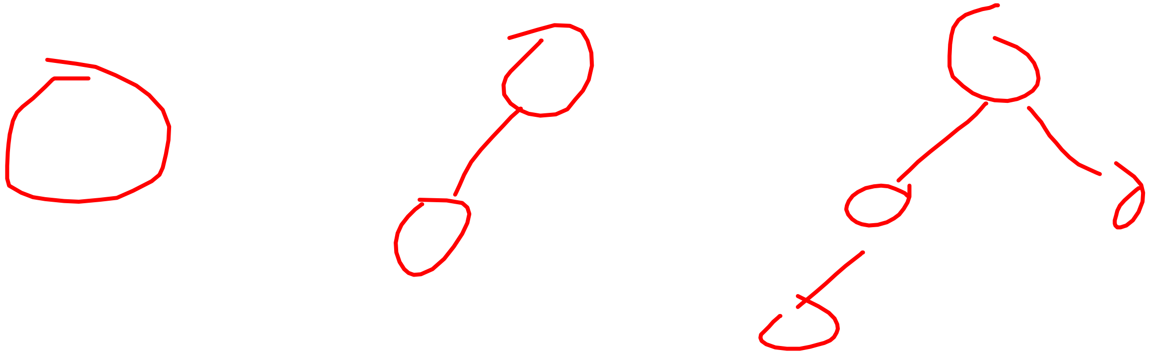
CSE 332 Autumn 2023
Lecture 6: Priority Queues 2

Nathan Brunelle

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

Warm Up!

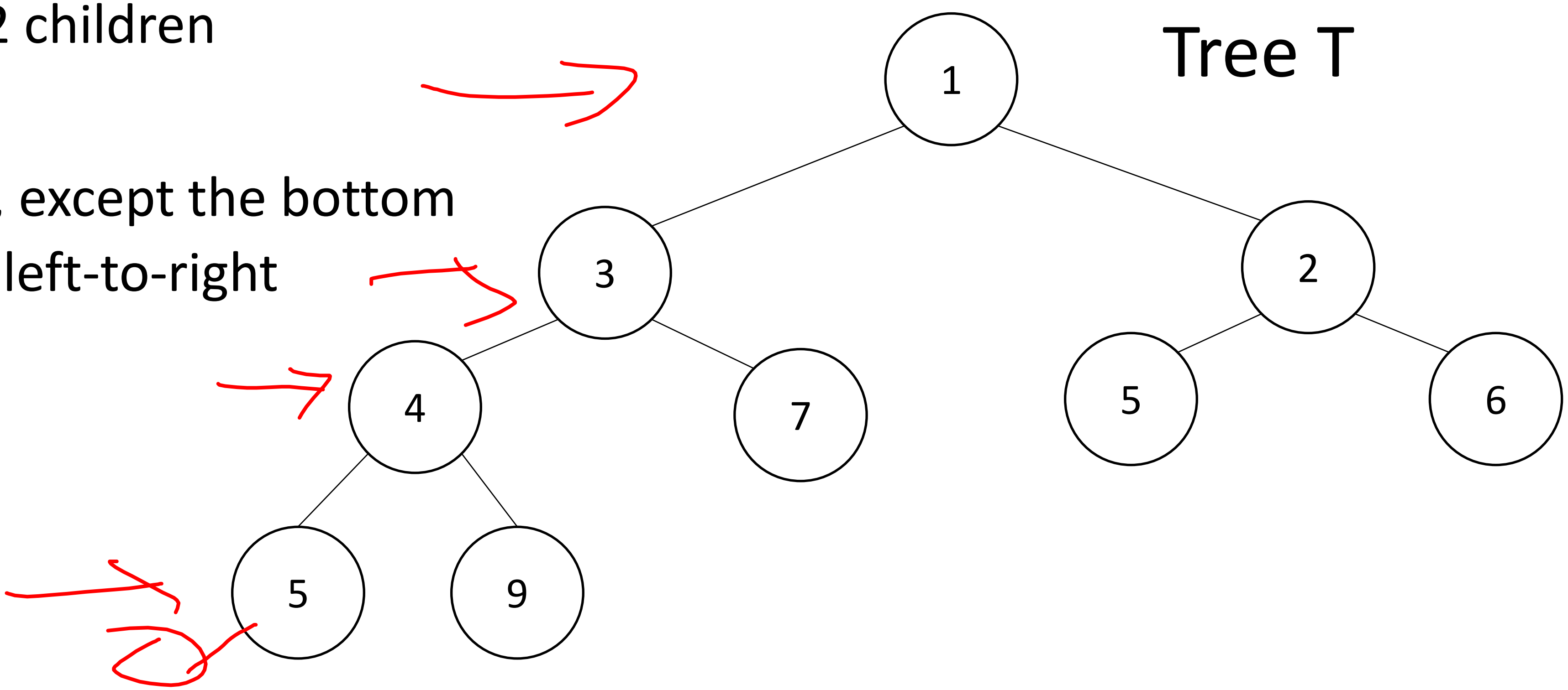


- What is the maximum number of total nodes in a binary tree of height h?
  - Height: The number of **edges** in the path from root to the deepest leaf
  - $2^{h+1} - 1$
  - 3, 7, 15, 31
  - $\Theta(2^h)$
- If I have n nodes in a binary tree, what is the its minimum height?
  - The height of the tree must be high enough that n nodes is possible
  - $2^{h+1} - 1 \geq n$
  - $2^{h+1} \geq n + 1$
  - $\log 2^{h+1} \geq \log(n + 1)$
  - $h \geq \log(n + 1) - 1$
  - $\Theta(\log n)$

Trees for Heaps

~log n

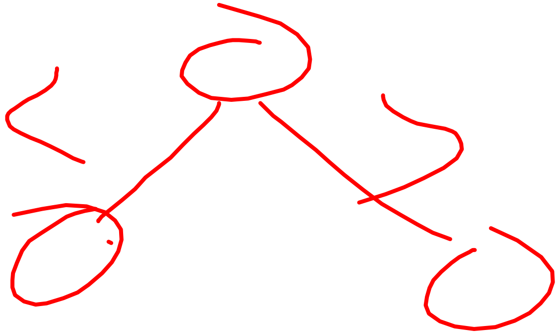
- Binary Trees:
  - The branching factor is 2
  - Every node has ≤ 2 children
- Complete Tree:
  - All "layers" are full, except the bottom
  - Bottom layer filled left-to-right



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
  - deleteMin
    - 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)

Thinking through implementations



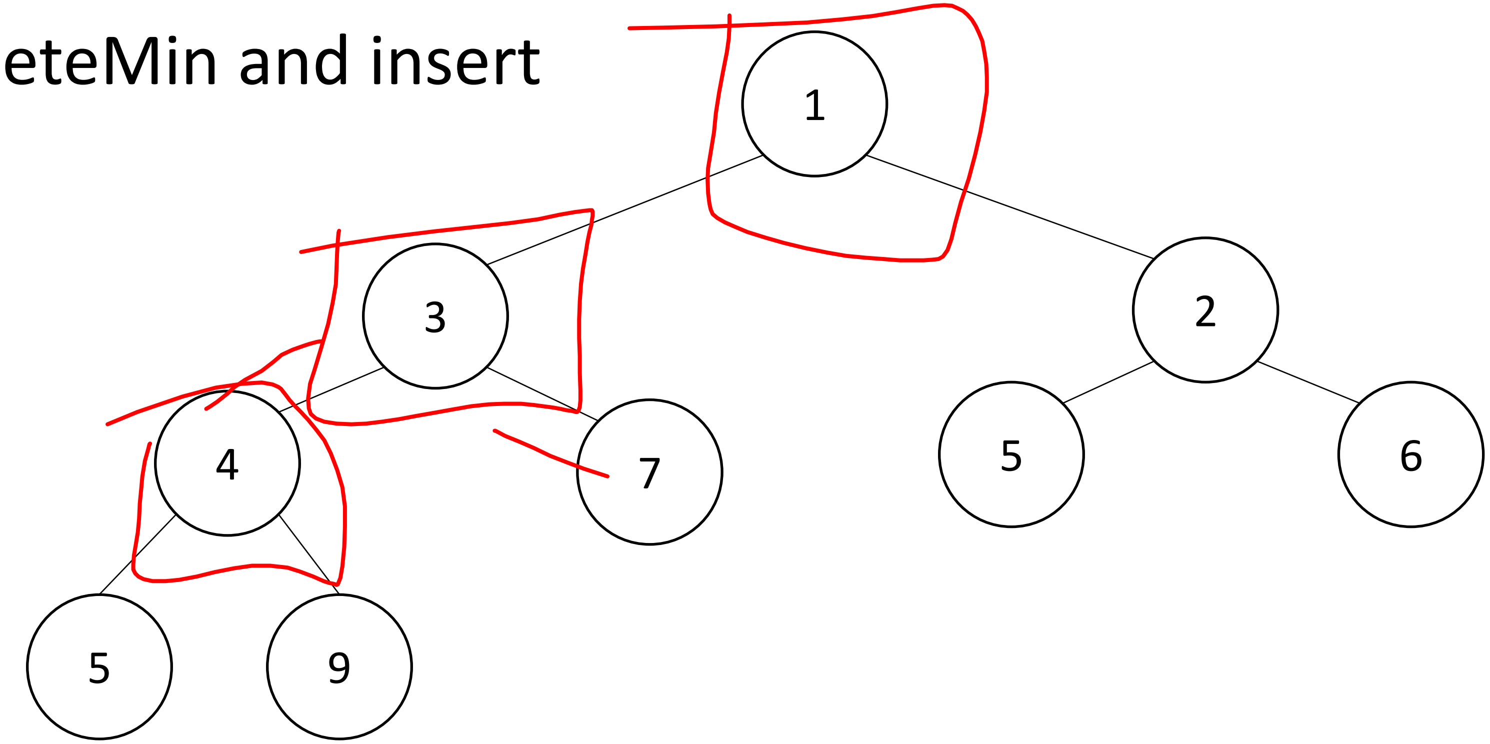
Data Structure	Worst case time to insert	Worst case time to deleteMin
Unsorted Array	$\Theta(1)$	$\Theta(n)$
Unsorted Linked List	$\Theta(1)$	$\Theta(n)$
Sorted Circular Array	$\Theta(n)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(1)$
Binary Search Tree	$\Theta(n)$	$\Theta(1)$
Binary Heap	$\Theta(\log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

Note: Assume we know the maximum size of the PQ in advance

Heap – Priority Queue Data Structure

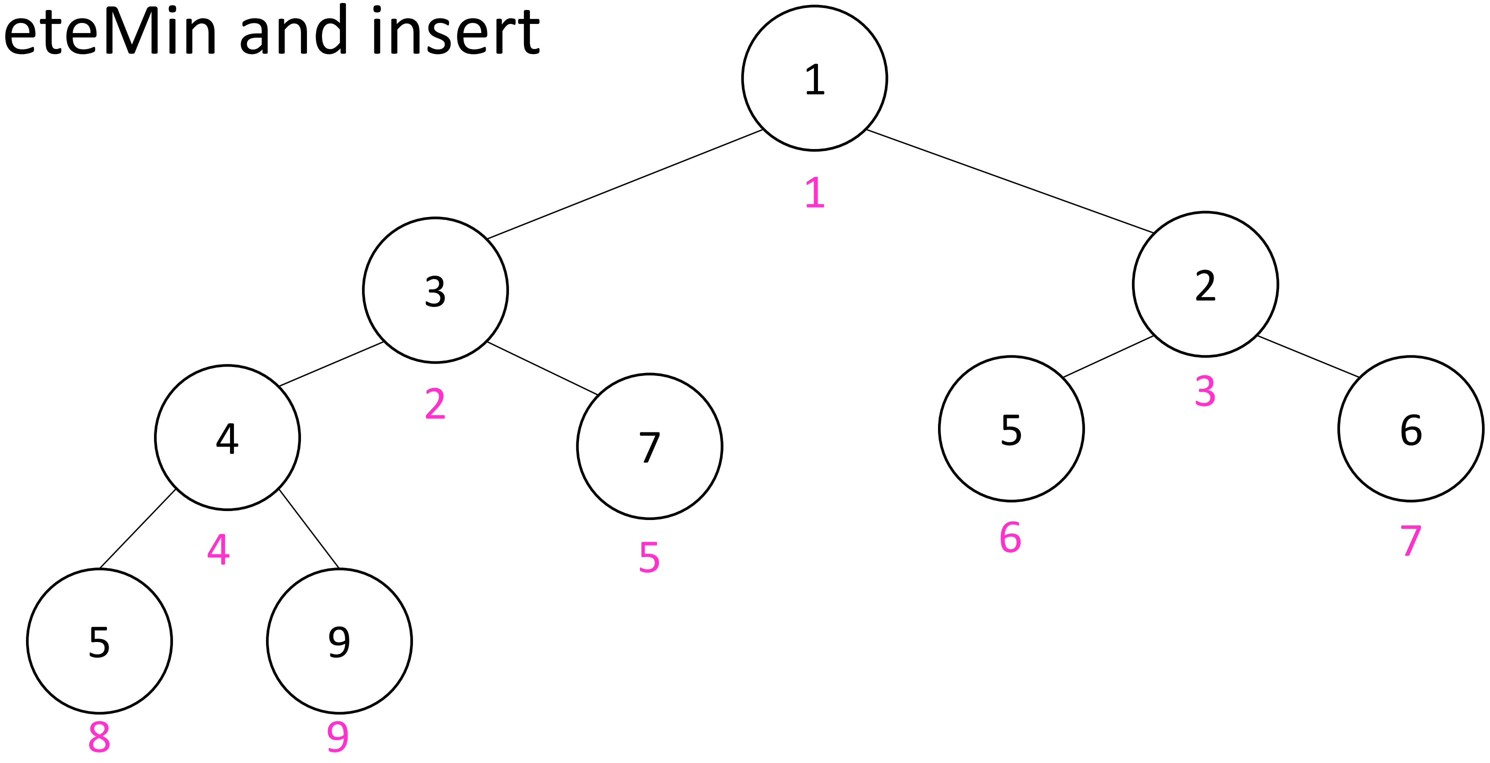
- Idea: We need to keep some ordering, but it doesn't need to be perfectly sorted
- $\Theta(\log n)$ worst case for deleteMin and insert

Heap
Property



Heap – Priority Queue Data Structure

- Idea: We need to keep some ordering, but it doesn't need to be perfectly sorted
- $\Theta(\log n)$ worst case for deleteMin and insert



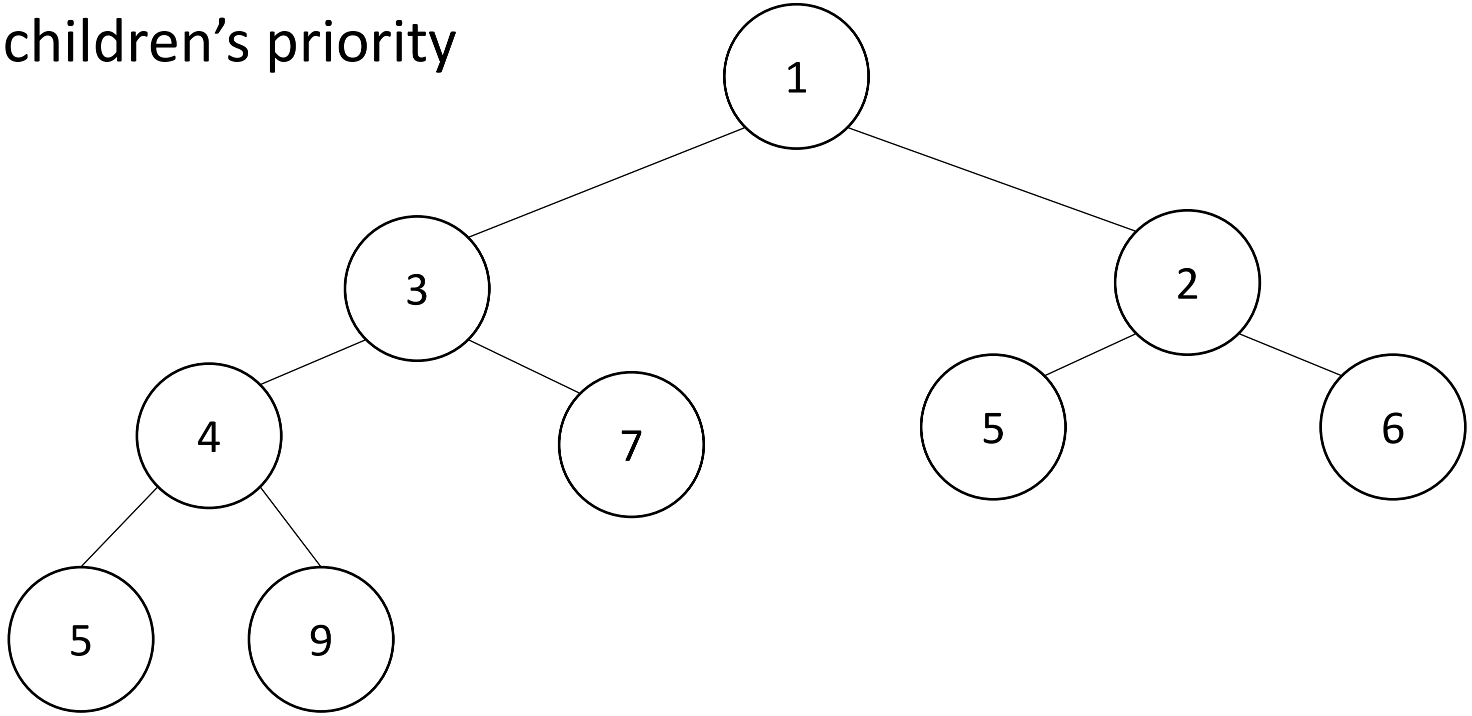
Challenge!

- 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 deleteMin requires just one “trip” from root to leaf

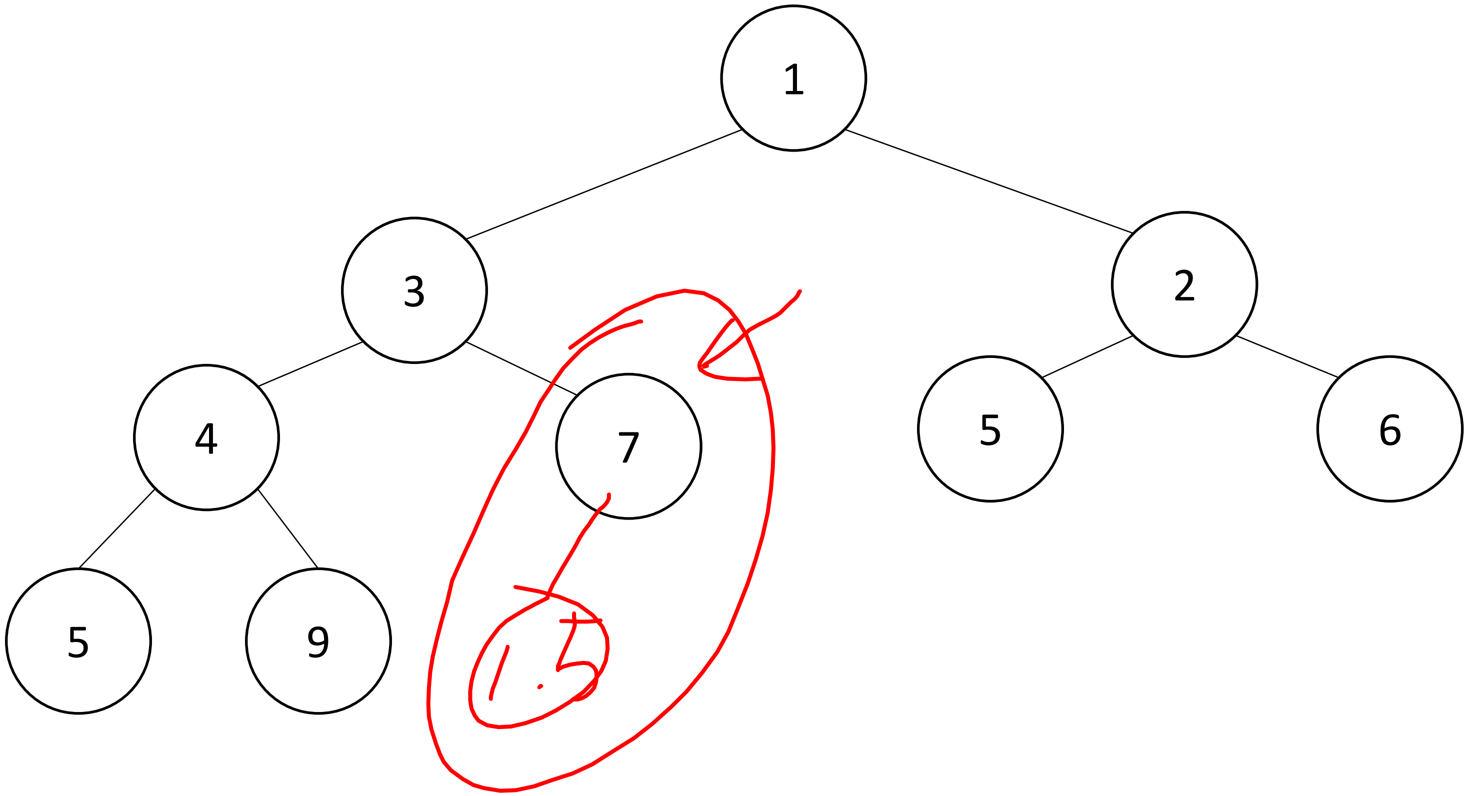
Heap Data Structure

- Keep items in a complete binary tree
- Maintain the “Heap Property” of the tree
  - Every node’s priority is $\leq$ its children’s priority

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

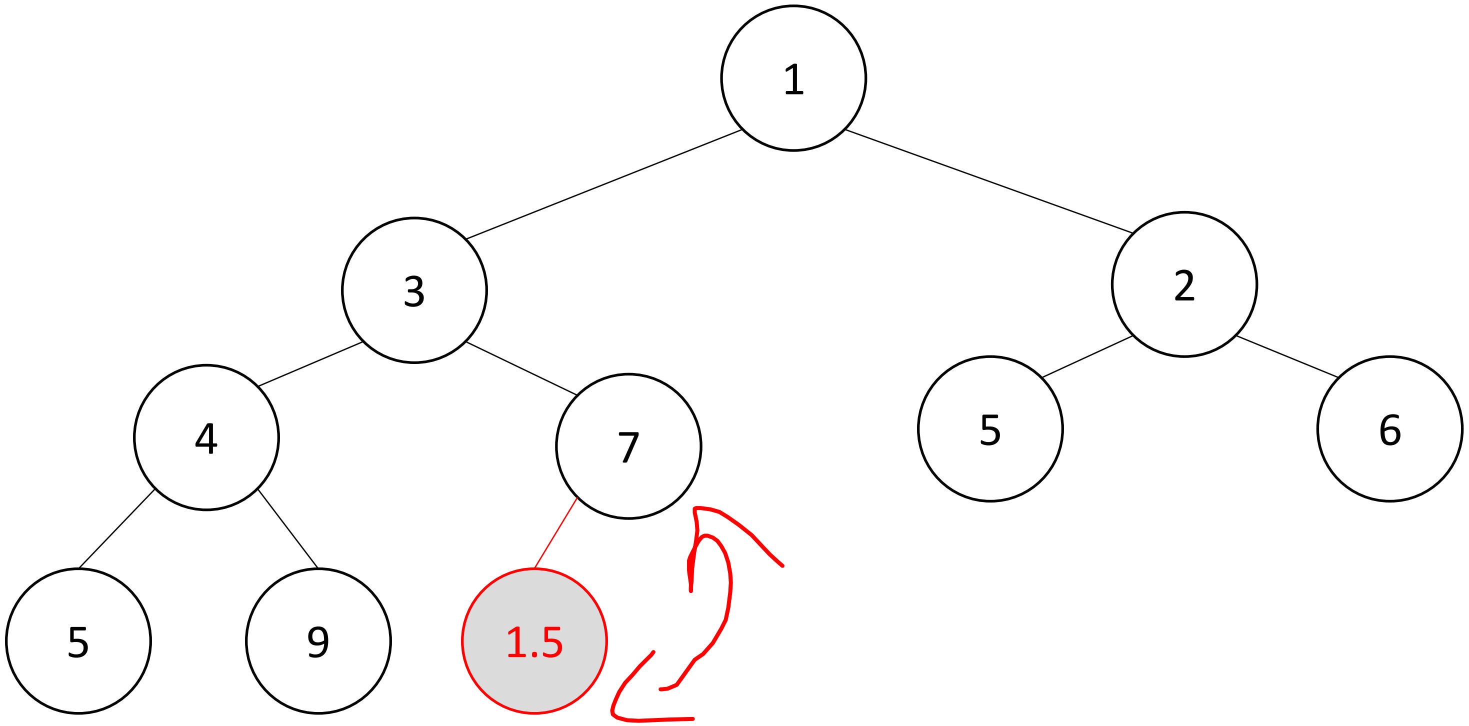


Heap Insert



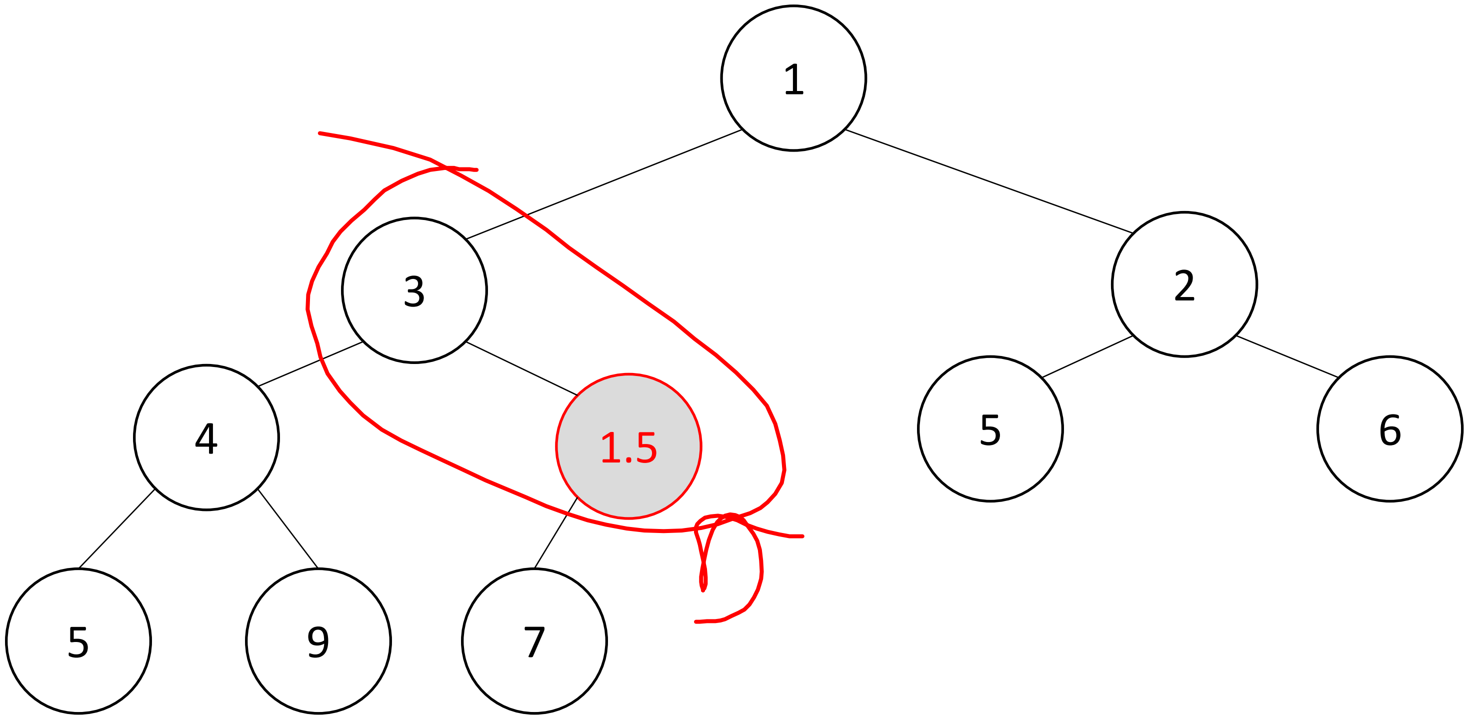

```
insert(item){
    put item in the "next open" spot (keep tree complete)
    while (item.priority < parent(item).priority){
        swap item with parent
    }
}
```

Heap Insert




```
insert(item){
    put item in the "next open" spot (keep tree complete)
    while (item.priority < parent(item).priority){
        swap item with parent
    }
}
```

Heap Insert



insert(item){

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

    while (item.priority < parent(item).priority){

        swap item with parent

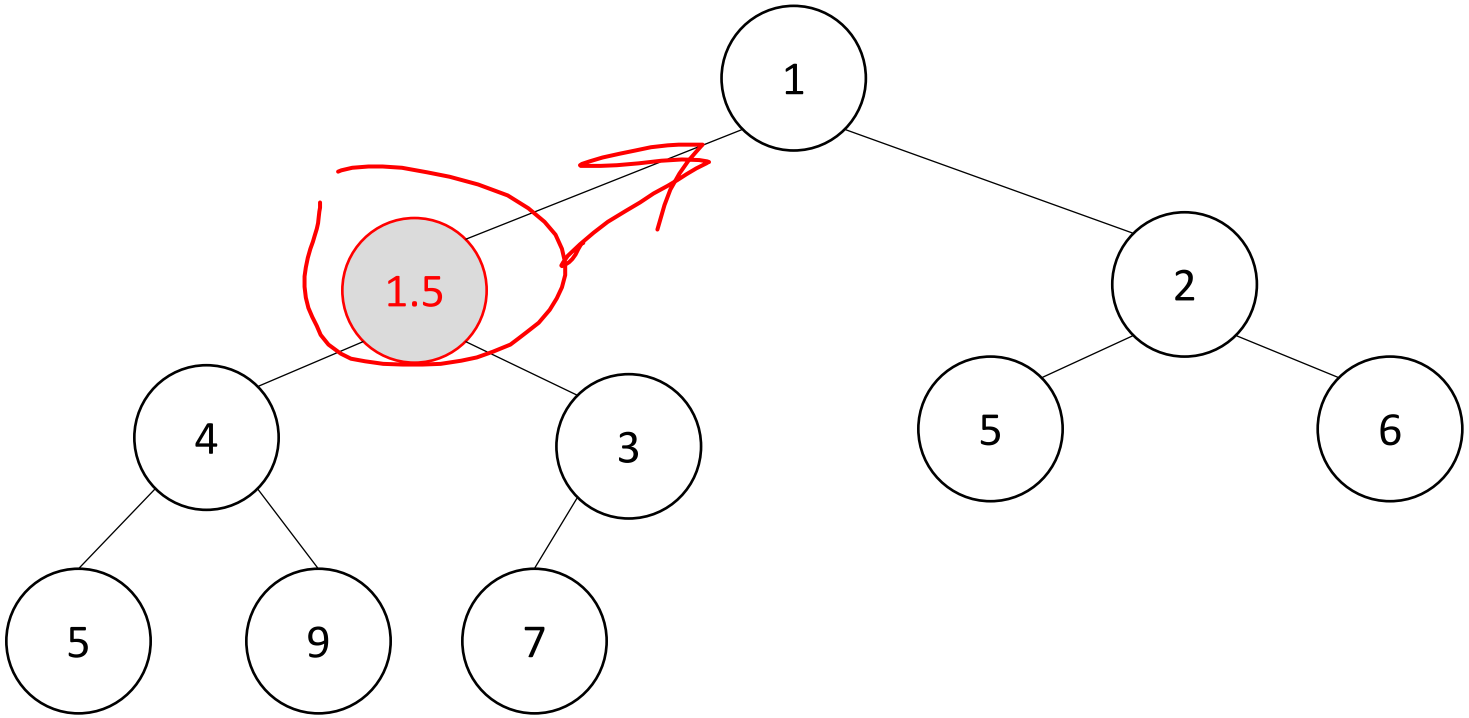
    }

}

Percolate Up

Heap Insert

log n



insert(item){

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

while (item.priority < parent(item).priority){

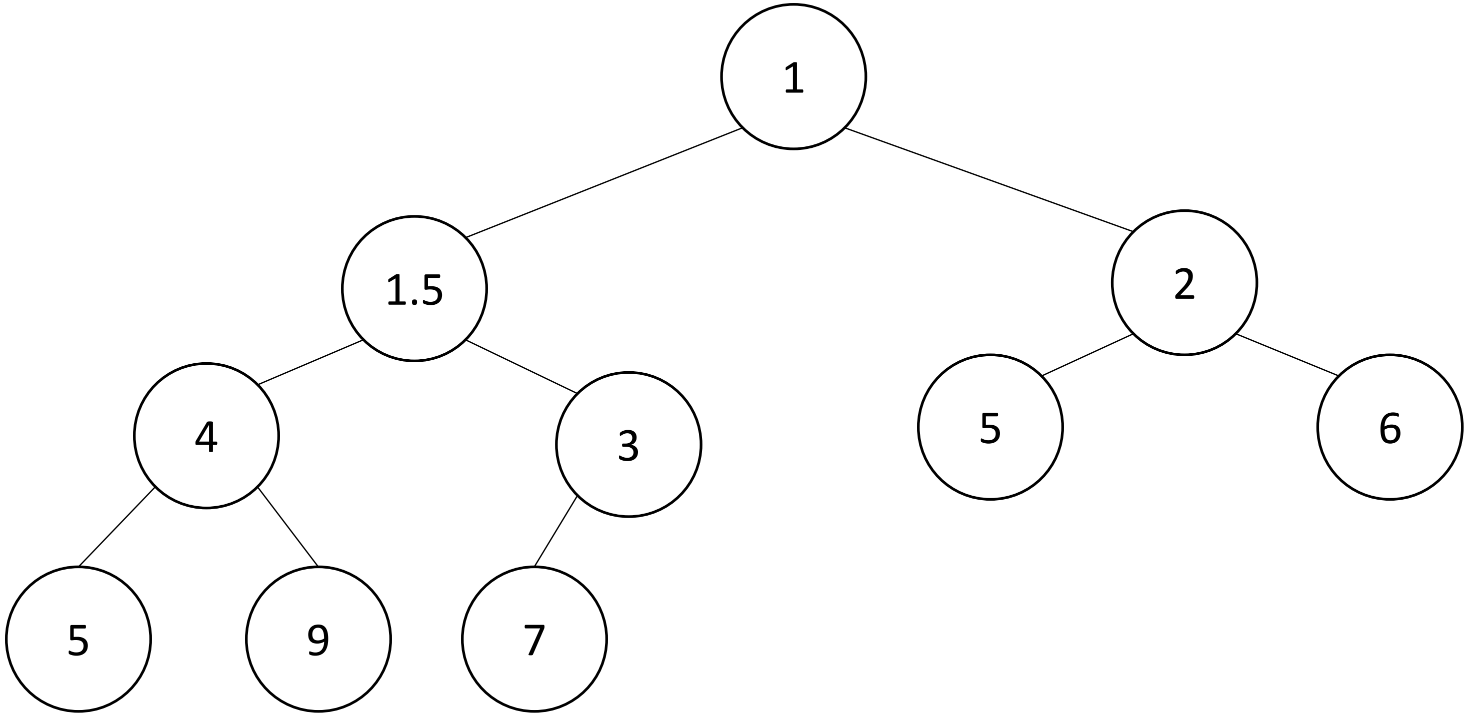
swap item with parent

}

}

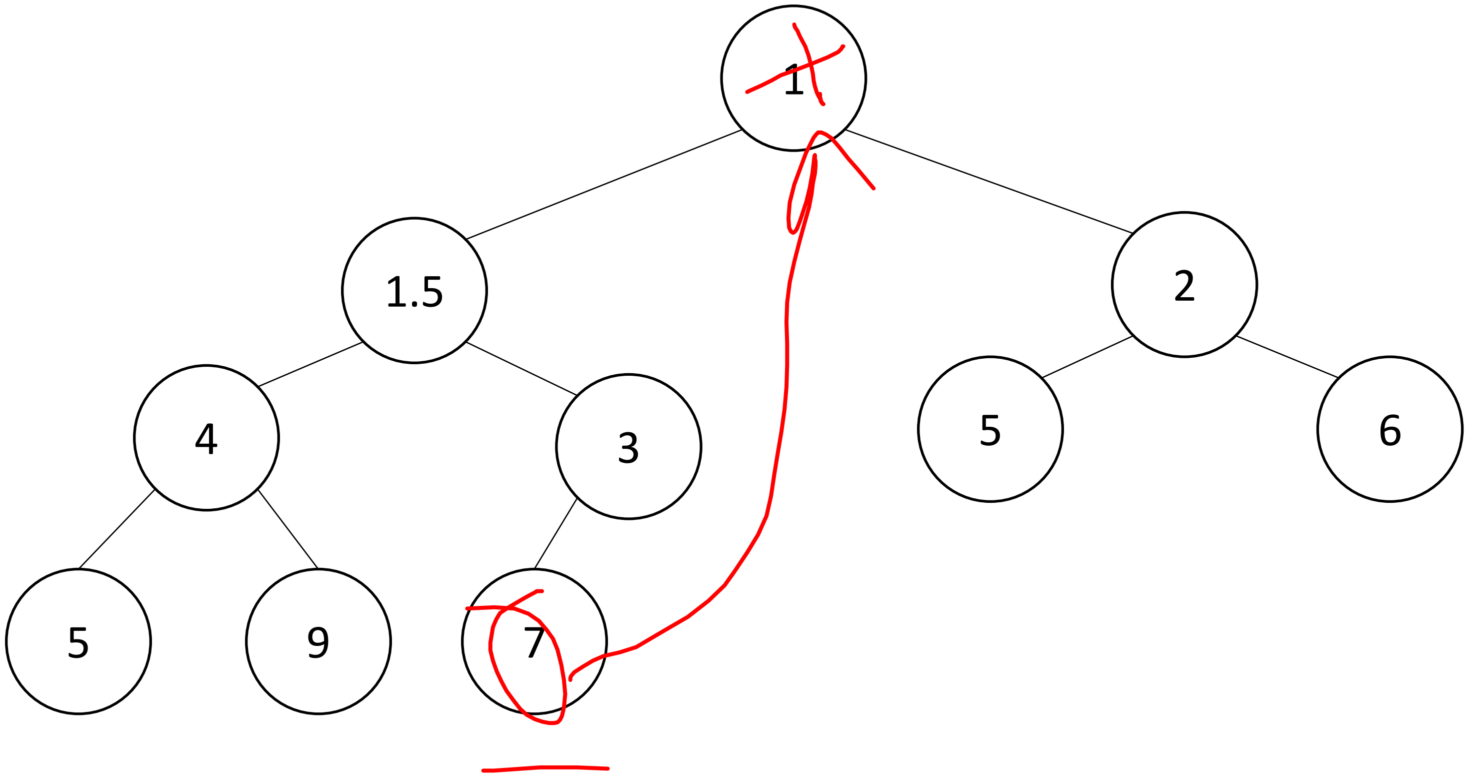
Percolate Up

Heap Insert



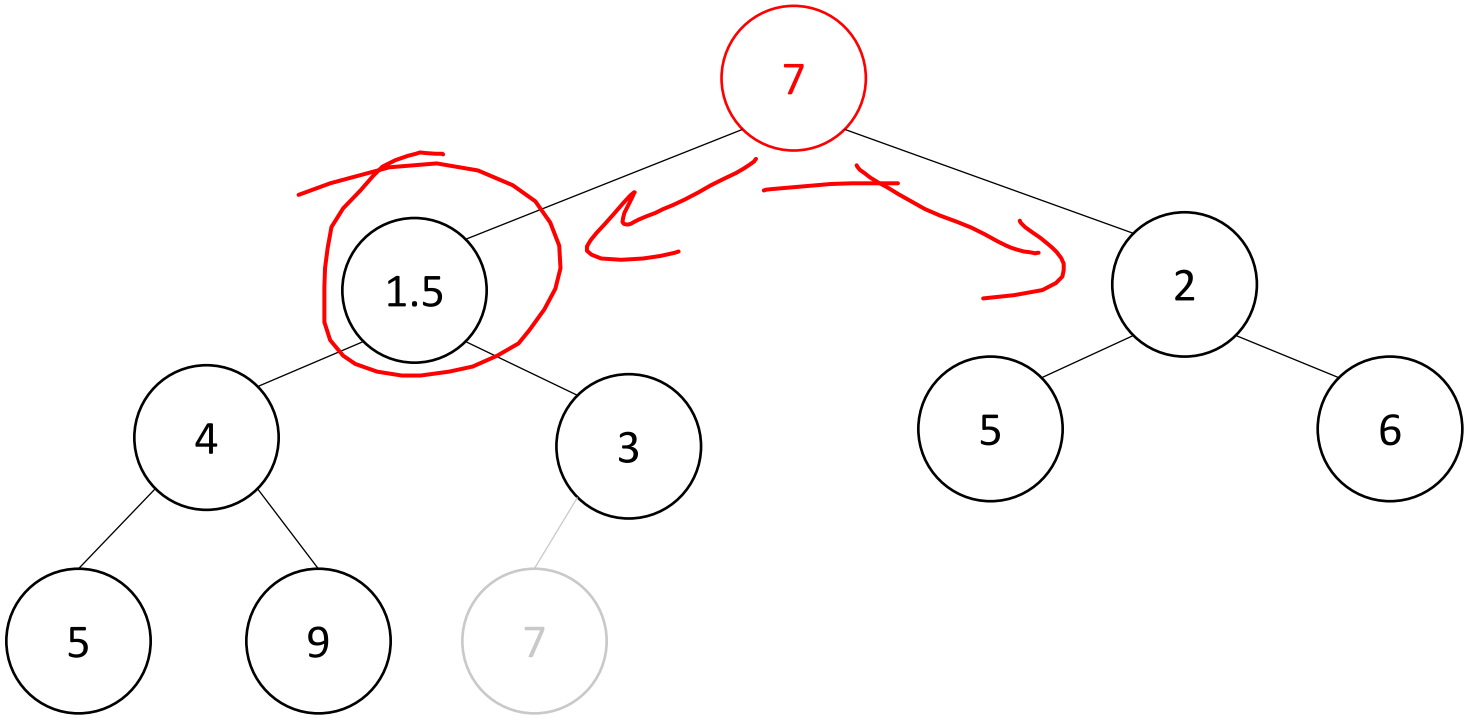

```
insert(item){
    put item in the "next open" spot (keep tree complete)
    while (item.priority < parent(item).priority){
        swap item with parent
    }
}
```

Heap deleteMin



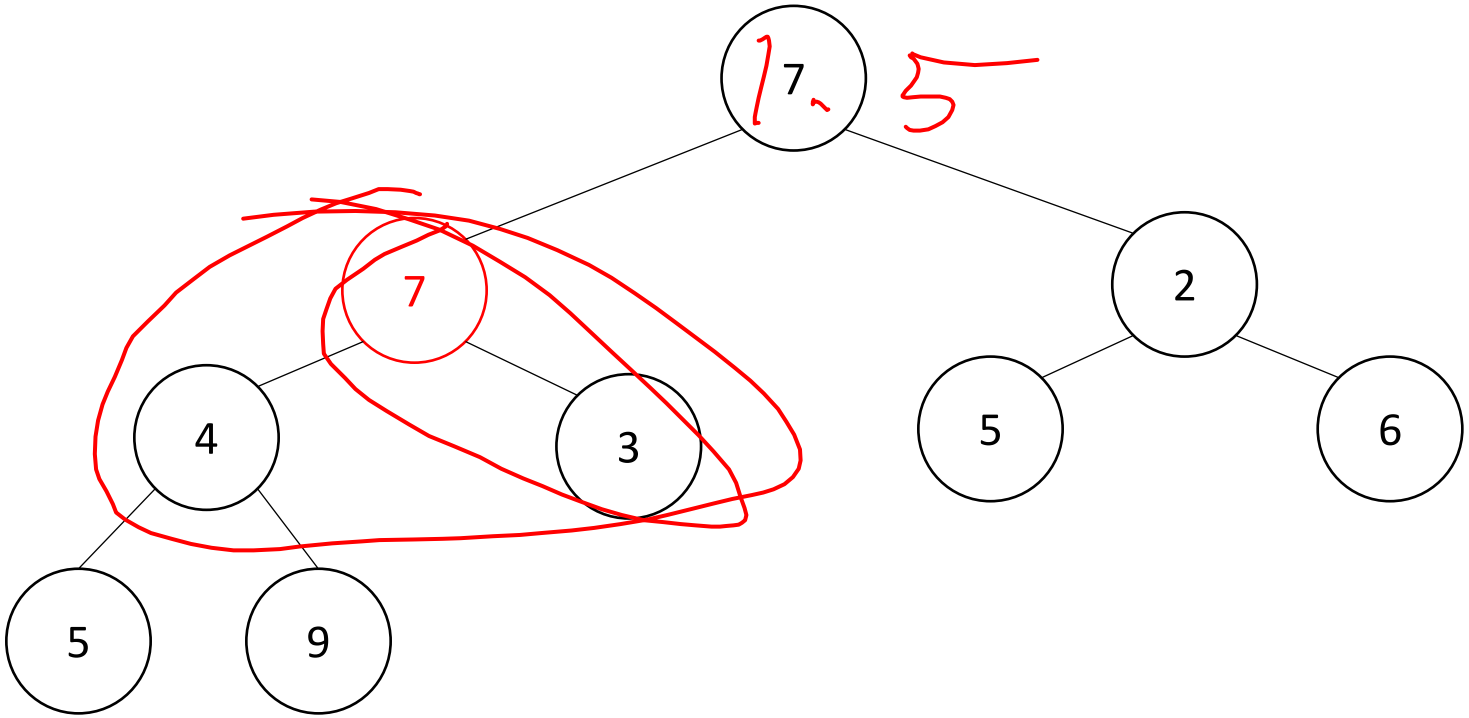

```
deleteMin(){
    min = root
    br = bottom-right item
    move br to the root
    while(br > either of its children){
        swap br with its smallest child
    }
    return min
}
```

Heap deleteMin




```
deleteMin(){
    min = root
    br = bottom-right item
    move br to the root
    while(br > either of its children){
        swap br with its smallest child
    }
    return min
}
```

Heap deleteMin

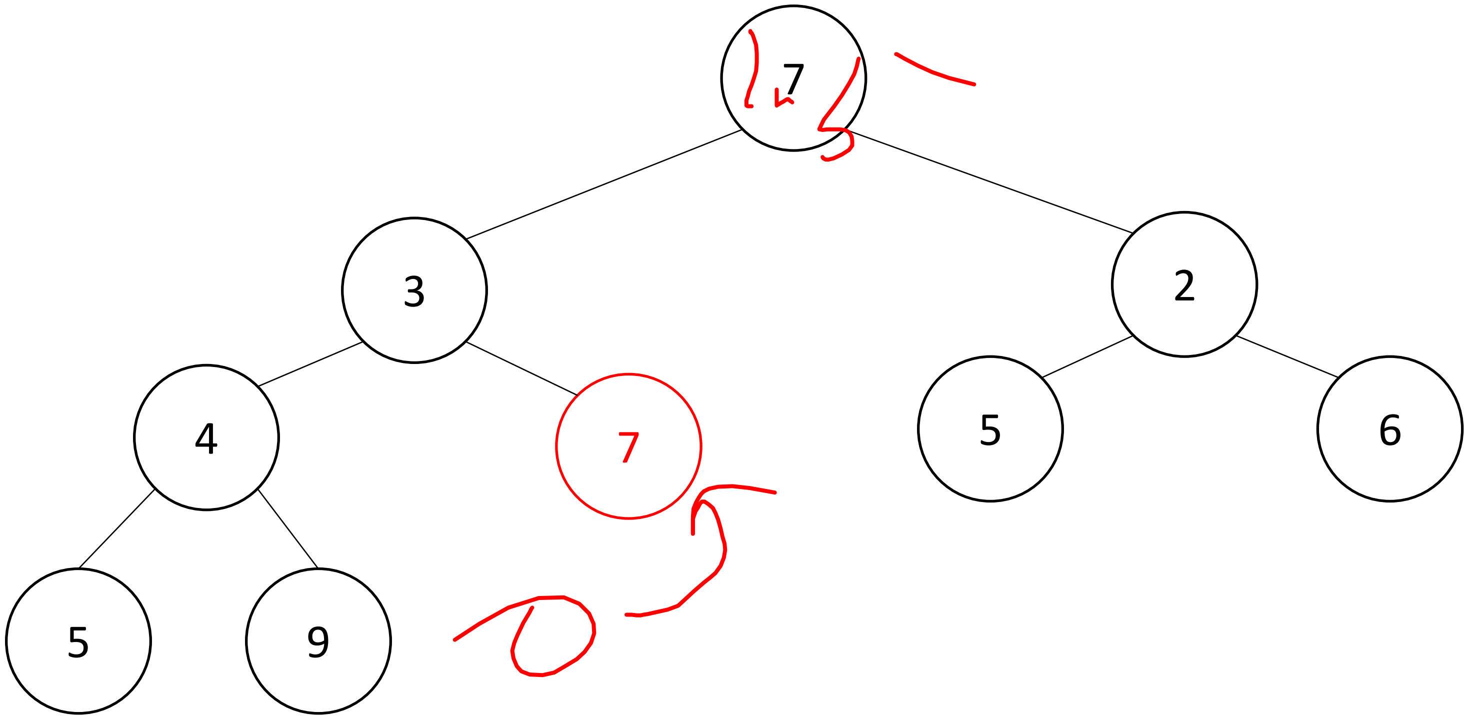



```
deleteMin(){
    min = root
    br = bottom-right item
    move br to the root
    while(br > either of its children){
        swap br with its smallest child
    }
    return min
}
```

Percolate Down

Heap deleteMin

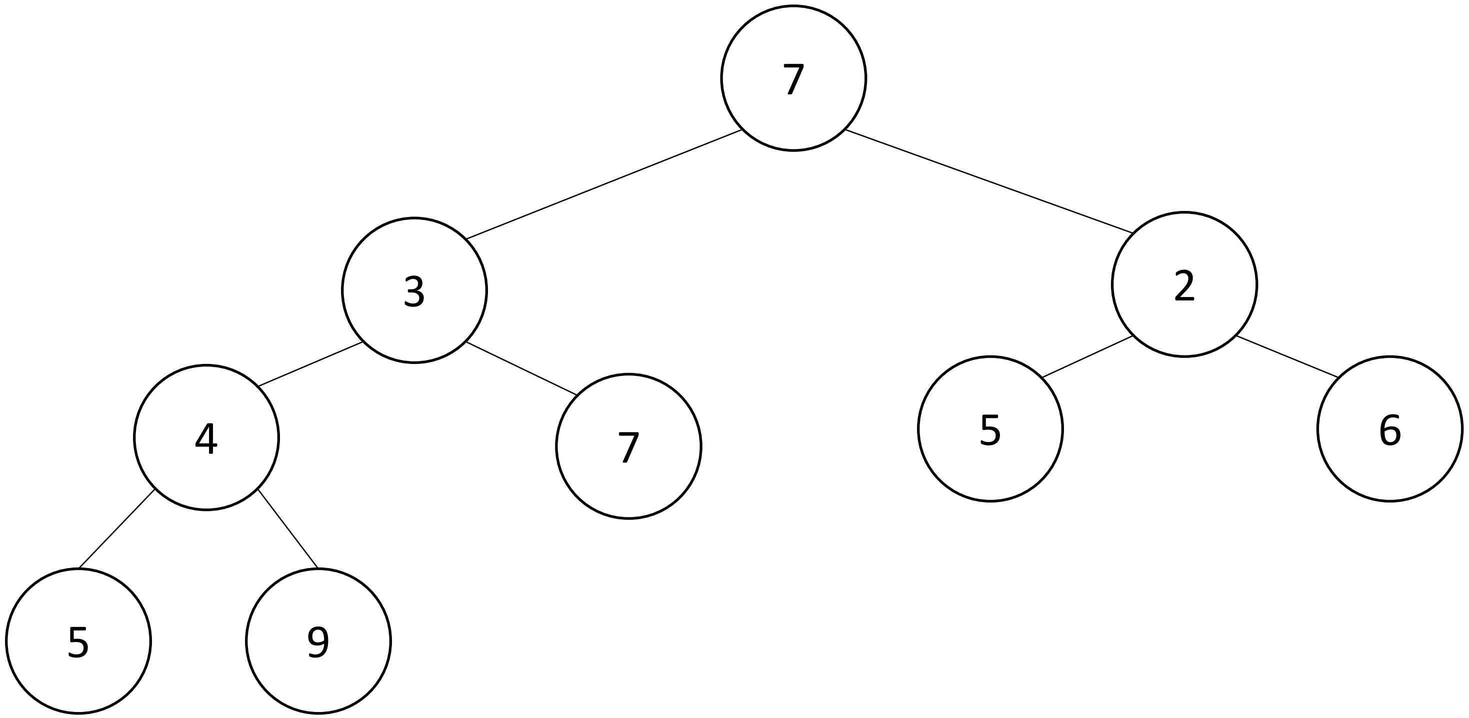
$\Theta(\log n)$




```
deleteMin(){
    min = root
    br = bottom-right item
    move br to the root
    while(br > either of its children){
        swap br with its smallest child
    }
    return min
}
```

Percolate Down

Heap deleteMin

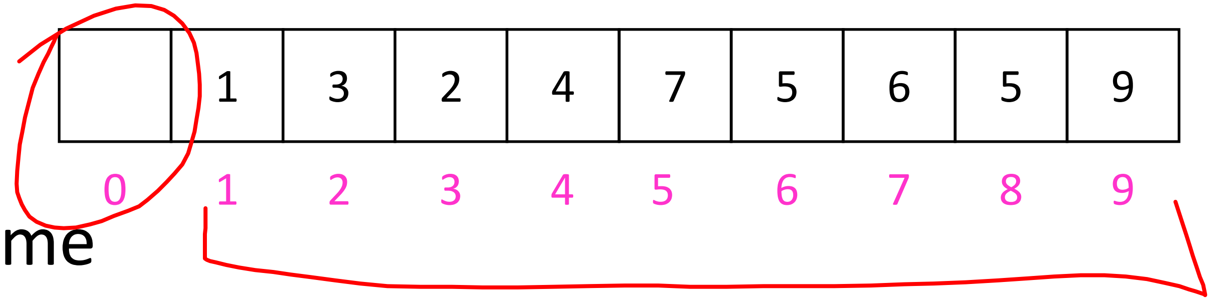



```
deleteMin(){
    min = root
    br = bottom-right item
    move br to the root
    while(br > either of its children){
        swap br with its smallest child
    }
    return min
}
```

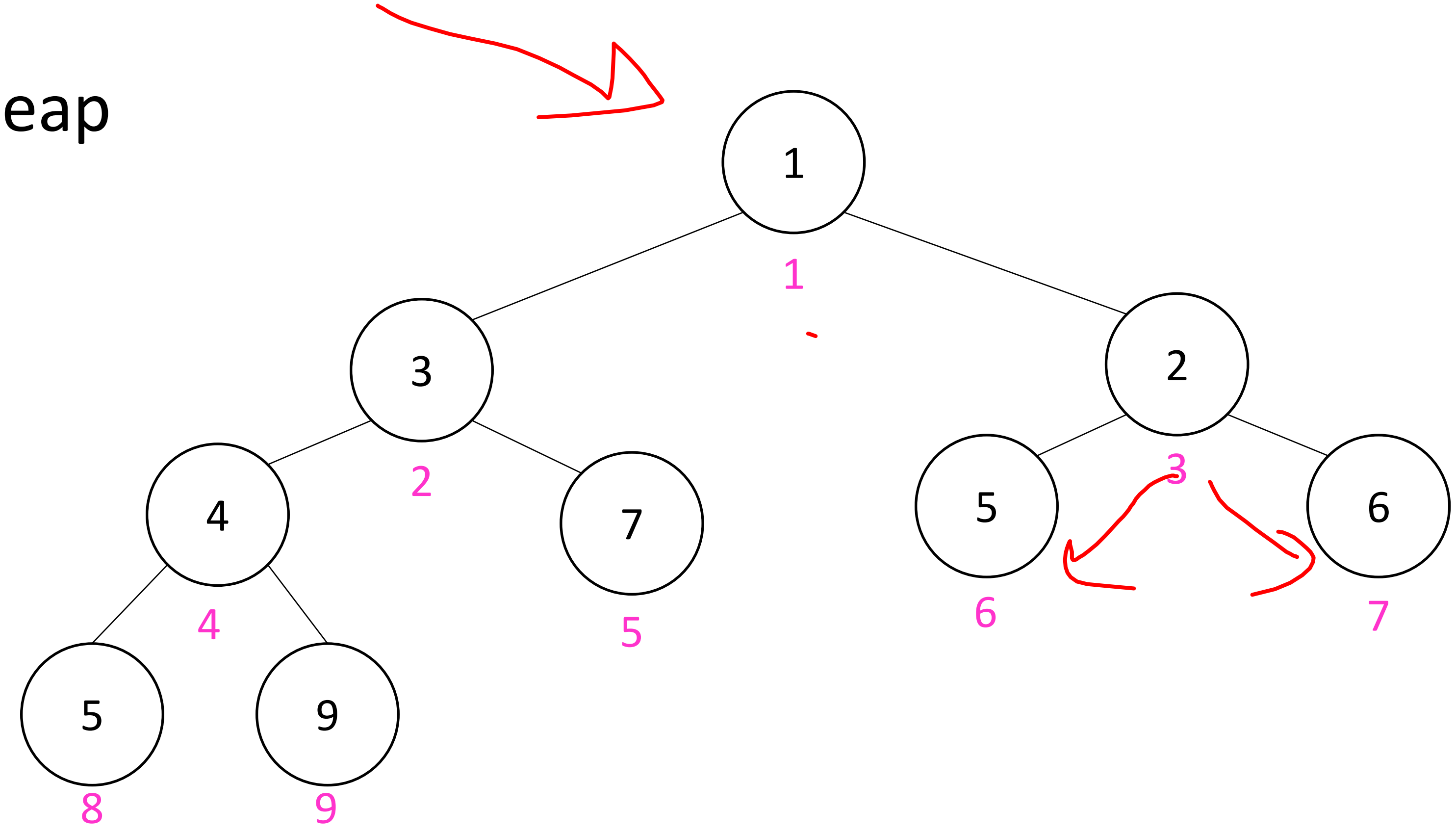
Percolate Up and Down

- Goal: restore the "Heap Property"
- Percolate Up:
  - Take a node that may be smaller than a parent, repeatedly swap with a parent until it is larger than its parent
- Percolate Down:
  - Take a node that may be larger than one of its children, repeatedly swap with smallest child until both children are larger
- Worst case running time of each:
  - $\Theta(\log n)$

Representing a Heap



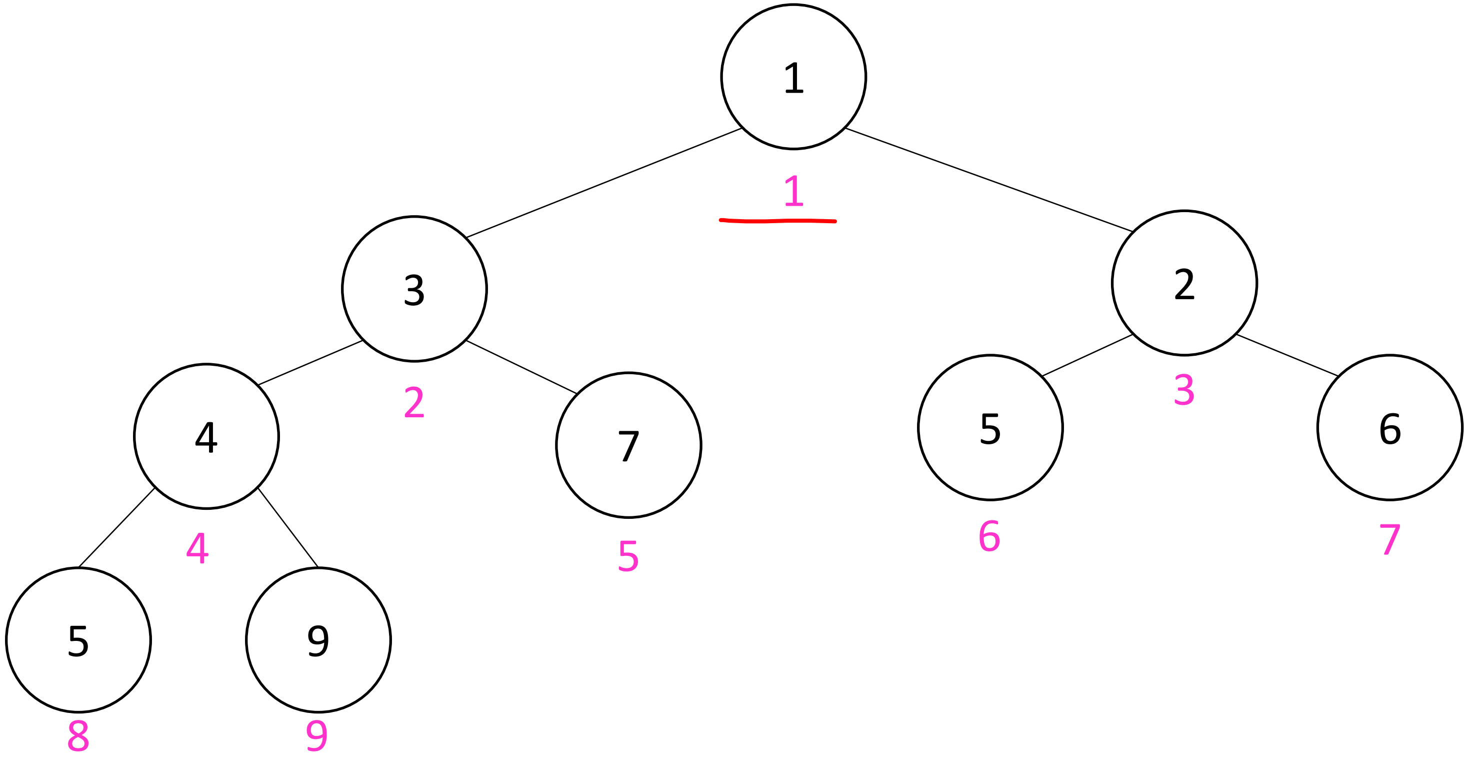
- Every complete binary tree with the same number of nodes uses the same positions and edges
- Use an array to represent the heap
- Index of root: 1
- Parent of node i: $\left\lfloor \frac{i}{2} \right\rfloor$
- Left child of node i: $2 \cdot i$
- Right child of node i: $2 \cdot i + 1$
- Location of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



Insert Psuedocode

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

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



Percolate Up

```
percolateUp(i){
    parent = i/2;  \\ index of parent
    val = arr[i];  \\ value at location
    while(i > 1 && arr[i] < arr[parent]){  \\ until location is root or heap property holds
        arr[i] = arr[parent];  \\ move parent value to this location
        arr[parent] = val; \\ put current value into parent's location
        i = parent;  \\ make current location the parent
        parent = i/2;  \\ update new parent
    }
}
```

DeleteMin Psuedocode

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

Percolate Down

```
percolateDown(i){
  left = i*2;  \\ index of left child
  right = i*2+1;  \\ index of right child
  val = arr[i];  \\ value at location
  while(left <= size){  \\ until location is leaf
    toSwap = right;
    if(right > size || arr[left] < arr[right]){  \\ if there is no right child or if left child is smaller
      toSwap = left;  \\ swap with left
    } \\ now toSwap has the smaller of left/right, or left if right does not exist
    if (arr[toSwap]< val){  \\ if the smaller child is less than the current value
      arr[i] = arr[toSwap];
      arr[toSwap] = val; \\ swap parent with smaller child
      i = toSwap; \\ update current node to be smaller child
      left = i*2;
      right = i*2+1;
    }
    else{ break;} \\ if we don't swap, then heal property holds
  }
}
```

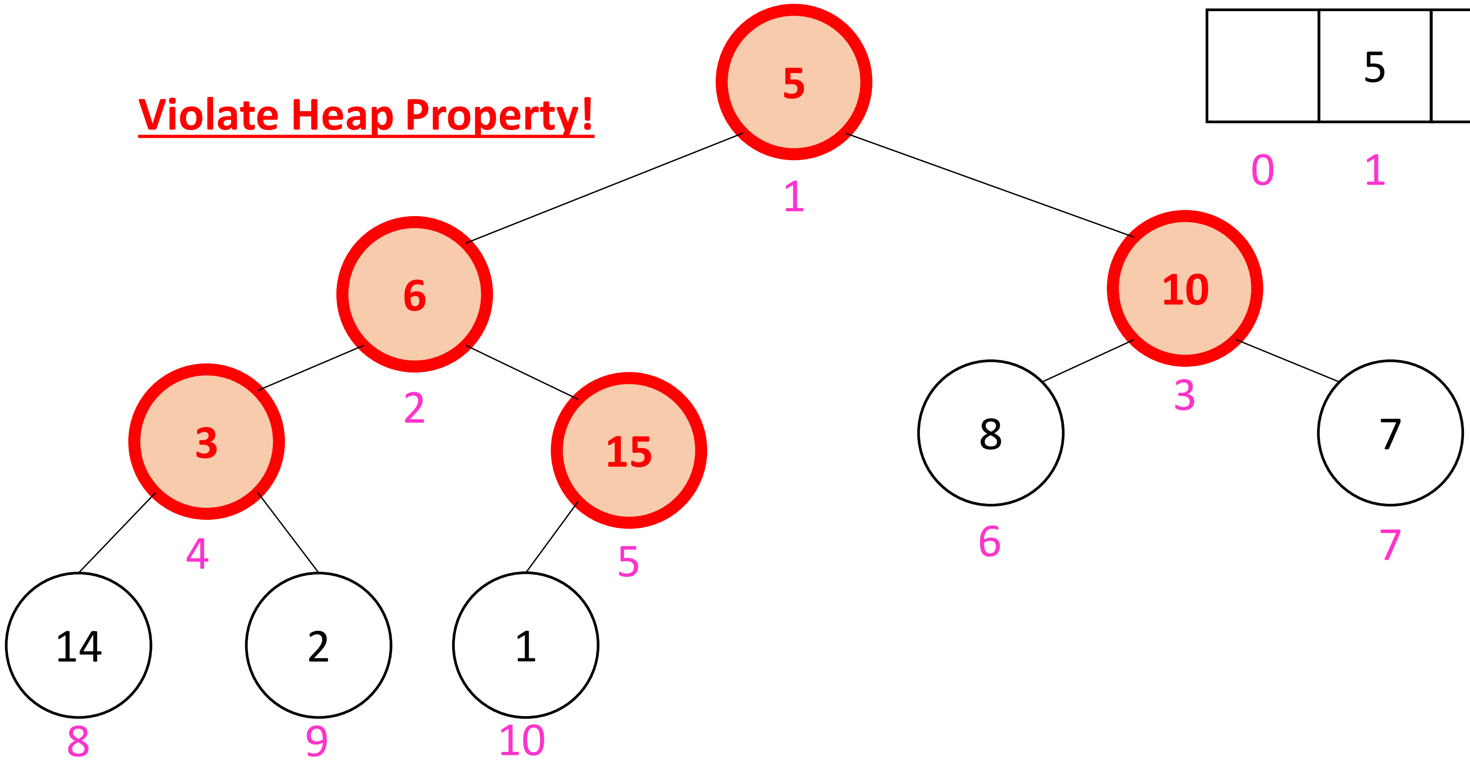
Other Operations

- Increase Key
  - Given the index of an item in the PQ, subtract from its priority value
- Decrease Key
  - Given the index of an item in the PQ, add to its priority value
- Remove
  - Given the item at the given index from the PQ

Aside: Expected Running time of Insert

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

Two ways for "fix" the heap:

1) Percolate Up
2) Percolate Down

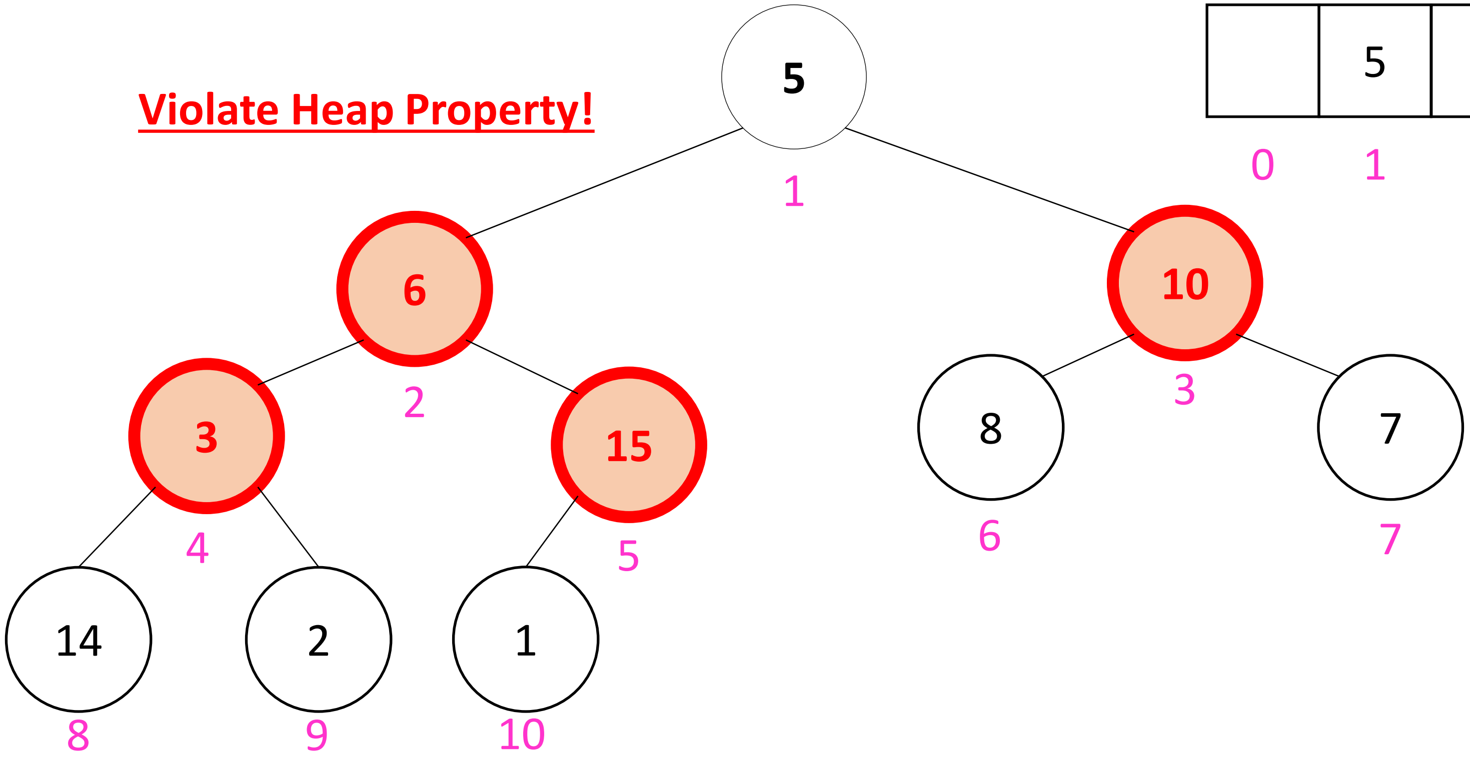
Floyd’s buildHeap method

- 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

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

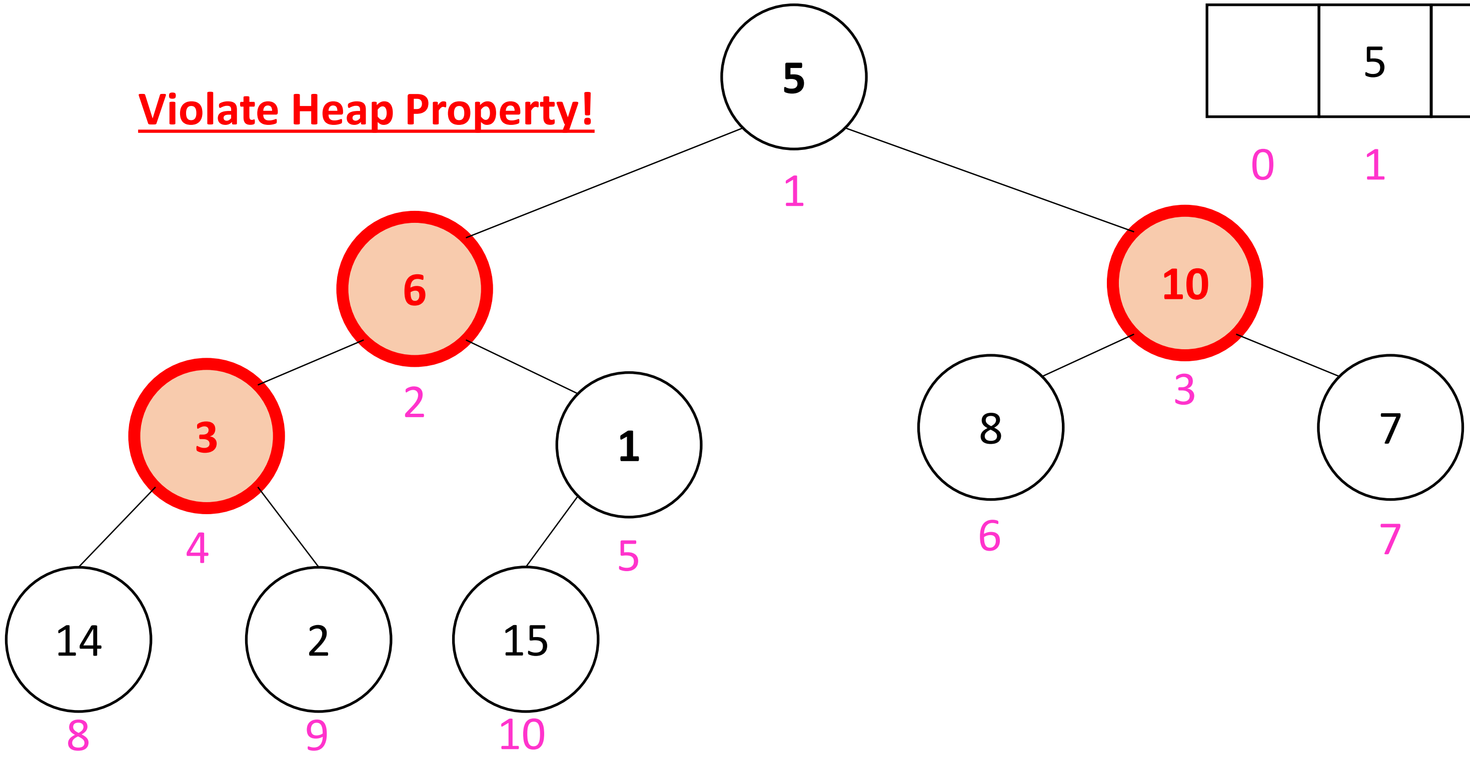


	5	6	10	3	15	8	7	14	2	1
0	1	2	3	4	5	6	7	8	9	10

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

Floyd’s buildHeap method

- Suppose we had n items and wanted to “heapify” them

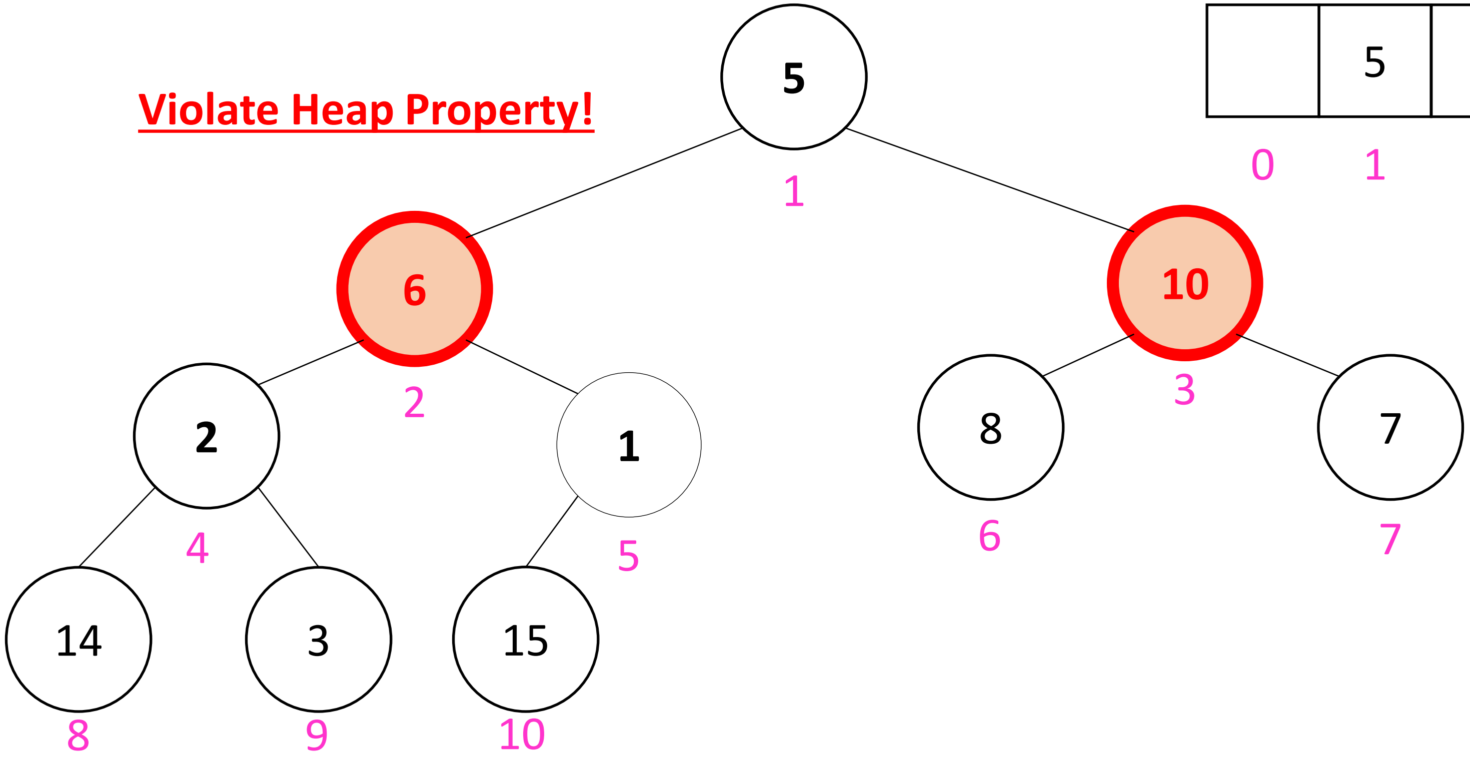


	5	6	10	3	15	8	7	14	2	1
0	1	2	3	4	5	6	7	8	9	10

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

Floyd’s buildHeap method

- Suppose we had n items and wanted to “heapify” them

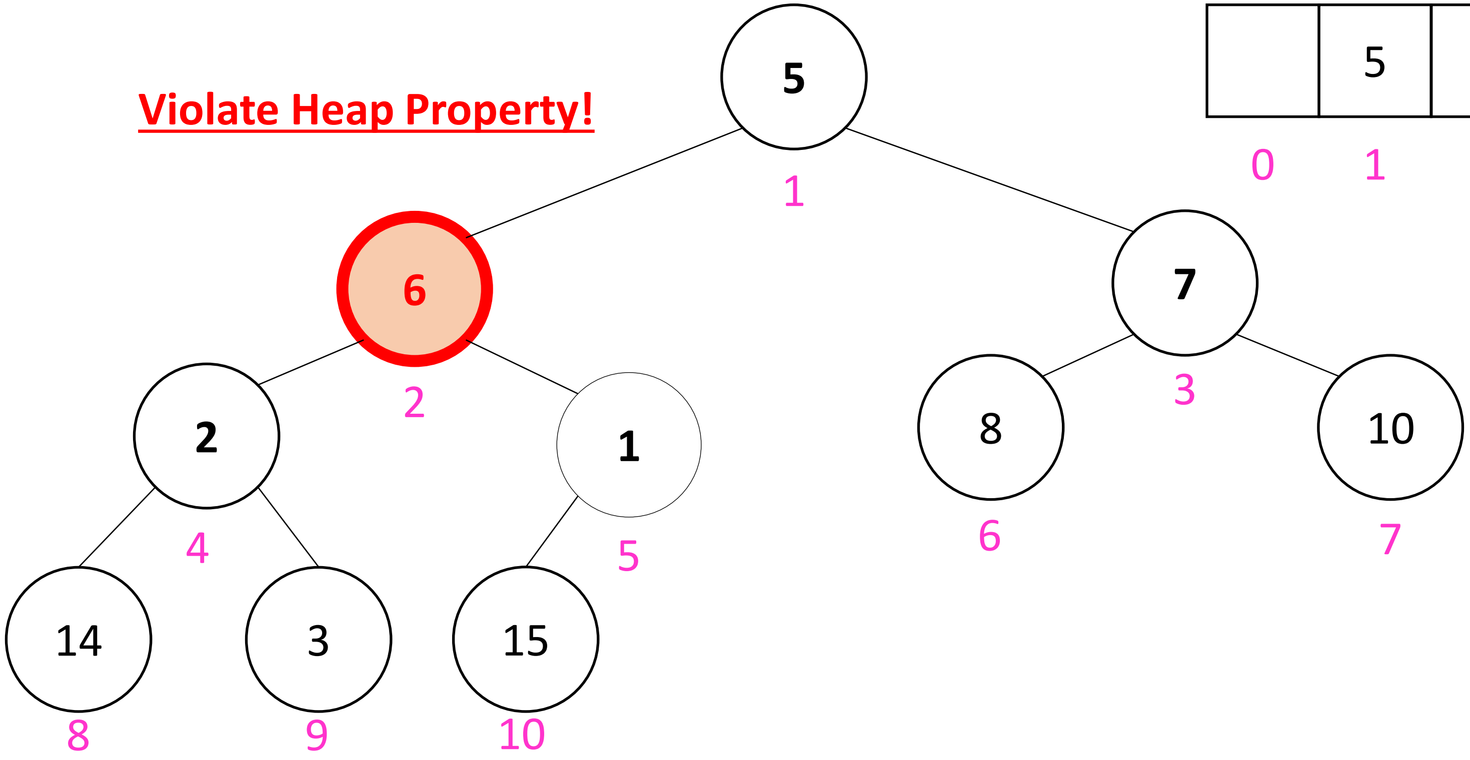


	5	6	10	3	15	8	7	14	2	1
0	1	2	3	4	5	6	7	8	9	10

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

Floyd's buildHeap method

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

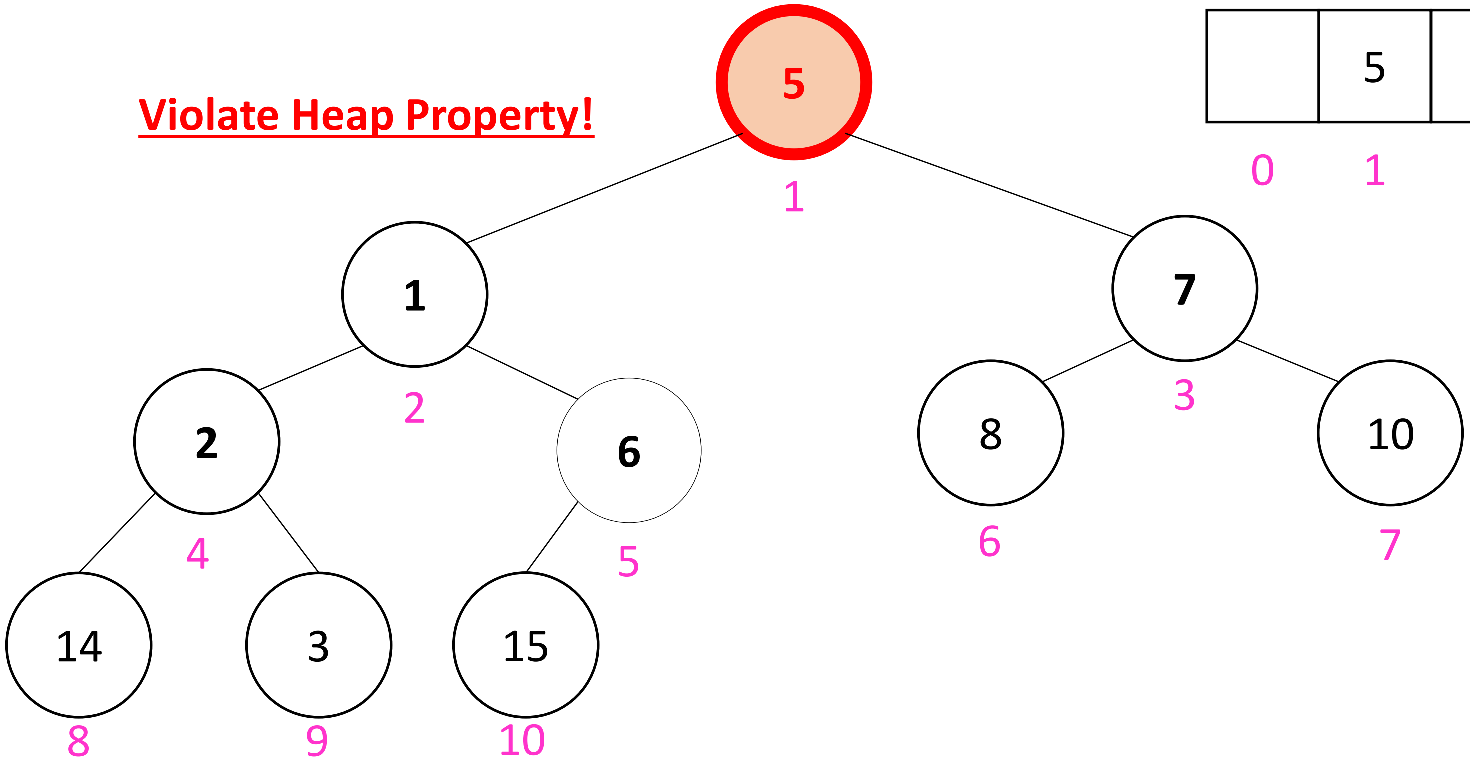


0	1	2	3	4	5	6	7	8	9	10
	5	6	10	3	15	8	7	14	2	1

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

Floyd’s buildHeap method

- Suppose we had n items and wanted to “heapify” them

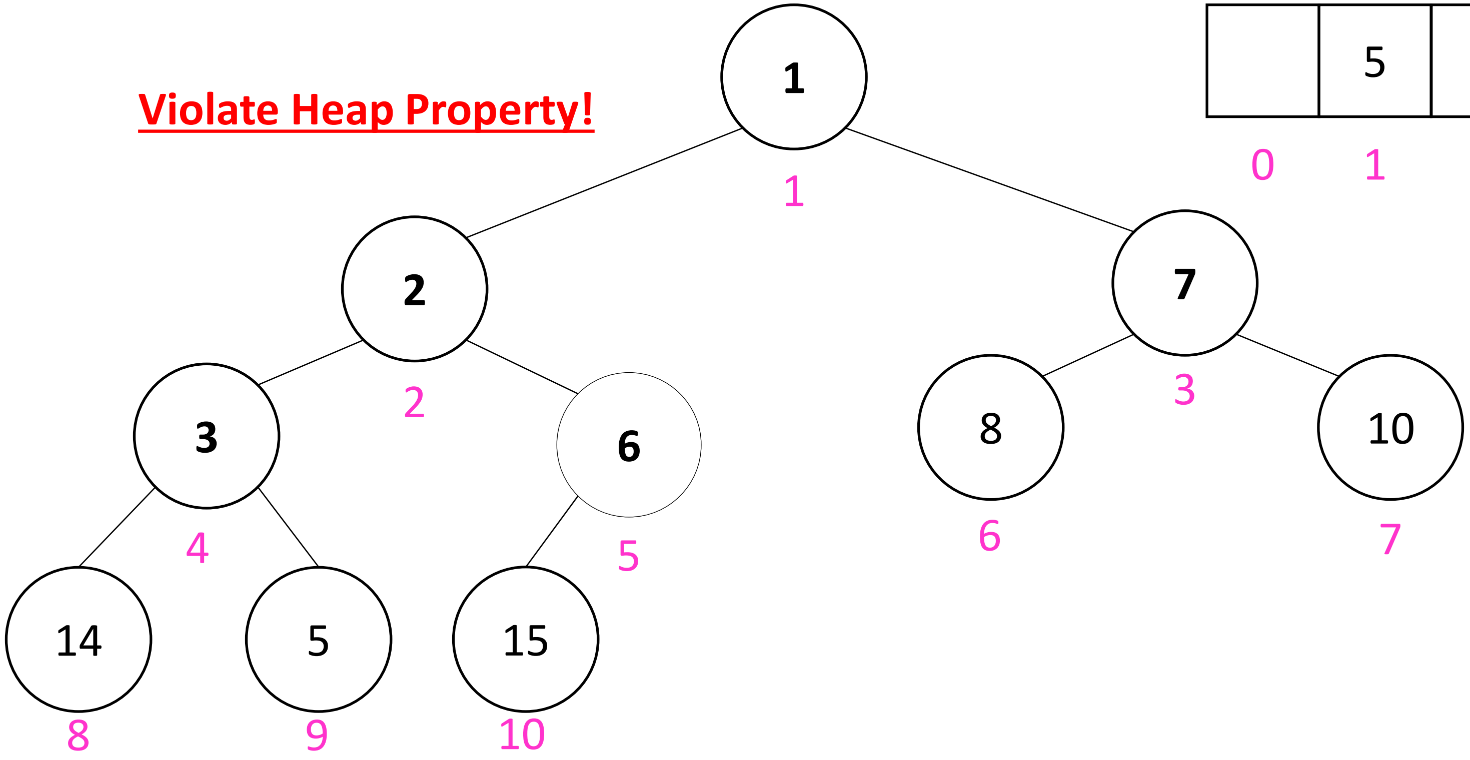


	5	6	10	3	15	8	7	14	2	1
0	1	2	3	4	5	6	7	8	9	10

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

Floyd's buildHeap method

- 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

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

How long did this take?

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

- 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: