CSE 332 Autumn 2024
Lecture 6: Priority Queues

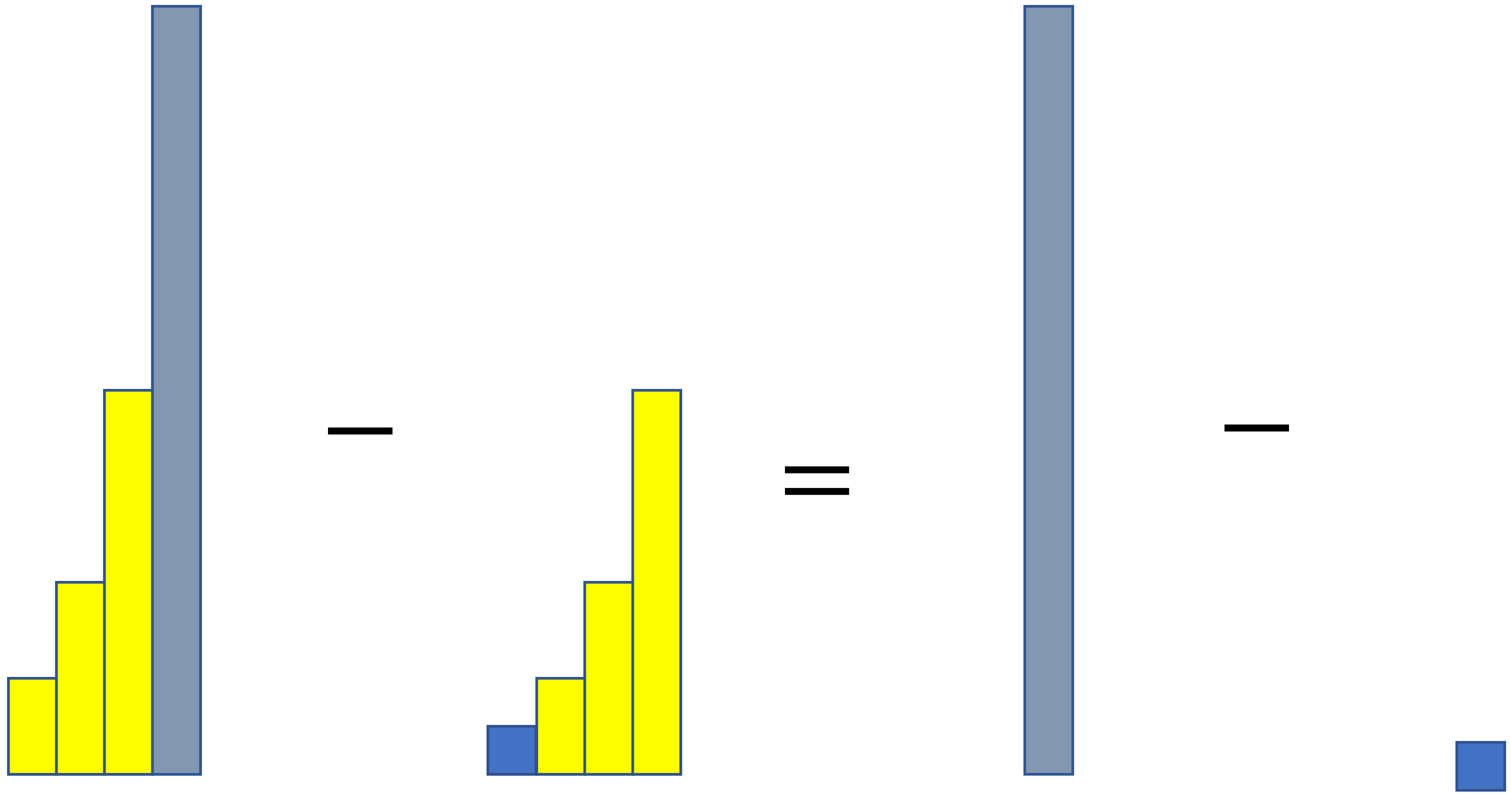
Nathan Brunelle

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

Finite Geometric Series

$$\sum_{i=0}^{L} a^i$$

If $a > 1$



The series multiplied by a

$(1 + a + a^2 + \cdots + a^L)a$

The series

$(1 + a + a^2 + \cdots + a^L)1$

The next term in the series

a^{L+1}

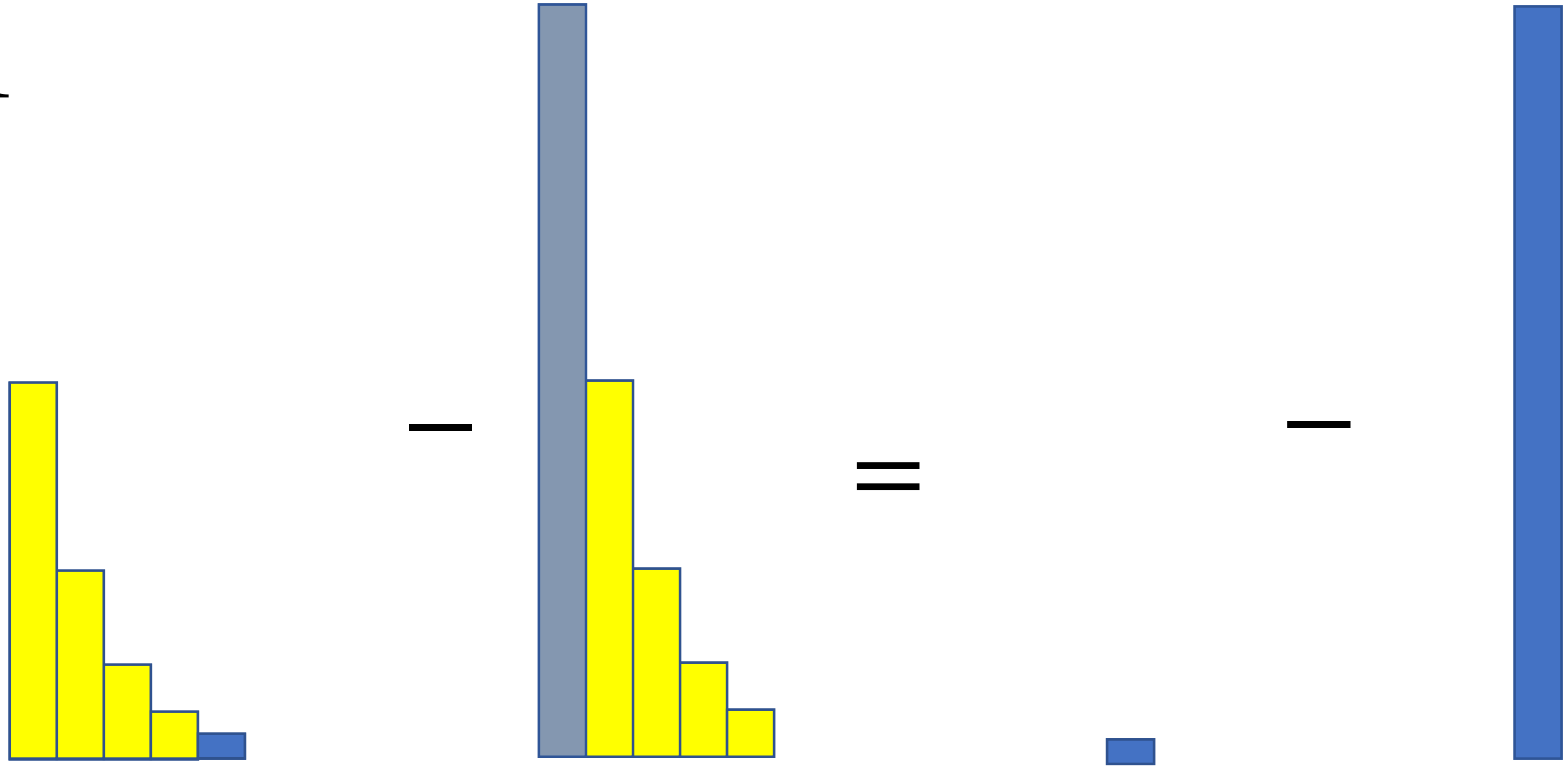
The first term

1

$$\sum_{i=0}^{L} a^i$$

Finite Geometric Series

If $a < 1$



The series multiplied by a | The series | The next term in the series | The first term

$$\underbrace{(1 + a + a^2 + \cdots + a^L)a \quad - \quad (1 + a + a^2 + \cdots + a^L)1 \quad = \quad a^{L+1} \quad - \quad 1}_{\text{Solve for the series}}$$

Let’s do some more!

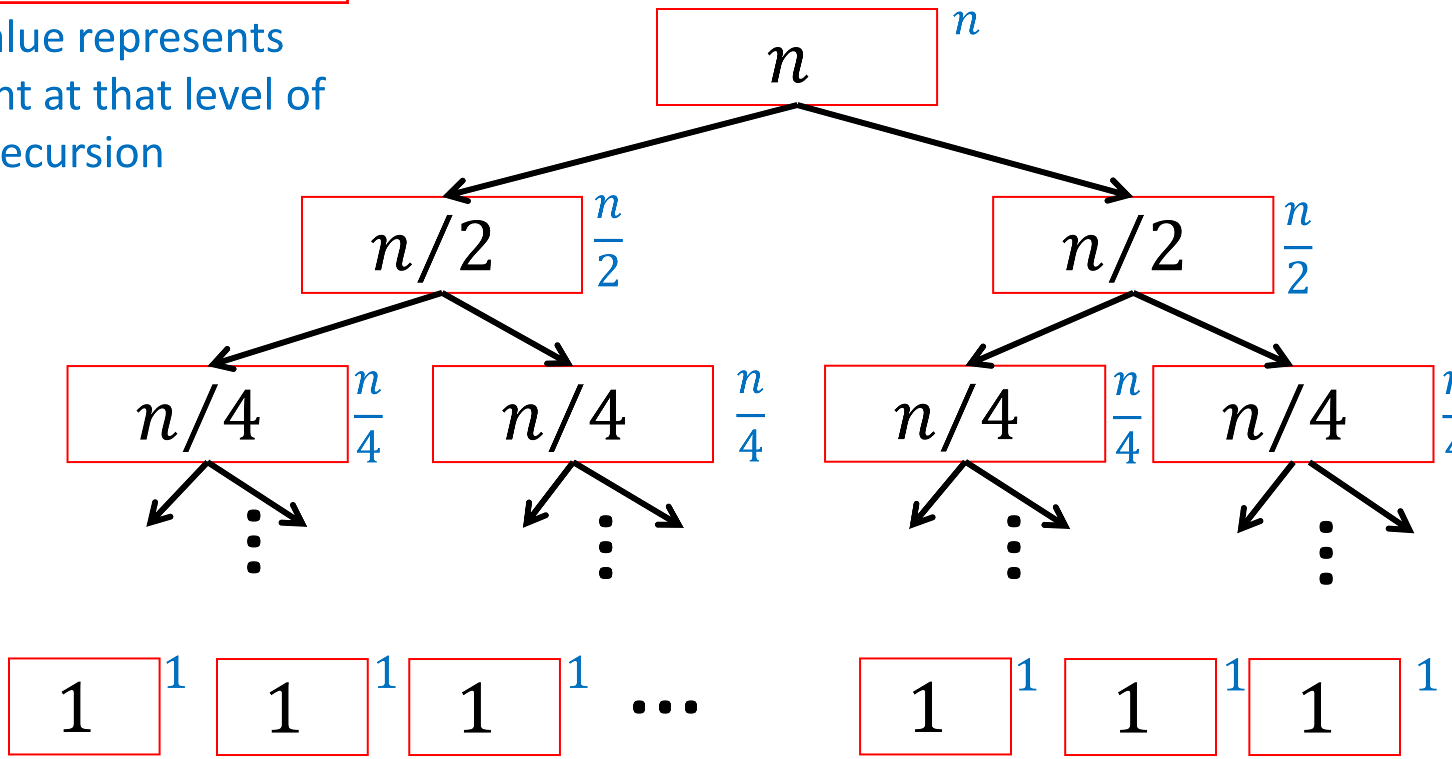
- For each, assume the base case is $n = 1$ and $T(1) = 1$
- $T(n) = 2T\left(\frac{n}{2}\right) + n$
- $T(n) = 2T\left(\frac{n}{2}\right) + n^2$
- $T(n) = 2T\left(\frac{n}{8}\right) + 1$

Tree Method

Red box represents a problem instance

Blue value represents time spent at that level of recursion

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$



$\Rightarrow n$ work per level

$\log_2 n$ levels of recursion

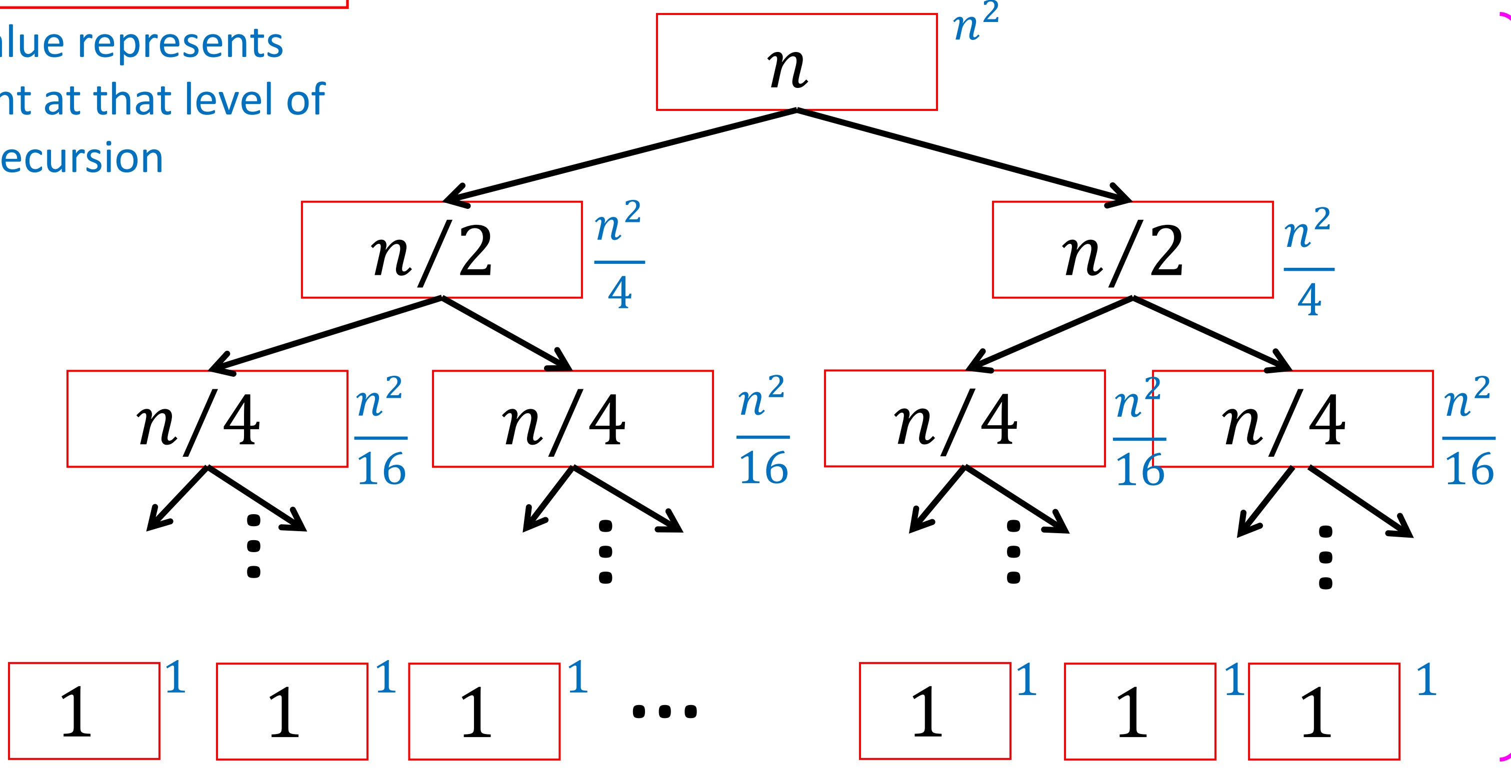
$$T(n) = \sum_{i=1}^{\log_2 n} n$$

Tree Method

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

Red box represents a problem instance

Blue value represents time spent at that level of recursion



$\Rightarrow$?? work per level

$\log_2 n$ levels of recursion

$$T(n) = \sum_{i=1}^{\log_2 n} ??$$

$$T(n) = \sum_{i=1}^{\log_2 n} \frac{n^2}{2^i}$$

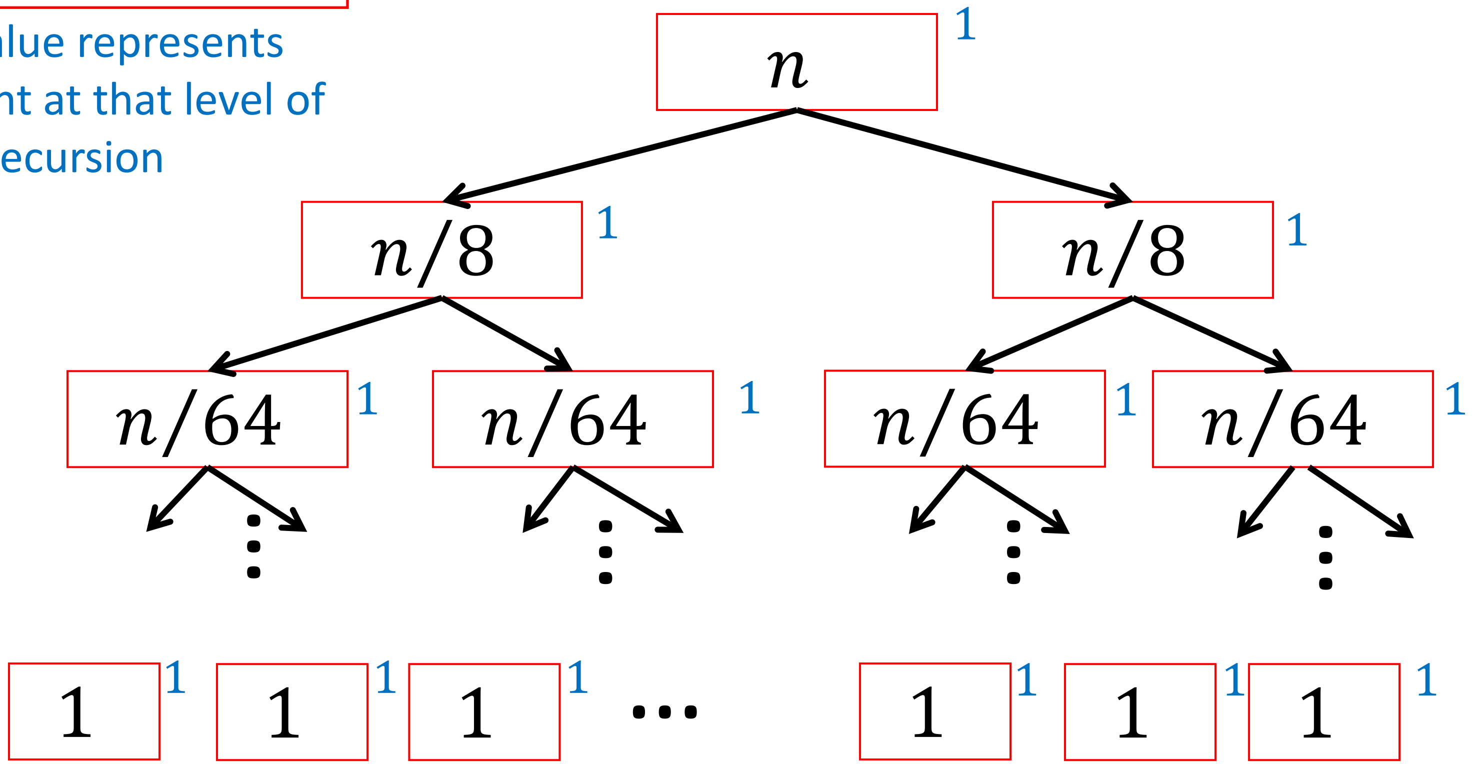
$$= n^2 \cdot \sum_{i=1}^{\log_2 n} \left(\frac{1}{2}\right)^i$$

Tree Method

Red box represents a problem instance

Blue value represents time spent at that level of recursion

$$T(n) = 2T\left(\frac{n}{8}\right) + 1$$



$\Rightarrow 2^i$ work per level

$\log_8 n$ levels of recursion

$$T(n) = \sum_{i=1}^{\log_8 n} 2^i$$

$$T(n) = \sum_{i=1}^{\log_8 n} 2^i$$

$$= \left(\frac{1 - 2^{\log_8 n}}{1 - 2}\right)$$

$$= 2^{\log_8 n} - 1$$

$$= n^{\log_8 2} = n^{\frac{1}{3}}$$

What matters, recursively

- For $T(n) = aT\left(\frac{n}{b}\right) + f(n)$
  - The following are important for asymptotic behavior:
    - The value of a
    - The value of b
    - Asymptotic behavior of $f(n)$
  - The following are not important for asymptotic behavior:
    - Constants and non-dominant terms in $f(n)$
    - The base case

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
  - IsEmpty
    - 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
    - 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)

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.extract())
Print(PQ.extract())
Print(PQ.extract())
Print(PQ.extract())
Print(PQ.extract())
```

Priority Queue, example

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

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

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	$\Theta(n)$	$\Theta(n)$

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…)

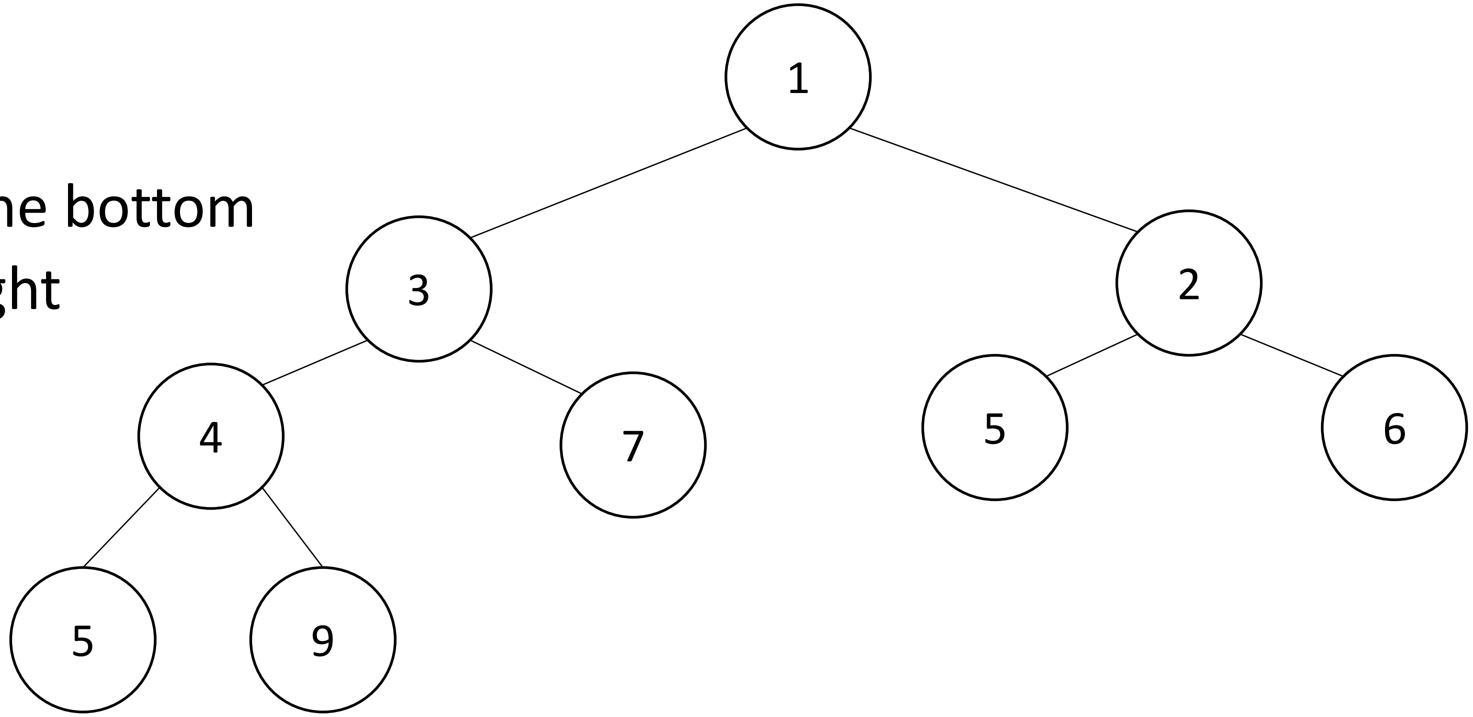
Thinking through implementations

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…)

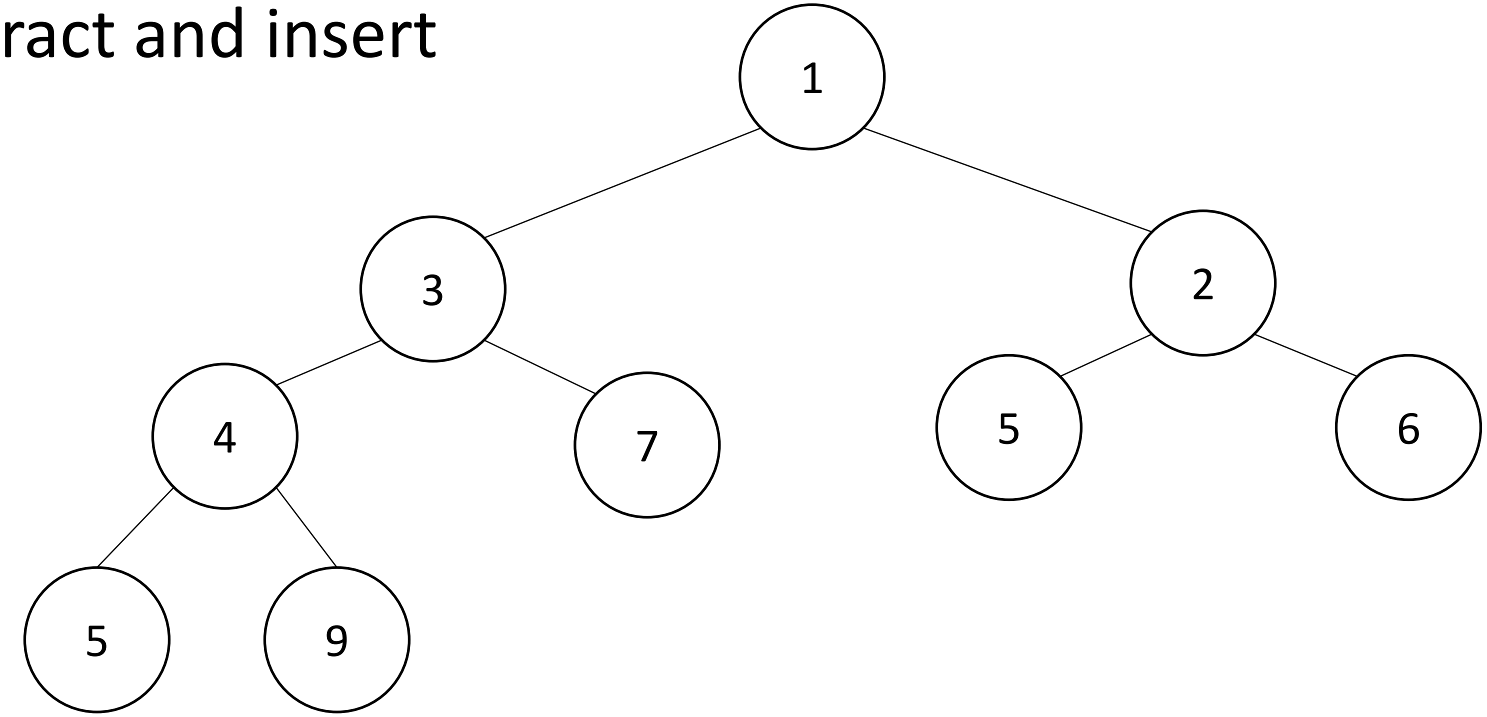
Trees for Heaps

- 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



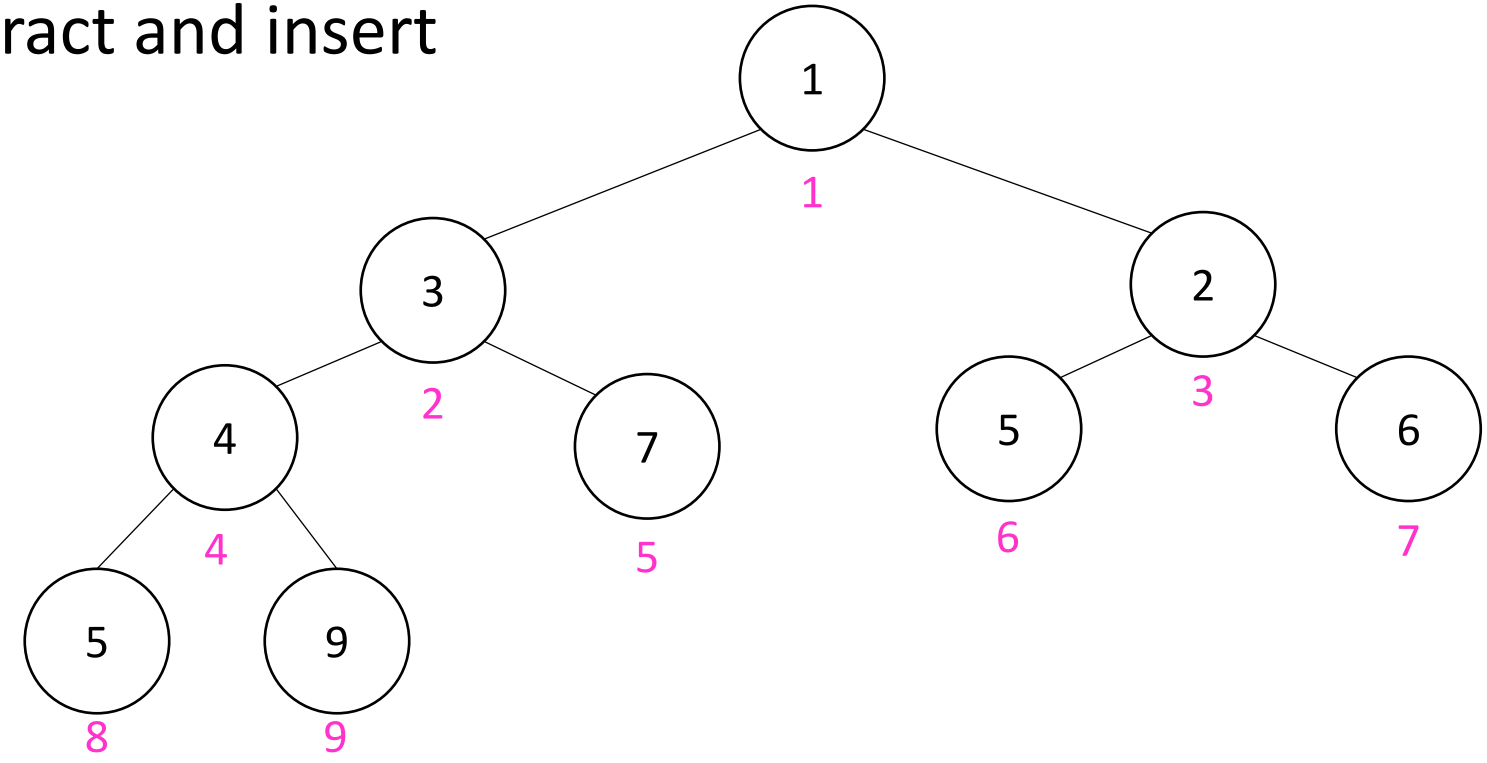
Heap – Priority Queue Data Structure

- 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



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



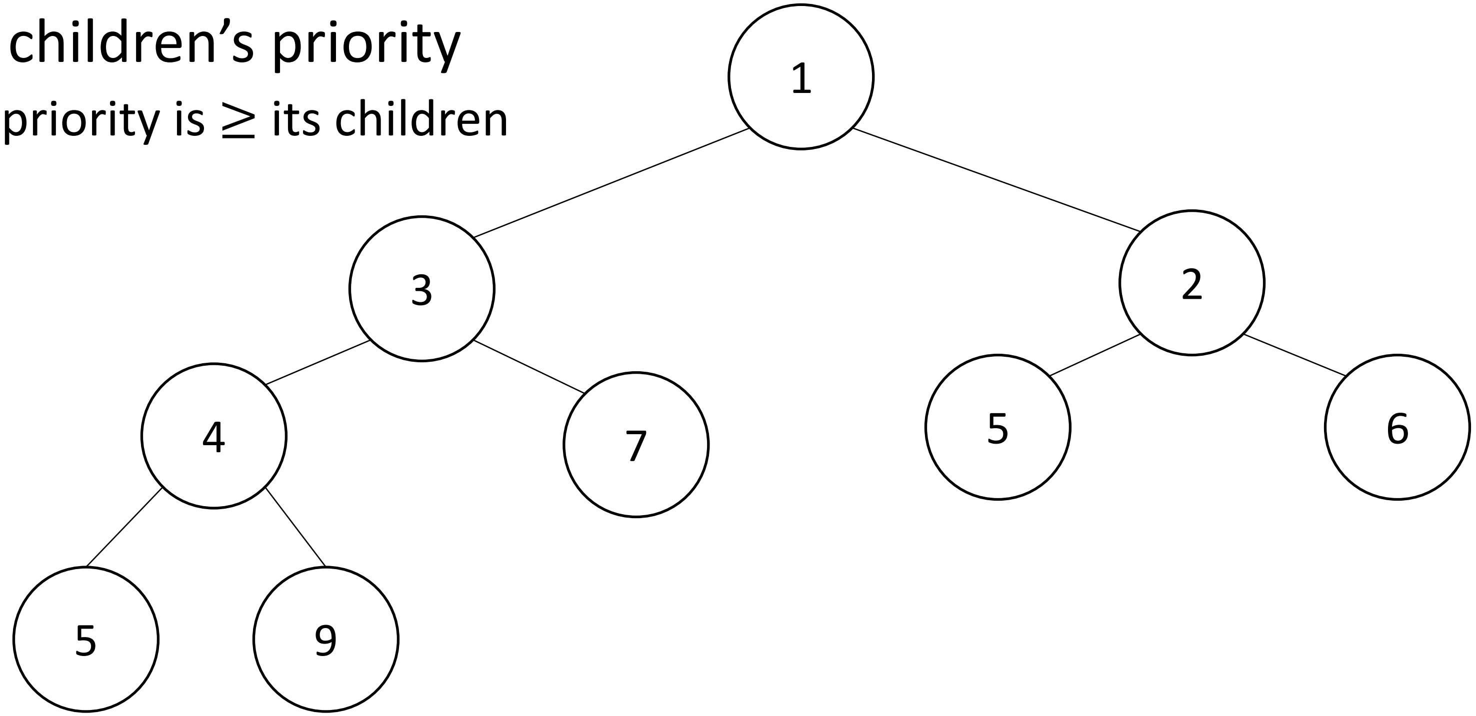
Challenge!

- 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?
  - $\Theta(\log n)$

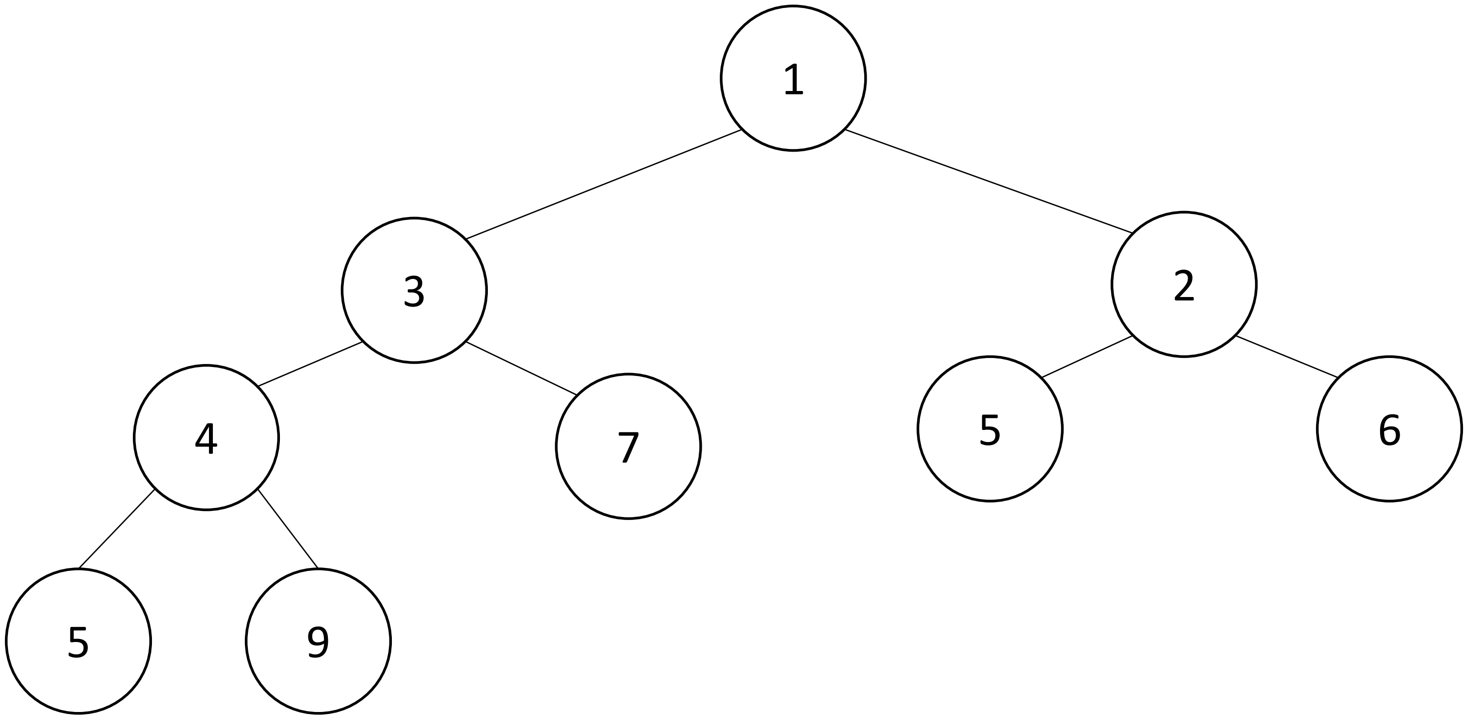
- Heap Idea:
  - If n values are inserted into a complete tree, the height will be roughly $\log n$
  - Ensure each insert and extract requires just one "trip" from root to leaf

(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?

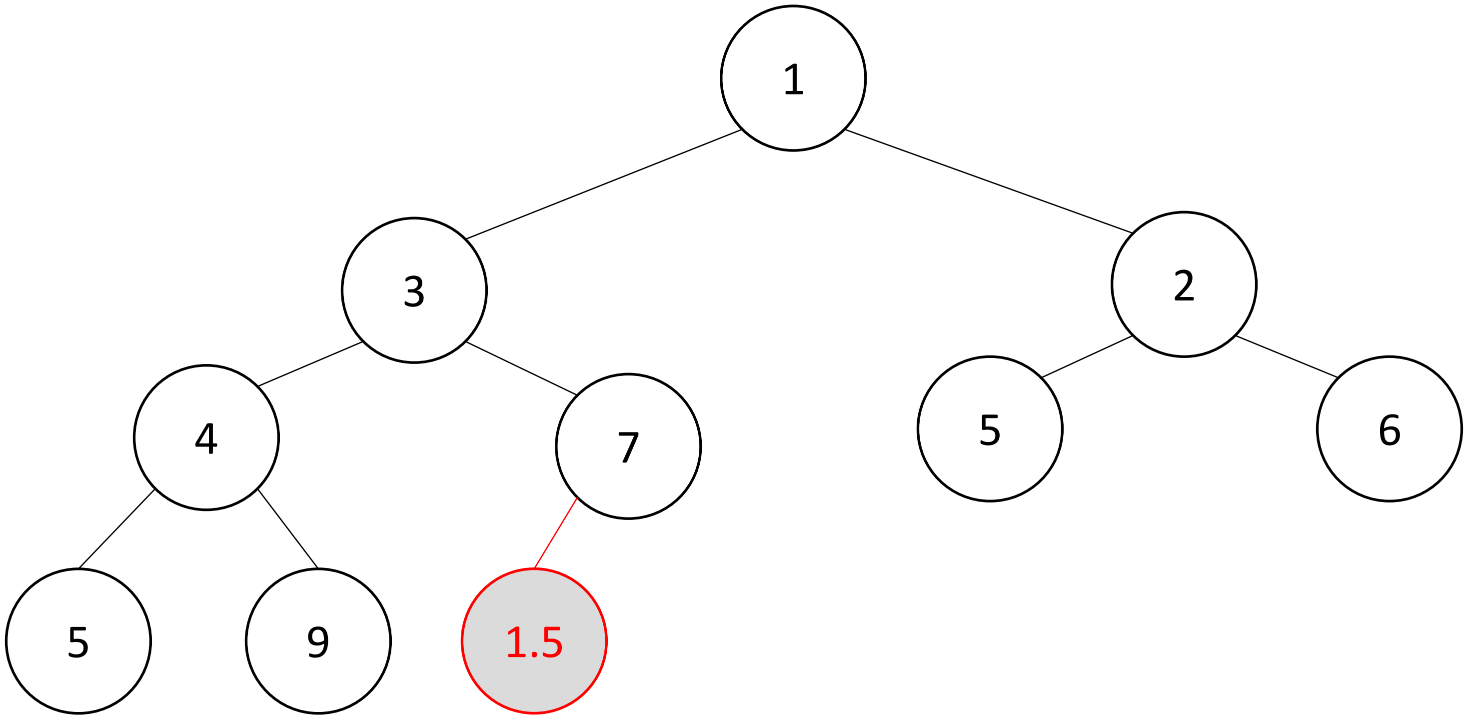


Heap Insert



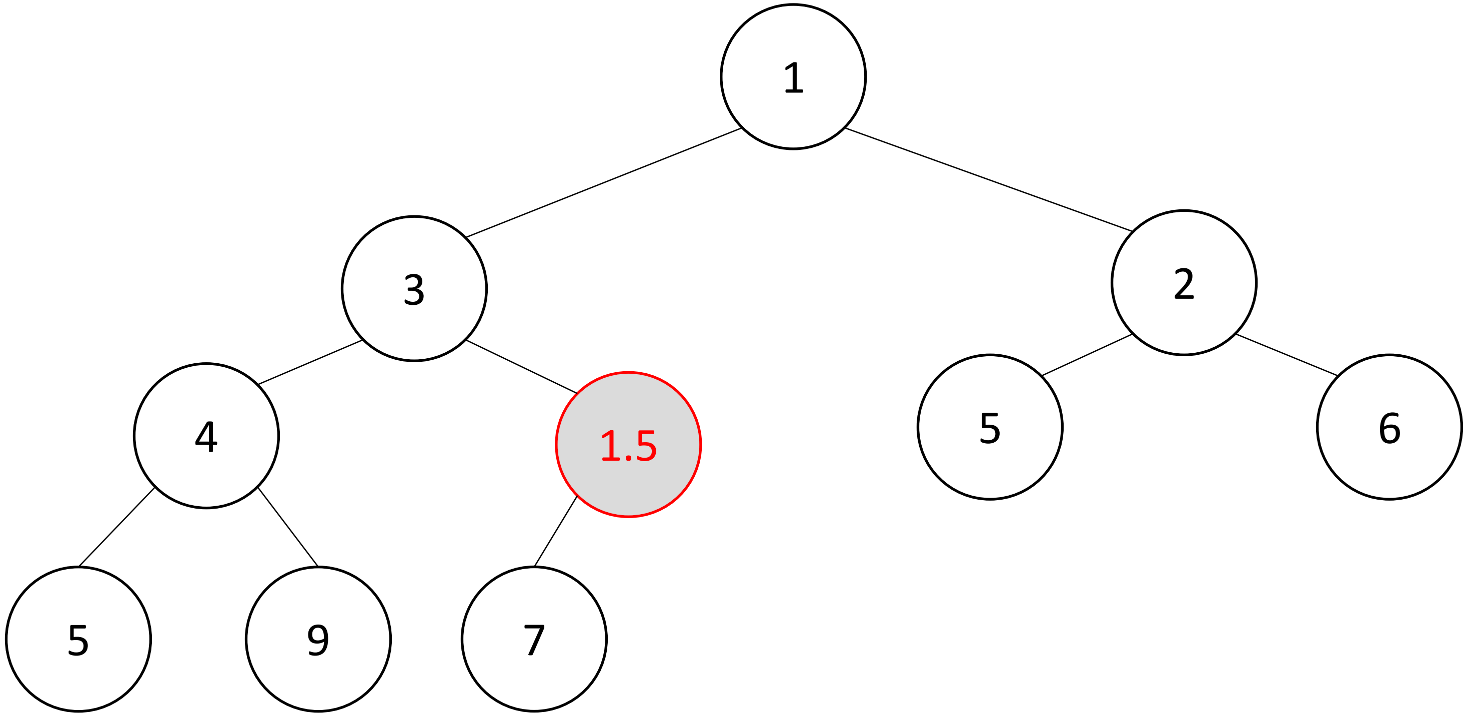

```
insert(item, priority){
  put item in the “next open” spot (keep tree complete)
  while (priority < parent’s priority){
    swap item with parent
  }
}
```

Heap Insert




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

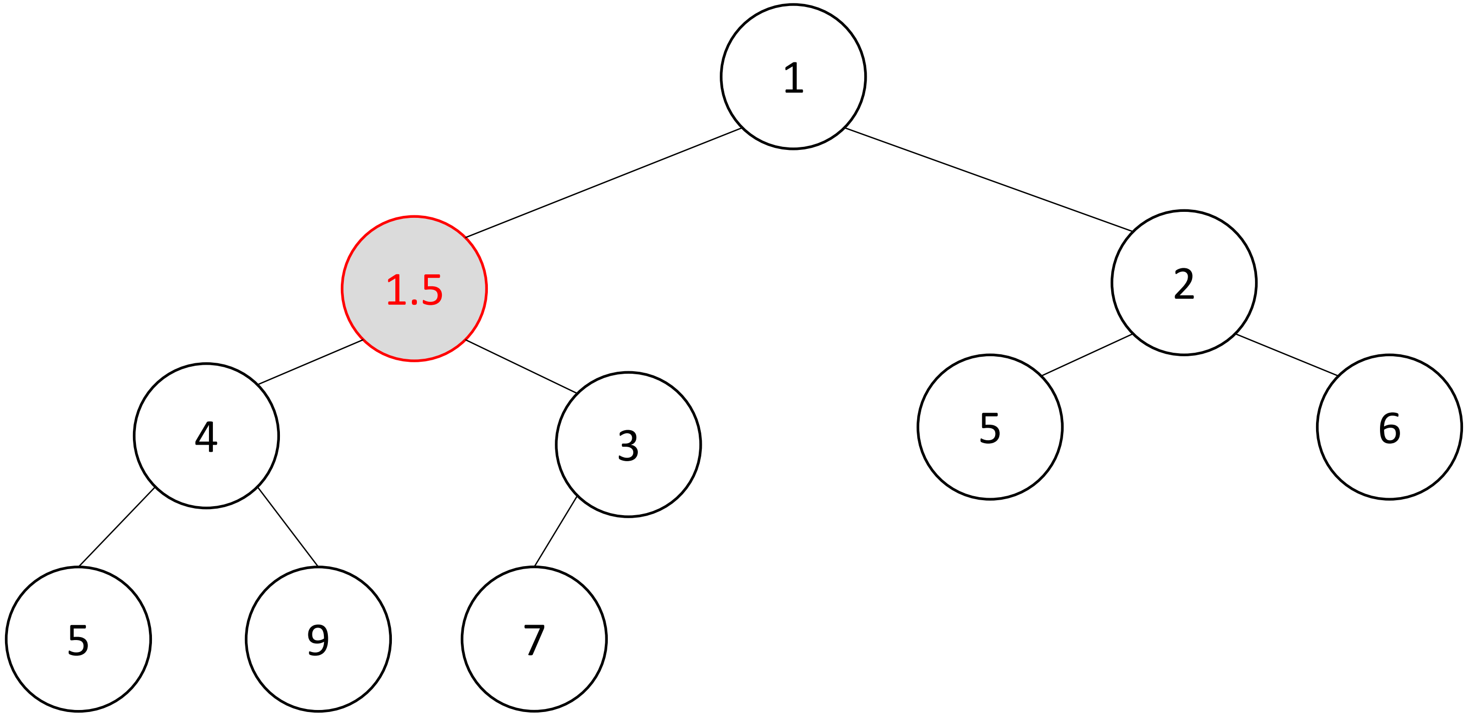
Heap Insert




```
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



insert(item, priority){

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

while (priority < parent's priority){

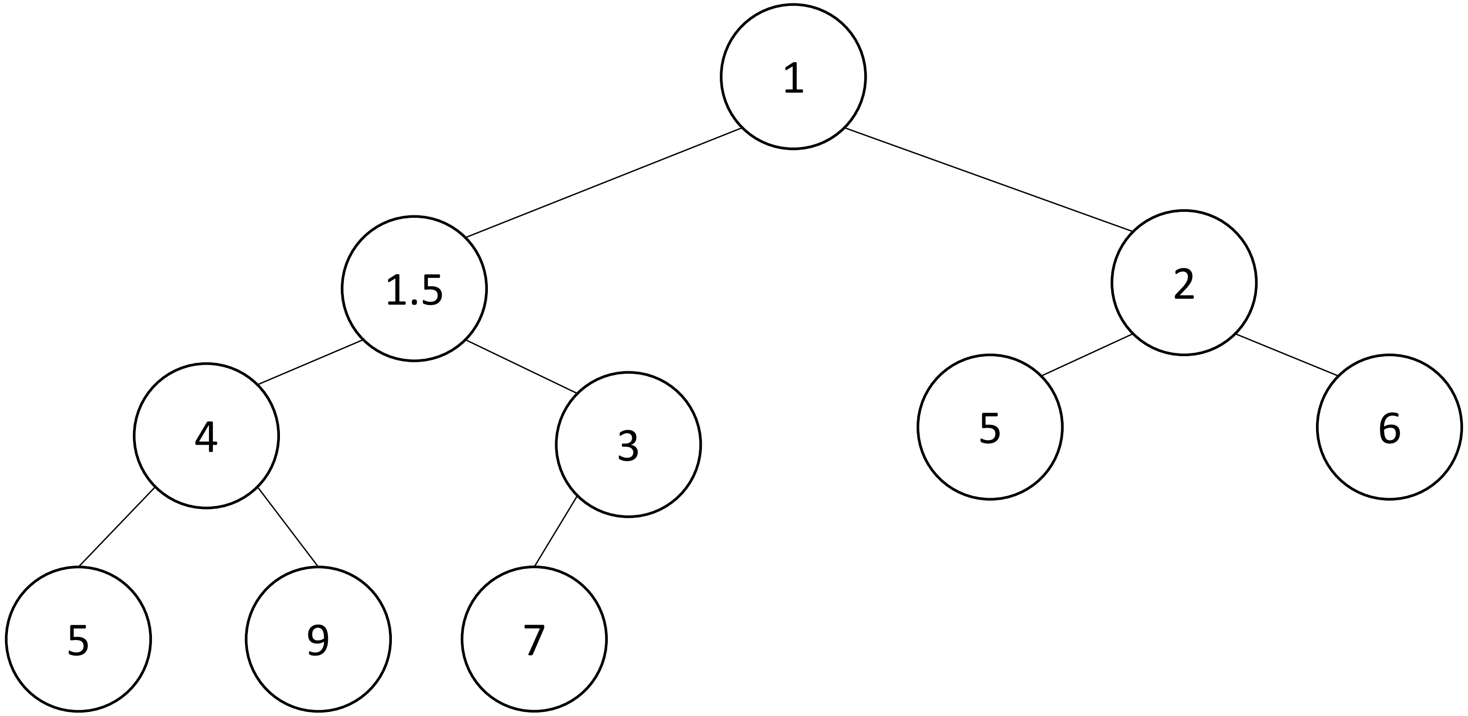
swap item with parent

}

}

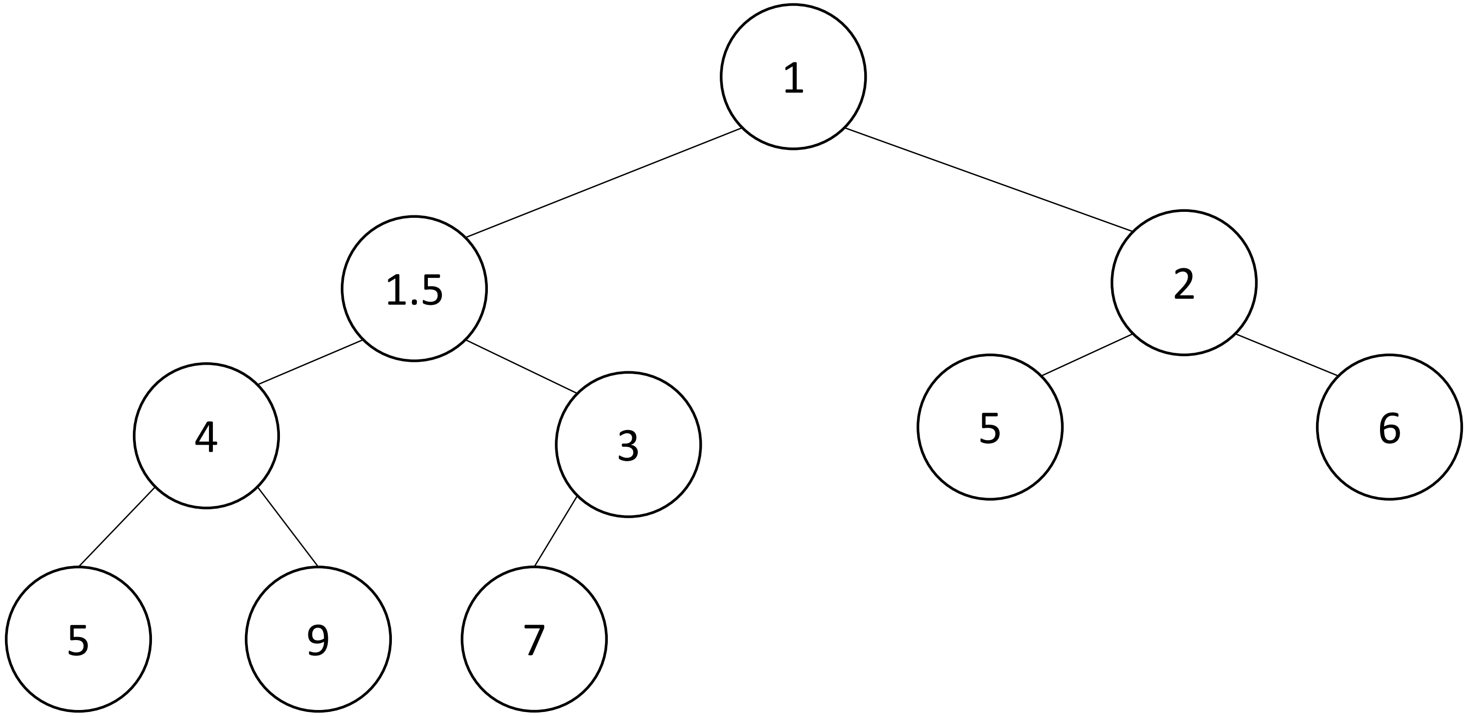
Percolate Up

Heap Insert



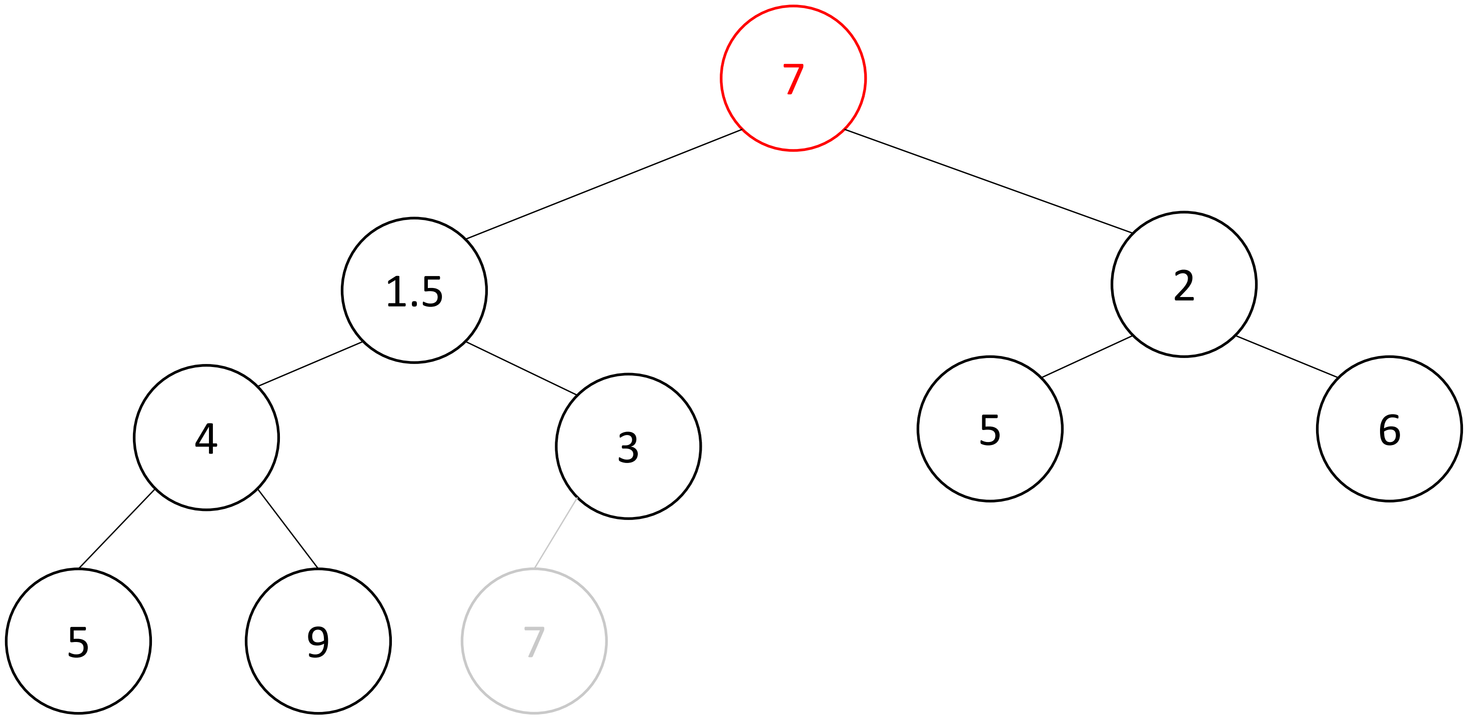

```
insert(item, priority){
    put item in the "next open" spot (keep tree complete)
    while (priority < parent's priority){
        swap item with parent
    }
}
```

Heap extract



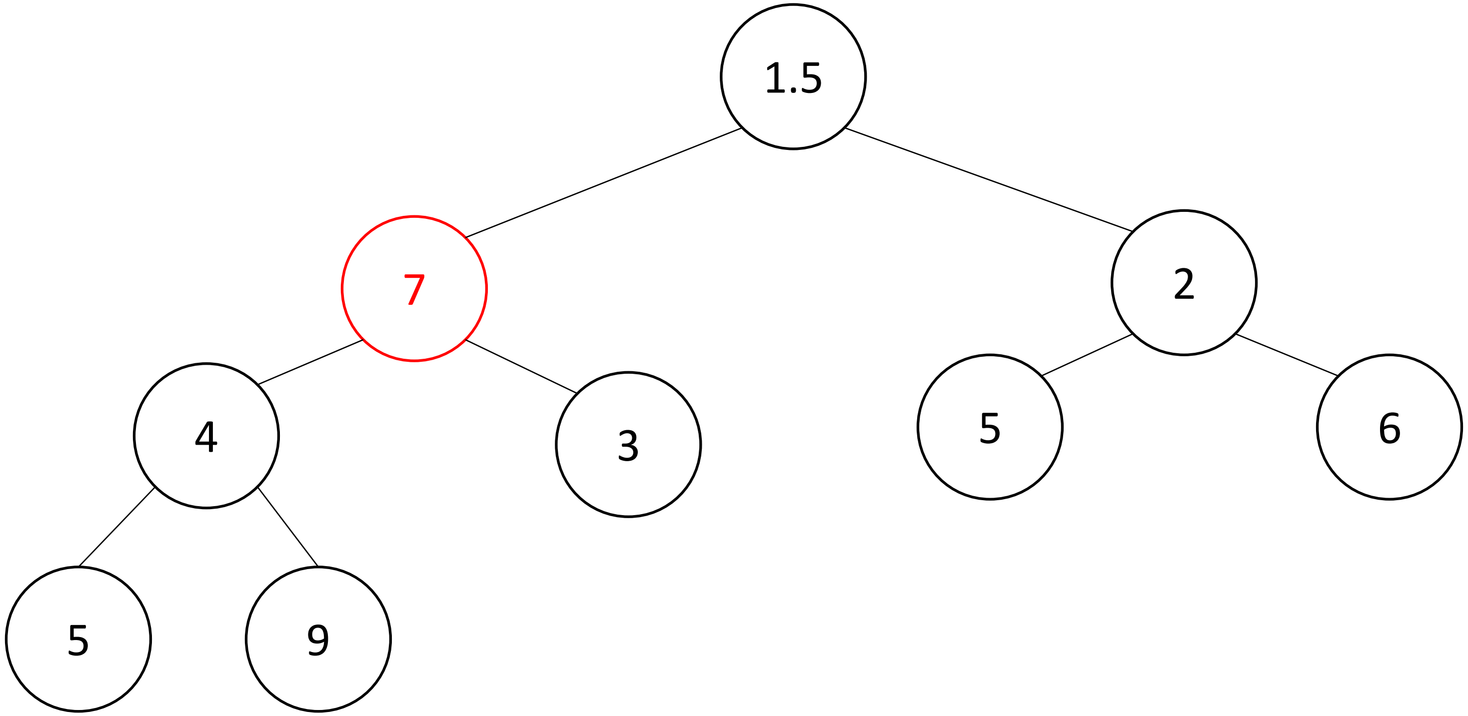

```
extract(){
  min = root
  curr = bottom-right item
  move curr to the root
  while(curr > curr.left || curr > curr.right){
    swap curr with its smallest child
  }
  return min
}
```

Heap extract




```
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
}
```

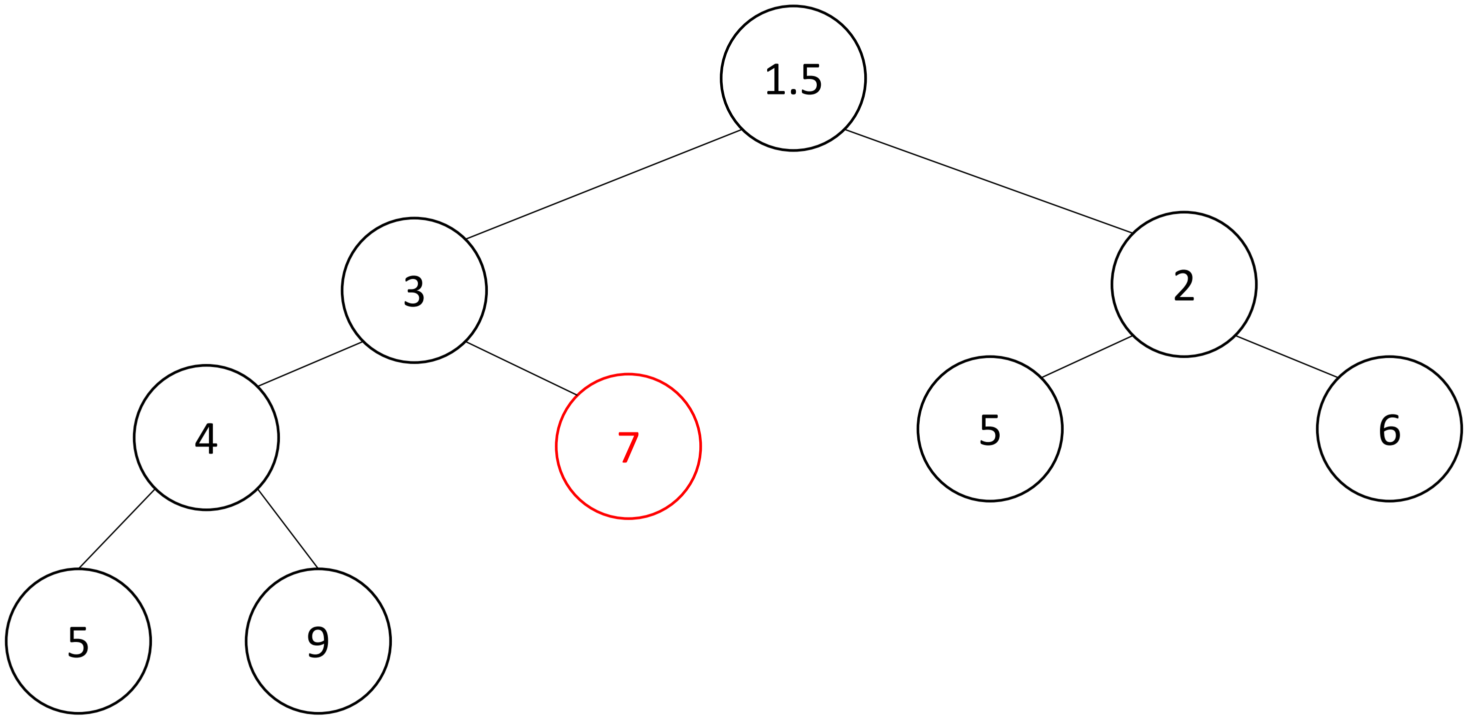
Heap extract




```
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

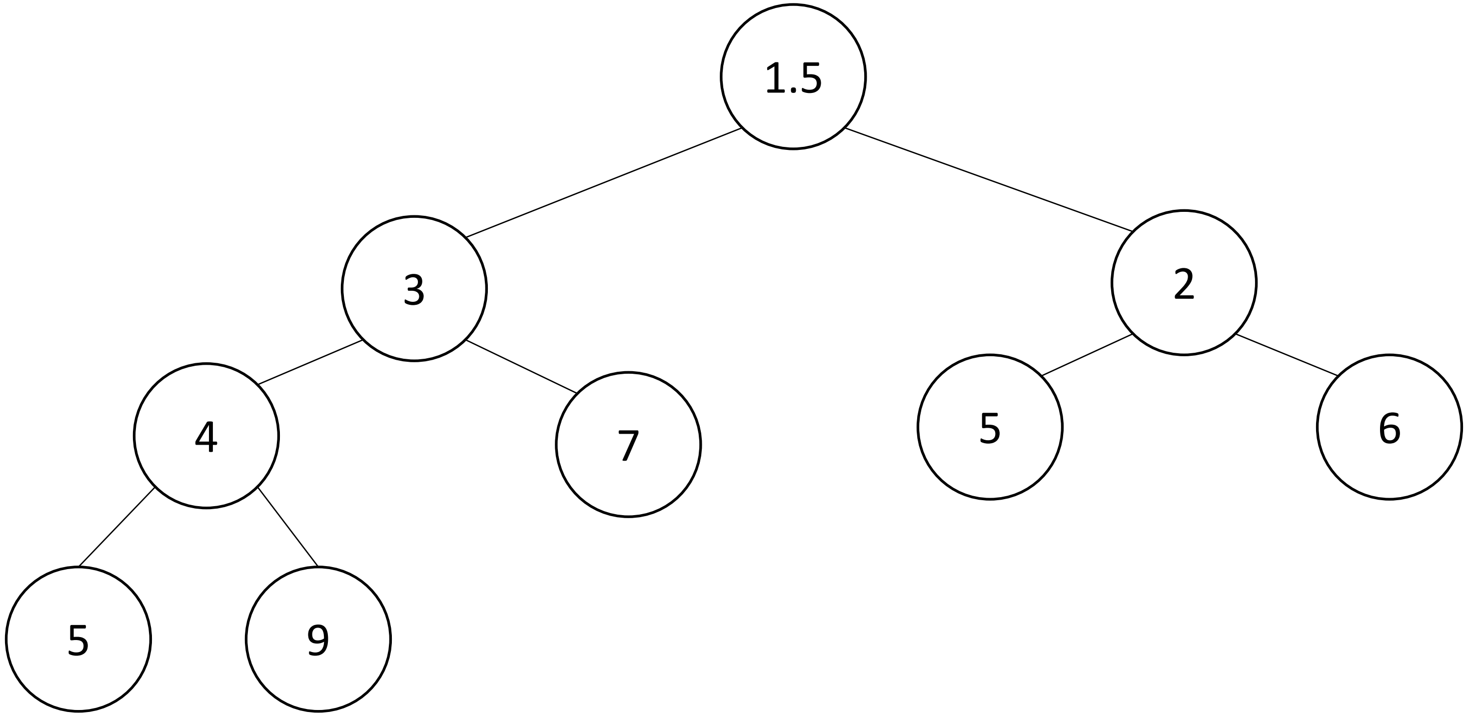
Heap extract




```
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




```
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
}
```

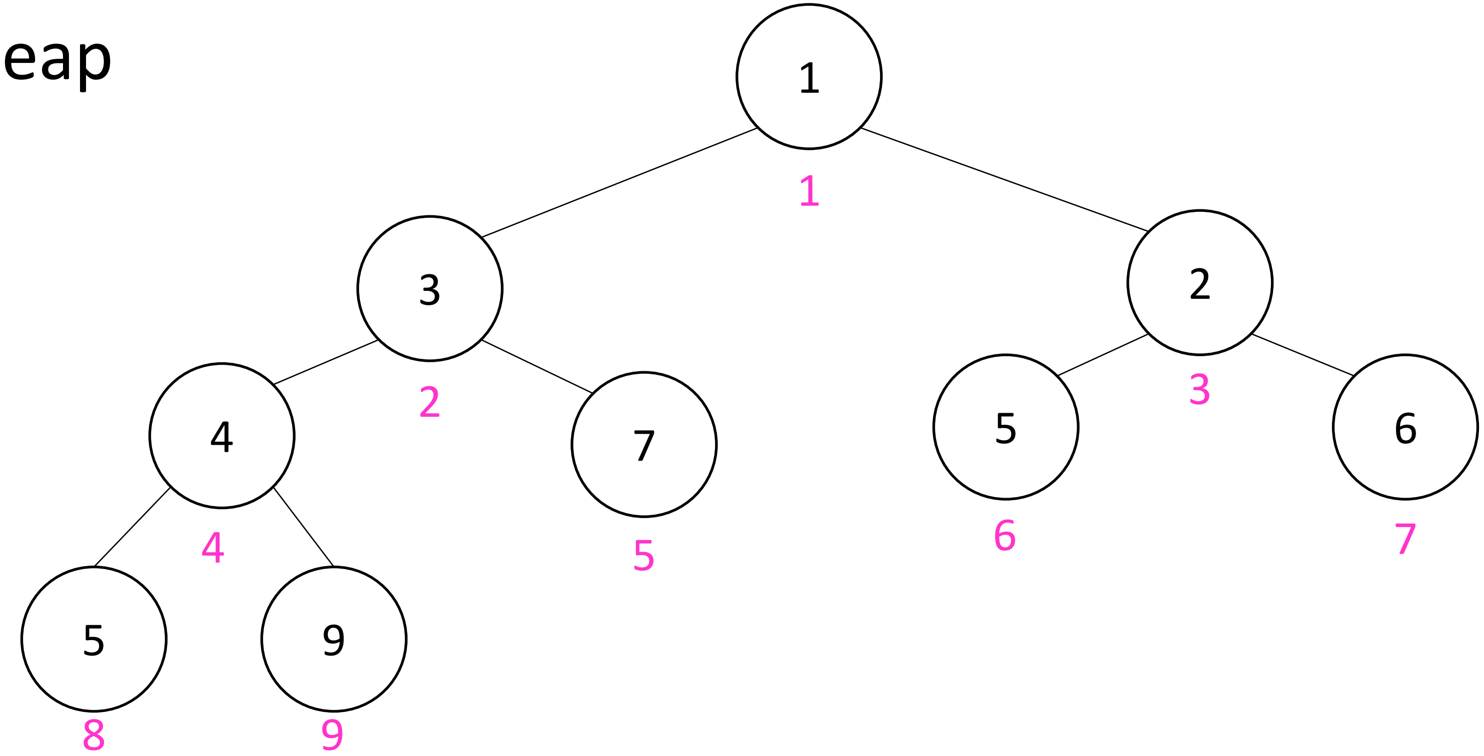
Percolate Up and Down (for a Min Heap)

- 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

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

- 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:

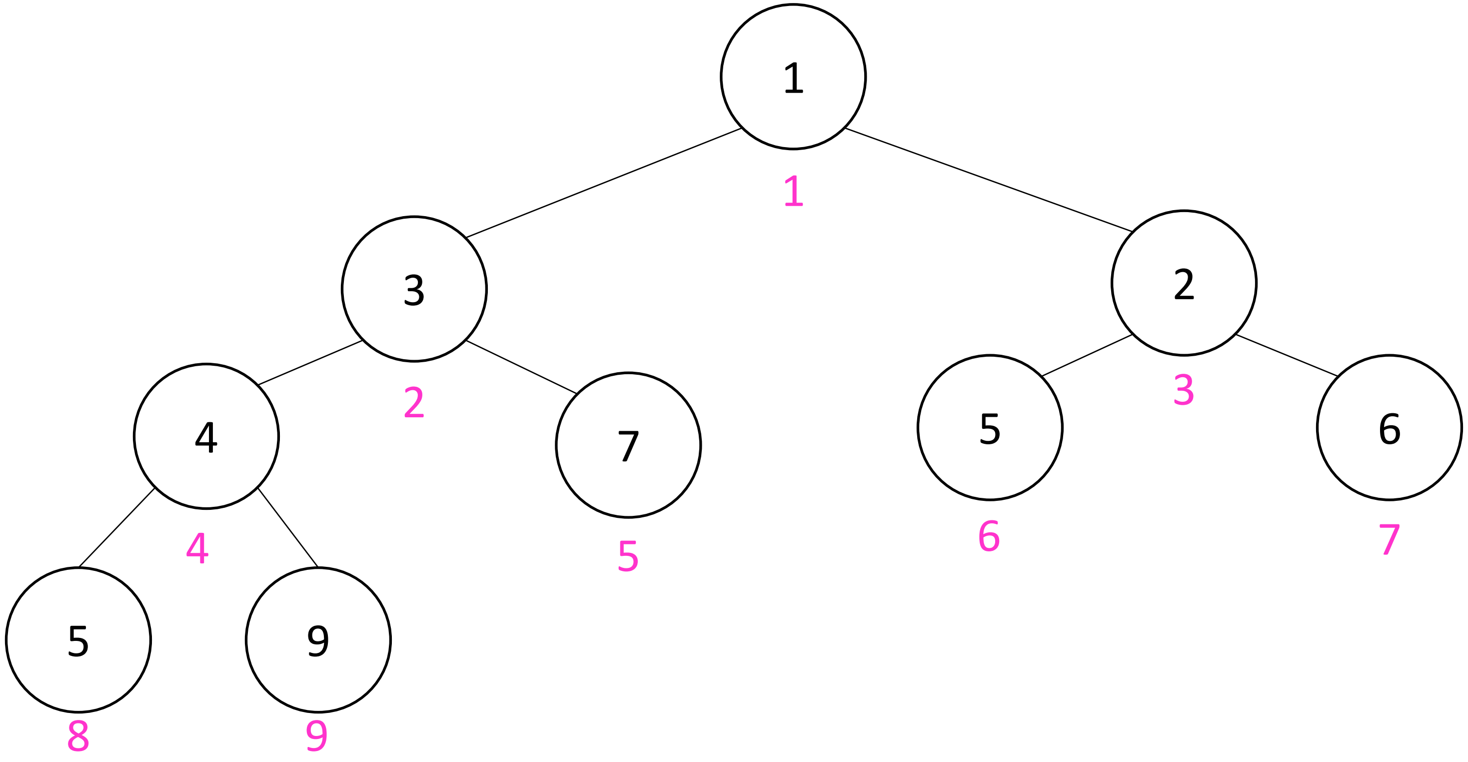


Insert Psuedocode

For simplicity, assume is the same as priority

	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(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 Psuedocode

```
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
  }
}
```

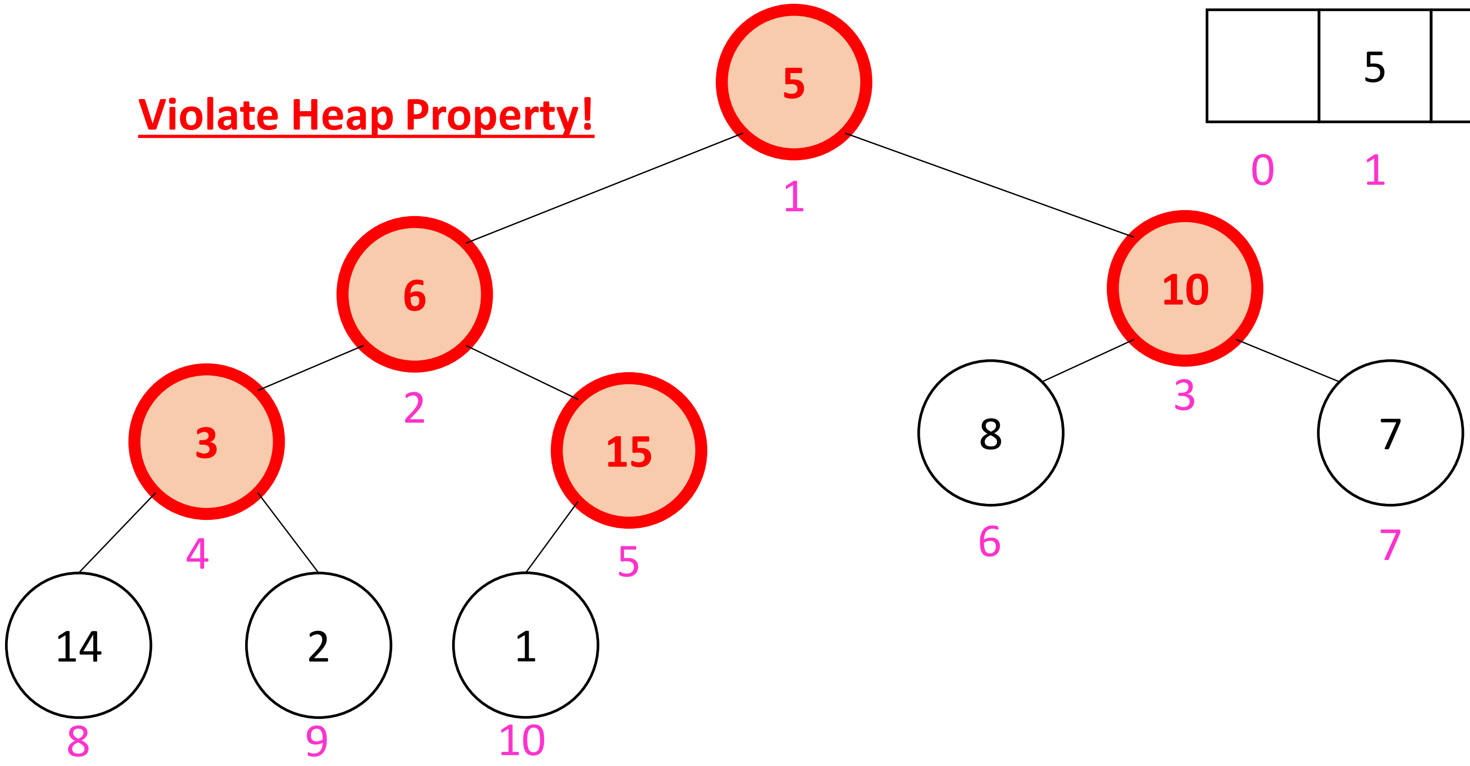
Other Operations

- 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

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

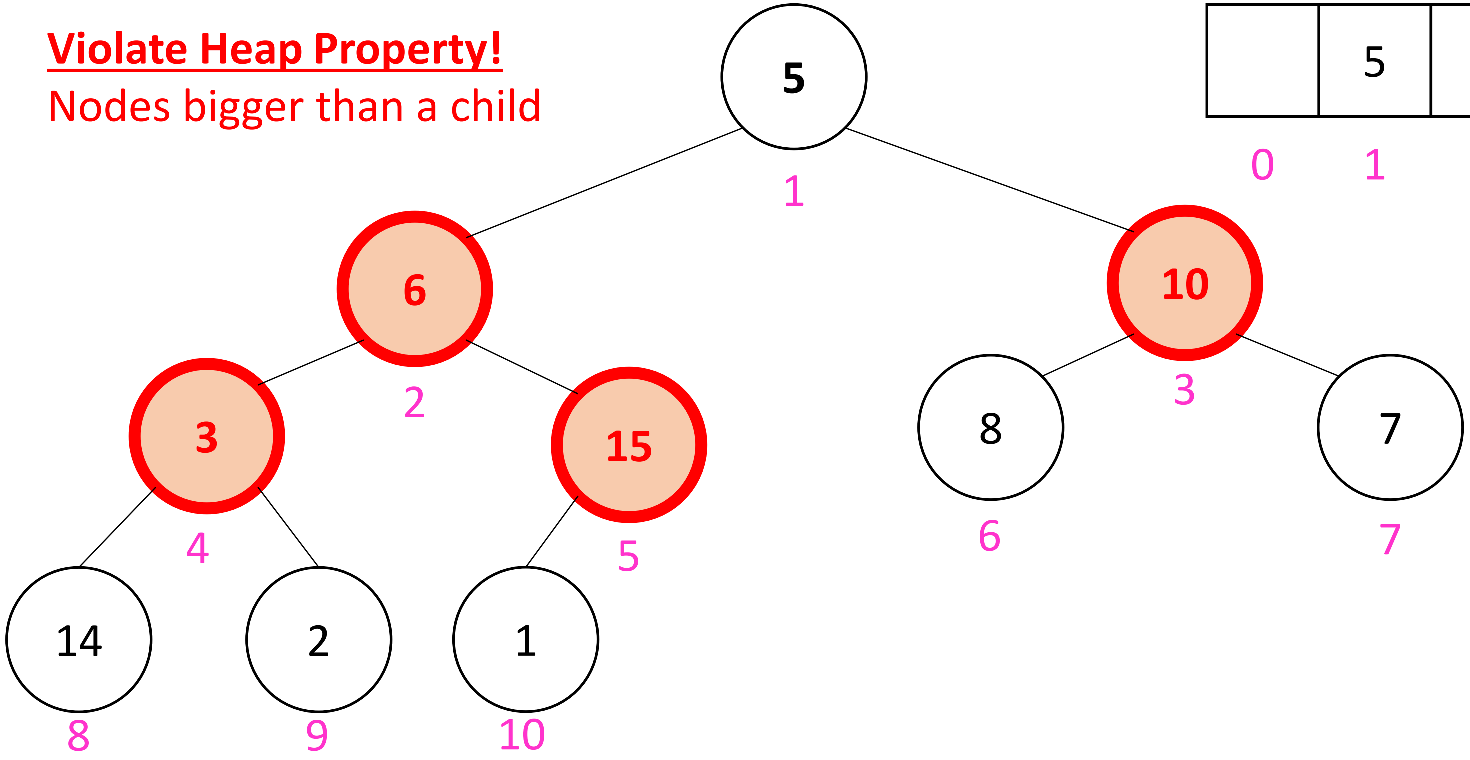
Floyd’s buildHeap method

- 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



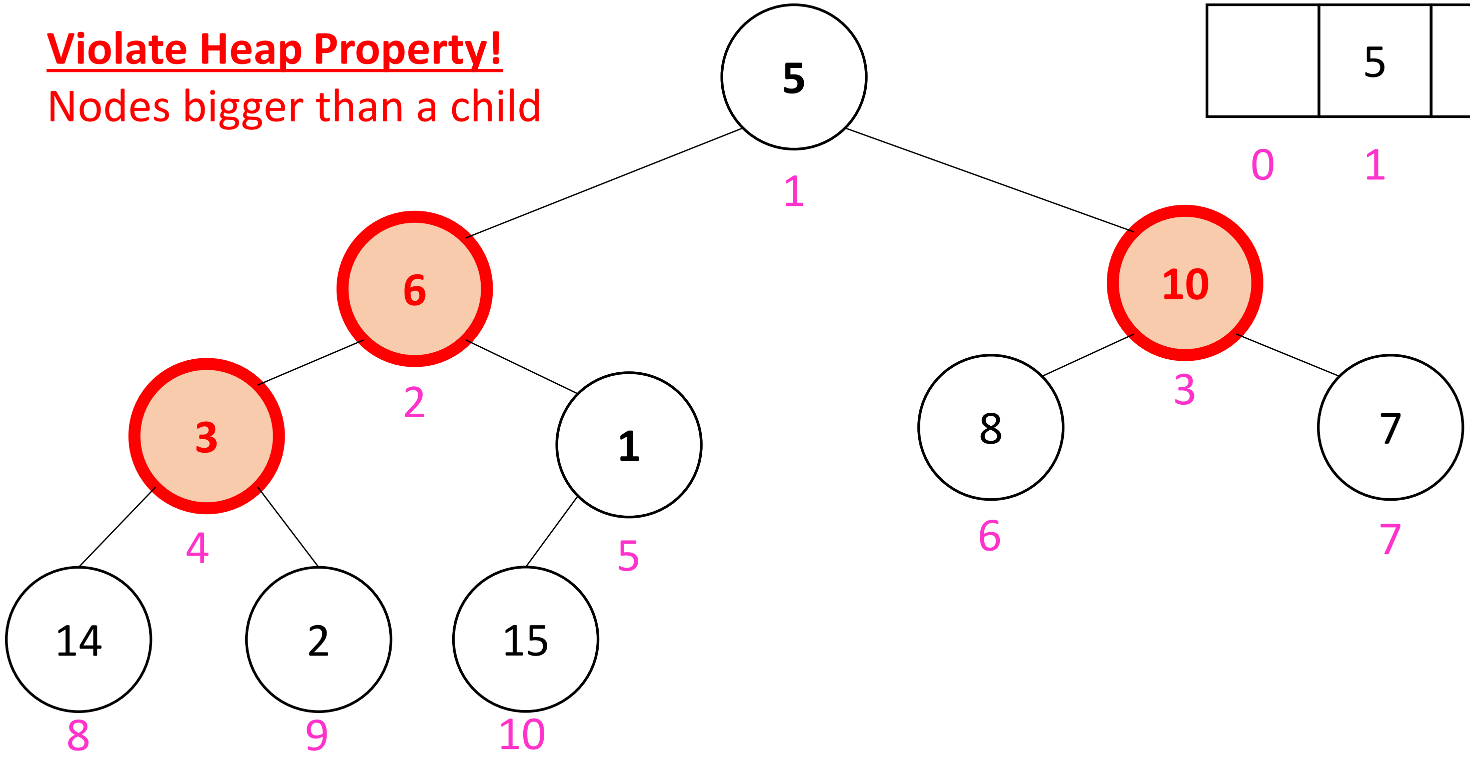
	5	6	10	3	15	8	7	14	2	1
0	1	2	3	4	5	6	7	8	9	10

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

Floyd’s buildHeap method

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

<u>Violate Heap Property!</u>
Nodes bigger than a child



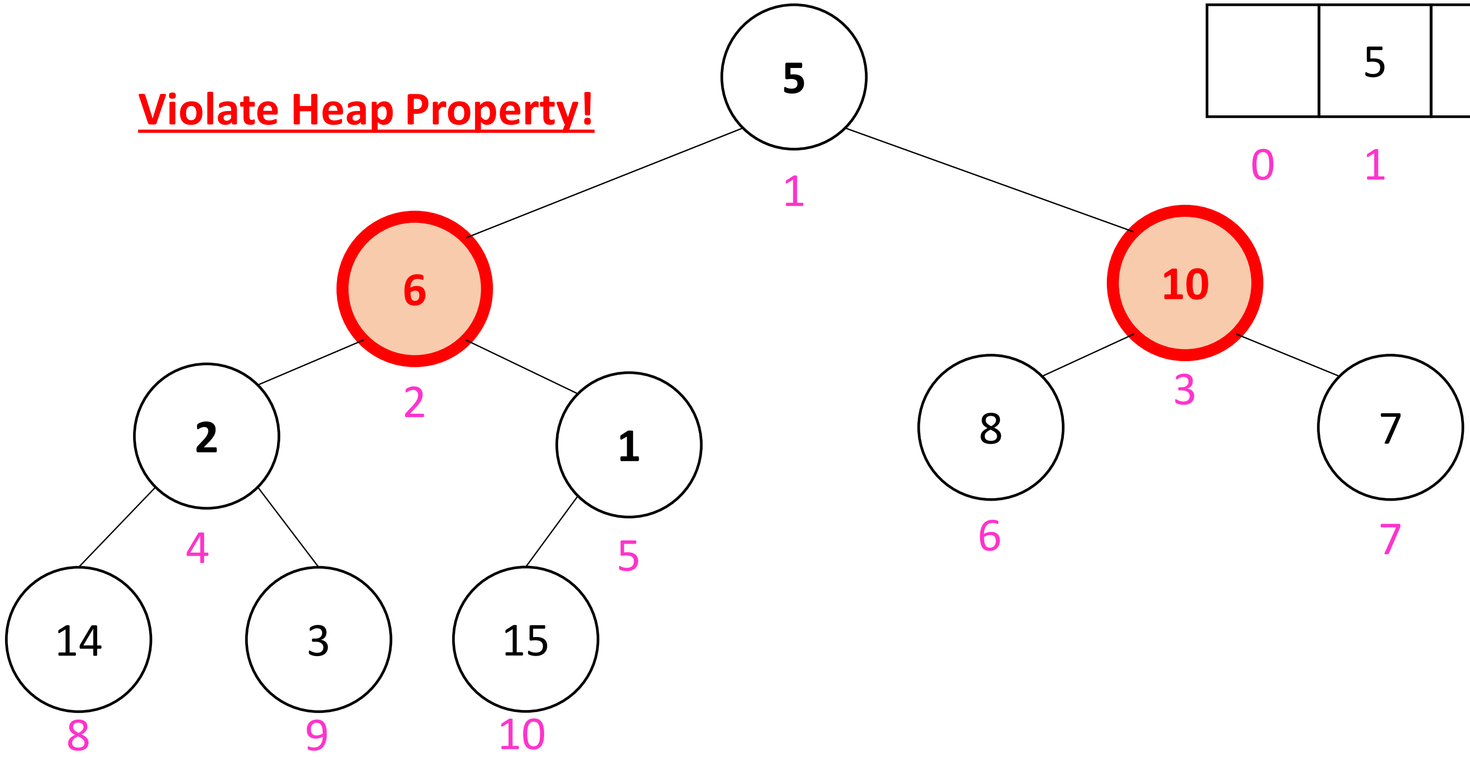
	5	6	10	3	15	8	7	14	2	1
0	1	2	3	4	5	6	7	8	9	10

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

Floyd's buildHeap method

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

Violate Heap Property!

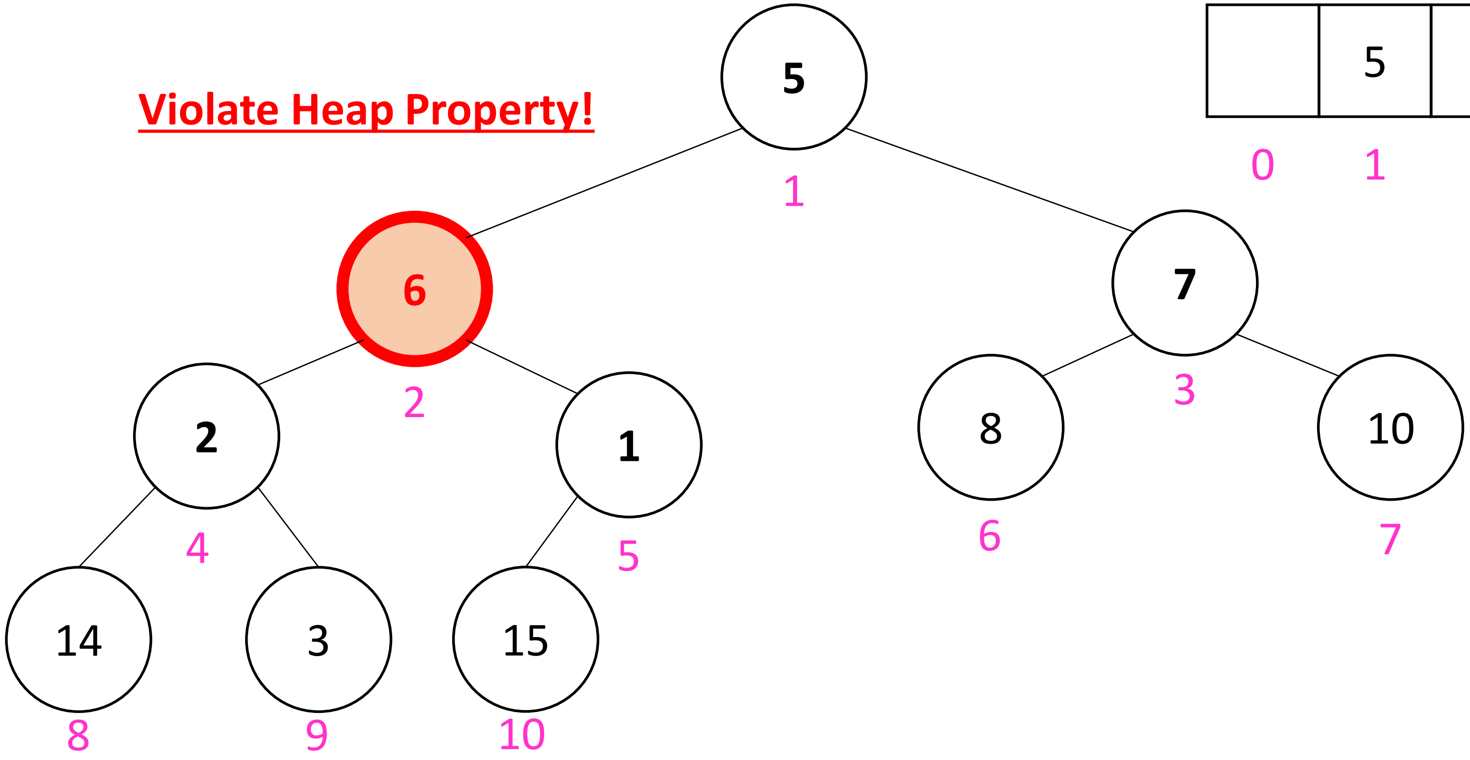


	5	6	10	3	15	8	7	14	2	1
0	1	2	3	4	5	6	7	8	9	10

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

Floyd's buildHeap method

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

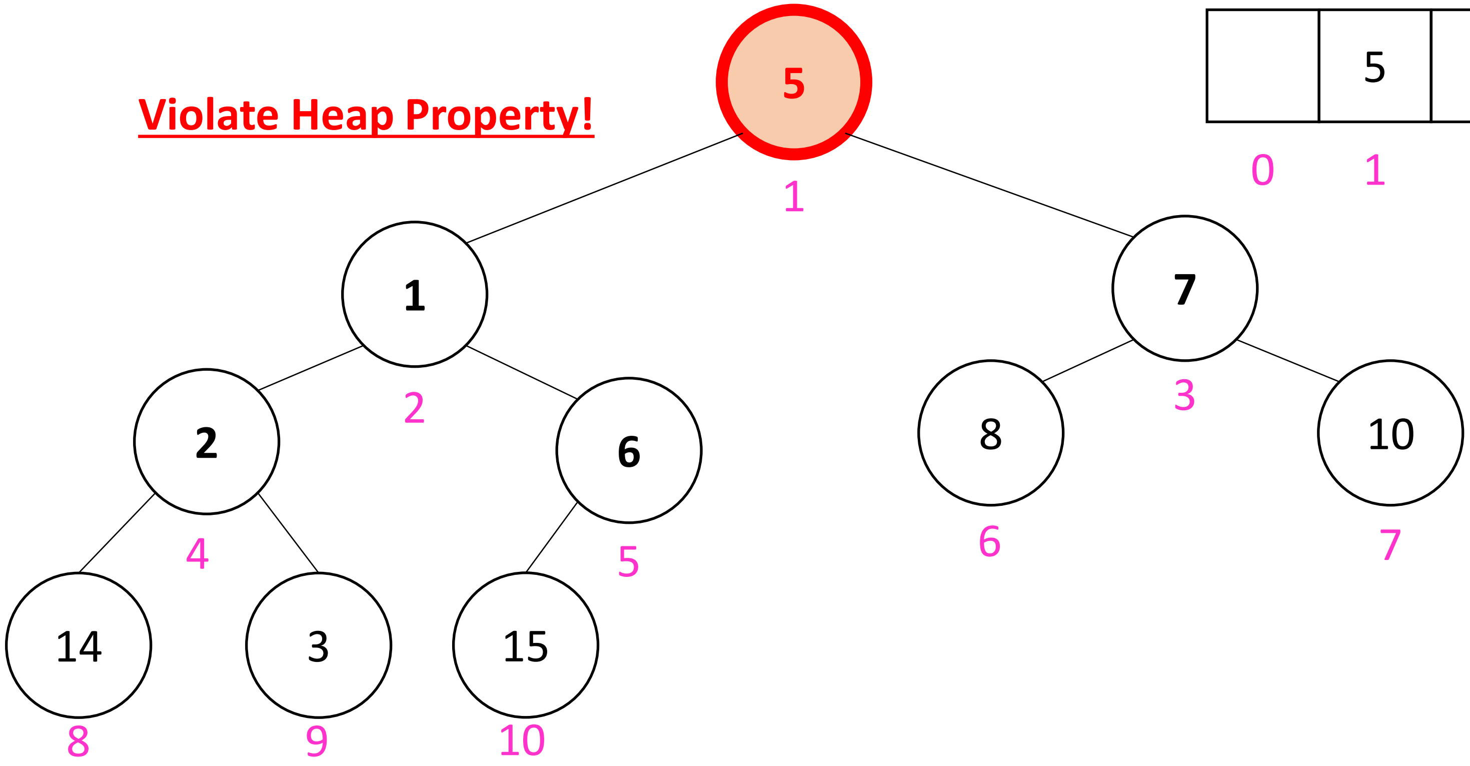


	5	6	10	3	15	8	7	14	2	1
0	1	2	3	4	5	6	7	8	9	10

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

Floyd's buildHeap method

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



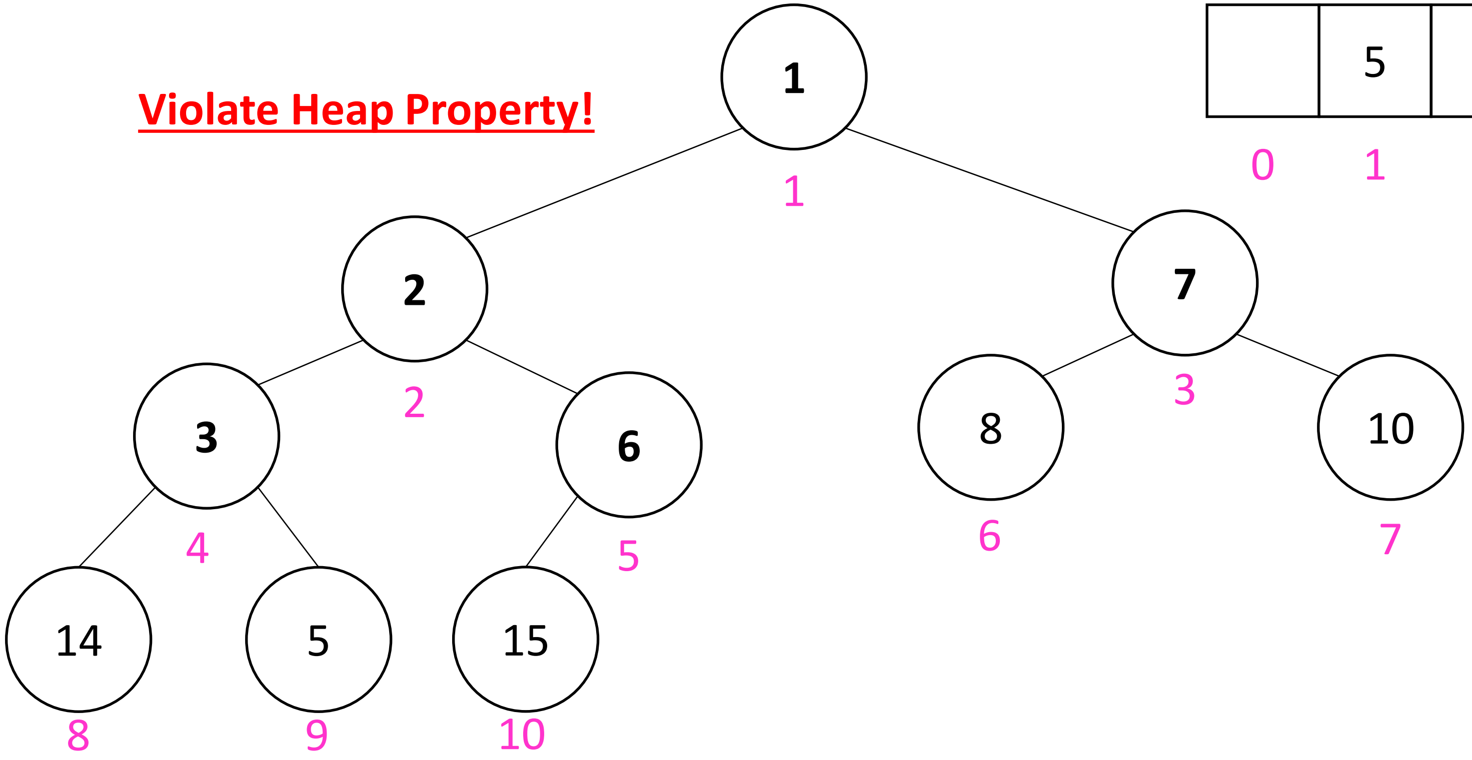
	5	6	10	3	15	8	7	14	2	1
0	1	2	3	4	5	6	7	8	9	10

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

Floyd’s buildHeap method

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

Violate Heap Property!



	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: