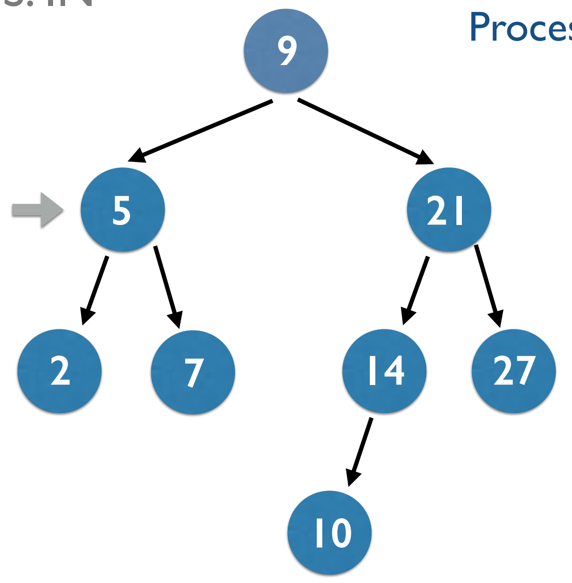
DFS: IN

Process left · Process root · Process right



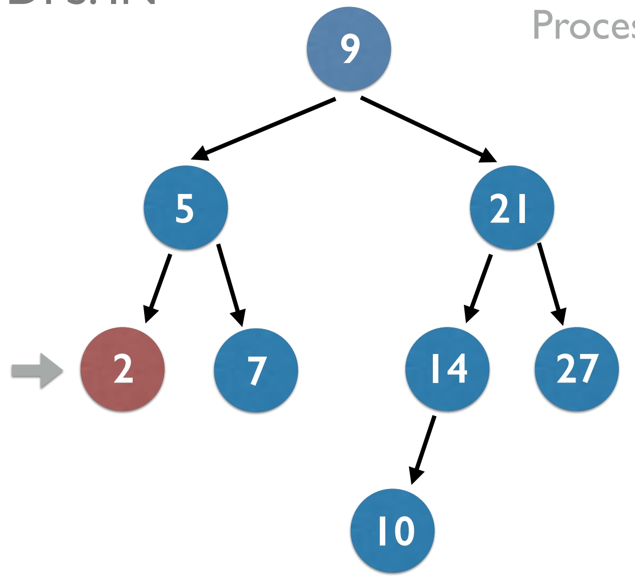
DFS: IN

Process left · Process root · Process right



DFS: IN

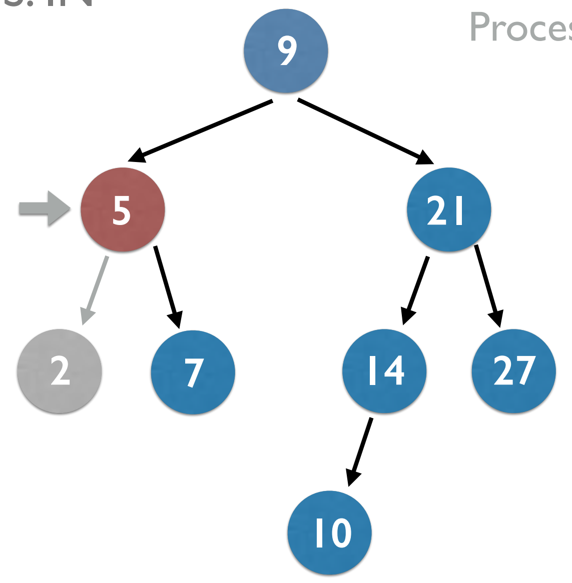
Process left · Process root · Process right



2

DFS: IN

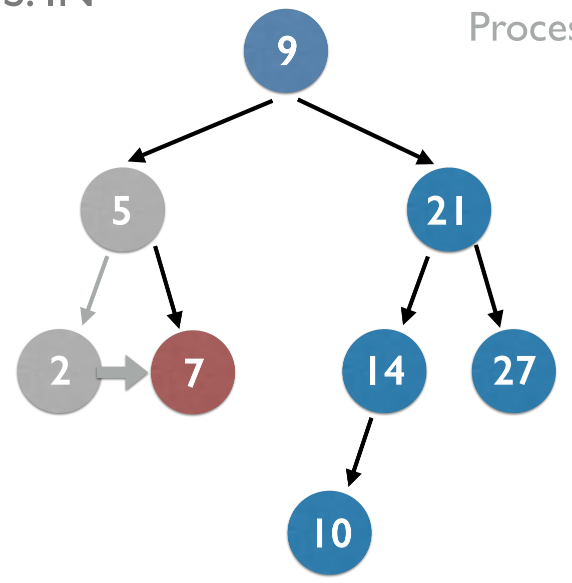
Process left · Process root · Process right



2, 5

DFS: IN

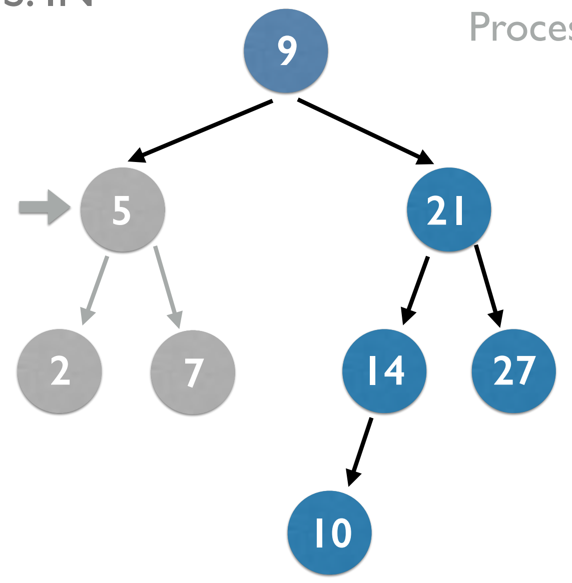
Process left · Process root · Process right



2, 5, 7

DFS: IN

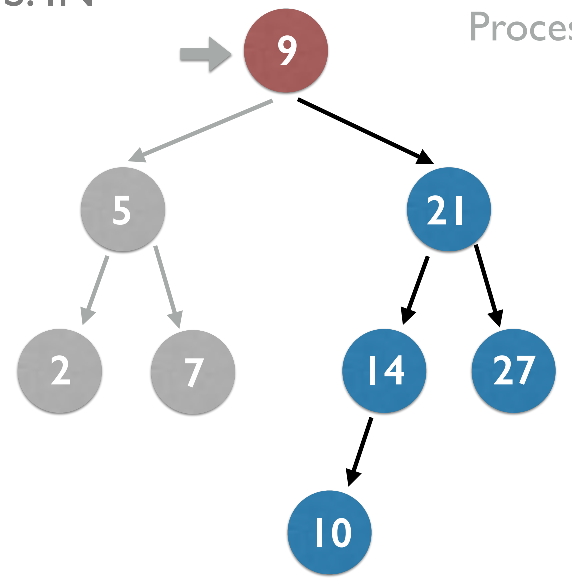
Process left · Process root · Process right



2, 5, 7

DFS: IN

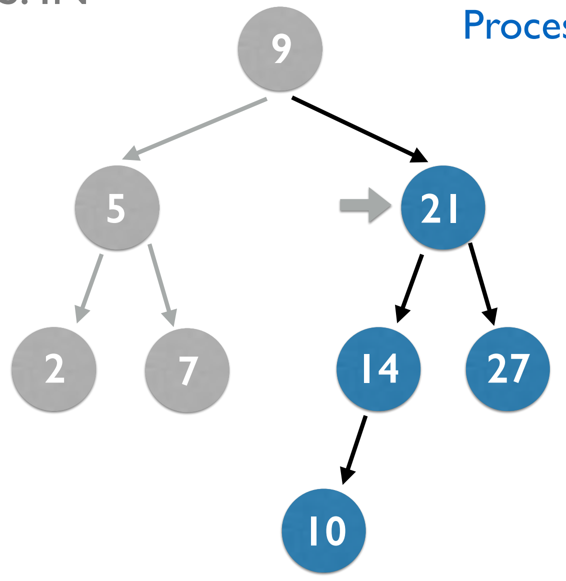
Process left · Process root · Process right



2, 5, 7, 9

DFS: IN

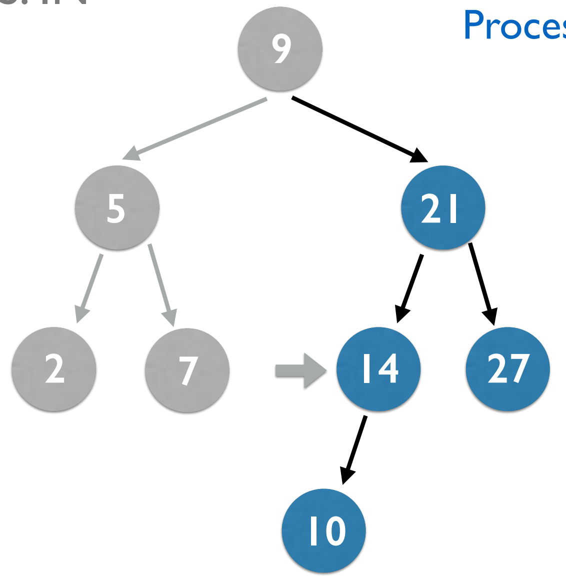
Process left · Process root · Process right



2, 5, 7, 9

DFS: IN

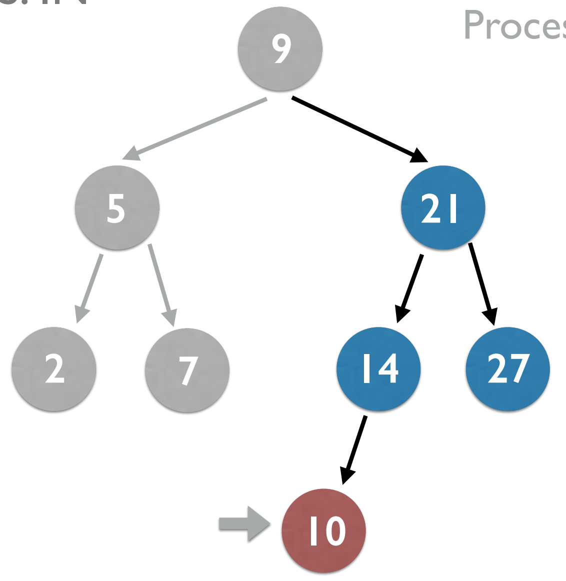
Process left · Process root · Process right



2, 5, 7, 9

DFS: IN

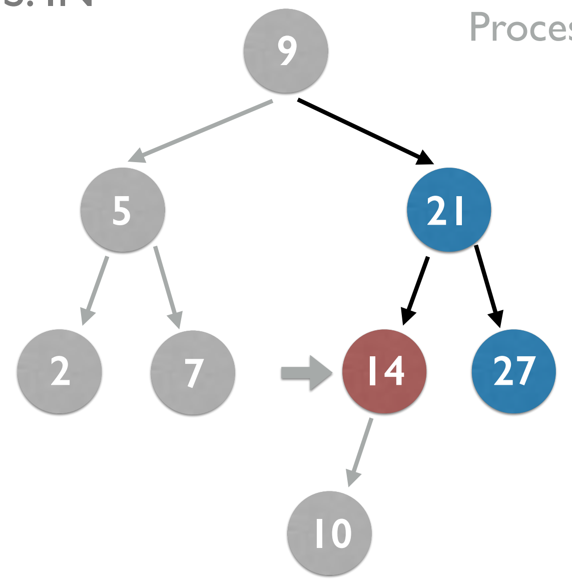
Process left · Process root · Process right



2, 5, 7, 9, 10

DFS: IN

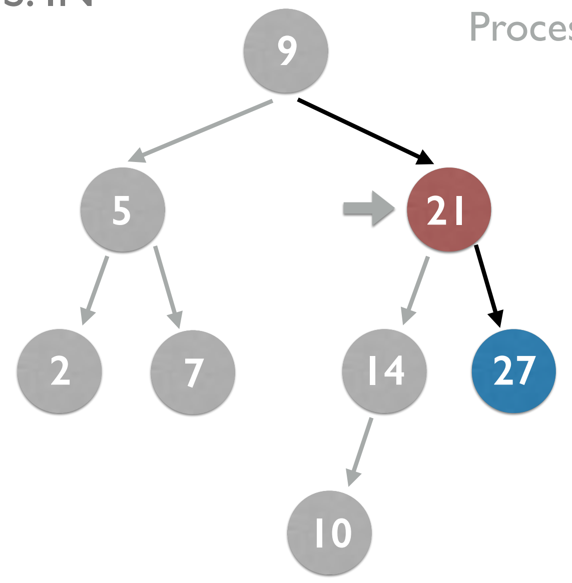
Process left · Process root · Process right



2, 5, 7, 9, 10, 14

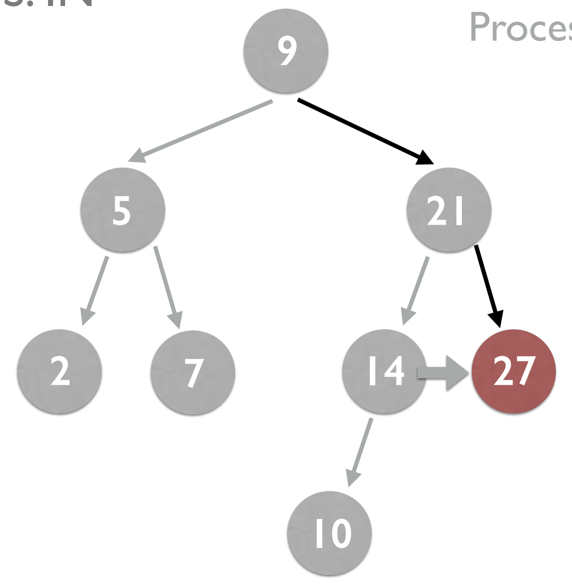
DFS: IN

Process left · Process root · Process right



2, 5, 7, 9, 10, 14, 21

DFS: IN

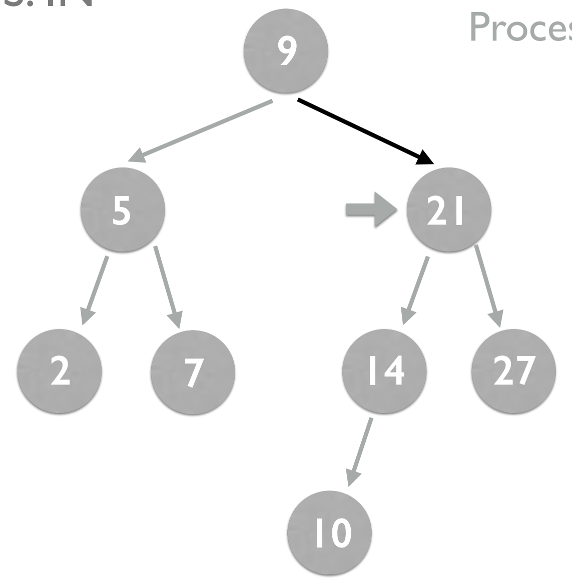


Process left · Process root · Process right

2, 5, 7, 9, 10, 14, 21, 27

DFS: IN

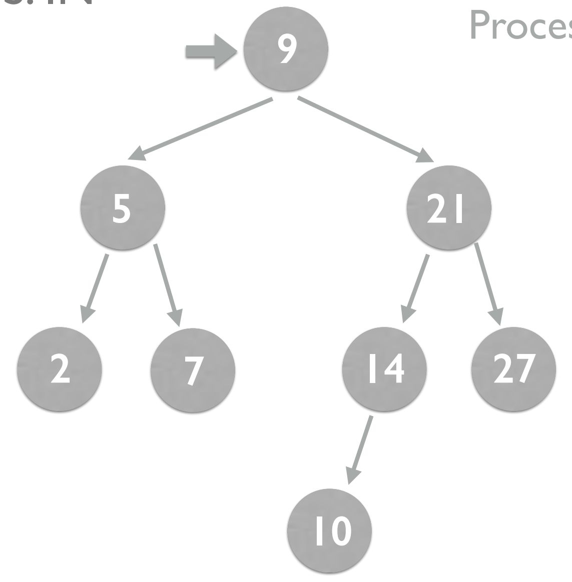
Process left · Process root · Process right



2, 5, 7, 9, 10, 14, 21, 27

DFS: IN

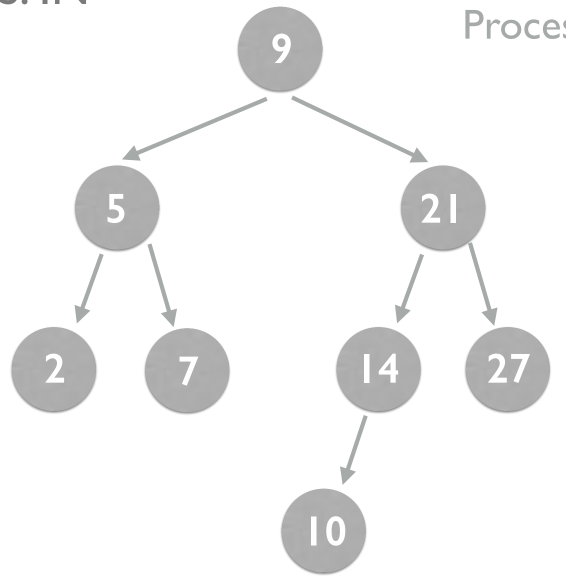
Process left · Process root · Process right



2, 5, 7, 9, 10, 14, 21, 27

DFS: IN

Process left · Process root · Process right

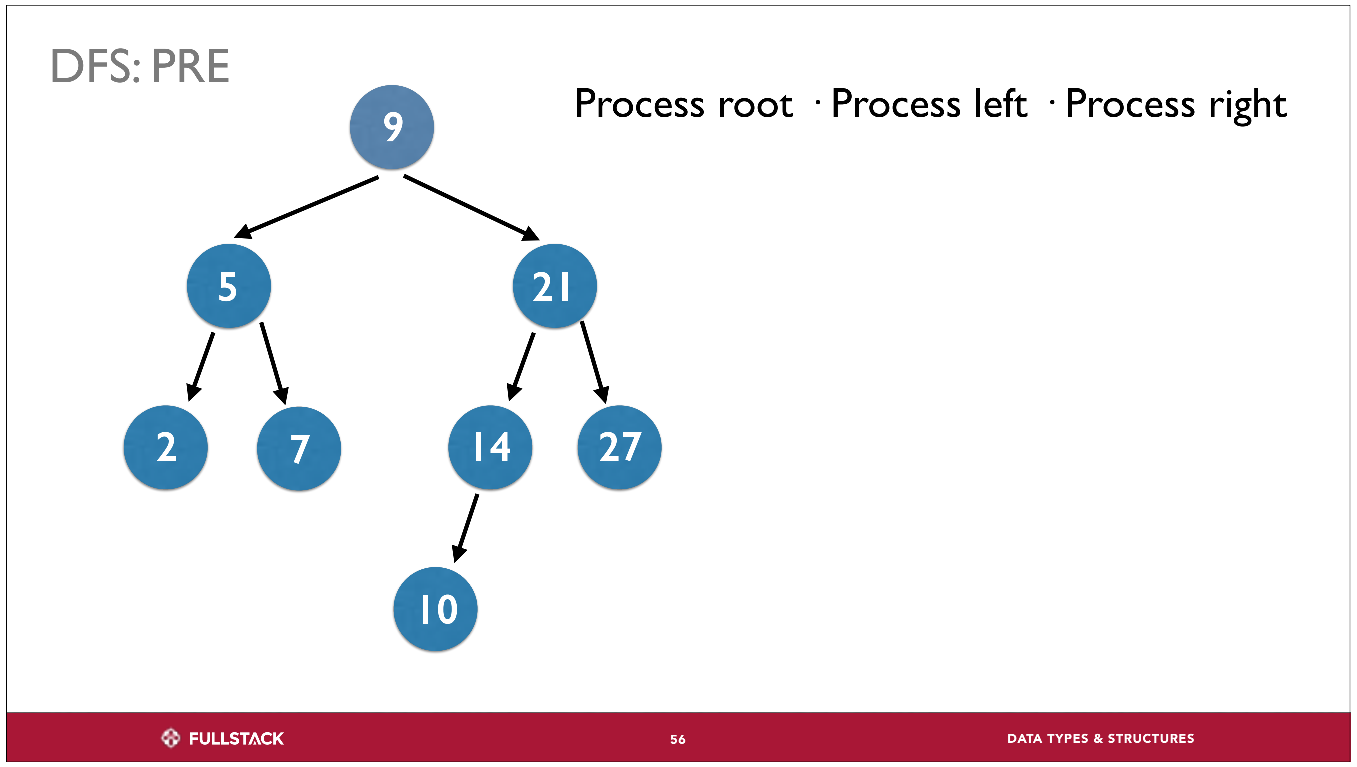


2, 5, 7, 9, 10, 14, 21, 27

- In-order because respects ordering of nodes — nodes are processed smallest to largest value (leftmost to rightmost).
- The most generally useful DFS strategy for BSTs.

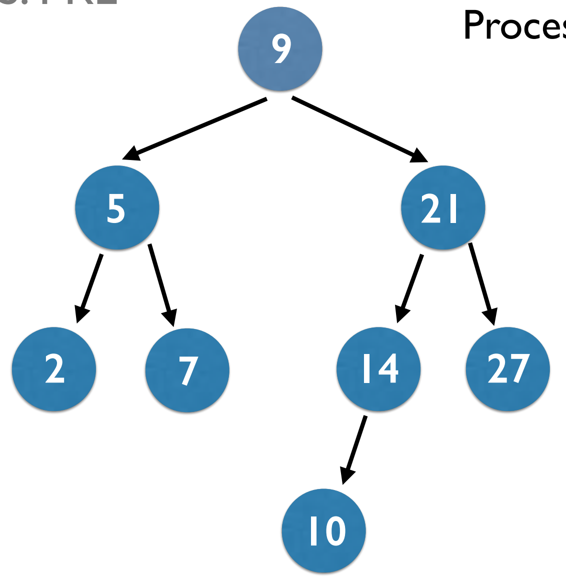
Depth First: Pre-Order

In pre-order traversal, nodes are processed immediately upon visitation, and then the left and right subtrees are processed after the fact. The biggest use case for this traversal is *copying* a tree — if you insert the processed node values into a new BST, you get a replica with the correct structure.



DFS: PRE

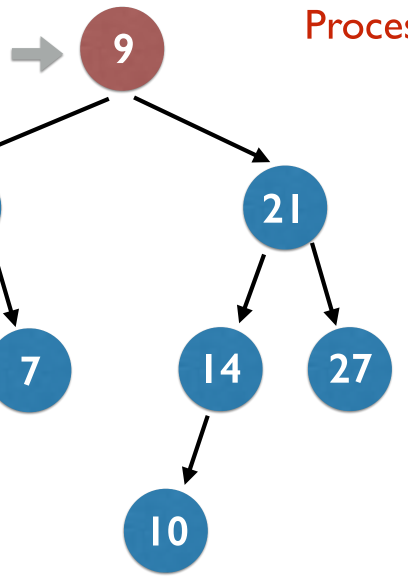
Process root · Process left · Process right



< ANIMATION BEGINS ON CLICK >

DFS: PRE

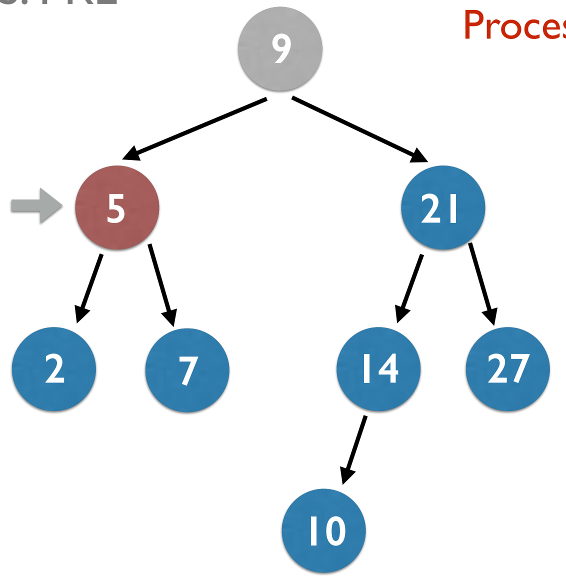
Process root · Process left · Process right



9

DFS: PRE

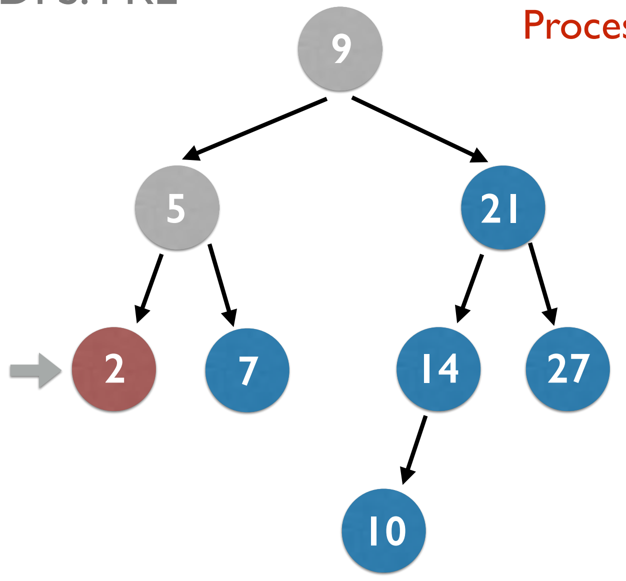
Process root · Process left · Process right



9, 5

DFS: PRE

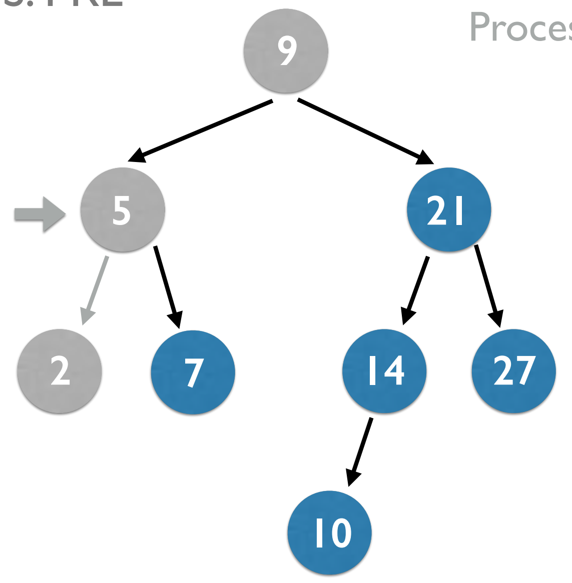
Process root · Process left · Process right



9, 5, 2

DFS: PRE

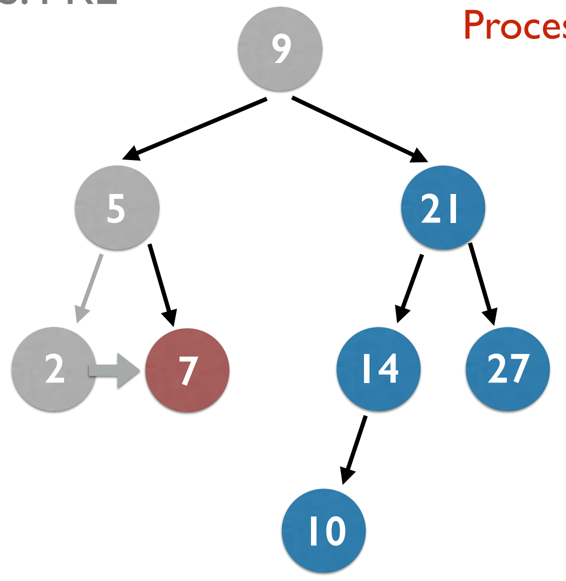
Process root · Process left · Process right



9, 5, 2

DFS: PRE

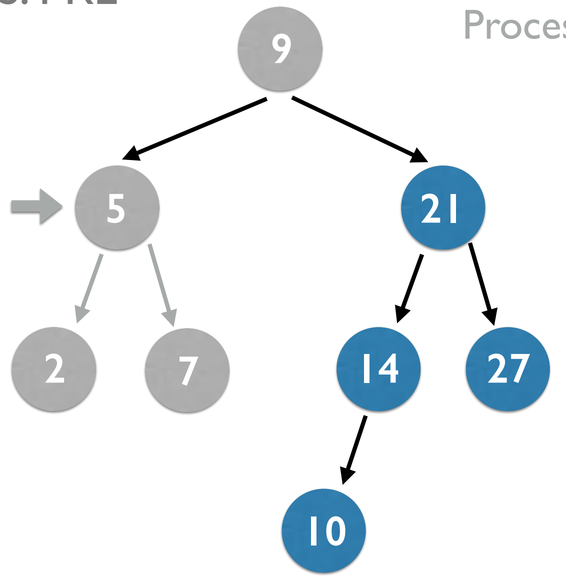
Process root · Process left · Process right



9, 5, 2, 7

DFS: PRE

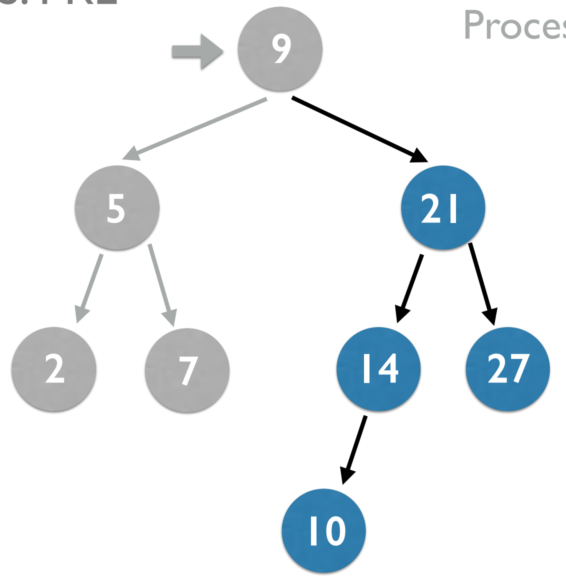
Process root · Process left · Process right



9, 5, 2, 7

DFS: PRE

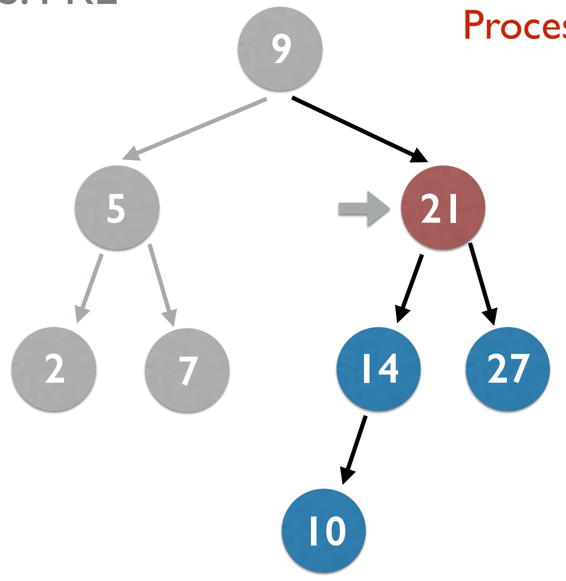
Process root · Process left · **Process right**



9, 5, 2, 7

DFS: PRE

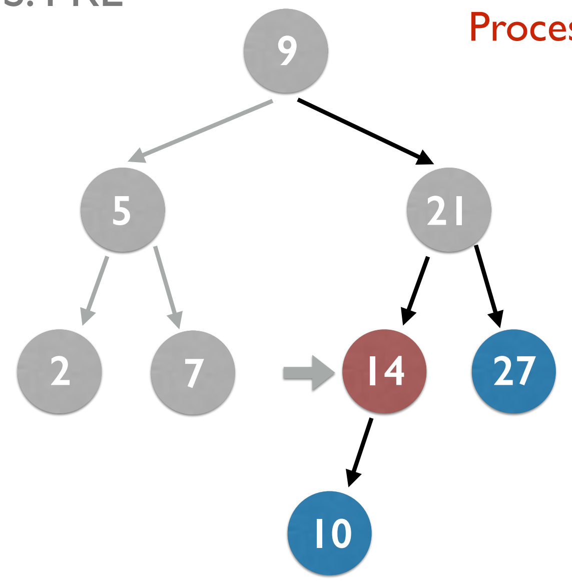
Process root · Process left · Process right



9, 5, 2, 7, 21

DFS: PRE

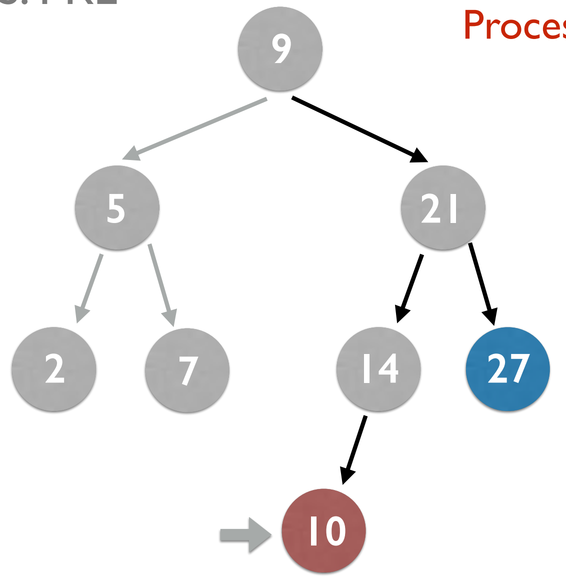
Process root · Process left · Process right



9, 5, 2, 7, 21, 14

DFS: PRE

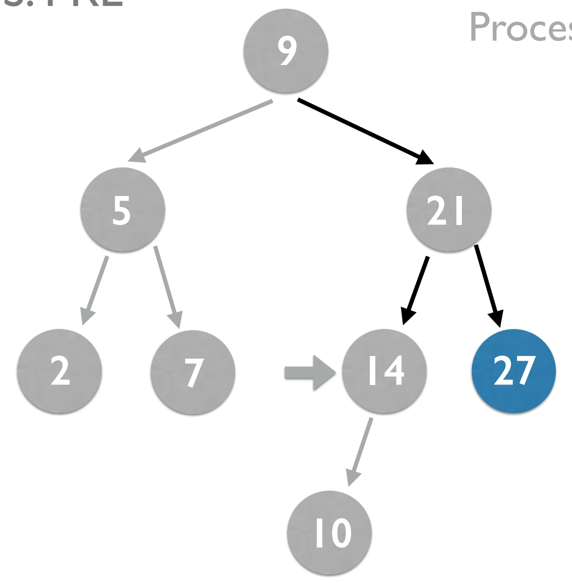
Process root · Process left · Process right



9, 5, 2, 7, 21, 14, 10

DFS: PRE

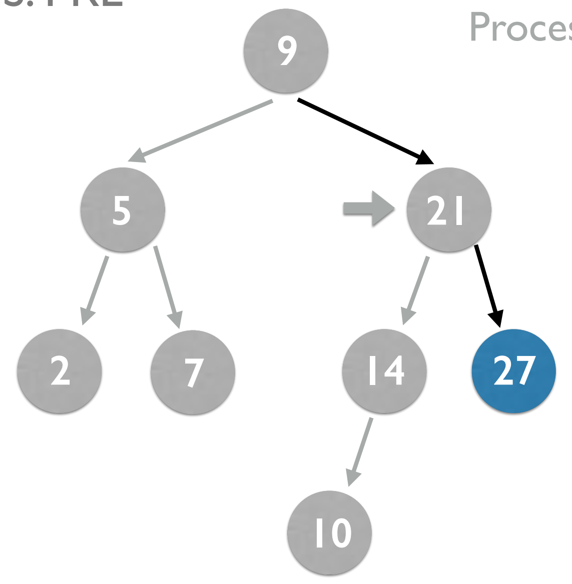
Process root · Process left · Process right



9, 5, 2, 7, 21, 14, 10

DFS: PRE

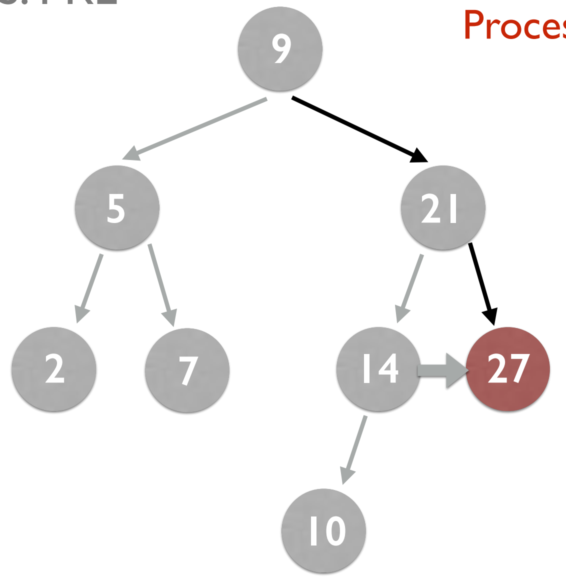
Process root · Process left · Process right



9, 5, 2, 7, 21, 14, 10

DFS: PRE

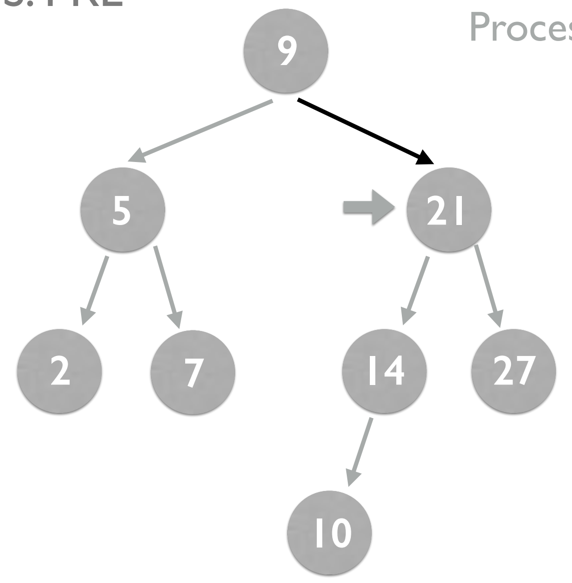
Process root · Process left · Process right



9, 5, 2, 7, 21, 14, 10, 27

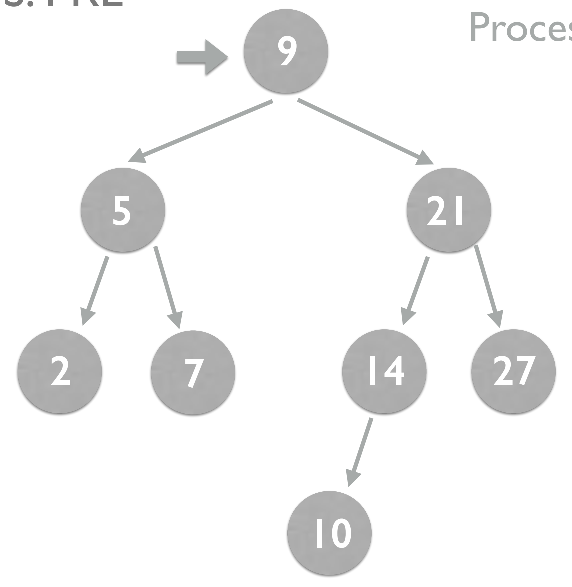
DFS: PRE

Process root · Process left · Process right



9, 5, 2, 7, 21, 14, 10, 27

DFS: PRE

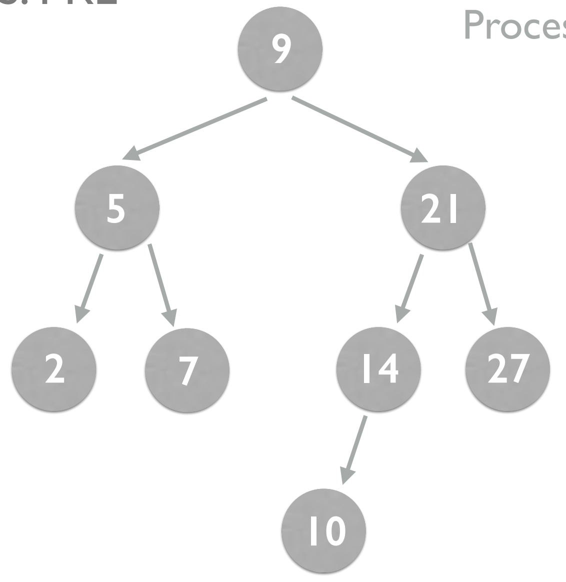


Process root · Process left · Process right

9, 5, 2, 7, 21, 14, 10, 27

DFS: PRE

Process root · Process left · Process right



9, 5, 2, 7, 21, 14, 10, 27

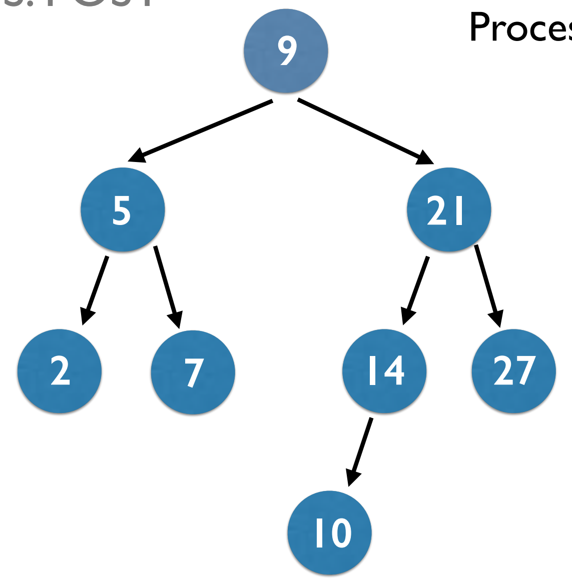
- Output seems random, but actually this has one notable use case. If you create a BST by inserting these values in this order, you get a copy of the original tree. However, the same is true for breadth-first.

Depth First: Post-Order

In post-order traversal, the left and right subtrees are processed first, and only then is the current node processed. The main use case for this traversal is safely deleting a tree in lower-level languages where you have to manage memory carefully (no automatic garbage collection).

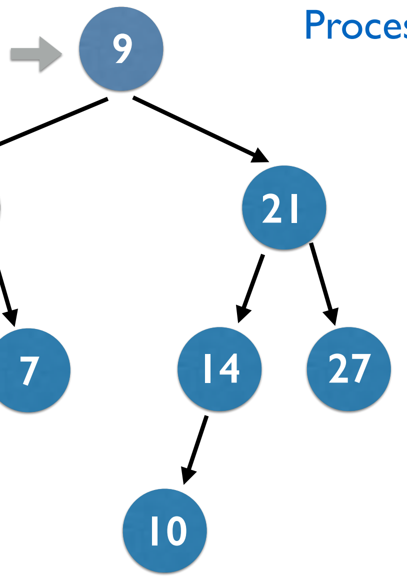
DFS: POST

Process left · Process right · Process root



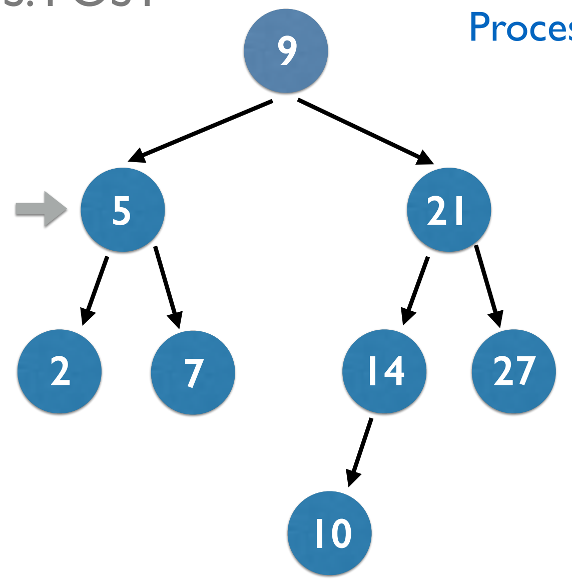
DFS: POST

Process left · Process right · Process root



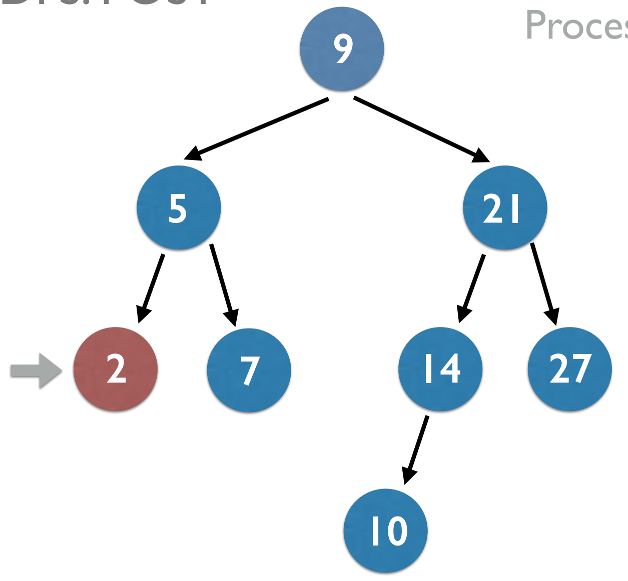
DFS: POST

Process left · Process right · Process root



DFS: POST

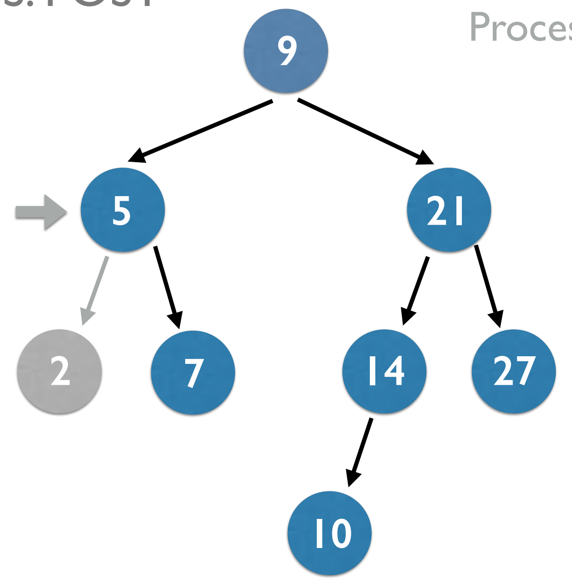
Process left · Process right · Process root



2

DFS: POST

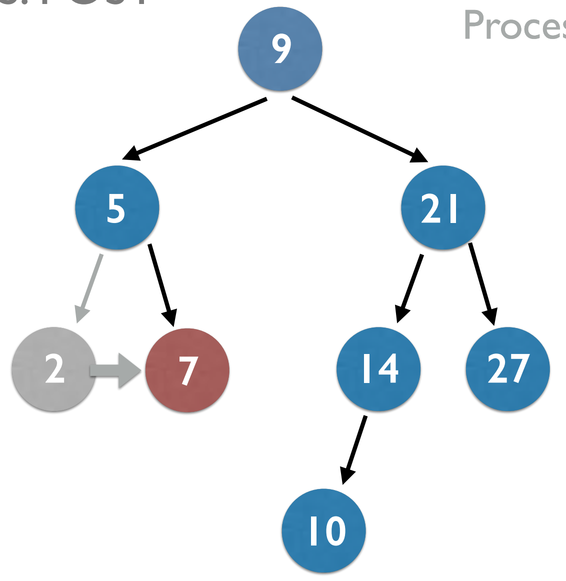
Process left · Process right · Process root



2

DFS: POST

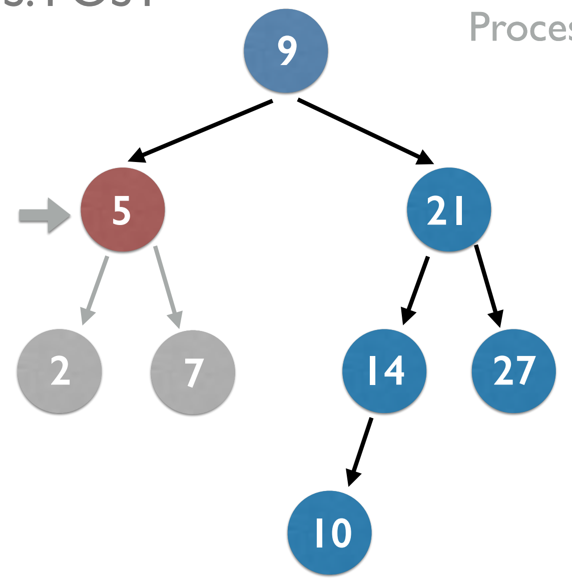
Process left · Process right · **Process root**



2, 7

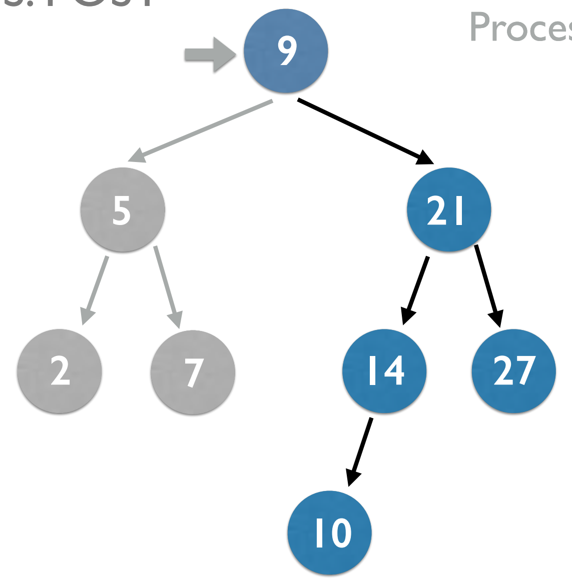
DFS: POST

Process left · Process right · Process root



2, 7, 5

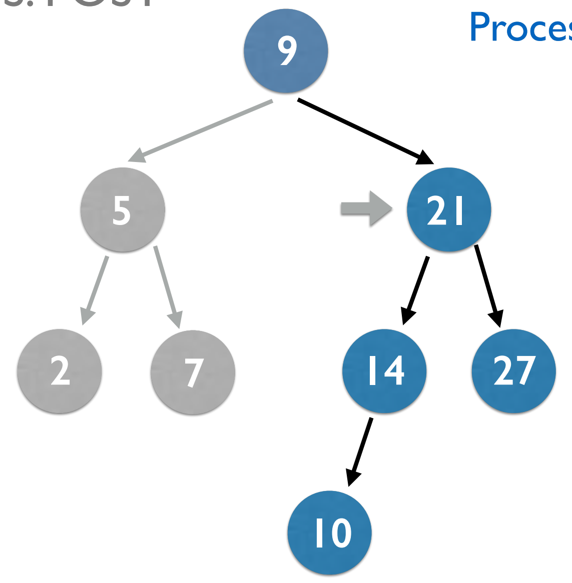
DFS: POST



Process left · Process right · Process root

2, 7, 5

DFS: POST

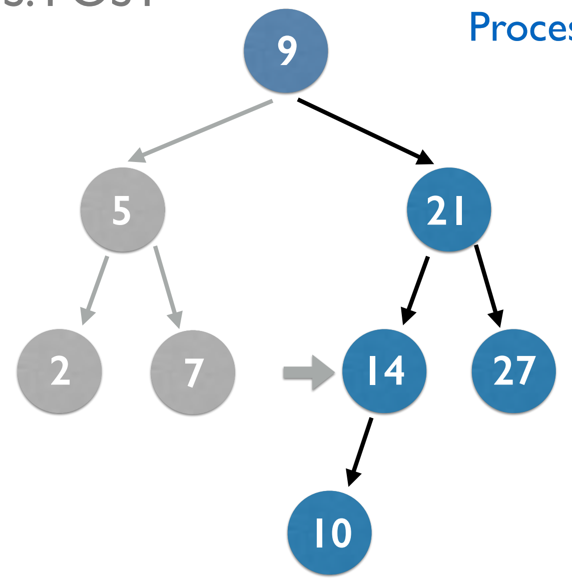


Process left · Process right · Process root

2, 7, 5

DFS: POST

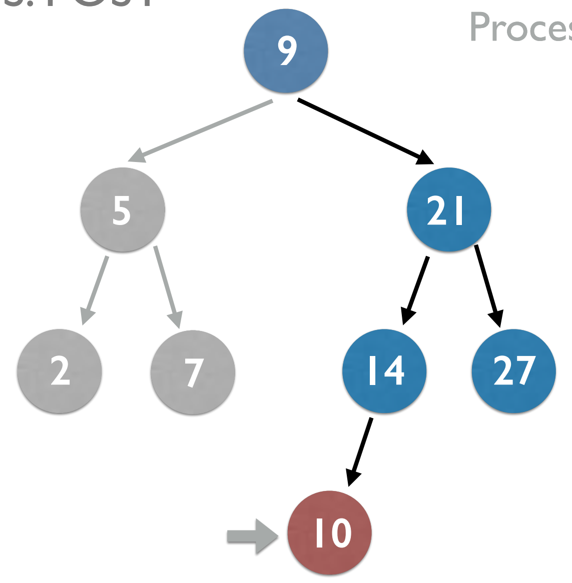
Process left · Process right · Process root



2, 7, 5

DFS: POST

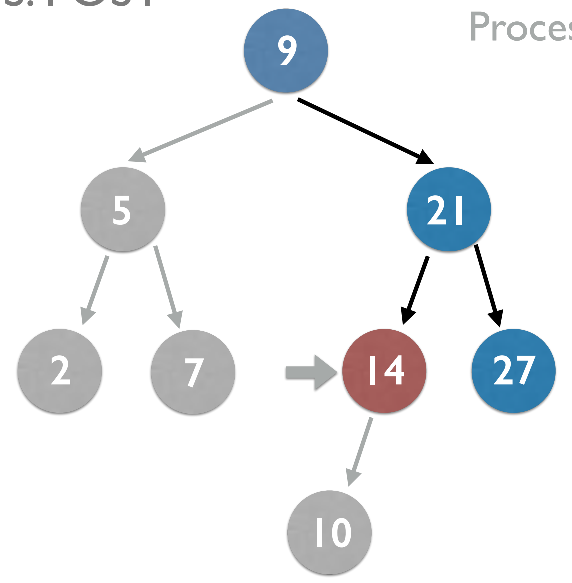
Process left · Process right · Process root



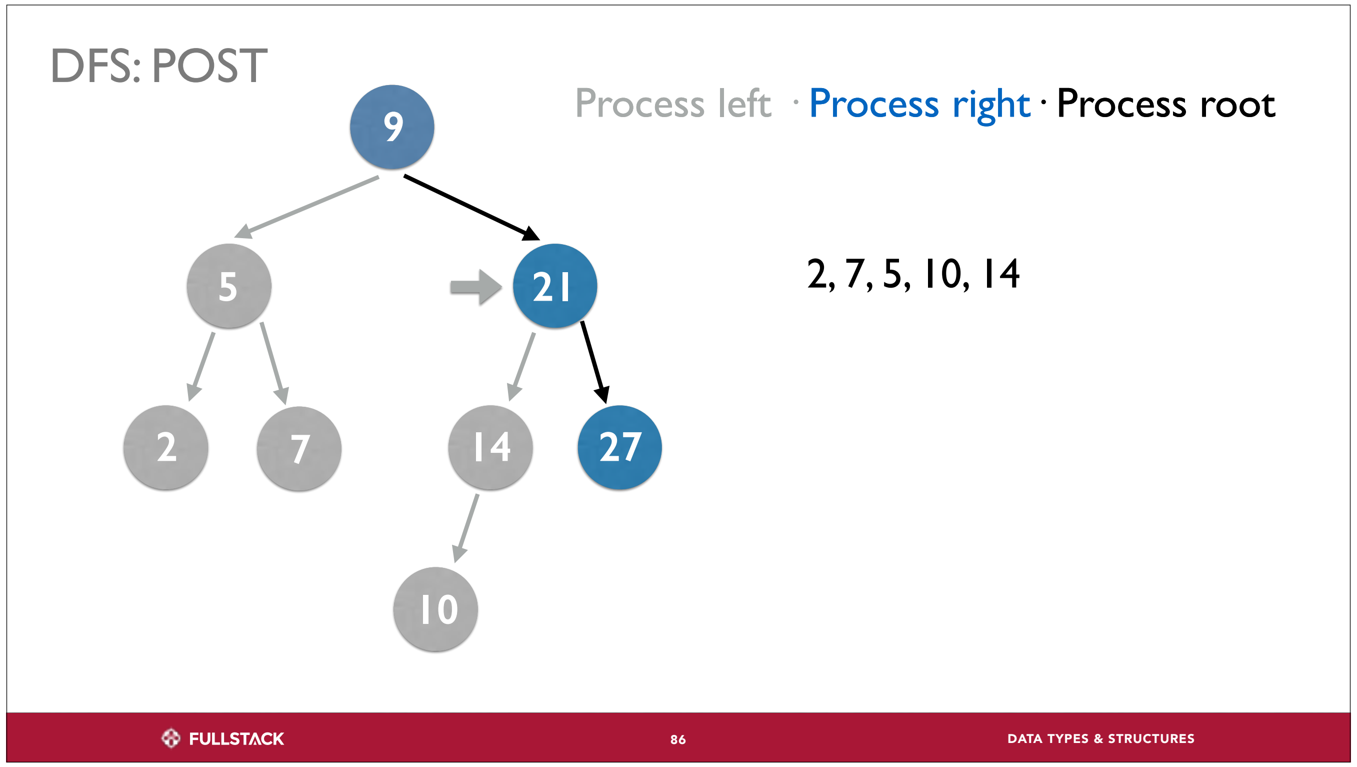
2, 7, 5, 10

DFS: POST

Process left · Process right · Process root

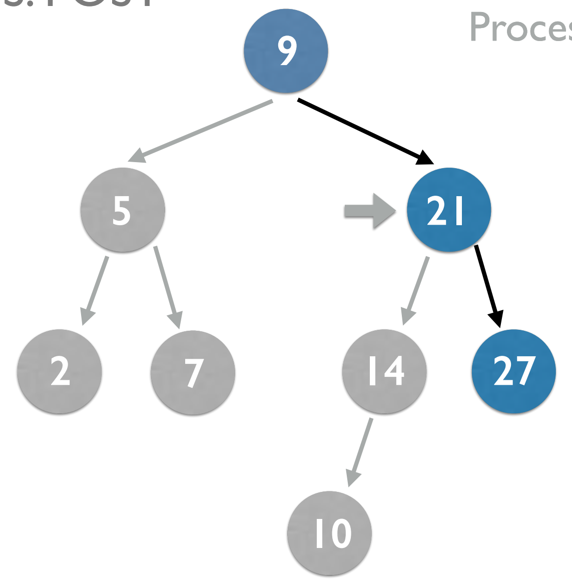


2, 7, 5, 10, 14



DFS: POST

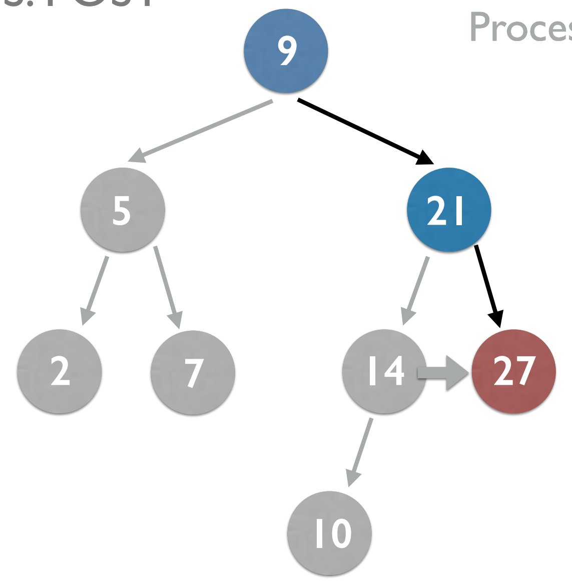
Process left · Process right · Process root



2, 7, 5, 10, 14

DFS: POST

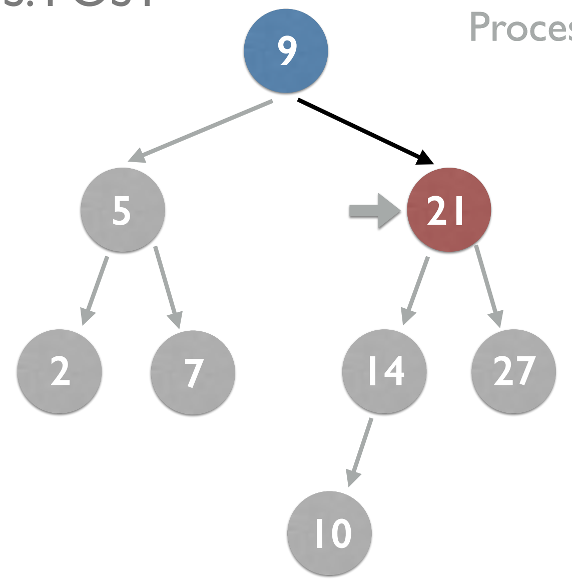
Process left · Process right · Process root



2, 7, 5, 10, 14, 27

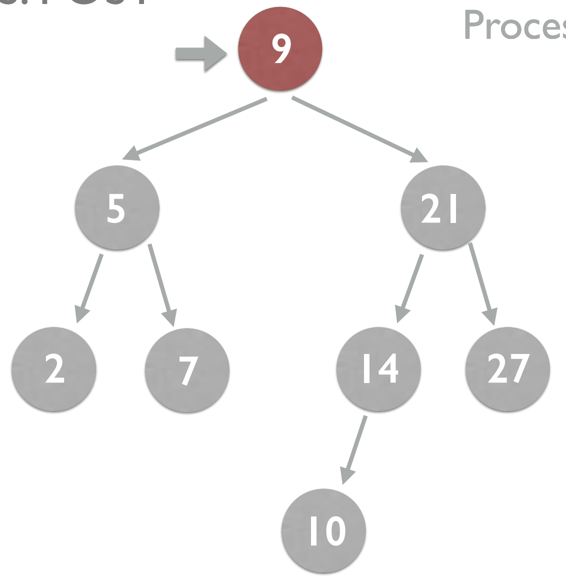
DFS: POST

Process left · Process right · Process root



2, 7, 5, 10, 14, 27, 21

DFS: POST

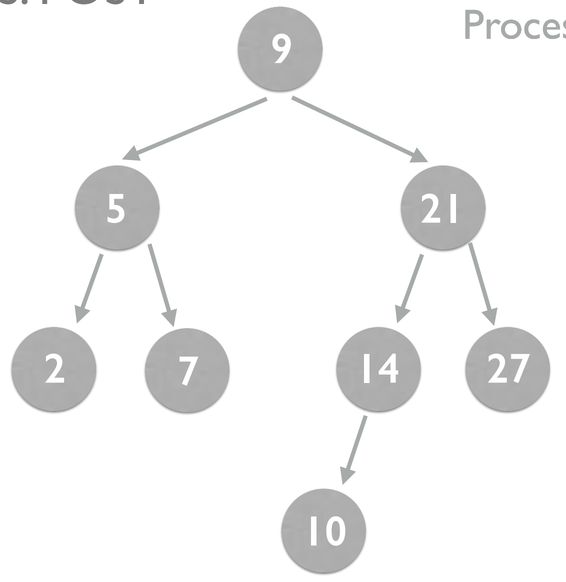


Process left · Process right · Process root

2, 7, 5, 10, 14, 27, 21, 9

DFS: POST

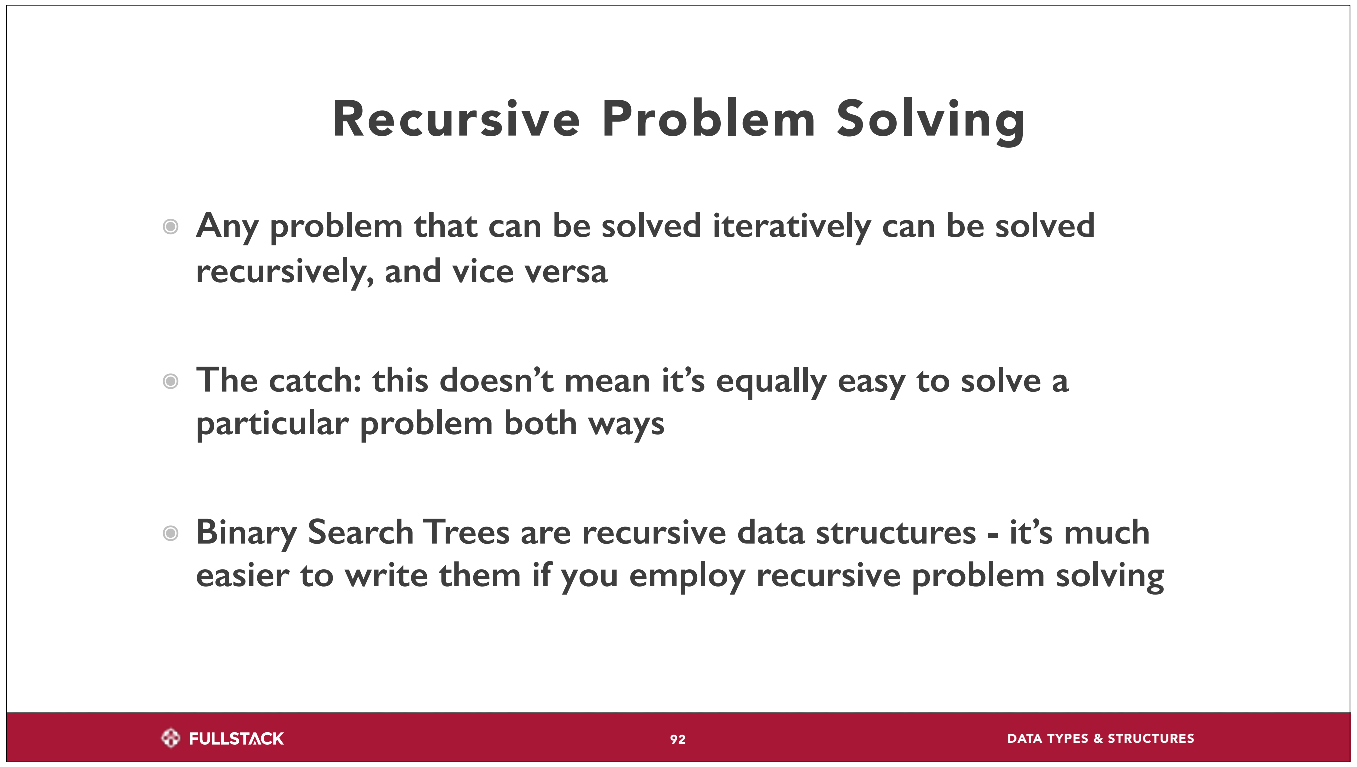
Process left · Process right · Process root



2, 7, 5, 10, 14, 27, 21, 9

- **Main use case is to safely delete a tree leaf by leaf, in lower-level languages (e.g. C) with no automatic garbage collection. Nodes are only processed once all their descendants have been processed.**

Recursive Problem Solving



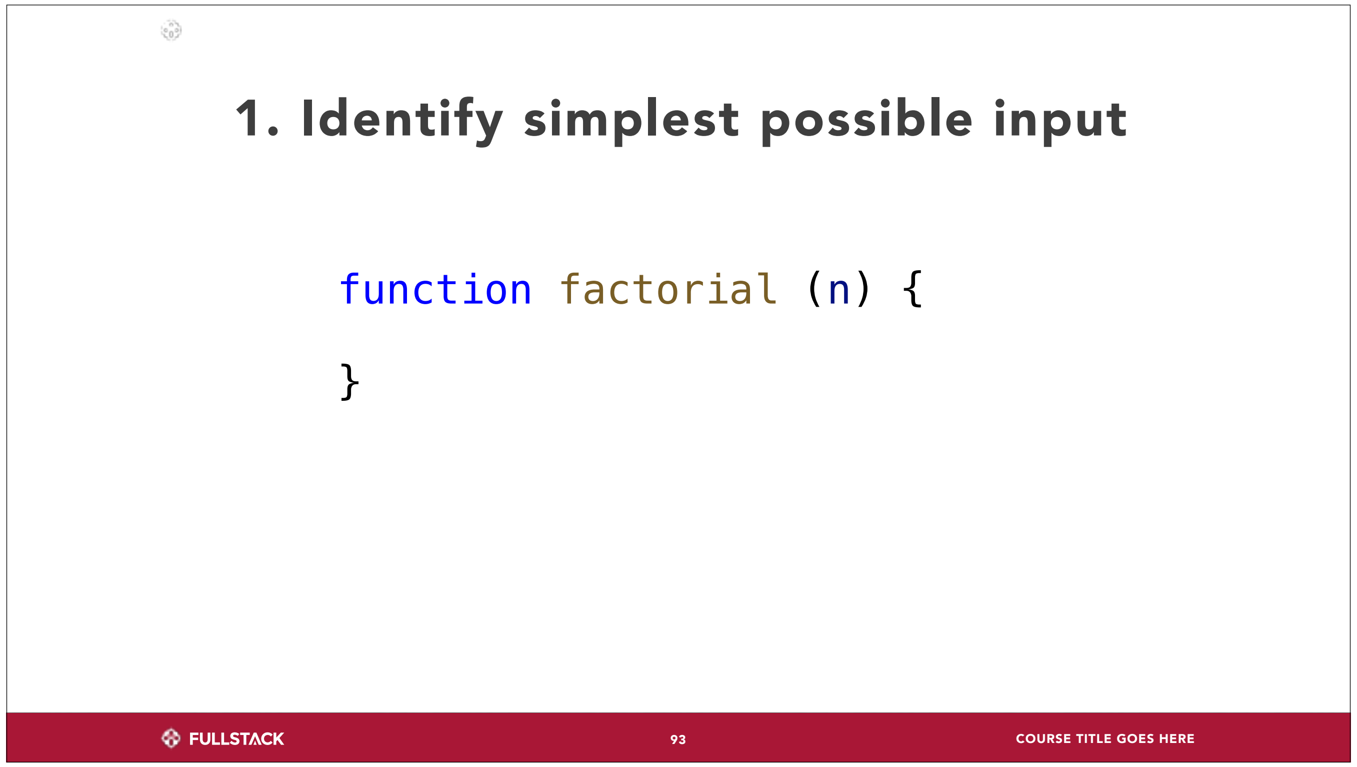
Recursive Problem Solving

- Any problem that can be solved iteratively can be solved recursively, and vice versa
- The catch: this doesn't mean it's equally easy to solve a particular problem both ways
- Binary Search Trees are recursive data structures - it's much easier to write them if you employ recursive problem solving

Any problem that can be solved iteratively (that is, using "for" or "while" loops) can also be solved recursively (and vice versa). However, this does not mean that the iterative solution for a given problem is as *easy* to implement as the recursive solution (and vice versa).

Because Binary Search Trees are a "recursive" data structure, it is *much* easier to implement their operations using recursion. While recursive code often looks simpler and easier to read than its iterative equivalent, it's often harder for beginners to visualize *why* it works, or to step through the logic.

To help you solve problems in a recursive manner, follow this methodology:



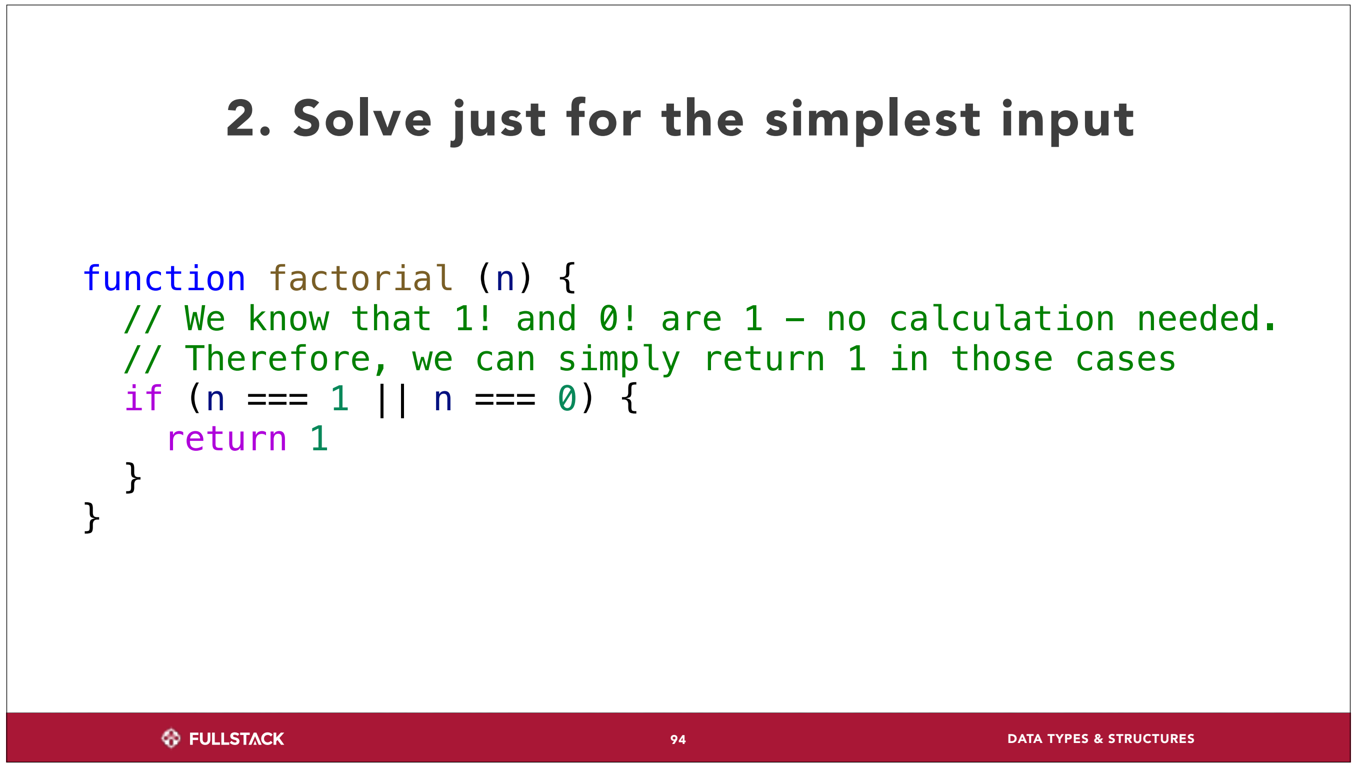
1. Identify simplest possible input

```javascript
function factorial (n) {

}
```

FULLSTACK

1. Identify the *simplest possible input* for the problem. For example, if the input to your function is a number, the simplest possible input might be `0`. If it's an array, it might be an empty array or an array with one element.

```javascript
function factorial (n) {
  // Our goal is to write a function that finds the factorial of a number 'n'
  // The factorial of a number is the product of all positive integers less than or
  // equal to n. For example, the factorial of 5 (in maths, notated 5!) is:

  // 5 * 4 * 3 * 2 * 1 === 120

  // Additionally, 0! is defined as 1

  // The "simplest input" to this function is 0 and/or 1
}
```

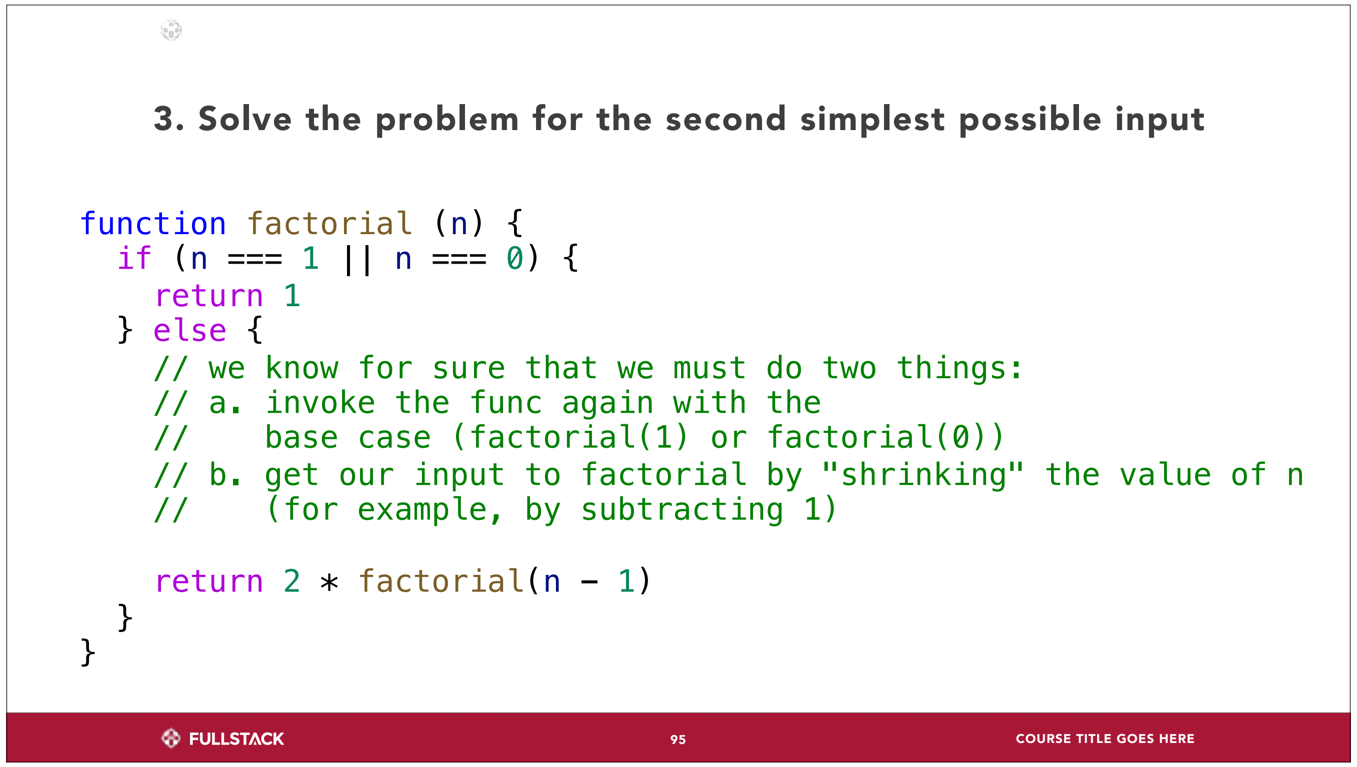


2. Solve just for the simplest input

```javascript
function factorial (n) {
  // We know that 1! and 0! are 1 - no calculation needed.
  // Therefore, we can simply return 1 in those cases
  if (n === 1 || n === 0) {
    return 1
  }
}
```

2. Solve the problem **specifically** for the "simplest possible input". This is known as the "base case".

```javascript
function factorial (n) {
  // We know that 1! and 0! are 1 - no calculation needed.
  // Therefore, we can simply return 1 in those cases
  if (n === 1 || n === 0) {
    return 1
  }
}
```



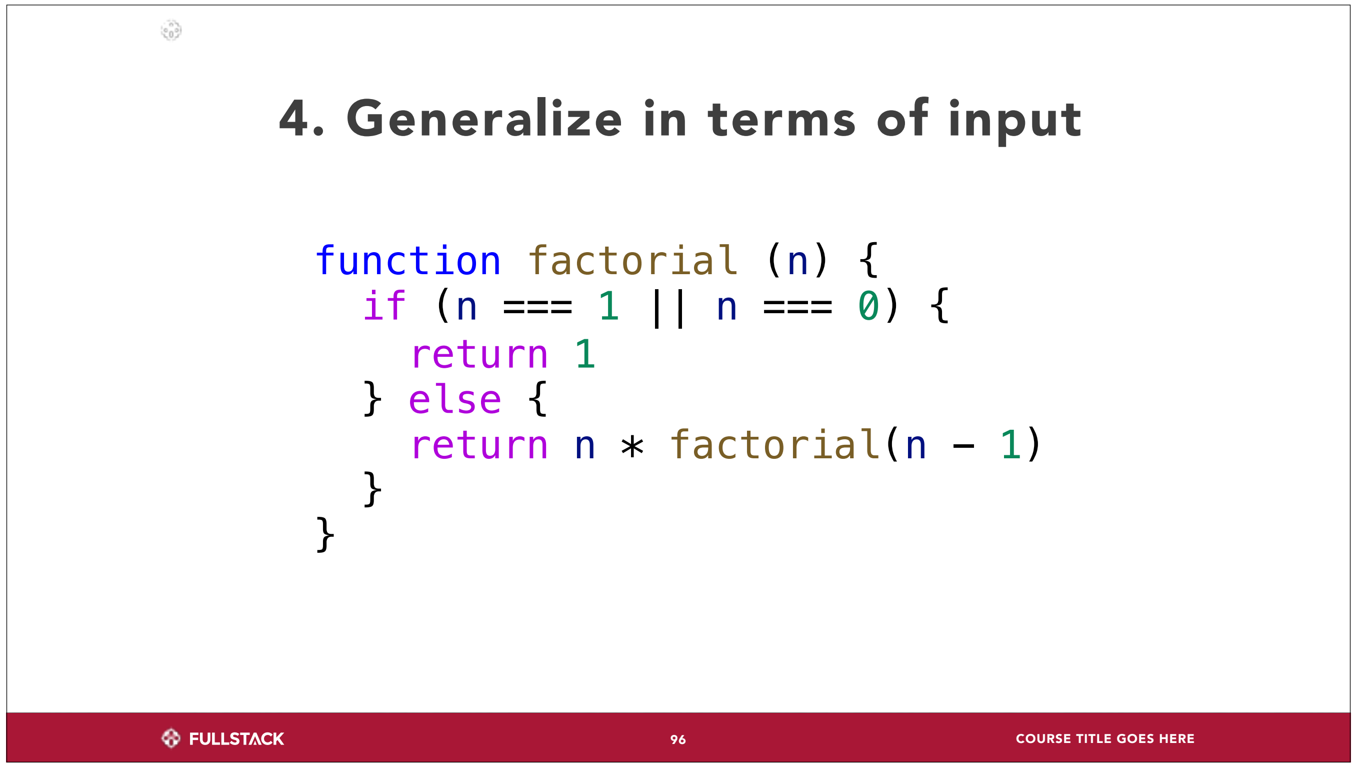
3. Solve the problem for the second simplest possible input

```
function factorial (n) {
  if (n === 1 || n === 0) {
    return 1
  } else {
    // we know for sure that we must do two things:
    // a. invoke the func again with the
    //    base case (factorial(1) or factorial(0))
    // b. get our input to factorial by "shrinking" the value of n
    //    (for example, by subtracting 1)

    return 2 * factorial(n - 1)
  }
}
```

3. Solve the problem **specifically** for the "second simplest possible input", making sure to
  a. **invoke the function recursively with the simplest possible input**.
  b. **shrink the "problem space"** when we recurse

This is the key to the trick. Beginners to recursion often get caught up trying to solve the problem for large values, which makes it difficult to visualize. However, if you work **very concretely** with the second simplest possible input, it will reduce the mental overhead, and more than likely solving the problem this way will also solve the problem for **any** input.

```javascript
// The "second simplest possible input for factorial is...2!"
// Let's imagine that n is 2
function factorial (n) {
  if (n === 1 || n === 0) { // this will be false...
    return 1
  } else { // so we write an else statement
    // in this else statement, we know for sure that we must do two things:
    // a. invoke factorial again with the base case (factorial(1) or factorial(0))
    // b. get our input to factorial by "shrinking" the value of n (for example, by subtracting 1)

    // Now, let's return to our concrete example: n is 2
    // We know that 2! should be 2 * 1 (which is 2)
    // We could just return two, but we also know we must do those other two tasks above.
    // We know that we can get 1 by subtracting n - 1
    // And we can see that if we invoke factorial(n - 1), that will give us 1,
    // so instead of saying 2 * 1, we could say 2 * factorial(n - 1).
    return 2 * factorial(n - 1)
  }
}
```



4. Generalize in terms of input

```javascript
function factorial (n) {
  if (n === 1 || n === 0) {
    return 1
  } else {
    return n * factorial(n - 1)
  }
}
```

4. Generalize anything concrete to be in terms of the input, and test with third simplest possible input and so on until satisifed.

```javascript
function factorial (n) {
  if (n === 1 || n === 0) {
    return 1
  } else {
    // Let's keep pretending that n is 2.
    // We were getting our answer by returning 2 * factorial(n - 1).
    // In the case where n is 2, we could then substitute the input n instead of the concrete value 2,
    // which gives us: n * factorial(n - 1).
    // This still solves the problem for 2, but we'll also discover that this
    // solves the problem for 3, 4, 5...and so on!
    return n * factorial(n - 1)
  }
}
```

Part 1 of the workshop!