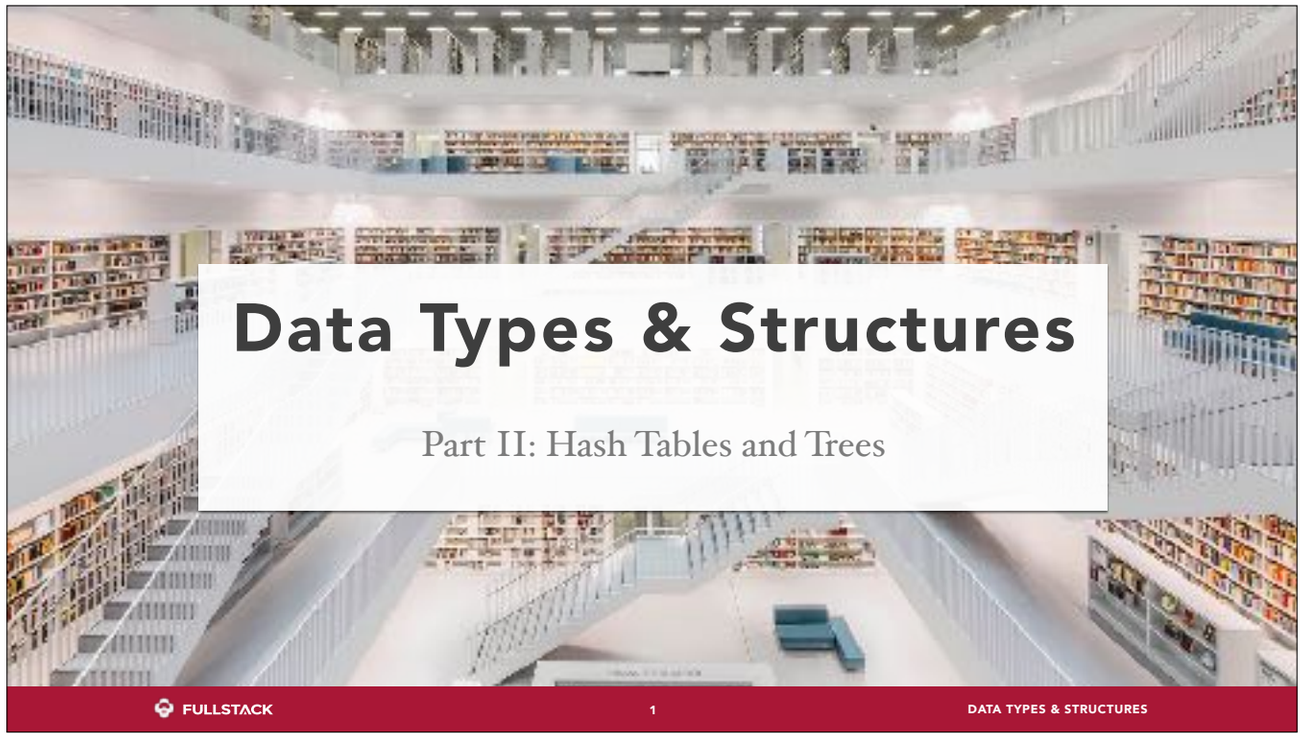
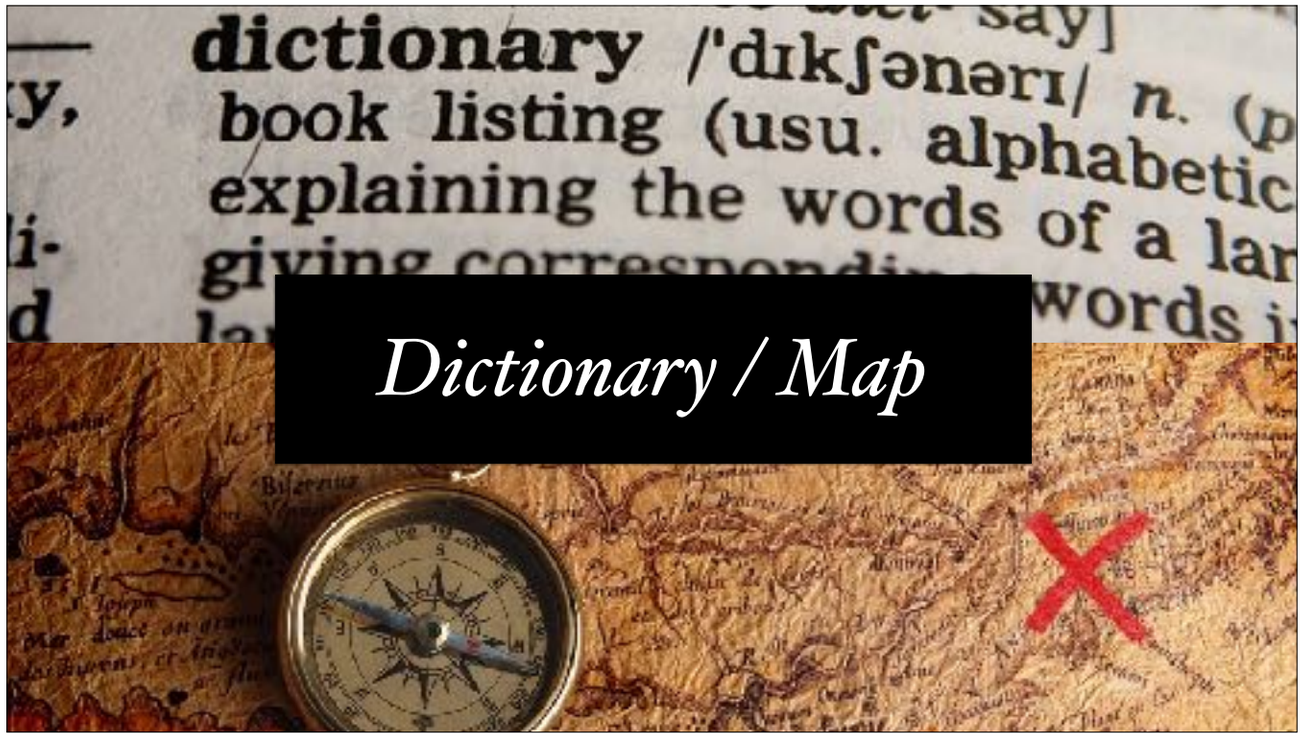


Data Types & Structures

Part II: Hash Tables and Trees


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

Note: the dynamic alteration aspect (adding and removing keys) is what makes a hash table much trickier than a simple *struct*.

How to implement this ADT?

(ask audience)

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
}
```


Note that good hashing functions are typically more complex than this. They need to have excellent distribution and other slightly more esoteric mathematical properties.

contact.name = 'Jenny'

contact.phone = 8675309

Our goal

Example with a 9-bucket array

1. We want to store the value 'Jenny' for the key 'name'.
2. Hashing the string 'name' yields the numerical index 3.
3. Store 'Jenny' 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	
4	
5	
6	
7	
8	

Example with a 9-bucket array

1. We want to store the value 'Jenny' for the key 'name'.
2. Hashing the string 'name' yields the numerical index 3.
3. Store 'Jenny' 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	
4	
5	
6	
7	
8	

Example with a 9-bucket array

1. We want to store the value 'Jenny' for the key 'name'.
2. Hashing the string 'name' yields the numerical index 3.
3. Store 'Jenny' 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	Jenny
4	
5	
6	
7	
8	

Example with a 9-bucket array

1. We want to store the value 'Jenny' for the key 'name'.
2. Hashing the string 'name' yields the numerical index 3.
3. Store 'Jenny' 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	Jenny
4	
5	
6	
7	
8	

Example with a 9-bucket array

1. We want to store the value 'Jenny' for the key 'name'.
2. Hashing the string 'name' yields the numerical index 3.
3. Store 'Jenny' 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	Jenny
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: 'Jenny'
5. Similar steps for checking the key 'phone', gets the value 8675309.

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

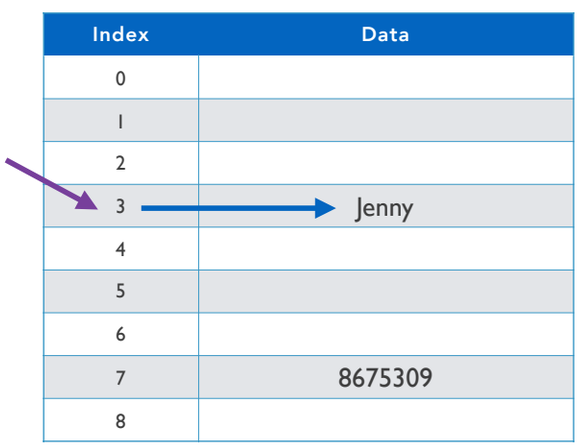
Fetching/changing values

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: 'Jenny'
5. Similar steps for checking the key 'phone', gets the value 8675309.

Index	Data
0	
1	
2	
3	Jenny
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: 'Jenny'
5. Similar steps for checking the key 'phone', gets the value 8675309.



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: 'Jenny'
5. Similar steps for checking the key 'phone', gets the value 8675309.

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

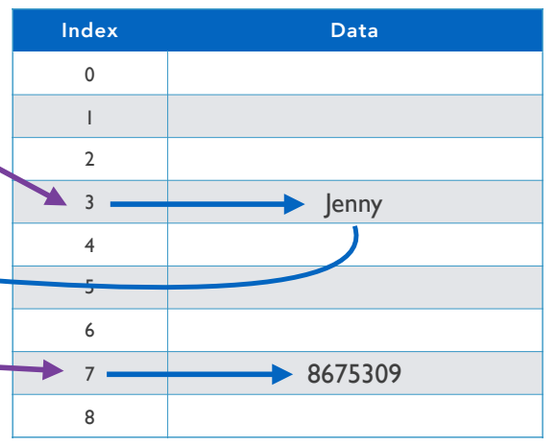
Fetching/changing values

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: 'Jenny'
5. Similar steps for checking the key 'phone', gets the value 8675309.

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

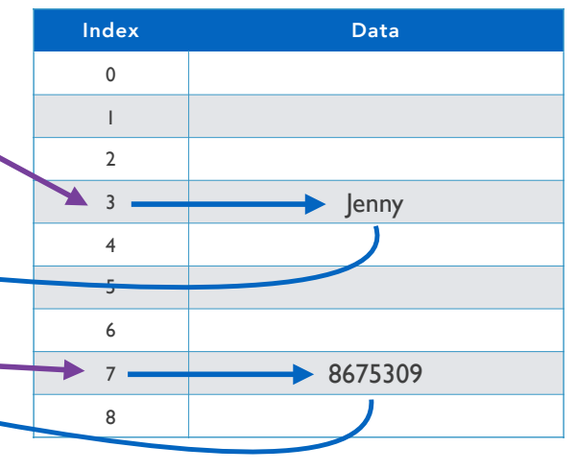
Fetching/changing values

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: 'Jenny'
5. Similar steps for checking the key 'phone', gets the value 8675309.



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: 'Jenny'
5. Similar steps for checking the key 'phone', gets the value 8675309.



Anyone see a problem?

(ask audience)

Collisions

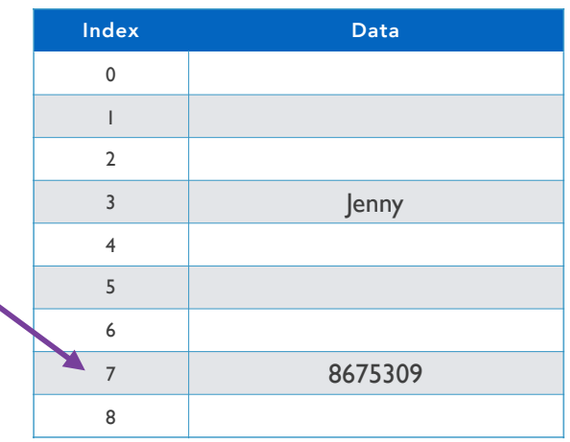
1. Now we want to store the value jenny@gmail.com for the key email.
2. Hashing email yields the numerical index 7.
3. But we already have a value there (for phone)!
4. Because there are many more possible keys than buckets, collisions are inevitable.

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

We are using an exaggeratedly small number of "buckets". Good hash tables have a lot more buckets (reducing odds of collision), and for rather esoteric reasons related to the hash function chosen, a prime number of buckets typically results in better distribution of values (again, fewer collisions). But even with a decent number of buckets, it is still going to be very limited compared to the gargantuan number of possible key strings. Note that if all permitted keys are known beforehand and enough buckets are provided, it is possible to have a "perfect" hash function which maps each key to a unique bucket. But that is not the typical case.

Collisions

1. Now we want to store the value jenny@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.

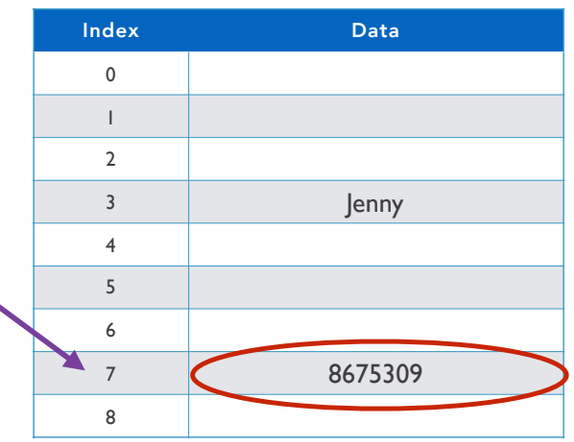


Index	Data
0	
1	
2	
3	Jenny
4	
5	
6	
7	8675309
8	

We are using an exaggeratedly small number of "buckets". Good hash tables have a lot more buckets (reducing odds of collision), and for rather esoteric reasons related to the hash function chosen, a prime number of buckets typically results in better distribution of values (again, fewer collisions). But even with a decent number of buckets, it is still going to be very limited compared to the gargantuan number of possible key strings. Note that if all permitted keys are known beforehand and enough buckets are provided, it is possible to have a "perfect" hash function which maps each key to a unique bucket. But that is not the typical case.

Collisions

1. Now we want to store the value jenny@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.



We are using an exaggeratedly small number of "buckets". Good hash tables have a lot more buckets (reducing odds of collision), and for rather esoteric reasons related to the hash function chosen, a prime number of buckets typically results in better distribution of values (again, fewer collisions). But even with a decent number of buckets, it is still going to be very limited compared to the gargantuan number of possible key strings. Note that if all permitted keys are known beforehand and enough buckets are provided, it is possible to have a "perfect" hash function which maps each key to a unique bucket. But that is not the typical case.

How to resolve collisions?

(ask audience)

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.

The workshop will take the second approach, known as "separate chaining."

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.

The workshop will take the second approach, known as "separate chaining."

Separate chaining: adding a value

1. We want to store the value jenny@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	<Jenny>
4	
5	
6	
7	<8675309>
8	

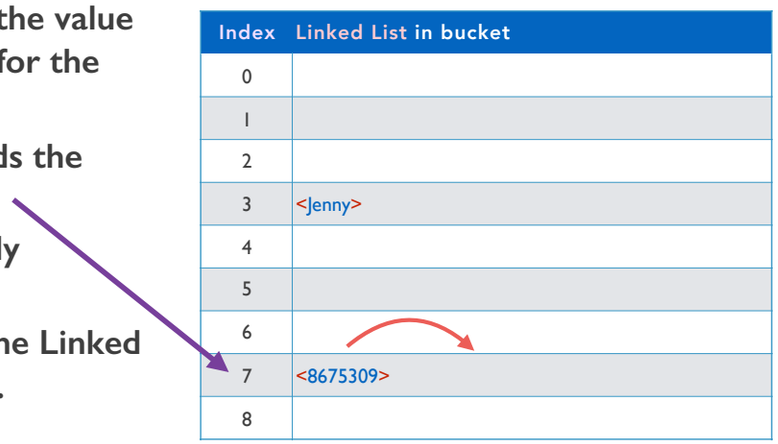
Separate chaining: adding a value

1. We want to store the value jenny@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	<Jenny>
4	
5	
6	
7	<8675309>
8	

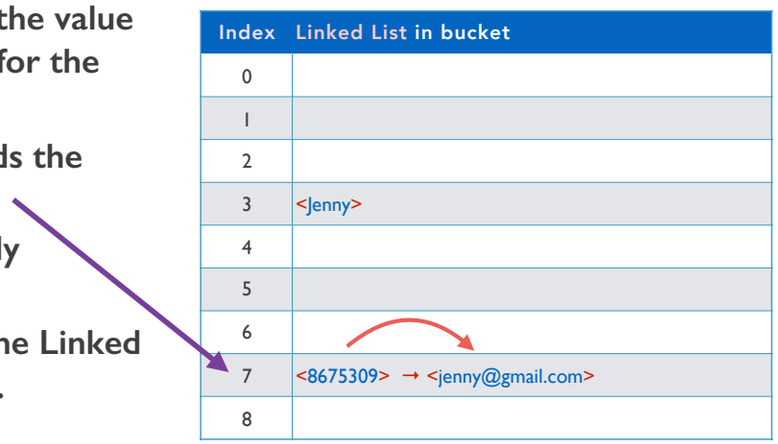
Separate chaining: adding a value

1. We want to store the value jenny@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.



Separate chaining: adding a value

1. We want to store the value jenny@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.



We still have a problem...

(ask audience)

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	<Jenny>
4	
5	
6	
7	<8675309> → <jenny@gmail.com>
8	

Any suggestions?

Separate chaining: retrieving a value

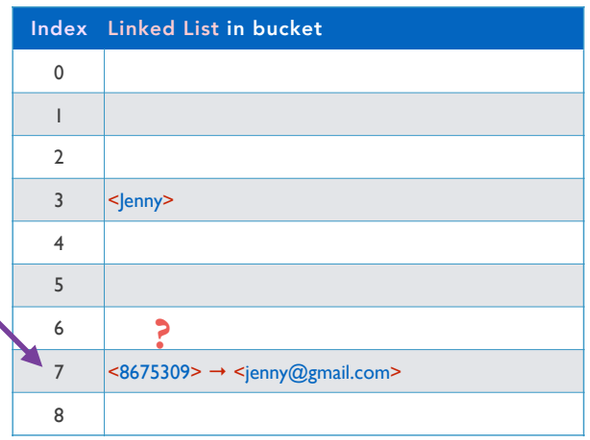
1. What is the value for the key email?
2. Hashing email yields the numerical index 7.
3. In bucket 7 there is a linked list.
4. There are *two* nodes in that list; how do we know which value is for the key email?

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

Any suggestions?

Separate chaining: retrieving a value

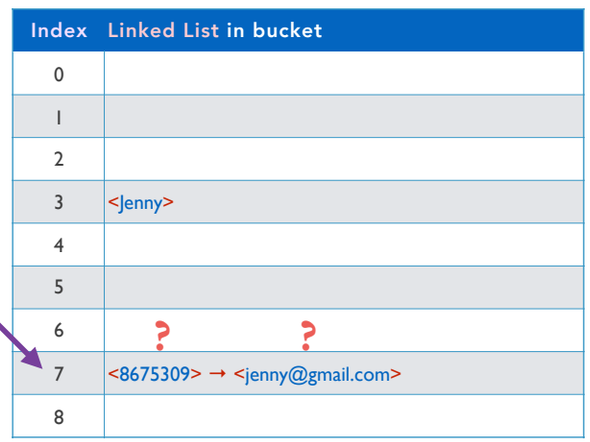
1. What is the value for the key email?
2. Hashing email yields the numerical index 7.
3. In bucket 7 there is a linked list.
4. There are *two* nodes in that list; how do we know which value is for the key email?



Any suggestions?

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	<Jenny>
4	
5	
6	? ?
7	<8675309> → <jenny@gmail.com>
8	

Any suggestions?

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: jenny@gmail.com

Index	Linked List in bucket
0	
1	
2	
3	<name, Jenny>
4	
5	
6	
7	<phone, 8675309> → <email, jenny@gmail.com>
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: jenny@gmail.com

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

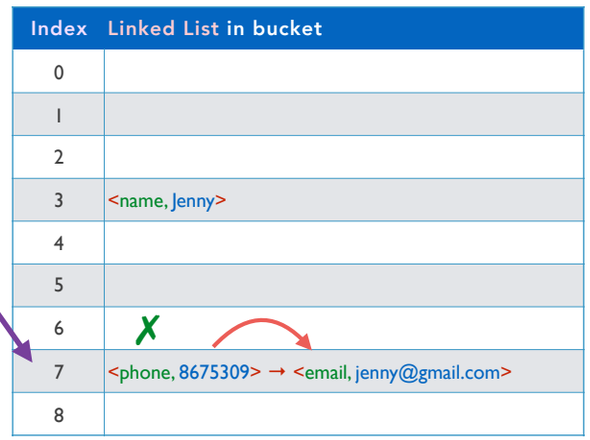
Separate chaining: store value *and* key

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: jenny@gmail.com

Index	Linked List in bucket
0	
1	
2	
3	<name, Jenny>
4	
5	
6	✗
7	<phone, 8675309> → <email, jenny@gmail.com>
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: jenny@gmail.com



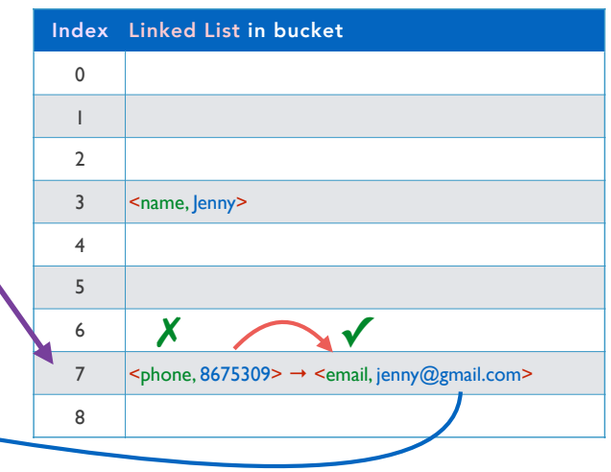
Separate chaining: store value *and* key

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: jenny@gmail.com

Index	Linked List in bucket
0	
1	
2	
3	<name, Jenny>
4	
5	
6	✗ ✓
7	<phone, 8675309> → <email, jenny@gmail.com>
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: jenny@gmail.com



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

(ask audience)

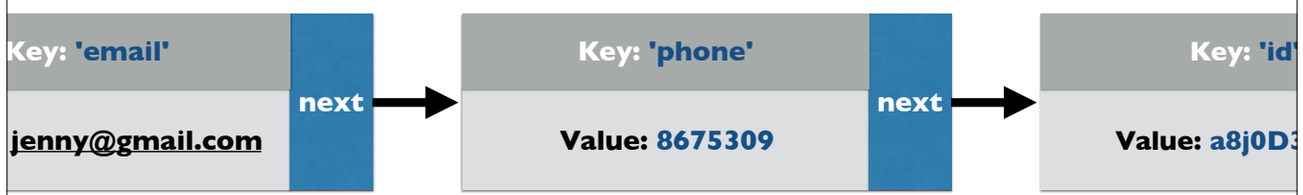
Performance

- 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?

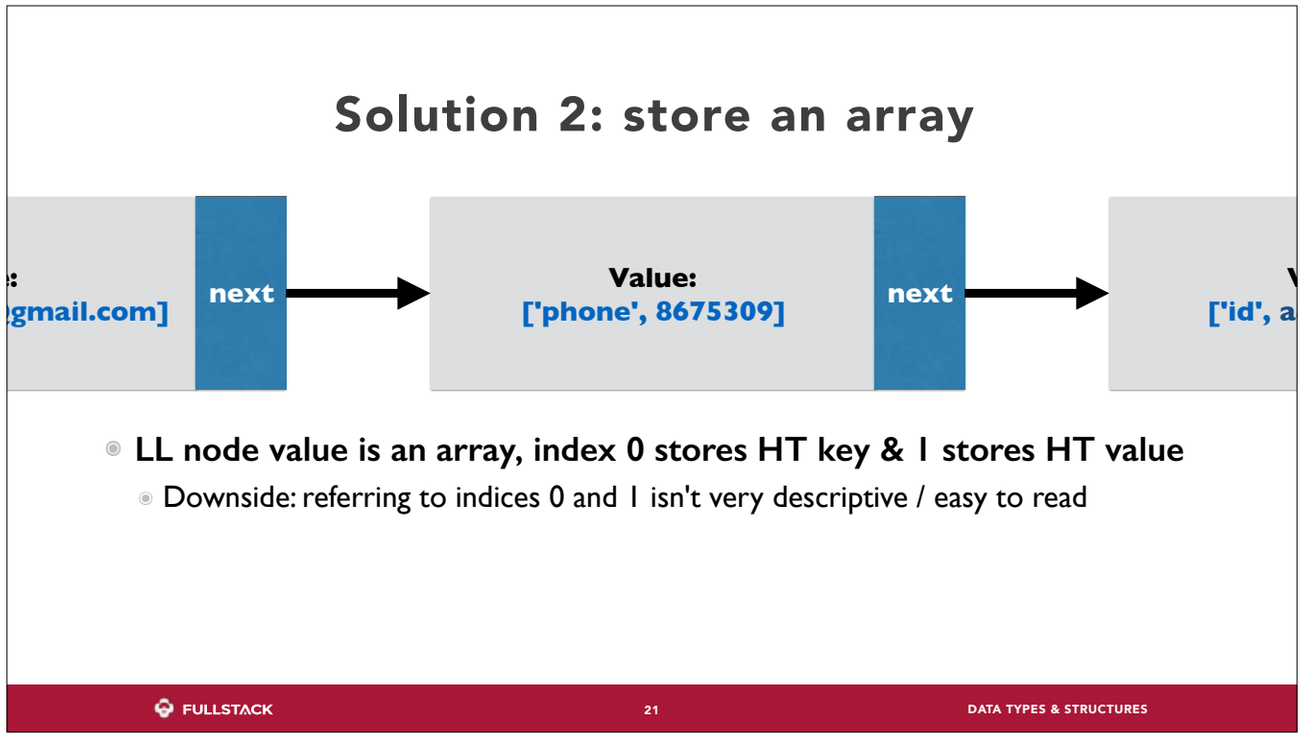
(ask audience)

Solution 1: *Association List*

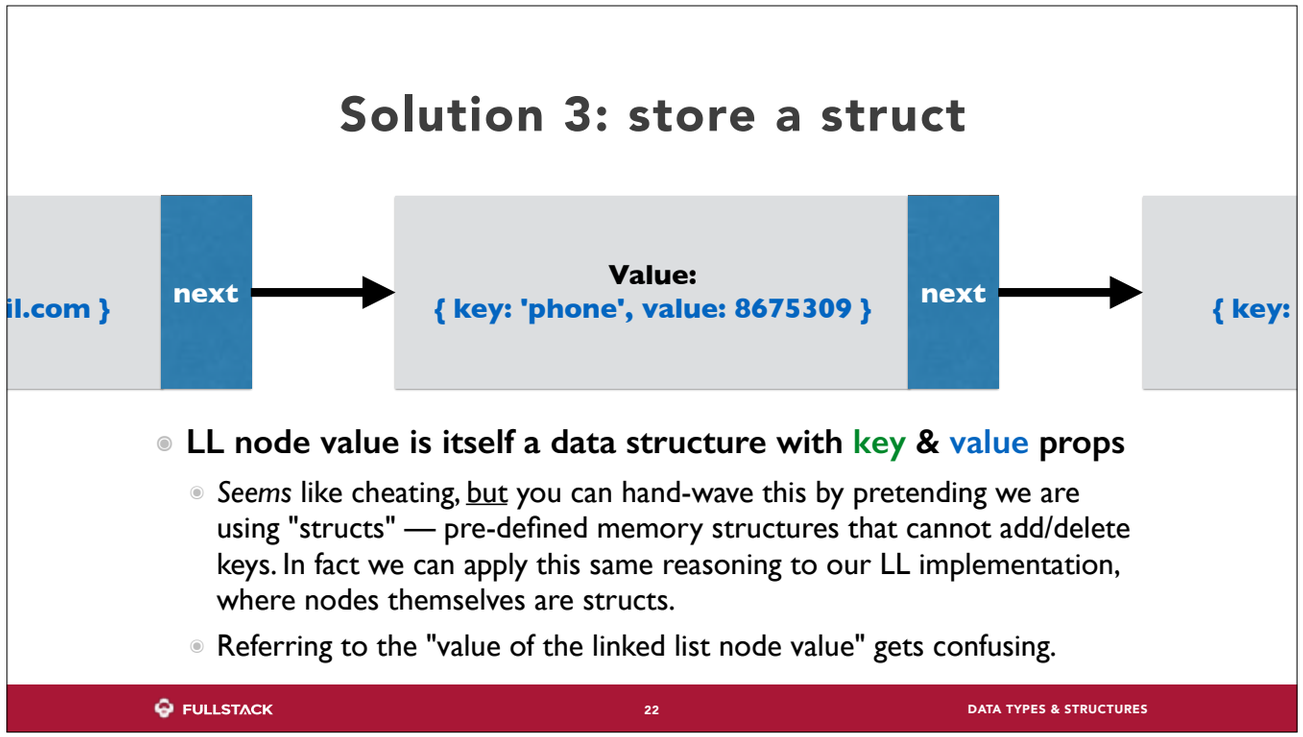


- Nodes have key as well as value & next
  - Advantage: best for purpose, most straightforward.
  - Downside: need a custom LL implementation

Association lists are linked lists which don't store just a value, but rather key-value pairs. That makes them ideal for use in hash table buckets. Our linked list doesn't support this behavior; if you want you can copy your linked list and modify it to support search by key, setting the value of a node by key, etc. That's a lot of extra work however.



We could alternatively use our existing linked list. How can we store multiple elements in the `value` field of a linked list node? Well, one way would be to wrap them in a container. An array would be one such container; you can pretend for the sake of the exercise that JS arrays are "real" arrays (and not just extensions of objects). Downside, storing the hash table key in node.value[0] and hash table value in node.value[1] is quite opaque.



Accessing the hash table key in node.value[0] and the hash table value in node.value[1] gets a little confusing. Slightly better would be if we could use a *struct*, which is another way to store properties with string keys. JS doesn't have structs, but you can pretend a simple object is a struct so long as you never add new keys (properties) to it. The main difference between a struct and a hash table is that you have to pre-define all the keys that a struct can ever have, and it can never have new keys added or removed. That ends up far simpler to implement such that structs are really like arrays where each key is just a pre-determined alias for an index. It makes it slightly more descriptive to talk about node.value.key and node.value.value. Of course it is still a bit confusing that we have Linked List nodes with a "value" and that value stores a struct with a "value."

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.**

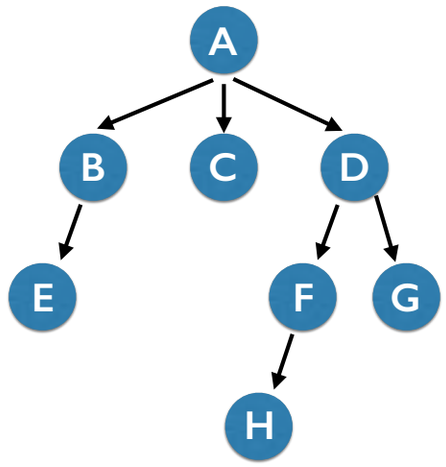
https://developers.google.com/v8/design#fast-property-access
http://jayconrod.com/posts/52/a-tour-of-v8-object-representation

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

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

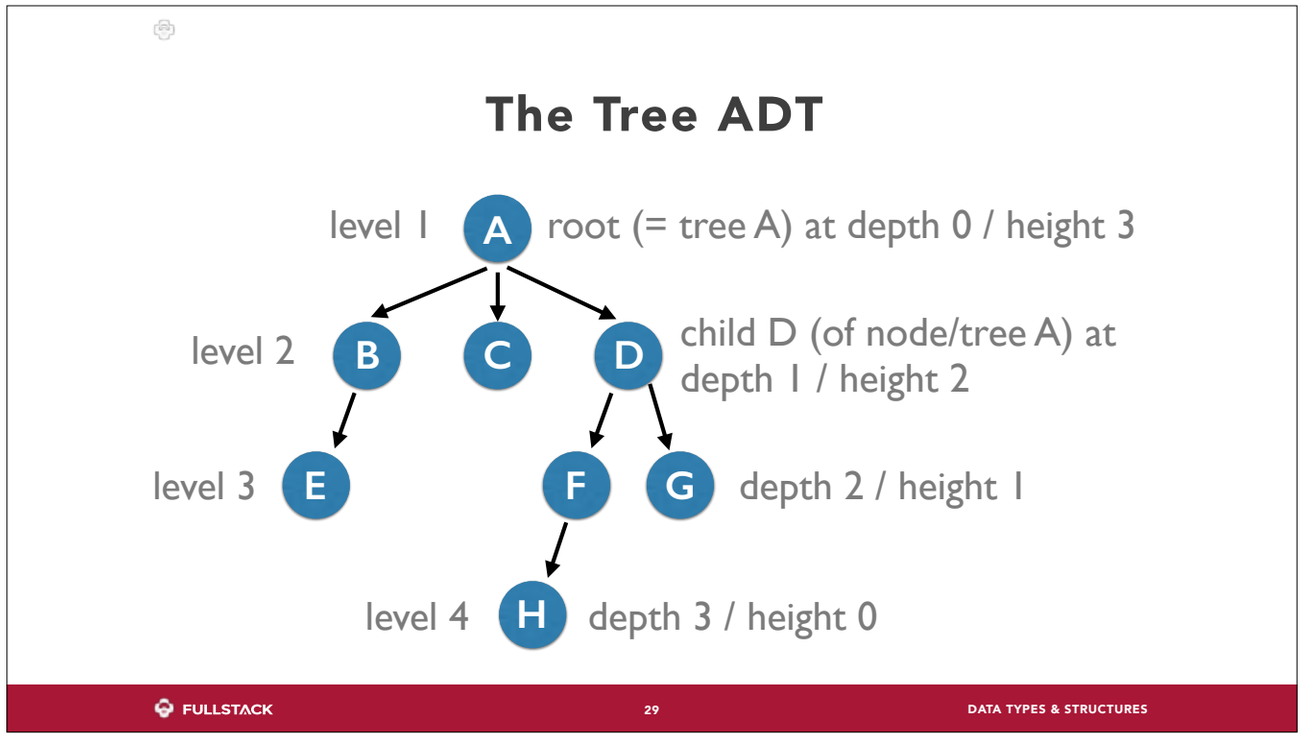
…think of trees with the root at the top and the leaves at the bottom.

The Tree ADT

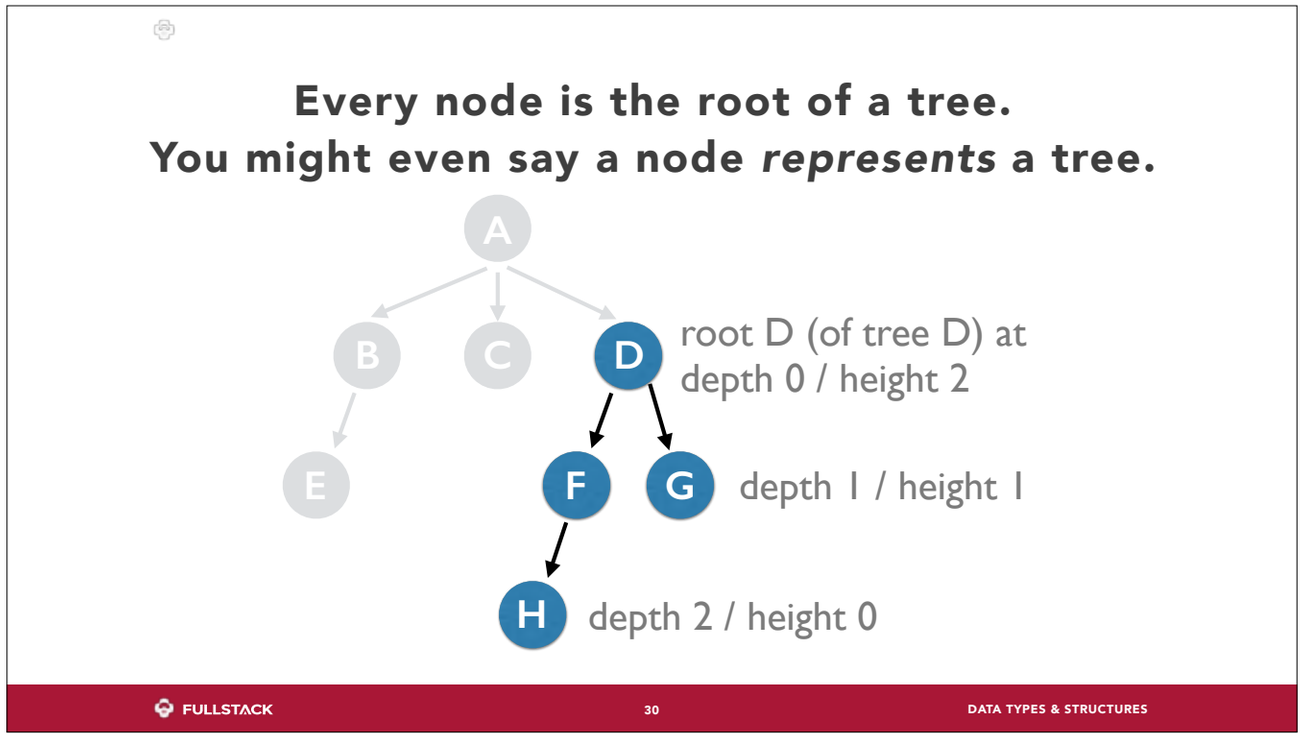


- Nodes contain value(s)
- A primary "root" node
- Children are subtrees (recursive!)
- No duplicated children (cycles); trees can *branch* but never *converge*
- Final nodes called "leaves"
- Height of tree = longest path to leaf
- Level = 1 + number of cnxns to root
- Depth = inverse of height

In the tree ADT, you can add children to the tree, look for elements, remove elements, etc. Usually, each parent has references to children, but it is rare that children have a reference to their parent.



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



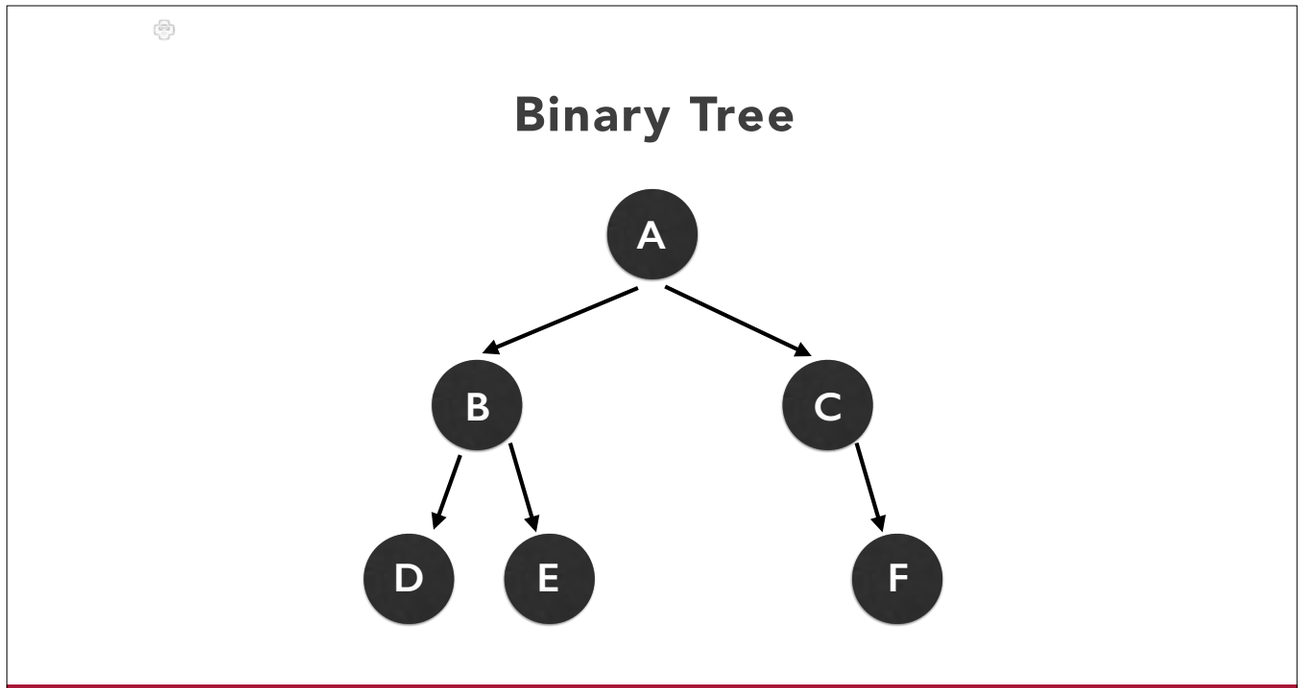
Trees are highly recursive structures! In fact one often considers "tree" and "node" as synonymous because every node is the root of a subtree. Side note: the height of a node (distance to farthest leaf) is invariant, but the depth is relative to which node you select as the root.

"Degenerate" trees are still trees

A tree of one node

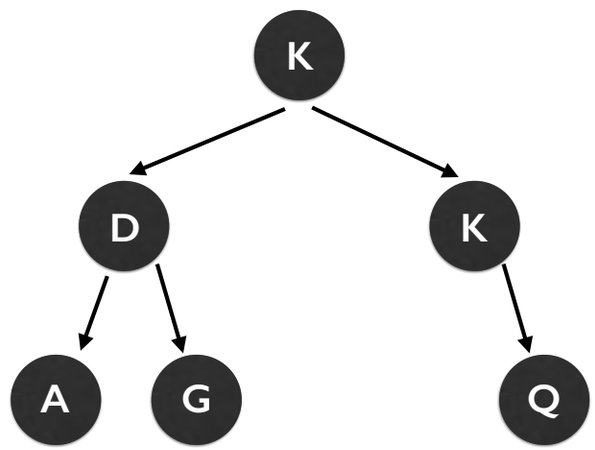
A tree of three nodes

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

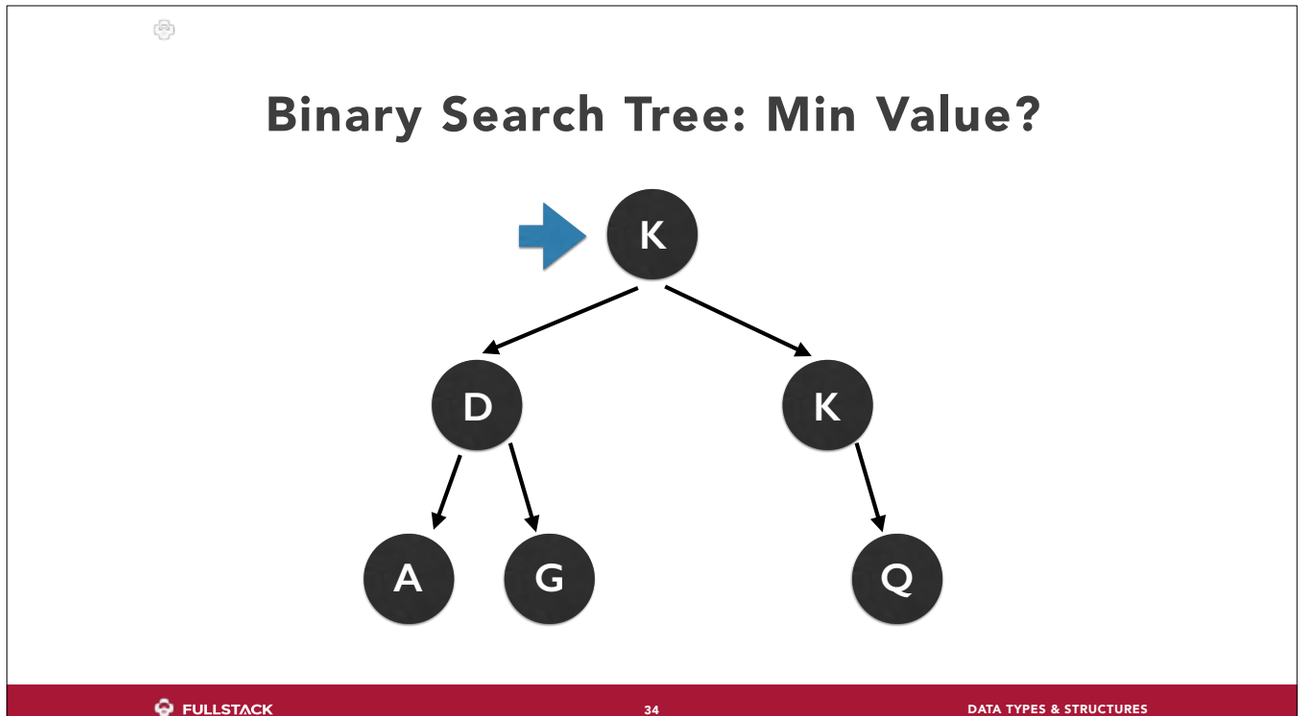


Binary trees are those for which every node has at most two children, typically called the *left* and *right* child with respect to its parent.

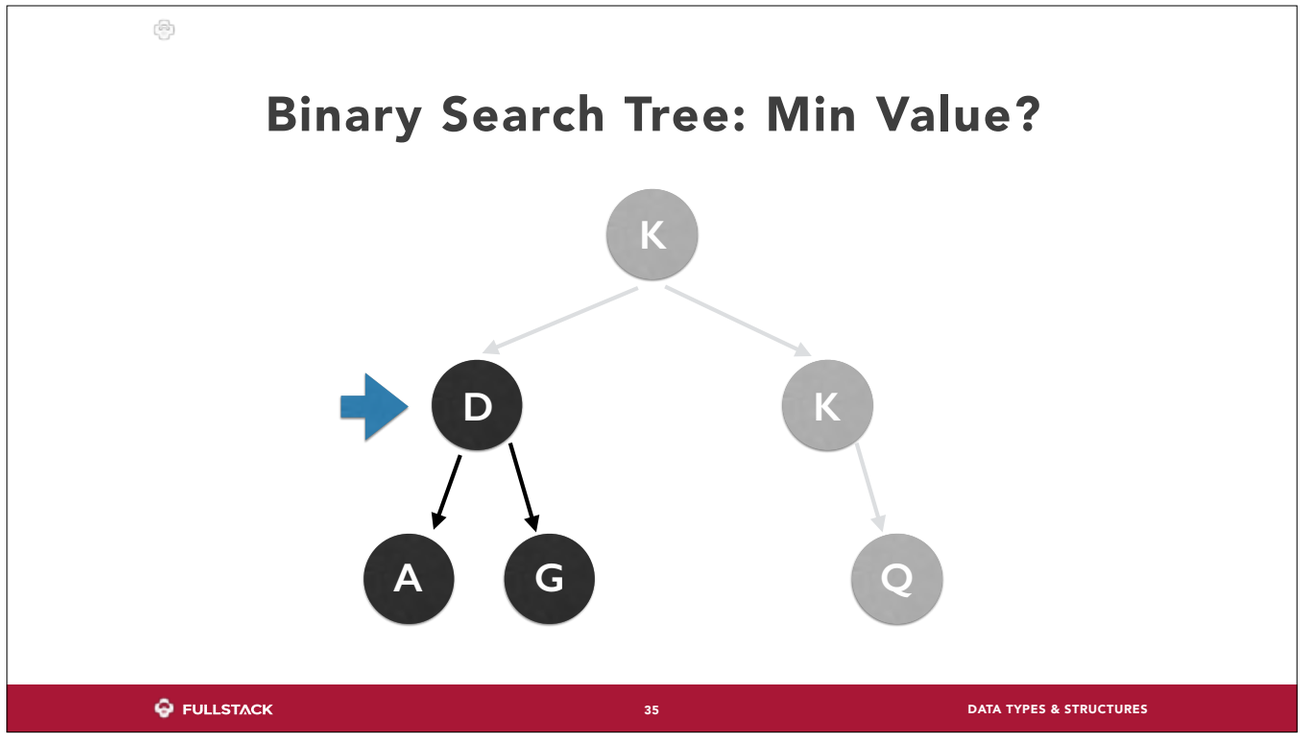
Binary *Search* Tree



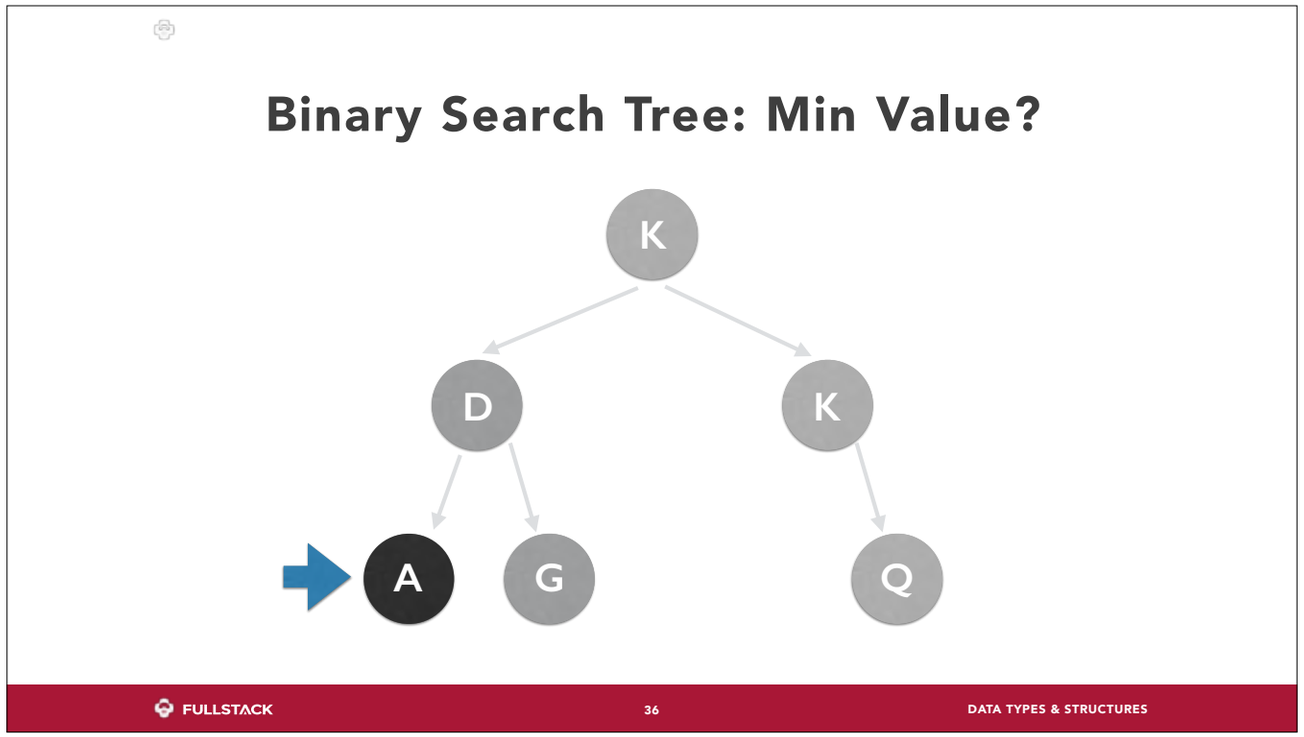
A binary SEARCH tree (BST) is one for which every node satisfies an *ordering*, i.e. all left descendants < node < all right descendants. Ties (equal values) can either go left or right so long as one consistent rule is chosen for the whole tree. Why is ordering useful?



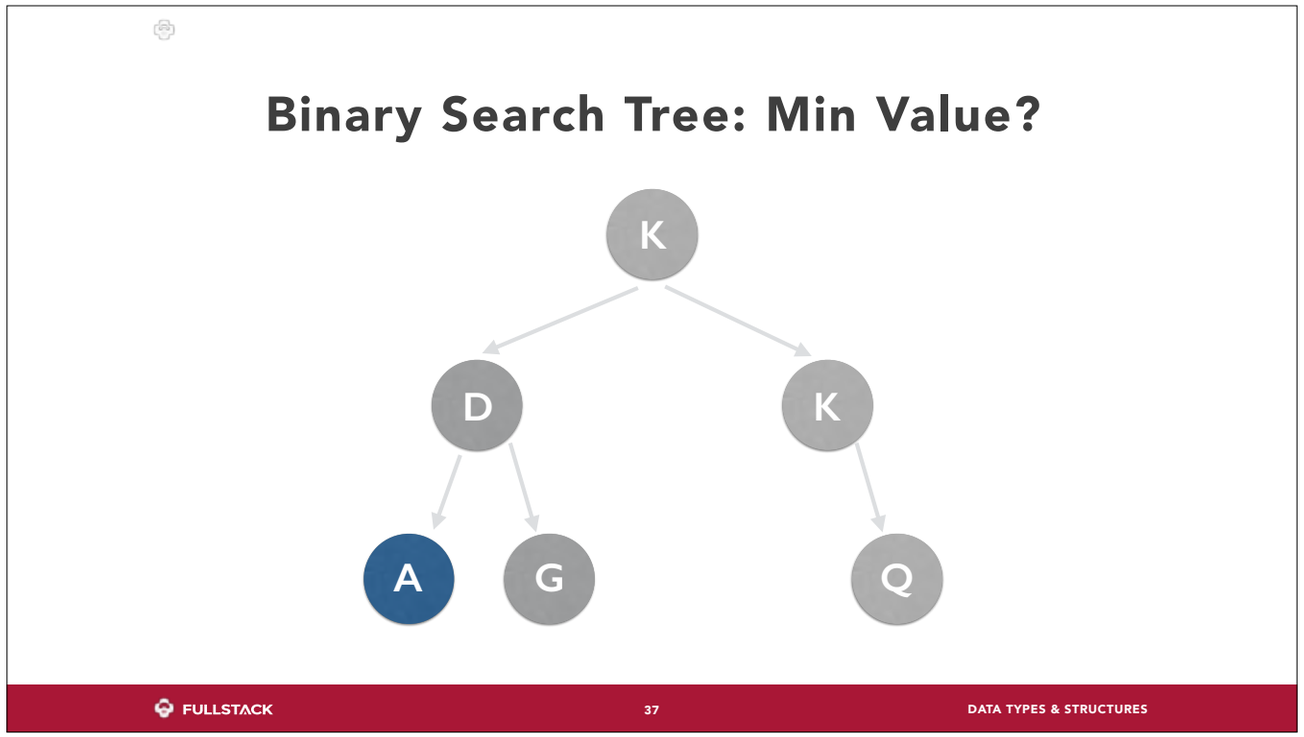
Let's see a BST in action, by SEARCHing for the minimum (leftmost) value. With a tree, we always start at the root (usually because that's the node we have a reference to).



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

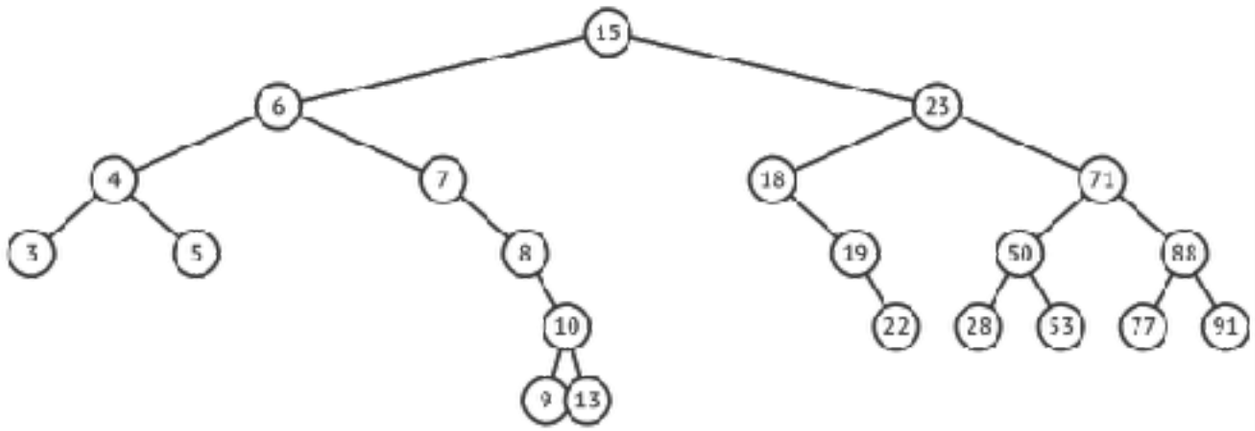


Again we move to the left child, again eliminating half of the remaining tree. We are eliminating nodes pretty quickly!

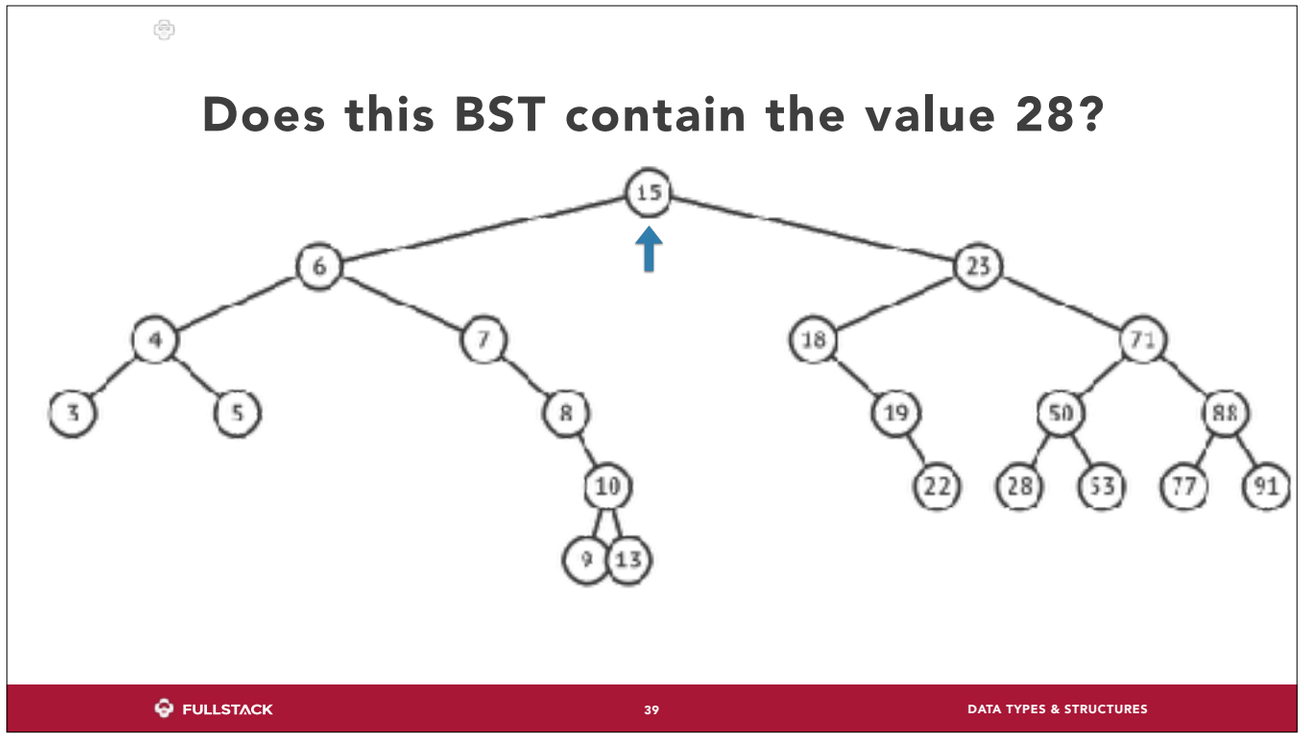


Since this node has no left child, we know it must be the minimum. We just found the minimum value in a tree of five nodes using only two jumps. This is the terrific power of BSTs — in a tree of N nodes, it only takes $\log_2(n)$ checks to find a particular value.

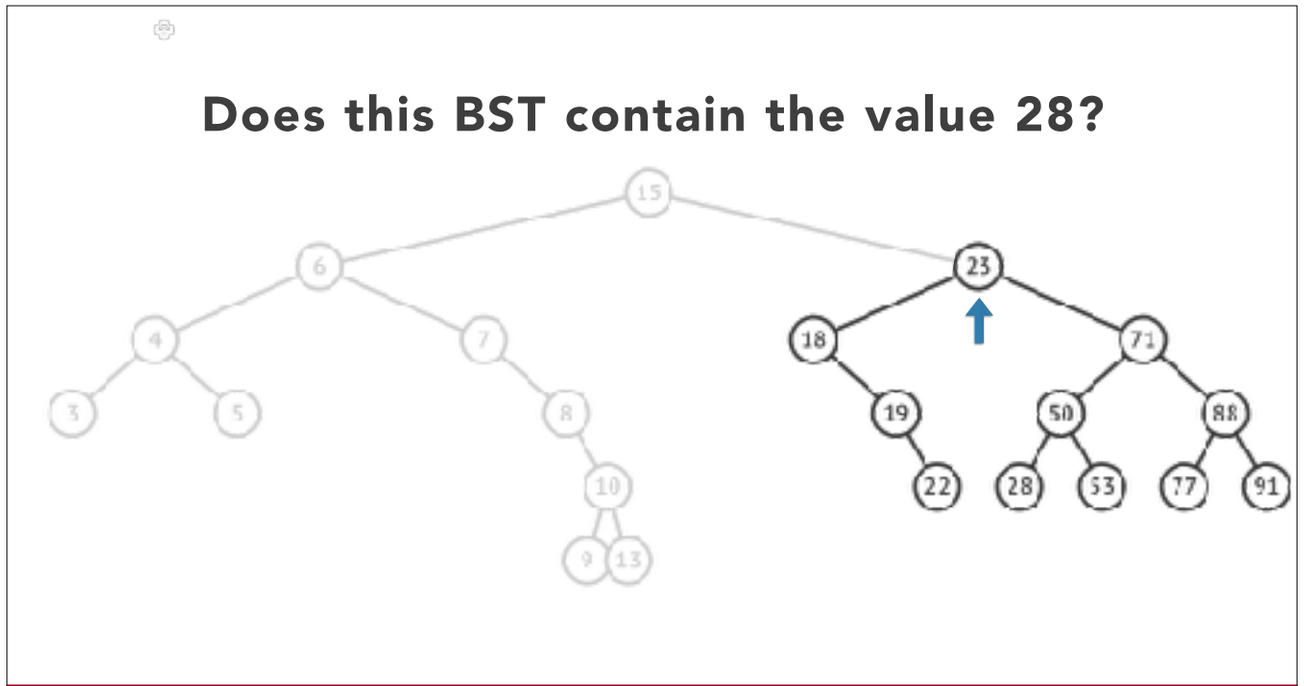
Does this BST contain the value 28?



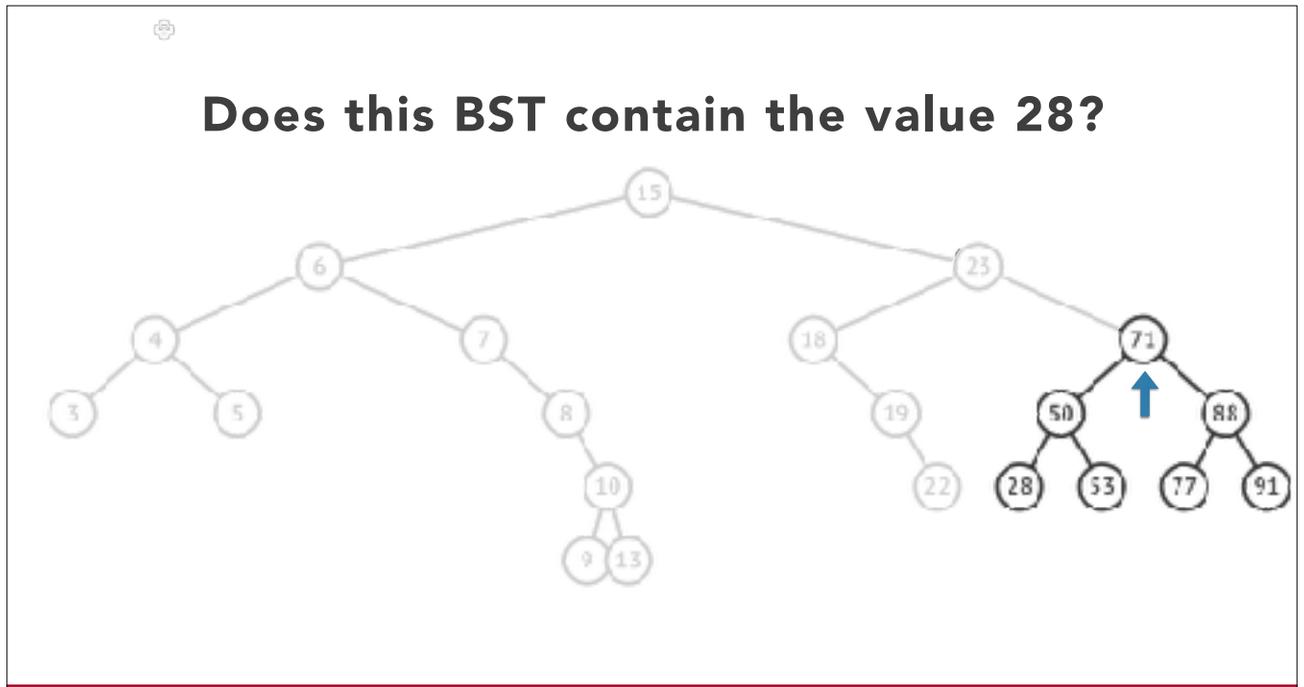
Here is a tree with 23 nodes. Does it contain the value 28? (Yes.) How many jumps did we need to determine that? (just 4).



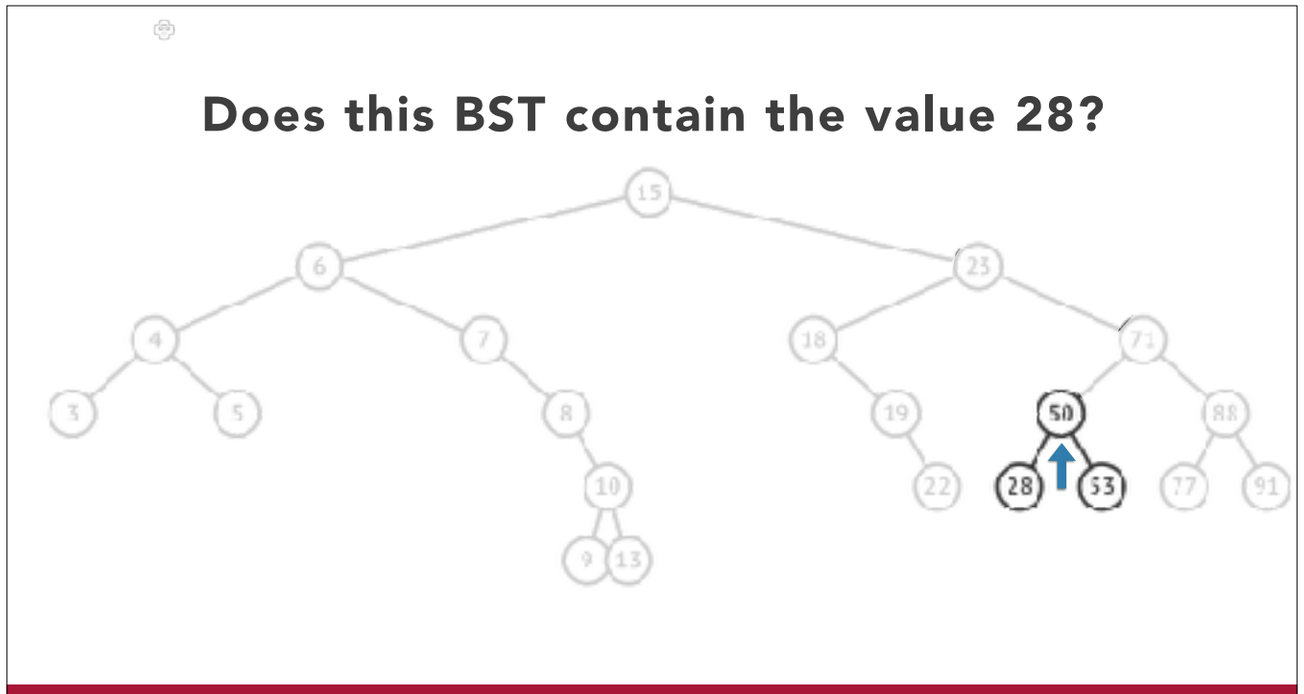
Here is a tree with 23 nodes. Does it contain the value 28? (Yes.) How many jumps did we need to determine that? (just 4).



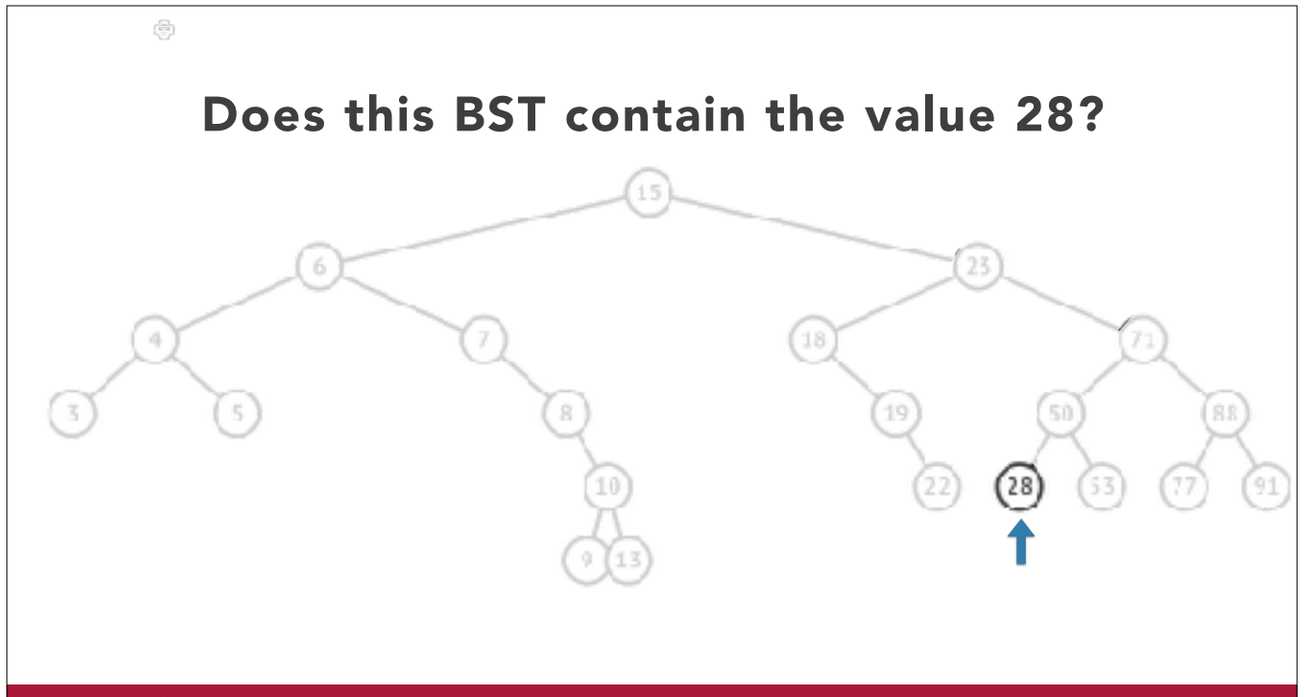
Here is a tree with 23 nodes. Does it contain the value 28? (Yes.) How many jumps did we need to determine that? (just 4).



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



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



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

BST ADT

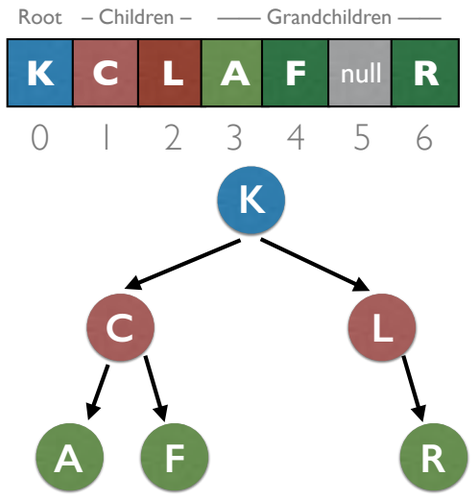
- **Root node satisfies ordering principle**
  - Left descendants < root value <= right descendants
- **Both children are BSTs (recursive definition)**
- **Operations**
  - **Insert** new values, *respecting the ordering principle*
  - **Find** existing values (takes advantage of ordering)
  - **Delete**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

How to implement this ADT?

(ask audience)

...Maybe an array (seriously)?

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

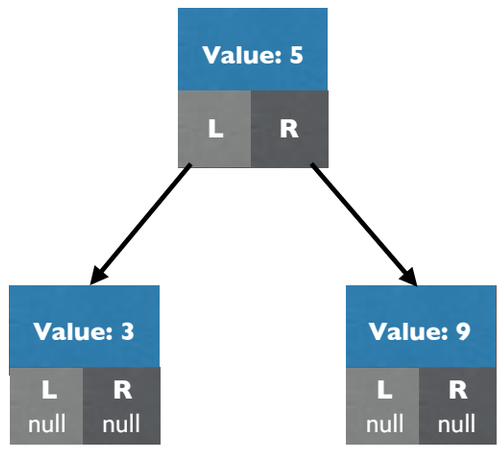


Storing a tree in an array is not as silly as it might seem. In fact, this is how a heap works, and it results in a number of surprising advantages with respect to memory, sorting, etc. You can navigate to any node in the tree using some basic math to determine which array index corresponds to which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

The Linked Tree Data Structure

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

  …the Linked Tree DS.



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

Tree traversal

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

Traversal: visiting every node

- Breadth-first search (level by level)

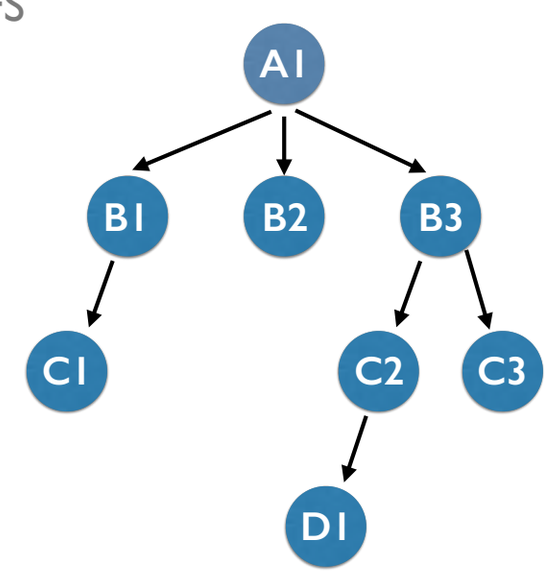
- Depth-first search (branch by branch)
  - Pre-order: process **root node**, process **left subtree**, process **right subtree**
  - **In-order**: process **left subtree**, process **root node**, process **right subtree**
  - Post-order: process **left subtree**, process **right subtree**, process **root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

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

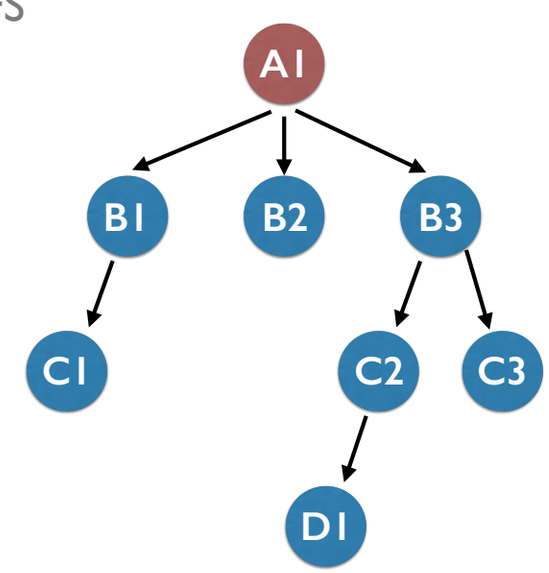
Breadth-First

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

BFS

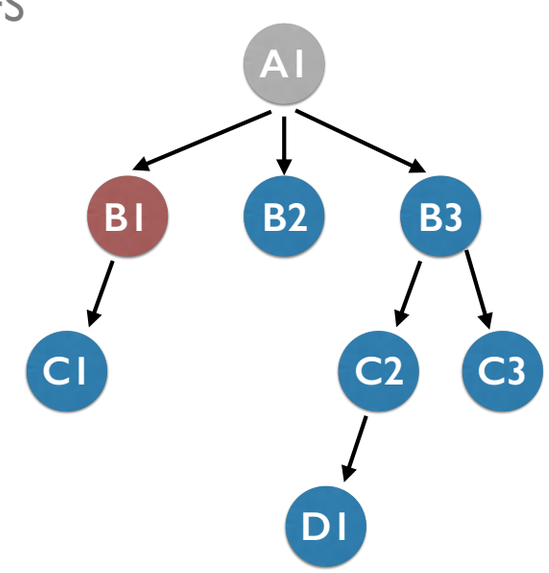


BFS



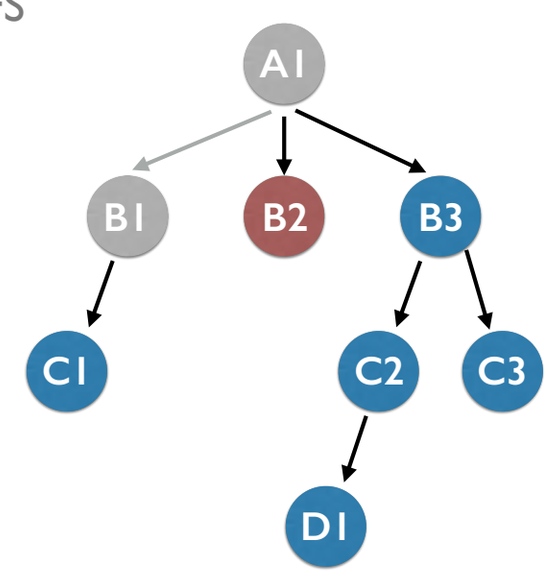
A1

BFS



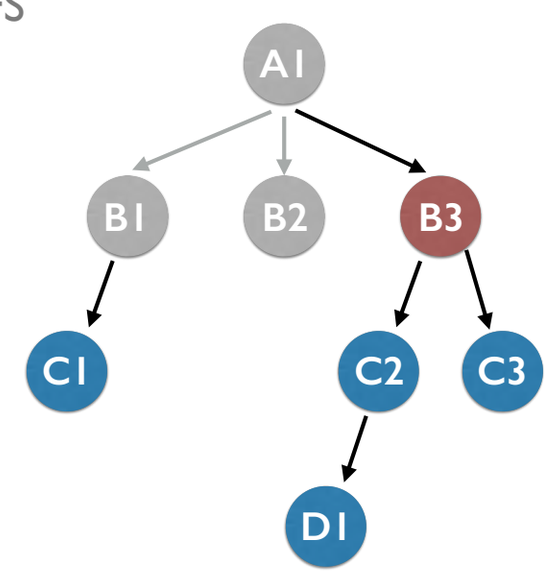
A1, B1

BFS



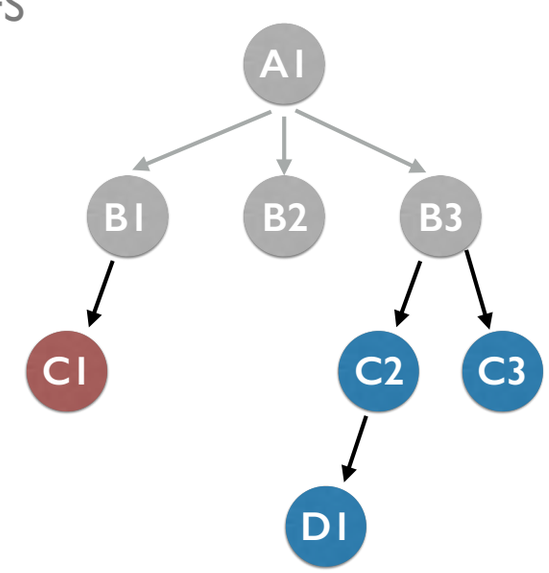
A1, B1, B2

BFS



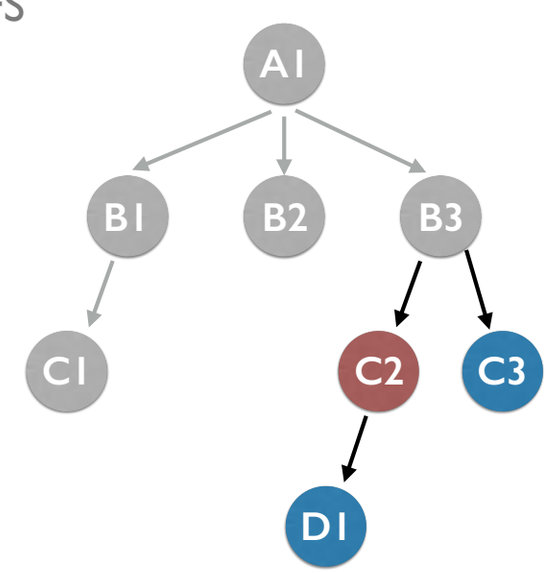
A1, B1, B2, B3

BFS



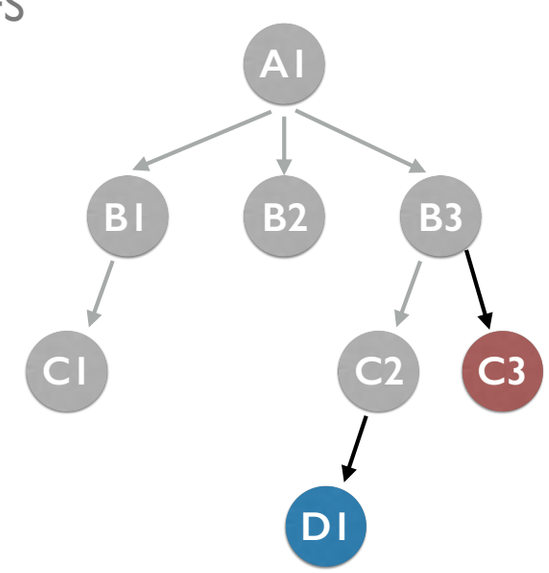
A1, B1, B2, B3, C1

BFS



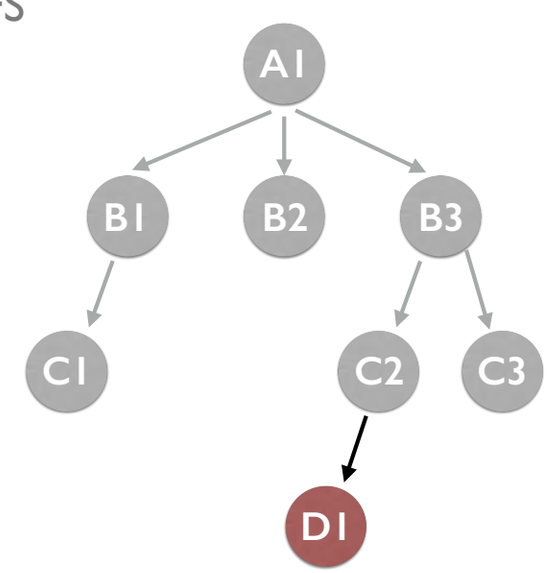
A1, B1, B2, B3, C1, C2

BFS



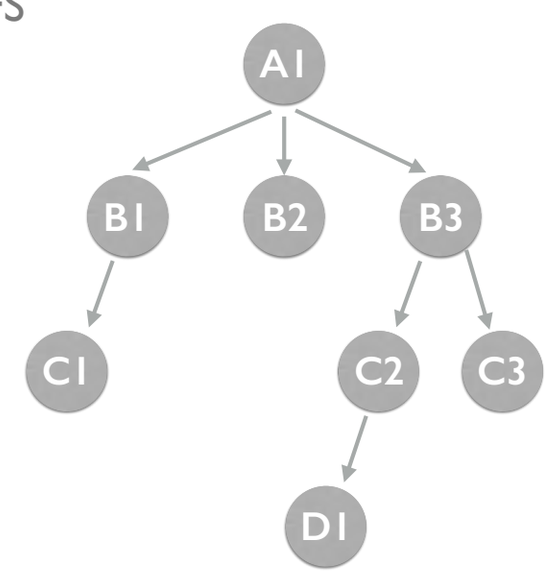
A1, B1, B2, B3, C1, C2, C3

BFS



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

BFS



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

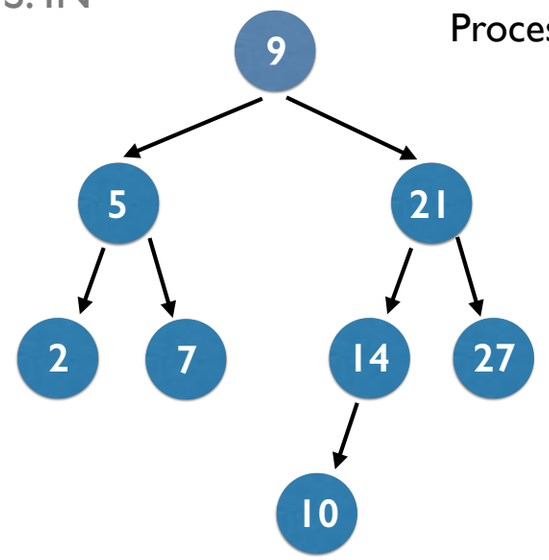
- Looks easy… but with a tree data structure it actually requires an elegant trick to do correctly.
- No full solution here, but a BIG hint: you'll need a queue!

Depth First: In-Order

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

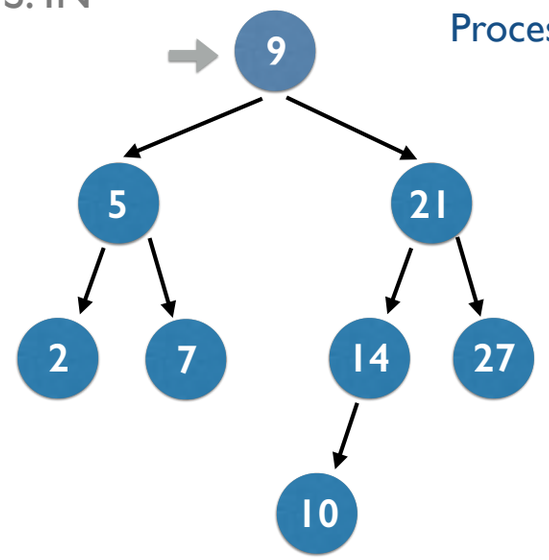
DFS: IN

Process left · Process root · Process right



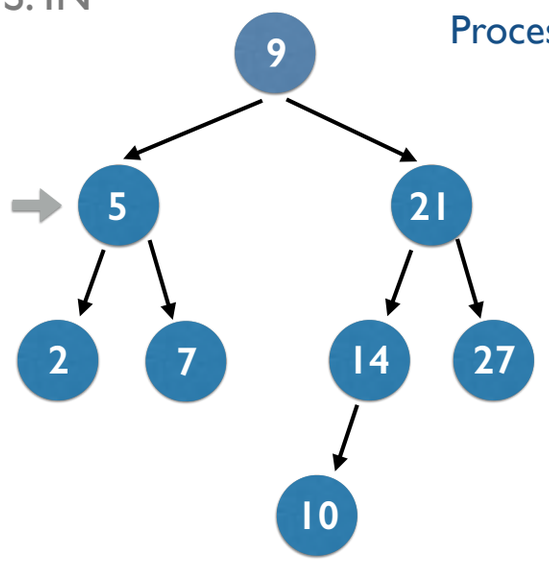
DFS: IN

Process left · Process root · Process right



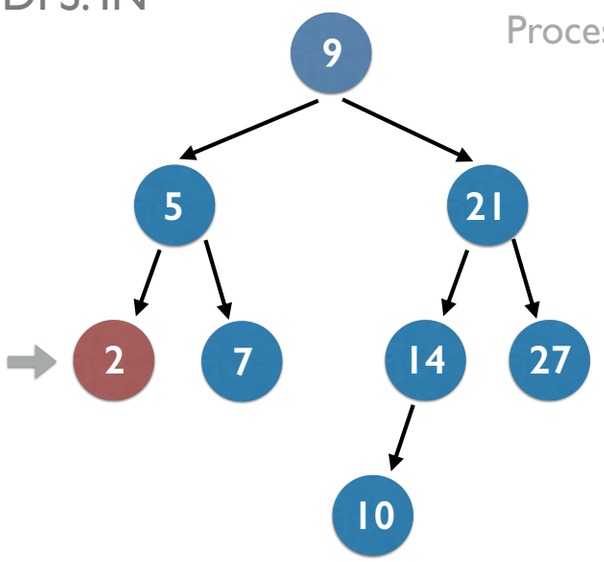
DFS: IN

Process left · Process root · Process right



DFS: IN

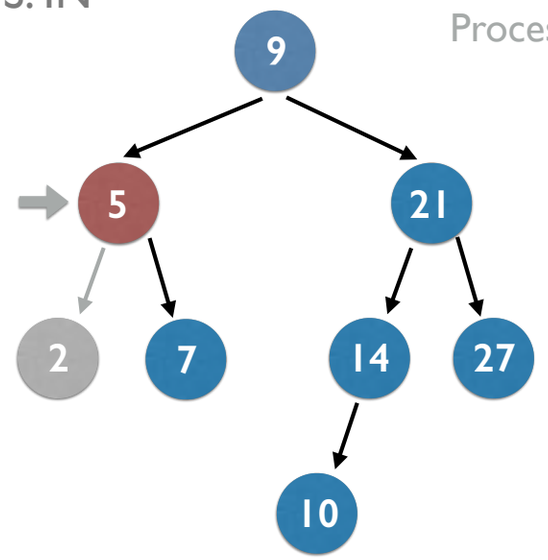
Process left · Process root · Process right



2

DFS: IN

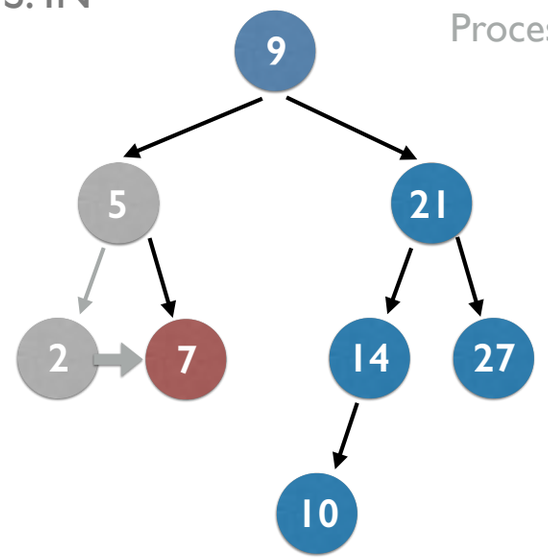
Process left · Process root · Process right



2, 5

DFS: IN

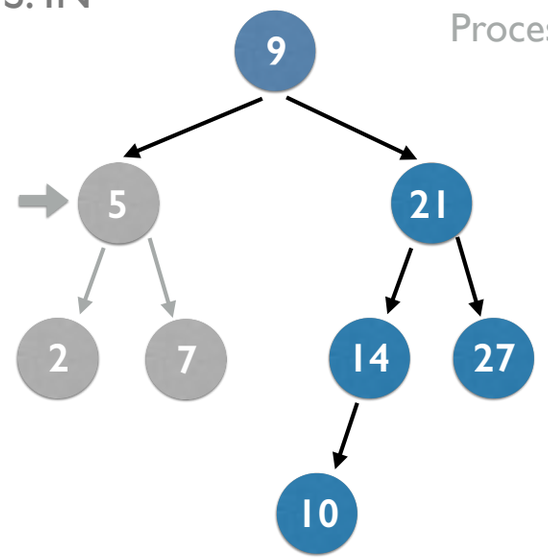
Process left · Process root · Process right



2, 5, 7

DFS: IN

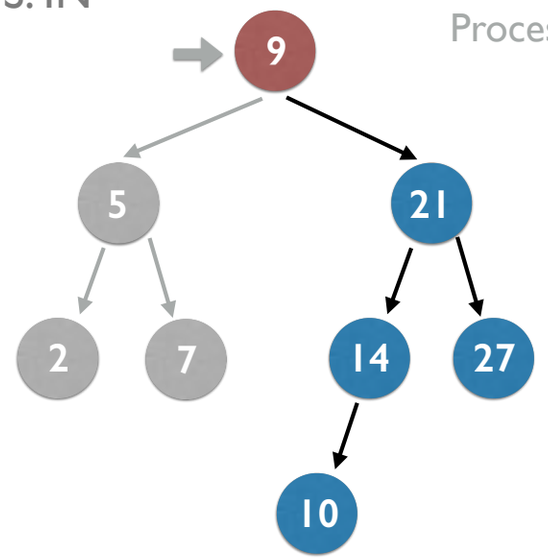
Process left · Process root · Process right



2, 5, 7

DFS: IN

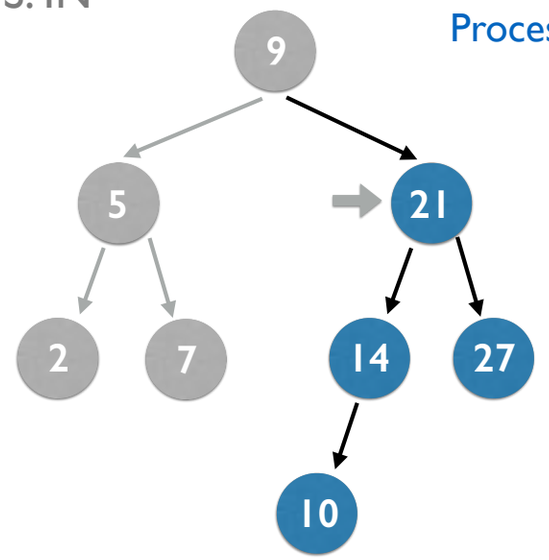
Process left · Process root · Process right



2, 5, 7, 9

DFS: IN

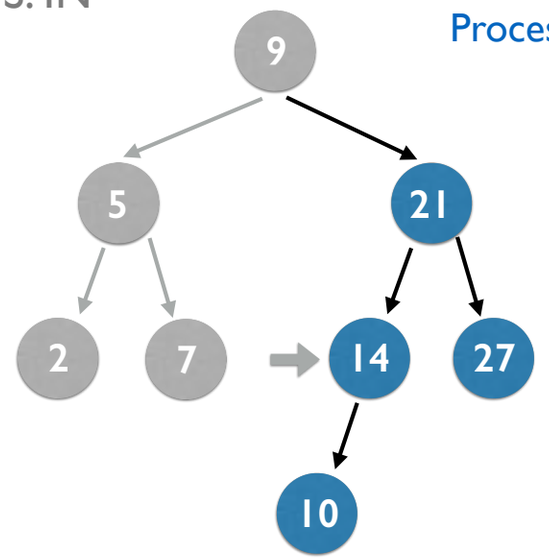
Process left · Process root · Process right



2, 5, 7, 9

DFS: IN

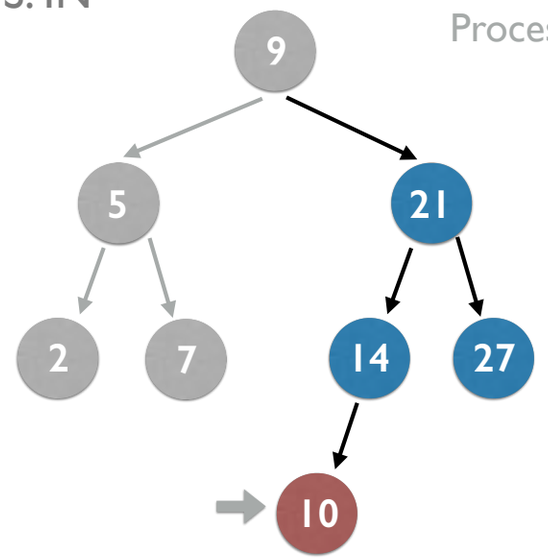
Process left · Process root · Process right



2, 5, 7, 9

DFS: IN

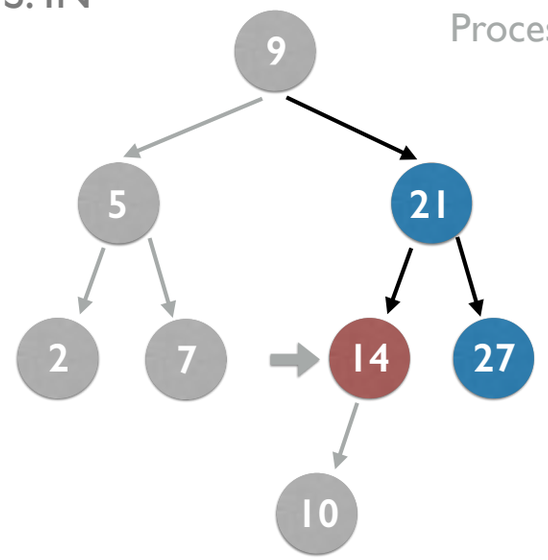
Process left · Process root · Process right



2, 5, 7, 9, 10

DFS: IN

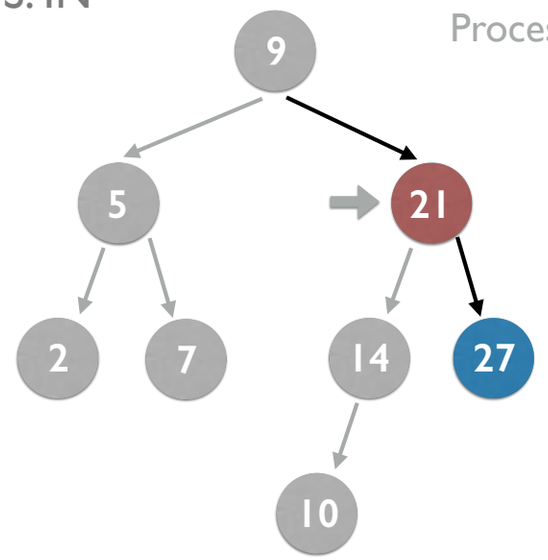
Process left · Process root · Process right



2, 5, 7, 9, 10, 14

DFS: IN

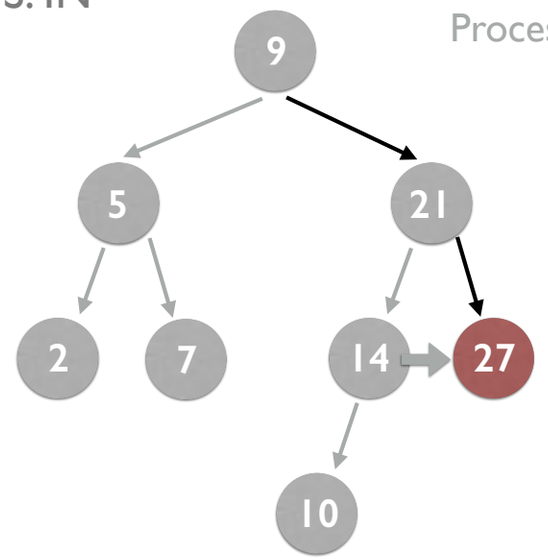
Process left · Process root · Process right



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

DFS: IN

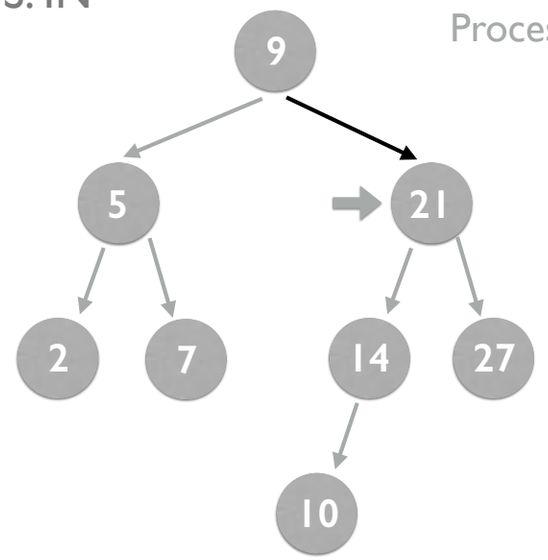
Process left · Process root · Process right



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

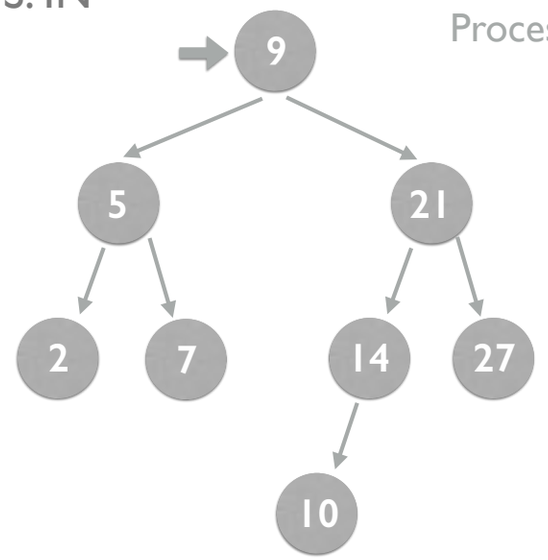
DFS: IN

Process left · Process root · Process right



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

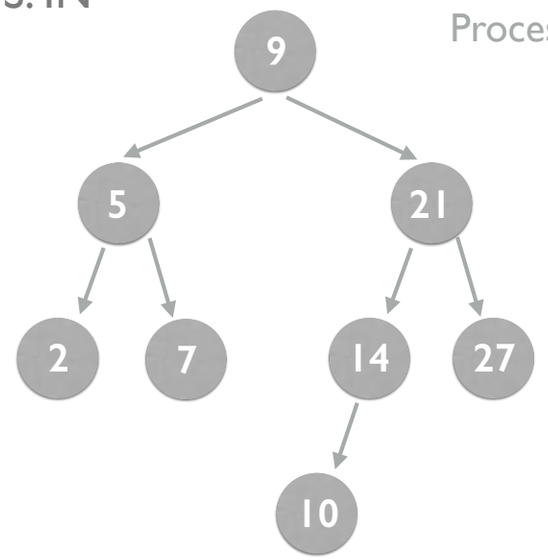
DFS: IN



Process left · Process root · Process right

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

DFS: IN



Process left · Process root · Process right

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

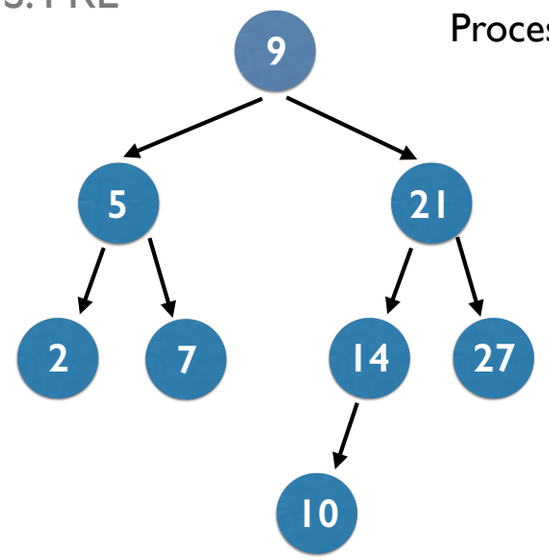
- In-order because respects ordering of nodes — nodes are processed smallest to largest value (leftmost to rightmost).
- The most generally useful DFS strategy for BSTs.

Depth First: Pre-Order

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

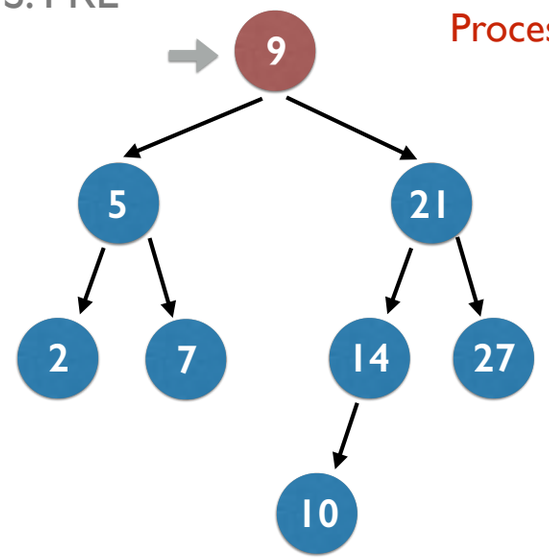
DFS: PRE

Process root · Process left · Process right



< ANIMATION BEGINS ON CLICK >

DFS: PRE

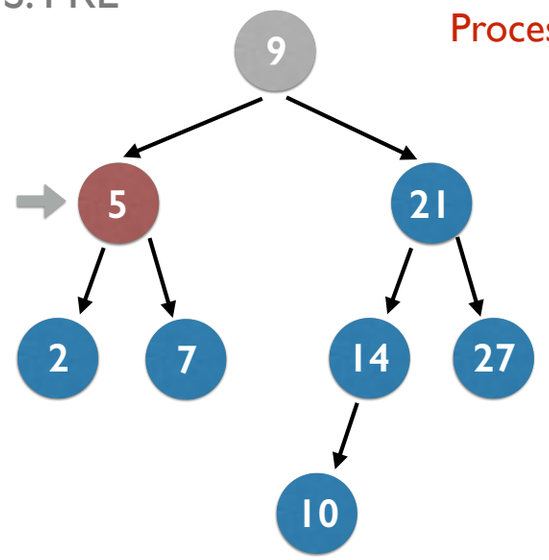


Process root · Process left · Process right

9

DFS: PRE

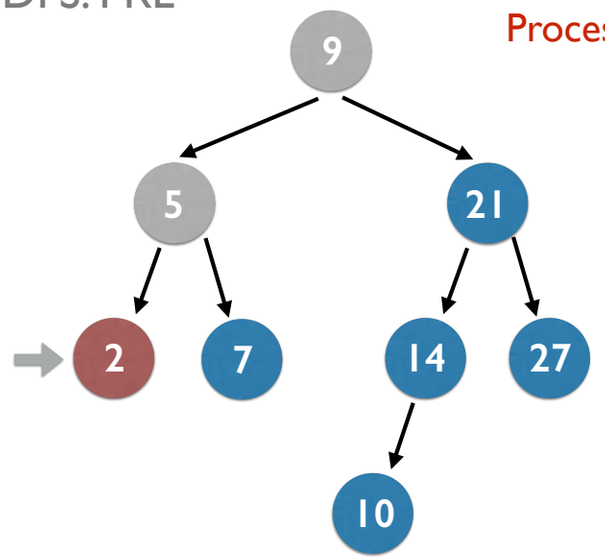
Process root · Process left · Process right



9, 5

DFS: PRE

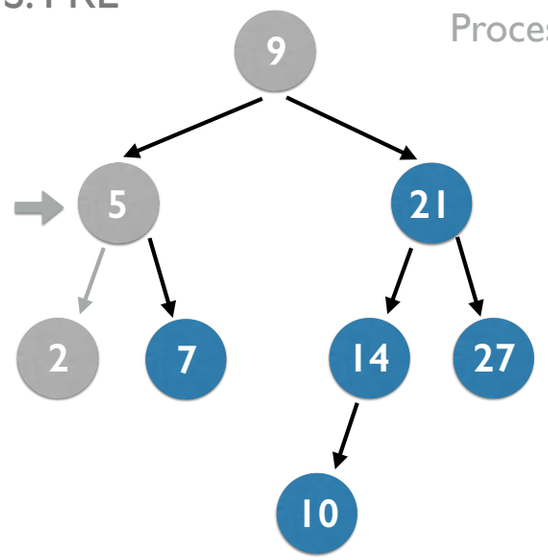
Process root · Process left · Process right



9, 5, 2

DFS: PRE

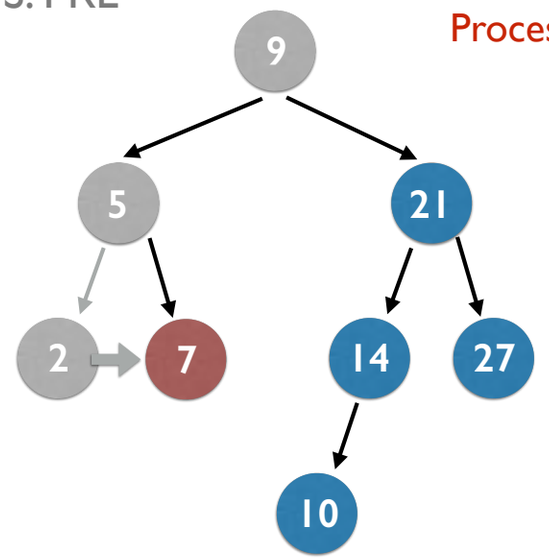
Process root · Process left · Process right



9, 5, 2

DFS: PRE

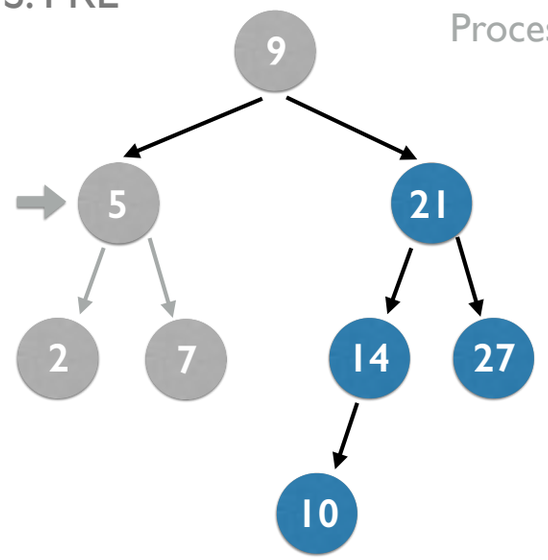
Process root · Process left · Process right



9, 5, 2, 7

DFS: PRE

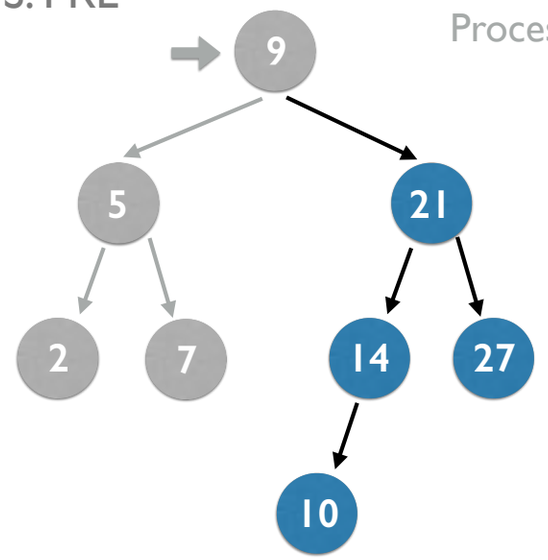
Process root · Process left · Process right



9, 5, 2, 7

DFS: PRE

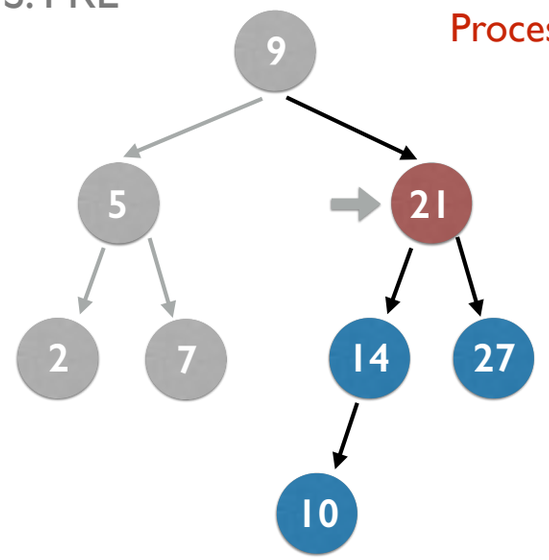
Process root · Process left · Process right



9, 5, 2, 7

DFS: PRE

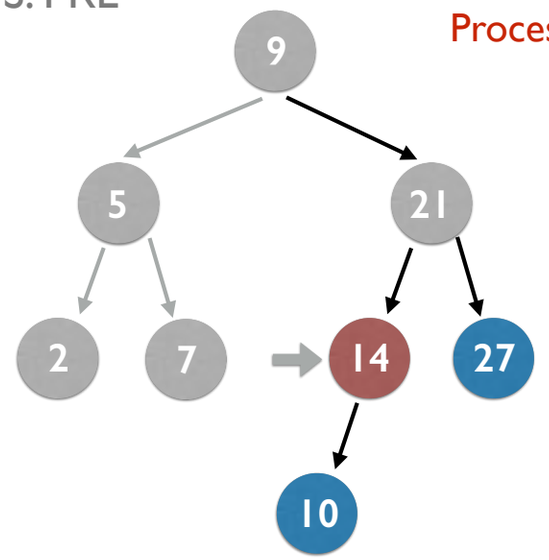
Process root · Process left · Process right



9, 5, 2, 7, 21

DFS: PRE

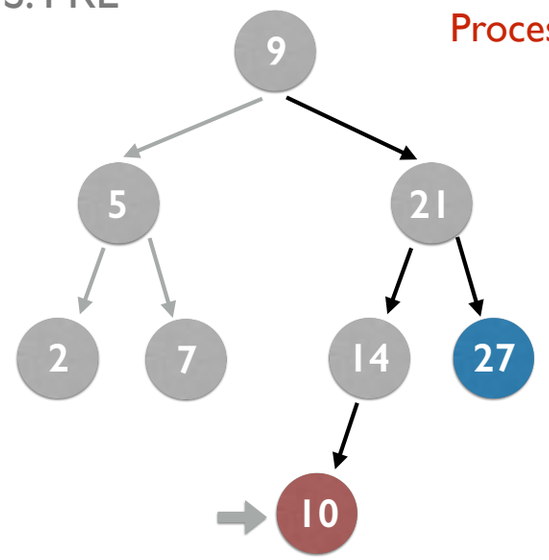
Process root · Process left · Process right



9, 5, 2, 7, 21, 14

DFS: PRE

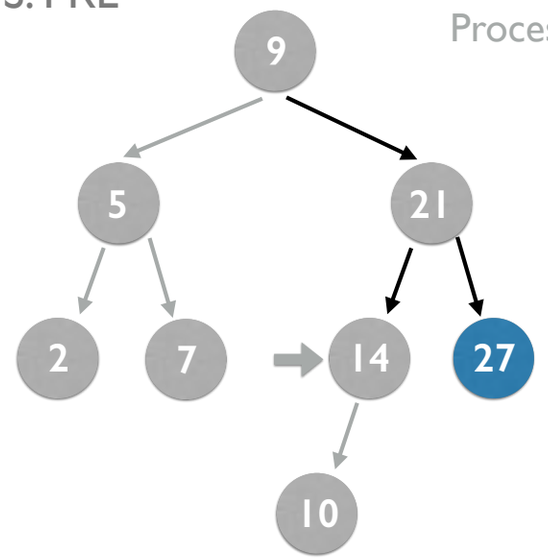
Process root · Process left · Process right



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

DFS: PRE

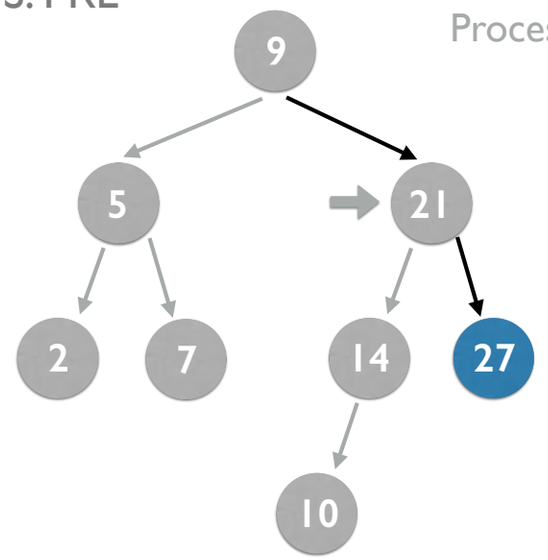
Process root · Process left · Process right



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

DFS: PRE

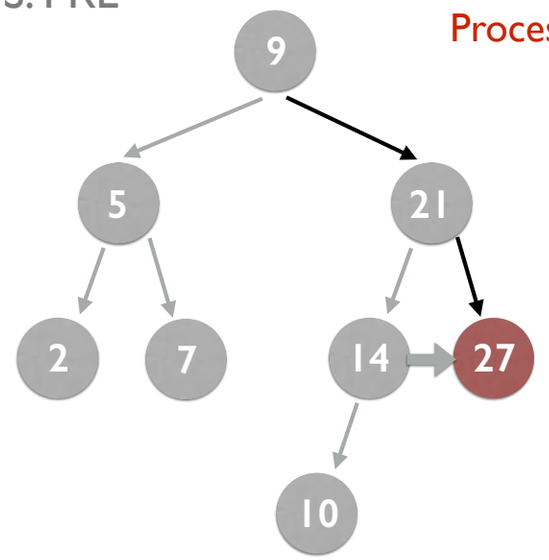
Process root · Process left · Process right



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

DFS: PRE

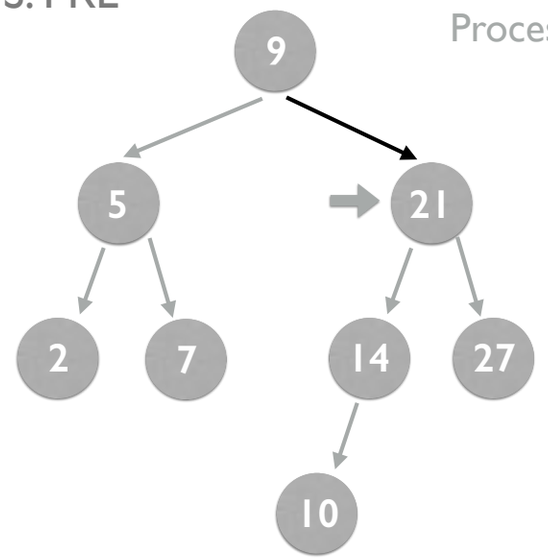
Process root · Process left · Process right



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

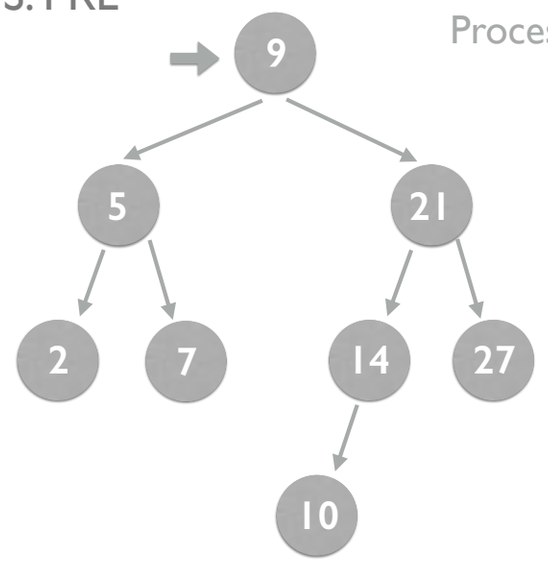
DFS: PRE

Process root · Process left · Process right



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

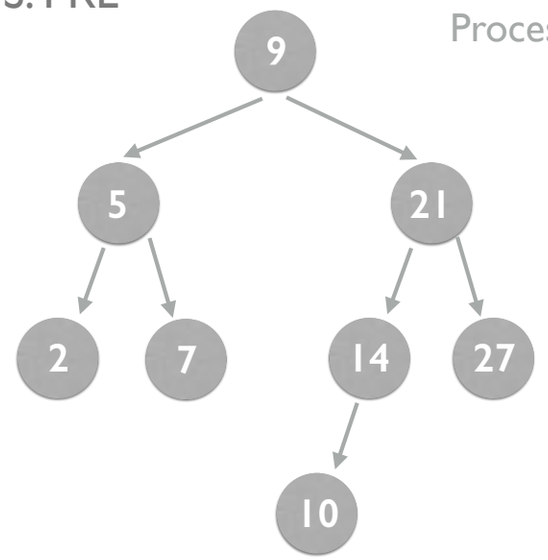
DFS: PRE



Process root · Process left · Process right

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

DFS: PRE



Process root · Process left · Process right

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

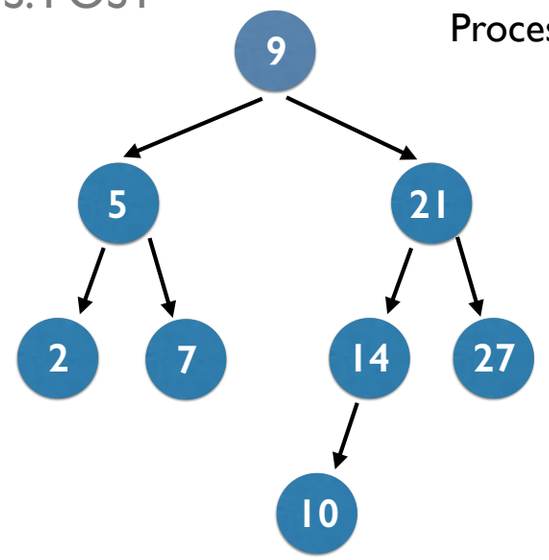
- **Output seems random, but actually this has one notable use case. If you create a BST by inserting these values in this order, you get a copy of the original tree. However, the same is true for breadth-first.**

Depth First: Post-Order

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

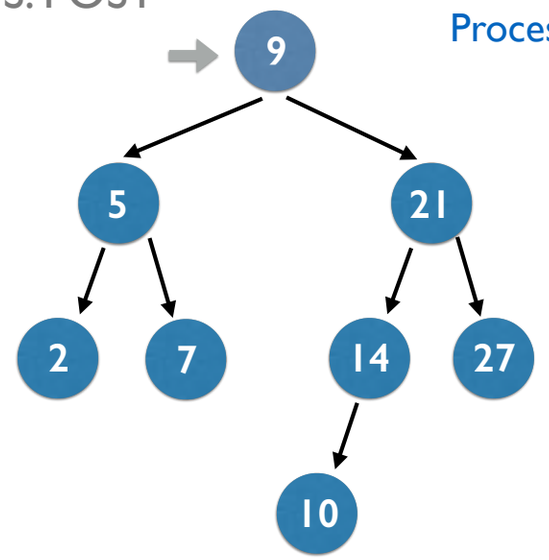
DFS: POST

Process left · Process right · Process root



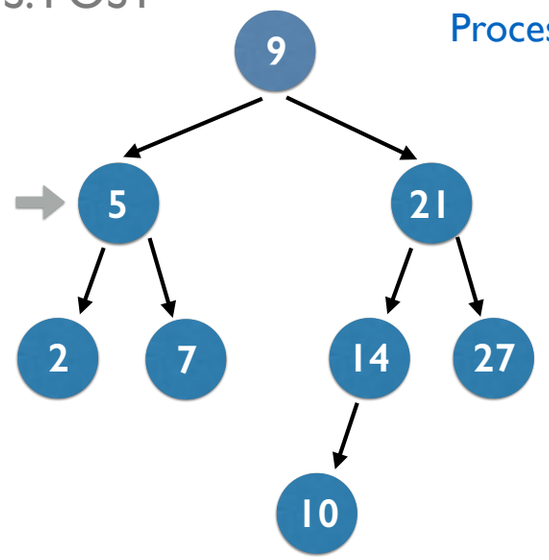
DFS: POST

Process left · Process right · Process root



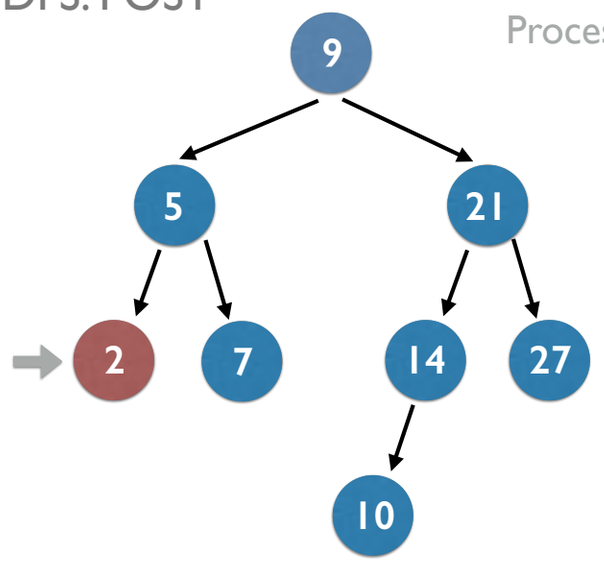
DFS: POST

Process left · Process right · Process root



DFS: POST

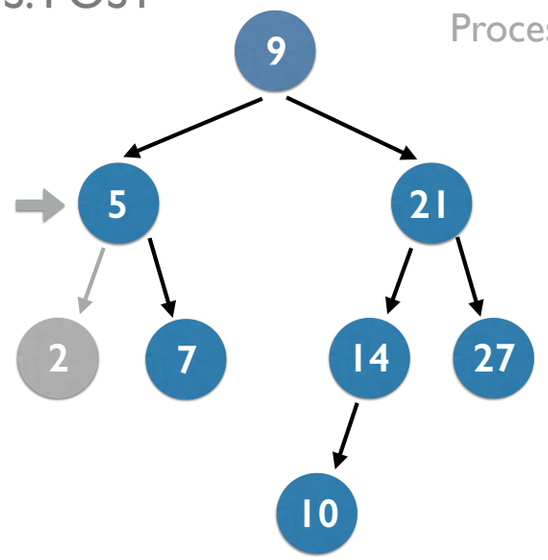
Process left · Process right · Process root



2

DFS: POST

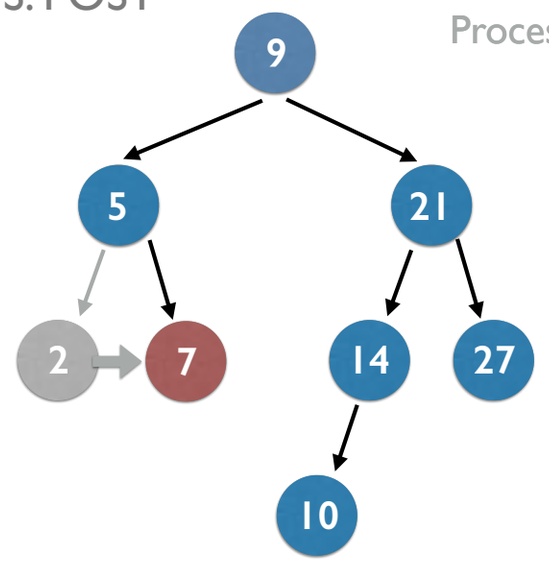
Process left · Process right · Process root



2

DFS: POST

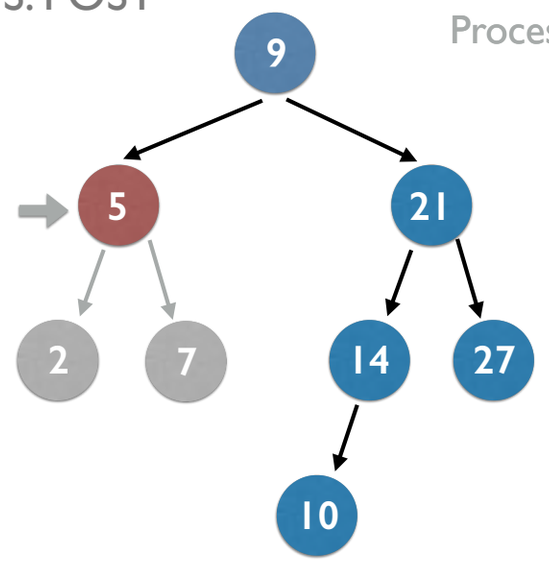
Process left · Process right · Process root



2, 7

DFS: POST

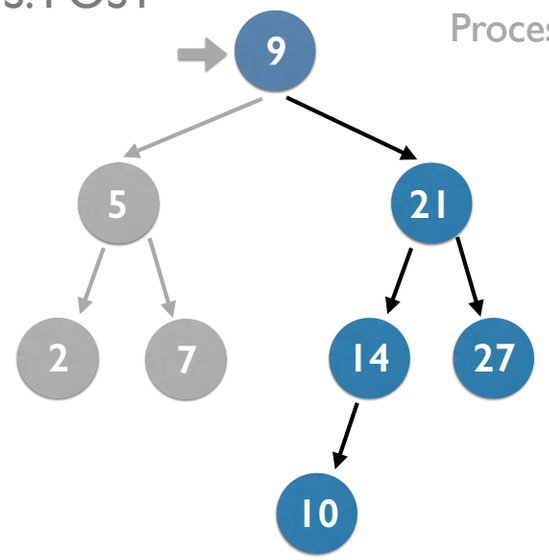
Process left · Process right · Process root



2, 7, 5

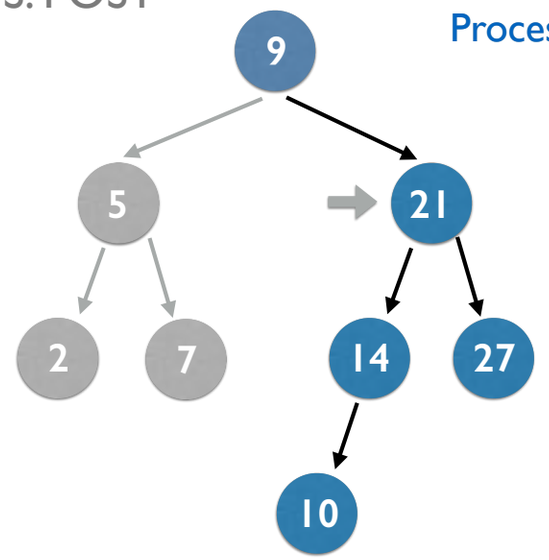
DFS: POST

Process left · Process right · Process root



2, 7, 5

DFS: POST

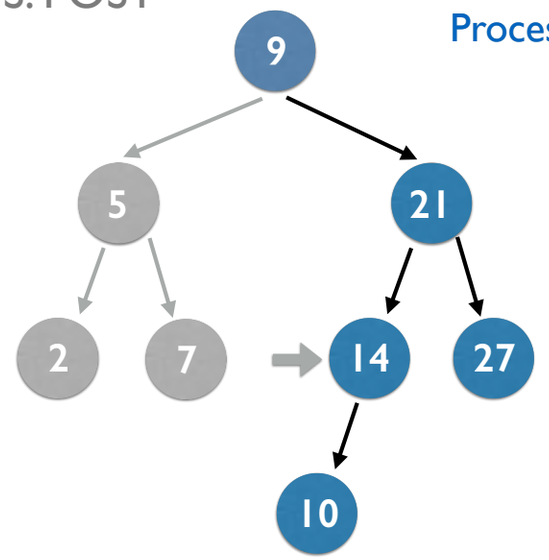


Process left · Process right · Process root

2, 7, 5

DFS: POST

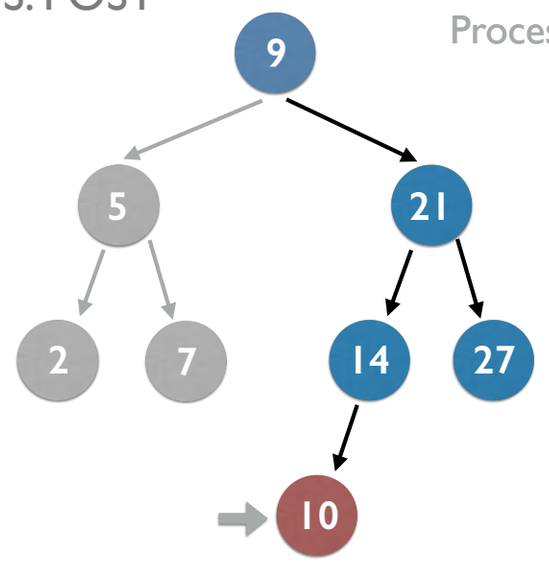
Process left · Process right · Process root



2, 7, 5

DFS: POST

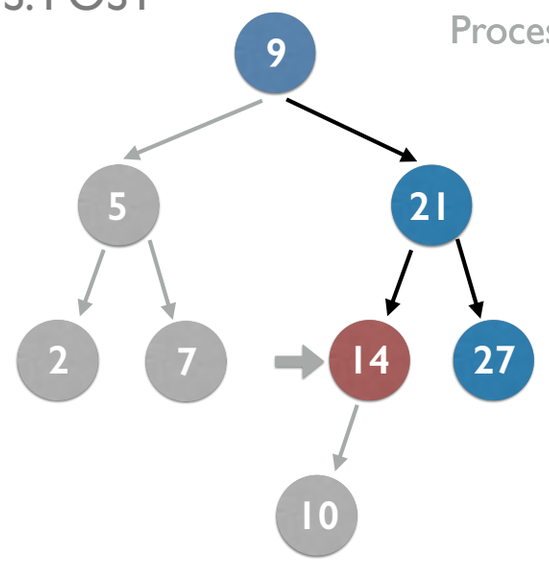
Process left · Process right · Process root



2, 7, 5, 10

DFS: POST

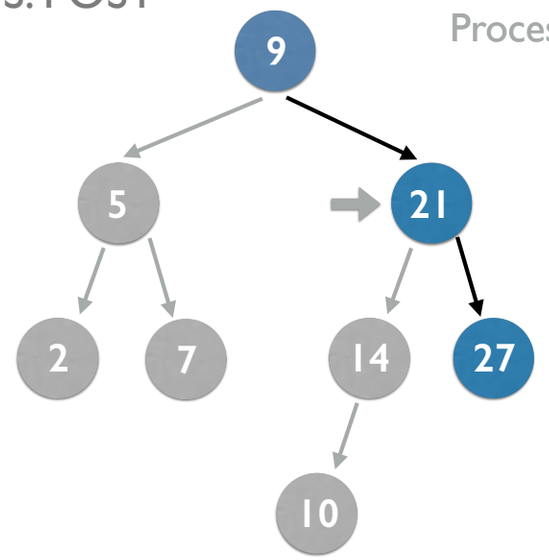
Process left · Process right · Process root



2, 7, 5, 10, 14

DFS: POST

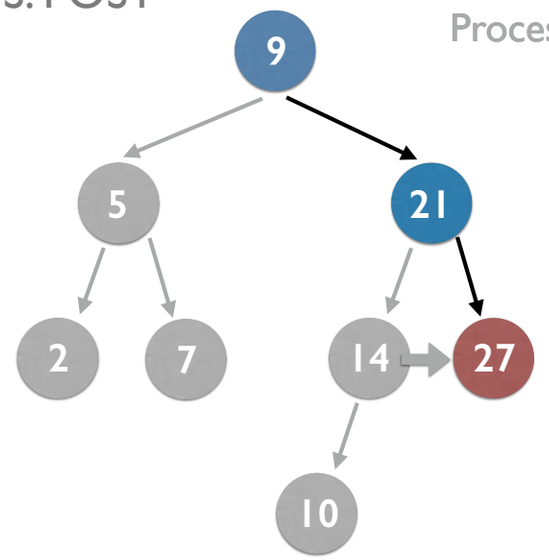
Process left · Process right · Process root



2, 7, 5, 10, 14

DFS: POST

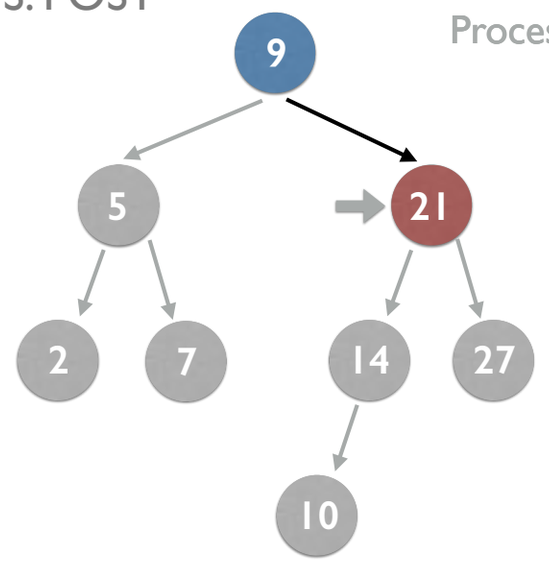
Process left · Process right · Process root



2, 7, 5, 10, 14, 27

DFS: POST

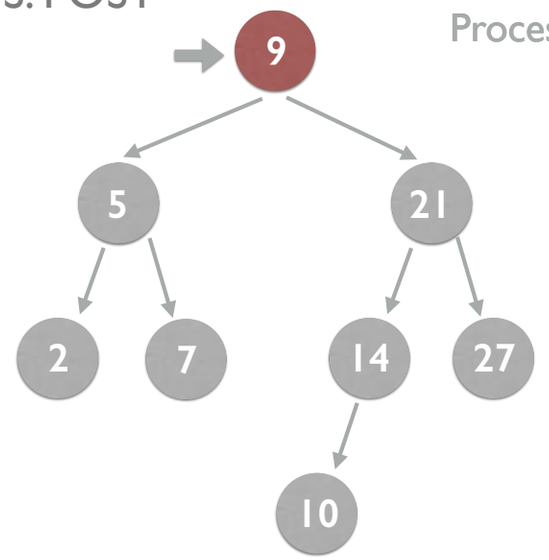
Process left · Process right · Process root



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

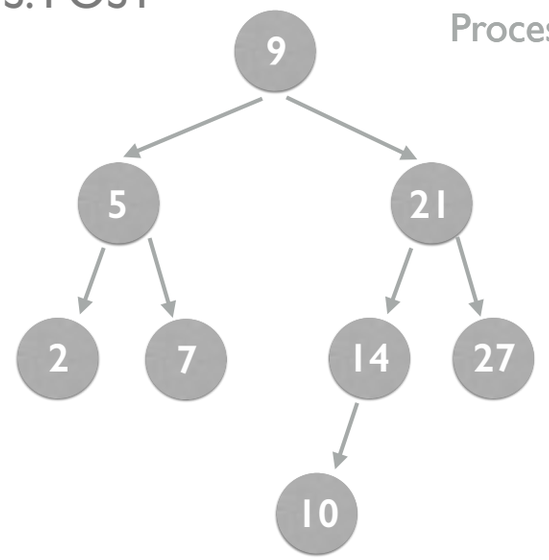
DFS: POST

Process left · Process right · Process root



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

DFS: POST



Process left · Process right · Process root

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

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

That's all for our formal data types & structures lectures. However, that should be just the beginning for you — we have only scratched the surface of this massive subject, one which can be studied for many years if you are so inclined. For example, there are *dozens* of more tree variations — and we didn't even mention Graphs, an essential topic in computer science! It is our hope that with this introduction as a basis, you will continue to independently learn more at your own pace. Check out the additional resources listed in the workshop.