Lecture 4: Priority Queue ADT

CSE 332: Data Structures & Parallelism

Yafqa Khan

Summer 2025

Announcements

- EX01 Analysis
  - Due **Today**
- EX02 Heaps
  - Releases Wednesday
  - Due next Friday

Today

- Priority Queue ADT
- Tree Stuff
- Binary Min-Heap Data Structure
  - Basics, Properties, Operations
  - Array Representation
- Floyd's BuildHeap

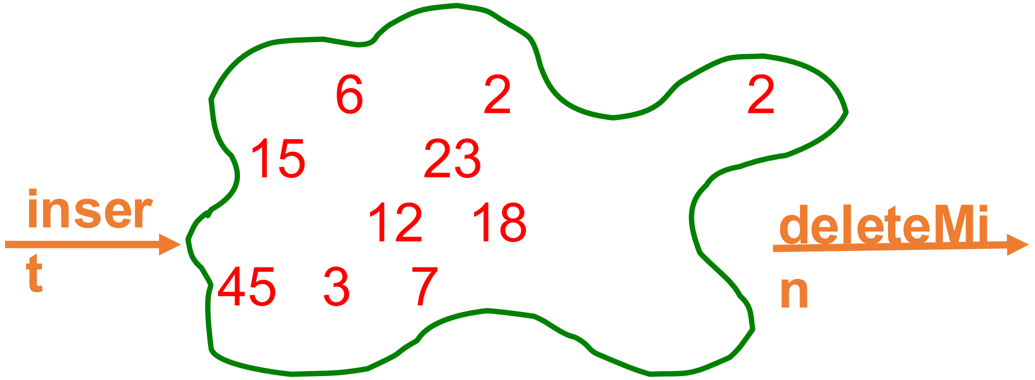
Priority Queue: Scenario

- What is the difference between waiting for service at
  - a pharmacy
  - VS an ER?
- Pharmacies usually follow the rule
  - First Come, First Served
- Emergency Rooms assign **priorities** based on everyone's needs

Priority Queue ADT

- Chapter 6 of Weiss
- The ***PriorityQueue*** ADT supports operations:
  - `insert` (`enqueue` equivalent):
    - adds an item at the end
  - `deleteMin` (`dequeue` equivalent):
    - finds, returns, and removes the minimum element in the priority queue
  - `findMin`, `isEmpty`, etc.
  - BUT! Also only holds **comparable data**
- An example of a PriorityQueue data structure is a heap, with its associated algorithms for the operations
- One implementation is in the library **`java.util.PriorityQueue`**

Priority Queue: ADT



- Holds **comparable data**
  - Each element has a "priority"
  - Lesser priority value = Higher priority
    = Closer to the "front of the priority queue"

  (For a <u>min</u> Priority Queue)
- Main operations: `insert` and `deleteMin`
  - `insert` (`enqueue` equivalent):
    - adds an item at the end
  - `deleteMin` (`dequeue` equivalent):
    - finds, returns, and removes the minimum element in the priority queue
      - break ties arbitrarily
    - minimum element/priority value = highest priority

Priority Queue: Simplifying in Lecture

- We will use `int`s as the data AND the priority
- e.g., `insert(5)` = insert the data `5` with priority value `5`
  - Remember: lower priority value = closer to the "front of priority queue"

Priority Queue: Preliminary Data Structures

	`insert`	`deleteMin`
Unsorted Array		
Unsorted Linked-List		
Sorted Circular Array		
Sorted Linked-List		
Binary Search Tree (BST)		

Note: Worst case, assume arrays have enough space

Priority Queue: Heap Data Structure

	insert	deleteMin
(Binary Min) Heap		

Extra Bonus: Good constant factors, If items arrive in random order, then the "average"-case of insert is $\Theta(1)$

Key idea: Only pay for functionality needed

- We need something better than scanning unsorted items
- But we do not need to maintain a full sorted list

- Does the $\log n$ remind you of anything? 🌲🌲

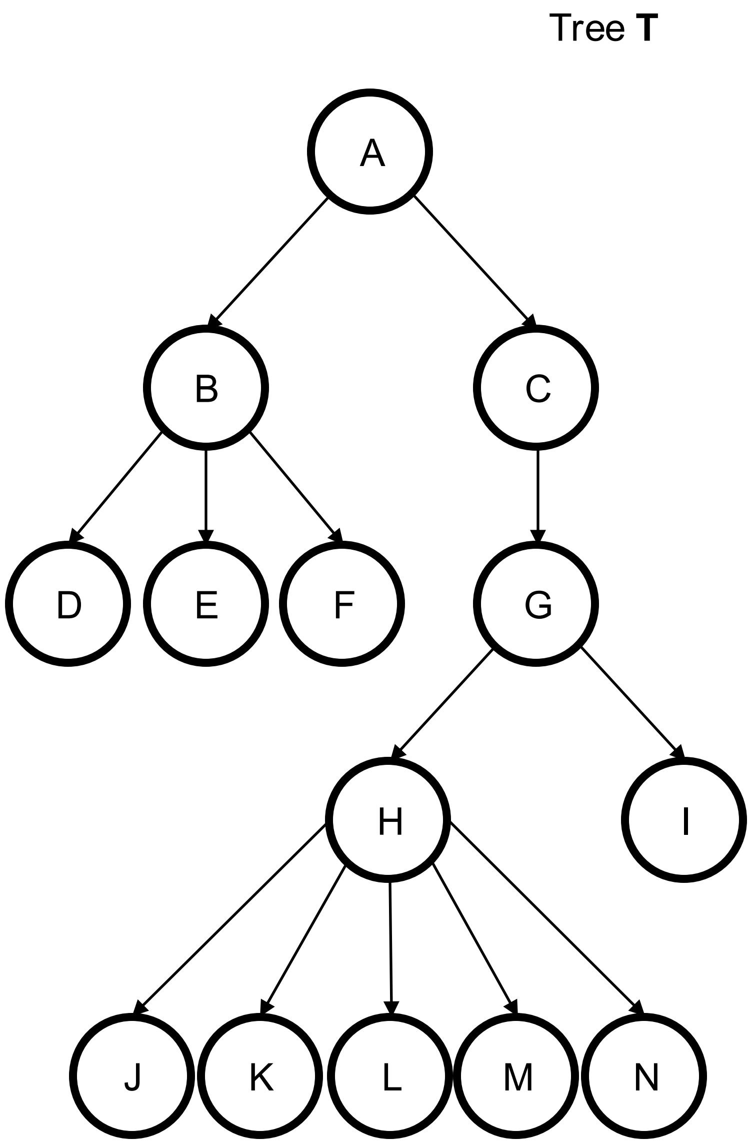
Any Questions?

Today

- Priority Queue ADT
- Tree Stuff
- Binary Min-Heap Data Structure
  - Basics, Properties, Operations
  - Array Representation
- Floyd's BuildHeap

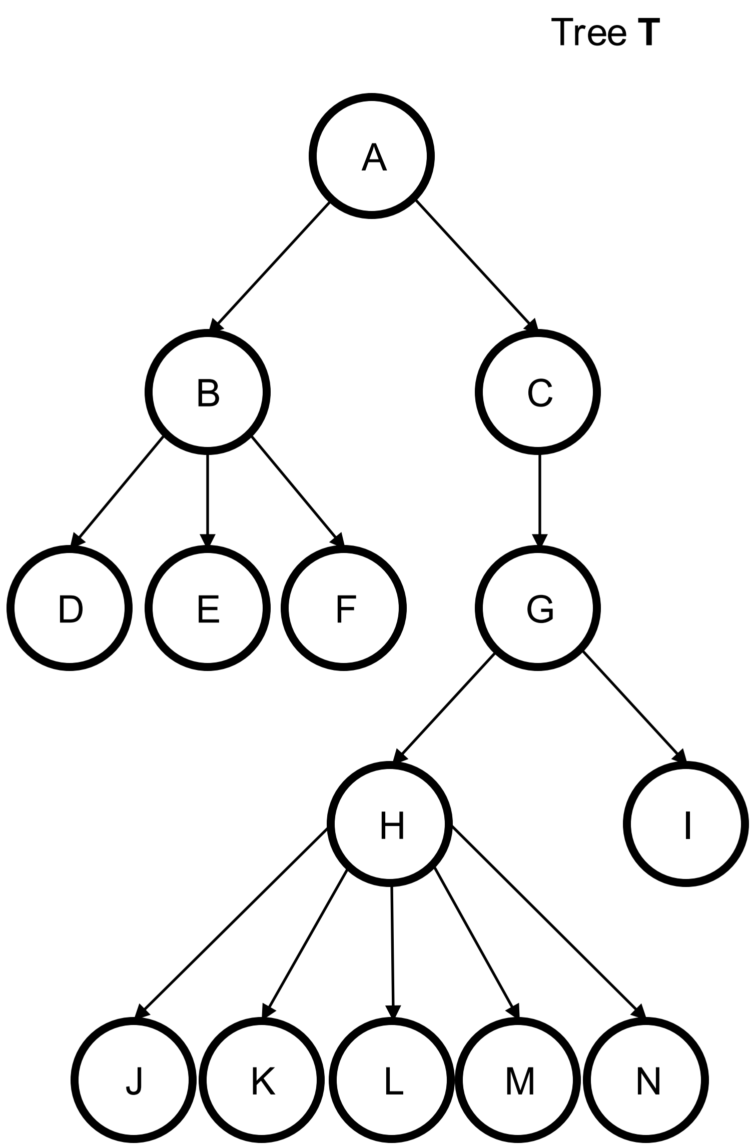
🌲Tree🌲 Terminology 1

- root(T):
- leaves(T):
- children(B):
- parent(H):
- siblings(E):
- ancestors(F):
- descendants(G):
- subtree(G):

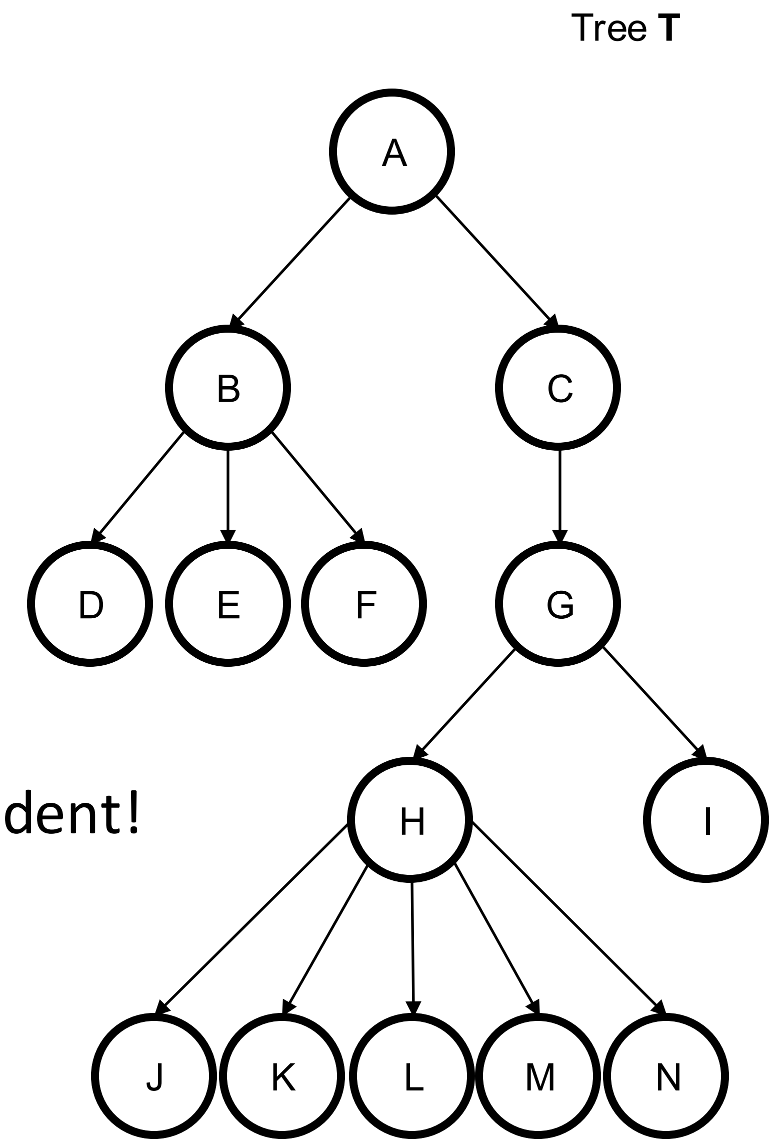


🌲Tree🌲 Terminology 2

- depth(B):
- height(G):
- height(T):
- degree(B):
- branchingFactor(T):



Tree Terminology 2

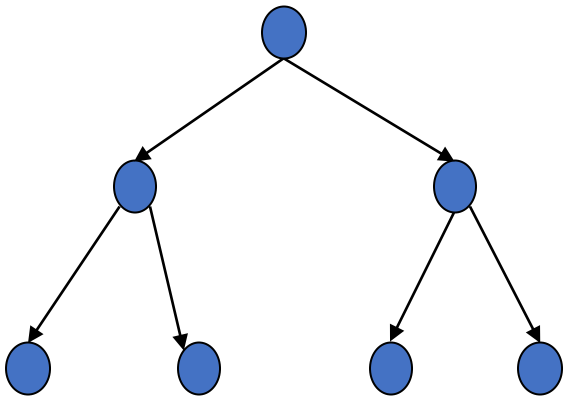


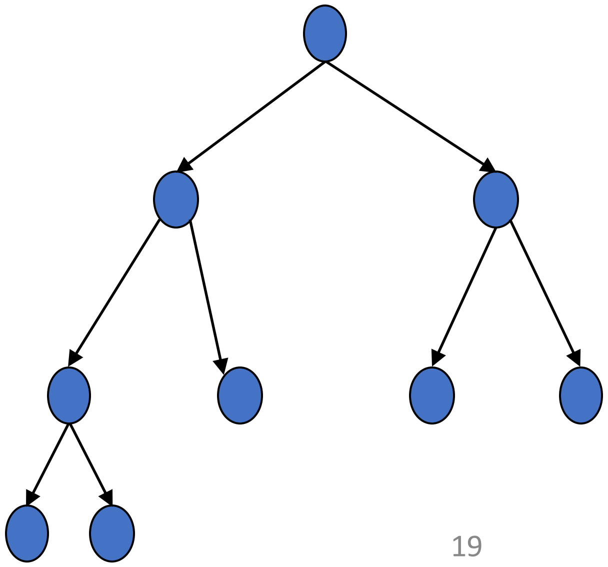
- depth(B):
- height(G):
- height(T):

- Height:
  - Count the arrows from node to deepest descendent!
- Depth:
  - Count the arrows from root to node!

🌲Tree🌲 Types

- **Binary** Tree:
  - Every node has max. **2** children
- n-ary Tree:
  - Every node has max. n children
- Perfect Tree:
  - Every row is completely full
- Complete Tree:
  - Every row is completely full except the bottom row
    - AND the bottom row is filled from left to right

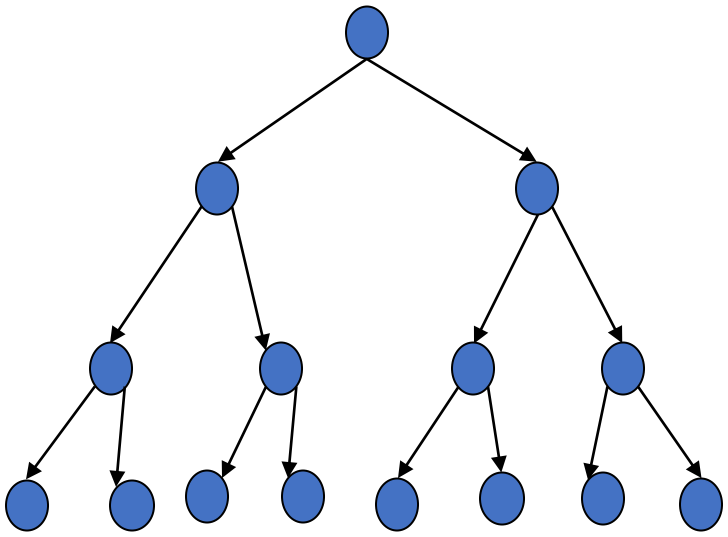




More on Perfect Tree

- Perfect Tree:
  - Every row is completely full

		# leaves
0	1	1
1	3	2
2	7	4
3	15	8
	?	



$$n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

$$\log n = \log 2^{h+1} - 1$$

$$h \in \mathcal{O}(\log n)$$

Any Questions?

Today

- Priority Queue ADT
- Tree Stuff
- Binary Min-Heap Data Structure
  - Basics, Properties, Operations
  - Array Representation
- Floyd's BuildHeap

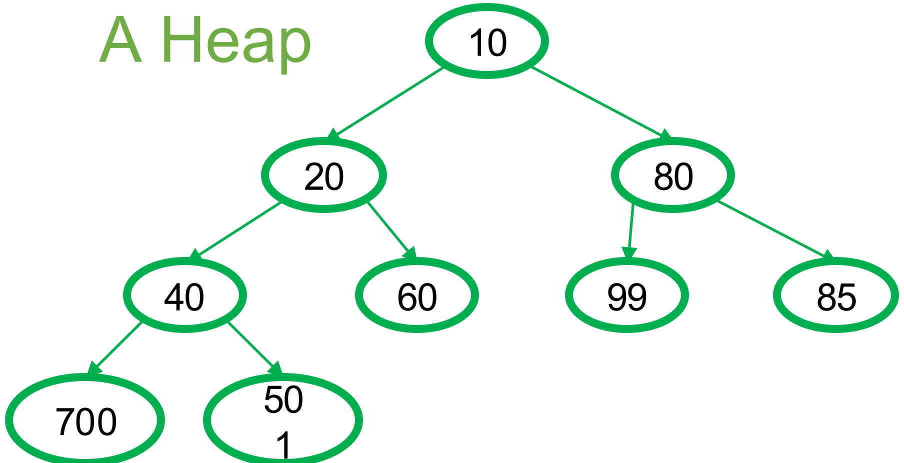
(Binary Min-) Heap: Basics & Properties

- More commonly known as a Binary Heap or simply a **Heap**

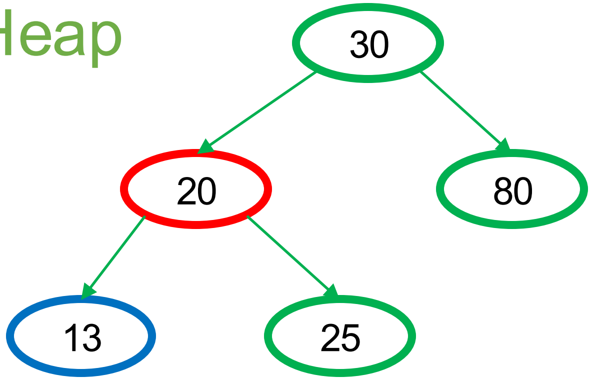
1. Structure Property:
   - A Complete (Binary) Tree
2. Heap Order Property:
   - Every (non-root) node has a priority value ≥ the priority value of its parent

- How is this different from a binary search tree?

A Heap

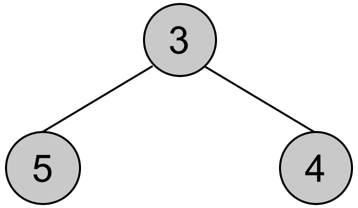


Not a Heap

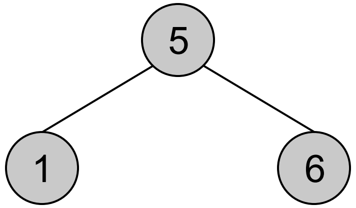


Heap or Not a Heap?

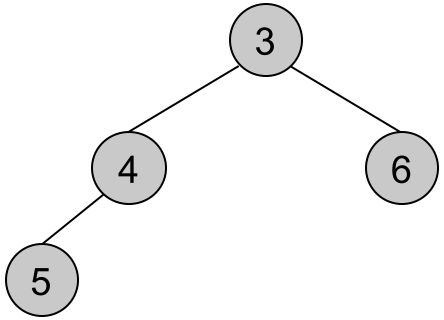
a)



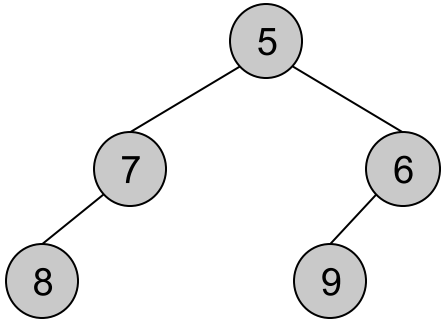
b)



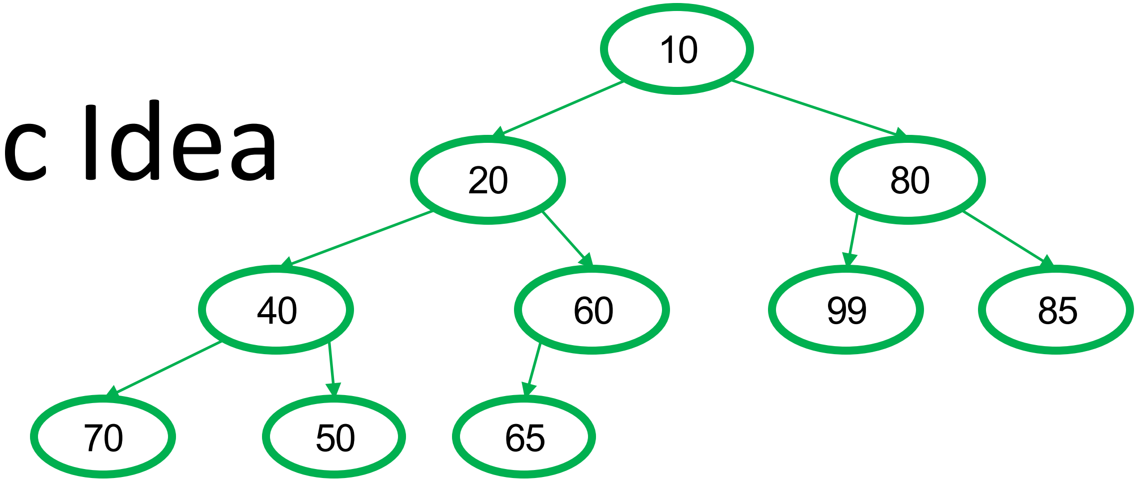
c)



d)



Heap: Operations Basic Idea



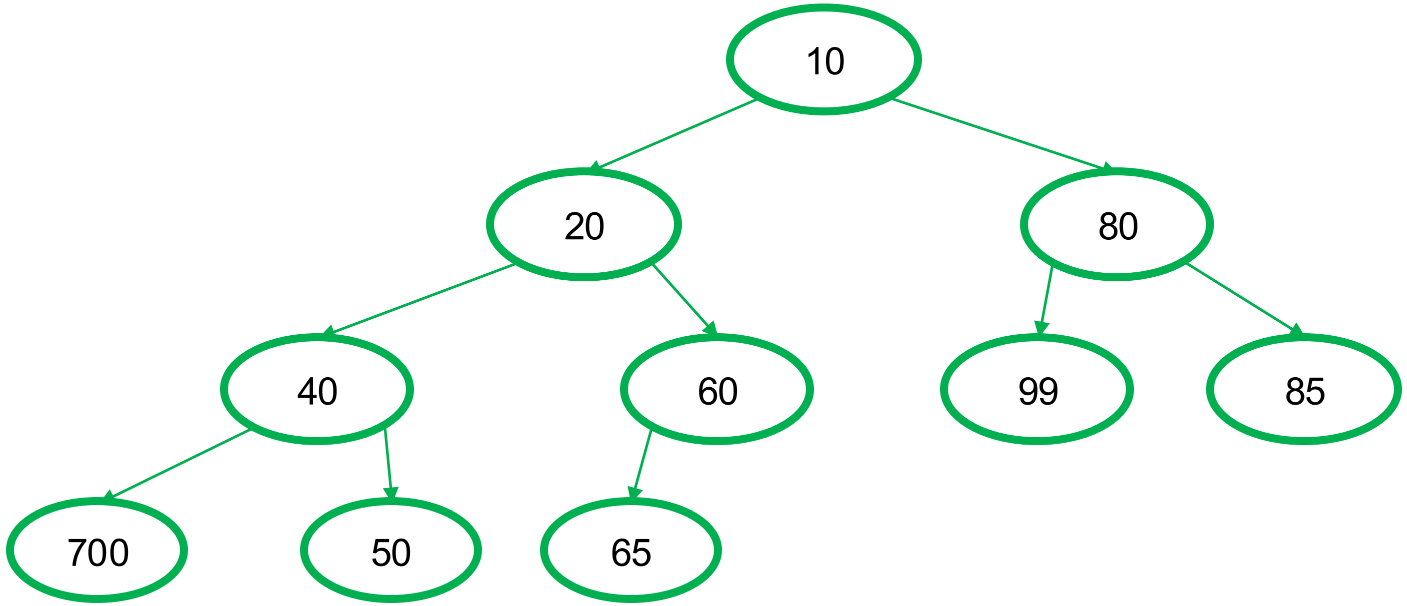
- `insert:`
  1. Structure Property:
     - Put new node in next position on bottom row
  2. Order Property:
     - `percolateUp()`
- `deleteMin:`
  1. `minElement = root.data`
  2. **Structure Property**:
     - Move right-most node in last row to root
  3. **Heap Order Property**:
     - `percolateDown` to restore

Overall strategy:

- *Preserve* ***Complete Tree*** *Structure Property*
- *This may break Heap Order Property*
- *Percolate to restore Heap Order Property*

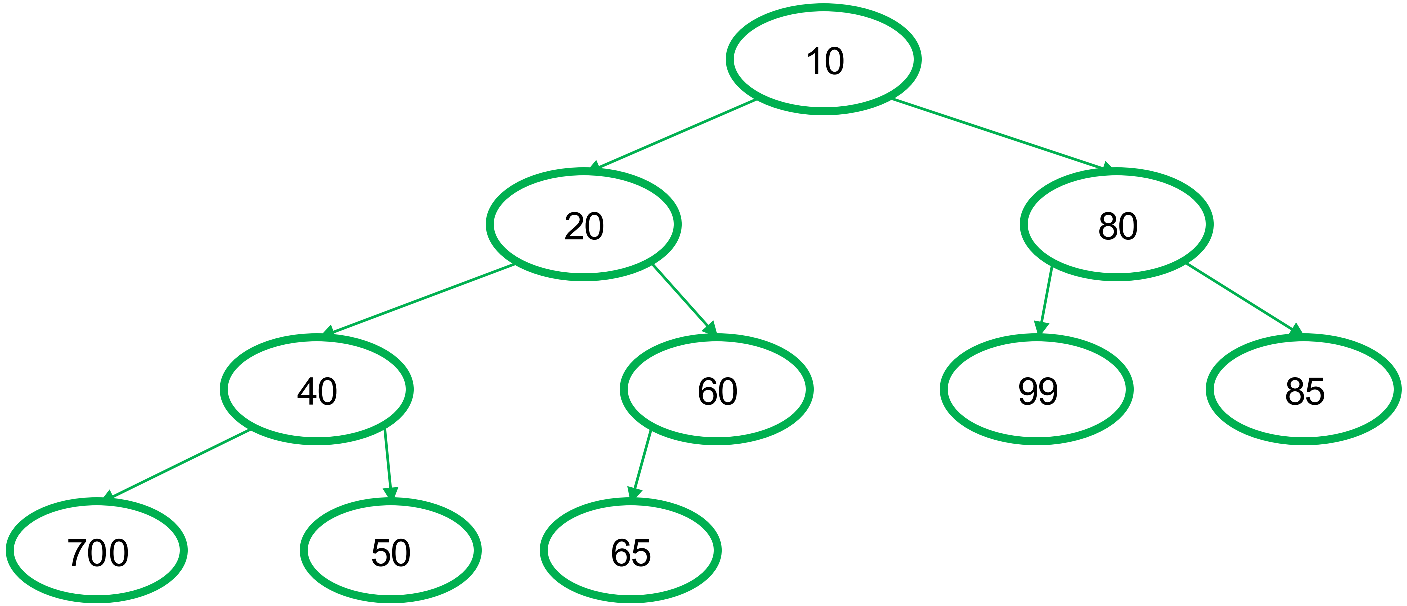
Heap: Operations (`insert`)

- `insert(16)`



Heap: Operations (`deleteMin`)

- `deleteMin()`



Any Questions?

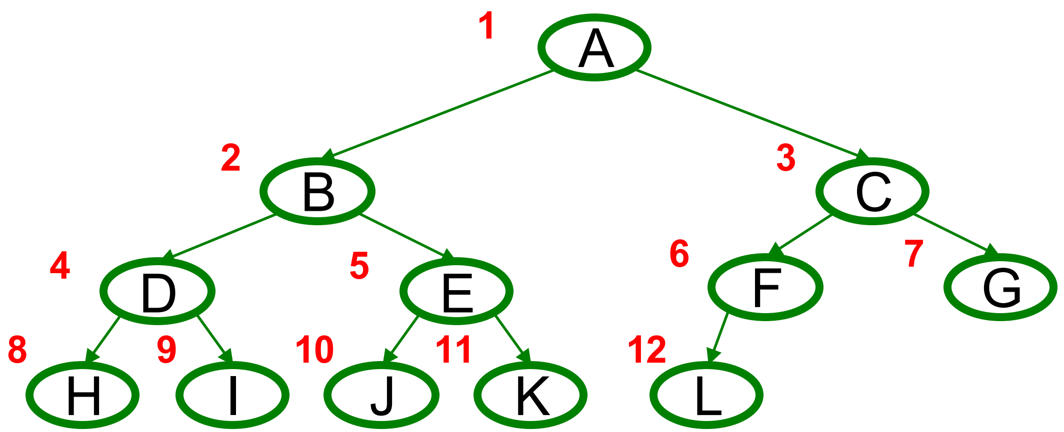
Today

- Priority Queue ADT
- Tree Stuff
- Binary Min-Heap Data Structure
  - Basics, Properties, Operations
  - Array Representation
- Floyd's BuildHeap

Heap: Array Representation

From node i,

- Left Child:
- Right Child:
- Parent:



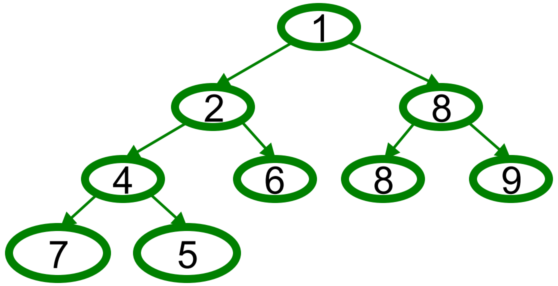
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

*Index 0 skipped so math is easier

Heap: `insert` Pseudocode (w/ Array)

```
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;
}
```

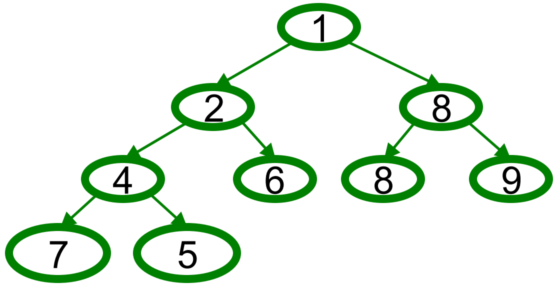


	10	20	80	40	60	85	99	700	50				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Heap: `deleteMin` Pseudocode (w/ Array)

```
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;
}
```



	1	2	8	4	6	8	9	7	5				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

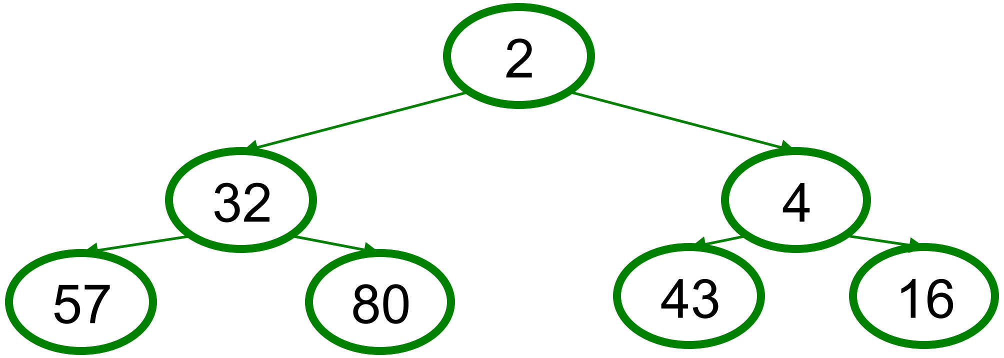
Heap: Operations Array Example (`insert`)

1. `insert`: 16, 32, 4, 57, 80, 43, 2

0	1	2	3	4	5	6	7

Heap: Operations Array Example (`deleteMin`)

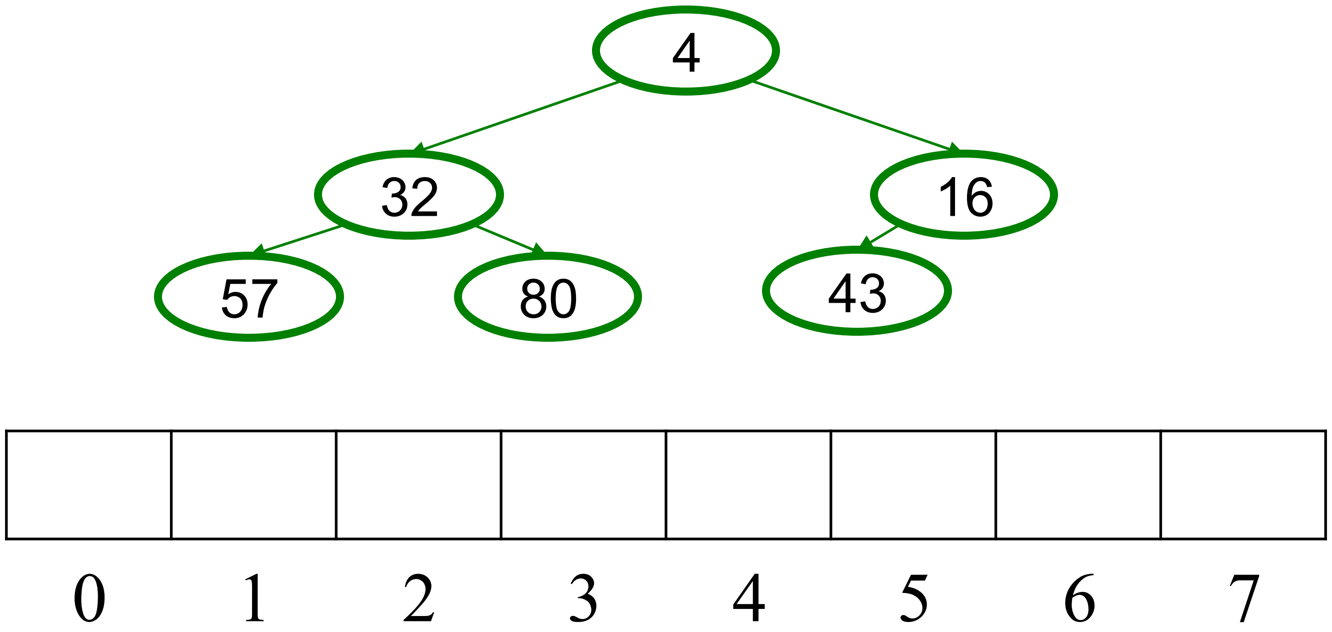
1. `deleteMin`



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

Heap: Operations Array Example (Solution)

1. `deleteMin`



Heap: Array Evaluation

Advantages:

1. Minimal wasted space:
    - If a tree node object, need pointers (expensive!)
2. Fast Lookups
    - Quick array lookup
    - Calculating index (multiplication + division by 2) is extremely fast

Disadvantages:

1. Resizing

Conclusion: It's too good so **almost always use array**

Heap: Other operations

- `decreaseKey(idx, Δ)` or `increaseKey(idx, Δ)`
  1. `arr[idx] -= Δ` or `arr[idx] += Δ`
  2. `percolateUp()` or `percolateDown()`

  Worst Case $\Theta(\log n)$

- `delete(idx)`
  1. `decreaseKey(idx, ∞)`
  2. `deleteMin()`

  Worst Case $\Theta(\log n)$

Heap: Note on `decrease/increaseKey`

- MORE COMMONLY CALLED `changePriority(key, prio)`
  1. Uses a map to go from `key` -> `idx`
  2. `arr[idx] = prio`
  3. `percolateUp()` or `percolateDown()`

  (Same as `decrease/increaseKey`)

Any Questions?

Heap: Building a Heap

Scenario: n elements into a blank Heap

- Call `insert()` n times
  - Runtime? $\mathcal{O}(n \log n)$

Can we do better?

- Yes! $\mathcal{O}(n)$ with Floyd's `buildHeap`

Heap: Floyd's `buildHeap`

Recall: Heap Properties

1. Structure Property: A Complete (Binary Tree)
2. Heap Order Property: All nodes' priority $\geq$ its parent's priority.

Floyd's `buildHeap` – $\mathcal{O}(n)$

1. Put the n elements in the array (any order fine)

	5	**3**	**…**	**1**	**7**
0	1	2			

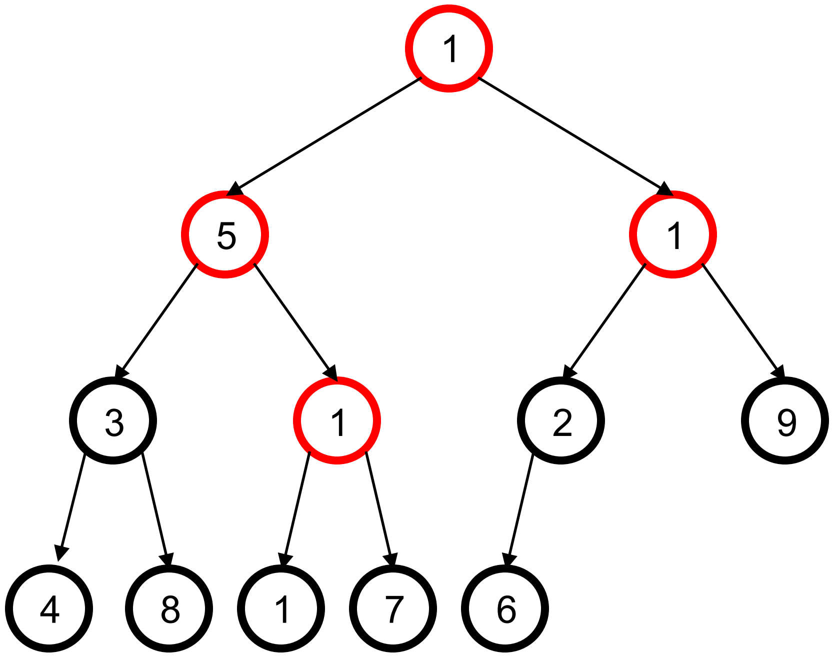
1. Just fix Heap Order Property:

`percolateDown`() from [one level above leaves] -> root

Heap: `buildHeap` Example

- `percolateDown()`, bottom-up:
  - Notice: leaves already Heap Order
  - Work up to root one at a time

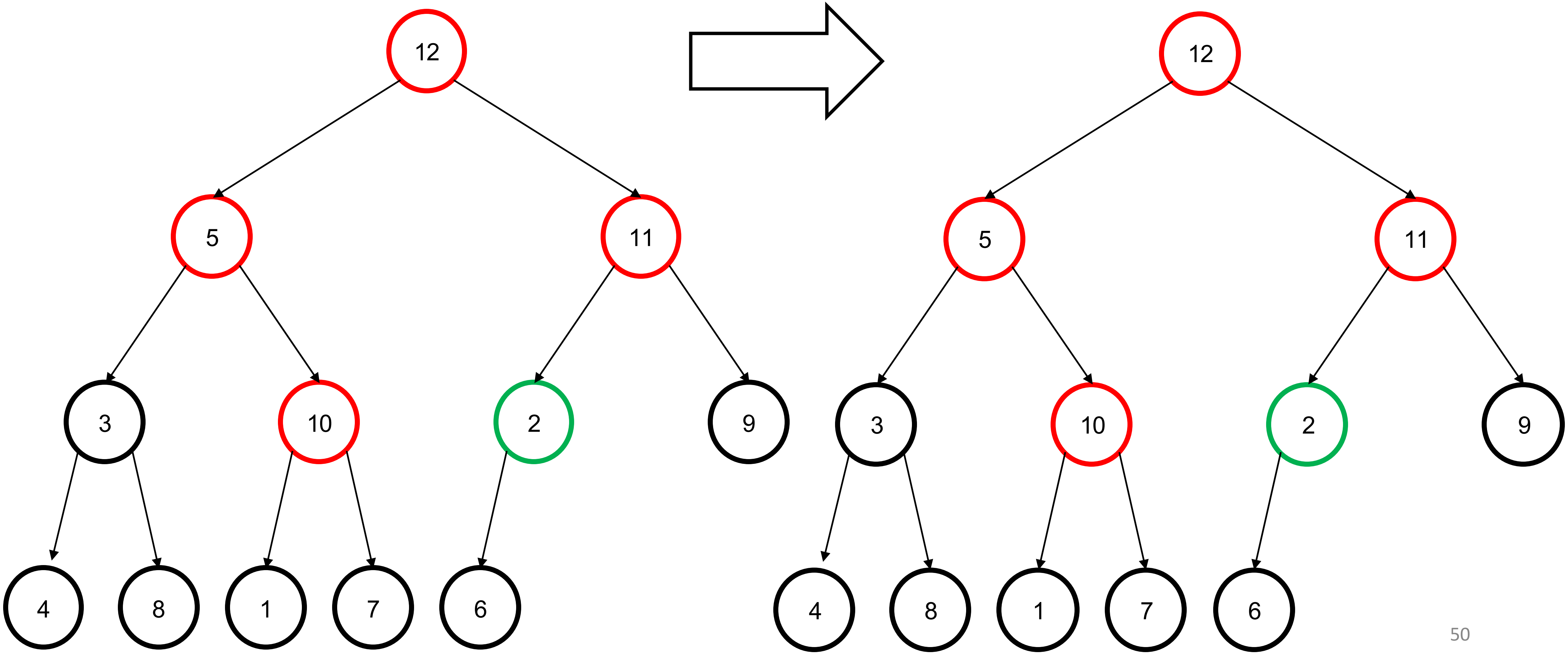
```
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val  = arr[i];
    hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```



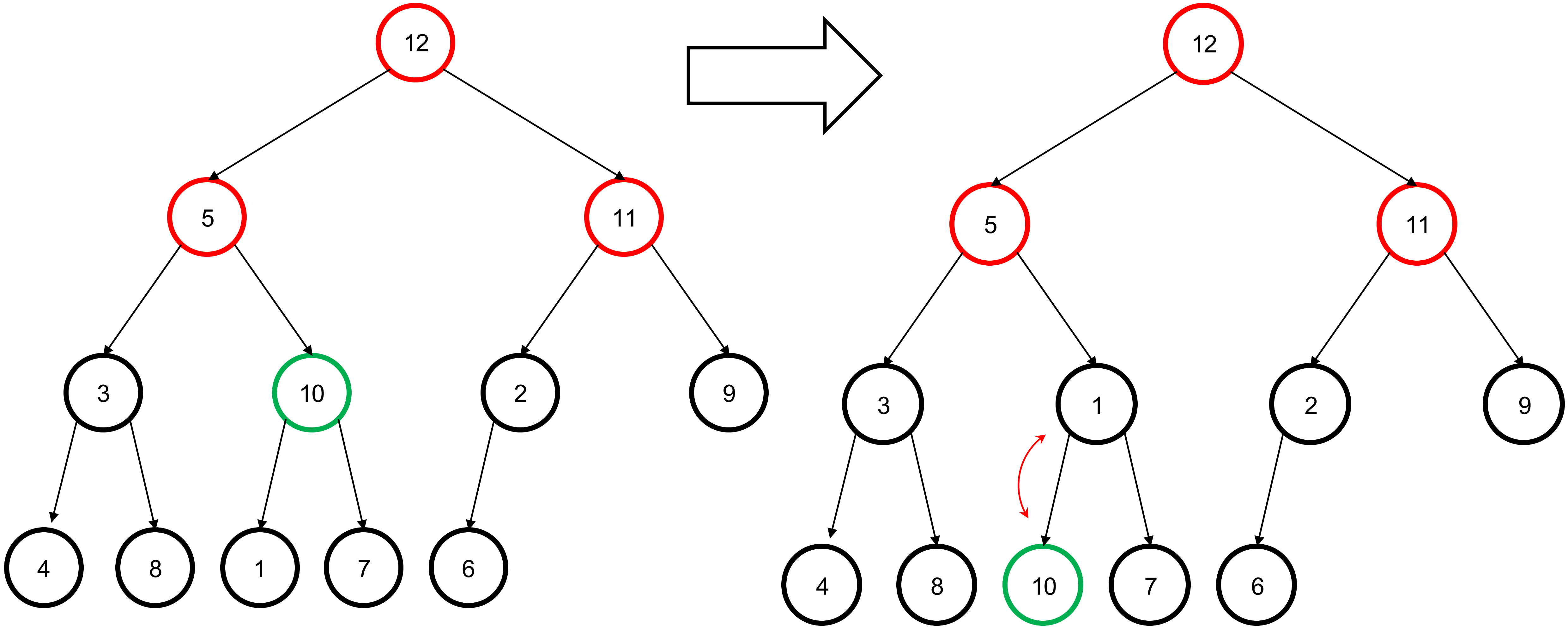
	12	5	11	3	10	2	9	4	8	1	7	6			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Heap: `buildHeap` Example (Solution 1)

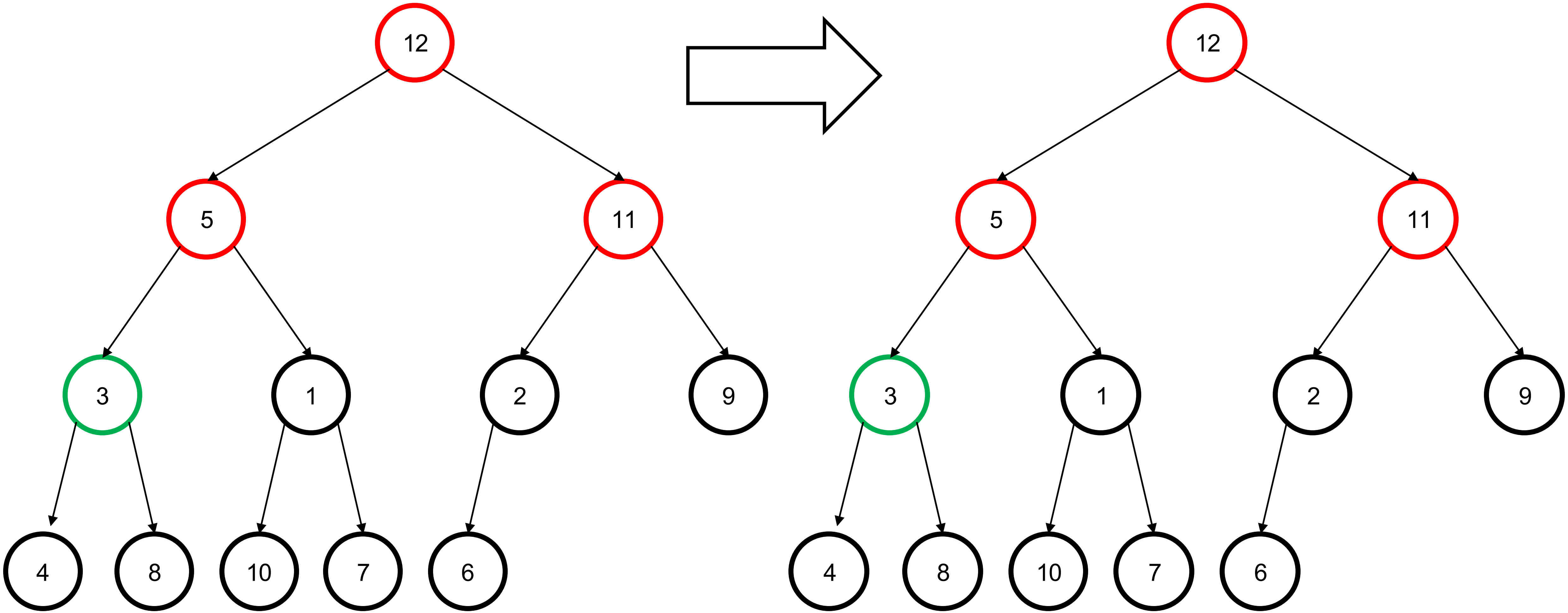
Red = Node that breaks Heap Order Property



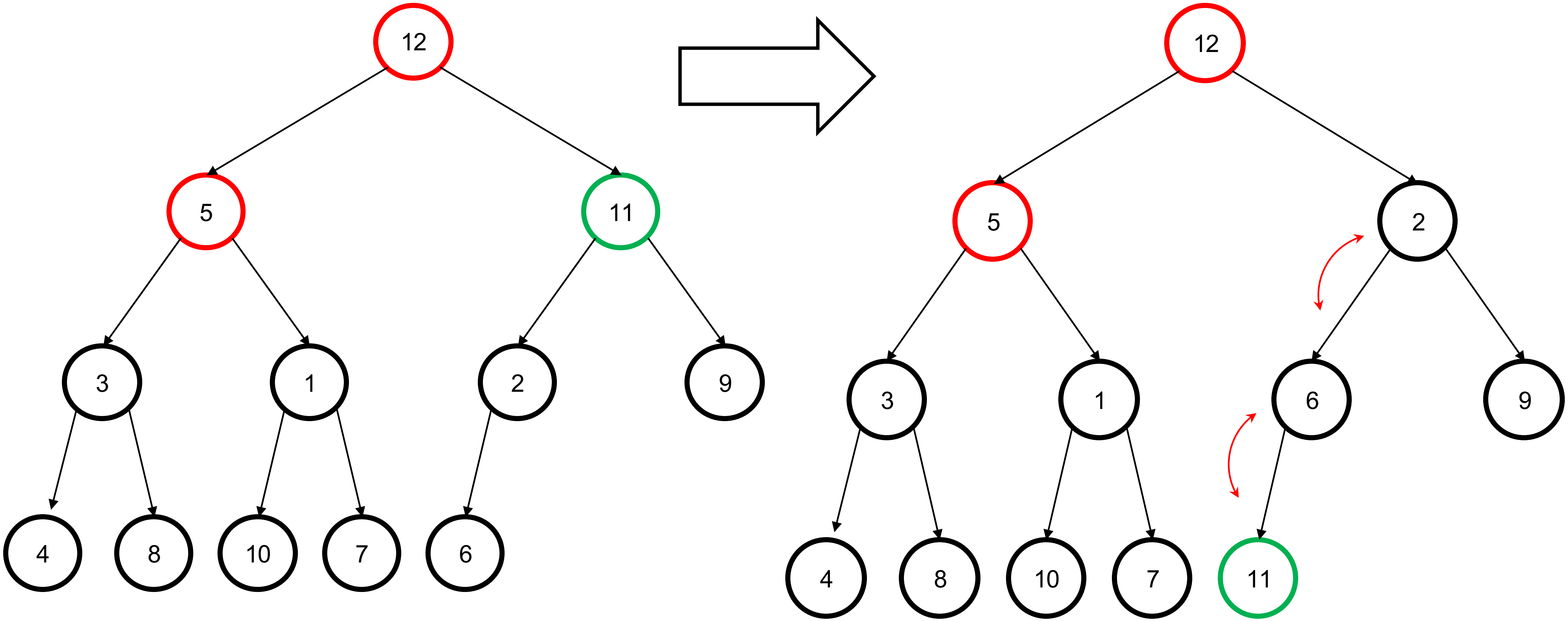
Heap: `buildHeap` Example (Solution 2)



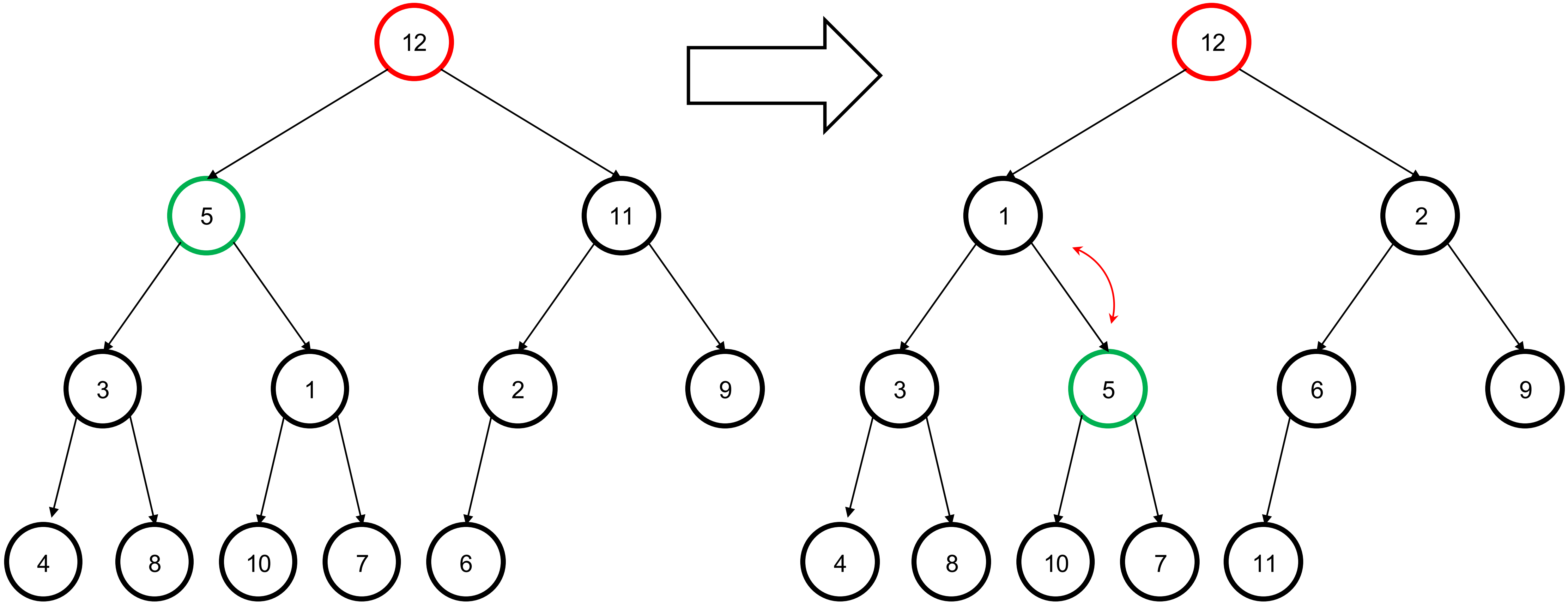
Heap: `buildHeap` Example (Solution 3)



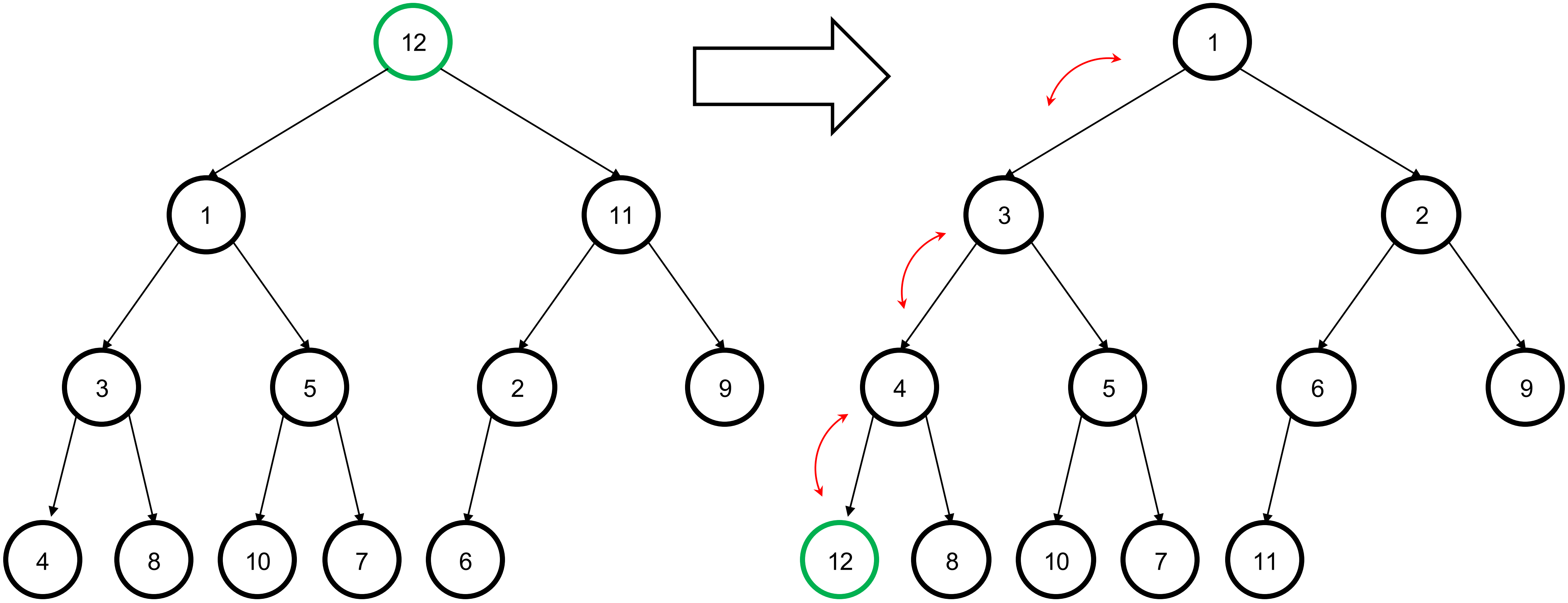
Heap: `buildHeap` Example (Solution 4)



Heap: `buildHeap` Example (Solution 5)



Heap: `buildHeap` Example (Solution 6)



`buildHeap` Correctness

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

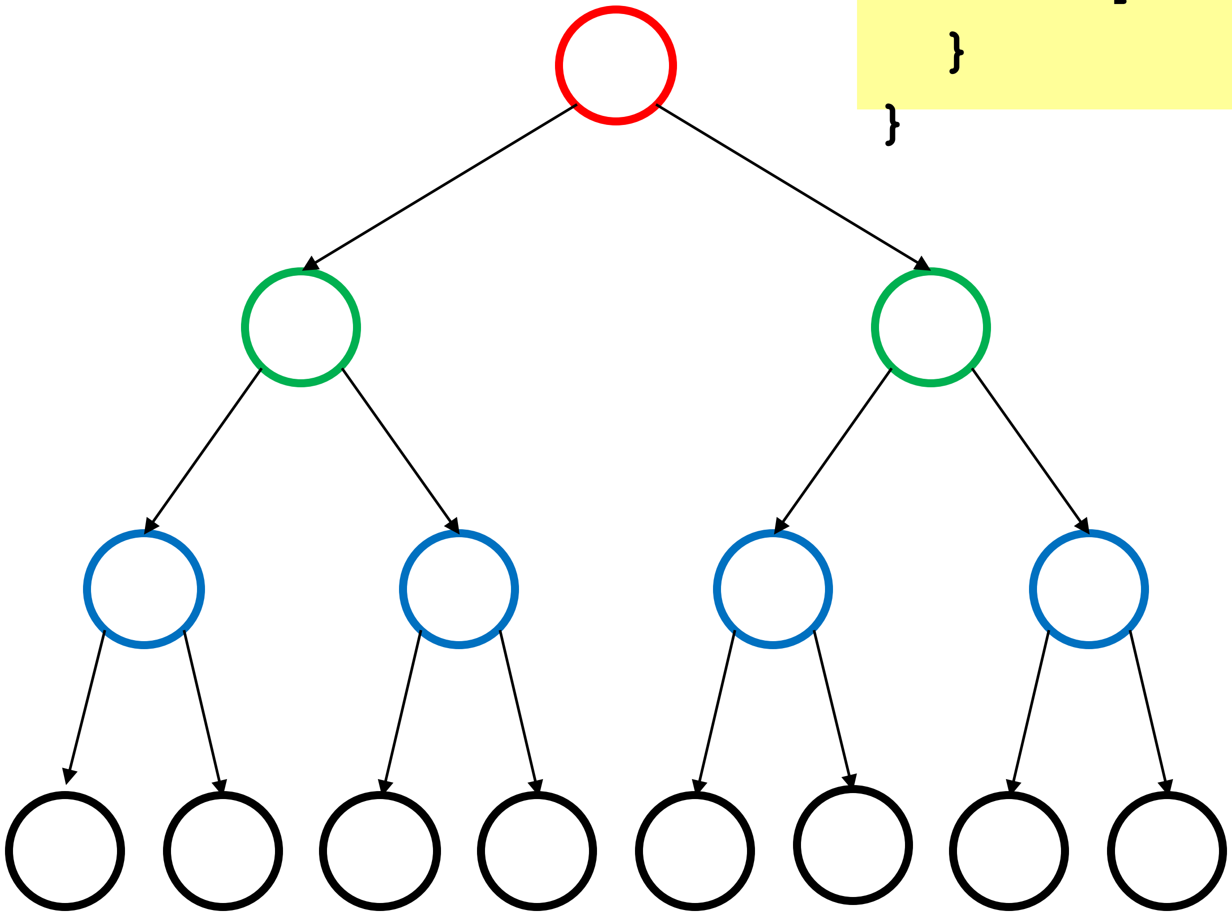
Loop Invariant: For all $j > i$, arr[j] is less than its children

- True initially: If $j > \frac{\text{size}}{2}$, 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

buildHeap Correctness

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



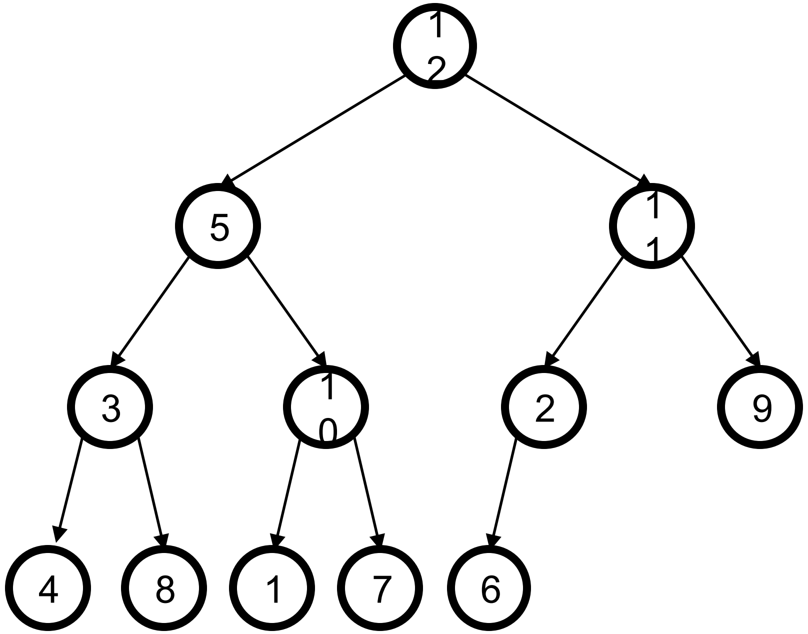
`buildHeap` Efficiency

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

- Runtime: $\mathcal{O}(n)$
- Total iterations: $n/2$
- 1. $\frac{1}{2}$ iterations: percolate at most one step so $\leq \frac{n}{4}$ cost
- 2. $\frac{1}{4}$ iterations: percolate at most two steps so $\leq \frac{2n}{8}$ cost
- 3. $\frac{1}{8}$ iterations: percolate at most three steps so $\leq \frac{3n}{16}$ cost

Summing cost: $\frac{n}{2} \cdot \underbrace{\left(\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \frac{5}{32} + \ \ldots\right)}_{= 2}$

which is $\mathcal{O}(n)$



Weiss 6.3.4

Any Questions?

Timeline

- Priority Queue ADT
- Tree Stuff
- Binary Min-Heap Data Structure
  - Basics, Properties, Operations
  - Array Representation
- Floyd's `buildHeap`
- Asymptotic Analysis: Recursive
  - Writing a Recurrence Relation
  - Solving a Recurrence Relation 1: Unrolling
  - Solving a Recurrence Relation 2: Tree Method