CSE 332 Autumn 2023
Lecture 5: Priority Queues

Nathan Brunelle

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

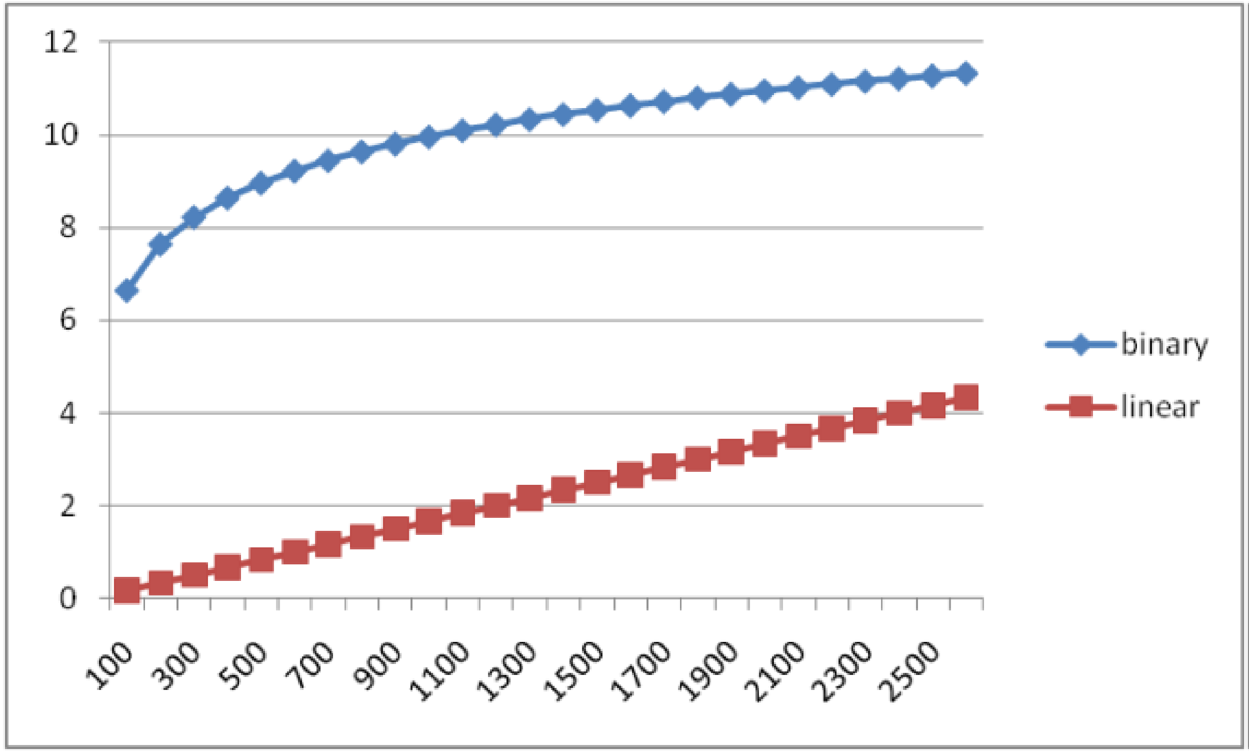
Goals for Algorithm Analysis

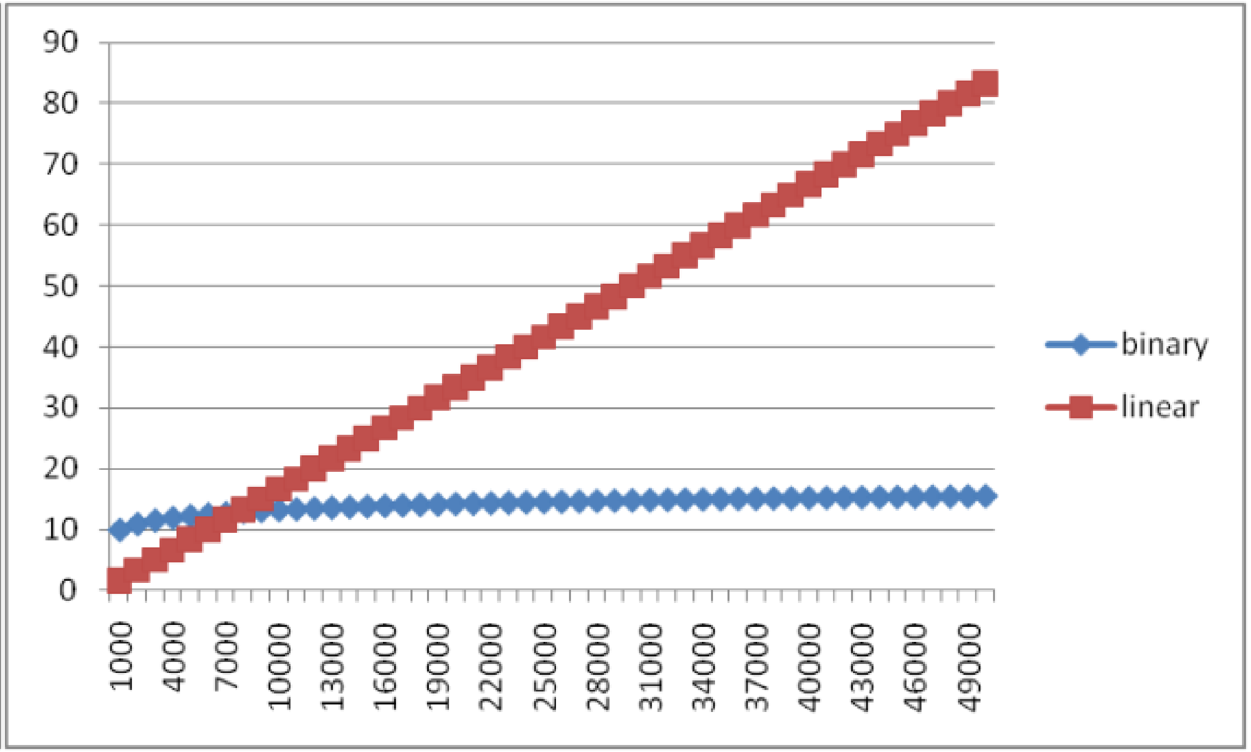
- Identify a *function* which maps the algorithm's input size to a measure of resources used
  - Domain of the function: **sizes** of the input
    - Number of characters in a string, number of items in a list, number of pixels in an image
  - Codomain of the function: **counts** of resources used
    - Number of times the algorithm adds two numbers together, number times the algorithm does a > or < comparison, maximum number of bytes of memory the algorithm uses at any time
- Important note: Make sure you know the "units" of your domain and codomain!
  - Domain = inputs to the function
  - Codomain = outputs to the function

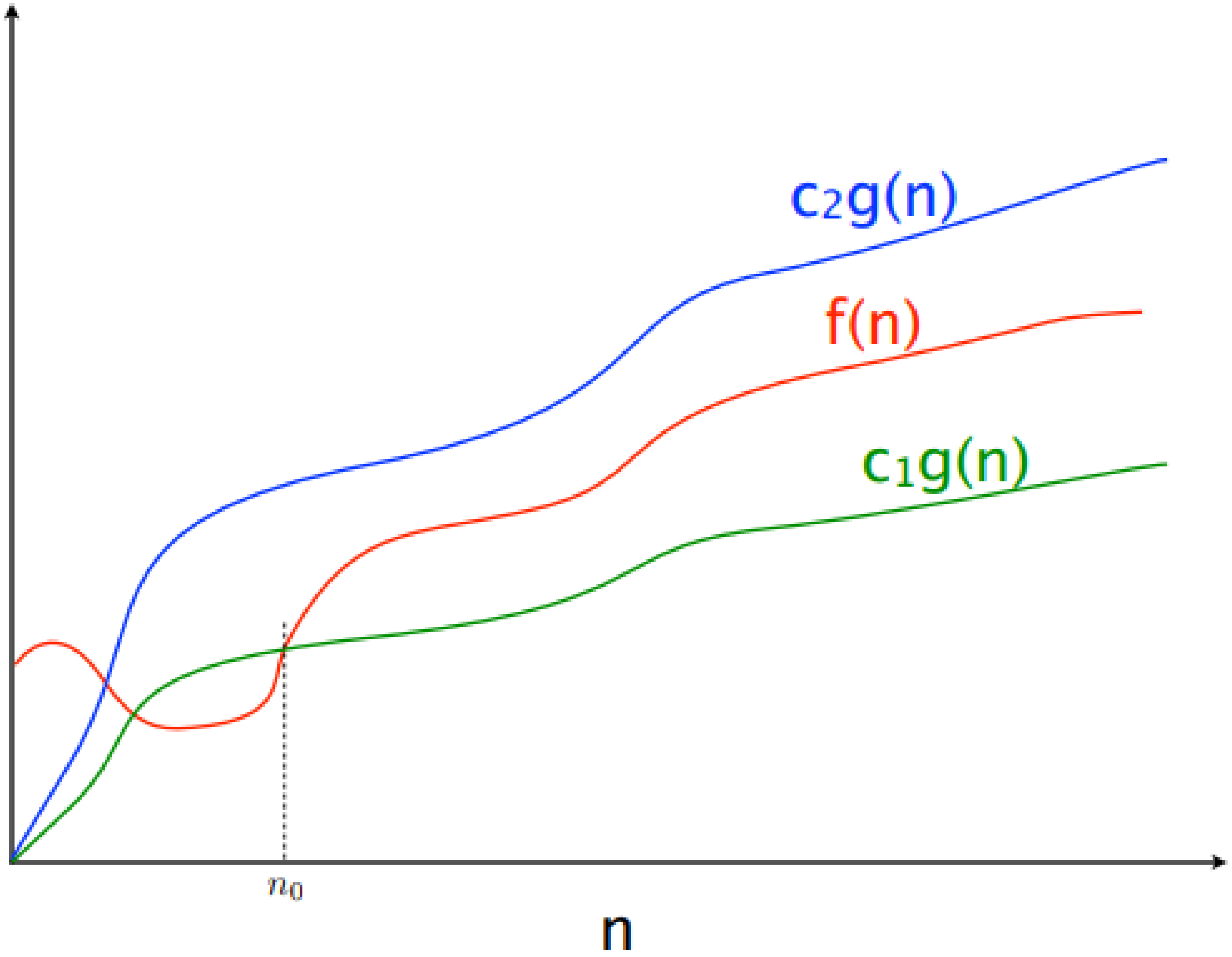
Worst Case Running Time Analysis

- If an algorithm has a worst case running time of $f(n)$
  - Among all possible size-n inputs, the “worst” one will do $f(n)$ “operations”
  - I.e. $f(n)$ gives the maximum operation count from among all inputs of size n

Comparing


12
10
8
6
4
2
0
100
300
500
700
900
1100
1300
1500
1700
1900
2100
2300
2500
binary
linear


90
80
70
60
50
40
30
20
10
0
1000
4000
7000
10000
13000
16000
19000
22000
25000
28000
31000
34000
37000
40000
43000
46000
49000
binary
linear



$$f(n) \in O(g(n))$$

$$f(n) \in \Theta(g(n))$$

$$f(n) \in \Omega(g(n))$$

Asymptotic Notation

- $O\big(g(n)\big)$
  - The **set of functions** with asymptotic behavior less than or equal to $g(n)$
  - Upper-bounded by a constant times g for large enough values n
  - $f \in O\big(g(n)\big) \equiv \exists c > 0. \exists n_0 > 0. \forall n \geq n_0. f(n) \leq c \cdot g(n)$
- $\Omega(g(n))$
  - the **set of functions** with asymptotic behavior greater than or equal to $g(n)$
  - Lower-bounded by a constant times g for large enough values n
  - $f \in \Omega\big(g(n)\big) \equiv \exists c > 0. \exists n_0 > 0. \forall n \geq n_0. f(n) \geq c \cdot g(n)$
- $\Theta\big(g(n)\big)$
  - "Tightly" within constant of g for large n
  - $\Omega\big(g(n)\big) \cap O(g(n))$

Asymptotic Notation Example

- Show: $10n + 100 \in O(n^2)$
  - **Technique:** find values $c > 0$ and $n_0 > 0$ such that $\forall n > n_0. 10n + 100 \leq c \cdot n^2$
  - **Proof:**

Asymptotic Notation Example

- Show: $10n + 100 \in O(n^2)$
  - **Technique:** find values $c > 0$ and $n_0 > 0$ such that $\forall n \geq n_0.\, 10n + 100 \leq c \cdot n^2$
  - **Proof:** Let $c = 10$ and $n_0 = 6$. Show $\forall n \geq 6. 10n + 100 \leq 10n^2$

    $10n + 100 \leq 10n^2$

    $\equiv n + 10 \leq n^2$

    $\equiv 10 \leq n^2 - n$

    $\equiv 10 \leq n(n-1)$

    This is True because $n(n-1)$ is strictly increasing and $6(6-1) > 10$

Asymptotic Notation Example

- Show: $13\mathrm{n}^2 - 50\mathrm{n} \in \Omega(n^2)$
  - **Technique:** find values $c > 0$ and $n_0 > 0$ such that $\forall n \geq n_0.\, 13n^2 - 50n \geq c \cdot n^2$
  - **Proof:**

Asymptotic Notation Example

- Show: $13\mathrm{n}^2 - 50\mathrm{n} \in \Omega(n^2)$
  - **Technique:** find values $c > 0$ and $n_0 > 0$ such that $\forall n \geq n_0.\, 13n^2 - 50n \geq c \cdot n^2$
  - **Proof:** let $c = 12$ and $n_0 = 50$. Show $\forall n \geq 50.\, 13n^2 - 50n \geq 12n^2$

$$13n^2 - 50n \geq 12n^2$$
$$\equiv n^2 - 50n \geq 0$$
$$\equiv n^2 \geq 50n$$
$$\equiv n \geq 50$$

This is certainly true $\forall n \geq 50$.

Worst Case Running Time - Example

```
myFunction(List n){
    b = 55 + 5;
    c = b / 3;
    b = c + 100;
    for (i = 0; i < n.size(); i++) {
        b++;
    }
    if (b % 2 == 0) {
        c++;
    }
    else {
        for (i = 0; i < n.size(); i++) {
            c++;
        }
    }
    return c;
}
```

Questions to ask:

- What are the units of the input size?
- What are the operations we’re counting?
- For each line:
  - How many times will it run?
  - How long does it take to run?
  - Does this change with the input size?

Worst Case Running Time – Example 2

```
beAnnoying(List n){
    List m = [];
    for (i=0; i < n.size(); i++){
        m.add(n[i]);
        for (j=0; j< n.size(); j++){
            print ("Hi, I'm annoying");
        }
    }
    return;
}
```

Questions to ask:

- What are the units of the input size?
- What are the operations we're counting?
- For each line:
    - How many times will it run?
    - How long does it take to run?
    - Does this change with the input size?

Gaining Intuition

- When doing asymptotic analysis of functions:
  - If multiple expressions are added together, ignore all but the "biggest"
    - If $f(n)$ grows asymptotically faster than $g(n)$, then $f(n) + g(n) \in \Theta(f(n))$
  - Ignore all multiplicative constants
    - $f(n) + c \in \Theta(f(n))$ for any constant $c \in \mathbb{R}$
  - Ignore bases of logarithms
  - Do NOT ignore:
    - Non-multiplicative and non-additive constants (e.g. in exponents, bases of exponents)
    - Logarithms themselves

More Examples

- Is each of the following True or False?
  - $4 + 3n \in O(n)$
  - $n + 2\log n \in O(\log n)$
  - $\log n + 2 \in O(1)$
  - $n^{50} \in O(1.1^n)$
  - $3^n \in \Theta(2^n)$

Common Categories

- $O(1)$ “constant”
- $O(\log n)$ “logarithmic”
- $O(n)$ “linear”
- $O(n \log n)$ “log-linear”
- $O(n^2)$ “quadratic”
- $O(n^3)$ “cubic”
- $O(n^k)$ “polynomial”
- $O(k^n)$ “exponential”

Defining your running time function

- Worst-case complexity:
  - max number of steps algorithm takes on “most challenging” input
- Best-case complexity:
  - min number of steps algorithm takes on “easiest” input
- Average/expected complexity:
  - avg number of steps algorithm takes on random inputs (context-dependent)
- Amortized complexity:
  - max total number of steps algorithm takes on M “most challenging” consecutive inputs, divided by M (i.e., divide the max total sum by M).

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
  - 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)

Priority Queue, example

```
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.deleteMin)
Print(PQ.deleteMin)
Print(PQ.deleteMin)
Print(PQ.deleteMin)
Print(PQ.deleteMin)
```

Priority Queue, example

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

Applications?

Thinking through implementations

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

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

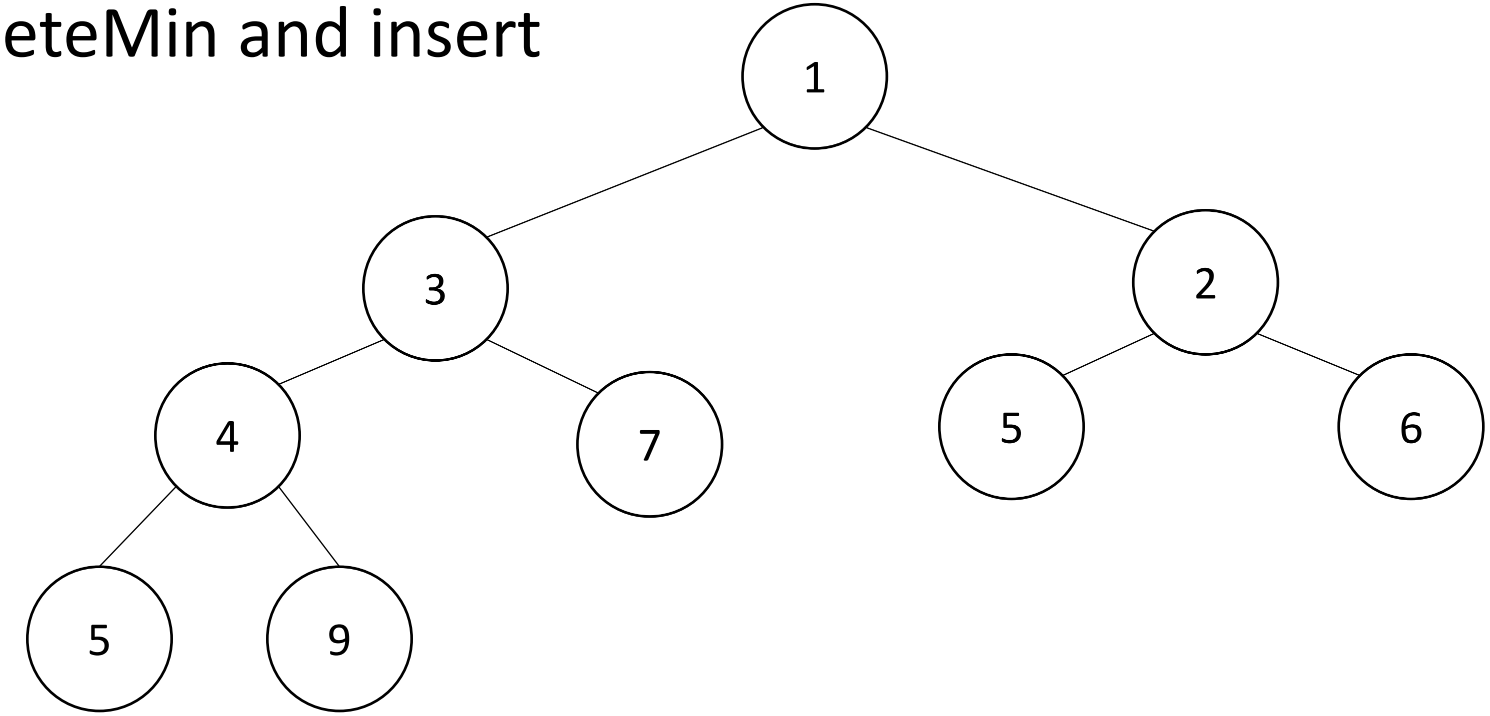
Thinking through implementations

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	$\Theta(\log n)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(1)$
Binary Search Tree	$\Theta(n)$	$\Theta(1)$

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

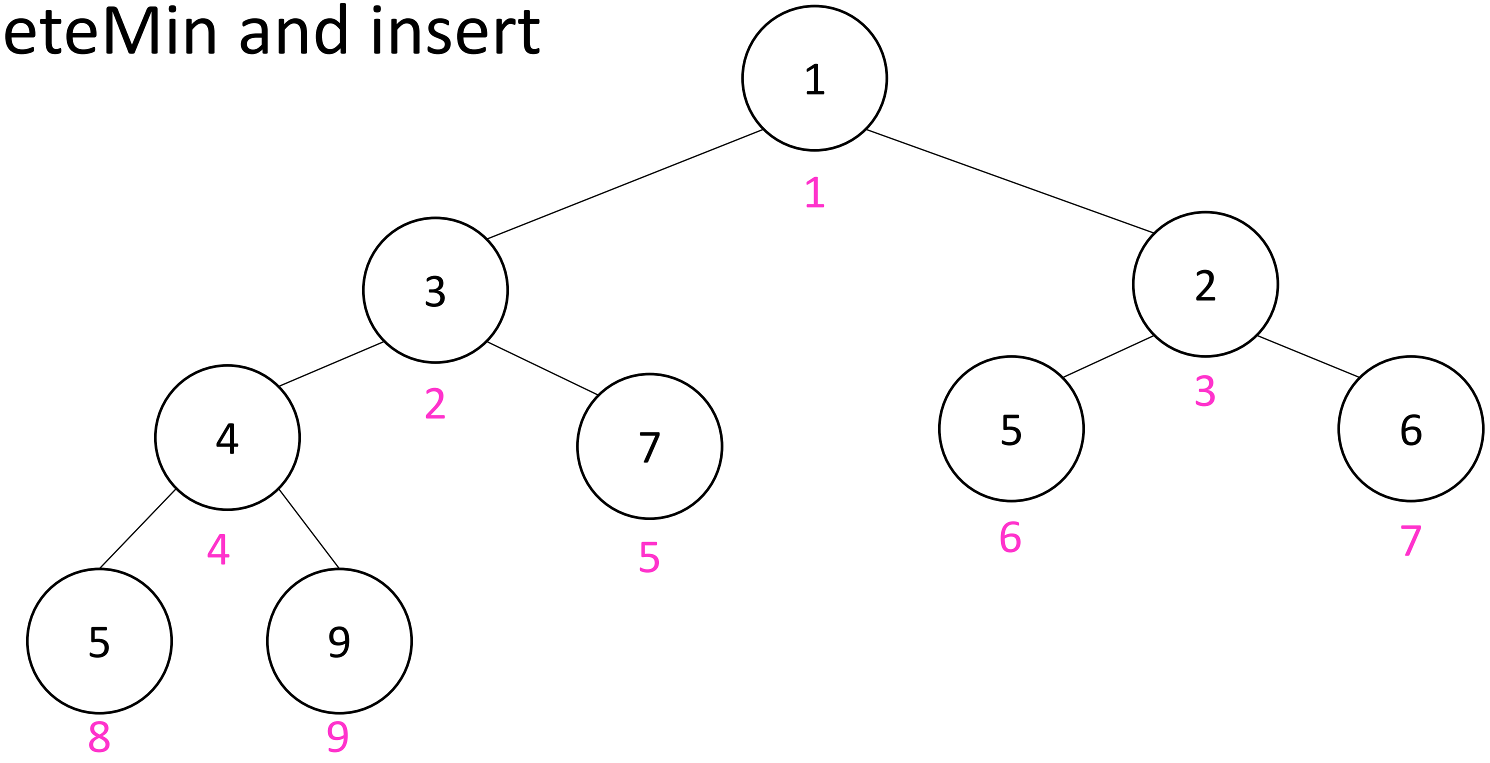
Heap – Priority Queue Data Structure

- 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



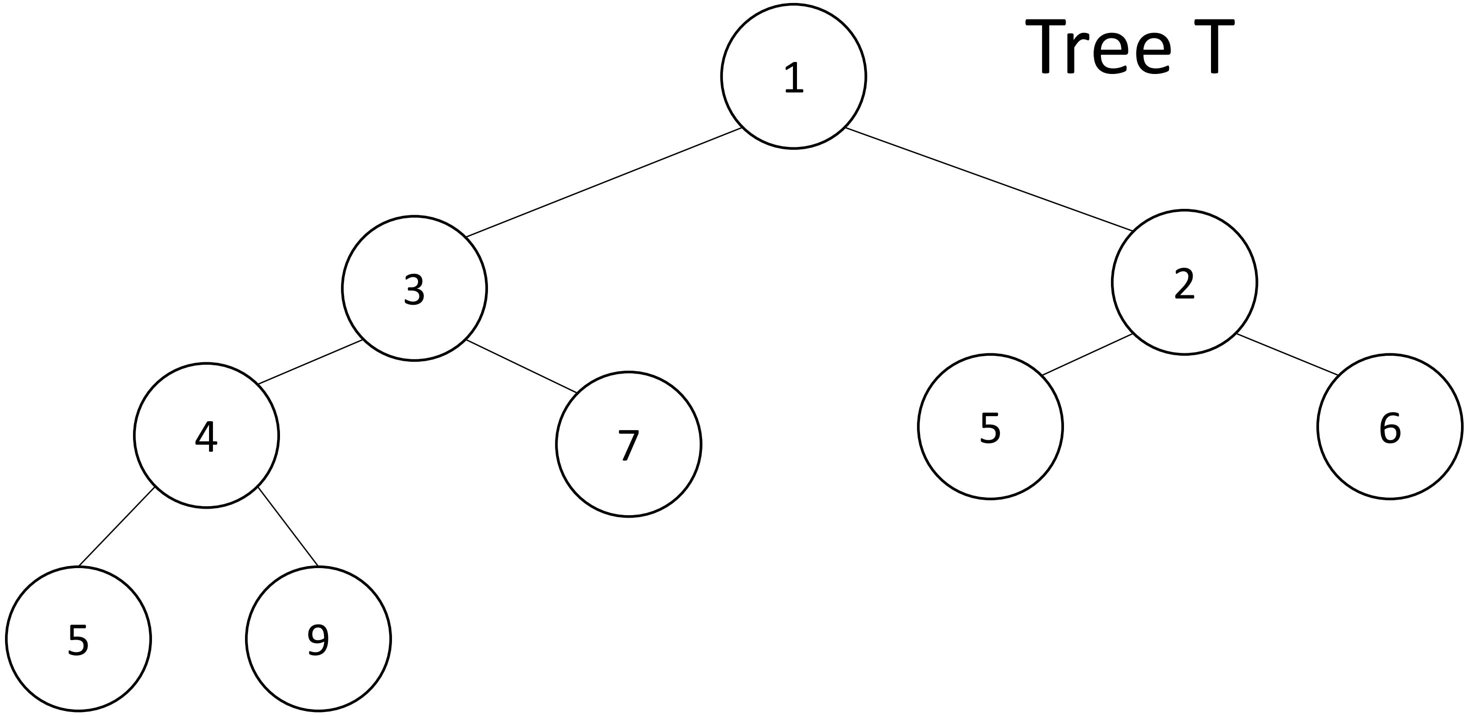
Heap – Priority Queue Data Structure

- 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



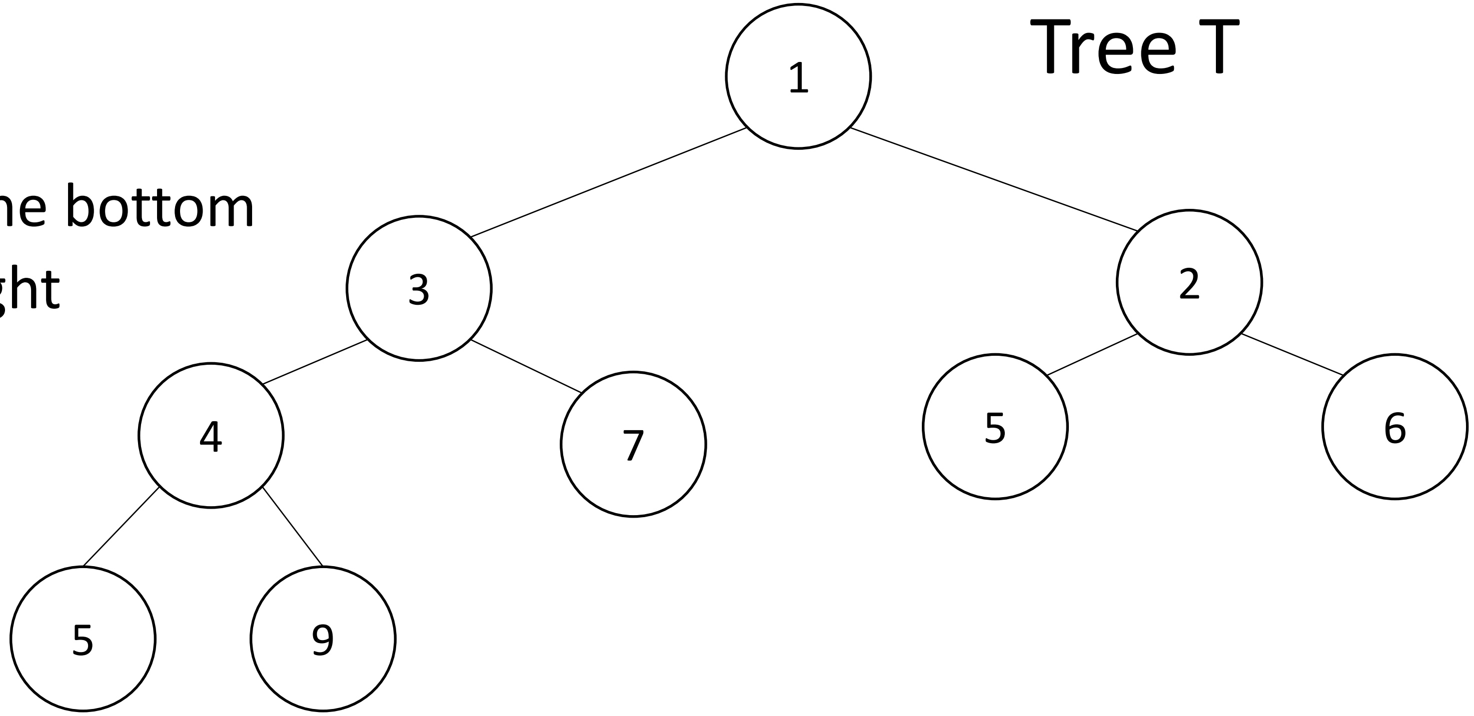
Tree Terminology – Review?

- root(T):
- leaves(T):
- children(3):
- parent(4):
- siblings(7):
- ancestors(9):
- descendents(3):
- subtree(4):
- height(T):
- depth(4):
- branchingFactor(T):



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



Challenge!

- What is the maximum number of total nodes in a binary tree of height h?
- If I have n nodes in a binary tree, what is the its minimum height?

Challenge!

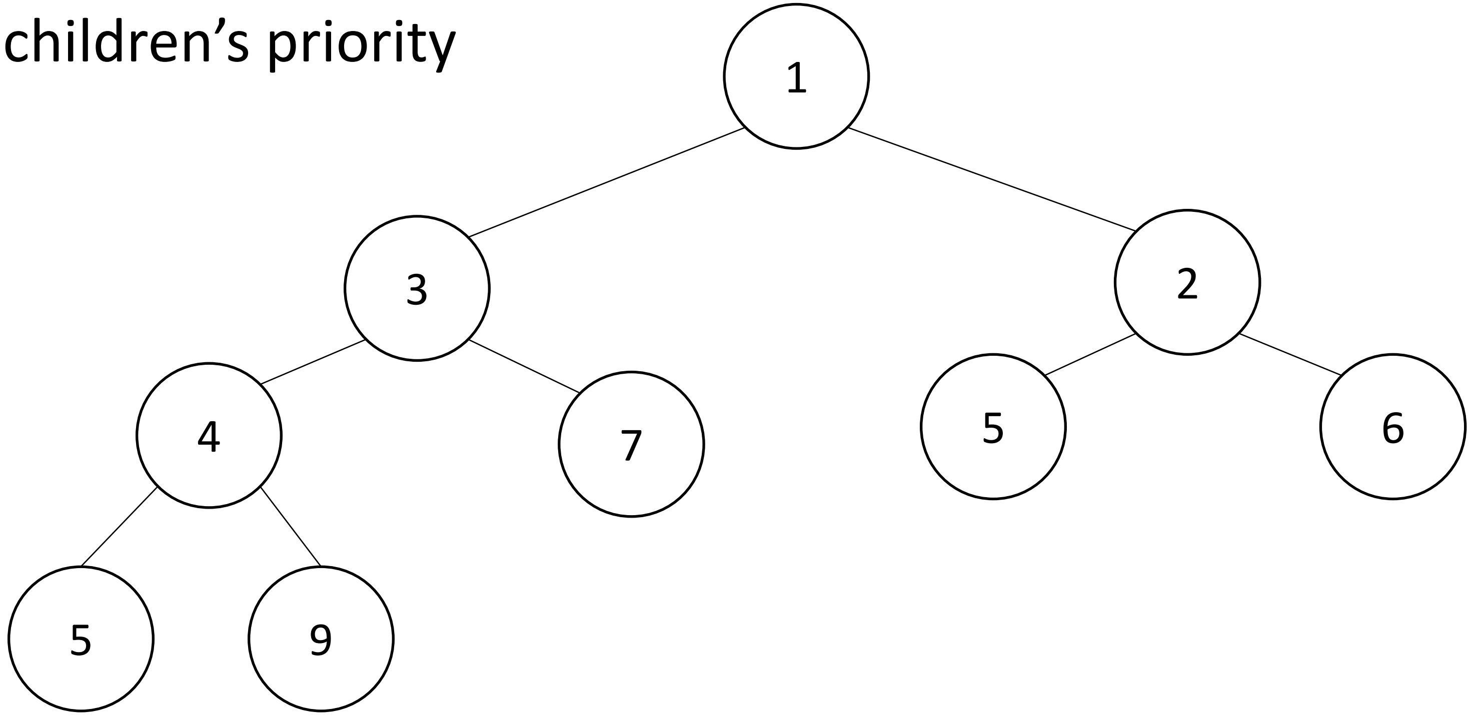
- What is the maximum number of total nodes in a binary tree of height h?
  - $2^{h+1} - 1$
  - $\Theta(2^h)$
- If I have n nodes in a binary tree, what is its minimum height?
  - $\lceil \log_2 n \rceil$
  - $\Theta(\log n)$

- 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

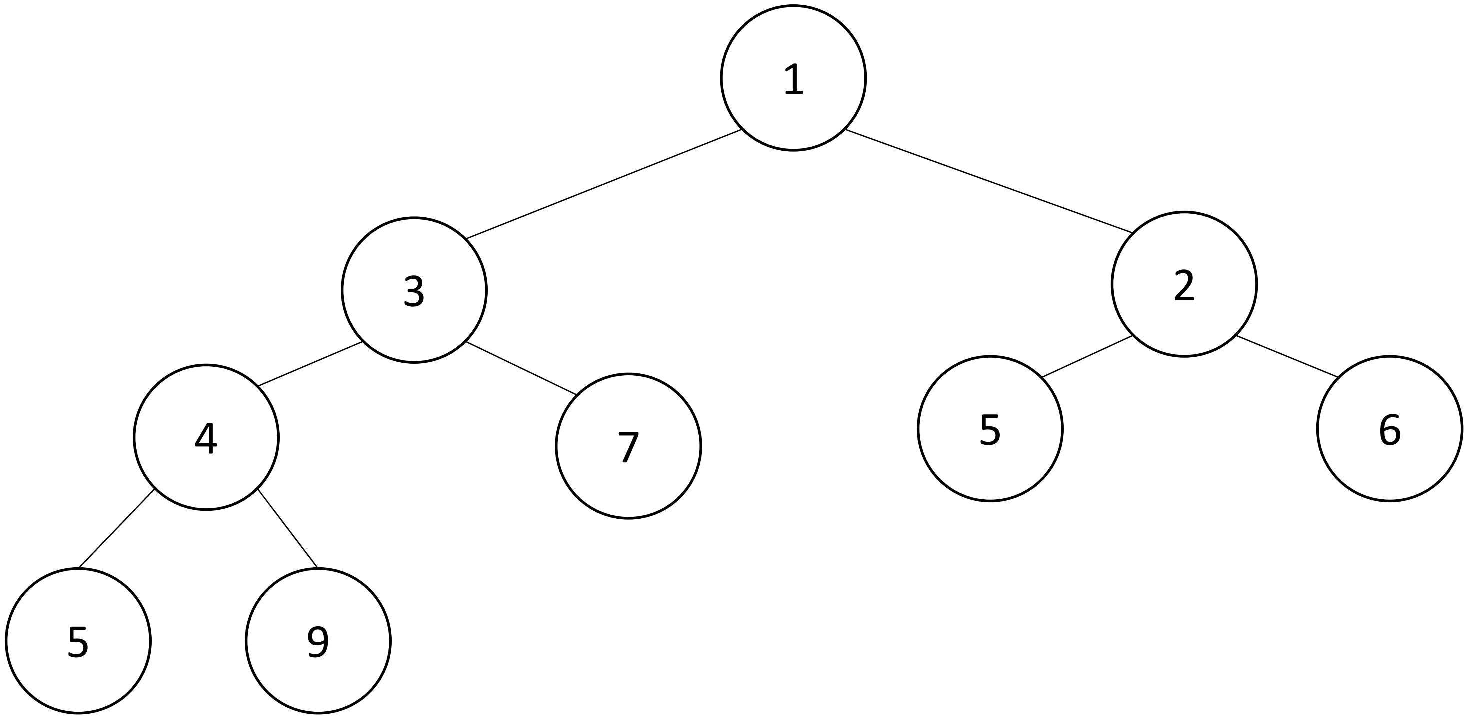
Heap Data Structure

- 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?
- How do I insert?
- How do I deleteMin?
- How to do it in Java?

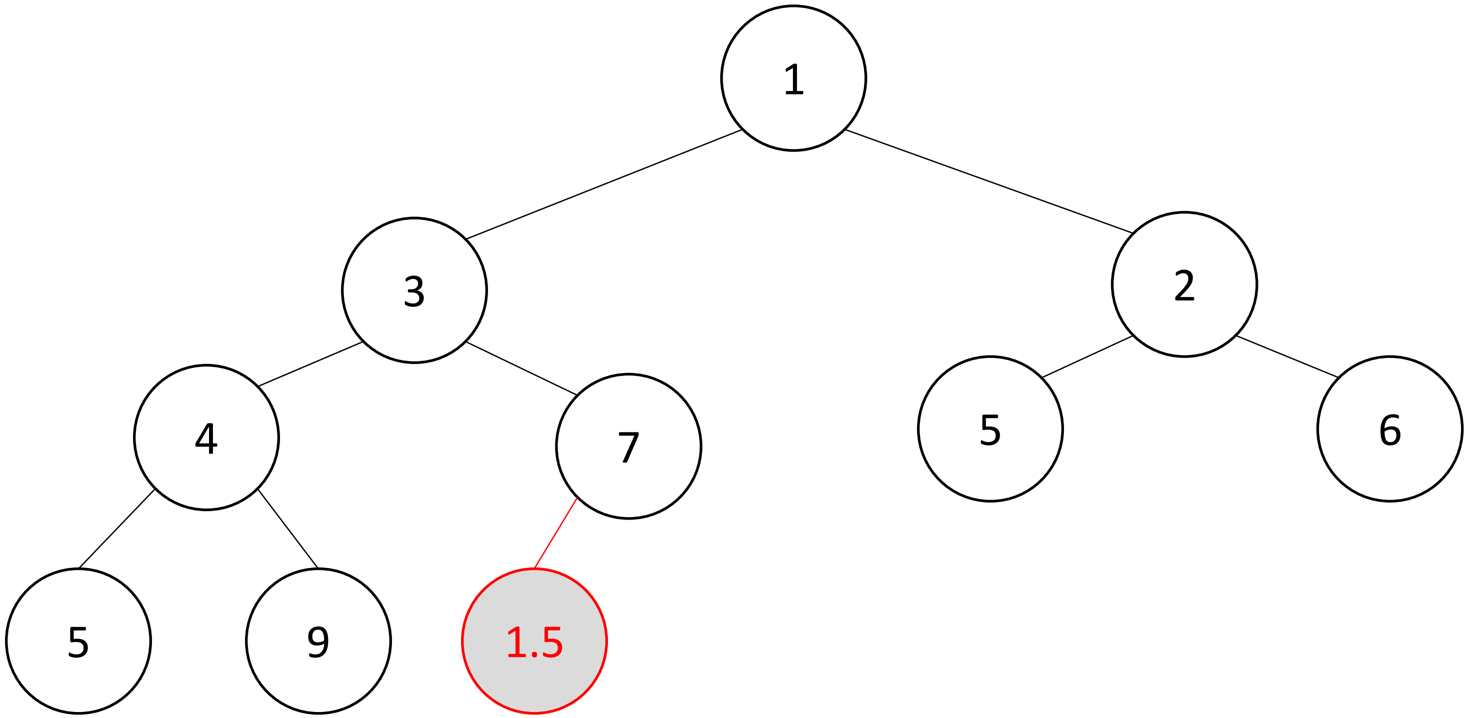


Heap Insert



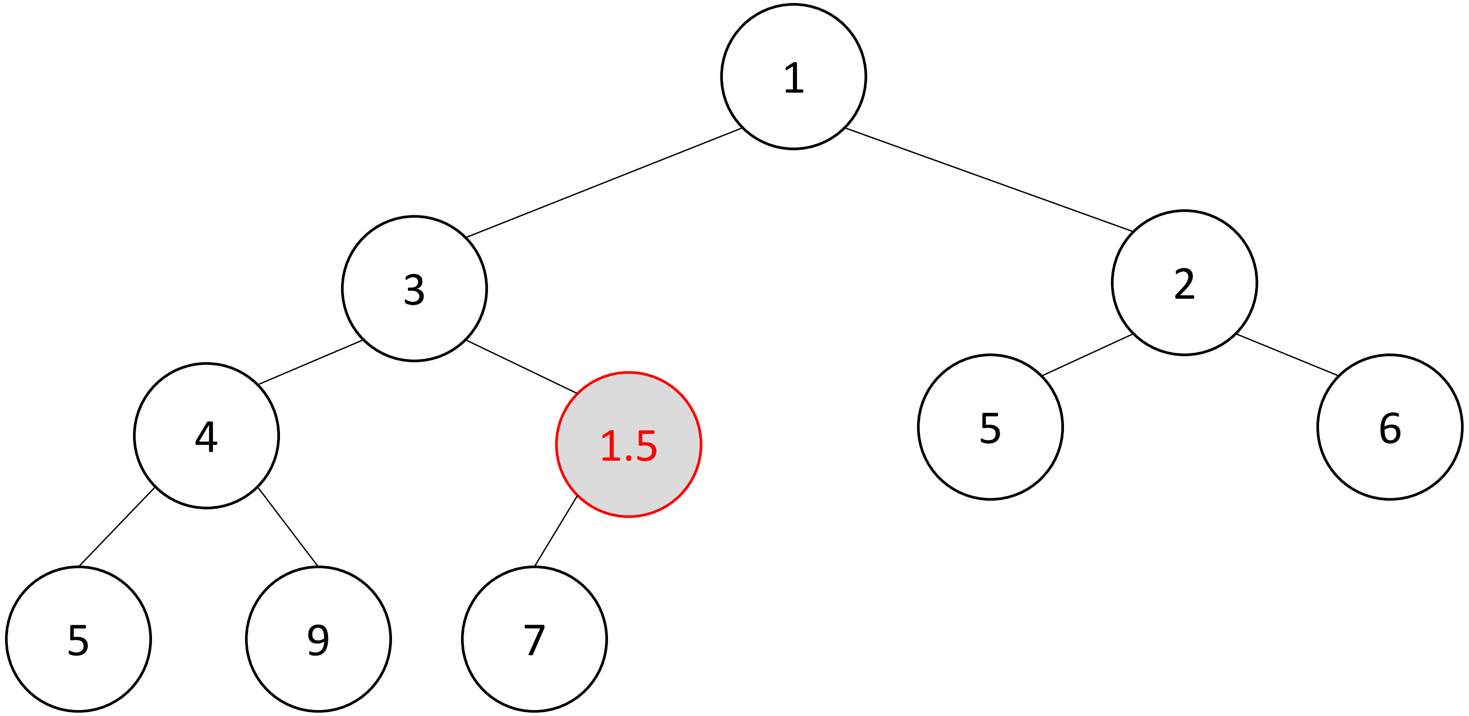

```
insert(item){
    put item in the "next open" spot (keep tree complete)
    while (item.priority < parent(item).priority){
        swap item with parent
    }
}
```

Heap Insert




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

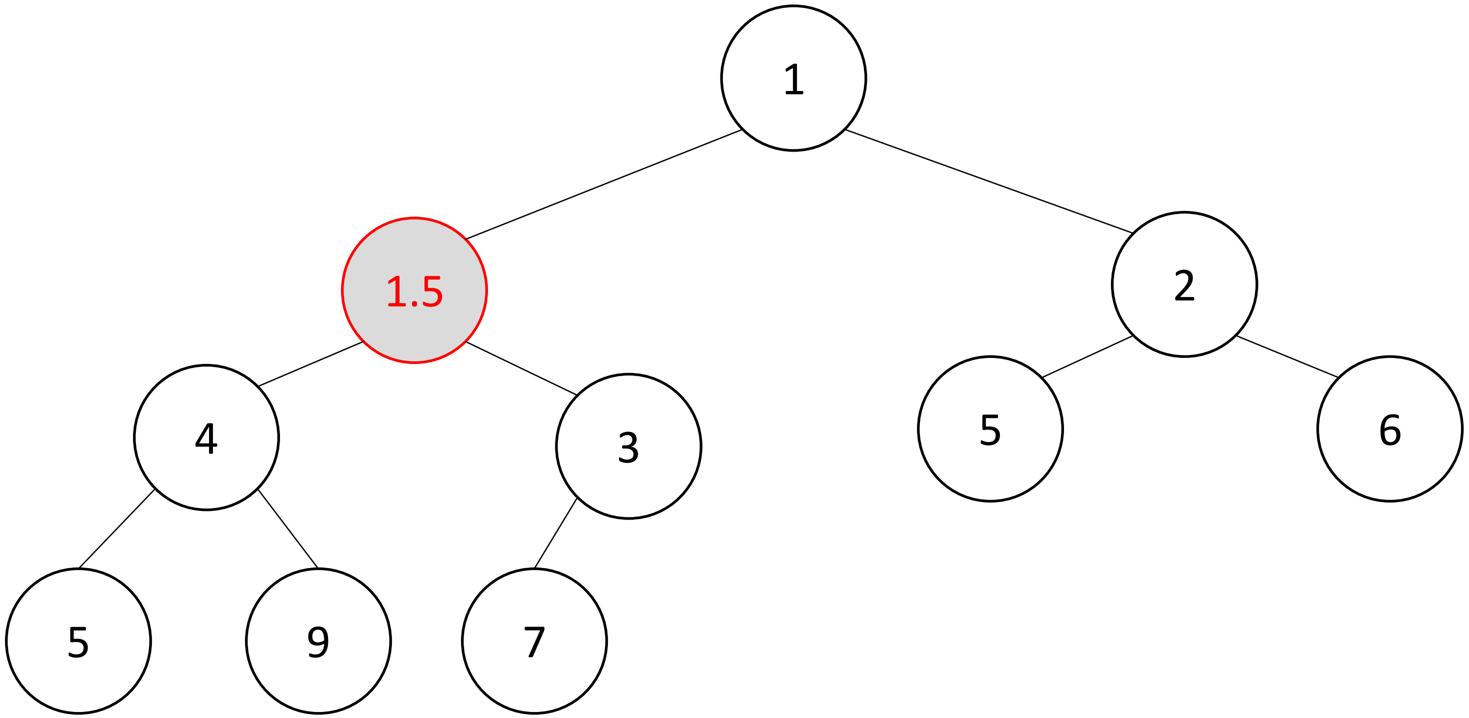
Heap Insert




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

Percolate Up

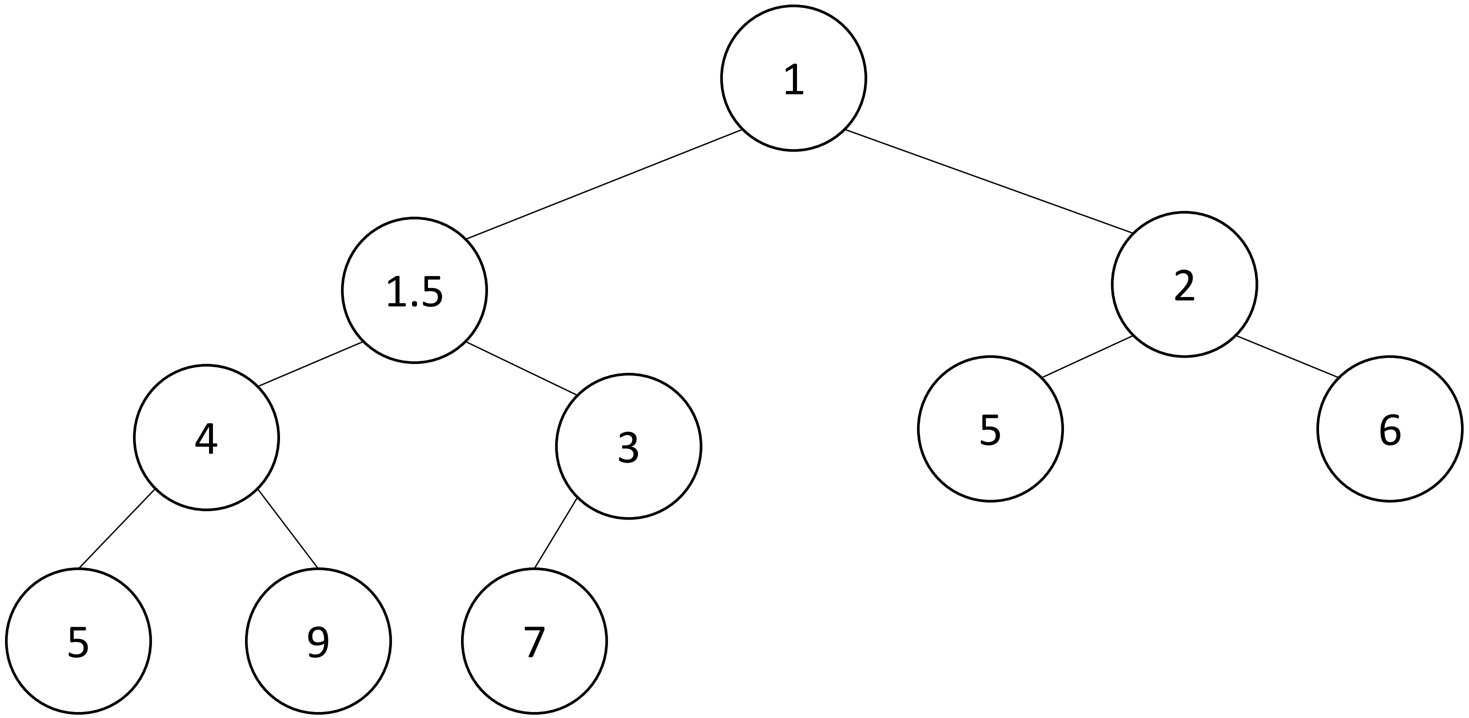
Heap Insert




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

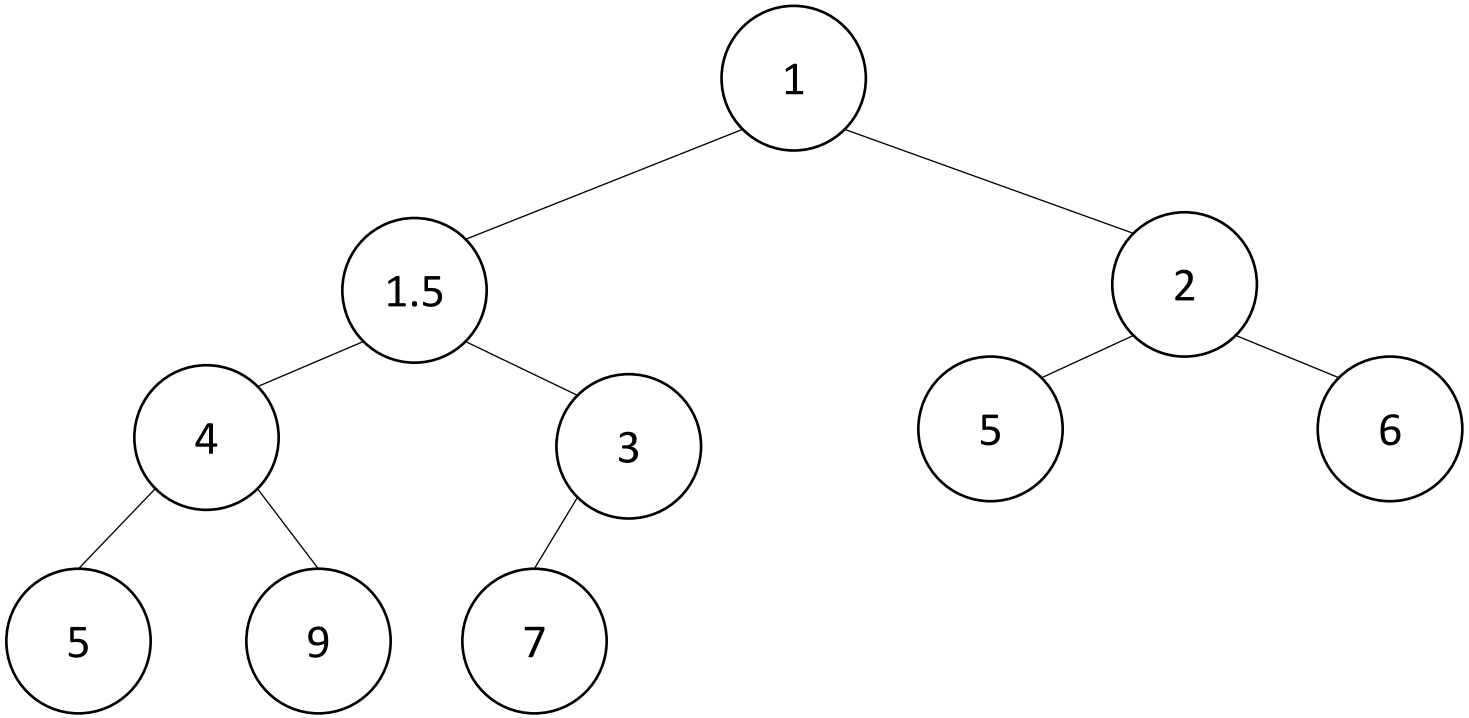
Percolate Up

Heap Insert



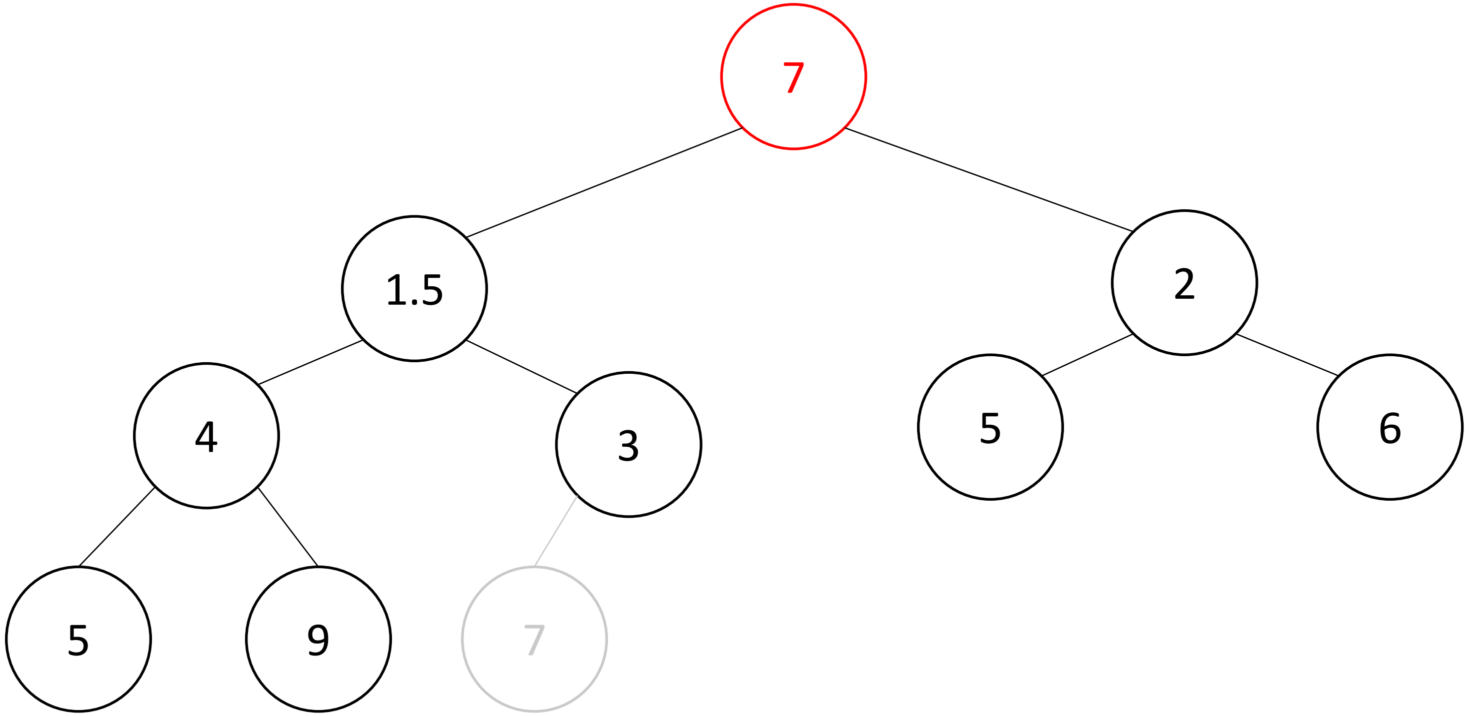

```
insert(item){
   put item in the "next open" spot (keep tree complete)
   while (item.priority < parent(item).priority){
      swap item with parent
   }
}
```

Heap deleteMin



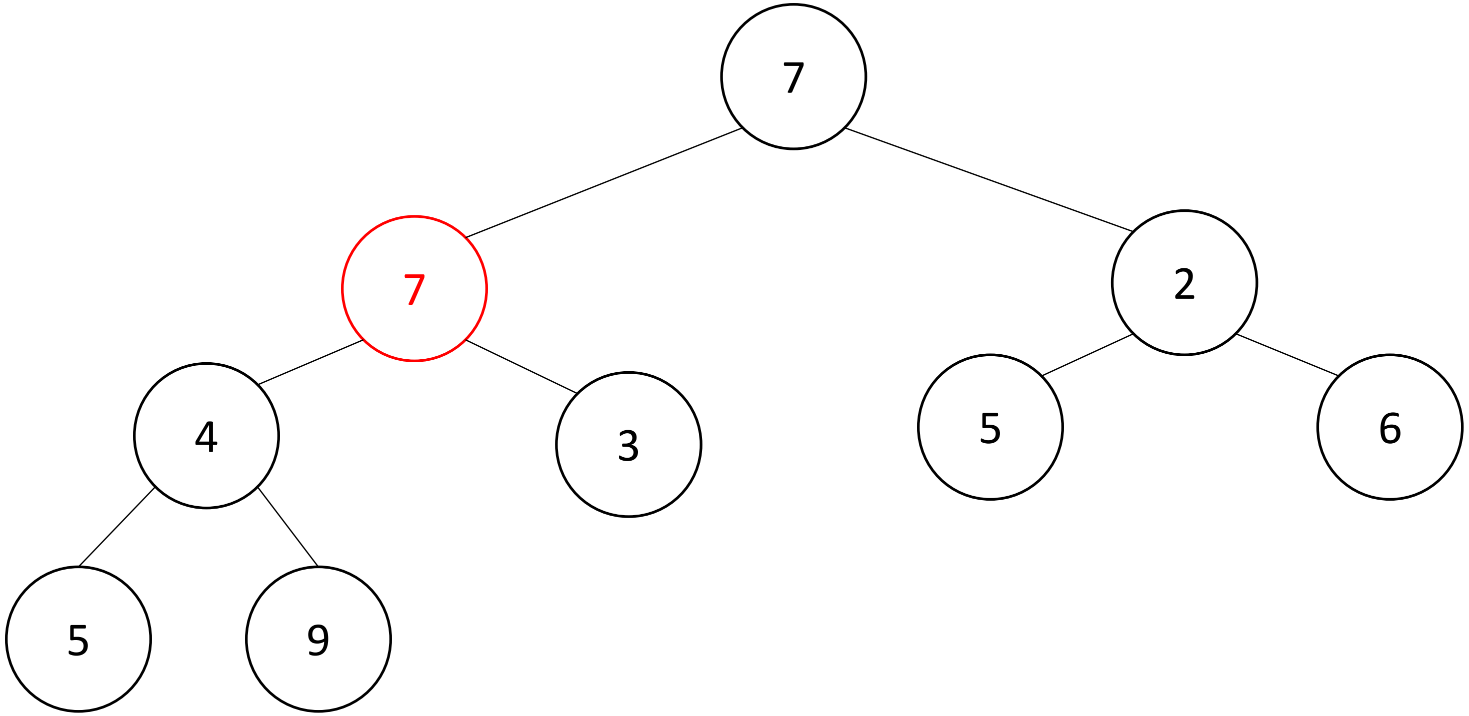

```
deleteMin(){
    min = root
    br = bottom-right item
    move br to the root
    while(br > either of its children){
        swap br with its smallest child
    }
    return min
}
```

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
}
```

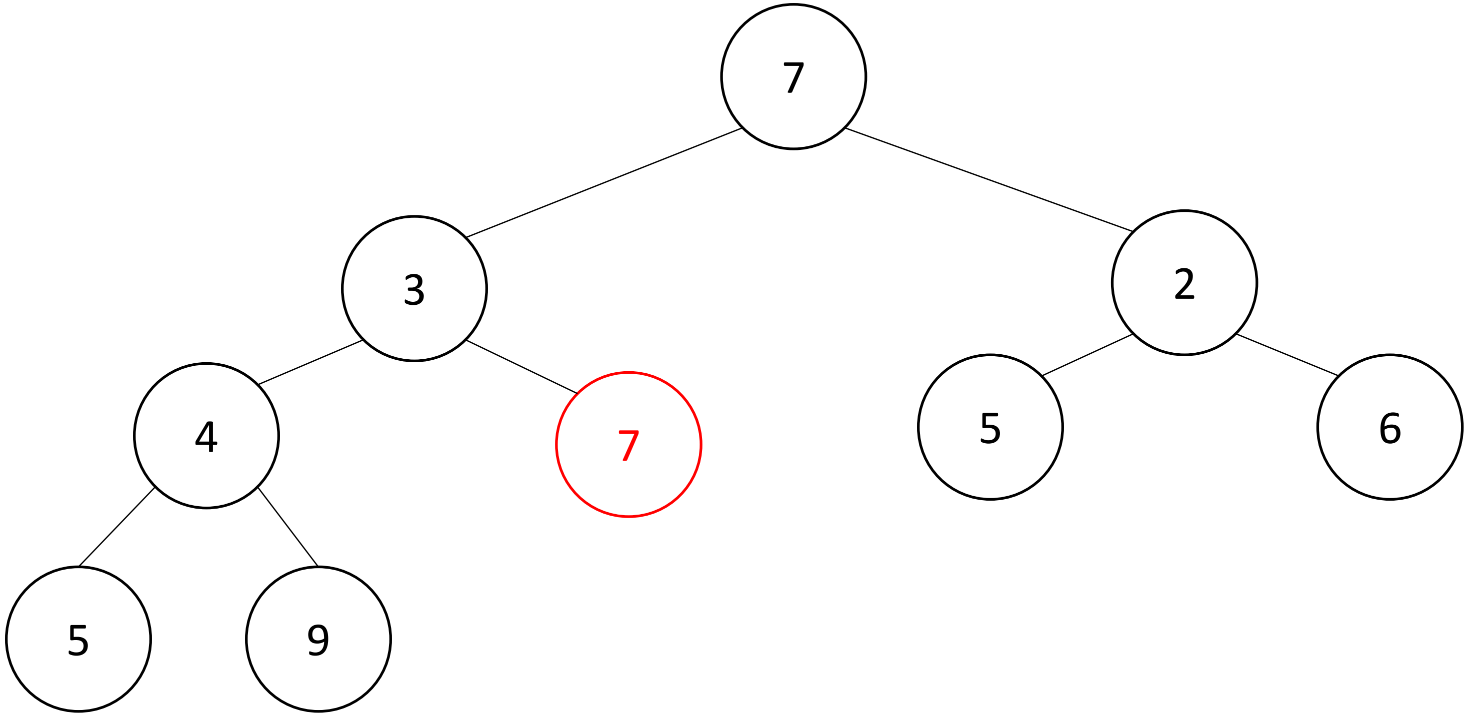
Heap deleteMin




```
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

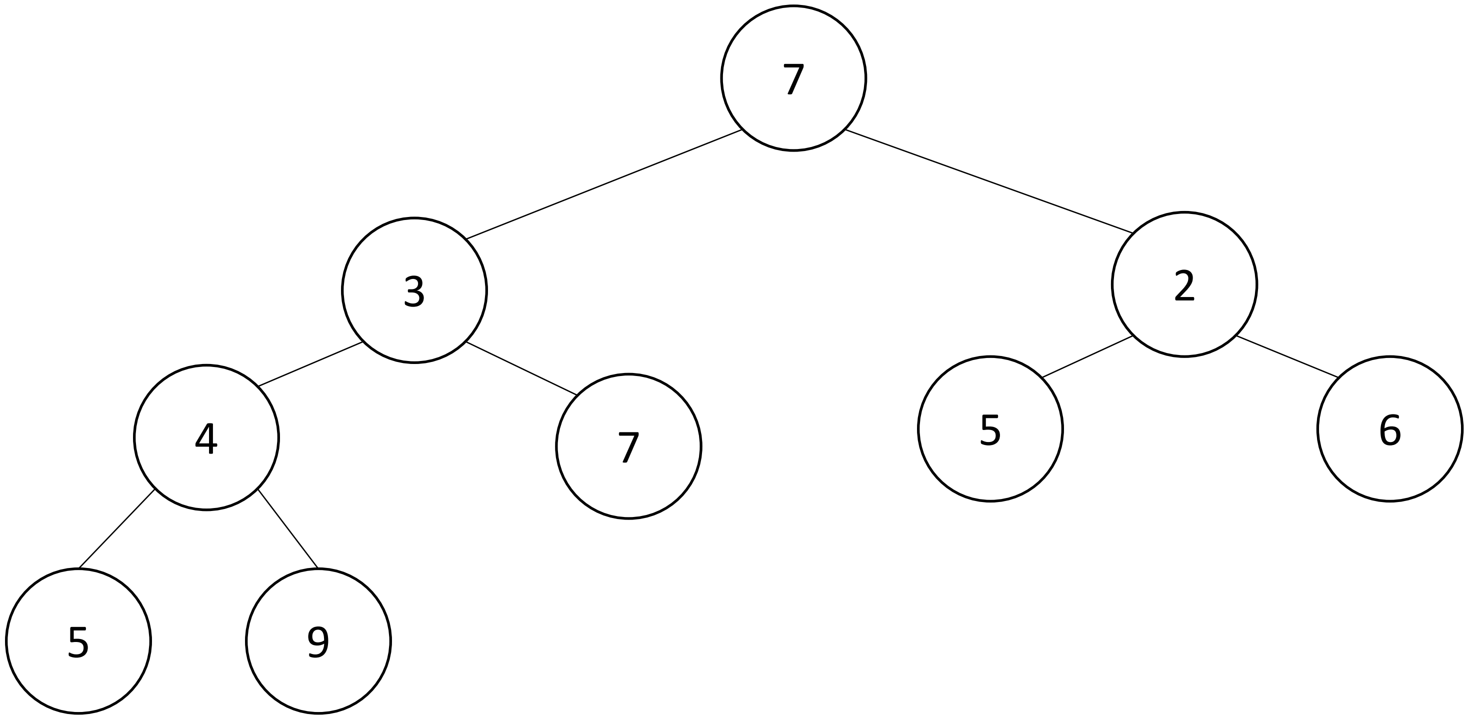
Heap deleteMin




```
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
}
```