CSE373: Data Structures & Algorithms

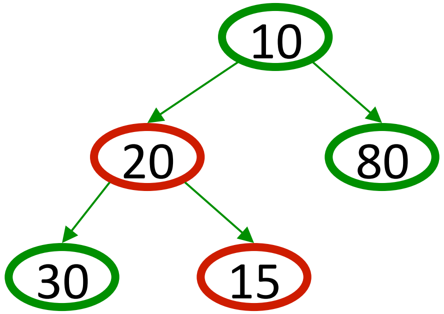
More Heaps; Dictionaries; Binary Search Trees

Riley Porter
Winter 2016

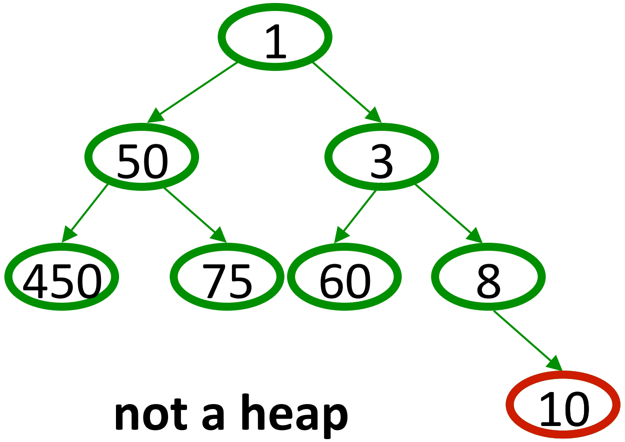
Review of last time: Heaps

Heaps follow the following two properties:

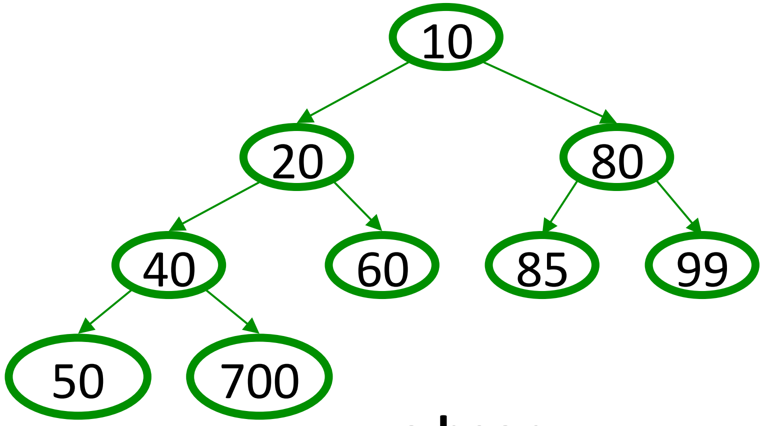
- Structure property: A *complete* binary tree
- Heap order property: The priority of the children is always a greater value than the parents (greater value means less priority / less importance)



not a heap

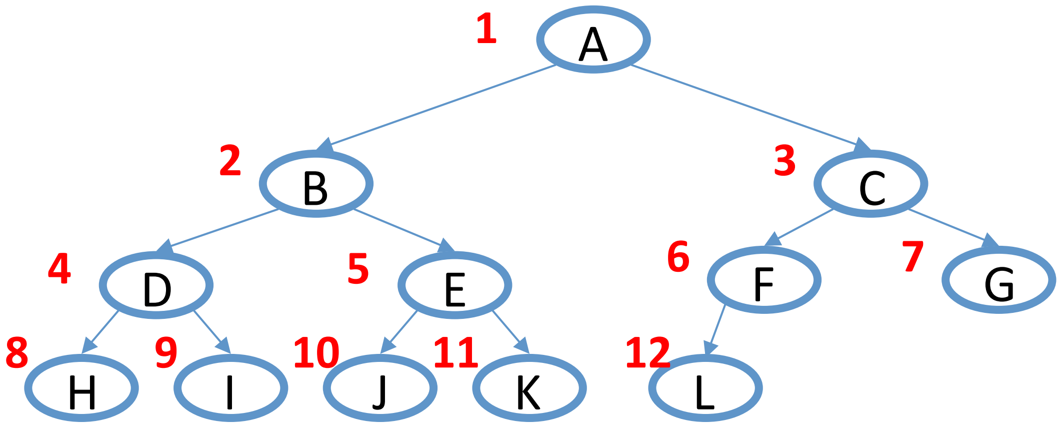


not a heap



a heap

Review: Array Representation



Starting at node `i`

left child: `i*2`
right child: `i*2+1`
parent: `i/2`

(wasting index 0 is convenient for the index arithmetic)

implicit (array) implementation:

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Review: Heap Operations

`insert:`

(1) add the new value at the next valid place in the structure

(2) (2) fix the ordering property by percolating value up to the right position

`deleteMin:`

(1) remove smallest value at root

(2) plug vacant spot at root with value from the last spot in the tree, keeping the structure valid

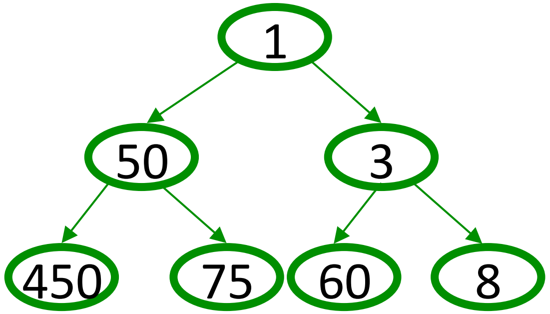
(3) fix the ordering property by percolating the value down to the right position

Review: Heap Operations Runtimes

`insert` and `deleteMin` both `O(logN)`

at worst case, the number of swaps you have to do is the height of the tree. The height of a complete tree with N nodes is logN.

Intuition:



1 Node — 2^0 Nodes

2 Nodes — 2^1 Nodes

4 Nodes — 2^2 Nodes

Build Heap

- Suppose you have *n* items to put in a new (empty) priority queue
  - Call this operation **`buildHeap`**

- *n* distinct **`insert`**s works (slowly)
  - Only choice if ADT doesn't provide **`buildHeap`** explicitly
  - *O*(*n* **`log`** *n*)

- Why would an ADT provide this unnecessary operation?
  - Convenience
  - Efficiency: an *O*(*n*) algorithm called Floyd's Method
  - Common tradeoff in ADT design: how many specialized operations

Floyd's Method

Intuition: if you have a lot of values to insert all at once, you can optimize by inserting them all and then doing a pass for swapping

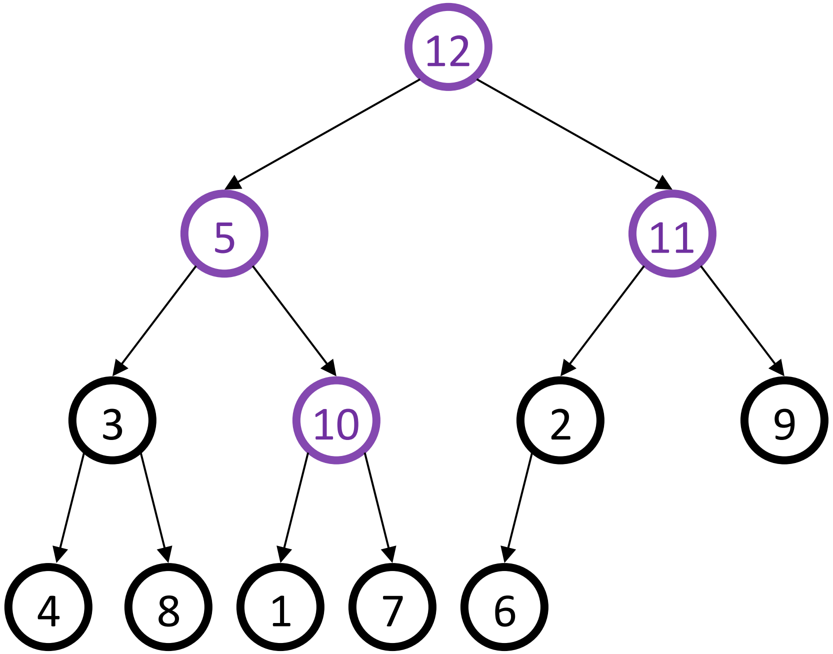
1. Put the *n* values anywhere to make a complete structural tree

2. Treat it as a heap and fix the heap-order property
   - Bottom-up: leaves are already in heap order, work up toward the root one level at a time

```
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val  = arr[i];
    hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```

Example

- Build a heap with the values:
  12, 5, 11, 3, 10, 2, 9, 4, 8, 1, 7, 6
- Stick them all in the tree to make a valid structure
- In tree form for readability. Notice:
  - Purple for node values to fix (heap-order problem)
  - Notice no leaves are purple
  - Check/fix each non-leaf bottom-up (6 steps here)

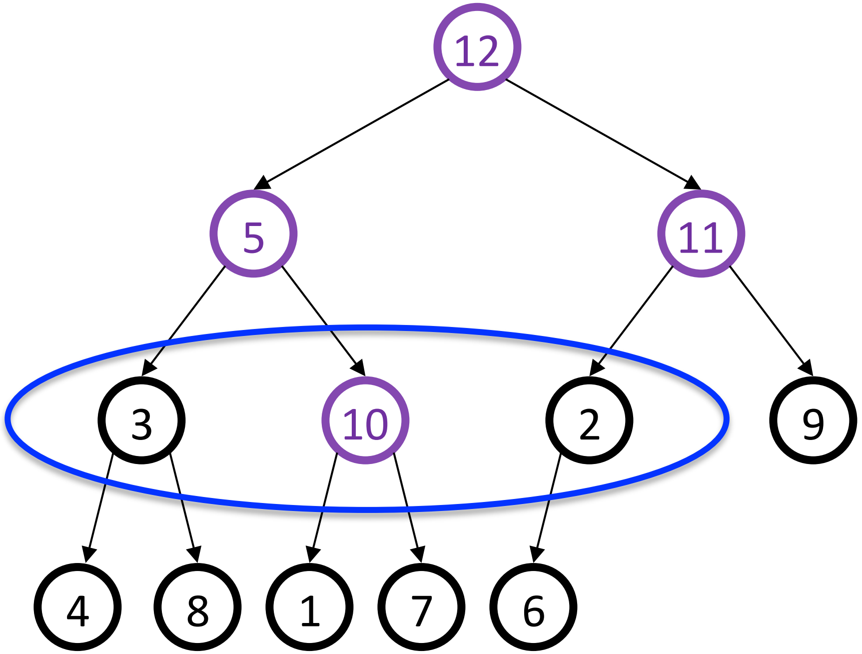


Algorithm Example

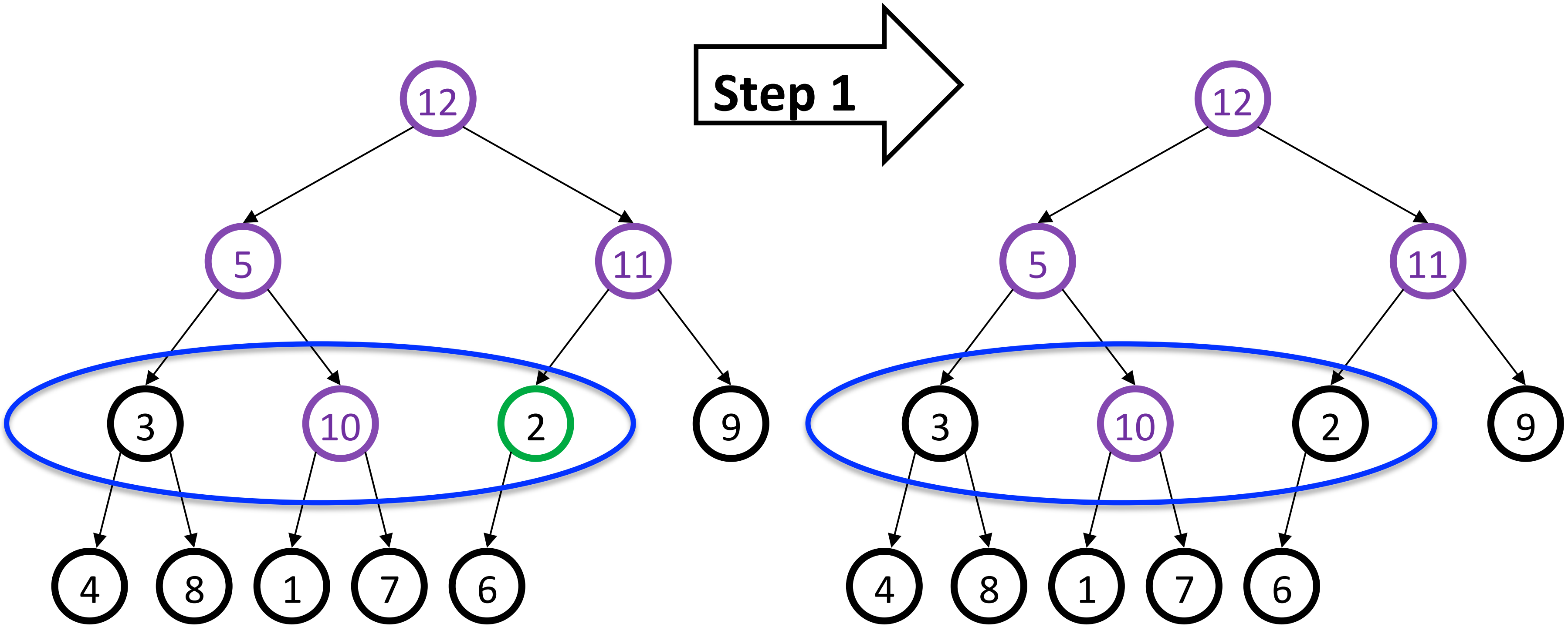
Purple shows the nodes that will need to be fixed.

We don't know which ones they are yet, so we'll traverse bottom up one level at a time and fix all the values.

Values to consider on each level circled in blue

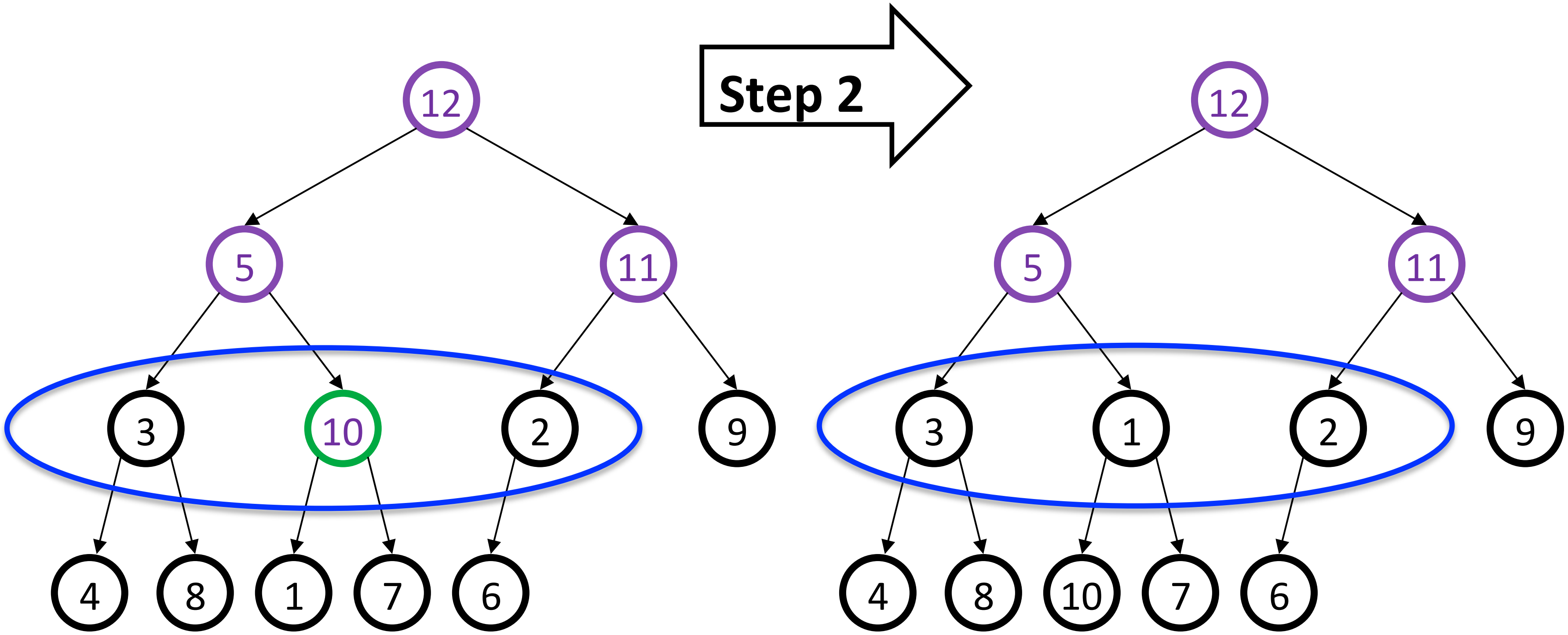


Algorithm Example



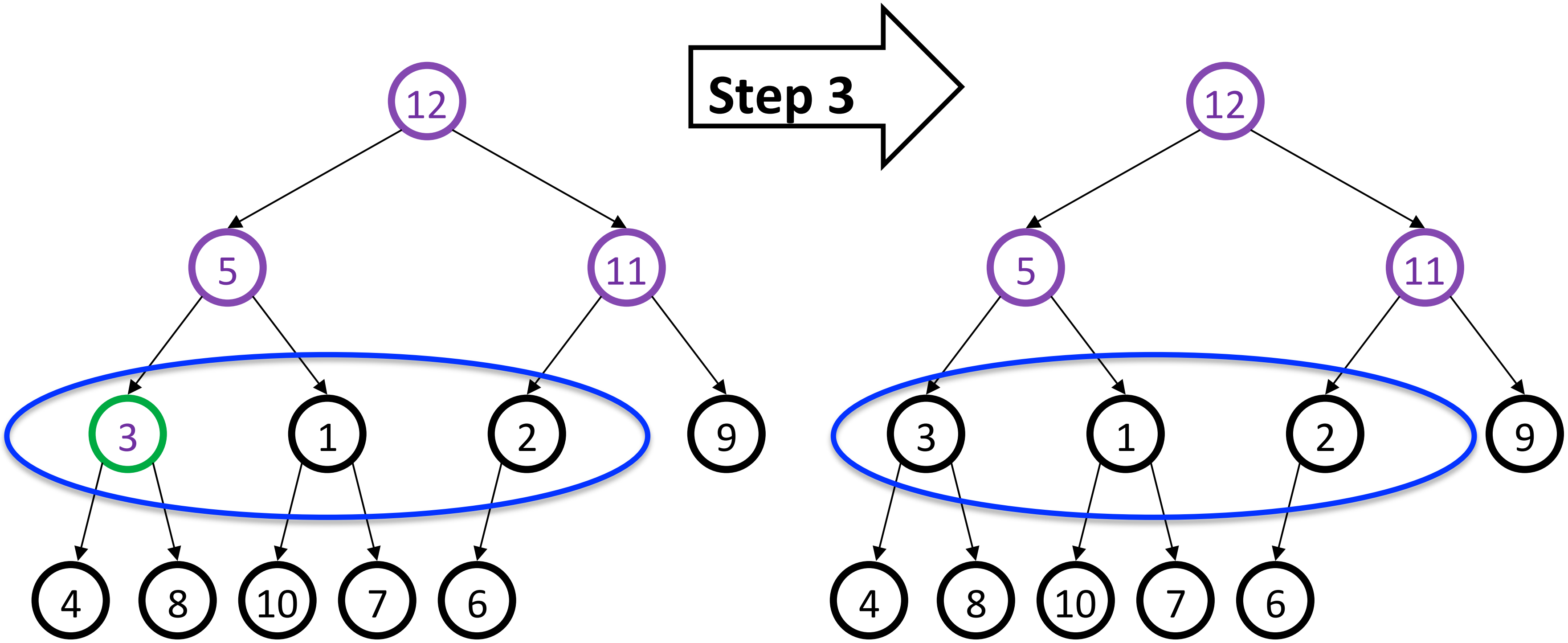
- Happens to already be less than it’s child

Example



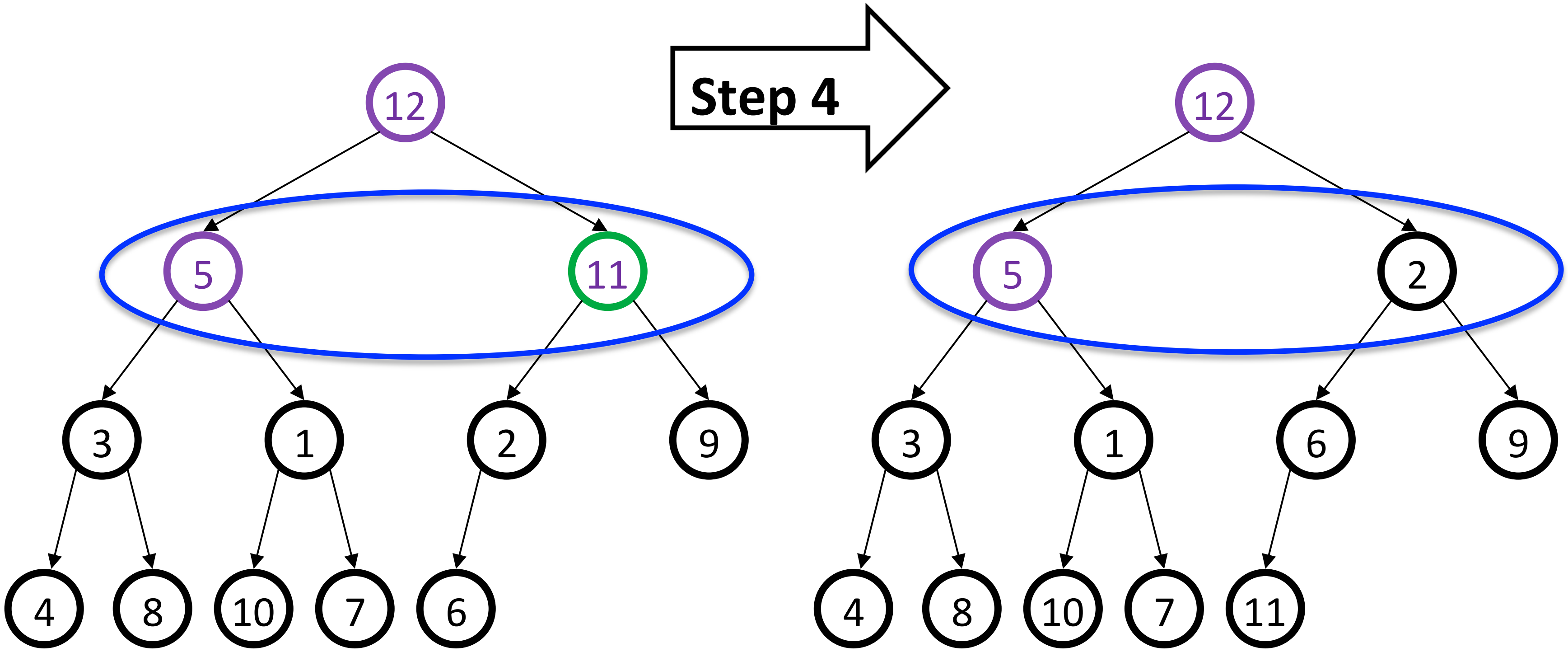
- Percolate down (notice that moves 1 up)

Example



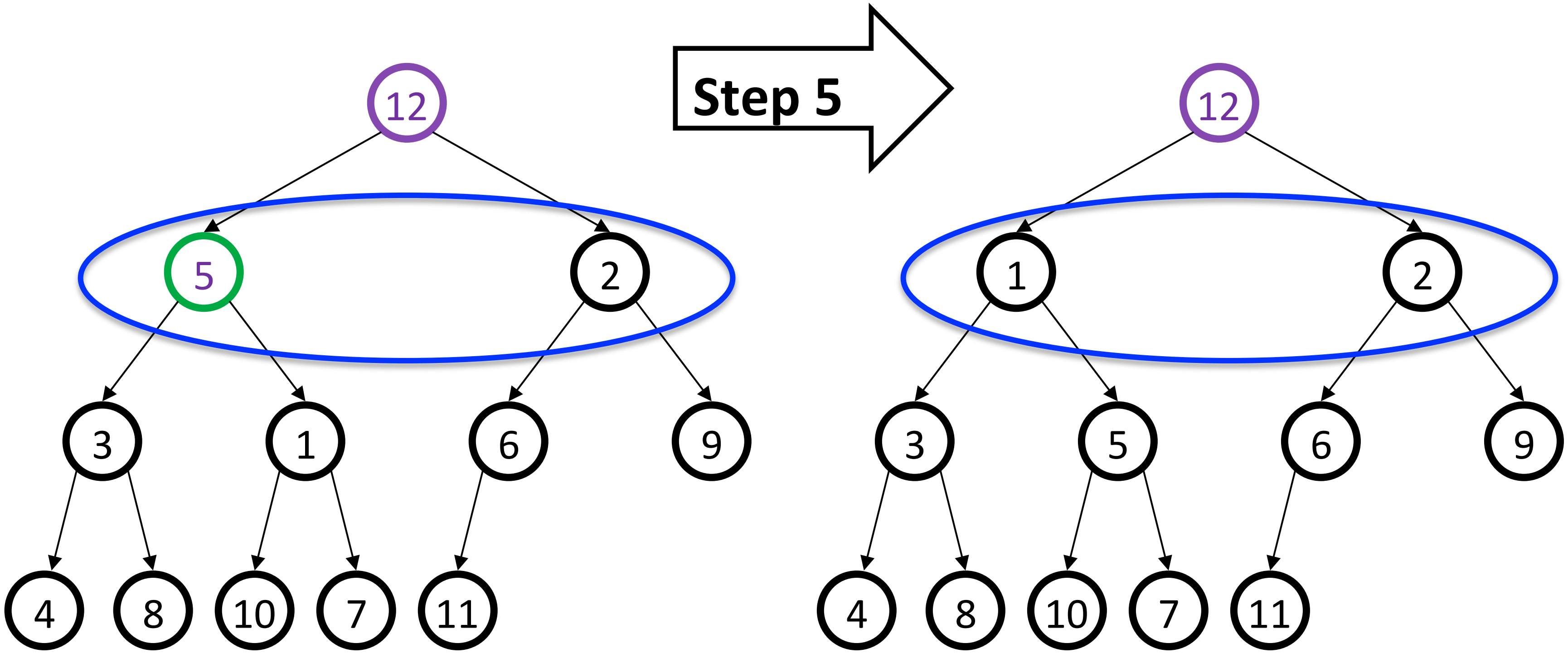
- Another nothing-to-do step

Example

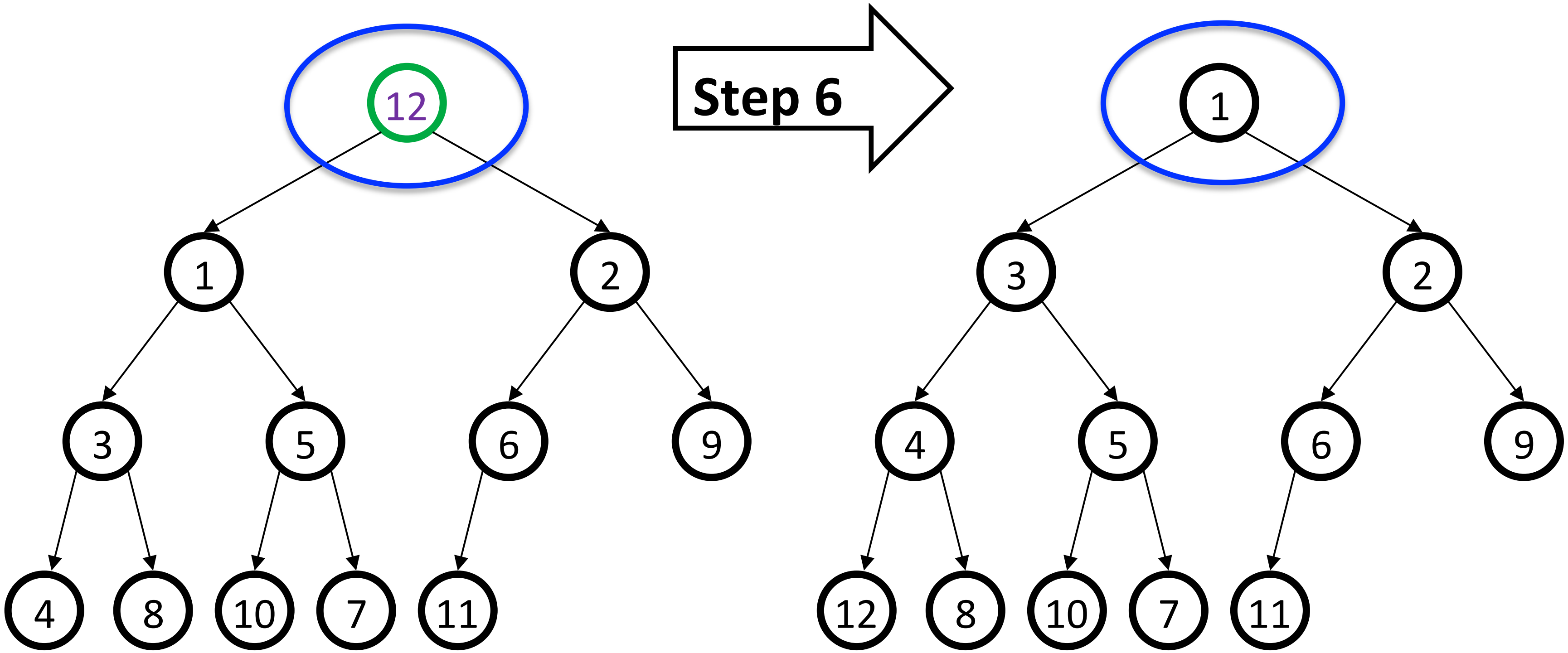


- Percolate down as necessary (steps 4a and 4b)

Example



Example



But is it right?

- “Seems to work”
  - Let’s *prove* it restores the heap property (correctness)
  - Then let’s *prove* its running time (efficiency)

```
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val  = arr[i];
    hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```

Correctness

```
void buildHeap() {
   for(i = size/2; i>0; i--) {
     val  = arr[i];
     hole = percolateDown(i,val);
     arr[hole] = val;
   }
}
```

Loop Invariant: For all **`j>i`**, **`arr[j]`** is less than its children

- True initially: If **`j > size/2`**, then **`j`** is a leaf
  - Otherwise its left child would be at position > **`size`**
- True after one more iteration: loop body and **`percolateDown`** make **`arr[i]`** less than children without breaking the property for any descendants

So after the loop finishes, all nodes are less than their children

Efficiency

```
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val  = arr[i];
    hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```

Easy argument: **`buildHeap`** is *O*(*n* **`log`** *n*) where *n* is **`size`**

- **`size/2`** loop iterations
- Each iteration does one **`percolateDown`**, each is *O*(**`log`** *n*)

This is correct, but there is a more precise ("tighter") analysis of the algorithm…

Efficiency

```
void buildHeap() {
   for(i = size/2; i>0; i--) {
     val  = arr[i];
     hole = percolateDown(i,val);
     arr[hole] = val;
   }
}
```

Better argument: **`buildHeap`** is *O*(*n*) where *n* is **`size`**

- **`size/2`** total loop iterations: *O*(*n*)
- 1/2 the loop iterations percolate at most 1 step
- 1/4 the loop iterations percolate at most 2 steps
- 1/8 the loop iterations percolate at most 3 steps
- …
- ((1/2) + (2/4) + (3/8) + (4/16) + (5/32) + …) < 2 (page 4 of Weiss)
  - So at most **`2(size/2)`** *total* percolate steps: *O*(*n*)

Lessons from `buildHeap`

- Without **`buildHeap`**, our ADT already let clients implement their own in $O(n \log n)$ worst case
  - Worst case is inserting better priority values later
- By providing a specialized operation internal to the data structure (with access to the internal data), we can do $O(n)$ worst case
  - Intuition: Most data is near a leaf, so better to percolate down
- Can analyze this algorithm for:
  - Correctness:
    - Non-trivial inductive proof using loop invariant
  - Efficiency:
    - First analysis easily proved it was $O(n \log n)$
    - Tighter analysis shows same algorithm is $O(n)$

What we're skipping

- **`merge`:** given two priority queues, make one priority queue
  - How might you merge binary heaps:
    - If one heap is much smaller than the other?
    - If both are about the same size?
  - Different pointer-based data structures for priority queues support logarithmic time **`merge`** operation (impossible with binary heaps)
    - Leftist heaps, skew heaps, binomial queues
    - Worse constant factors
    - Trade-offs!

Take a breath

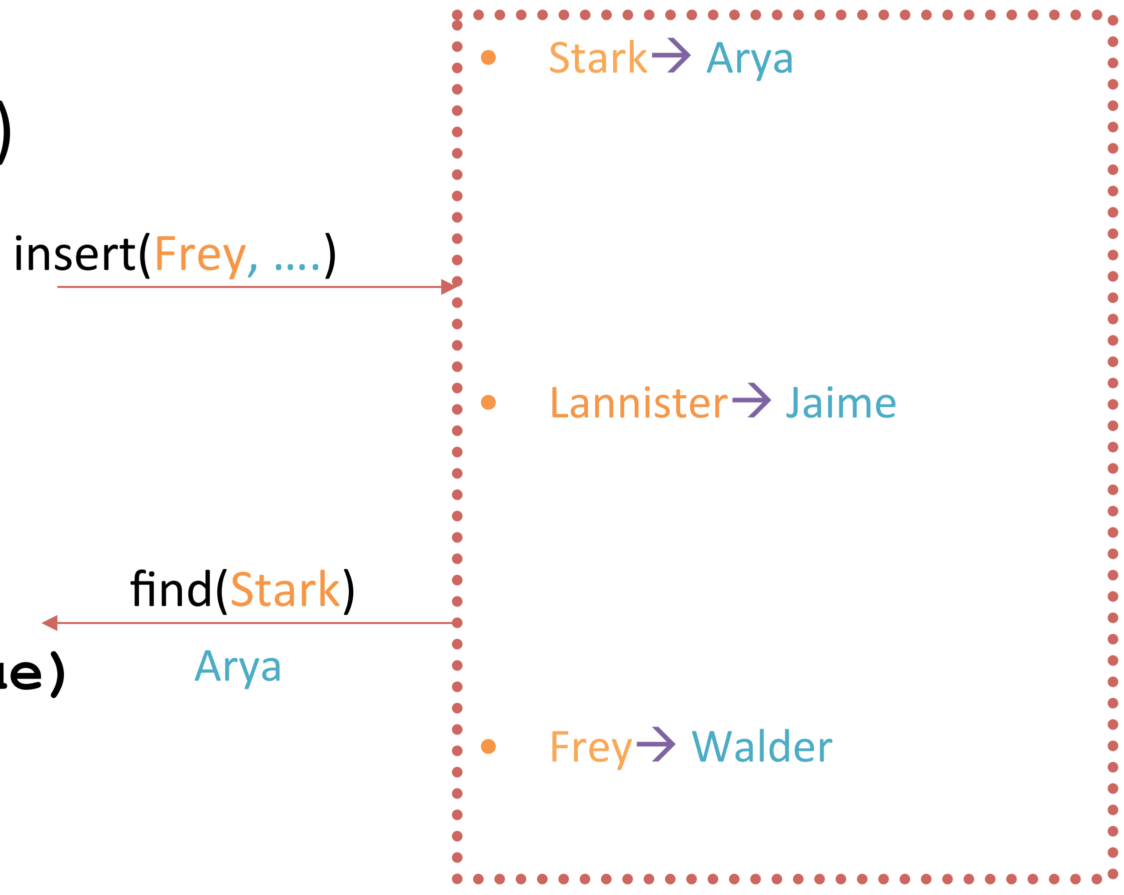
Let’s talk about more ADTs and Data Structures:

- Dictionaries/Maps (and briefly Sets)
- Binary Search Trees

Clear your mind with this picture of a kitten:

The Dictionary (a.k.a. Map) ADT

- Data:
  - *set* of (key, value) *pairs*
  - keys must be *comparable*
- Operations:
  - **`insert(key,value)`**
  - **`find(key)`**
  - **`delete(key)`**
  - …


insert(Frey, ….)

find(Stark)

Arya

- Stark→ Arya
- Lannister→ Jaime
- Frey→ Walder

Will tend to emphasize the keys; don't forget about the stored values

Comparison: The Set ADT

The *Set* ADT is like a Dictionary without any values
  - A key is *present* or not (no duplicates)

For **`find`**, **`insert`**, **`delete`**, there is little difference
  - In dictionary, values are "just along for the ride"
  - So *same data-structure ideas* work for dictionaries and sets

But if your Set ADT has other important operations this may not hold
  - **`union`**, **`intersection`**, **`is_subset`**
  - Notice these are binary operators on sets

binary operation: a rule for combining two objects of a given type, to obtain another object of that type

Applications

Any time you want to store information according to some key and be able to retrieve it efficiently. Lots of programs do that!

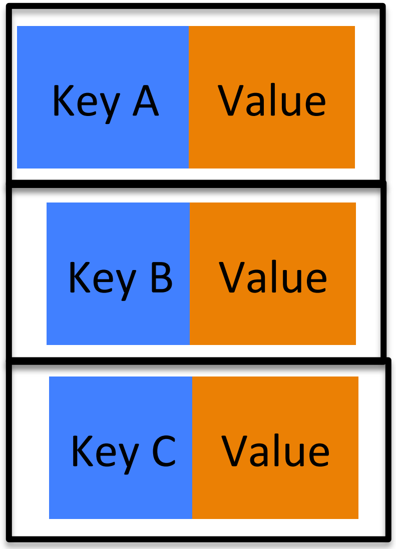
- Lots of fast look-up uses in search: inverted indexes, storing a phone directory, etc
- Routing information through a Network
- Operating systems looking up information in page tables
- Compilers looking up information in symbol tables
- Databases storing data in fast searchable indexes
- Biology genome maps

Dictionary Implementation Intuition

We store the keys with their values so all we really care about is how the keys are stored.

- want fast operations for iterating over the keys

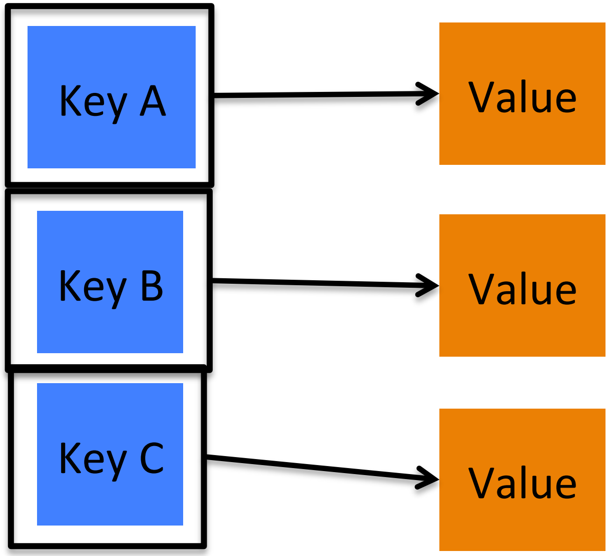
You could think about this in a couple ways:



Key A Value

Key B Value

Key C Value



Key A → Value

Key B → Value

Key C → Value

Simple implementations

For dictionary with *n* key/value pairs

	`insert`	`find`	`delete`
Unsorted linked-list	$O(1)$*	$O(n)$	$O(n)$
Unsorted array	$O(1)$*	$O(n)$	$O(n)$
Sorted linked list	$O(n)$	$O(n)$	$O(n)$
Sorted array	$O(n)$	$O(\log n)$	$O(n)$

* Unless we need to check for duplicates

We'll see a Binary Search Tree (BST) probably does better, but not in the worst case unless we keep it balanced

Implementations we'll see soon

There are many good data structures for (large) dictionaries

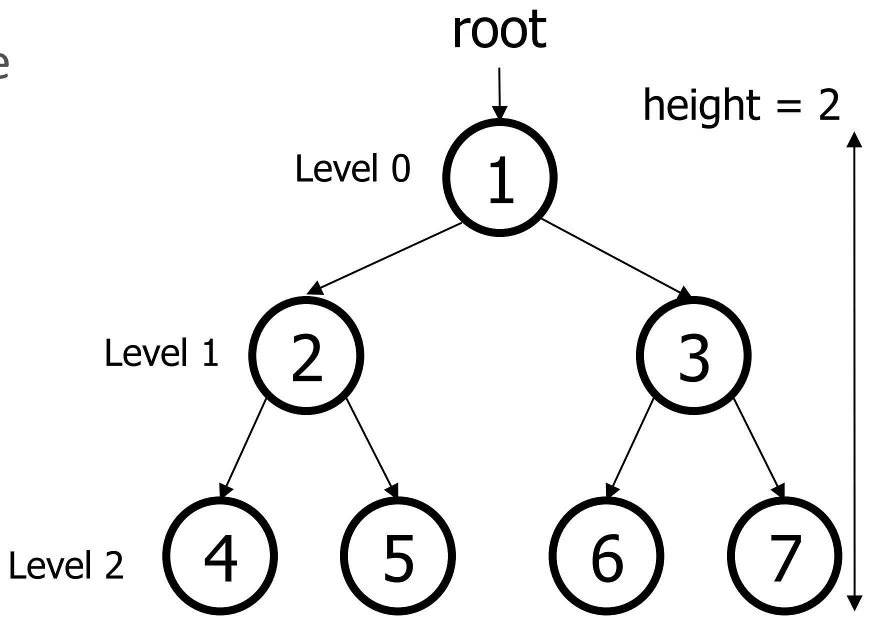
1. AVL trees (next week)
   - Binary search trees with *guaranteed balancing*
2. B-Trees (an extra topic we might have time for)
   - Also always balanced, but different and shallower
   - B ≠ Binary; B-Trees generally have large branching factor
3. Hashtables (in two weeks)
   - Not tree-like at all

Skipping: Other, really cool, balanced trees (e.g., red-black, splay)

Reference: Tree Terminology

- **node**: an object containing a data value and left/ right children
  - **root**: topmost node of a tree
  - **leaf**: a node that has no children
  - **branch**: any internal node (non-root)
  - **parent**: a node that refers to this one
  - **child**: a node that this node refers to
  - **sibling**: a node with a common

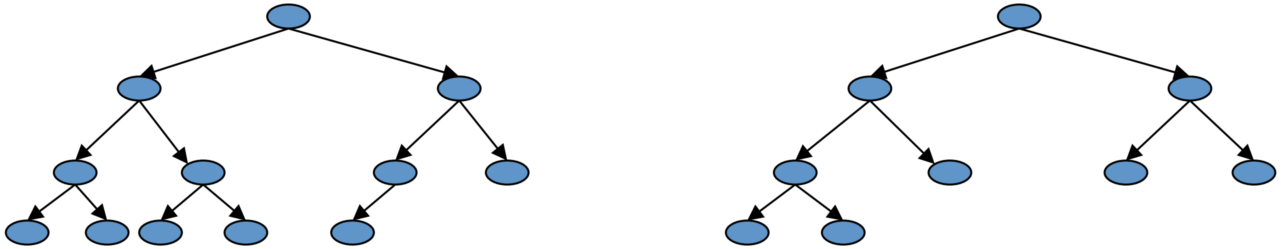
- **subtree**: the smaller tree of nodes on the left or right of the current node
- **height**: length of the longest path from the root to any node (count edges)
- **level** or **depth**: length of the path from a root to a given node



Reference: kinds of trees

Certain terms define trees with specific structure

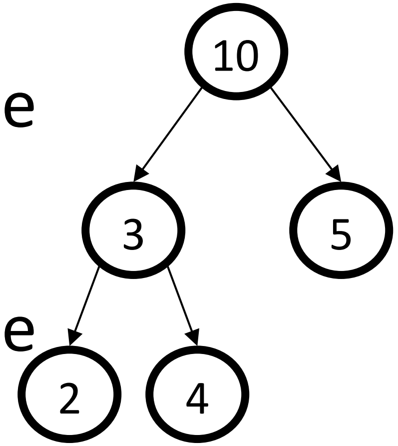
- Binary tree: Each node has at most 2 children (branching factor 2)
- *n*-ary tree: Each node has at most *n* children (branching factor *n*)
- Perfect tree: Each row completely full
- Full tree: Each node has 0 or 2 children
- Complete tree: Each row completely full except maybe the bottom row, which is filled from left to right



Review from 143: Tree Traversals

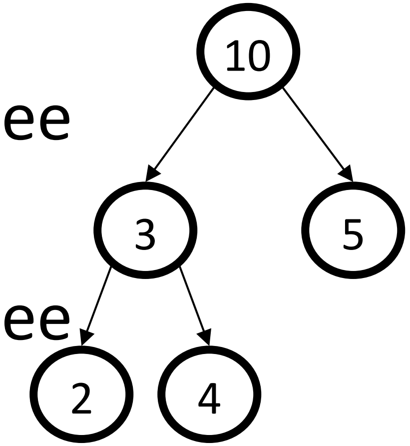
A *traversal* is an order for visiting all the nodes of a tree

- *Pre-order*: root, left subtree, right subtree
- *In-order*: left subtree, root, right subtree
- *Post-order*: left subtree, right subtree, root



Review from 143: Tree Traversals

A *traversal* is an order for visiting all the nodes of a tree



- ***Pre-order*:** root, left subtree, right subtree

  10 3 2 4 5

- ***In-order*:** left subtree, root, right subtree

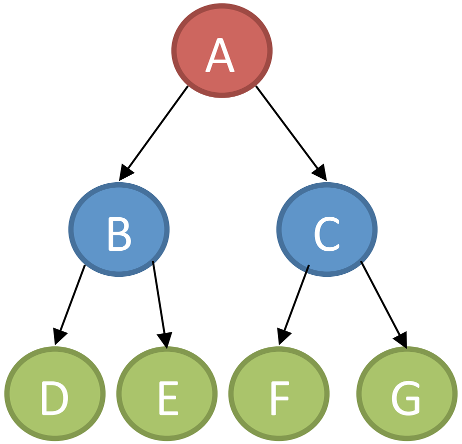
  2 3 4 10 5

- ***Post-order*:** left subtree, right subtree, root

  2 4 3 5 10

More on traversals

```
void inOrderTraversal(Node t){
  if(t != null) {
    inOrderTraversal(t.left);
    process(t.element);
    inOrderTraversal(t.right);
  }
}
```



Sometimes order doesn't matter

- Example: sum all elements

Sometimes order matters

- Example: print tree with parent above indented children (pre-order)
- Example: evaluate an expression tree (post-order)

```
A
  B
    D
    E
  C
    F
    G
```

Computable data for Binary Trees

Recall: height of a tree = longest path from root to leaf (count edges)

For binary tree of height *h*:

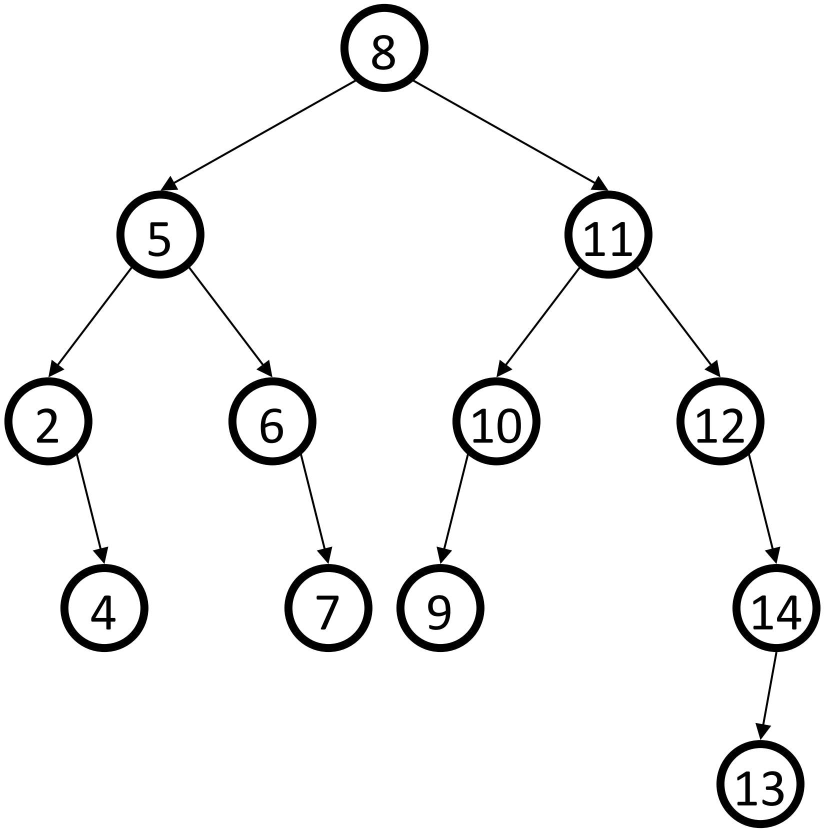
- max # of leaves: 2^h
- max # of nodes: $2^{(h+1)} - 1$
- min # of leaves: 1
- min # of nodes: $h + 1$

For n nodes:

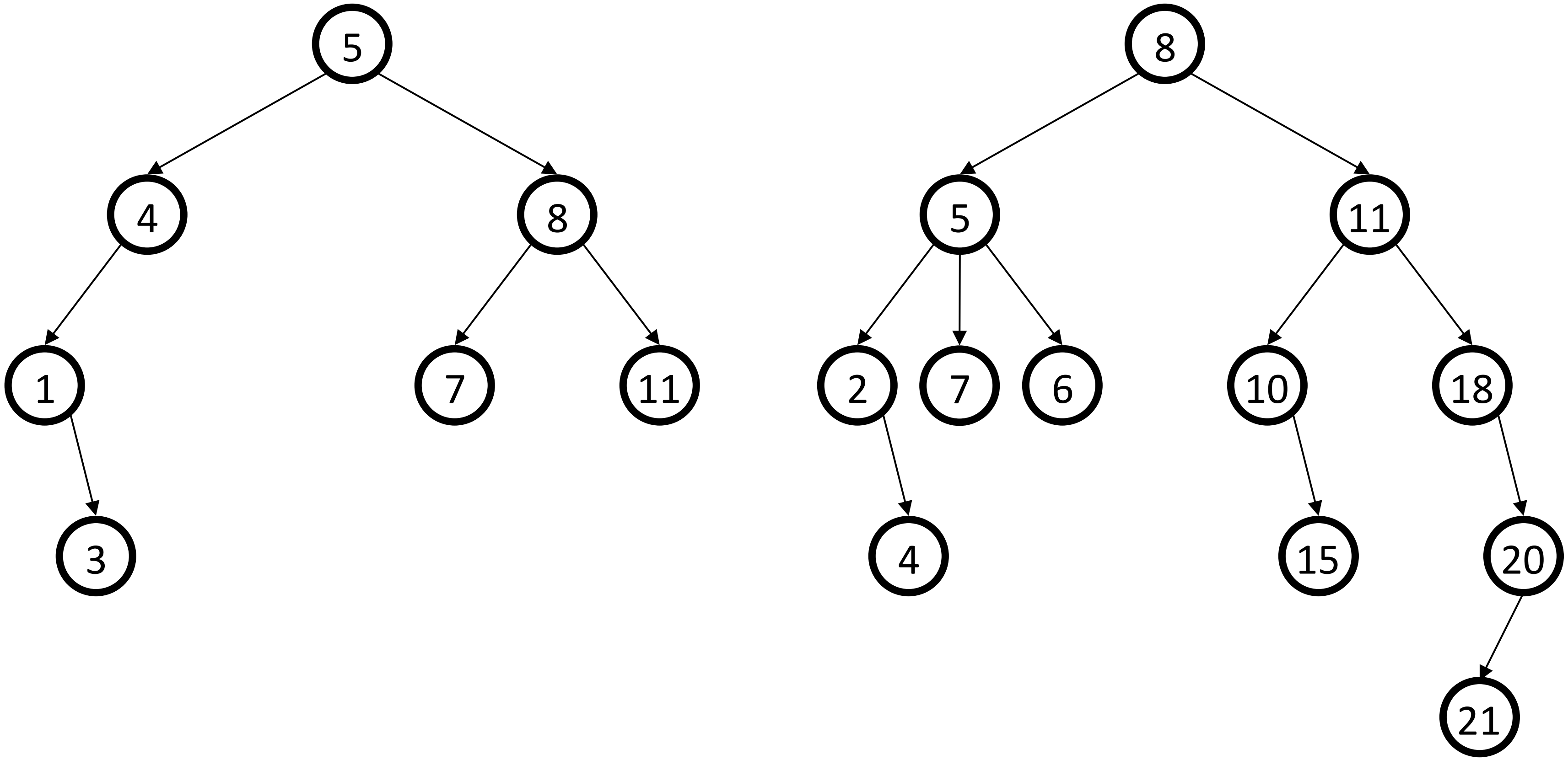
- *best case is O(log n) height*
- *worst case is O(n) height*

Review: Binary Search Tree

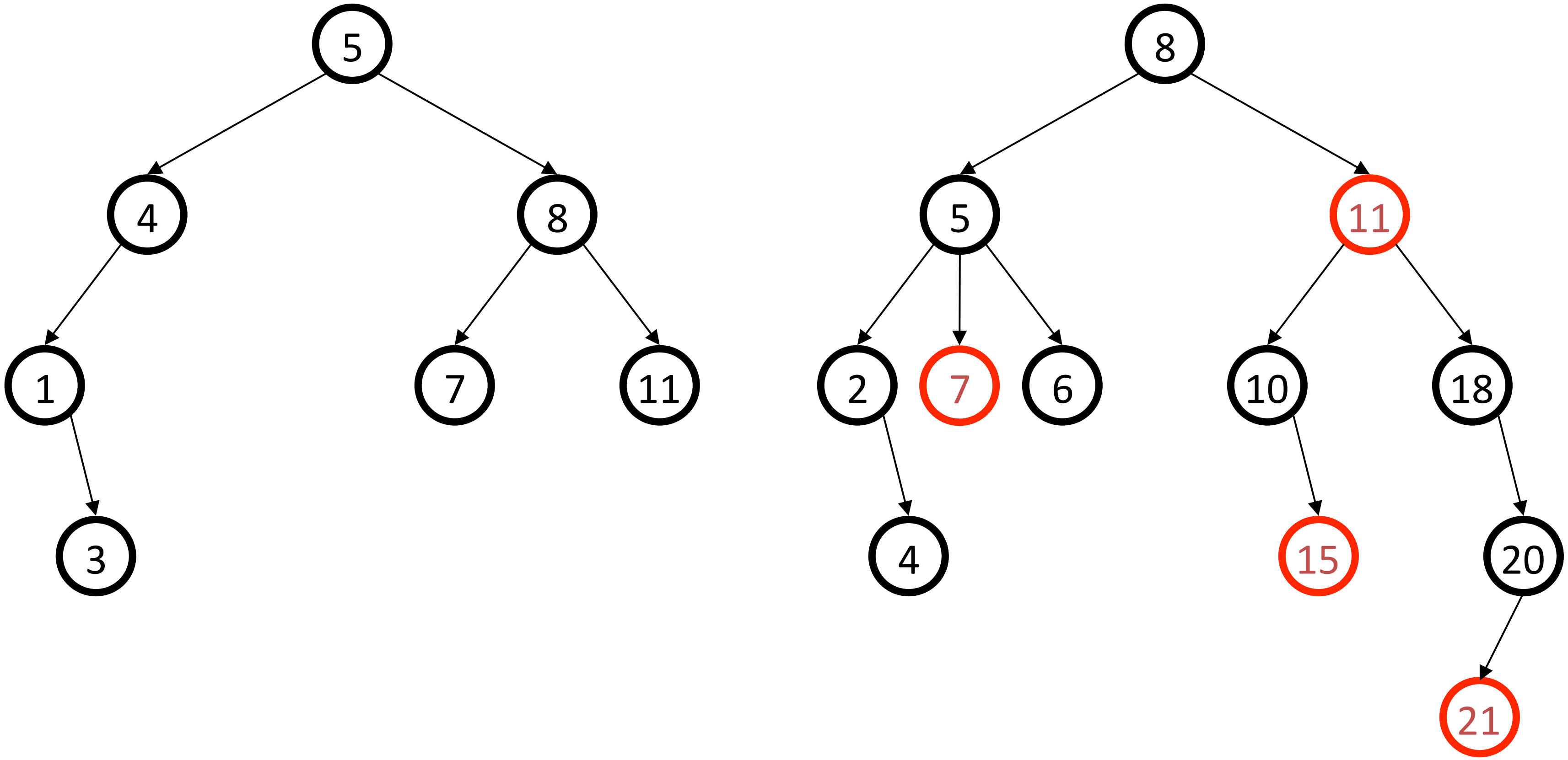
- Structure property ("binary")
  - Each node has ≤ 2 children
  - Result: keeps operations simple
- Order property
  - All keys in left subtree smaller than node's key
  - All keys in right subtree larger than node's key
  - Result: easy to find any given key



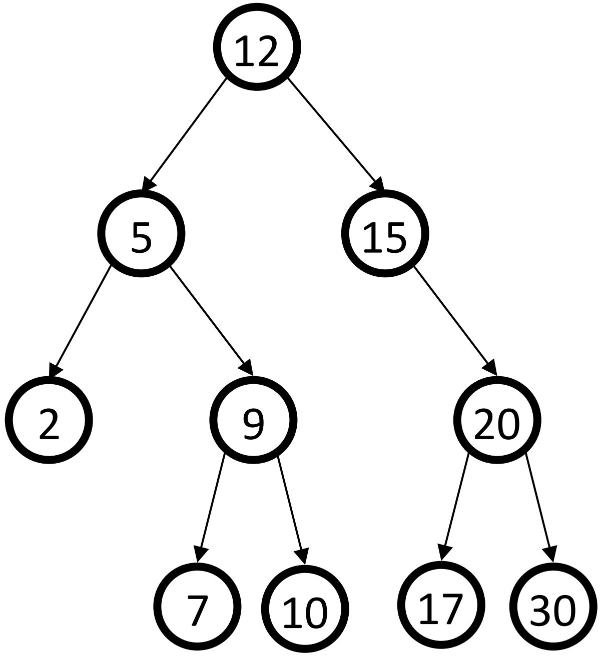
Are these BSTs?



Are these BSTs?

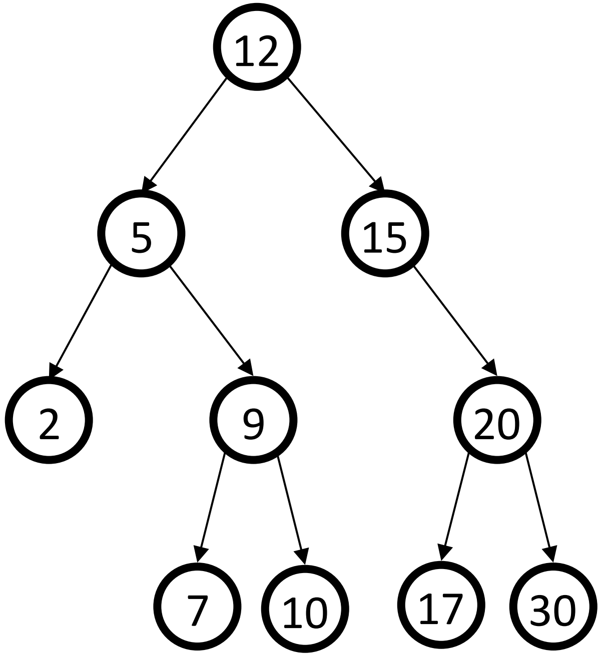


Find in BST, Recursive



```
int find(Key key, Node root){
 if(root == null)
   return null;
 if(key < root.key)
   return find(key,root.left);
 if(key > root.key)
   return find(key,root.right);
 return root.data;
}
```

Find in BST, Iterative

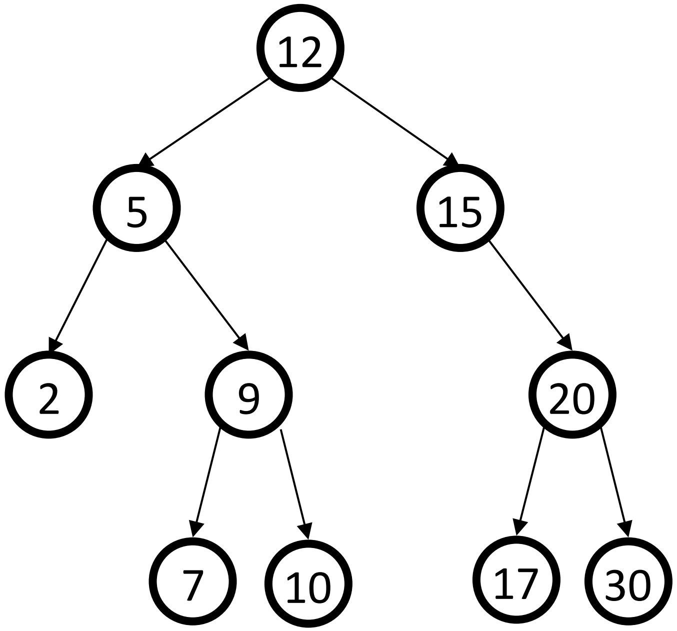


```
int find(Key key, Node root){
 while(root != null && root.key != key){
  if(key < root.key)
    root = root.left;
  else(key > root.key)
    root = root.right;
 }
 if(root == null)
    return null;
 return root.data;
}
```

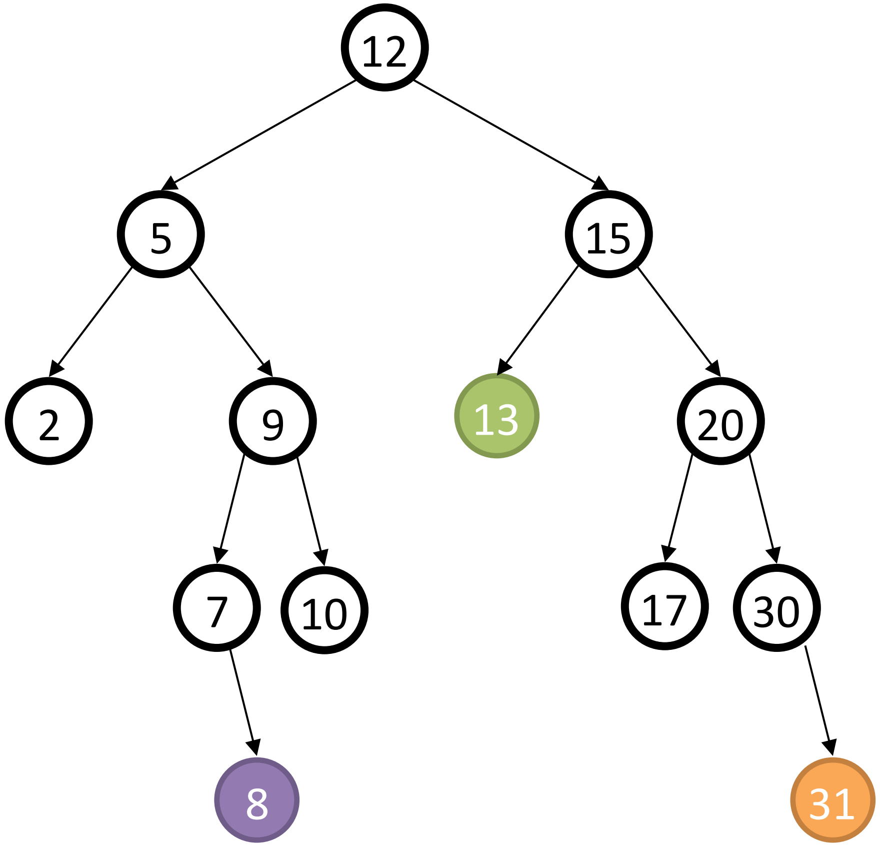
Other "Finding" Operations

- Find *minimum* node
- Find *maximum* node

- Find *predecessor* of a non-leaf
- Find *successor* of a non-leaf
- Find *predecessor* of a leaf
- Find *successor* of a leaf



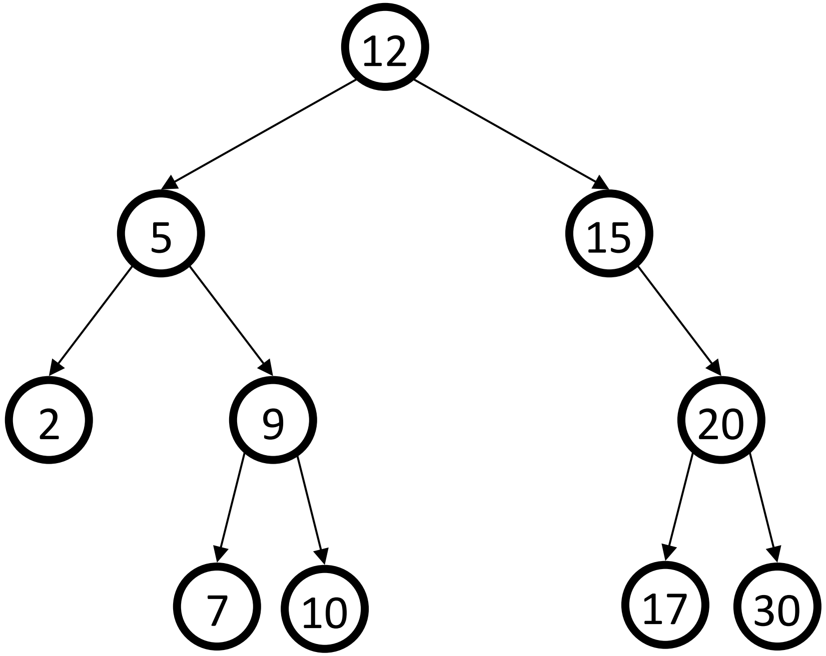
Insert in BST



```
insert(13)
insert(8)
insert(31)
```

(New) insertions happen only at leaves – easy!

Deletion in BST



Why might deletion be harder than insertion?

Deletion

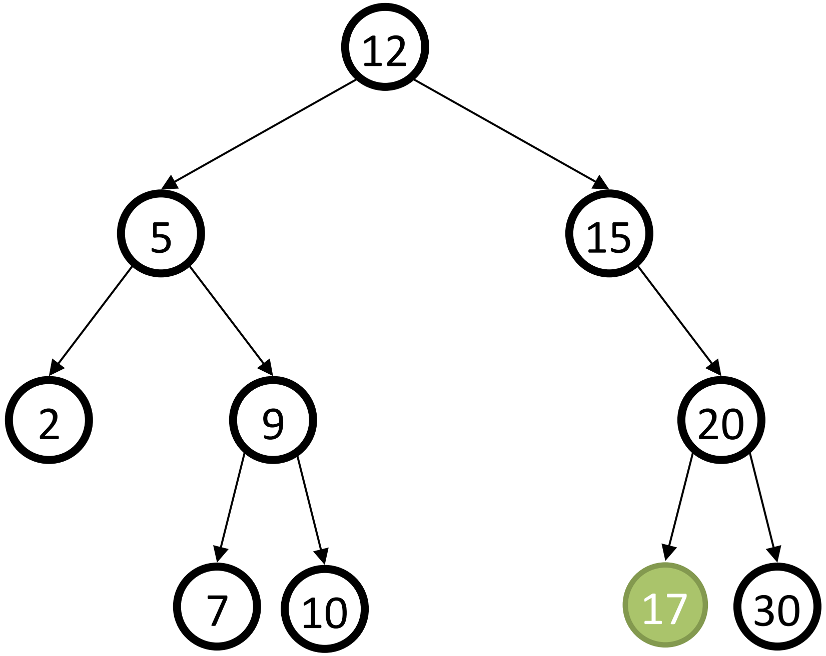
- Removing an item disrupts the tree structure

- Basic idea: **`find`** the node to be removed, then "fix" the tree so that it is still a binary search tree

- Three cases:
  - Node has no children (leaf)
  - Node has one child
  - Node has two children

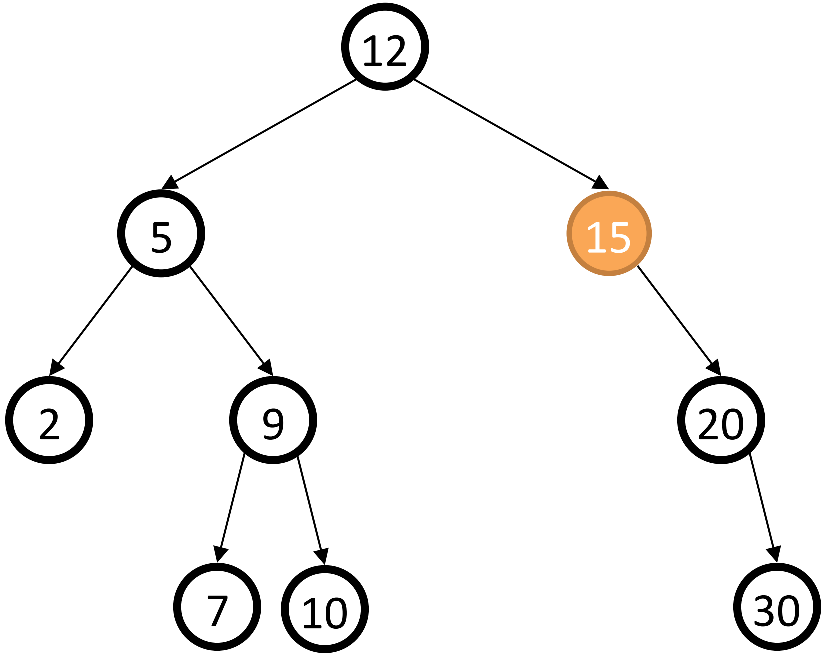
Deletion – The Leaf Case

`delete(17)`



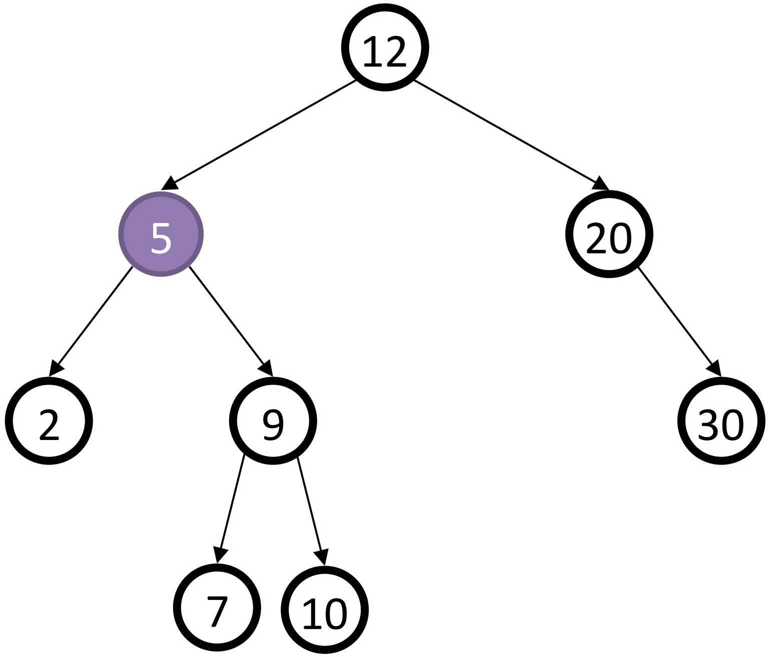
Deletion – The One Child Case

`delete(15)`



Deletion – The Two Child Case

`delete(5)`



What can we replace 5 with?

Deletion – The Two Child Case

Idea: Replace the deleted node with a value guaranteed to be between the two child subtrees

Options:

- *successor* from right subtree: **`findMin(node.right)`**
- *predecessor* from left subtree: **`findMax(node.left)`**
  - These are the easy cases of predecessor/successor

Now delete the original node containing *successor* or *predecessor*

- Leaf or one child case – easy cases of delete!

Today's Takeaways

- Floyd's Algorithm for building heaps: understand why it works and how it's implemented.
- Review Dictionaries/Maps/Sets: understand how to be a client of them and the ADT, think about tradeoffs for implementations.
- Review BSTs: Understand the terms, how to insert, delete, and evaluate the runtime of those operations.