

CSE 332 Autumn 2026

Lecture 4: Priority Queues, pt 1

Ruth Anderson

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

Announcements

- Exercise 0
 - Due Friday 10/09/2026
 - There are 2 separate Gradescope submissions
- “Meet the staff” activity - Sometime during the first 4 weeks of class.
- Participate in Section, earn a late day (see Ed Board post)
- Exercise 1 Running Time:
 - Due Mon Oct 12
 - Entirely in Gradescope. Save your answers outside of Gradescope and confirm that you have answered/uploaded everything that is needed!
- Exercise 2 Heaps: coming end of week, due Fri Oct 16

Review: 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

Scenario

What is the difference between waiting for service at a pharmacy versus an ER?

Pharmacies usually follow the rule
First Come, First Served

Emergency Rooms assign priorities
based on each individual's need

Scenario (answer)

What is the difference between waiting for service at a pharmacy versus an ER?

Queue

Pharmacies usually follow the rule
First Come, First Served

Priority
Queue

Emergency Rooms assign priorities
based on each individual's need

ADT: Priority Queue

- What is it?
 - A collection of items and their “priorities”
 - Allows quick access/removal to the “top priority” thing
 - Usually a smaller priority value means the item is “more important”
- What Operations do we need?
 - insert(item, priority)
 - Add a new item to the PQ with indicated priority
 - extract
 - Remove and return the “top priority” item from the queue
 - Usually the item with the smallest priority value
 - isEmpty
 - 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)

A priority queue holds compare-able data

- Unlike stacks and queues need to *compare items*
 - Given x and y , is x less than, equal to, or greater than y
 - What this means can depend on your data
 - Much of course will require comparable data: e.g. sorting
- Integers are comparable, so will use them in examples
 - But the priority queue ADT is much more general
 - Typically two fields, the *priority* and the *data*
- For simplicity in lecture, we'll often suppose items are just **ints** and the **int** is also the priority

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?

Some Example Applications

- Treat hospital patients in order of severity (or triage)
- Run multiple programs in the operating system
 - “critical” before “interactive” before “compute-intensive”
 - Maybe let users set priority level
- Forward network packets in order of urgency
- Select most frequent symbols for data compression (see CSE123?)
- Will see an example when we study graphs in a few weeks
- Sort: **insert** all, then repeatedly **extract**

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...)

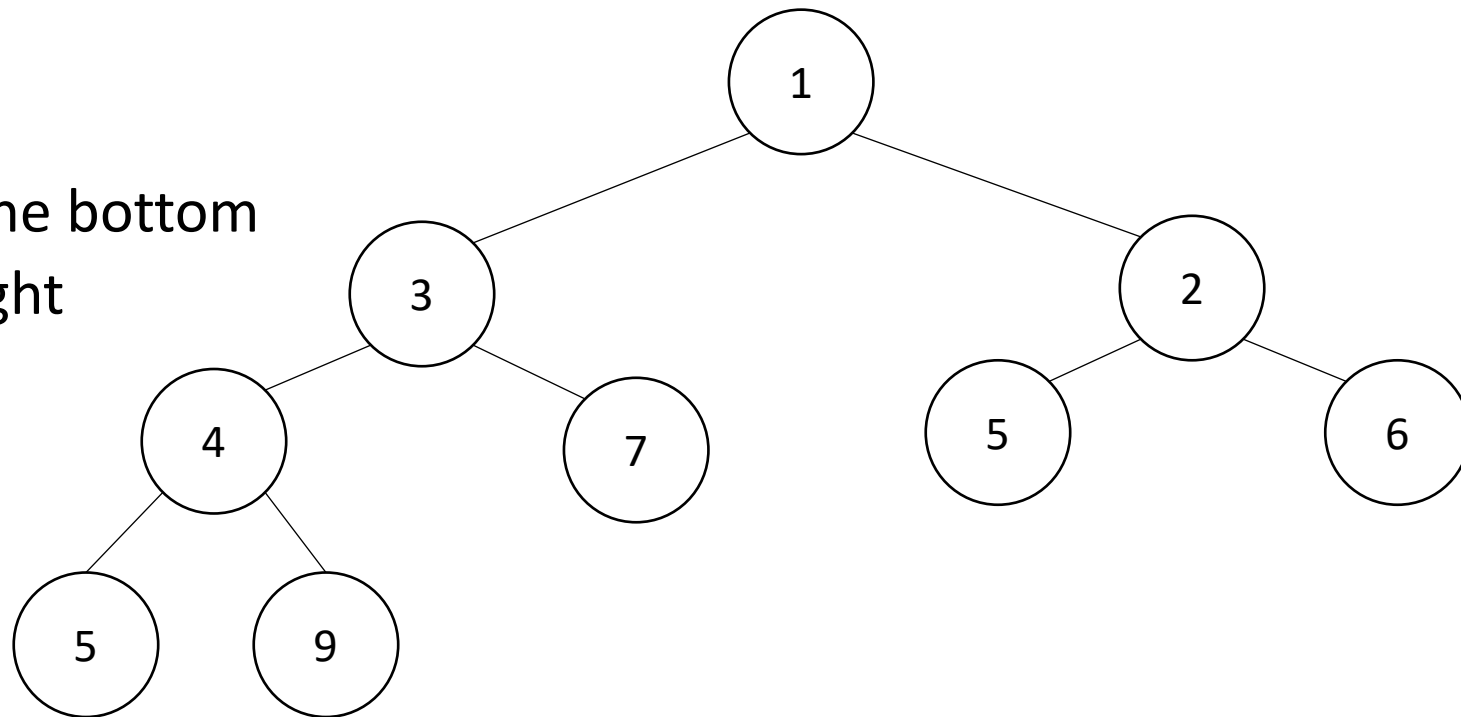
Thinking through implementations (solutions)

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...)

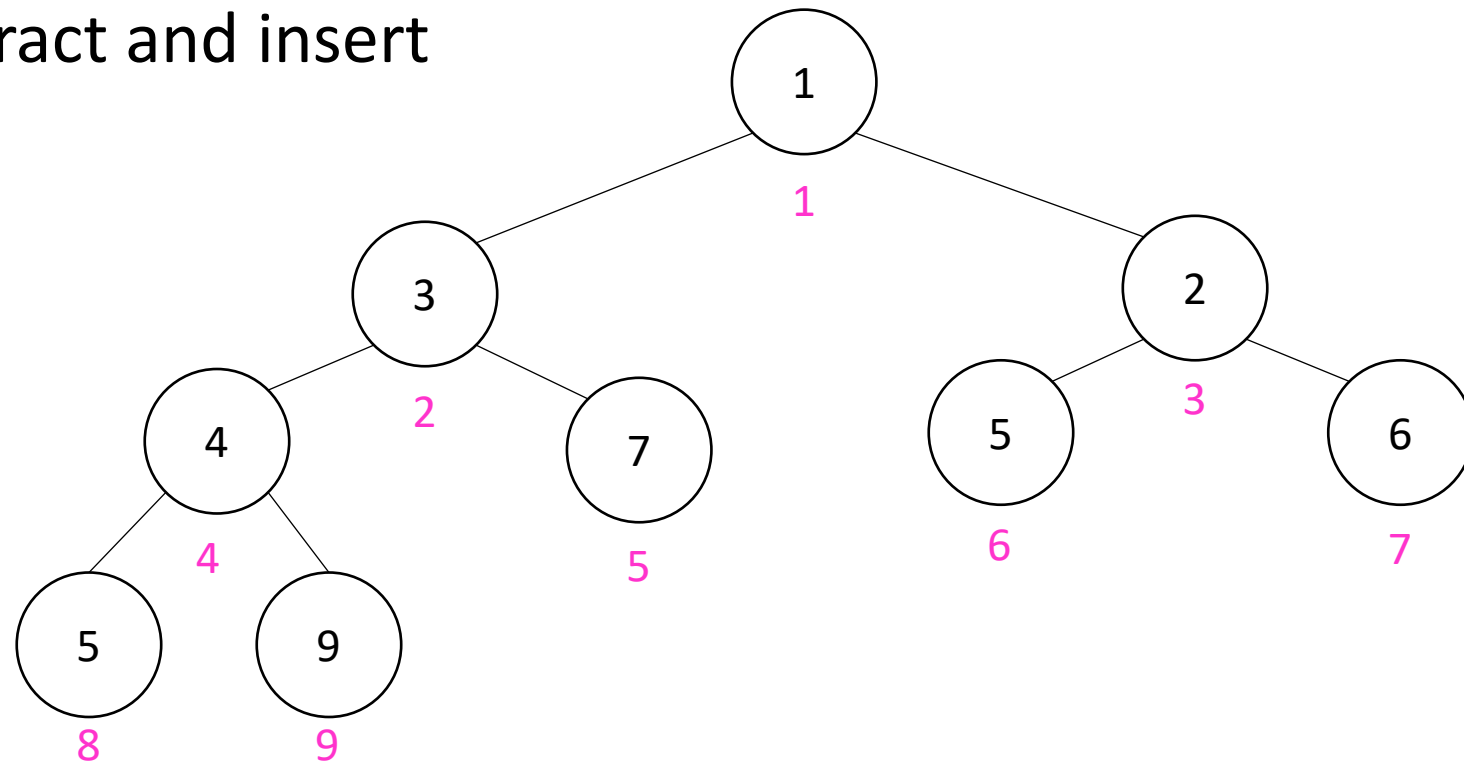
Trees for Heaps

- 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



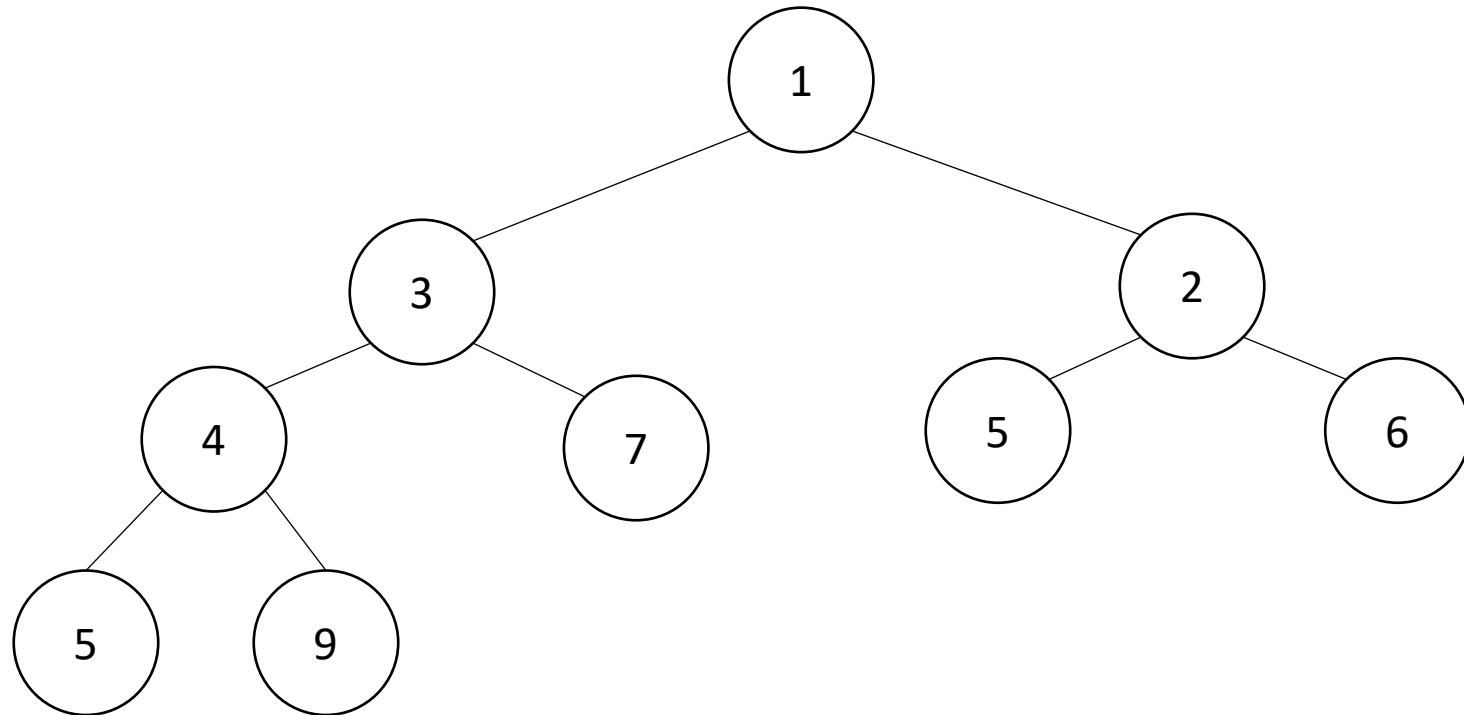
Heap – Priority Queue Data Structure

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



How $\log n$ Height?

- **height** – number of edges in path from node to deepest descendent

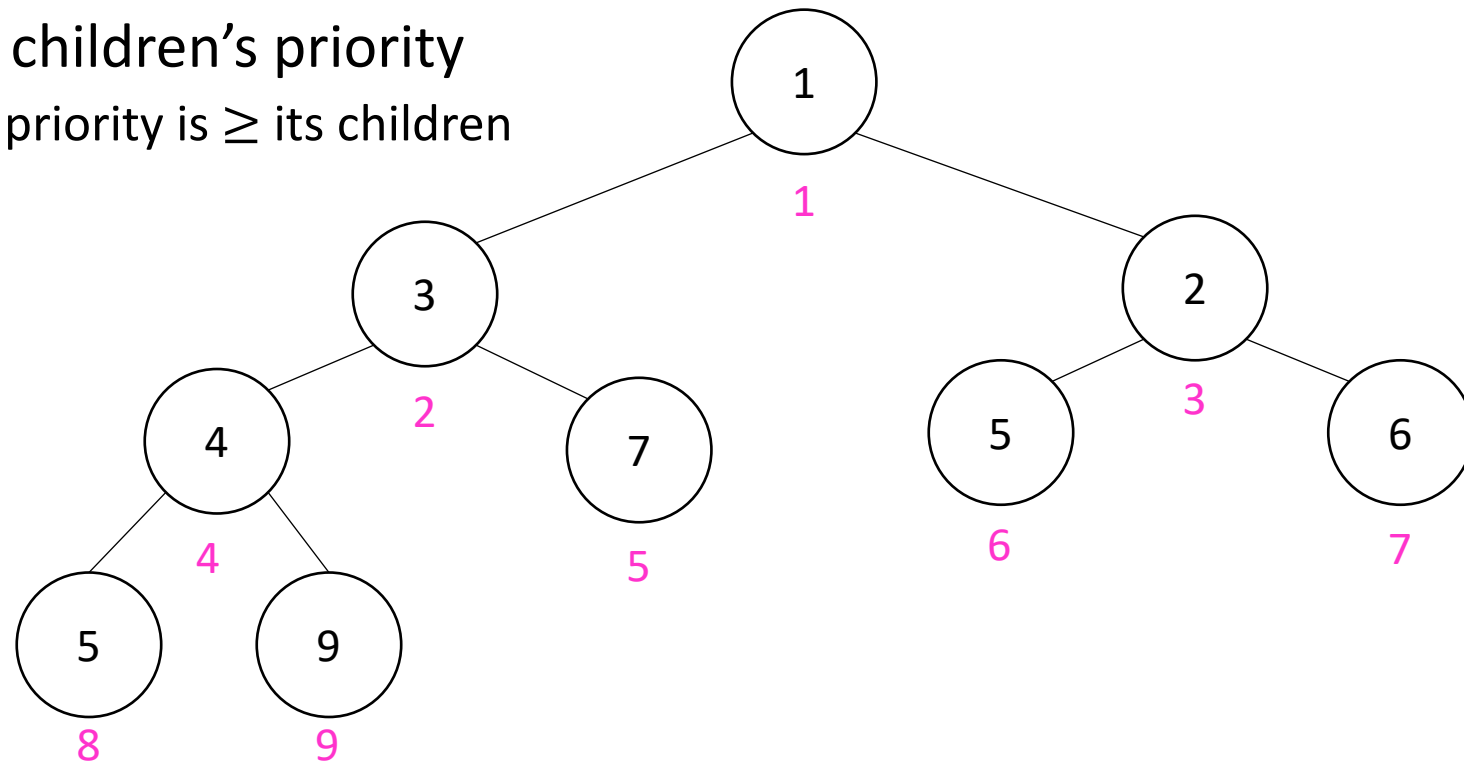


Achieving $\log n$ Running Time

- 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?
 - Find the smallest h such that: $n \leq 2^{h+1} - 1$
 - Solve for h : $\lceil \log(n + 1) \rceil - 1 = h$
 - Height is $\Theta(\log n)$
- **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

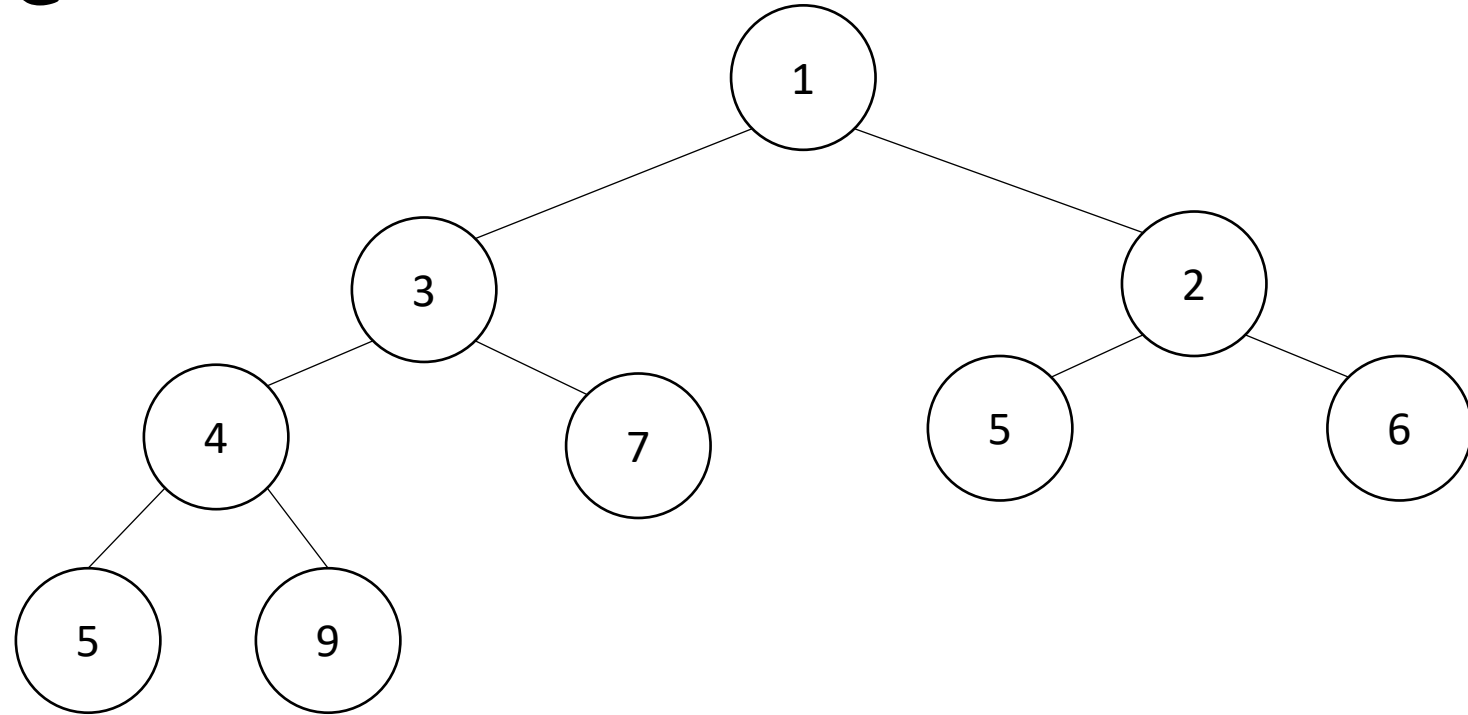
(Min) Heap Data Structure

- Keep items in a complete binary tree
- Maintain the “(Min) Heap Property” of the tree
 - Every node's priority is $\leq$ its children's priority
 - Max Heap Property: every node's priority is $\geq$ its children
- Where is the min?
- How do I insert?
- How do I extract?
- How to do it in Java?



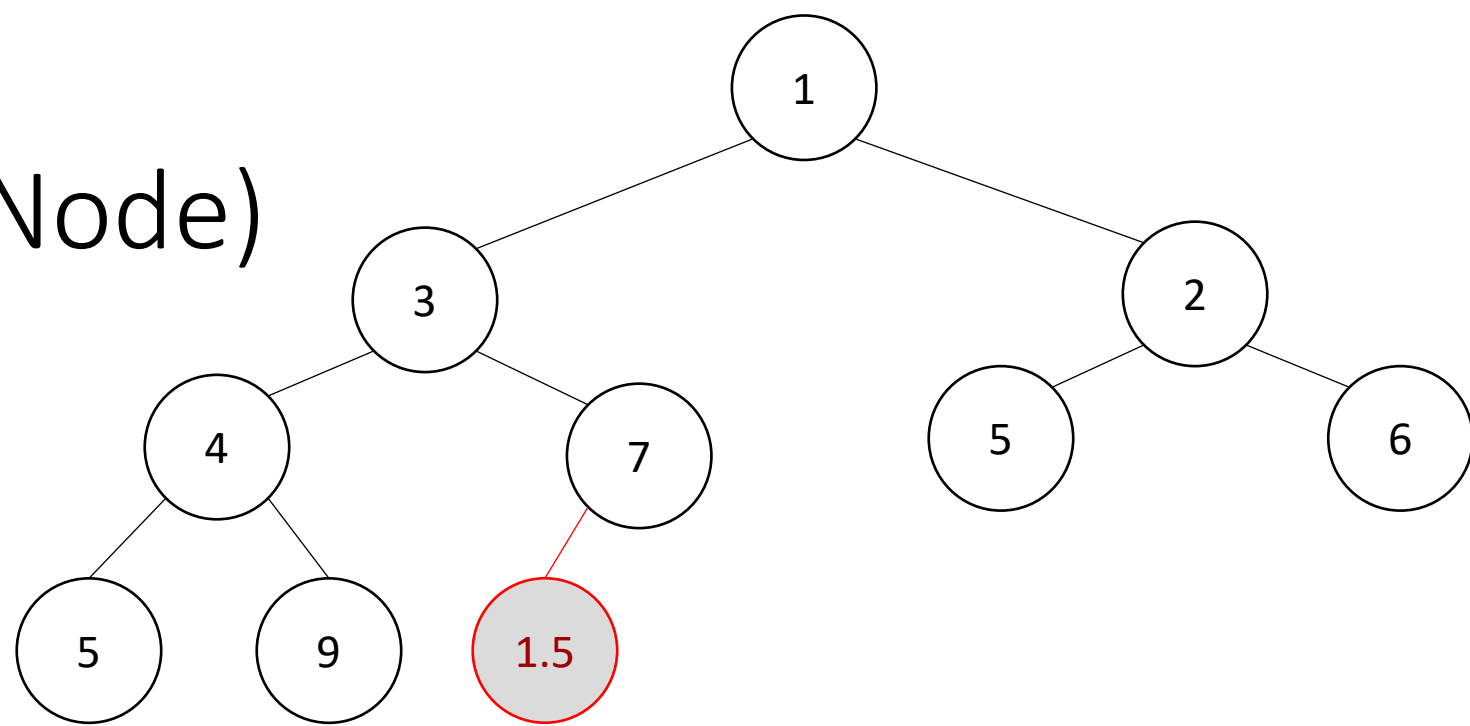
Heap Insert Example

1.5



```
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)
```

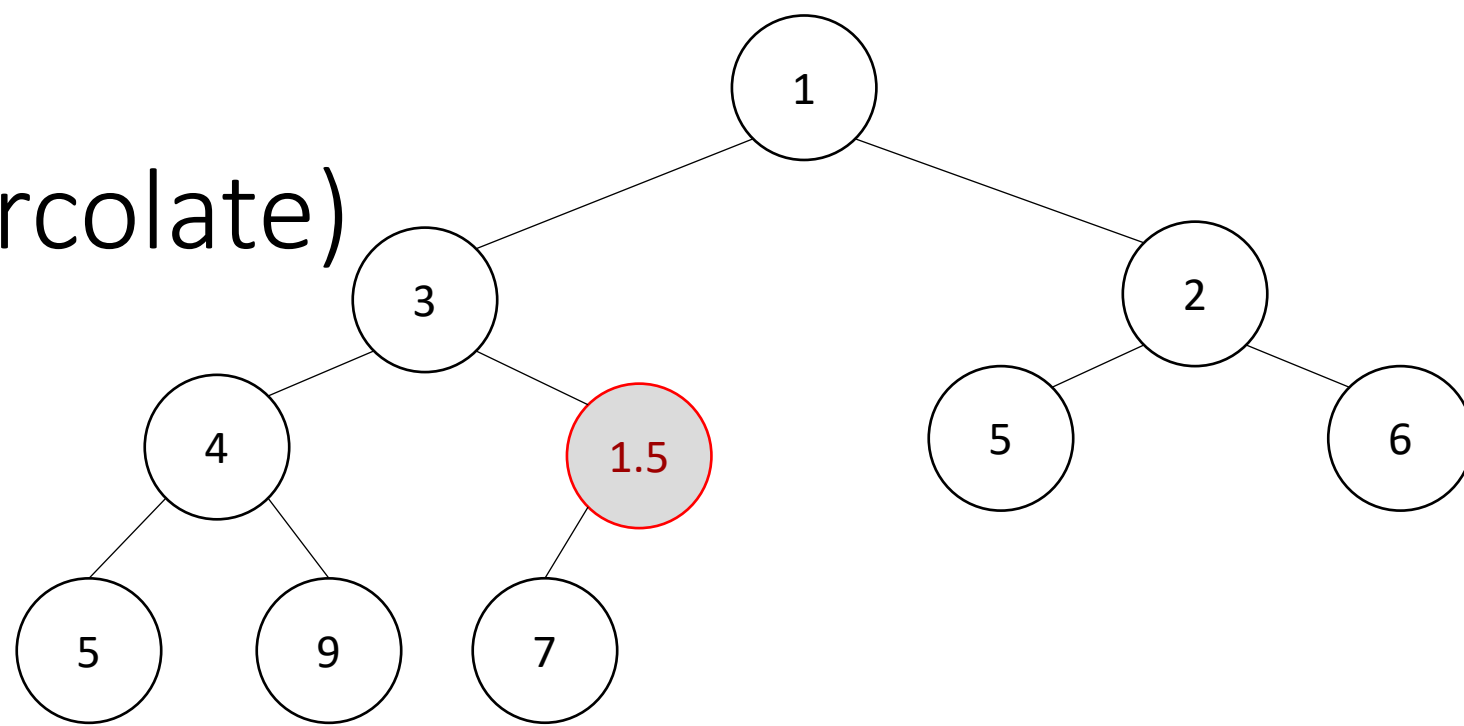
```
    while (priority < parent's priority){
```

```
        swap item with parent
```

```
    }
```

```
}
```

Heap Insert (1st Percolate)



```
insert(item, priority){
```

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

```
    while (priority < parent's priority){
```

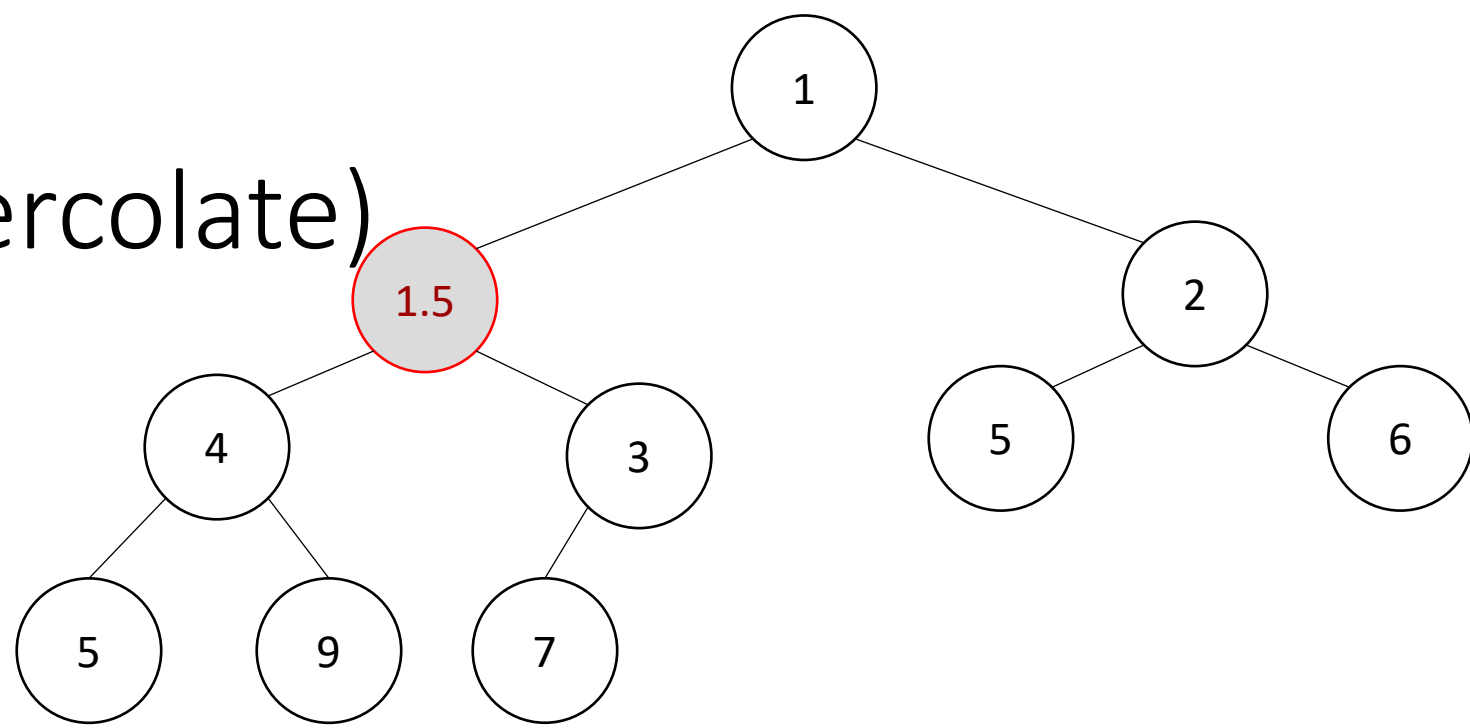
```
        swap item with parent
```

```
    }
```

```
}
```

Percolate Up

Heap Insert (2nd Percolate)



```
insert(item, priority){
```

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

```
    while (priority < parent's priority){
```

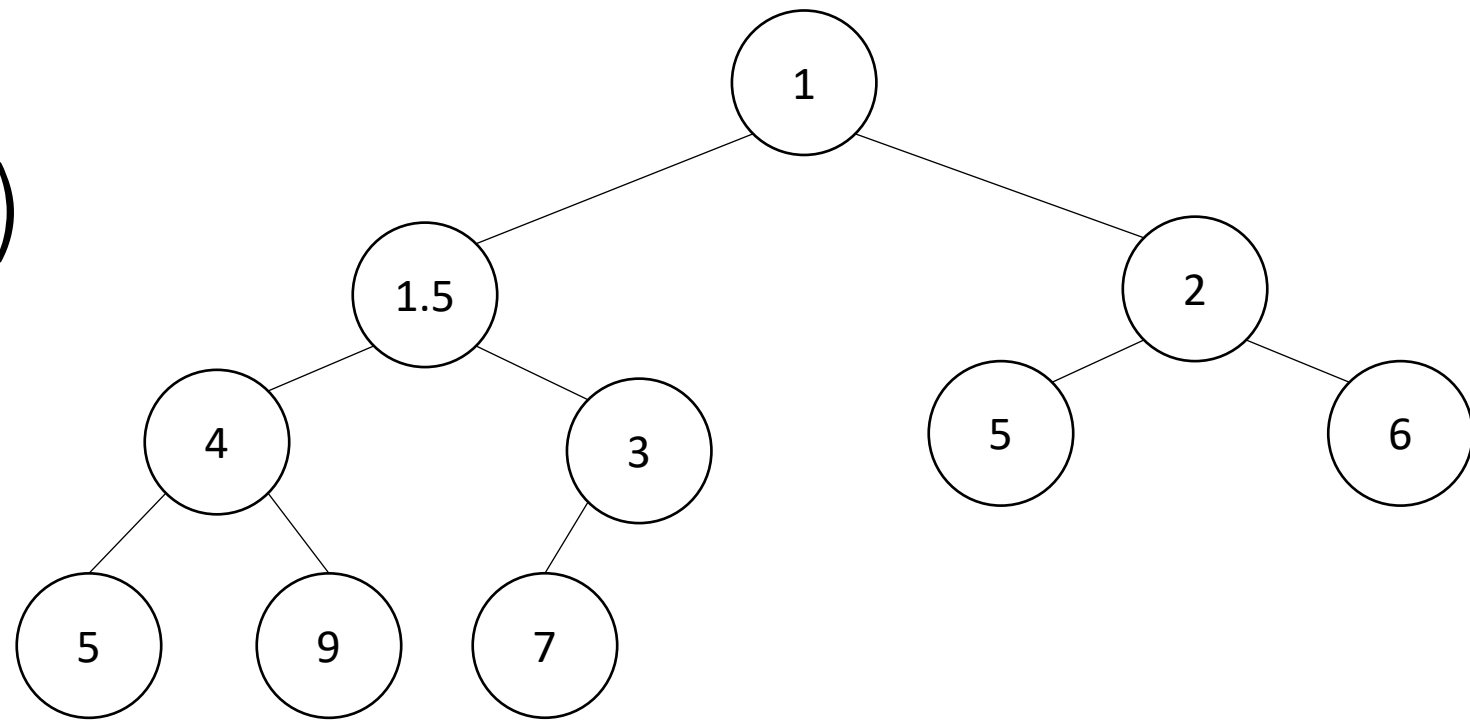
```
        swap item with parent
```

```
    }
```

```
}
```

Percolate Up

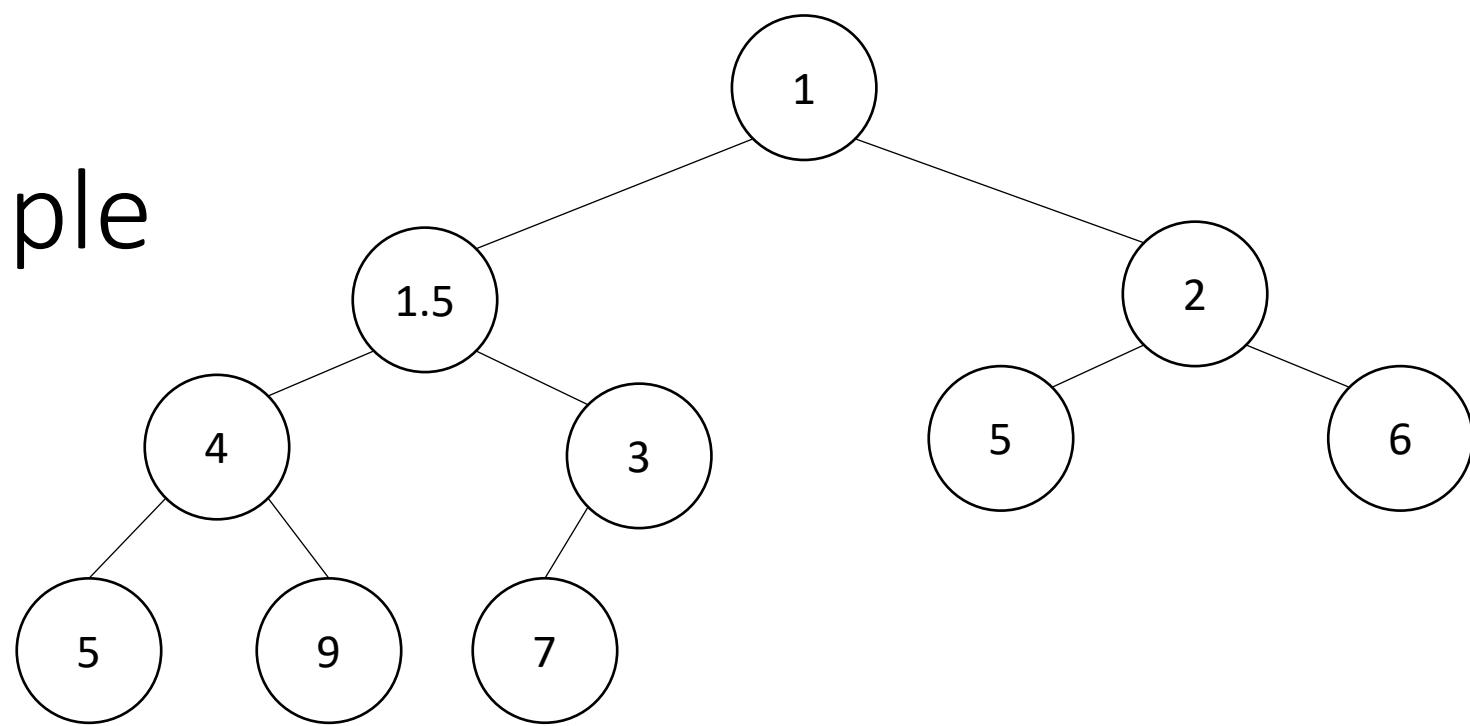
Heap Insert (Done)



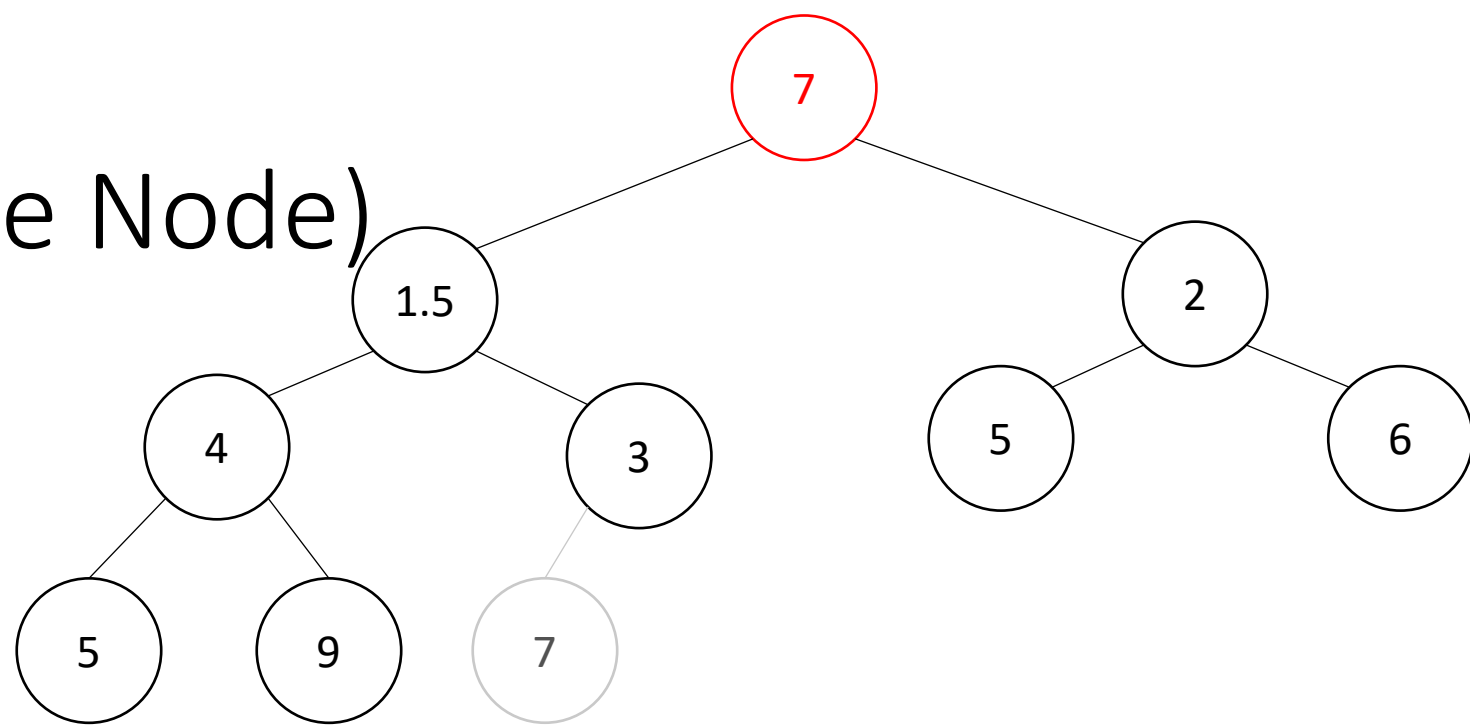
```
insert(item, priority){  
    put item in the “next open” spot (keep tree complete)  
    while (priority < parent’s priority){  
        swap item with parent  
    }  
}
```

Heap extract Example

```
extract(){  
  min = root  
  curr = bottom-right item  
  move curr to the root  
  while(curr > curr.left || curr > curr.right){  
    swap curr with its smallest child  
  }  
  return min  
}
```

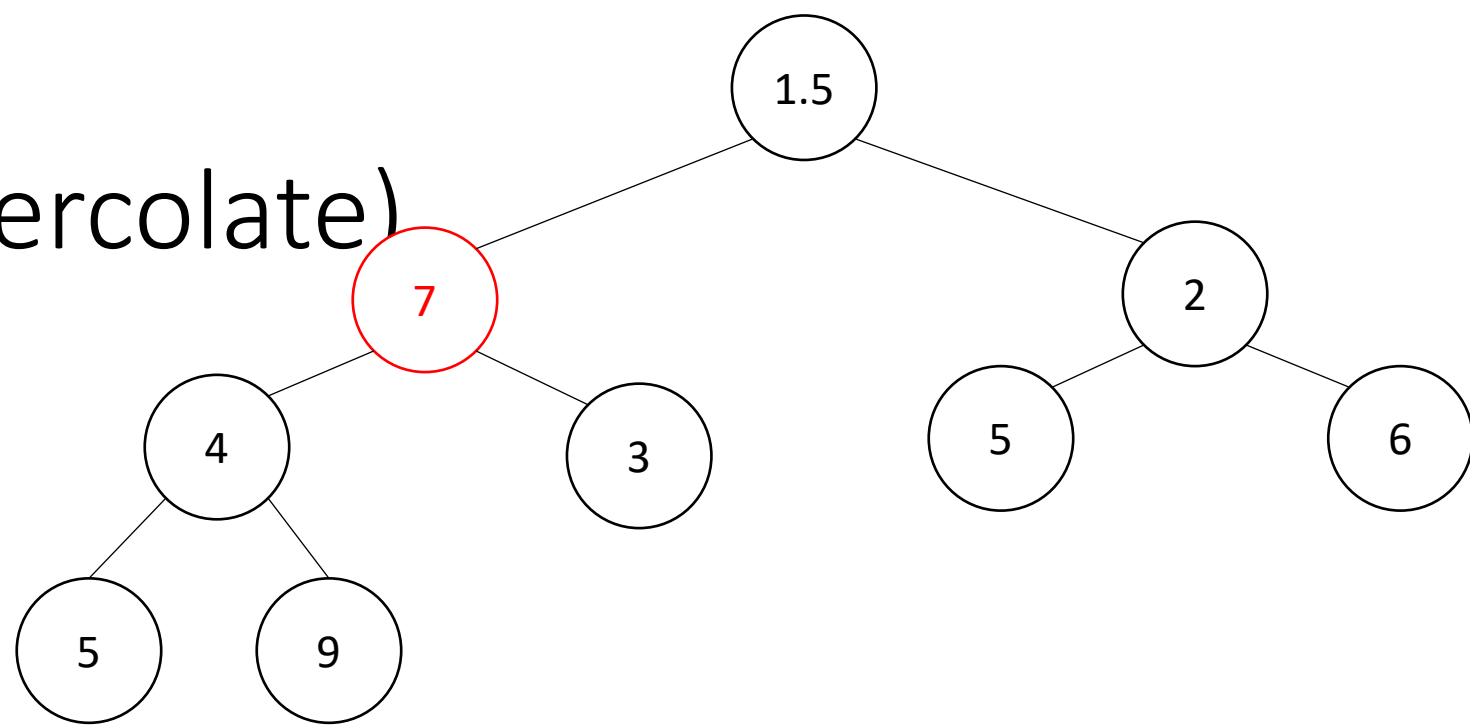


Heap extract (Move Node)



```
extract(){  
    min = root  
    curr = bottom-right item  
    move curr to the root  
    while(curr > curr.left || curr > curr.right){  
        swap curr with its smallest child  
    }  
    return min  
}
```

Heap extract (1st Percolate)



```
extract(){
```

```
    min = root
```

```
    curr = bottom-right item
```

```
    move curr to the root
```

```
    while(curr > curr.left || curr > curr.right){
```

```
        swap curr with its smallest child
```

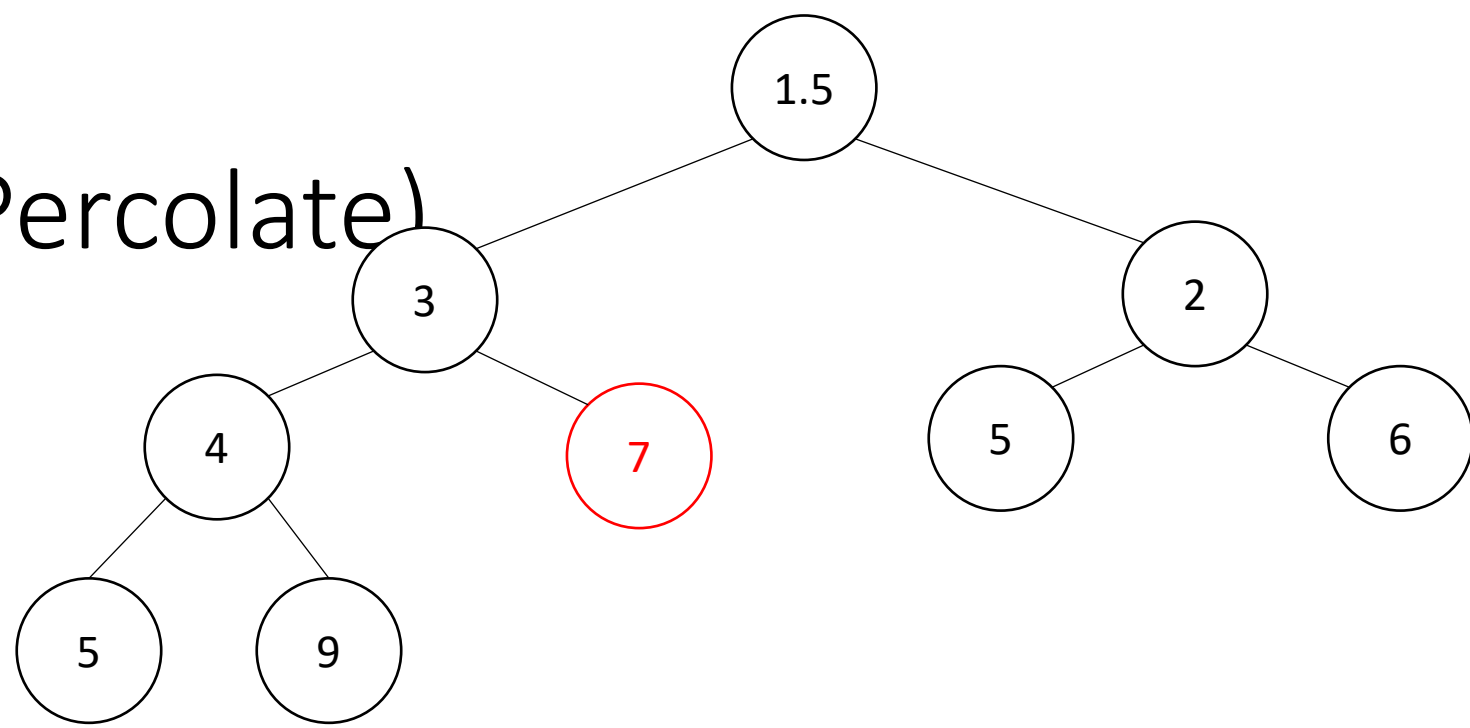
```
    }
```

```
    return min
```

```
}
```

Percolate Down

Heap extract (2nd Percolate)



```
extract(){
```

```
    min = root
```

```
    curr = bottom-right item
```

```
    move curr to the root
```

```
    while(curr > curr.left || curr > curr.right){
```

```
        swap curr with its smallest child
```

```
    }
```

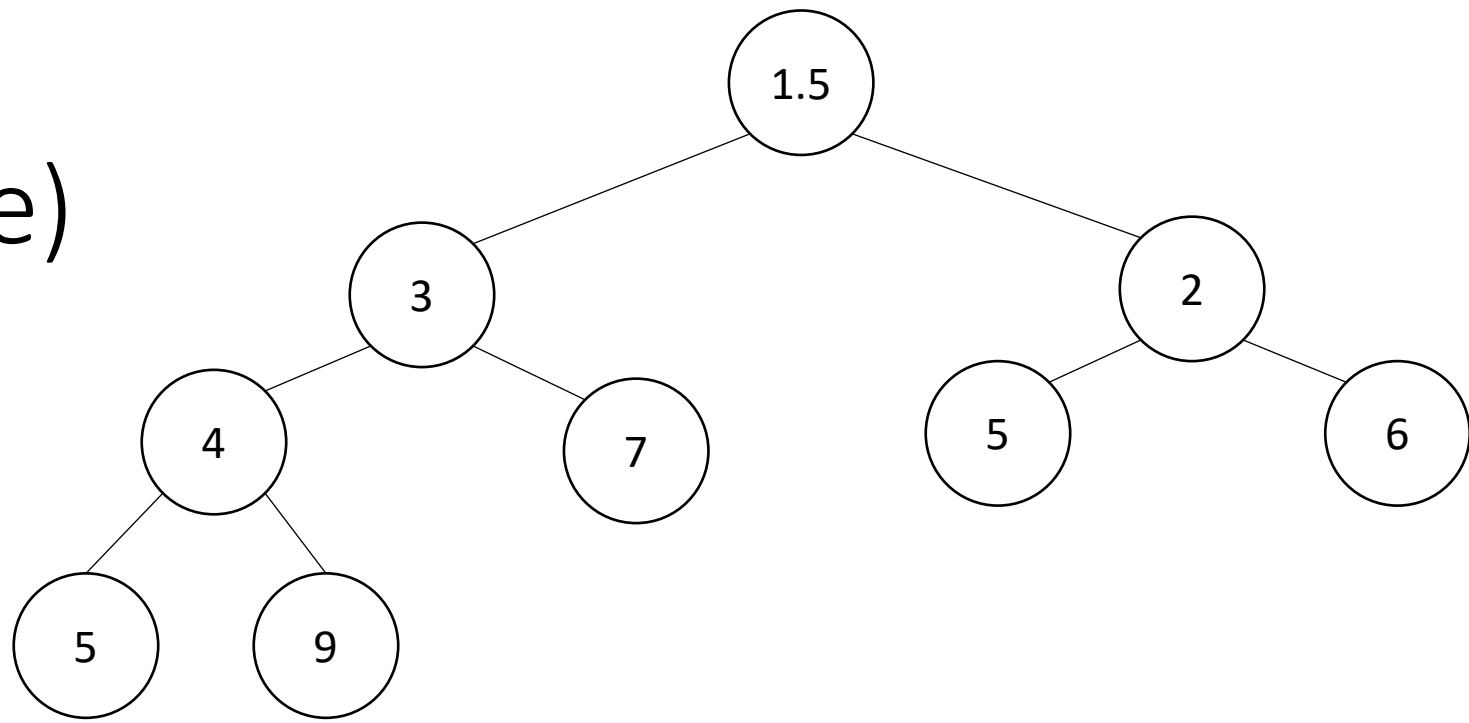
```
    return min
```

```
}
```

Percolate Down

Heap extract (Done)

```
extract(){  
  min = root  
  curr = bottom-right item  
  move curr to the root  
  while(curr > curr.left || curr > curr.right){  
    swap curr with its smallest child  
  }  
  return min  
}
```

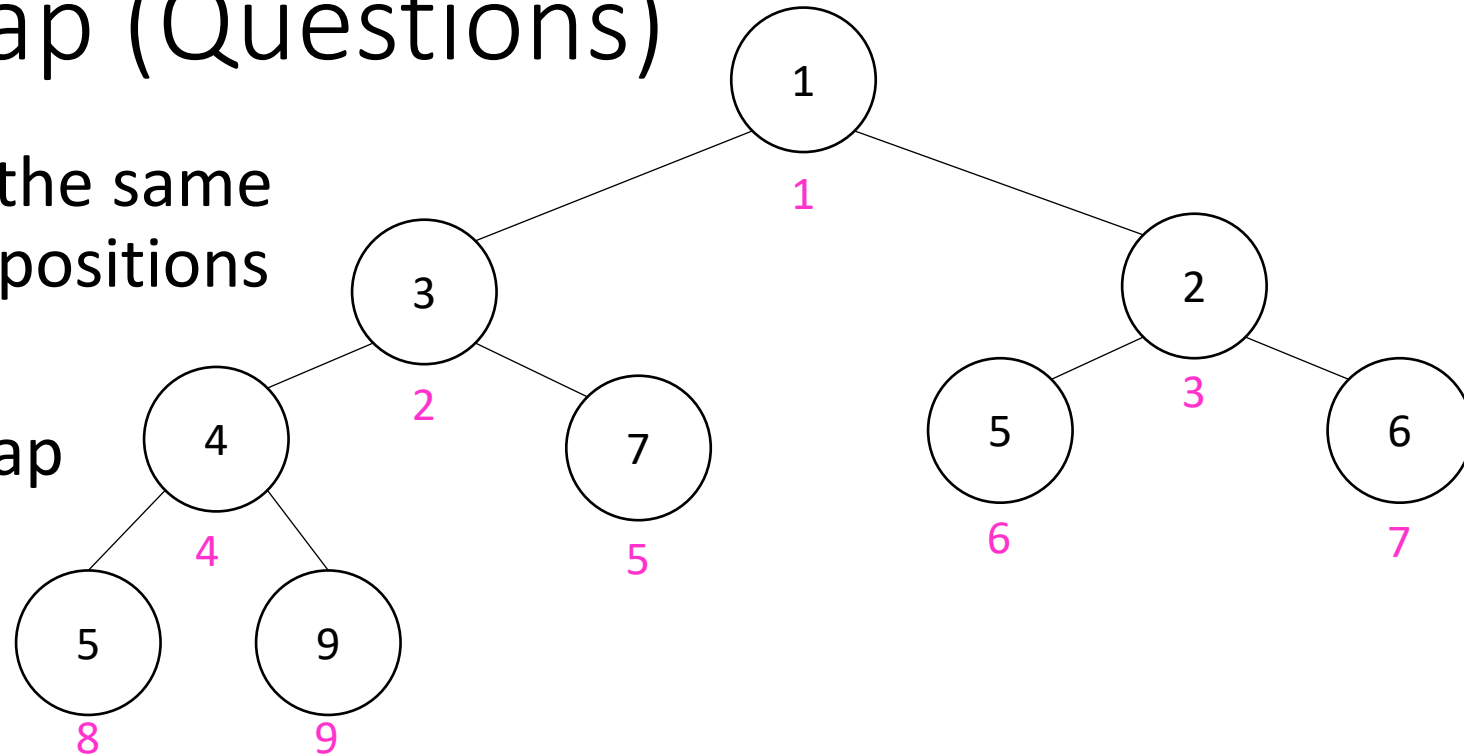


Percolate Up and Down (for a Min Heap)

- 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 (Questions)

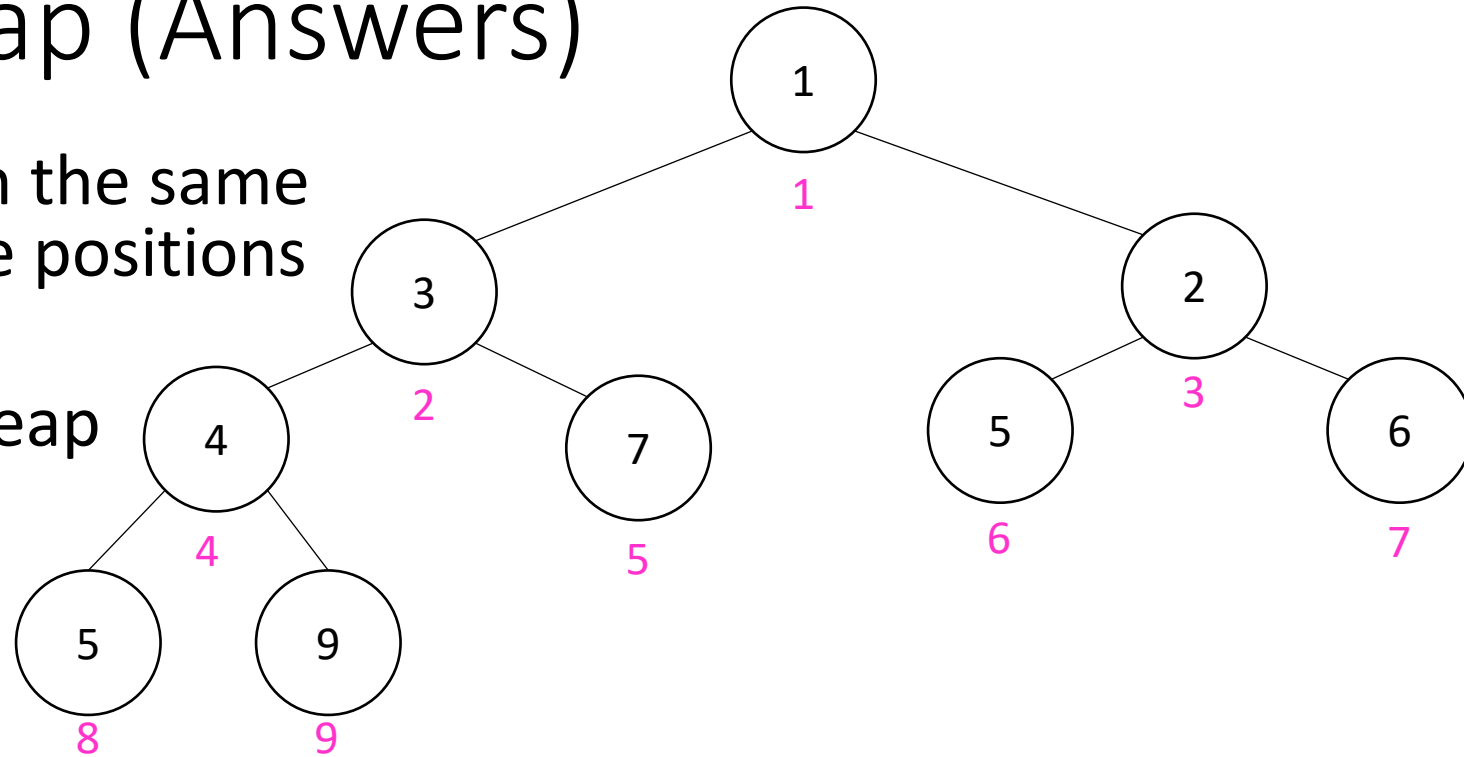
- Every complete binary tree with the same number of nodes uses the same positions and edges
- Use an array to represent the heap
- Index of root:
- Parent of node i :
- Left child of node i :
- Right child of node i :
- Location of the leaves:



	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

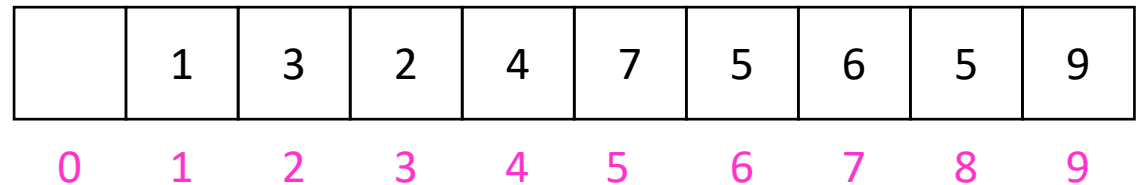
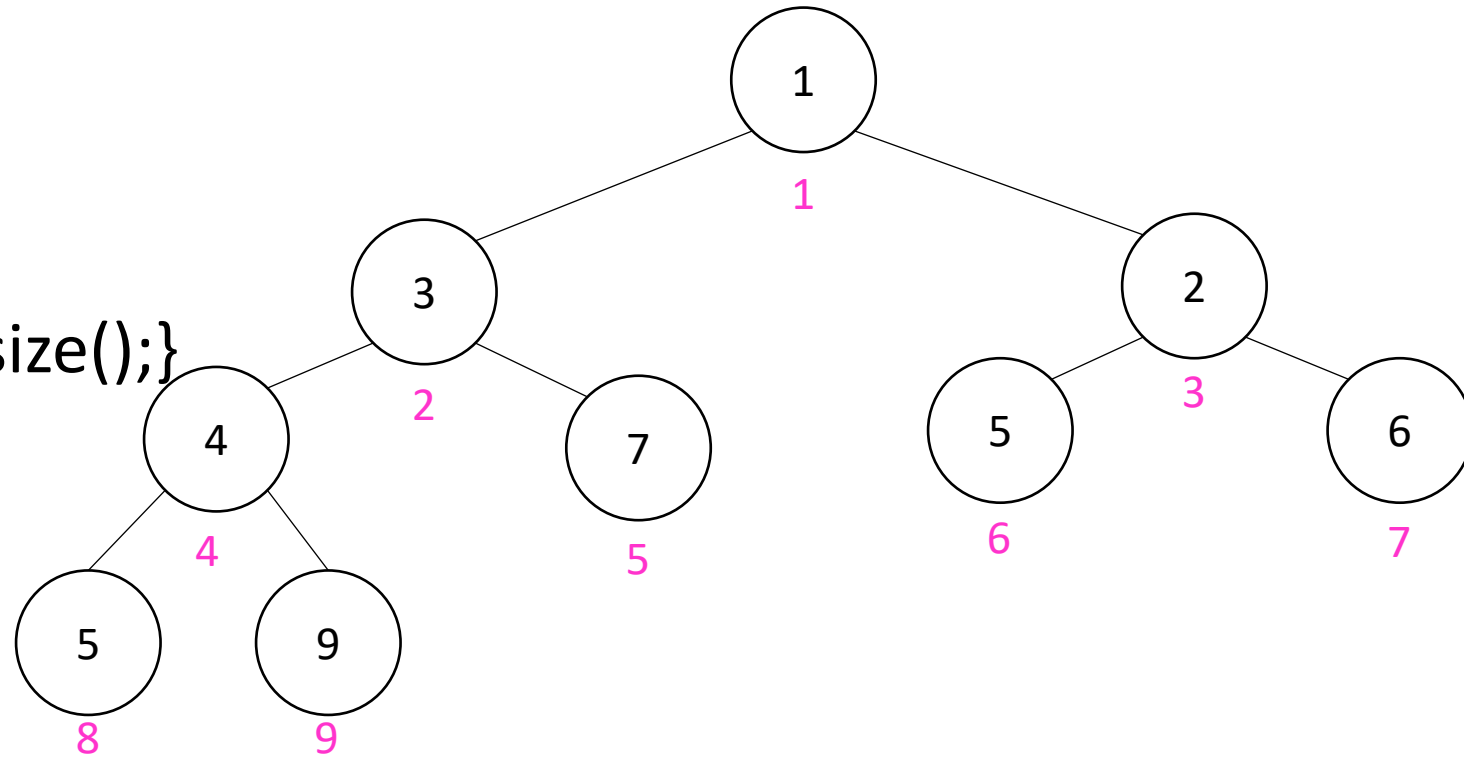


	1	3	2	4	7	5	6	5	9
0	1	2	3	4	5	6	7	8	9

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)  
}
```



Percolate Up

```
percolateUp(int i){  
    int parent = i/2; \\ index of parent  
    Item val = arr[i]; \\ value at current 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  
    }  
}
```

extract Pseudocode

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

Percolate Down

```
percolateDown(int i){
    int left = i*2; \\ index of left child
    int right = i*2+1; \\ index of right child
    Item val = arr[i]; \\ value at location
    while(left <= size){ \\ until location is leaf
        int 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{ return;} \\ if we don't swap, then heap property holds
    }
}
```

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