CSCI 232:
Data Structures and Algorithms

Heaps

Reese Pearsall

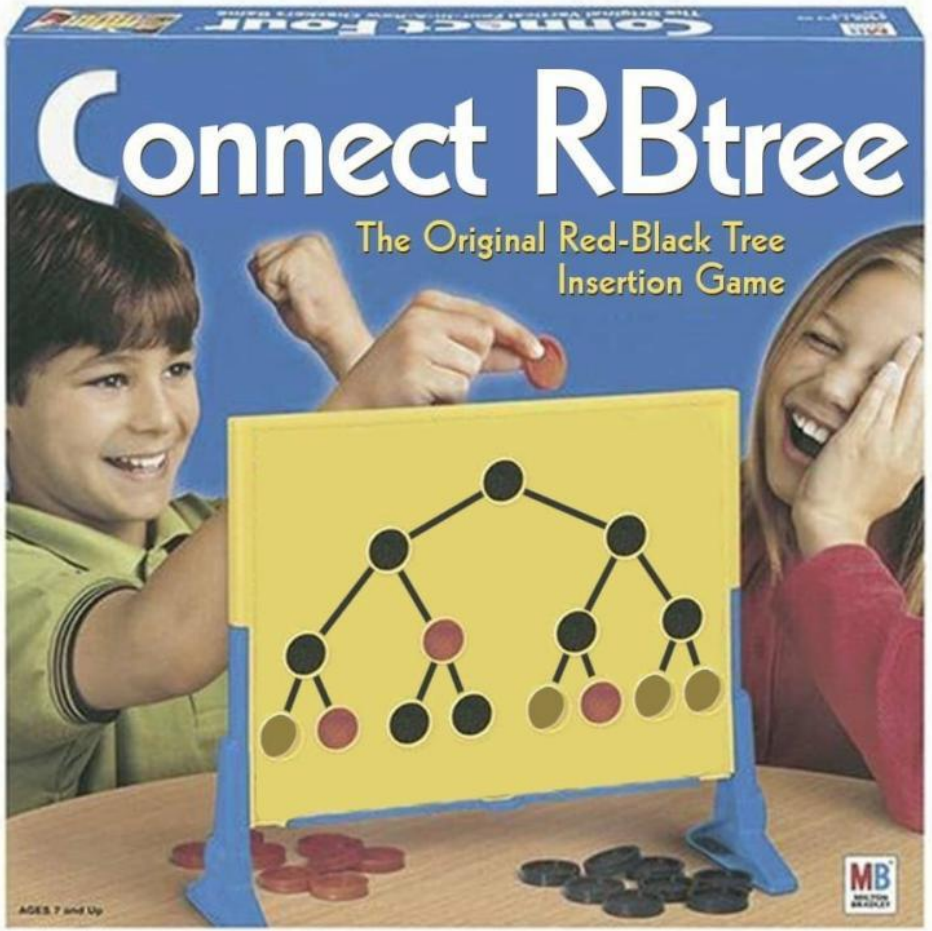
Spring 2024

Announcements

Lab 7 due **tomorrow** at 11:59 PM

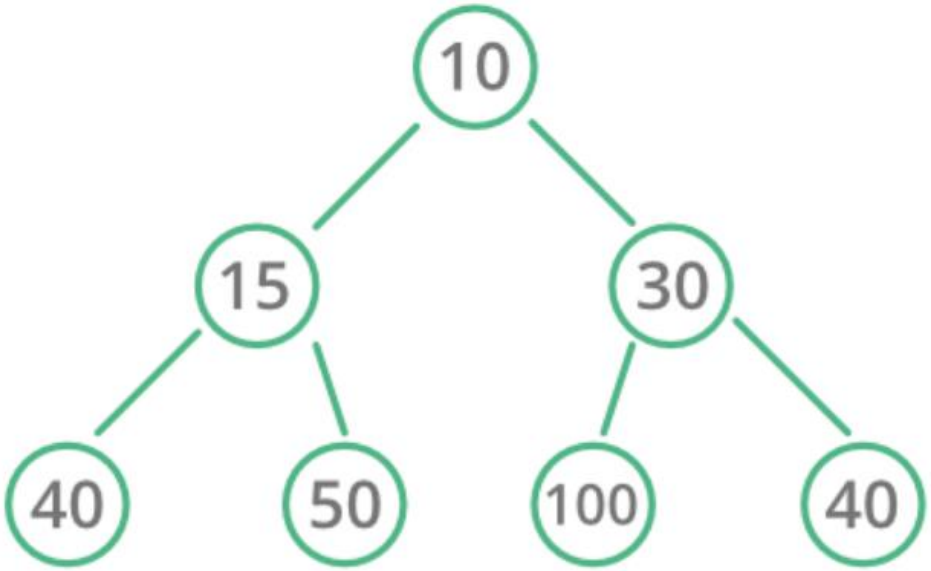
Program 2 due **Sunday** at 11:59 PM

No class next week



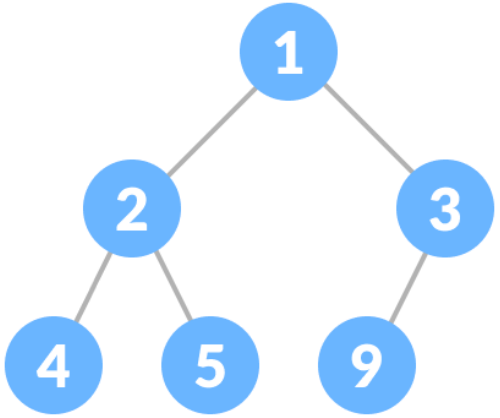
the game that you play in nightmares ^

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

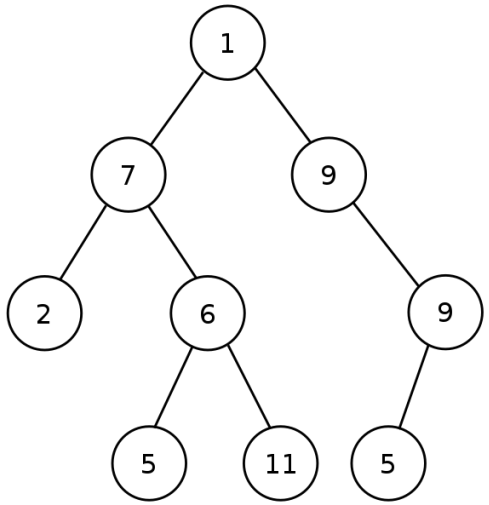


The **Heap** data structure is **complete** binary tree that follows the heap property

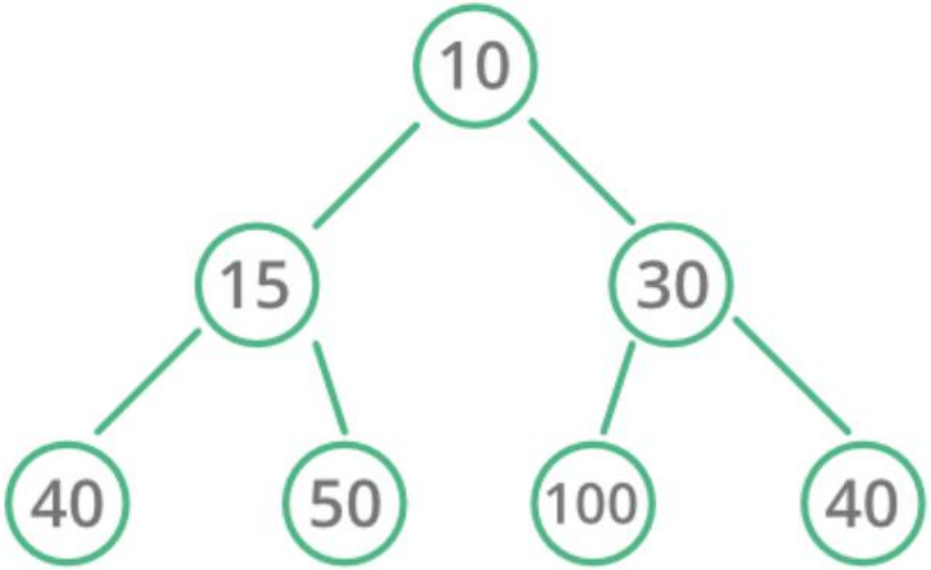
Complete tree - Every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible



complete



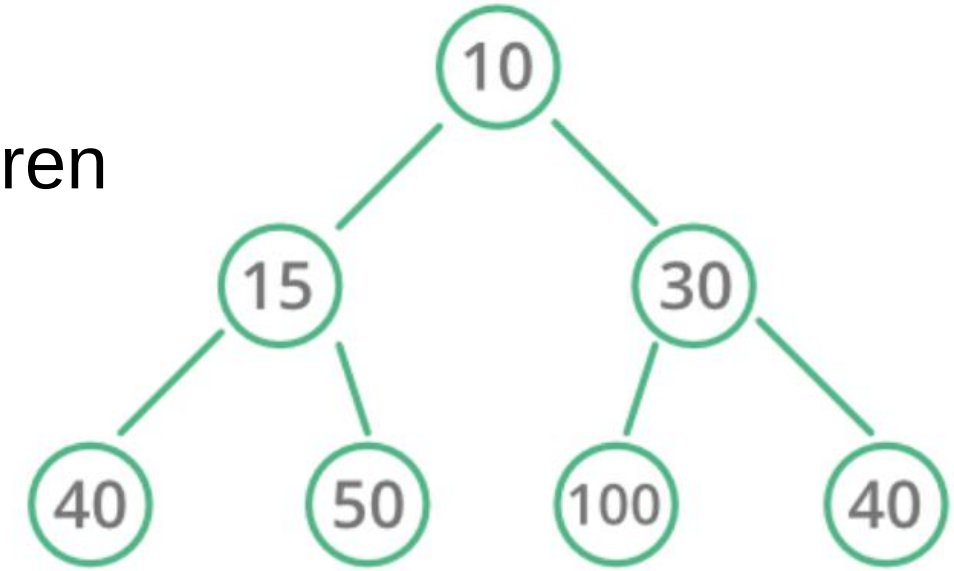
Not complete



complete

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

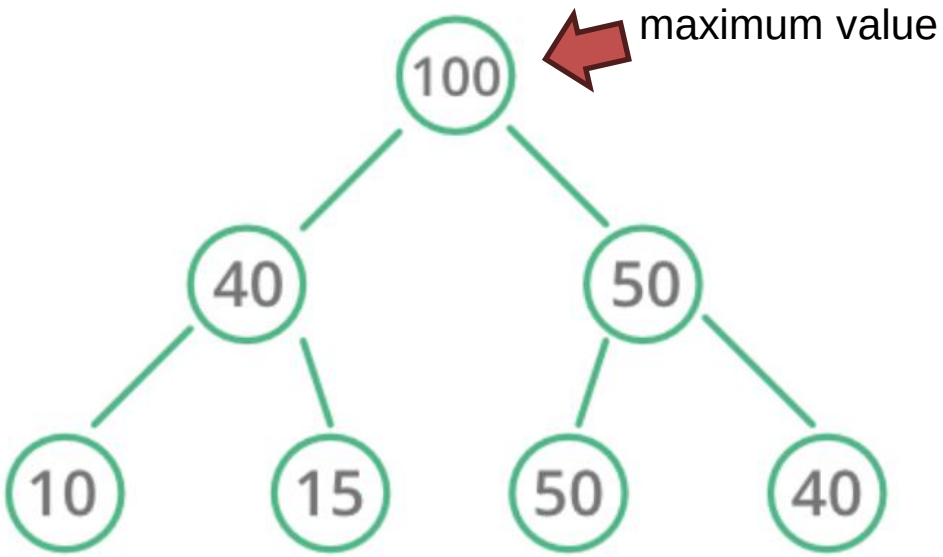
Binary – cannot have more than two children



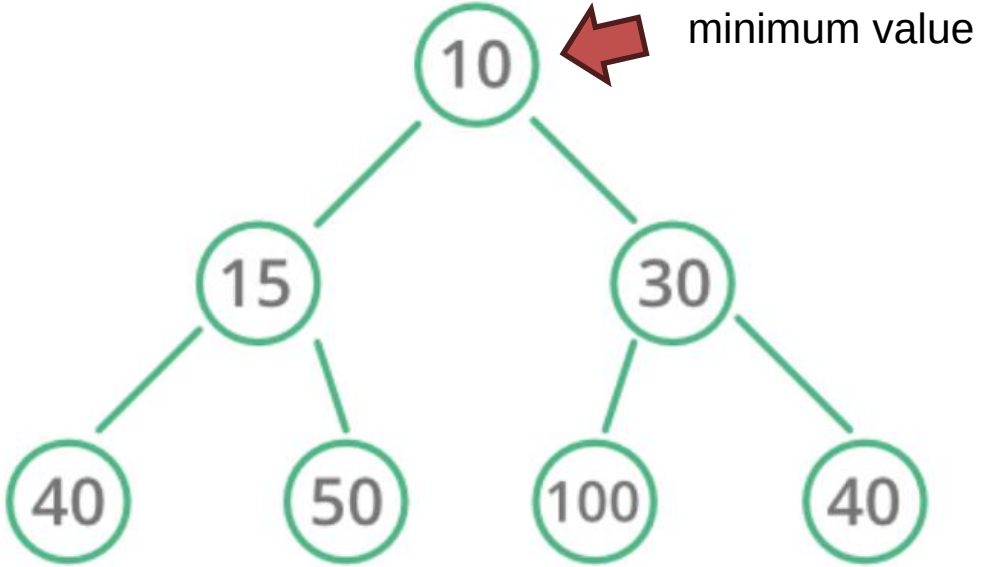
The **Heap** data structure is complete binary tree that follows the **heap property**

Two types of heaps

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

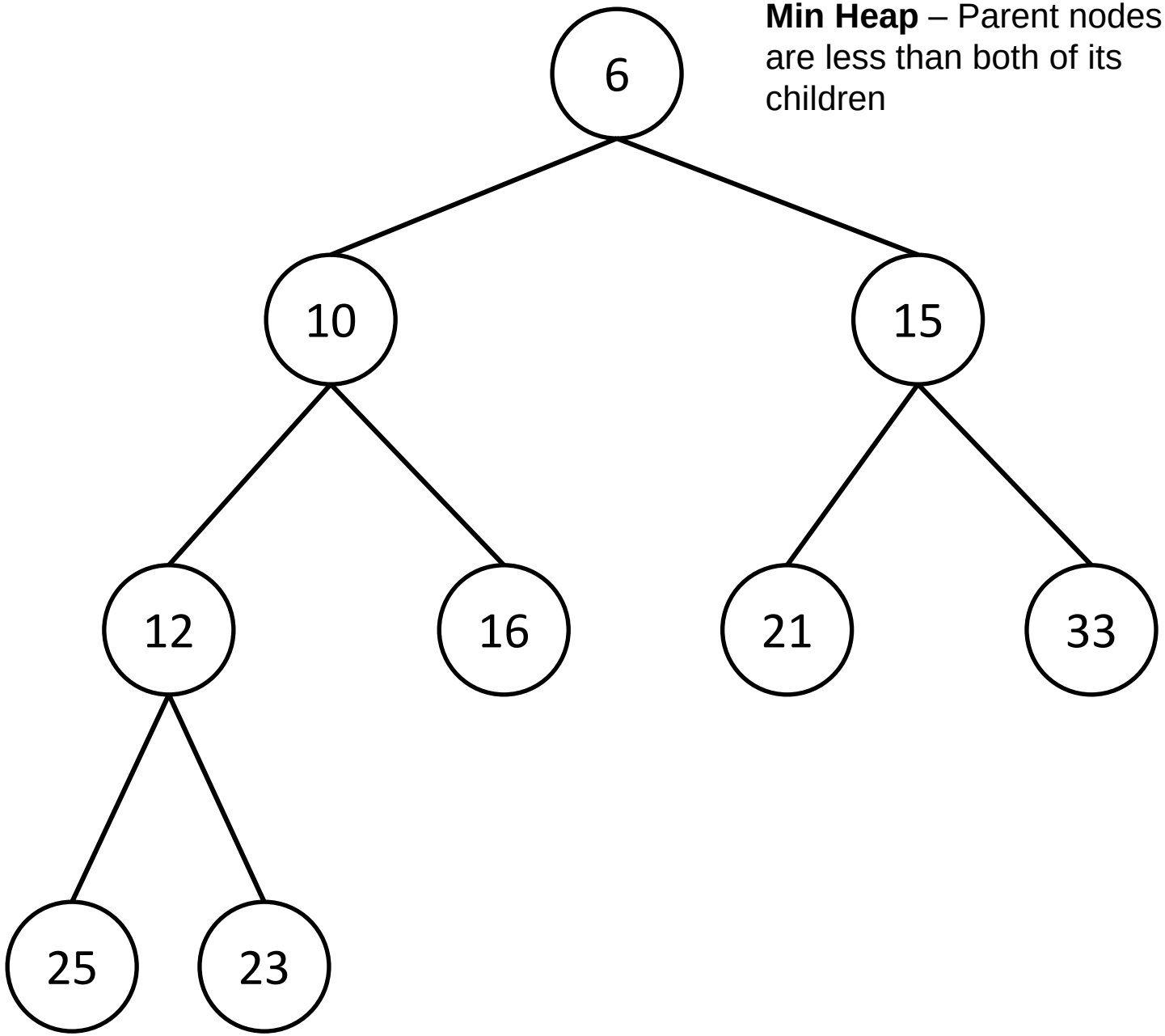


Min Heap – Parent nodes are less than both of its children



Heap Operations - Insert

```
add(7);
```

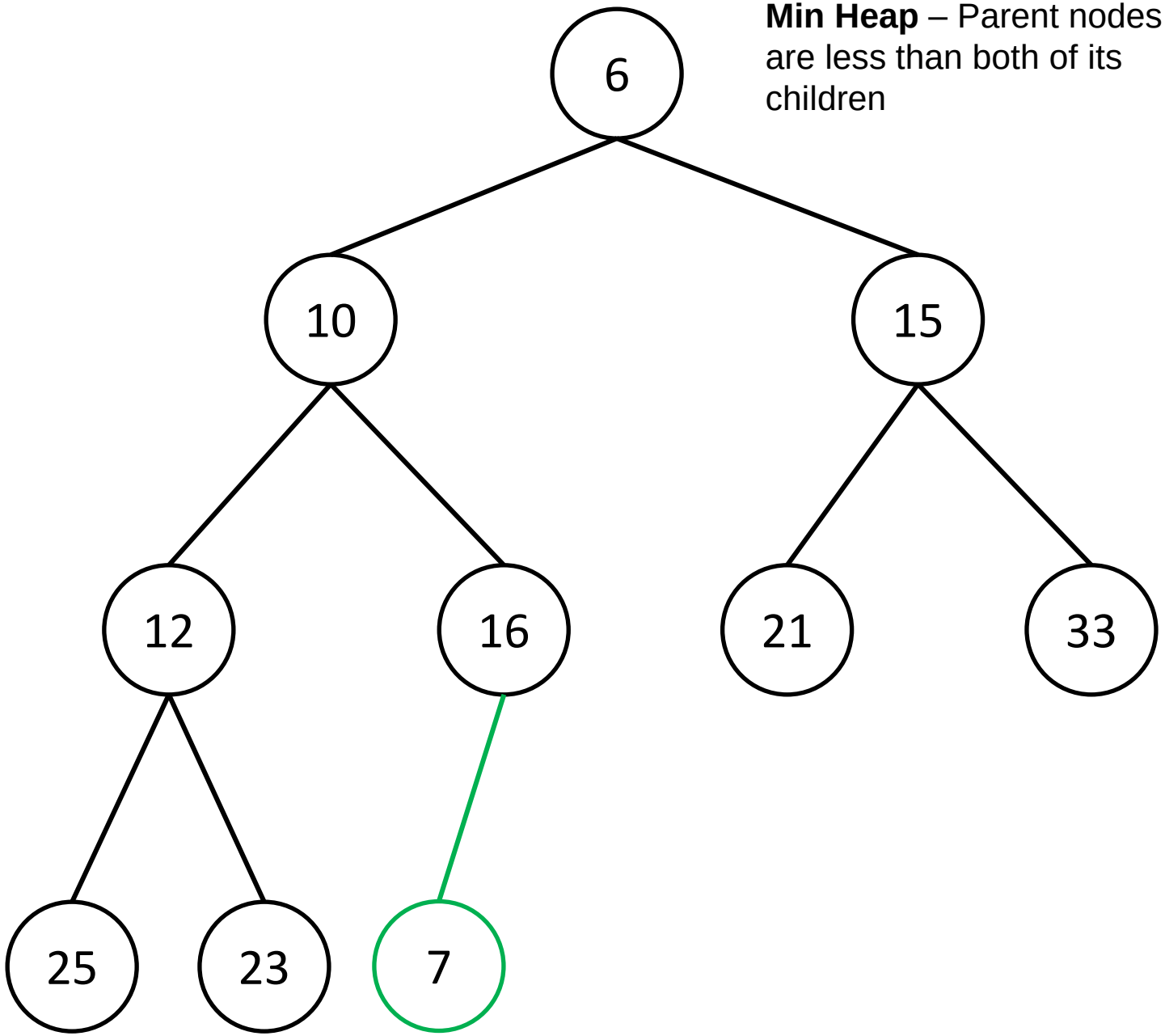


Min Heap – Parent nodes are less than both of its children

Heap Operations - Insert

```
add(7);
```

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



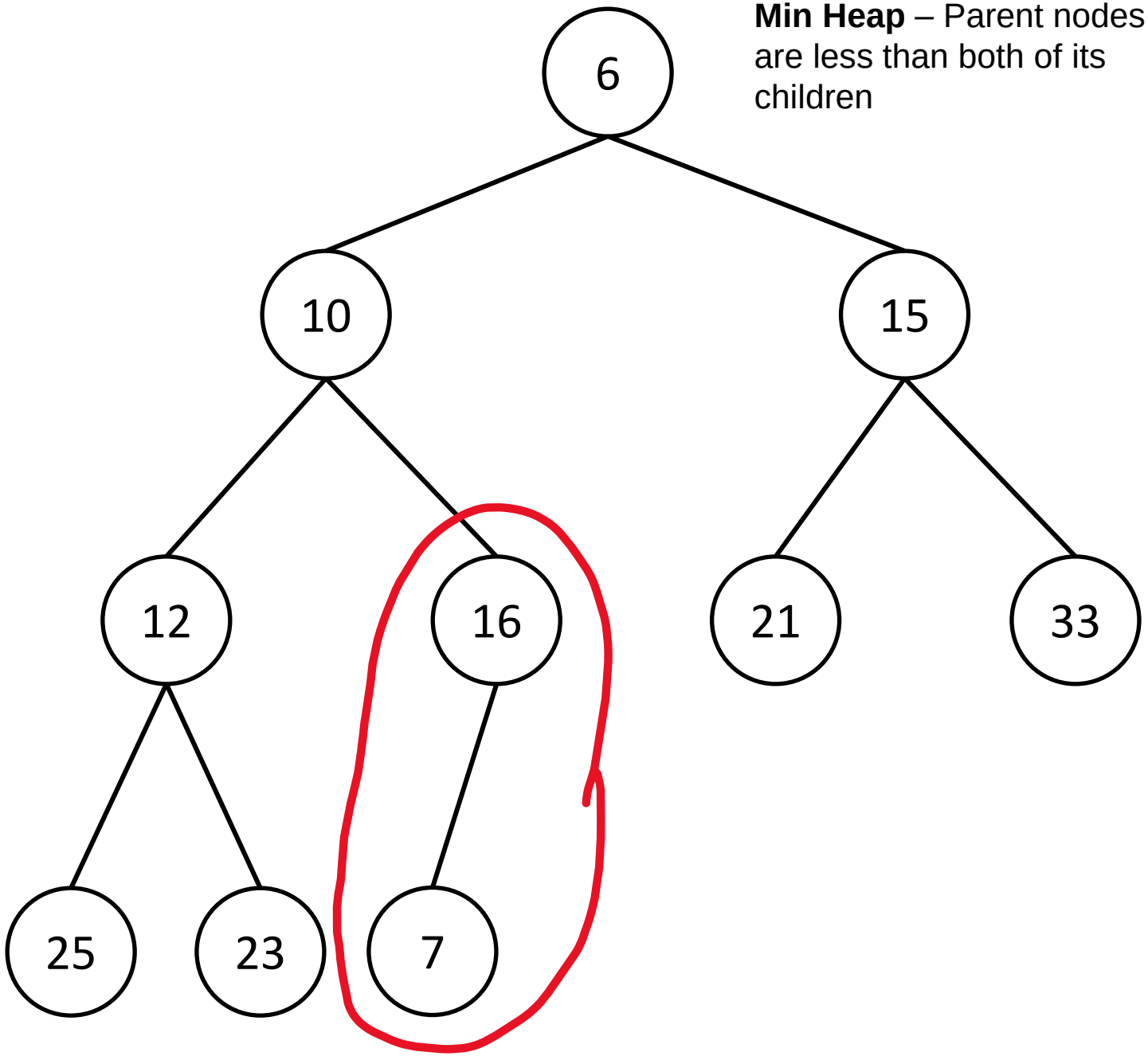
Min Heap – Parent nodes are less than both of its children

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

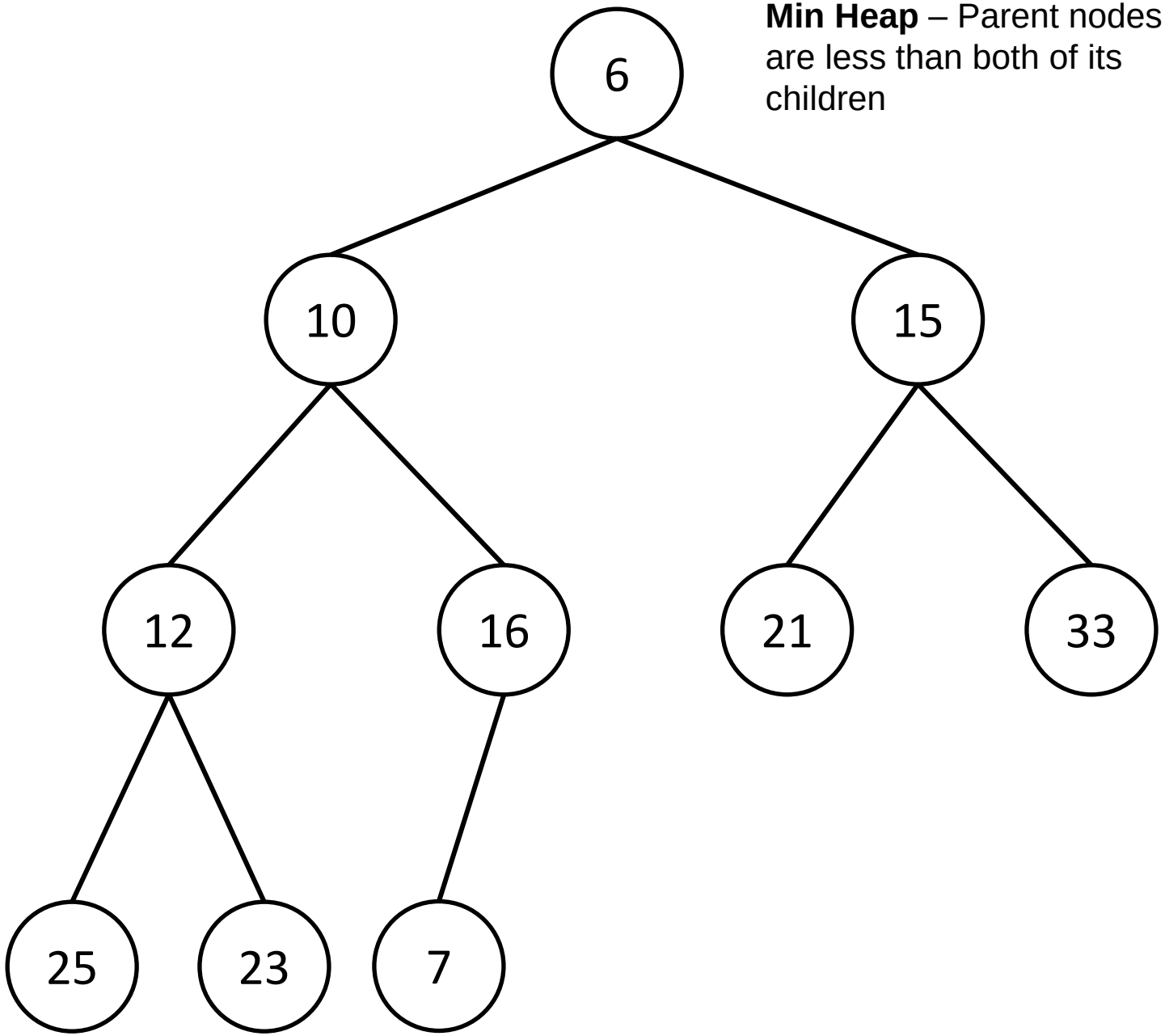
Heap Operations - Insert

```
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

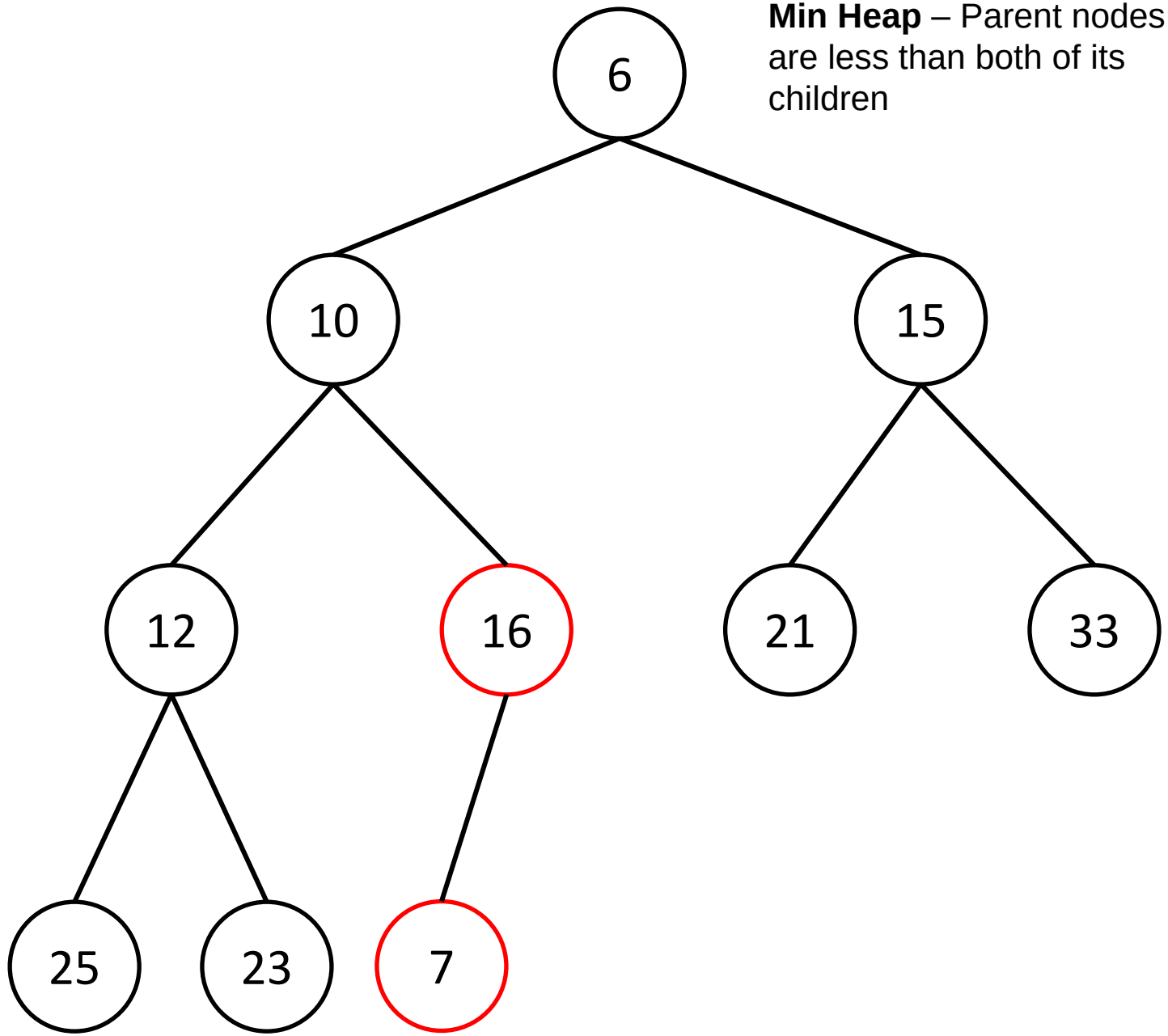
Heap Operations - Insert

```
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

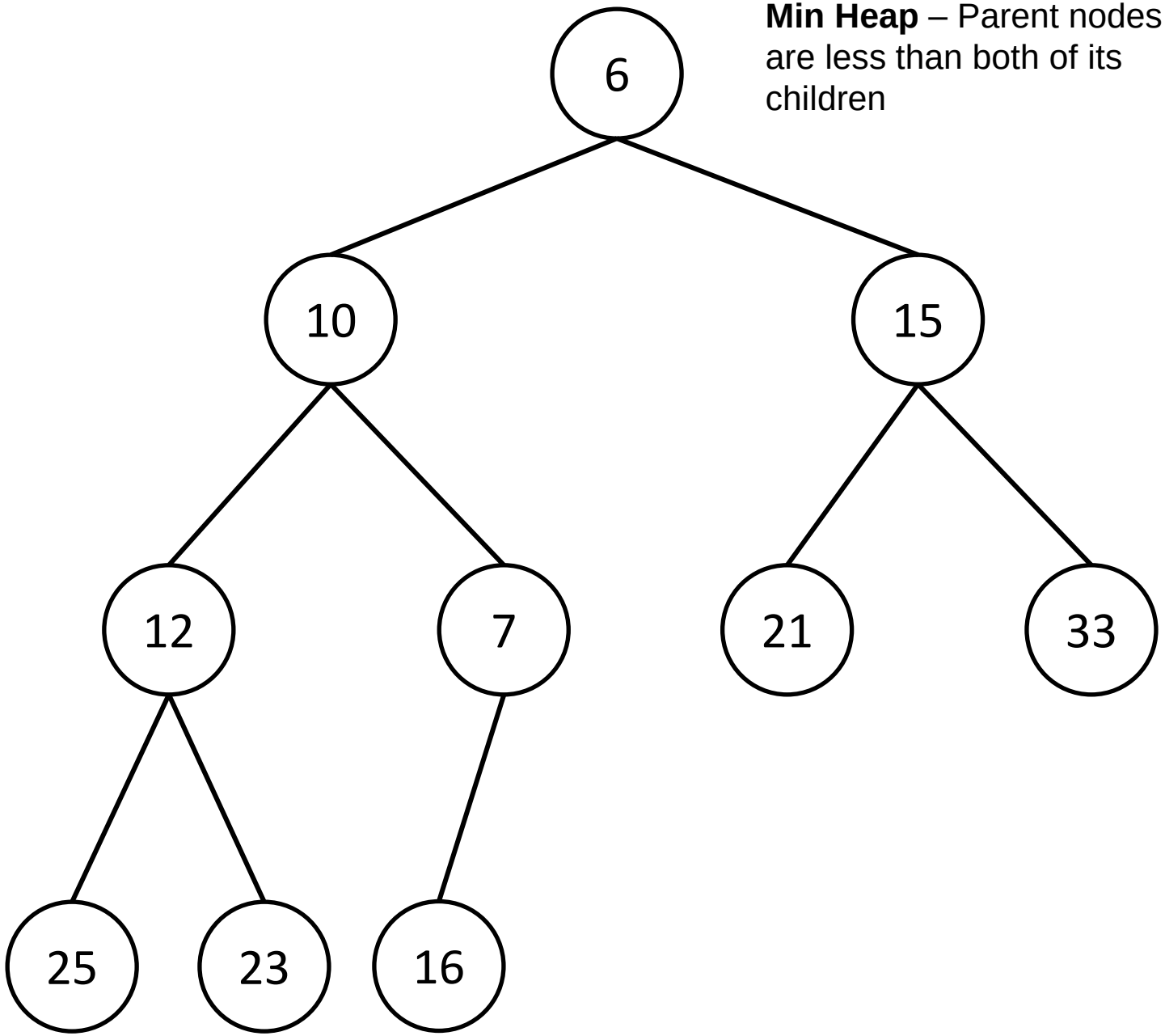
Heap Operations - Insert

```
add(7);
```

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

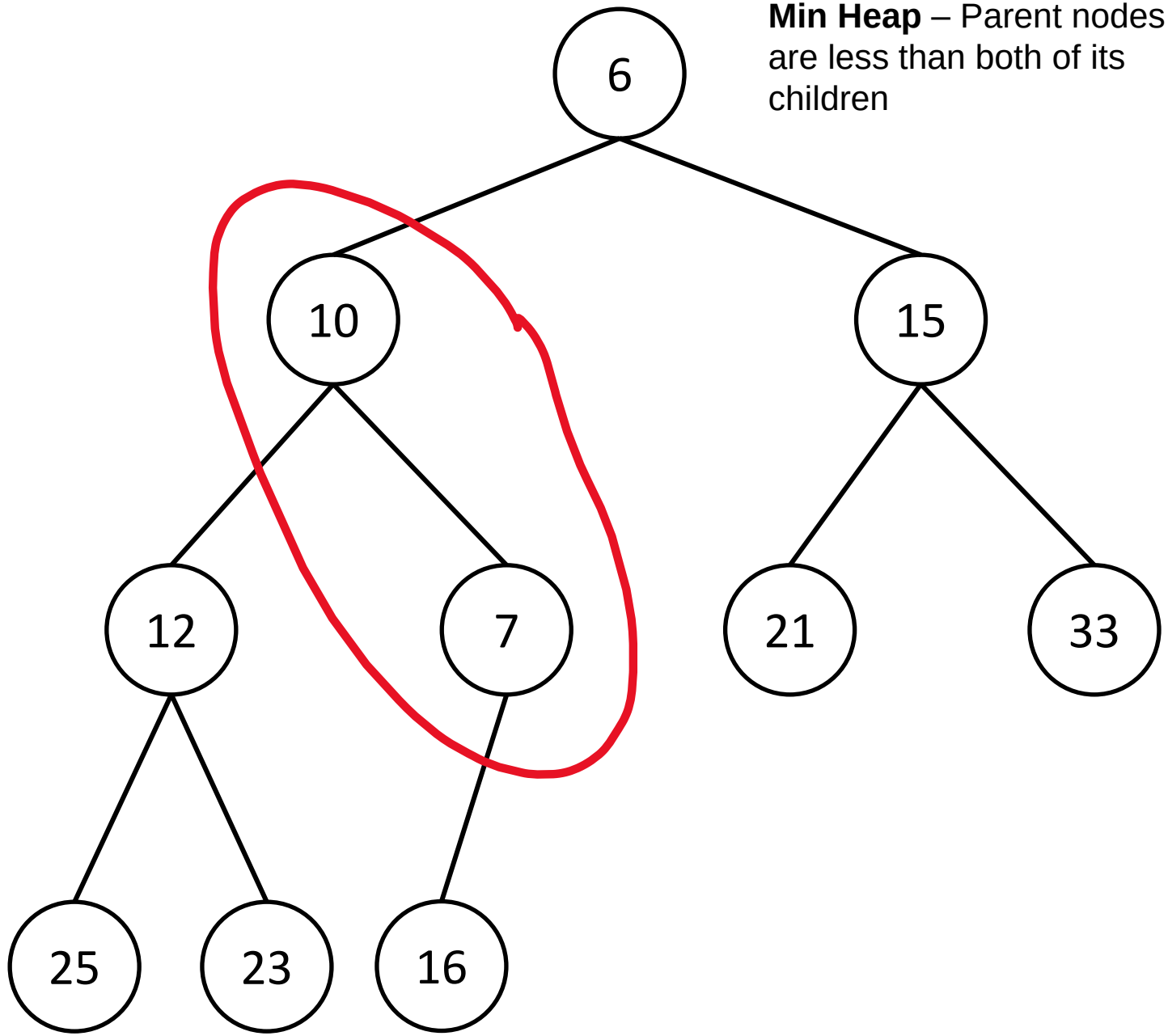
Heap Operations - Insert

```
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

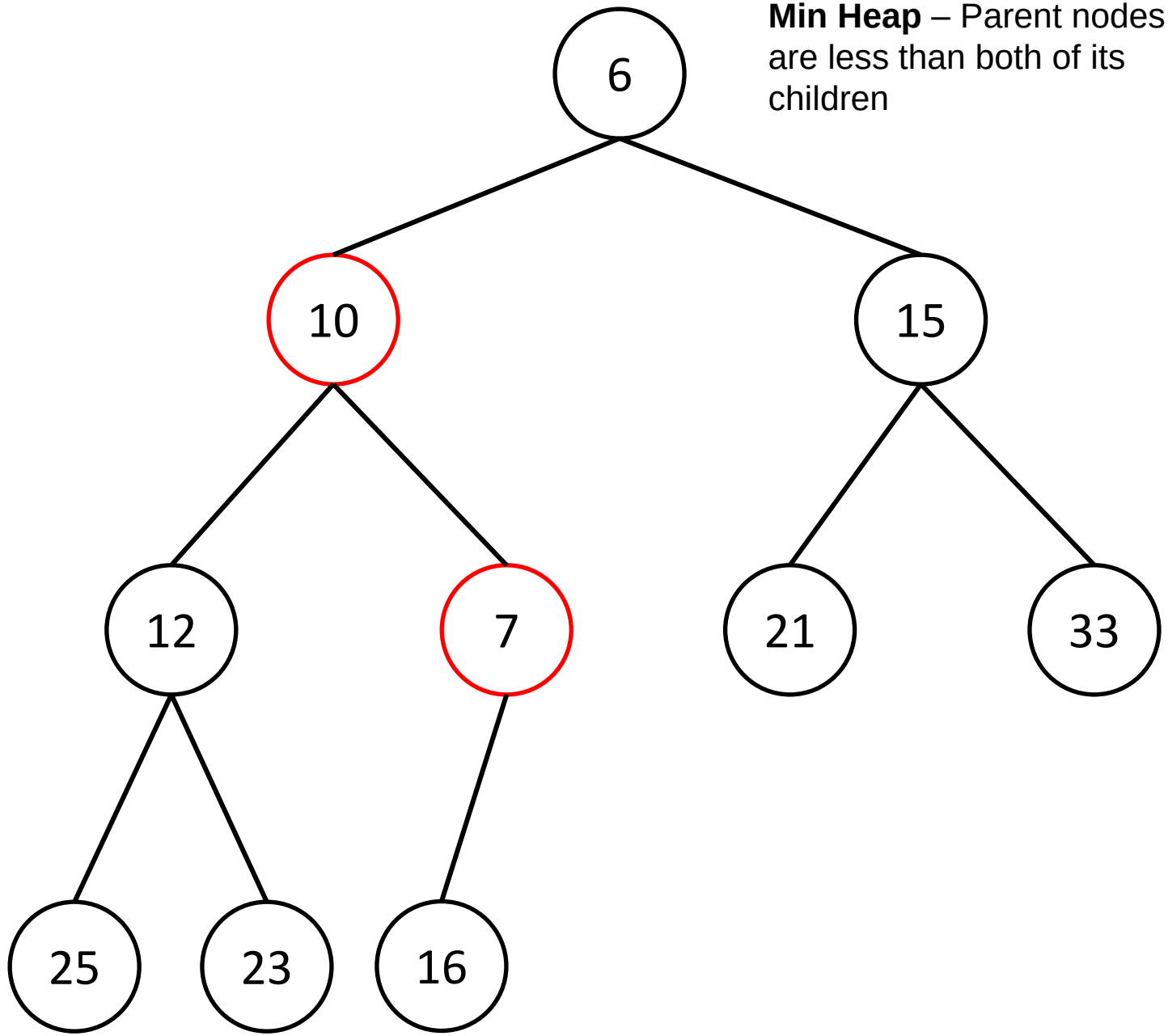
Heap Operations - Insert

```
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);
```

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

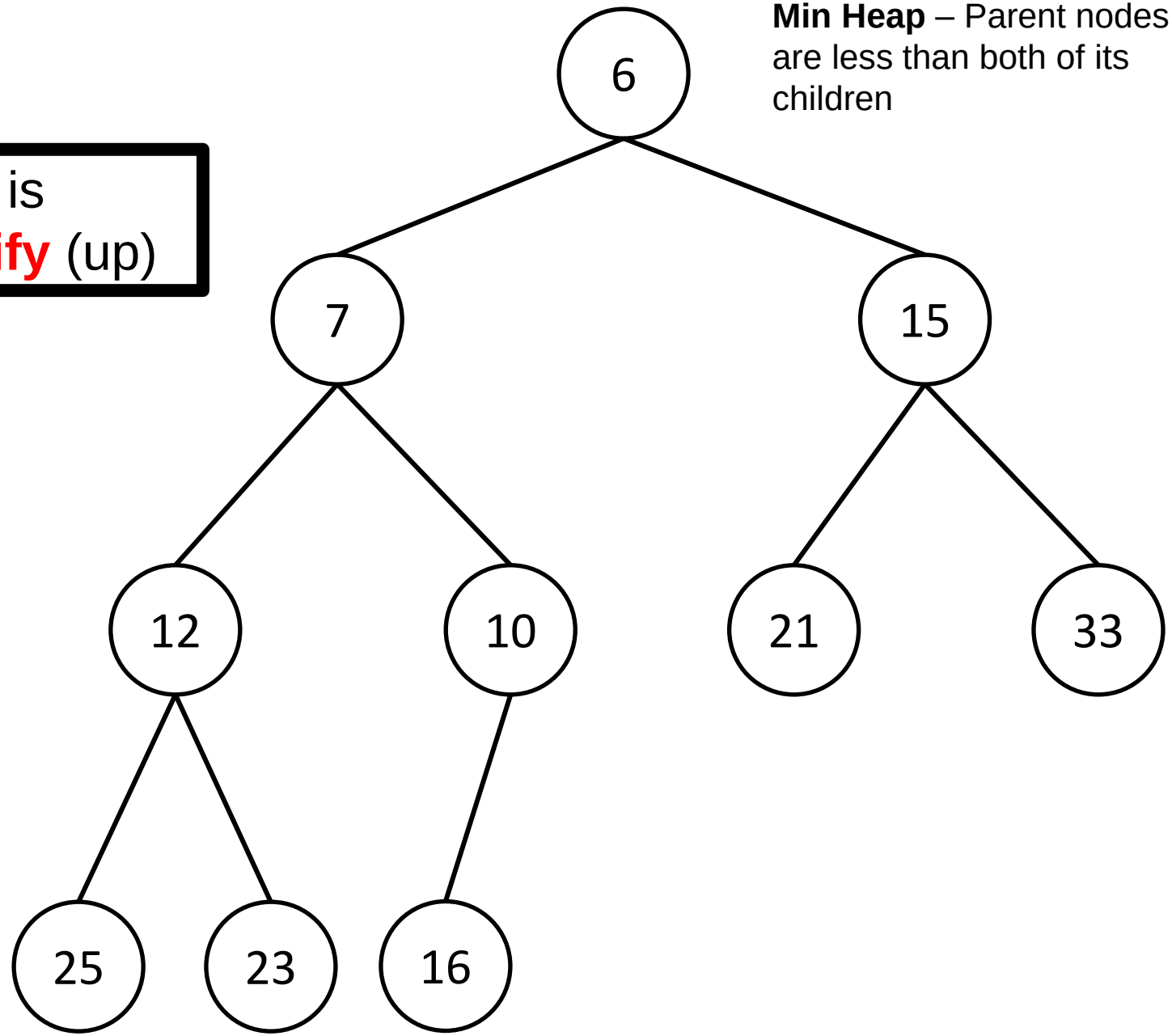
Min Heap – Parent nodes are less than both of its children

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

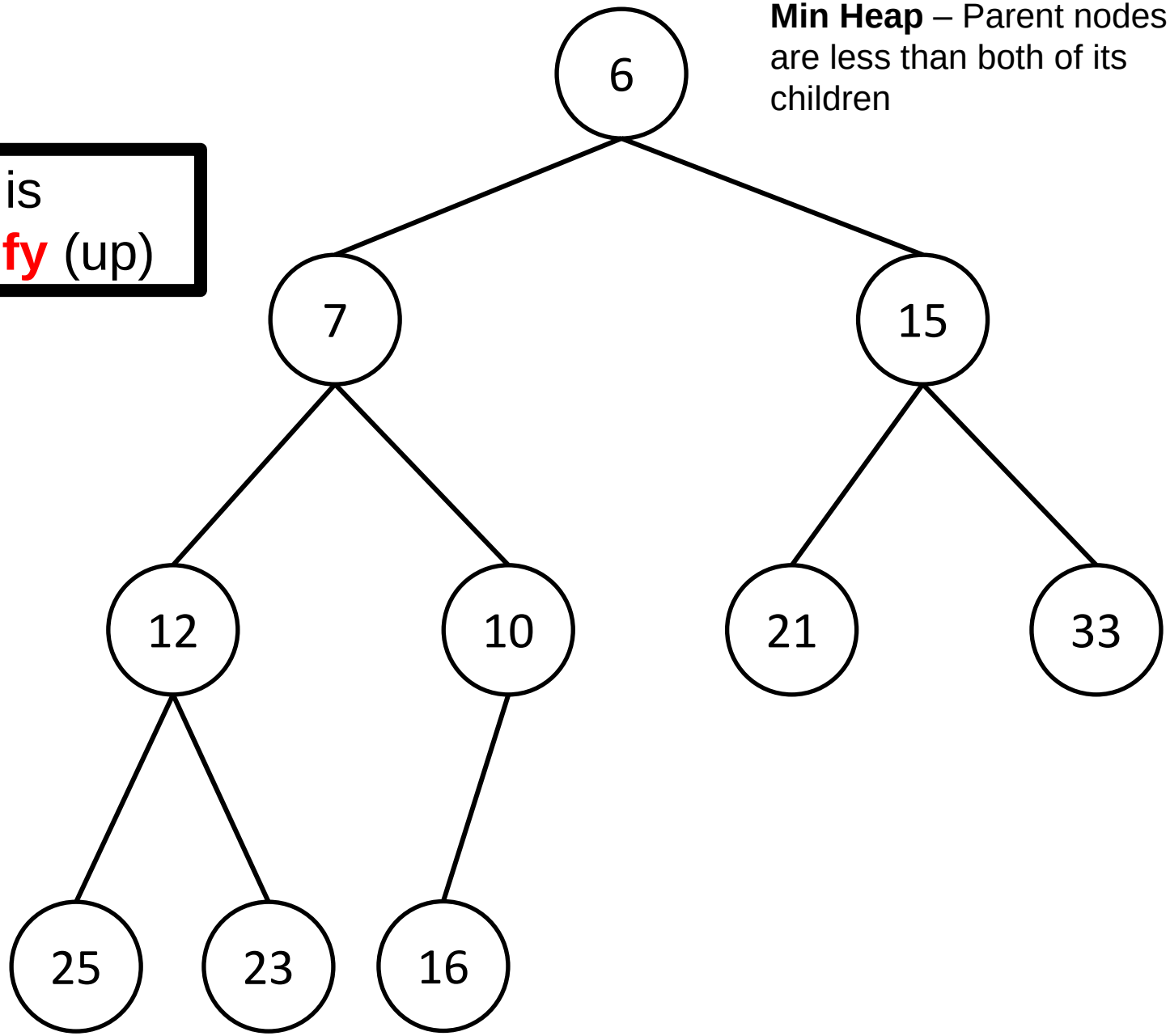
When new nodes are added, we may need to move it up in the tree

Heap Operations - Insert

```
add(7);

add(14);
```

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



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

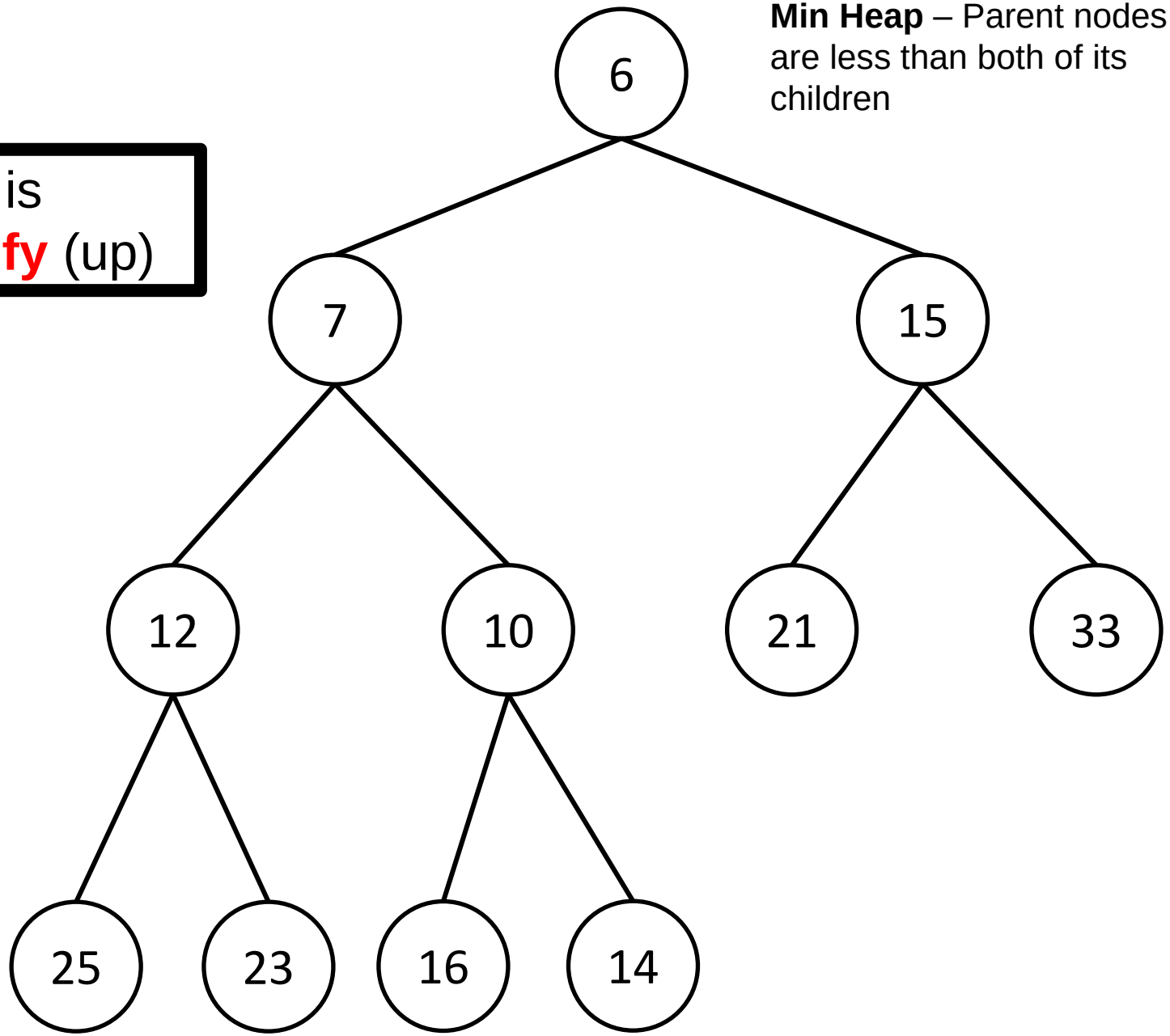
Heap Operations - Insert

```
add(7);

add(14);
```

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

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



Heap Operations - Insert

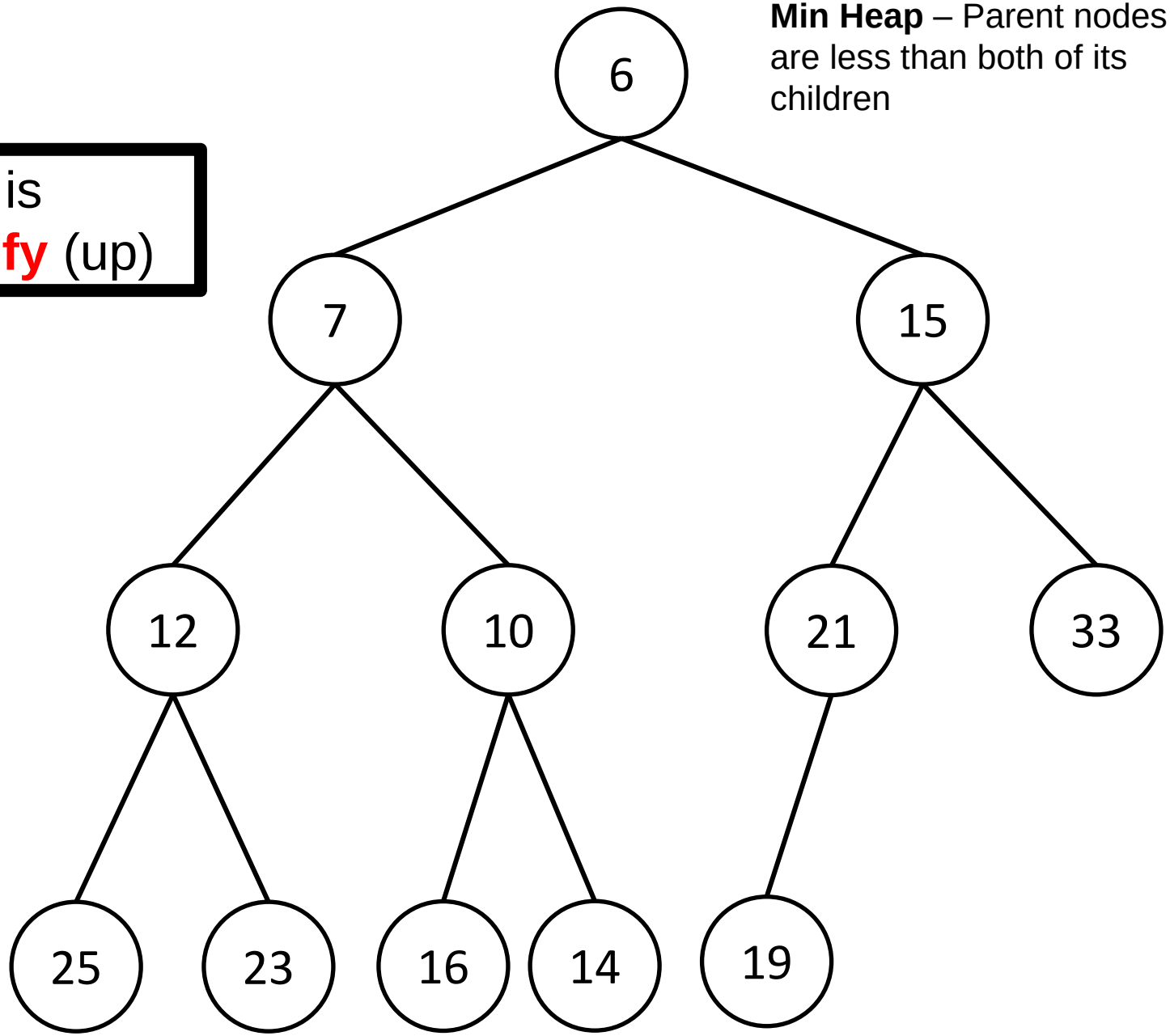
```
add(7);

add(14);

add(19);
```

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

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



Heap Operations - Insert

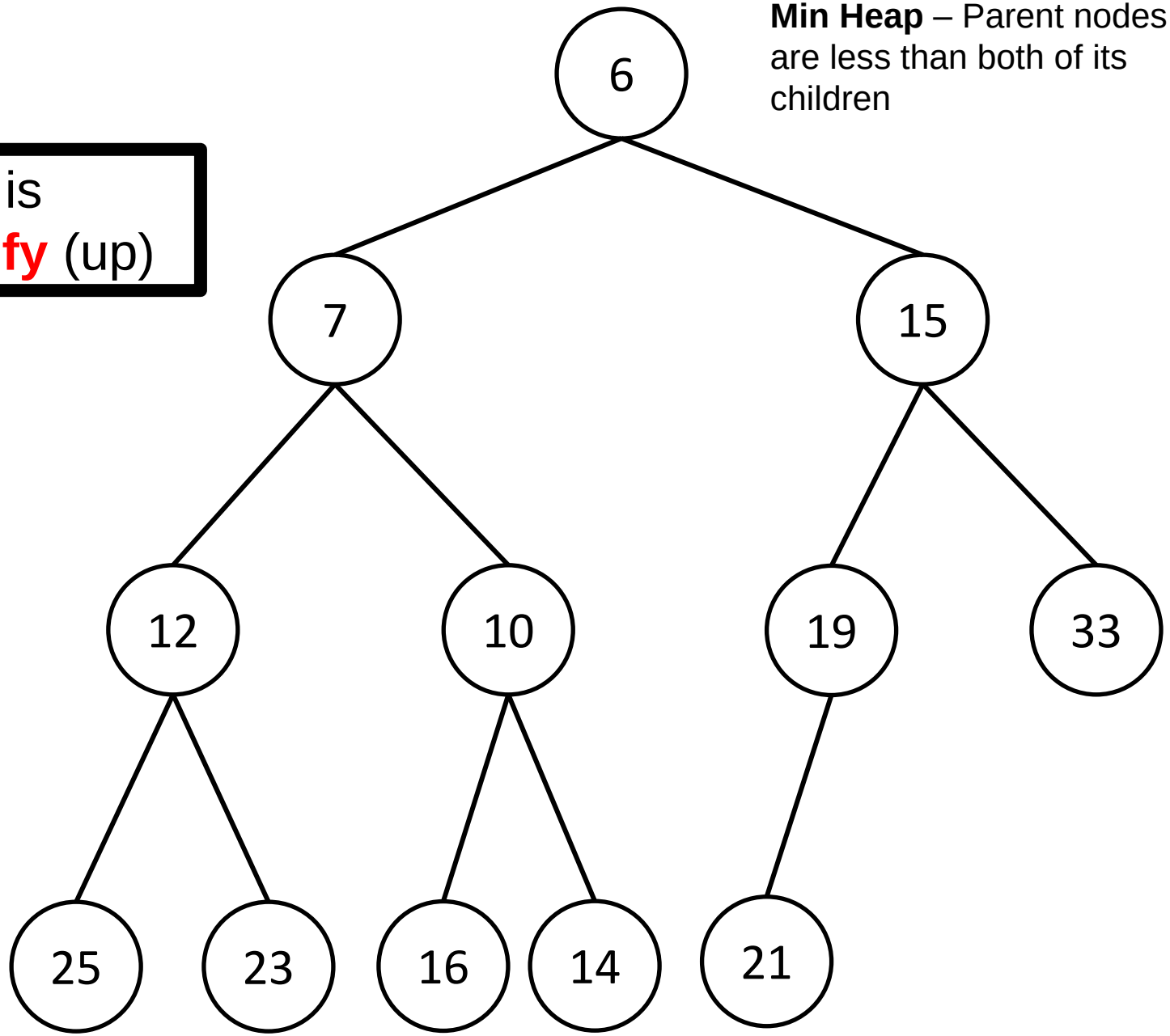
```
add(7);

add(14);

add(19);
```

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

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



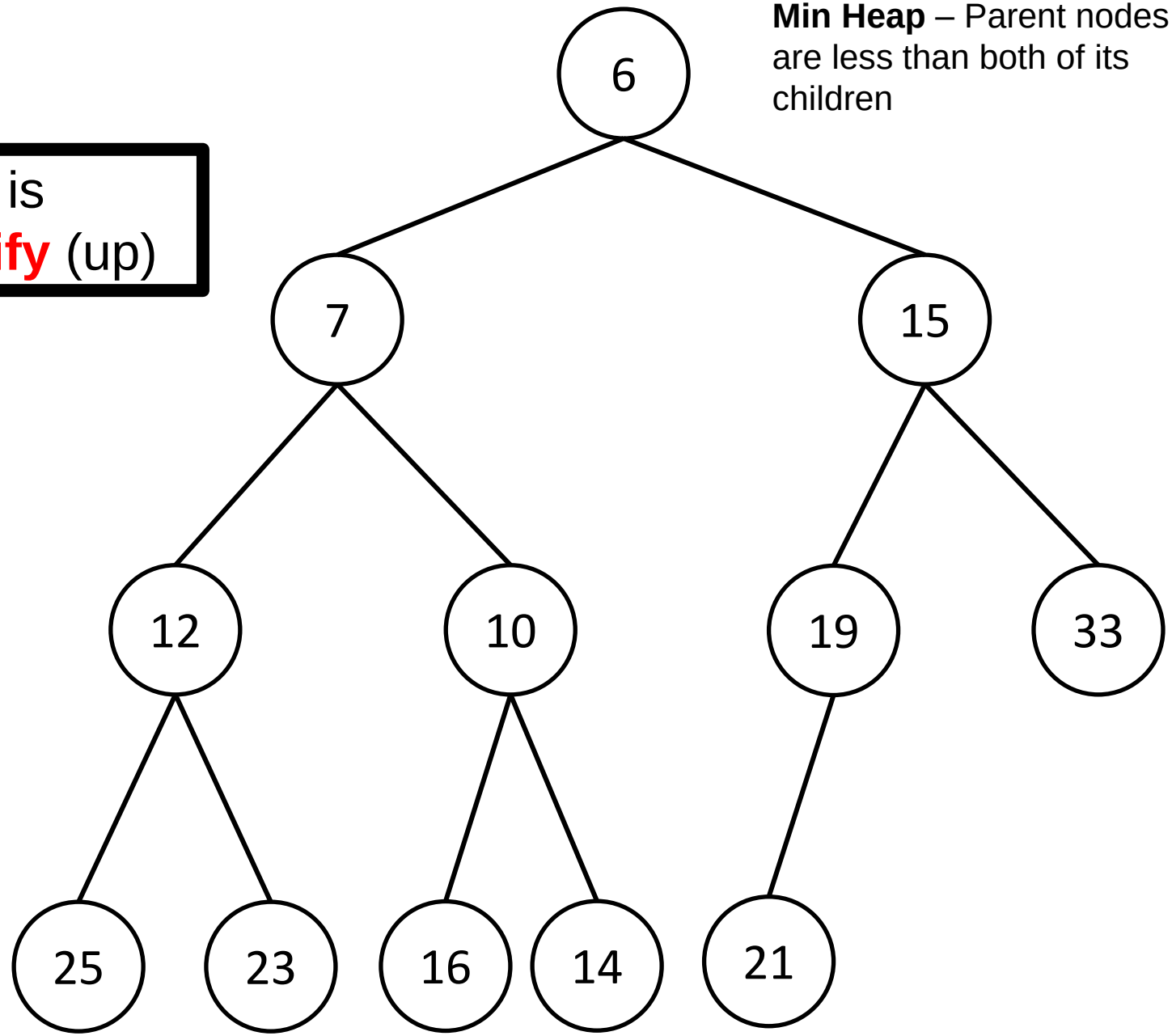
Heap Operations - Insert

```
add(7);

add(14);

add(19);
```

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



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

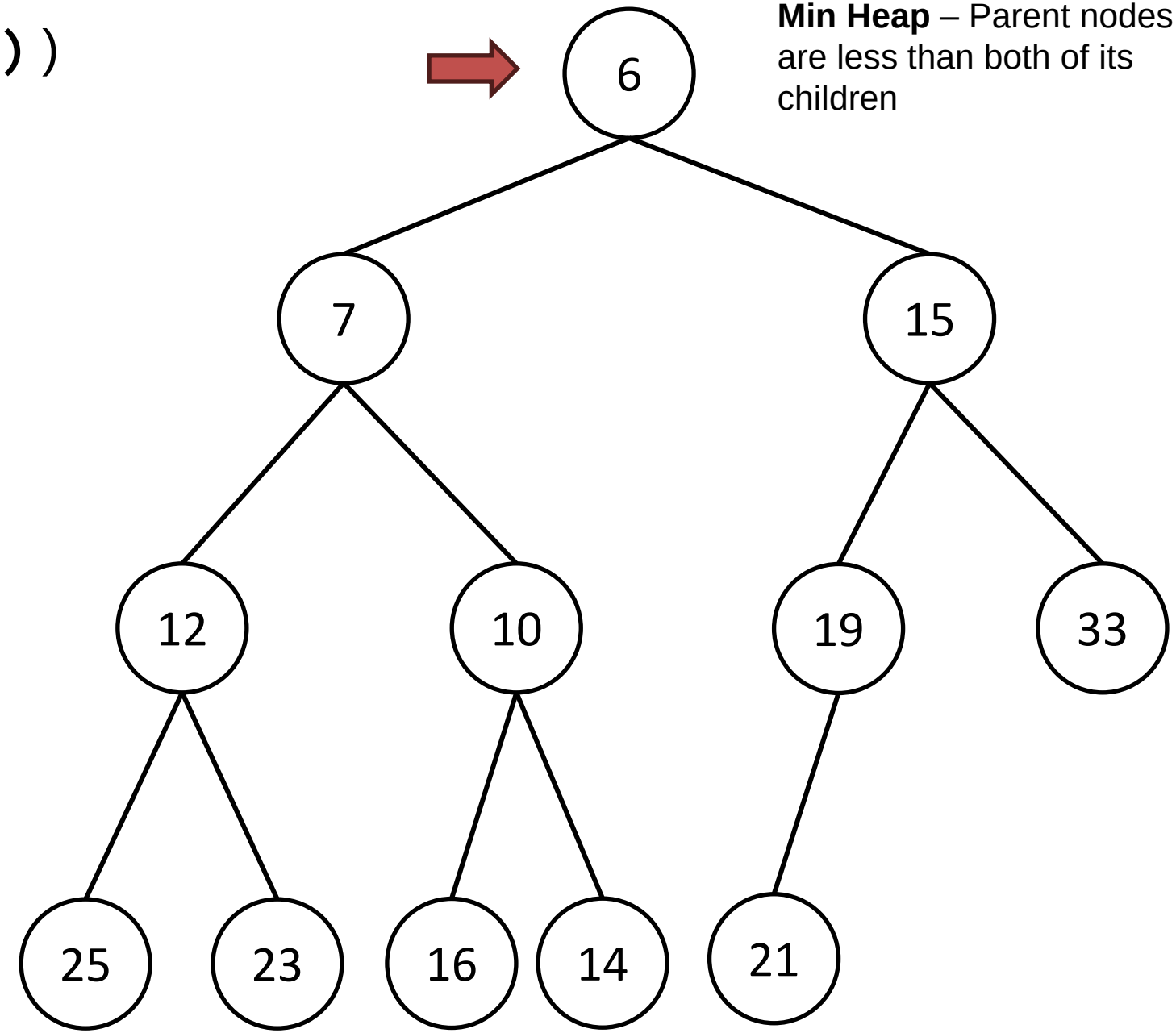
Running time?

- 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

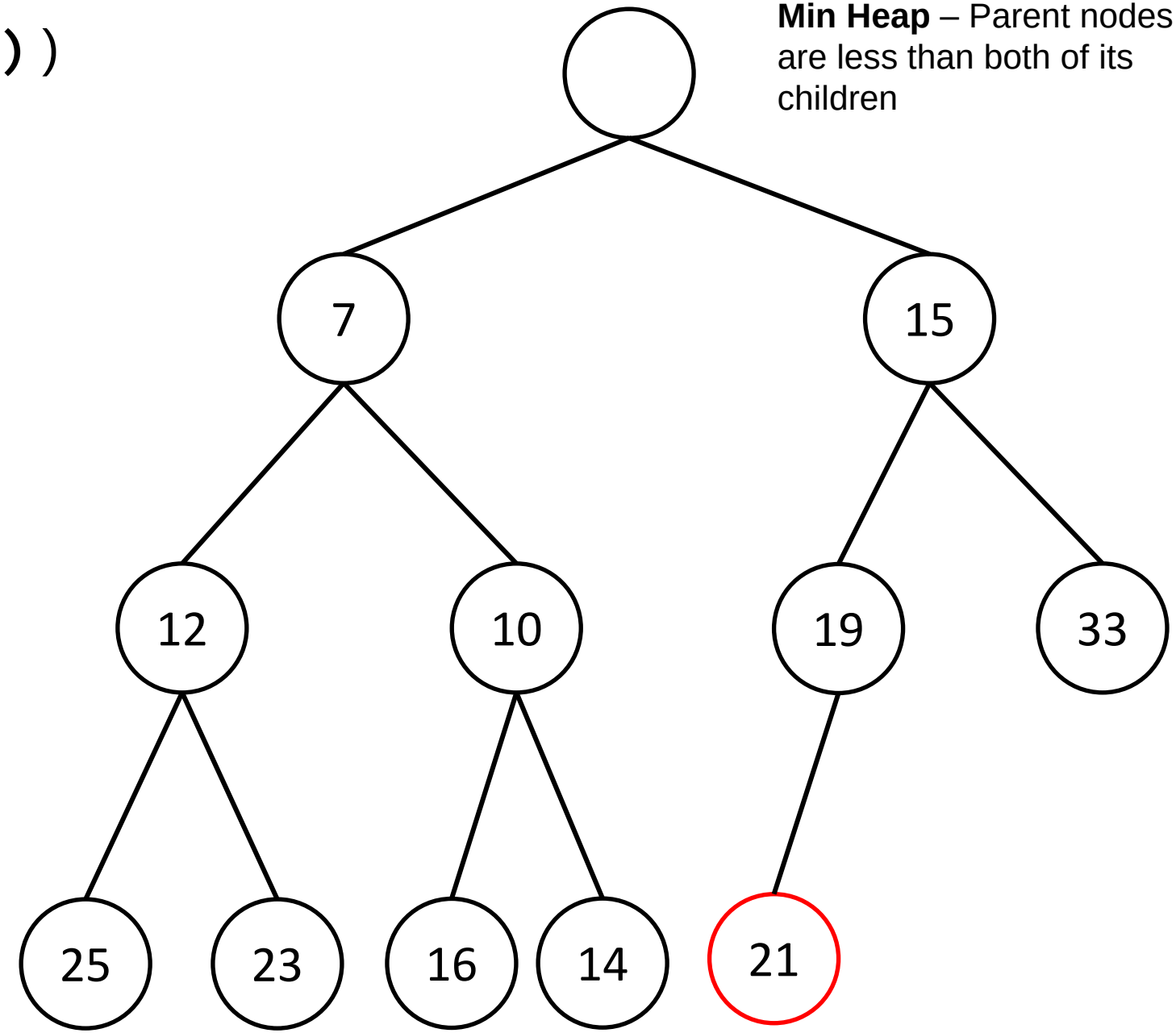


Min Heap – Parent nodes are less than both of its children

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**

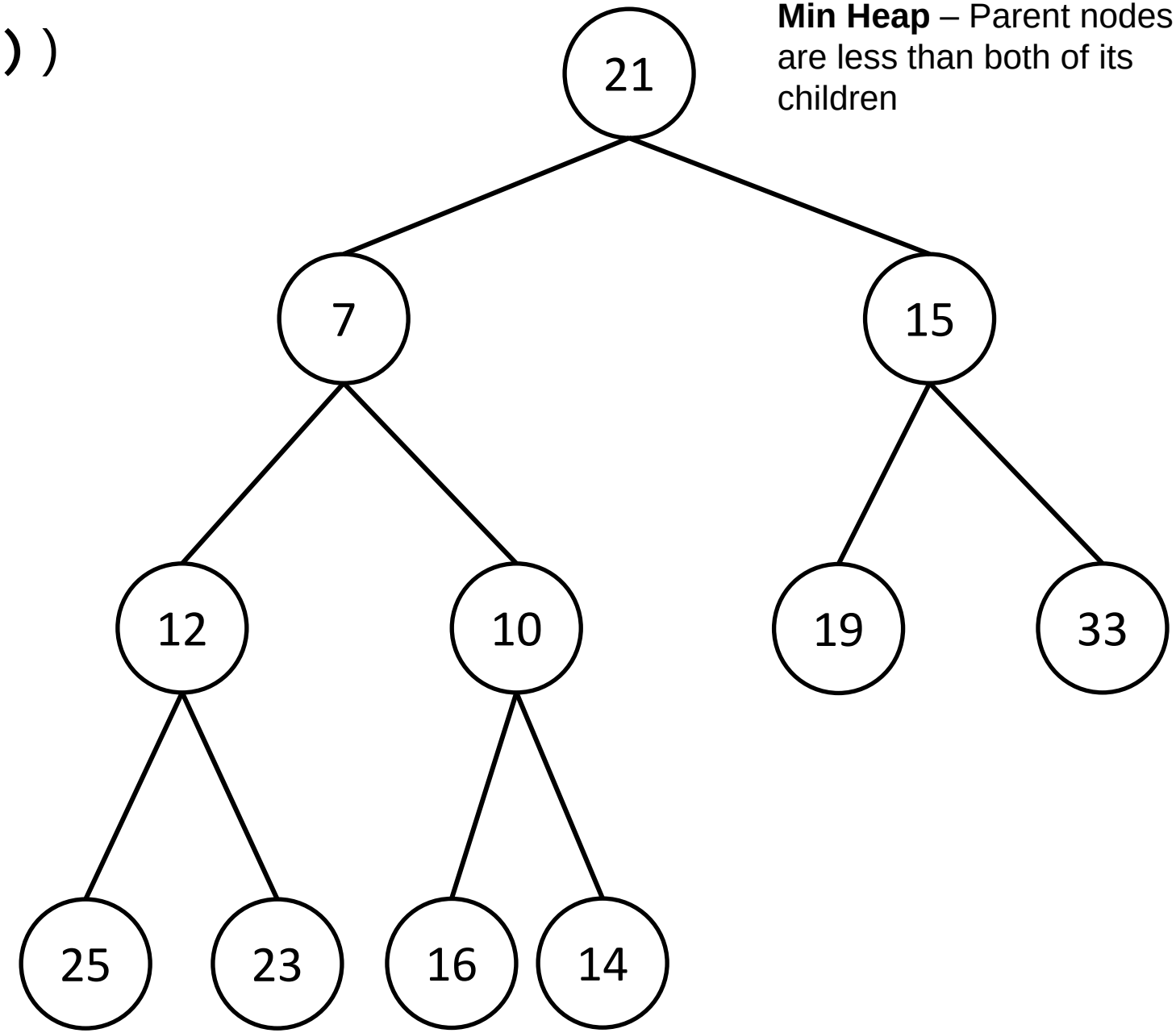


Min Heap – Parent nodes are less than both of its children

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

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**

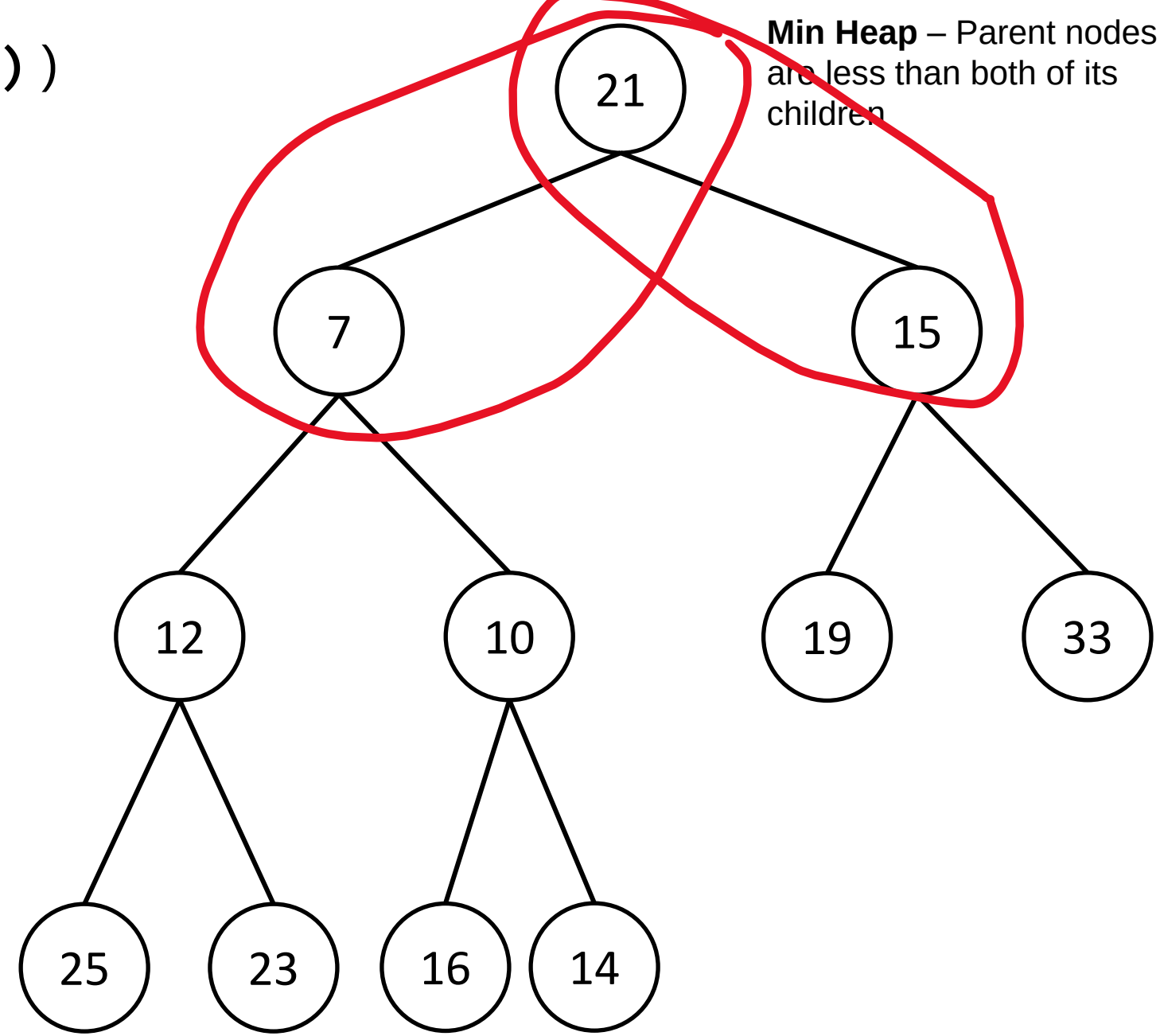


Min Heap – Parent nodes are less than both of its children

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**

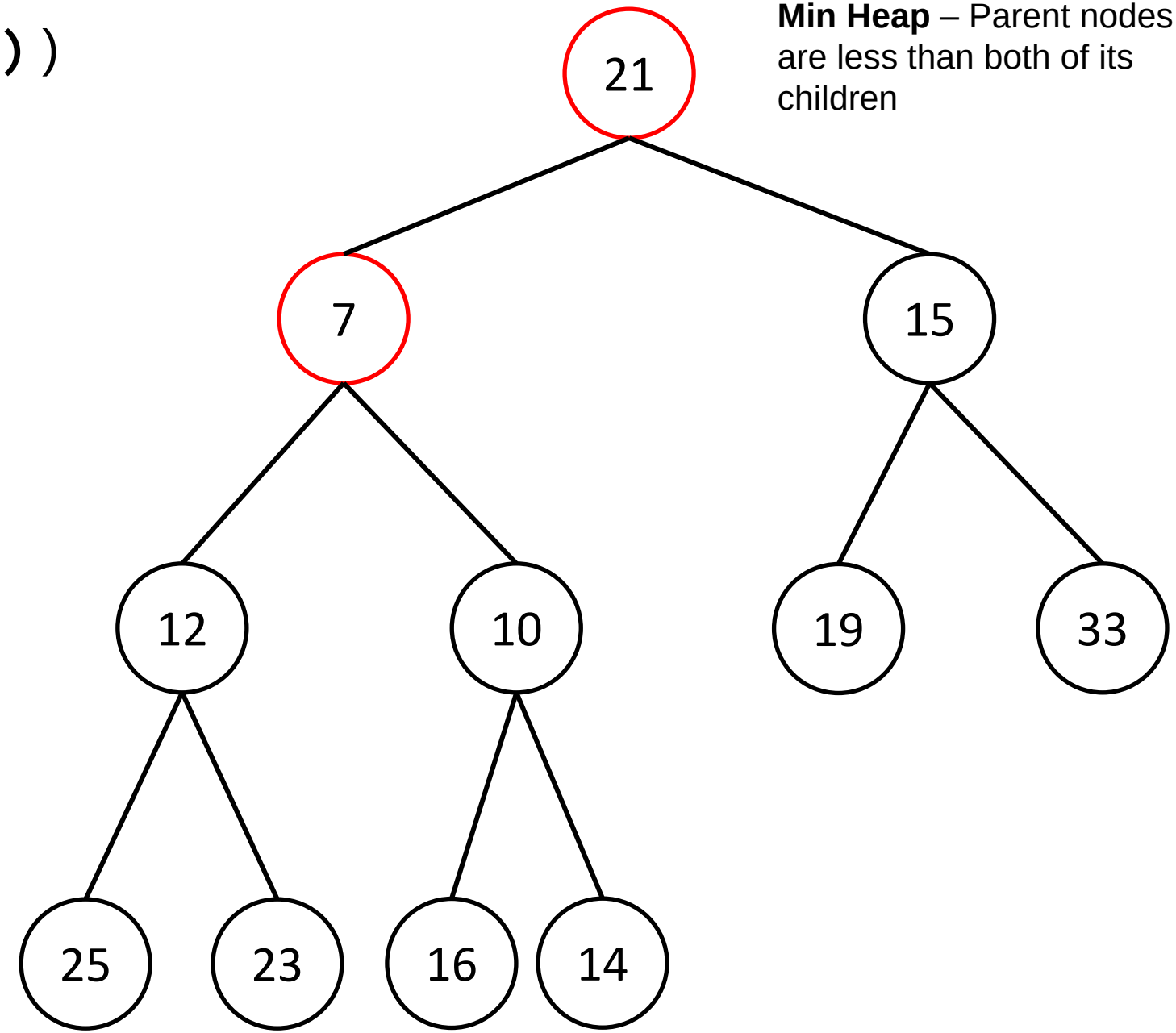


Heap Operations – Removal (**`poll()`**)

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



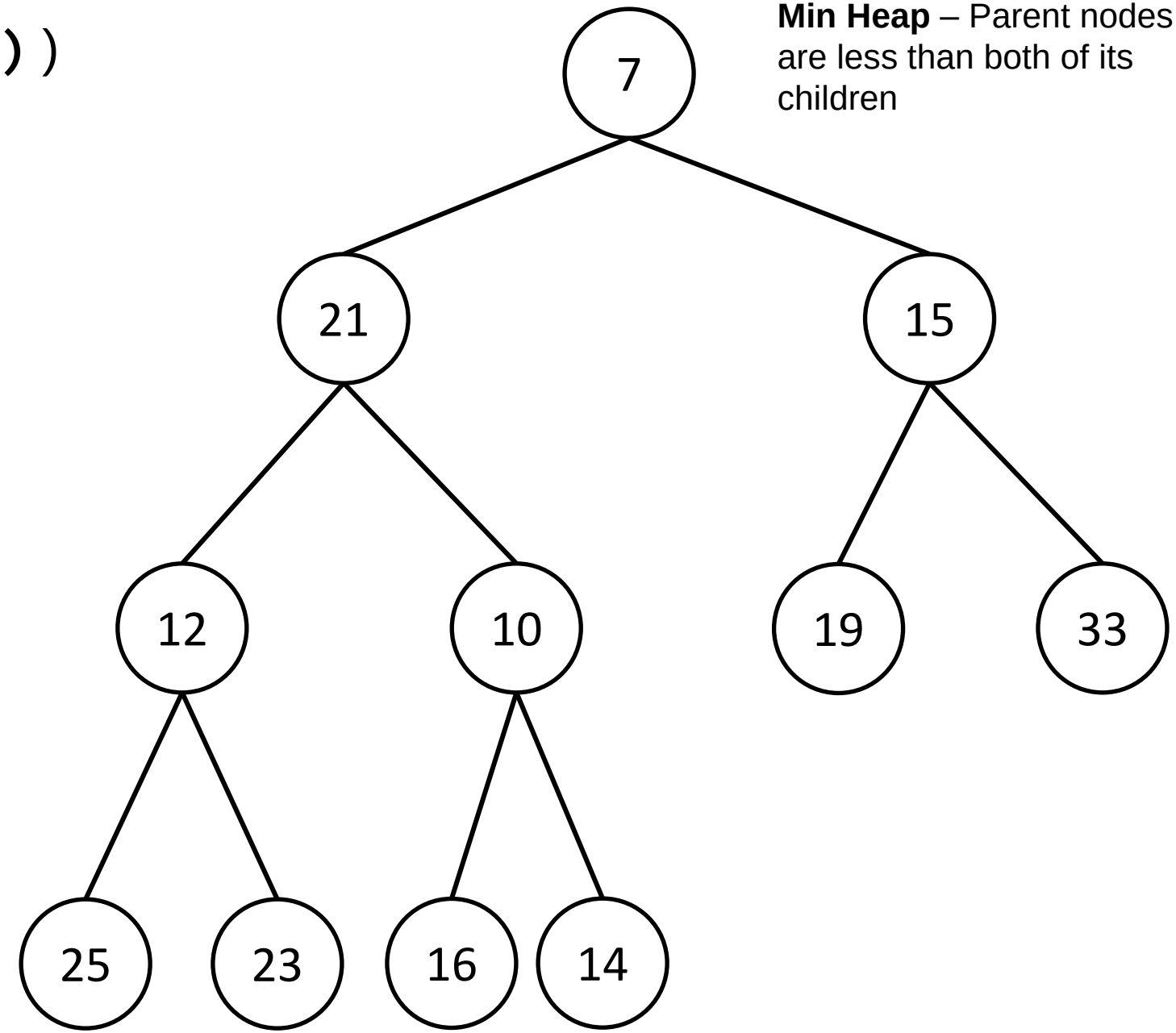
Min Heap – Parent nodes are less than both of its children

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



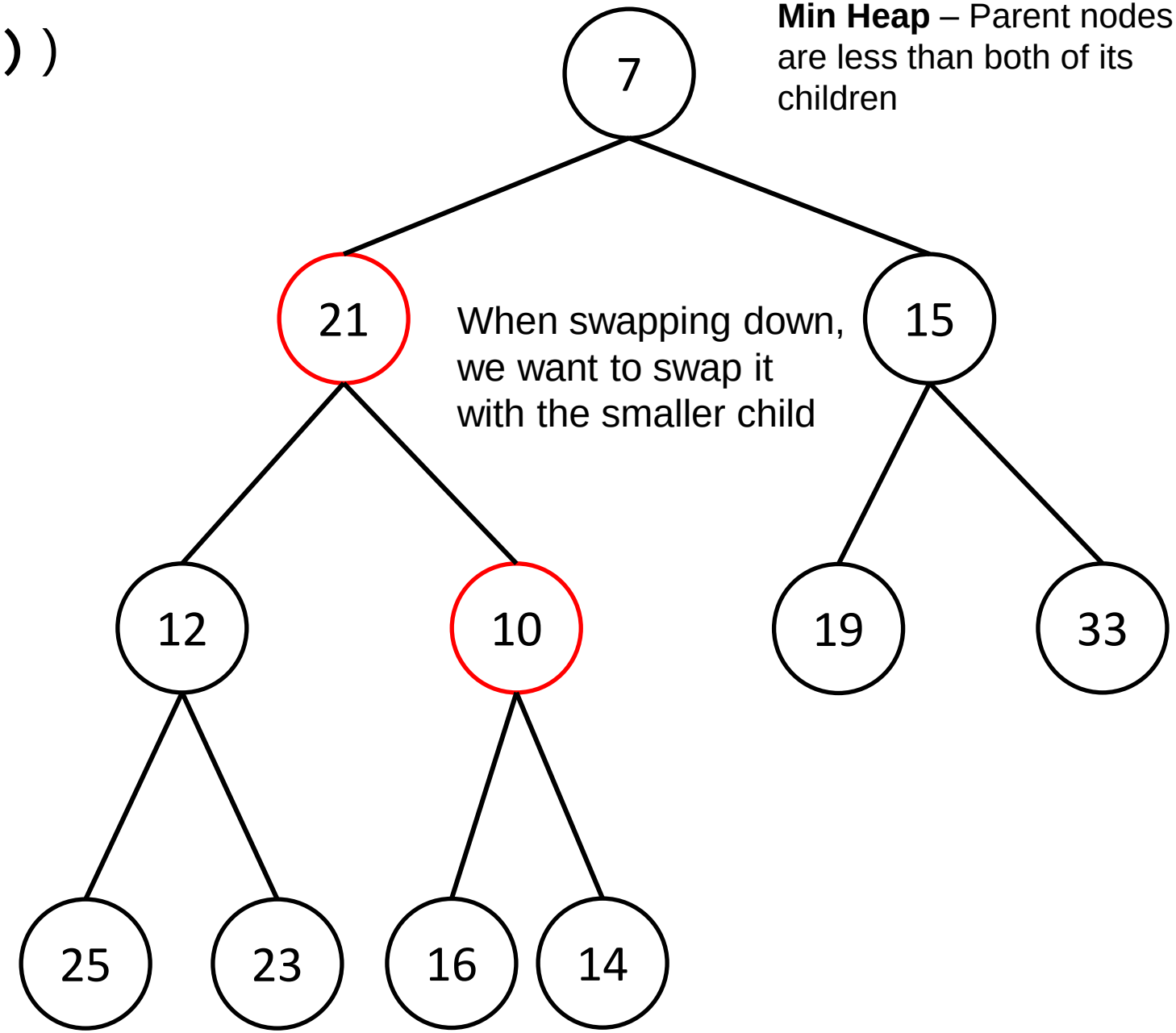
Min Heap – Parent nodes are less than both of its children

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



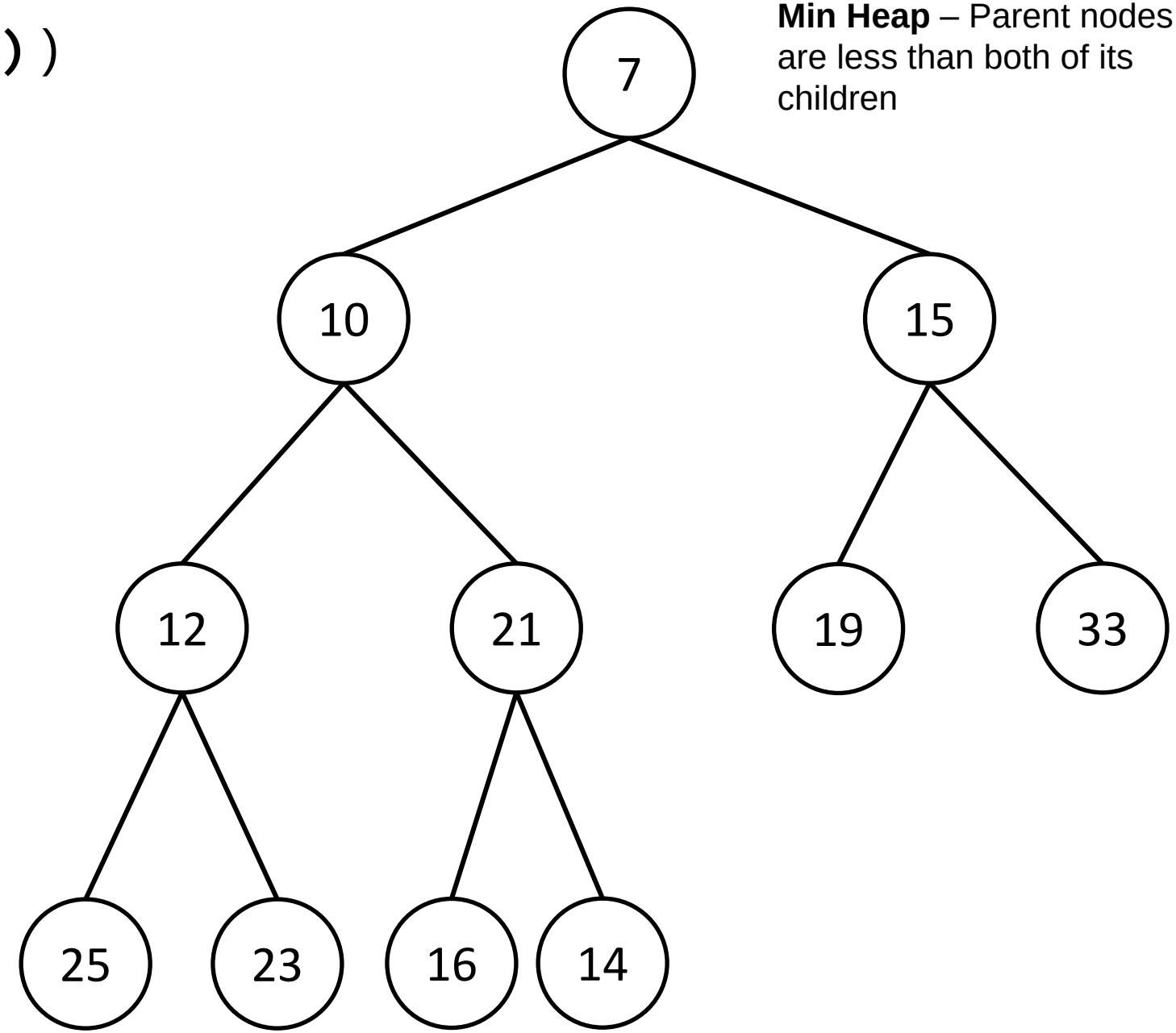
Min Heap – Parent nodes are less than both of its children

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



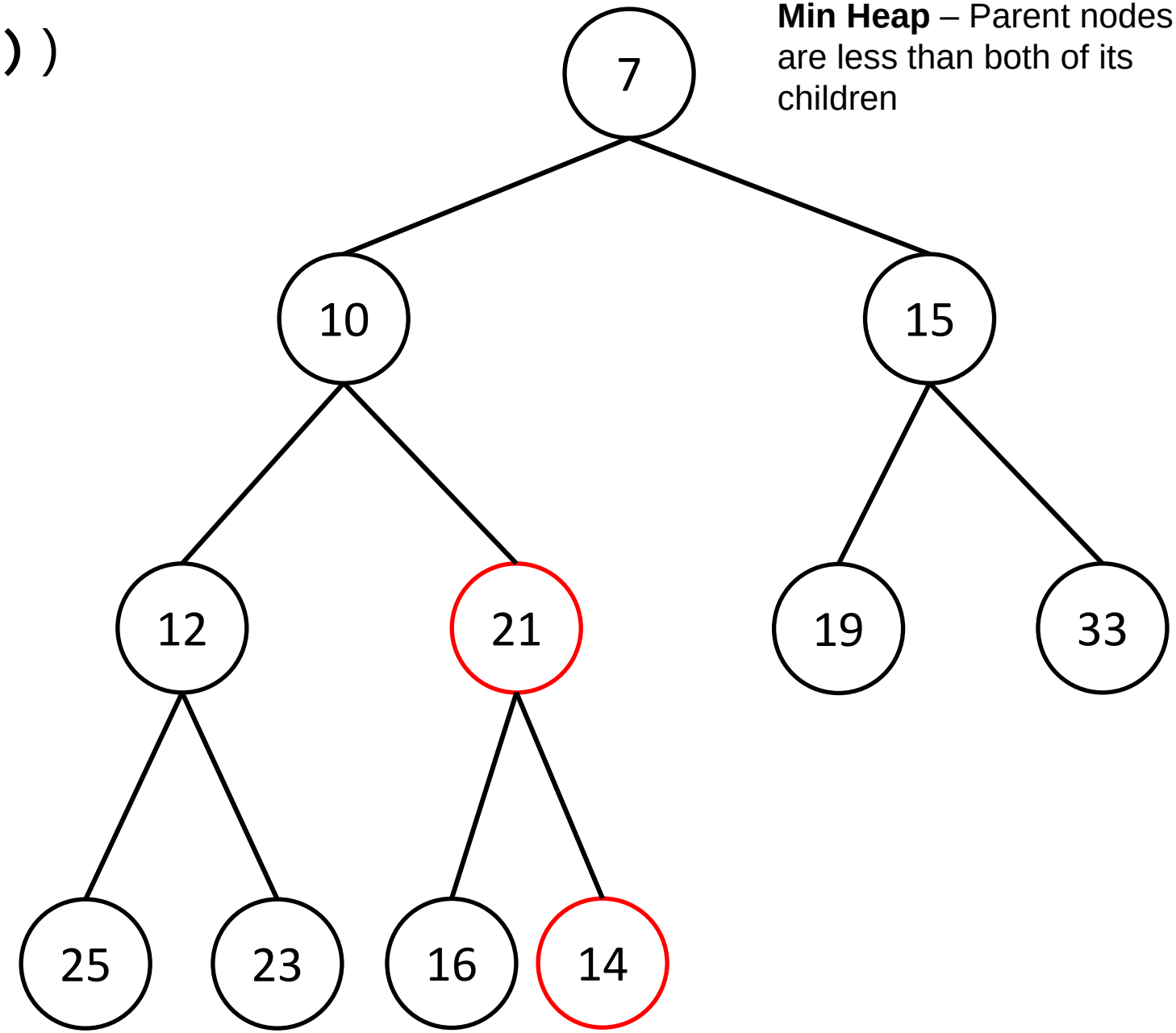
Min Heap – Parent nodes are less than both of its children

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



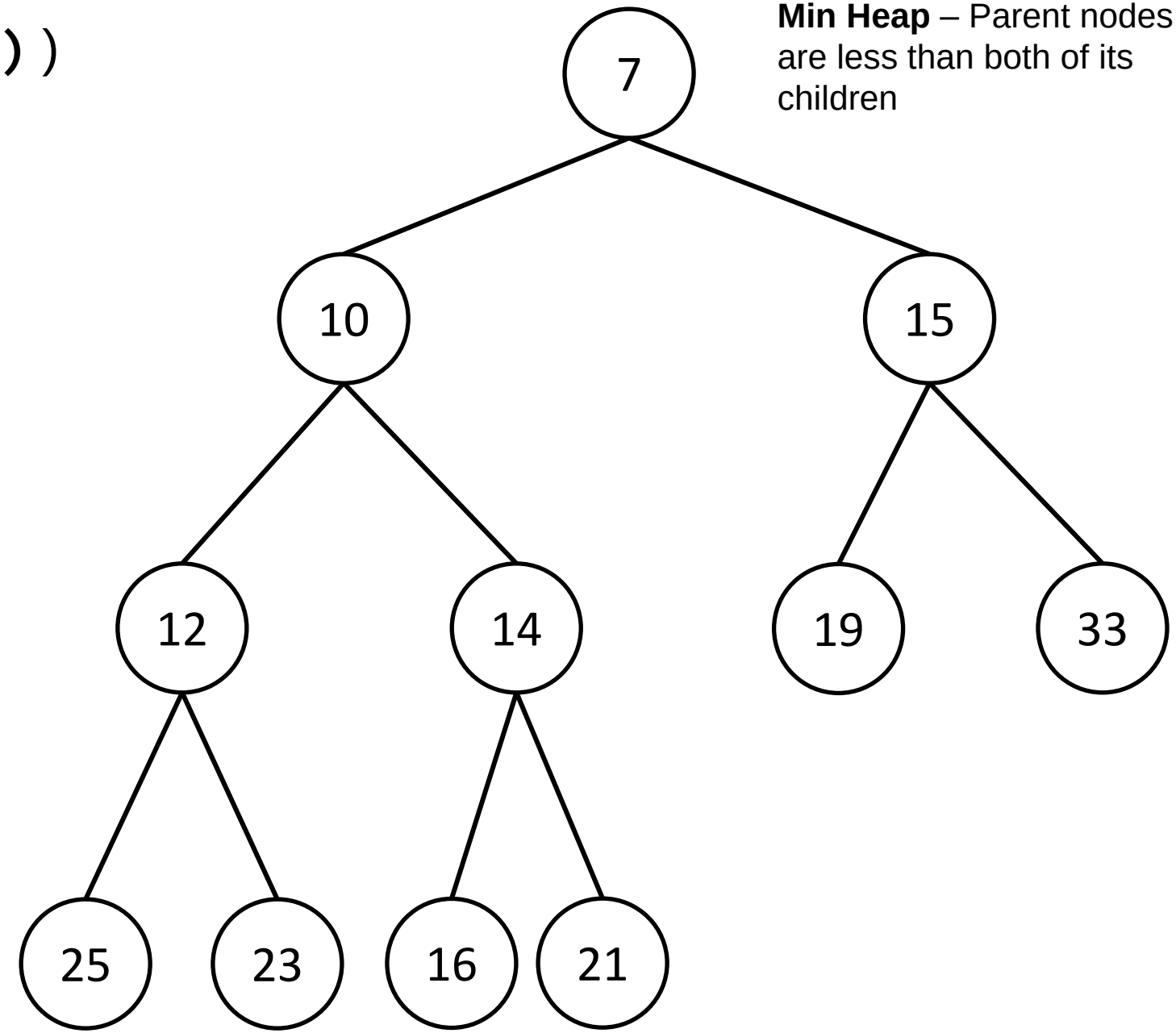
Min Heap – Parent nodes are less than both of its children

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

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()`)

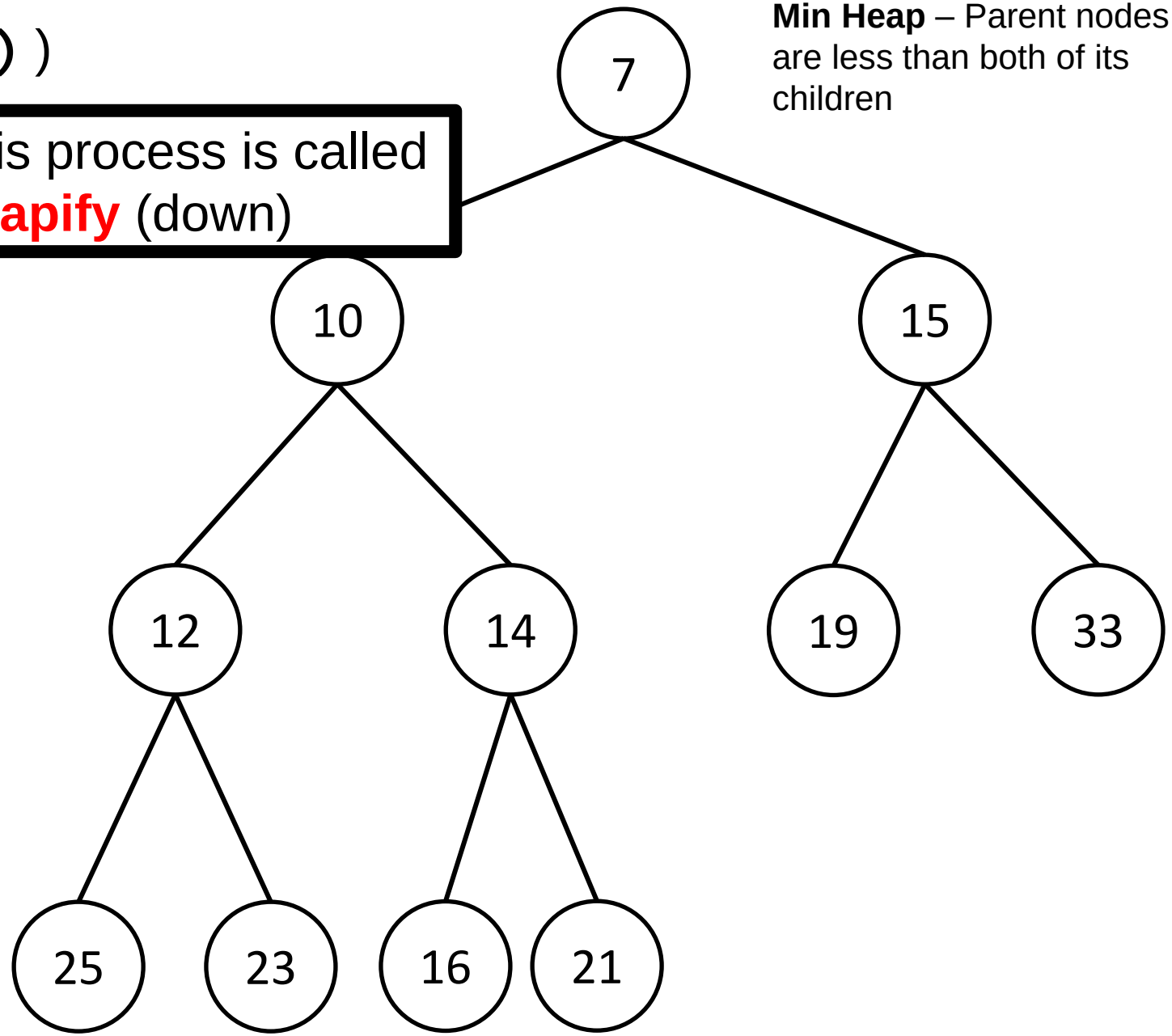
Min Heap – Parent nodes are less than both of its children

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

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**

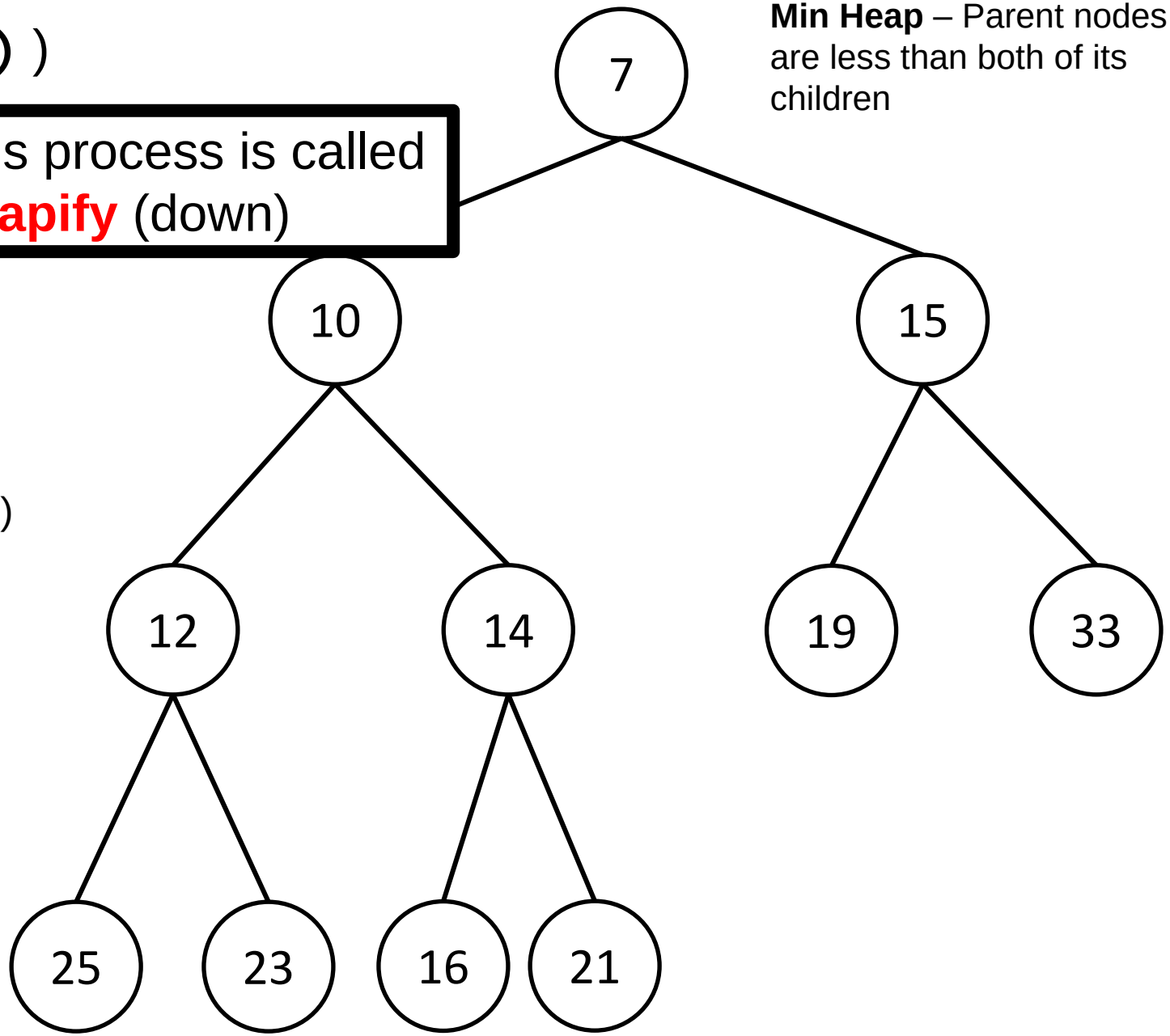
When the root is replaced, it may need to be moved down in the tree



Heap Operations – Removal (`poll()`)

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

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

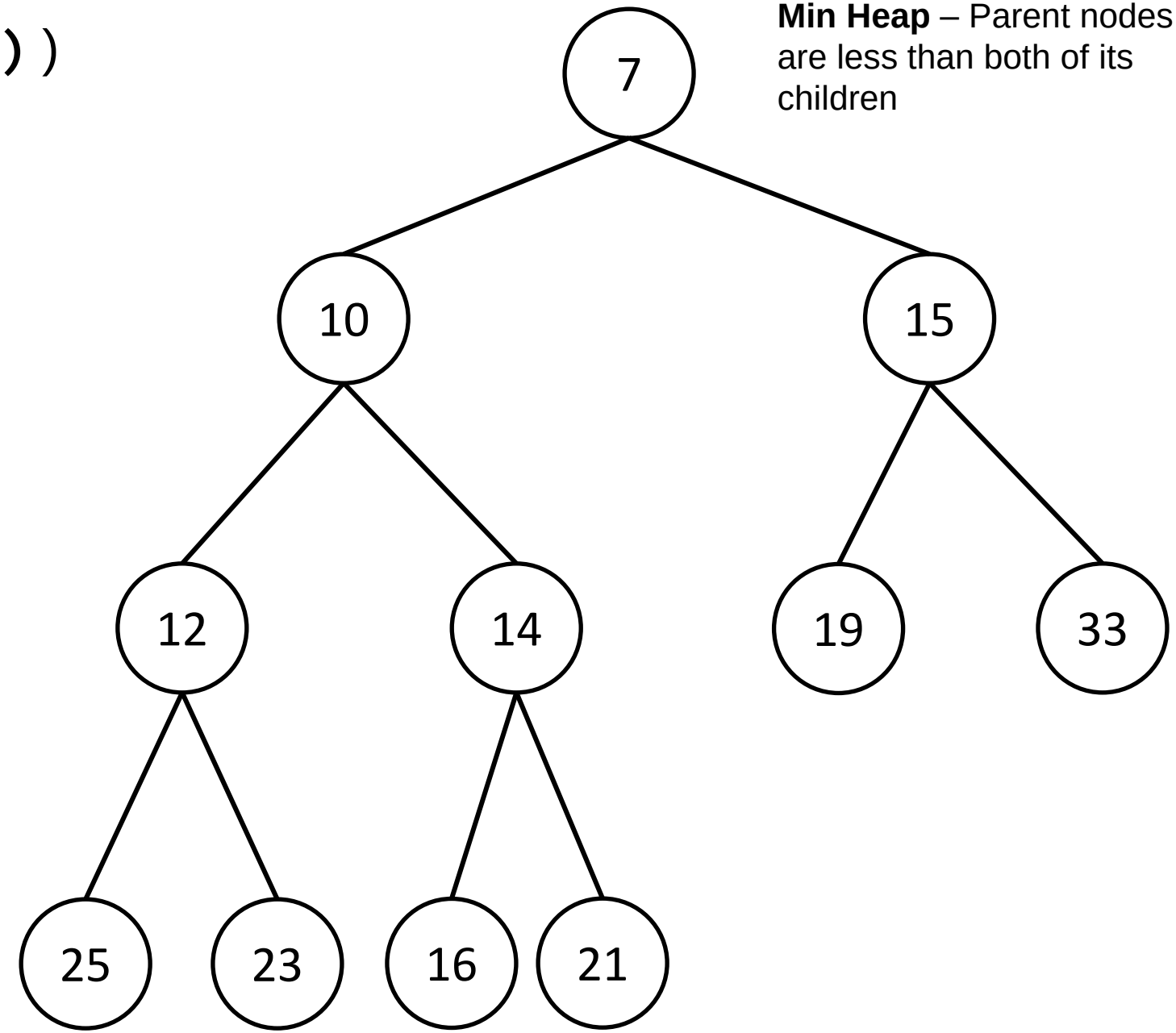


Running time?
- Removing root: **O(1)**
- Replacing root: **O(1)** (this will make sense later)
- Heapify down: **O(logn)**

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

Heap Operations – Removal (`poll()`)

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



Heapify (up)

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

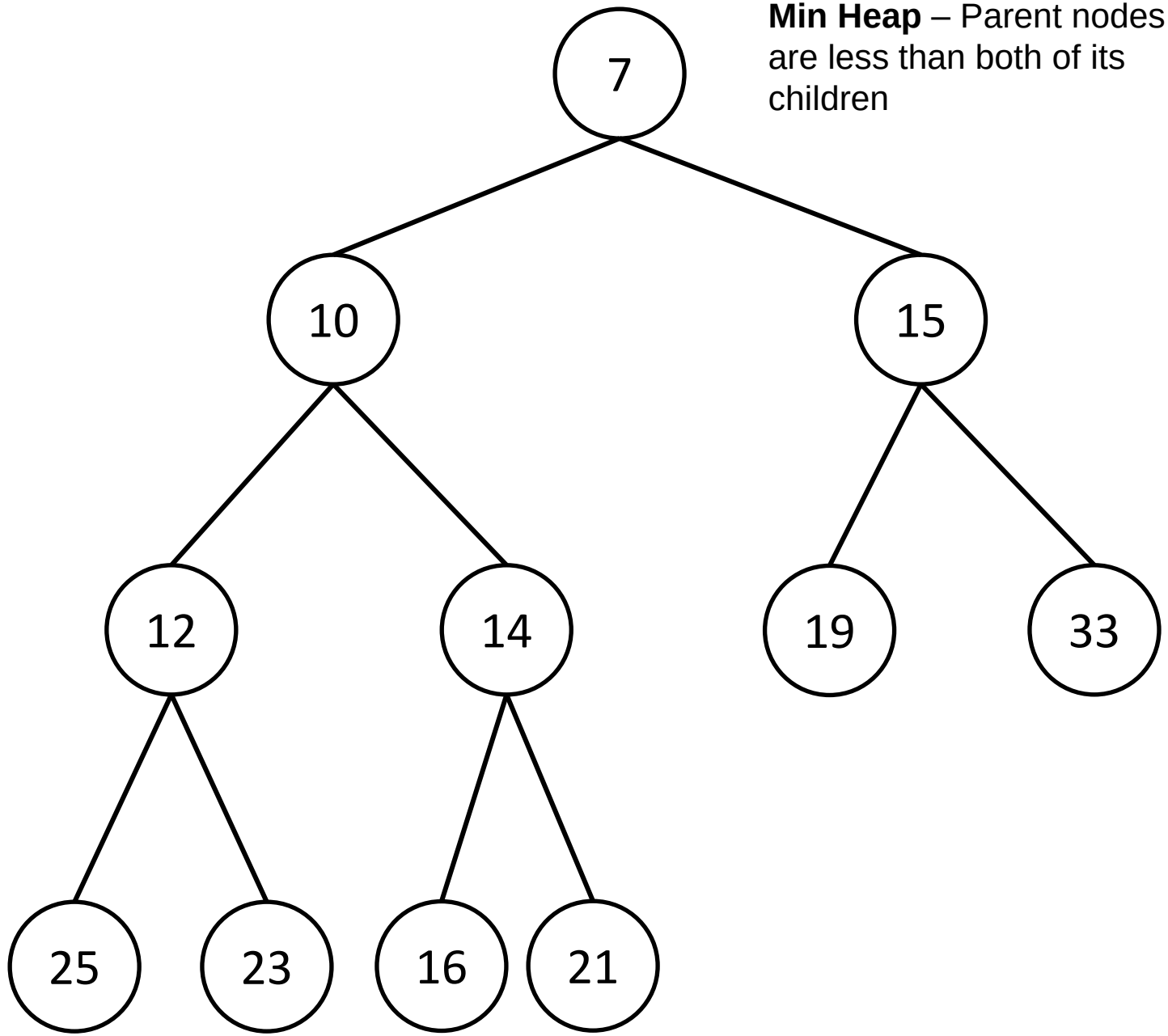
Heapify (down)

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

Heap Representation

How to represent a heap?

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

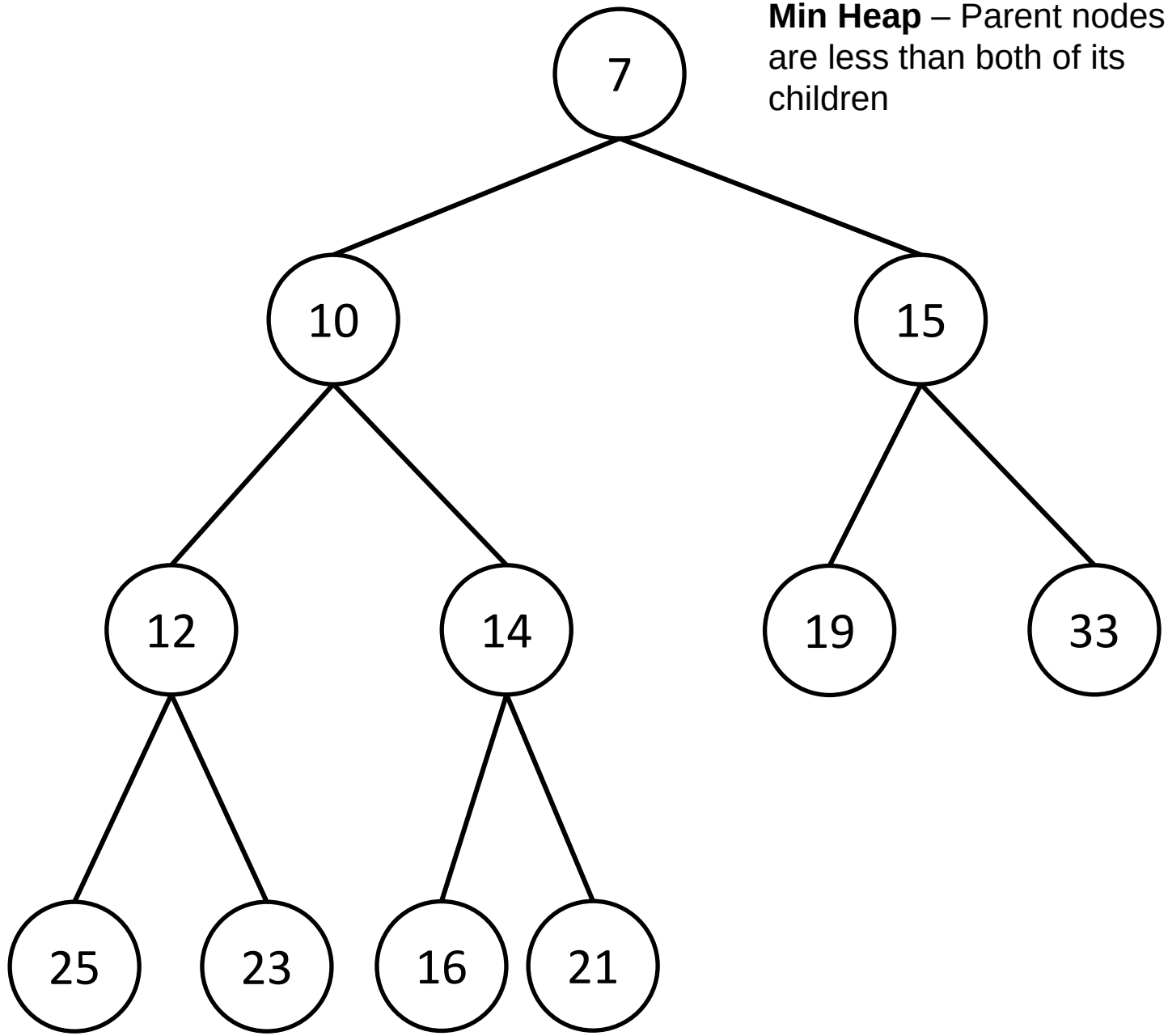


Min Heap – Parent nodes are less than both of its children

Heap Representation

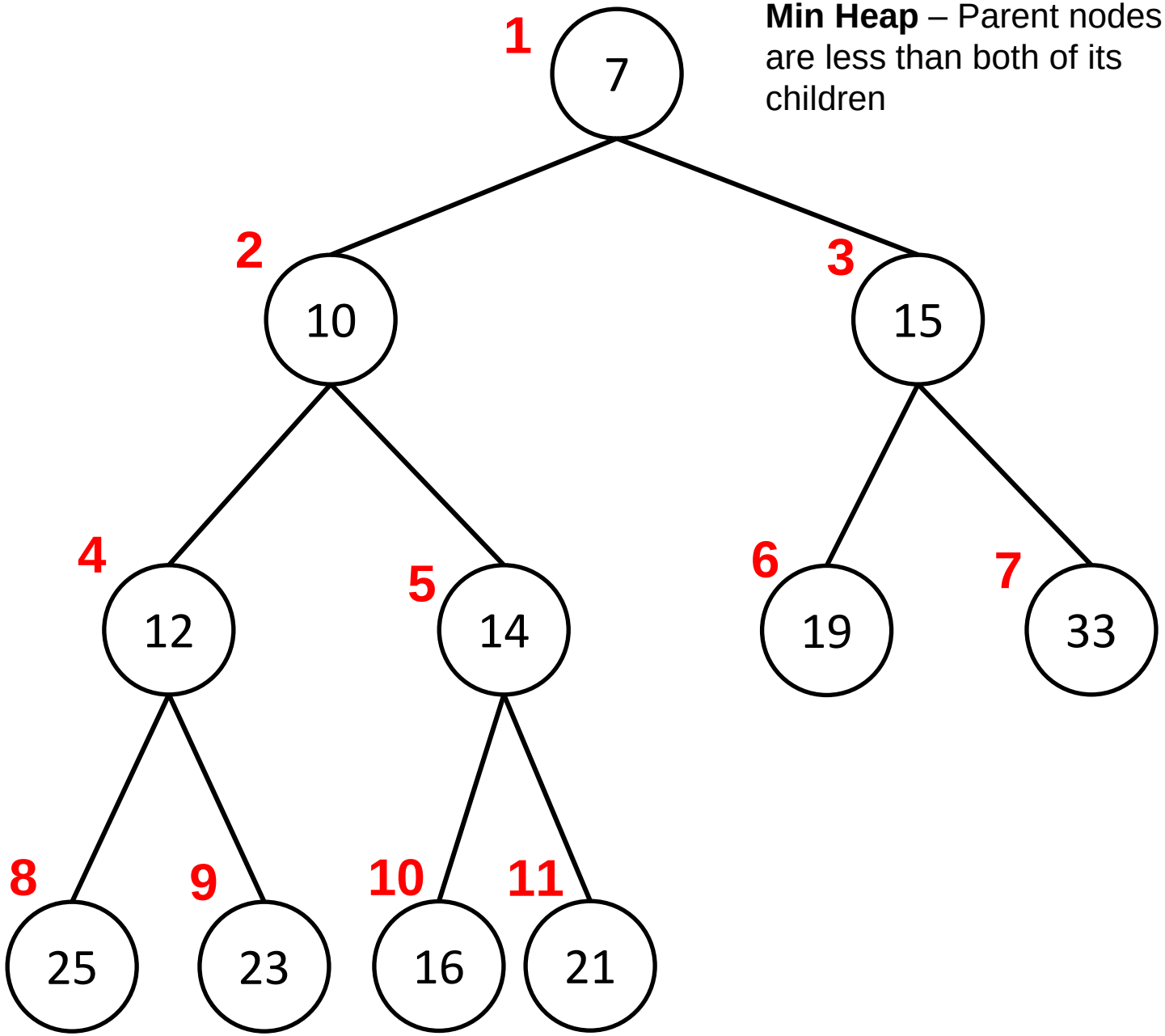
How to represent a heap?

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



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

Heap Representation

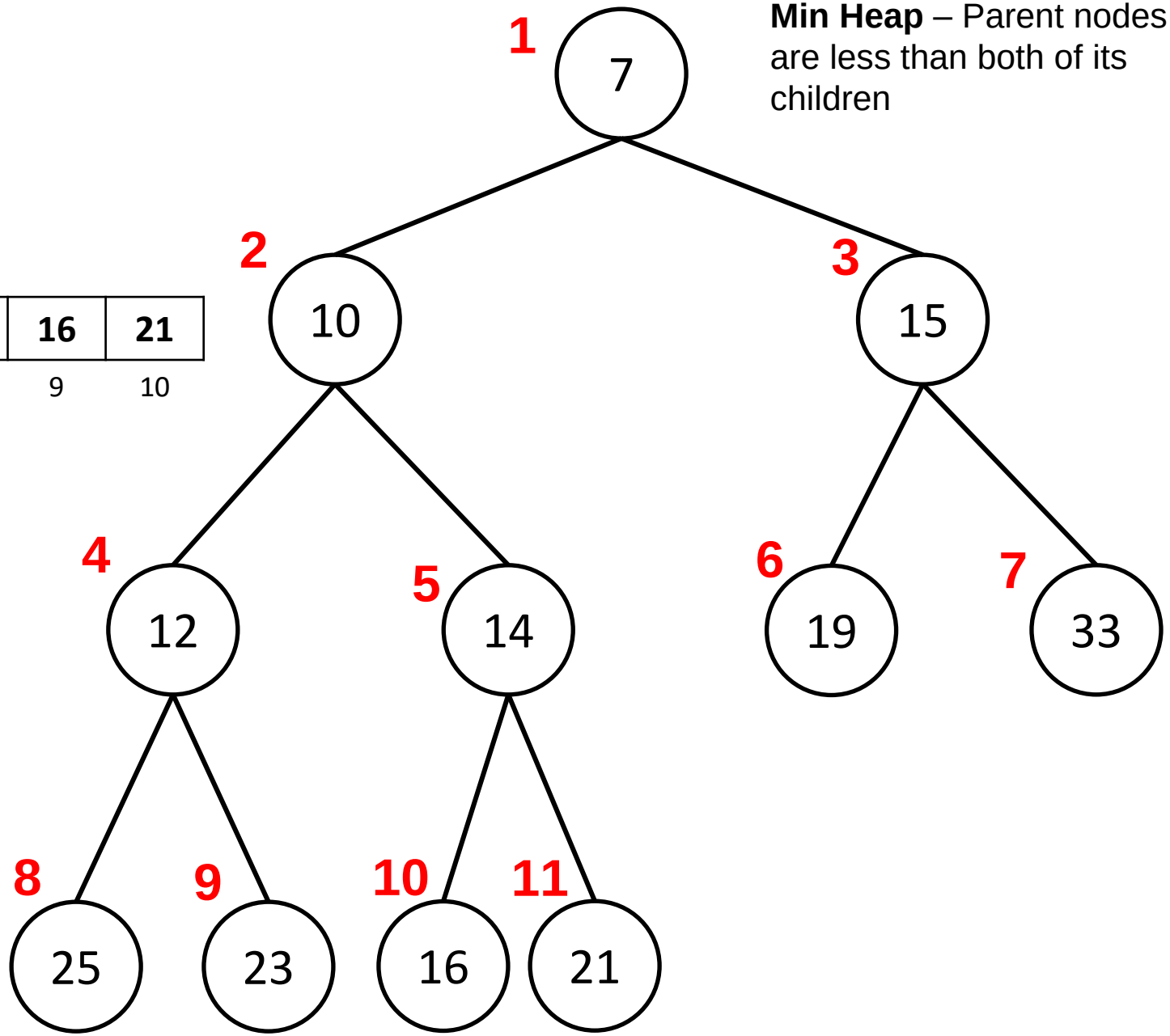


Min Heap – Parent nodes are less than both of its children

Heap Representation

Array

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



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

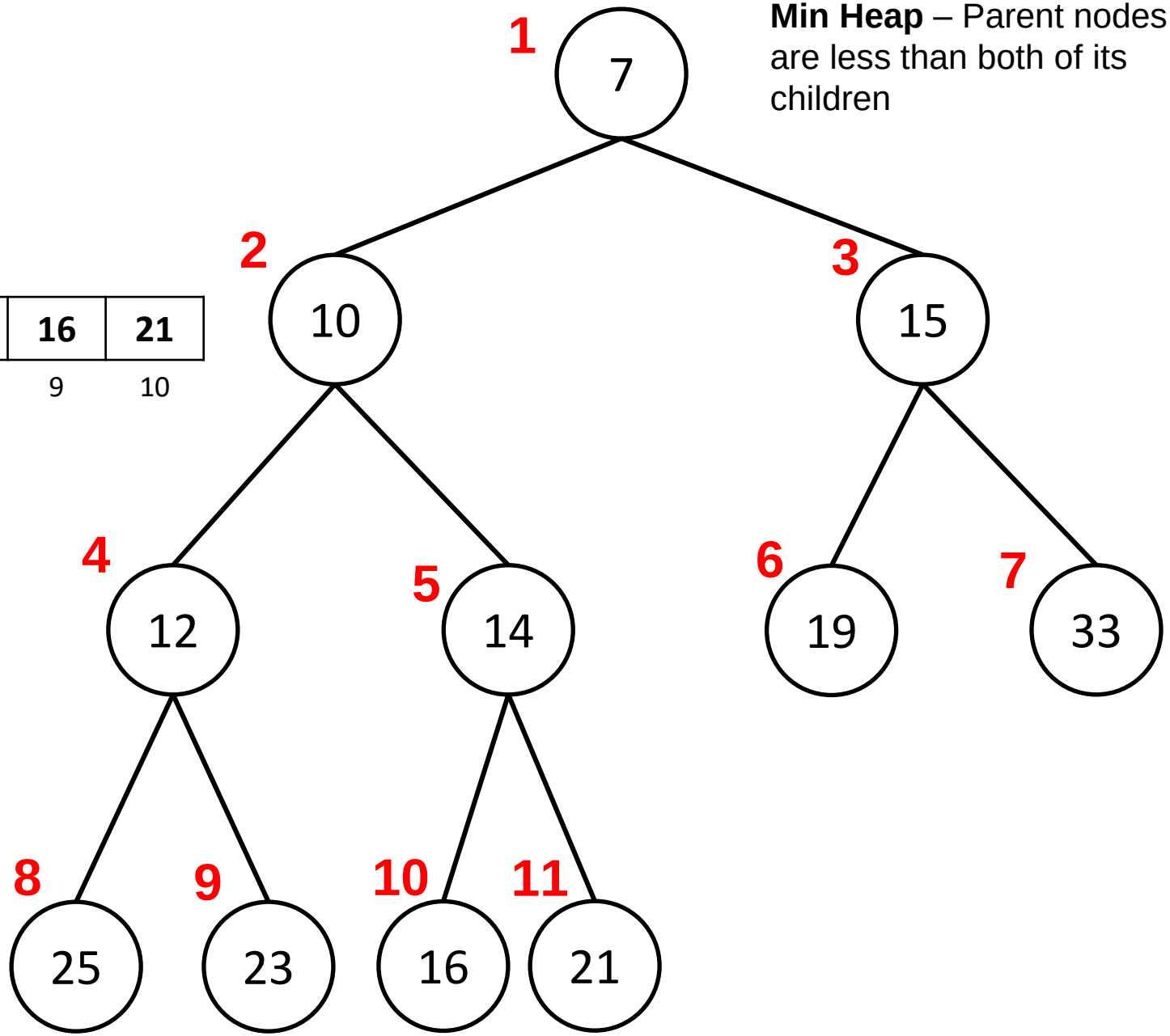
Heap Representation

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

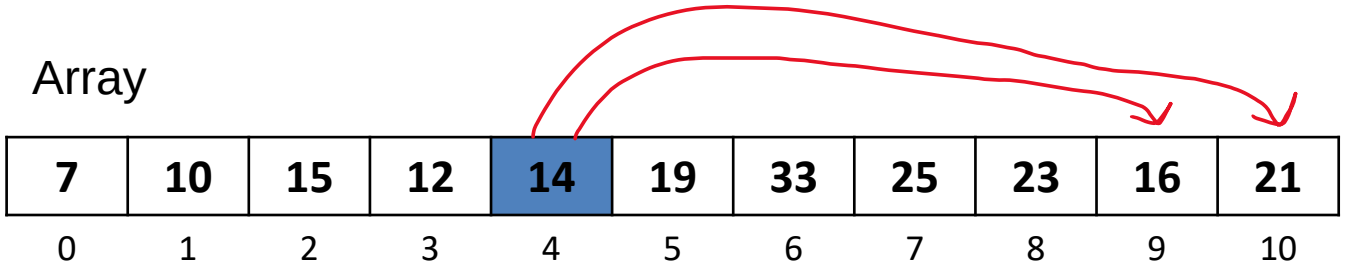
Given a spot in the array, how can we find its children?



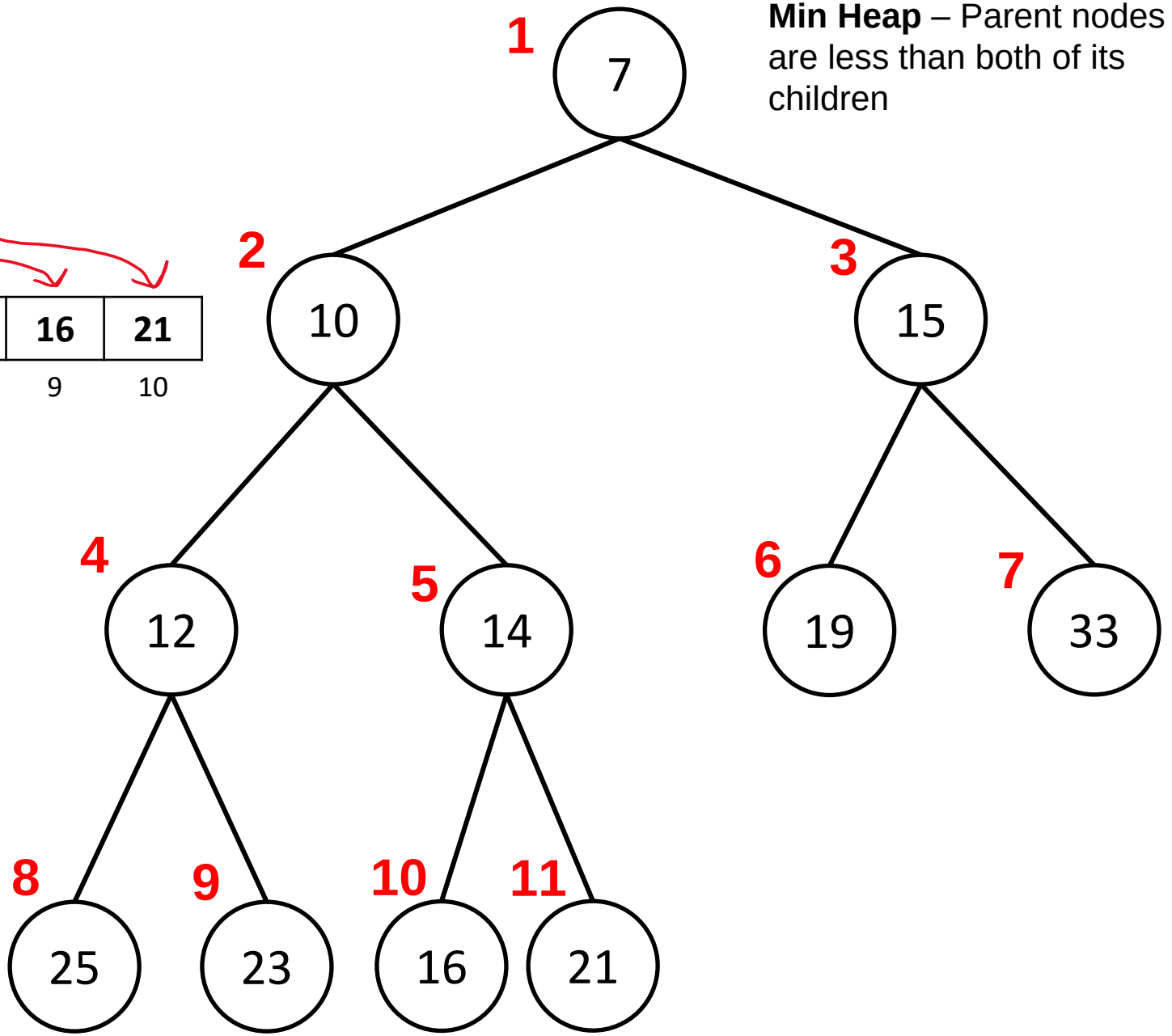
Heap Representation

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

Array



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



Heap Representation

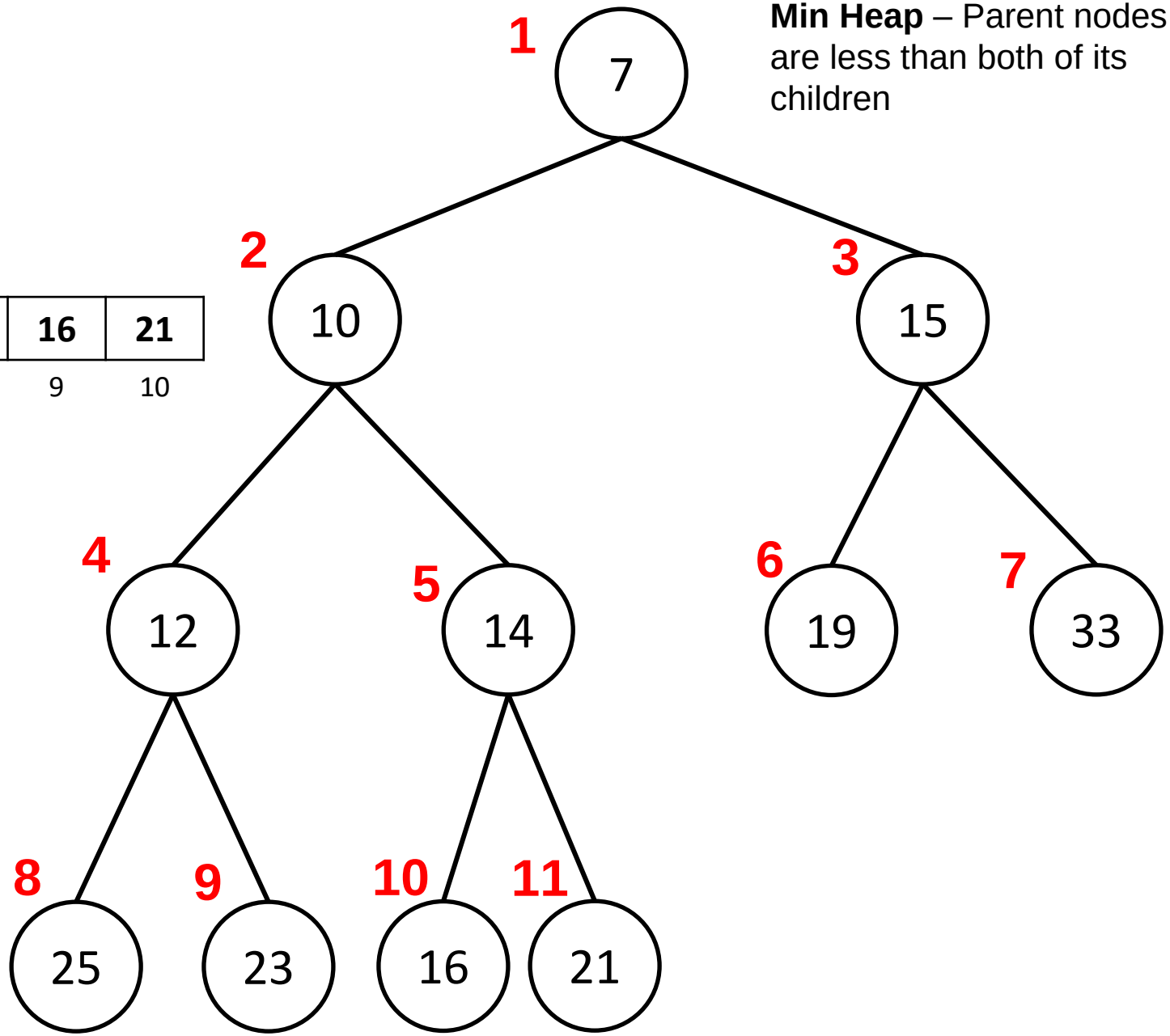
Min Heap – Parent nodes are less than both of its children

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

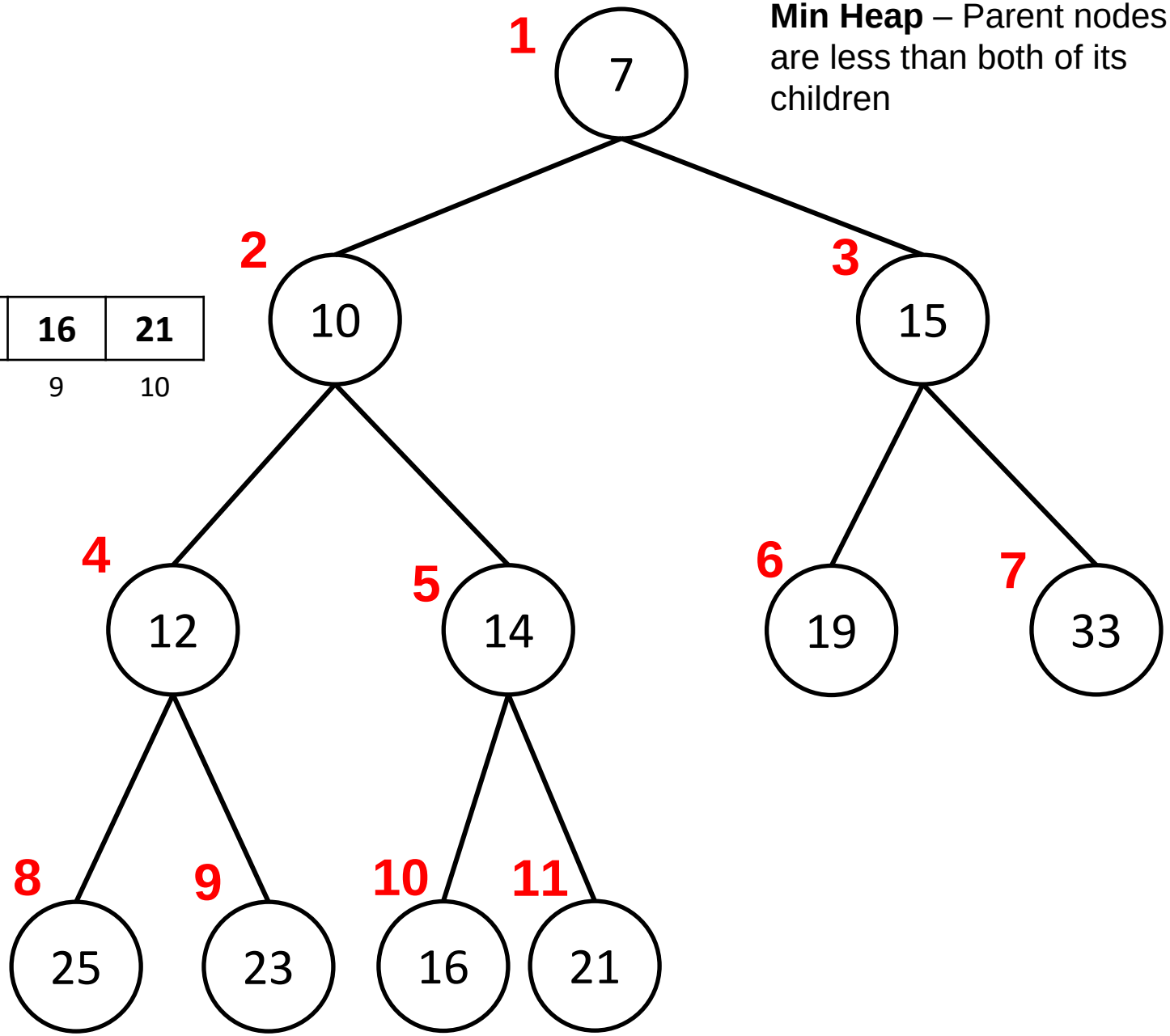


Heap Representation

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
```

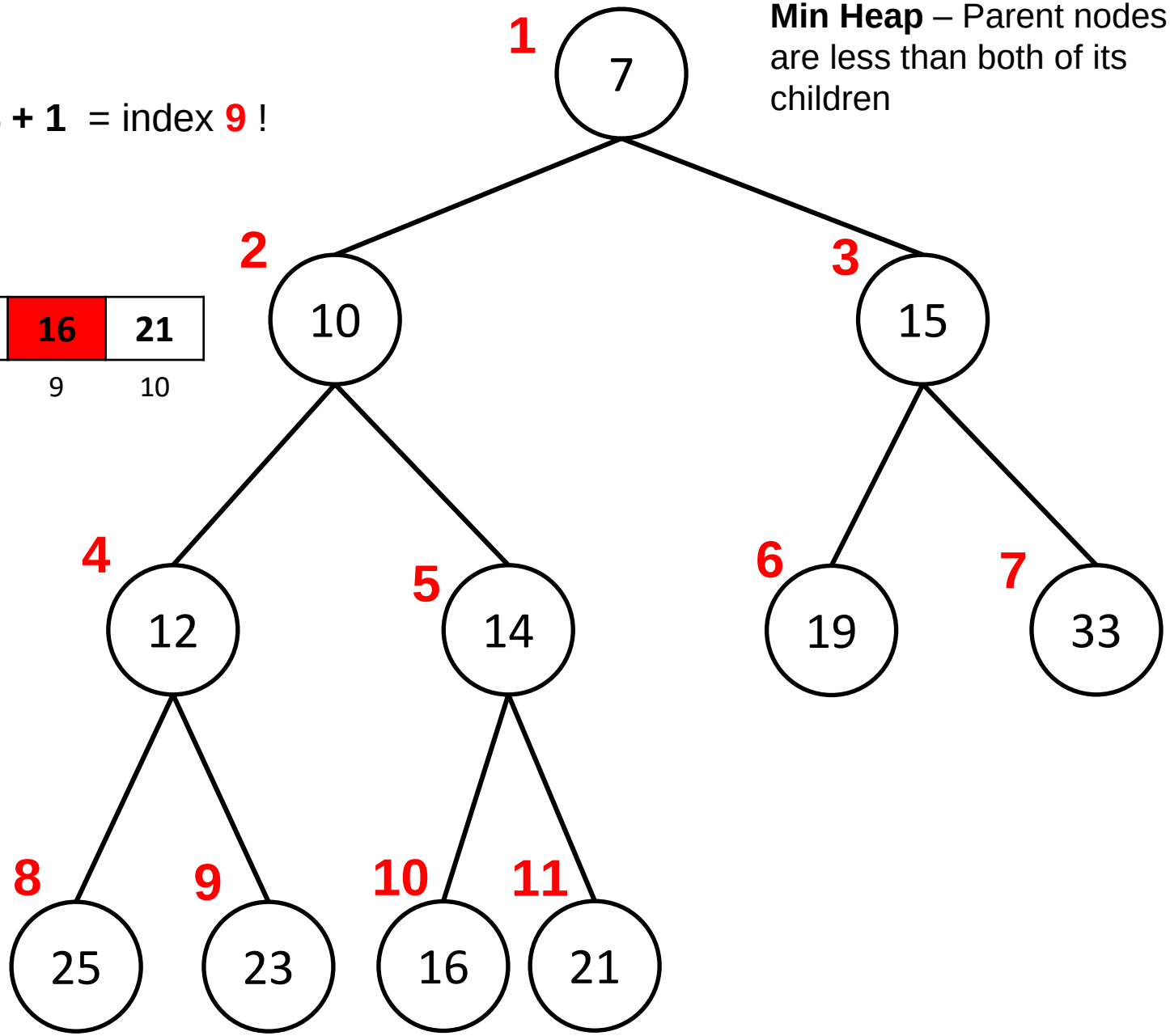
Heap Representation

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

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`

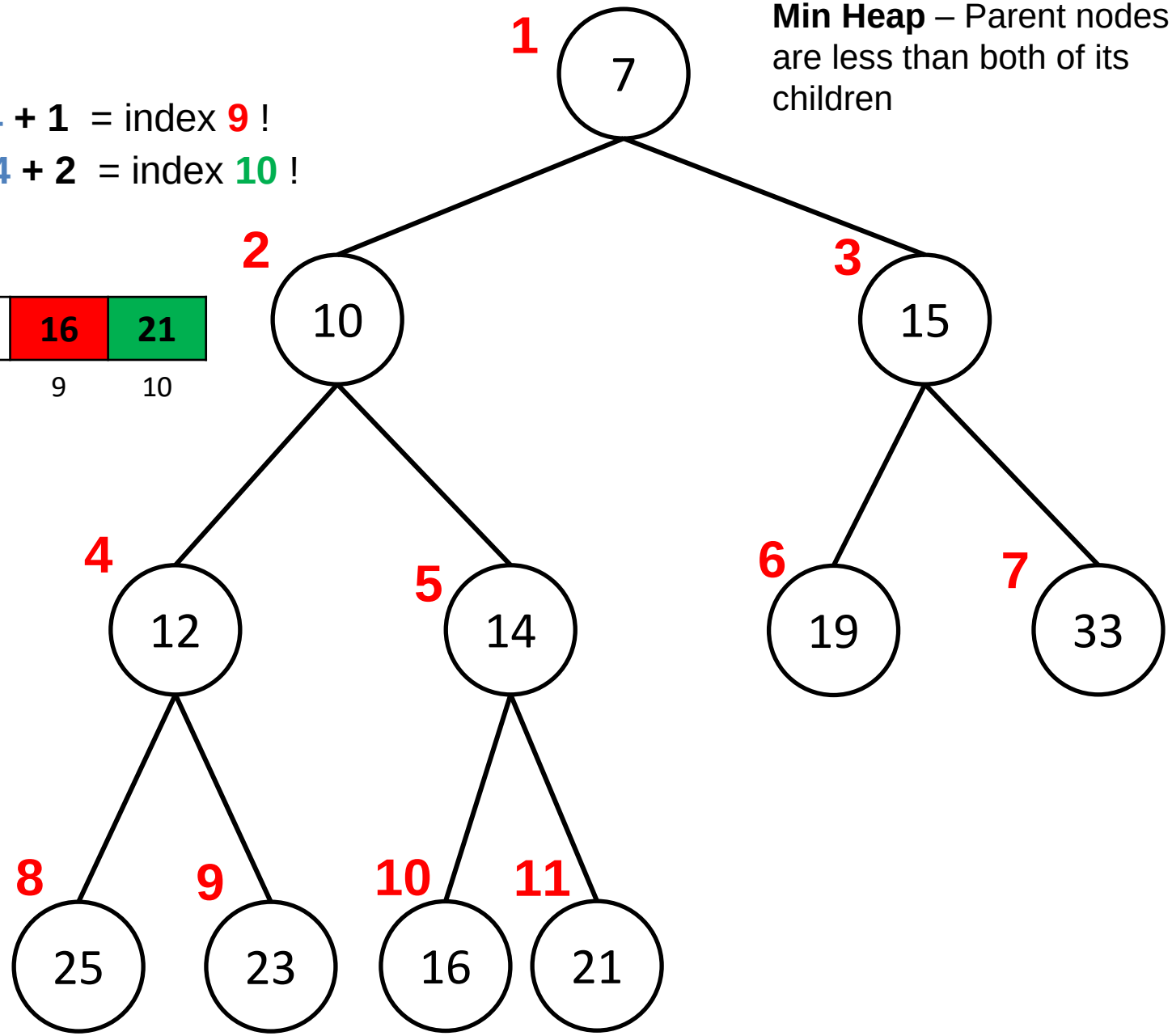
Heap Representation

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

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



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

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`

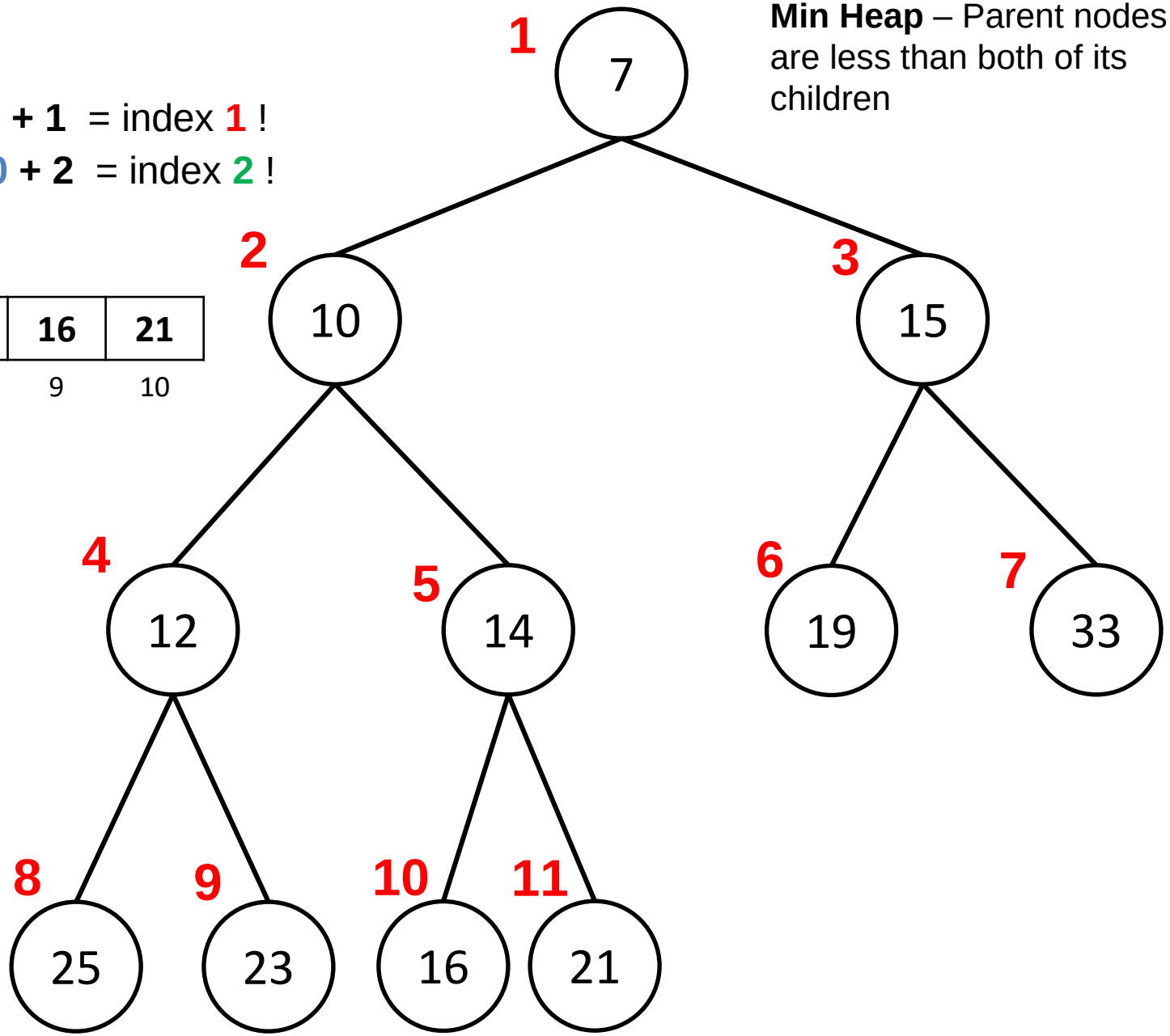
Heap Representation

Left Child = **2 * 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



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

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`

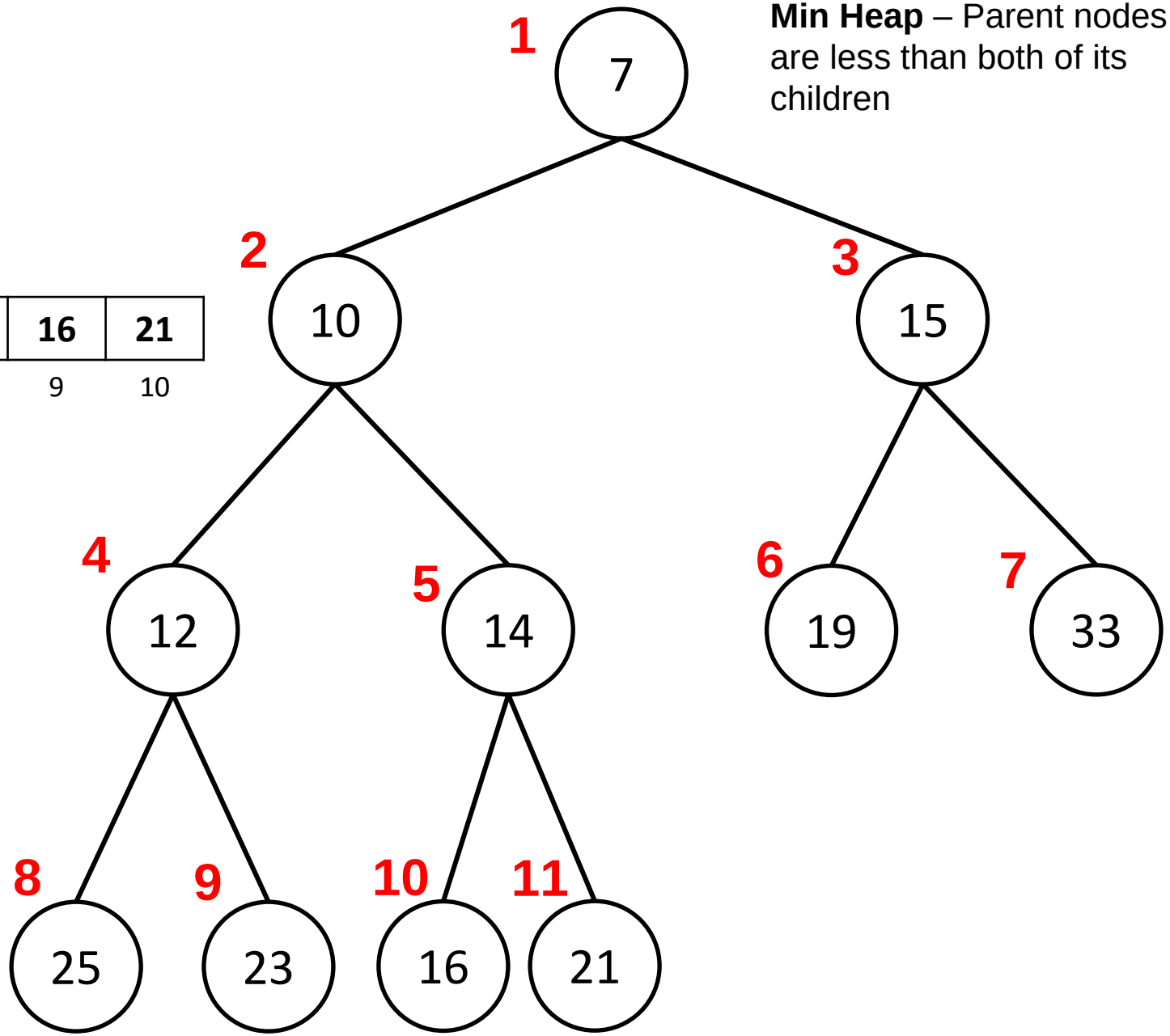
Heap Representation

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 parent?

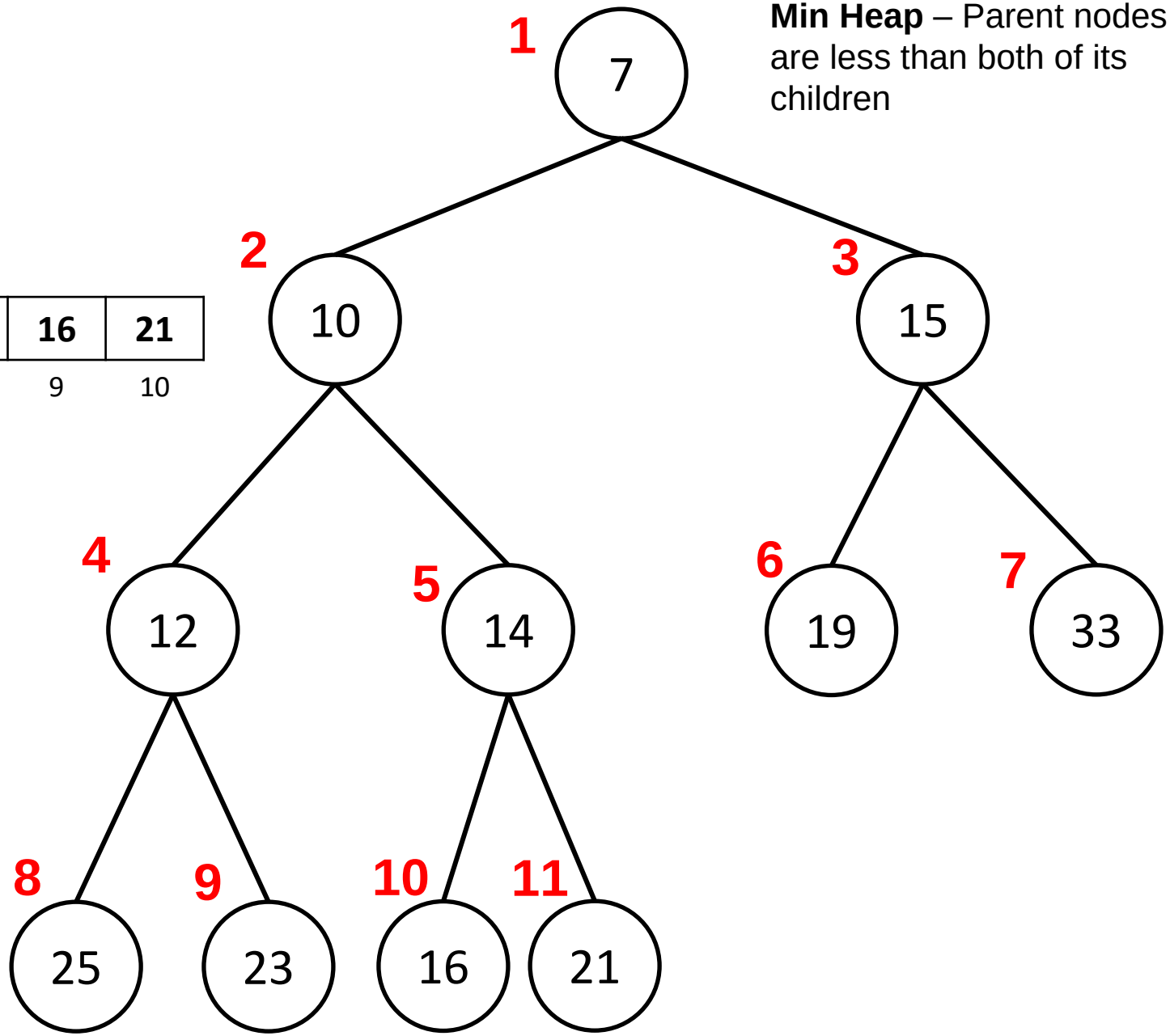


Heap Representation

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 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)

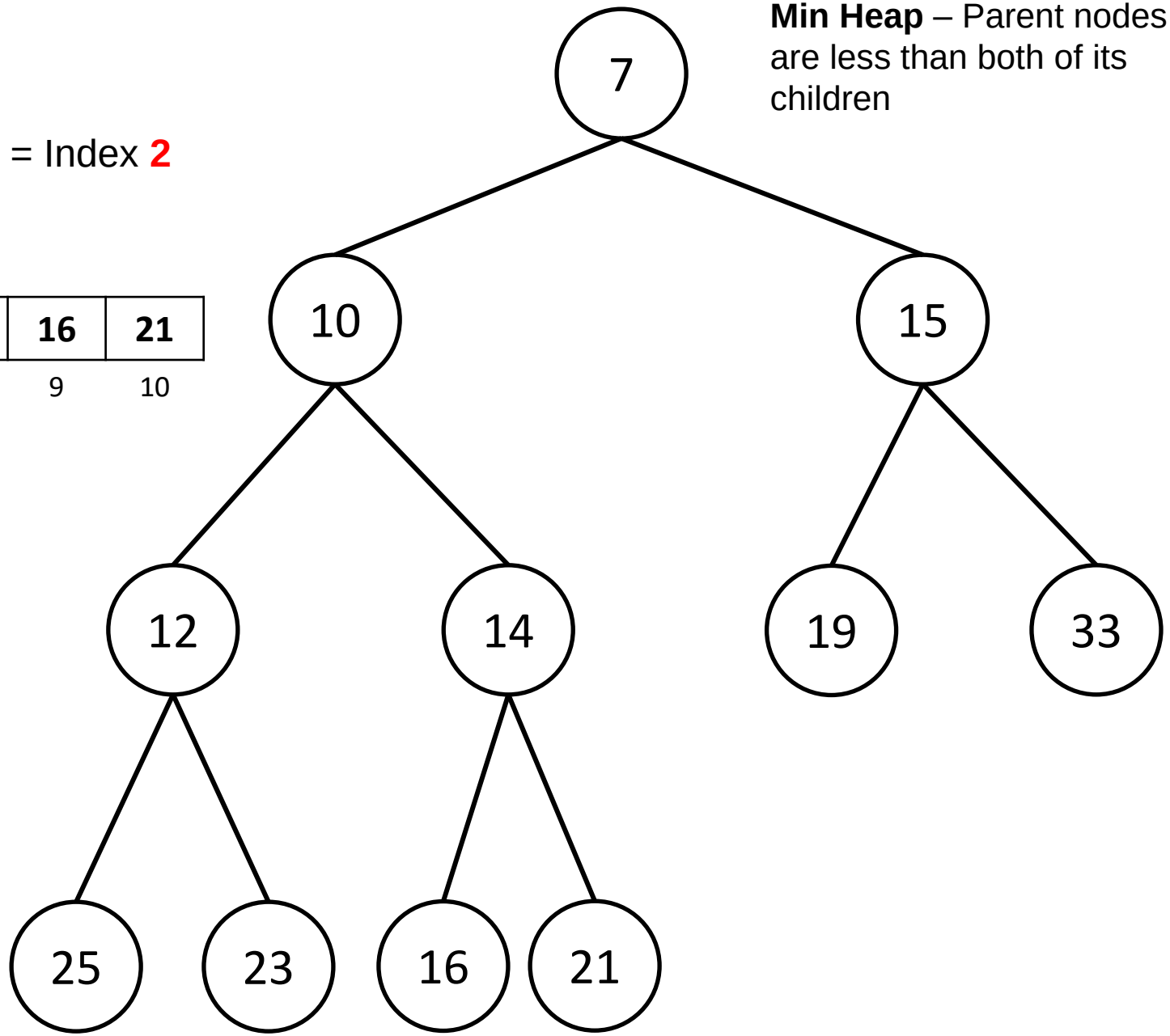
Heap Representation

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

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)

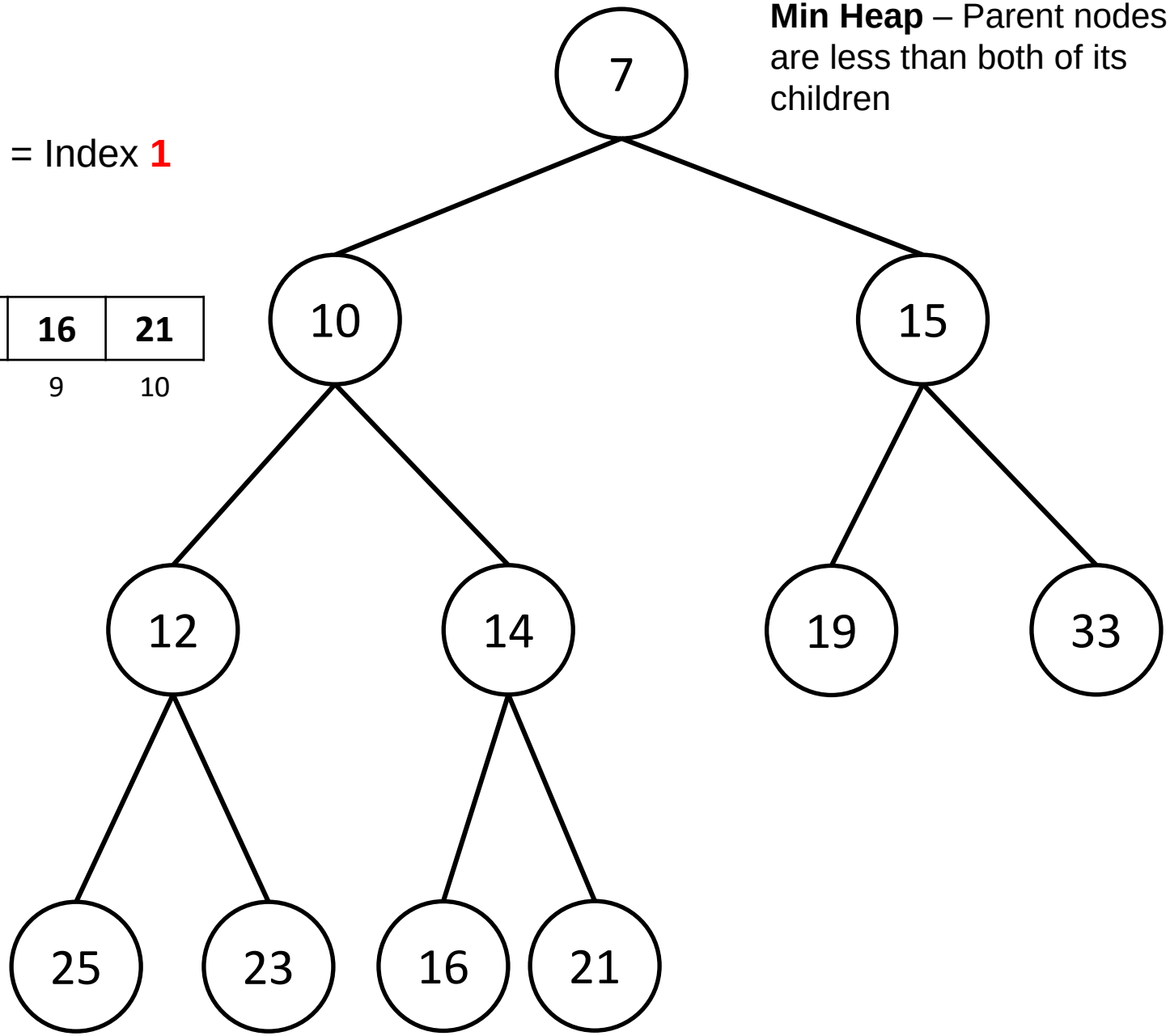
Heap Representation

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

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)

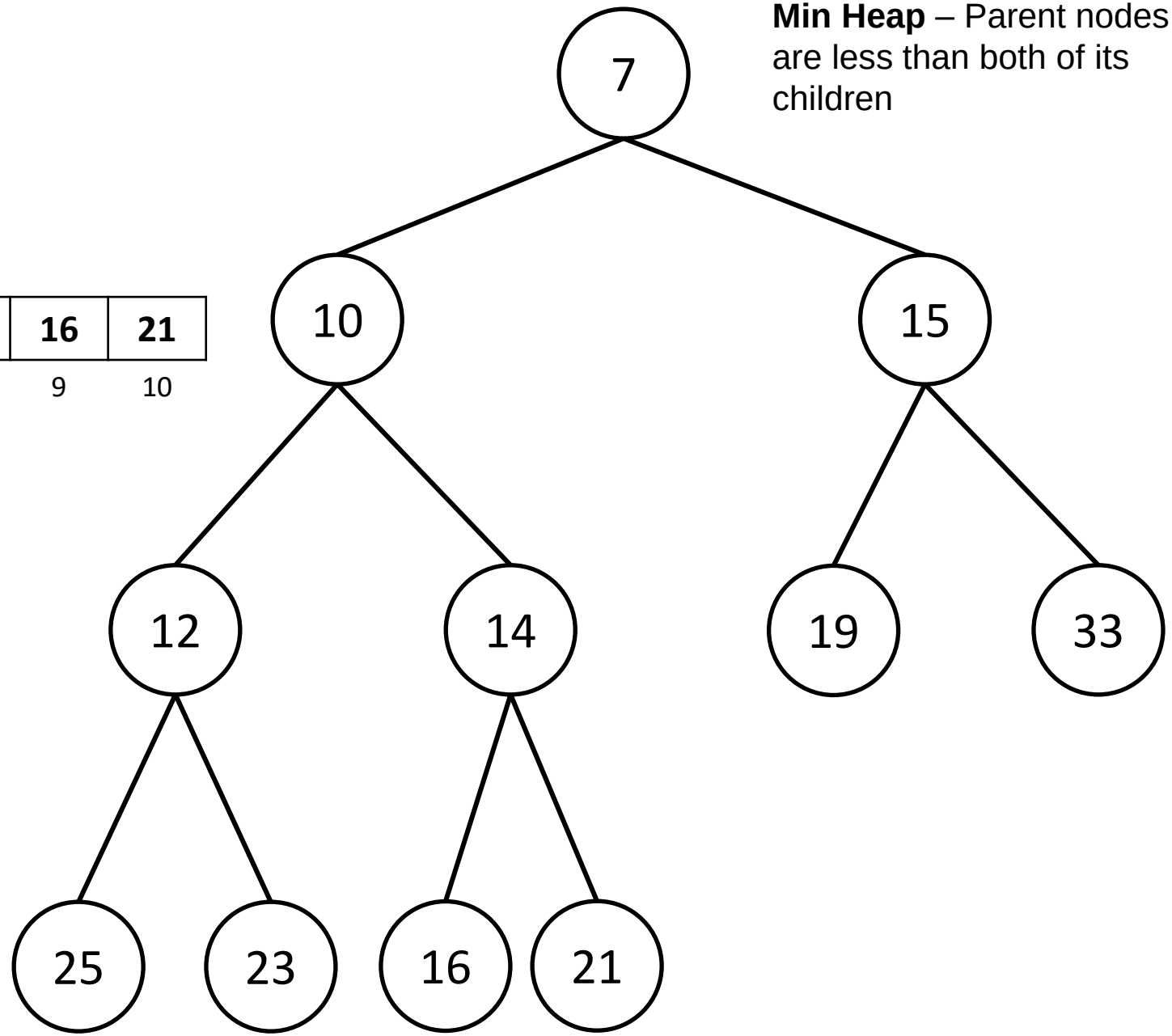
Heap Representation

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

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`

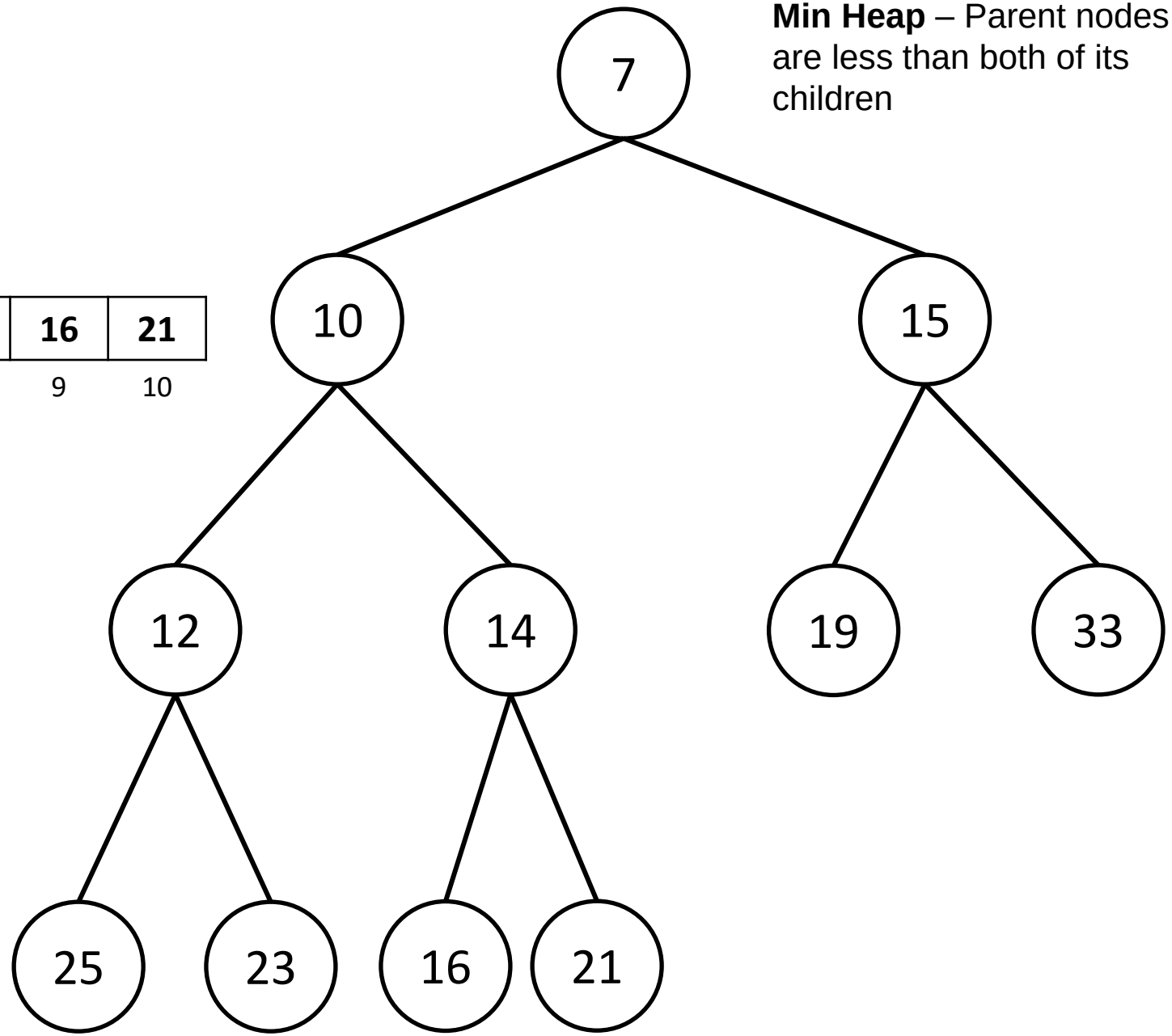
Heap Representation

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

```
insert(11);
```



Heap Representation

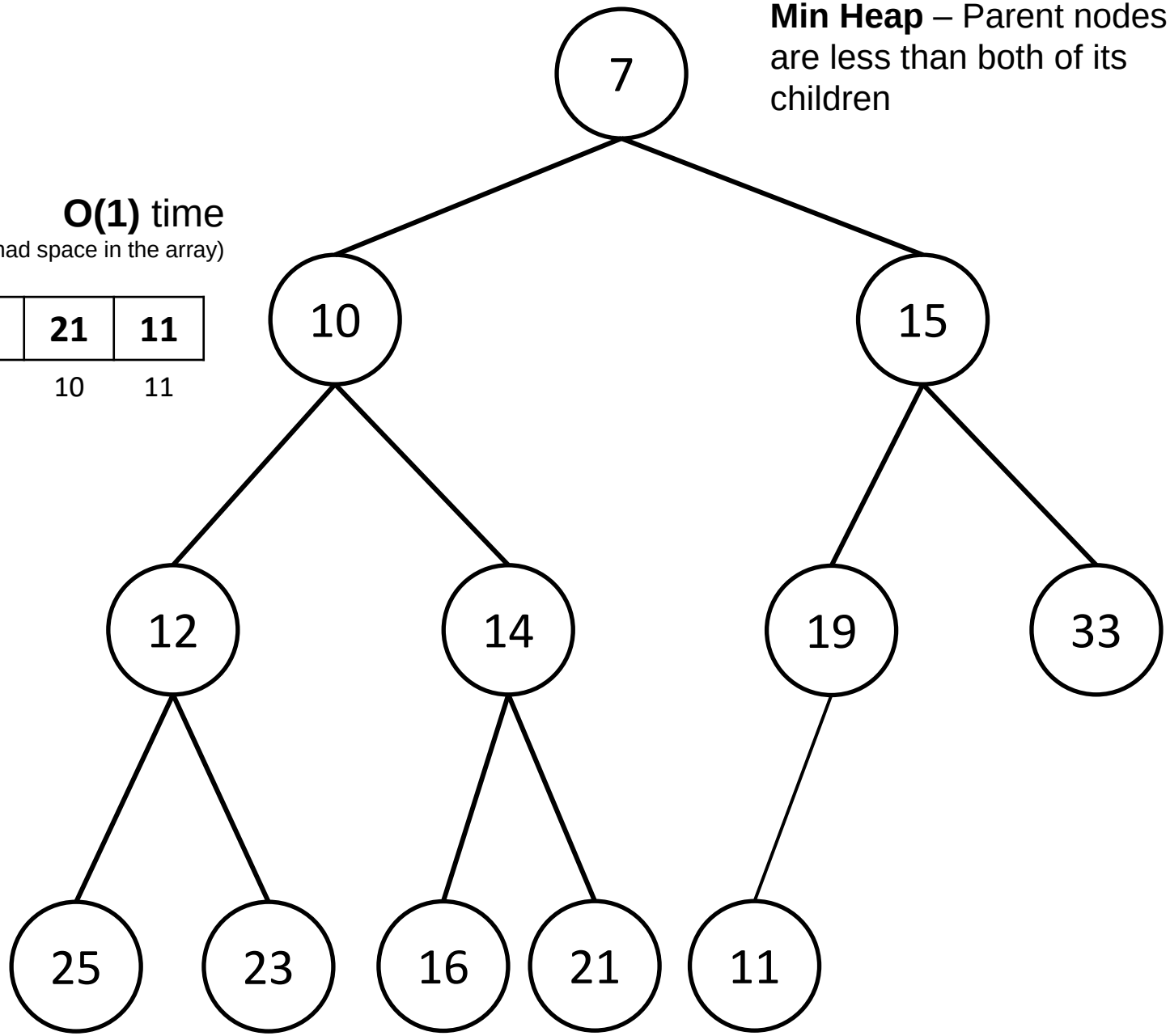
Min Heap – Parent nodes are less than both of its children

Array

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

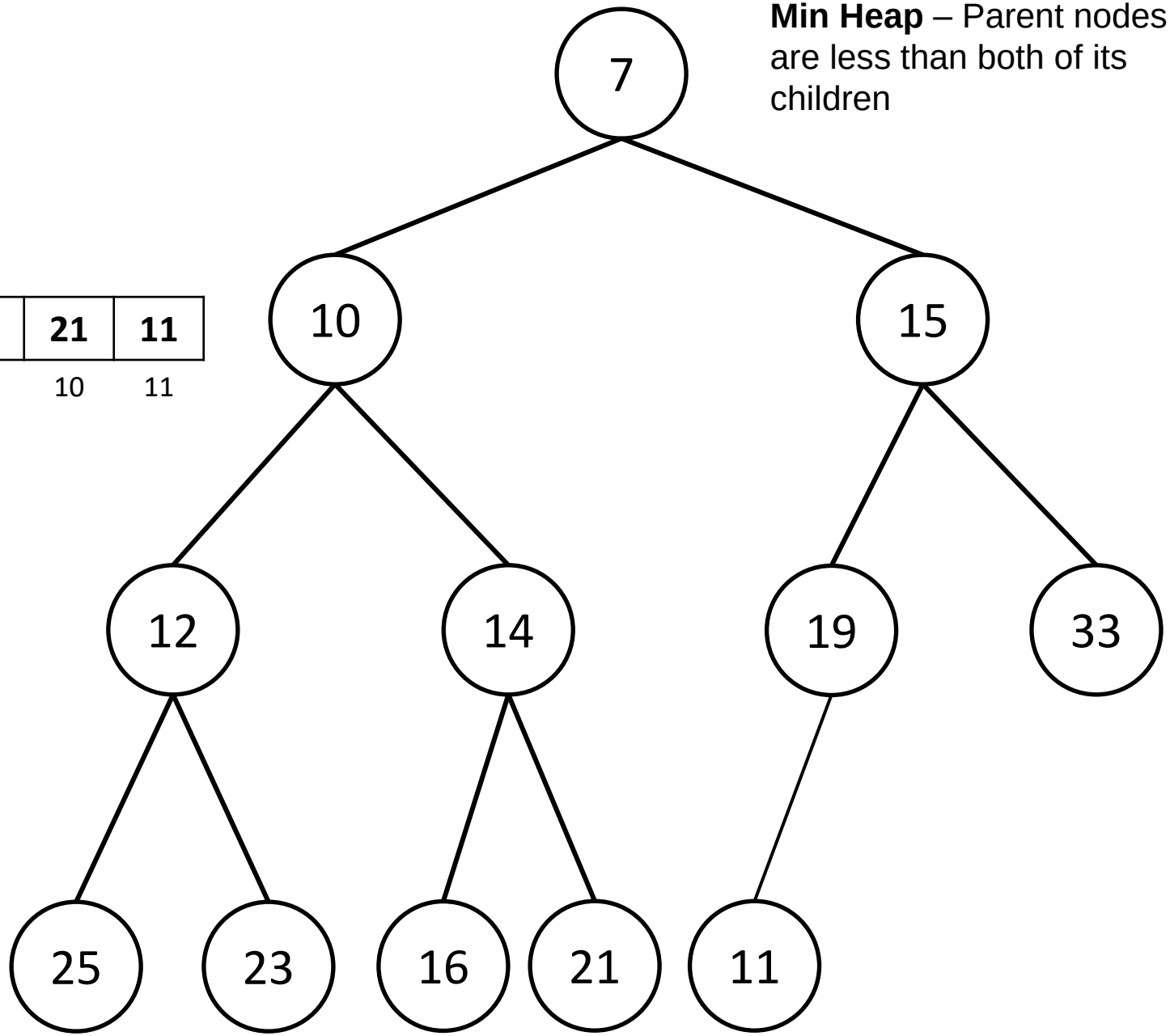
Min Heap – Parent nodes are less than both of its children

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!



Heap Representation

Left Child `2 * i + 1`

Right Child `2 * i + 2`

Parent `(i - 1) / 2`

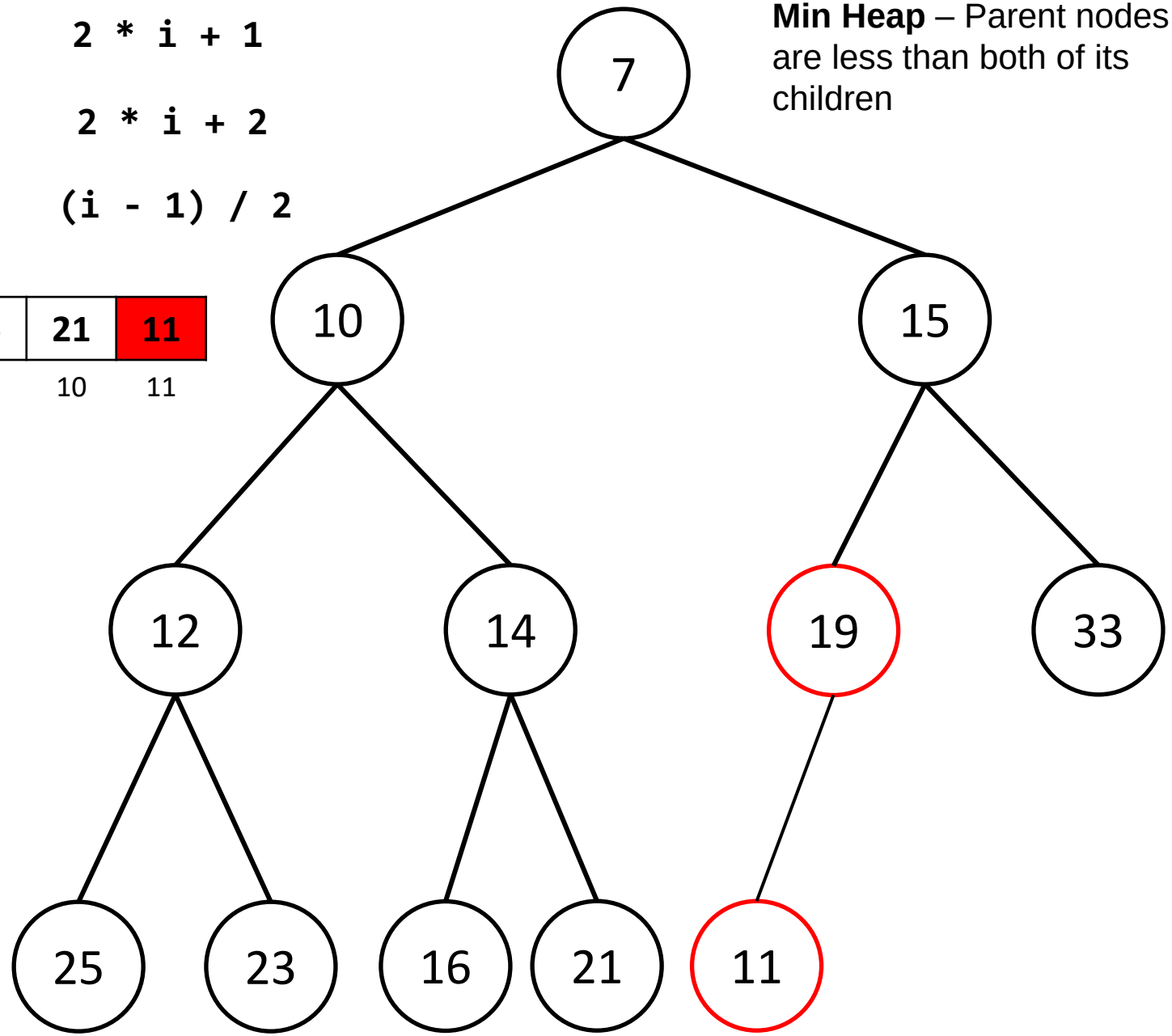
Array

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`

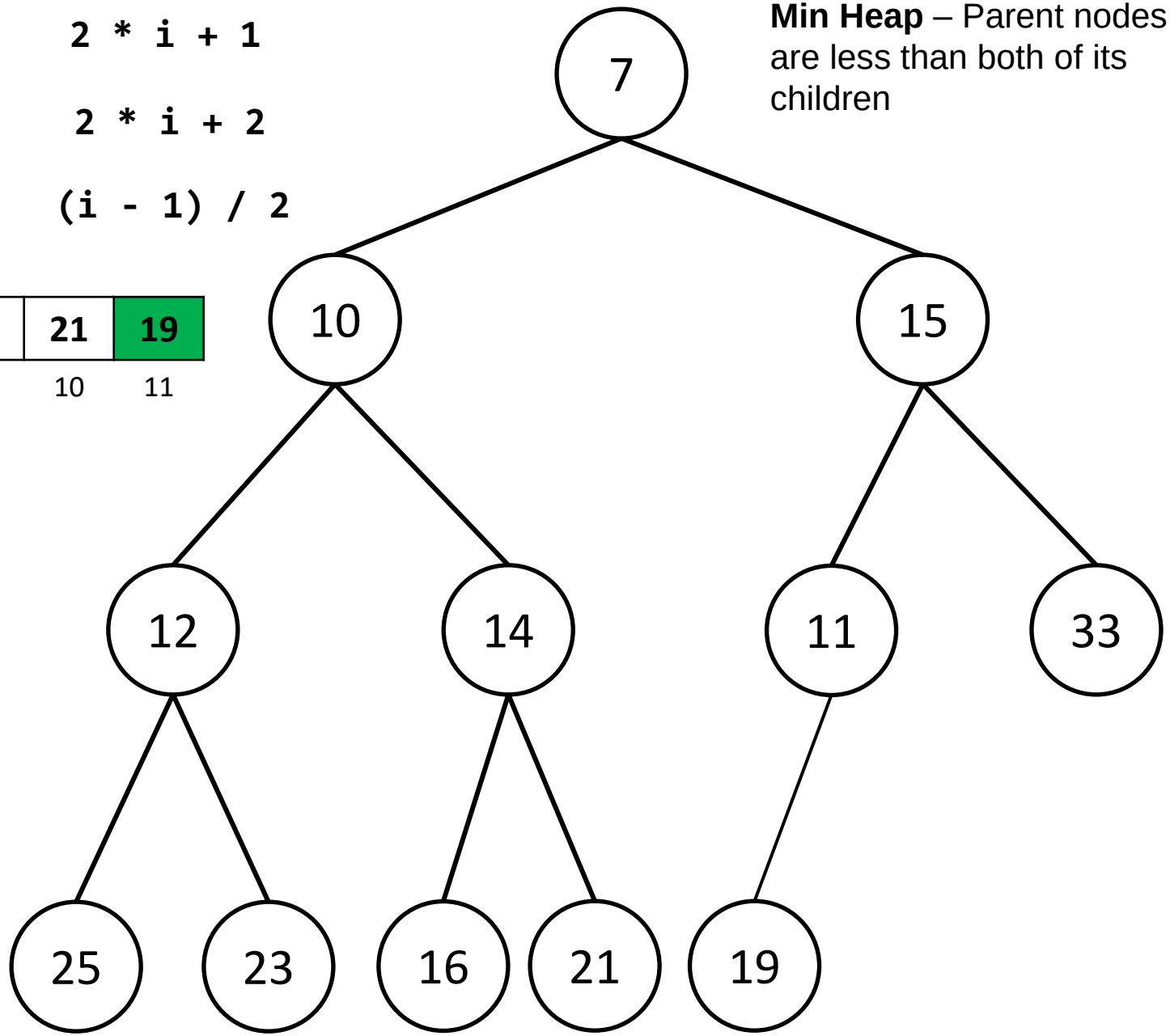
Parent `(i - 1) / 2`

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`

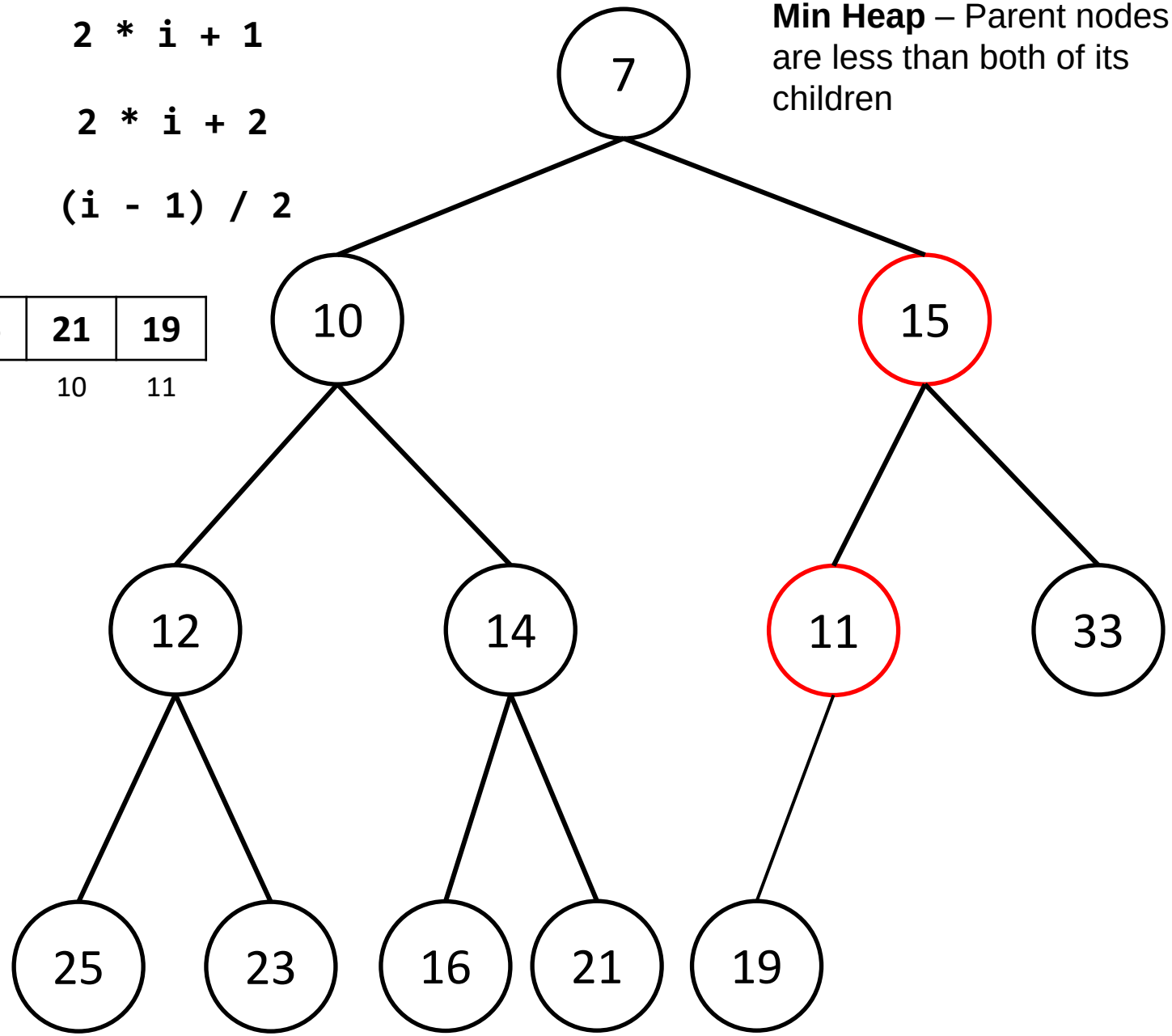
Array

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**



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`

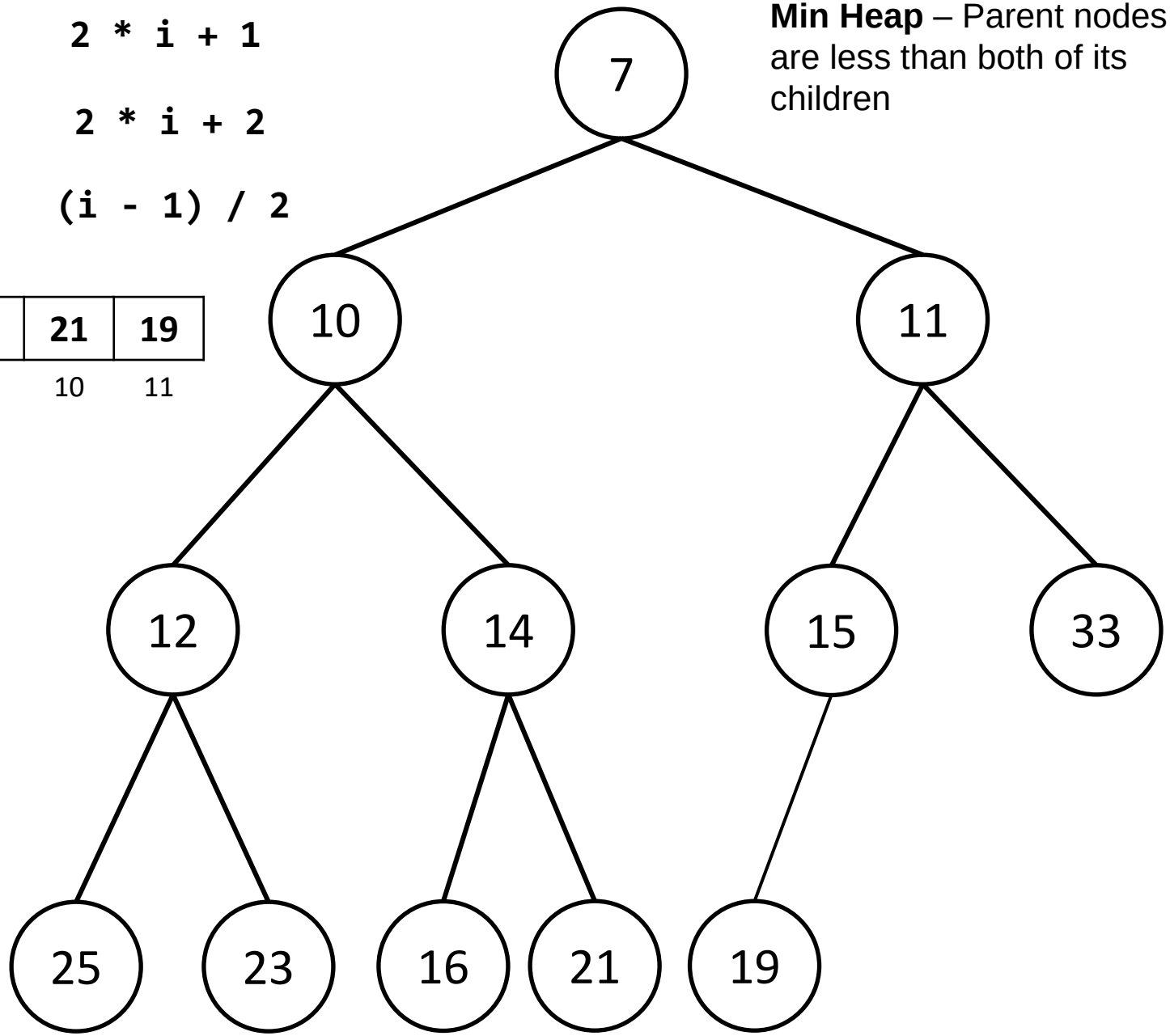
Array

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!

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



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

Heap Representation

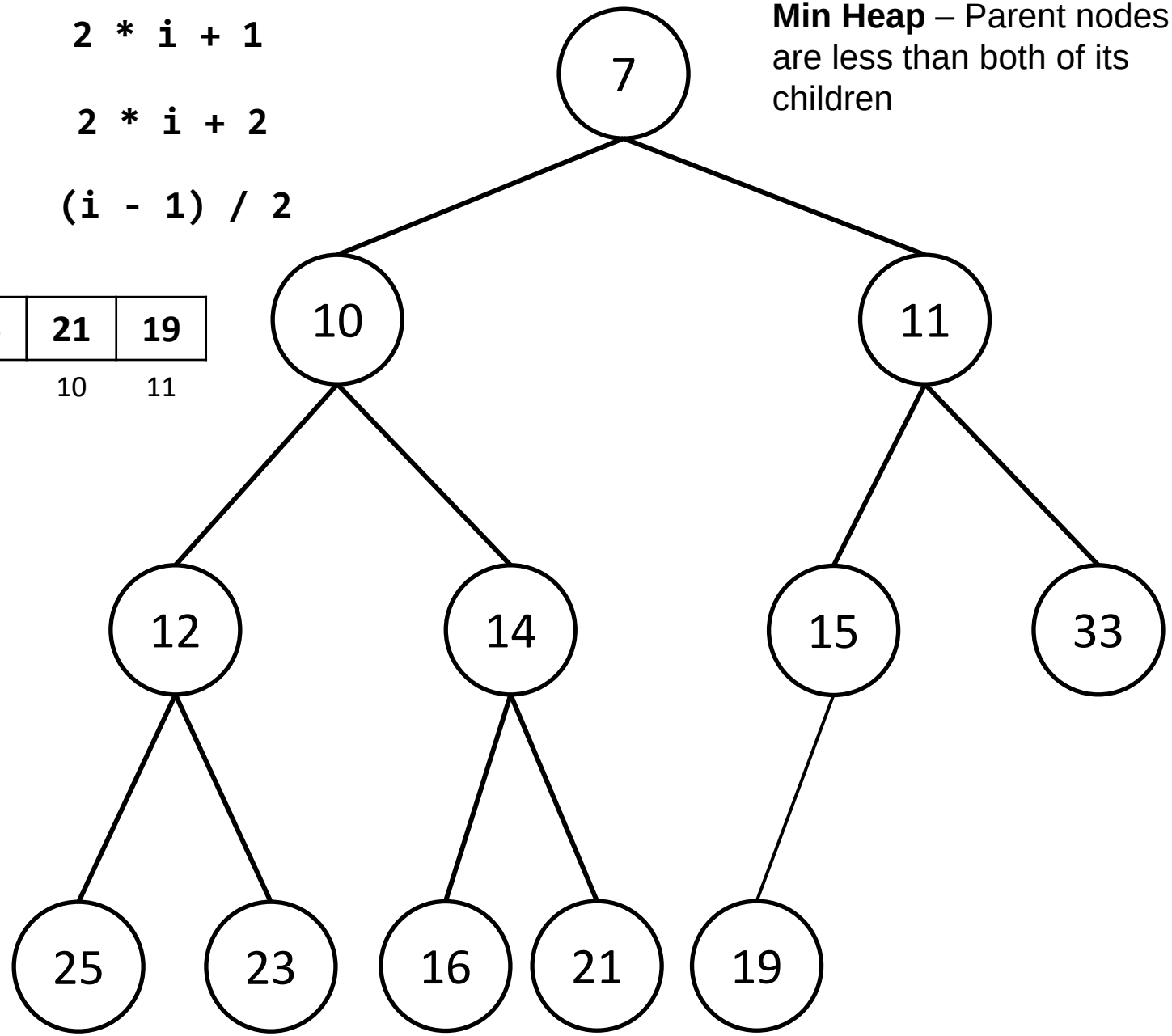
Left Child	`2 * i + 1`
Right Child	`2 * i + 2`
Parent	`(i - 1) / 2`

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`

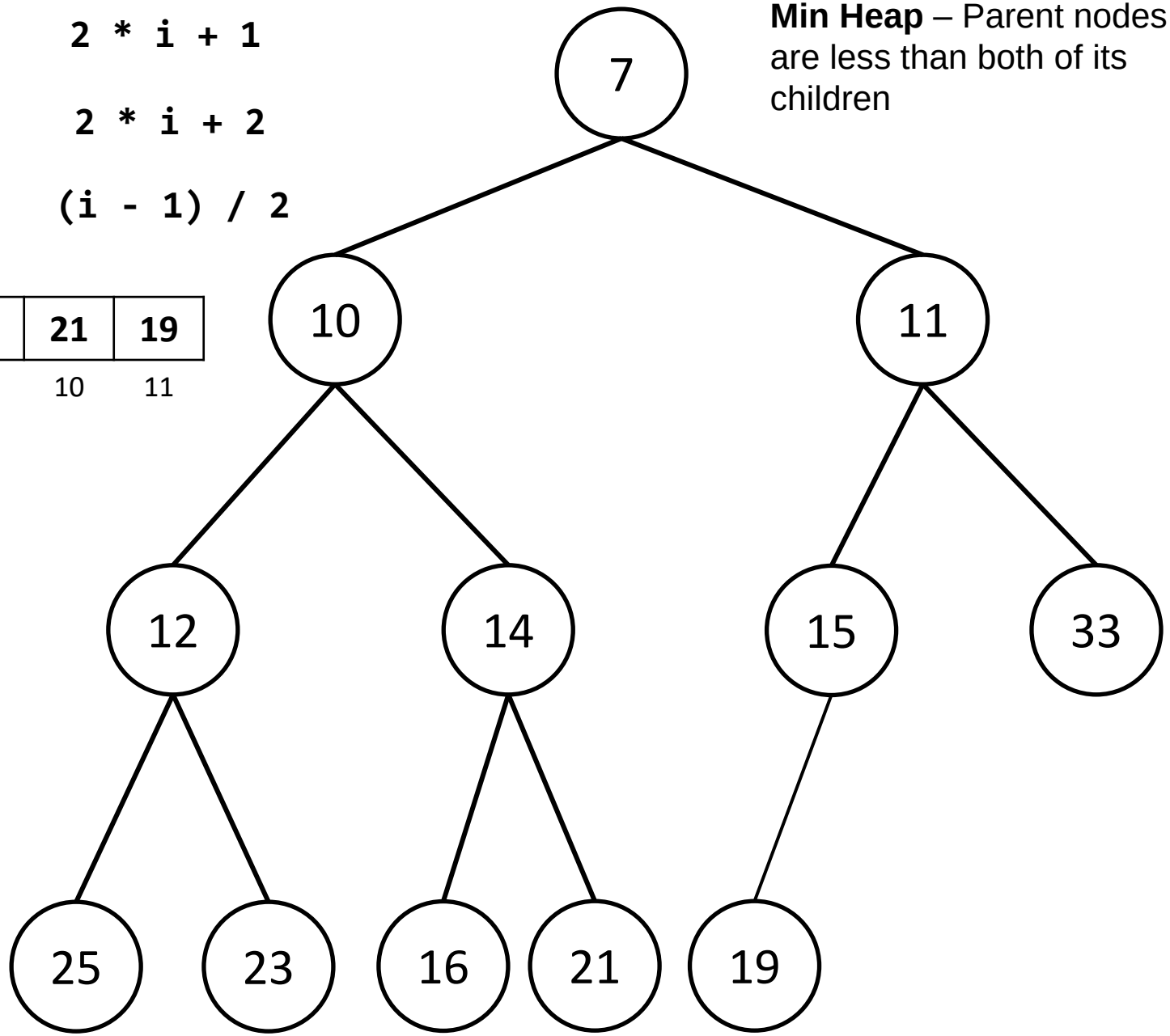
Right Child `2 * i + 2`

Parent `(i - 1) / 2`

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();
```



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

Heap Representation

Left Child `2 * i + 1`

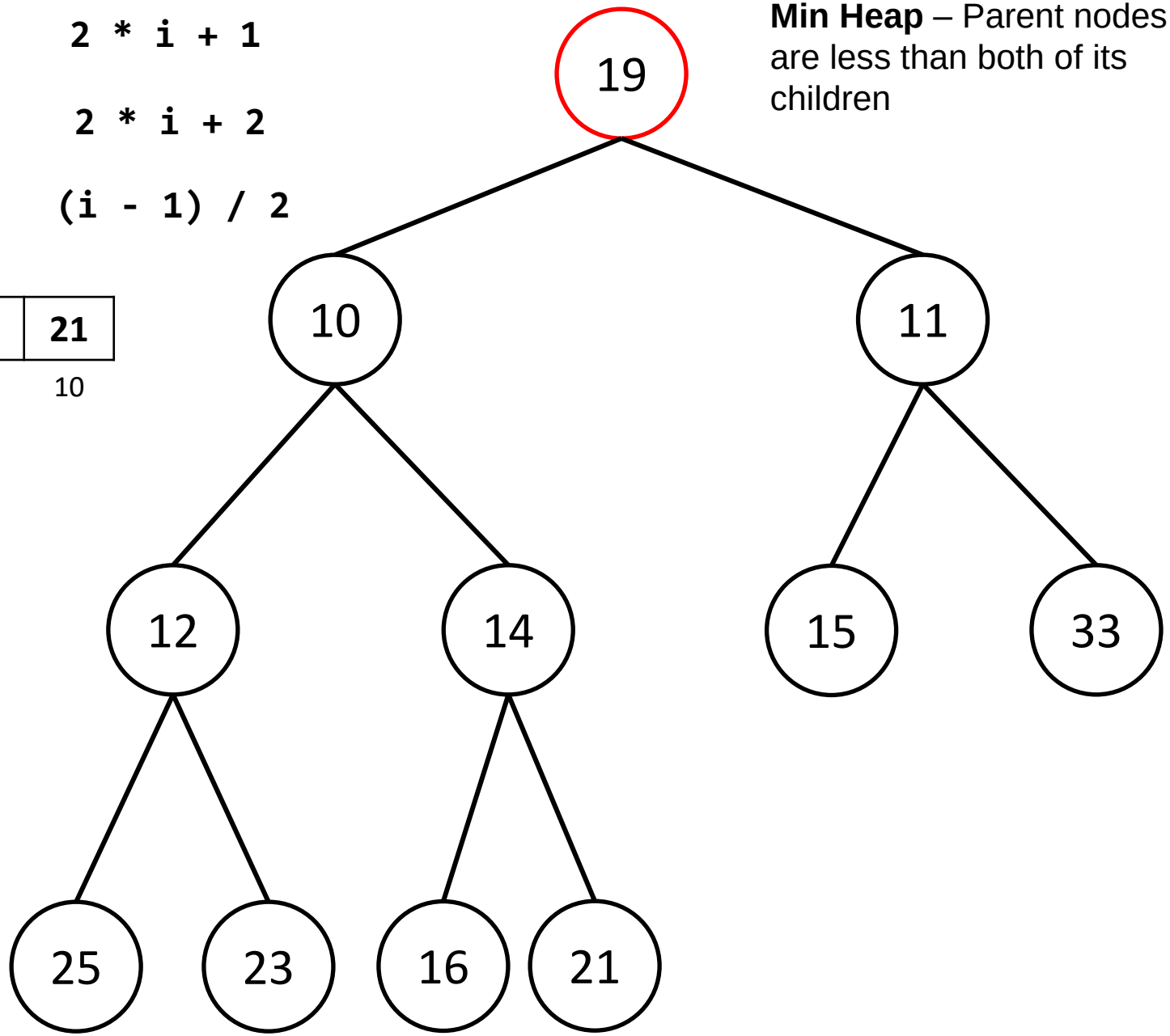
Right Child `2 * i + 2`

Parent `(i - 1) / 2`

Array **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`

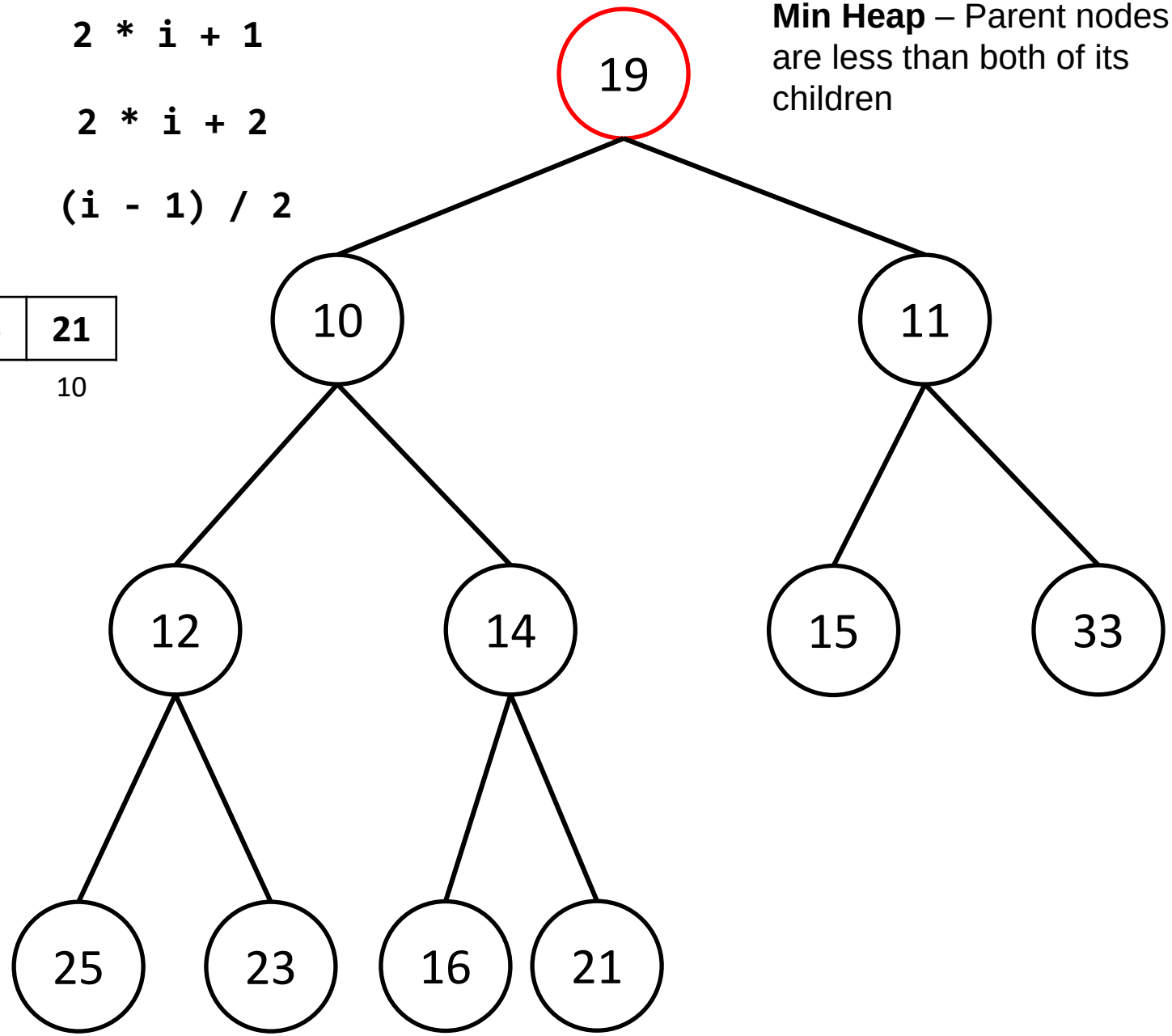
Parent `(i - 1) / 2`

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`

Array

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

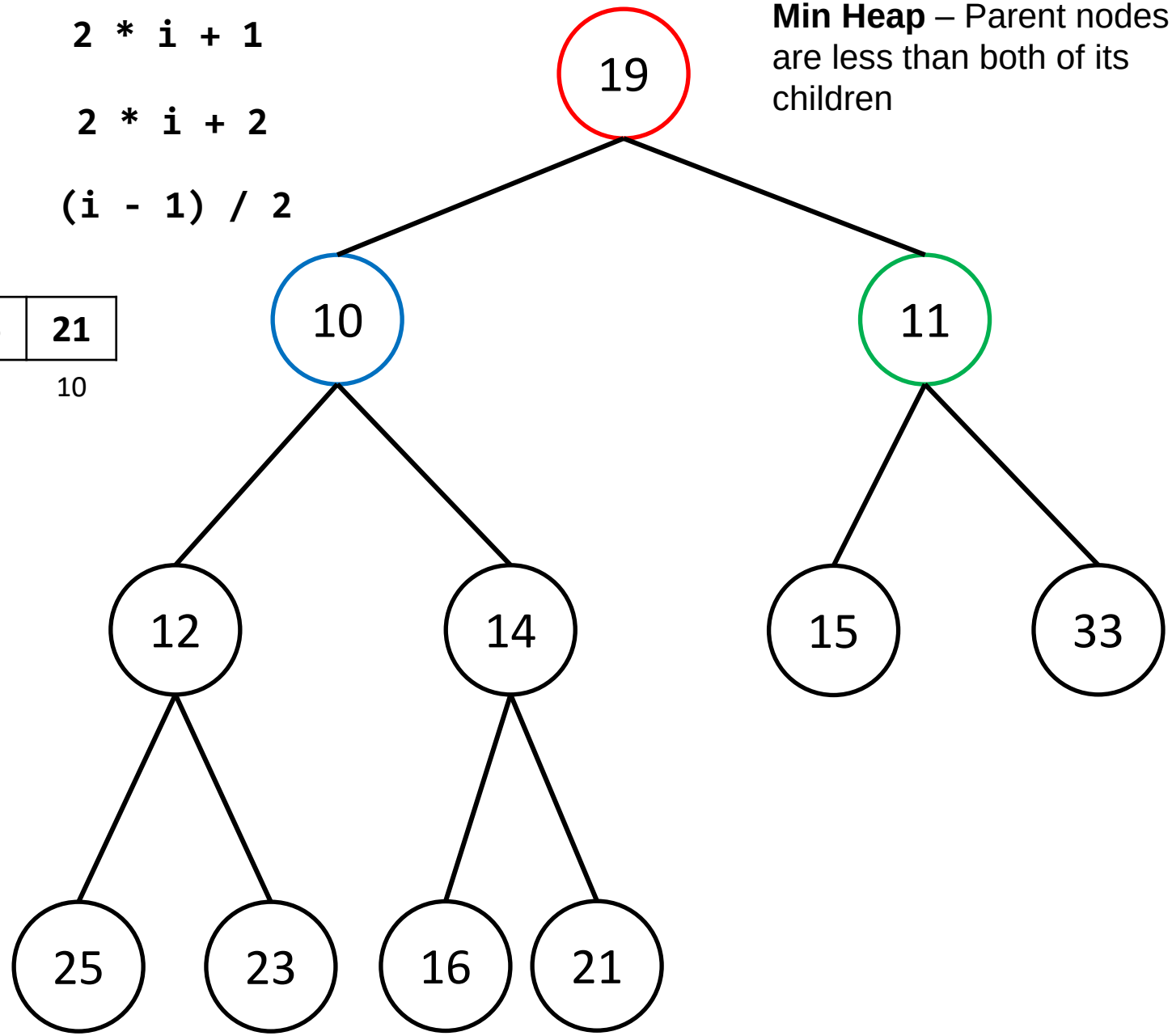
```
poll();
```

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)



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

Heap Representation

Left Child `2 * i + 1`

Right Child `2 * i + 2`

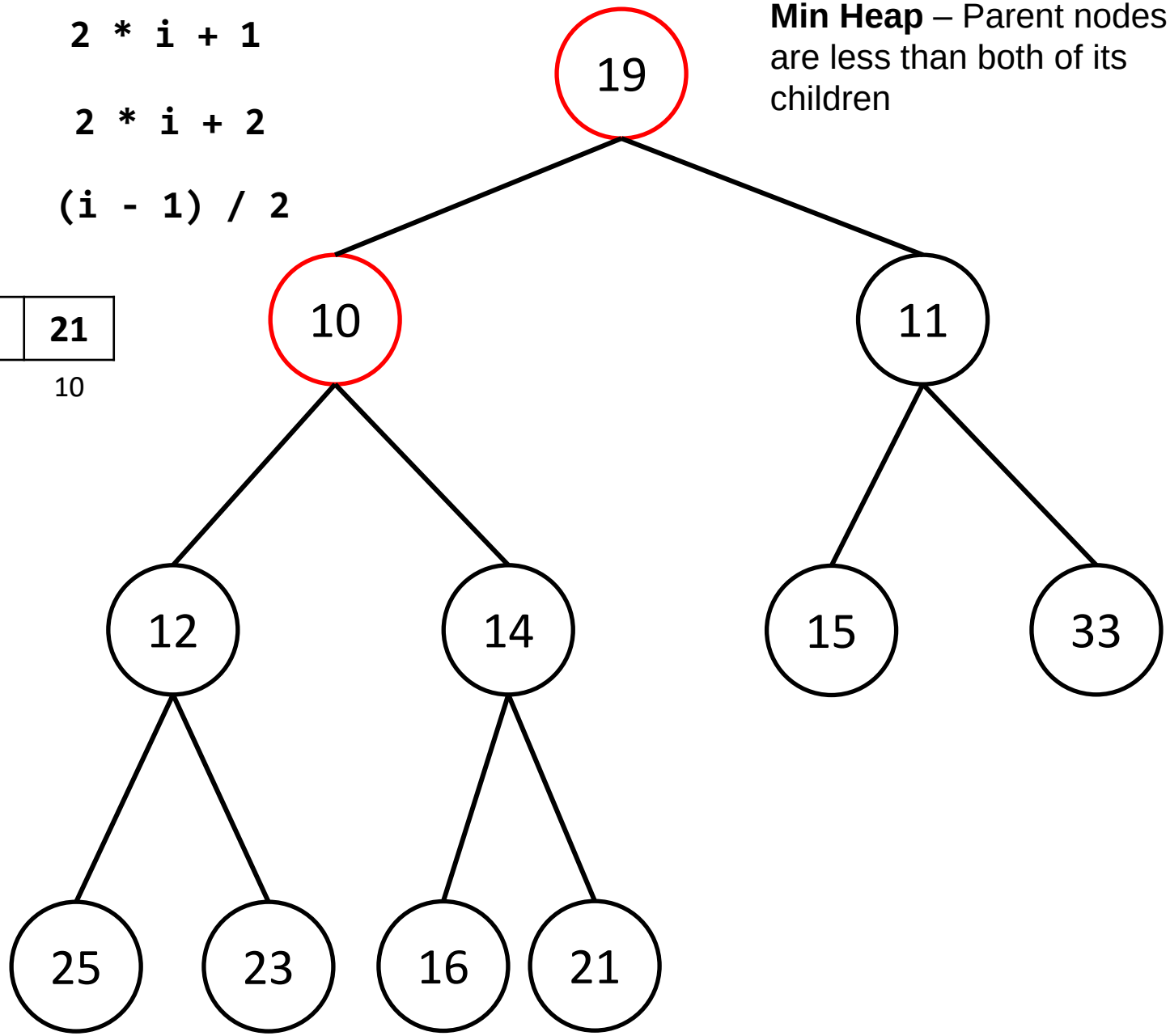
Parent `(i - 1) / 2`

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`

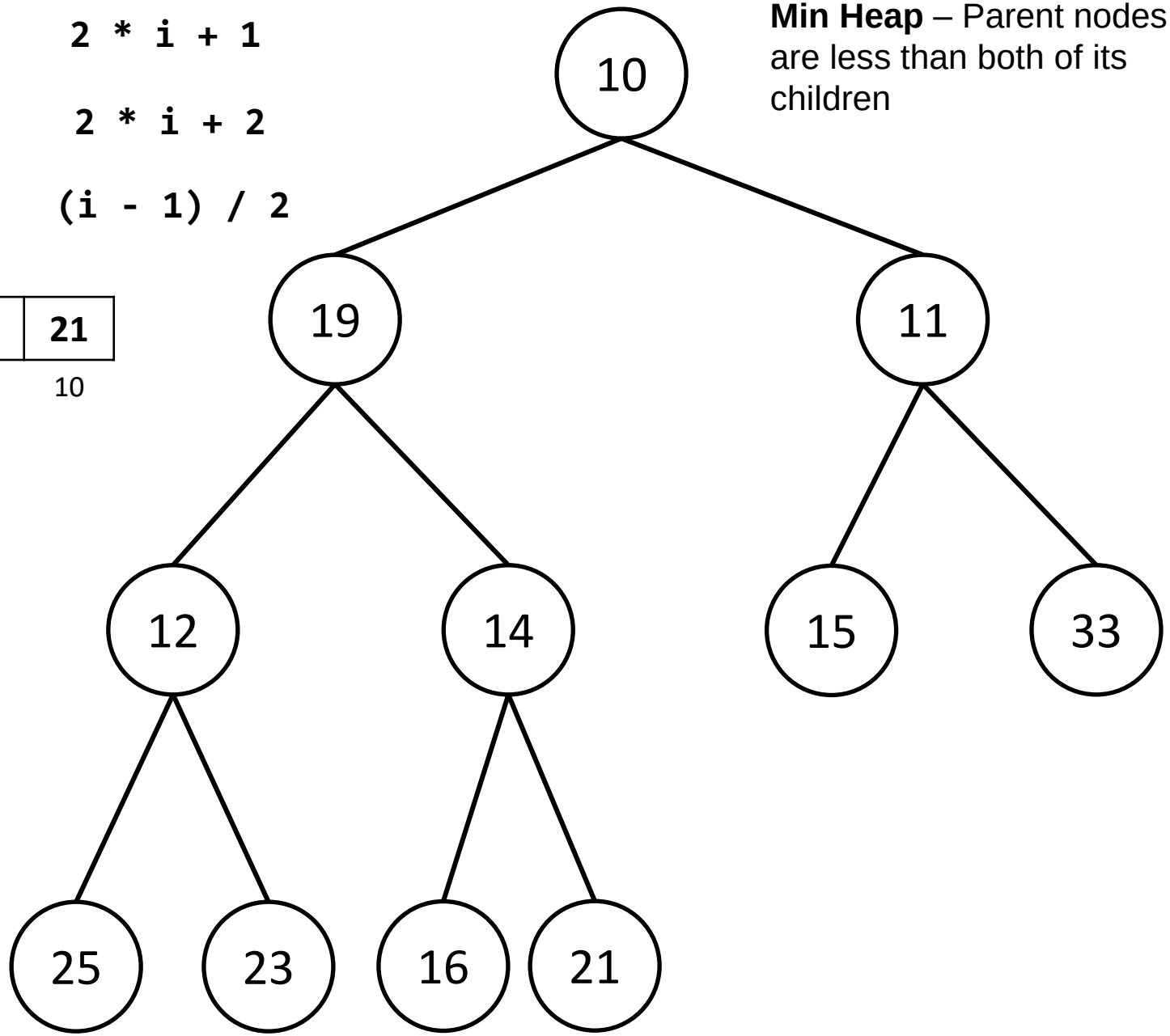
Parent `(i - 1) / 2`

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

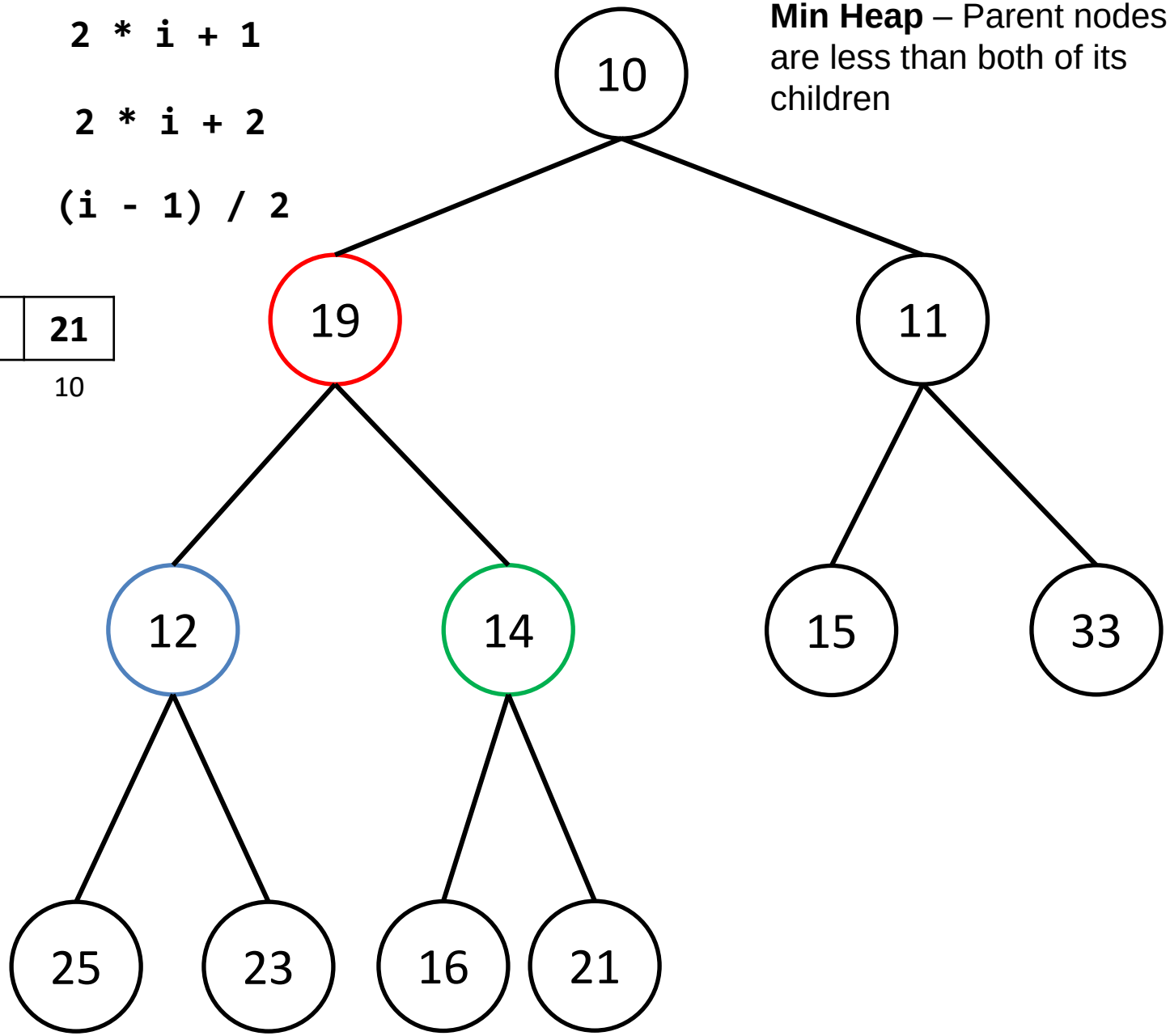
Left Child `2 * i + 1`

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



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

```
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)

Heap Representation

Left Child `2 * i + 1`

Right Child `2 * i + 2`

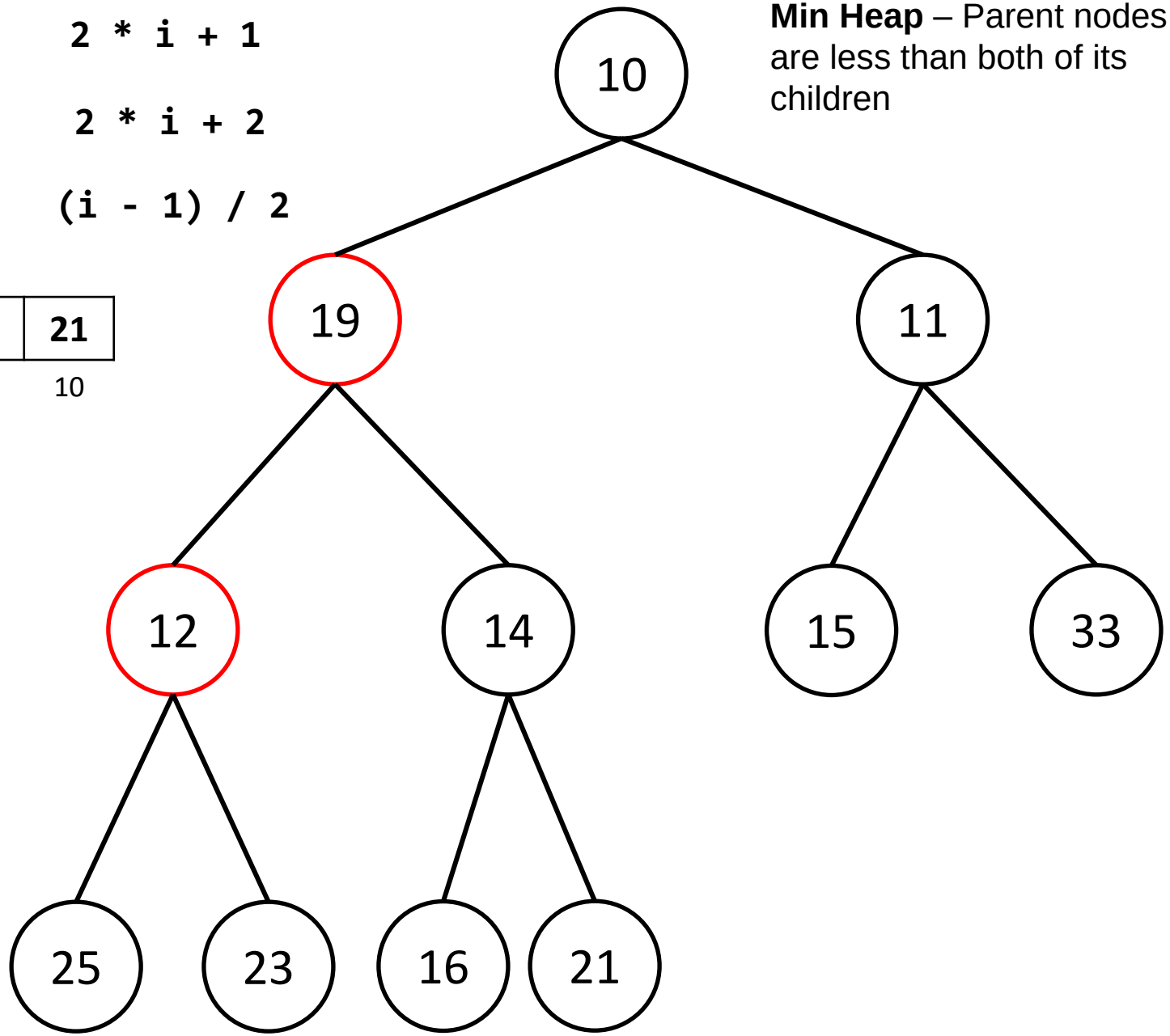
Parent `(i - 1) / 2`

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`

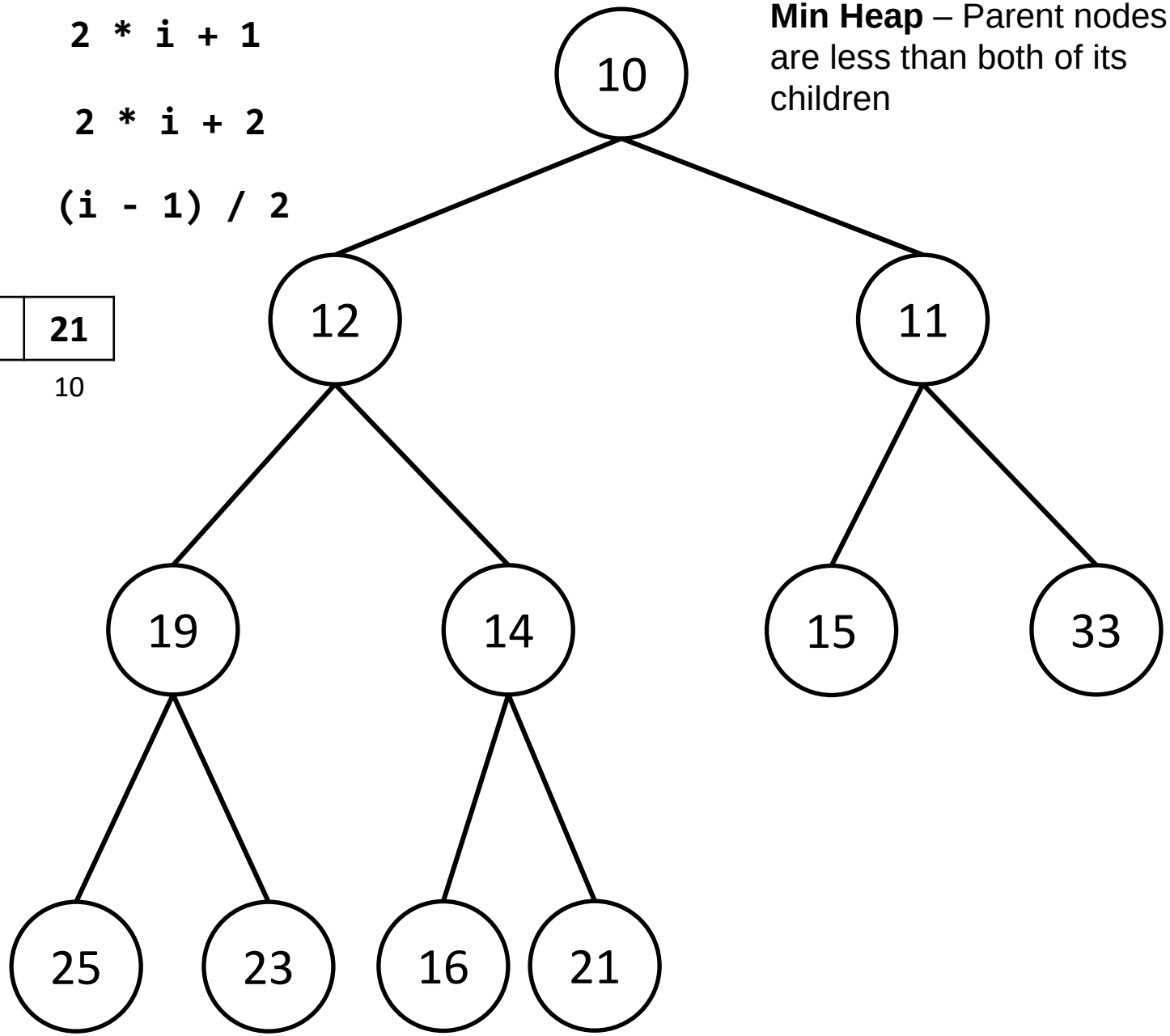
Parent `(i - 1) / 2`

Array

10	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`

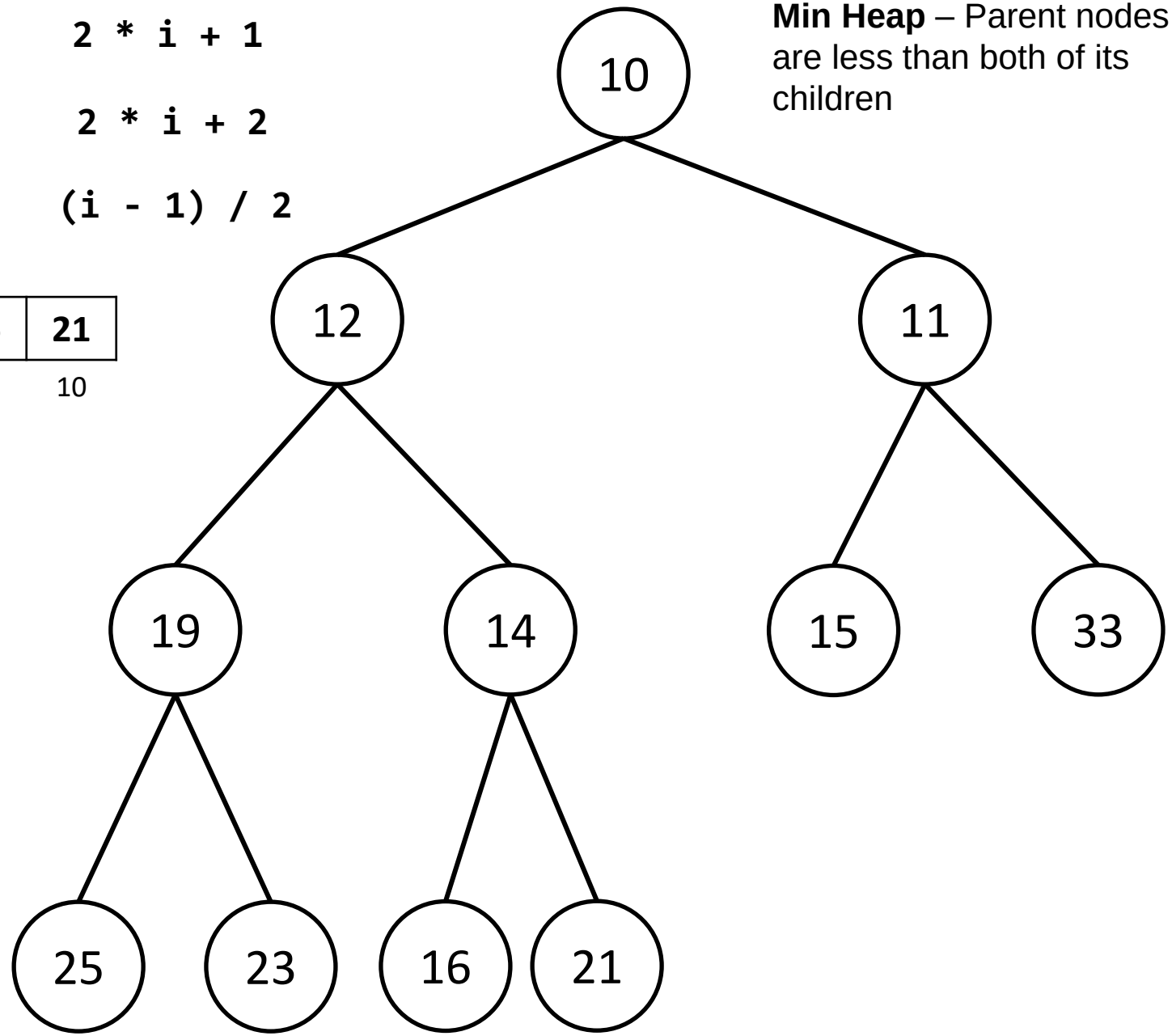
Parent `(i - 1) / 2`

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!



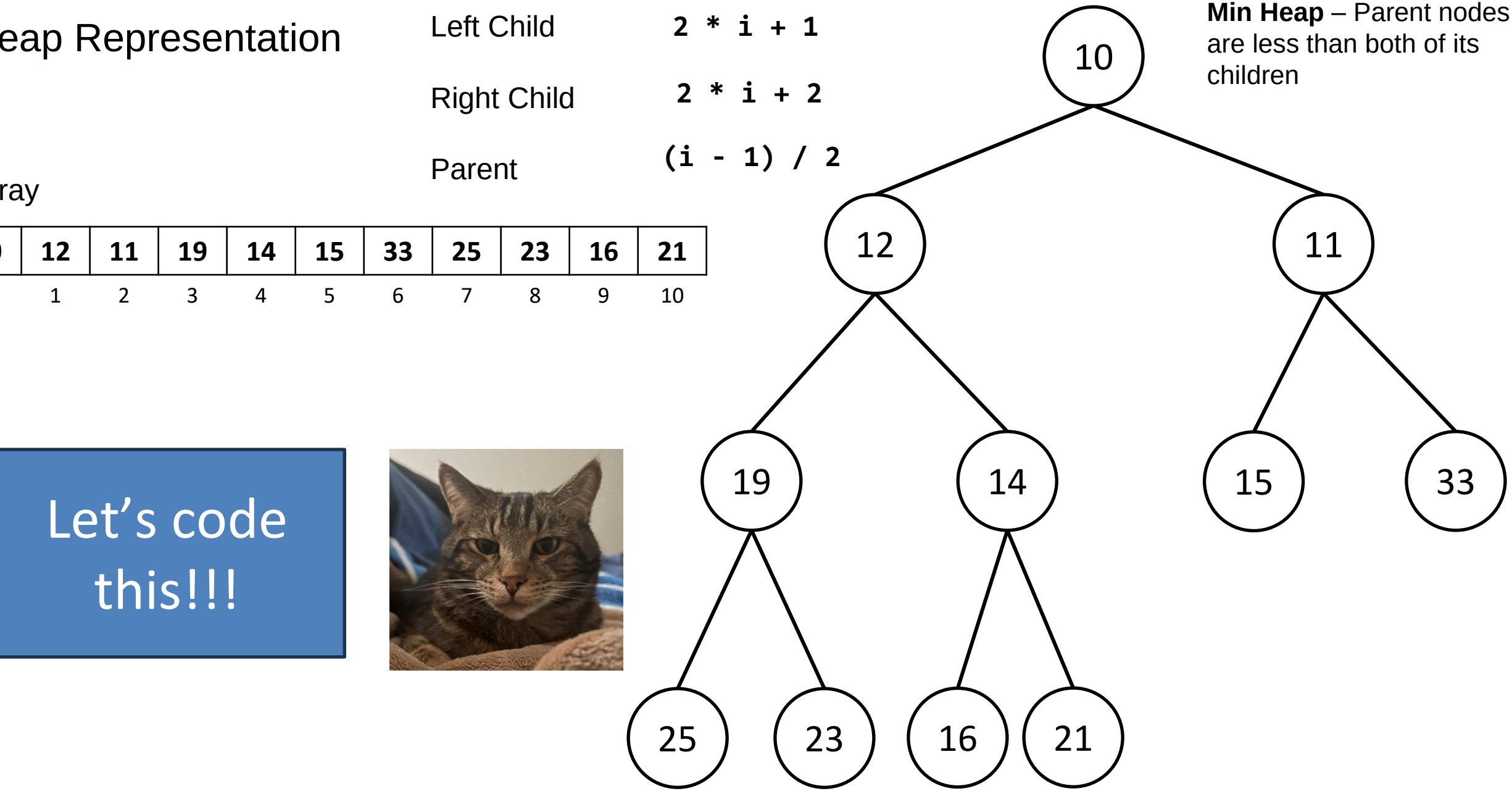
Min Heap – Parent nodes are less than both of its children

Heap Representation

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

Array

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



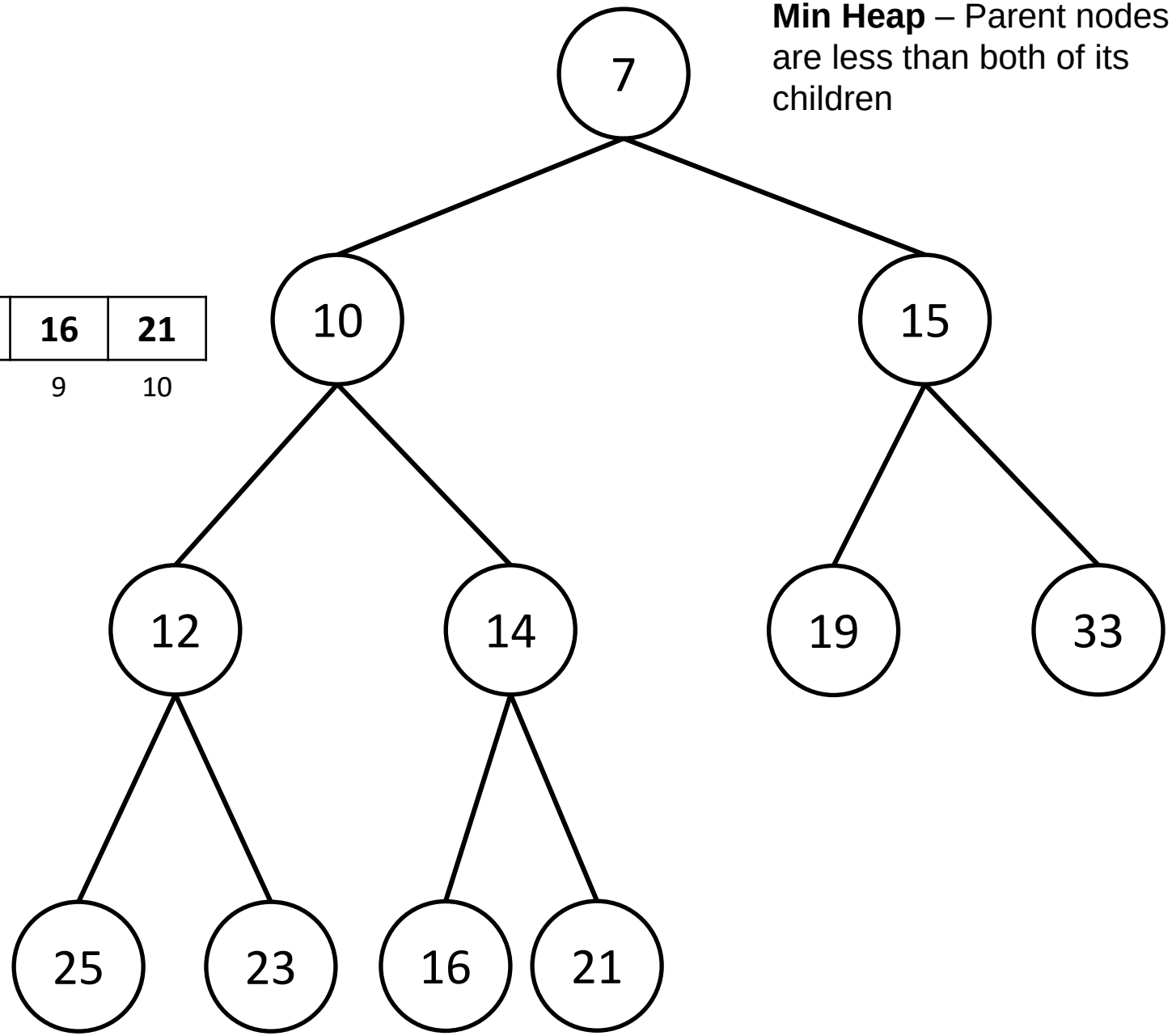
Min Heap – Parent nodes are less than both of its children

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

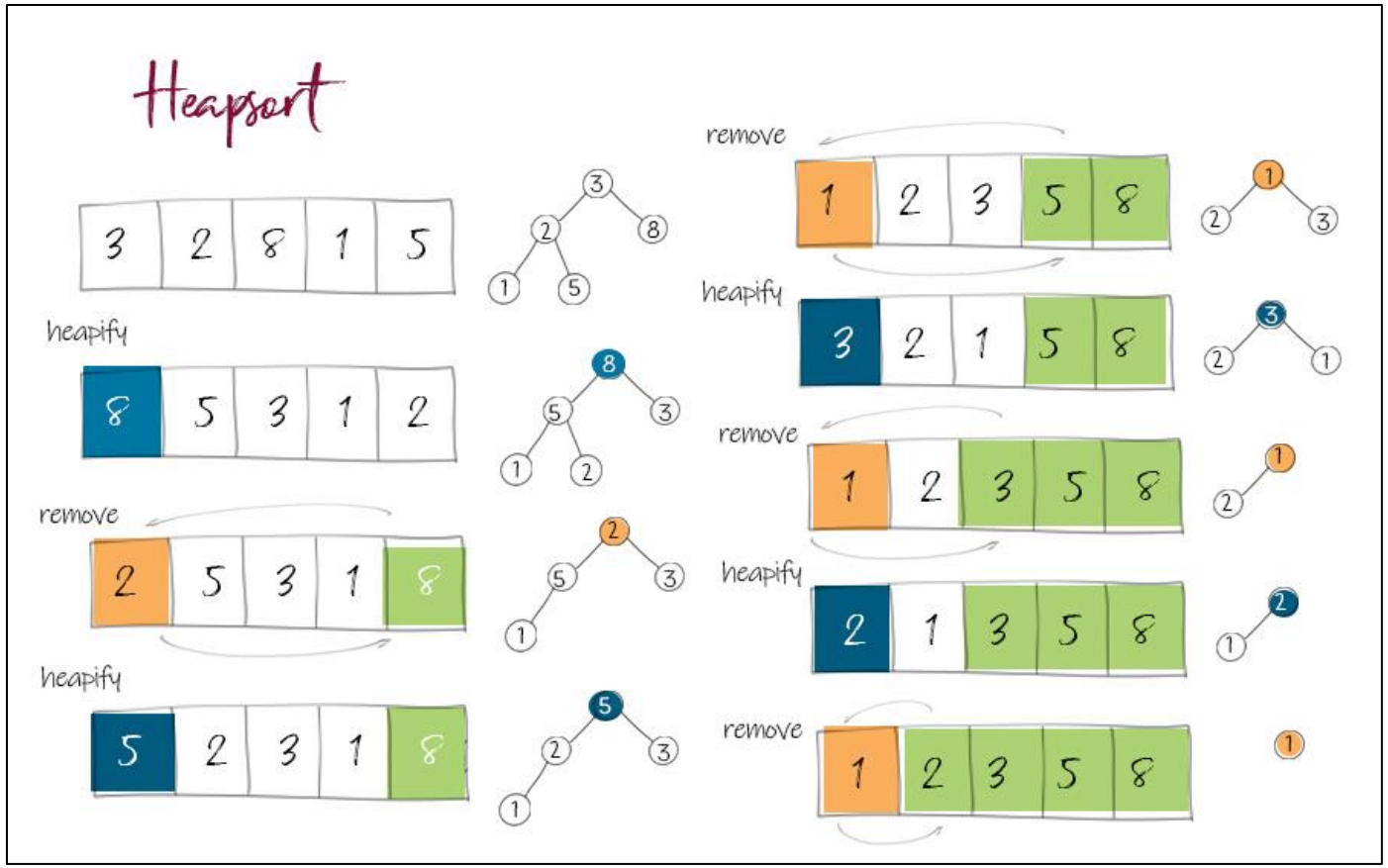
Getting the maximum/minimum value happens in O(1) time

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



("Heapify")

```
Convert unsorted Array to Heap: O(n log n)

Make array: sortedArray          O(n)

while(!heap.isEmpty()):          O(n)

    x = heap.poll()              O(logn)
    insert x into sortedArray    O(1)


return sortedArray               O(1)
```

Total Running Time: **O(n log n)**

(Same as merge sort, quick sort)