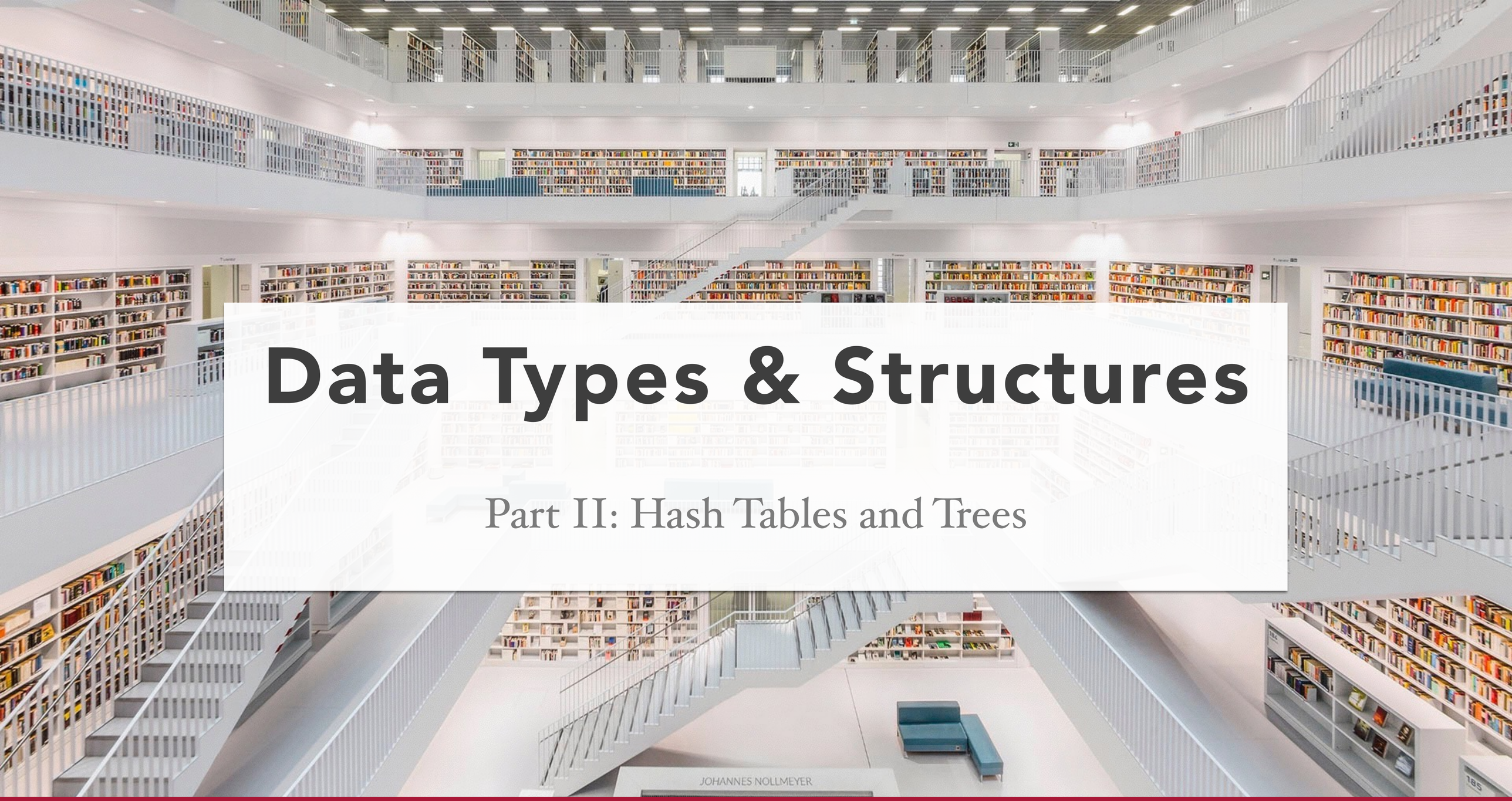
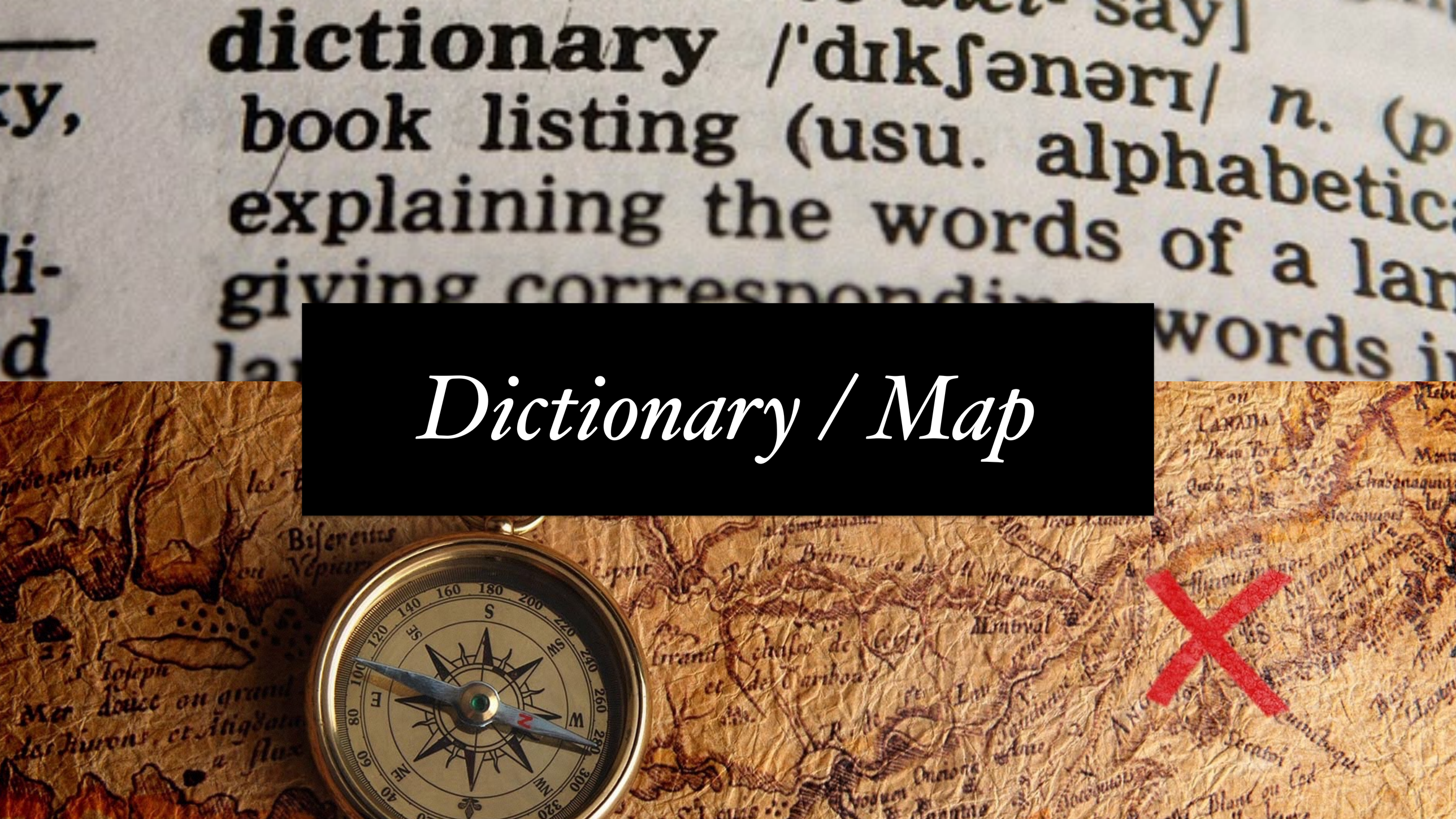


Data Types & Structures

Part II: Hash Tables and Trees


dictionary /'dɪkʃənərɪ/ n.
book listing (usu. alphabetic
explaining the words of a lan
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?

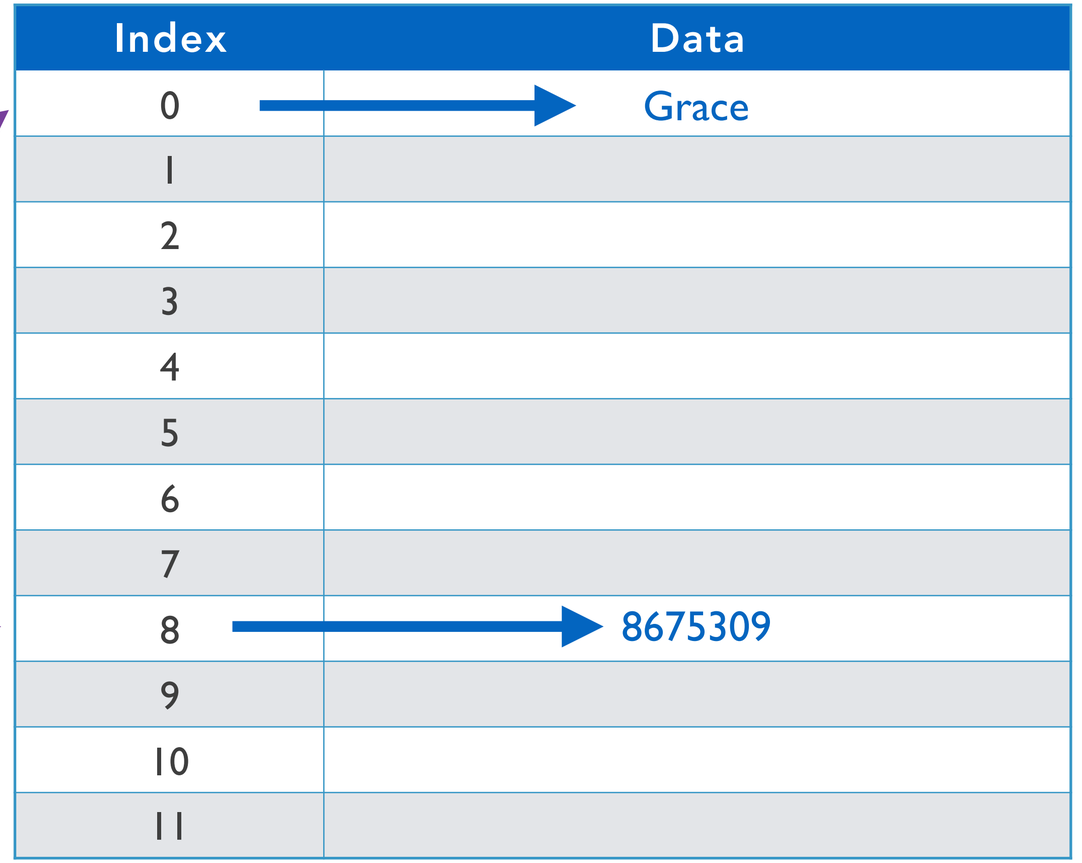
The classic data structure: a Hash Table

- 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 % 12; // number of spaces in array
}
```

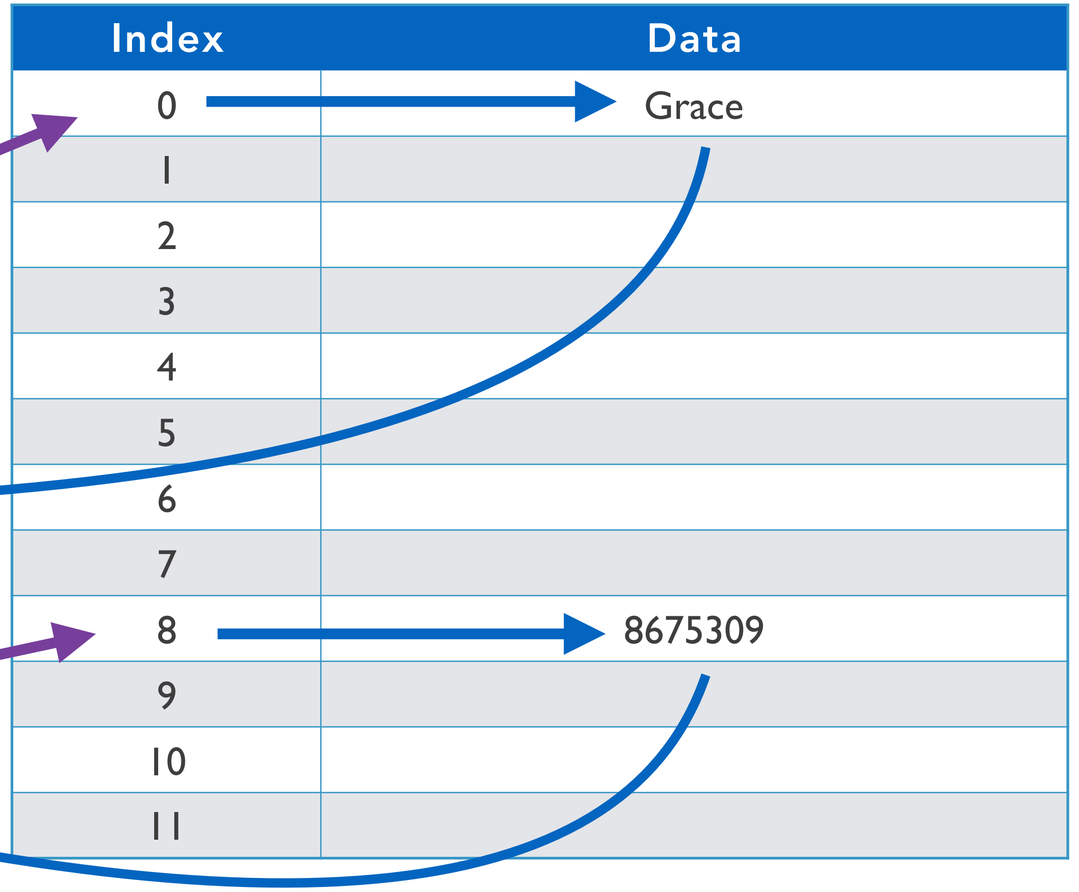
Example with a 12-bucket array

1. We want to store the value Grace for the key name.
2. Hashing name yields the numerical index 0.
3. Store Grace at index 0.
4. Now store the value 8675309 for key phone
5. phone hashes to 8
6. Store 8675309 at index 8



Fetching/changing values

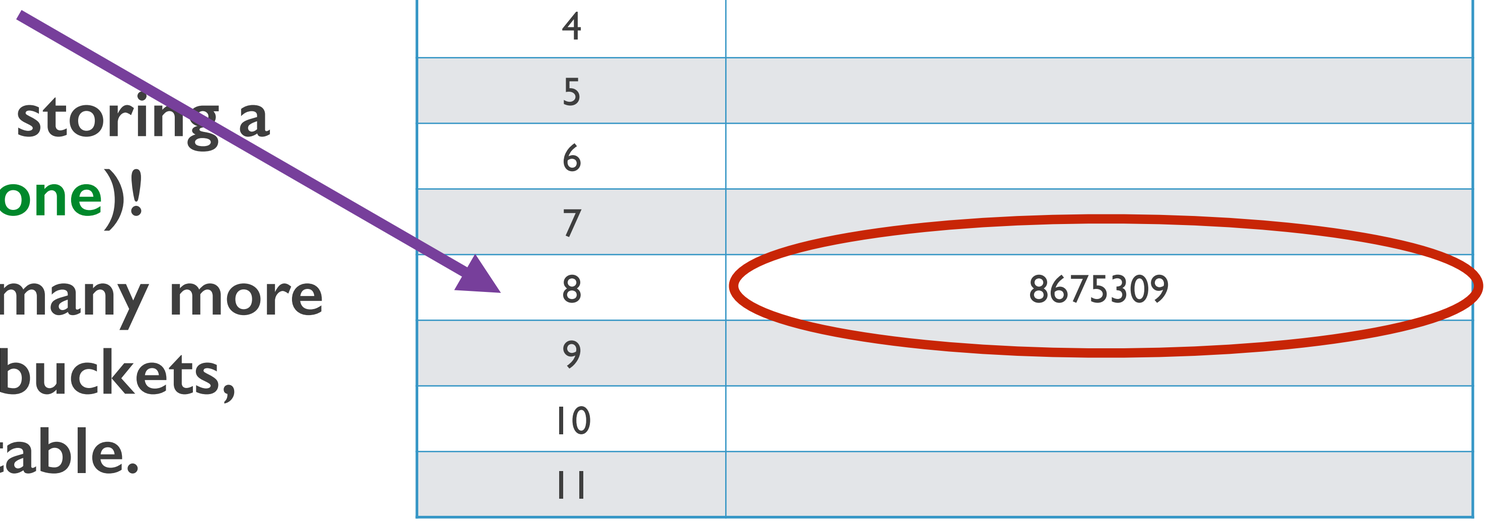
1. What is the value for the key name?
2. Hashing name yields the numerical index 0.
3. Find the value at index 0.
4. Result: Grace
5. Similar steps for retrieving the value 8675309 for the key phone.



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 8.
3. But we are already storing a value there (for phone)!
4. Because there are many more possible keys than buckets, collisions are inevitable.



Index	Data
0	Grace
1	
2	
3	
4	
5	
6	
7	
8	8675309
9	
10	
11	

How to resolve collisions?

Two main strategies

- **Open addressing**: if a bucket is full, look for the next empty bucket. Then you need to store keys and values together so you know if you have reached the correct bucket.

- **Separate chaining**: every bucket stores a secondary data structure, like a linked list. Each list node contains both key and value, so you know if you've reached the right node.

Separate chaining: adding a pair

1. We want to store the value grace@gmail.com for the key email.
2. Hashing email yields the numerical index 8.
3. Add the value to the Linked List as a new node.

Index	Linked List in bucket
0	<Grace>
1	
2	
3	
4	
5	
6	
7	
8	<8675309> → <grace@gmail.com>
9	
10	
11	

We still have a problem...

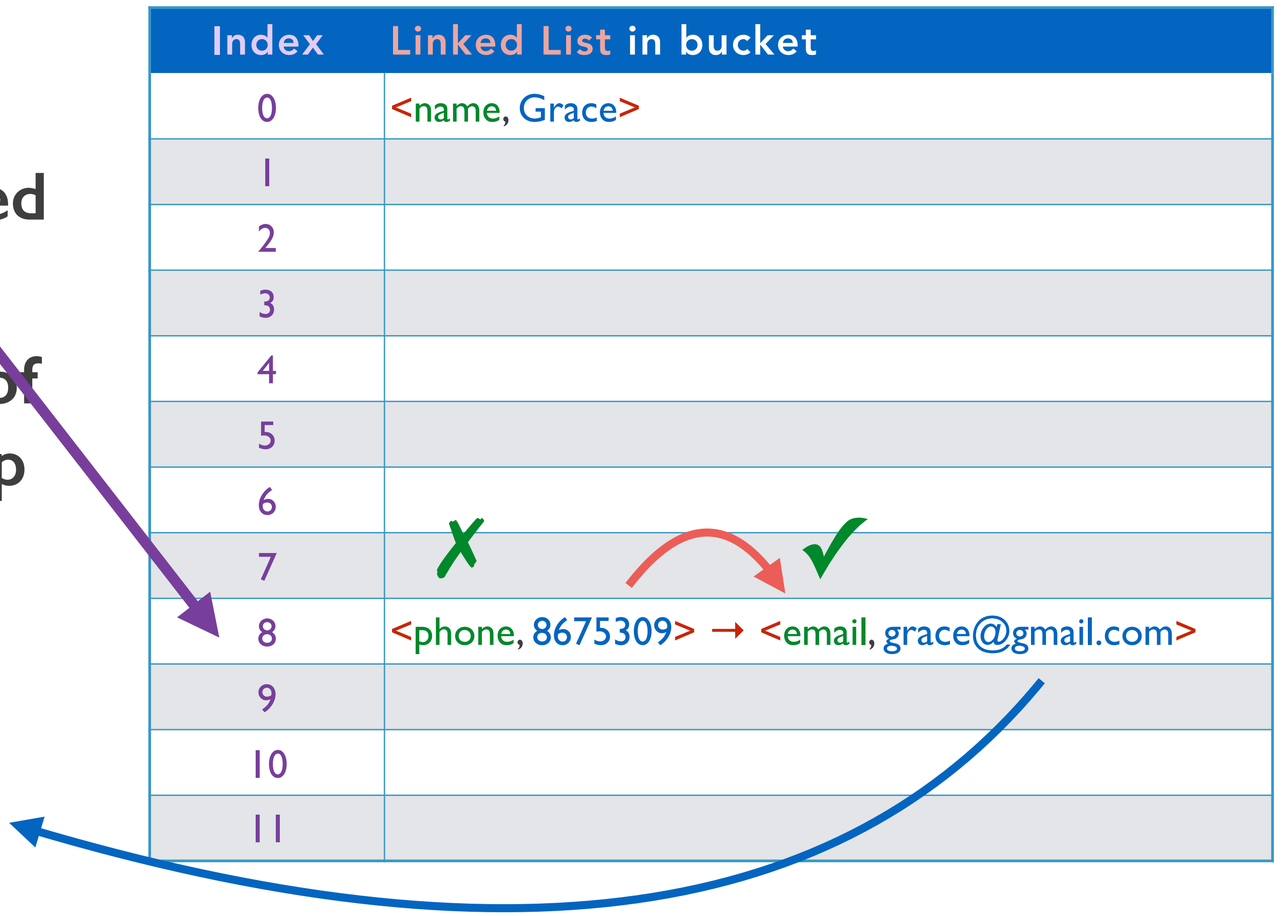
Separate chaining: retrieving a value

1. What is the value for the key email?
2. Hashing email yields the numerical index 8.
3. In bucket 8 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	<Grace>
1	
2	
3	
4	
5	
6	
7	? ?
8	<8675309> → <grace@gmail.com>
9	
10	
11	

Separate chaining: store key-val pairs

1. Hashing email yields the numerical index 8.
2. In bucket 8 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



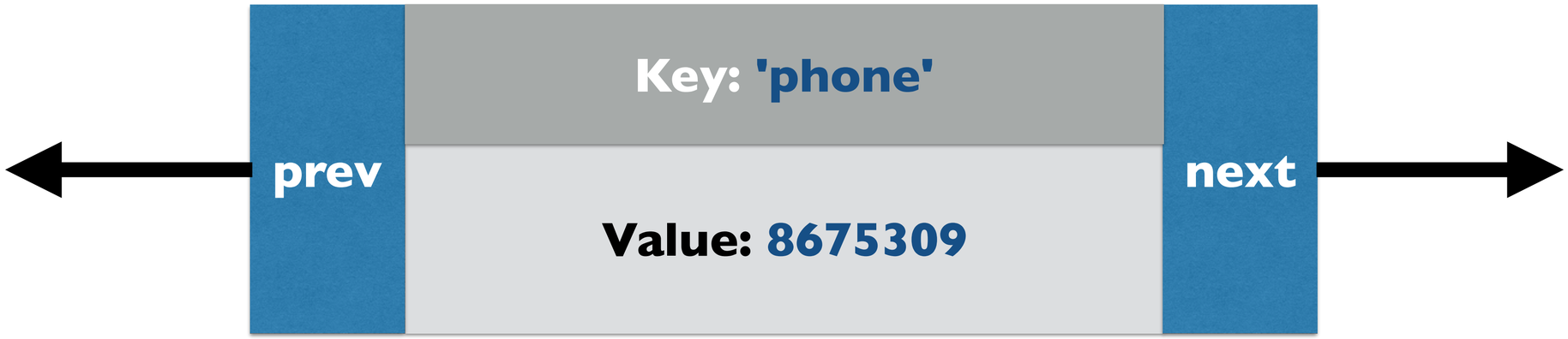
How is a hash table better than one big linked list?

Performance

- In a hash table with a healthy number of buckets and a good hashing function, *most of the time* accessing or changing a value by key occurs in as much time as it takes to hash the key (fast)
- *Sometimes* collisions occur and you'll have to traverse a few nodes of a linked list; but when that happens, usually it will just be 2–3 nodes total (still pretty fast)
- That's better than having to traverse, say, several dozen nodes every time you want to access/change a specific key!

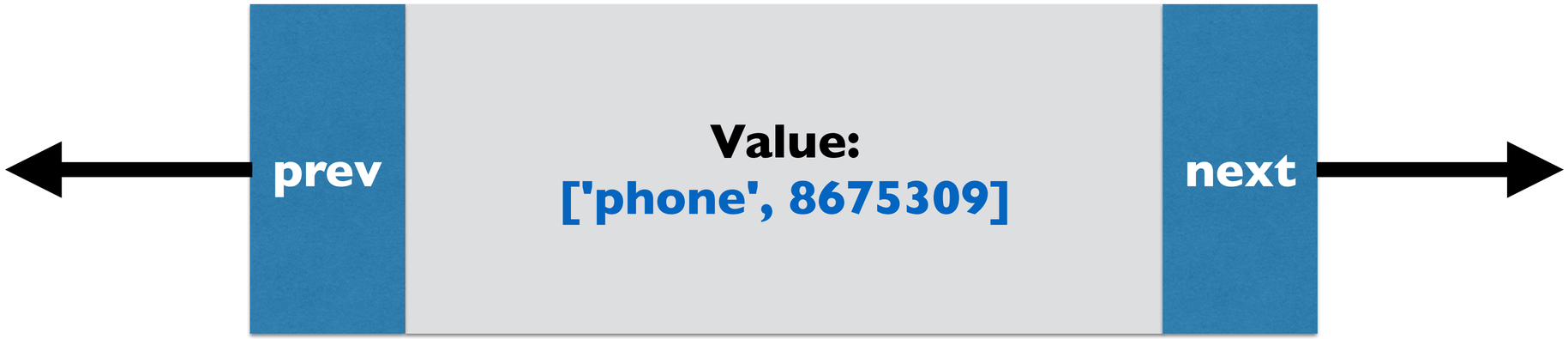
List nodes with key-value pairs — how?

Solution 1: *Association List*



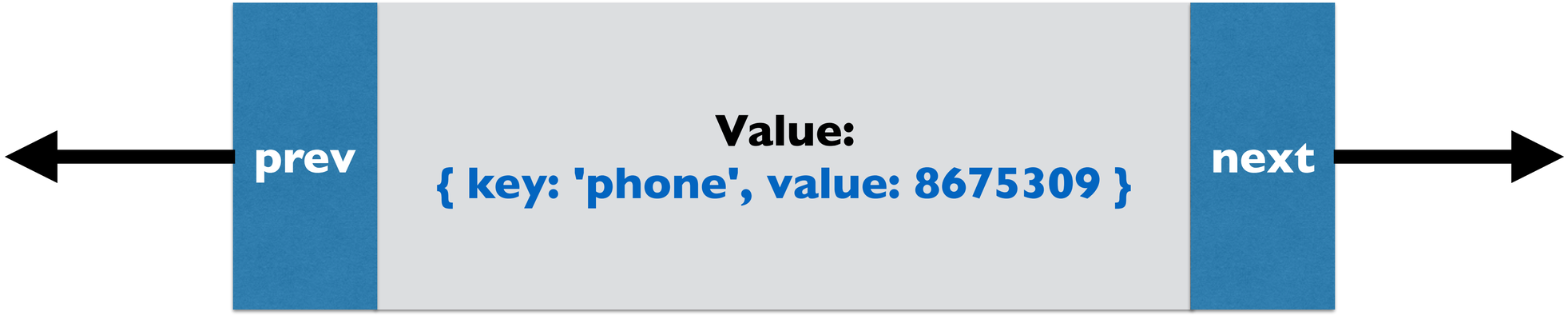
- **Nodes have key as well as value & next / prev**
  - Advantage: best for purpose, most straightforward.
  - Downside: need a new custom LL implementation (extra work)

Solution 2: store an array



- **LL node value is an array, index 0 stores key & 1 stores 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.

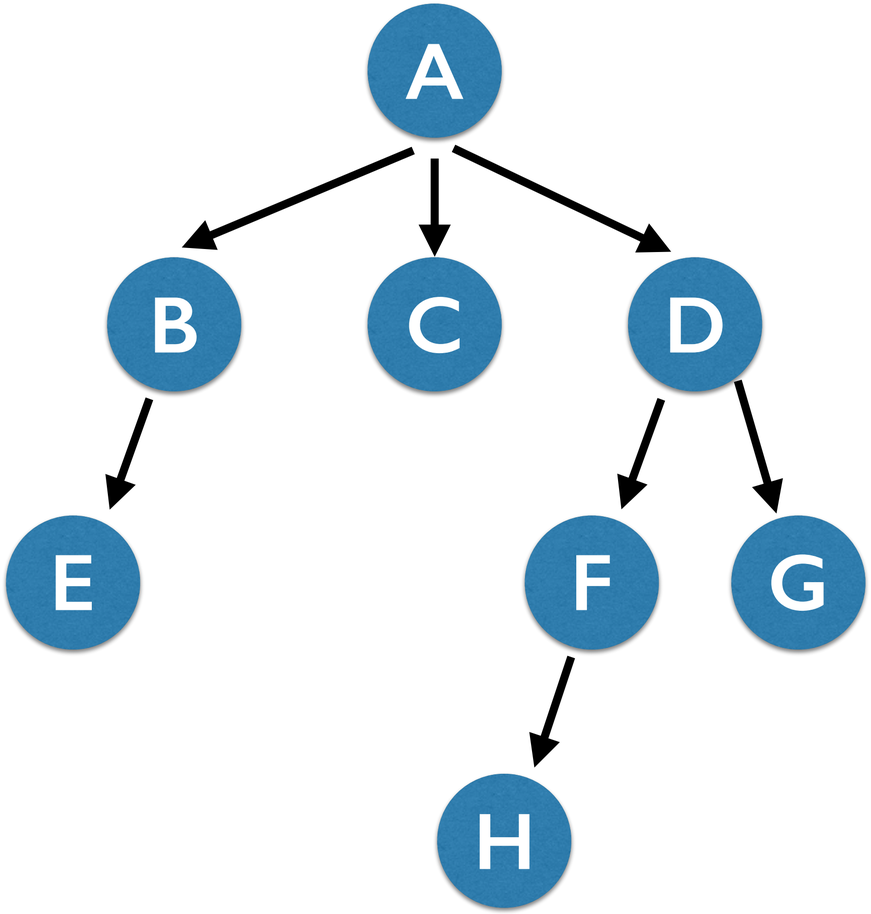
WHAT ABOUT JS?

Sound familiar?

- **JavaScript Objects** are containers for key-value pairs, to which you can add new pairs, modify existing values, and delete pairs.
- As you might expect, JS *engines* (like V8) often implement Objects using hash tables.
- However, sometimes the engine uses arrays plus additional overflow arrays, or similar structures.
- 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.

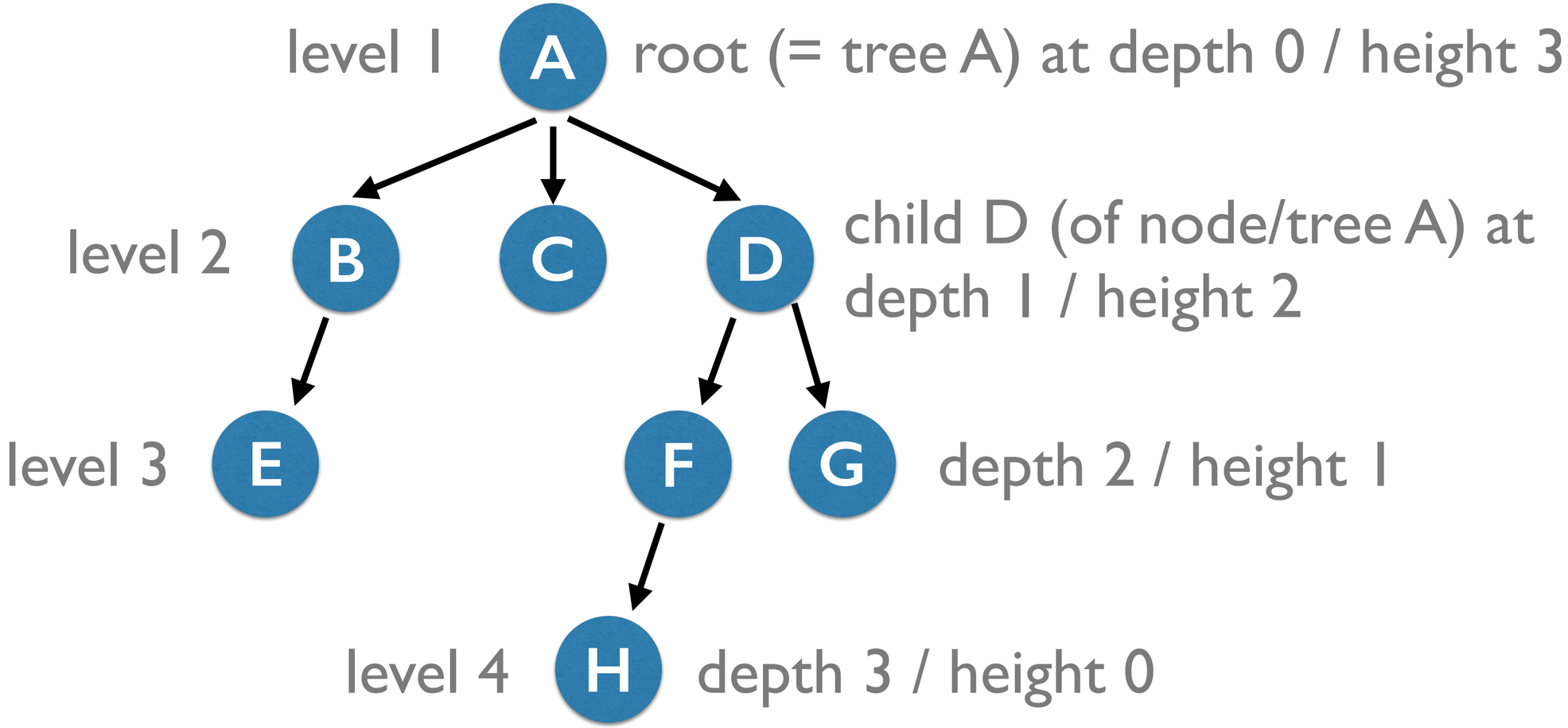

Trees

The Tree ADT

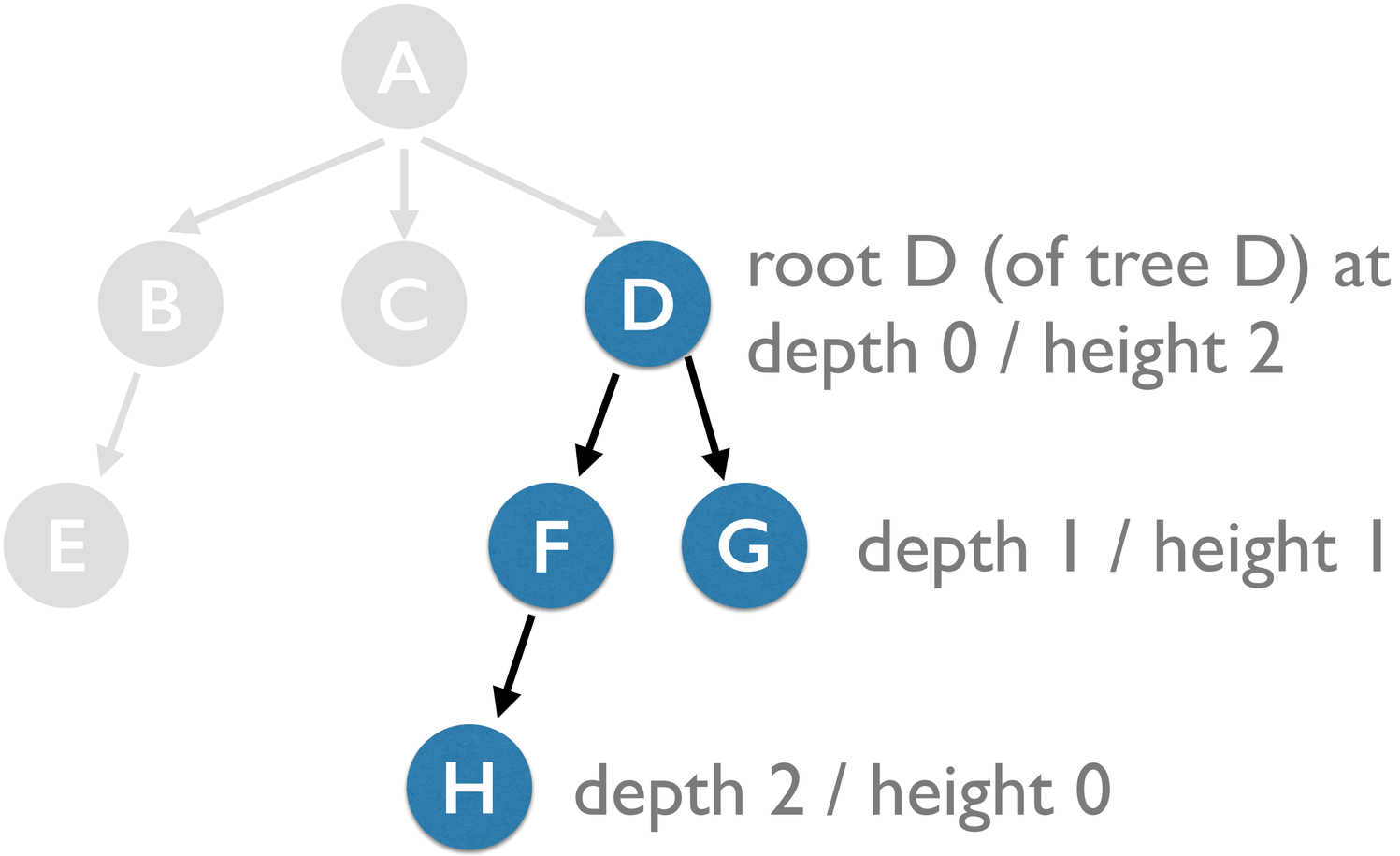


- Nodes contain value(s)
- A primary "root" node
- Children are subtrees (recursive!)
- No duplicated children (cycles); trees can *branch* but never *converge*
- Final nodes called "leaves"
- Height of tree = longest path to leaf
- Level = 1 + number of cnxns to root
- Depth = inverse of height

The Tree ADT

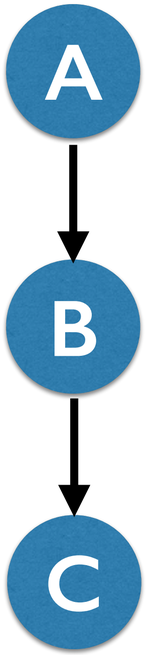

level 1
A
root (= tree A) at depth 0 / height 3
level 2
B
C
D
child D (of node/tree A) at depth 1 / height 2
level 3
E
F
G
depth 2 / height 1
level 4
H
depth 3 / height 0

Every node is the root of a tree



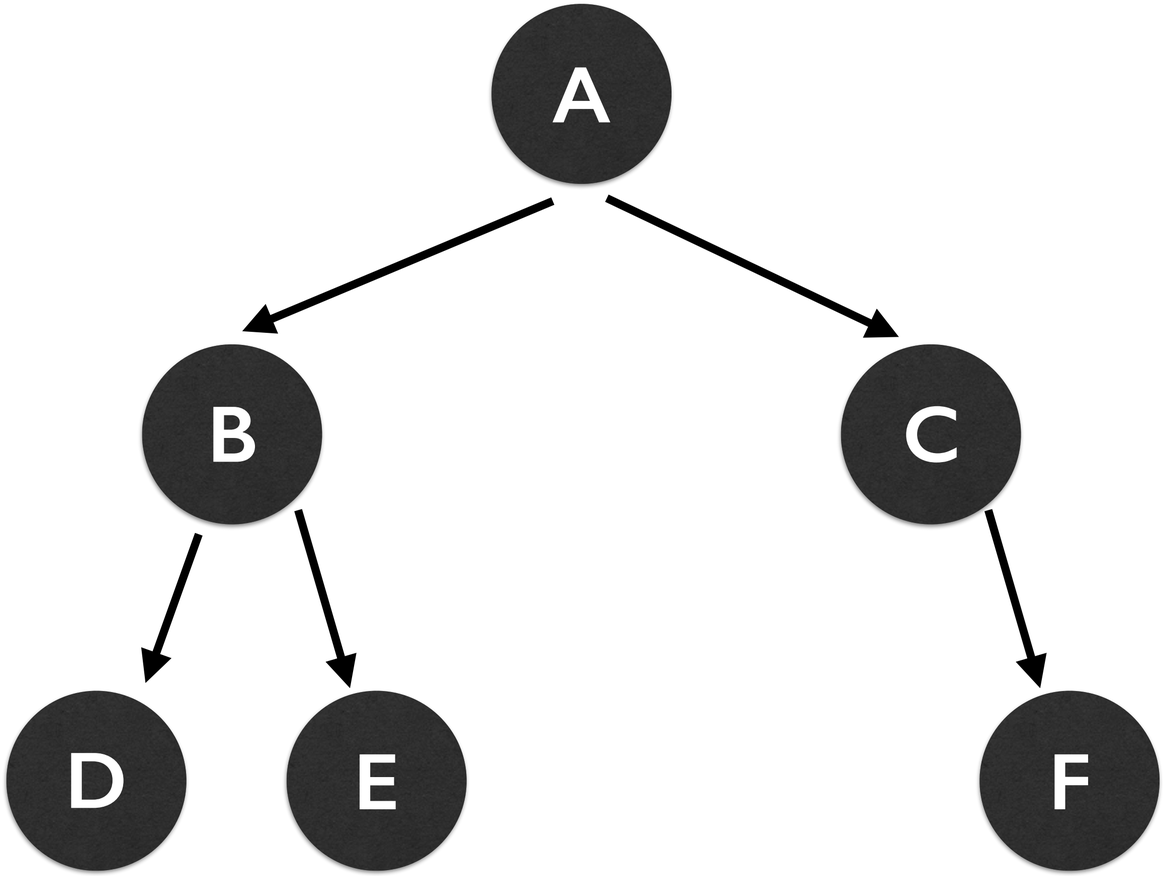
"Degenerate" trees are still trees

A tree of one node

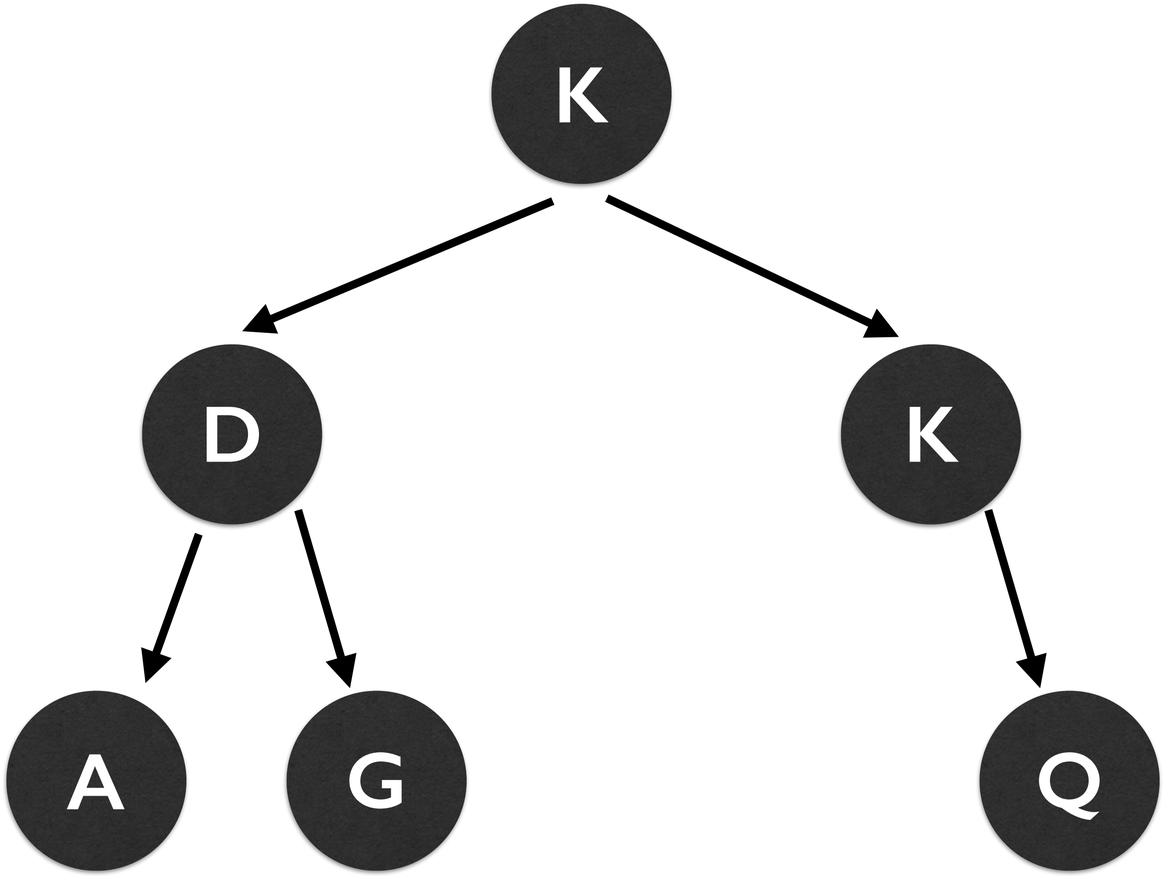


A tree of three nodes

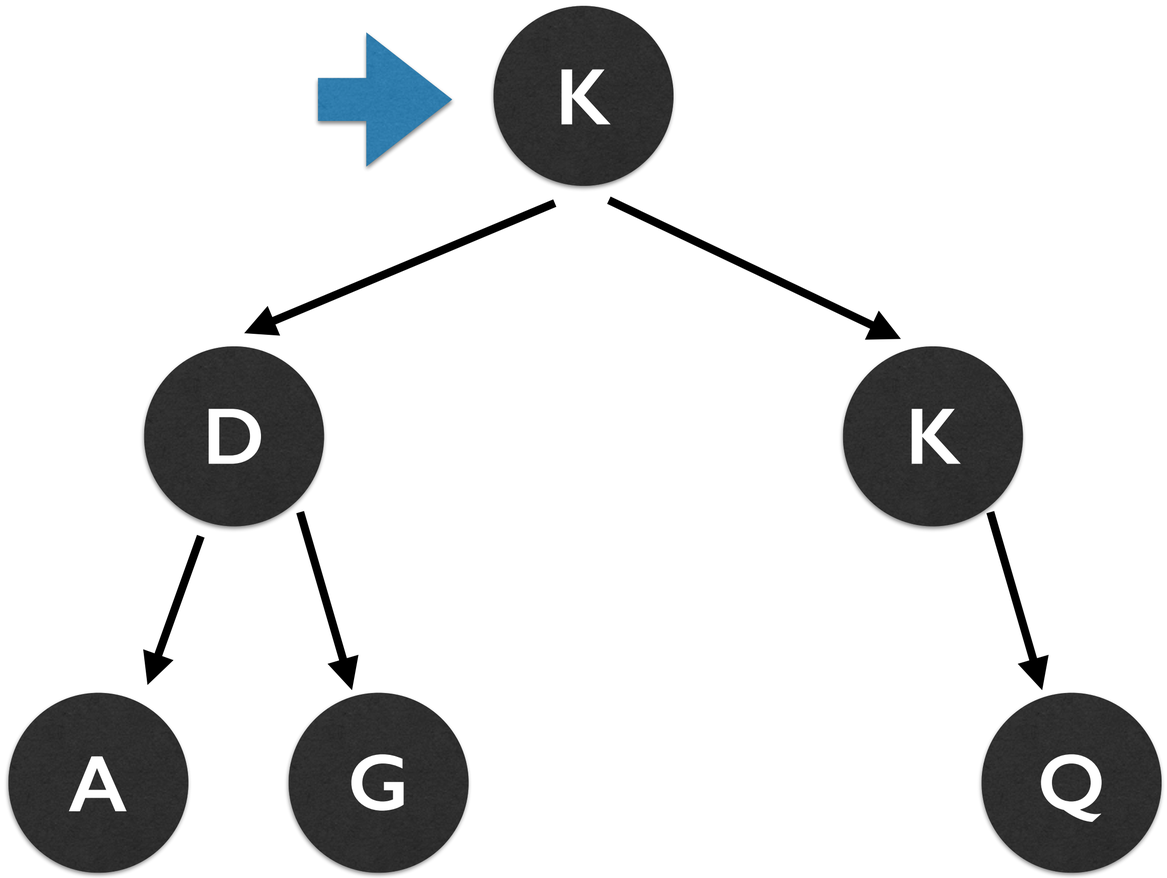
Binary Tree



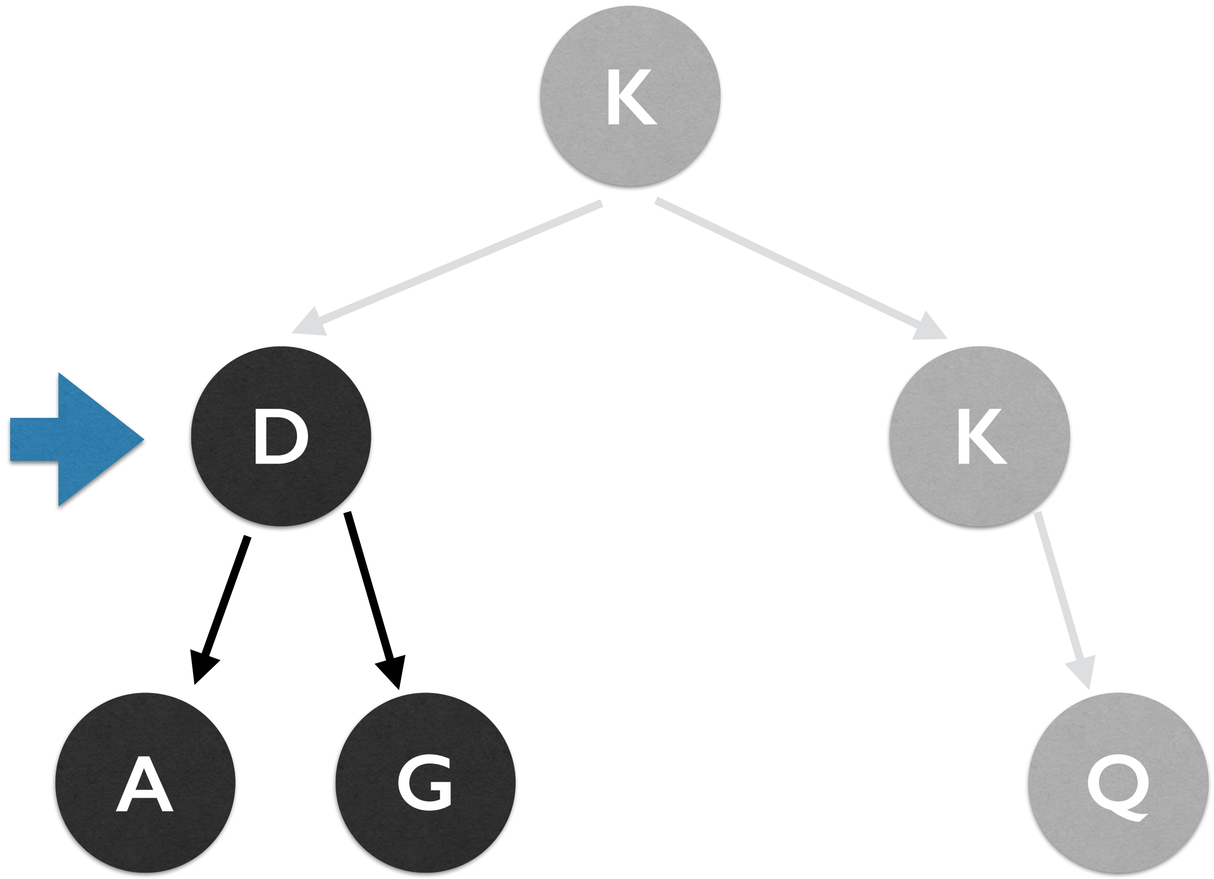
Binary *Search* Tree



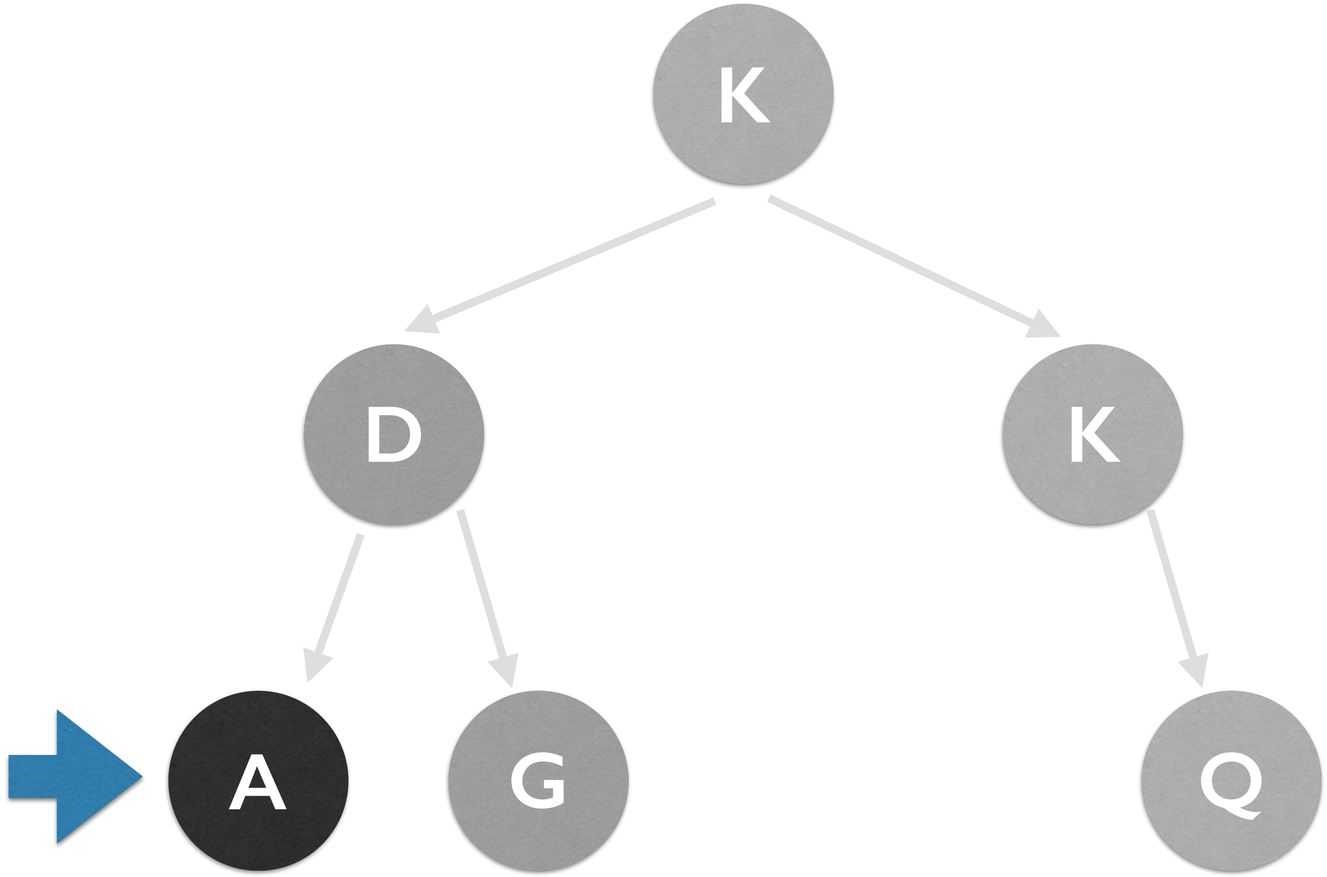
Binary Search Tree: Min Value?



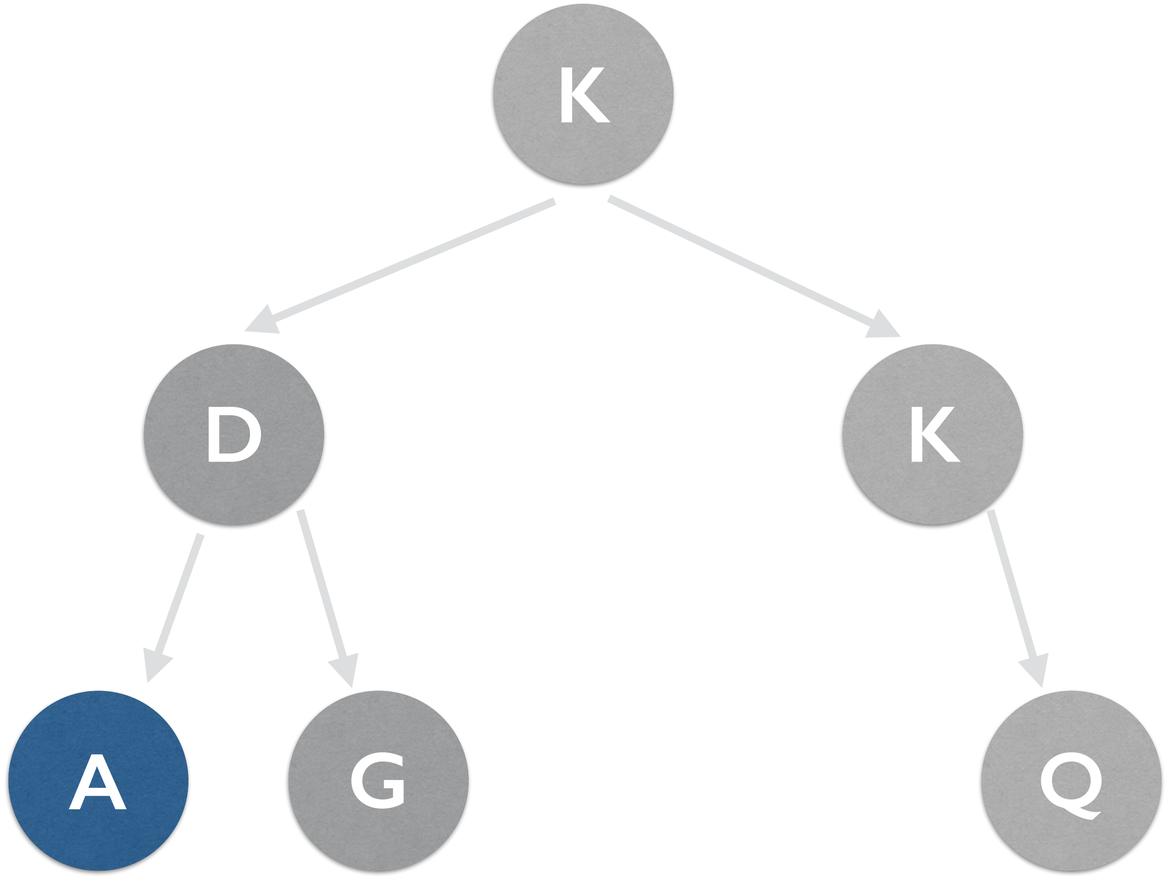
Binary Search Tree: Min Value?



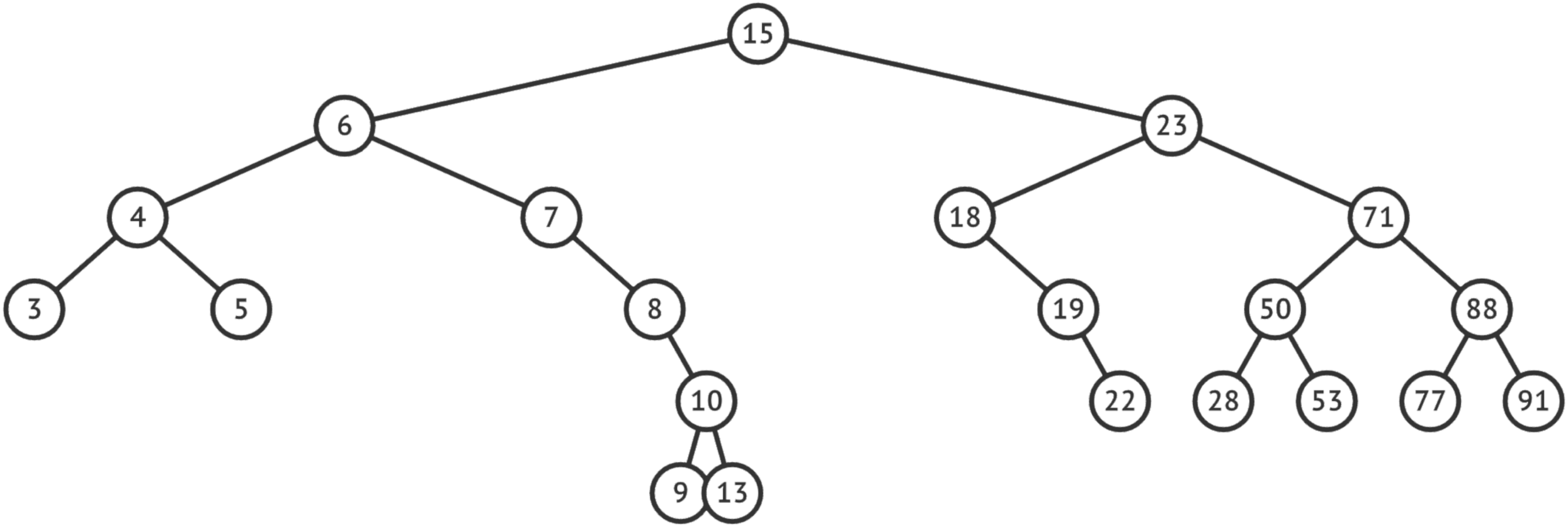
Binary Search Tree: Min Value?



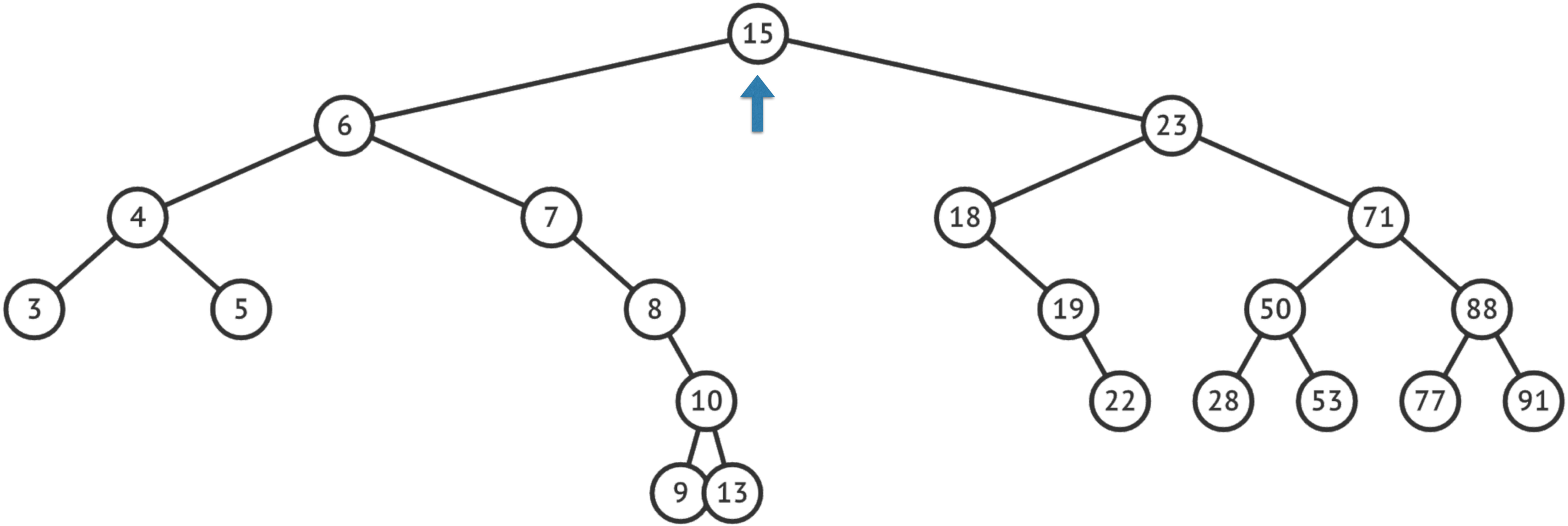
Binary Search Tree: Min Value?



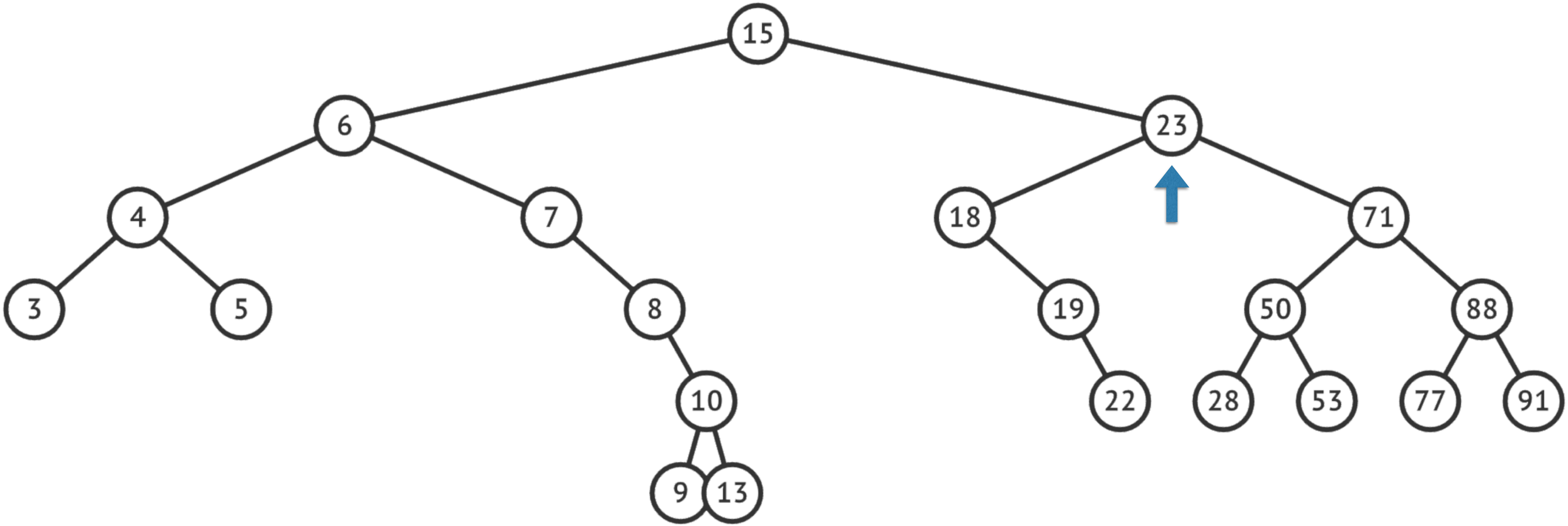
Does this BST contain the value 28?



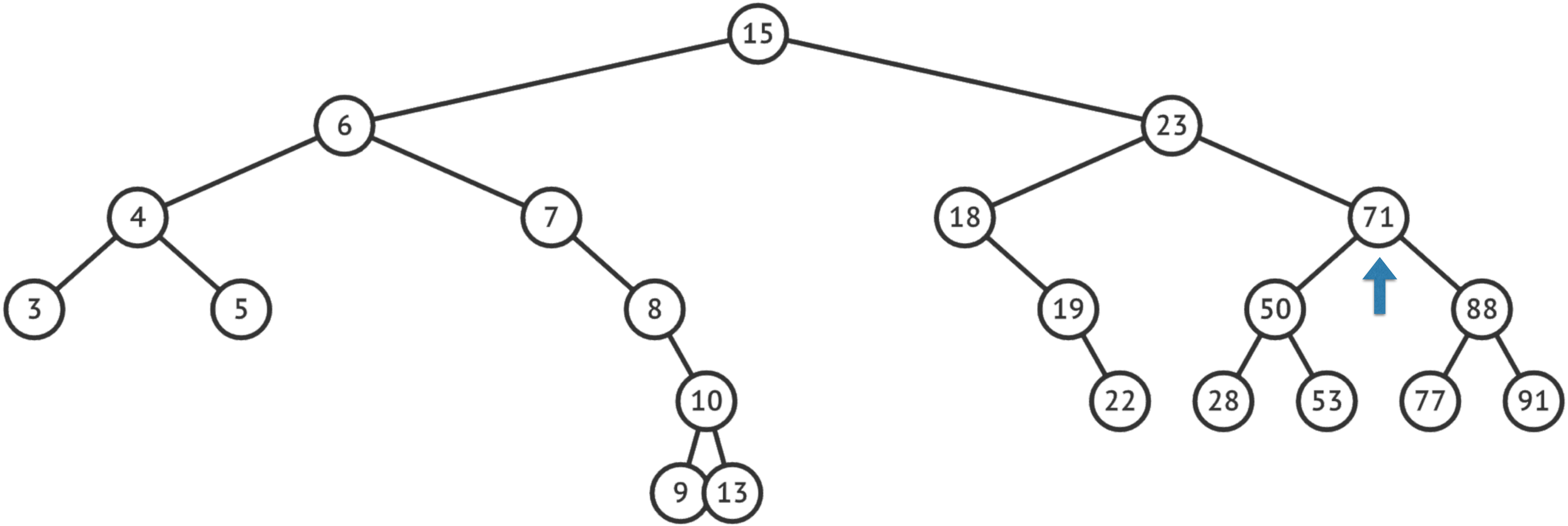
Does this BST contain the value 28?



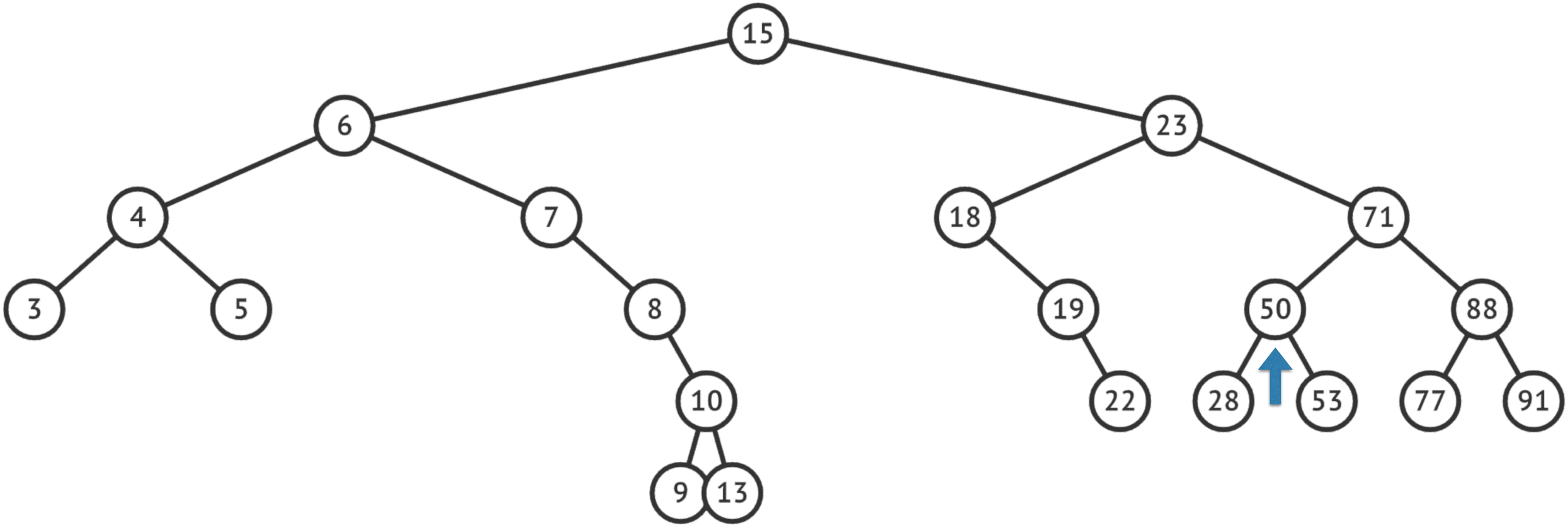
Does this BST contain the value 28?



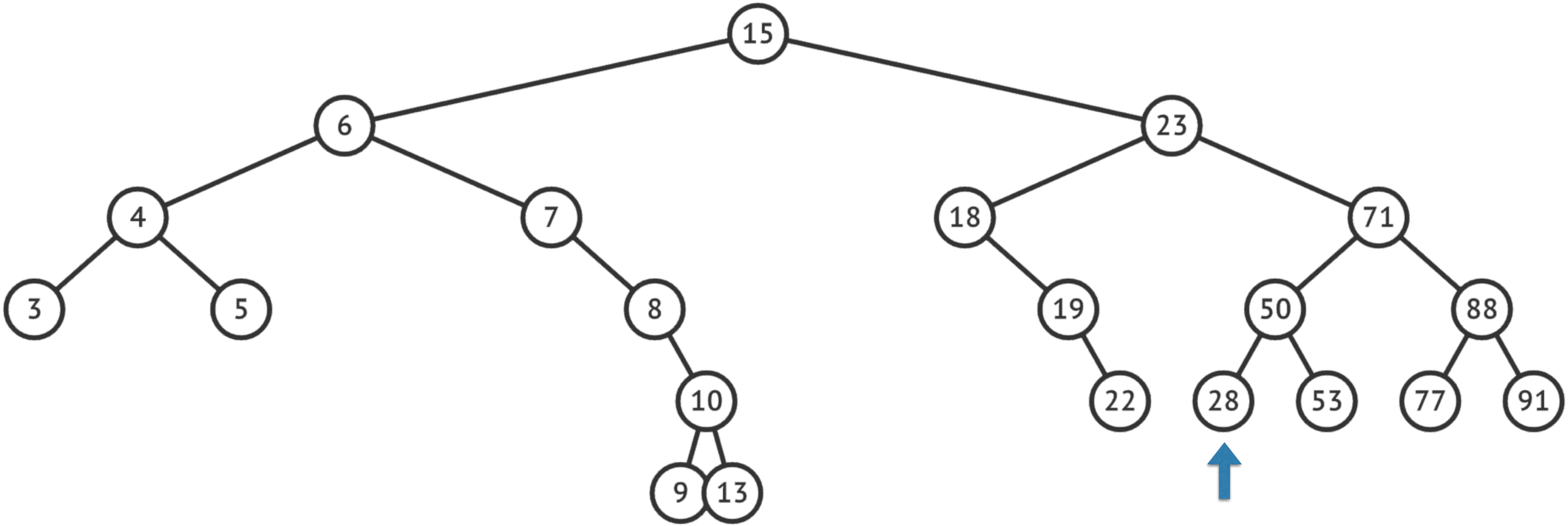
Does this BST contain the value 28?



Does this BST contain the value 28?



Does this BST contain the value 28?



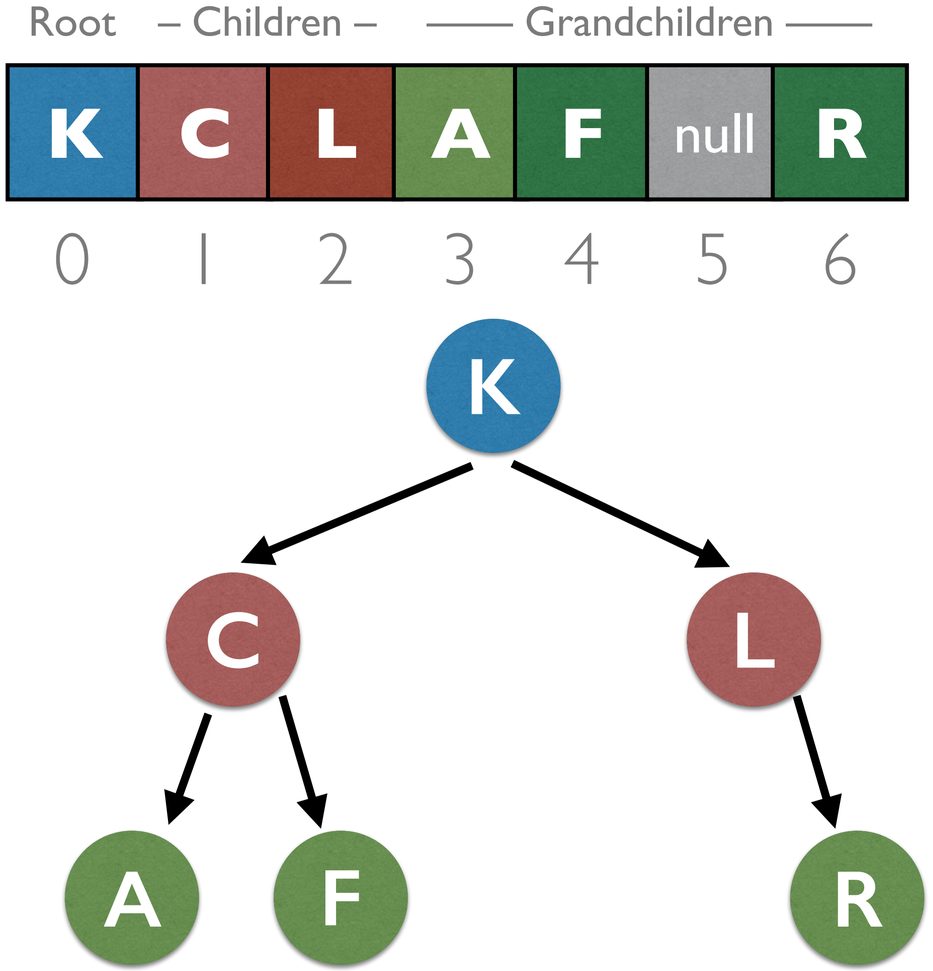
BST ADT

- **Root node satisfies ordering principle**
  - Left child value < root value <= right child value
- **Both children are BSTs (recursive definition - if insert works)**
  - Alternative (holistic) definition: all left nodes < root <= all right nodes
- **Operations**
  - **Insert** new values (respects ordering!)
  - **Find** existing values
  - **Delete** values
    (tricky to delete nodes in middle of tree, skipped in workshop)

How to implement this ADT?

...Maybe an array (seriously)?

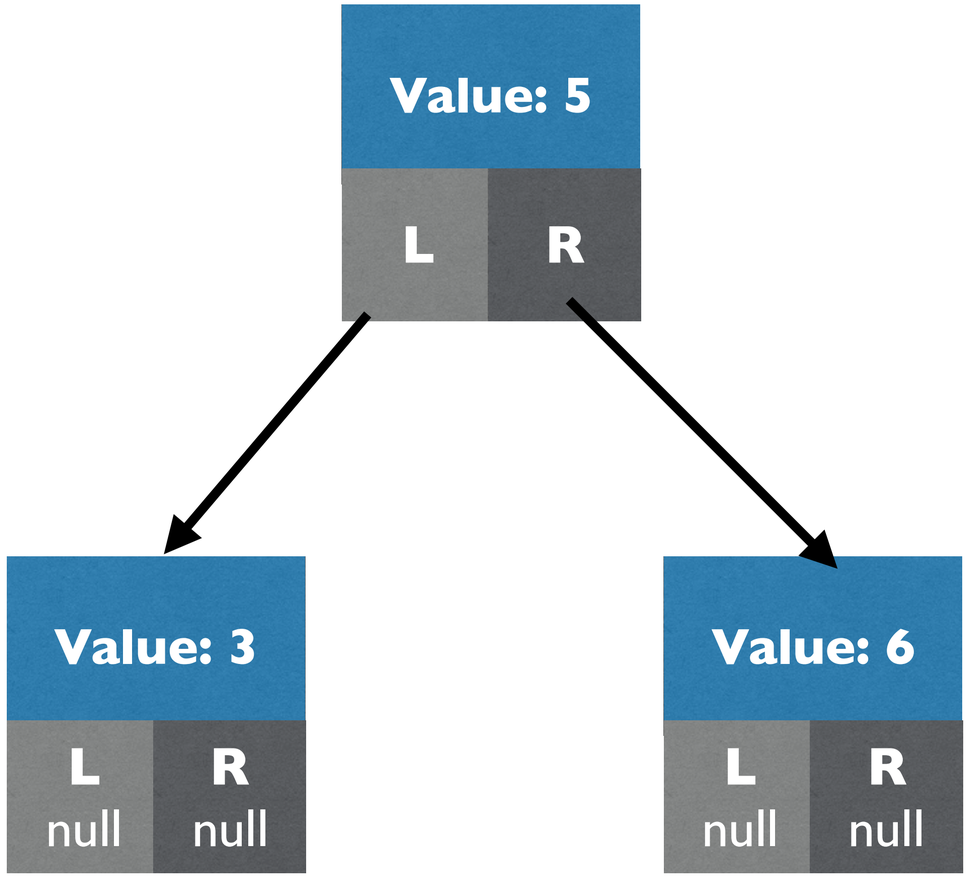
- The Tree ADT, with its talk of "nodes" and "references," seems so obviously to describe a data *structure* that it is perhaps confusing to tell the two apart.
- In fact, a tree can be stored in a few different ways. For example, if you knew your tree nodes always had at most two children, you could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



The 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 Tree DS.



Tree traversal

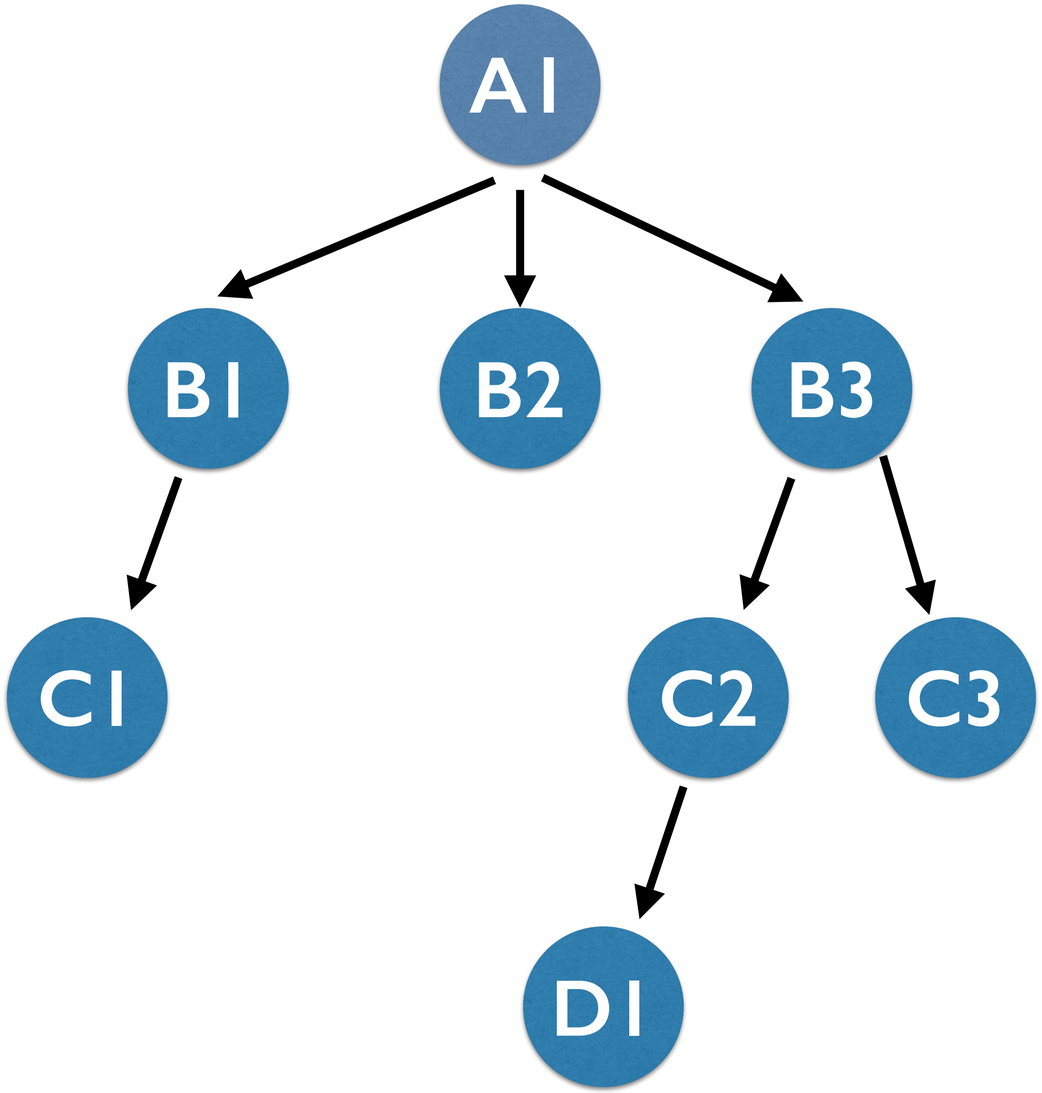
Traversal: visiting every node

- **Breadth-first search (level by level)**

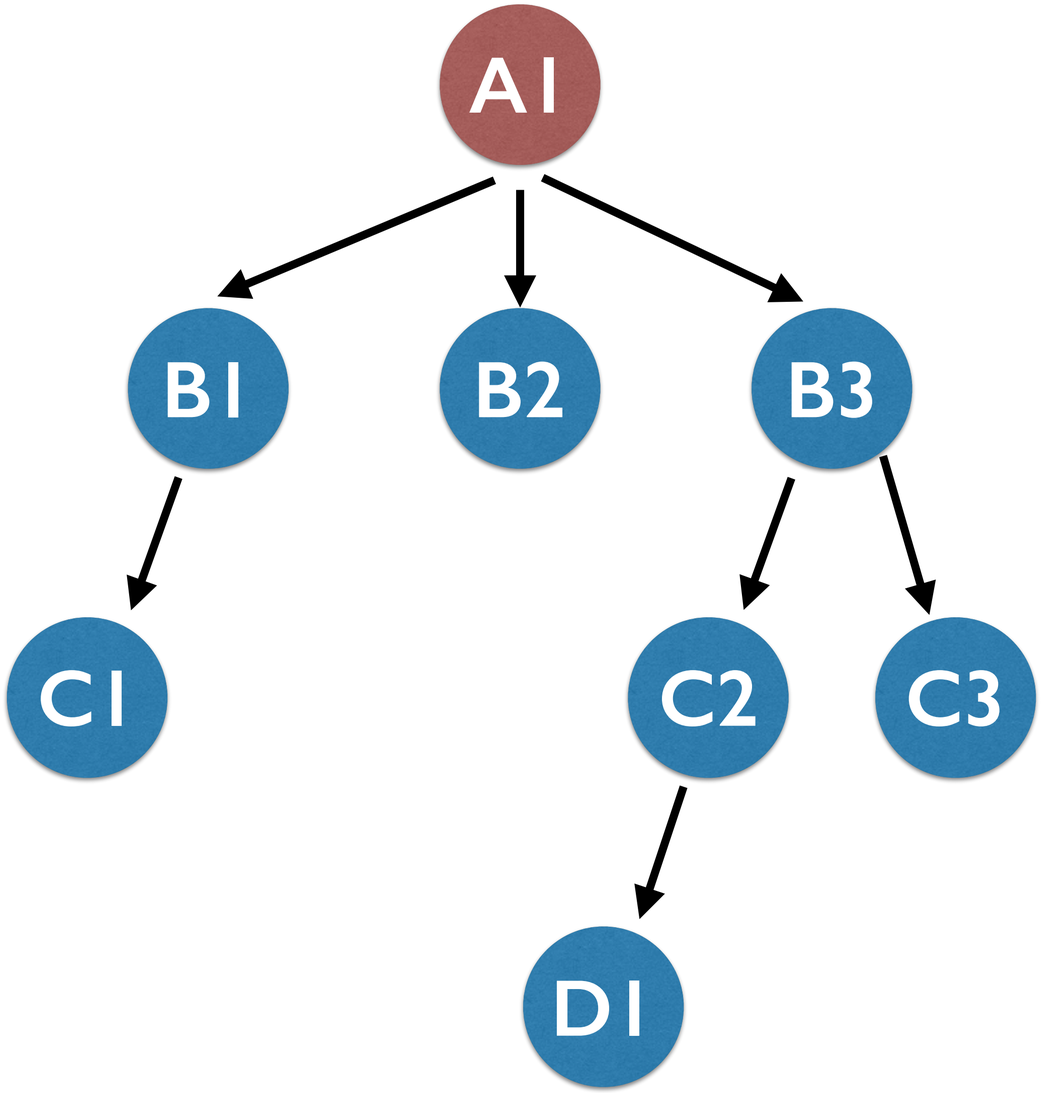
- **Depth-first search (branch by branch)**
  - Pre-order: process **root node**, process **left subtree**, process **right subtree**
  - **In-order**: process **left subtree**, process **root node**, process **right subtree**
  - Post-order: process **left subtree**, process **right subtree**, process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

Breadth-First

BFS

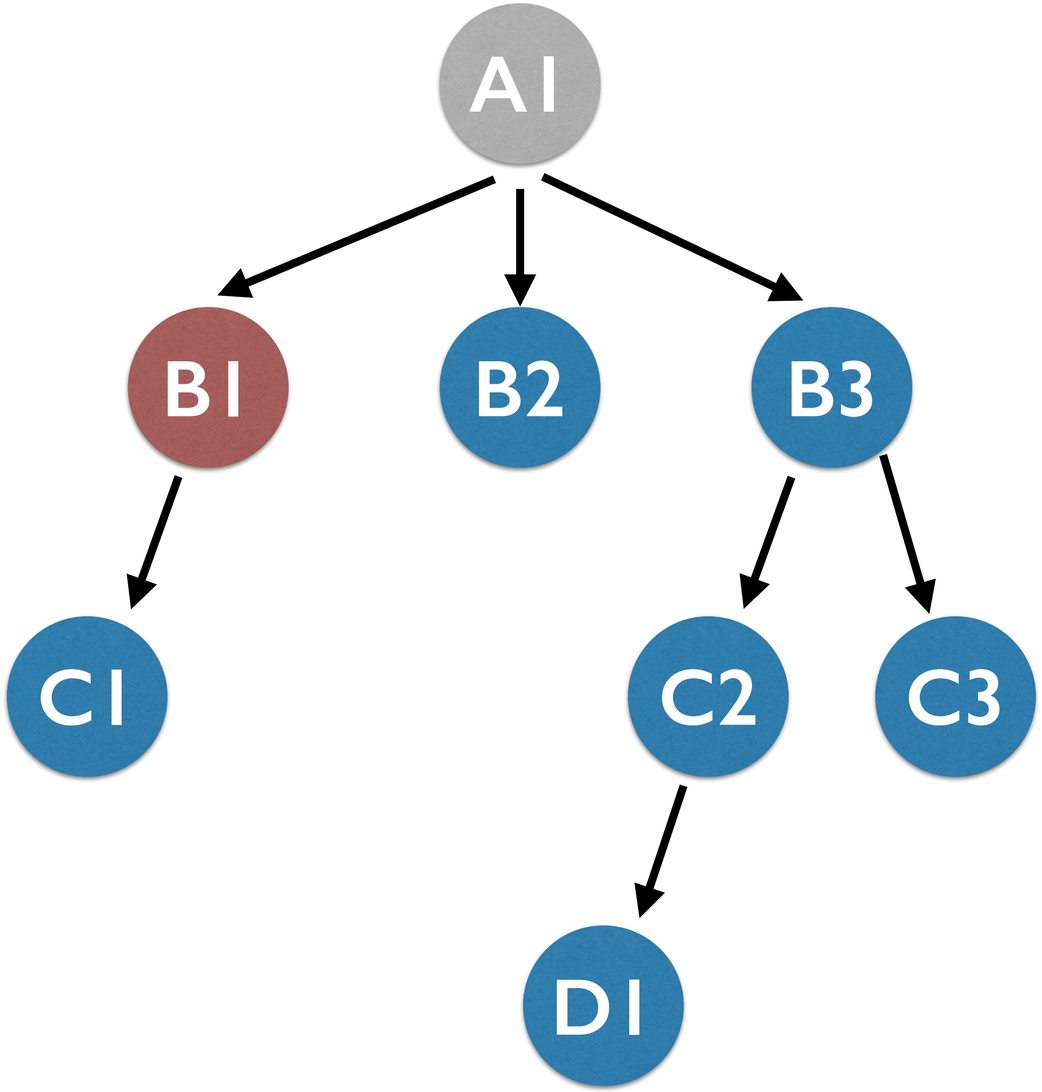


BFS



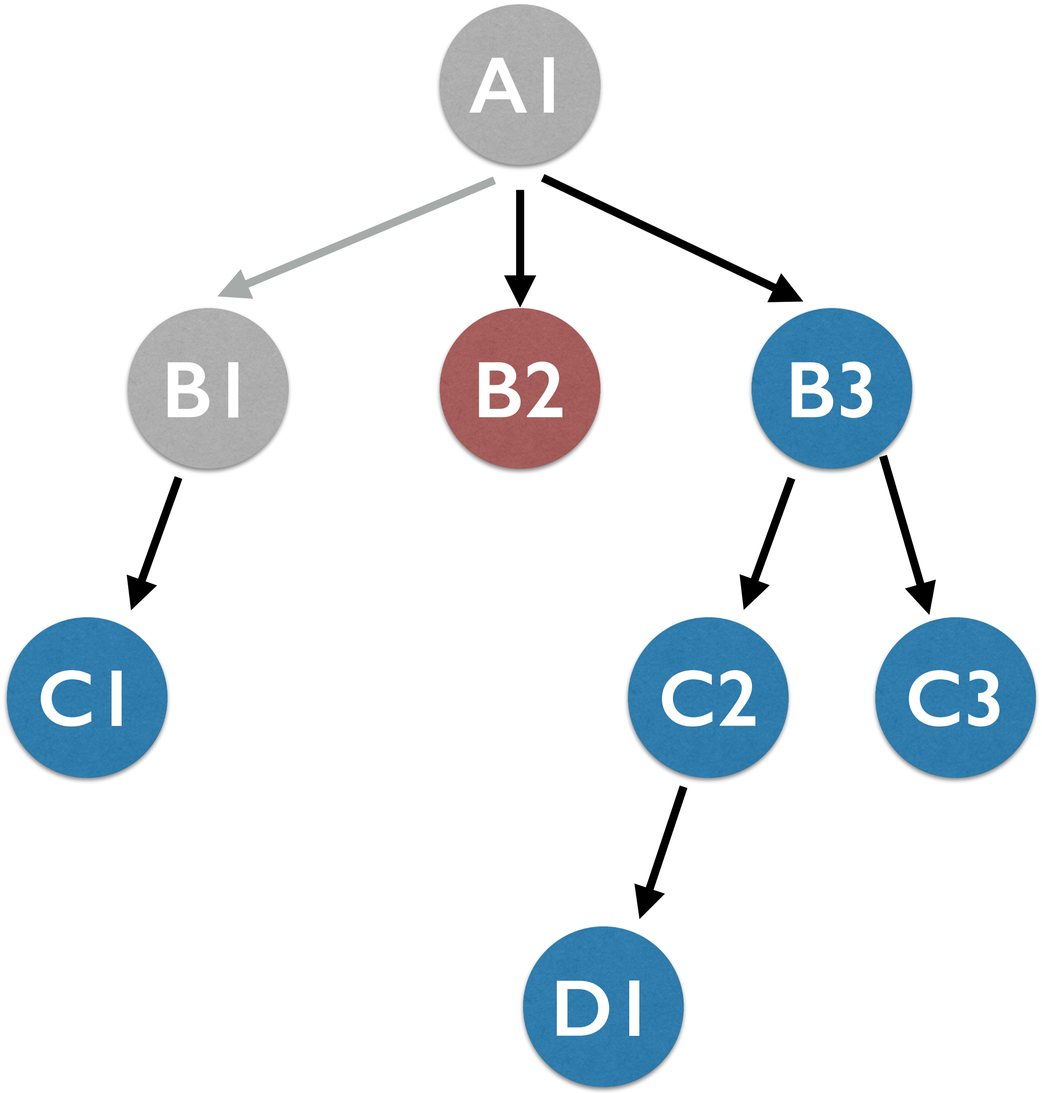
A1

BFS



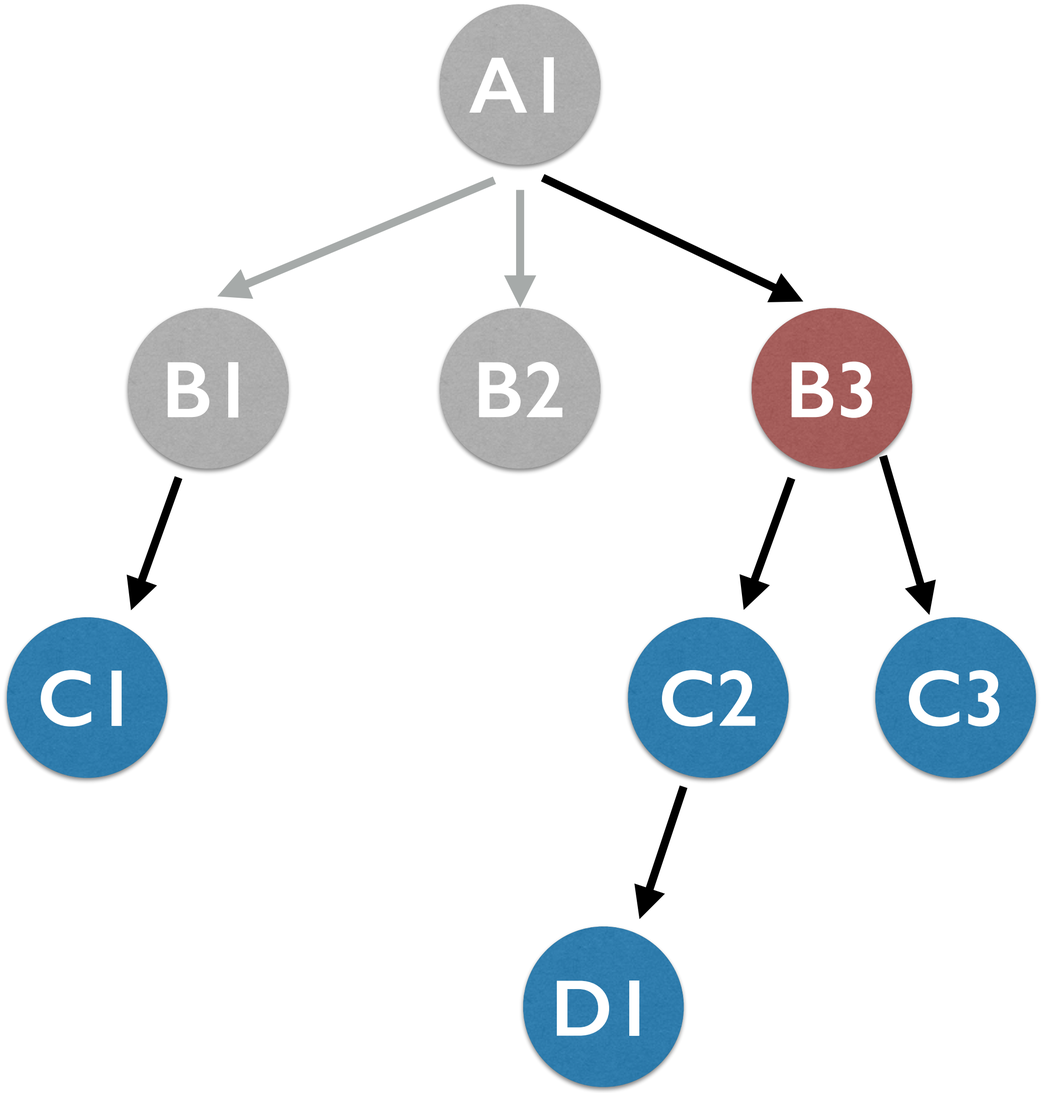
A1, B1

BFS



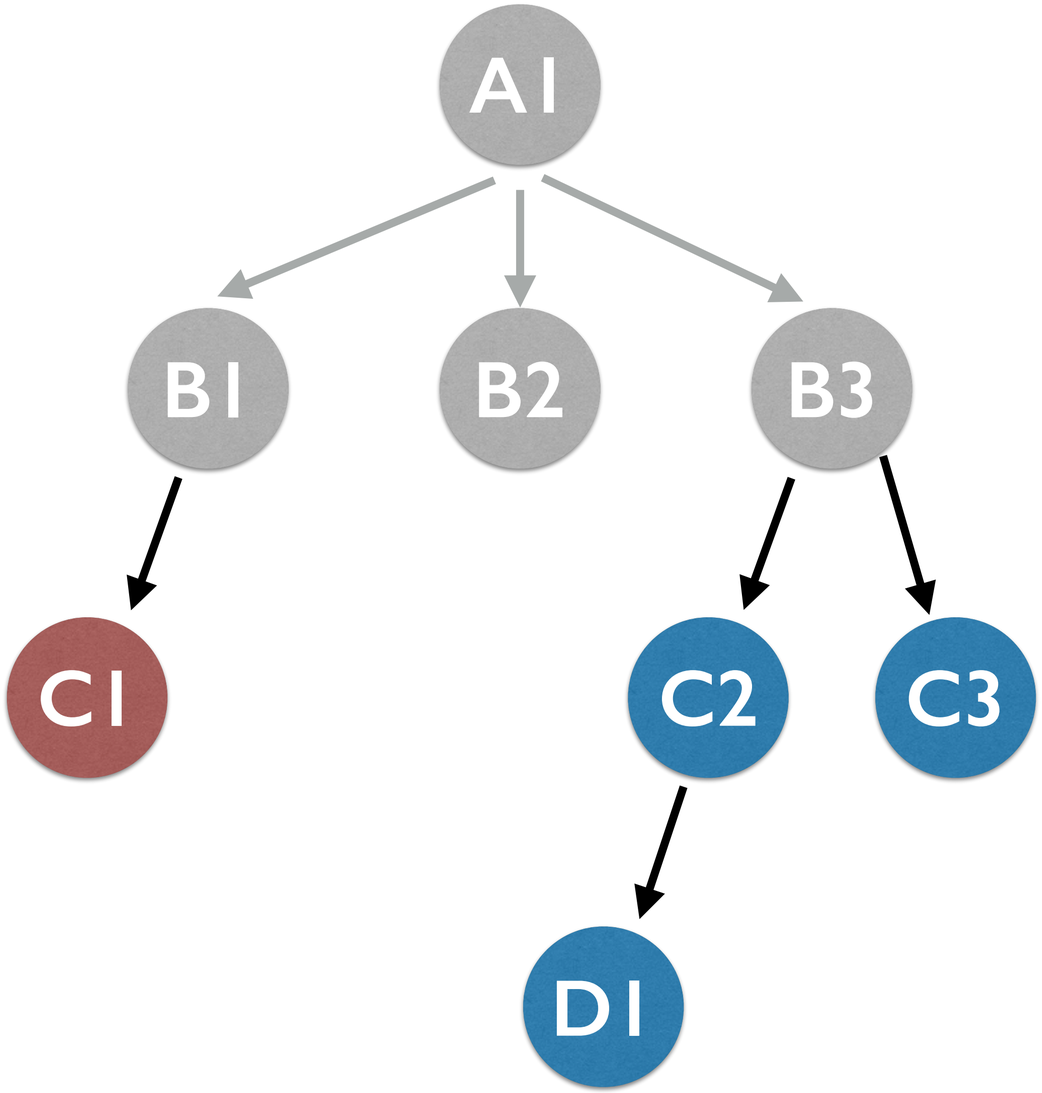
A1, B1, B2

BFS



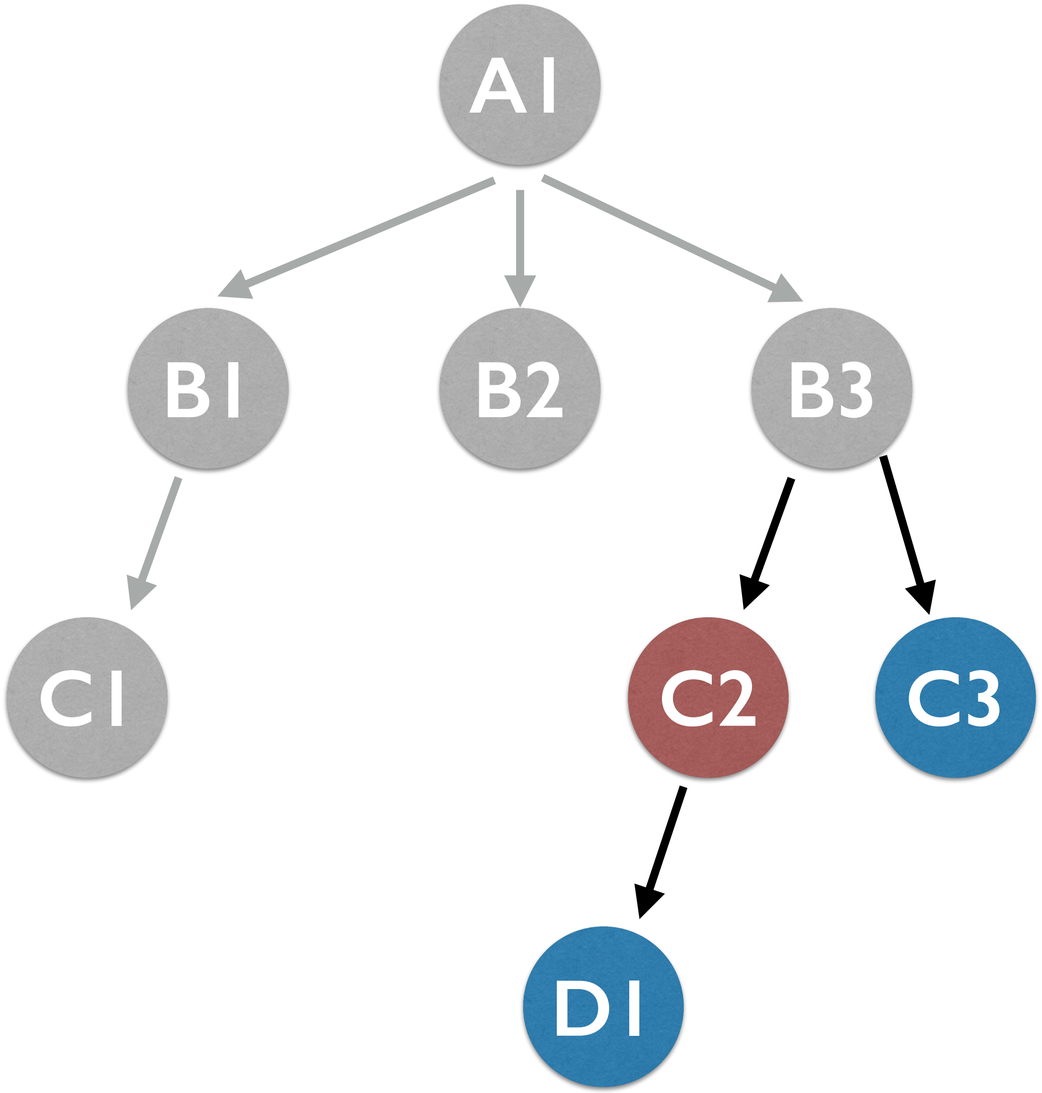
A1, B1, B2, B3

BFS



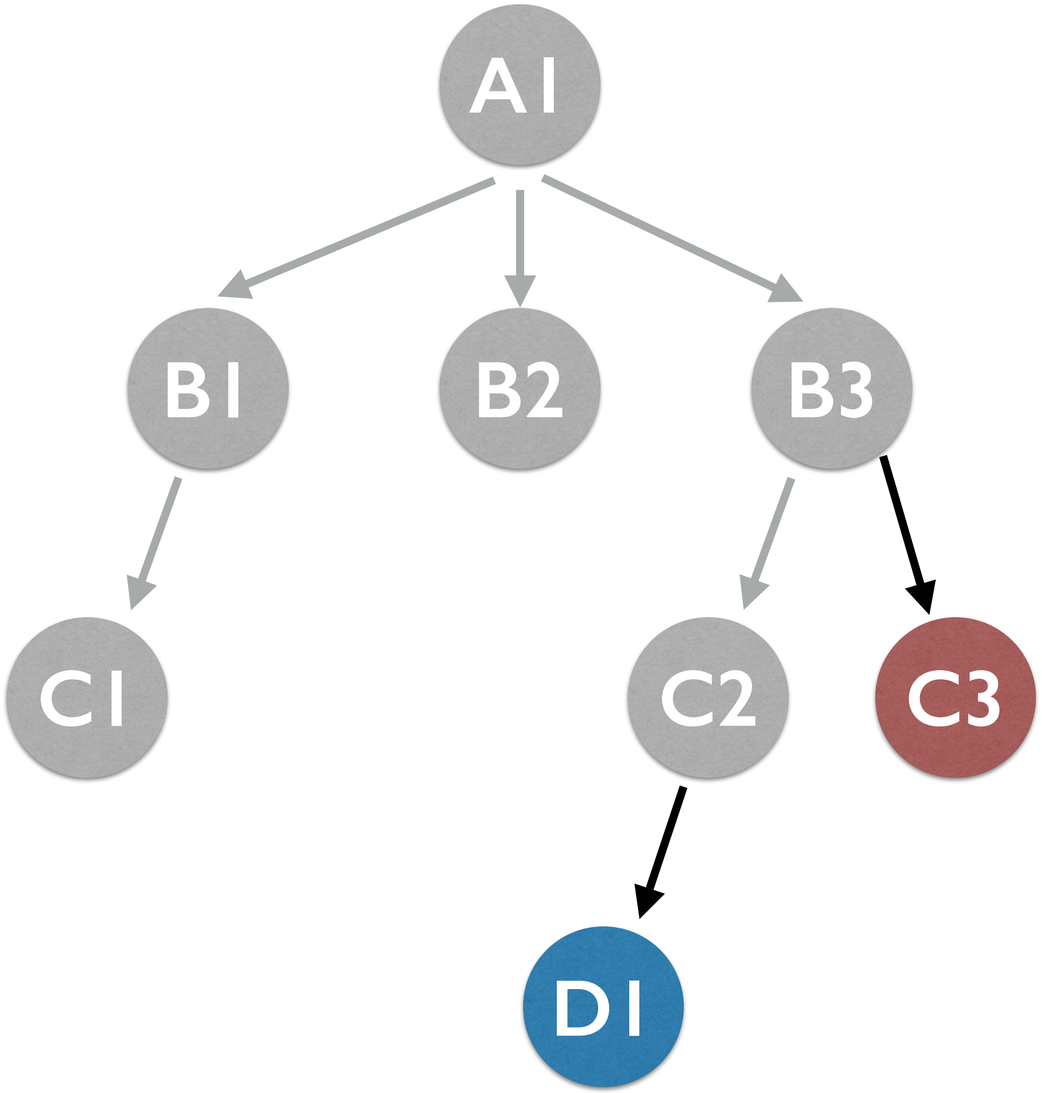
A1, B1, B2, B3, C1

BFS



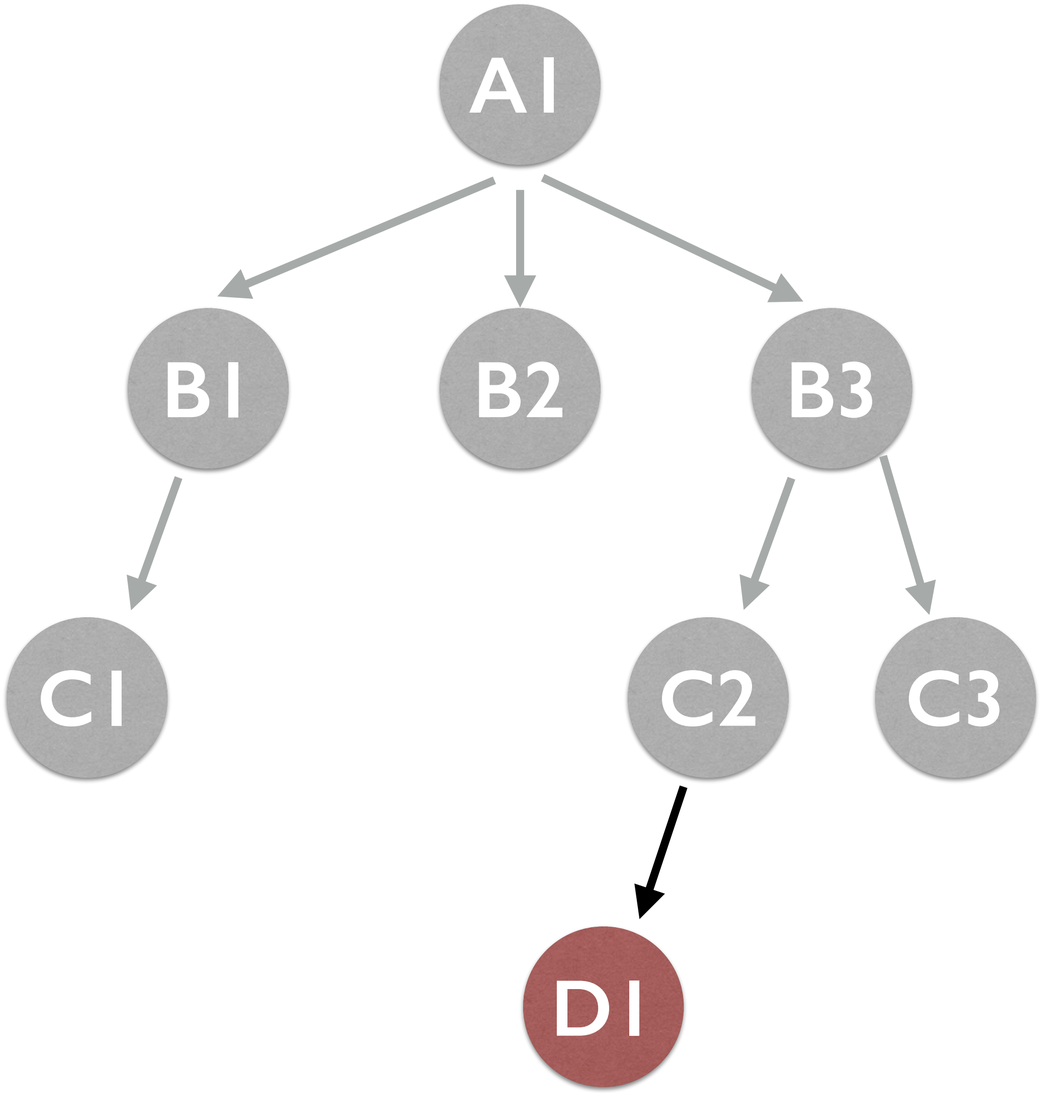
A1, B1, B2, B3, C1, C2

BFS



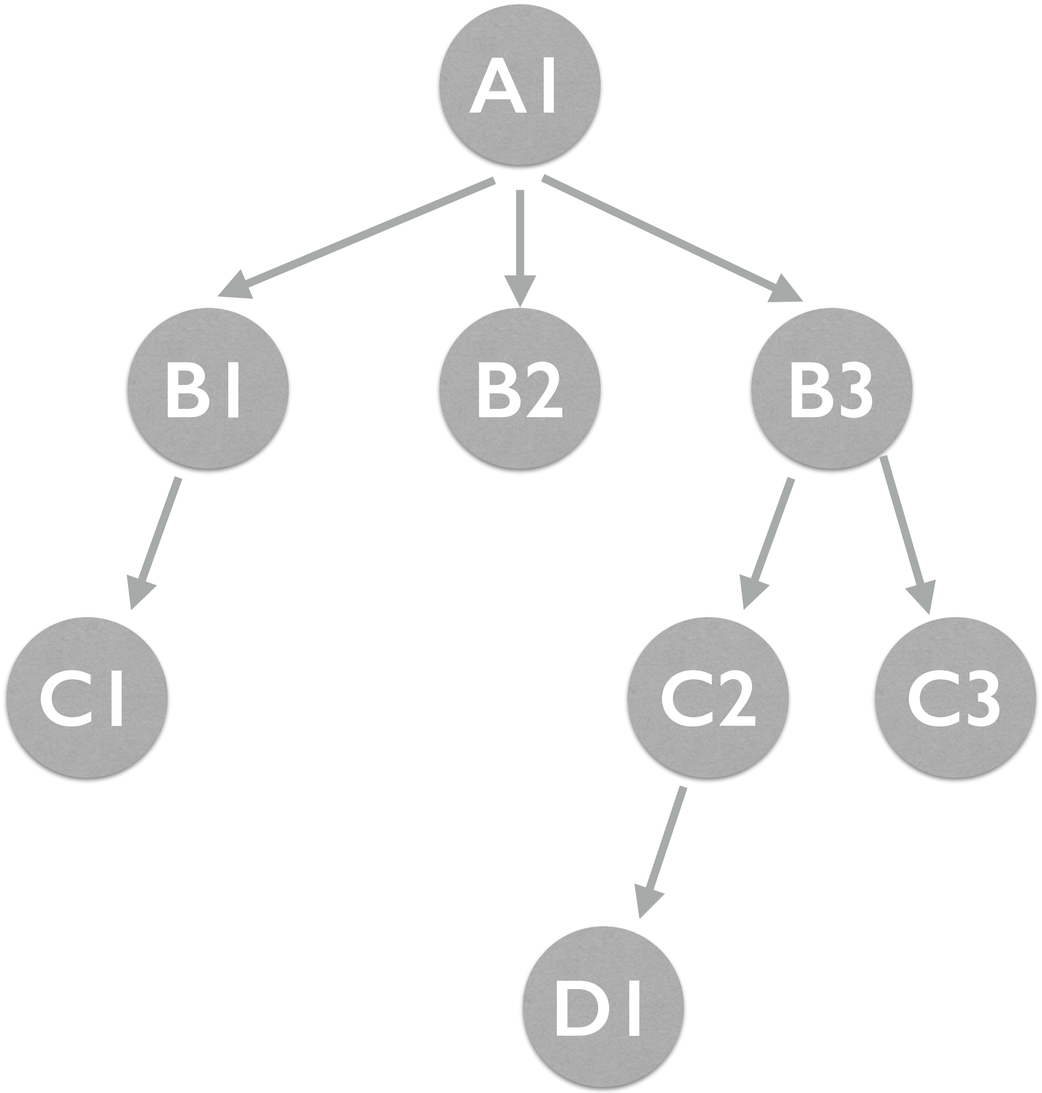
A1, B1, B2, B3, C1, C2, C3

BFS



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

BFS

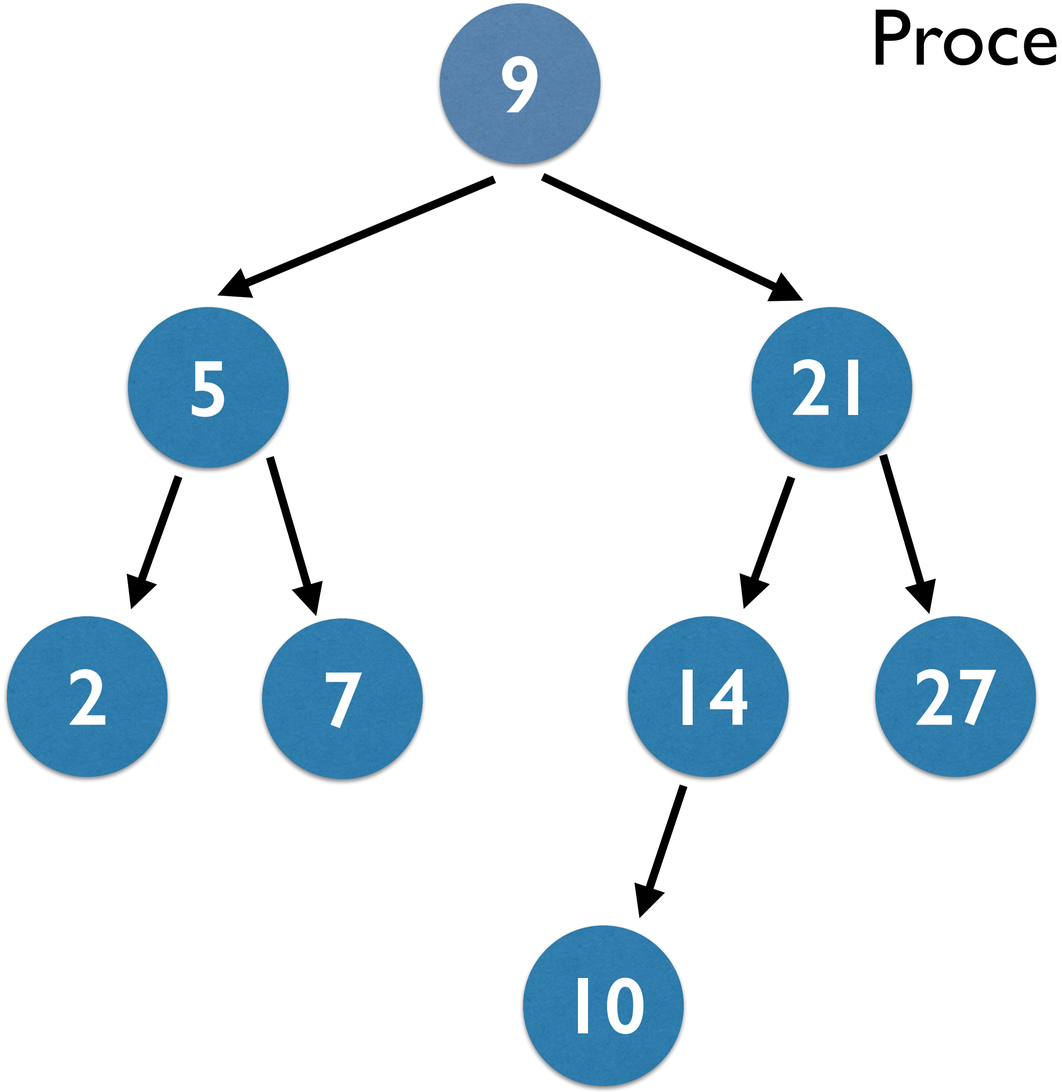


A1, B1, B2, B3, C1, C2, C3, D1

- 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!

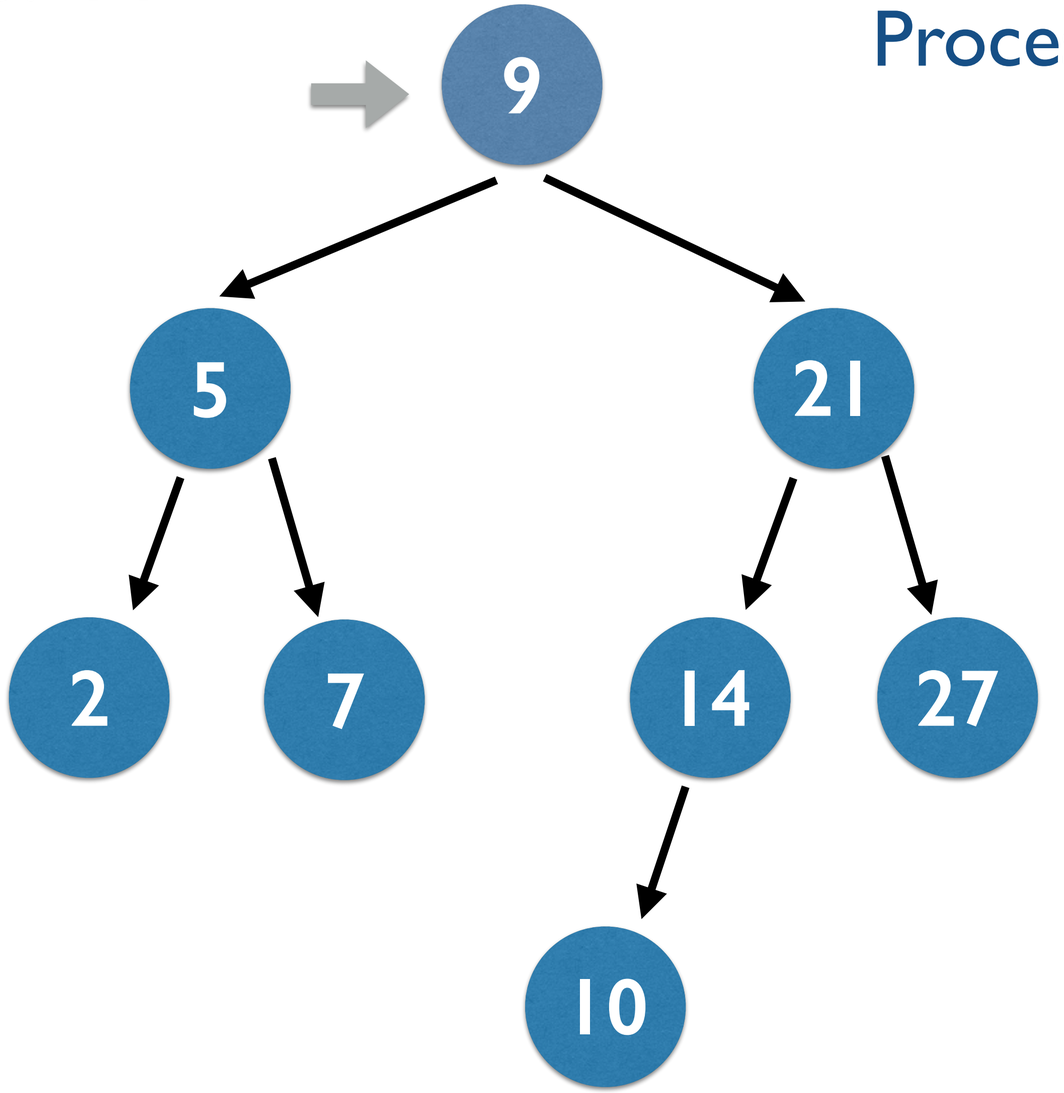
Depth First: In-Order

DFS: IN



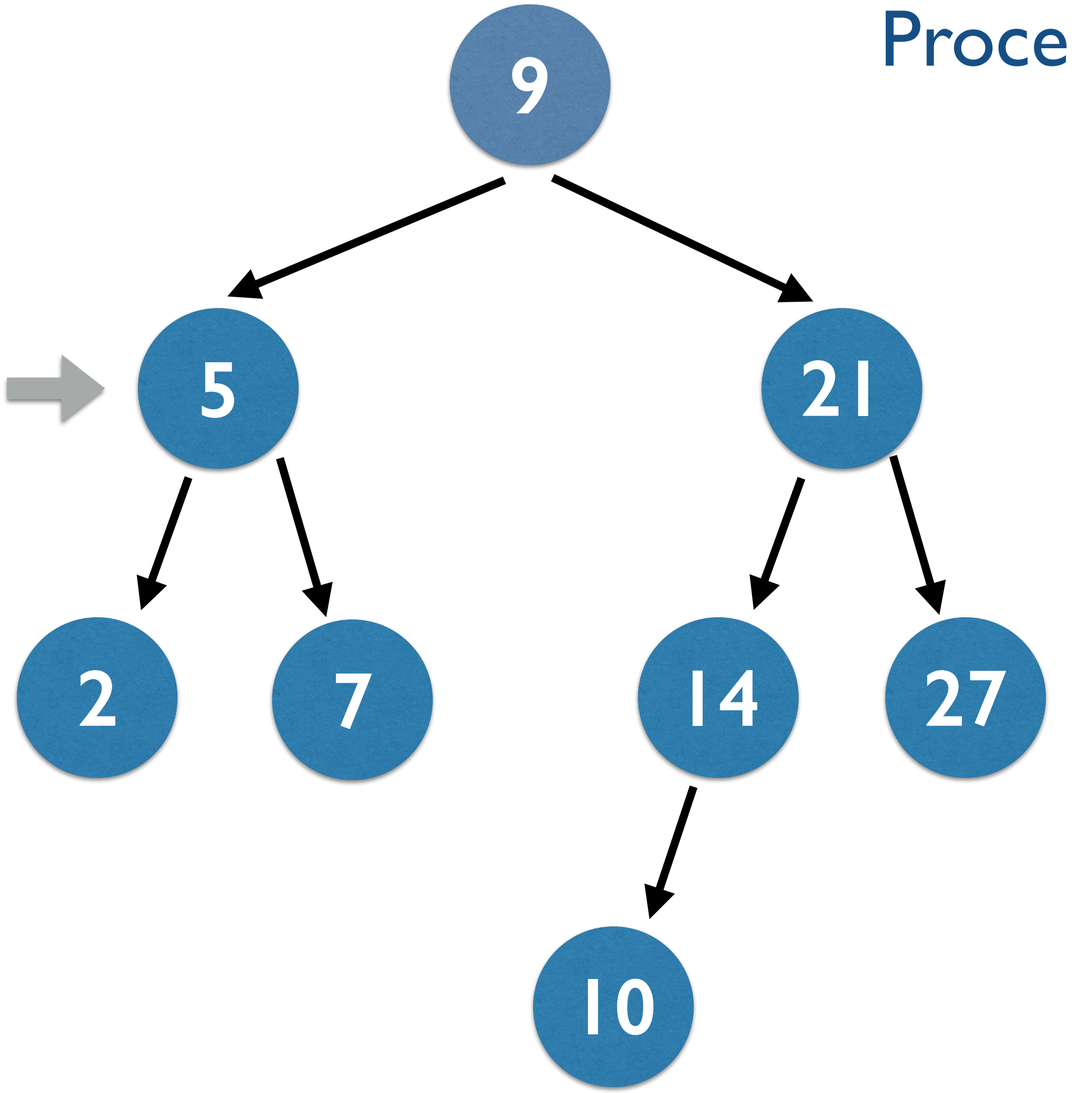
Process left · Process root · Process right

DFS: IN



Process left · Process root · Process right

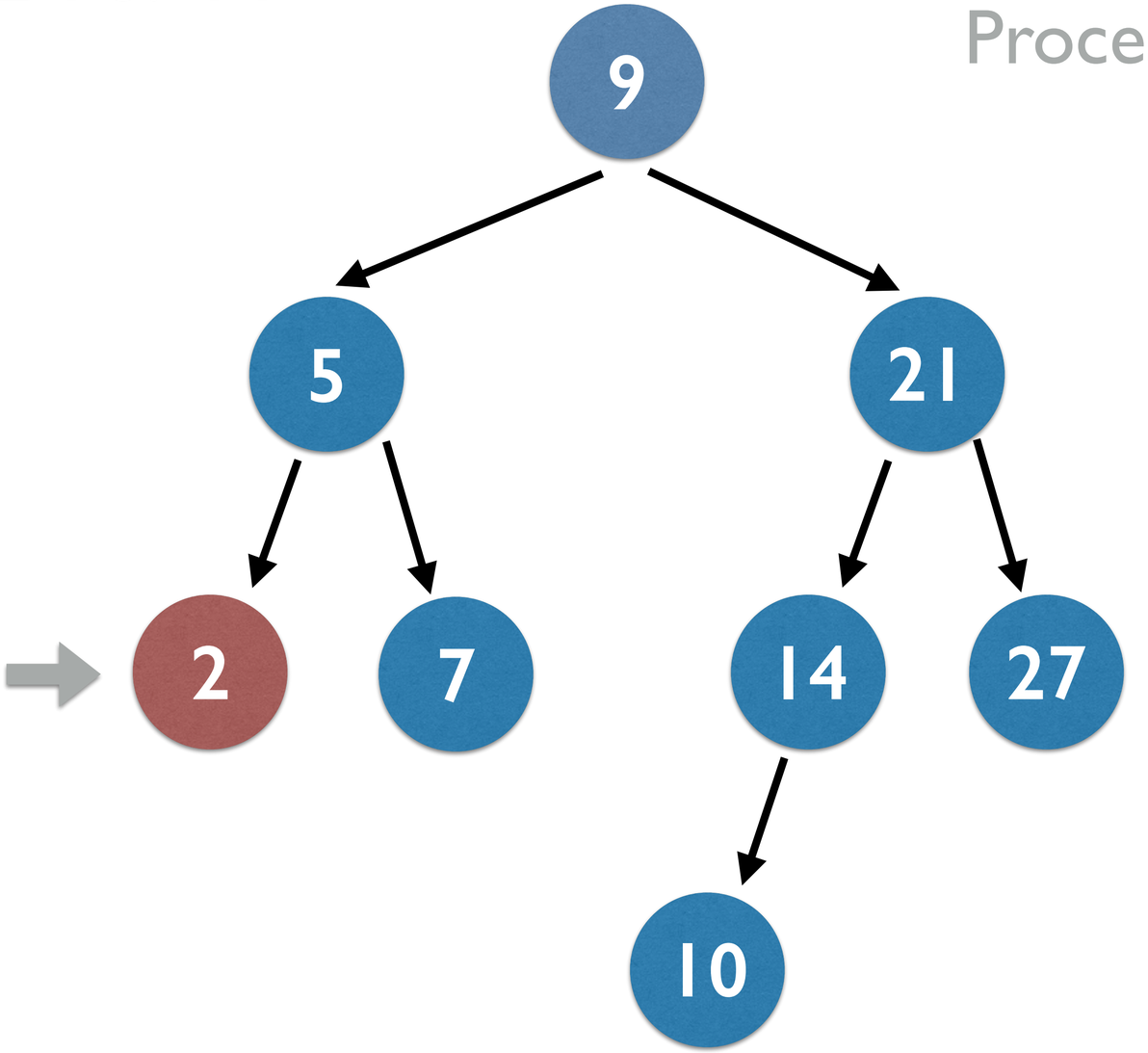
DFS: IN



Process left · Process root · Process right

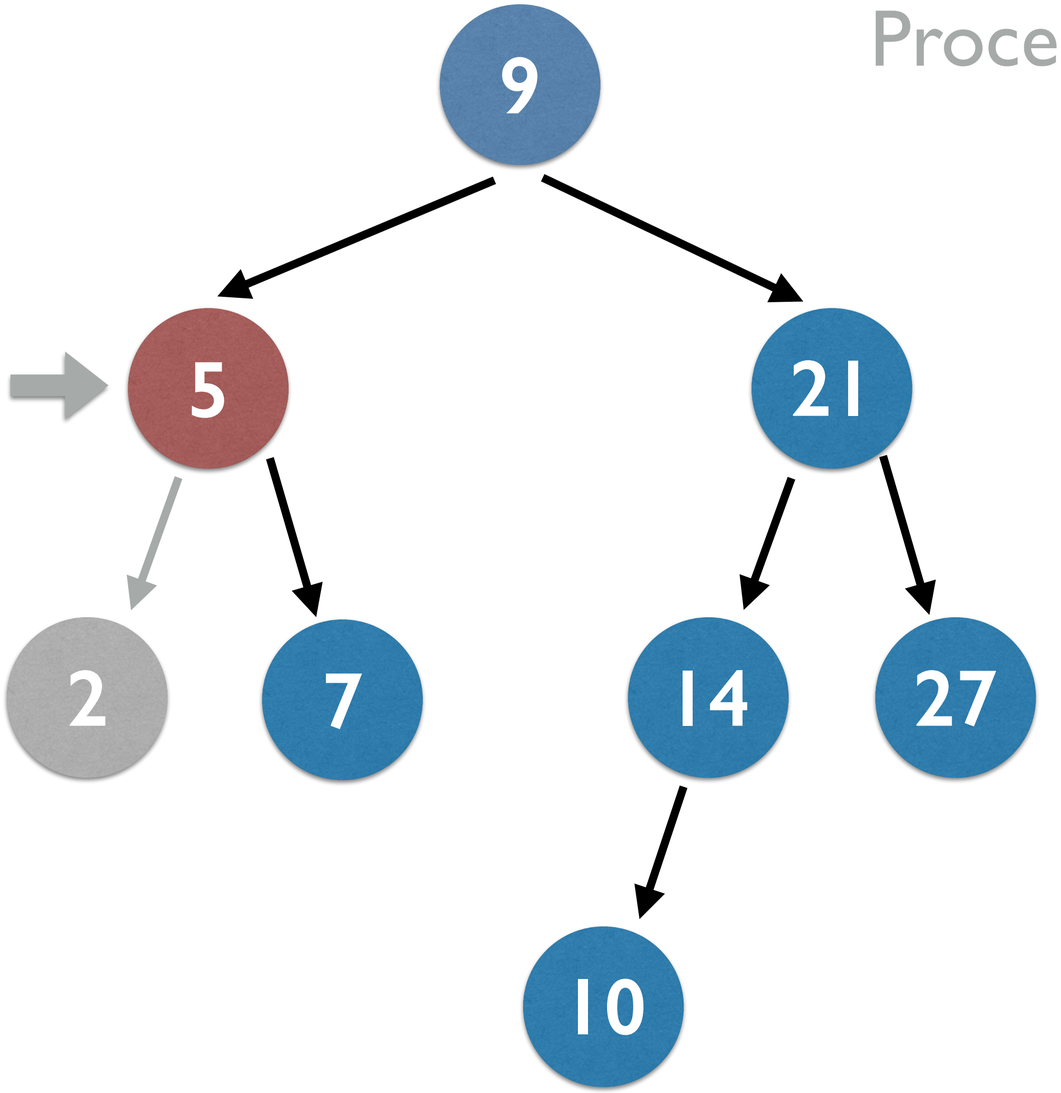
DFS: IN

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



2

DFS: IN

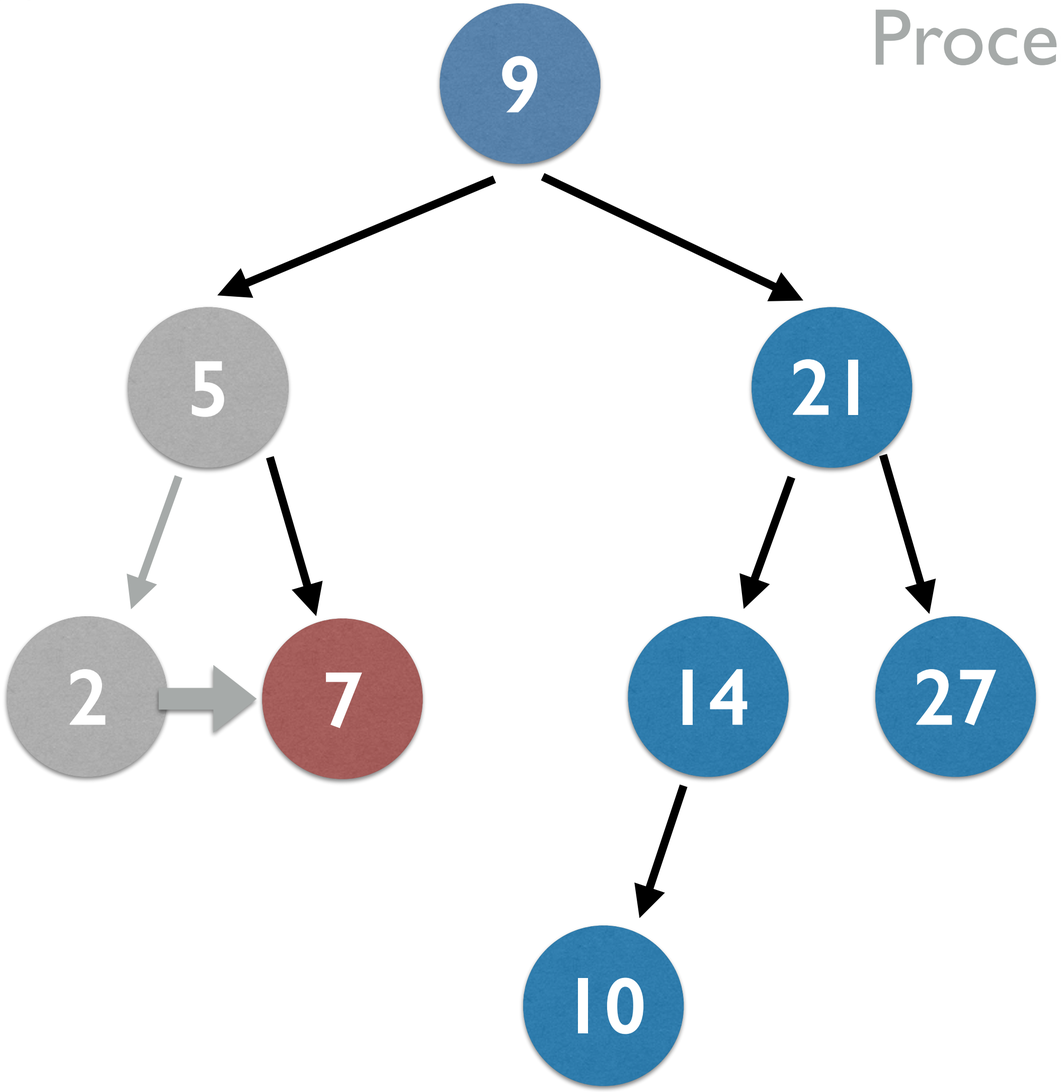


Process left · Process root · Process right

2, 5

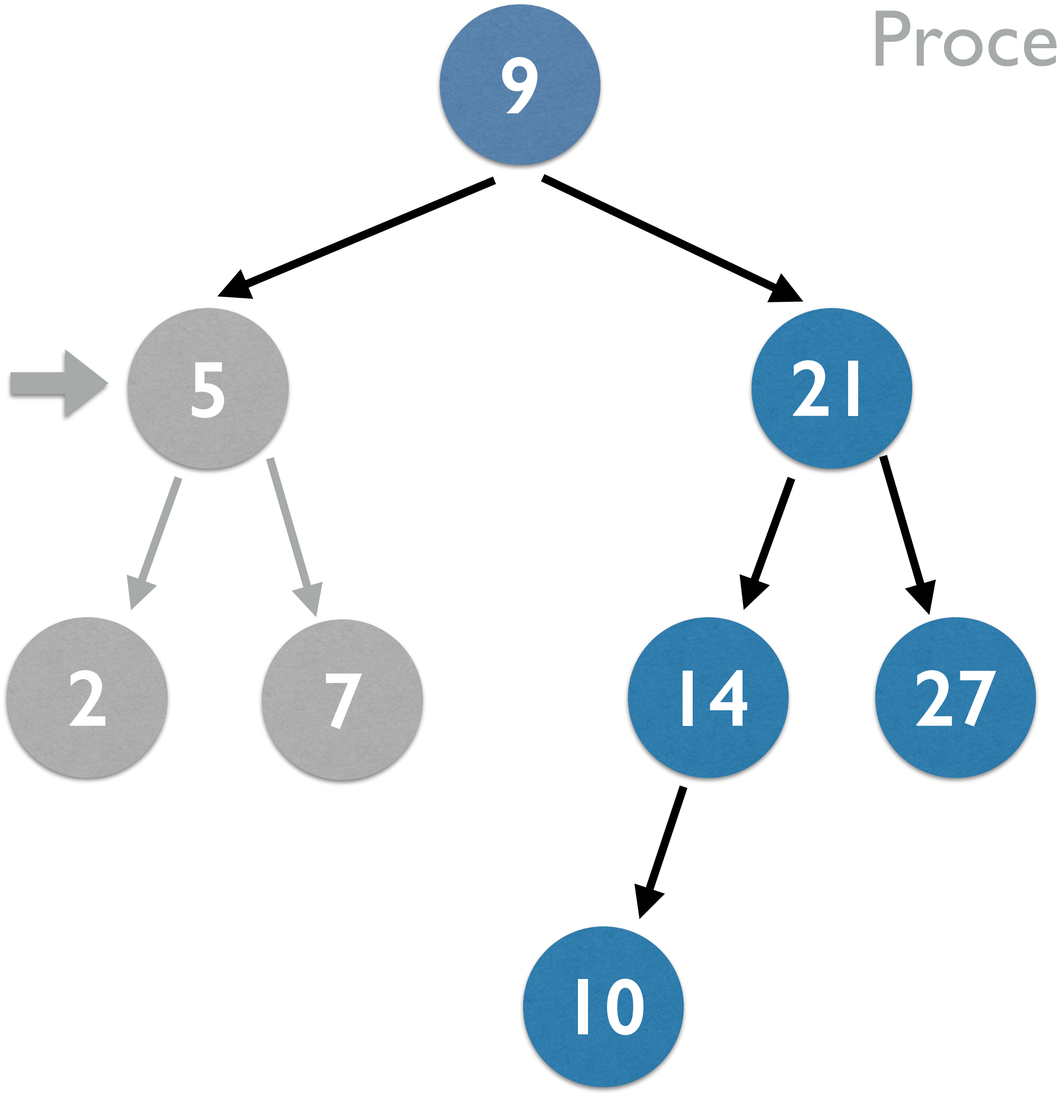
DFS: IN

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



2, 5, 7

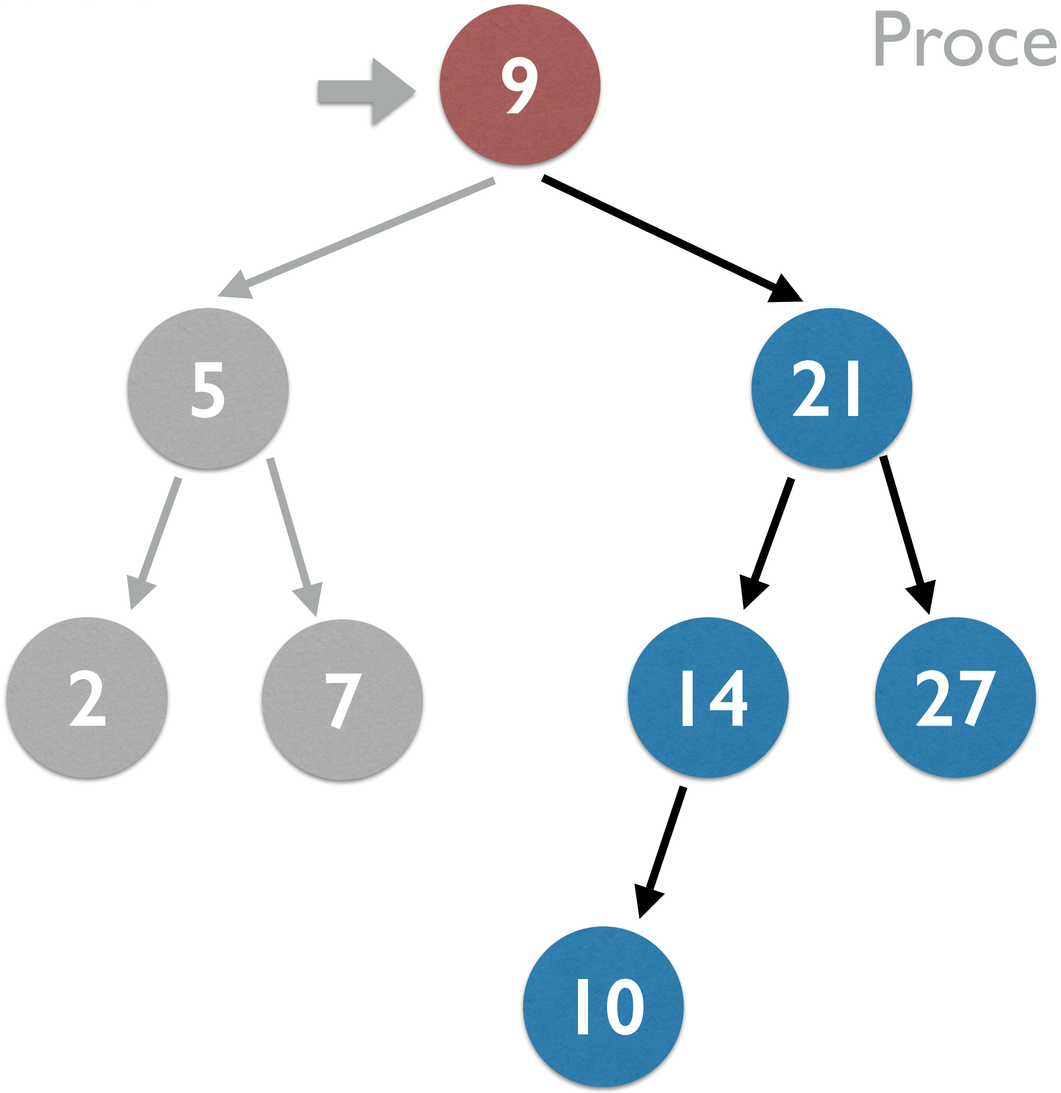
DFS: IN



Process left · Process root · Process right

2, 5, 7

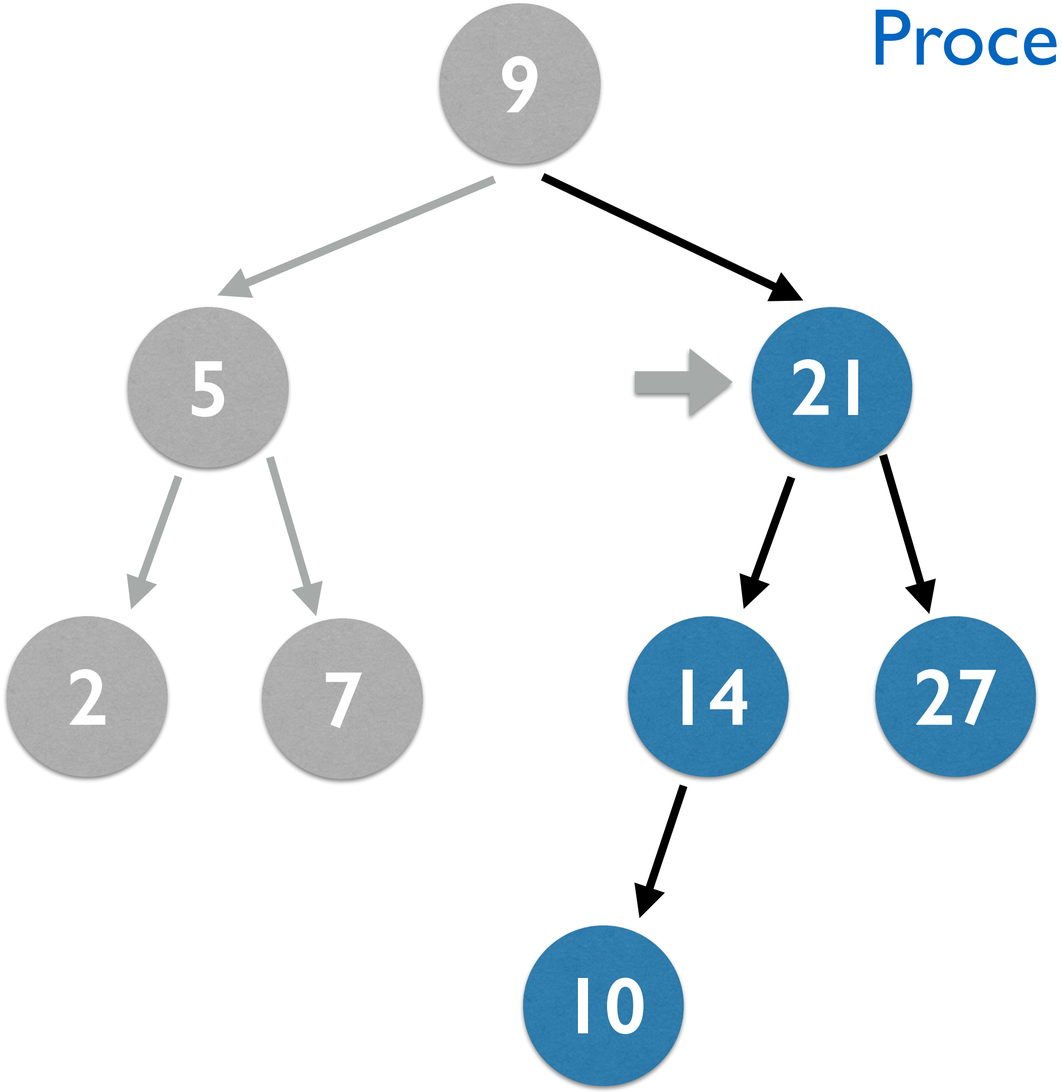
DFS: IN



Process left · Process root · Process right

2, 5, 7, 9

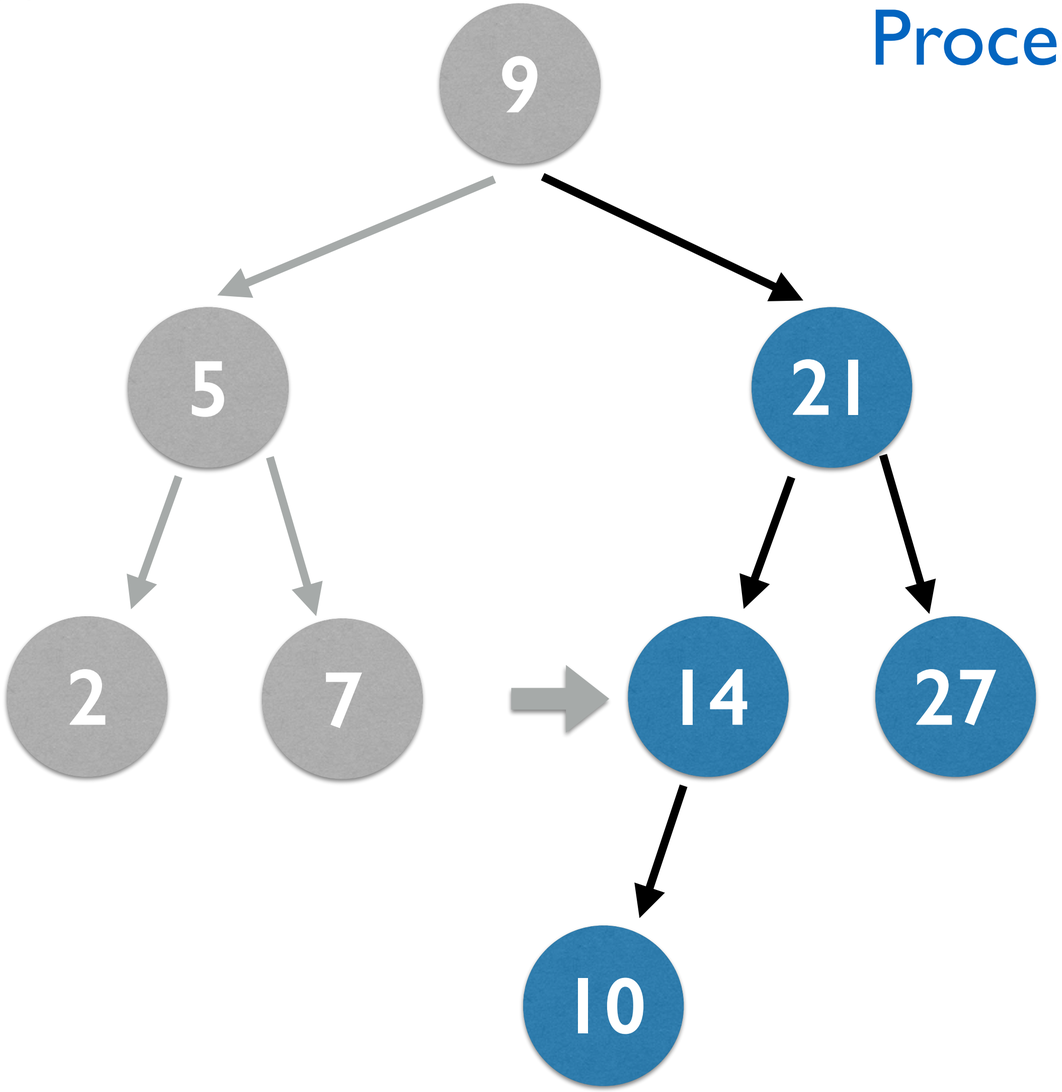
DFS: IN



Process left · Process root · Process right

2, 5, 7, 9

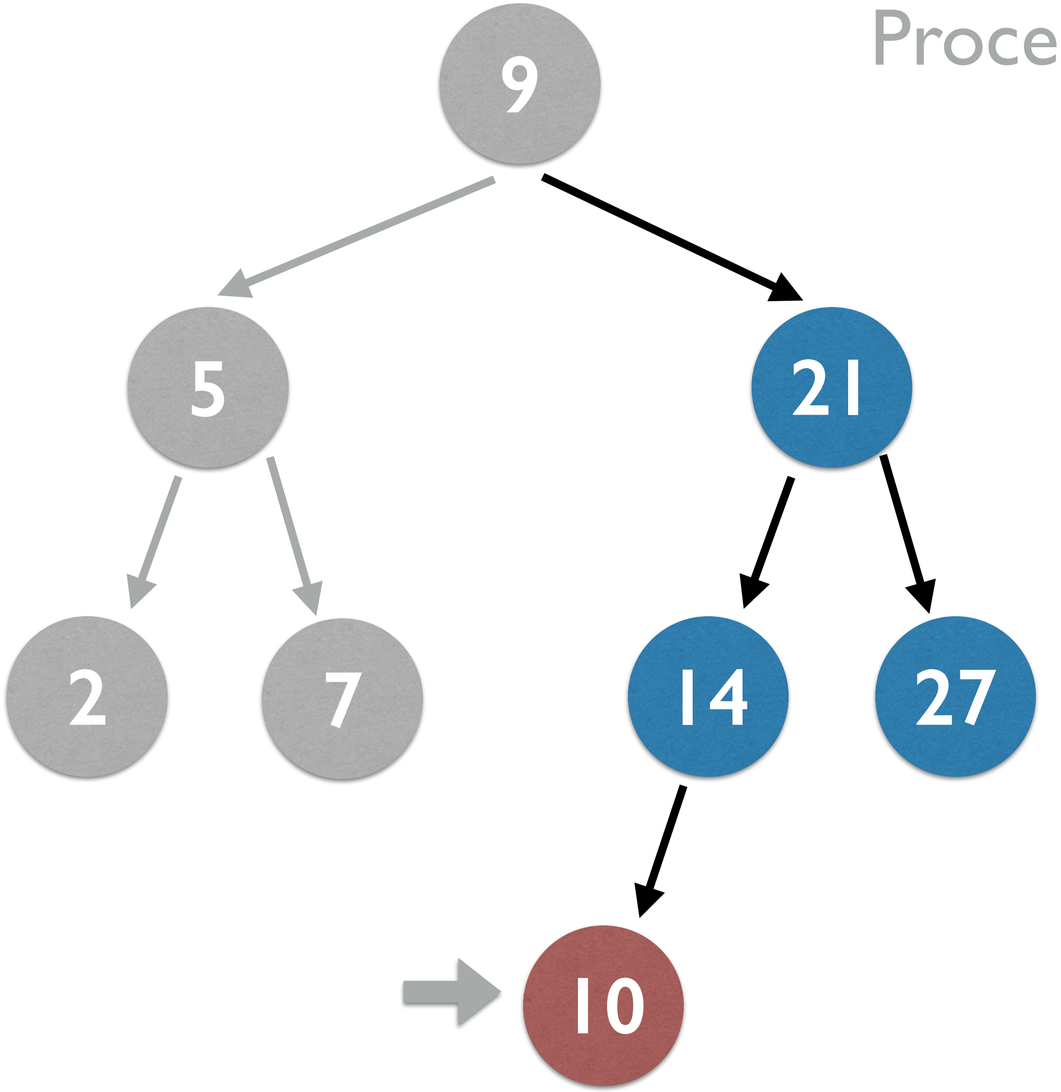
DFS: IN



Process left · Process root · Process right

2, 5, 7, 9

DFS: IN

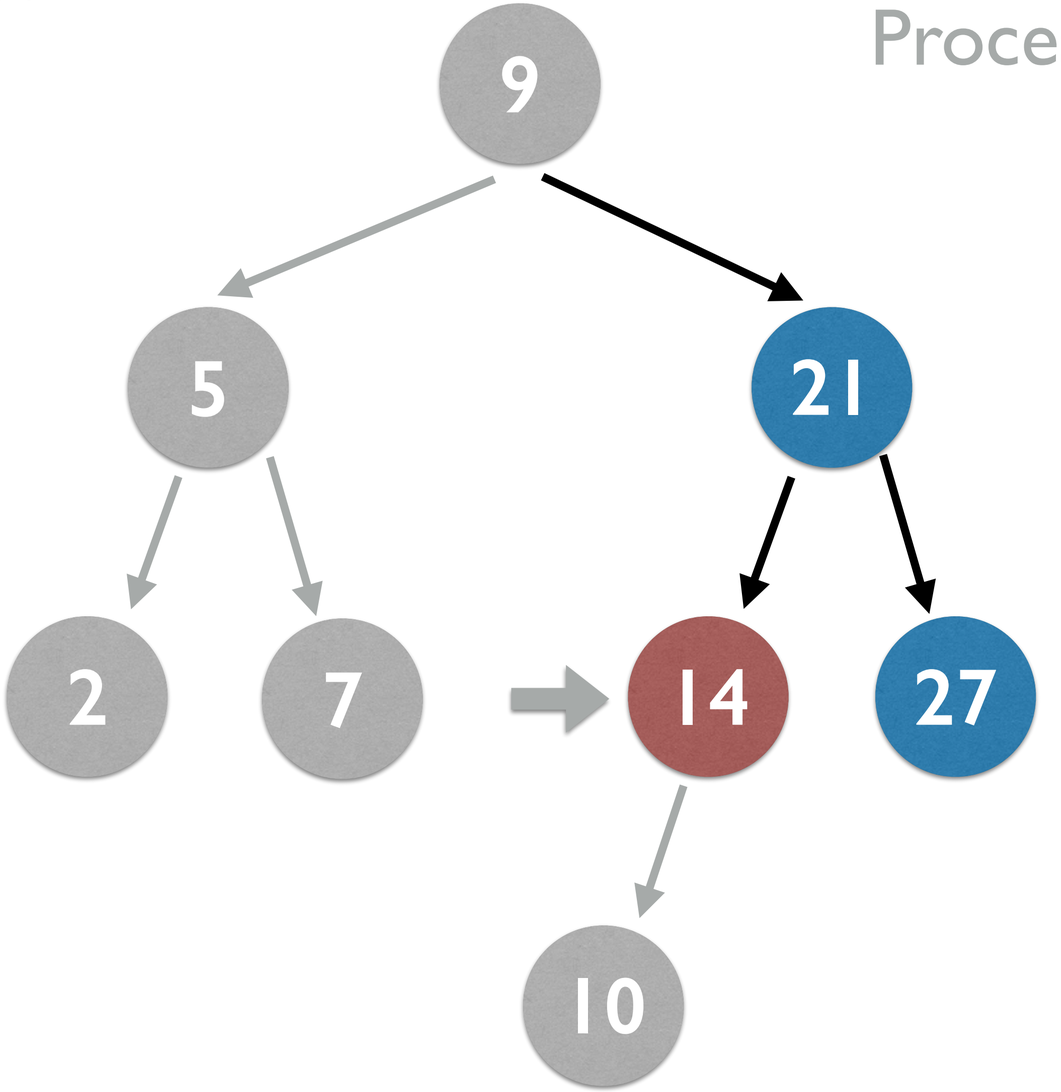


Process left · **Process root** · Process right

2, 5, 7, 9, 10

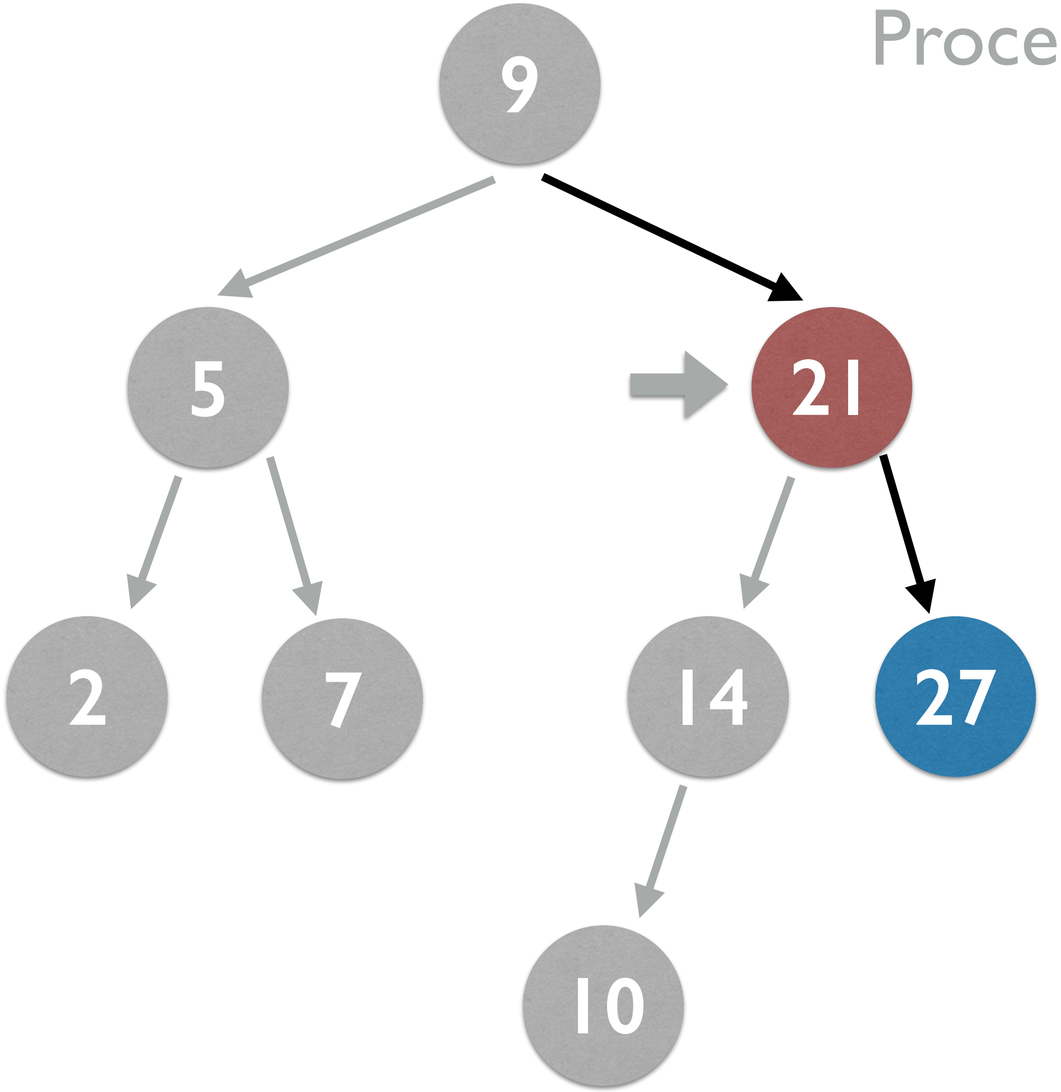
DFS: IN

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



2, 5, 7, 9, 10, 14

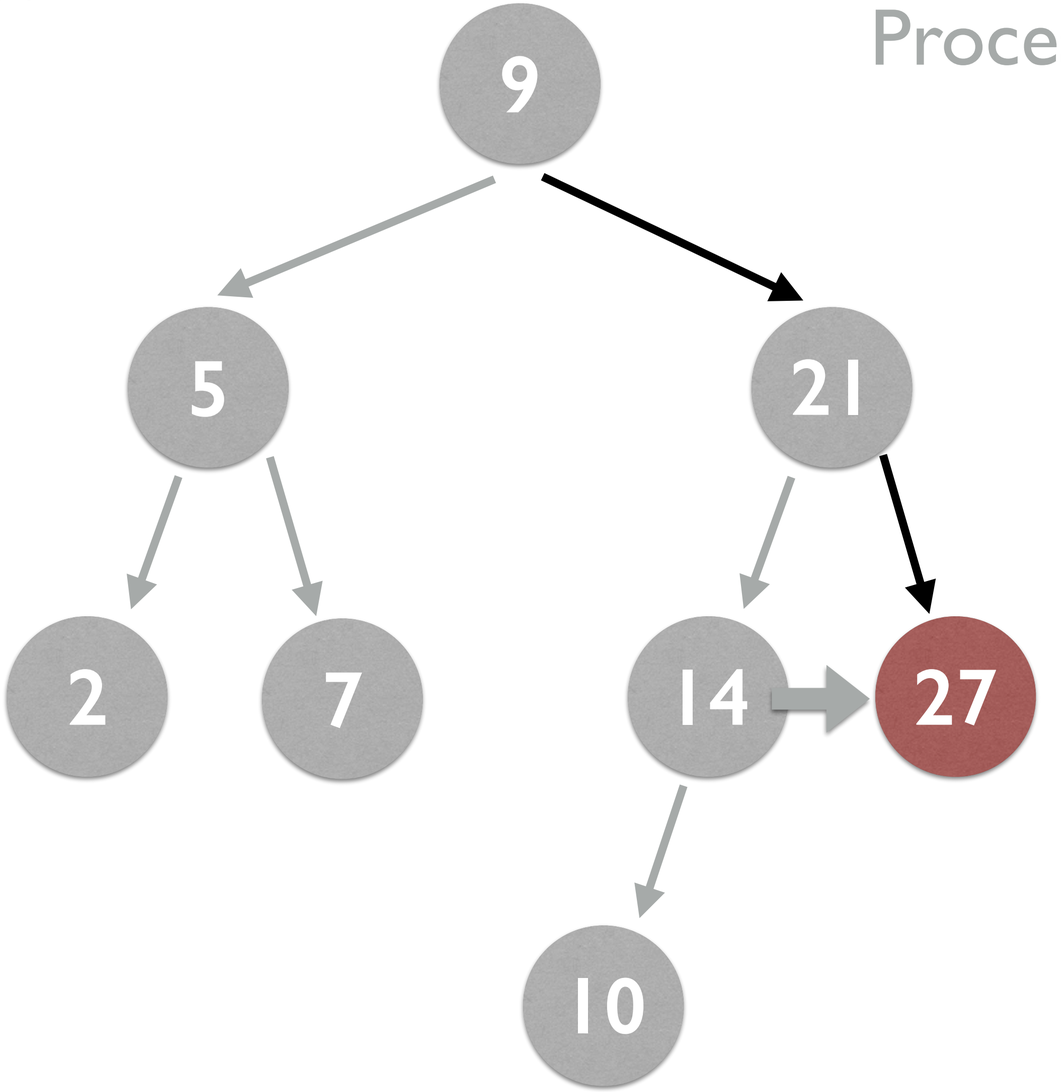
DFS: IN



Process left · Process root · Process right

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

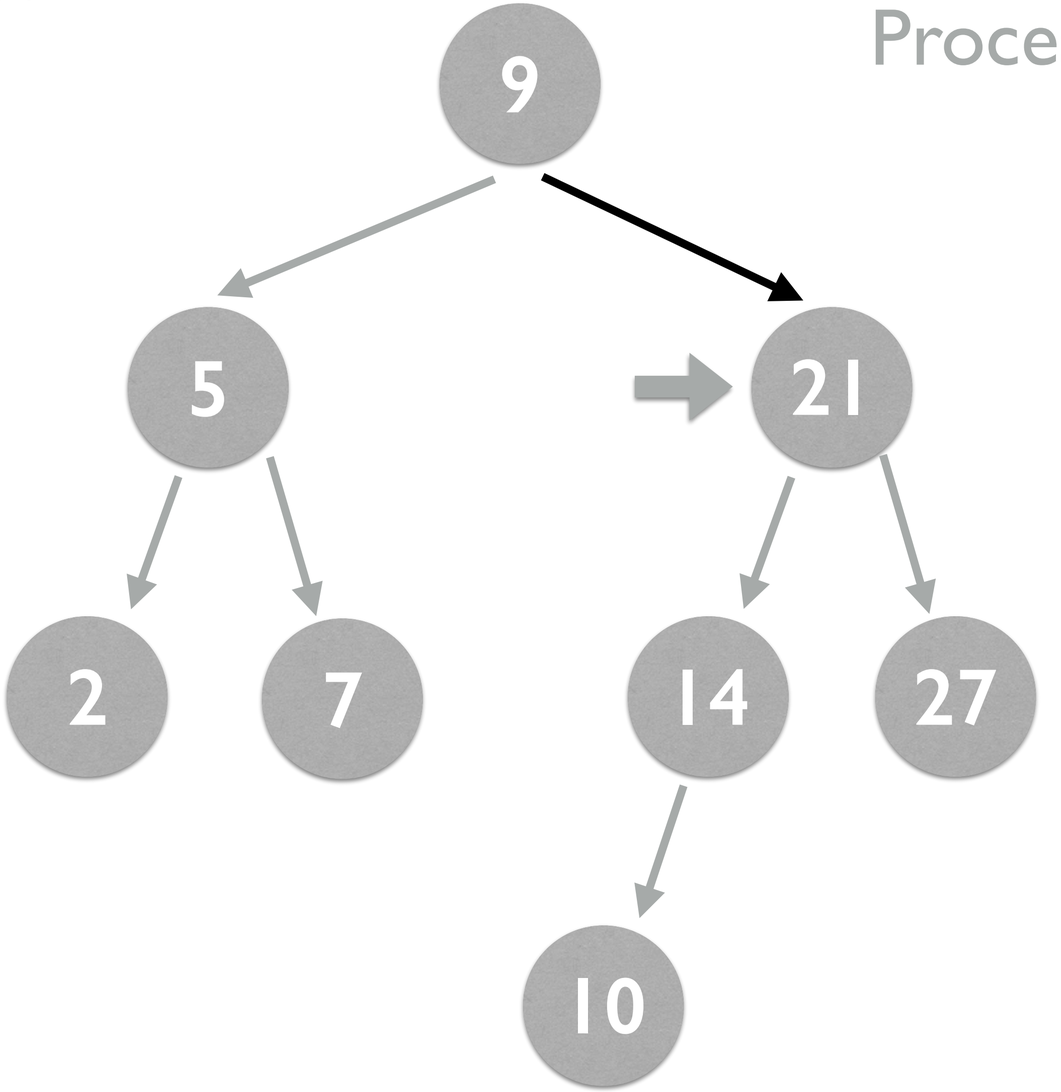
DFS: IN



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

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

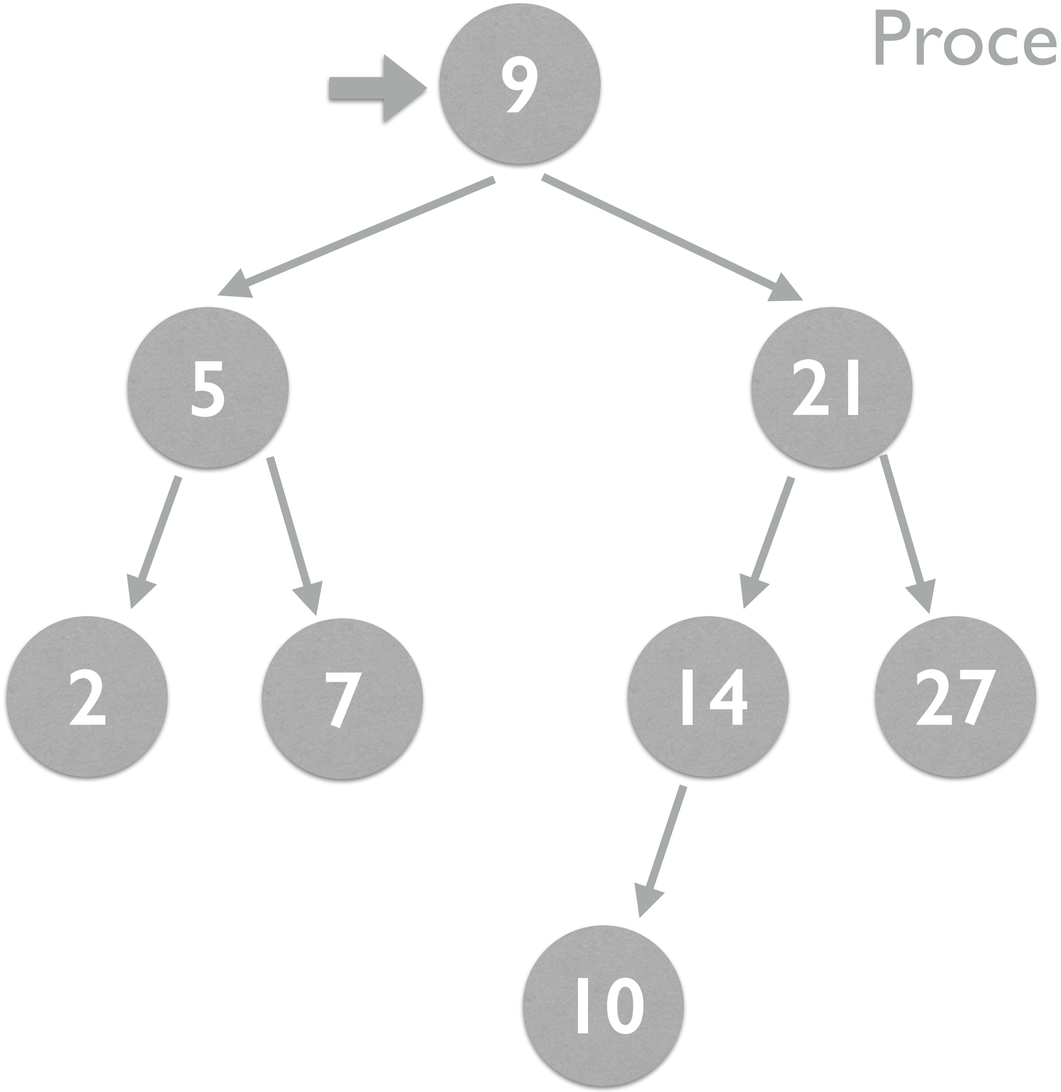
DFS: IN



Process left · Process root · Process right

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

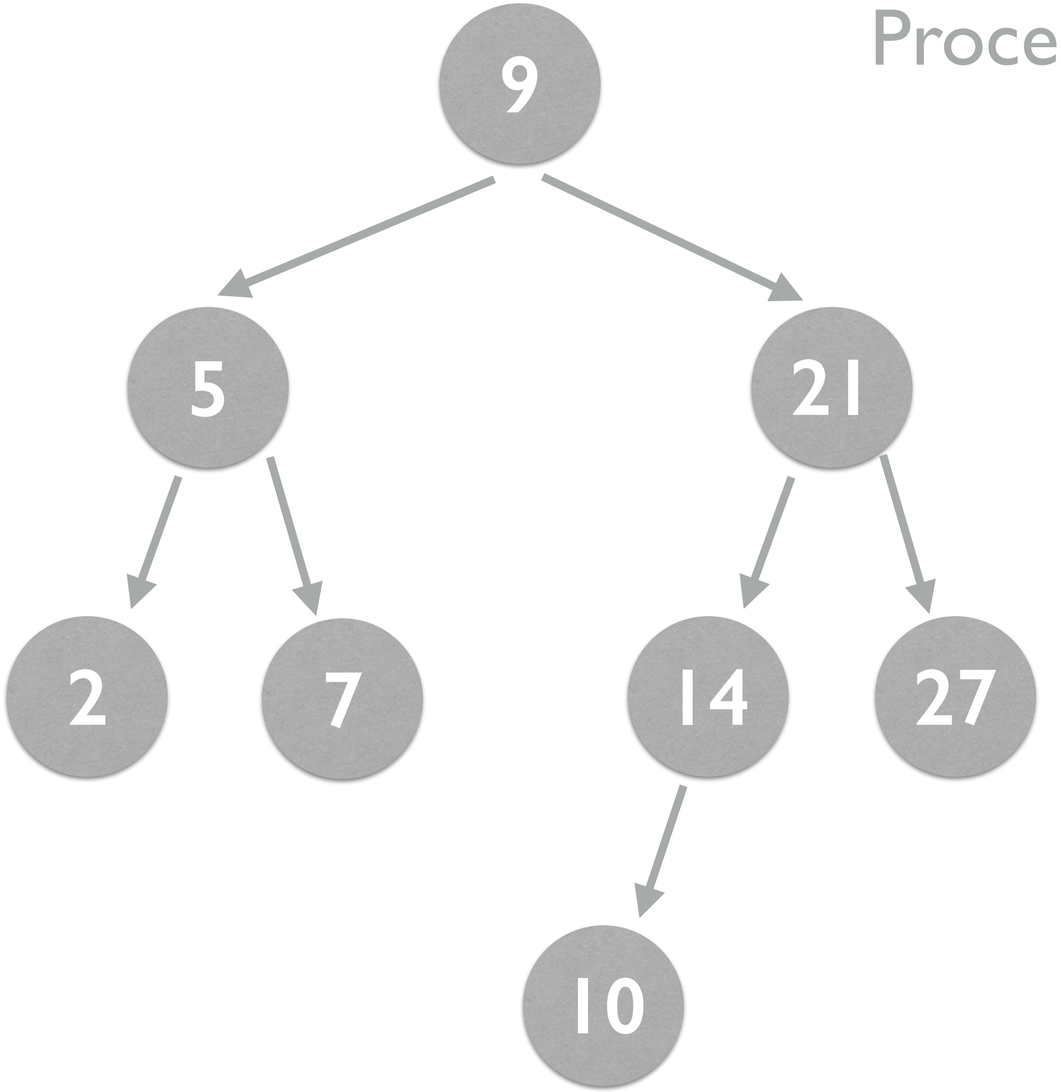
DFS: IN



Process left · Process root · Process right

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

DFS: IN



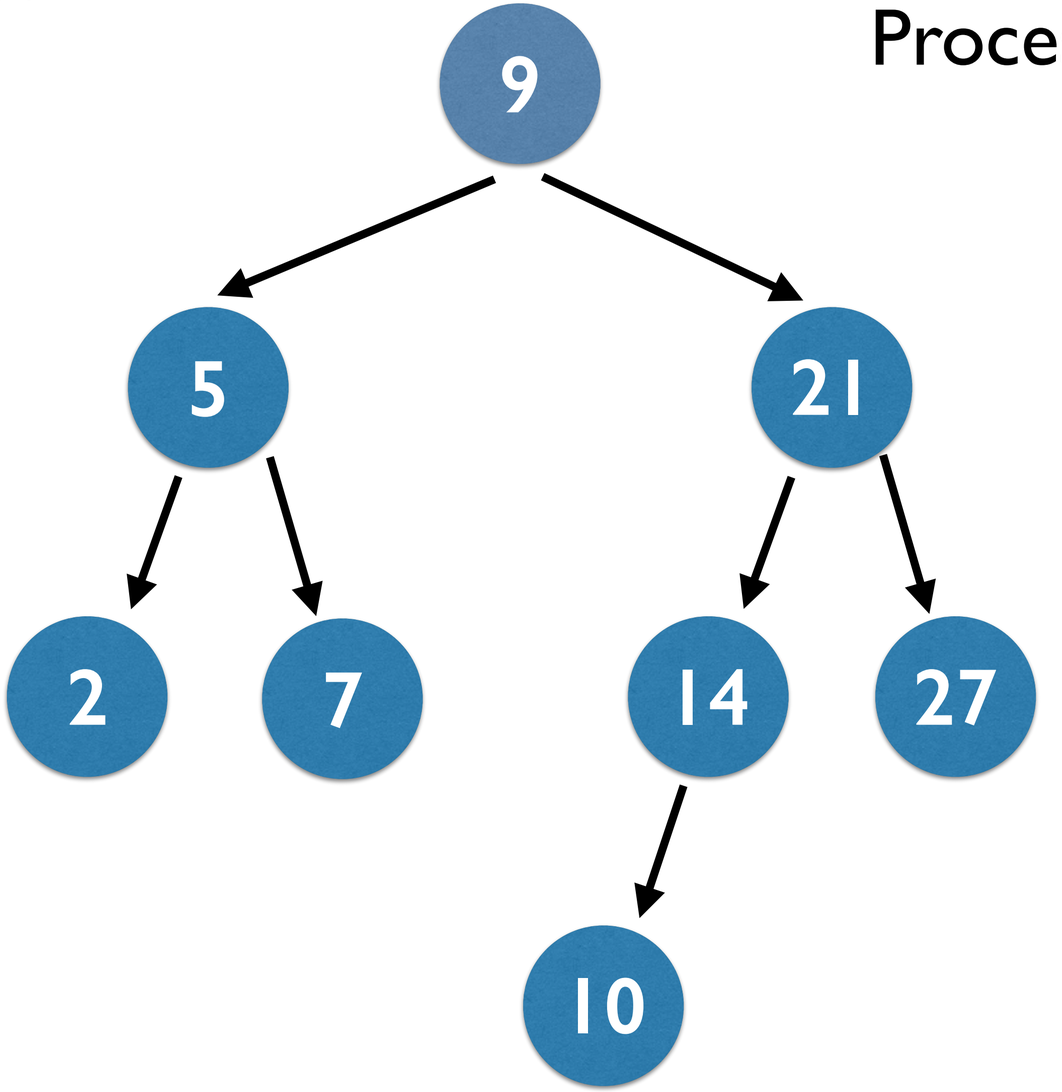
Process left · Process root · Process right

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.

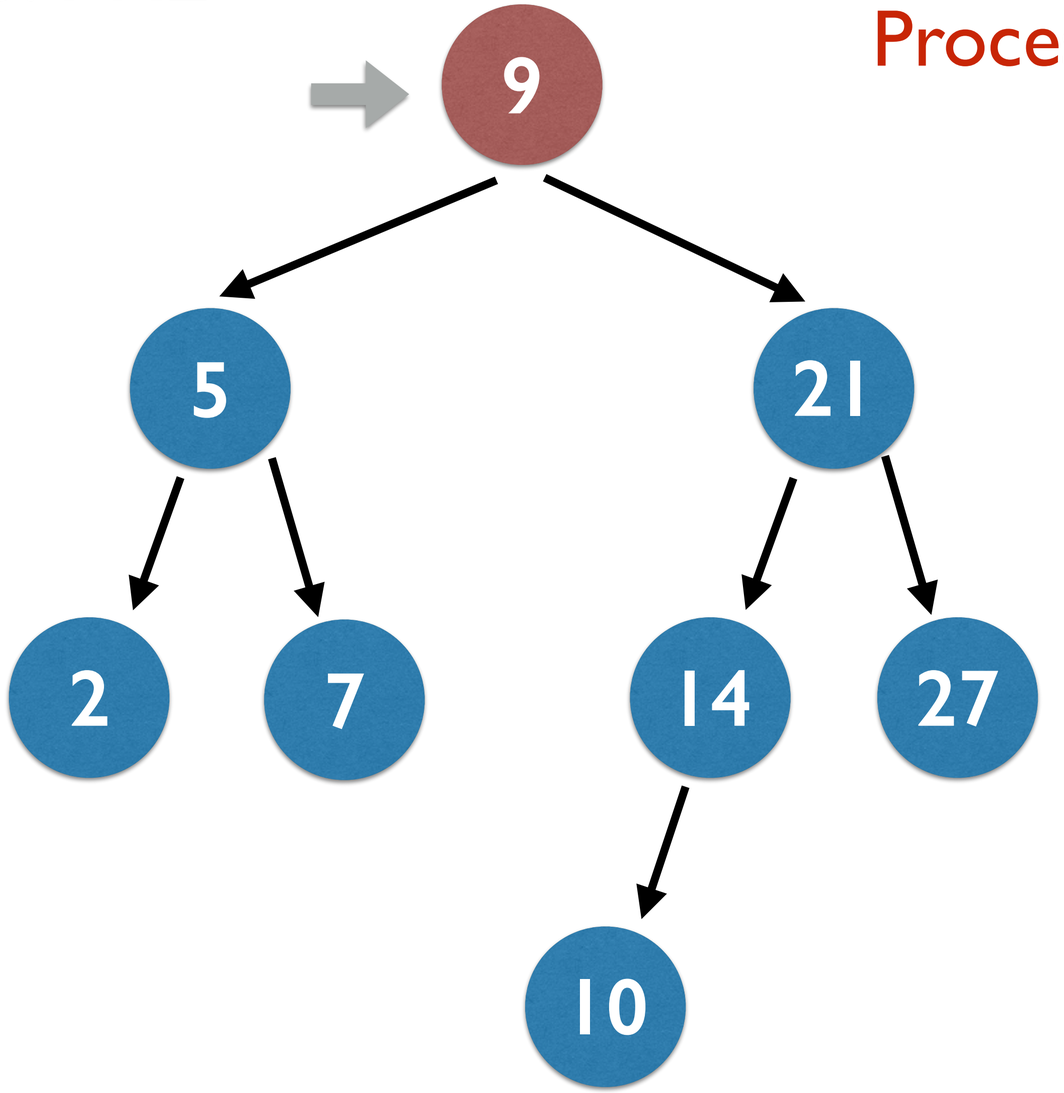
Depth First: Pre-Order

DFS: PRE



Process root · Process left · Process right

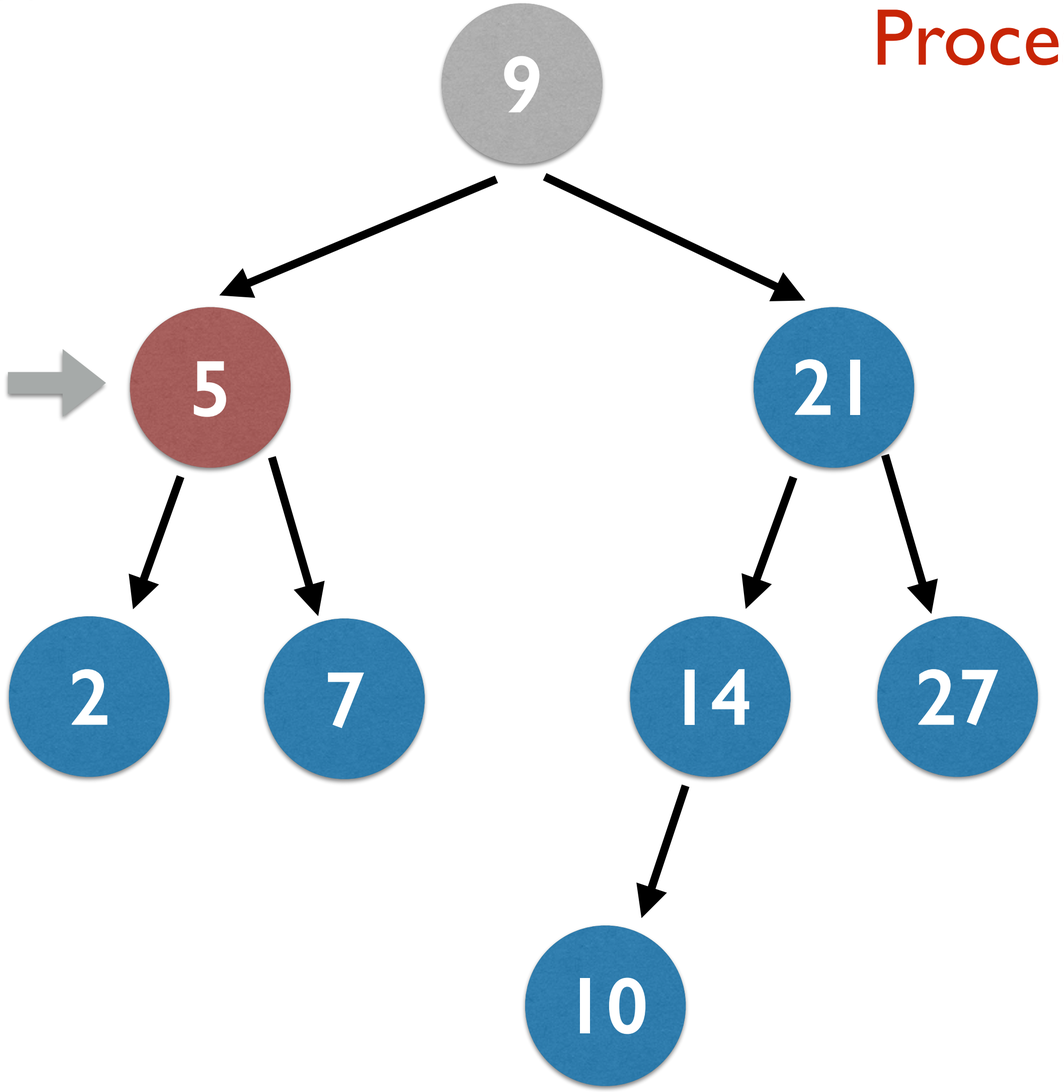
DFS: PRE



Process root · Process left · Process right

9

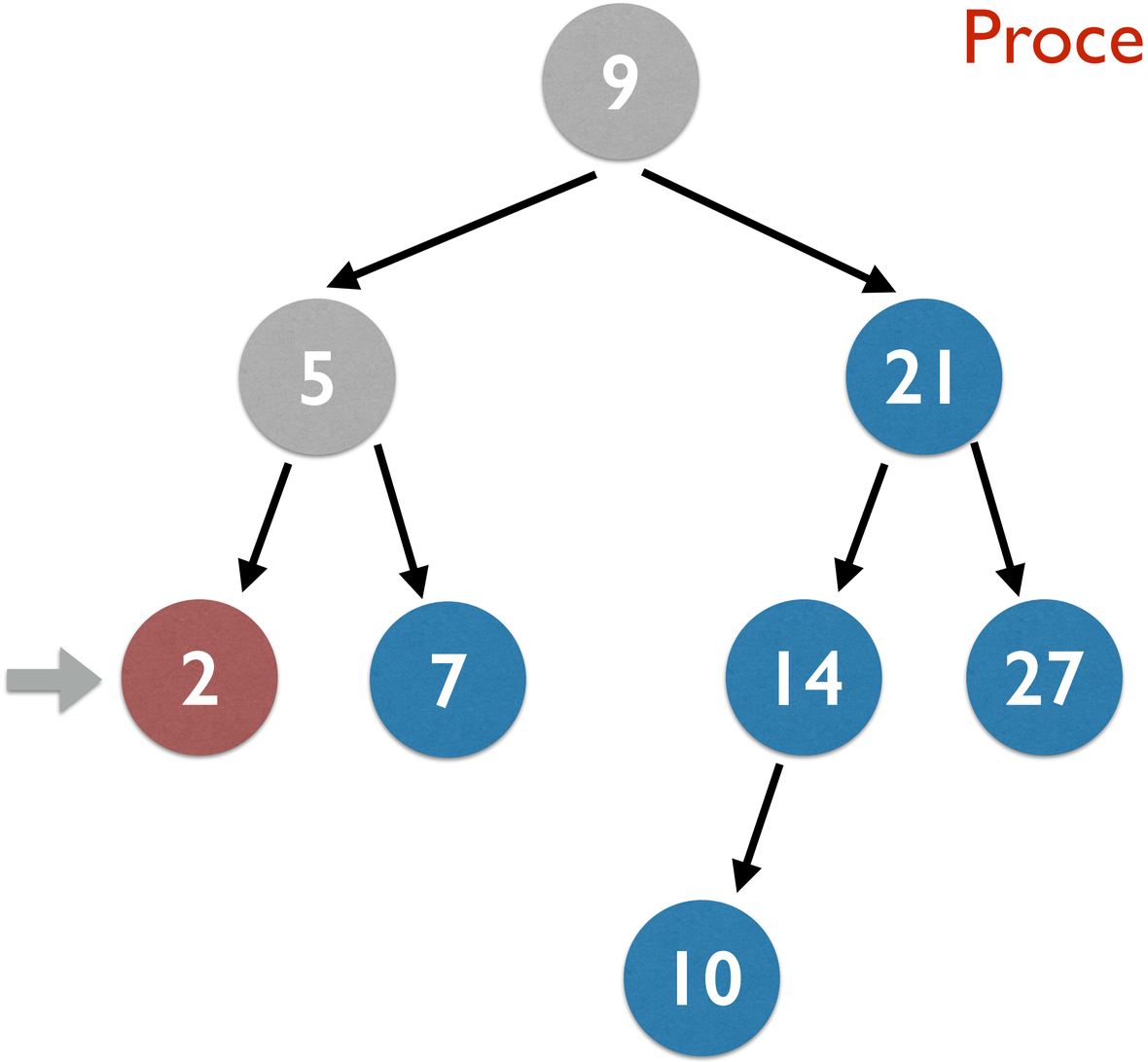
DFS: PRE



Process root · Process left · Process right

9, 5

DFS: PRE

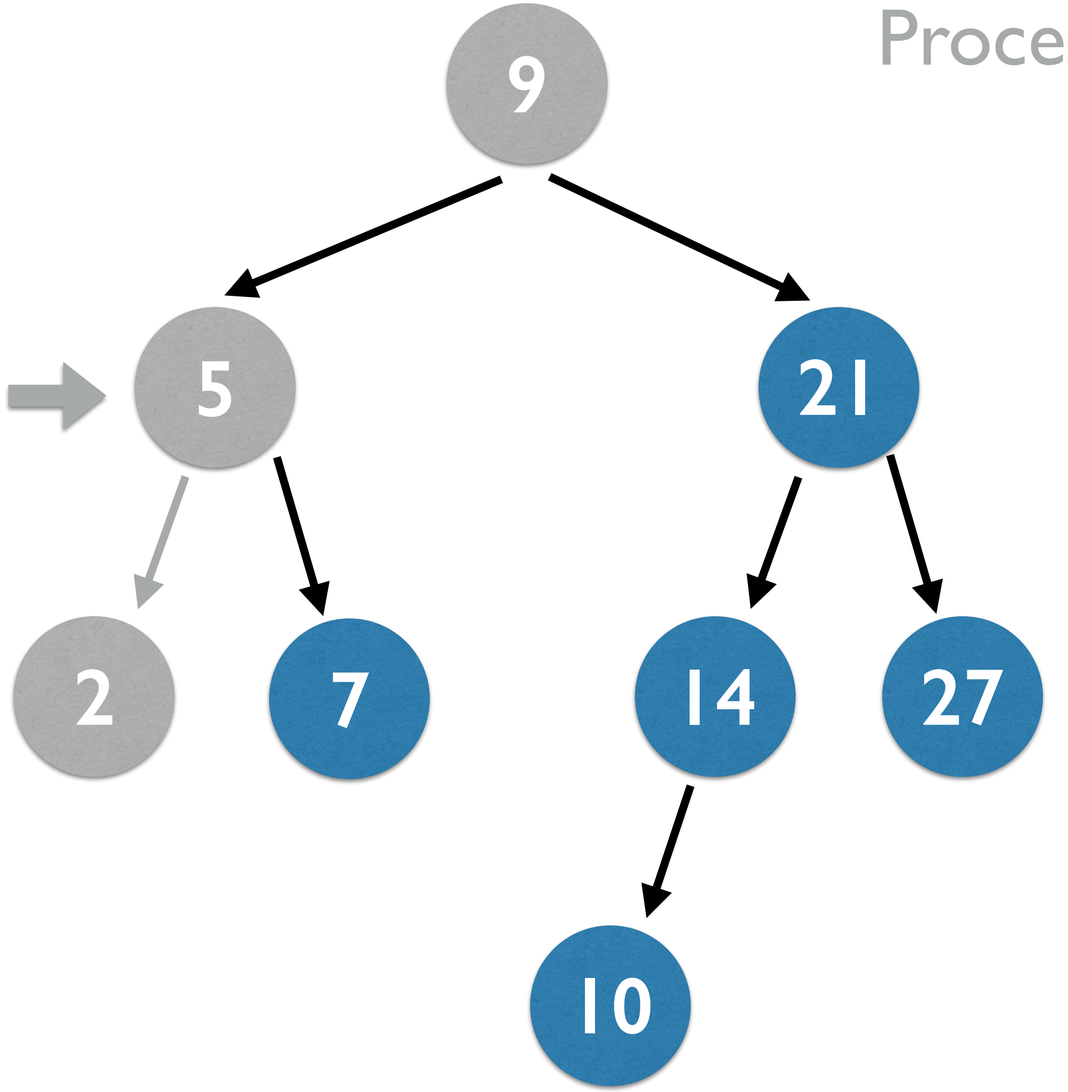


Process root · Process left · Process right

9, 5, 2

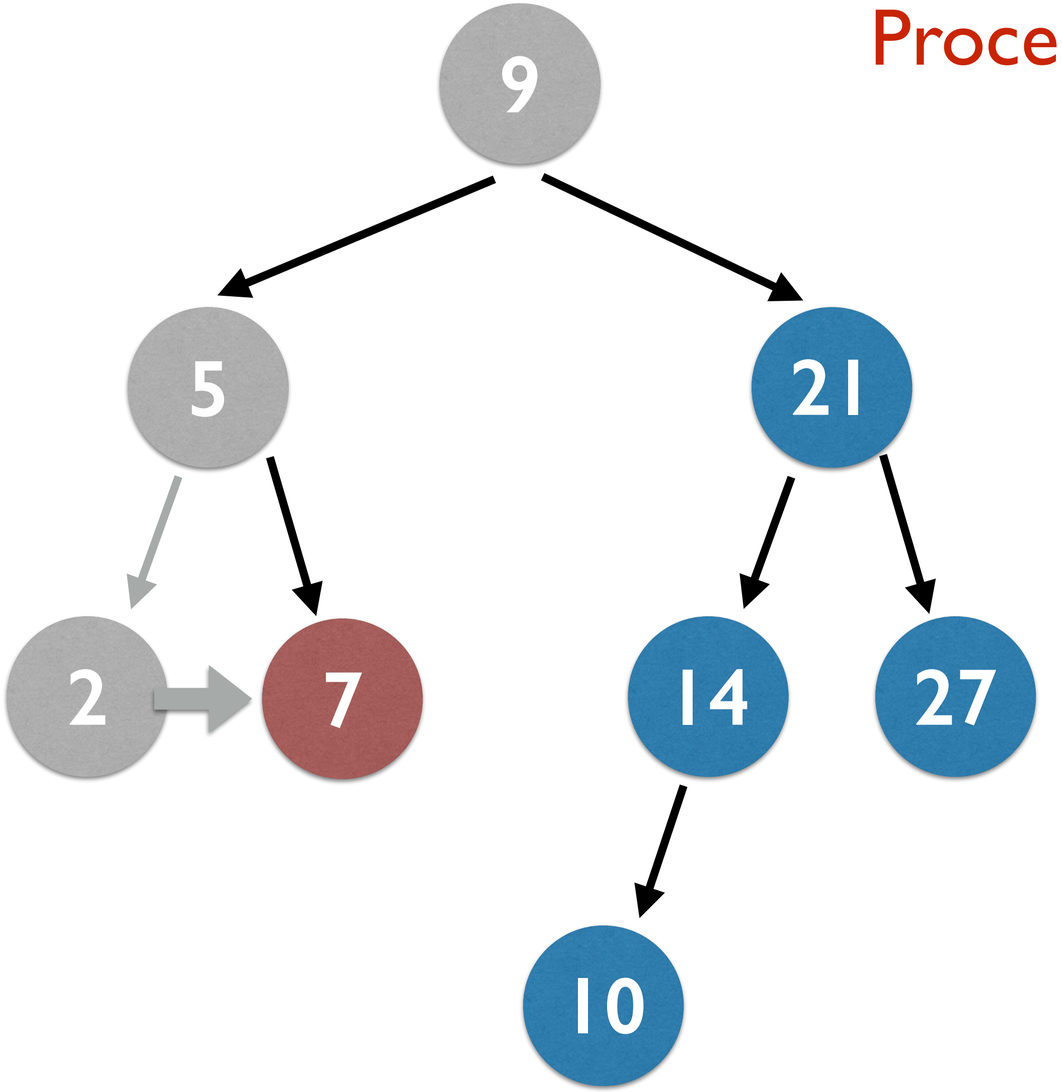
DFS: PRE

Process root · Process left · Process right



9, 5, 2

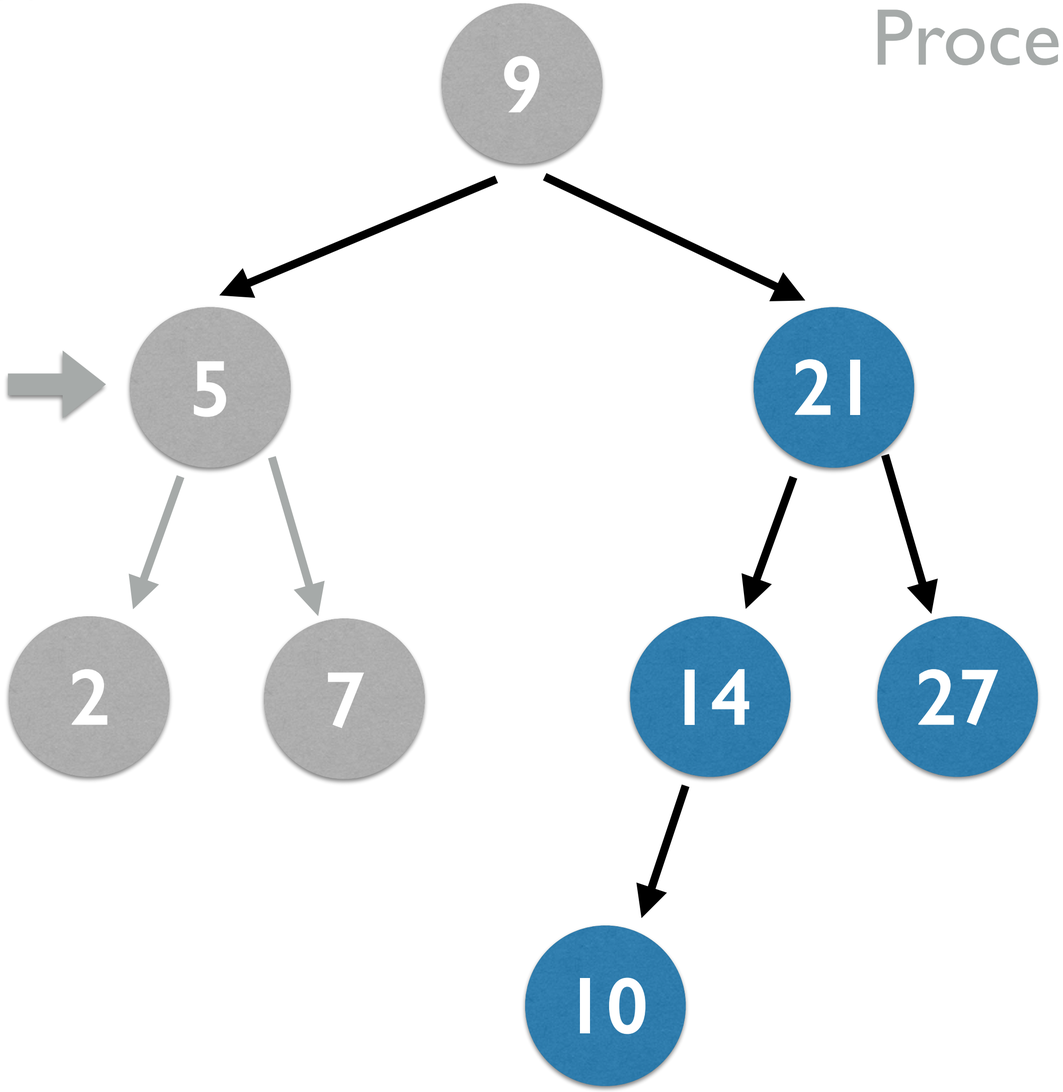
DFS: PRE



Process root · Process left · Process right

9, 5, 2, 7

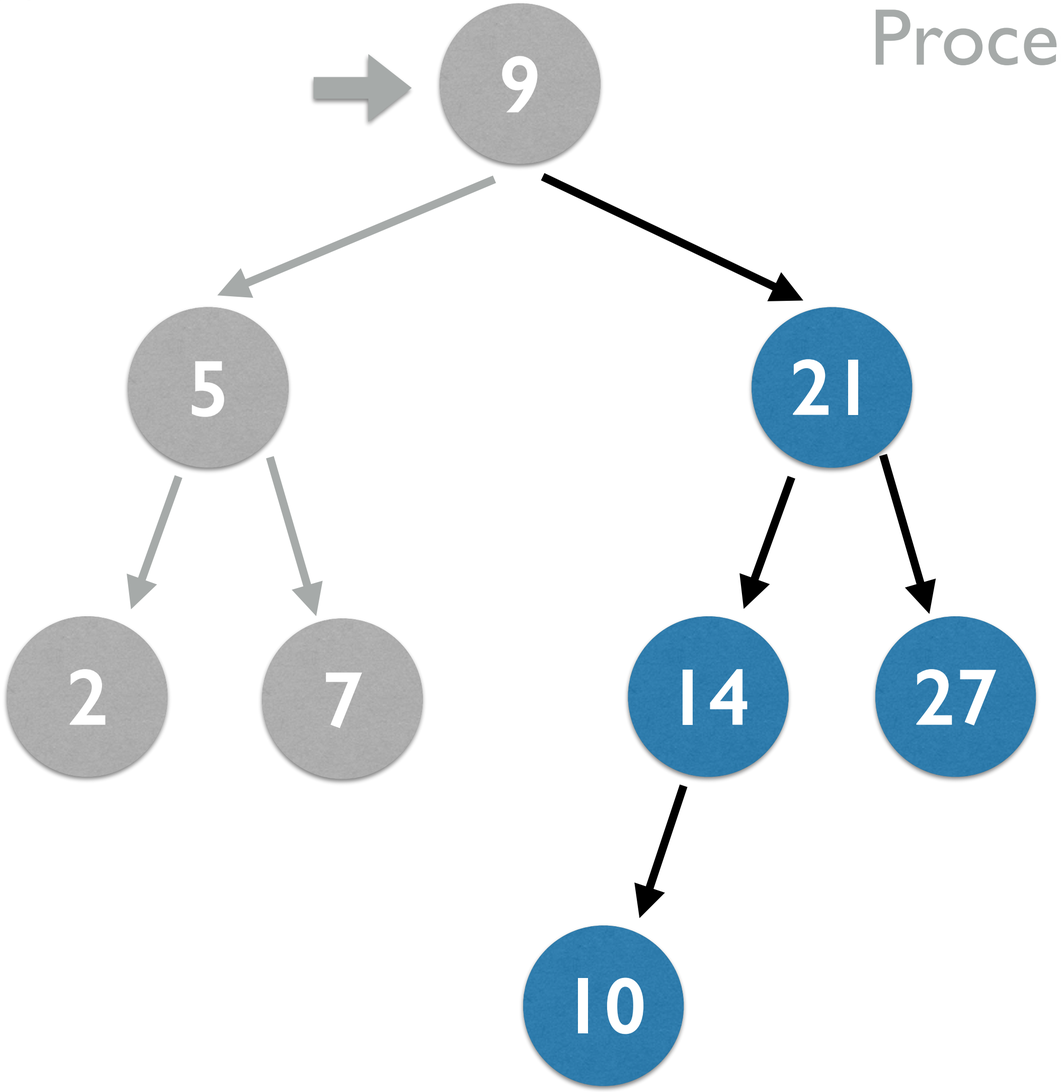
DFS: PRE



Process root · Process left · Process right

9, 5, 2, 7

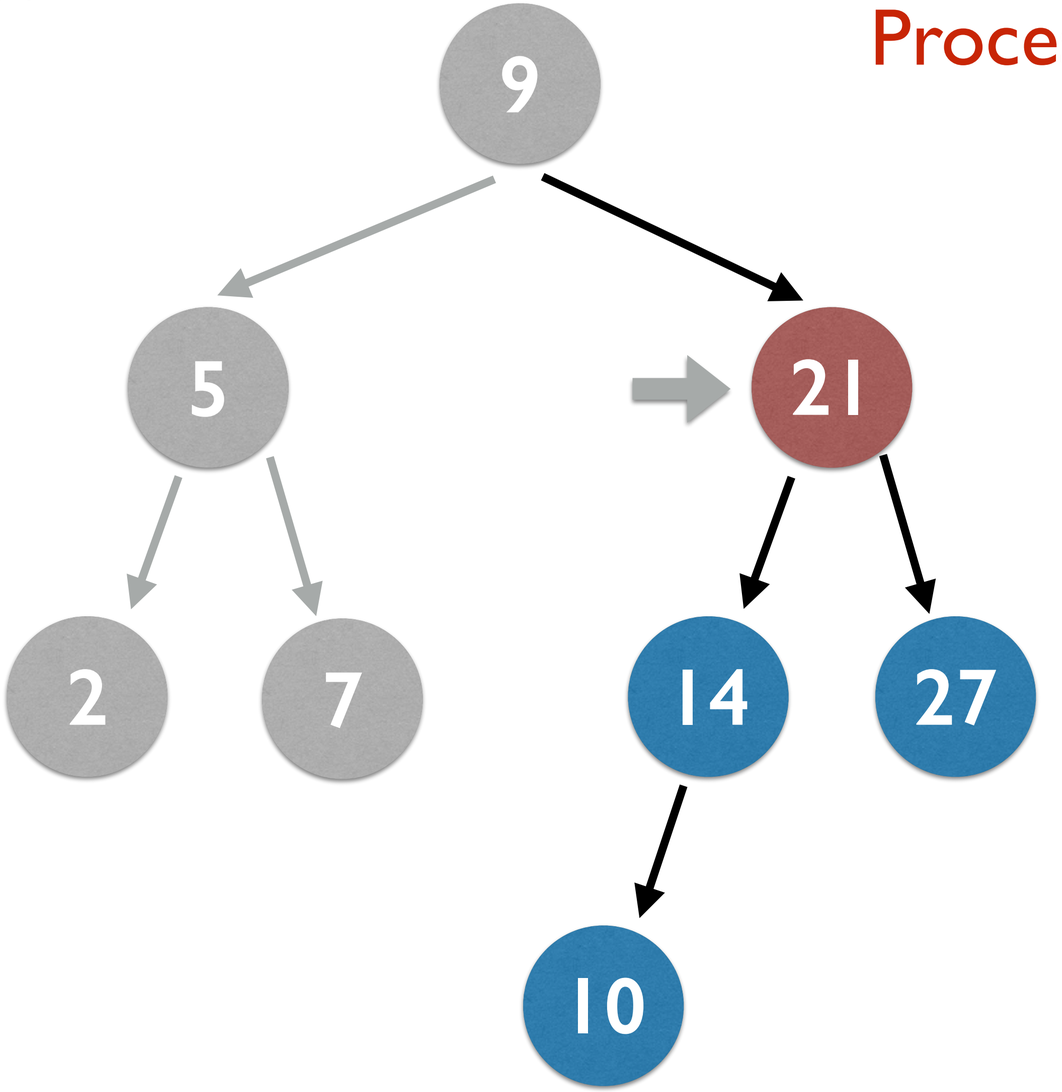
DFS: PRE



Process root · Process left · Process right

9, 5, 2, 7

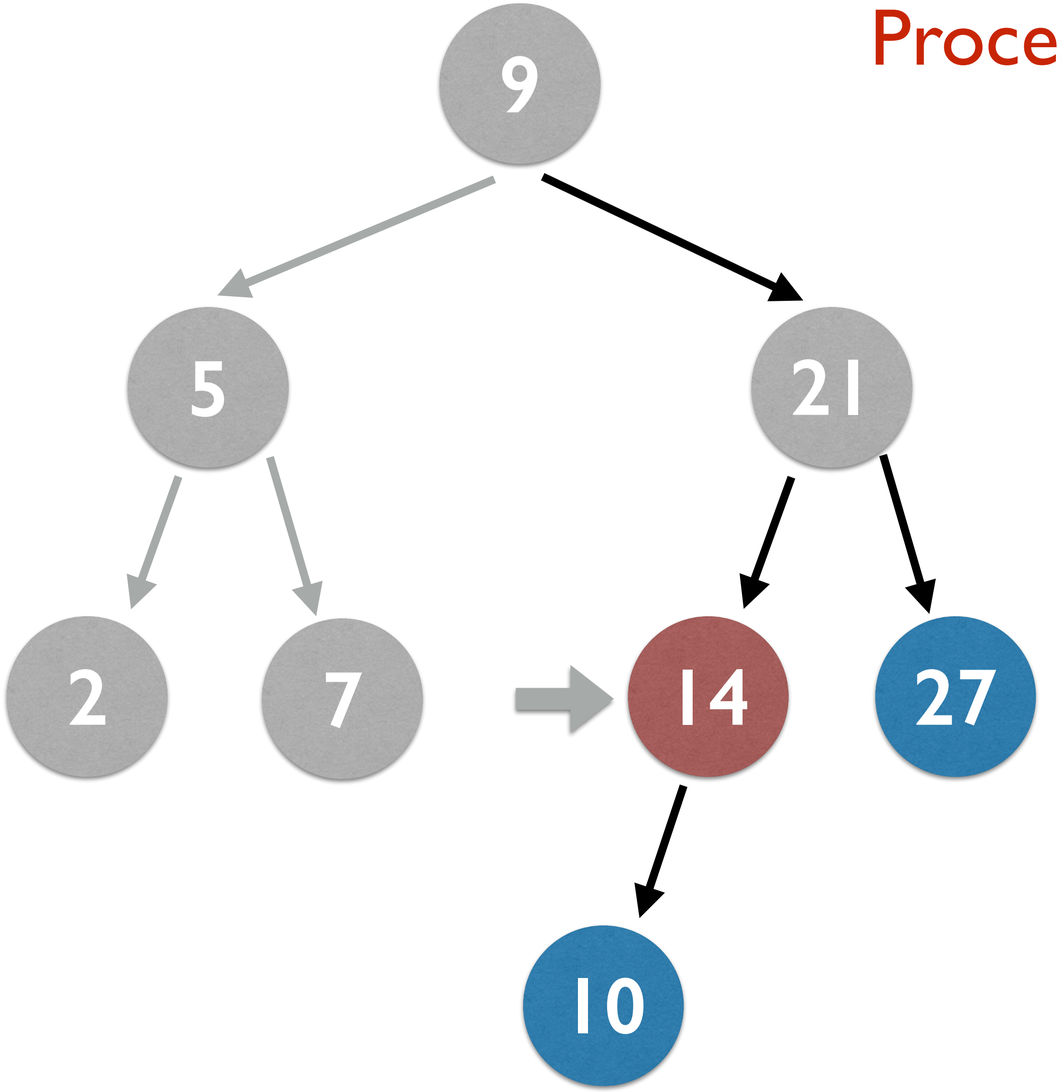
DFS: PRE



Process root · Process left · Process right

9, 5, 2, 7, 21

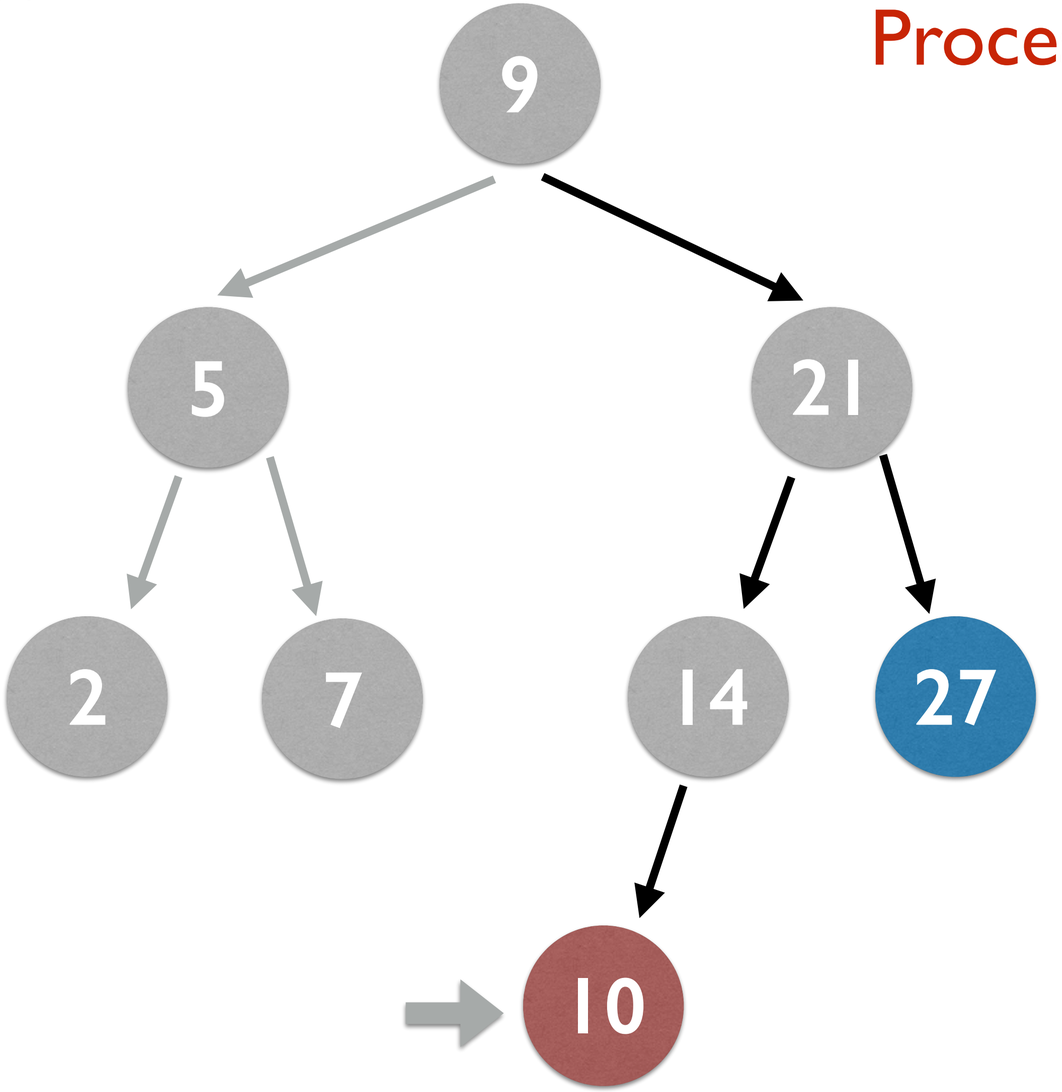
DFS: PRE



Process root · Process left · Process right

9, 5, 2, 7, 21, 14

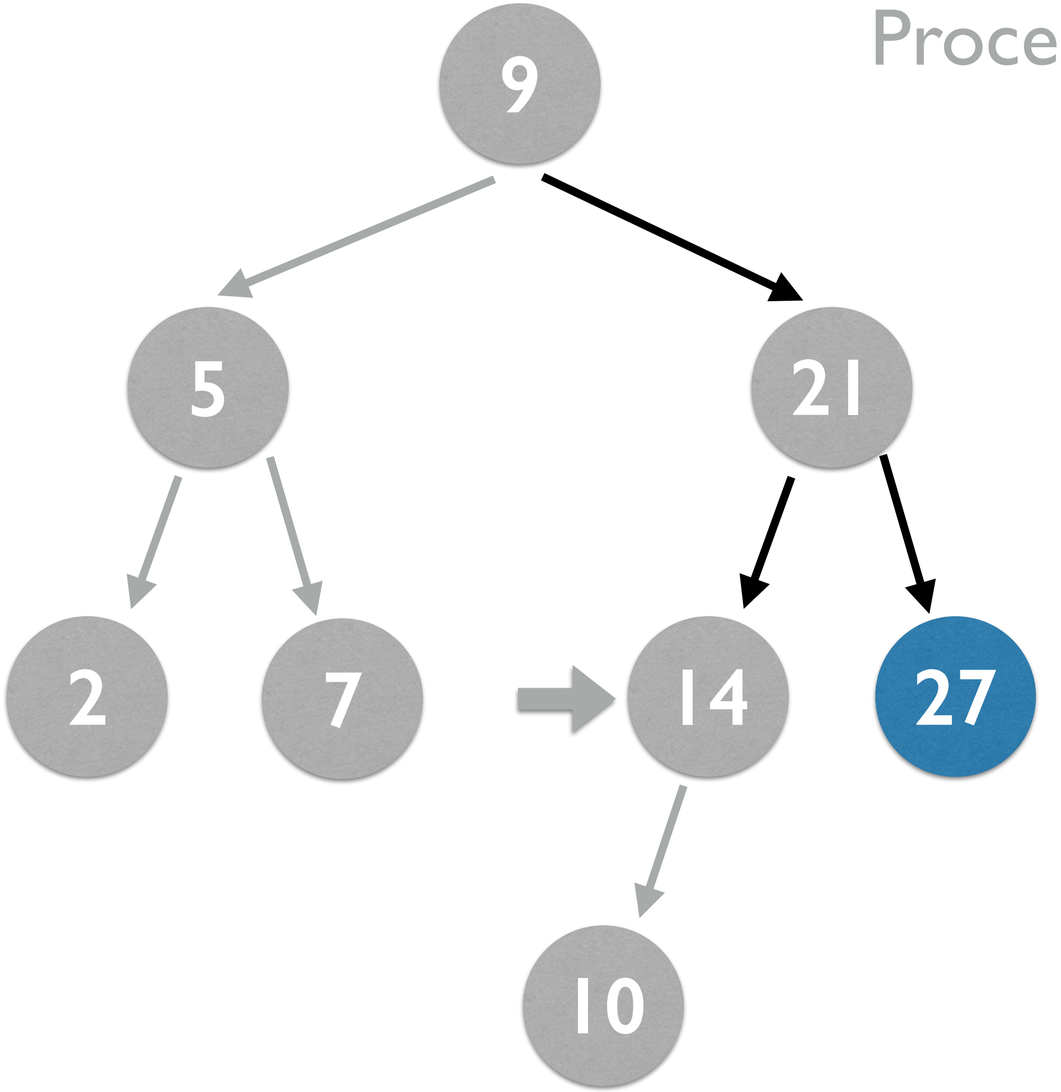
DFS: PRE



Process root · Process left · Process right

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

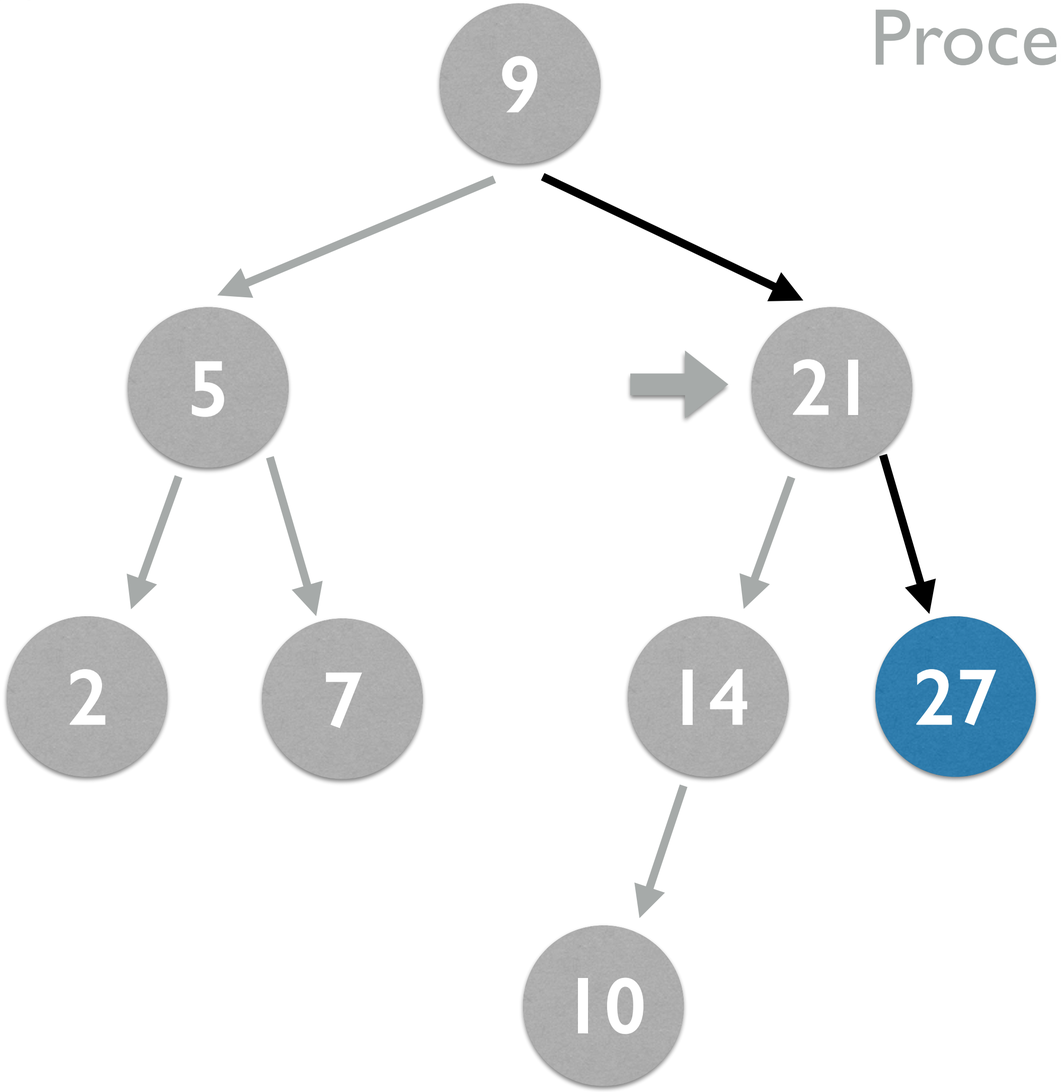
DFS: PRE



Process root · Process left · Process right

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

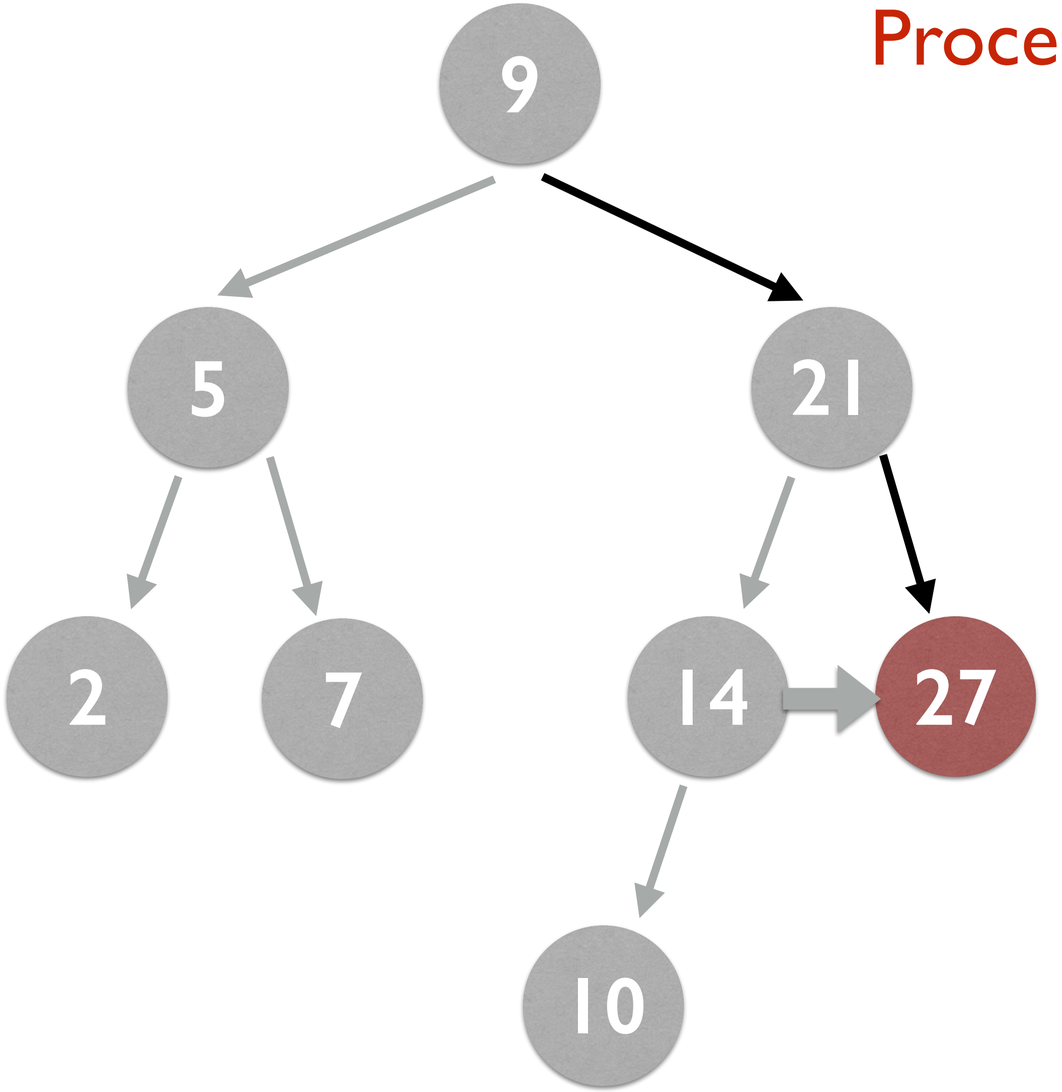
DFS: PRE



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

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

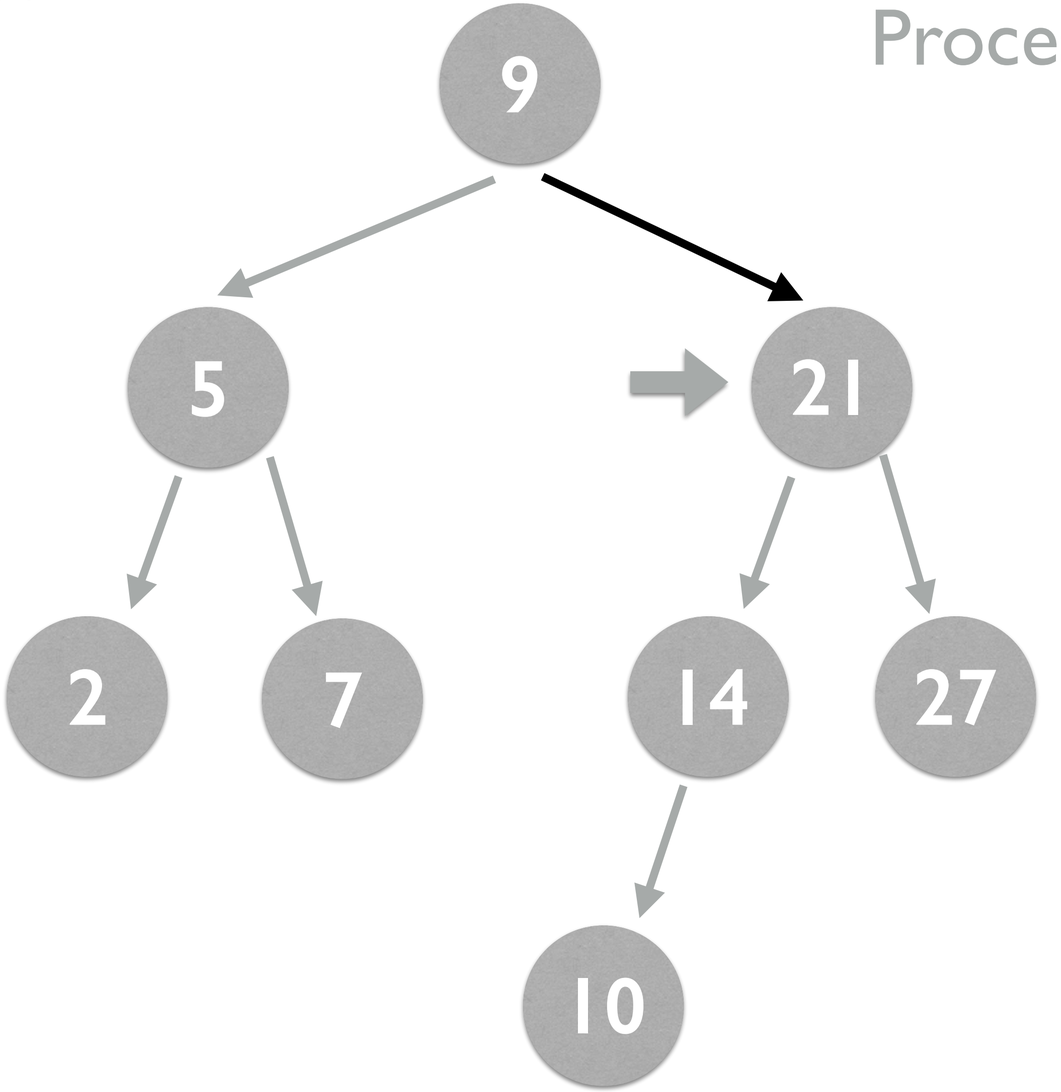
DFS: PRE



Process root · Process left · Process right

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

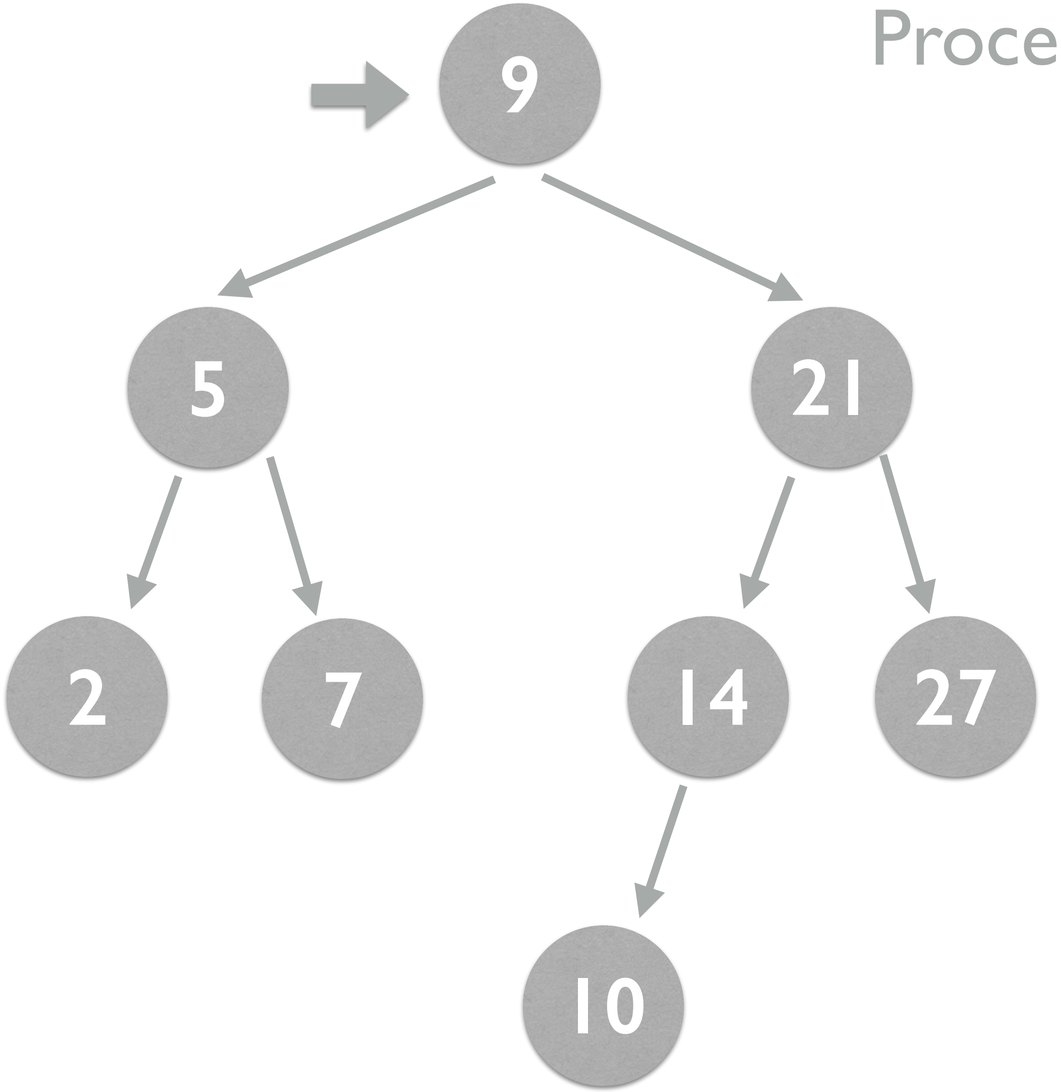
DFS: PRE



Process root · Process left · Process right

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

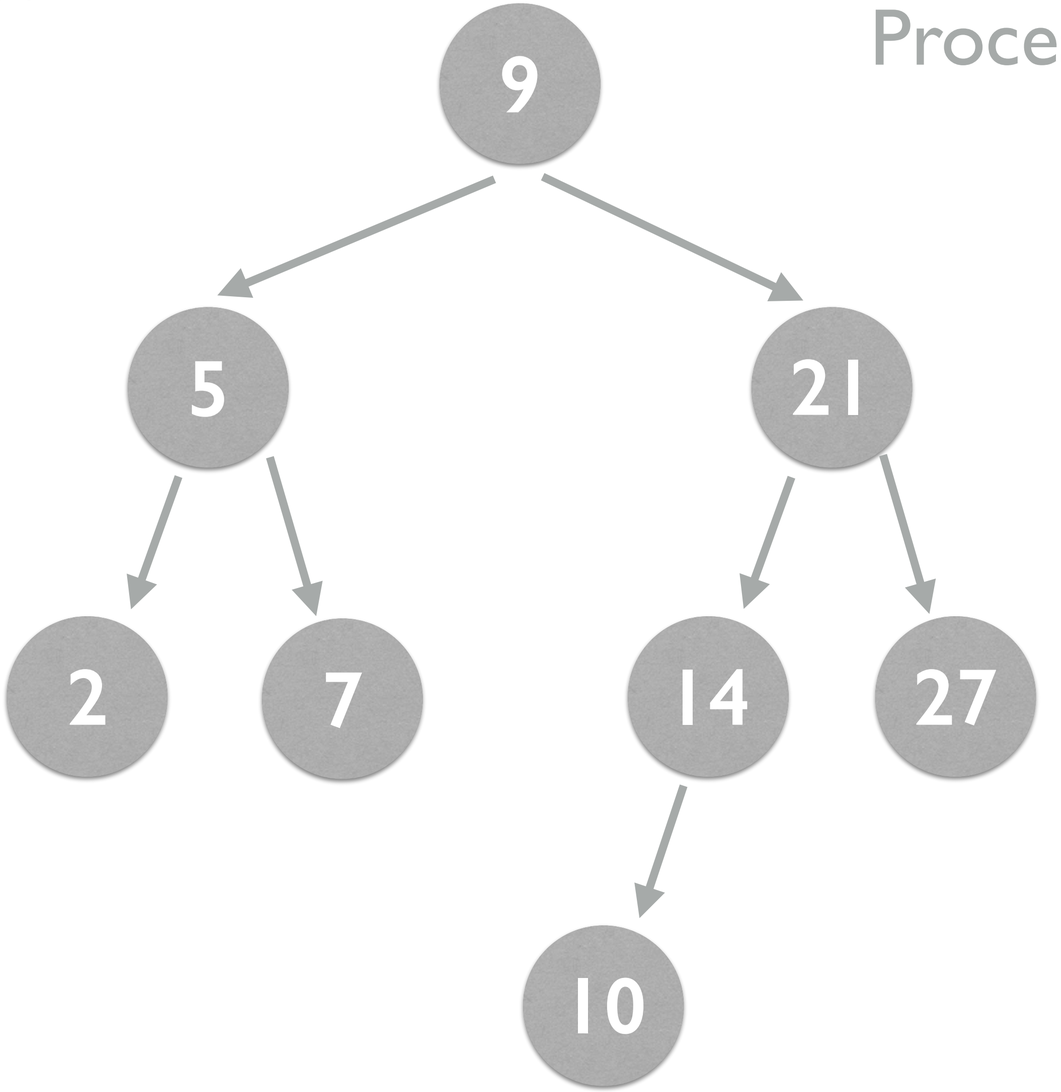
DFS: PRE



Process root · Process left · Process right

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

DFS: PRE



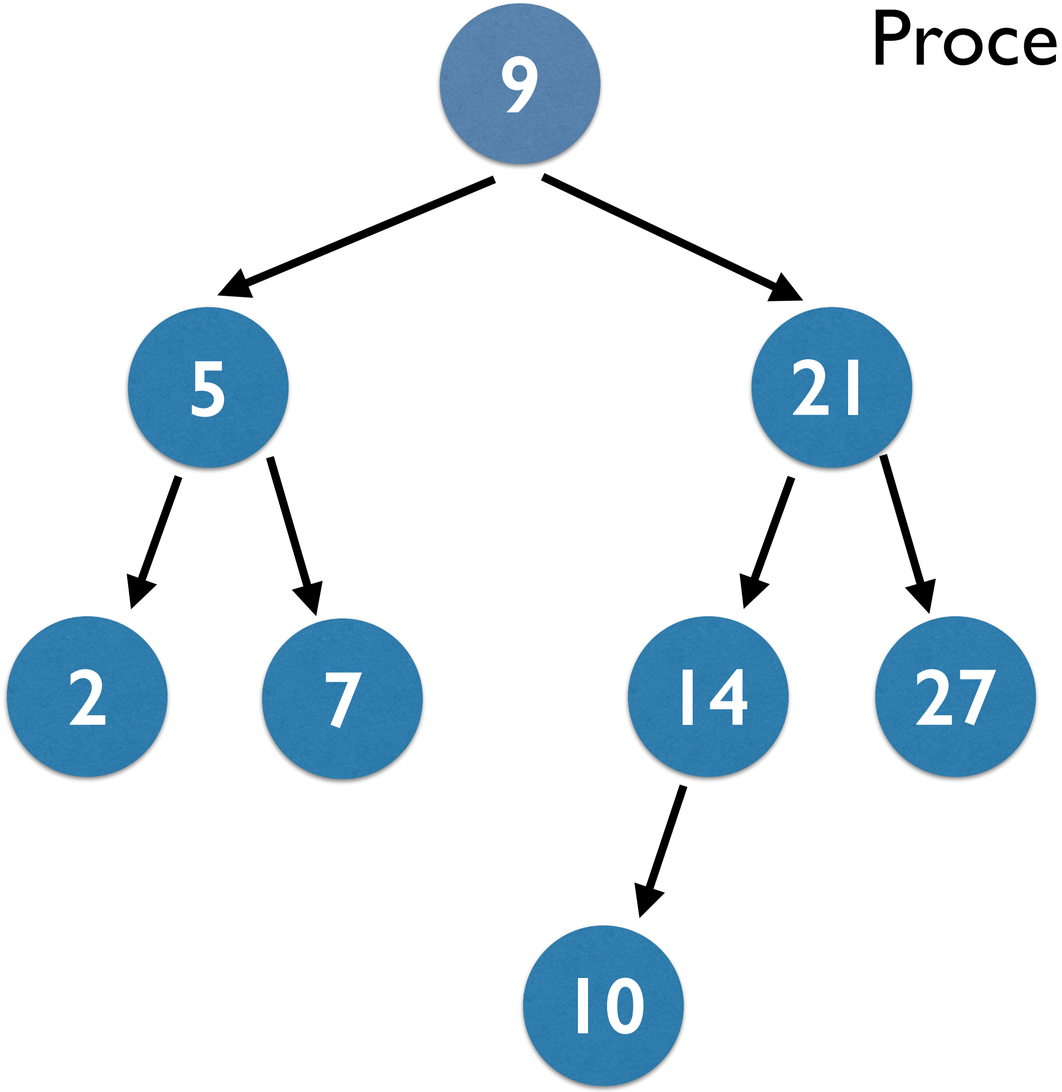
Process root · Process left · Process right

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.**

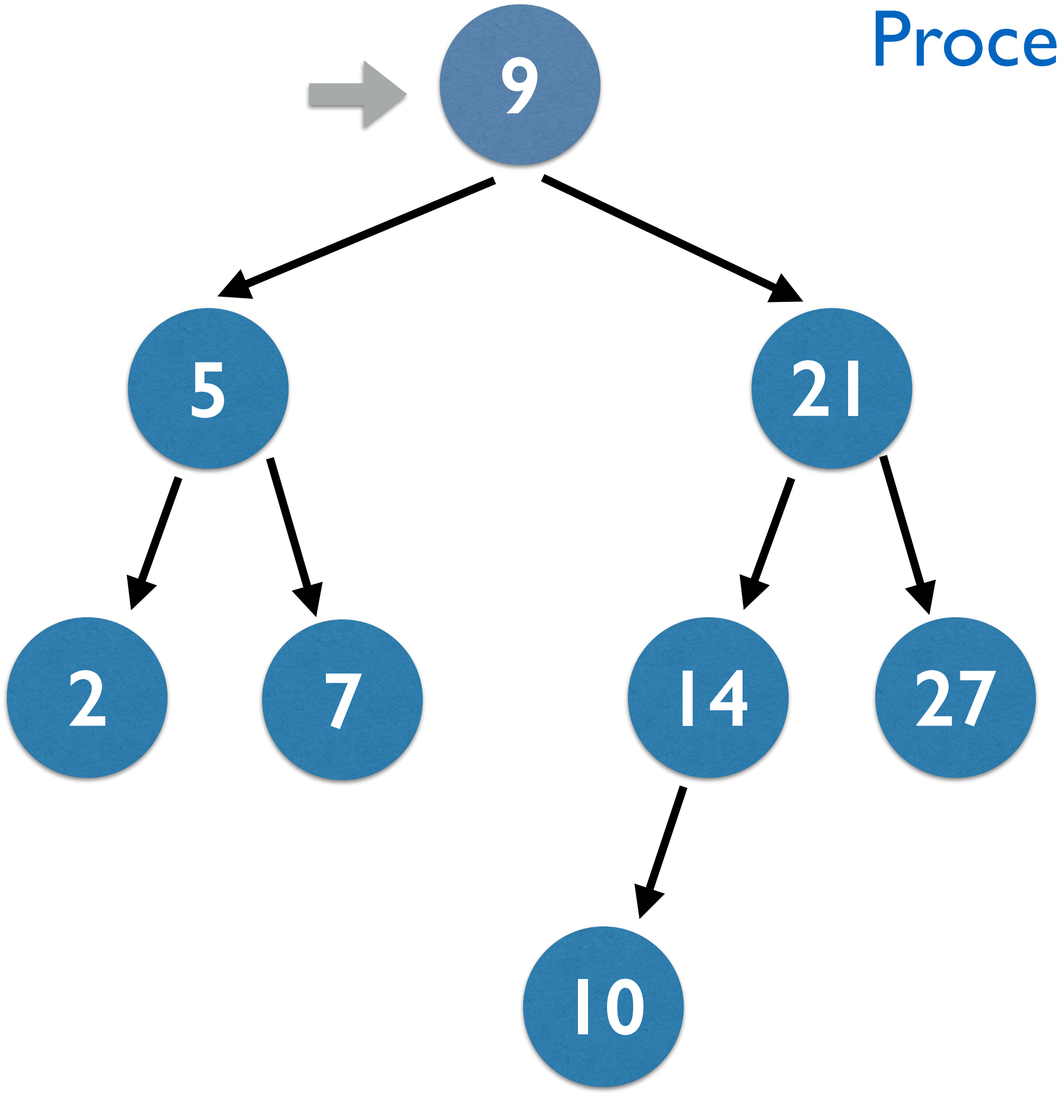
Depth First: Post-Order

DFS: POST



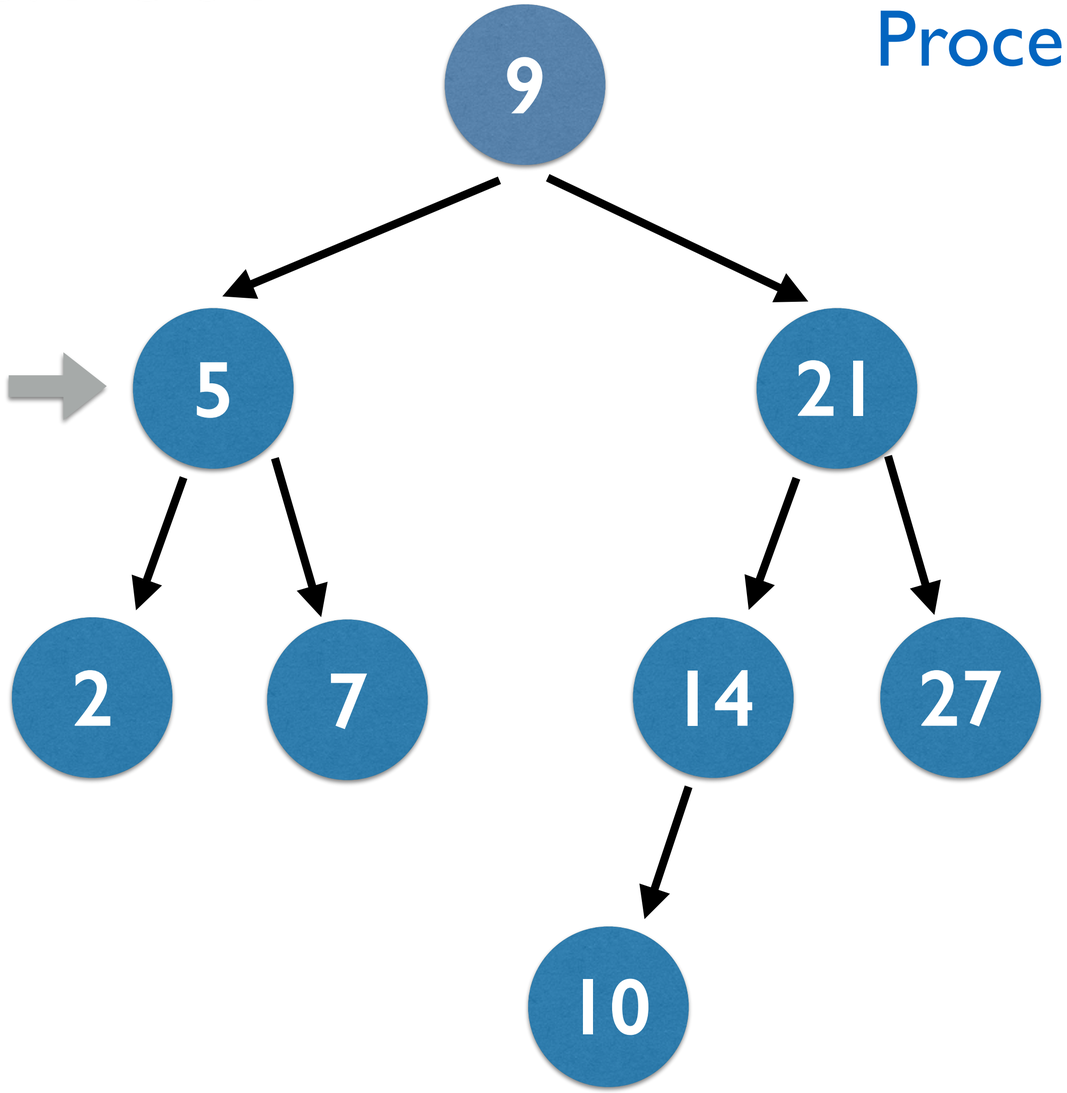
Process left · Process right · Process root

DFS: POST



Process left · Process right · Process root

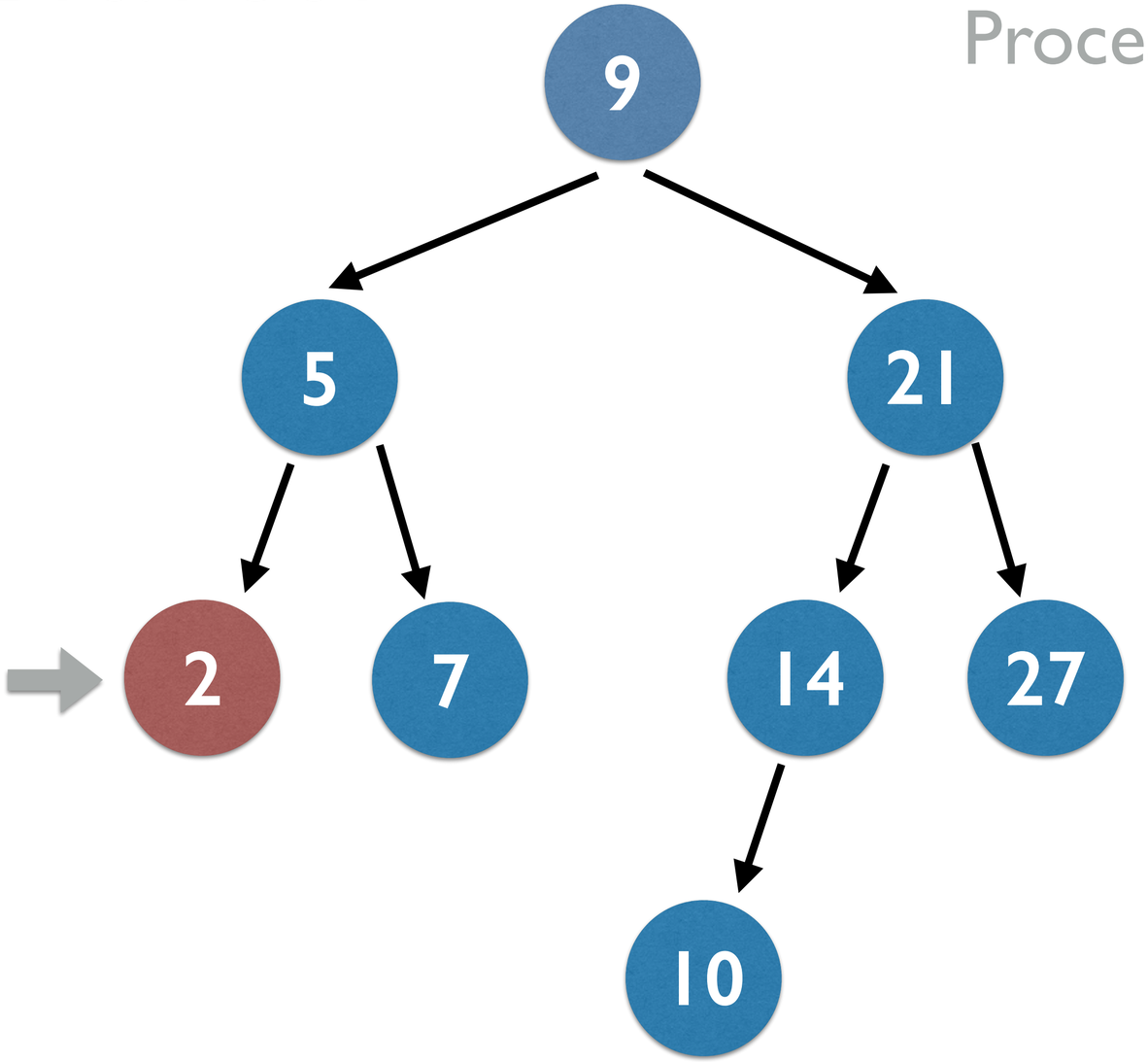
DFS: POST



Process left · Process right · Process root

DFS: POST

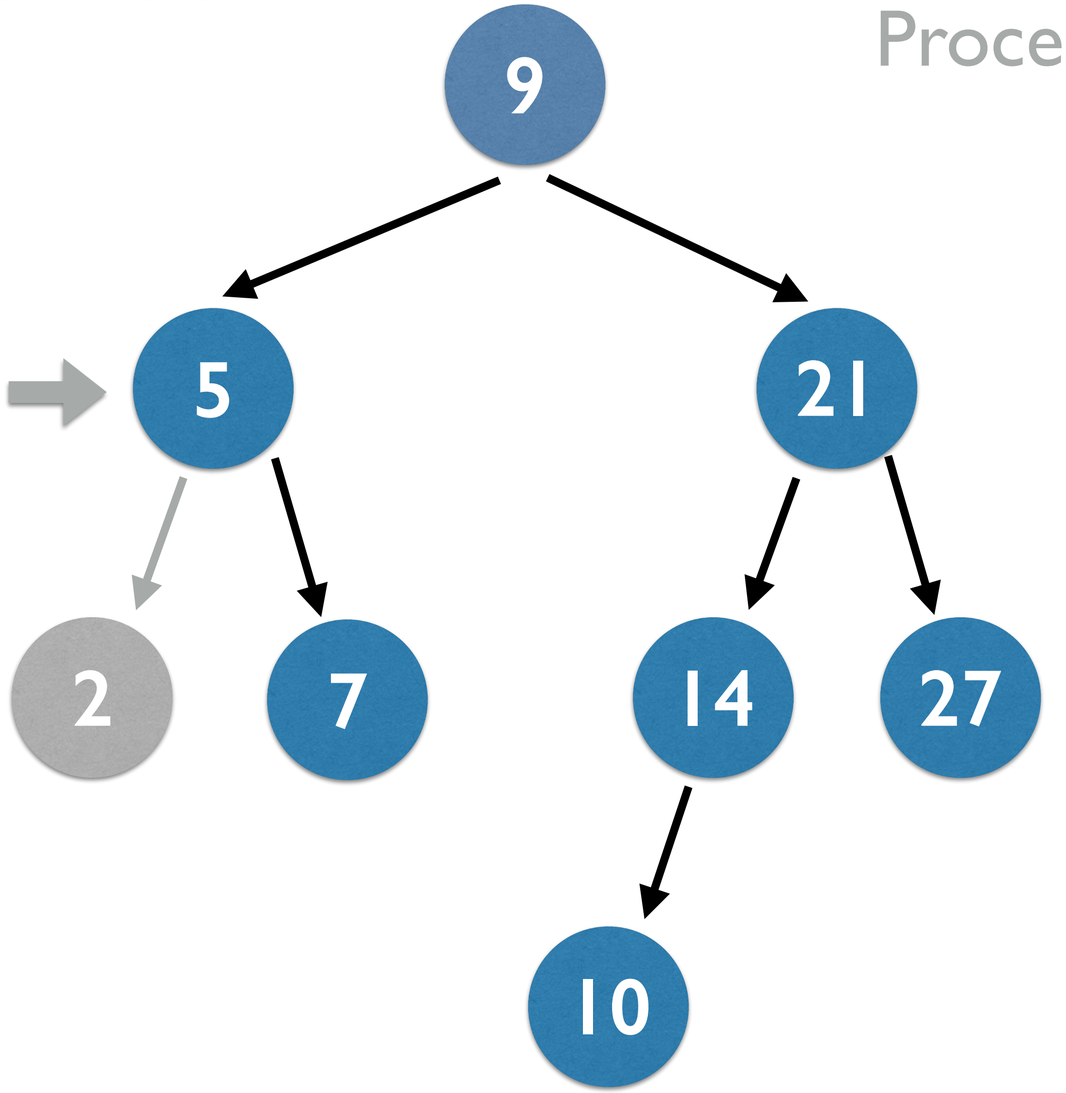
Process left · Process right · Process root



2

DFS: POST

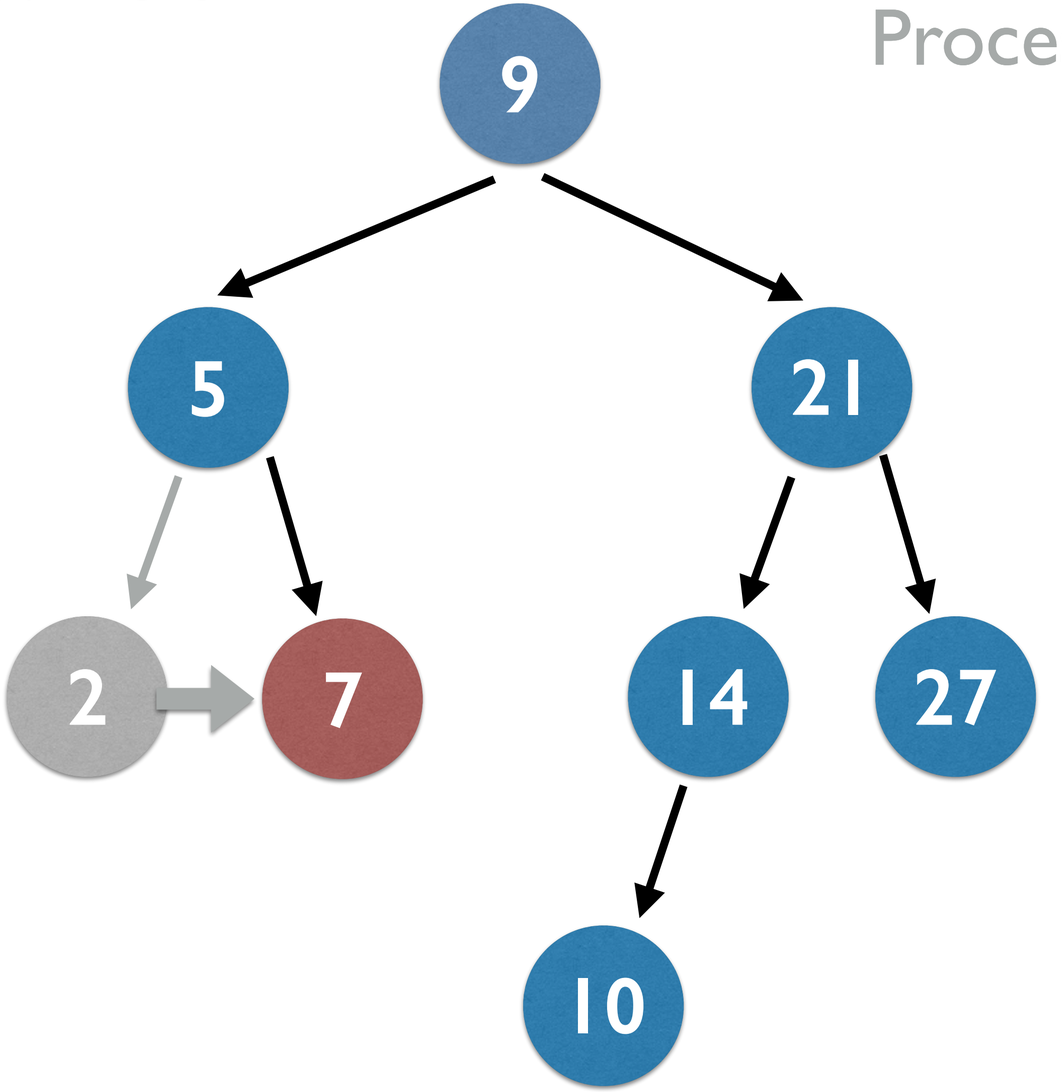
Process left · Process right · Process root



2

DFS: POST

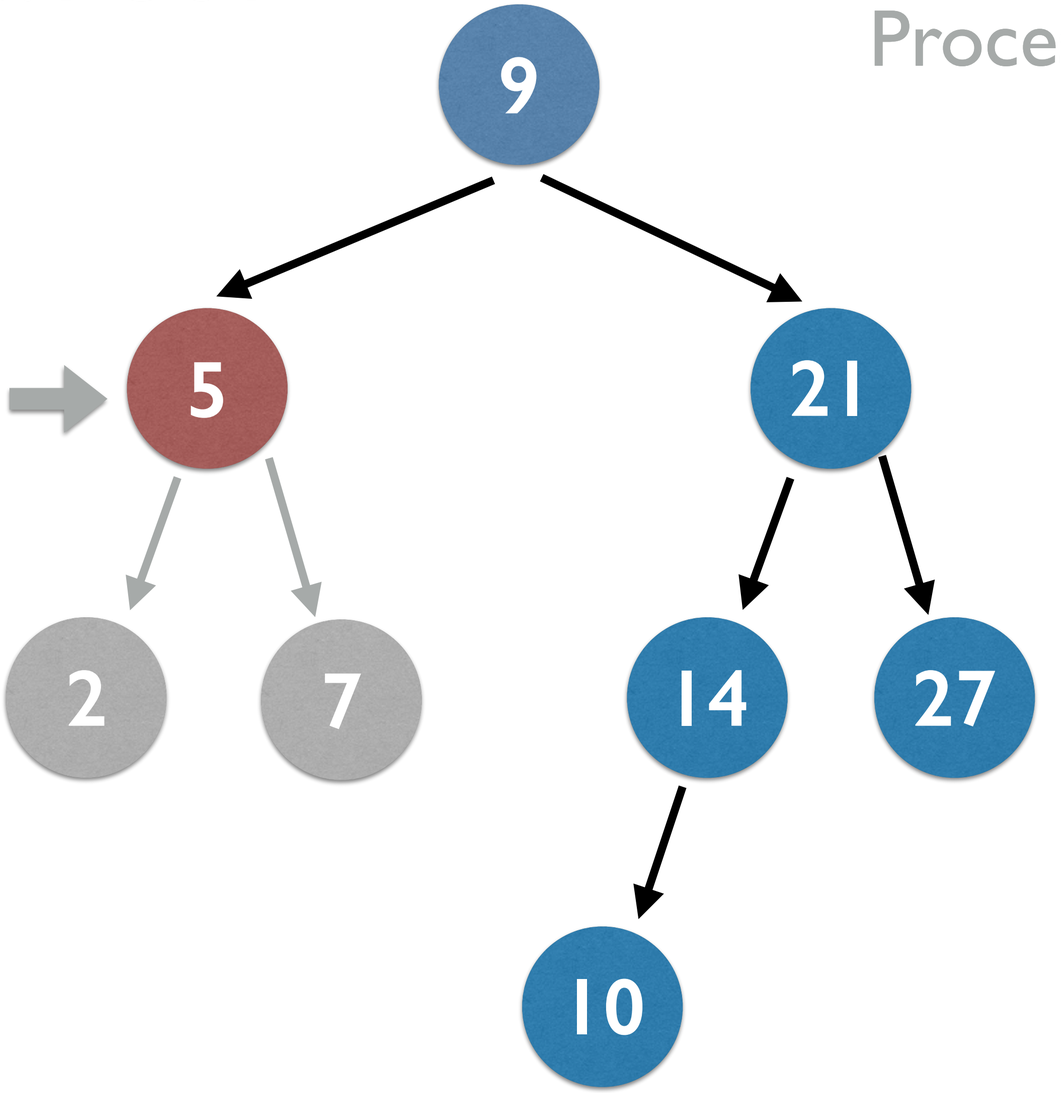
Process left · Process right · **Process root**



2, 7

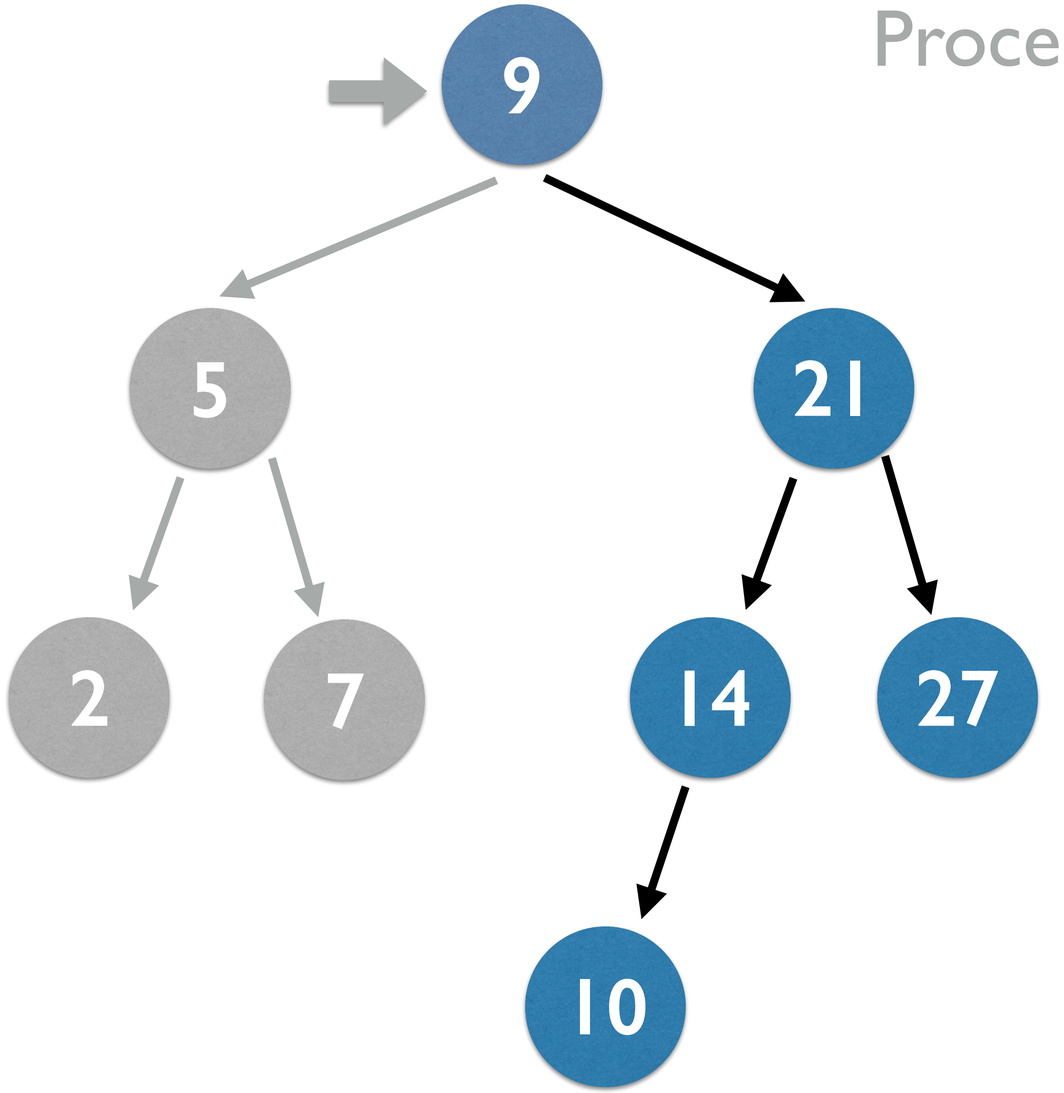
DFS: POST

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



2, 7, 5

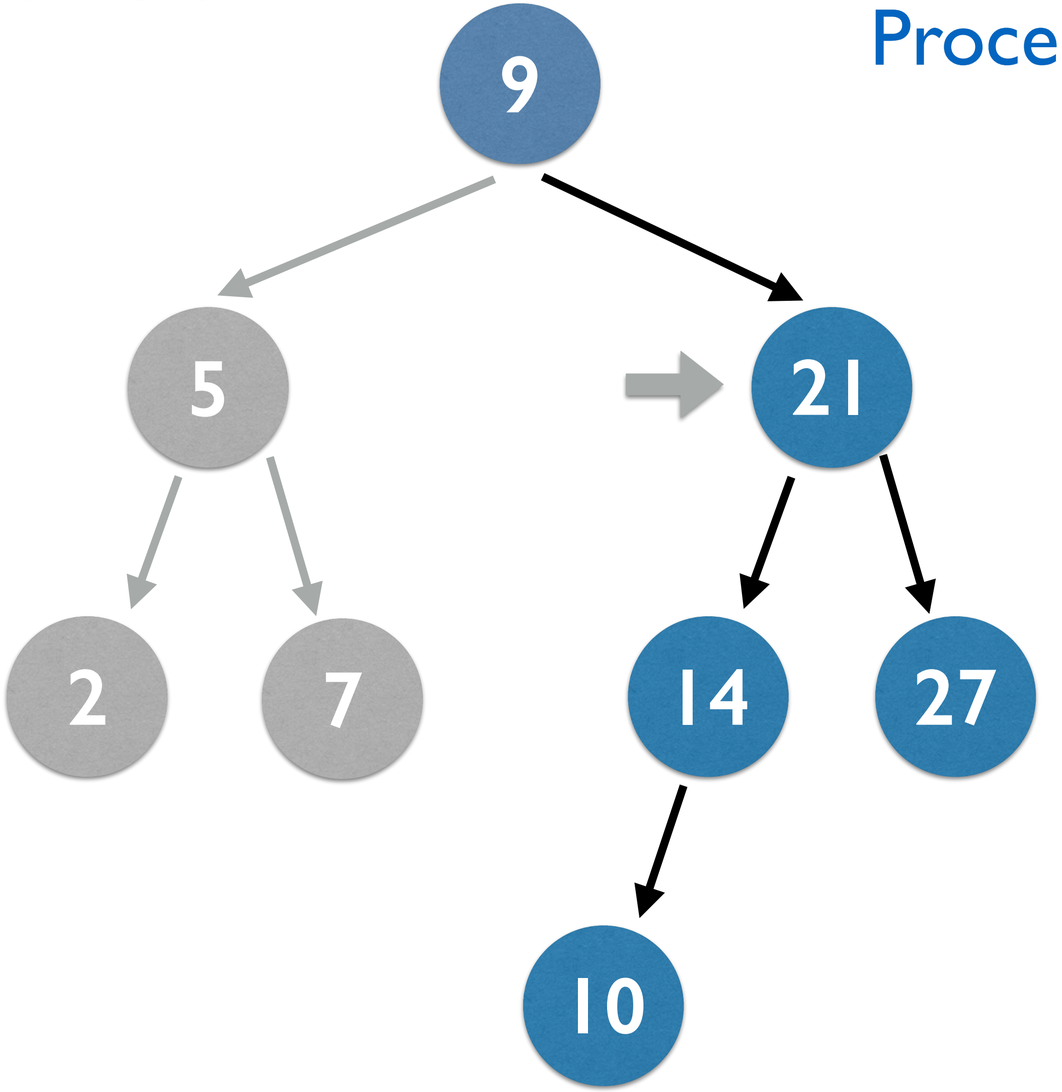
DFS: POST



Process left · Process right · Process root

2, 7, 5

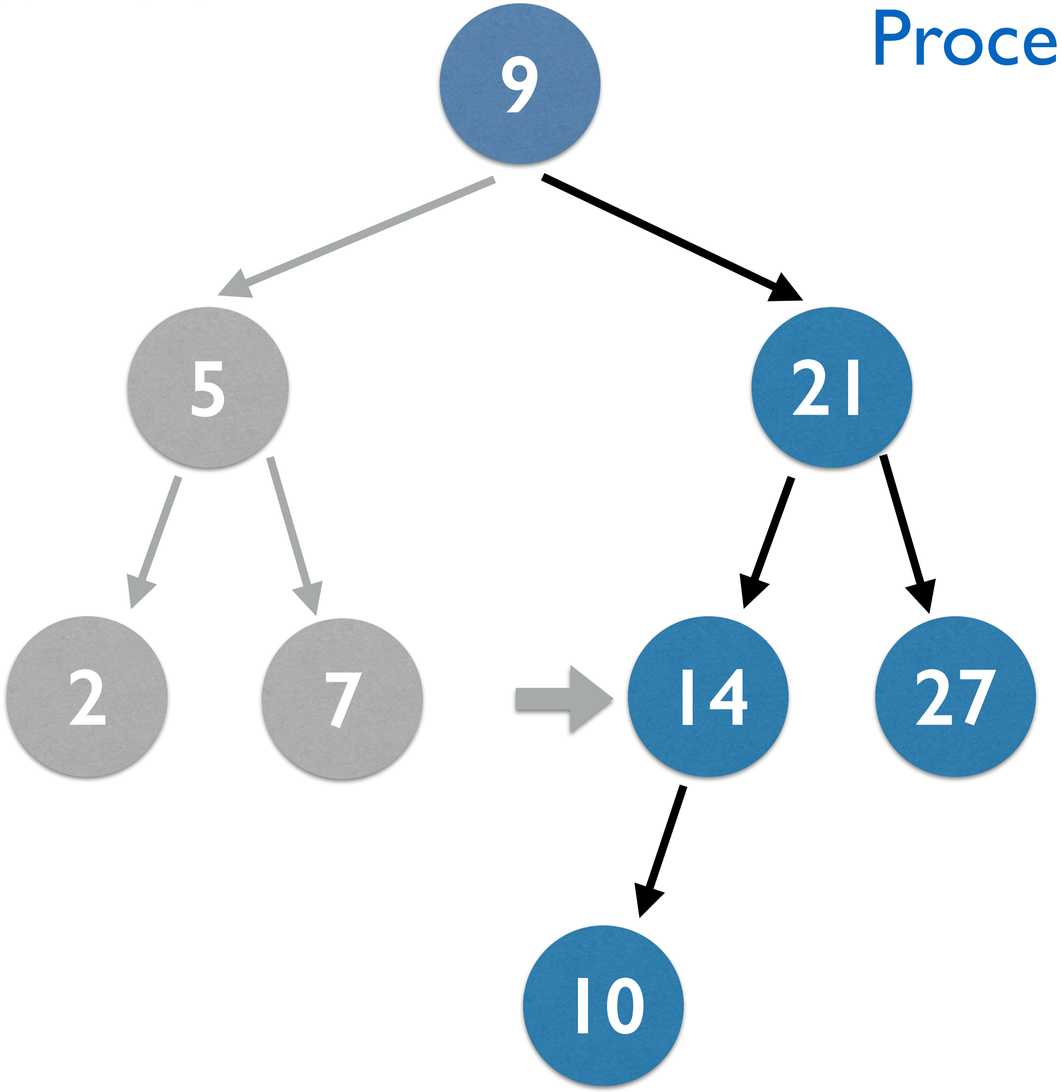
DFS: POST



Process left · Process right · Process root

2, 7, 5

DFS: POST

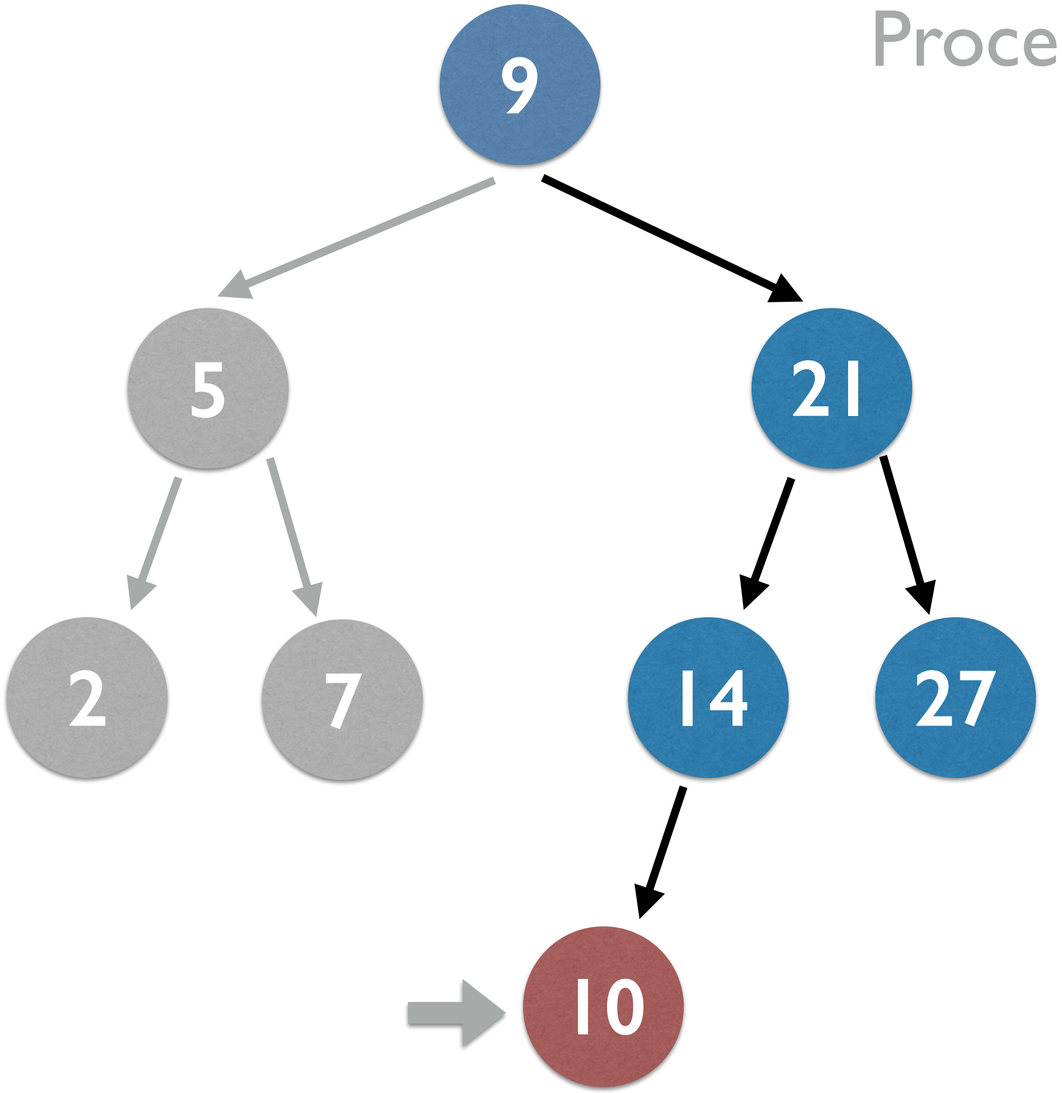


Process left · Process right · Process root

2, 7, 5

DFS: POST

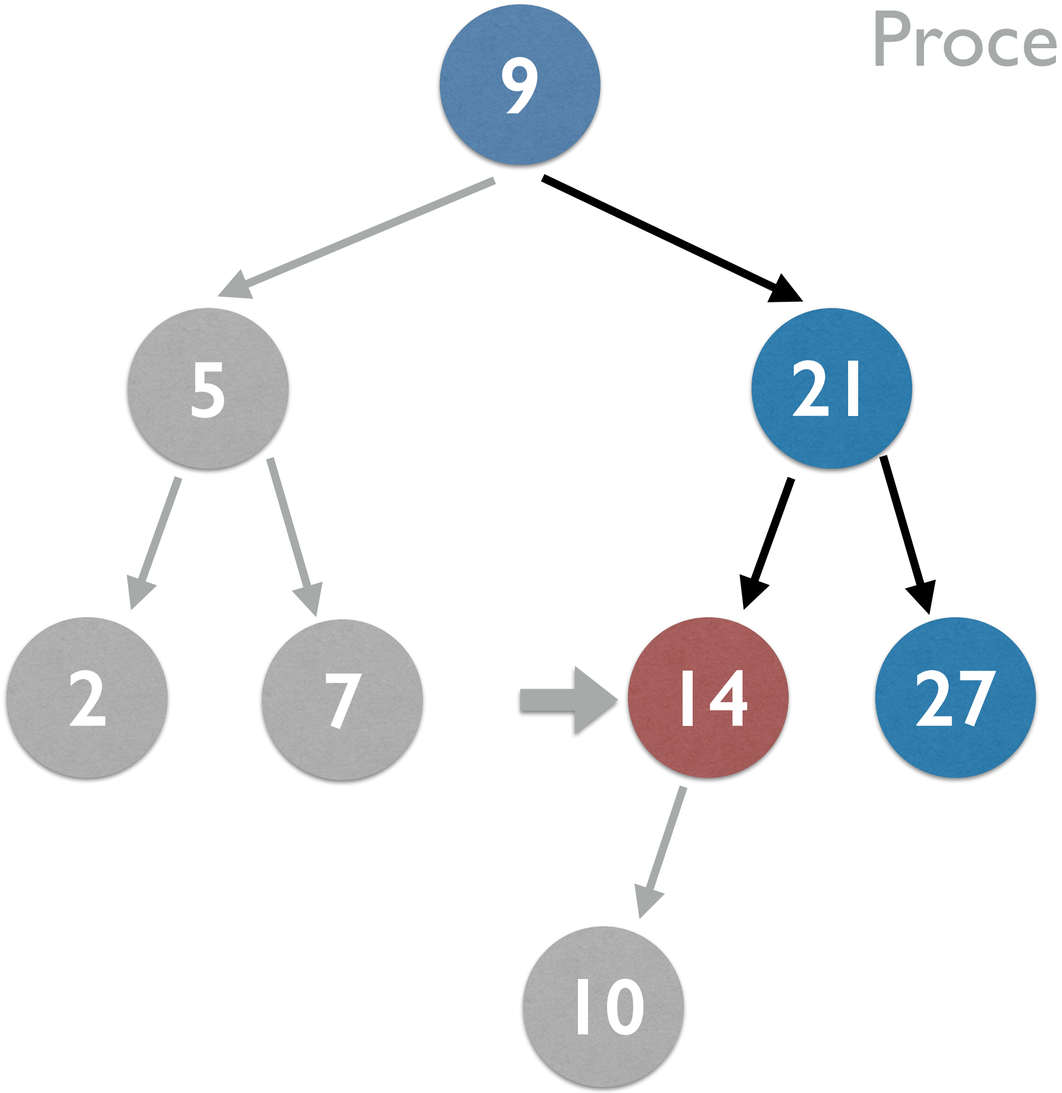
Process left · Process right · **Process root**



2, 7, 5, 10

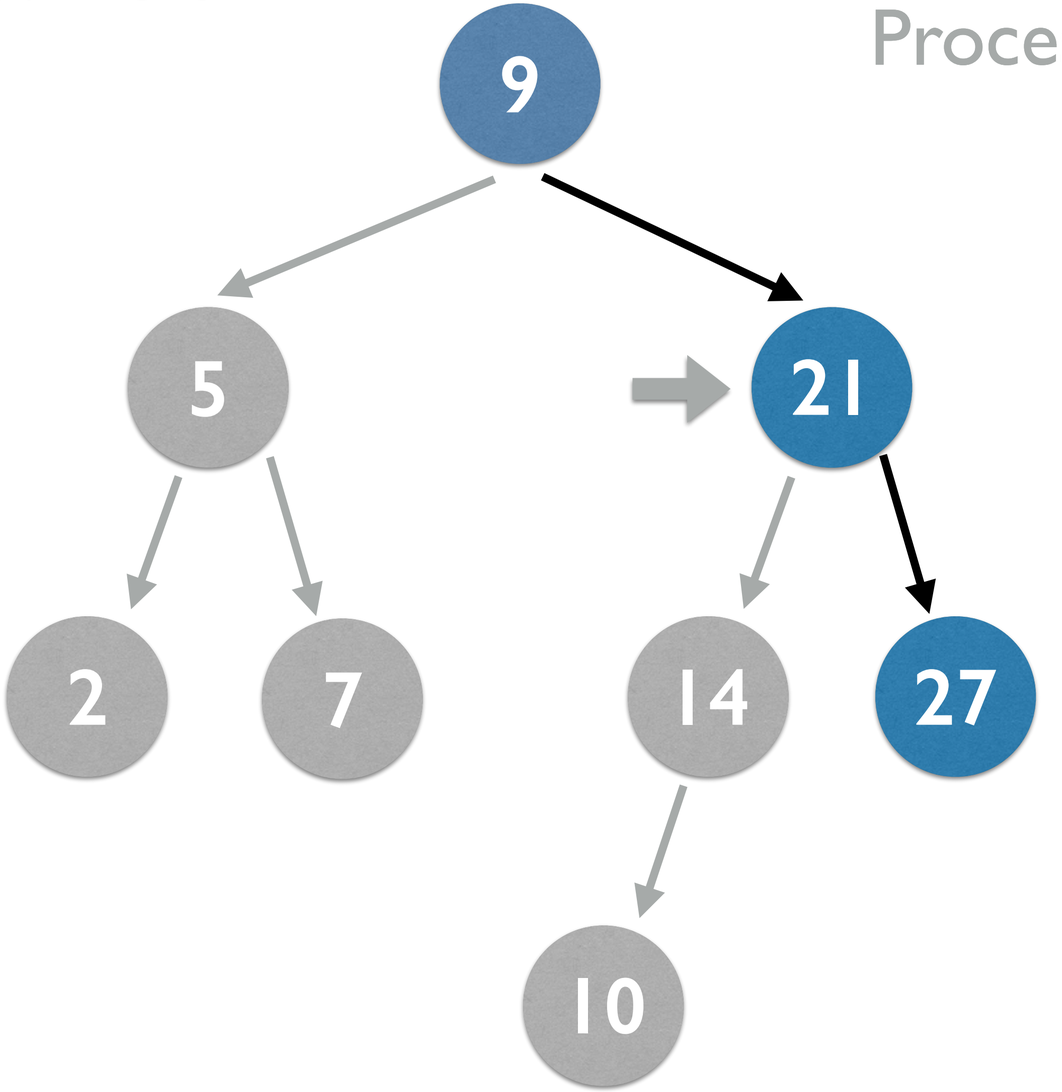
DFS: POST

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



2, 7, 5, 10, 14

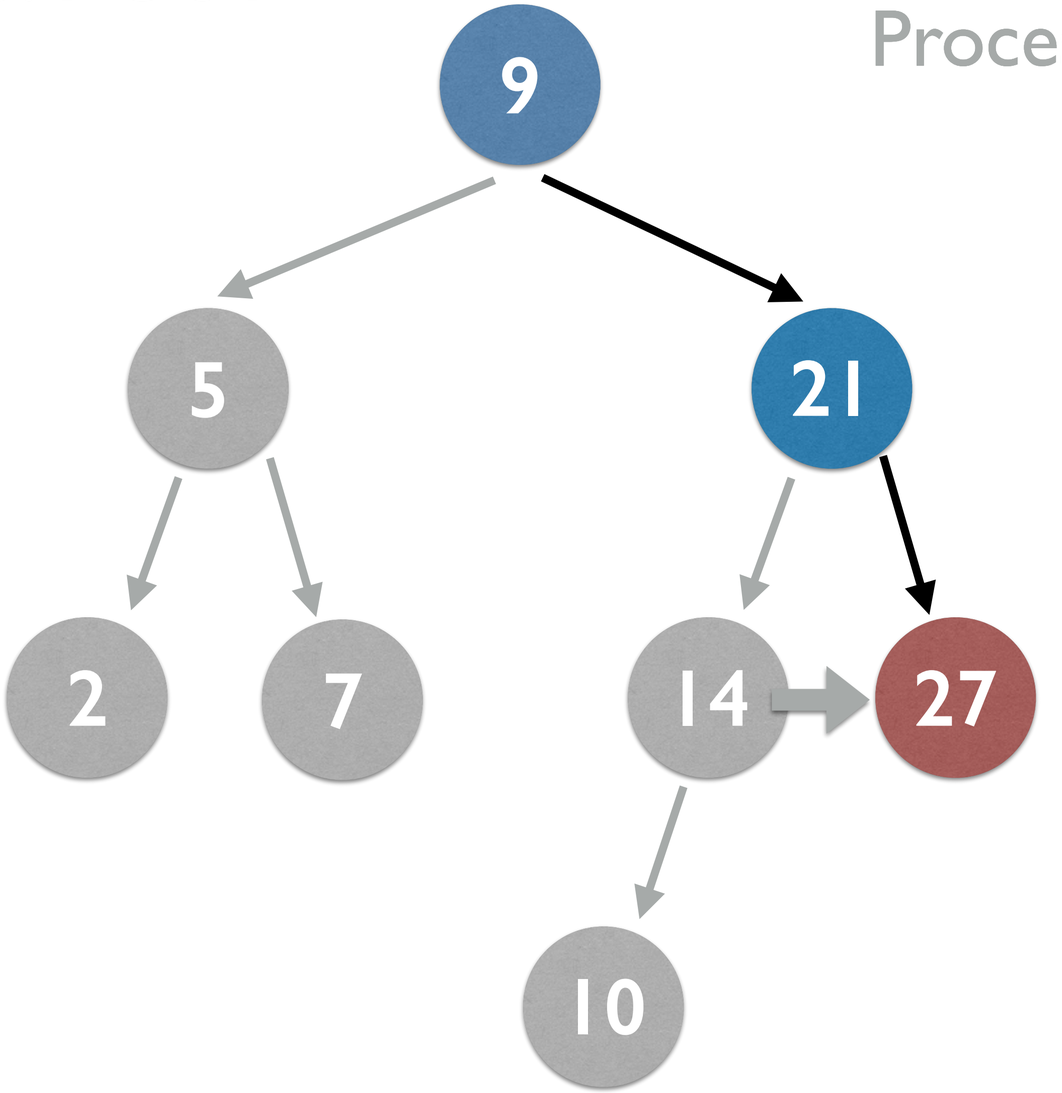
DFS: POST



Process left · Process right · Process root

2, 7, 5, 10, 14

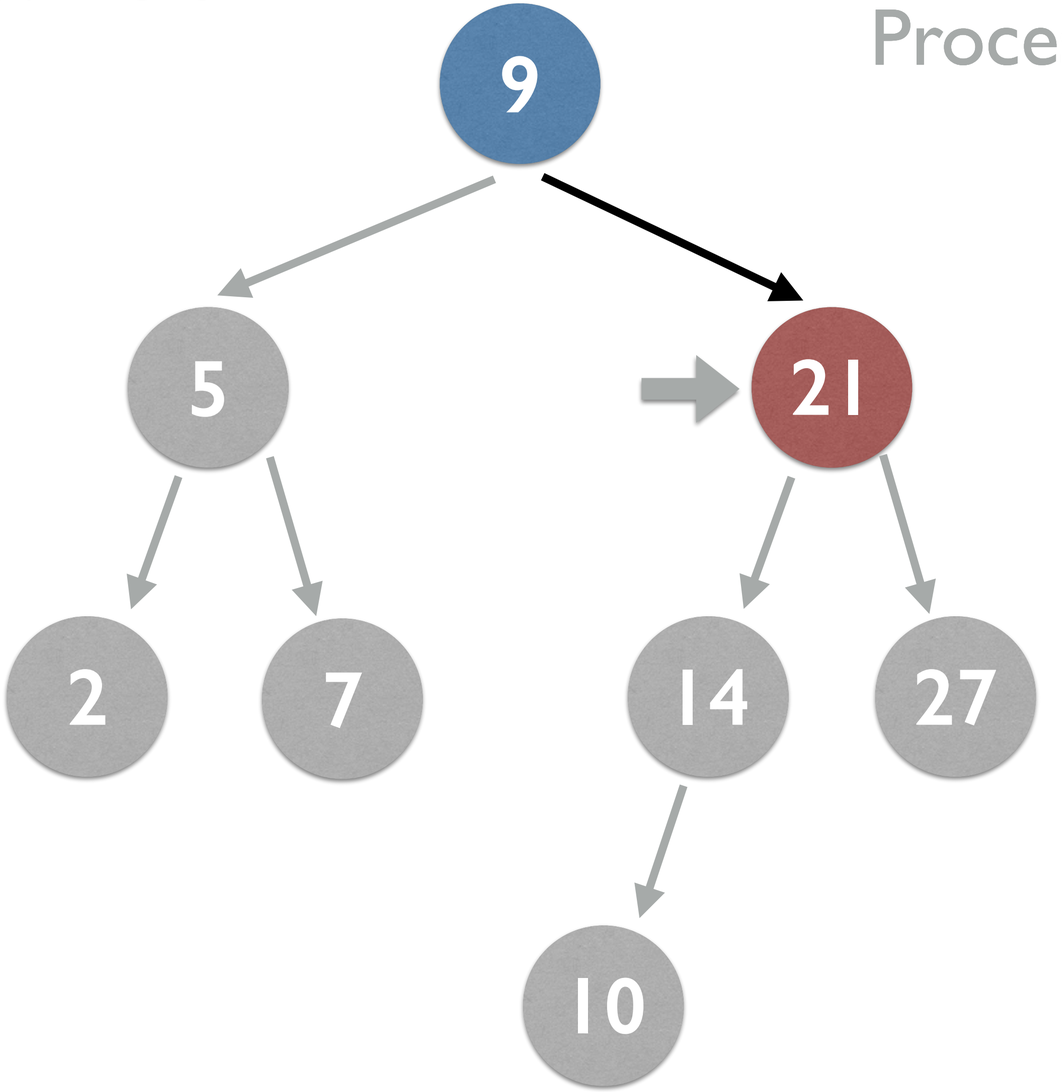
DFS: POST



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

2, 7, 5, 10, 14, 27

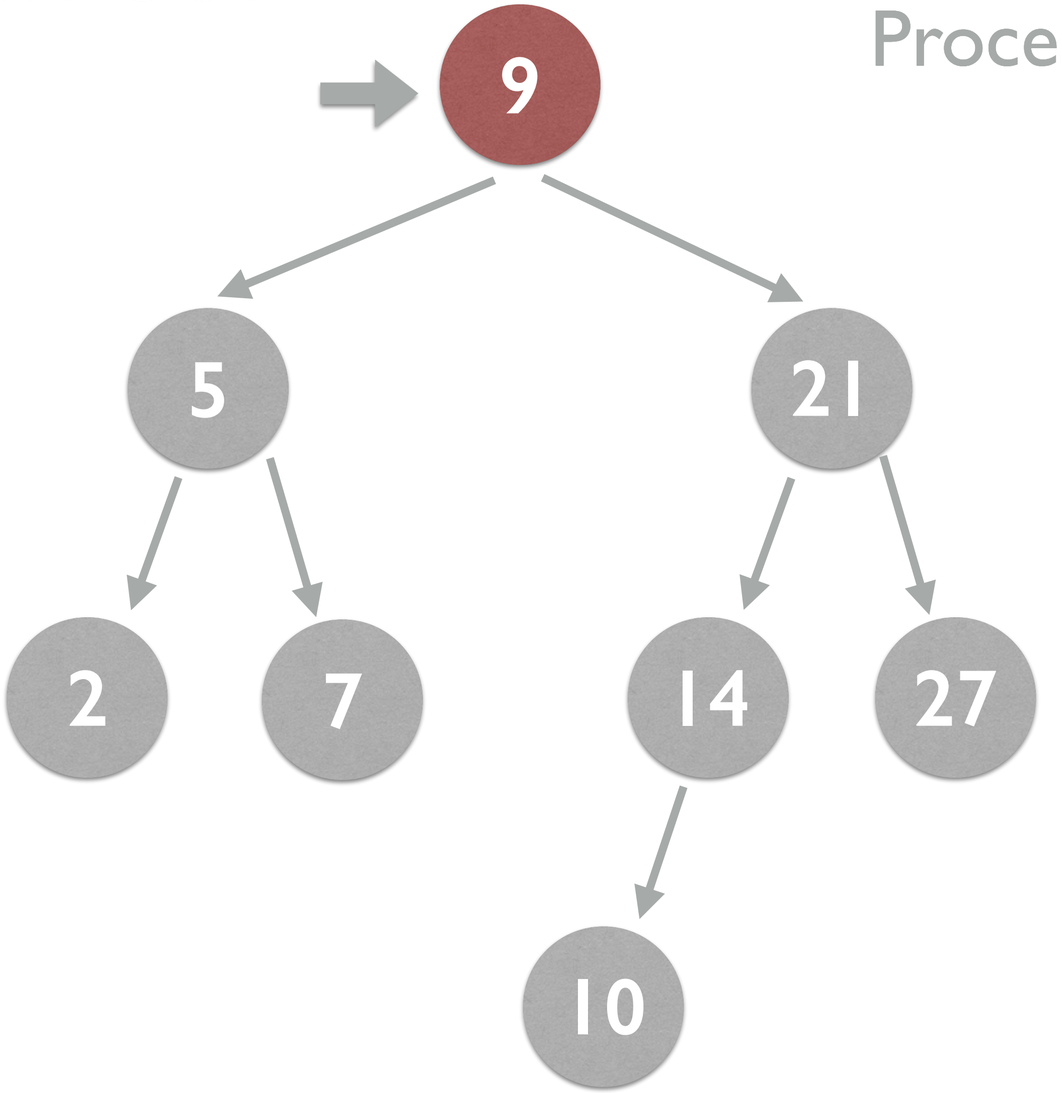
DFS: POST



Process left · Process right · Process root

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

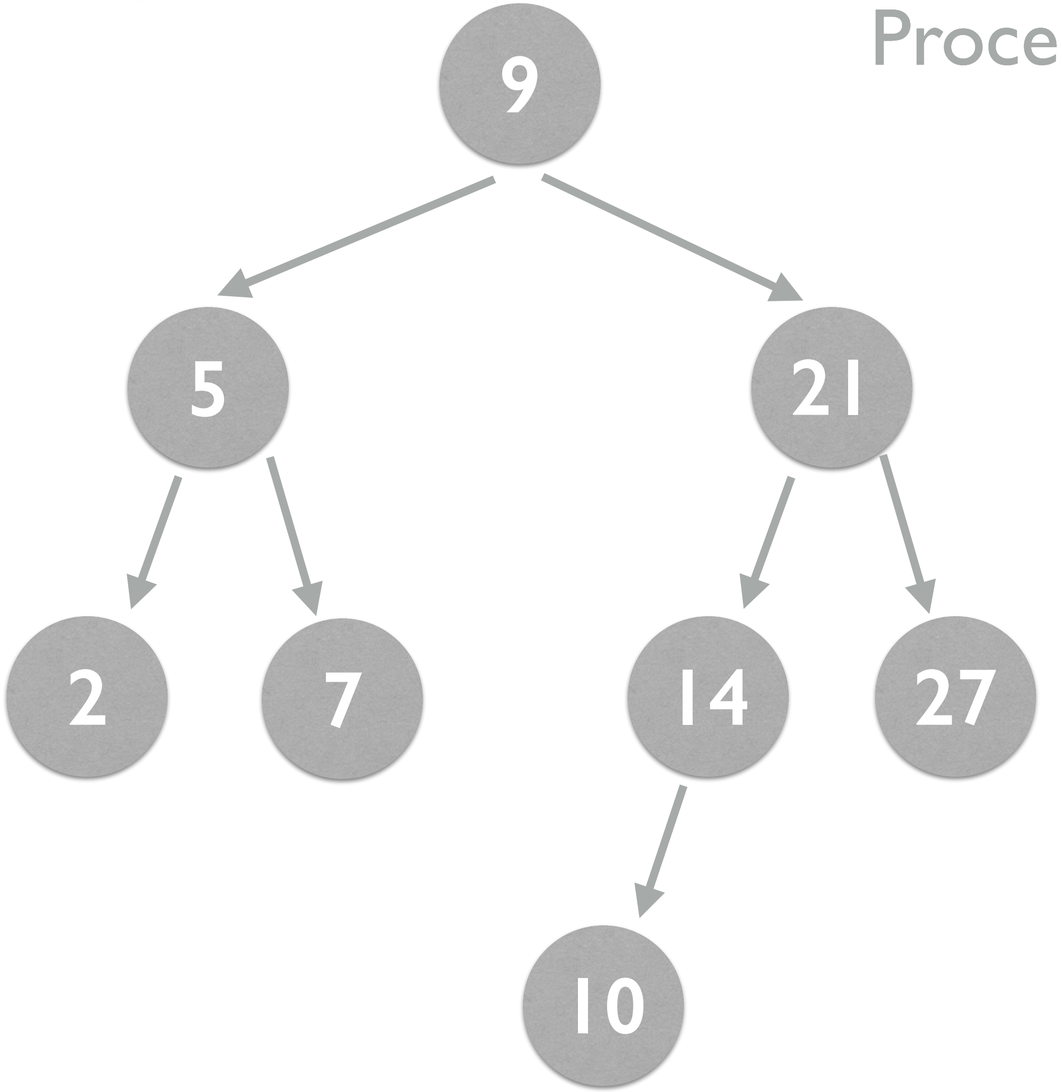
DFS: POST



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)