More Heaps

Data Structures and Parallelism

Logistics

Gitlab should run the tests and a "static analysis" every time you push your P1 code.

- Let us know if that's not happening.

The spec for stack says operations should take "amortized O(1) time" You will meet this requirement if you:

- double the size of the array when it fills up
- Take O(n) time to resize
- And take O(1) time when the array isn't full

Lecture on Wednesday will explain "amortized" time fully.

Everyone should now have GradeScope access, will be how you turn in Exercises and Project Writeups

Outline

More Heaps

- Some more operations
- Building a heap all at once

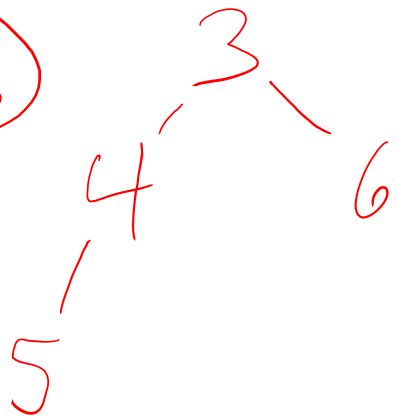
More Algorithm Analysis!

Heap Refresher

Are these binary min heaps?

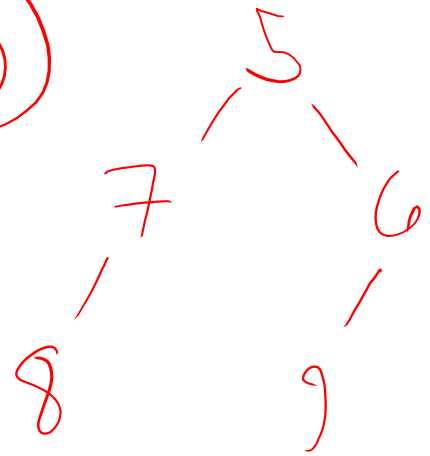
A

B



C

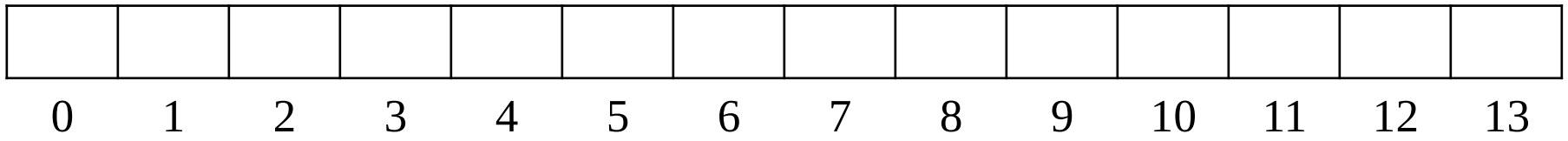
D



Worked Example

Insert **16, 32, 4, 69, 105, 43, 2** into empty heap

Then deleteMin

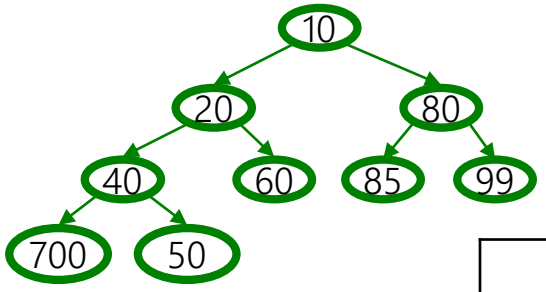


insert Pseudocode

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

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

Note: These are counting from 1, project 1 and exercise 1 count from 0



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

deleteMin Pseudocode

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

Note: These are counting from 1,

project 1 and exercise 1 count from 0

```
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;
}
```

Insert Gets Better

Insert's worst case is O(log n)

The trick is that it all depends on the order the items are inserted (What is the worst case order?)

Experimental studies of randomly ordered inputs shows the following:

-Average 2.607 comparisons per insert
(# of percolation passes)

-An element usually moves up 1.607 levels

deleteMin is average O(log n)

-Moving a leaf to the root usually requires re-percolating that value back to the bottom

More Operations

Let's do more things with heaps! Assuming we have access insert, deleteMin, percolateUp, and percolateDown how to we make:

IncreaseKey(element,priority) Given a pointer to an element of the heap and a new, larger priority, update that object's priority.

-Change priority value of given element, then percolate down

DecreaseKey(element,priority) Given a pointer to an element of the heap and a new, smaller priority, update that object's priority.

-Change priority value of given element, then percolate up

More Operations

Delete(element) Given a pointer to an element of the heap, remove that element.

-DecreaseKey to lowest possible, then call deleteMin

Needing a pointer to the element for these operations isn't normal.

Exercise02 will have you think more about why we need it here.

Even More Operations

BuildHeap(elements $e_1, \ldots, e_n$) – Given n elements, create a heap containing exactly those n elements.

Try 1: Just call insert n times.

Worst case running time?

n calls, each worst-case $\Theta(\log n)$. $O(n \log n)$ for the whole thing then

Can We Do Better?

What's bottleneck of the n insert strategy?

Most nodes are near the bottom, and we make them all go all the way up.

We need a new strategy beyond just calling insert a bunch of times.

Heap structure is easy to fill using an array, let's just stuff the input elements into a heap array and swap them around from there.

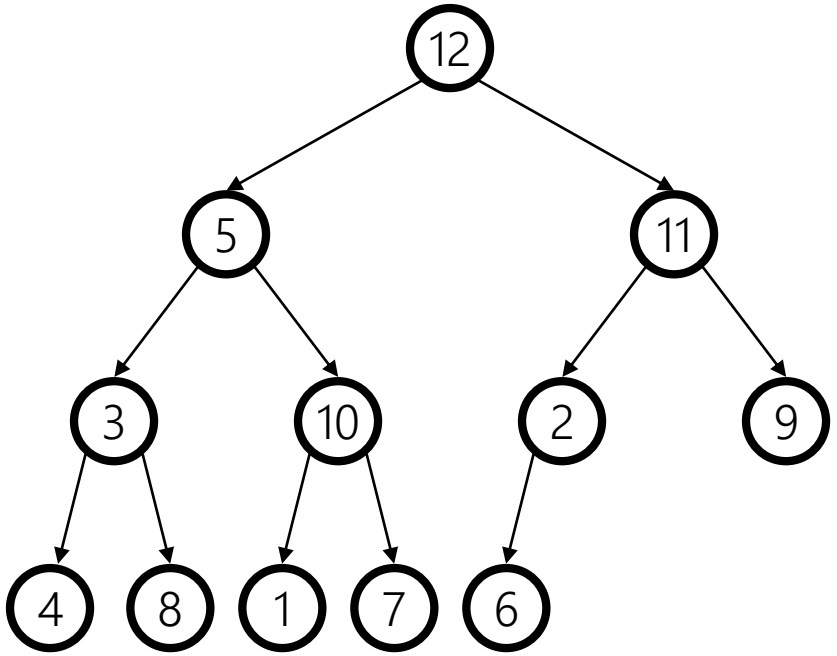
Thinking about buildHeap

Say we are given these elements to build a heap

-[12,5,11,3,10,2,9,4,8,1,7,6]

What do we do with these?

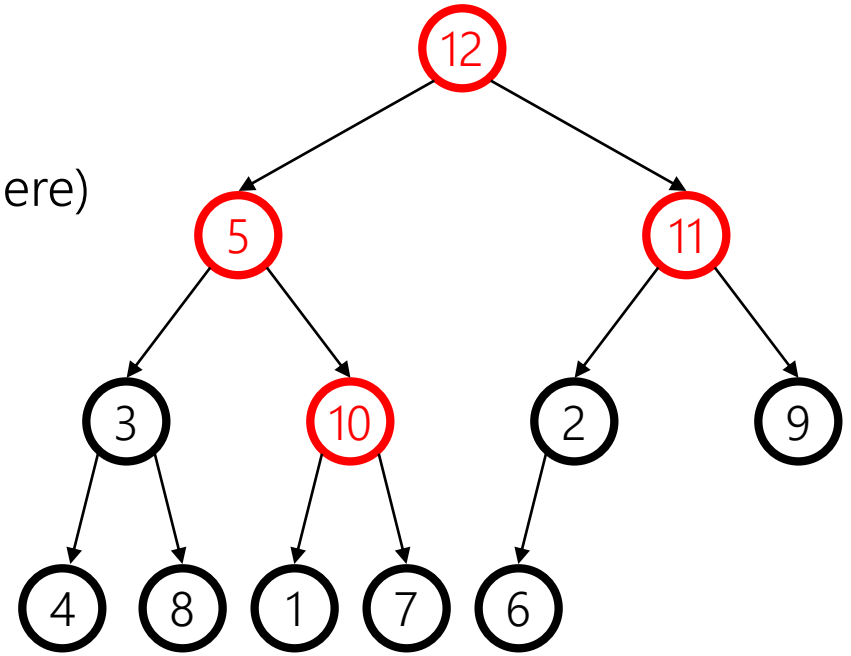
-Percolate down? Percolate up?



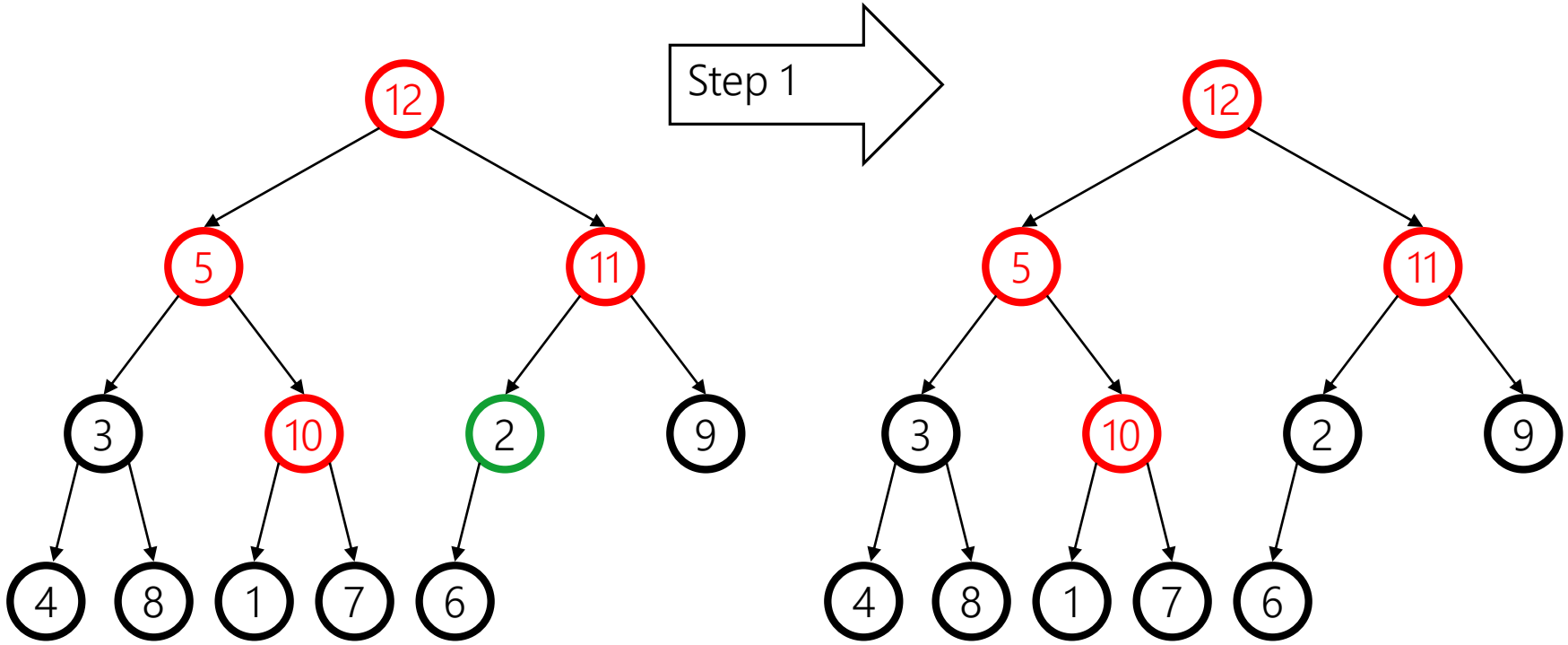
`buildHeap` Example

Color tree for readability

- Red for node not less than descendants
  - heap-order problem
- Notice no leaves are red
- Check/fix each non-leaf bottom-up (6 steps here)

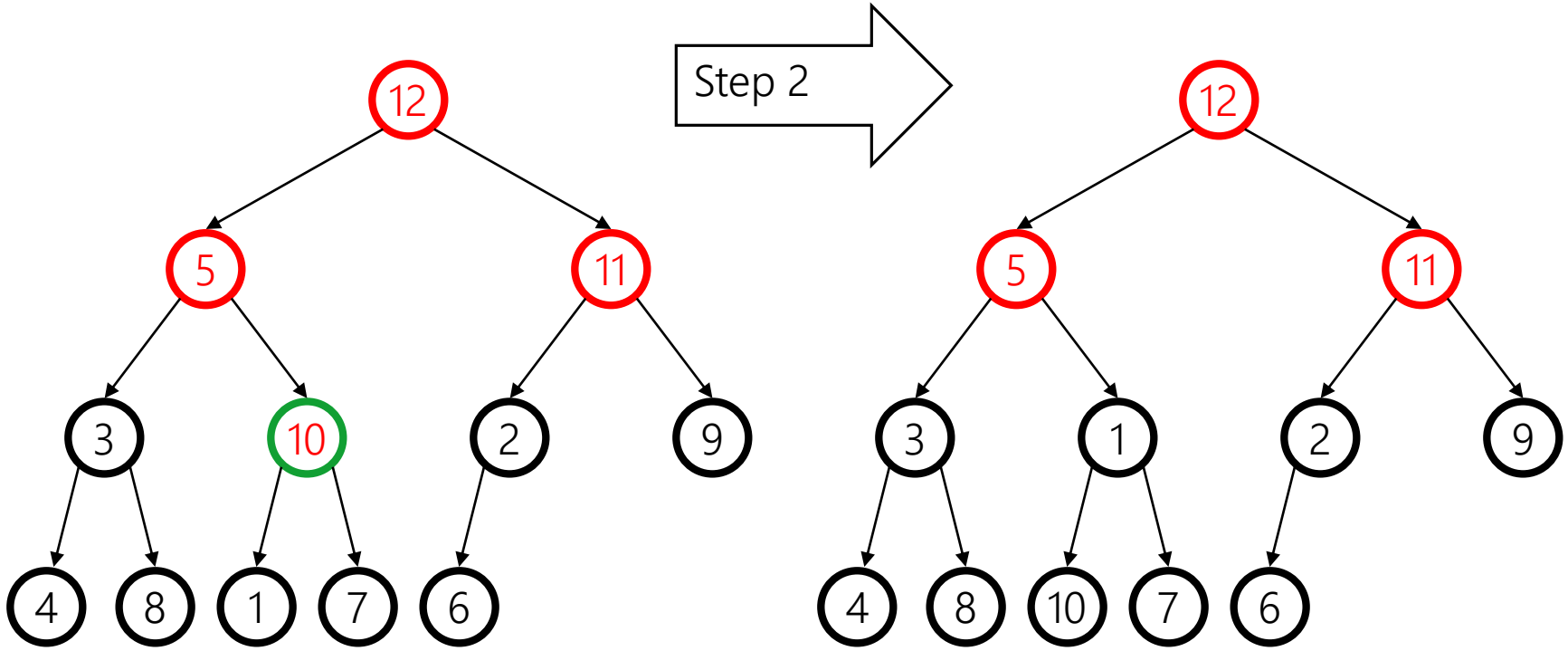


`buildHeap` Example



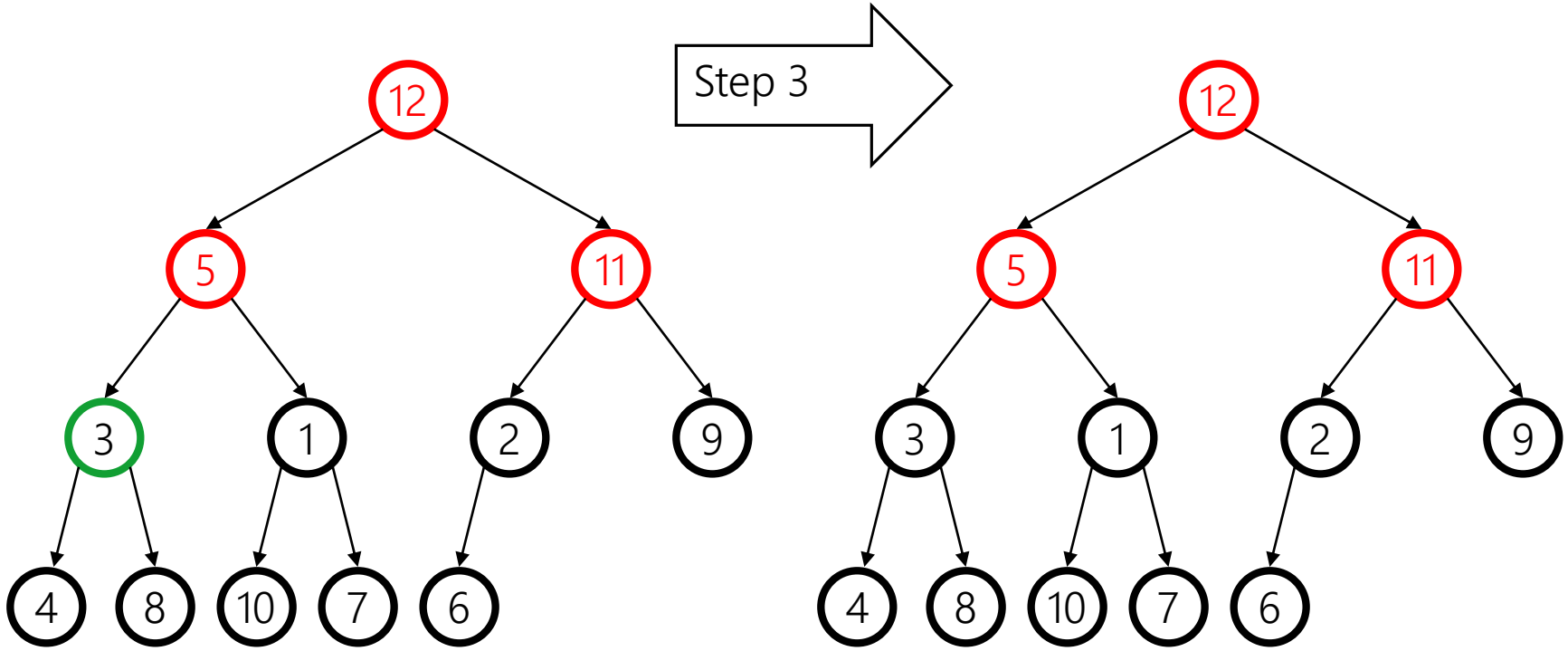
Happens to already be less than child

`buildHeap` Example



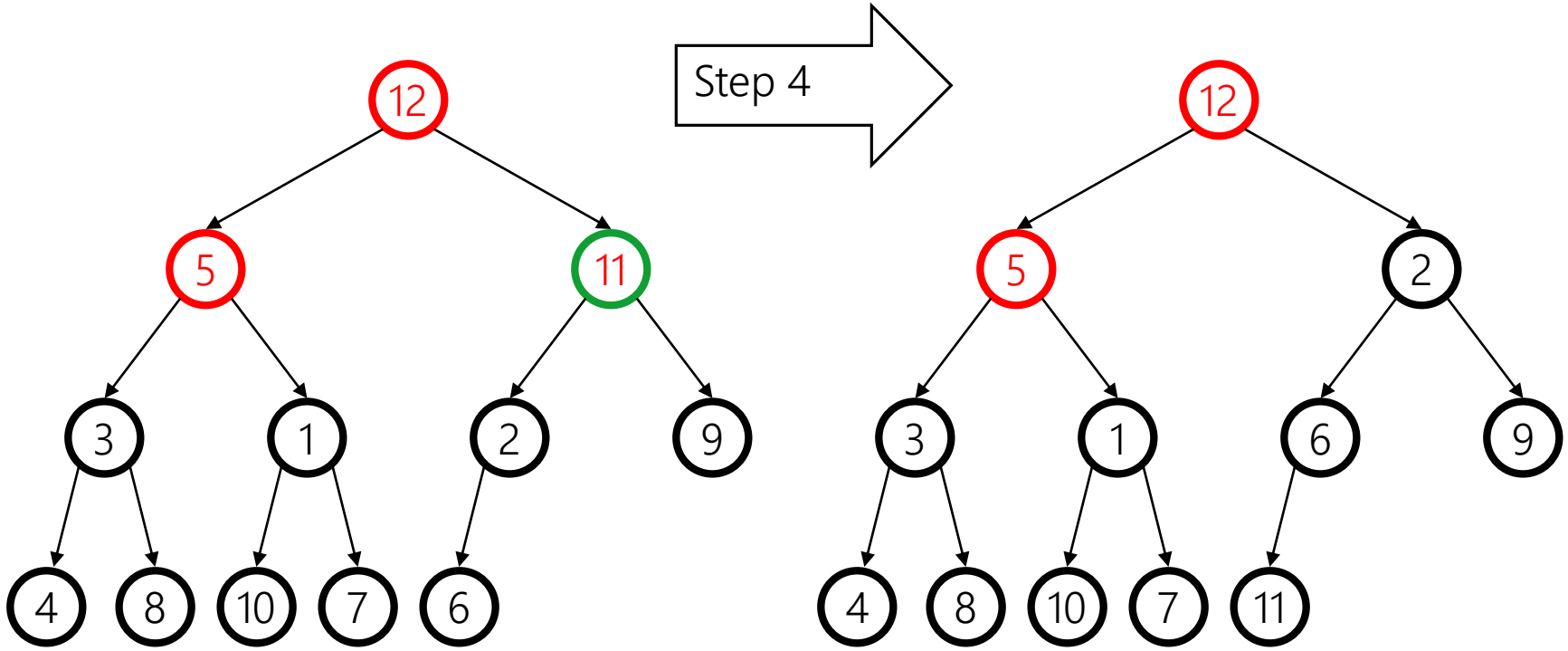
Percolate down (notice that moves 1 up)

`buildHeap` Example



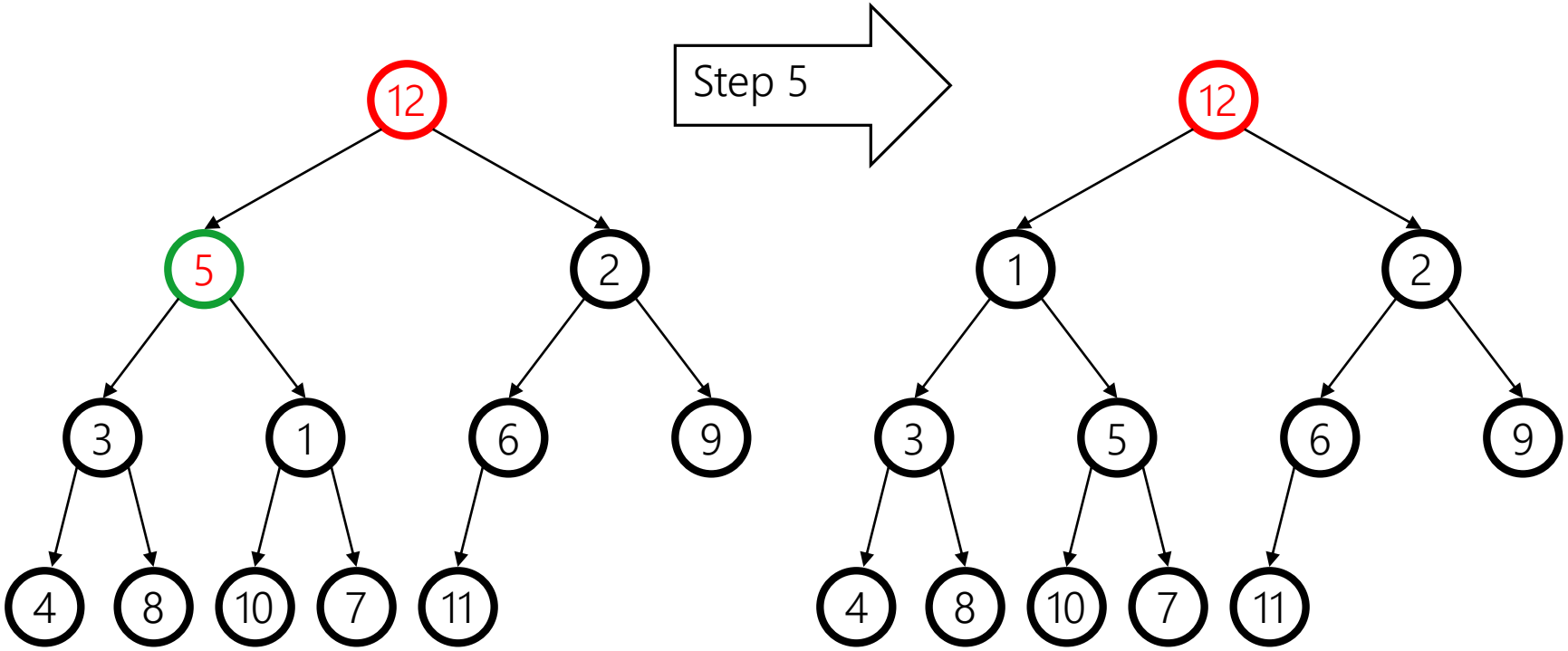
Another nothing-to-do step

`buildHeap` Example

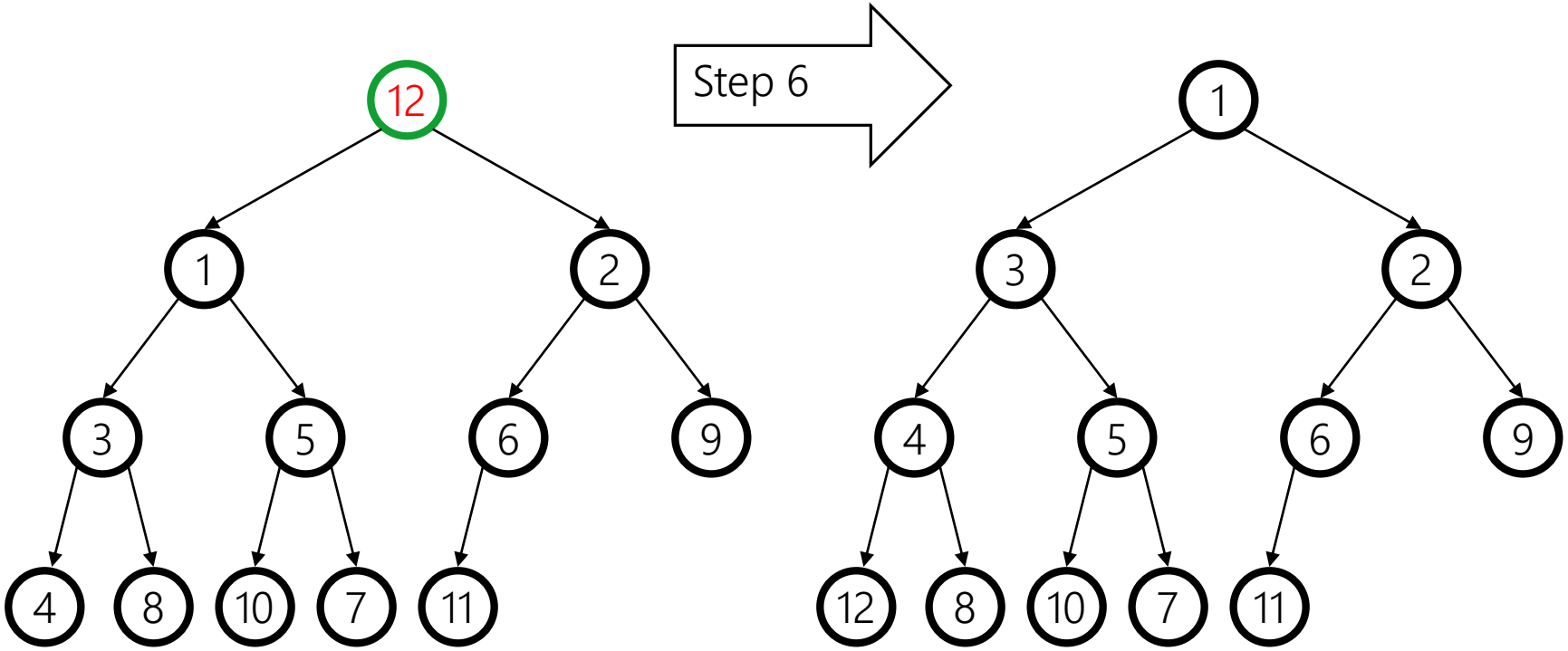


Percolate down as necessary (steps 4a and 4b)

`buildHeap` Example

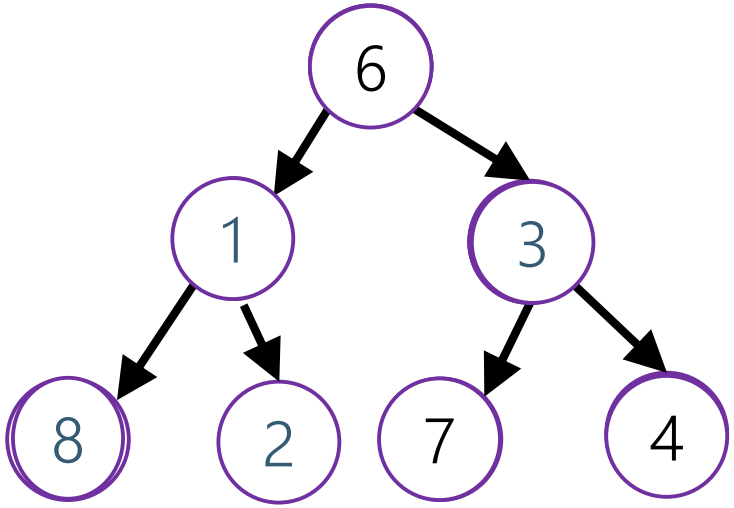


`buildHeap` Example



By the way…

If you run starting from the top of this heap, it will fail.



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

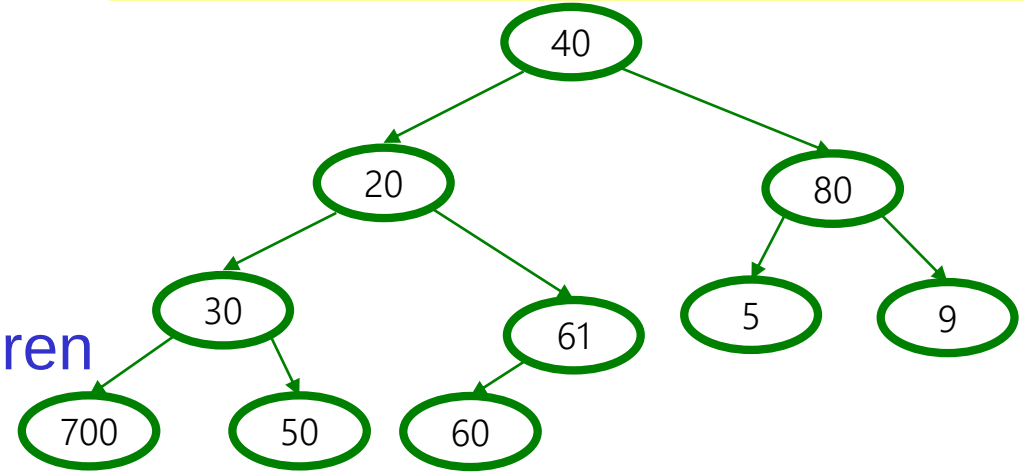
Loop Invariant:

For all `j>i`, `arr[j]` is less than its children

- True initially:
  If `j > size/2`, then `j` is a leaf
- 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

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



	40	20	80	30	61	5	9	700	50	60			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

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… etc.

((1/2) + (2/4) + (3/8) + (4/16) + (5/32) + …) = 2 (page 4 of Weiss)

- So at most **`2(size/2)`** *total* percolate steps: $O(n)$
- Also see Weiss 6.3.4, sum of heights of nodes in a perfect tree

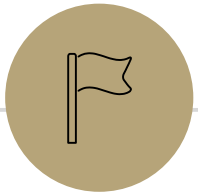
What we're skipping (see text if curious)

***d*-heaps**: have *d* children instead of 2 (Weiss 6.5)

- Makes heaps shallower, useful for heaps too big for memory
- How does this affect the asymptotic run-time (for small d's)?
- **Leftist heaps, skew heaps, binomial queues** (Weiss 6.6-6.8)
- Different data structures for priority queues that support a logarithmic time **`merge`** operation (impossible with binary heaps)
- **`merge:`** given two priority queues, make one priority queue
- Insert & deleteMin defined in terms of merge

Aside: How might you merge *binary* heaps:

- If one heap is much smaller than the other?
- If both are about the same size?



Analyzing Recursive Code

Calculating Running Times

Here's some code for calculating the length of a linked list:

What's its running time?

```
Length(Node curr){
    if(curr.next == null)
        return 1;
    return 1 + Length(curr.next);
}
```

We can analyze all the "non-recursive" work like usual

What about the recursive work?

Writing a Recurrence

If the function runs recursively, our formula for the running time should probably be recursive as well.

Such a formula is a **recurrence**.

$$T(n) = \begin{cases} T(n-1) + 2 & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

What does this say?

The input to T is the size of the input to the Length.

If the input to T() is large, the running time depends on the recusive call.

If not we can just use the base case.

Another example

```
Mystery(int n){
    if(n == 1)
        return 1;
    for(int i=0; i < n*n; i++){
        for(int j = 0; j < n; j++){
            System.out.println("hi!");
        }
    }
    return Mystery(n/2)
}
```

$$T(n) = \begin{cases} T(n/2) + n^3 + n^2 + 1 & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

Try It On Your Own

```
Mystery(int n){
    if(n <= 4)
        return 1;
    for(int i=0; i < n; i++){
        if(i % 3 == 2)
            break;
    }
    return Mystery(n - 5)
}
```

$$T(n) = \begin{cases} T(n-5)+3 & \text{if } n > 4 \\ 1 & \text{otherwise} \end{cases}$$

What Do We Do With That

That's nice. So what's the big-Θ bound.

You will find out how to get the big-Θ bound on Wednesday!