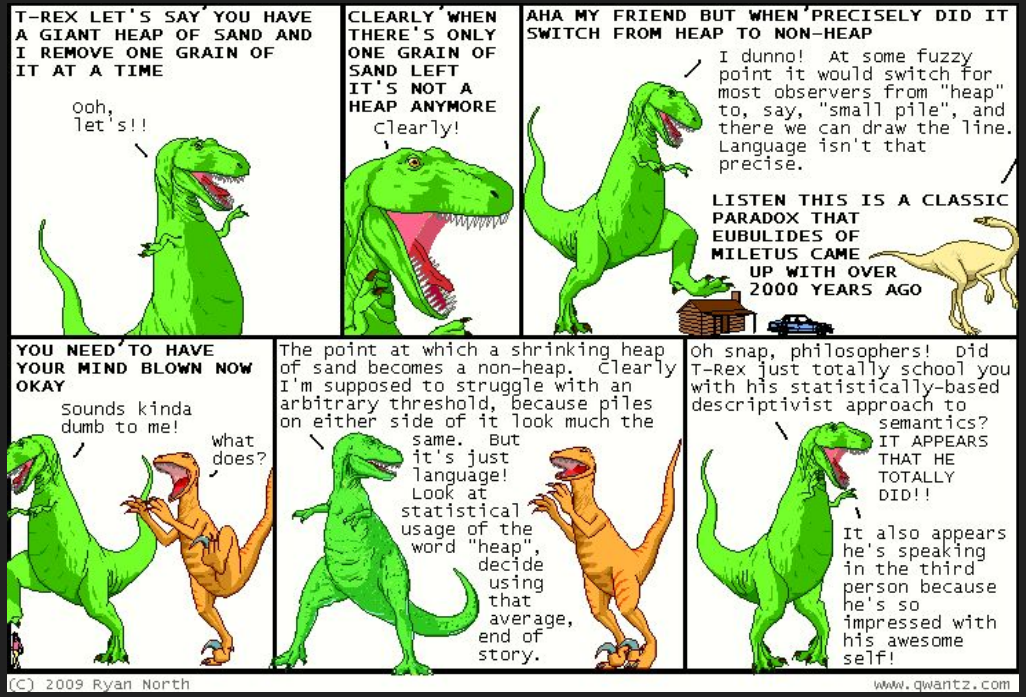
# Priority Queues & Heaps



# `IQueue` Interface

Based on *time* (recency)

| | |
|---|---|
| `enqueue(val)` | Adds `val` to the queue |
| `peek() / dequeue()` | Returns the item in the queue that was enqueued ***longest ago***. |

# `IQueue` Interface

Based on *time* (recency)

| | |
|---|---|
| `enqueue(val)` | Adds `val` to the queue |
| `peek() / dequeue()` | Returns the item in the queue that was enqueued ***longest ago***. |

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

# `IQueue` Interface

Based on *time* (recency)

| | |
|---|---|
| `enqueue(val)` | Adds `val` to the queue |
| `peek() / dequeue()` | Returns the item in the queue that was enqueued ***longest ago***. |

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

Examples:

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

# `PriorityQueue` Interface

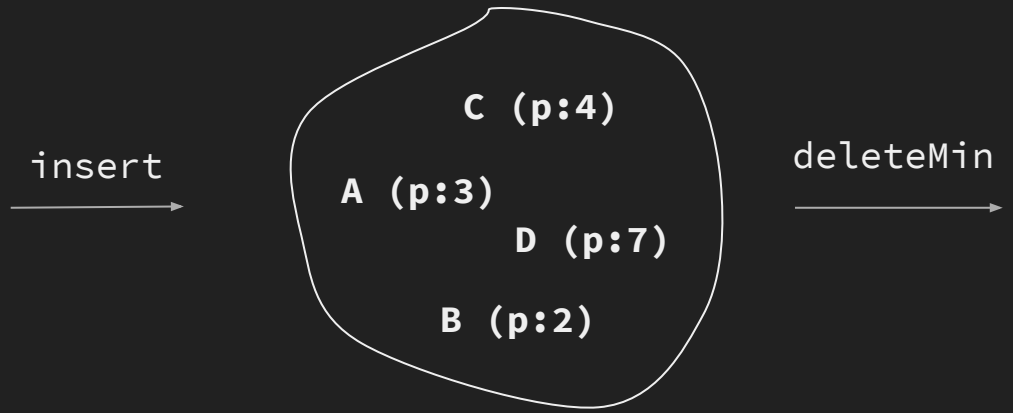
| | |
|---|---|
| `insert(val)` | Adds `val` to the queue |
| `findMin() / deleteMin()` | Returns the item in the queue with the ***highest priority***. |

- Data in `PriorityQueues` **must be comparable** (by priority)
- Highest Priority == Lowest Priority Value
- No specification on how to deal with ties

# `PriorityQueue` Interface

| | |
|---|---|
| `insert(val)` | Adds `val` to the queue |
| `findMin() / deleteMin()` | Returns the item in the queue with the ***highest priority***. |

- Highest Priority == Lowest Priority Value

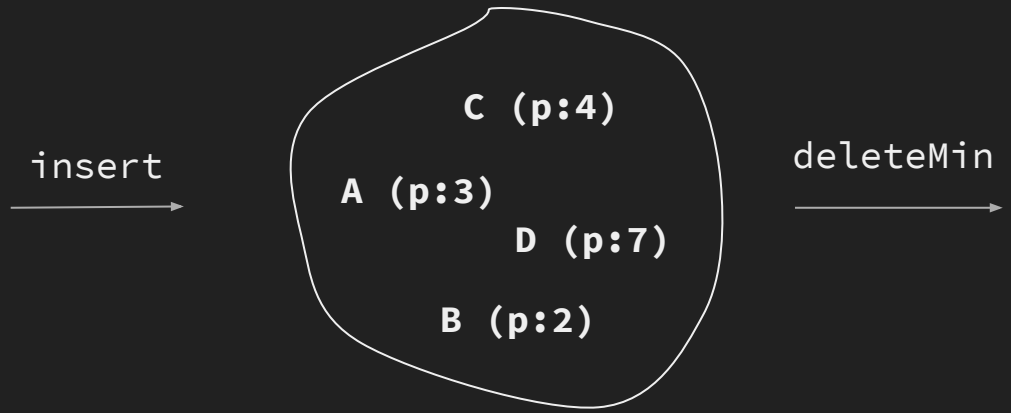



- `findMin`
- `deleteMin`
- `insert(E (p: 1))`
- `deleteMin`
- `insert(F (p:5))`
- `deleteMin`

# `PriorityQueue` Interface

| | |
|---|---|
| `insert(val)` | Adds `val` to the queue |
| `findMin() / deleteMin()` | Returns the item in the queue with the ***highest priority***. |

- Highest Priority == Lowest Priority Value




- `findMin → B`
- `deleteMin → B`
- `insert(E (p: 1))`
- `deleteMin → E`
- `insert(F (p:5))`
- `deleteMin → A`

# `PriorityQueue`: Implementing?

For each possible implementation, what is the ***worst-case*** runtime for `insert` and `deleteMin`? (Assume arrays do not need to resize.)

| | `insert` | `deleteMin` |
|---|---|---|
| Unsorted Array | | |
| Sorted Linked List | | |
| Binary Search Tree | | |

# `PriorityQueue`: Implementing?

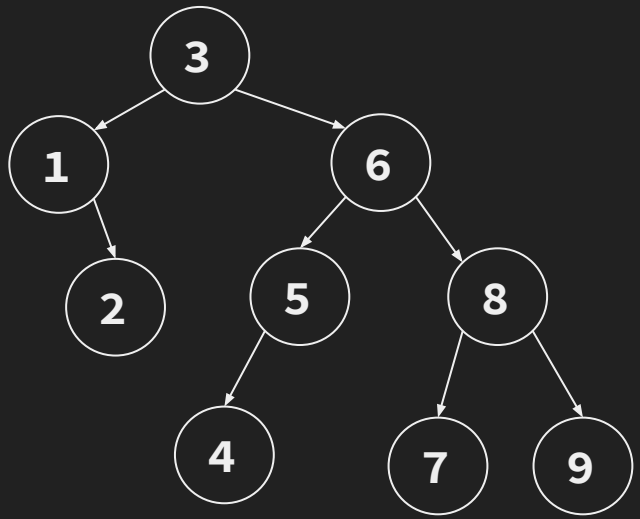
For each possible implementation, what is the ***worst-case*** runtime for `insert` and `deleteMin`? (Assume arrays do not need to resize.)

| | `insert` | `deleteMin` |
|---|---|---|
| Unsorted Array | Insert at the end - O(1) | Linear search - O(n) |
| Sorted Linked List | Linear search - O(n) | Remove front - O(1) |
| Binary Search Tree | Search - **O(n)** | findMin - **O(n)** |

# Data Structure: Heap

**BST Property (recursive invariant)**

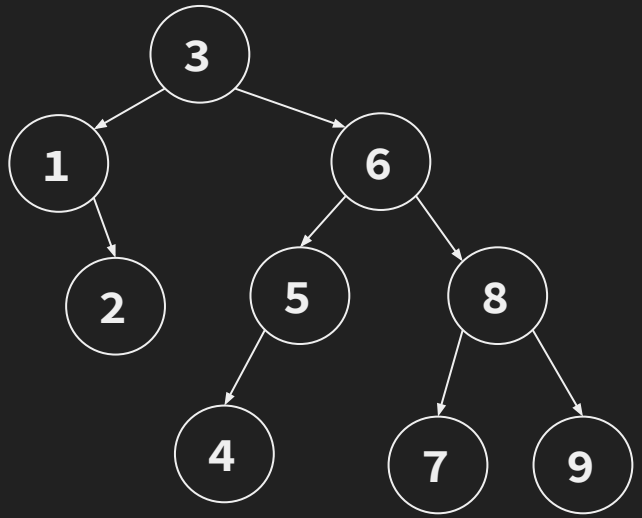
- Left Children are smaller
- Right Children are larger



# Data Structure: Heap

**BST Property (recursive invariant)**

- Left Children are smaller
- Right Children are larger

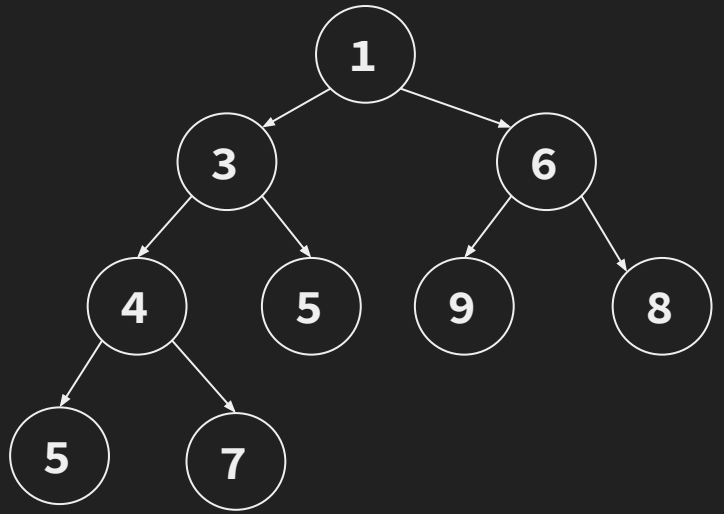



**Heap Property (recursive invariants)**

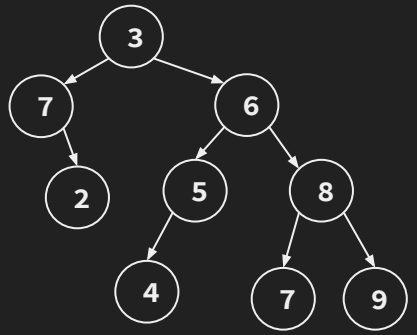
- All Children are larger

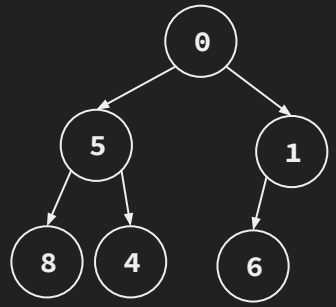
**Structure Property**

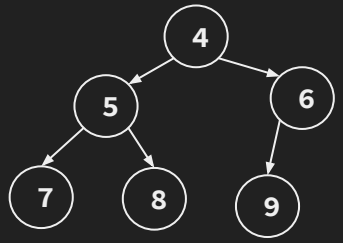
- Tree has no “gaps”

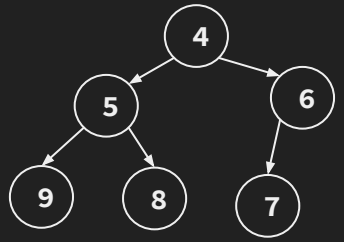


# Does it Heap?

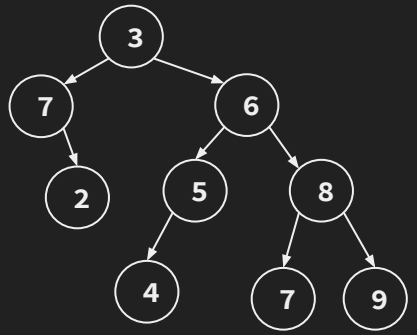


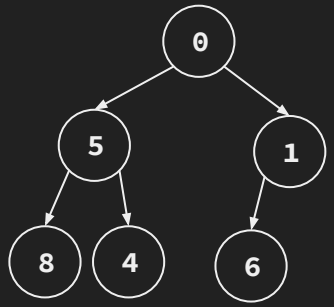



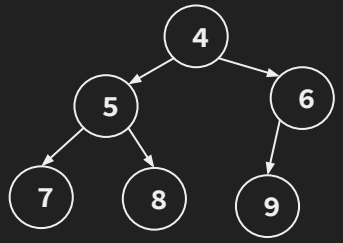
# Does it Heap?



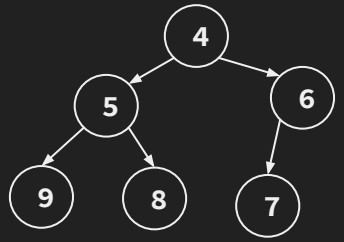

No - fails both invariants




No, fails heap invariant.

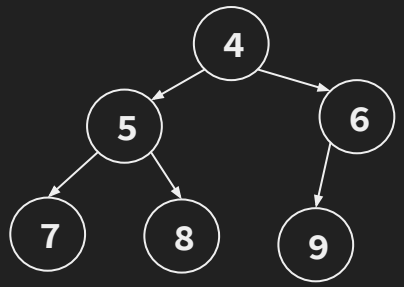



Yes!



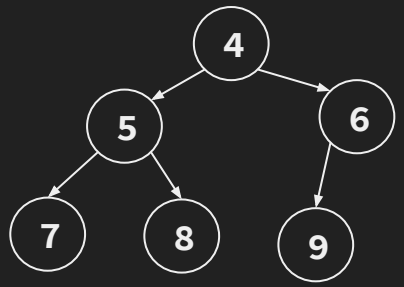

Yes!

# Heap Properties




- `findMin` - where?

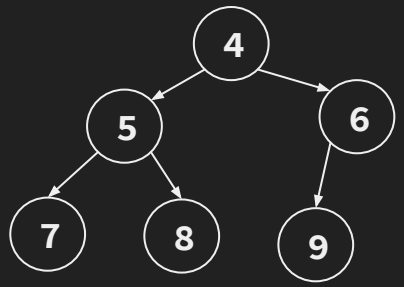
# Heap Properties




- `findMin` - where?

  Top of the heap!

# Heap Properties

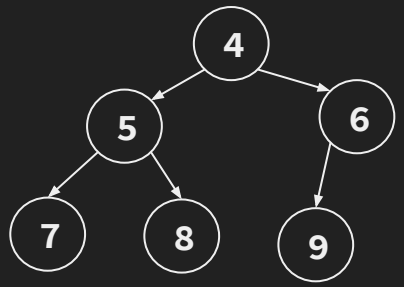



- `findMin` - where?

  Top of the heap!

- Heap height with N items?

# Heap Properties




- `findMin` - where?

  Top of the heap!

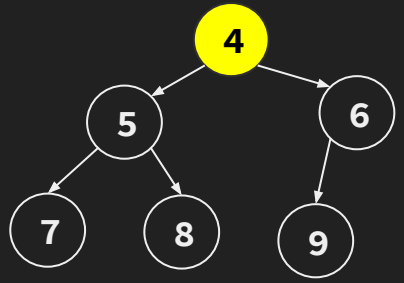
- Heap height with N items?

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

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

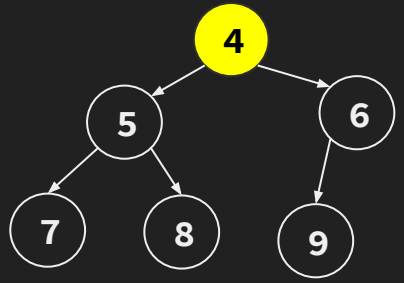
# deleteMin

Find min

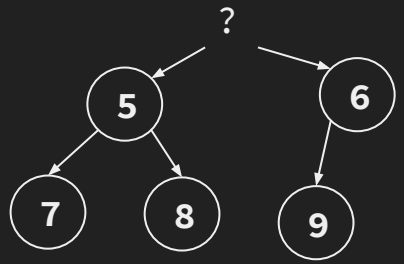


# deleteMin

Find min

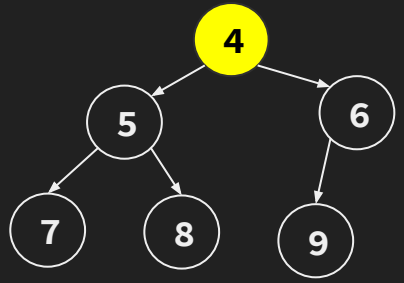



Delete min

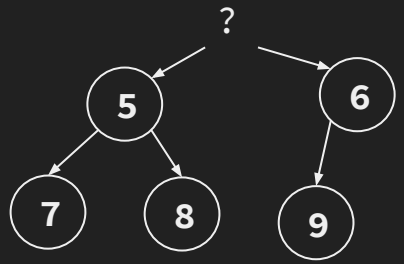


# deleteMin

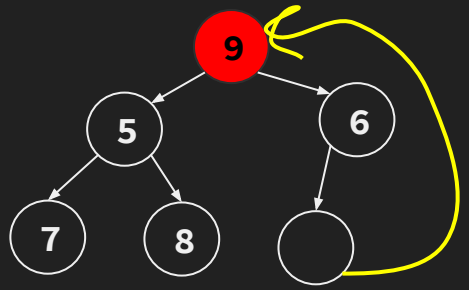
Find min




Delete min

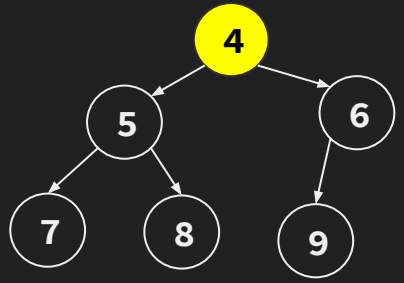



Fill hole with last child

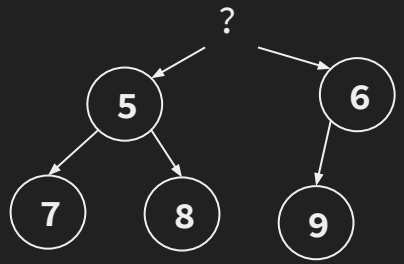


# deleteMin

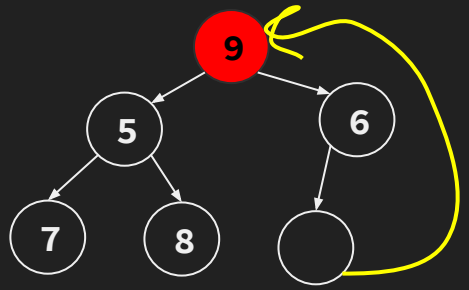
Find min



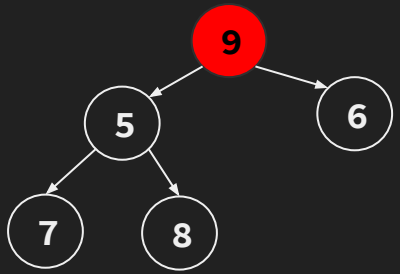

Delete min

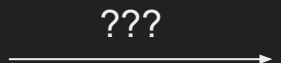


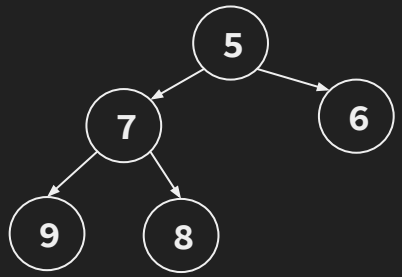

Fill hole with last child




“Percolate Down” to fix heap invariant

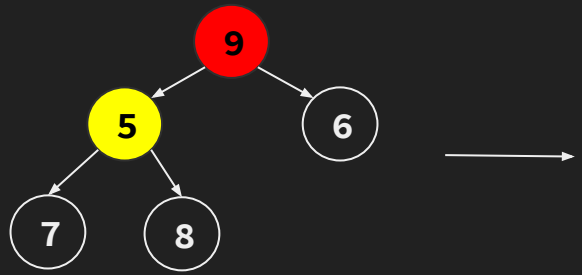


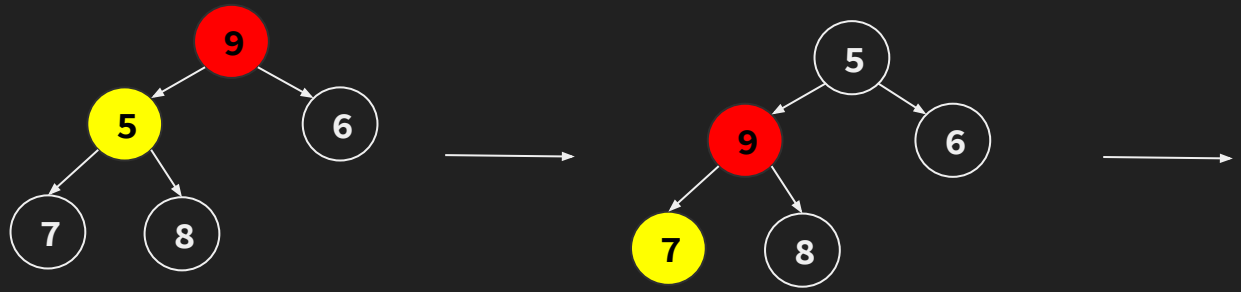
# “Percolate Down”

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



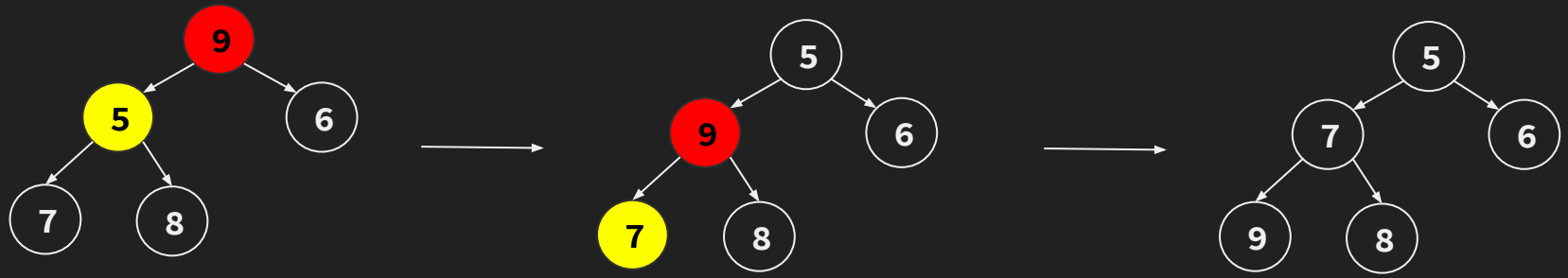
# “Percolate Down”

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



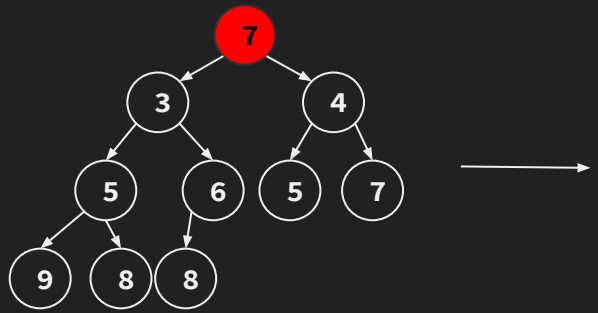
# “Percolate Down”

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



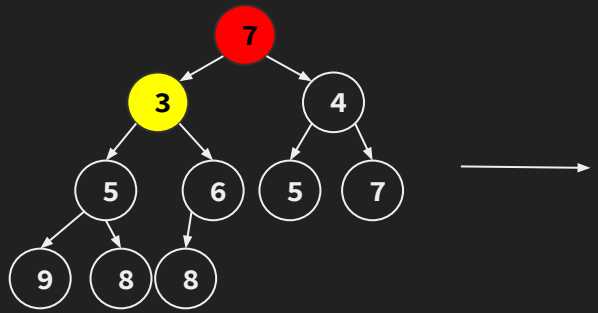
# “Percolate Down”

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



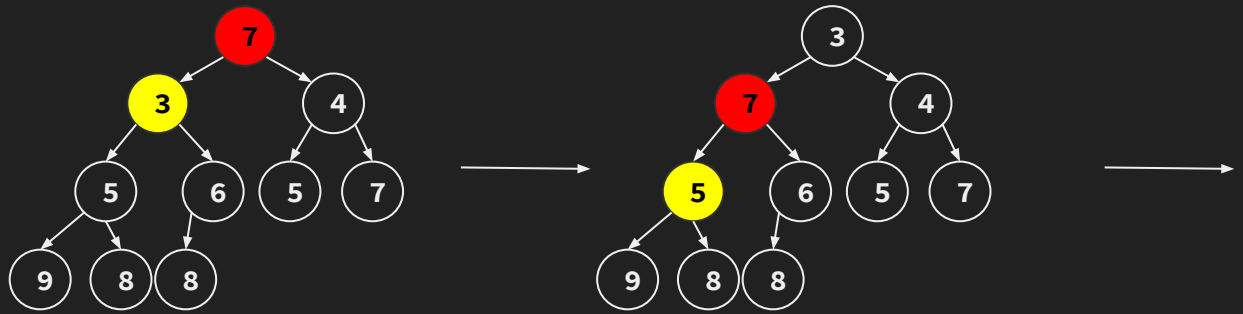
# “Percolate Down”

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



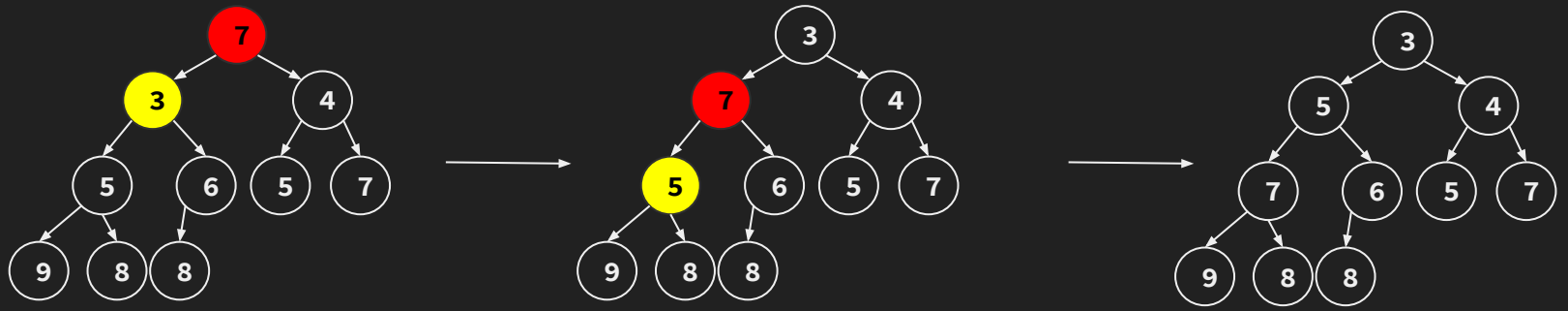
# “Percolate Down”

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



# “Percolate Down”

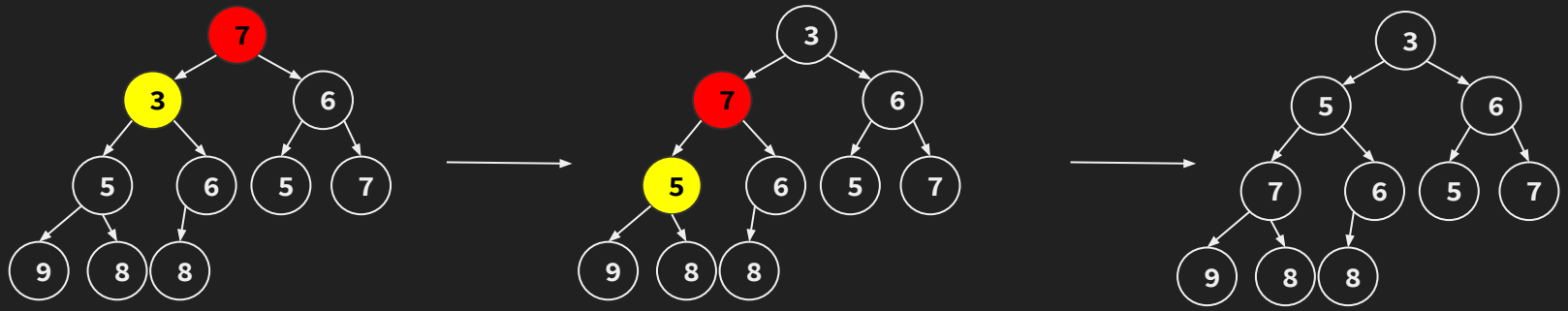
```
percolateDown(node) {
    while (node.data is greater than either child) {
        Swap data with smaller child
    }
}
```




Runtime?

# “Percolate Down”

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

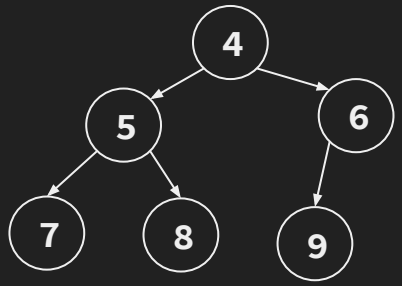



Runtime? Worst case, swap through every layer. Heap height is ~lg n, so runtime is O(lg n).

# insert

`insert(1).`

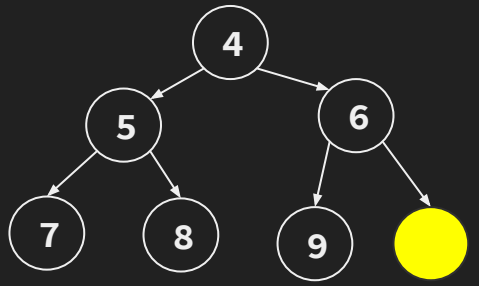
Where should the new item go first?



# insert

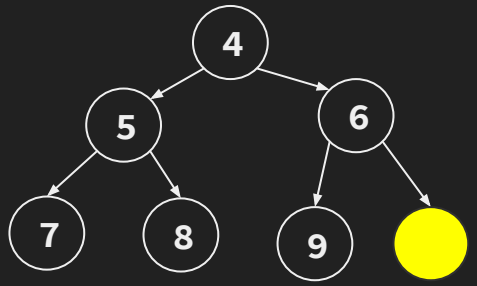
`insert(1).`

Where should the new item go first?

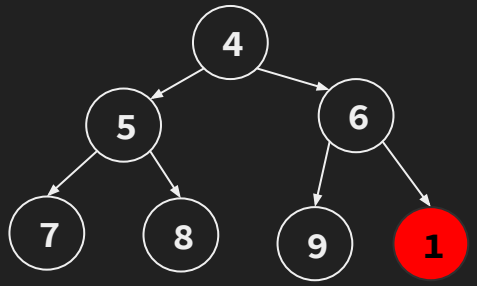


# insert

`insert(1).`
Where should the new item go first?

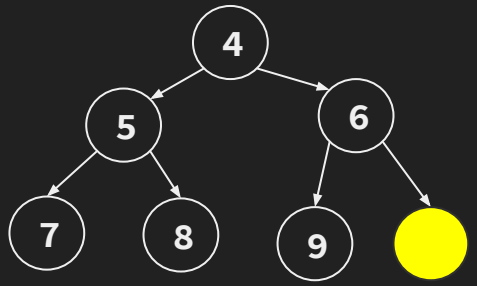



Fill last “hole” with 1.

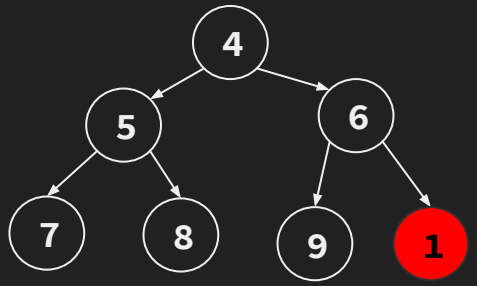


# insert

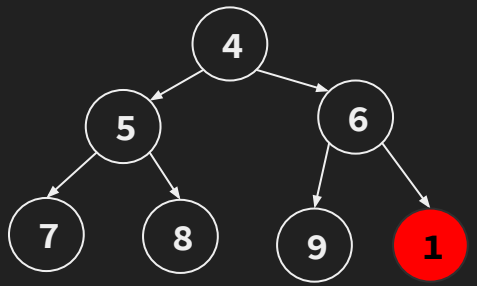
`insert(1).`
Where should the new item go first?



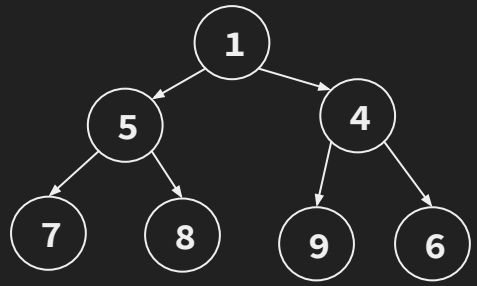

Fill last “hole” with 1.




“Percolate Up” to fix heap invariant

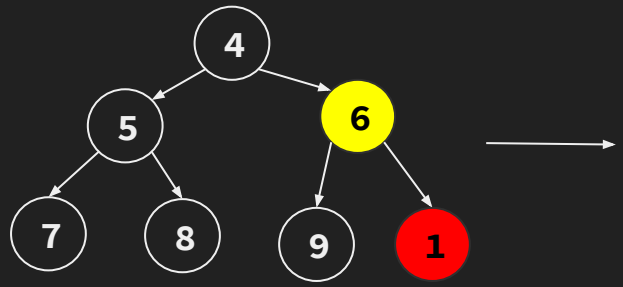



???



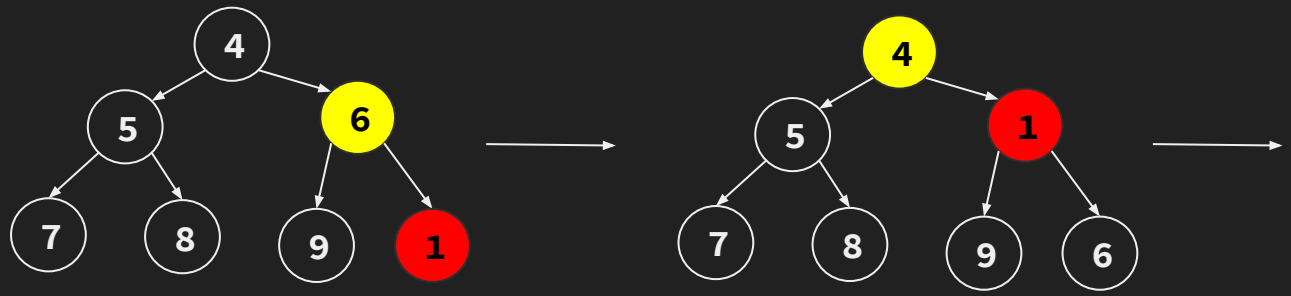
# “Percolate Up”

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



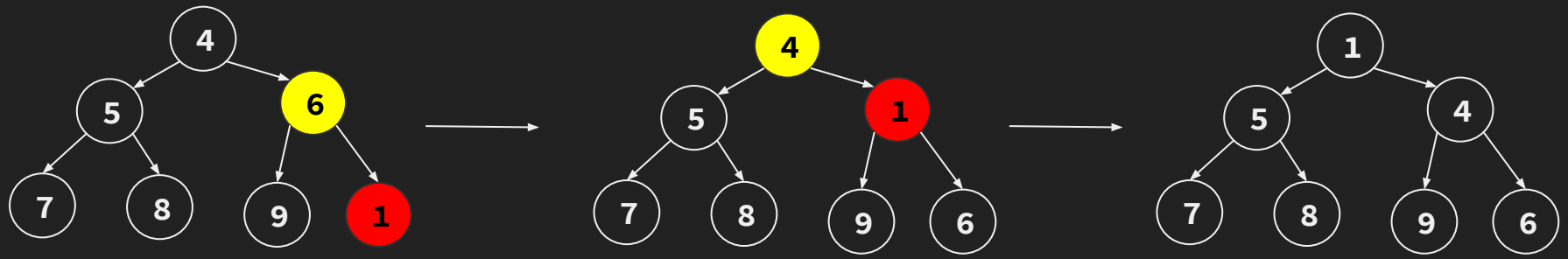
# “Percolate Up”

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



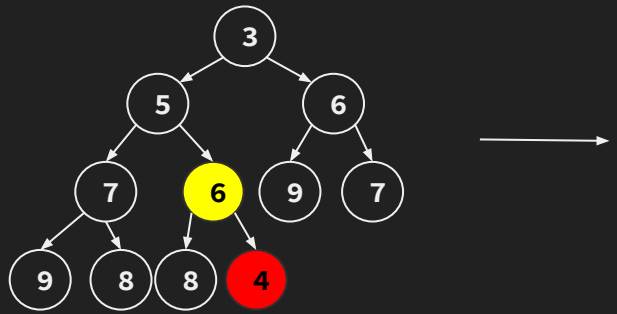
# “Percolate Up”

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



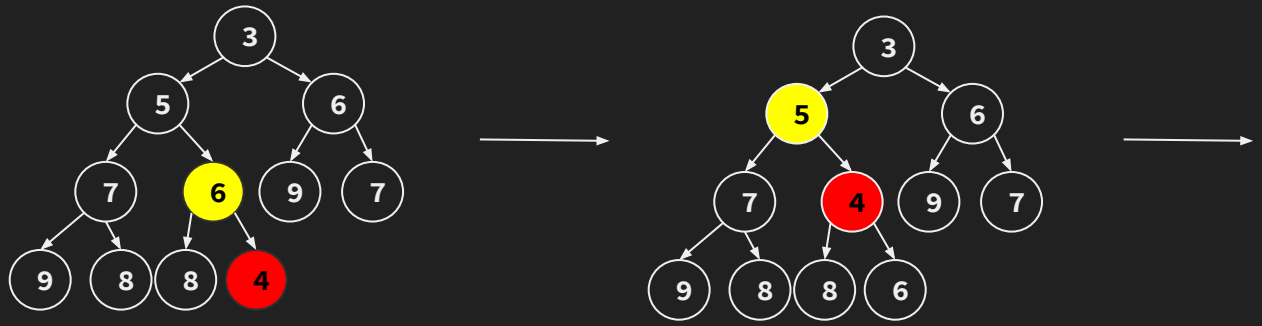
# “Percolate Up”

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



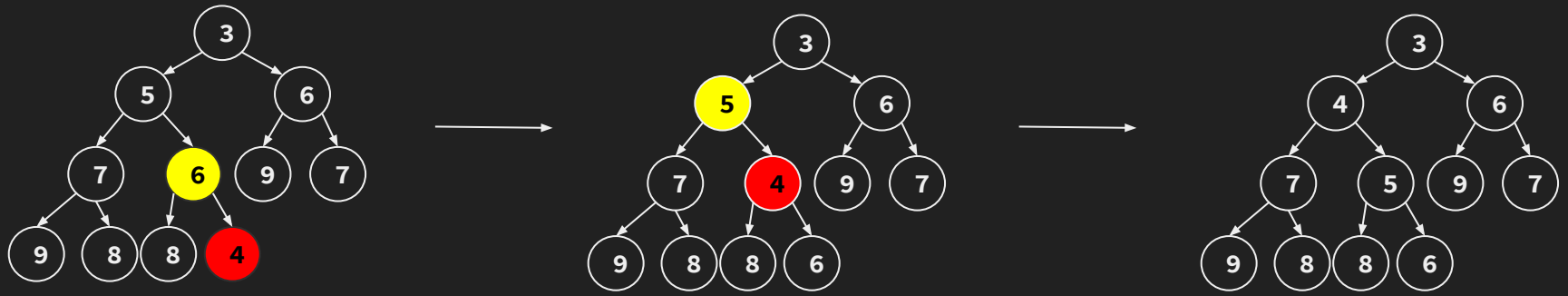
# “Percolate Up”

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



# “Percolate Up”

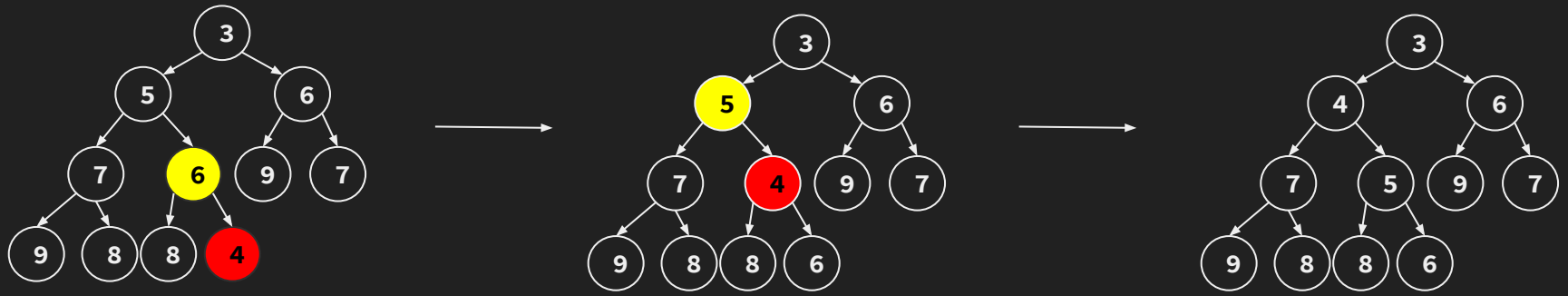
```
percolateUp(node) {
    while (node.data is smaller than parent) {
        Swap data with parent
    }
}
```




Runtime?

# “Percolate Up”

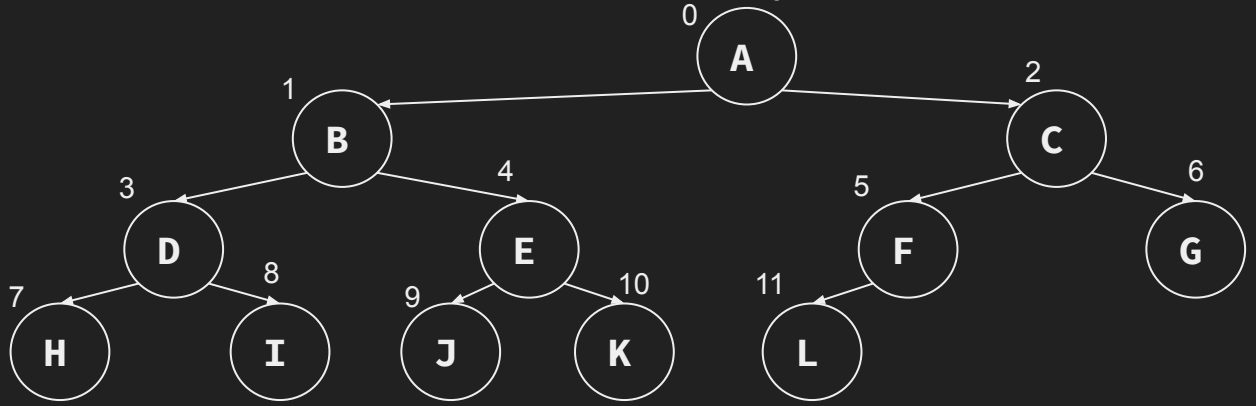
```
percolateUp(node) {
    while (node.data is smaller than parent) {
        Swap data with parent
    }
}
```




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

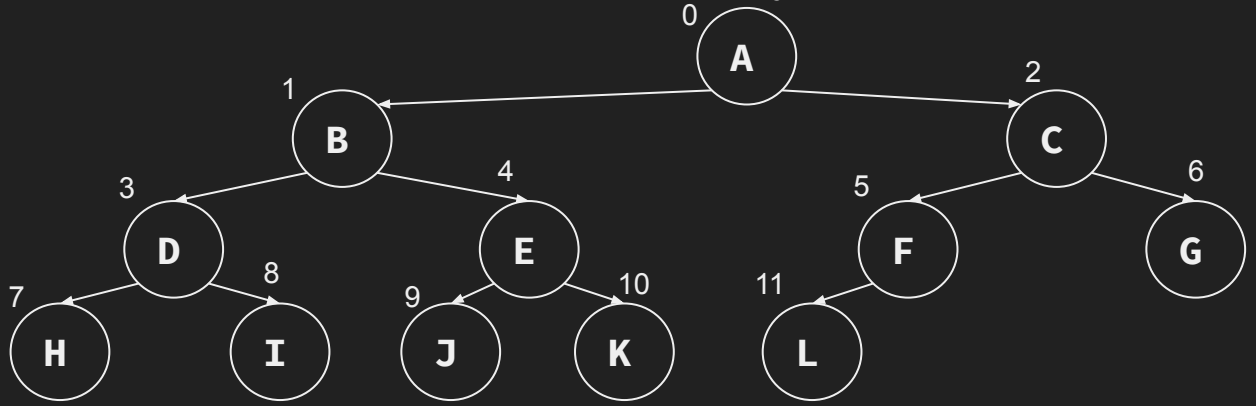
# Heap: Implementation

Why did we require the “structure property” (tree is *complete*)?



# Heap: Implementation

Why did we require the “structure property” (tree is *complete*)?



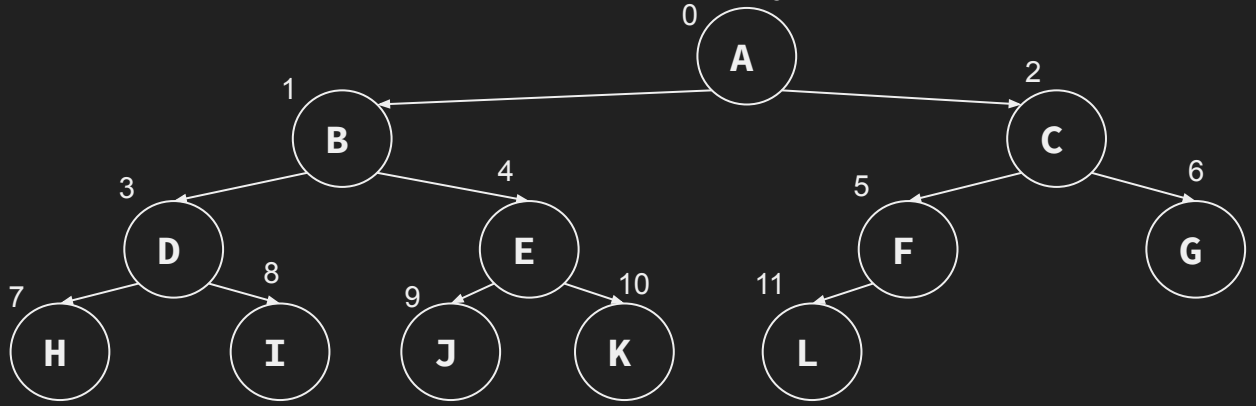

Fill an array in **level-order** of the tree:

heap:

| A | B | C | D | E | F | G | H | I | J | K | L | ∅ | ∅ | ∅ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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] |

# Heap: Implementation

Why did we require the “structure property” (tree is *complete*)?




Fill an array in **level-order** of the tree:

heap:

| A | B | C | D | E | F | G | H | I | J | K | L | ∅ | ∅ | ∅ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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] |

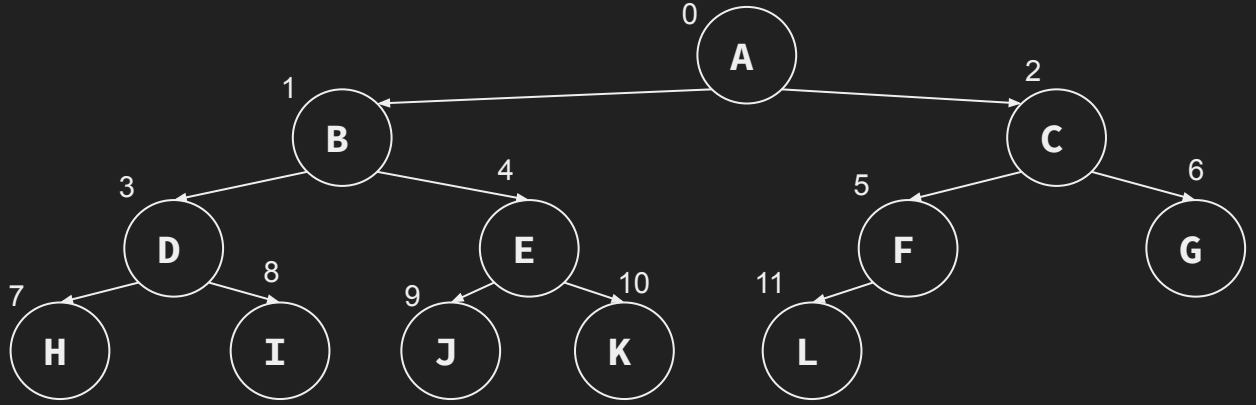
Node at index *i* - getting its…

- Parent?

- Left child?
- Right child?

# Heap: Implementation

Why did we require the “structure property” (tree is *complete*)?




Fill an array in **level-order** of the tree:

heap:

| A | B | C | D | E | F | G | H | I | J | K | L | ∅ | ∅ | ∅ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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] |

Node at index *i* - getting its…

- Parent? 3 → 1; 4 → 1; 10 → 4; 9 → 4;  1 → 0
  ~ n/2. Exactly (n-1) / 2
- Left child? 2 (n+1) - 1
- Right child? 2 (n+1)