CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
Capen 208

Lecture 23 - 24
Heaps

Priority Queue ADT

PriorityQueue ADT

`void add(T value)`

Insert **`value`** into the priority queue

`T poll()`

Remove and return the **highest priority** value from the priority queue

`T peek()`

Return the **highest priority** value from the priority queue

PriorityQueue ADT

`void add(T value)`

Insert **`value`** into the priority queue

`T poll()`

Remove and return the **highest priority** value from the priority queue

`T peek()`

Return the **highest priority** value from the priority queue

To use a **`PriorityQueue`** we must give it an ordering!

- Only need a partial ordering (there may not be a unique highest priority value)
- Removing elements will create a topological sort (by breaking ties arbitrarily)

Note: In Java the default ordering is smaller values have higher priority

Usage Example

Understanding the behavior of a priority queue:

Assume **`pq`** is a **`PriorityQueue`** where priority is given to smaller values (like Java)

`pq.add(3)`

Usage Example

Understanding the behavior of a priority queue:

Assume **`pq`** is a **`PriorityQueue`** where priority is given to smaller values (like Java)

```
pq.add(3)
pq.add(7)
```

Usage Example

Understanding the behavior of a priority queue:

Assume **`pq`** is a **`PriorityQueue`** where priority is given to smaller values (like Java)

```
pq.add(3)
pq.add(7)
pq.add(2)
```

Usage Example

Understanding the behavior of a priority queue:

Assume **`pq`** is a **`PriorityQueue`** where priority is given to smaller values (like Java)

```
pq.add(3)
pq.add(7)
pq.add(2)
pq.add(5)
```

Usage Example

Understanding the behavior of a priority queue:

Assume **`pq`** is a **`PriorityQueue`** where priority is given to smaller values (like Java)

```
pq.add(3)
pq.add(7)
pq.add(2)
pq.add(5)
pq.poll()  // what gets removed??
```

Usage Example

Understanding the behavior of a priority queue:

Assume **`pq`** is a **`PriorityQueue`** where priority is given to smaller values (like Java)

```
pq.add(3)
pq.add(7)
pq.add(2)
pq.add(5)
pq.poll()  // removes and returns 2
```

Usage Example

Understanding the behavior of a priority queue:

Assume **pq** is a **PriorityQueue** where priority is given to smaller values (like Java)

```
pq.add(3)
pq.add(7)
pq.add(2)
pq.add(5)
pq.poll()  // removes and returns 2
pq.peek()  // what gets returned??
```

Usage Example

Understanding the behavior of a priority queue:

Assume **`pq`** is a **`PriorityQueue`** where priority is given to smaller values (like Java)

```
pq.add(3)
pq.add(7)
pq.add(2)
pq.add(5)
pq.poll()  // removes and returns 2
pq.peek()  // returns 3
```

Usage Example

Understanding the behavior of a priority queue:

Assume **`pq`** is a **`PriorityQueue`** where priority is given to smaller values (like Java)

```
pq.add(3)
pq.add(7)
pq.add(2)
pq.add(5)
pq.poll()  // removes and returns 2
pq.peek()  // returns 3
pq.poll() // removes and returns 3
```

Usage Example

Understanding the behavior of a priority queue:

Assume **pq** is a **PriorityQueue** where priority is given to smaller values (like Java)

```
pq.add(3)
pq.add(7)
pq.add(2)
pq.add(5)
pq.poll()  // removes and returns 2
pq.peek()  // returns 3
pq.poll() // removes and returns 3
pq.poll() // removes and returns 5
```

Usage Example

Understanding the behavior of a priority queue:

Assume **pq** is a **PriorityQueue** where priority is given to smaller values (like Java)

```
pq.add(3)
pq.add(7)
pq.add(2)
pq.add(5)
pq.poll()  // removes and returns 2
pq.peek()  // returns 3
pq.poll() // removes and returns 3
pq.poll() // removes and returns 5
pq.poll() // removes and returns 7
```

Usage Example

Understanding the behavior of a priority queue:

Assume **`pq`** is a **`PriorityQueue`** where priority is given to smaller values (like Java)

```
pq.add(3)
pq.add(7)
pq.add(2)
pq.add(5)
pq.poll()  // removes and returns 2
pq.peek()  // returns 3
pq.poll() // removes and returns 3
pq.poll() // removes and returns 5
pq.poll() // removes and returns 7
```

How should the values be stored in the `PriorityQueue`?

Usage Example

Understanding the behavior of a priority queue:

Assume **`pq`** is a **`PriorityQueue`** where priority is given to smaller values (like Java)

```
pq.add(3)
pq.add(7)
pq.add(2)
pq.add(5)
pq.poll()  // removes and returns 2
pq.peek()  // returns 3
pq.poll() // removes and returns 3
pq.poll() // removes and returns 5
pq.poll() // removes and returns 7
```

How should the values be stored in the `PriorityQueue`?

In order? **`2,3,5,7`**
Reverse order? **`7,5,3,2`**
Insertion order? **`3,7,2,5`**

Usage Example

Understanding the behavior of a priority queue:

Assume **`pq`** is a **`PriorityQueue`** where priority is given to smaller values (like Java)

```
pq.add(3)
pq.add(7)
pq.add(2)
pq.add(5)
pq.poll()  // removes and returns 2
pq.peek()  // returns 3
pq.poll() // removes and returns 3
pq.poll() // removes and returns 5
pq.poll() // removes and returns 7
```

How should the values be stored in the `PriorityQueue`?

In order? **`2,3,5,7`**
Reverse order? **`7,5,3,2`**
Insertion order? **`3,7,2,5`**

Any of these could work!!!

Remember: The ADT doesn't specify HOW the data is stored, just what behavior is required

Lazy vs Proactive Approaches

Lazy vs Proactive

There are two mentalities we can take in this instance (and many others):

Lazy vs Proactive

There are two mentalities we can take in this instance (and many others):

Lazy: Put off work as long as you can

- Keep everything a mess until you need to find something
- Lazy has a negative connotation in english...not in CS!

Lazy vs Proactive

There are two mentalities we can take in this instance (and many others):

Lazy: Put off work as long as you can

- Keep everything a mess until you need to find something
- Lazy has a negative connotation in english...not in CS!

Proactive: Keep things nice and neat at all times

- More work up front, but pays off when you need to find something

Lazy `PriorityQueue` (Unsorted List)

Base Data Structure: `LinkedList`

`void add(T value)`

Append **`value`** to the end of the **`LinkedList`**

`T peek()/T poll()`

Do a linear search to find the smallest value

Runtimes?

Lazy `PriorityQueue` (Unsorted List)

Base Data Structure: **`LinkedList`**

`void add(T value)`

Append **`value`** to the end of the **`LinkedList`**

`T peek()/T poll()`

Do a linear search to find the smallest value

Runtimes? **`add`** is $\Theta(1)$, **`peek`** and **`poll`** are $\Theta(n)$

Proactive `PriorityQueue` (Sorted List)

Base Data Structure: **`LinkedList`** kept in sorted order

`void add(T value)`

Insert **`value`** into the **`LinkedList`**, maintaining sorted order

`T peek()/T poll()`

Return the head of the **`LinkedList`** (and remove it for **`poll`**)

Runtimes?

Proactive `PriorityQueue` (Sorted List)

Base Data Structure: **`LinkedList`** kept in sorted order

`void add(T value)`

Insert **`value`** into the **`LinkedList`**, maintaining sorted order

`T peek()/T poll()`

Return the head of the **`LinkedList`** (and remove it for **`poll`**)

Runtimes? **`add`** is $O(n)$, **`peek`** and **`poll`** are $\Theta(1)$

Lazy vs Proactive

So…which implementation is better, lazy or proactive?

Do we expect to only care about one of **`add`**/**`poll`**?

Let’s look at a real example: sorting

Sorting with a `PriorityQueue`

```
 1  Input: in_list (an unsorted list)
 2  Output: out_list (a sorted list containing all items from input)
 3
 4  pq = new PriorityQueue()
 5  for item in in_list:
 6      pq.add(item)
 7  while not pq.isEmpty():
 8      out_list.add(pq.poll())
 9
10  return out_list
```

Sorting with a `PriorityQueue`

```
 1  Input: in_list (an unsorted list)
 2  Output: out_list (a sorted list containing all items from input)
 3
 4  pq = new PriorityQueue()
 5  for item in in_list:
 6      pq.add(item)
 7  while not pq.isEmpty():
 8      out_list.add(pq.poll())
 9
10  return out_list
```

Add all items to a **`PriorityQueue`**

Sorting with a `PriorityQueue`

```
1  Input: in_list (an unsorted list)
2  Output: out_list (a sorted list containing all items from input)
3
4  pq = new PriorityQueue()
5  for item in in_list:
6      pq.add(item)
7  while not pq.isEmpty():
8      out_list.add(pq.poll())
9
10 return out_list
```

Remove all items from the **`PriorityQueue`** and append to our output list

Sorting with a `PriorityQueue`

```
 1  Input: in_list (an unsorted list)
 2  Output: out_list (a sorted list containing all items from input)
 3
 4  pq = new PriorityQueue()
 5  for item in in_list:
 6      pq.add(item)
 7  while not pq.isEmpty():
 8      out_list.add(pq.poll())
 9
10  return out_list
```

Is the output sorted?

Sorting with a `PriorityQueue`

```
1  Input: in_list (an unsorted list)
2  Output: out_list (a sorted list containing all items from input)
3
4  pq = new PriorityQueue()
5  for item in in_list:
6      pq.add(item)
7  while not pq.isEmpty():
8      out_list.add(pq.poll())
9
10 return out_list
```

Is the output sorted? Yes! The **`PriorityQueue`** ADT enforces ordered removal

Sorting with a `PriorityQueue`

```
 1  Input: in_list (an unsorted list)
 2  Output: out_list (a sorted list containing all items from input)
 3
 4  pq = new PriorityQueue()
 5  for item in in_list:
 6      pq.add(item)
 7  while not pq.isEmpty():
 8      out_list.add(pq.poll())
 9
10  return out_list
```

What is the complexity?

Sorting with a `PriorityQueue`

```
 1  Input: in_list (an unsorted list)
 2  Output: out_list (a sorted list containing all items from input)
 3
 4  pq = new PriorityQueue()
 5  for item in in_list:
 6      pq.add(item)
 7  while not pq.isEmpty():
 8      out_list.add(pq.poll())
 9
10  return out_list
```

What is the complexity? Depends on our **`PriorityQueue`** implementation...

Sorting w/Lazy PQ (Selection Sort)

Input: [3,2,7,1,6,8,5,4]

First, insert everything into the **PriorityQueue**

PriorityQueue: [ ]
Output: []

Sorting w/Lazy PQ (Selection Sort)

Input: [3,2,7,1,6,8,5,4]

First, insert everything into the **PriorityQueue**

PriorityQueue: [3]
Output: []

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

First, insert everything into the **`PriorityQueue`**

`PriorityQueue: [ 3,2 ]`
`Output: []`

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

First, insert everything into the **`PriorityQueue`**

`PriorityQueue: [ 3,2,7 ]`
`Output: []`

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

First, insert everything into the **`PriorityQueue`**

`PriorityQueue: [ 3,2,7,1 ]`
`Output: []`

Sorting w/Lazy PQ (Selection Sort)

Input: [3,2,7,1,6,8,5,4]

First, insert everything into the **PriorityQueue**

PriorityQueue: [3,2,7,1,6]
Output: []

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

First, insert everything into the **`PriorityQueue`**

`PriorityQueue: [ 3,2,7,1,6,8 ]`
`Output: []`

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

First, insert everything into the **`PriorityQueue`**

`PriorityQueue: [ 3,2,7,1,6,8,5 ]`
`Output: []`

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

First, insert everything into the **`PriorityQueue`**

`PriorityQueue: [ 3,2,7,1,6,8,5,4 ]`
`Output: []`

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

First, insert everything into the **`PriorityQueue`**

`PriorityQueue: [ 3,2,7,1,6,8,5,4 ]`
`Output: []`

How long did that take?

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

First, insert everything into the **`PriorityQueue`**

`PriorityQueue: [ 3,2,7,1,6,8,5,4 ]`
`Output: []`

How long did that take? $\Theta(n)$

Each of the n inserts was just lazily appending to the end of a linked list

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ 3,2,7,1,6,8,5,4 ]`
`Output: [ ]`

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ 3,2,7,6,8,5,4 ]`
`Output: [ 1 ]`

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ 3,7,6,8,5,4 ]`
`Output: [ 1,2 ]`

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ 7,6,8,5,4 ]`
`Output: [ 1,2,3 ]`

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ 7,6,8,5 ]`
`Output: [ 1,2,3,4 ]`

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ 7,6,8 ]`
`Output: [ 1,2,3,4,5 ]`

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ 7,8 ]`
`Output: [ 1,2,3,4,5,6 ]`

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ 8 ]`
`Output: [ 1,2,3,4,5,6,7 ]`

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ ]`
`Output: [ 1,2,3,4,5,6,7,8 ]`

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ ]`
`Output: [ 1,2,3,4,5,6,7,8 ]`

How long did that take?

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ ]`
`Output: [ 1,2,3,4,5,6,7,8 ]`

How long did that take?

Each removal required us to search the whole PQ to **select** the smallest element

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ ]`
`Output: [ 1,2,3,4,5,6,7,8 ]`

How long did that take?

Each removal required us to search the whole PQ to **select** the smallest element

# of steps to do that n times: $n + (n-1) + (n-2) + ... + 1 = \sum_{i=1}^{n} i = \Theta(n^2)$

Sorting w/Lazy PQ (Selection Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ ]`
`Output: [ 1,2,3,4,5,6,7,8 ]`

How long did that take?

Each removal required us to search the whole PQ to **select** the smallest element

# of steps to do that n times: $n + (n-1) + (n-2) + ... + 1 = \sum_{i=1}^{n} i = \Theta(n^2)$

Total runtime for the whole sort?

Sorting w/Lazy PQ (Selection Sort)

Input: [3,2,7,1,6,8,5,4]

Second, remove everything from the **PriorityQueue** and append to output

PriorityQueue: [ ]
Output: [1,2,3,4,5,6,7,8]

How long did that take?

Each removal required us to search the whole PQ to **select** the smallest element

# of steps to do that n times: $n + (n-1) + (n-2) + ... + 1 = \sum_{i=1}^{n} i = \Theta(n^2)$

Total runtime for the whole sort? $\Theta(n) + \Theta(n^2) = \Theta(n^2)$

Sorting w/Proactive PQ (Insertion Sort)

Input: [3,2,7,1,6,8,5,4]

First, insert everything into the **PriorityQueue**

PriorityQueue: [ ]
Output: []

Sorting w/Proactive PQ (Insertion Sort)

Input: [3,2,7,1,6,8,5,4]

First, insert everything into the **PriorityQueue**

PriorityQueue: [3]
Output: []

Sorting w/Proactive PQ (Insertion Sort)

Input: [3,2,7,1,6,8,5,4]

First, insert everything into the **PriorityQueue**

PriorityQueue: [2,3]
Output: []

Sorting w/Proactive PQ (Insertion Sort)

Input: [3,2,7,1,6,8,5,4]

First, insert everything into the **PriorityQueue**

PriorityQueue: [2,3,7]
Output: []

Sorting w/Proactive PQ (Insertion Sort)

Input: [3,2,7,1,6,8,5,4]

First, insert everything into the **PriorityQueue**

PriorityQueue: [1,2,3,7]
Output: []

Sorting w/Proactive PQ (Insertion Sort)

Input: [3,2,7,1,6,8,5,4]

First, insert everything into the **PriorityQueue**

PriorityQueue: [1,2,3,6,7]
Output: []

Sorting w/Proactive PQ (Insertion Sort)

Input: [3,2,7,1,6,8,5,4]

First, insert everything into the **PriorityQueue**

PriorityQueue: [1,2,3,6,7,8]
Output: []

Sorting w/Proactive PQ (Insertion Sort)

`Input: [3,2,7,1,6,8,5,4]`

First, insert everything into the **`PriorityQueue`**

`PriorityQueue: [ 1,2,3,5,6,7,8 ]`
`Output: []`

Sorting w/Proactive PQ (Insertion Sort)

`Input: [3,2,7,1,6,8,5,4]`

First, insert everything into the **`PriorityQueue`**

`PriorityQueue: [ 1,2,3,4,5,6,7,8 ]`
`Output: []`

Sorting w/Proactive PQ (Insertion Sort)

`Input: [3,2,7,1,6,8,5,4]`

First, insert everything into the **`PriorityQueue`**

`PriorityQueue: [ 1,2,3,4,5,6,7,8 ]`
`Output: []`

How long did that take?

Sorting w/Proactive PQ (Insertion Sort)

`Input: [3,2,7,1,6,8,5,4]`

First, insert everything into the **`PriorityQueue`**

`PriorityQueue: [ 1,2,3,4,5,6,7,8 ]`
`Output: []`

How long did that take?

Each **insert** required us to potentially search the entire PQ

Sorting w/Proactive PQ (Insertion Sort)

Input: [3,2,7,1,6,8,5,4]

First, insert everything into the **PriorityQueue**

PriorityQueue: [1,2,3,4,5,6,7,8]
Output: []

How long did that take?

Each **insert** required us to potentially search the entire PQ

# of steps to do that n times: $O(1) + O(2) + O(3) + ... + O(n) = \sum_{i=1}^{n} O(i) = O(n^2)$

Sorting w/Proactive PQ (Insertion Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ 1,2,3,4,5,6,7,8 ]`
`Output: [ ]`

Sorting w/Proactive PQ (Insertion Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ 2,3,4,5,6,7,8 ]`
`Output: [ 1 ]`

Sorting w/Proactive PQ (Insertion Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ 3,4,5,6,7,8 ]`
`Output: [ 1,2 ]`

Sorting w/Proactive PQ (Insertion Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ 4,5,6,7,8 ]`
`Output: [ 1,2,3 ]`

Sorting w/Proactive PQ (Insertion Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ 5,6,7,8 ]`
`Output: [ 1,2,3,4 ]`

Sorting w/Proactive PQ (Insertion Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ 6,7,8 ]`
`Output: [ 1,2,3,4,5 ]`

Sorting w/Proactive PQ (Insertion Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ 7,8 ]`
`Output: [ 1,2,3,4,5,6 ]`

Sorting w/Proactive PQ (Insertion Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ 8 ]`
`Output: [ 1,2,3,4,5,6,7 ]`

Sorting w/Proactive PQ (Insertion Sort)

Input: [3,2,7,1,6,8,5,4]

Second, remove everything from the **PriorityQueue** and append to output

PriorityQueue: [ ]
Output: [1,2,3,4,5,6,7,8]

Sorting w/Proactive PQ (Insertion Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ ]`
`Output: [ 1,2,3,4,5,6,7,8 ]`

How long did that take?

Sorting w/Proactive PQ (Insertion Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ ]`
`Output: [ 1,2,3,4,5,6,7,8 ]`

How long did that take? $\Theta(n)$

Each of the n removes was removing the head of the linked list

Sorting w/Proactive PQ (Insertion Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ ]`
`Output: [ 1,2,3,4,5,6,7,8 ]`

How long did that take? $\Theta(n)$

Each of the n removes was removing the head of the linked list

Total runtime for the whole sort?

Sorting w/Proactive PQ (Insertion Sort)

`Input: [3,2,7,1,6,8,5,4]`

Second, remove everything from the **`PriorityQueue`** and append to output

`PriorityQueue: [ ]`
`Output: [ 1,2,3,4,5,6,7,8 ]`

How long did that take? $\Theta(n)$

Each of the n removes was removing the head of the linked list

Total runtime for the whole sort? $O(n^2) + \Theta(n) = O(n^2)$

Improving Our Implementation

Keeping our **`PriorityQueue`** fully sorted makes adding expensive

Letting our **`PriorityQueue`** stay fully unsorted makes removal expensive

Improving Our Implementation

Keeping our **`PriorityQueue`** fully sorted makes adding expensive

Letting our **`PriorityQueue`** stay fully unsorted makes removal expensive

Idea: What if we keep the PriorityQueue “kinda” sorted

- Keep higher priority towards the front
- Keep the front more sorted than the back

The hope is that adding will be faster than when it was fully sorted and removal will be faster than when it was fully unsorted

Improving Our Implementation

Keeping our **`PriorityQueue`** fully sorted makes adding expensive

Letting our **`PriorityQueue`** stay fully unsorted makes removal expensive

Idea: What if we keep the PriorityQueue “kinda” sorted

- Keep higher priority towards the front
- Keep the front more sorted than the back

The hope is that adding will be faster than when it was fully sorted and removal will be faster than when it was fully unsorted

Challenge: If it’s only partially sorted, how do we know how elements are ordered (with respect to one another)?

Improving Our Implementation

Keeping our **`PriorityQueue`** fully sorted makes adding expensive

Letting our **`PriorityQueue`** stay fully unsorted makes removal expensive

Idea: What if we keep the PriorityQueue “kinda” sorted

- Keep higher priority towards the front
- Keep the front more sorted than the back

The hope is that adding will be faster than when it was fully sorted and removal will be faster than when it was fully unsorted

Challenge: If it’s only partially sorted, how do we know how elements are ordered (with respect to one another)?

Idea: Organize our priority queue as a rooted tree

Heaps

Tree Terminology

Child (of X)

A node adjacent to X, connected by an out-edge

- **B**, **C** are children of **A**
- **D** is a child of **C**
- **E** and **F** are children of **D**

Root

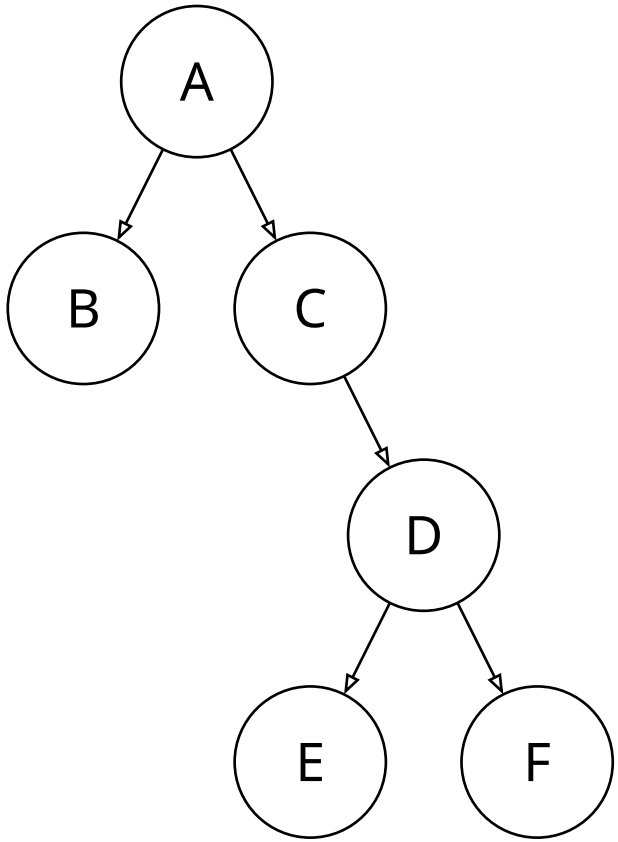
A node that is not the child of any node

- **A** is the root of the tree

Leaf

A node without any children

- **B**, **E**, and **F** are leaves



Tree Terminology

Depth (of node X)

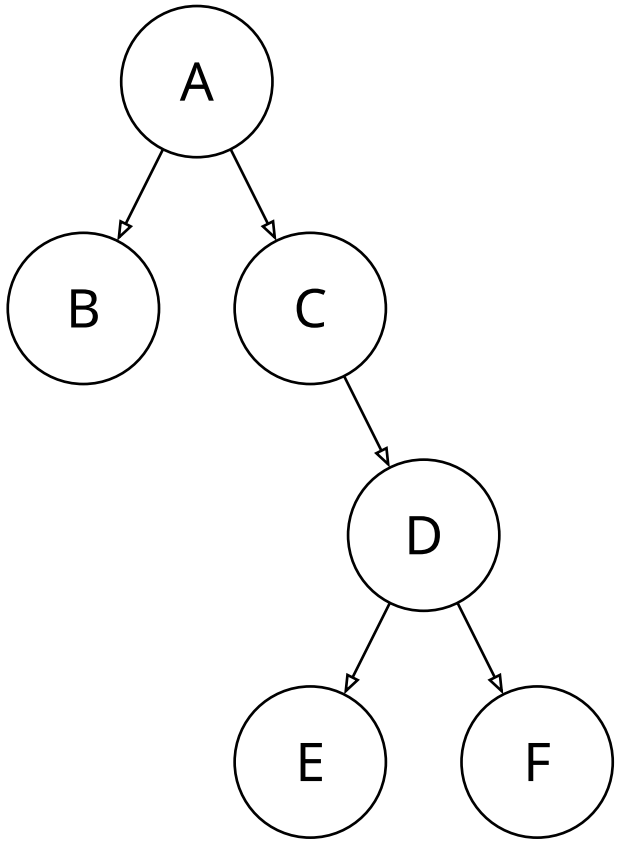
The number of edges from the root to X

- **A** is at a depth of 0
- **B**, **C** are at a depth of 1
- **D** is at a depth of 2
- **E**, **F** are at a depth of 3

Depth (of a tree)

The maximum depth of any node in the tree

- The tree has a depth of 3



Binary Heaps

A **Binary Heap** is a *complete*, *rooted*, *binary* tree that enforces an ordering between parent/child nodes

Binary Heaps

A **Binary Heap** is a *complete*, *rooted*, *binary* tree that enforces an ordering between parent/child nodes

- A **rooted** tree is a directed tree with one node designated as the root

Binary Heaps

A **Binary Heap** is a *complete*, *rooted*, *binary* tree that enforces an ordering between parent/child nodes

- A **rooted** tree is a directed tree with one node designated as the root
- A **binary** tree is a tree where every node has at most 2 children

Binary Heaps

A **Binary Heap** is a *complete*, *rooted*, *binary* tree that enforces an ordering between parent/child nodes

- A **rooted** tree is a directed tree with one node designated as the root
- A **binary** tree is a tree where every node has at most 2 children
- A **complete** binary tree is a tree where every level but the last must be full (and the last is filled from left to right)

Binary Heaps

A **Binary Heap** is a *complete*, *rooted*, *binary* tree that enforces an ordering between parent/child nodes

- A **rooted** tree is a directed tree with one node designated as the root
- A **binary** tree is a tree where every node has at most 2 children
- A **complete** binary tree is a tree where every level but the last must be full (and the last is filled from left to right)

So what is the ordering constraint imposed?

Binary Heaps

A **Binary Heap** is a *complete*, *rooted*, *binary* tree that enforces an ordering between parent/child nodes

- A **rooted** tree is a directed tree with one node designated as the root
- A **binary** tree is a tree where every node has at most 2 children
- A **complete** binary tree is a tree where every level but the last must be full (and the last is filled from left to right)

So what is the ordering constraint imposed?

Binary **Min** Heap: Every parent must be less than or equal to its children

- An edge from a to b means $a \leq b$

Binary Heaps

A **Binary Heap** is a *complete*, *rooted*, *binary* tree that enforces an ordering between parent/child nodes

- A **rooted** tree is a directed tree with one node designated as the root
- A **binary** tree is a tree where every node has at most 2 children
- A **complete** binary tree is a tree where every level but the last must be full (and the last is filled from left to right)

So what is the ordering constraint imposed?

Binary **Min** Heap: Every parent must be less than or equal to its children

- An edge from a to b means $a \leq b$

Binary **Max** Heap: Every parent must be greater than or equal to its children

- An edge from a to b means $a \geq b$

Binary Heaps

A **Binary Heap** is a *complete*, *rooted*, *binary* tree that enforces an ordering between parent/child nodes

- A **rooted** tree is a directed tree with one node designated as the root
- A **binary** tree is a tree where every node has at most 2 children
- A **complete** binary tree is a tree where every level but the last must be full (and the last is filled from left to right)

So what is the ordering constraint imposed?

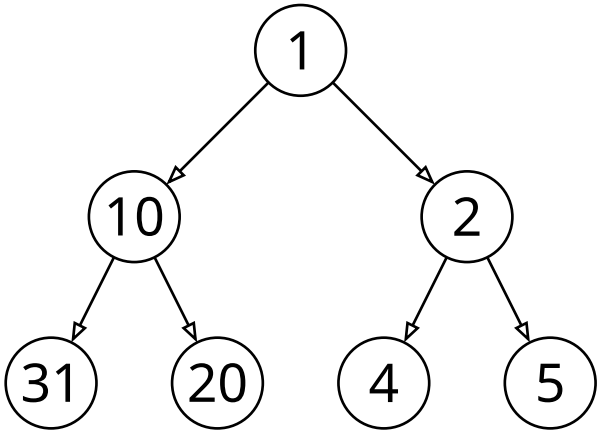
Binary **Min** Heap: Every parent must be less than or equal to its children

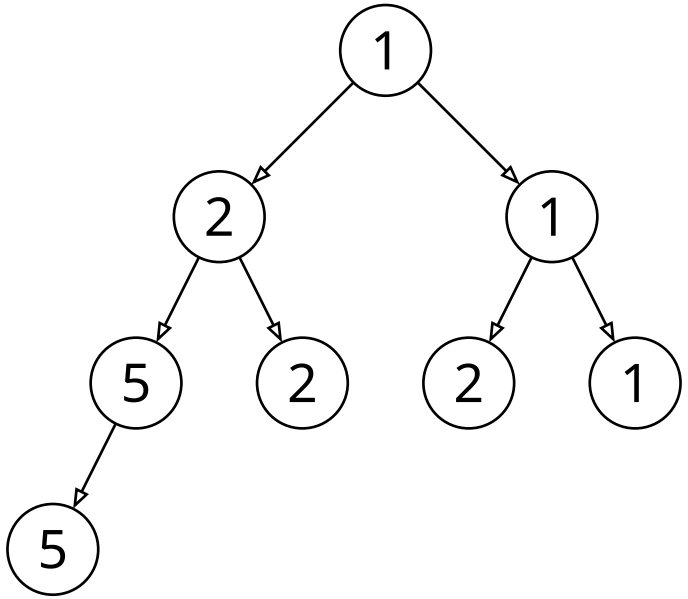
- An edge from a to b means $a \leq b$

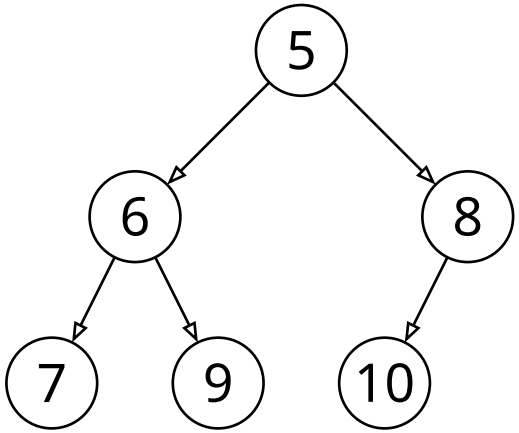
Binary **Max** Heap: Every parent must be greater than or equal to its children

- An edge from a to b means $a \geq b$

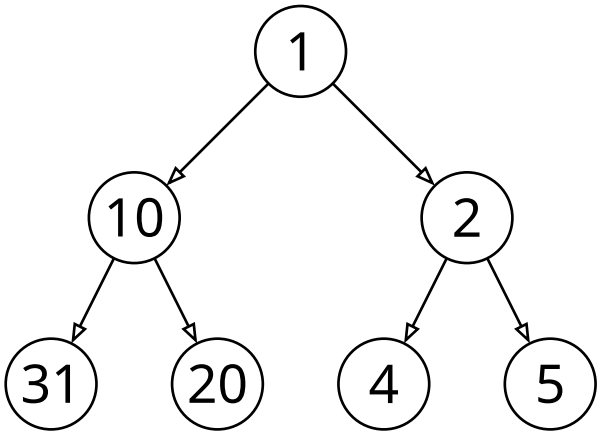
Valid Min Heaps

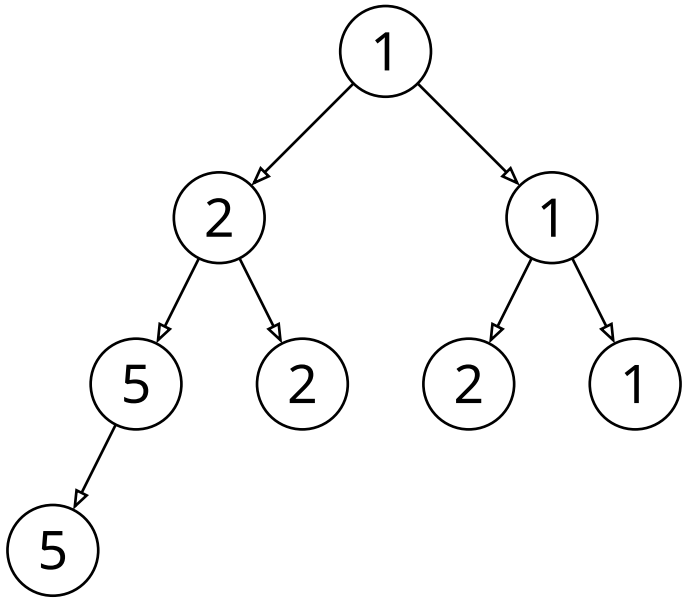


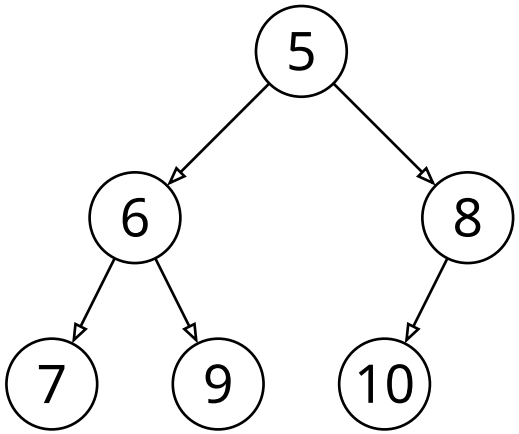




Valid Min Heaps





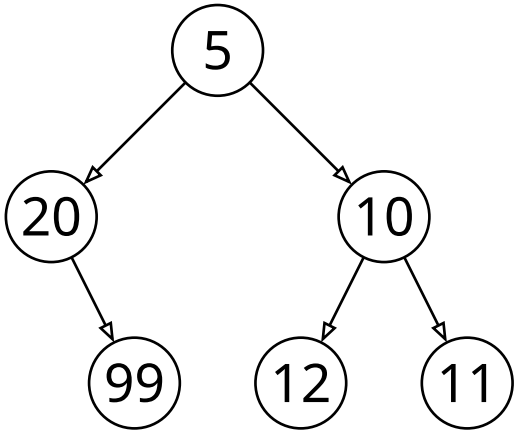


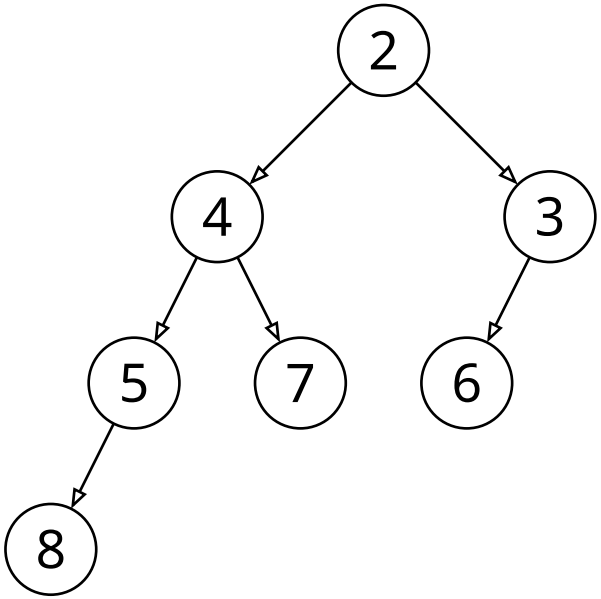
✓ **Complete**
✓ **Correctly Ordered**

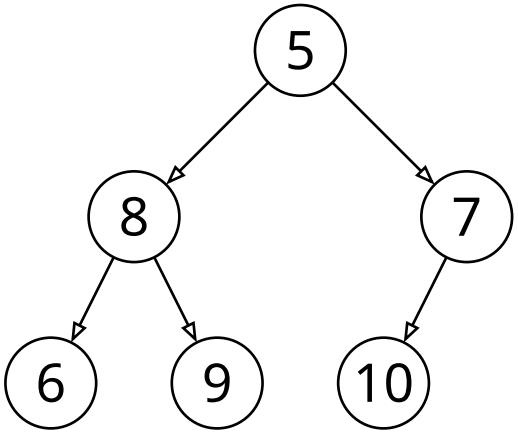
✓ **Complete**
✓ **Correctly Ordered**

✓ **Complete**
✓ **Correctly Ordered**

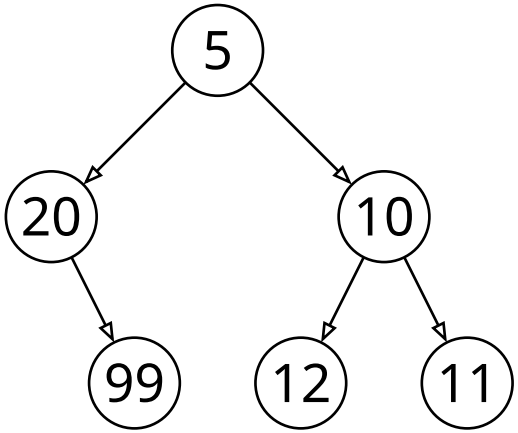
Invalid Min Heaps

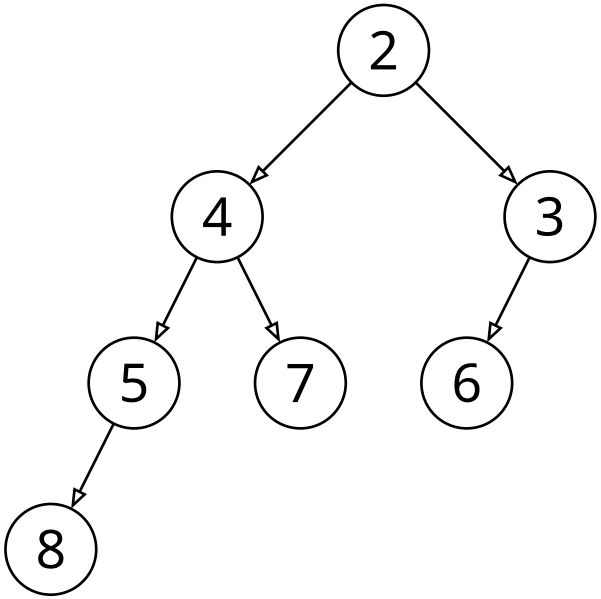


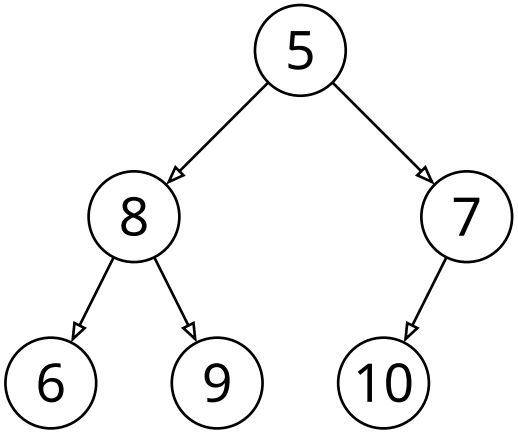




Invalid Min Heaps



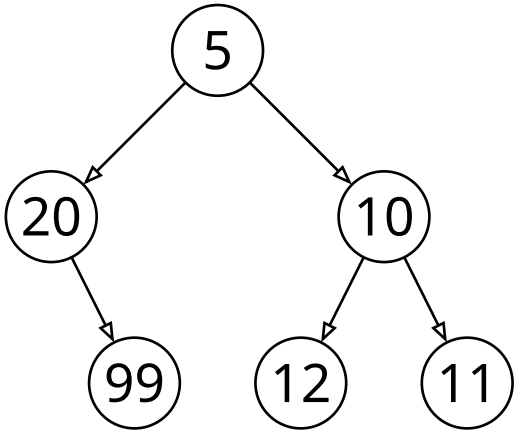




Not Complete!

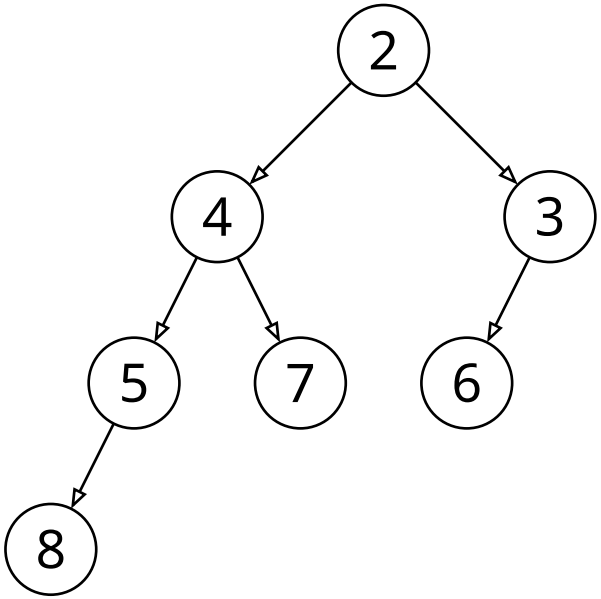
Last row must be filled from left to right

Invalid Min Heaps



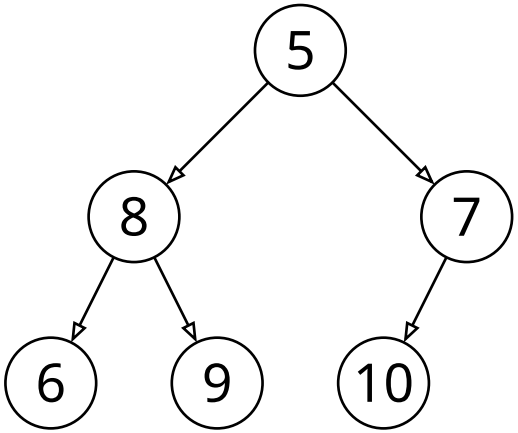
Not Complete!

Last row must be filled from left to right

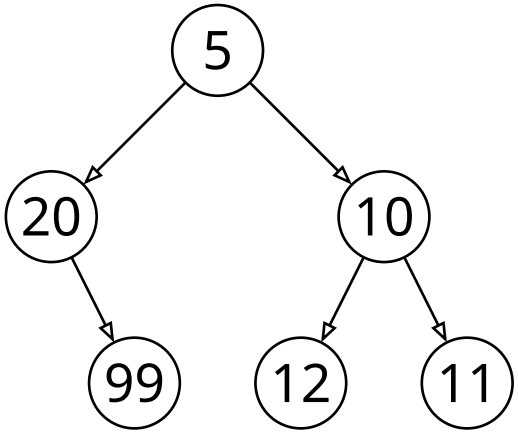


Not Complete!

Rows that aren't the last row must be full

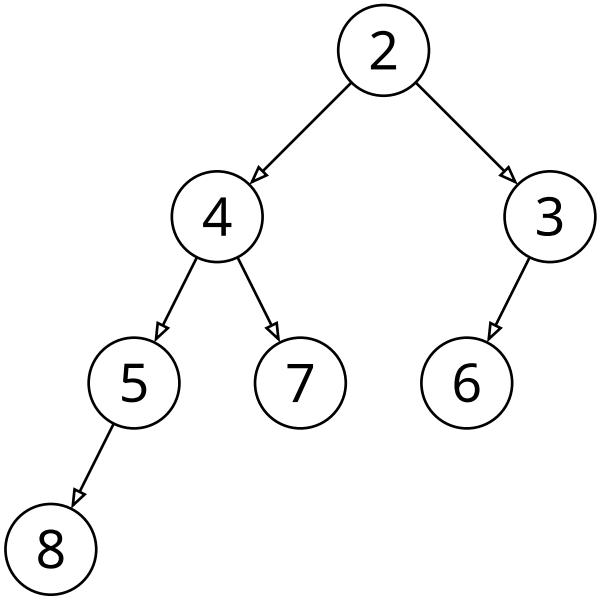


Invalid Min Heaps



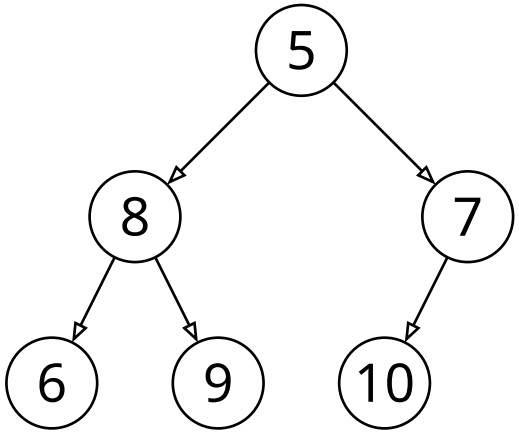
Not Complete!

Last row must be filled from left to right



Not Complete!

Rows that aren't the last row must be full



Incorrect Order!

Parents must be less than or equal to children ($8 \not\leq 6$)

(Min) Heap Operations

`void pushHeap(T value)`

Place **`value`** into the heap

`T popHeap()`

Remove and return the minimal element from the heap

`T peek()`

Return the minimal element from the heap without removal

(Min) Heap Operations

`void pushHeap(T value)`

Place **`value`** into the heap

`T popHeap()`

Remove and return the minimal element from the heap

`T peek()`

Return the minimal element from the heap without removal

Approach for `pushHeap/popHeap`:

Do whatever is easiest to preserve structure, then fix any broken ordering

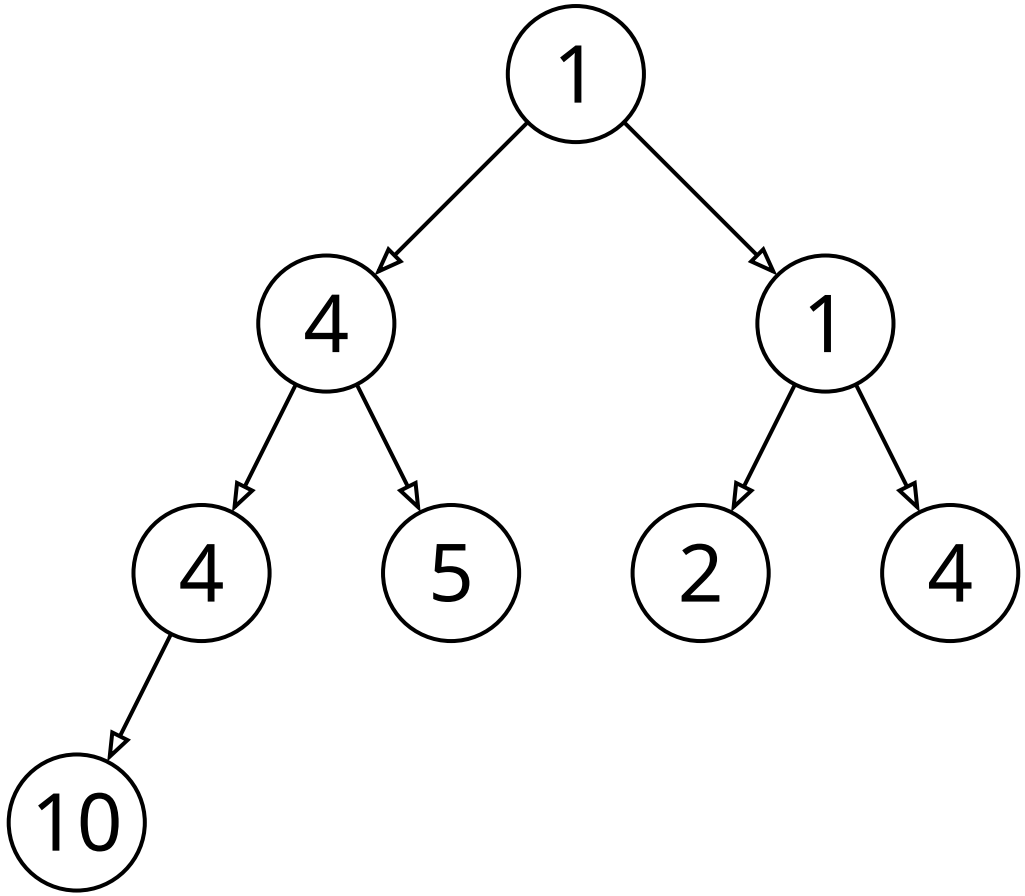
Algorithm: pushHeap

pushHeap(T value)

```
1 Add value to next available position // keeps the tree complete
2
3 // Fix any ordering issues by moving the value UP in the heap
4 while newNode.value < newNode.parent.value:
5     swap newNode with newNode.parent
```

Example: `pushHeap`

Imagine we want to **`pushHeap(3)`**

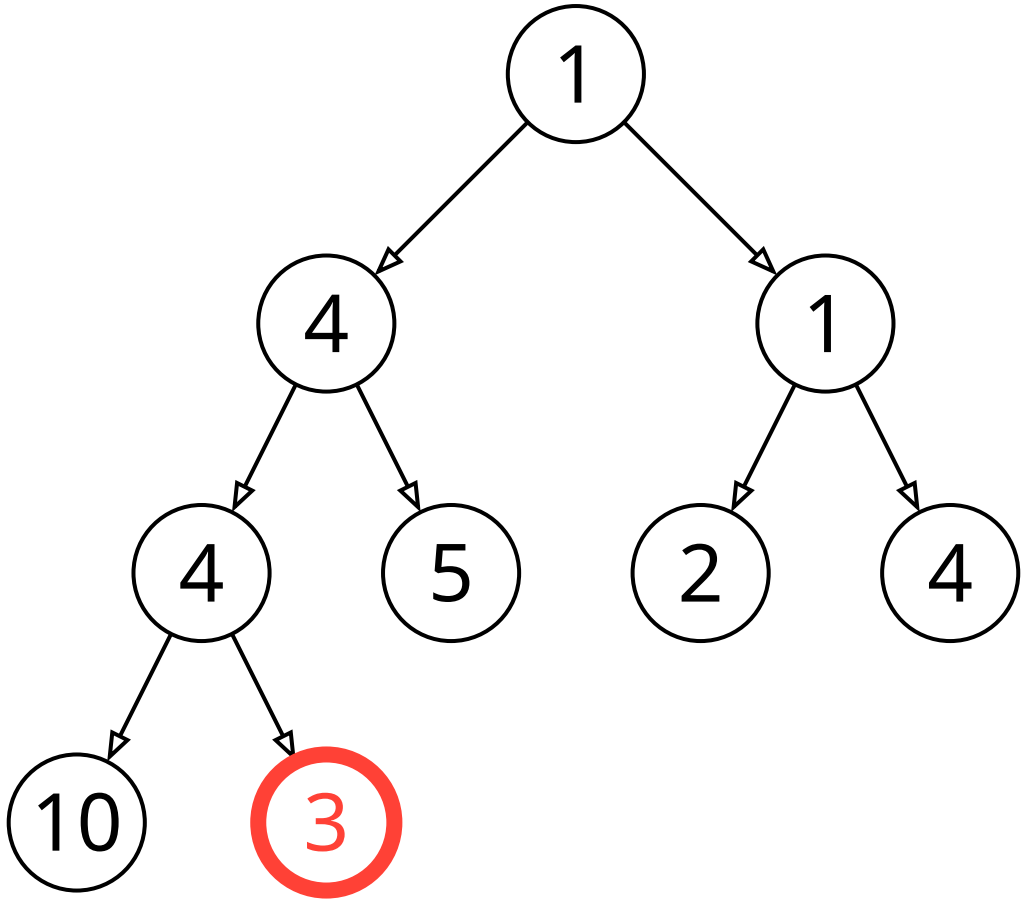


Example: pushHeap

Imagine we want to **`pushHeap(3)`**

First add 3 to the next available spot

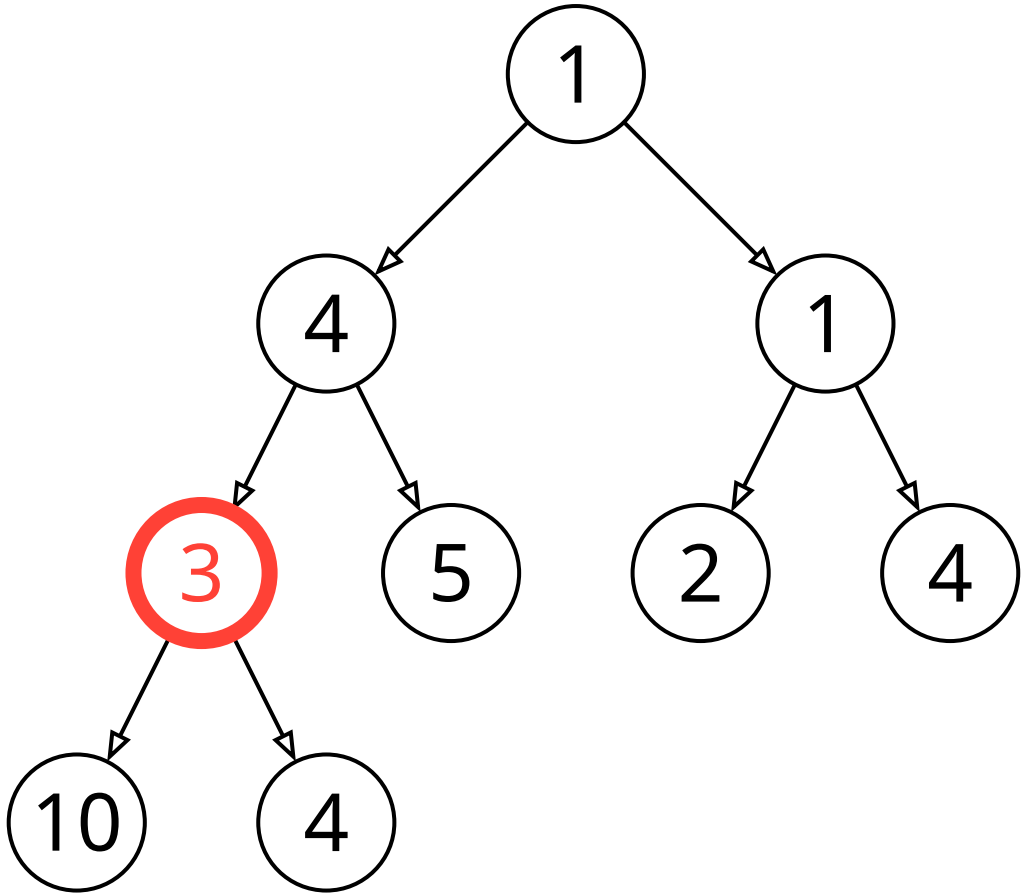
We've potentially broken the ordering of our heap so we need to check the parent of 3 and swap if 3 is smaller



Example: pushHeap

Imagine we want to **`pushHeap(3)`**

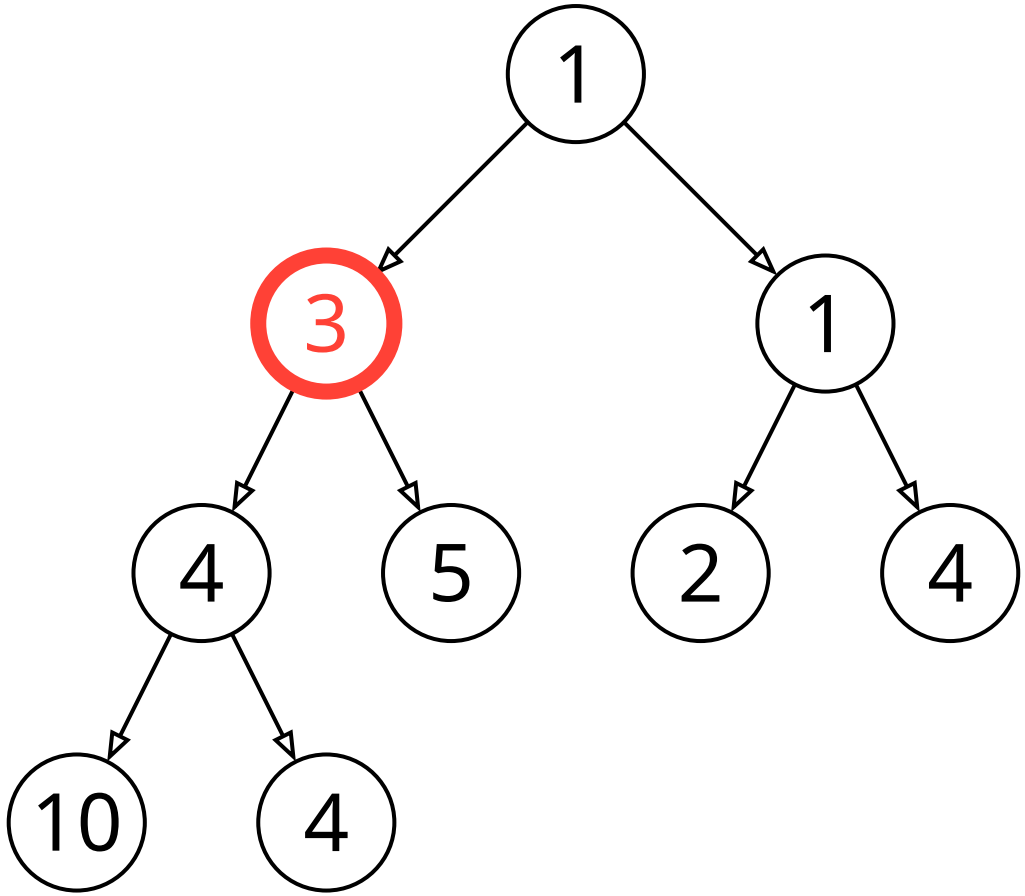
After one swap, we still may have ordering problems so repeat the process of checking the parent of 3



Example: `pushHeap`

Imagine we want to **`pushHeap(3)`**

This process is called **`fixUp`** – we move the out of place node UP the heap until we find the right spot

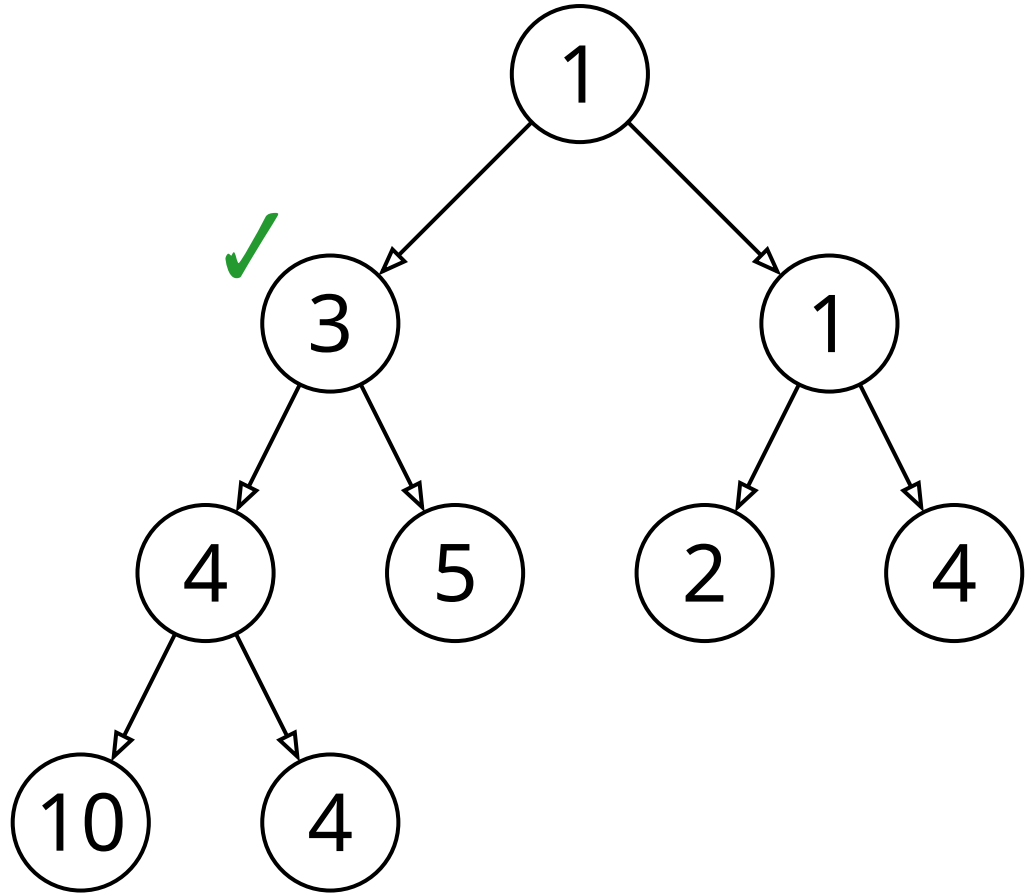


Example: pushHeap

Imagine we want to **`pushHeap(3)`**

At this point 3 is no longer smaller than it's parent so we've fixed the heap!

The **`pushHeap`** operation is now complete!



Implementation: popHeap

Problem: The value we want to remove is the root, but that would break the tree

Implementation: popHeap

Problem: The value we want to remove is the root, but that would break the tree

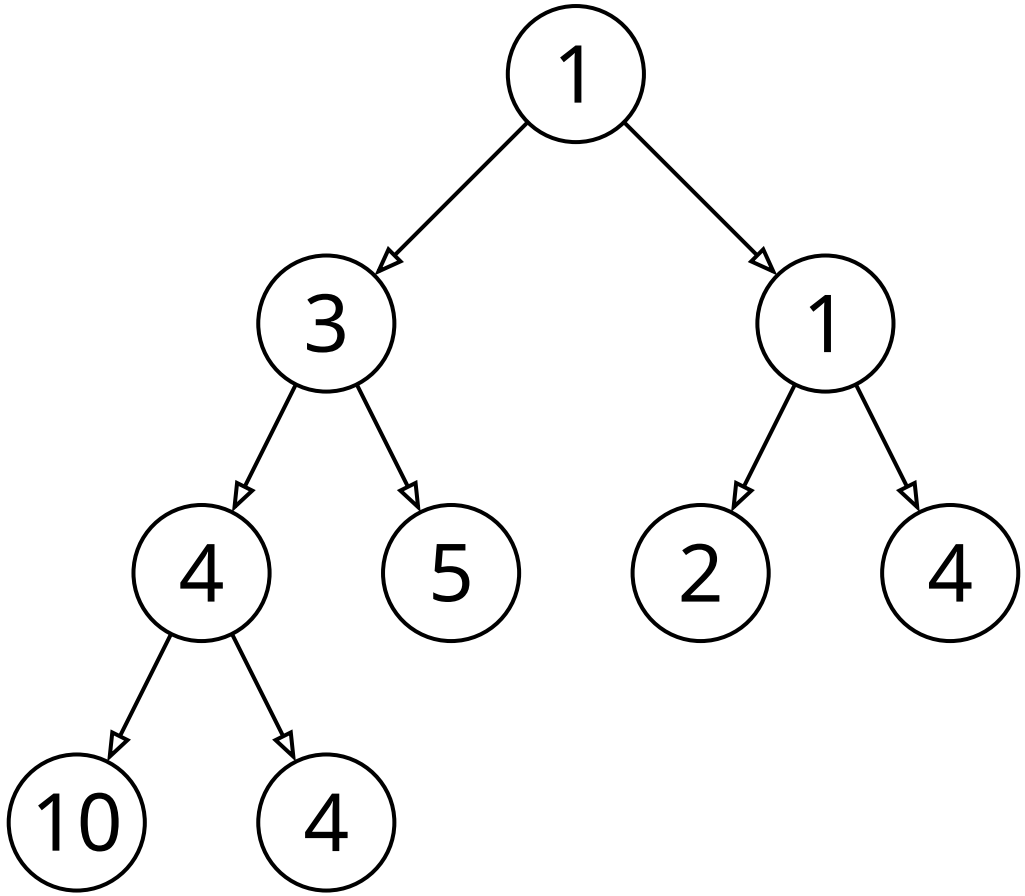
Idea: Swap the root with the last node, then remove it and tree stays complete

popHeap()

```
1 swap root with last node
2 remove last node
3
4 // Fix any ordering issues by moving the new root DOWN the heap
5 currNode = root
6 while currNode.value > minChild(currNode).value:
7     swap currNode with minChild(currNode)
```

Example: popHeap

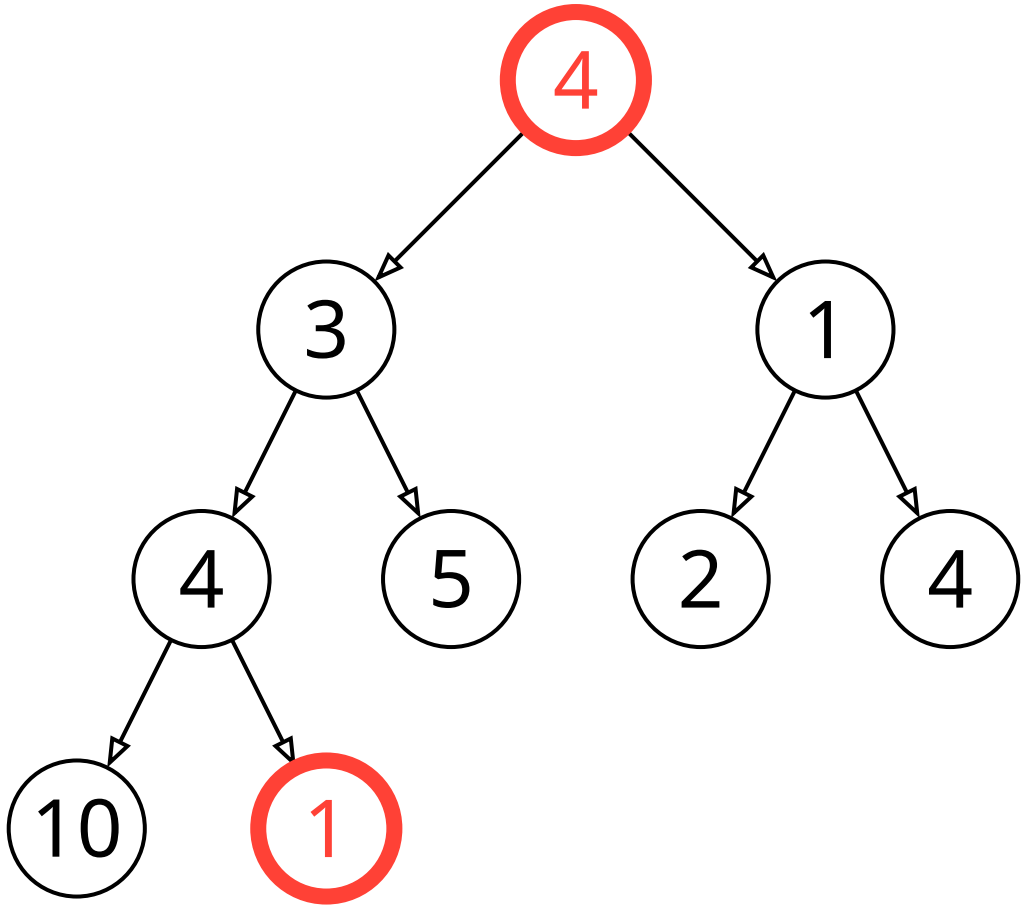
Now we want to pop the smallest value from our heap, which we know is at the root



Example: popHeap

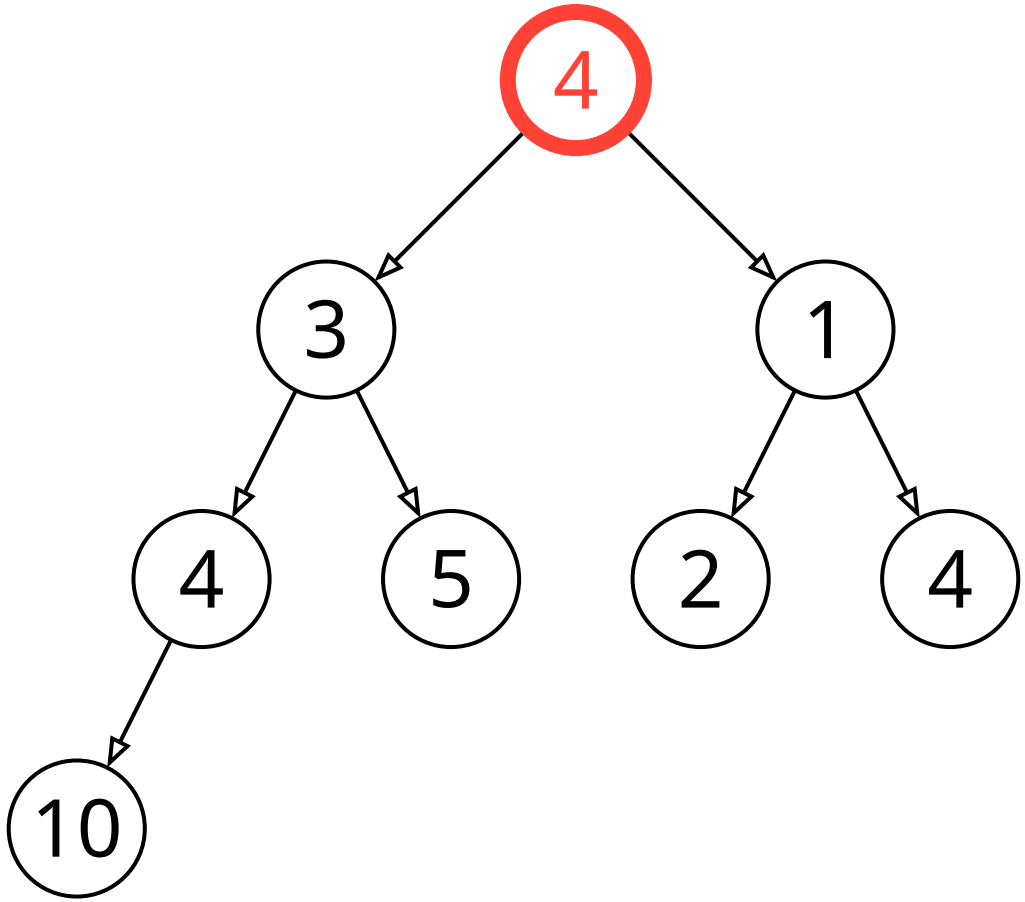
Start by swapping the root with the last node

We can now remove the last node without breaking the completeness property



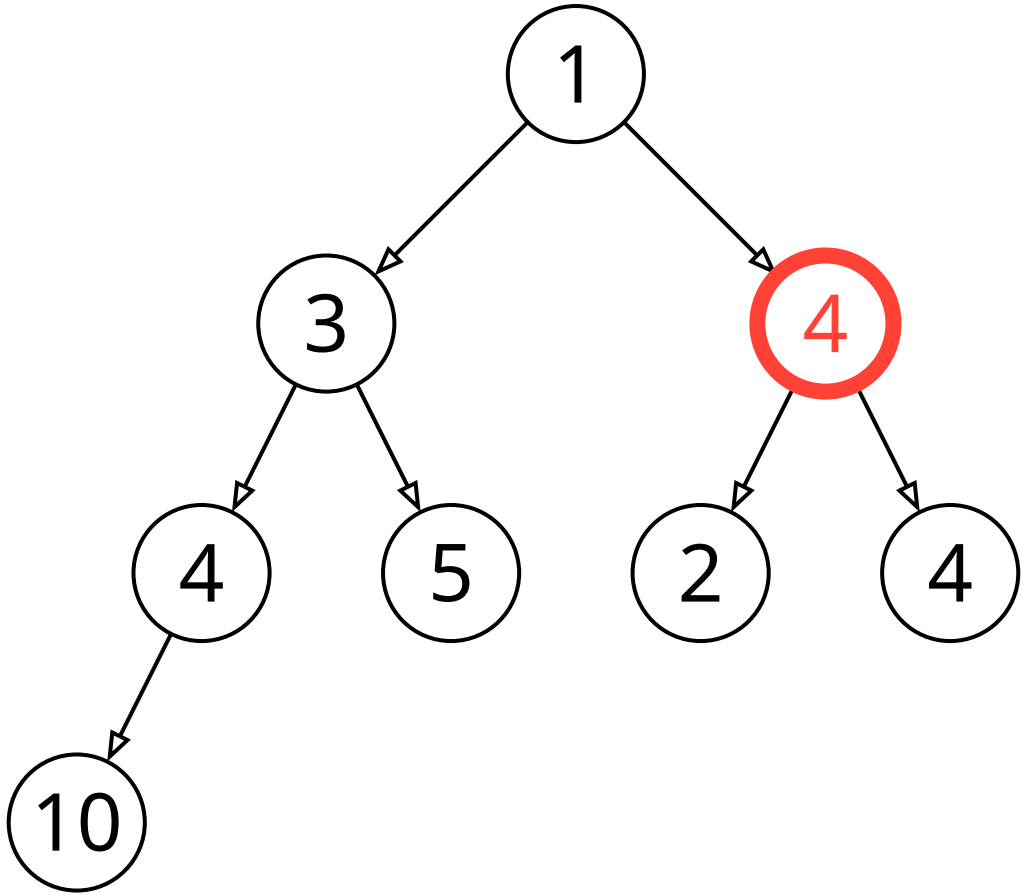
Example: popHeap

Now the root is potentially out of order – if it is larger than at least one of its children we must swap it with its smallest child!



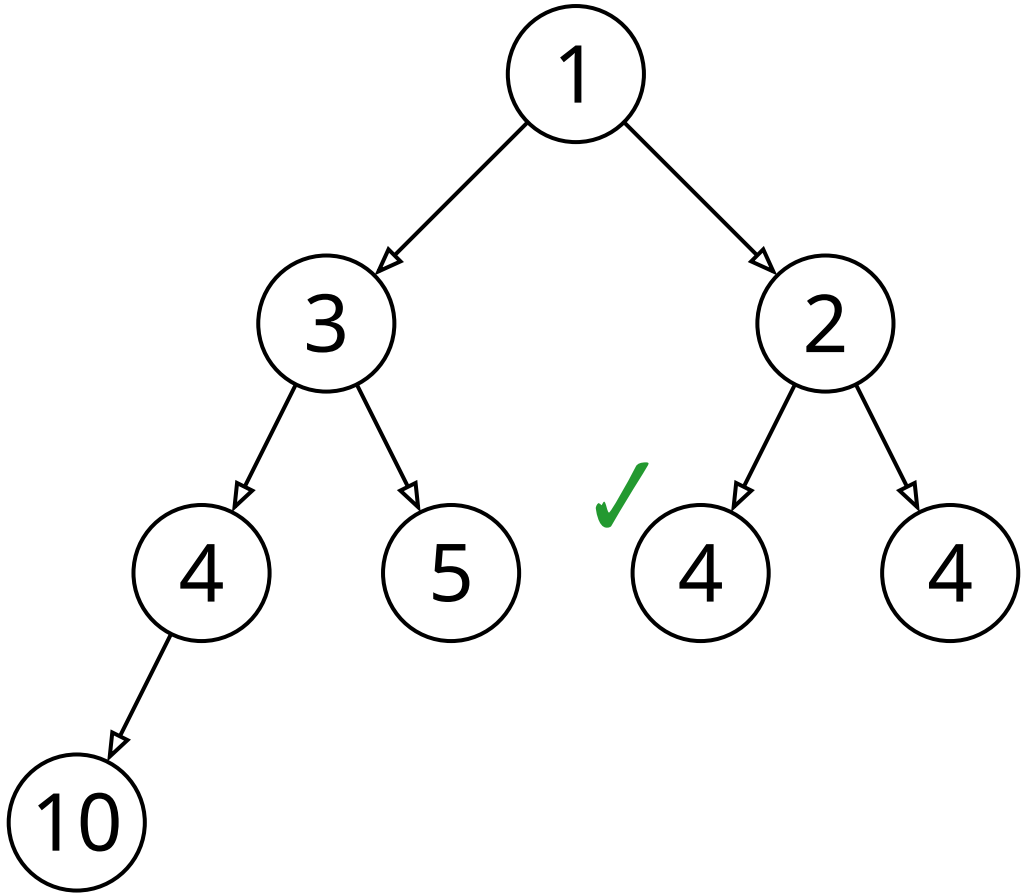
Example: popHeap

This is called **`fixDown`** and just like with **`fixUp`** we have to continue down the tree until we fix all ordering problems.



Example: popHeap

At this point the 4 is correctly ordered and our **popHeap** operation is complete!



Complexity

How many operations were required for **`pushHeap`**?

Complexity

How many operations were required for **`pushHeap`**?

- 1 to insert the new value
- At most one swap per level in the heap to **`fixUp`**

How many operations were required for **`popHeap`**?

Complexity

How many operations were required for **`pushHeap`**?

- 1 to insert the new value
- At most one swap per level in the heap to **`fixUp`**

How many operations were required for **`popHeap`**?

- 1 to swap and remove the root
- At most one swap per level in the heap to **`fixDown`**

Complexity

How many operations were required for **`pushHeap`**?

- 1 to insert the new value
- At most one swap per level in the heap to **`fixUp`**

How many operations were required for **`popHeap`**?

- 1 to swap and remove the root
- At most one swap per level in the heap to **`fixDown`**

In both cases, the number of steps is $O(d)$, where d is the depth of our heap!

So what is the depth of a binary heap?

Depth of a Heap

How many levels are in a binary heap containing n items?

Depth of a Heap

How many levels are in a binary heap containing n items?

Depth 0: holds up to 1 item

Depth of a Heap

How many levels are in a binary heap containing n items?

Depth 0: holds up to 1 item

Depth 1: holds up to 2 items

Depth of a Heap

How many levels are in a binary heap containing n items?

Depth 0: holds up to 1 item

Depth 1: holds up to 2 items

Depth 2: holds up to 4 items

Depth of a Heap

How many levels are in a binary heap containing n items?

Depth 0: holds up to 1 item

Depth 1: holds up to 2 items

Depth 2: holds up to 4 items

Depth 3: holds up to 8 items

Depth of a Heap

How many levels are in a binary heap containing n items?

Depth 0: holds up to 1 item

Depth 1: holds up to 2 items

Depth 2: holds up to 4 items

Depth 3: holds up to 8 items

...

Depth i: holds up to 2^i items

Depth of a Heap

How many levels are in a binary heap containing n items?

Depth 0: holds up to 1 item

Depth 1: holds up to 2 items

Depth 2: holds up to 4 items

Depth 3: holds up to 8 items

...

Depth i: holds up to 2^i items

Remember: Since a binary heap is **complete** only the last level may be holding less than its max capacity

Depth of a Heap

Assume d is depth of our heap.

- The first $d-1$ levels are full
- The last level contains between 1 and 2^d items.

$$\sum_{i=0}^{d-1} 2^i + 1 \leq n \leq \sum_{i=0}^{d-1} 2^i + 2^d$$

Depth of a Heap

Assume d is depth of our heap.

- The first $d-1$ levels are full
- The last level contains between 1 and 2^d items.

$$\sum_{i=0}^{d-1} 2^i + 1 \leq n \leq \sum_{i=0}^{d-1} 2^i + 2^d$$

$$2^d \leq n \leq \left(2^d - 1\right) + 2^d$$

Depth of a Heap

Assume d is depth of our heap.

- The first $d-1$ levels are full
- The last level contains between 1 and 2^d items.

$$\sum_{i=0}^{d-1} 2^i + 1 \leq n \leq \sum_{i=0}^{d-1} 2^i + 2^d$$

$$2^d \leq n \leq \left(2^d - 1\right) + 2^d$$

$$2^d \leq n \leq 2^{d+1} - 1$$

Depth of a Heap

Assume d is depth of our heap.

- The first $d-1$ levels are full
- The last level contains between 1 and 2^d items.

$$\sum_{i=0}^{d-1} 2^i + 1 \leq n \leq \sum_{i=0}^{d-1} 2^i + 2^d$$

$$2^d \leq n \leq \left(2^d - 1\right) + 2^d$$

$$2^d \leq n \leq 2^{d+1} - 1$$

Therefore $n \in \Theta\left(2^d\right)$, and taking $\log$ of both sides gives: $d \in \Theta(\log(n))$

Depth of a Heap

Assume d is depth of our heap.

- The first $d-1$ levels are full
- The last level contains between 1 and 2^d items.

$$\sum_{i=0}^{d-1} 2^i + 1 \leq n \leq \sum_{i=0}^{d-1} 2^i + 2^d$$

$$2^d \leq n \leq \left(2^d - 1\right) + 2^d$$

$$2^d \leq n \leq 2^{d+1} - 1$$

Therefore $n \in \Theta\left(2^d\right)$, and taking $\log$ of both sides gives: $d \in \Theta(\log(n))$

Therefore a heap has $\log(n)$ levels, and **`fixUp`/`fixDown`** require $O(\log(n))$ swaps!

Storing the Heap Data

But how do we actually store the values in our heap?

We could store them in an actual tree (like you saw in 116), where each node has a left and right child.

Storing the Heap Data

But how do we actually store the values in our heap?

We could store them in an actual tree (like you saw in 116), where each node has a left and right child.

But since we know our tree is complete, there's a simpler way...

Storing the Heap Data

But how do we actually store the values in our heap?

We could store them in an actual tree (like you saw in 116), where each node has a left and right child.

But since we know our tree is complete, there's a simpler way...

- Every levels but the last is full

Storing the Heap Data

But how do we actually store the values in our heap?

We could store them in an actual tree (like you saw in 116), where each node has a left and right child.

But since we know our tree is complete, there's a simpler way...

- Every levels but the last is full
- Levels are filled from left to right

Storing the Heap Data

But how do we actually store the values in our heap?

We could store them in an actual tree (like you saw in 116), where each node has a left and right child.

But since we know our tree is complete, there's a simpler way...

- Every levels but the last is full
- Levels are filled from left to right
- Any adding or removing happens only on the right of the last level

Storing the Heap Data

But how do we actually store the values in our heap?

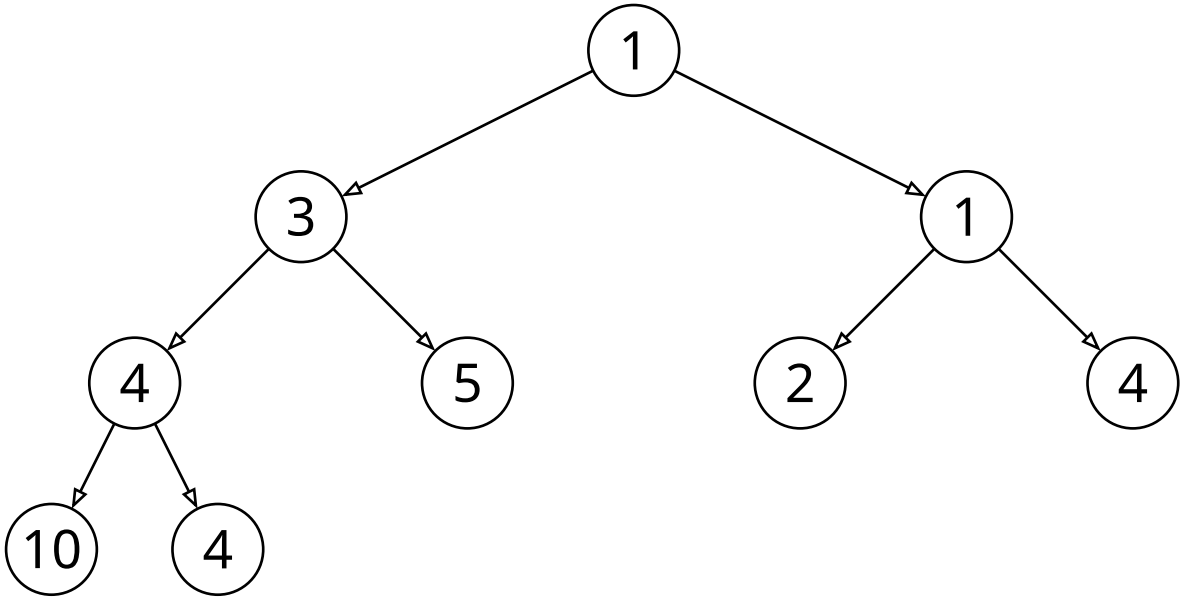
We could store them in an actual tree (like you saw in 116), where each node has a left and right child.

But since we know our tree is complete, there's a simpler way...

- Every levels but the last is full
- Levels are filled from left to right
- Any adding or removing happens only on the right of the last level

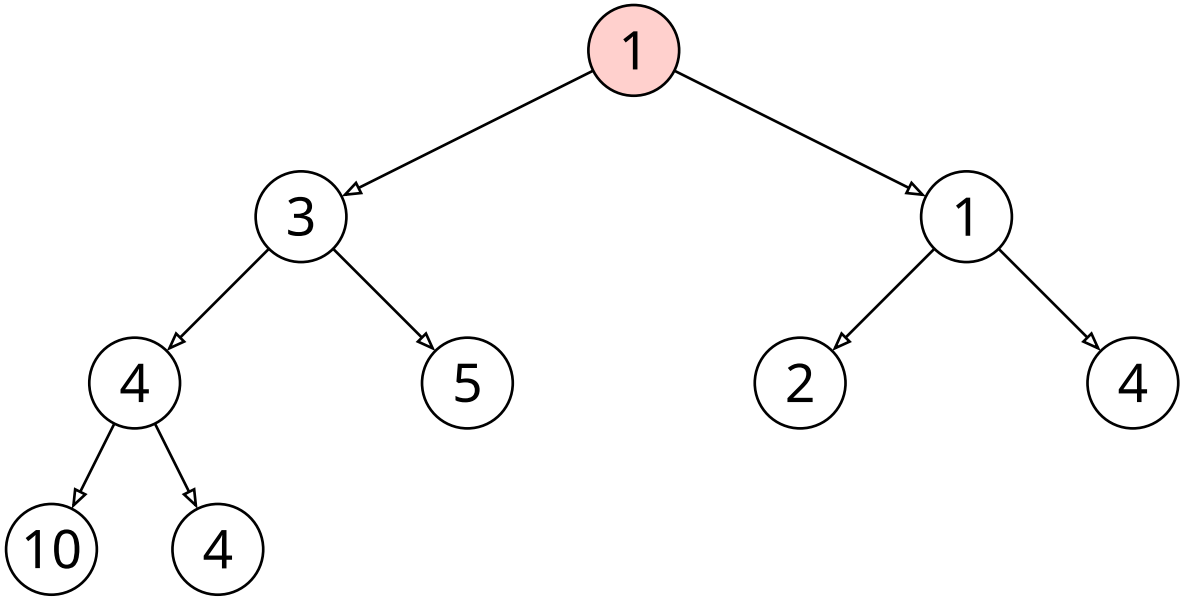
Idea: Let's store the elements of our heap in an **`ArrayList`**

Heaps as an `ArrayList`



Heaps as an `ArrayList`

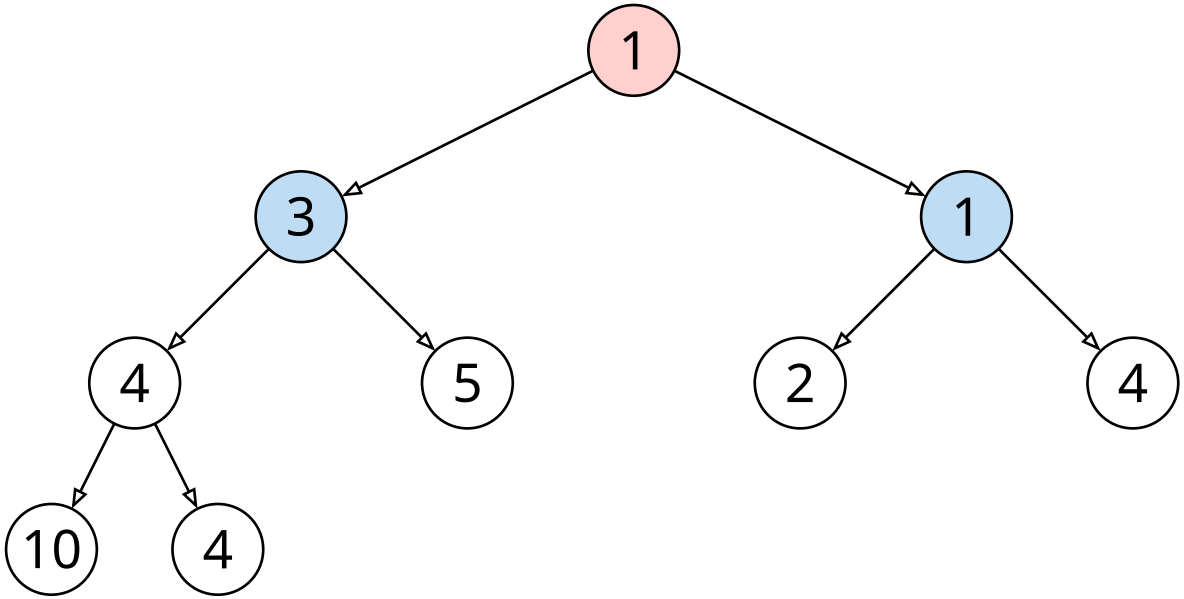
The root is the first value



Heaps as an `ArrayList`

The root is the first value

Store each level left-to-right...

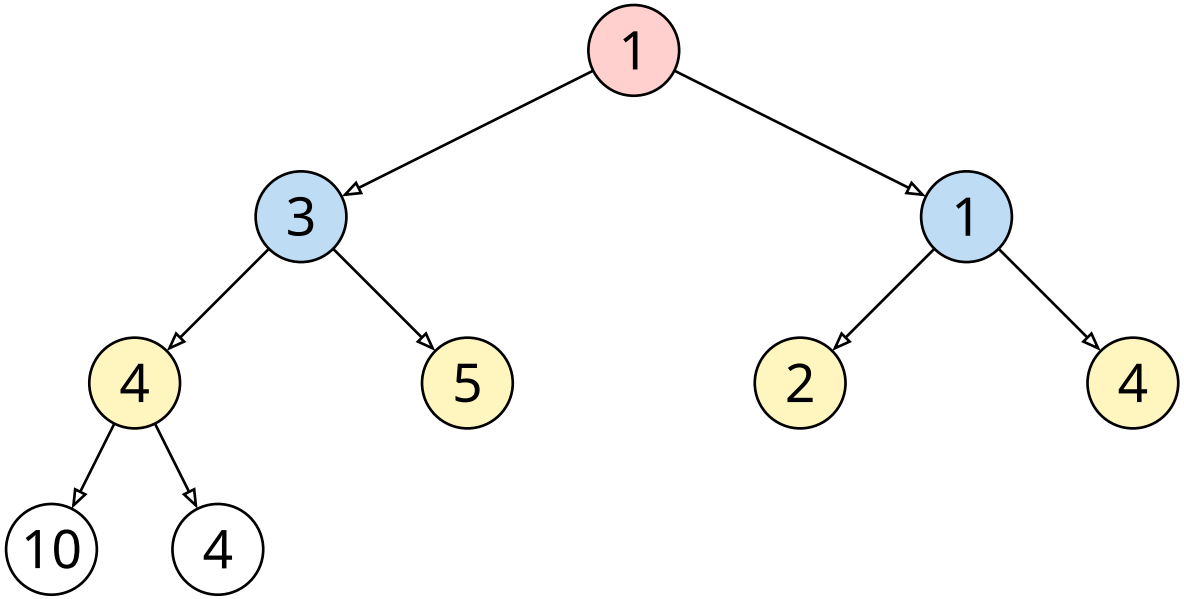


Heaps as an `ArrayList`

The root is the first value

Store each level left-to-right...

...with one level immediately following the previous

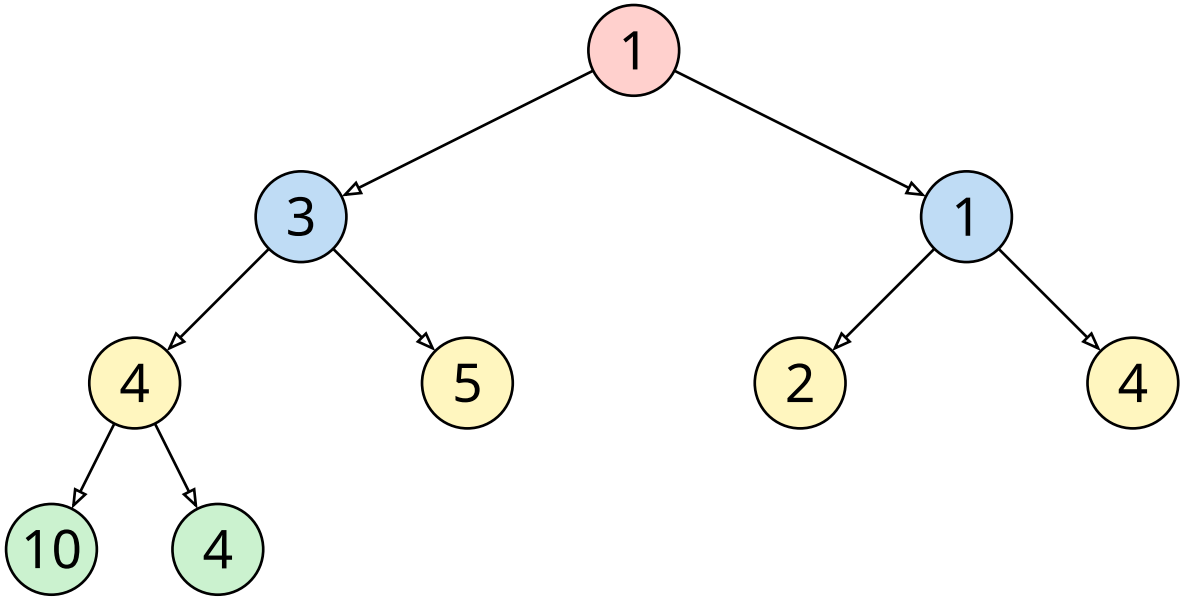


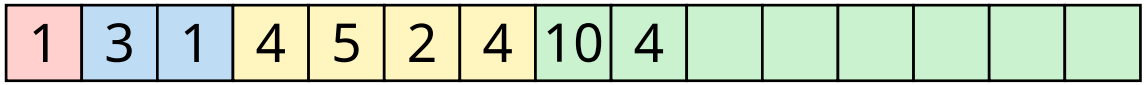
Heaps as an `ArrayList`

The root is the first value

Store each level left-to-right...

...with one level immediately following the previous





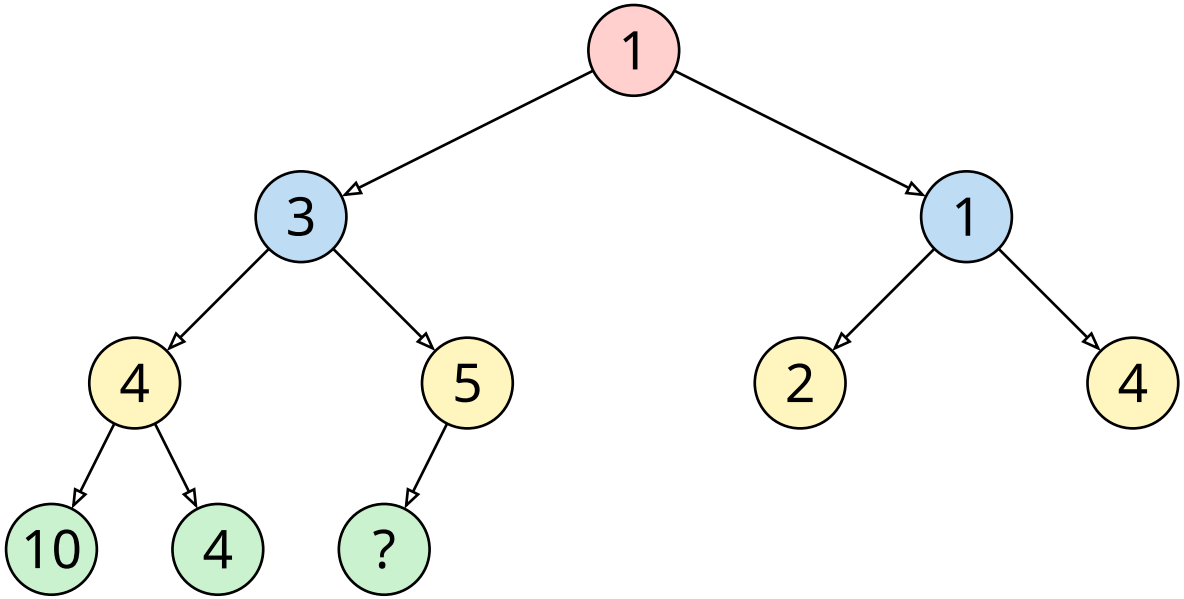
Heaps as an `ArrayList`

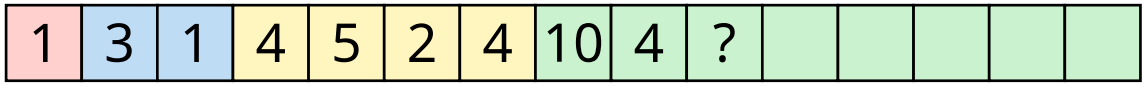
The root is the first value

Store each level left-to-right…

…with one level immediately following the previous

`pushHeap` just needs to append a value to the array (and then **`fixUp`**)

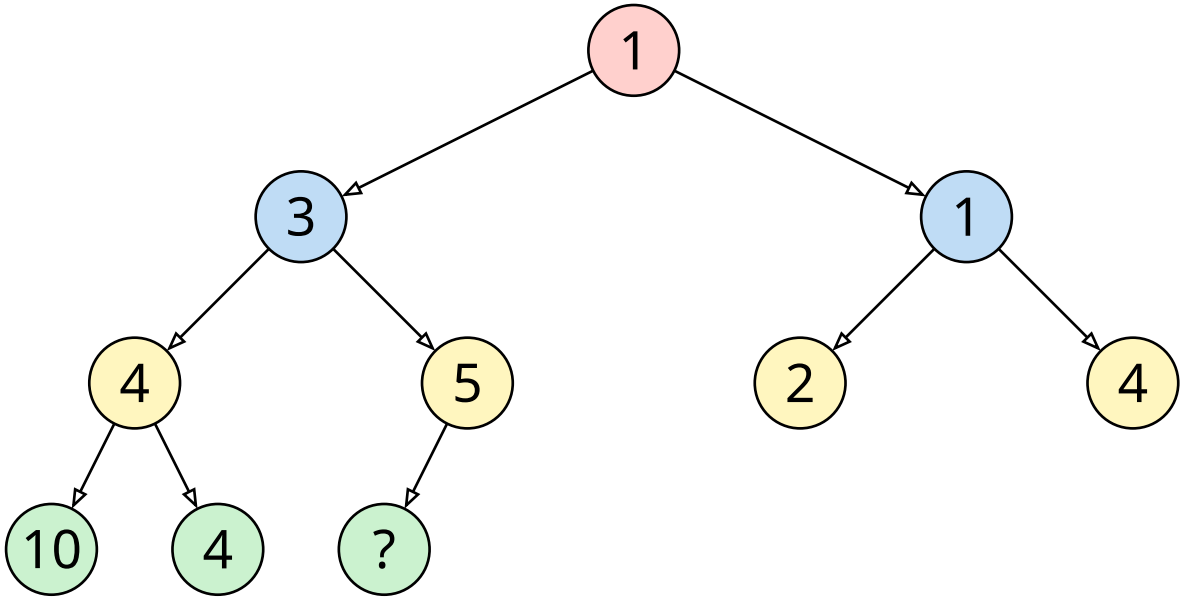




Heaps as an `ArrayList`

The parent of a node at index **`i`** is at index

`(i - 1) / 2`



Heaps as an `ArrayList`

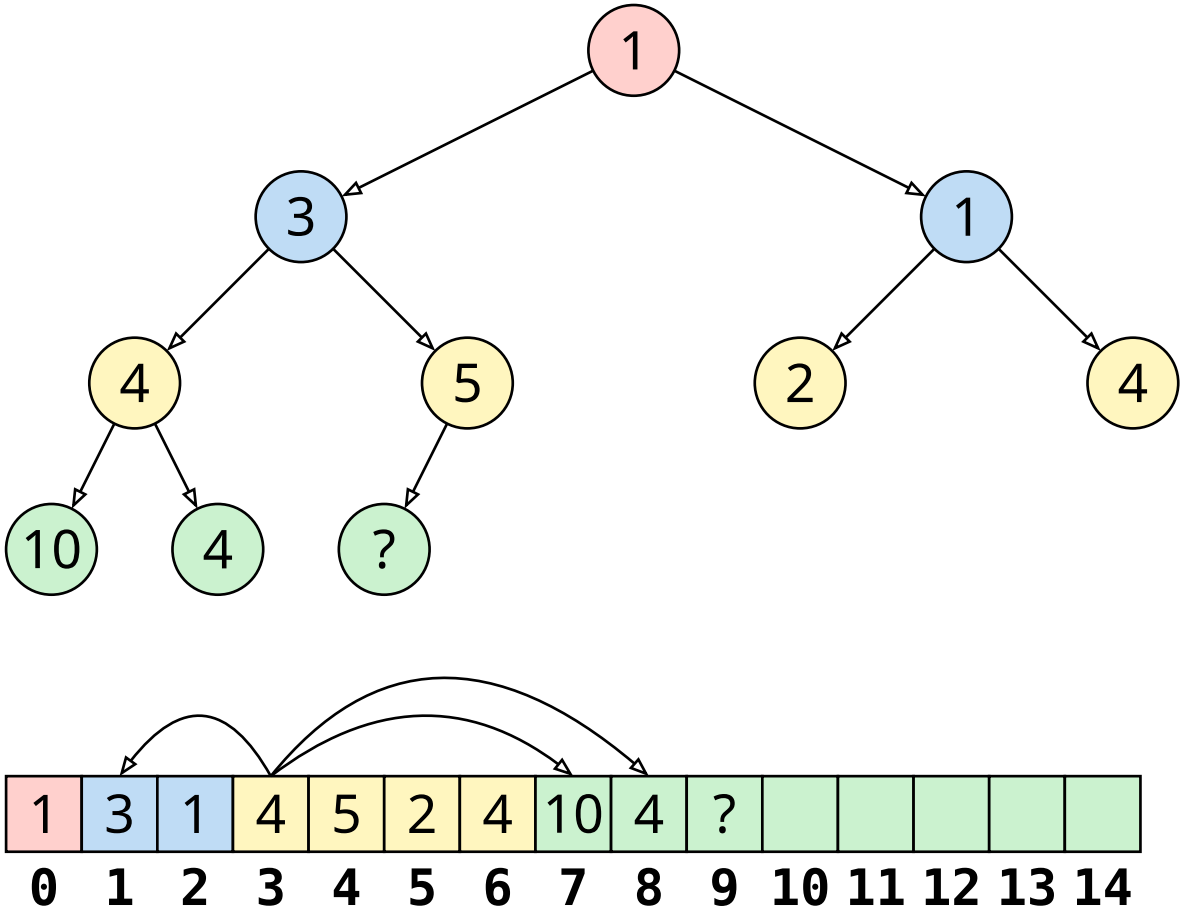
The parent of a node at index **`i`** is at index

`(i - 1) / 2`

The children of a node at index **`i`** are at indices

`(i * 2 + 1)` and

`(i * 2 + 2)`



Runtime Analysis

`pushHeap`

1. Append the new value to our **`ArrayList`**	Amortized $\Theta(1)$
2. Use **`fixUp`** to move the new value up the heap	$O(\log(n))$

Total: Amortized $O(\log(n))$

`popHeap`

1. Swap the root value with the last value	$\Theta(1)$
2. Remove the last value	$\Theta(1)$
3. Use **`fixDown`** to move the new root down the heap	$O(\log(n))$

Total: $O(\log(n))$

PriorityQueue Summary

	Lazy (LinkedList)	Proactive (Sorted LinkedList)	Heap
add	$\Theta(1)$	$O(n)$	Amortized $O(\log(n))$
poll	$\Theta(n)$	$\Theta(1)$	$O(\log(n))$
peek	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

`PriorityQueue` Summary

	Lazy (`LinkedList`)	Proactive (Sorted `LinkedList`)	Heap
`add`	$\Theta(1)$	$O(n)$	Amortized $O(\log(n))$
`poll`	$\Theta(n)$	$\Theta(1)$	$O(\log(n))$
`peek`	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

We met our goal of improving on the worst runtimes of Lazy and Proactive...

But how does this affect our sorting algorithm?

Heap Sort

Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: []

Phase 1

Add everything to our **PriorityQueue**

Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: []

Phase 1

Add everything to our **PriorityQueue**

pq.add(3)

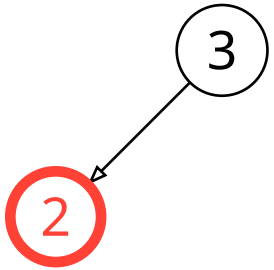
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: []

Phase 1

Add everything to our **PriorityQueue**

pq.add(3)
pq.add(2)



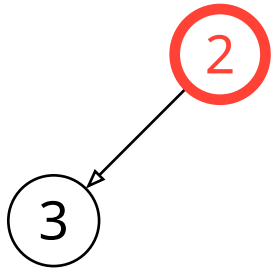
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: []

Phase 1

Add everything to our **PriorityQueue**

pq.add(3)
pq.add(2)



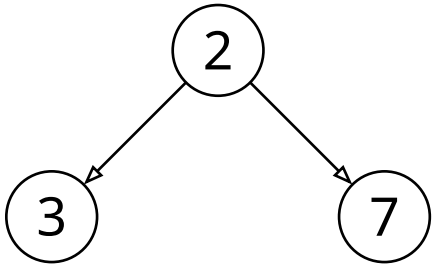
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: []

Phase 1

Add everything to our **PriorityQueue**

```
pq.add(3)
pq.add(2)
pq.add(7)
```



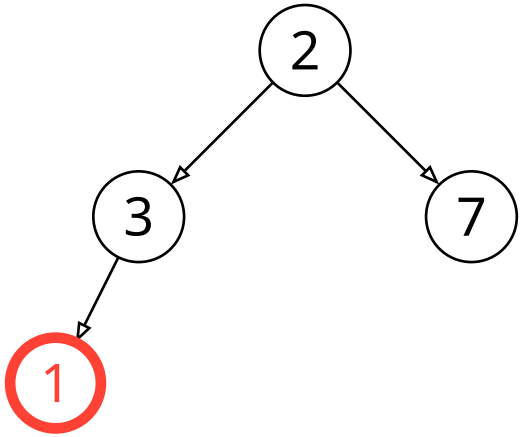
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: []

Phase 1

Add everything to our **PriorityQueue**

```
pq.add(3)
pq.add(2)
pq.add(7)
pq.add(1)
```



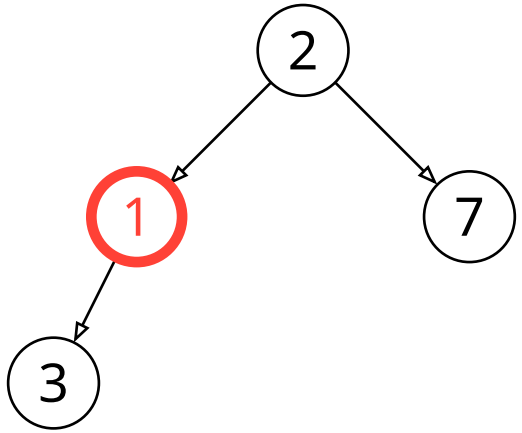
Heap Sort

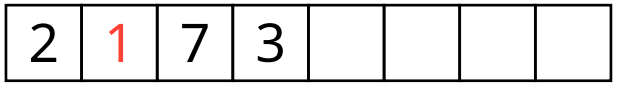
Input: [3,2,7,1,6,8,5,4]
Output: []

Phase 1

Add everything to our **PriorityQueue**

```
pq.add(3)
pq.add(2)
pq.add(7)
pq.add(1)
```





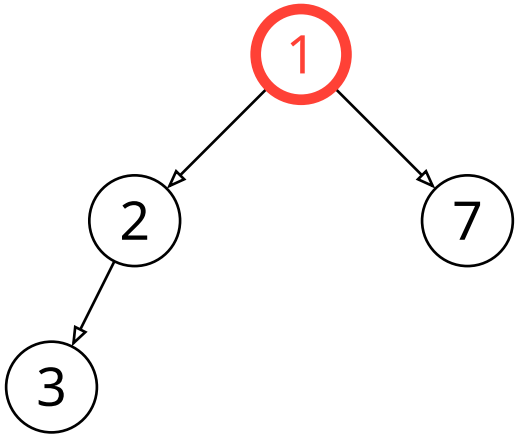
Heap Sort

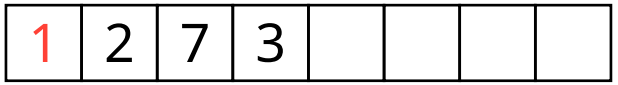
Input: [3,2,7,1,6,8,5,4]
Output: []

Phase 1

Add everything to our **PriorityQueue**

```
pq.add(3)
pq.add(2)
pq.add(7)
pq.add(1)
```





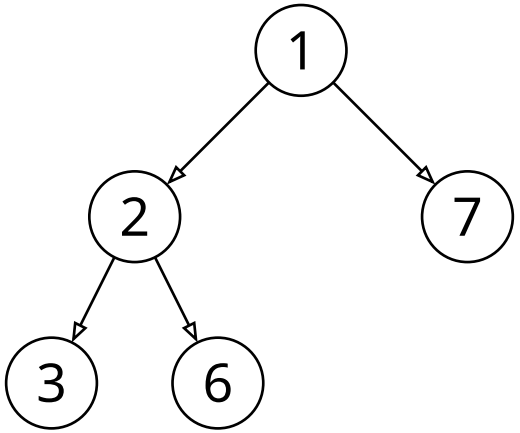
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: []

Phase 1

Add everything to our **PriorityQueue**

```
pq.add(3)
pq.add(2)
pq.add(7)
pq.add(1)
pq.add(6)
```



1	2	7	3	6			

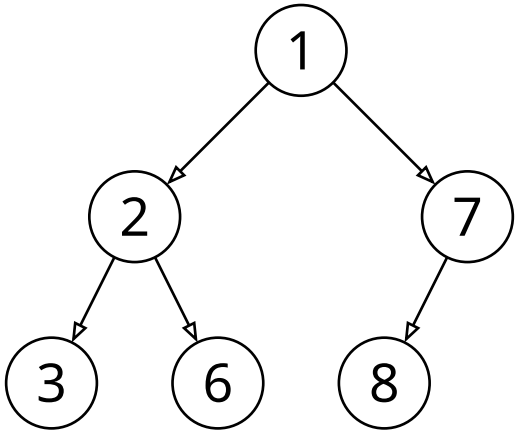
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: []

Phase 1

Add everything to our **PriorityQueue**

```
pq.add(3)
pq.add(2)
pq.add(7)
pq.add(1)
pq.add(6)
pq.add(8)
```



1	2	7	3	6	8		

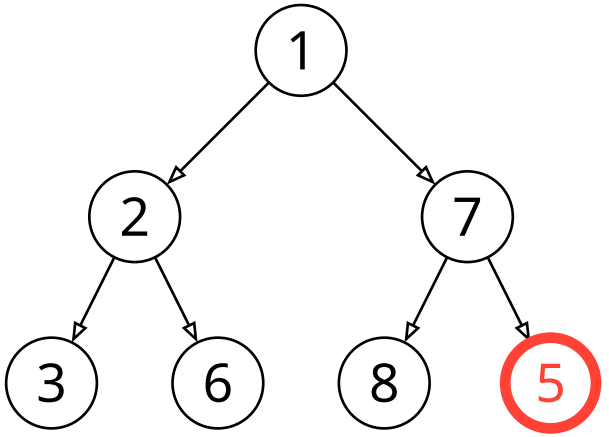
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: []

Phase 1

Add everything to our **PriorityQueue**

```
pq.add(3)
pq.add(2)
pq.add(7)
pq.add(1)
pq.add(6)
pq.add(8)
pq.add(5)
```



1	2	7	3	6	8	5	

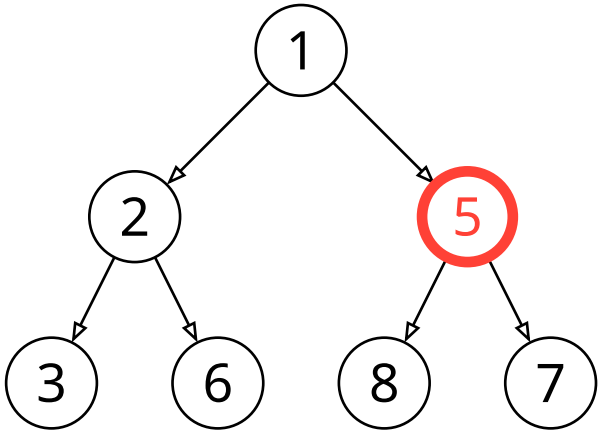
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: []

Phase 1

Add everything to our **PriorityQueue**

```
pq.add(3)
pq.add(2)
pq.add(7)
pq.add(1)
pq.add(6)
pq.add(8)
pq.add(5)
```



1	2	5	3	6	8	7	

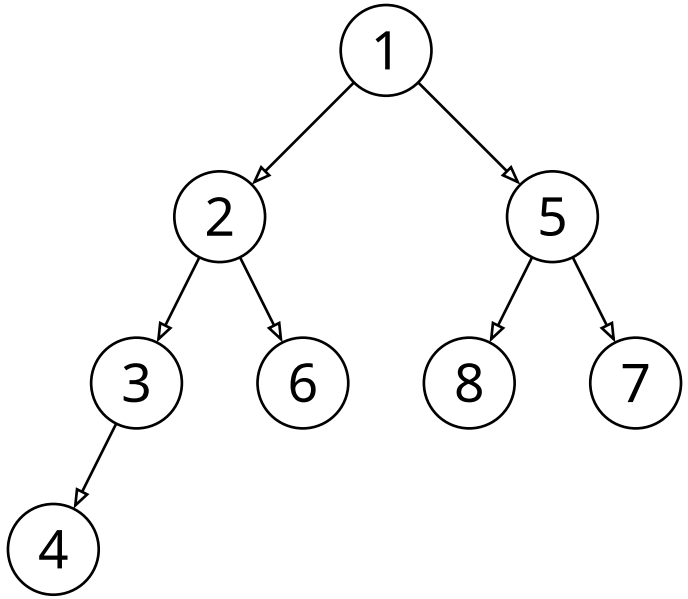
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: []

Phase 1

Add everything to our **PriorityQueue**

```
pq.add(3)
pq.add(2)
pq.add(7)
pq.add(1)
pq.add(6)
pq.add(8)
pq.add(5)
pq.add(4)
```



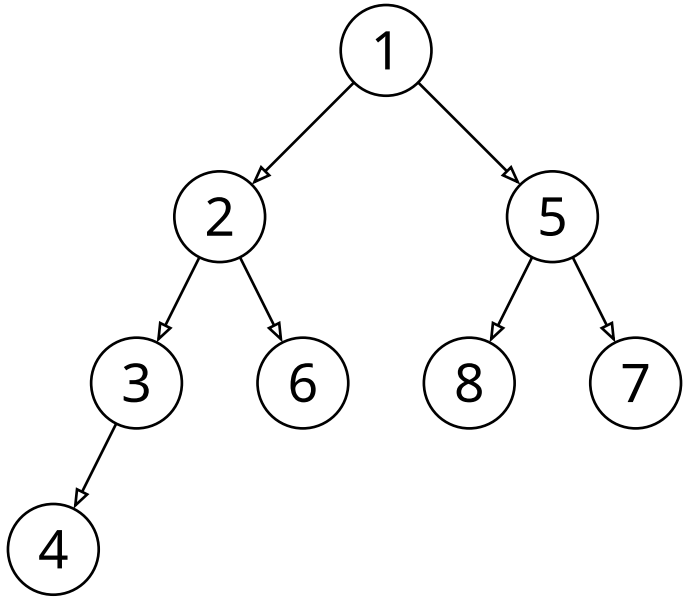
1	2	5	3	6	8	7	4

Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: []

Phase 2

Remove and append to output



1	2	5	3	6	8	7	4

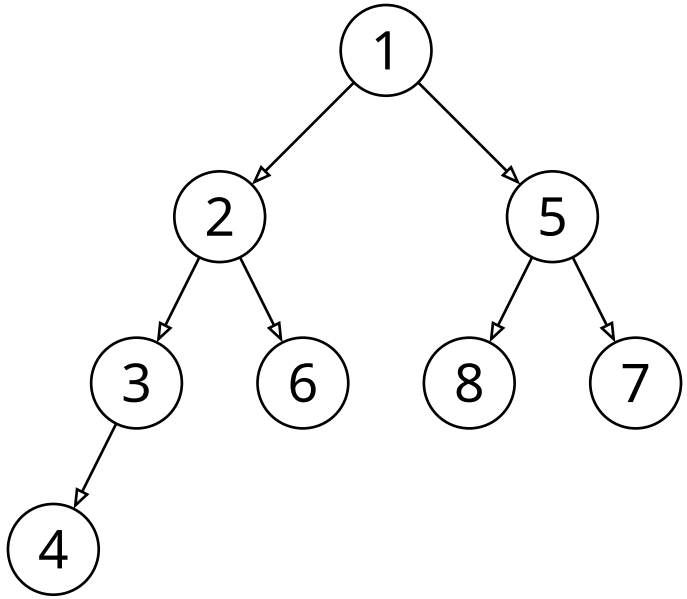
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: []

Phase 2

Remove and append to output

output.add(pq.poll())



1	2	5	3	6	8	7	4

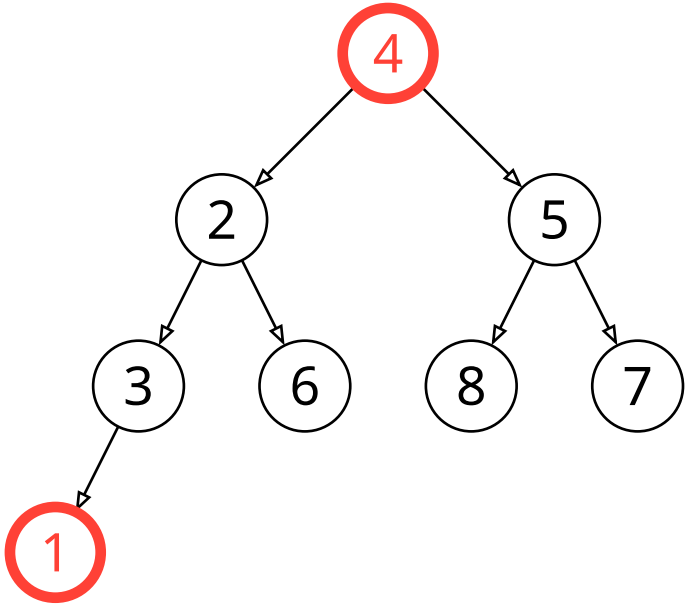
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: []

Phase 2

Remove and append to output

output.add(pq.poll())



4	2	5	3	6	8	7	1

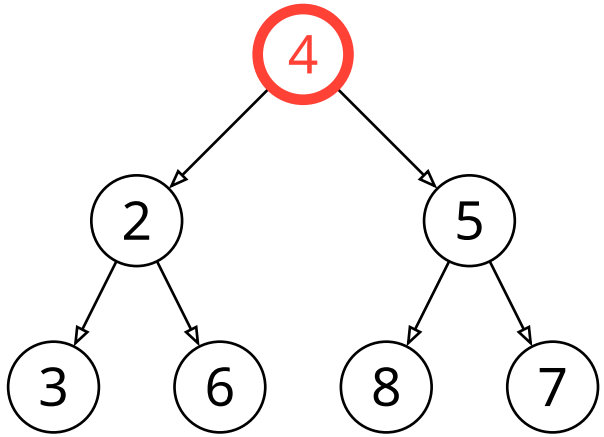
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: []

Phase 2

Remove and append to output

output.add(pq.poll())



Will Return: 1

4	2	5	3	6	8	7	

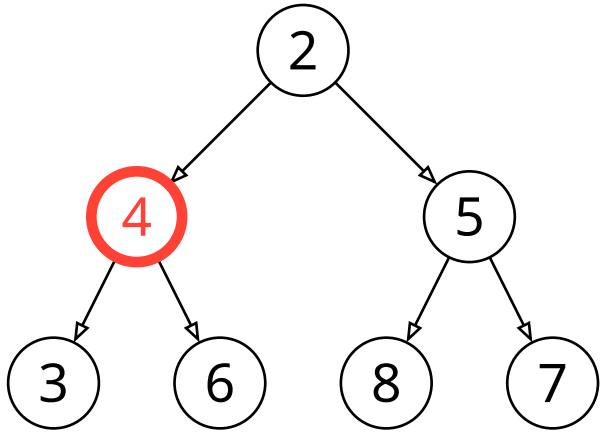
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: []

Phase 2

Remove and append to output

output.add(pq.poll())



Will Return: 1

2	4	5	3	6	8	7	

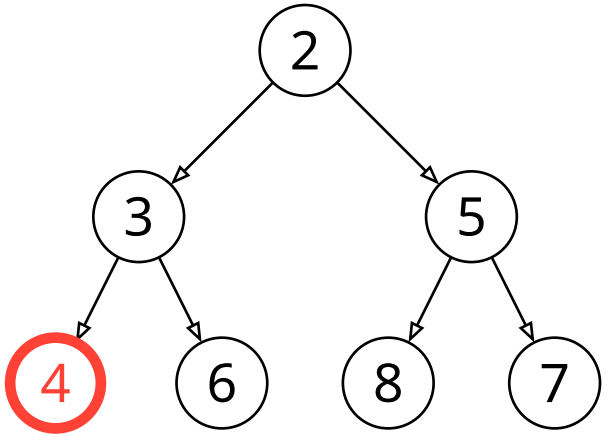
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: []

Phase 2

Remove and append to output

output.add(pq.poll())



Will Return: 1

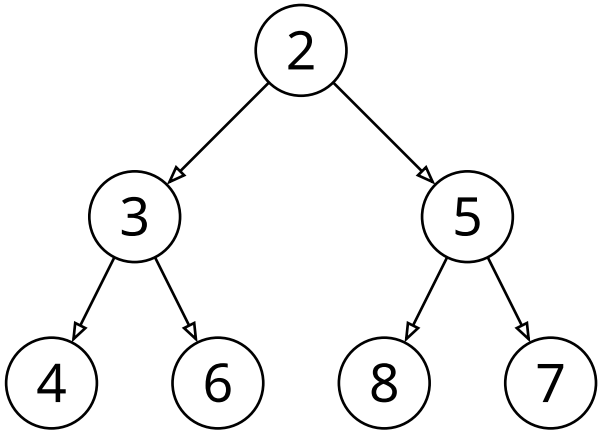
Heap Sort

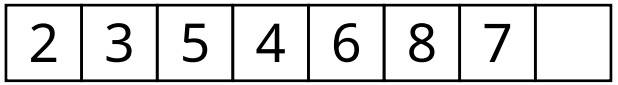
Input: [3,2,7,1,6,8,5,4]
Output: [1]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll())
```





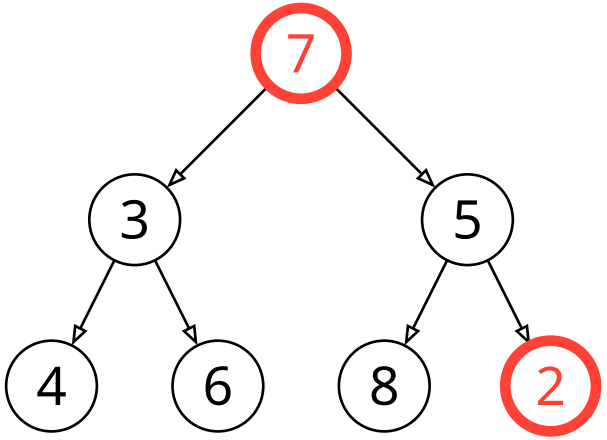
Heap Sort

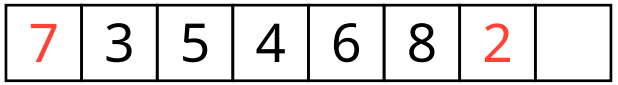
Input: [3,2,7,1,6,8,5,4]
Output: [1]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll())
```





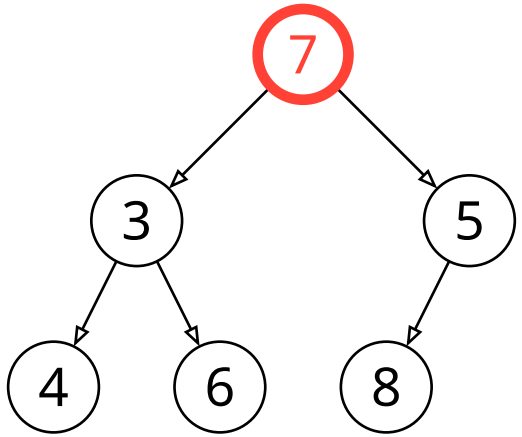
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1]

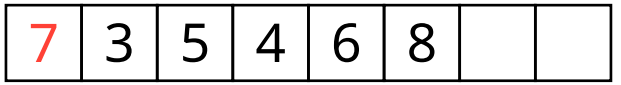
Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll())
```



Will Return: 2



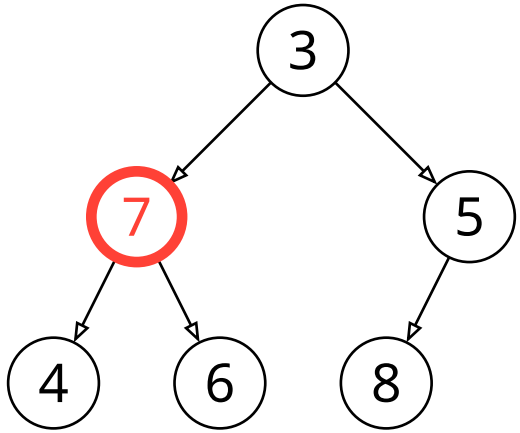
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1]

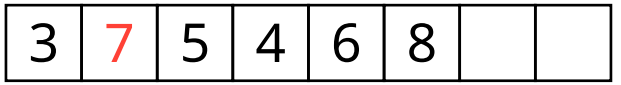
Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll())
```



Will Return: 2



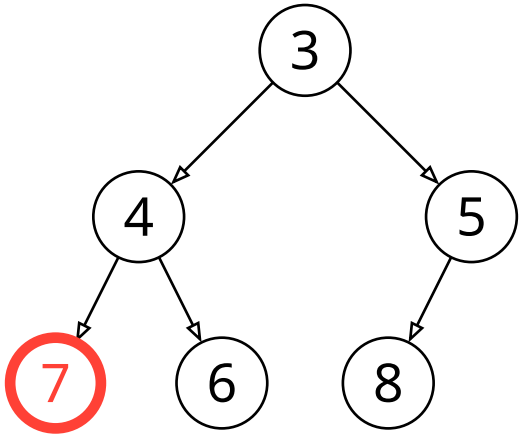
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll())
```



Will Return: 2

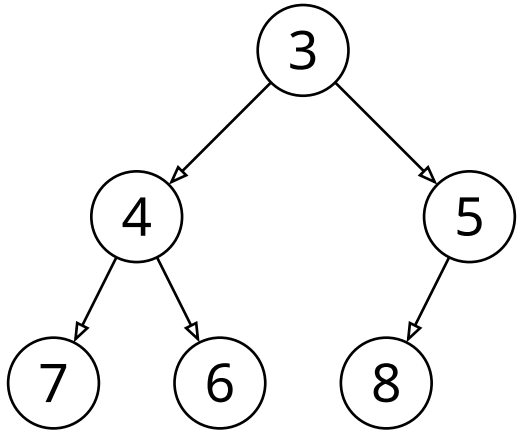
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1,2]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll())
```



3	4	5	7	6	8		

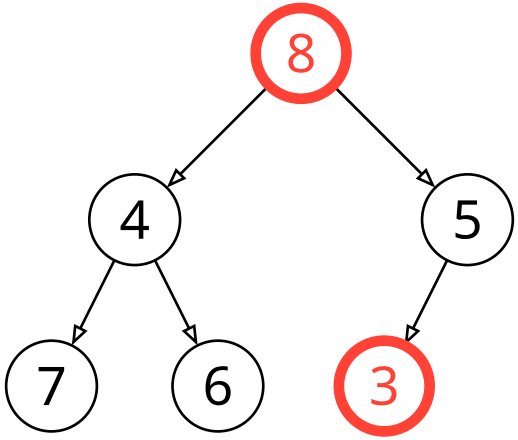
Heap Sort

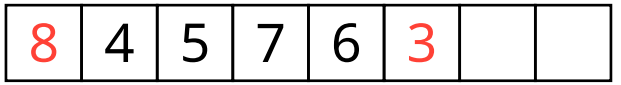
Input: [3,2,7,1,6,8,5,4]
Output: [1,2]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll())
```





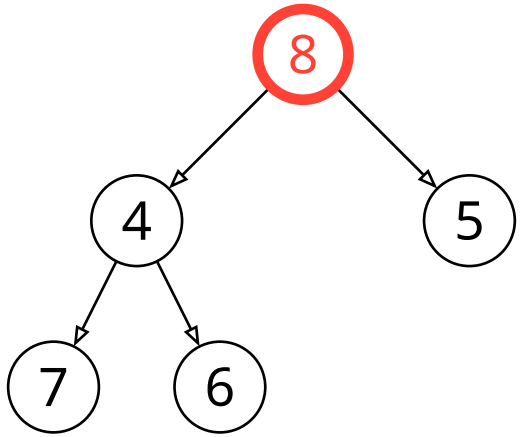
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1,2]

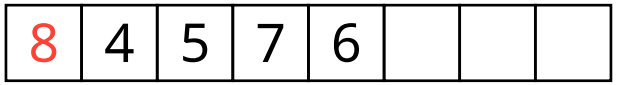
Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll())
```



Will Return: 3



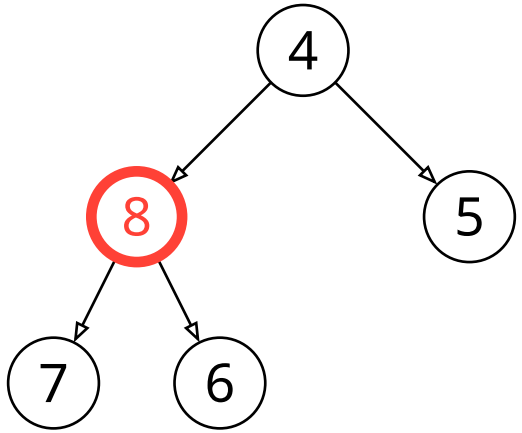
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1,2]

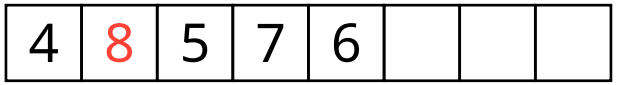
Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll())
```



Will Return: 3



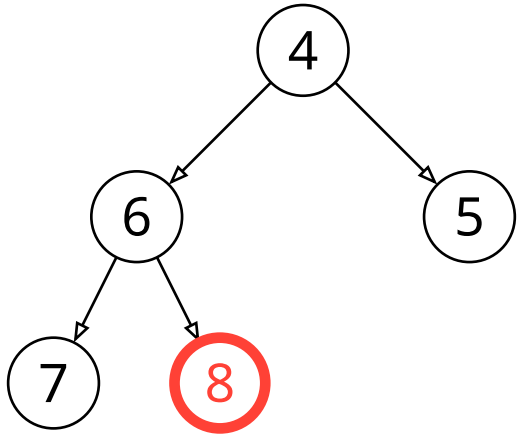
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1,2]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll())
```



Will Return: 3

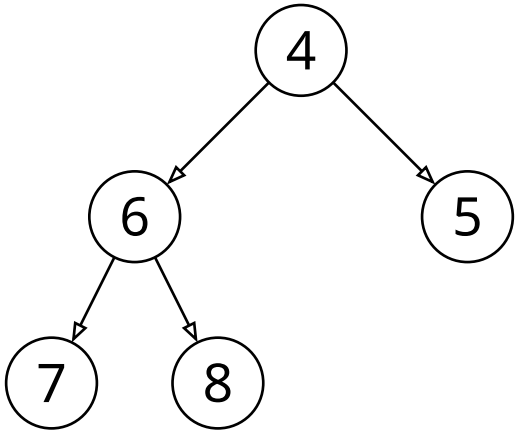
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1,2,3]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll()) // adds 3
output.add(pq.poll())
```



4	6	5	7	8			

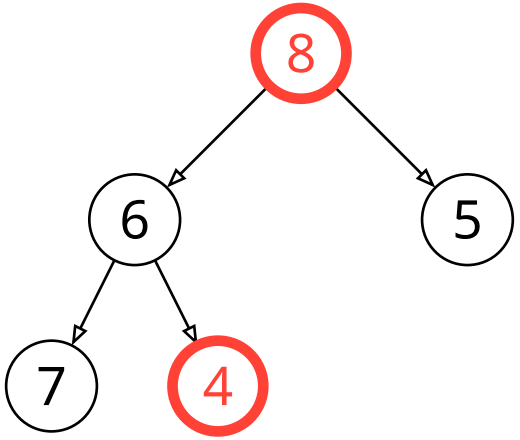
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1,2,3]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll()) // adds 3
output.add(pq.poll())
```



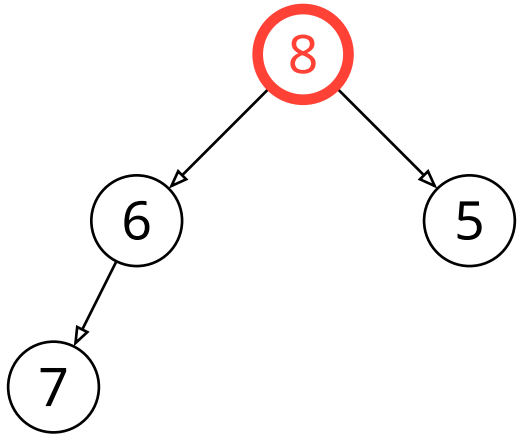
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1,2,3]

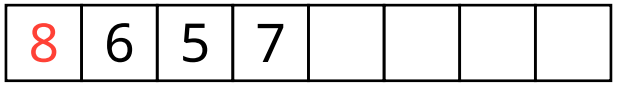
Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll()) // adds 3
output.add(pq.poll())
```



Will Return: 4



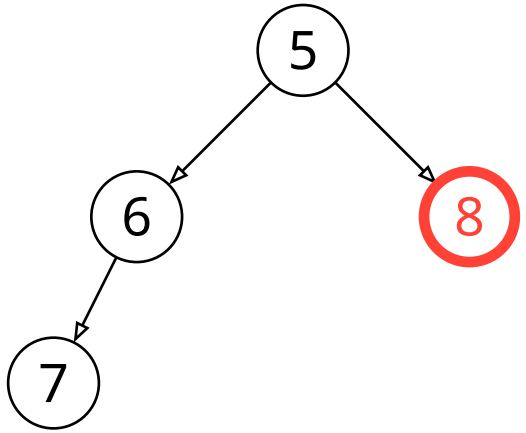
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1,2,3]

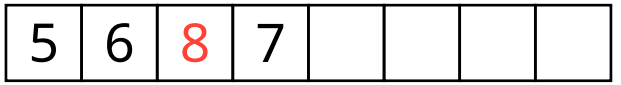
Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll()) // adds 3
output.add(pq.poll())
```



Will Return: 4



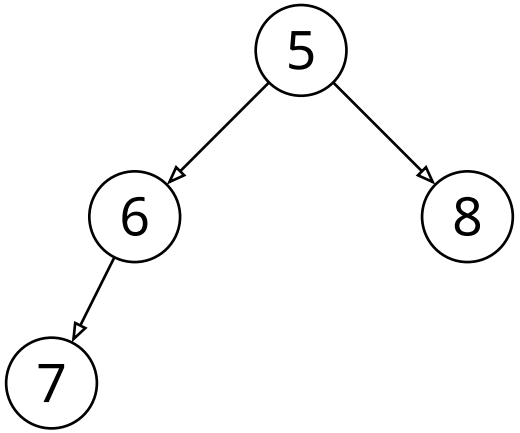
Heap Sort

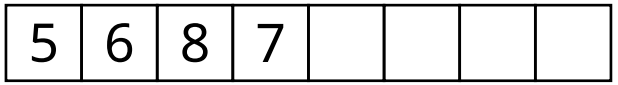
Input: [3,2,7,1,6,8,5,4]
Output: [1,2,3,4]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll()) // adds 3
output.add(pq.poll()) // adds 4
output.add(pq.poll())
```





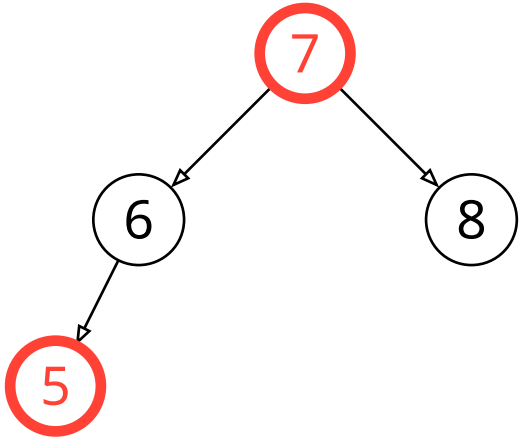
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1,2,3,4]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll()) // adds 3
output.add(pq.poll()) // adds 4
output.add(pq.poll())
```



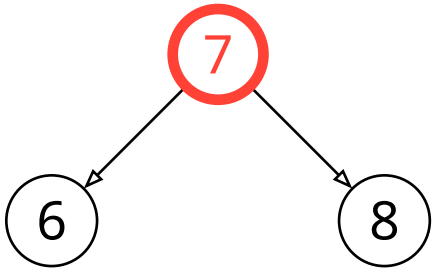
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1,2,3,4]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll()) // adds 3
output.add(pq.poll()) // adds 4
output.add(pq.poll())
```



Will Return: 5

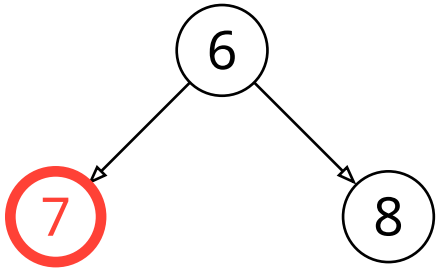
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1,2,3,4]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll()) // adds 3
output.add(pq.poll()) // adds 4
output.add(pq.poll())
```



Will Return: 5

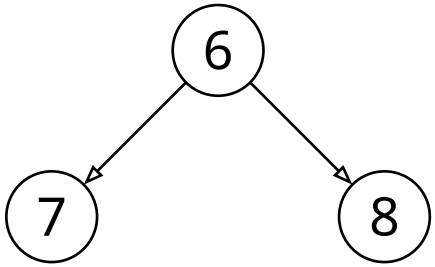
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1,2,3,4,5]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll()) // adds 3
output.add(pq.poll()) // adds 4
output.add(pq.poll()) // adds 5
output.add(pq.poll())
```



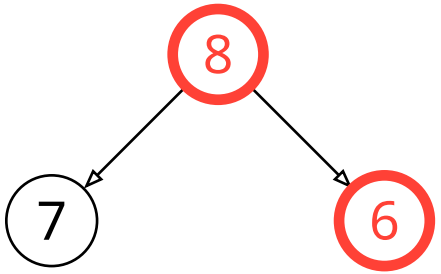
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1,2,3,4,5]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll()) // adds 3
output.add(pq.poll()) // adds 4
output.add(pq.poll()) // adds 5
output.add(pq.poll())
```



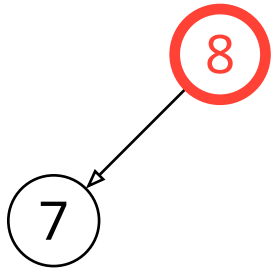
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1,2,3,4,5]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll()) // adds 3
output.add(pq.poll()) // adds 4
output.add(pq.poll()) // adds 5
output.add(pq.poll())
```



Will Return: 6

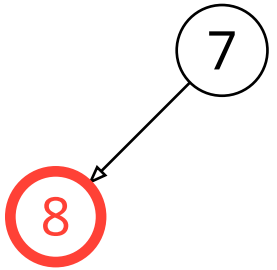
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1,2,3,4,5]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll()) // adds 3
output.add(pq.poll()) // adds 4
output.add(pq.poll()) // adds 5
output.add(pq.poll())
```



Will Return: 6

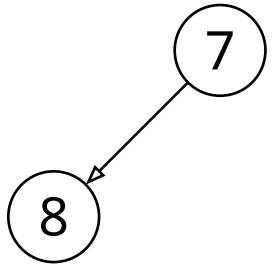
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1,2,3,4,5,6]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll()) // adds 3
output.add(pq.poll()) // adds 4
output.add(pq.poll()) // adds 5
output.add(pq.poll()) // adds 6
output.add(pq.poll())
```



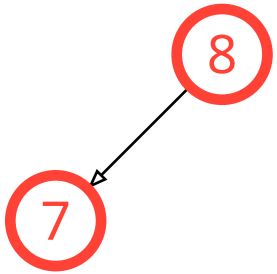
Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1,2,3,4,5,6]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll()) // adds 3
output.add(pq.poll()) // adds 4
output.add(pq.poll()) // adds 5
output.add(pq.poll()) // adds 6
output.add(pq.poll())
```



Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1,2,3,4,5,6]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll()) // adds 3
output.add(pq.poll()) // adds 4
output.add(pq.poll()) // adds 5
output.add(pq.poll()) // adds 6
output.add(pq.poll())
```

Will Return: 7

Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1,2,3,4,5,6,7]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll()) // adds 3
output.add(pq.poll()) // adds 4
output.add(pq.poll()) // adds 5
output.add(pq.poll()) // adds 6
output.add(pq.poll()) // adds 7
output.add(pq.poll())
```

Heap Sort

Input: [3,2,7,1,6,8,5,4]
Output: [1,2,3,4,5,6,7,8]

Phase 2

Remove and append to output

```
output.add(pq.poll()) // adds 1
output.add(pq.poll()) // adds 2
output.add(pq.poll()) // adds 3
output.add(pq.poll()) // adds 4
output.add(pq.poll()) // adds 5
output.add(pq.poll()) // adds 6
output.add(pq.poll()) // adds 7
output.add(pq.poll()) // adds 8
```

Heap Sort Runtime

To add an element to a heap takes amortized $O(\log(n))$ time

Heap Sort Runtime

To add an element to a heap takes amortized $O(\log(n))$ time

So to add everything to our heap in Phase 1 it takes:

$$O(\log(1)) + O(\log(2)) + ... + O(\log(n)) = \sum_{i=1}^{n} O(\log(i))$$

Heap Sort Runtime

To add an element to a heap takes amortized $O(\log(n))$ time

So to add everything to our heap in Phase 1 it takes:

$$O(\log(1)) + O(\log(2)) + ... + O(\log(n)) = \sum_{i=1}^{n} O(\log(i)) \leq \sum_{i=1}^{n} O(\log(n))$$

Heap Sort Runtime

To add an element to a heap takes amortized $O(\log(n))$ time

So to add everything to our heap in Phase 1 it takes:

$$O(\log(1)) + O(\log(2)) + ... + O(\log(n)) = \sum_{i=1}^{n} O(\log(i)) \leq \sum_{i=1}^{n} O(\log(n)) = O(n \log(n))$$

Heap Sort Runtime

To add an element to a heap takes amortized $O(\log(n))$ time

So to add everything to our heap in Phase 1 it takes:

$$O(\log(1)) + O(\log(2)) + ... + O(\log(n)) = \sum_{i=1}^{n} O(\log(i)) \leq \sum_{i=1}^{n} O(\log(n)) = O(n \log(n))$$

To remove the smallest value from a heap takes $O(\log(n))$ time

Heap Sort Runtime

To add an element to a heap takes amortized $O(\log(n))$ time

So to add everything to our heap in Phase 1 it takes:

$$O(\log(1)) + O(\log(2)) + ... + O(\log(n)) = \sum_{i=1}^{n} O(\log(i)) \leq \sum_{i=1}^{n} O(\log(n)) = O(n \log(n))$$

To remove the smallest value from a heap takes $O(\log(n))$ time

So to remove everything from our heap in Phase 2 it takes:

$$O(\log(n)) + O(\log(n-1)) + ... + O(\log(1)) = \sum_{i=1}^{n} O(\log(i))$$

Heap Sort Runtime

To add an element to a heap takes amortized $O(\log(n))$ time

So to add everything to our heap in Phase 1 it takes:

$$O(\log(1)) + O(\log(2)) + ... + O(\log(n)) = \sum_{i=1}^{n} O(\log(i)) \leq \sum_{i=1}^{n} O(\log(n)) = O(n \log(n))$$

To remove the smallest value from a heap takes $O(\log(n))$ time

So to remove everything from our heap in Phase 2 it takes:

$$O(\log(n)) + O(\log(n-1)) + ... + O(\log(1)) = \sum_{i=1}^{n} O(\log(i)) \leq \sum_{i=1}^{n} O(\log(n))$$

Heap Sort Runtime

To add an element to a heap takes amortized $O(\log(n))$ time

So to add everything to our heap in Phase 1 it takes:

$$O(\log(1)) + O(\log(2)) + ... + O(\log(n)) = \sum_{i=1}^{n} O(\log(i)) \leq \sum_{i=1}^{n} O(\log(n)) = O(n \log(n))$$

To remove the smallest value from a heap takes $O(\log(n))$ time

So to remove everything from our heap in Phase 2 it takes:

$$O(\log(n)) + O(\log(n-1)) + ... + O(\log(1)) = \sum_{i=1}^{n} O(\log(i)) \leq \sum_{i=1}^{n} O(\log(n)) = O(n \log(n))$$

Heap Sort Runtime

To add an element to a heap takes amortized $O(\log(n))$ time

So to add everything to our heap in Phase 1 it takes:

$$O(\log(1)) + O(\log(2)) + ... + O(\log(n)) = \sum_{i=1}^{n} O(\log(i)) \leq \sum_{i=1}^{n} O(\log(n)) = O(n \log(n))$$

To remove the smallest value from a heap takes $O(\log(n))$ time

So to remove everything from our heap in Phase 2 it takes:

$$O(\log(n)) + O(\log(n-1)) + ... + O(\log(1)) = \sum_{i=1}^{n} O(\log(i)) \leq \sum_{i=1}^{n} O(\log(n)) = O(n \log(n))$$

Full sort runtime: $O(n \log(n)) + O(n \log(n)) = O(n \log(n))$
(also $\Omega(n \log(n))$ but that proof is trickier...)

Sort Runtimes (so far...)

Algorithm	Worst-Case Runtime
Bubble Sort	$O(n^2)$
Selection Sort	$O(n^2)$
Insertion Sort	$O(n^2)$
Heap Sort	$O(n \log(n))$

Sort Runtimes (so far...)

Algorithm	Worst-Case Runtime
Bubble Sort	$O(n^2)$
Selection Sort	$O(n^2)$
Insertion Sort	$O(n^2)$
Heap Sort	$O(n \log(n))$

We see clear improvement when using a **`Heap`** vs our Lazy/Proactive **`PriorityQueue`** implementations for sorting

Sort Runtimes (so far...)

Algorithm	Worst-Case Runtime
Bubble Sort	$O(n^2)$
Selection Sort	$O(n^2)$
Insertion Sort	$O(n^2)$
Heap Sort	$O(n \log(n))$

We see clear improvement when using a **`Heap`** vs our Lazy/Proactive **`PriorityQueue`** implementations for sorting

Fun fact: $O(n \log(n))$ is the best possible Big-O bound you can get for sorting (you may see a proof of this in 331)

Sort Runtimes (so far...)

Algorithm	Worst-Case Runtime
Bubble Sort	$O(n^2)$
Selection Sort	$O(n^2)$
Insertion Sort	$O(n^2)$
Heap Sort	$O(n \log(n))$

We see clear improvement when using a **`Heap`** vs our Lazy/Proactive **`PriorityQueue`** implementations for sorting

Fun fact: $O(n \log(n))$ is the best possible Big-O bound you can get for sorting (you may see a proof of this in 331)

Note: Optimized implementations of Selection, Insertion, and Heap Sort can be done in-place rather than adding/removing everything from a **`PriorityQueue`**, but the asymptotic performance remains the same

Revisiting Dijkstra’s Algorithm

Dijkstra's Algorithm

```
 1 todo = PriorityQueue()
 2 todo.add((start, 0))
 3 while not todo.isEmpty():
 4   ((curr, dist)) = todo.poll()
 5   if curr.label != VISITED:
 6     curr.label = VISITED
 7     for e in curr.outgoingEdges():
 8       next = e.dest
 9       if next.label != VISITED:
10         todo.add((next, dist+e.weight))
```

Dijkstra's Algorithm

```
 1  todo = PriorityQueue()
 2  todo.add((start, 0))
 3  while not todo.isEmpty():
 4    ((curr, dist)) = todo.poll()
 5    if curr.label != VISITED:
 6      curr.label = VISITED
 7      for e in curr.outgoingEdges():
 8        next = e.dest
 9        if next.label != VISITED:
10          todo.add((next, dist+e.weight))
```

Our TODO now keeps track of discovered verties (and their distance from start)

Dijkstra's Algorithm

```
 1  todo = PriorityQueue()
 2  todo.add((start, 0))
 3  while not todo.isEmpty():
 4    ((curr, dist)) = todo.poll()
 5    if curr.label != VISITED:
 6      curr.label = VISITED
 7      for e in curr.outgoingEdges():
 8        next = e.dest
 9        if next.label != VISITED:
10          todo.add((next, dist+e.weight))
```

The only thing on our TODO list when we start is our starting vertex (which is 0 units away)

Dijkstra's Algorithm

```
 1  todo = PriorityQueue()
 2  todo.add((start, 0))
 3  while not todo.isEmpty():
 4    ((curr, dist)) = todo.poll()
 5    if curr.label != VISITED:
 6      curr.label = VISITED
 7      for e in curr.outgoingEdges():
 8        next = e.dest
 9        if next.label != VISITED:
10          todo.add((next, dist+e.weight))
```

Our TODO list is a **`PriorityQueue`**, giving higher priority to closer vertices

The vertex that is removed is therefore the **closest vertex to our starting point**

Dijkstra's Algorithm

```
 1  todo = PriorityQueue()
 2  todo.add((start, 0))
 3  while not todo.isEmpty():
 4    ((curr, dist)) = todo.poll()
 5    if curr.label != VISITED:
 6      curr.label = VISITED
 7      for e in curr.outgoingEdges():
 8        next = e.dest
 9        if next.label != VISITED:
10          todo.add((next, dist+e.weight))
```

When we find something new, add it to our TODO list, along with the distance it is from start

IMPORTANT: We've only discovered the vertex, we haven't visited it yet!

Dijkstra's Algorithm

```
 1  todo = PriorityQueue()
 2  todo.add((start, 0))
 3  while not todo.isEmpty():
 4    ((curr, dist)) = todo.poll()
 5    if curr.label != VISITED:
 6      curr.label = VISITED
 7      for e in curr.outgoingEdges():
 8        next = e.dest
 9        if next.label != VISITED:
10          todo.add((next, dist+e.weight))
```

Vertices only get visited when we remove them from our TODO (and they haven't already been visited)

Dijkstra's Analysis

How many times is each `Vertex` added to our TODO list?

Dijkstra's Analysis

How many times is each `Vertex` added to our TODO list?

$O(\deg(v))$ times!

We can't mark it as **`VISITED`** until we remove it from the TODO list, so we might discover it many times!

How many times is each `Vertex` removed from our TODO list?

Dijkstra's Analysis

How many times is each `Vertex` added to our TODO list?

$O(\deg(v))$ times!

We can't mark it as **`VISITED`** until we remove it from the TODO list, so we might discover it many times!

How many times is each `Vertex` removed from our TODO list?

$O(\deg(v))$ times (we remove it as many times as we add it)

What is the total number of add / remove operations?

$$\sum_{v \in V} \deg(v) = 2|E|$$

Therefore we do a total of $O(|E|)$ adds and $O(|E|)$ removes

Dijkstra's Analysis

What is the cost to add / remove the vertex from our TODO list?

Dijkstra's Analysis

What is the cost to add / remove the vertex from our TODO list?

If our **`PriorityQueue`** is a **`Heap`**, $O(\log(n))$ time to add / remove

What is the maximum size of our `Heap`?

Dijkstra's Analysis

What is the cost to add / remove the vertex from our TODO list?

If our **`PriorityQueue`** is a **`Heap`**, $O(\log(n))$ time to add / remove

What is the maximum size of our `Heap`? $O(|E|)$

Total Runtime:

Dijkstra's Analysis

What is the cost to add / remove the vertex from our TODO list?

If our **`PriorityQueue`** is a **`Heap`**, $O(\log(n))$ time to add / remove

What is the maximum size of our `Heap`? $O(|E|)$

Total Runtime: $O(|E| \log(|E|))$

Note: There are optimizations that can be made to bring the runtime down to $O(|E| \log(|V|))$ or $O(|E| + |V| \log(|V|))$

Bonus: Heapify

Heapify

How long would it take to create a heap of n items by inserting them one at a time?

Heapify

How long would it take to create a heap of n items by inserting them one at a time?

n insertions, each costing $O(\log(n))$

Total: $O(n \log(n))$

Heapify

How long would it take to create a heap of n items by inserting them one at a time?

n insertions, each costing $O(\log(n))$

Total: $O(n \log(n))$

What if we know all of the elements up front, can we do better?

Challenge: Given an arbitrary array of data, turn it into a heap

Heapify

How long would it take to create a heap of n items by inserting them one at a time?

n insertions, each costing $O(\log(n))$

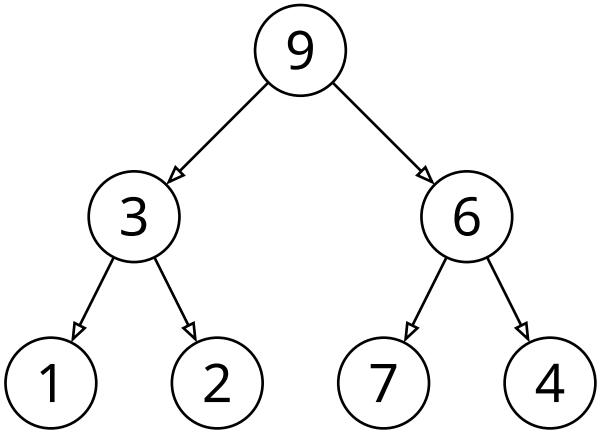
Total: $O(n \log(n))$

What if we know all of the elements up front, can we do better?

Challenge: Given an arbitrary array of data, turn it into a heap

Idea: Call either **`fixUp`** or **`fixDown`** on every element

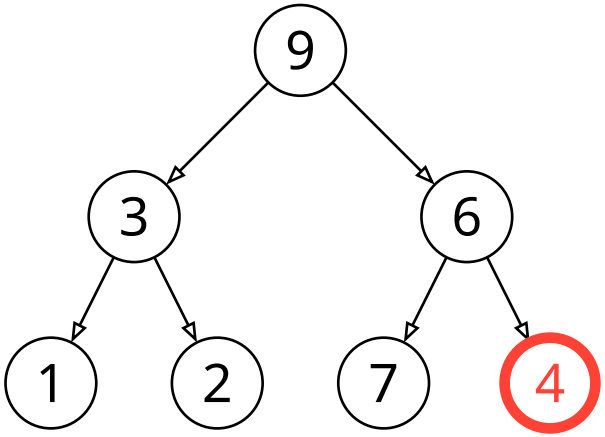
Heapify Example



9	3	6	1	2	7	4

Heapify Example

Start at the bottom (back of the array) and **`fixDown`** each element...



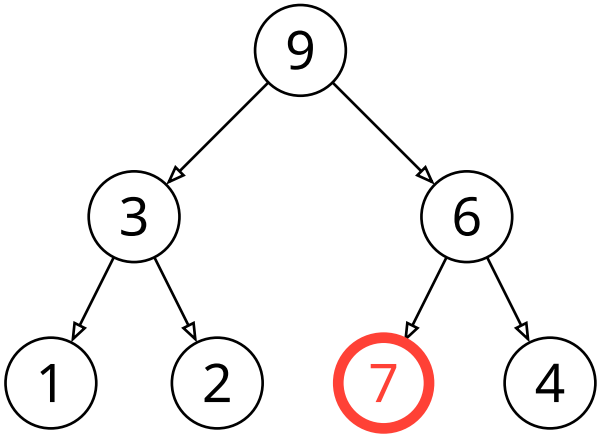
9	3	6	1	2	7	4

Heapify Example

Start at the bottom (back of the array) and **`fixDown`** each element…

The element can’t go down any further!

0 steps per element in the bottom level



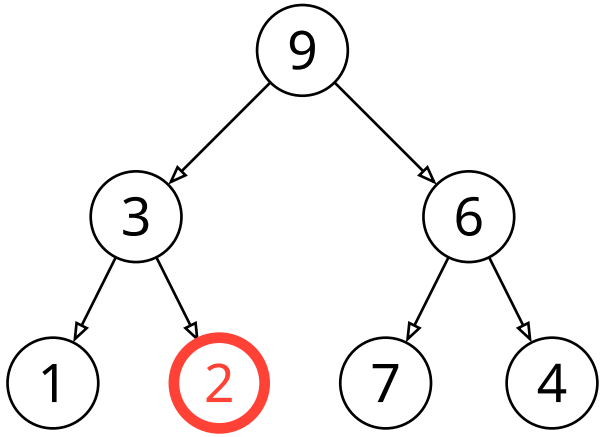
9	3	6	1	2	7	4

Heapify Example

Start at the bottom (back of the array) and **`fixDown`** each element…

The element can’t go down any further!

0 steps per element in the bottom level



9	3	6	1	2	7	4

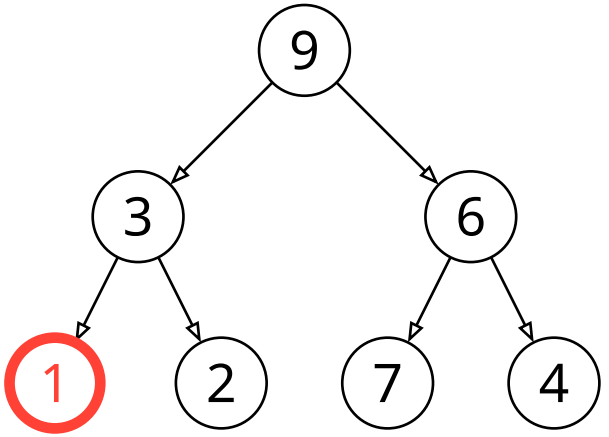
Heapify Example

Start at the bottom (back of the array) and **`fixDown`** each element…

The element can’t go down any further!

0 steps per element in the bottom level

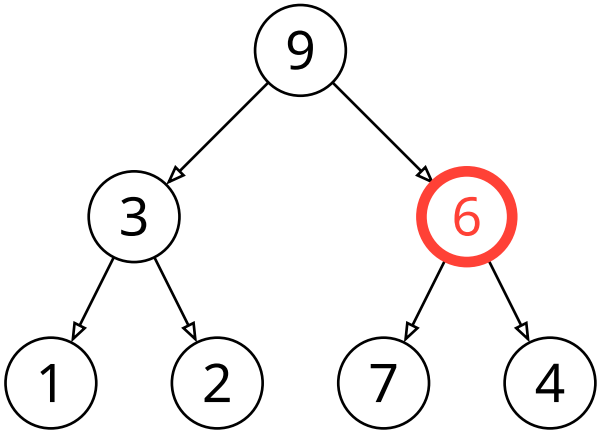
We’ve now fixed over half of the array without doing any work!



9	3	6	1	2	7	4

Heapify Example

Continue to the next lowest level...

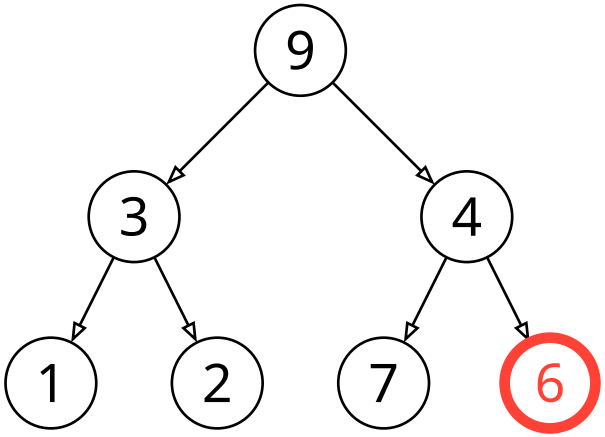


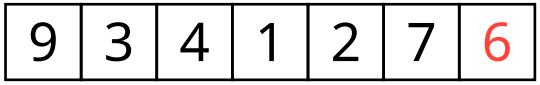
9	3	6	1	2	7	4

Heapify Example

Continue to the next lowest level...

Each of these elements require **AT MOST** 1 swap

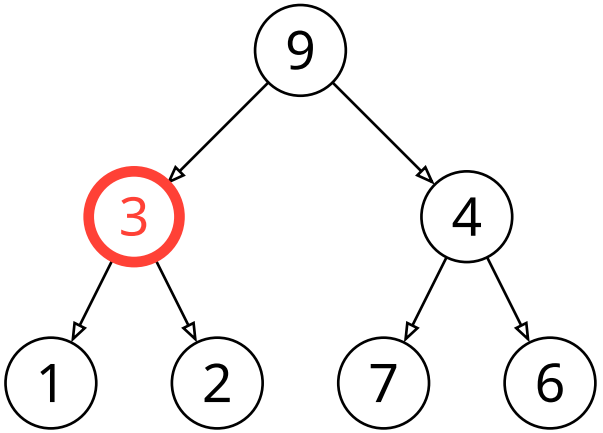


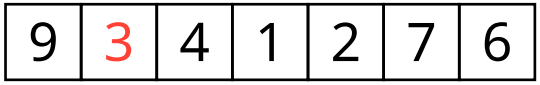


Heapify Example

Continue to the next lowest level...

Each of these elements require **AT MOST** 1 swap



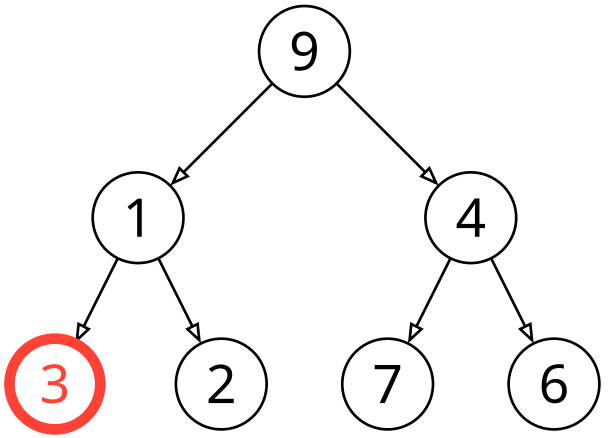


Heapify Example

Continue to the next lowest level...

Each of these elements require **AT MOST** 1 swap

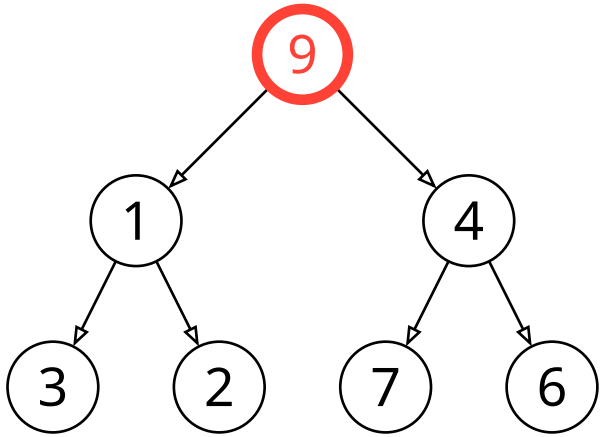
Notice how each subtree is now a valid heap...



9	1	4	3	2	7	6

Heapify Example

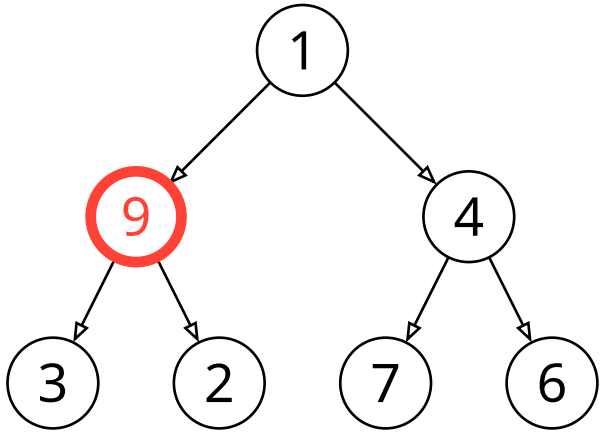
When we finally reach the root, it may take up to $\log(n)$ swaps...



Heapify Example

When we finally reach the root, it may take up to $\log(n)$ swaps...

But there's only one root!



1	9	4	3	2	7	6

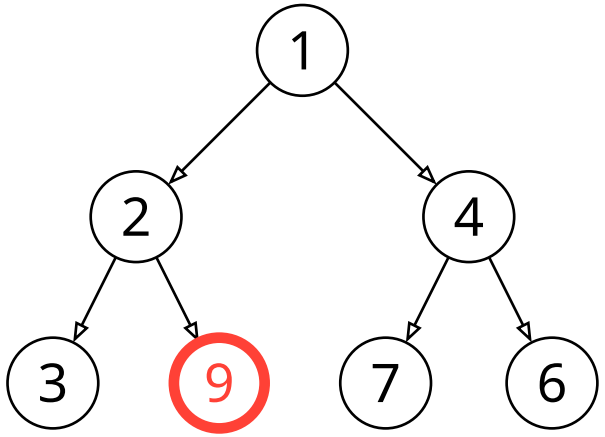
Heapify Example

When we finally reach the root, it may take up to $\log(n)$ swaps...

But there's only one root!

We now have a valid heap!

Note: It's still not fully sorted



1	2	4	3	9	7	6

Heapify Runtime

So how long did that take...?

Heapify Runtime

So how long did that take...?

Depth 0: 1 node requiring $O(\log(n))$ swaps

Heapify Runtime

So how long did that take...?

Depth 0: 1 node requiring $O(\log(n))$ swaps

Depth 1: 2 nodes requiring $O(\log(n)) - 1$ swaps each

Heapify Runtime

So how long did that take...?

Depth 0: 1 node requiring $O(\log(n))$ swaps

Depth 1: 2 nodes requiring $O(\log(n)) - 1$ swaps each

Depth 2: 4 nodes requiring $O(\log(n)) - 2$ swaps each

Heapify Runtime

So how long did that take...?

Depth 0: 1 node requiring $O(\log(n))$ swaps

Depth 1: 2 nodes requiring $O(\log(n)) - 1$ swaps each

Depth 2: 4 nodes requiring $O(\log(n)) - 2$ swaps each

...

Depth $\log(n) - 1$: $\frac{n}{4}$ nodes requiring 1 swap each

Heapify Runtime

So how long did that take...?

Depth 0: 1 node requiring $O(\log(n))$ swaps

Depth 1: 2 nodes requiring $O(\log(n)) - 1$ swaps each

Depth 2: 4 nodes requiring $O(\log(n)) - 2$ swaps each

...

Depth $\log(n) - 1$: $\frac{n}{4}$ nodes requiring 1 swap each

Depth $\log(n)$: $\frac{n}{2}$ nodes requiring 0 swap each

Heapify Runtime

So how long did that take...?

Depth 0: 1 node requiring $O(\log(n))$ swaps

Depth 1: 2 nodes requiring $O(\log(n)) - 1$ swaps each

Depth 2: 4 nodes requiring $O(\log(n)) - 2$ swaps each

...

Depth $\log(n) - 1$: $\frac{n}{4}$ nodes requiring 1 swap each

Depth $\log(n)$: $\frac{n}{2}$ nodes requiring 0 swap each

$$\sum_{i=0}^{\log(n)} i \cdot \frac{n}{2^i}$$

Heapify Runtime

So how long did that take...?

Depth 0: 1 node requiring $O(\log(n))$ swaps

Depth 1: 2 nodes requiring $O(\log(n)) - 1$ swaps each

Depth 2: 4 nodes requiring $O(\log(n)) - 2$ swaps each

...

Depth $\log(n) - 1$: $\frac{n}{4}$ nodes requiring 1 swap each

Depth $\log(n)$: $\frac{n}{2}$ nodes requiring 0 swap each

$$\sum_{i=0}^{\log(n)} i \cdot \frac{n}{2^i} = n \sum_{i=0}^{\log(n)} \frac{i}{2^i}$$

Heapify Runtime

So how long did that take...?

Depth 0: 1 node requiring $O(\log(n))$ swaps

Depth 1: 2 nodes requiring $O(\log(n)) - 1$ swaps each

Depth 2: 4 nodes requiring $O(\log(n)) - 2$ swaps each

...

Depth $\log(n) - 1$: $\frac{n}{4}$ nodes requiring 1 swap each

Depth $\log(n)$: $\frac{n}{2}$ nodes requiring 0 swap each

$$\sum_{i=0}^{\log(n)} i \cdot \frac{n}{2^i} = n \sum_{i=0}^{\log(n)} \frac{i}{2^i} \leq n \in O(n)$$

Heapify Conclusion

To build a heap from an arbitrary array using heapify takes $O(n)$ time

Heapify Conclusion

To build a heap from an arbitrary array using heapify takes $O(n)$ time

To build the same heap by inserting one at a time takes $O(n \log(n))$ time!

Heapify Conclusion

To build a heap from an arbitrary array using heapify takes $O(n)$ time

To build the same heap by inserting one at a time takes $O(n \log(n))$ time!

Sometimes you can solve problems faster if you have all of the data up front!

We see similar ideas with sorting:

- Inserting n elements into a sorted list would take $O(n^2)$ time
- But sorting a list with those elements would only take $O(n \log(n))$ time (with Heap Sort for example)