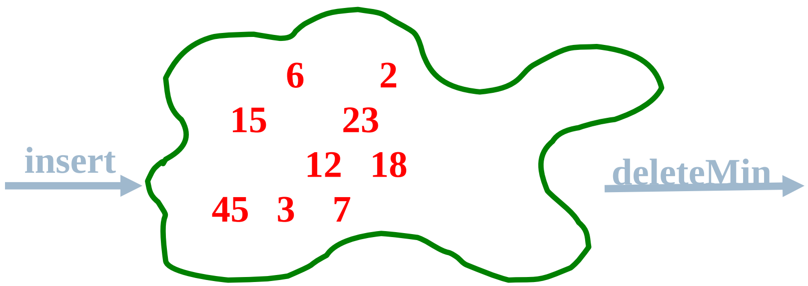
CSE332: Data Abstractions

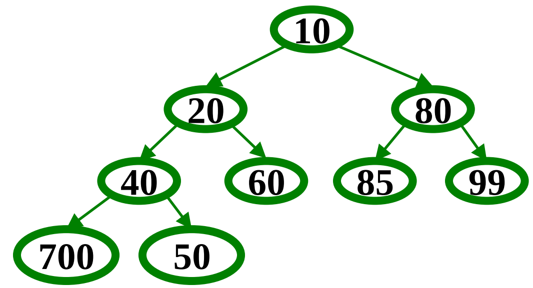
Lecture 5: Binary Heaps, Continued

Tyler Robison

Summer 2010

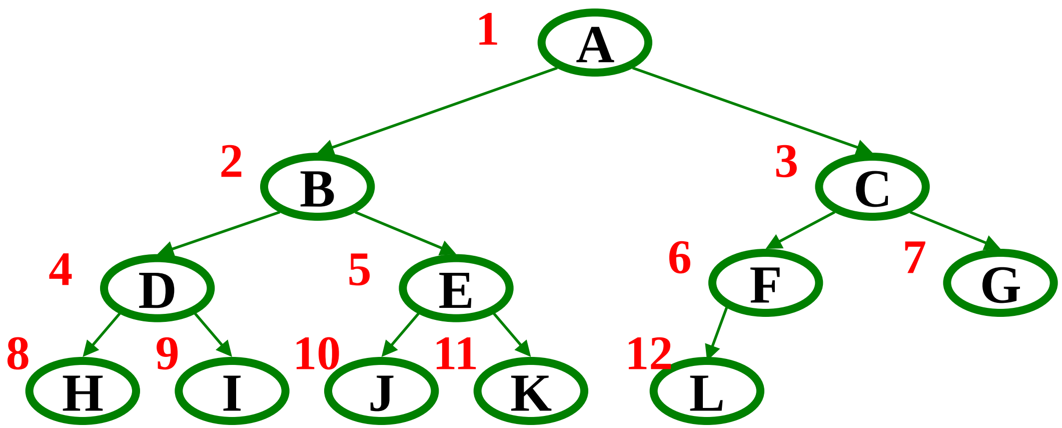
Review





- Priority Queue ADT: **`insert`** comparable object, **`deleteMin`**
- Binary heap data structure: Complete binary tree where each node has a lesser priority than its parent (greater value)
- *O*(height-of-tree)=*O*(**`log`** *n*) **`insert`** and **`deleteMin`** operations
  - **`insert`**: put at new last position in tree and percolate-up
  - **`deleteMin`**: remove root, put last element at root and percolate-down
- But: tracking the "last position" is painful and we can do better

Clever Trick: Array Representation of Complete Binary Trees



From node `i`:

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

(wasting index 0 is convenient)

implicit (array) implementation:

12	**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

We can use index 0 to store other info, such as the size

Judging the array implementation

Plusses:

- Non-data 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
- For reasons you learn in CSE351 / CSE378, multiplying and dividing by 2 is very fast
- **`size` is the index of the last node**

Minuses:

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

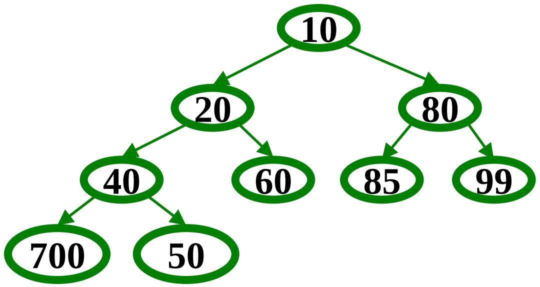
Plusses outweigh minuses: “this is how people do it”

Pseudocode: insert

Note this pseudocode inserts ints, not useful data with priorities

```
void insert(int val) {
  if(size==arr.length-1)
    resize();
  size++;
  i=percolateUp(size,val);
  arr[i] = val;
}
```

```
int percolateUp(int hole,
                int val) {
  while(hole > 1 &&
          val < arr[hole/2])
    arr[hole] = arr[hole/2];
    hole = hole / 2;
  }
  return hole;
}
```



O(logn): Or is it…

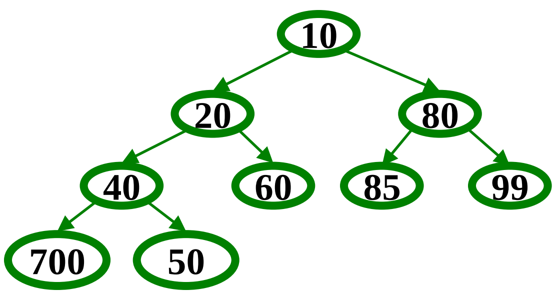
	10	20	80	40	60	85	99	700	50				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Note this pseudocode deletes ints,
not useful data with priorities

Pseudocode: deleteMin

```
int deleteMin() {
  if(isEmpty()) throw…
  ans = arr[1];
  hole = percolateDown
          (1,arr[size]);
  arr[hole] = arr[size];
  size--;
  return ans;
}
```

```
int percolateDown(int hole,
                  int val) {
 while(2*hole <= size) {
  left  = 2*hole;
  right = left + 1;
  if(arr[left] < arr[right]
     || right > size)
    target = left;
  else
    target = right;
  if(arr[target] < val) {
    arr[hole] = arr[target];
    hole = target;
  } else
      break;
 }
 return hole;
}
```

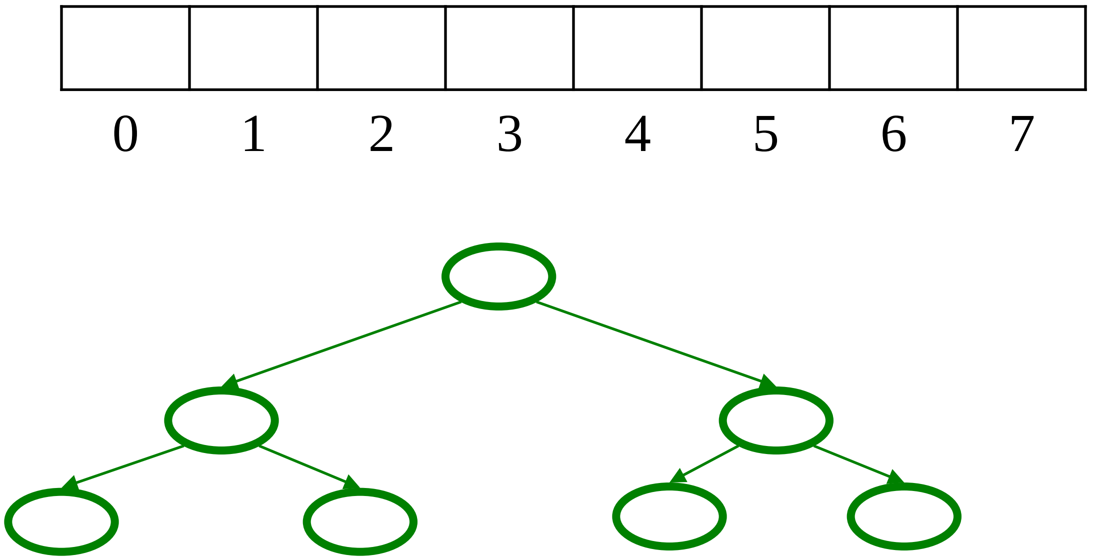


O(logn)

	10	20	80	40	60	85	99	700	50				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Example

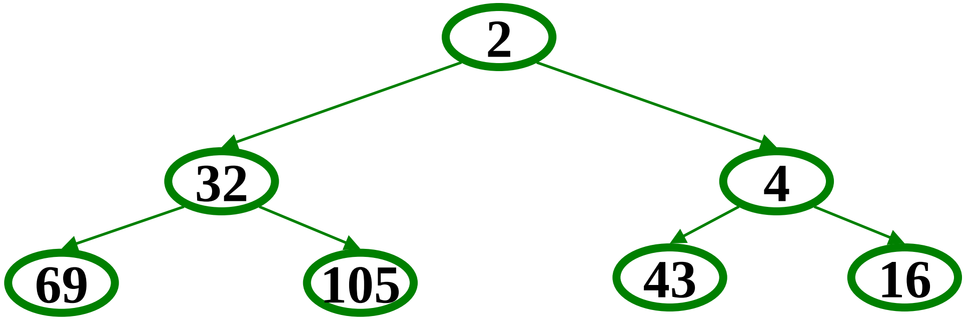
1. insert: 16, 32, 4, 69, 105, 43, 2
2. deleteMin



Example: After insertion

1. insert: 16, 32, 4, 69, 105, 43, 2
2. deleteMin

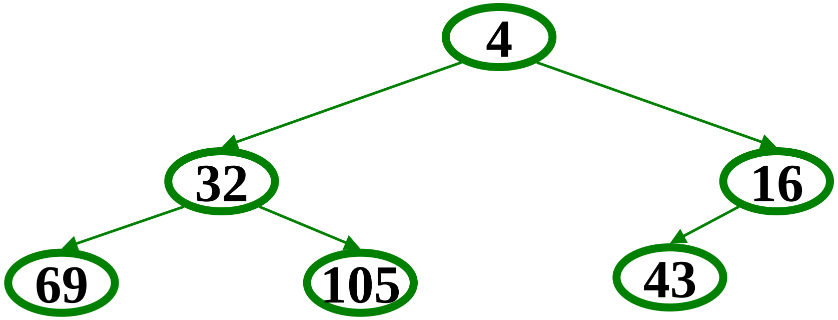
	2	**32**	**4**	**69**	**105**	**43**	**16**
0	1	2	3	4	5	6	7



Example: After deletion

1. insert: 16, 32, 4, 69, 105, 43, 2
2. deleteMin

	4	**32**	**16**	**69**	**105**	**43**	
0	1	2	3	4	5	6	7



Other operations

- **`decreaseKey`**: given pointer to object in priority queue (e.g., its array index), lower its priority value by *p*
  - Change priority and percolate up **O(logn)**

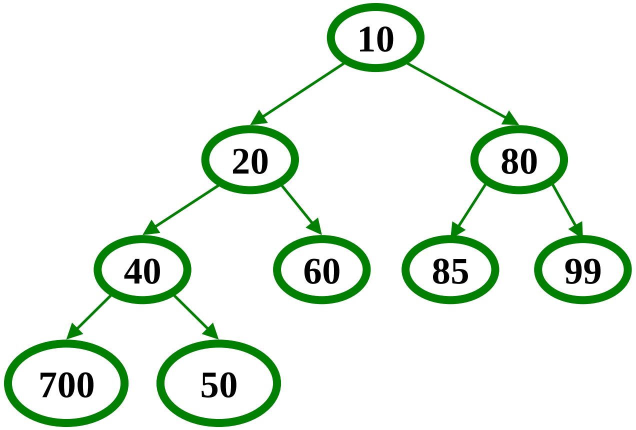
- **`increaseKey`**: given pointer to object in priority queue (e.g., its array index), raise its priority value by *p*
  - Change priority and percolate down **O(logn)**

- **`remove`**: given pointer to object, take it out of the queue
  - **`decreaseKey`**: set to -∞, then **`deleteMin`** **O(logn)**

Running time for all these operations?

Insert run-time: Take 2

- Insert: Place in next spot, percUp
- How high do we expect it to go?
- Aside: Complete Binary Tree
  - Each full row has 2x nodes of parent row
  - $1+2+4+8+\ldots+2^k= 2^{k+1}-1$
  - Bottom level has ~1/2 of all nodes
  - Second to bottom has ~1/4 of all nodes



- PercUp Intuition:
  - Move up if value is less than parent
  - Inserting a random value, likely to have value not near highest, nor lowest; somewhere in middle
  - Given a random distribution of values in the heap, bottom row should have the upper half of values, 2^{nd} from bottom row, next 1/4
  - Expect to only raise a level or 2, even if h is large
- Worst case: still O(logn)
- Expected case: O(1)
- Of course, there's no guarantee; it may percUp to the root

Build Heap

- Suppose you started with *n* items to put in a new priority queue
  - Call this the `buildHeap` operation

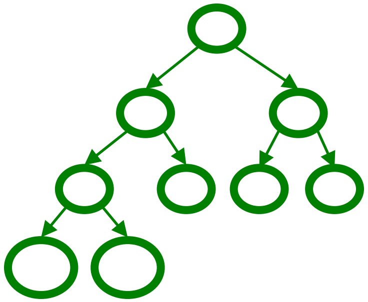
- `create`, followed by *n* `insert`s works
  - Only choice if ADT doesn't provide `buildHeap` explicitly
  - $O(n \log n)$

- Why would an ADT provide this unnecessary operation?
  - Convenience
  - Efficiency: an $O(n)$ algorithm called Floyd's Method

Floyd's Method

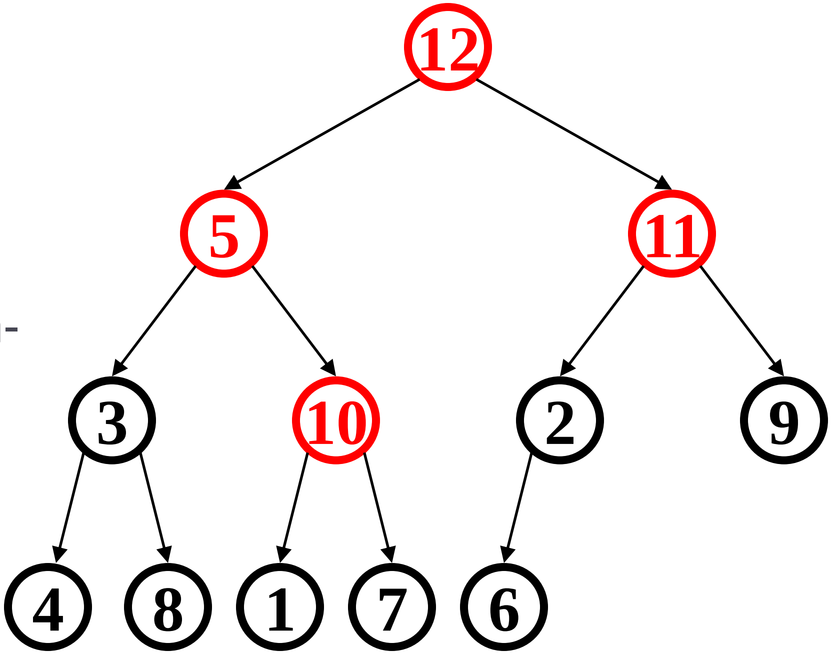
1. Use *n* items to make any complete tree you want
   - That is, put them in array indices 1,…,*n*
2. Treat it as a heap by fixing the heap-order property
   - Bottom-up: leaves are already in heap order, work up toward the root one level at a time

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

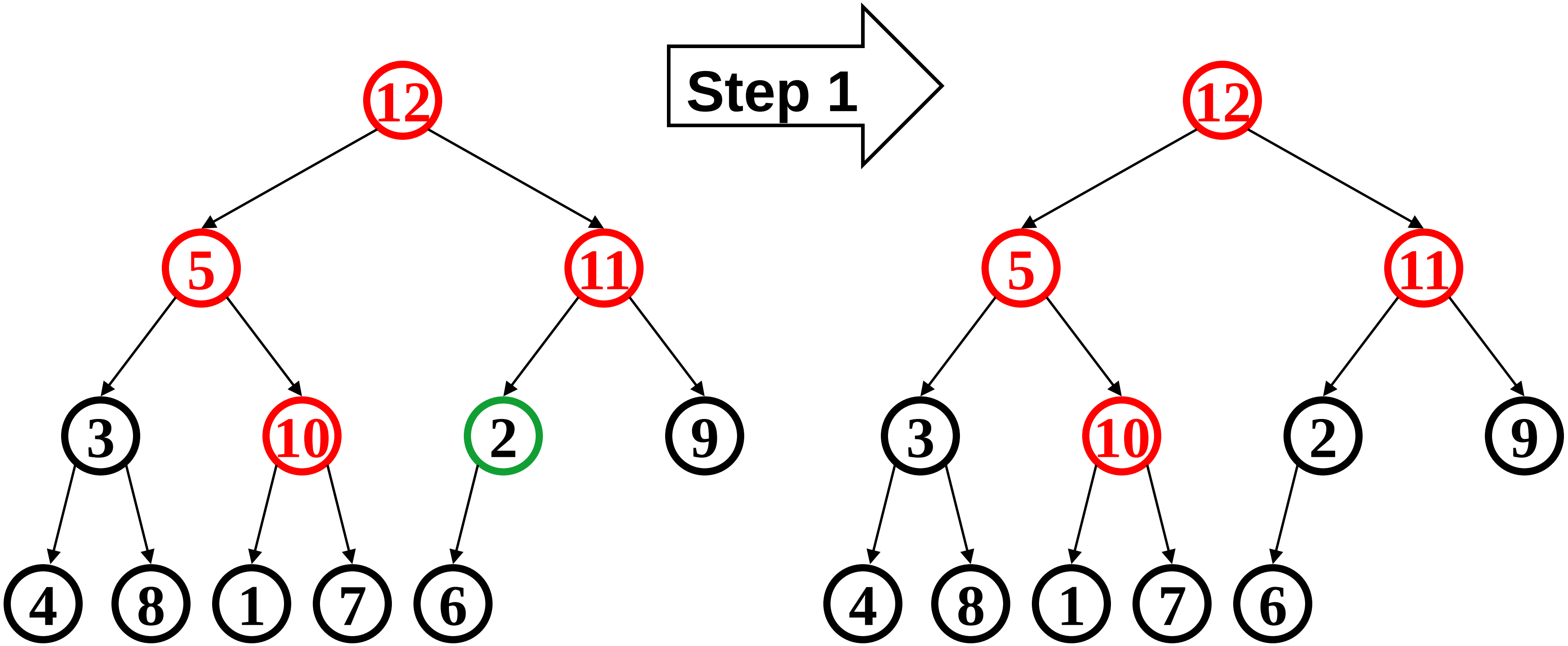


Example

- Say we start with
- [12,5,11,3,10,2,9,4,8,1,7,6]
- In tree form for readability
  - Red for node not less than descendants
    - Heap-order violation
  - Notice no leaves are red
  - Check/fix each non-leaf bottom-up (6 steps here)

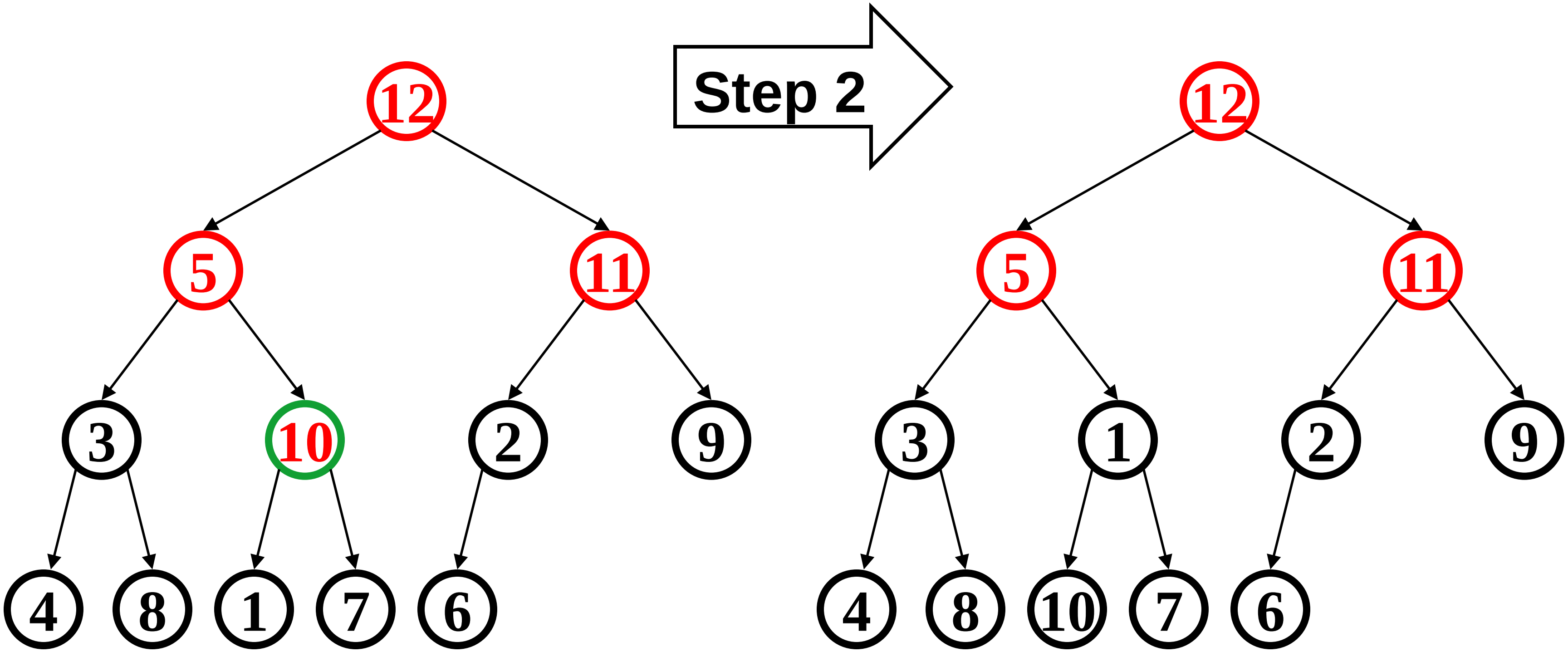


Example



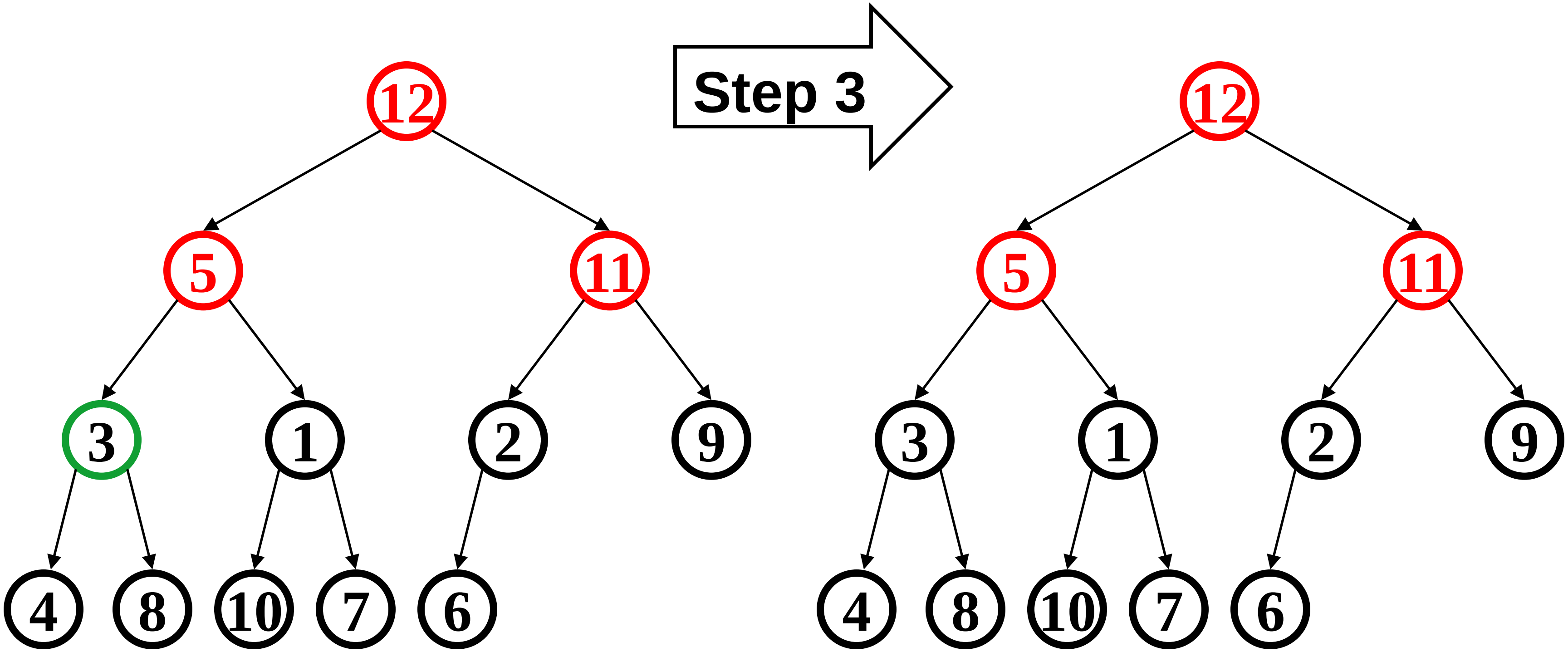
- Happens to already be less than children (er, child)

Example



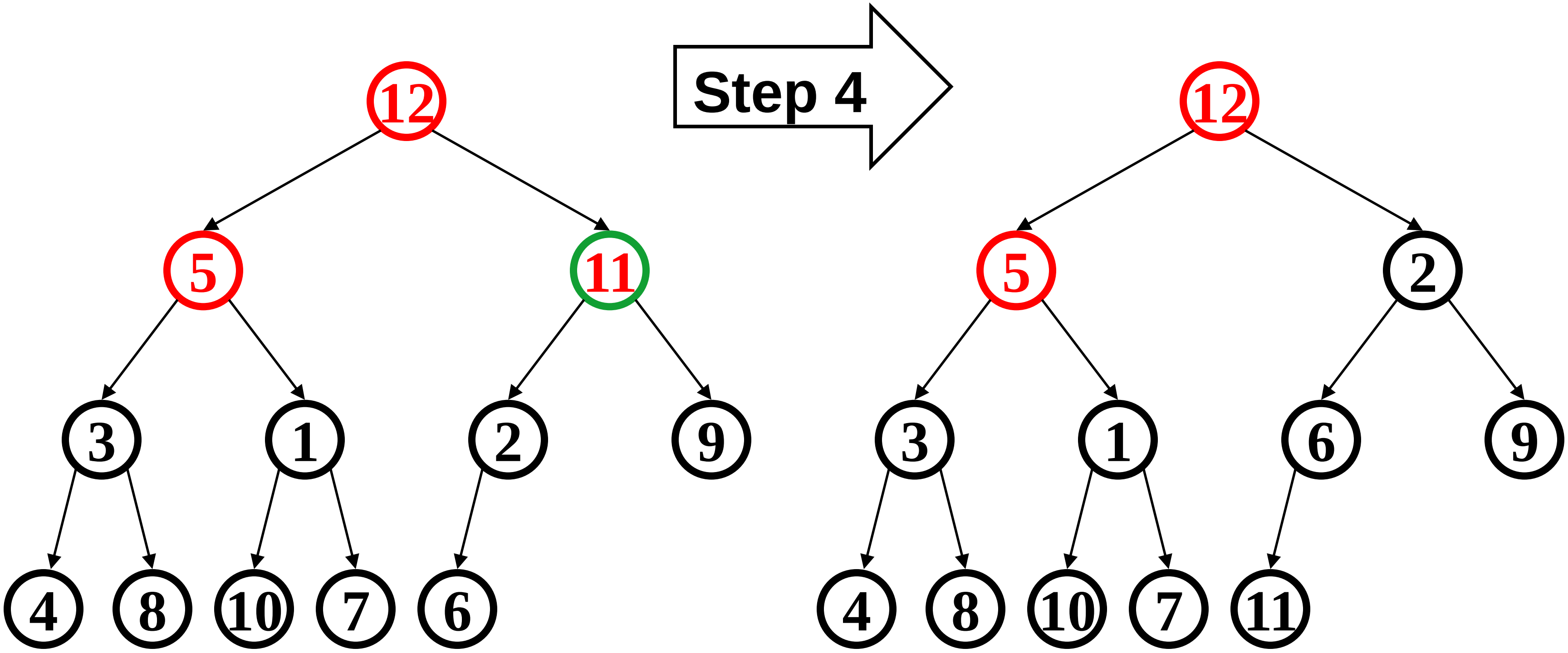
- Percolate down (notice that moves 1 up)

Example



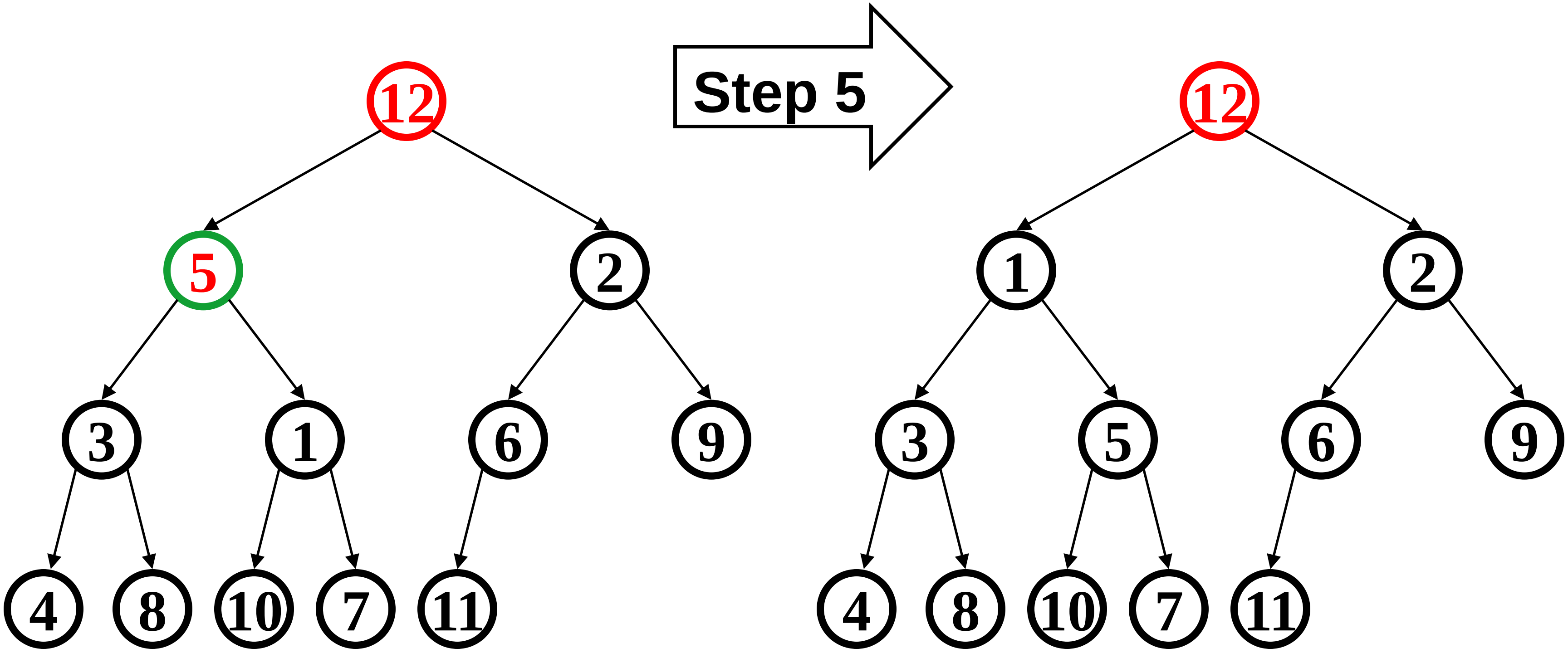
- Another nothing-to-do step

Example

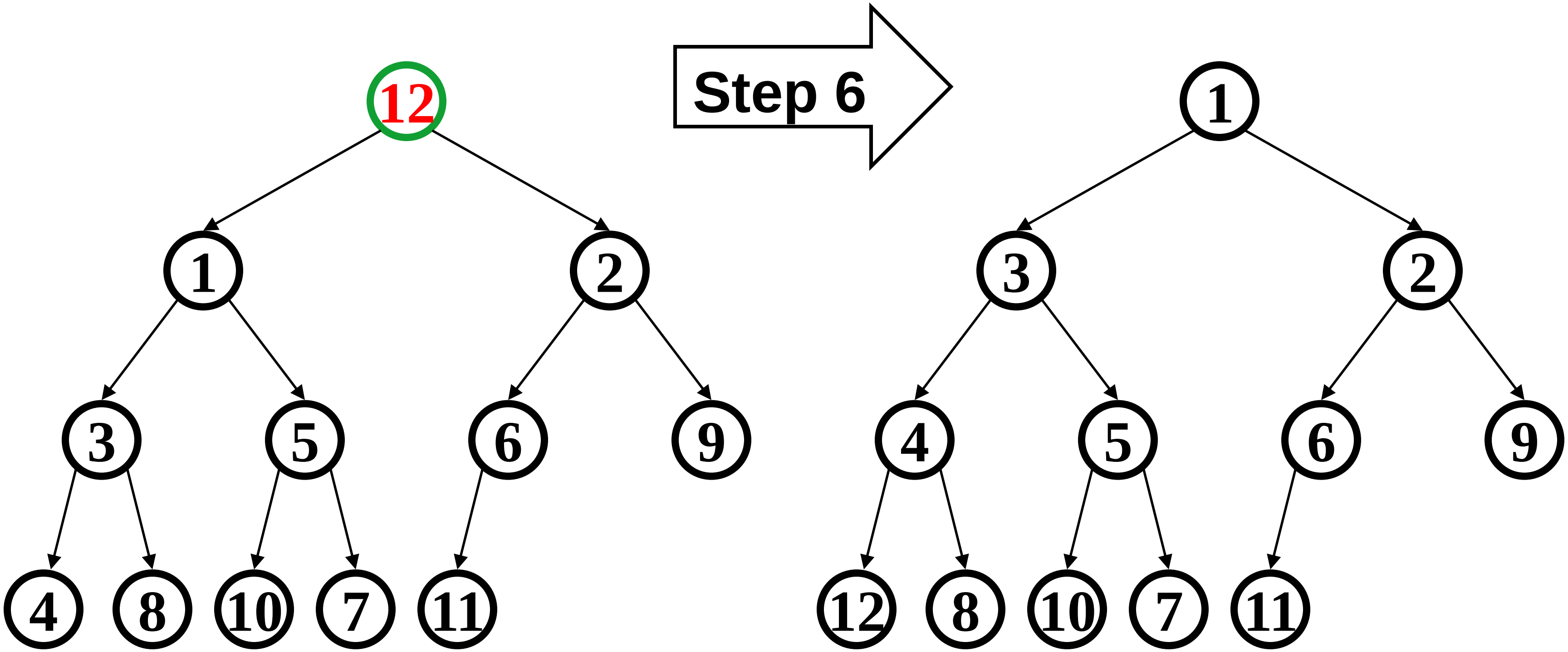


- Percolate down as necessary (steps 4a and 4b)

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: Equivalent to the heap ordering property

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 $\theta(n\ \texttt{log}\ n)$ worst case
  - Worst case is inserting lower priority values later
- By providing a specialized operation internally (with access to the data structure), 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 $O(n\ \texttt{log}\ n)$
    - A "tighter" analysis shows same algorithm is $O(n)$

What we're skipping (see text if curious)

- *d*-heaps: have *d* children instead of 2
  - Makes heaps shallower
    - Approximate height of a complete d-ary tree with n nodes?
    - How does this affect the asymptotic run-time (for small d's)?
  - Useful for huge tree data structures that are too large to fit in memory; accessing a node will require accessing the hard-drive (incredibly slow) – limit nodes accessed: B-Trees
- Aside: How would we do a 'merge' for 2 binary heaps?
  - Answer: Slowly; have to buildHeap; O(n) time
    - Will always have to copy over data from one array
  - Different data structures for priority queues that support a logarithmic time `merge` operation (impossible with binary heaps)
    - Leftist heaps, skew heaps, binomial queue: Insert & deleteMin defined in terms of merge
  - Special case: How might you merge binary heaps if one heap is much smaller than the other?