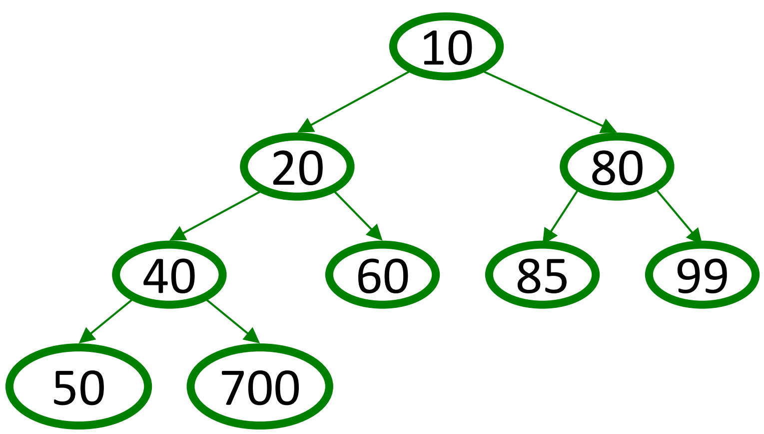
Lecture 7
Binary Heaps

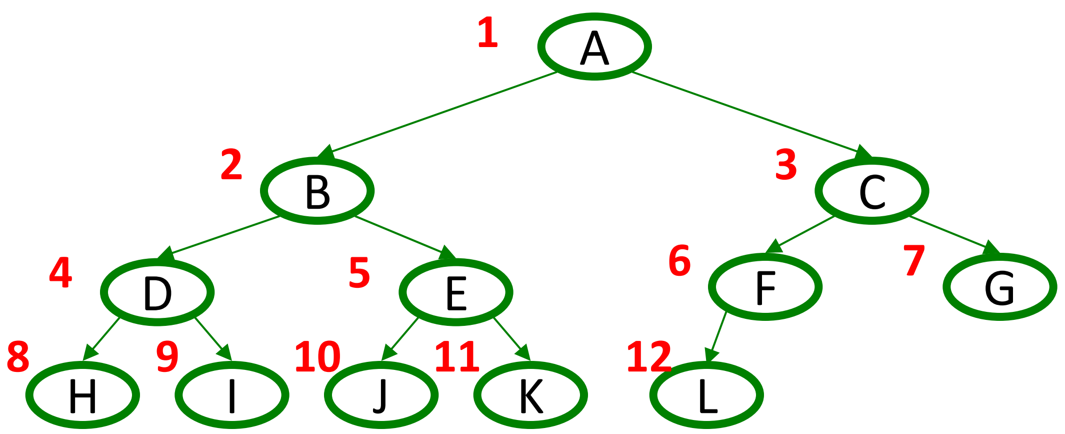
Heaps

A *binary min-heap* (or just *binary heap* or just *heap*) is:

- Structure property: A *complete*e binary tree
- Heap property: The priority of every (non-root) node is greater than (or equal to) the priority of its parent. AKA the children are always greater than the parents.



Array Representation of Binary Trees



From 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

Heap Operations Runtimes

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

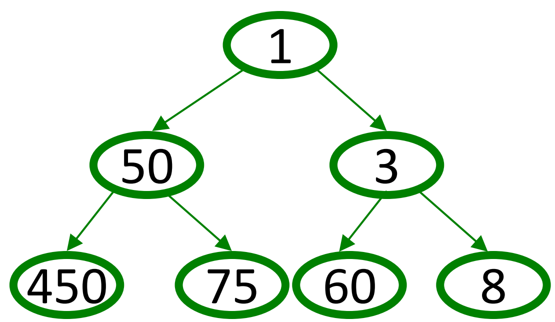
at worst case, the number of swaps you have to do is the height of the tree. The height of a complete tree with N nodes is logN.

Intuition:

1 Node

2 Nodes

4 Nodes



2^0 Nodes

2^1 Nodes

2^2 Nodes

Judging the array implementation

Plusses:

- Less "wasted" space
  - Just index 0 and unused space on right
  - In conventional tree representation, one edge per node (except for root), so *n*-1 wasted space (like linked lists)
  - Array would waste more space if tree were not complete
- Multiplying and dividing by 2 is very fast (shift operations in hardware)
- Last used position is just index **`size`**

Minuses:

- Same might-be-empty or might-get-full problems we saw with stacks and queues (resize by doubling as necessary)

Plusses outweigh minuses: "this is how people do it"

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

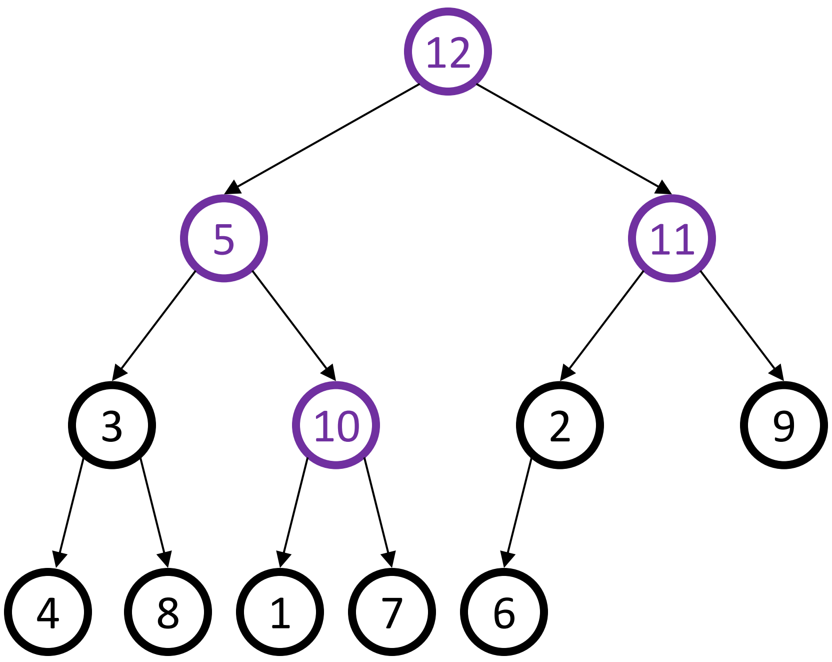
1. Put the *n* values anywhere to make a complete structural tree

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)

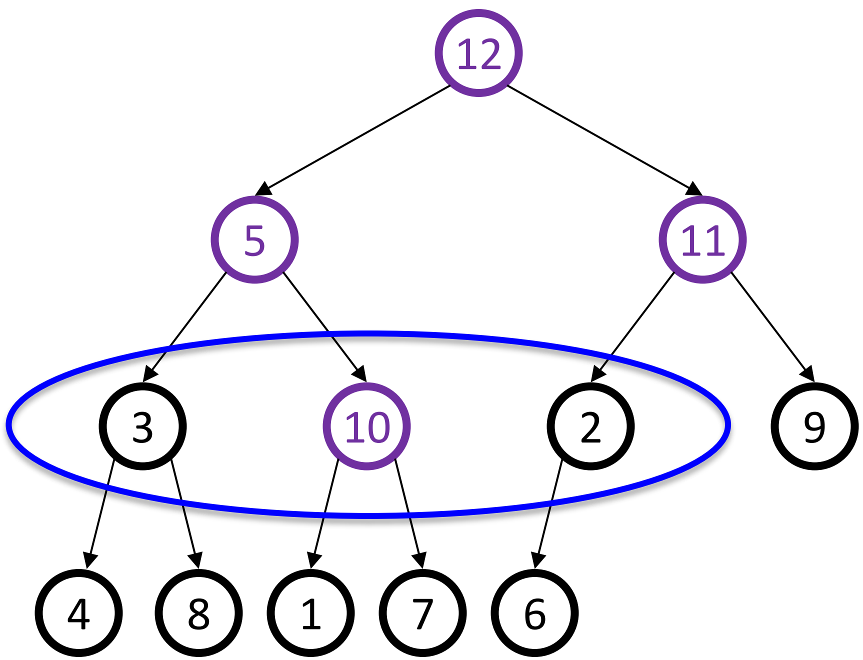


Algorithm Example

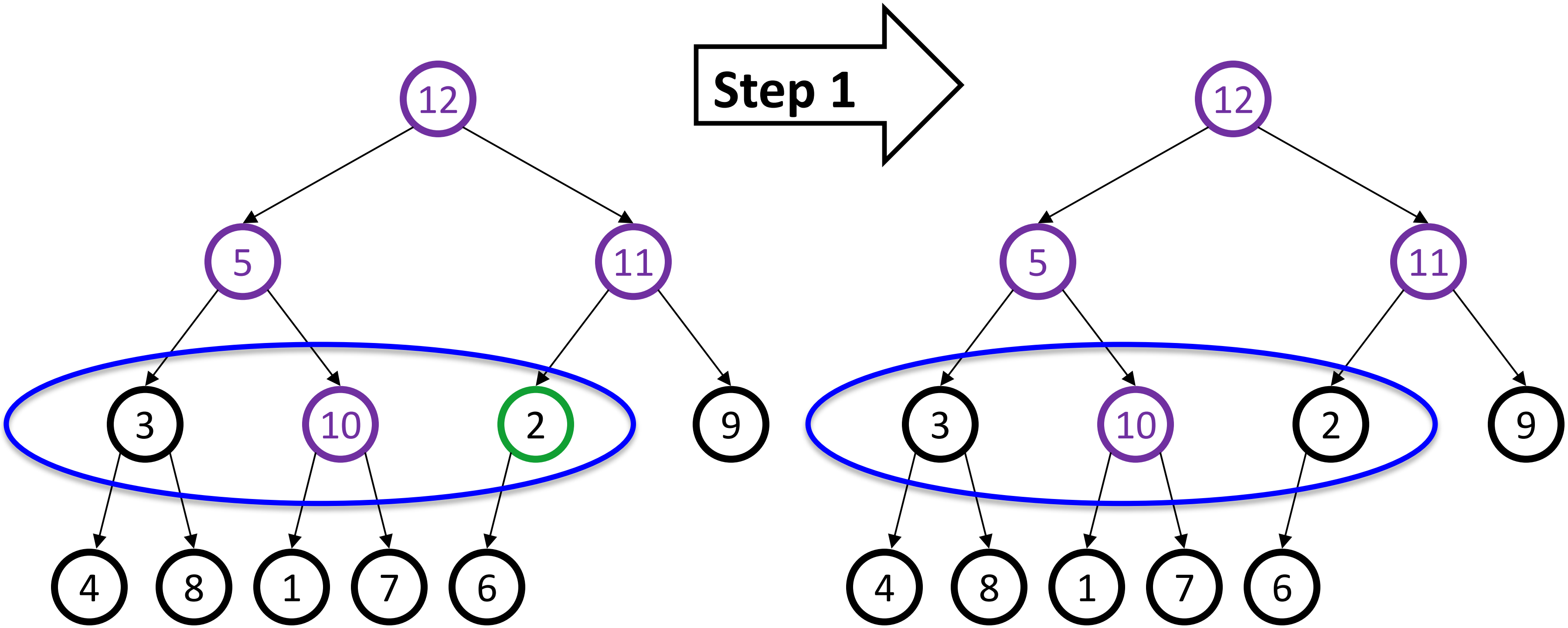
Purple shows the nodes that will need to be fixed.

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

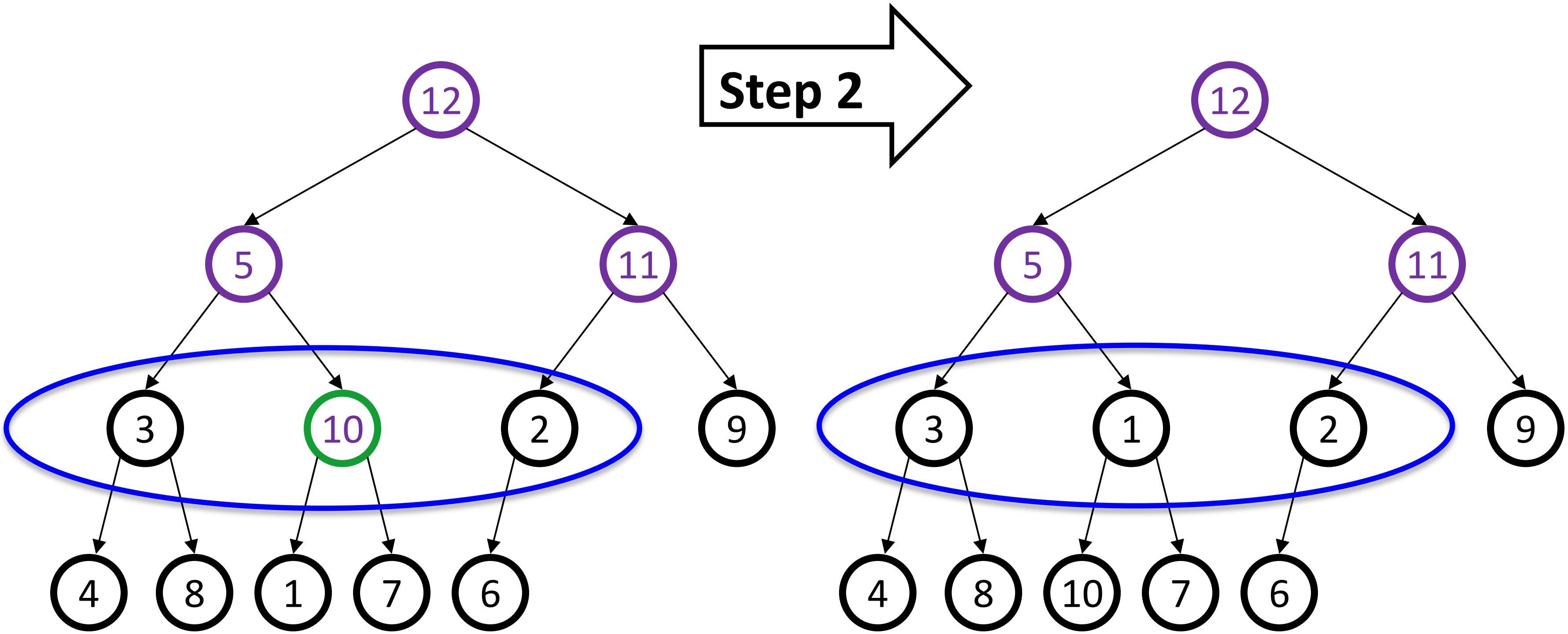


Algorithm Example



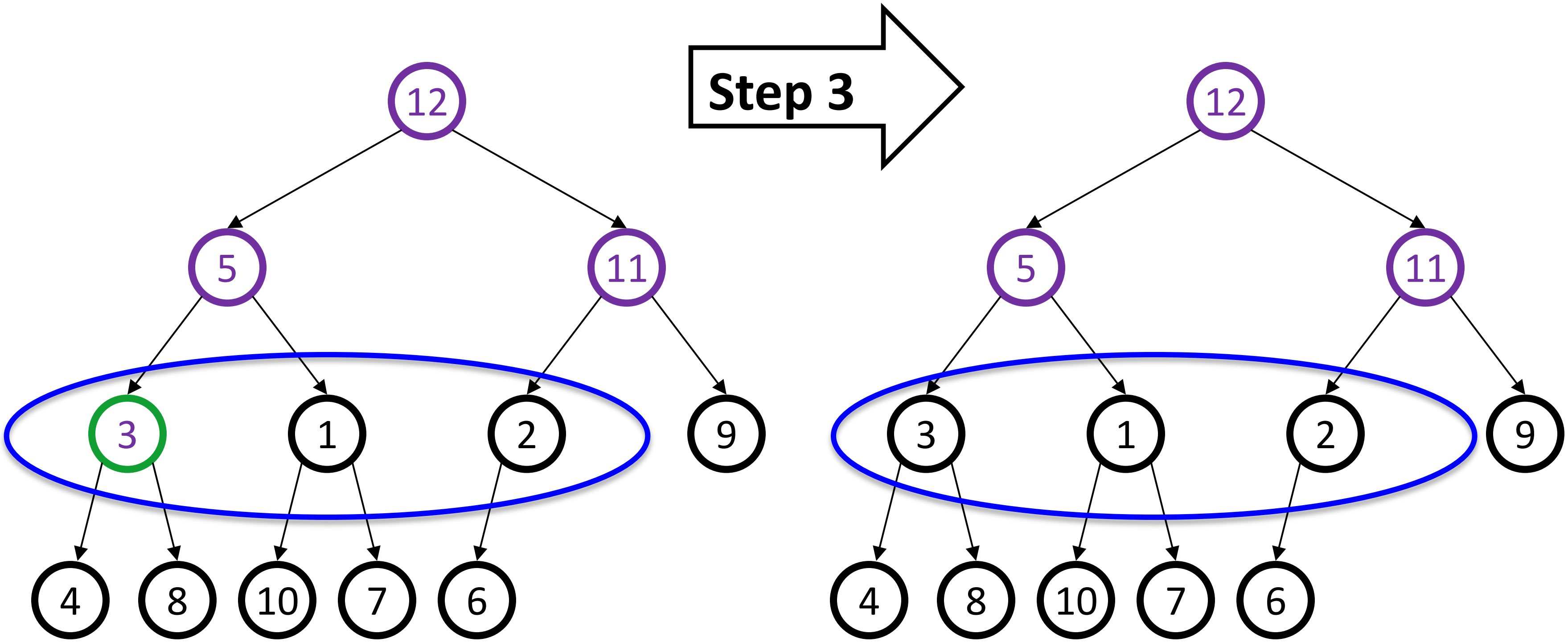
- Happens to already be less than it's child

Example



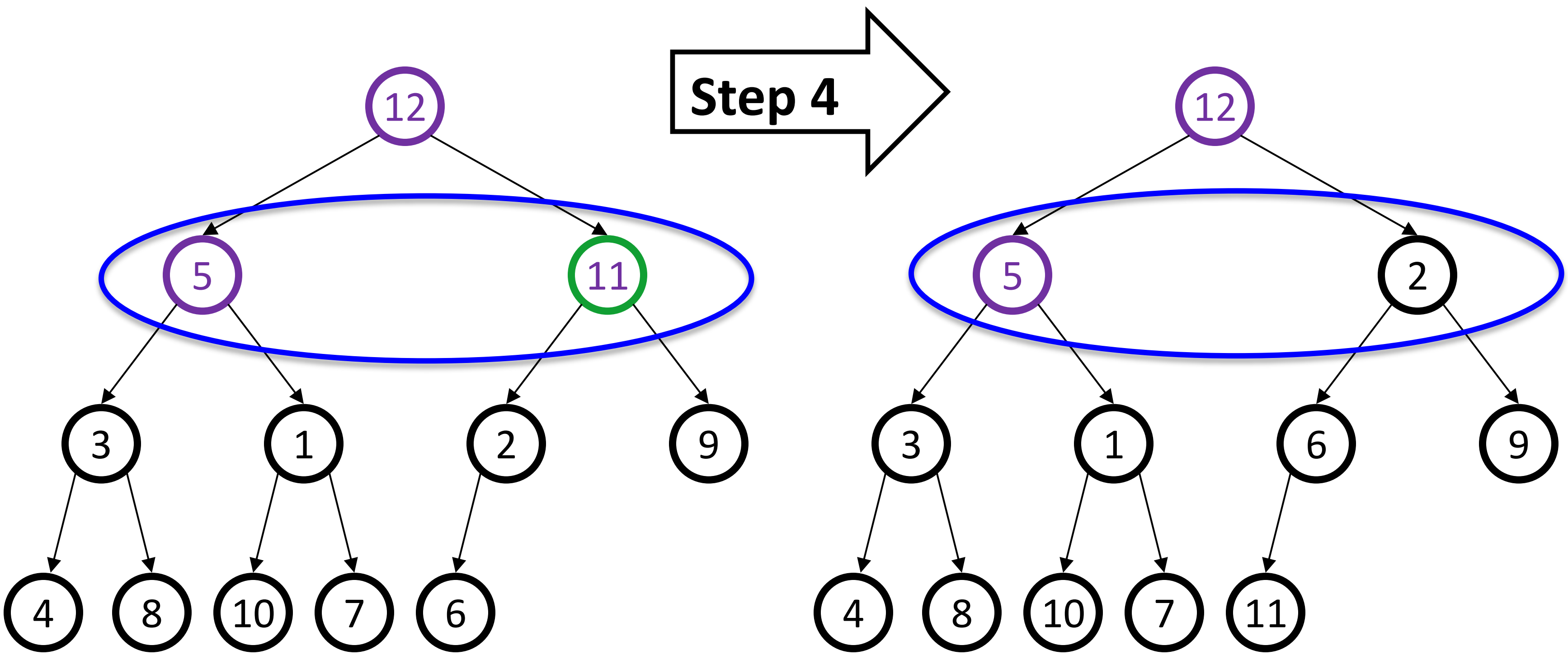
- Percolate down (notice that moves 1 up)

Example



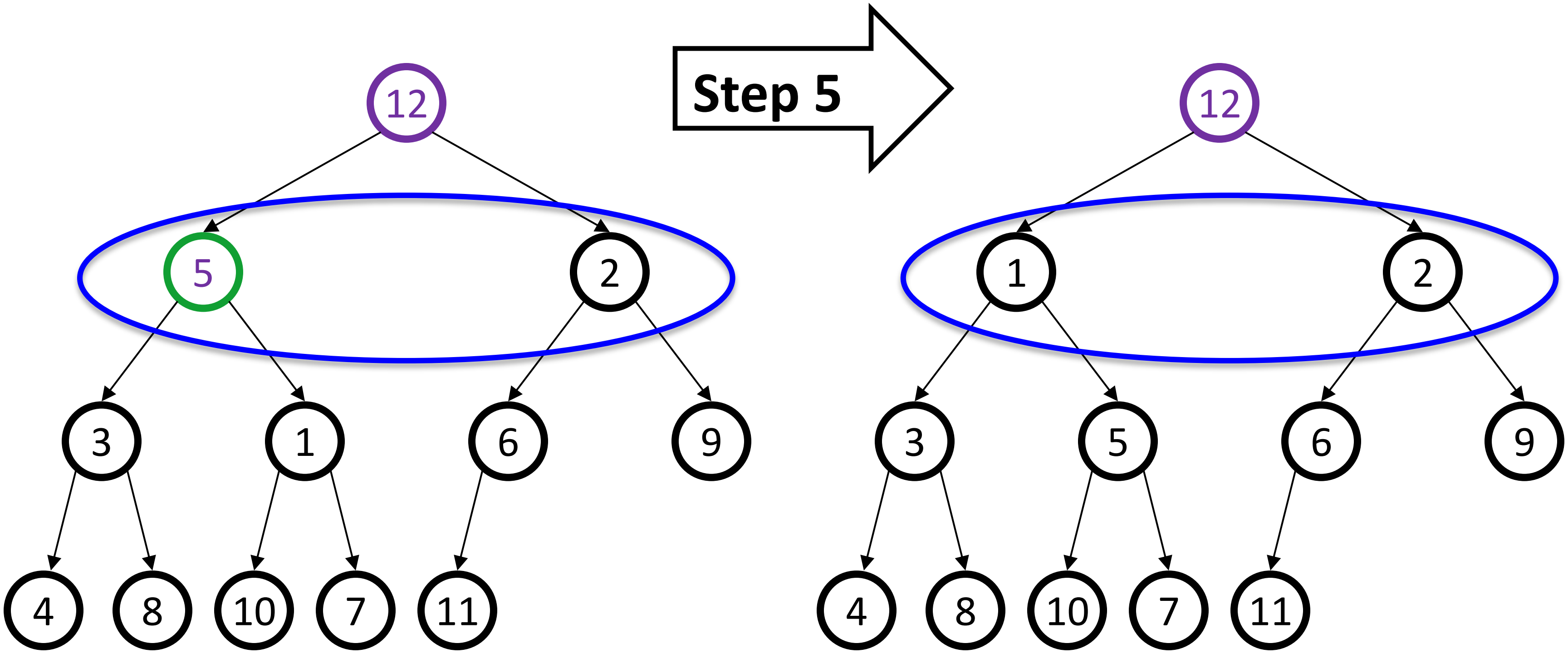
- Another nothing-to-do step

Example

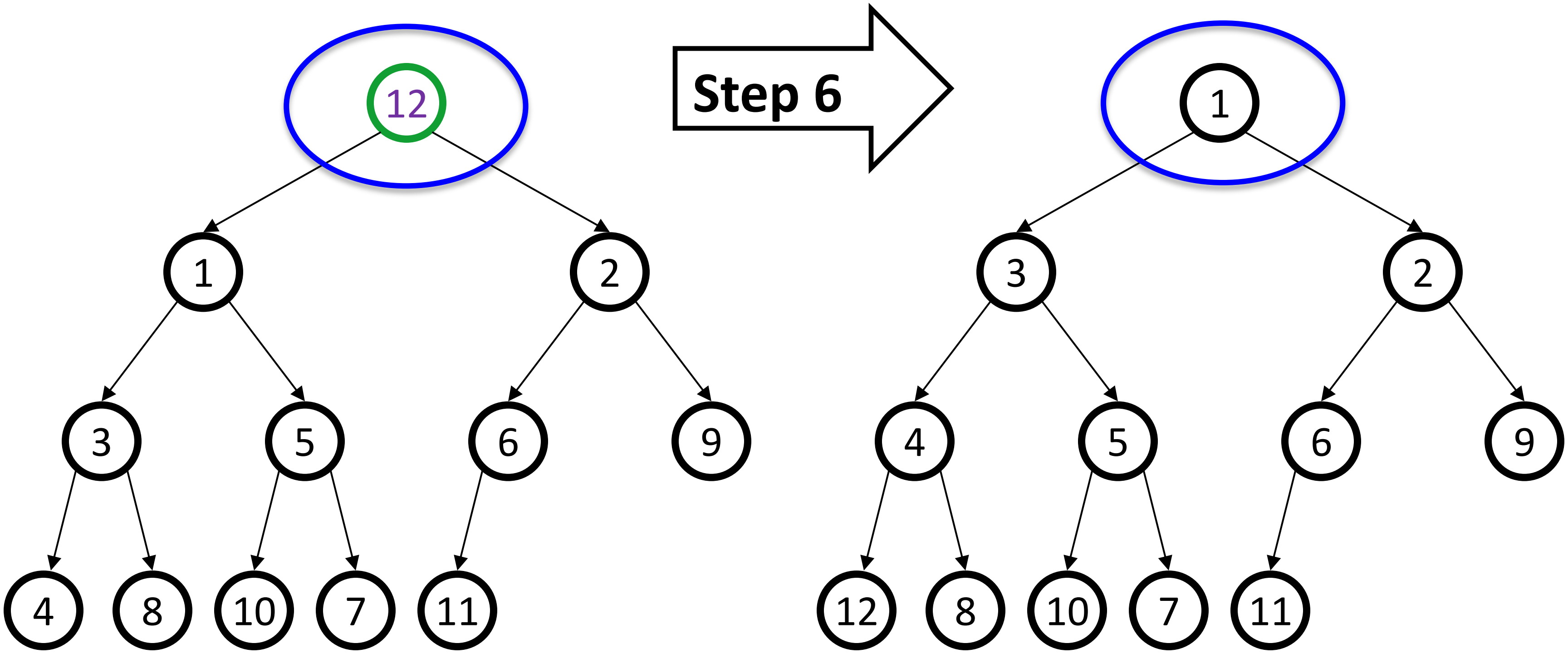


- Percolate down as necessary (steps 4a and 4b)

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 $O(n)$

Other function

- `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!