

Data Types & Structures

Part II: Hash Tables and Trees

JOHANNES NOLLMAYER

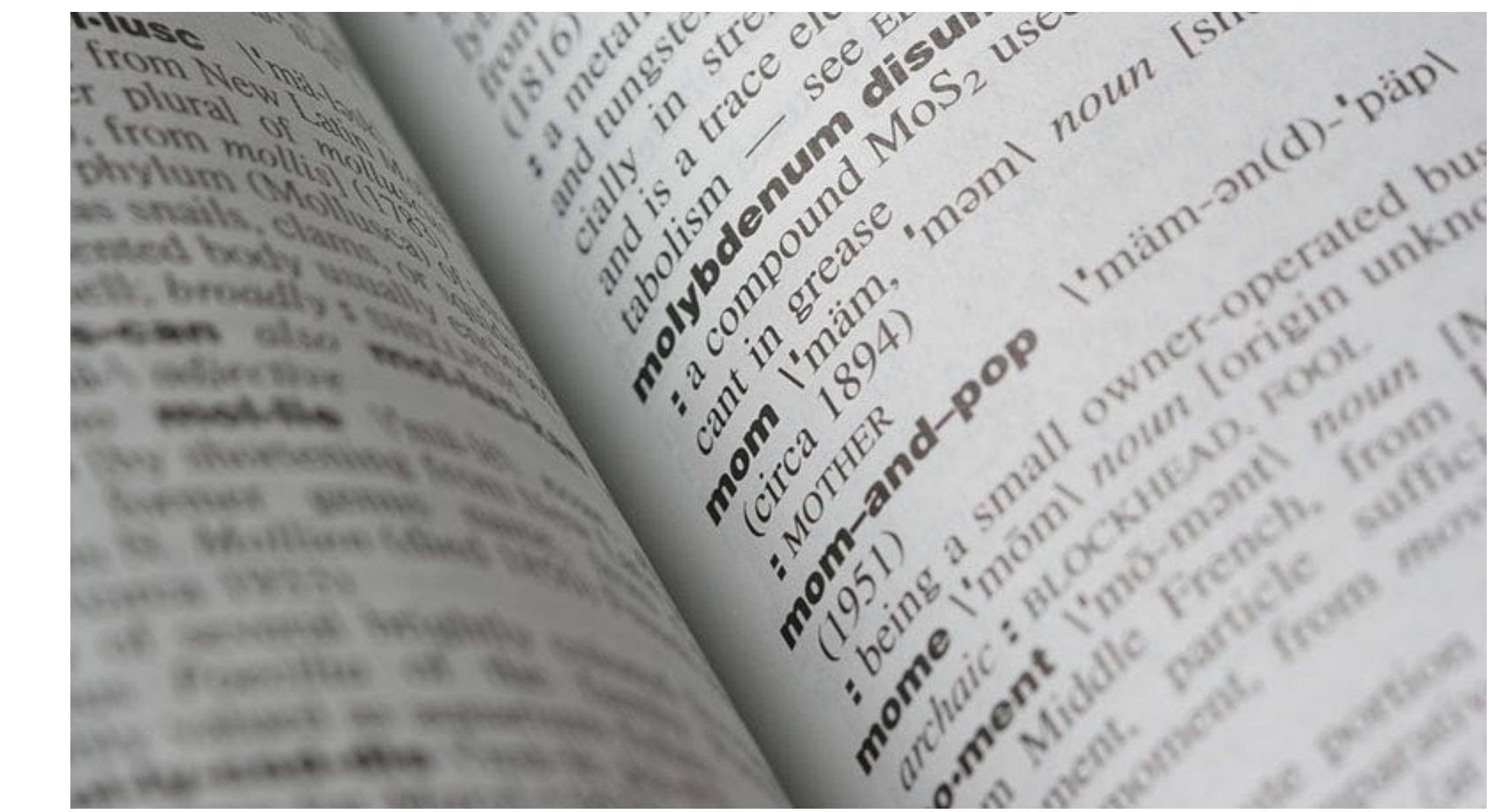
— dictionary /'dɪkʃənəri/ n. (P
book listing (usu. alphabetica
explaining the words of a lan
giving corresponding words in
language

Dictionary / Map



The Dictionary ADT

- AKA *Associative Array, Map, or Symbol Table*
- Stores **key-value** pairs (e.g. "location" : "NY")
- **Unique keys**
- **Modify & lookup**
 - Set value for key
 - Get value for key
- **Dynamic alteration**
 - Add new pairs
 - Delete existing pairs



How to implement this ADT?

The classic data structure: a Hash Table

- High-concept: an **array** to hold **values**, and a **hash function** that transforms a string **key** into a numerical **index**
- Example of a very simple hash function:

```
function hash (keyString) {  
  var hashed = 0;  
  for (var i = 0; i < keyString.length; i++) {  
    hashed += keyString.charCodeAt(i);  
  }  
  return hashed % 9; // number of spaces in array  
}
```

```
contact.name = 'Grace'  
contact.phone = 8675309
```

Example with a 9-bucket array

1. We want to store the value '**Grace**' for the key '**name**'.
2. Hashing the string '**name**' yields the numerical index **3**.
3. Store '**Grace**' at index **3**.
4. Now store the value **8675309** for key '**phone**'
5. '**phone**' hashes to **7**
6. Store **8675309** at index **7**

Index	Data
0	
1	
2	
3	Grace
4	
5	
6	
7	8675309
8	

Fetching/changing values

1. Q: what is the value for the key '**name**'?
2. Hashing the string '**name**' yields the numerical index **3**.
3. Find the value at index **3**.
4. Result: '**Grace**'
5. Similar steps for checking the key '**phone**', gets the value **8675309**.

Index	Data
0	
1	
2	
3	Grace
4	
5	
6	
7	8675309
8	

Anyone see a problem?

Collisions

1. Now we want to store the value **grace@gmail.com** for the key **email**.
2. Hashing **email** yields the numerical index **7**.
3. But we **already have** a value there (for **phone**)!
4. Because there are many more possible keys than buckets, collisions are inevitable.

Index	Data
0	
1	
2	
3	Grace
4	
5	
6	
7	8675309
8	

How to resolve collisions?

Two main strategies

- **Open addressing:** if a bucket is full, find the next empty bucket. Place the value in that spot instead of the original.
- **Separate chaining:** every bucket stores a secondary data structure, like a linked list. Collisions create new entries in that data structure.

Separate chaining: adding a value

1. We want to store the value **grace@gmail.com** for the key **email**.
2. Hashing **email** yields the numerical index **7**.
3. That bucket already contains **8675309**.
4. Add the value to the Linked List as a new **node**.

Index	Linked List in bucket
0	
1	
2	
3	<Grace>
4	
5	
6	
7	<8675309> → <grace@gmail.com>
8	

We still have a problem...

Separate chaining: retrieving a value

1. What is the value for the key **email**?
2. Hashing **email** yields the numerical index **7**.
3. In bucket **7** there is a linked list.
4. There are **two nodes** in that list; how do we know which value is for the key **email**?

Index	Linked List in bucket
0	
1	
2	
3	<Grace>
4	
5	
6	?
7	?
8	

Separate chaining: store value and key

1. Hashing **email** yields the numerical index **7**.
2. In bucket **7** there is a linked list.
3. The **head** node has a key of **phone**, so we need to keep looking.
4. The **next** node has a key **email**, so we return the value: **grace@gmail.com**

Index	Linked List in bucket
0	
1	
2	
3	<name, Grace>
4	
5	
6	X
7	<phone, 8675309> → <email, grace@gmail.com>
8	

**So... how is a hash table better
than one big linked list?**

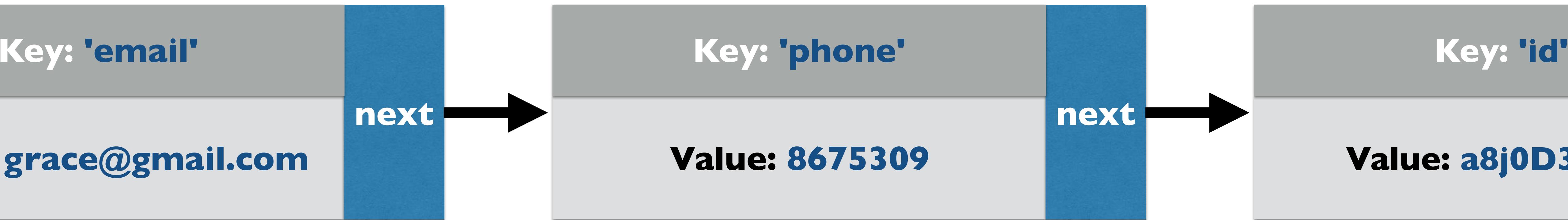
Performance

- Assume many buckets and a good hashing function
- *Usually*: assign or check pair takes just 1 step (hash invocation)
- *Sometimes*: collisions occur
 - Traverse a few nodes of a linked list; but **just a few** (still pretty fast)
 - Way better than having to traverse **all the data** in the entire structure!



List nodes with key-value pairs — how?

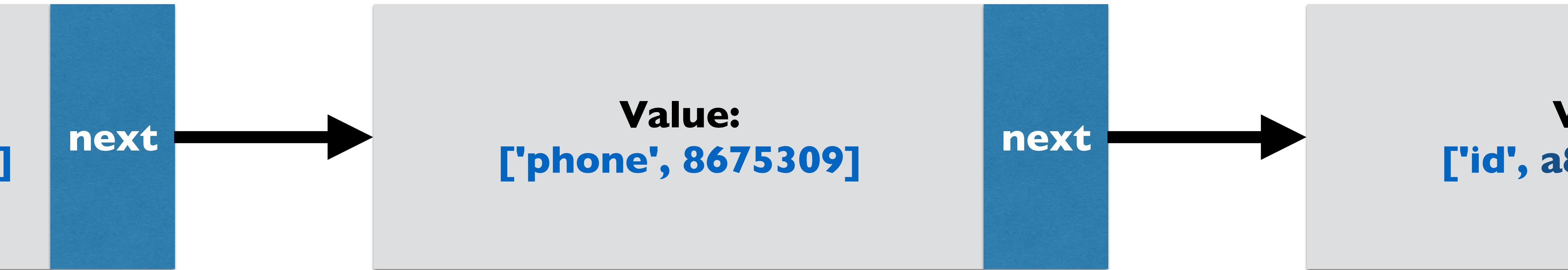
Solution 1: Association List



- Nodes have key as well as **value** & **next**
 - Advantage: best for purpose, most straightforward.
 - Downside: need a custom LL implementation

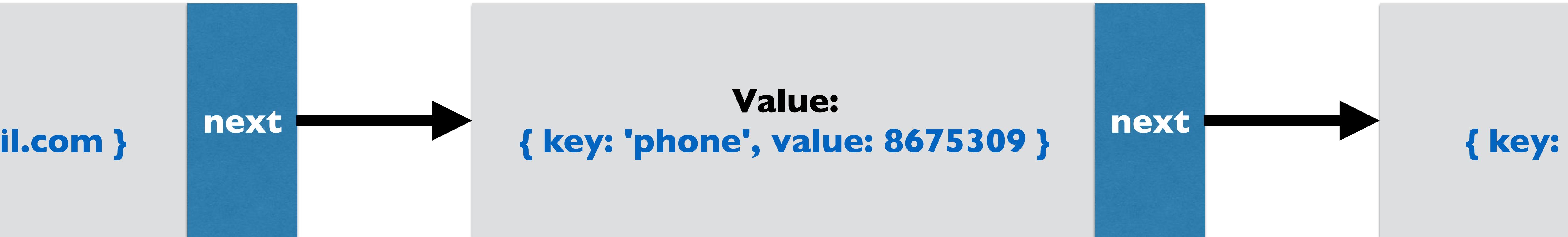
Solution 2: store an array

e:
@gmail.com]



- LL node value is an array, index 0 stores HT key & 1 stores HT value
 - Downside: referring to indices 0 and 1 isn't very descriptive / easy to read

Solution 3: store a struct



- LL node value is itself a data structure with **key** & **value** prop.s
 - Seems like cheating, but you can hand-wave this by pretending we are using "structs" — pre-defined memory structures that cannot add/delete keys. In fact we can apply this same reasoning to our LL implementation, where nodes themselves are structs.
 - Referring to the "value of the linked list node value" gets confusing.

WHAT ABOUT JS ?

Sound familiar?

- **JavaScript Objects**
- **JS Engines (like V8) implement most Objects as structs**
 - V8 defines a new struct every time you add a property
 - If this would be madness (ex: you are storing a phone book), it switches to using a hash table
- **In the end, the Object specified in EcmaScript is a data type; we know what behavior it should exhibit, but not necessarily how it is implemented at runtime. That's up to the engine.**

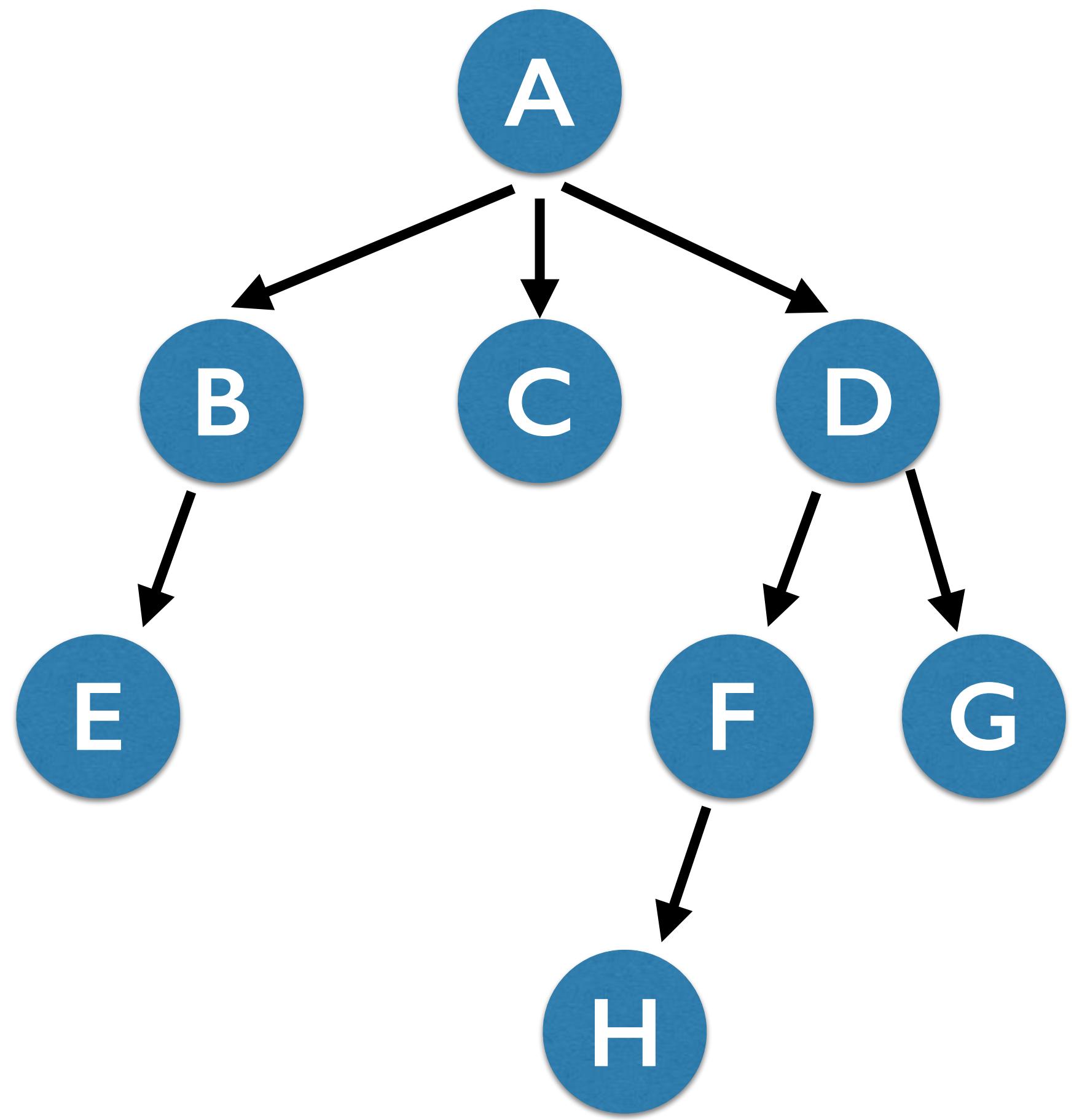


Trees



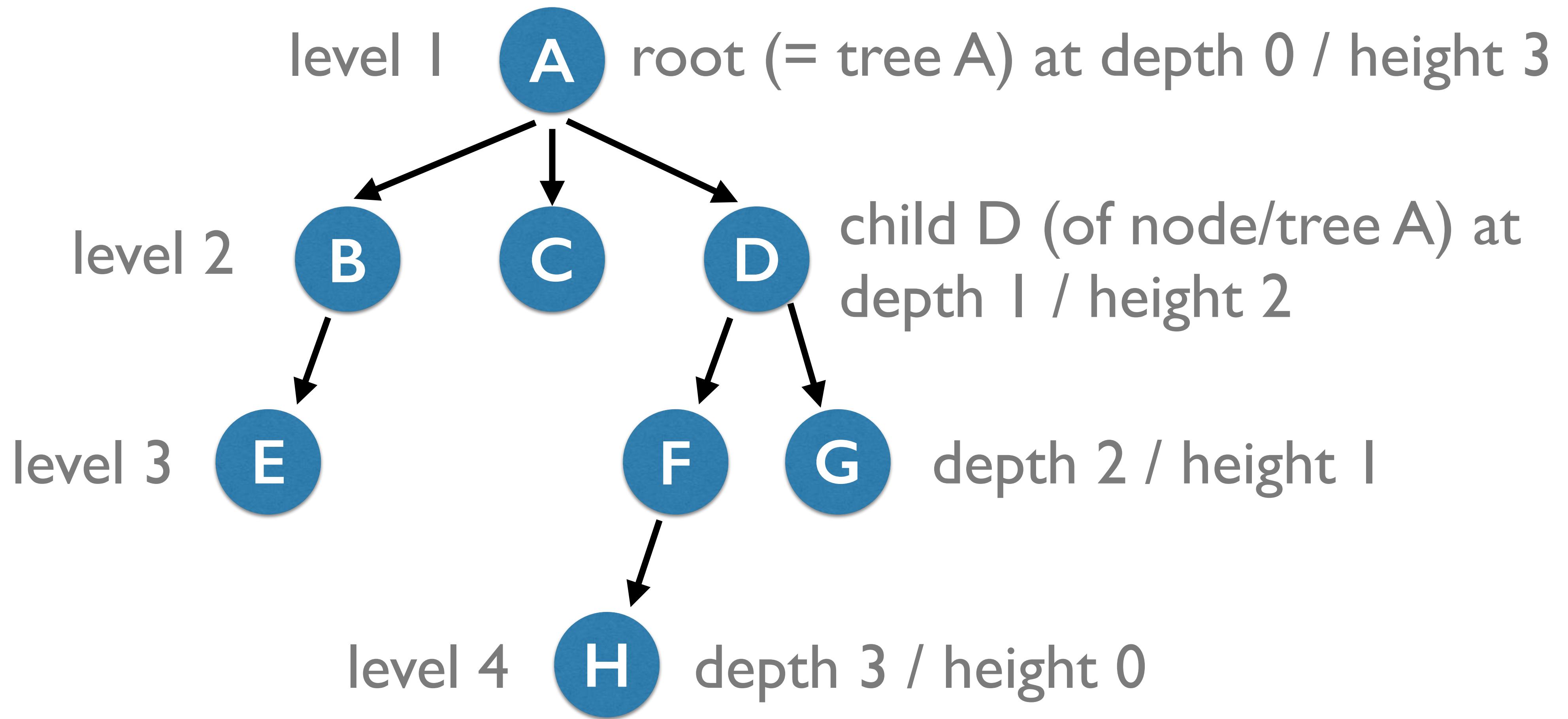


The Tree ADT



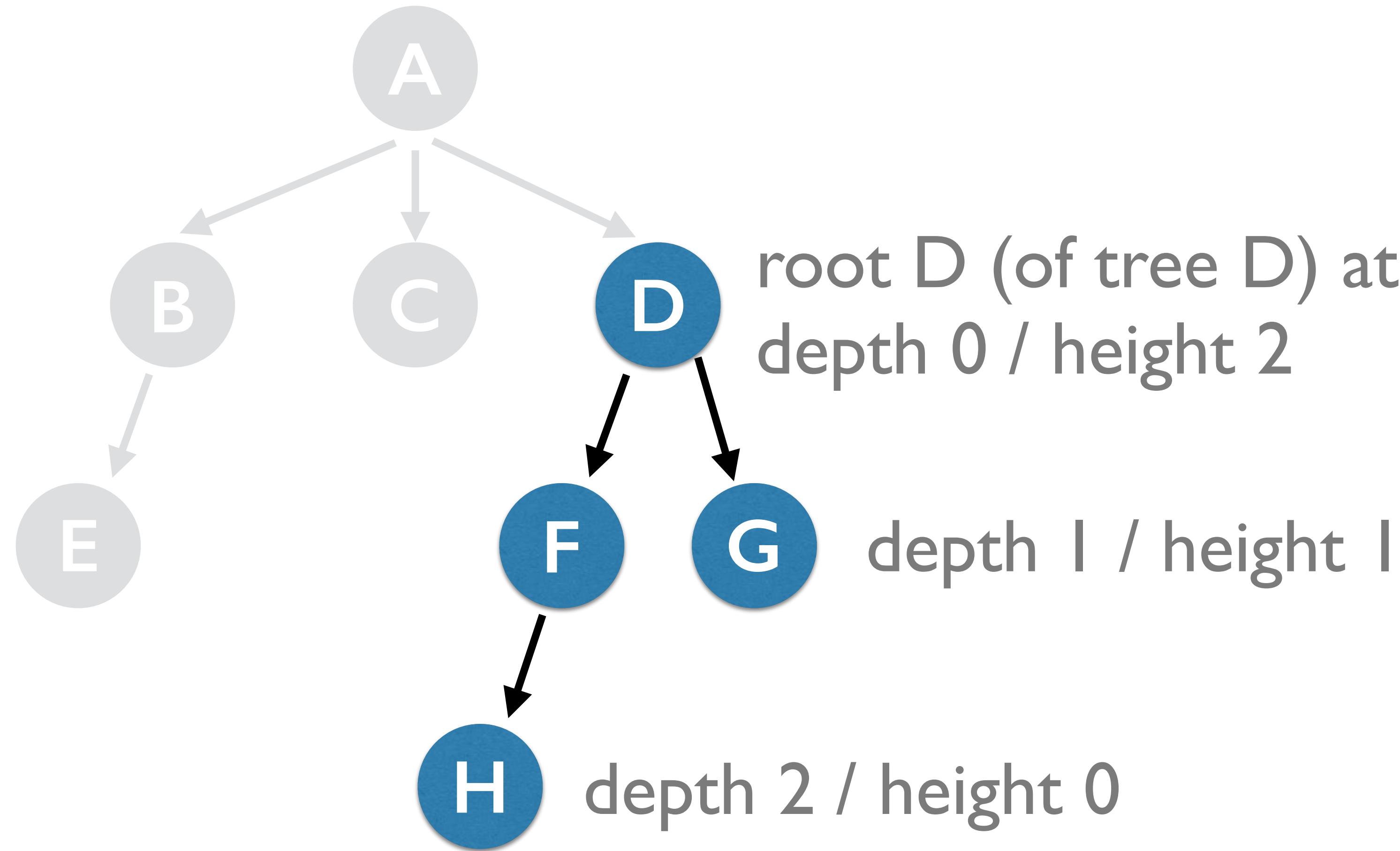
- Nodes contain value(s)
- A primary "root" node
- Children are subtrees (recursive!)
- No duplicated children (cycles); trees can *branch* but never *converge*
- Final nodes called "leaves"
- Height of tree = longest path to leaf
- Level = 1 + number of cnxns to root
- Depth = inverse of height

The Tree ADT





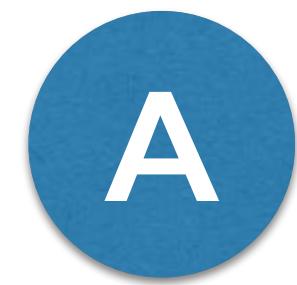
Every node is the root of a tree.
You might even say a node *represents* a tree.



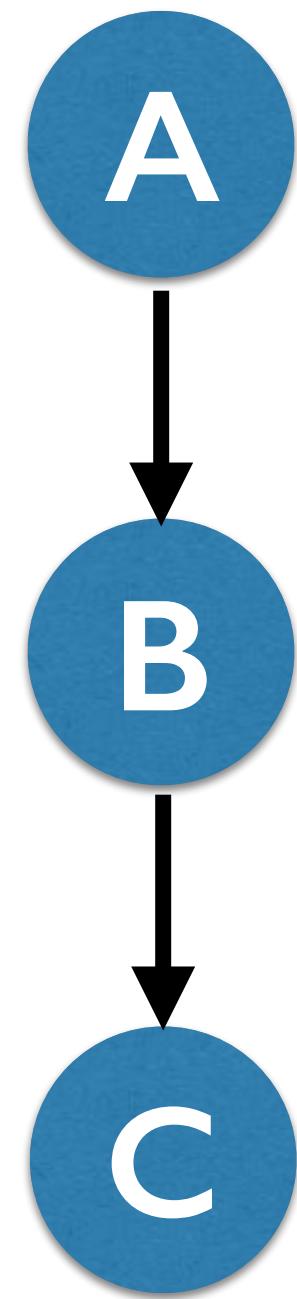


"Degenerate" trees are still trees

A tree of one node

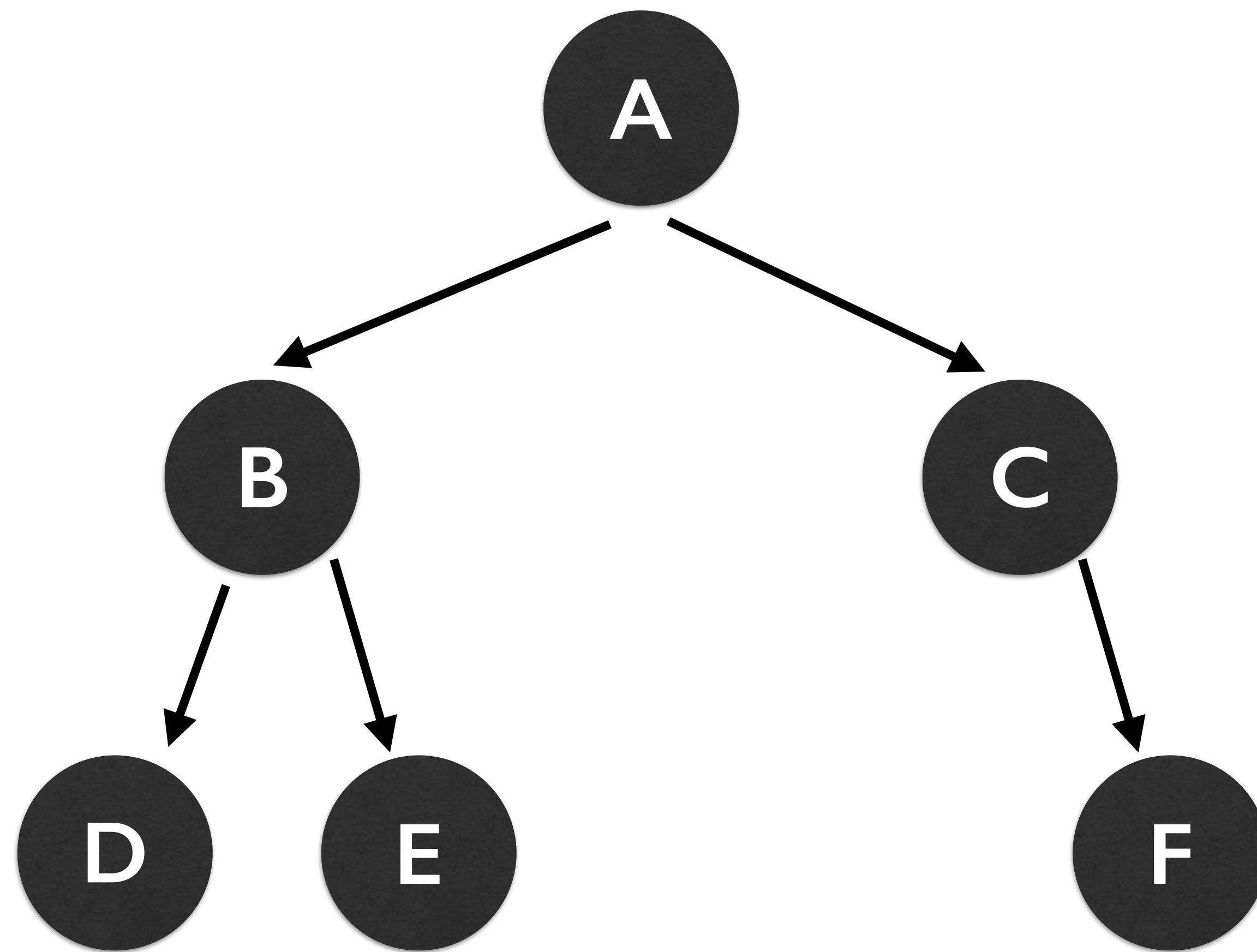


A tree of three nodes

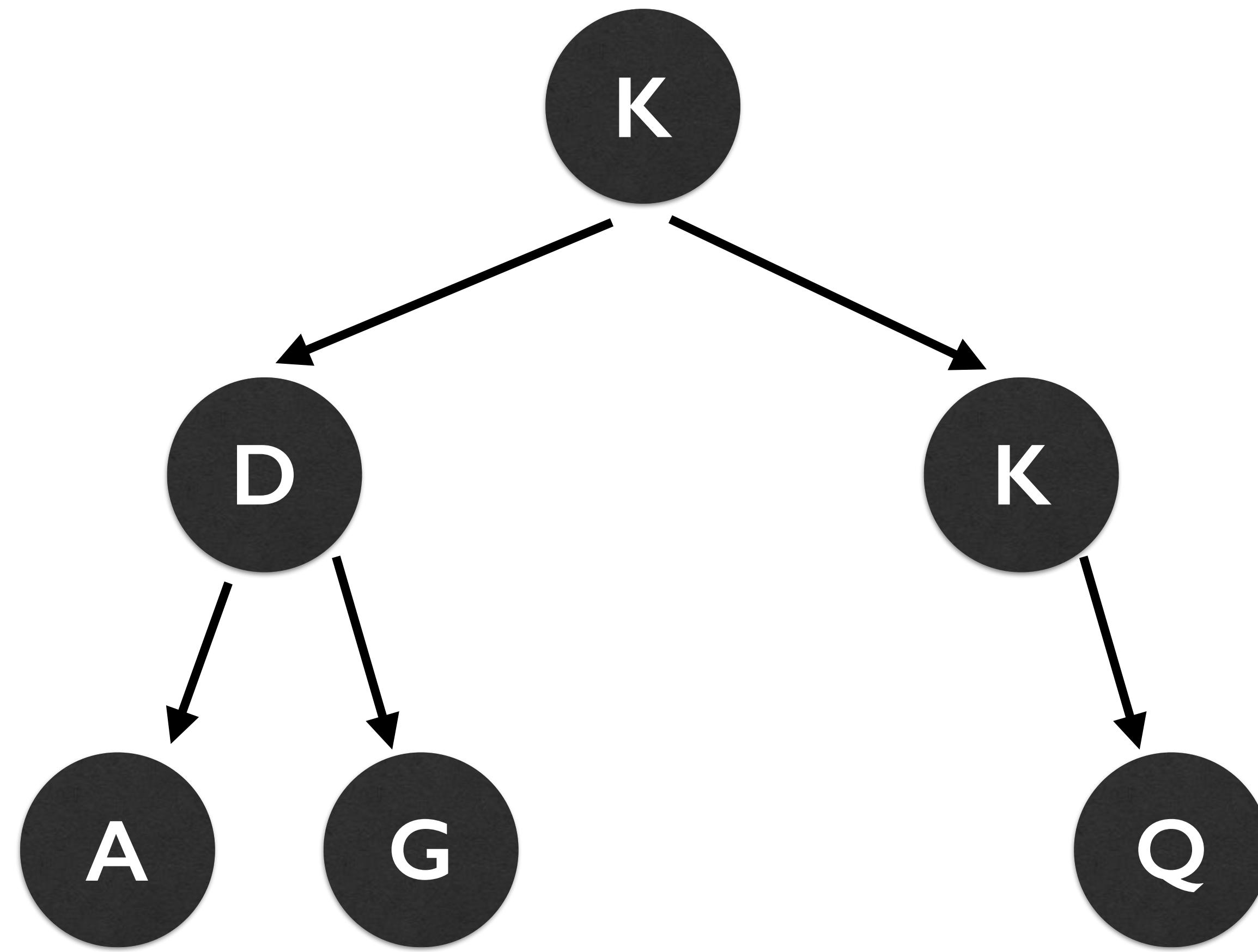




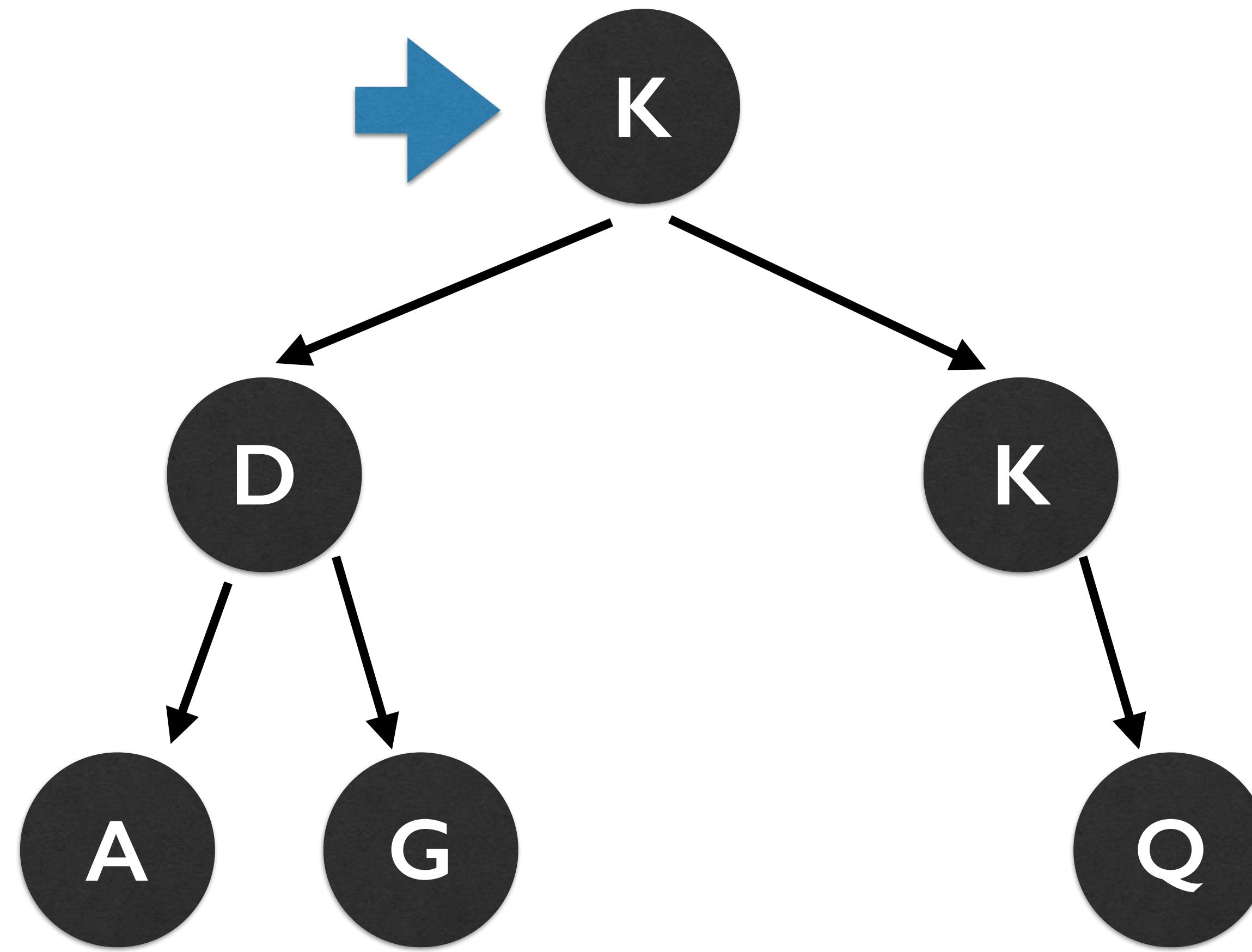
Binary Tree



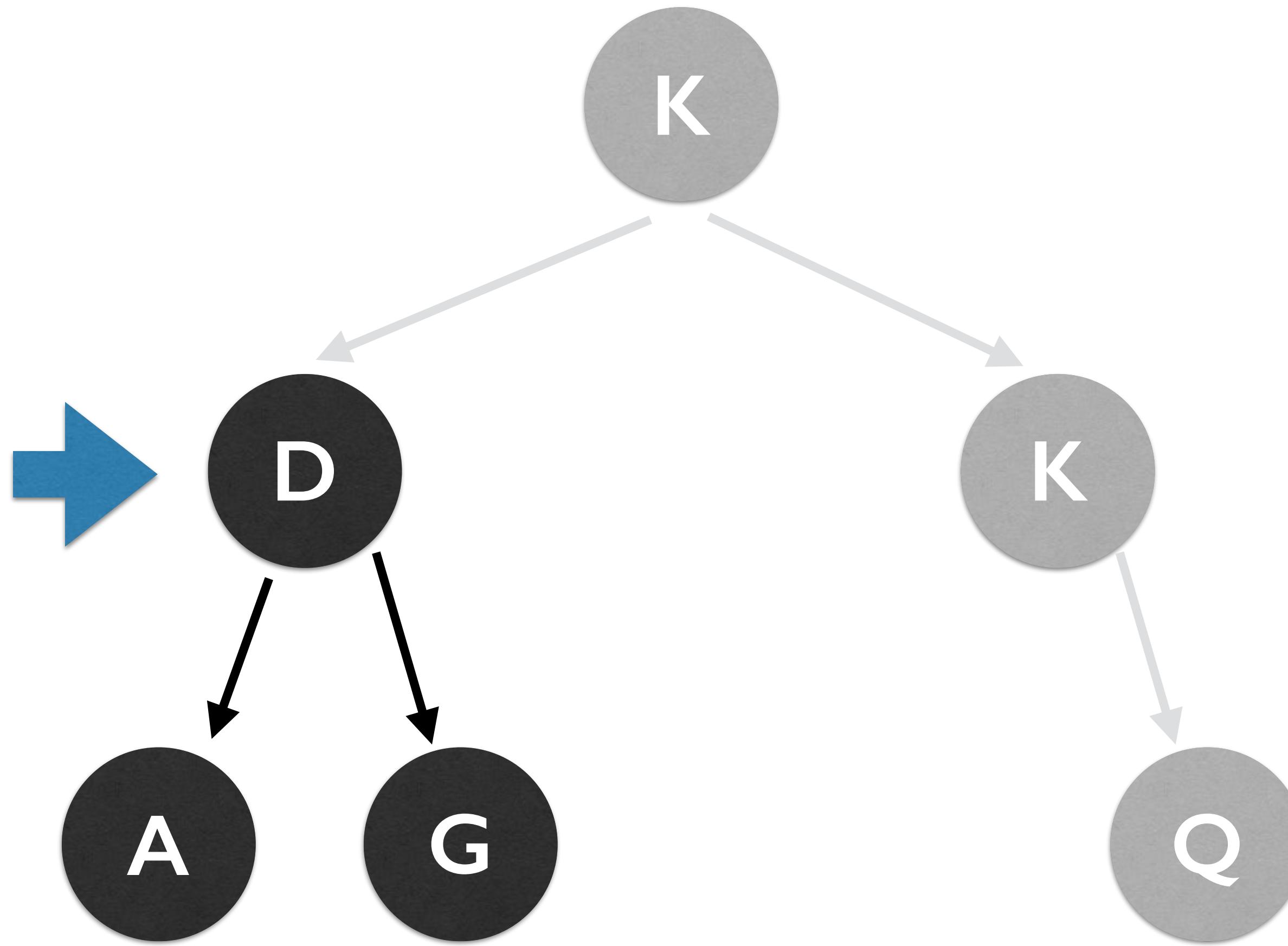
Binary Search Tree



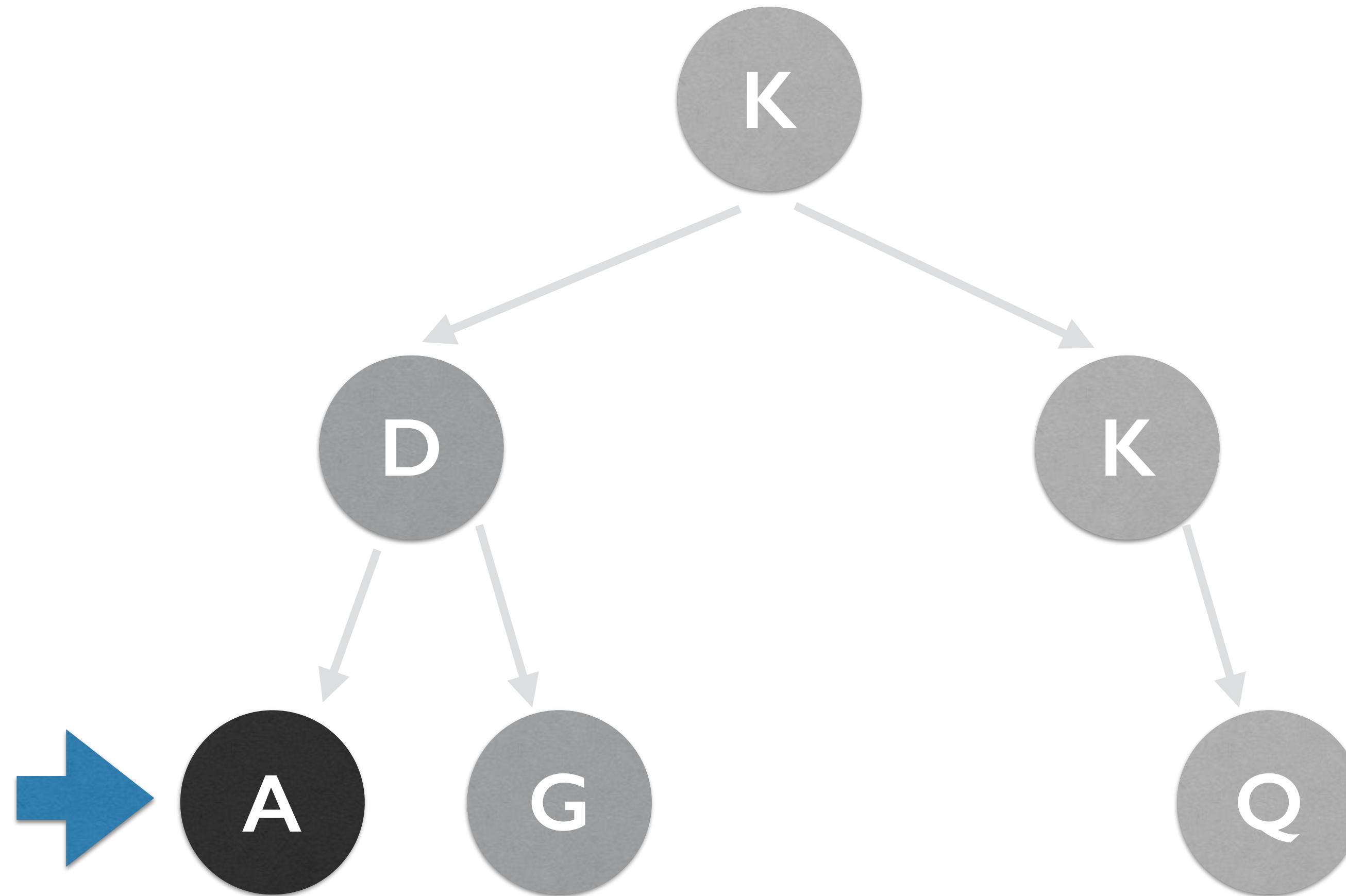
Binary Search Tree: Min Value?



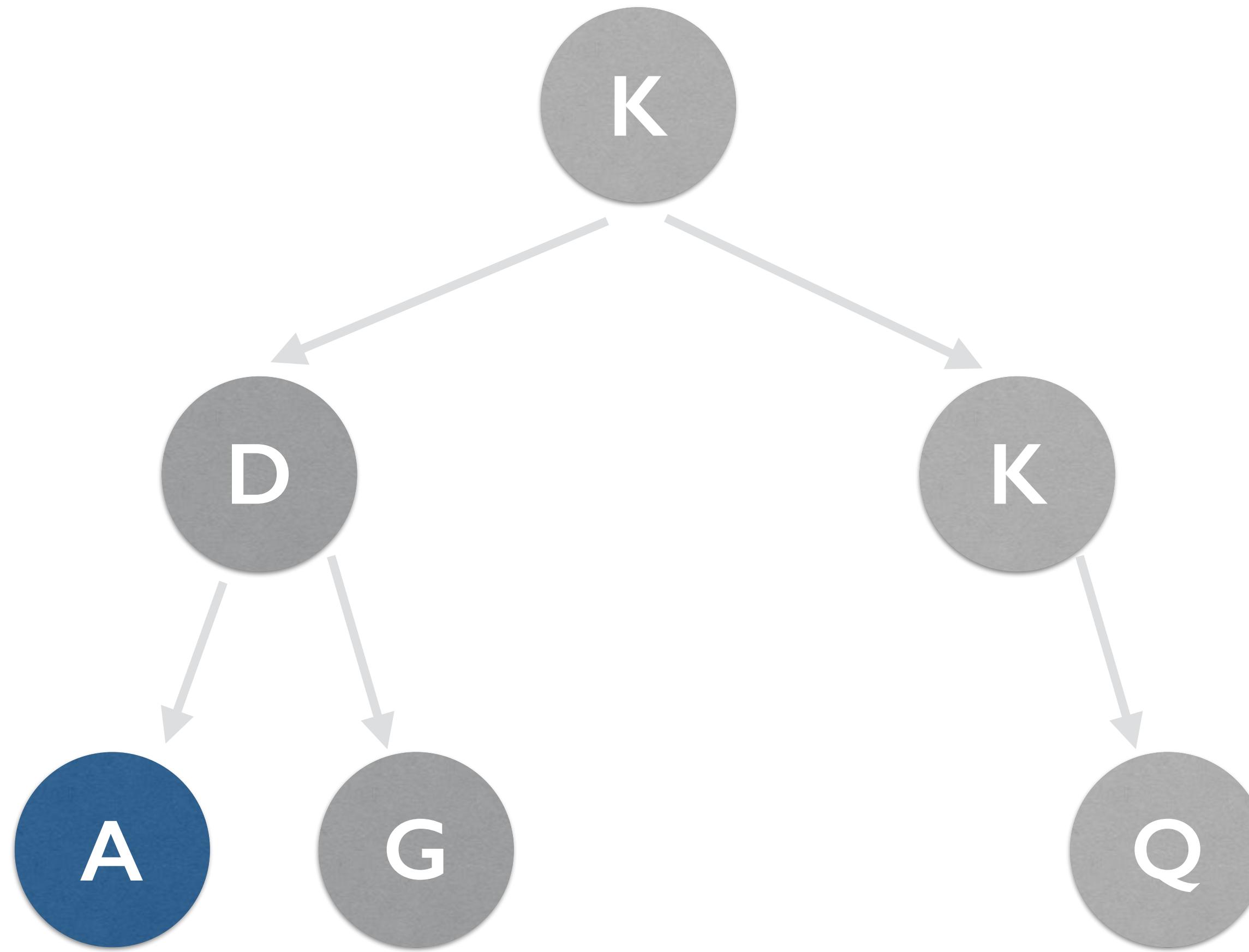
Binary Search Tree: Min Value?



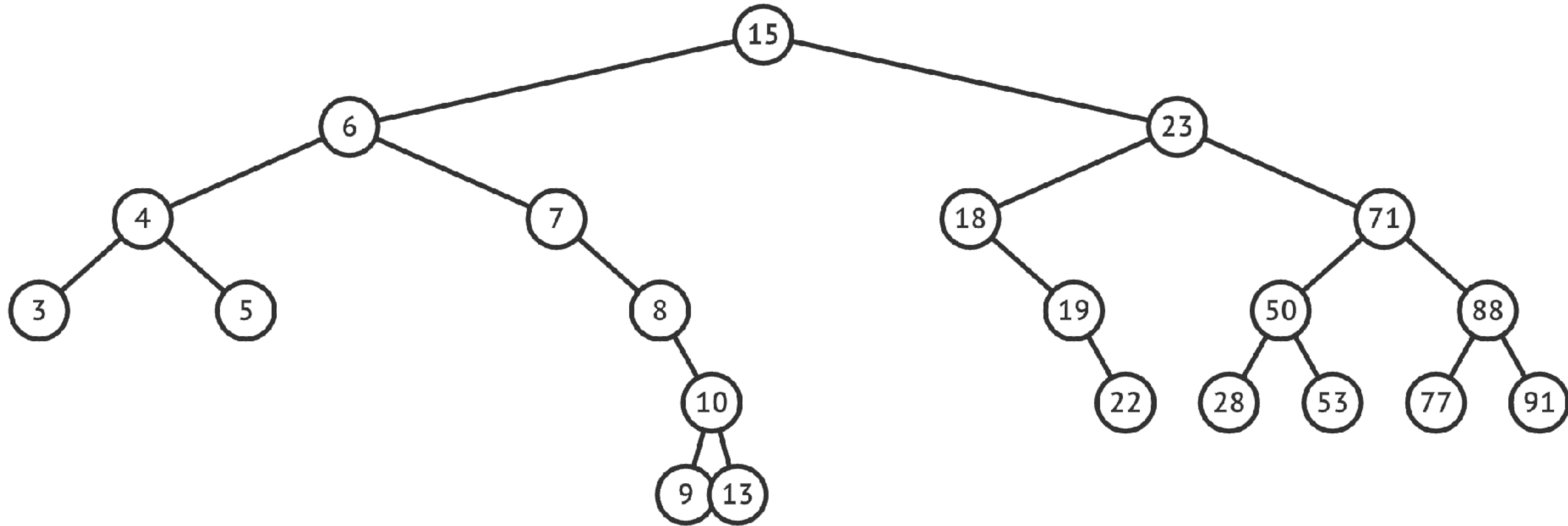
Binary Search Tree: Min Value?



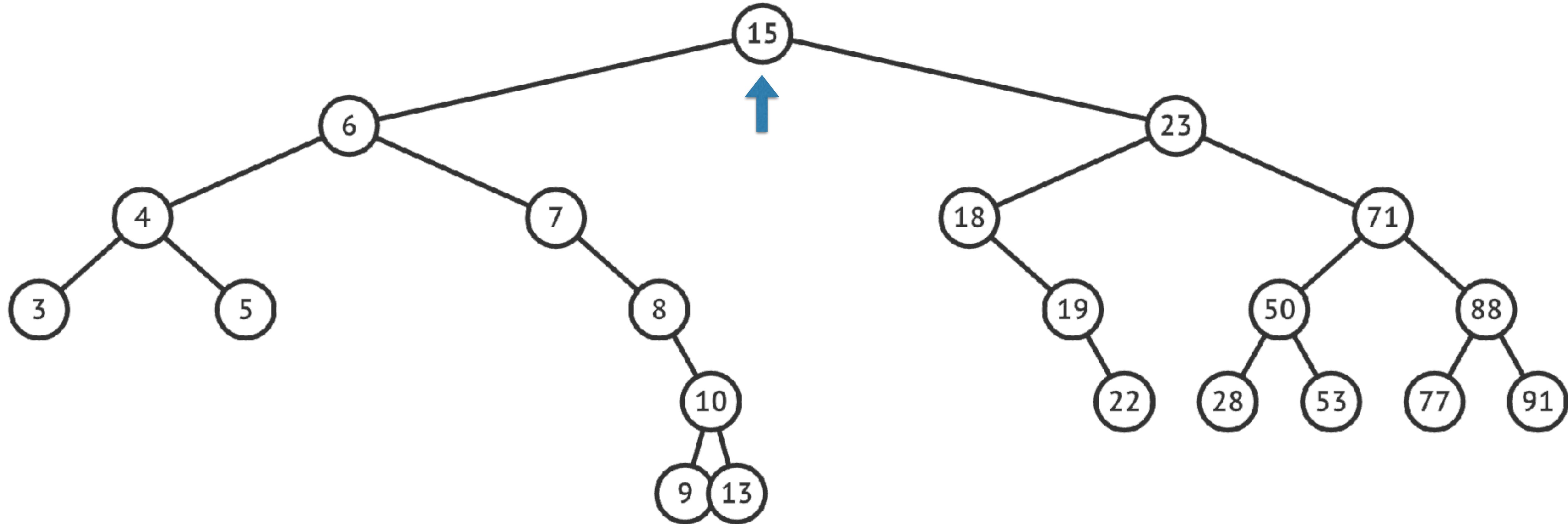
Binary Search Tree: Min Value?



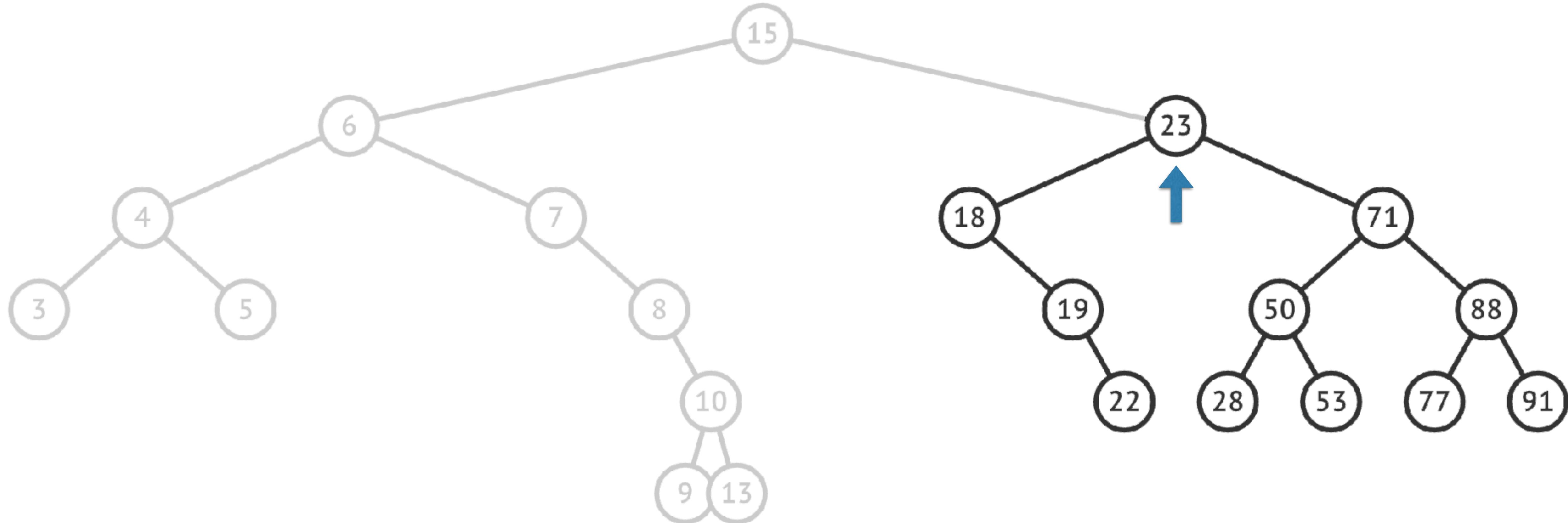
Does this BST contain the value 28?



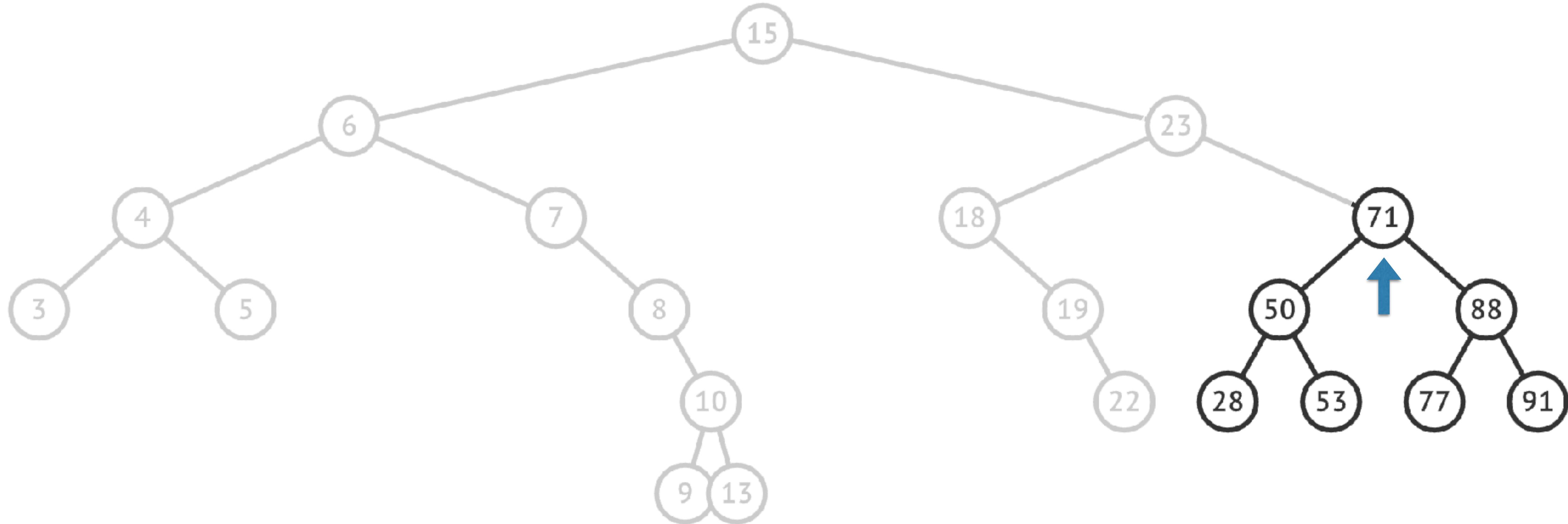
Does this BST contain the value 28?



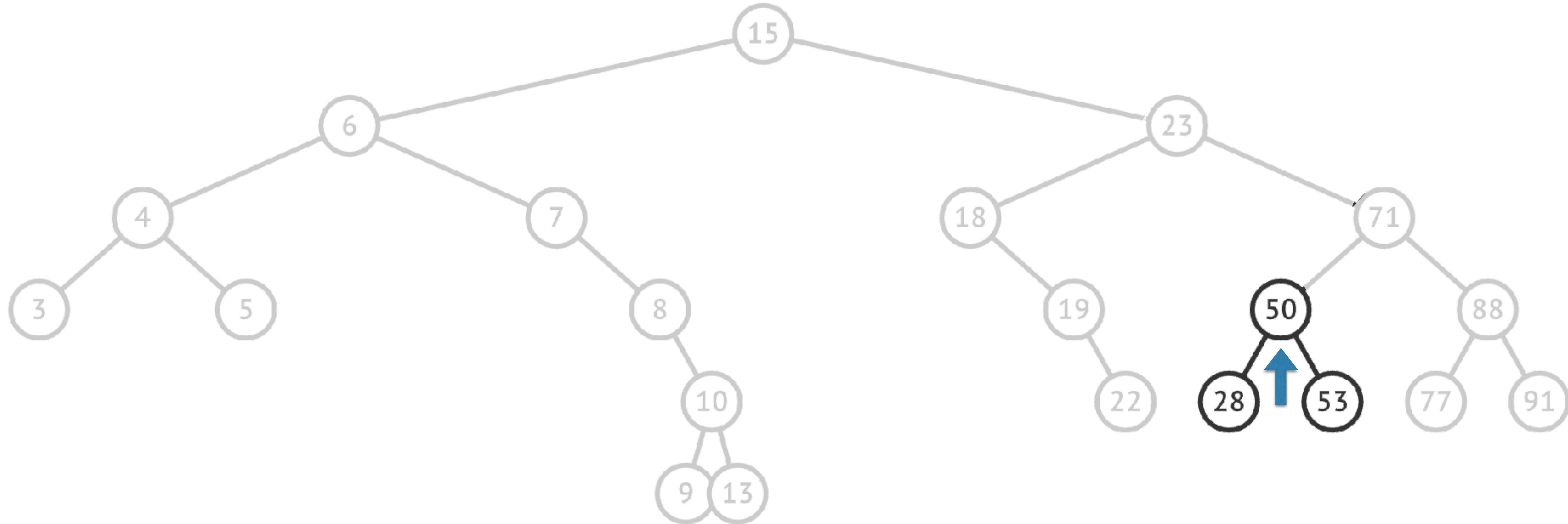
Does this BST contain the value 28?



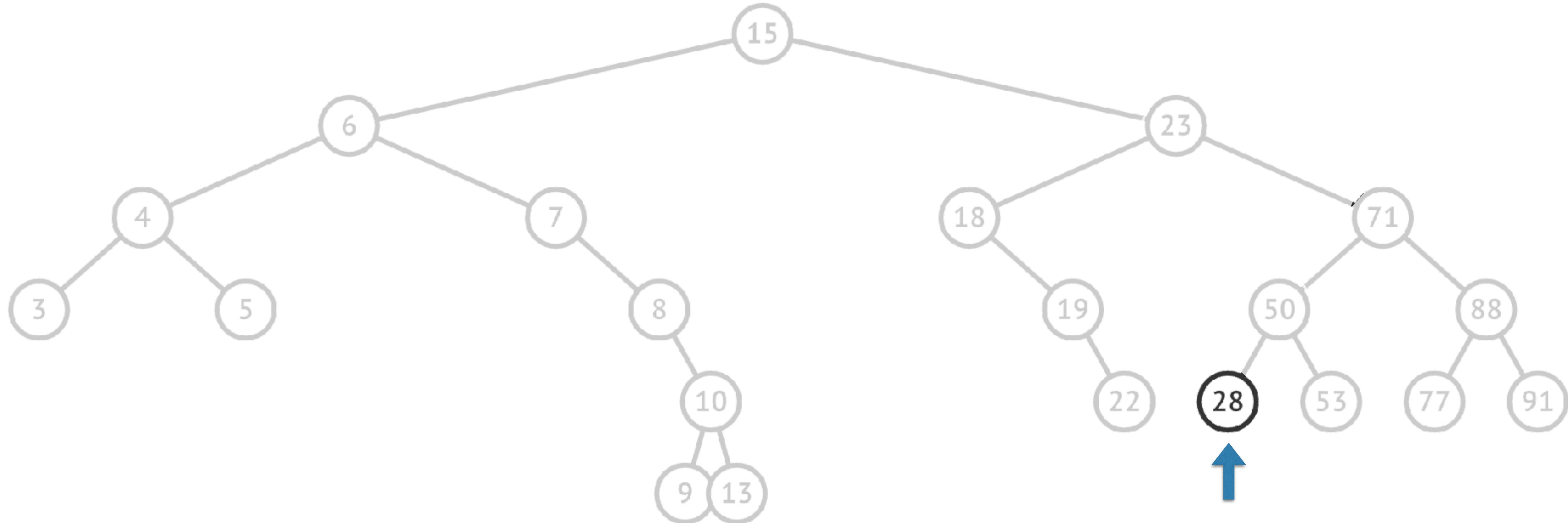
Does this BST contain the value 28?



Does this BST contain the value 28?



Does this BST contain the value 28?



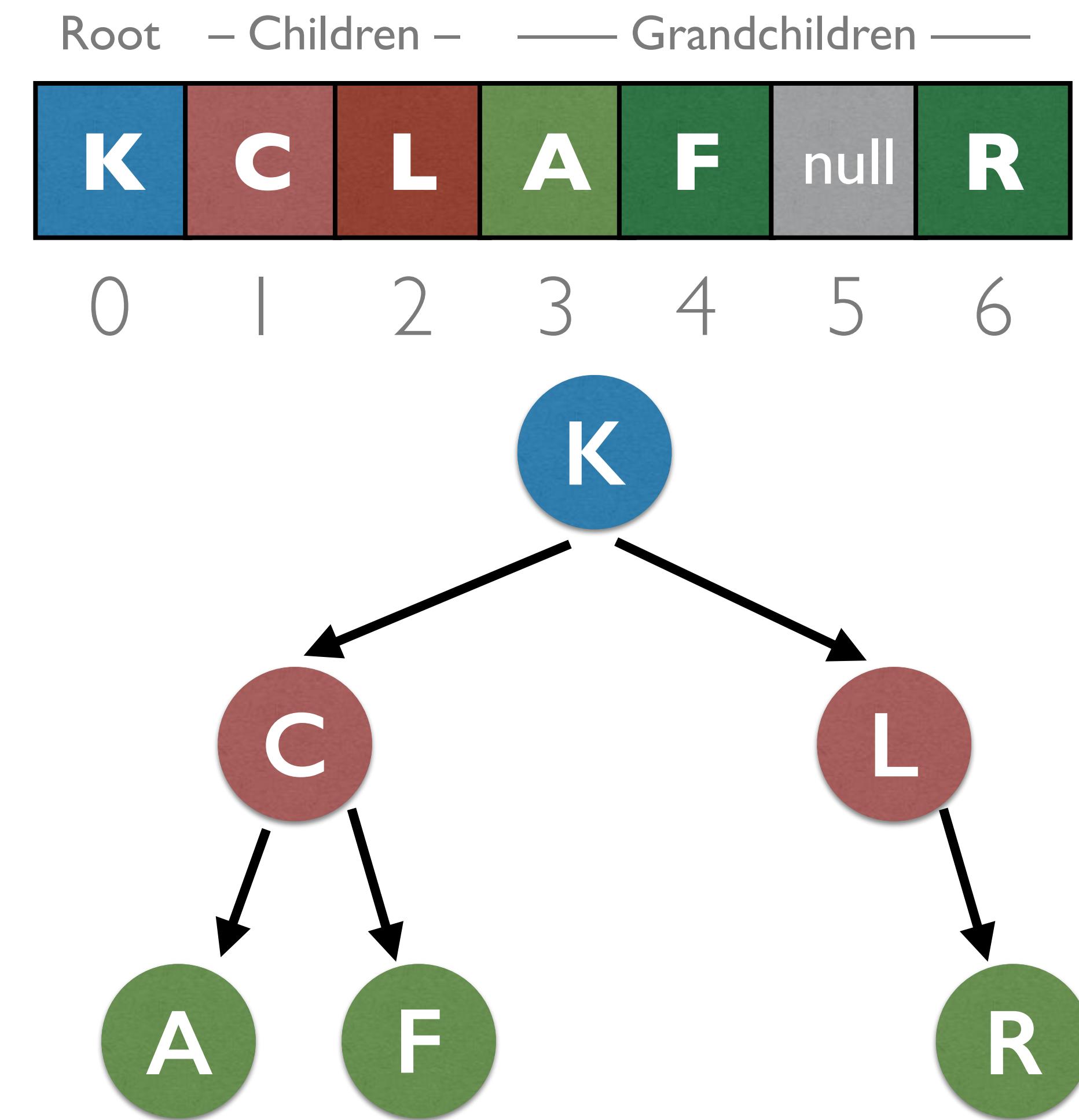
BST ADT

- 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)?

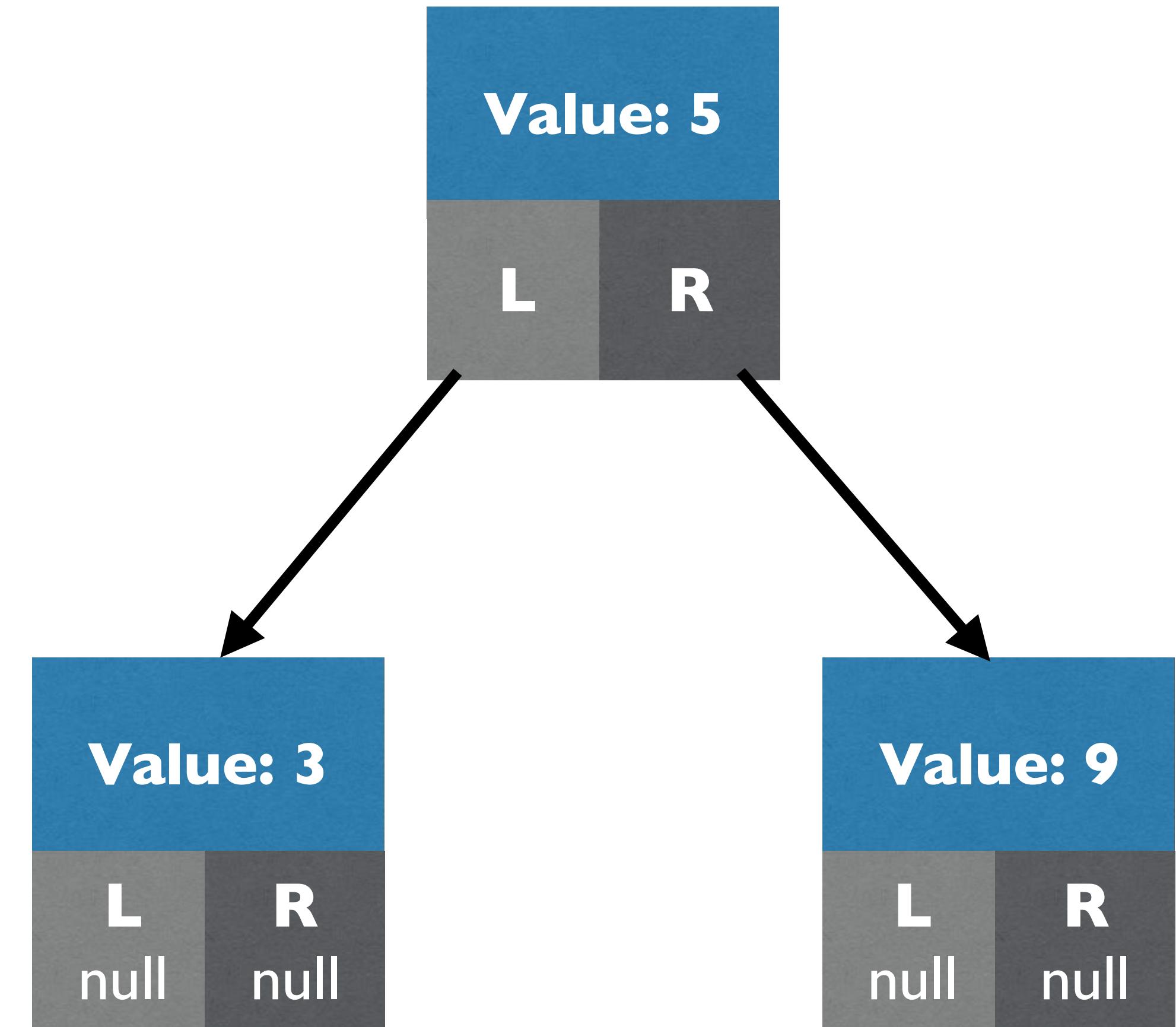
- The Tree ADT, with its talk of "nodes" and "references," seems so obviously to describe a data *structure* that it is perhaps confusing to tell the two apart.
- In fact, a tree can be stored in a few different ways. For example, if you knew your tree nodes always had at most two children, you could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





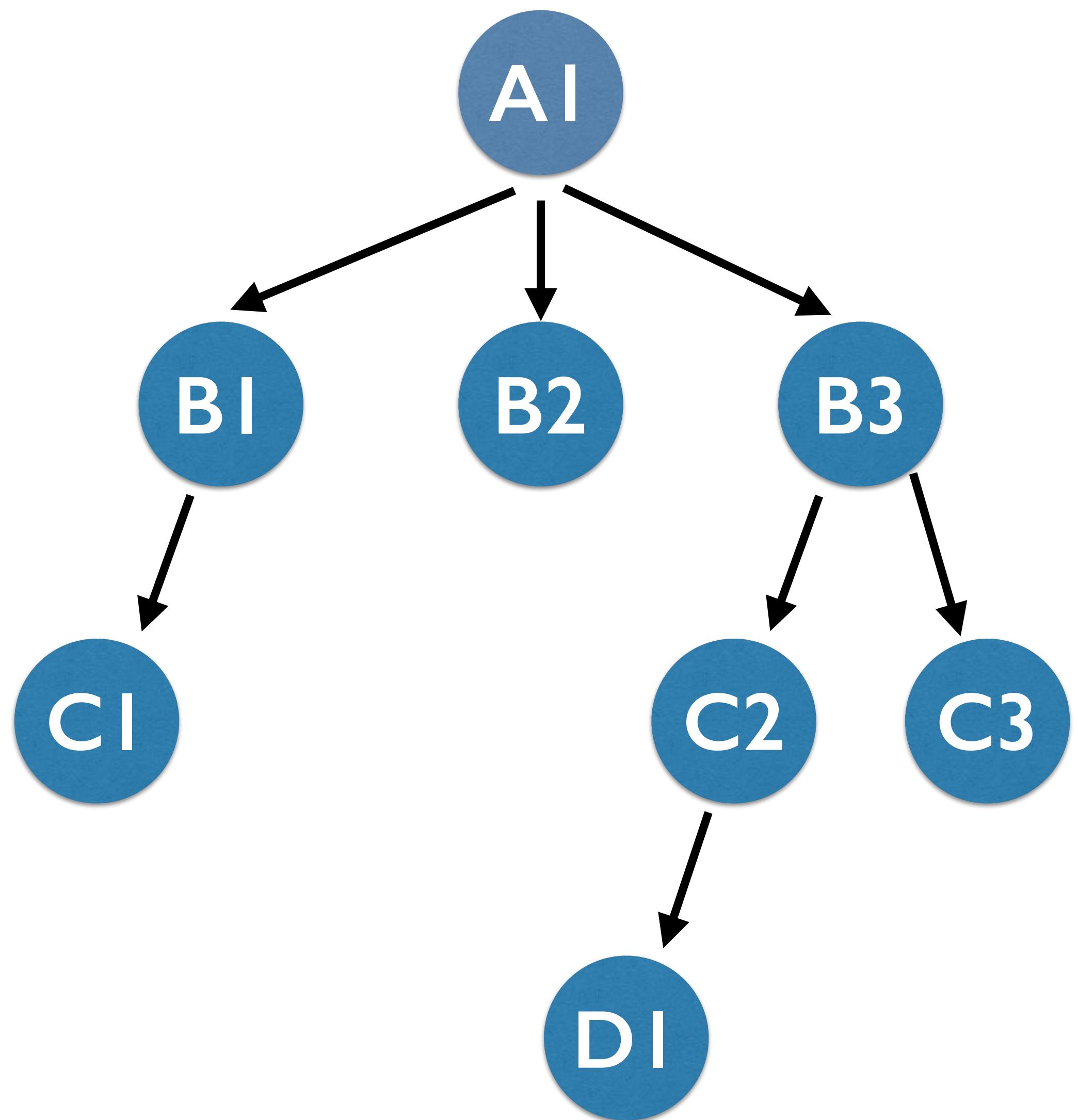
Tree traversal

Traversal: visiting every node

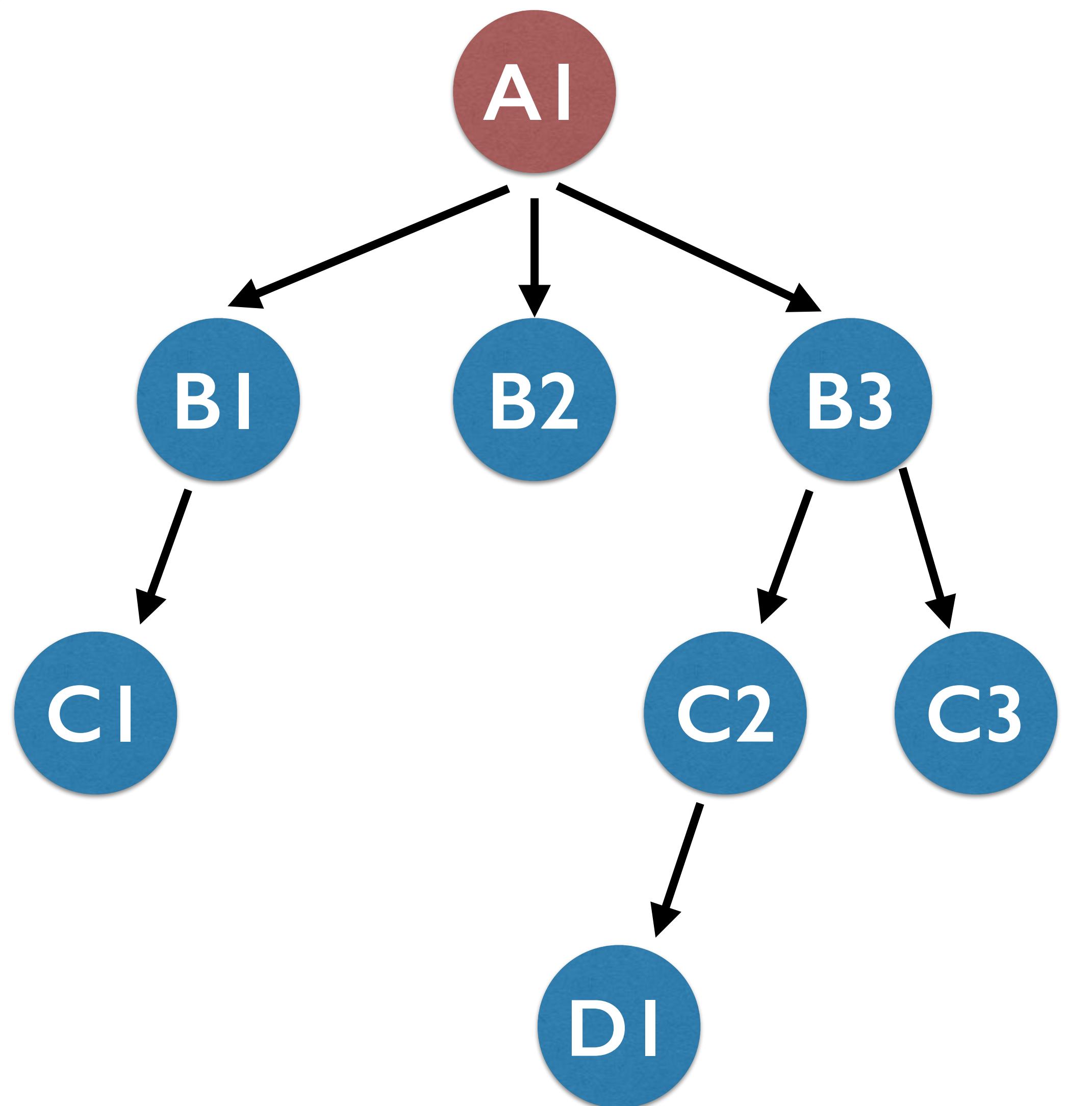
- Breadth-first search (level by level)
- Depth-first search (branch by branch)
 - Pre-order: process **root node**, process **left subtree**, process **right subtree**
 - In-order: process **left subtree**, process **root node**, process **right subtree**
 - Post-order: process **left subtree**, process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

Breadth-First

BFS

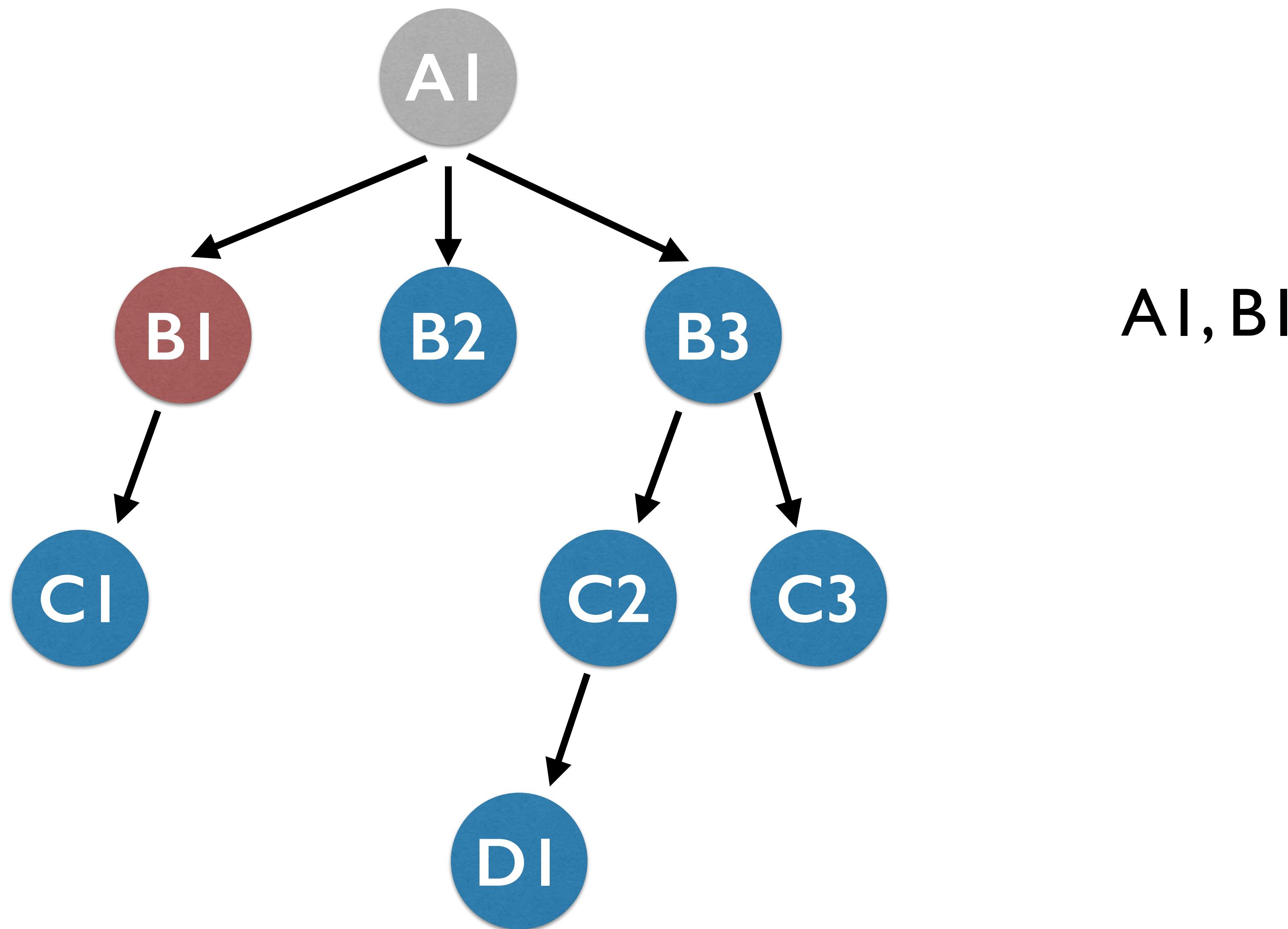


BFS

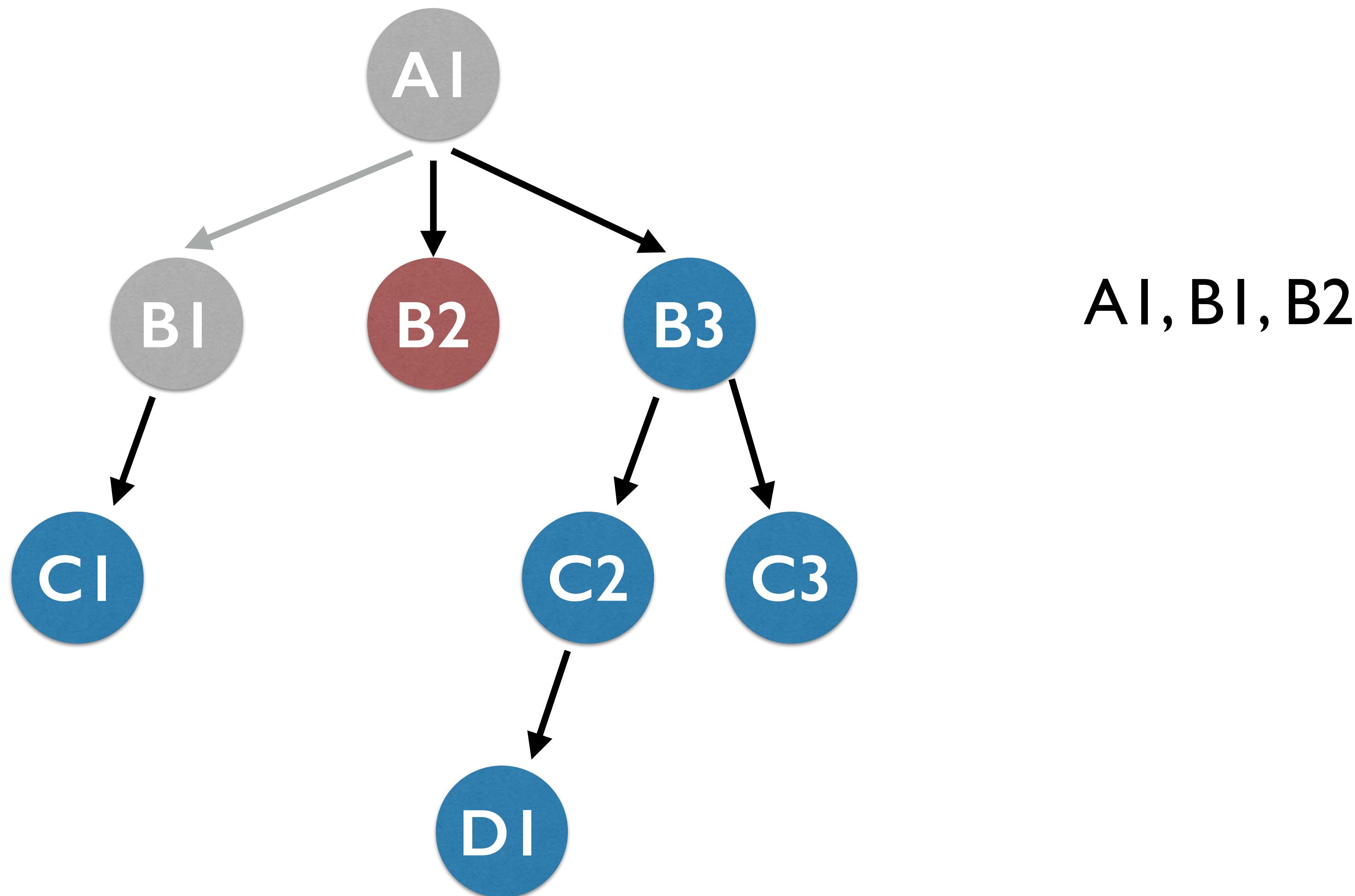


AI

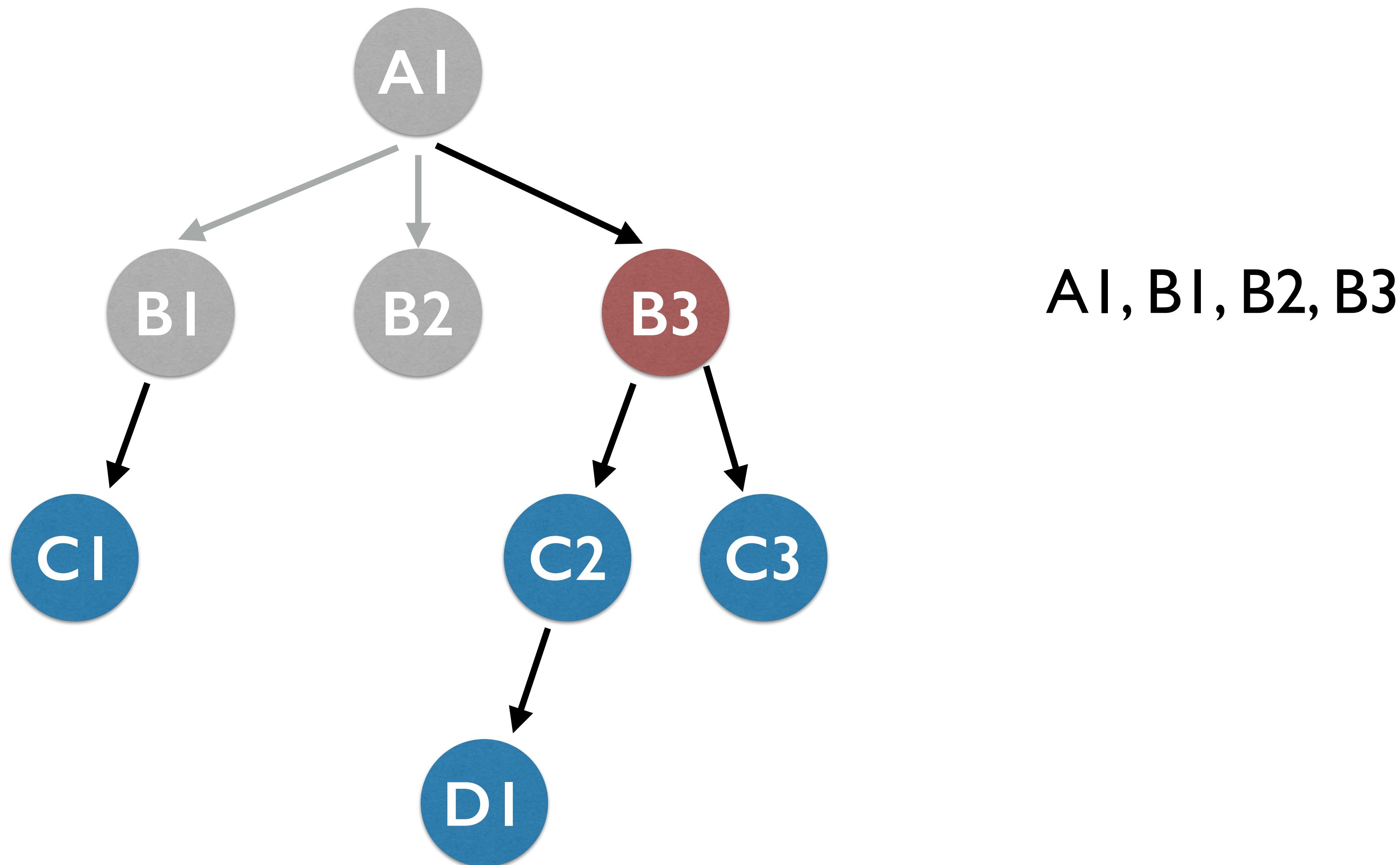
BFS



BFS

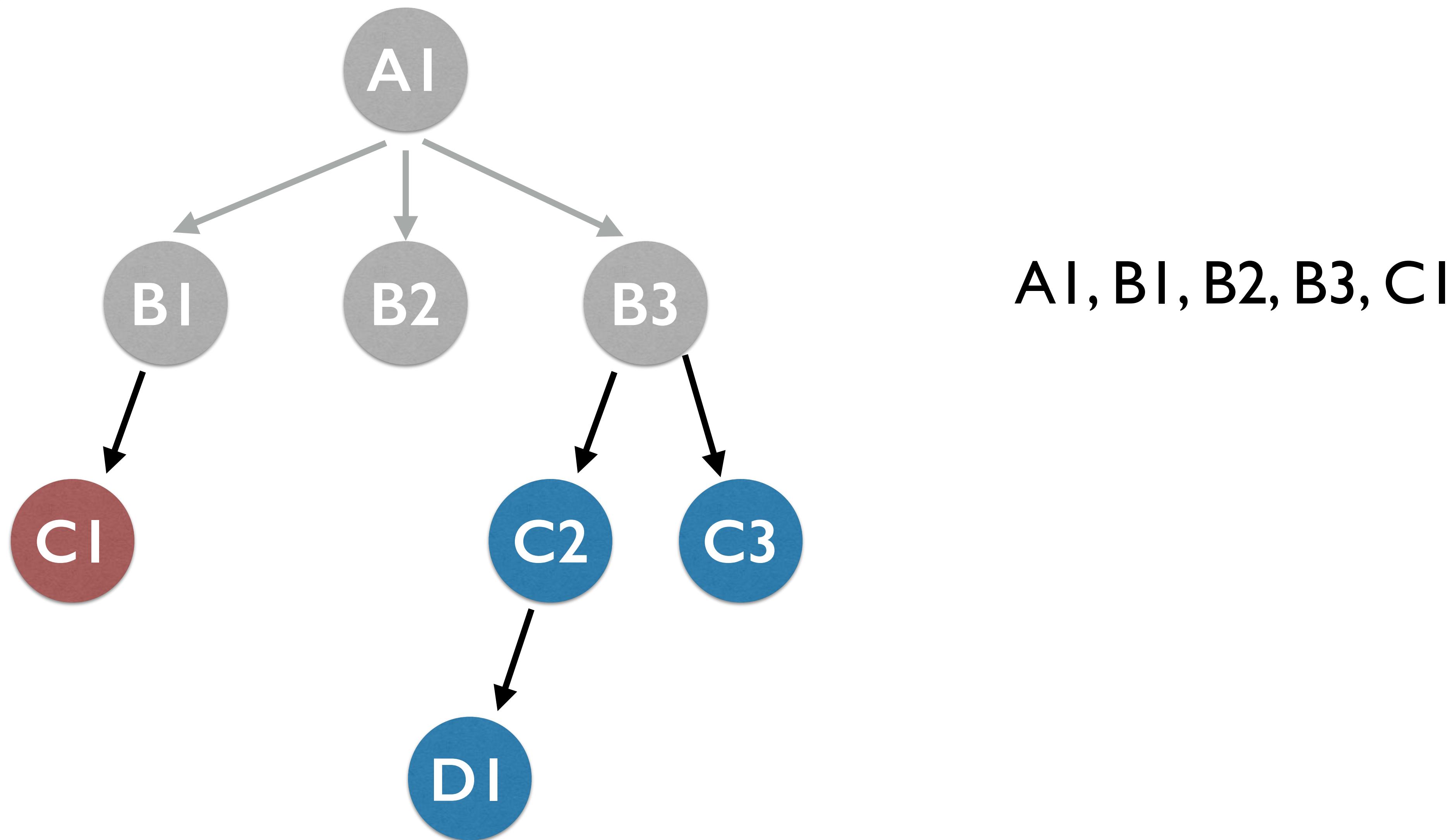


BFS

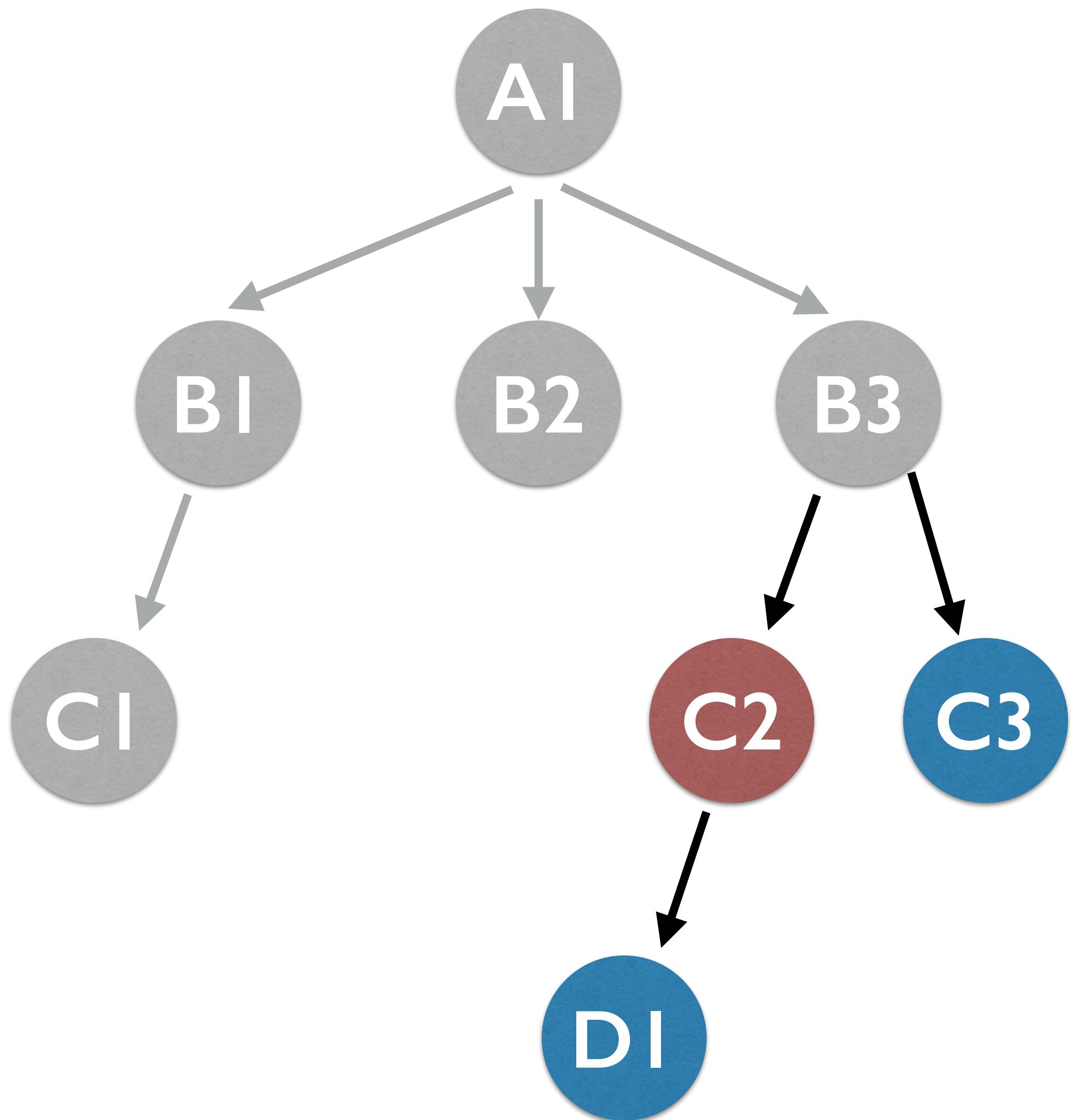


A1, B1, B2, B3

BFS

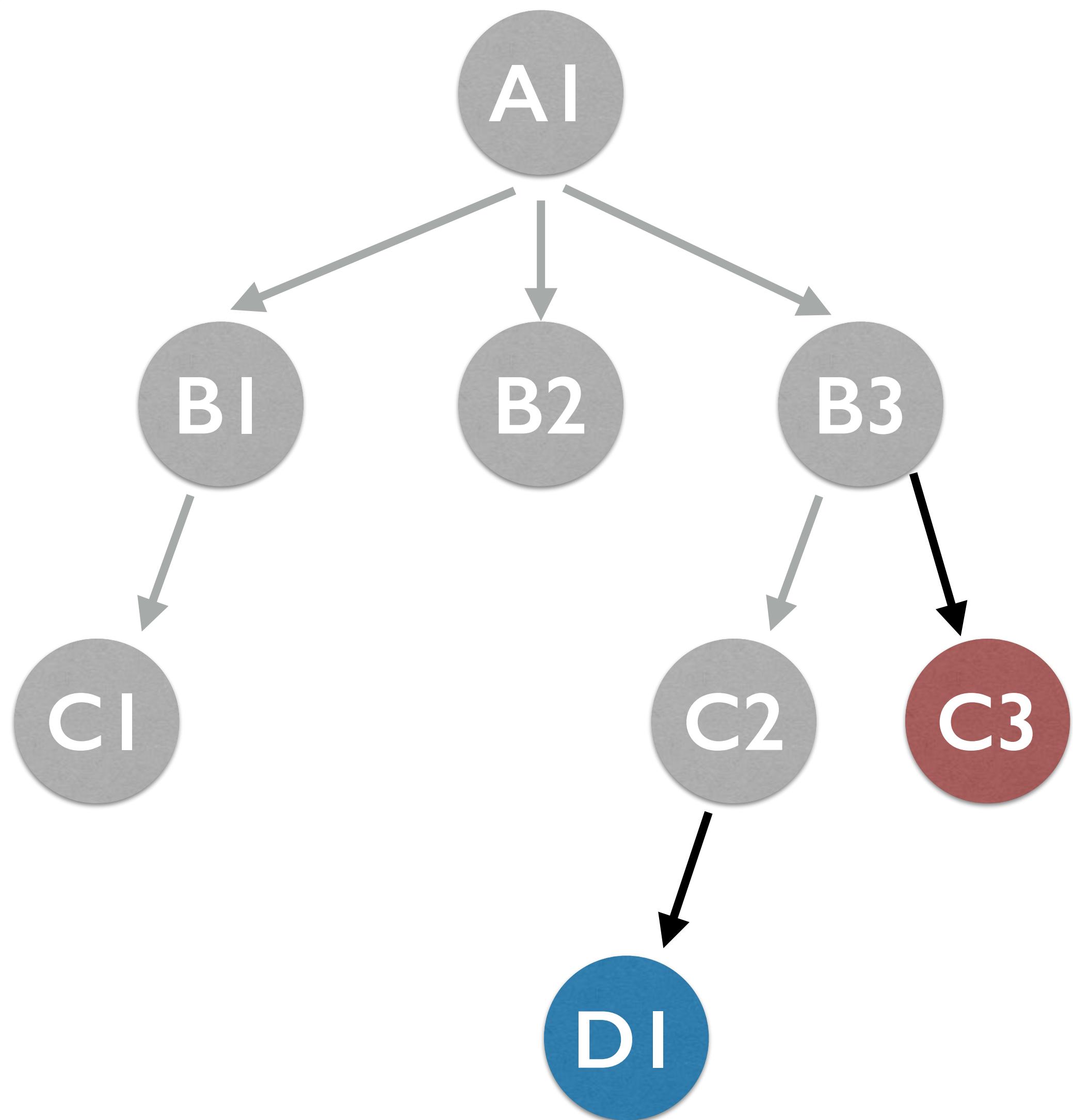


BFS



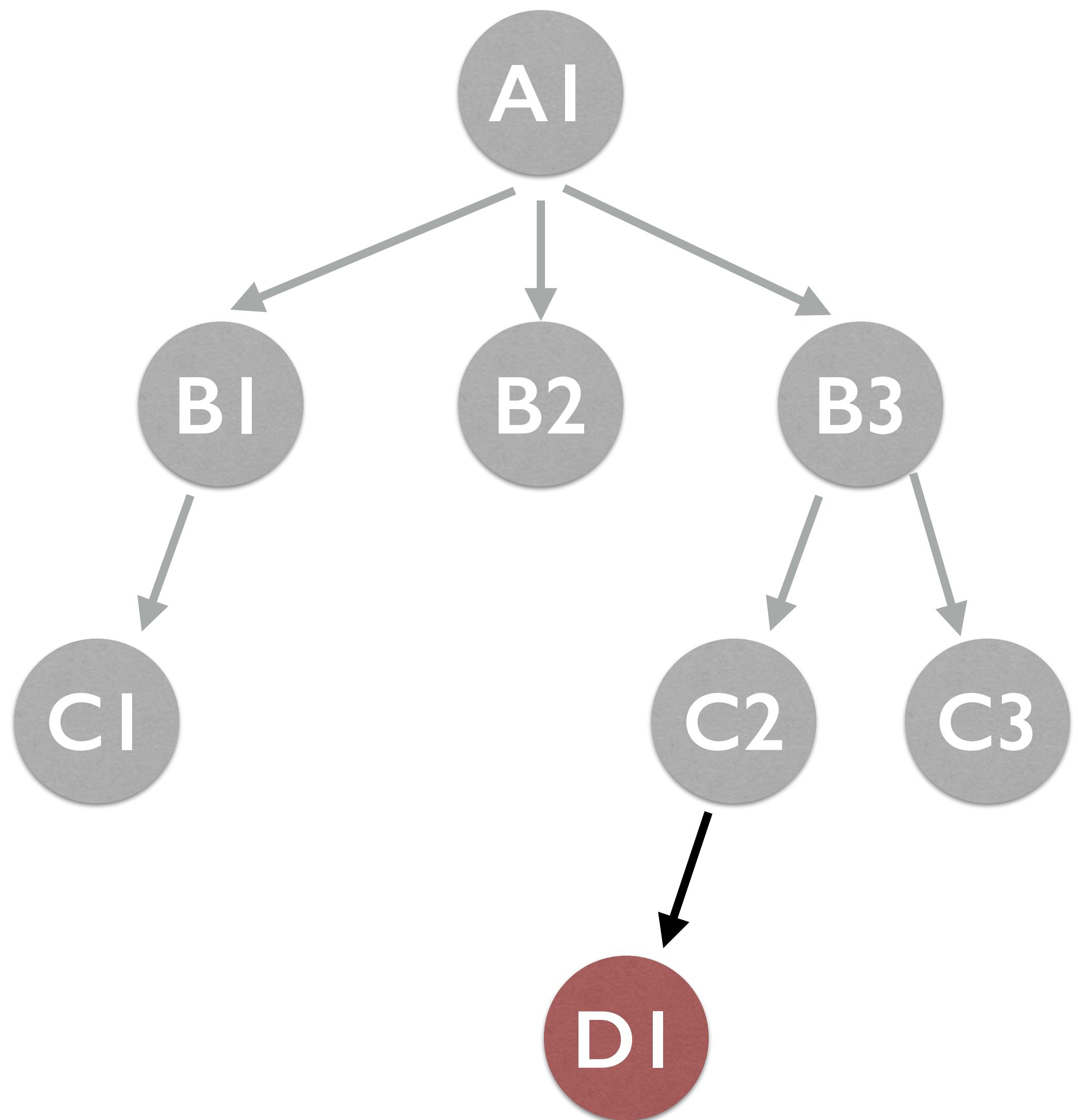
A1, B1, B2, B3, C1, C2

BFS



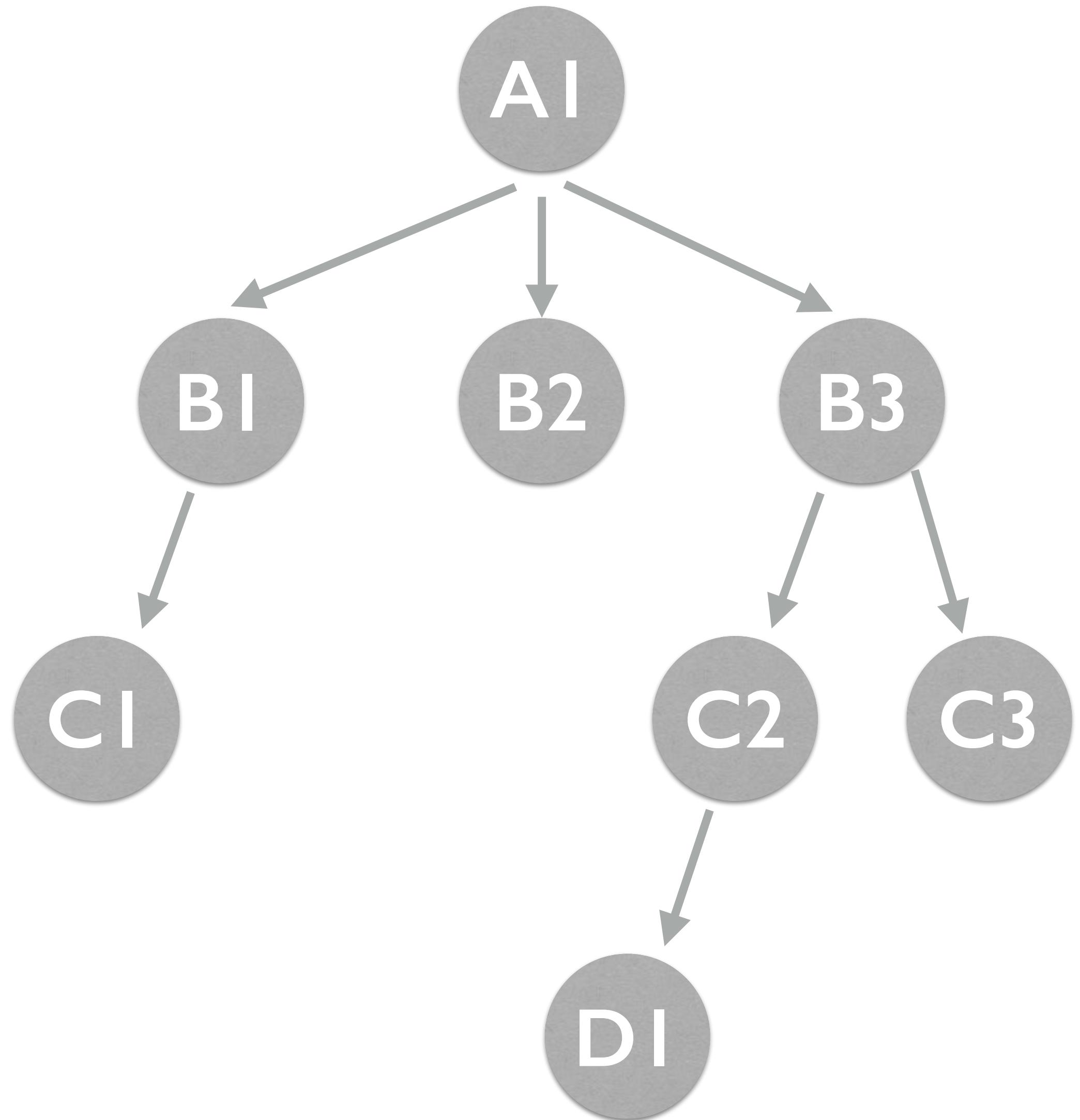
A1, B1, B2, B3, C1, C2, C3

BFS



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

BFS

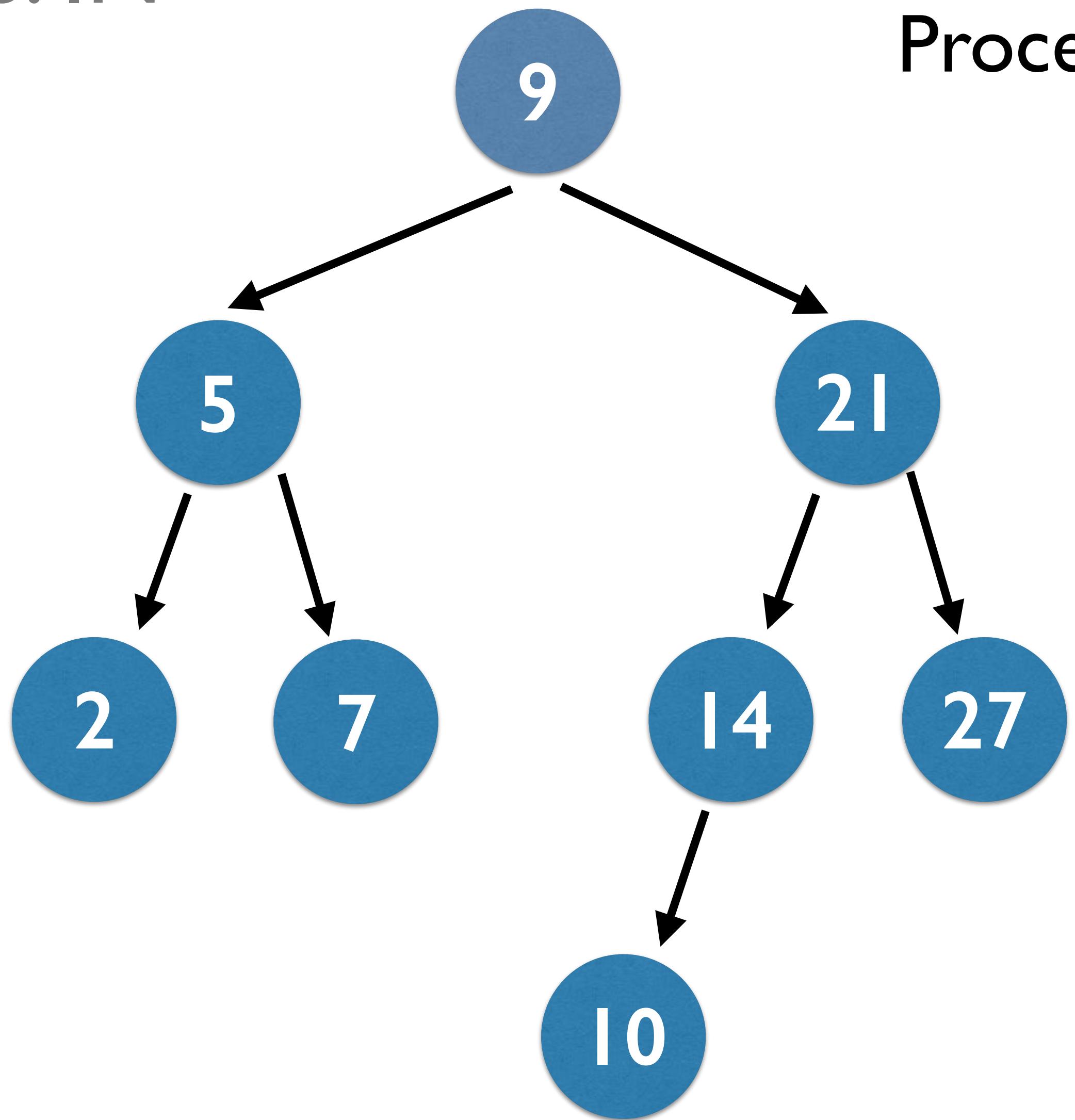


A1, B1, B2, B3, C1, C2, C3, D1

- 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!

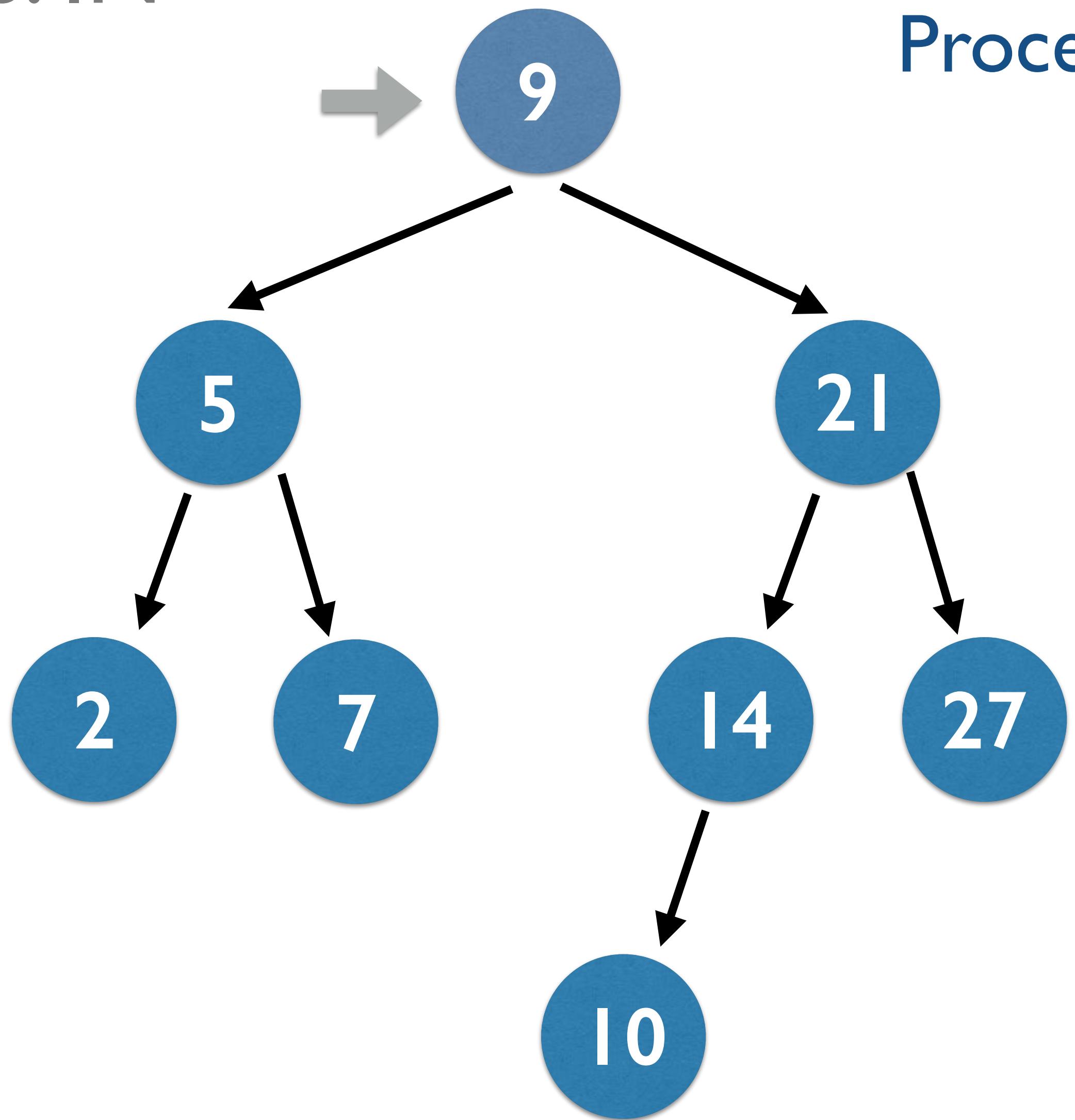
Depth First: In-Order

DFS: IN



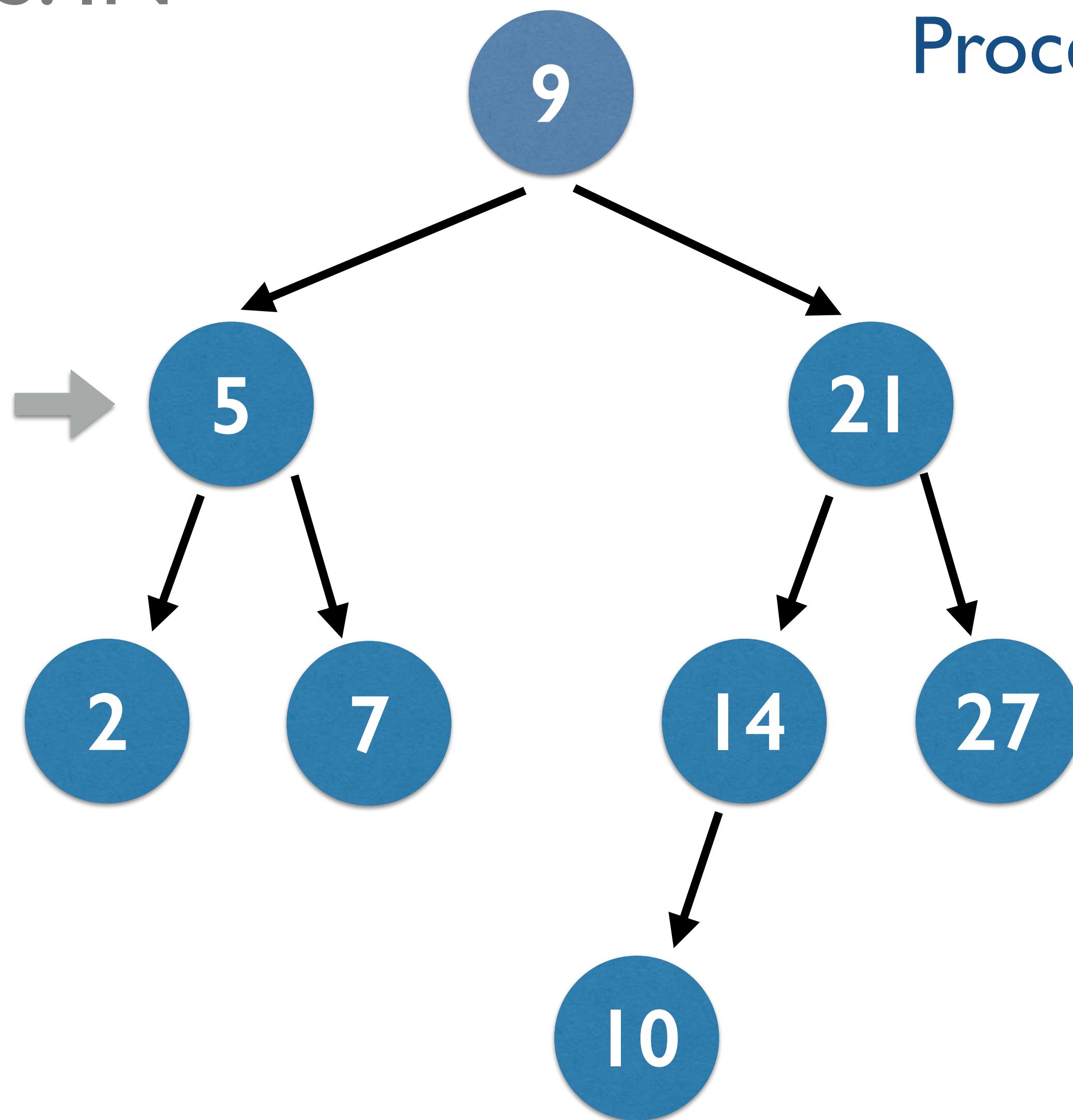
Process left · Process root · Process right

DFS: IN



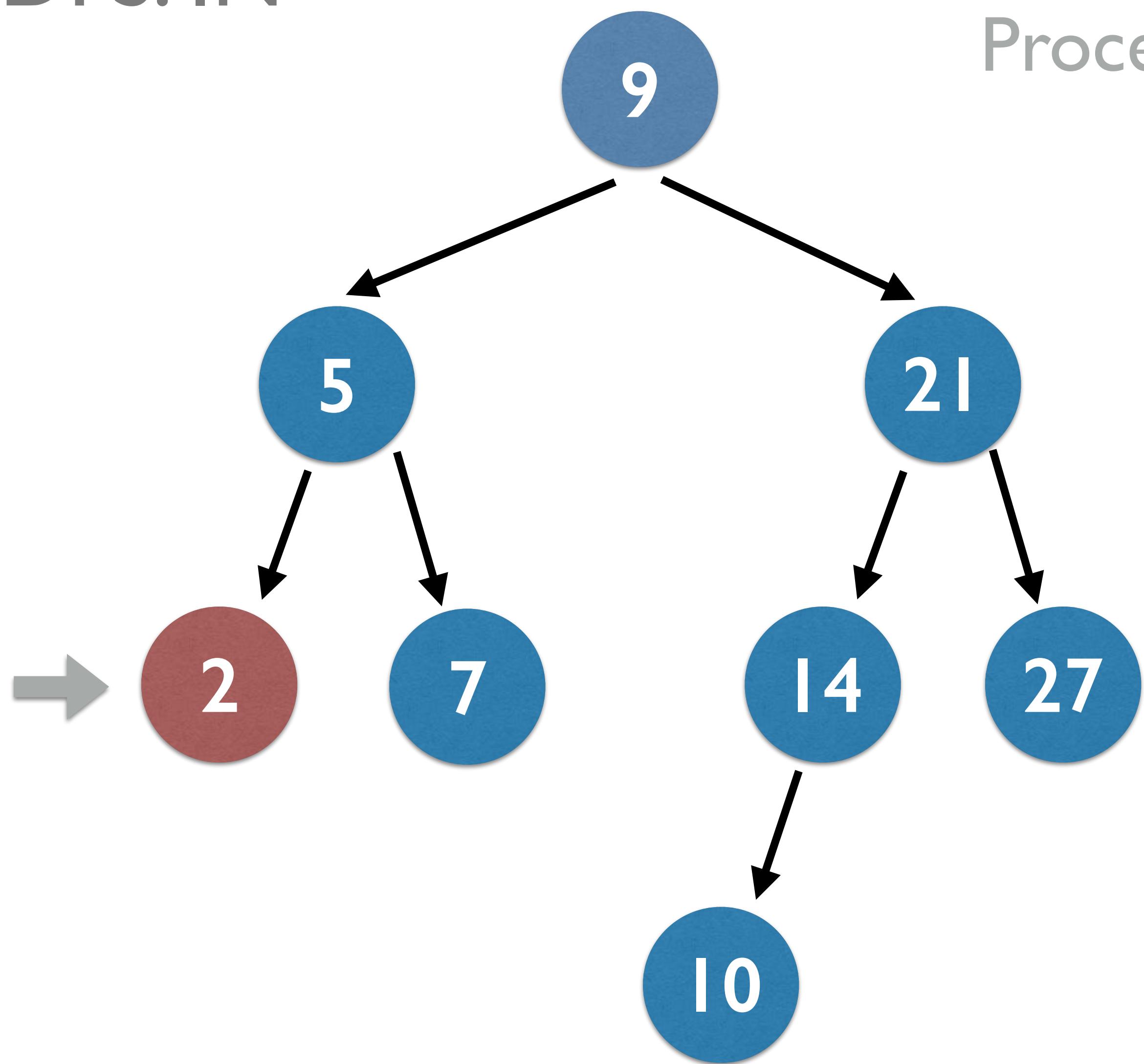
Process left · Process root · Process right

DFS: IN



Process left · Process root · Process right

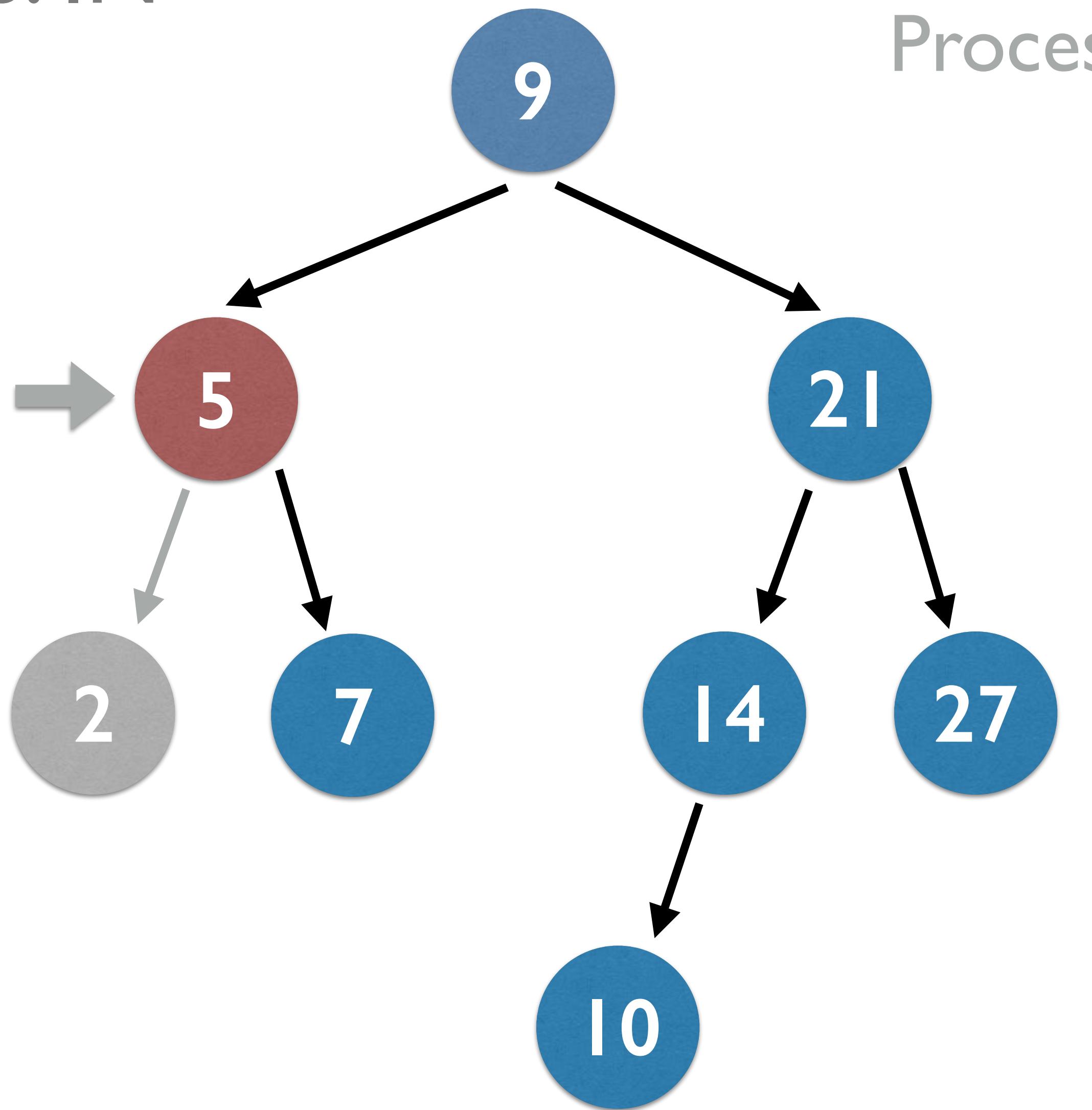
DFS: IN



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

2

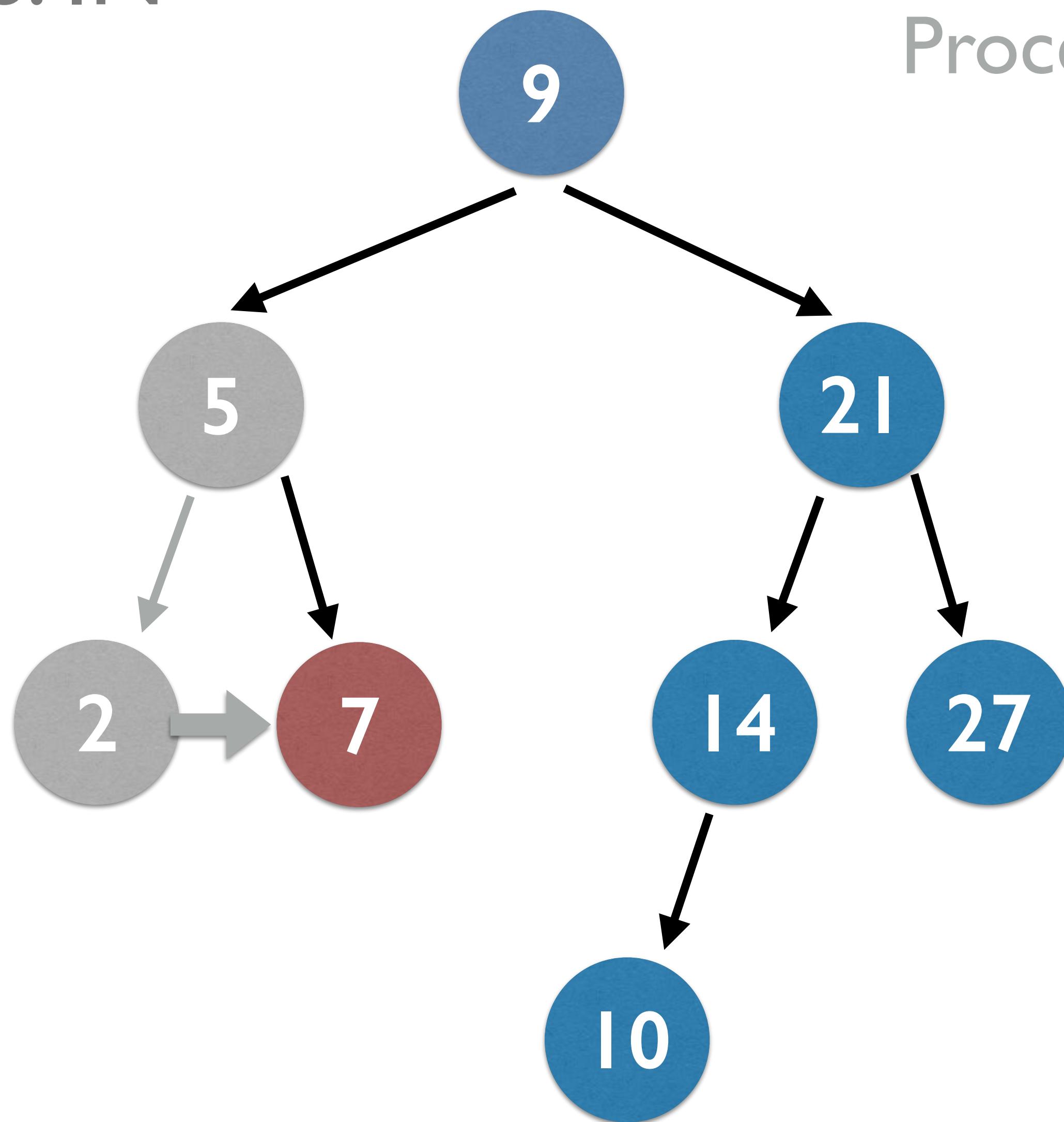
DFS: IN



Process left · Process root · Process right

2, 5

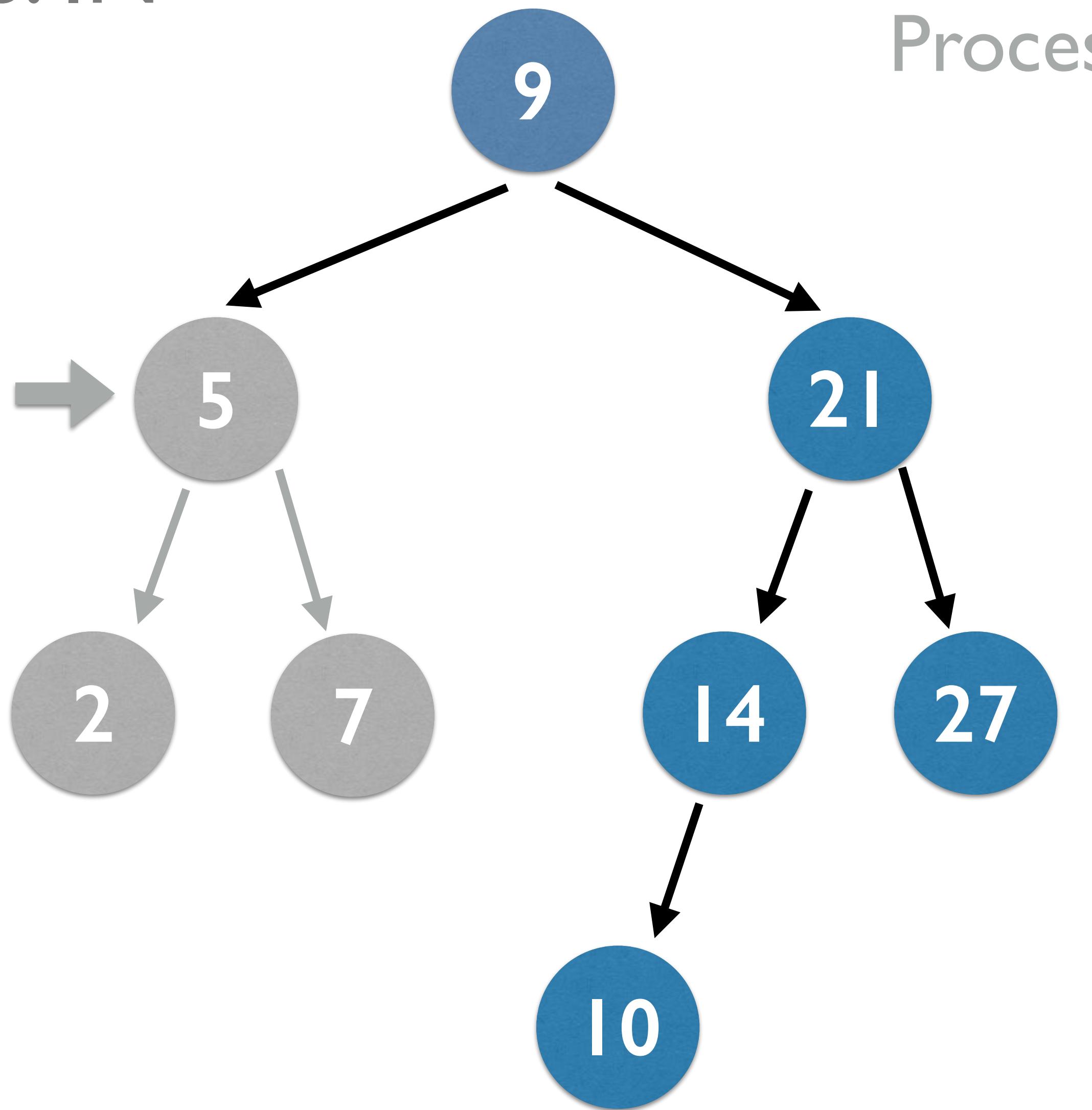
DFS: IN



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

2, 5, 7

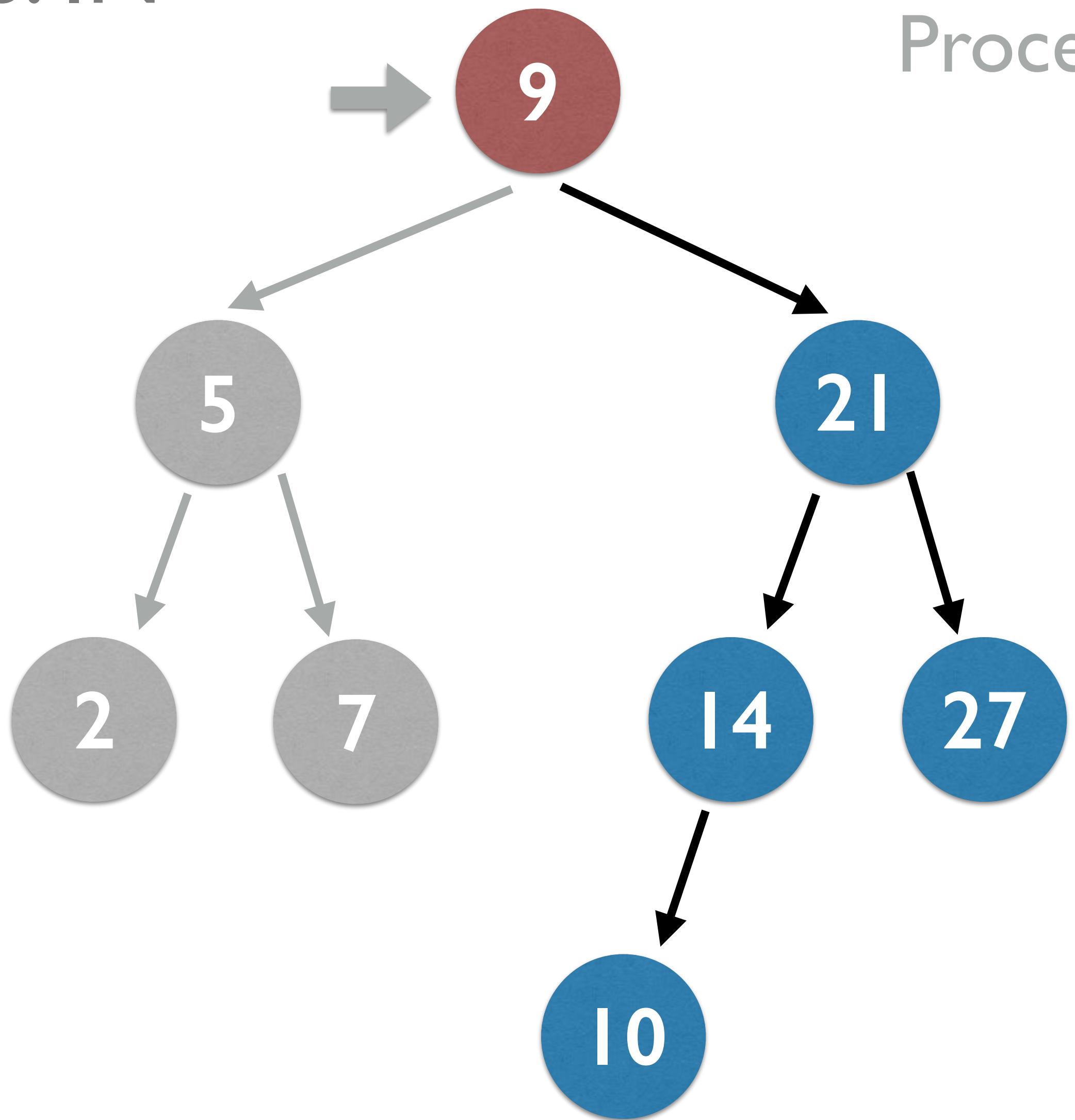
DFS: IN



Process left · Process root · Process right

2, 5, 7

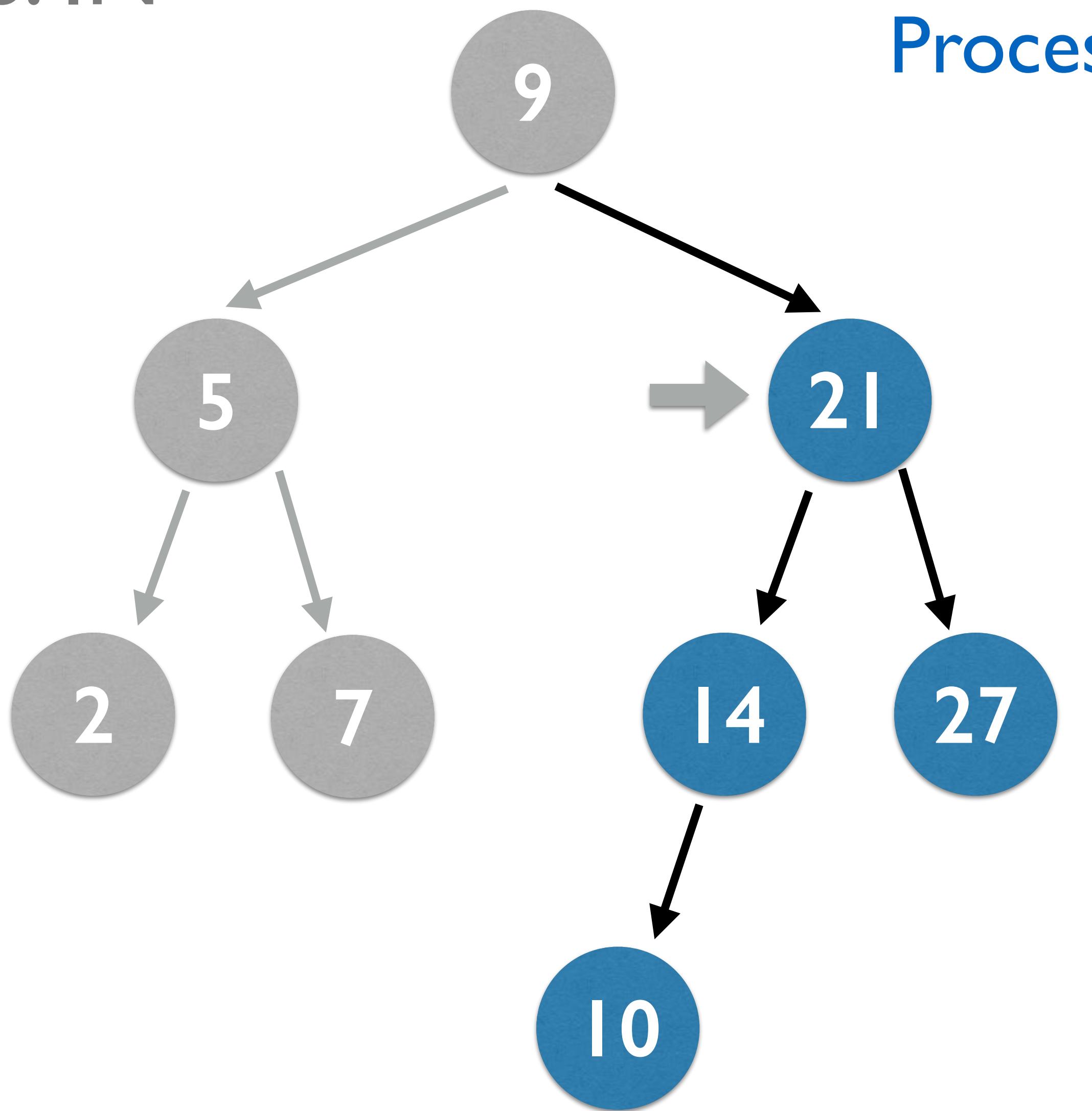
DFS: IN



Process left · Process root · Process right

2, 5, 7, 9

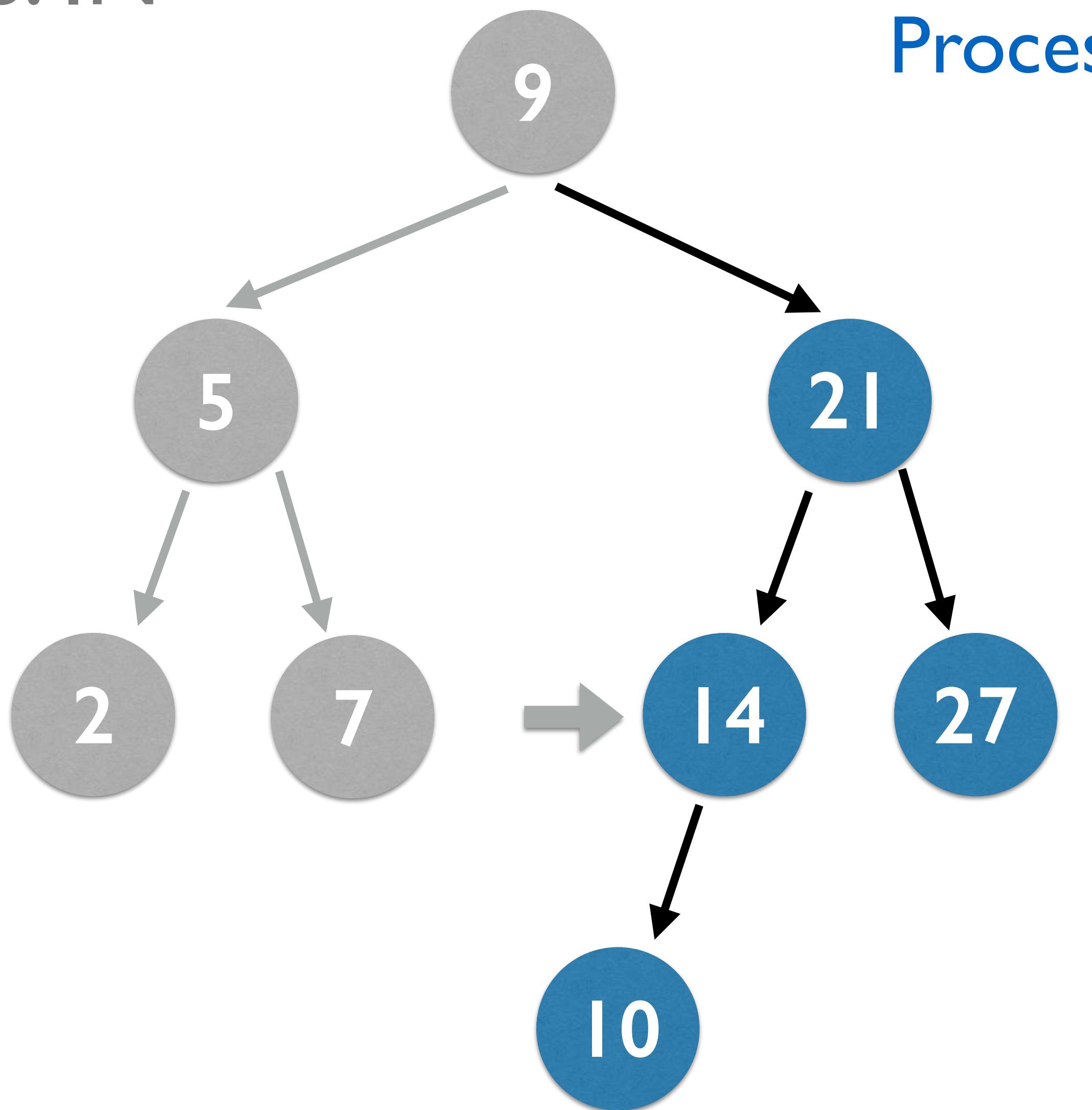
DFS: IN



Process left · Process root · Process right

2, 5, 7, 9

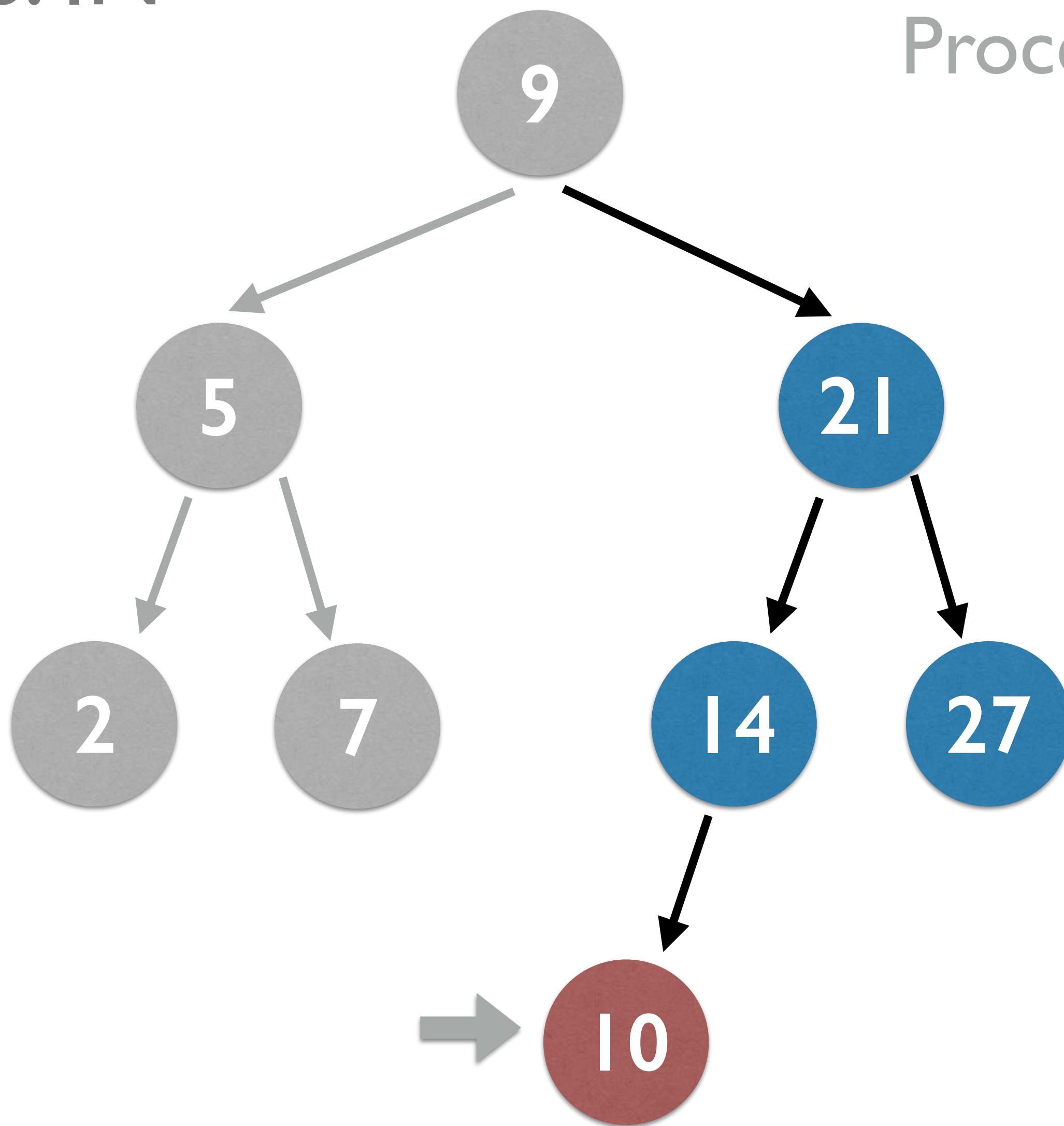
DFS: IN



Process left · Process root · Process right

2, 5, 7, 9

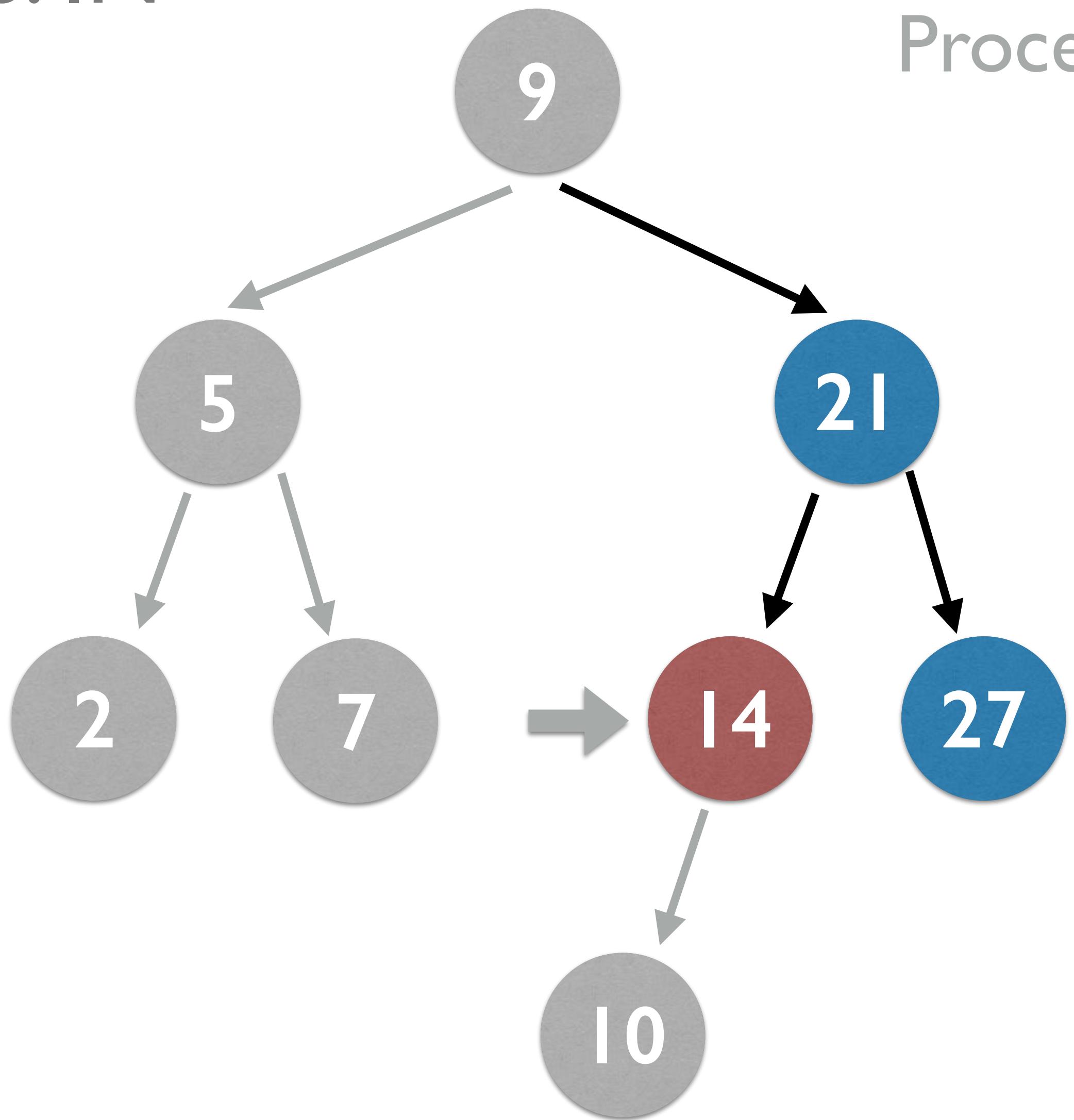
DFS: IN



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

2, 5, 7, 9, 10

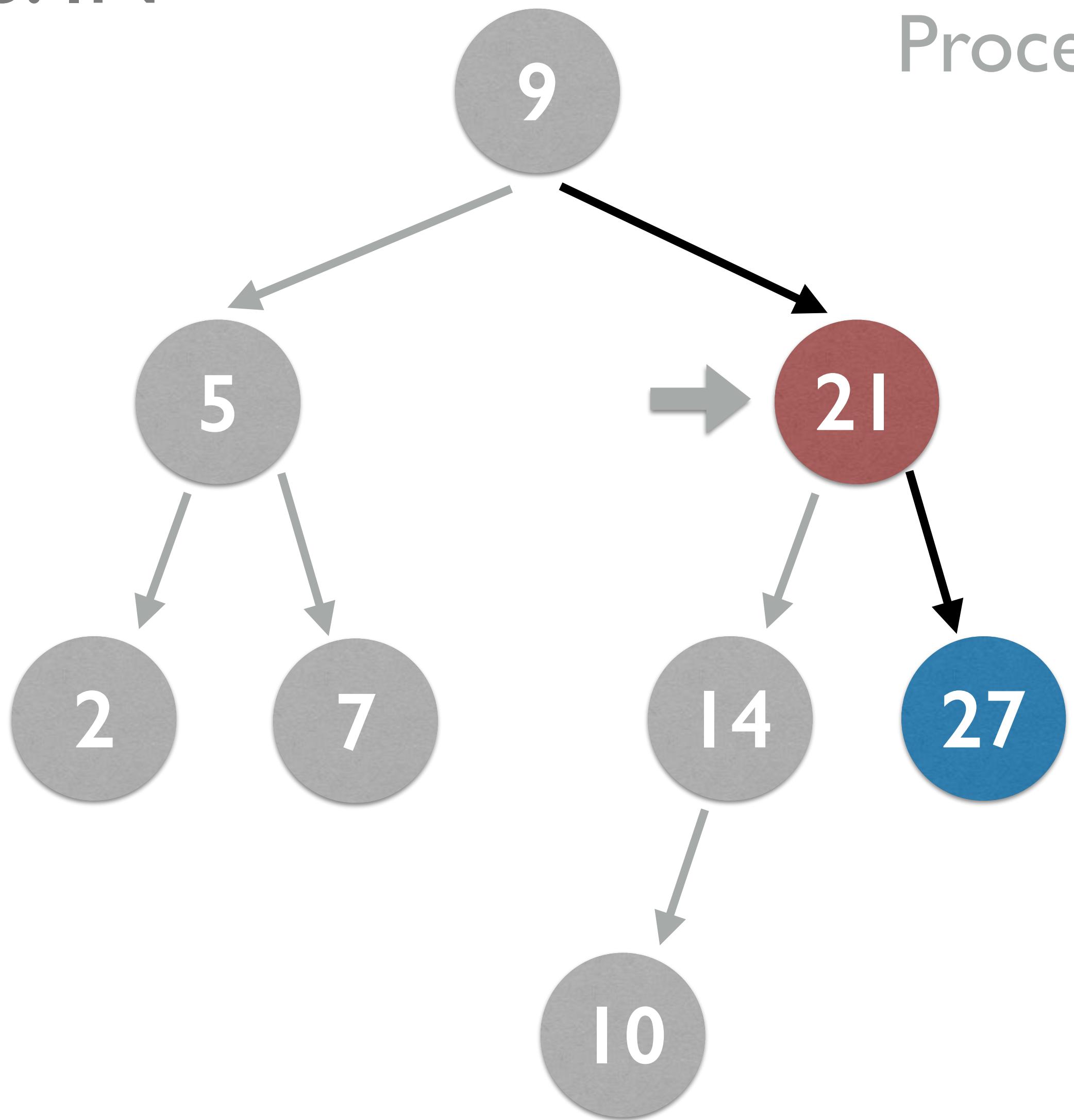
DFS: IN



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

2, 5, 7, 9, 10, 14

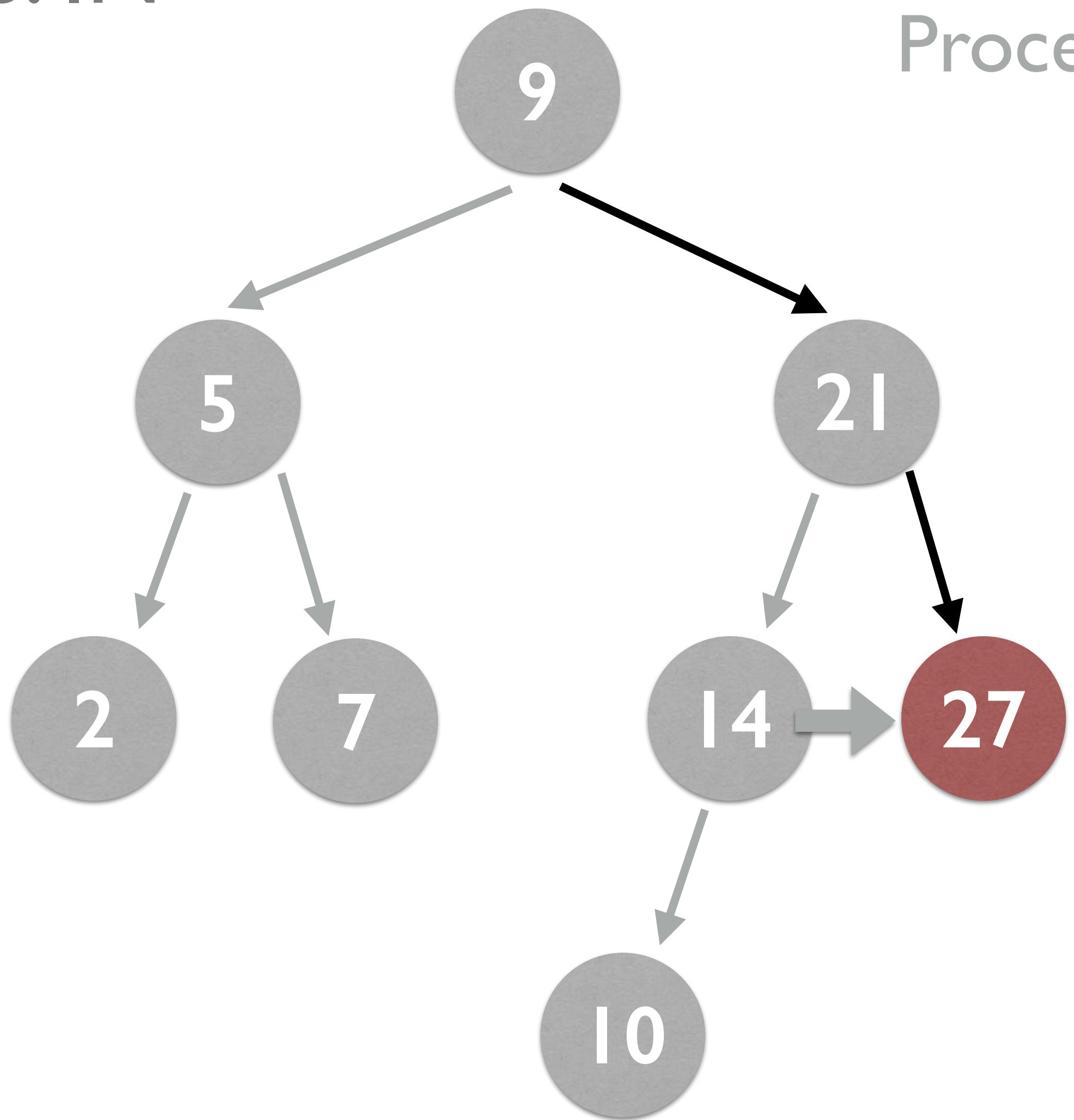
DFS: IN



Process left · Process root · Process right

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

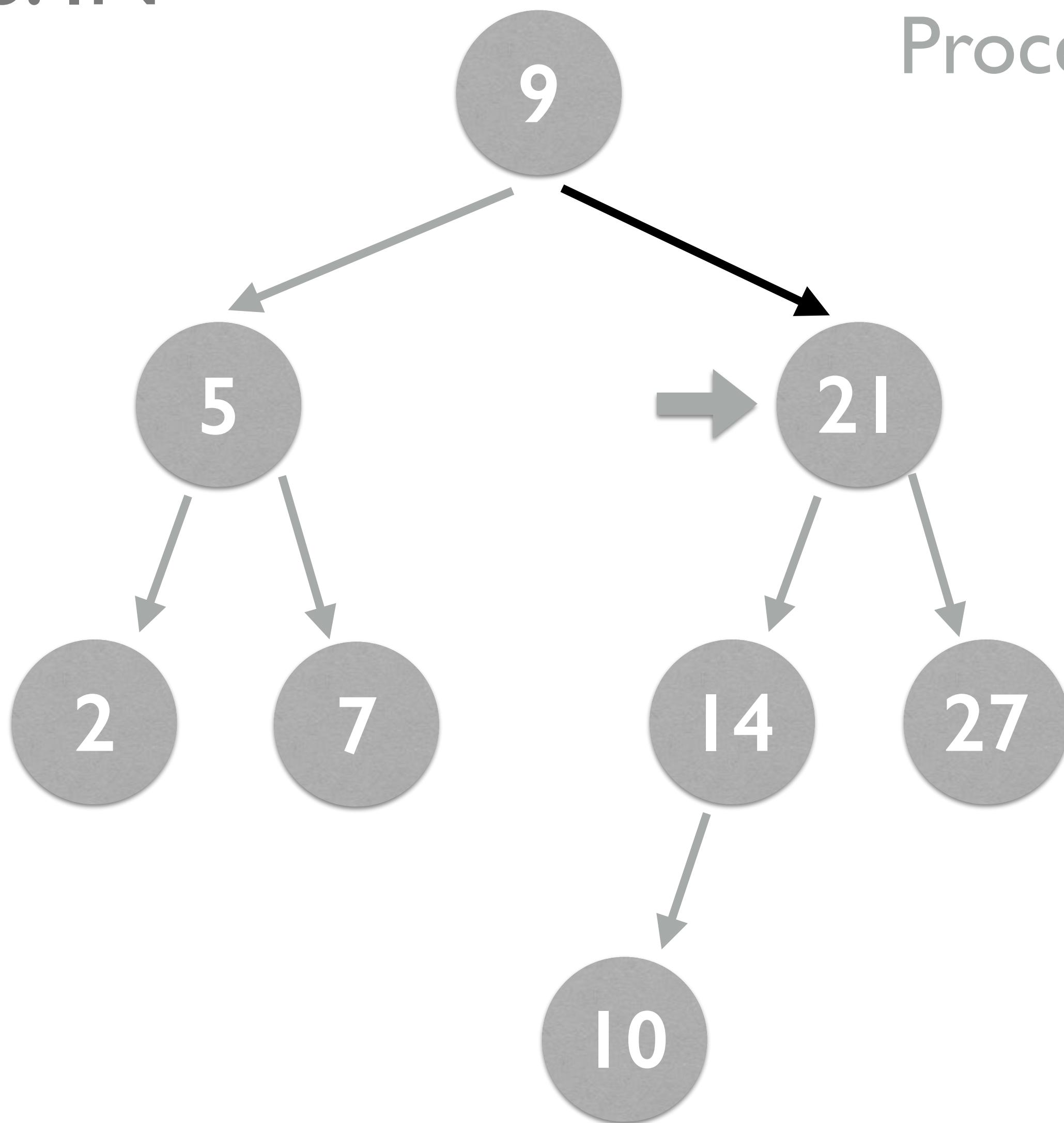
DFS: IN



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

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

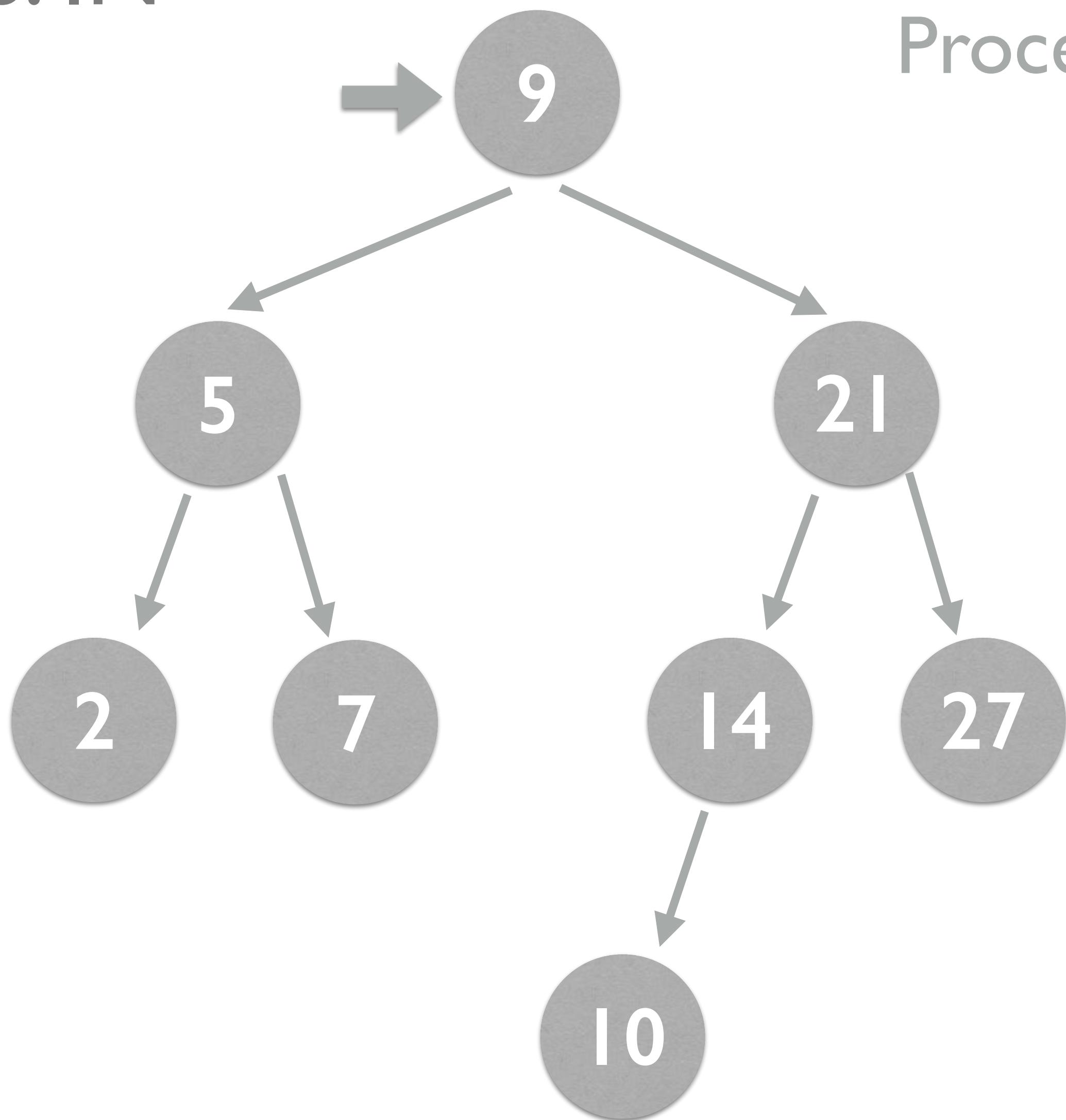
DFS: IN



Process left · Process root · Process right

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

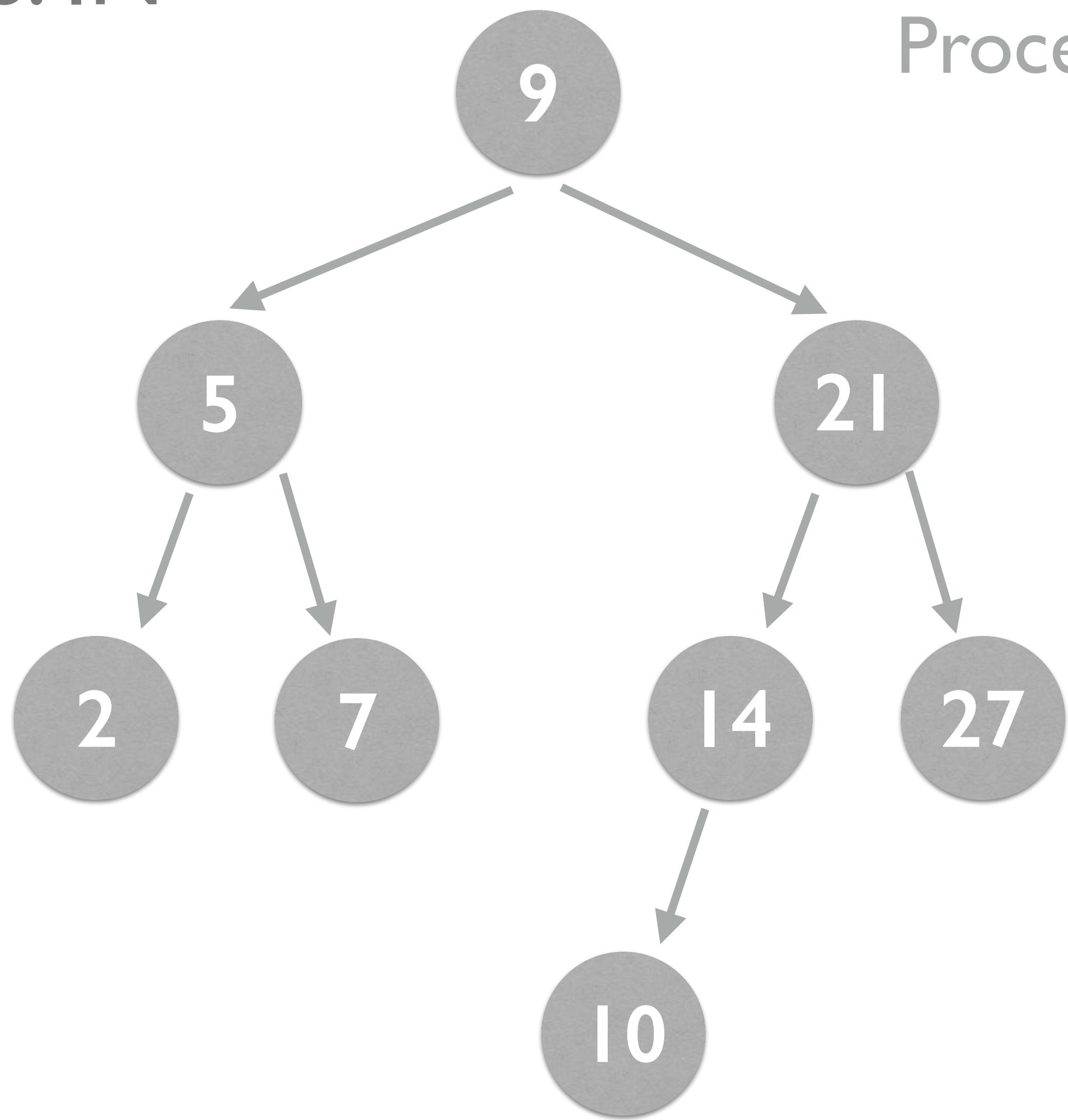
DFS: IN



Process left · Process root · Process right

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

DFS: IN



Process left · Process root · Process right

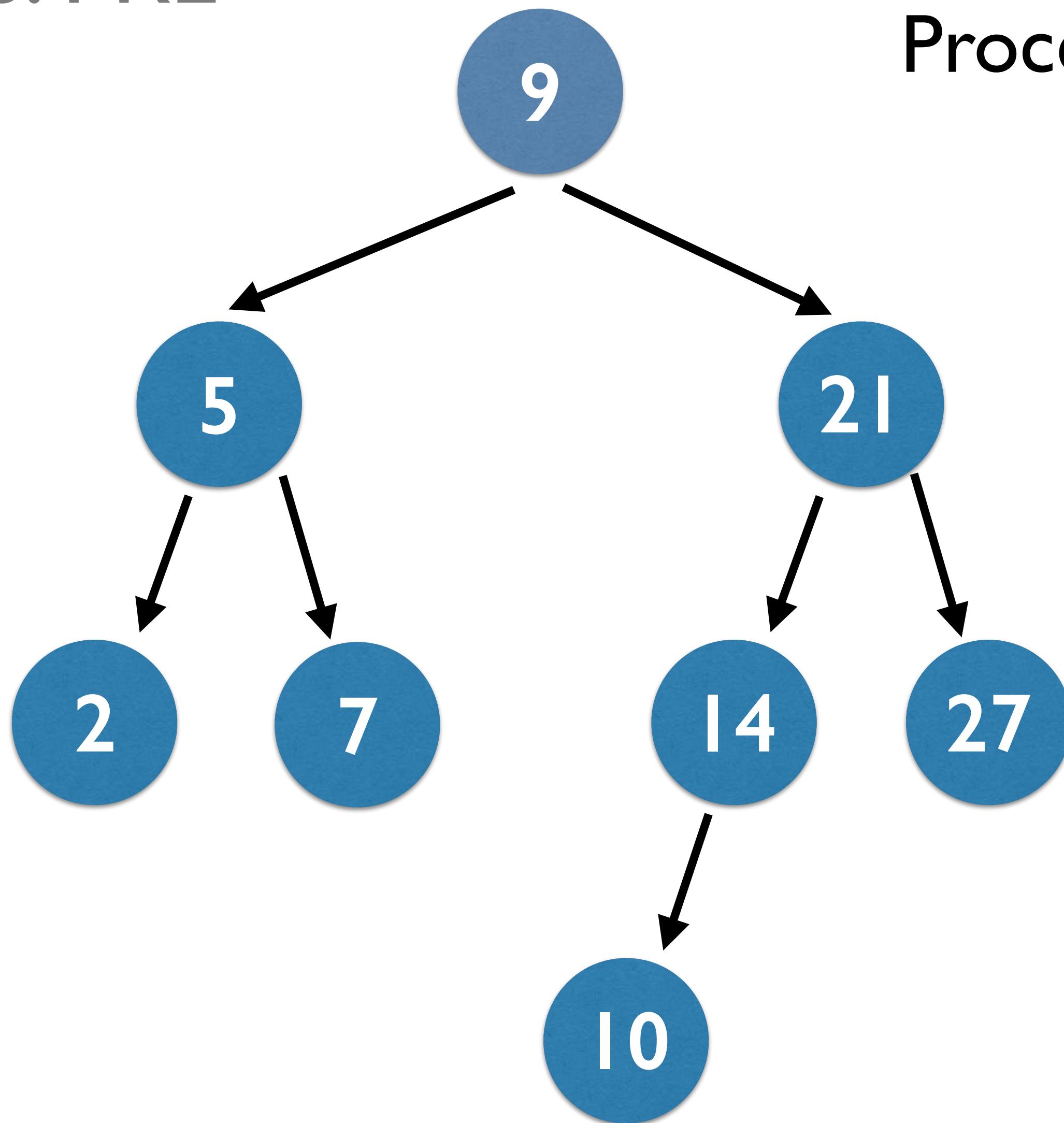
2, 5, 7, 9, 10, 14, 21, 27

- 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.

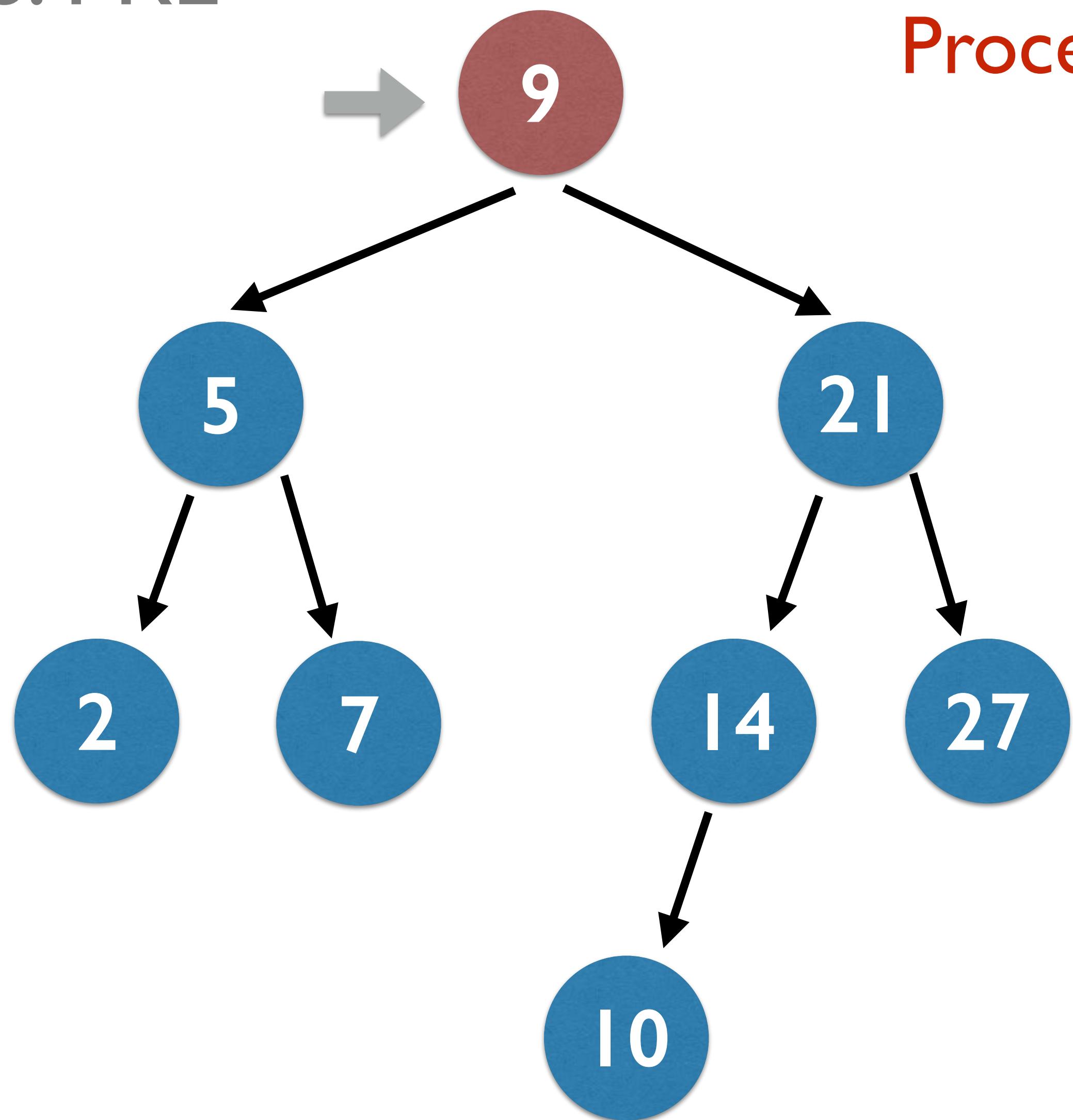
Depth First: Pre-Order

DFS: PRE

Process root · Process left · Process right



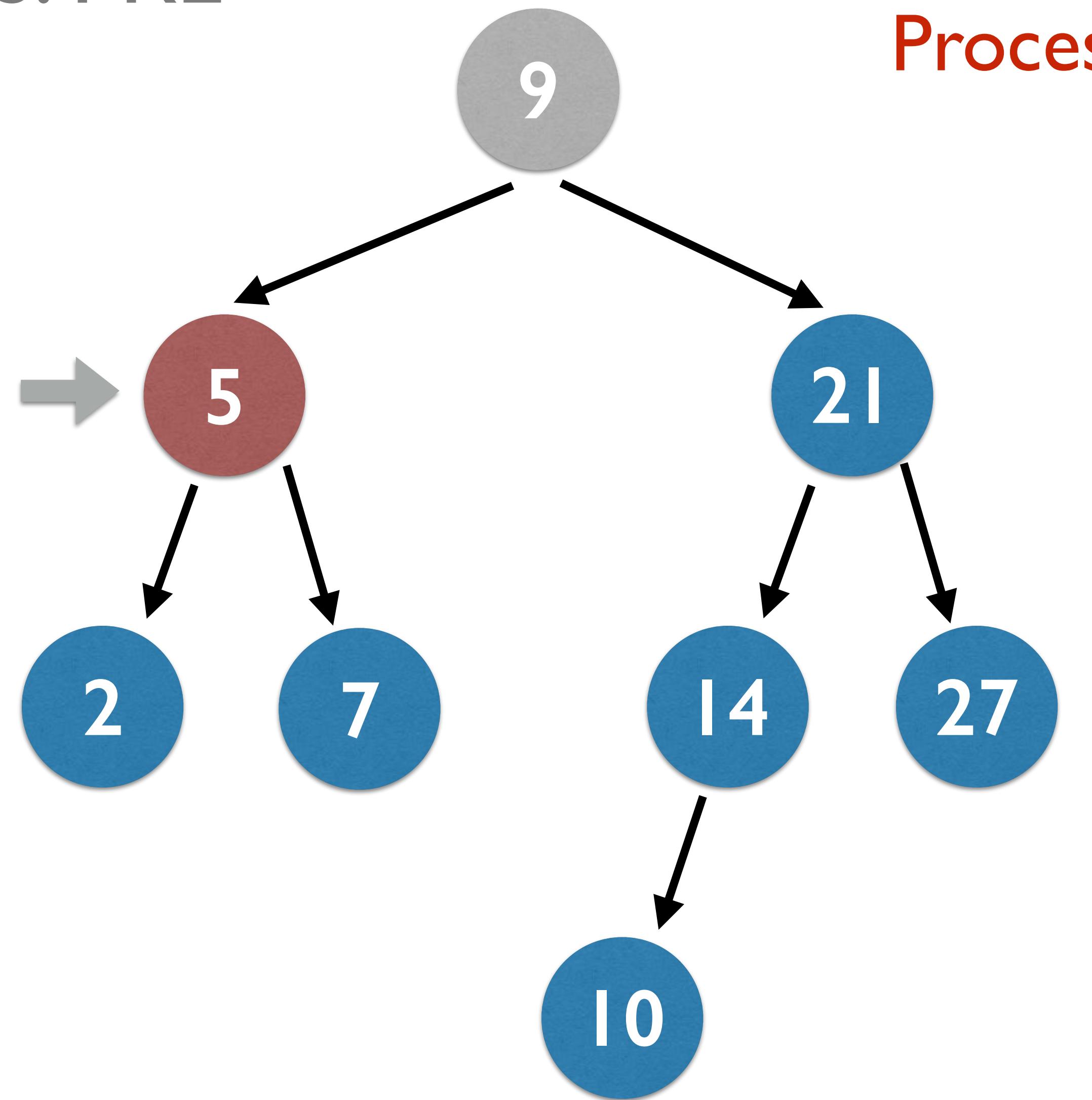
DFS: PRE



Process root • Process left • Process right

9

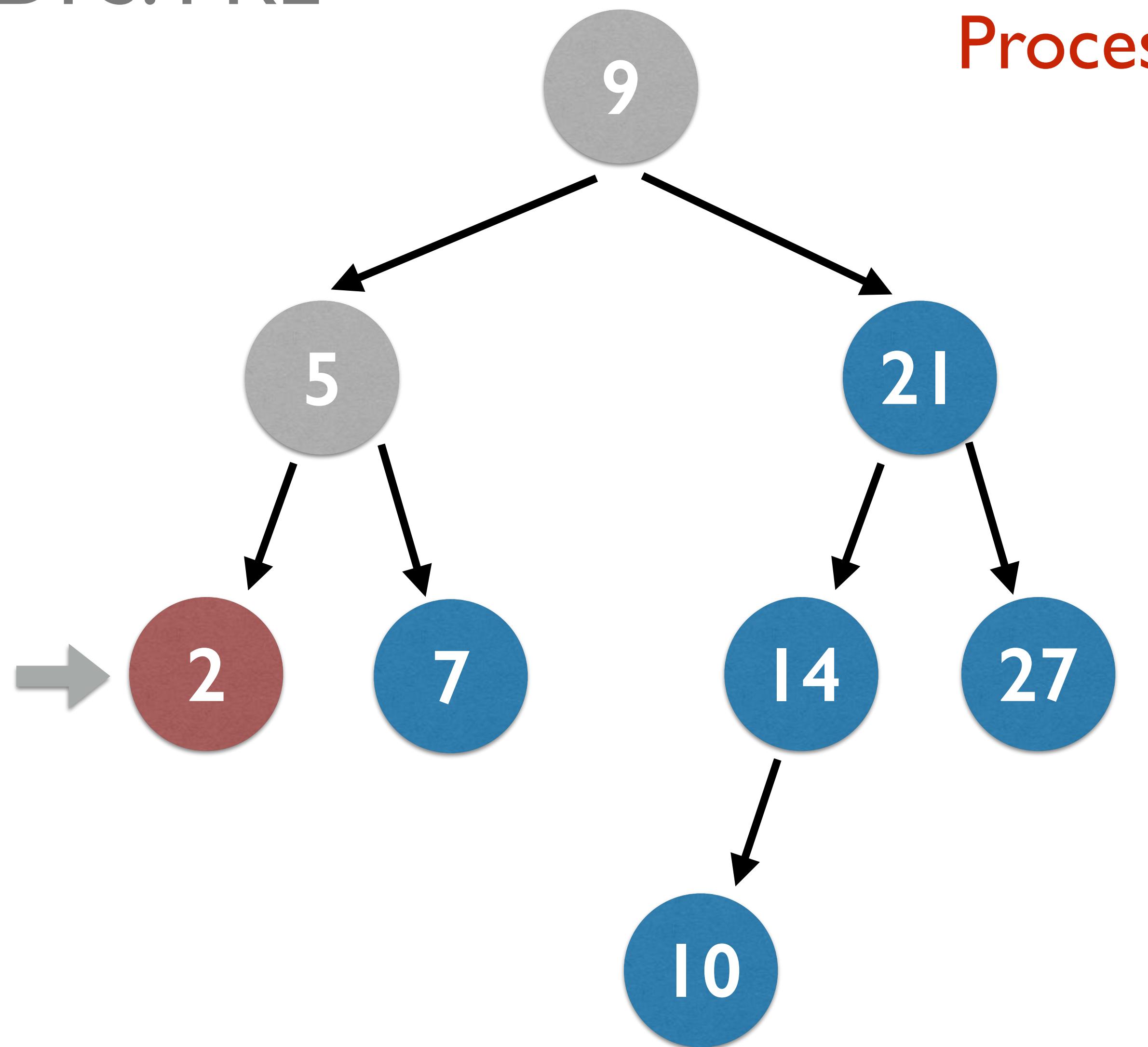
DFS: PRE



Process root • Process left • Process right

9, 5

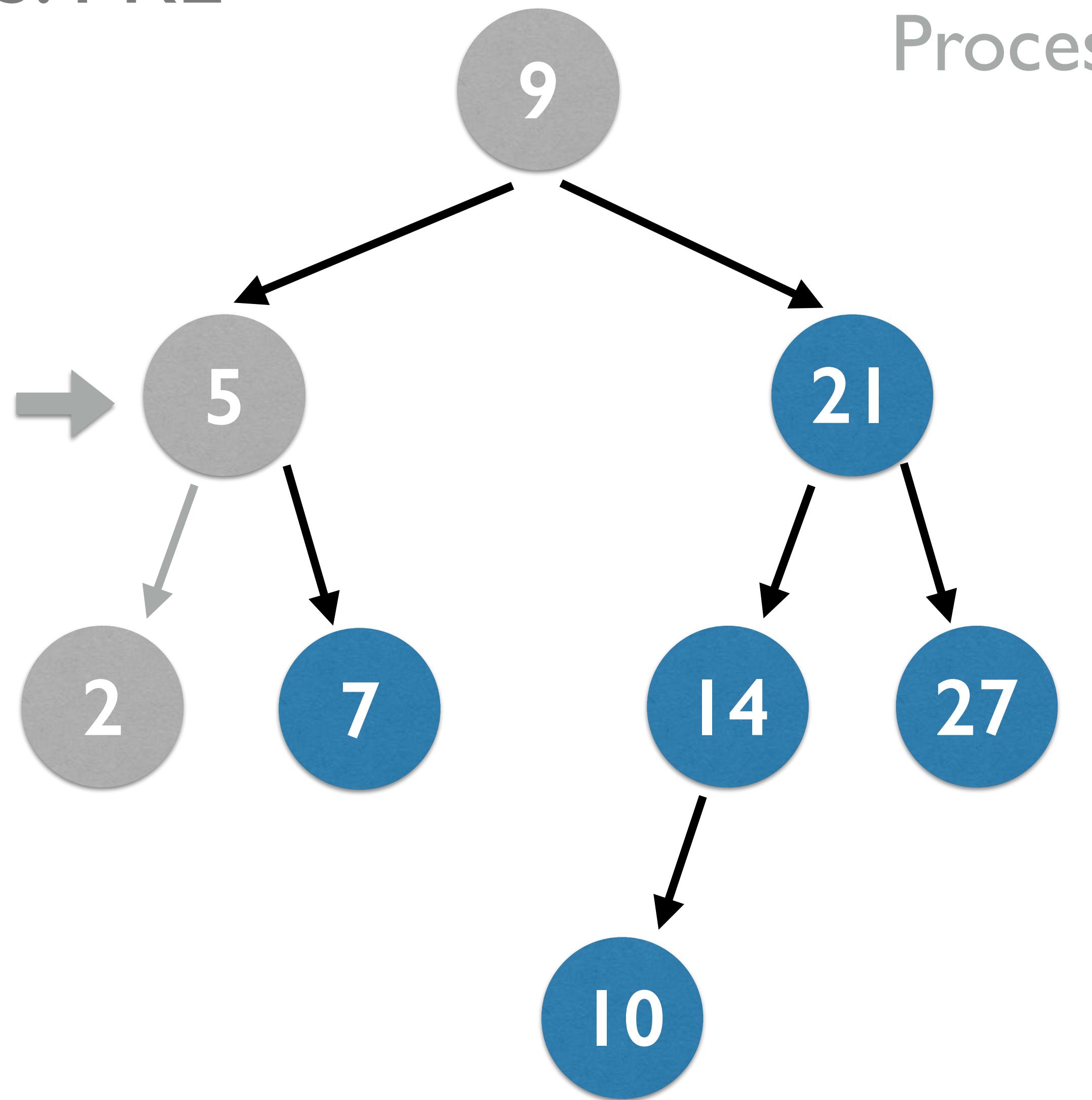
DFS: PRE



Process root • Process left • Process right

9, 5, 2

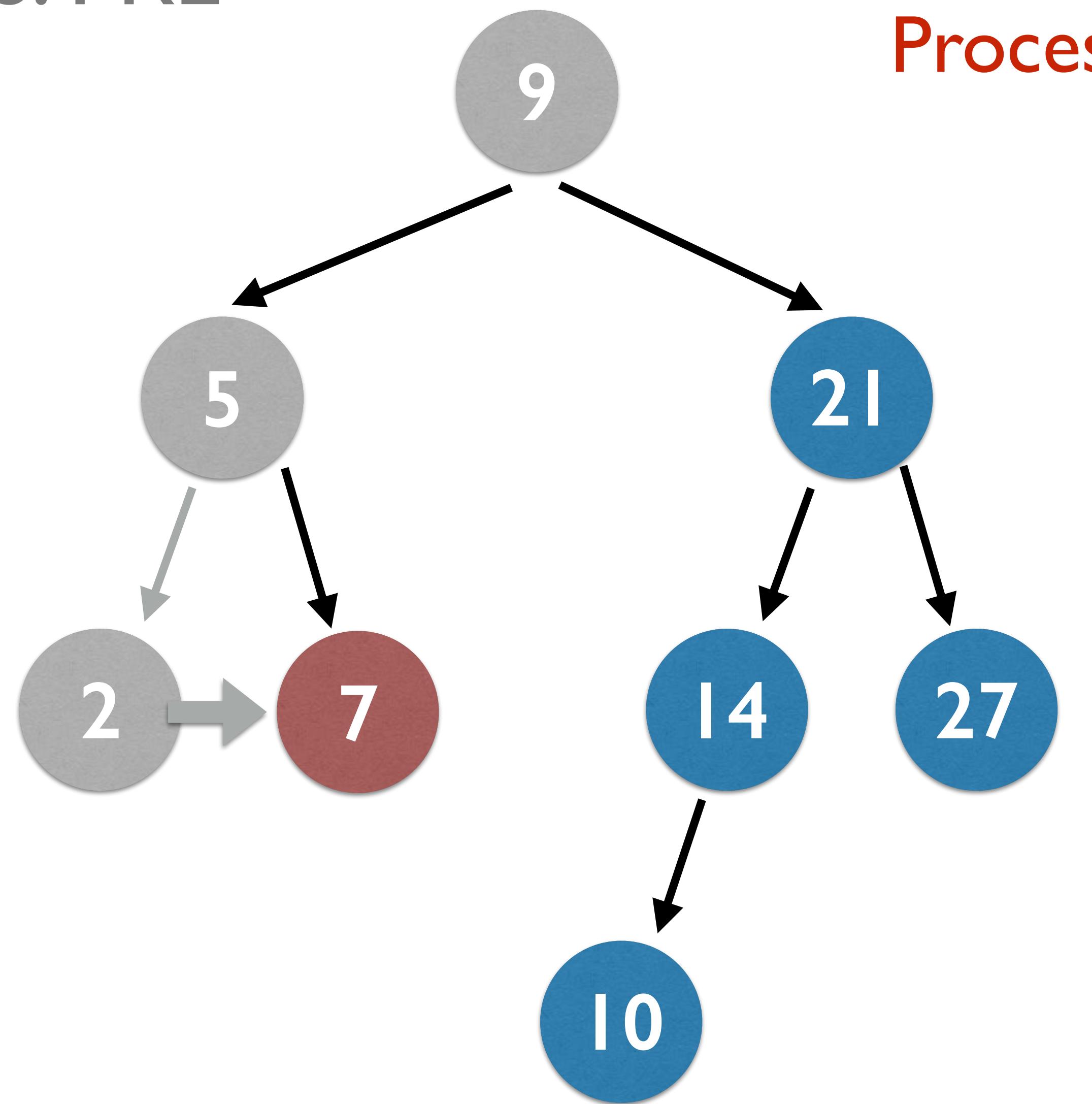
DFS: PRE



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

9, 5, 2

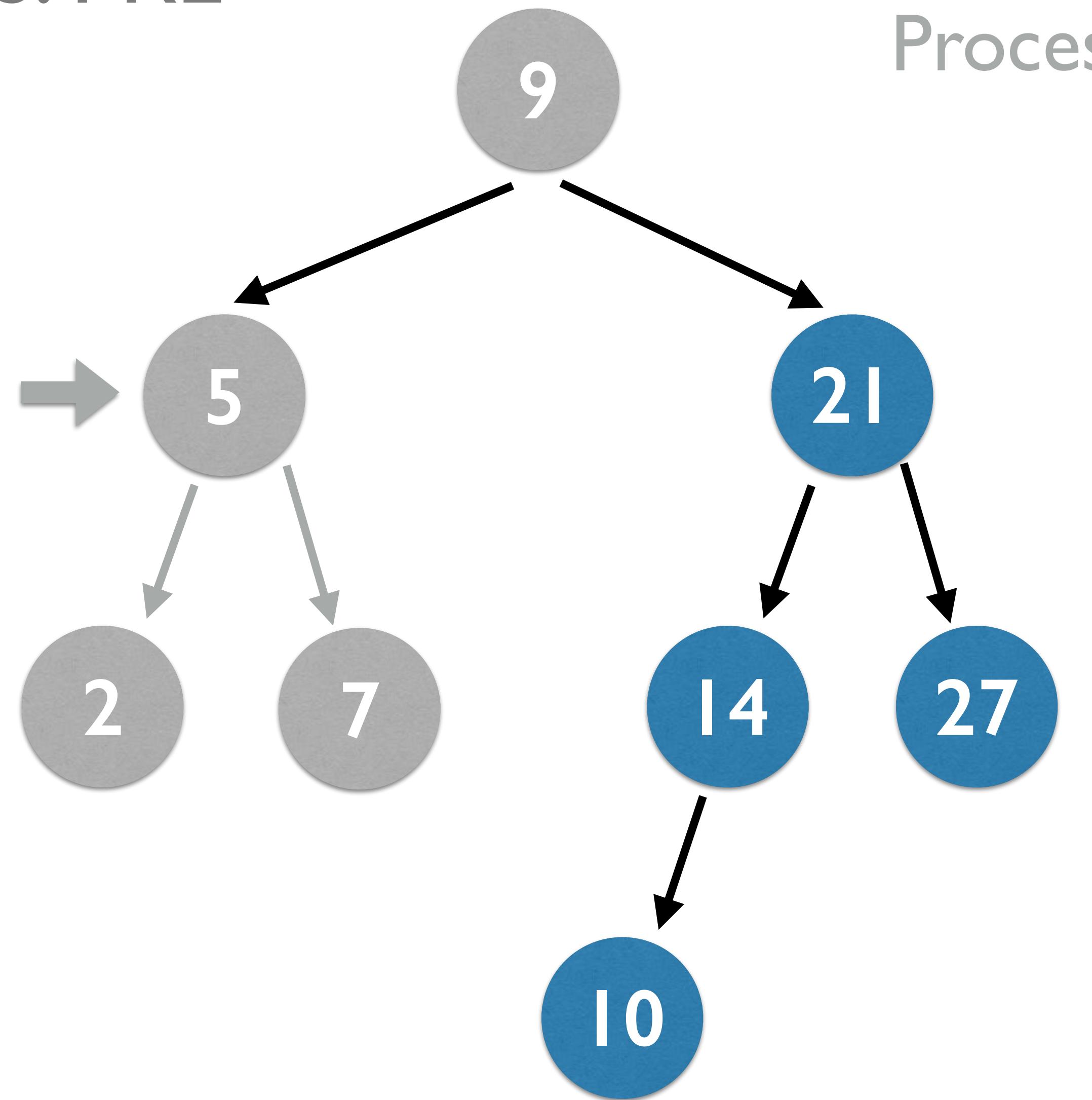
DFS: PRE



Process root • Process left • Process right

9, 5, 2, 7

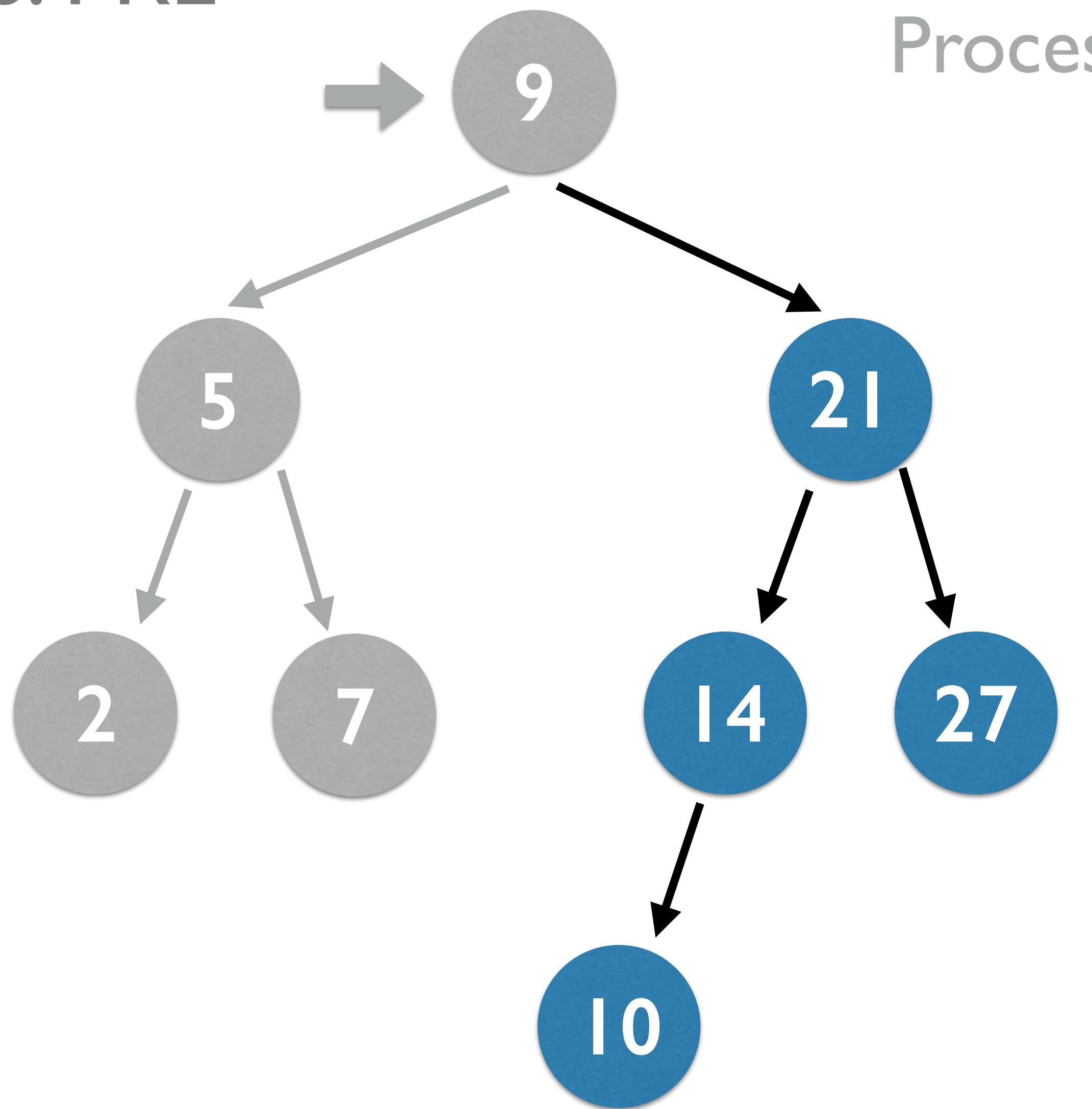
DFS: PRE



Process root · Process left · Process right

9, 5, 2, 7

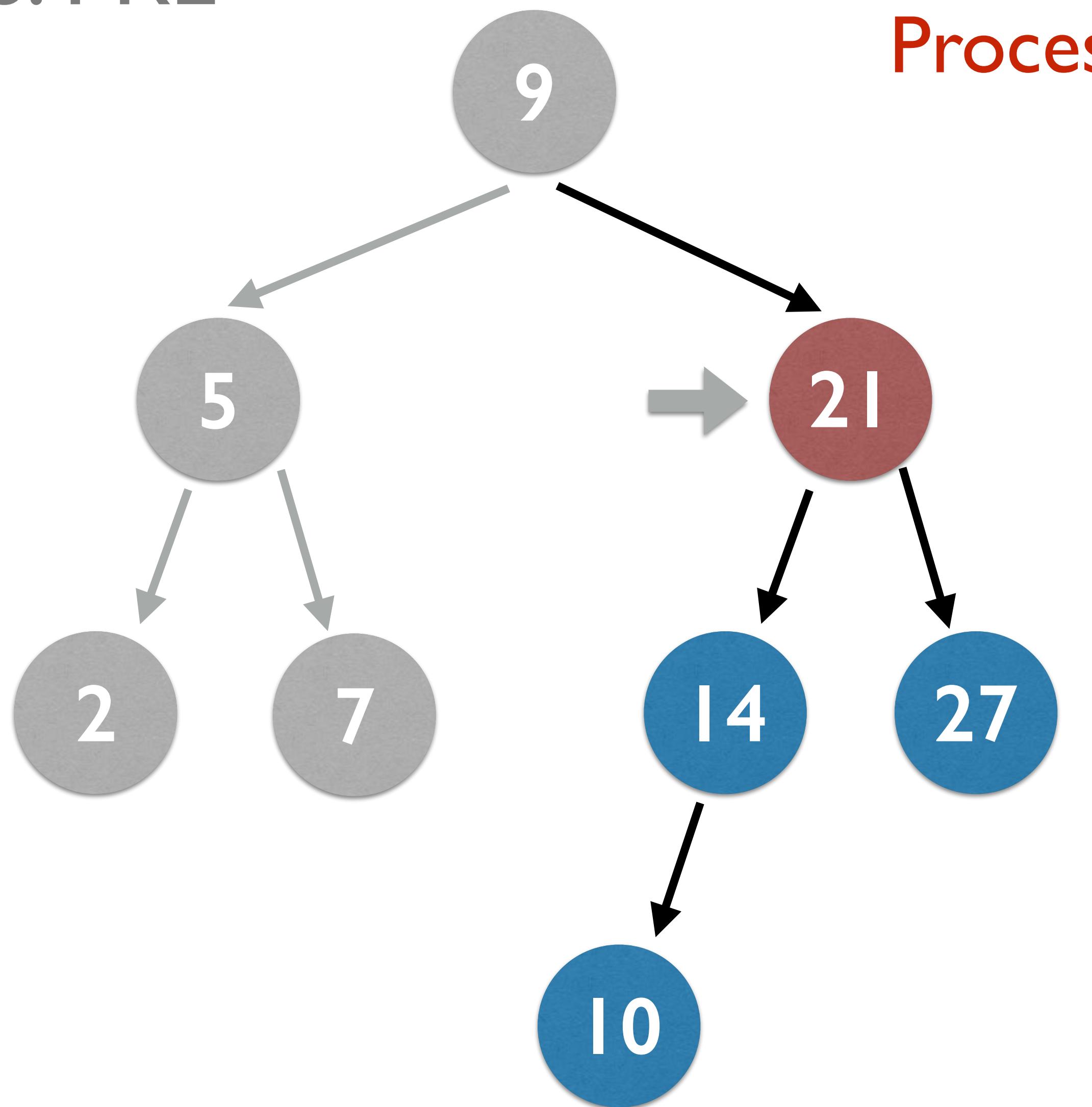
DFS: PRE



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

9, 5, 2, 7

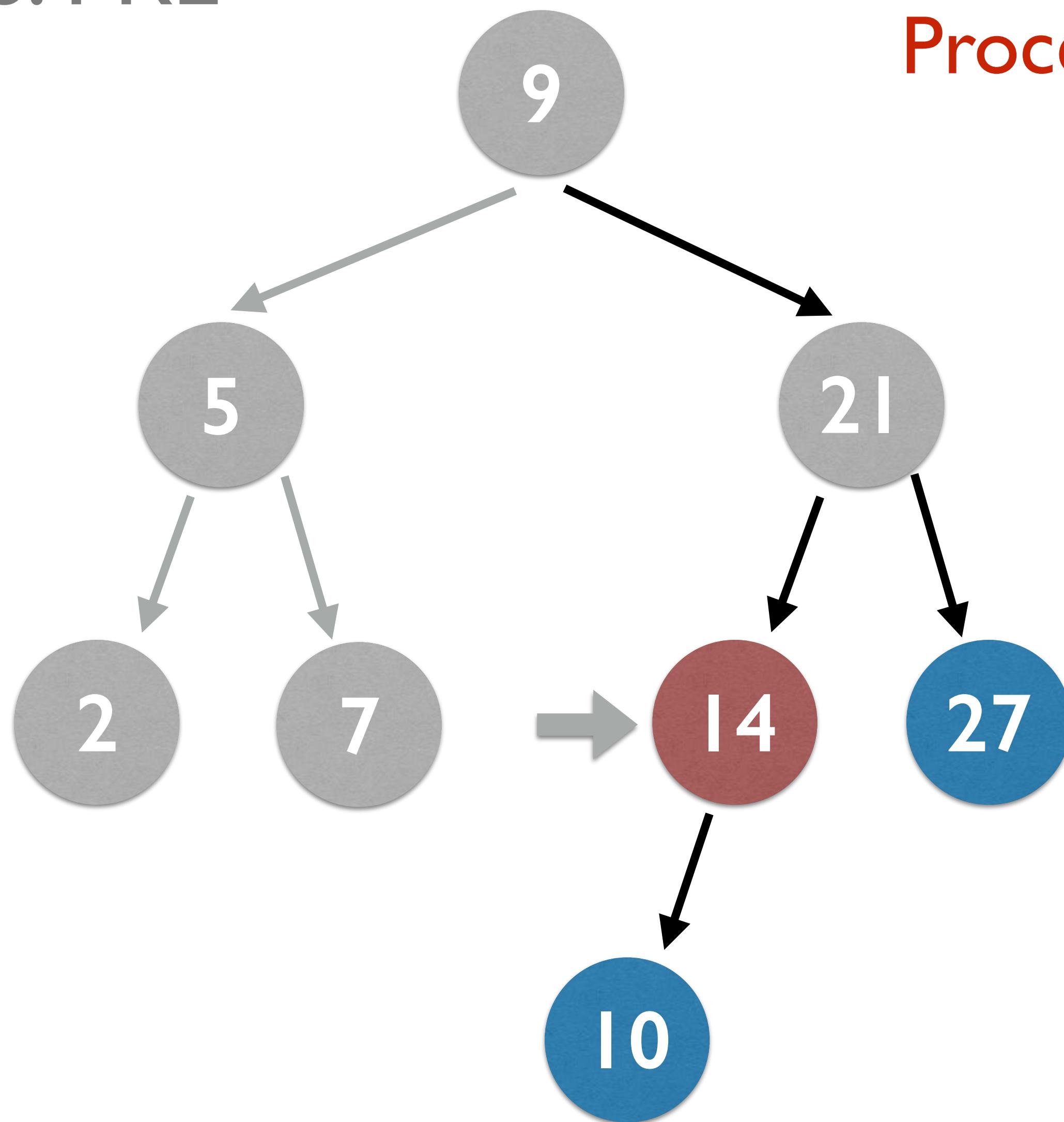
DFS: PRE



Process root · Process left · Process right

9, 5, 2, 7, 21

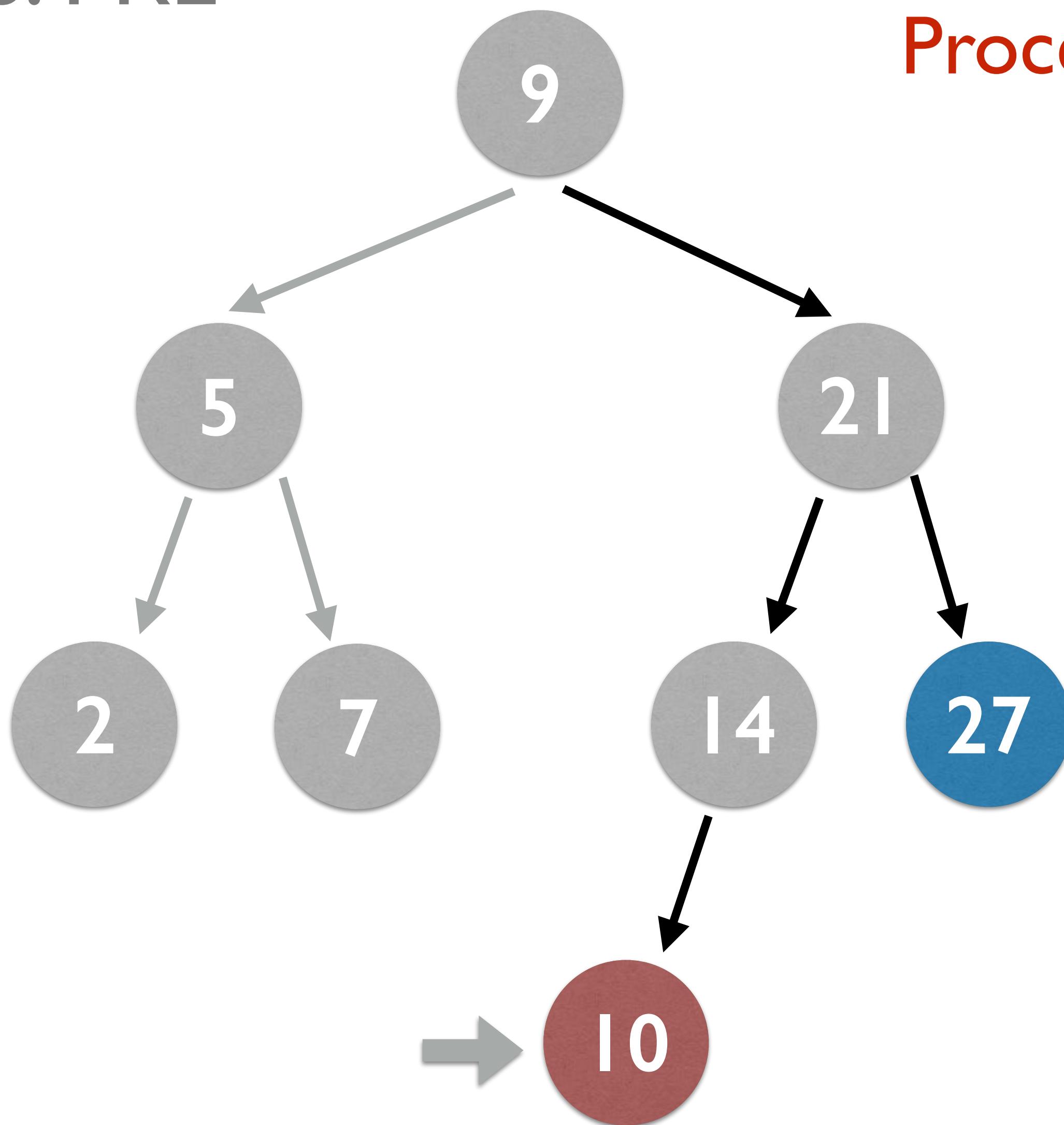
DFS: PRE



Process root · Process left · Process right

9, 5, 2, 7, 21, 14

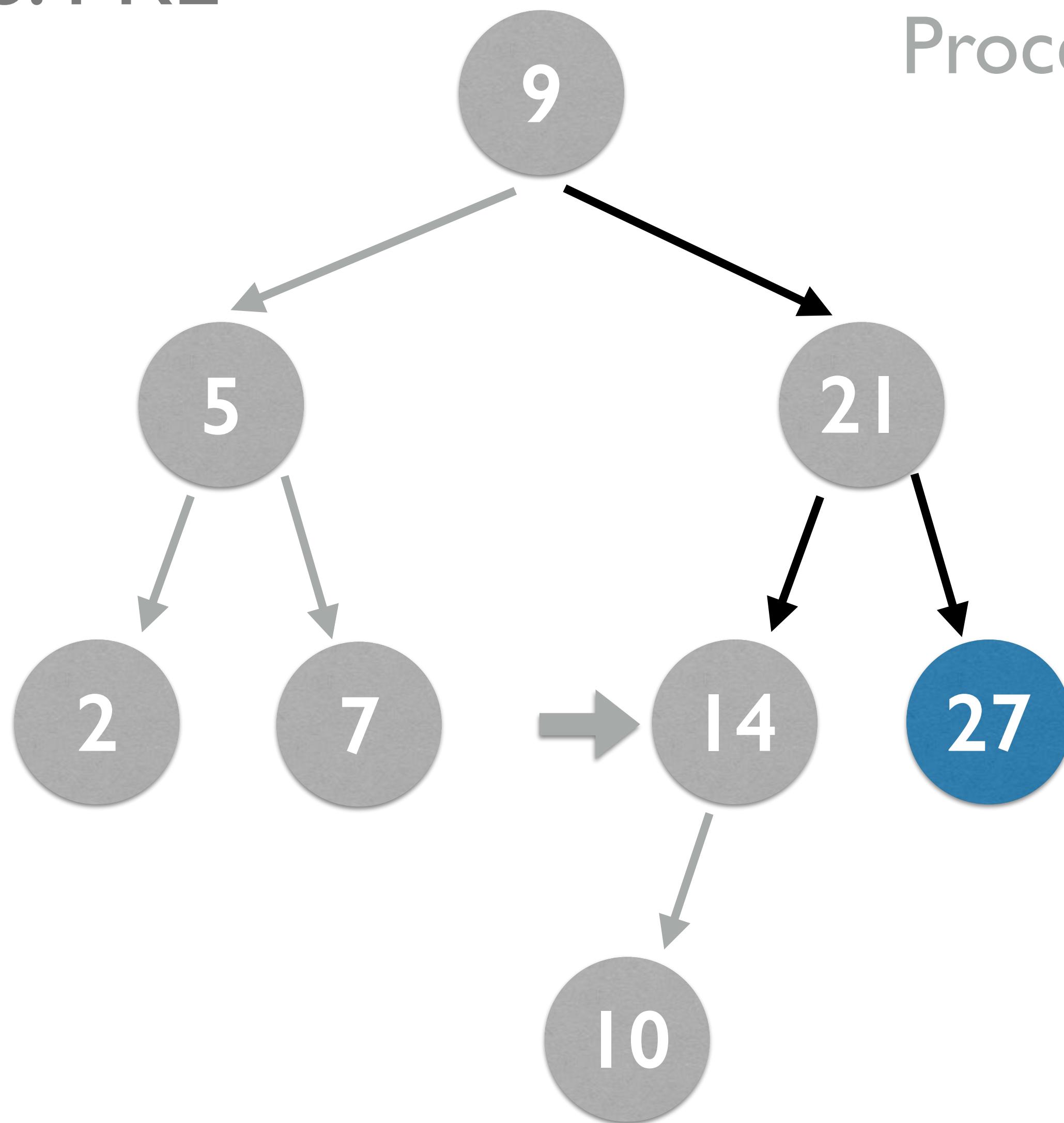
DFS: PRE



Process root • Process left • Process right

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

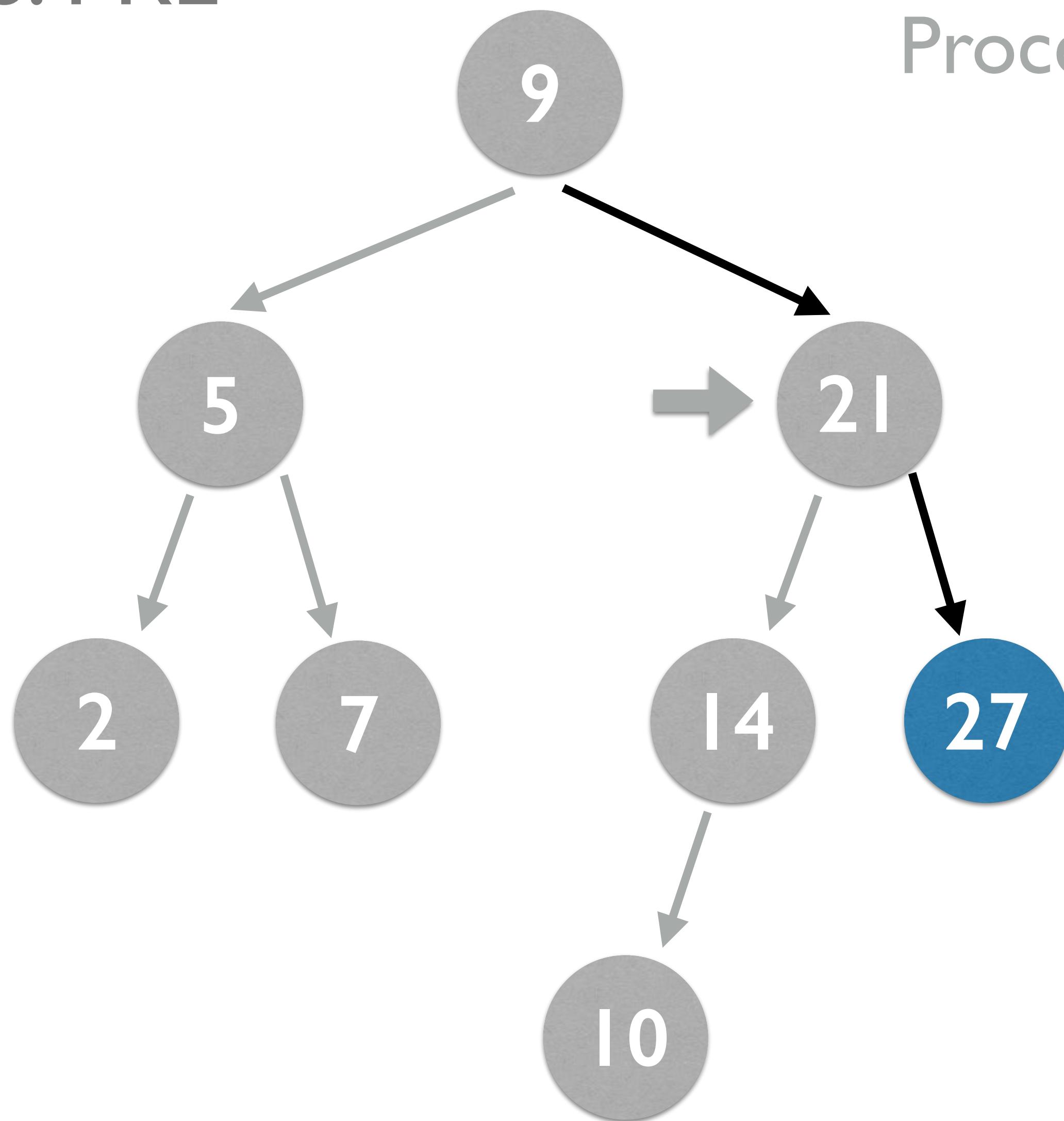
DFS: PRE



Process root · Process left · Process right

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

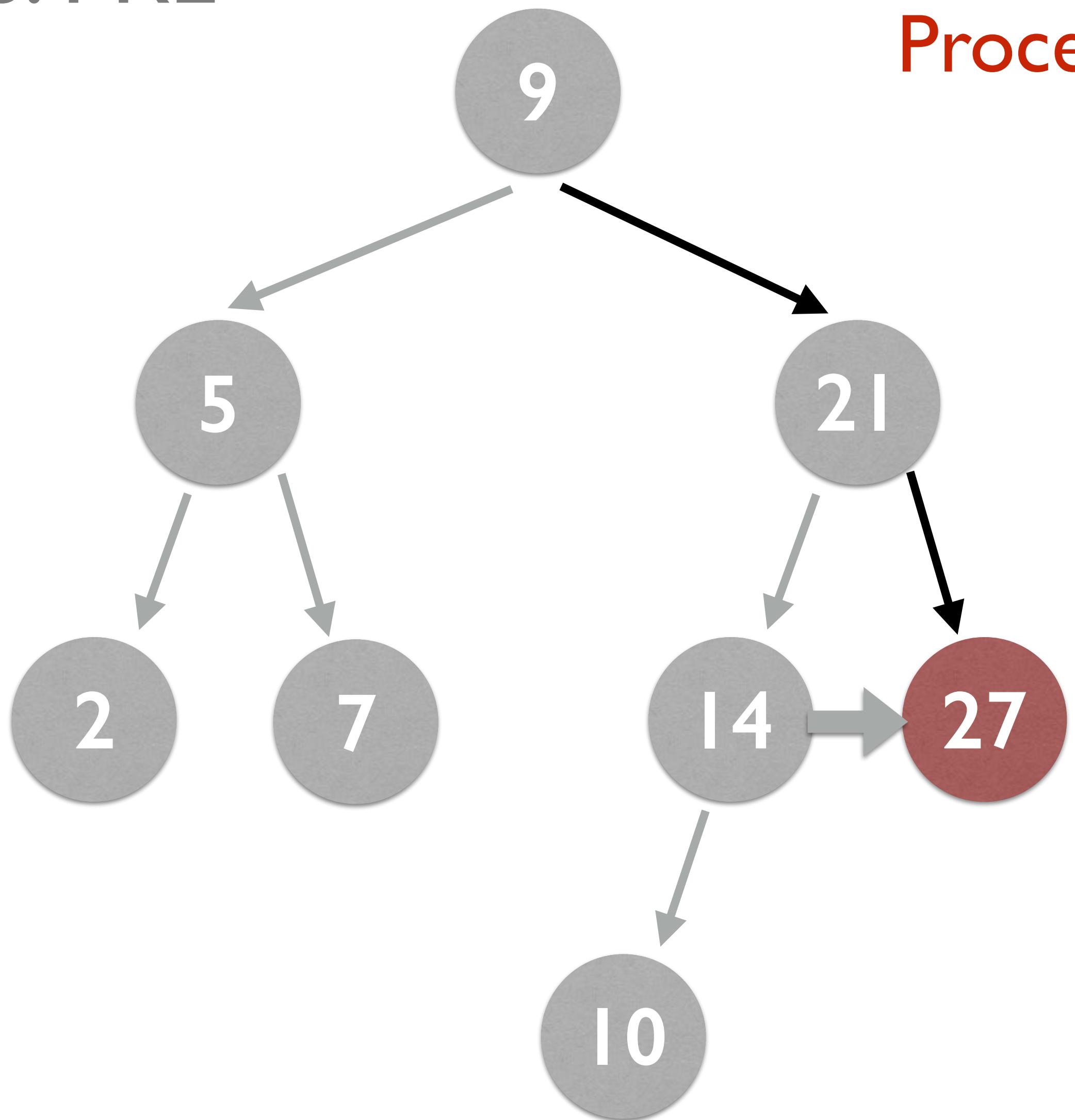
DFS: PRE



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

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

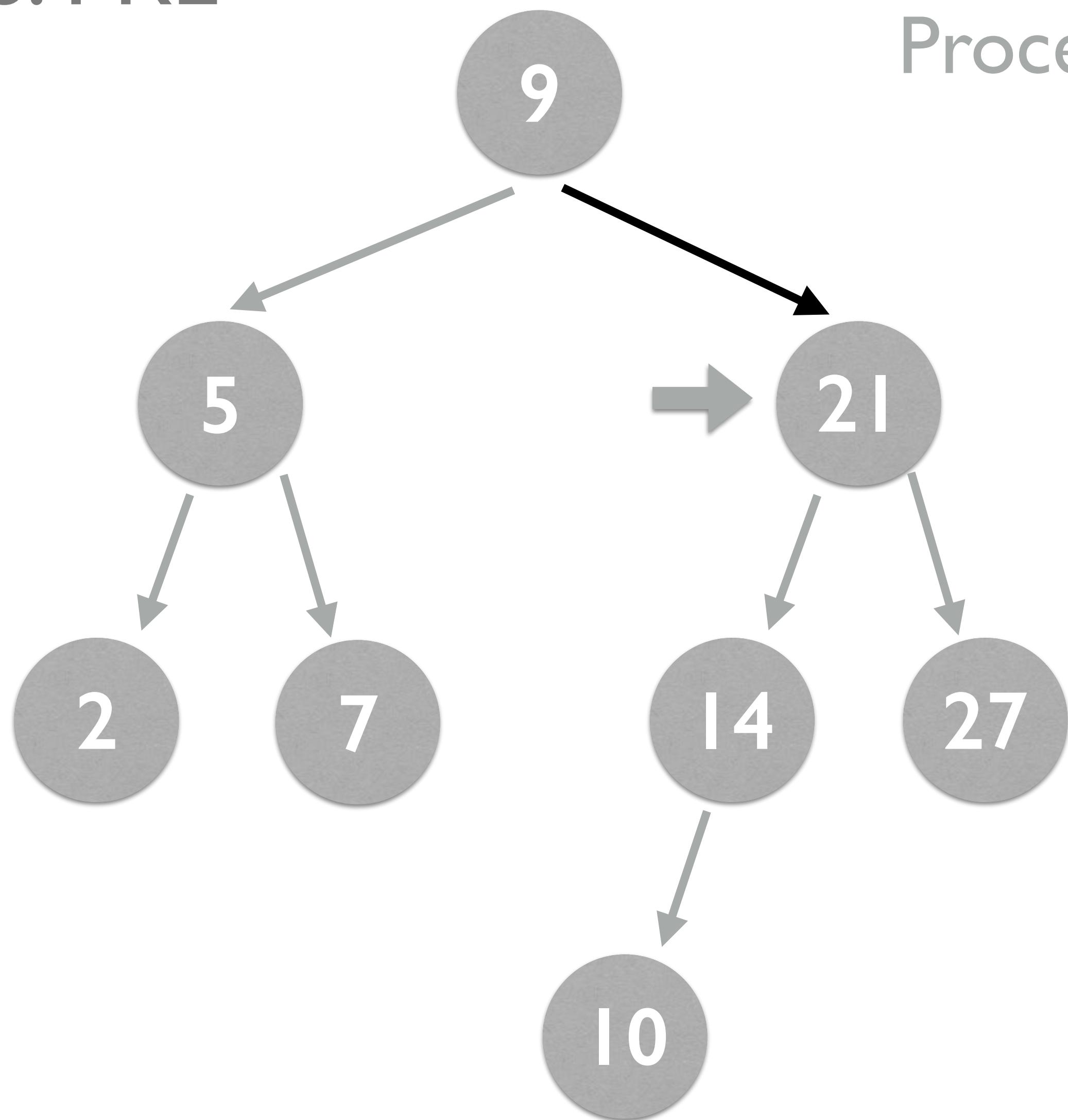
DFS: PRE



Process root · Process left · Process right

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

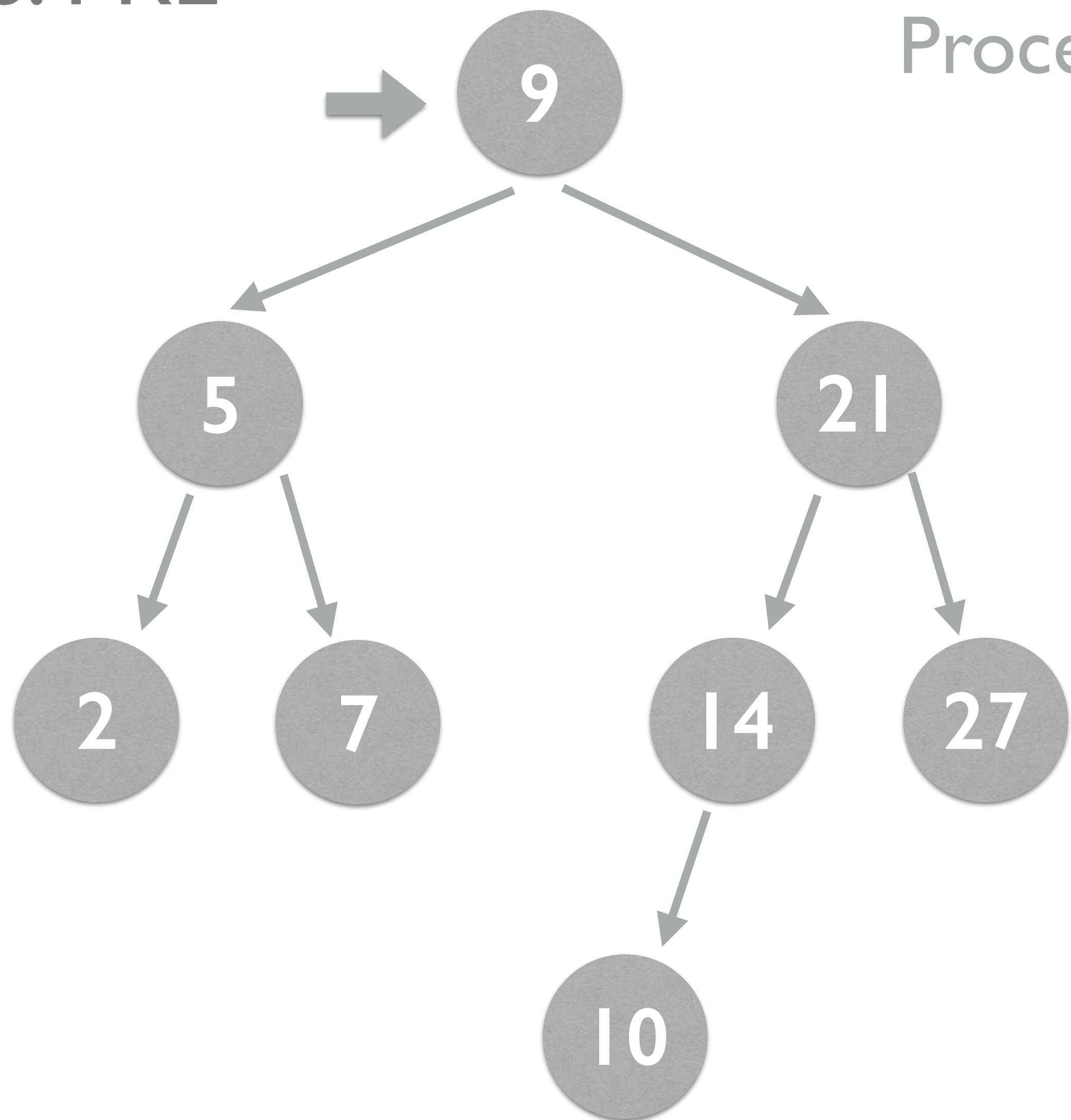
DFS: PRE



Process root · Process left · Process right

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

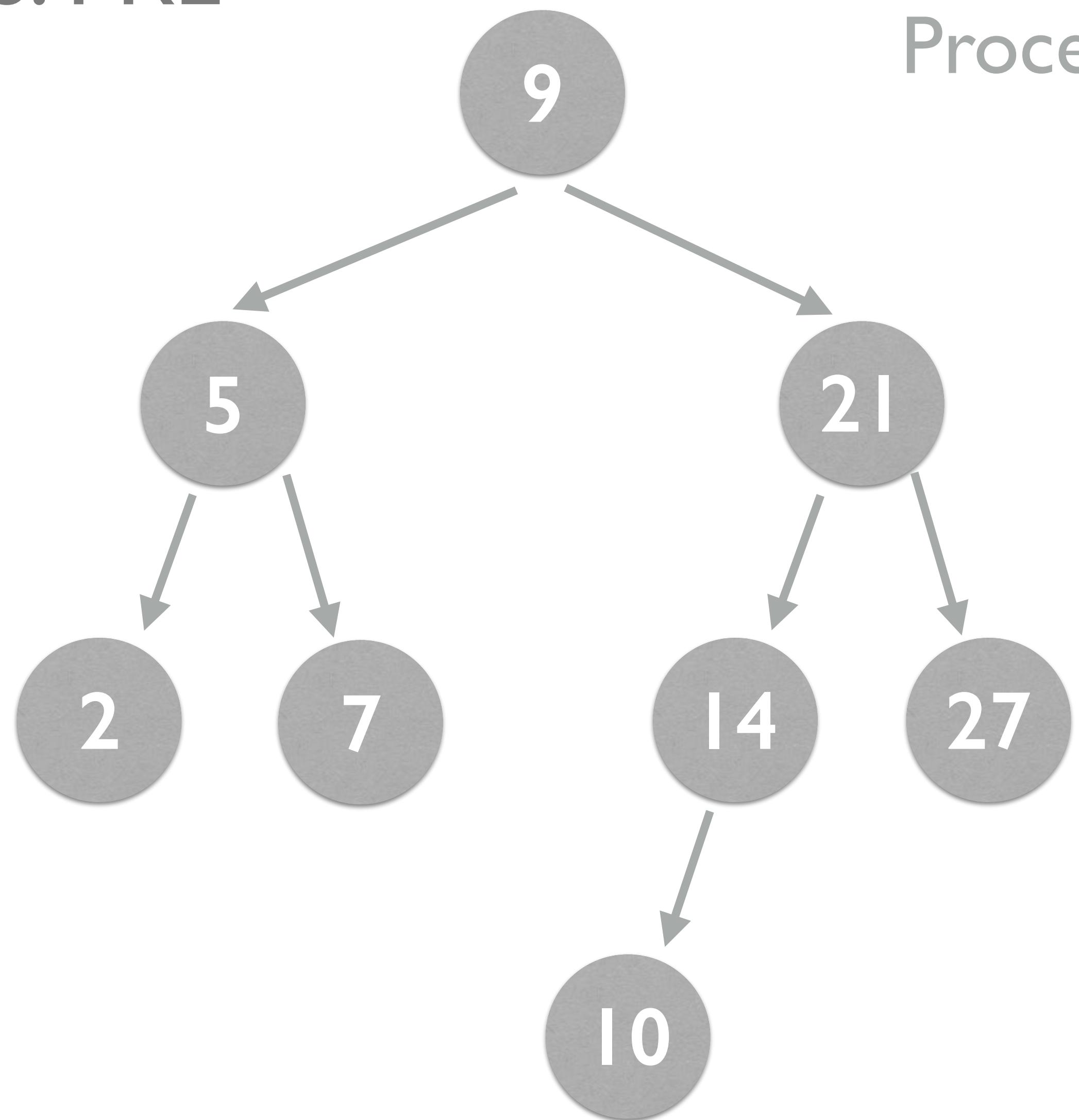
DFS: PRE



Process root · Process left · Process right

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

DFS: PRE



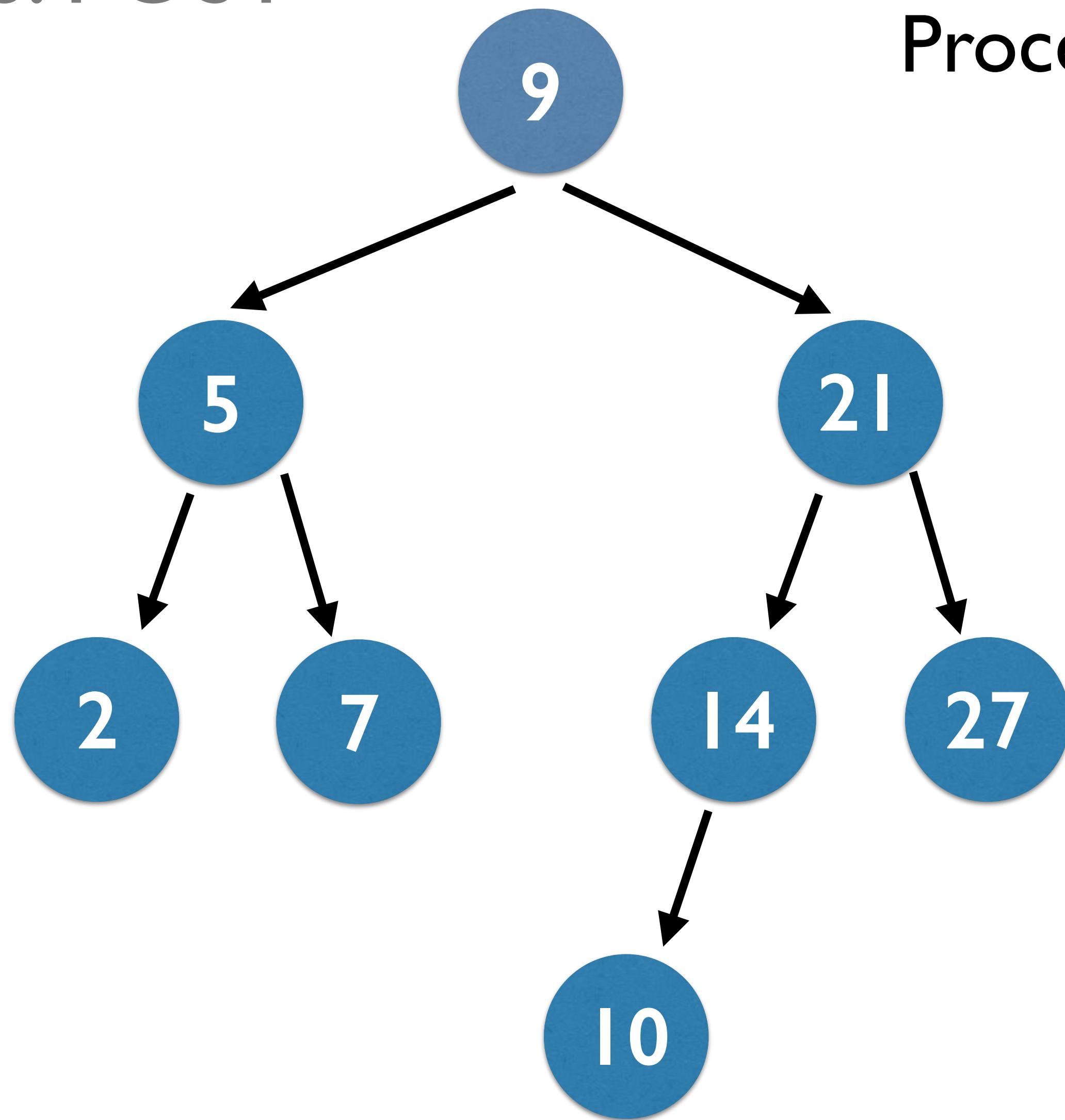
Process root · Process left · Process right

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.

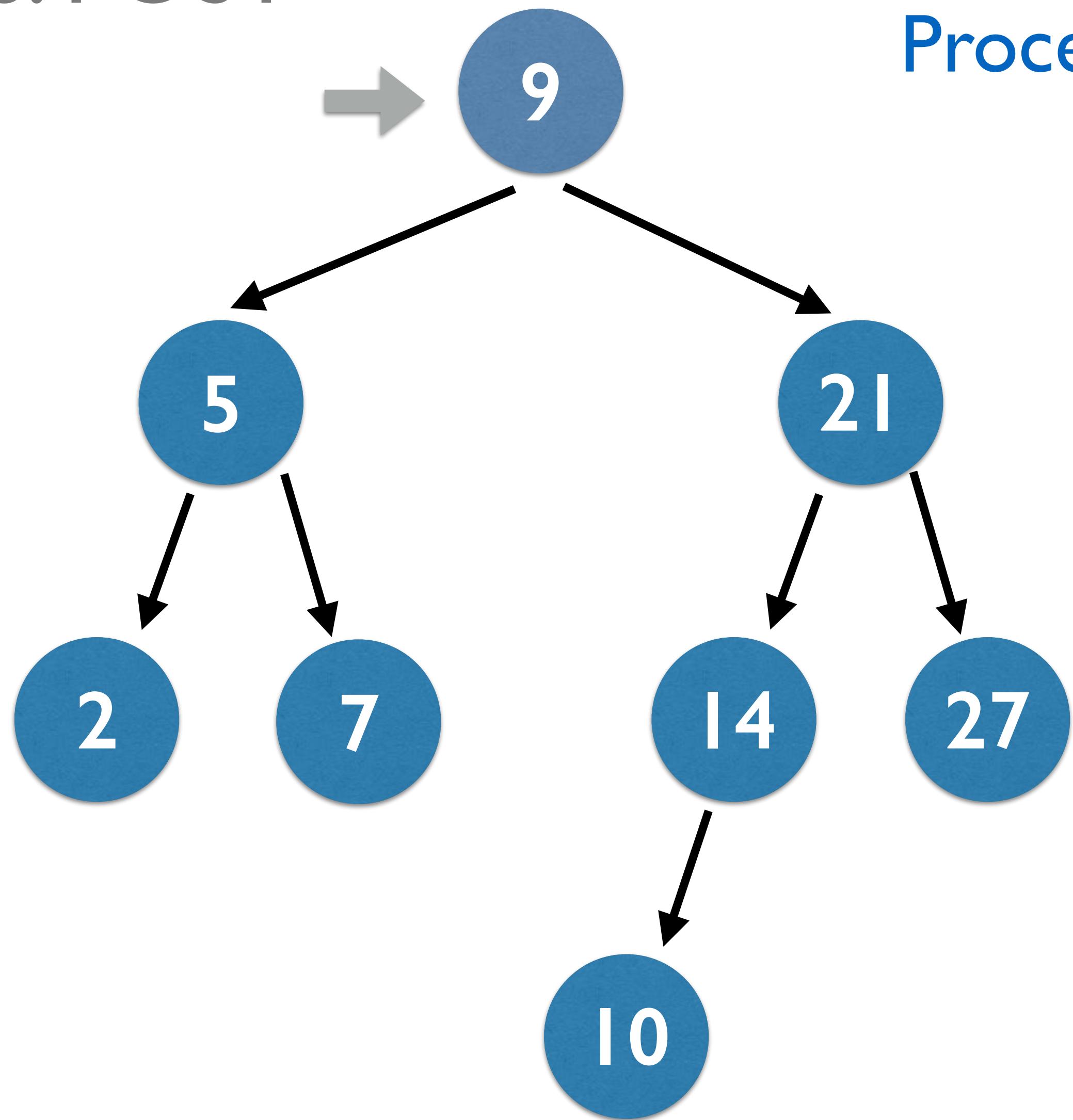
Depth First: Post-Order

DFS: POST



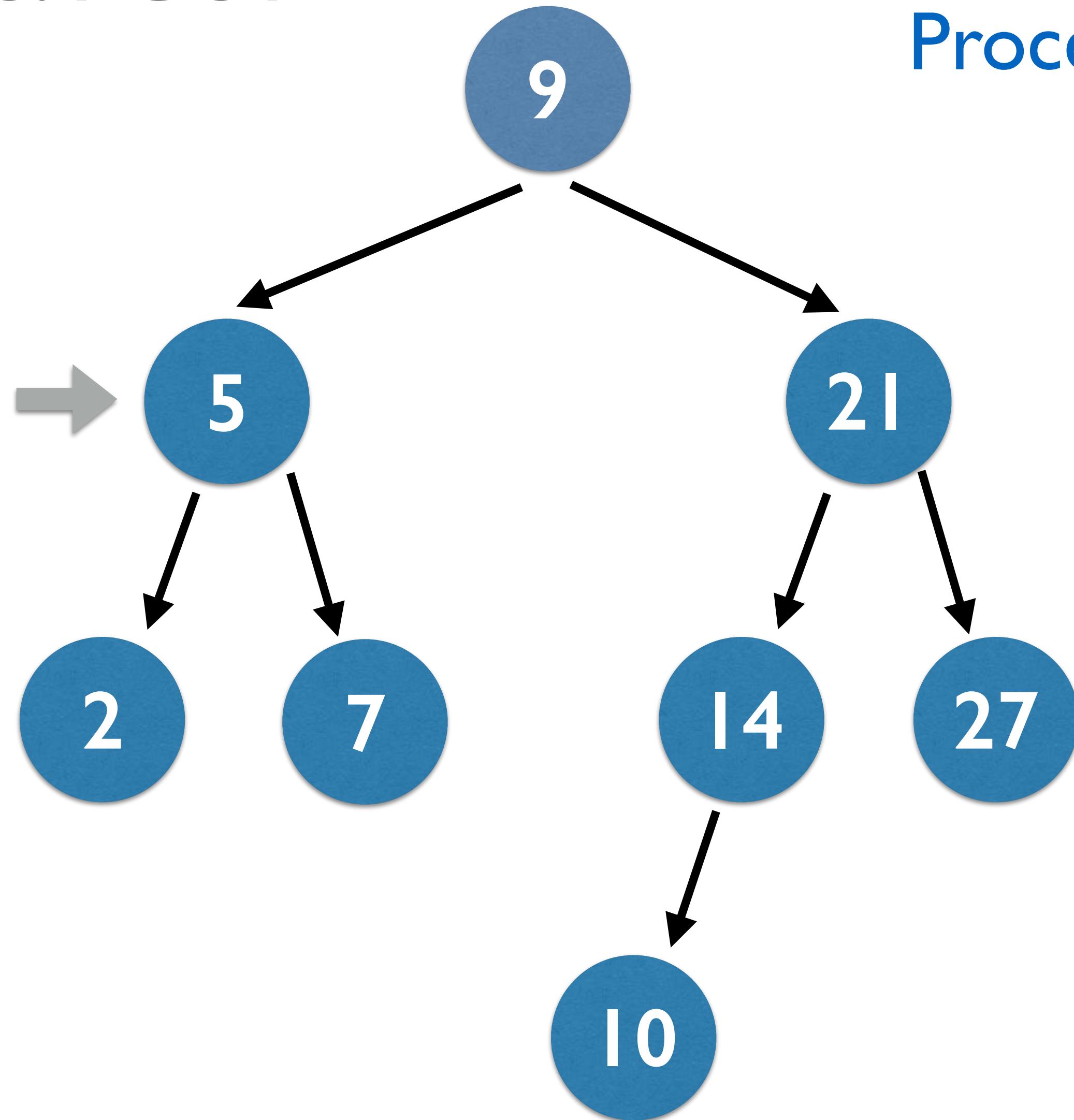
Process left · Process right · Process root

DFS: POST



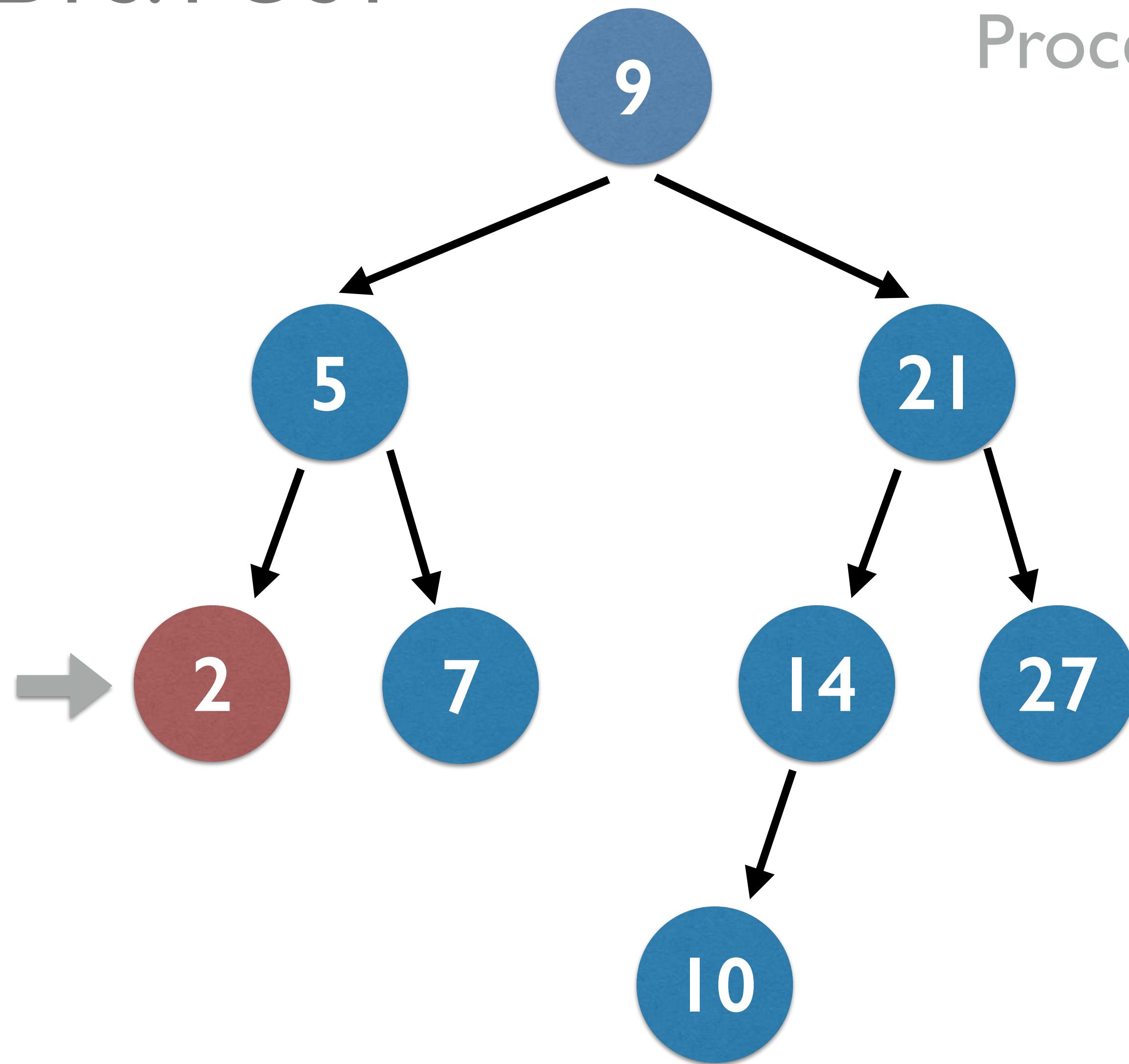
Process left · Process right · Process root

DFS: POST



Process left · Process right · Process root

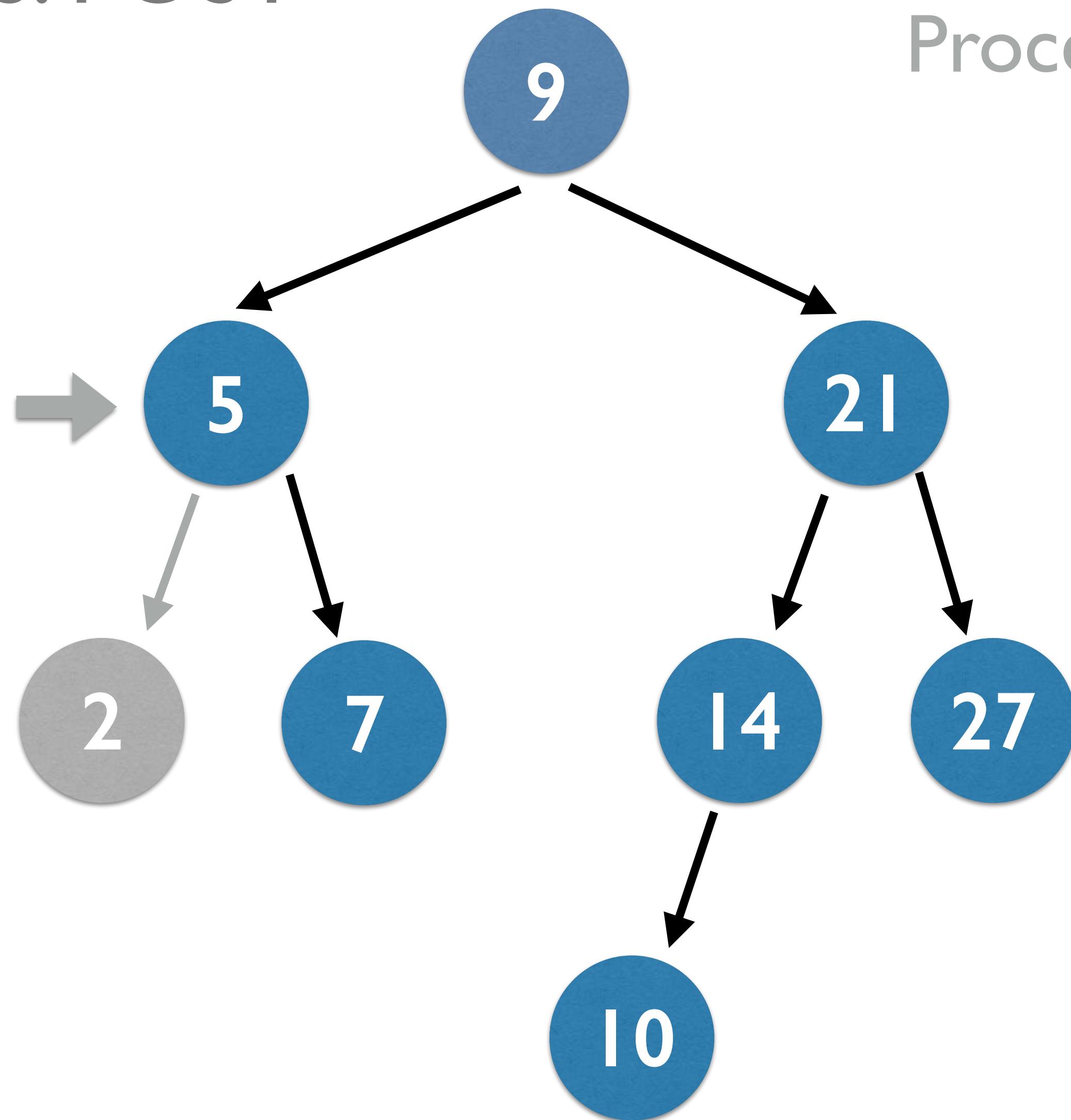
DFS: POST



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

2

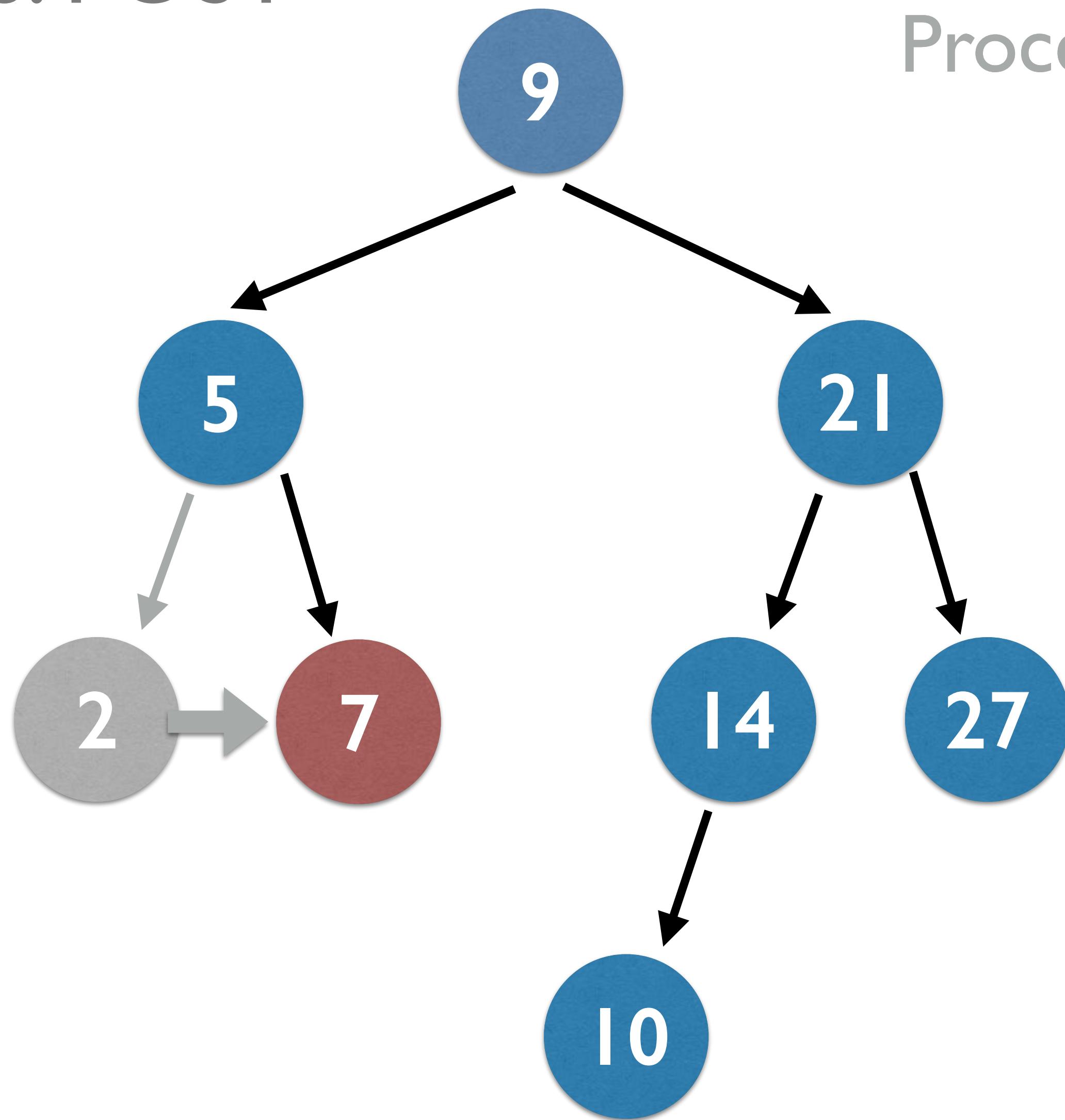
DFS: POST



Process left · Process right · Process root

2

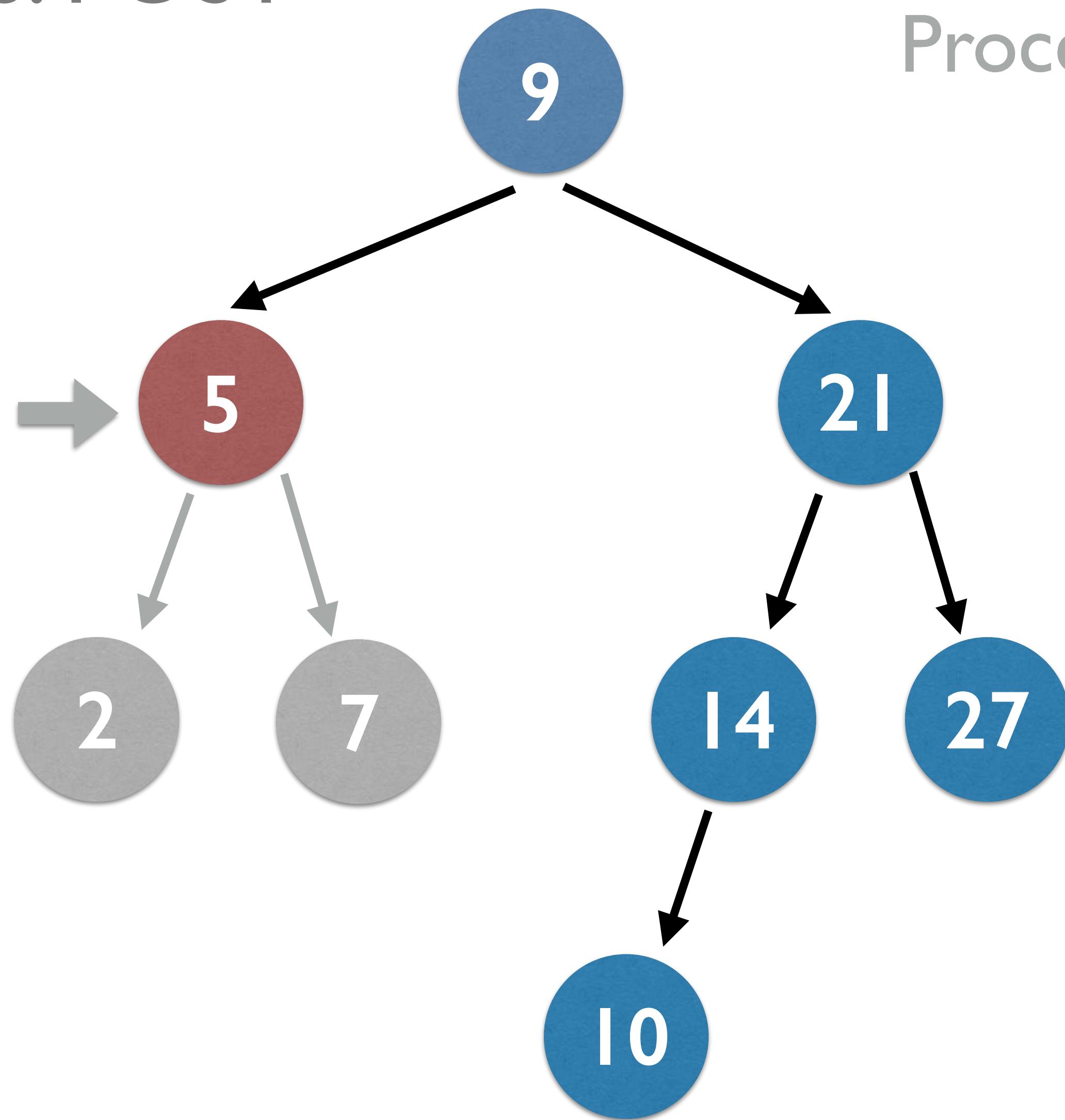
DFS: POST



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

2, 7

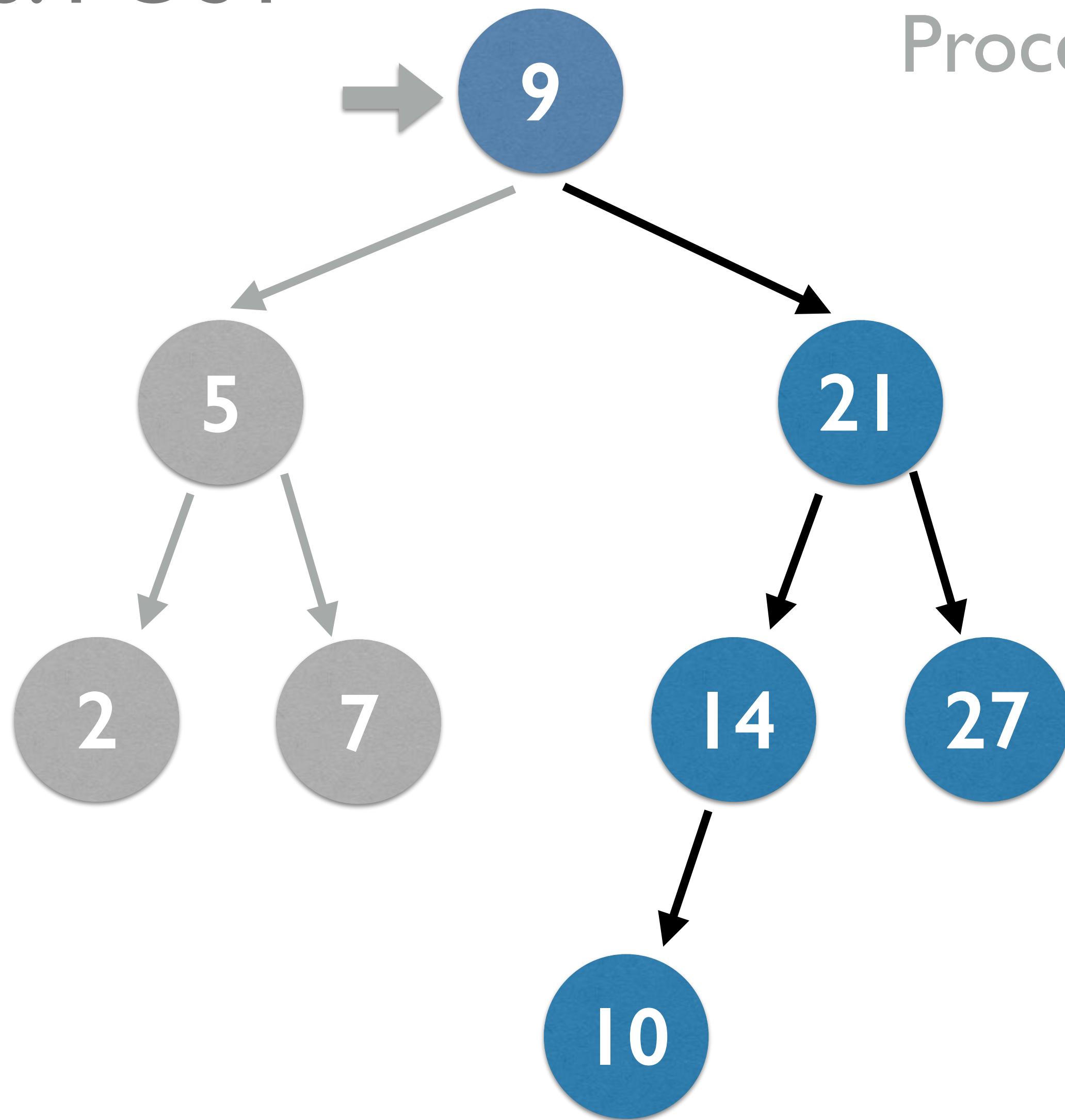
DFS: POST



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

2, 7, 5

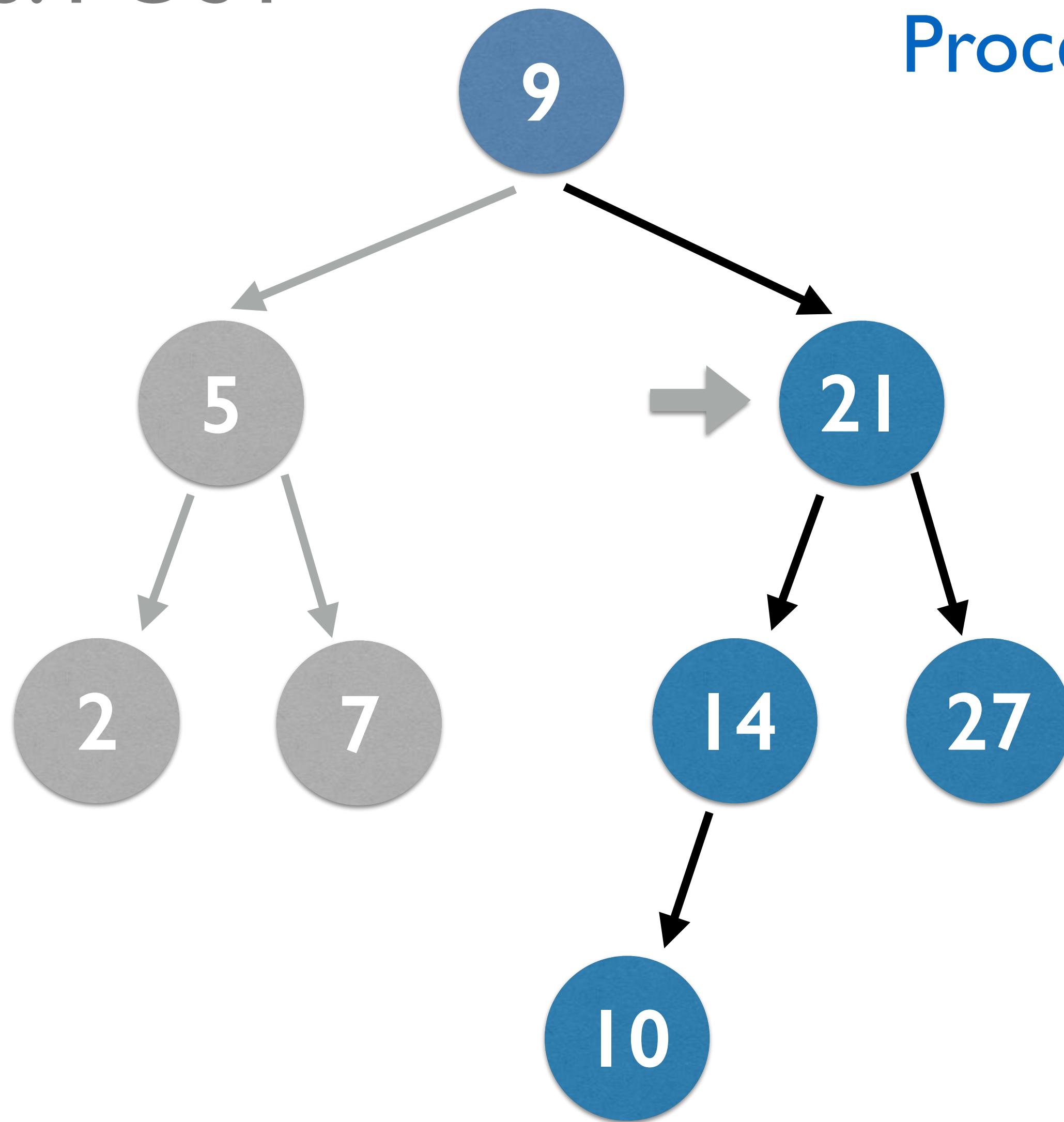
DFS: POST



Process left · Process right · Process root

2, 7, 5

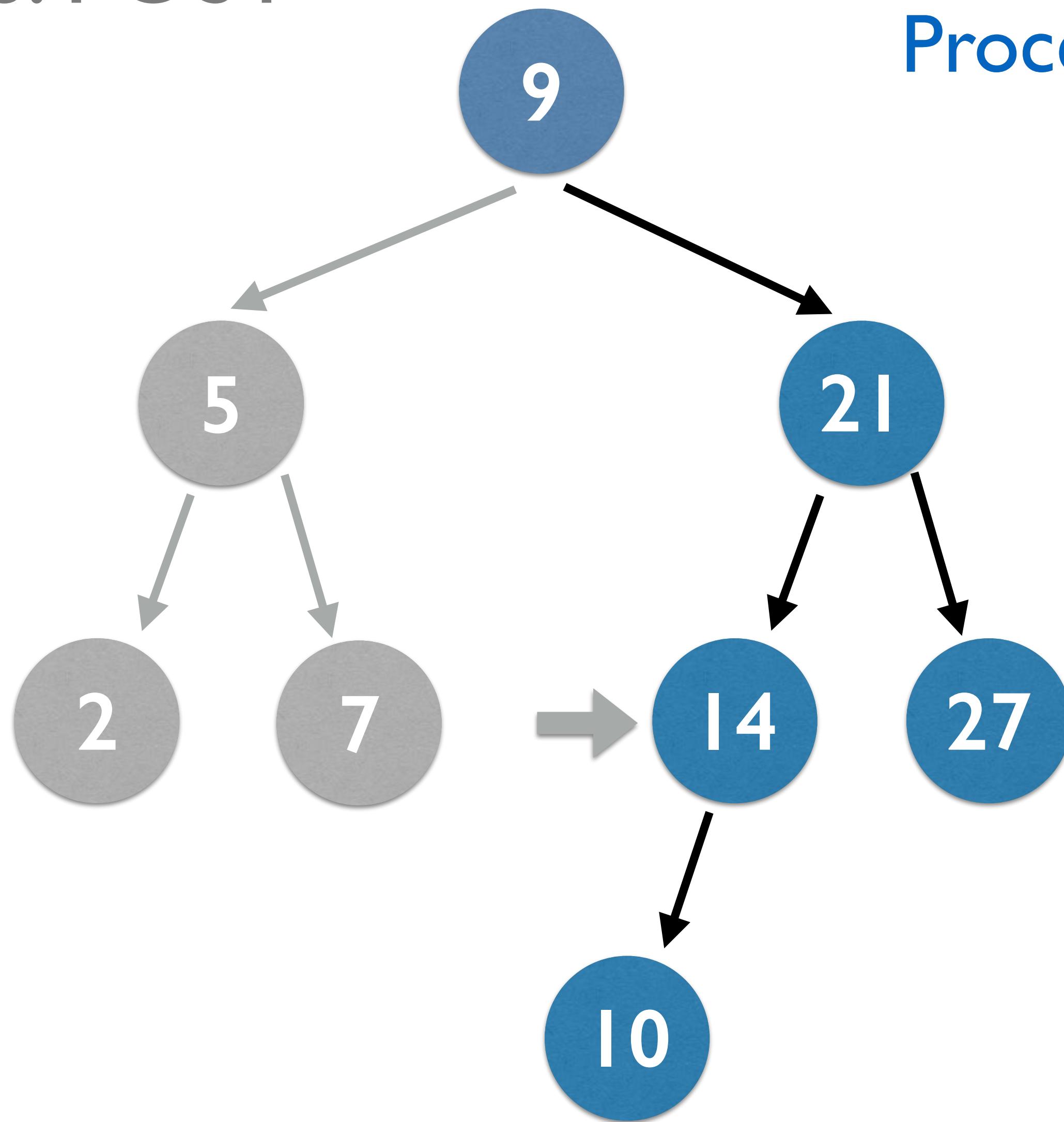
DFS: POST



Process left · Process right · Process root

2, 7, 5

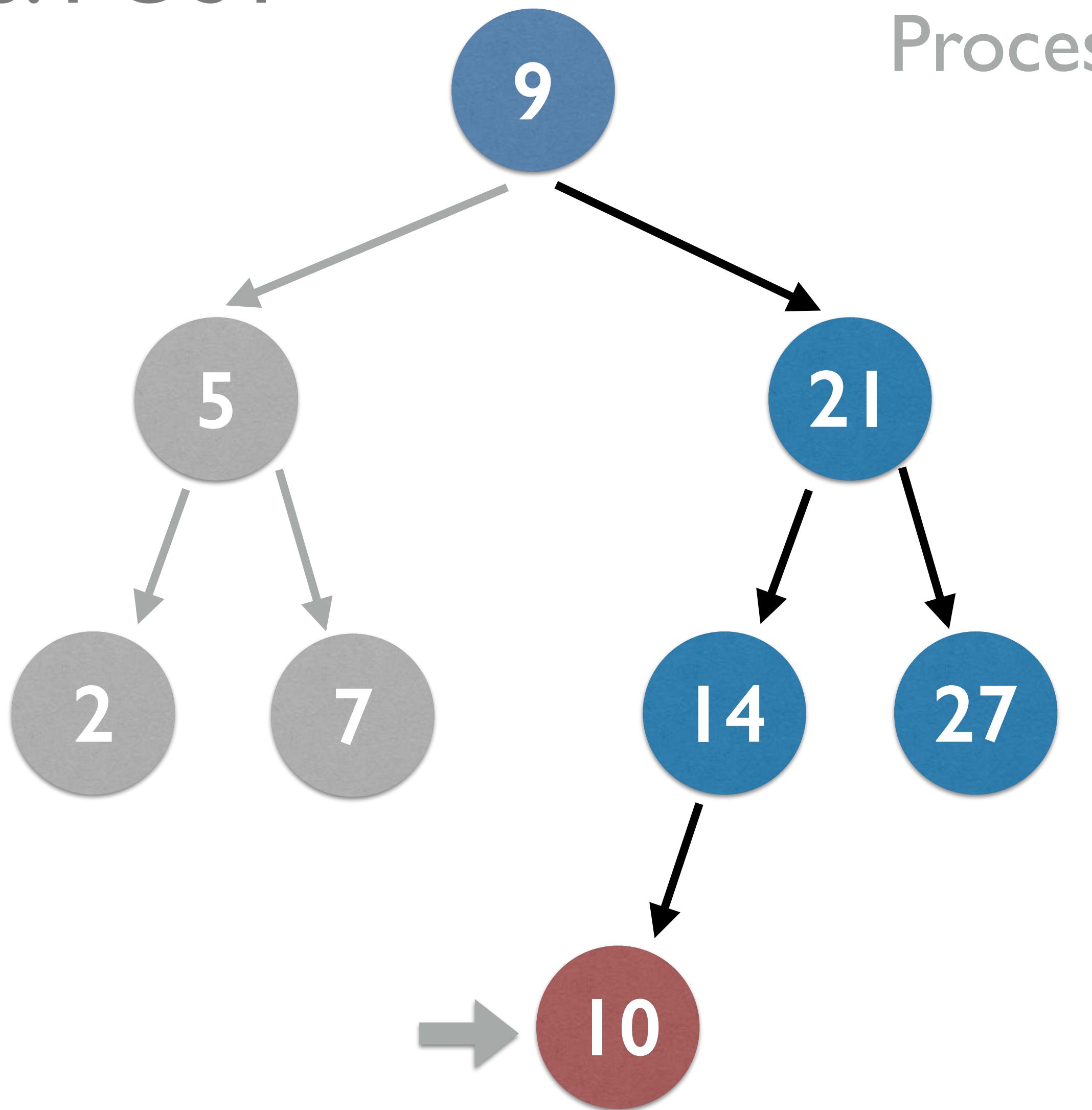
DFS: POST



Process left · Process right · Process root

2, 7, 5

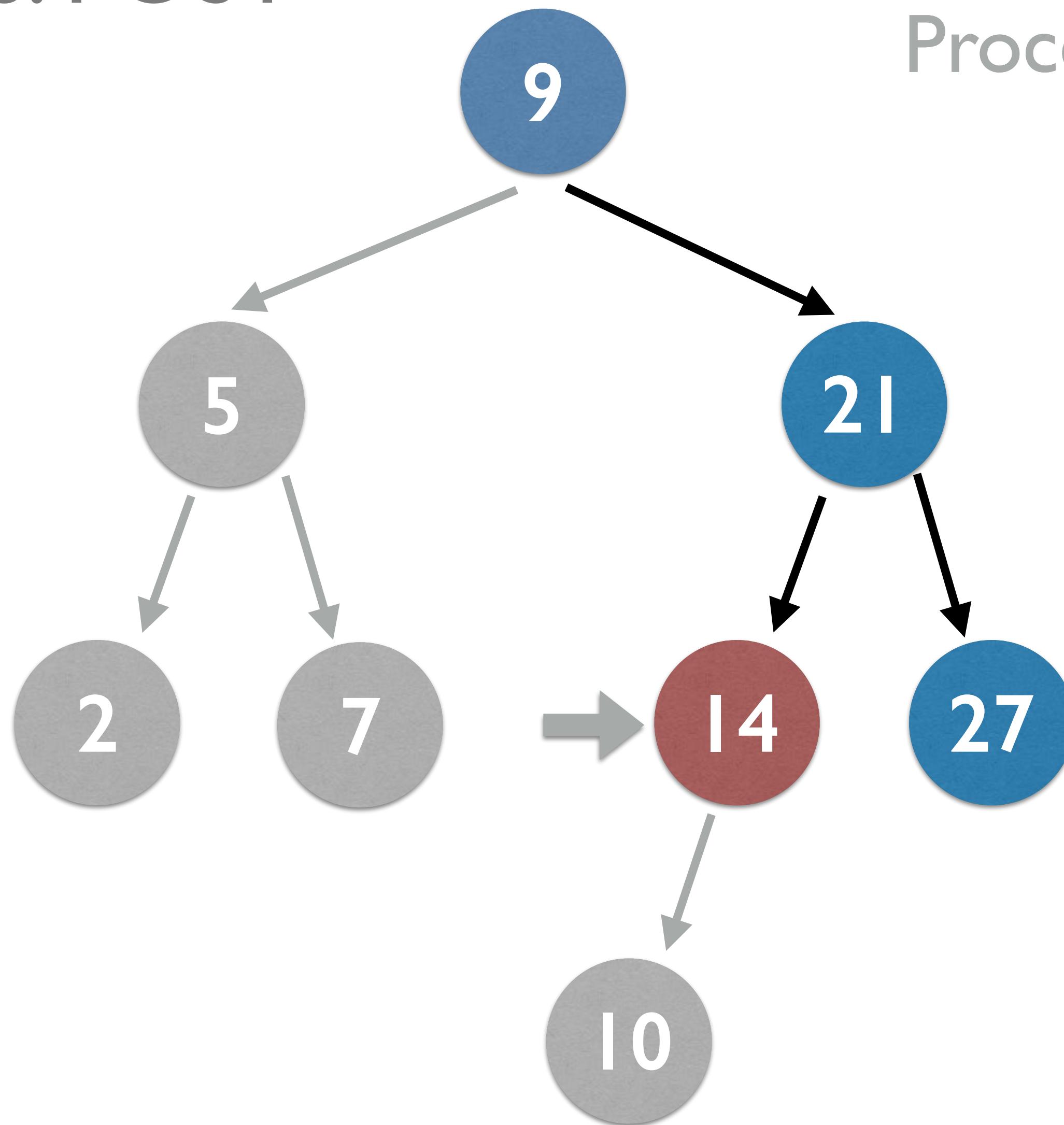
DFS: POST



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

2, 7, 5, 10

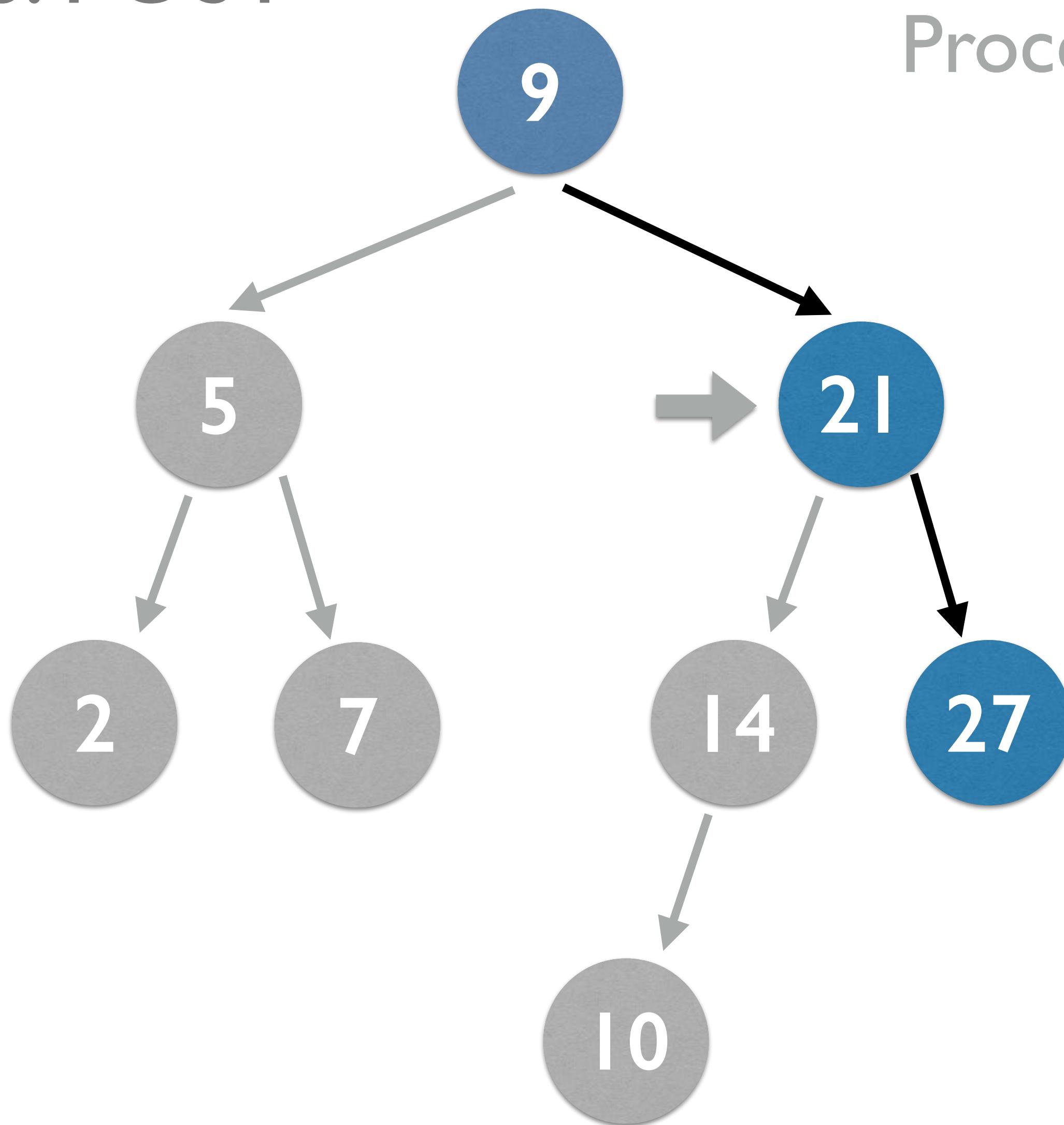
DFS: POST



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

2, 7, 5, 10, 14

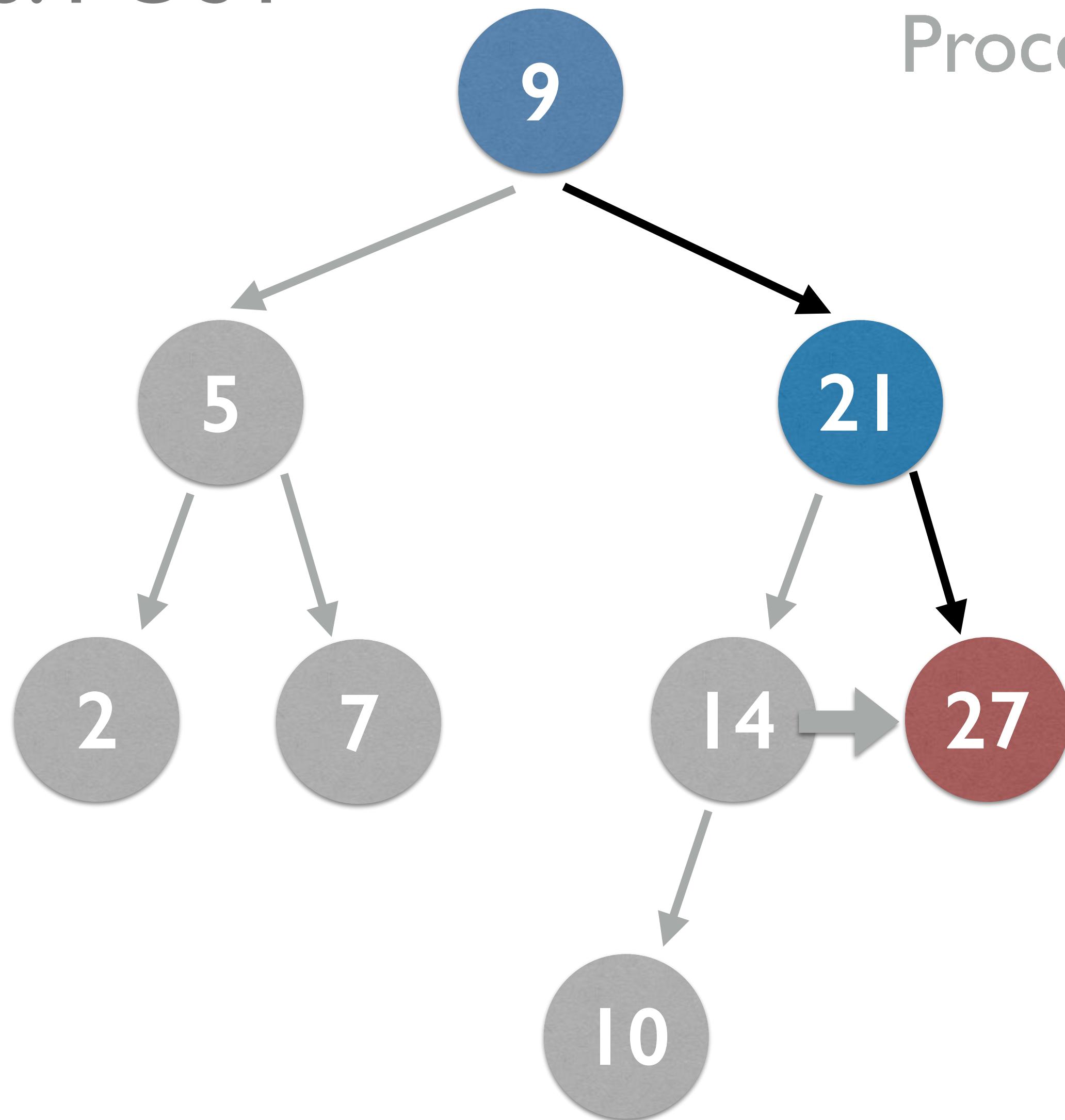
DFS: POST



Process left · Process right · Process root

2, 7, 5, 10, 14

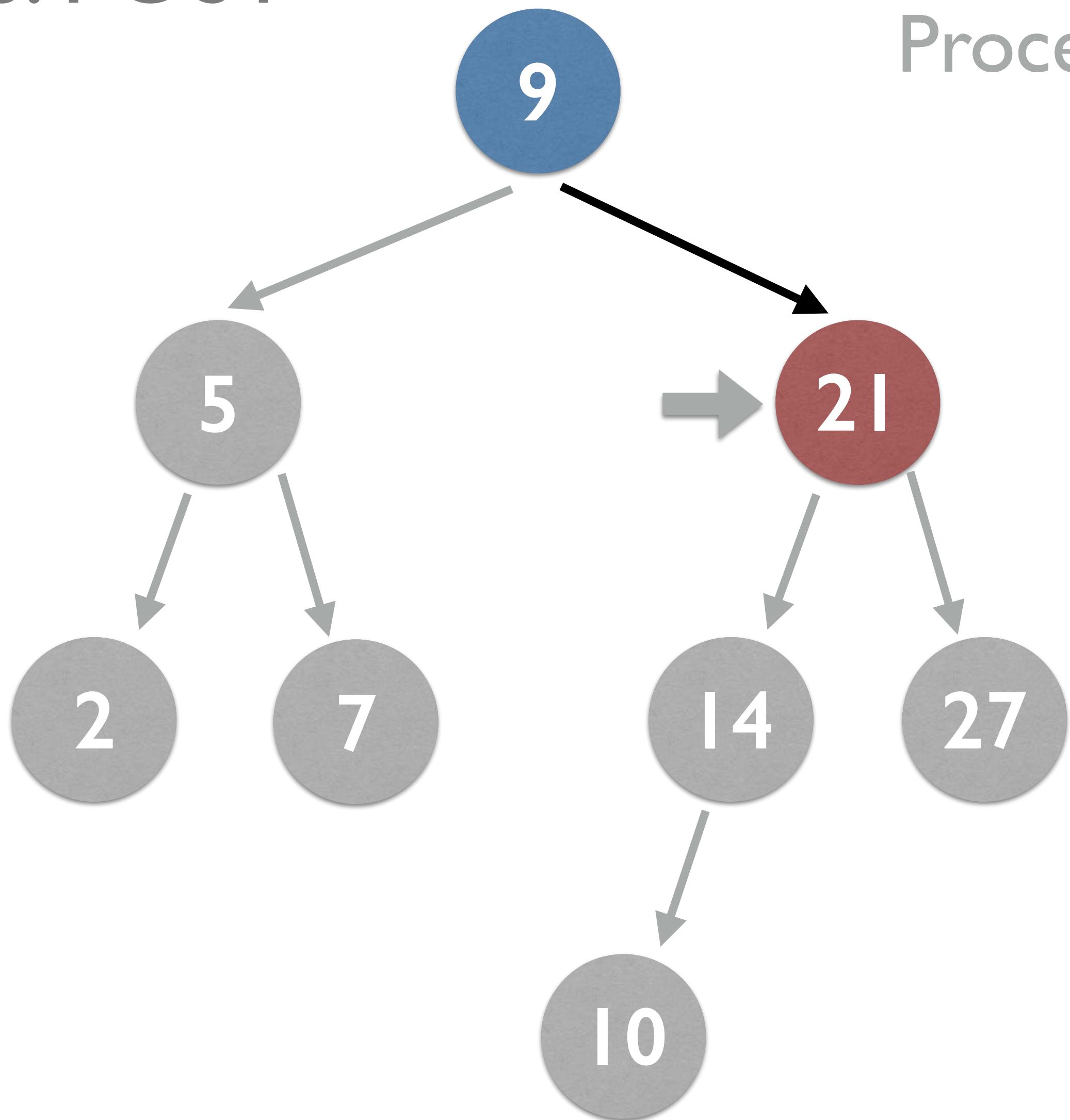
DFS: POST



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

2, 7, 5, 10, 14, 27

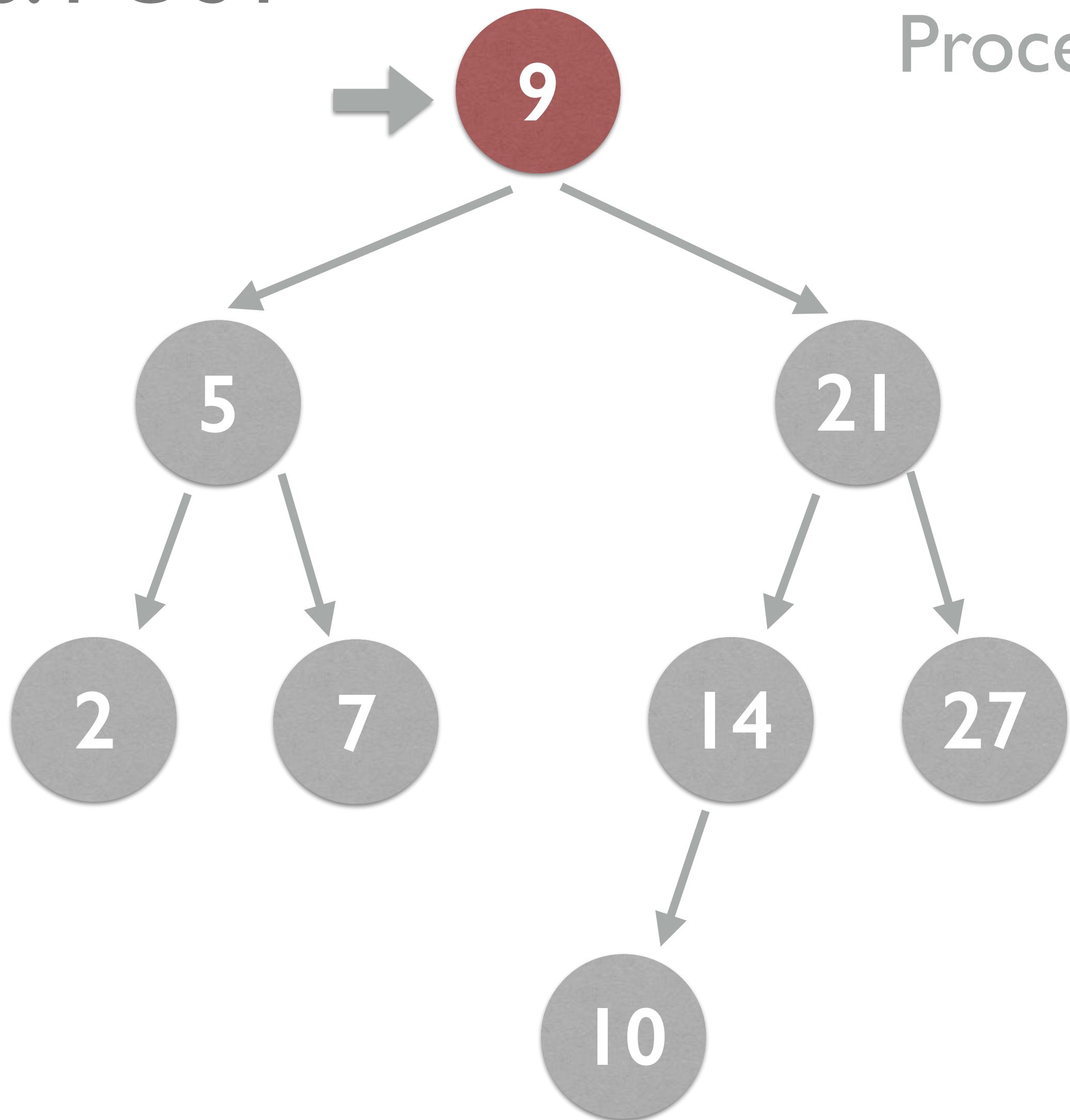
DFS: POST



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

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

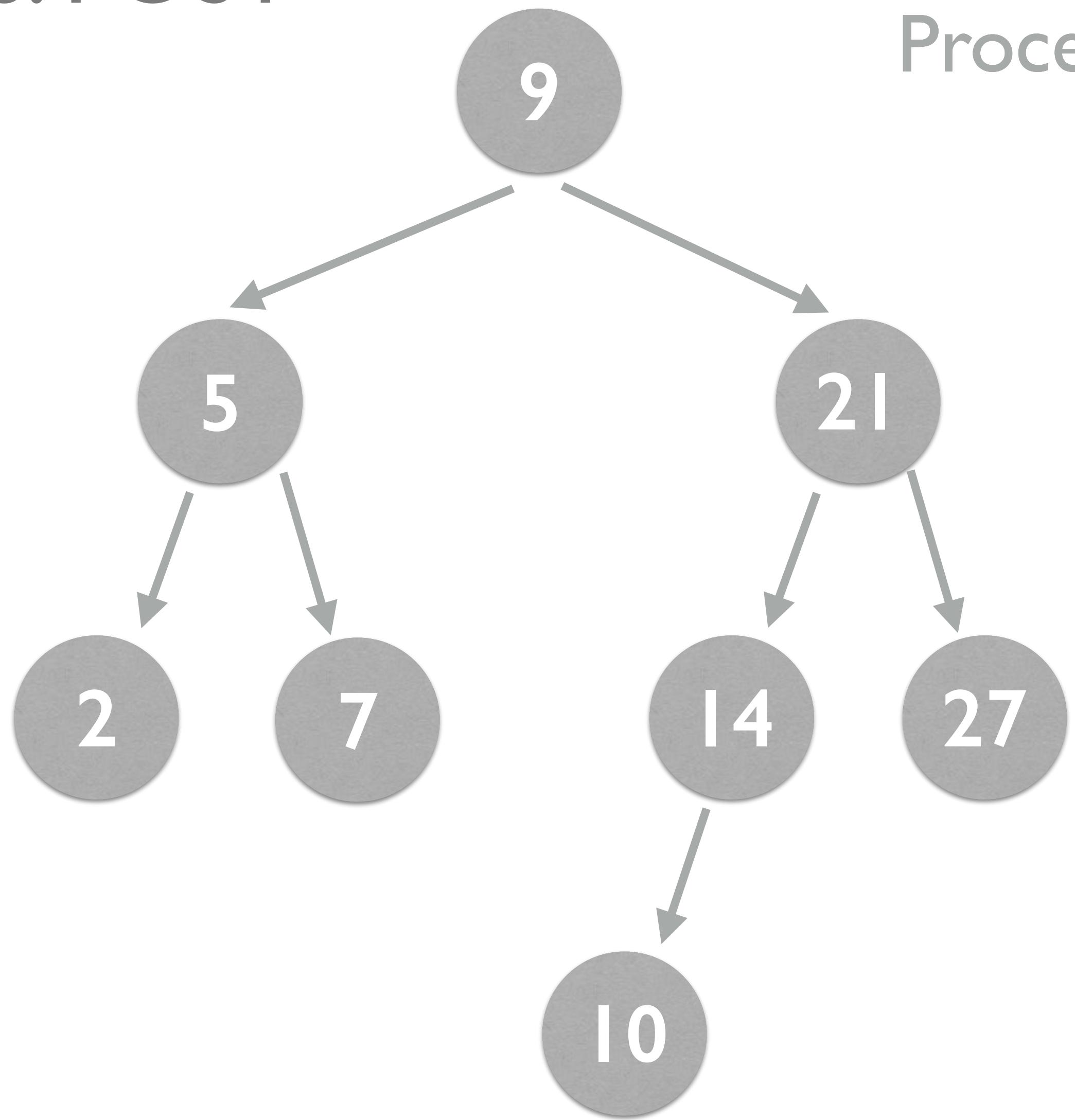
DFS: POST



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.



the end...? (no)