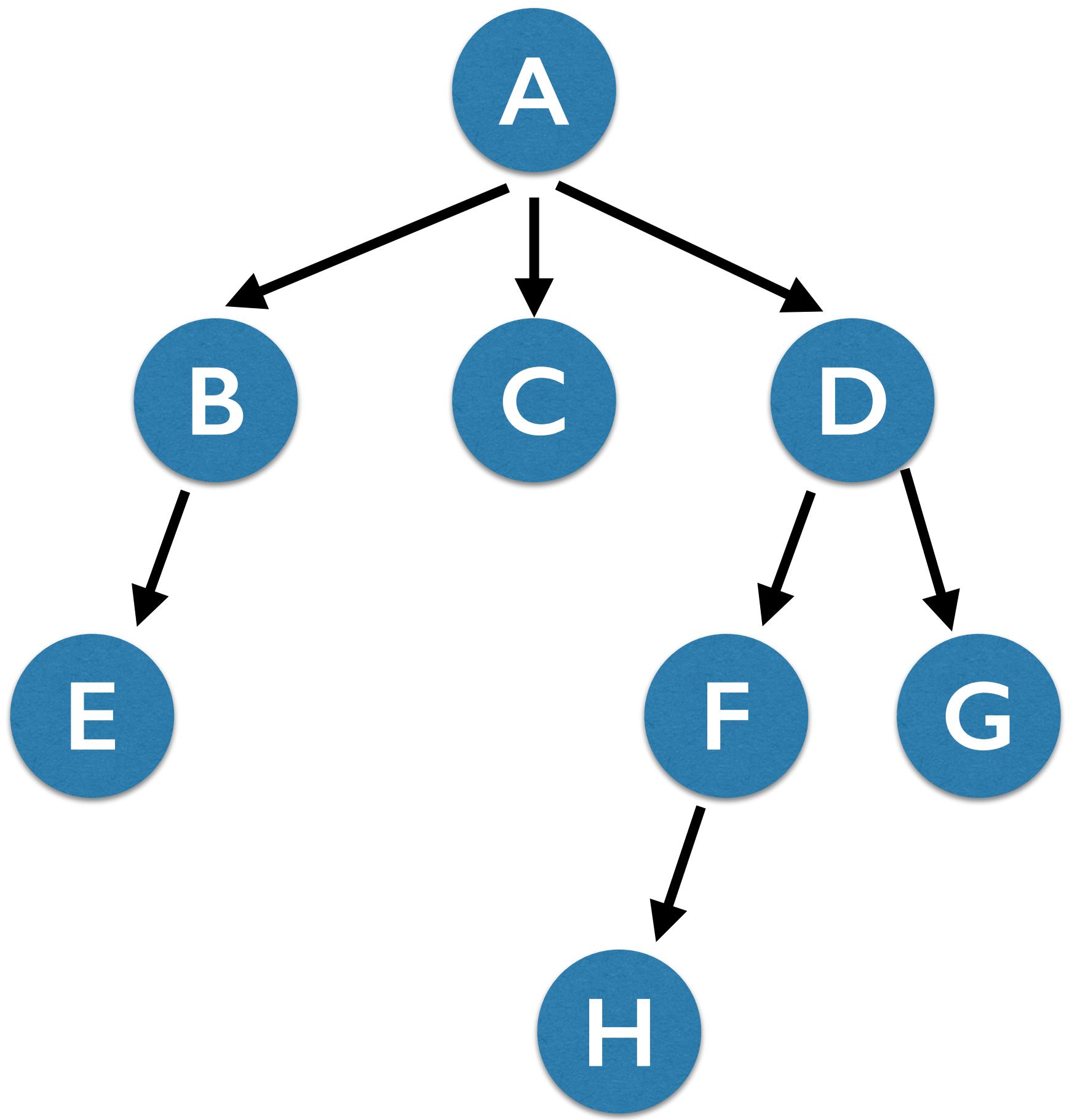




Trees

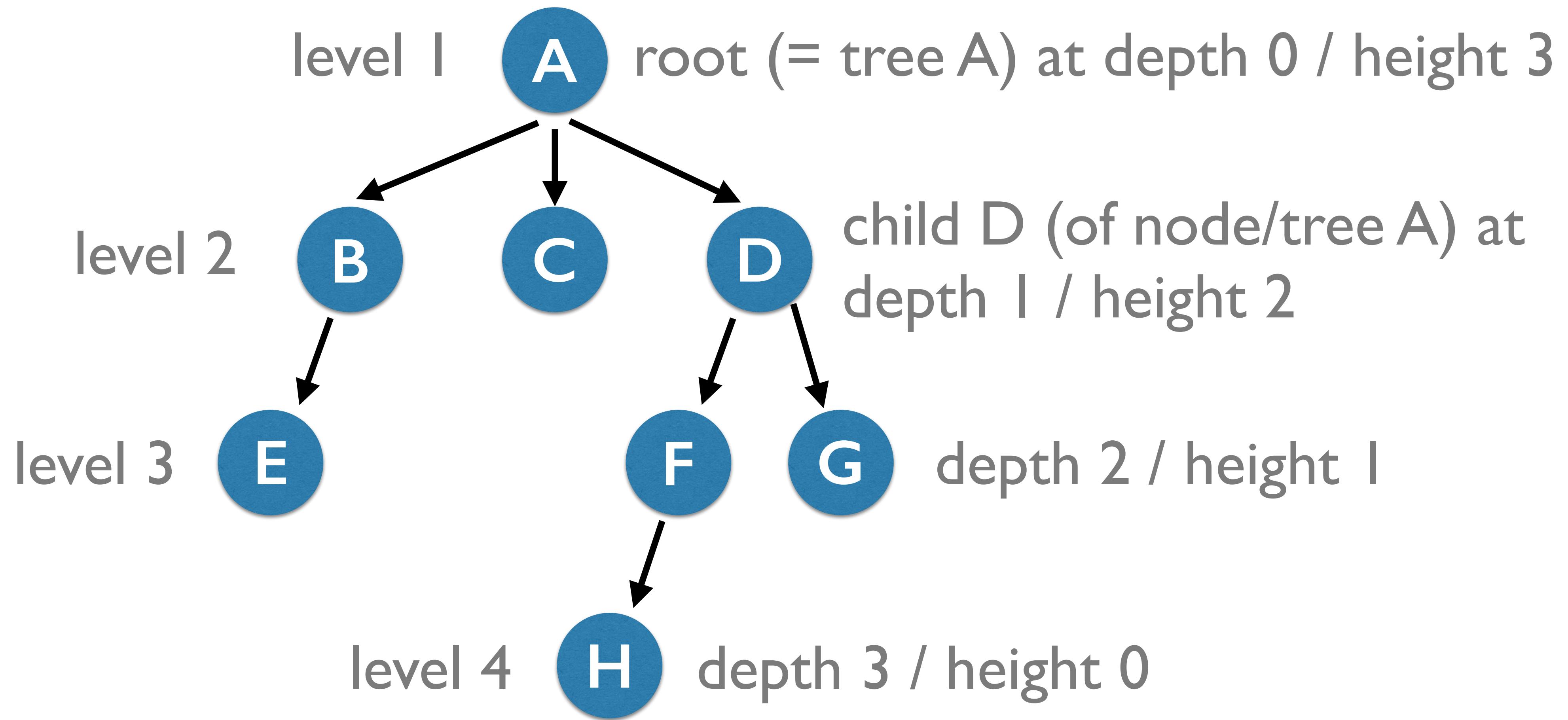
The Tree ADT



- 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

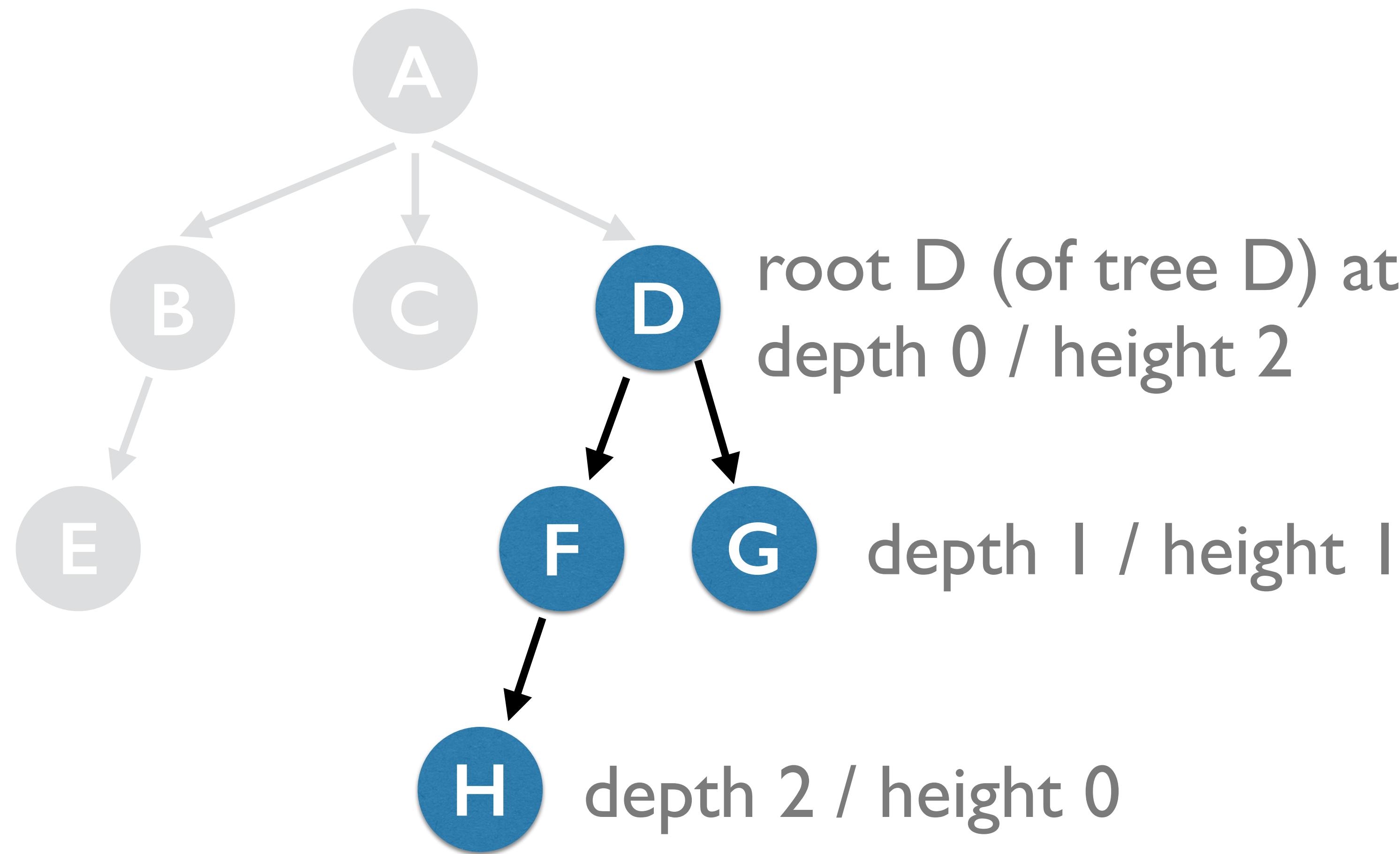


The Tree ADT





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



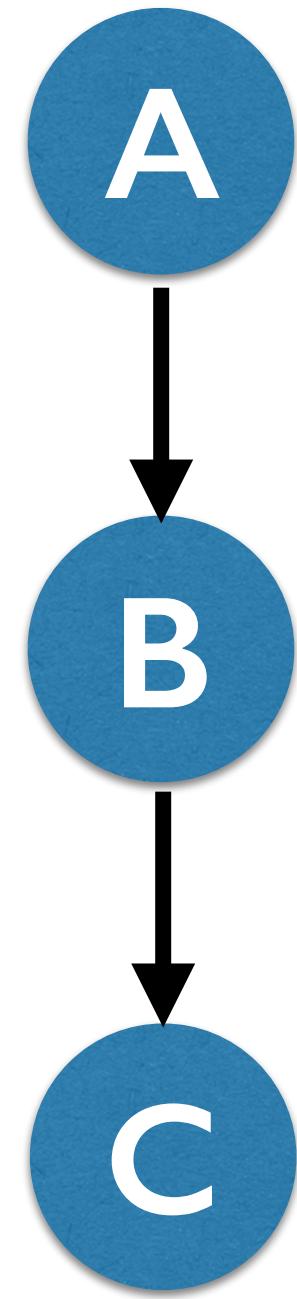


"Degenerate" trees are still trees

A tree of one node

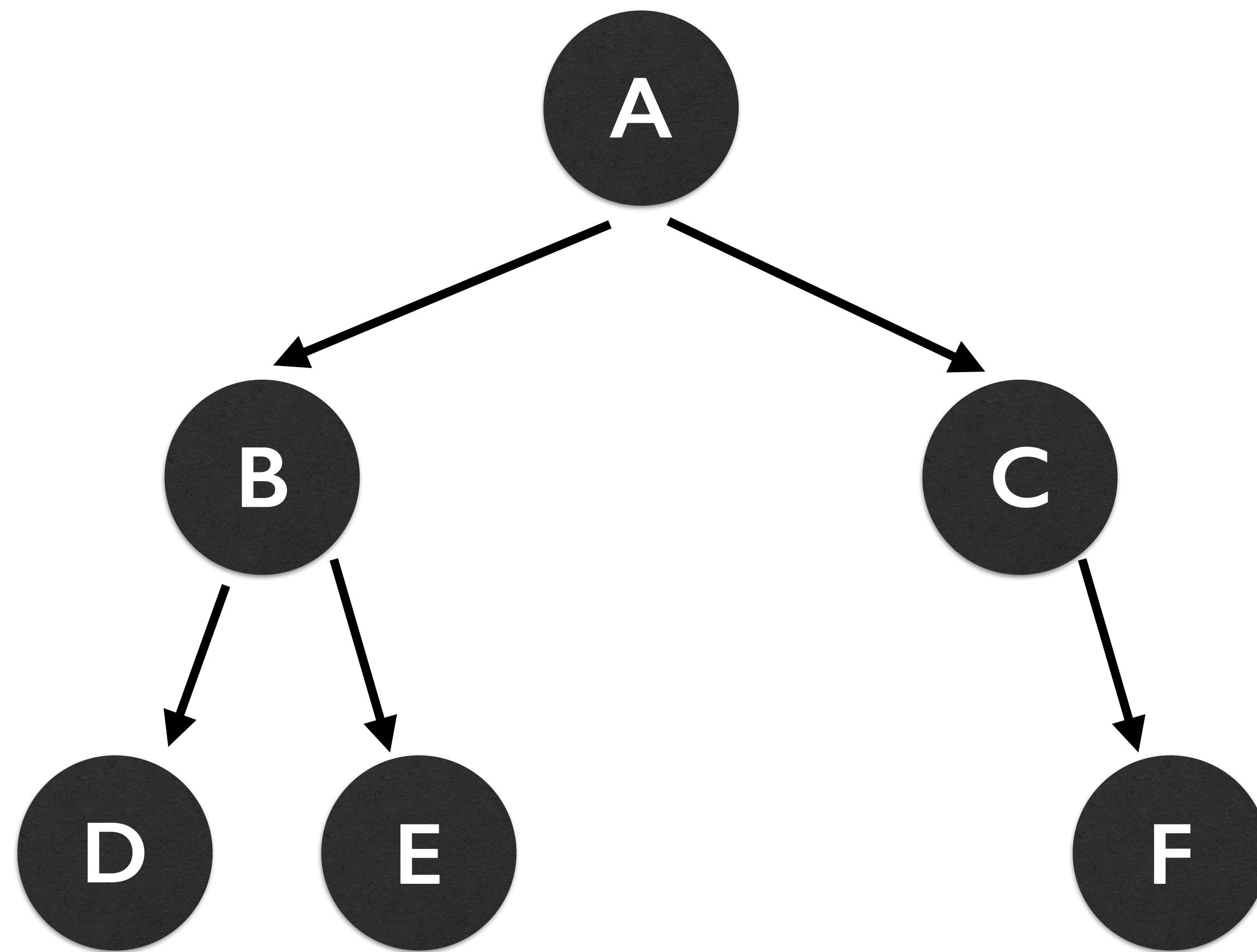


A tree of three nodes



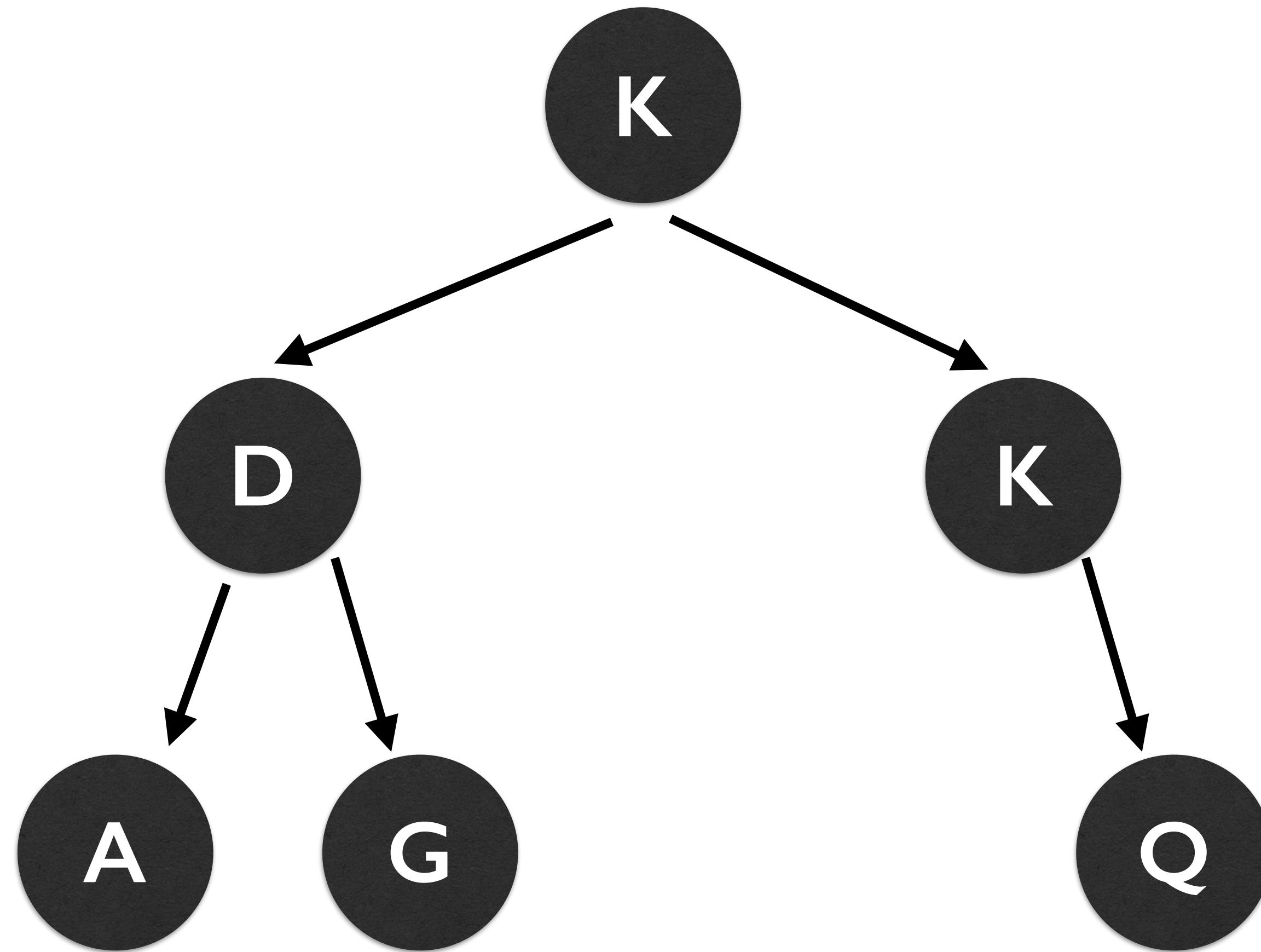


Binary Tree



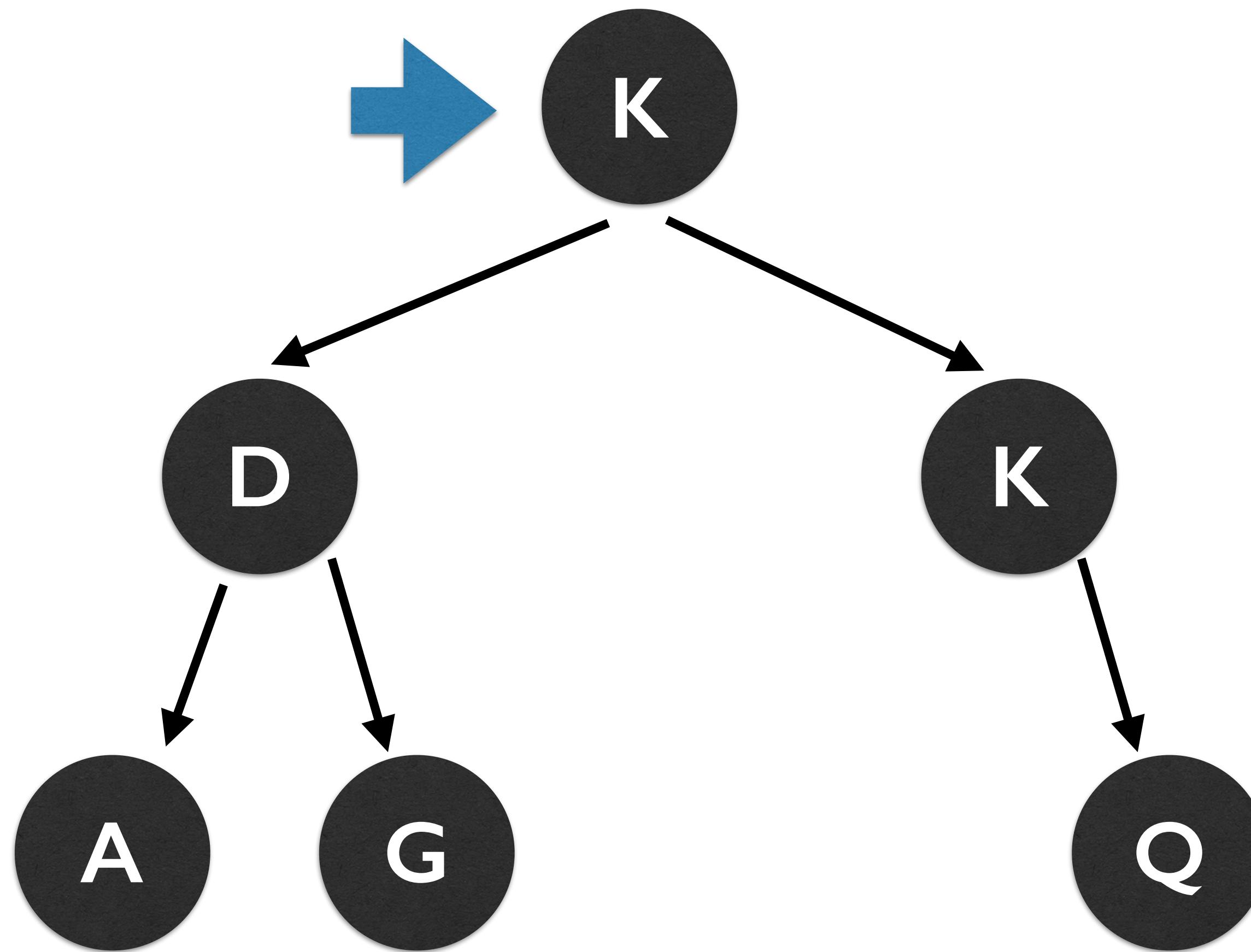


Binary Search Tree



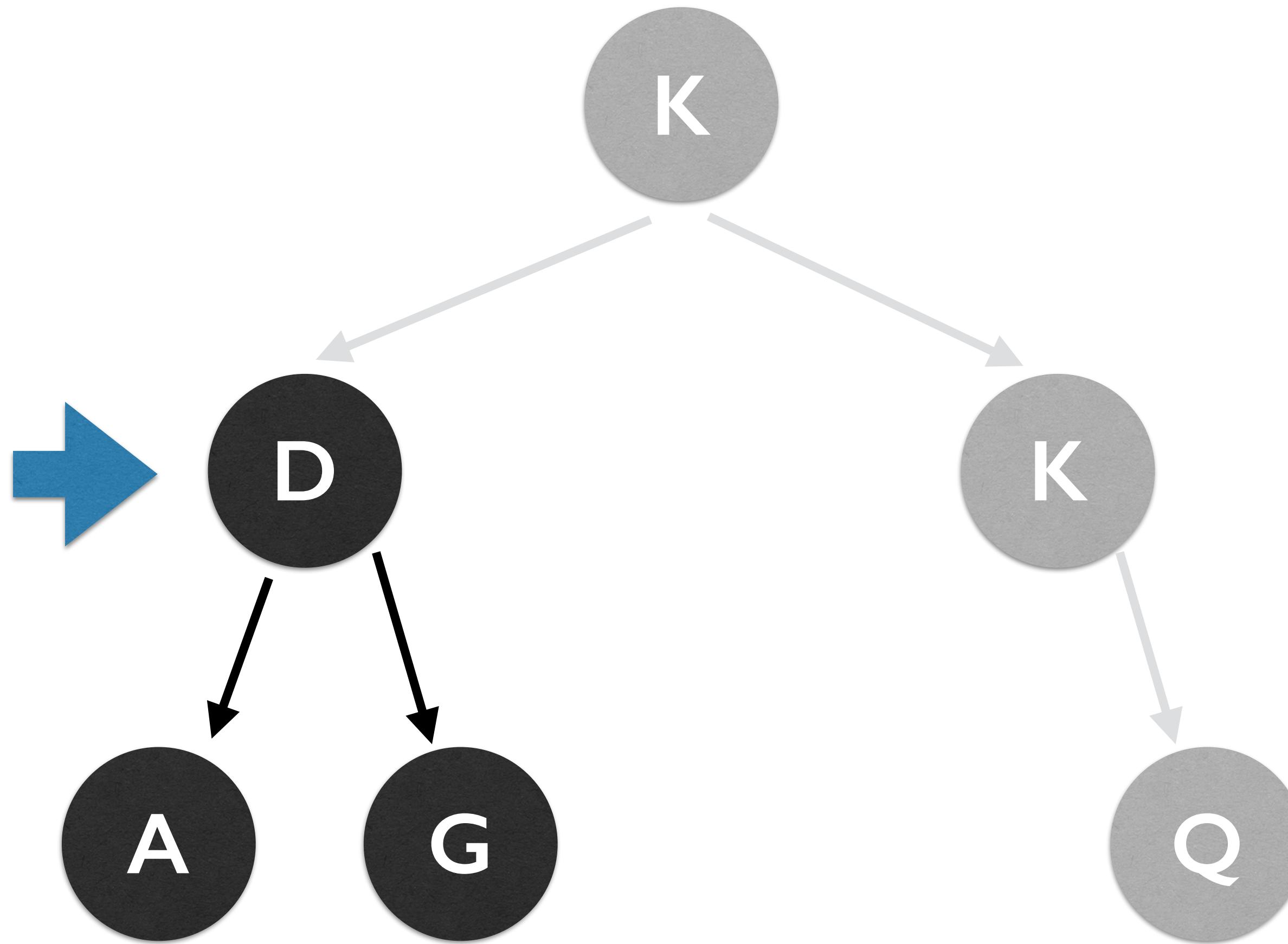


Binary Search Tree: Min Value?



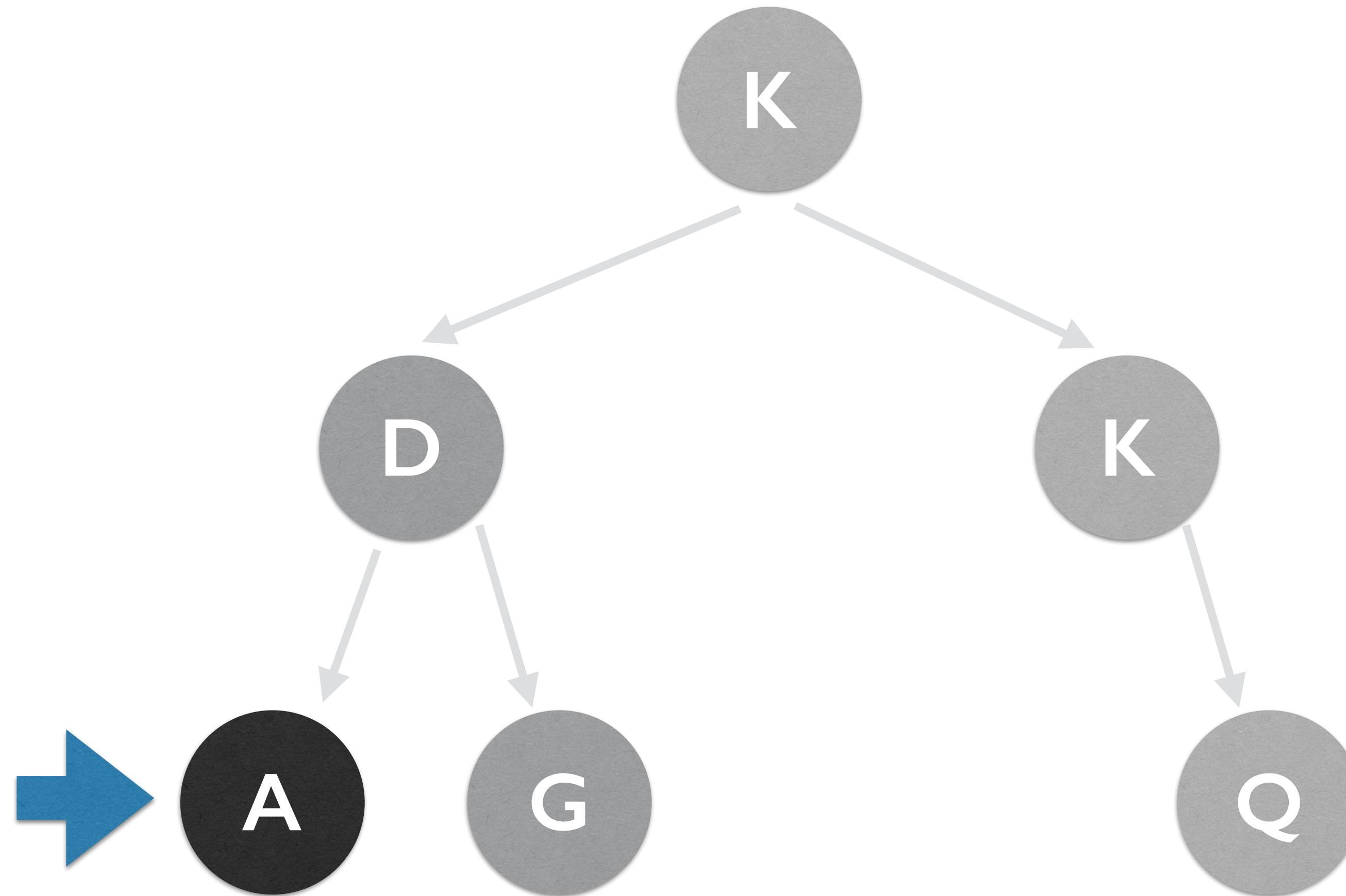


Binary Search Tree: Min Value?



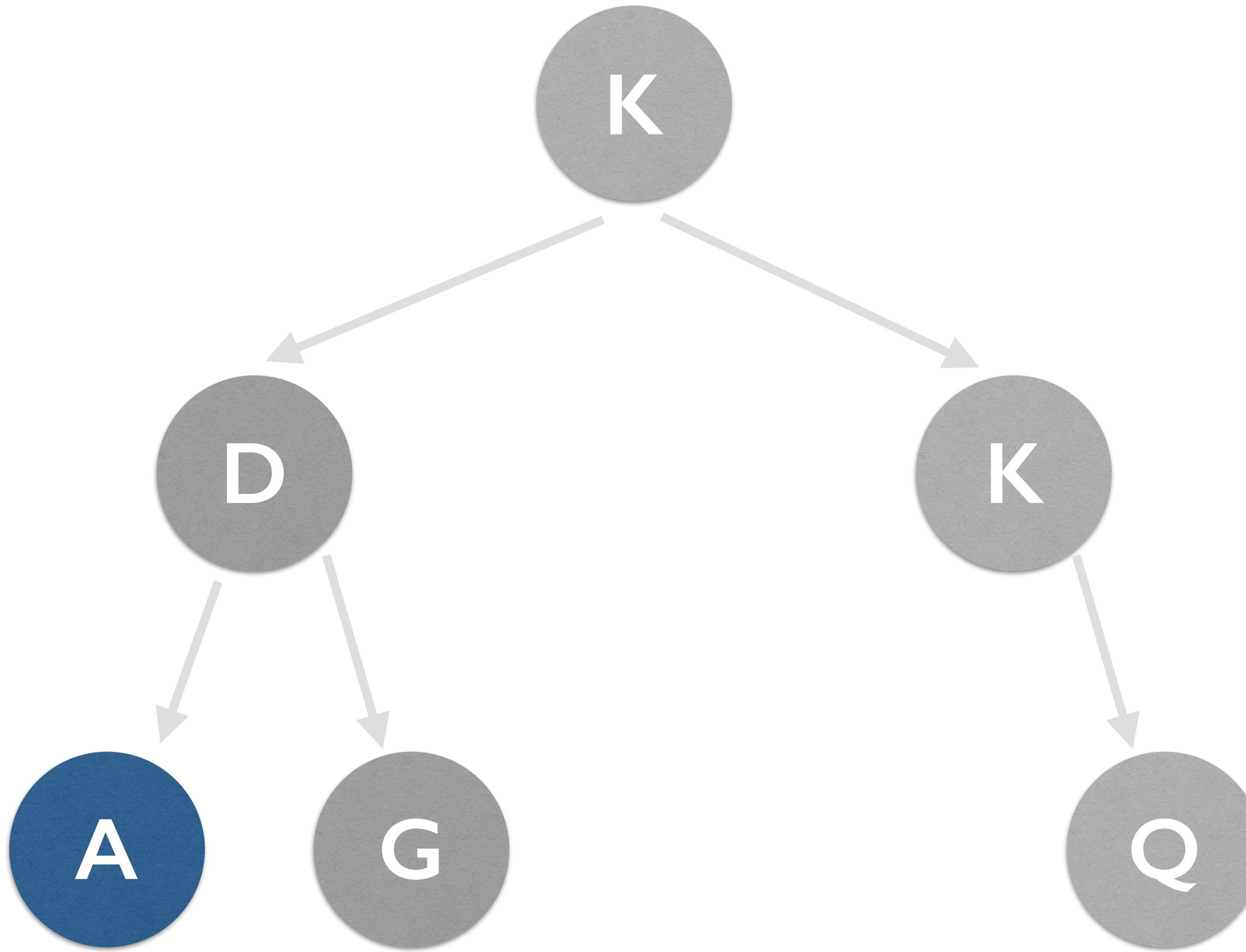


Binary Search Tree: Min Value?



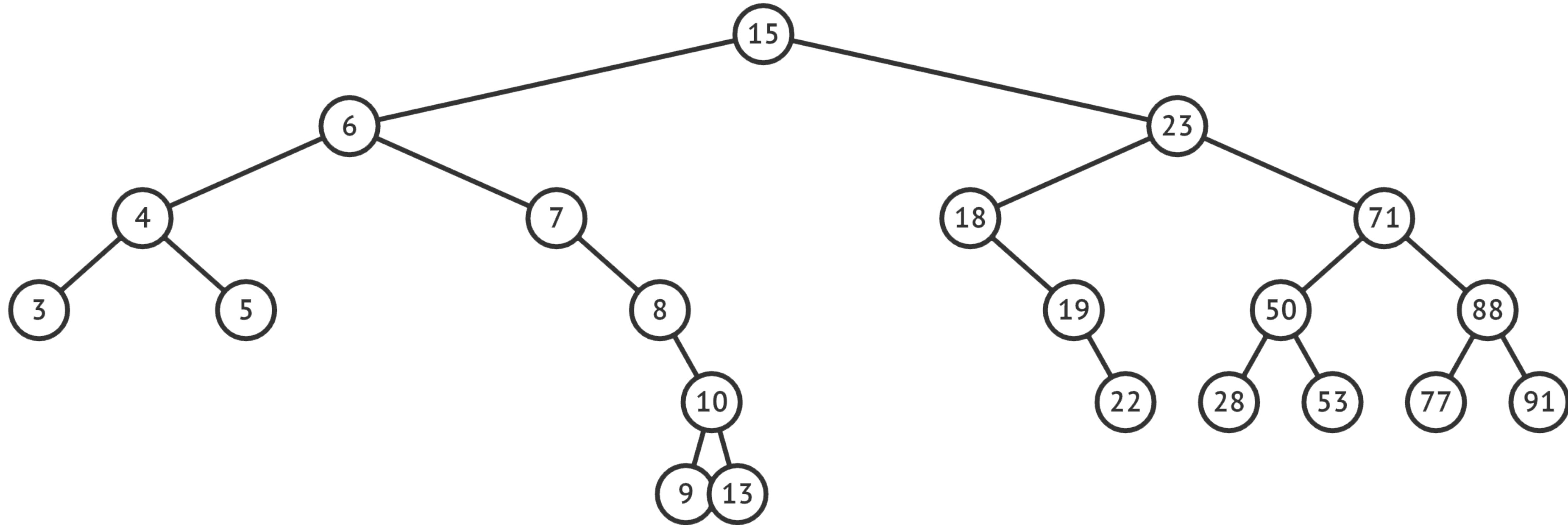


Binary Search Tree: Min Value?

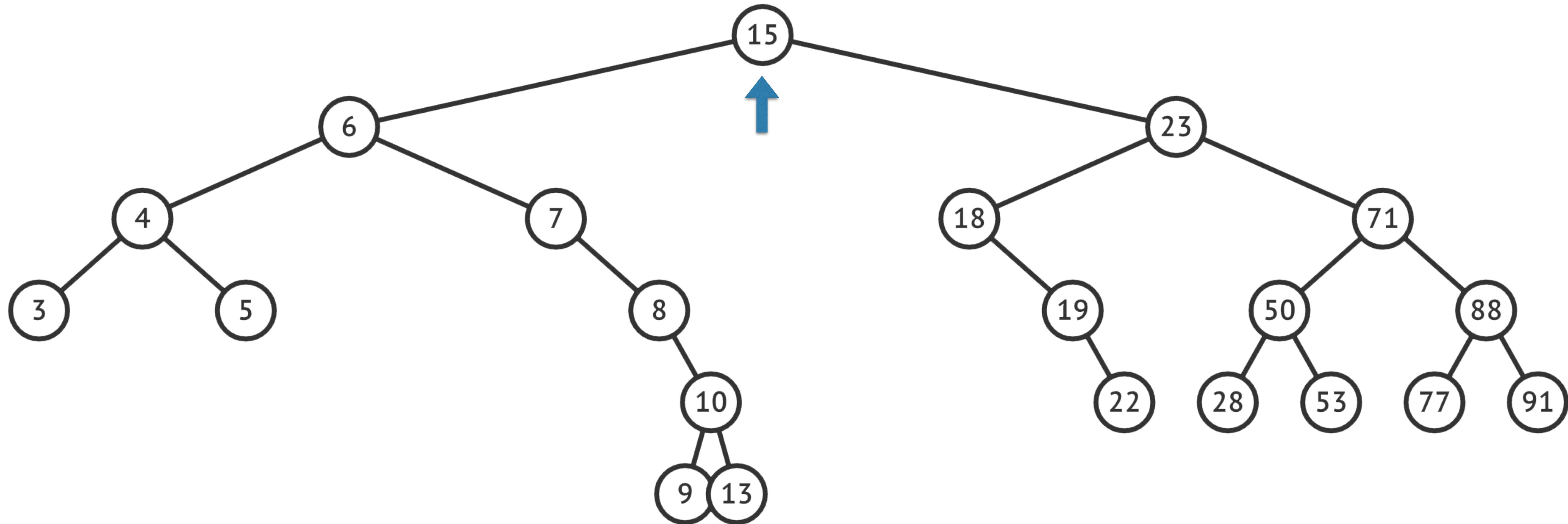




Does this BST contain the value 28?

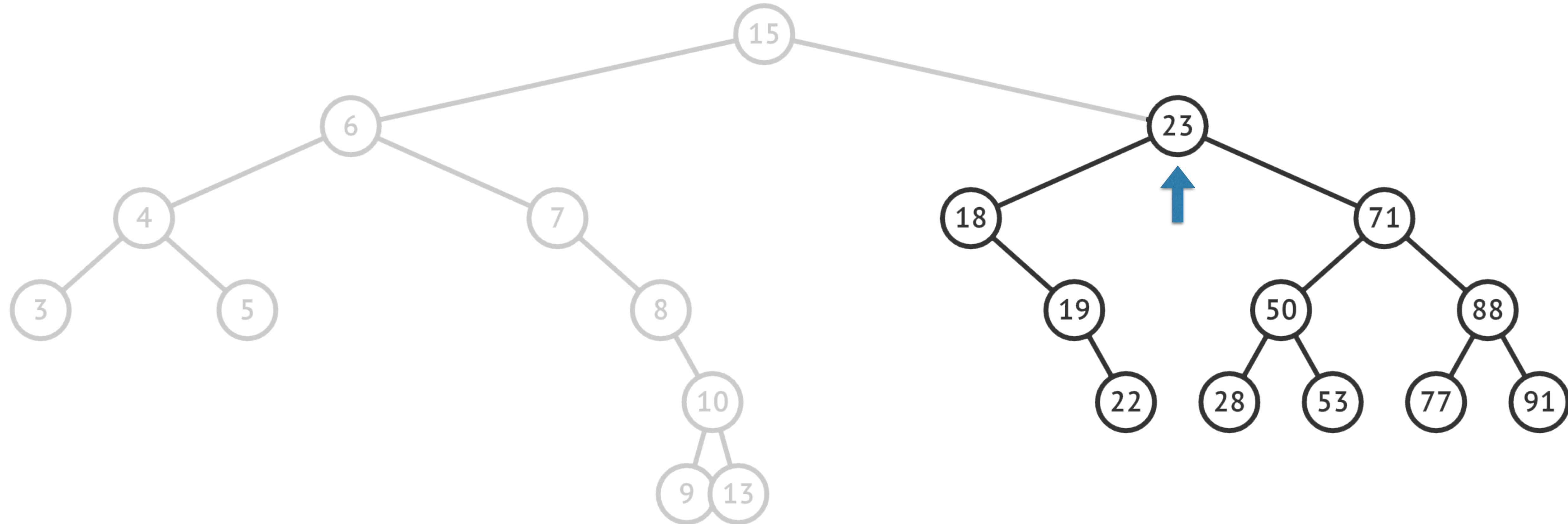


Does this BST contain the value 28?



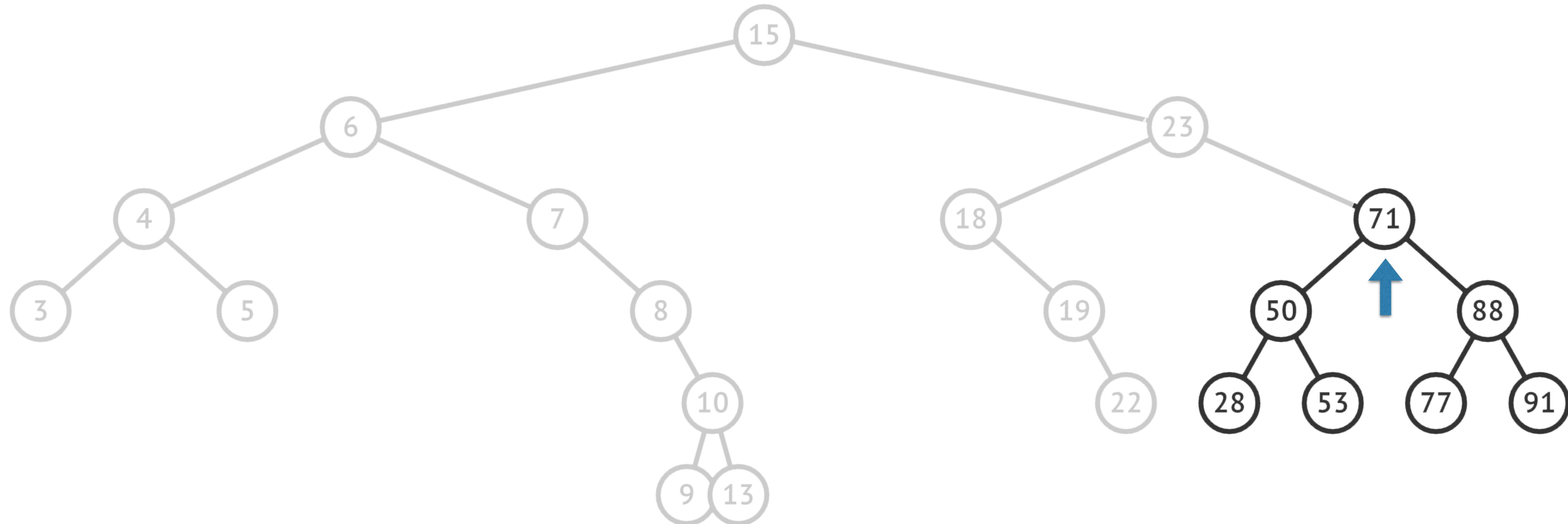


Does this BST contain the value 28?



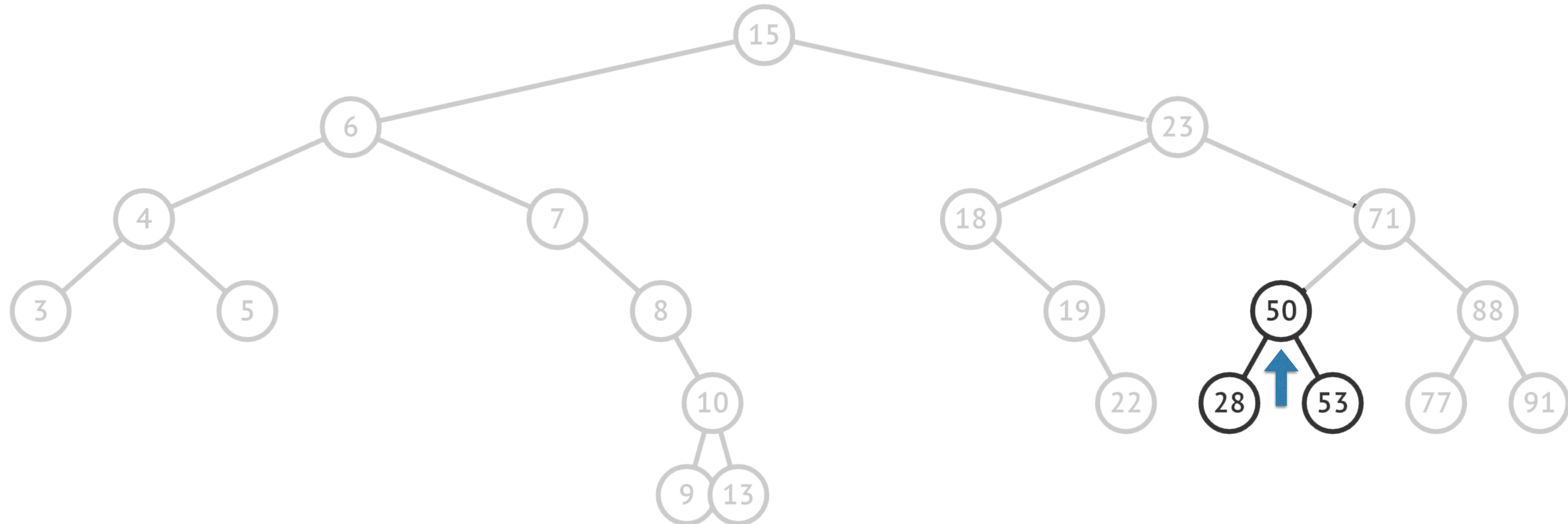


Does this BST contain the value 28?



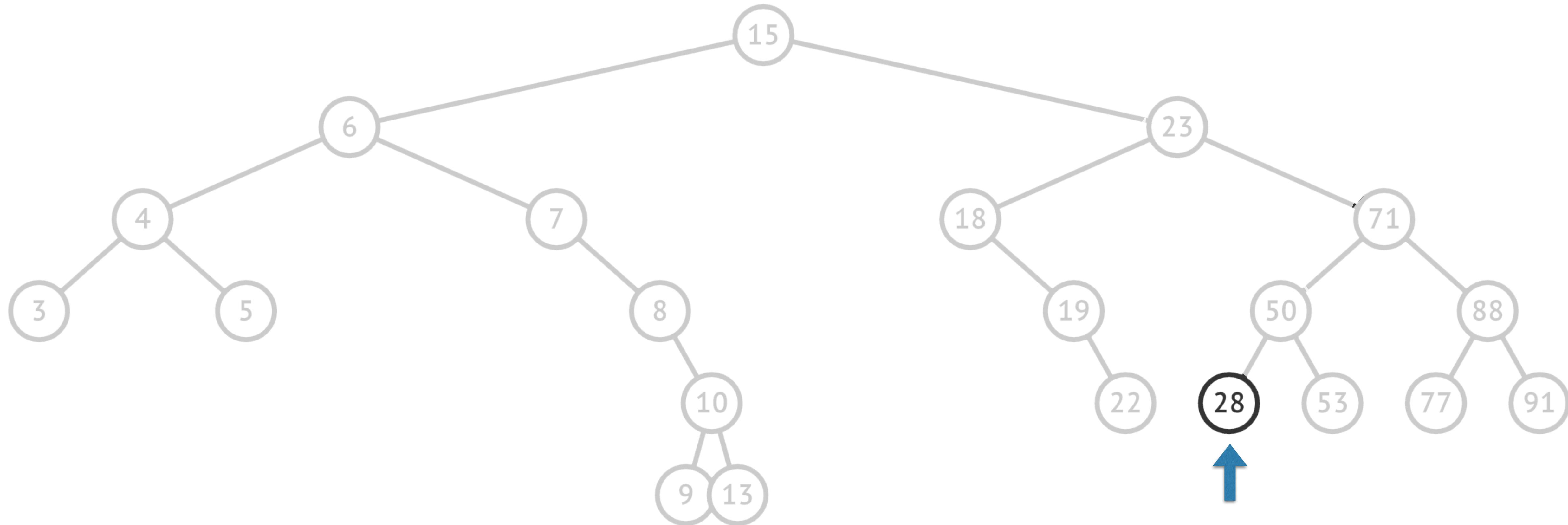


Does this BST contain the value 28?





Does this BST contain the value 28?



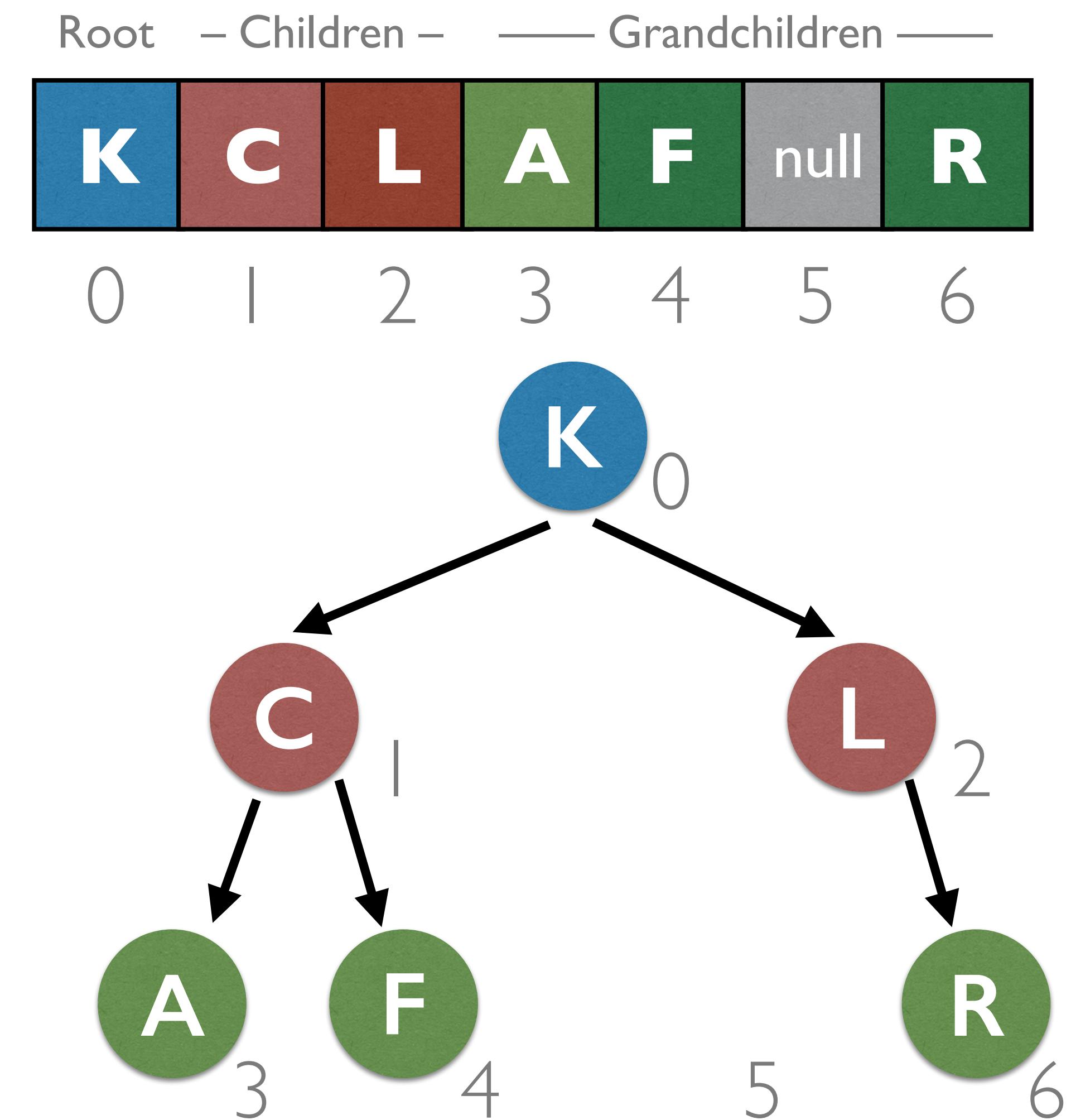
BST ADT

- Root node satisfies ordering principle
 - Left descendants < root value \leq 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?

...Maybe an array (seriously)?

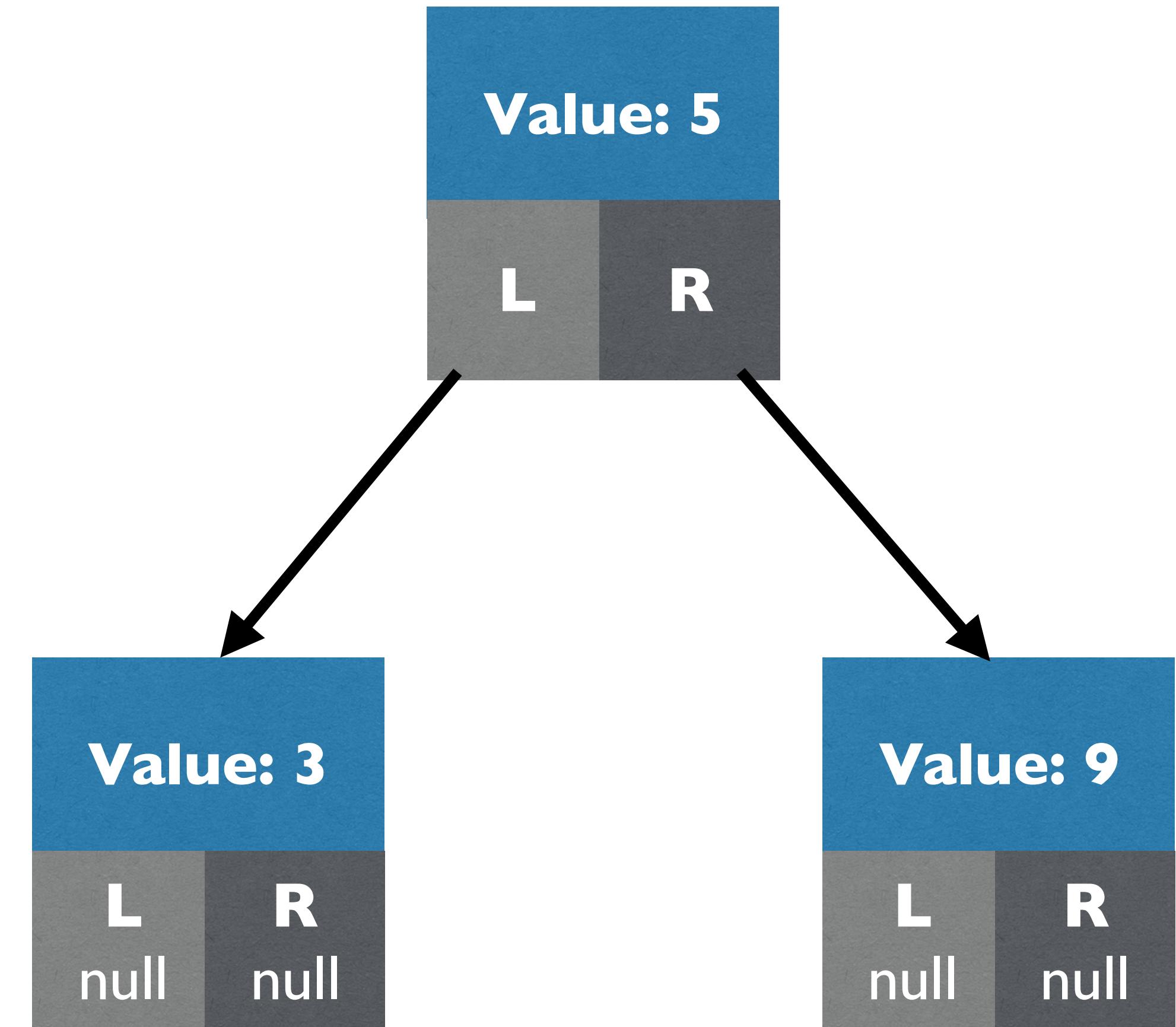
- The Tree ADT, with its talk of "nodes" and "references," seems so obviously to describe a data *structure* that it is perhaps confusing to tell the two apart.
- In fact, a tree can be stored in a few different ways. For example, if you knew your tree nodes always had at most two children, you could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



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.





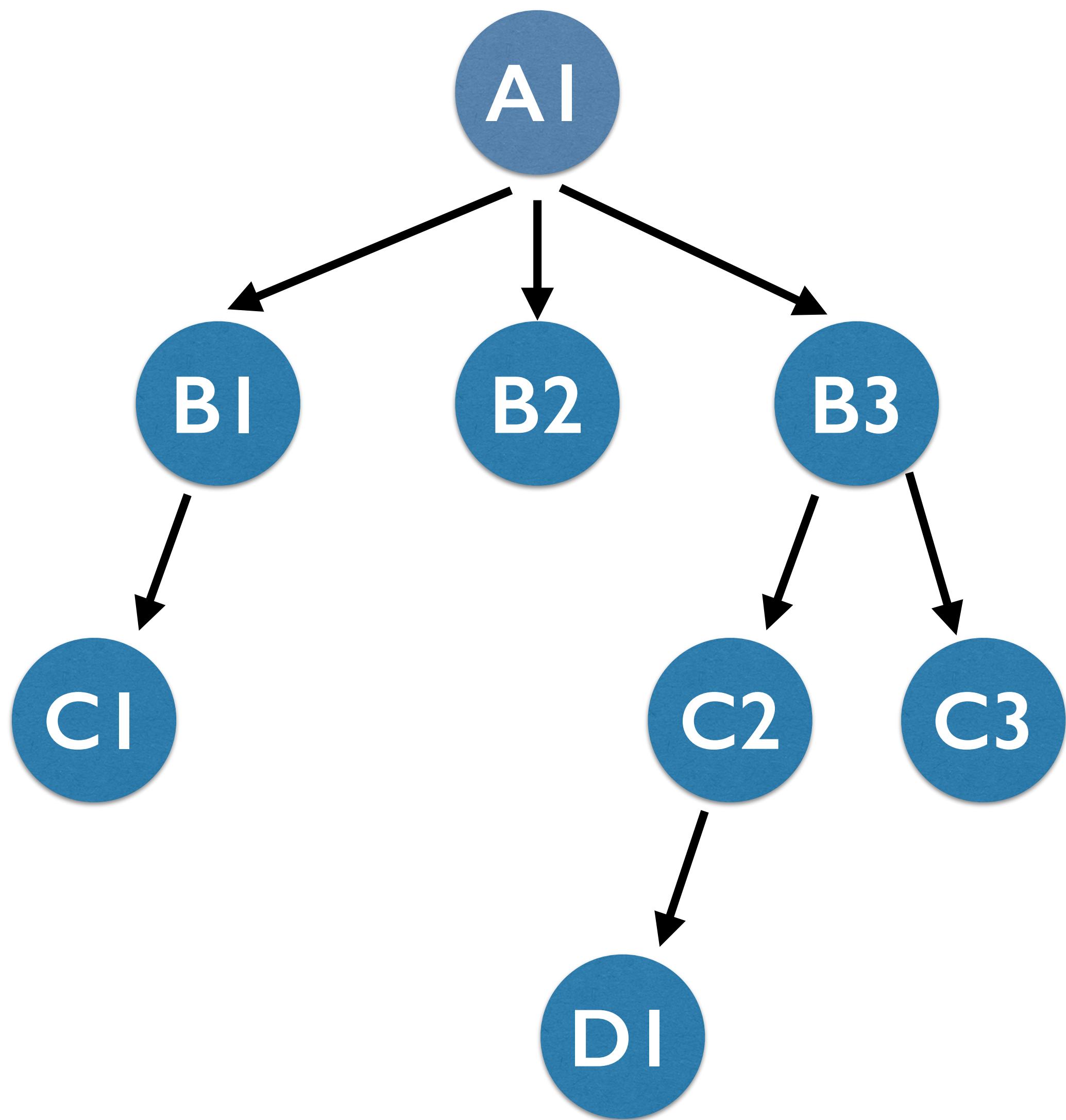
Tree traversal

Traversal: visiting every node

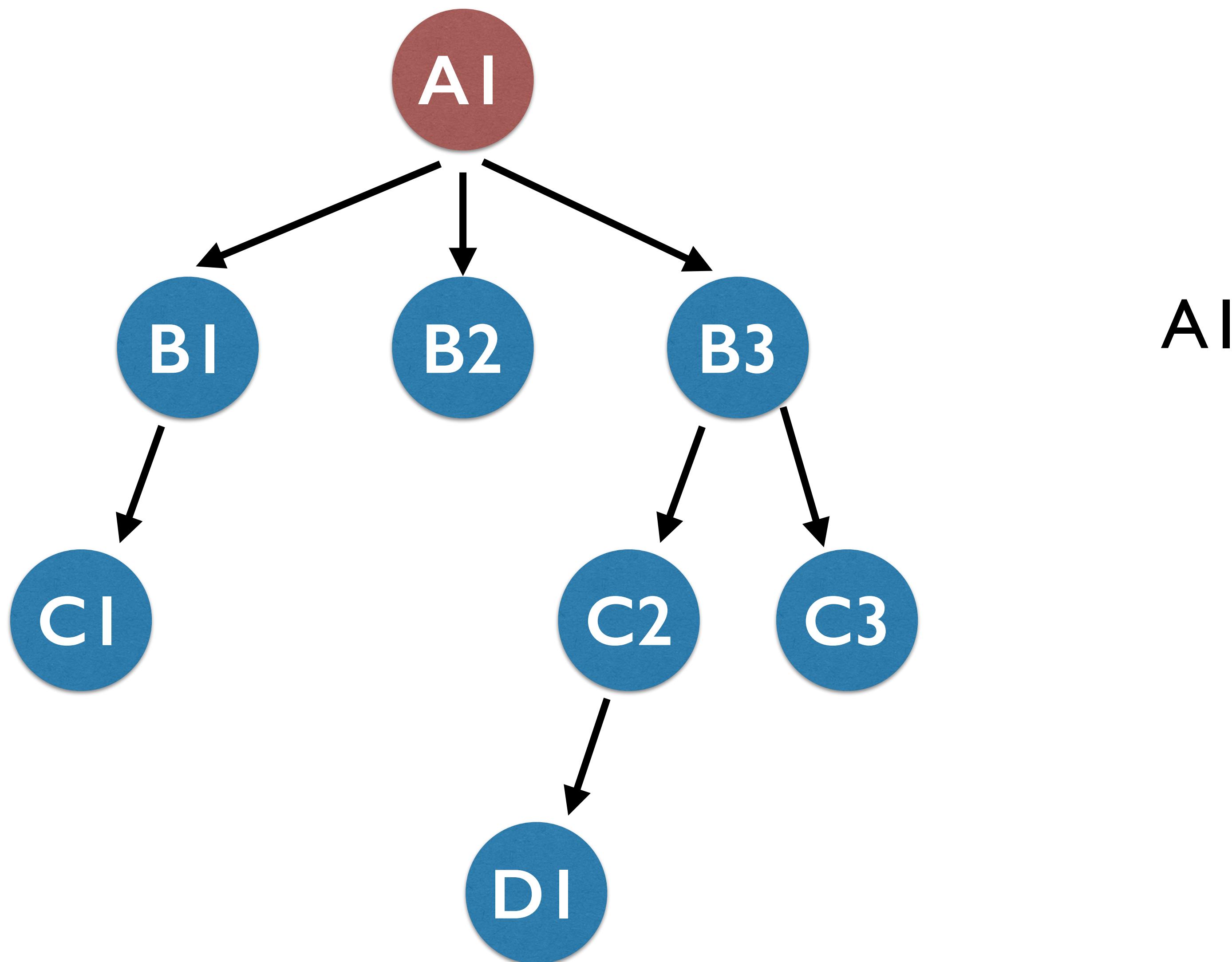
- Breadth-first (level by level)
- Depth-first (branch by branch)
 - Pre-order: process **root node**, process **left subtree**, process **right subtree**
 - In-order: process **left subtree**, process **root node**, process **right subtree**
 - Post-order: process **left subtree**, process **right subtree**,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

Breadth-First

Breadth First Traversal

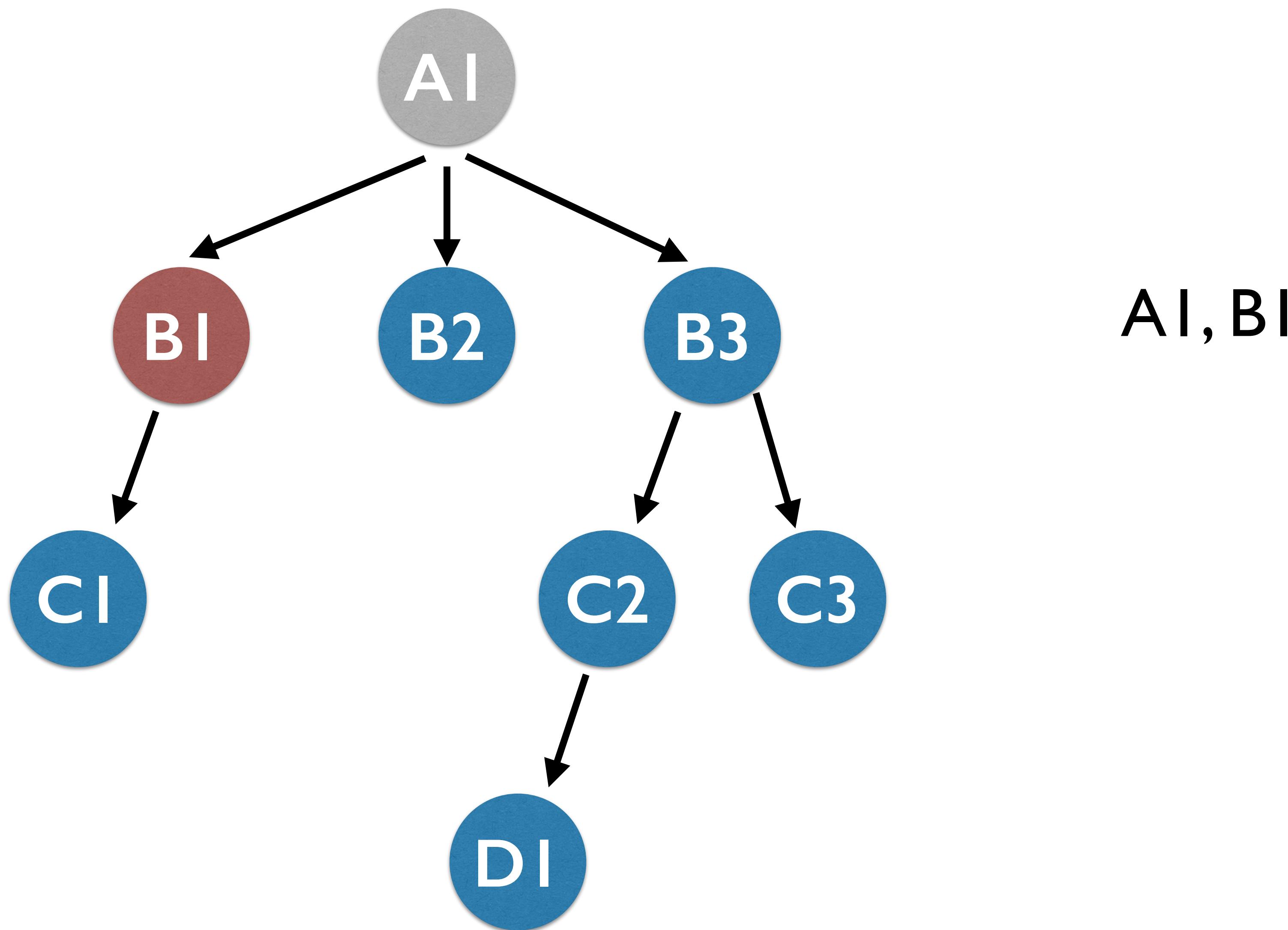


Breadth First Traversal



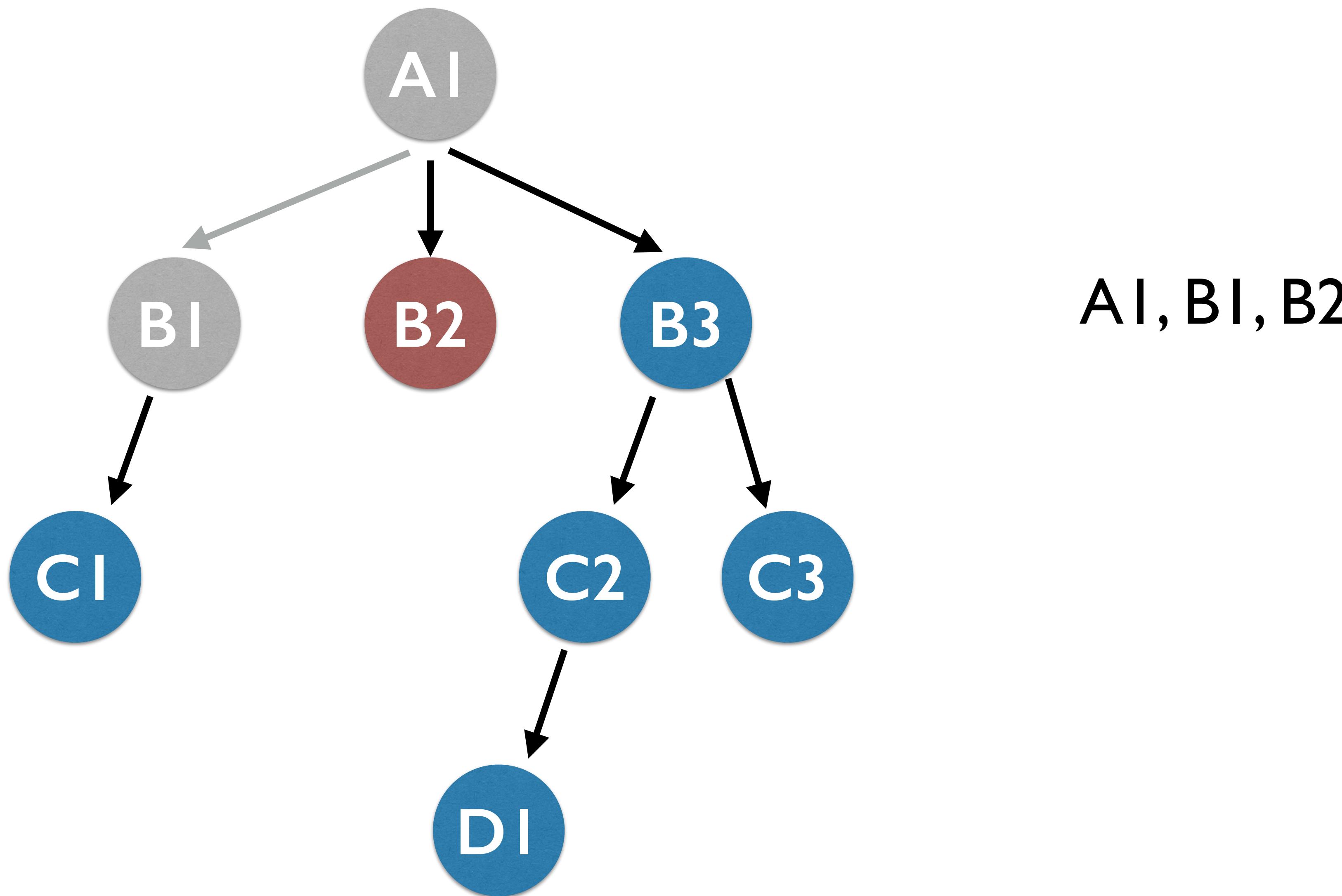
AI

Breadth First Traversal

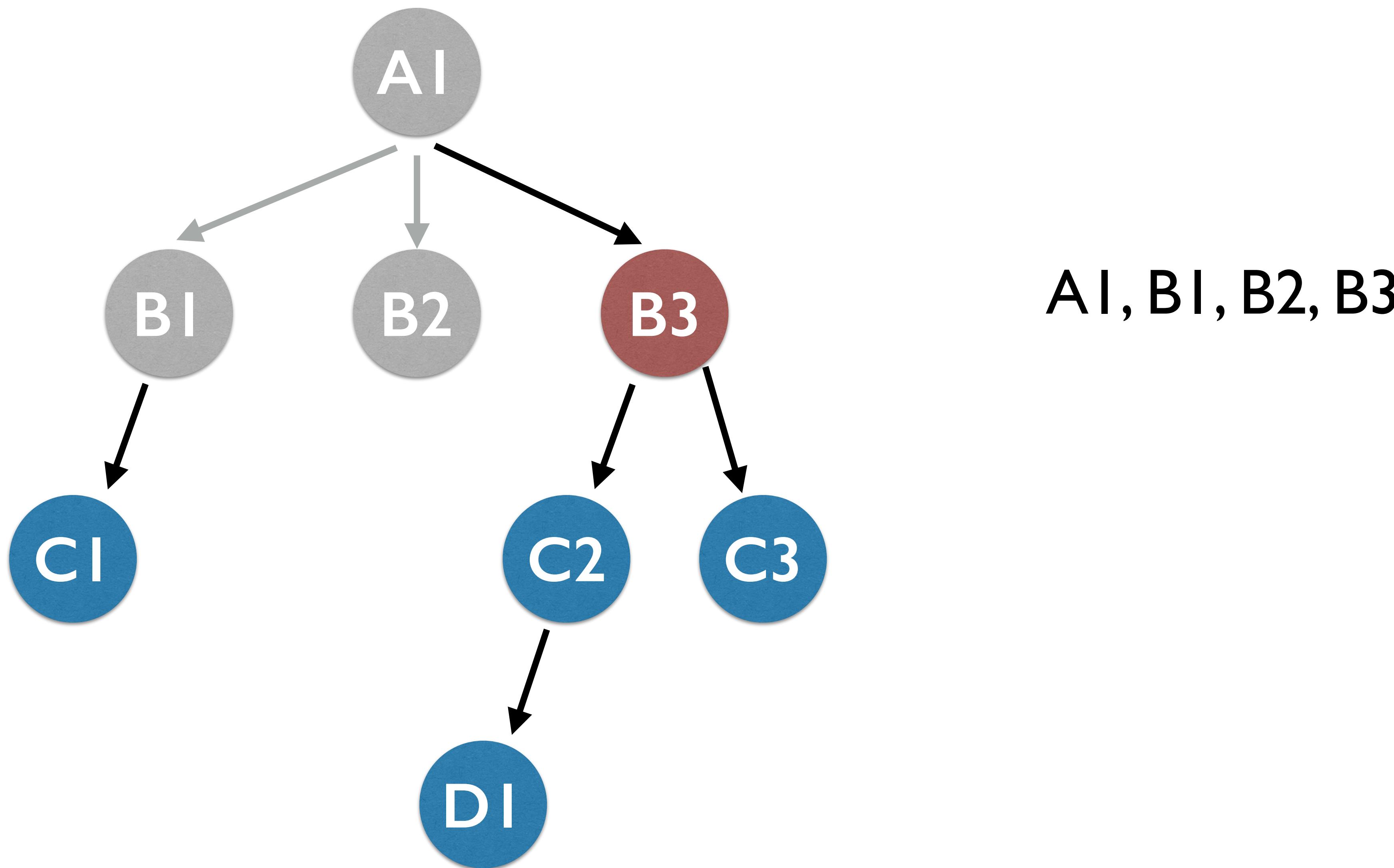


A1, B1

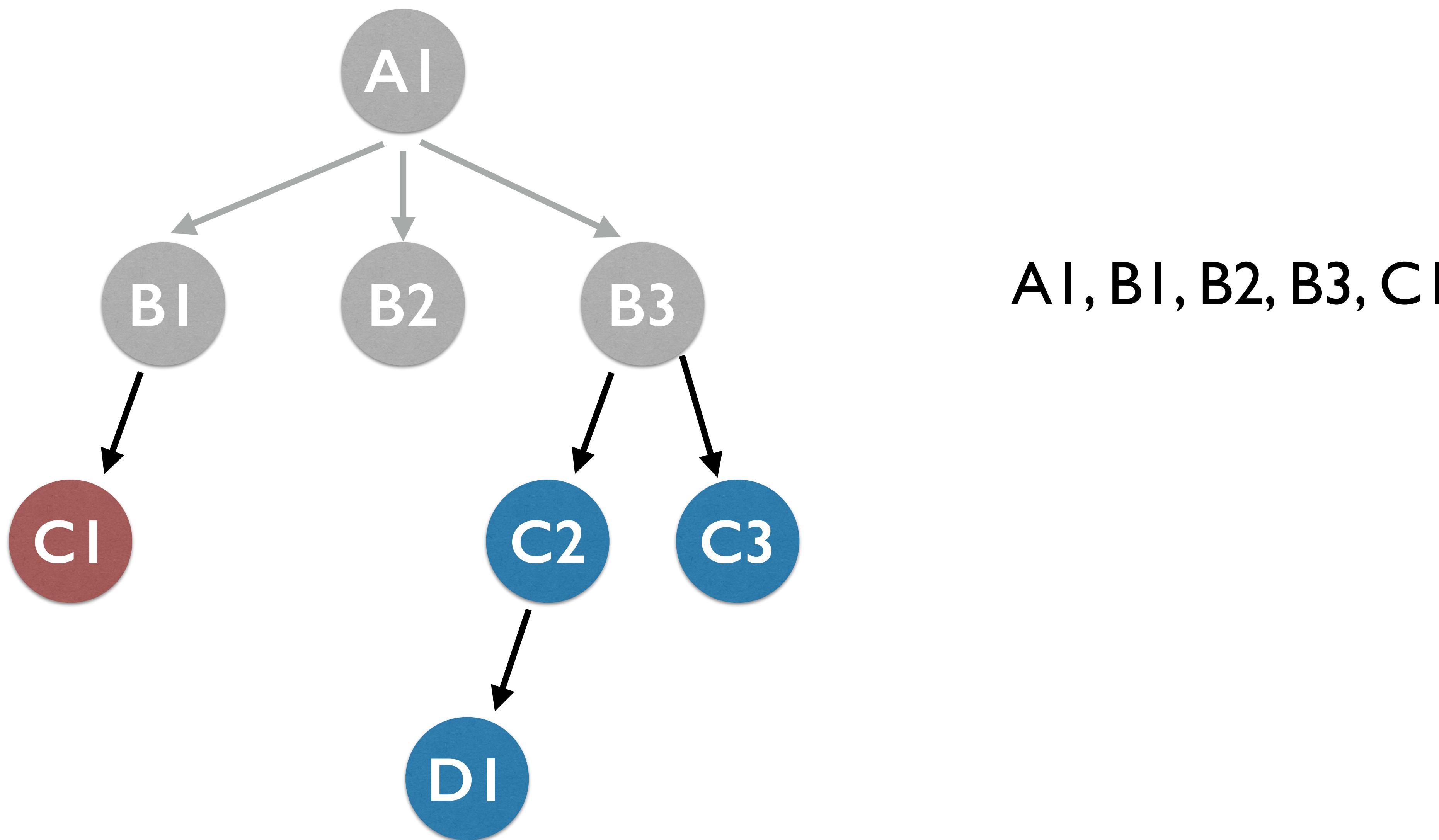
Breadth First Traversal



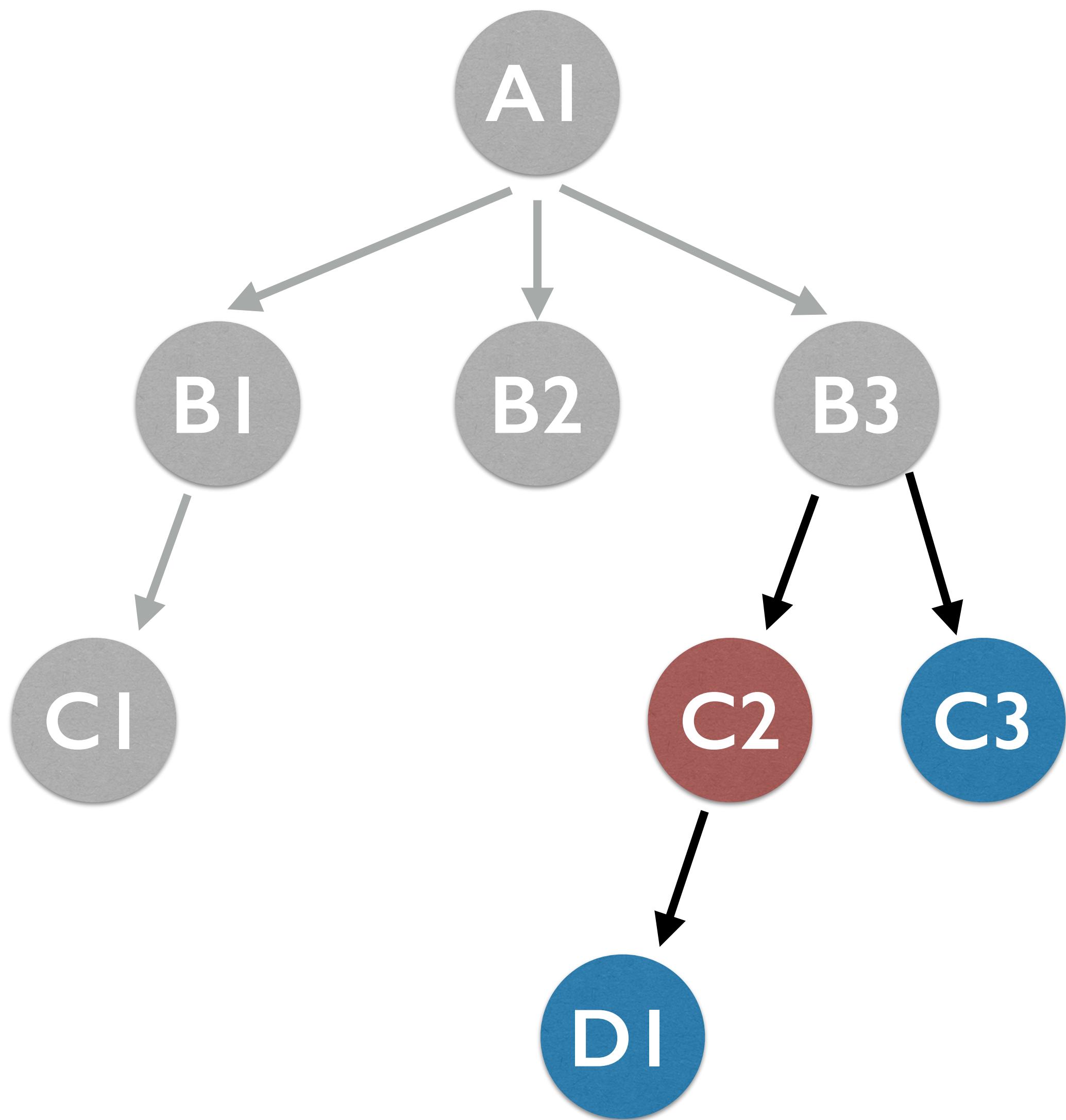
Breadth First Traversal



Breadth First Traversal

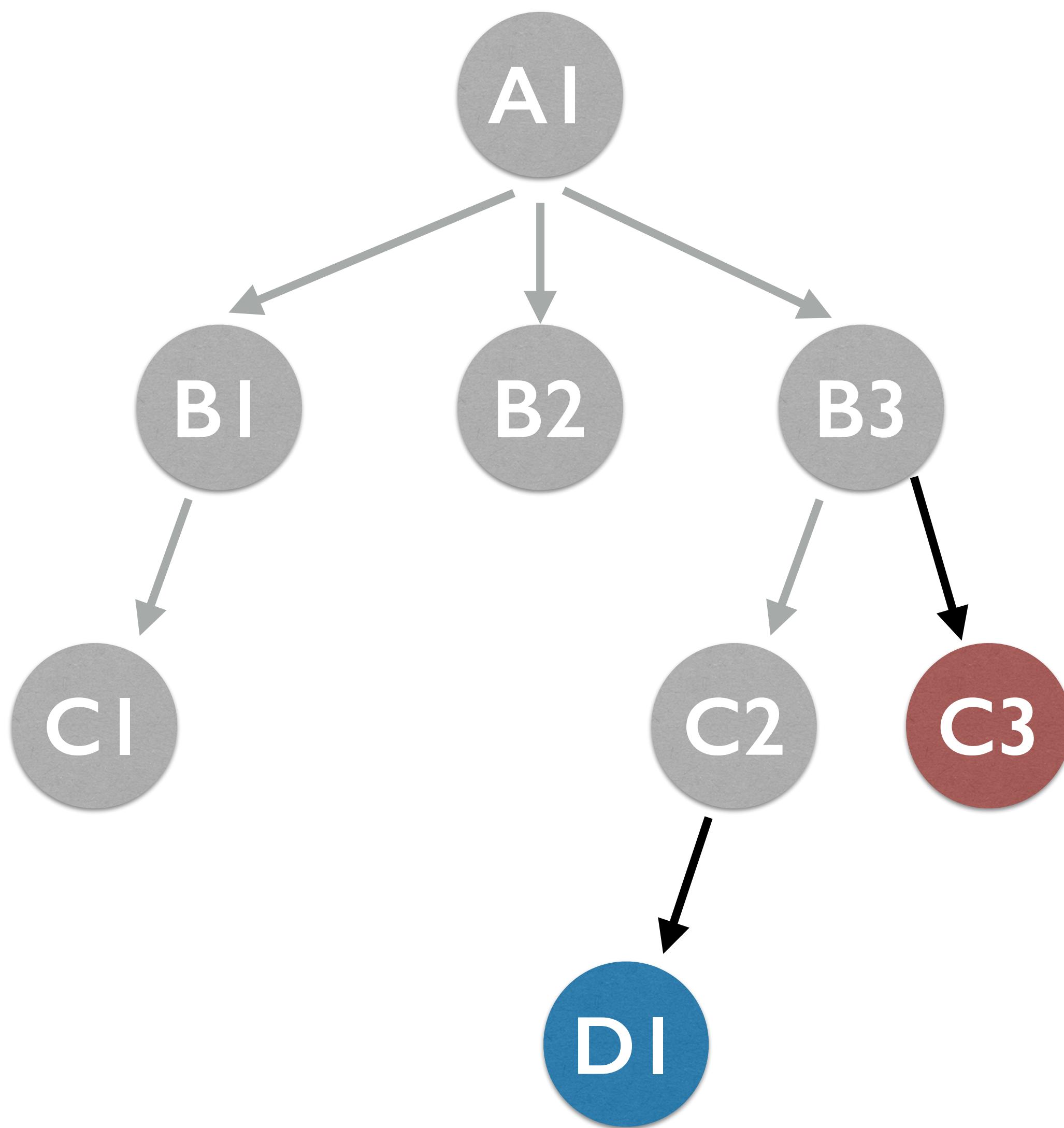


Breadth First Traversal



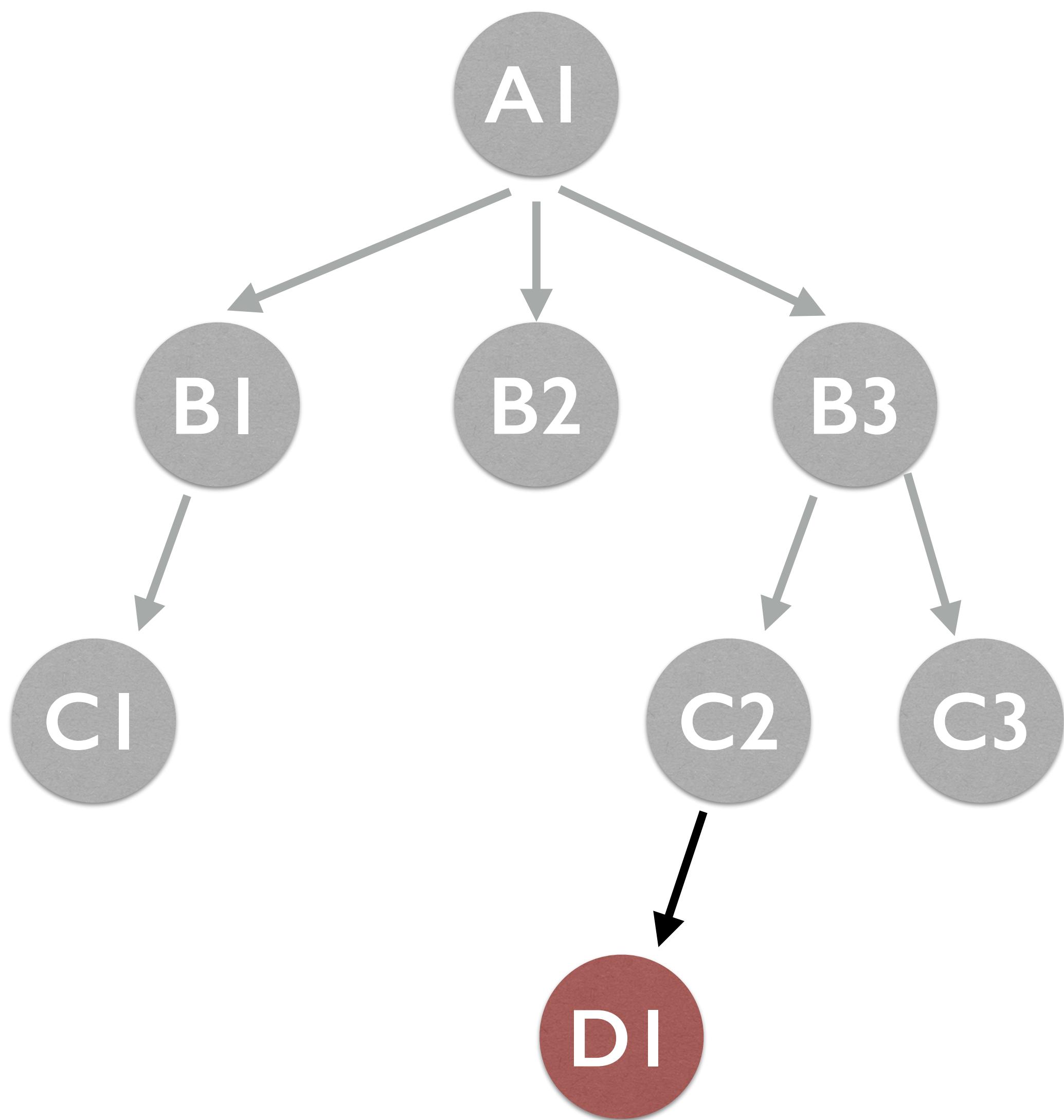
A1, B1, B2, B3, C1, C2

Breadth First Traversal



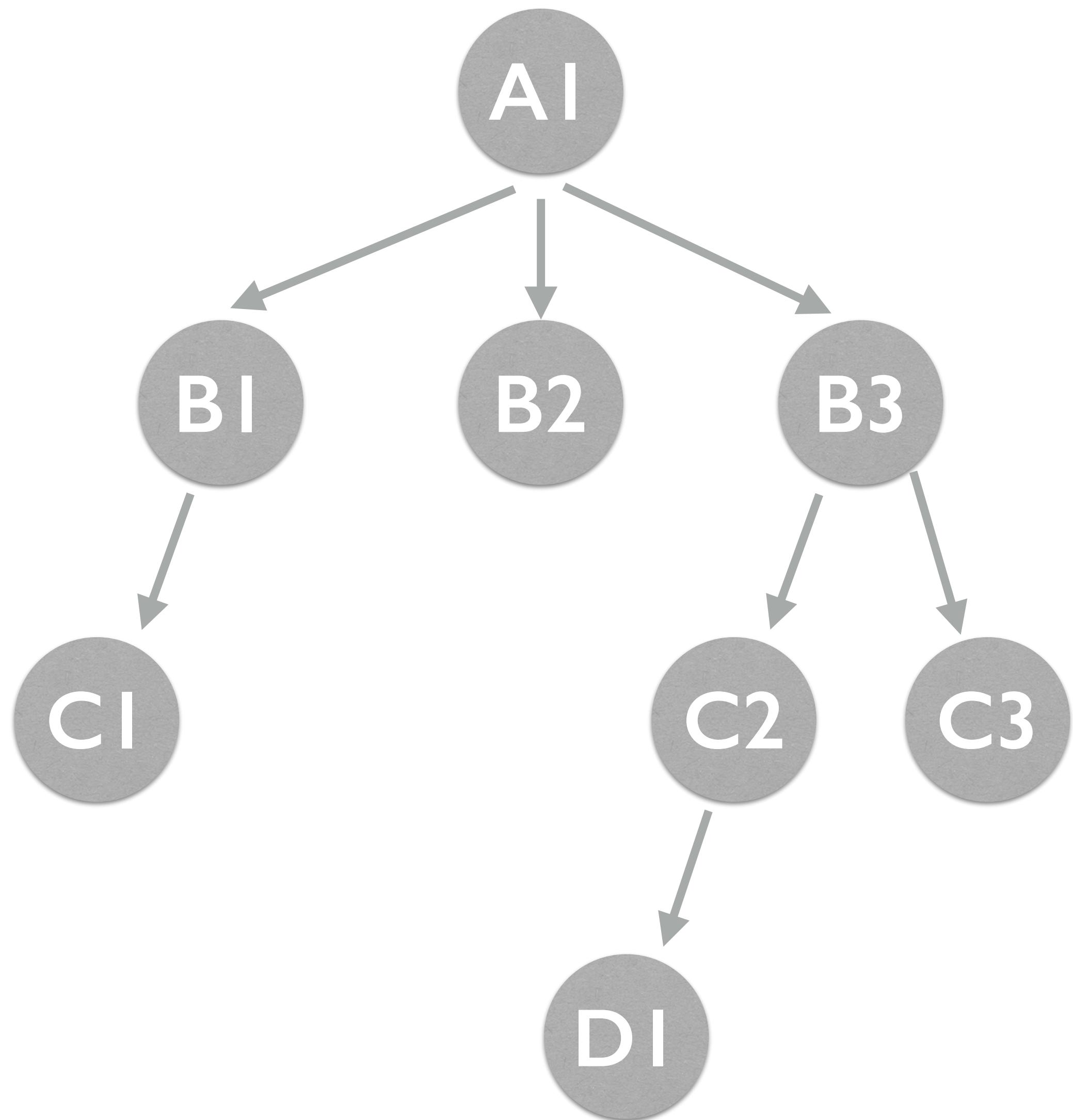
A1, B1, B2, B3, C1, C2, C3

Breadth First Traversal



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

Breadth First Traversal



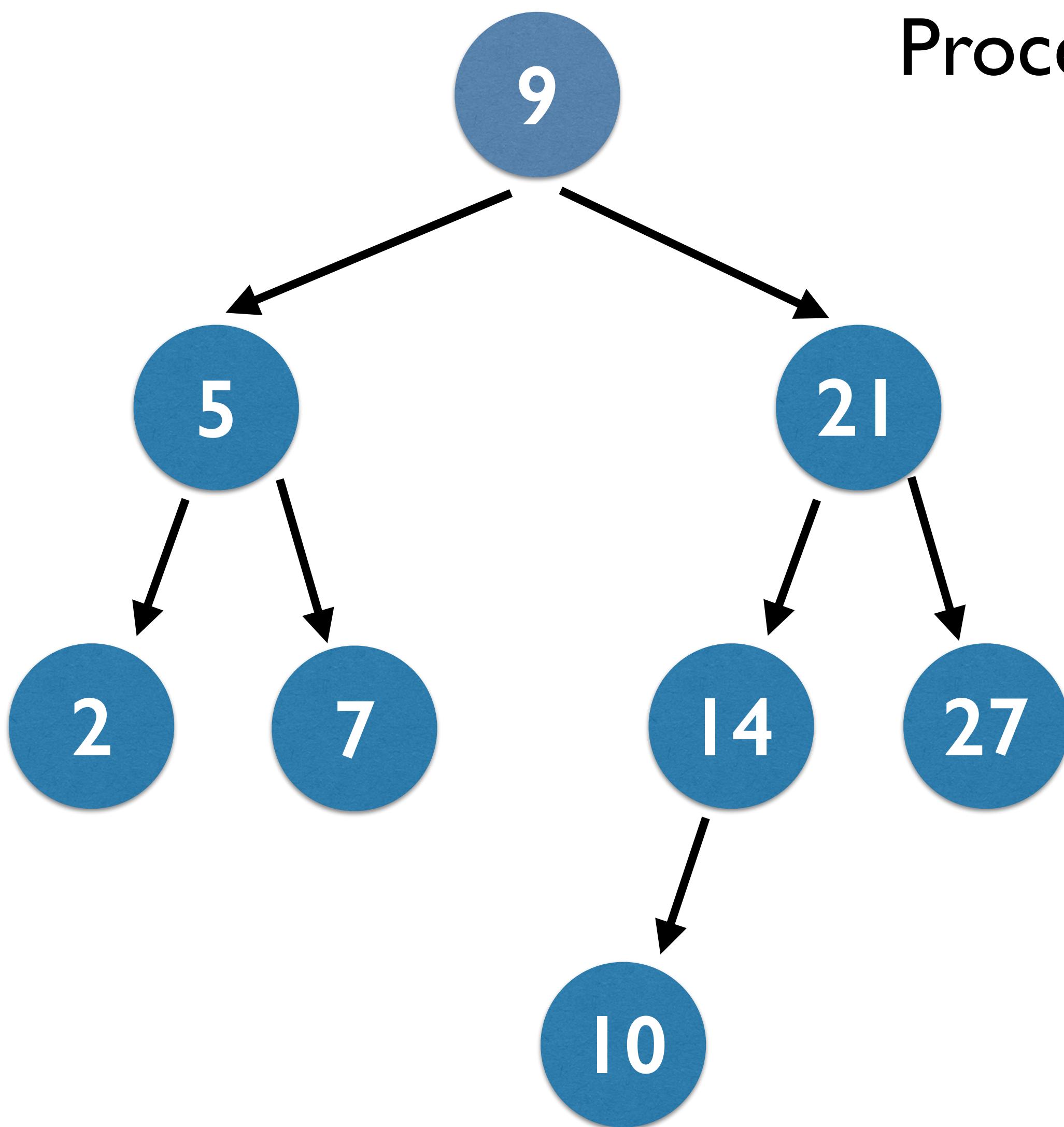
A1, B1, B2, B3, C1, C2, C3, D1

- 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

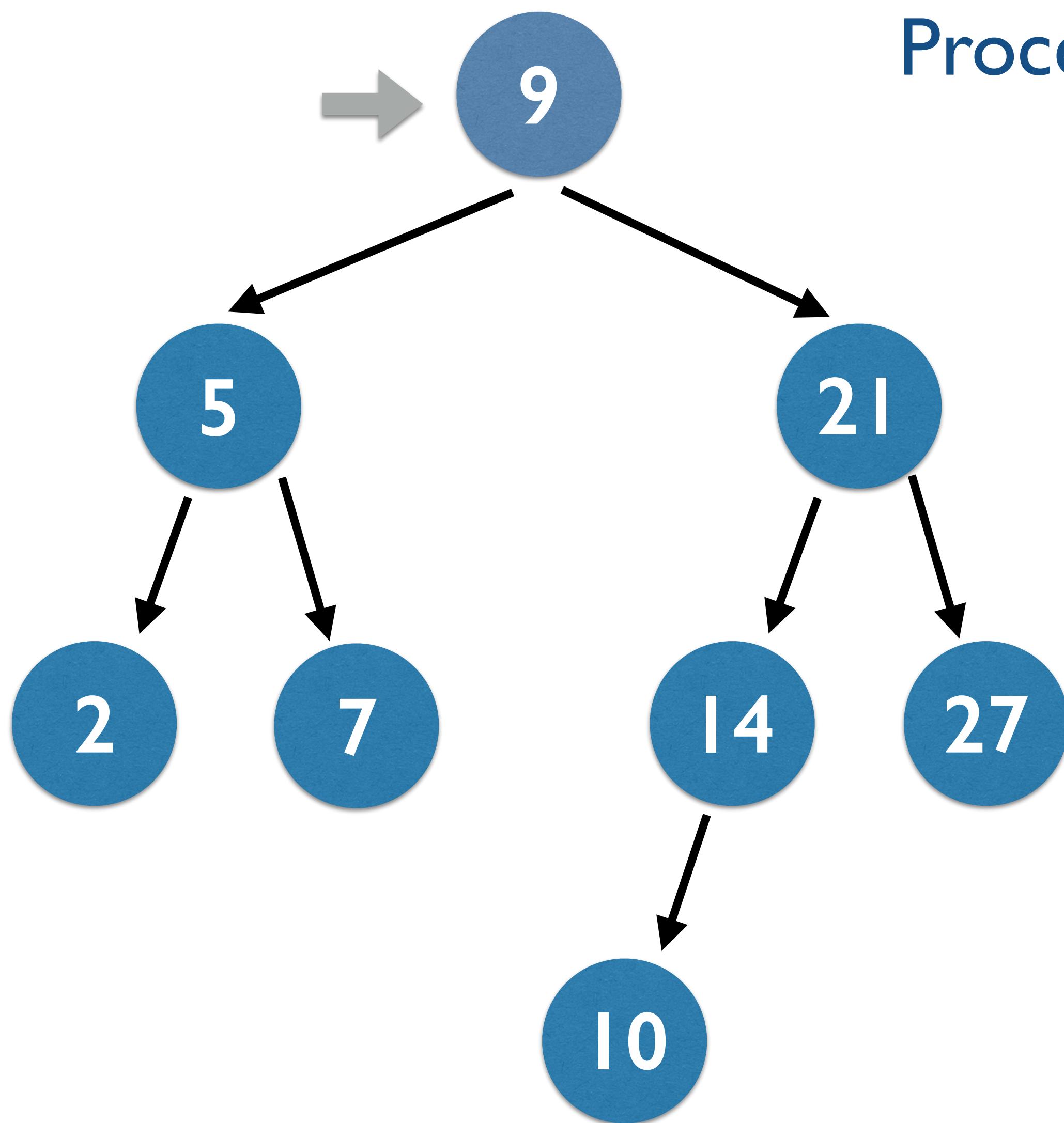
Depth First Traversal: In Order

Process left · Process root · Process right

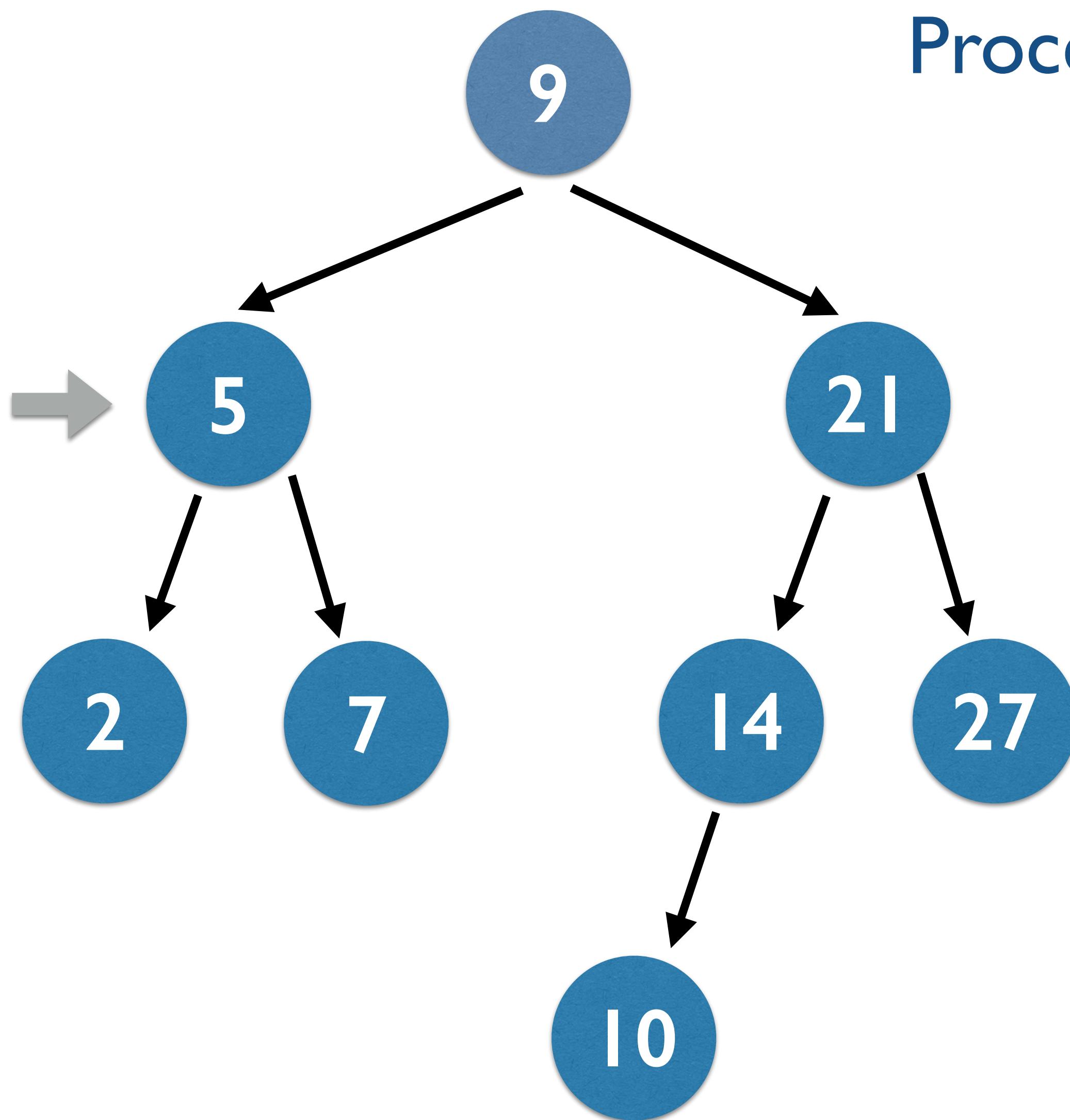


Depth First Traversal: In Order

Process left · Process root · Process right

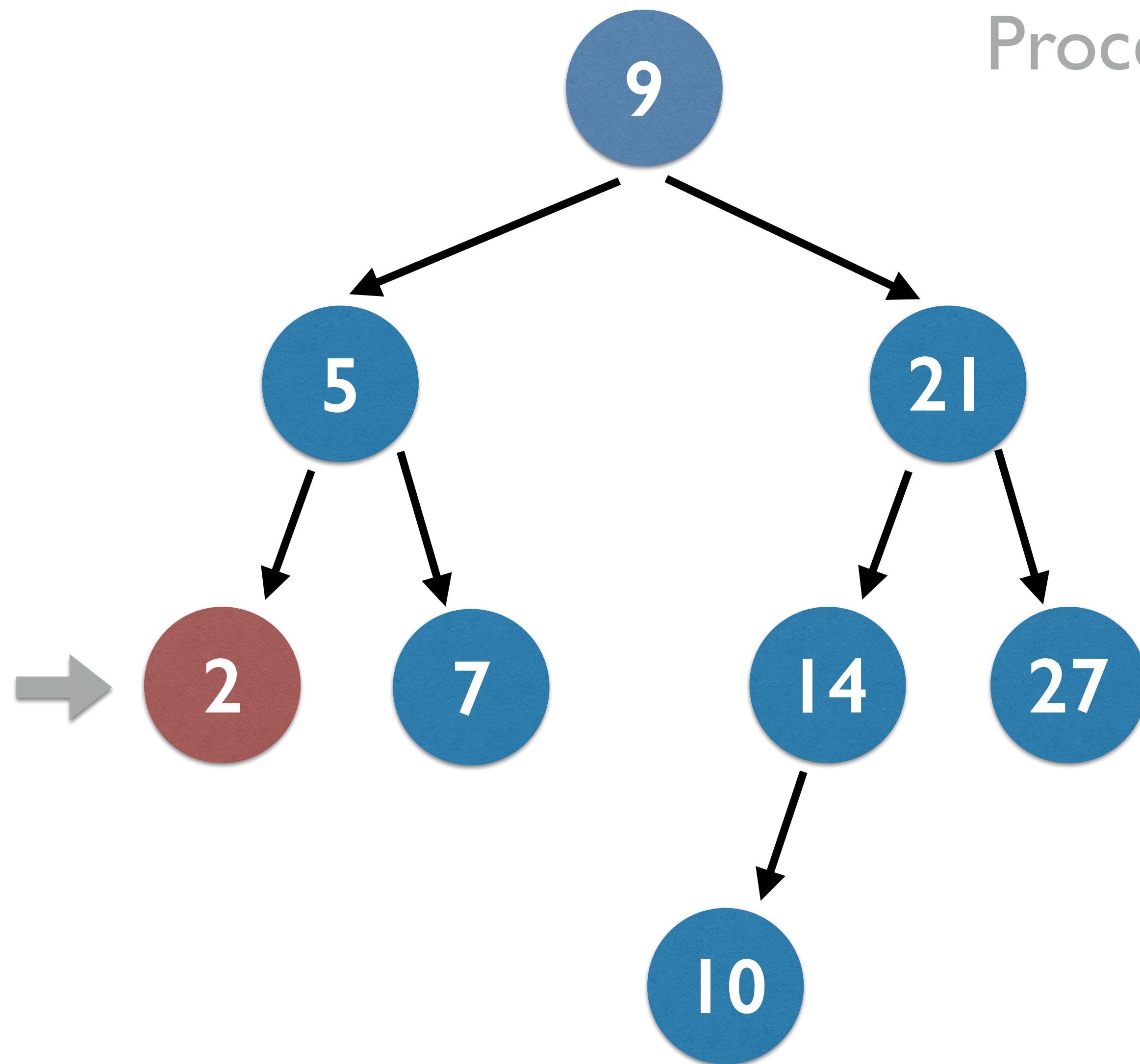


Depth First Traversal: In Order



Process left · Process root · Process right

Depth First Traversal: In Order

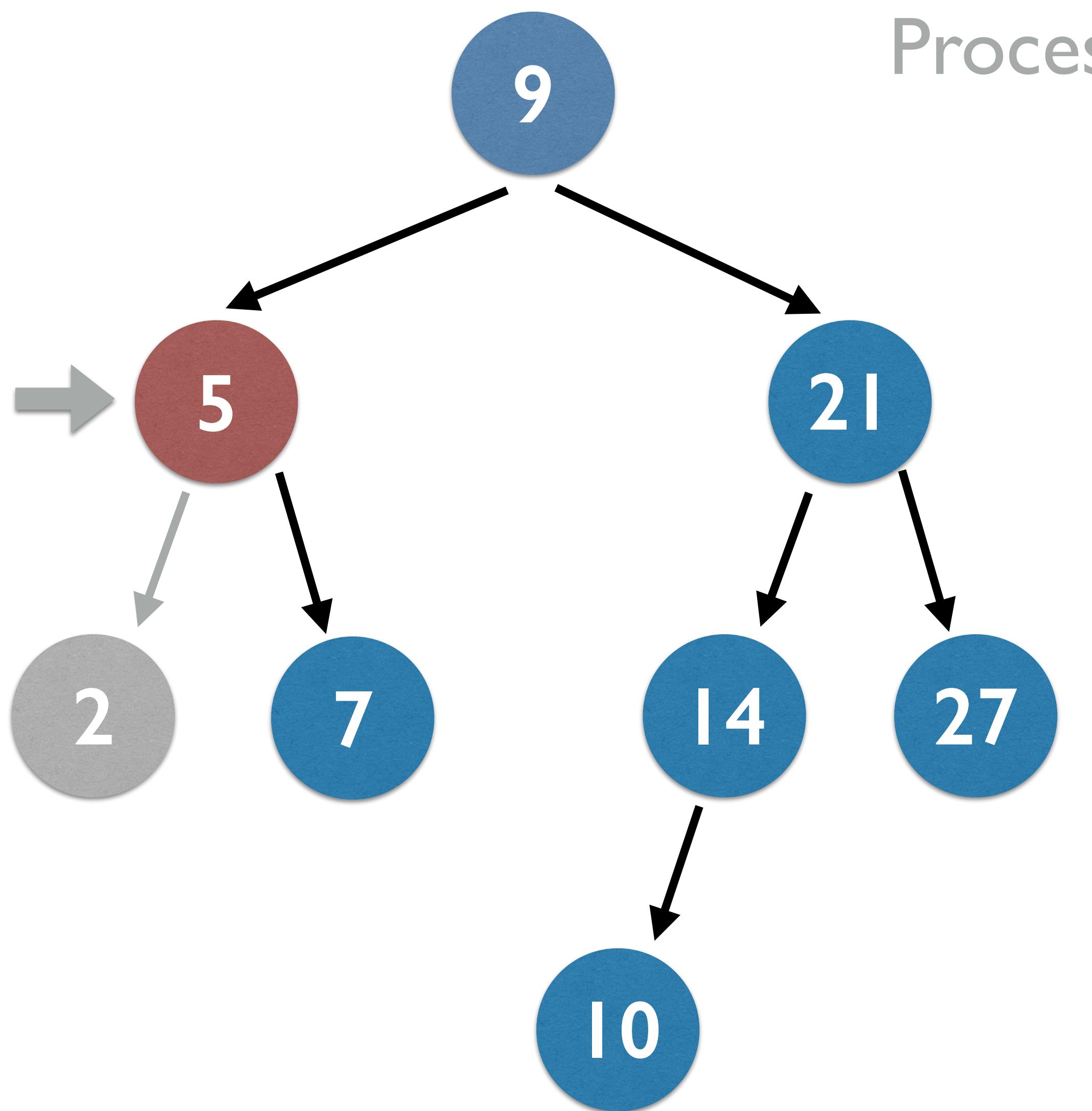


Process left • **Process root** • Process right

2

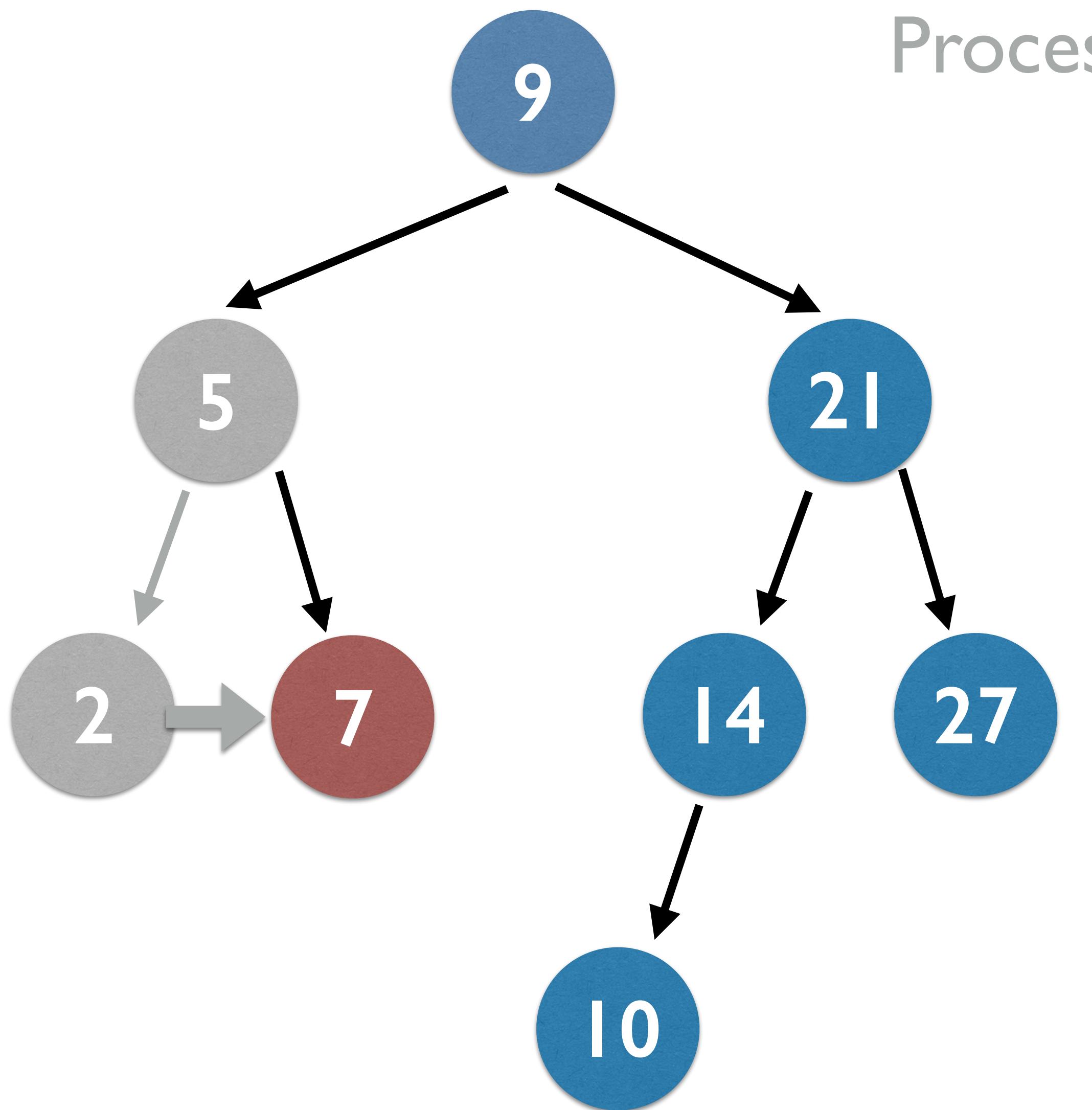
Depth First Traversal: In Order

Process left • **Process root** • Process right



2, 5

Depth First Traversal: In Order

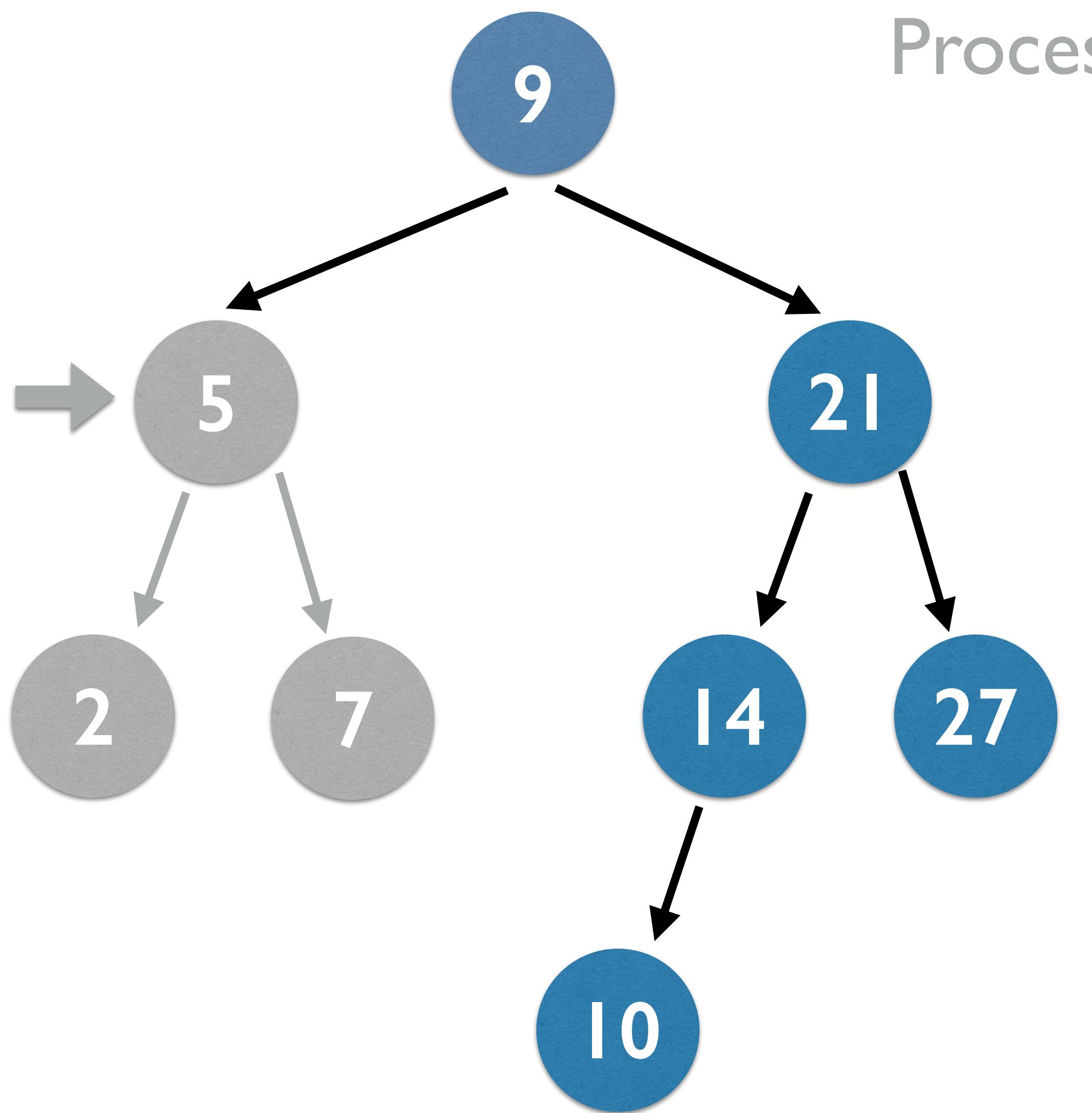


Process left · **Process root** · Process right

2, 5, 7

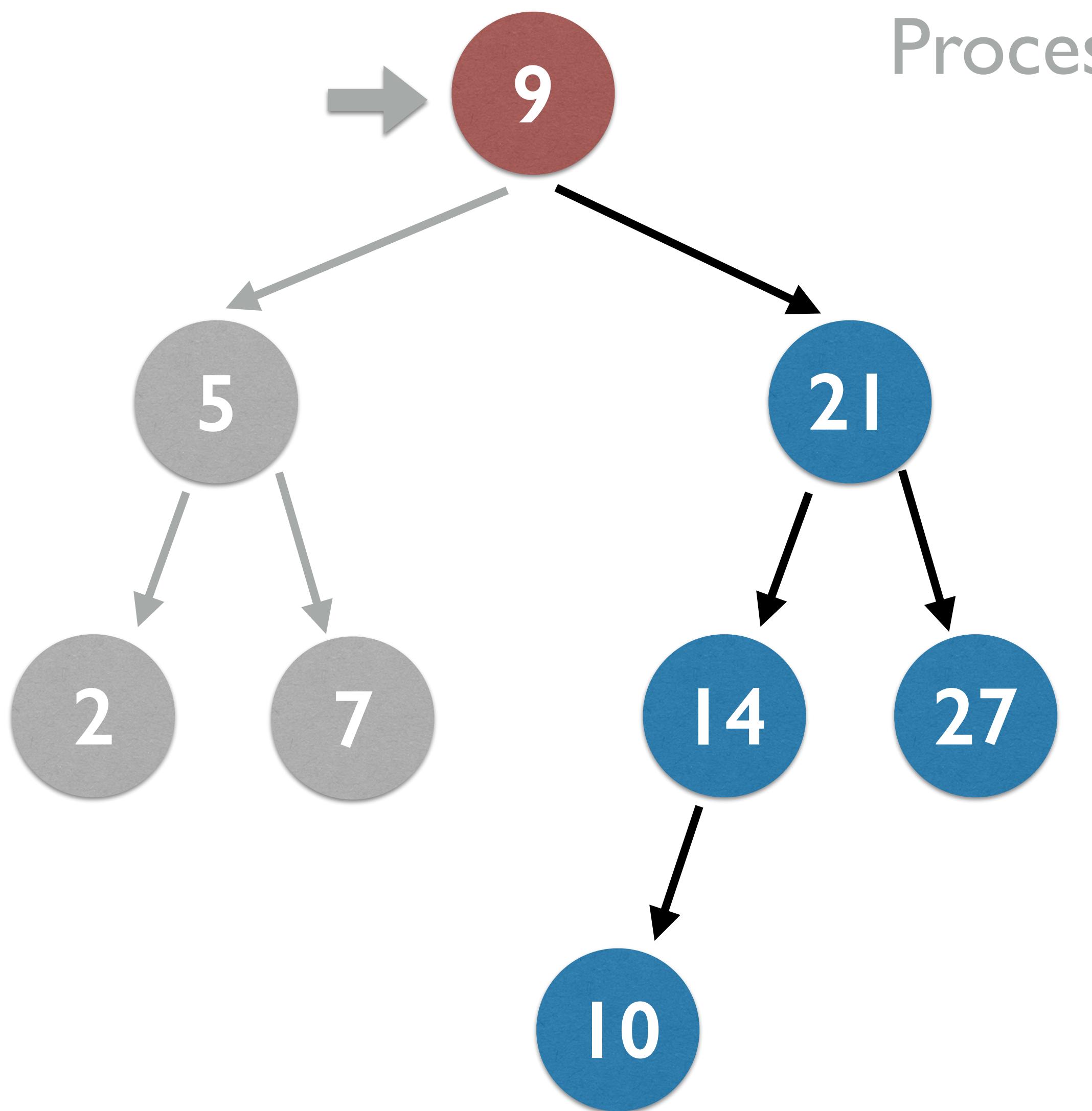
Depth First Traversal: In Order

Process left · Process root · Process right



2, 5, 7

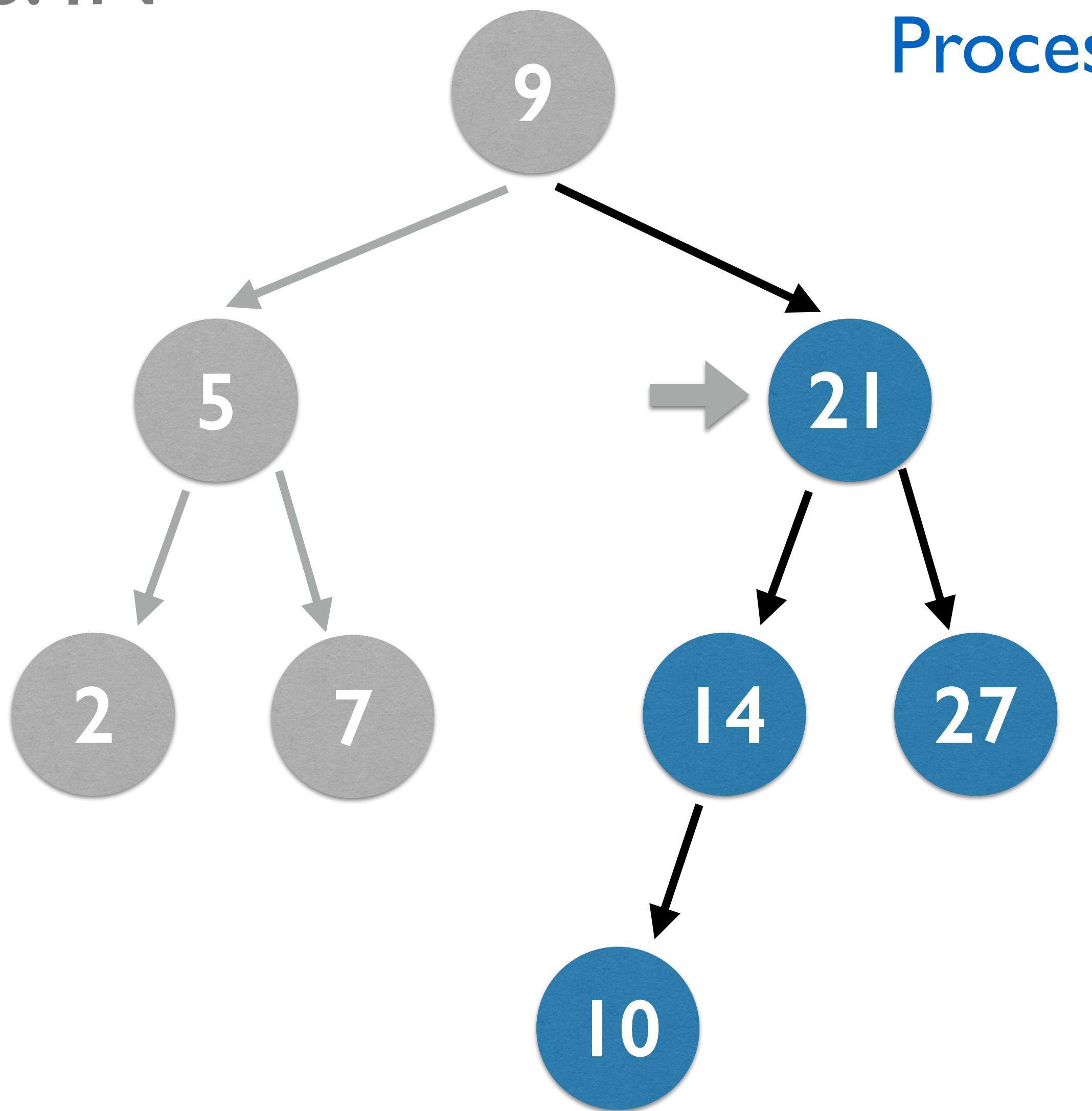
Depth First Traversal: In Order



Process left • **Process root** • Process right

2, 5, 7, 9

DFS: IN

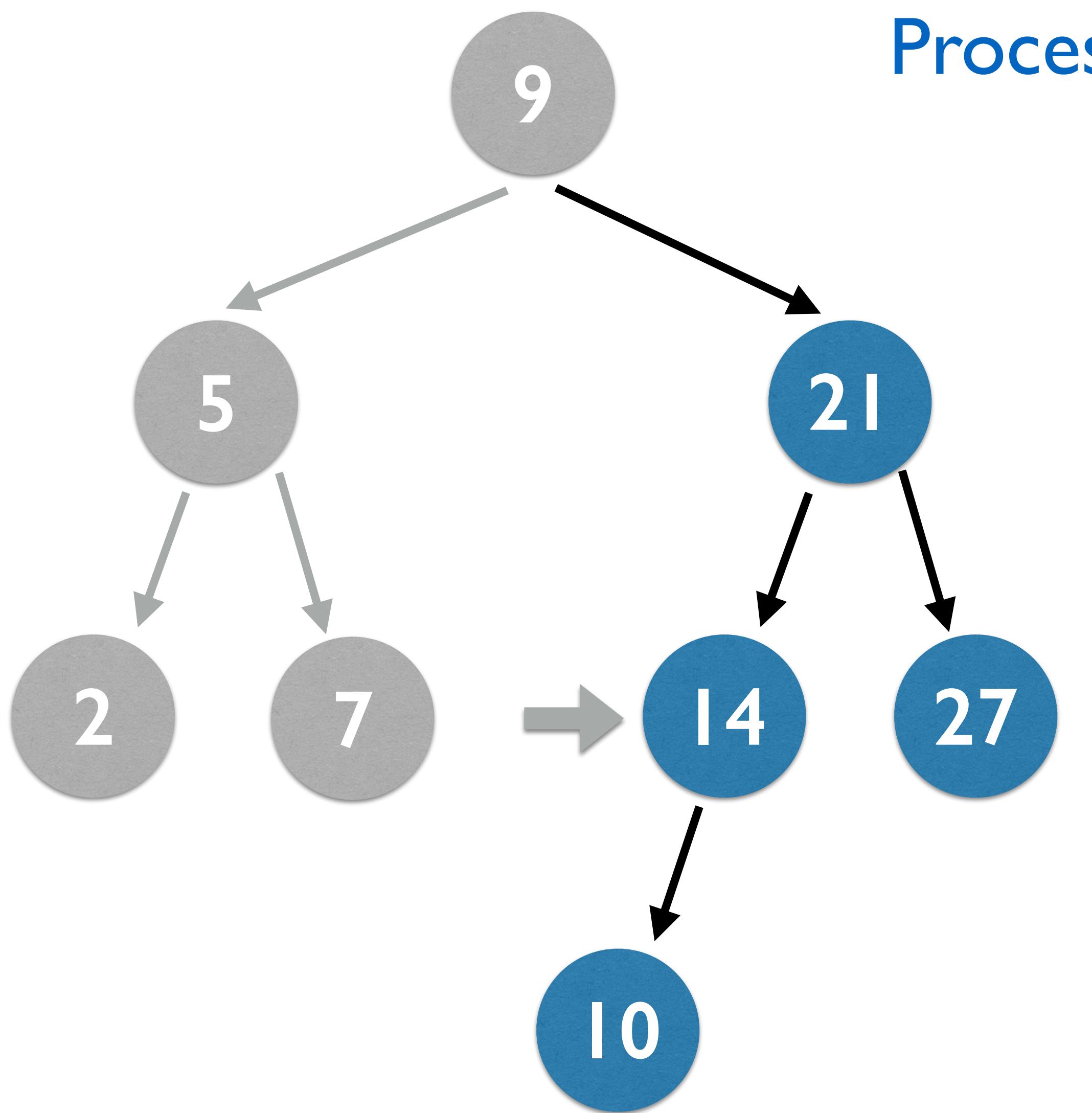


Process left · Process root · Process right

2, 5, 7, 9

Depth First Traversal: In Order

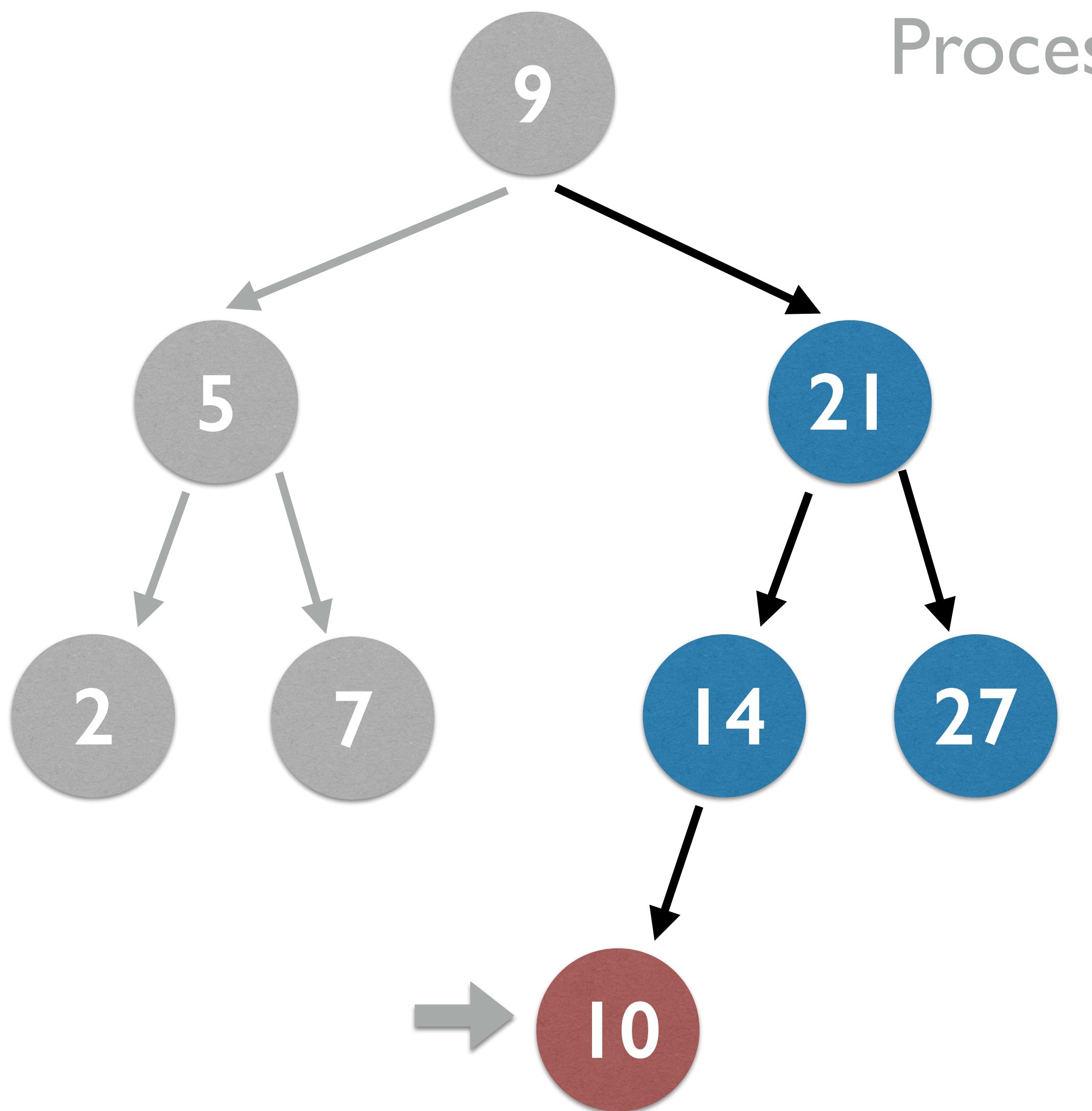
Process left · Process root · Process right



2, 5, 7, 9

Depth First Traversal: In Order

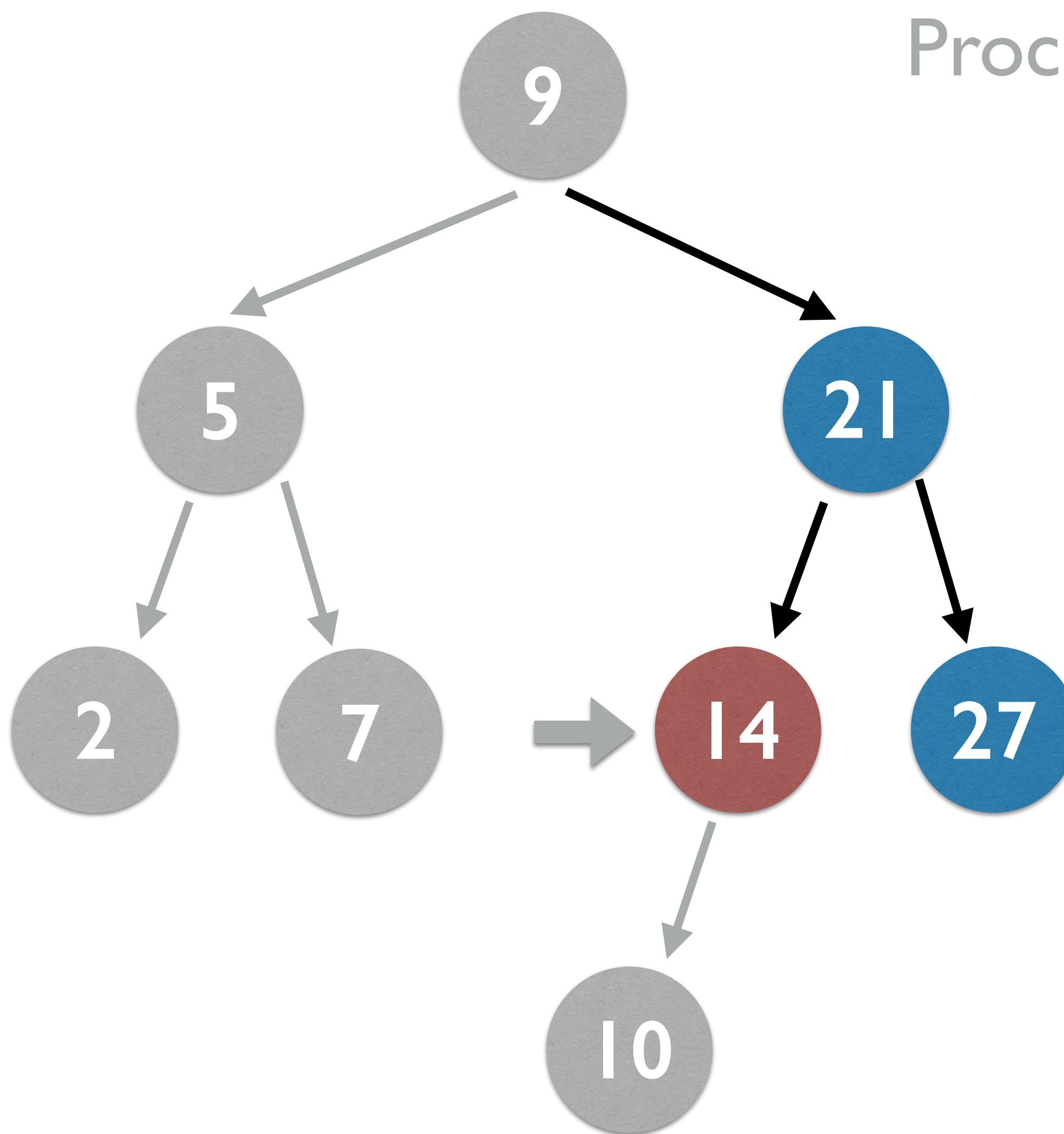
Process left · **Process root** · Process right



2, 5, 7, 9, 10

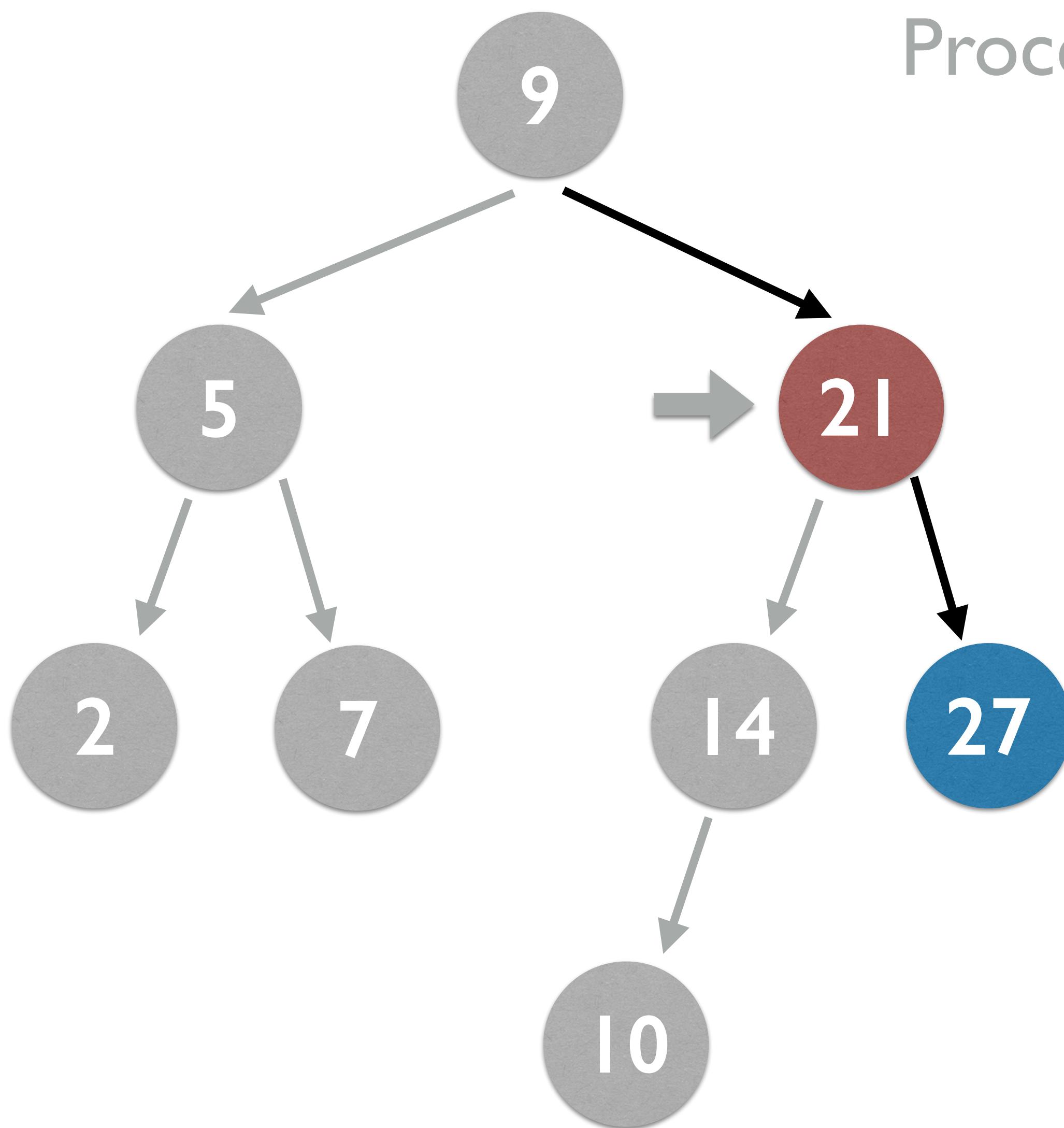
Depth First Traversal: In Order

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



2, 5, 7, 9, 10, 14

Depth First Traversal: In Order

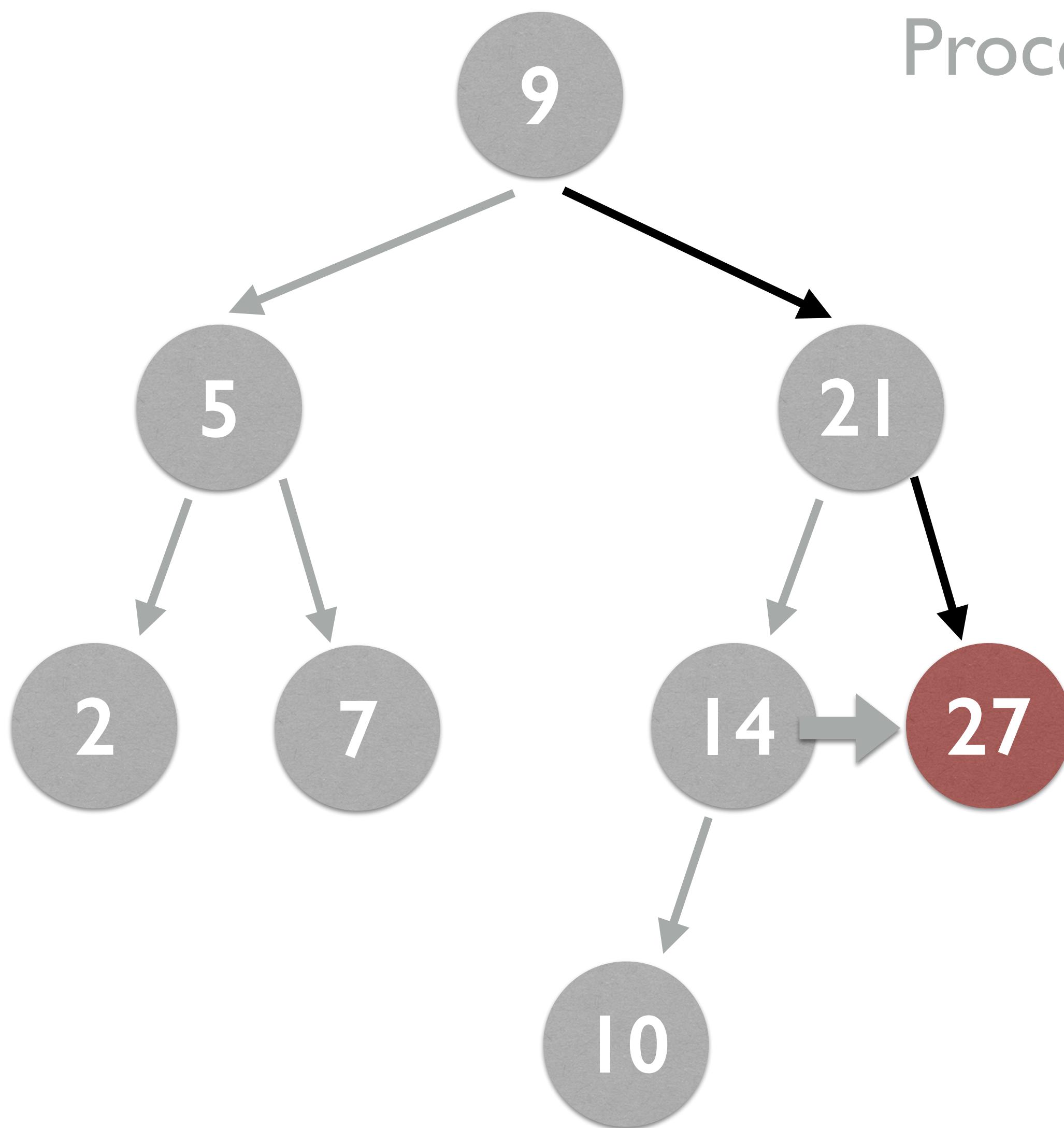


Process left • **Process root** • Process right

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

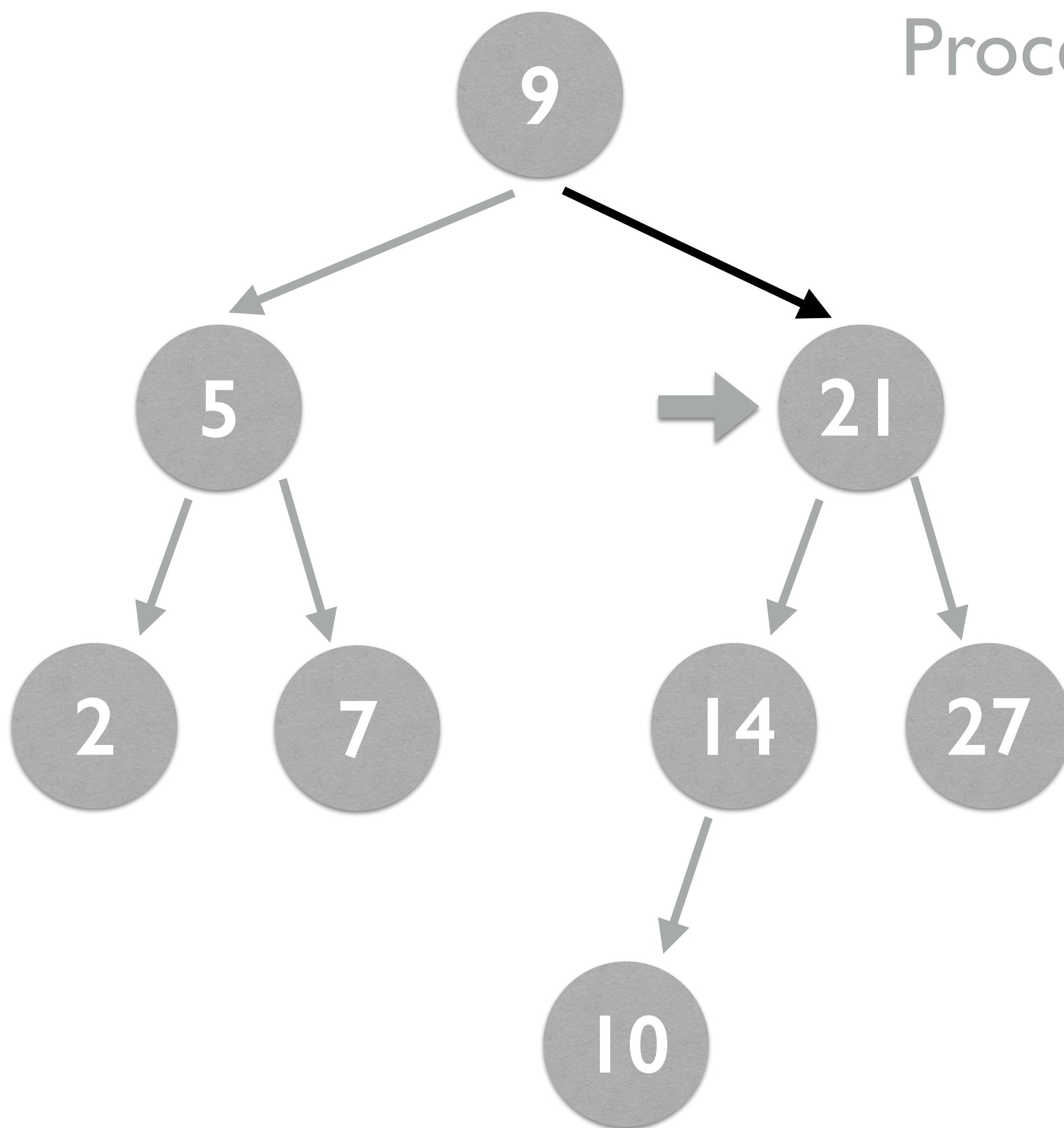
Depth First Traversal: In Order

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



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

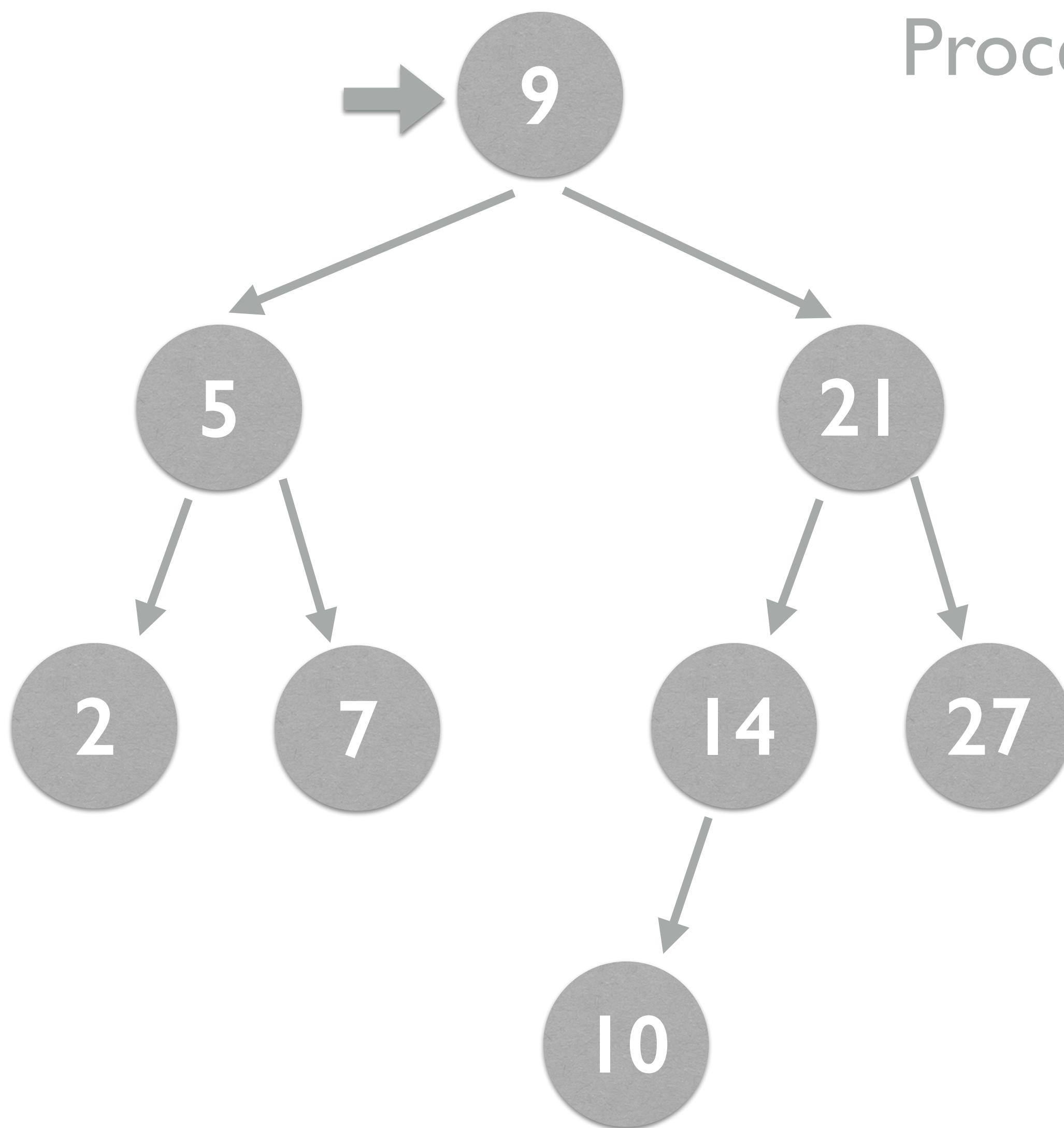
Depth First Traversal: In Order



Process left · Process root · Process right

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

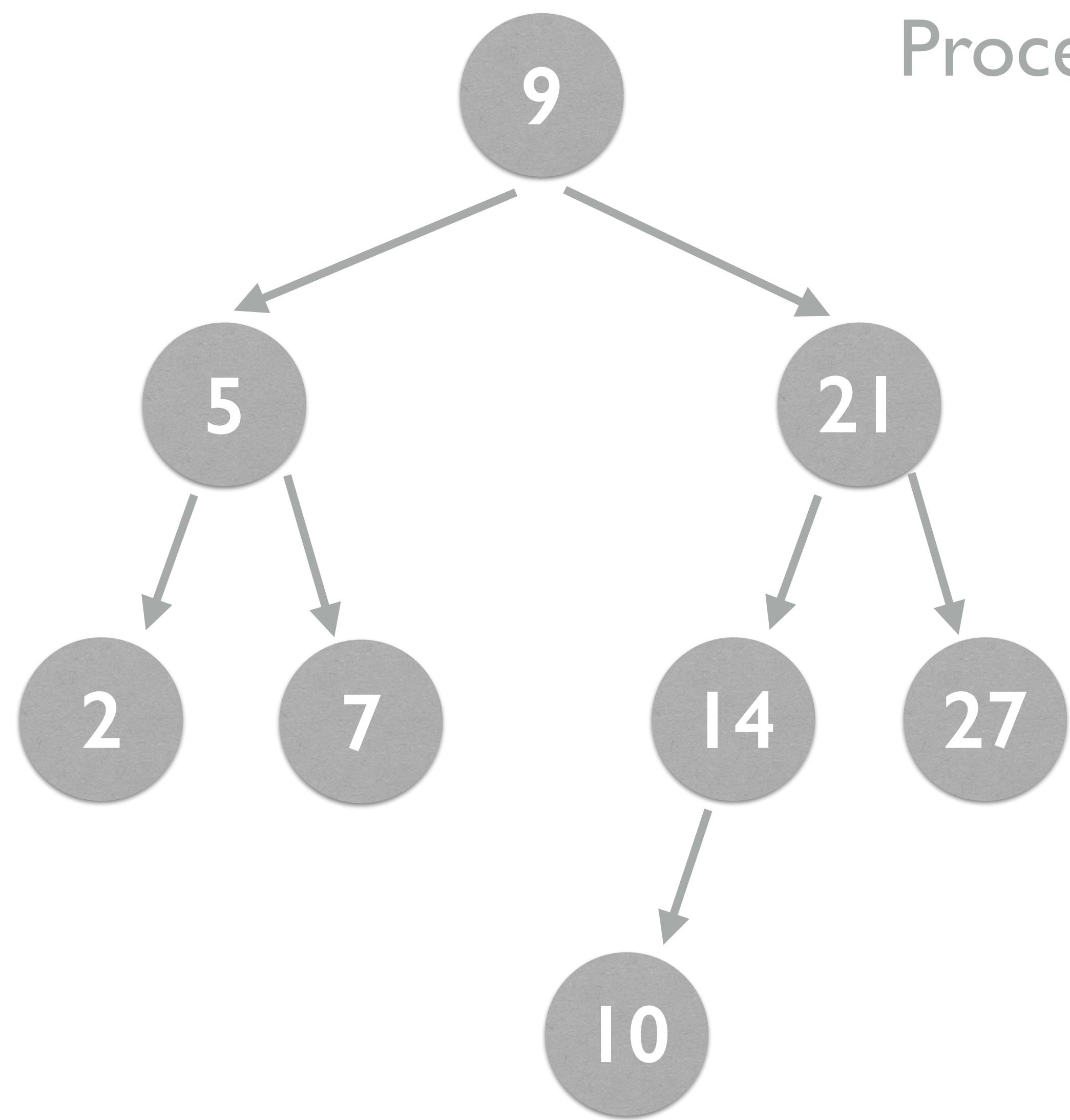
Depth First Traversal: In Order



Process left · Process root · Process right

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

Depth First Traversal: In Order



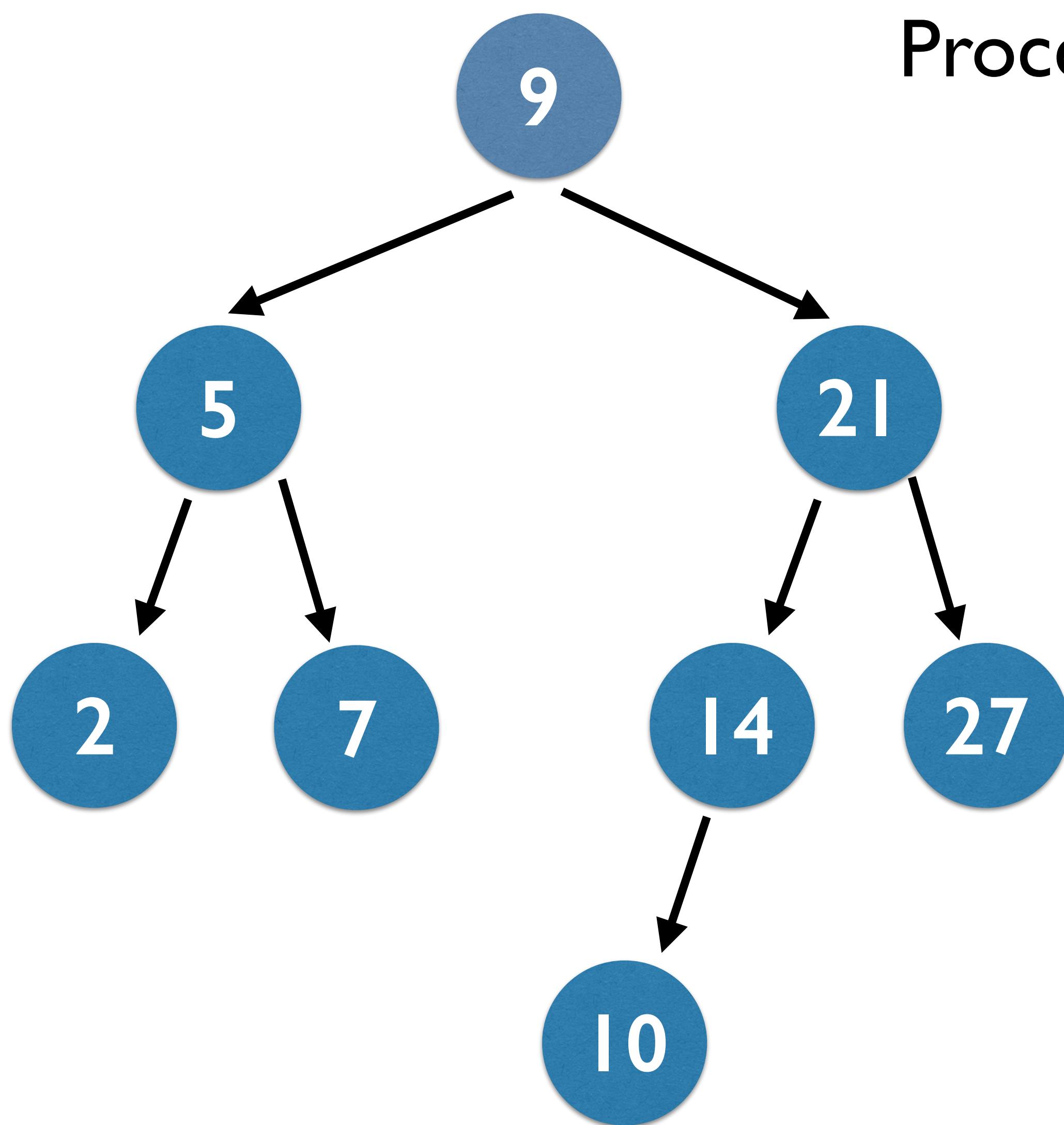
Process left • Process root • Process right

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

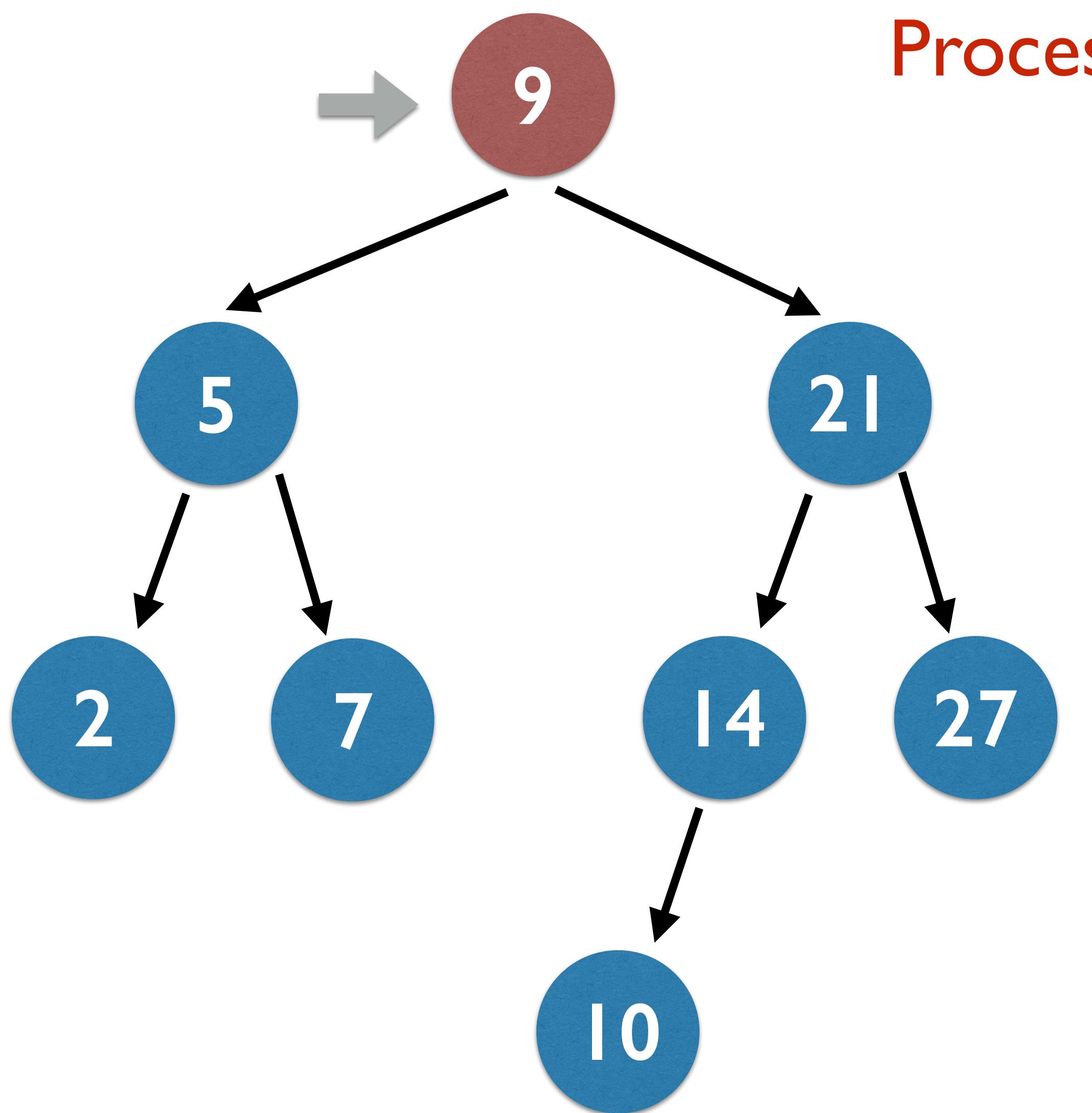
- In-order because respects ordering of nodes — nodes are processed smallest to largest value (leftmost to rightmost).
- The most generally useful depth-first traversal strategy for BSTs.

Depth First: Pre-Order

Depth First Traversal: Pre Order



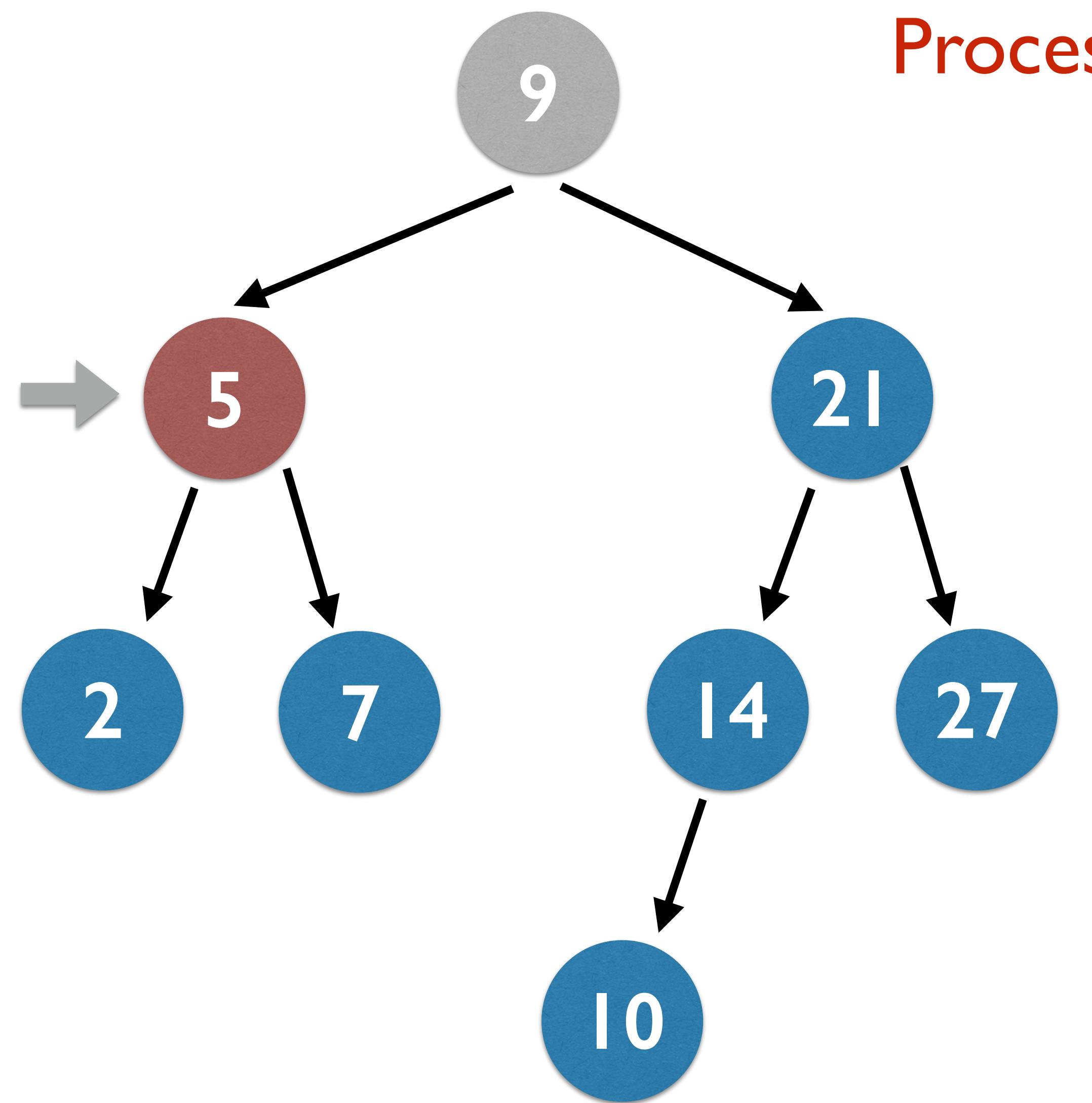
Depth First Traversal: Pre Order



Process root • Process left • Process right

9

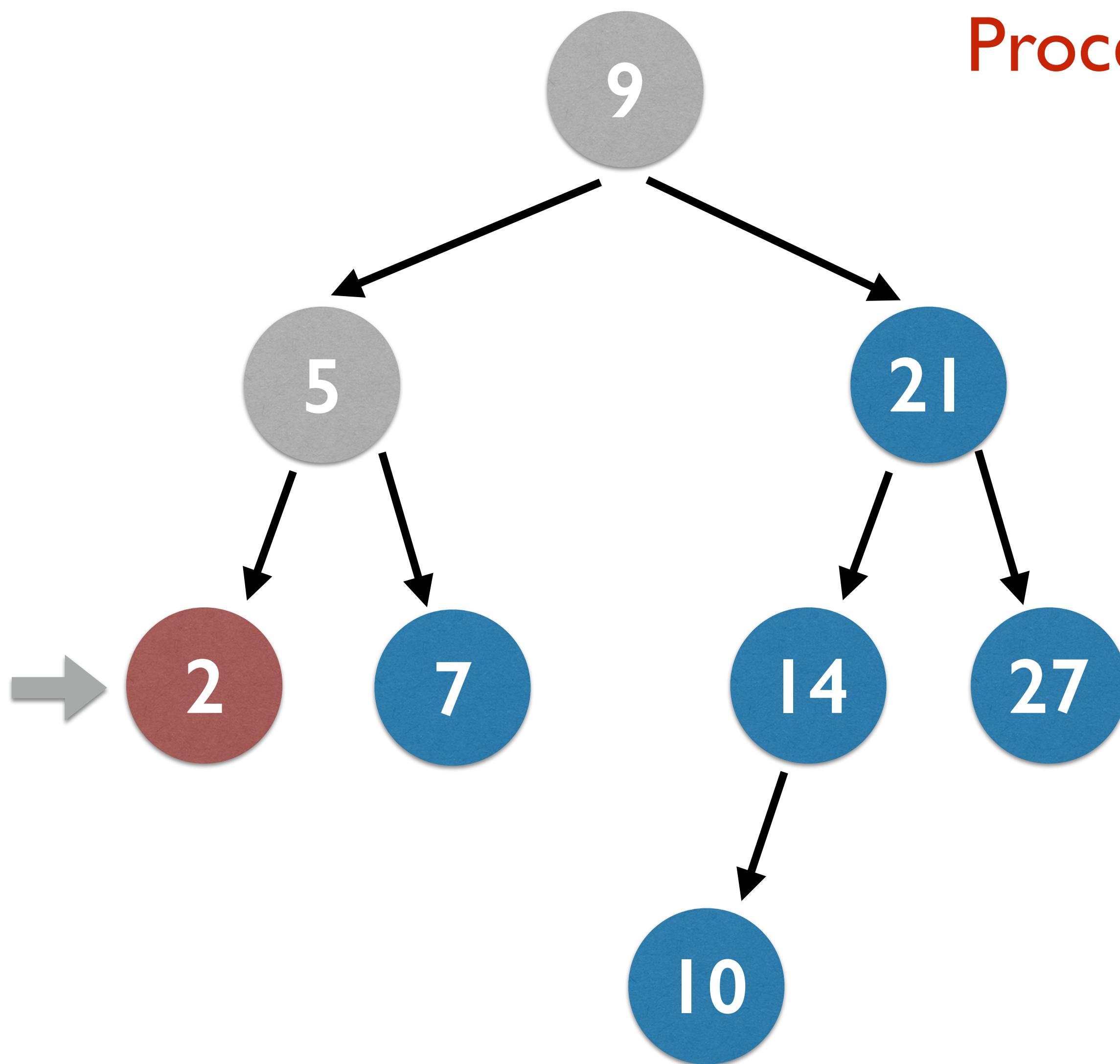
Depth First Traversal: Pre Order



Process root • Process left • Process right

9, 5

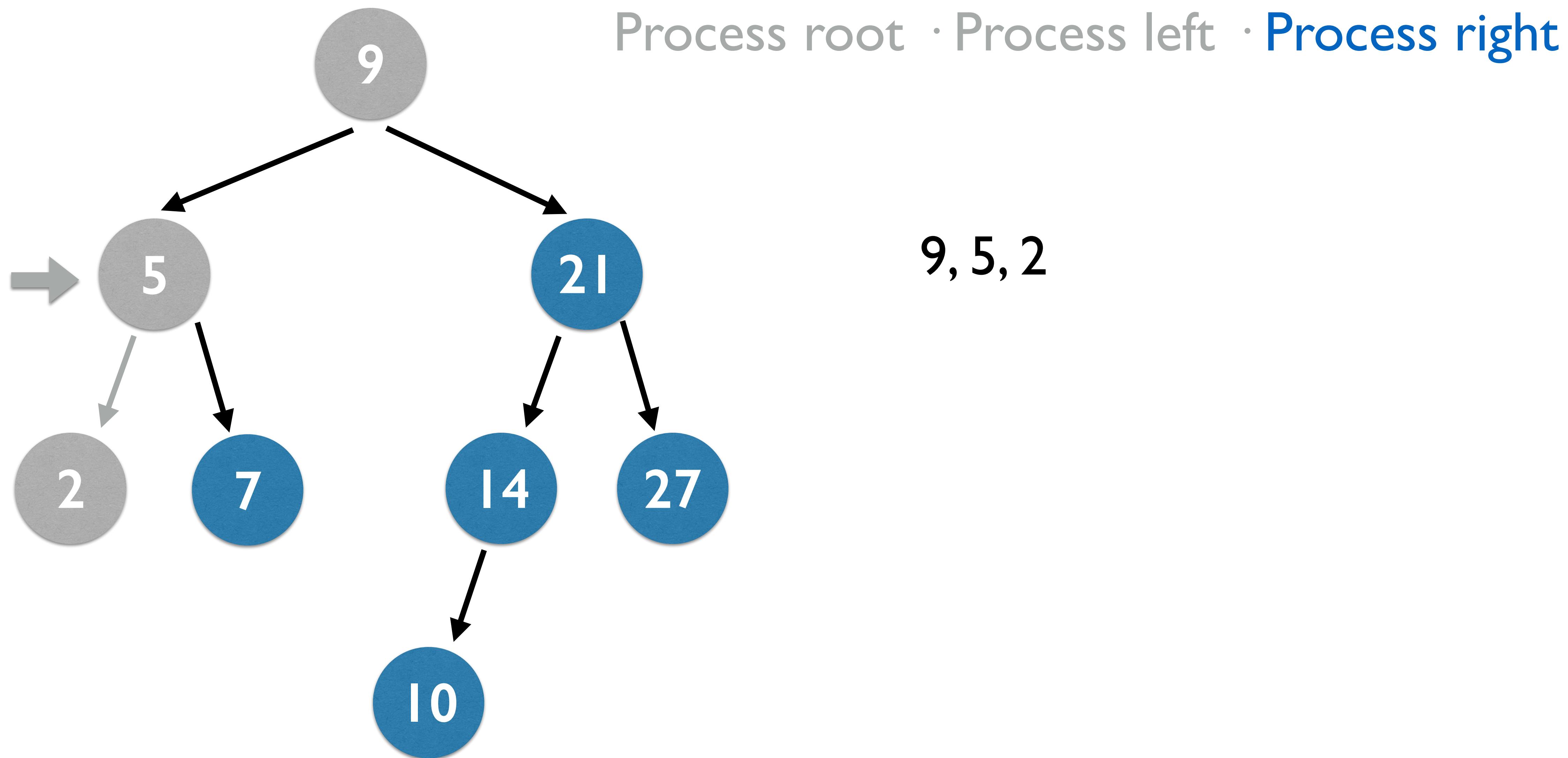
Depth First Traversal: Pre Order



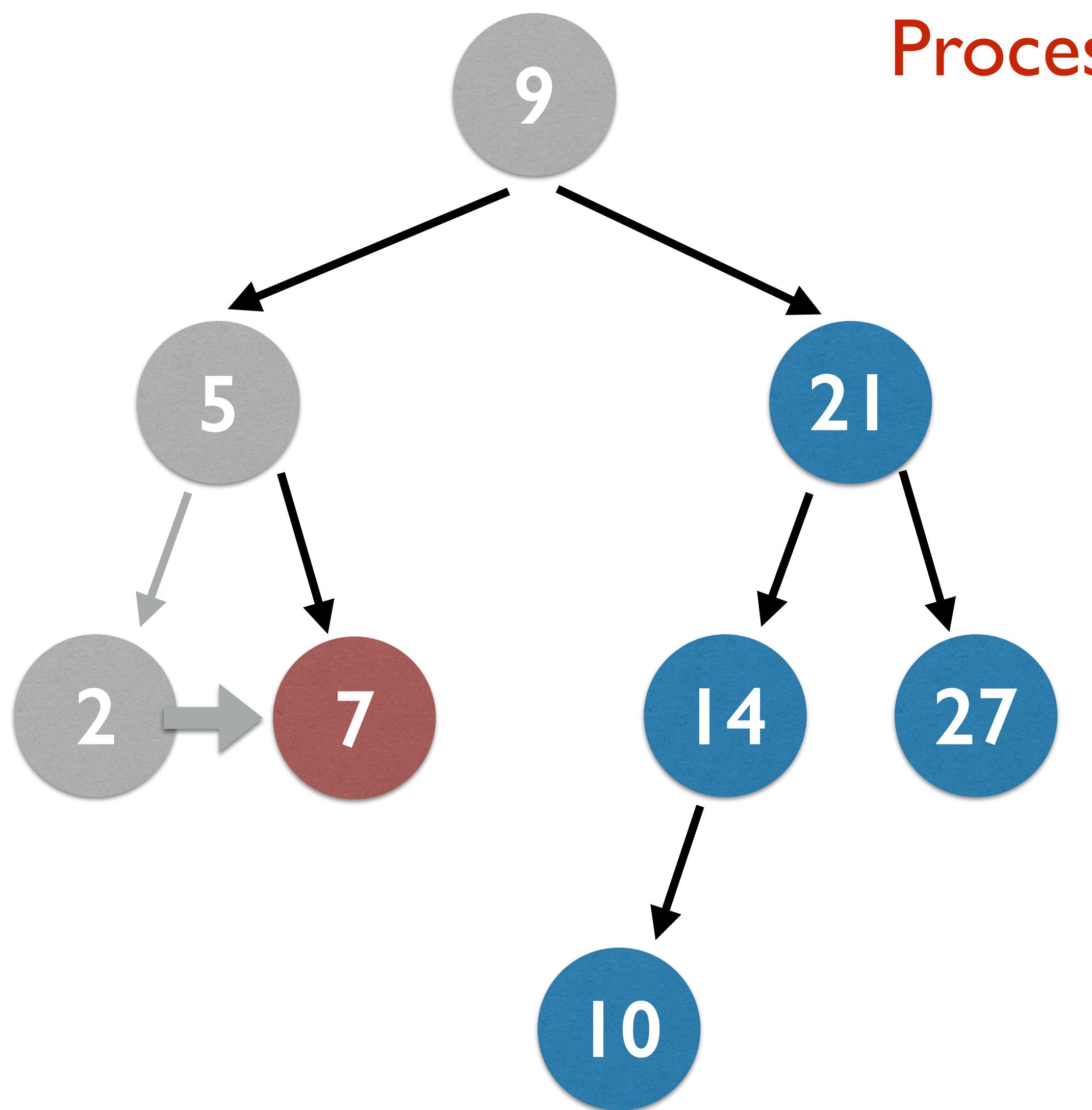
Process root • Process left • Process right

9, 5, 2

Depth First Traversal: Pre Order



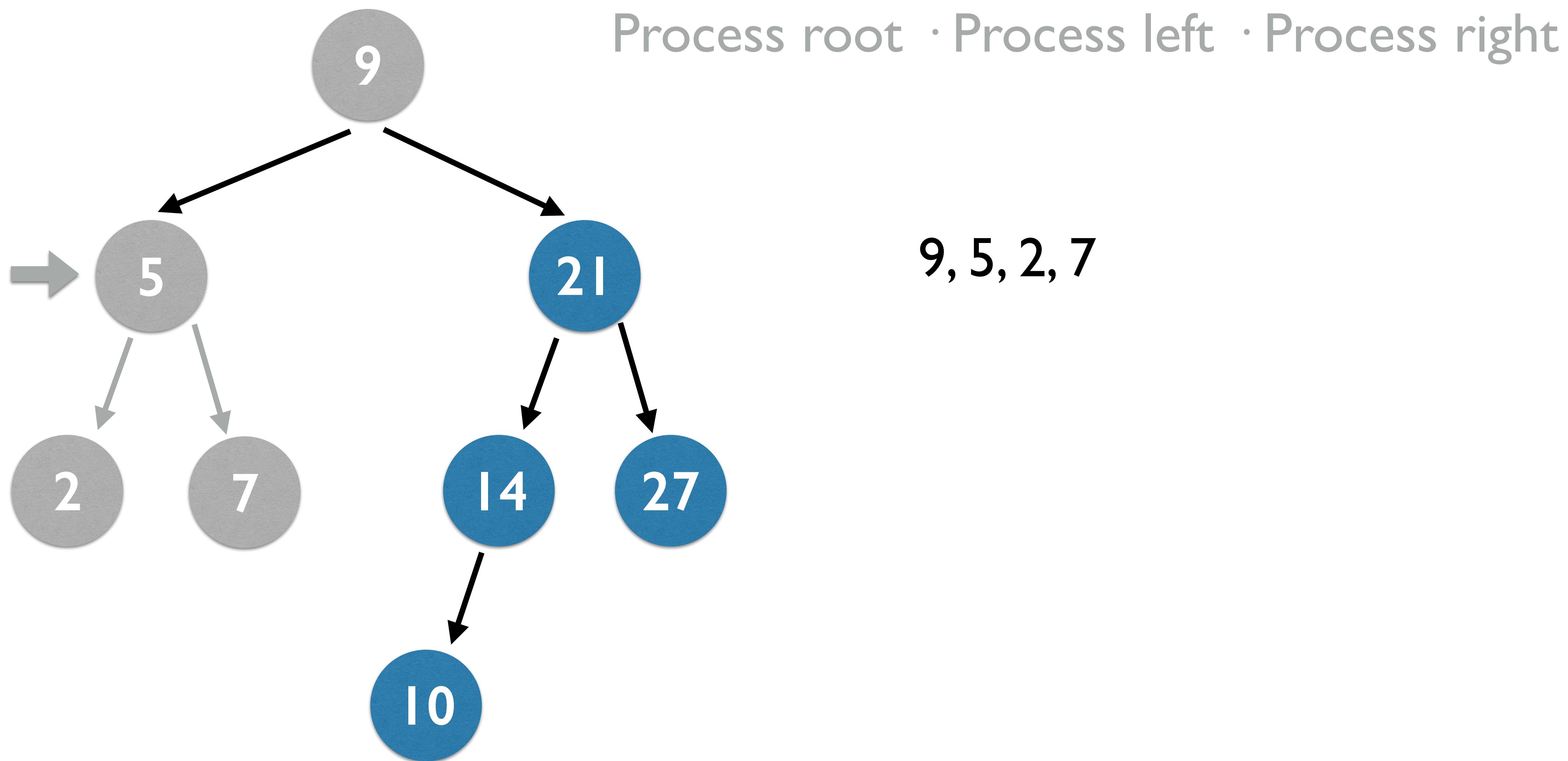
Depth First Traversal: Pre Order



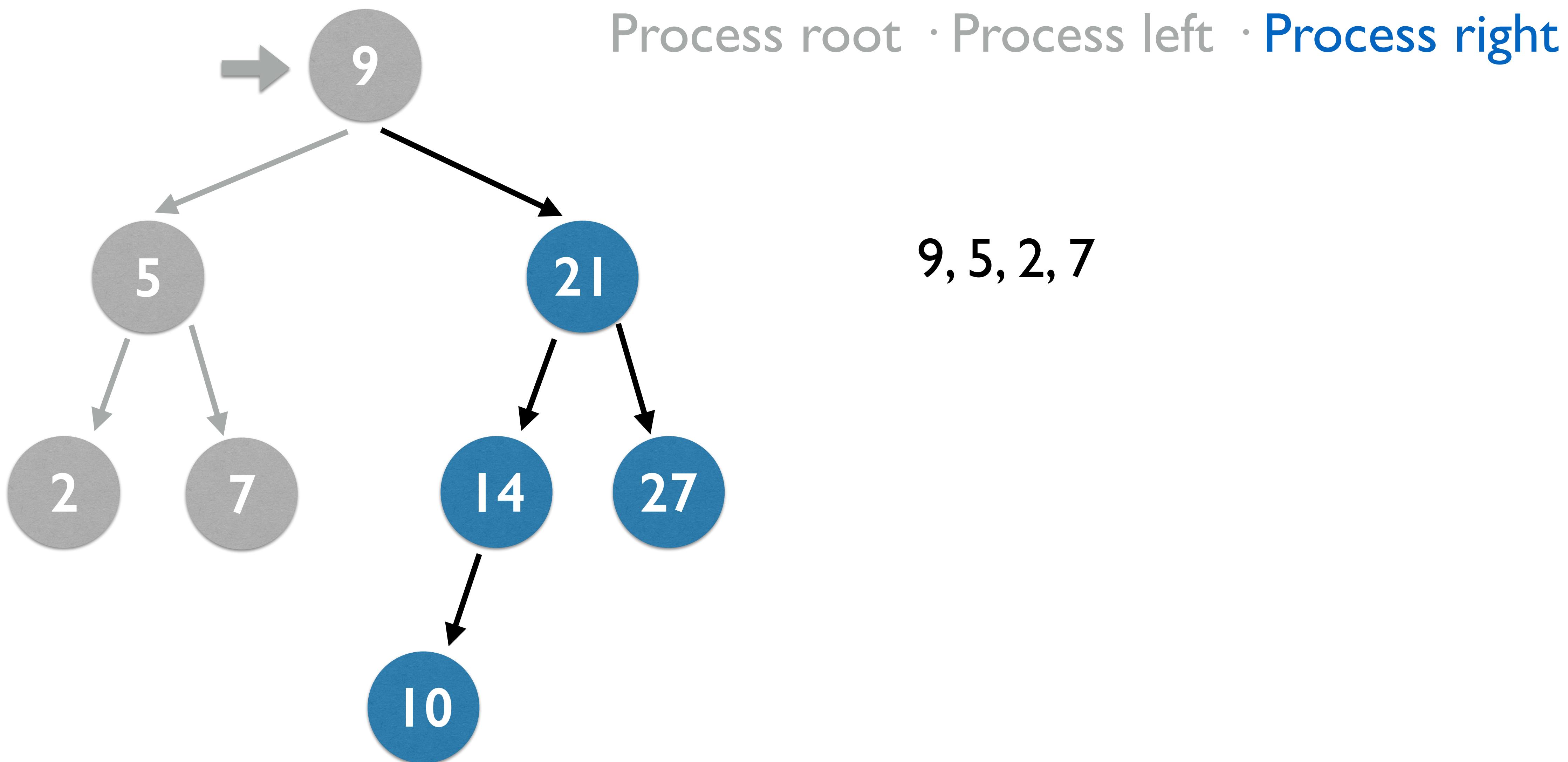
Process root • Process left • Process right

9, 5, 2, 7

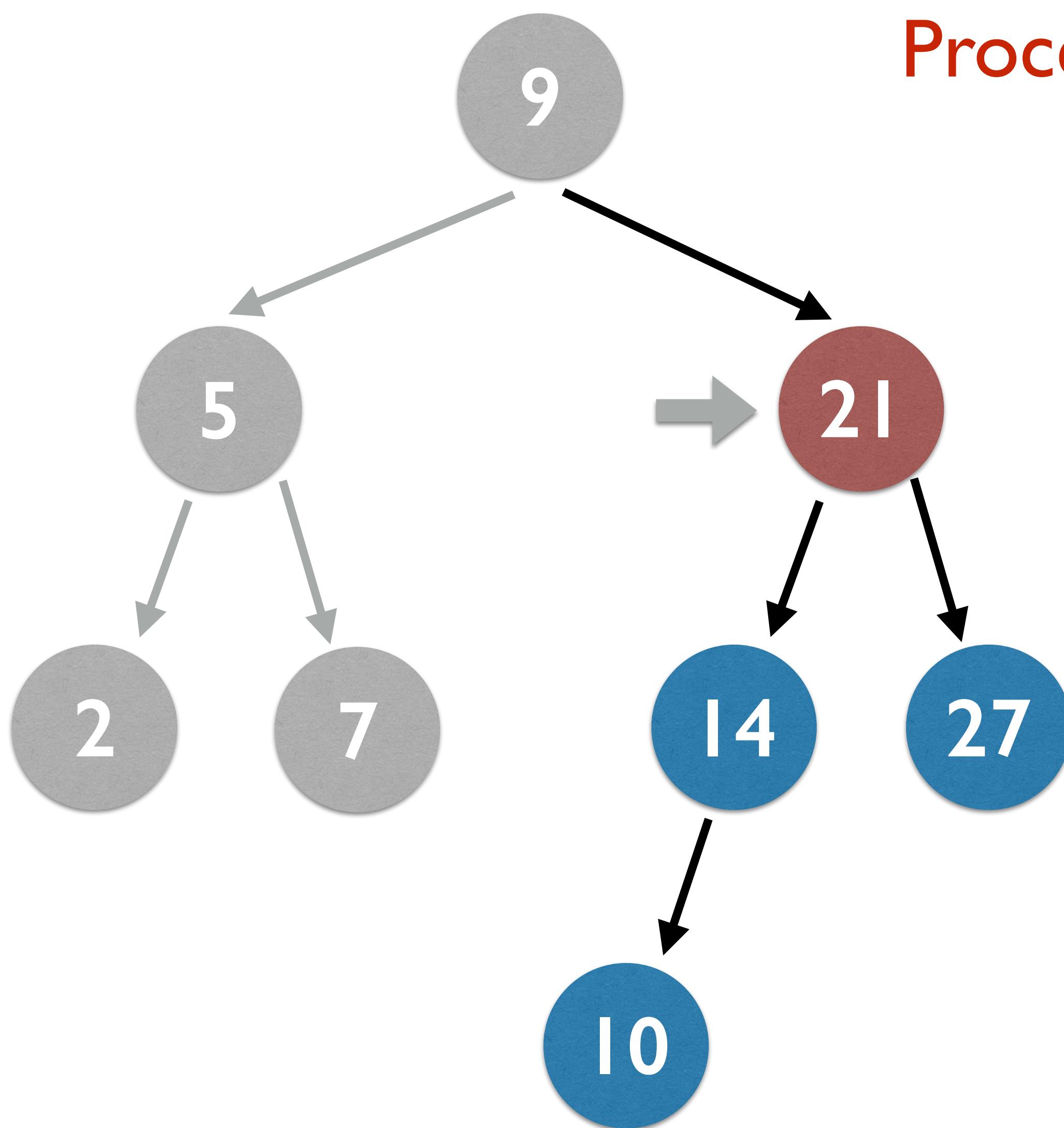
Depth First Traversal: Pre Order



Depth First Traversal: Pre Order



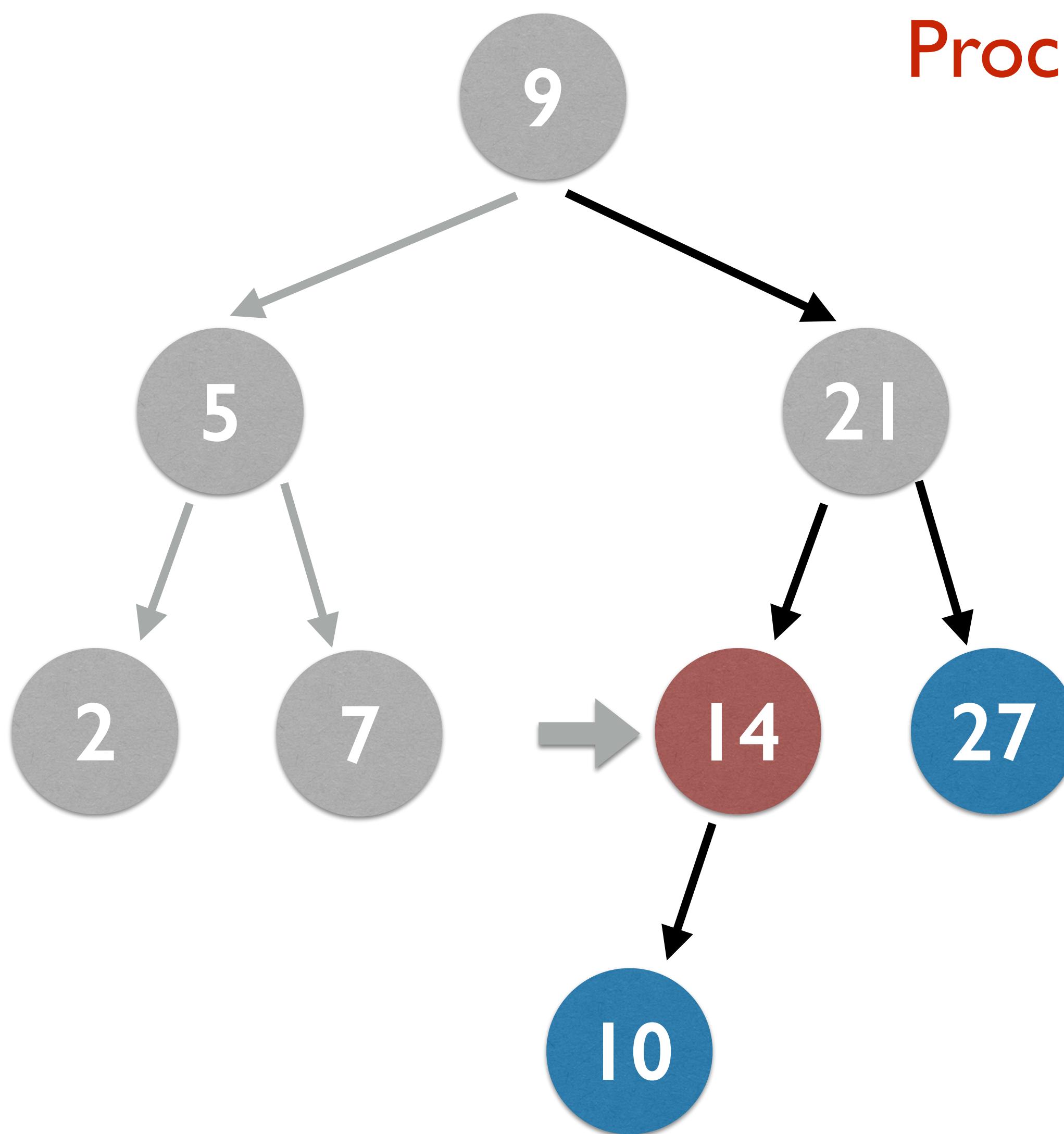
Depth First Traversal: Pre Order



Process root · Process left · Process right

9, 5, 2, 7, 21

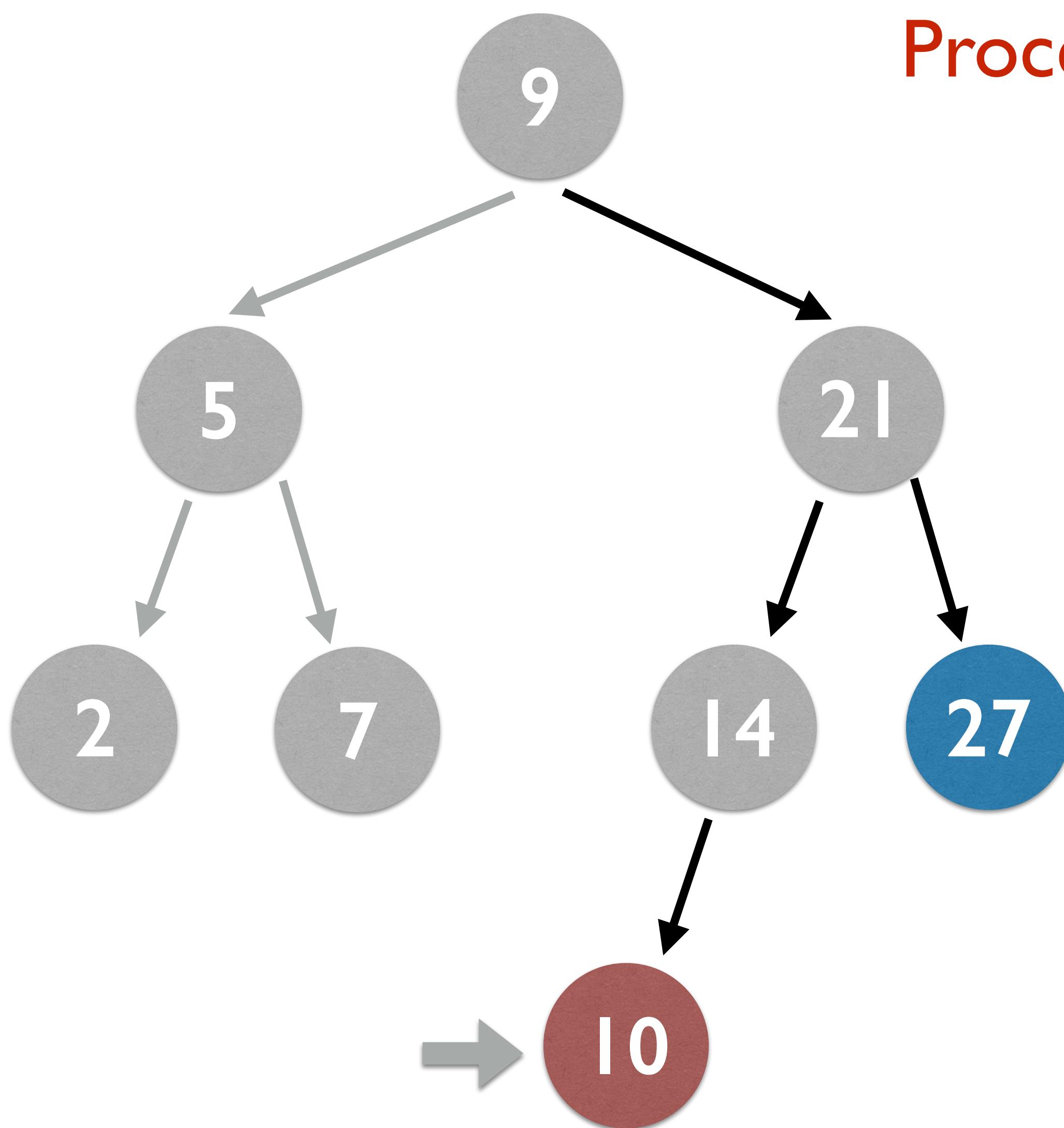
Depth First Traversal: Pre Order



Process root · Process left · Process right

9, 5, 2, 7, 21, 14

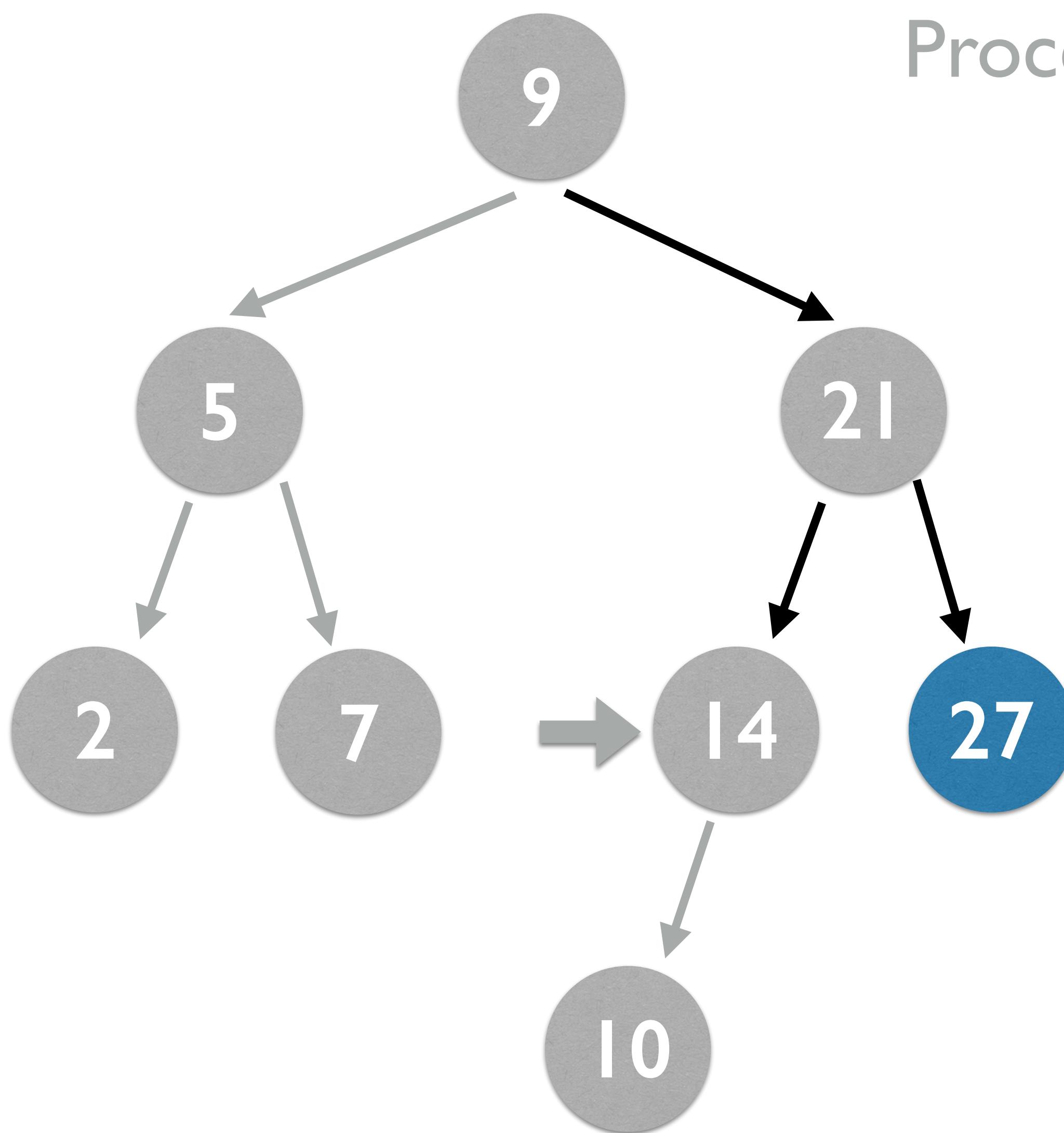
Depth First Traversal: Pre Order



Process root • Process left • Process right

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

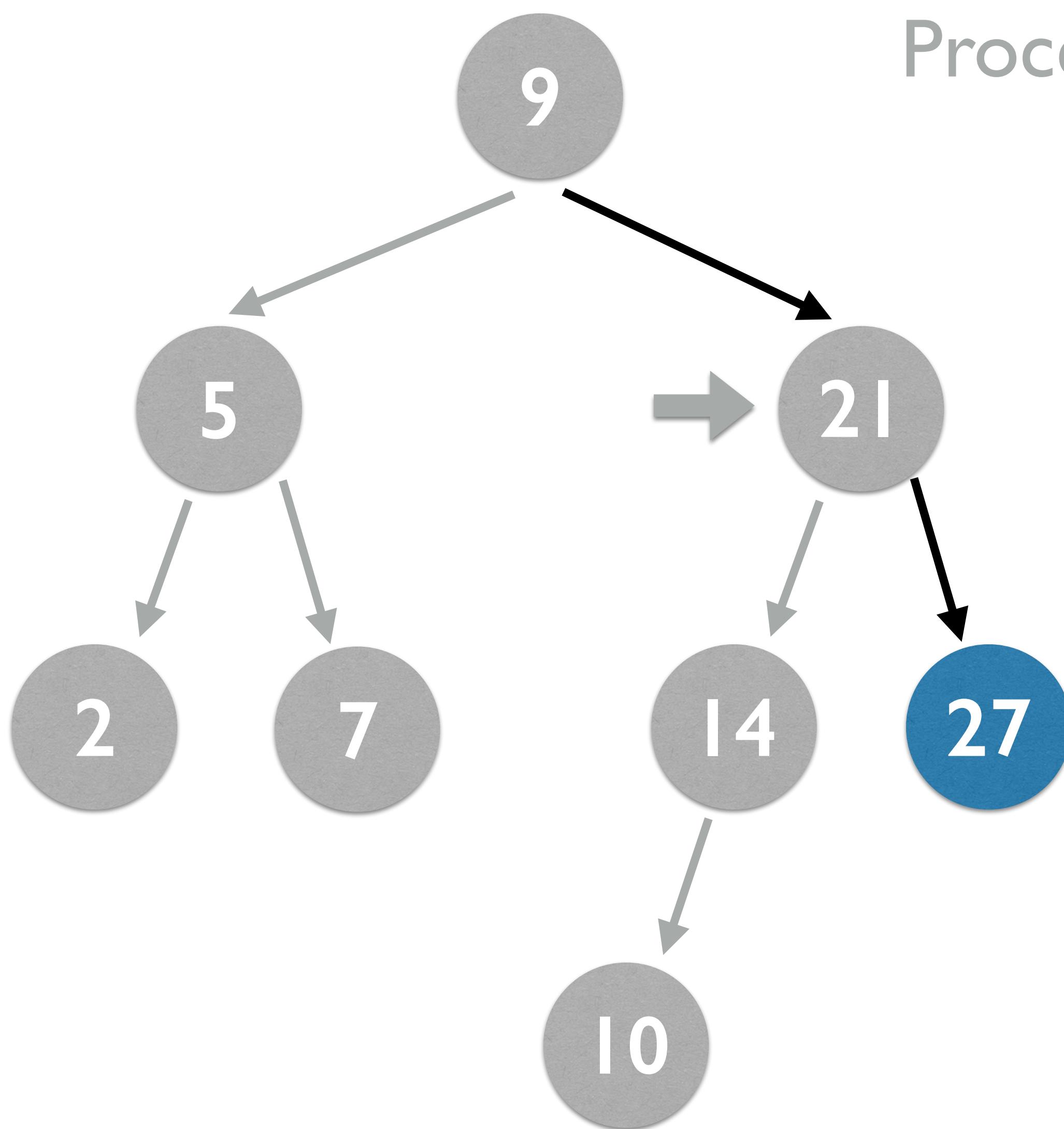
Depth First Traversal: Pre Order



Process root · Process left · Process right

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

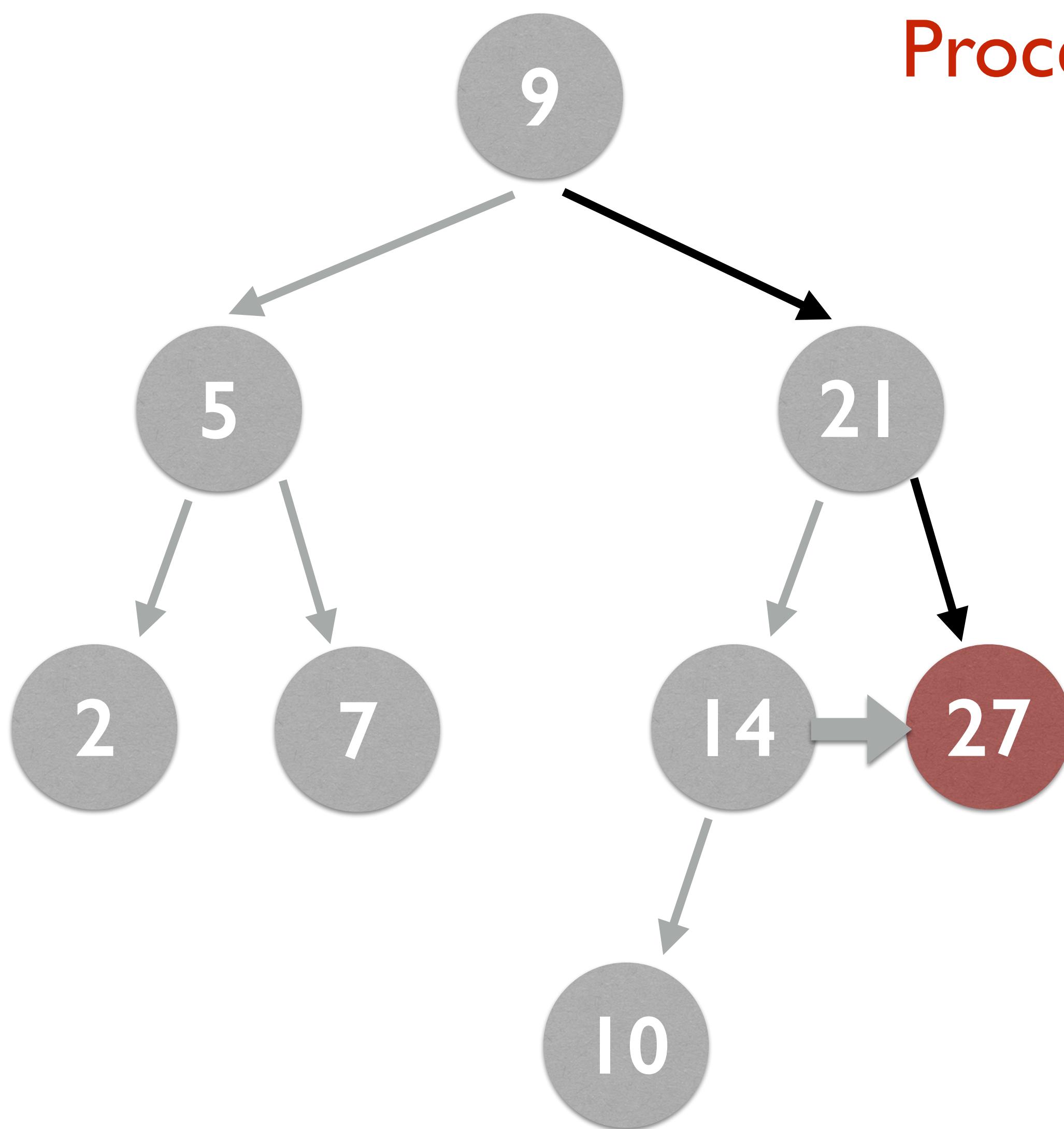
Depth First Traversal: Pre Order



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

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

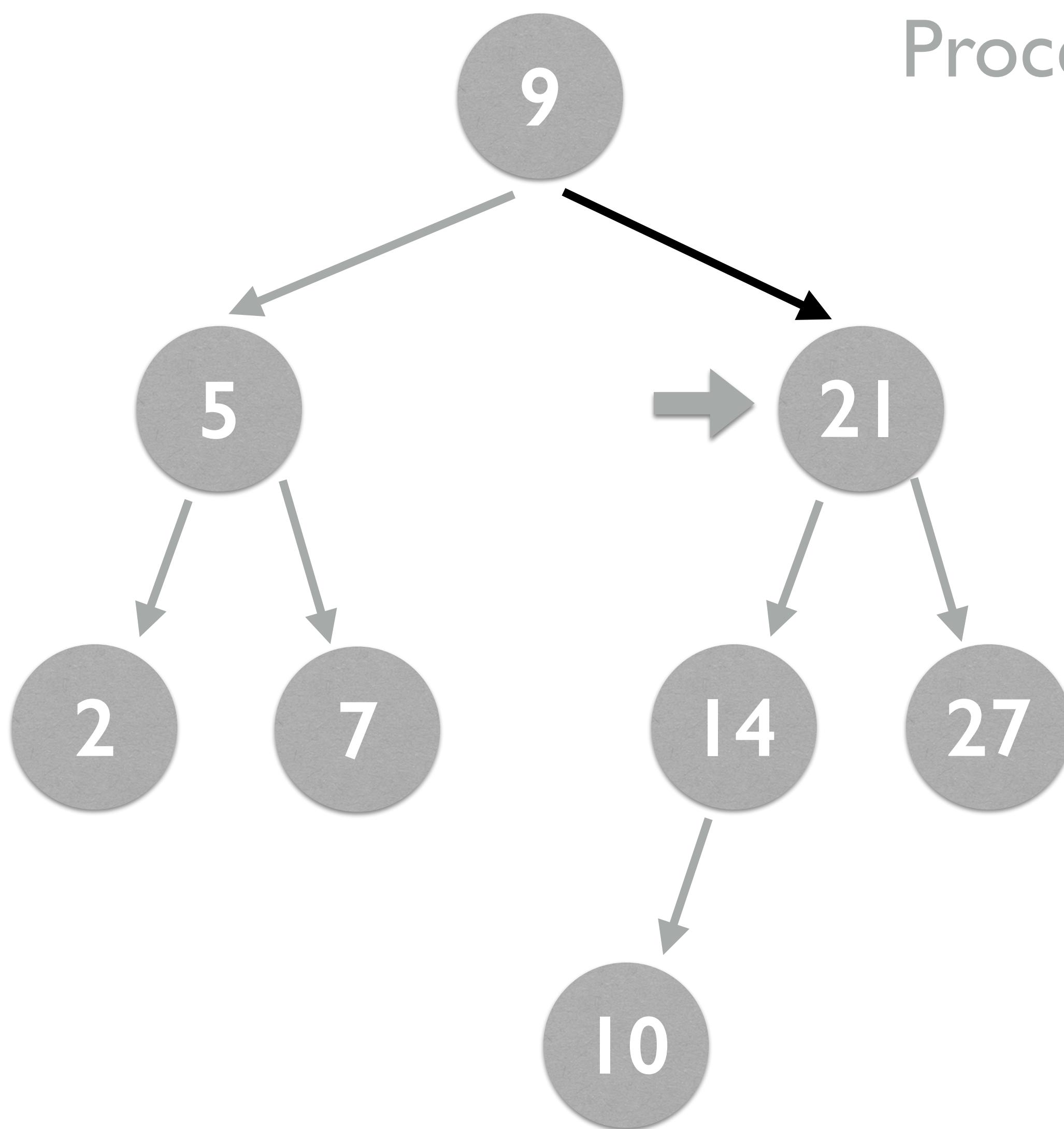
Depth First Traversal: Pre Order



Process root • Process left • Process right

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

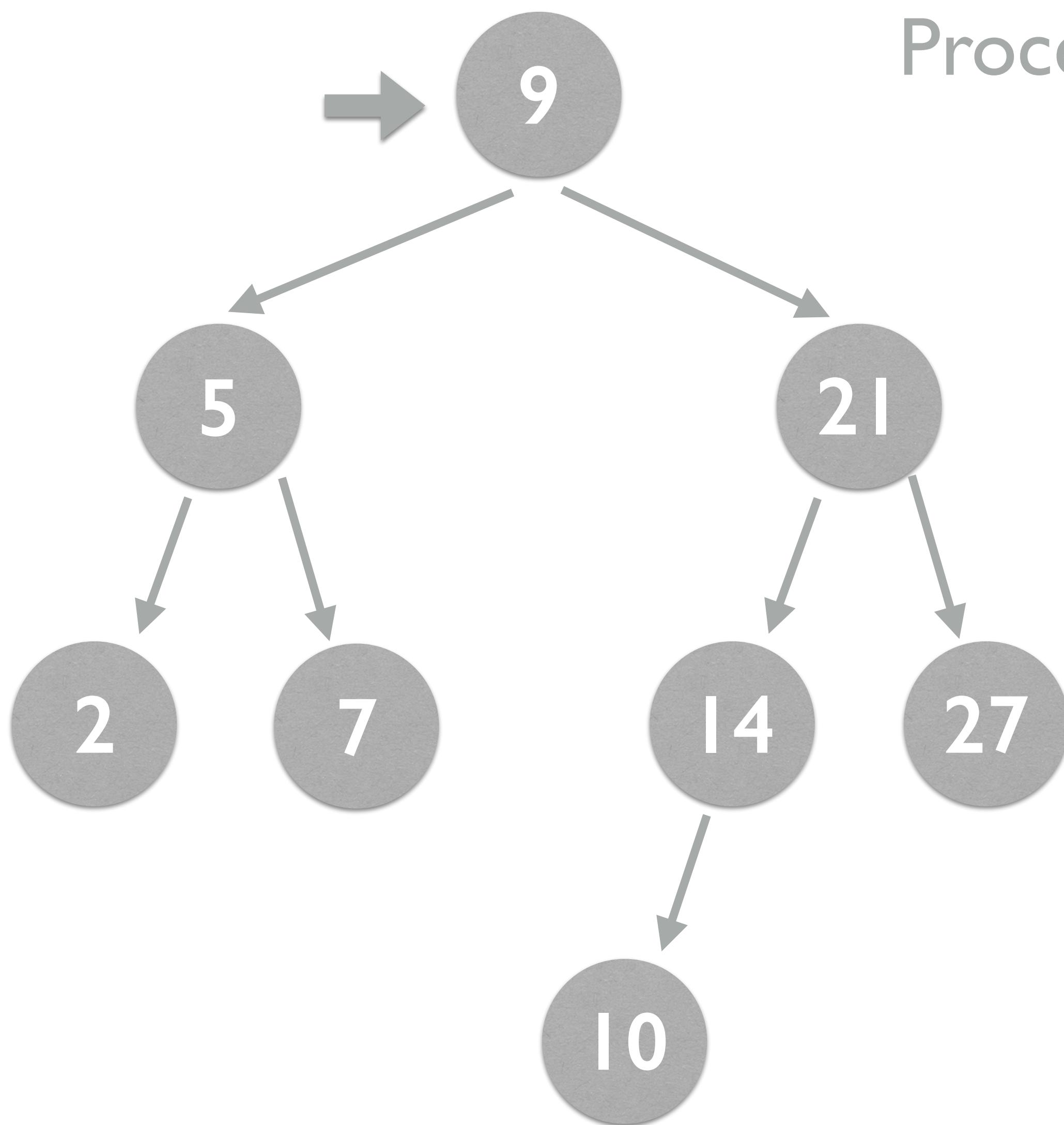
Depth First Traversal: Pre Order



Process root · Process left · Process right

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

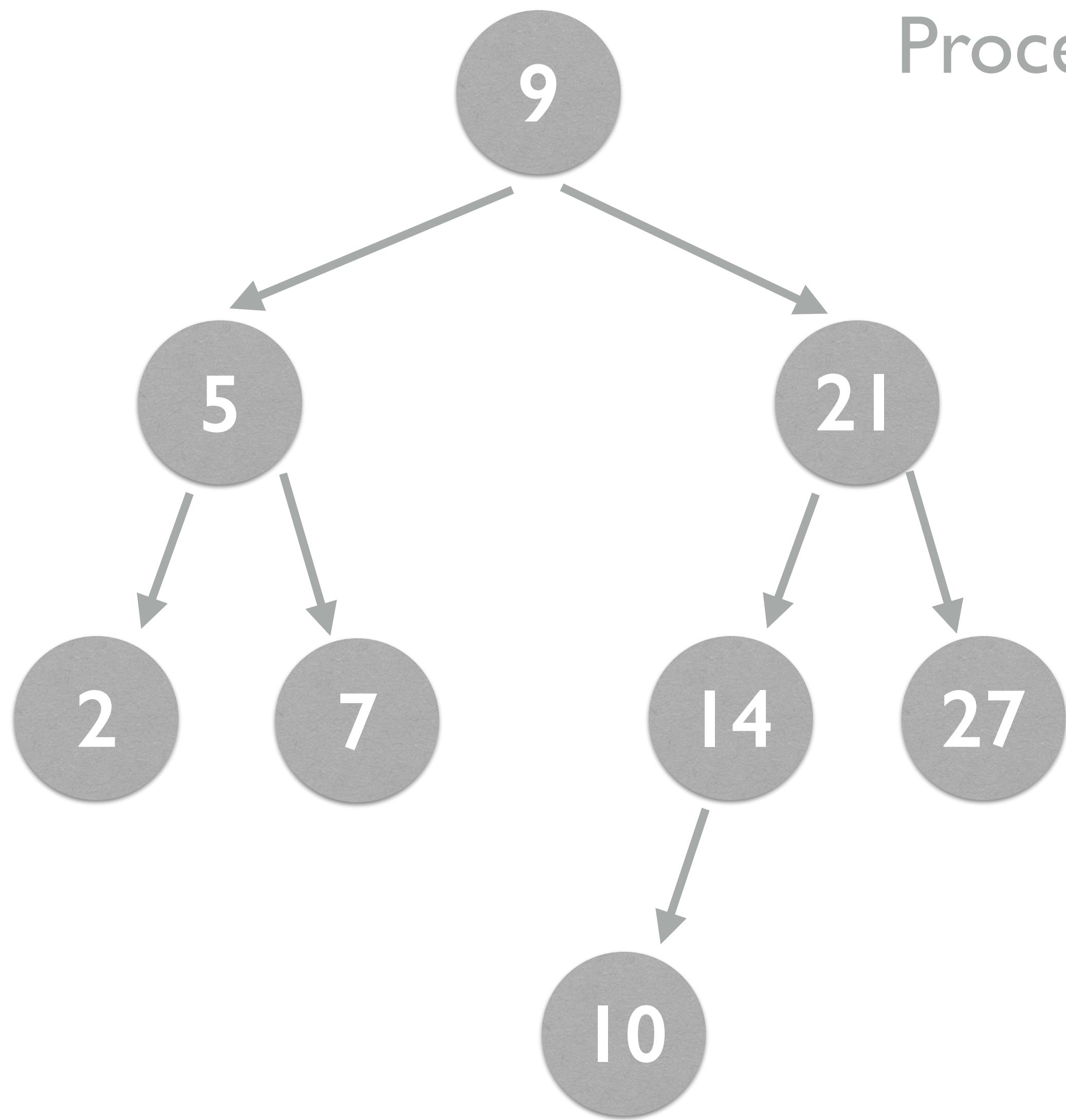
Depth First Traversal: Pre Order



Process root · Process left · Process right

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

Depth First Traversal: Pre Order



Process root · Process left · Process right

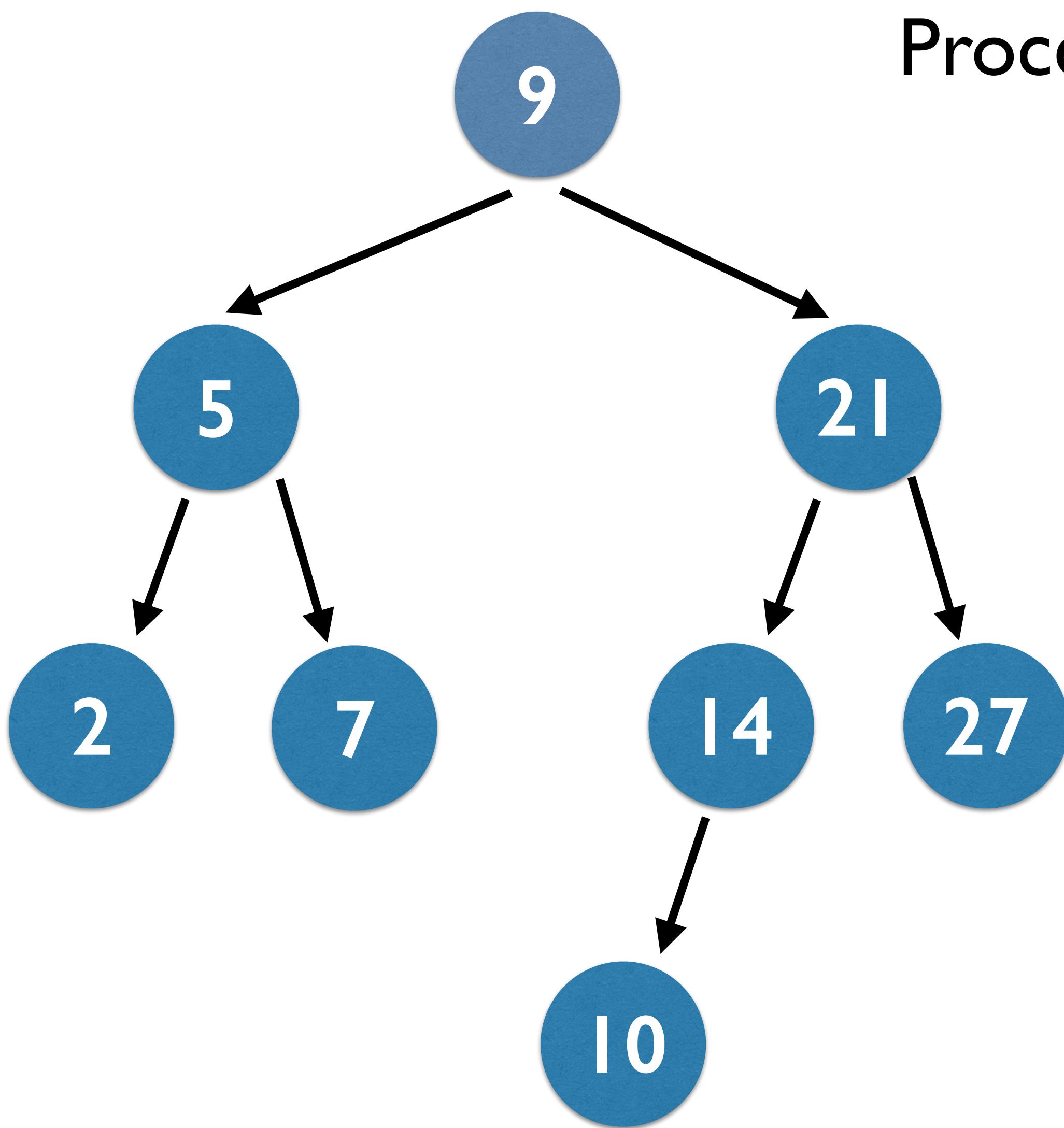
9, 5, 2, 7, 21, 14, 10, 27

- 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

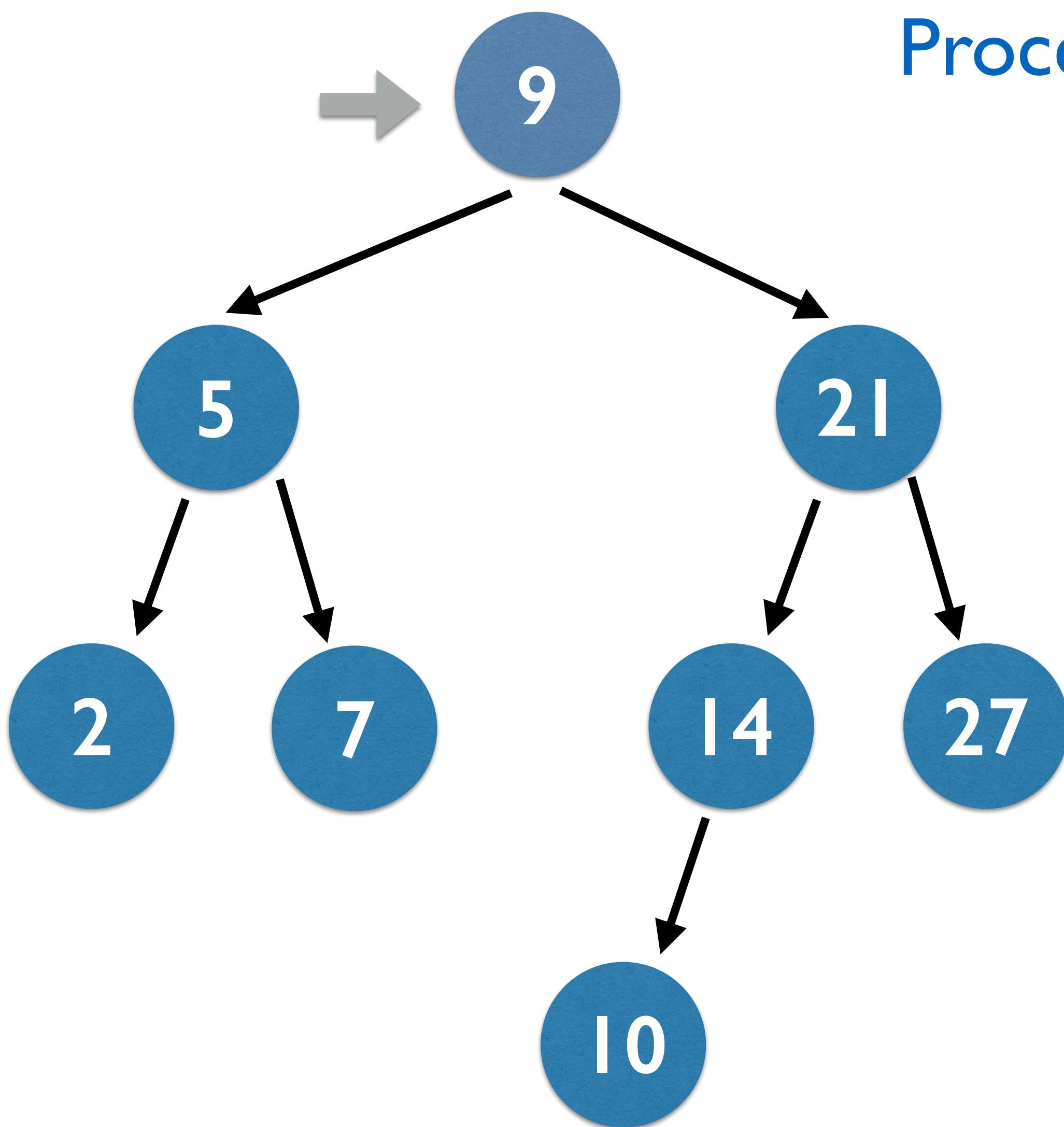
Depth First Traversal: Post Order

Process left · Process right · Process root



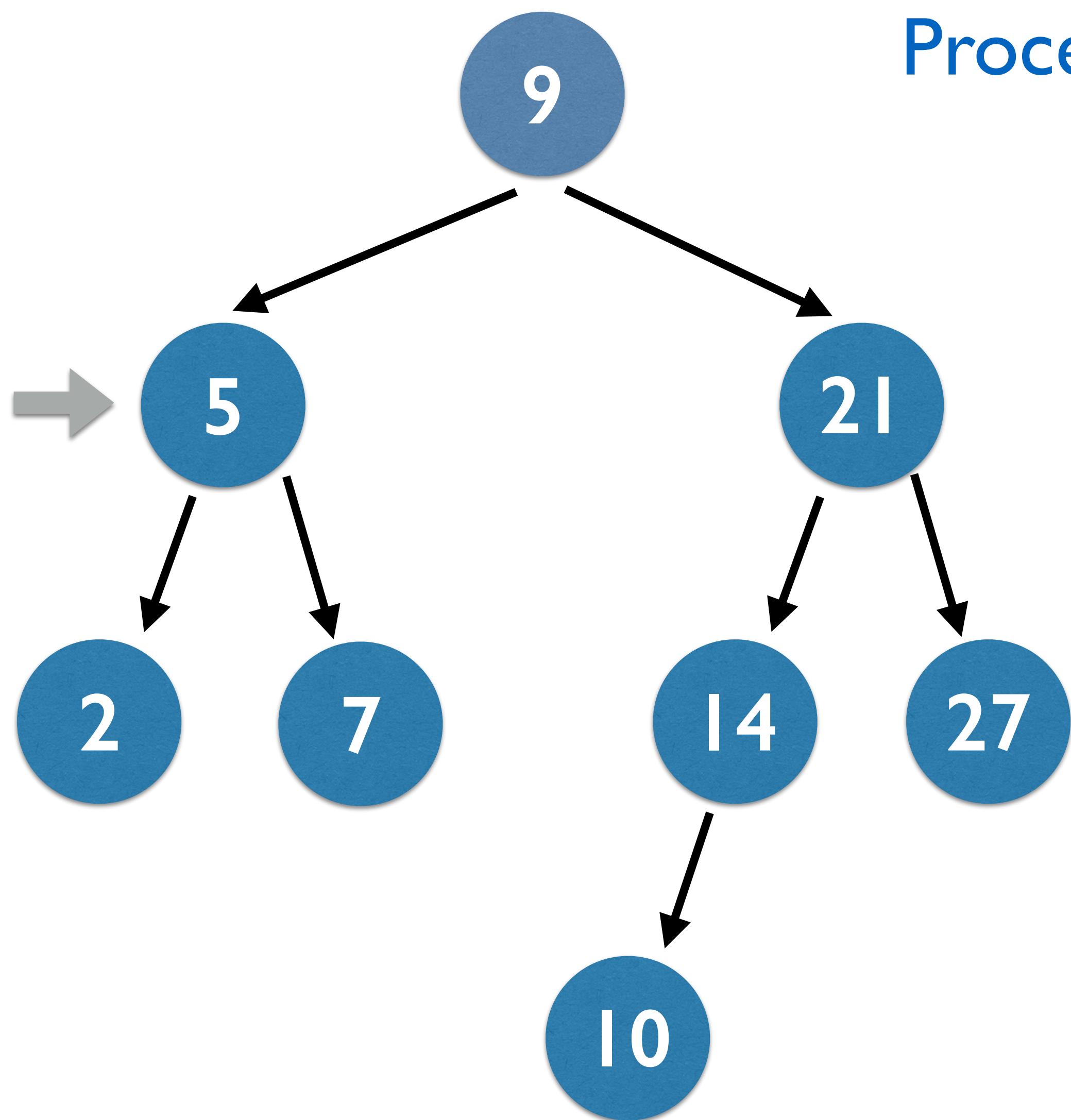
Depth First Traversal: Post Order

Process left · Process right · Process root

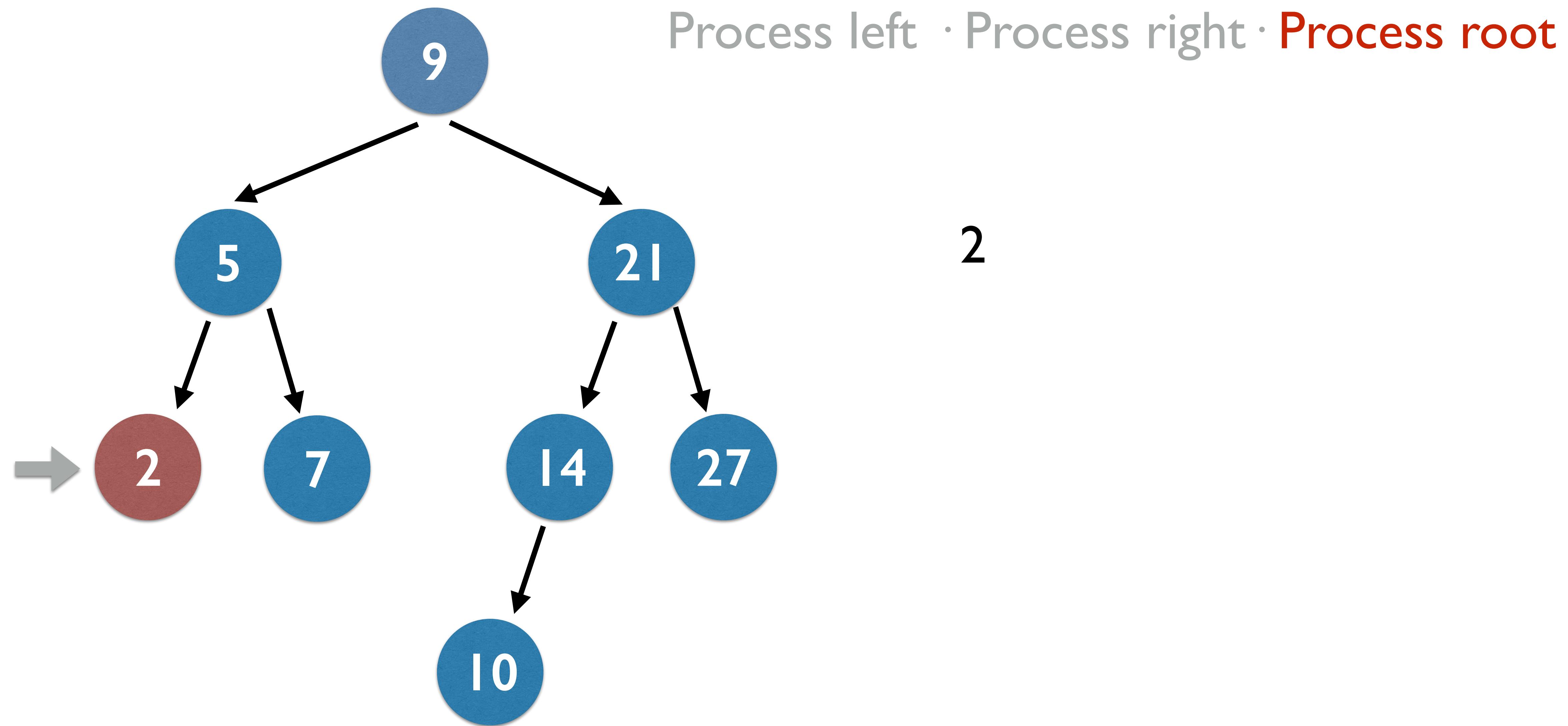


Depth First Traversal: Post Order

Process left · Process right · Process root

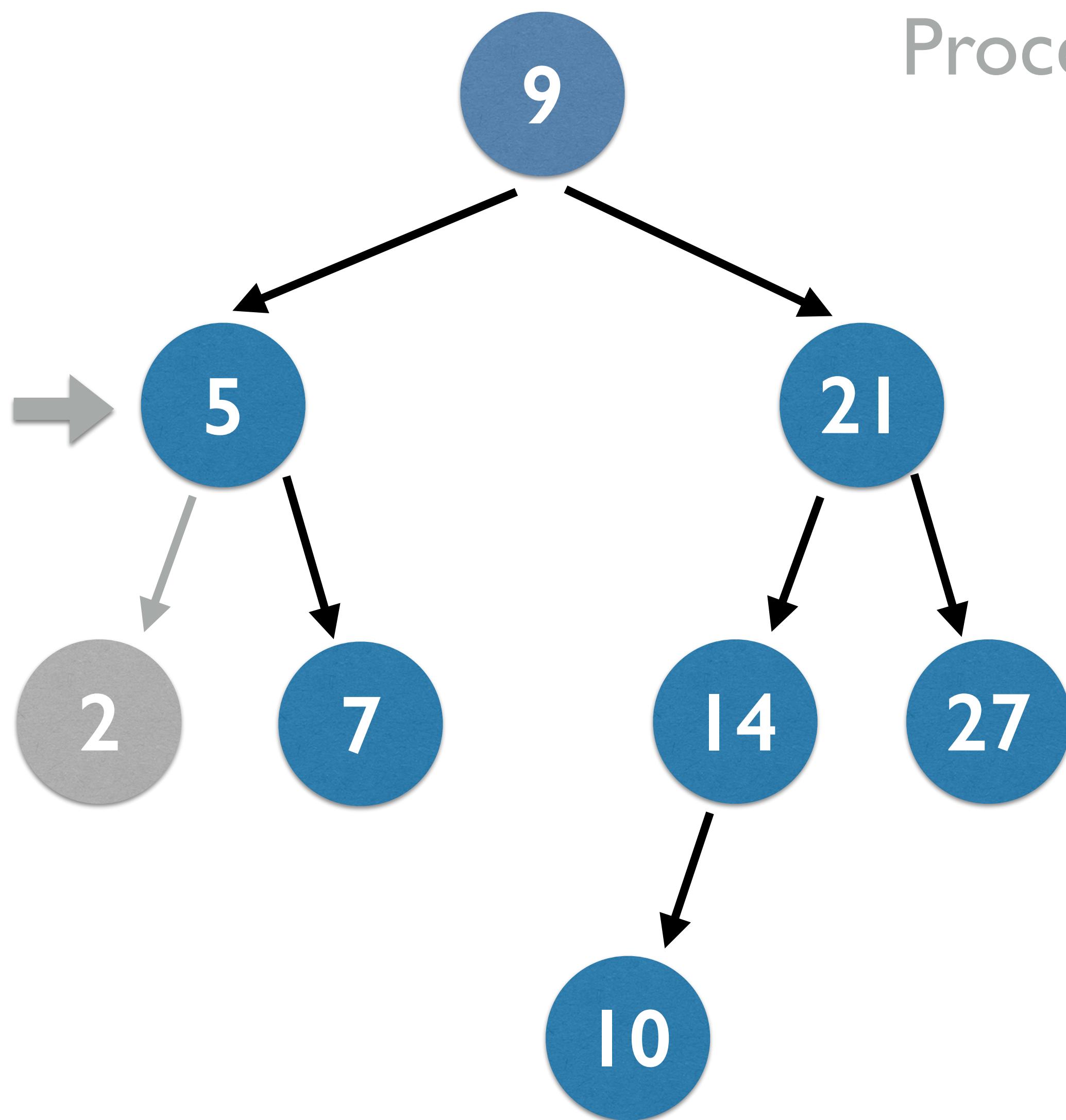


Depth First Traversal: Post Order



Depth First Traversal: Post Order

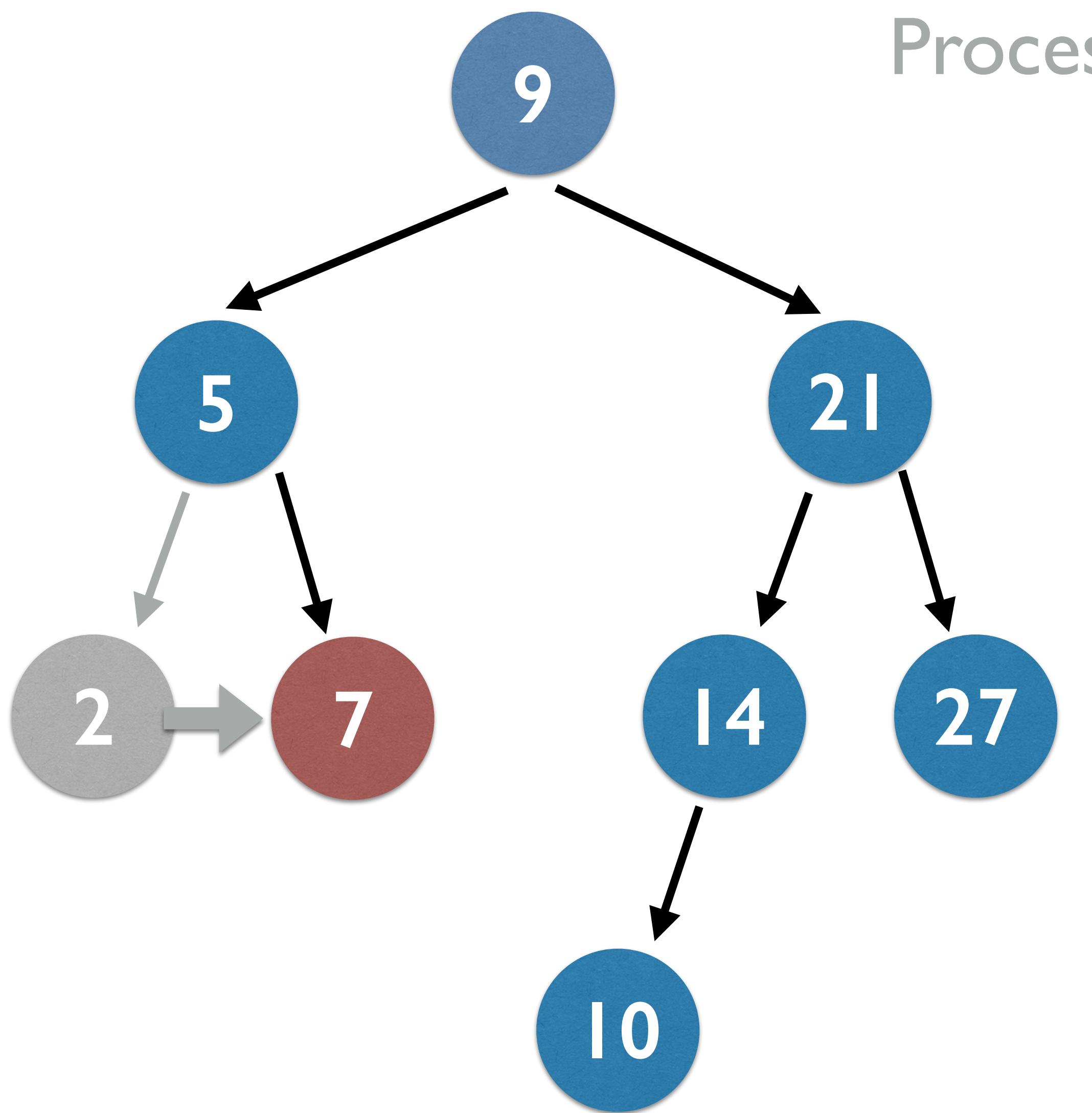
Process left • Process right • Process root



2

Depth First Traversal: Post Order

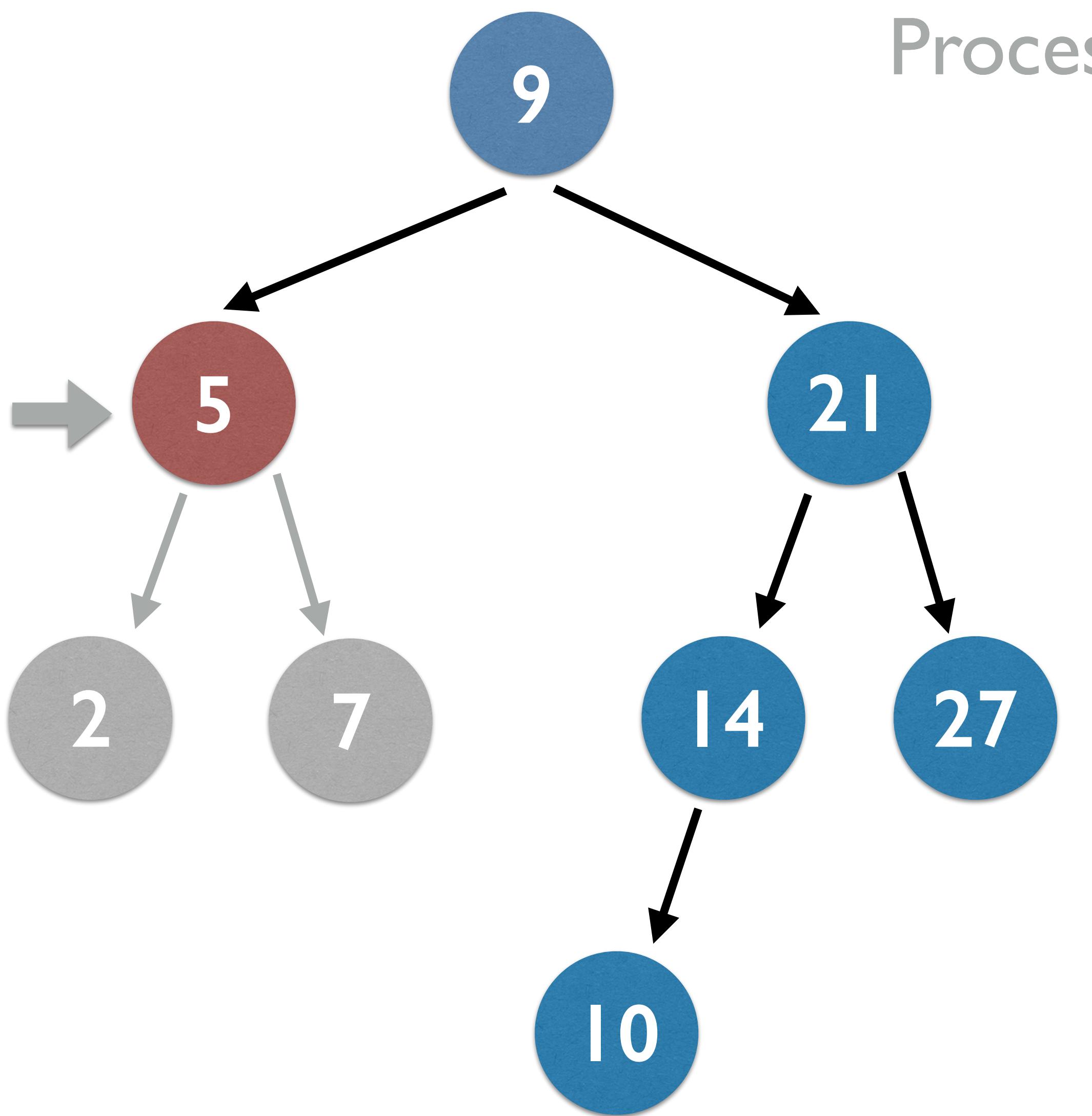
Process left · Process right · **Process root**



2, 7

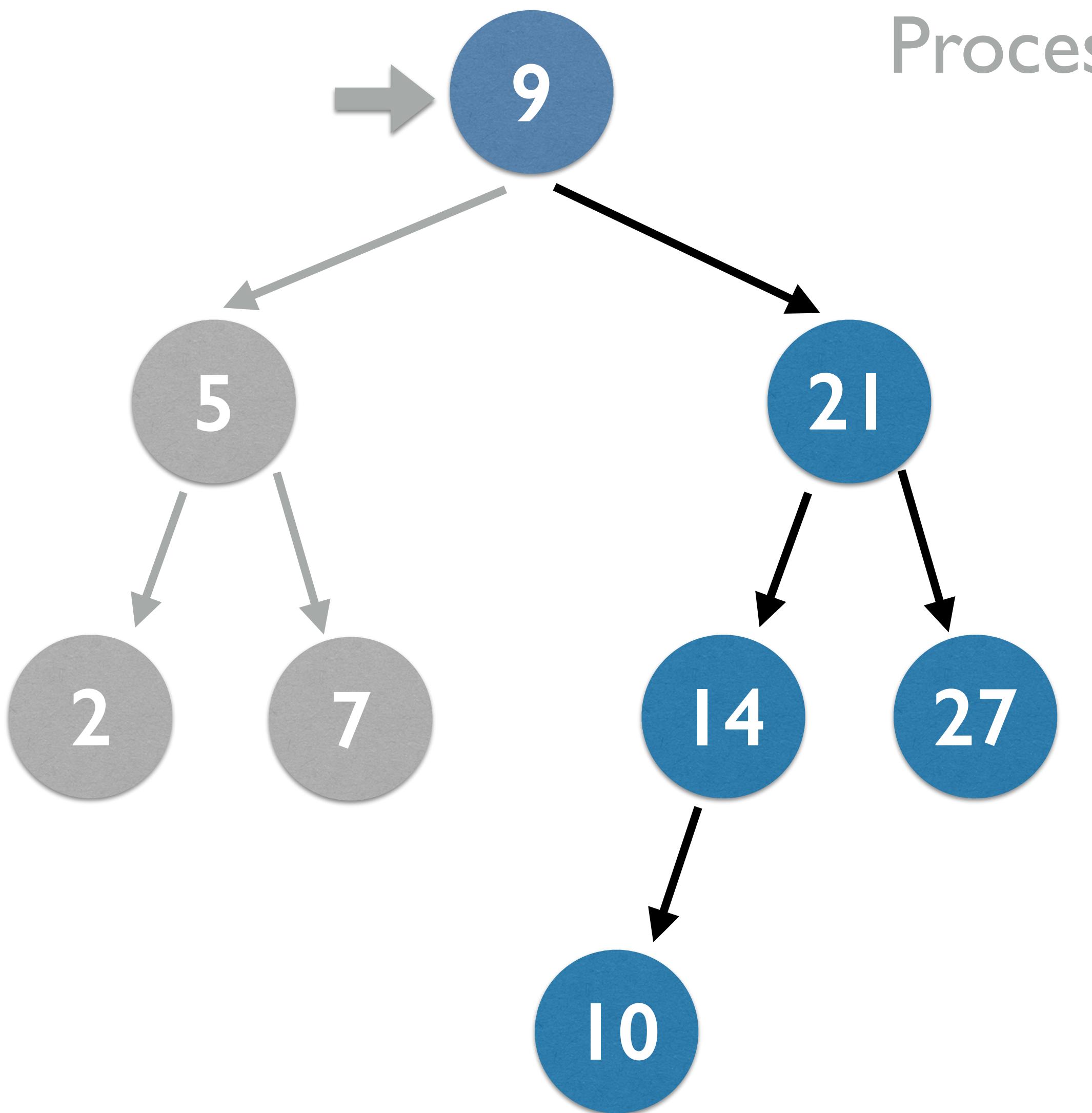
Depth First Traversal: Post Order

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



2, 7, 5

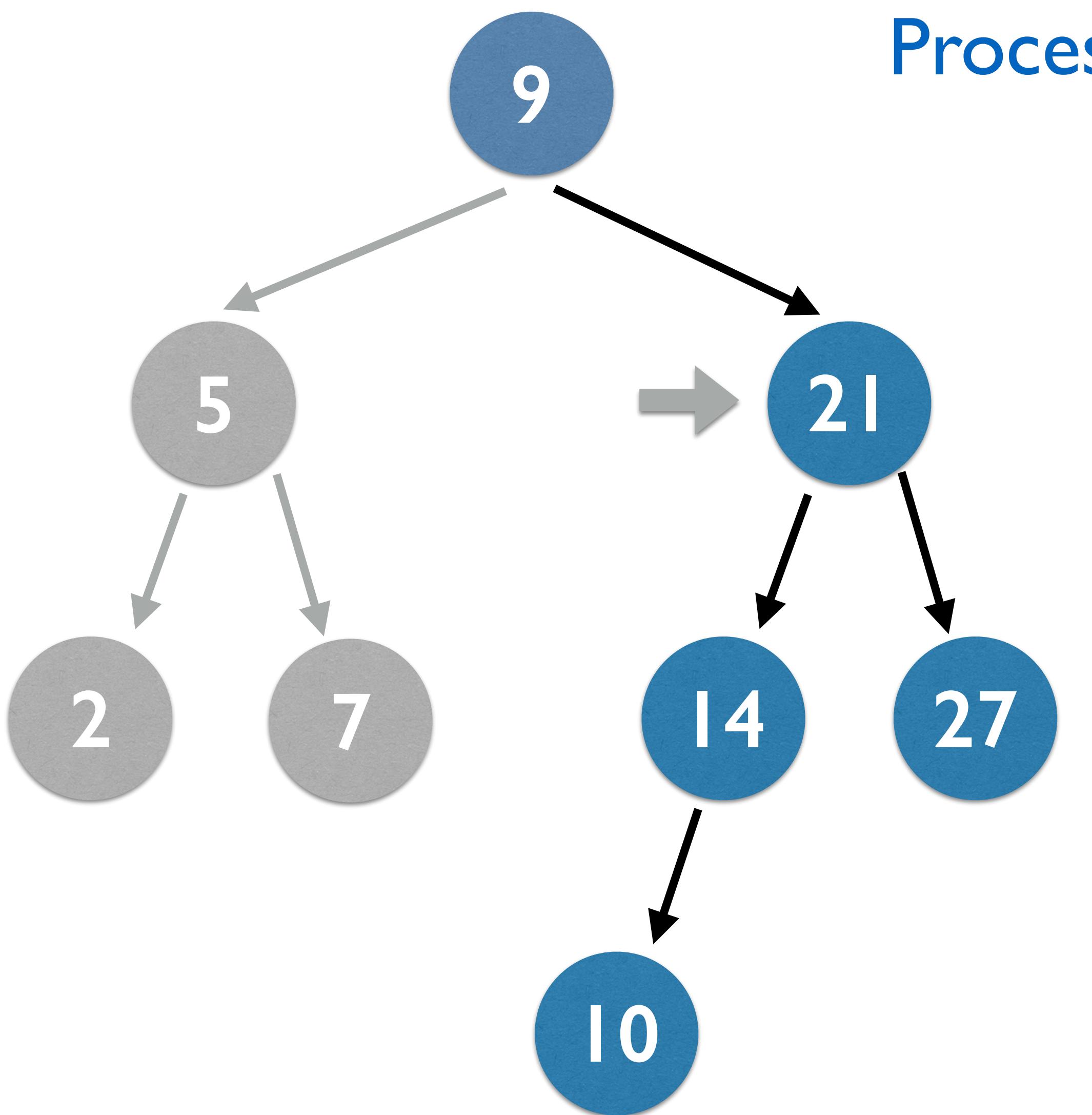
Depth First Traversal: Post Order



Process left • Process right • Process root

2, 7, 5

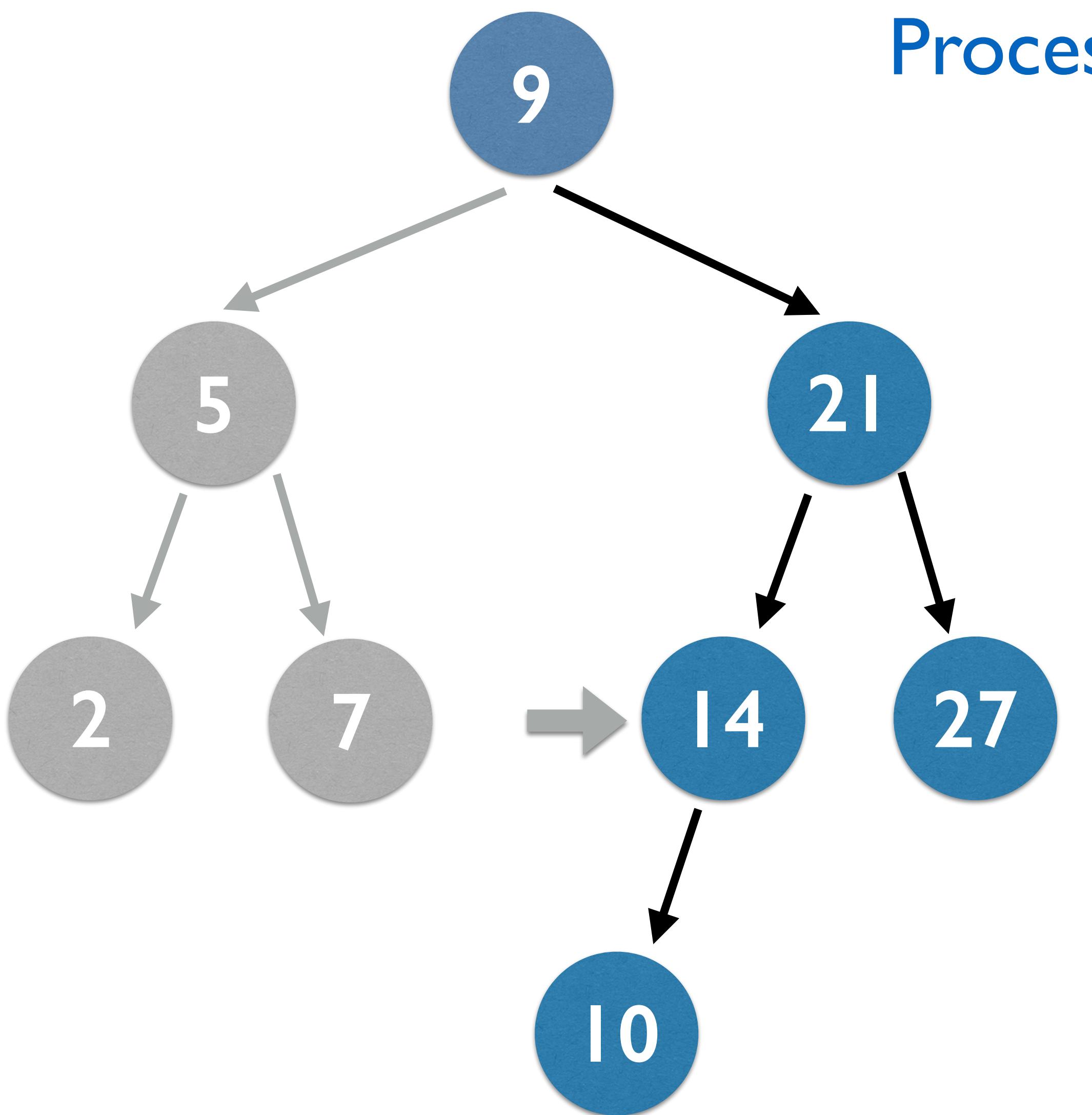
Depth First Traversal: Post Order



Process left · Process right · Process root

2, 7, 5

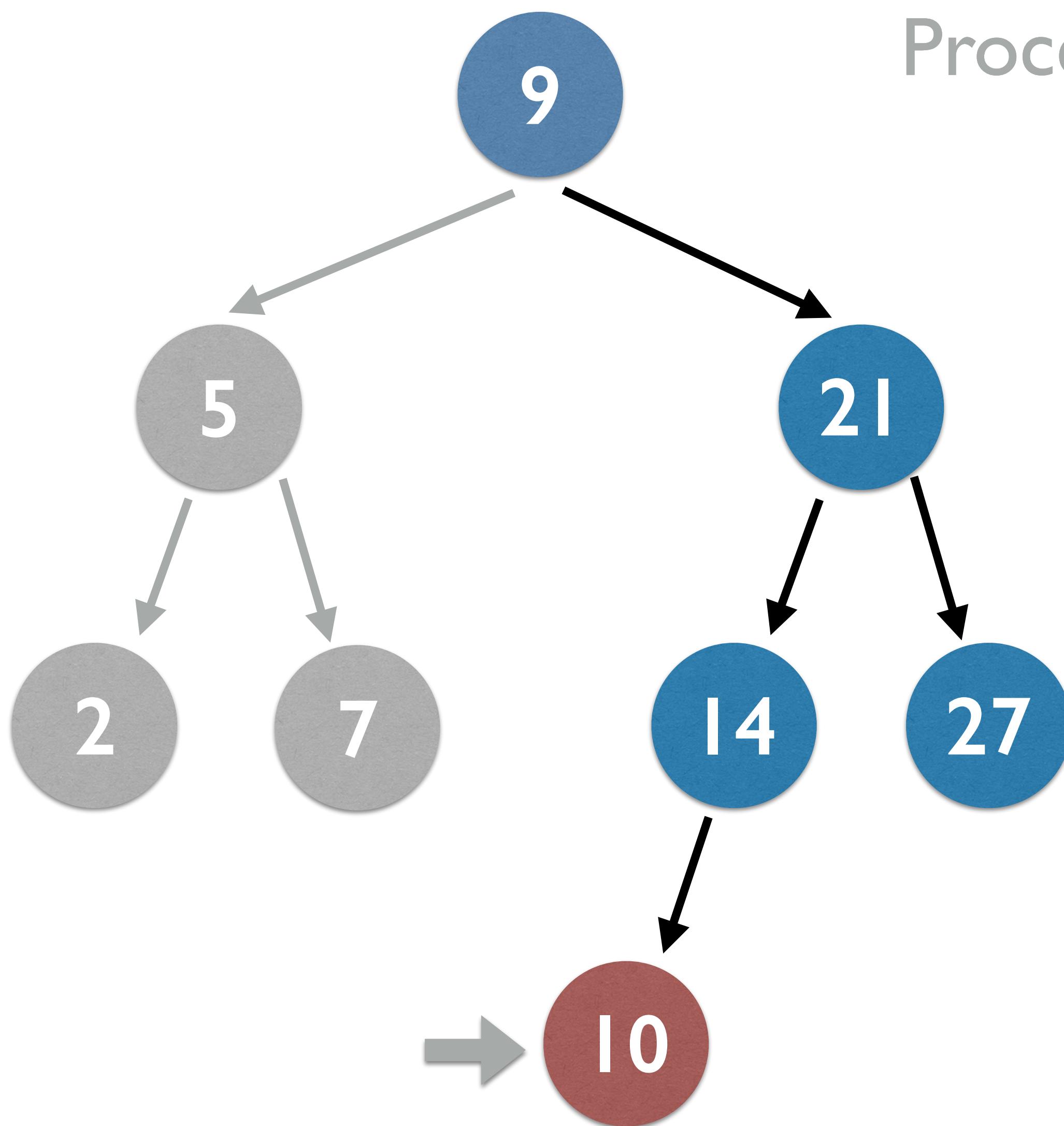
Depth First Traversal: Post Order



Process left · Process right · Process root

2, 7, 5

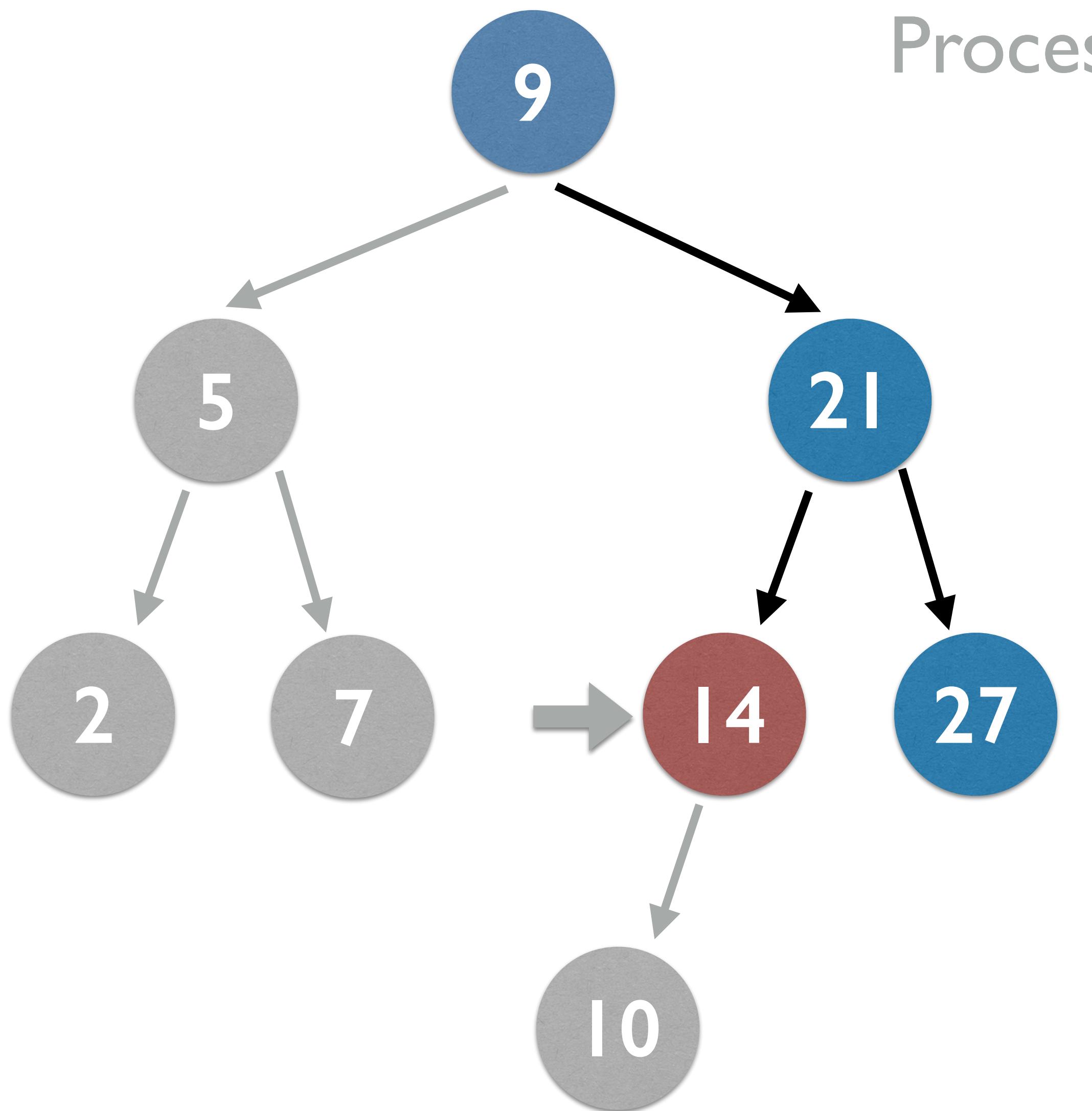
Depth First Traversal: Post Order



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

2, 7, 5, 10

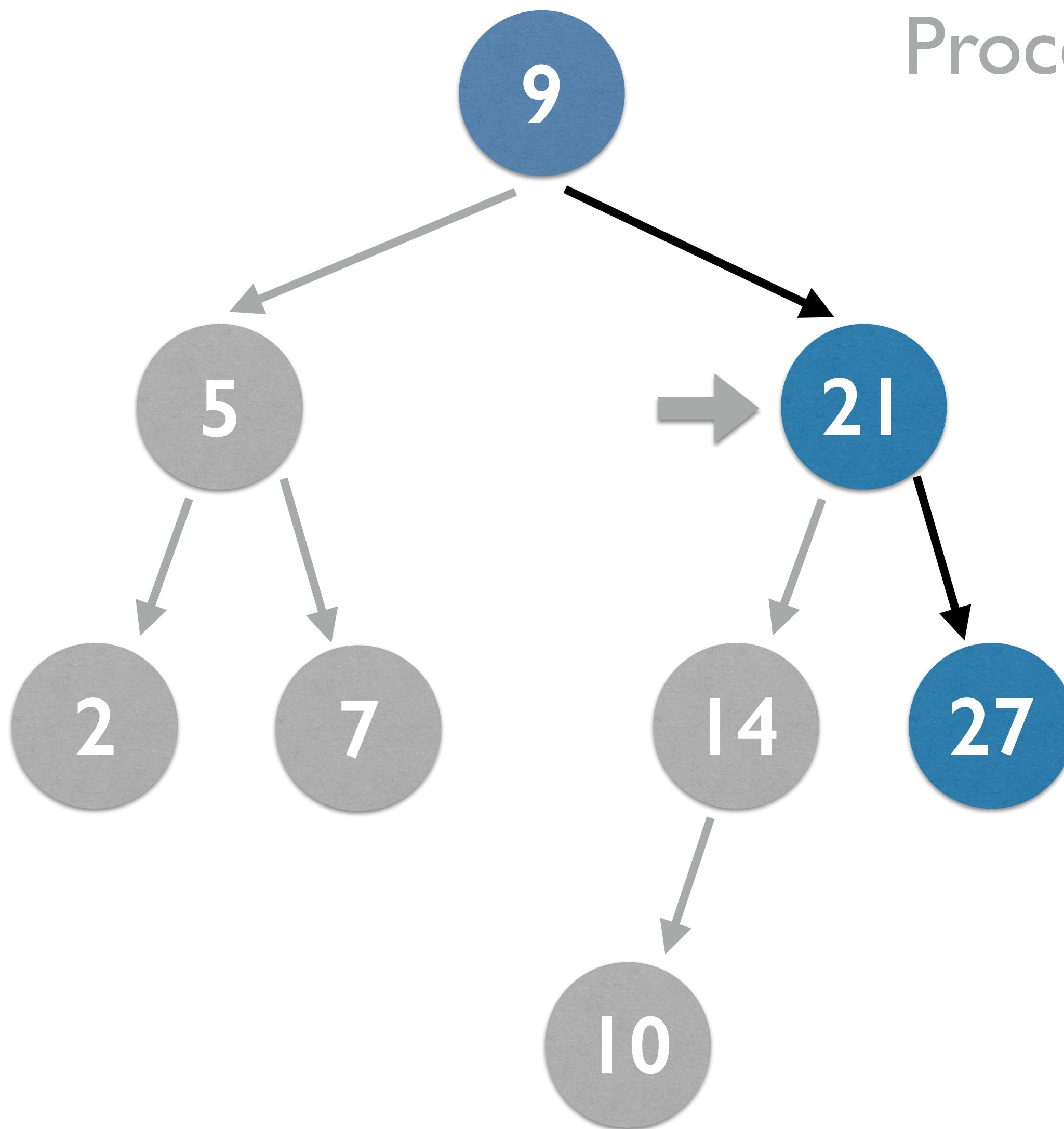
Depth First Traversal: Post Order



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

2, 7, 5, 10, 14

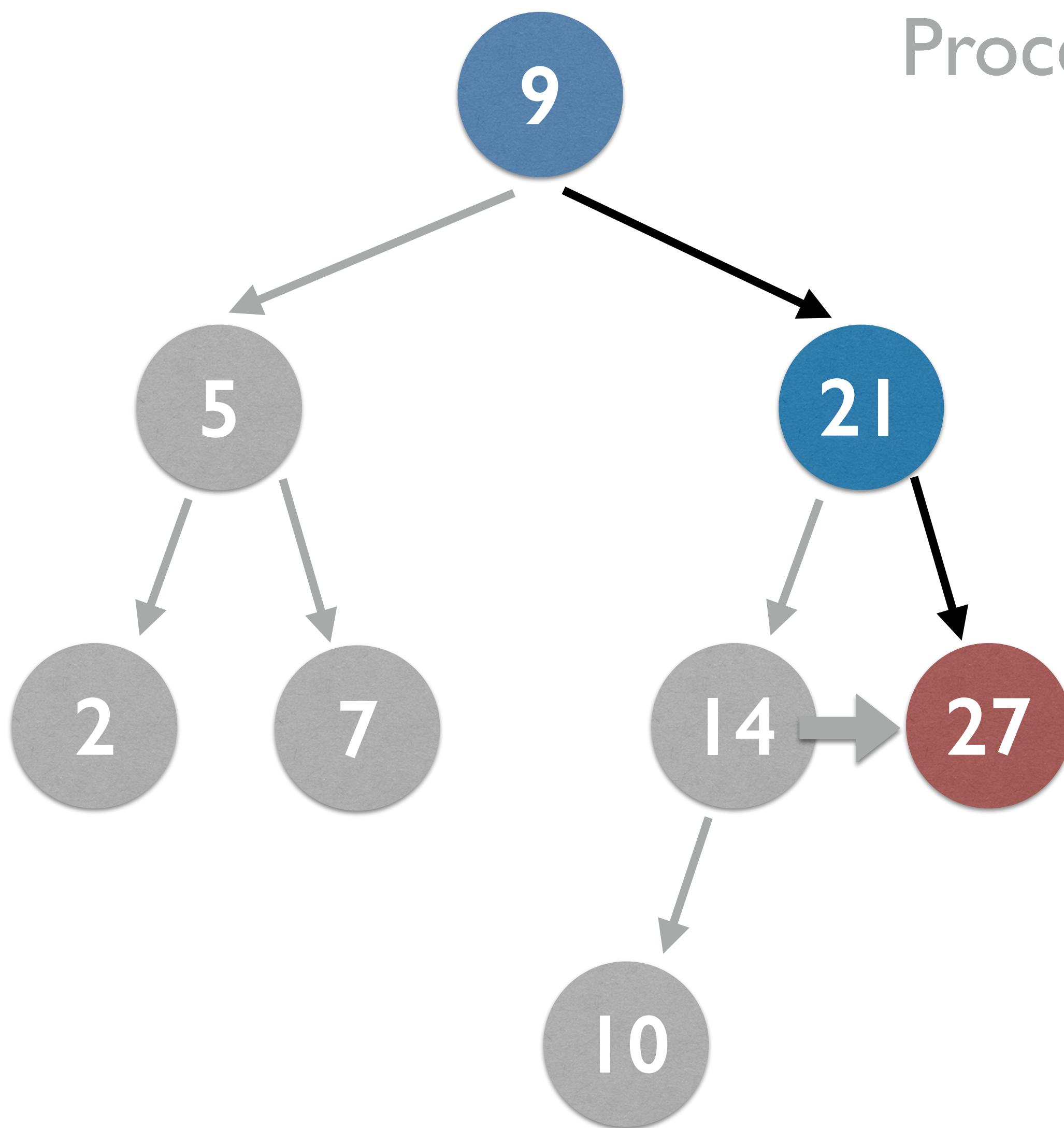
Depth First Traversal: Post Order



Process left • Process right • Process root

2, 7, 5, 10, 14

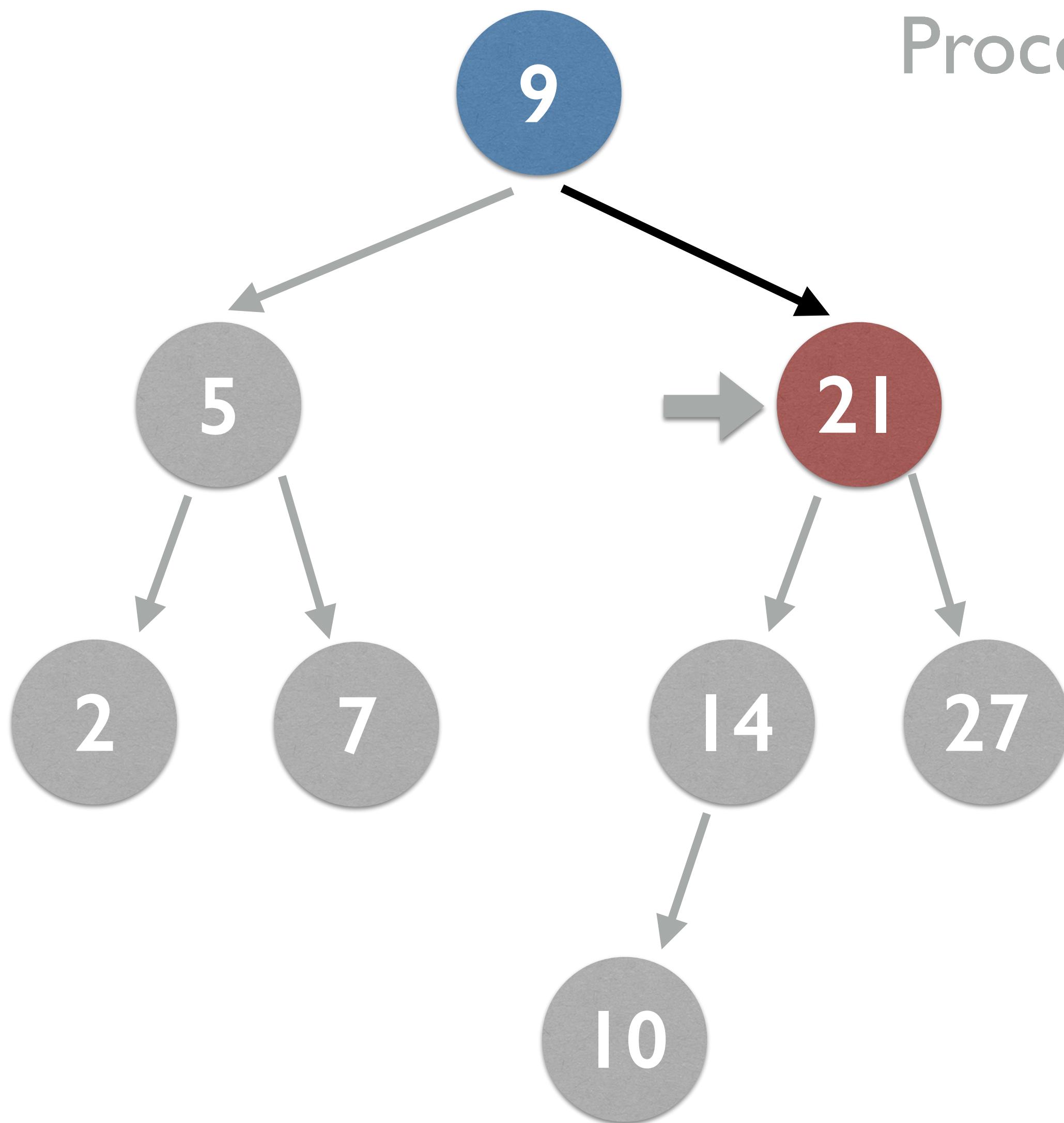
Depth First Traversal: Post Order



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

2, 7, 5, 10, 14, 27

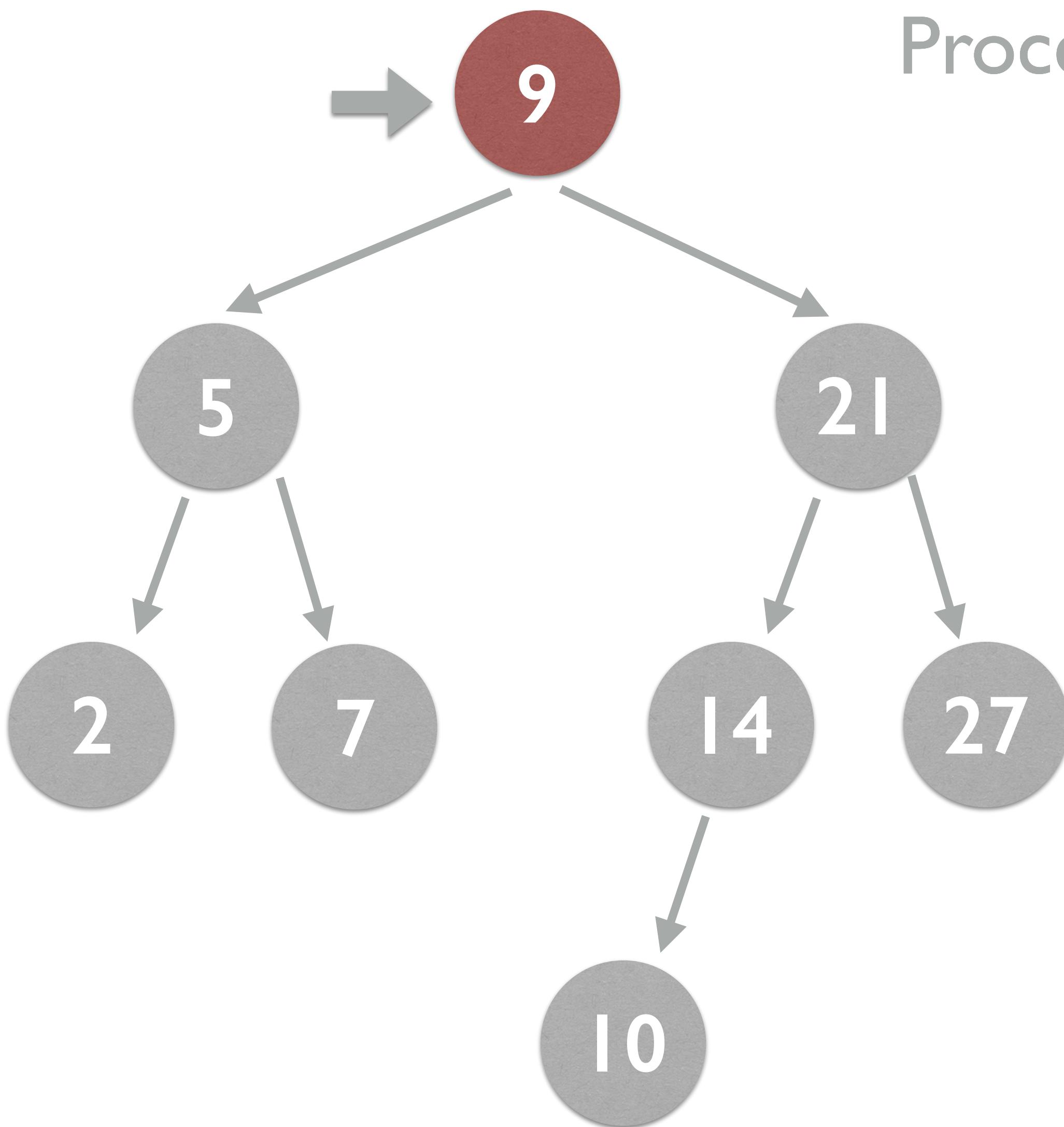
Depth First Traversal: Post Order



Process left • Process right • **Process root**

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

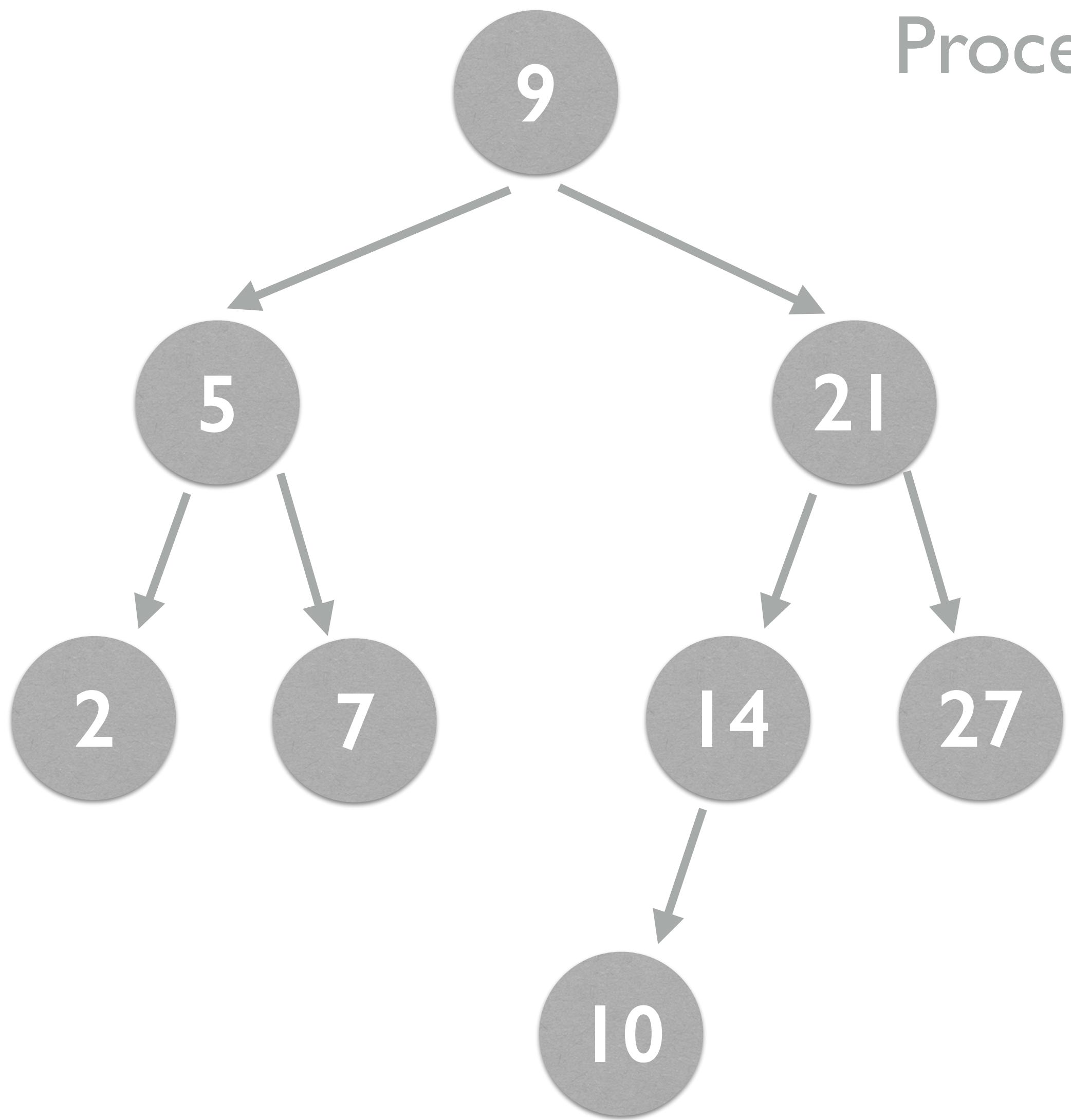
Depth First Traversal: Post Order



Process left • Process right • **Process root**

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

Depth First Traversal: Post Order



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 one way or the other
- Binary Search Trees are a recursive data structure - it's much easier to write them if you employ recursive problem solving



1. Identify simplest possible input

```
function factorial (n) {  
}  
}
```

2. Solve just for the simplest input

```
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)  
  }  
}
```



4. Generalize in terms of input

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

WORKSHOP

