CSE 332: Data Structures & Parallelism

Lecture 4: Binary Heaps, Continued

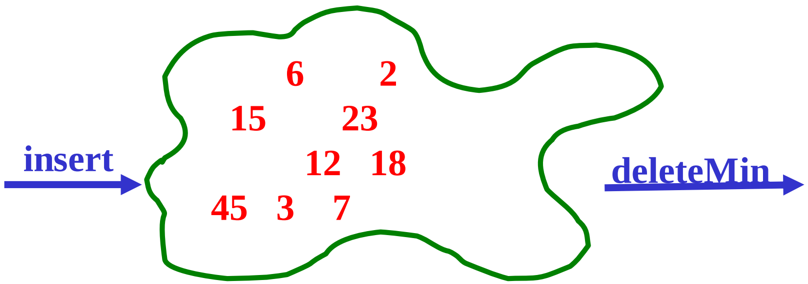
Ruth Anderson

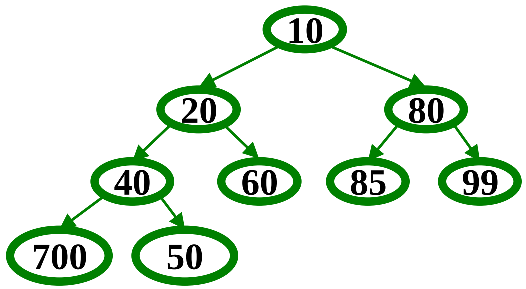
Winter 2018

Today

- Binary Min Heap implementation
  - Insert
  - Deletemin
  - Buildheap

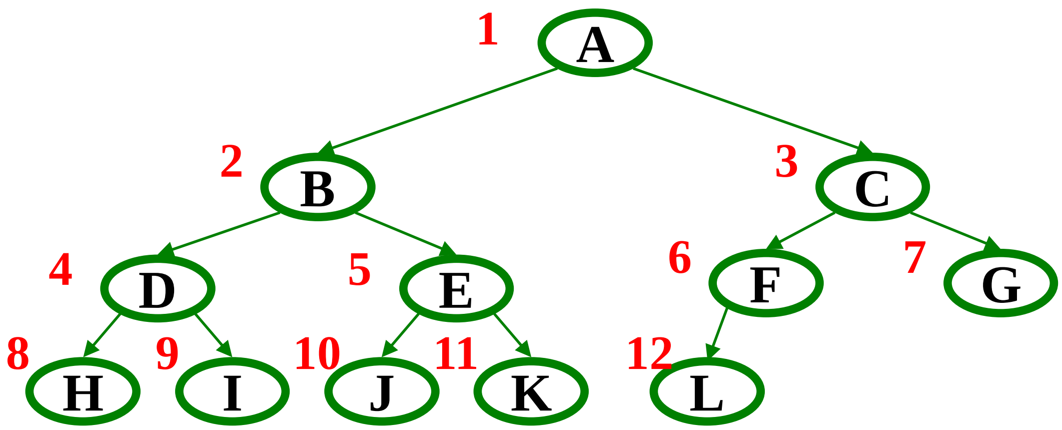
Review





- Priority Queue ADT: **`insert`** comparable object, **`deleteMin`**
- Binary heap data structure: Complete binary tree where each node has priority value greater than its parent
- *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

Array Representation of Binary Trees



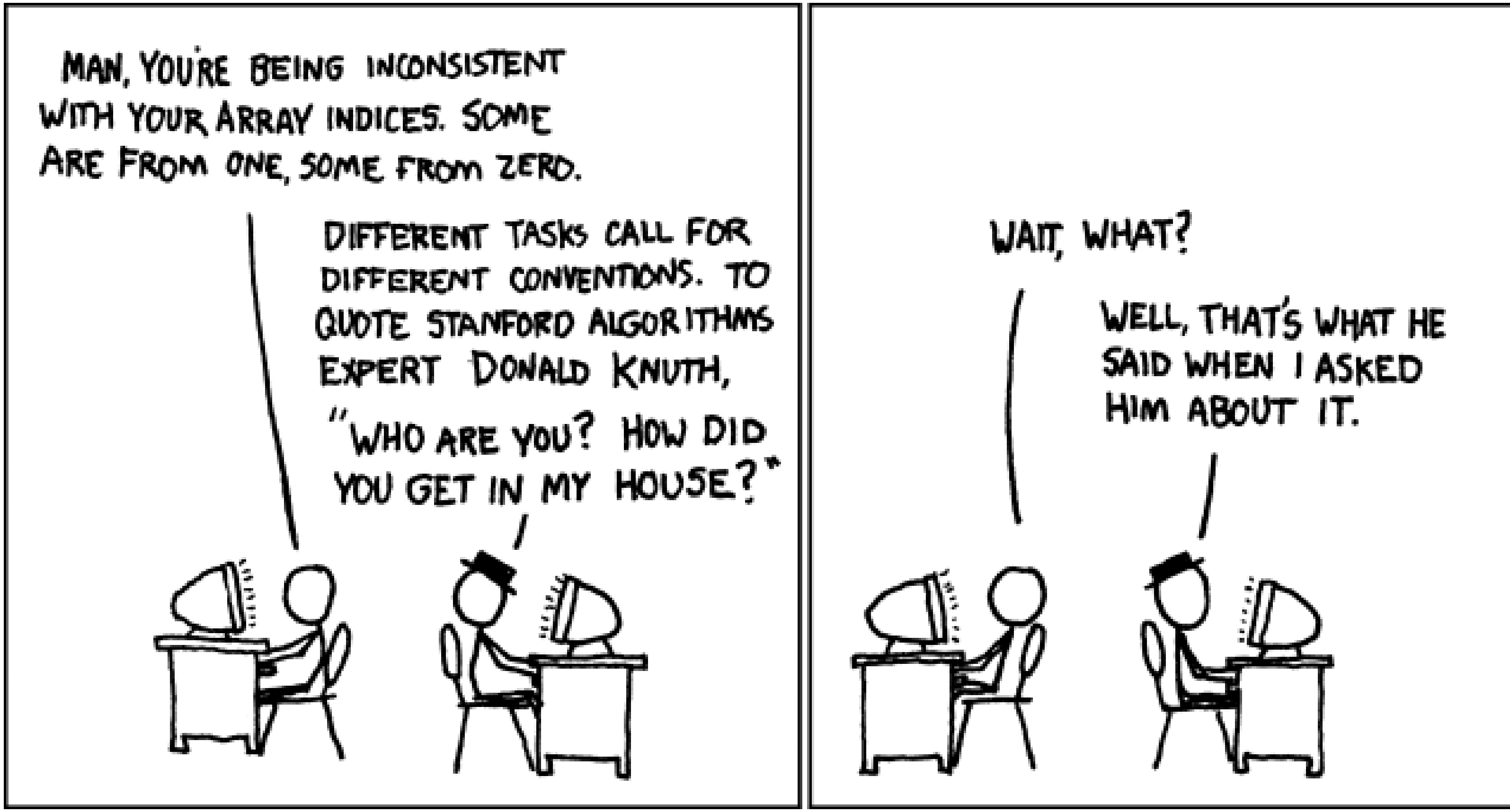
From node `i`:

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



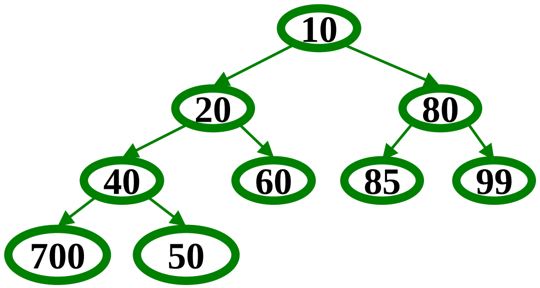
http://xkcd.com/163

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;
}
```



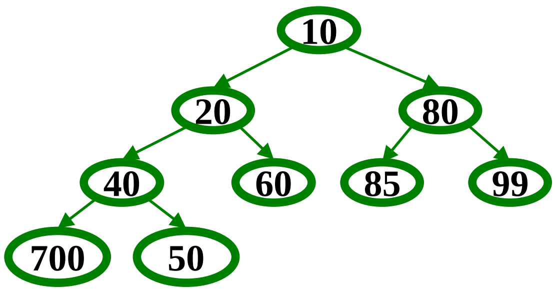
	10	20	80	40	60	85	99	700	50				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

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;
}
```

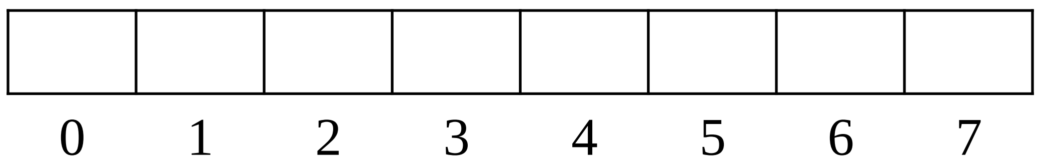
```
int percolateDown(int hole,
                  int val) {
 while(2*hole <= size) {
  left  = 2*hole;
  right = left + 1;
  if(arr[left] < arr[right]
     || right > size)
    target = left;
  else
    target = right;
  if(arr[target] < val) {
    arr[hole] = arr[target];
    hole = target;
  } else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

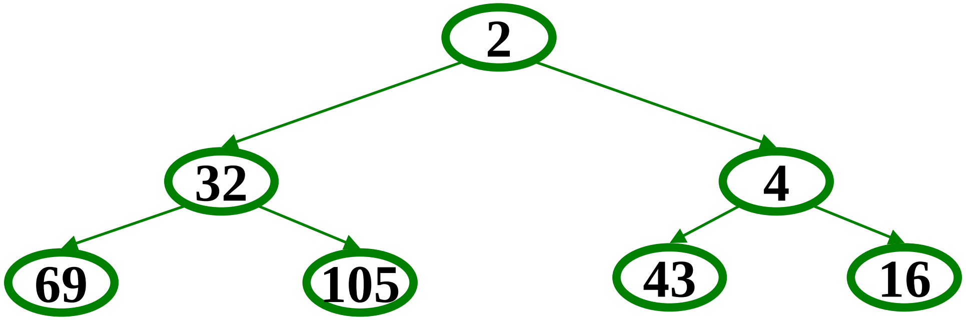
1. insert: 16, 32, 4, 69, 105, 43, 2
2. deleteMin



Example: After insertion

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

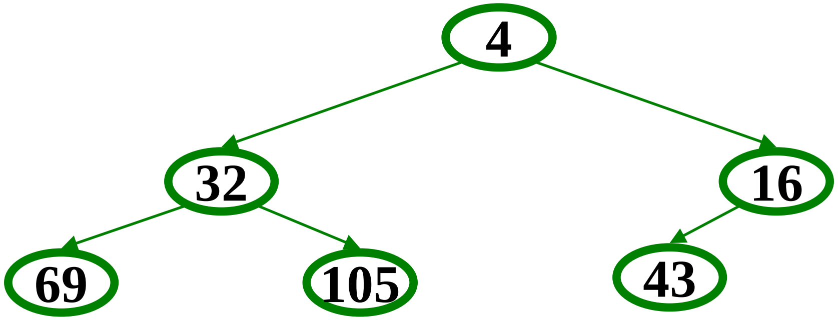
	2	**32**	**4**	**69**	**105**	**43**	**16**
0	1	2	3	4	5	6	7



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

- **`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 $p = \infty$, then **`deleteMin`**

Running time for all these operations?

Evaluating the Array Implementation…

Advantages:

Minimal amount of wasted space:

- Only index 0 and any unused space on right in the array
- No "holes" due to complete tree property
- No wasted space representing tree edges

Fast lookups:

- Benefit of array lookup speed
- Multiplying and dividing by 2 is extremely fast (can be done through bit shifting (see CSE 351)
- Last used position is easily found by using the PQueue's size for the index

Disdvantages:

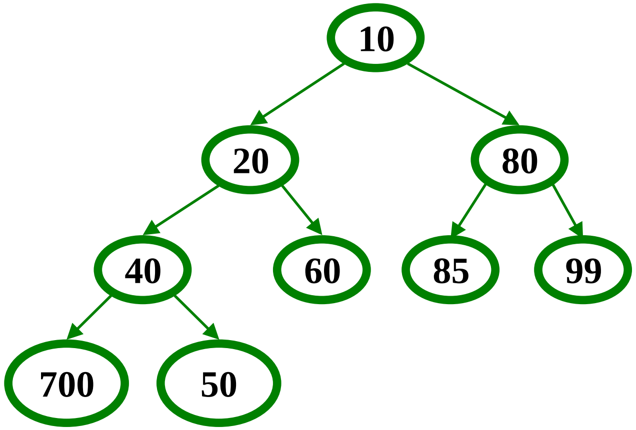
- What if the array gets too full (or wastes space by being too empty)? Array will have to be resized.

Advantages outweigh Disadvantages: This is how it is done!

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

Aside: 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

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?

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

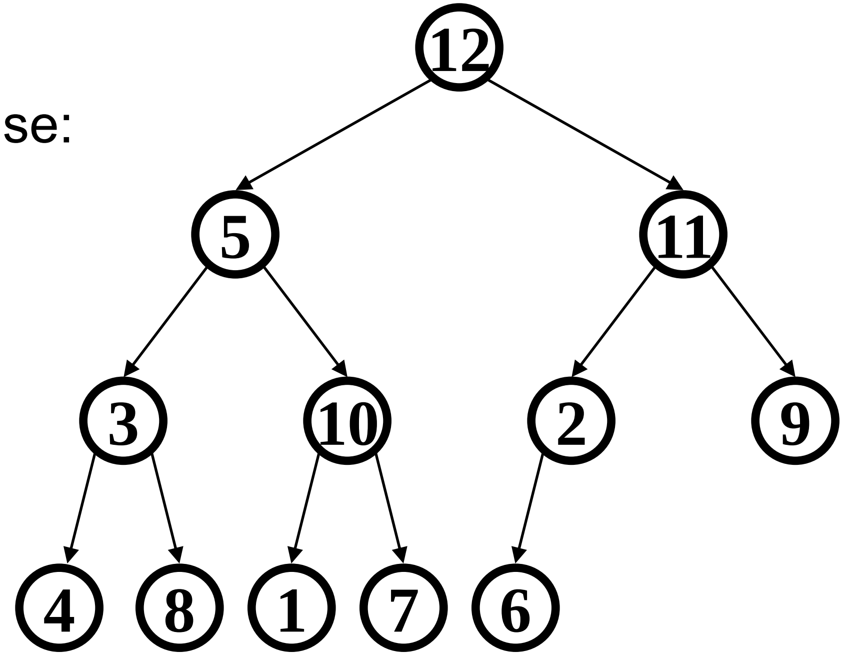
Floyd's ***`buildHeap`***:

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?
  - percoalteDown?



Floyd's ***`buildHeap`*** *Method*

Bottom-up:

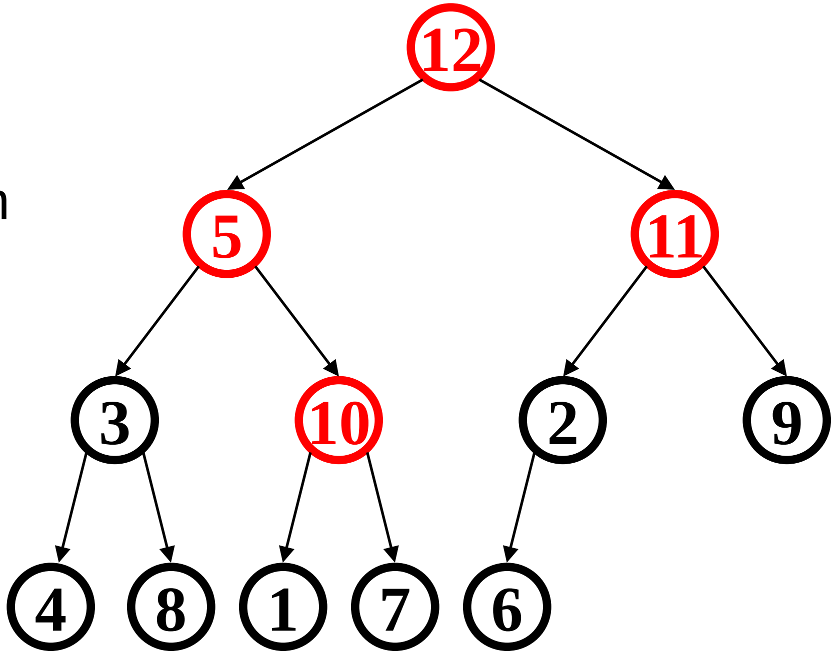
- Leaves are already in heap order
- Work up toward the root one level at a time

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

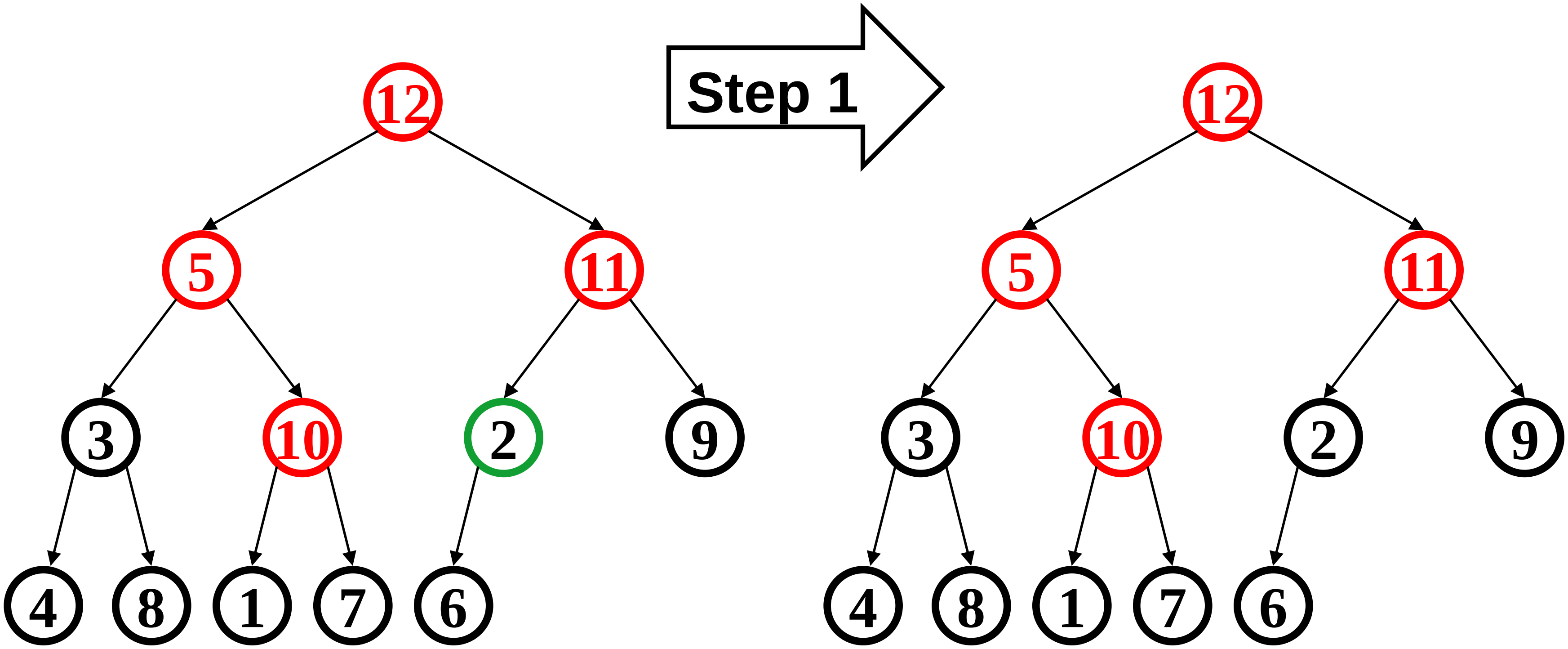
buildHeap *Example*

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

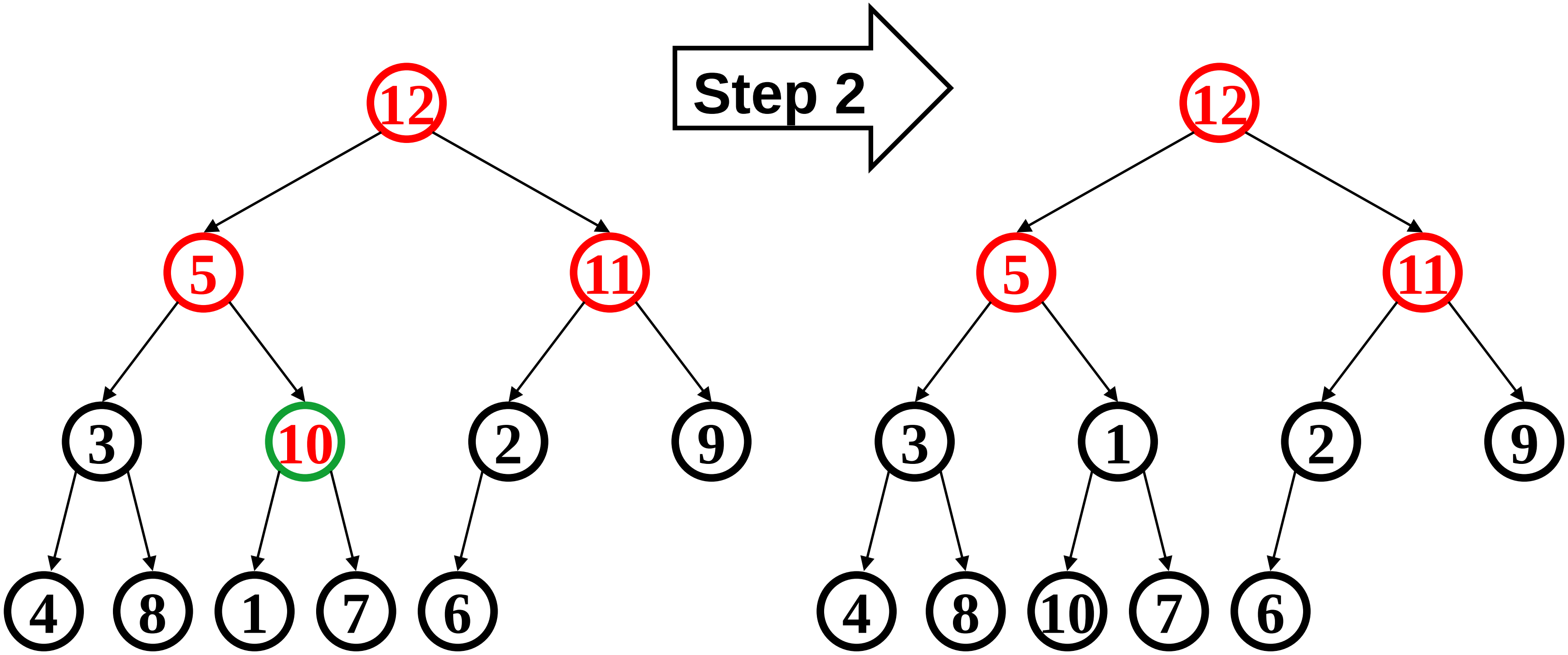


buildHeap *Example*



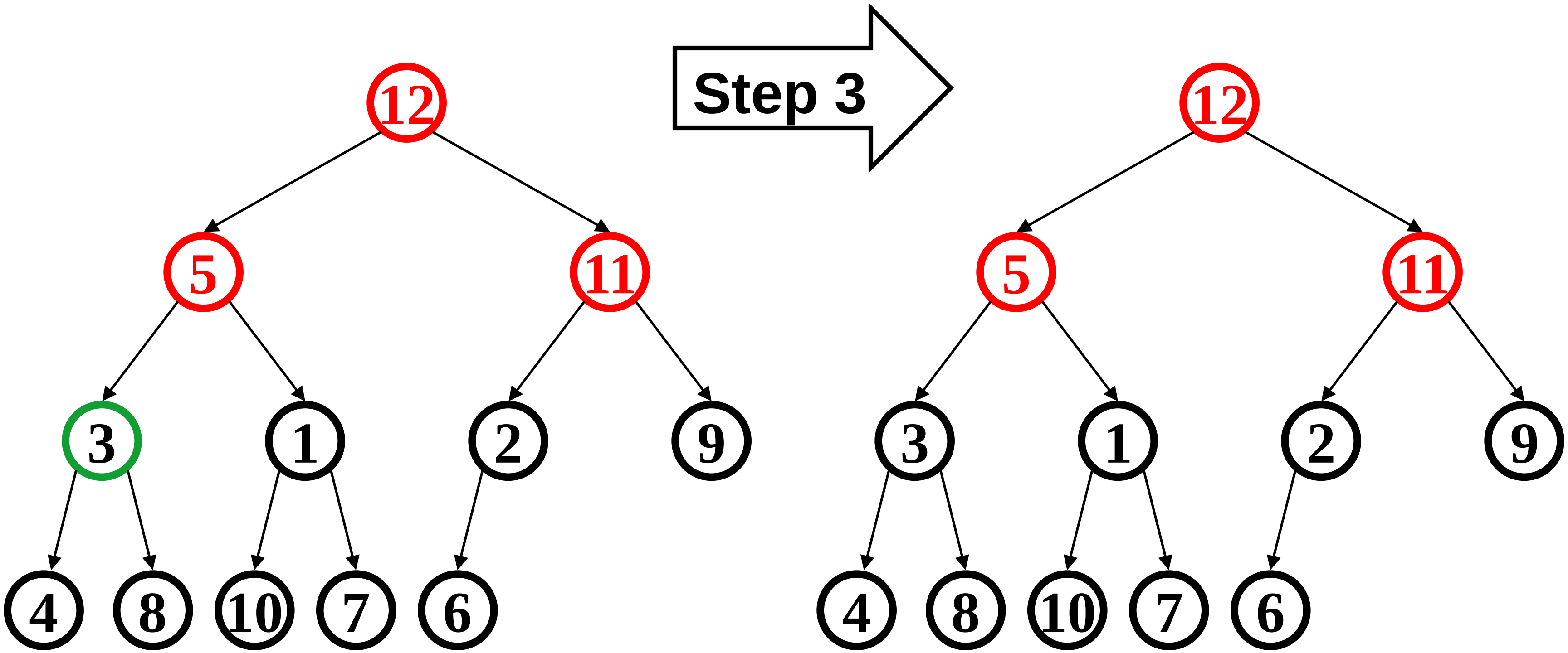
- Happens to already be less than child

`buildHeap` *Example*



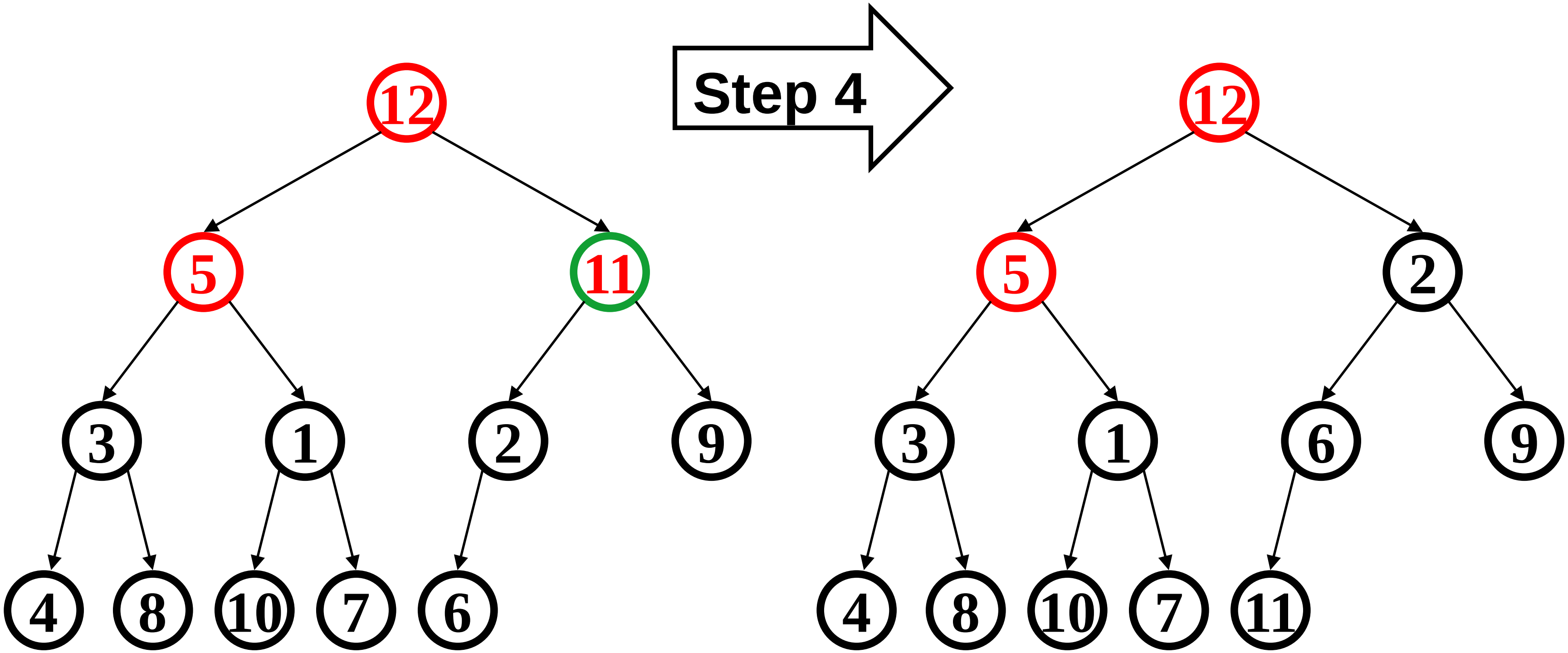
- Percolate down (notice that moves 1 up)

`buildHeap` *Example*



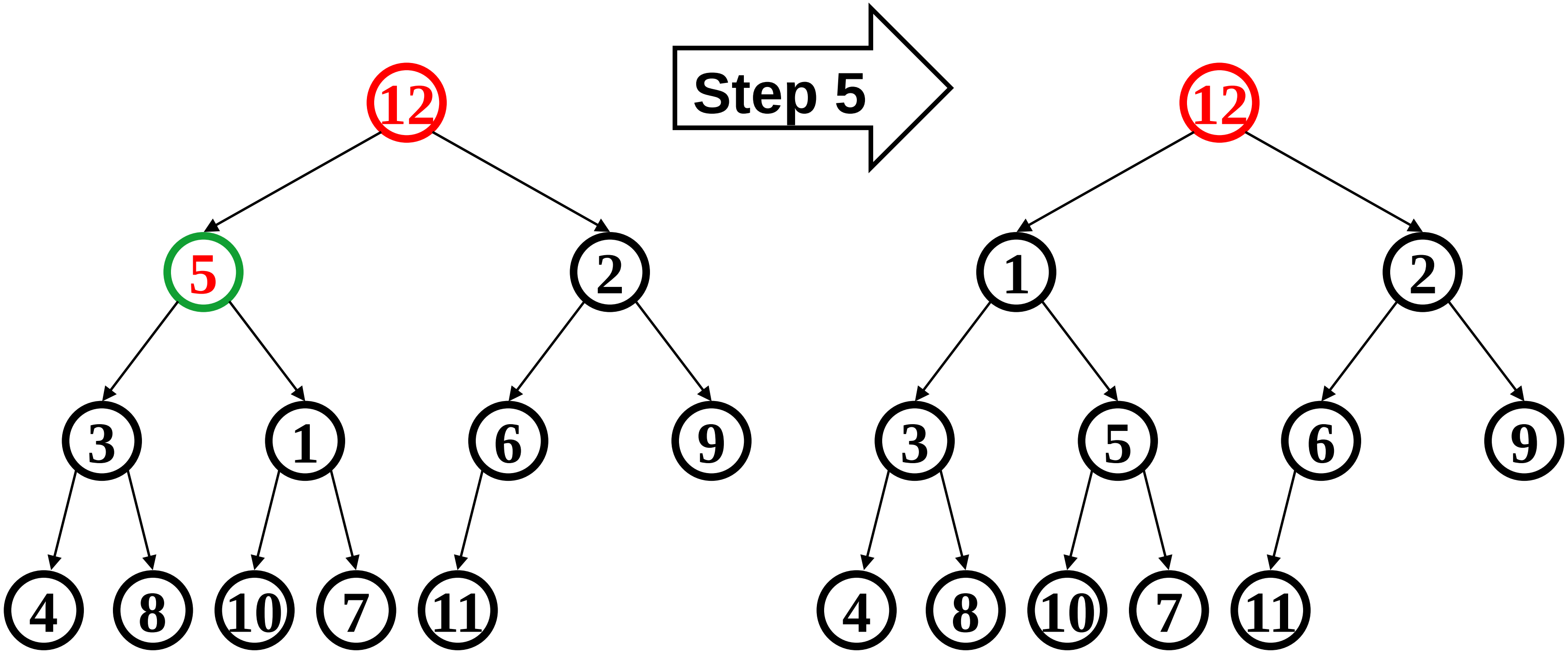
- Another nothing-to-do step

`buildHeap` *Example*

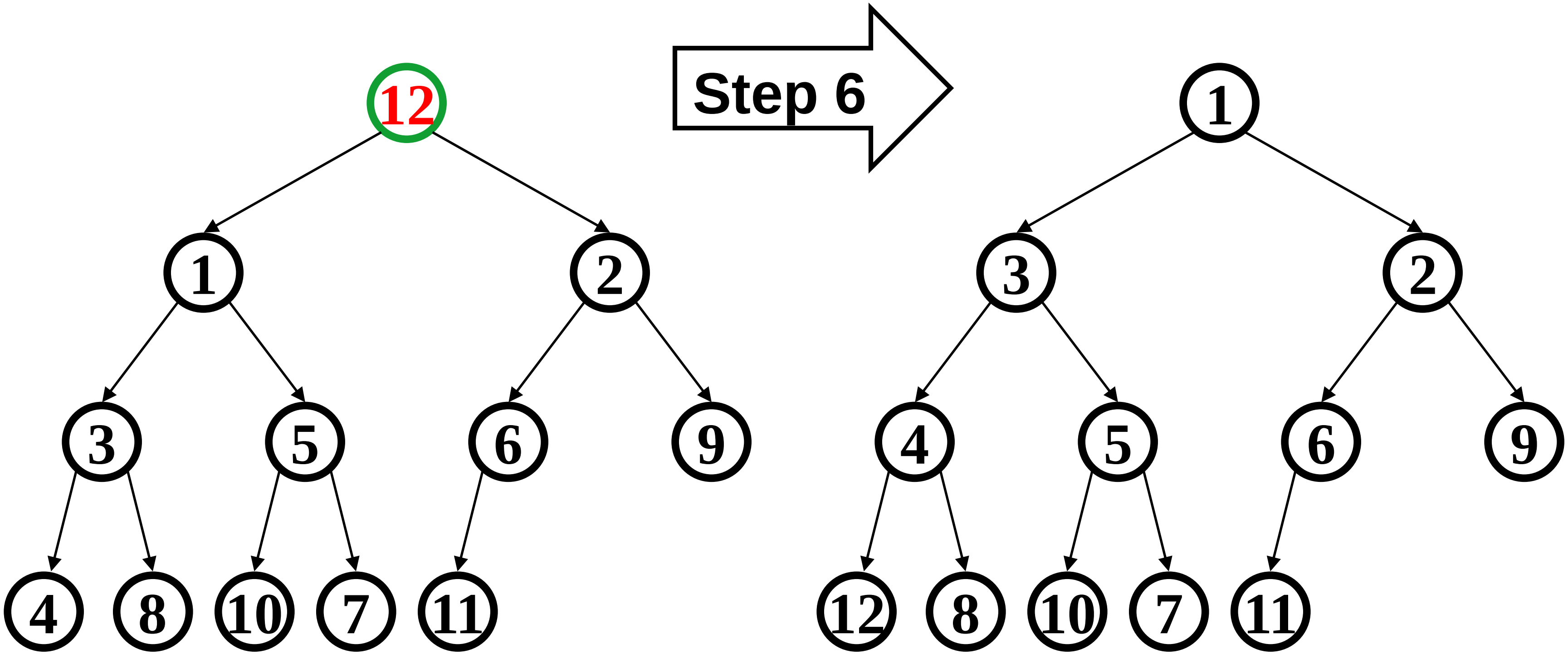


- Percolate down as necessary (steps 4a and 4b)

buildHeap *Example*



buildHeap *Example*



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

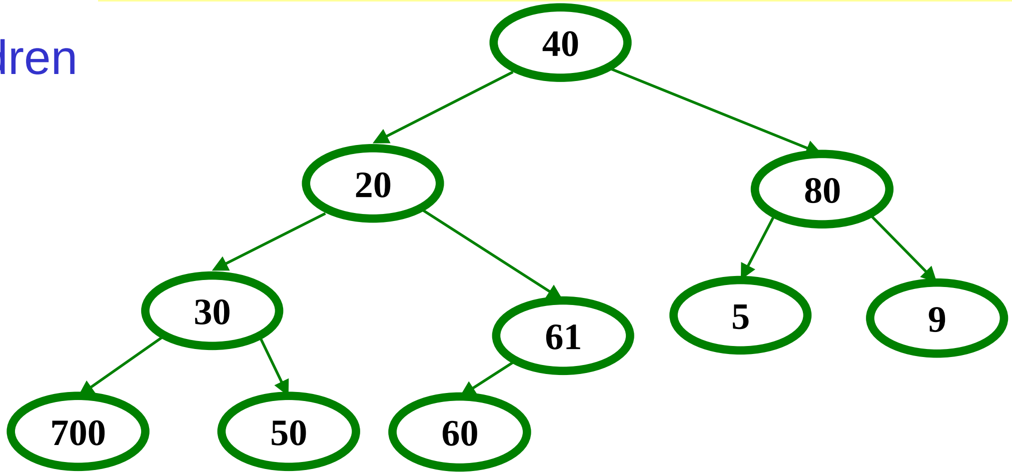
Loop Invariant:

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;
   }
}
```



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

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

Lessons from **`buildHeap`**

- Without **`buildHeap`**, our ADT already let clients implement their own in $\theta(n \log n)$ 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 $O(n \log n)$
    - 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 (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?