CSCI 232:
Data Structures and Algorithms

Red Black Trees

Reese Pearsall
Spring 2025

Announcements

Quiz on Monday (no lecture)

- Taken online via Canvas
- Opens at 6 AM, closes at 11:59 PM
- Not timed, but you only have one attempt

20-ish Questions

- Short Answer
- Multiple Choice
- True/False

Quiz Topics

- Linked Lists, Arrays
- Trees
- Binary Search Trees
- Red Black Trees
- Heaps

Binary Search Tree Running Times

(If we have a way to ensure the BST is balanced)

Operation	Running Time
Insertion	O(logn)
Removal	O(logn)
Searching	O(logn)
Printing	O(n)

n = # of nodes

Binary Search Tree Running Times

(If we have a way to ensure the BST is balanced)

Operation	Running Time
Insertion	O(logn)
Removal	O(logn)
Searching	O(logn)
Printing	O(n)

n = # of nodes

Sorted Array

Operation	Running Time
Insertion	O(n)
Removal	O(n)
Searching	O(logn)
Printing	O(n)

LinkedList

Operation	Running Time
Insertion	O(1)
Removal (by element)	O(n)
Searching	O(n)
Printing	O(n)

Binary Search Tree Running Times

(If we have a way to ensure the BST is balanced)

Operation	Running Time
Insertion	O(logn)
Removal (by element)	O(logn)
Searching	O(logn)
Printing	O(n)

n = # of nodes

Inserting/removal in a BST is faster than inserting into a sorted Array

`Sorted Array`

Operation	Running Time
Insertion	O(n) *(shifting elements)*
Removal (by element)	O(n) *(shifting elements)*
Searching	O(logn) *(binary search)*
Printing	O(n)

`LinkedList`

Operation	Running Time
Insertion	O(1)
Removal (by element)	O(n)
Searching	O(n)
Printing	O(n)

Binary Search Tree Running Times

(If we have a way to ensure the BST is balanced)

Operation	Running Time
Insertion	O(logn)
Removal (by element)	O(logn)
Searching	O(logn)
Printing	O(n)

n = # of nodes

While LinkedLists provide faster insertion times, navigating a Binary Search Tree is faster than a Linked List

(We don't really have a way to start at the "middle" node of a linked and do binary search)

`Sorted Array`

Operation	Running Time
Insertion	O(n)
Removal (by element)	O(n)
Searching	O(logn)
Printing	O(n)

`LinkedList`

Operation	Running Time
Insertion	O(1)
Removal (by element)	O(n) *(linear search)*
Searching	O(n) *(linear search)*
Printing	O(n)

Binary Search Tree Running Times

(If we have a way to ensure the BST is balanced)

Operation	Running Time
Insertion	O(logn)
Removal (by element)	O(logn)
Searching	O(logn)
Printing	O(n)

n = # of nodes

Which is the best tool for the job?

Depends on what you need!

Sorted Array

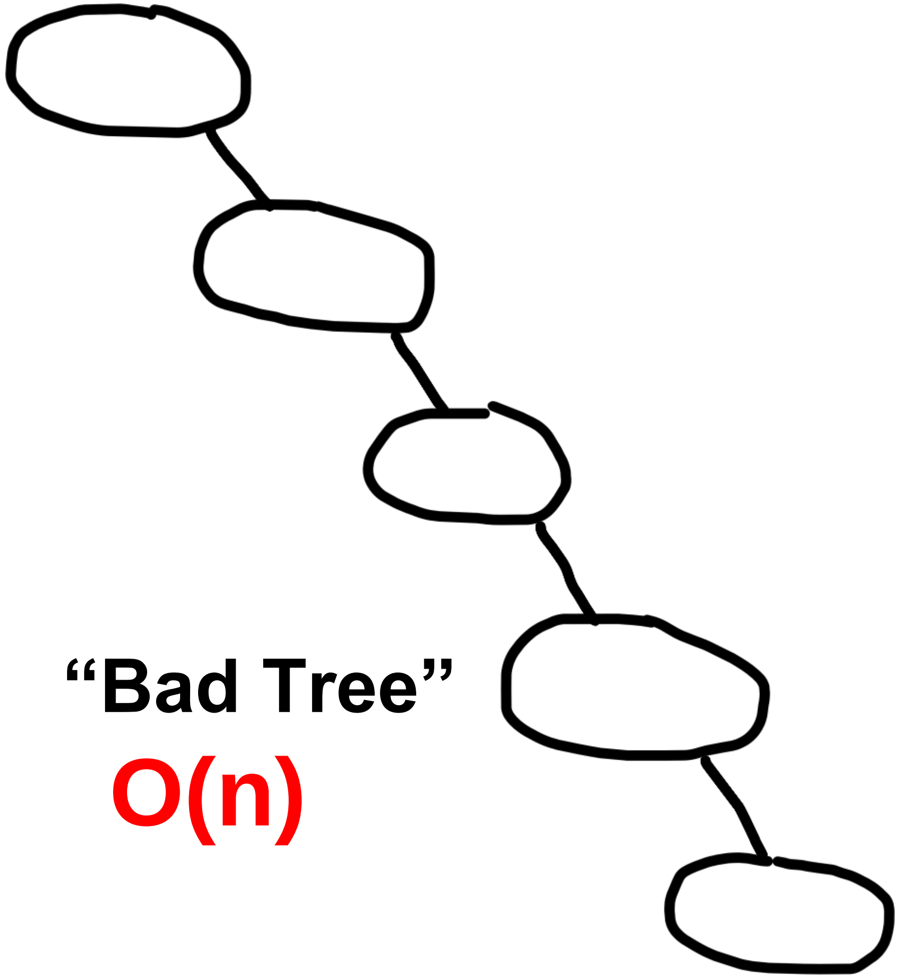
Operation	Running Time
Insertion	O(n)
Removal (by element)	O(n)
Searching	O(logn)
Printing	O(n)

LinkedList

Operation	Running Time
Insertion	O(1)
Removal (by element)	O(n)
Searching	O(n)
Printing	O(n)

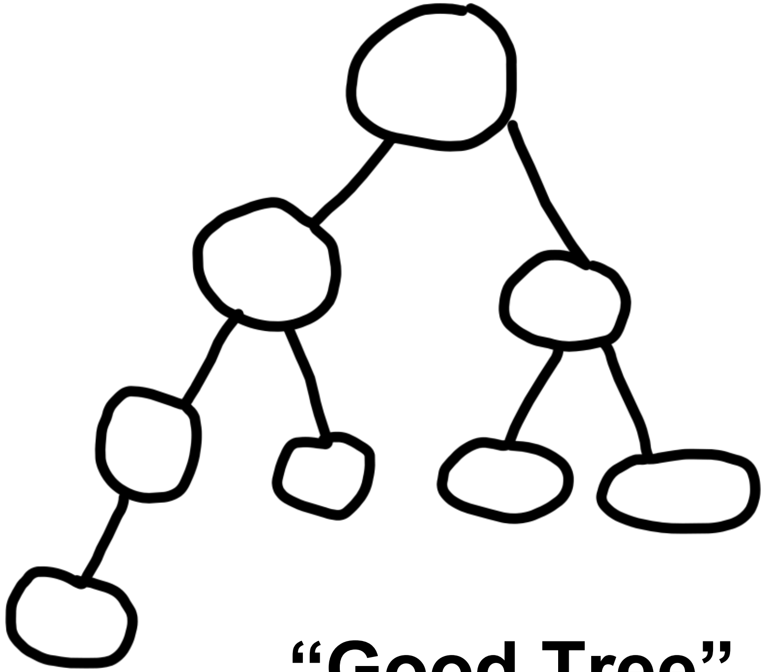
Binary Search Tree – Insertion/Searching/Removing

Running time?



“Bad Tree”

O(n)

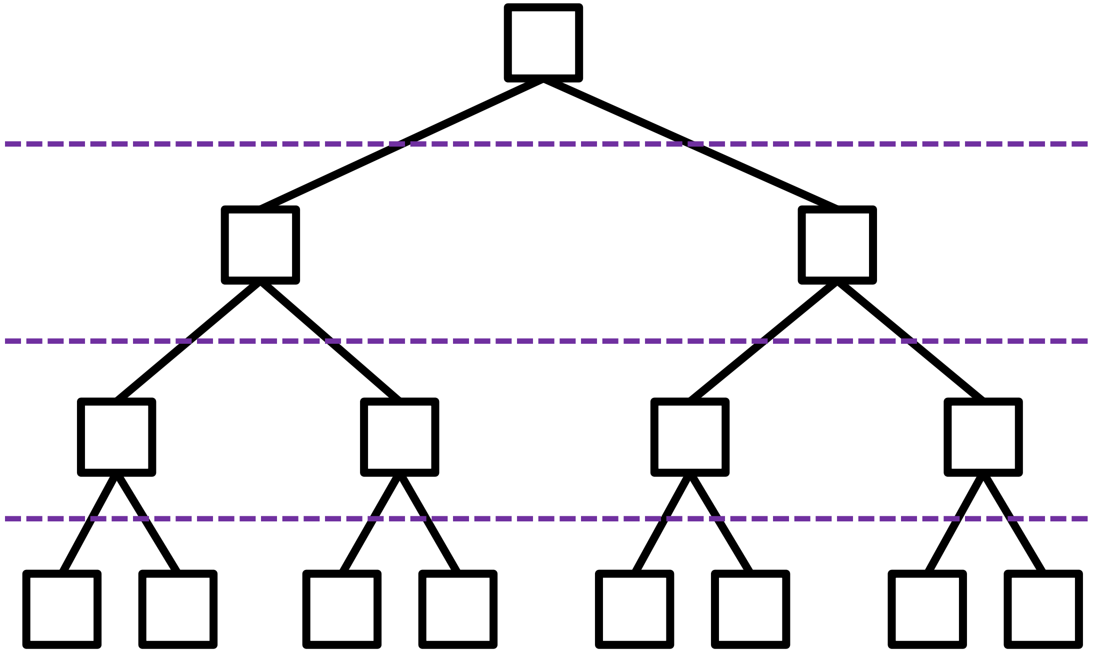


“Good Tree”

O(logn)

Balanced BST

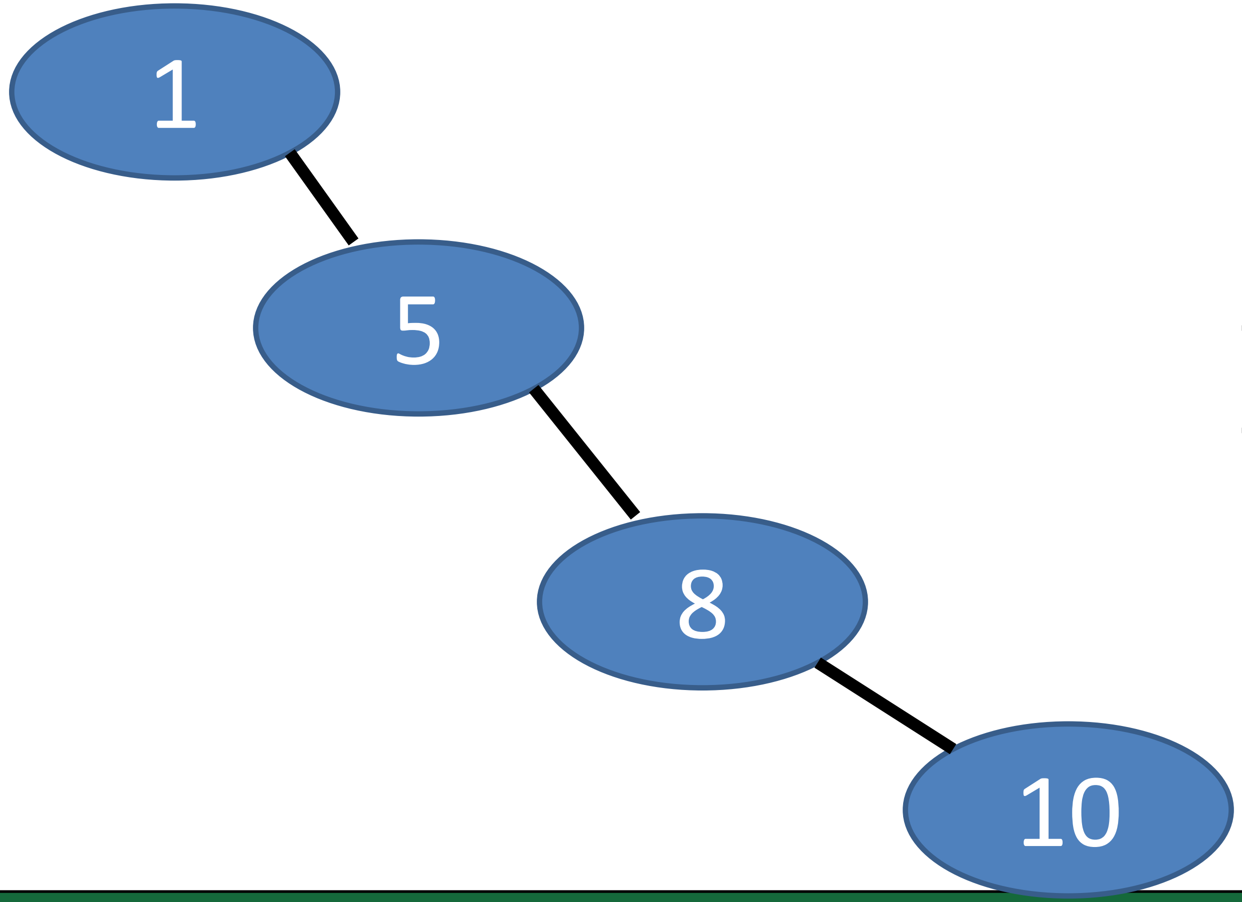
A **balanced** binary tree, is defined as a binary tree in which given n nodes, the height of the tree is `O(logn)`.



Depth	Num Nodes
0	1
1	2
2	4
3	8

Balanced BST

A **balanced** binary tree, is defined as a binary tree in which given n nodes, the height of the tree is `O(logn)`.

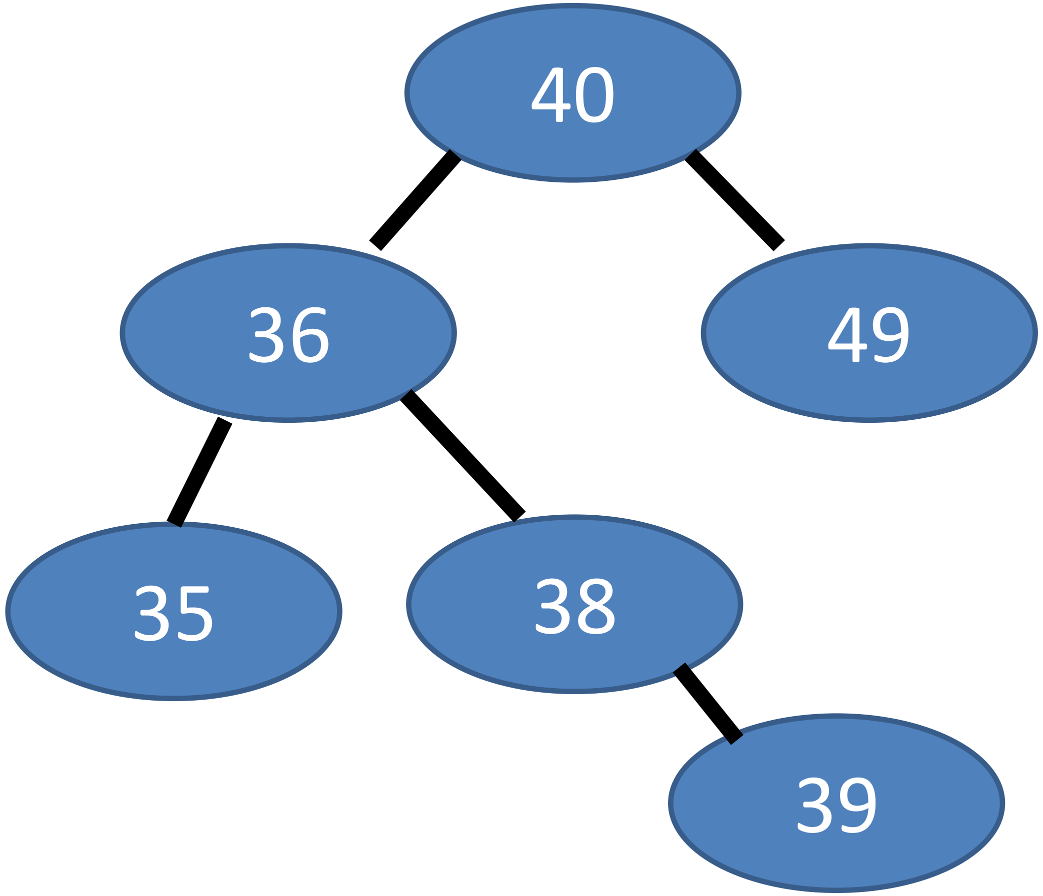


4 nodes
→ If this is a balanced tree, the height should be less than or equal to 2 (log(4))

Height = 3 → not balanced

Balanced BST

A **balanced** binary tree, is defined as a binary tree in which given n nodes, the height of the tree is **`O(logn)`**.

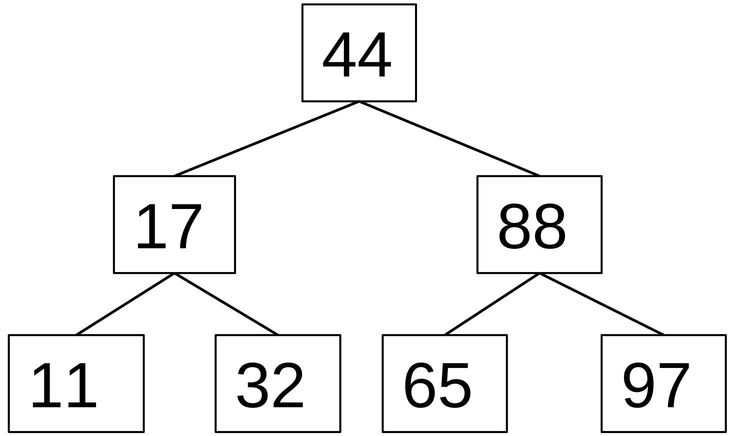


6 nodes
→ If this is a balanced tree, the height should be less than or equal to 3
ceil(log(6))

Height = 3 → balanced

Balanced BST

If we are building a BST, there is no guarantee that the tree will be balanced (it depends on the order that we add nodes)



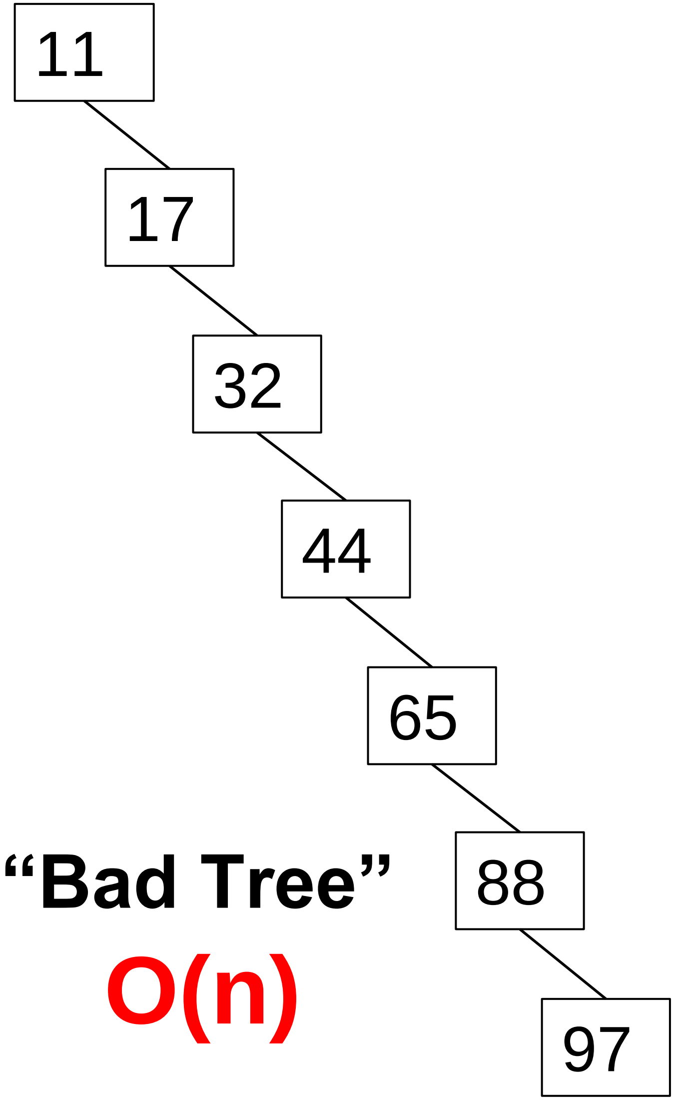
44, 17, 88, 11, 32, 65, 97

44, 17, 32, 88, 11, 97, 65

44, 88, 65, 97, 17, 32, 11

“Good Tree”

O(logn)



“Bad Tree”

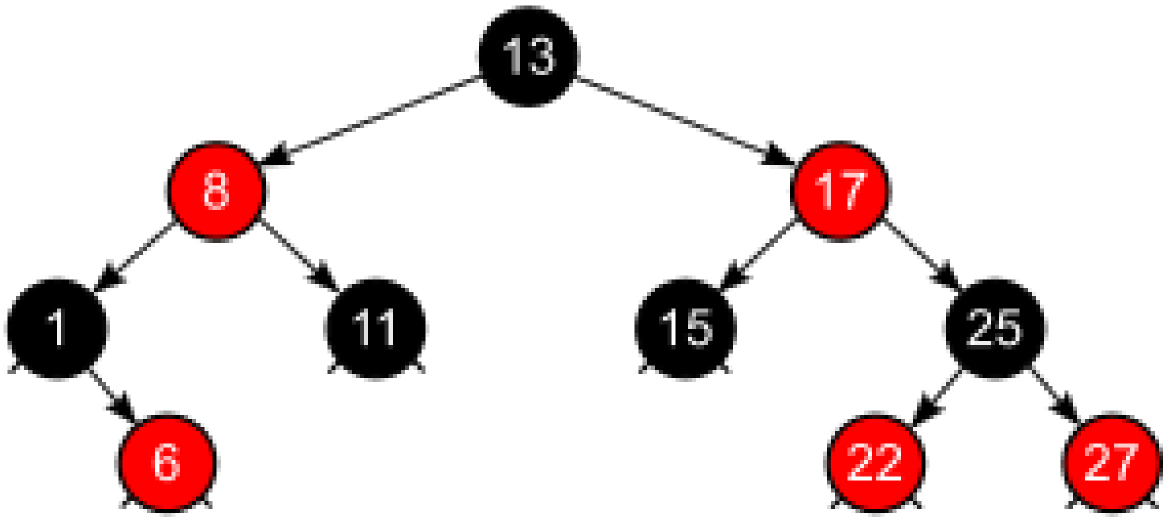
O(n)

Balanced BST

Red-Black Trees are a type of BST with some more rules, and if we follow the rules, we will be guaranteed a balanced BST

Guaranteed Balanced BST =

- **O(logn)** insertion time
- **O(logn)** deletion time
- **O(logn)** searching time

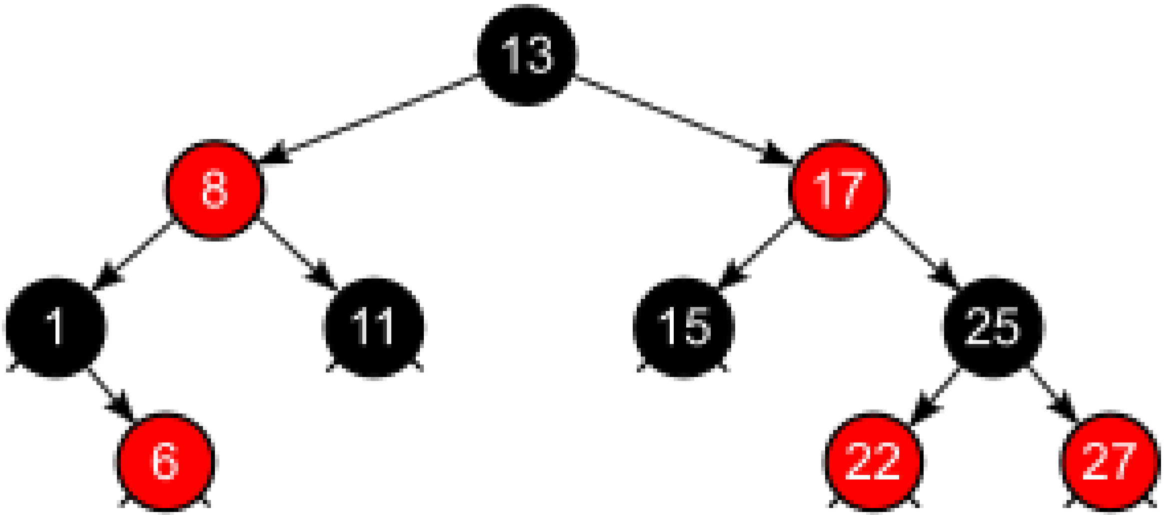


Balanced BST

Because a RBT is a BST, we still need to make sure

- Everything to the left of the node is less than the node
- Everything to the right of the node is greater than the node
- A node cannot have more than two children
- No duplicate nodes

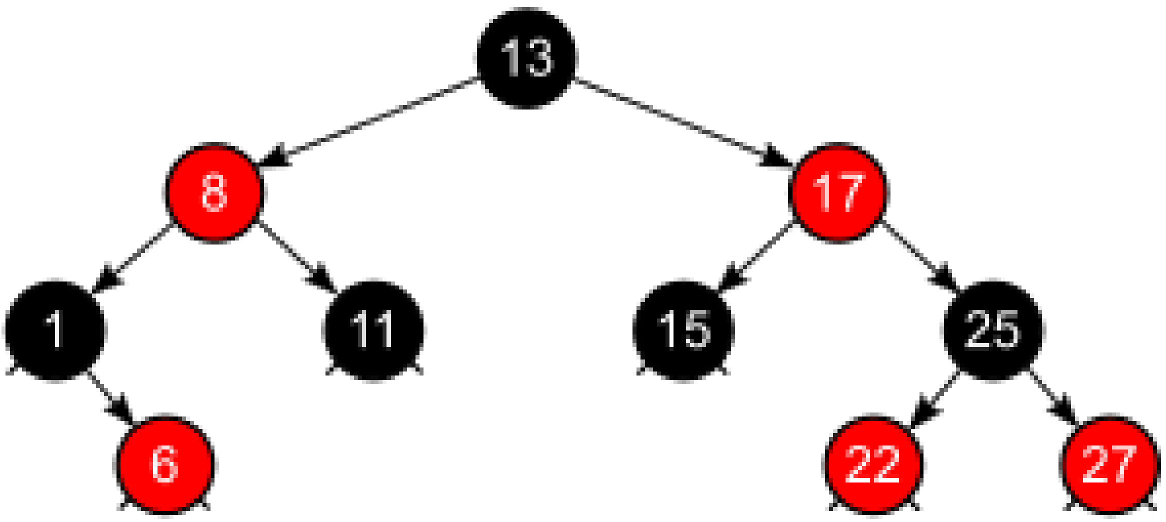
(BST Rules)



Red-Black Tree Rules

Each Node now has a **color** (red or black)

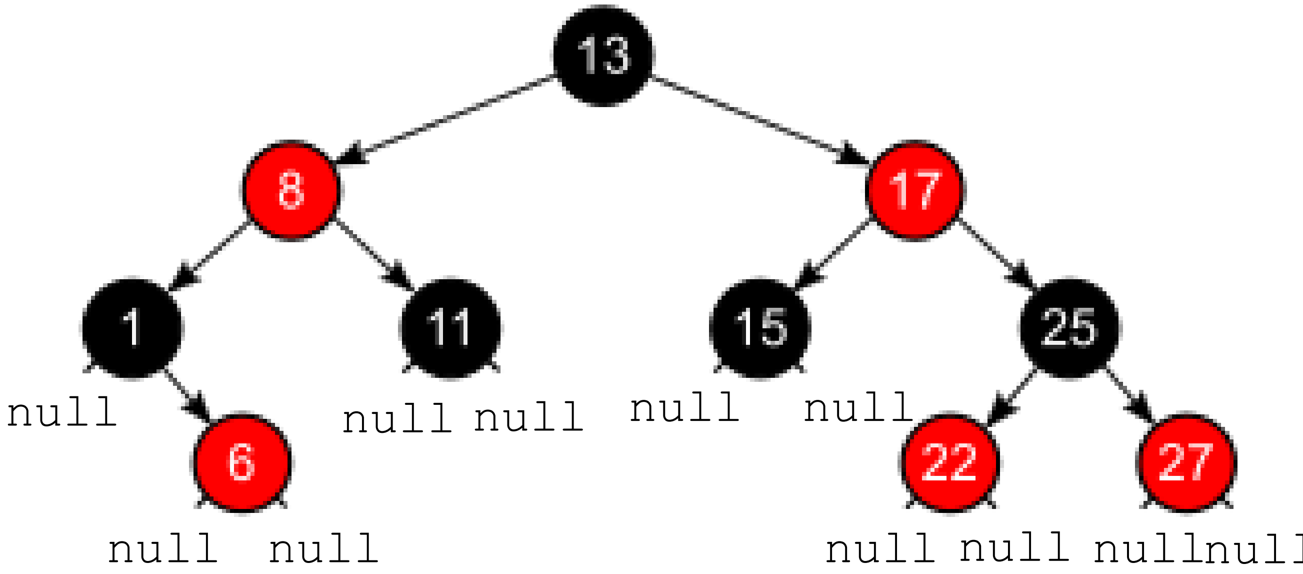
1. Every node is either **red** or **black**



Red-Black Tree Rules

Each Node now has a **color** (red or black)

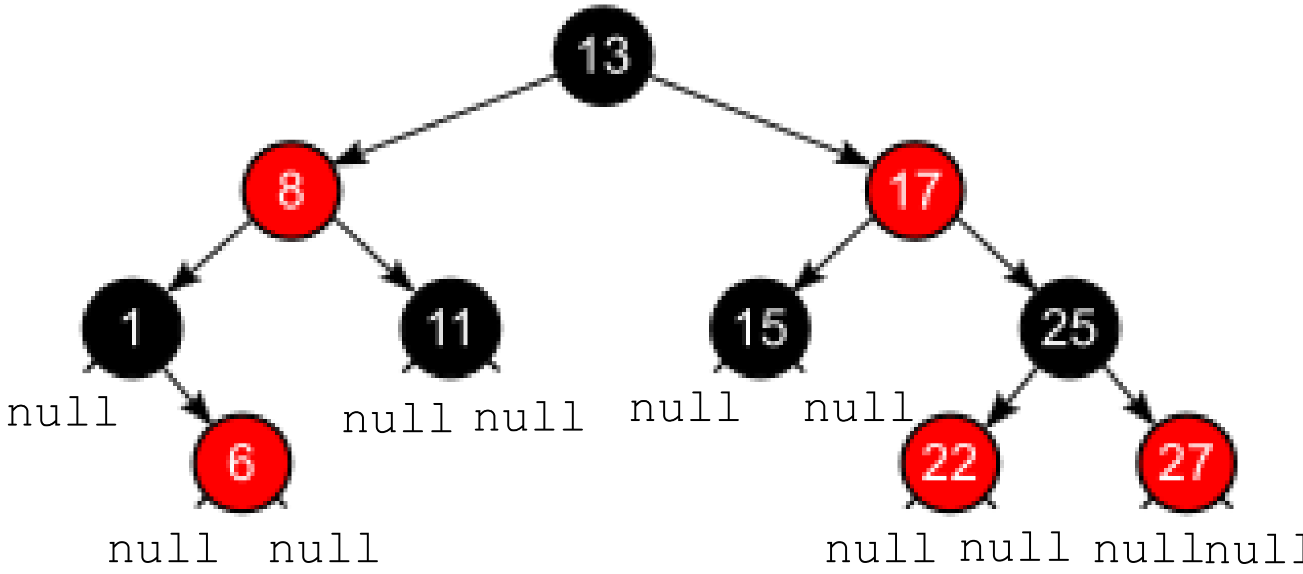
1. Every node is either **red** or **black**
2. The `null` children are **black**



Red-Black Tree Rules

Each Node now has a **color** (red or black)

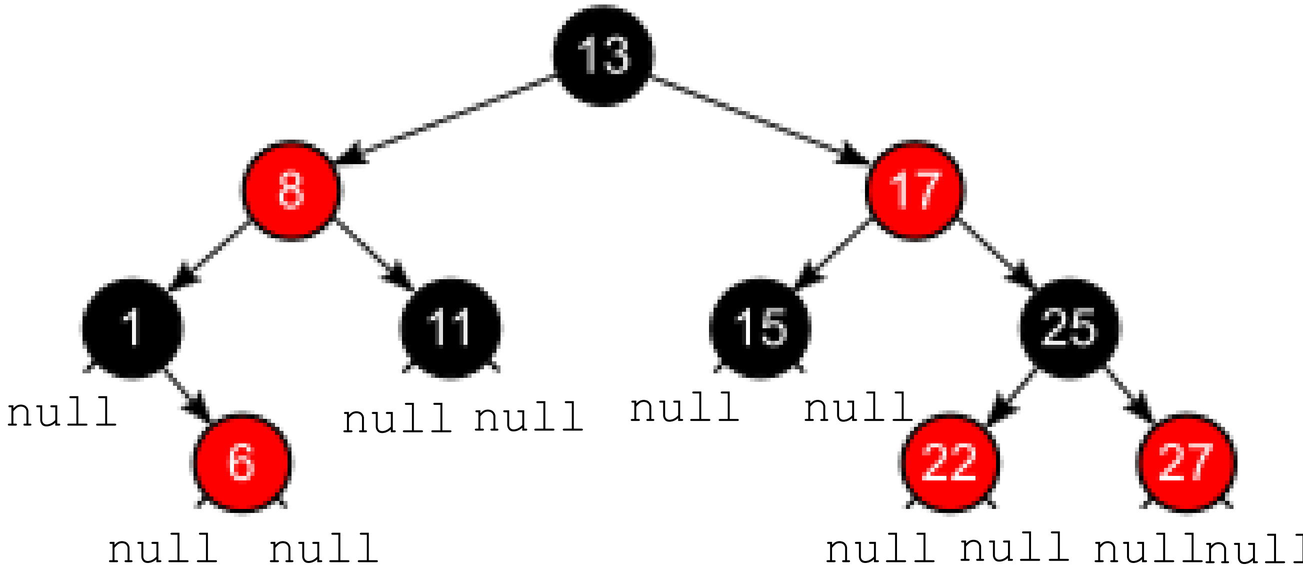
1. Every node is either **red** or **black**
2. The `null` children are **black**
3. The root node is **black**



Red-Black Tree Rules

Each Node now has a **color** (red or black)

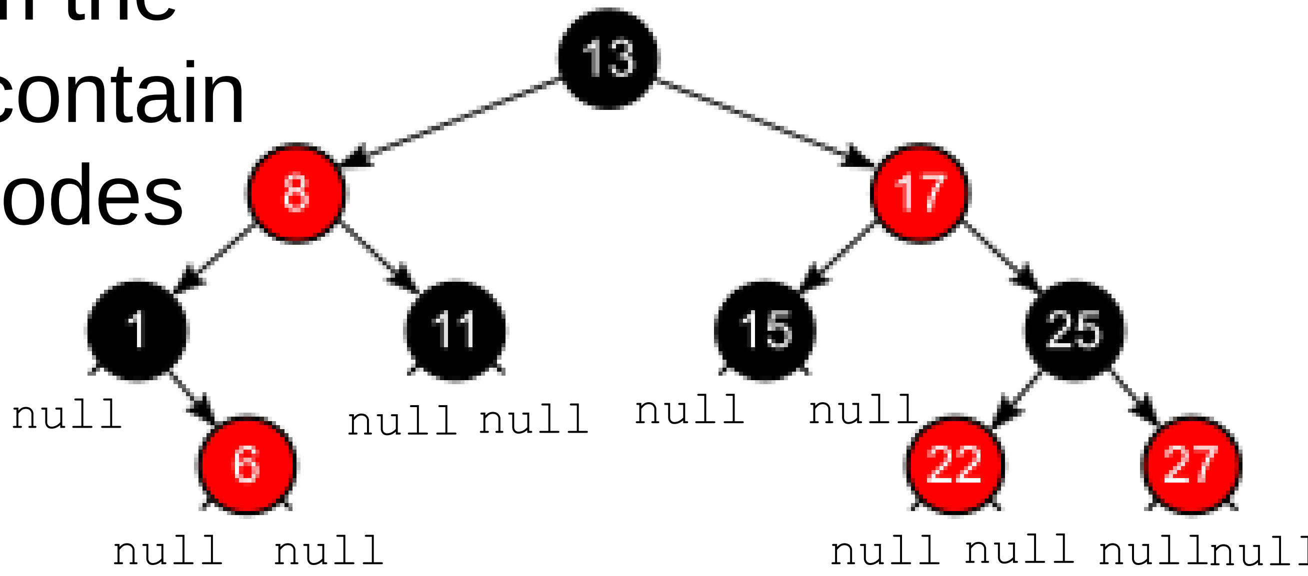
1. Every node is either **red** or **black**
2. The `null` children are **black**
3. The root node is **black**
4. If a node is **red**, both children must be **black**



Red-Black Tree Rules

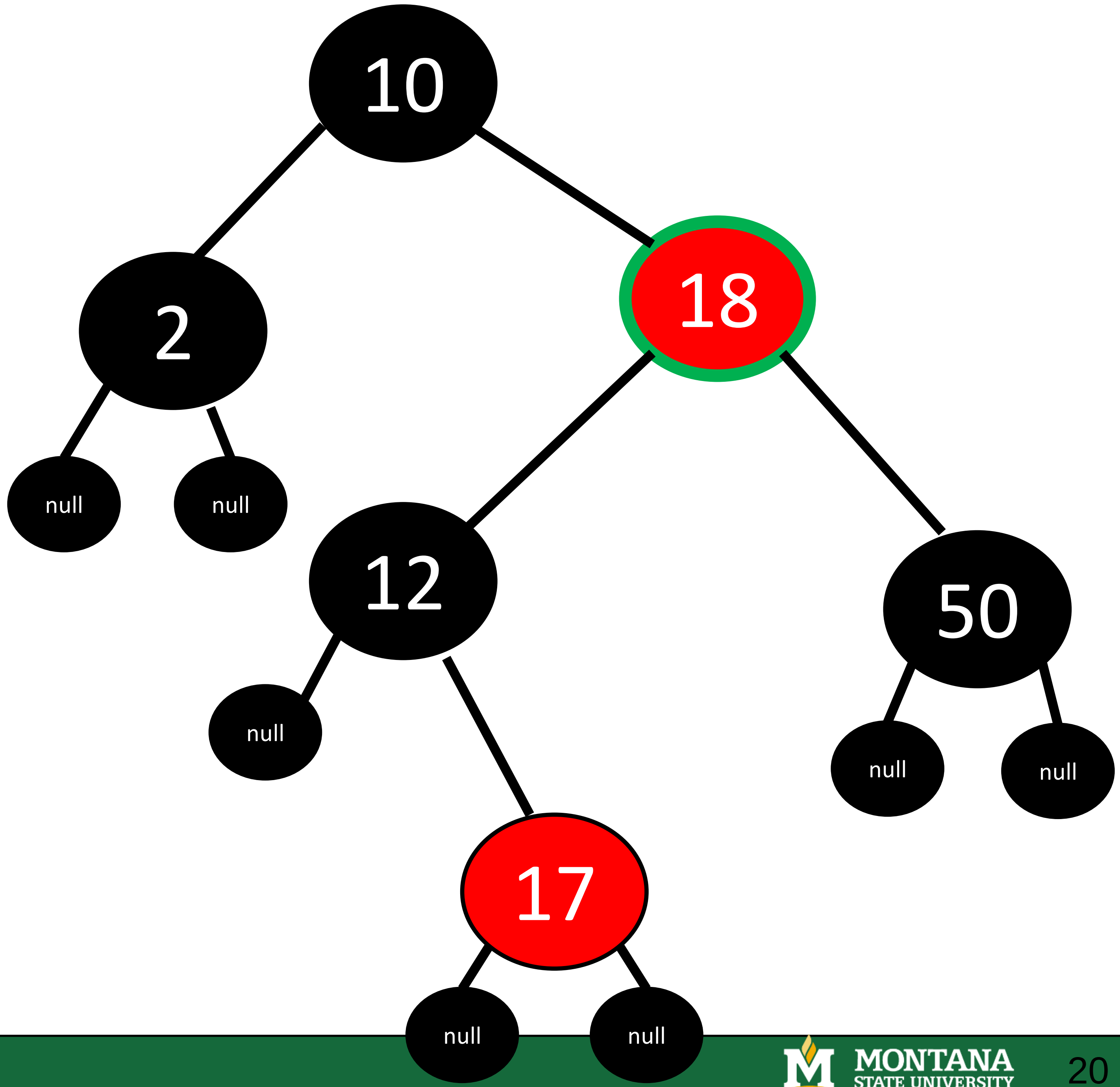
Each Node now has a **color** (red or black)

1. Every node is either **red** or **black**
2. The `null` children are **black**
3. The root node is **black**
4. If a node is **red**, both children must be **black**
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes



Red-Black Tree Rules

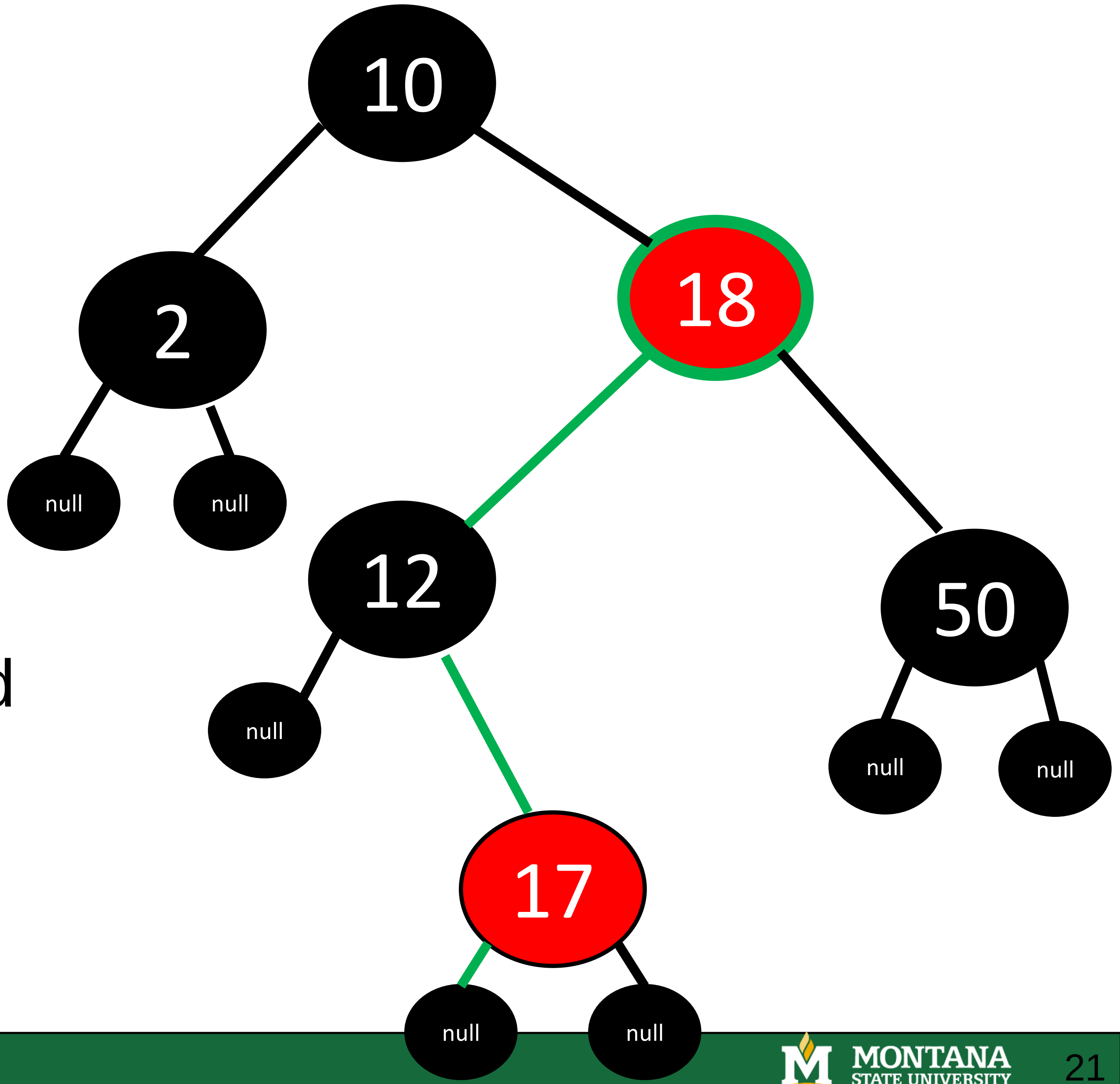
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes



Red-Black Tree Rules

5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

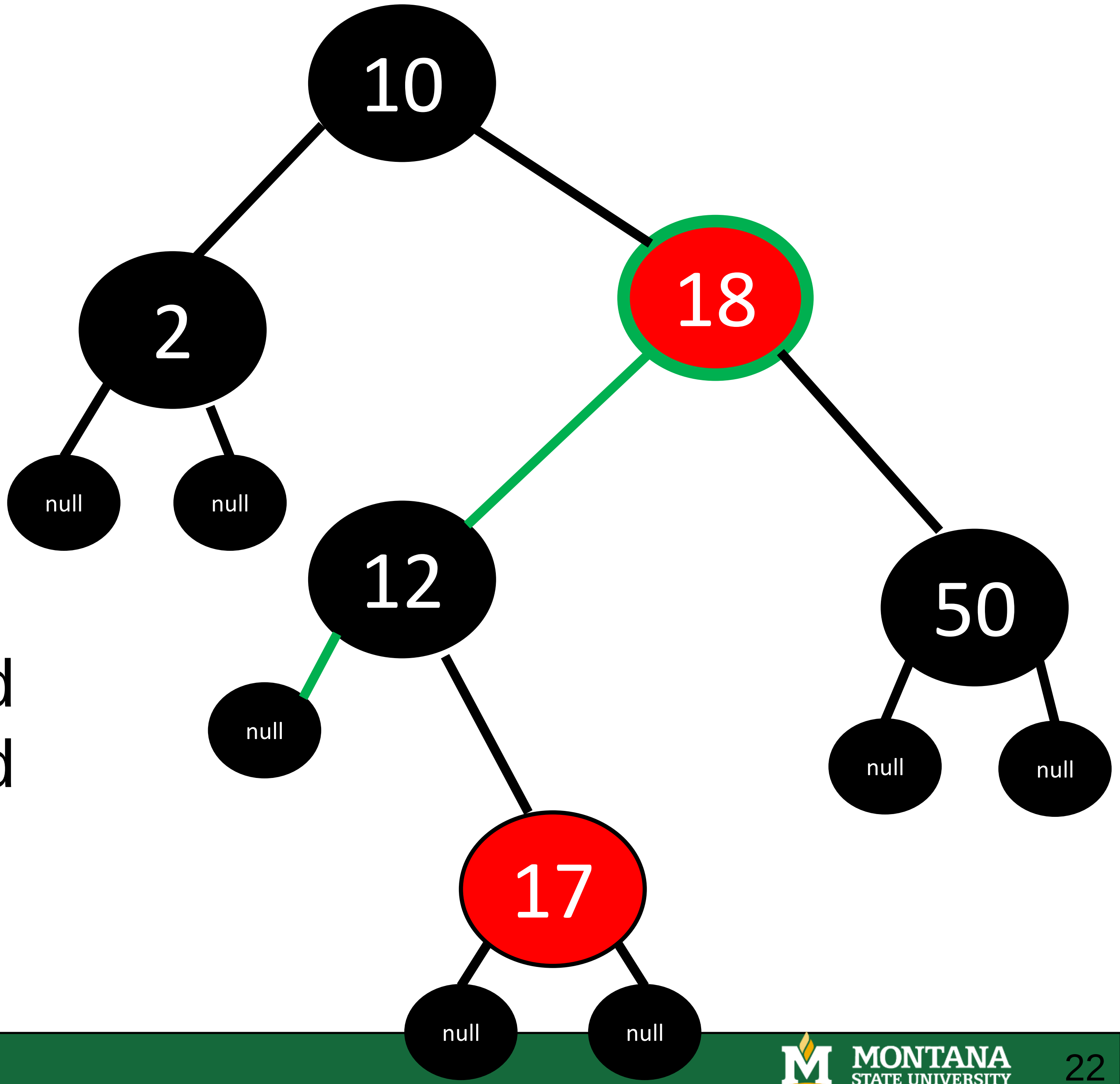
Path 1: 2 black nodes visited



Red-Black Tree Rules

5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

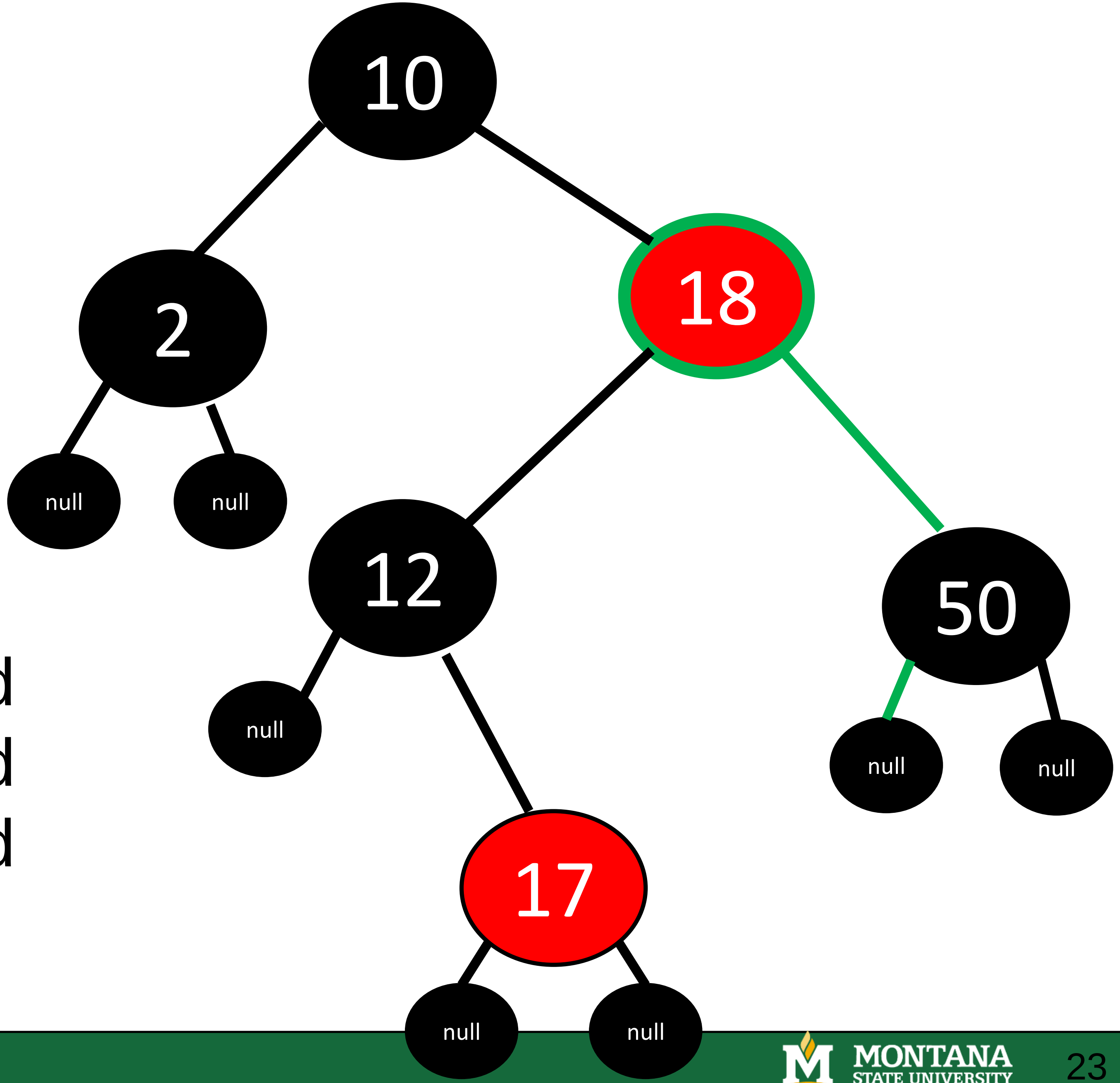
Path 1: 2 black nodes visited
Path 2: 2 black nodes visited



Red-Black Tree Rules

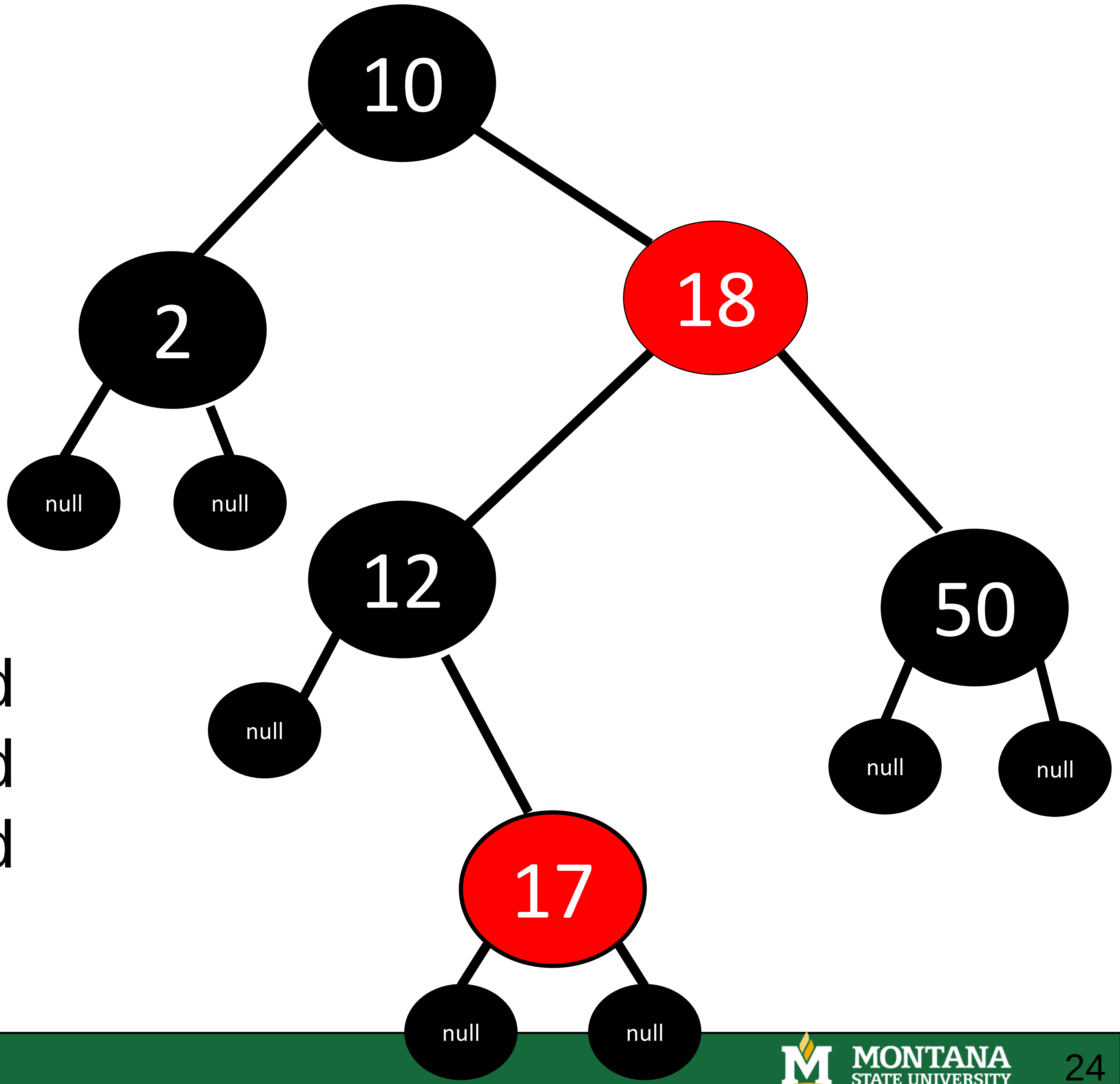
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

Path 1: 2 black nodes visited
Path 2: 2 black nodes visited
Path 3: 2 black nodes visited



Red-Black Tree Rules

5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

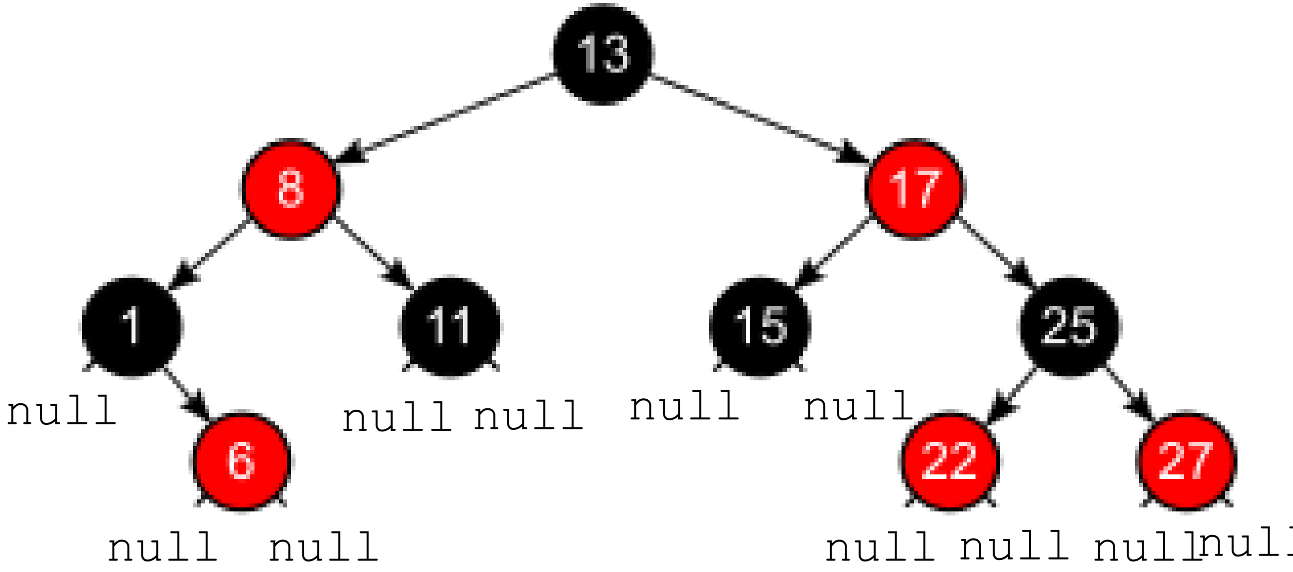


Path 1: 2 black nodes visited
Path 2: 2 black nodes visited
Path 3: 2 black nodes visited

Red-Black Tree Rules

1. Every node is either **red** or **black**
2. The `null` children are **black**
3. The root node is **black**
4. If a node is **red**, both children must be **black**
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

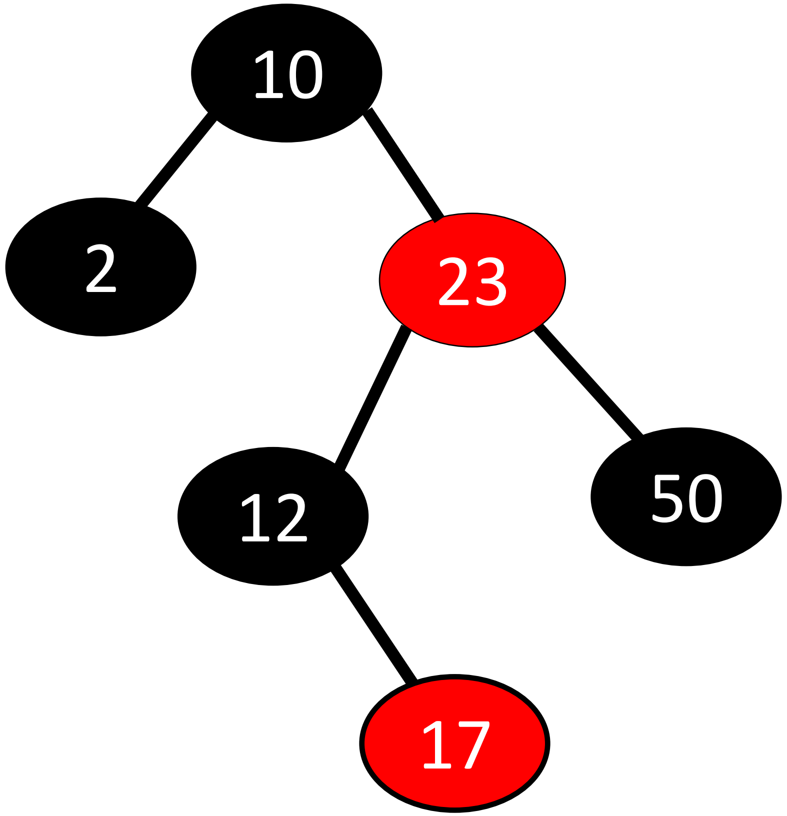
When we **insert** or **delete** something from a Red-Black tree, the new tree may **violate** one of these rules



Red-Black Tree Insertion/Deletion

```
insert(15)
```

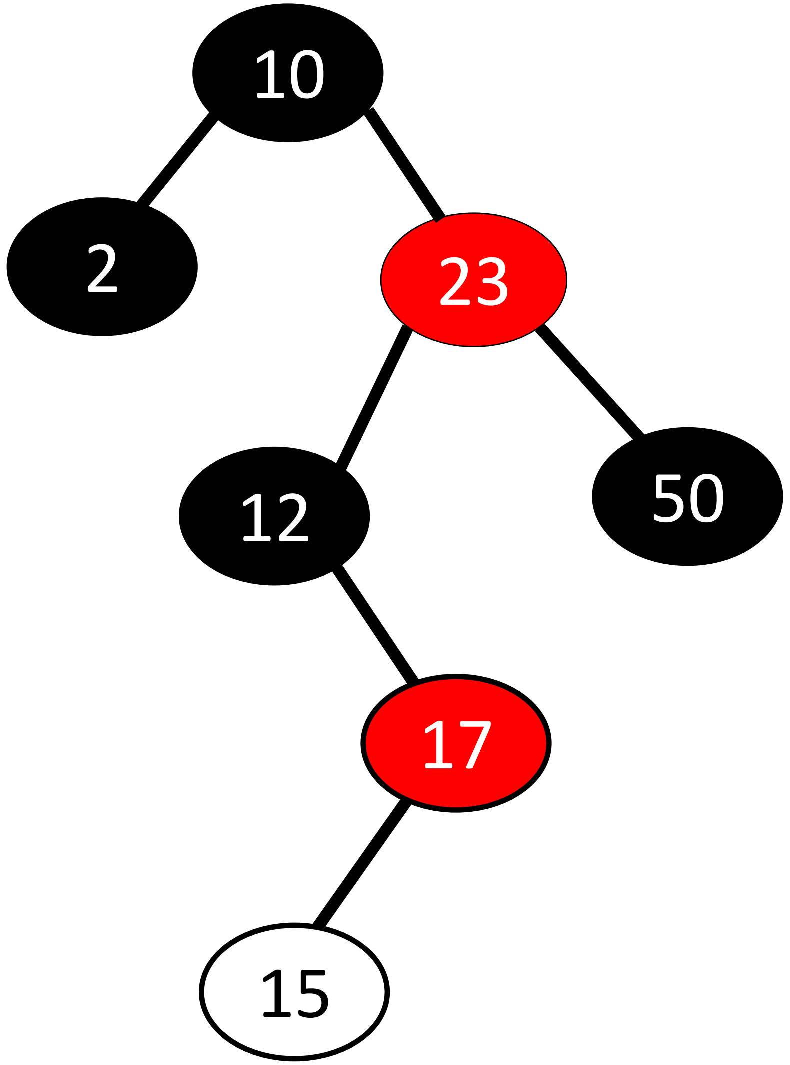
Step 1: Do the normal BST insertion



Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion

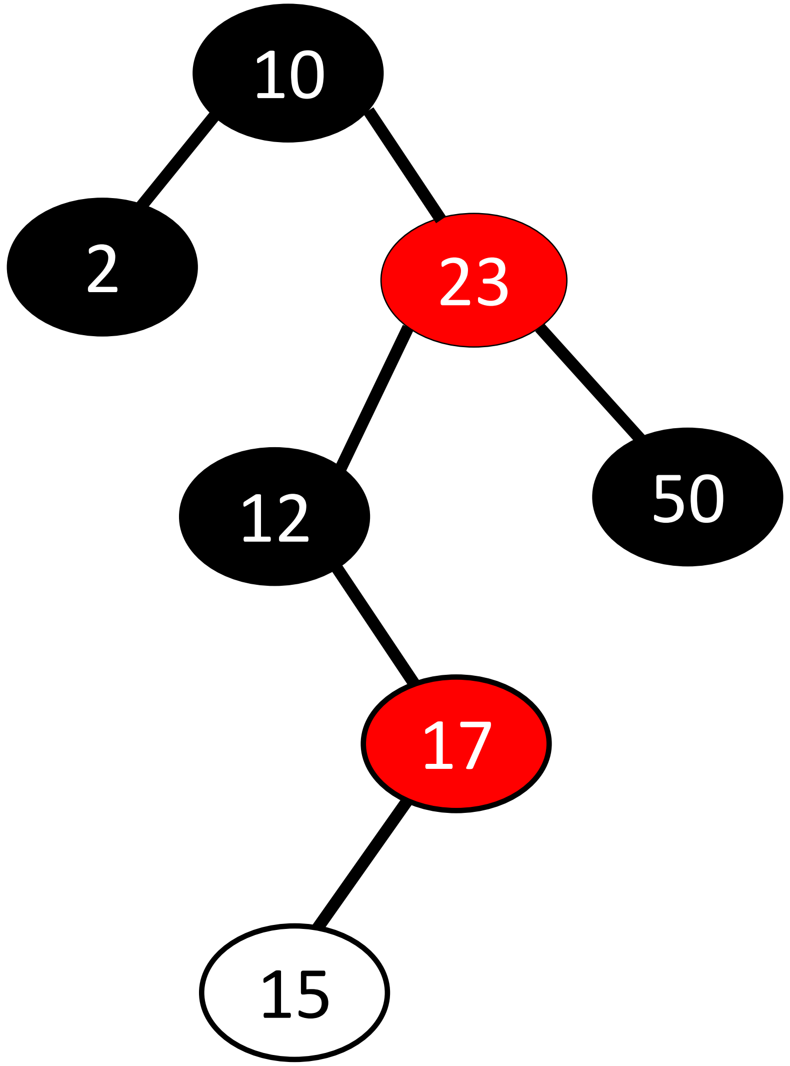


Our tree no longer has log(n) height, so we need to do some operations to reduce the height of the tree

Red-Black Tree Insertion/Deletion

`insert(15)`

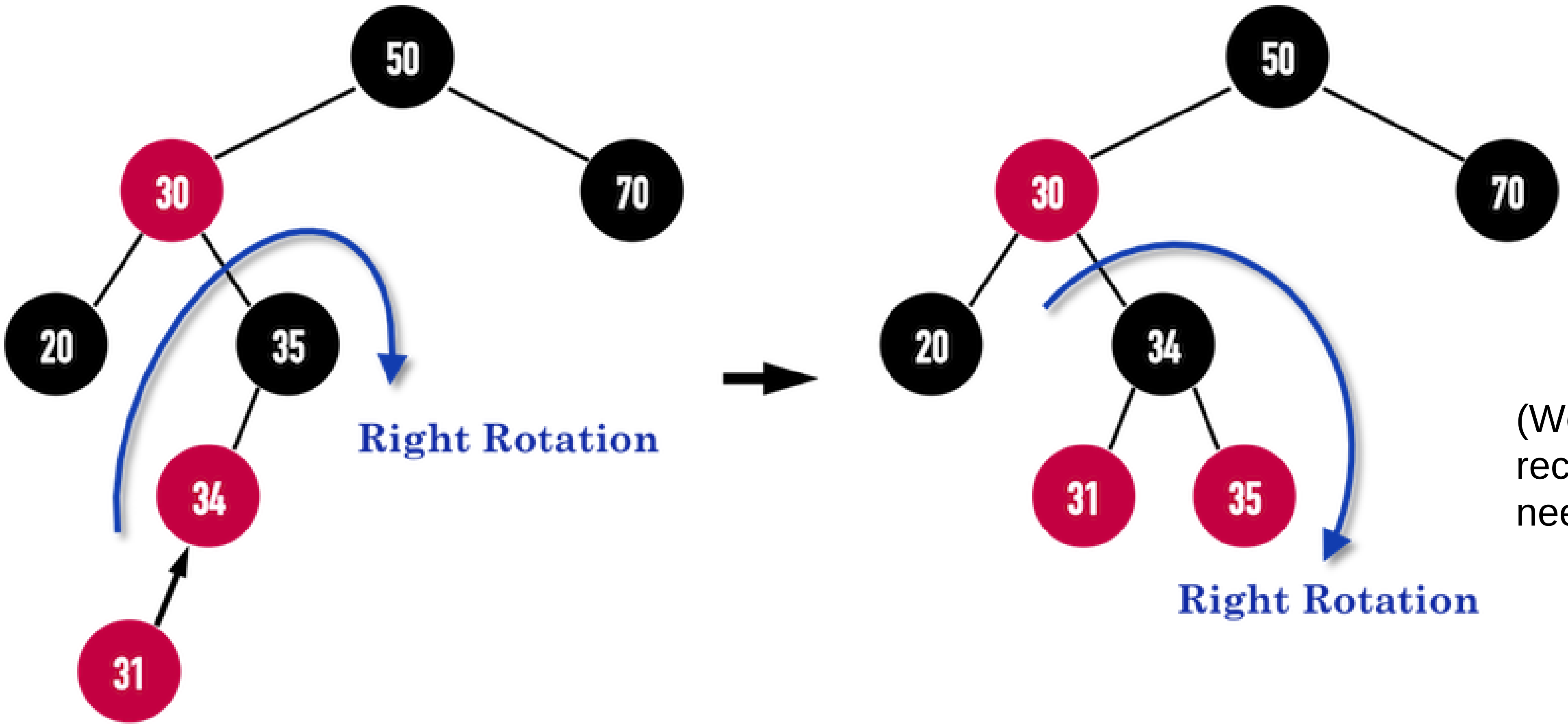
Step 1: Do the normal BST insertion



Our tree no longer has log(n) height, so we need to do some operations to reduce the height of the tree

These operations are known as **rotations**

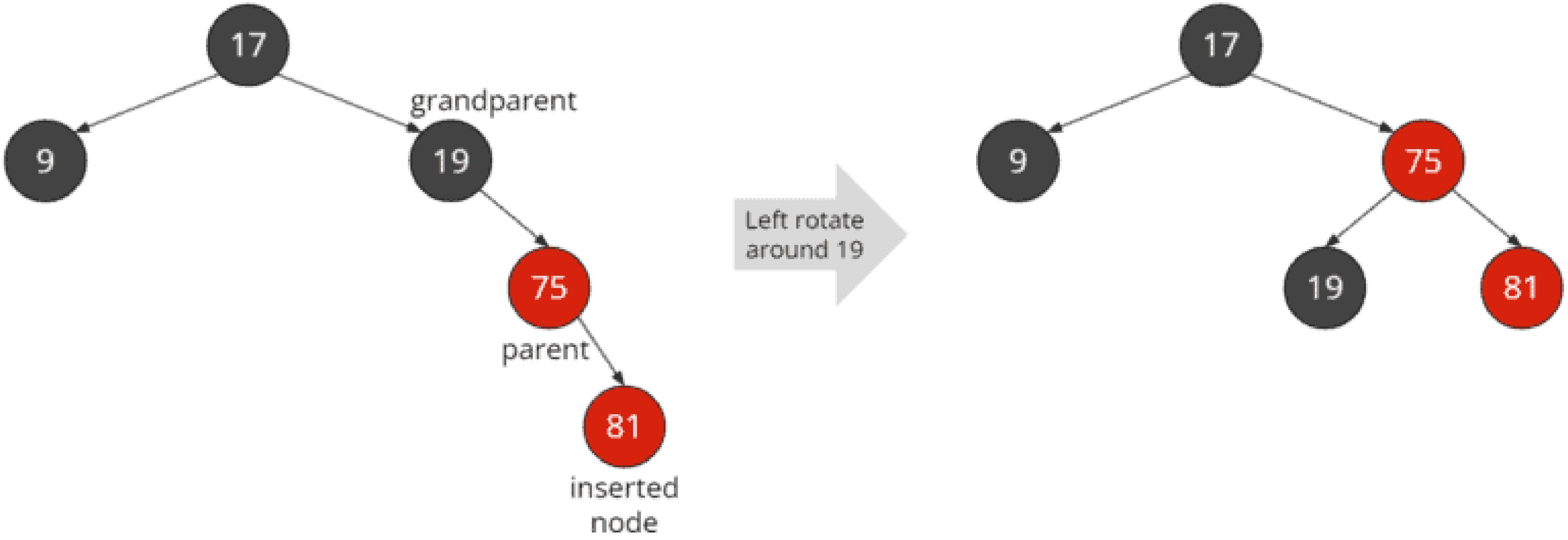
Red-Black Tree Rotation



(We also do some recoloring if needed!)

Local transformation (we rotate just a section– not the entire tree)

Red-Black Tree Rotation

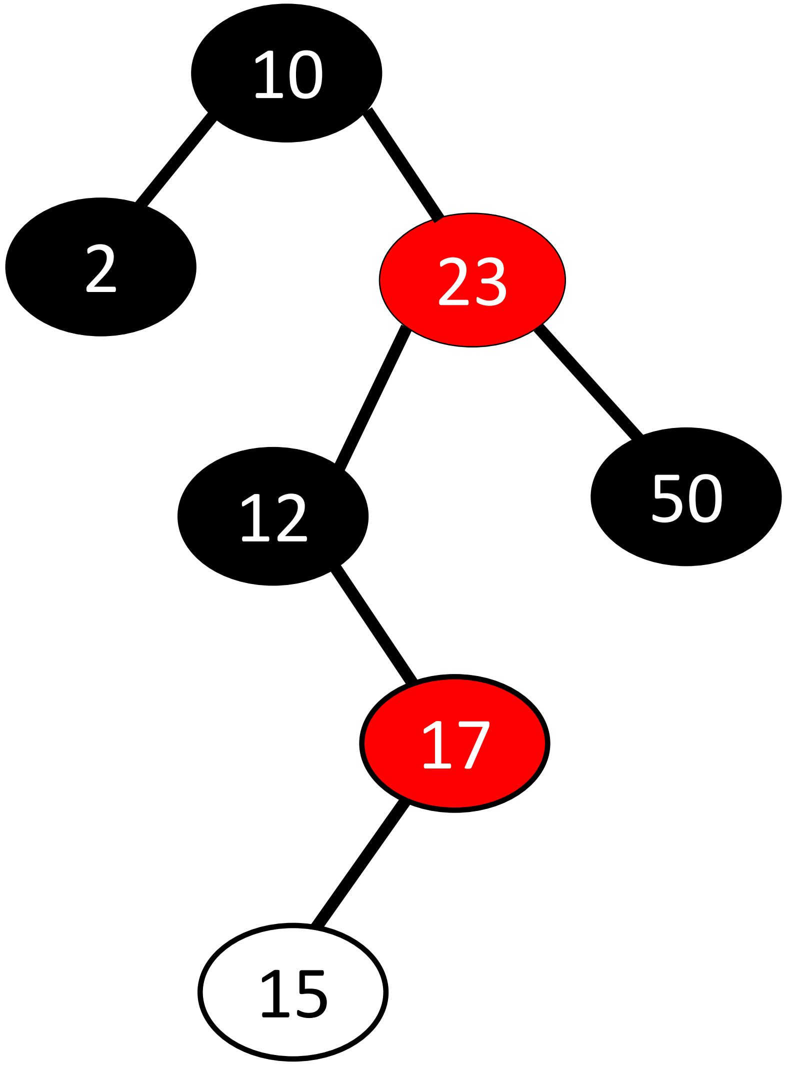


Local transformation (we rotate just a section– not the entire tree)

Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion



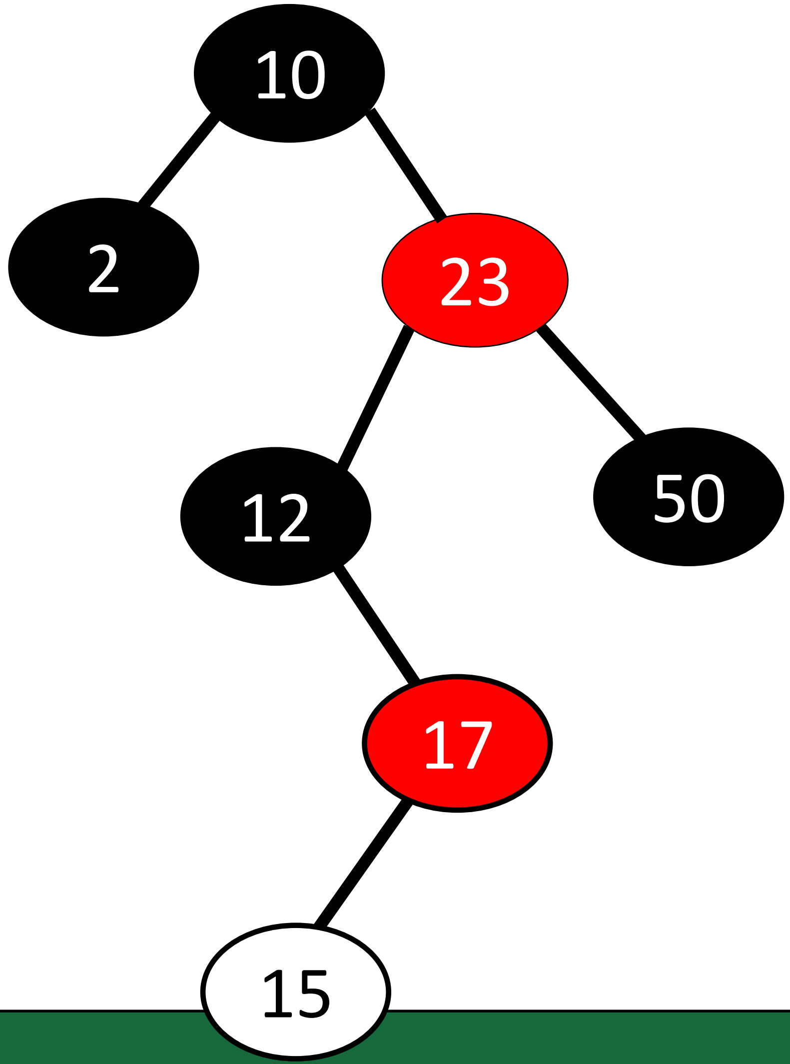
Our tree no longer has log(n) height, so we need to do some operations to reduce the height of the tree

These operations are known as **rotations**

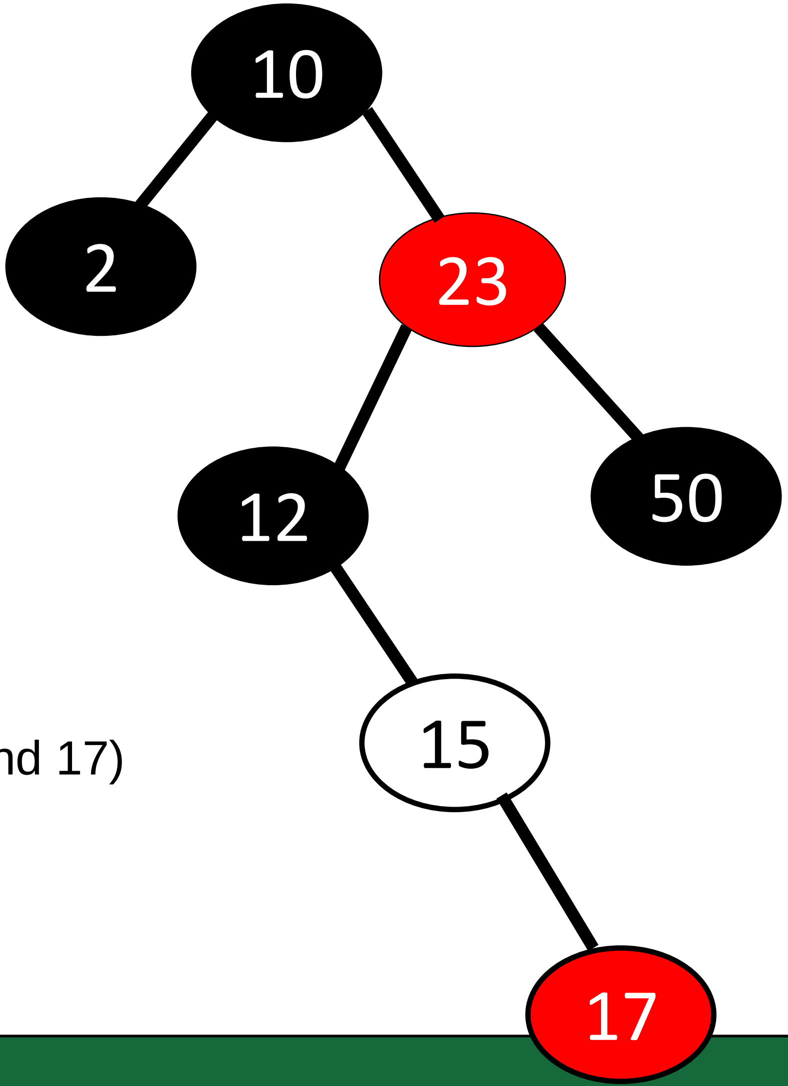
Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion
Step 2: Do rotation(s)



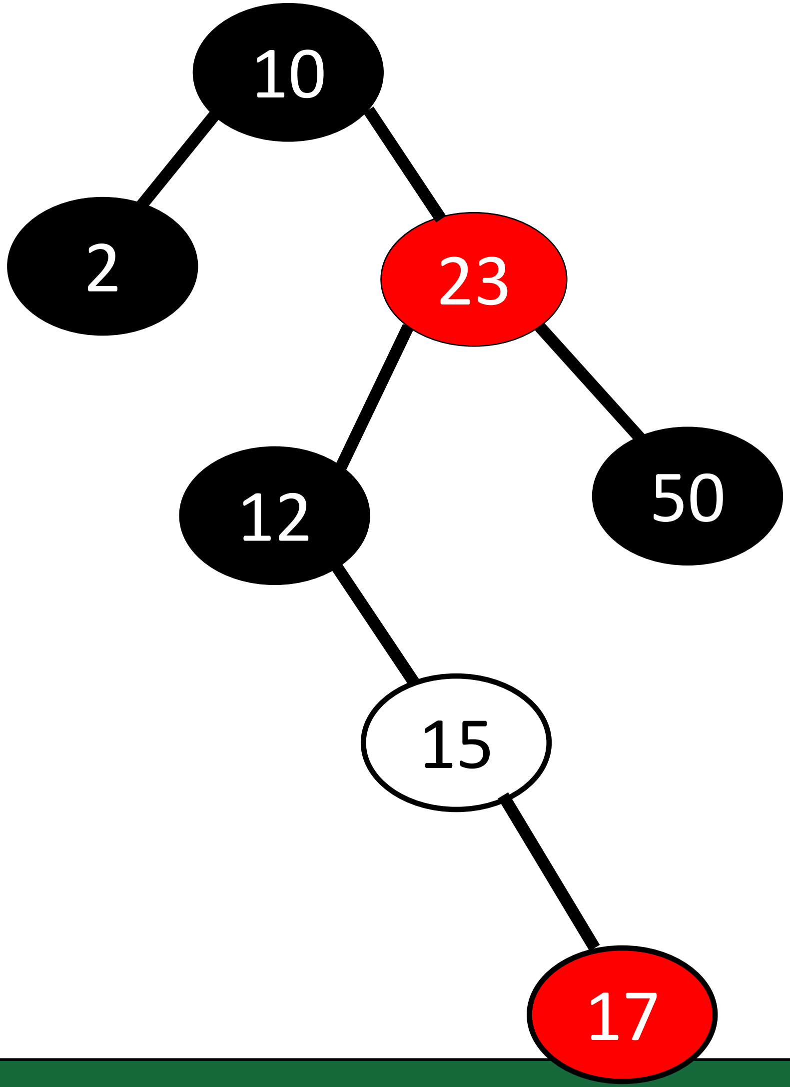
(Rotate Right around 17)



Red-Black Tree Insertion/Deletion

`insert(15)`

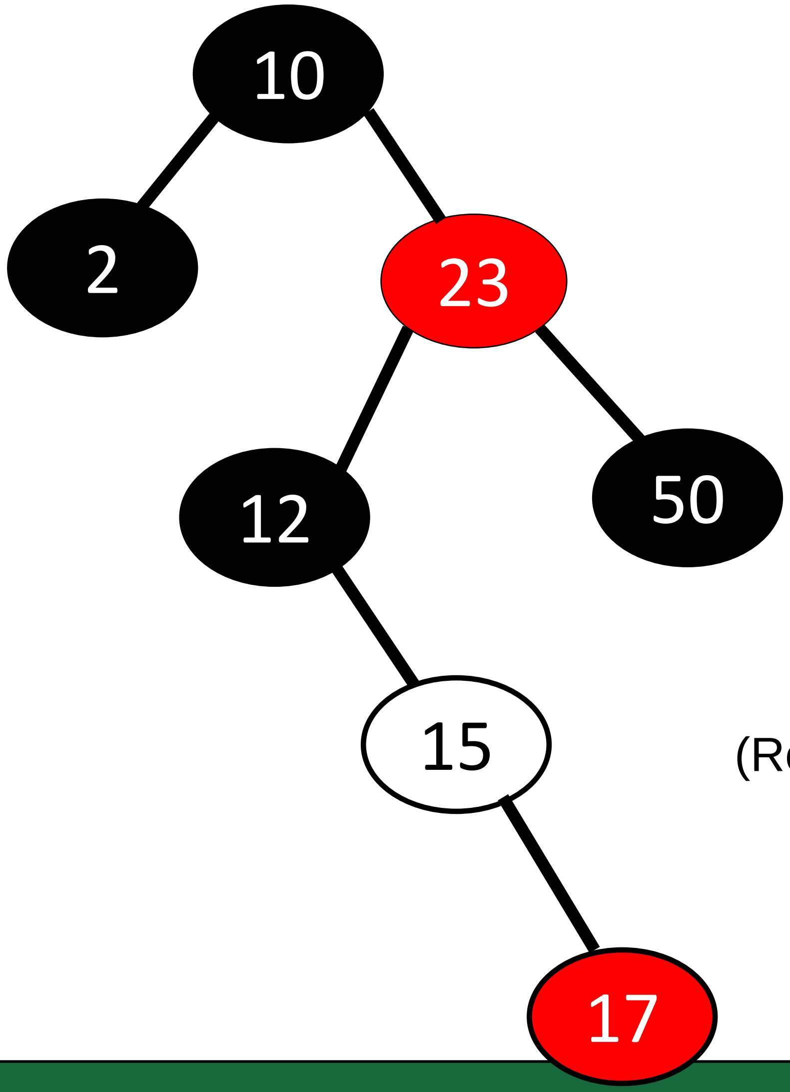
Step 1: Do the normal BST insertion
Step 2: Do rotation(s)



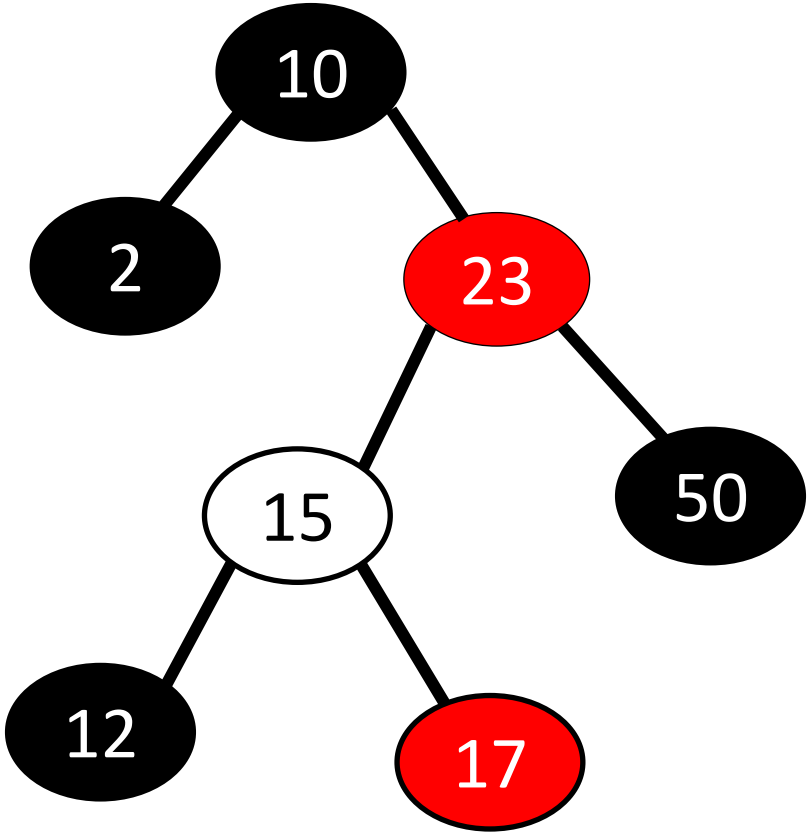
Red-Black Tree Insertion/Deletion

Step 1: Do the normal BST insertion
Step 2: Do rotation(s)

`insert(15)`



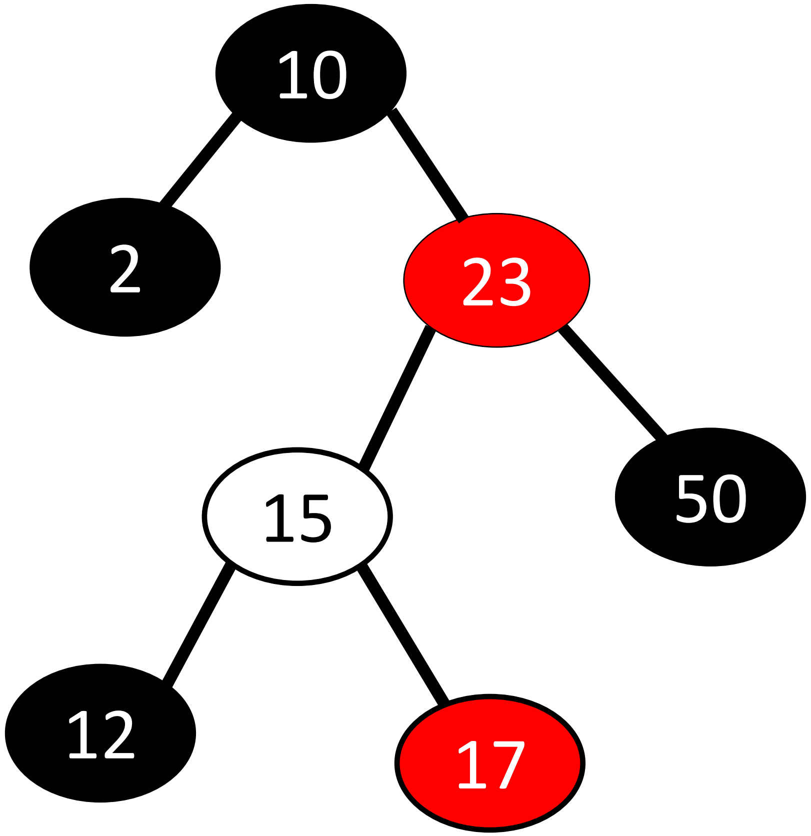
(Rotate left around 12)



Red-Black Tree Insertion/Deletion

`insert(15)`

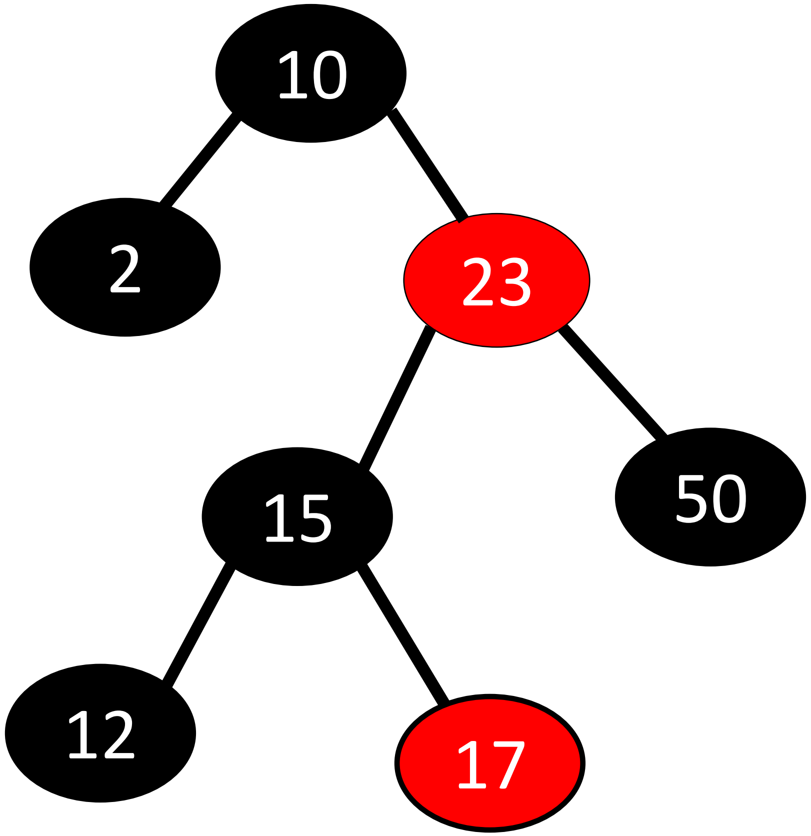
Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor



Red-Black Tree Insertion/Deletion

```
insert(15)
```

Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor

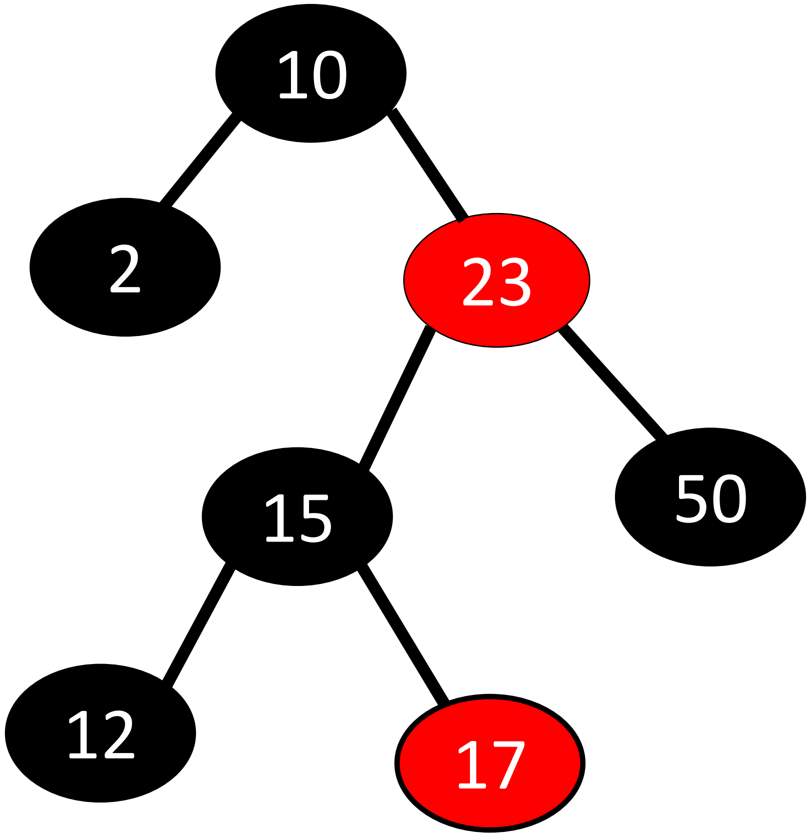


15 has to be black because….

Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor

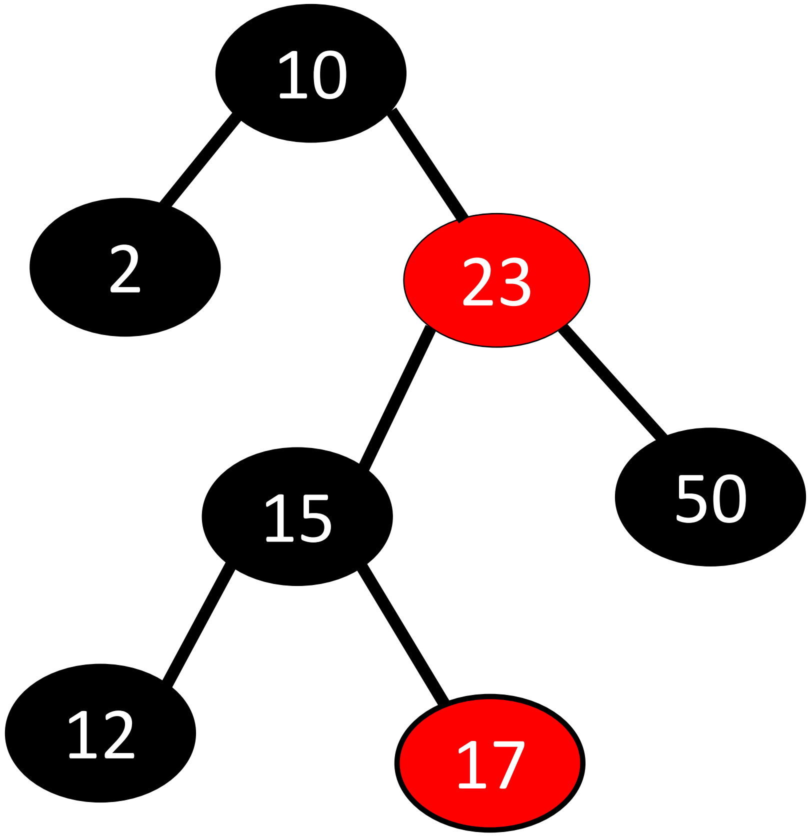


3. If a node is **red**, both children must be **black**

15 has to be black because 23 is red

Red-Black Tree Insertion/Deletion

`insert(15)`



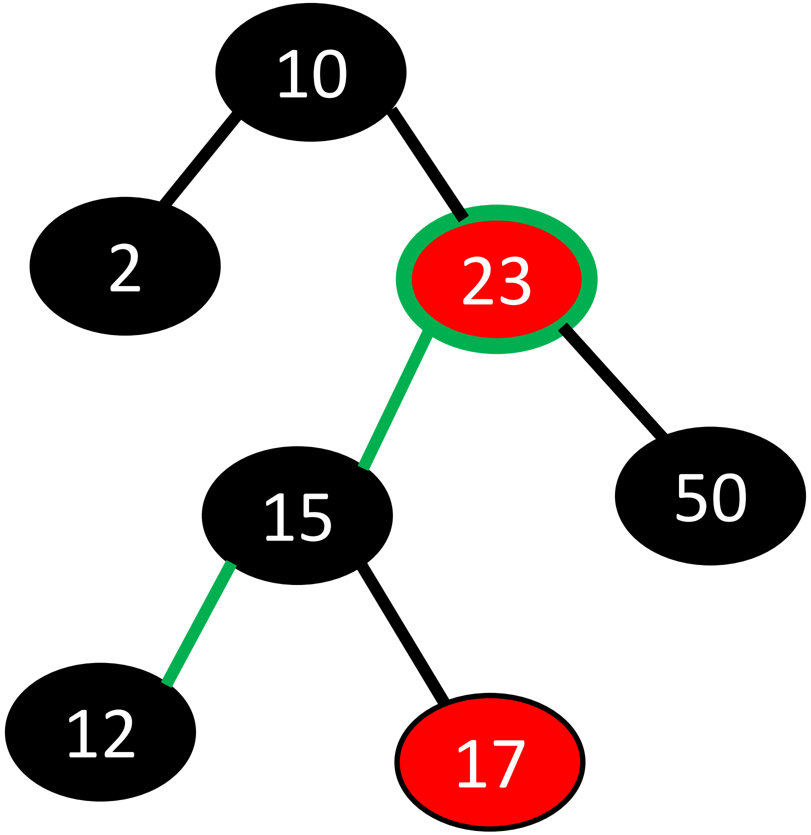
Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor

Is this a Red-Black tree?

Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor



Is this a Red-Black tree?

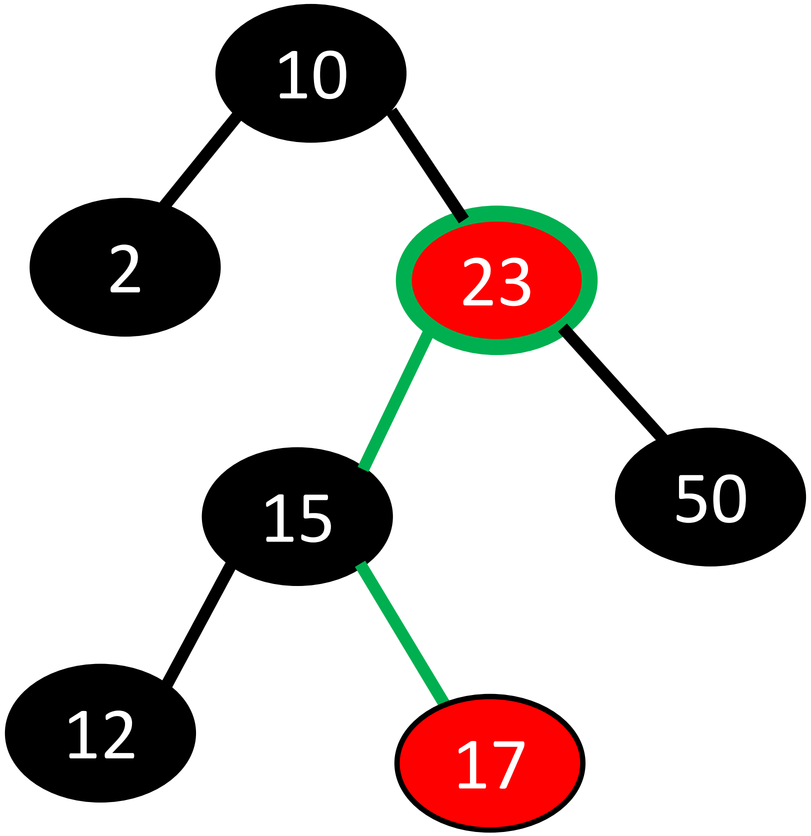
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

Path 1: 3 black nodes (including null nodes)

Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor



Is this a Red-Black tree?

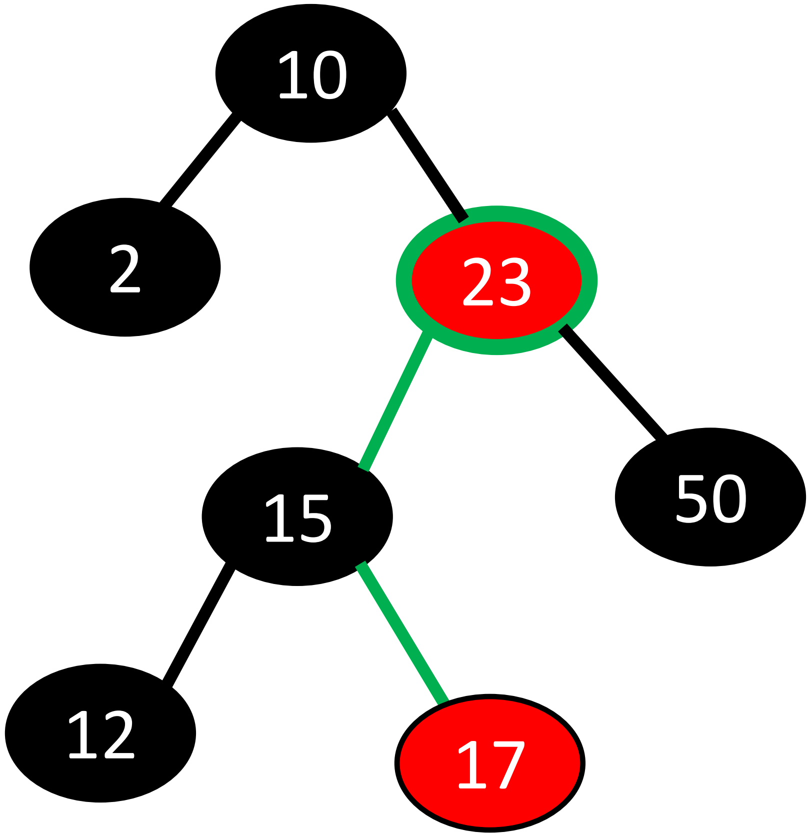
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

Path 1: 3 black nodes (including null nodes)
Path 2: 2 black nodes (including null nodes)

Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor



Is this a Red-Black tree?

5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

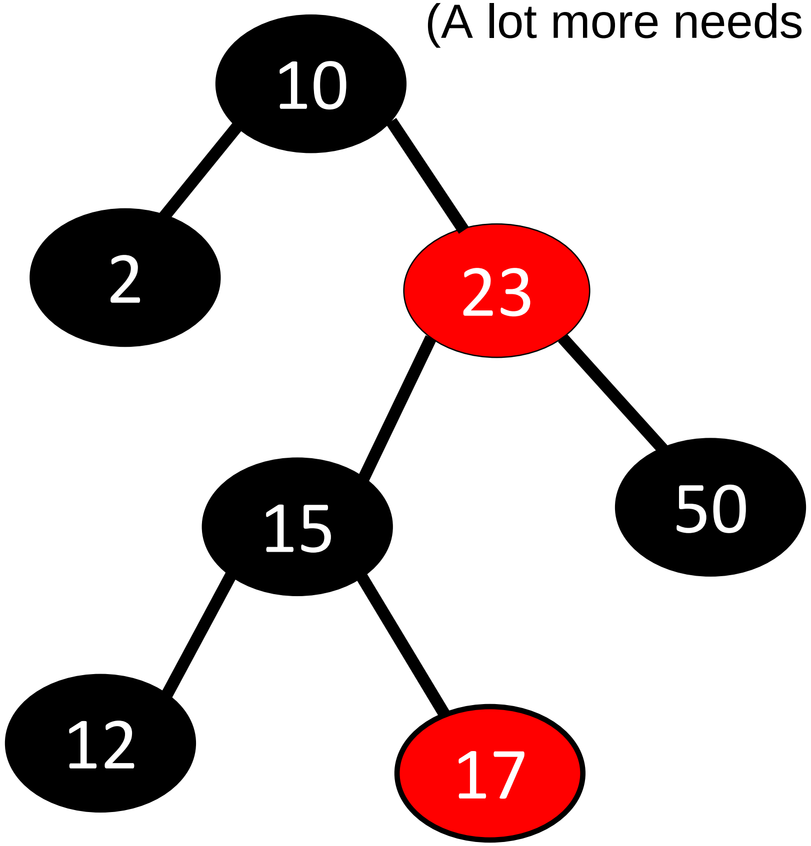
Path 1: 3 black nodes (including null nodes)
Path 2: 2 black nodes (including null nodes)

Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor

(A lot more needs to be done here)

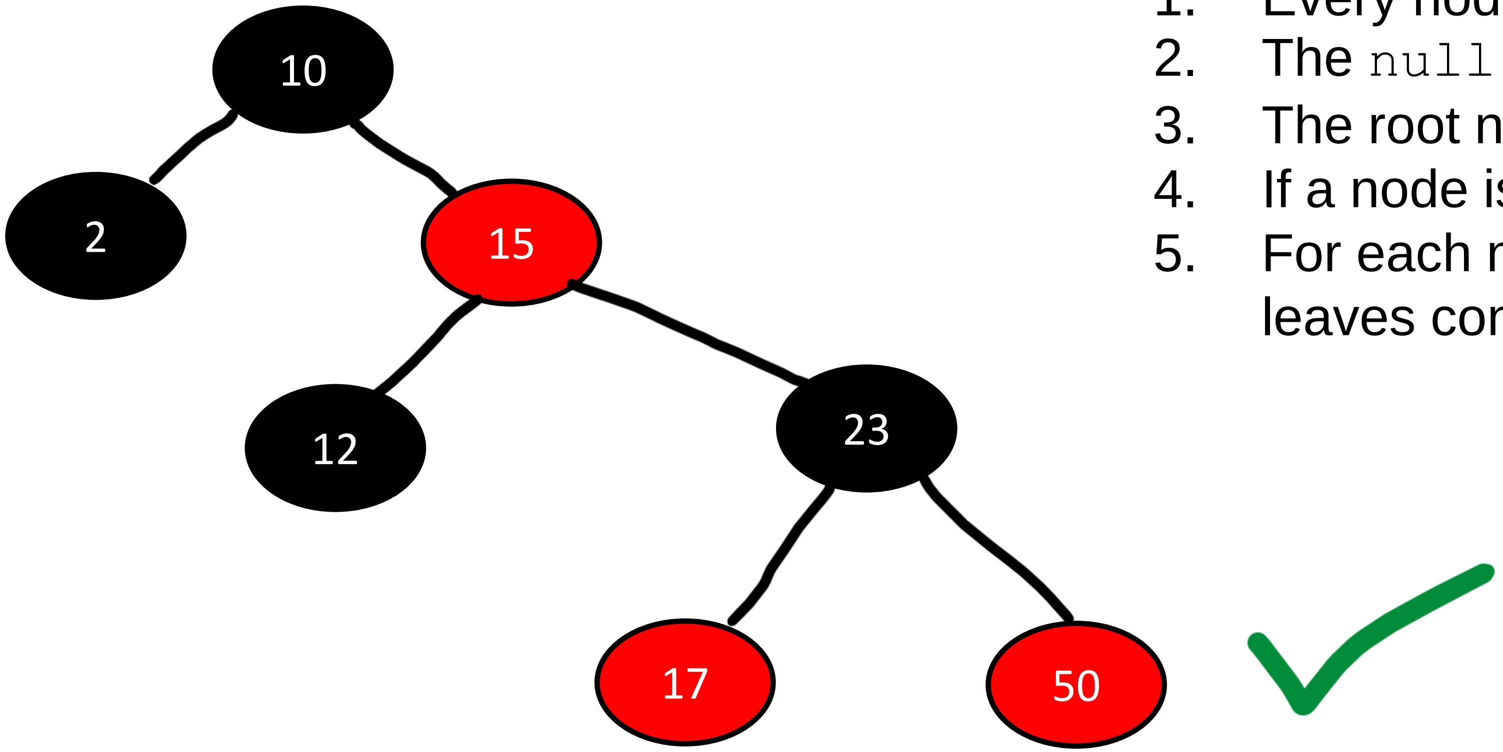


1. Every node is either **red** or **black**
2. The `null` children are **black**
3. The root node is **black**
4. If a node is **red**, both children must be **black**
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

Red-Black Tree Insertion/Deletion

`insert(15)`

Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor

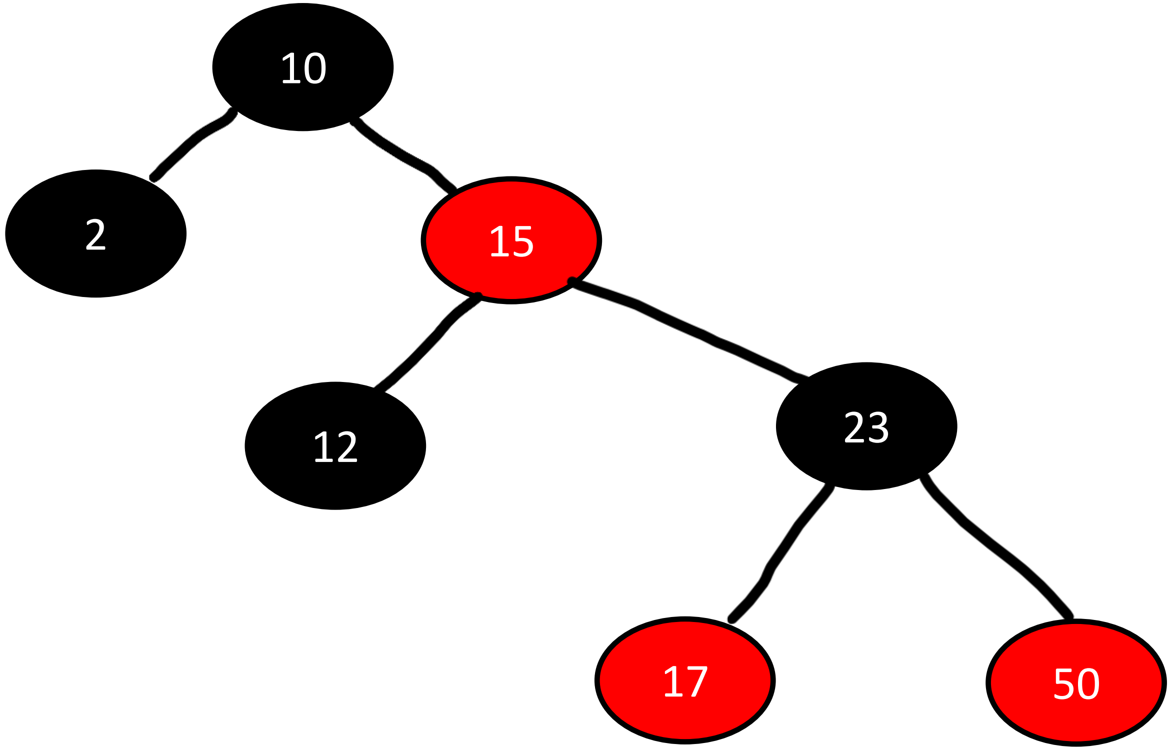


1. Every node is either **red** or **black**
2. The `null` children are **black**
3. The root node is **black**
4. If a node is **red**, both children must be **black**
5. For each node, all paths from the node to descendant leaves contain the same number of **black** nodes

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

Red-Black Tree Insertion/Deletion

`insert(15)`



Step 1: Do the normal BST insertion
Step 2: Do rotation(s)
Step 3: Recolor

Fact:

There will at most 3 rotations needed, and each rotation happens in O(1) time

So, maintaining a Red/Black try happens in O(1) time

Red-Black Tree Insertion/Deletion

`delete(15)`

(Deleting is not as scary, because deleting a node will never increase the height of the tree)

Step 1: Do the normal BST deletion

- Case 1: no children
- Case 2: 1 child
- Case 3: 2 children

Step 2: Do rotation(s) (optional?)

Step 3: Recolor

Fact:

There will at most 3 rotations needed, and each rotation happens in O(1) time

So, maintaining a Red/Black try happens in O(1) time

Takeaways

We can add a color (red or black) instance field to our nodes to create a Red Black Tree

If we follow the rules of a Red Black Tree, and follow the proper rotations/recoloring steps, we can guarantee that our tree will be balanced

Guaranteed Balanced BST =

- ❑ O(logn) insertion
- ❑ O(logn) deletion
- ❑ O(logn) Searching/Contains

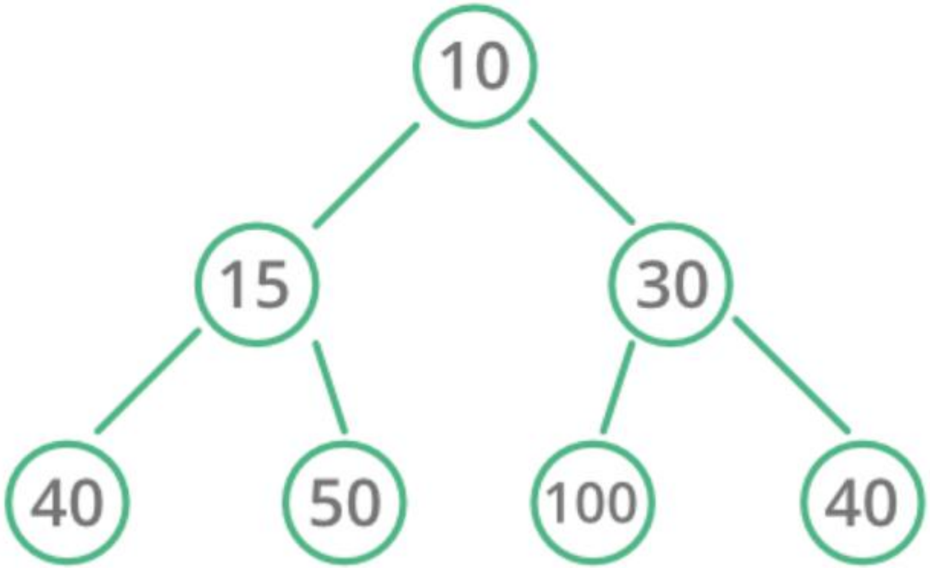
Adding Red/Black functionality to a BST does not affect the running time

There are also BSTs called **AVL tree** and **2-3 trees** that serve the same purpose of RB trees

You will never have to write code for a red black tree, but you should know the purpose of red black trees, and be able to verify if a red black tree is valid or not

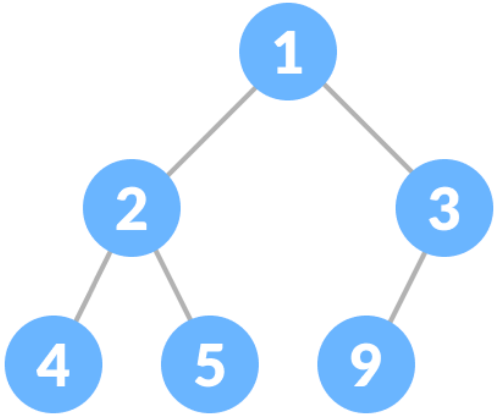
Binary Search Tree Java Library?

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

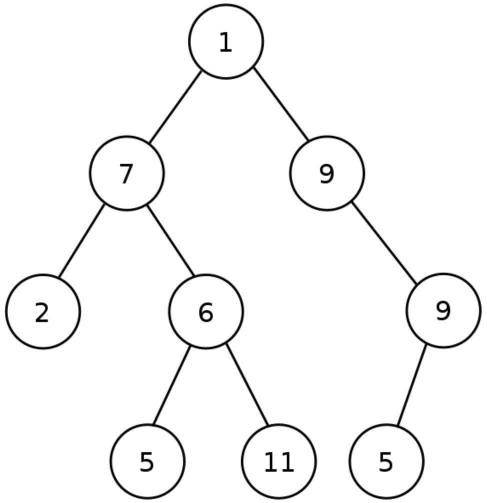


The **Heap** data structure is **complete** binary tree that follows the heap property

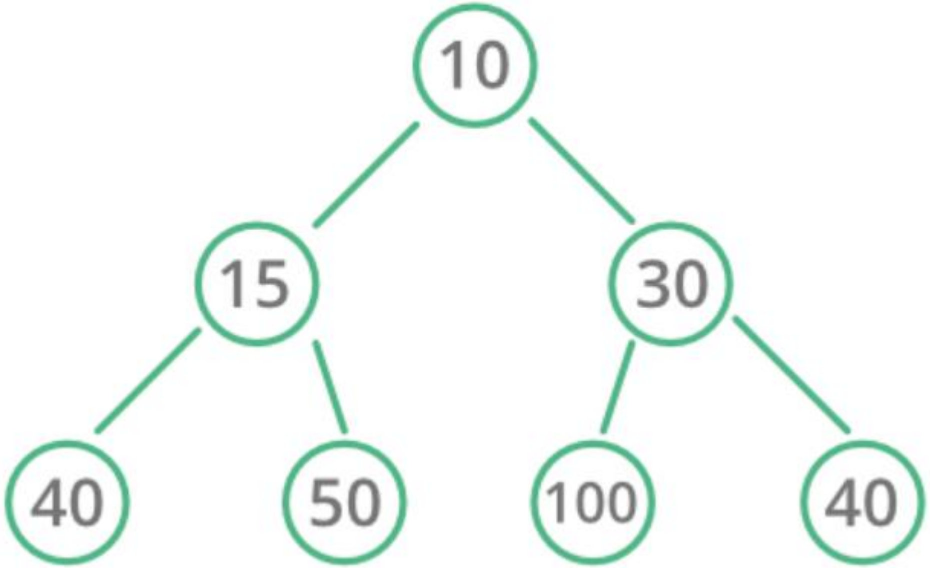
Complete tree - Every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible



complete



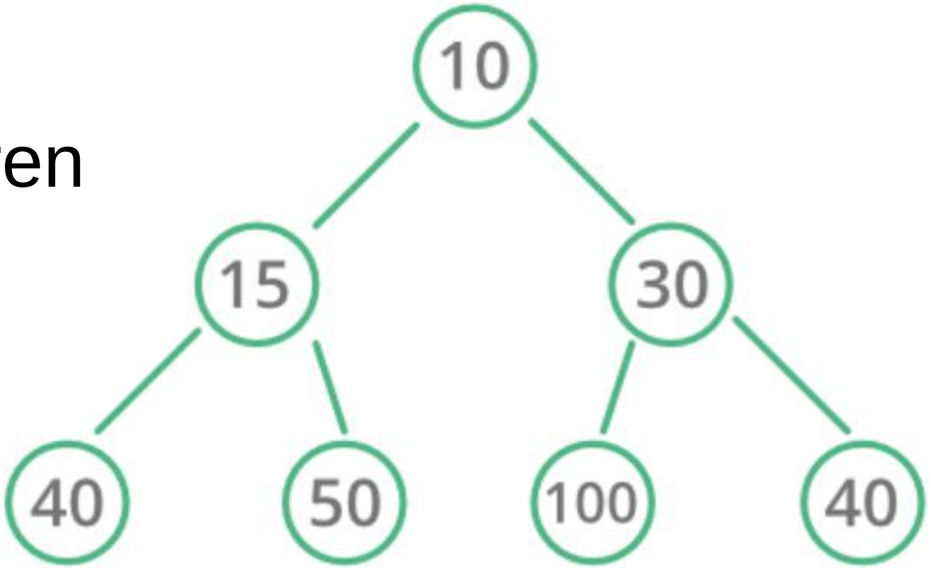
Not complete



complete

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

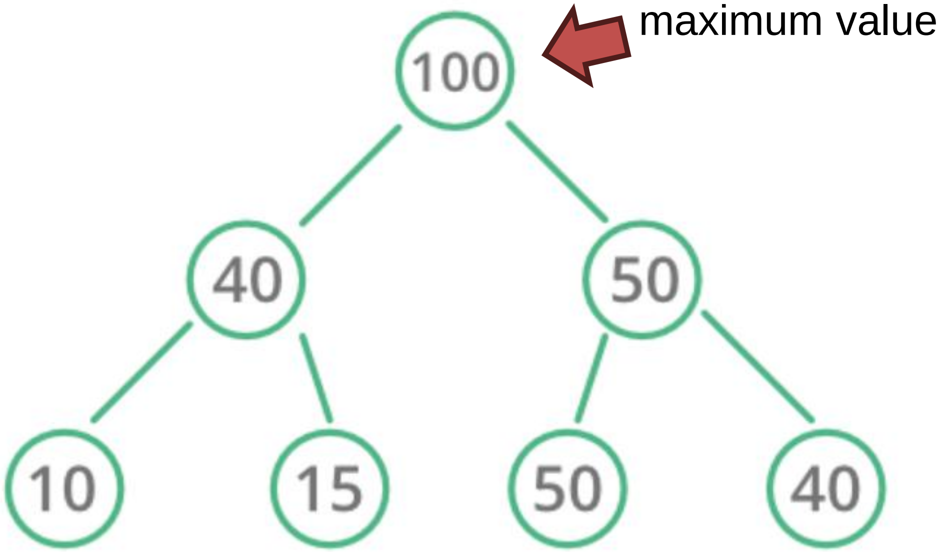
Binary – cannot have more than two children



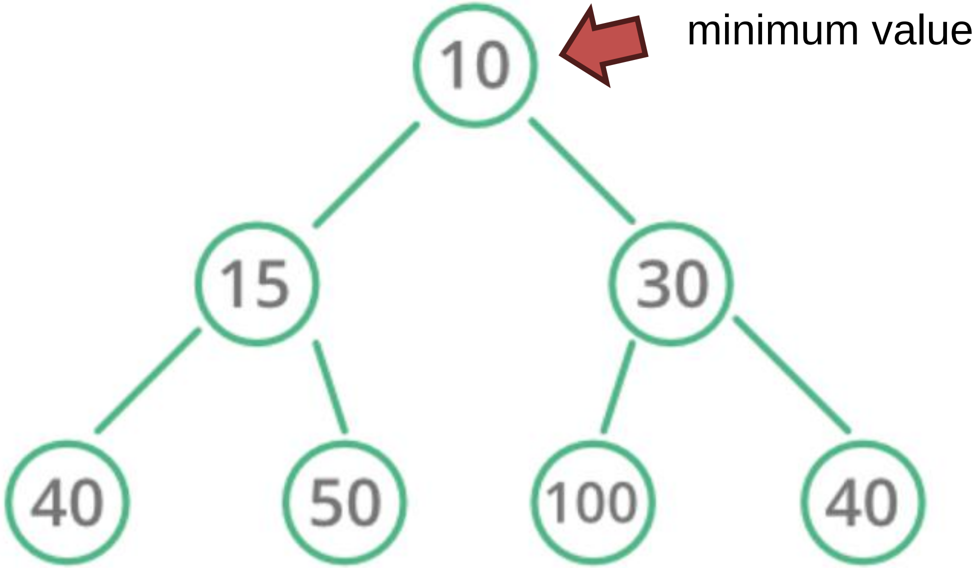
The **Heap** data structure is complete binary tree that follows the **heap property**

Two types of heaps

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

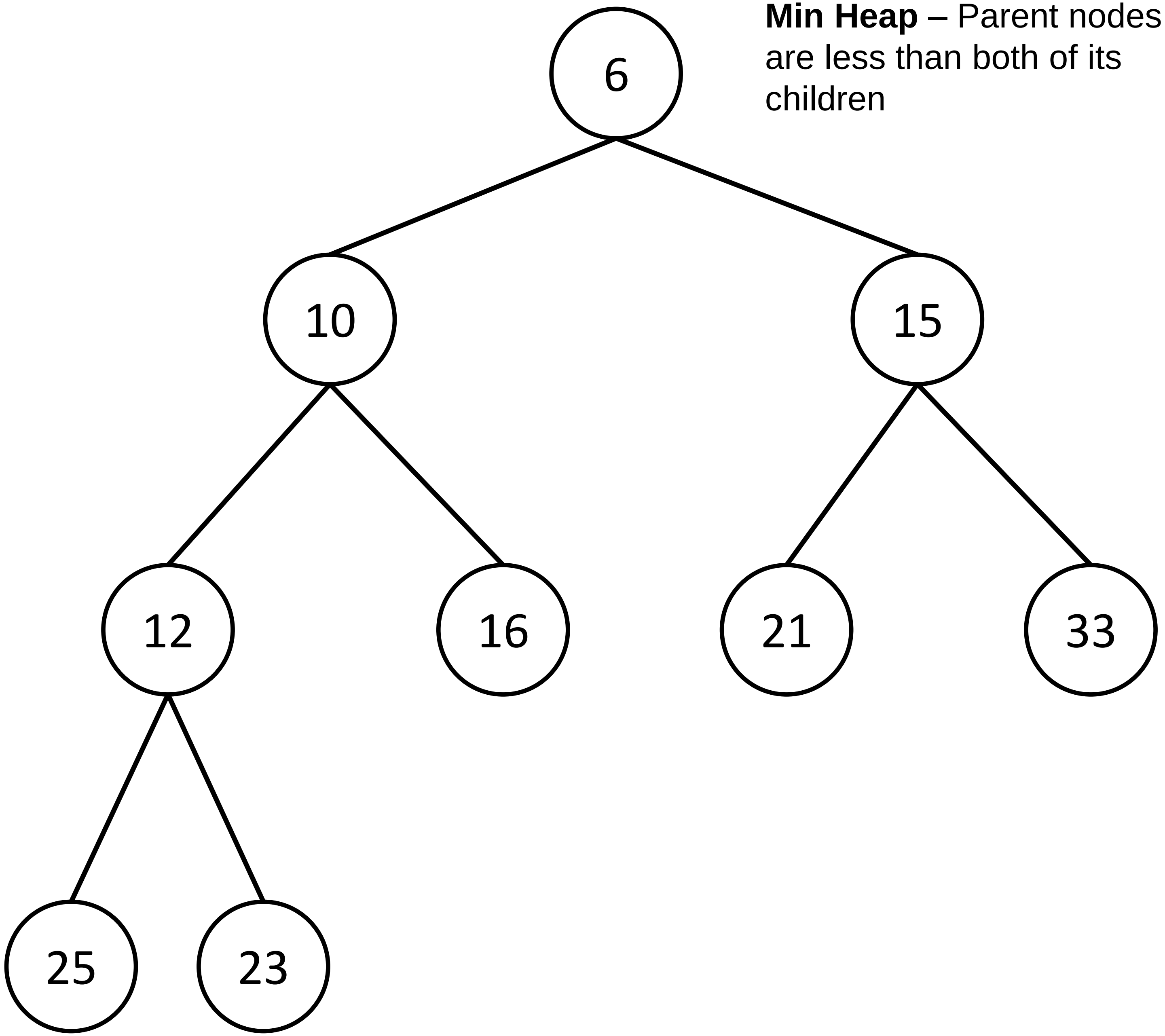


Min Heap – Parent nodes are less than both of its children



Heap Operations - Insert

```
add(7);
```



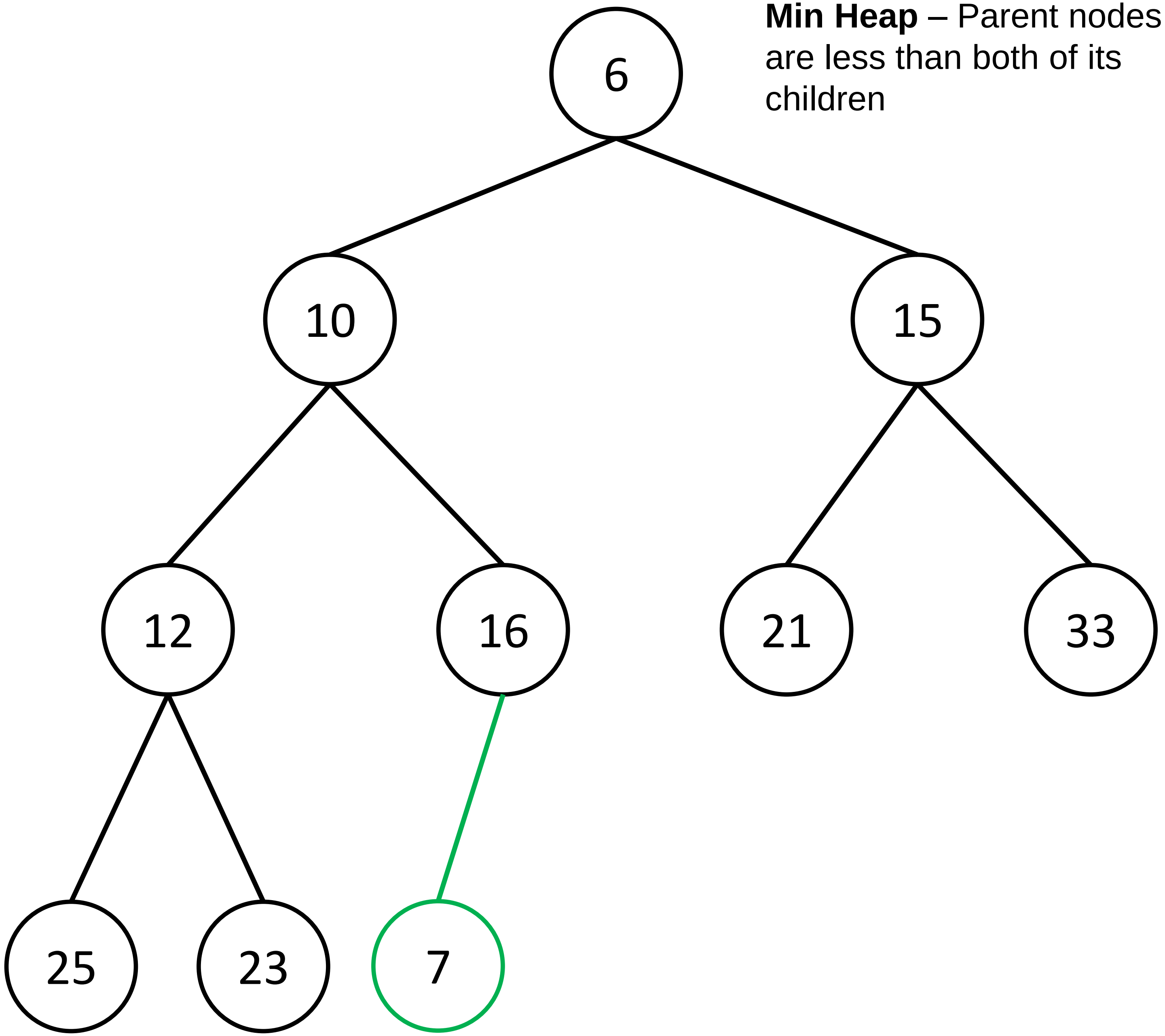
Min Heap – Parent nodes are less than both of its children

Heap Operations - Insert

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

```
add(7);
```

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

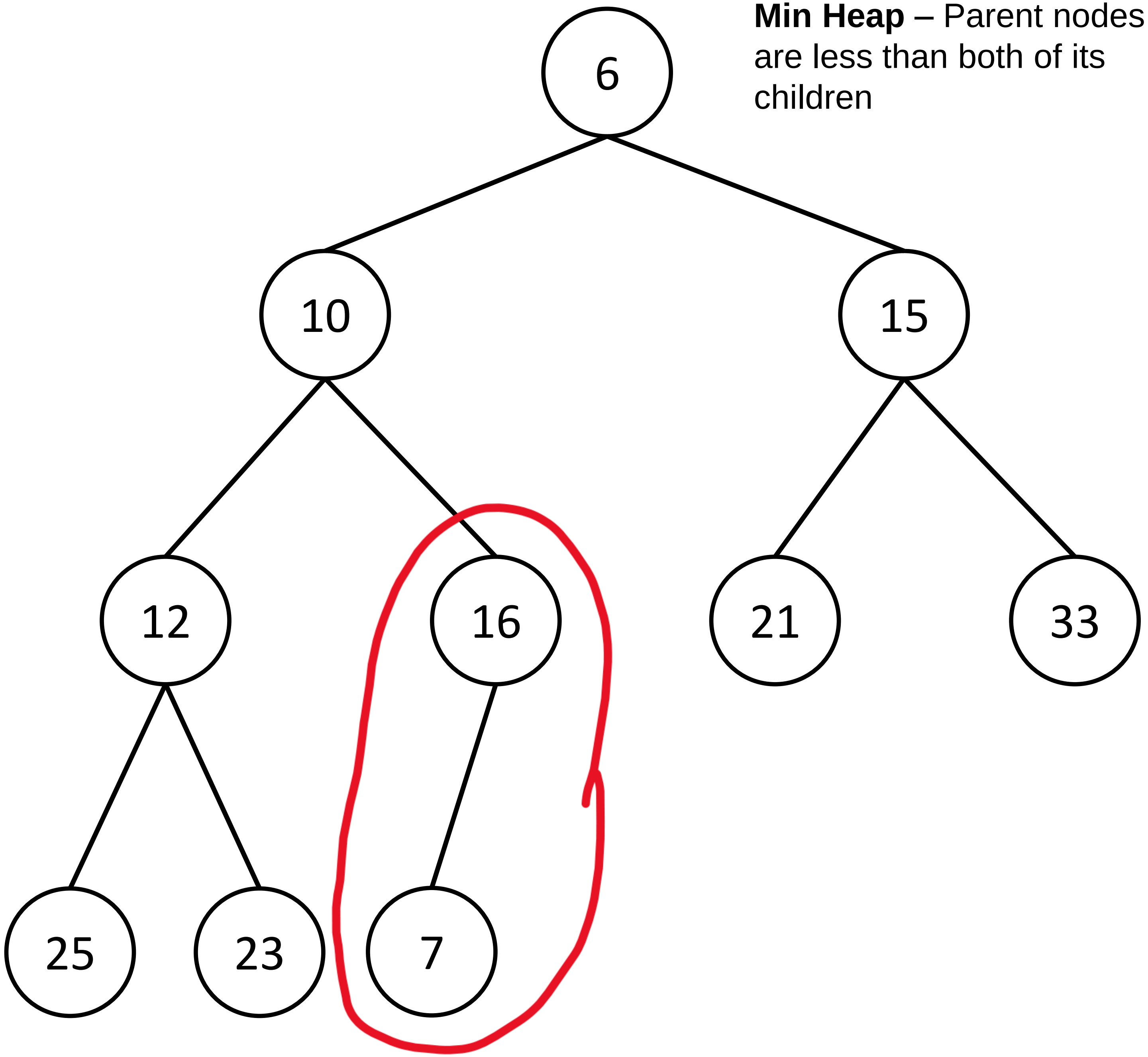


Heap Operations - Insert

```
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

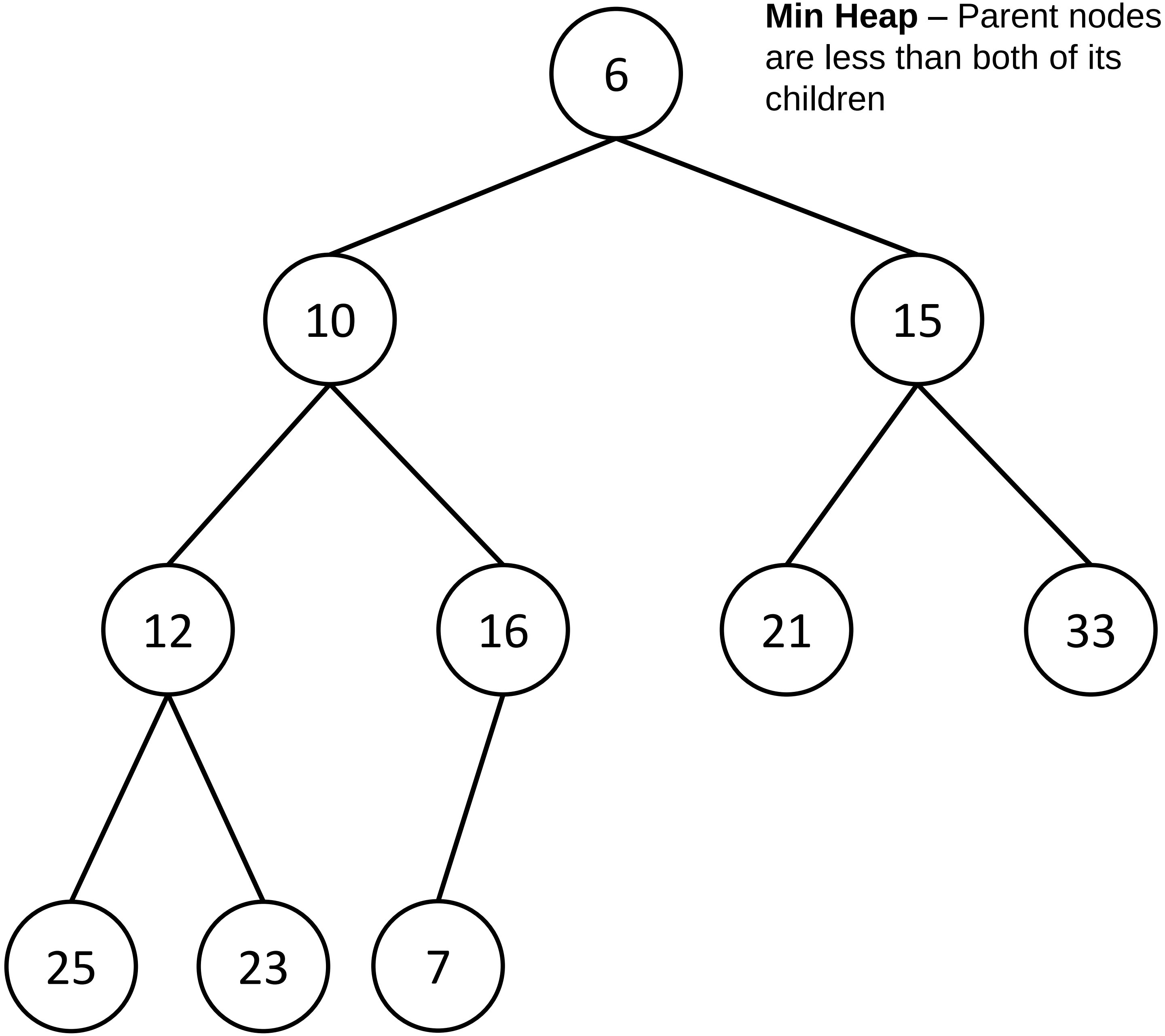


Min Heap – Parent nodes are less than both of its children

Heap Operations - Insert

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

```
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

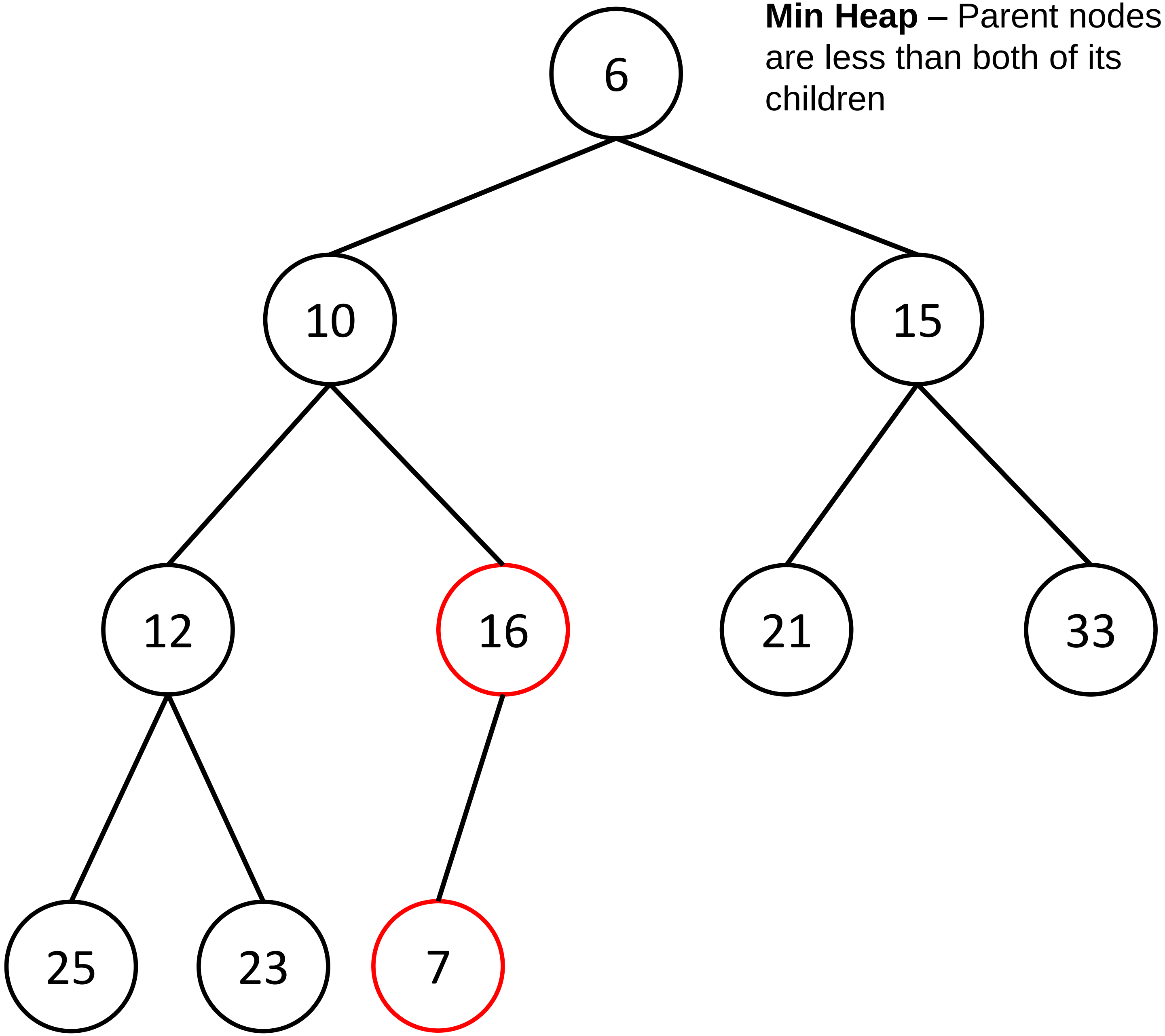
Heap Operations - Insert

```
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

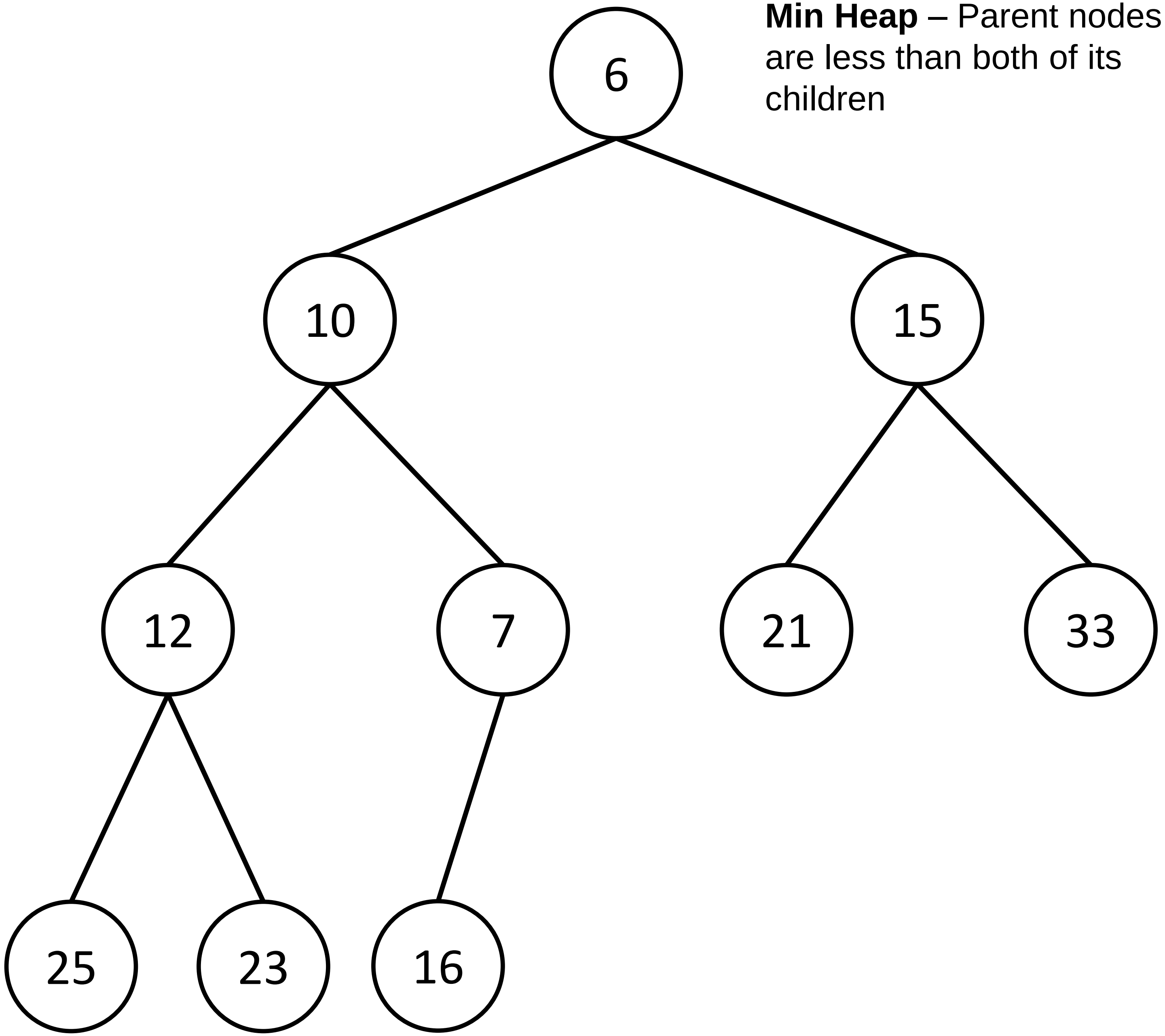
Heap Operations - Insert

```
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

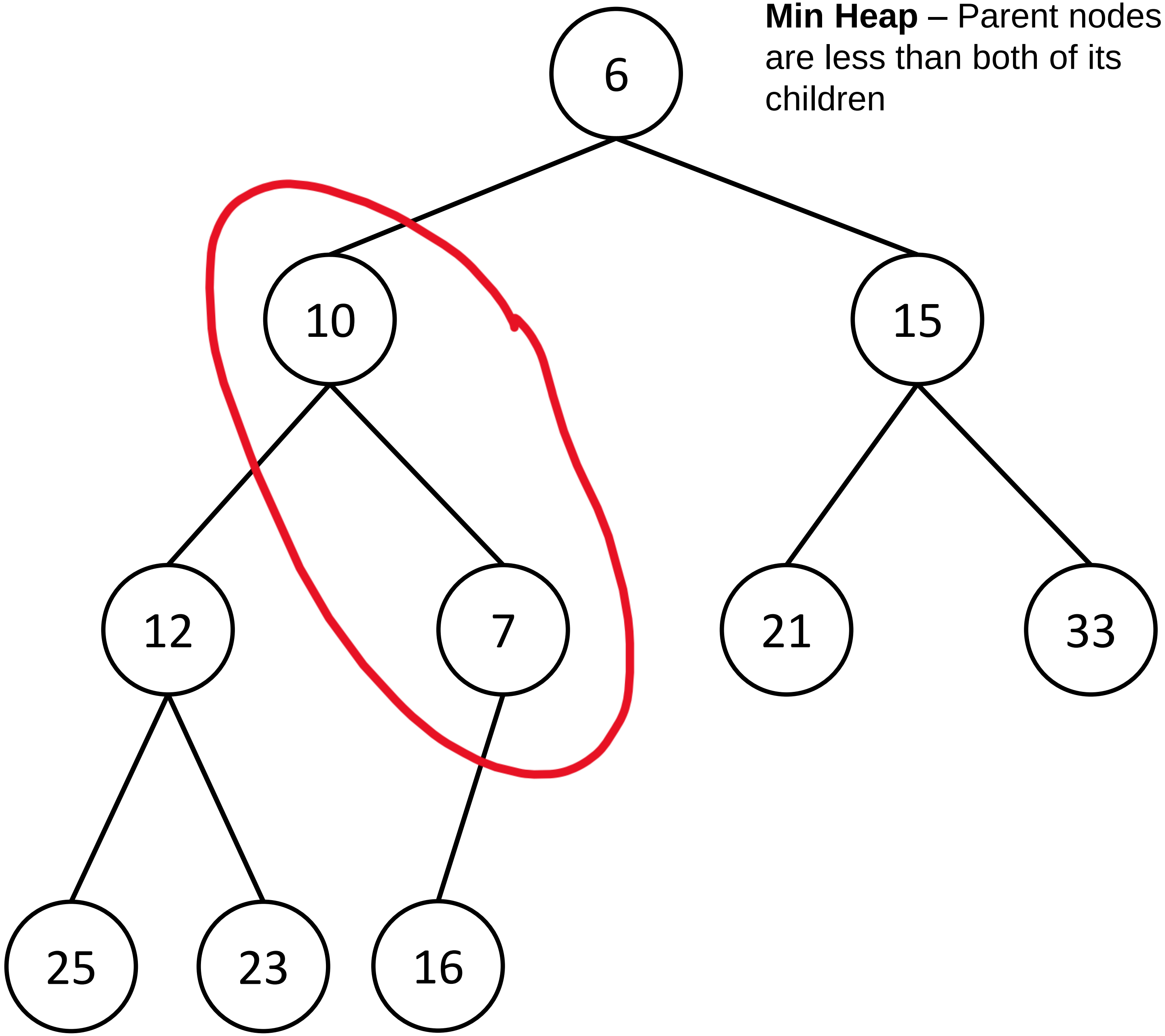


Min Heap – Parent nodes are less than both of its children

Heap Operations - Insert

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

```
add(7);
```



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

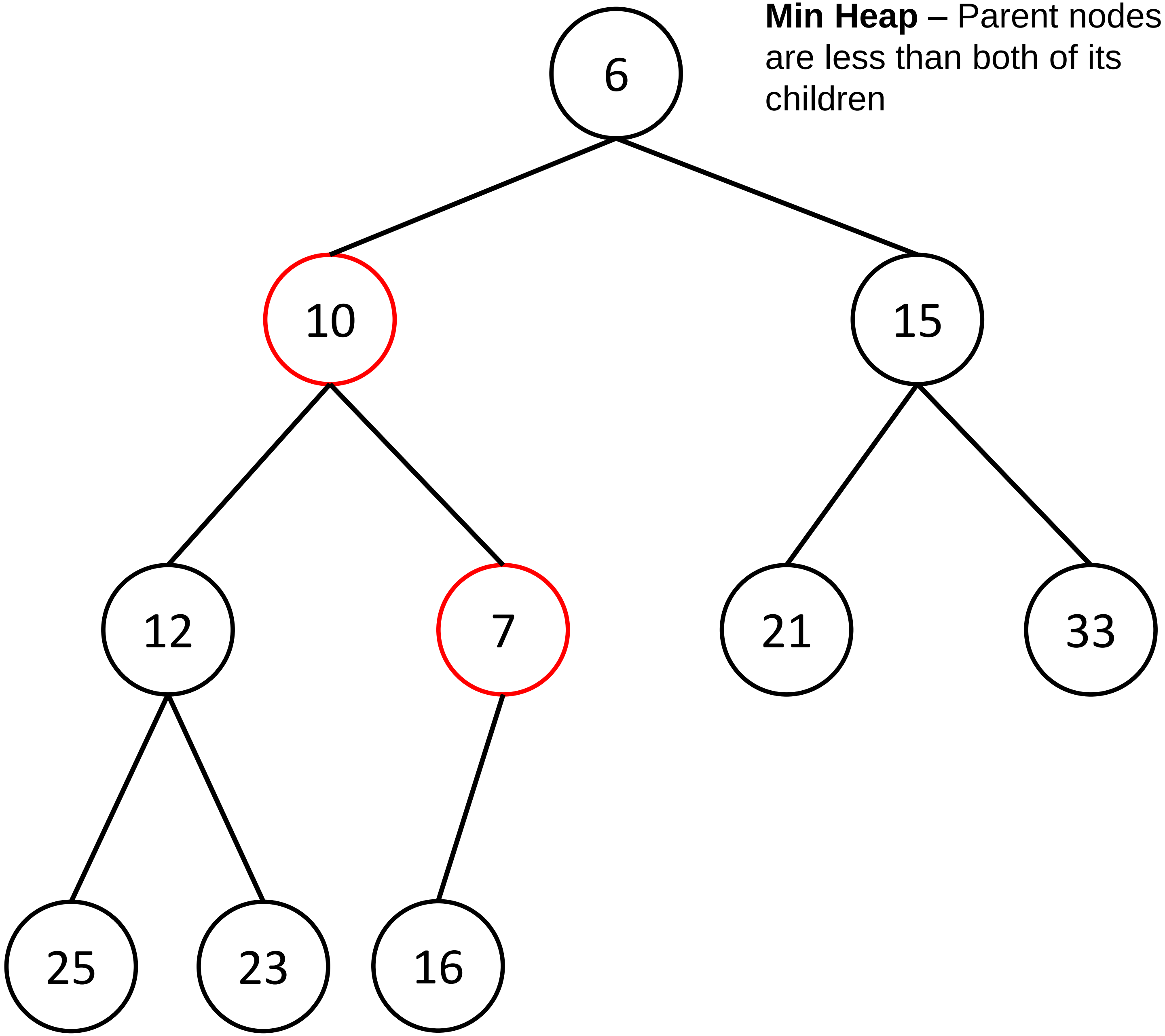
However, we are now violating the heap property

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

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

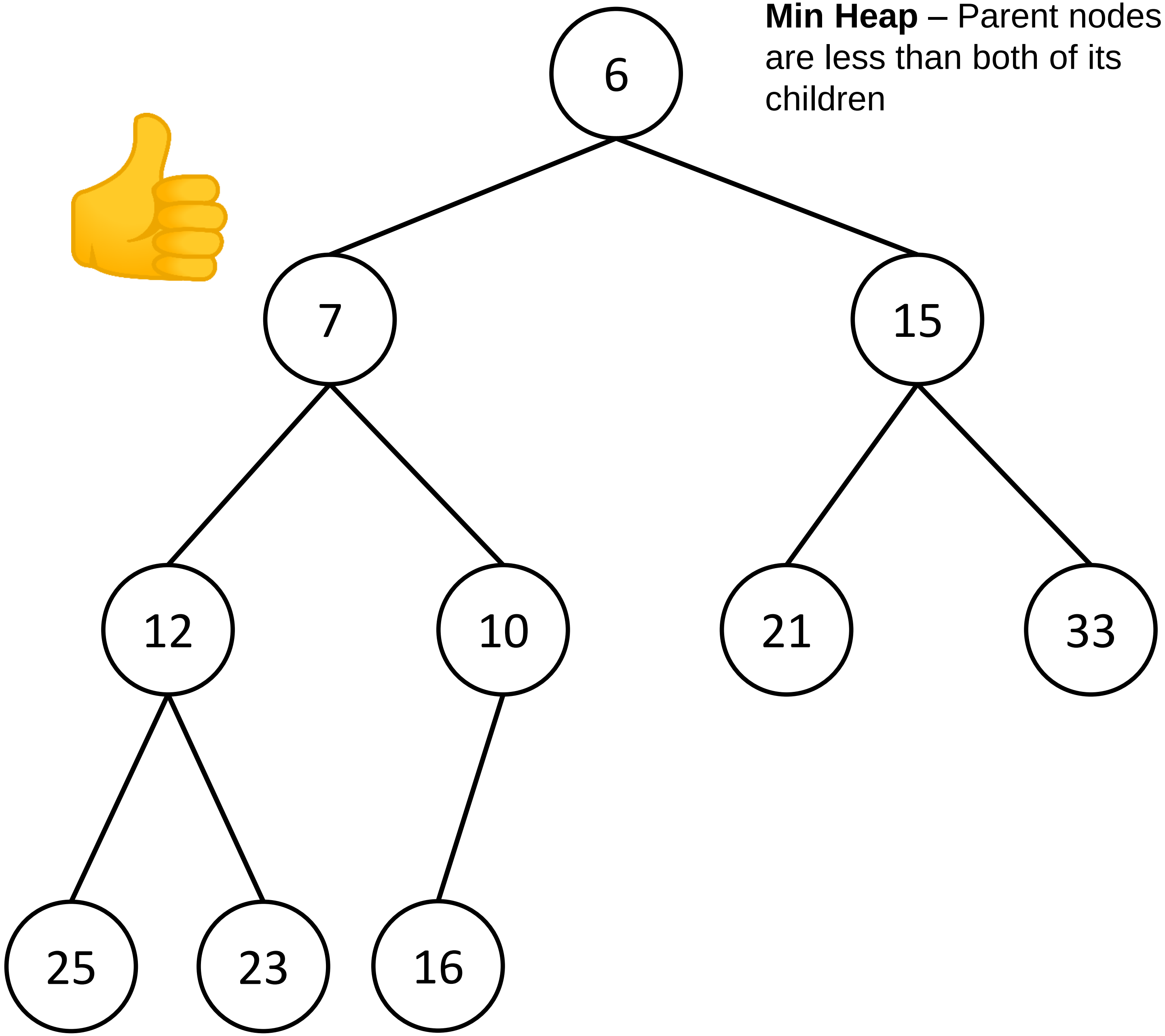
However, we are now violating the heap property

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

Heap Operations - Insert

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

```
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

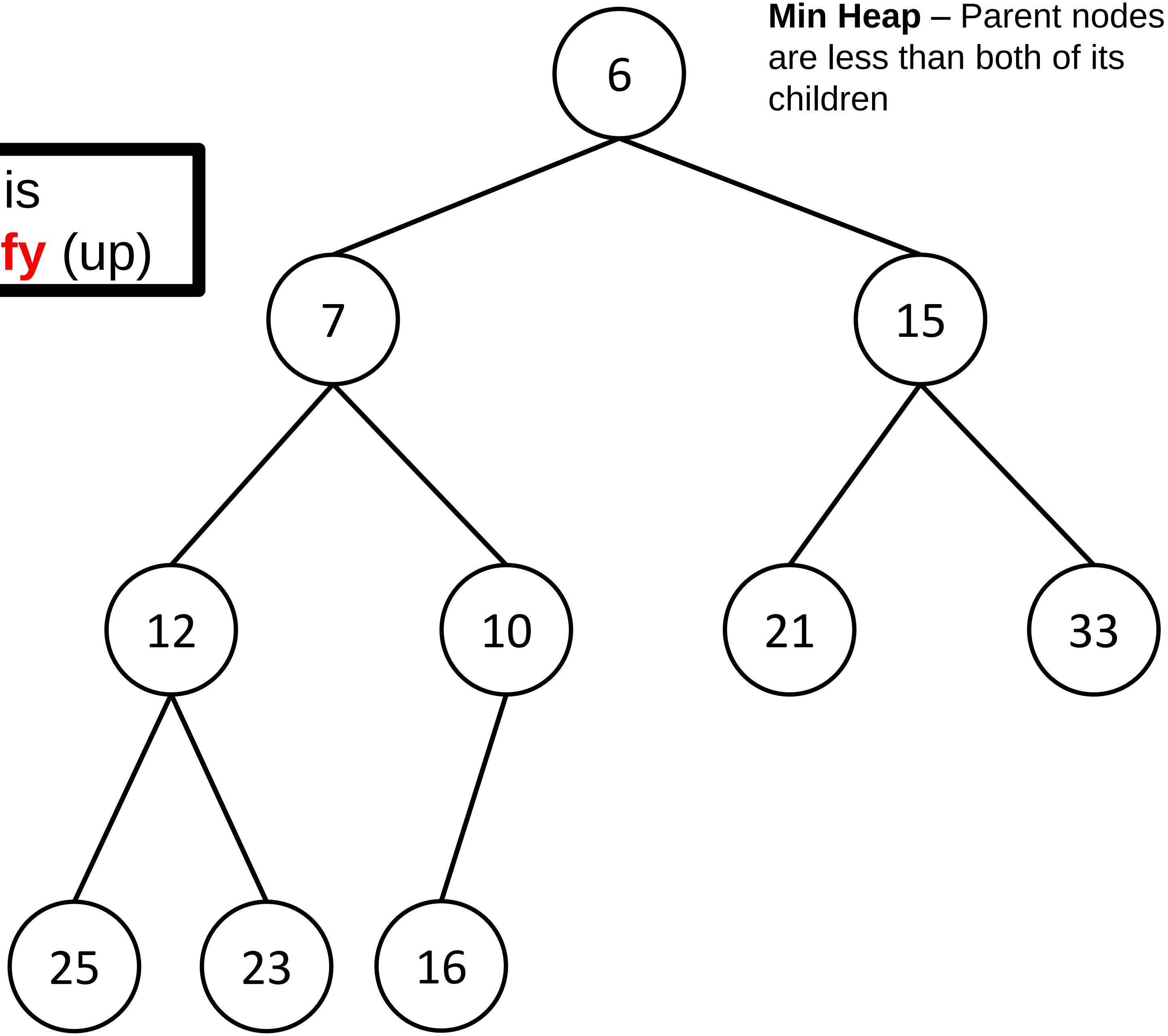
When new nodes are added, we may need to move it up in the tree

Heap Operations - Insert

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

```
add(7);
```

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



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

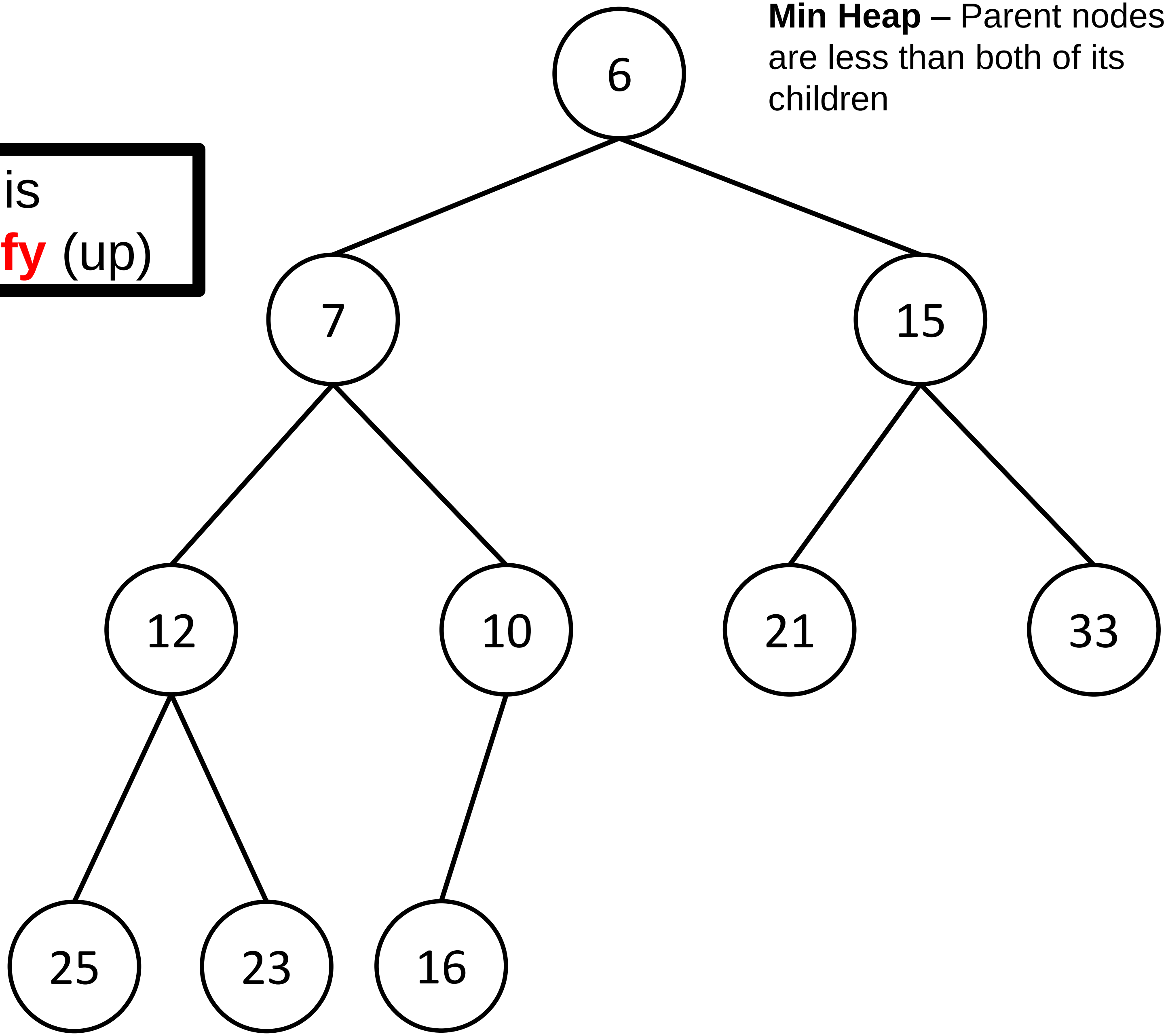
However, we are now violating the heap property

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)



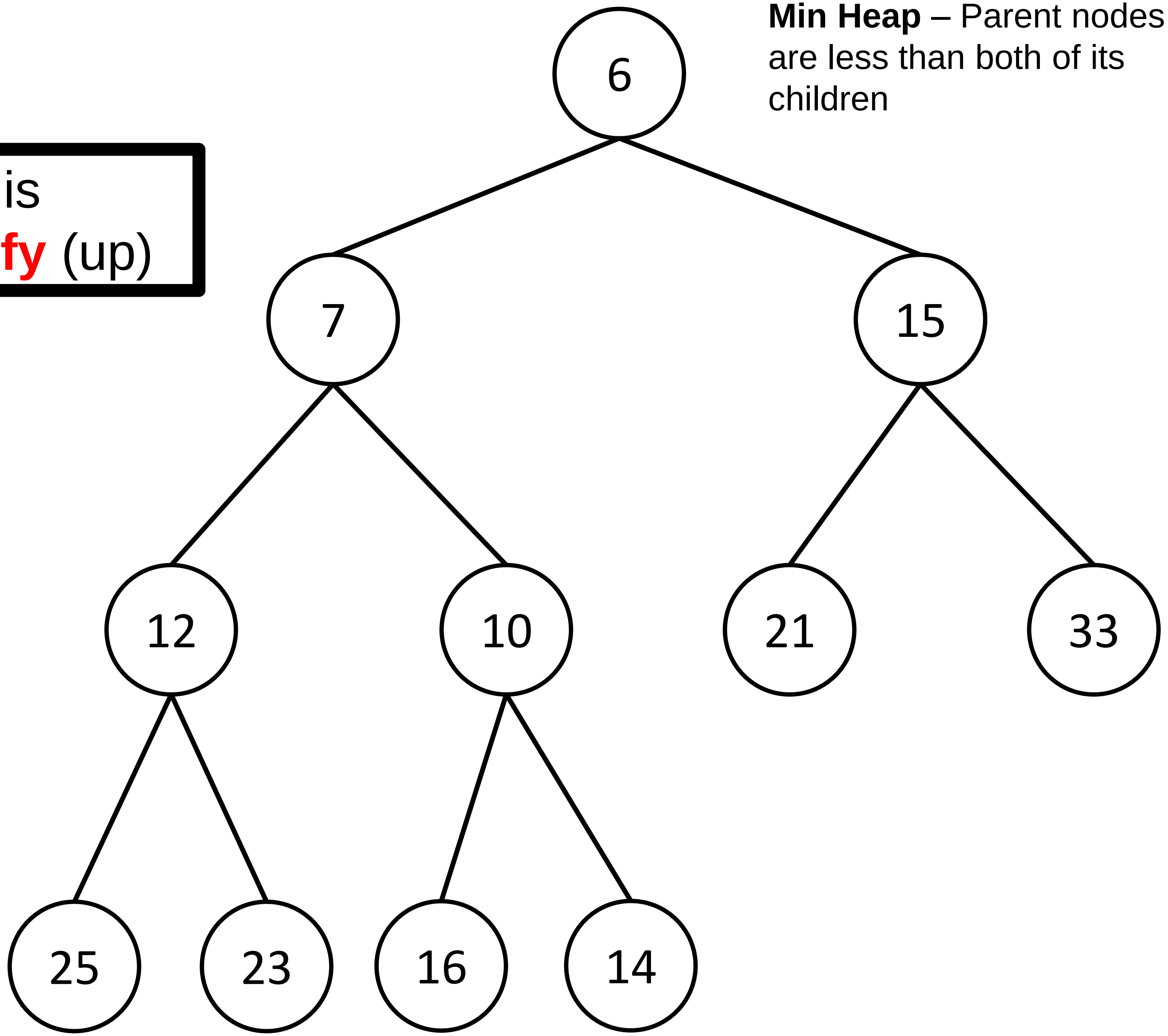
Min Heap – Parent nodes are less than both of its children

Heap Operations - Insert

```
add(7);

add(14);
```

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

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

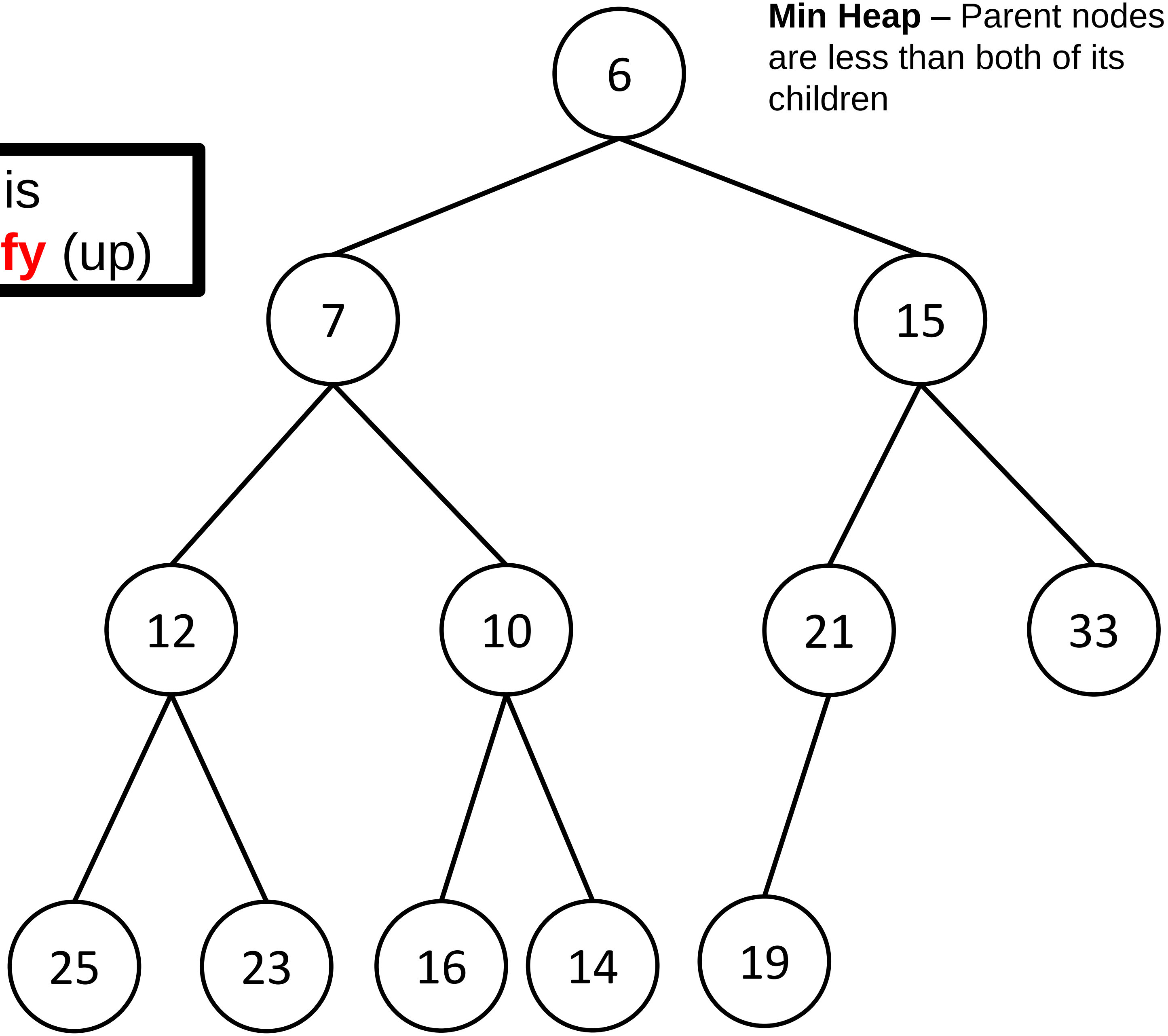


Heap Operations - Insert

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

```
add(7);

add(14);

add(19);
```

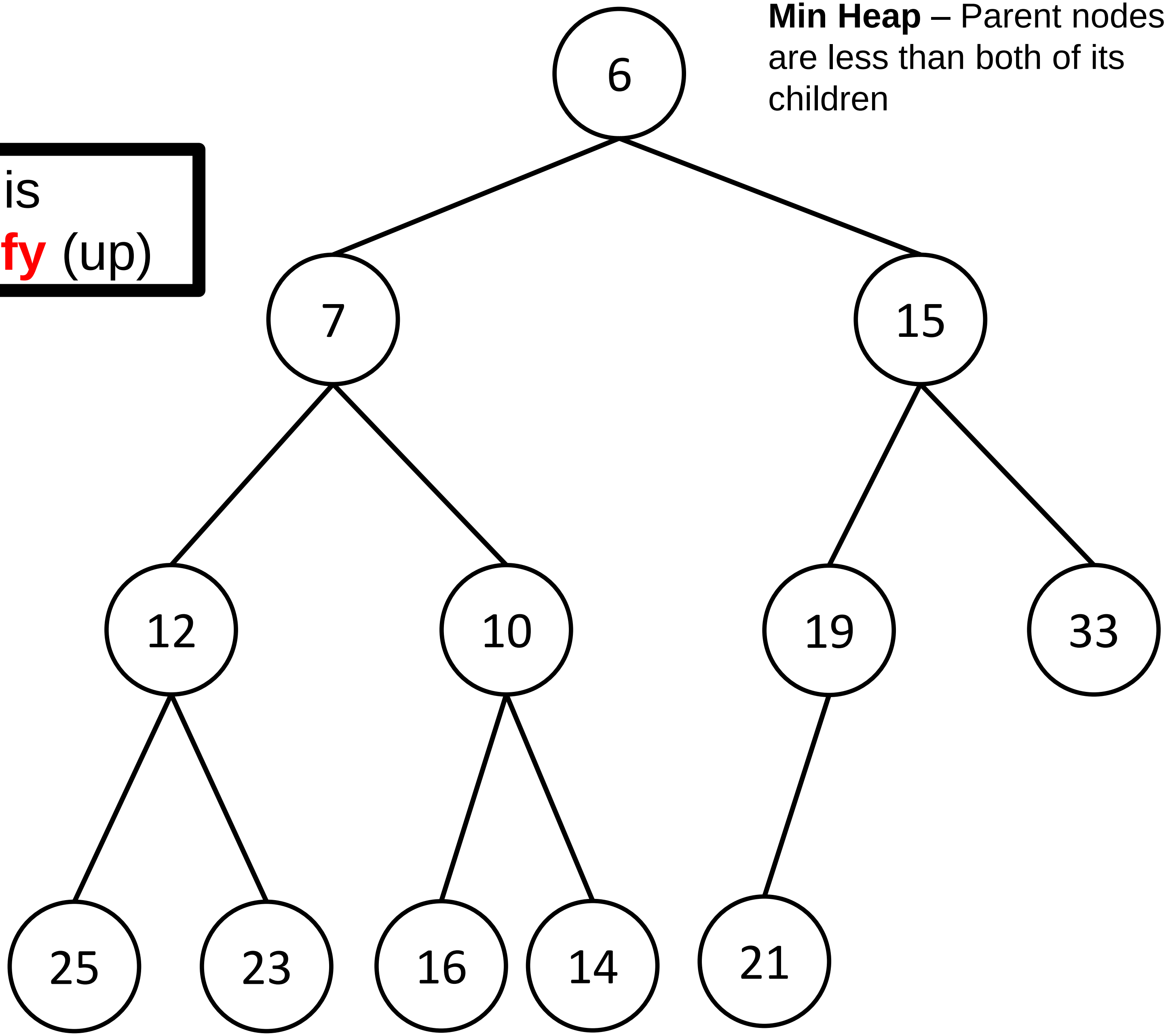
This process is called **Heapify** (up)



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

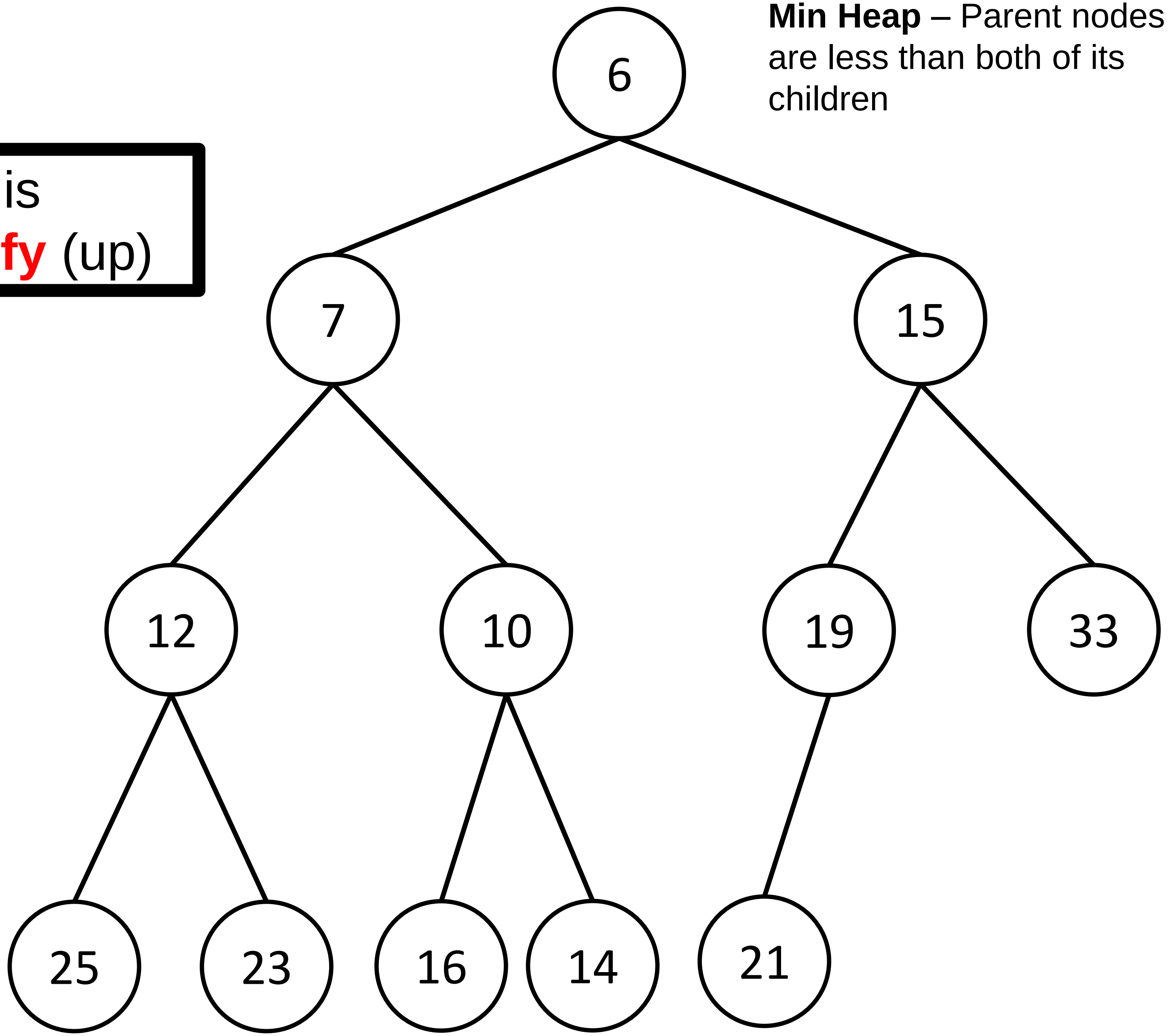
```
add(7);

add(14);

add(19);
```

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

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)**

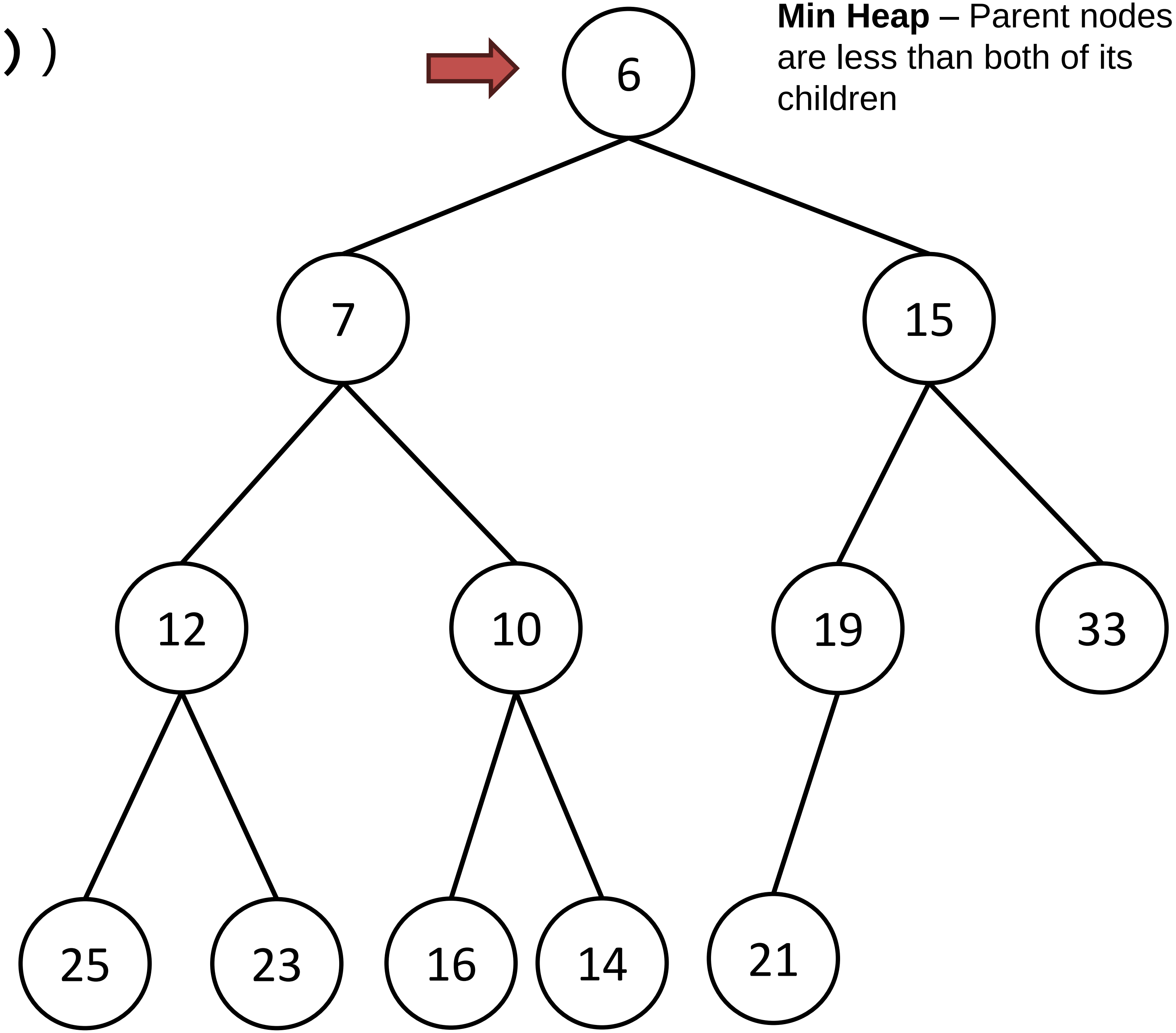


Min Heap – Parent nodes are less than both of its children

Heap Operations – Removal (`poll()`)

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

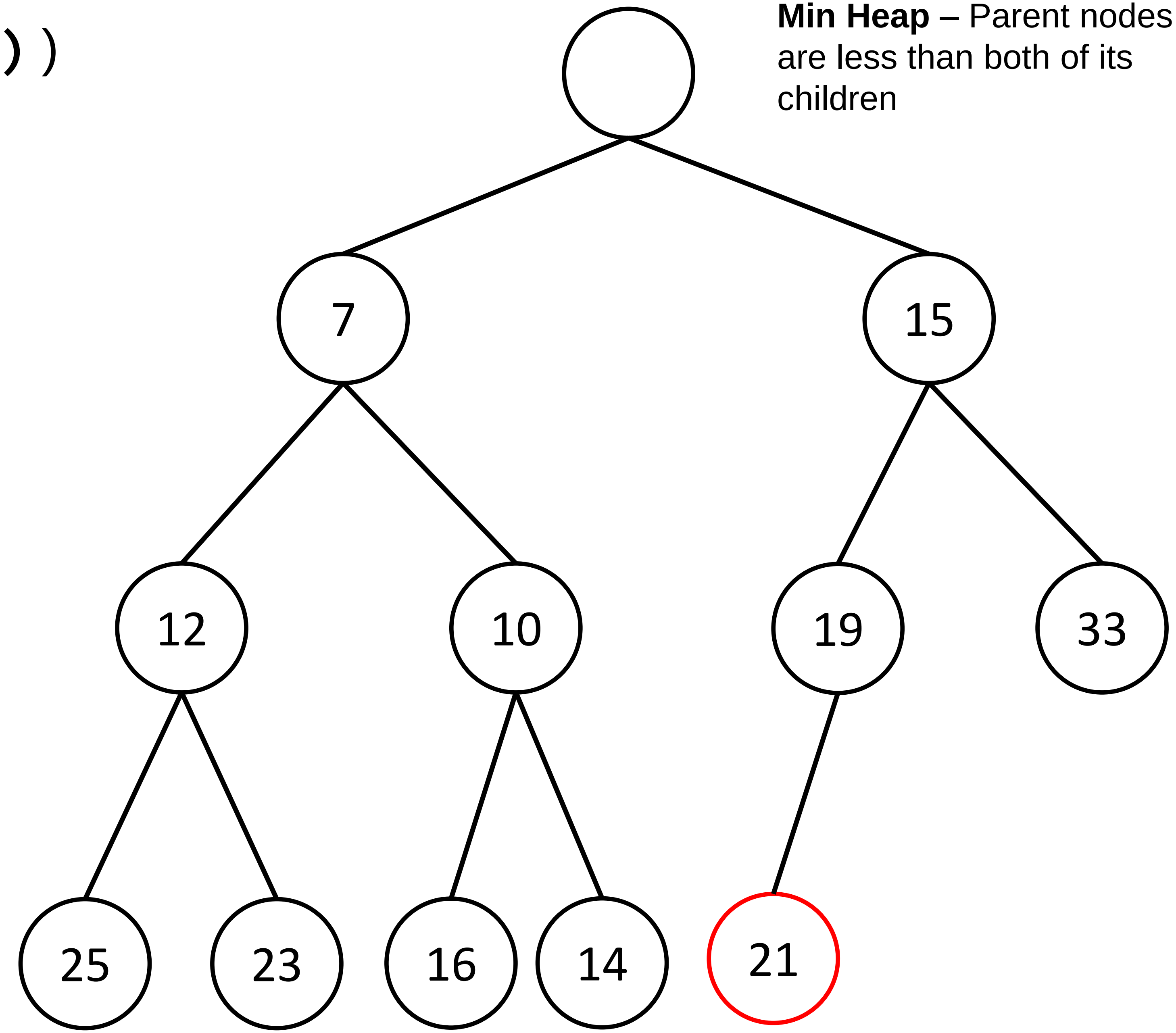
Min Heap – Parent nodes are less than both of its children



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**



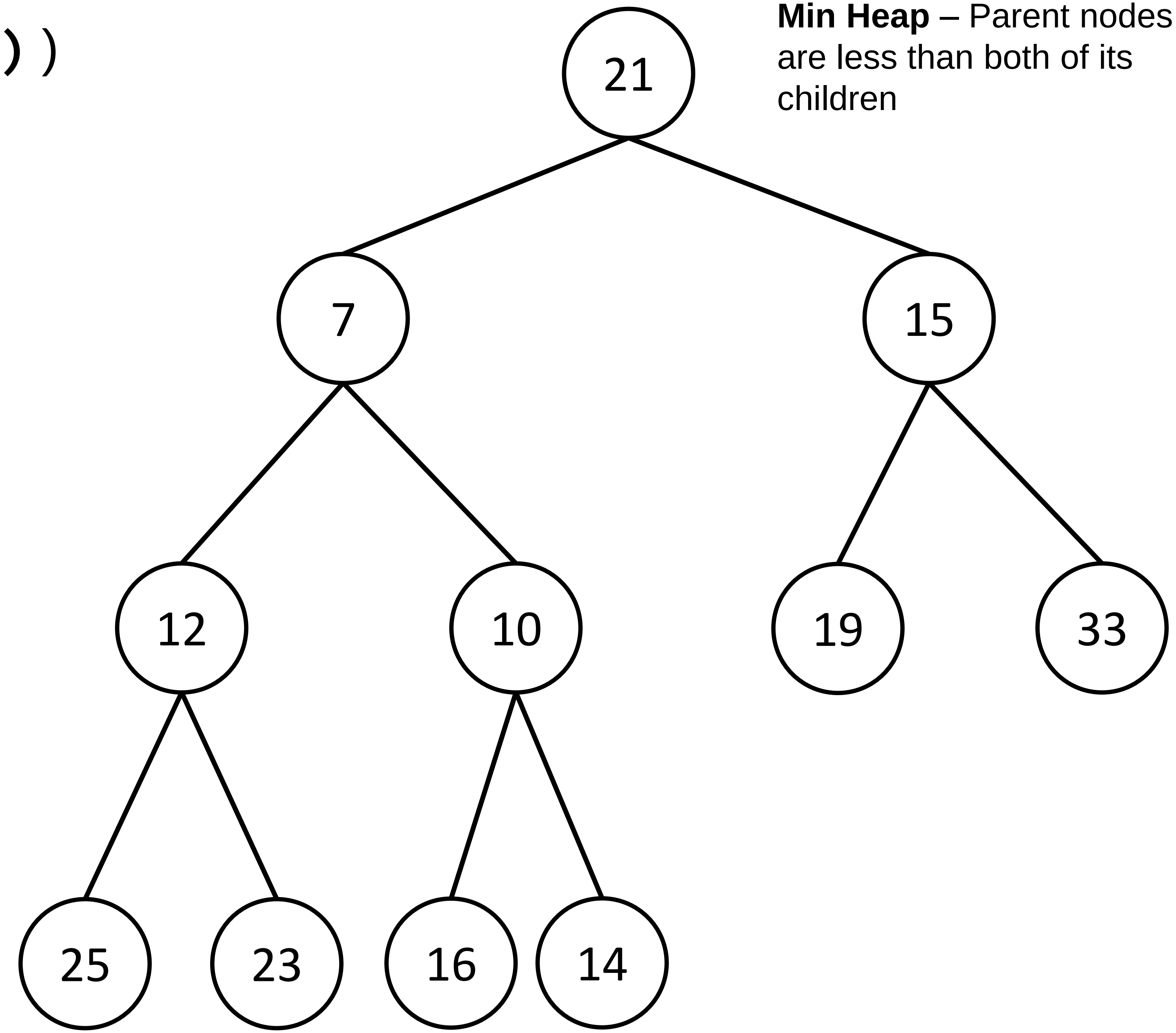
Min Heap – Parent nodes are less than both of its children

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

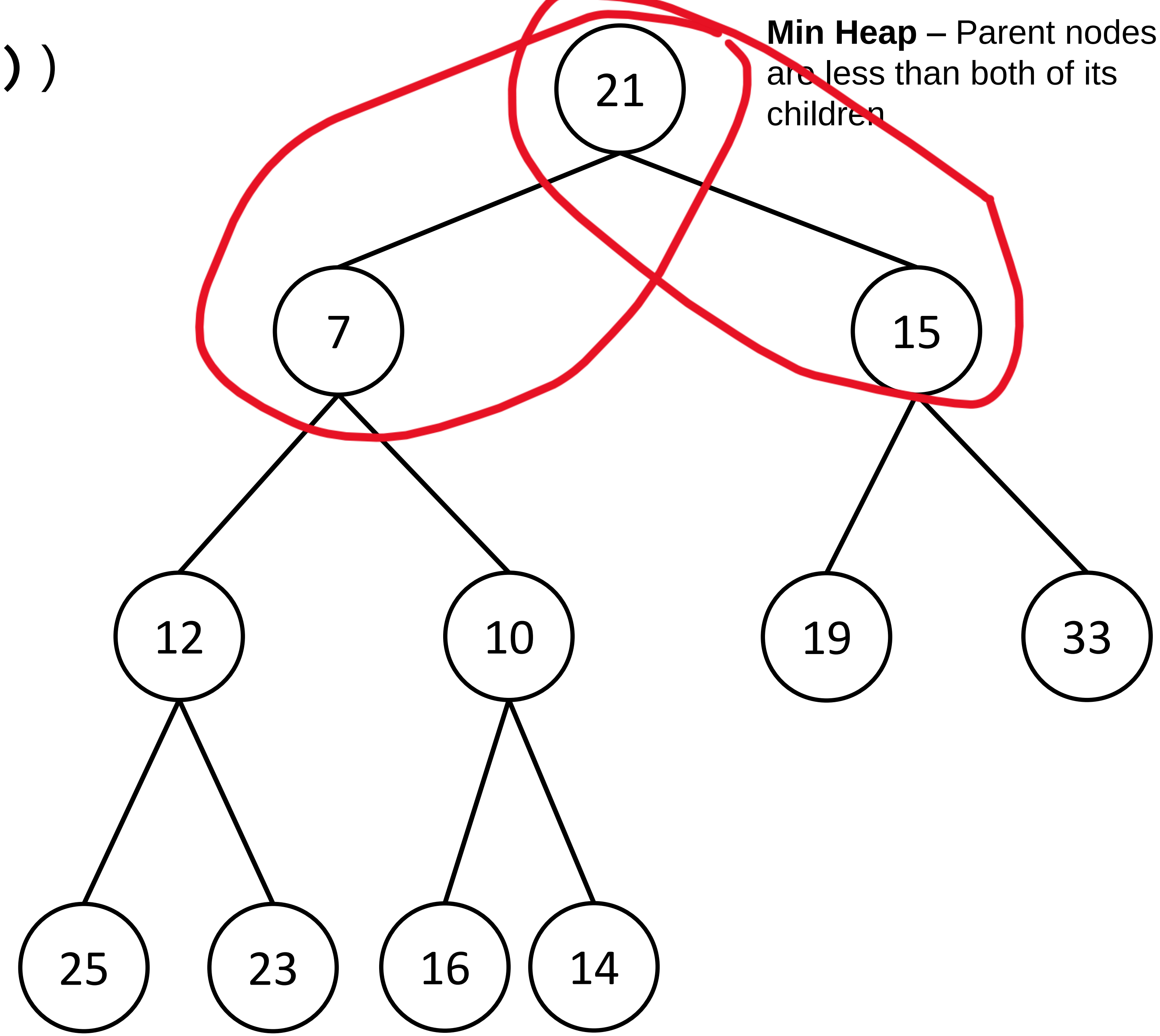
When the root is removed, we replace it with **the last node that was added to the heap**



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

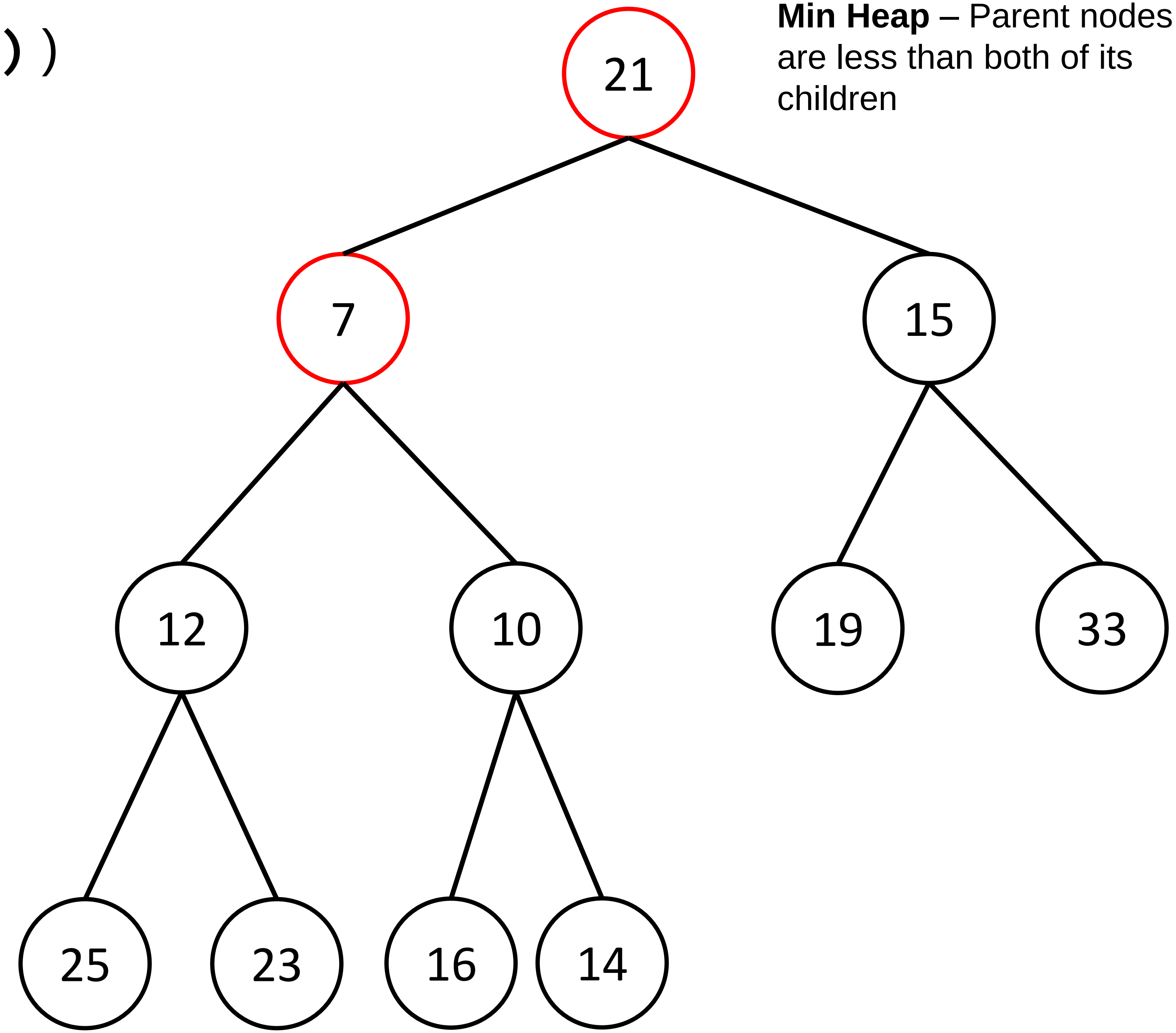
Heap Operations – Removal (`poll()`)

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



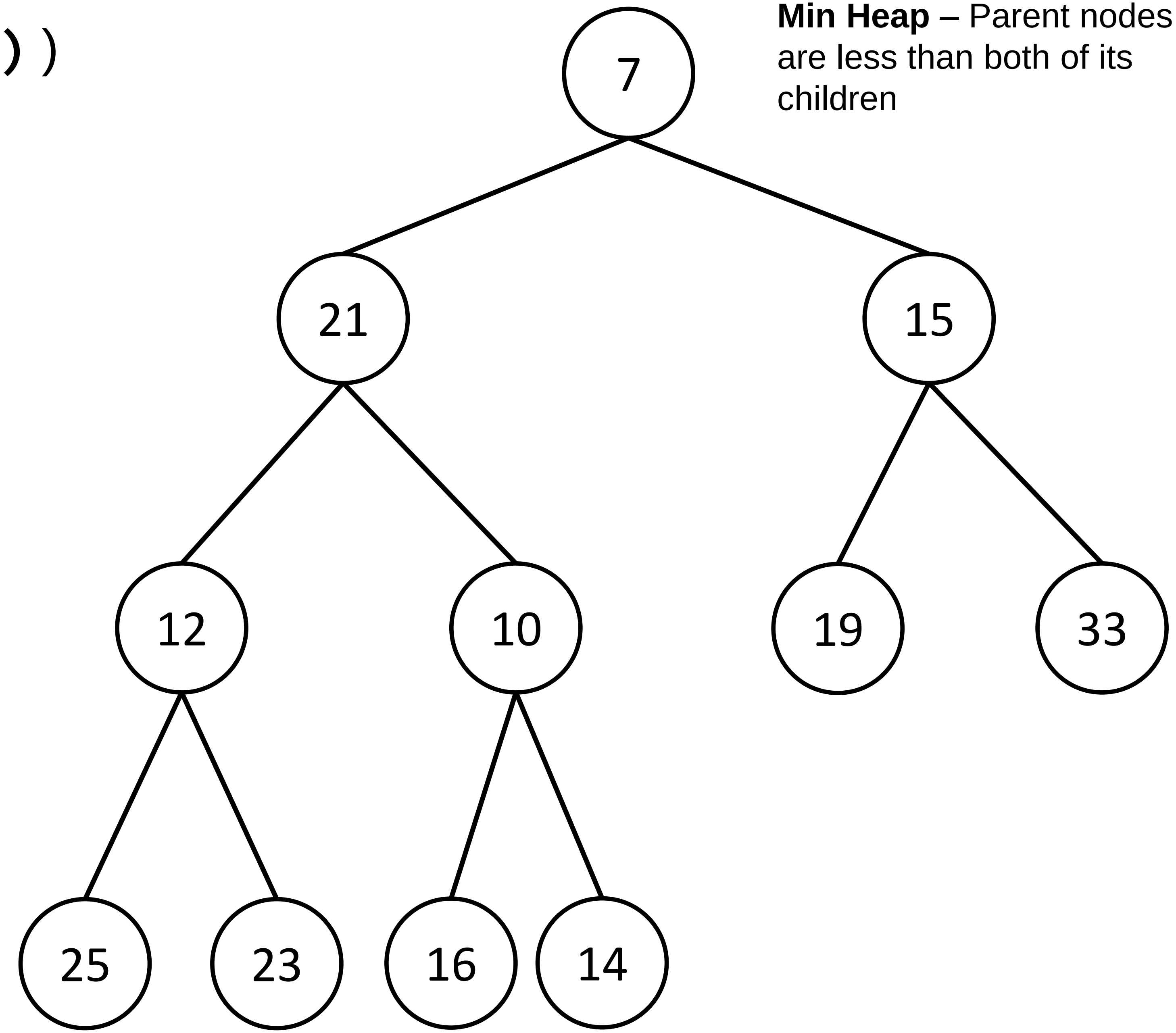
Heap Operations – Removal (`poll()`)

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

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

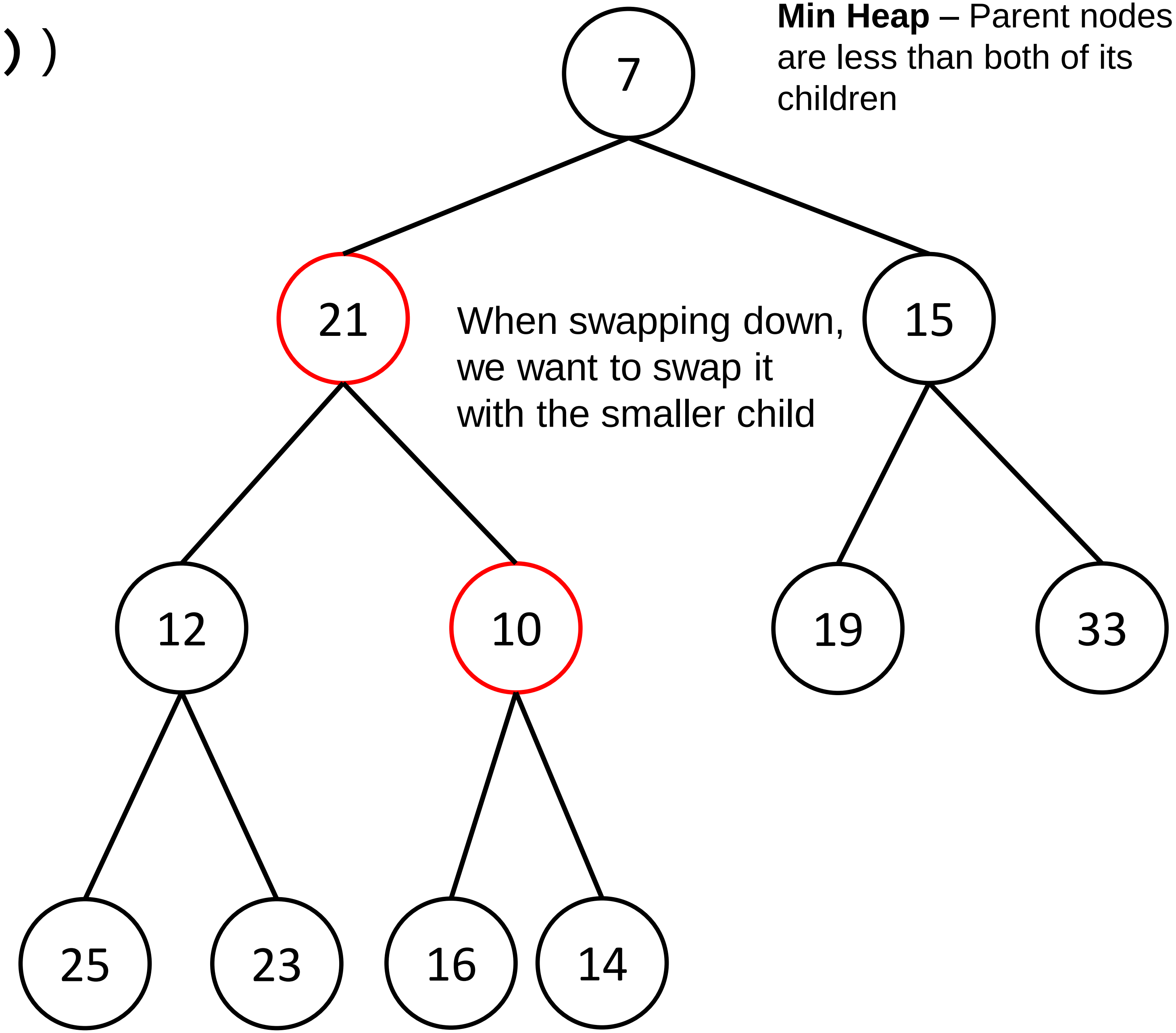


Heap Operations – Removal (**`poll()`**)

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



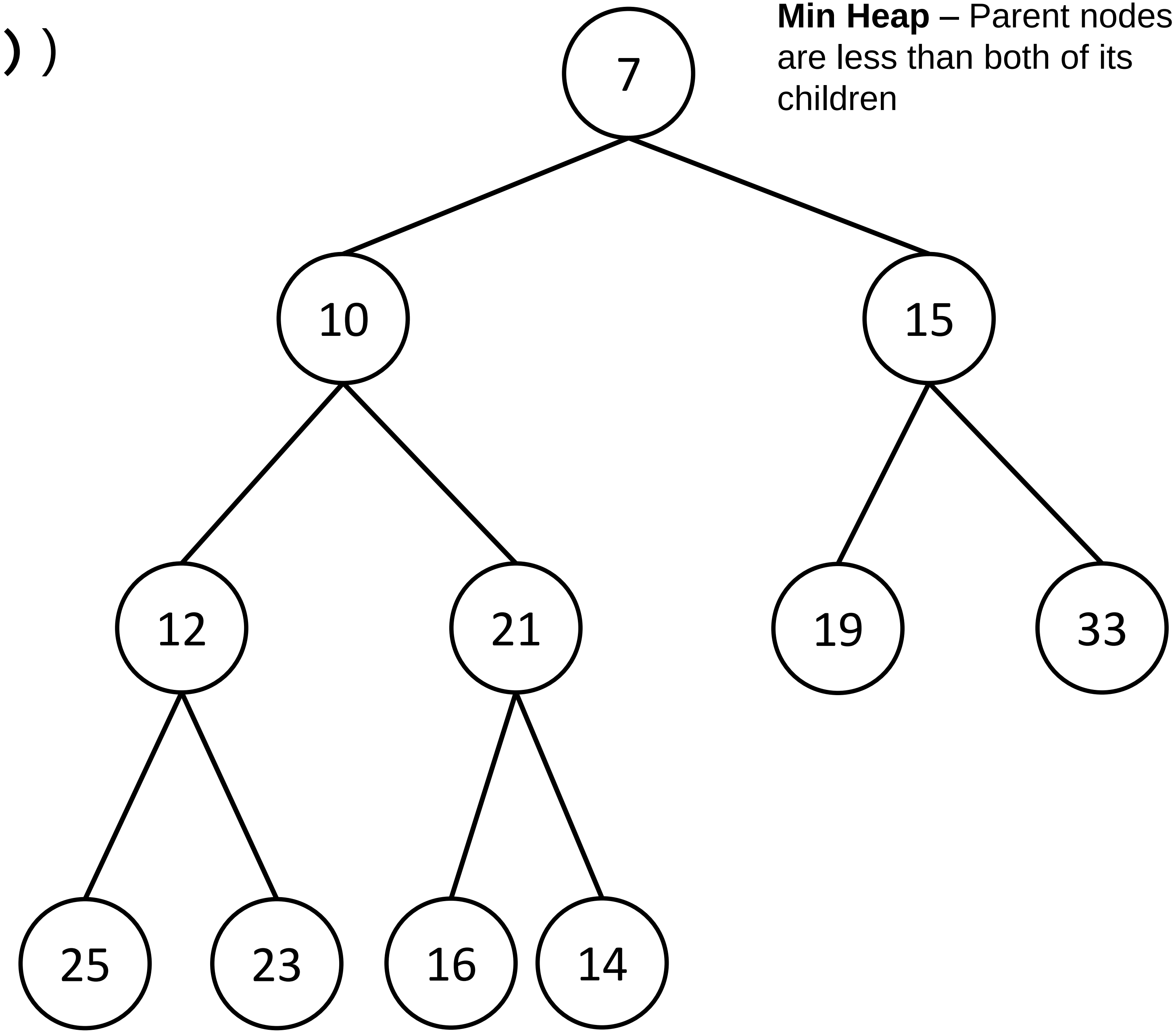
Heap Operations – Removal (`poll()`)

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



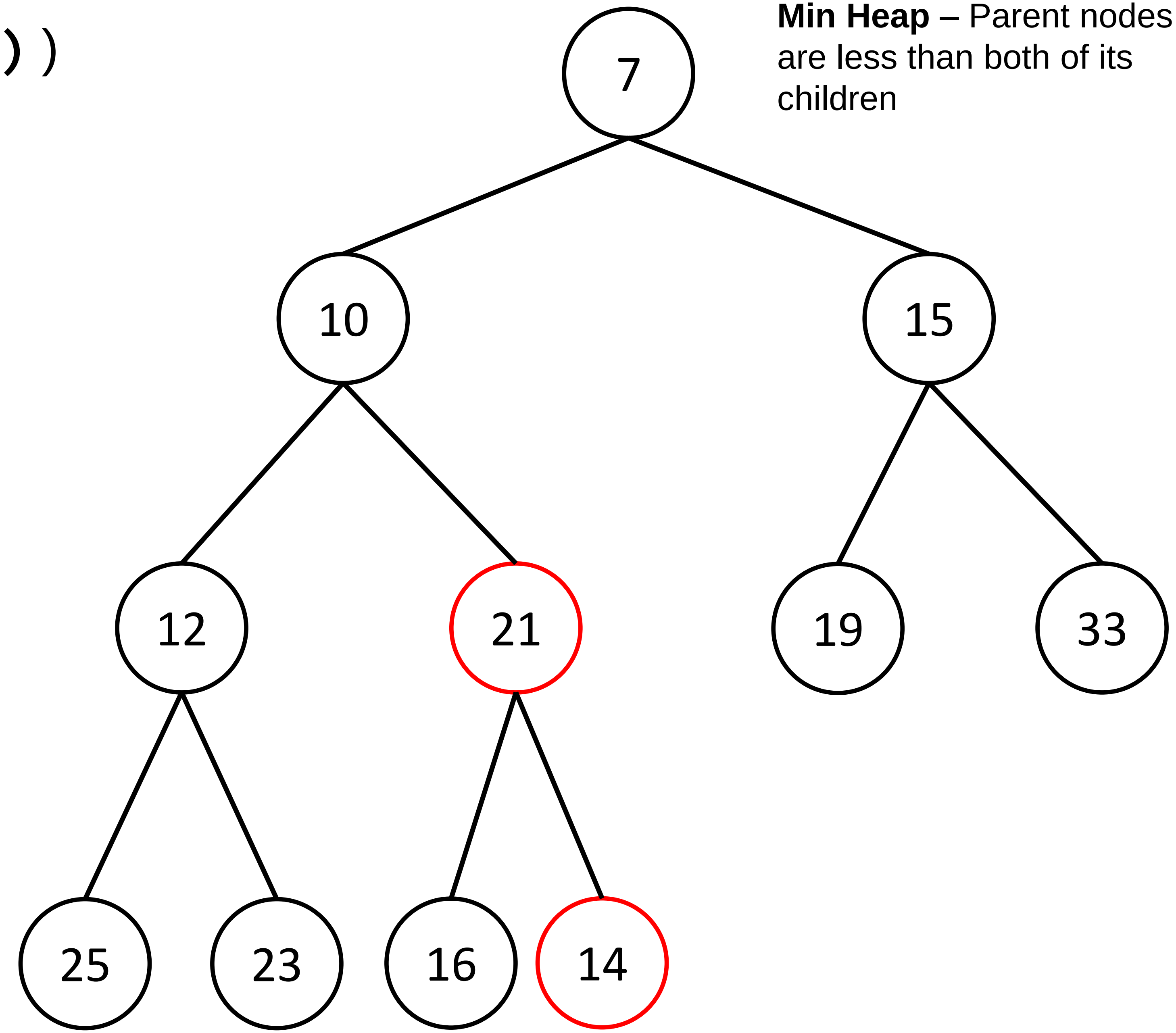
Heap Operations – Removal (`poll()`)

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

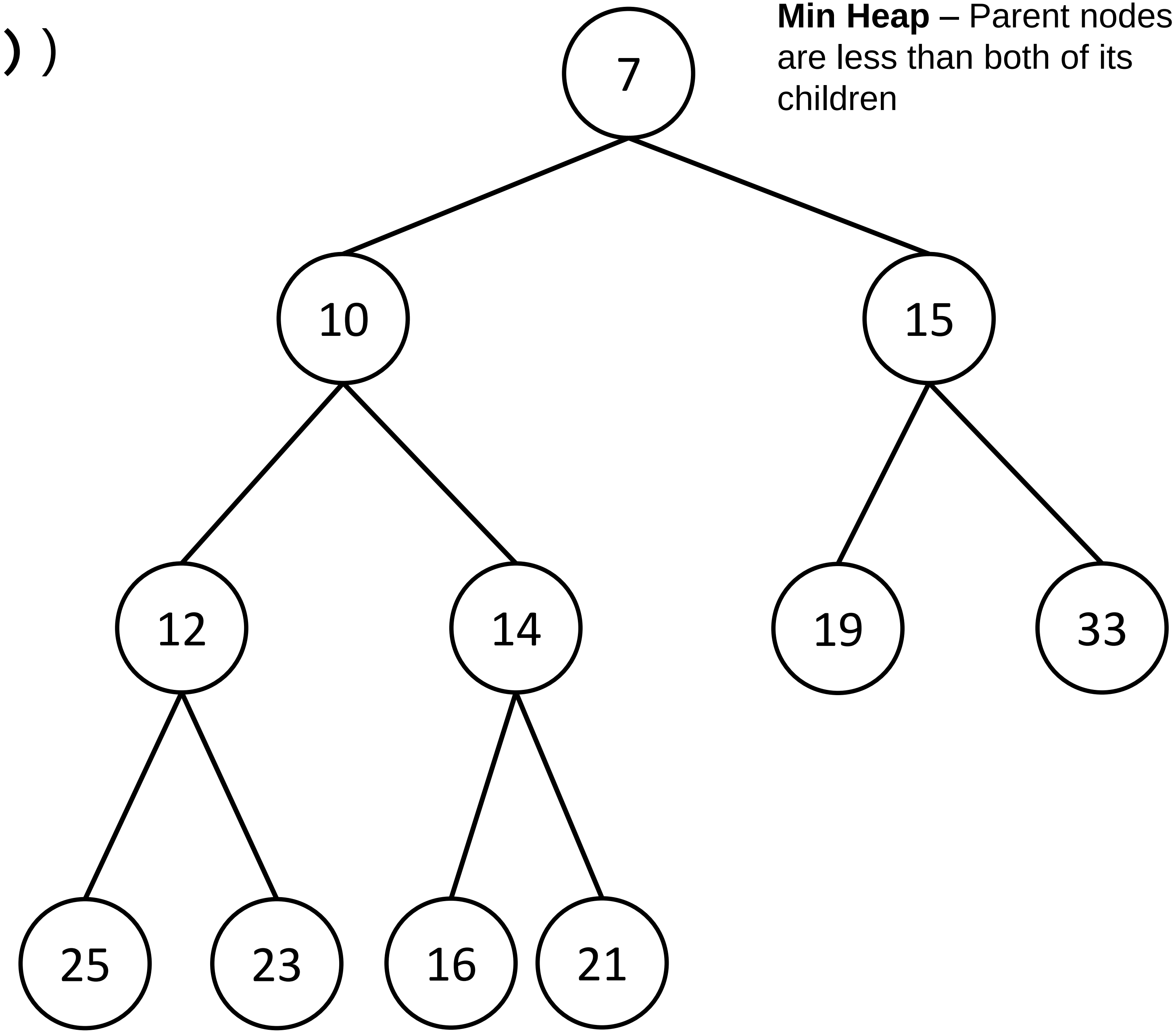


Heap Operations – Removal (`poll()`)

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

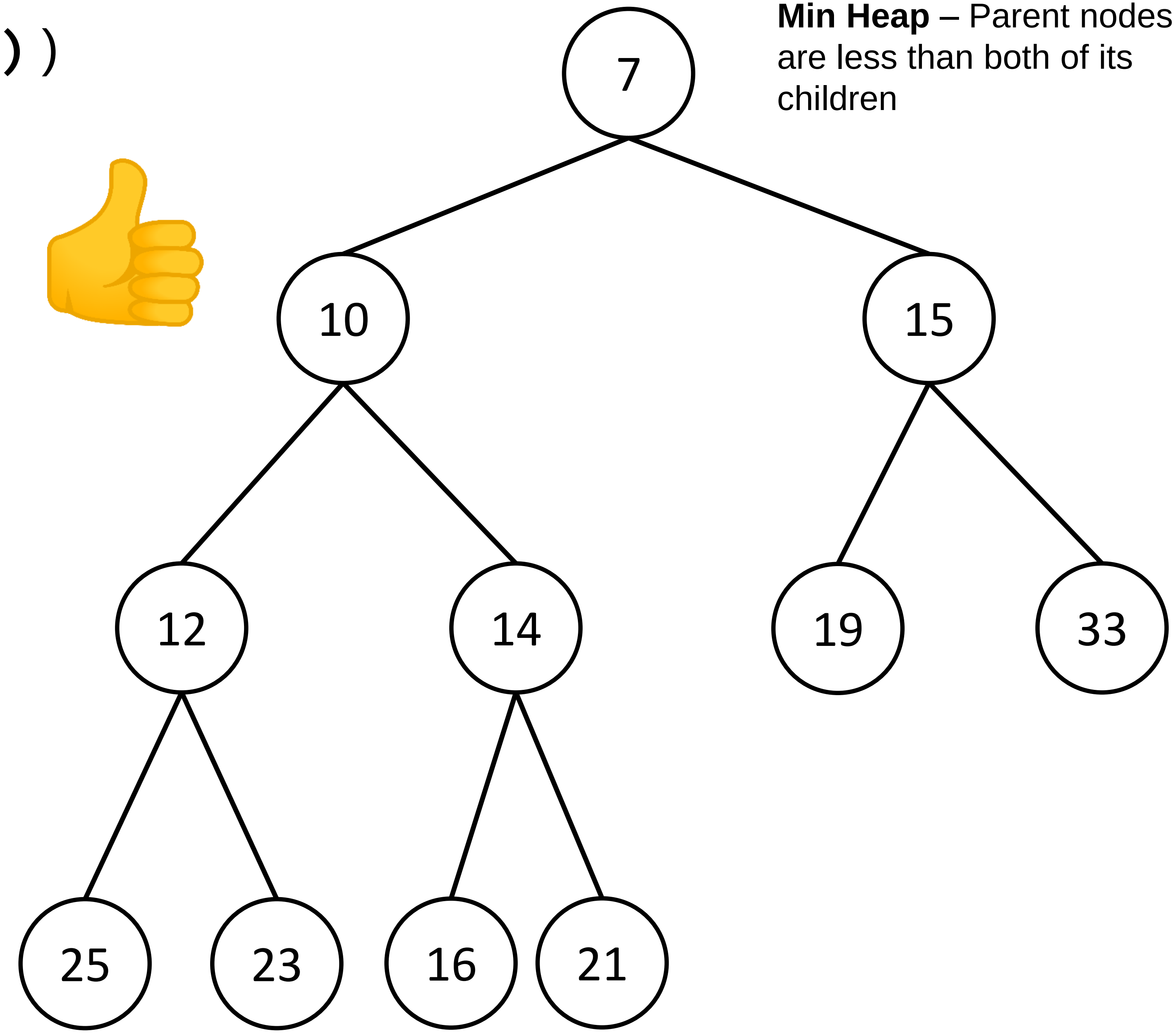
Heap Operations – Removal (**`poll()`**)

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()`)

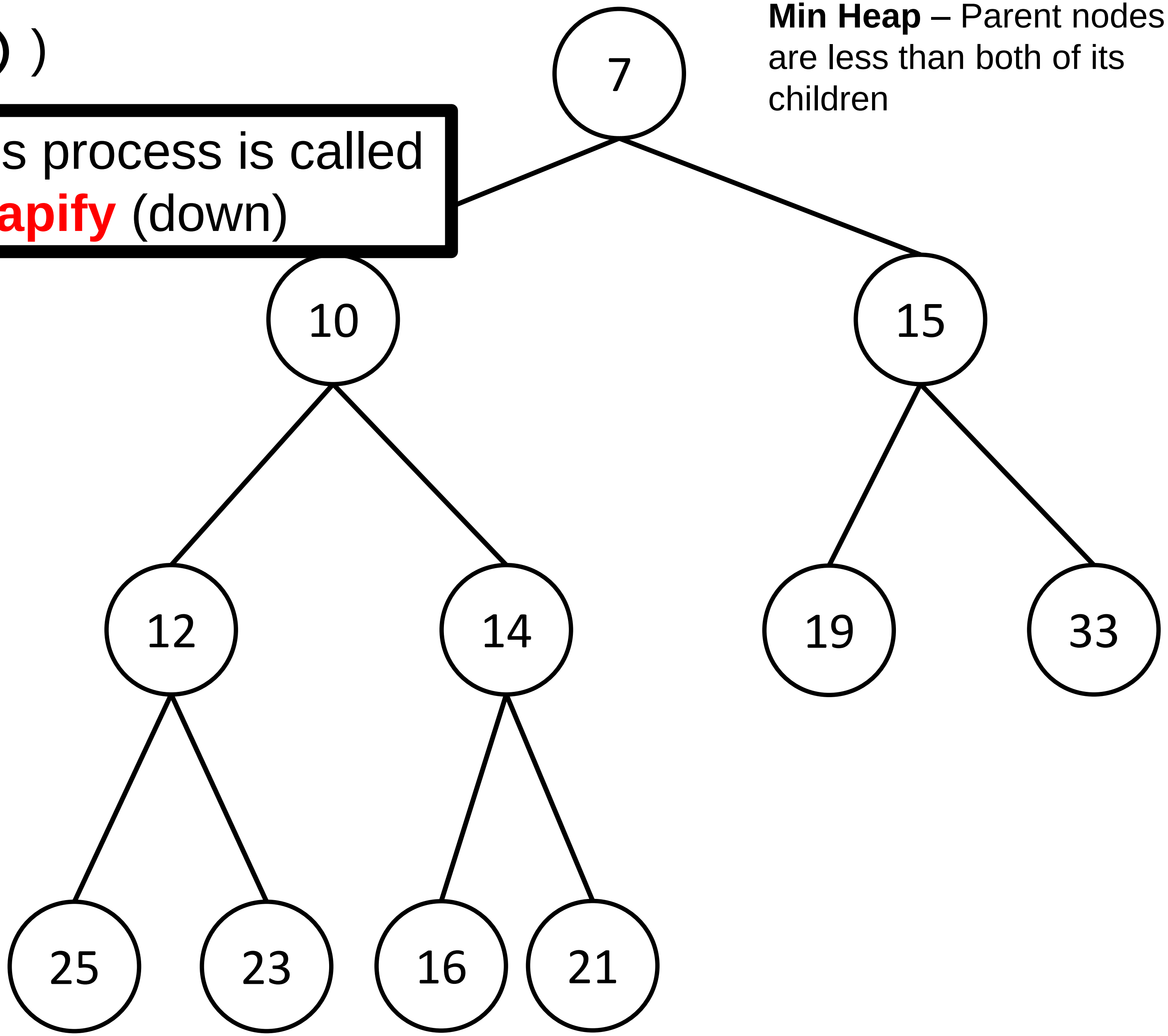
Min Heap – Parent nodes are less than both of its children

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**

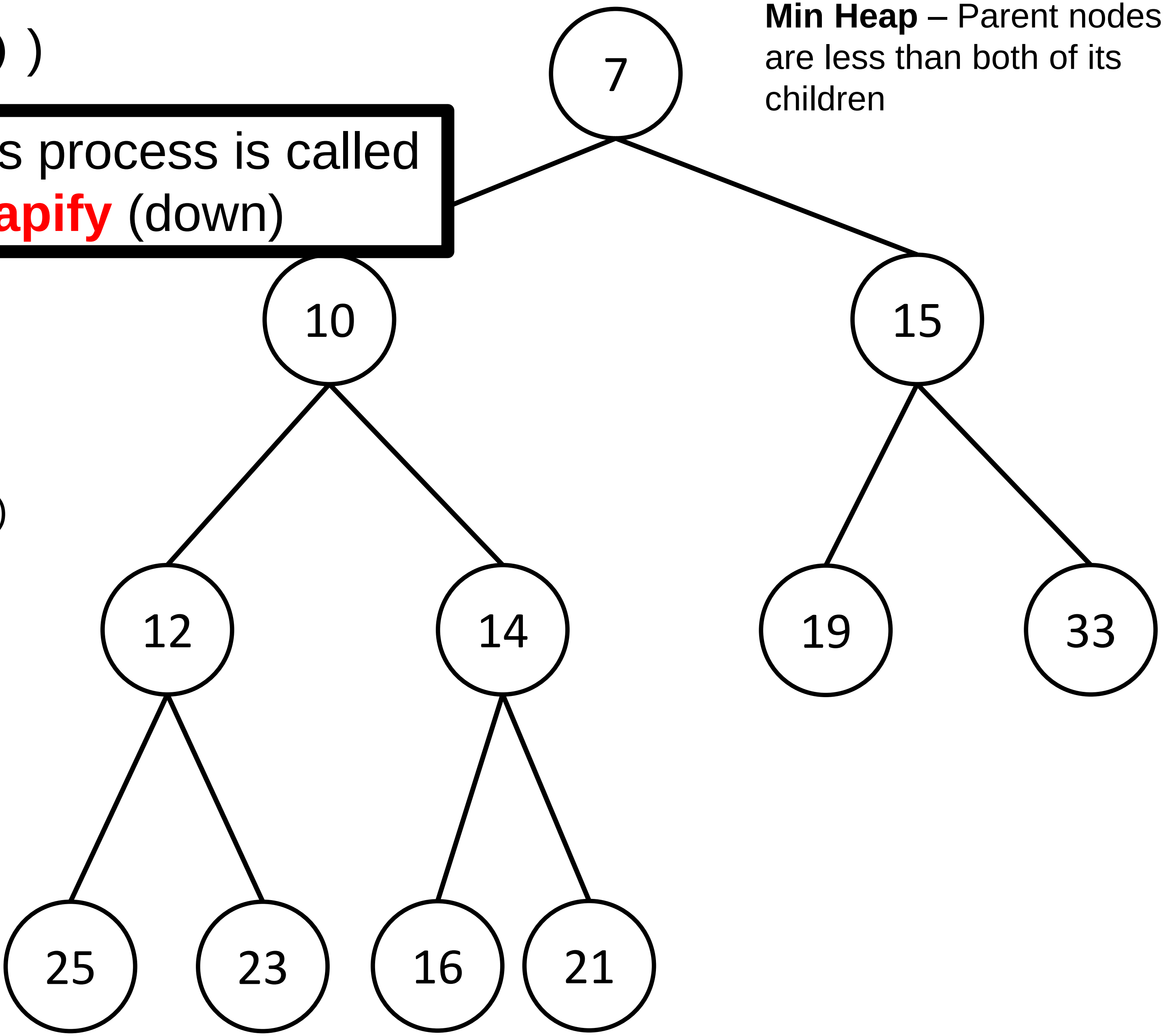
When the root is replaced, it may need to be moved down in the tree



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)**

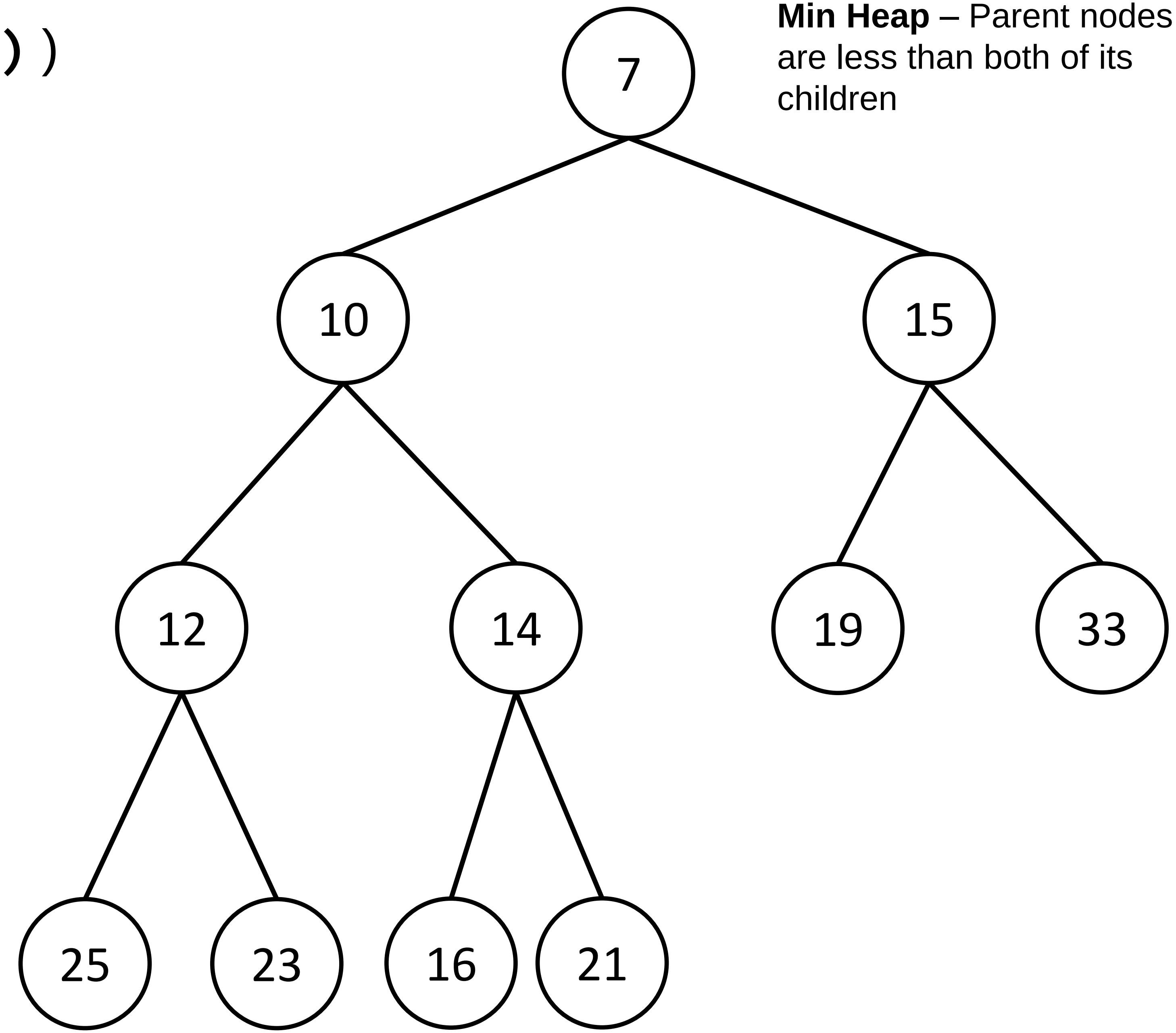
Heap Operations – Removal (`poll()`)

Heapify (up)

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

Heapify (down)

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

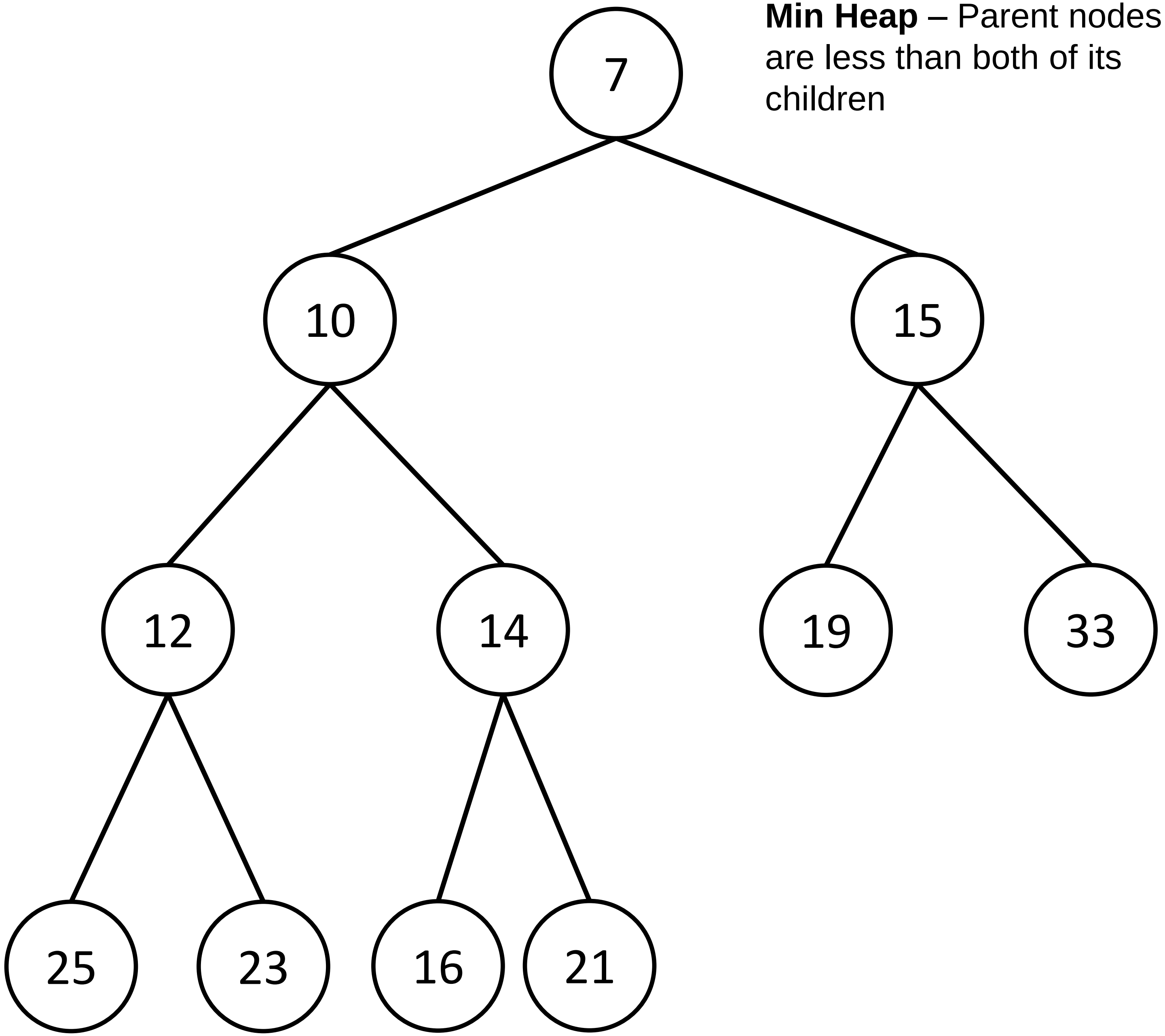


Min Heap – Parent nodes are less than both of its children

Heap Representation

How to represent a heap?

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



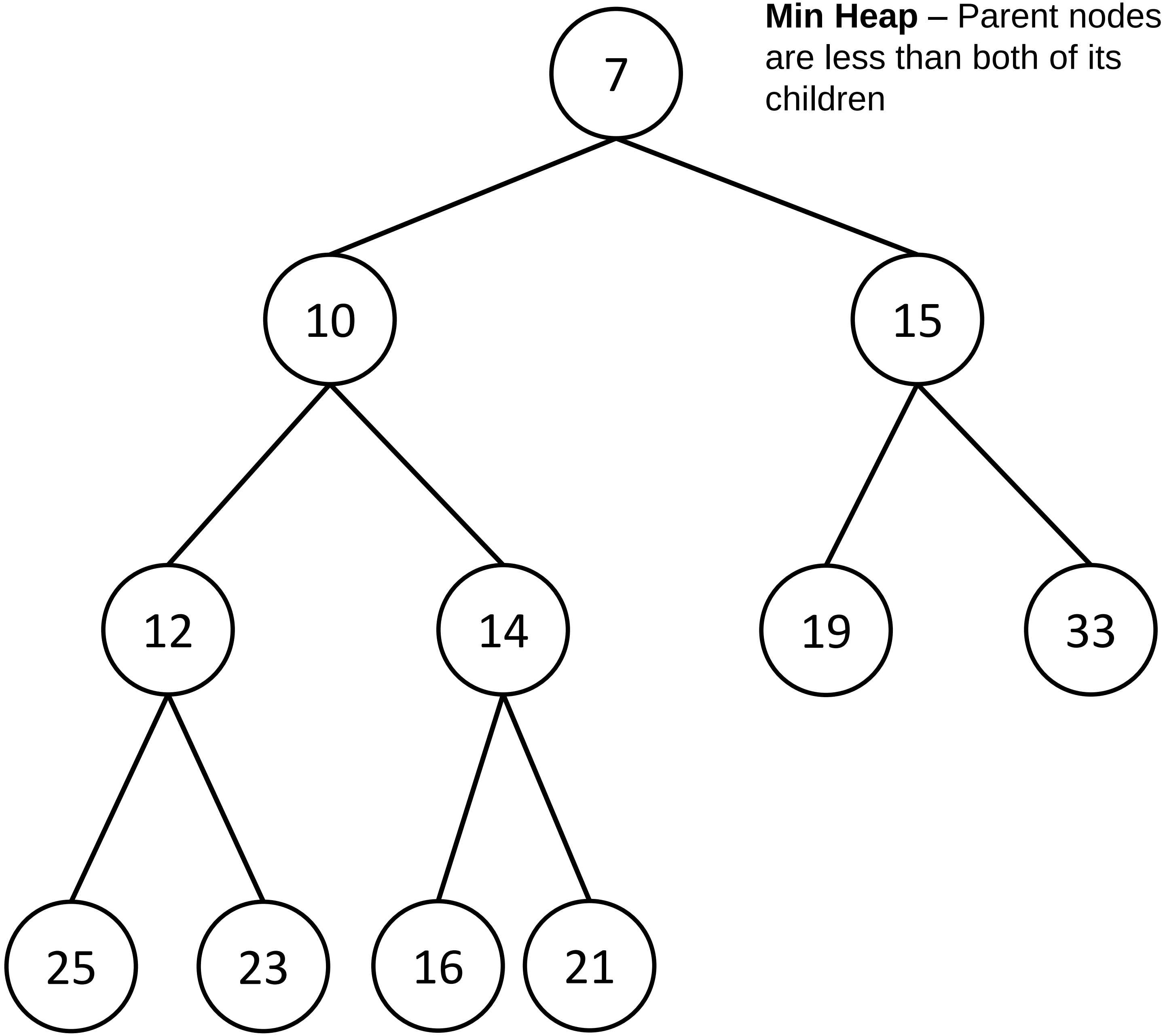
Min Heap – Parent nodes are less than both of its children

Heap Representation

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

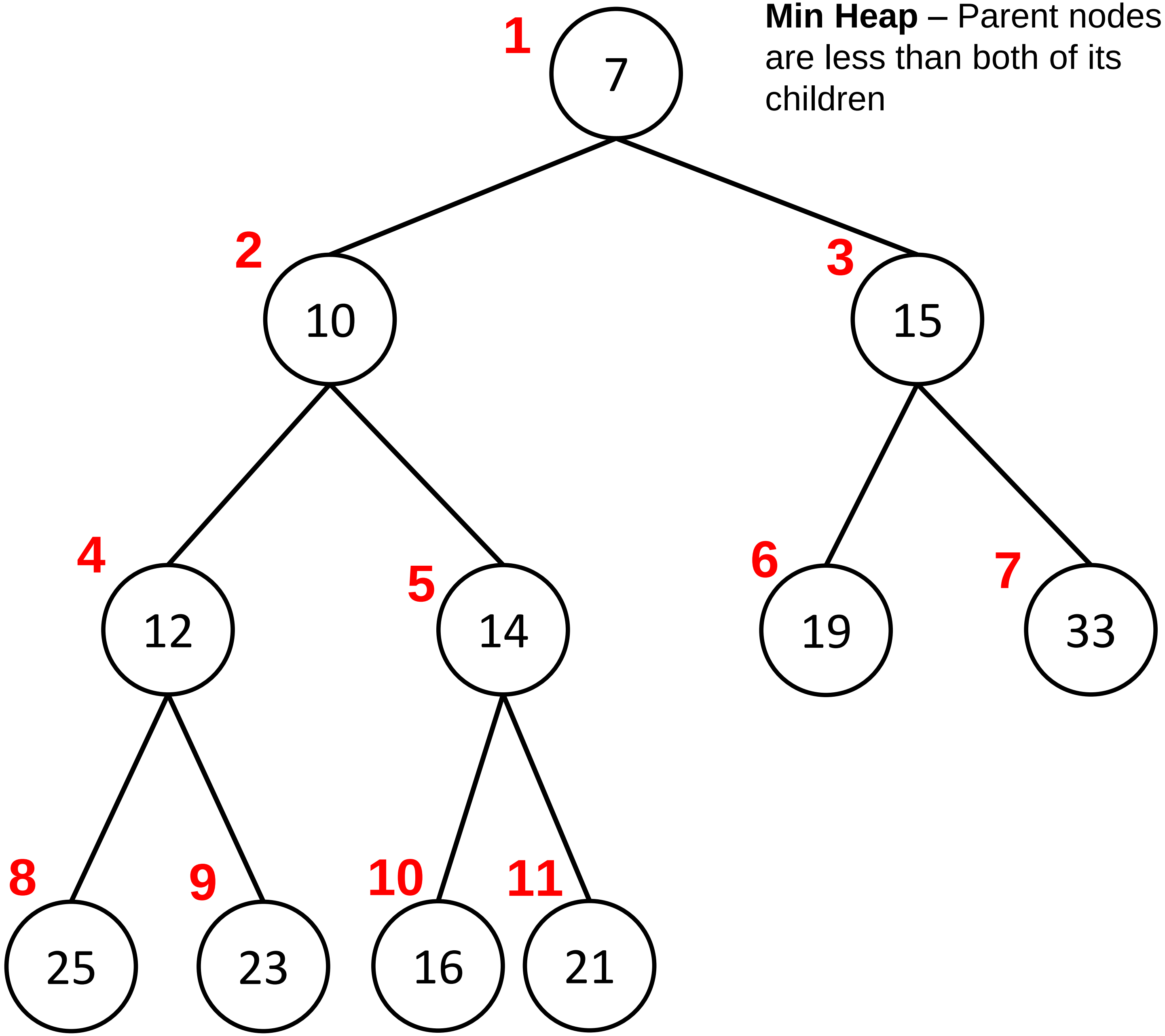
How to represent a heap?

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



Heap Representation

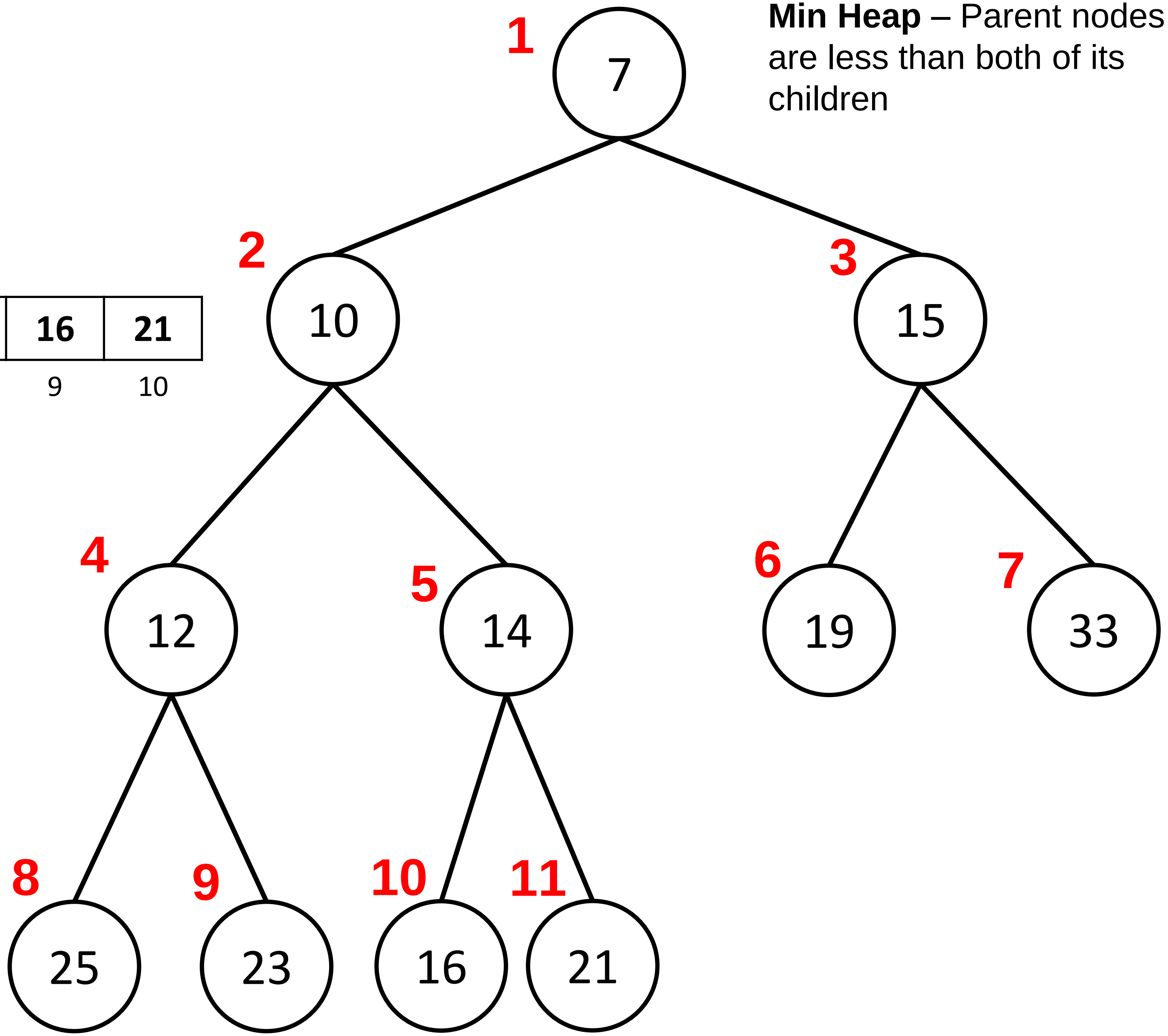
Min Heap – Parent nodes are less than both of its children



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

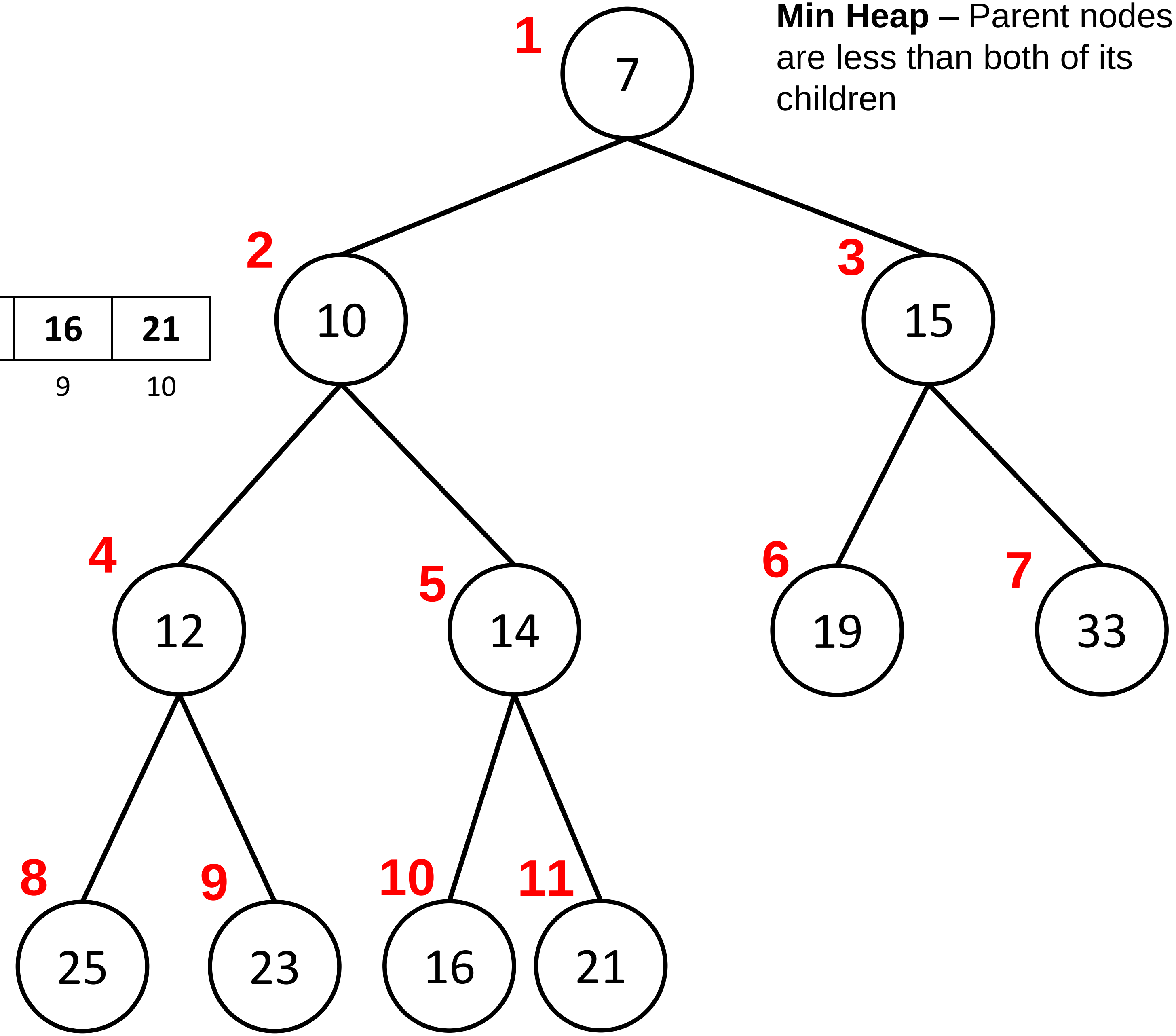
Heap Representation

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?

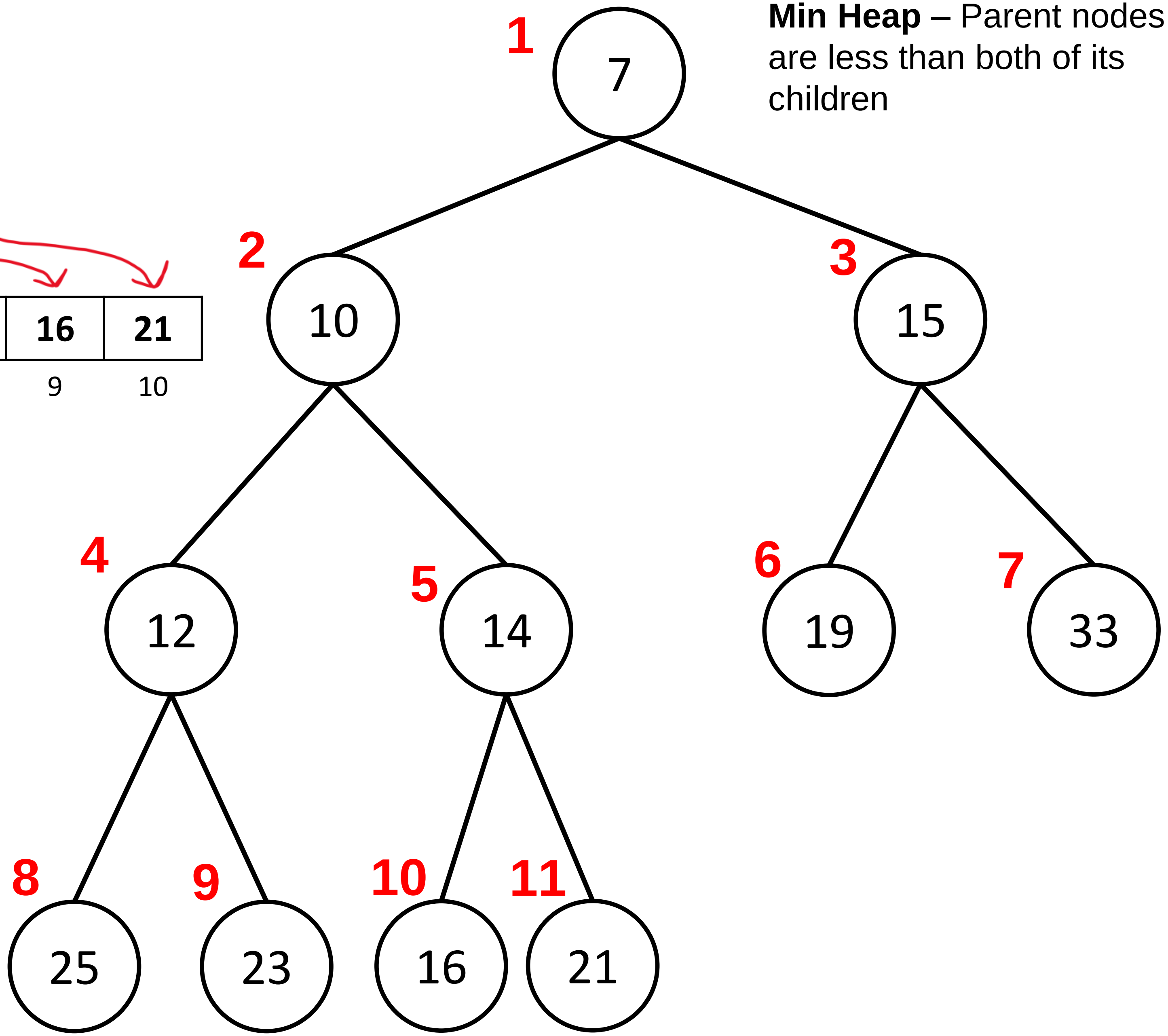


Heap Representation

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

Array

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

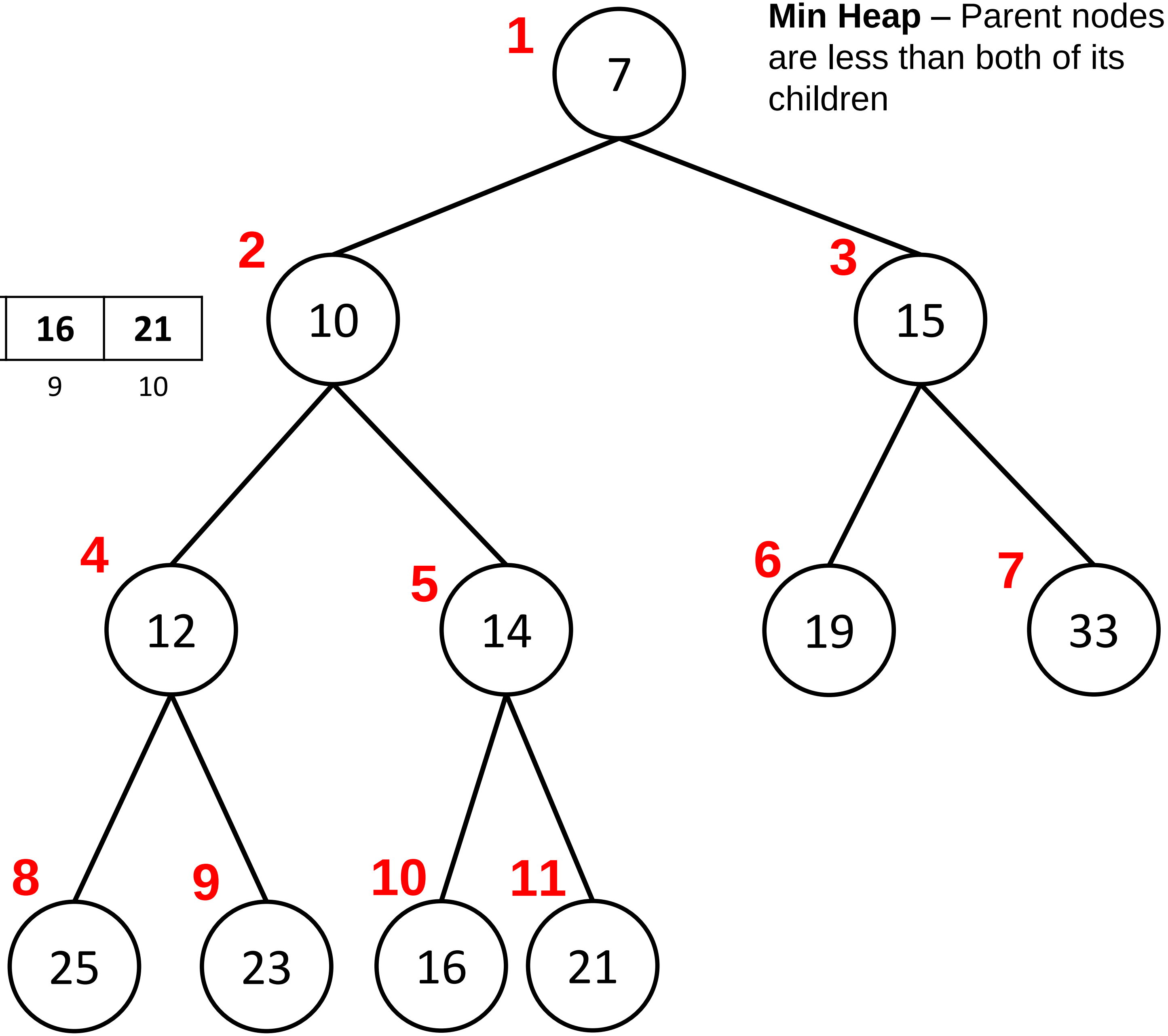
Min Heap – Parent nodes are less than both of its children

Array

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

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

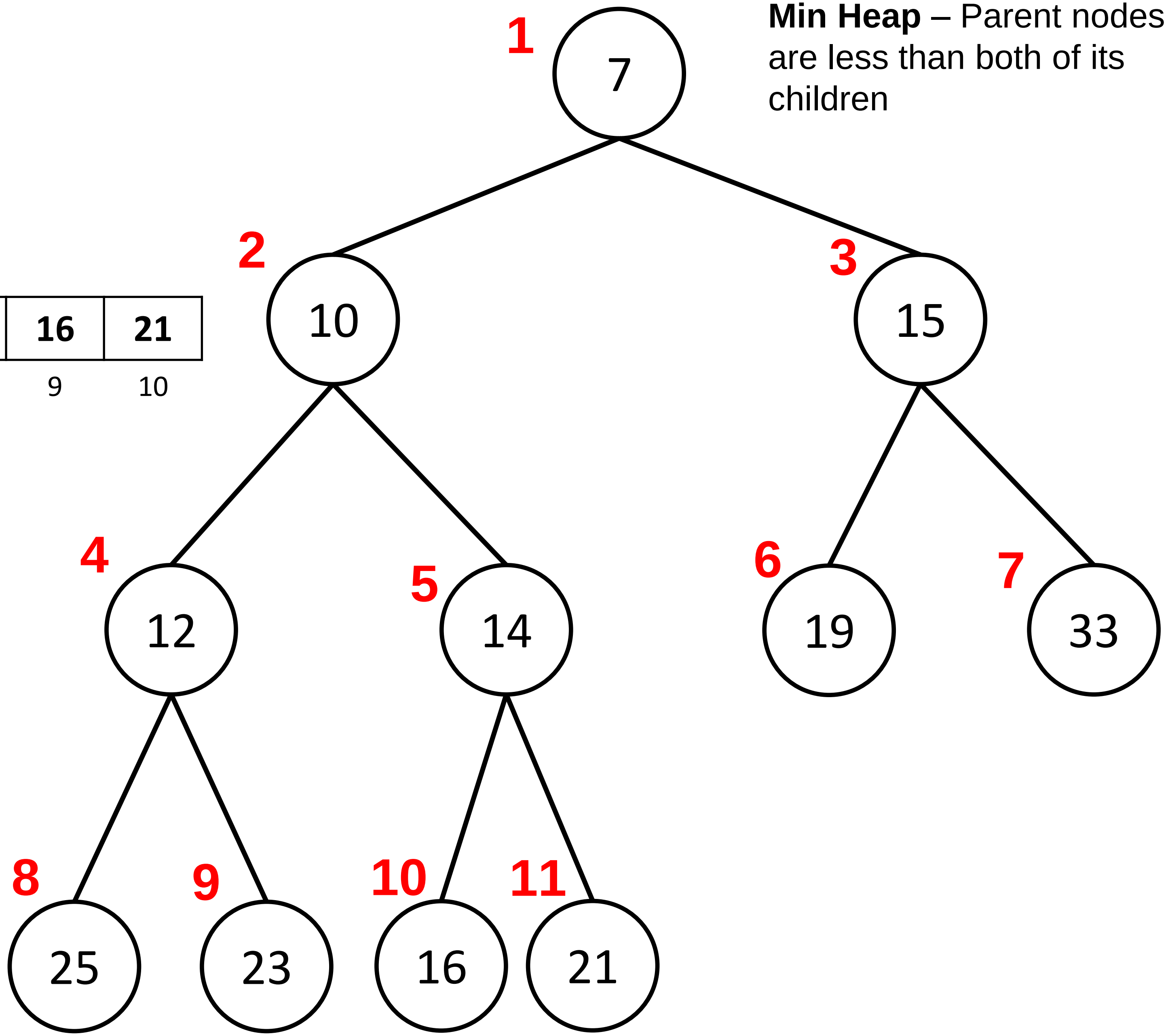
For a given element at index **`i`**

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

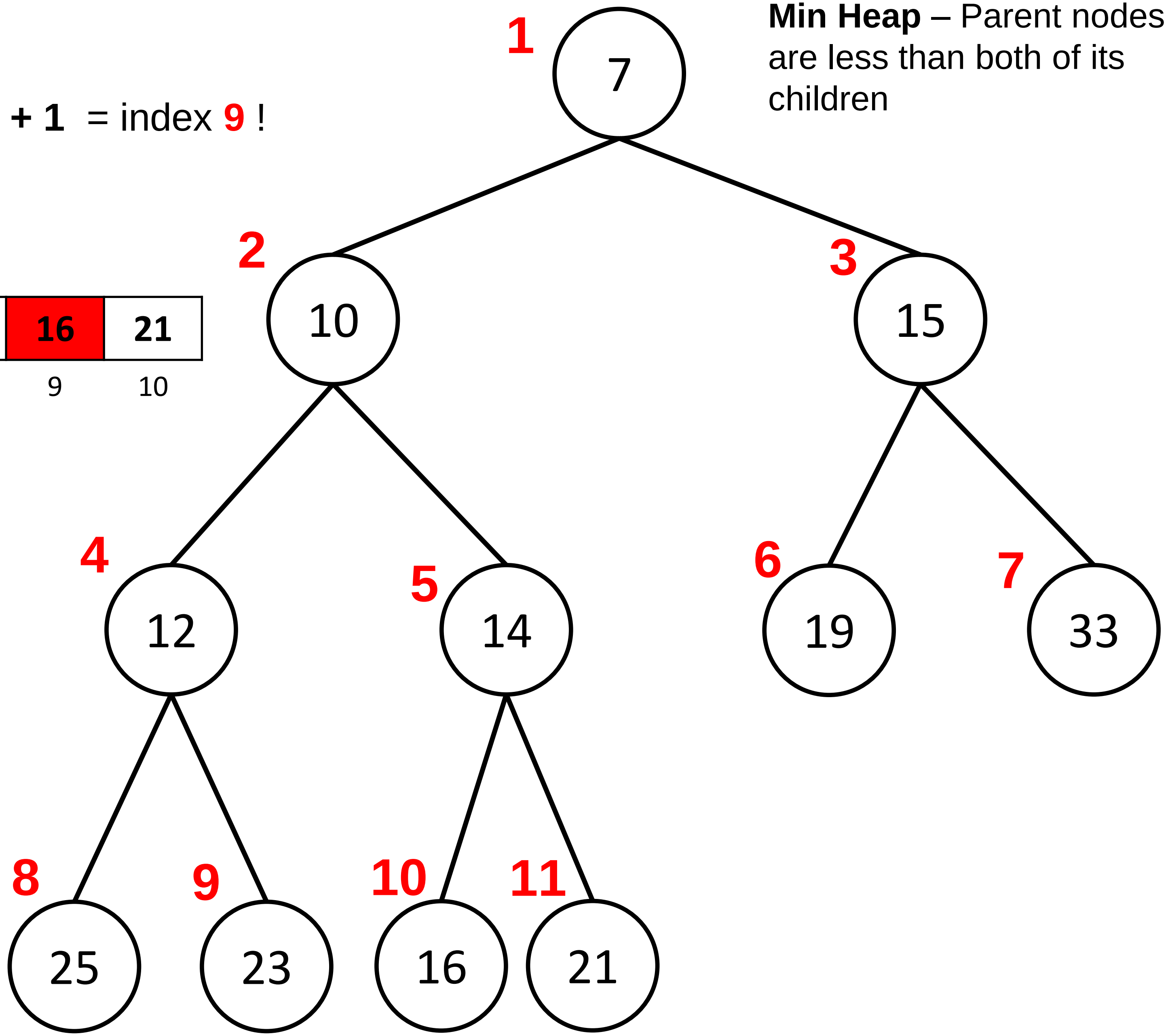
Heap Representation

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

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

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`**

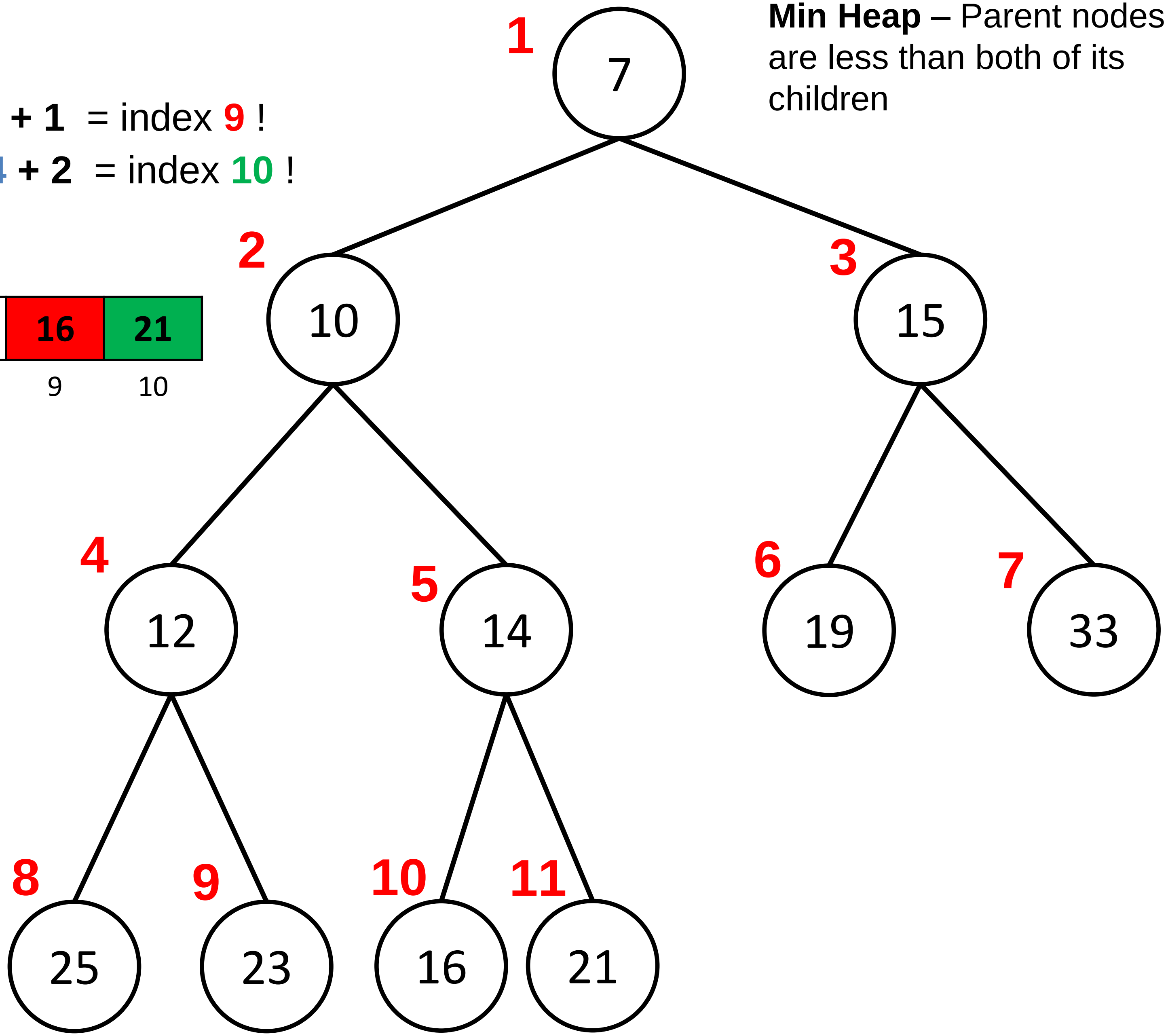
Its left child will be located at index:

`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



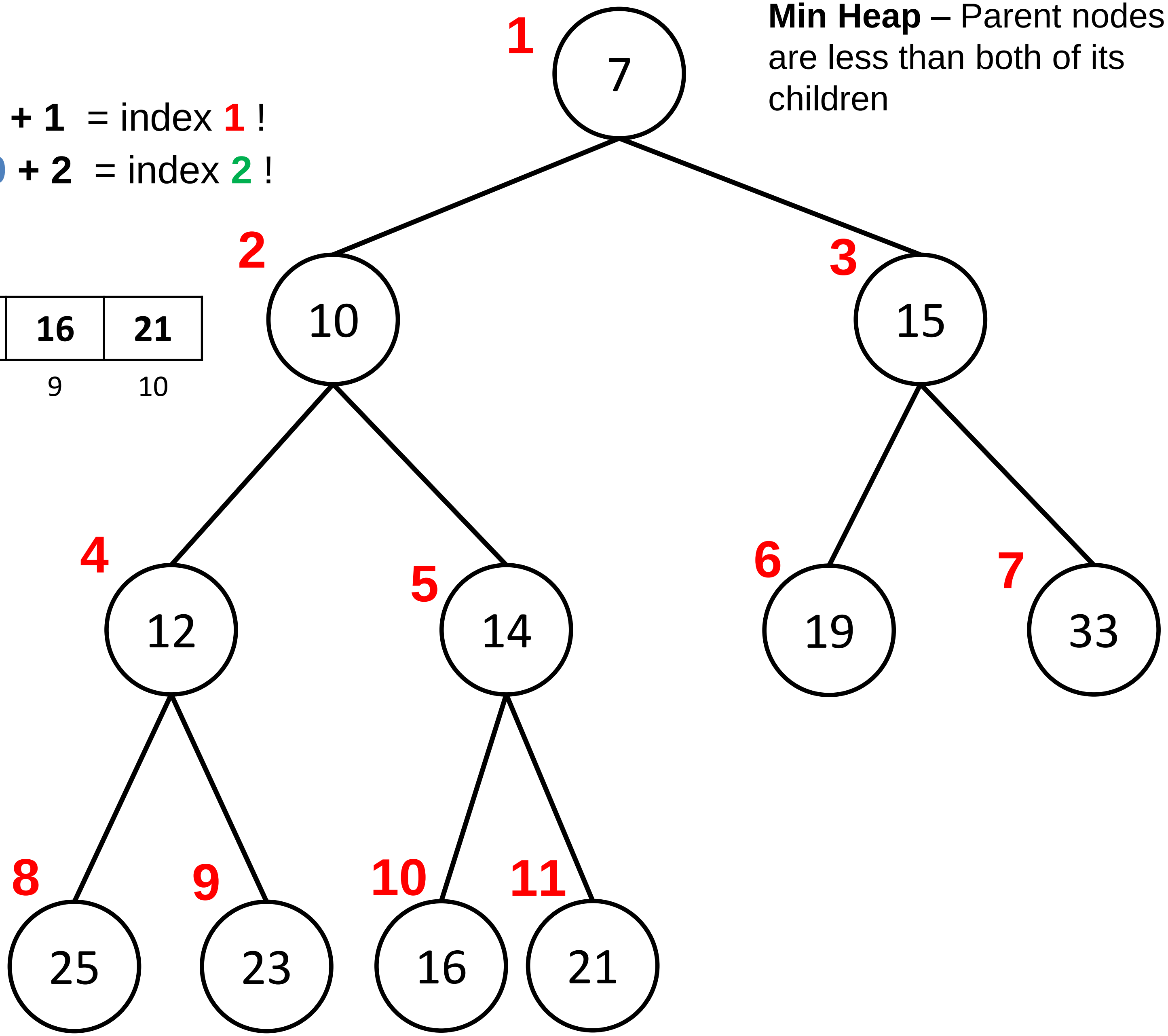
Heap Representation

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

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:

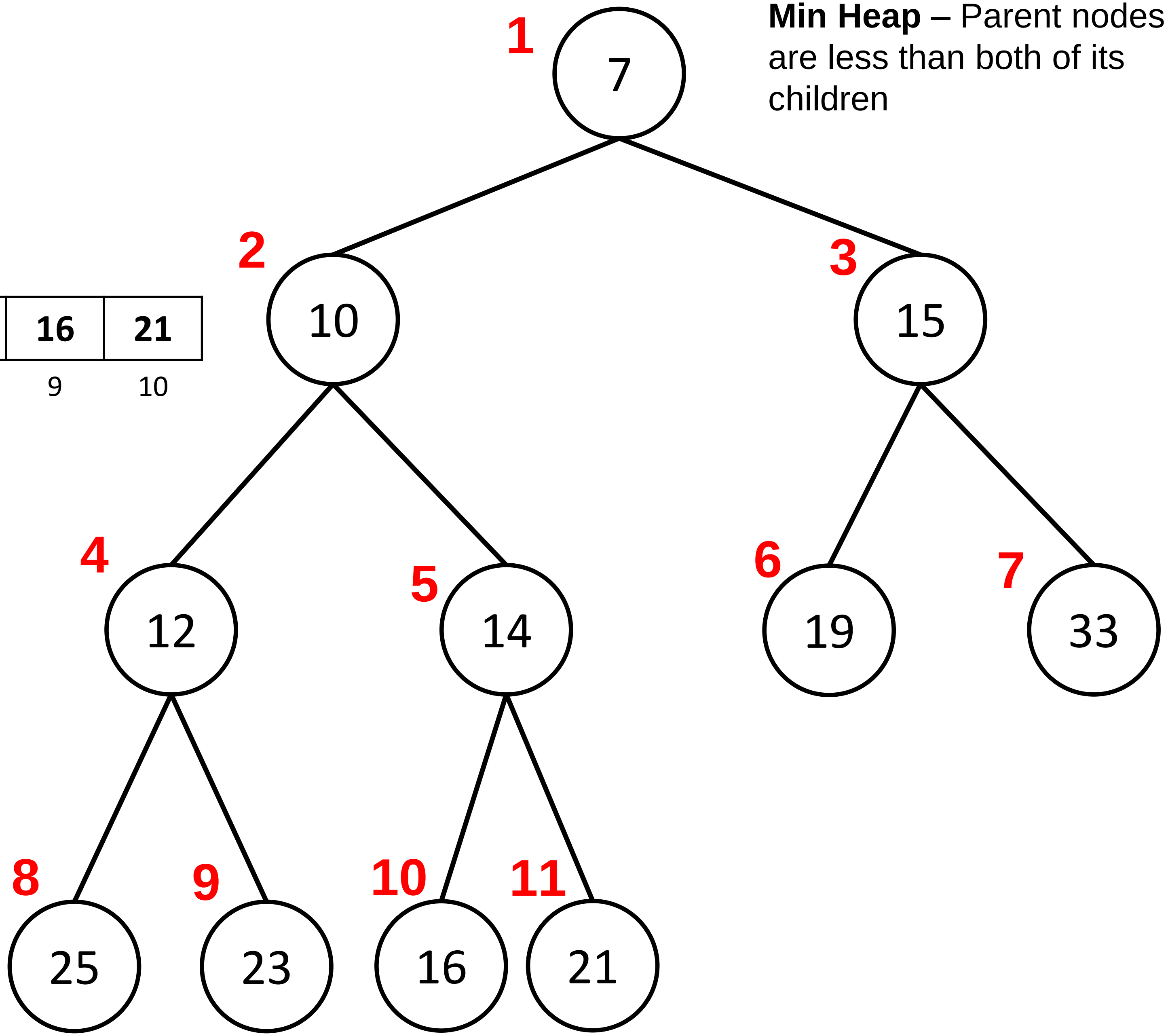
`2 * i + 2`

Heap Representation

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

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

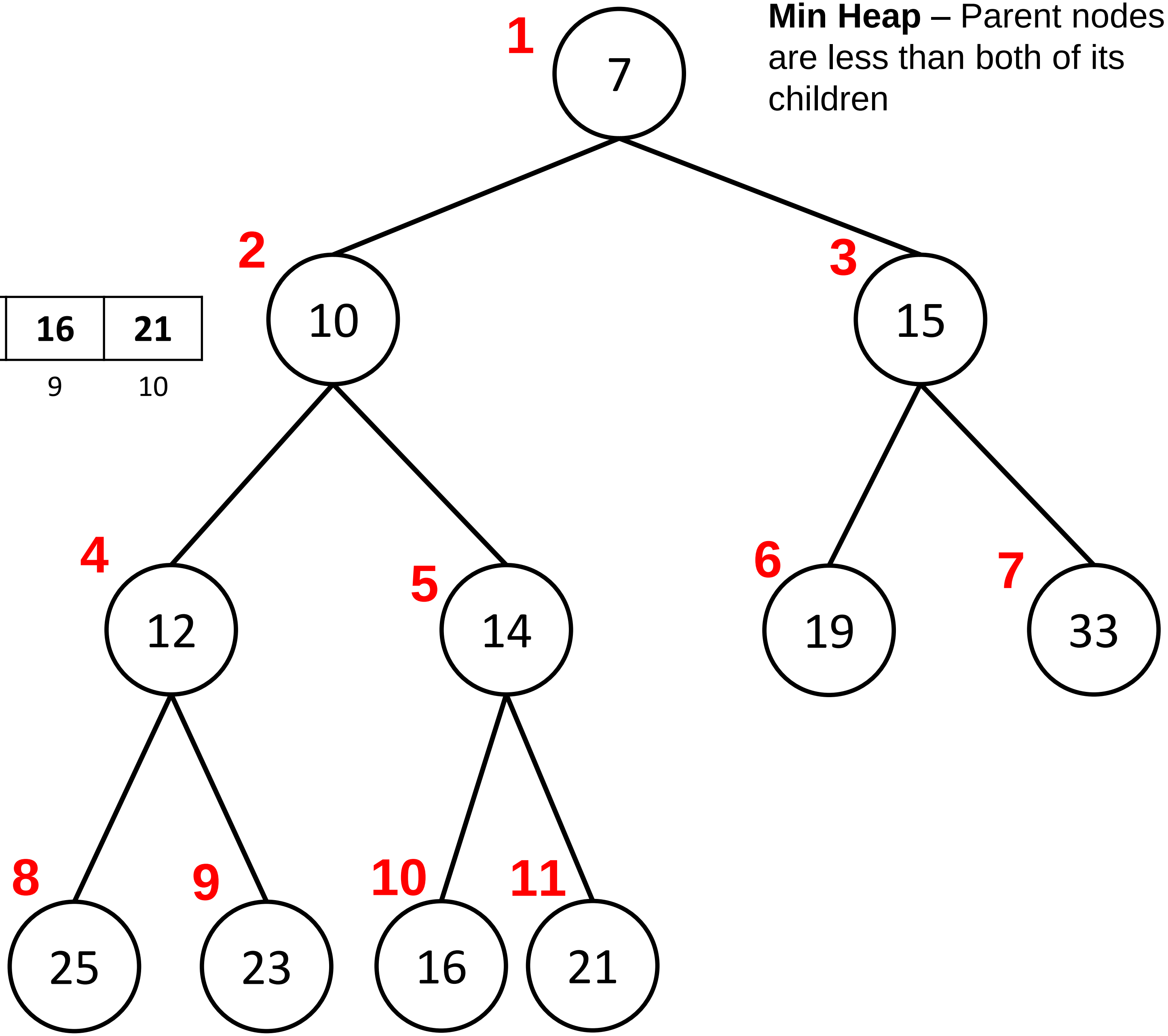
Given a spot in the array, how can we find its parent?

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

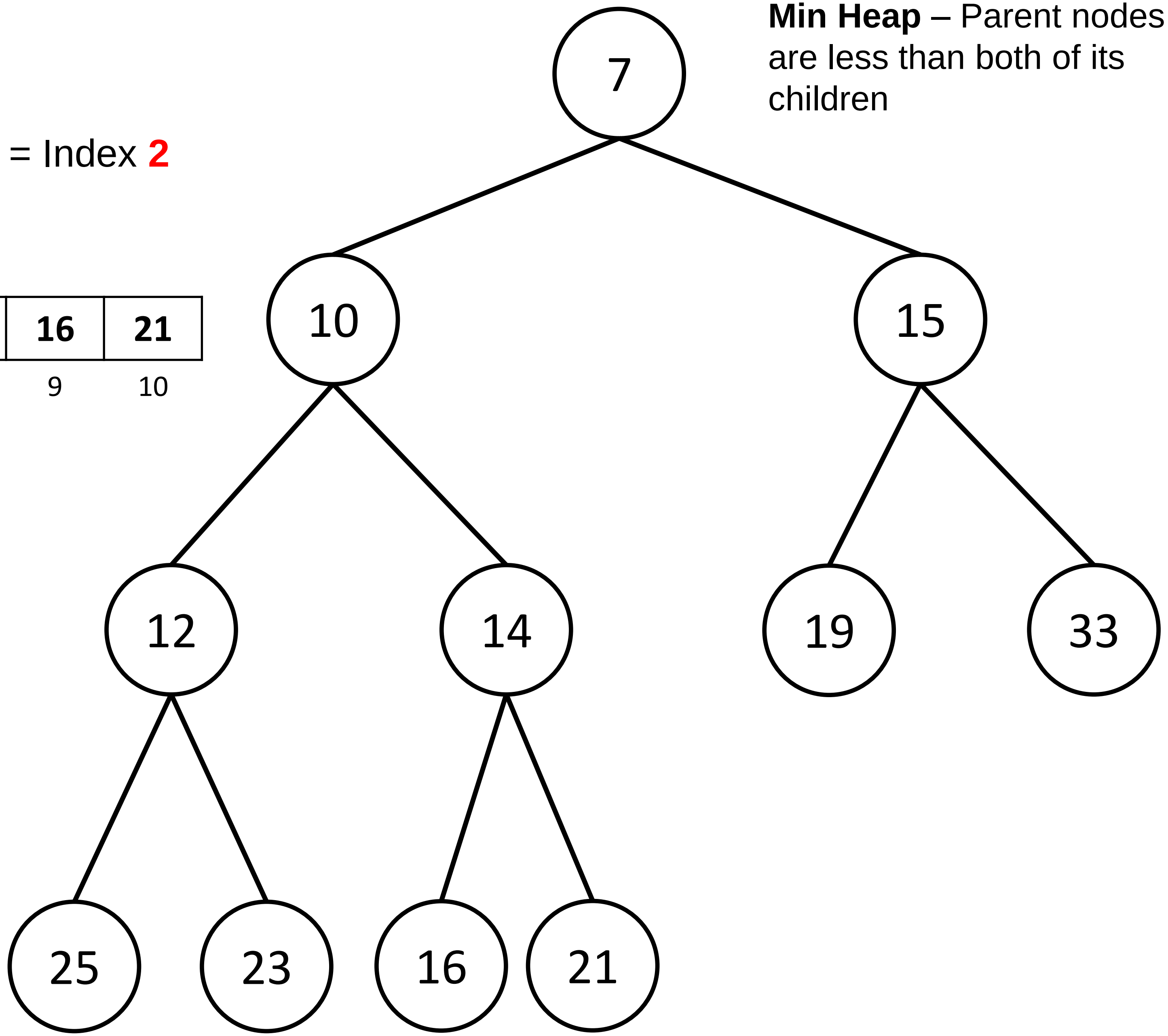
Given a spot in the array, how can we find its parent?

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

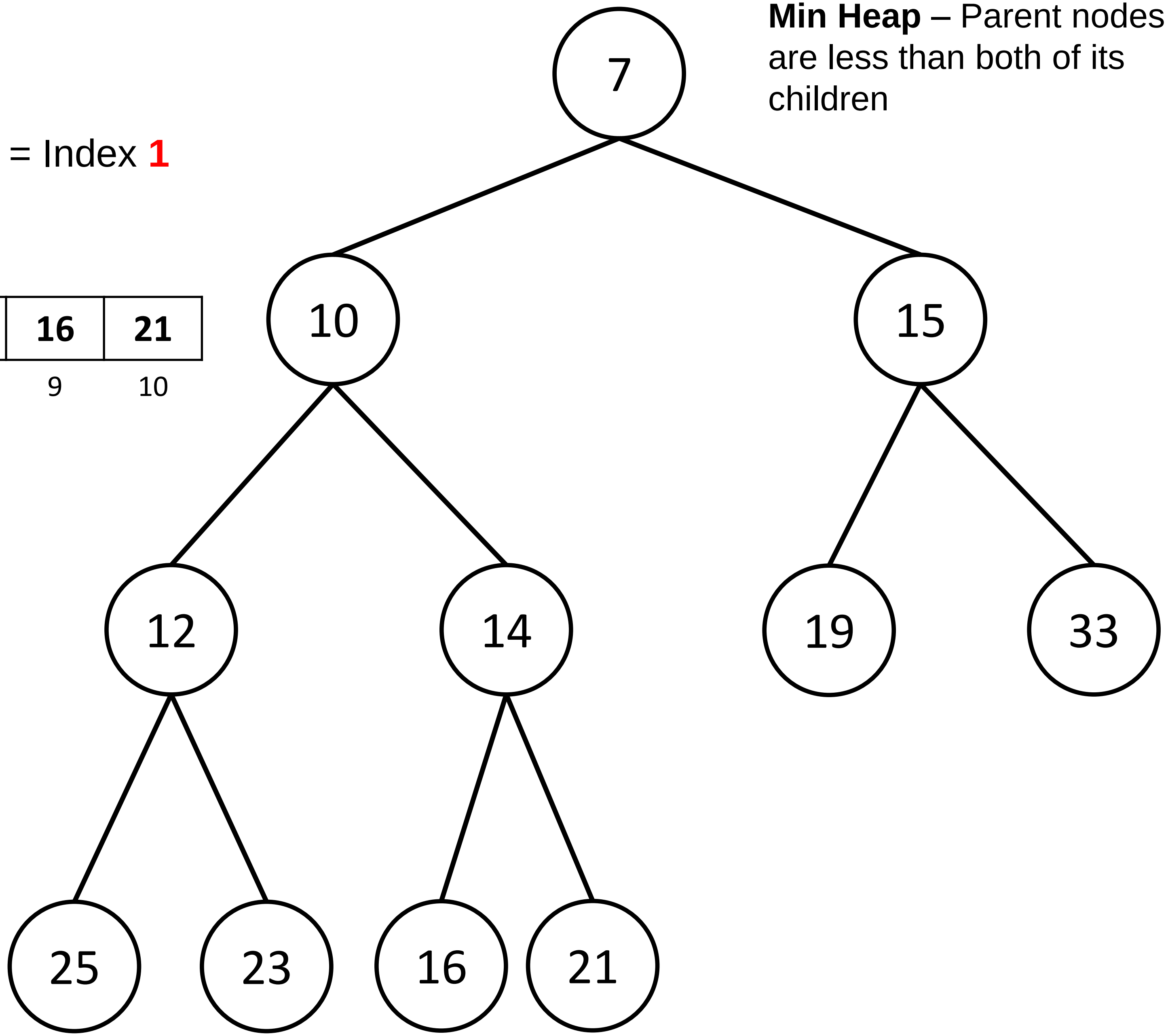
Given a spot in the array, how can we find its parent?

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)

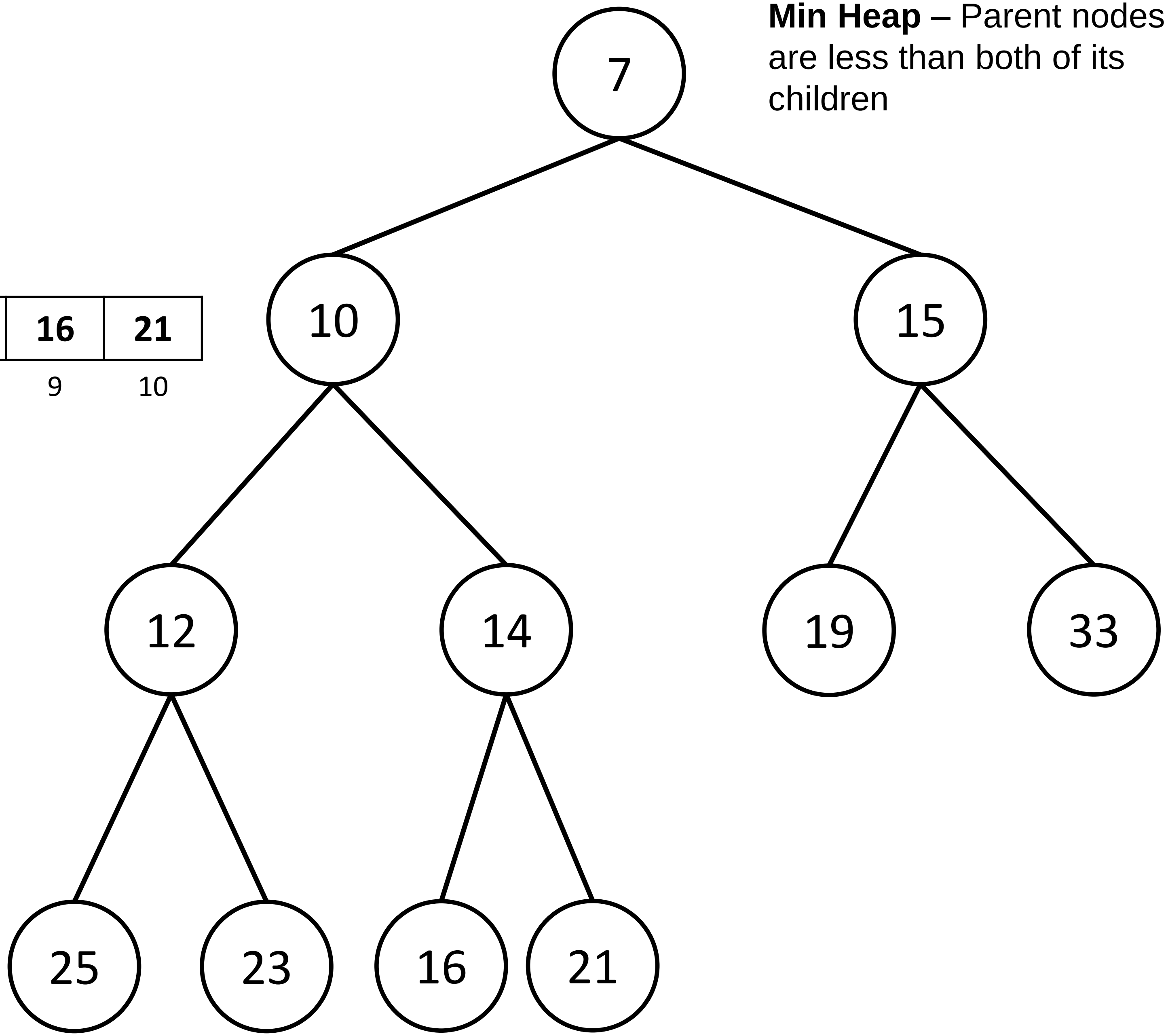


Heap Representation

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`

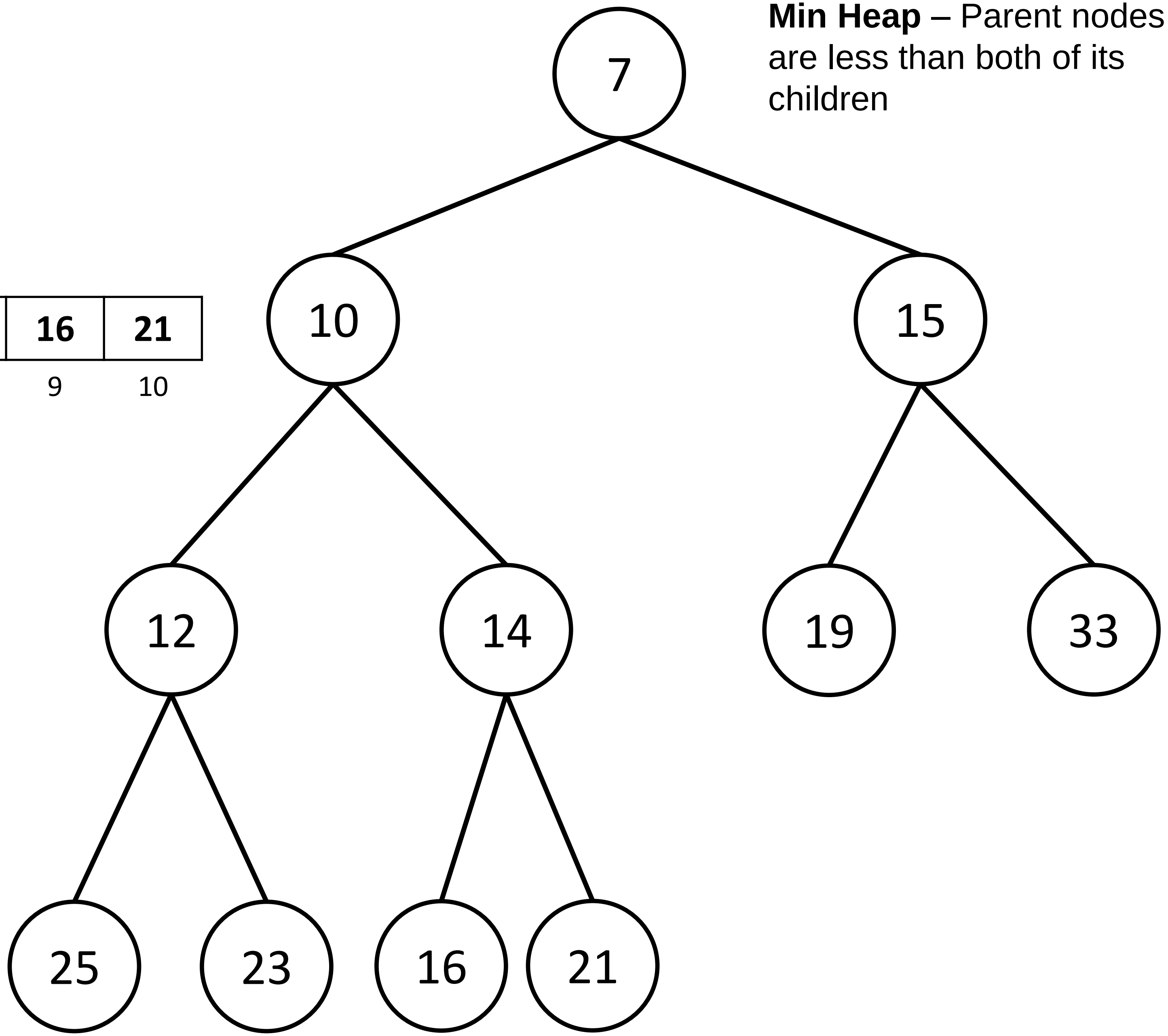
Heap Representation

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

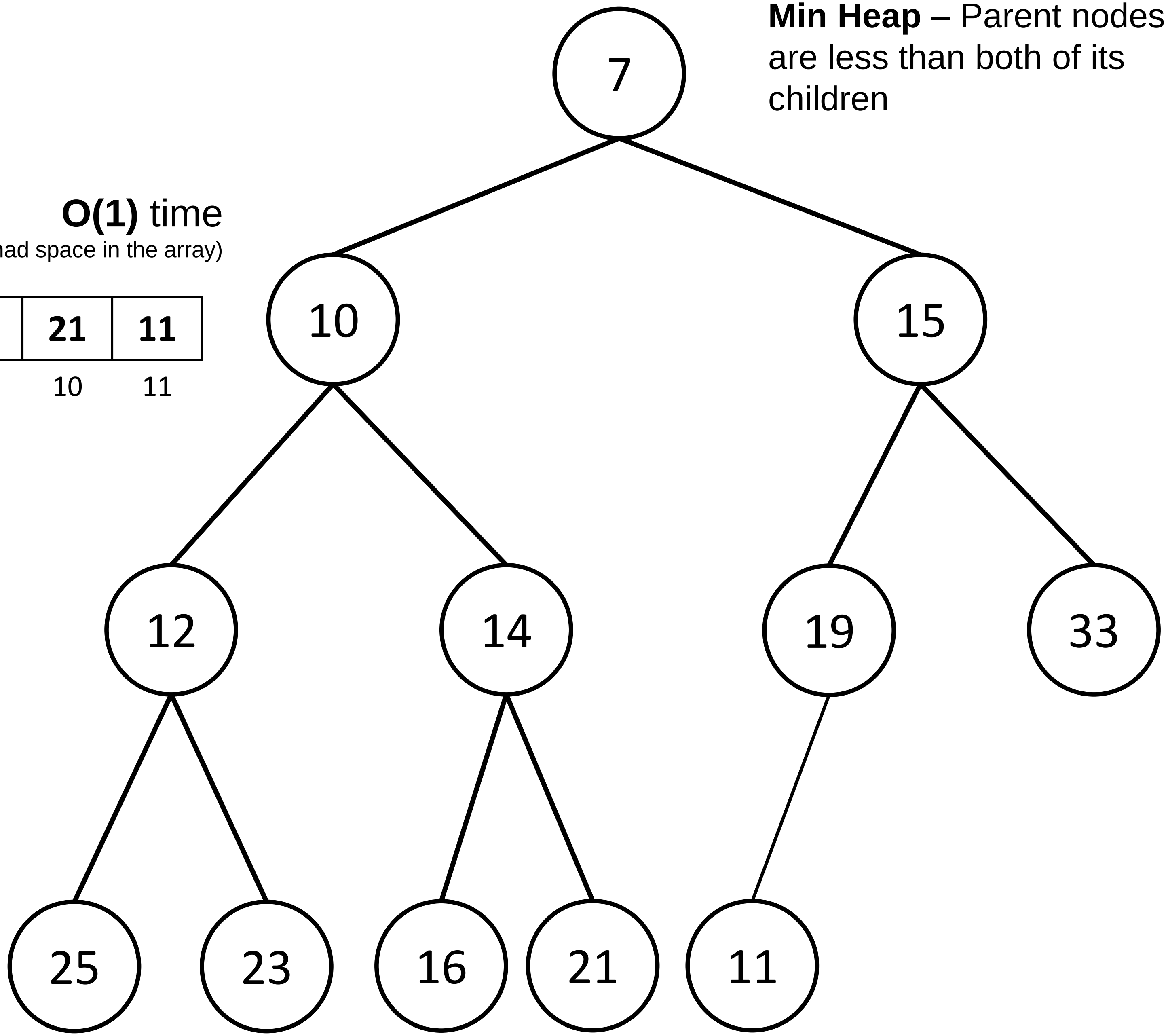
Min Heap – Parent nodes are less than both of its children

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

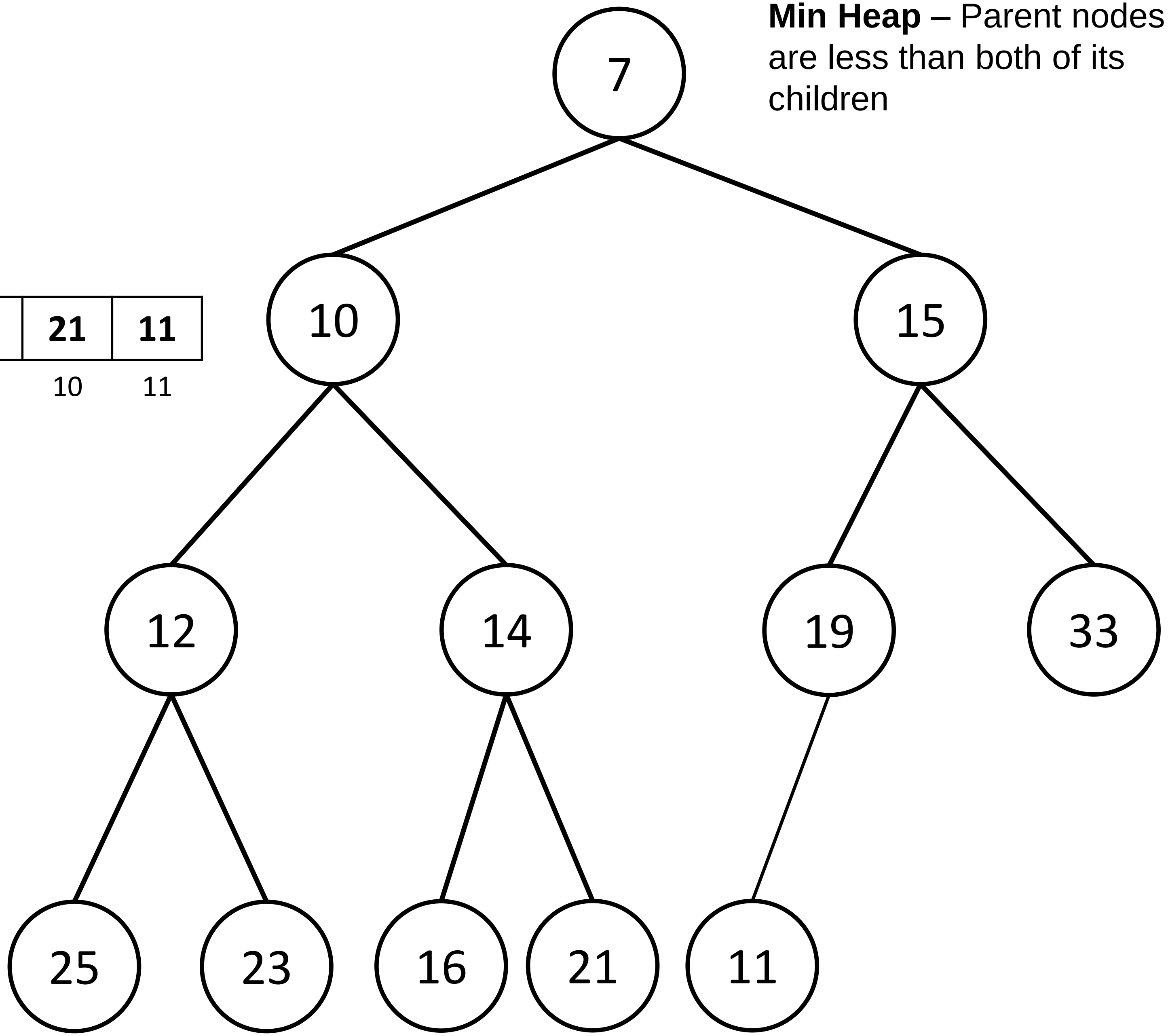
Min Heap – Parent nodes are less than both of its children

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`

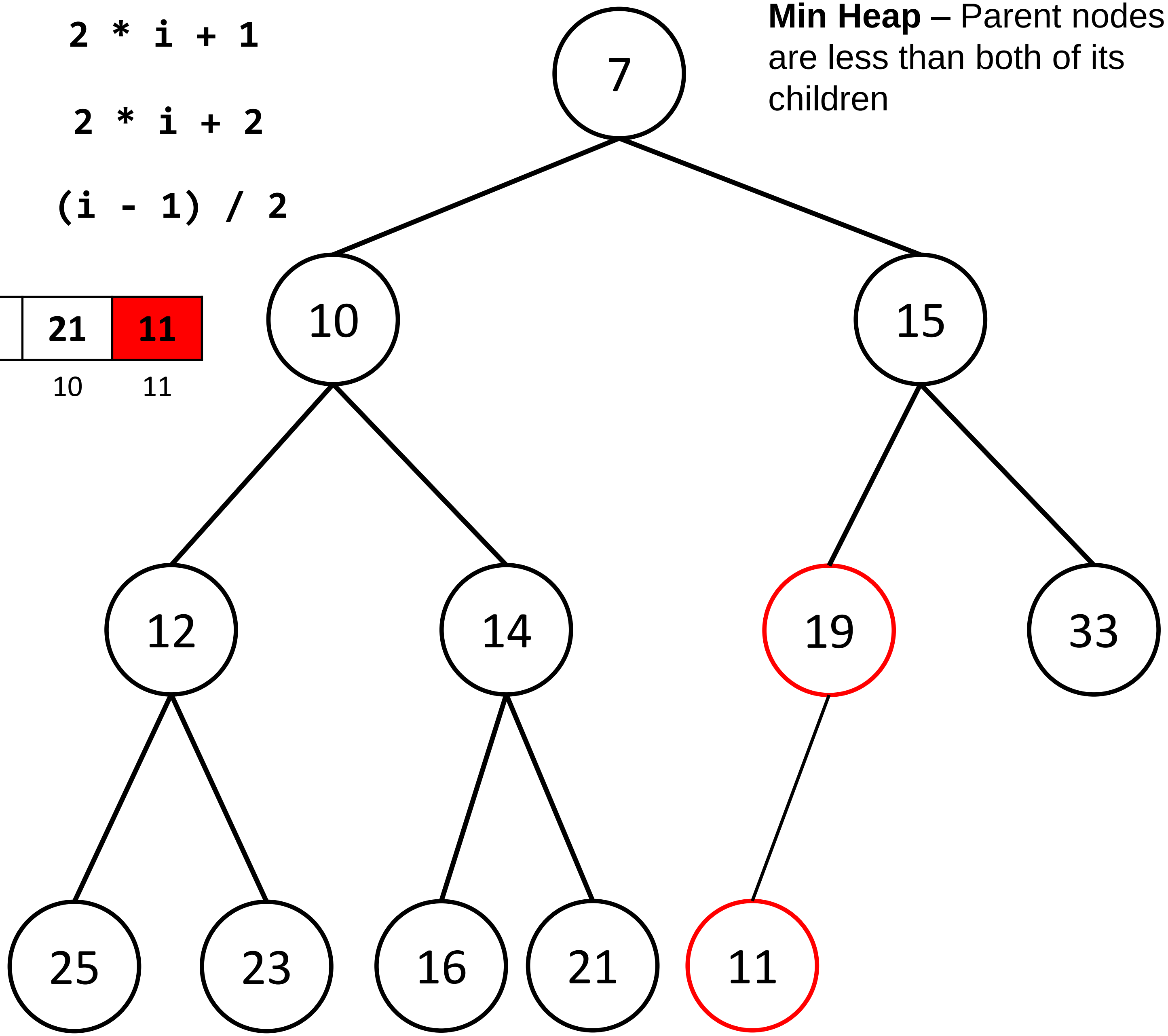
Right Child `2 * i + 2`

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

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



`insert(11);`

Time to Heapify Up!

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

Heap Representation

Left Child `2 * i + 1`

Right Child `2 * i + 2`

Parent `(i - 1) / 2`

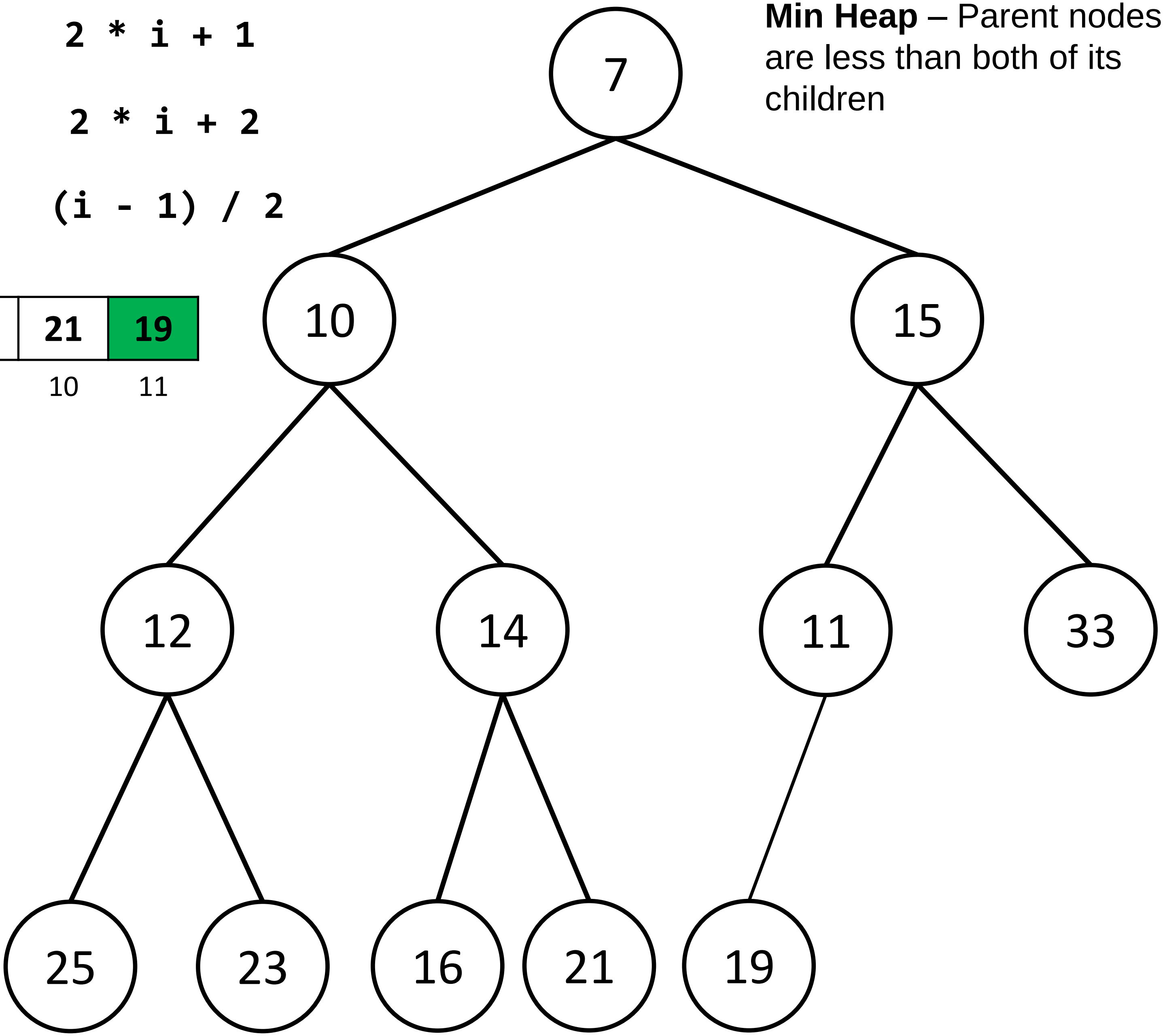
Min Heap – Parent nodes are less than both of its children

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!



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

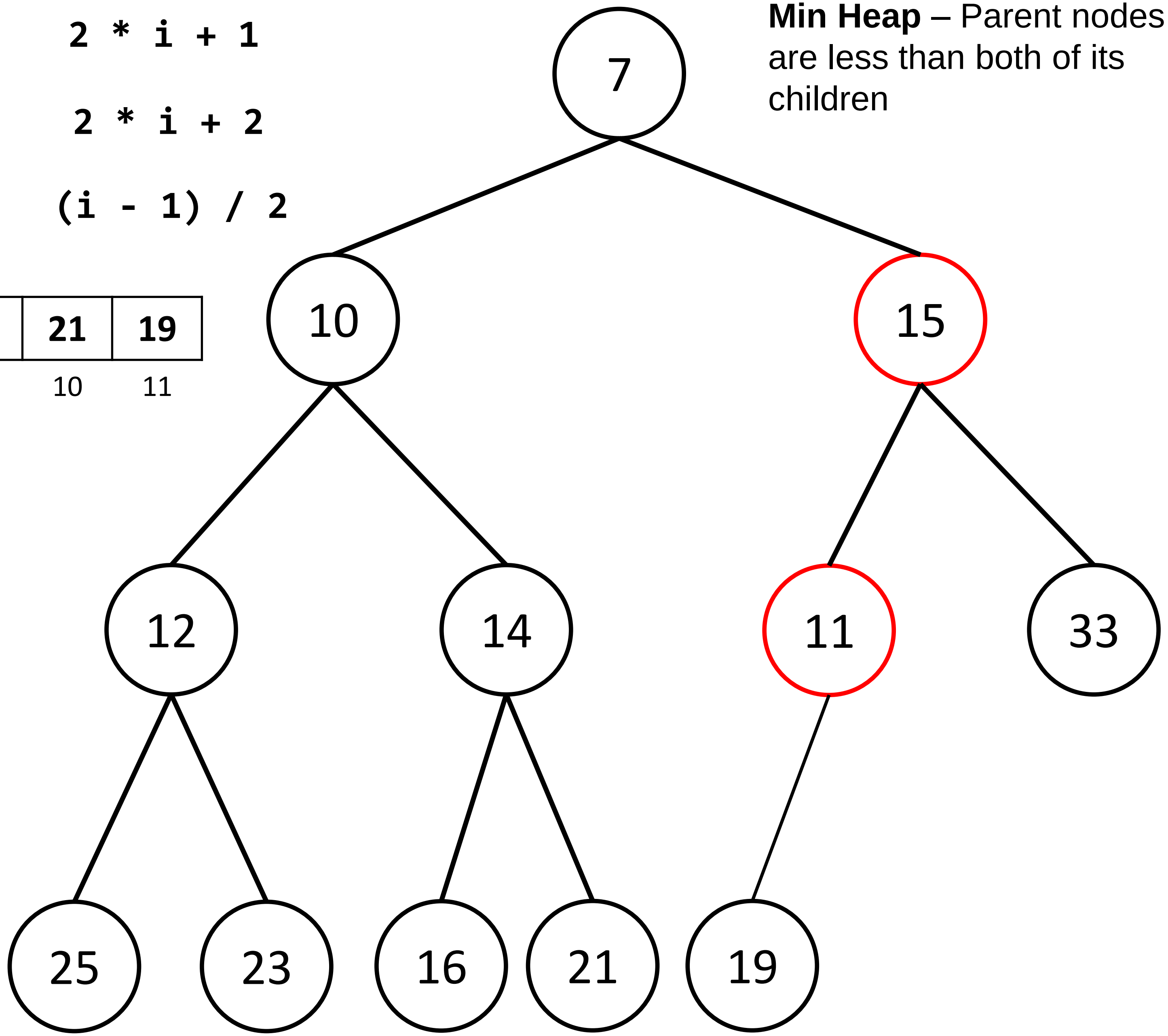
Array

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`

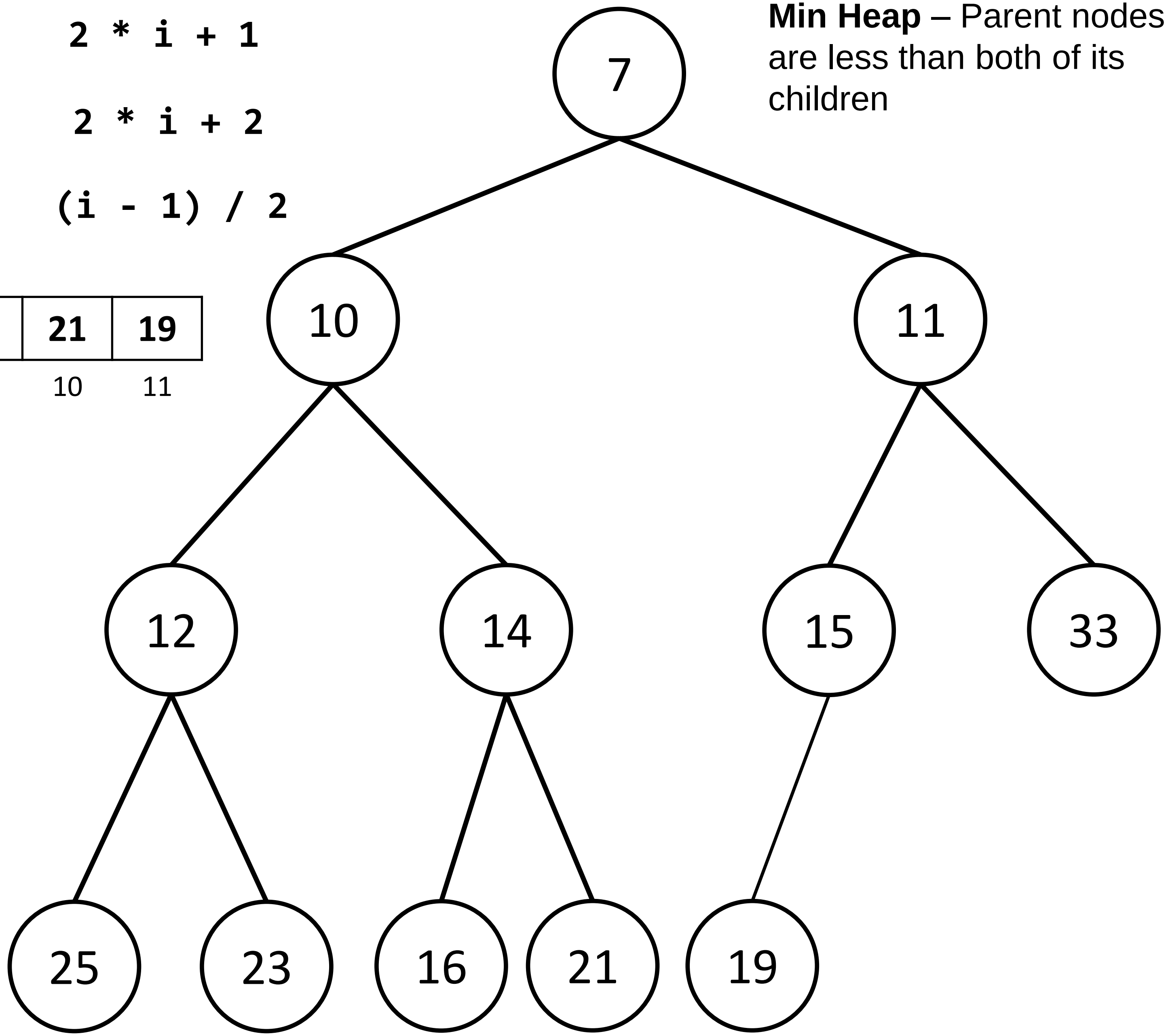
Right Child `2 * i + 2`

Parent `(i - 1) / 2`

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



`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`

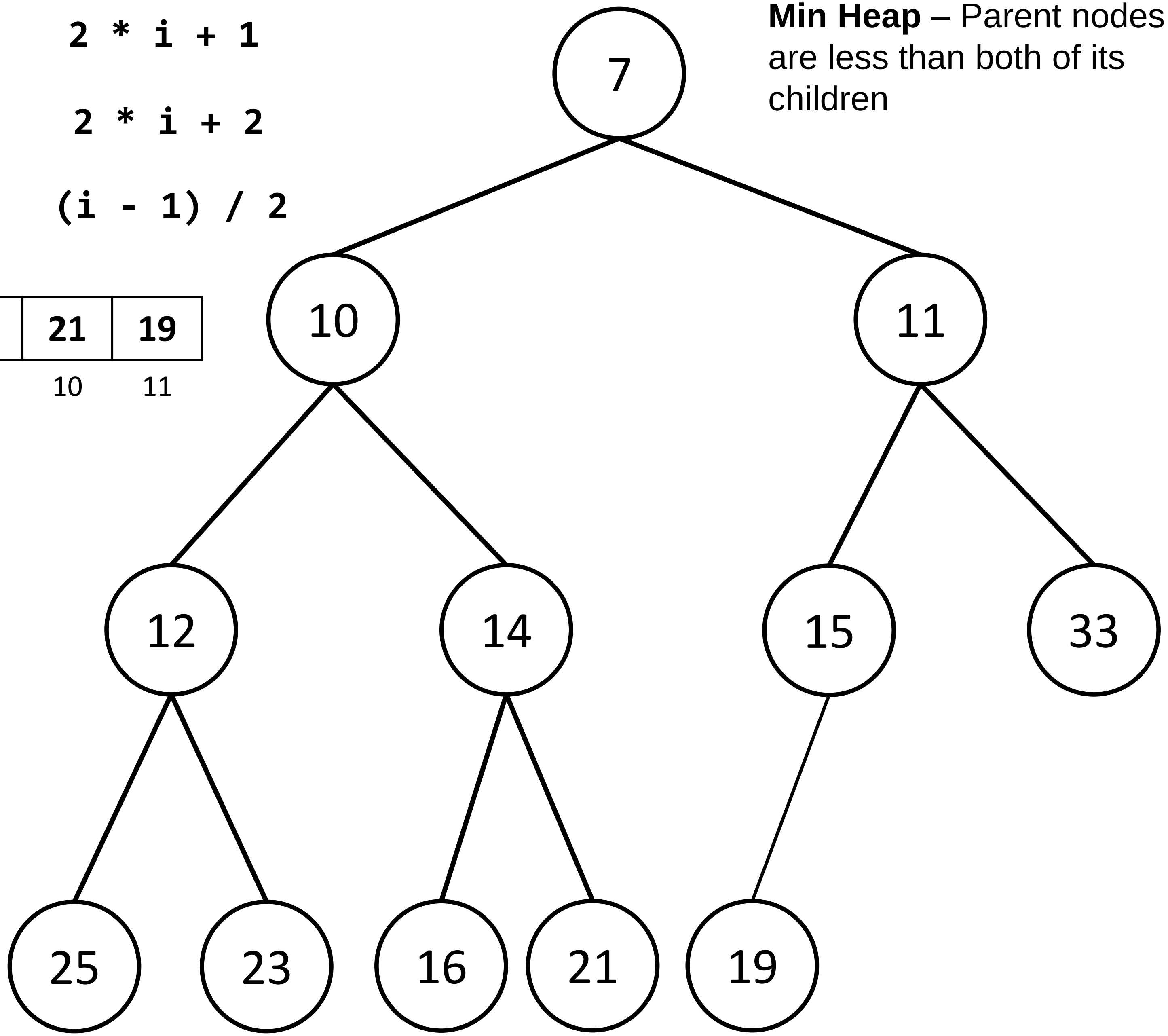
Array

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!

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



Heap Representation

Left Child `2 * i + 1`

Right Child `2 * i + 2`

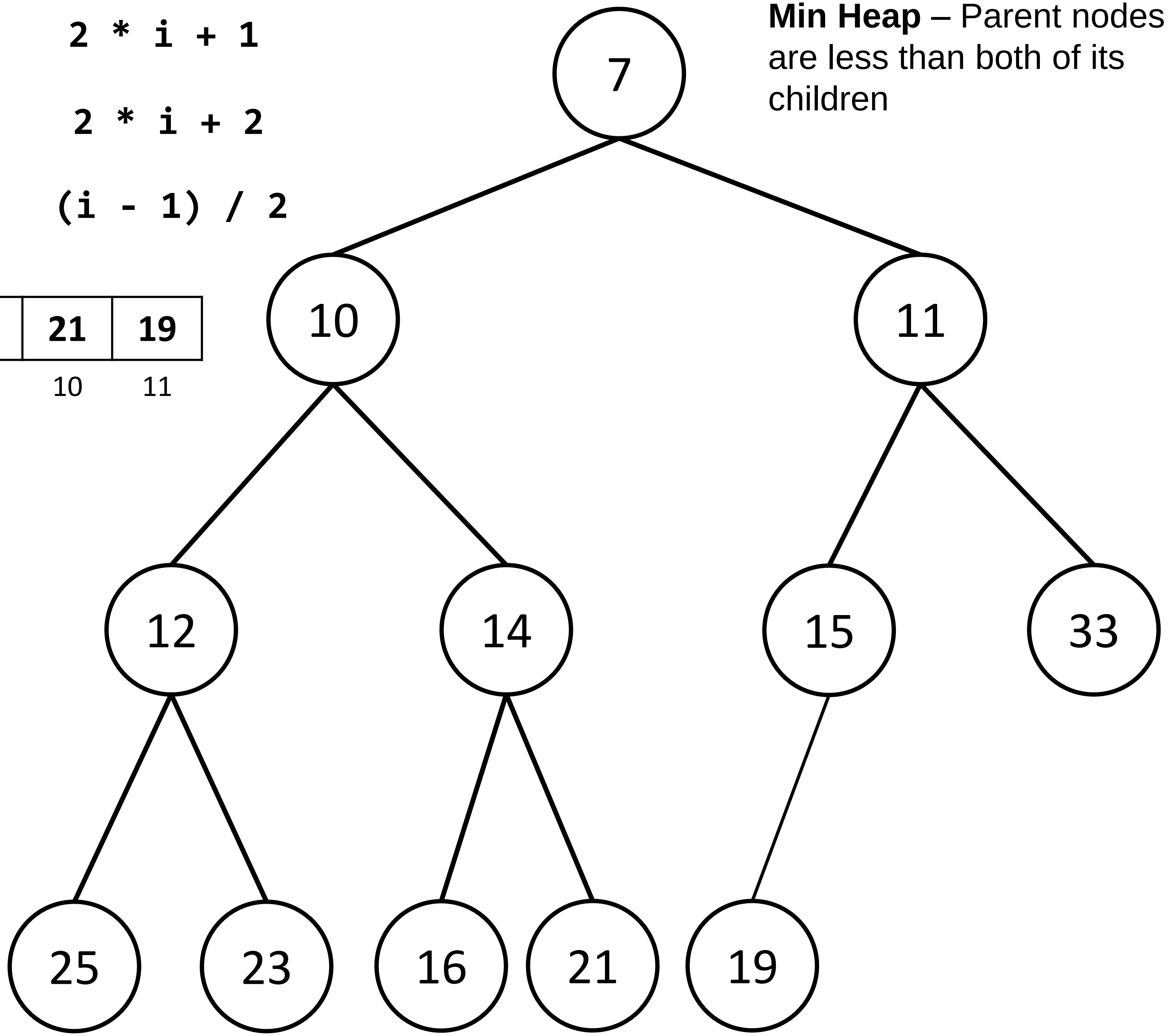
Parent `(i - 1) / 2`

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`

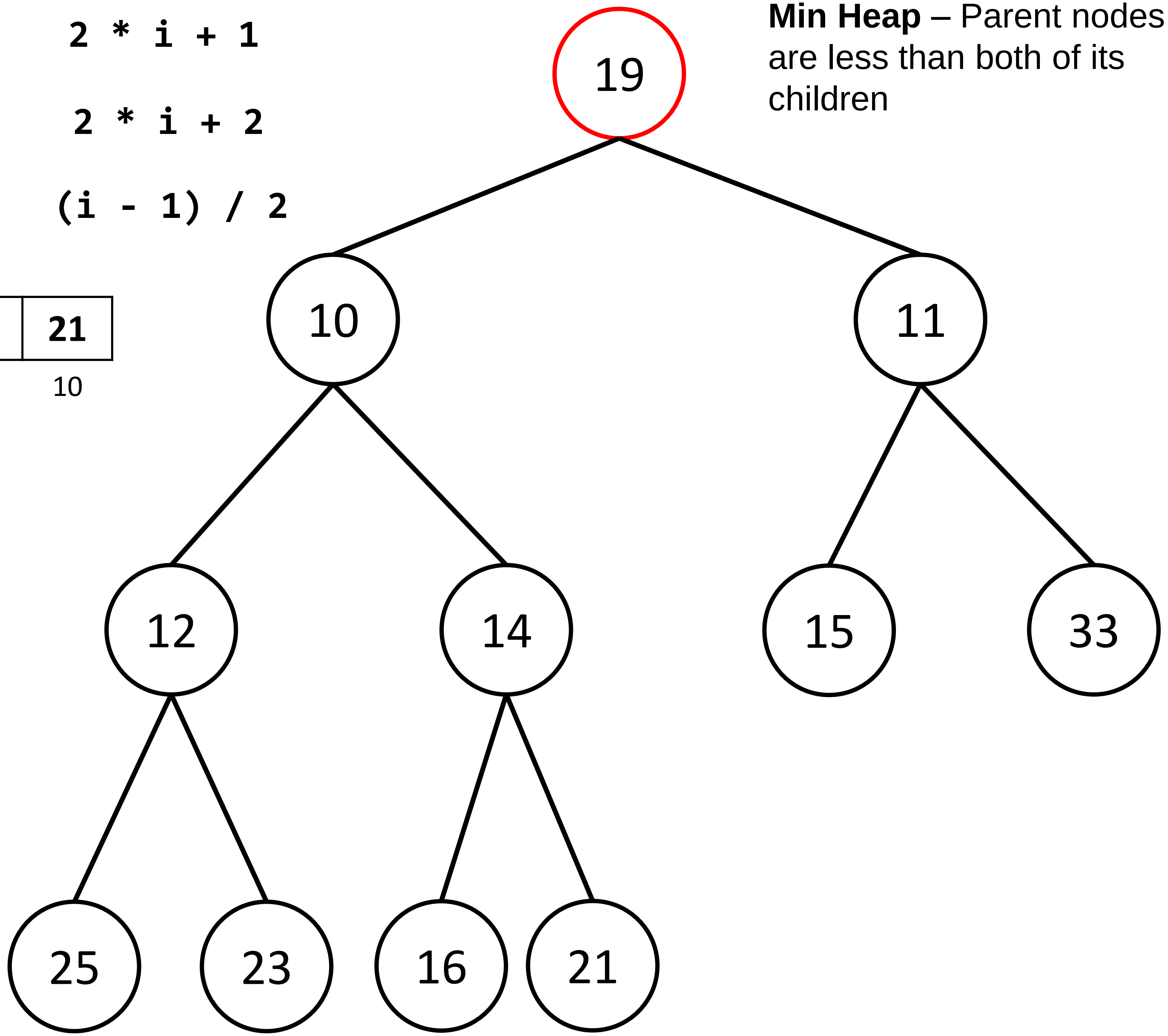
Parent `(i - 1) / 2`

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

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();
```



Heap Representation

Left Child `2 * i + 1`

Right Child `2 * i + 2`

Parent `(i - 1) / 2`

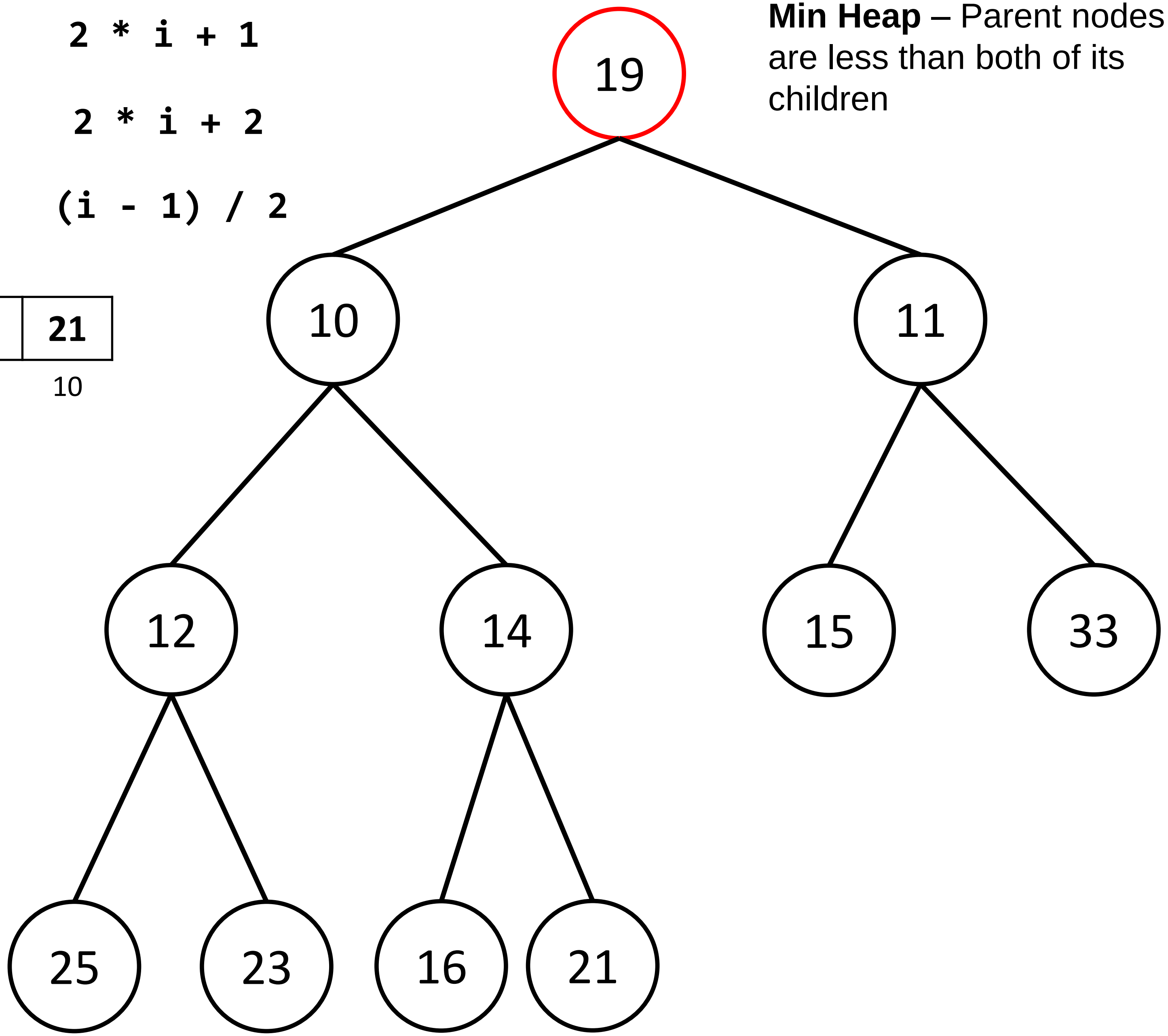
Array

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

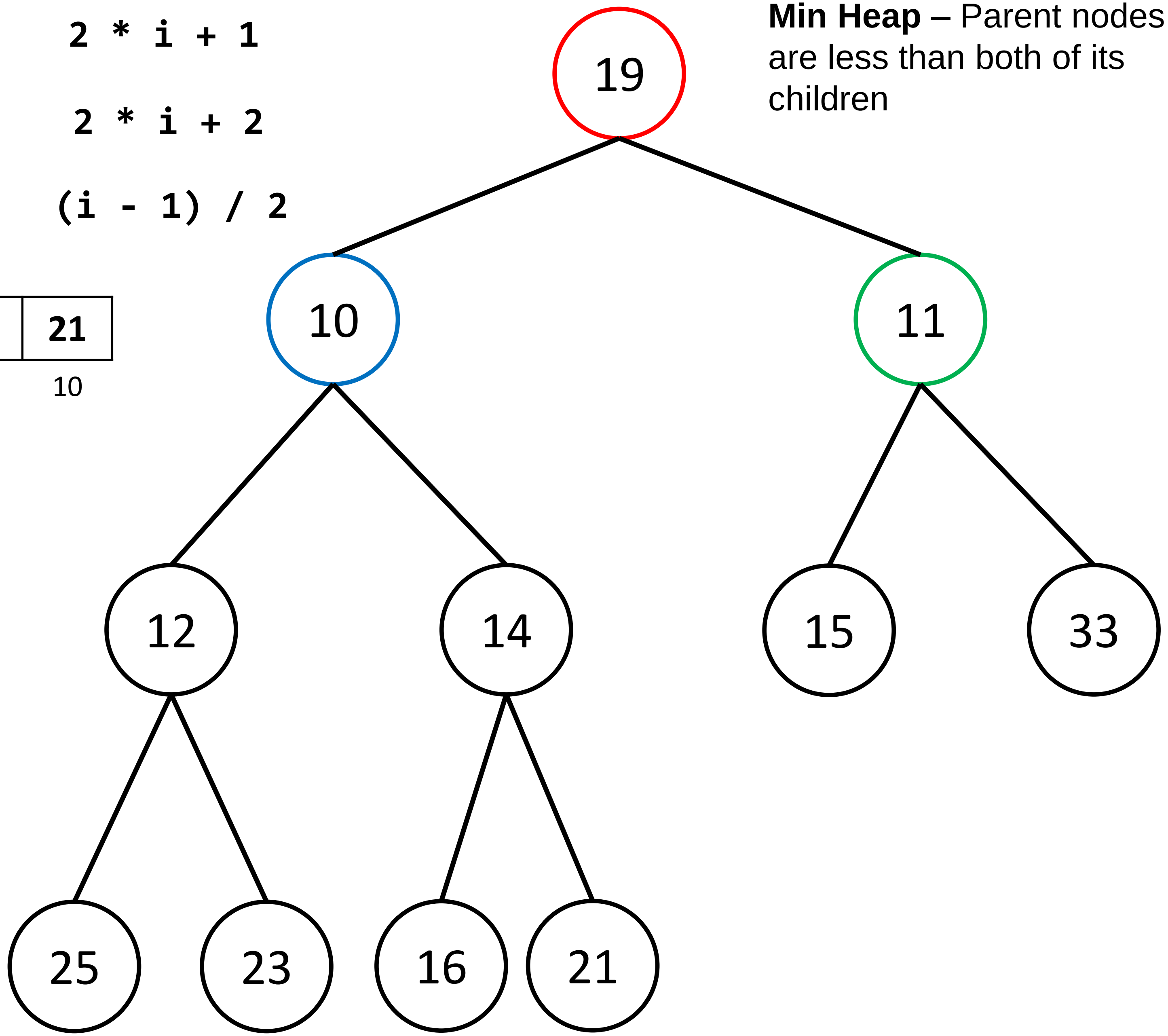
```
poll();
```

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`

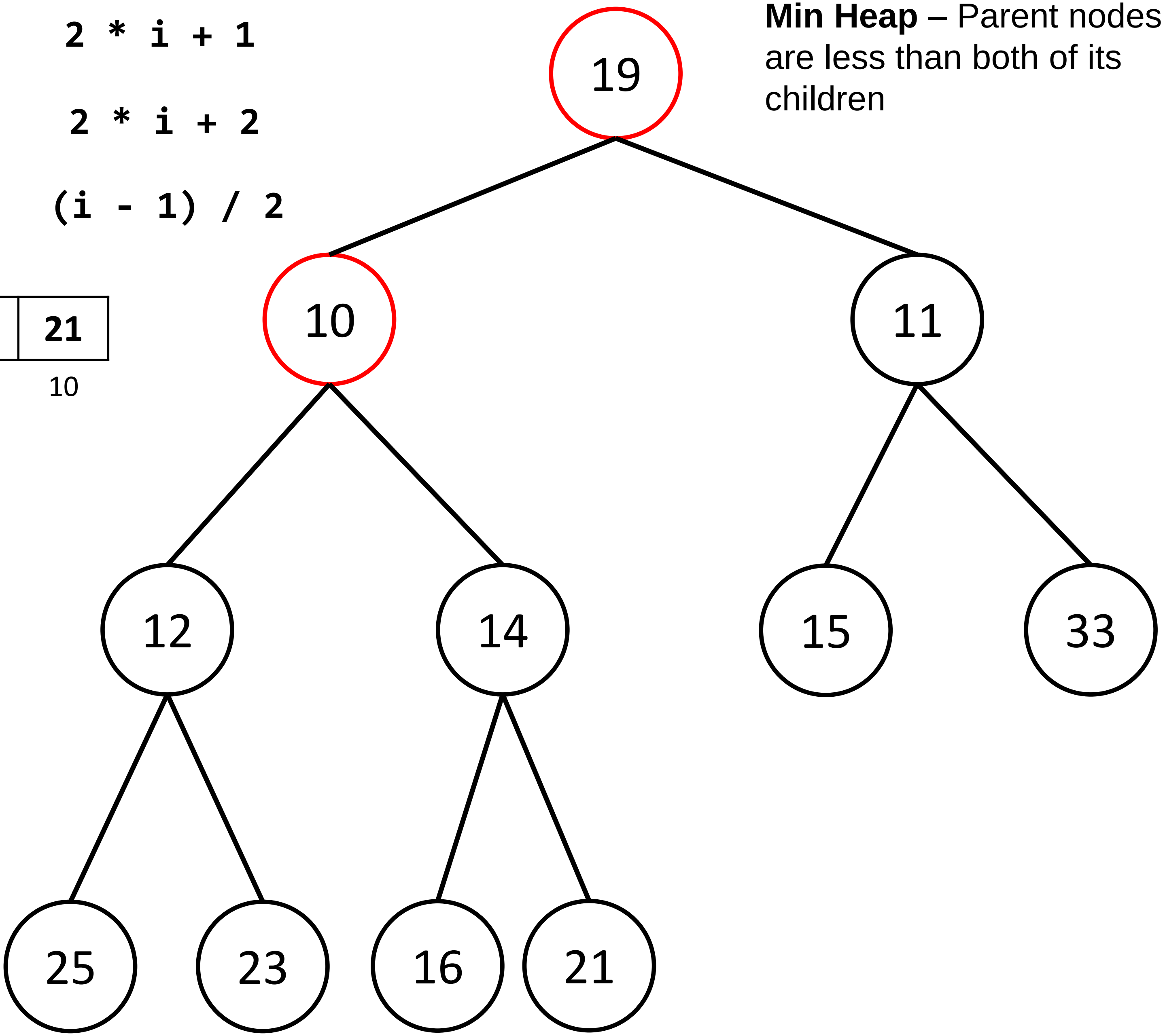
Right Child `2 * i + 2`

Parent `(i - 1) / 2`

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!

Heap Representation

Left Child `2 * i + 1`

Right Child `2 * i + 2`

Parent `(i - 1) / 2`

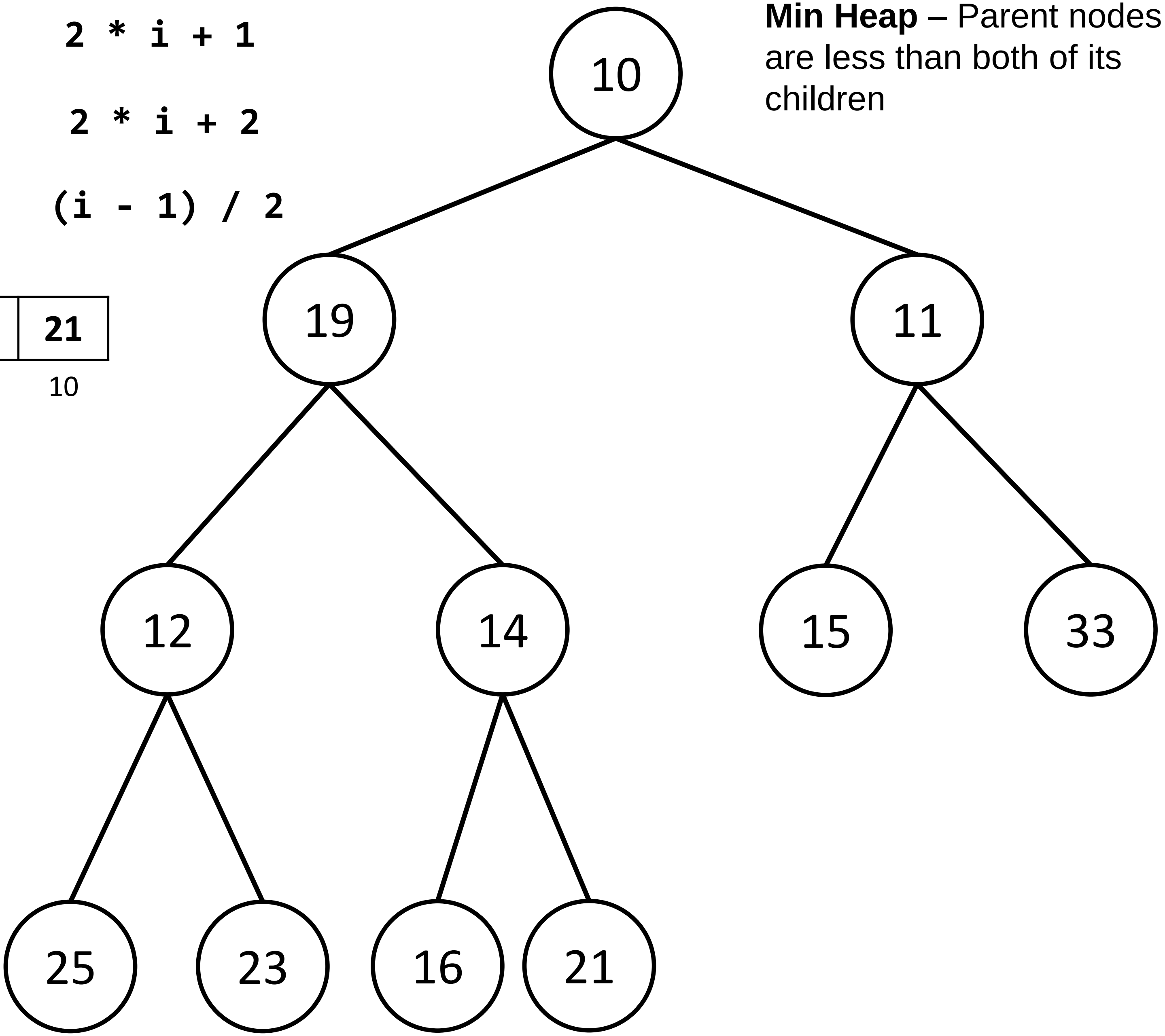
Min Heap – Parent nodes are less than both of its children

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`

Array

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

```
poll();
```

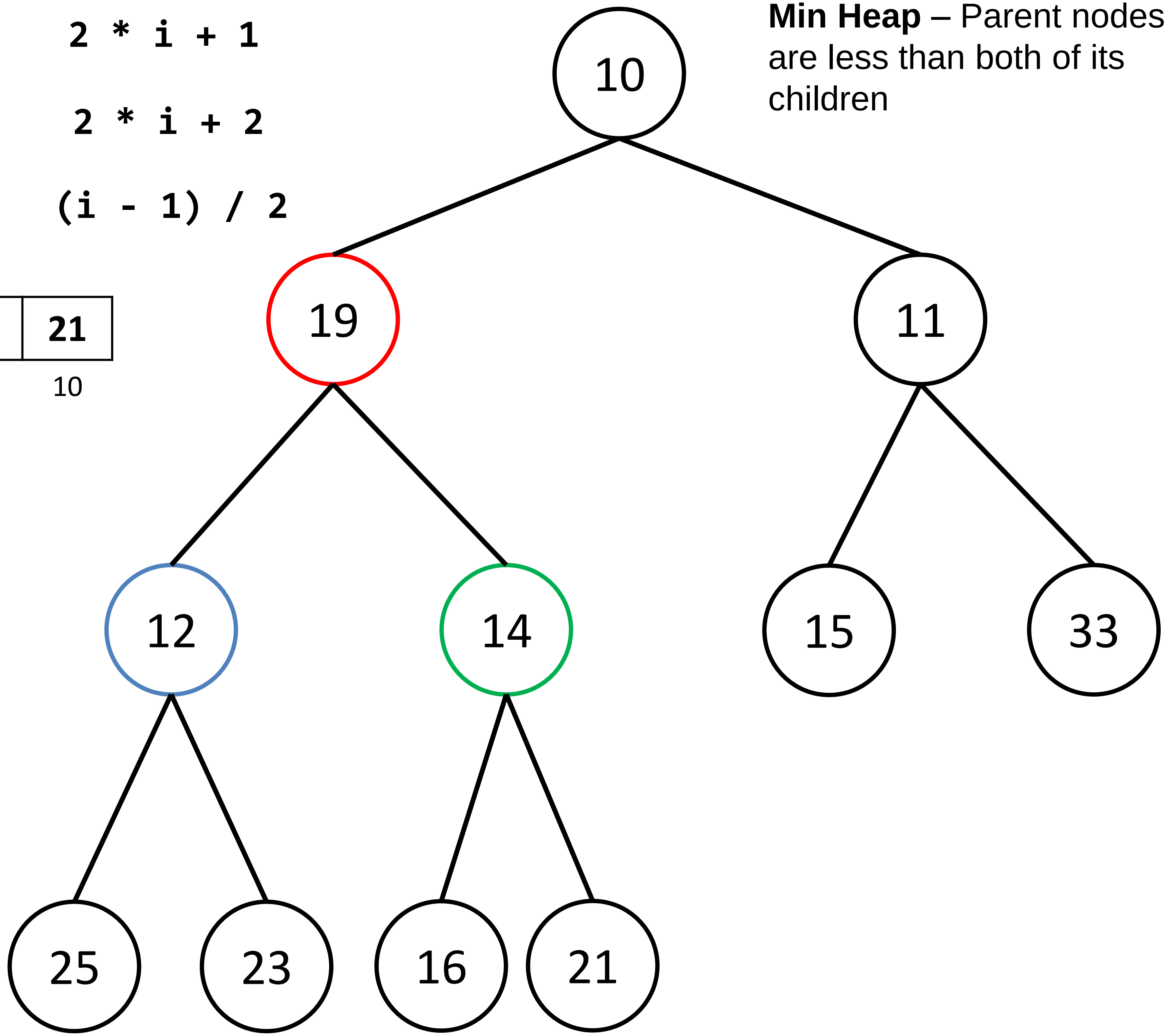
Time to Heapify down!

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`

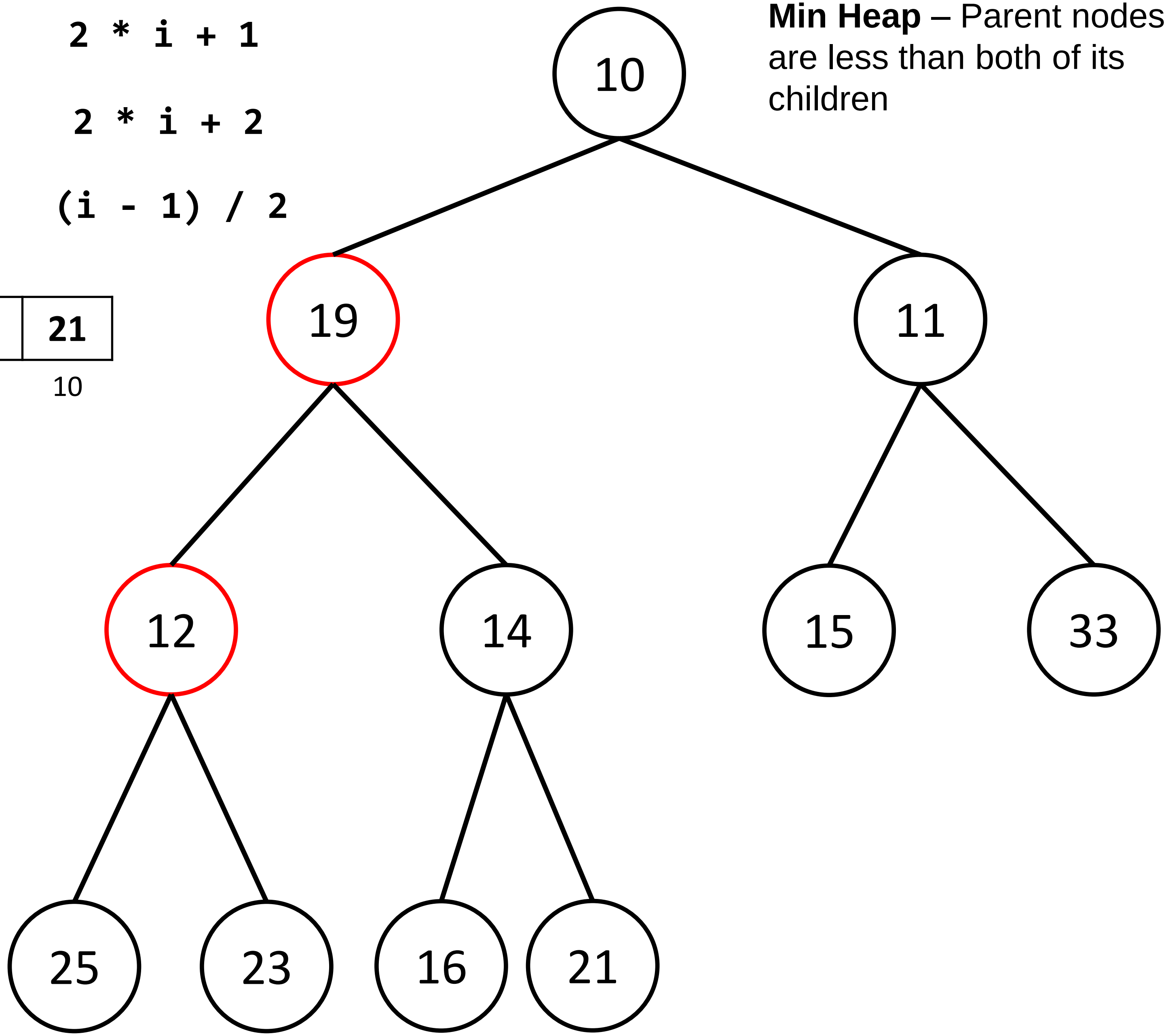
Min Heap – Parent nodes are less than both of its children

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`

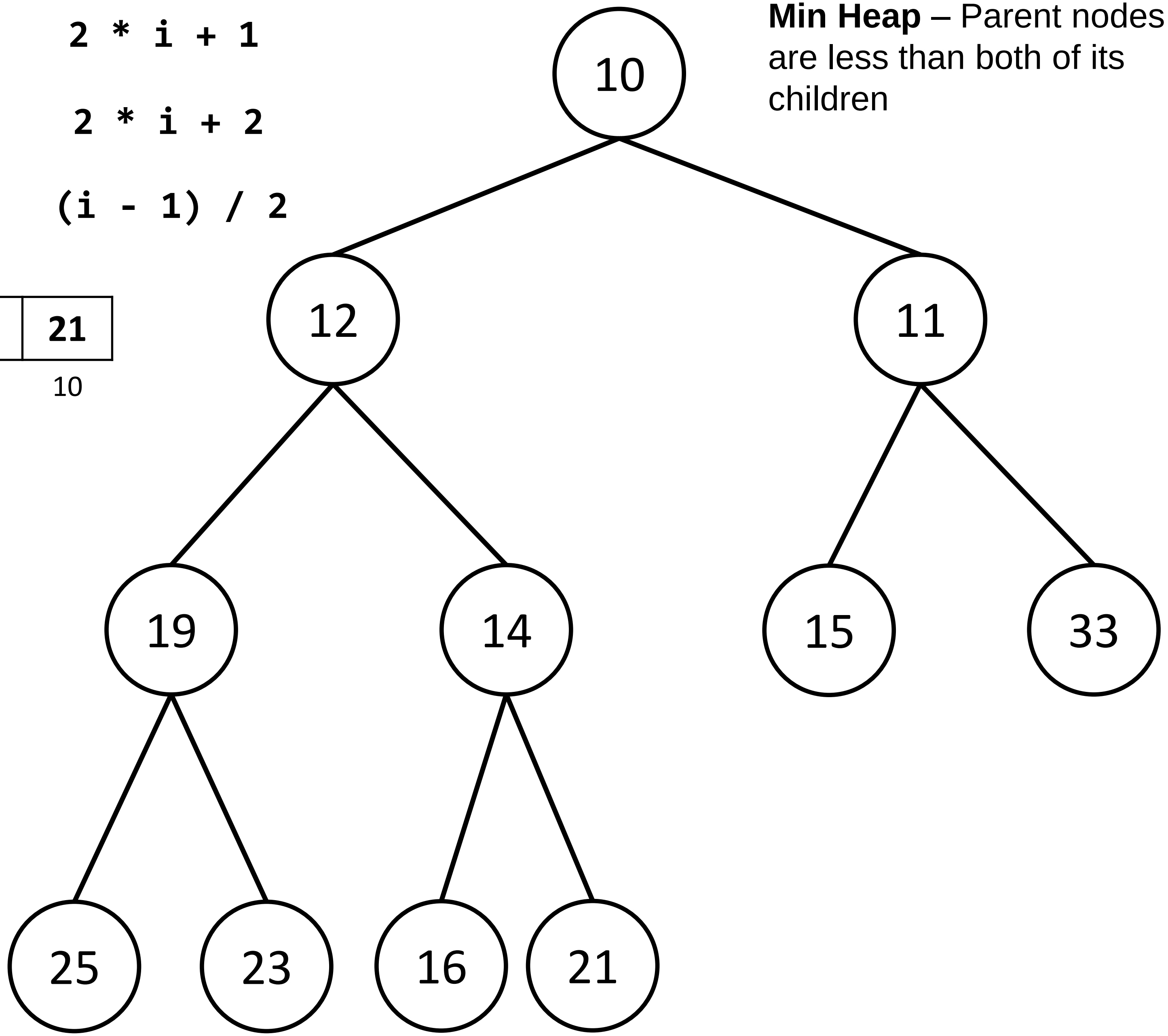
Min Heap – Parent nodes are less than both of its children

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`

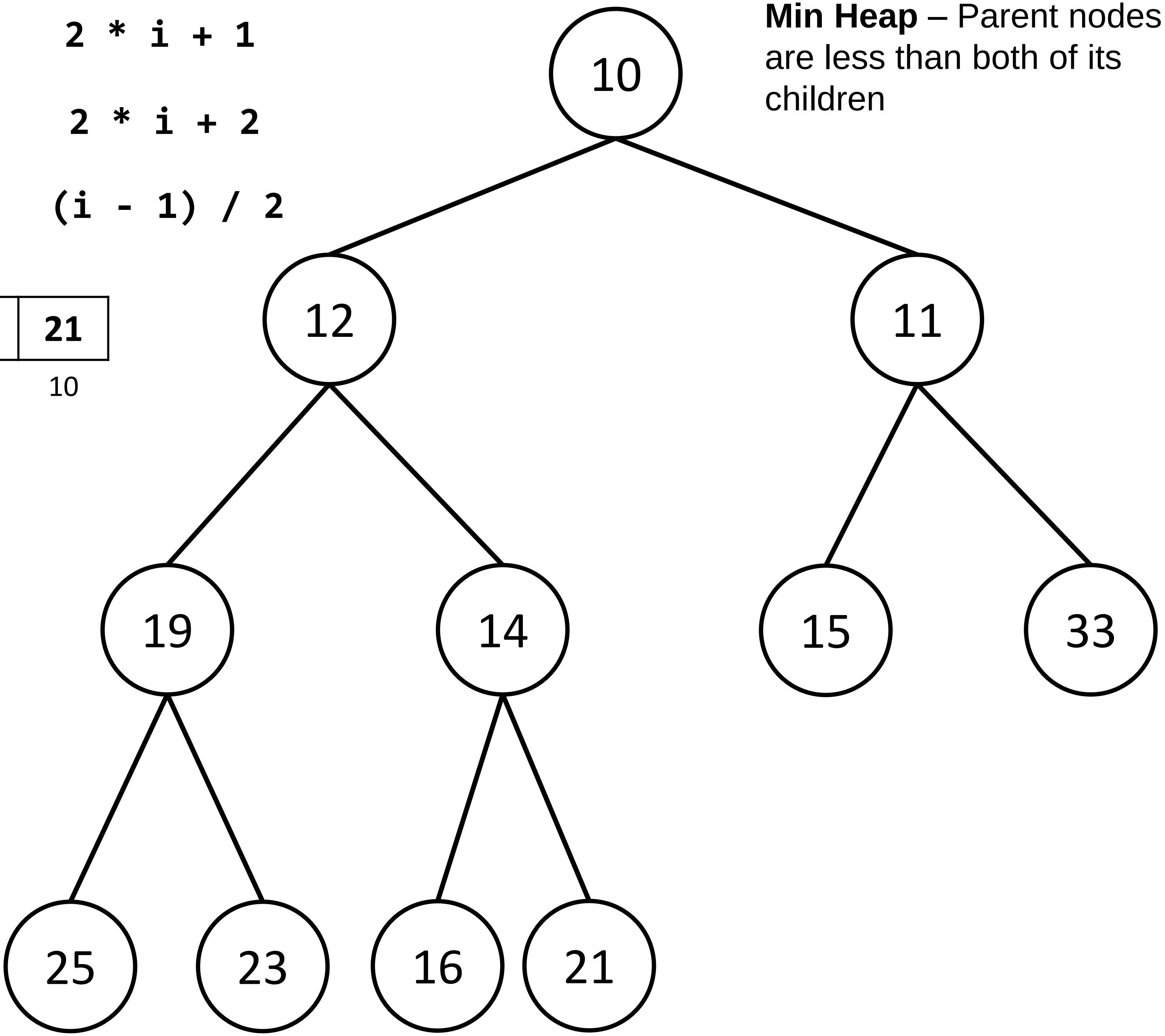
Right Child `2 * i + 2`

Parent `(i - 1) / 2`

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`

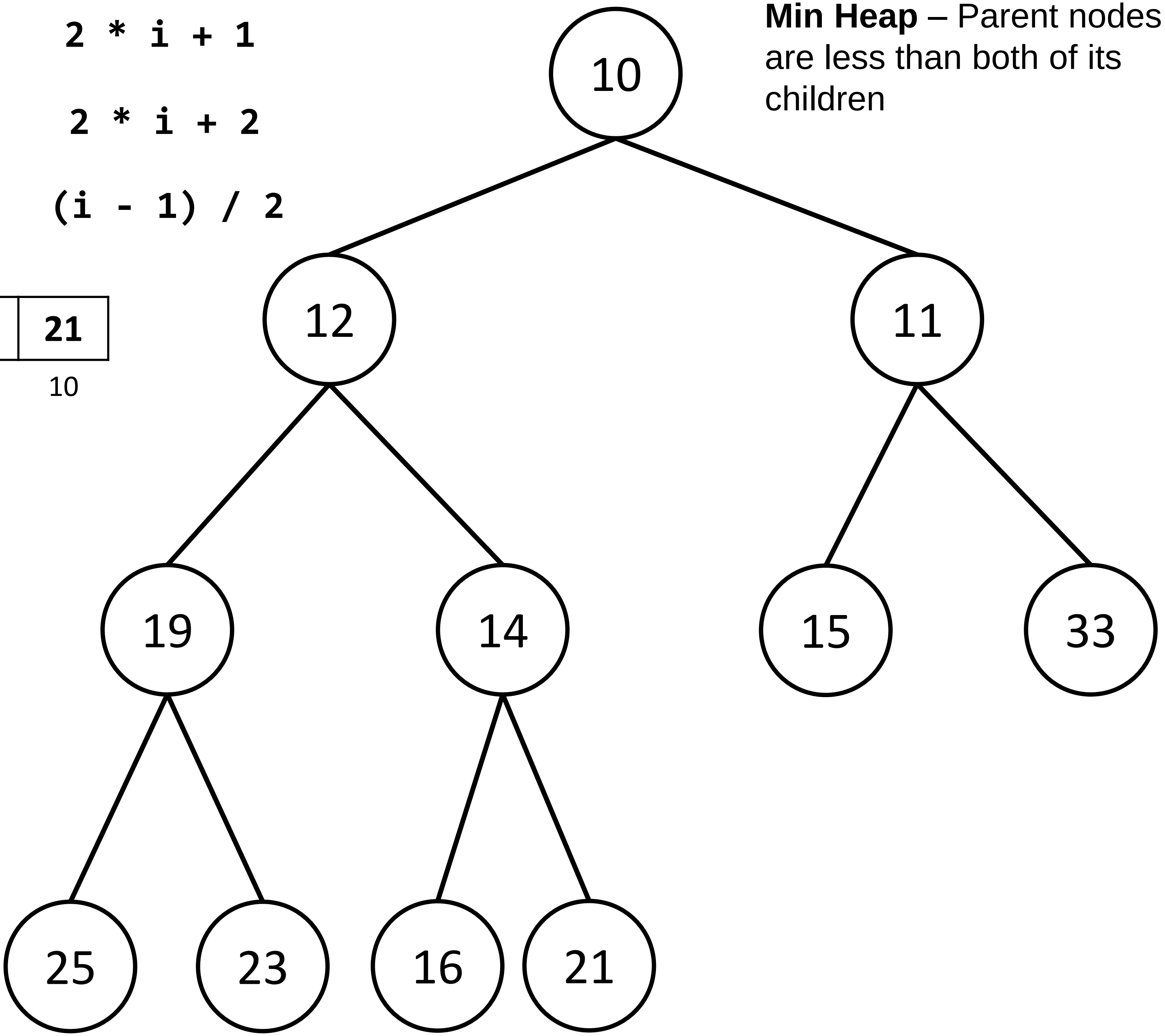
Right Child `2 * i + 2`

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

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

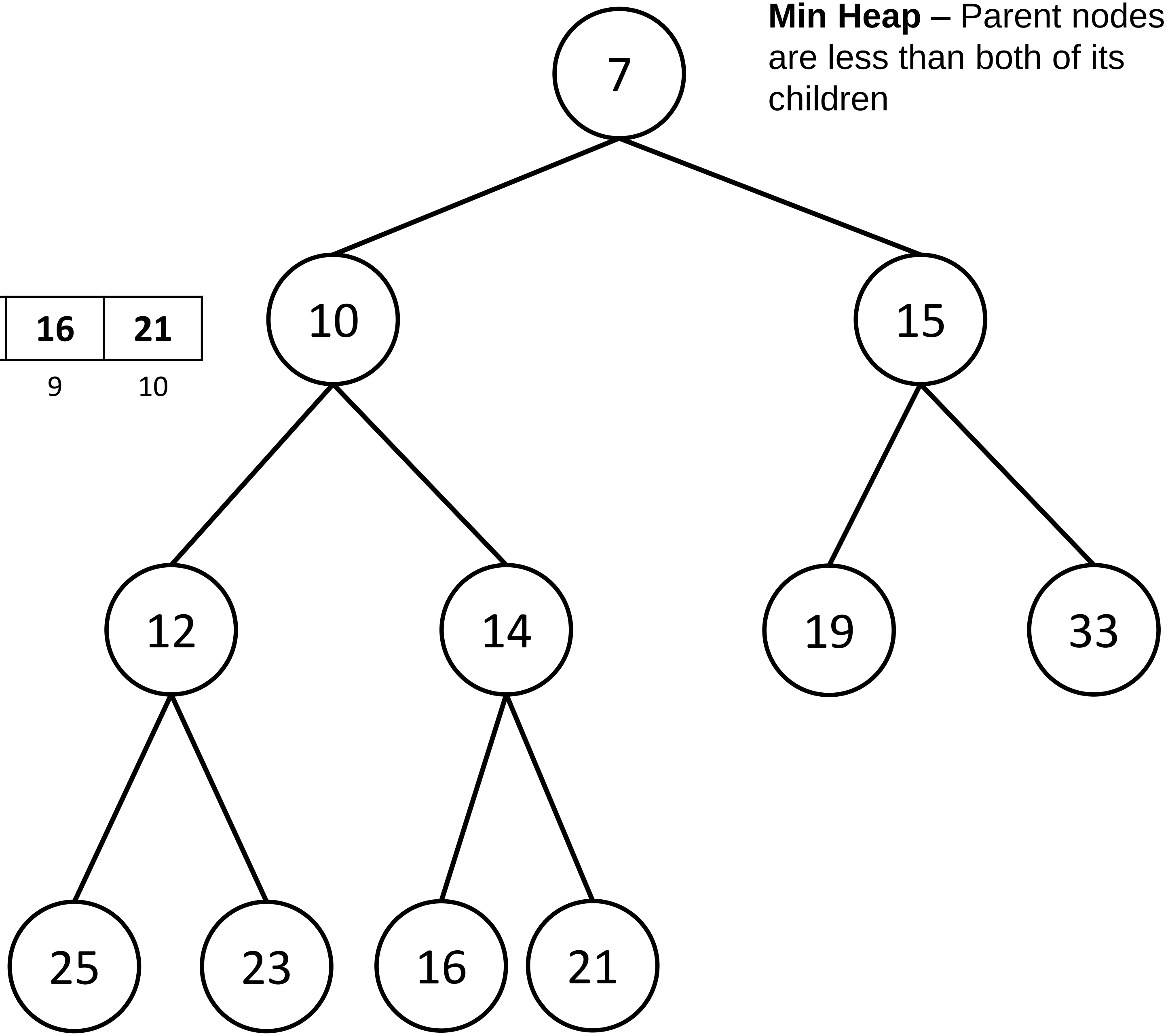


Let’s code this!!!

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

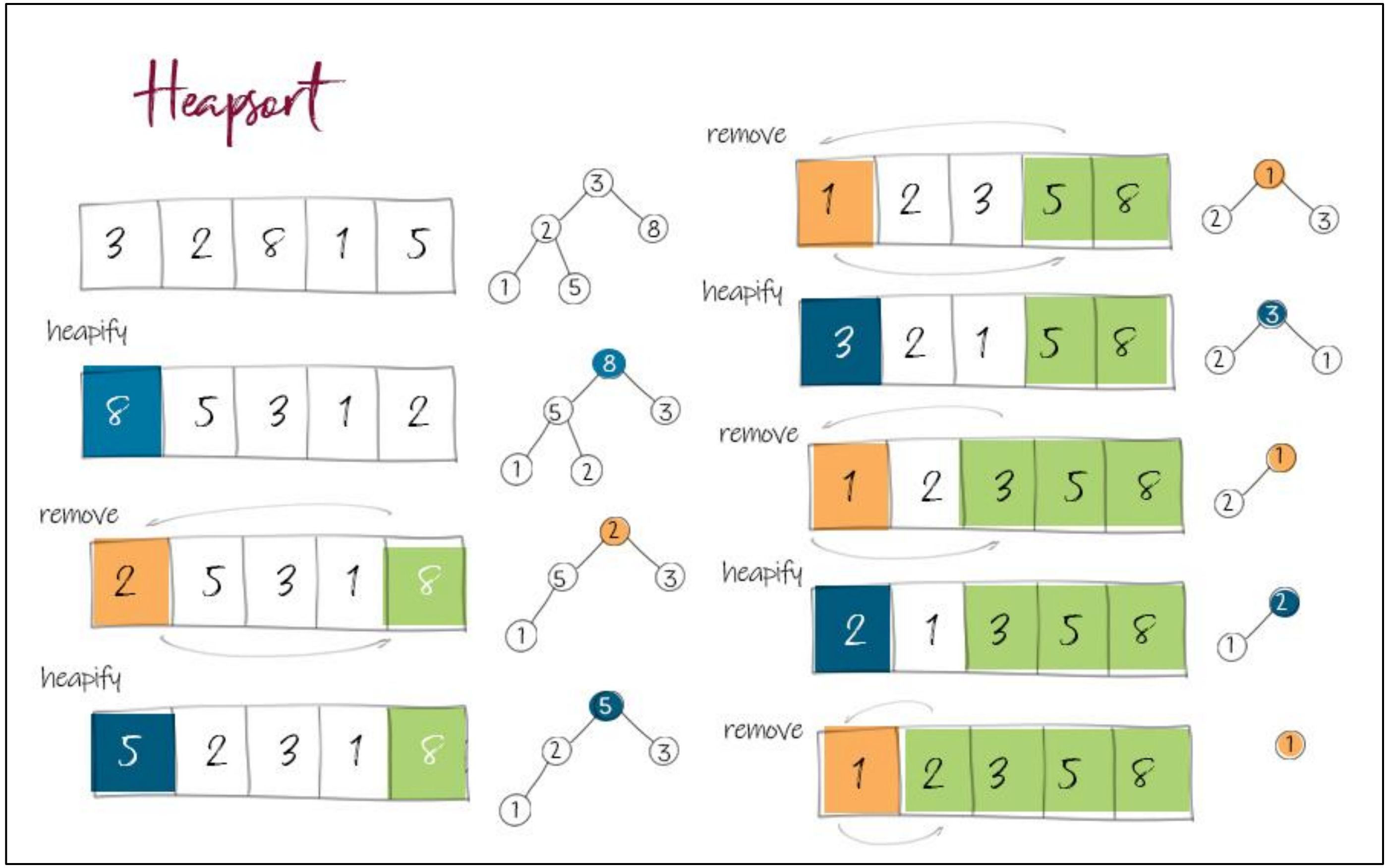
Getting the maximum/minimum value happens in O(1) time

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

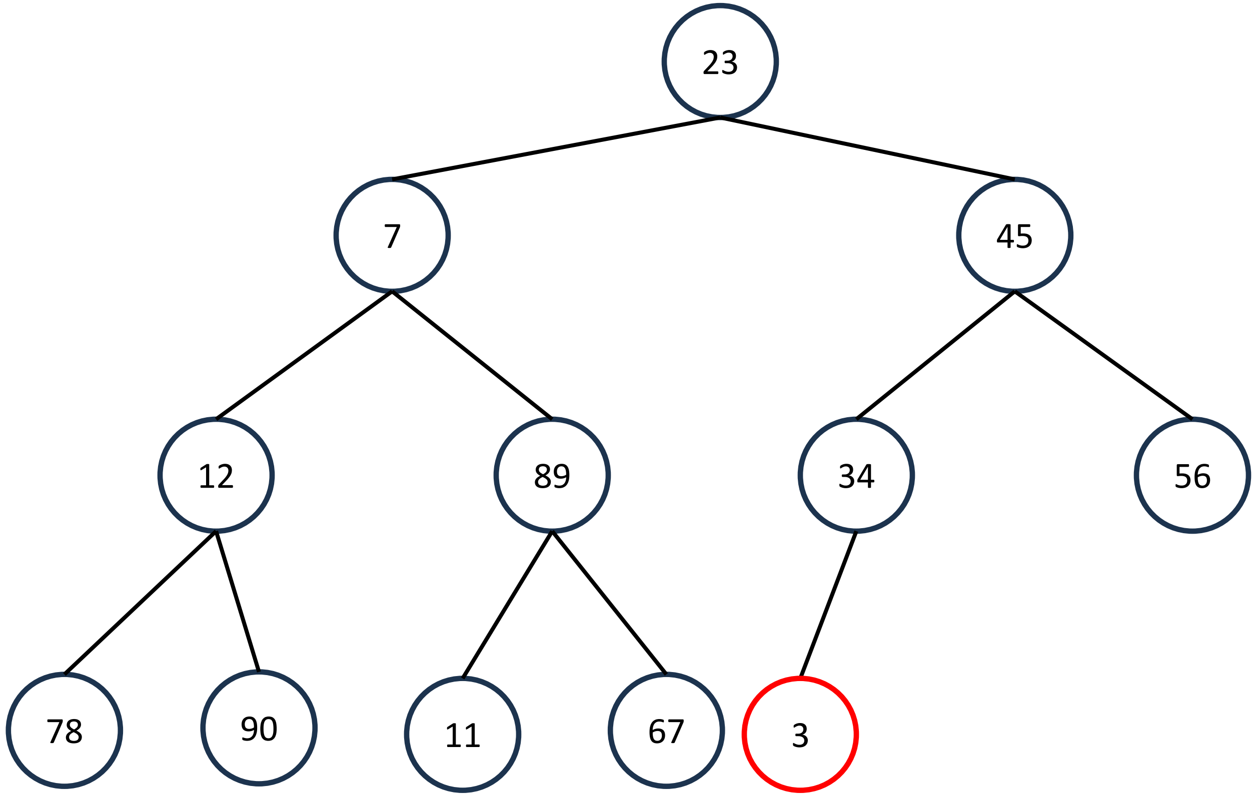


Heap Sort

```
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

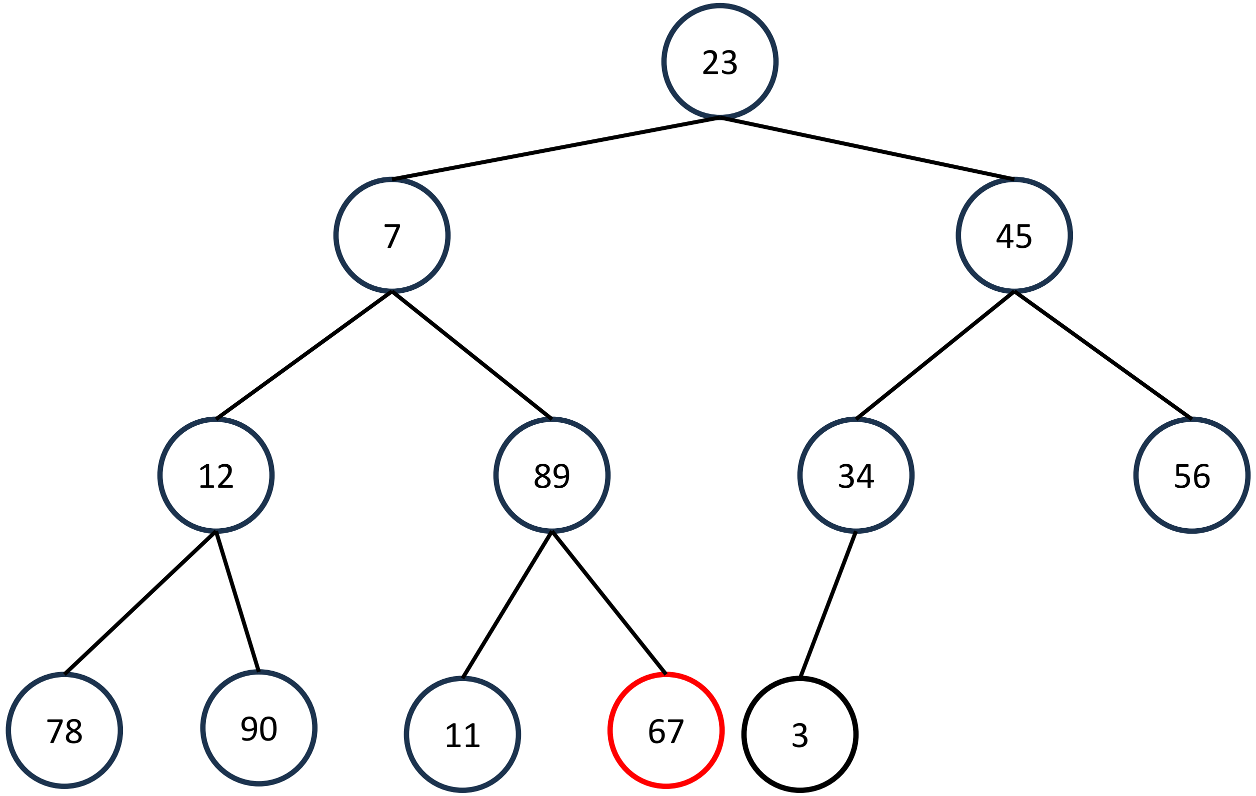


Heap Sort

```
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

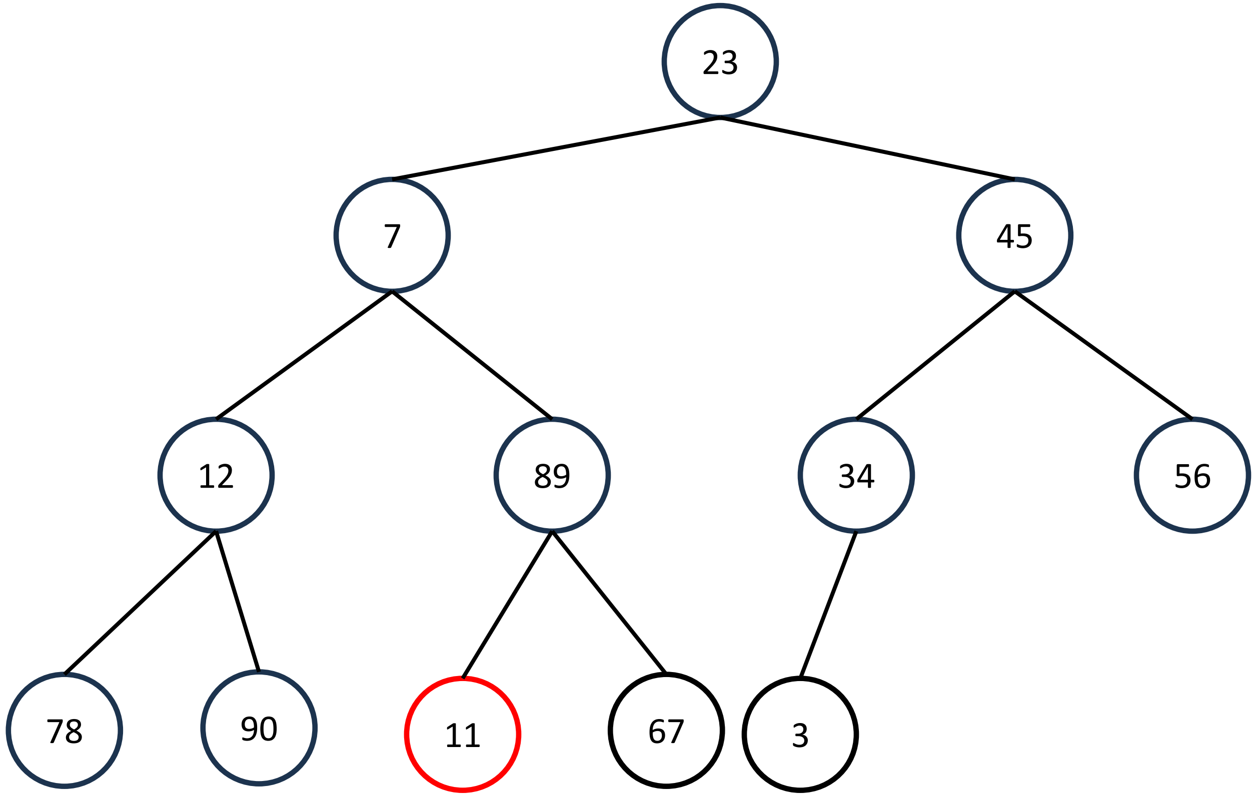


Heap Sort

```
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

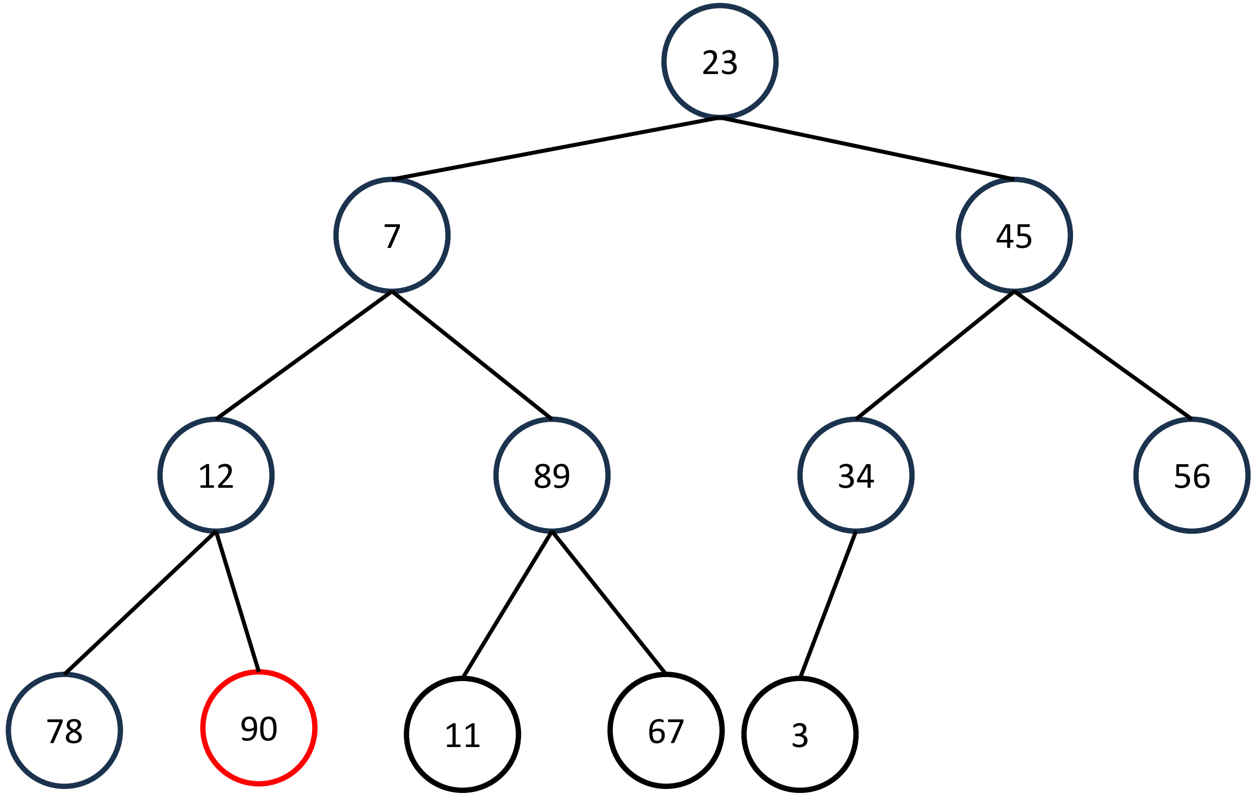


Heap Sort

```
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

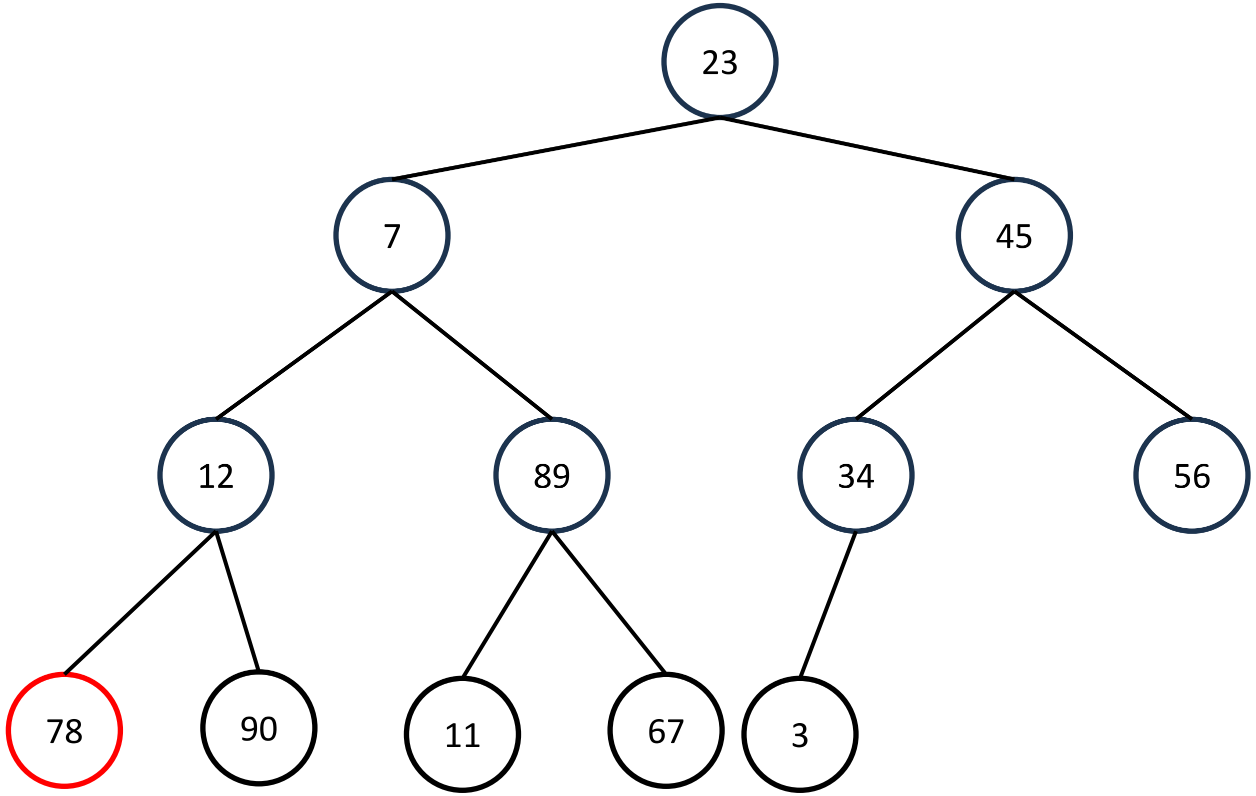


Heap Sort

```
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

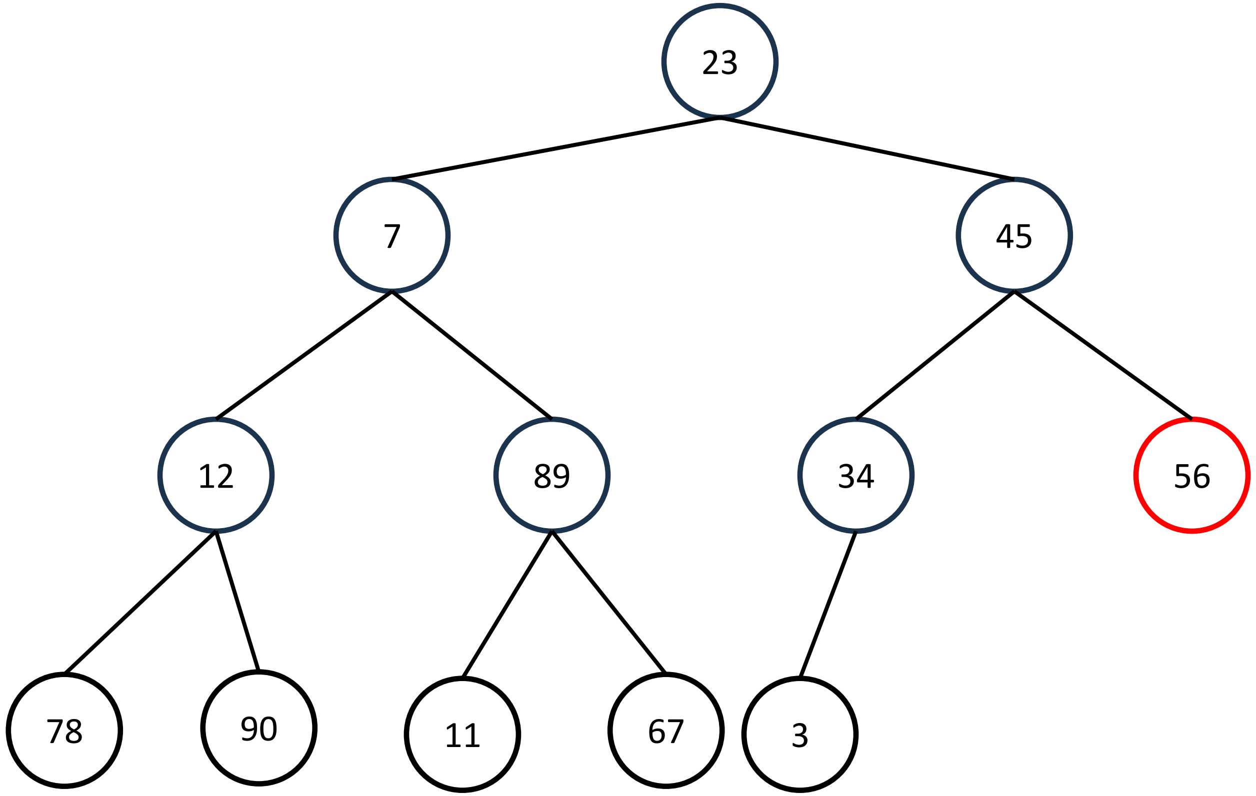


Heap Sort

```
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

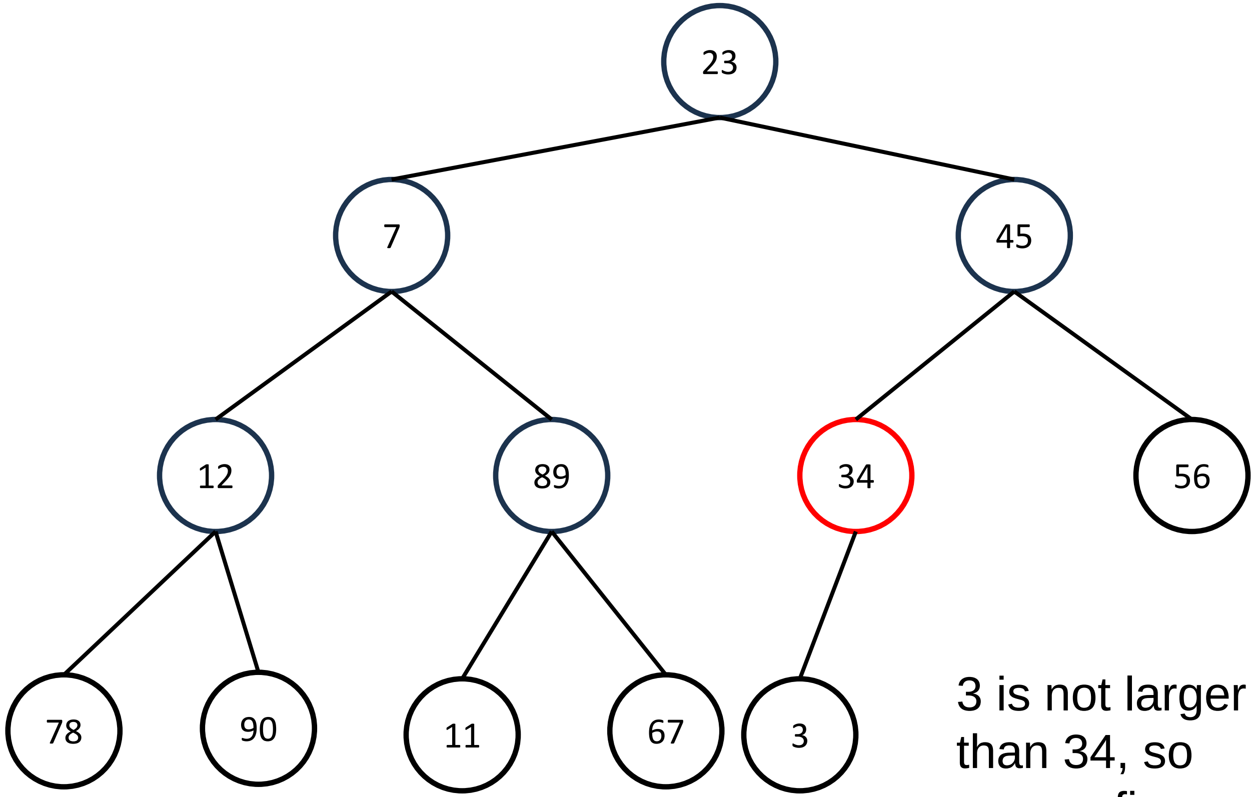


Heap Sort

```
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



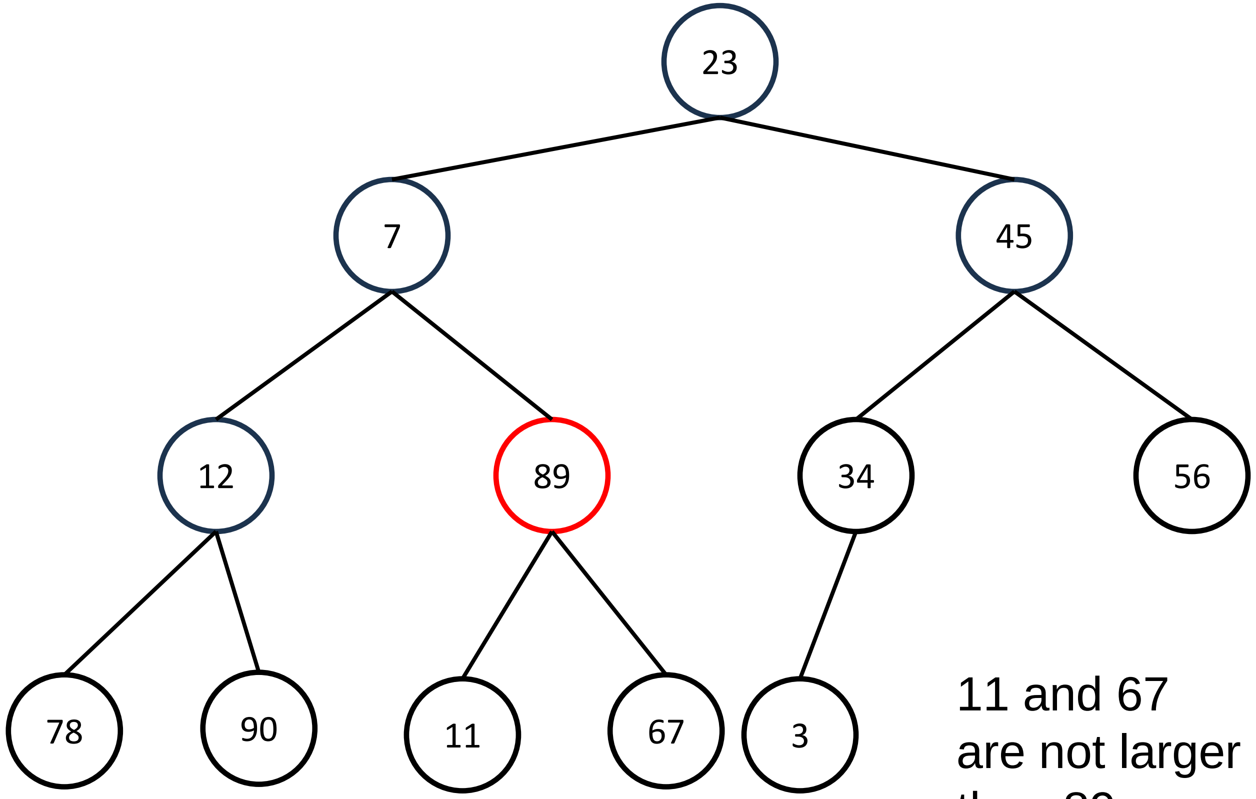
3 is not larger than 34, so we are fine

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



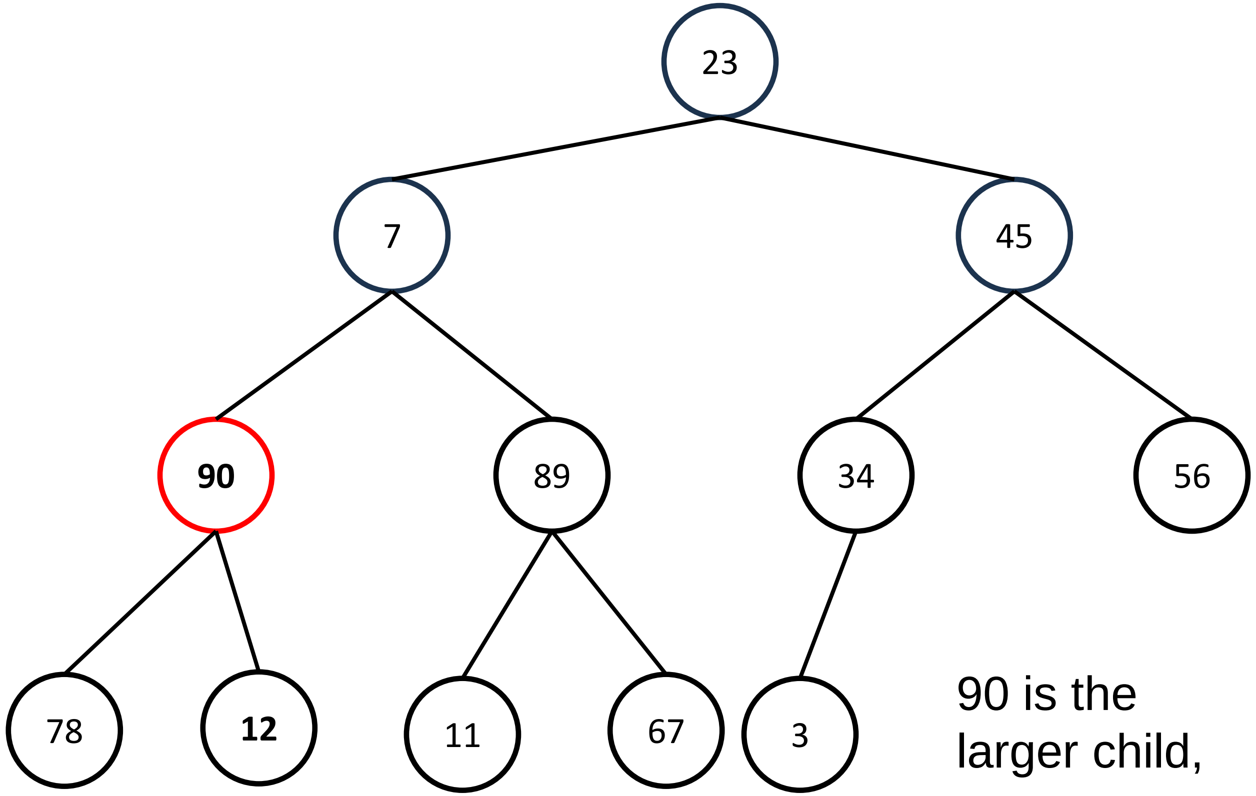
11 and 67 are not larger than 89, so continue

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



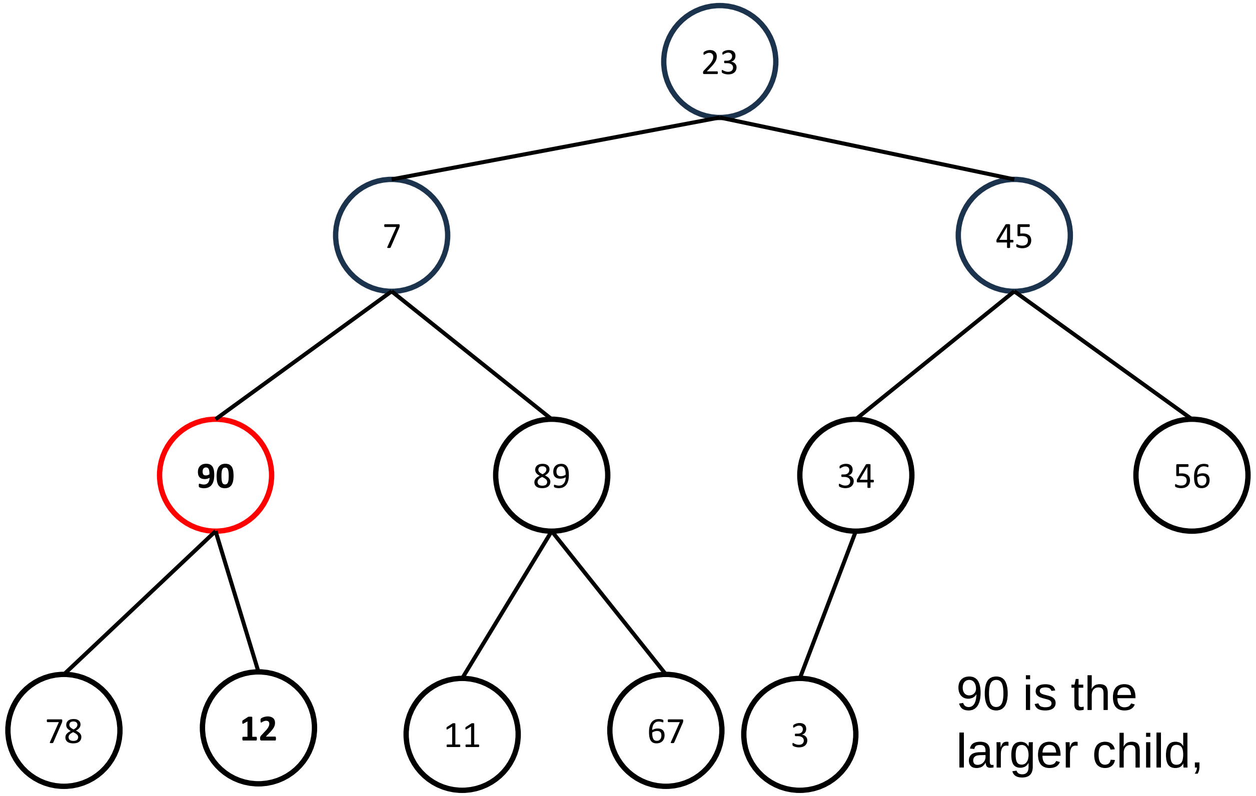
90 is the larger child, so swap

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

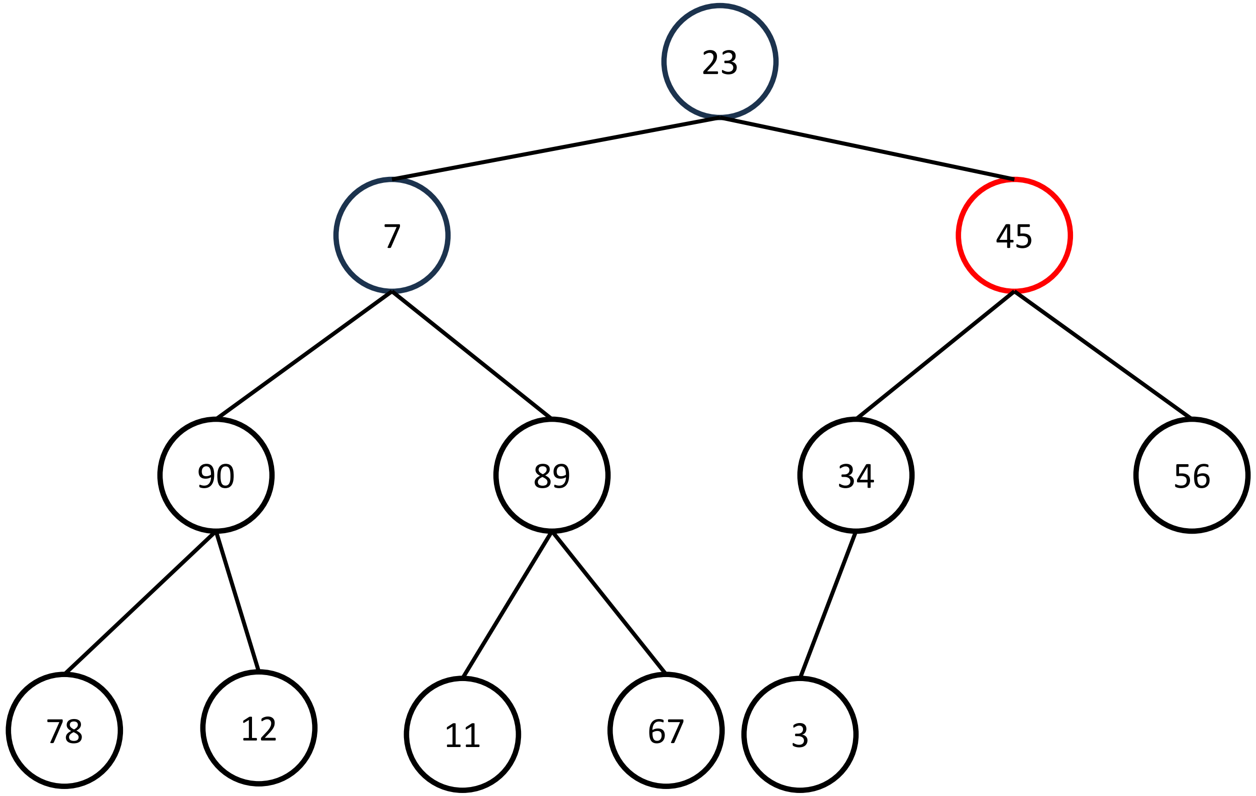
90 is larger than 78 and 12 so continue

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

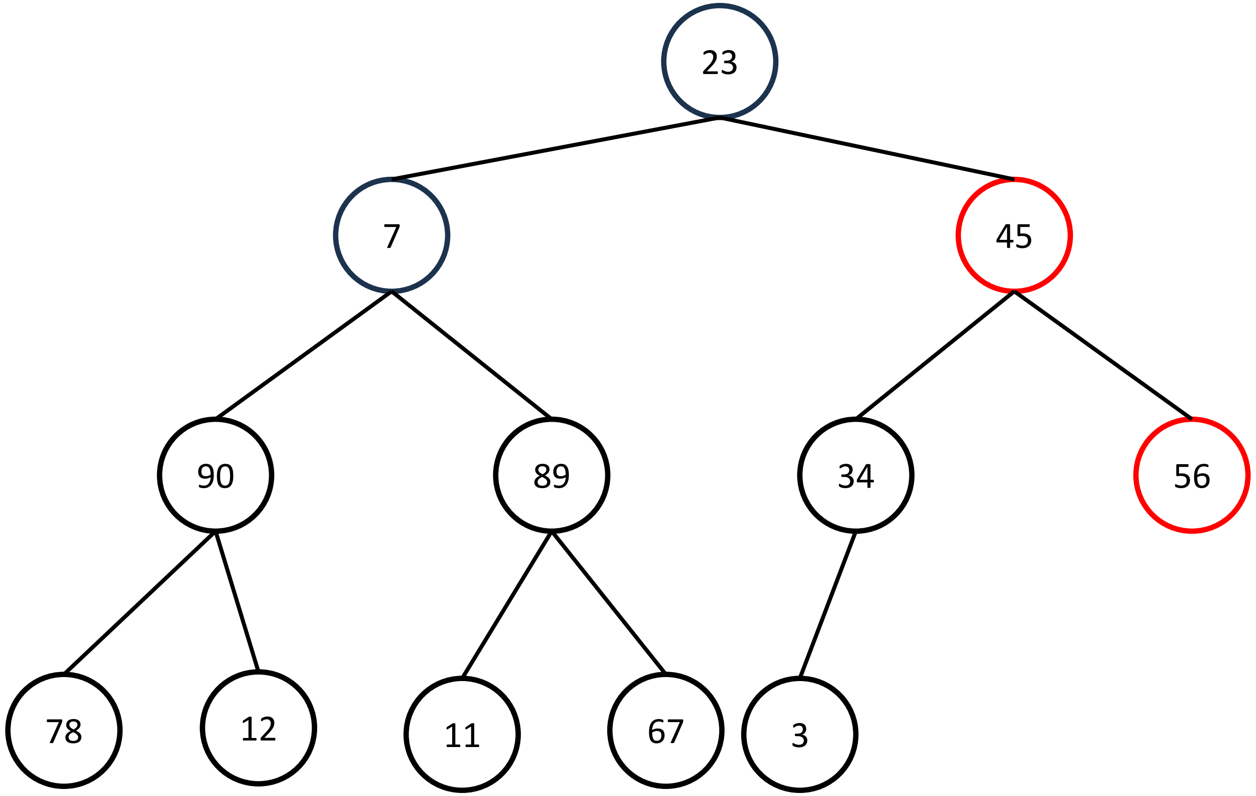


Heap Sort

```
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

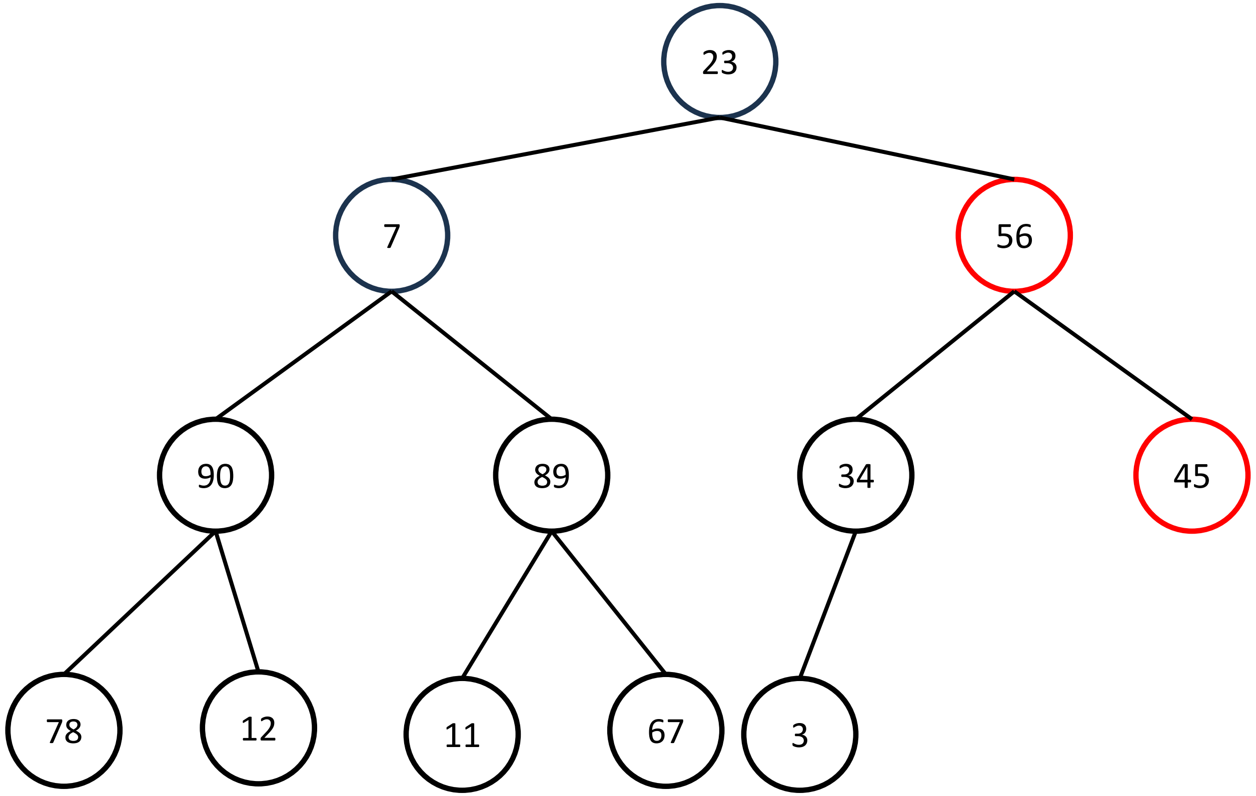


Heap Sort

```
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

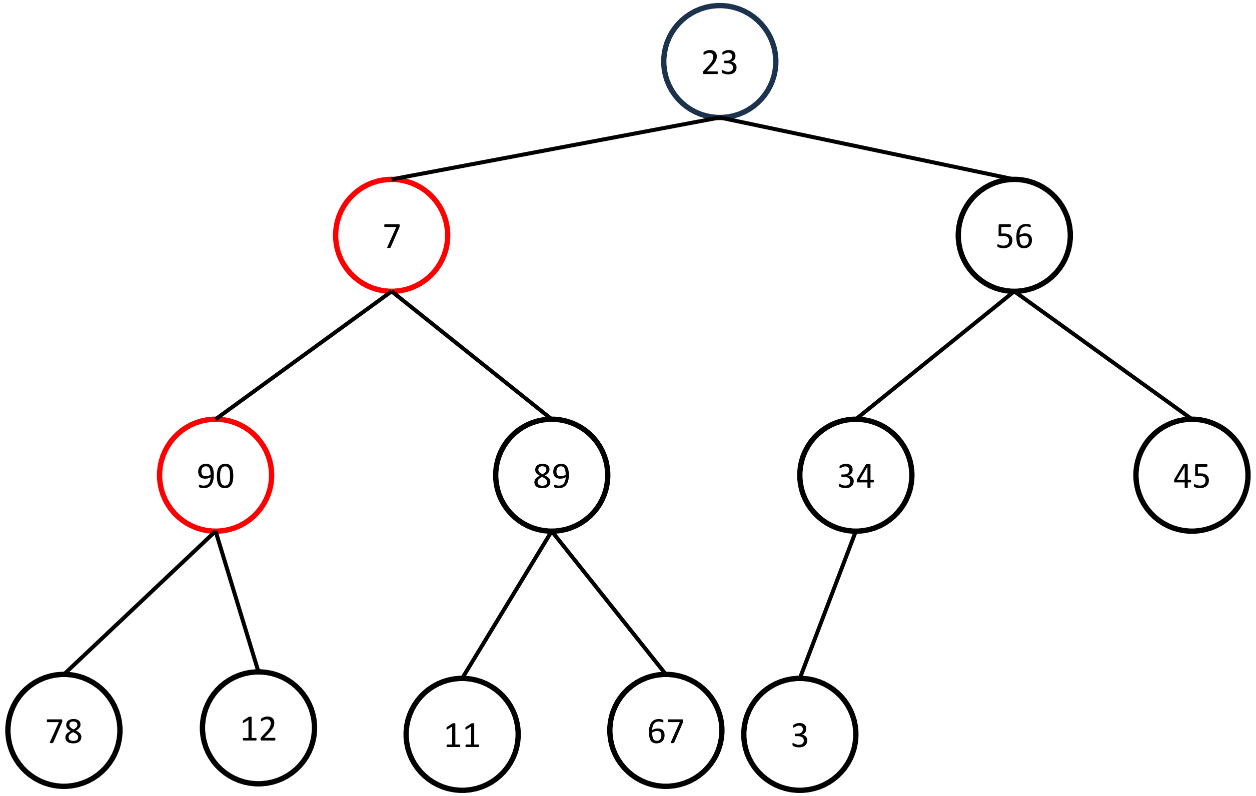


Heap Sort

```
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

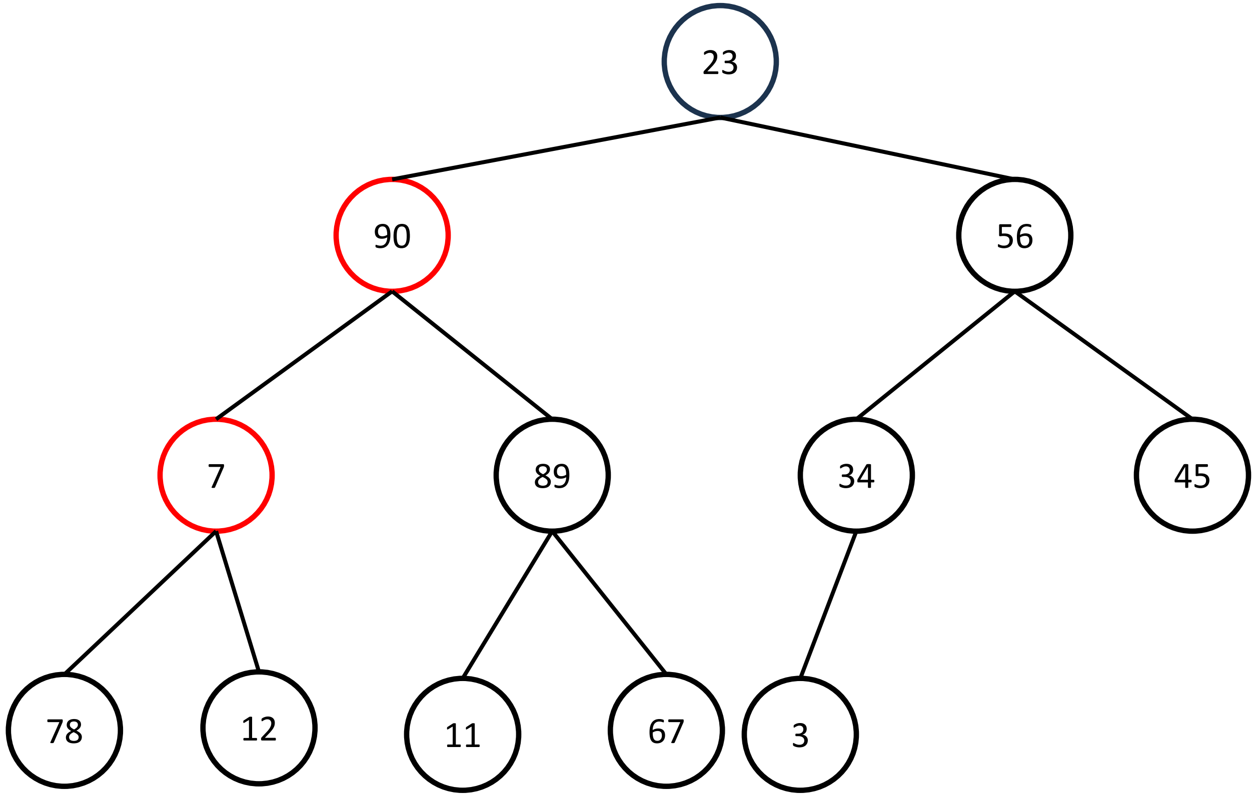


Heap Sort

```
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



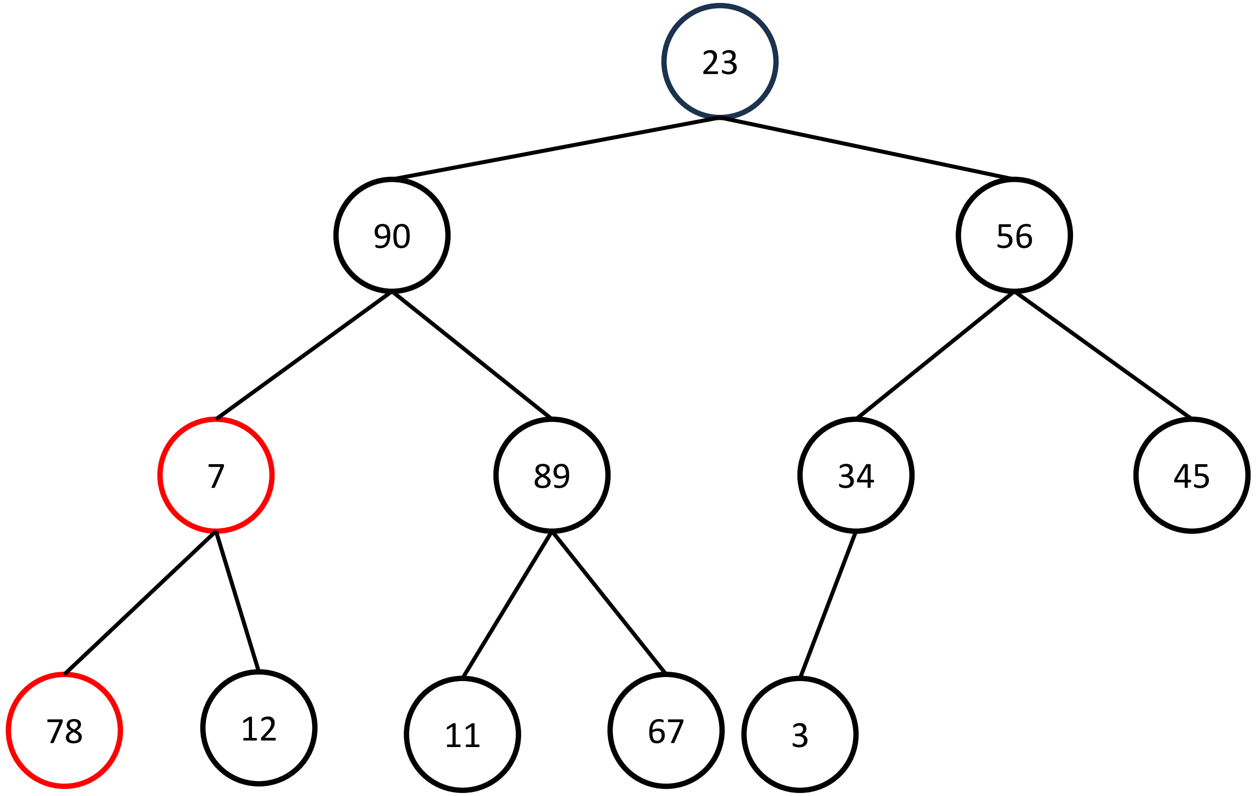
Heapify Down 7 !

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



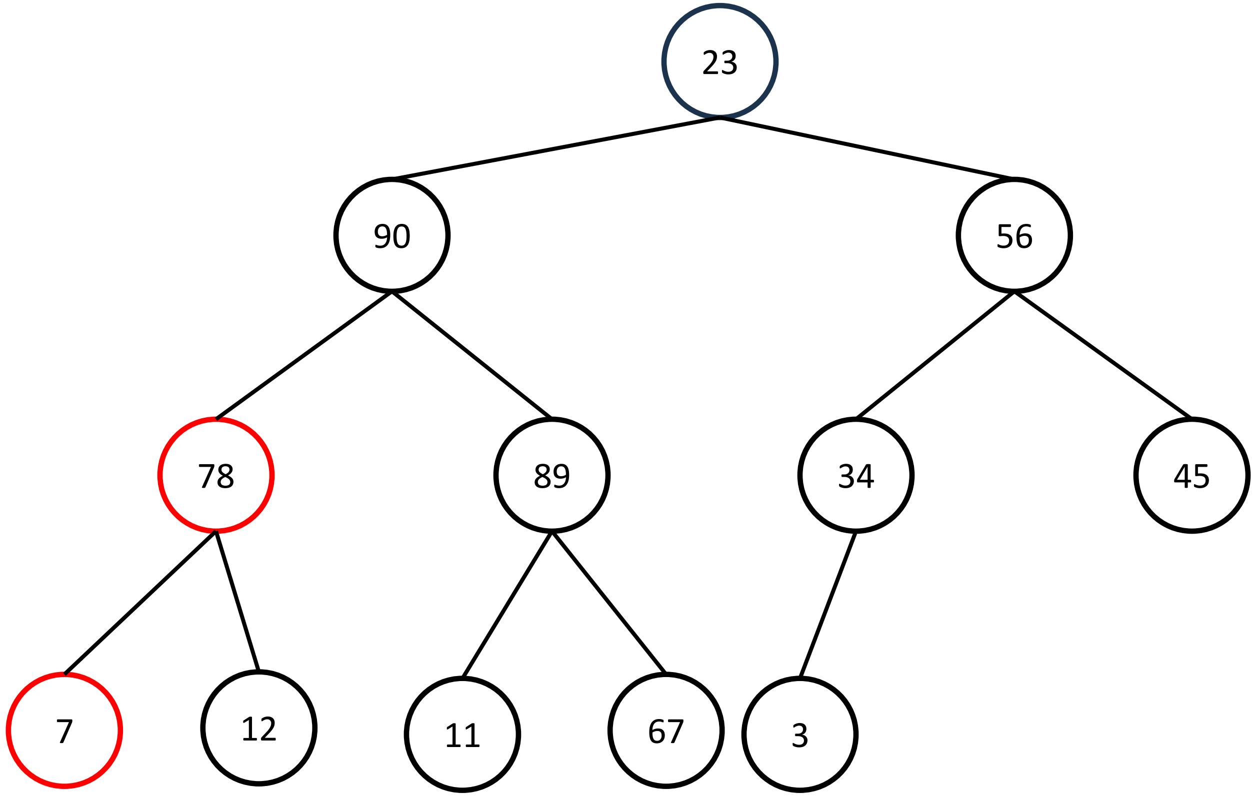
Heapify Down 7 !

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



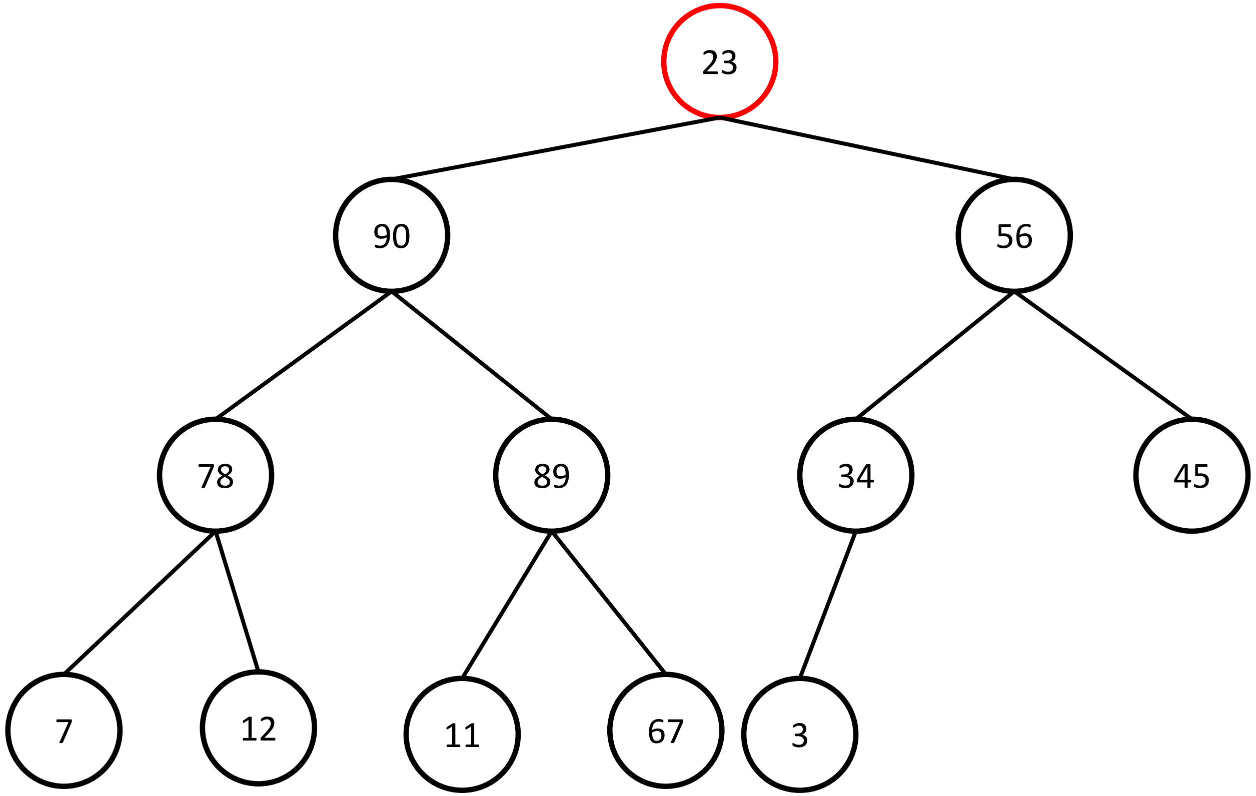
Heapify Down 7 !

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



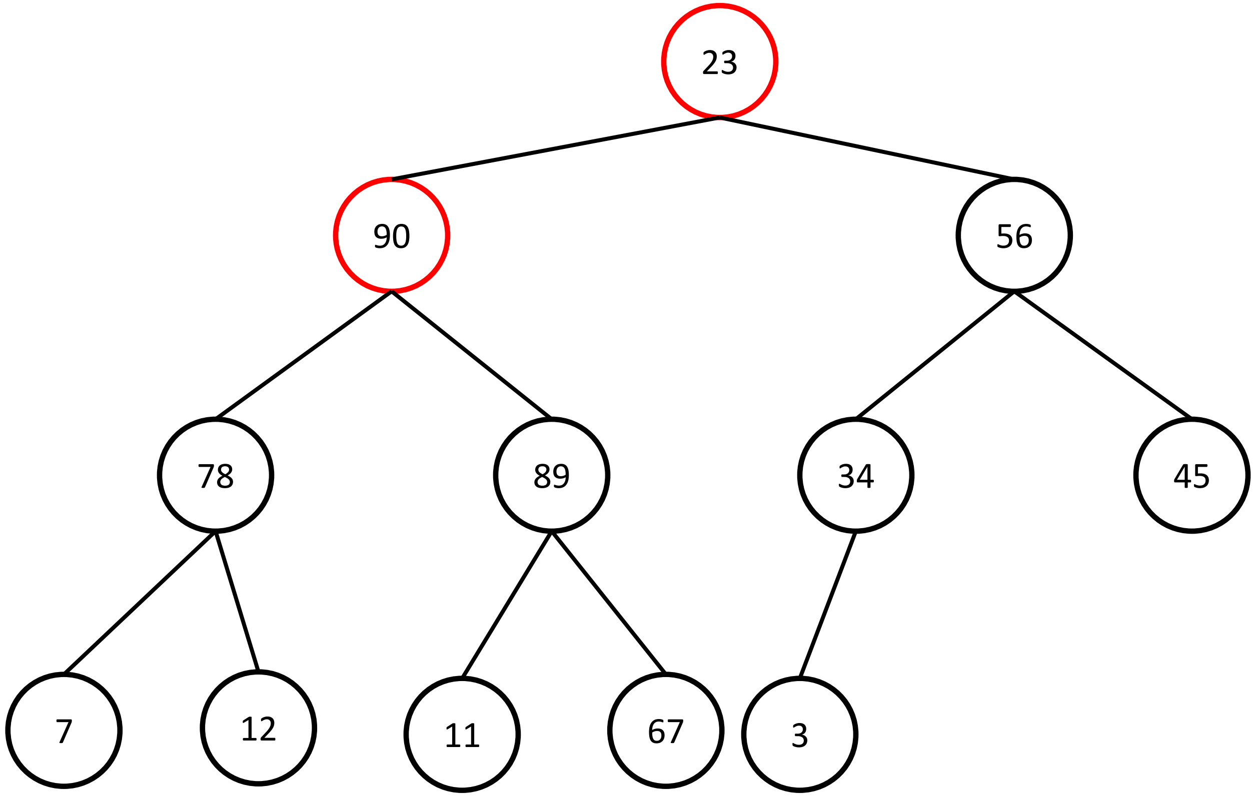
Heapify down 23 !

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



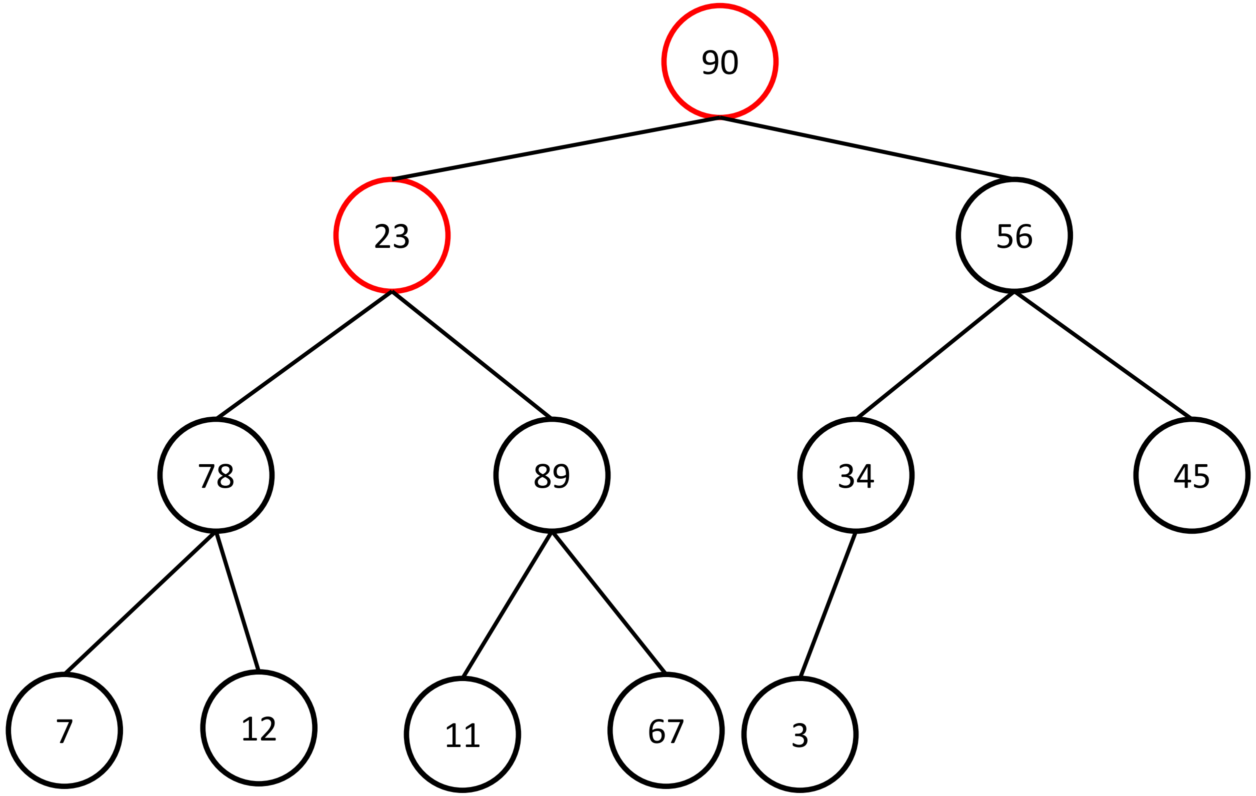
Heapify down 23 !

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



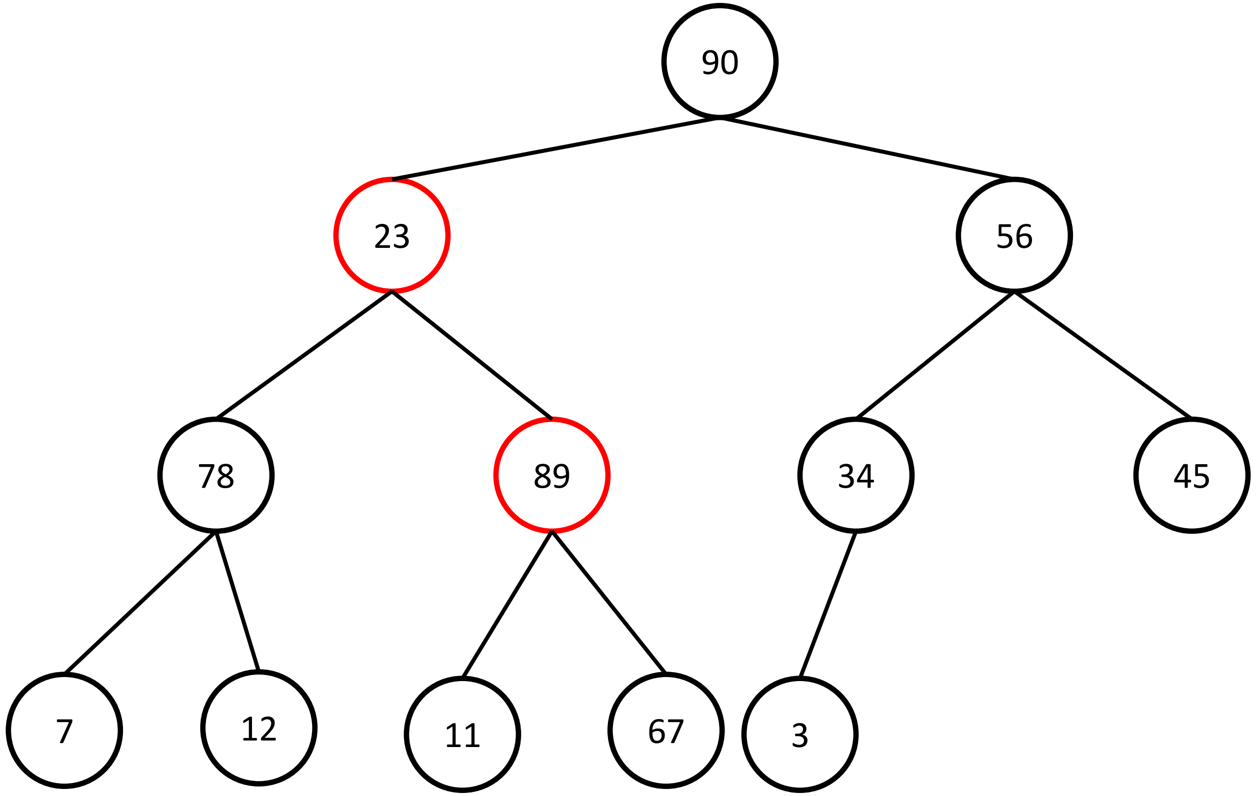
Heapify down 23 !

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



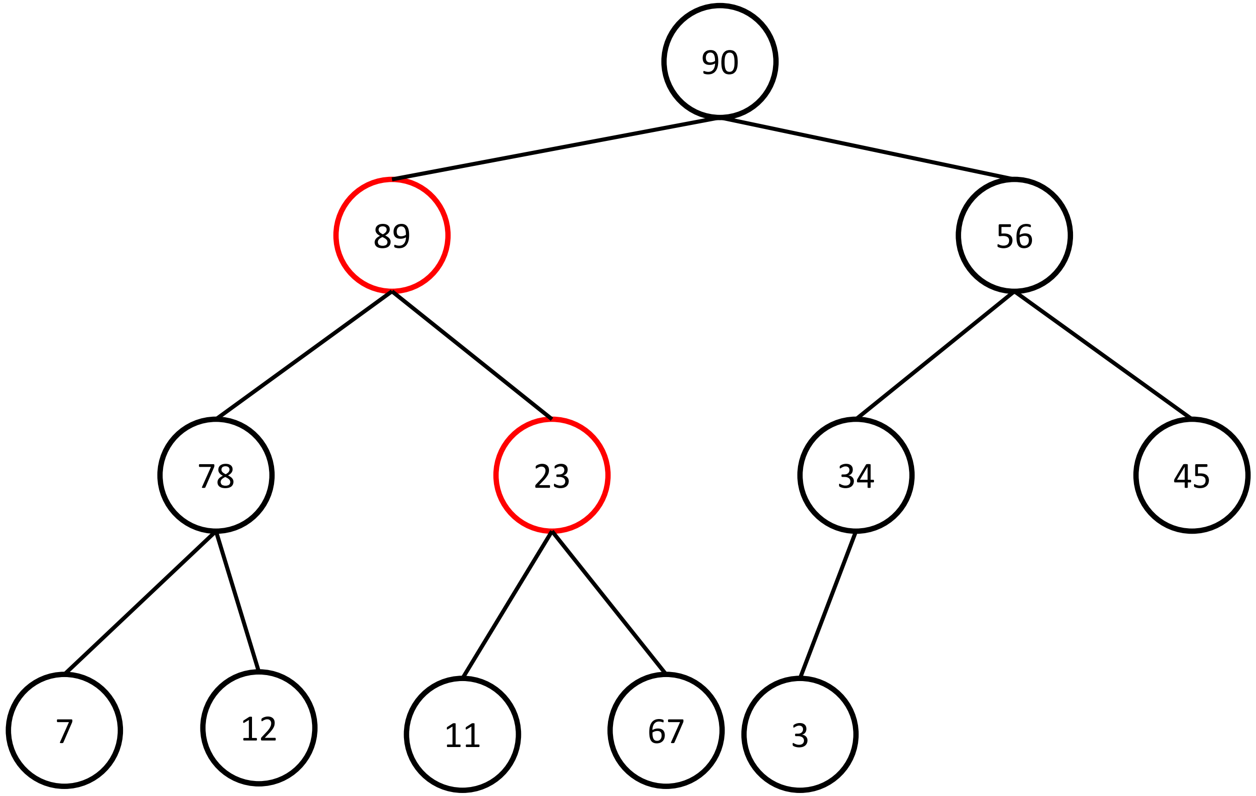
Heapify down 23 !

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



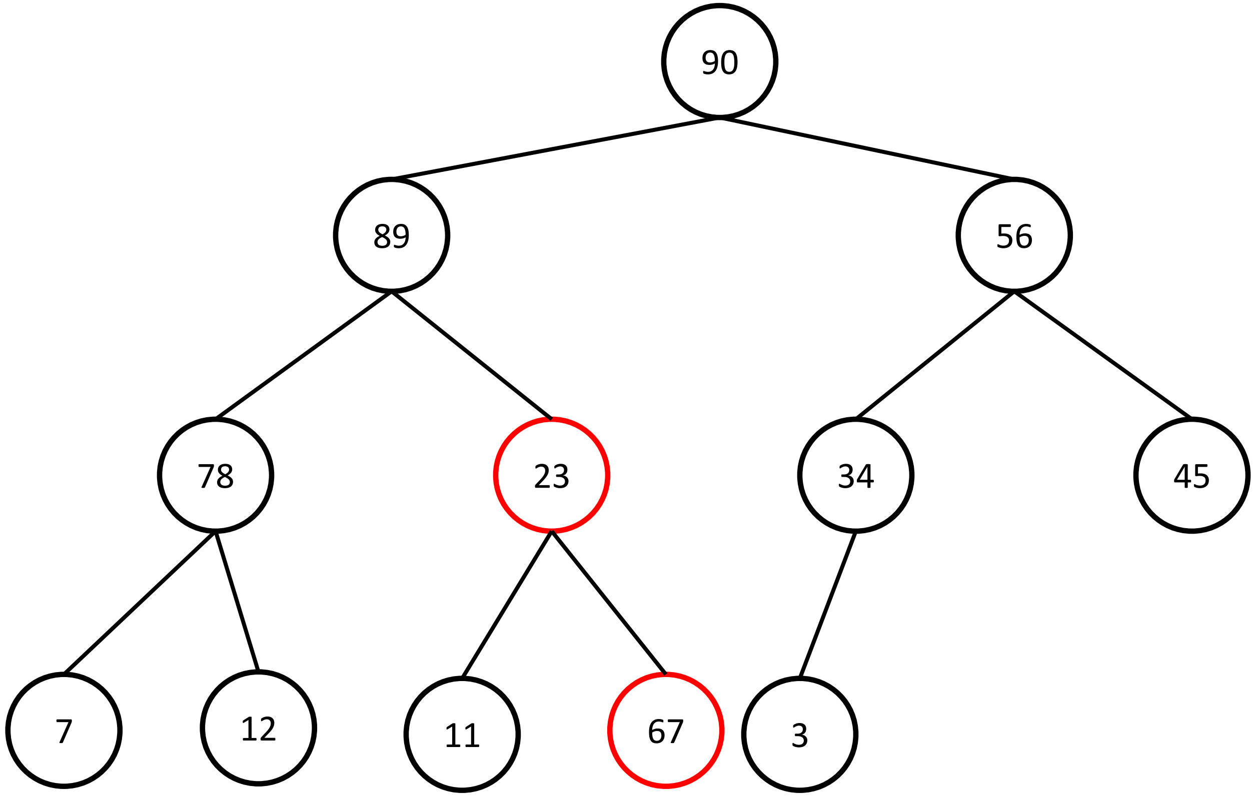
Heapify down 23 !

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

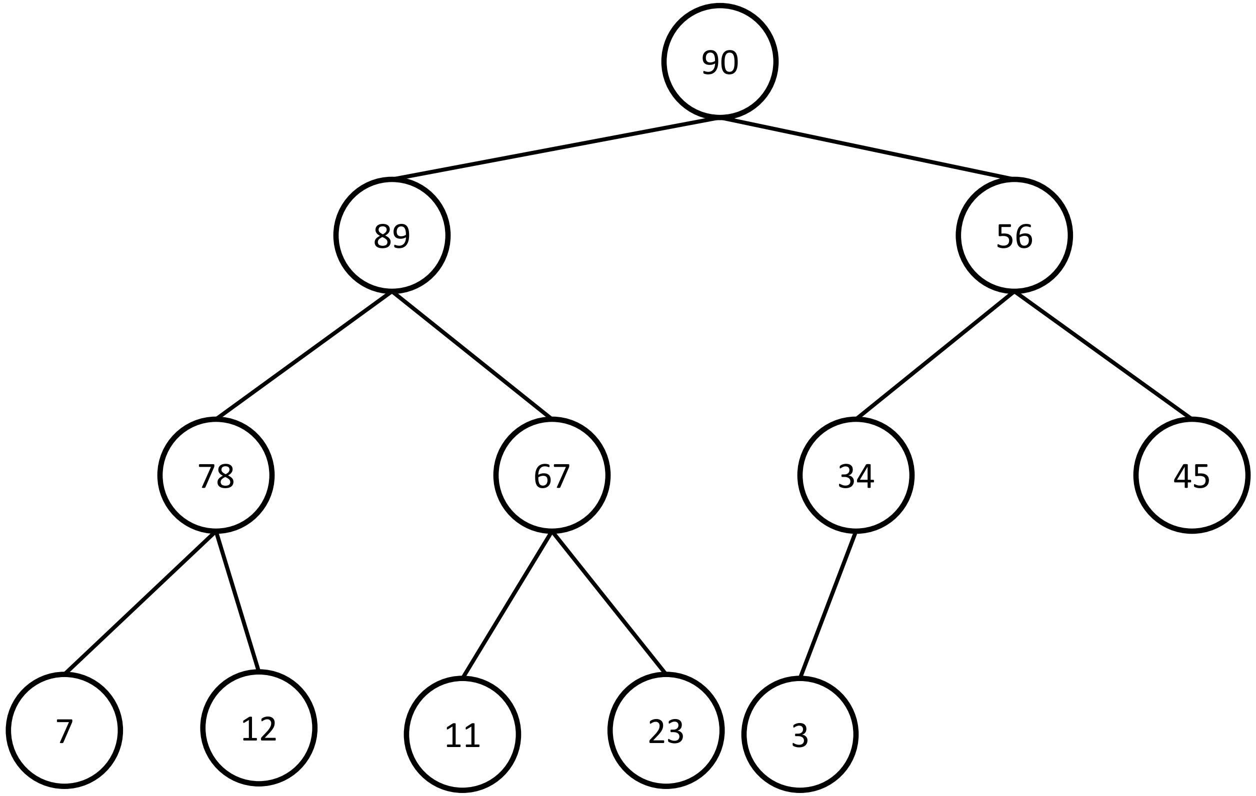
```
int[] data = {90, 89, 56, 78, 67, 34, 45, 7, 12, 11, 23, 3}
```

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

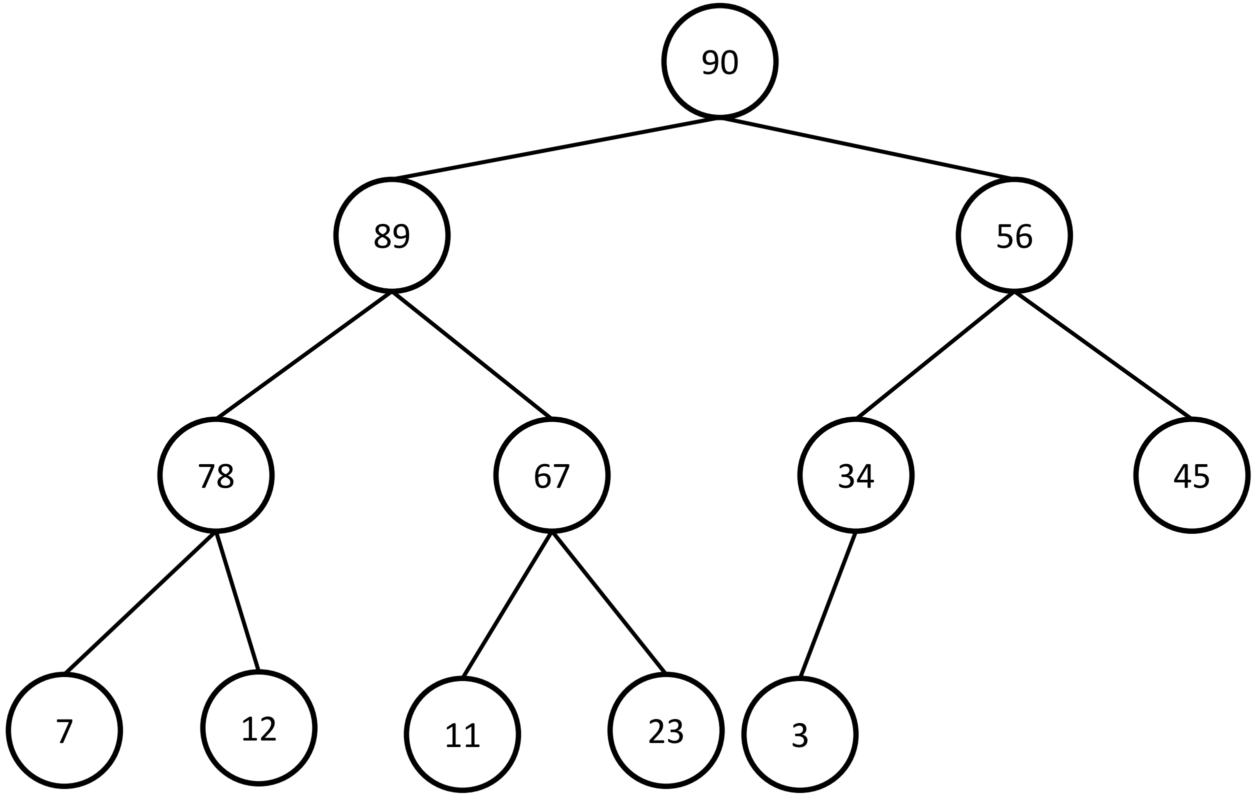
Heap Sort

```
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



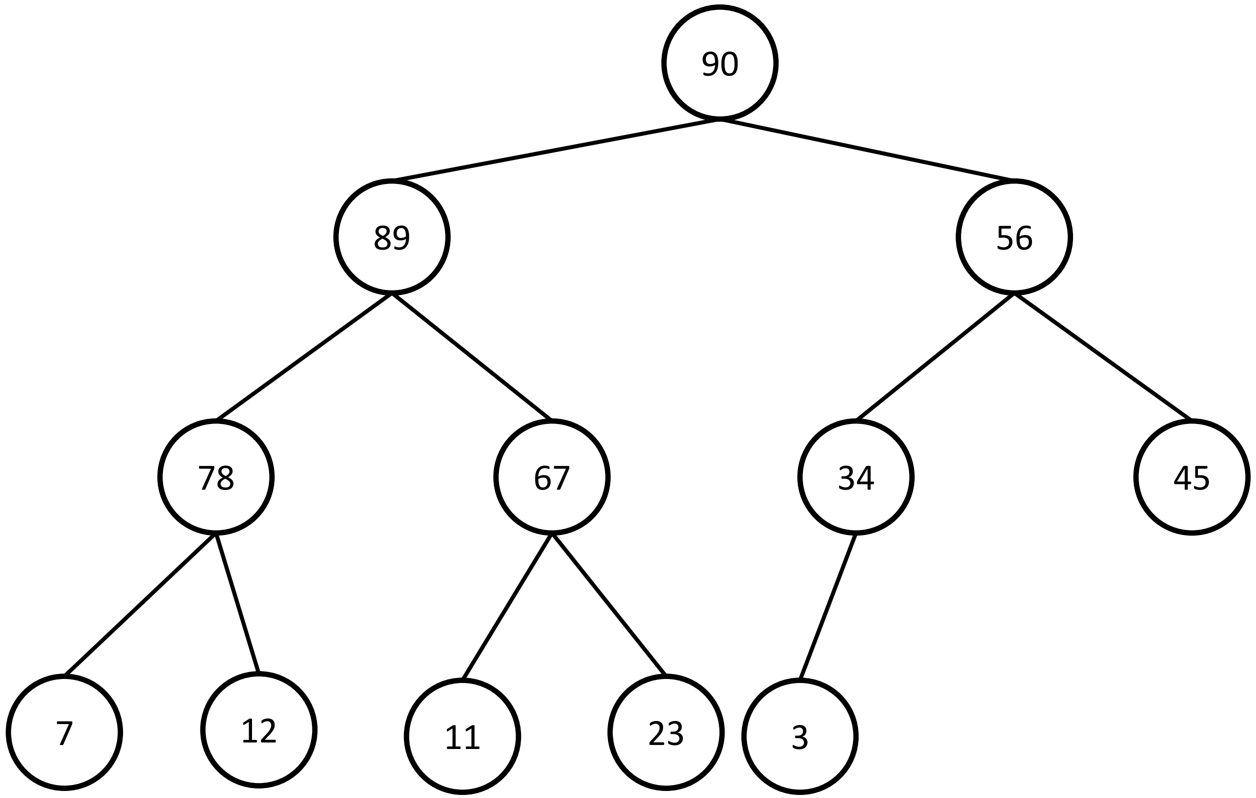
Heap Sort

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

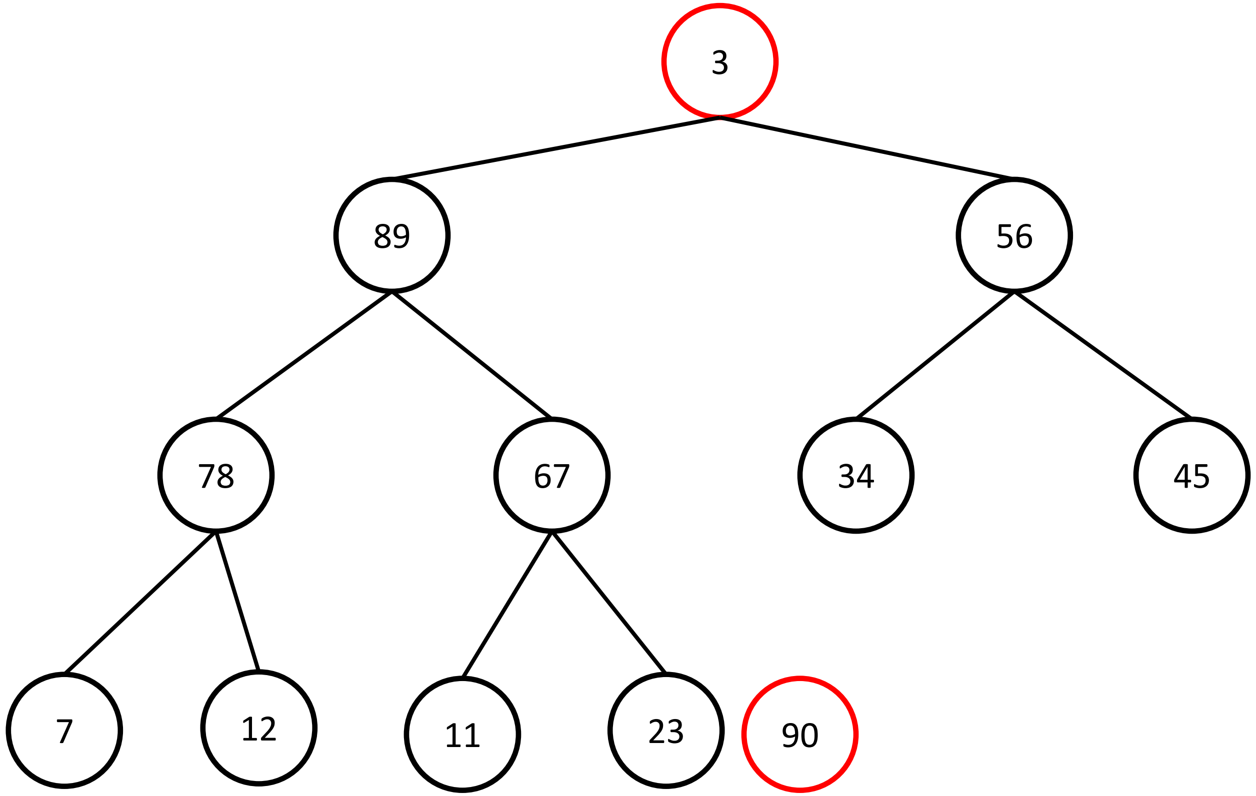
Heap Sort

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

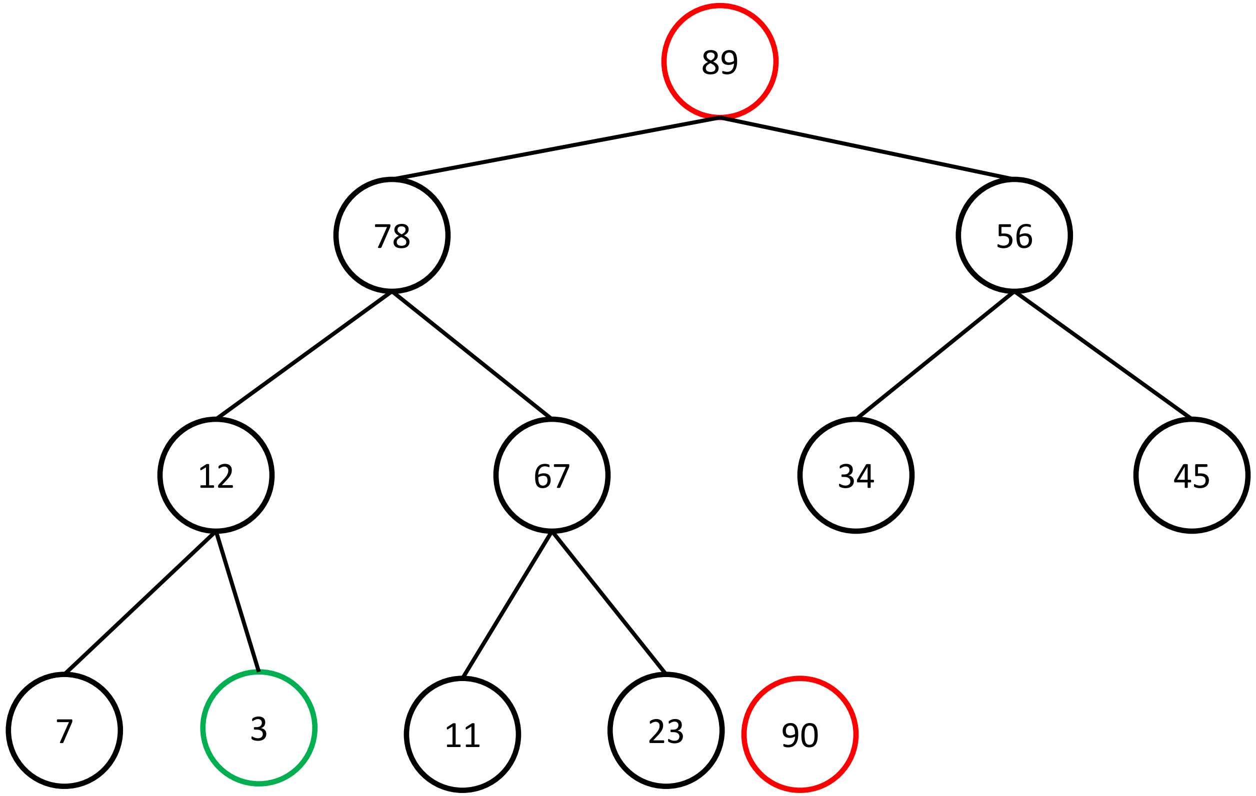
Heap Sort

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

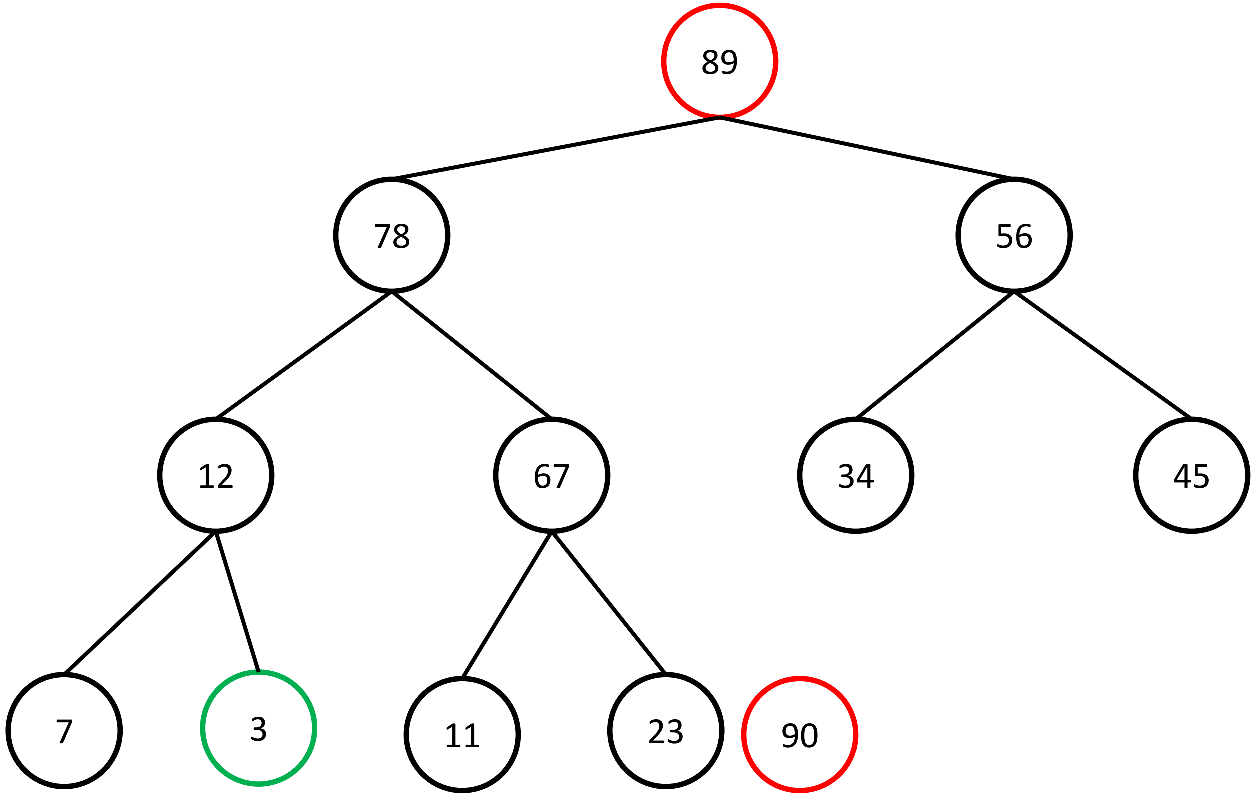
Heap Sort

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

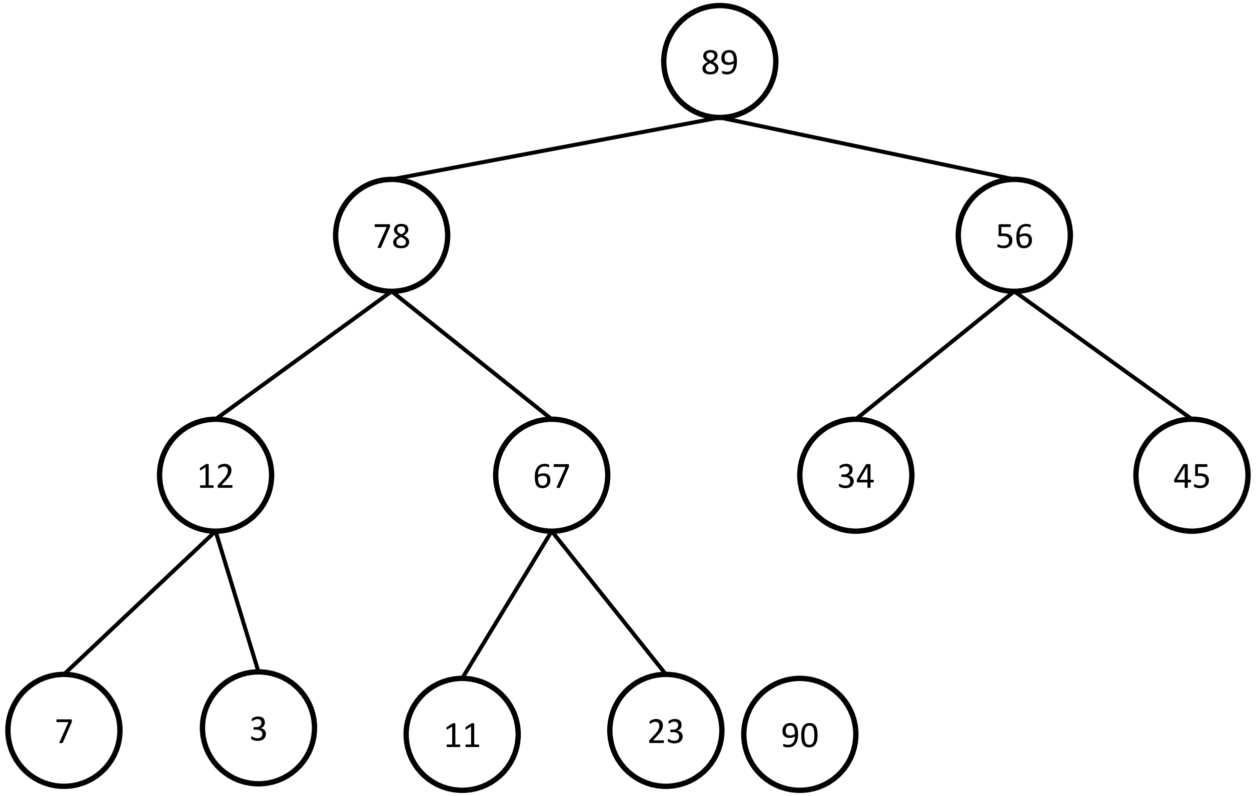
Heap Sort

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

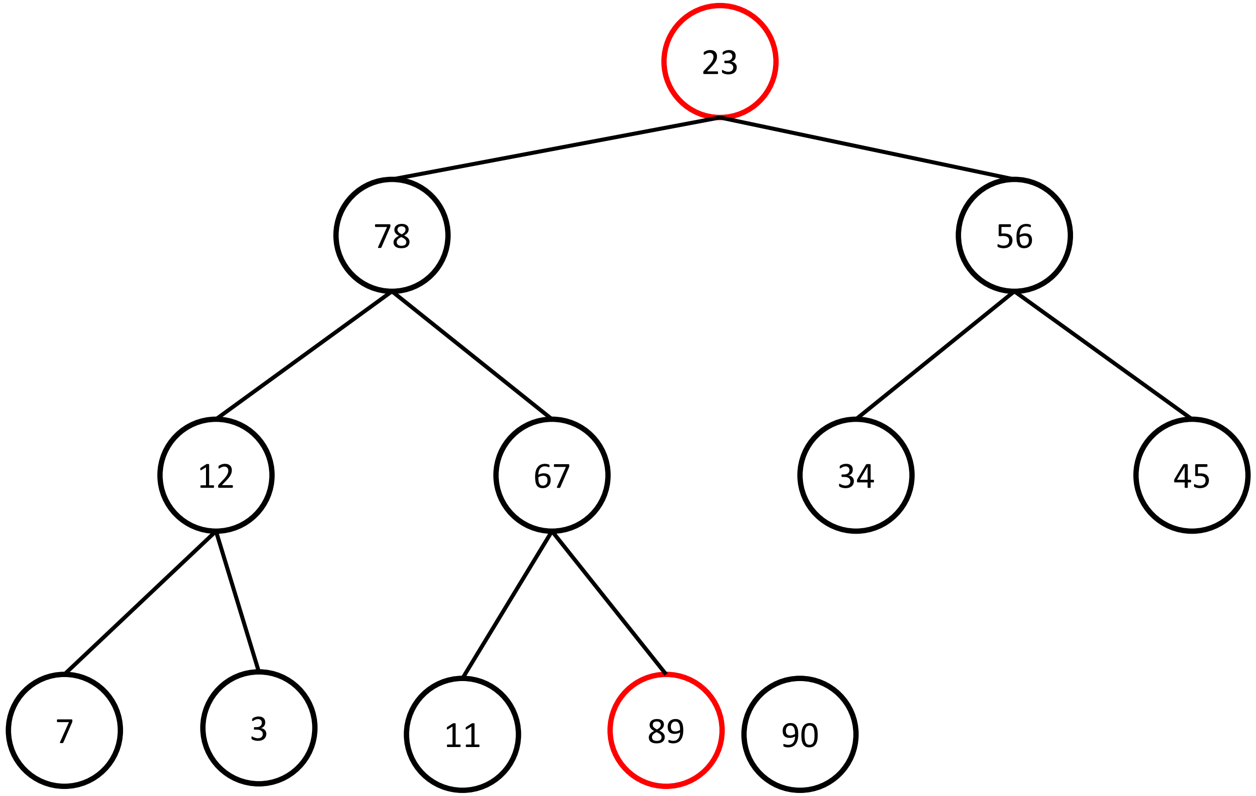
Heap Sort

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

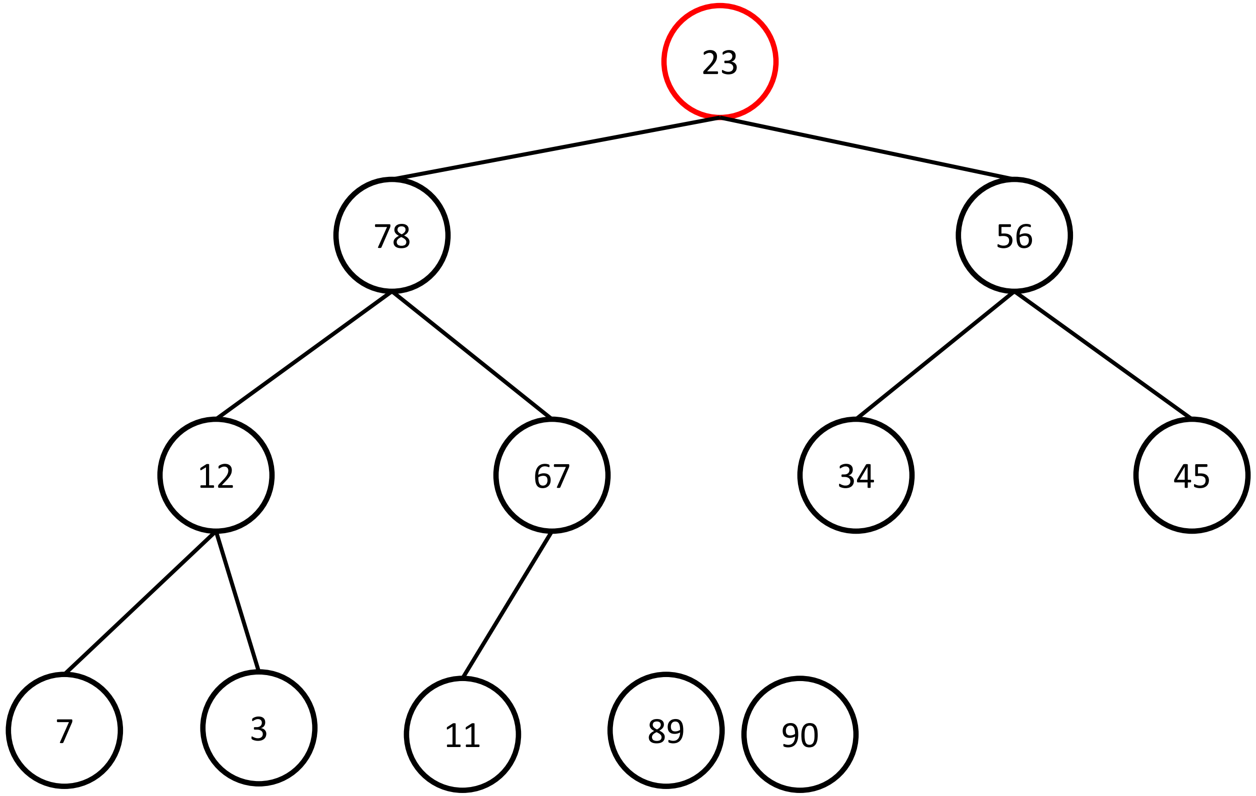
Heap Sort

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

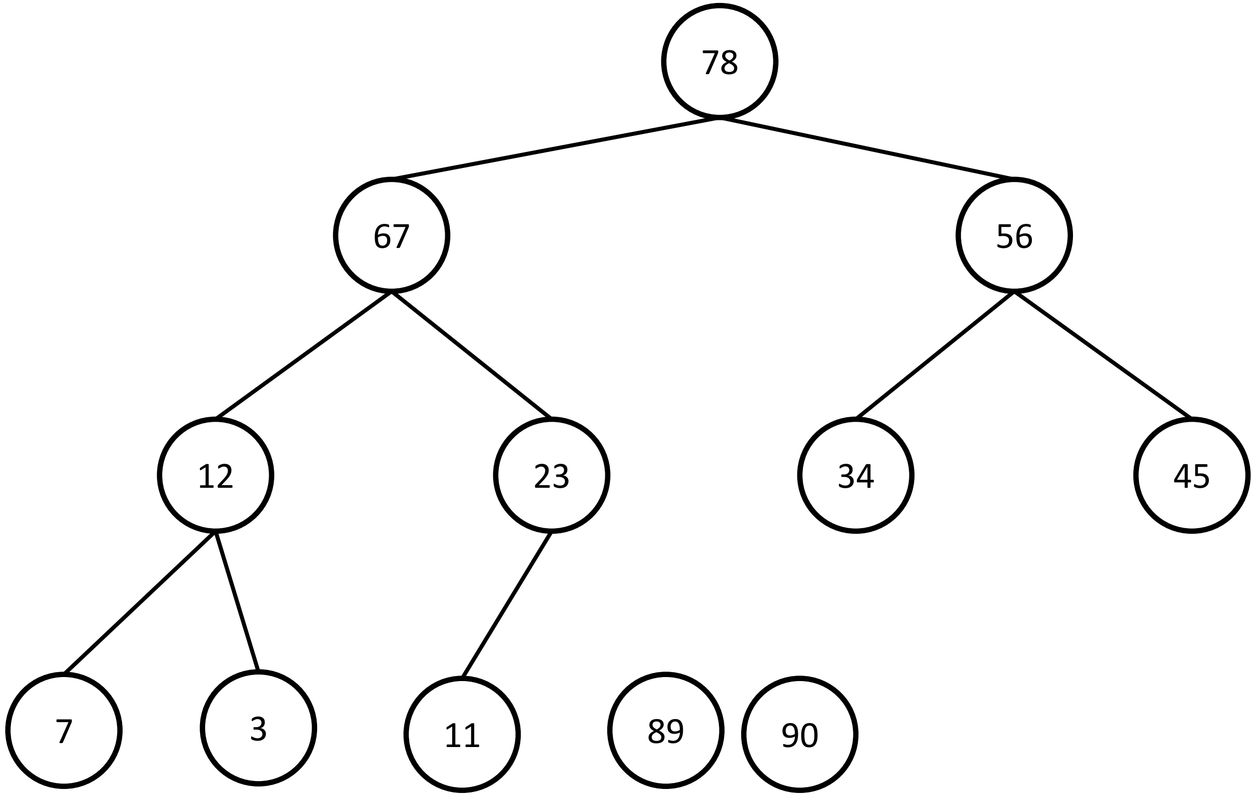
Heap Sort

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

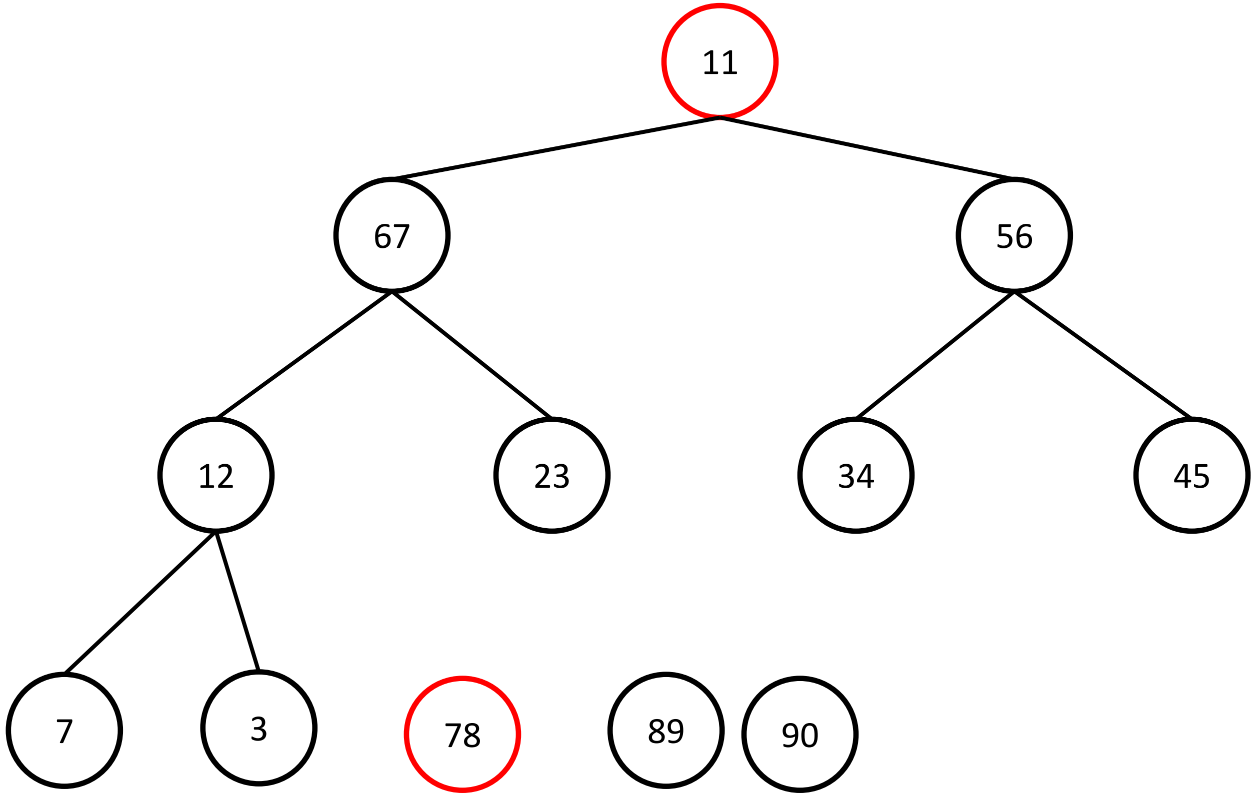
Heap Sort

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



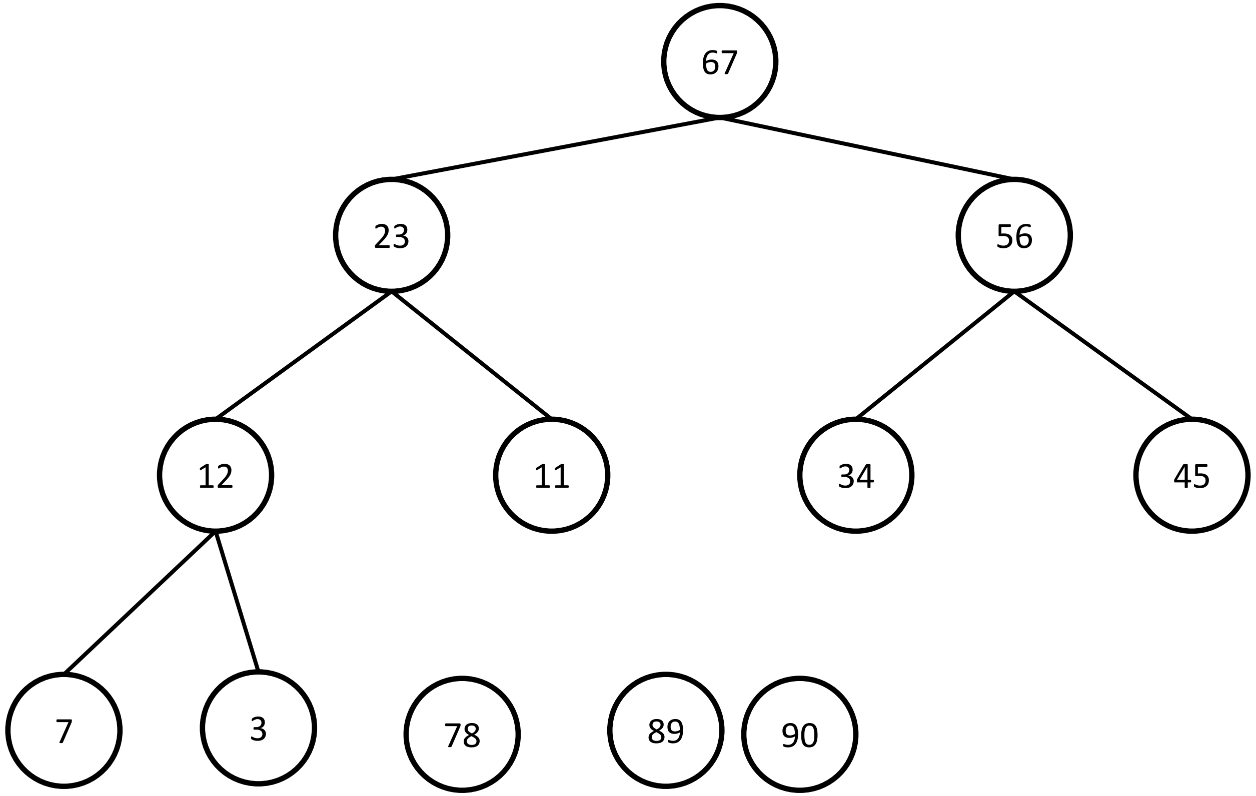
Heap Sort

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

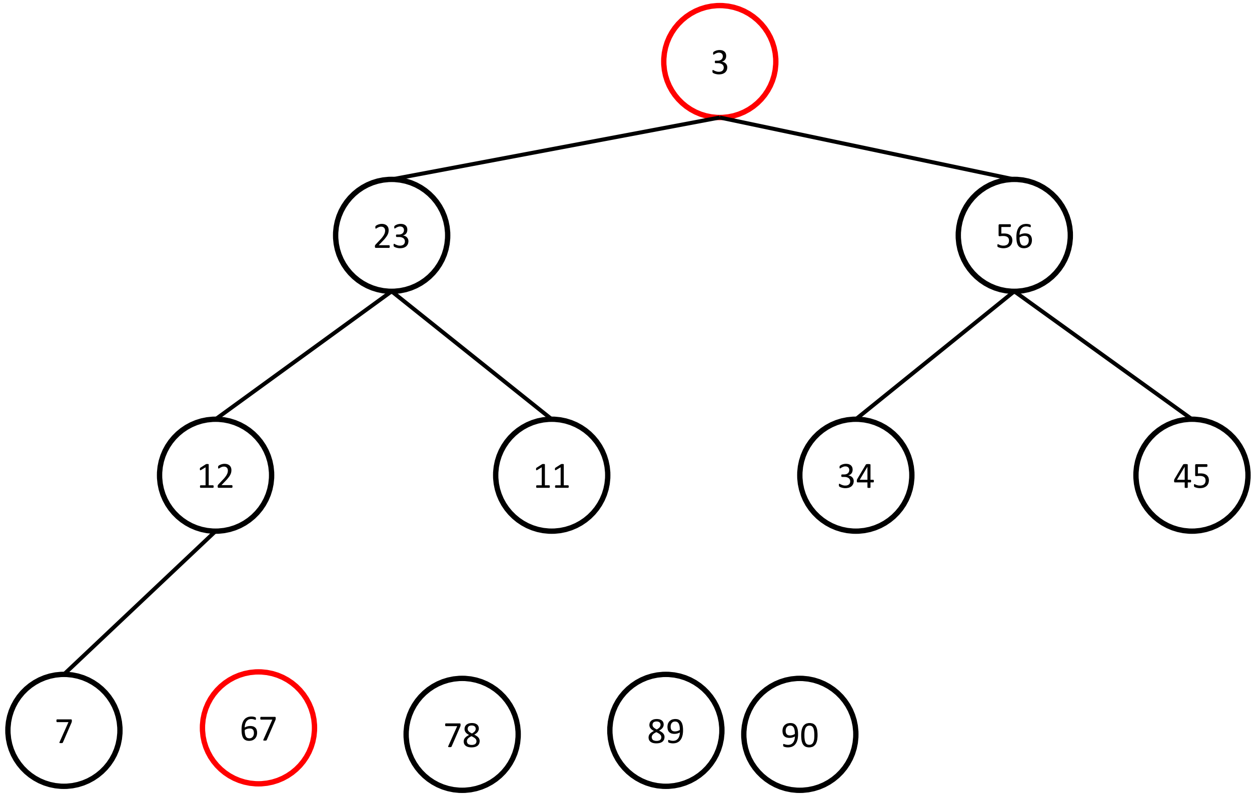
Heap Sort

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

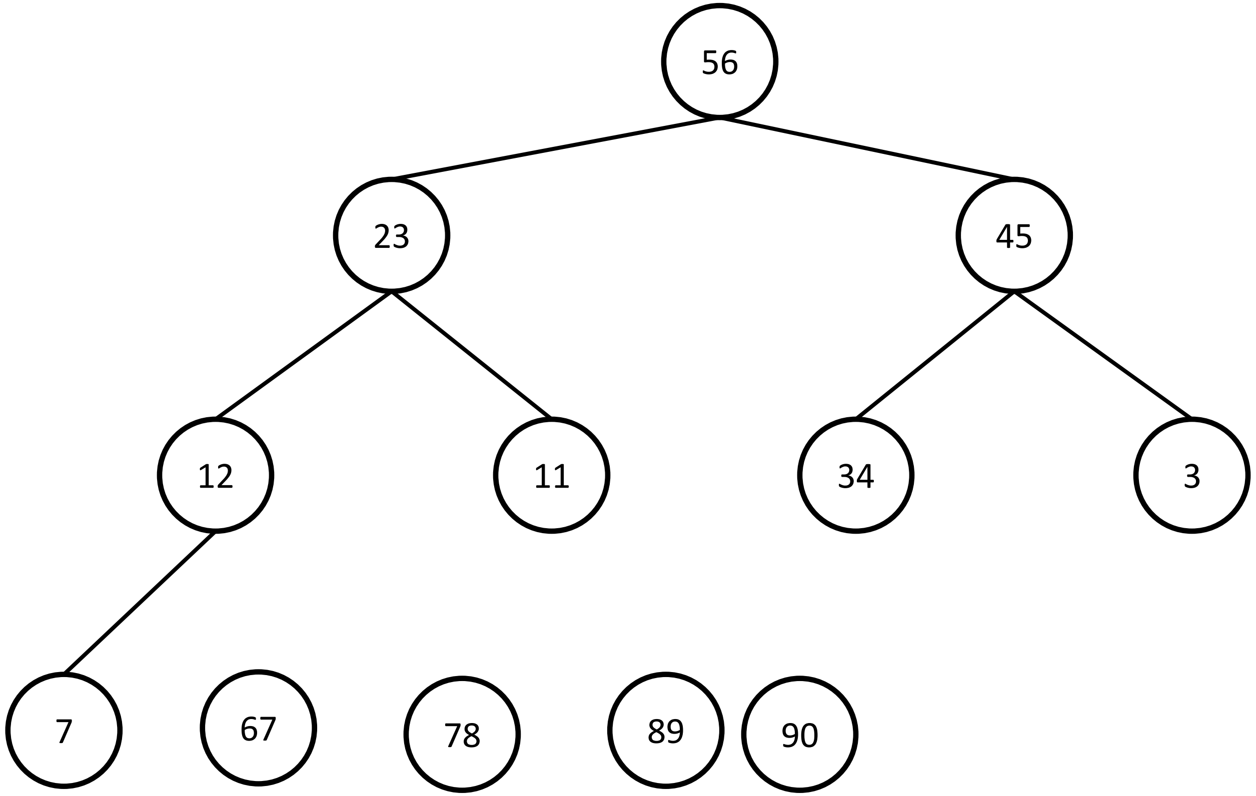
Heap Sort

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

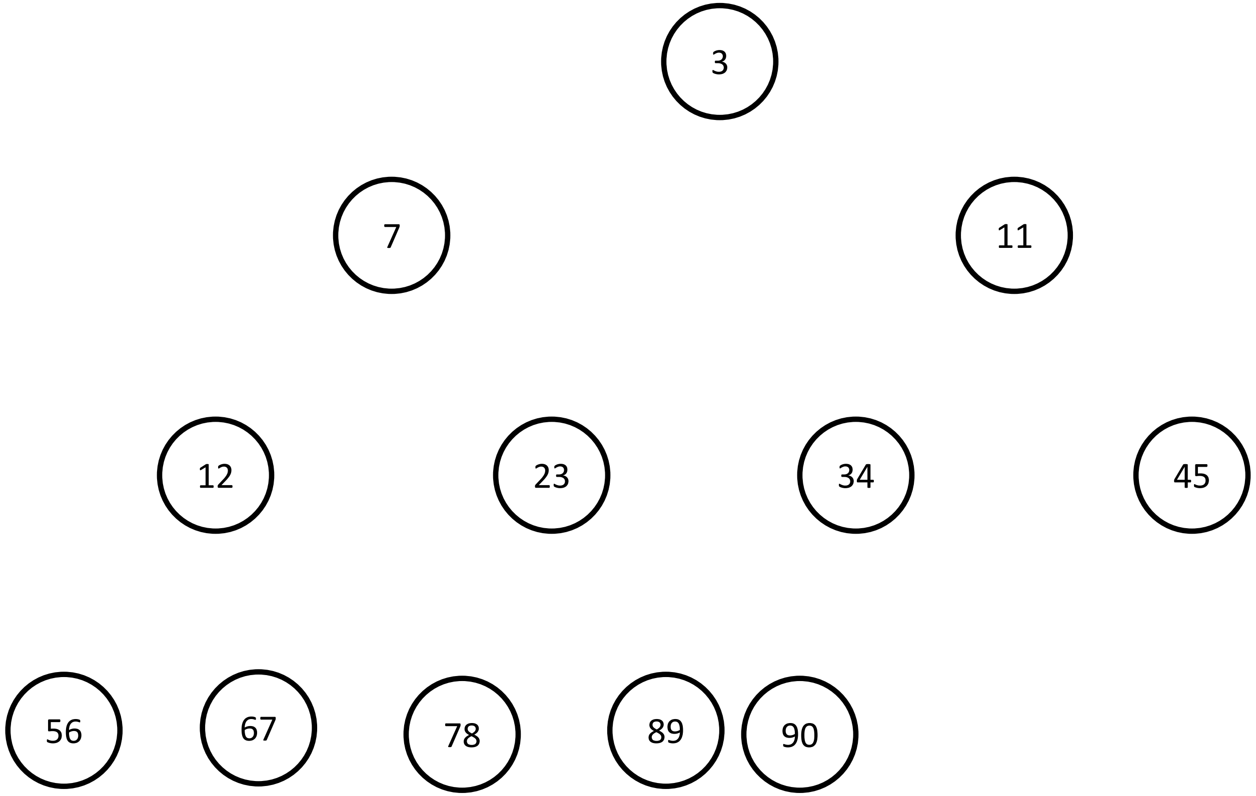
Heap Sort

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

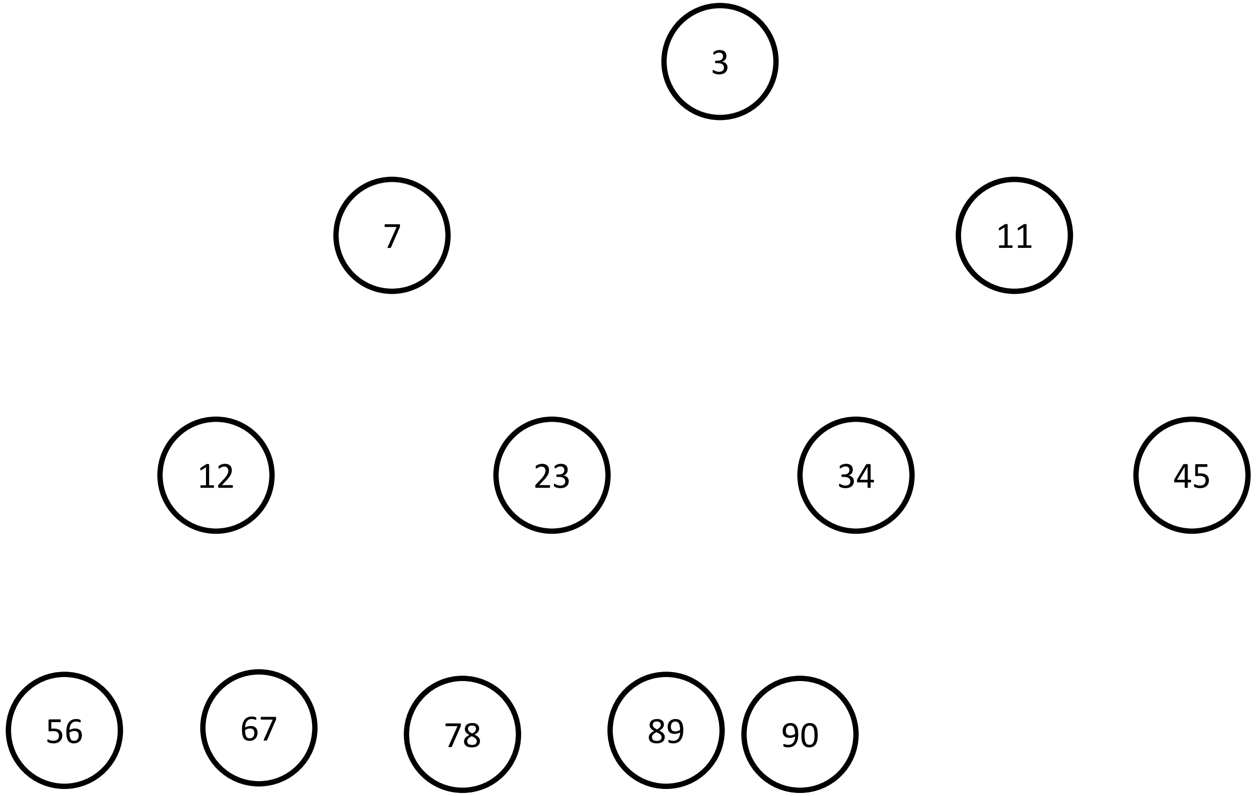
1. Build a **Max Heap** from the unsorted array

   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)

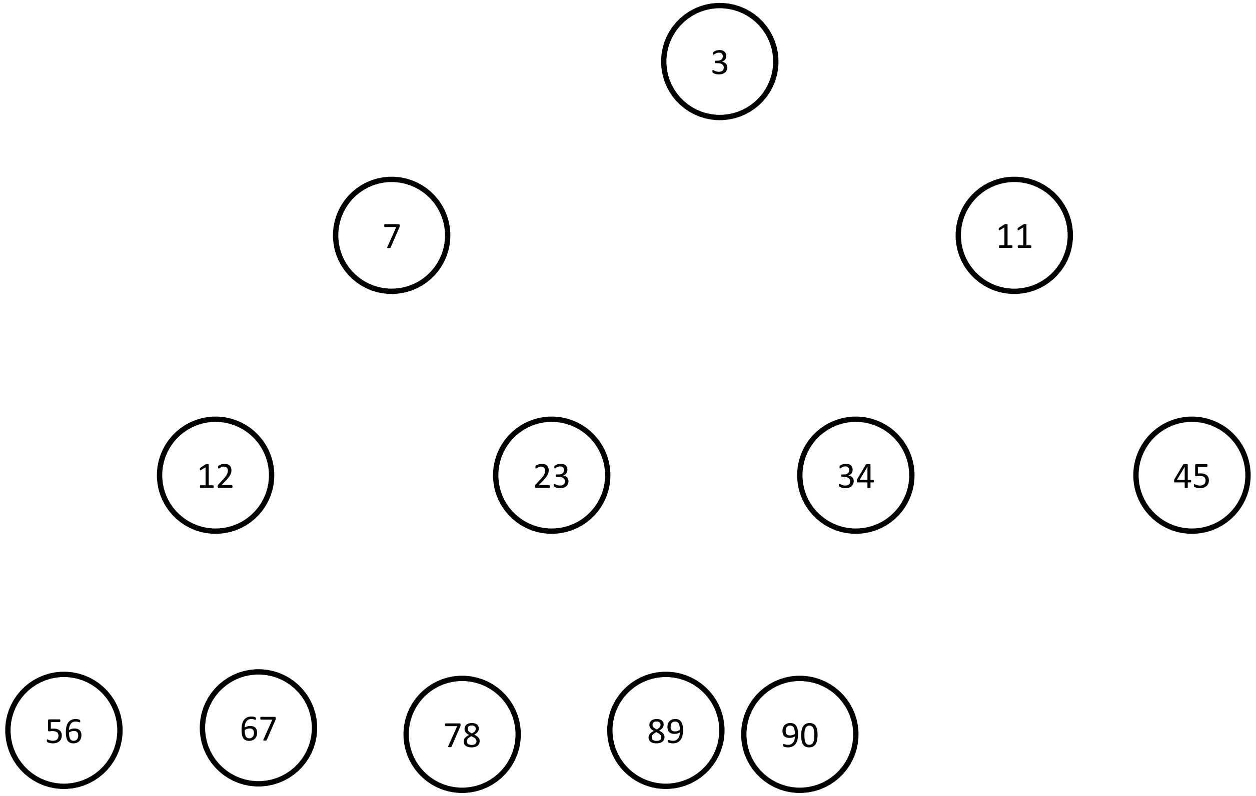


3	7	11	12	23	34	45	56	67	78	89	90

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