CSE 332 Summer 2026
Lecture 5: Priority Queues

Michael Whitmeyer

http://www.cs.uw.edu/332

ADT: Queue

- What is it?
  - A “First In First Out” (FIFO) collection of items
- What Operations do we need?
  - Enqueue
    - Add a new item to the queue
  - Dequeue
    - Remove the “oldest” item from the queue
  - Is_empty
    - Indicate whether or not there are items still on the queue

ADT: Priority Queue

- What is it?
  - A collection of items and their “priorities”
  - Allows quick access/removal to the “top priority” thing
- What Operations do we need?
  - **insert**(item, priority)
    - Add a new item to the PQ with indicated priority
    - Usually, smaller priority value means more important
  - **extract**
    - Remove and return the “top priority” item from the queue
  - Is_empty
    - Indicate whether or not there are items still on the queue
- Note: the “priority” value can be any type/class so long as it’s comparable (i.e. you can use “<“ or “compareTo” with it)

Priority Queue Example 1

```
PriorityQueue PQ = new PriorityQueue();
PQ.insert(5,5)
PQ.insert(6,6)
PQ.insert(1,1)
PQ.insert(3,3)
PQ.insert(8,8)
Print(PQ.extract)
Print(PQ.extract)
Print(PQ.extract)
Print(PQ.extract)
Print(PQ.extract)
```

Prints:
1
3
5
6
8

Priority Queue Example 2

```
PriorityQueue PQ = new PriorityQueue();
PQ.insert(5,5)
PQ.insert(6,6)
PQ.insert(3,3)
Print(PQ.extract)
PQ.insert(1,1)
Print(PQ.extract)
Print(PQ.extract)
PQ.insert(8,8)
Print(PQ.extract)
Print(PQ.extract)
```

Prints:
3
1
5
6
8

Applications?

Thinking through implementations

Data Structure	Worst case time to insert	Worst case time to extract
Unsorted Array		
Unsorted Linked List		
Sorted Array		
Sorted Linked List		
Binary Search Tree		

For simplicity, Assume we know the maximum size of the PQ in advance (otherwise we'd do an amortized analysis, but get the same answers…)

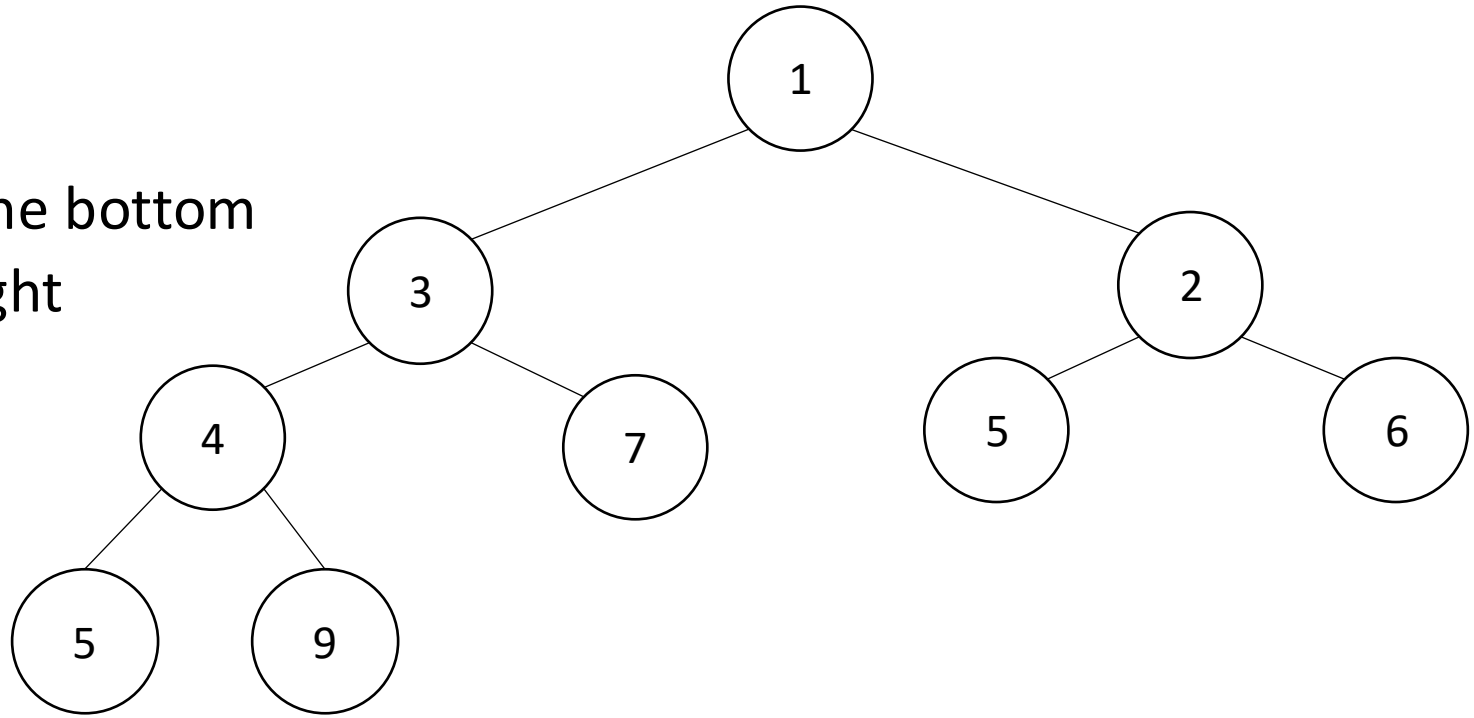
Thinking through implementations (solutions)

Data Structure	Worst case time to insert	Worst case time to extract
Unsorted Array	$\Theta(1)$	$\Theta(n)$
Unsorted Linked List	$\Theta(1)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(1)$
Sorted Linked List	$\Theta(n)$	$\Theta(1)$
Binary Search Tree	$\Theta(n)$	$\Theta(n)$
Binary Heap	$\Theta(\log n)$	$\Theta(\log n)$

For simplicity, Assume we know the maximum size of the PQ in advance (otherwise we'd do an amortized analysis, but get the same answers…)

Trees for Heaps

- Binary Trees:
  - The branching factor is 2
  - Every node has ≤ 2 children
- Complete Tree:
  - All “layers” are full, except the bottom
  - Bottom layer filled left-to-right



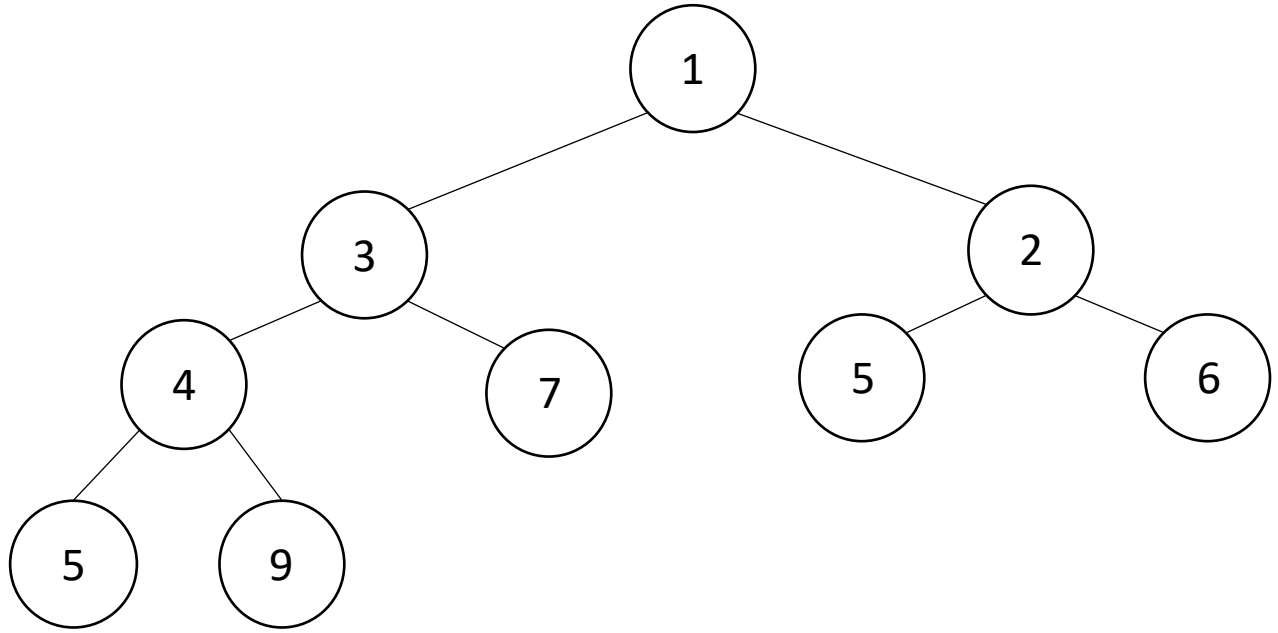
Interlude: Tree Measurements Challenge!

- What is the maximum number of total nodes in a binary tree of height h?
  - $2^{h+1} - 1$
  - $\Theta(2^h)$
- If I have n nodes in a binary tree, what is its minimum height?
  - $\Theta(\log n)$

- Heap Idea:
  - If n values are inserted into a **complete** tree, the height will be roughly $\log n$
  - Ensure each insert and extract requires just one "trip" from root to leaf

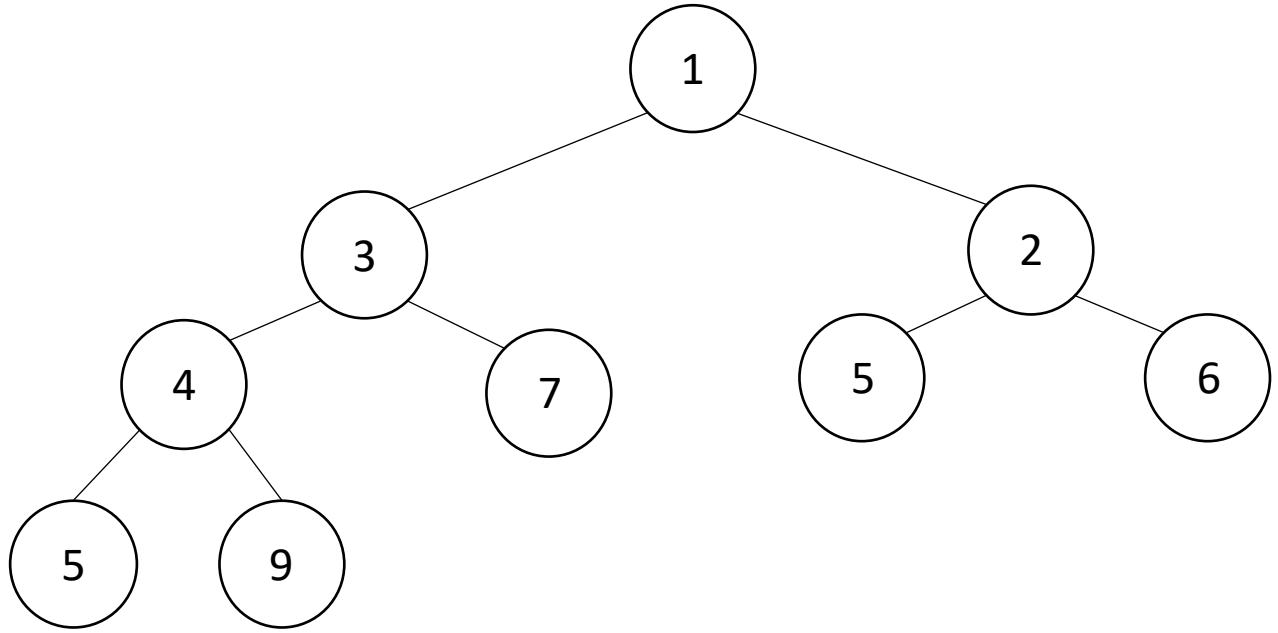
Heap – Priority Queue Data Structure

- Idea: We need to keep some structure/ordering, but not entirely sorted
- **Invariant 1**: tree will be "complete"
  - All layers full except possibly last one, filled left to right
- **Invariant 2**: priority(parent) ≤ priority(children)
  - "Heap Property"

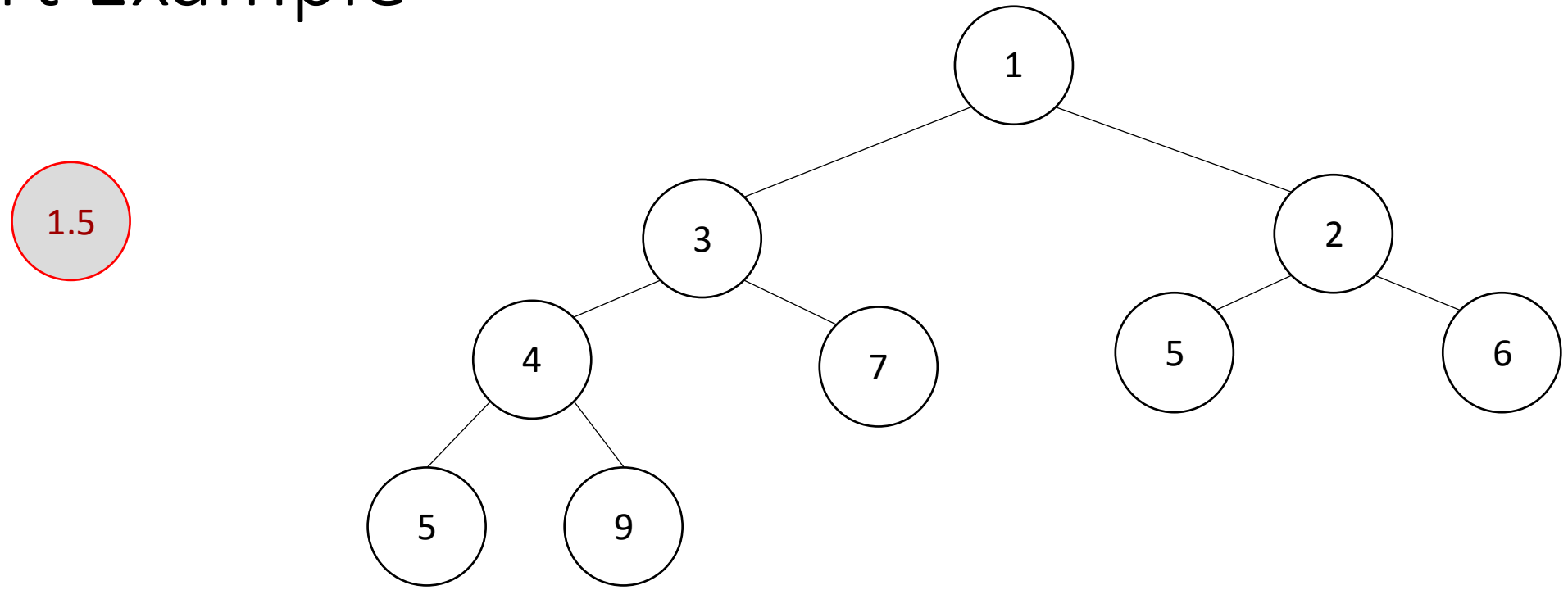


(Min) Heap Data Structure

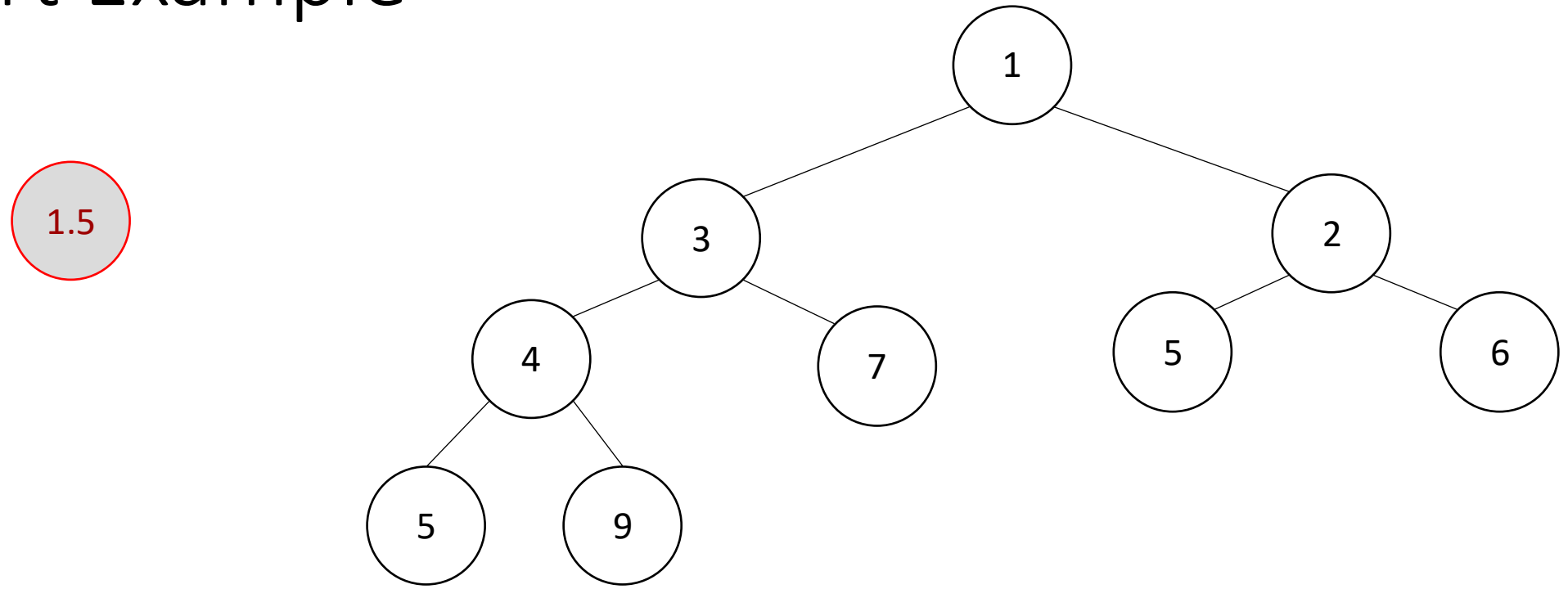
- Where is the min?
- How do I insert?
- How do I extract?
- How to do it in Java?



Heap Insert Example



Heap Insert Example



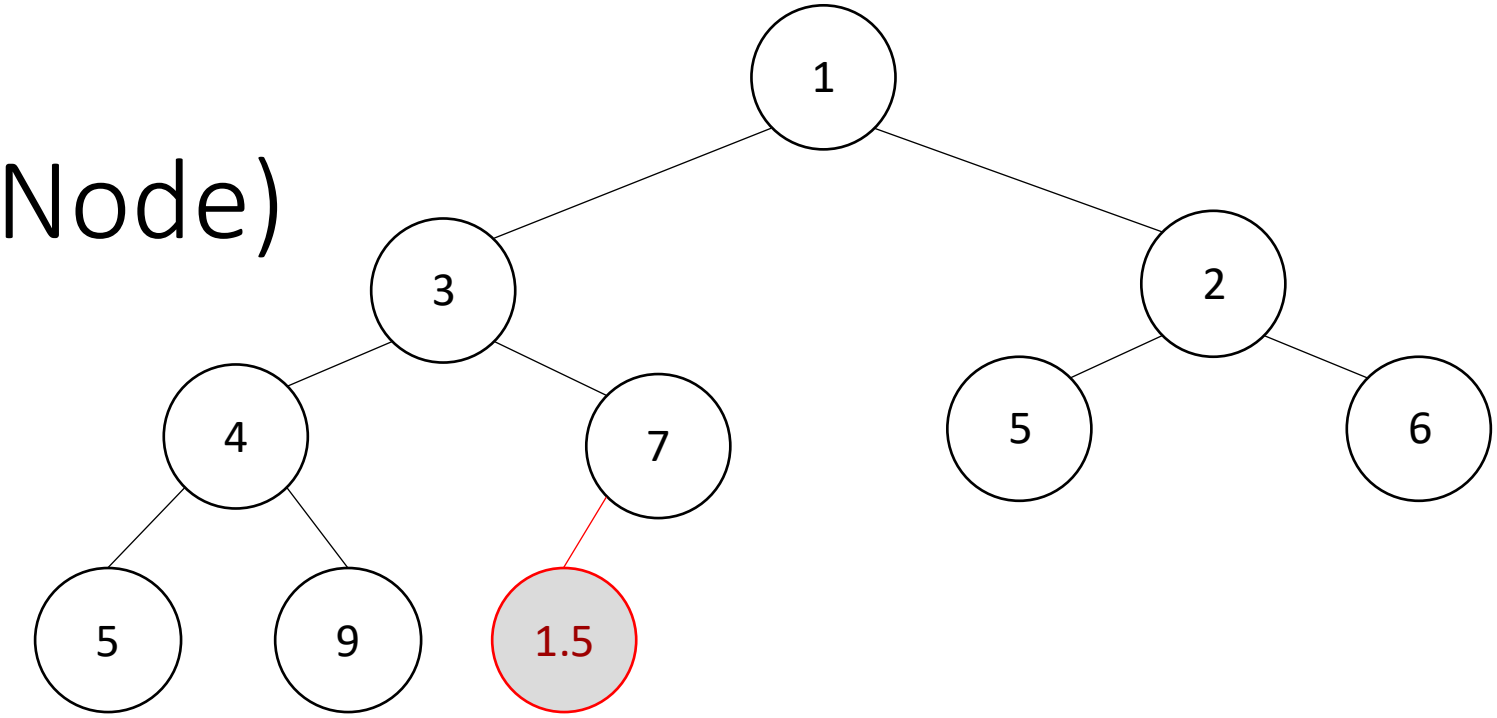
insert(item, priority):

put item in the “next open” spot (keep tree complete)

while (priority < parent’s priority):

swap item with parent

Heap Insert (New Node)



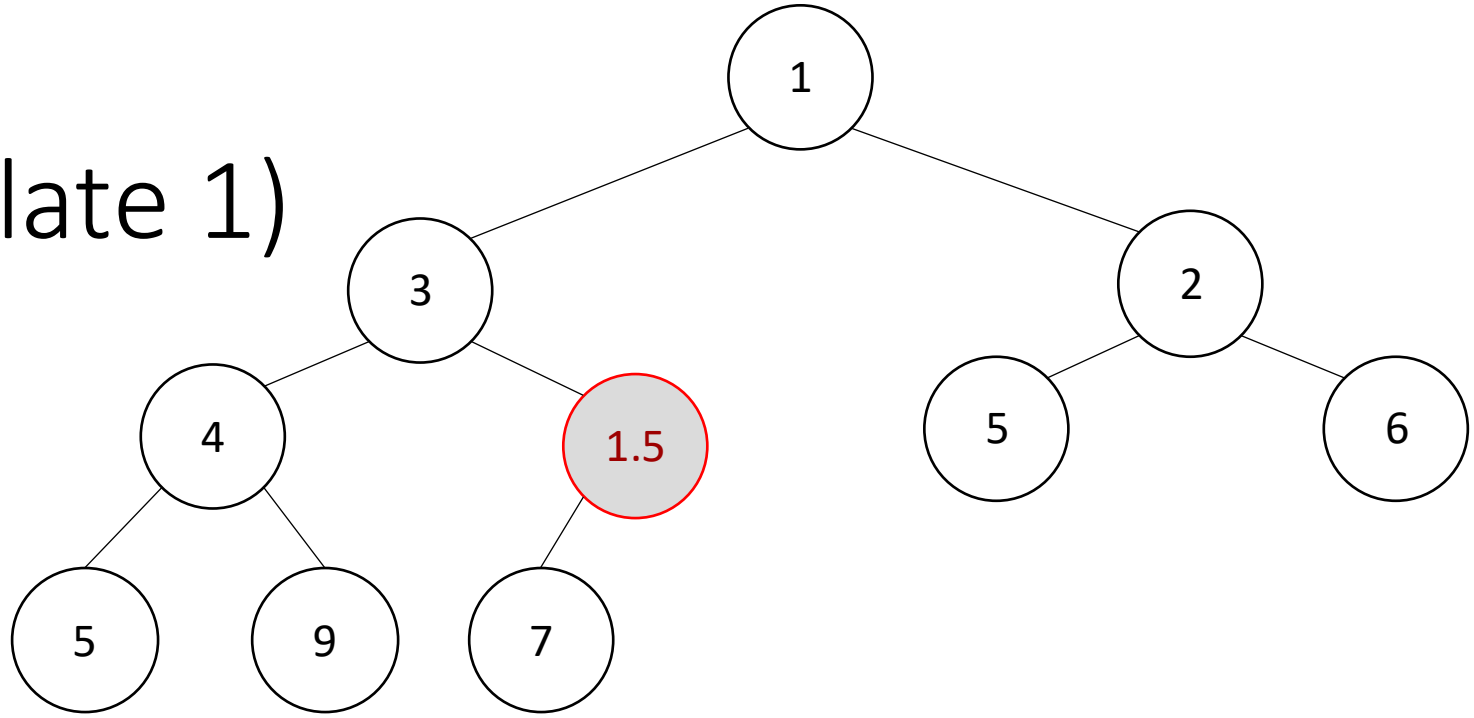
insert(item, priority):

put item in the “next open” spot (keep tree complete)

while (priority < parent’s priority):

swap item with parent

Heap Insert (Percolate 1)



insert(item, priority):

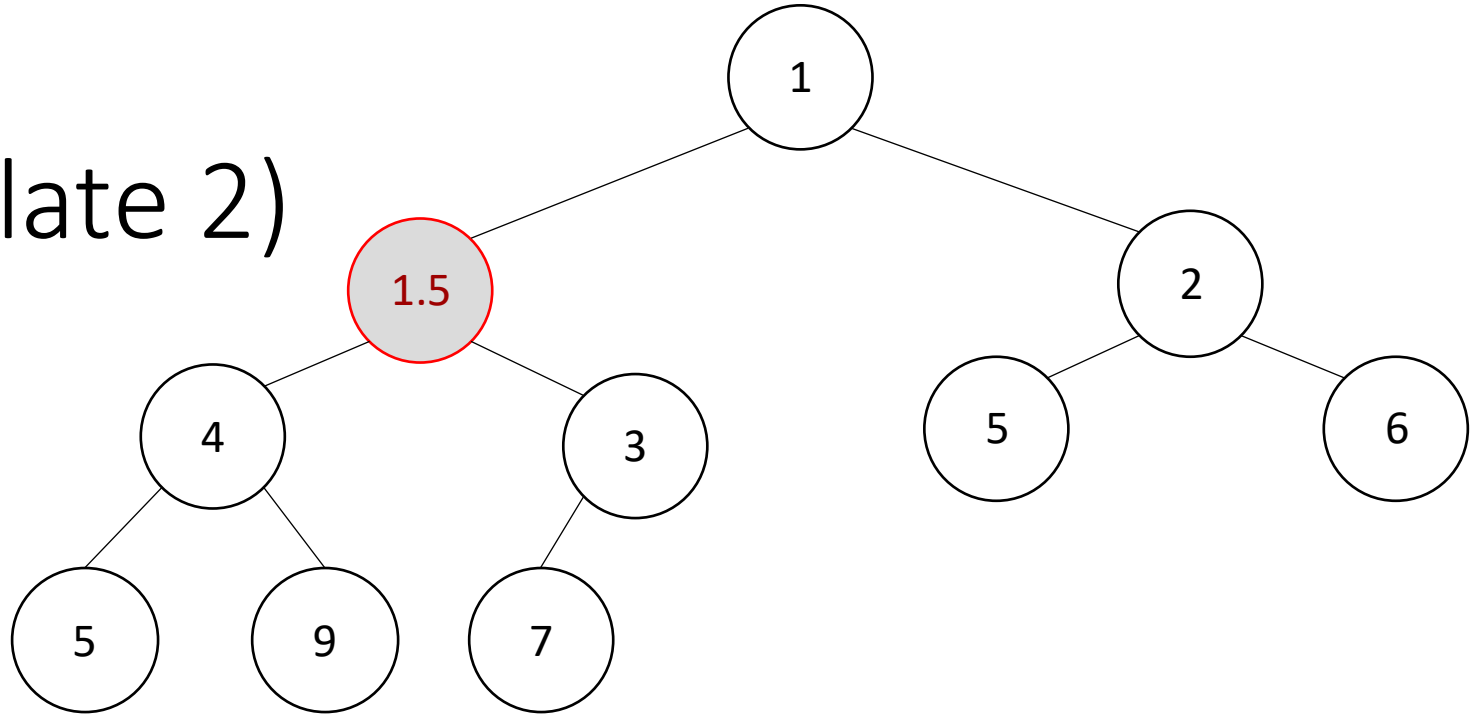
put item in the “next open” spot (keep tree complete)

while (priority < parent’s priority):

swap item with parent

Percolate Up

Heap Insert (Percolate 2)



insert(item, priority):

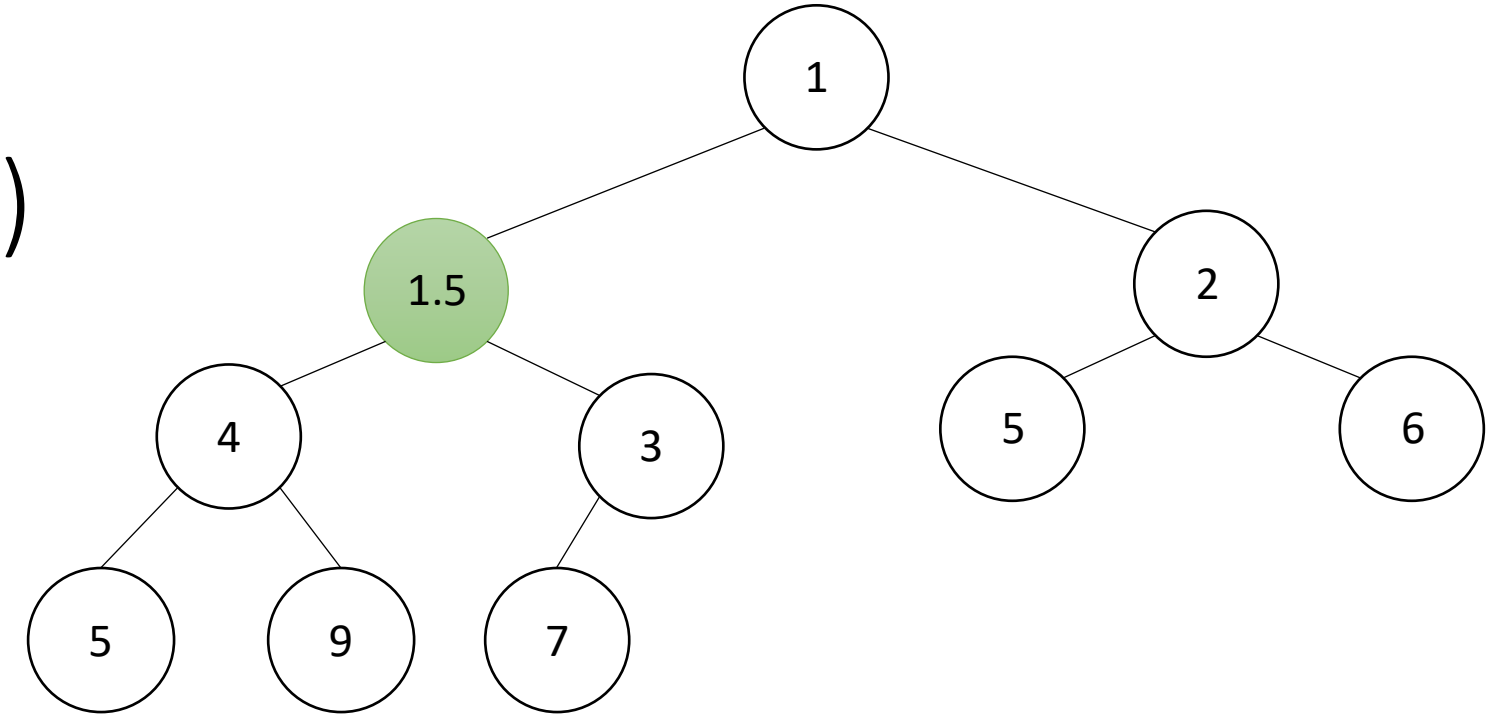
put item in the “next open” spot (keep tree complete)

while (priority < parent’s priority):

swap item with parent

Percolate Up

Heap Insert (Done)



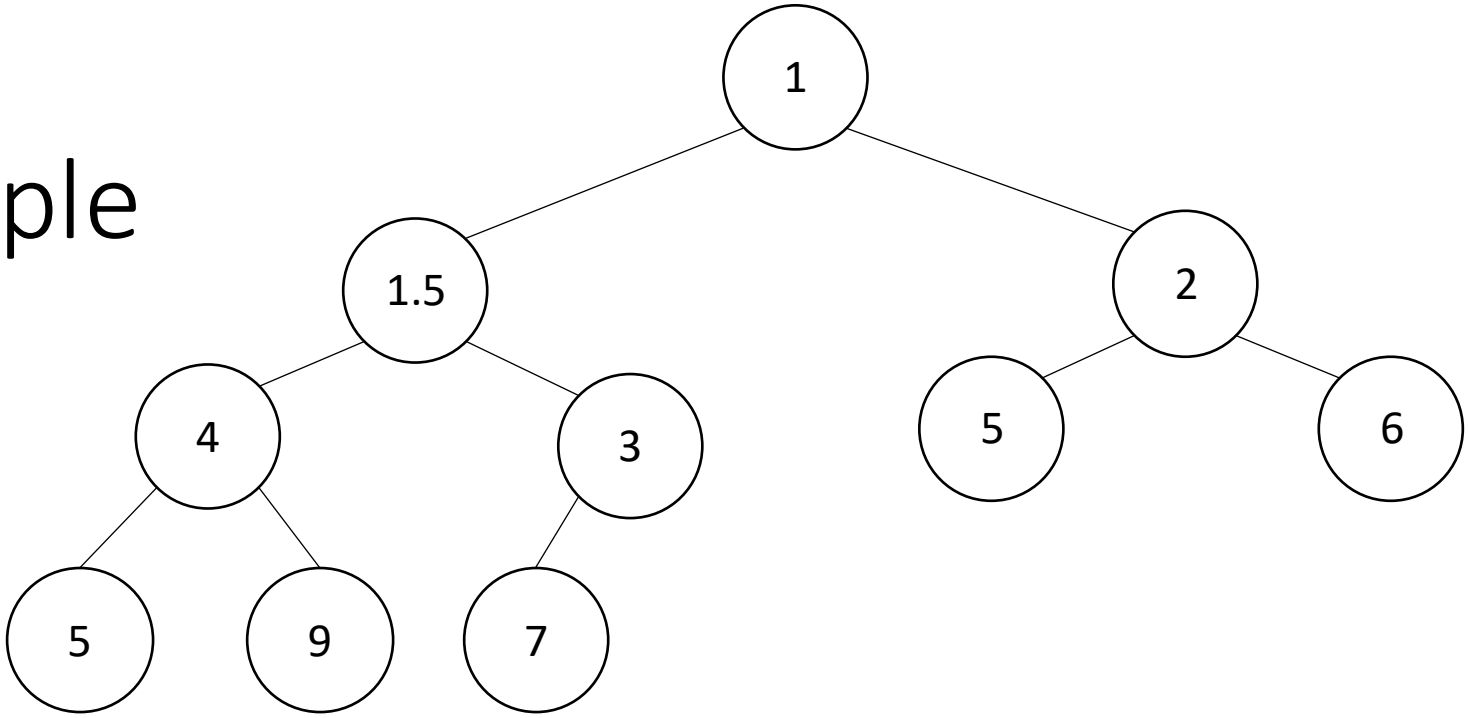
insert(item, priority):

put item in the “next open” spot (keep tree complete)

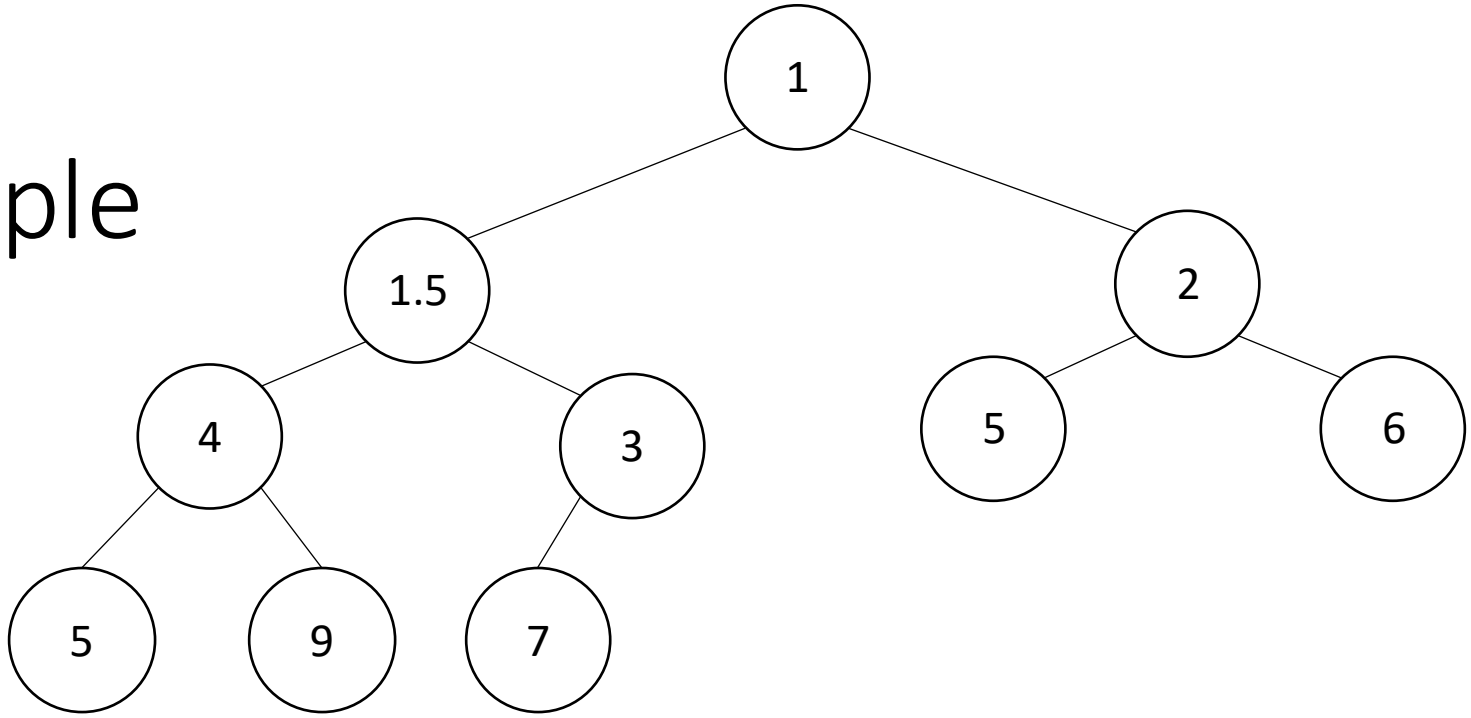
while (priority < parent’s priority):

swap item with parent

Heap extract Example



Heap extract Example



extract():

min = root

curr = bottom-right item

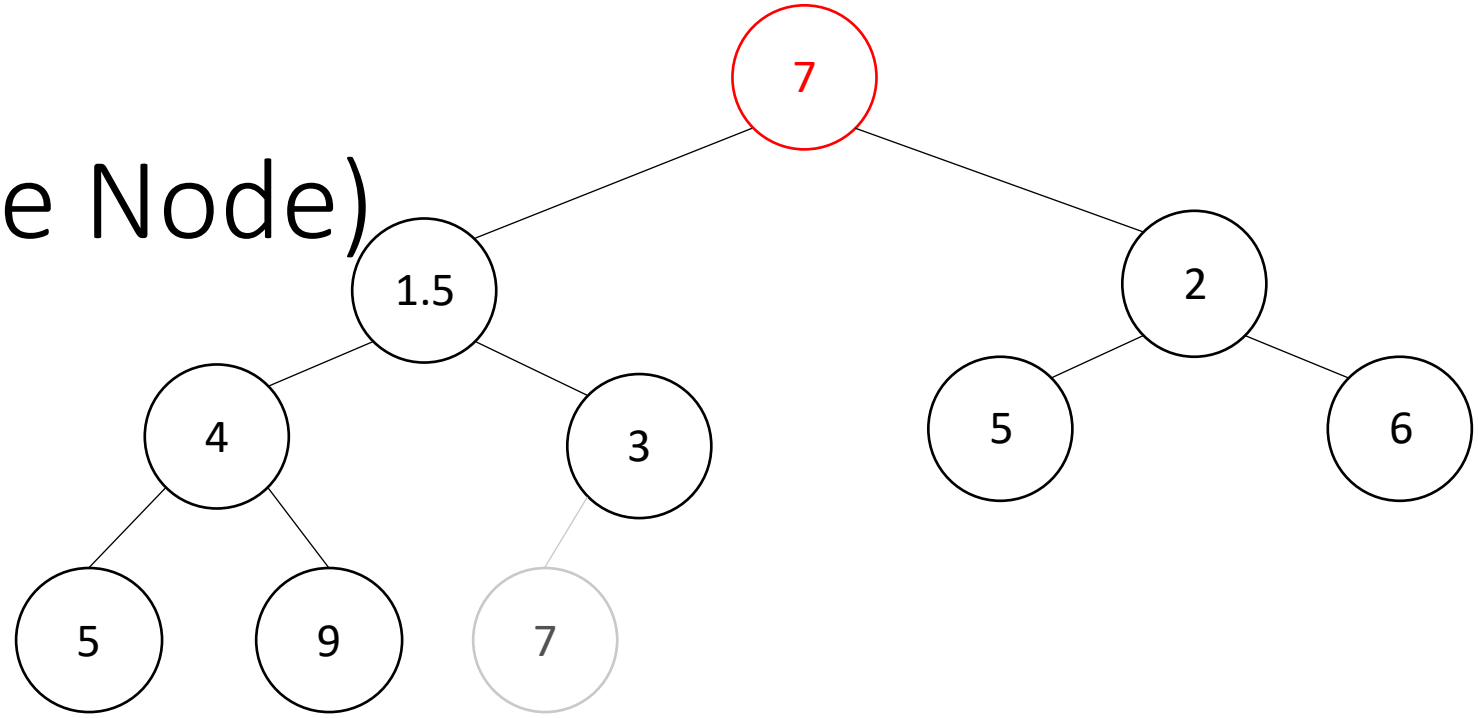
move curr to the root

while(curr > curr.left OR curr > curr.right):

swap curr with its smallest child

return min

Heap extract (Move Node)



extract()

min = root

curr = bottom-right item

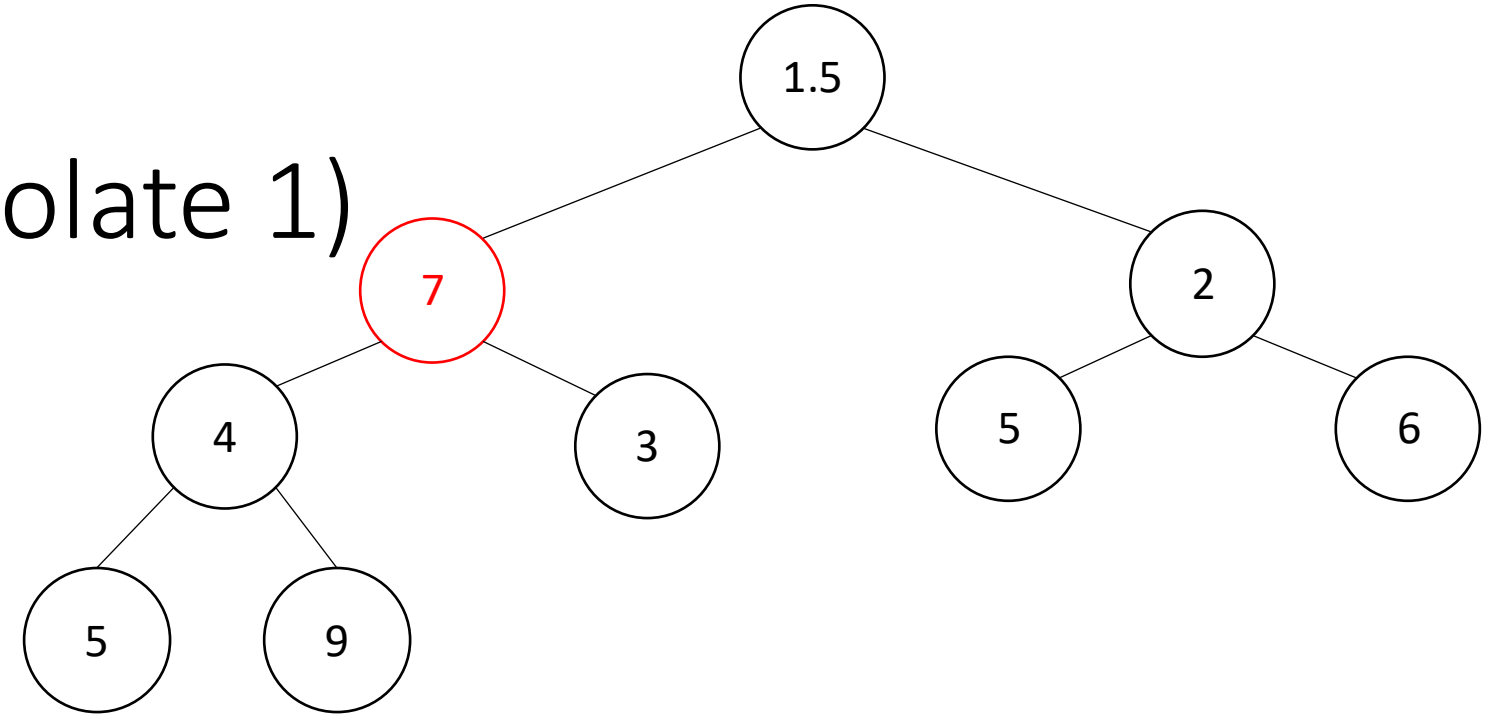
move curr to the root

while(curr > curr.left OR curr > curr.right):

swap curr with its smallest child

return min

Heap extract (Percolate 1)



extract():

min = root

curr = bottom-right item

move curr to the root

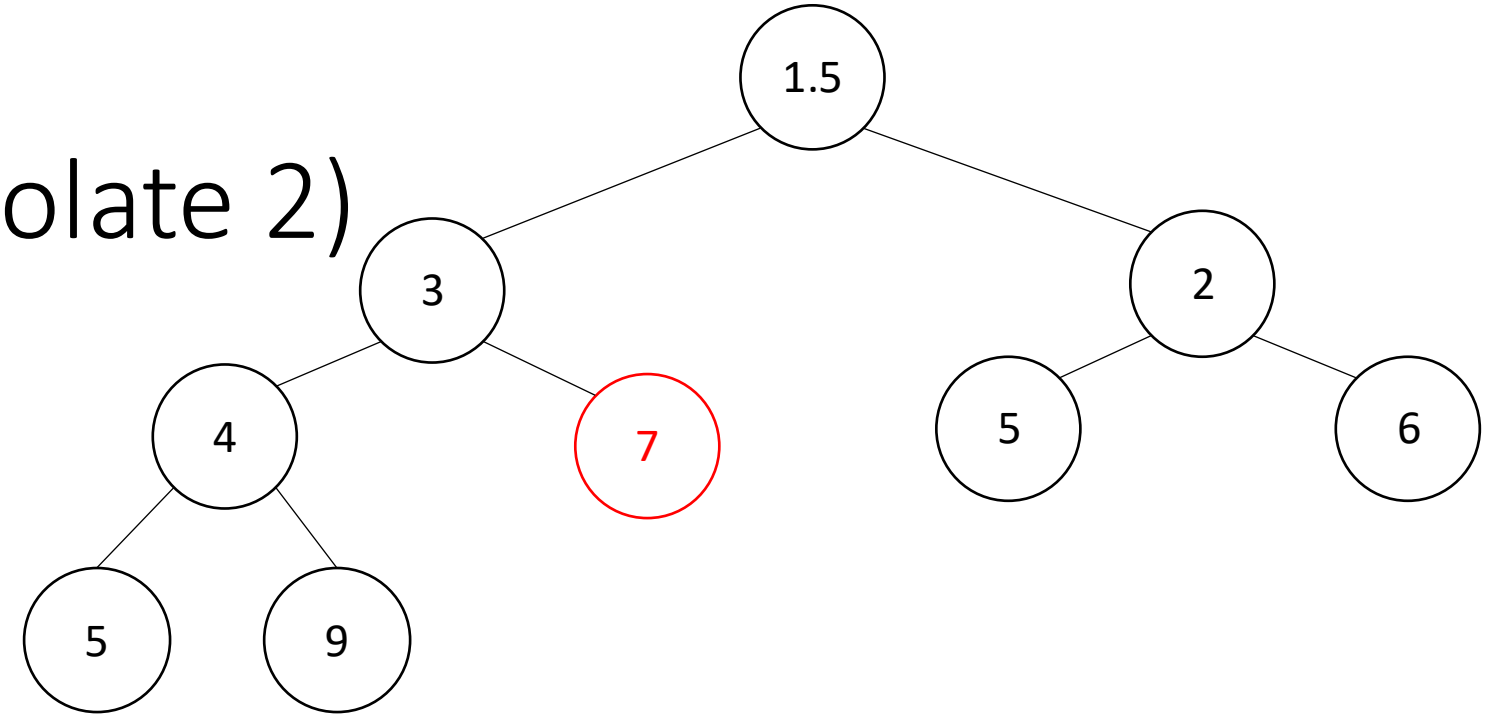
while(curr > curr.left OR curr > curr.right):

swap curr with its smallest child

Percolate Down

return min

Heap extract (Percolate 2)



extract():

min = root

curr = bottom-right item

move curr to the root

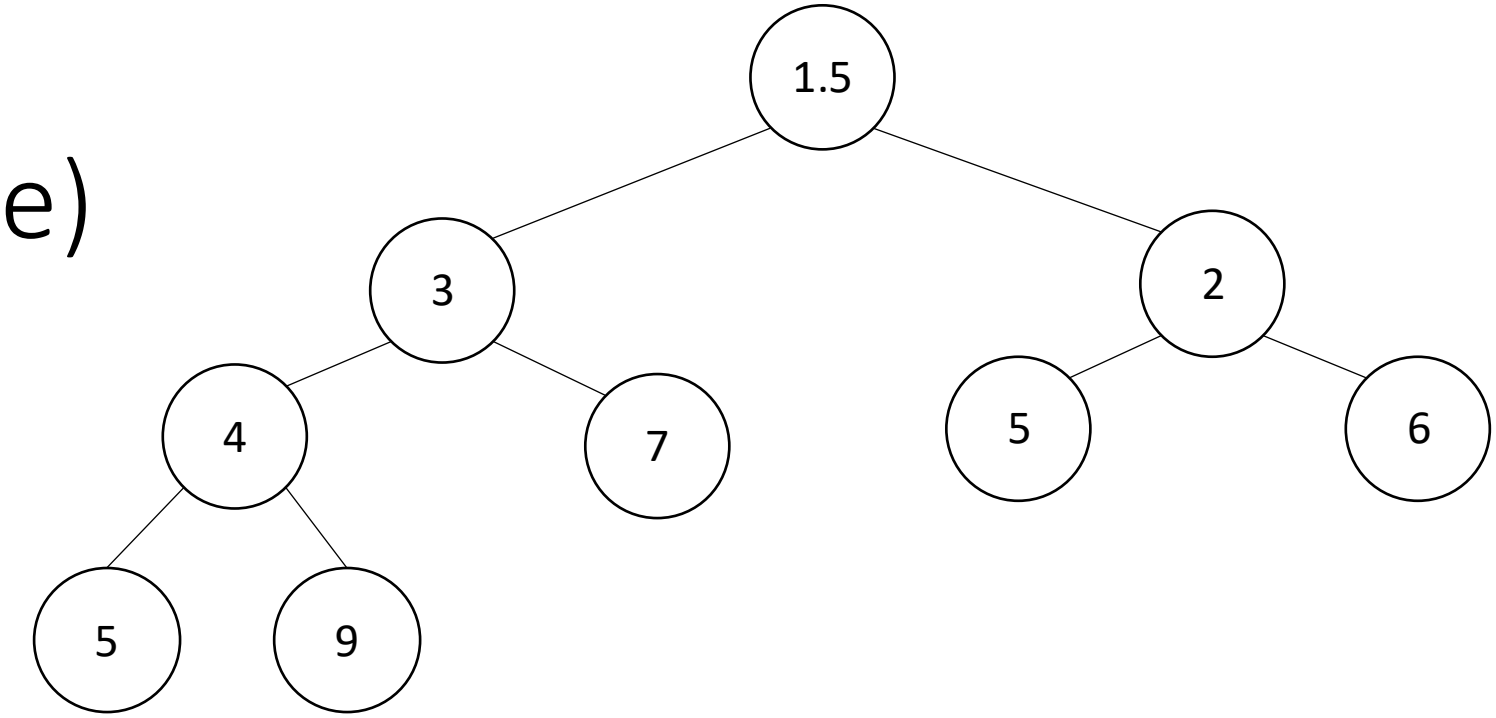
while(curr > curr.left OR curr > curr.right):

swap curr with its smallest child

Percolate Down

return min

Heap extract (Done)



extract():

min = root

curr = bottom-right item

move curr to the root

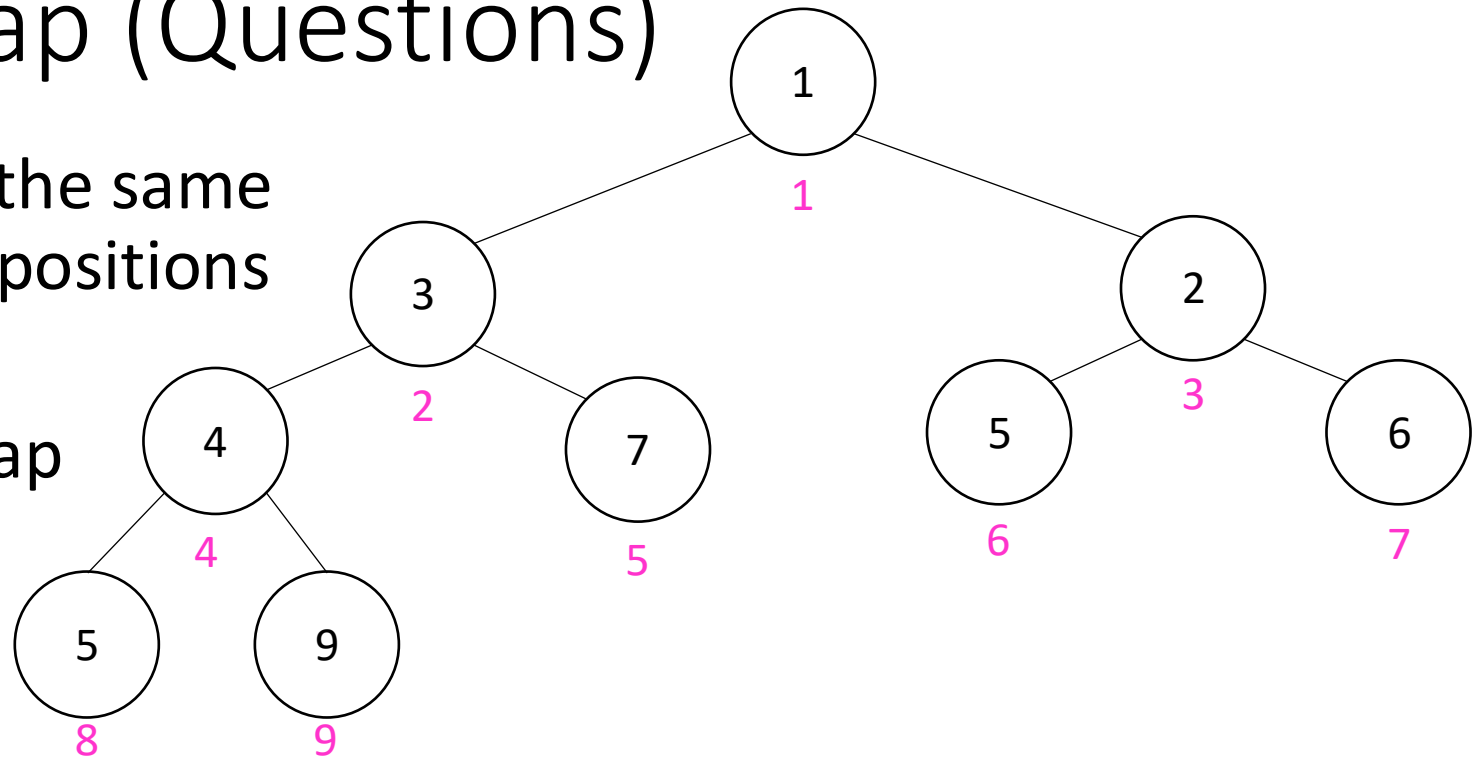
while(curr > curr.left OR curr > curr.right):

swap curr with its smallest child

return min

Representing a Heap (Questions)

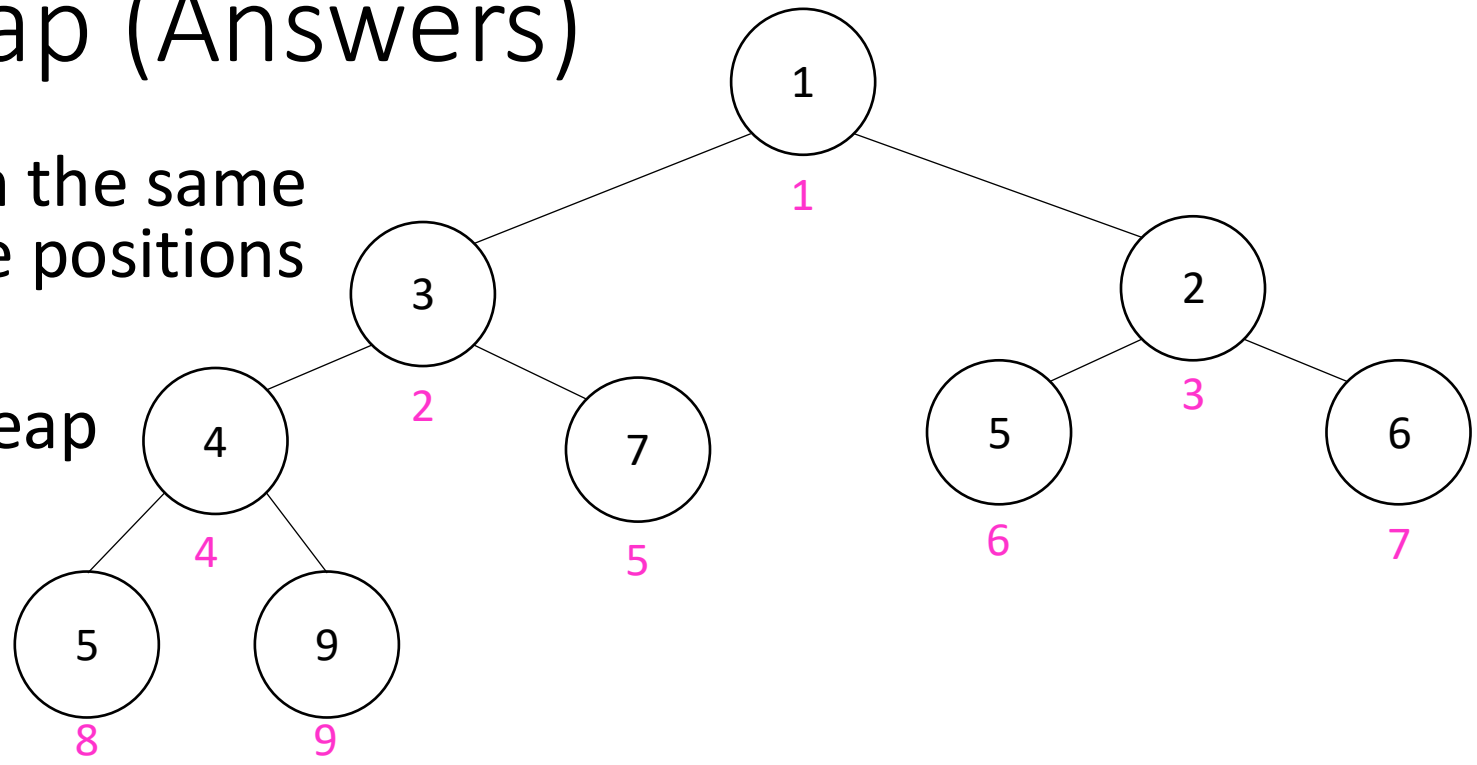
- Every complete binary tree with the same number of nodes uses the same positions and edges
- Use an array to represent the heap
- Index of root:
- Parent of node i:
- Left child of node i:
- Right child of node i:
- Location of the leaves:



	1	3	2	4	7	5	6	5	9
0	1	2	3	4	5	6	7	8	9

Representing a Heap (Answers)

- Every complete binary tree with the same number of nodes uses the same positions and edges
- Use an array to represent the heap
- Index of root: 1
- Parent of node i: $\left\lfloor \frac{i}{2} \right\rfloor$
- Left child of node i: $2i$
- Right child of node i: $2i + 1$
- Location of the leaves: $\left\lfloor \frac{n}{2} \right\rfloor + 1$ to n

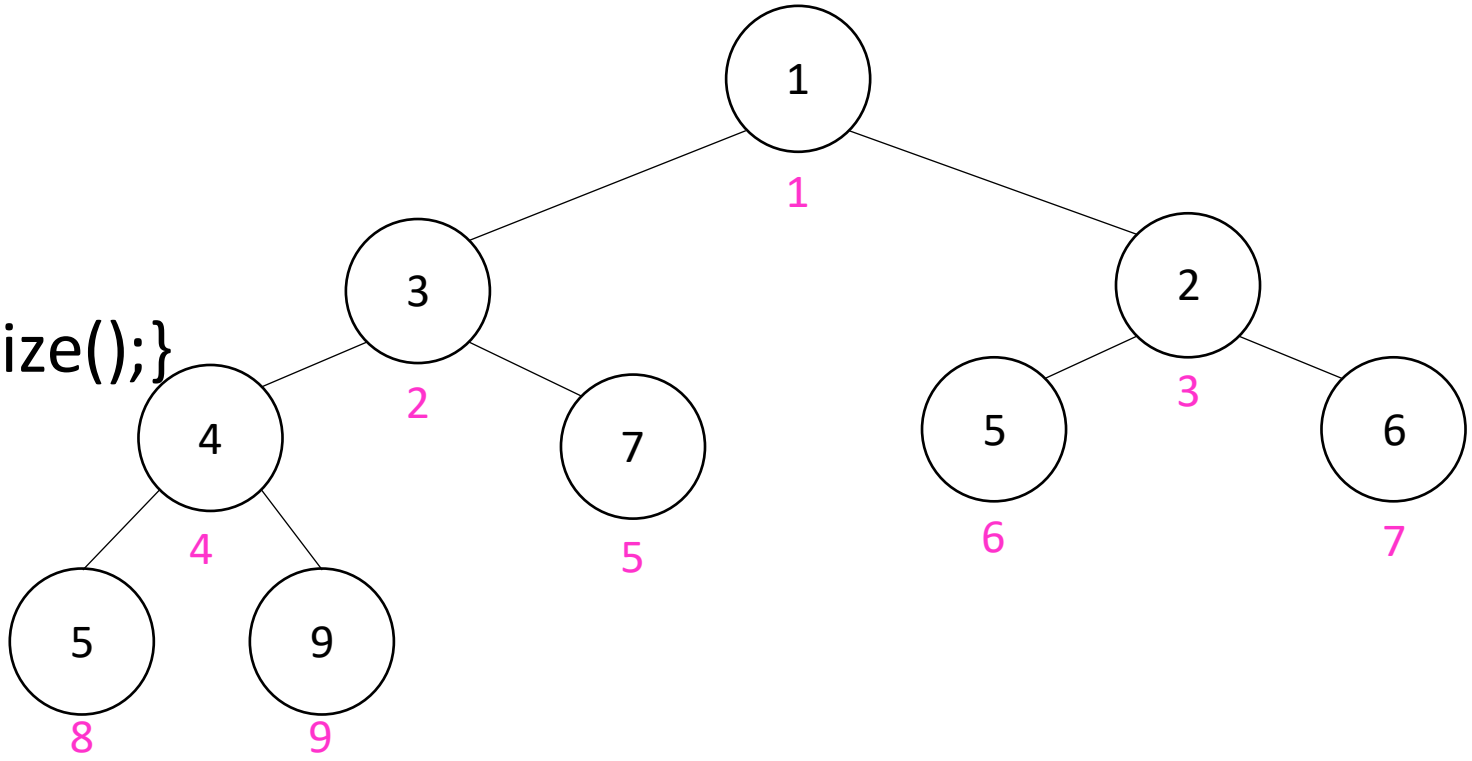


	1	3	2	4	7	5	6	5	9
0	1	2	3	4	5	6	7	8	9

Insert Pseudocode

For simplicity, assume value is the same as priority

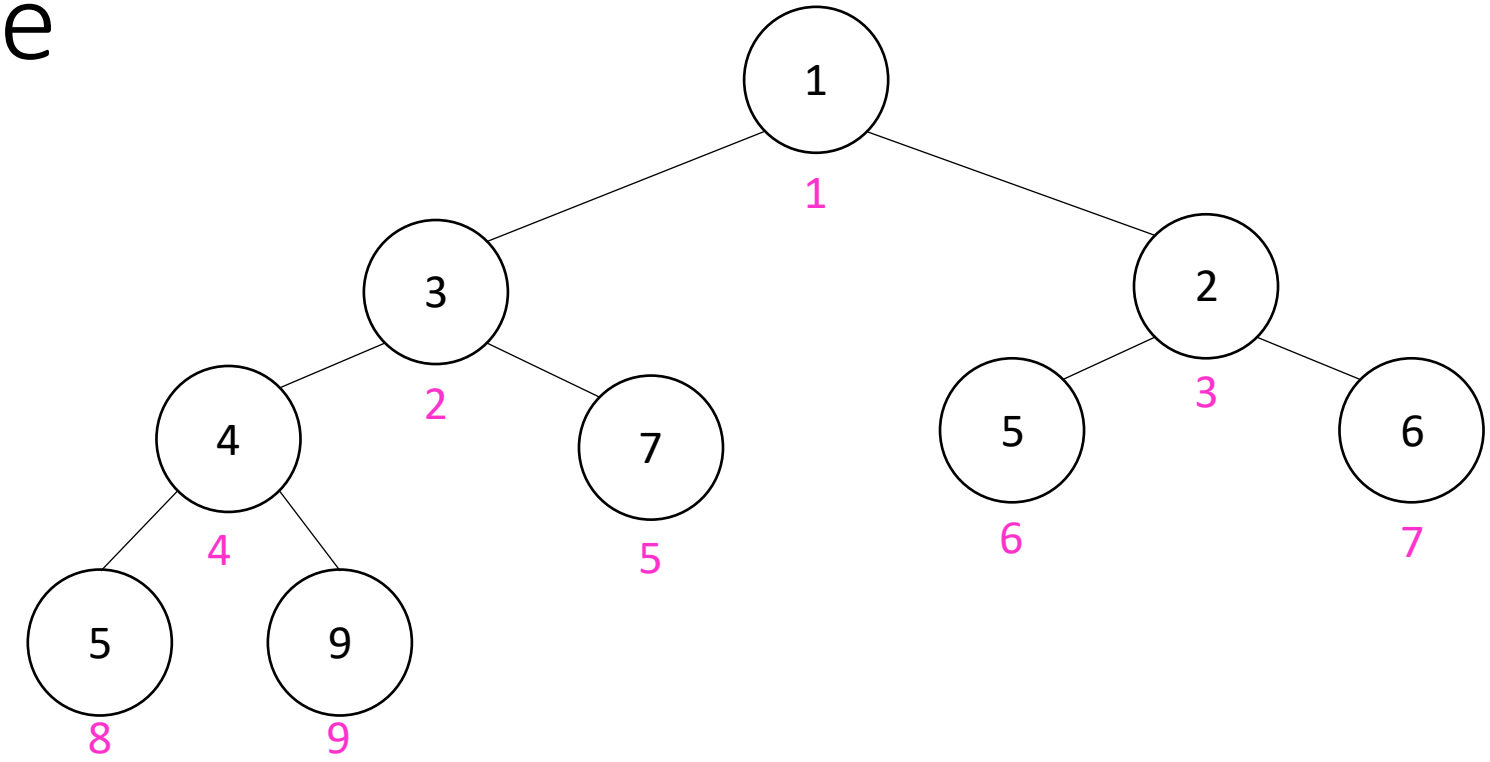
```
insert(item){
    if(size == arr.length – 1){resize();}
    size++;
    arr[size] = item;
    percolateUp(size)
}
```

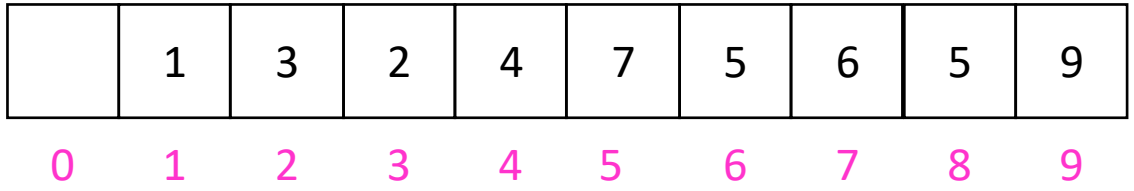


	1	3	2	4	7	5	6	5	9
0	1	2	3	4	5	6	7	8	9

extract Pseudocode

```
extract(){
    theMin = arr[1];
    arr[1] = arr[size];
    size--;
    percolateDown(1);
    return theMin;
}
```





Other Operations?

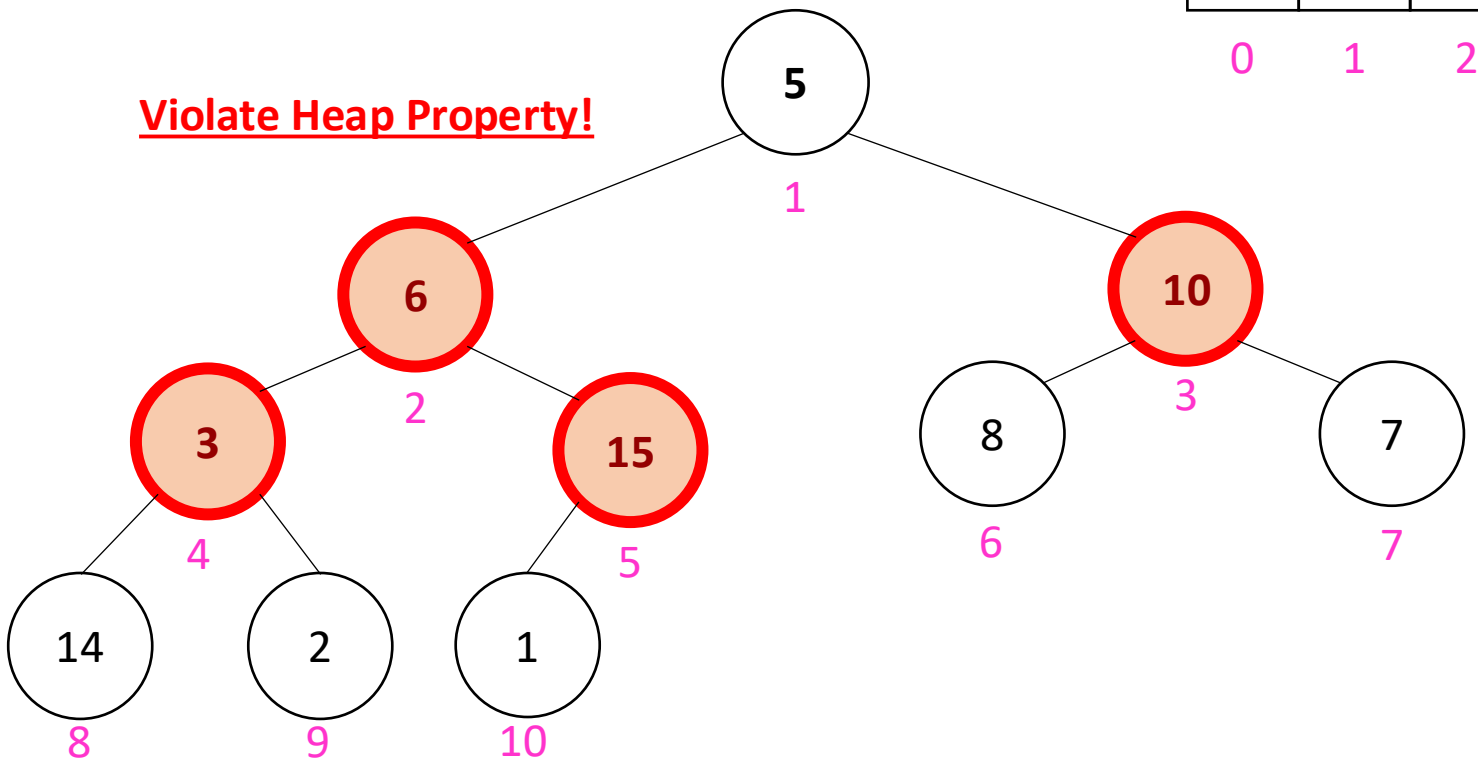
- Update Key
  - Increase Key
    - Given the index of an item in the PQ, make its priority value larger
      - Min Heap: Then percolate down
      - Max Heap: Then percolate up
  - Decrease Key
    - Given the index of an item in the PQ, make its priority value smaller
      - Min Heap: Then percolate up
      - Max Heap: Then percolate down
- Remove
  - Given the item at the given index from the PQ

Building a Heap From "Scratch"

- Suppose we had n items and wanted to "heapify" them

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

Violate Heap Property!



Two ways for "fix" the heap:

1) Percolate Up
2) Percolate Down

Floyd's buildHeap method

- Working towards the root, one row at a time, percolate down

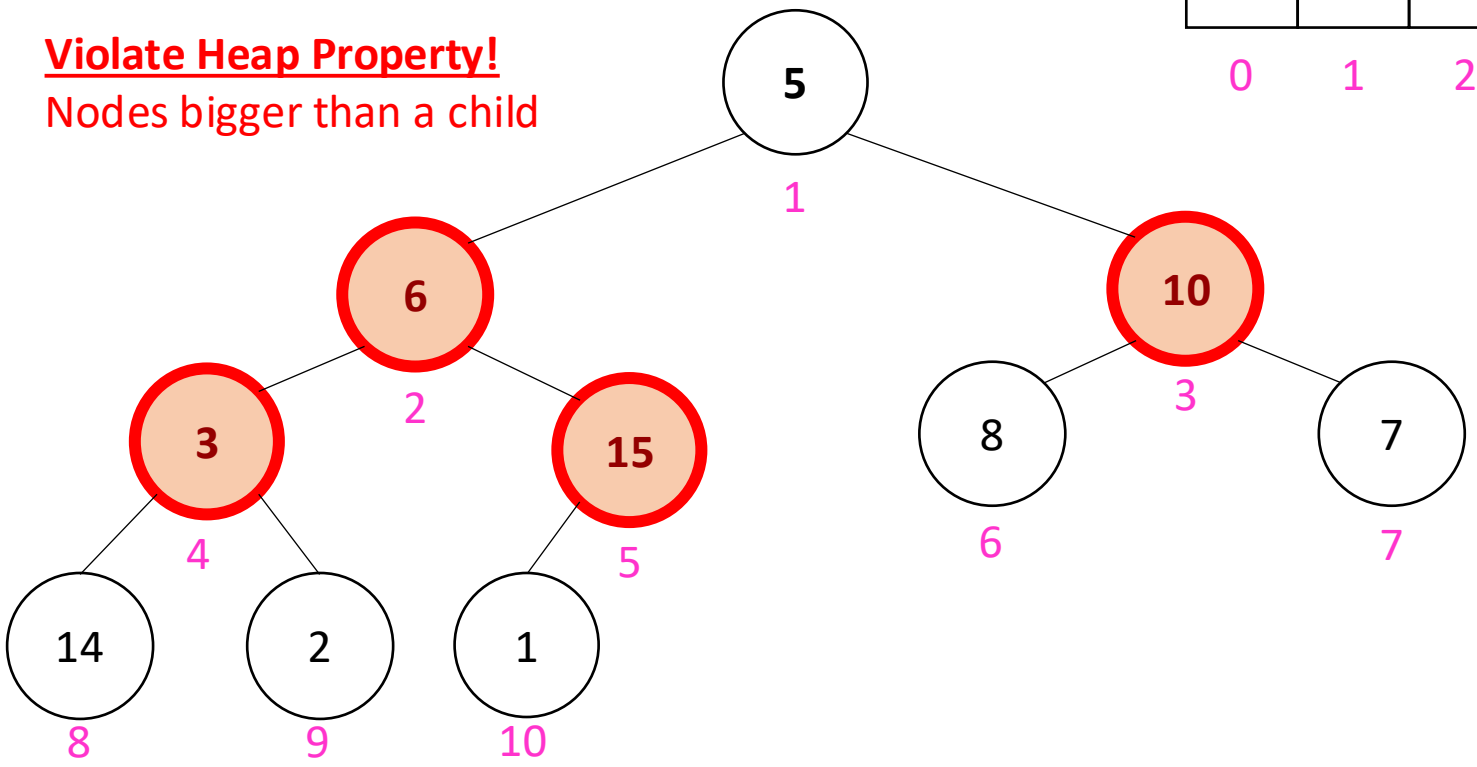
```
buildHeap(){
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```

Floyd's buildHeap method (Percolate 15)

- Suppose we had n items and wanted to "heapify" them

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

Violate Heap Property!
Nodes bigger than a child

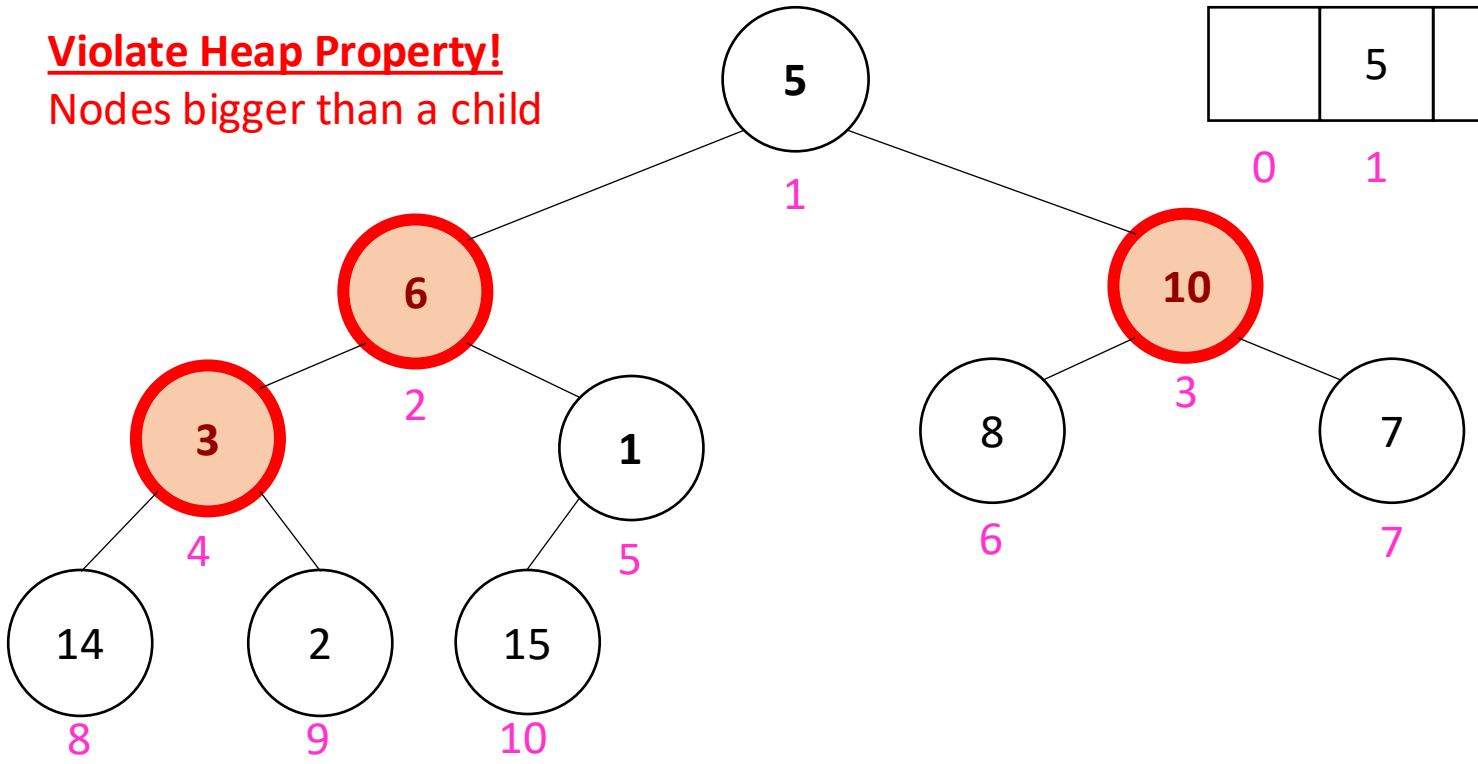


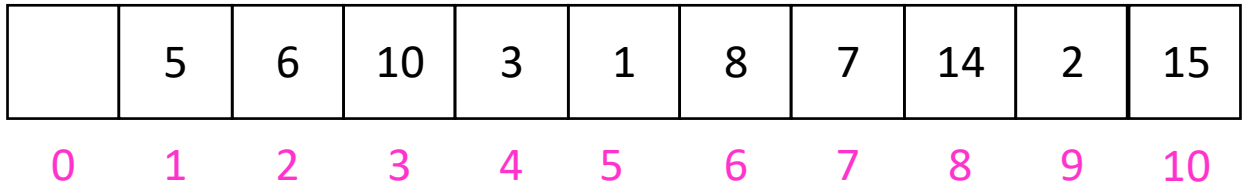

```
buildHeap(){
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```

Floyd's buildHeap method (Percolate 3)

- Suppose we had n items and wanted to "heapify" them

Violate Heap Property!
Nodes bigger than a child

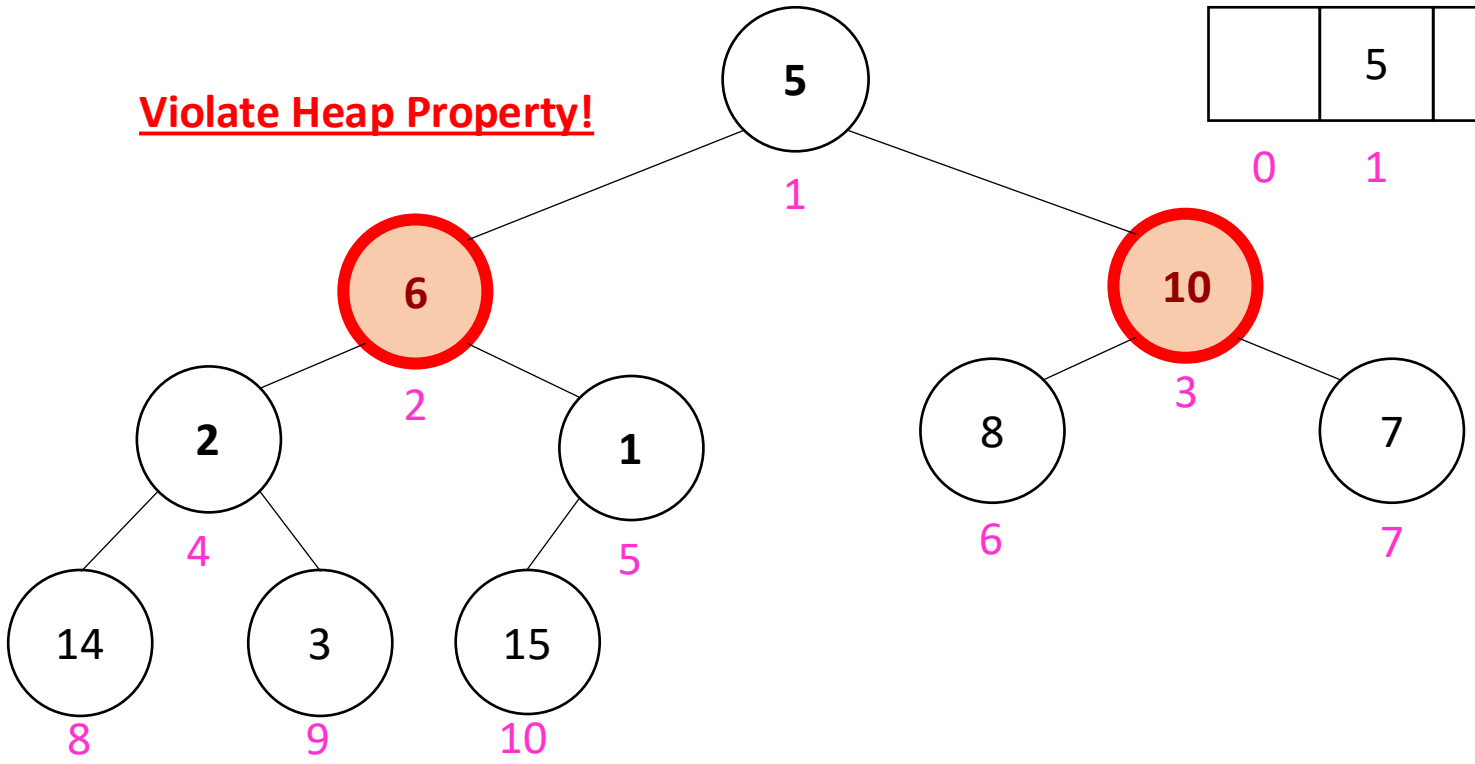





```
buildHeap(){
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```

Floyd's buildHeap method (Percolate 10)

- Suppose we had n items and wanted to "heapify" them

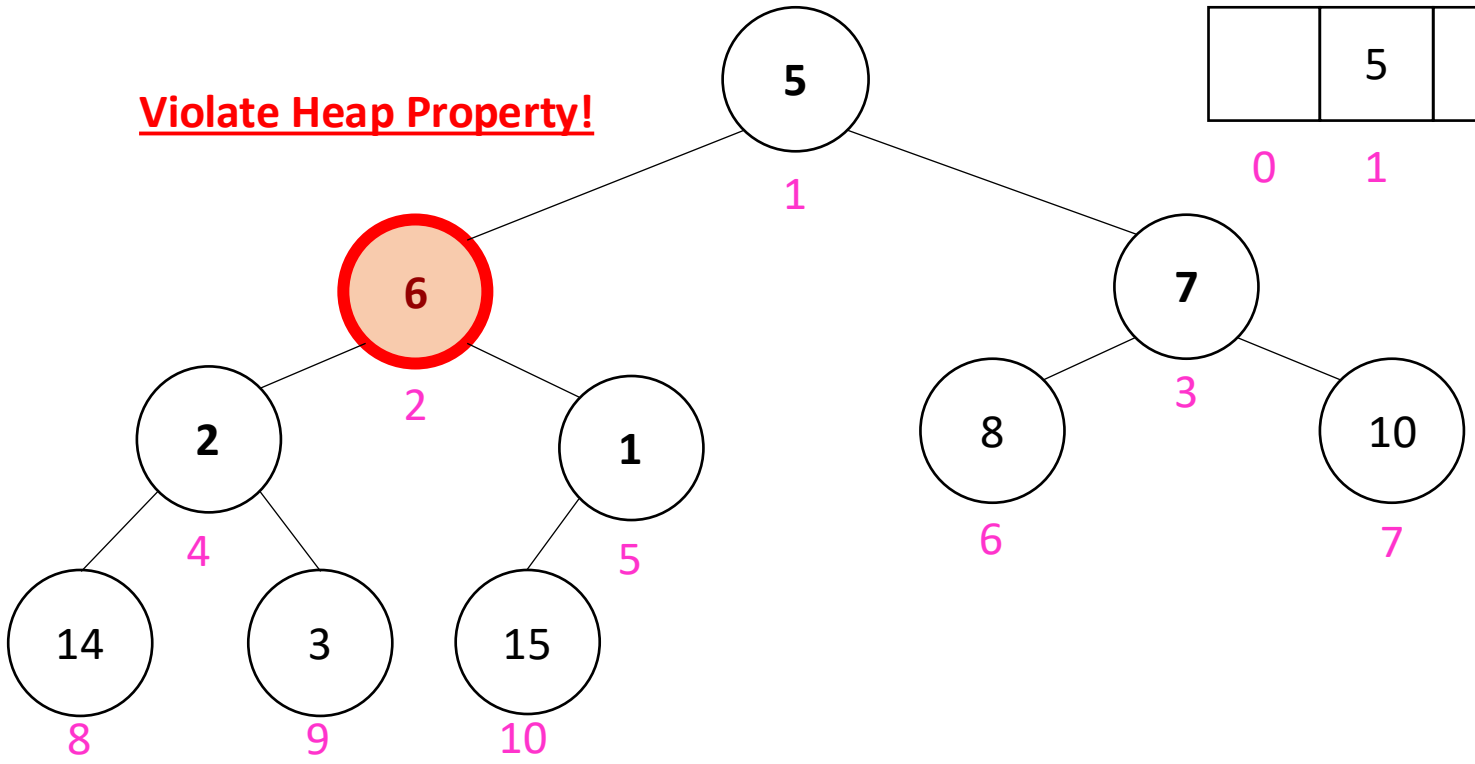


	5	6	10	2	1	8	7	14	3	15
0	1	2	3	4	5	6	7	8	9	10

```
buildHeap(){
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```

Floyd's buildHeap method (Percolate 6)

- Suppose we had n items and wanted to "heapify" them

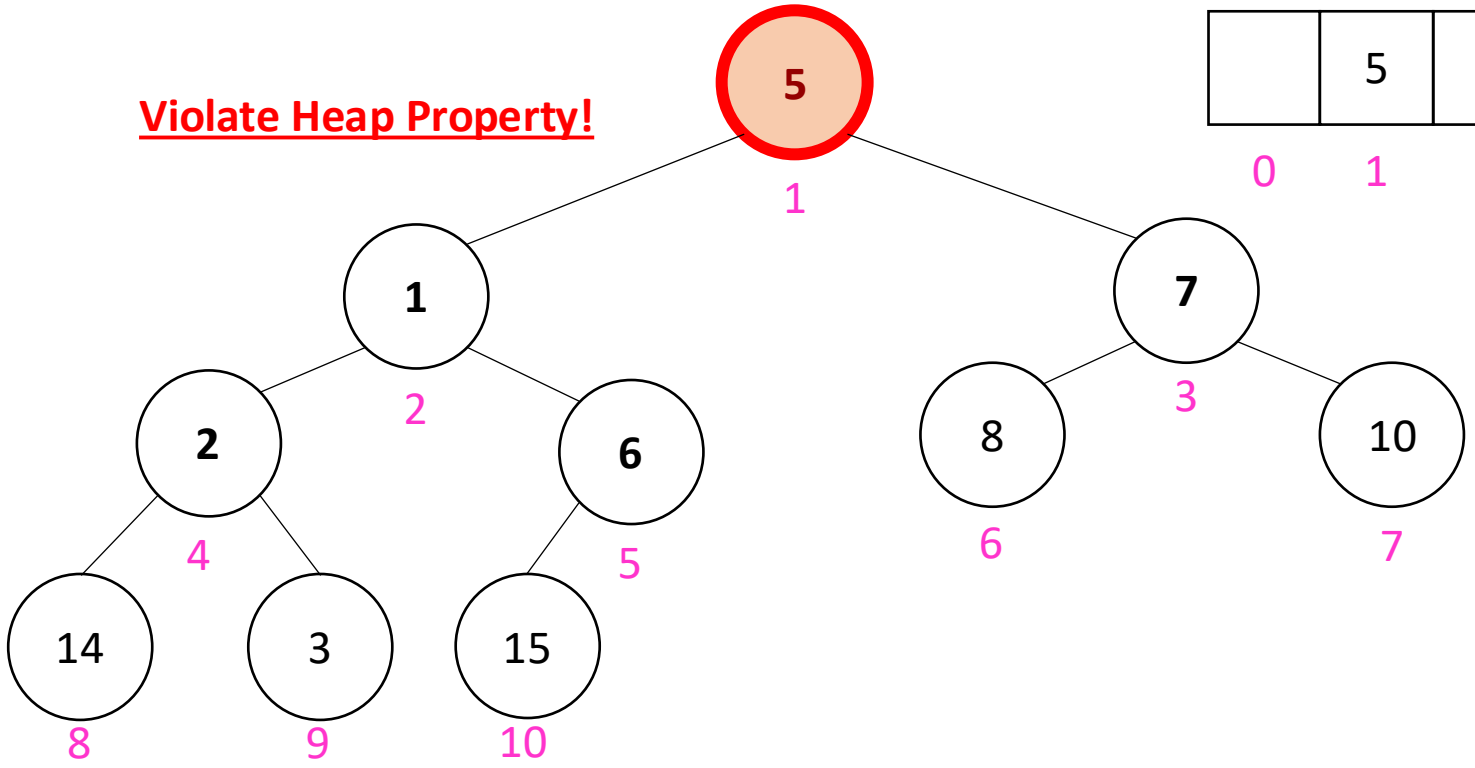


	5	6	7	2	1	8	10	14	3	15
0	1	2	3	4	5	6	7	8	9	10

```
buildHeap(){
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```

Floyd's buildHeap method (Percolate 5)

- Suppose we had n items and wanted to "heapify" them

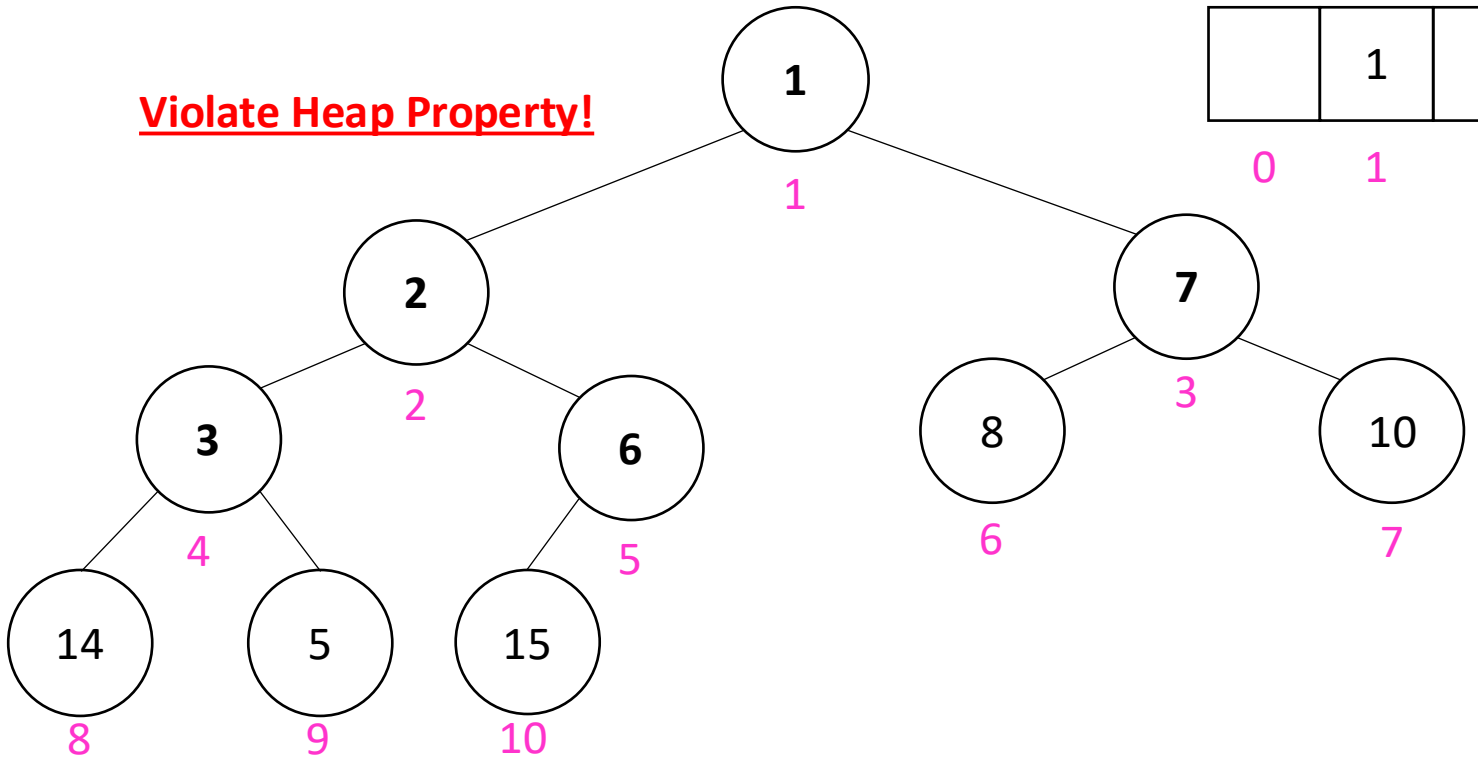


	5	1	7	2	6	8	10	14	3	15
0	1	2	3	4	5	6	7	8	9	10

```
buildHeap(){
   for(int i = size; i>0; i--){
      percolateDown(i);
   }
}
```

Floyd's buildHeap method (Done)

- Suppose we had n items and wanted to "heapify" them



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

```
buildHeap(){
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```

How long did this take?

- Worst case running time of buildHeap:
- No node can percolate down more than the height of its subtree
  - When i is a leaf:
  - When i is second-from-last level:
  - When i is third-from-last level:
- Overall Running time:

```
buildHeap(){
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```

How long did this take?

- Worst case running time of buildHeap: $n \sum_{i=1}^{\log_2(n)} \frac{i}{2^i}$
- No node can percolate down more than the height of its subtree
  - When i is a leaf ($n/2$ of these): 0 comparisons
  - When i is second-from-last level ($n/4$ of these): ≤ 2 comparisons
  - When i is third-from-last level ($n/8$ of these): ≤ 4 comparisons
  - …
- Overall Running time:

```
buildHeap(){
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```

A Serious Series Calculation

- **Lemma:** $E \coloneqq n \sum_{i=1}^{\log_2(n)} \frac{i}{2^i} \in \Theta(n)$
- **Proof:** First term is $n/2$, so $E \in \Omega(n)$ is immediate.
- Next, observe $\sum_{i=1}^{\log_2(n)} \frac{i}{2^i} \leq \sum_{i=1}^{\infty} \frac{i}{2^i}$
- Recall for $|x| < 1$ we have $\sum_{i=1}^{\infty} x^i = \frac{1}{1-x}$
- Differentiate: $\sum_{i=1}^{\infty} i x^{i-1} = \frac{1}{(1-x)^2}$
- Multiply both sides by x: $\sum_{i=1}^{\infty} i x^i = \frac{x}{(1-x)^2}$
- Plug in $x = 1/2$: $\sum_{i=1}^{\infty} i/2^i = \frac{1/2}{(1/2)^2} = 2.$