CSE332: Data Abstractions

Lecture 5: Heaps

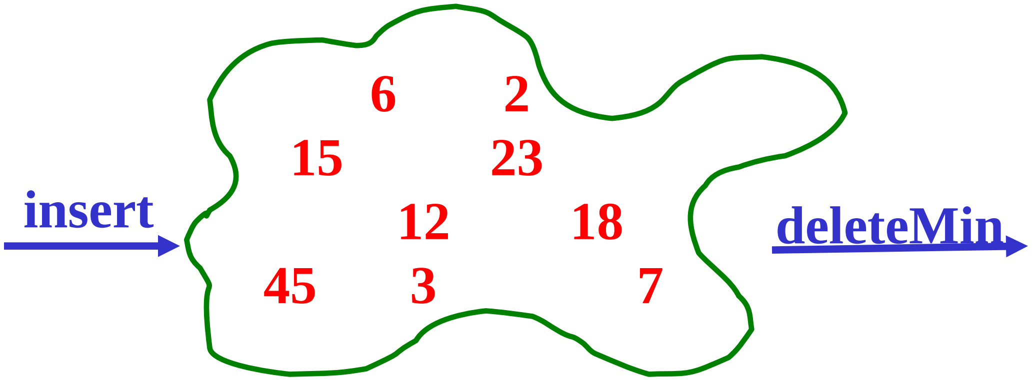
James Fogarty

Winter 2012

ADT: Priority Queue

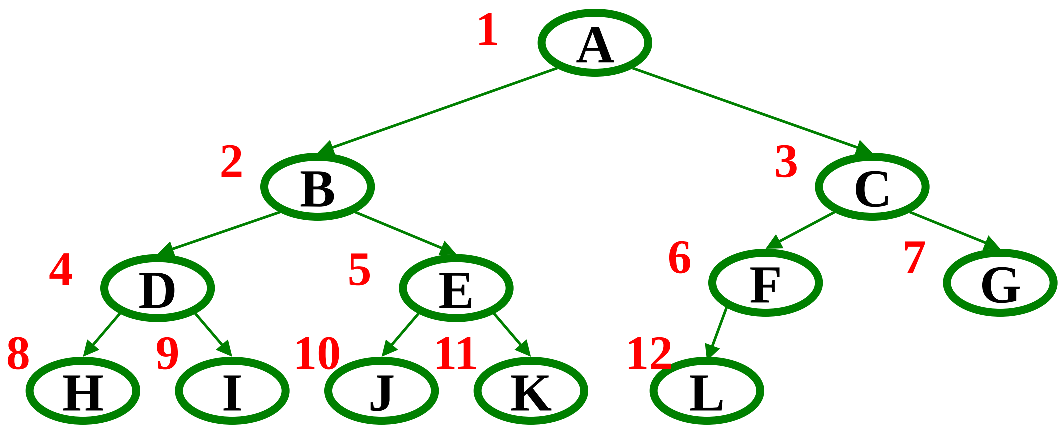
- Each item has a "priority"
  - The *next* or *best* item is the one with the *lowest* priority
  - So "priority 1" should come before "priority 4"
  - Simply by convention, could also do maximum priority

- Operations:
  - **`insert`**
  - **`deleteMin`**



- **`deleteMin`** *returns* and *deletes* item with lowest priority
  - Can resolve ties arbitrarily

Array Representation of a Binary Heap



From node **`i`**:

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

wasting index 0 is convenient for the math

Array implementation:

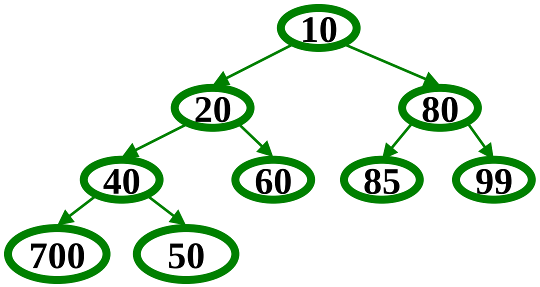
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Pseudocode: insert

This pseudocode uses ints. In real use, you will have data nodes 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;
}
```



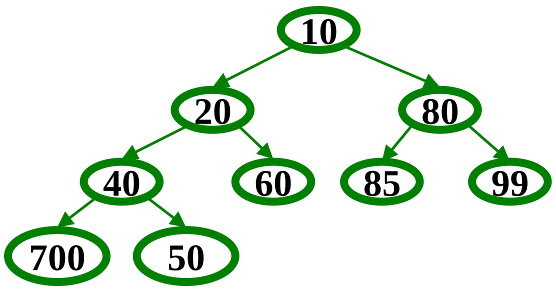
	10	20	80	40	60	85	99	700	50				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Pseudocode: deleteMin

This pseudocode uses ints. In real use, you will have data nodes with priorities.

```
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;
}
```

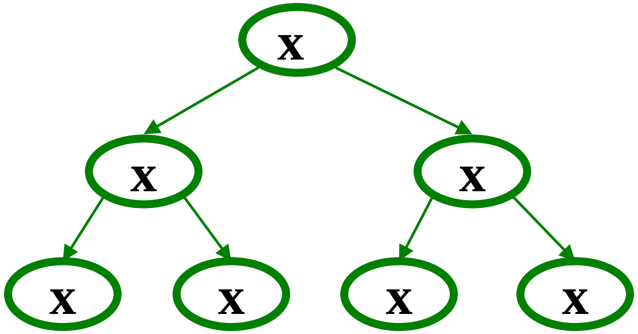


	10	20	80	40	60	85	99	700	50				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Example

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

0	1	2	3	4	5	6	7



Other Operations

What is the runtime?
O(log n)

- **`decreaseKey`**:
  - given pointer to object in priority queue (e.g., its array index), lower its priority to *p*
  - Change priority and percolate up
- **`increaseKey`**:
  - given pointer to object in priority queue (e.g., its array index), raise its priority to *p*
  - Change priority and percolate down
- **`remove`**:
  - given pointer to object in priority queue (e.g., its array index), remove it from the queue
  - **`decreaseKey`** to $p = -\infty$, then **`deleteMin`**

Build Heap

- Suppose you have *n* items to put in a new priority queue
  - Sequence of *n* **`insert`**s, *O*(*n* **`log`** *n*)

- Can we do better?
  - Above is only choice if ADT does not provide **`buildHeap`**

- Important issue in ADT design: how many specialized operations
  - Tradeoff: Convenience, Efficiency, Simplicity

- In this case, we are motivated by efficiency
  - We can **`buildHeap`** using *O*(*n*) algorithm called Floyd's Method

Floyd's Method

Recall our general strategy for working with the heap:

- Preserve structure property
- Break and restore heap property

1. Use our n items to make a complete tree
   - Put them in array indices 1,…,n

2. Treat it as a heap and fix the heap-order property
   - Exactly how we do this is where we gain efficiency

Floyd's Method

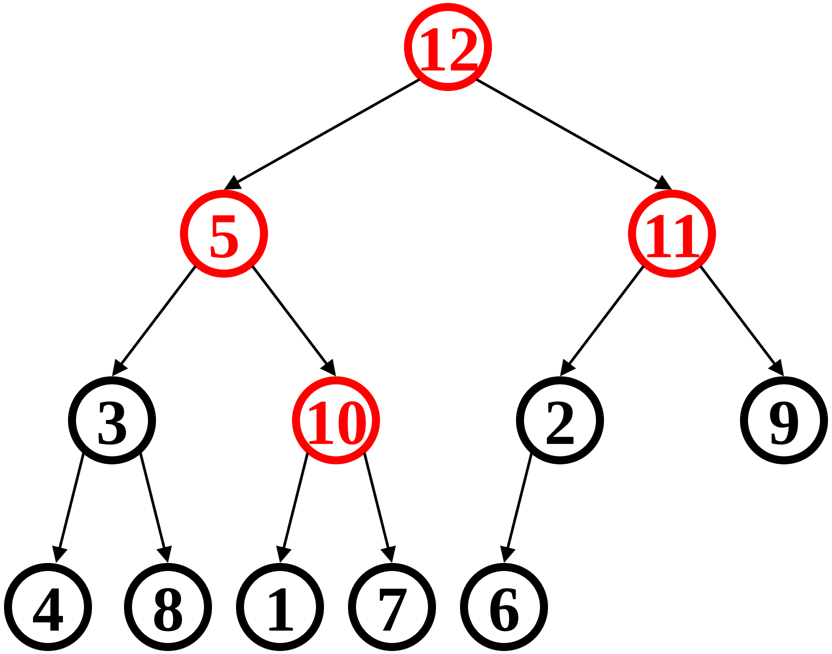
Bottom-up

- Leaves are already in heap order
- Work up toward the root one level at a time

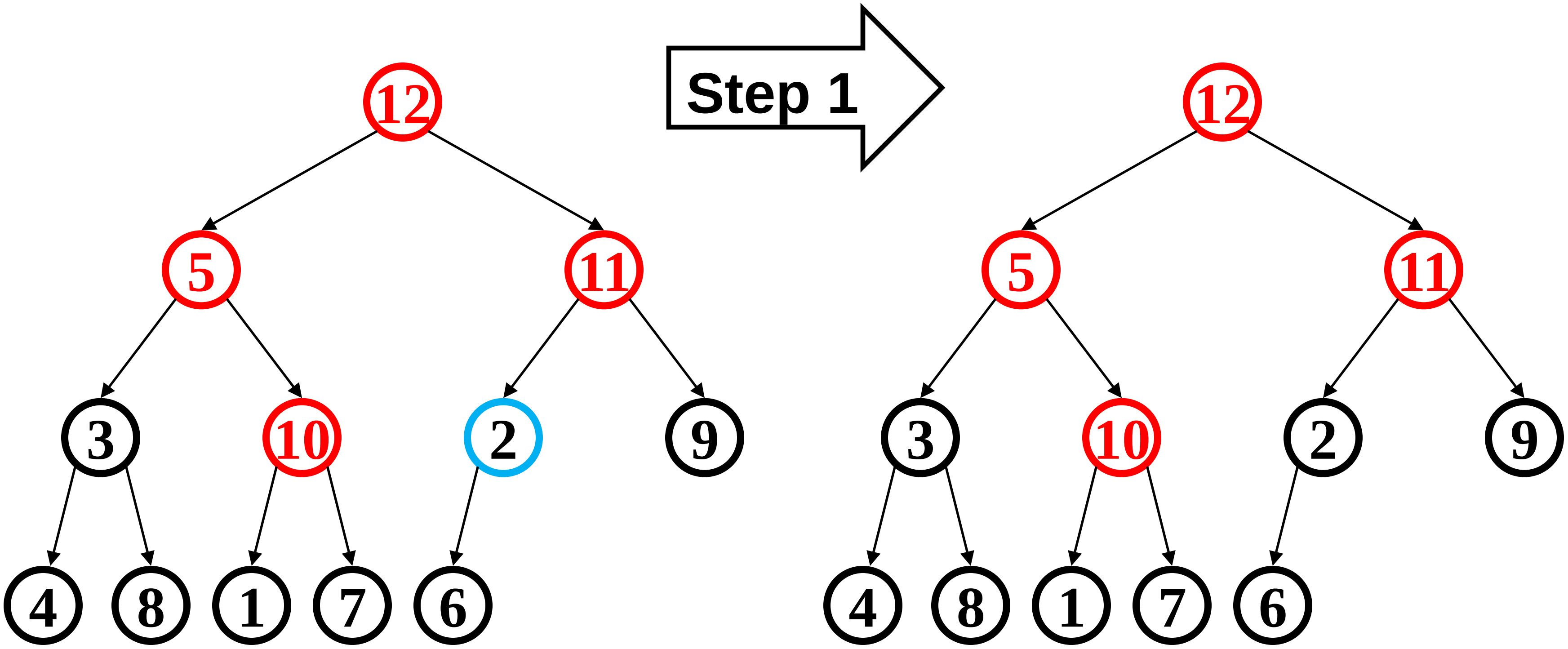
```
void buildHeap() {
   for(i = size/2; i>0; i--) {
     val  = arr[i];
     hole = percolateDown(i,val);
     arr[hole] = val;
   }
}
```

Example

- In tree form for readability
  - Red for nodes which are not less than descendants
  - Notice no leaves are red
  - Check/fix each non-leaf bottom-up (6 steps here)

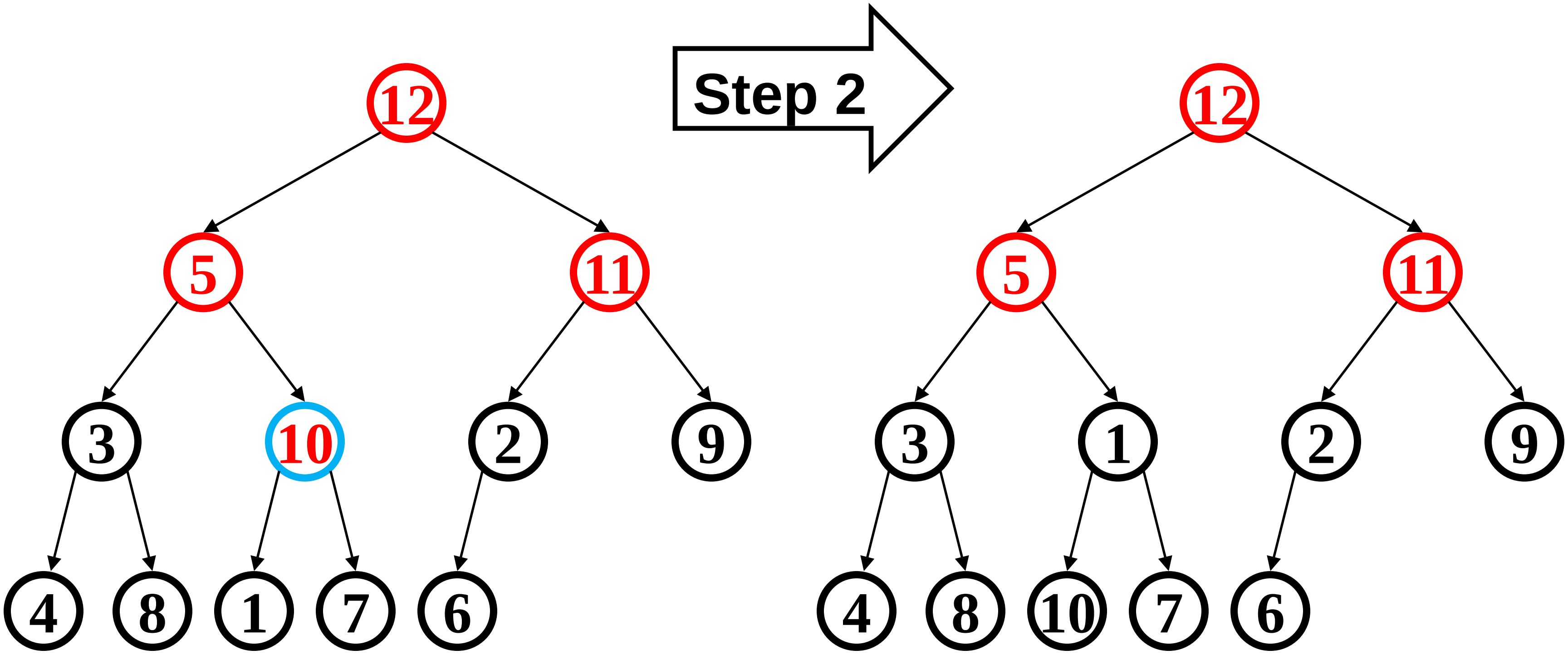


Example



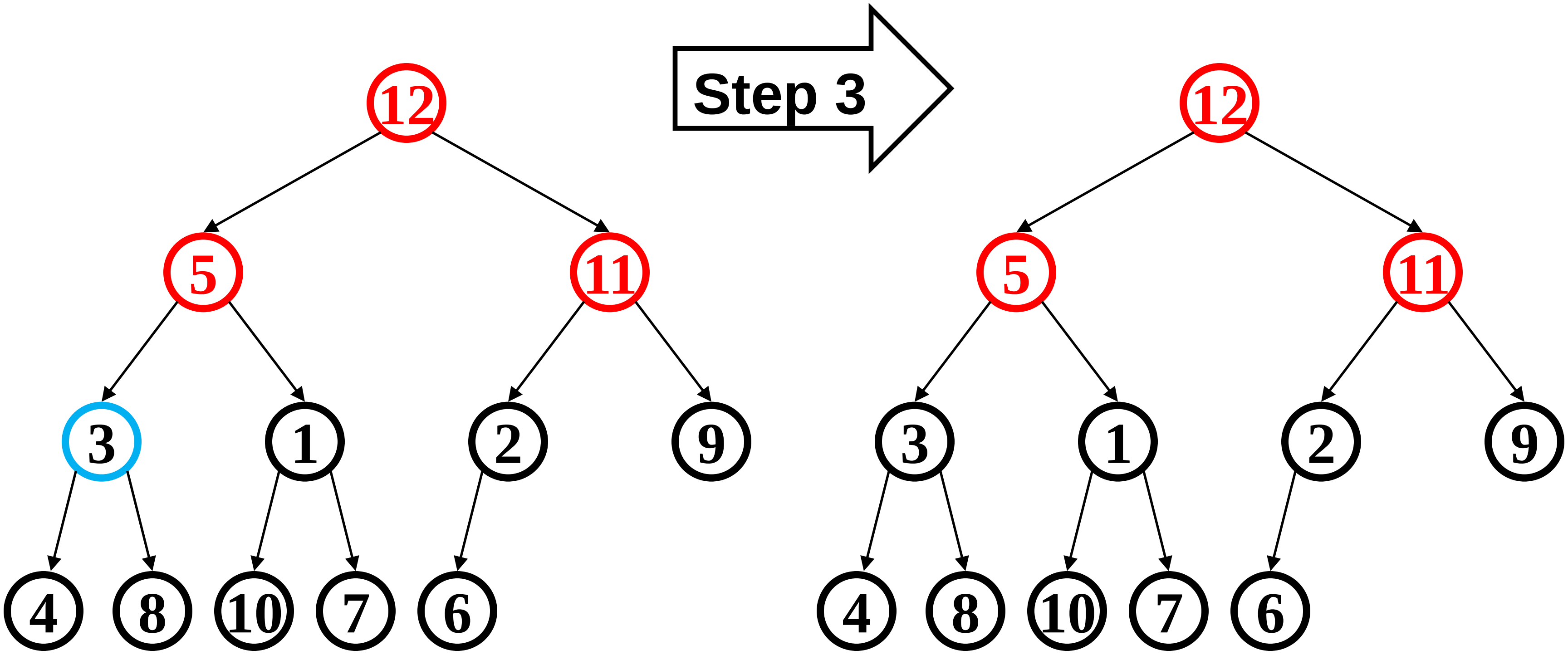
- Happens to already be less than children

Example



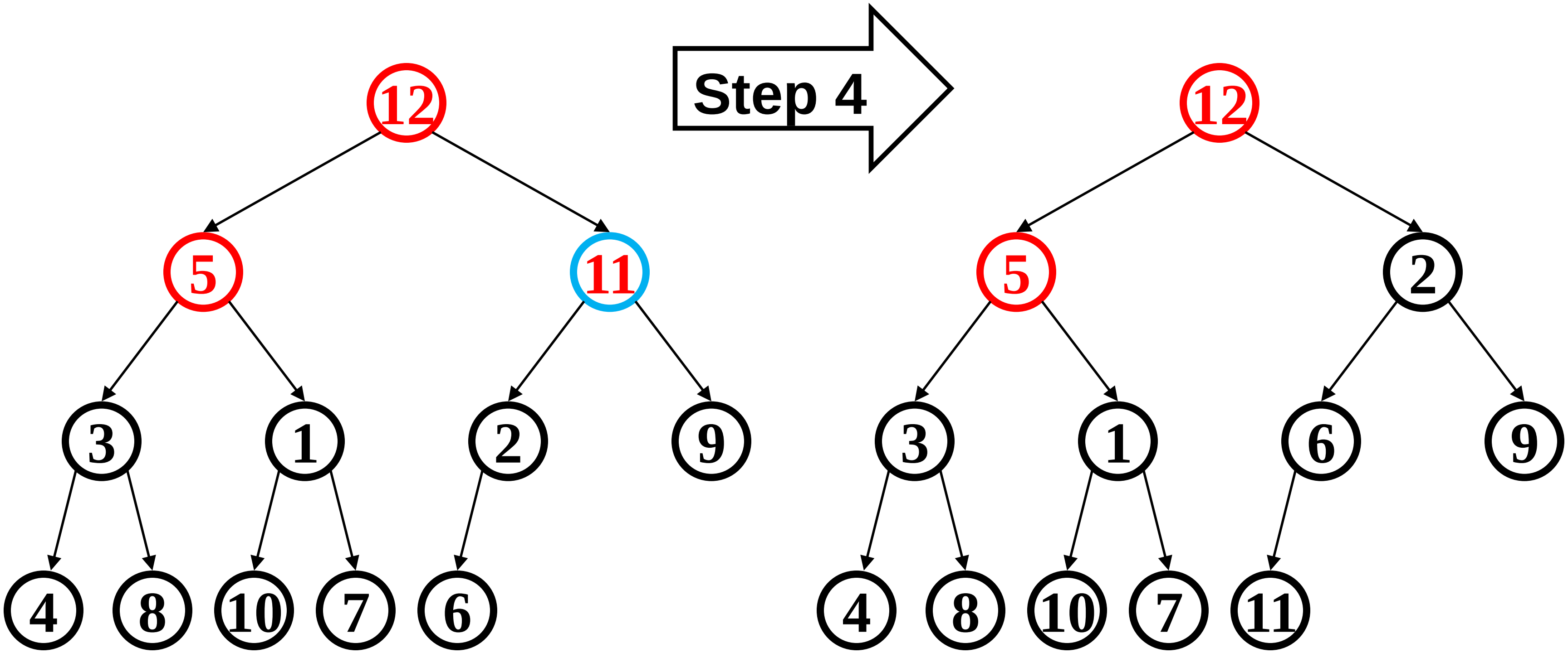
- 10 percolates down (and notice that 1 moves up)

Example



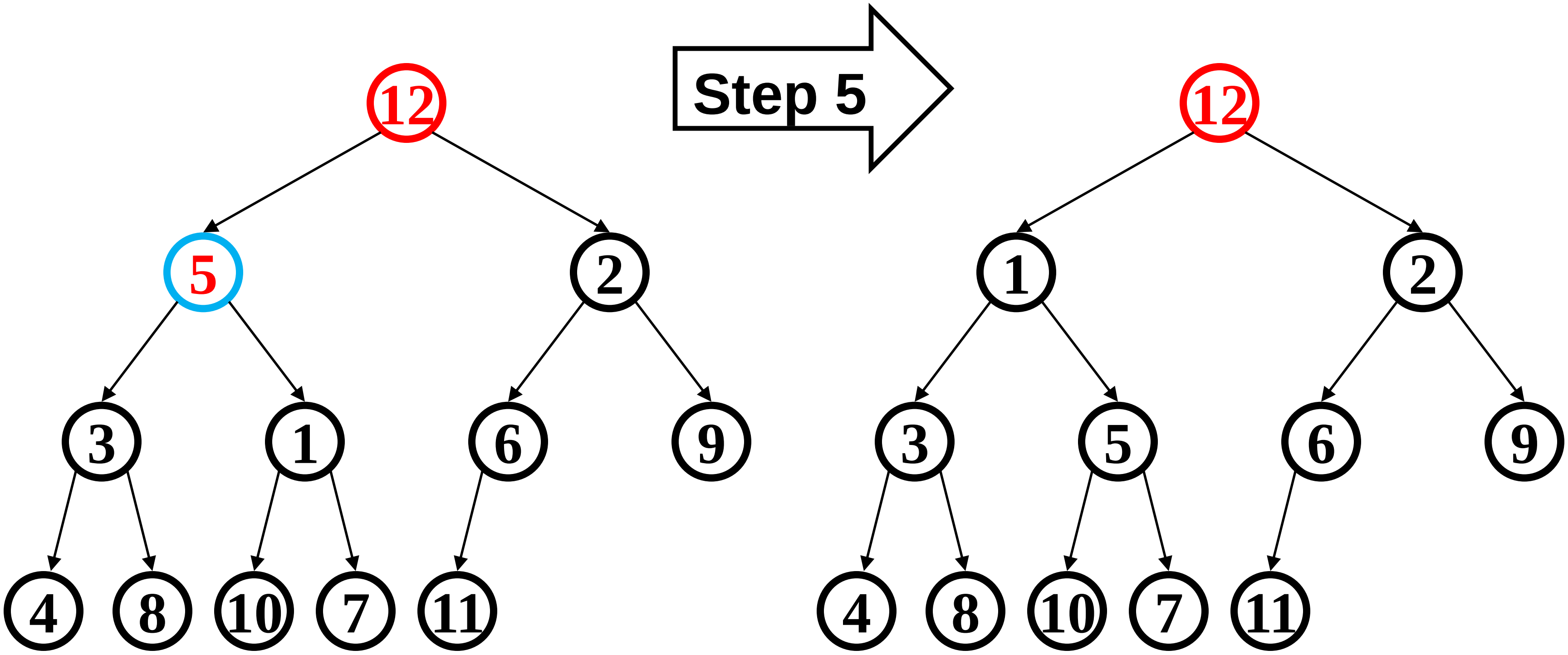
- Another nothing-to-do step

Example



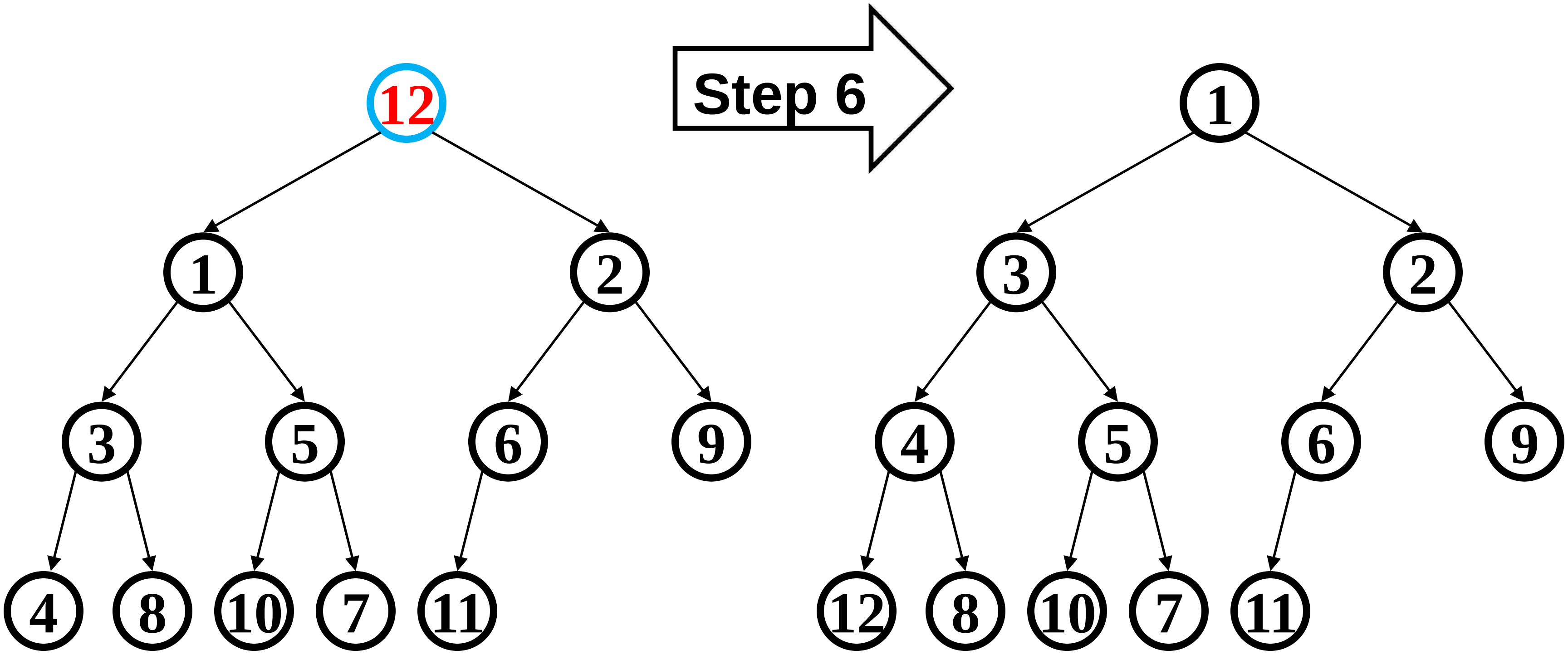
- Percolate down as necessary (first 2, then 6)

Example



- Percolate down as necessary (the 1 again)

Example



- Percolate down as necessary (first 1, then 3, then 4)

But is it right?

- "Seems to work"
  - First we will *prove* it restores the heap property (correctness)
  - Then we will *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 “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 allows clients to implement their own in worst-case $O(n$ **`log`** $n)$
  - Worst case is inserting lower 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)
    - A "tighter" analysis shows same algorithm is $O(n)$

What we are Skipping (see text if curious)

- *d*-heaps: have *d* children instead of 2
  - Makes heaps shallower, useful for heaps too big for memory
  - The same issue arises for balanced binary search trees and we *will* study "B-Trees"

- **`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)