CSCI 232:
Data Structures and Algorithms

Heaps

Reese Pearsall

Spring 2025

Announcements

Lab 6 due **Friday** at 11:59 PM

Program 2 due **Sunday** at 11:59 PM

There will be a lab next week,
but I will try to make it easy

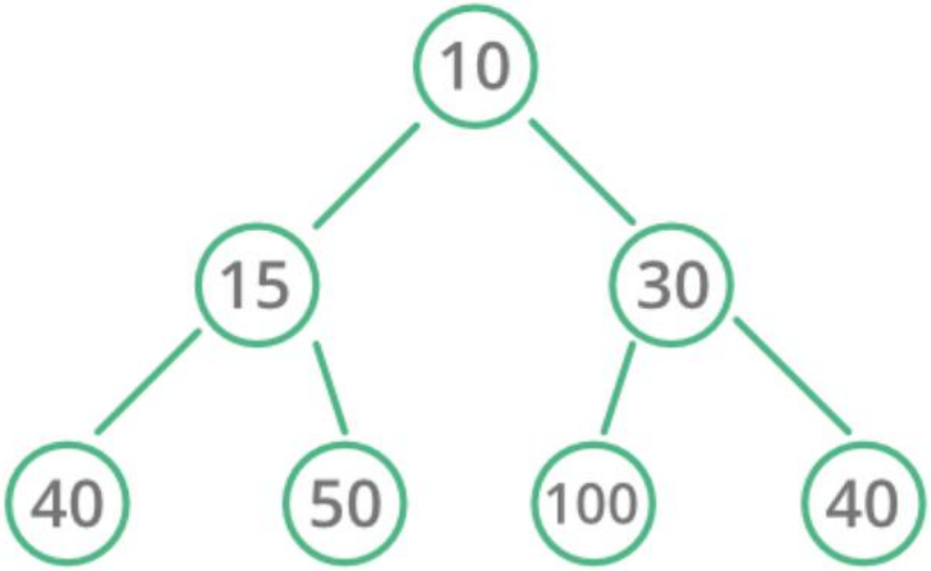
Tweaked a few dates on the schedule

- 10 labs → 11 labs
  * I will now drop your lowest lab grade
- Quiz 2 moved back a week, Quiz 3 will take place in this classroom during finals week

the game that you play in nightmares ^

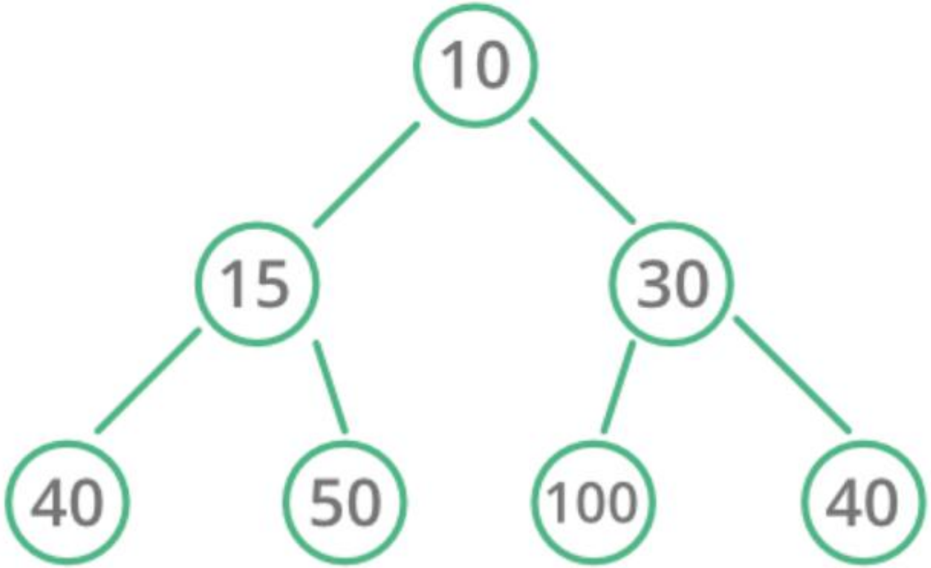
Quiz 1

The **Heap** data structure is complete binary tree that follows the heap property

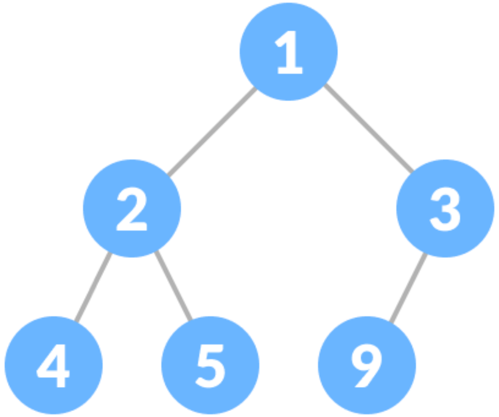


The **Heap** data structure is **complete** binary tree that follows the heap property

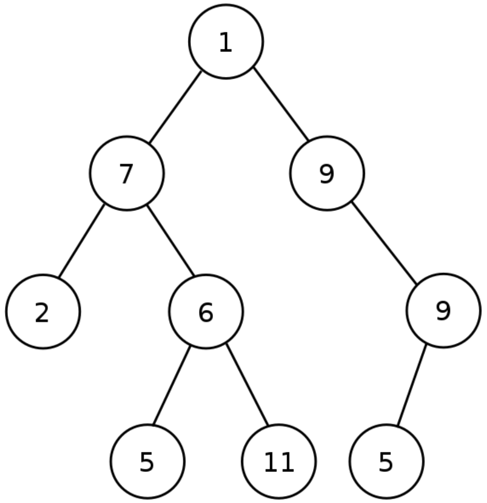
Complete tree - Every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible



complete



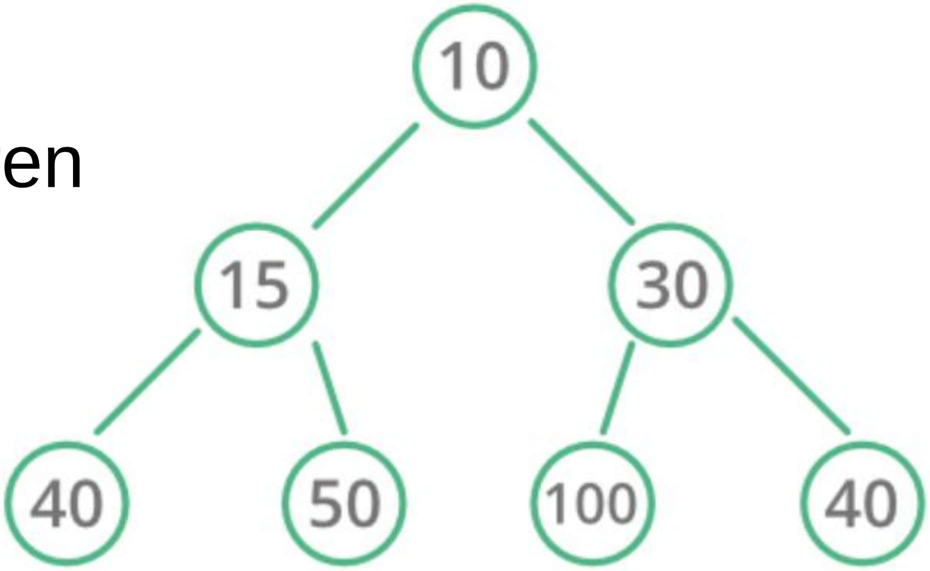
complete



Not complete

The **Heap** data structure is complete **binary** tree that follows the heap property

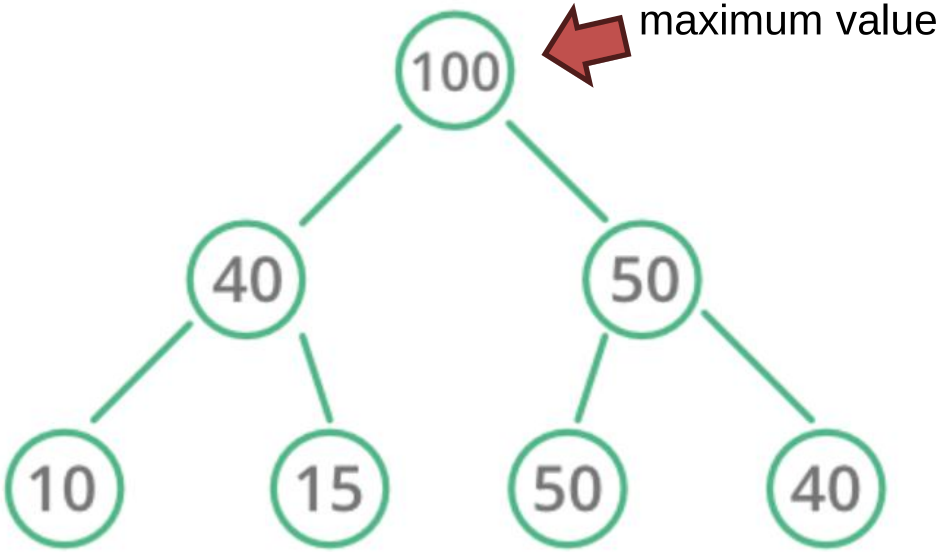
Binary – cannot have more than two children



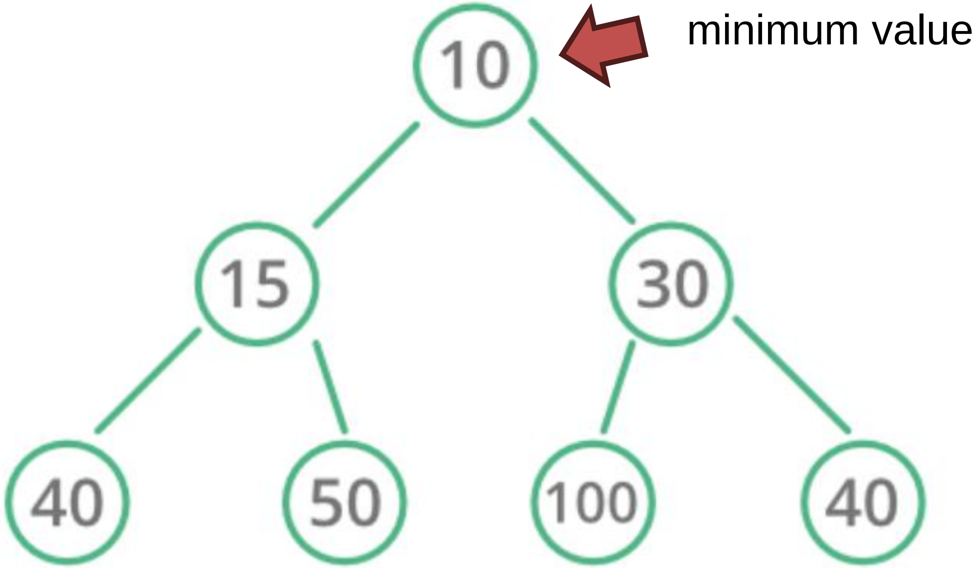
The **Heap** data structure is complete binary tree that follows the **heap property**

Two types of heaps

Max Heap – Parent nodes are greater than both of its children

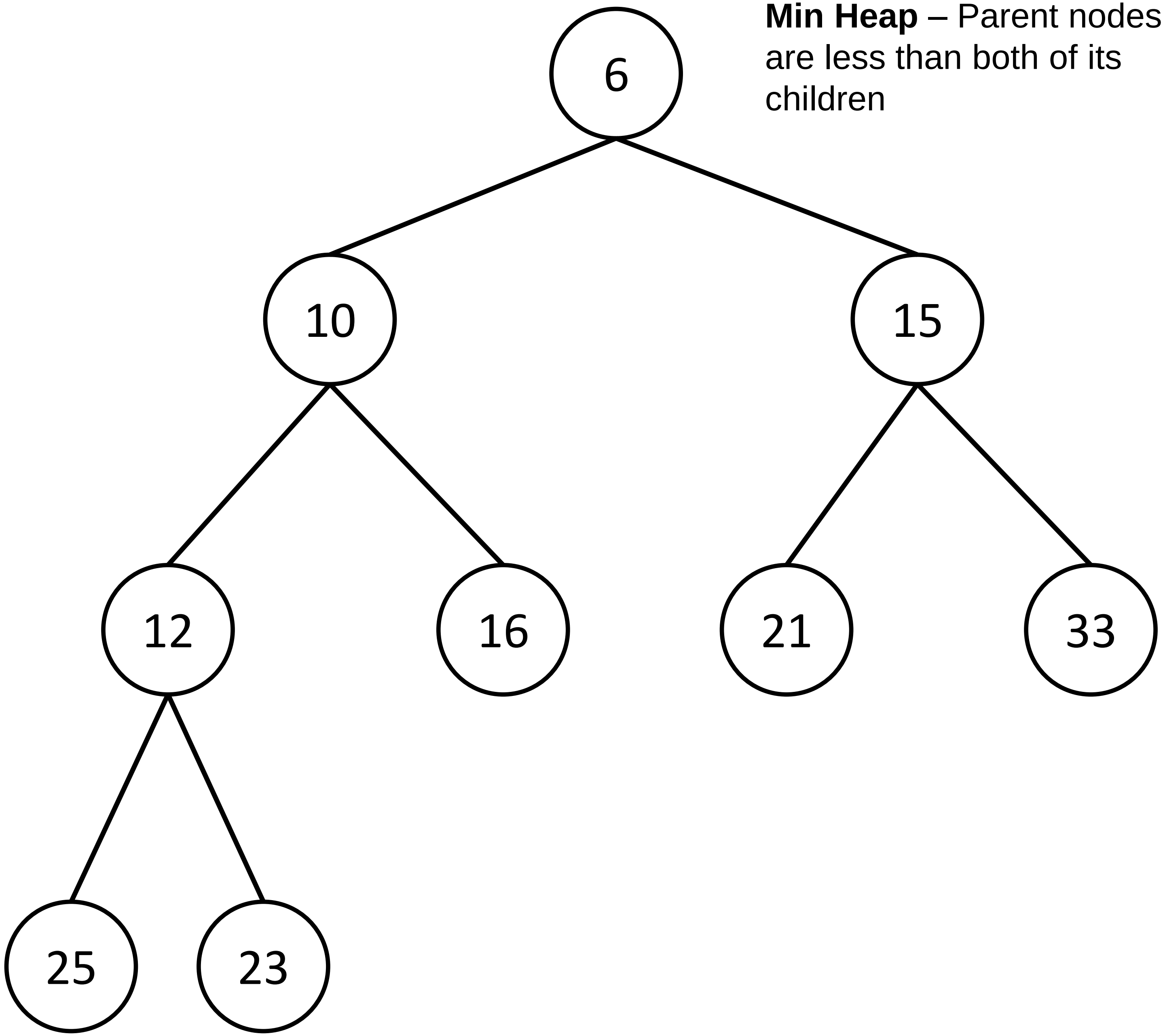


Min Heap – Parent nodes are less than both of its children



Heap Operations - Insert

```
add(7);
```

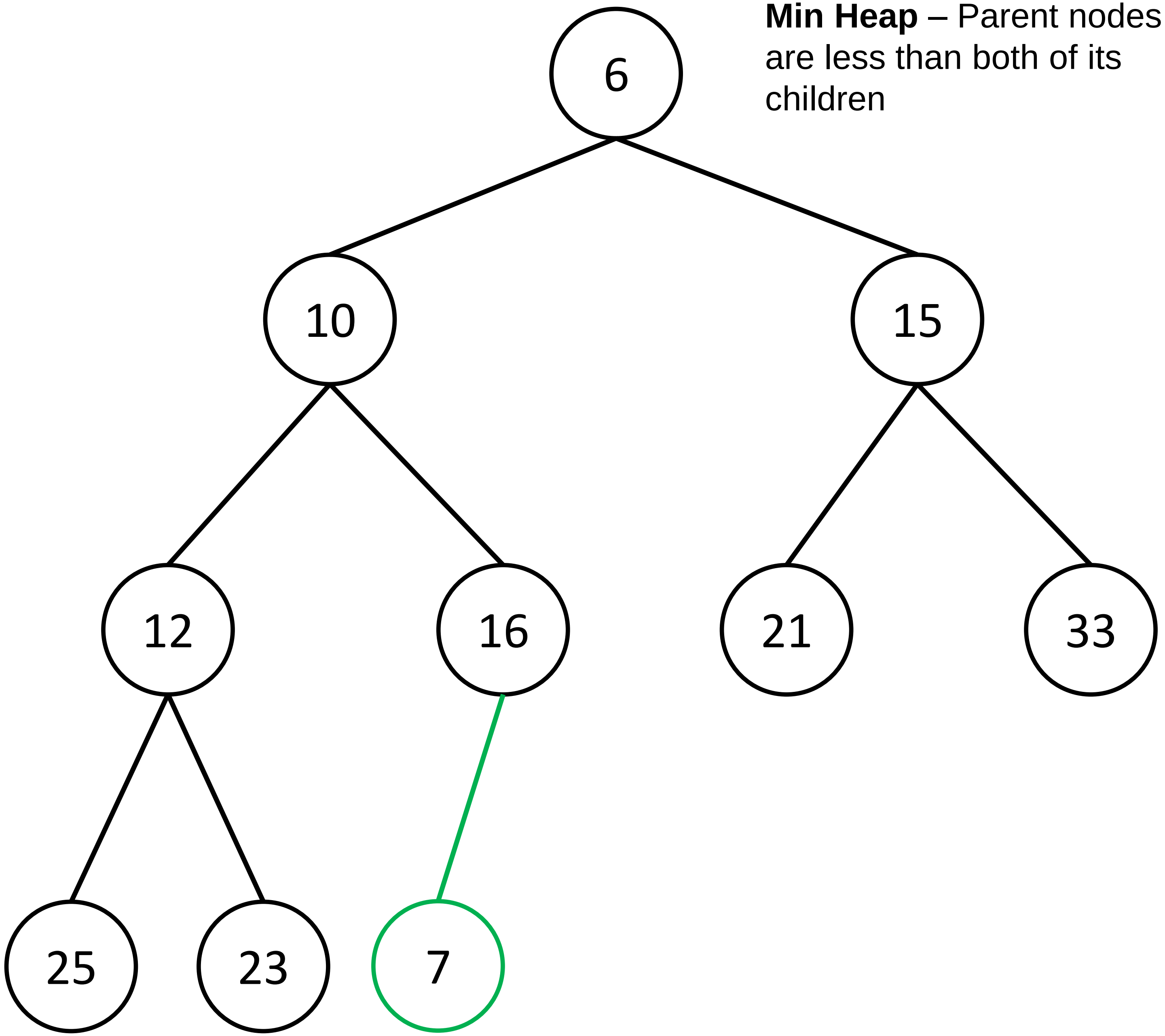


Min Heap – Parent nodes are less than both of its children

Heap Operations - Insert

```
add(7);
```

Because this is a complete binary tree, this is the only place a new node can go



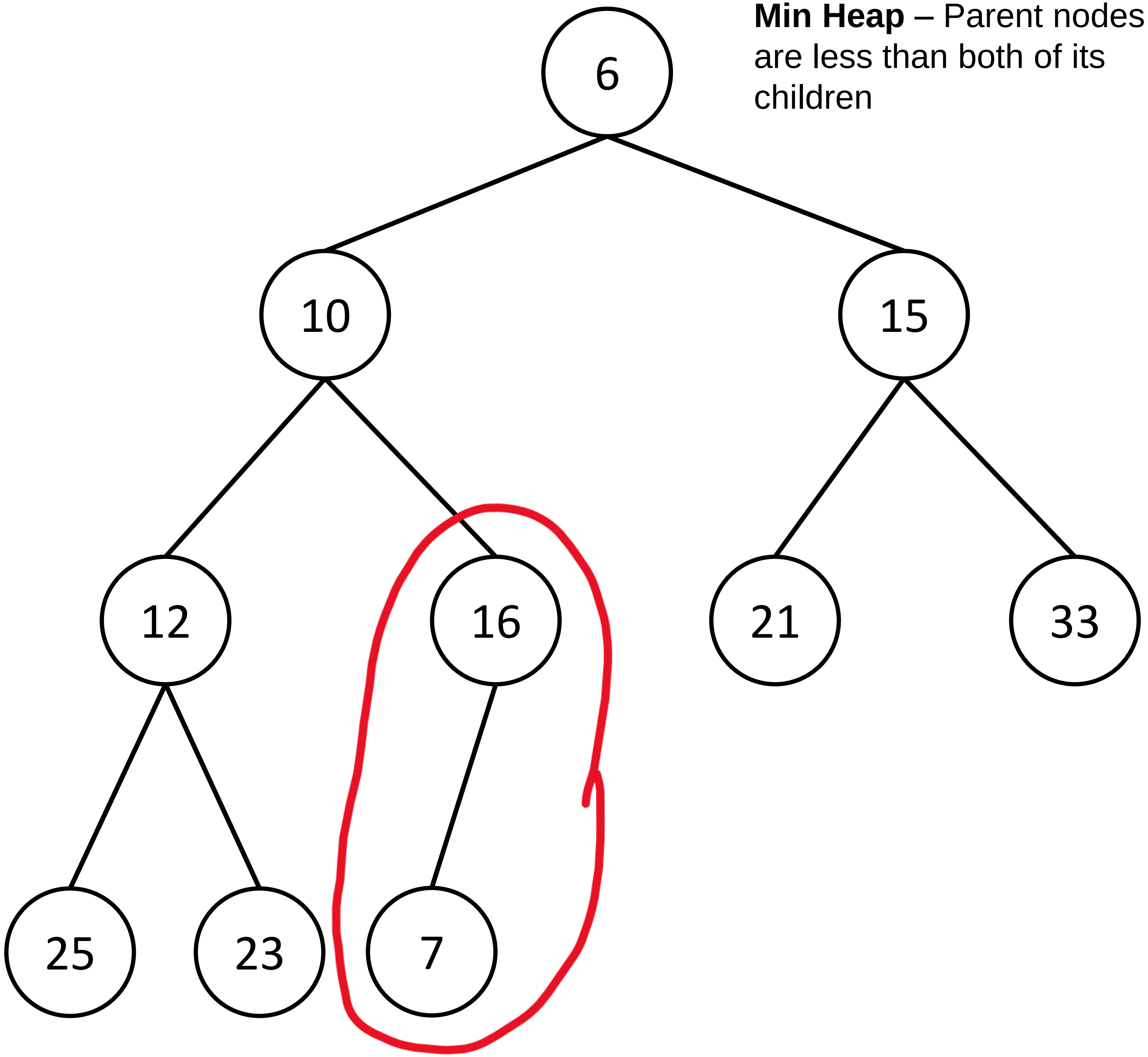
Min Heap – Parent nodes are less than both of its children

Heap Operations - Insert

```
add(7);
```

Because this is a complete binary tree, this is the only place a new node can go

However, we are now violating the heap property



Min Heap – Parent nodes are less than both of its children

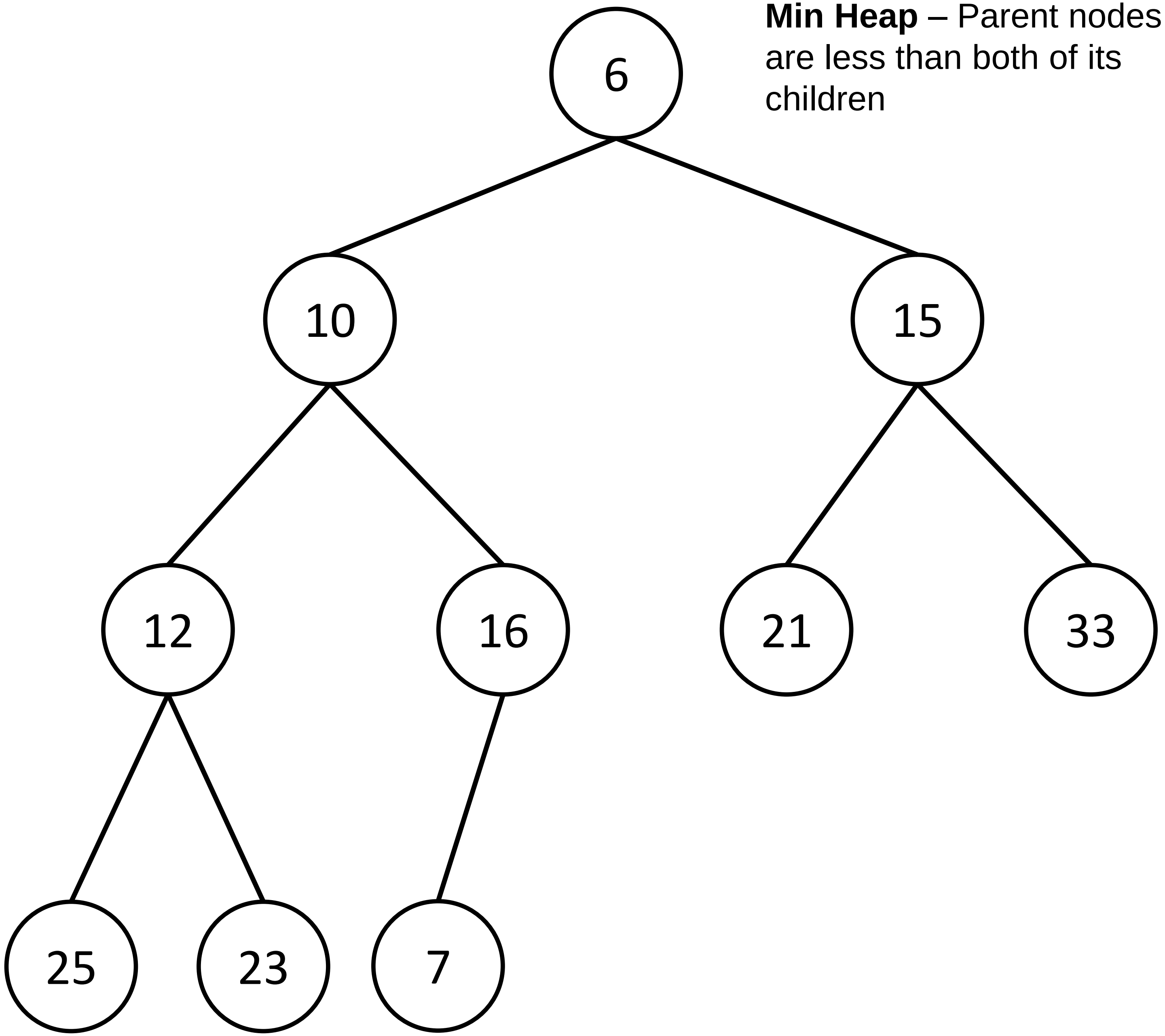
Heap Operations - Insert

```
add(7);
```

Because this is a complete binary tree, this is the only place a new node can go

However, we are now violating the heap property

When new nodes are added, we may need to move it up in the tree



Min Heap – Parent nodes are less than both of its children

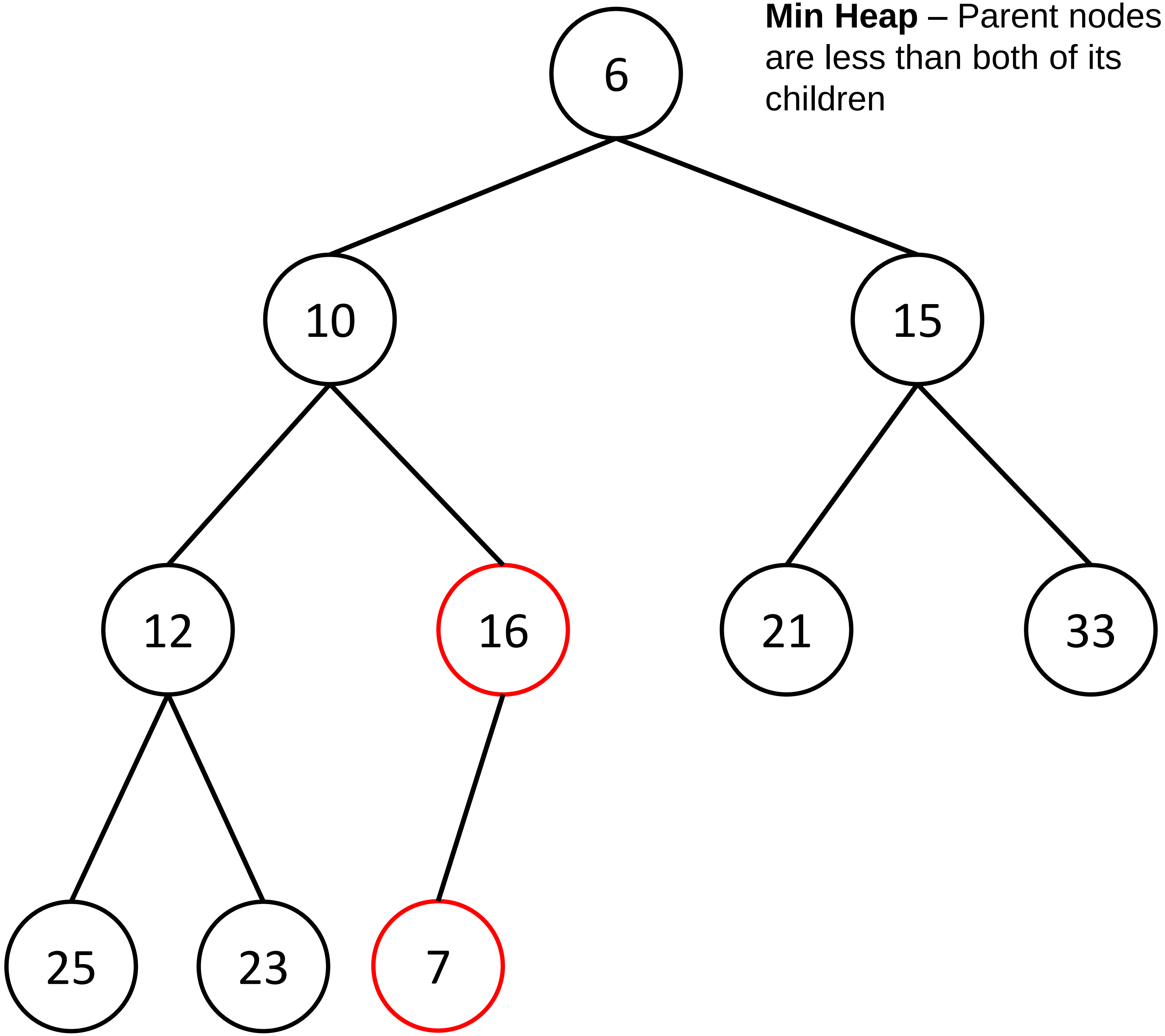
Heap Operations - Insert

```
add(7);
```

Because this is a complete binary tree, this is the only place a new node can go

However, we are now violating the heap property

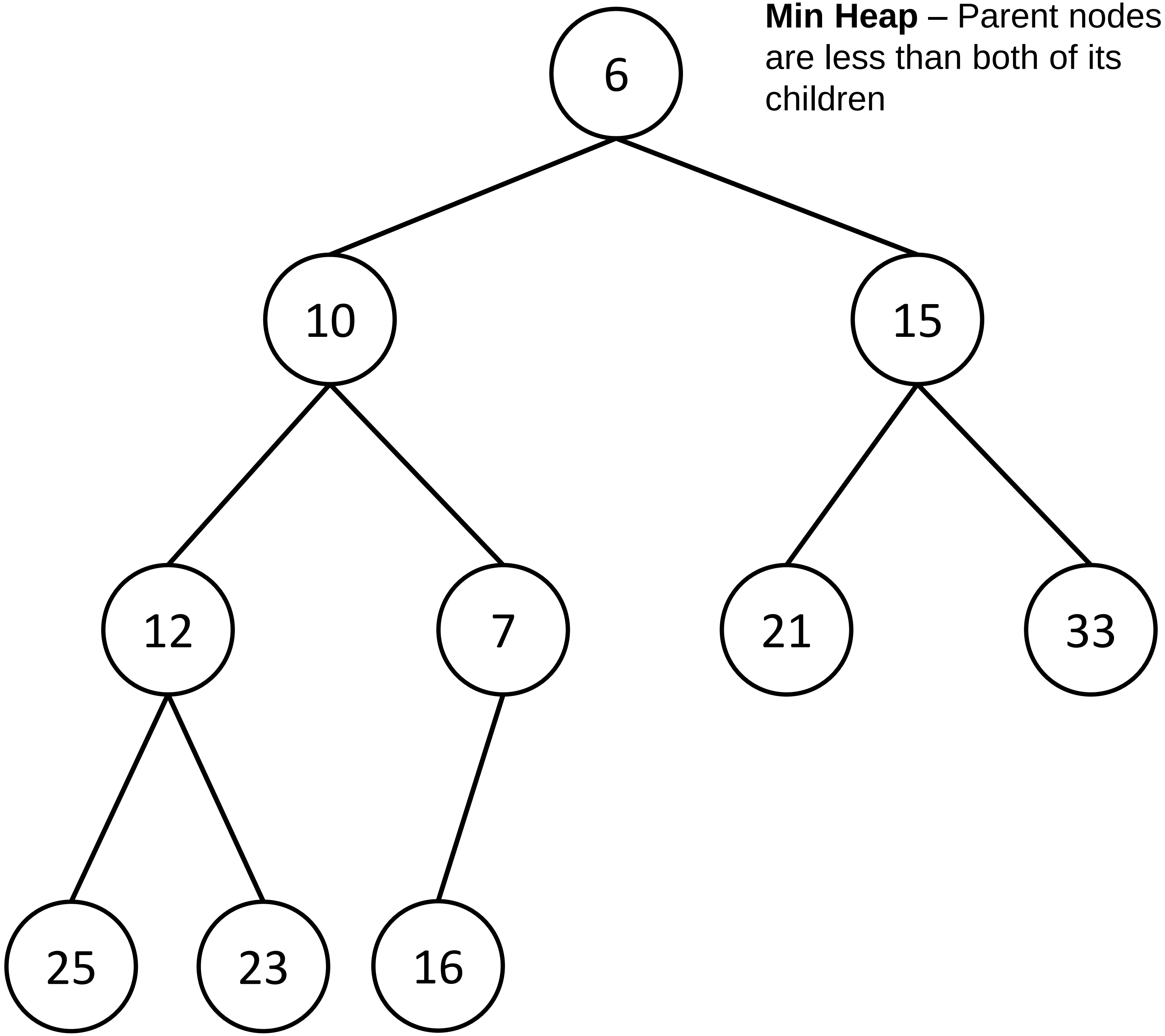
When new nodes are added, we may need to move it up in the tree



Min Heap – Parent nodes are less than both of its children

Heap Operations - Insert

```
add(7);
```



Min Heap – Parent nodes are less than both of its children

Because this is a complete binary tree, this is the only place a new node can go

However, we are now violating the heap property

When new nodes are added, we may need to move it up in the tree

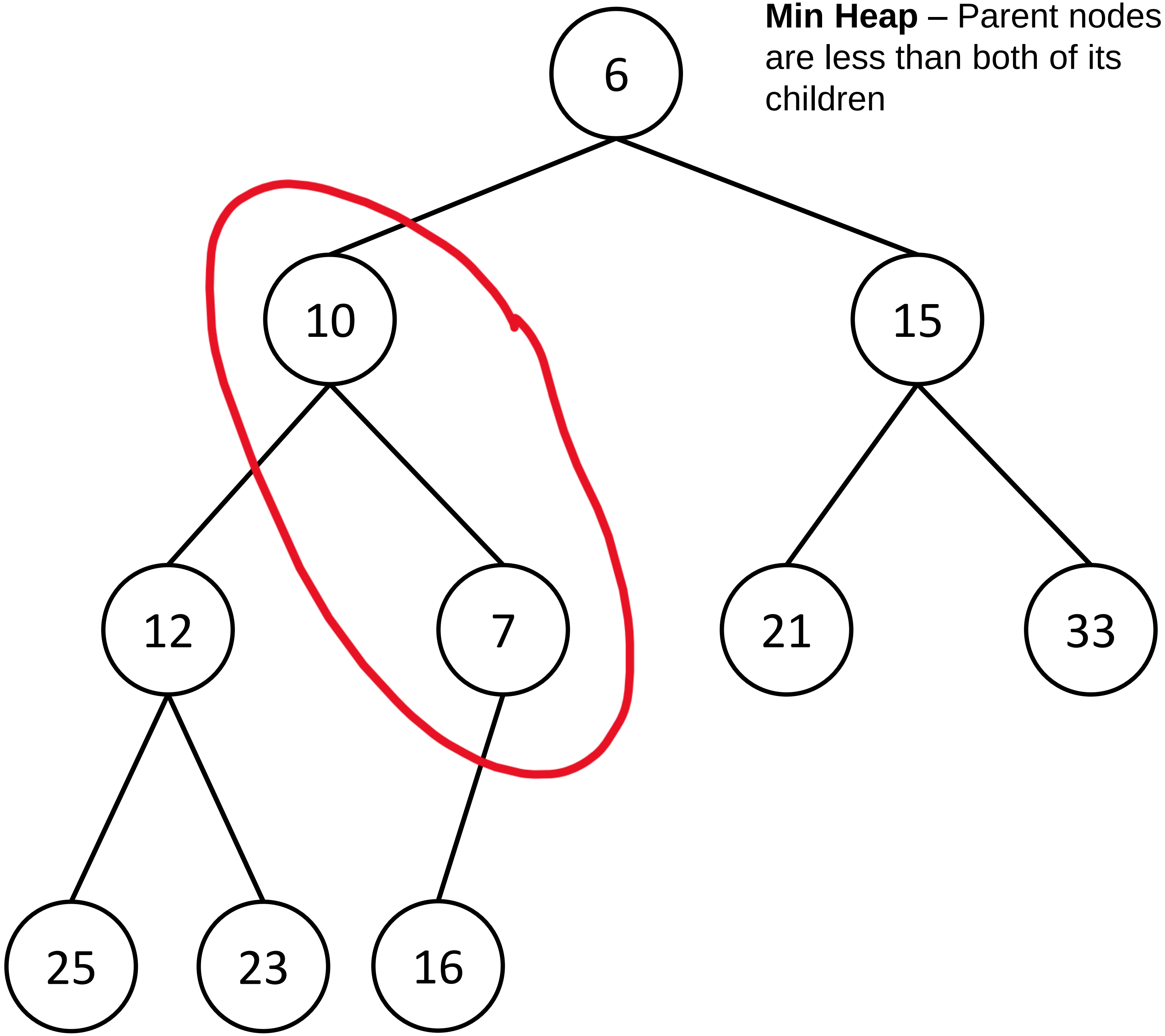
Heap Operations - Insert

```
add(7);
```

Because this is a complete binary tree, this is the only place a new node can go

However, we are now violating the heap property

When new nodes are added, we may need to move it up in the tree



Min Heap – Parent nodes are less than both of its children

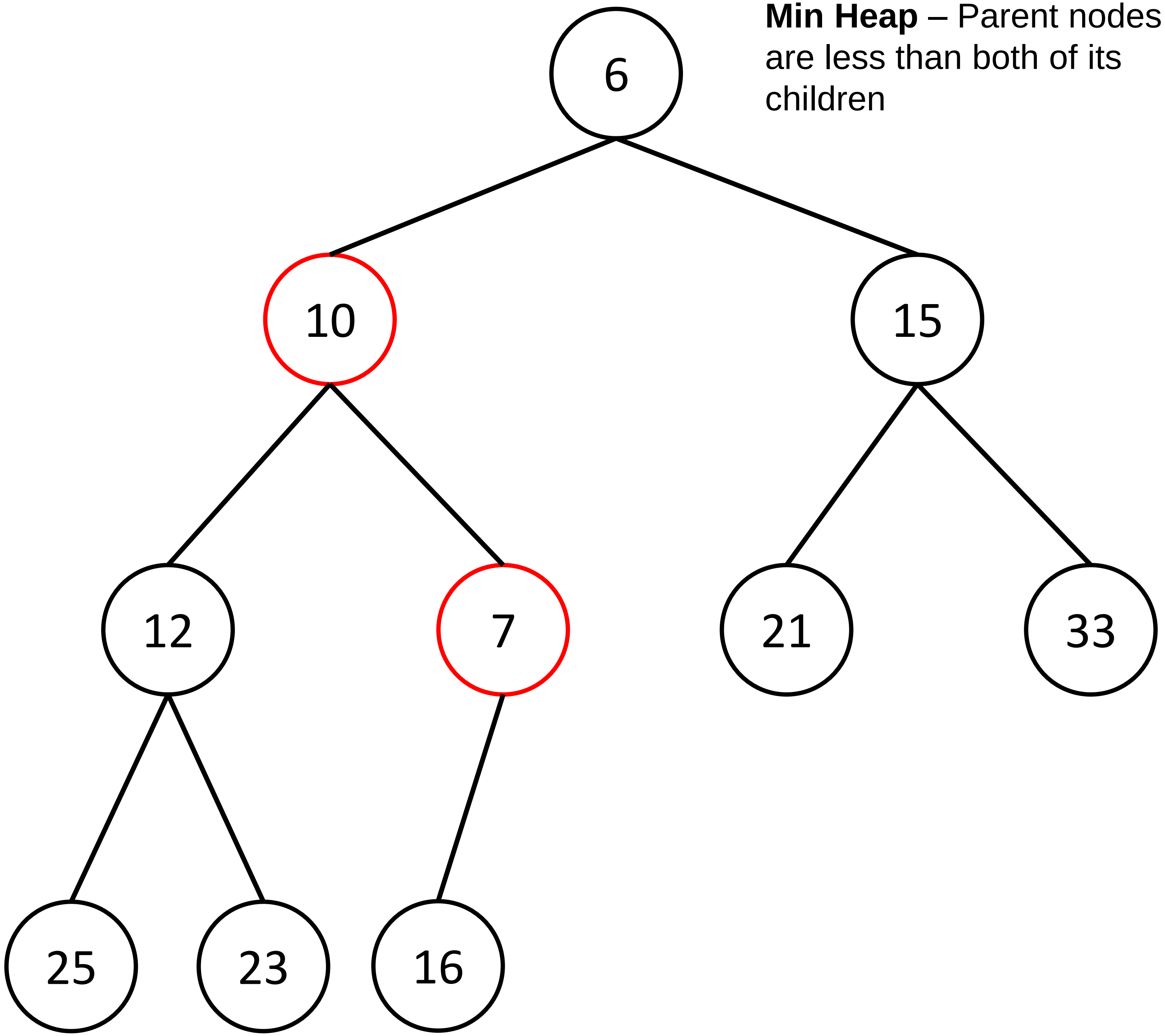
Heap Operations - Insert

```
add(7);
```

Because this is a complete binary tree, this is the only place a new node can go

However, we are now violating the heap property

When new nodes are added, we may need to move it up in the tree



Min Heap – Parent nodes are less than both of its children

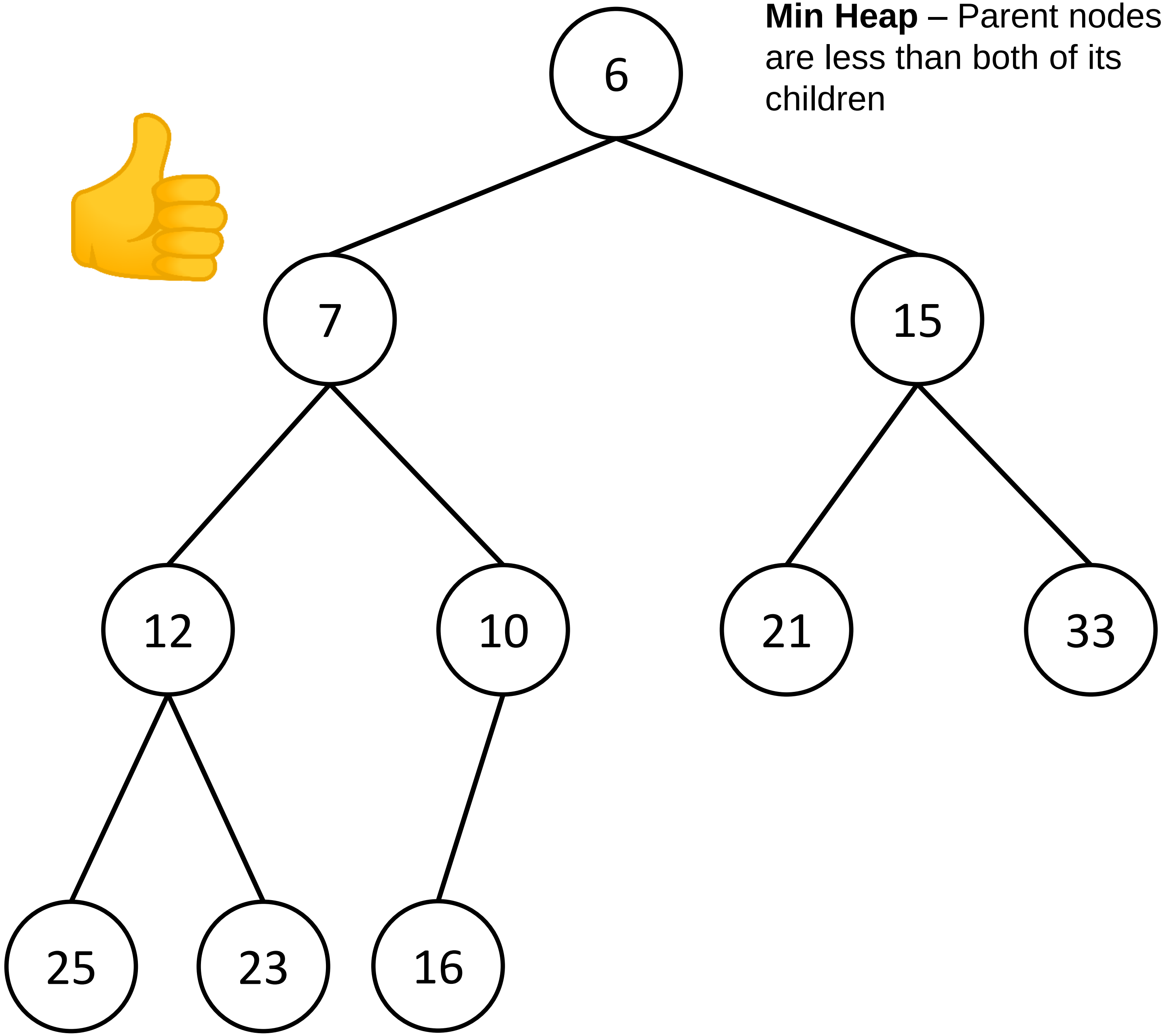
Heap Operations - Insert

```
add(7);
```

Because this is a complete binary tree, this is the only place a new node can go

However, we are now violating the heap property

When new nodes are added, we may need to move it up in the tree

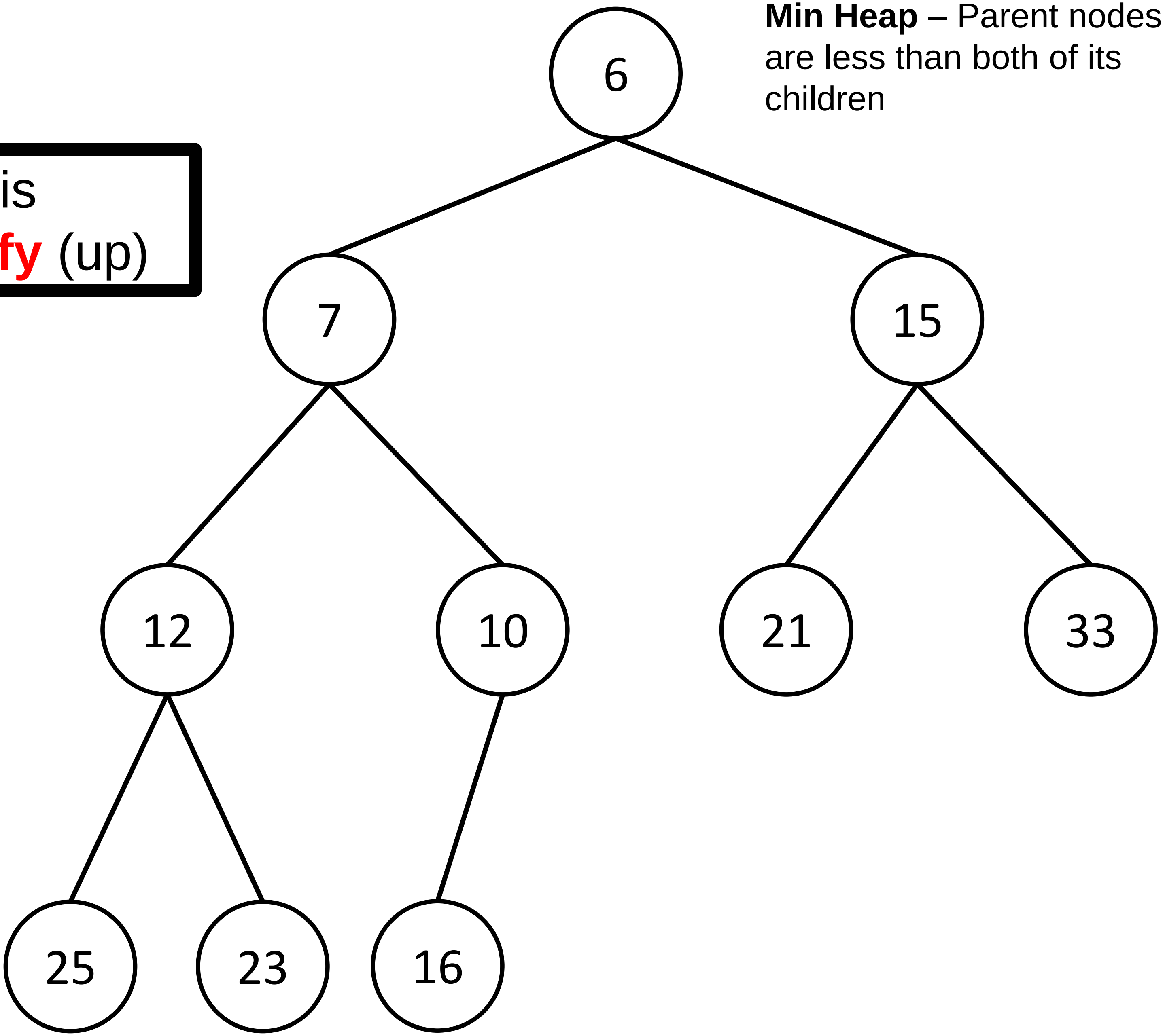


Min Heap – Parent nodes are less than both of its children

Heap Operations - Insert

```
add(7);
```

This process is called **Heapify** (up)



Min Heap – Parent nodes are less than both of its children

Because this is a complete binary tree, this is the only place a new node can go

However, we are now violating the heap property

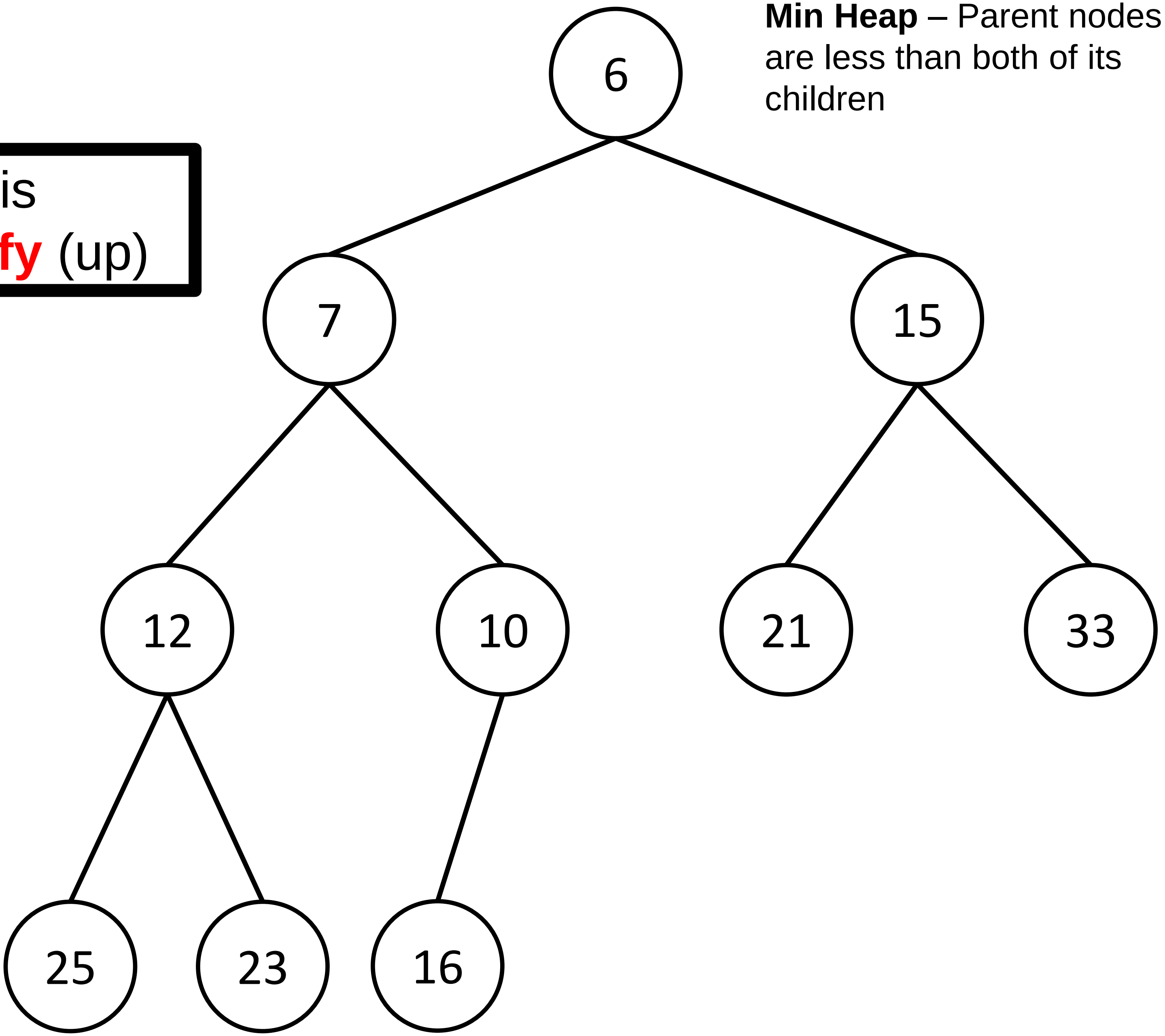
When new nodes are added, we may need to move it up in the tree

Heap Operations - Insert

```
add(7);

add(14);
```

This process is called **Heapify** (up)



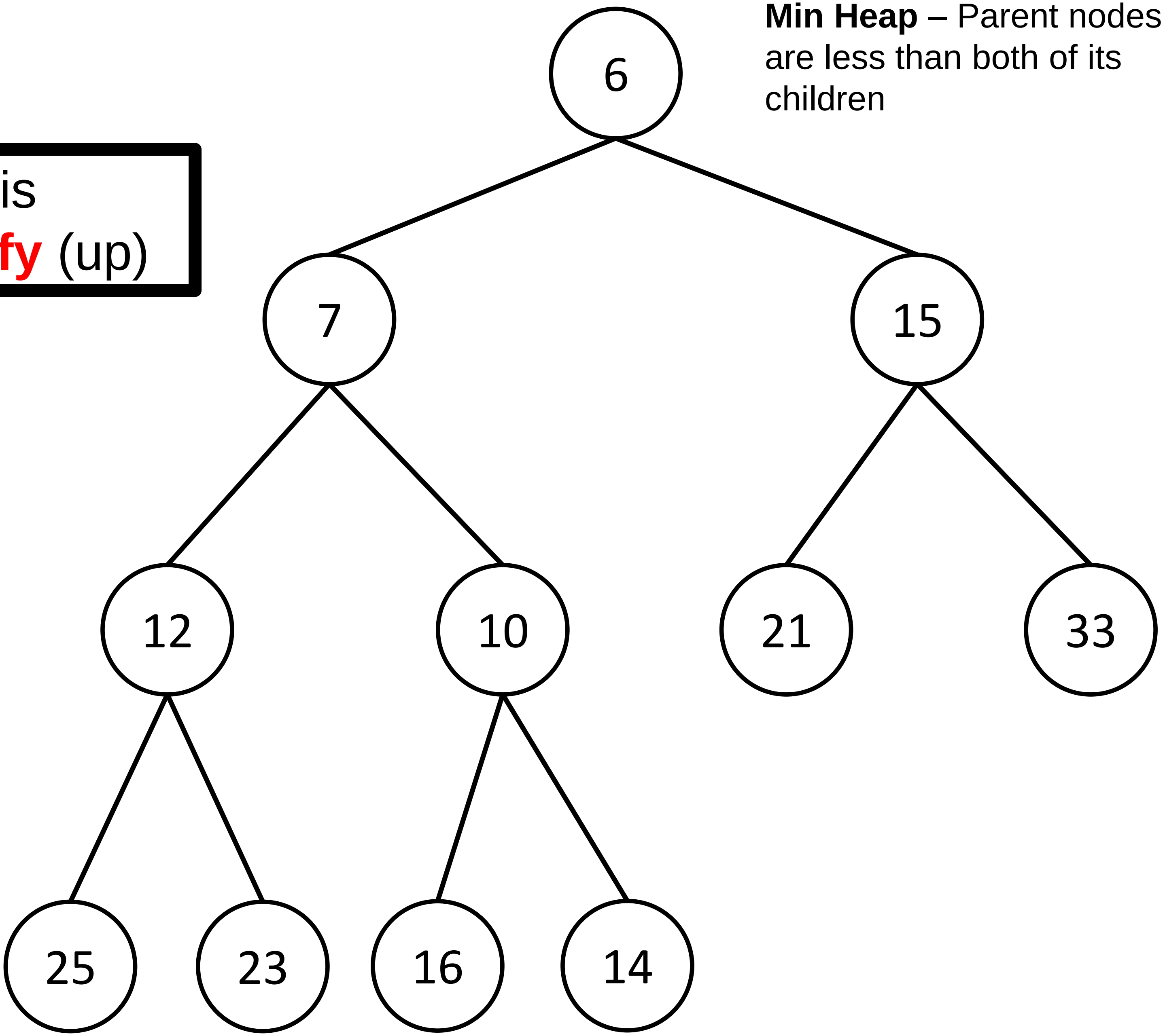
Min Heap – Parent nodes are less than both of its children

Heap Operations - Insert

```
add(7);

add(14);
```

This process is called **Heapify** (up)



Min Heap – Parent nodes are less than both of its children

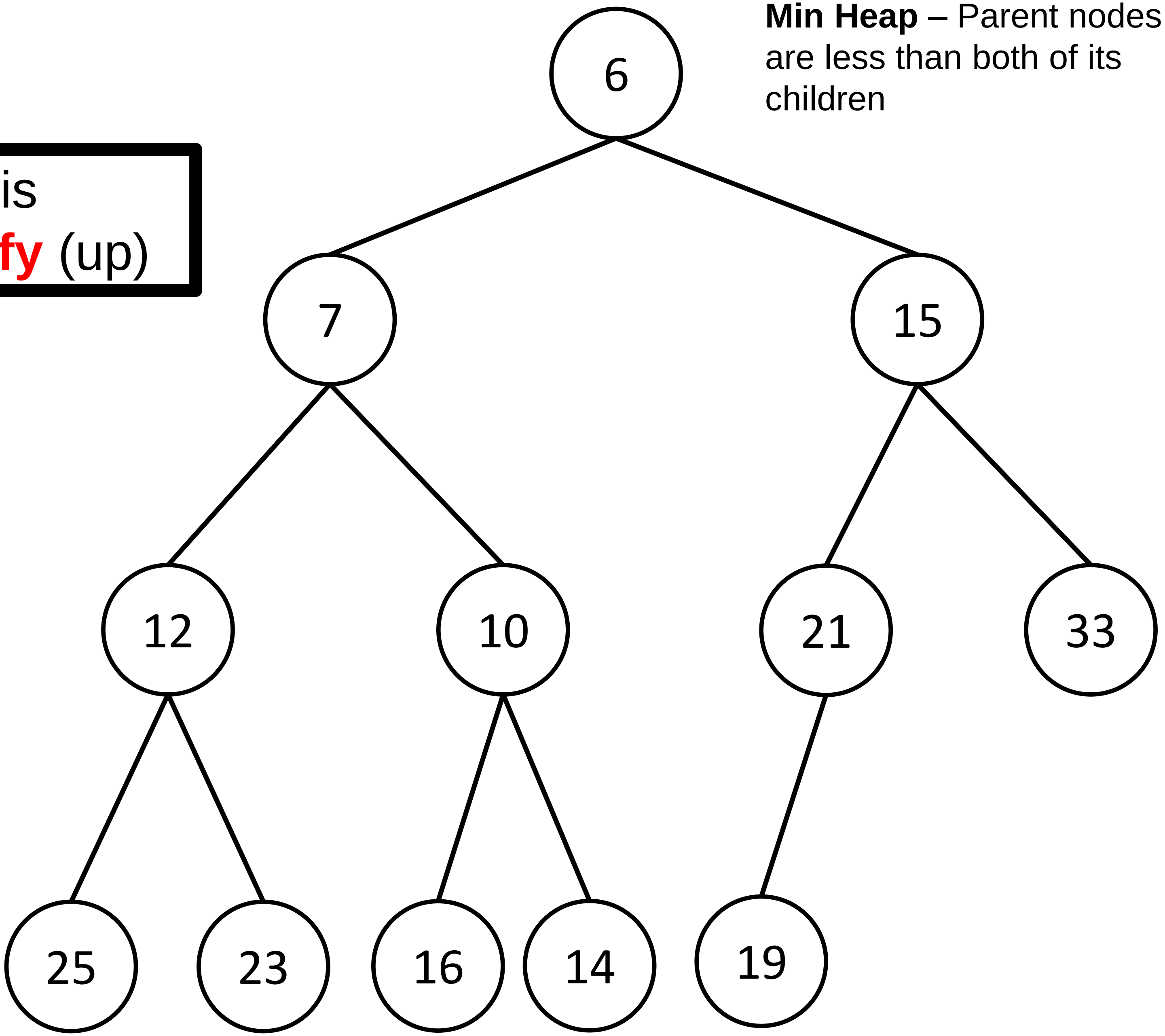
Heap Operations - Insert

```
add(7);

add(14);

add(19);
```

This process is called **Heapify** (up)



Min Heap – Parent nodes are less than both of its children

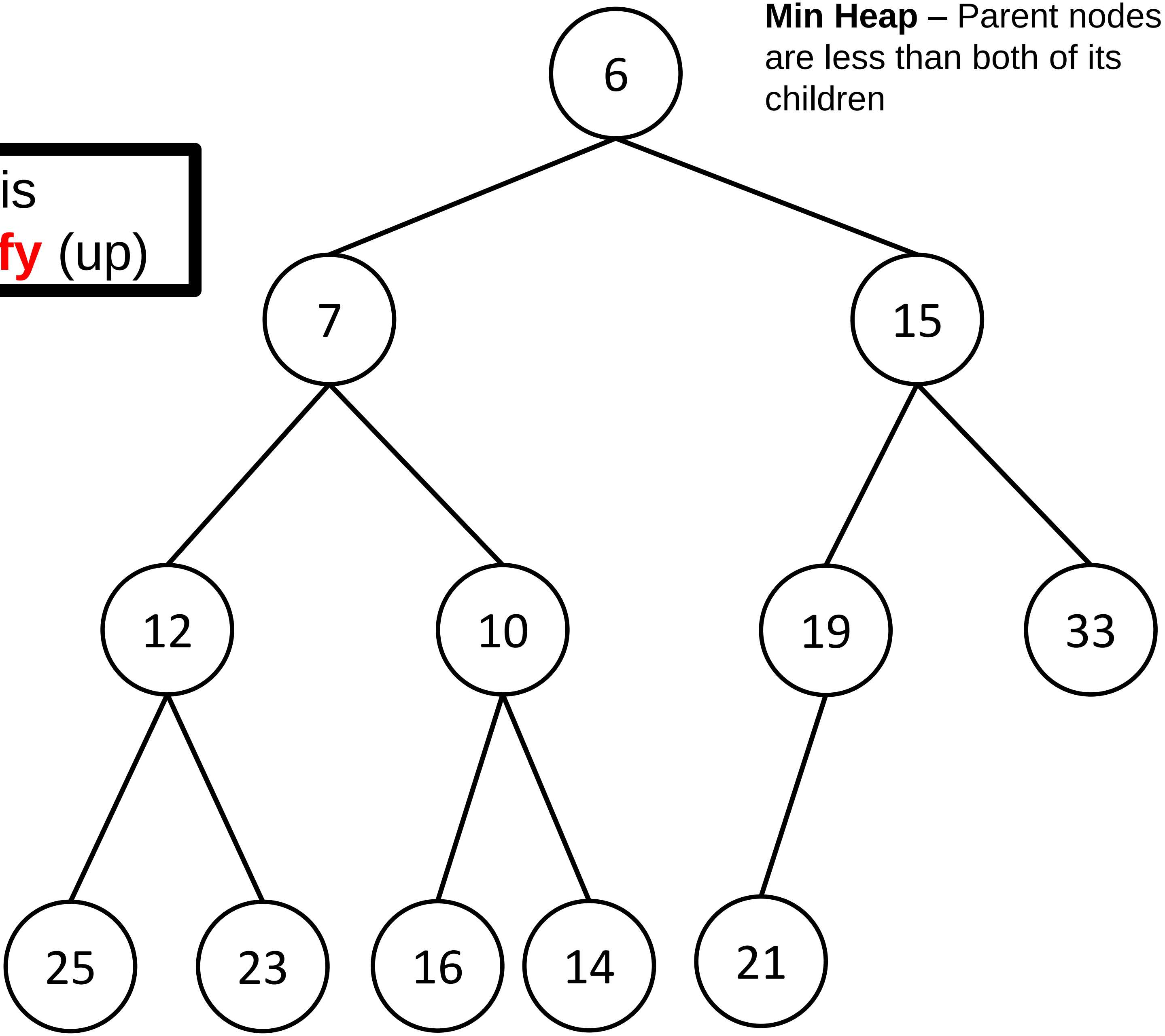
Heap Operations - Insert

```
add(7);

add(14);

add(19);
```

This process is called **Heapify** (up)



Min Heap – Parent nodes are less than both of its children

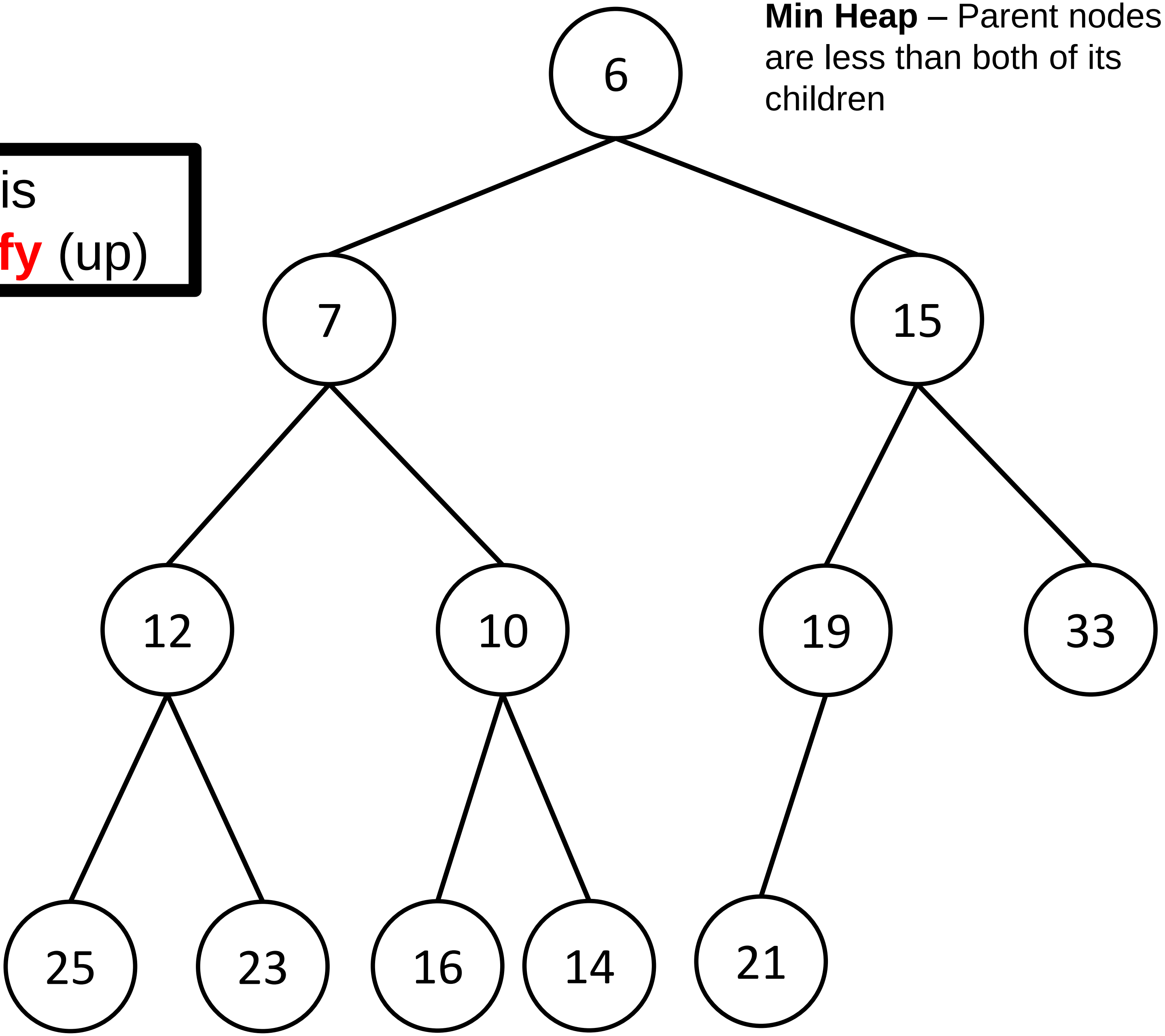
Heap Operations - Insert

```
add(7);
```

This process is called **Heapify** (up)

```
add(14);
```

```
add(19);
```



Min Heap – Parent nodes are less than both of its children

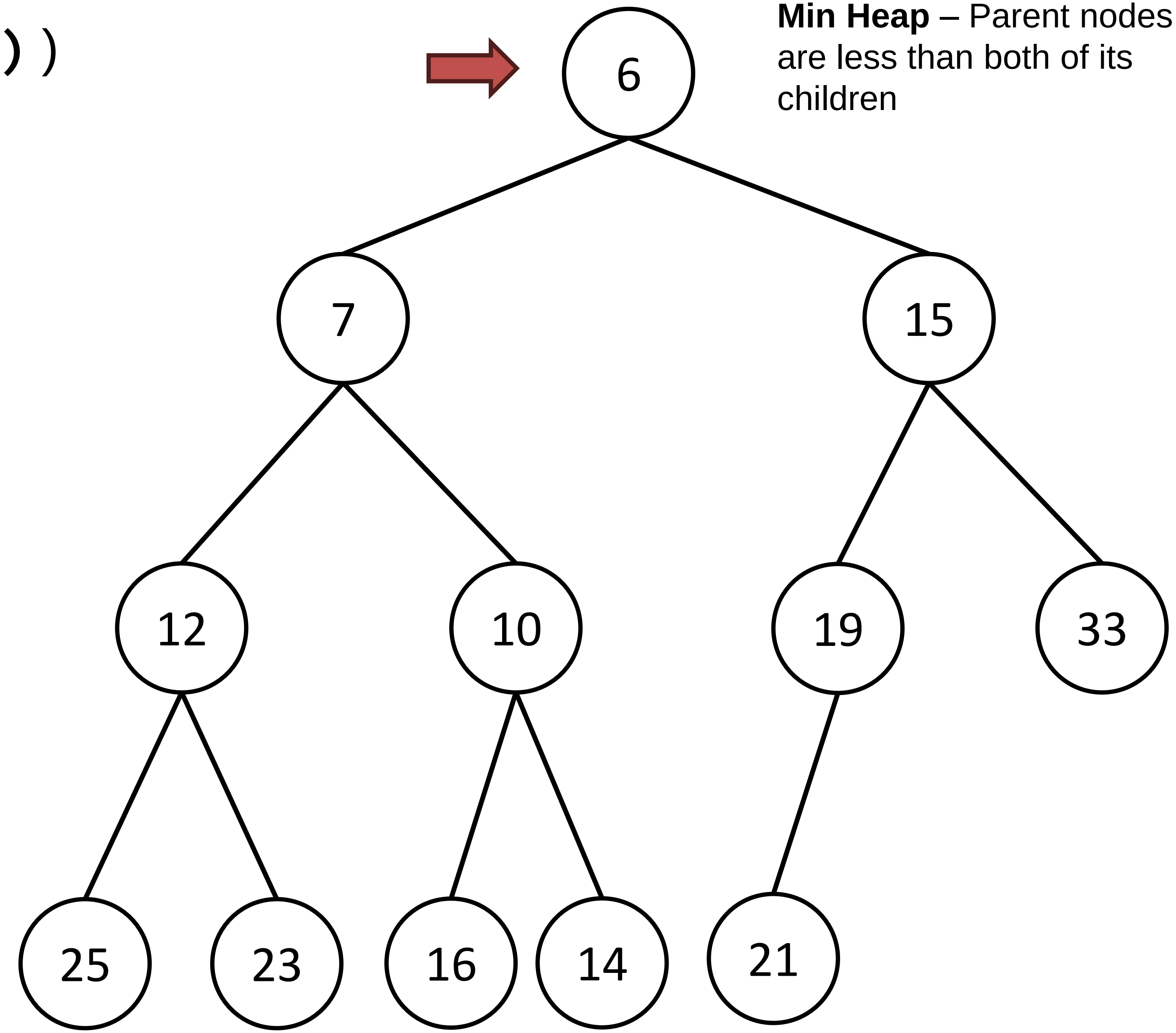
Running time?

- Finding where to place new node: **O(1)** (this will make sense later)
- Insertion – **O(1)**
- Heapify Up – **O(logn)**

Total Running Time: **O(logn)**

Heap Operations – Removal (`poll()`)

When using a Heap, we only remove the root node, which will be either the maximum value or minimum value

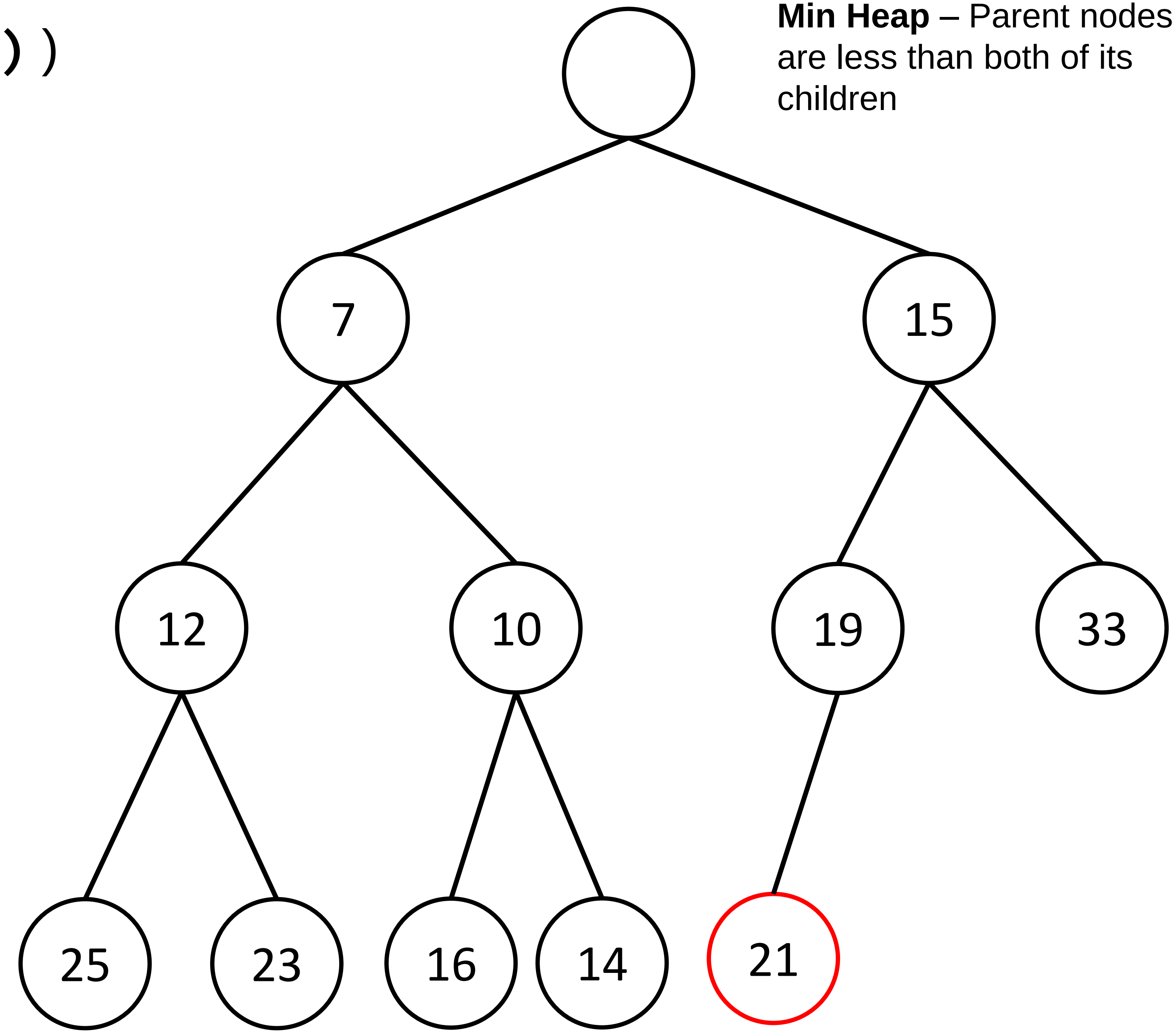


Min Heap – Parent nodes are less than both of its children

Heap Operations – Removal (**`poll()`**)

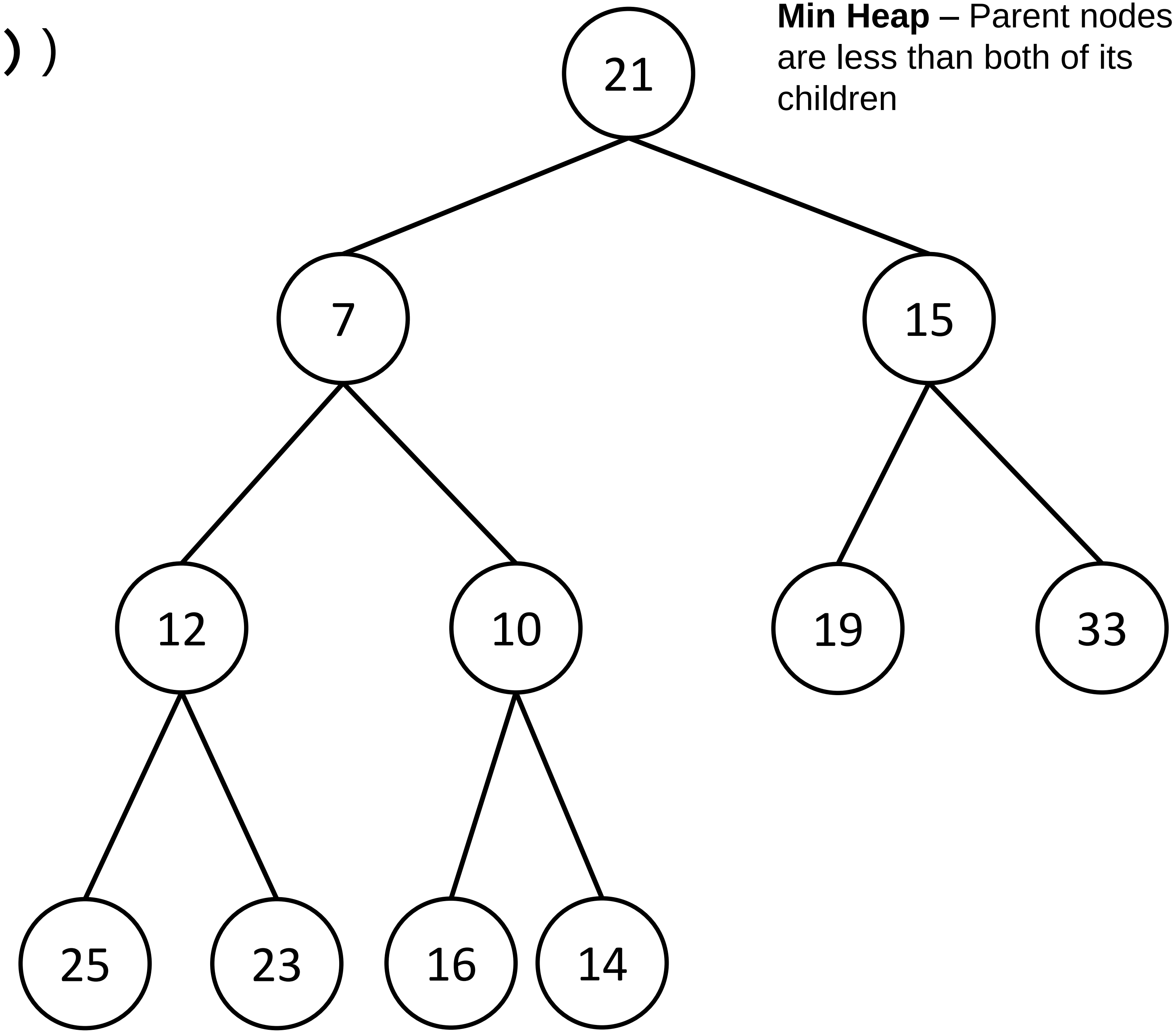
When using a Heap, we only remove the root node, which will be either the maximum value or minimum value

When the root is removed, we replace it with **the last node that was added to the heap**



Min Heap – Parent nodes are less than both of its children

Heap Operations – Removal (`poll()`)



Min Heap – Parent nodes are less than both of its children

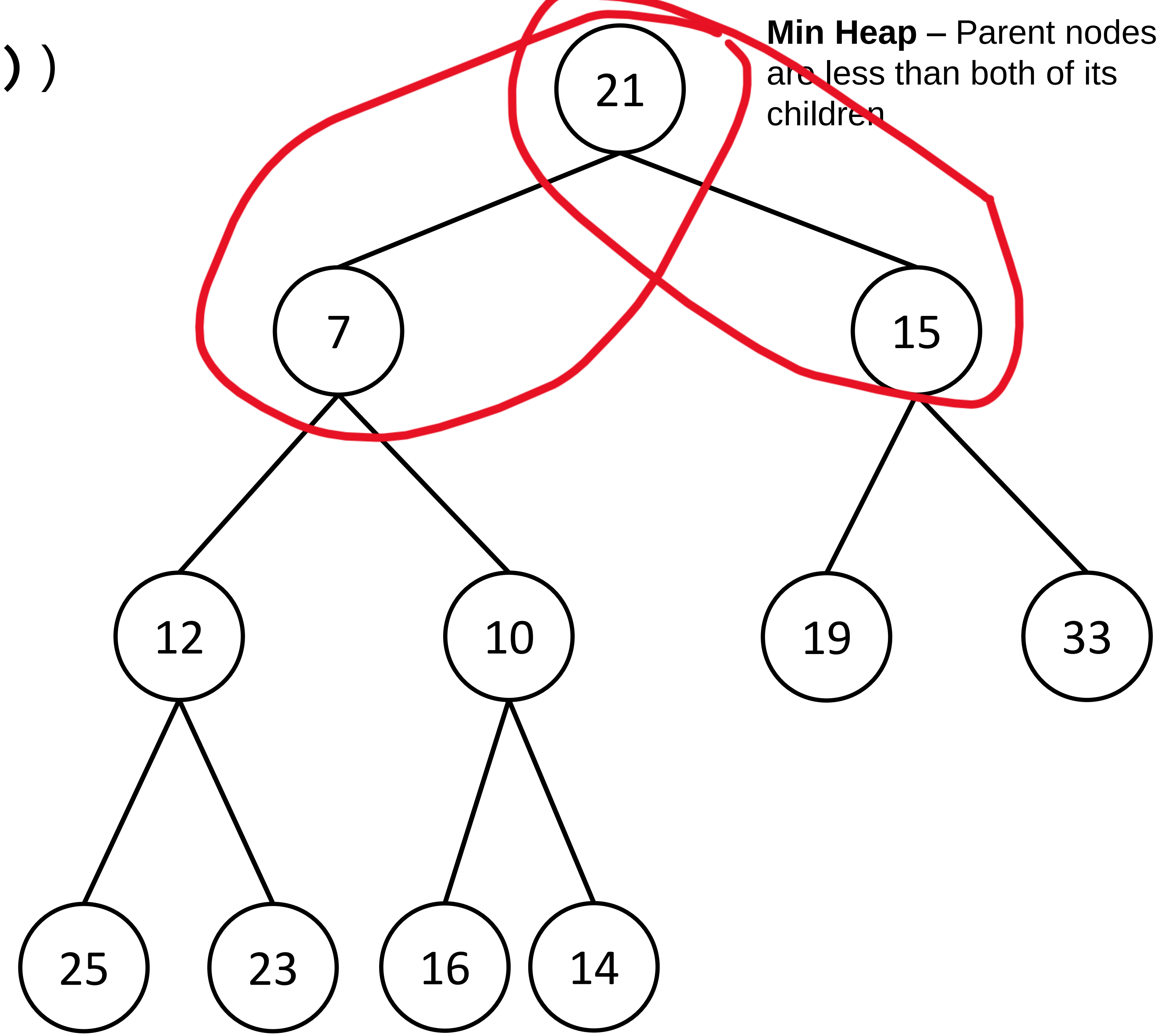
When using a Heap, we only remove the root node, which will be either the maximum value or minimum value

When the root is removed, we replace it with **the last node that was added to the heap**

Heap Operations – Removal (**`poll()`**)

When using a Heap, we only remove the root node, which will be either the maximum value or minimum value

When the root is removed, we replace it with **the last node that was added to the heap**



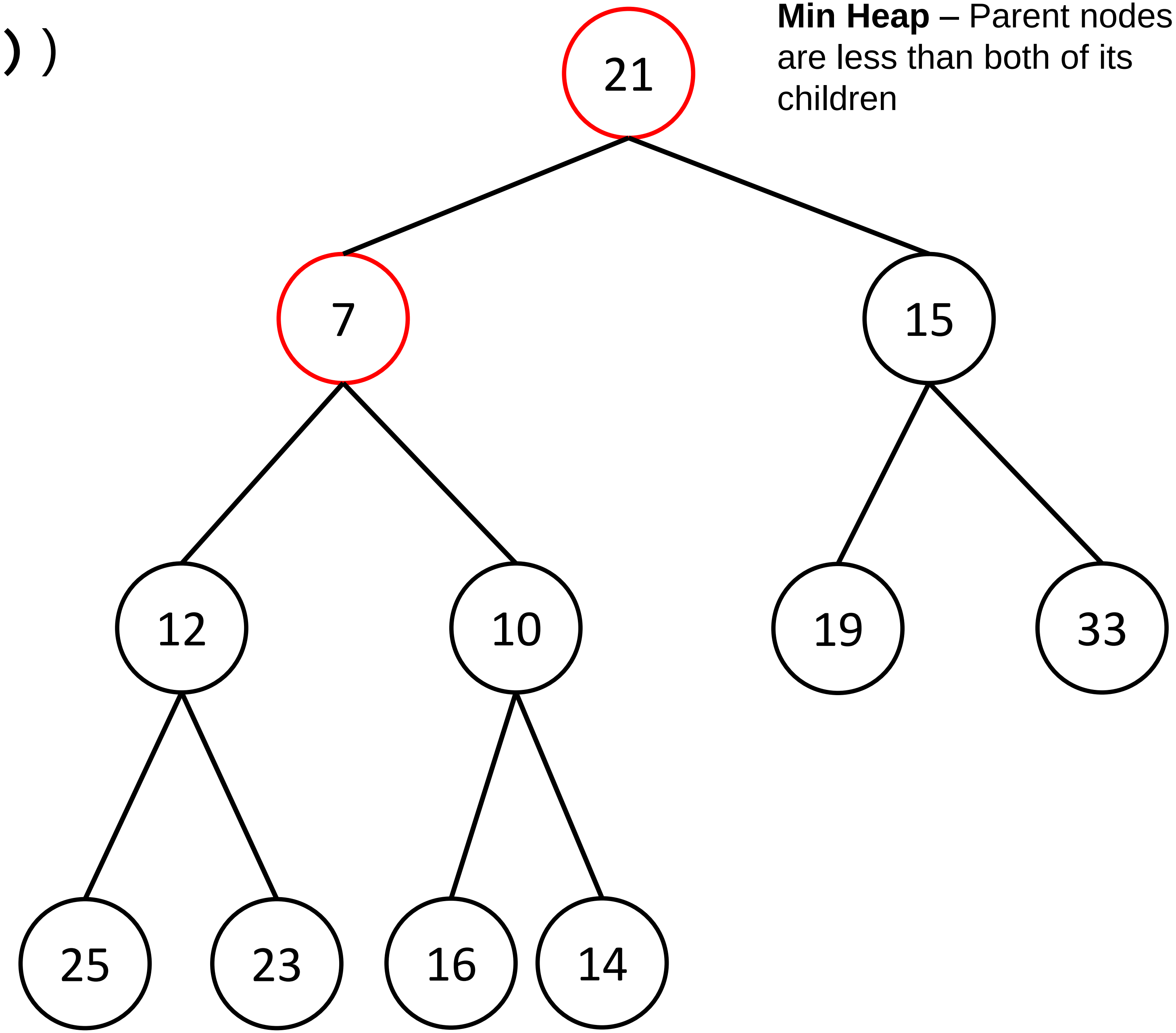
Min Heap – Parent nodes are less than both of its children

Heap Operations – Removal (**`poll()`**)

When using a Heap, we only remove the root node, which will be either the maximum value or minimum value

When the root is removed, we replace it with **the last node that was added to the heap**

When the root is replaced, it may need to be moved down in the tree



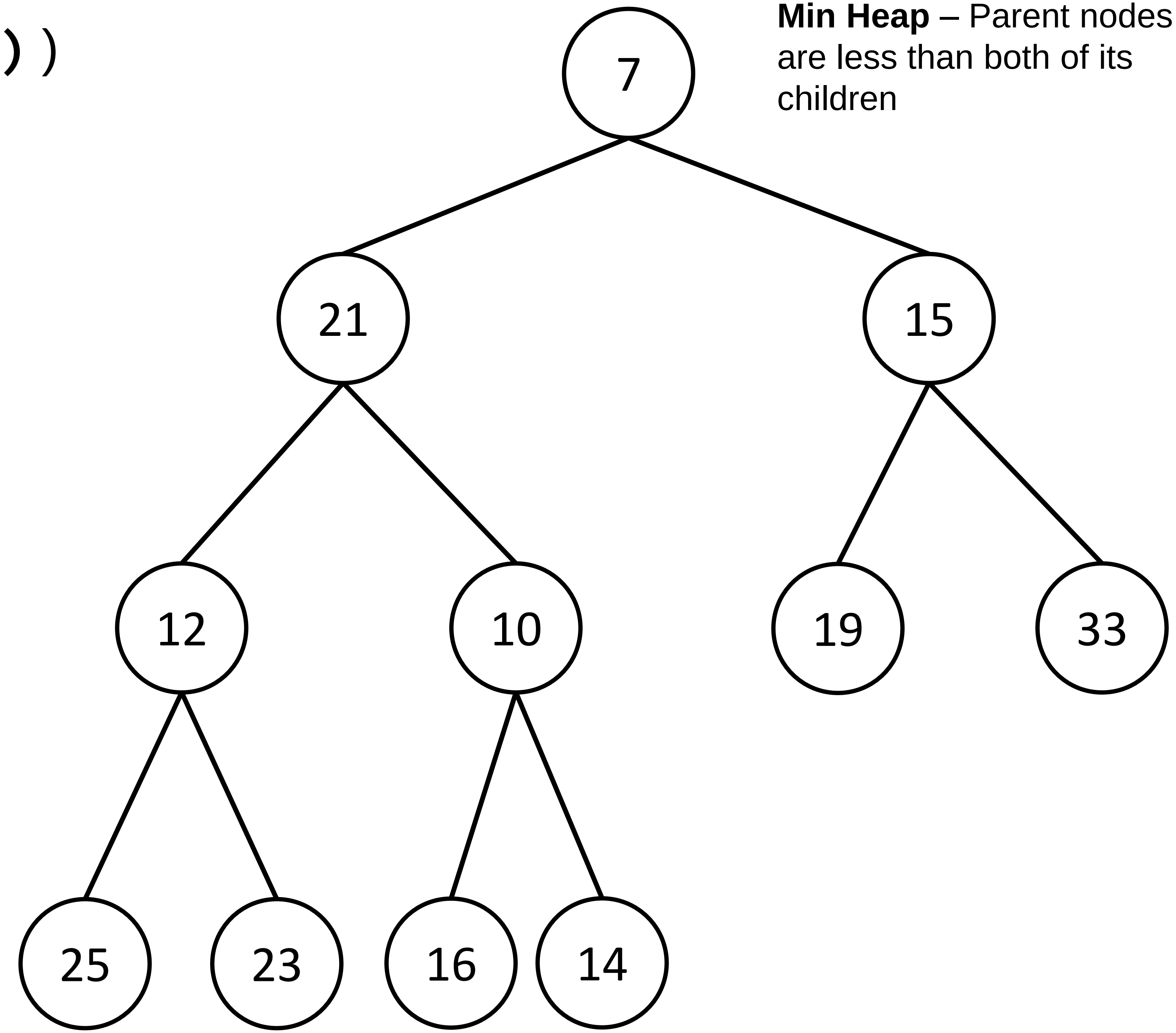
Min Heap – Parent nodes are less than both of its children

Heap Operations – Removal (**`poll()`**)

When using a Heap, we only remove the root node, which will be either the maximum value or minimum value

When the root is removed, we replace it with **the last node that was added to the heap**

When the root is replaced, it may need to be moved down in the tree



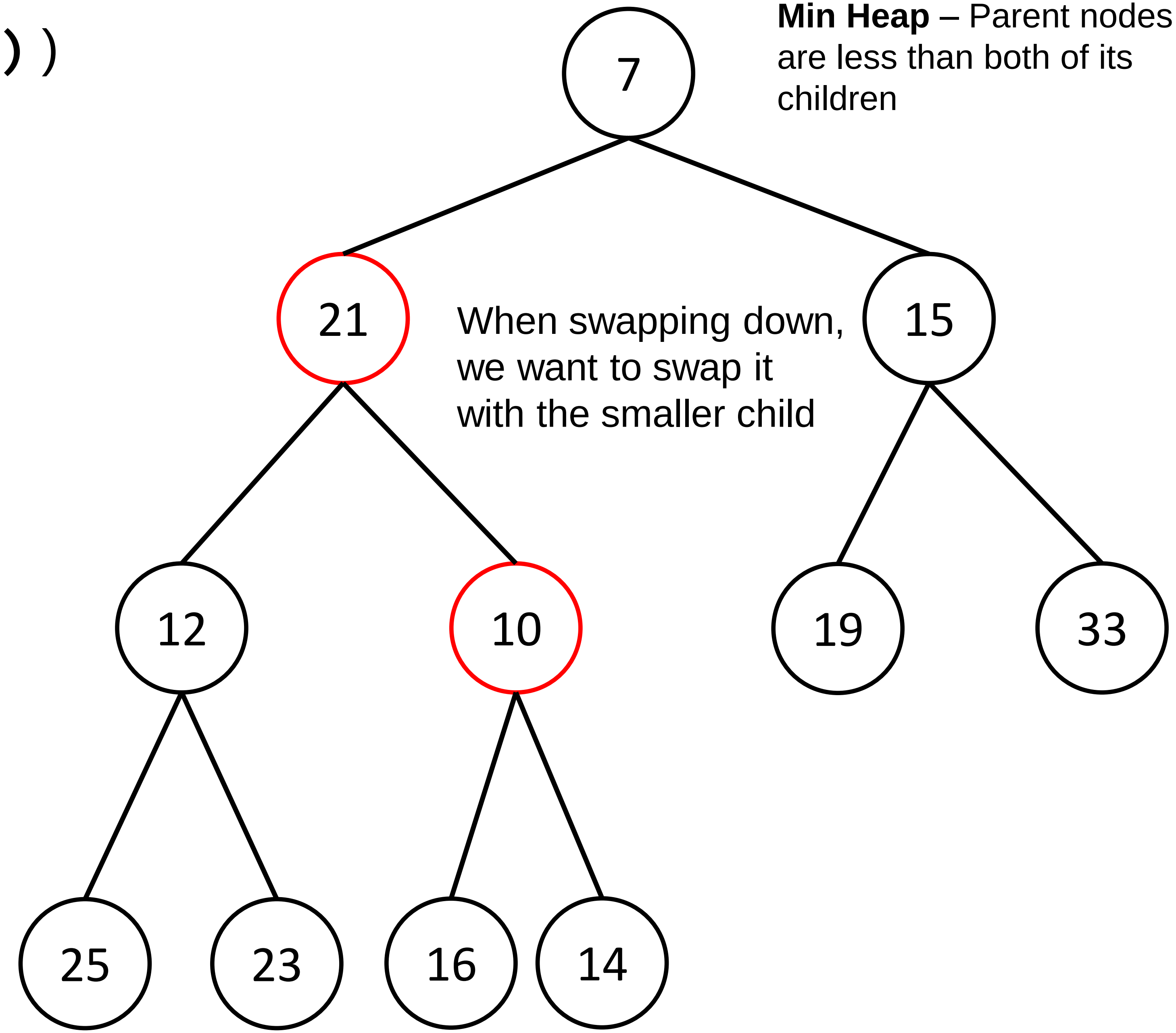
Min Heap – Parent nodes are less than both of its children

Heap Operations – Removal (**`poll()`**)

When using a Heap, we only remove the root node, which will be either the maximum value or minimum value

When the root is removed, we replace it with **the last node that was added to the heap**

When the root is replaced, it may need to be moved down in the tree



Min Heap – Parent nodes are less than both of its children

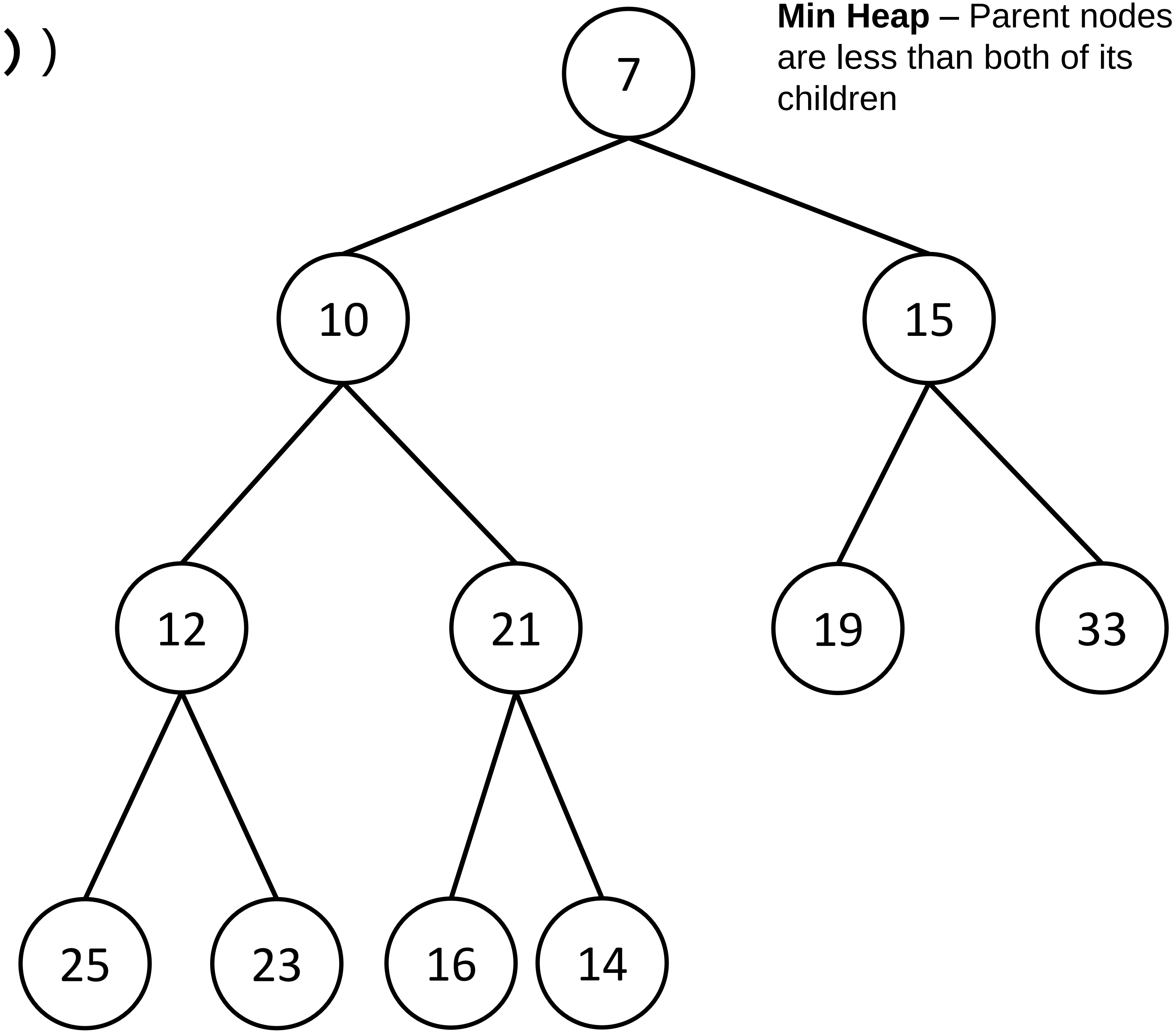
When swapping down, we want to swap it with the smaller child

Heap Operations – Removal (`poll()`)

When using a Heap, we only remove the root node, which will be either the maximum value or minimum value

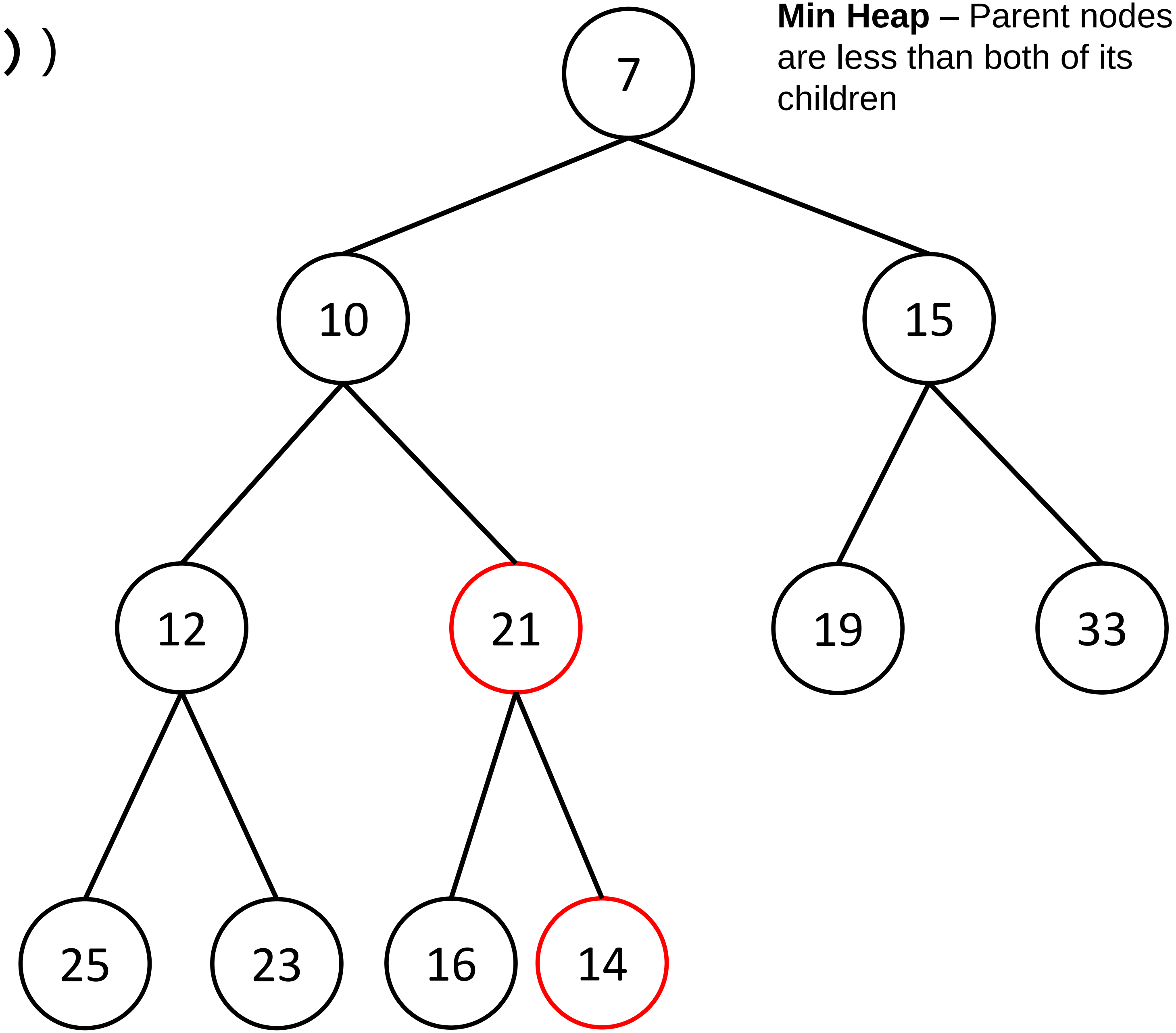
When the root is removed, we replace it with **the last node that was added to the heap**

When the root is replaced, it may need to be moved down in the tree



Min Heap – Parent nodes are less than both of its children

Heap Operations – Removal (`poll()`)



Min Heap – Parent nodes are less than both of its children

When using a Heap, we only remove the root node, which will be either the maximum value or minimum value

When the root is removed, we replace it with **the last node that was added to the heap**

When the root is replaced, it may need to be moved down in the tree

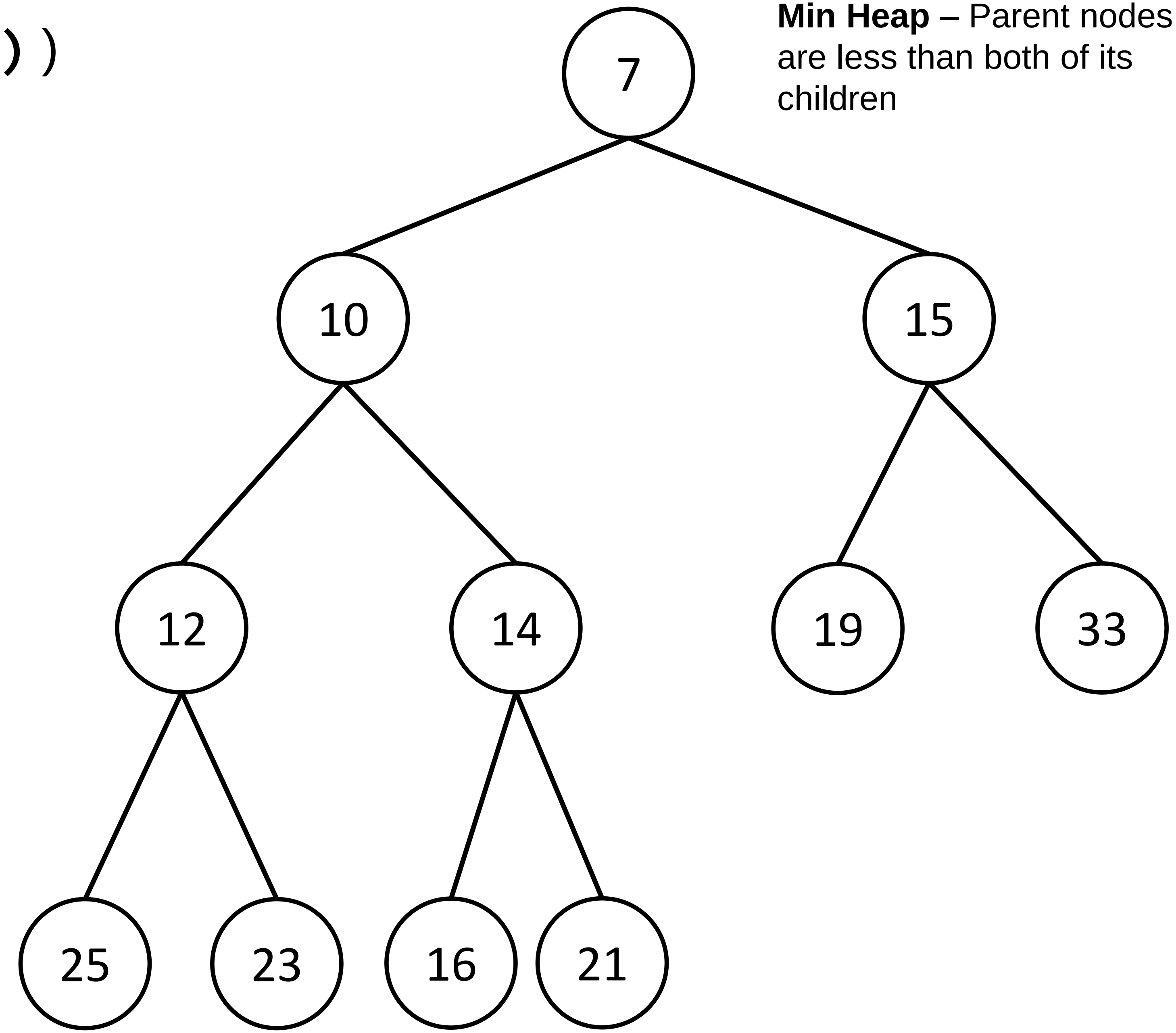
Heap Operations – Removal (**`poll()`**)

When using a Heap, we only remove the root node, which will be either the maximum value or minimum value

When the root is removed, we replace it with **the last node that was added to the heap**

When the root is replaced, it may need to be moved down in the tree

Min Heap – Parent nodes are less than both of its children



Heap Operations – Removal (`poll()`)

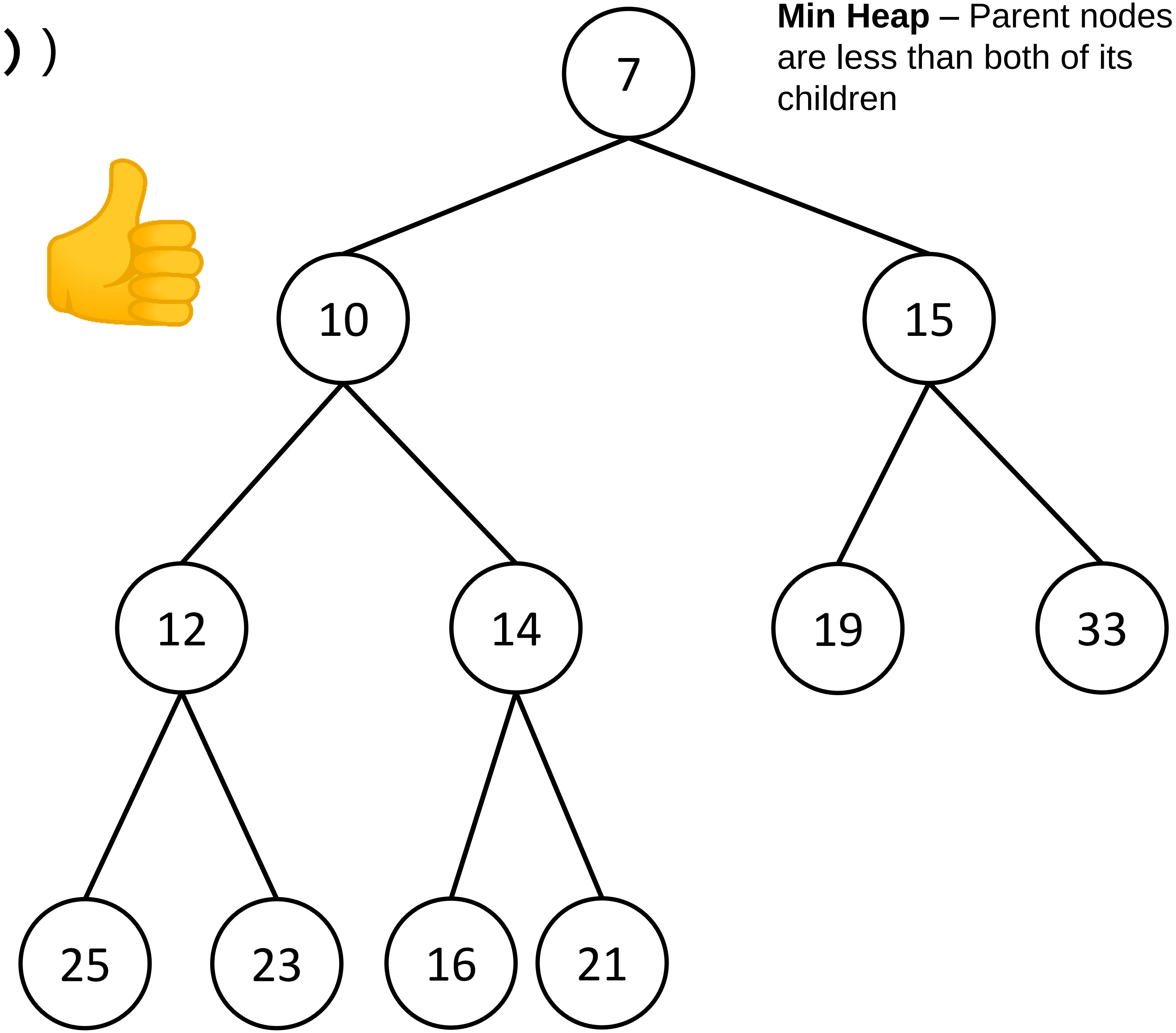
Min Heap – Parent nodes are less than both of its children

When using a Heap, we only remove the root node, which will be either the maximum value or minimum value

👍

When the root is removed, we replace it with **the last node that was added to the heap**

When the root is replaced, it may need to be moved down in the tree



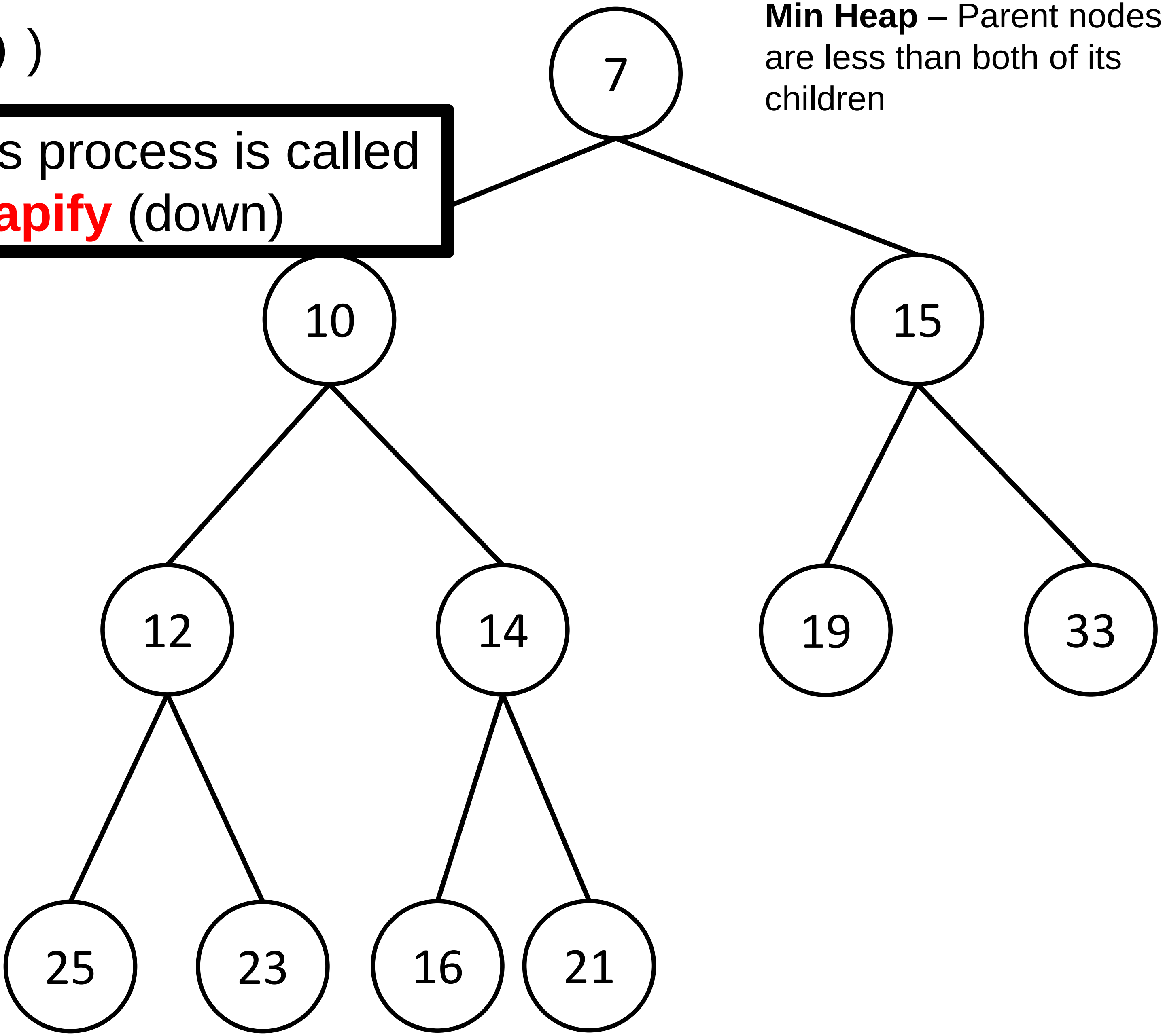
Heap Operations – Removal (**`poll()`**)

When using a Heap, we only remove the root node, which will be either the maximum value or minimum value

When the root is removed, we replace it with **the last node that was added to the heap**

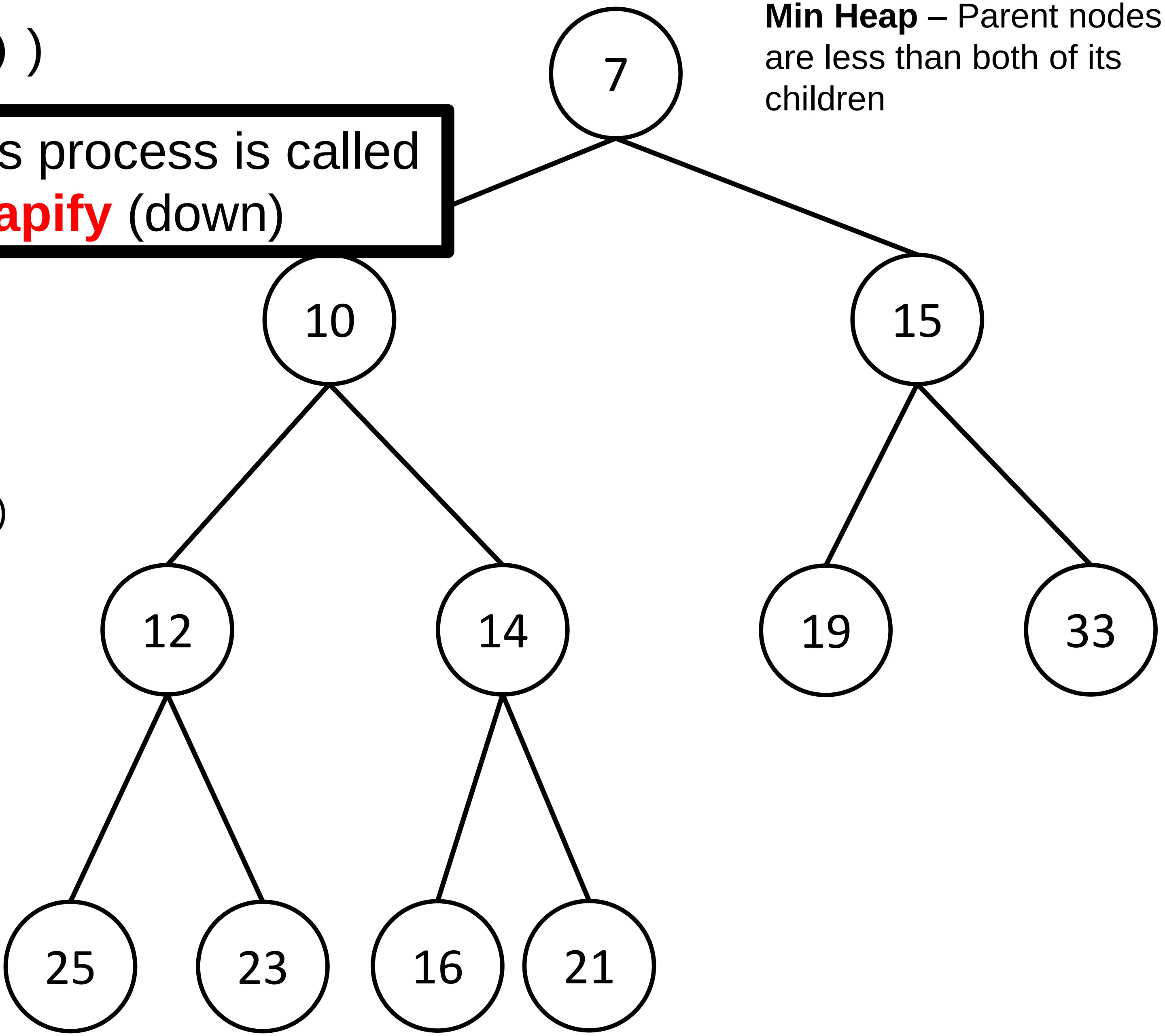
When the root is replaced, it may need to be moved down in the tree

This process is called **Heapify** (down)



Min Heap – Parent nodes are less than both of its children

Heap Operations – Removal (`poll()`)



This process is called **Heapify** (down)

Min Heap – Parent nodes are less than both of its children

Running time?
- Removing root: **O(1)**
- Replacing root: **O(1)** (this will make sense later)
- Heapify down: **O(logn)**

Total running time: **O(logn)**

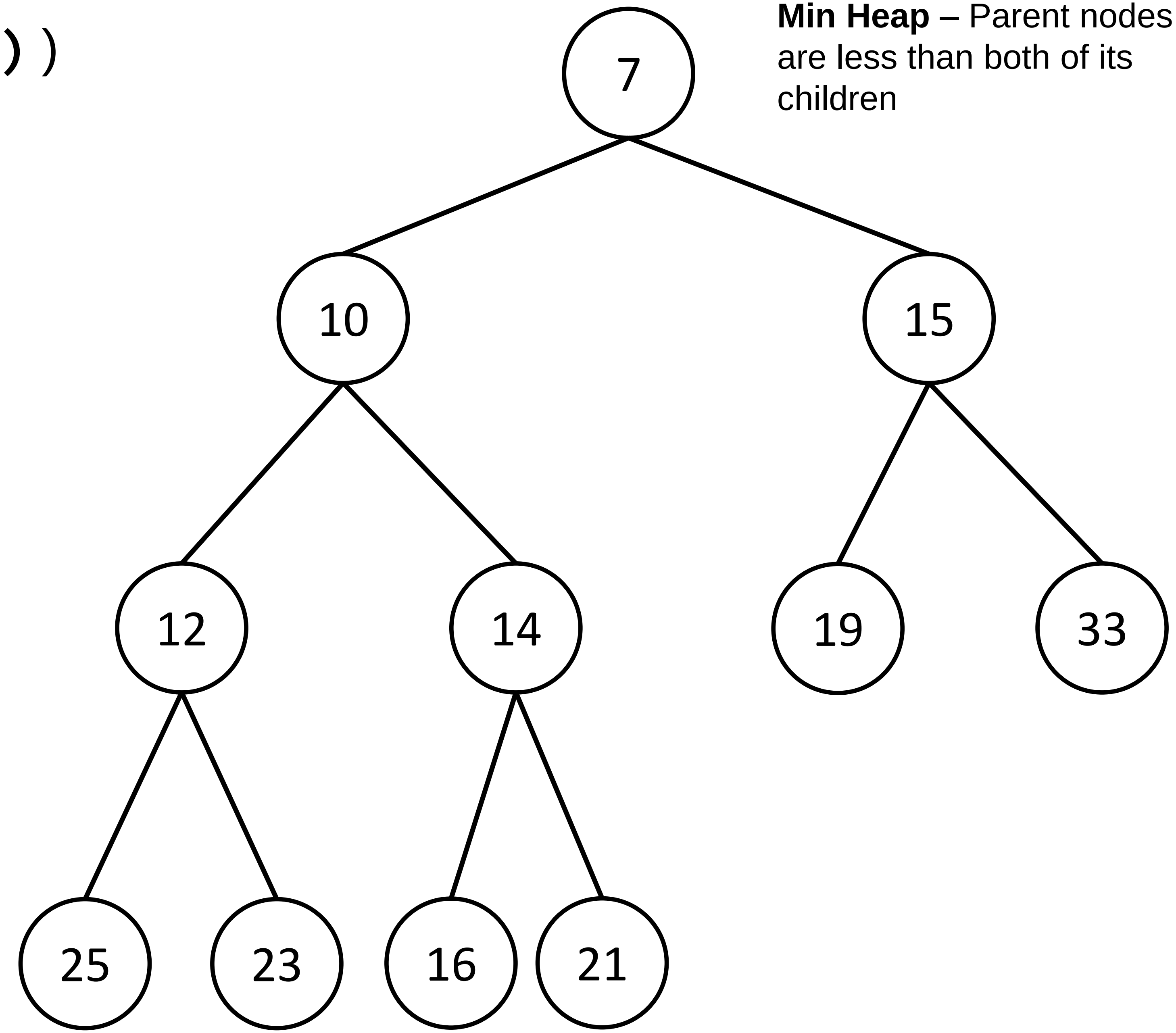
Heap Operations – Removal (**`poll()`**)

Heapify (up)

Moving the new leaf node **up** in the tree

Heapify (down)

Moving the new root node **down** in the tree

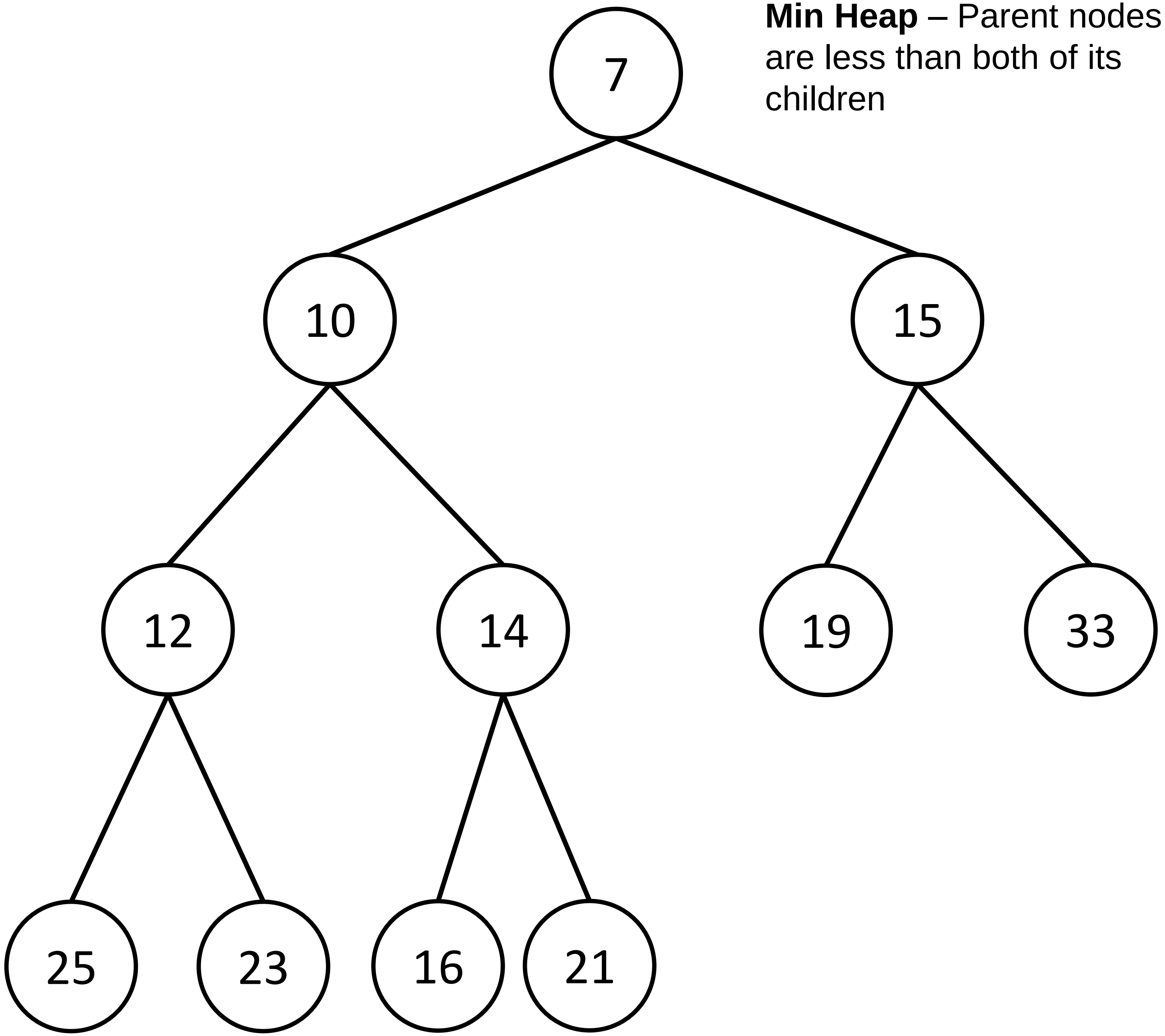


Min Heap – Parent nodes are less than both of its children

Heap Representation

How to represent a heap?

```
public class HeapNode{
    Node leftChild;
    Node rightChild;
    Node parent;
    (…)
}
```

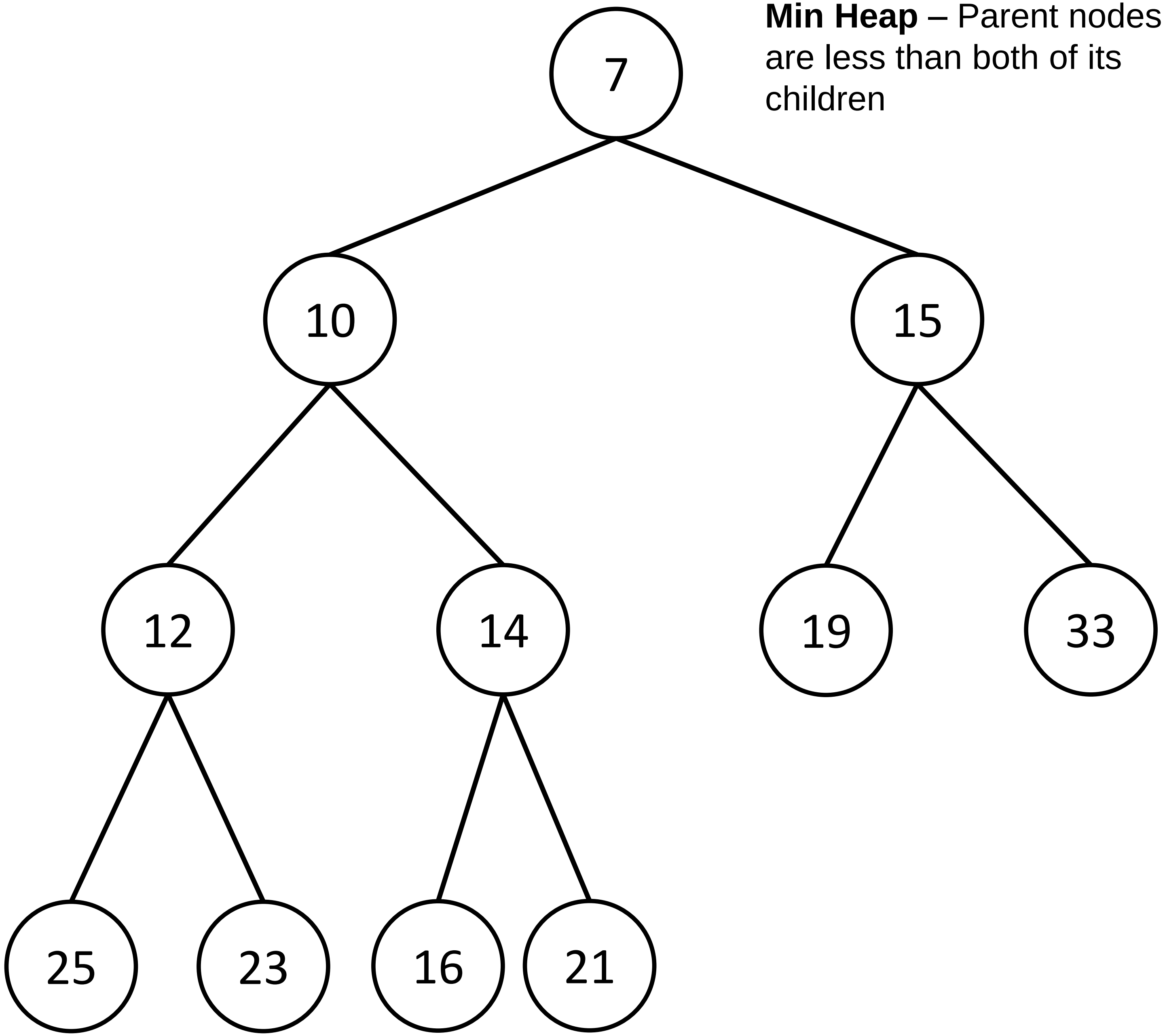


Min Heap – Parent nodes are less than both of its children

Heap Representation

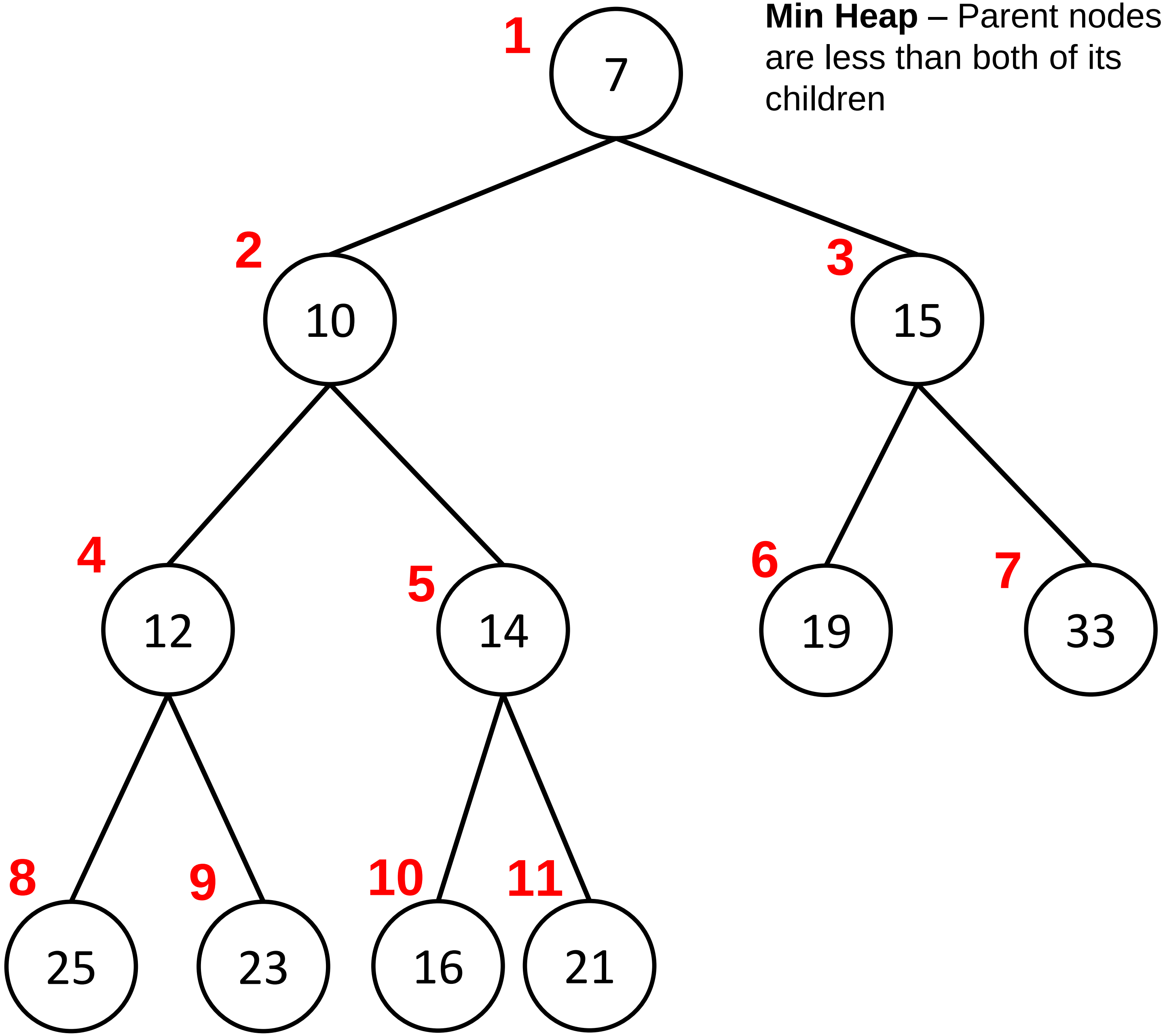
How to represent a heap?

```
public class HeapNode{
      Node leftChild;
      Node rightChild;
      Node parent;
      (…)
}
```



Min Heap – Parent nodes are less than both of its children

Heap Representation



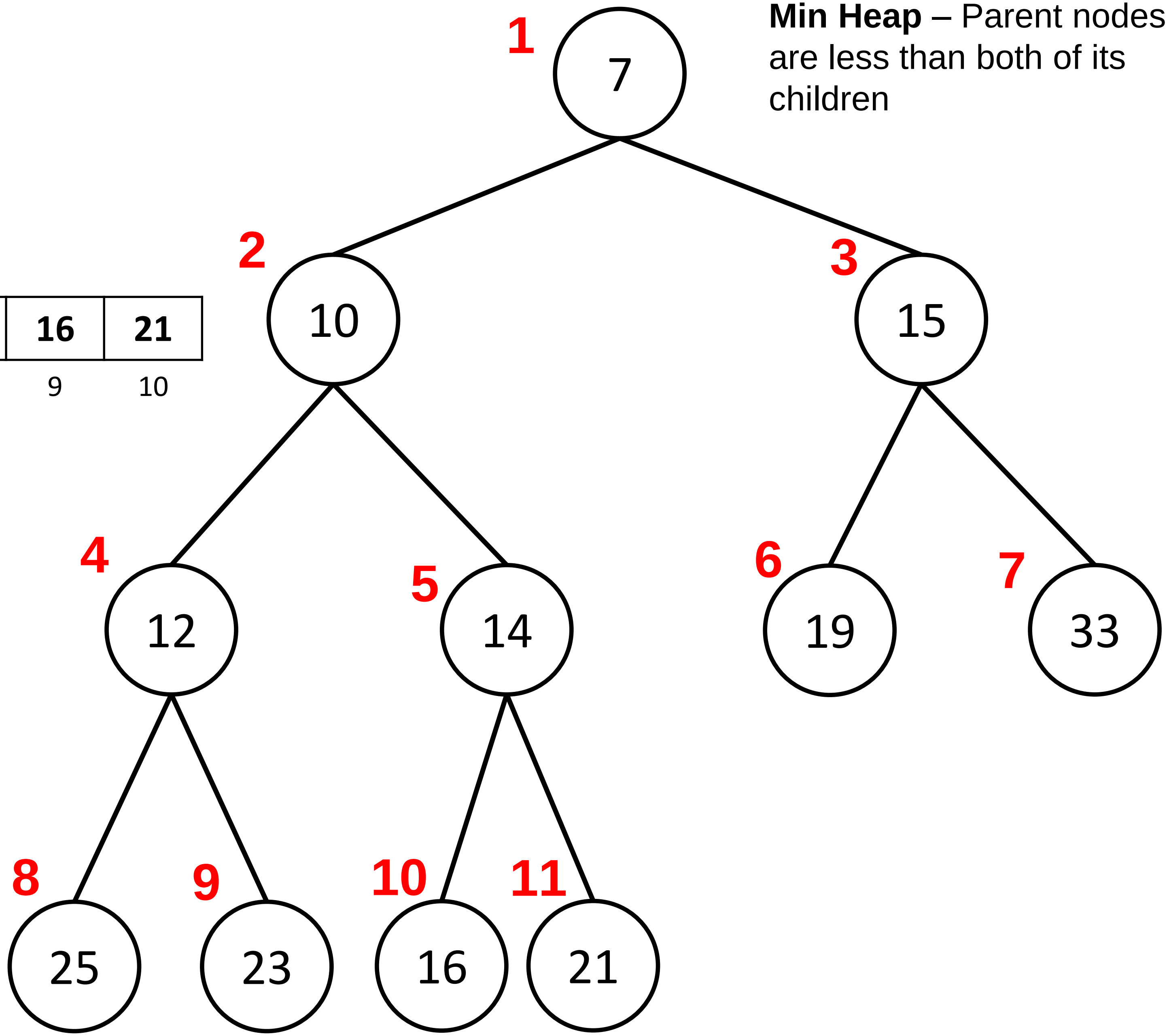
Min Heap – Parent nodes are less than both of its children

Heap Representation

Min Heap – Parent nodes are less than both of its children

Array

7	10	15	12	14	19	33	25	23	16	21
0	1	2	3	4	5	6	7	8	9	10



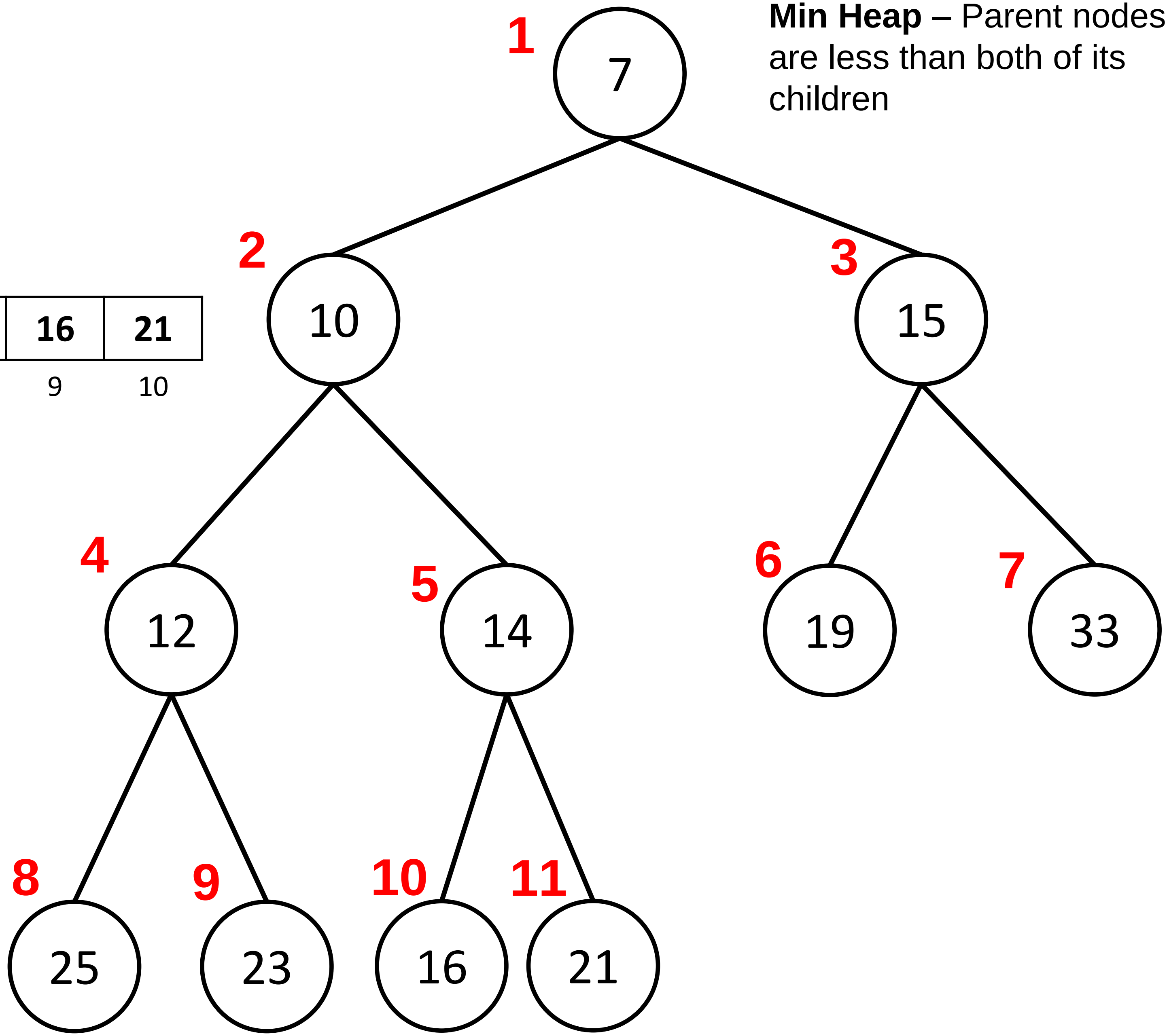
Heap Representation

Array

7	**10**	**15**	**12**	**14**	**19**	**33**	**25**	**23**	**16**	**21**
0	1	2	3	4	5	6	7	8	9	10

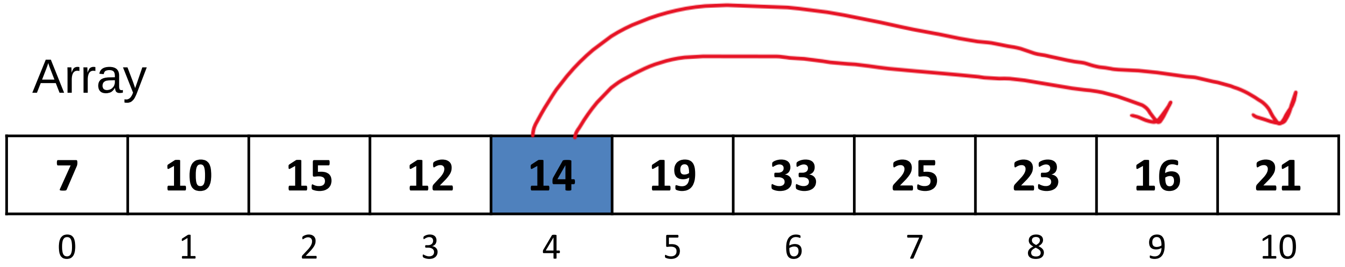
Given a spot in the array, how can we find its children?

Min Heap – Parent nodes are less than both of its children



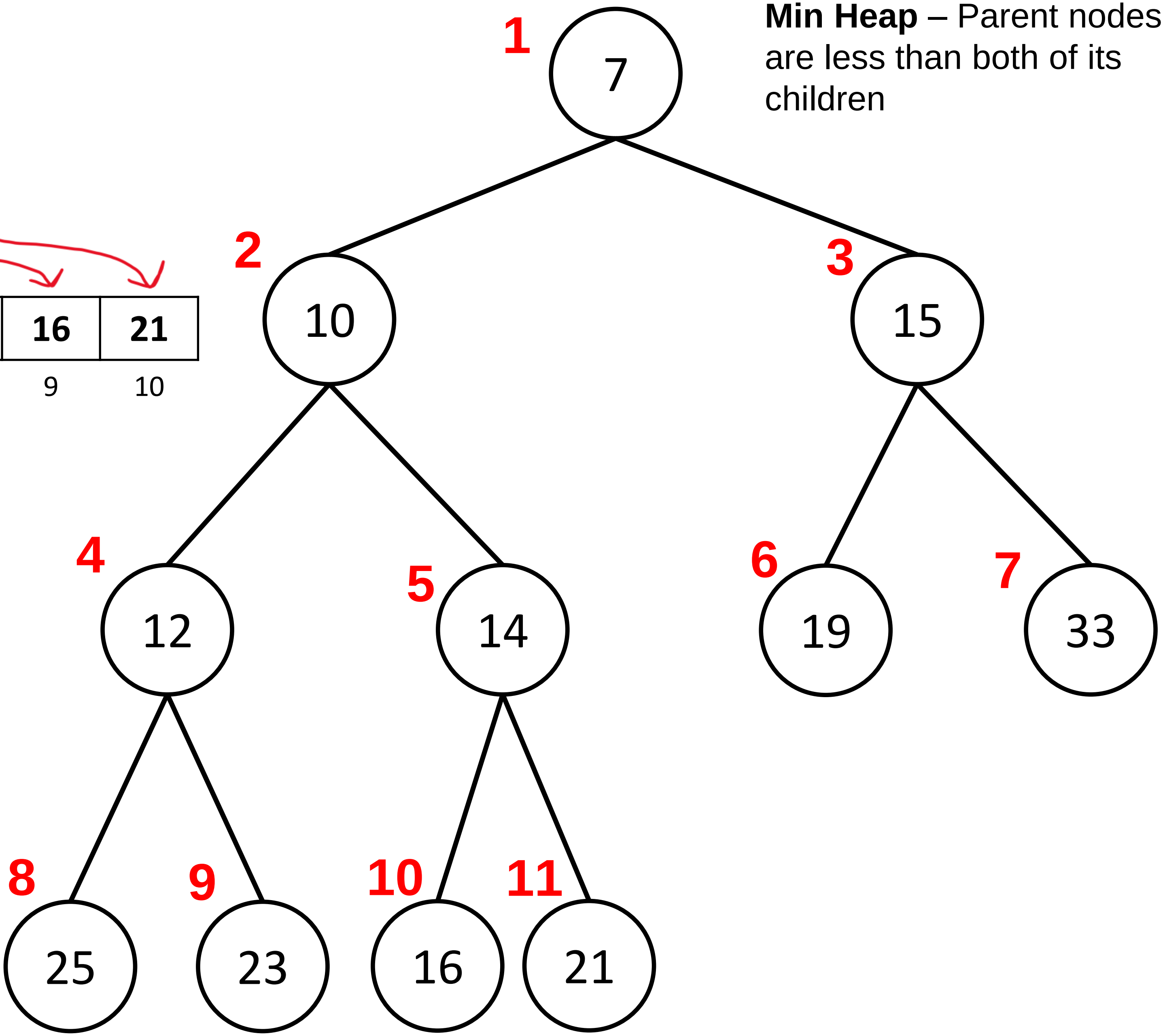
Heap Representation

Array



7	10	15	12	14	19	33	25	23	16	21
0	1	2	3	4	5	6	7	8	9	10

Given a spot in the array, how can we find its children?



Min Heap – Parent nodes are less than both of its children

Heap Representation

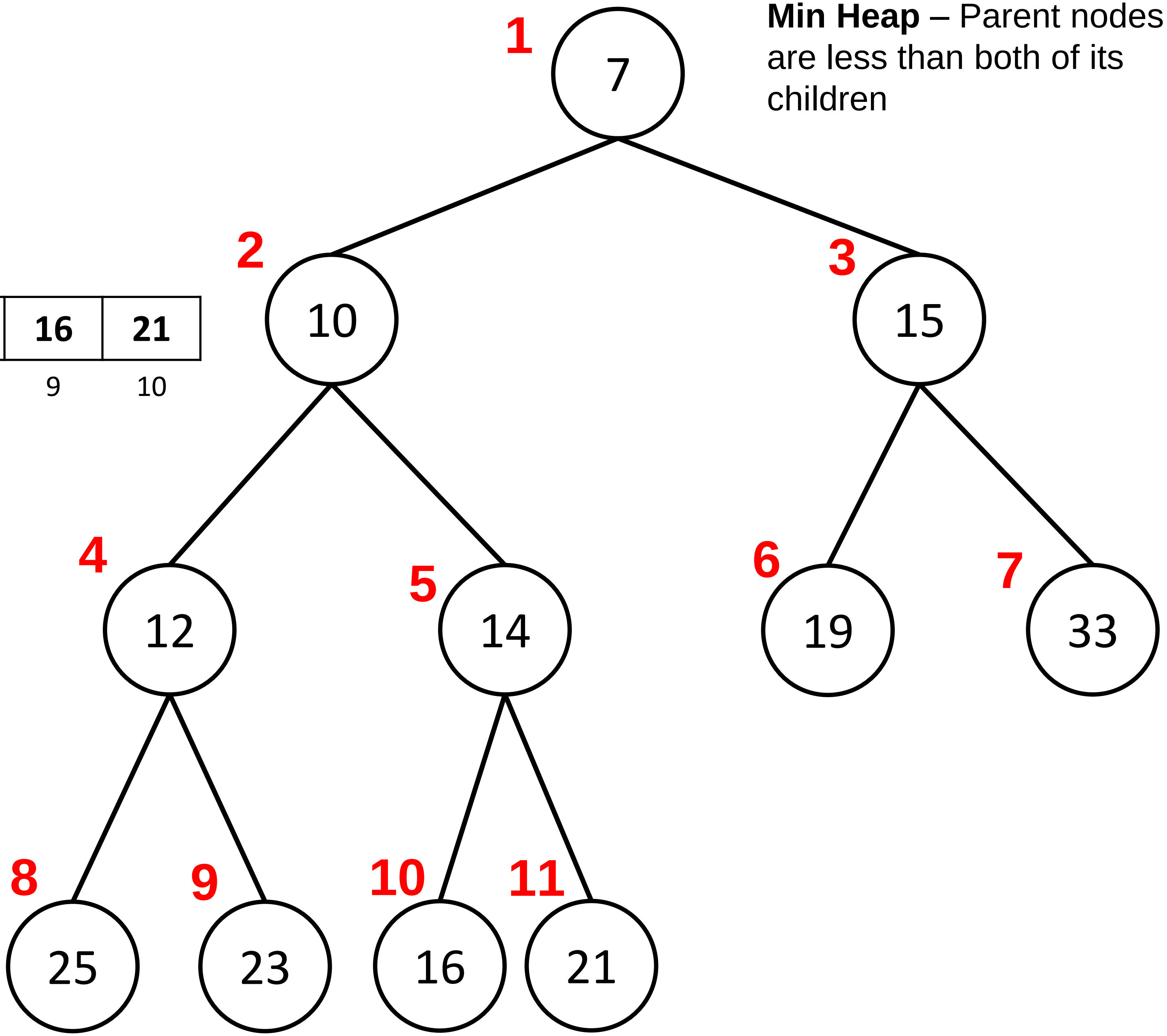
Min Heap – Parent nodes are less than both of its children

Array

7	**10**	**15**	**12**	**14**	**19**	**33**	**25**	**23**	**16**	**21**
0	1	2	3	4	5	6	7	8	9	10

Given a spot in the array, how can we find its children?

Because this is a complete binary tree, there is a pretty nifty formula for this



Heap Representation

Min Heap – Parent nodes are less than both of its children

Array

7	10	15	12	14	19	33	25	23	16	21
0	1	2	3	4	5	6	7	8	9	10

Given a spot in the array, how can we find its children?

Because this is a complete binary tree, there is a pretty nifty formula for this

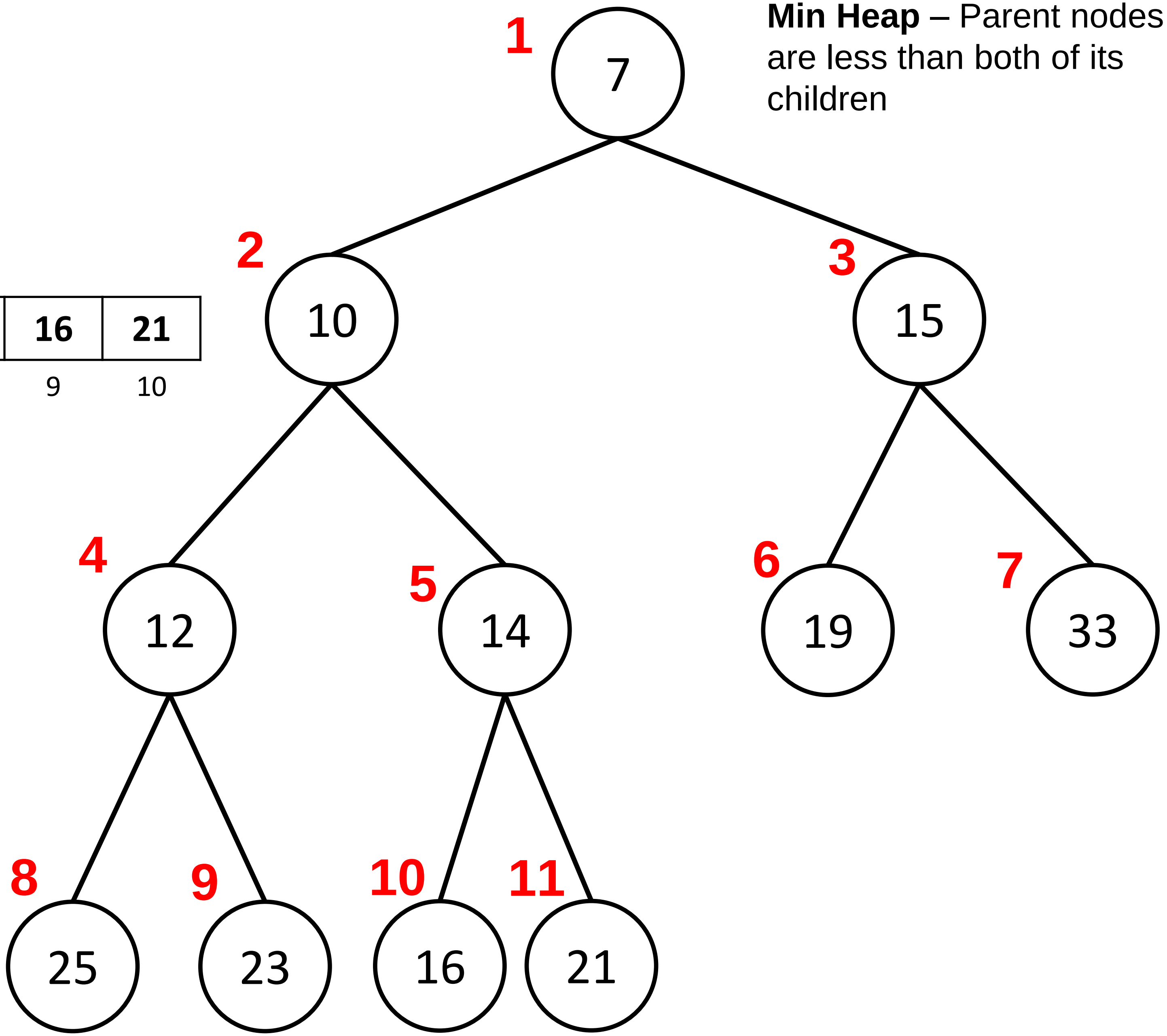
For a given element at index `i`

Its left child will be located at index:

```
2 * i + 1
```

Its right child will be located at index:

```
2 * i + 2
```



Heap Representation

Left Child = **2** * **4** **+ 1** = index **9** !

Array

7	10	15	12	14	19	33	25	23	16	21
0	1	2	3	4	5	6	7	8	9	10

Given a spot in the array, how can we find its children?

Because this is a complete binary tree, there is a pretty nifty formula for this

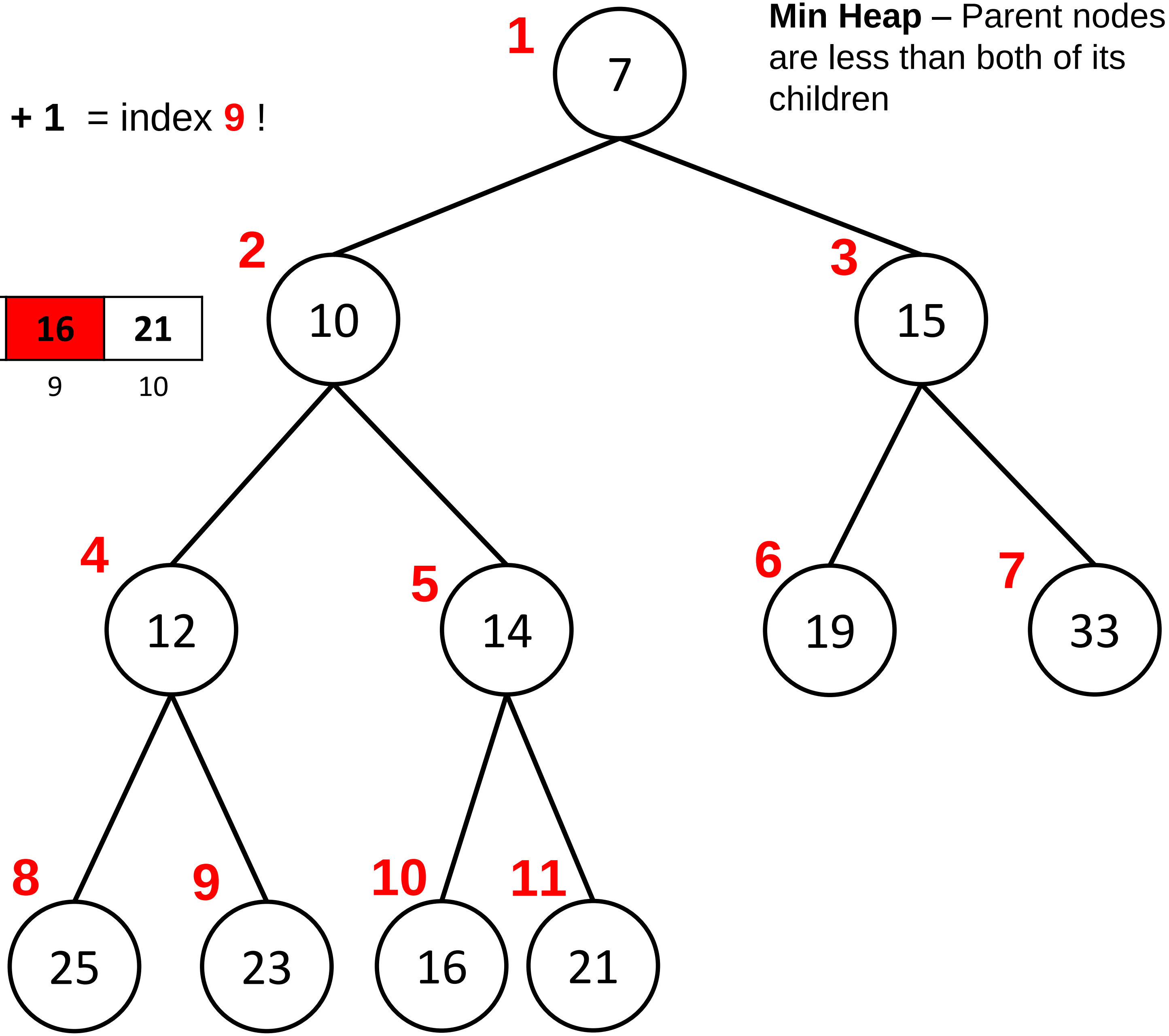
For a given element at index `i`

Its left child will be located at index:

```
2 * i + 1
```

Its right child will be located at index:

```
2 * i + 2
```



Min Heap – Parent nodes are less than both of its children

Heap Representation

Left Child = **2 * 4 + 1** = index **9** !

Right Child = **2 * 4 + 2** = index **10** !

Array

7	10	15	12	14	19	33	25	23	16	21
0	1	2	3	4	5	6	7	8	9	10

Given a spot in the array, how can we find its children?

Because this is a complete binary tree, there is a pretty nifty formula for this

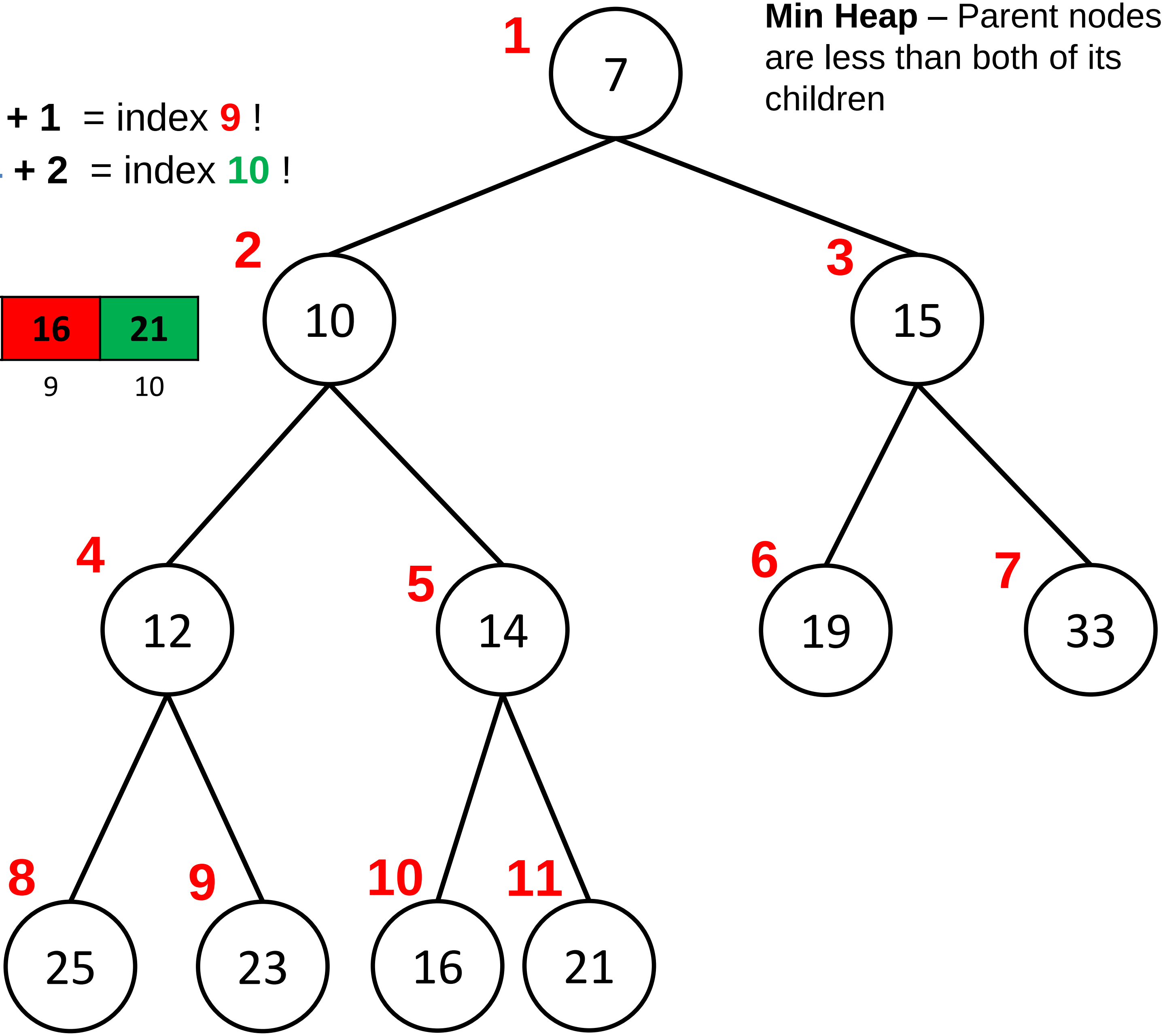
For a given element at index `i`

Its left child will be located at index:

```
2 * i + 1
```

Its right child will be located at index:

```
2 * i + 2
```



Min Heap – Parent nodes are less than both of its children

Heap Representation

Left Child = **2 * 0 + 1** = index 1 !

Right Child = **2 * 0 + 2** = index 2 !

Array

7	**10**	**15**	**12**	**14**	**19**	**33**	**25**	**23**	**16**	**21**
0	1	2	3	4	5	6	7	8	9	10

Given a spot in the array, how can we find its children?

Because this is a complete binary tree, there is a pretty nifty formula for this

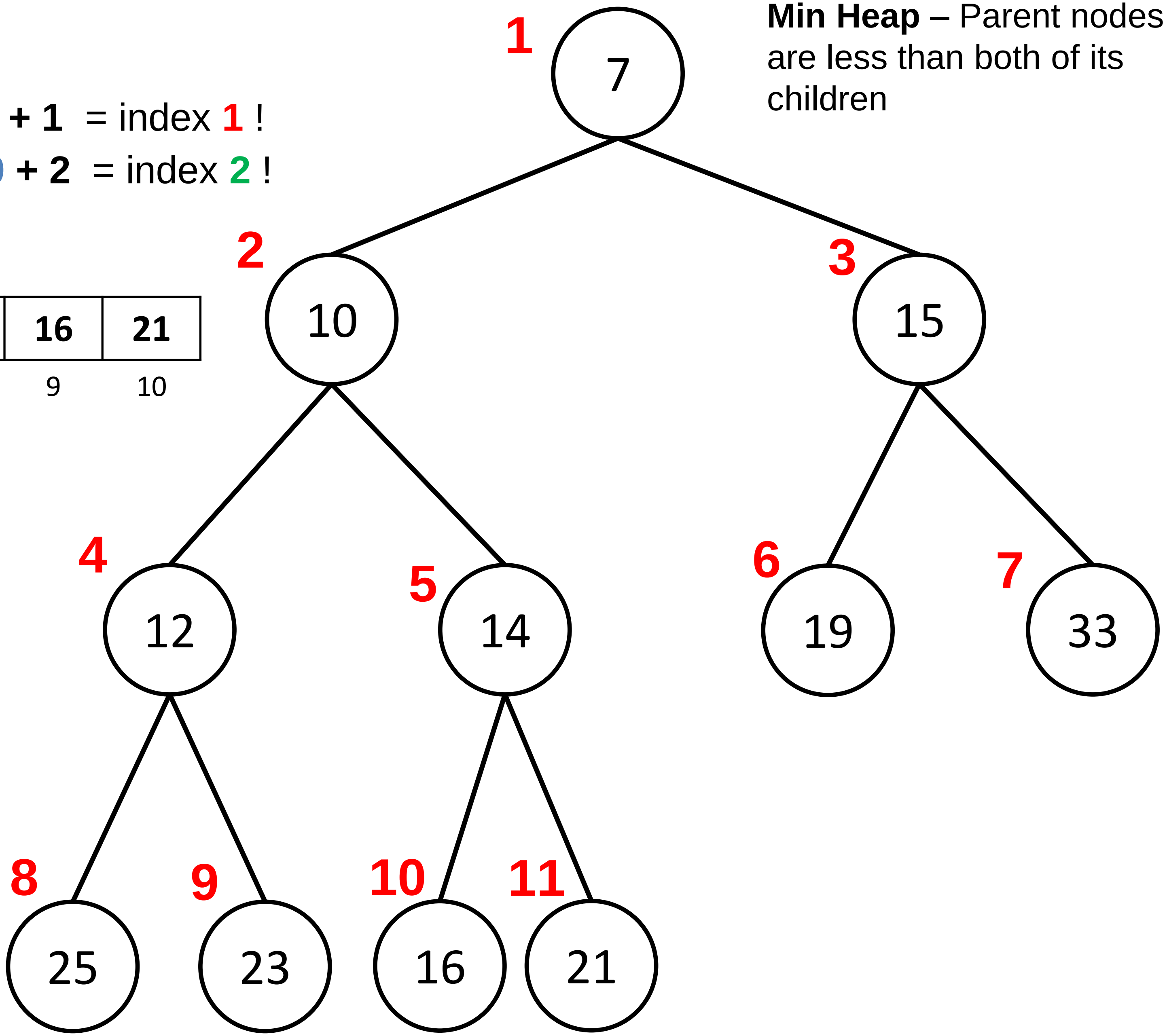
For a given element at index **`i`**

Its left child will be located at index:

`2 * i + 1`

Its right child will be located at index:

`2 * i + 2`



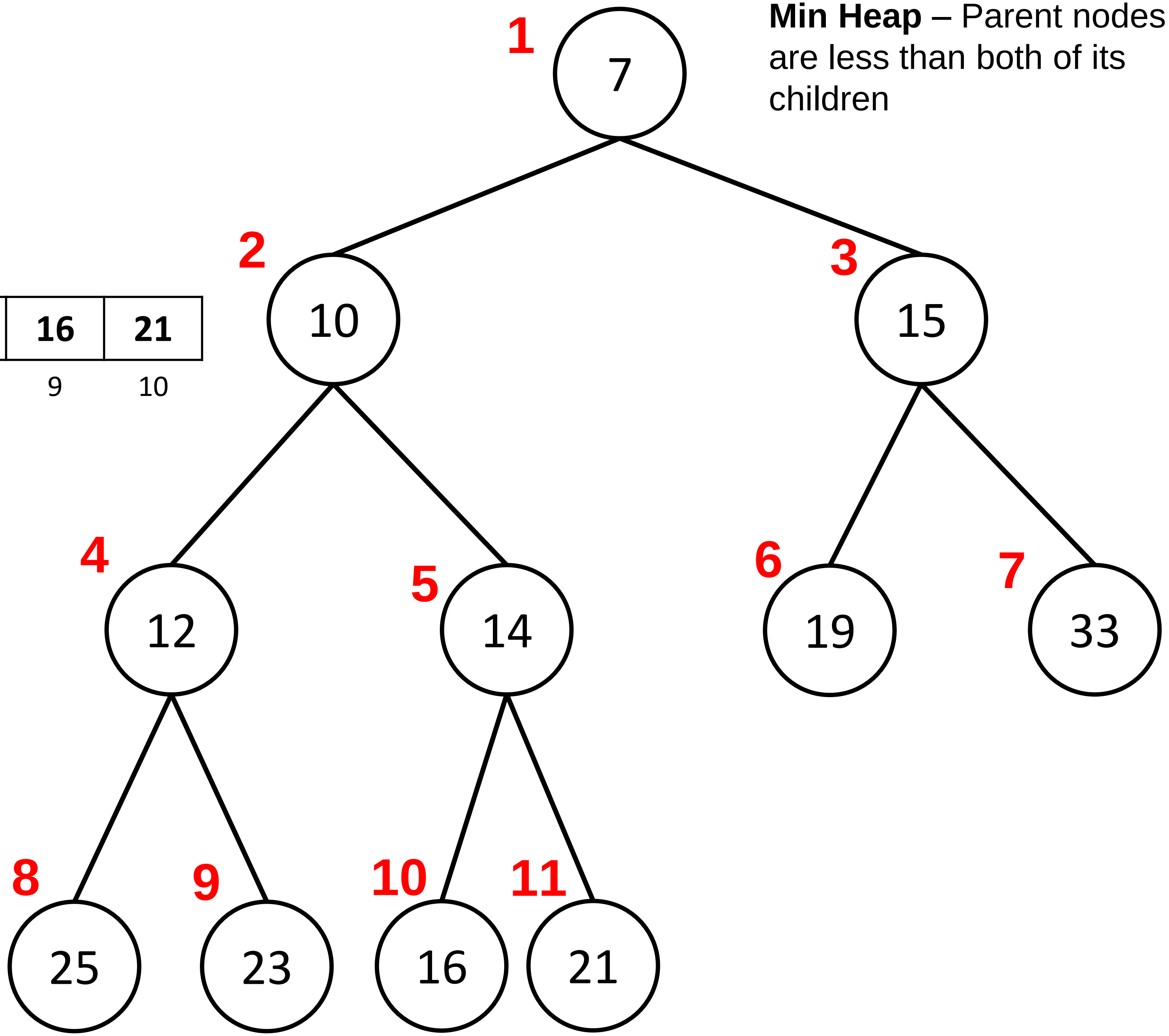
Min Heap – Parent nodes are less than both of its children

Heap Representation

Array

7	10	15	12	14	19	33	25	23	16	21
0	1	2	3	4	5	6	7	8	9	10

Given a spot in the array, how can we find its parent?



Min Heap – Parent nodes are less than both of its children

Heap Representation

Array

7	**10**	**15**	**12**	**14**	**19**	**33**	**25**	**23**	**16**	**21**
0	1	2	3	4	5	6	7	8	9	10

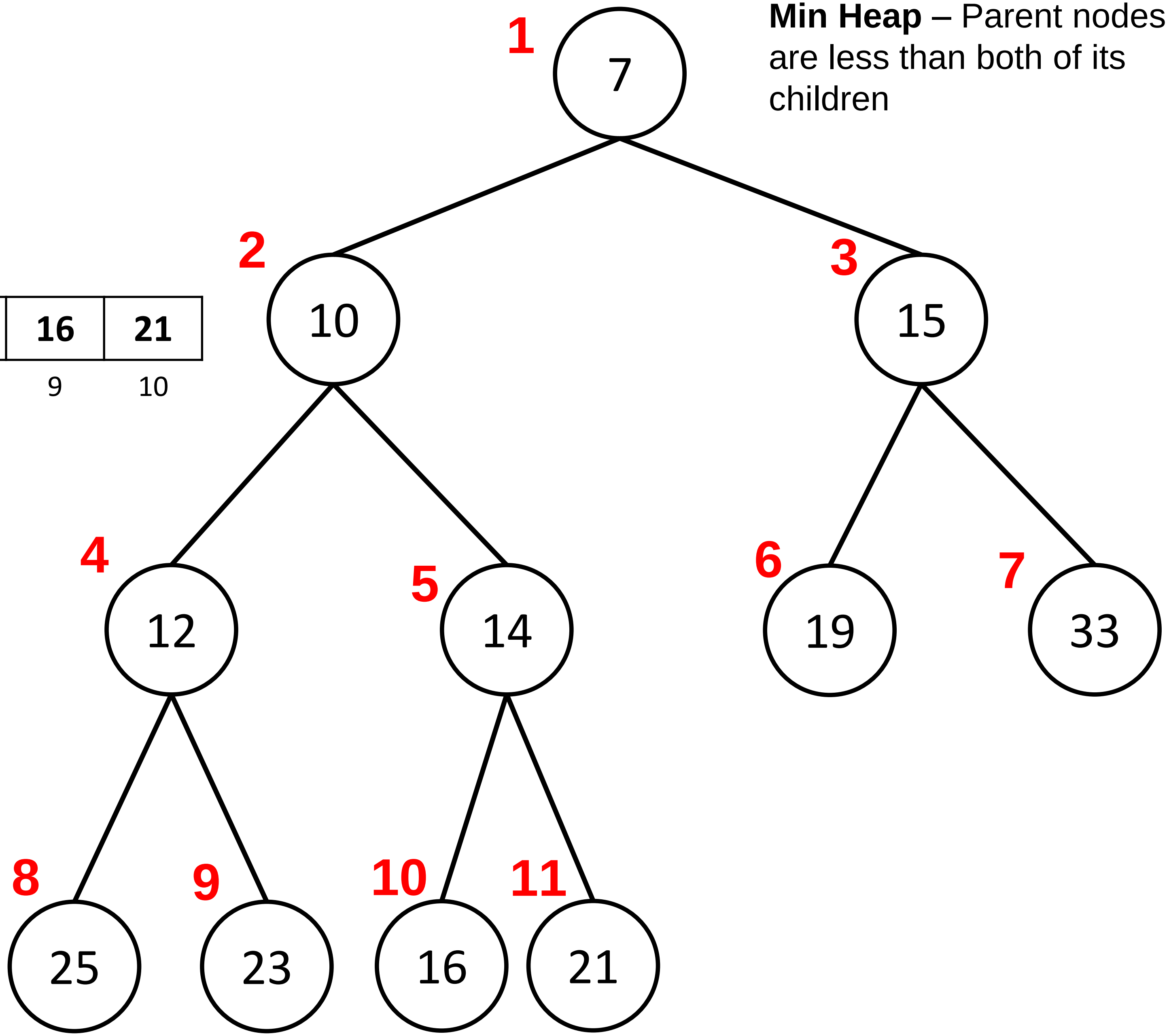
Given a spot in the array, how can we find its parent?

Because this is a complete binary tree, there is a pretty nifty formula for this

Given an index **`i`**
Its parent will be located at index:

`(i - 1) / 2`

(remember that the **`/`** operator will **floor** the answer)



Min Heap – Parent nodes are less than both of its children

Heap Representation

Parent = (6 - 1) / 2 = Index **2**

Array

7	**10**	**15**	**12**	**14**	**19**	**33**	**25**	**23**	**16**	**21**
0	1	2	3	4	5	6	7	8	9	10

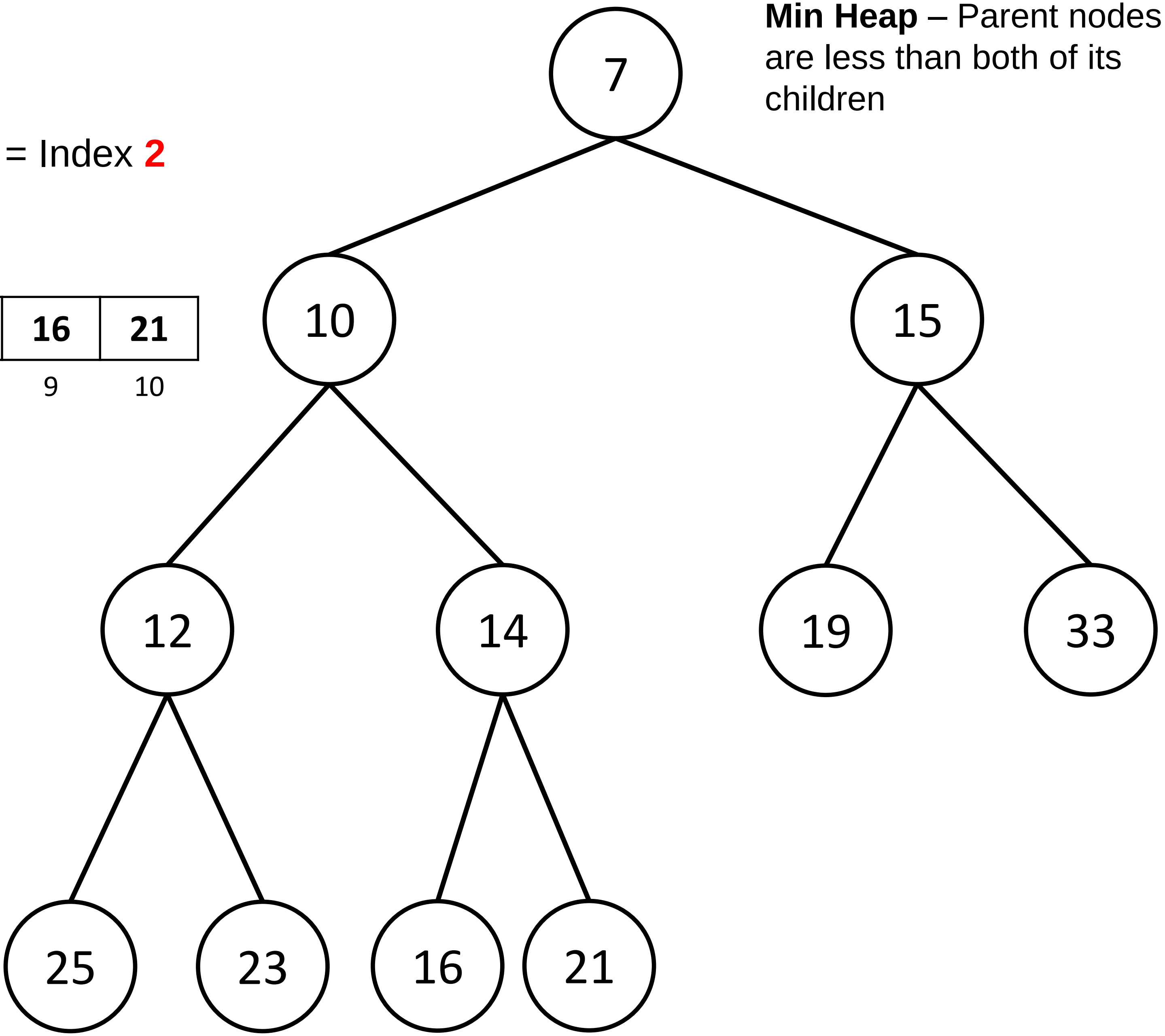
Given a spot in the array, how can we find its parent?

Because this is a complete binary tree, there is a pretty nifty formula for this

Given an index `i`
Its parent will be located at index:

```
(i - 1) / 2
```

(remember that the `/` operator will **floor** the answer)



Min Heap – Parent nodes are less than both of its children

Heap Representation

Parent = (3 - 1) / 2 = Index **1**

Array

7	**10**	**15**	**12**	**14**	**19**	**33**	**25**	**23**	**16**	**21**
0	1	2	3	4	5	6	7	8	9	10

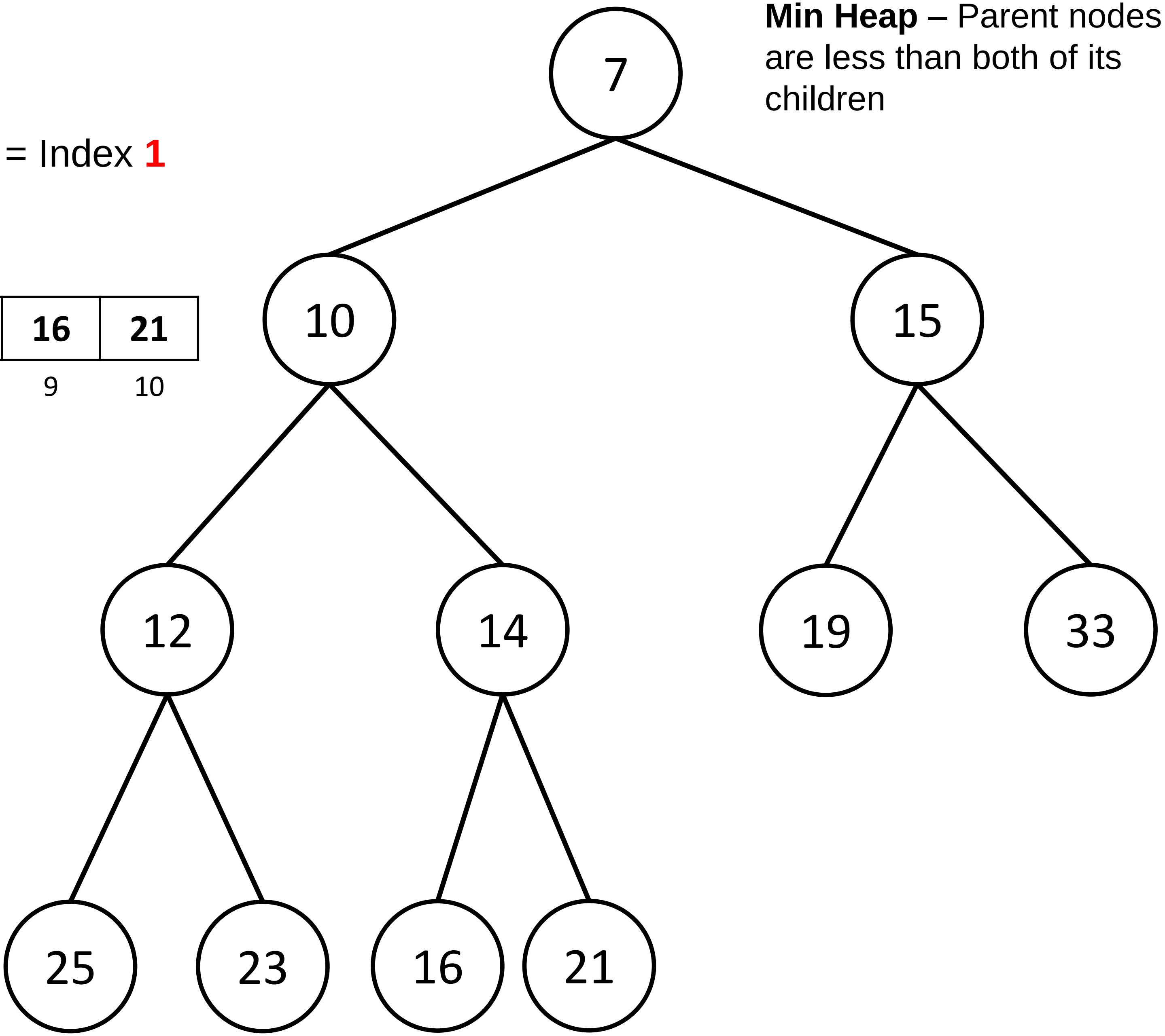
Given a spot in the array, how can we find its parent?

Because this is a complete binary tree, there is a pretty nifty formula for this

Given an index **`i`**
Its parent will be located at index:

`(i - 1) / 2`

(remember that the **`/`** operator will **floor** the answer)



Min Heap – Parent nodes are less than both of its children

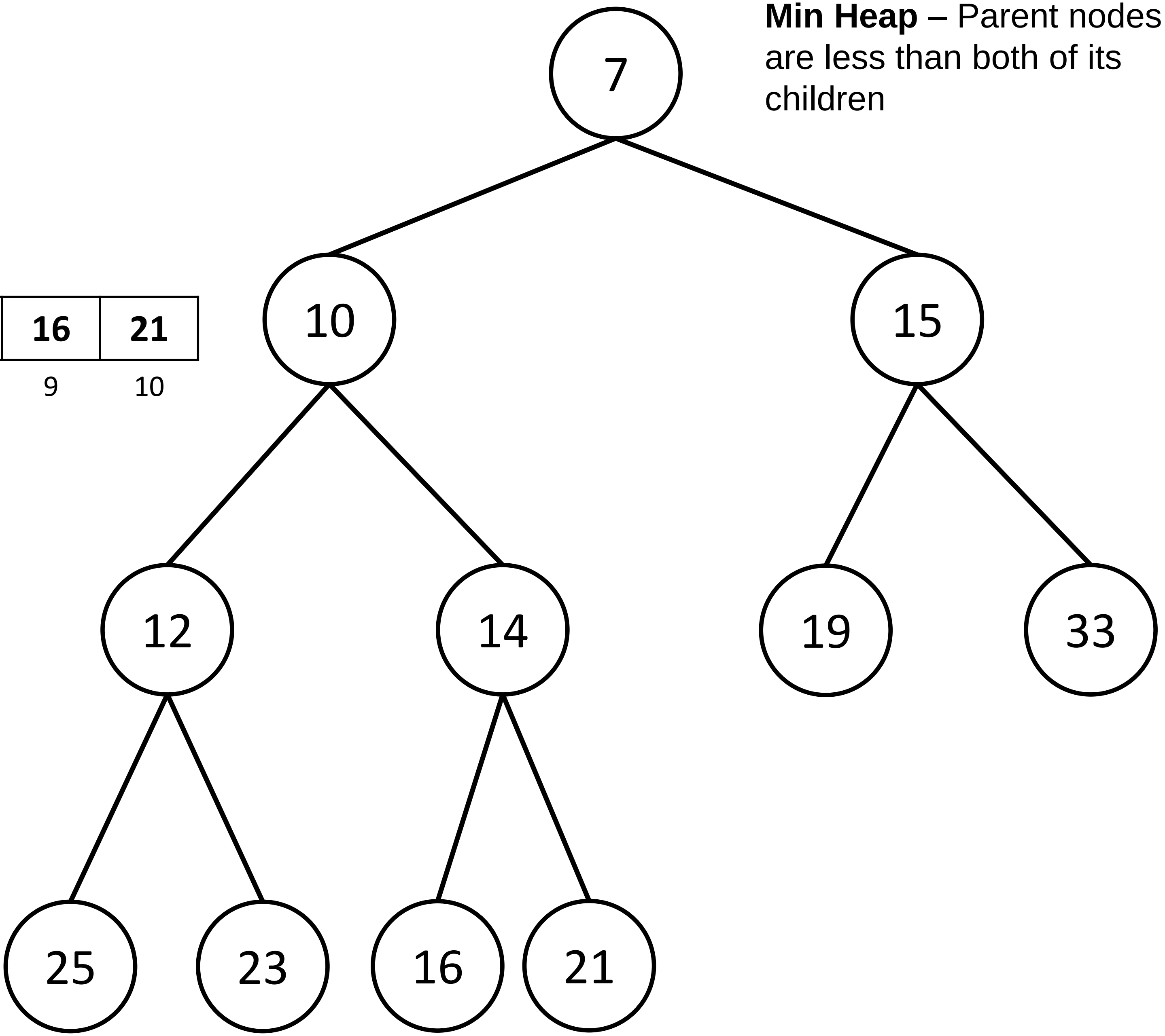
Heap Representation

Array

7	**10**	**15**	**12**	**14**	**19**	**33**	**25**	**23**	**16**	**21**
0	1	2	3	4	5	6	7	8	9	10

We can represent our tree with an array! We have formulas to find the left child, right child, and parent for a given node

Left Child	**2 * i + 1**
Right Child	**2 * i + 2**
Parent	**(i - 1) / 2**



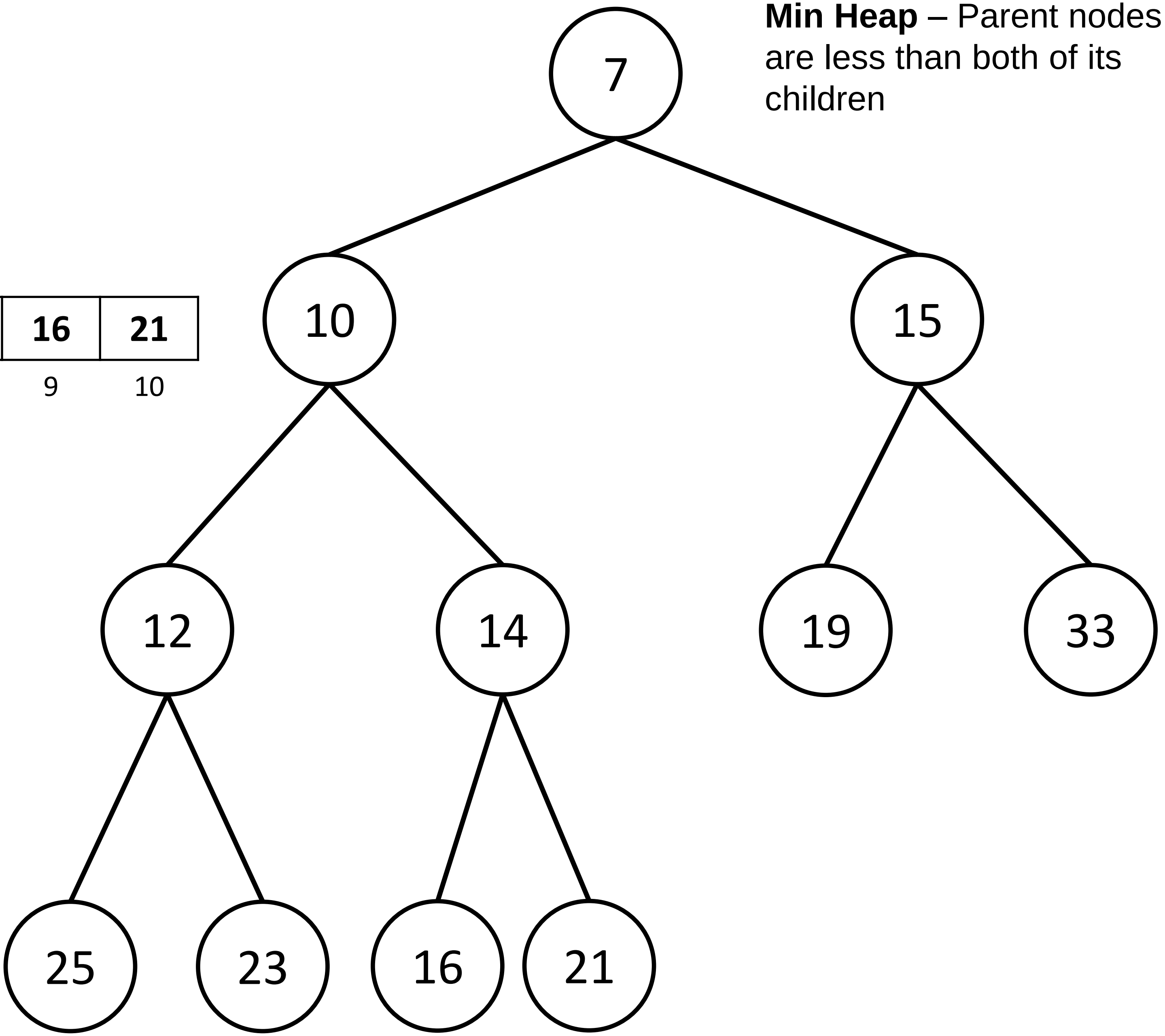
Min Heap – Parent nodes are less than both of its children

Heap Representation

Array

7	**10**	**15**	**12**	**14**	**19**	**33**	**25**	**23**	**16**	**21**
0	1	2	3	4	5	6	7	8	9	10

```
insert(11);
```



Min Heap – Parent nodes are less than both of its children

Heap Representation

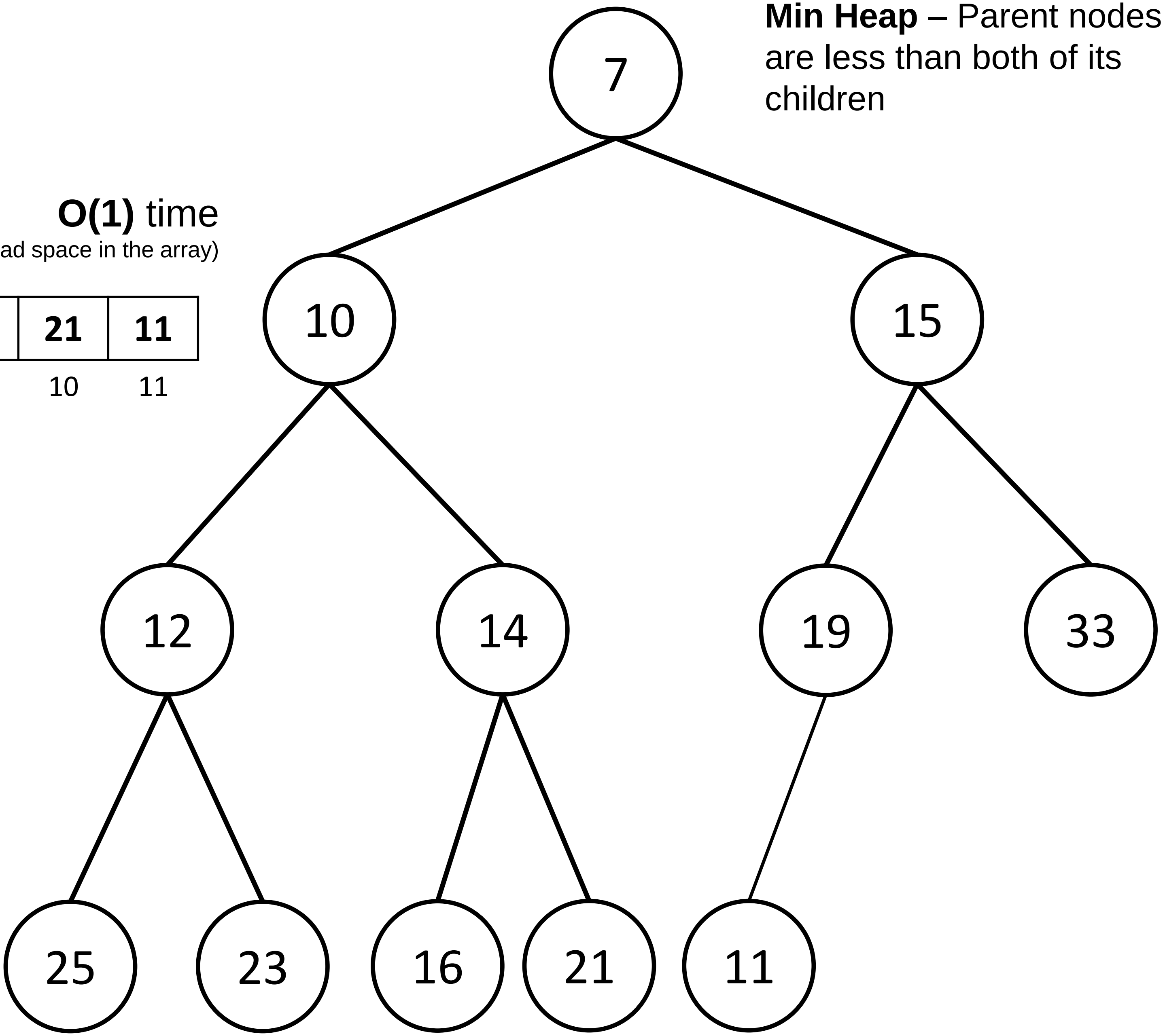
Min Heap – Parent nodes are less than both of its children

O(1) time
(assuming we had space in the array)

Array

7	**10**	**15**	**12**	**14**	**19**	**33**	**25**	**23**	**16**	**21**	**11**
0	1	2	3	4	5	6	7	8	9	10	11

```
insert(11);
```



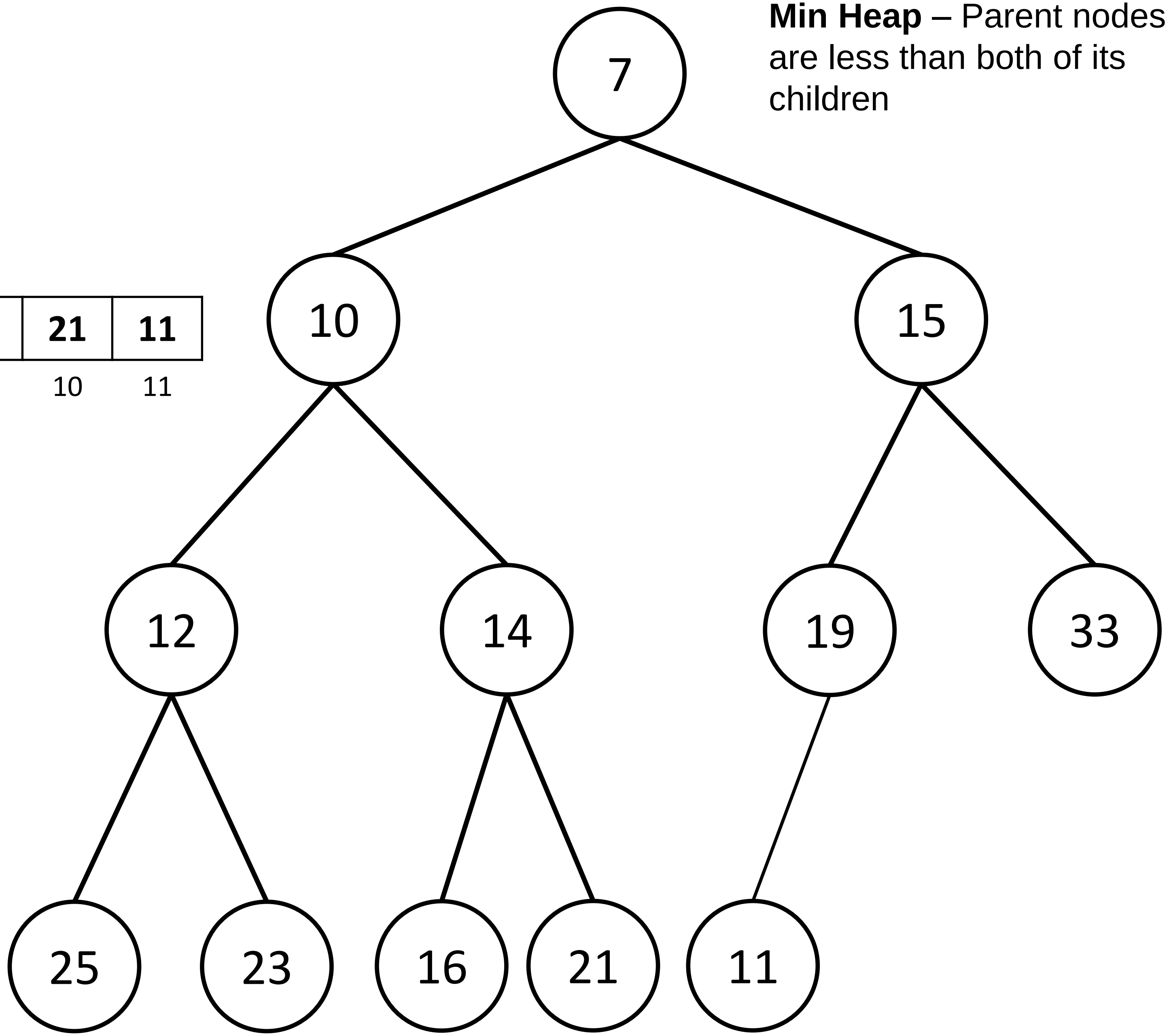
Heap Representation

Array

7	**10**	**15**	**12**	**14**	**19**	**33**	**25**	**23**	**16**	**21**	**11**
0	1	2	3	4	5	6	7	8	9	10	11

```
insert(11);
```

Time to Heapify Up!



Min Heap – Parent nodes are less than both of its children

Heap Representation

Left Child `2 * i + 1`

Right Child `2 * i + 2`

Parent `(i - 1) / 2`

Array

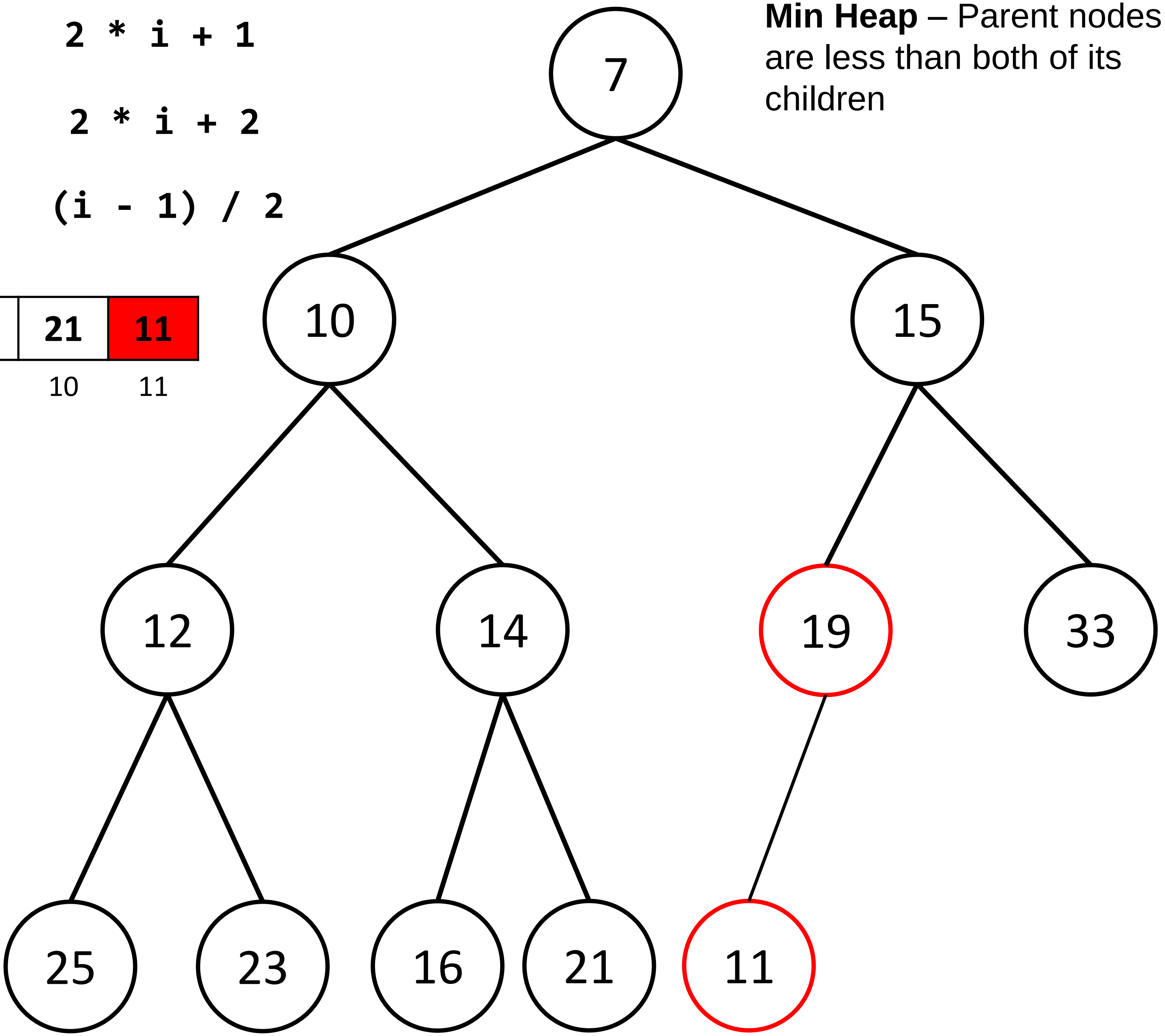
7	**10**	**15**	**12**	**14**	**19**	**33**	**25**	**23**	**16**	**21**	**11**
0	1	2	3	4	5	6	7	8	9	10	11

```
insert(11);
```

Time to Heapify Up!

11’s parent is located at (11 – 1) / 2 = **5**

Min Heap – Parent nodes are less than both of its children



Heap Representation

Left Child `2 * i + 1`

Right Child `2 * i + 2`

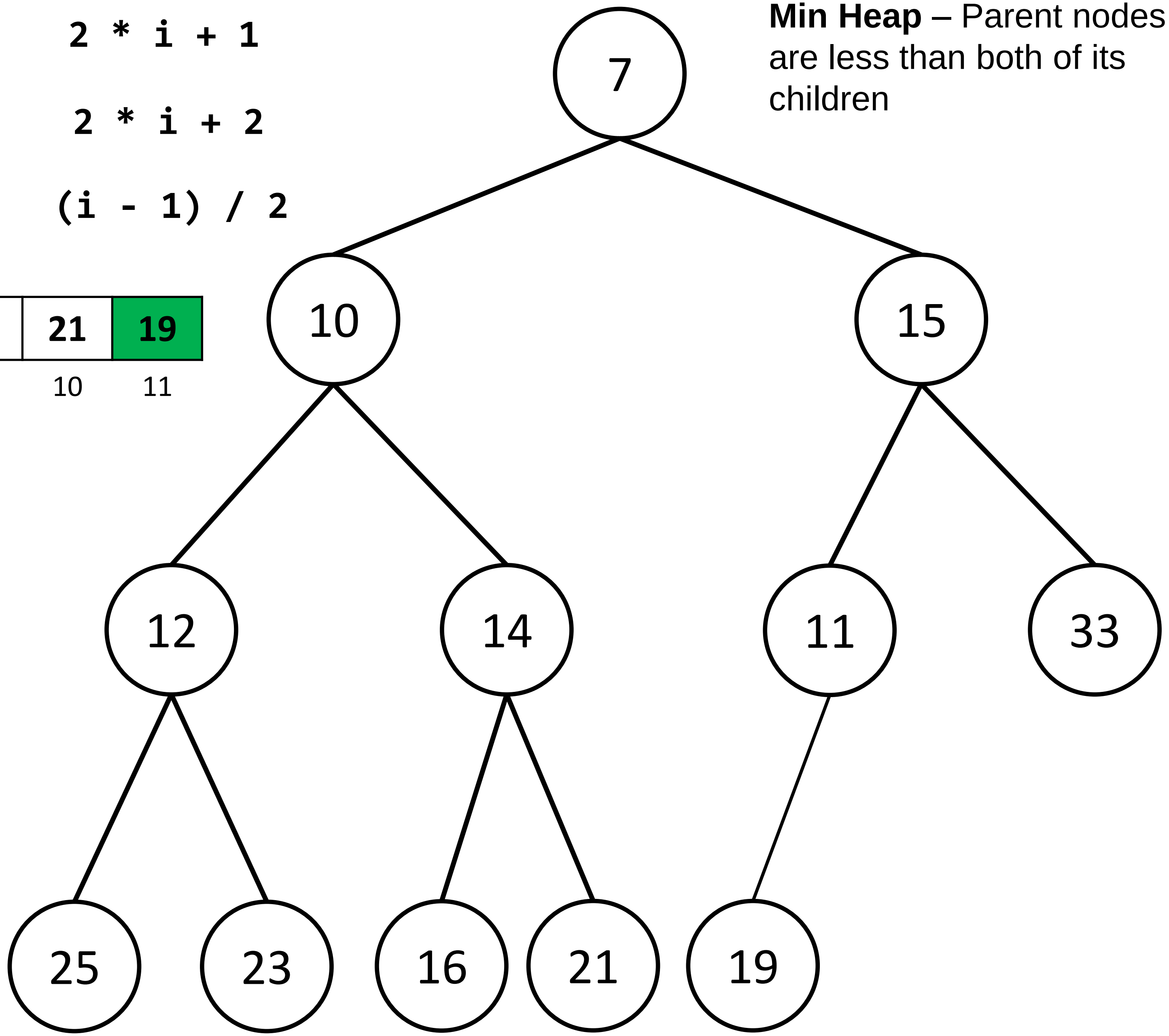
Parent `(i - 1) / 2`

Array

7	**10**	**15**	**12**	**14**	**11**	**33**	**25**	**23**	**16**	**21**	**19**
0	1	2	3	4	5	6	7	8	9	10	11

```
insert(11);
```

Time to Heapify Up!



Min Heap – Parent nodes are less than both of its children

Heap Representation

Left Child	`2 * i + 1`
Right Child	`2 * i + 2`
Parent	`(i - 1) / 2`

Min Heap – Parent nodes are less than both of its children

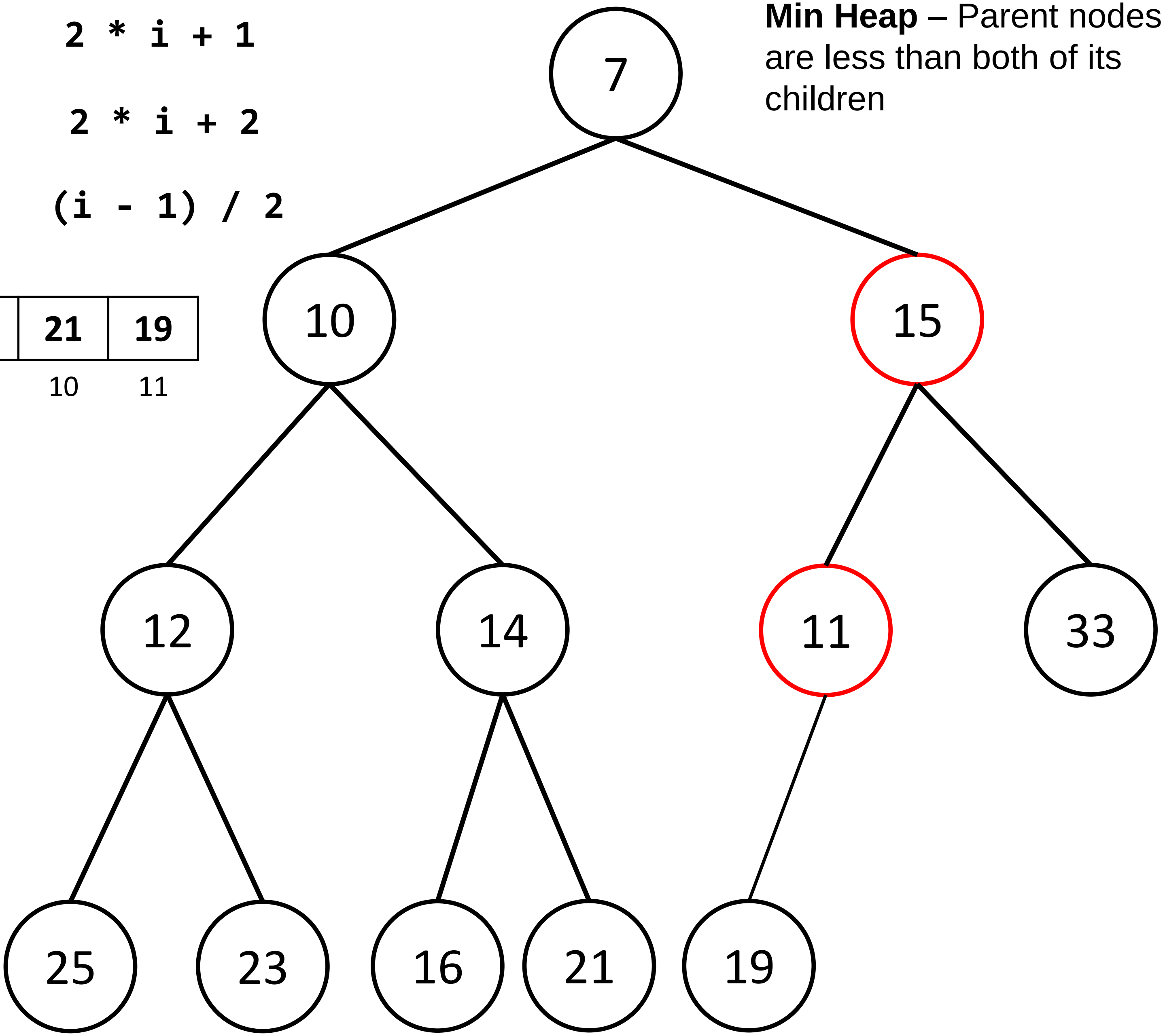
Array

7	10	15	12	14	11	33	25	23	16	21	19
0	1	2	3	4	5	6	7	8	9	10	11

```
insert(11);
```

Time to Heapify Up!

11’s parent is located at (5 – 1) / 2 = **2**



Heap Representation

Left Child `2 * i + 1`

Right Child `2 * i + 2`

Parent `(i - 1) / 2`

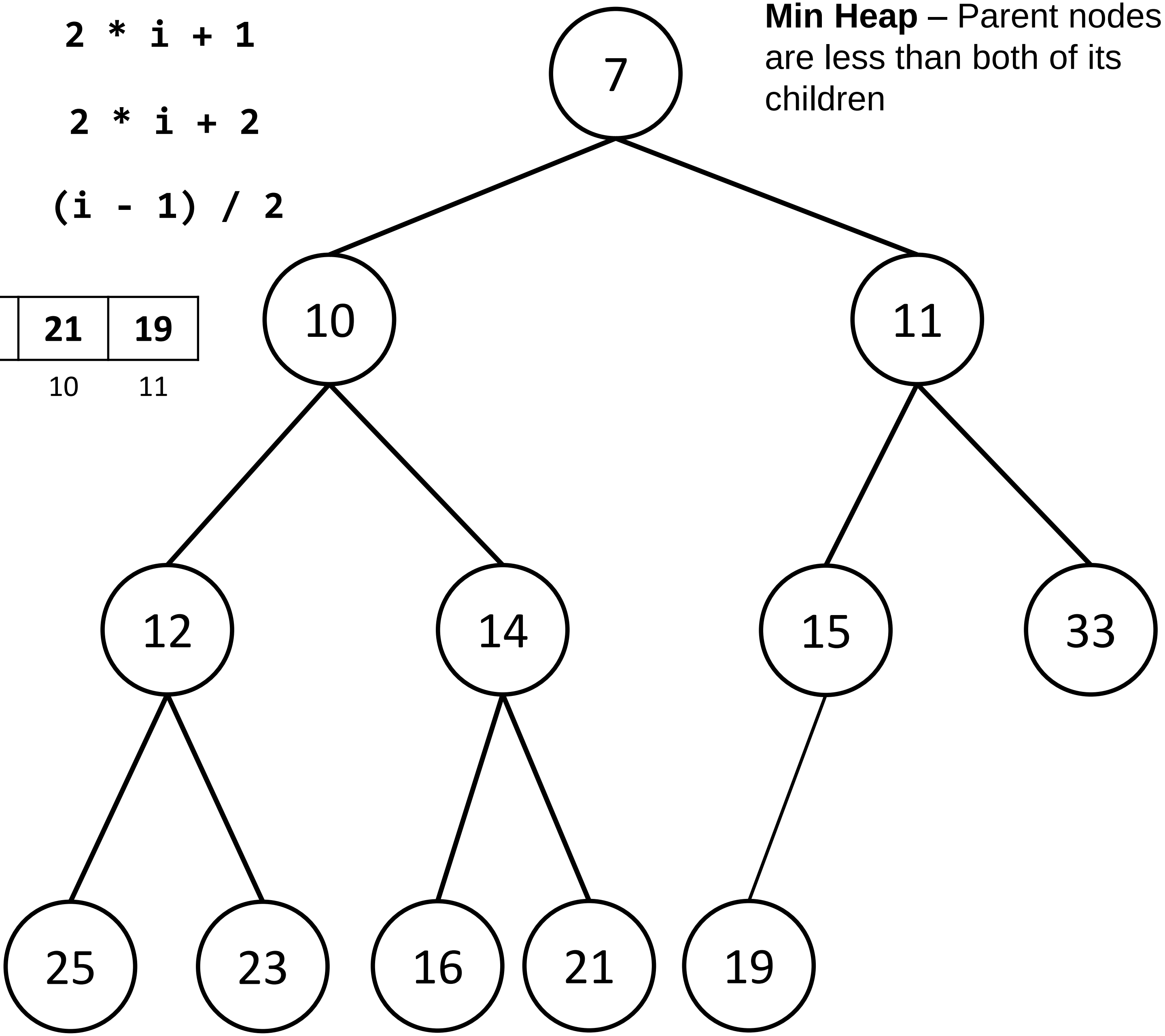
Array

7	**10**	**11**	**12**	**14**	**15**	**33**	**25**	**23**	**16**	**21**	**19**
0	1	2	3	4	5	6	7	8	9	10	11

```
insert(11);
```

Time to Heapify Up!

11’s parent is located at (5 – 1) / 2 = **2**



Min Heap – Parent nodes are less than both of its children

Heap Representation

Left Child `2 * i + 1`

Right Child `2 * i + 2`

Parent `(i - 1) / 2`

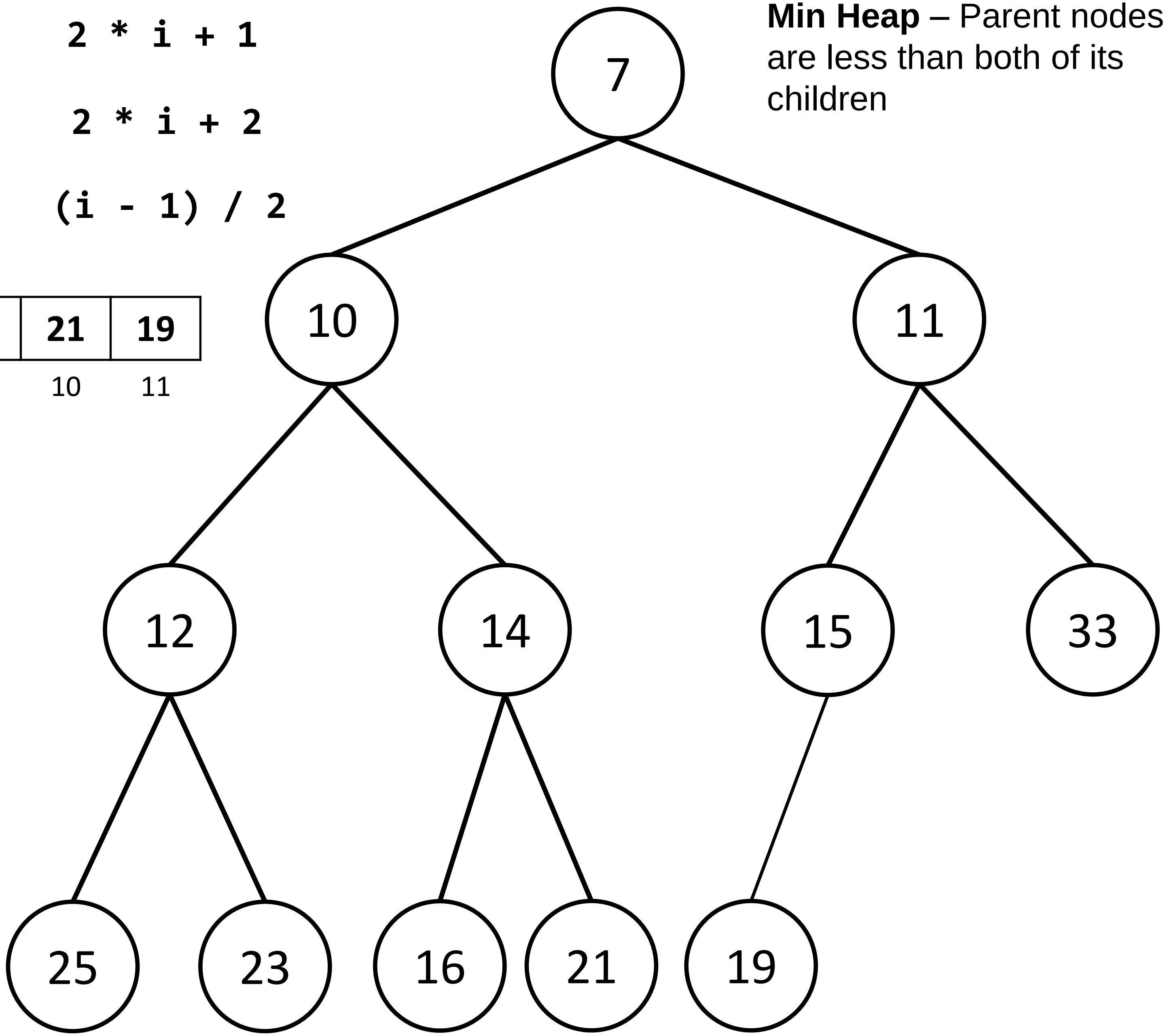
Array

7	**10**	**11**	**12**	**14**	**15**	**33**	**25**	**23**	**16**	**21**	**19**
0	1	2	3	4	5	6	7	8	9	10	11

```
insert(11);
```

Time to Heapify Up!

👍



Min Heap – Parent nodes are less than both of its children

Heap Representation

Left Child `2 * i + 1`

Right Child `2 * i + 2`

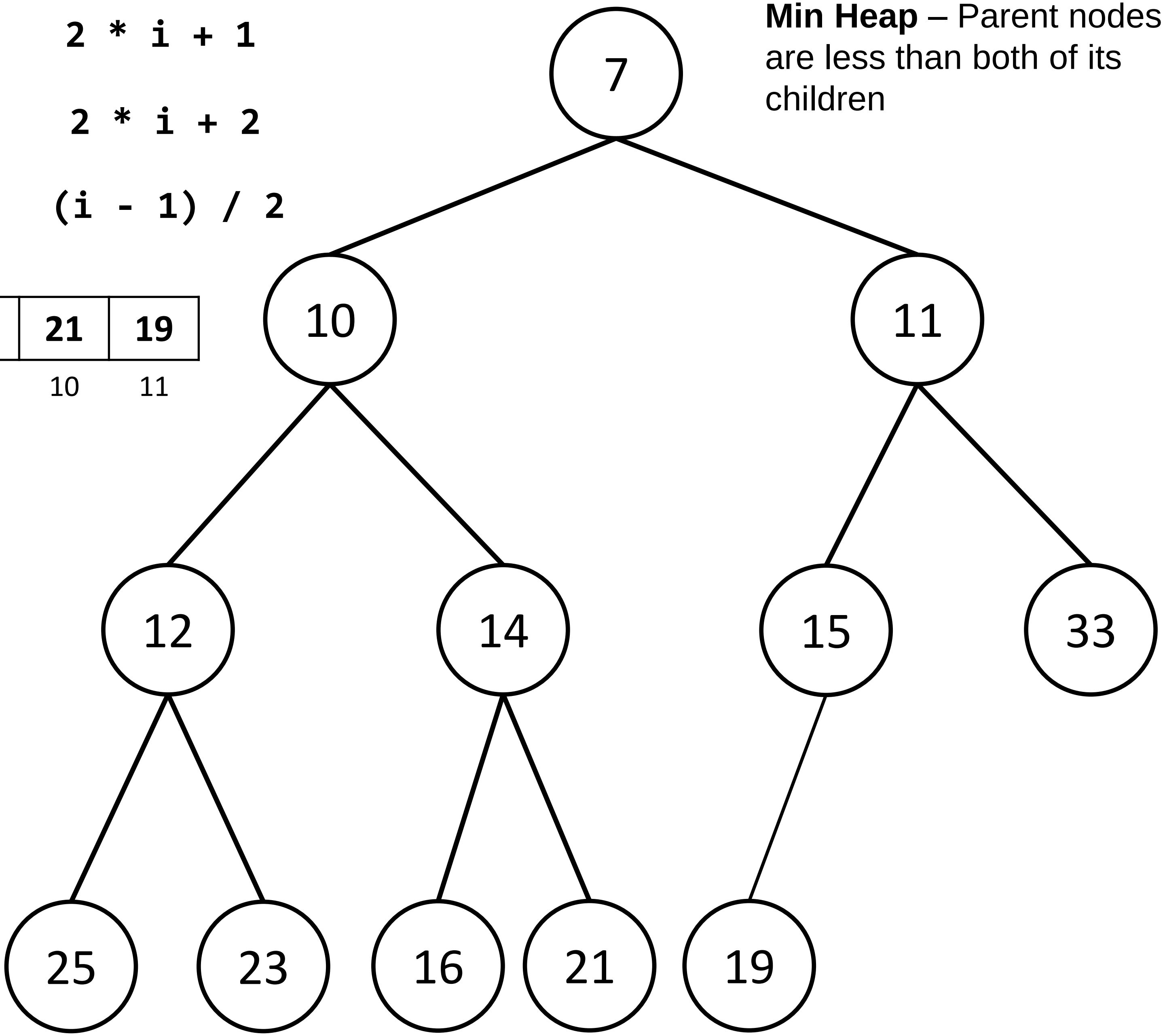
Parent `(i - 1) / 2`

Min Heap – Parent nodes are less than both of its children

Array

7	10	11	12	14	15	33	25	23	16	21	19
0	1	2	3	4	5	6	7	8	9	10	11

```
poll();
```



Heap Representation

Left Child `2 * i + 1`

Right Child `2 * i + 2`

Parent `(i - 1) / 2`

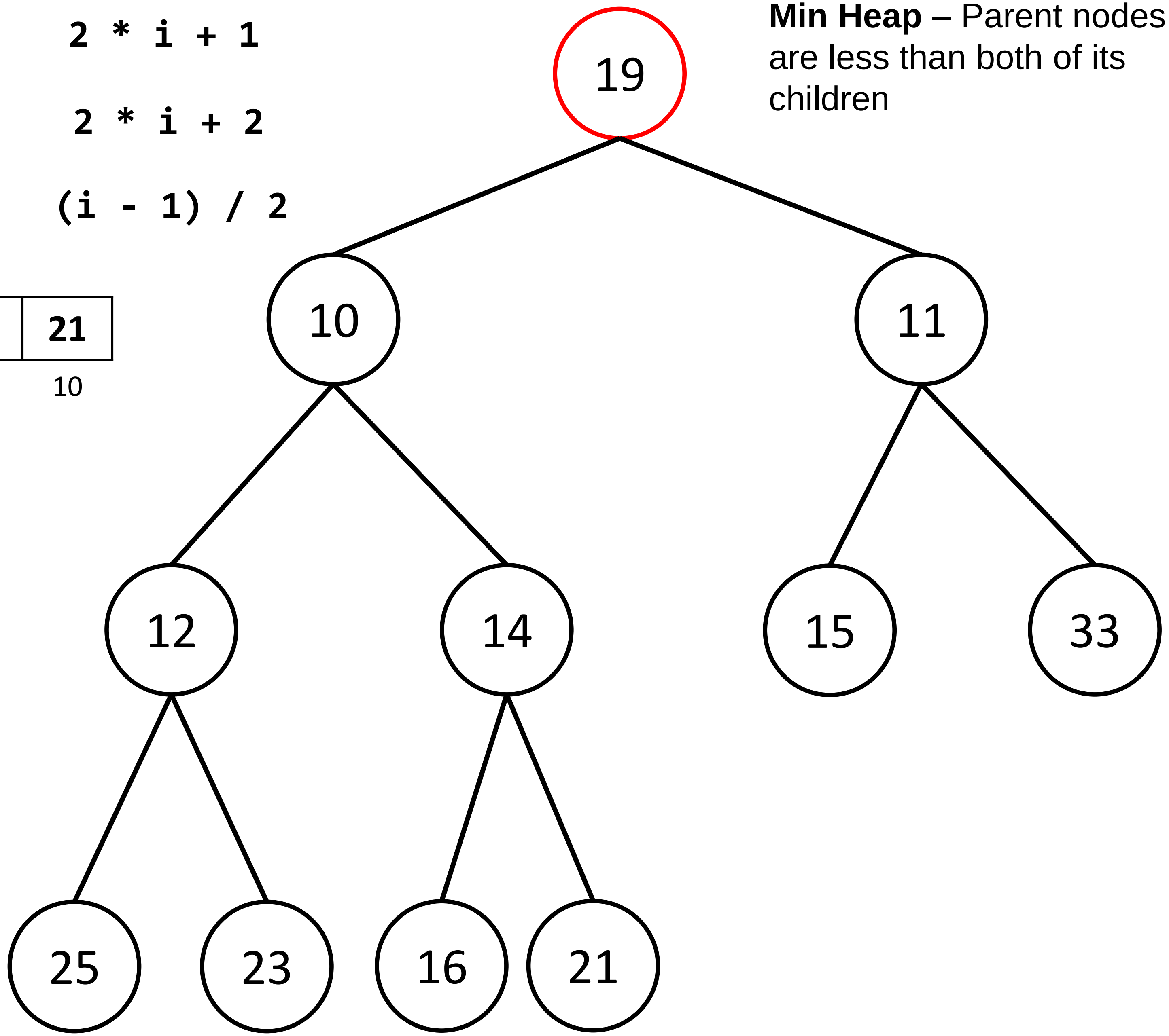
Array

O(1) time

19	**10**	**11**	**12**	**14**	**15**	**33**	**25**	**23**	**16**	**21**
0	1	2	3	4	5	6	7	8	9	10

```
poll();
```

Min Heap – Parent nodes are less than both of its children



Heap Representation

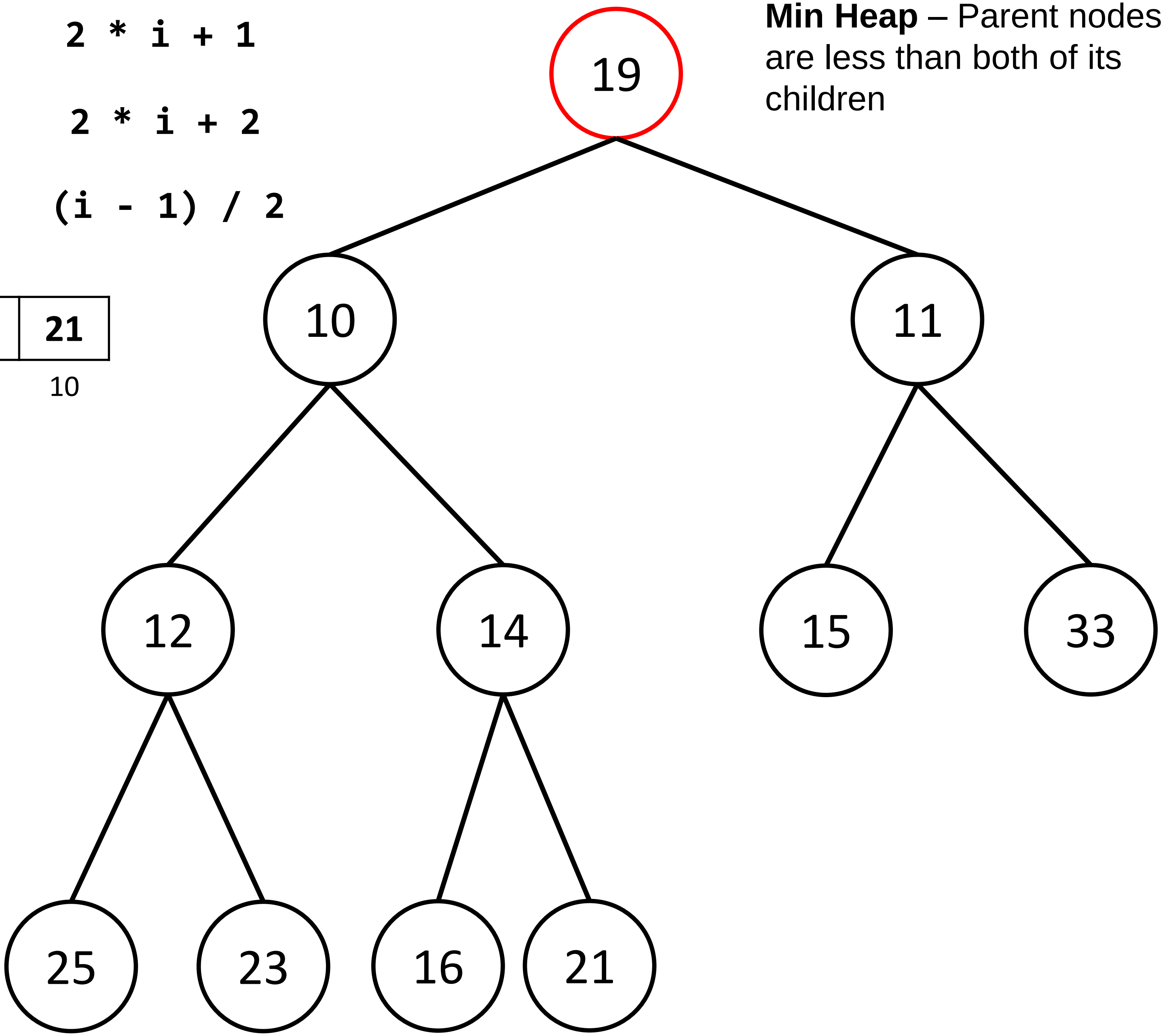
Left Child	`2 * i + 1`
Right Child	`2 * i + 2`
Parent	`(i - 1) / 2`

Array

19	**10**	**11**	**12**	**14**	**15**	**33**	**25**	**23**	**16**	**21**
0	1	2	3	4	5	6	7	8	9	10

```
poll();
```

Time to Heapify down!



Min Heap – Parent nodes are less than both of its children

Heap Representation

Left Child `2 * i + 1`

Right Child `2 * i + 2`

Parent `(i - 1) / 2`

Min Heap – Parent nodes are less than both of its children

Array

🟥 19	🟦 10	🟩 11	12	14	15	33	25	23	16	21
0	1	2	3	4	5	6	7	8	9	10

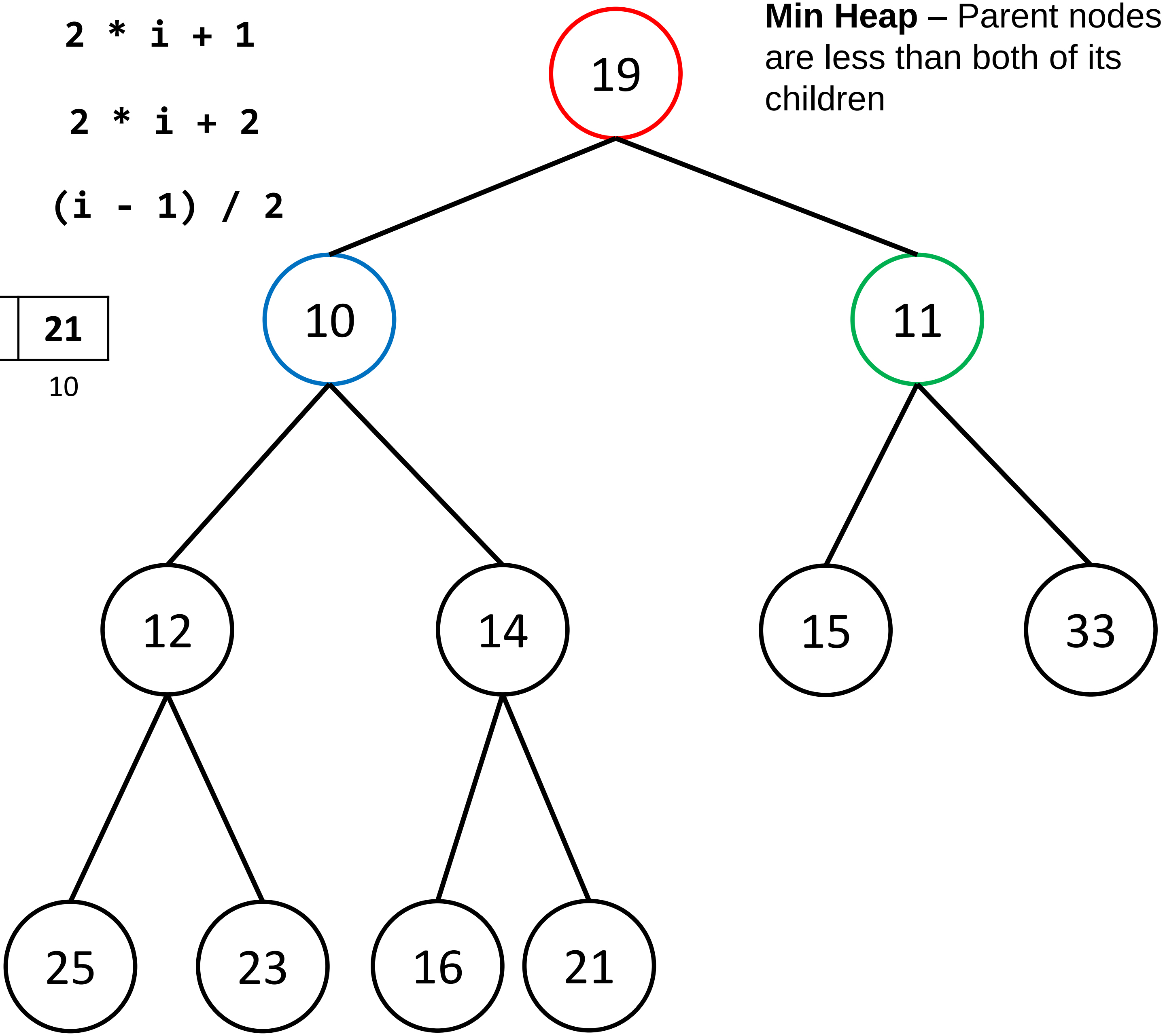
```
poll();
```

Time to Heapify down!

19’s left child is located at 2 * 0 + 1 = 1

19’s left child is located at 2 * 0 + 2 = 2

(We want to swap it with the lower value)



Heap Representation

Left Child `2 * i + 1`

Right Child `2 * i + 2`

Parent `(i - 1) / 2`

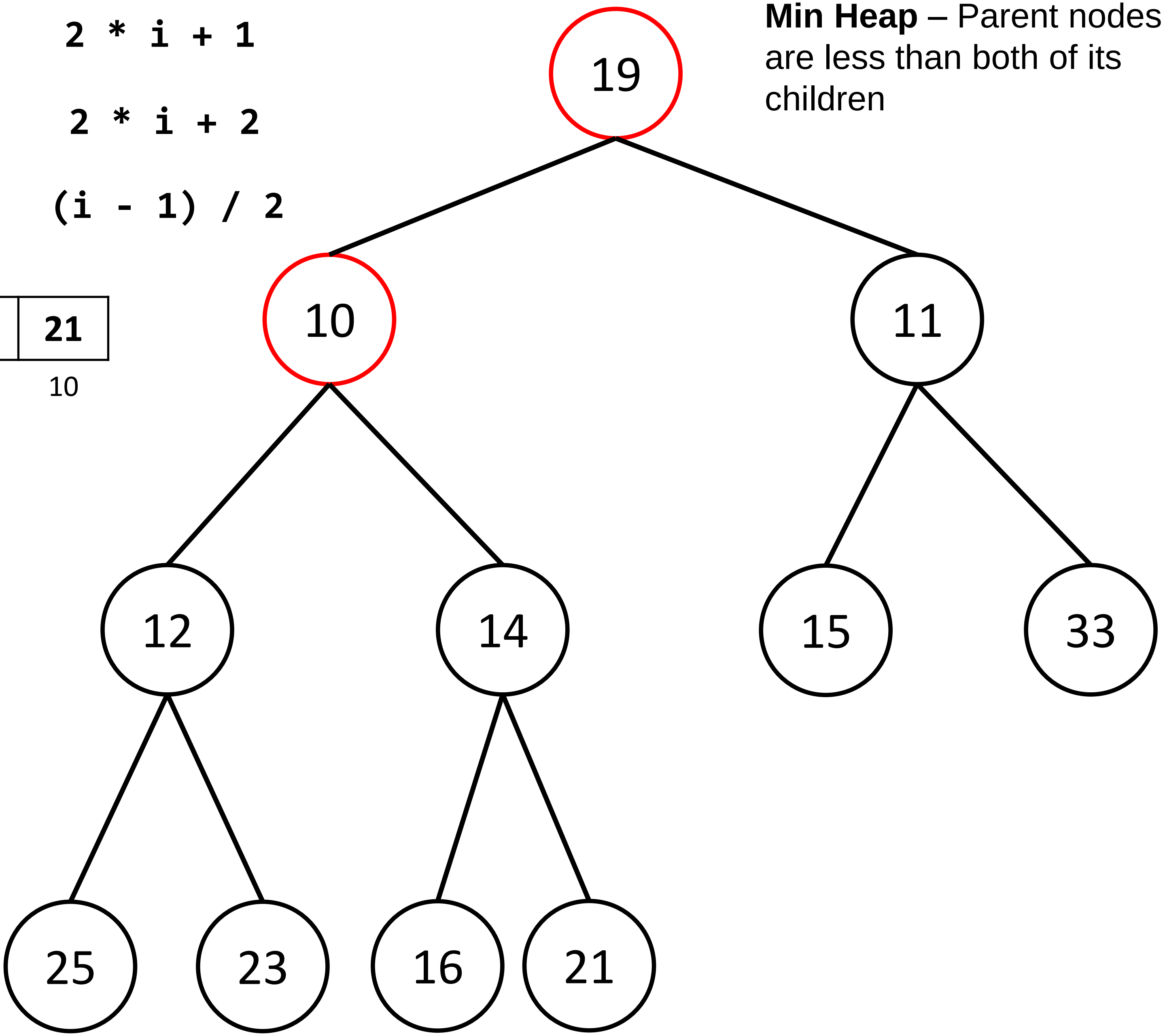
Array

19	10	11	12	14	15	33	25	23	16	21
0	1	2	3	4	5	6	7	8	9	10

```
poll();
```

Time to Heapify down!

Min Heap – Parent nodes are less than both of its children



Heap Representation

Left Child `2 * i + 1`

Right Child `2 * i + 2`

Parent `(i - 1) / 2`

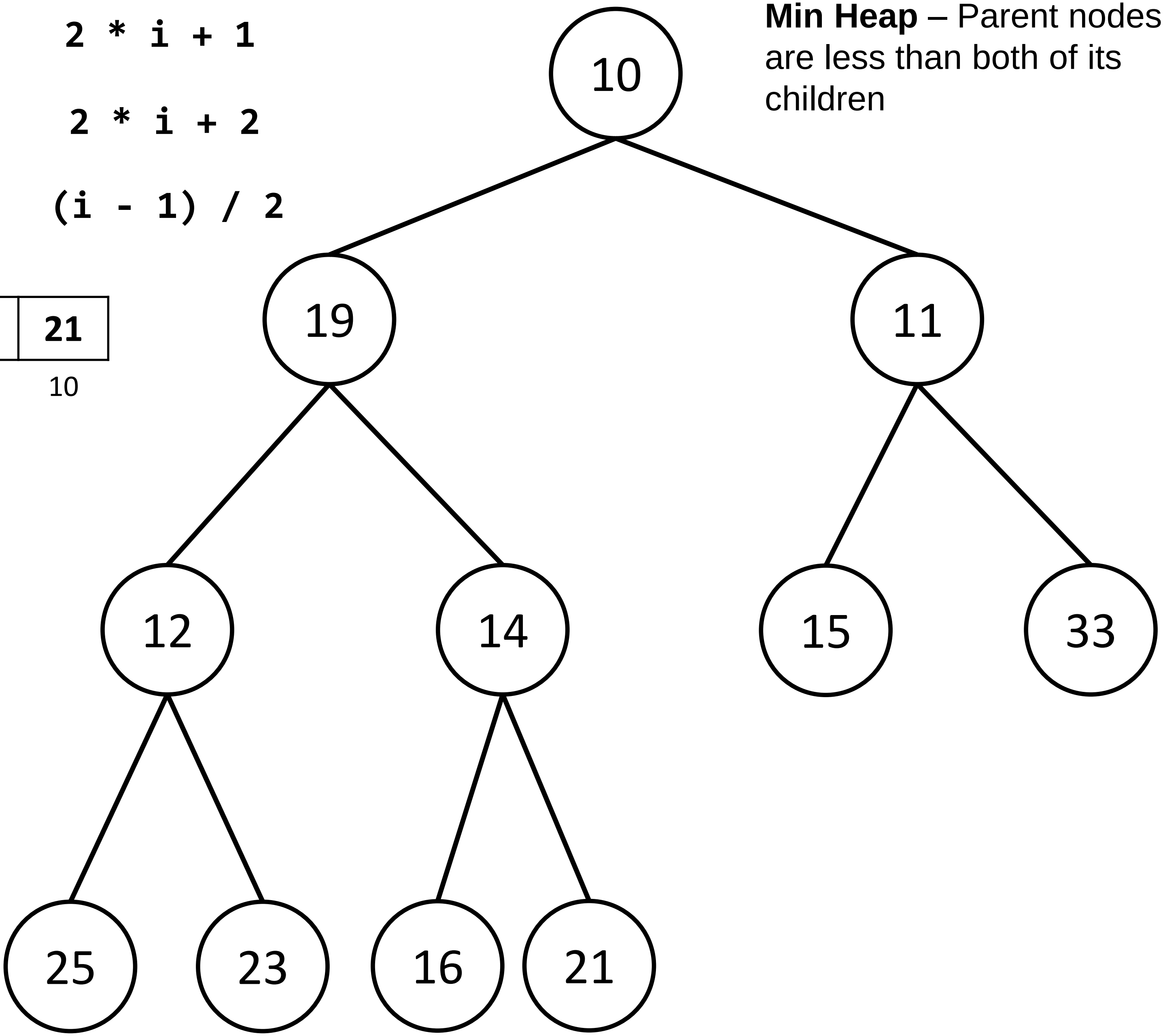
Array

10	19	11	12	14	15	33	25	23	16	21
0	1	2	3	4	5	6	7	8	9	10

```
poll();
```

Time to Heapify down!

Min Heap – Parent nodes are less than both of its children



Heap Representation

Left Child	`2 * i + 1`
Right Child	`2 * i + 2`
Parent	`(i - 1) / 2`

Array

10	**19**	**11**	**12**	**14**	**15**	**33**	**25**	**23**	**16**	**21**
0	1	2	3	4	5	6	7	8	9	10

```
poll();
```

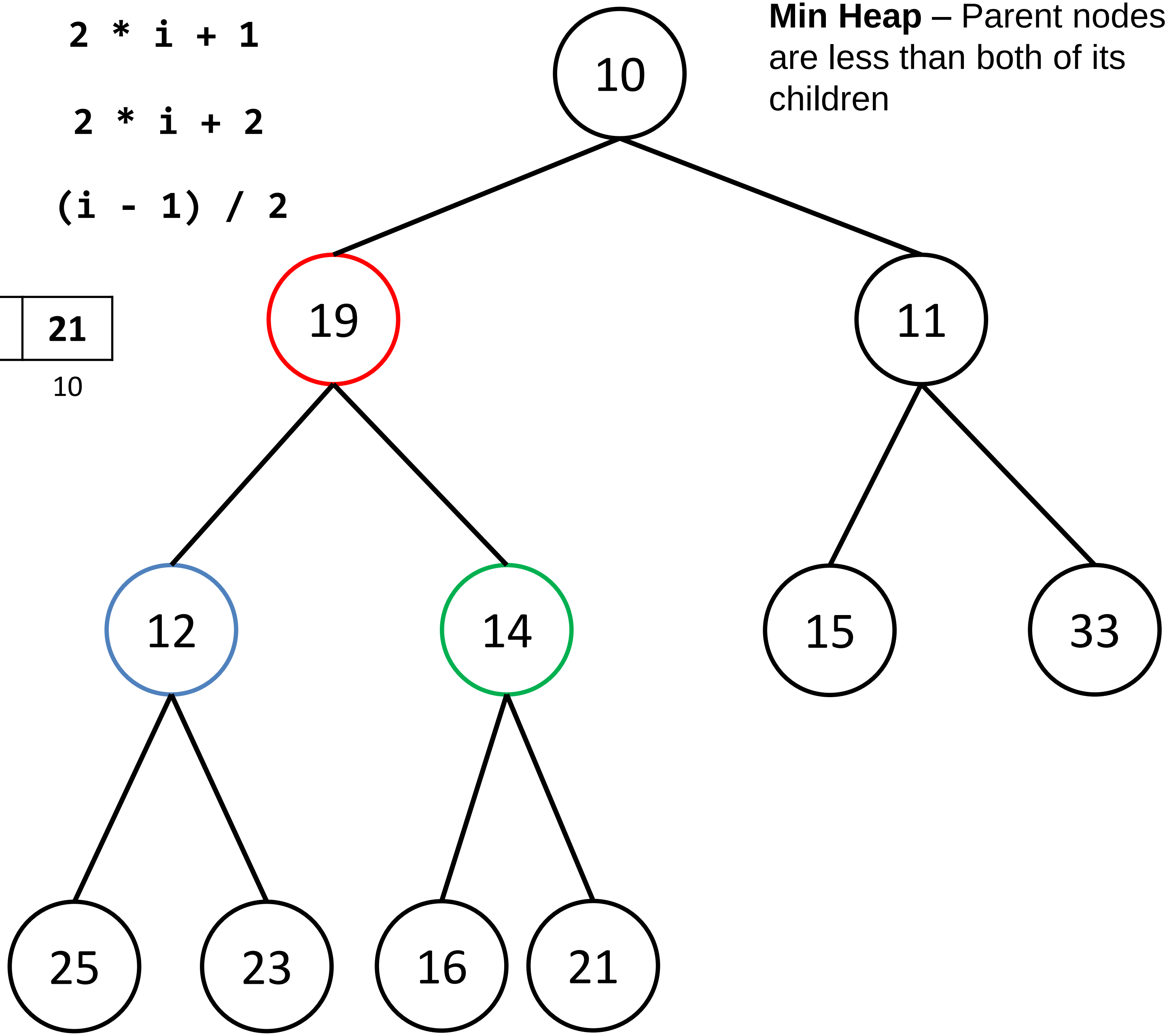
Time to Heapify down!

19’s left child is located at 2 * 1 + 1 = **3**

19’s left child is located at 2 * 1 + 2 = **4**

(We want to swap it with the lower value)

Min Heap – Parent nodes are less than both of its children



Heap Representation

Left Child `2 * i + 1`

Right Child `2 * i + 2`

Parent `(i - 1) / 2`

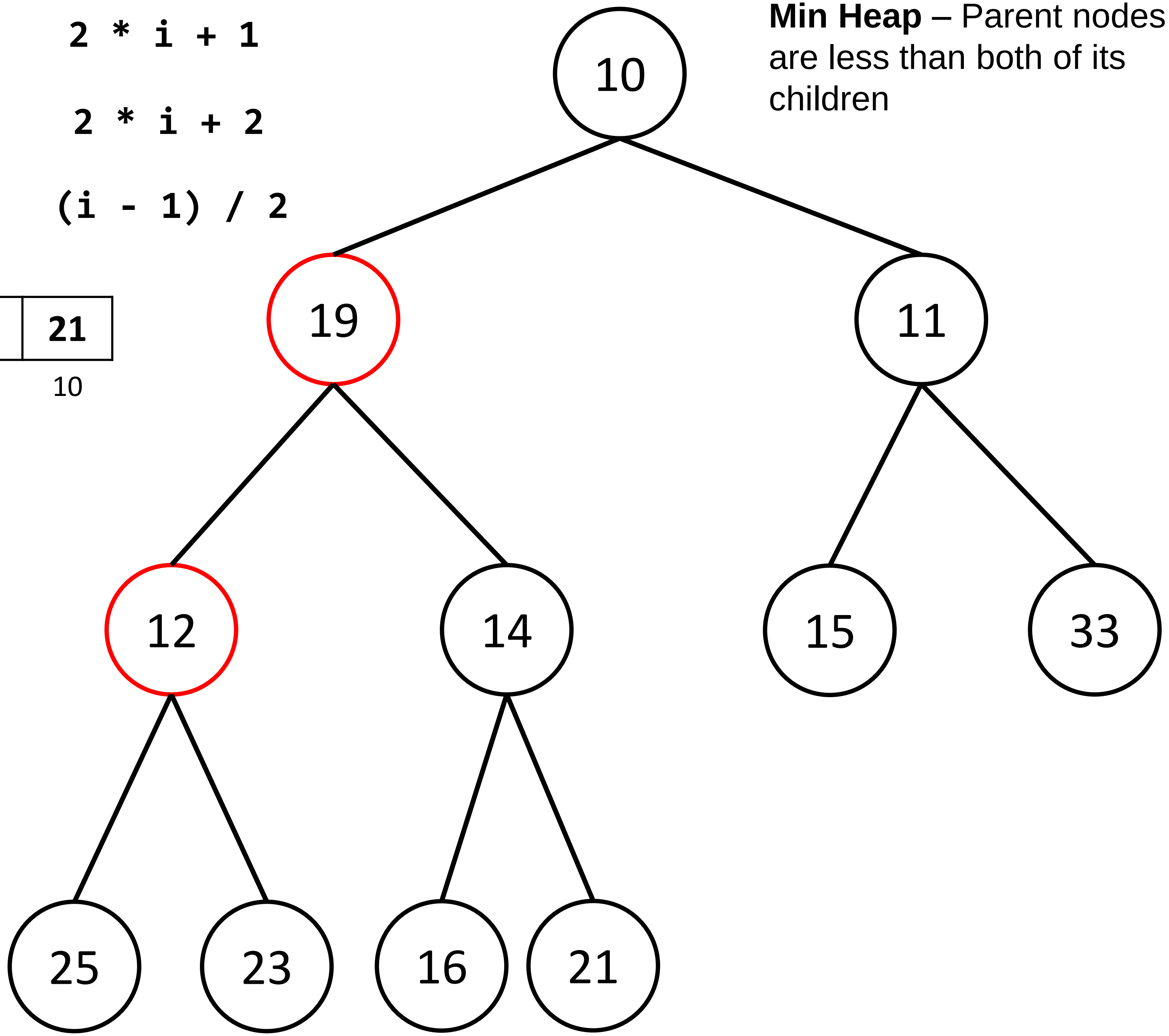
Min Heap – Parent nodes are less than both of its children

Array

10	**19**	**11**	**12**	**14**	**15**	**33**	**25**	**23**	**16**	**21**
0	1	2	3	4	5	6	7	8	9	10

```
poll();
```

Time to Heapify down!



Heap Representation

Left Child `2 * i + 1`

Right Child `2 * i + 2`

Parent `(i - 1) / 2`

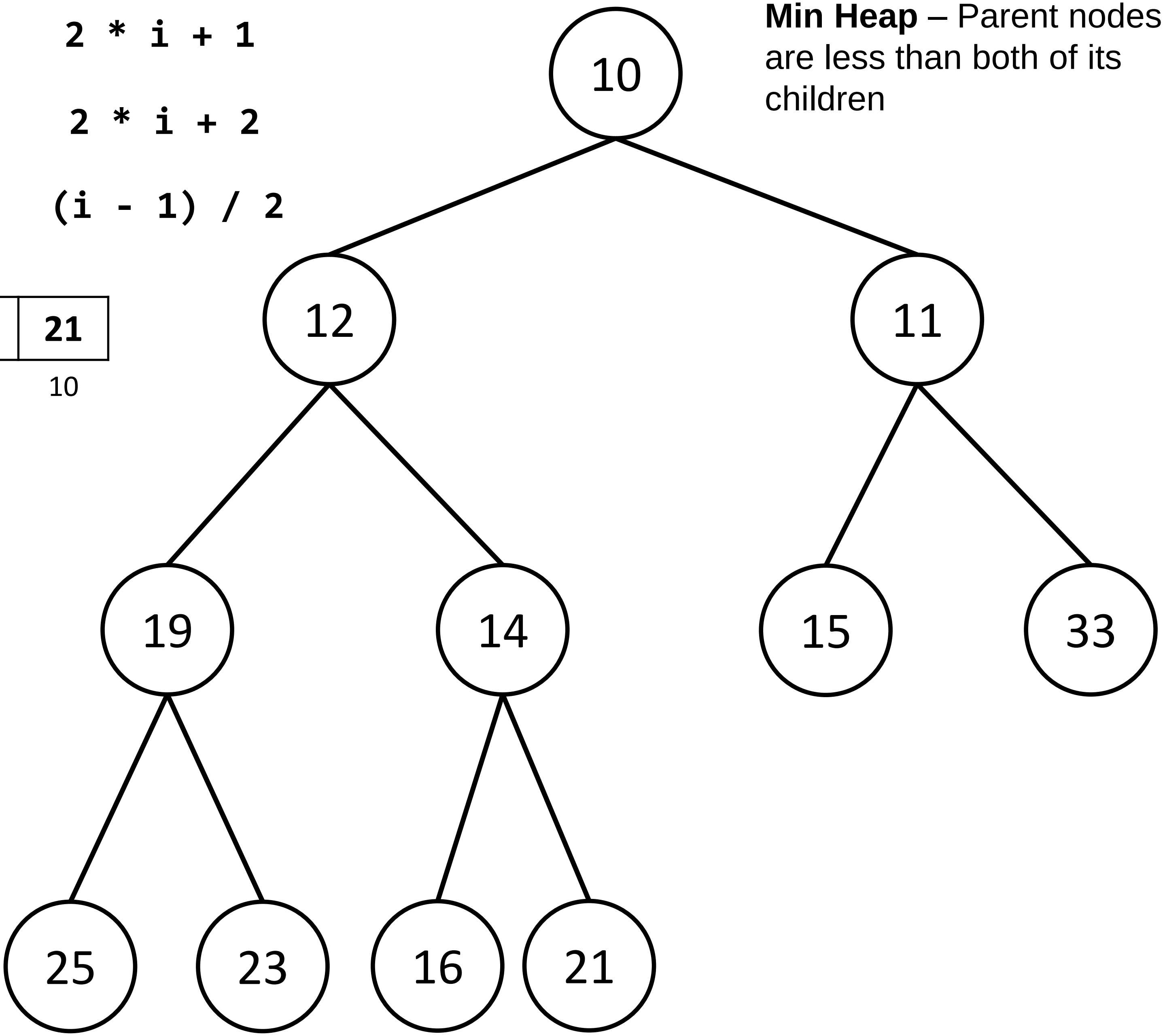
Min Heap – Parent nodes are less than both of its children

Array

10	12	11	19	14	15	33	25	23	16	21
0	1	2	3	4	5	6	7	8	9	10

```
poll();
```

Time to Heapify down!



Heap Representation

Left Child `2 * i + 1`

Right Child `2 * i + 2`

Parent `(i - 1) / 2`

Array

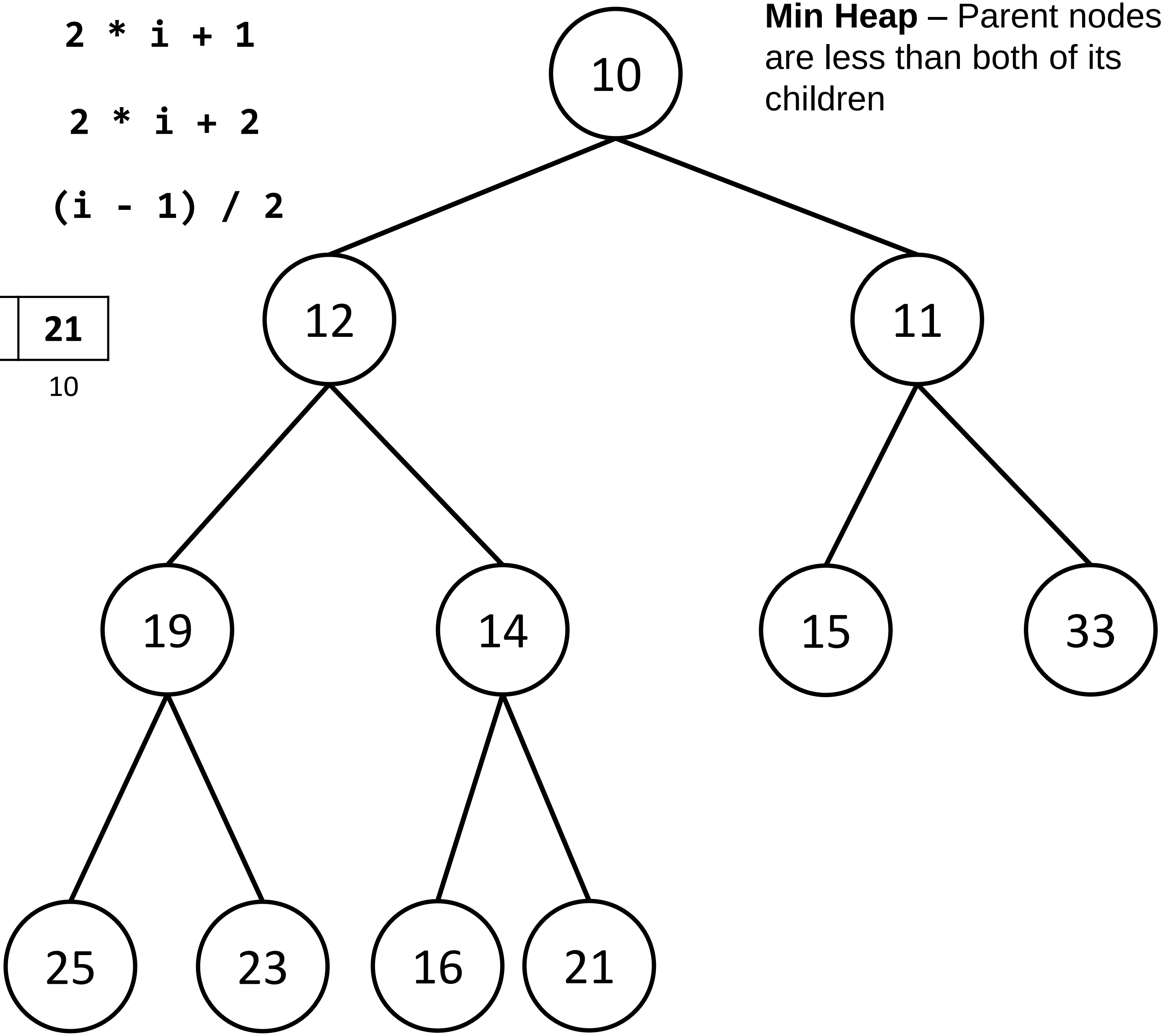
10	**12**	**11**	**19**	**14**	**15**	**33**	**25**	**23**	**16**	**21**
0	1	2	3	4	5	6	7	8	9	10

```
poll();
```

Time to Heapify down!

👍

Min Heap – Parent nodes are less than both of its children



Heap Representation

Left Child `2 * i + 1`

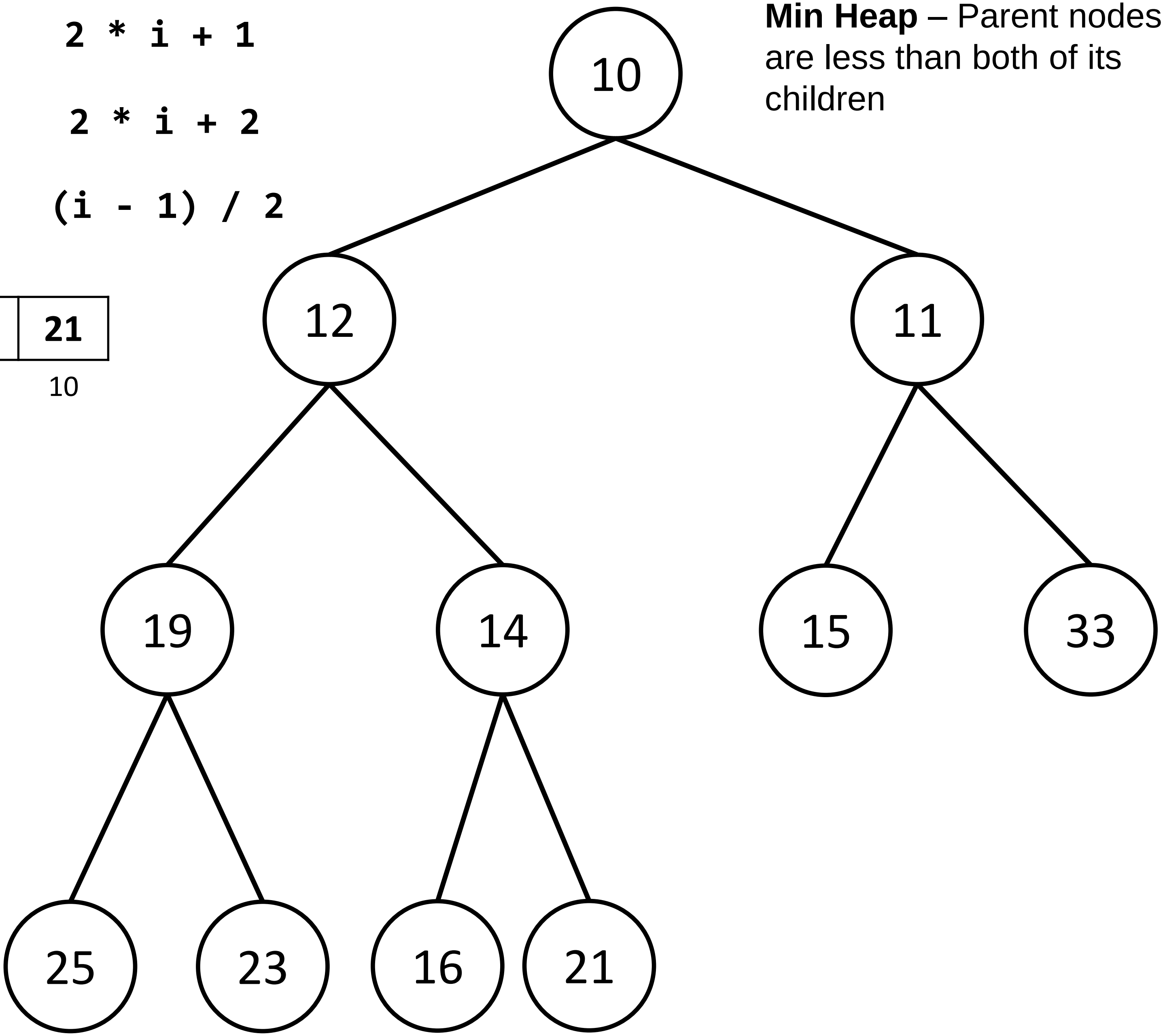
Right Child `2 * i + 2`

Parent `(i - 1) / 2`

Min Heap – Parent nodes are less than both of its children

Array

10	**12**	**11**	**19**	**14**	**15**	**33**	**25**	**23**	**16**	**21**
0	1	2	3	4	5	6	7	8	9	10

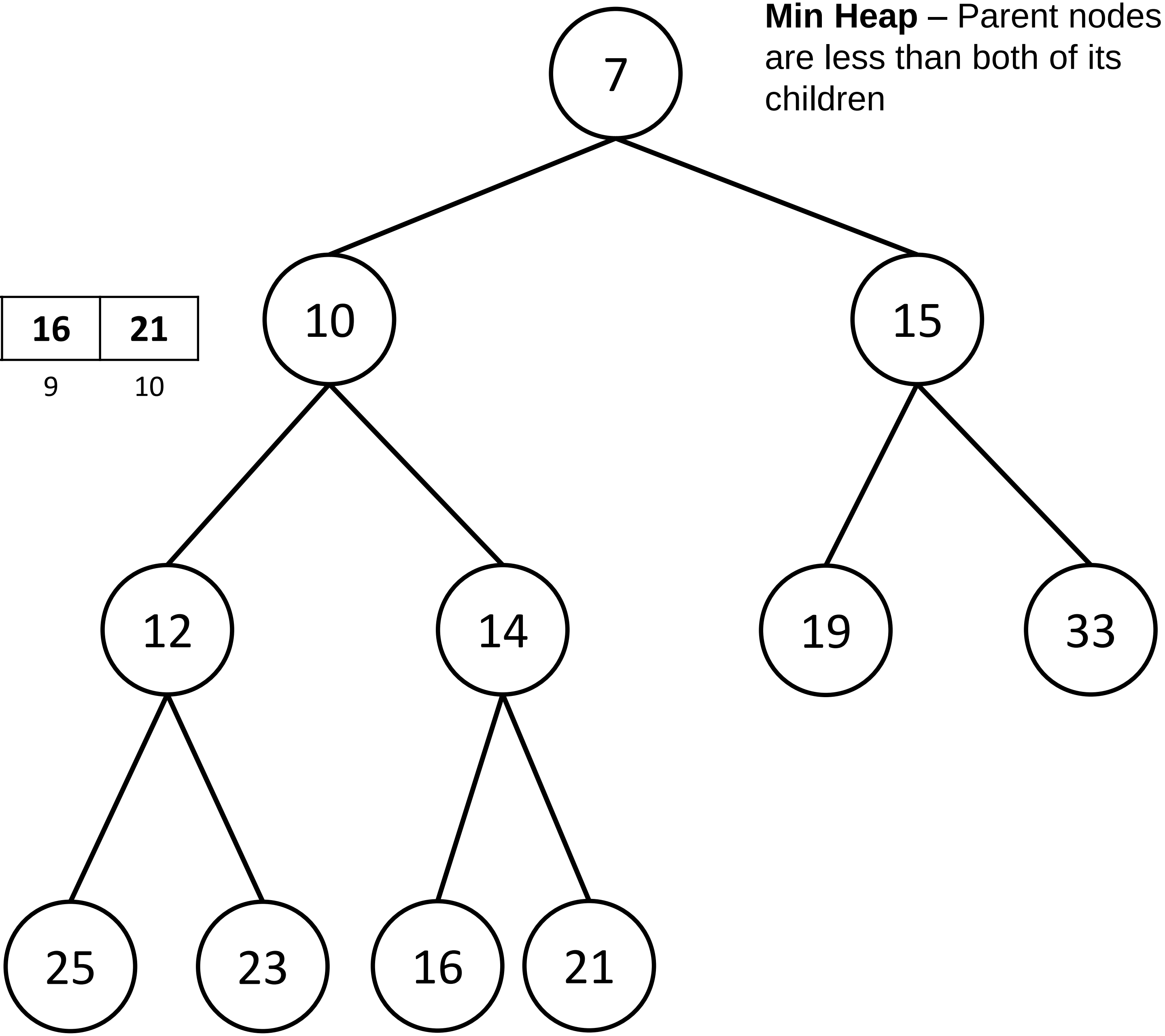


Let’s code this!!!

Min Heap – Parent nodes are less than both of its children

Array

7	**10**	**15**	**12**	**14**	**19**	**33**	**25**	**23**	**16**	**21**
0	1	2	3	4	5	6	7	8	9	10



What can a Heap do well that other data structures cannot as well?

What can a Heap do well that other data structures cannot as well?

Finding the largest/smallest element happens in **O(1)** time

Because we use an array, it might be more memory efficient than a standard tree

Does a Heap remind you of any other data structures?

Does a Heap remind you of any other data structures?

Priority Queue

Does a Heap remind you of any other data structures?

Priority Queue

Whenever we remove an element, we always remove the smallest/largest value (**`poll()`**)

Whenever we add an element, it initially gets added to the back of the array, and then swaps itself within the array

Takeaways

A Heap **is** a priority queue

Whenever we remove an element, we always remove the smallest/largest value (**`poll()`**)

Whenever we add an element, it initially gets added to the back of the array, and then swaps itself within the array

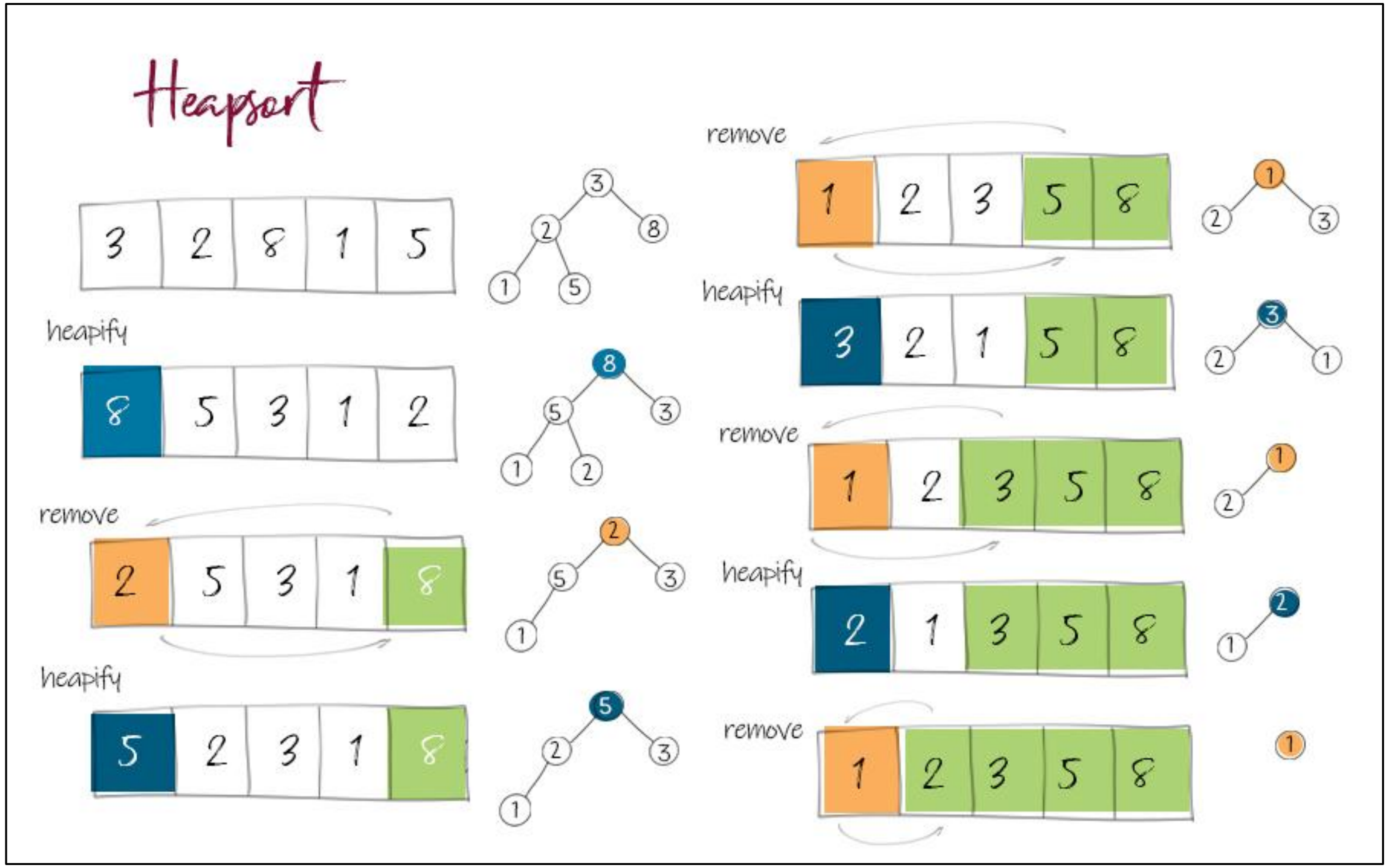
Getting the maximum/minimum value happens in O(1) time

Class PriorityQueue<E>

There is a section of memory in your computer called "The Heap", which is something totally unrelated to this data structure

Applications

Heapsort- Sorting algorithm that converts an unsorted array to a Heap, and then repeatedly remove the root node

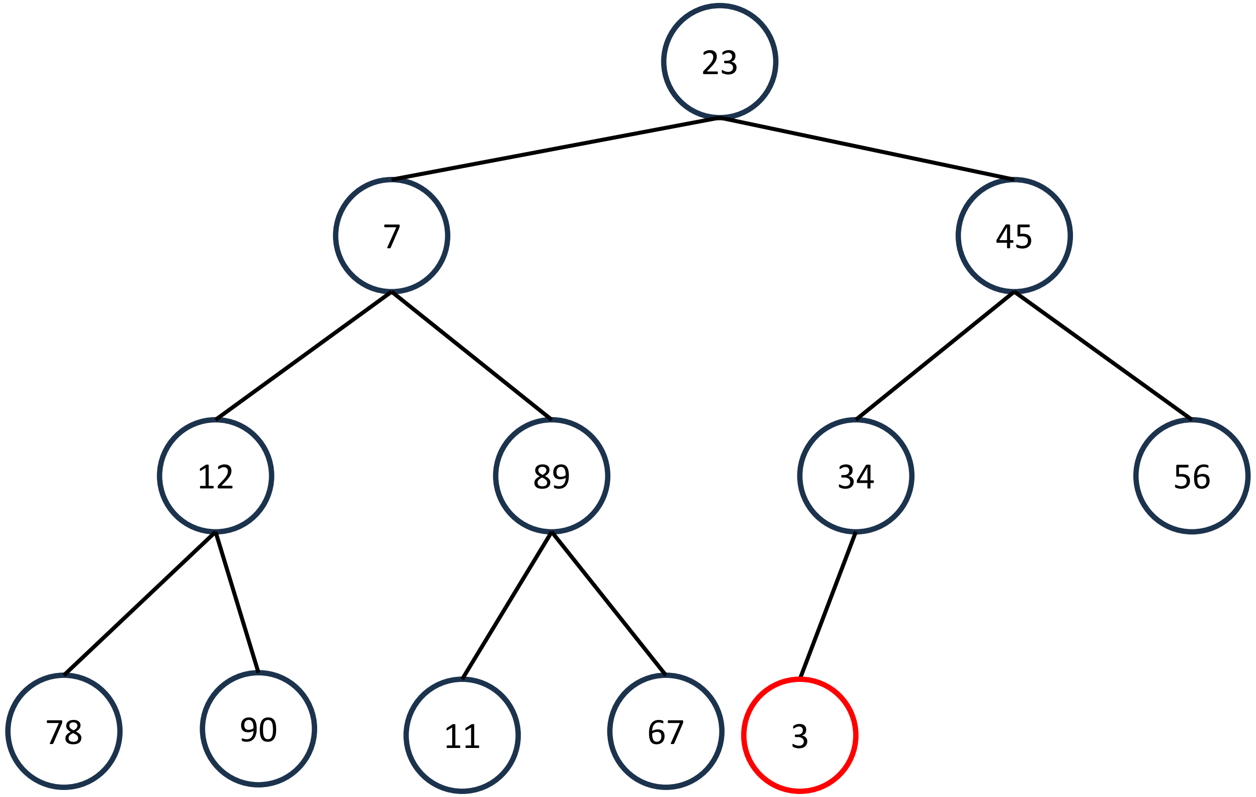


Heap Sort

```
int[] data = {23, 7, 45, 12, 89, 34, 56, 78, 90, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

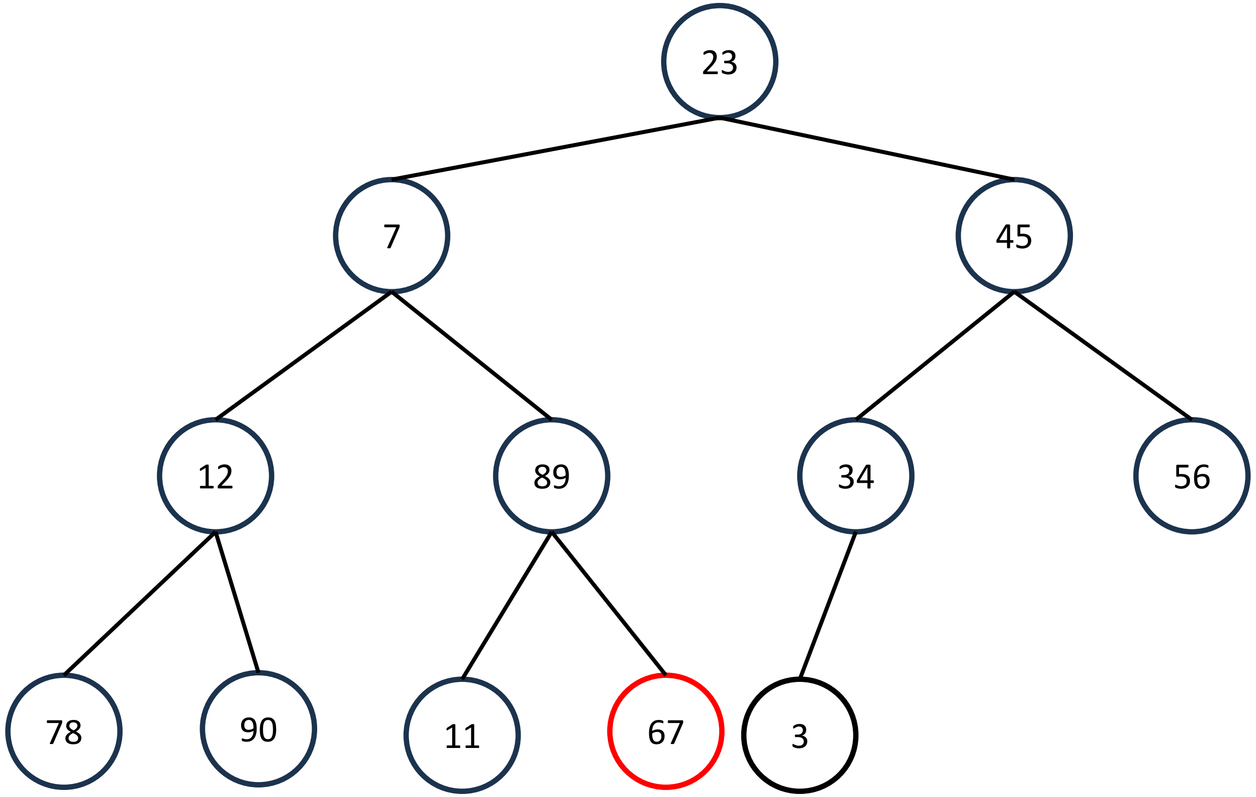


Heap Sort

```
int[] data = {23, 7, 45, 12, 89, 34, 56, 78, 90, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

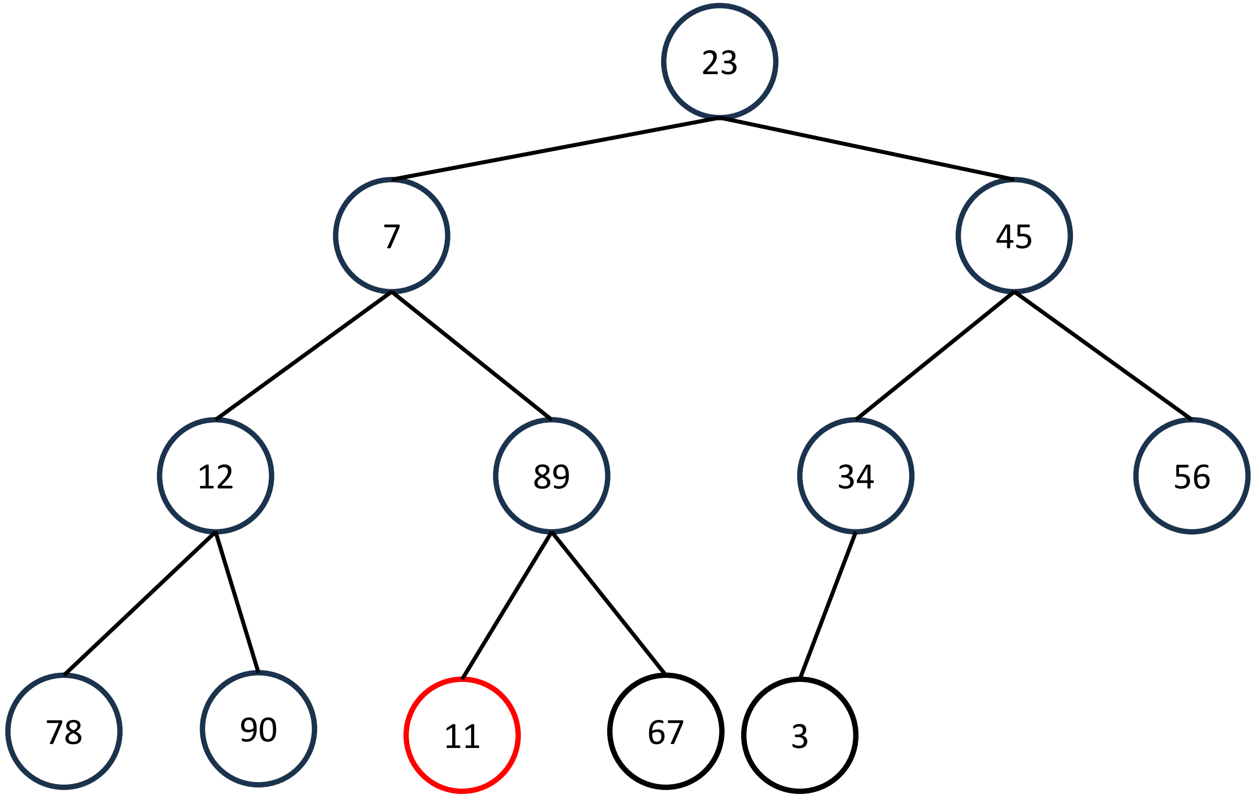


Heap Sort

```
int[] data = {23, 7, 45, 12, 89, 34, 56, 78, 90, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

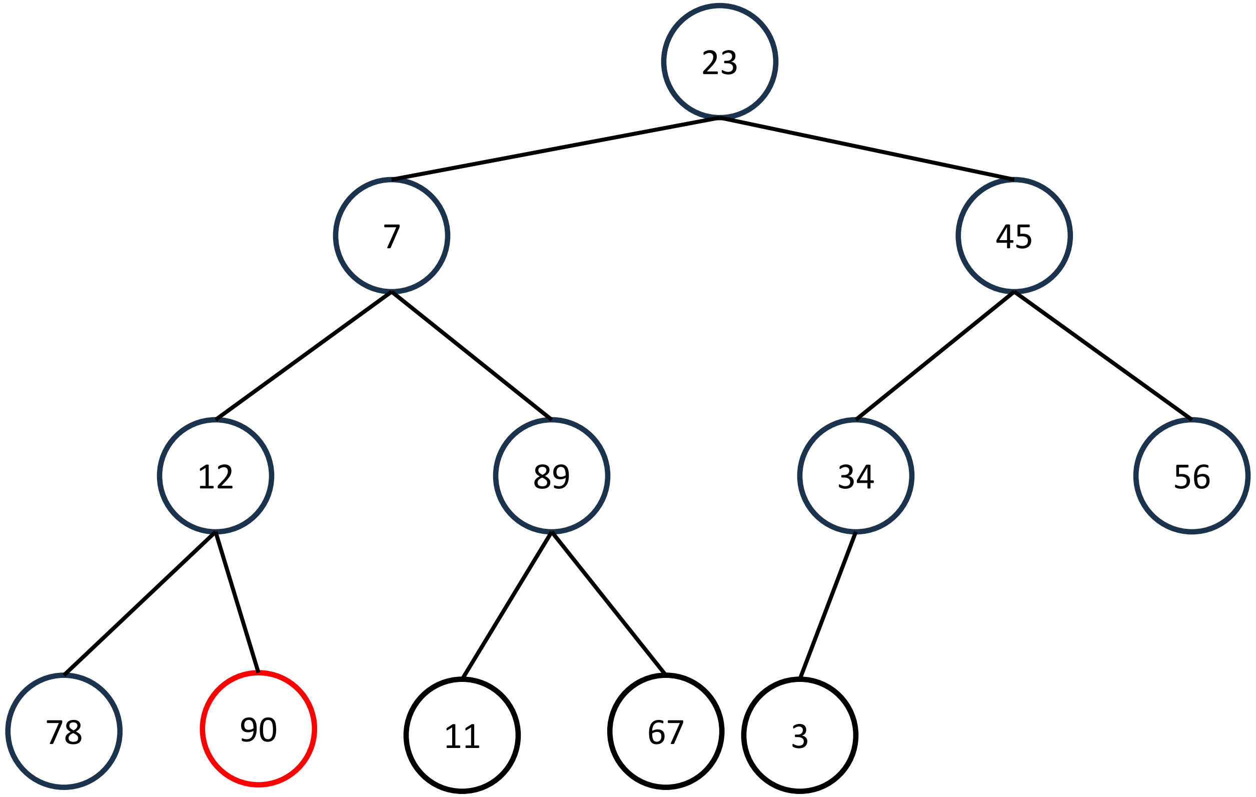


Heap Sort

```
int[] data = {23, 7, 45, 12, 89, 34, 56, 78, 90, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

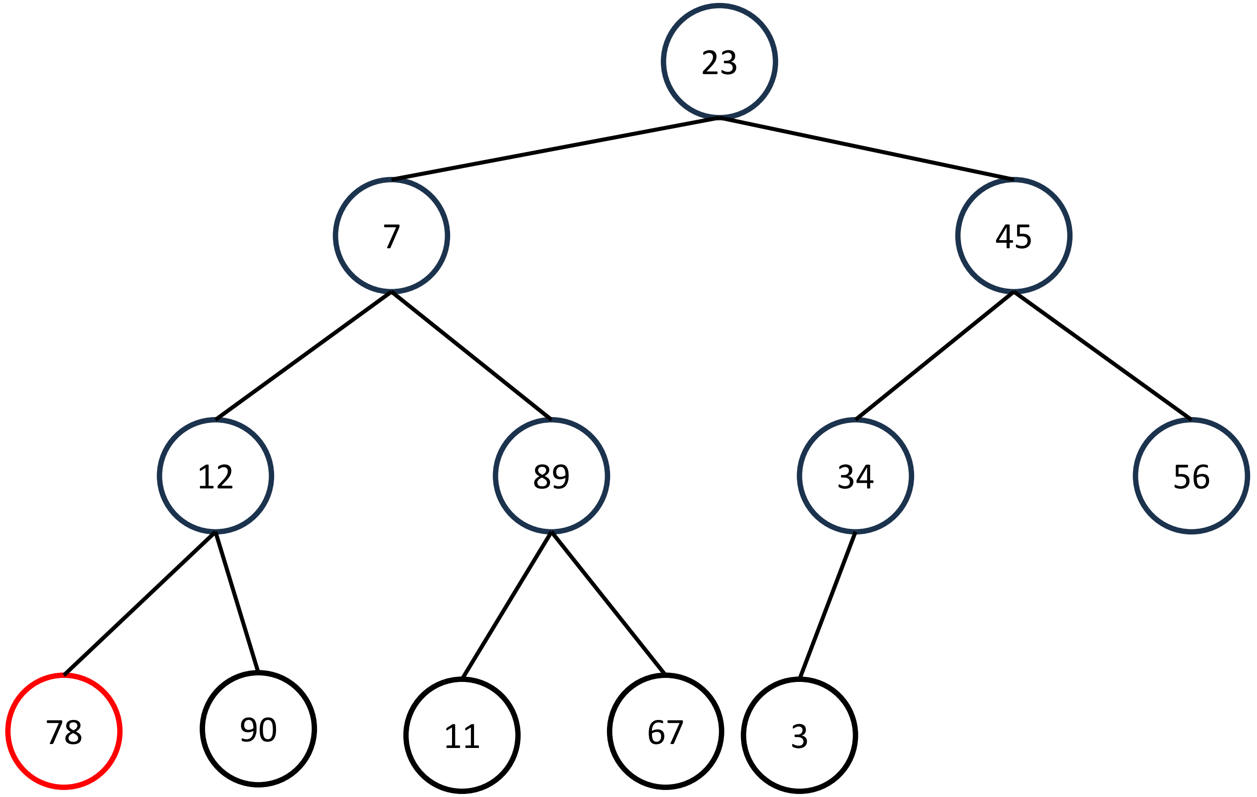


Heap Sort

```
int[] data = {23, 7, 45, 12, 89, 34, 56, 78, 90, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

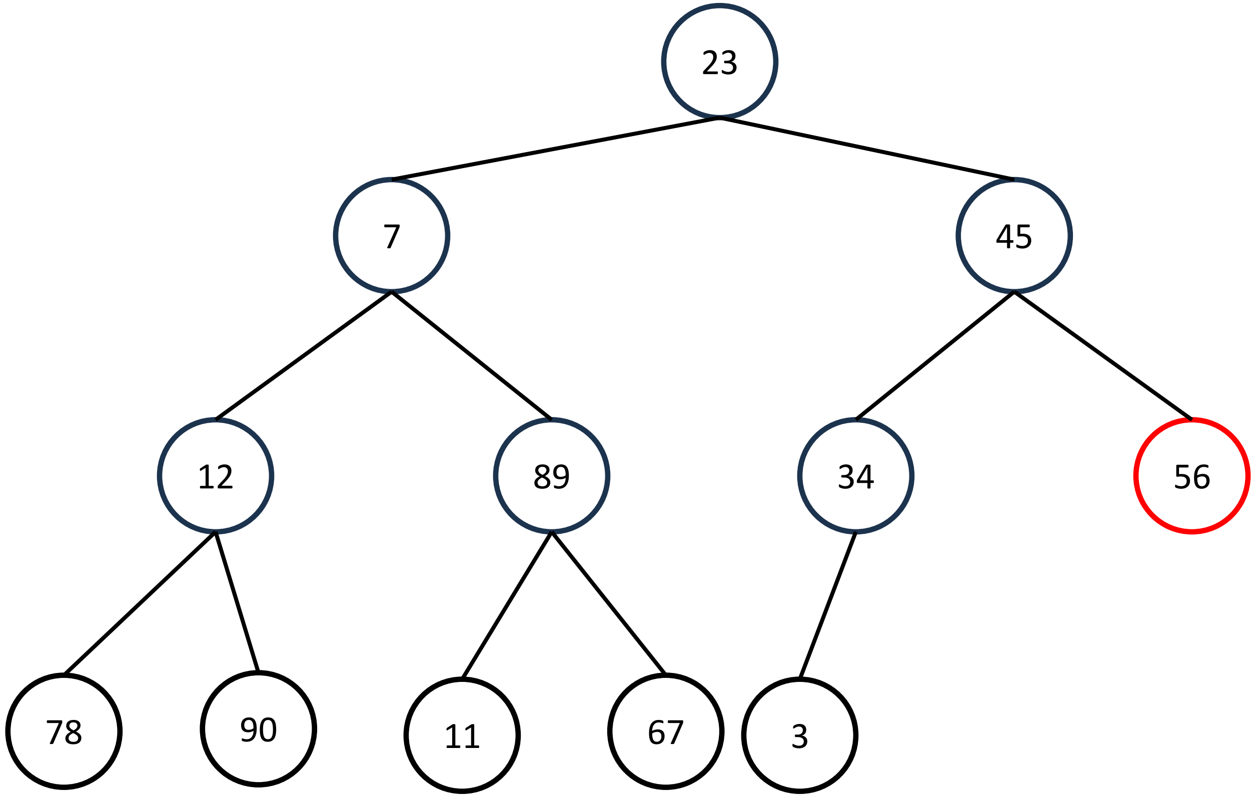


Heap Sort

```
int[] data = {23, 7, 45, 12, 89, 34, 56, 78, 90, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

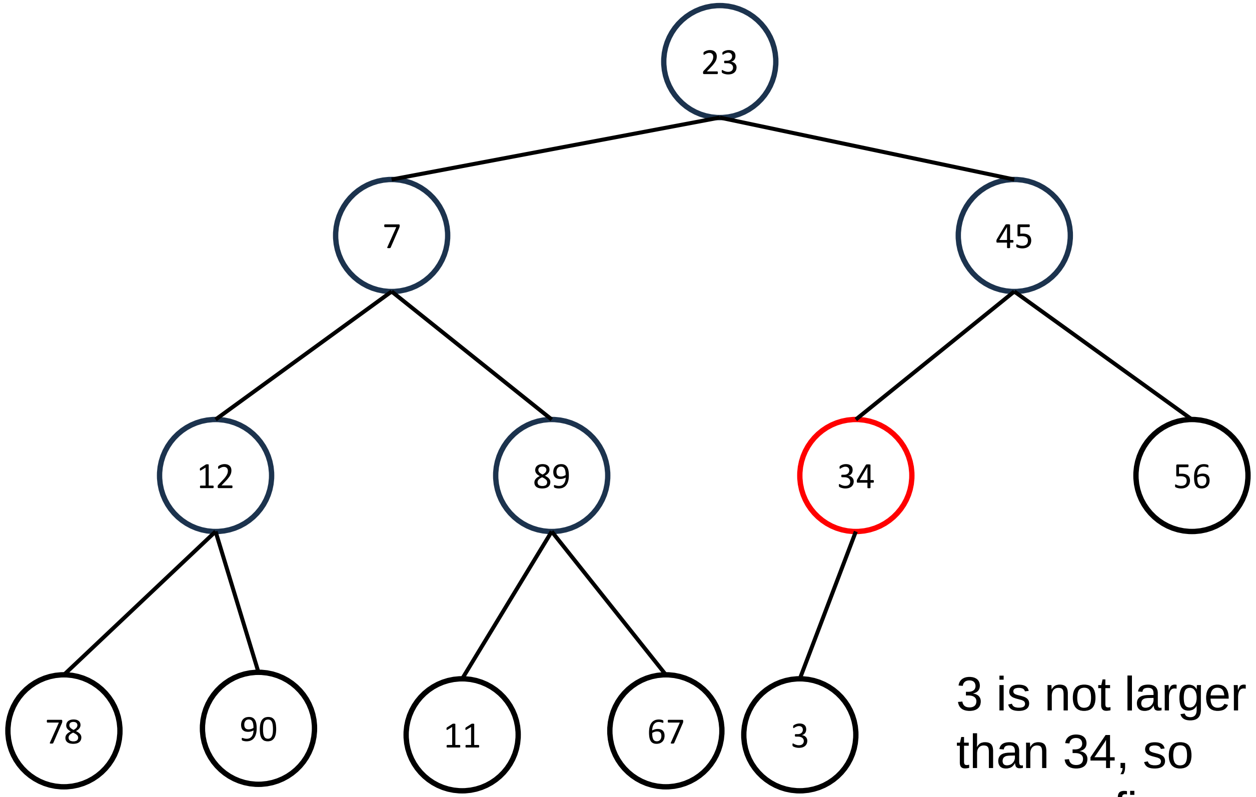


Heap Sort

```
int[] data = {23, 7, 45, 12, 89, 34, 56, 78, 90, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger



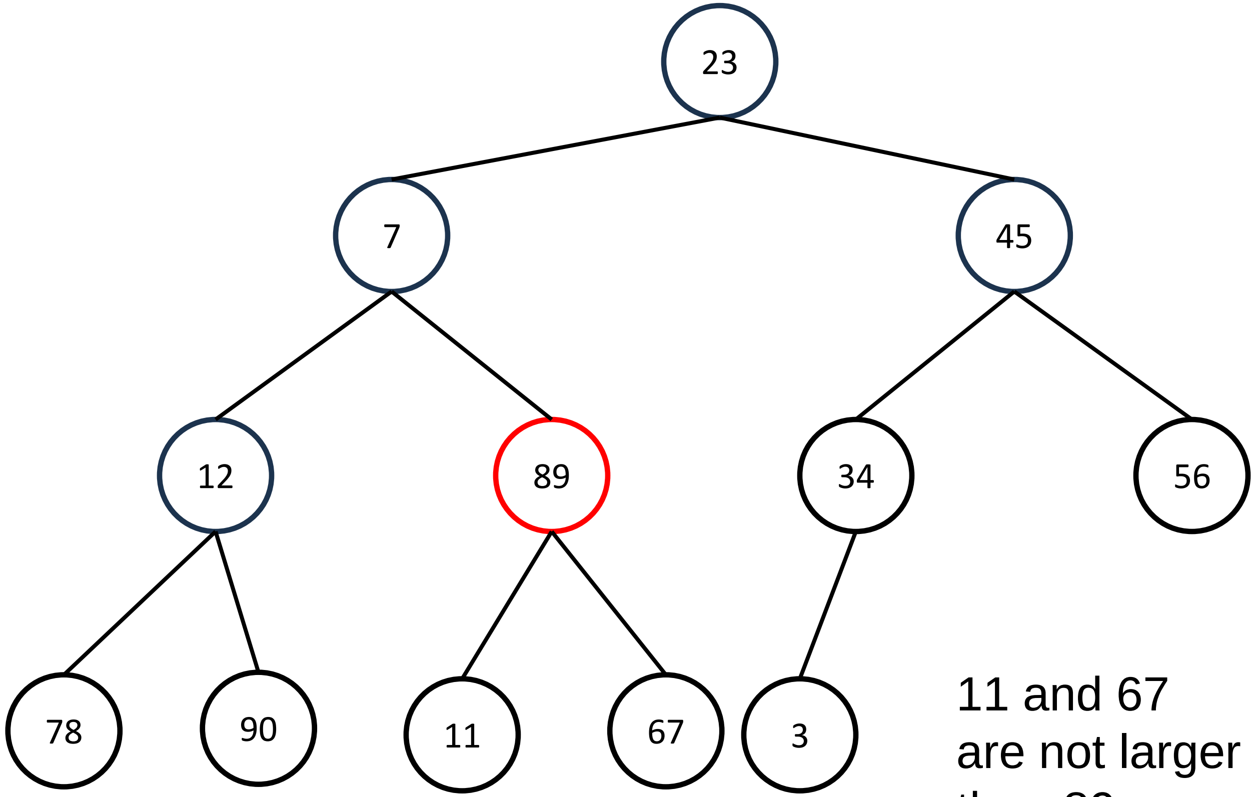
3 is not larger than 34, so we are fine

Heap Sort

```
int[] data = {23, 7, 45, 12, 89, 34, 56, 78, 90, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger



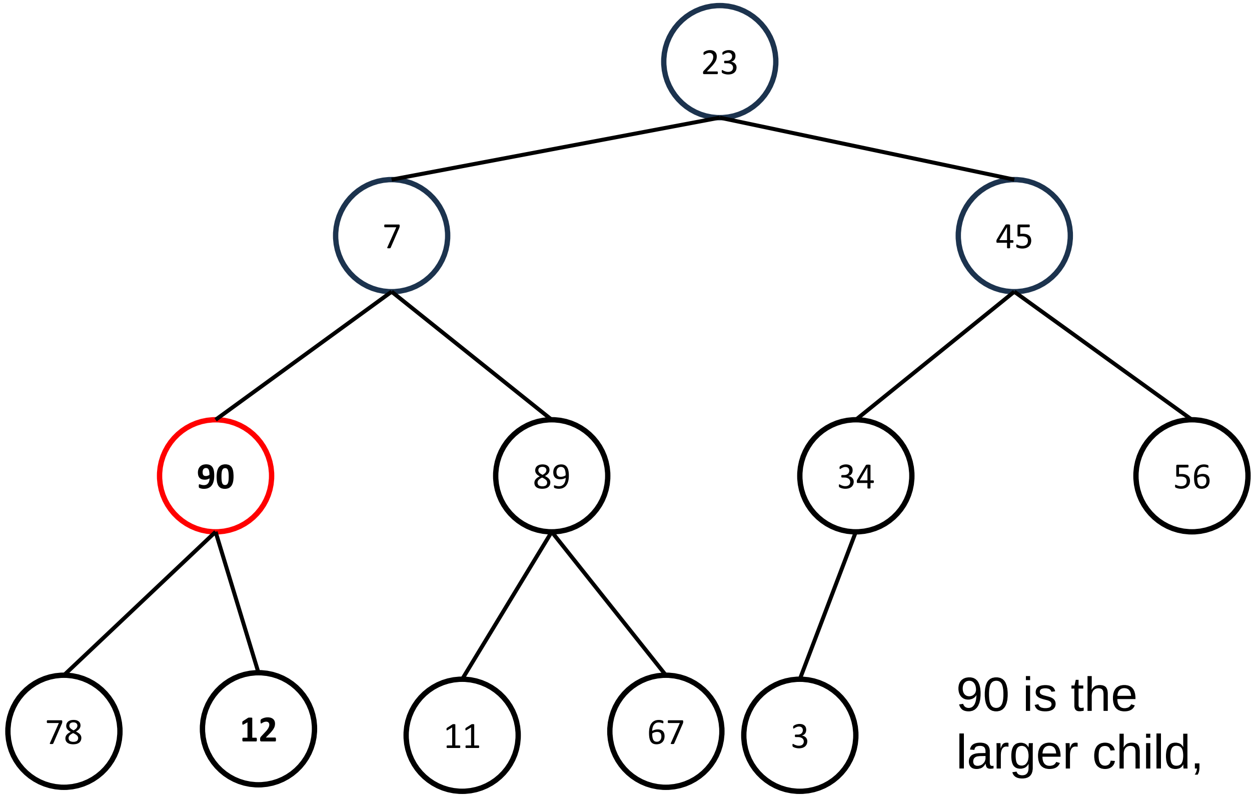
11 and 67 are not larger than 89, so continue

Heap Sort

```
int[] data = {23, 7, 45, 90, 89, 34, 56, 78, 12, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger



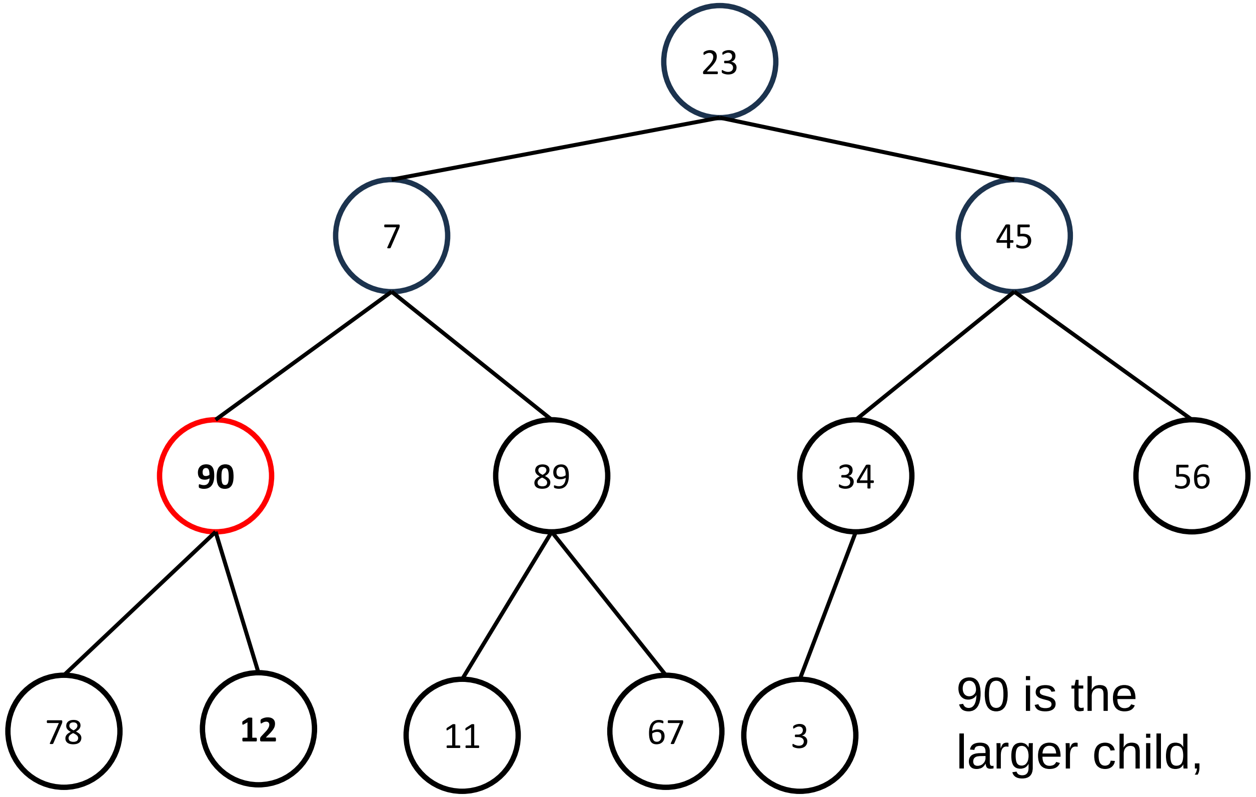
90 is the larger child, so swap

Heap Sort

```
int[] data = {23, 7, 45, 90, 89, 34, 56, 78, 12, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger



90 is the larger child, so swap

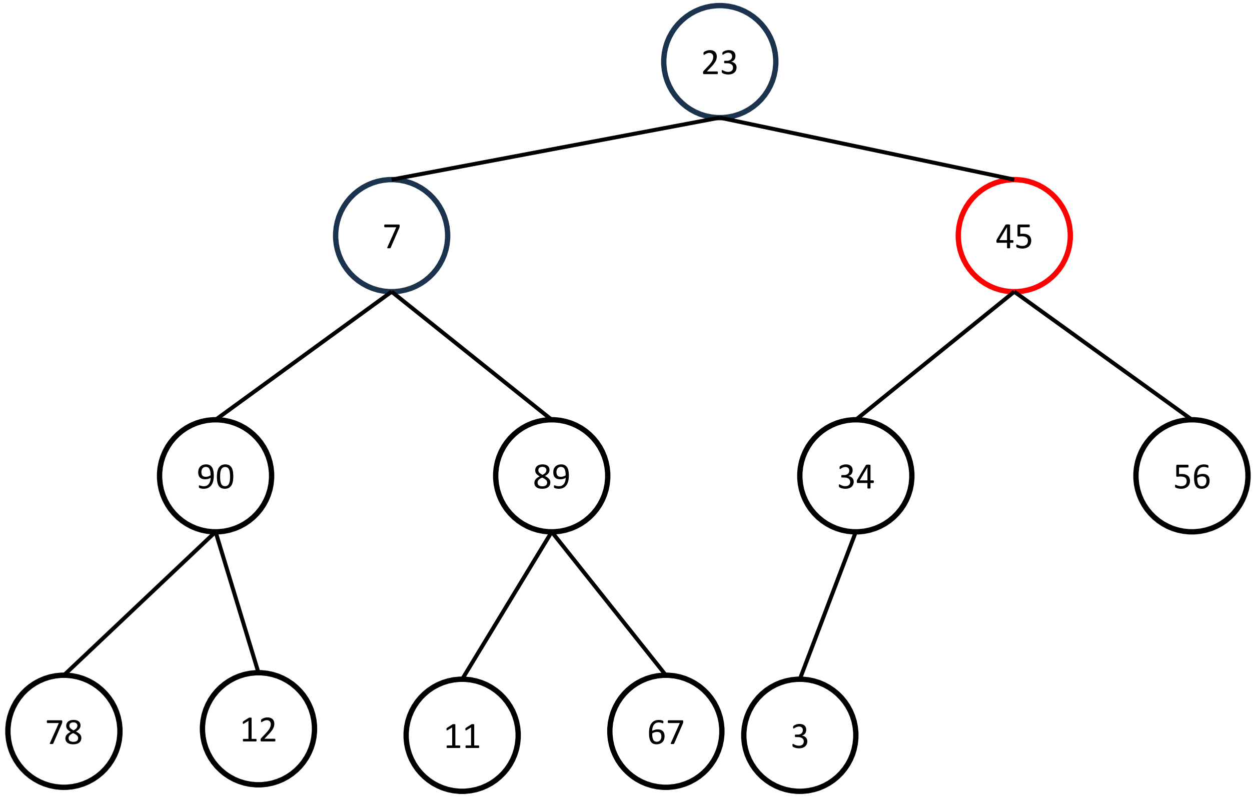
90 is larger than 78 and 12 so continue

Heap Sort

```
int[] data = {23, 7, 45, 90, 89, 34, 56, 78, 12, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

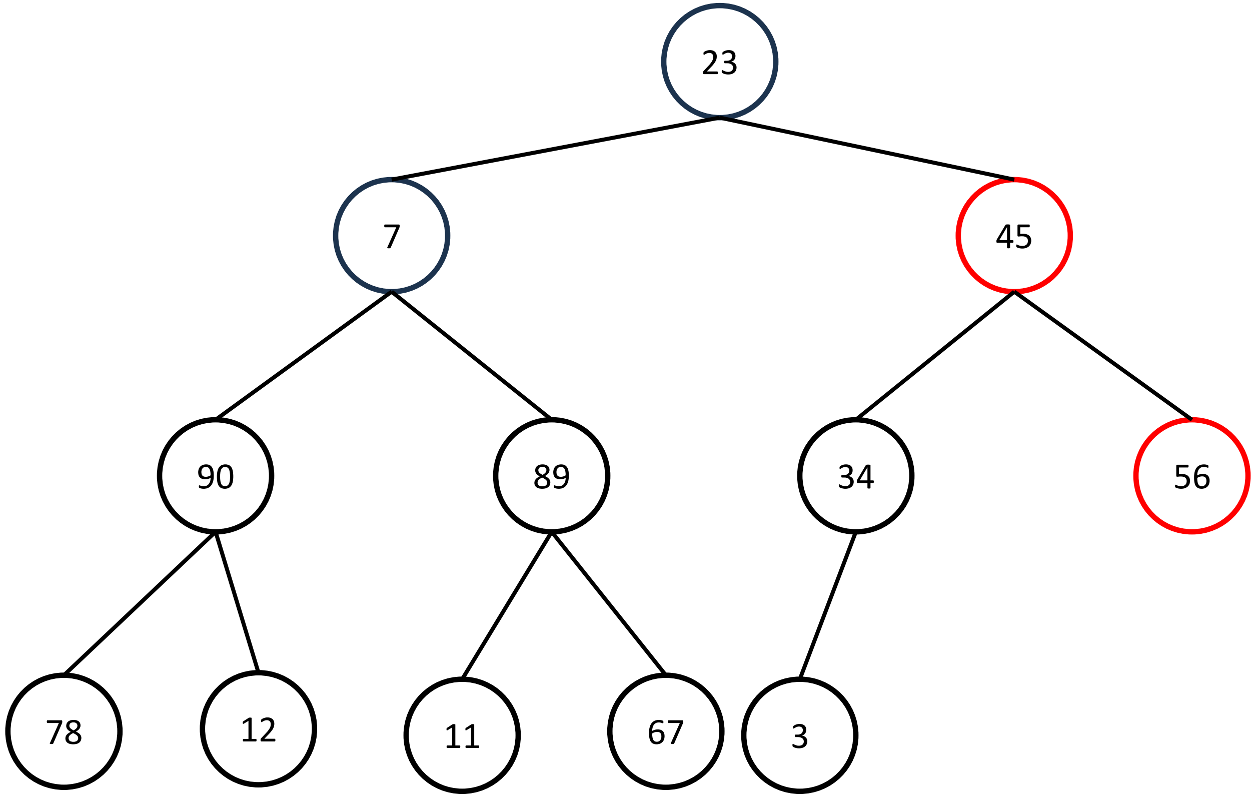


Heap Sort

```
int[] data = {23, 7, 45, 90, 89, 34, 56, 78, 12, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

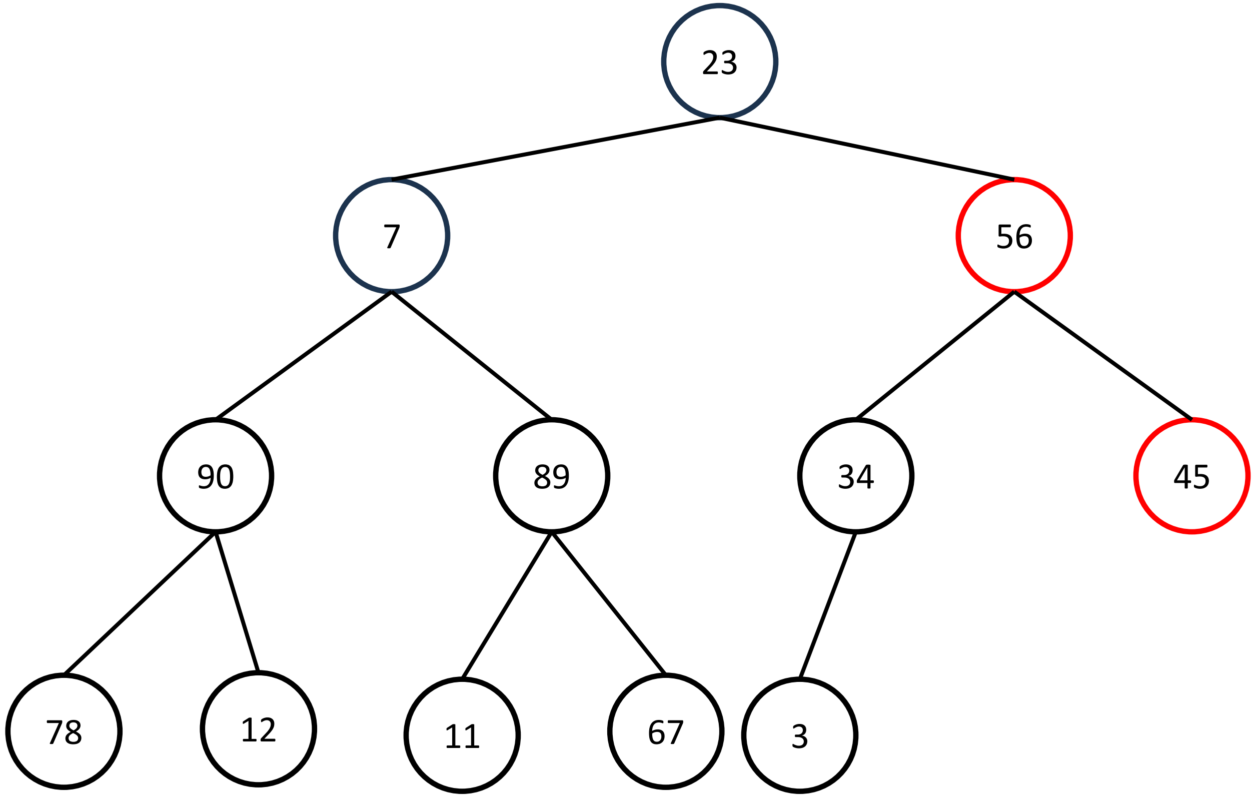


Heap Sort

```
int[] data = {23, 7, 56, 90, 89, 34, 45, 78, 12, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

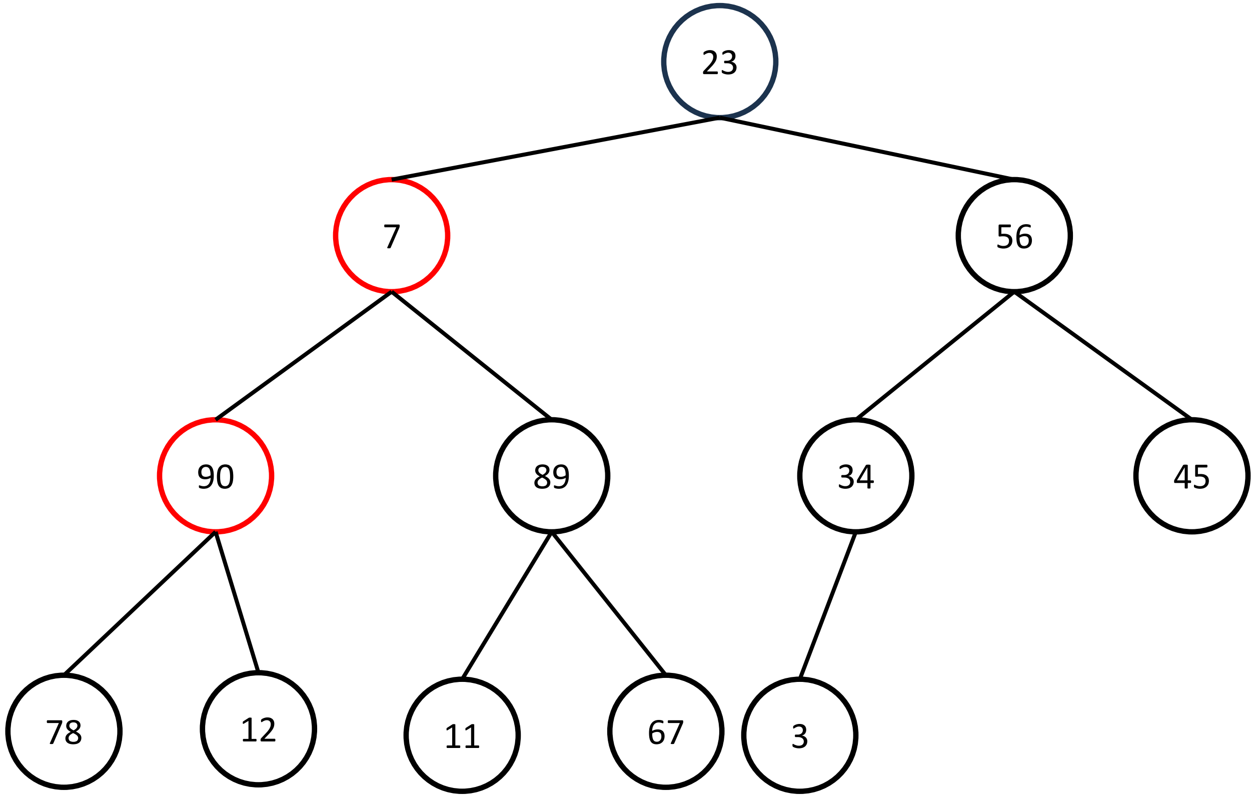


Heap Sort

```
int[] data = {23, 7, 56, 90, 89, 34, 45, 78, 12, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

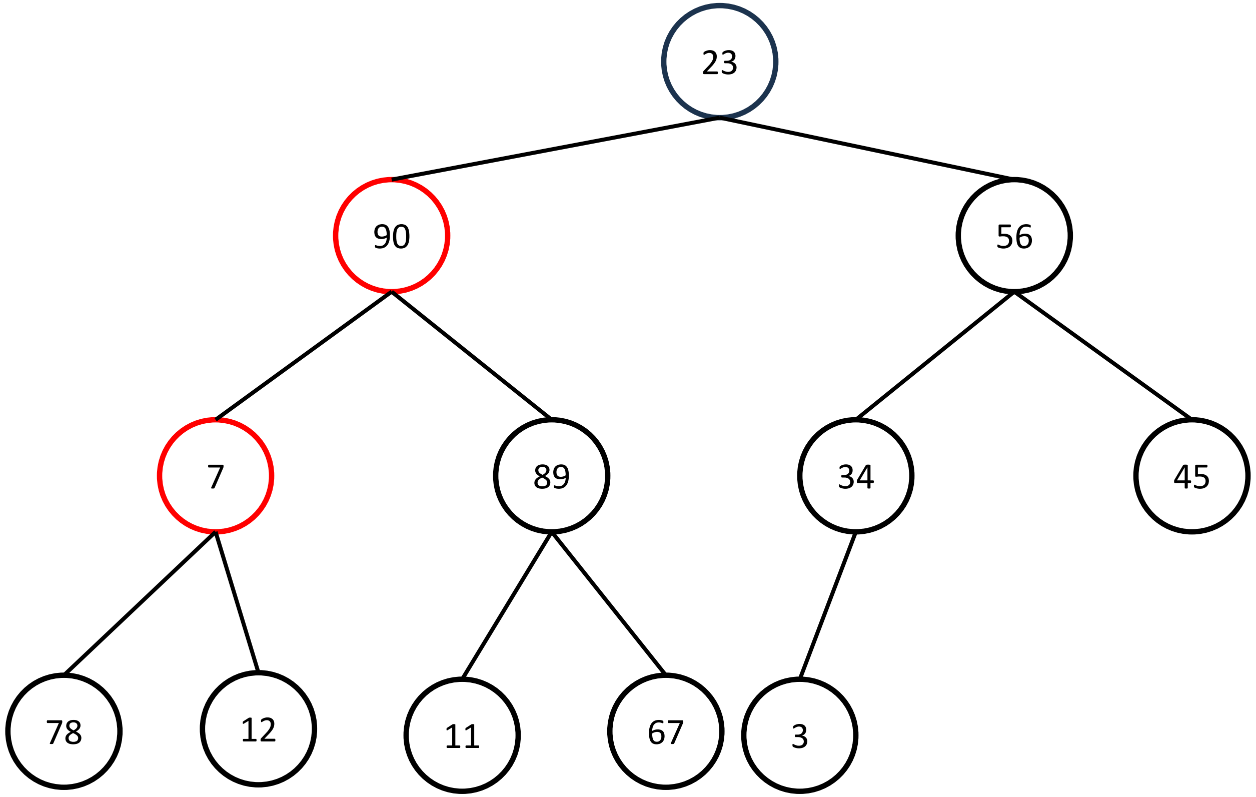


Heap Sort

```
int[] data = {23, 90, 56, 7, 89, 34, 45, 78, 12, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger



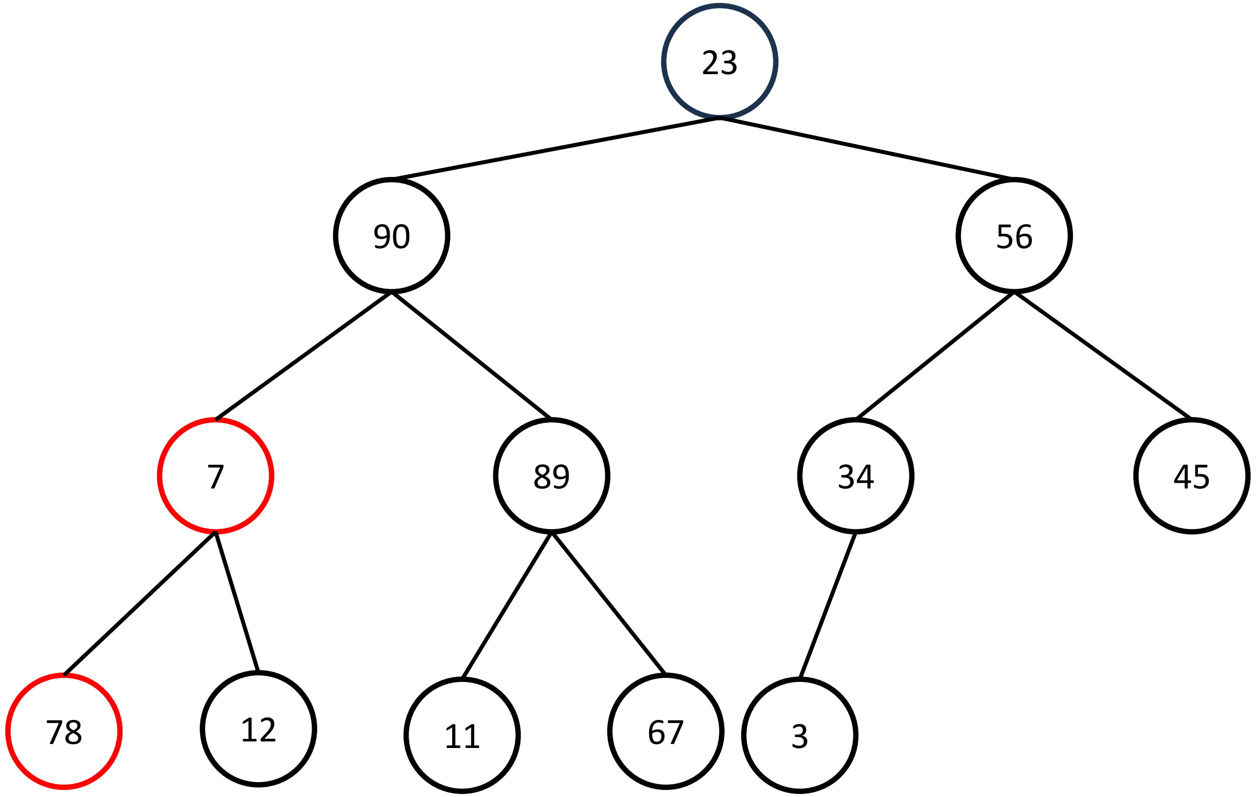
Heapify Down 7 !

Heap Sort

```
int[] data = {23, 90, 56, 7, 89, 34, 45, 78, 12, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger



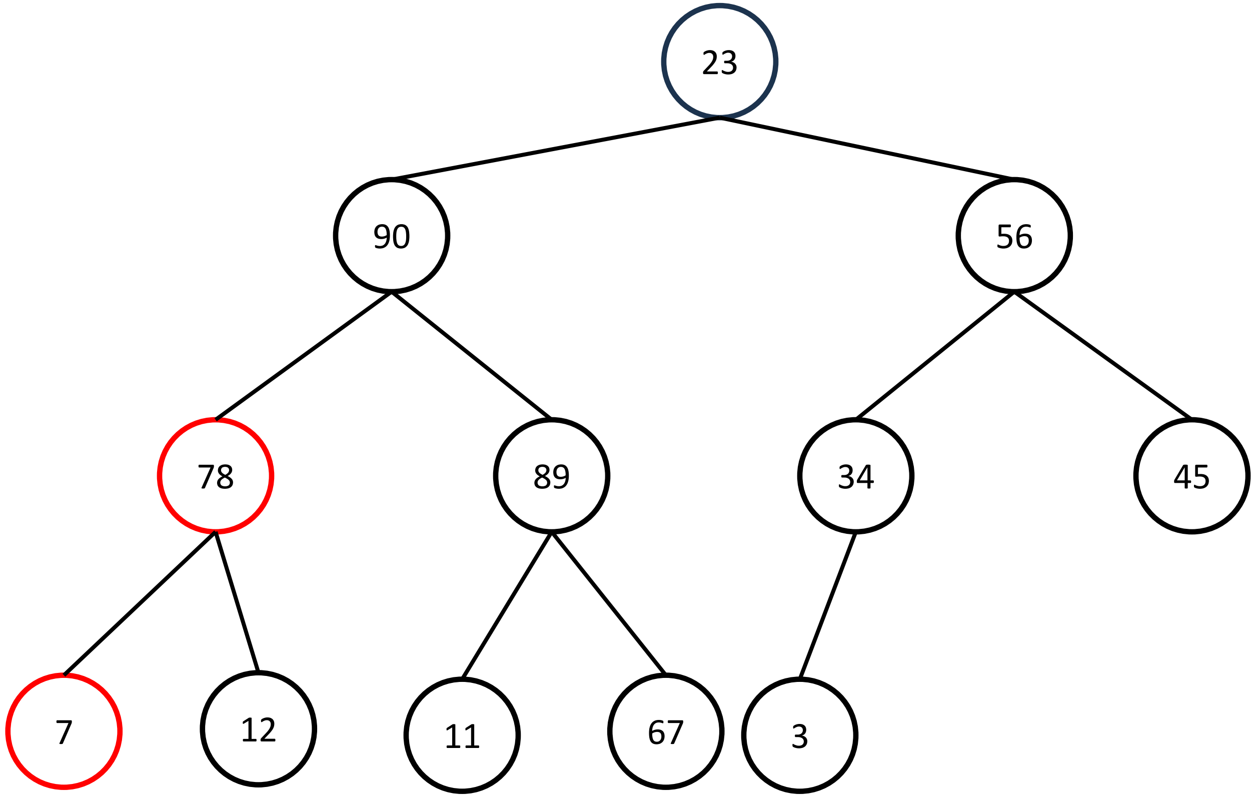
Heapify Down 7 !

Heap Sort

```
int[] data = {23, 90, 56, 78, 89, 34, 45, 7, 12, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger



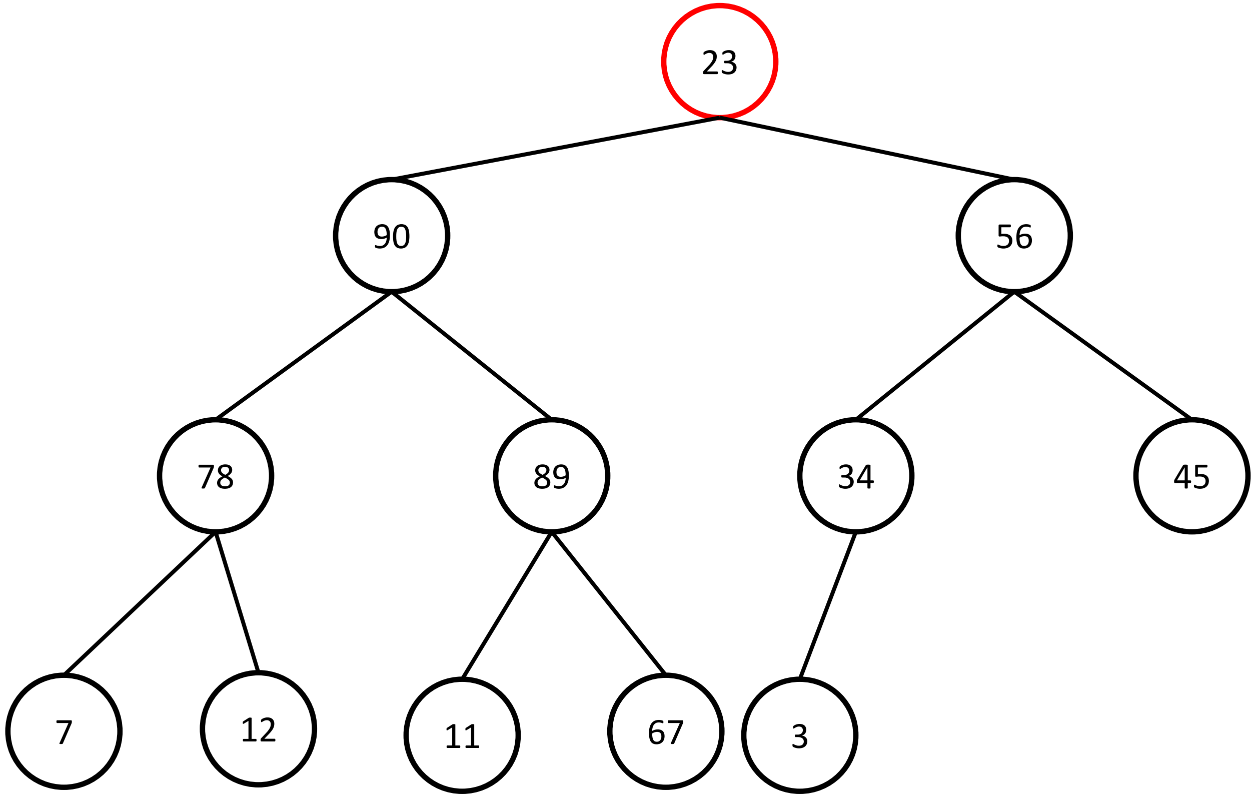
Heapify Down 7 !

Heap Sort

```
int[] data = {23, 90, 56, 78, 89, 34, 45, 7, 12, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger



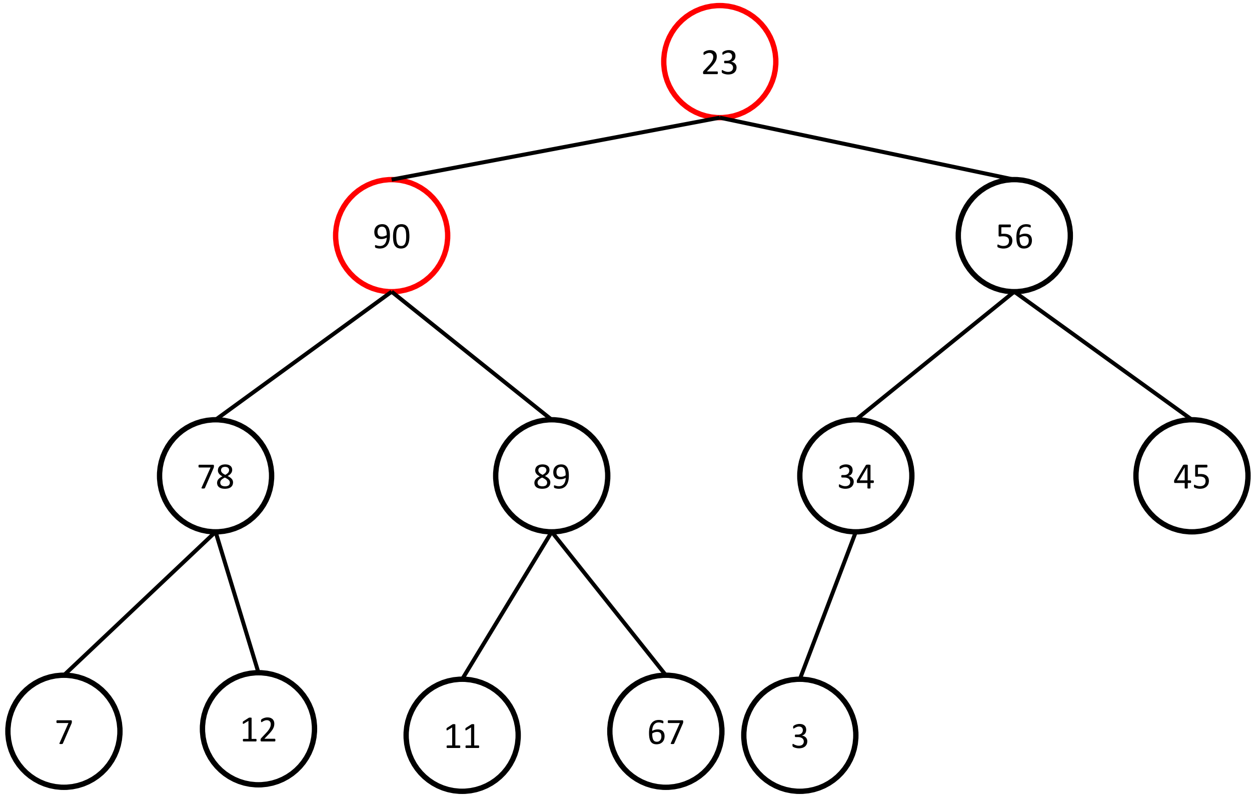
Heapify down 23 !

Heap Sort

```
int[] data = {23, 90, 56, 78, 89, 34, 45, 7, 12, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger



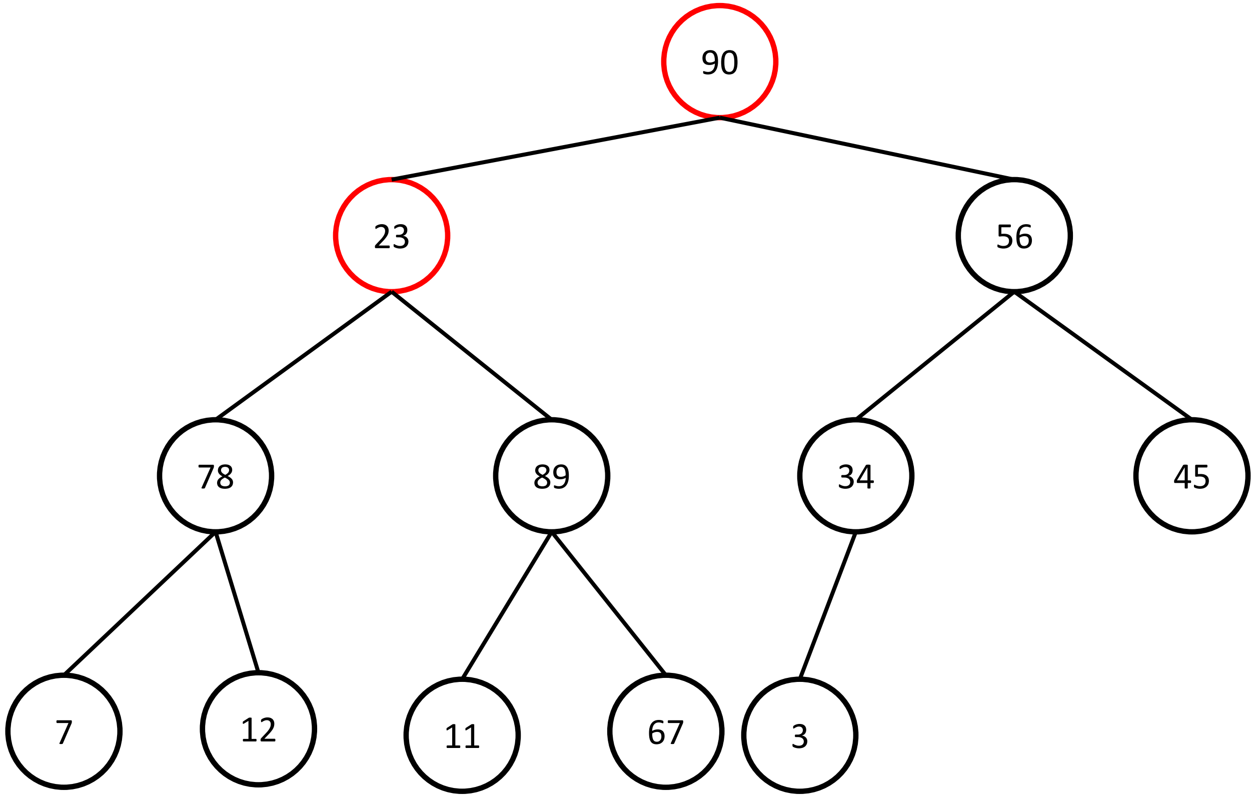
Heapify down 23 !

Heap Sort

```
int[] data = {23, 90, 56, 78, 89, 34, 45, 7, 12, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger



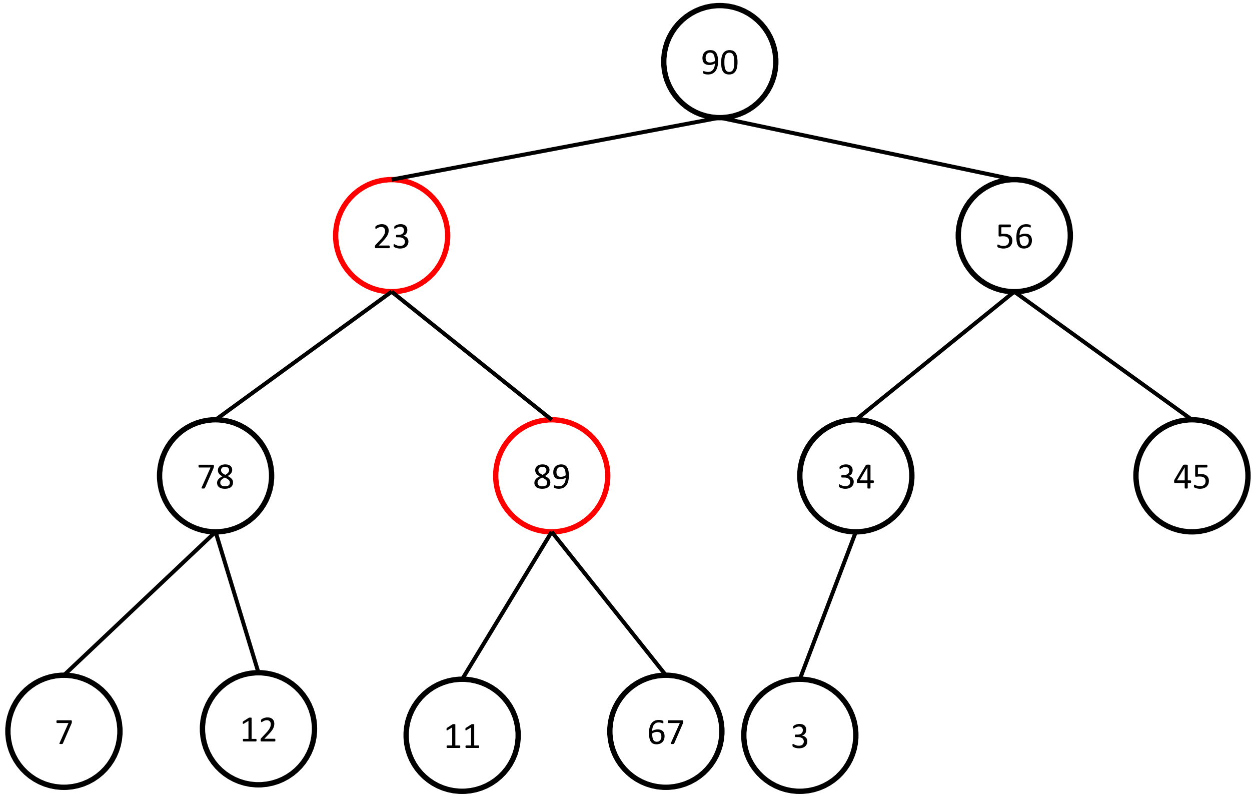
Heapify down 23 !

Heap Sort

```
int[] data = {90, 23, 56, 78, 89, 34, 45, 7, 12, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger



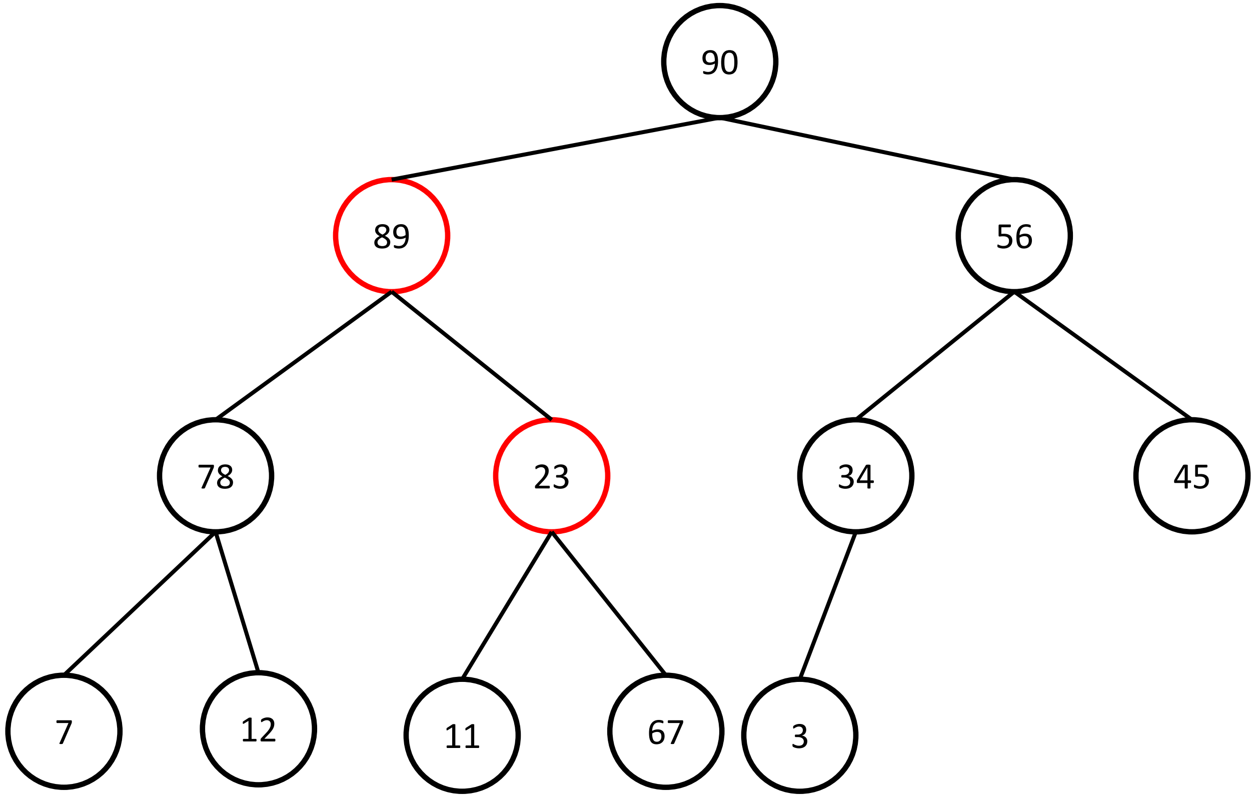
Heapify down 23 !

Heap Sort

```
int[] data = {90, 23, 56, 78, 89, 34, 45, 7, 12, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger



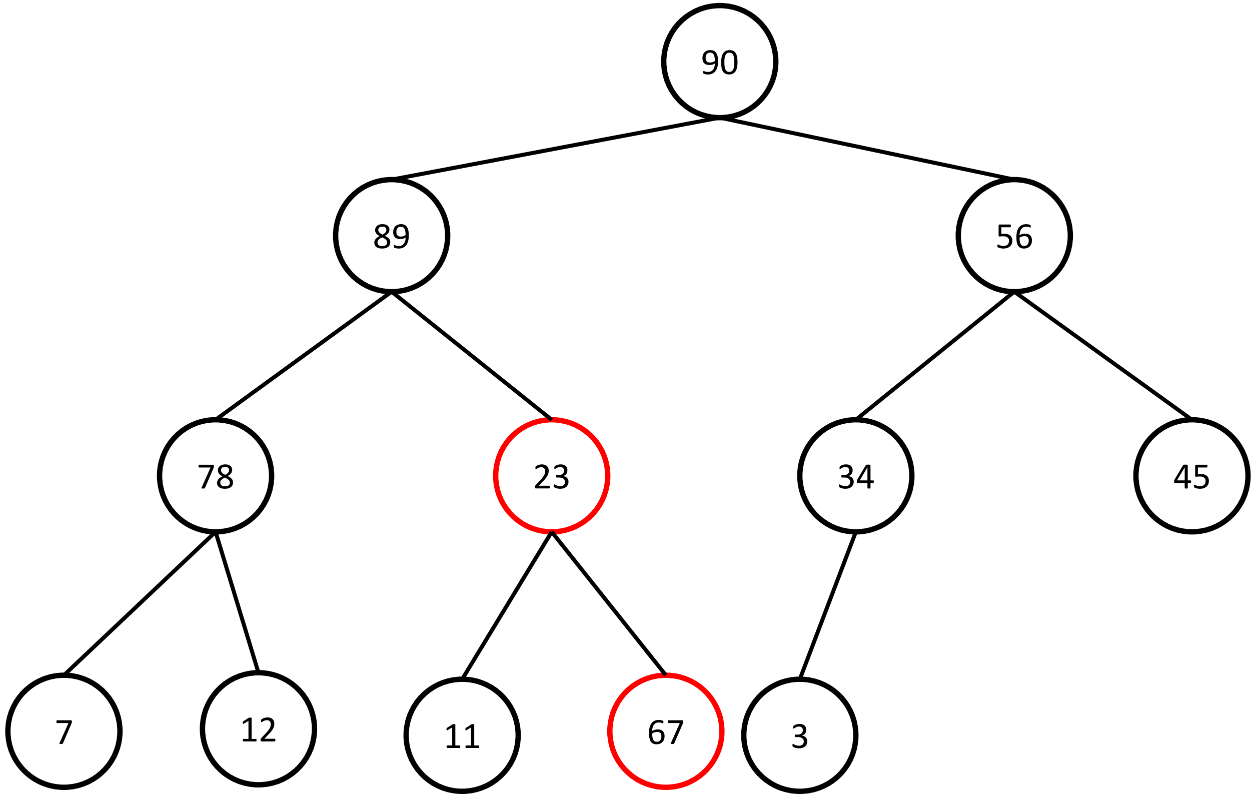
Heapify down 23 !

Heap Sort

```
int[] data = {90, 89, 56, 78, 23, 34, 45, 7, 12, 11, 67, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger



Heapify down 23 !

Heap Sort

```
int[] data = {90, 89, 56, 78, 67, 34, 45, 7, 12, 11, 23, 3}
```

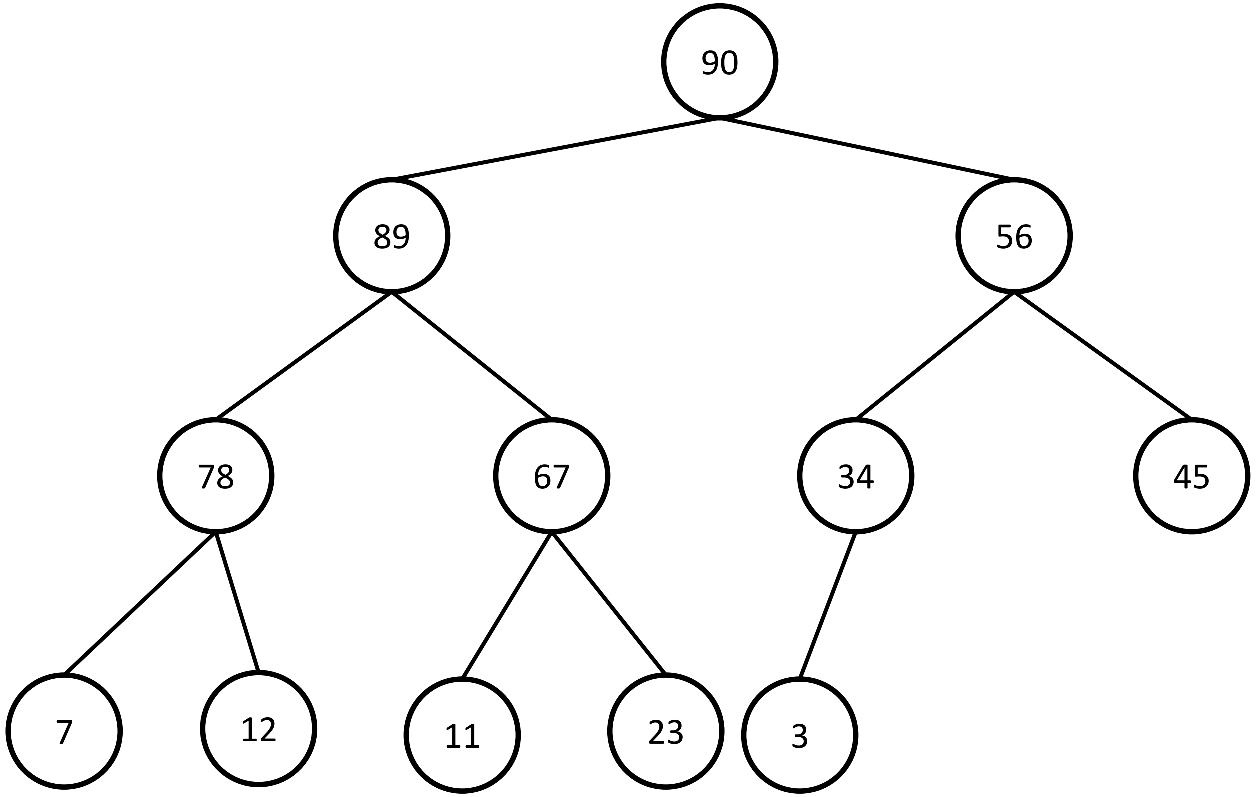
1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

```
for index i data.size to 0 :        O(n)

    heapifyDown(data.size, i)       O(logn)
```

Total for building heap: O(nlogn)



We now have a max heap

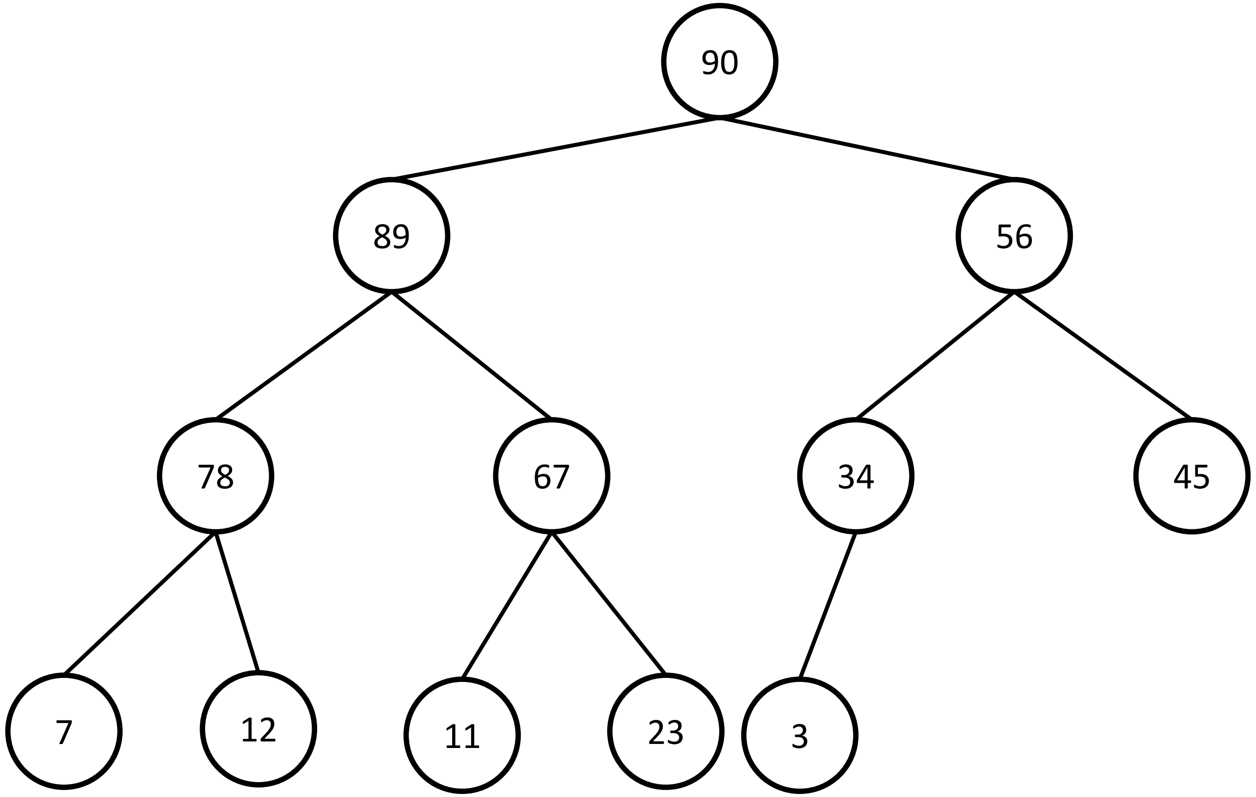
Heap Sort

```
int[] data = {90, 89, 56, 78, 67, 34, 45, 7, 12, 11, 23, 3}
```

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

2. Swap the root with last element, and heapify down the new root



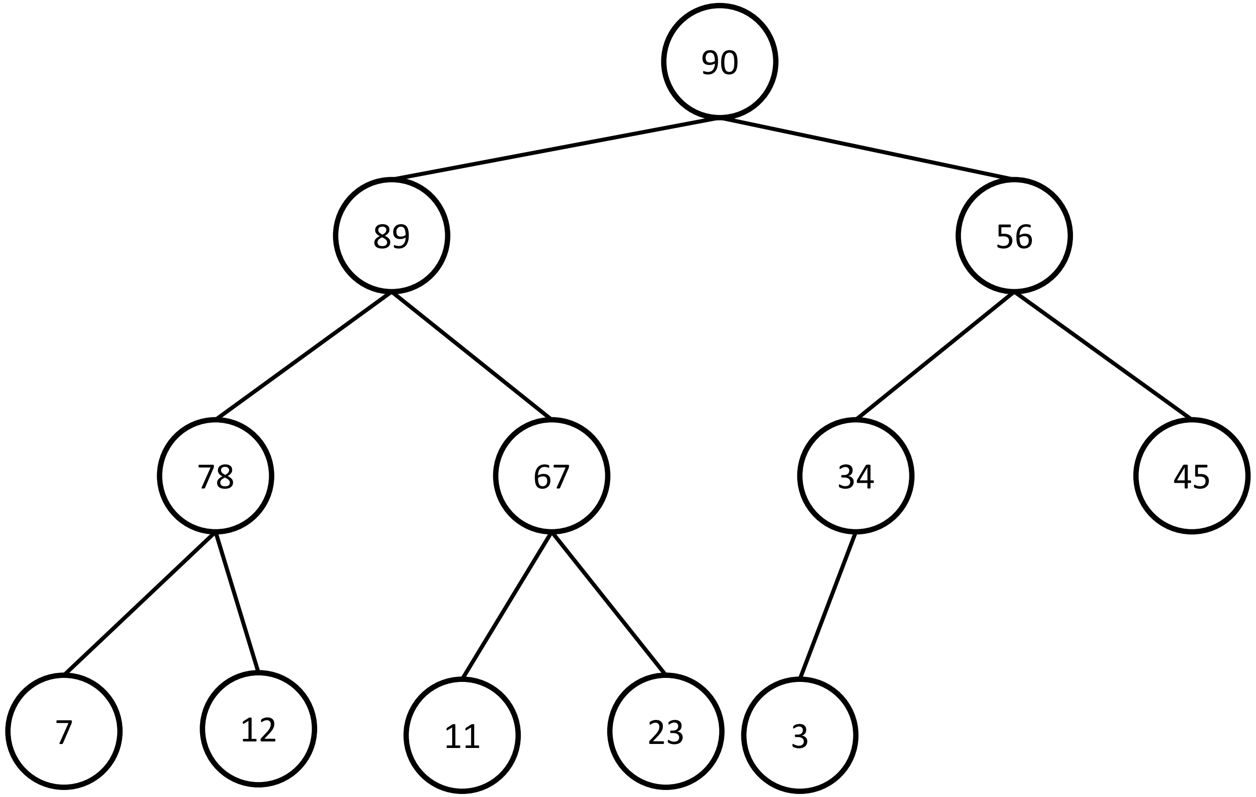
Heap Sort

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

2. Swap the root with last element, and heapify down the new root

Repeat N amount of times



90	89	56	78	67	34	45	7	12	11	23	3

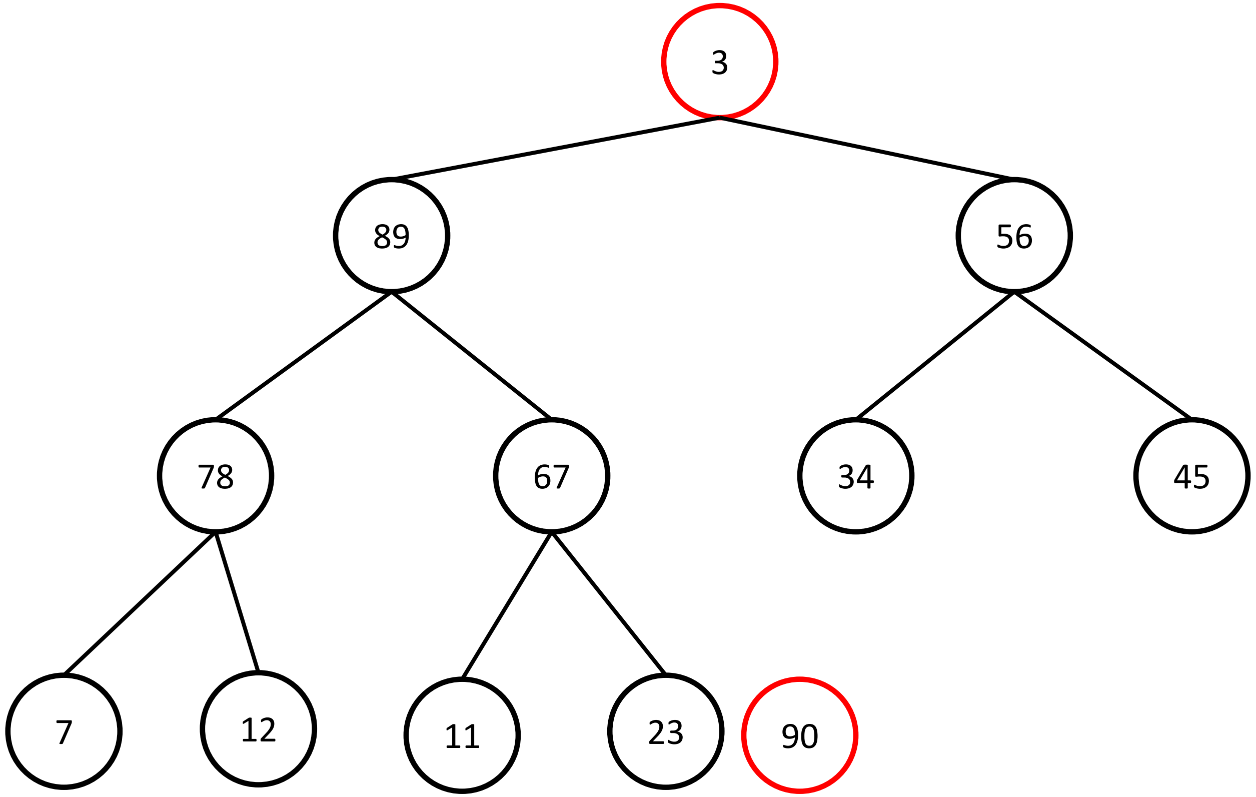
Heap Sort

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

2. Swap the root with last element, and heapify down the new root

Repeat N amount of times



Heapify Down 3

3	89	56	78	67	34	45	7	12	11	23	90

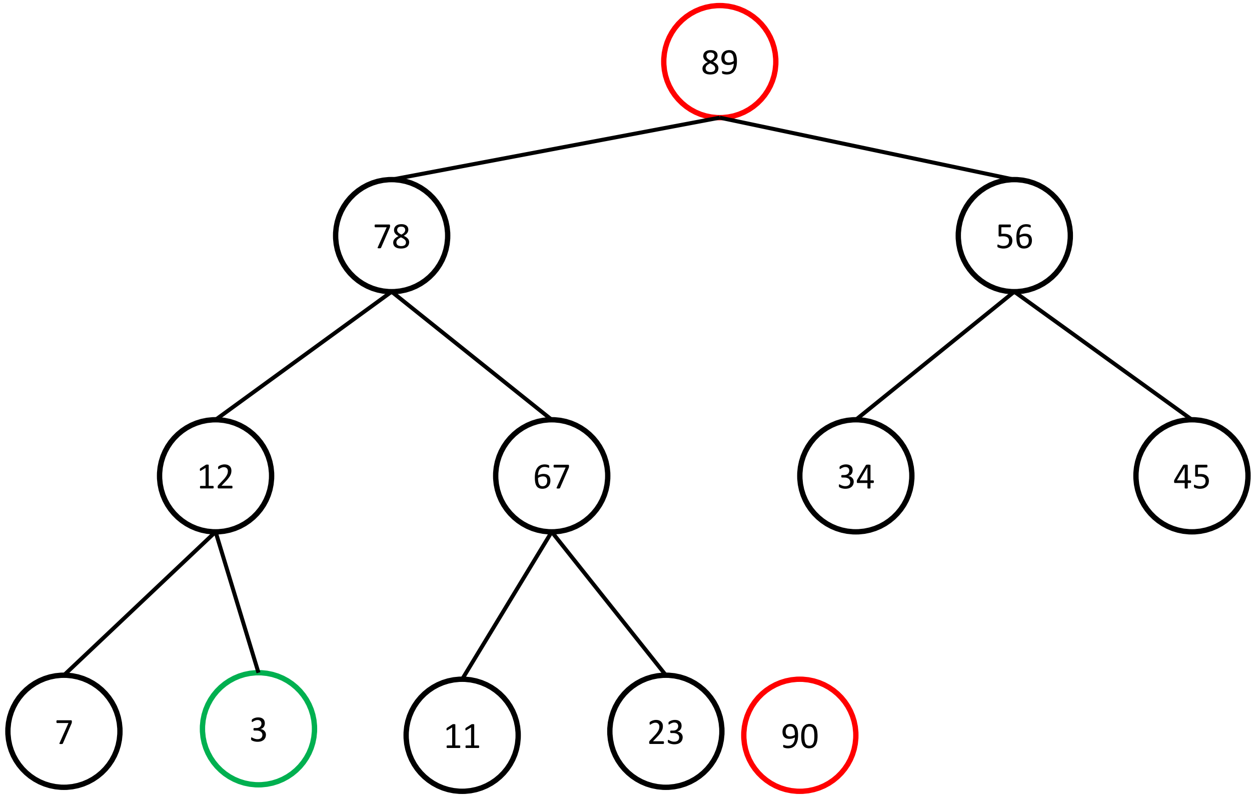
Heap Sort

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

2. Swap the root with last element, and heapify down the new root

Repeat N amount of times



We have one element in the correct spot. Now repeat N times (N = heap size)

89	78	56	12	67	34	45	7	3	11	23	90

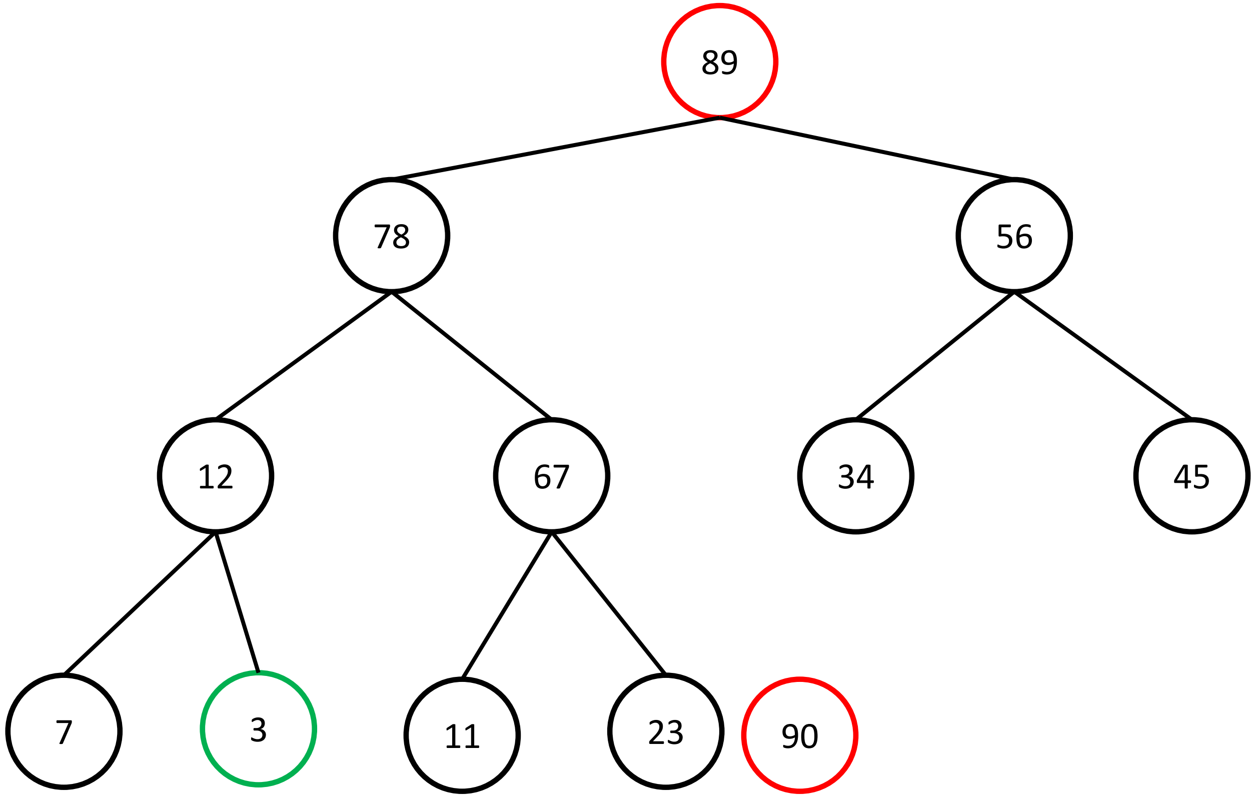
Heap Sort

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

2. Swap the root with last element, and heapify down the new root

Repeat N amount of times



We don't want to "shrink" our array, but we need to change the bounds during Heapify Down

89	78	56	12	67	34	45	7	3	11	23	90

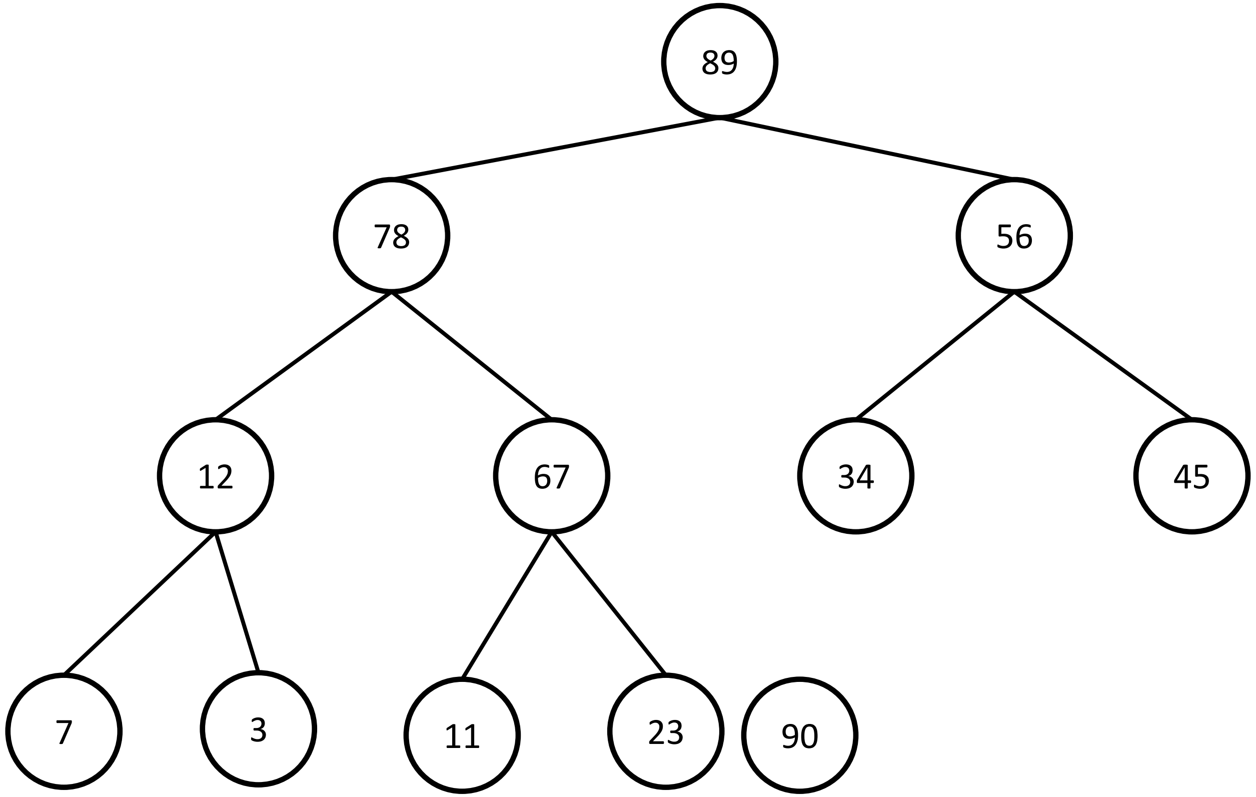
Heap Sort

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

2. Swap the root with last element, and heapify down the new root

Repeat N amount of times



New bounds for Heapify Down

89	78	56	12	67	34	45	7	3	11	23	90

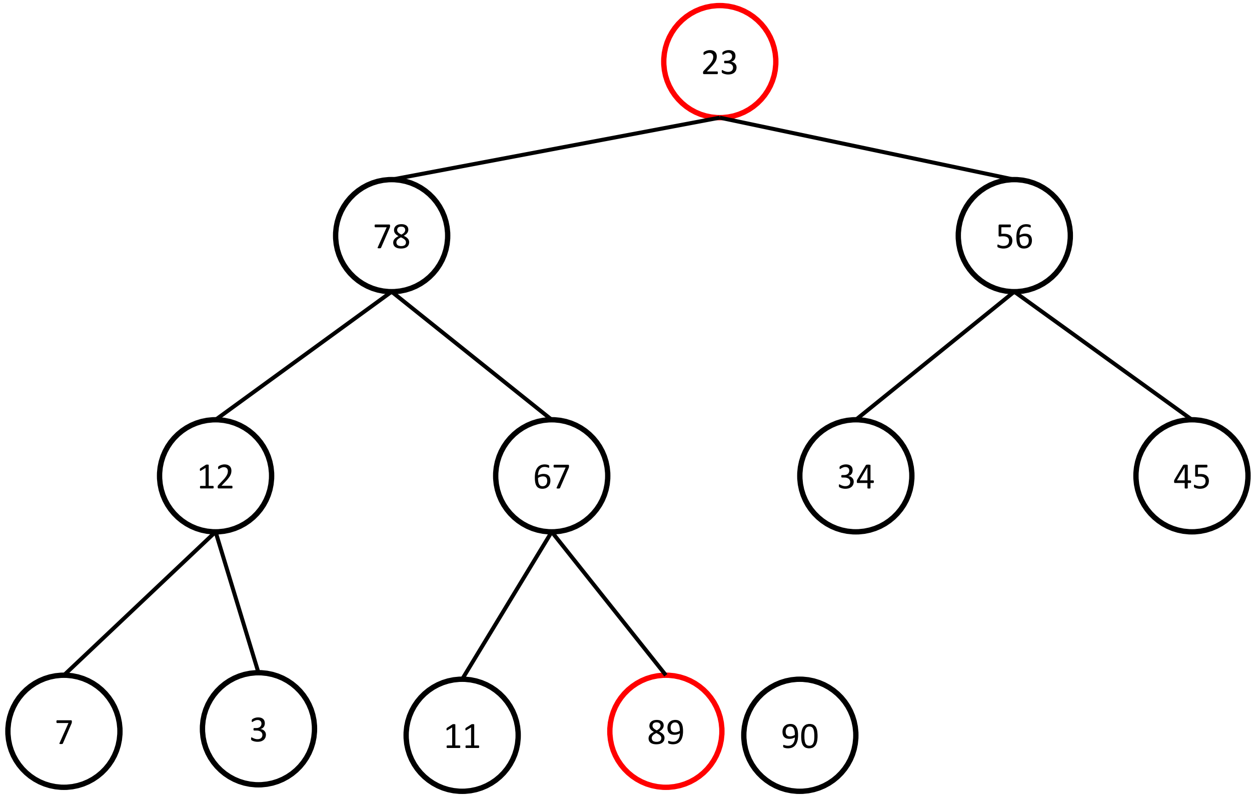
Heap Sort

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

2. Swap the root with last element, and heapify down the new root

Repeat N amount of times



New bounds for Heapify Down

23	78	56	12	67	34	45	7	3	11	89	90

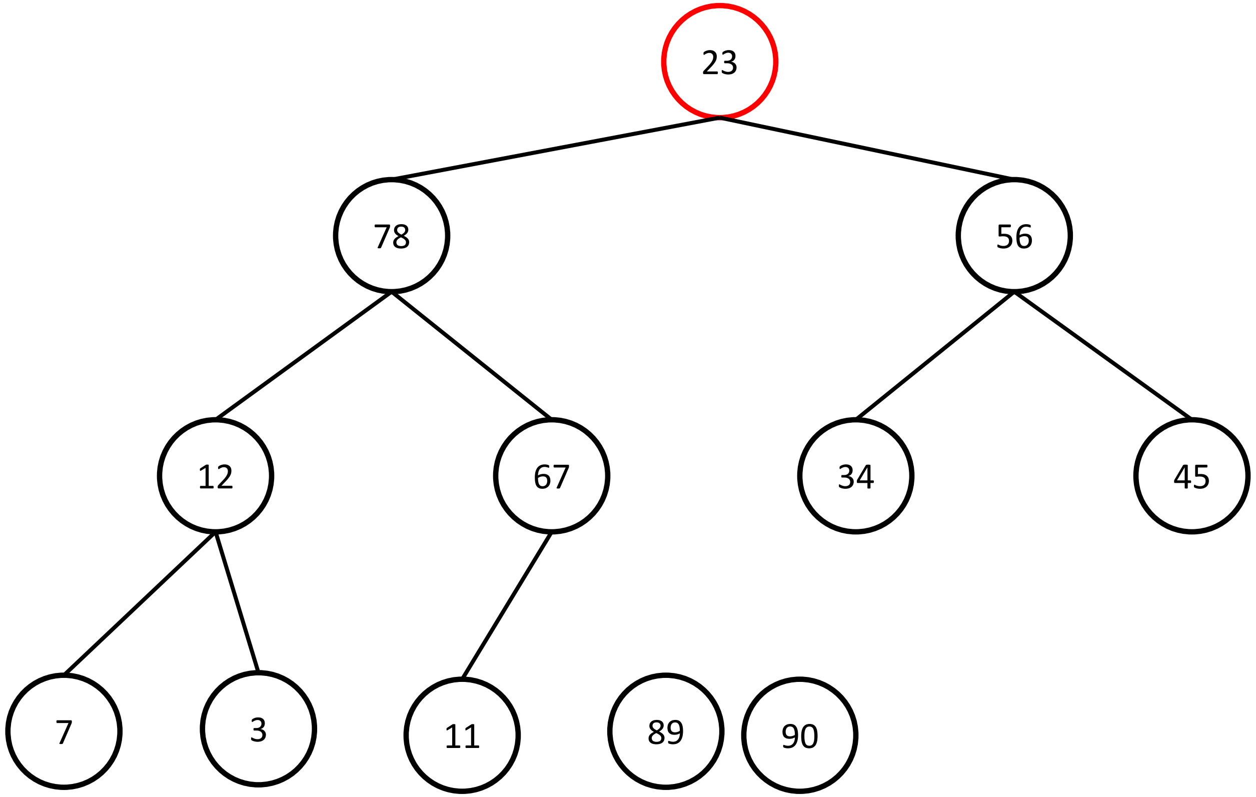
Heap Sort

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

2. Swap the root with last element, and heapify down the new root

Repeat N amount of times



Heapify Down 23

23	78	56	12	67	34	45	7	3	11	89	90

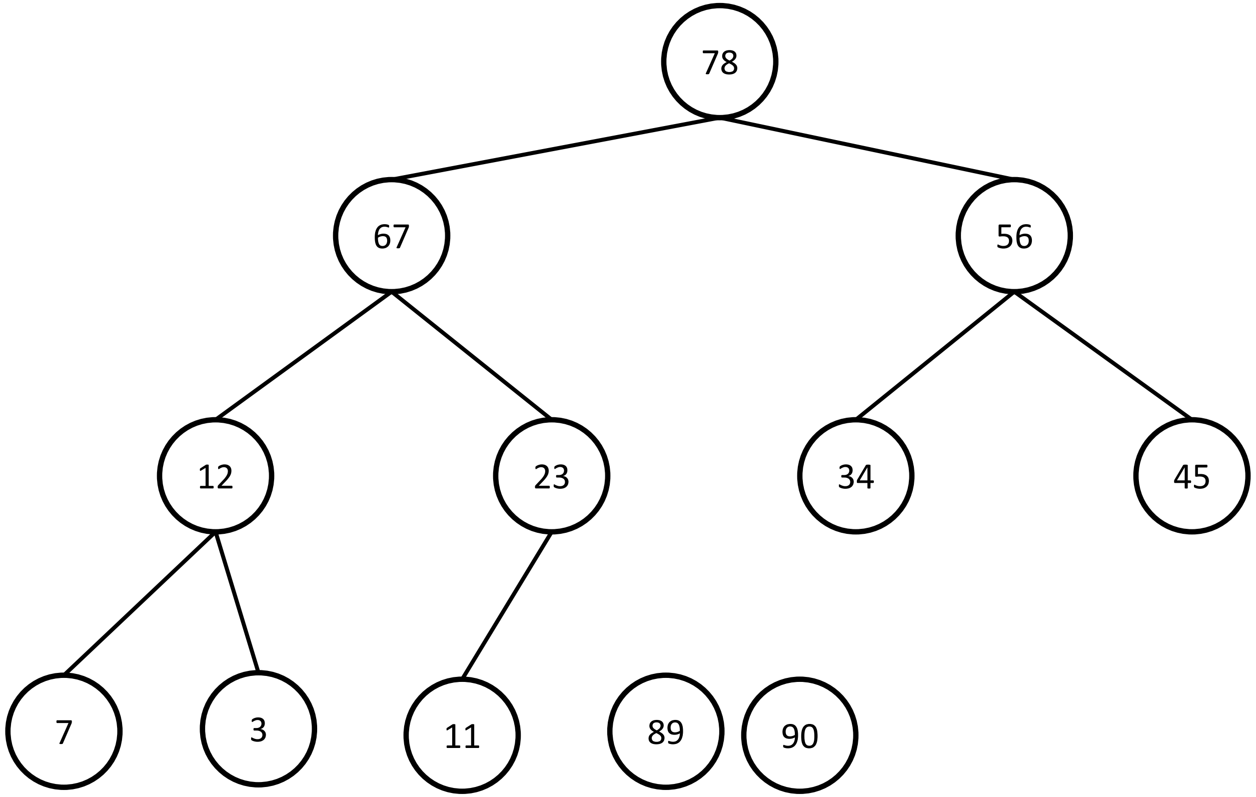
Heap Sort

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

2. Swap the root with last element, and heapify down the new root

Repeat N amount of times



Heapify Down 23

78	67	56	12	23	34	45	7	3	11	89	90

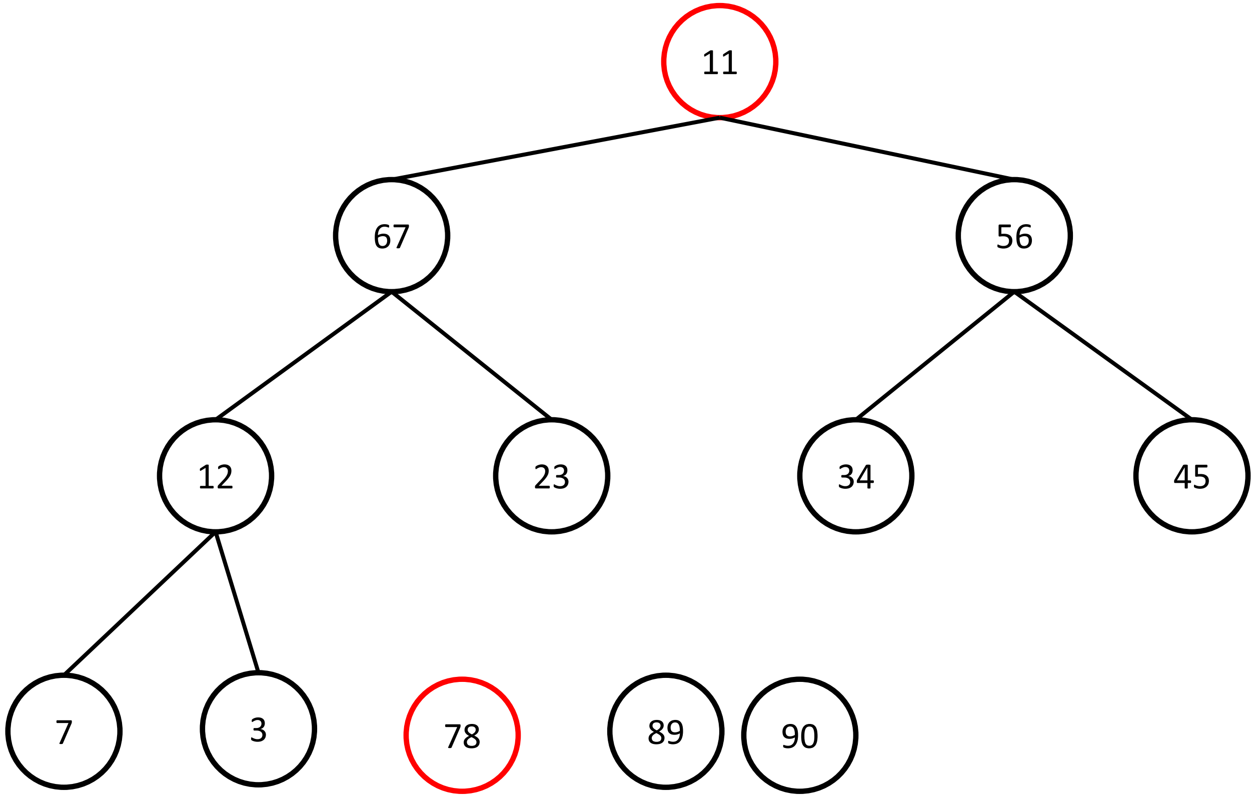
Heap Sort

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

2. Swap the root with last element, and heapify down the new root

Repeat N amount of times



Heapify Down 11

11	67	56	12	23	34	45	7	3	78	89	90

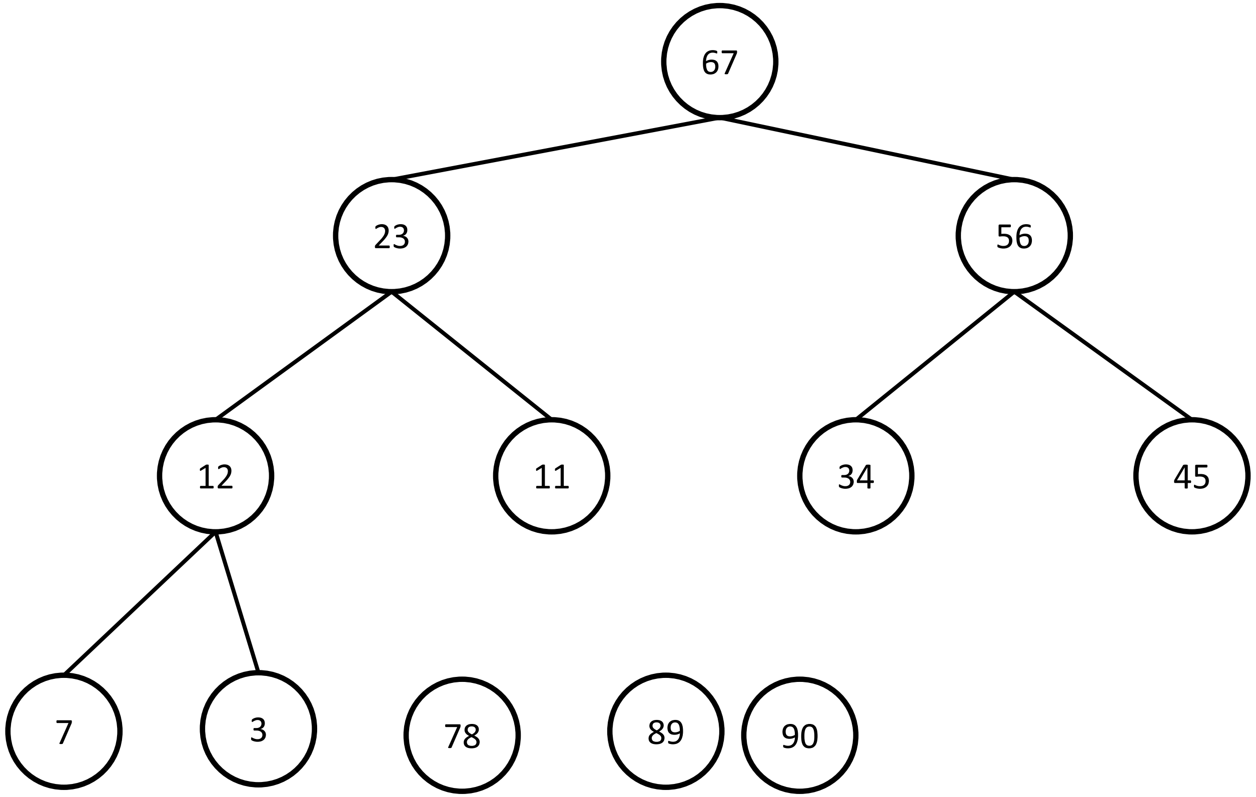
Heap Sort

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

2. Swap the root with last element, and heapify down the new root

Repeat N amount of times



Heapify Down 11

67	23	56	12	11	34	45	7	3	78	89	90

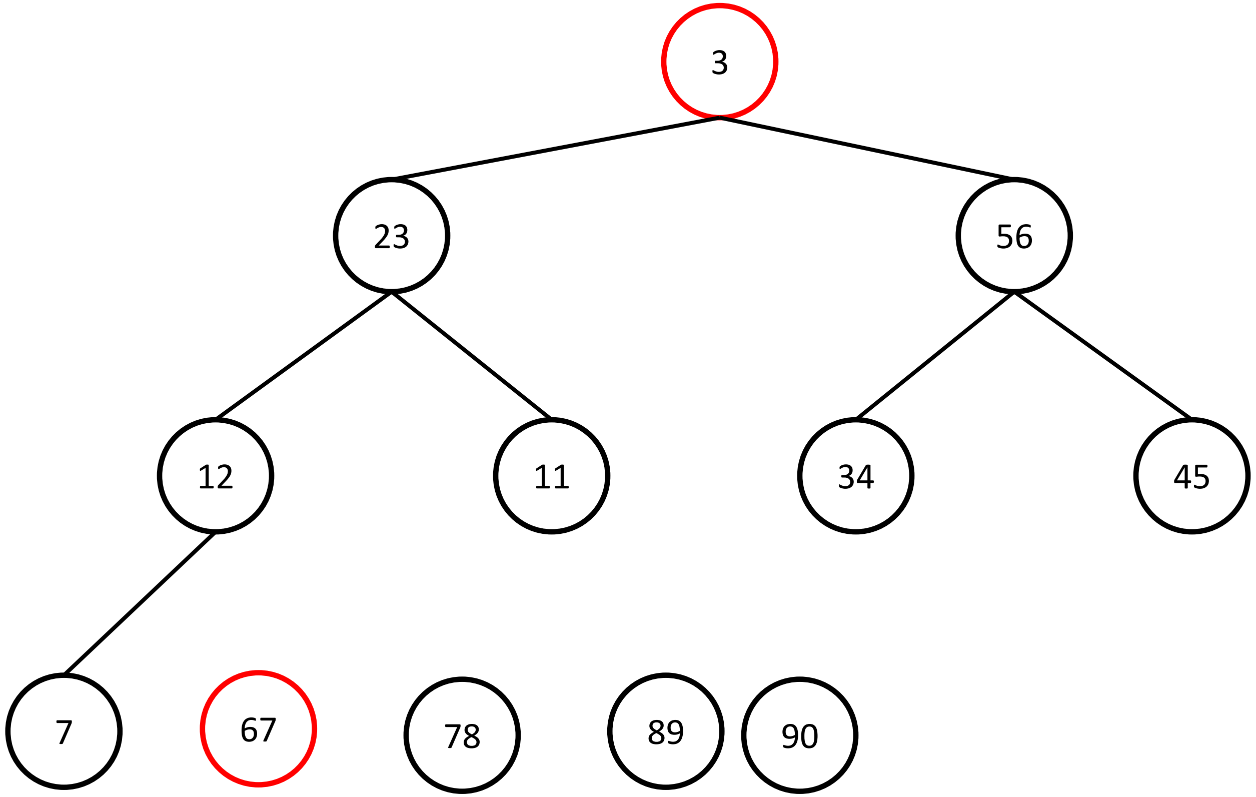
Heap Sort

1. Build a **Max Heap** from the unsorted array

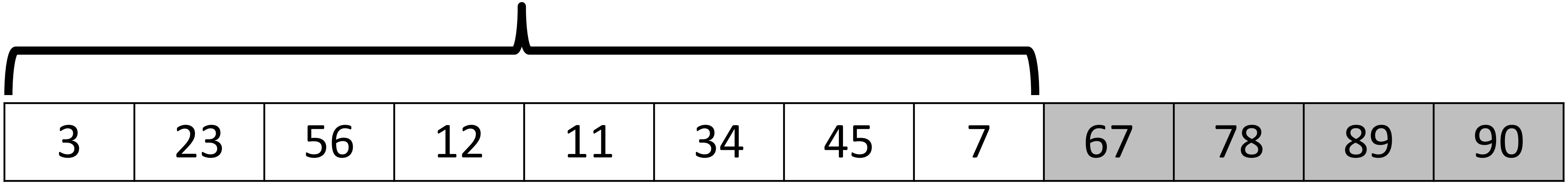
   Work through the array backwards, and swap a node with a child if its larger

2. Swap the root with last element, and heapify down the new root

   Repeat N amount of times



Heapify Down 3



3	23	56	12	11	34	45	7	67	78	89	90

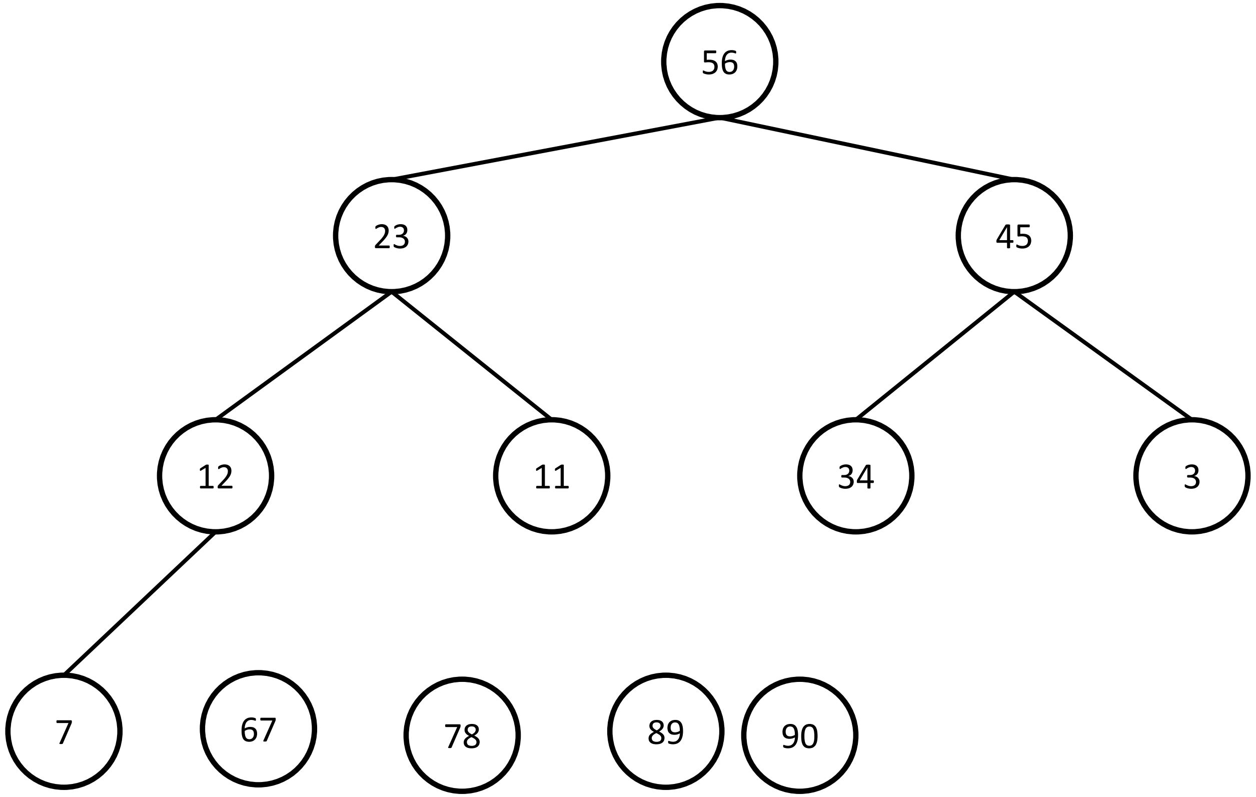
Heap Sort

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

2. Swap the root with last element, and heapify down the new root

Repeat N amount of times



Heapify Down 3

56	23	45	12	11	34	3	7	67	78	89	90

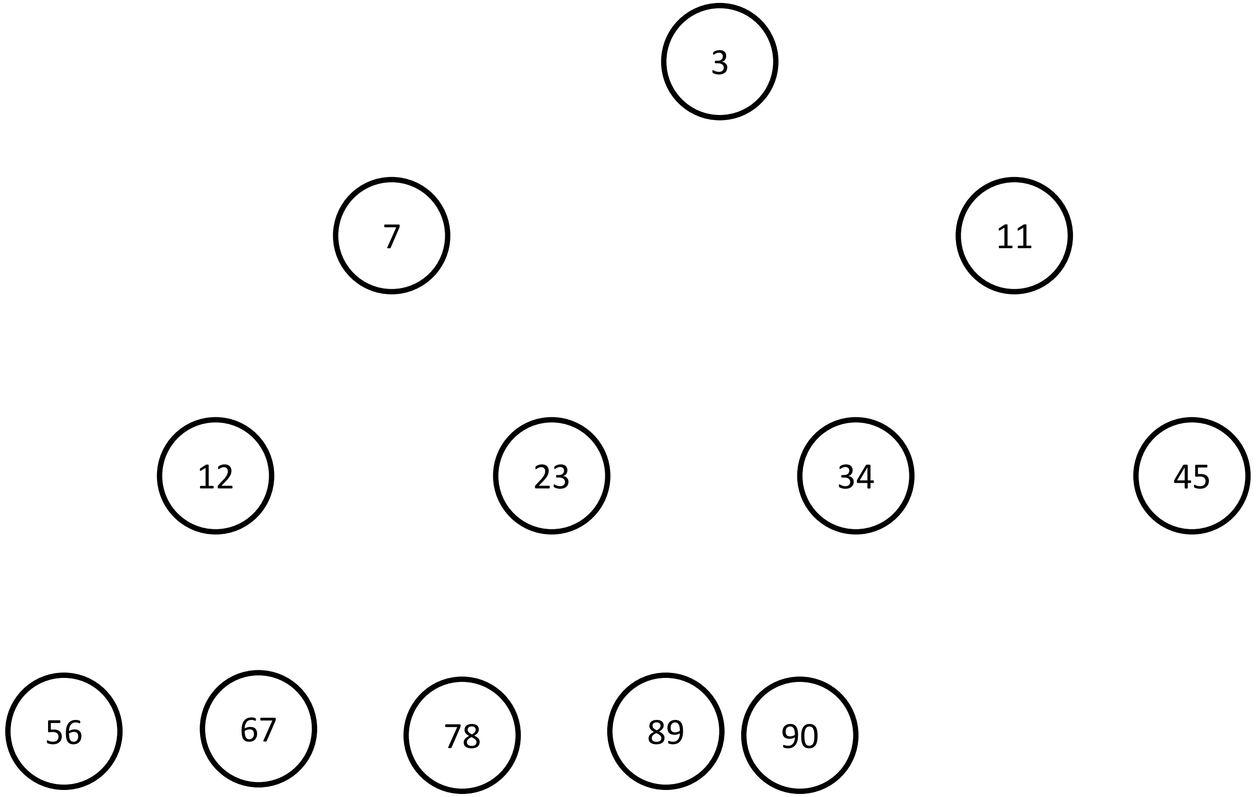
Heap Sort

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

2. Swap the root with last element, and heapify down the new root

Repeat N amount of times



(Fast forward…)

3	7	11	12	23	34	45	56	67	78	89	90

Heap Sort

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

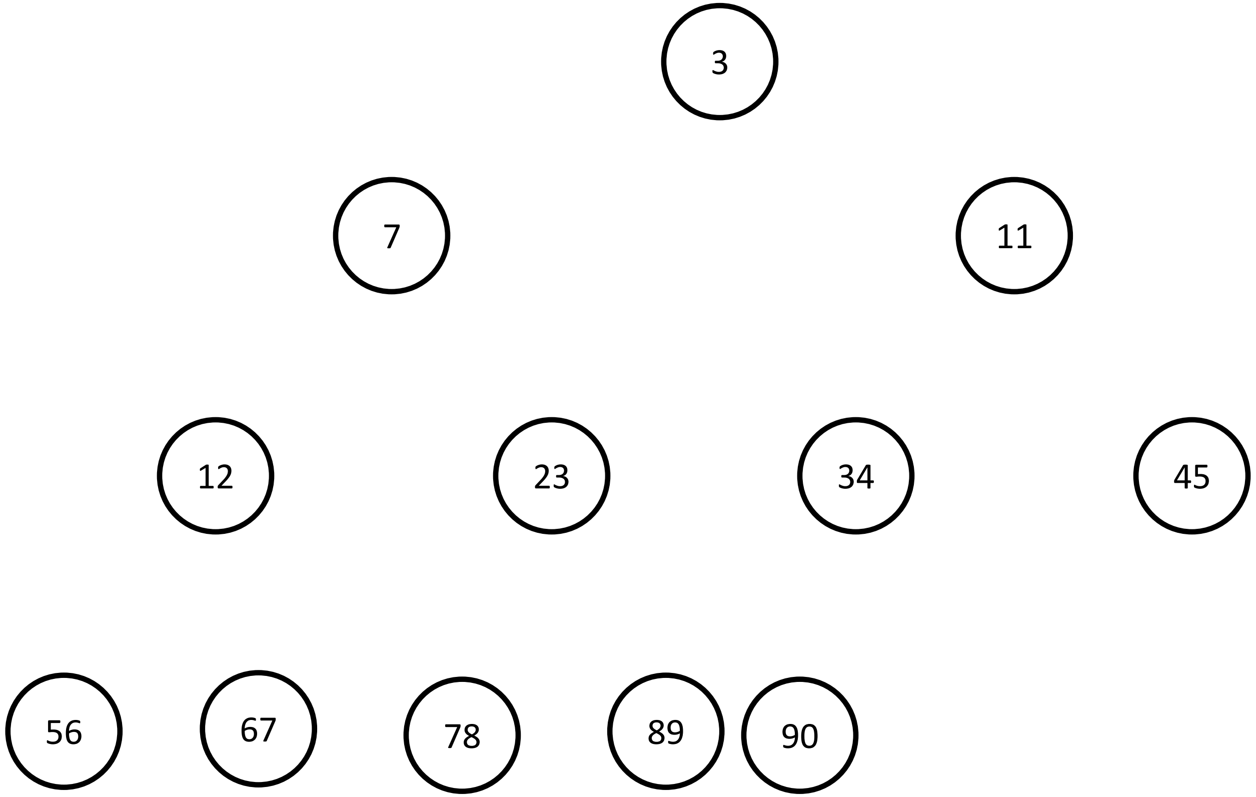
2. Swap the root with last element, and heapify down the new root

O(log n)

Repeat N amount of times

O(n)

“Sorting” step = O(nlogn)



✓

3	7	11	12	23	34	45	56	67	78	89	90

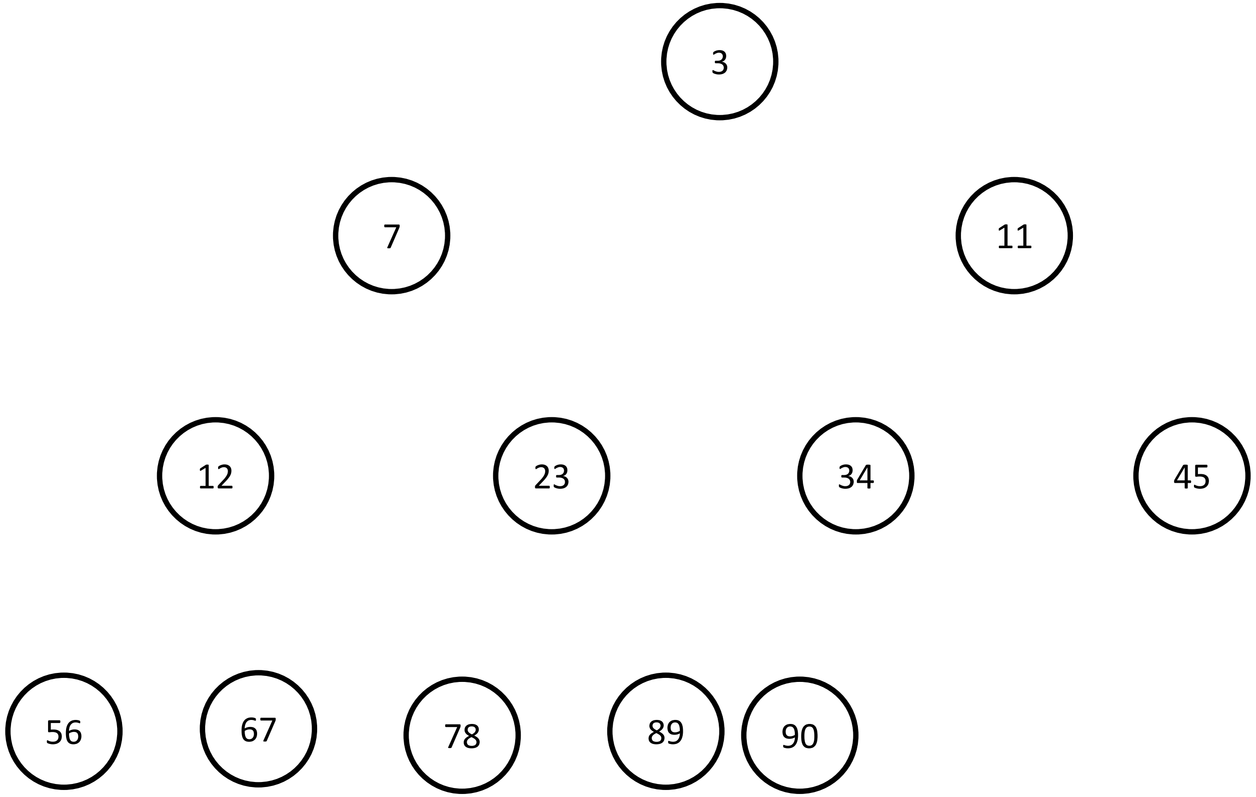
Heap Sort

1. Build a **Max Heap** from the unsorted array

Work through the array backwards, and swap a node with a child if its larger

2. Swap the root with last element, and heapify down the new root

Repeat N amount of times



O(nlogn) + O(nlogn) $\in O(n\log n)$ ✓

3	7	11	12	23	34	45	56	67	78	89	90

Heap Sort

https://www.youtube.com/watch?v=iXAjiDQbPSw

Lab 6