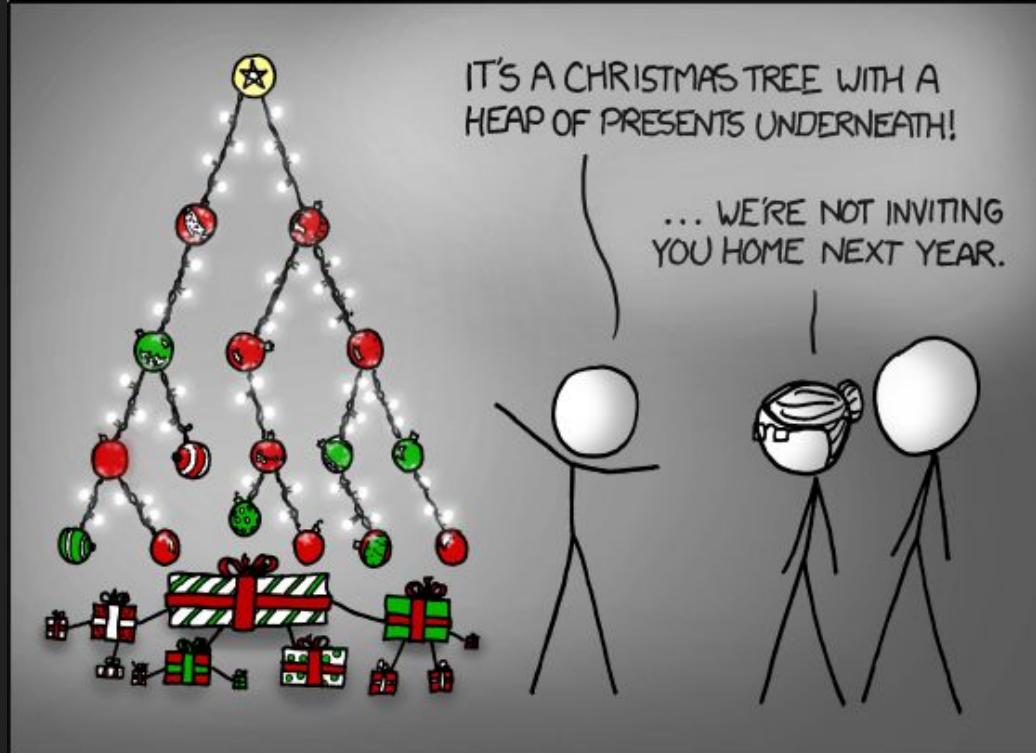


Lab 9: Priority Queues and Heaps



Announcements

- Lab 9 - due Friday 03/18
- HW 6 - due Tuesday 03/29
- Project 2: Ataxx
 - Checkpoint - due Friday 03/18
 - Project - due Friday 04/01

T-REX LET'S SAY YOU HAVE A GIANT HEAP OF SAND AND I REMOVE ONE GRAIN OF IT AT A TIME

Ooh, let's!!



CLEARLY WHEN THERE'S ONLY ONE GRAIN OF SAND LEFT IT'S NOT A HEAP ANYMORE Clearly!

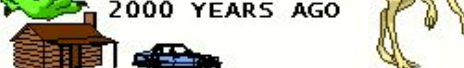


AHA MY FRIEND BUT WHEN PRECISELY DID IT SWITCH FROM HEAP TO NON-HEAP

I dunno! At some fuzzy point it would switch for most observers from "heap" to, say, "small pile", and there we can draw the line. Language isn't that precise.



LISTEN THIS IS A CLASSIC PARADOX THAT EUBULIDES OF MILETUS CAME UP WITH OVER 2000 YEARS AGO



YOU NEED TO HAVE YOUR MIND BLOWN NOW OKAY

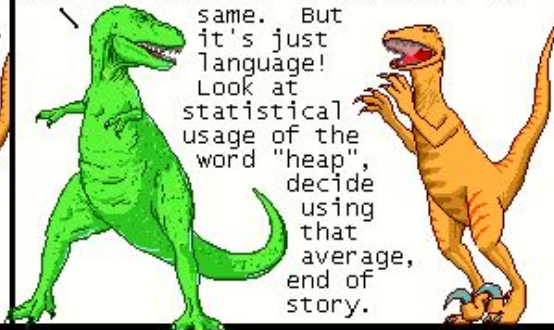
sounds kinda dumb to me!



what does?

The point at which a shrinking heap of sand becomes a non-heap. Clearly I'm supposed to struggle with an arbitrary threshold, because piles on either side of it look much the same. But

it's just language! Look at statistical usage of the word "heap", decide using that average, end of story.



Oh snap, philosophers! Did T-Rex just totally school you with his statistically-based descriptivist approach to semantics?

IT APPEARS THAT HE TOTALLY DID!!



It also appears he's speaking in the third person because he's so impressed with his awesome self!

Theory - Priority Queues

Queue Interface

Based on *time* (recency)

<code>enqueue(val)</code>	Adds <code>val</code> to the queue
<code>peek()</code> / <code>poll()</code>	Returns the item in the queue that was enqueued <i>longest ago</i> .

Queue Interface

Based on *time* (recency)

<code>enqueue(val)</code>	Adds <code>val</code> to the queue
<code>peek()</code> / <code>poll()</code>	Returns the item in the queue that was enqueued <i>longest ago</i> .

PriorityQueue: A Queue that prioritizes certain items (e.g hospital ER)

Examples:

Queue Interface

Based on *time* (recency)

<code>enqueue(val)</code>	Adds <code>val</code> to the queue
<code>peek()</code> / <code>poll()</code>	Returns the item in the queue that was enqueued <i>longest ago</i> .

PriorityQueue: A Queue that prioritizes certain items (e.g hospital ER)

Examples:

- OS Process Scheduling
- Sorting
- Greedy algorithms (e.g. “shortest path”)

PriorityQueue ADT

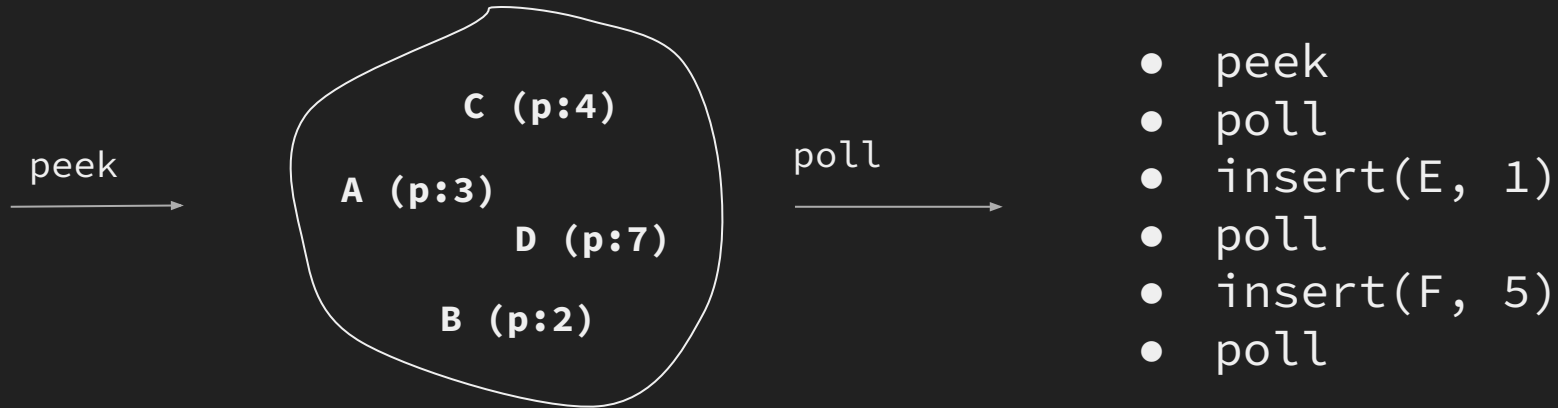
<code>insert(val, priority)</code>	Adds <code>val</code> to the queue with <i>priority value</i> <code>priority</code>
<code>peek() / poll()</code>	Returns the item in the queue with the <i>highest priority</i> .

- Min priority queue: highest priority == *lowest priority value*
 - There's also a "max priority queue" / max heap, where highest priority value is highest priority
- No specification on how to deal with ties

PriorityQueue ADT

<code>insert(val, priority)</code>	Adds <code>val</code> to the queue with <i>priority value</i> <code>priority</code>
<code>peek()</code> / <code>poll()</code>	Returns the item in the queue with the <i>highest priority</i> .

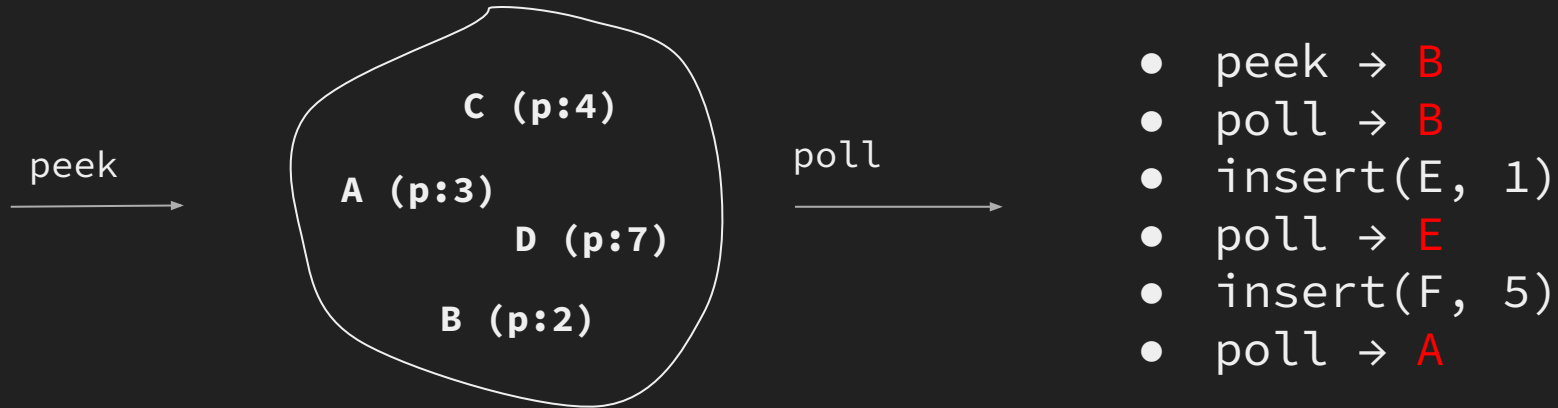
- Highest Priority == Lowest Priority Value



PriorityQueue ADT

<code>insert(val, priority)</code>	Adds <code>val</code> to the queue with <i>priority value</i> <code>priority</code>
<code>peek()</code> / <code>poll()</code>	Returns the item in the queue with the <i>highest priority</i> .

- Highest Priority == Lowest Priority Value

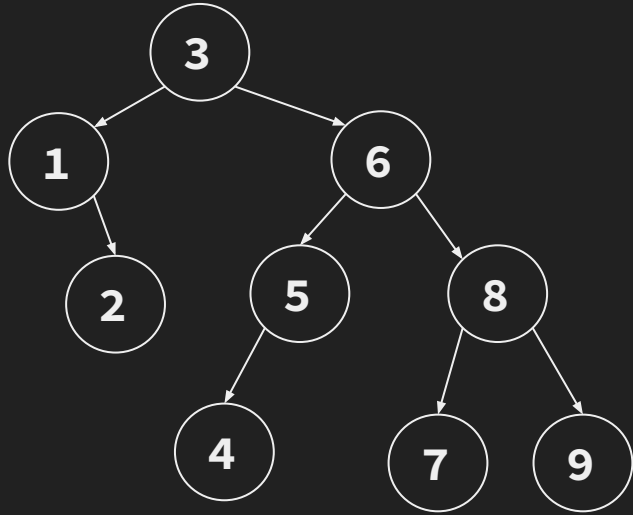


Theory - Heaps

Review: BST Properties

BST Property (recursive invariant)

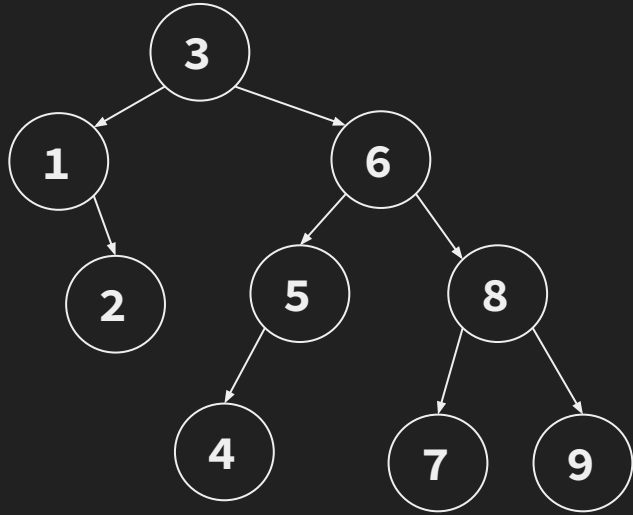
- Left Children are smaller
- Right Children are larger



In Contrast: Heap Properties

BST Property (recursive invariant)

- Left Children are smaller
- Right Children are larger

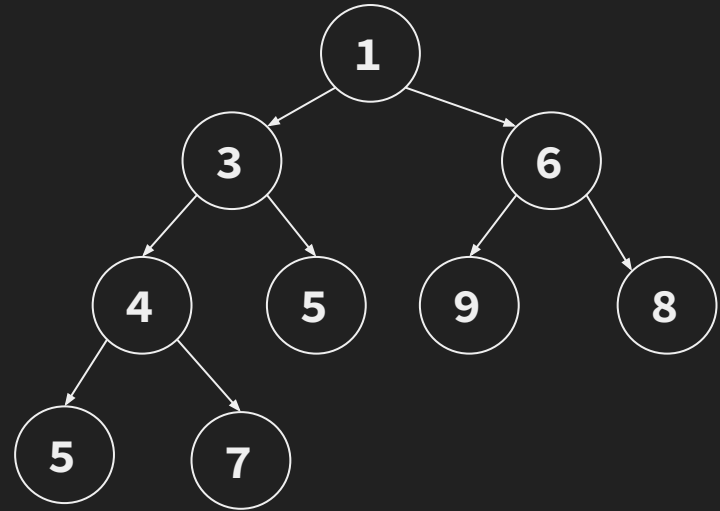


Min-Heap Property (recursive invariants)

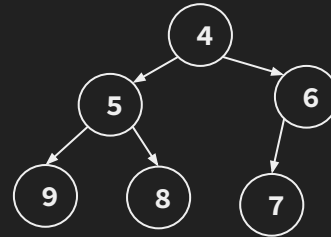
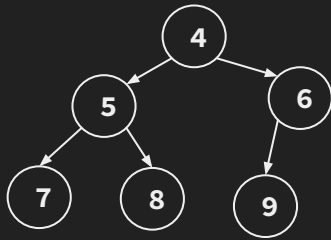
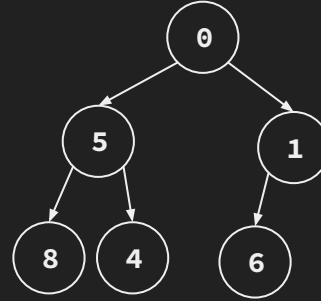
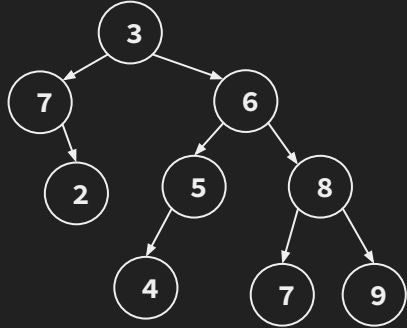
- All Children are larger

Completeness Property

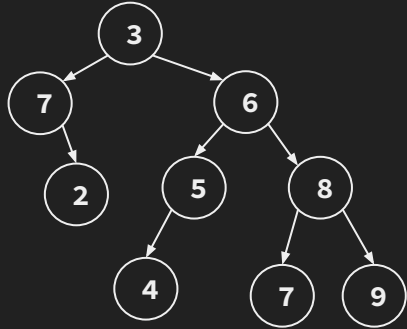
- Tree has no “gaps” (left-packed)



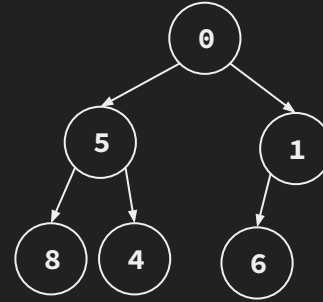
Does it Heap?



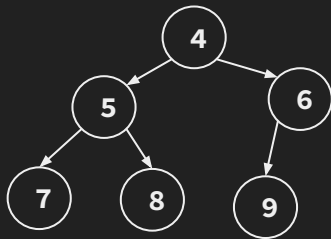
Does it Heap?



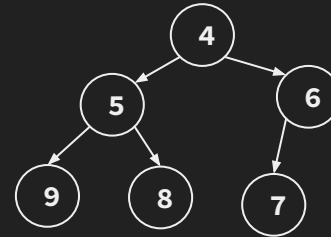
No - fails both invariants



No, fails min-heap invariant.

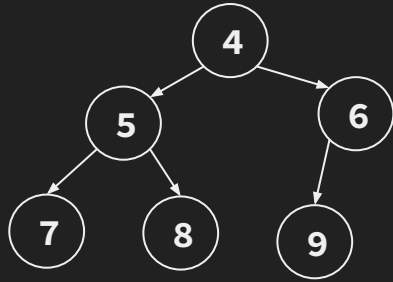


Yes!



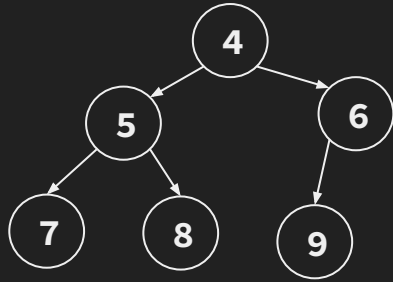
Yes!

Heap Properties



- peek - where's the element with the smallest priority?

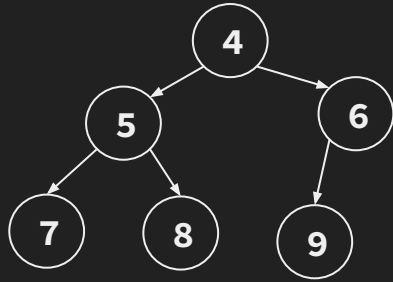
Heap Properties



- peek - where's the element with the smallest priority?

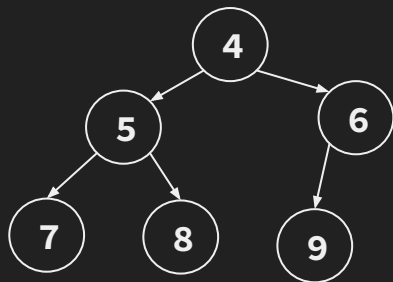
Top of the heap!

Heap Properties



- peek - where's the element with the smallest priority?
Top of the heap!
- Heap height with N items?

Heap Properties



- peek - where's the element with the smallest priority?

Top of the heap!

- Heap height with N items?

Count total # of elements in full heap with K layers
- # of elements in each layer doubles

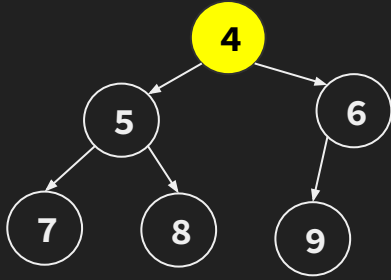
$$N \approx 2^0 + 2^1 + \dots + 2^{K-1} = 2^K - 1.$$

$$\lg N \approx \lg 2^K - 1 \approx \lg 2^K = K$$

Theory-Implementation: removeMin

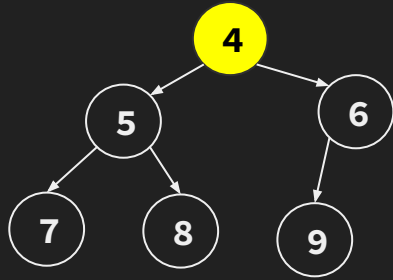
removeMin

Find min

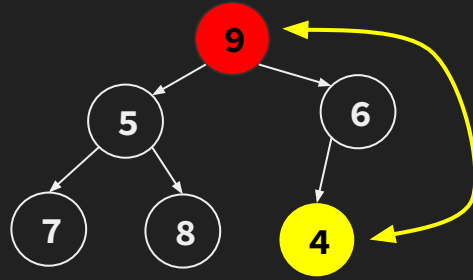


removeMin

Find min

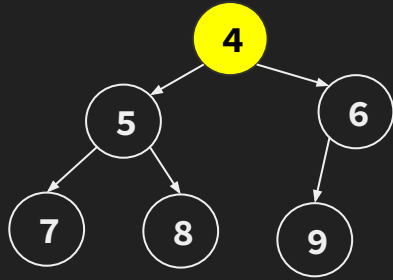


Swap with last child

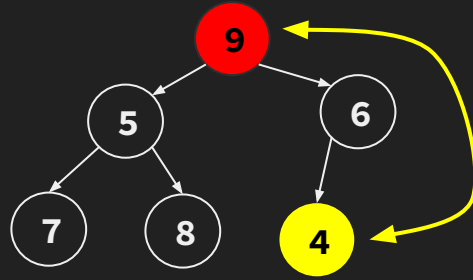


removeMin

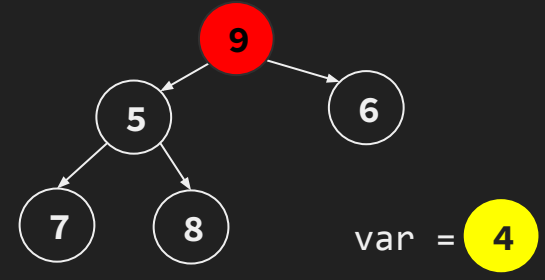
Find min



Swap with last child

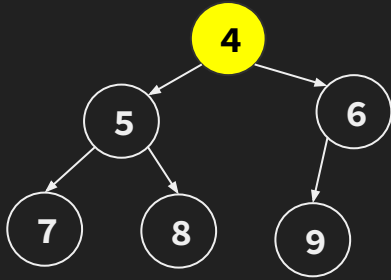


Delete and save previous min

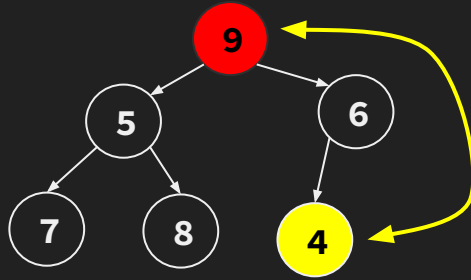


removeMin

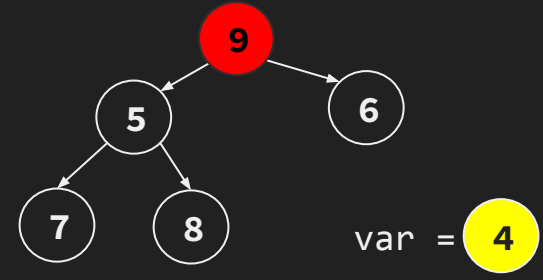
Find min



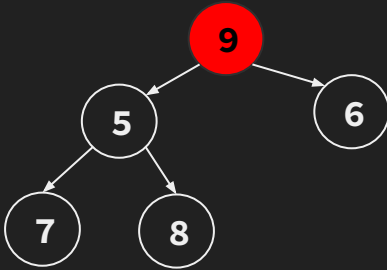
Swap with last child



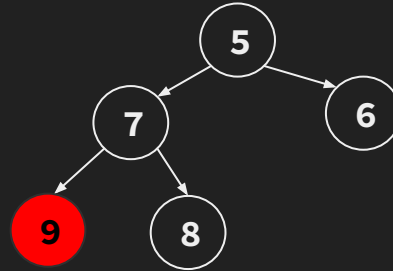
Delete and save previous min



“Bubble Down” to fix heap invariant

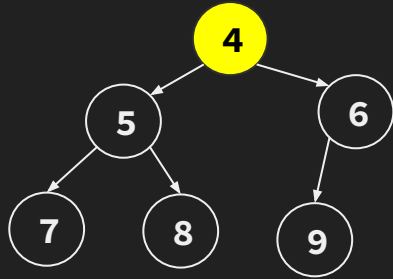


???

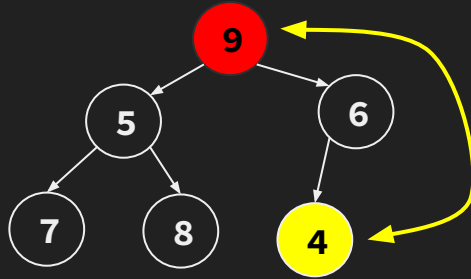


removeMin

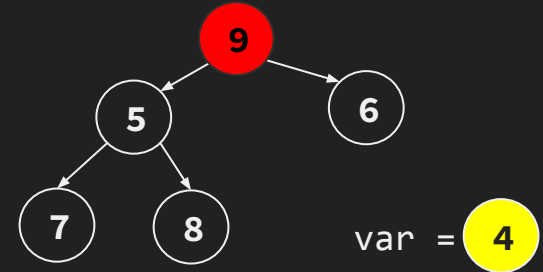
Find min



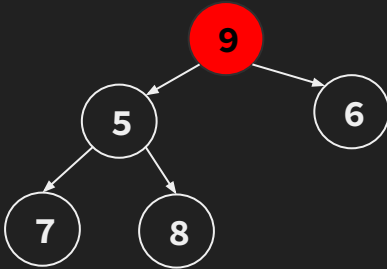
Swap with last child



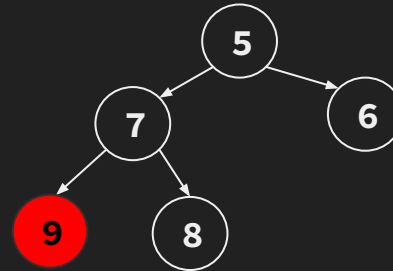
Delete and save previous min



“Bubble Down” to fix heap invariant



???

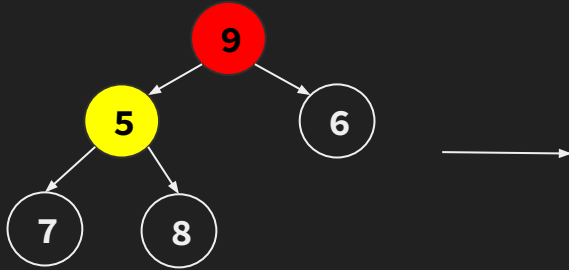


Return min

return 4

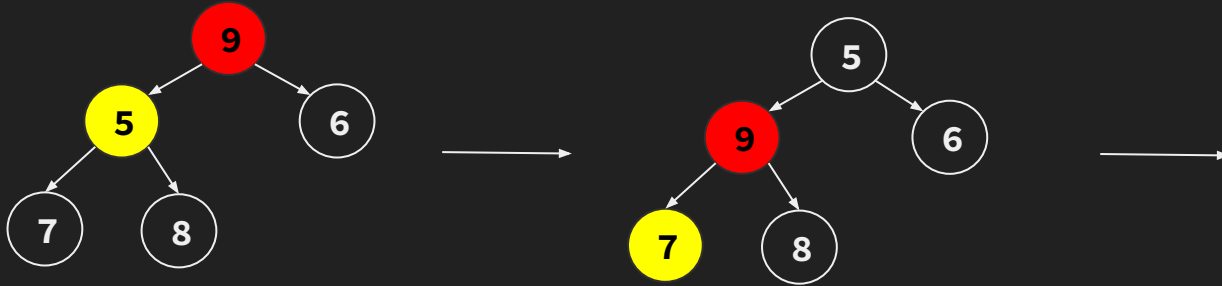
“Bubble Down”

```
bubbleDown(node) {  
    while (node.priority is greater than either child) {  
        Swap data with smaller child  
    }  
}
```



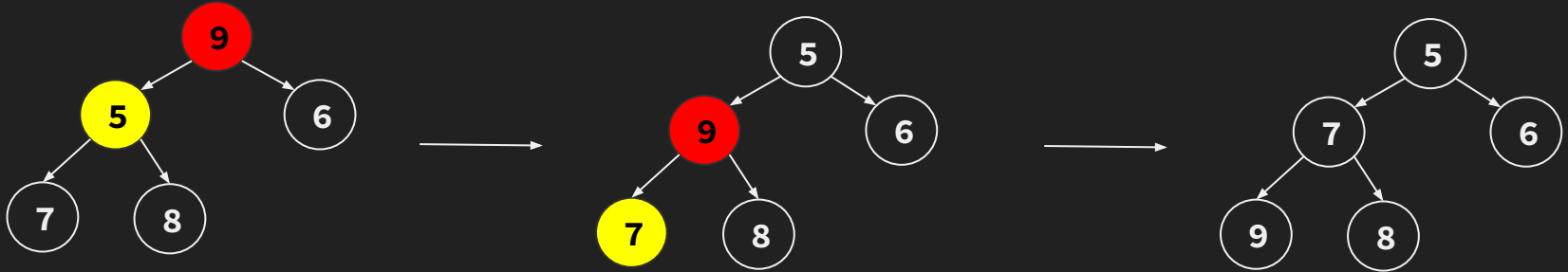
“Bubble Down”

```
bubbleDown(node) {  
  while (node.priority is greater than either child) {  
    Swap data with smaller child  
  }  
}
```



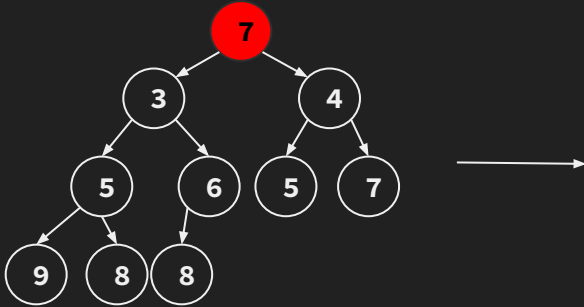
“Bubble Down”

```
bubbleDown(node) {  
  while (node.priority is greater than either child) {  
    Swap data with smaller child  
  }  
}
```



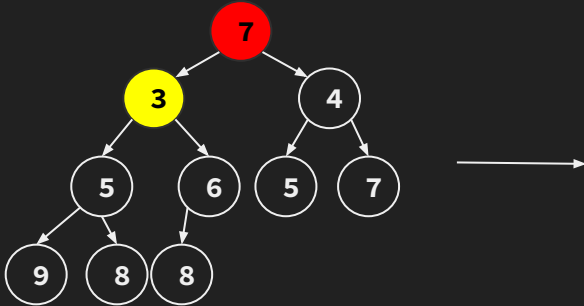
“Bubble Down”

```
bubbleDown(node) {  
    while (node.priority is greater than either child) {  
        Swap data with smaller child  
    }  
}
```



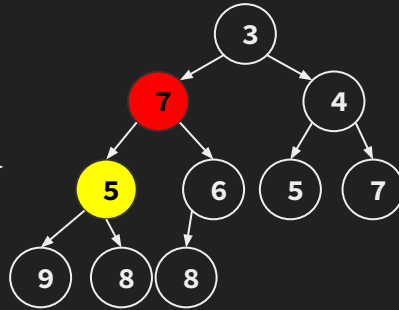
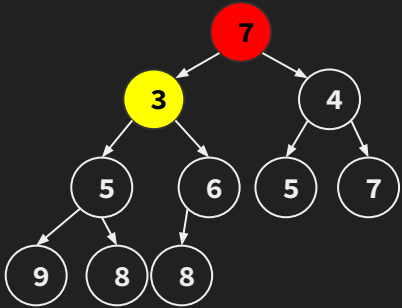
“Bubble Down”

```
bubbleDown(node) {  
    while (node.priority is greater than either child) {  
        Swap data with smaller child  
    }  
}
```



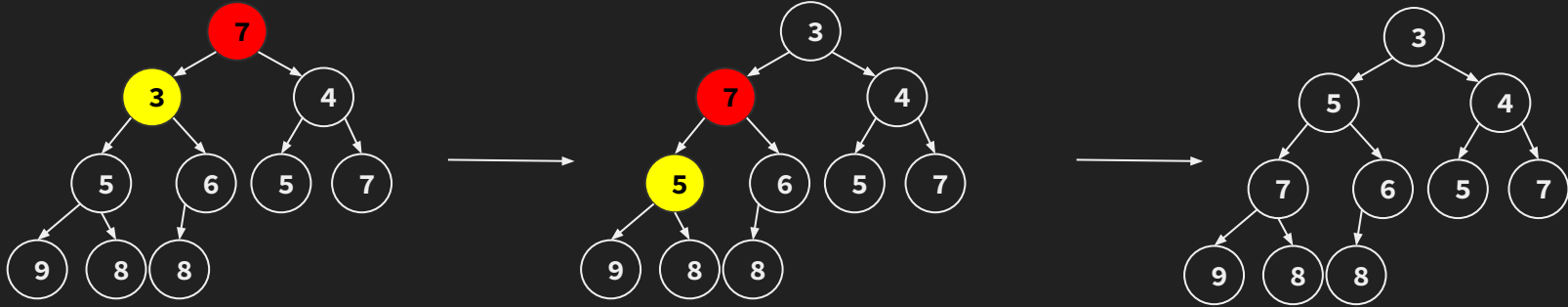
“Bubble Down”

```
bubbleDown(node) {  
    while (node.priority is greater than either child) {  
        Swap data with smaller child  
    }  
}
```



“Bubble Down”

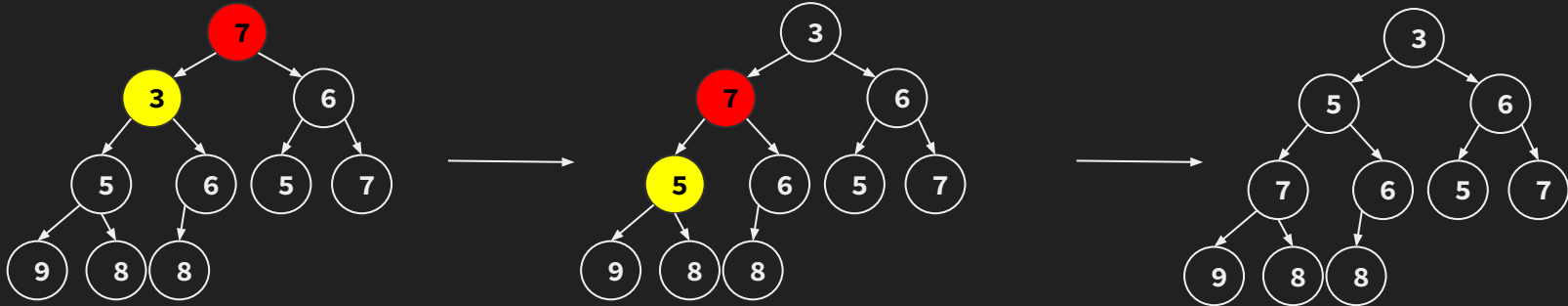
```
bubbleDown(node) {  
  while (node.priority is greater than either child) {  
    Swap data with smaller child  
  }  
}
```



Runtime?

“Bubble Down”

```
bubbleDown(node) {  
    while (node.priority is greater than either child) {  
        Swap data with smaller child  
    }  
}
```



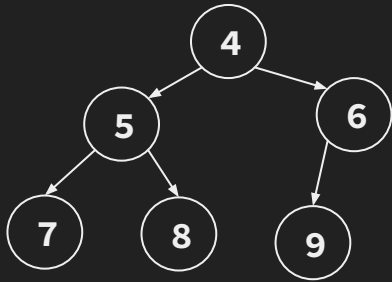
Runtime? Worst case, swap through every layer. Heap height is $\sim \lg n$, so runtime is $O(\lg n)$.

Theory-Implementation: insert

insert

`insert(1)`.

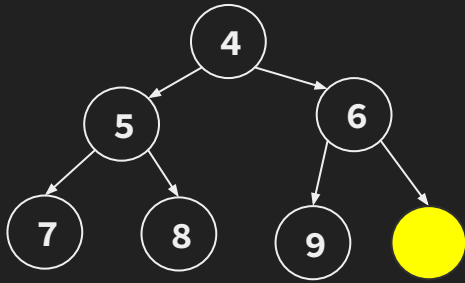
Where should the new item go first?



insert

insert(1).

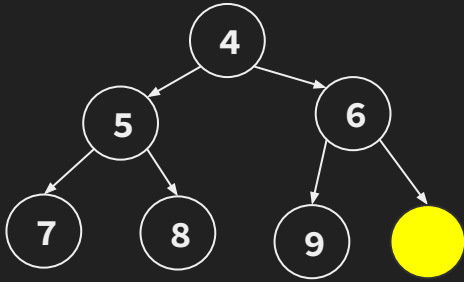
Where should the new item go first?



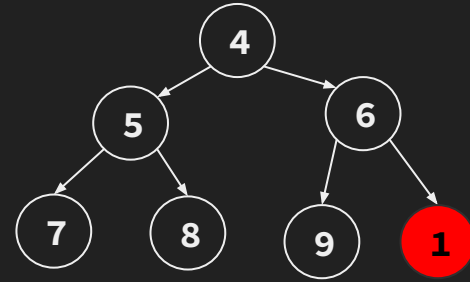
insert

insert(1).

Where should the new item go first?



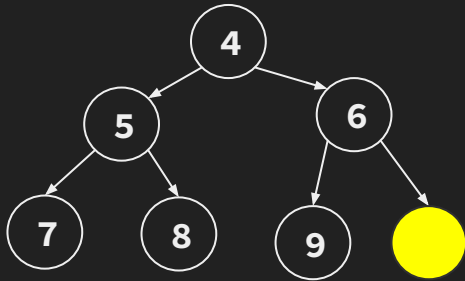
Fill last "hole" with 1.



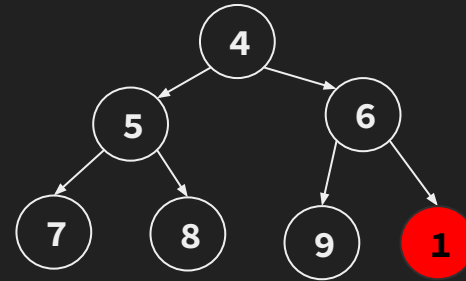
insert

insert(1).

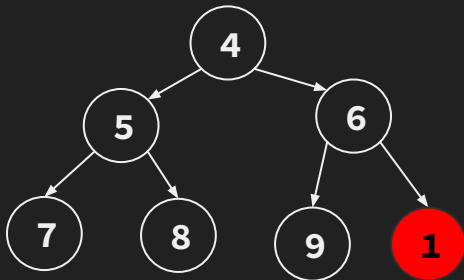
Where should the new item go first?



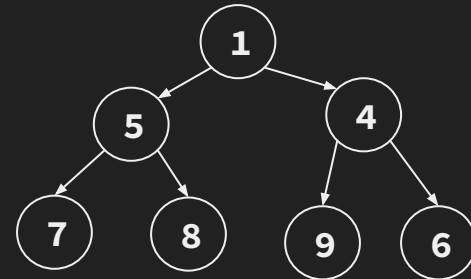
Fill last "hole" with 1.



"Bubble Up" to fix heap invariant

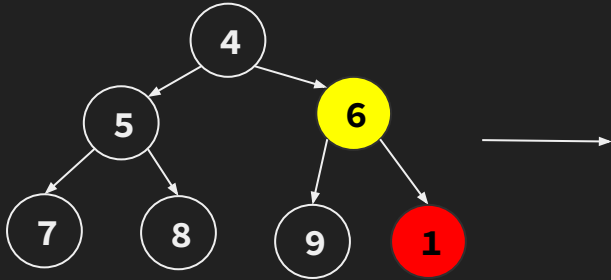


???



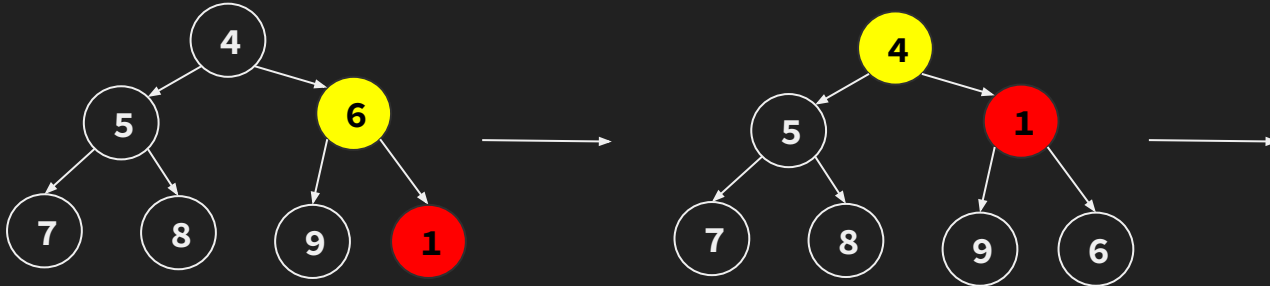
“Bubble Up”

```
bubbleUp(node) {  
    while (node.priority is smaller than parent) {  
        Swap data with parent  
    }  
}
```



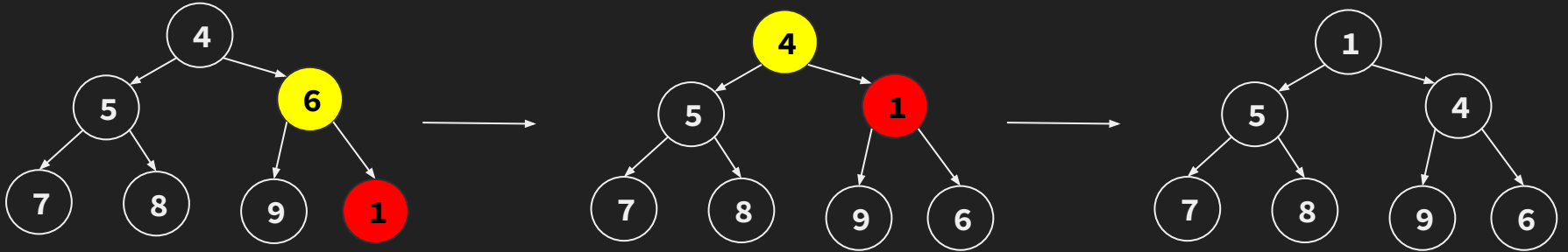
“Bubble Up”

```
bubbleUp(node) {  
  while (node.priority is smaller than parent) {  
    Swap data with parent  
  }  
}
```



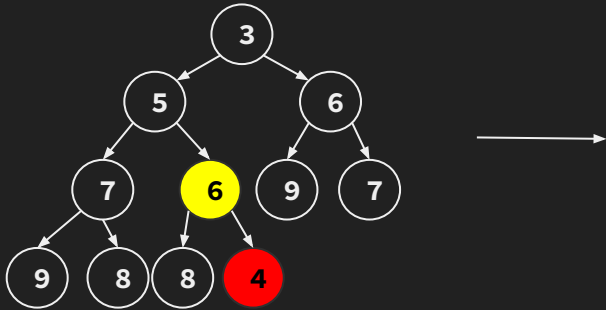
“Bubble Up”

```
bubbleUp(node) {  
  while (node.priority is smaller than parent) {  
    Swap data with parent  
  }  
}
```



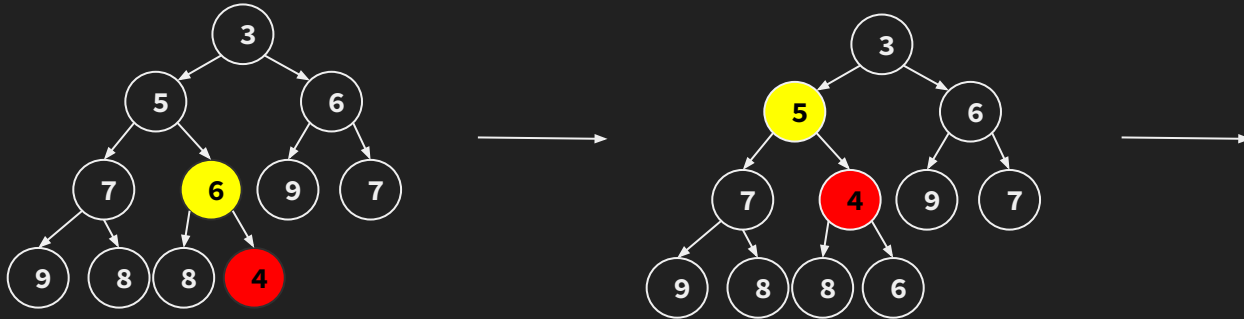
“Bubble Up”

```
bubbleUp(node) {  
  while (node.priority is smaller than parent) {  
    Swap data with parent  
  }  
}
```



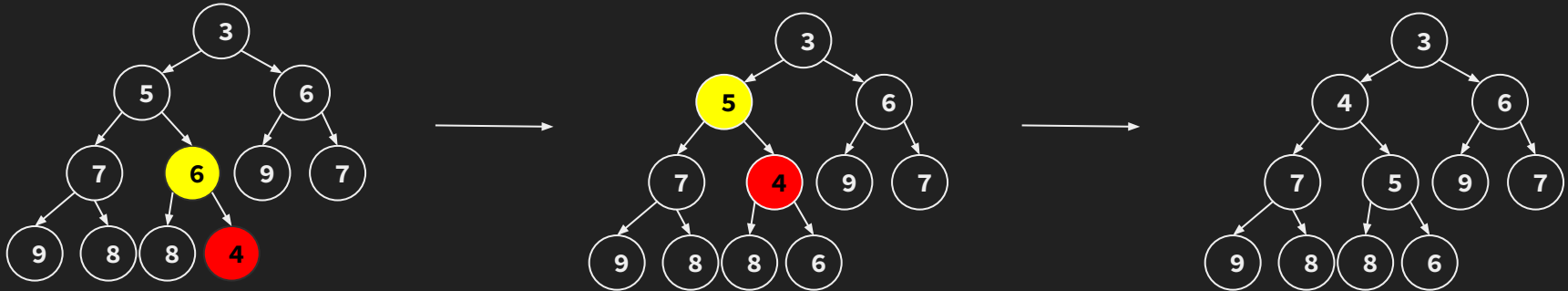
“Bubble Up”

```
bubbleUp(node) {  
  while (node.priority is smaller than parent) {  
    Swap data with parent  
  }  
}
```



“Bubble Up”

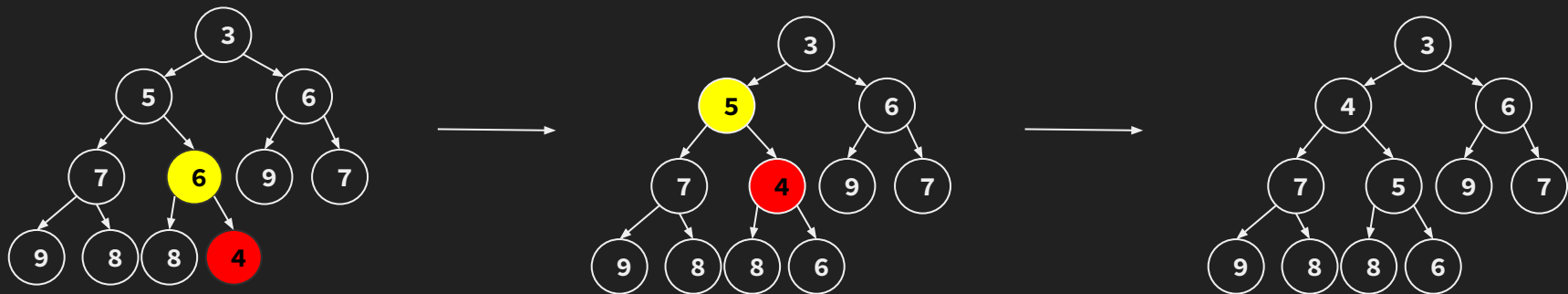
```
bubbleUp(node) {  
  while (node.priority is smaller than parent) {  
    Swap data with parent  
  }  
}
```



Runtime?

“Bubble Up”

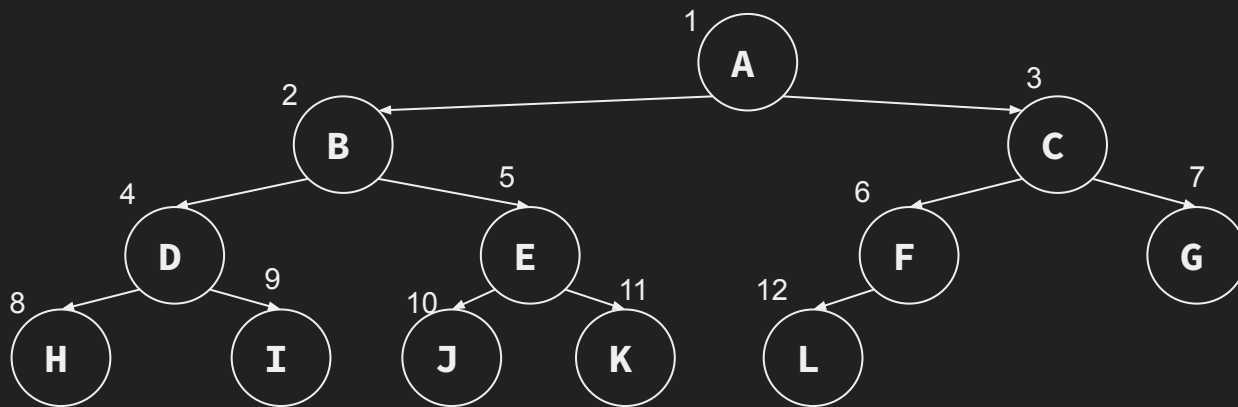
```
bubbleUp(node) {  
    while (node.priority is smaller than parent) {  
        Swap data with parent  
    }  
}
```



Runtime? Worst case, swap through every layer. Heap height is $\sim \lg n$, so runtime is $O(\lg n)$.

Implementation: array

Heap: Implementation

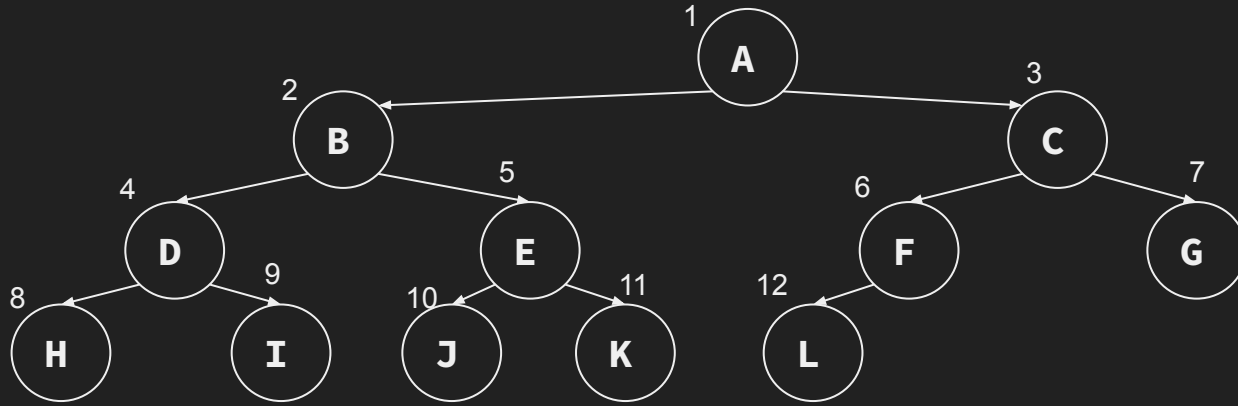


Fill an array in **level-order** of the tree (starting from index 1):

heap:

\emptyset	A	B	C	D	E	F	G	H	I	J	K	L	\emptyset	\emptyset	\emptyset
h[0]	h[1]	h[2]	h[3]	h[4]	h[5]	h[6]	h[7]	h[8]	h[9]	h[10]	h[11]	h[12]	h[13]	h[14]	h[15]

Heap: Implementation



Fill an array in **level-order** of the tree (starting from index 1):

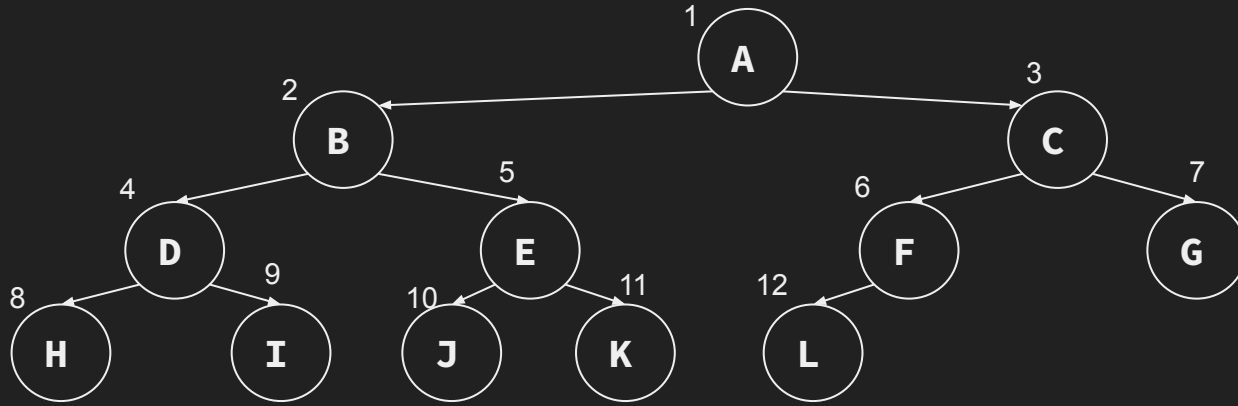
heap:

\emptyset	A	B	C	D	E	F	G	H	I	J	K	L	\emptyset	\emptyset	\emptyset
h[0]	h[1]	h[2]	h[3]	h[4]	h[5]	h[6]	h[7]	h[8]	h[9]	h[10]	h[11]	h[12]	h[13]	h[14]	h[15]

Node at index i - getting its...

- Parent?
- Left child?
- Right child?

Heap: Implementation



Fill an array in **level-order** of the tree (starting from index 1):

heap:

\emptyset	A	B	C	D	E	F	G	H	I	J	K	L	\emptyset	\emptyset	\emptyset
h[0]	h[1]	h[2]	h[3]	h[4]	h[5]	h[6]	h[7]	h[8]	h[9]	h[10]	h[11]	h[12]	h[13]	h[14]	h[15]

Node at index i - getting its...

- Parent? $i / 2$, rounding down
- Left child? $2i$
- Right child? $2i + 1$