CSE 332
Data Structures & Parallelism

Priority Queues 2

Melissa Winstanley

Spring 2024

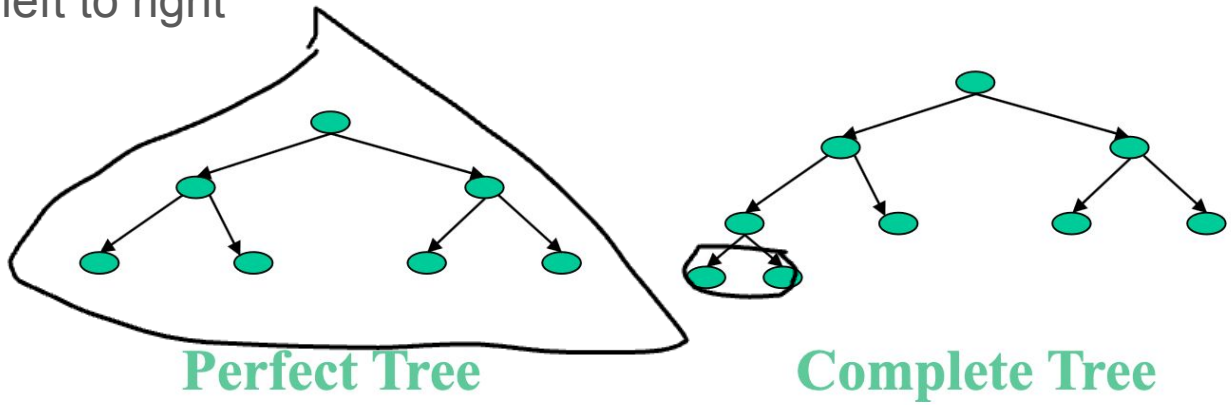
Types of Trees

Binary tree: Every node has ≤2 children

n-ary tree: Every node has ≤n children

Perfect tree: Every row is completely full

Complete tree: All rows except possibly the bottom are completely full, and it is filled from left to right



Properties of a Binary Min-Heap

More commonly known as a binary heap or simply a heap

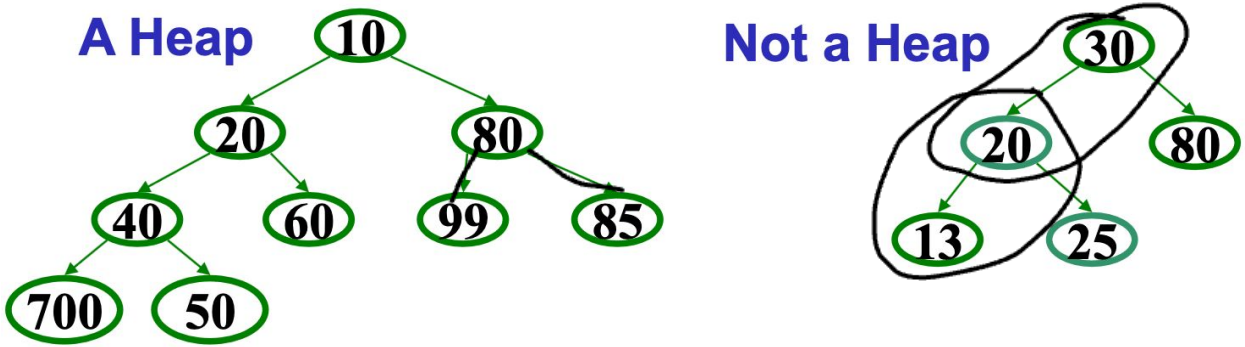
- Structure Property
  - A complete [binary] tree
- Heap Property
  - Every non-root node has a priority value larger than (or possibly equal to) the priority of its parent

How is this different from a binary search tree?

Properties of a Binary Min-Heap

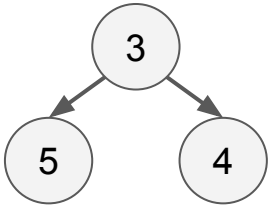
More commonly known as a binary heap or simply a heap

- Structure Property
  - A complete [binary] tree
- Heap Property
  - Every non-root node has a priority value larger than (or possibly equal to) the priority of its parent



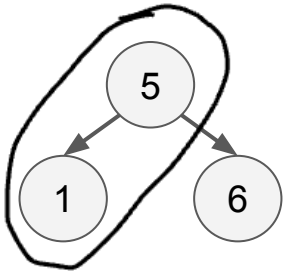
Is it a heap?

A



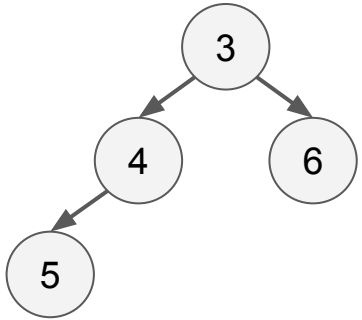
Y

B



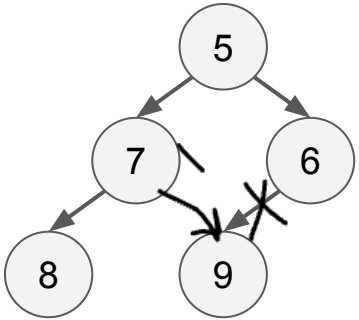
N

C



Y

D



N

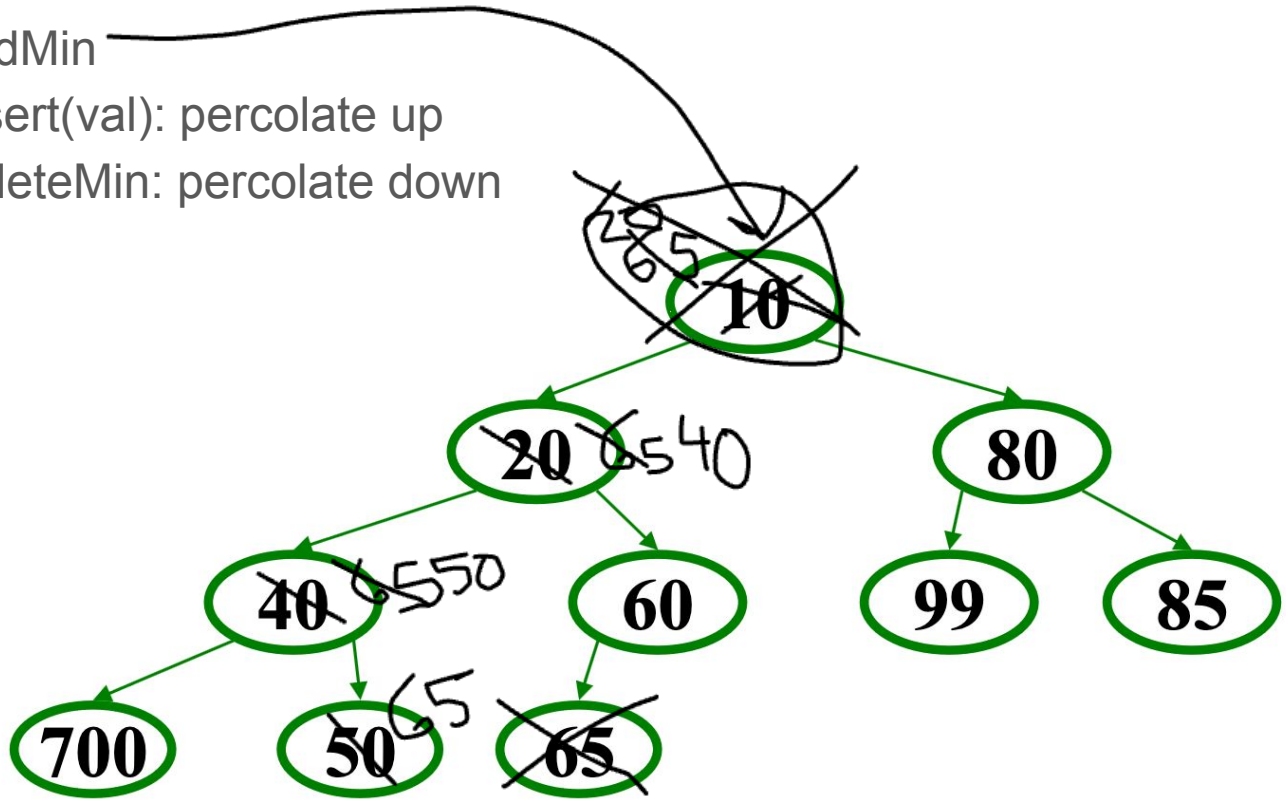
Is this a binary min heap? yes/no

Today - Priority Queues / Heaps

- What is a Priority Queue?
- Introduction to the heap
- Heap operations
- Heap implementation
- Building a heap

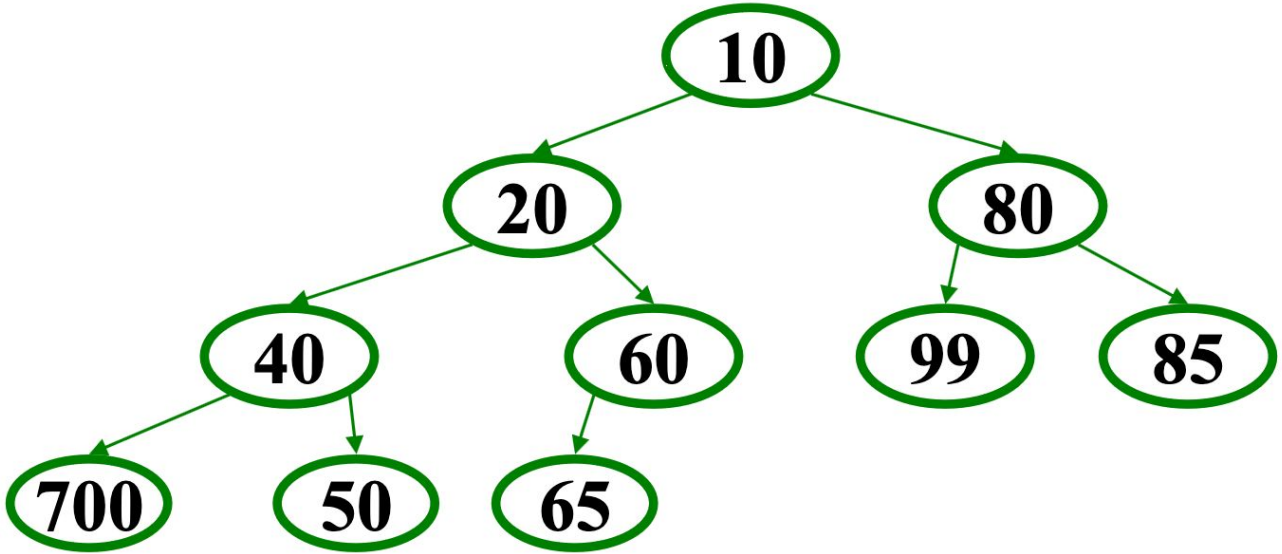
Heap Operations

- findMin
- insert(val): percolate up
- deleteMin: percolate down



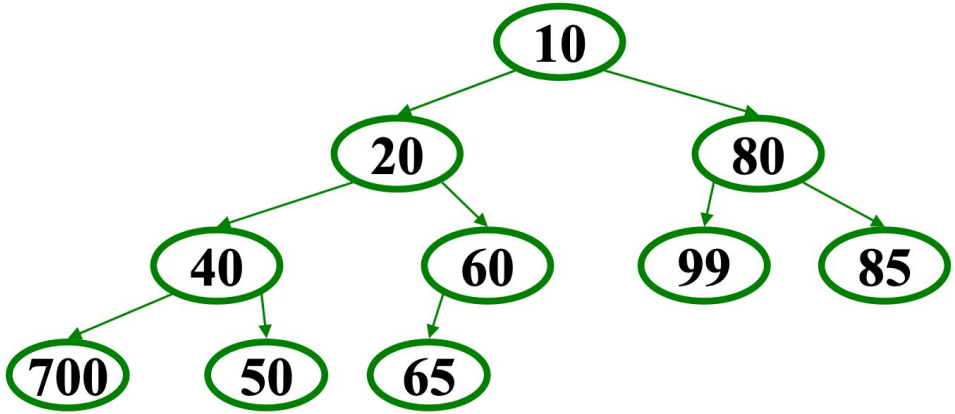
Heap Operations

- findMin
- insert(val): percolate up
- deleteMin: percolate down



Operations: basic idea

- findMin:
  return root.data
- deleteMin:
  1. answer = root.data
  2. Move right-most node in last row to root to restore structure property
  3. “Percolate down” to restore heap order property
- insert:
  1. Put new node in next position on bottom row to restore structure property
  2. “Percolate up” to restore heap order property



Overall strategy:

- Preserve complete tree structure property
- This may break heap order property
- Percolate to restore heap order property

A Clever Trick for Storing the Heap

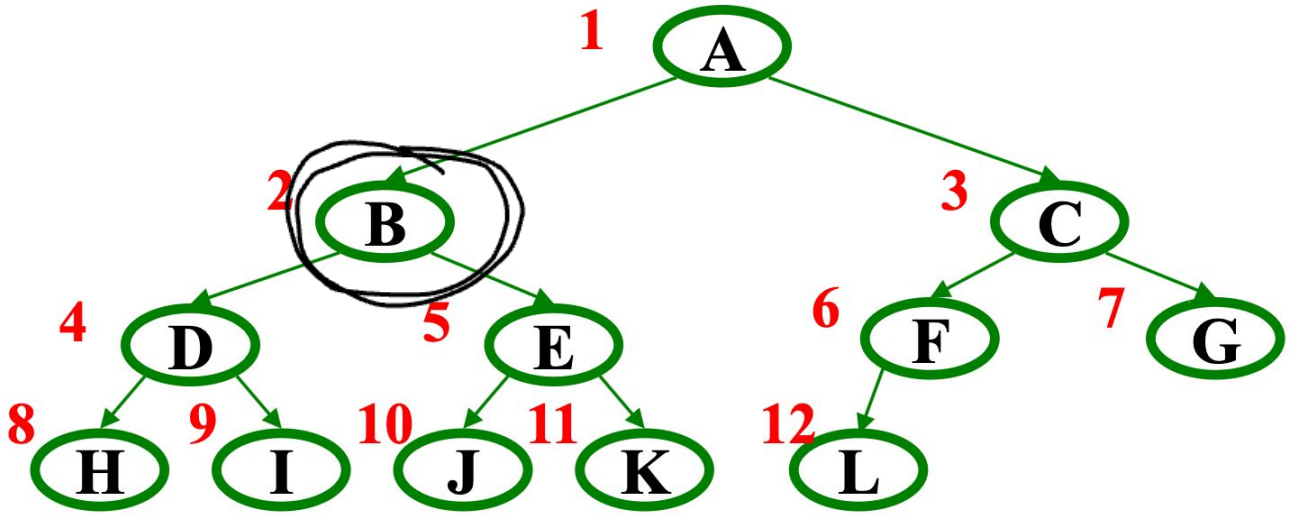
- Clearly, insert and deleteMin are worst-case O(log n)
  - But we promised average-case O(1) insert (how??)
- Insert requires access to the “next to use” position in the tree
  - Walking the tree from root to leaf requires O(log n) steps
  - insert and deleteMin would have to update the “next to use” reference each time: O(log n)
- We should only pay for the functionality we need!!
  - Why have we insisted the tree be complete? ☺
- All complete trees of size n contain the same edges
  - So why are we even representing the edges?

Here comes the really clever bit about implementing heaps!!!

Array Representation of a Binary Heap

From node i:

- left child: 2i
- right child: 2i+1
- parent: $\lfloor \frac{i}{2} \rfloor$



	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 skip index 0 to make the math simpler
- Actually, it can be a good place to store the current size of the heap

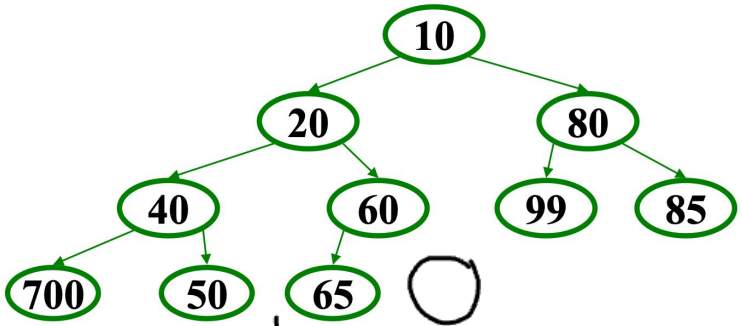
Today - Priority Queues / Heaps

- What is a Priority Queue?
- Introduction to the heap
- Heap operations
- Heap implementation
- Building a heap

Pseudocode: insert

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

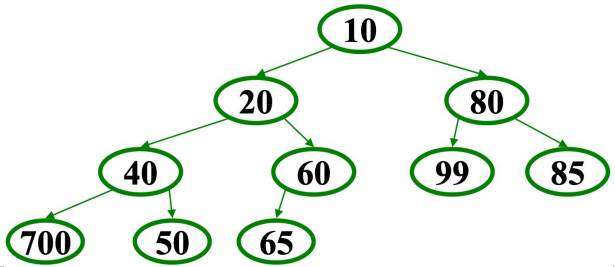
```
int percolateUp(int hole,
                int val) {
  while(hole > 1 &&
        val < arr[hole/2]){
    arr[hole] = arr[hole/2];
    hole = hole / 2;
  }
  return hole;
}
```



	10	20	80	40	60	99	85	700	50	65			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Pseudocode: deleteMin

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

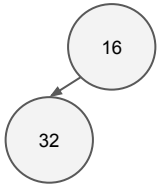



```
int percolateDown(int hole,
                  int val) {
  while(2*hole <= size) {
    left = 2*hole;
    right = left + 1;
    if(arr[left] < arr[right]
       || right > size)
      target = left;
    else
      target = right;
    if(arr[target] < val) {
      arr[hole] = arr[target];
      hole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

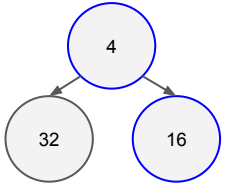
	10	20	80	40	60	99	85	700	50	65			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Example

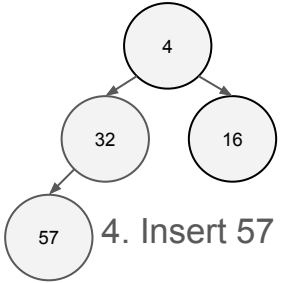
1. Insert 16



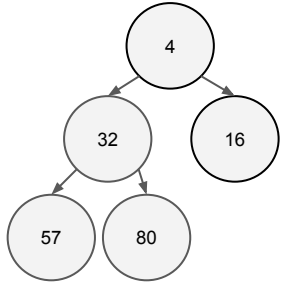
2. Insert 32



3. Insert 4, percDown



4. Insert 57



5. Insert 80

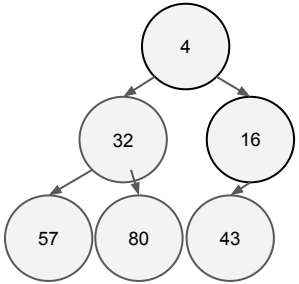
Insert: 16, 32, 4, 57, 80, 43, 2

Then deleteMin

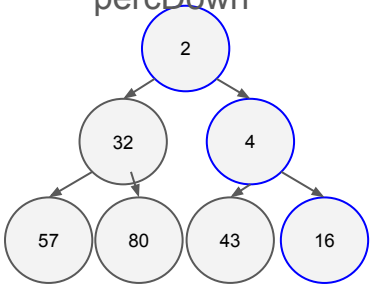
Do as a tree

What is the result?

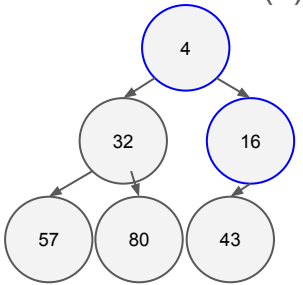
6. Insert 43



7. Insert 2, percDown



8. deleteMin (2)



	4	32	16	57	80	43	
0	1	2	3	4	5	6	7

Other Operations

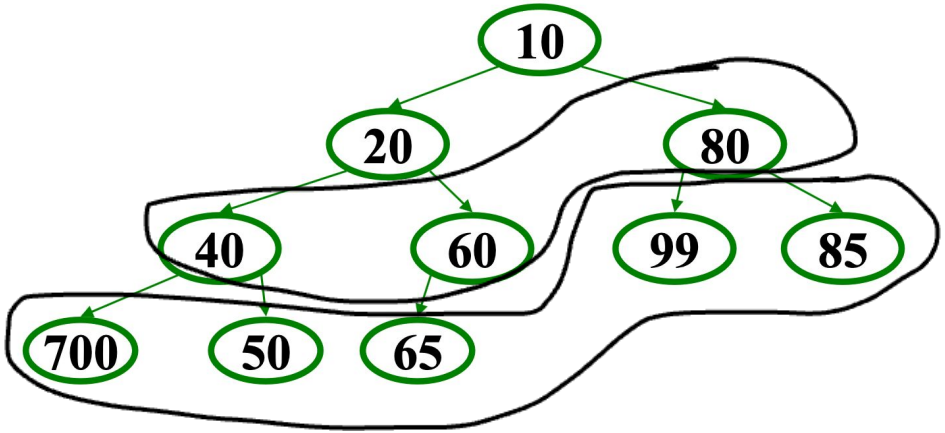
- **`decreaseKey`**: given pointer to object in priority queue (e.g., its array index), lower its priority value by p
  - Change priority and percolate up
- **`increaseKey`**: given pointer to object in priority queue (e.g., its array index), raise its priority value by p
  - Change priority and percolate down
- **`remove`**: given pointer to object in priority queue (e.g., its array index), remove it from the queue
  - decreaseKey with p = ∞, then deleteMin
- Running time for all these operations?

So why O(1) average-case insert?

- Yes, **`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

Intuition for O(1) runtime average case

- 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
  - 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, 2nd 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



Today - Priority Queues / Heaps

- What is a Priority Queue?
- Introduction to the heap
- Heap operations
- Heap implementation
- Building a heap

Building a Heap

Suppose you have n items you want to put in a new priority queue

- A sequence of n **`insert`** operations works
- Runtime? $O(n \log n)$

Can we do better?

- If we only have access to **`insert`** and **`deleteMin`** operations, then NO.
- There is a faster way - O(n), but that requires the ADT to have a specialized **`buildHeap`** operation

Important issue in ADT design: how many specialized operations?

- Tradeoff: Convenience, Efficiency, Simplicity

Floyd's `buildHeap` Method

Recall our general strategy for working with the heap:

- Preserve structure property
- Break and restore heap ordering property

Floyd's `buildHeap`:

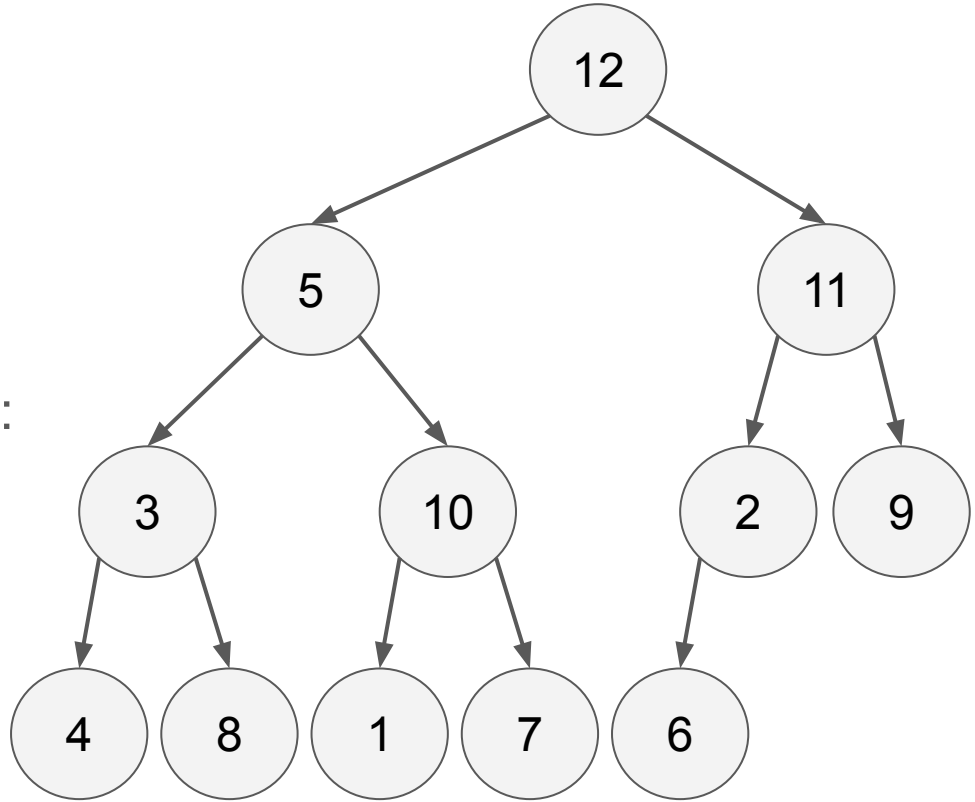
1. Create a complete tree by putting the n items in array indices 1, . . . n
2. Treat the array as a heap and fix the heap-order property
    - Exactly how we do this is where we gain efficiency

Thinking about **`buildHeap`**

- Say we start with this array:

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

- To “fix” the ordering can we use:
  - percolateUp?
  - percolateDown?



Floyd's `buildHeap` Method

Bottom-up:

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

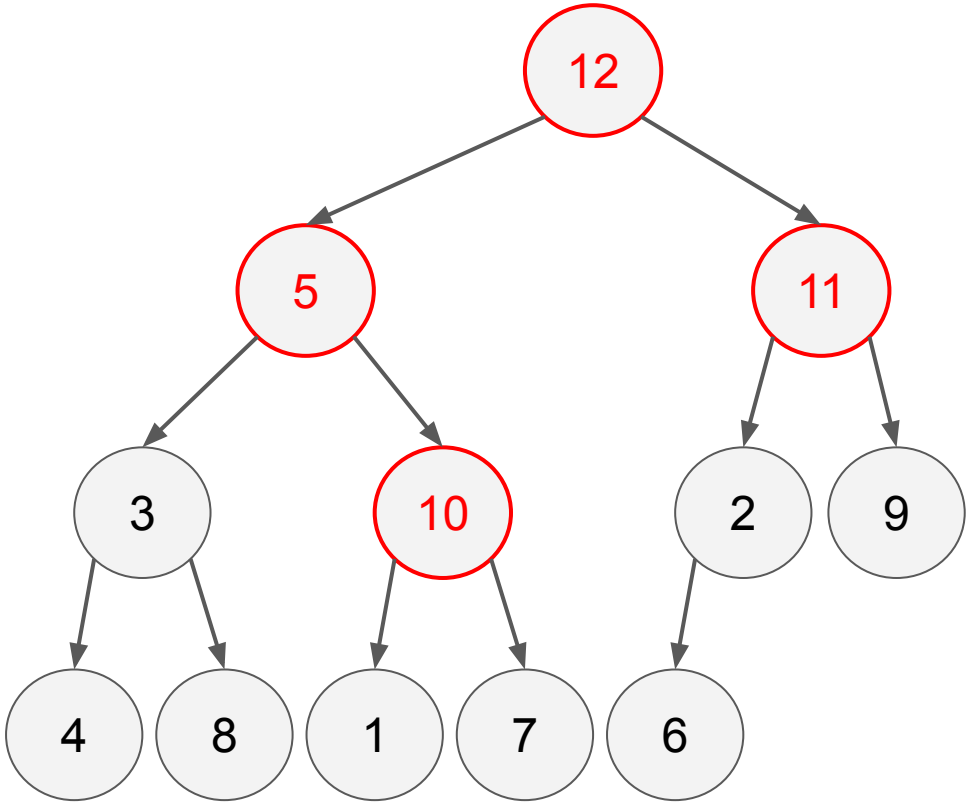
```
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val = arr[i];
    hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```

Thinking about `buildHeap`

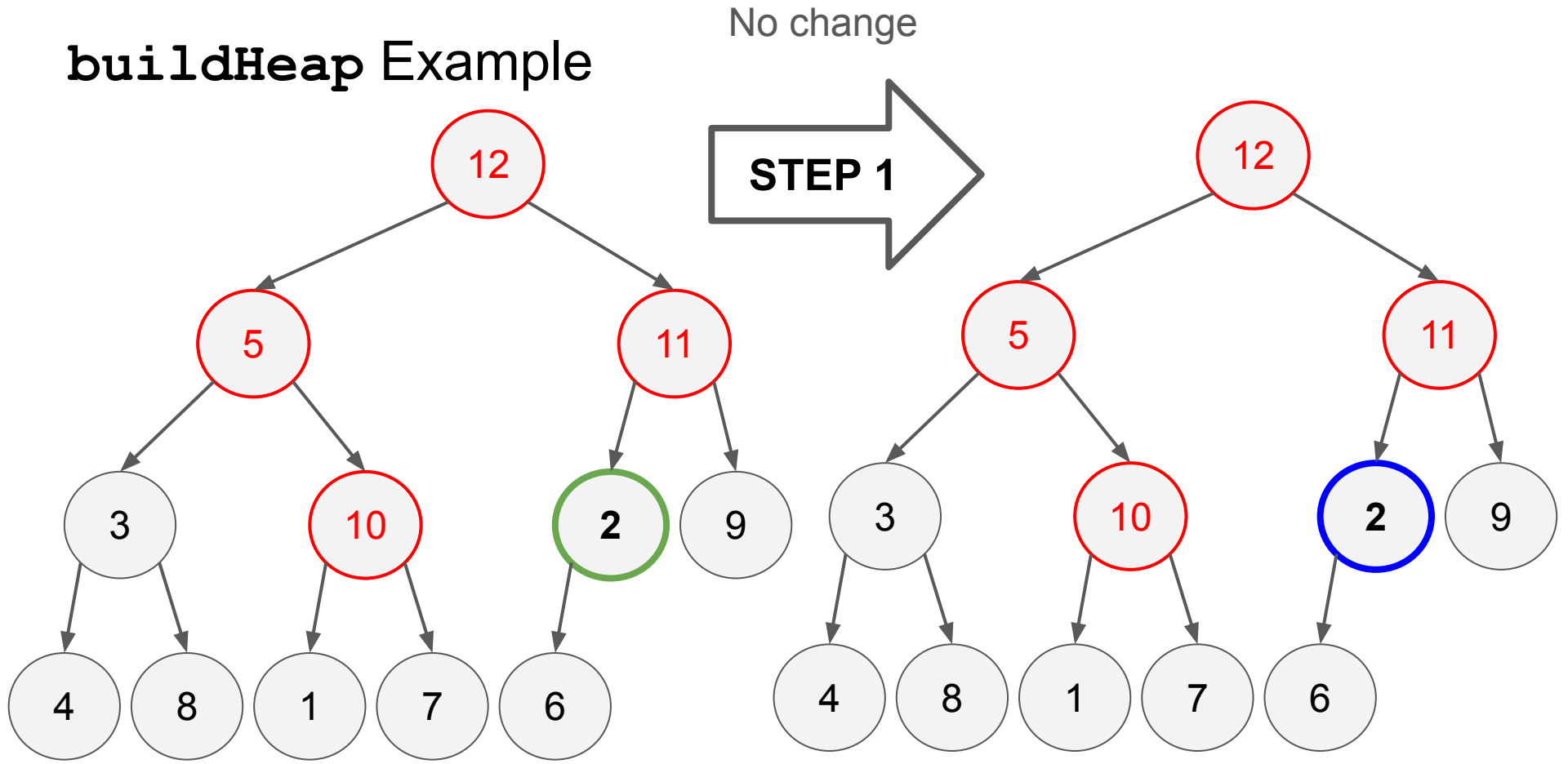
- Say we start with this array:

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

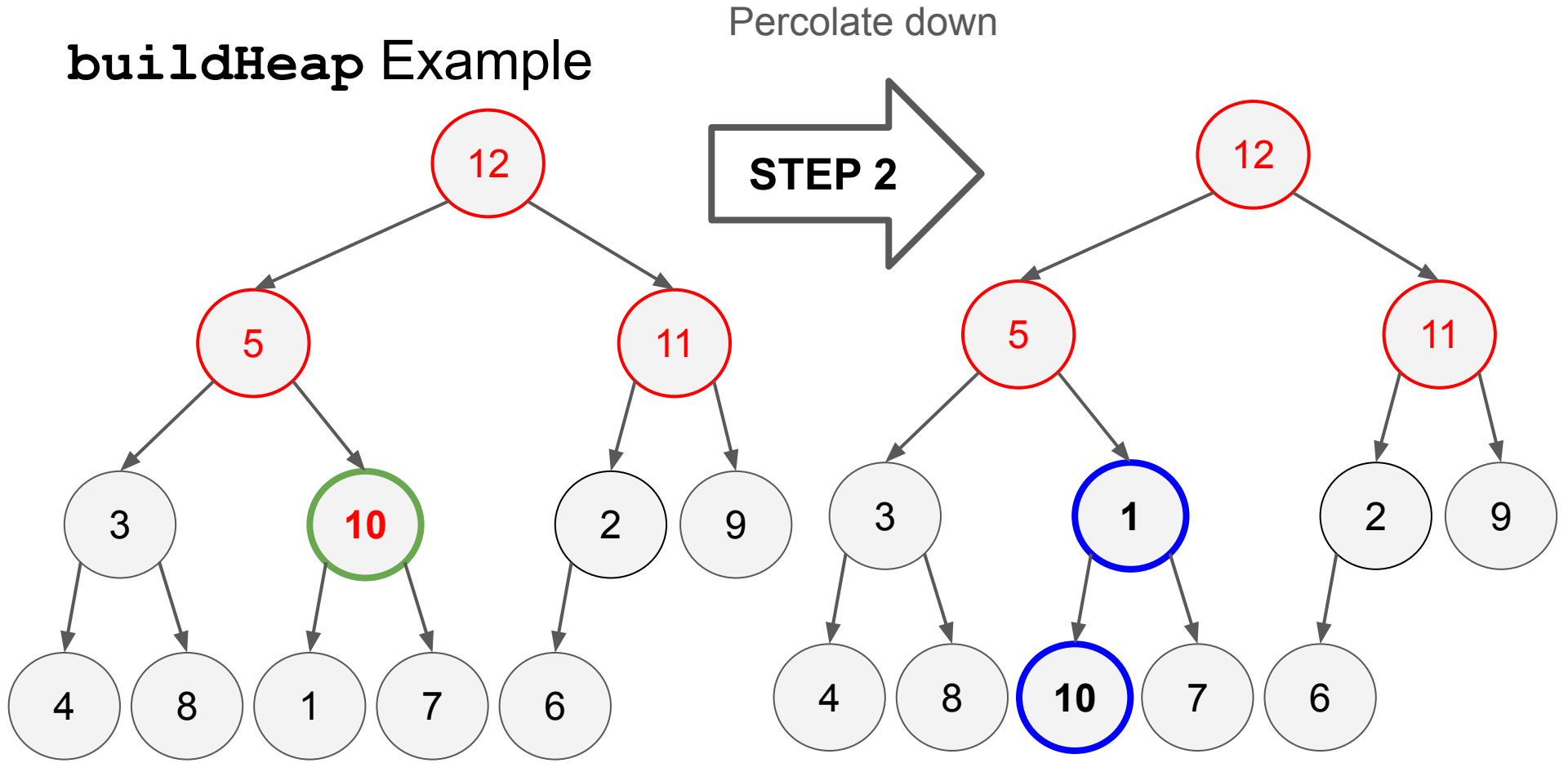
- In tree form for readability
  - Red for node out of place
- Notice no leaves are red
- Check/fix each non-leaf, bottom-up (6 steps here)



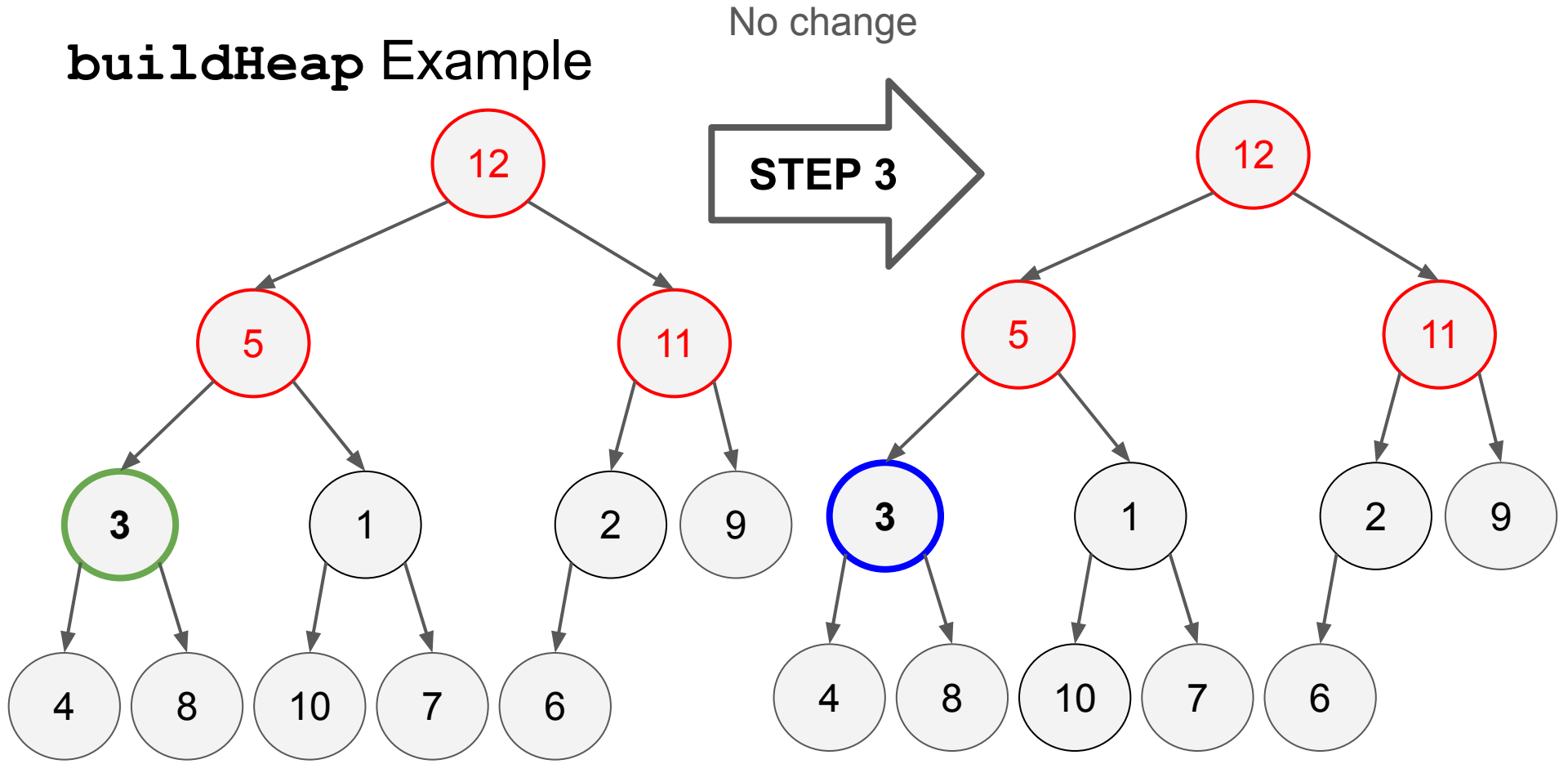
`buildHeap` Example



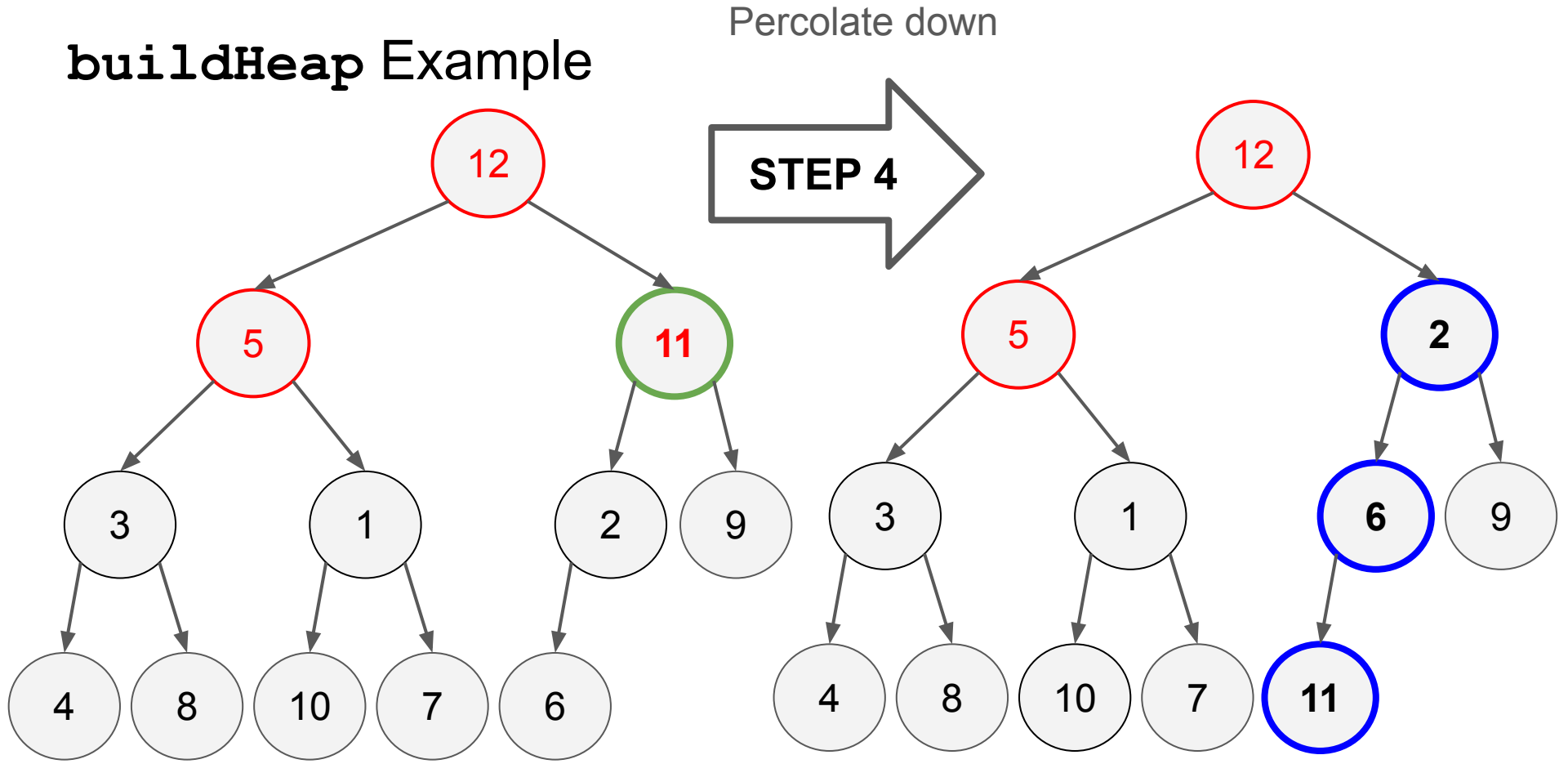
`buildHeap` Example



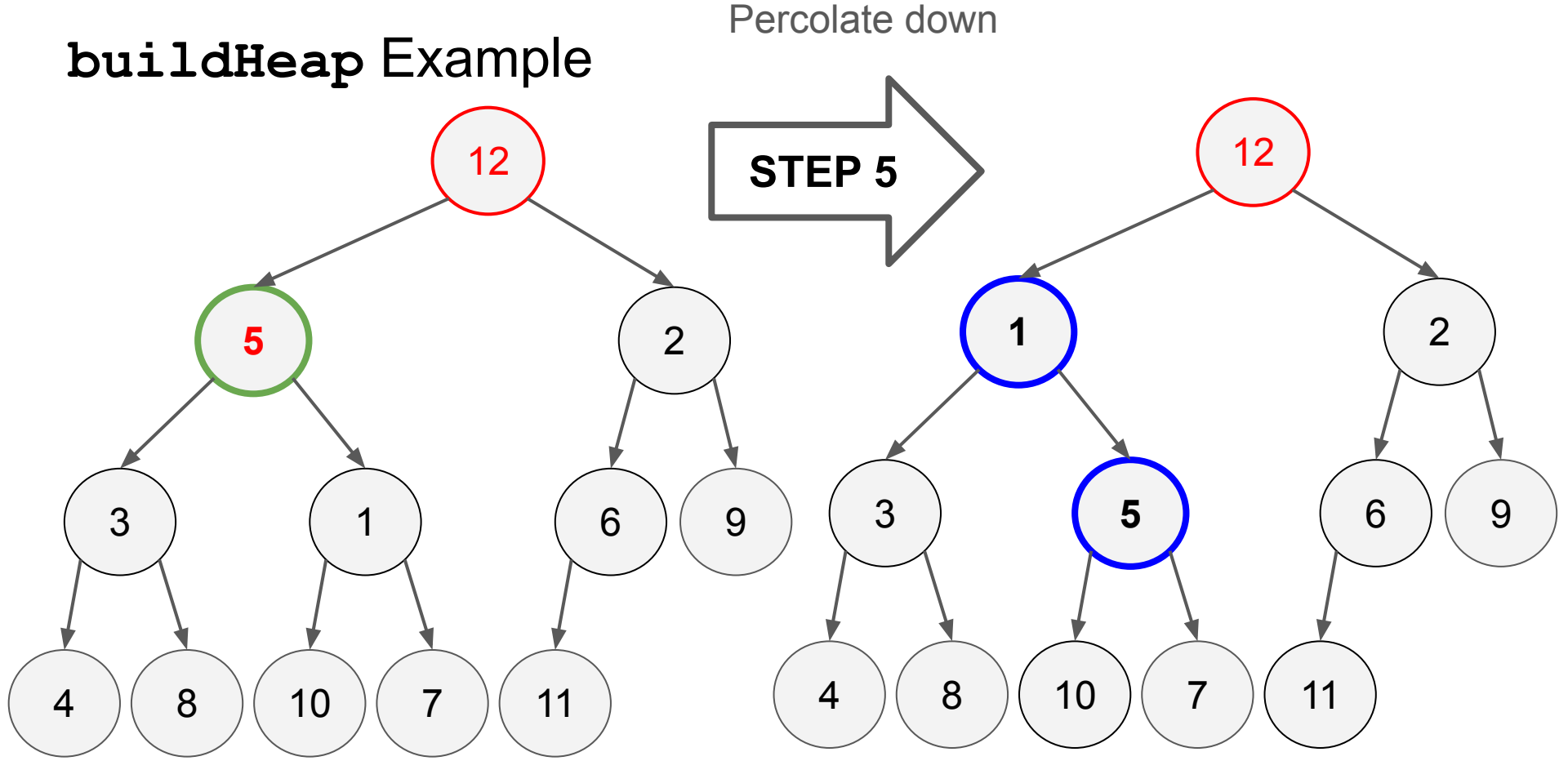
`buildHeap` Example



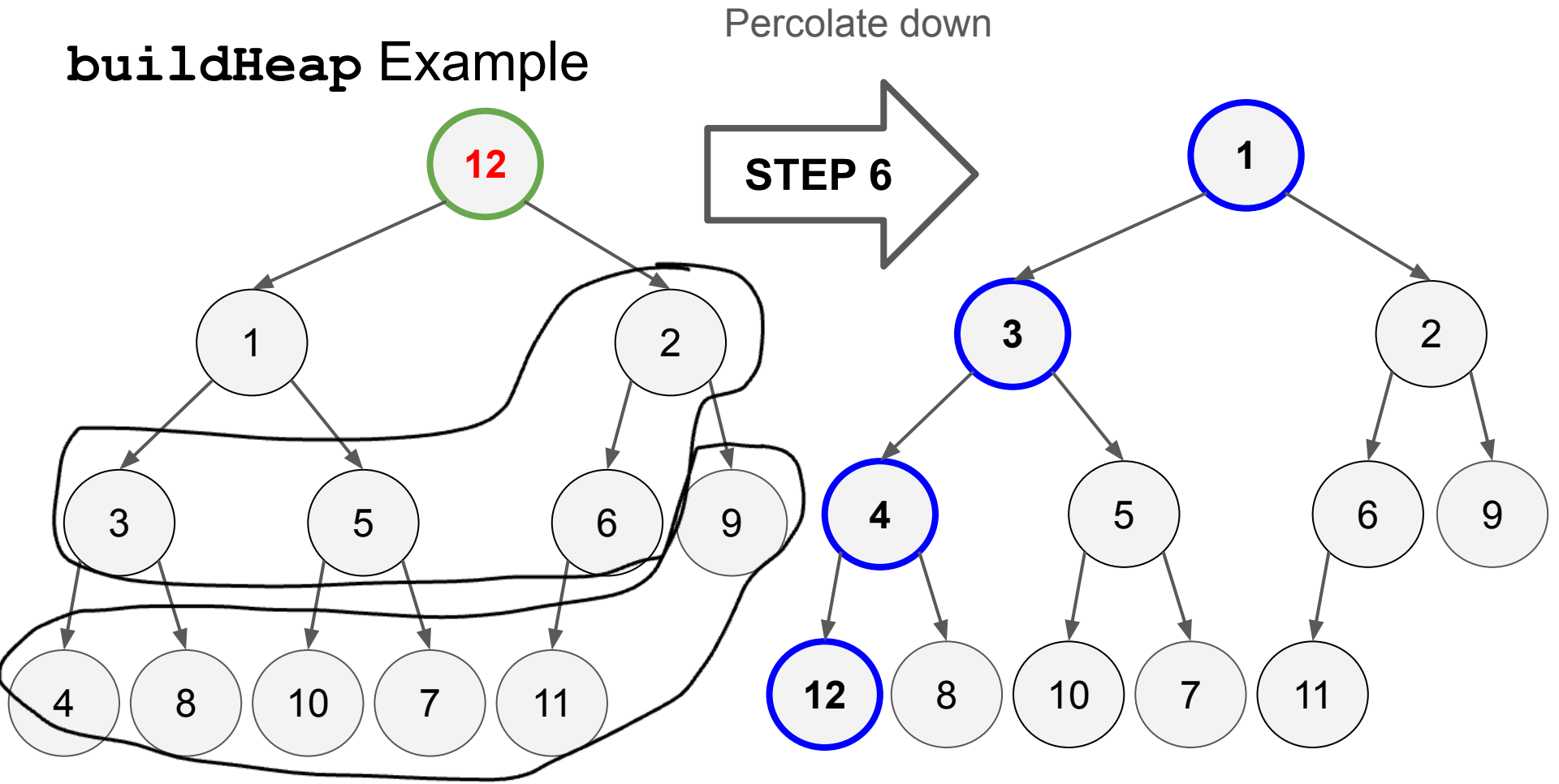
`buildHeap` Example



`buildHeap` Example



`buildHeap` Example



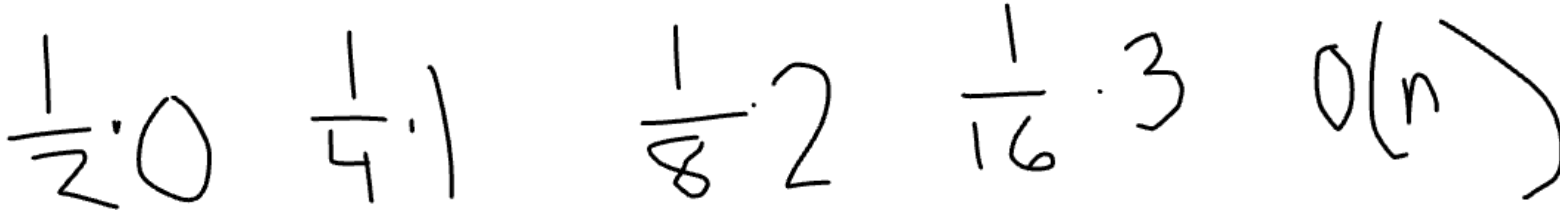
`buildHeap` Efficiency

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…

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



`buildHeap` Efficiency

- No node can percolate down more than the height of its subtree
  - When i is a leaf: 0
  - When i is second-from-last level: 1
  - When i is third-from-last level: 2
- Overall Running time:

k = height of tree = log n but it's actually easier math if we say that it's infinity

$$0\left(\frac{n}{2}\right) + 1\left(\frac{n}{4}\right) + 2\left(\frac{n}{8}\right) .... + k\left(\frac{n}{2^{k+1}}\right)$$

$$\sum_{k=0}^{\infty} \frac{kn}{2^{k+1}} = n\sum_{k=0}^{\infty} \frac{k}{2^{k+1}} = n \times 1 = n$$