Priority Queues (PQs) with an interlude on heaps

William Fiset

Outline

- Discussion & Examples of PQs
  - What is a PQ?
  - What is a heap?
  - When and where is a PQ used?
  - How to turn a Min PQ into a Max PQ
  - Complexity Analysis
- Binary heap PQ Implementation Details
  - Heap sinking and swimming (also called sift down & sift up or bubble up & bubble down)
  - Adding elements to PQ
  - Removing (polling) elements from PQ
- Code Implementation

Discussion & Examples

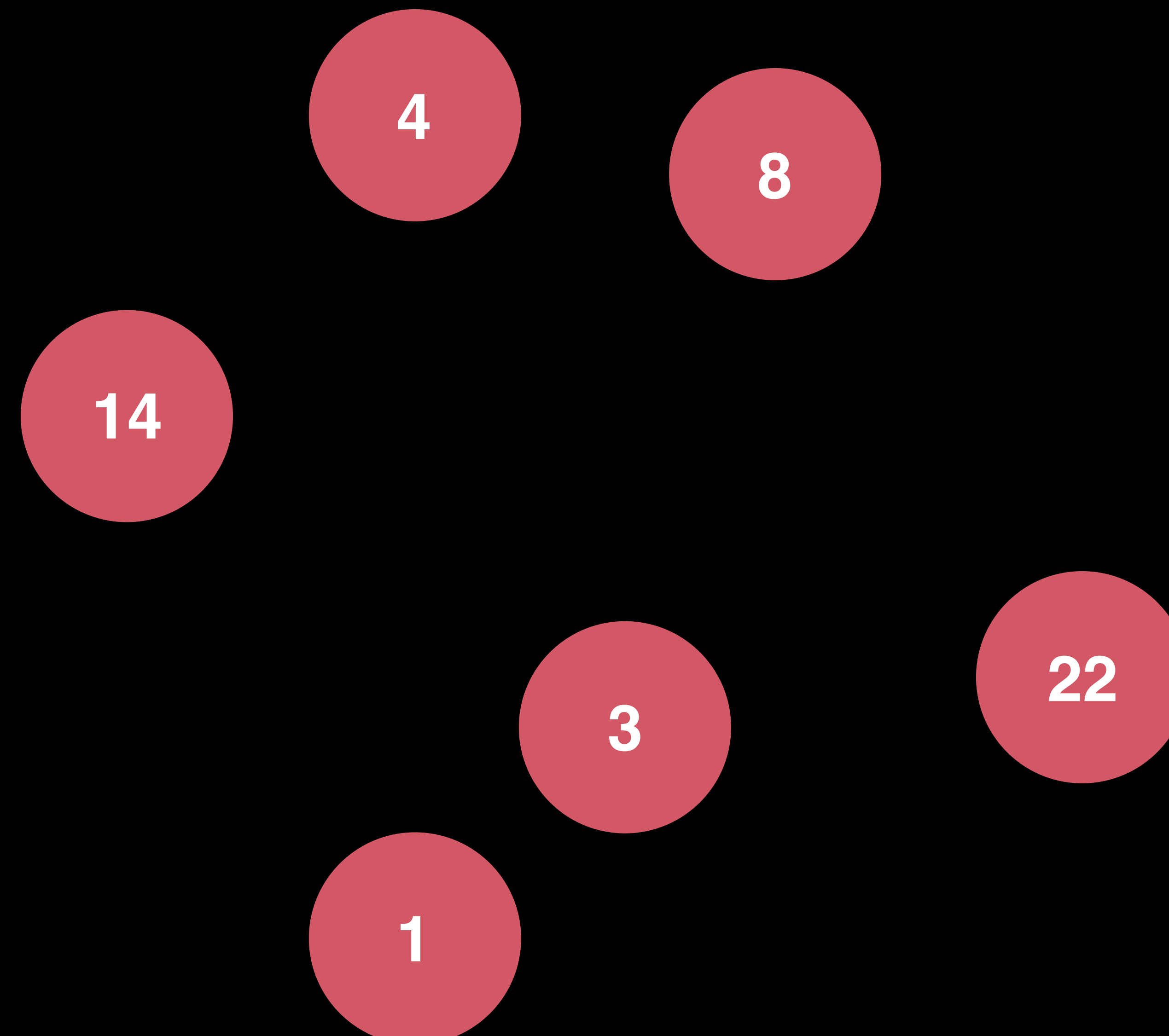
What is a Priority Queue?

A priority queue is an Abstract Data Type (ADT) that operates similar to a normal queue except that **each element has a certain priority**. The priority of the elements in the priority queue determine the order in which elements are removed from the PQ.

NOTE: Priority queues only supports **comparable data**, meaning the data inserted into the priority queue must be able to be ordered in some way either from least to greatest or greatest to least. This is so that we are able to assign relative priorities to each element.

What is a Priority Queue?

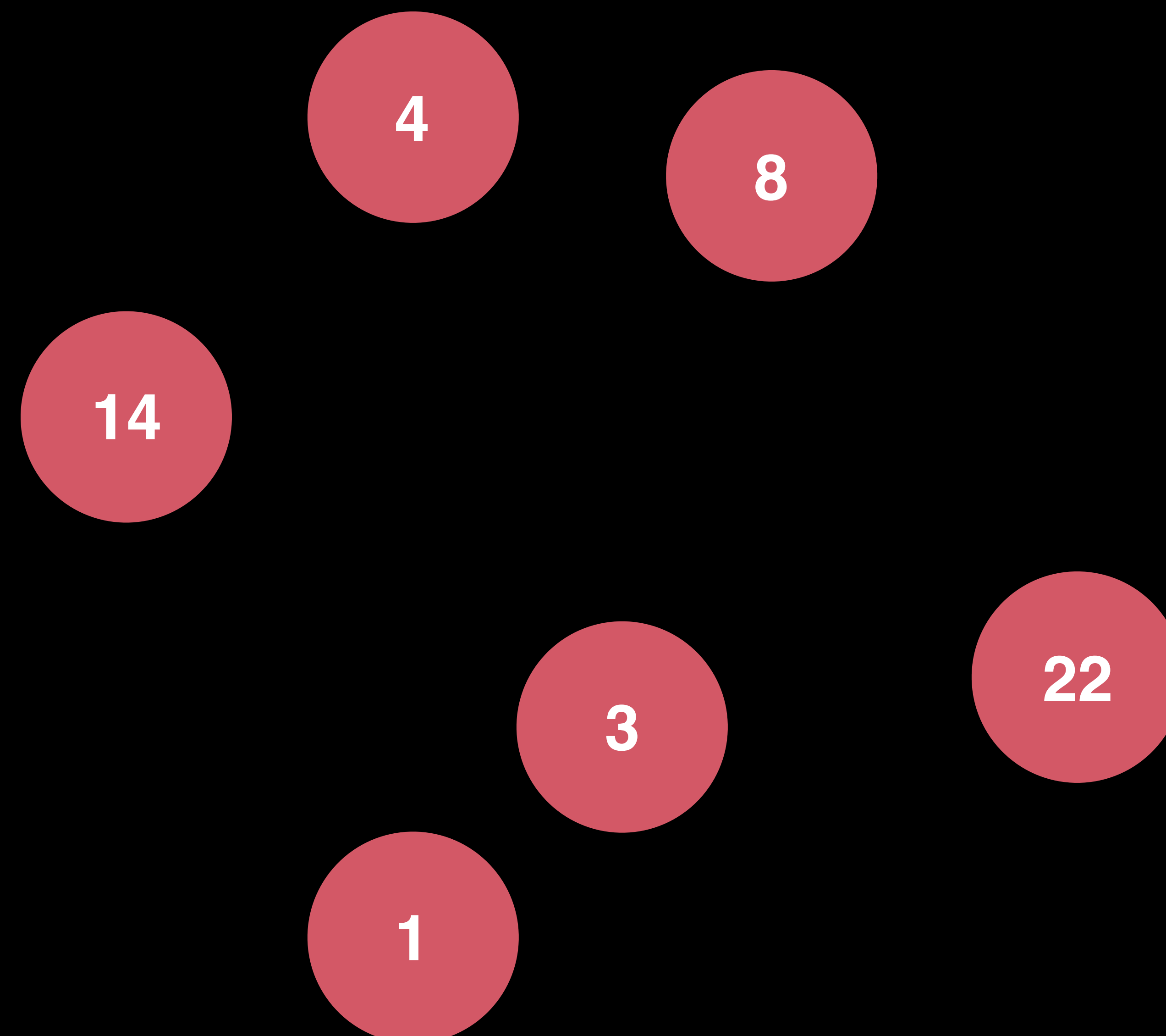
Suppose all these values are inserted into a PQ with an ordering imposed on the numbers to be from least to greatest.



What is a Priority Queue?

Instructions:

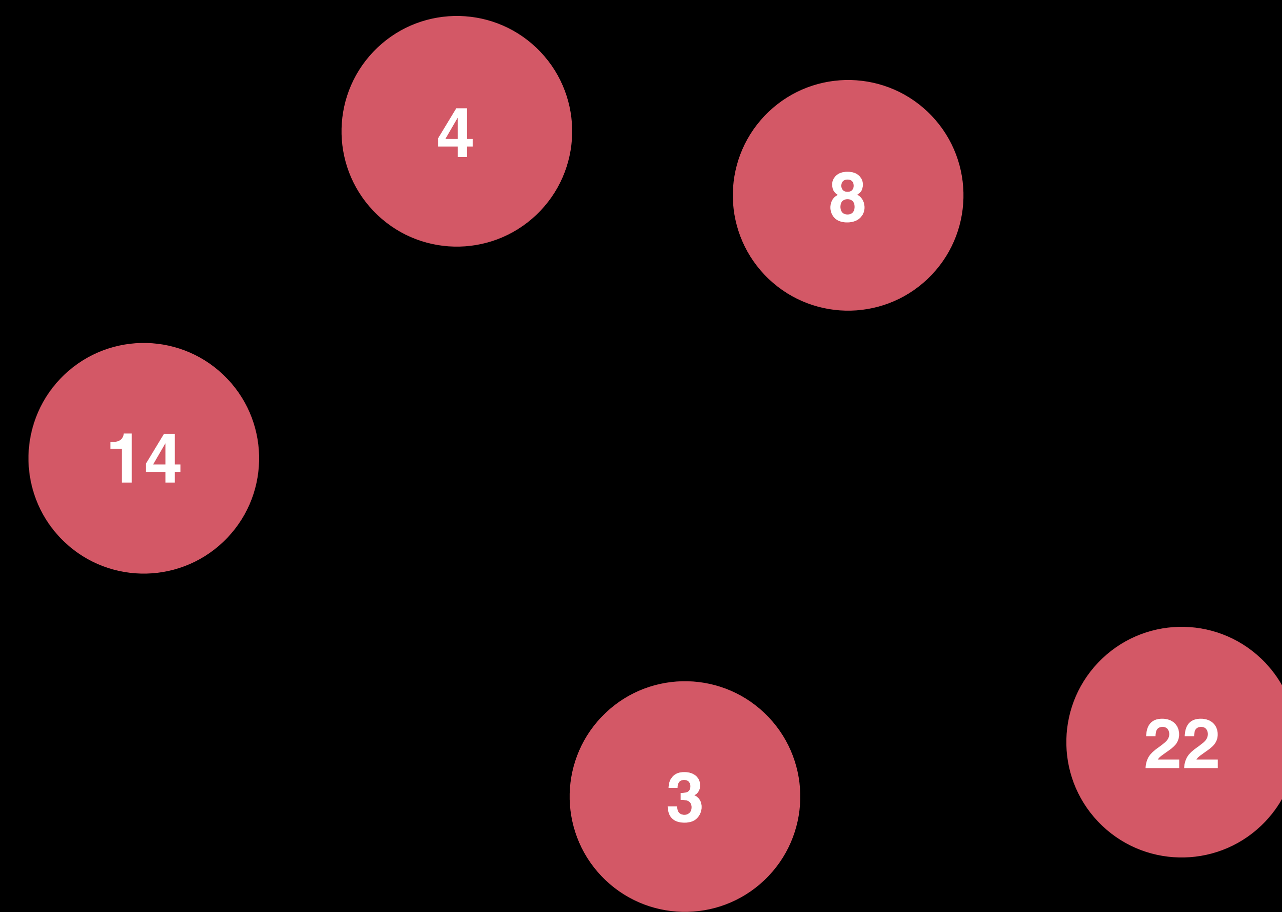
```
poll()
add(2)
poll()
add(4)
poll()
add(5)
add(9)
poll rest
```



What is a Priority Queue?

Instructions:

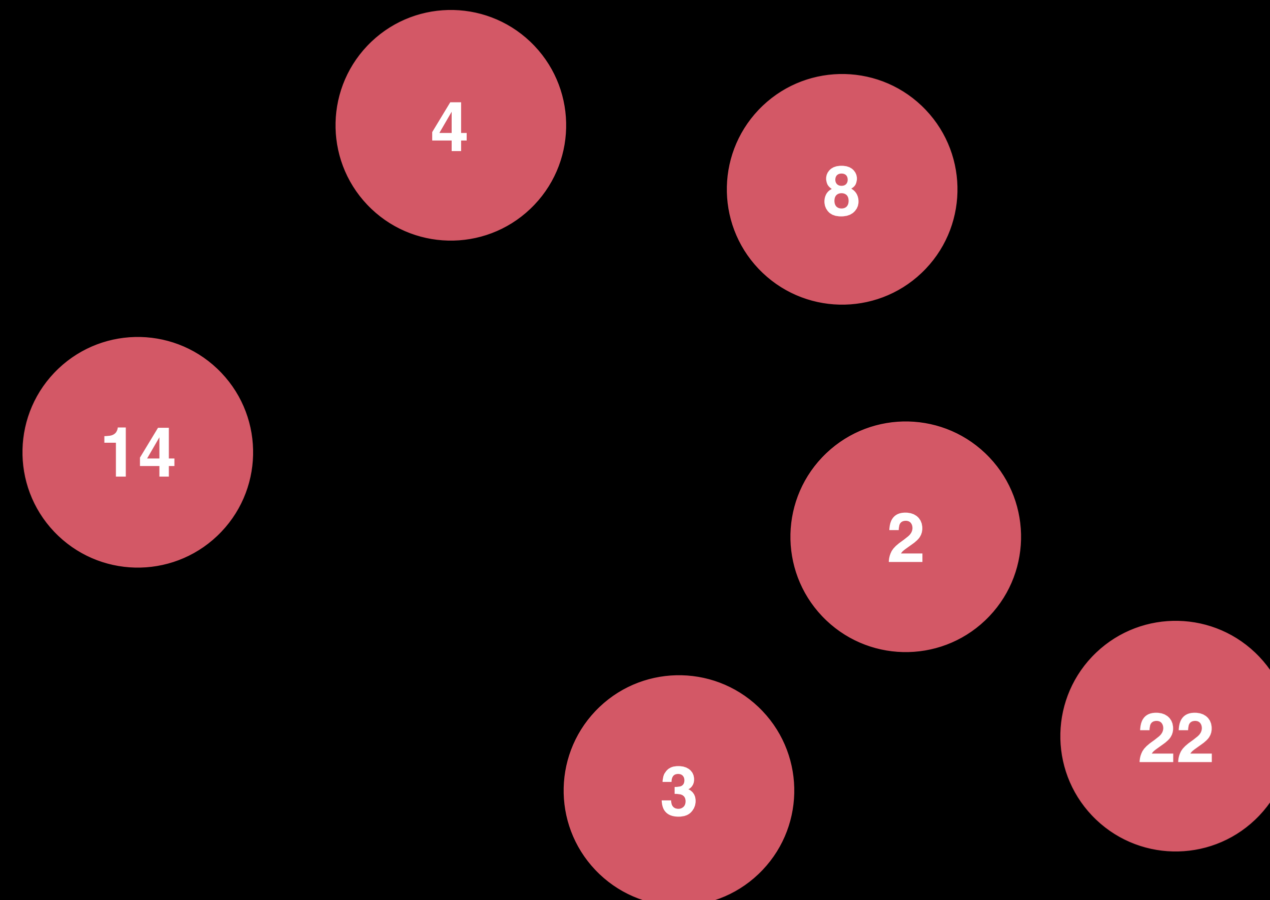
```
poll()
add(2)
poll()
add(4)
poll()
add(5)
add(9)
poll rest
```



What is a Priority Queue?

Instructions:

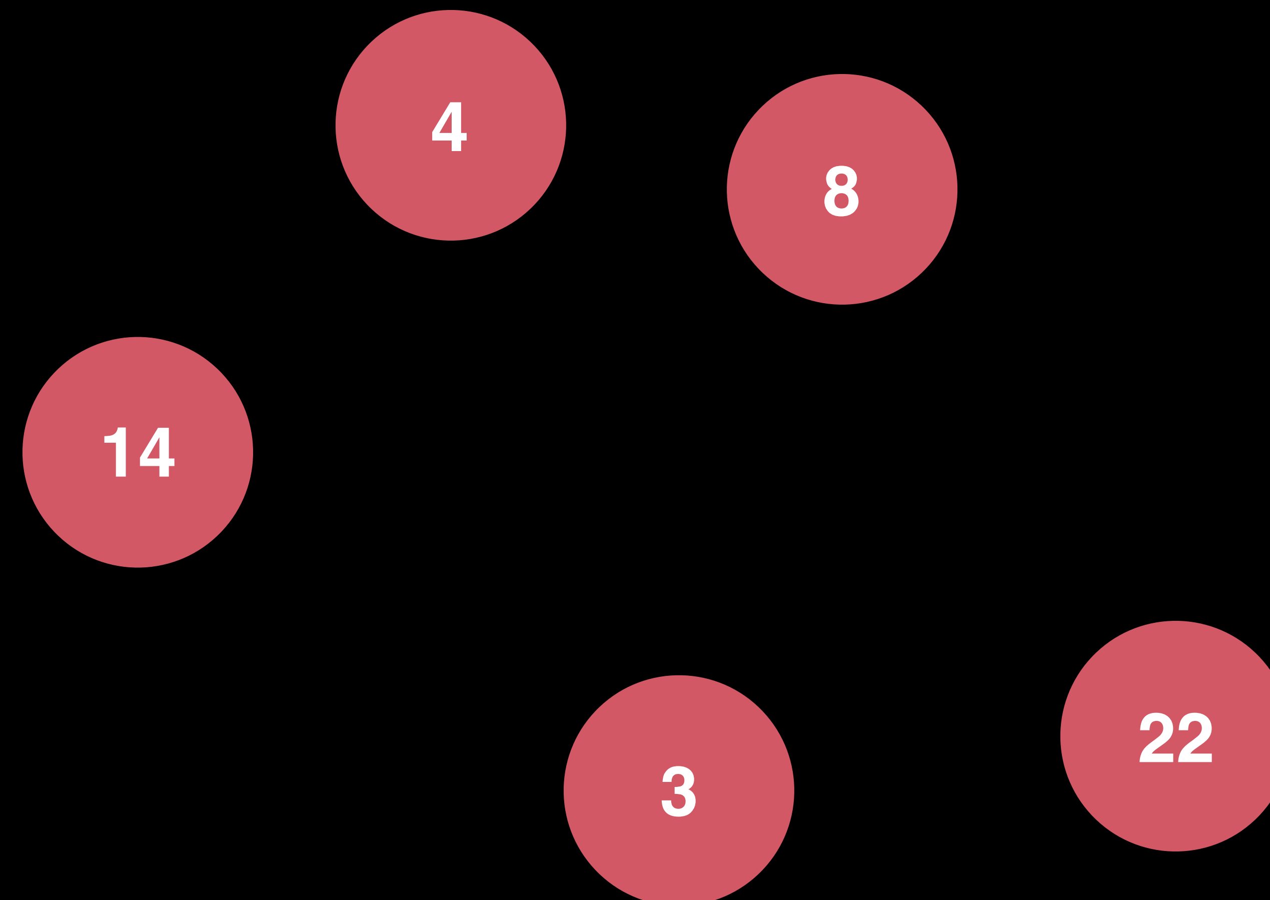
```
poll()
add(2)
poll()
add(4)
poll()
add(5)
add(9)
poll rest
```

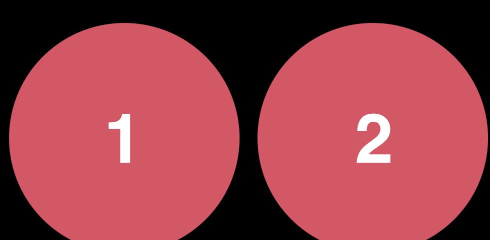


What is a Priority Queue?

Instructions:

```
poll()
add(2)
poll()
add(4)
poll()
add(5)
add(9)
poll rest
```

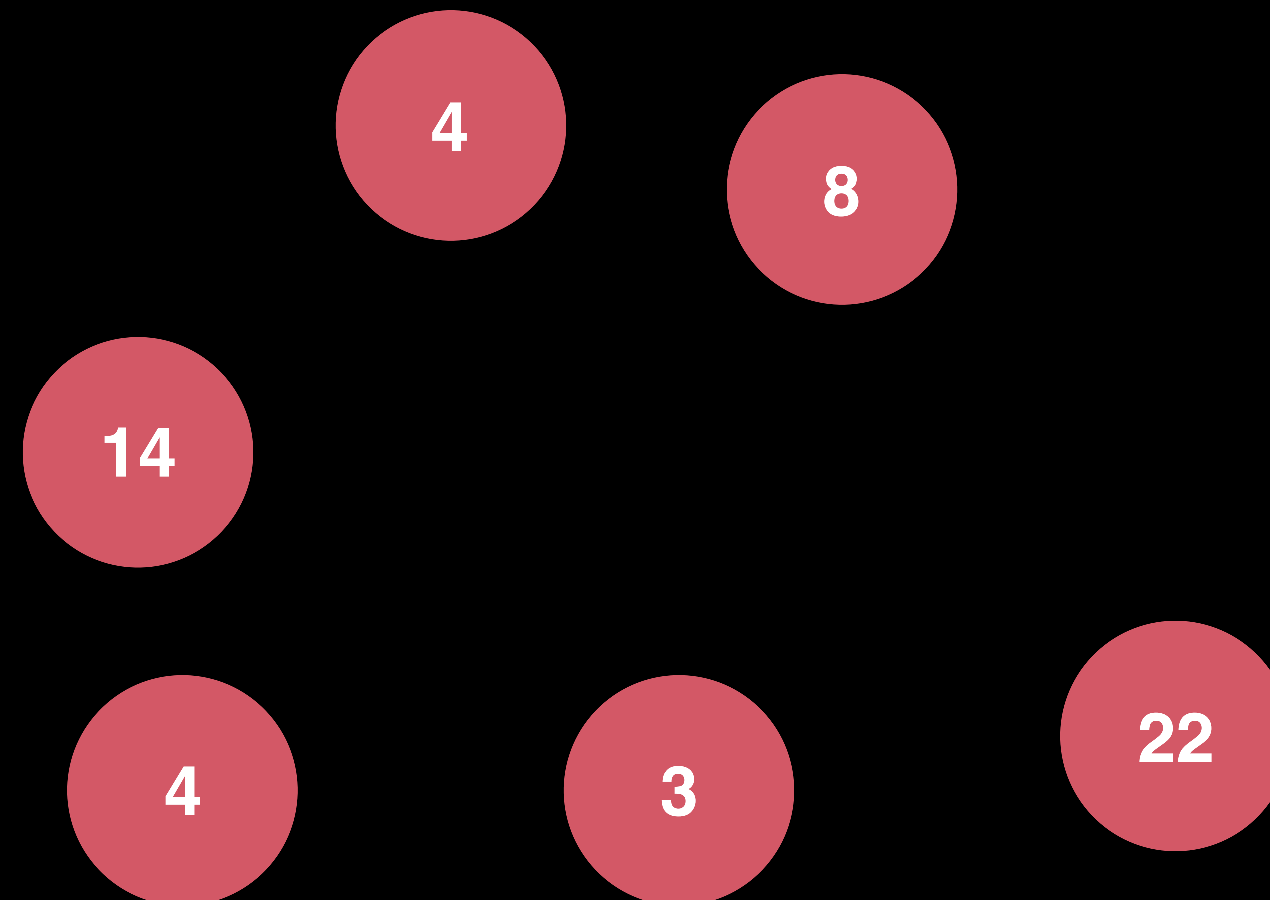




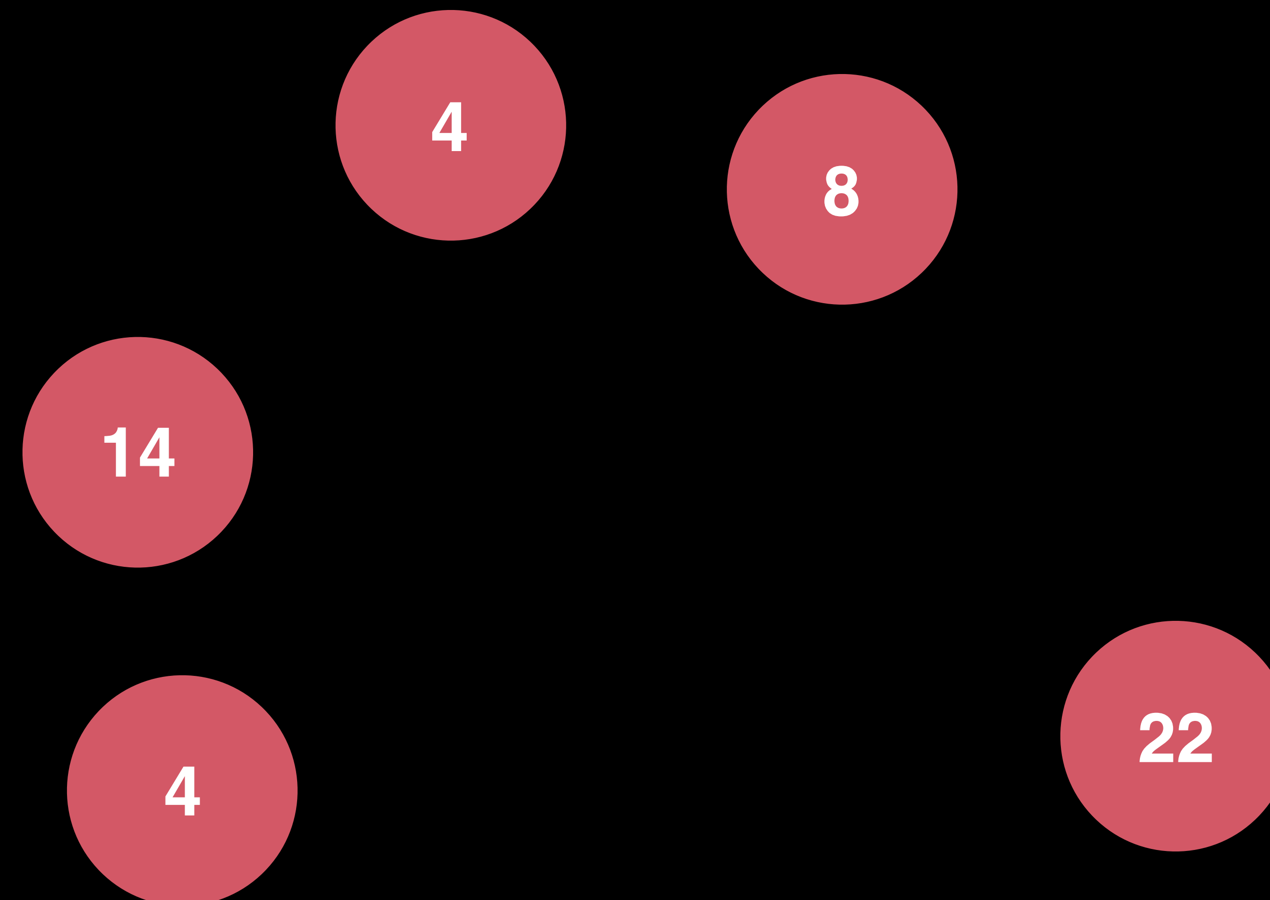
What is a Priority Queue?

Instructions:

```
poll()
add(2)
poll()
add(4)
poll()
add(5)
add(9)
poll rest
```



What is a Priority Queue?



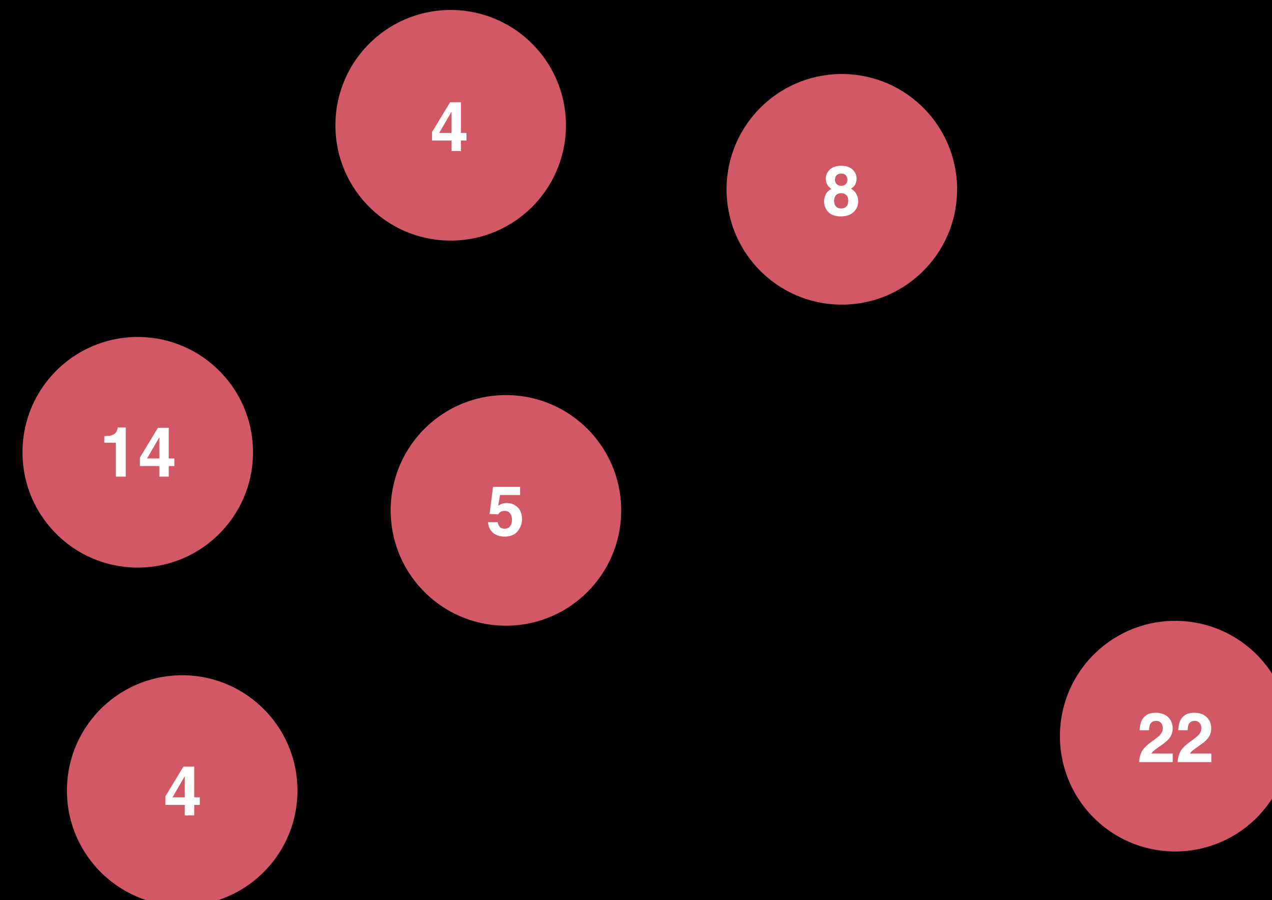
Instructions:

```
poll()
add(2)
poll()
add(4)
poll()
add(5)
add(9)
poll rest
```

What is a Priority Queue?

Instructions:

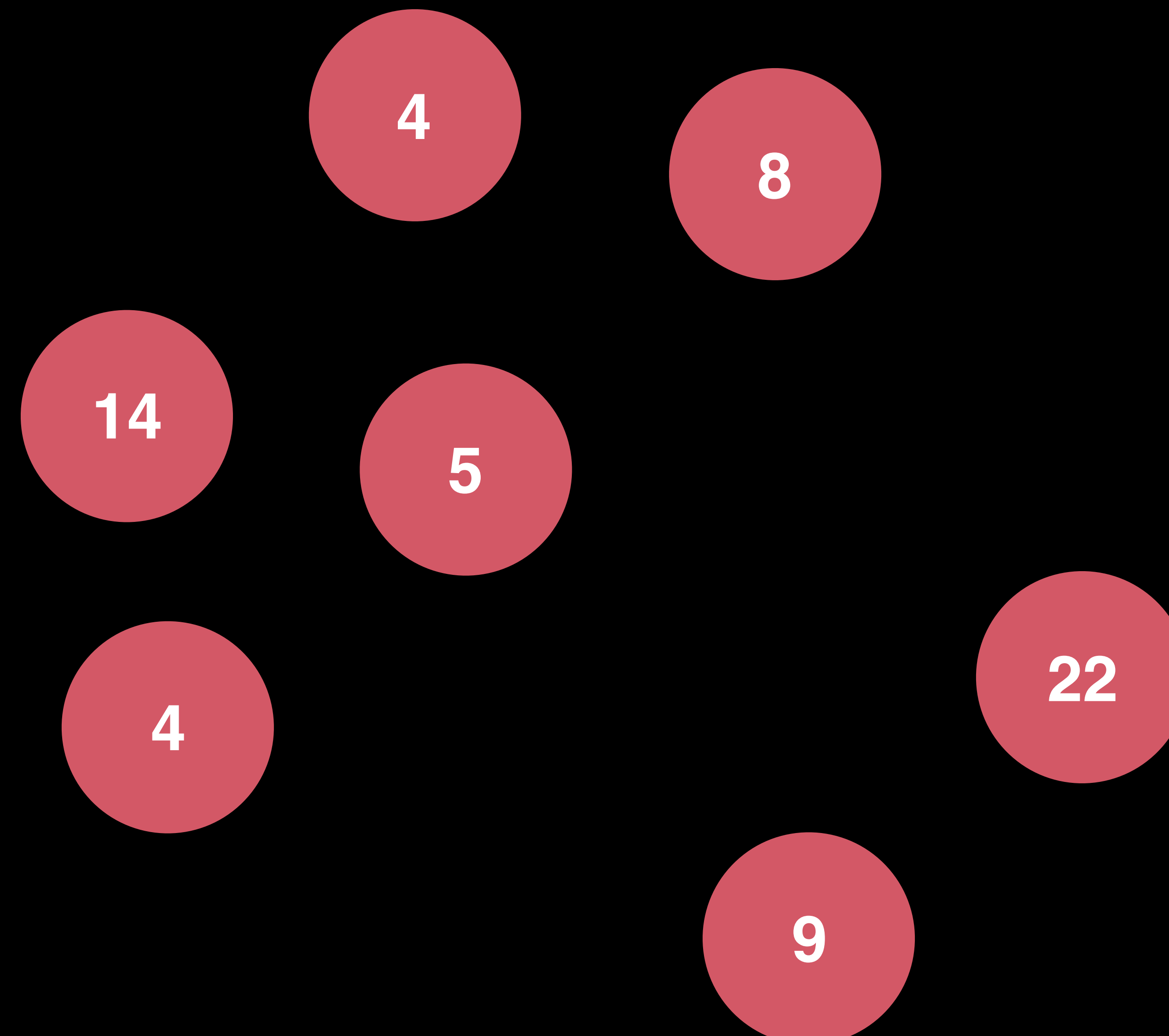
```
poll()
add(2)
poll()
add(4)
poll()
add(5)
add(9)
poll rest
```



What is a Priority Queue?

Instructions:

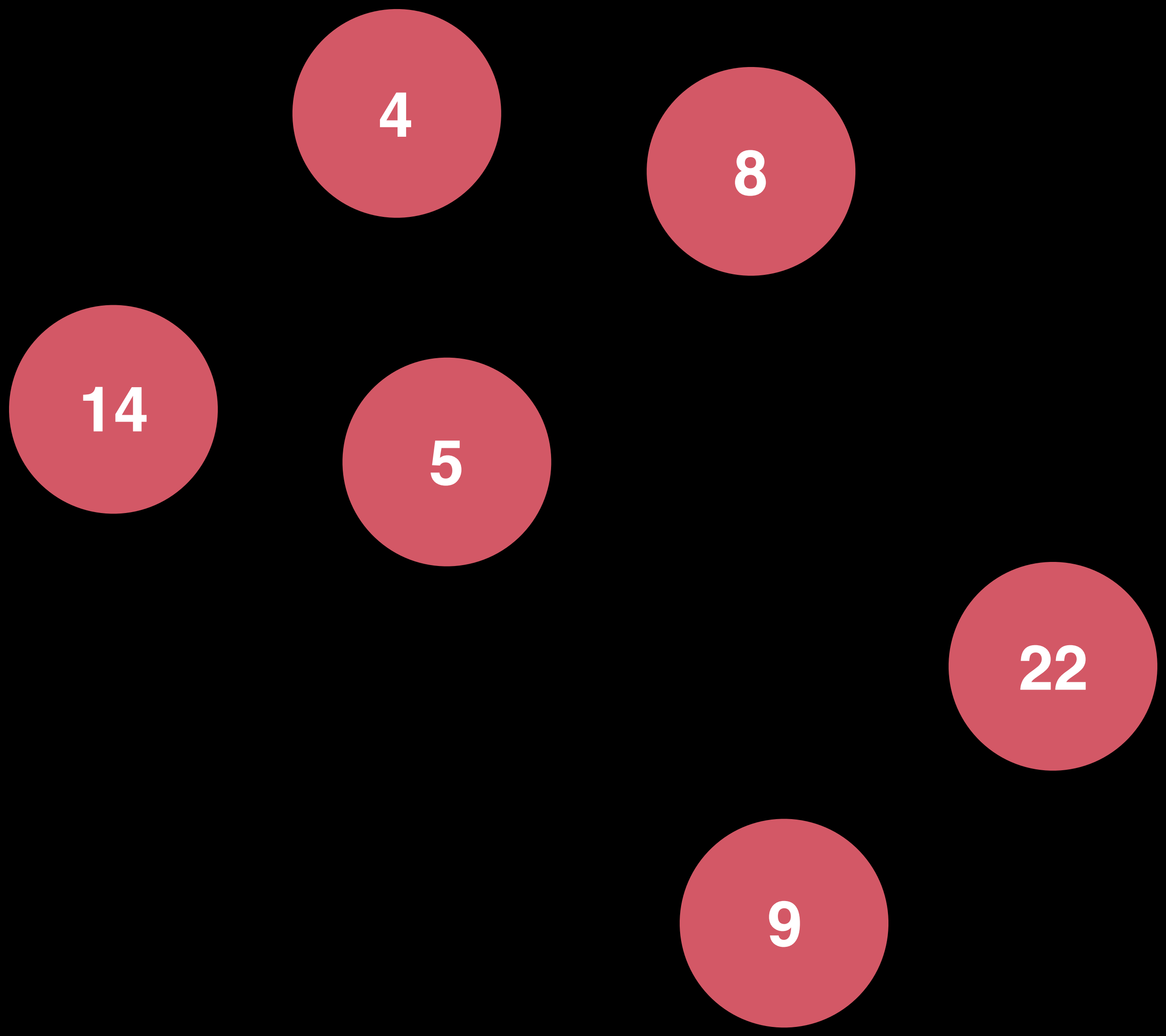
```
poll()
add(2)
poll()
add(4)
poll()
add(5)
add(9)
poll rest
```



What is a Priority Queue?

Instructions:

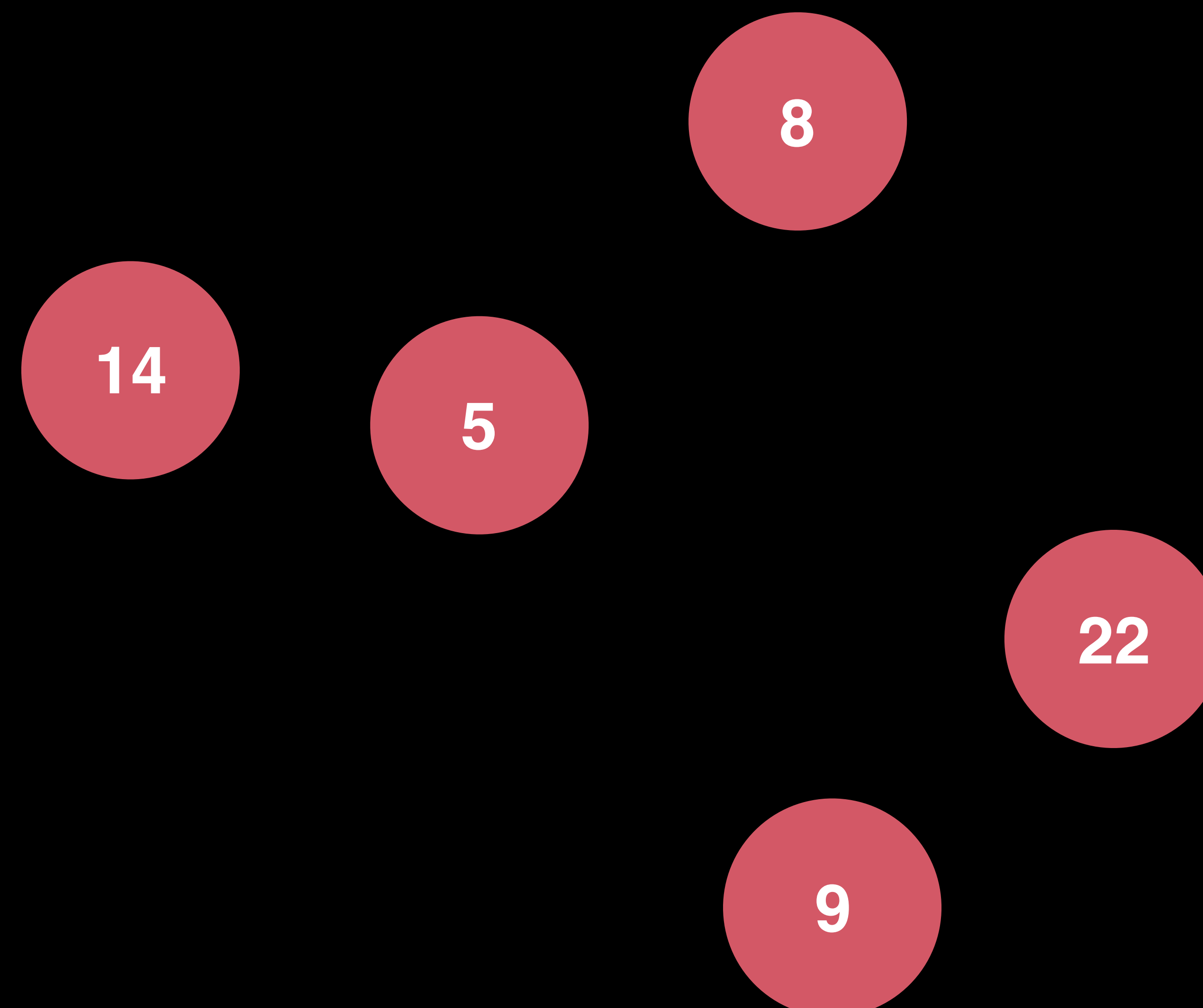
```
poll()
add(2)
poll()
add(4)
poll()
add(5)
add(9)
poll rest
```



What is a Priority Queue?

Instructions:

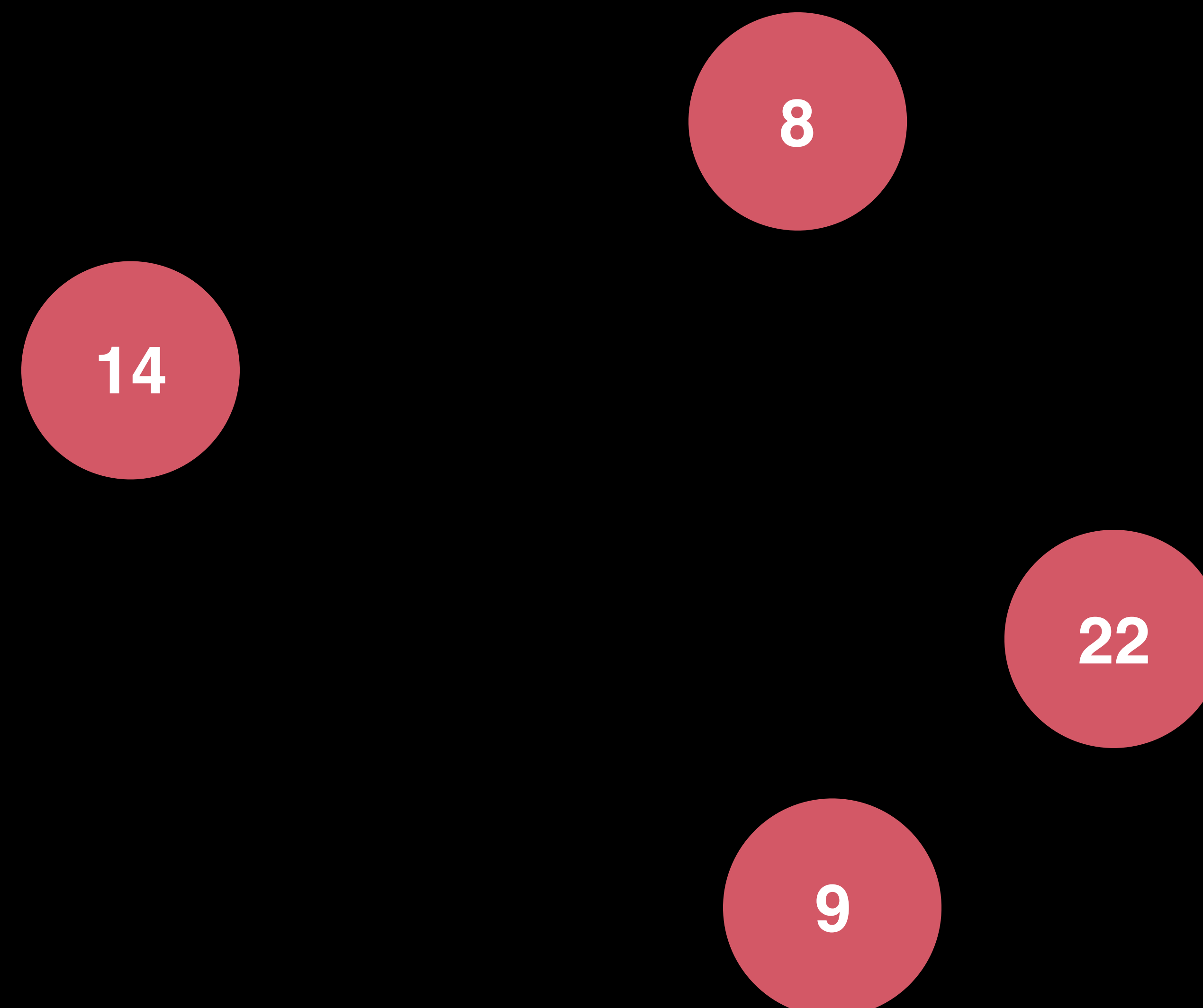
```
poll()
add(2)
poll()
add(4)
poll()
add(5)
add(9)
poll rest
```

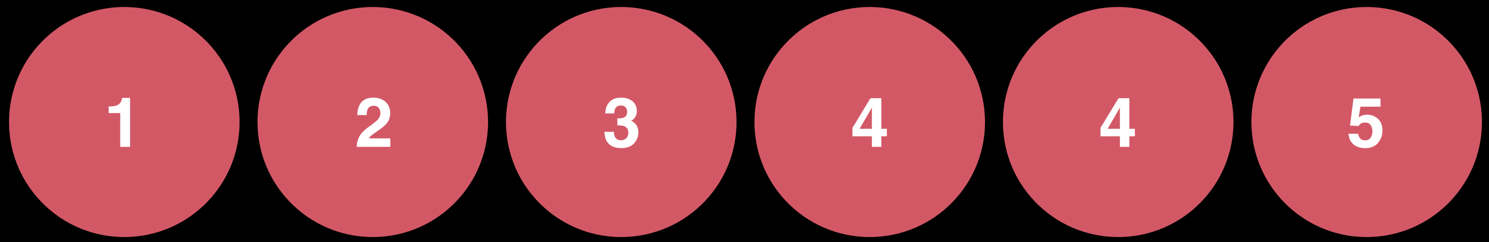


What is a Priority Queue?

Instructions:

```
poll()
add(2)
poll()
add(4)
poll()
add(5)
add(9)
poll rest
```

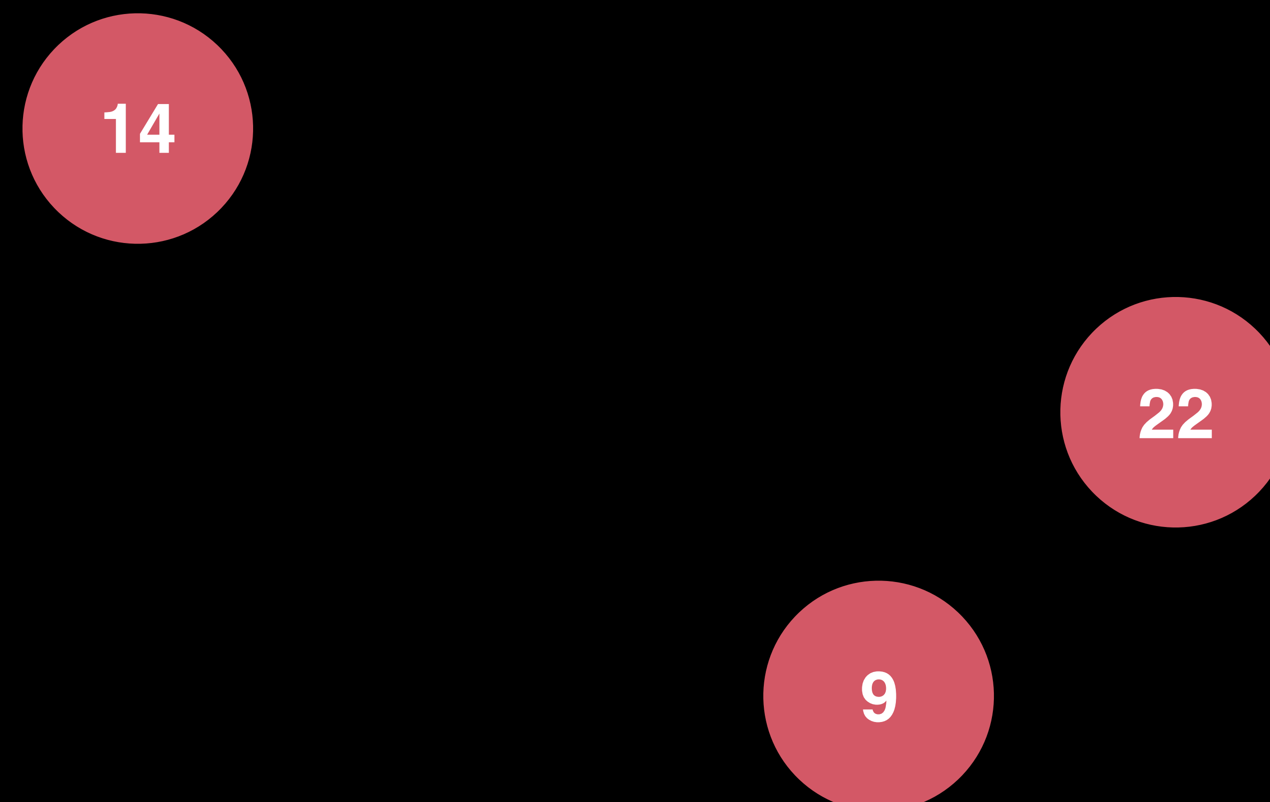




What is a Priority Queue?

Instructions:

```
poll()
add(2)
poll()
add(4)
poll()
add(5)
add(9)
poll rest
```



What is a Priority Queue?

Instructions:

```
poll()
add(2)
poll()
add(4)
poll()
add(5)
add(9)
poll rest
```

14

22

1 2 3 4 4 5 8 9

What is a Priority Queue?

Instructions:

```
poll()
add(2)
poll()
add(4)
poll()
add(5)
add(9)
poll rest
```

22

1 2 3 4 4 5 8 9 14

What is a Priority Queue?

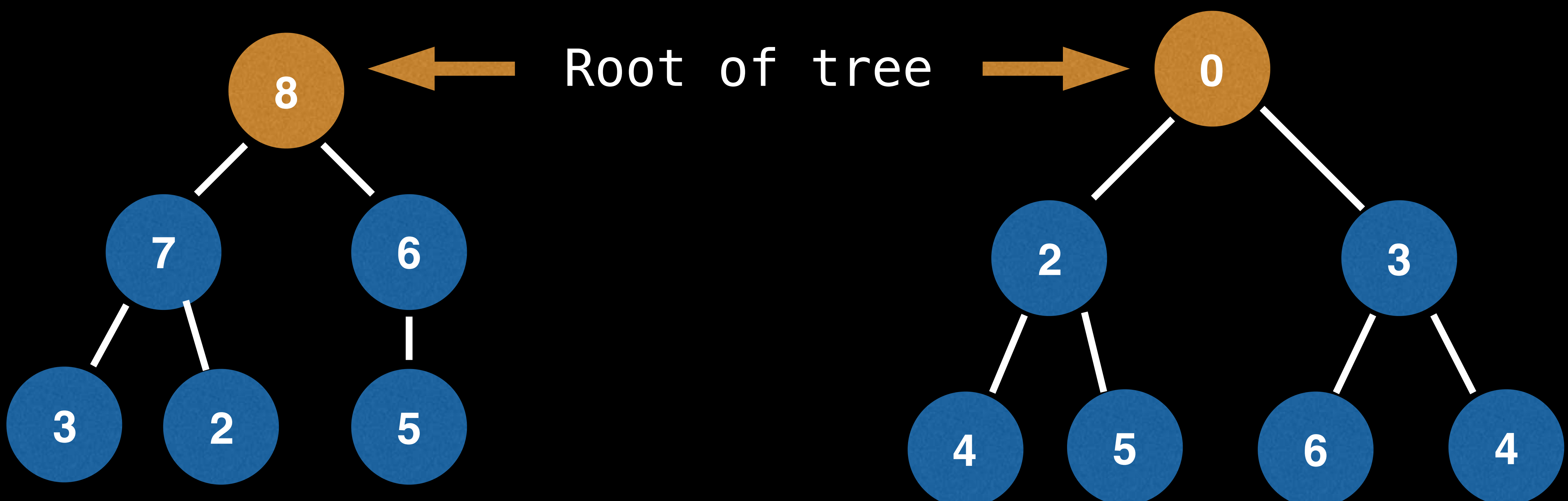
Instructions:

```
poll()
add(2)
poll()
add(4)
poll()
add(5)
add(9)
poll rest
```

1 2 3 4 4 5 8 9 14 22

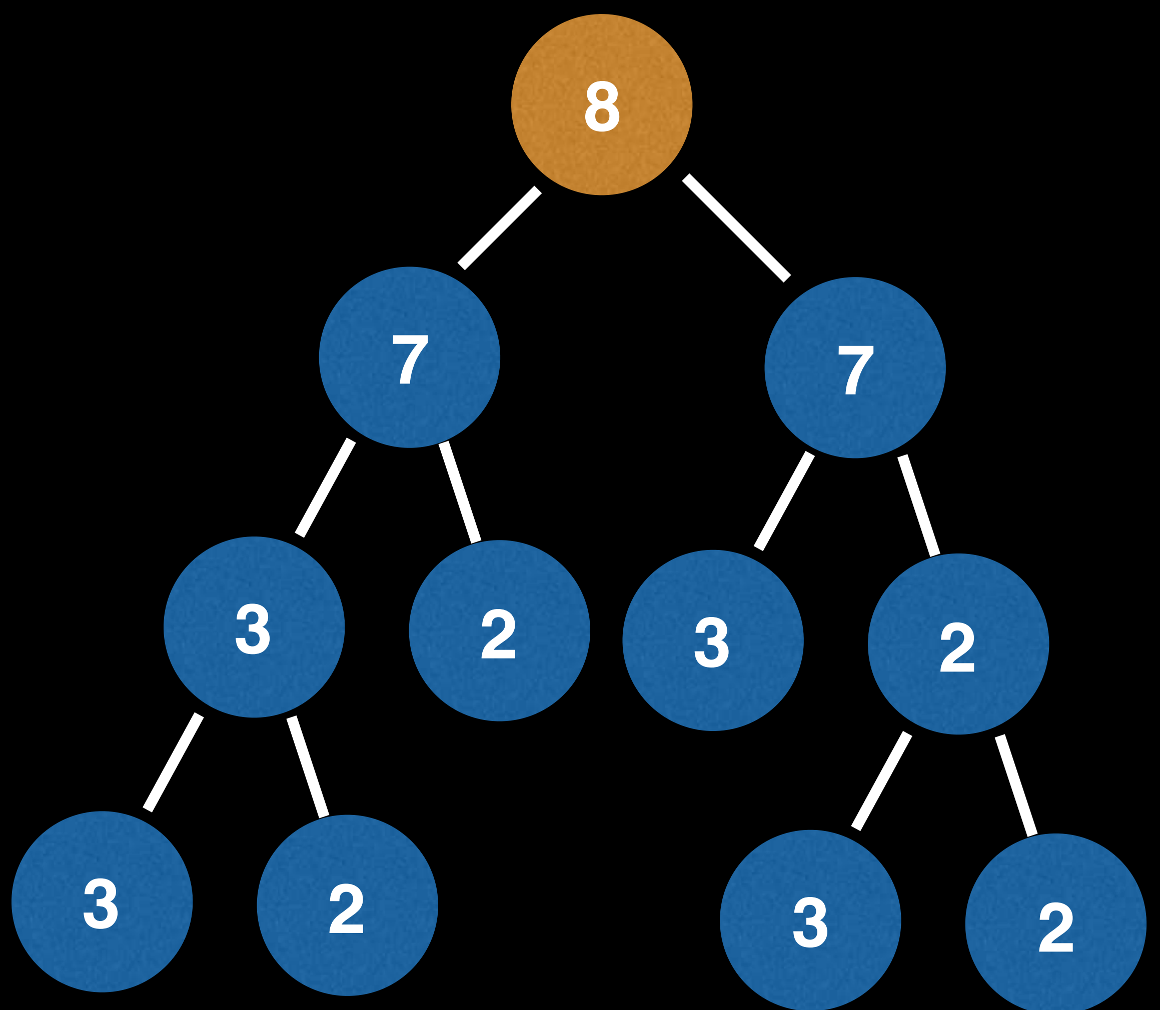
What is a Heap?

A heap is a **tree** based DS that satisfies the **heap invariant** (also called heap property): If A is a parent node of B then A is ordered with respect to B for all nodes A, B in the heap.

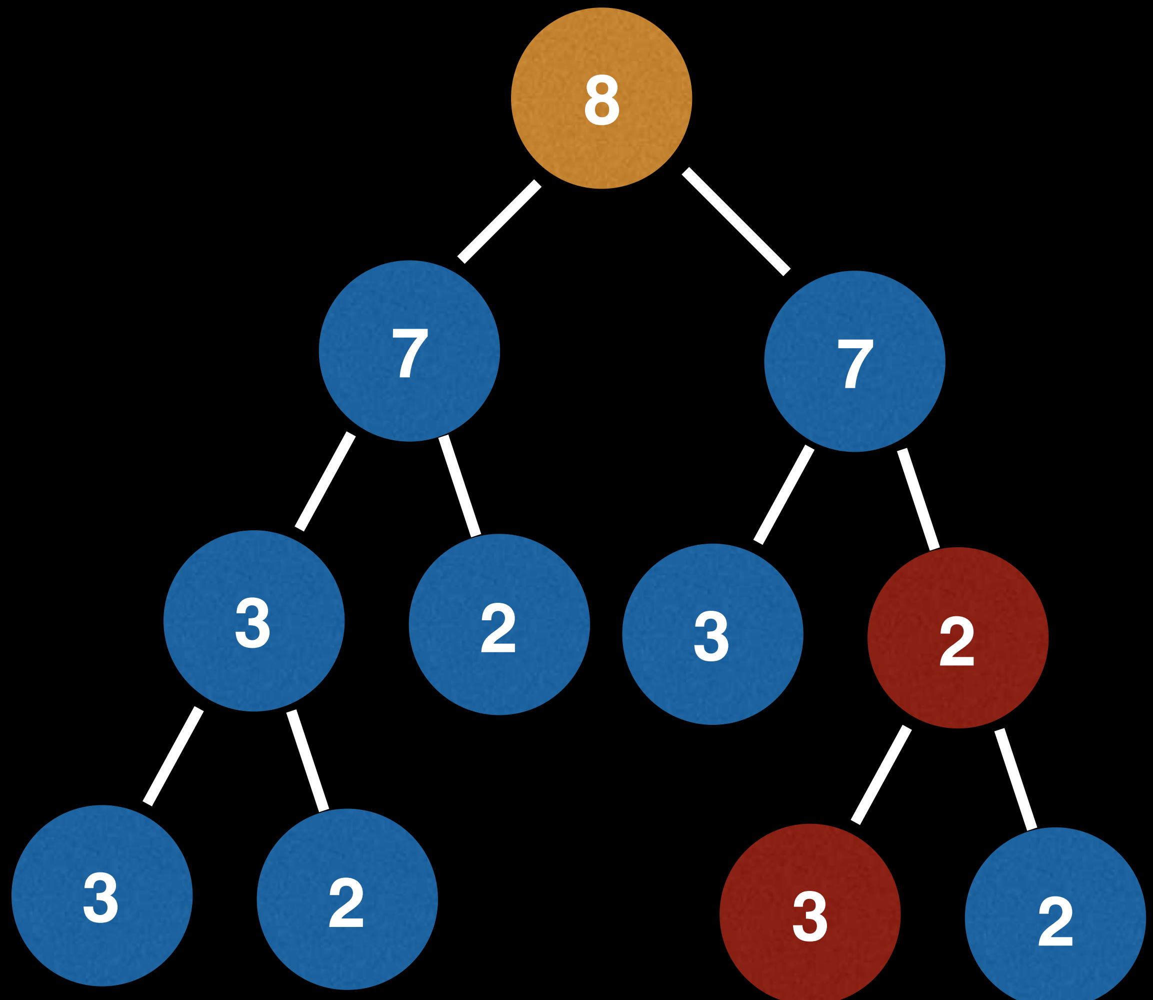


Max Heap **Min Heap**

Is this a valid heap?

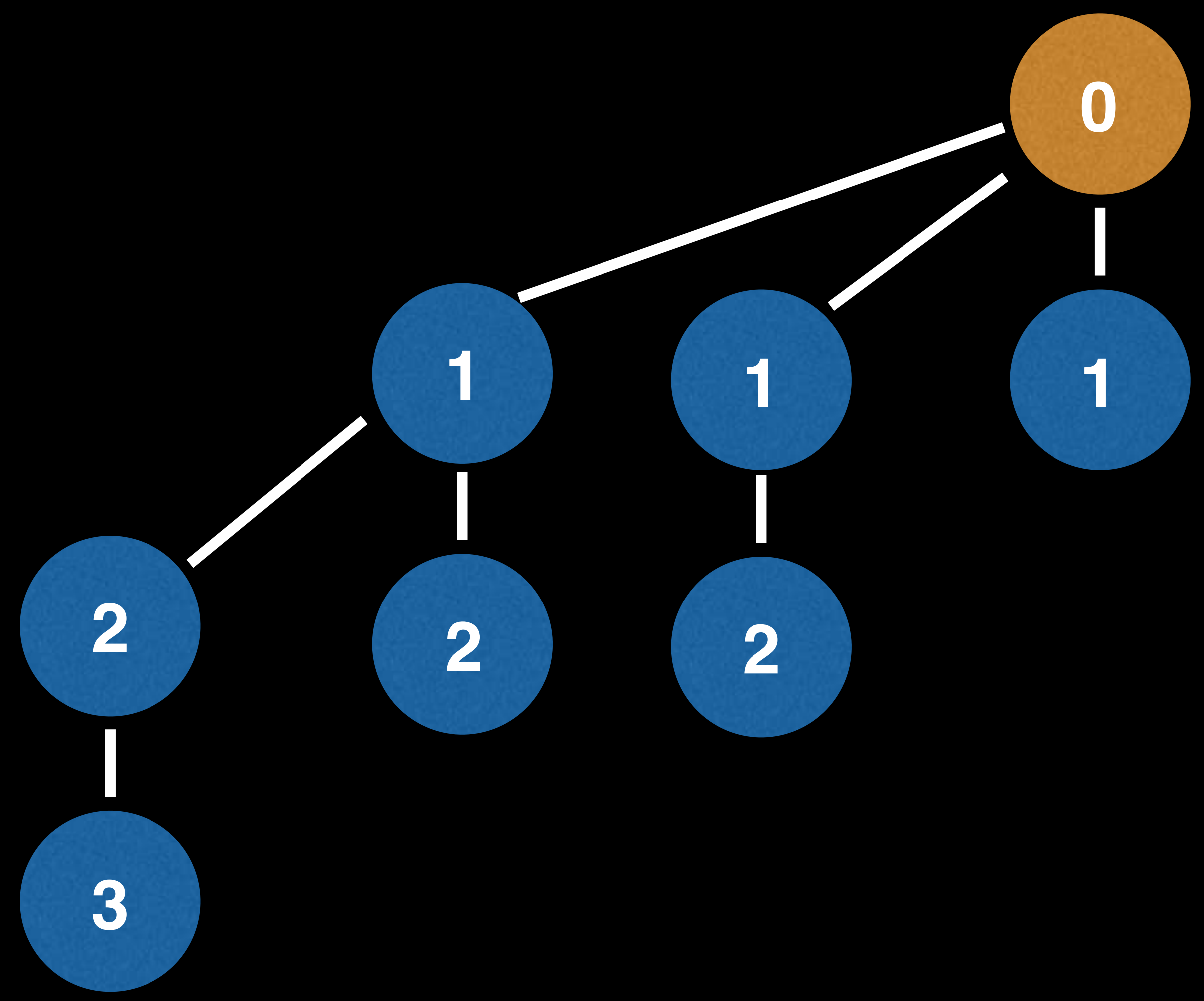


Is this a valid heap?

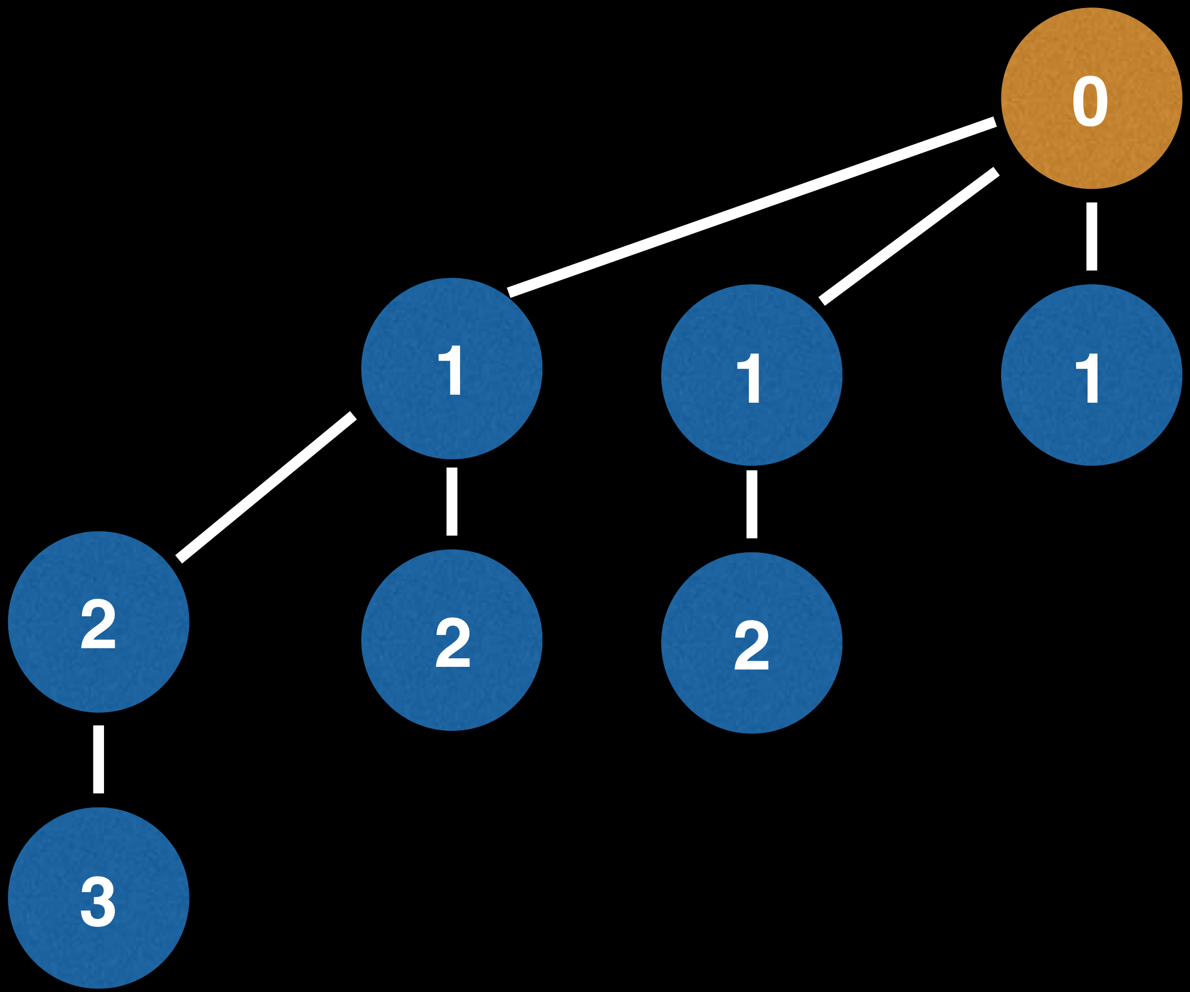


No, we have a violation of the heap invariant.

Is this a valid heap?

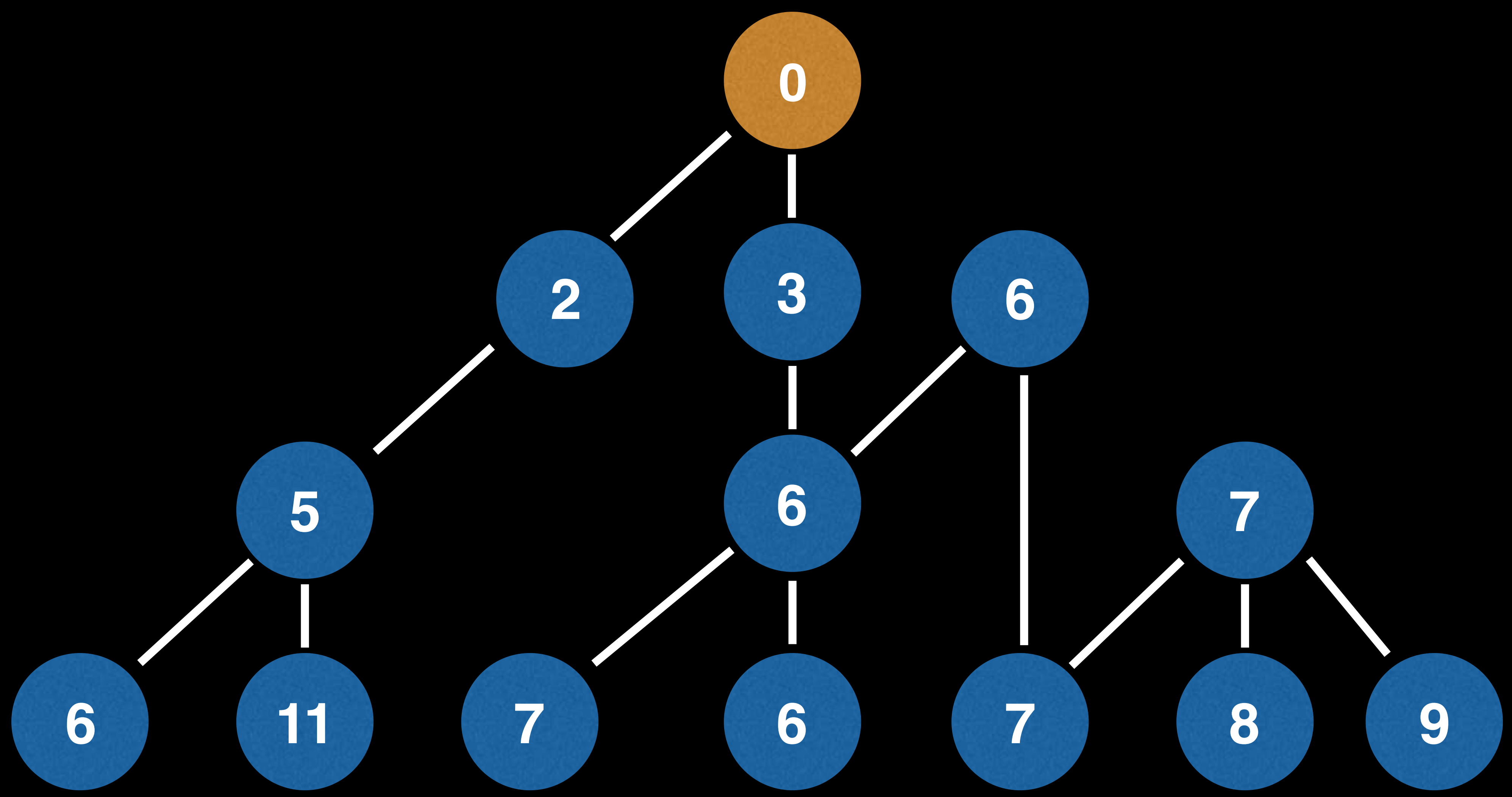


Is this a valid heap?

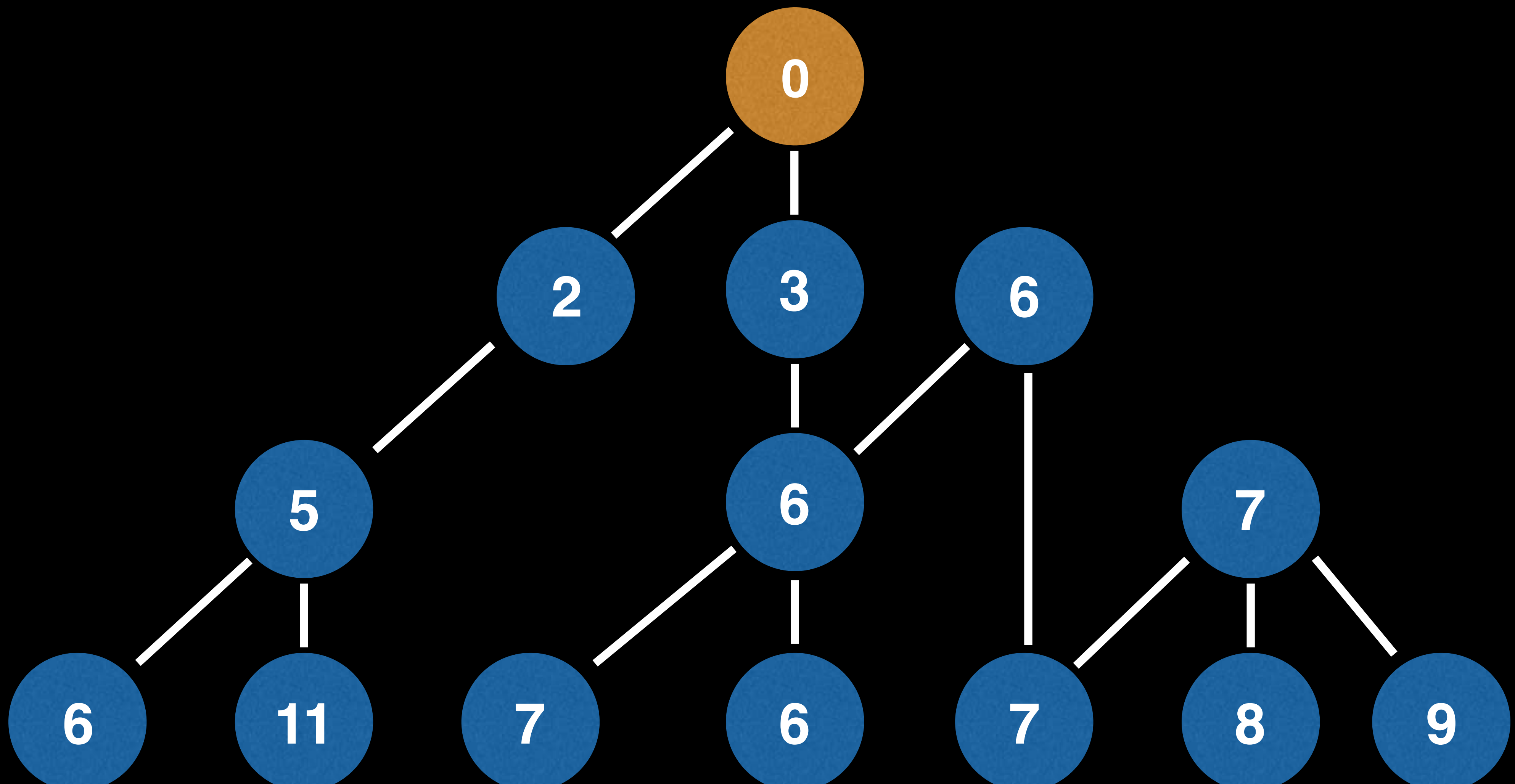


Yes! This is a tree and it satisfies the heap invariant. Heaps like these are often seen in binomial heaps.

Is this a valid heap?

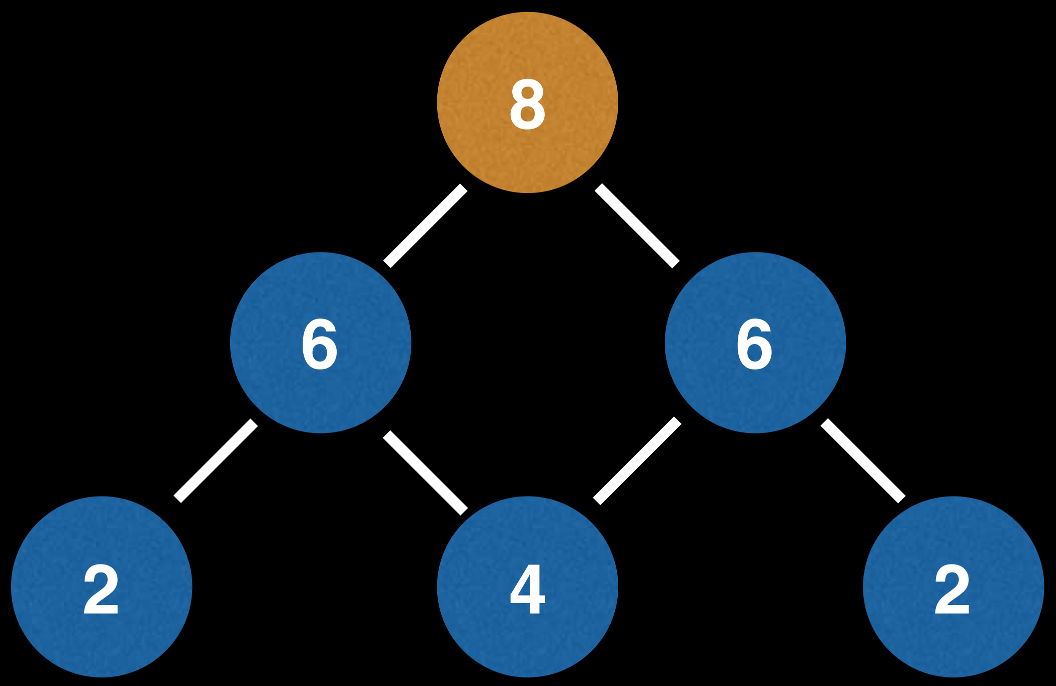


Is this a valid heap?

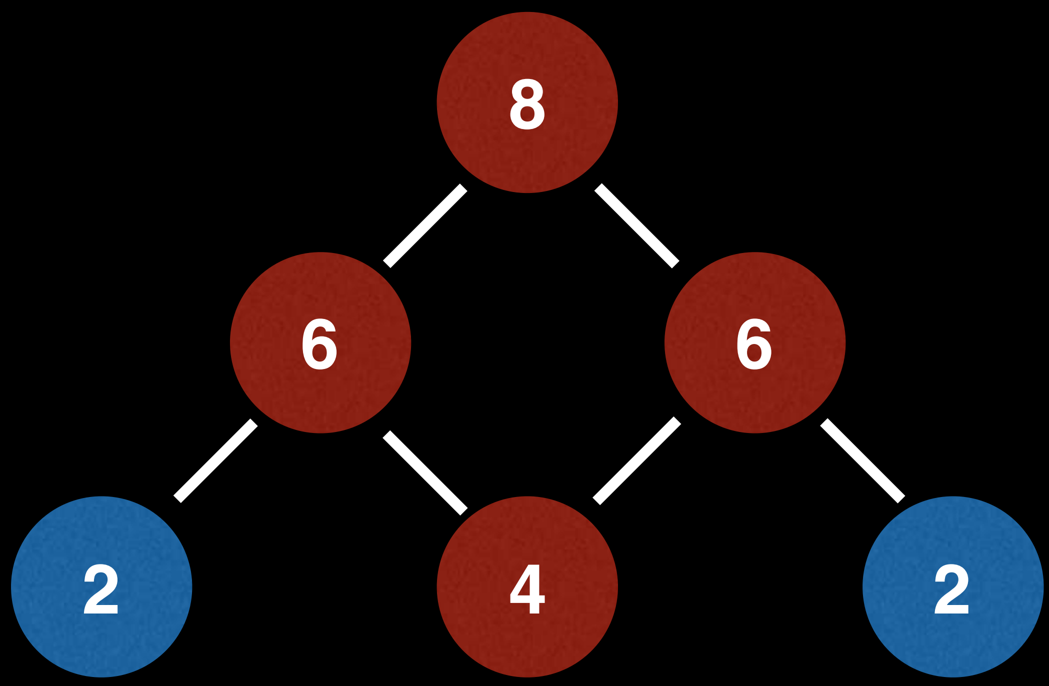


Yes!

Is this a valid heap?



Is this a valid heap?



No. This structure is not a tree because it contains a cycle. Heaps must be trees.

Is this a valid heap?

Is this a valid heap?

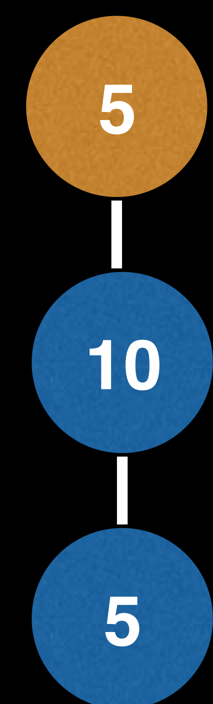
Yes!

Is this a valid heap?

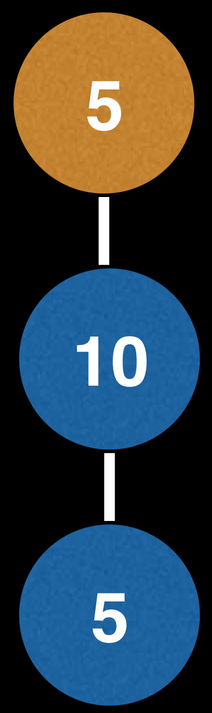
Is this a valid heap?

Yes!

Is this a valid heap?

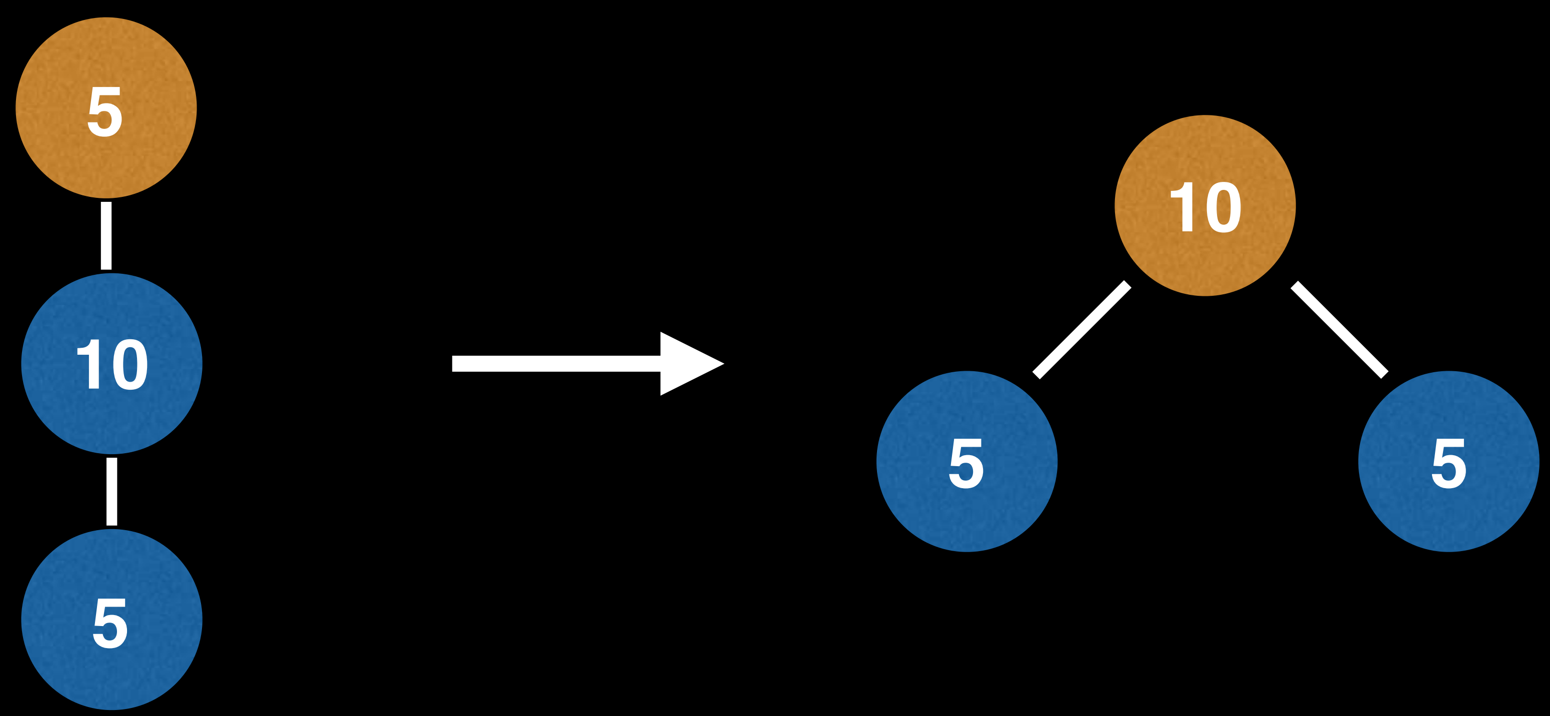


Is this a valid heap?



No.

Is this a valid heap?



However, if we change the root to be 10 then we can satisfy the heap property.

When and where is a PQ used?

- Used in certain implementations of Dijkstra's Shortest Path algorithm.
- Anytime you need the dynamically fetch the ‘next best’ or ‘next worst’ element.
- Used in Huffman coding (which is often used for lossless data compression).
- Best First Search (BFS) algorithms such as A* use PQs to continuously grab the next most promising node.
- Used by Minimum Spanning Tree (MST) algorithms.

Complexity PQ with binary heap

Binary Heap construction	O(n)
Polling	O(log(n))
Peeking	O(1)
Adding	O(log(n))

Complexity PQ with binary heap

Naive Removing	$O(n)$
Advanced removing with help from a hash table *	$O(\log(n))$
Naive contains	$O(n)$
Contains check with help of a hash table *	$O(1)$

* Using a hash table to help optimize these operations does take up linear space and also adds some overhead to the binary heap implementation.

Turning Min PQ into Max PQ

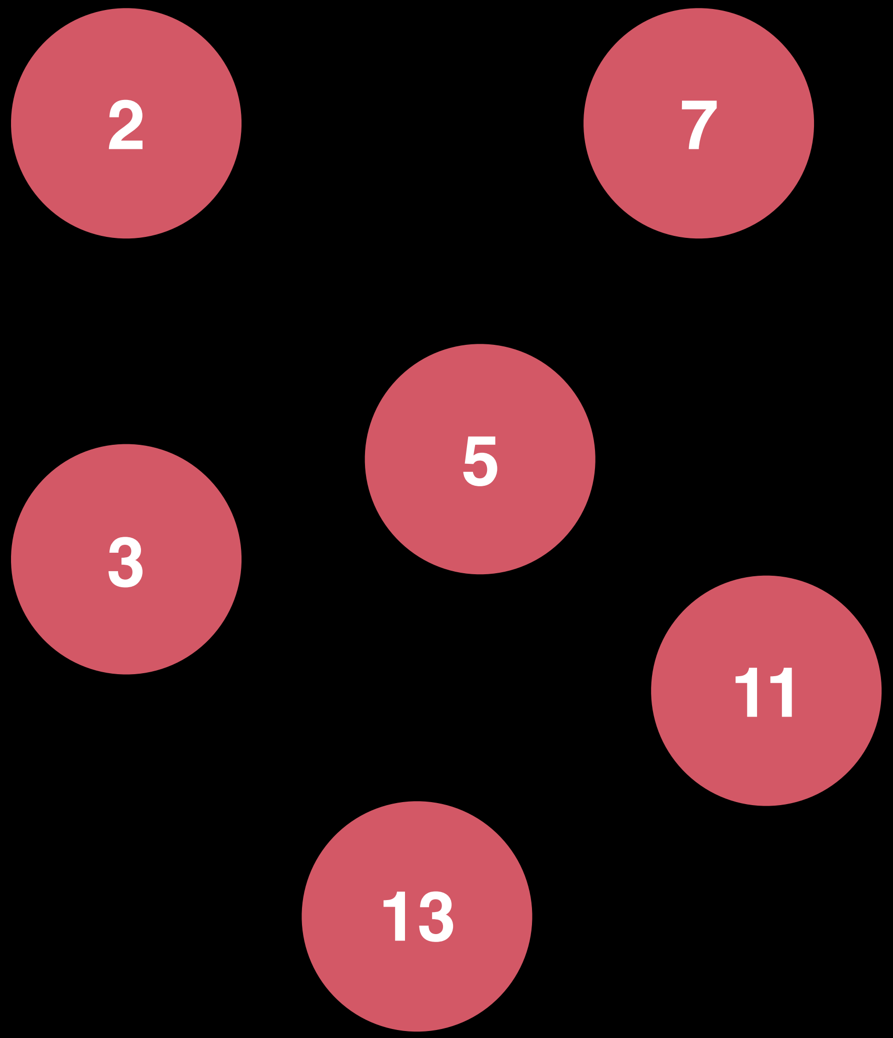
Turning Min PQ into Max PQ

Problem: Often the standard library of most programming languages only provide a min PQ which sorts by smallest elements first, but sometimes we need a Max PQ.

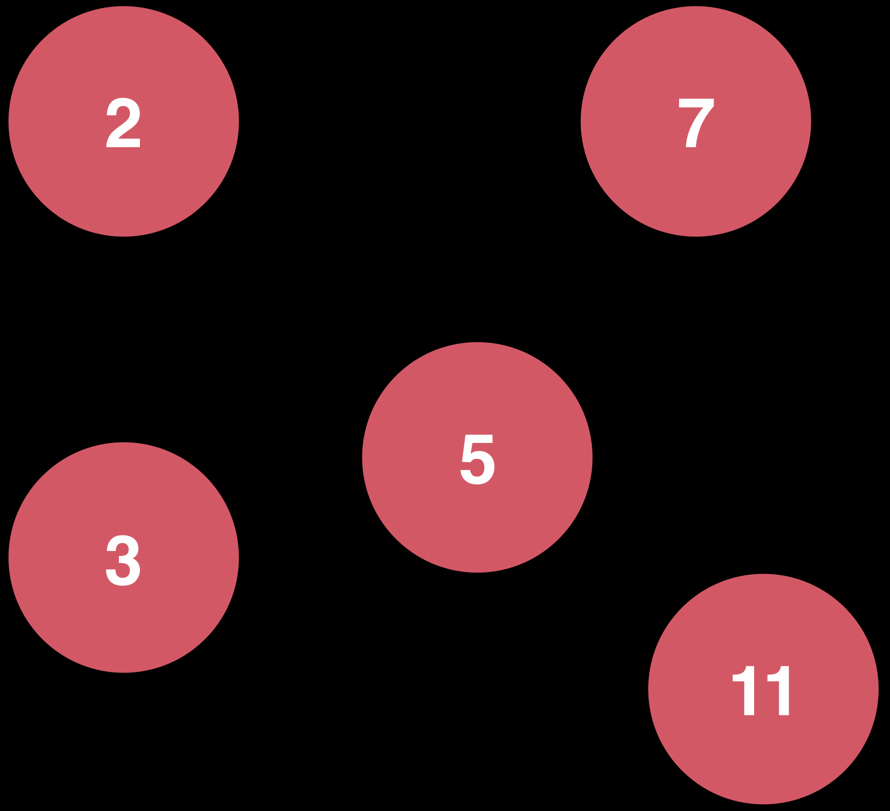
Since elements in a priority queue are comparable they implement some sort of **comparable interface** which we can simply **negate** to achieve a Max heap.

Turning Min PQ into Max PQ

Let x, y be numbers in the PQ. For a min PQ, if x <= y then x comes out of the PQ before y, so the negation of this is if x >= y then y comes out before x.



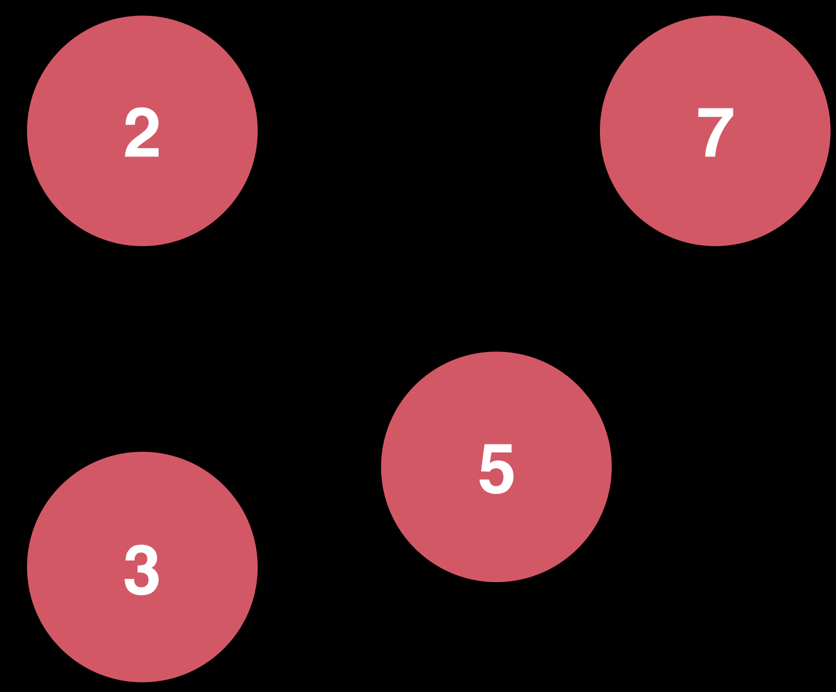
Turning Min PQ into Max PQ

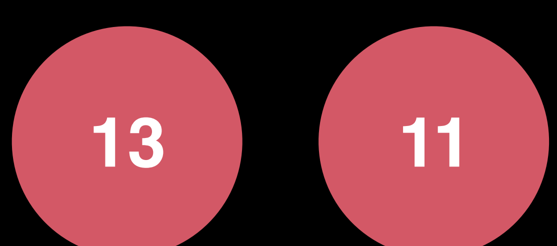


Let x, y be numbers in the PQ. For a min PQ, if x <= y then x comes out of the PQ before y, so the negation of this is if x >= y then y comes out before x.

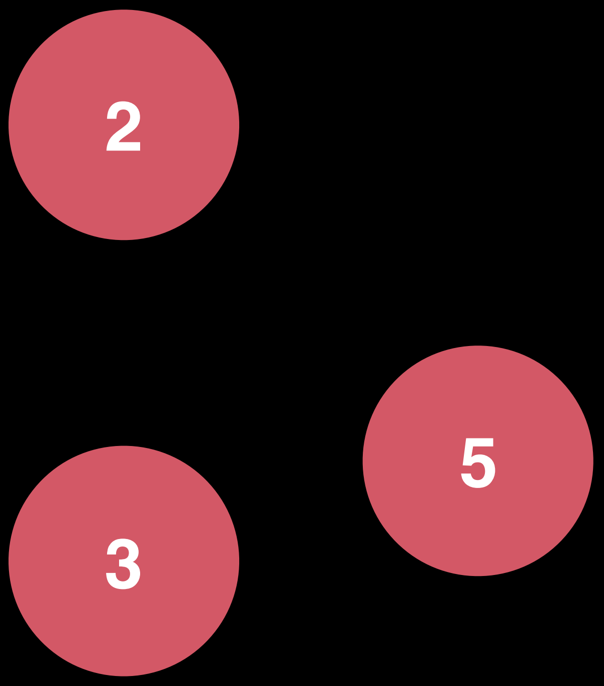
Turning Min PQ into Max PQ

Let x, y be numbers in the PQ. For a min PQ, if x <= y then x comes out of the PQ before y, so the negation of this is if x >= y then y comes out before x.



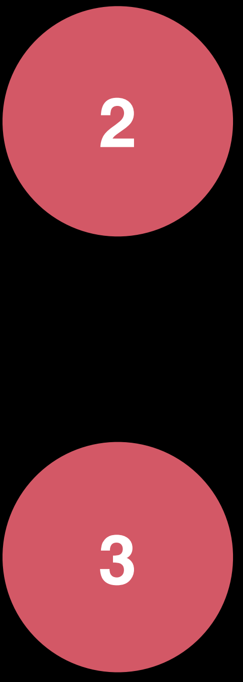


Turning Min PQ into Max PQ

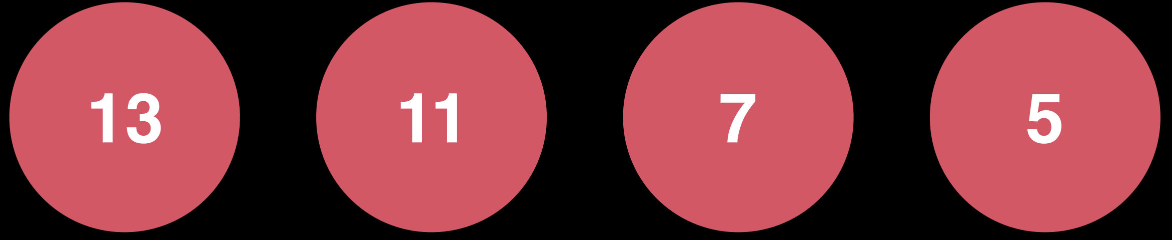


Let x, y be numbers in the PQ. For a min PQ, if x <= y then x comes out of the PQ before y, so the negation of this is if x >= y then y comes out before x.

Turning Min PQ into Max PQ



Let x, y be numbers in the PQ. For a min PQ, if x <= y then x comes out of the PQ before y, so the negation of this is if x >= y then y comes out before x.

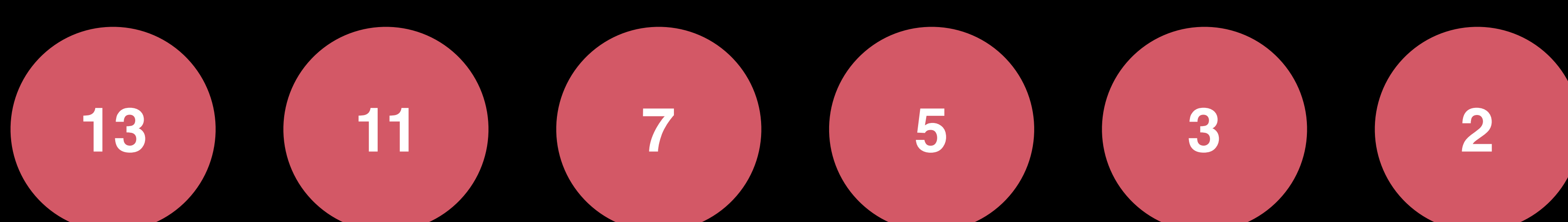


Turning Min PQ into Max PQ

Let x, y be numbers in the PQ. For a min PQ, if x <= y then x comes out of the PQ before y, so the negation of this is if x >= y then y comes out before x.

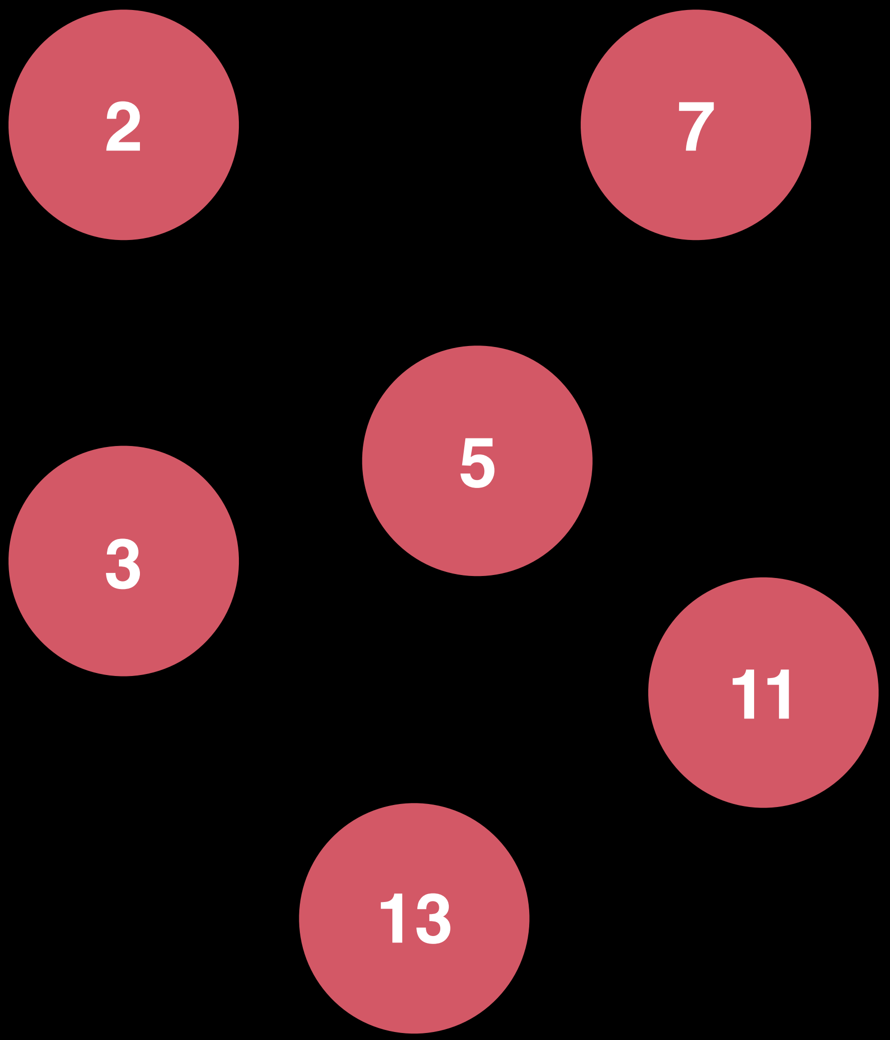
Turning Min PQ into Max PQ

Let x, y be numbers in the PQ. For a min PQ, if x <= y then x comes out of the PQ before y, so the negation of this is if x >= y then y comes out before x.



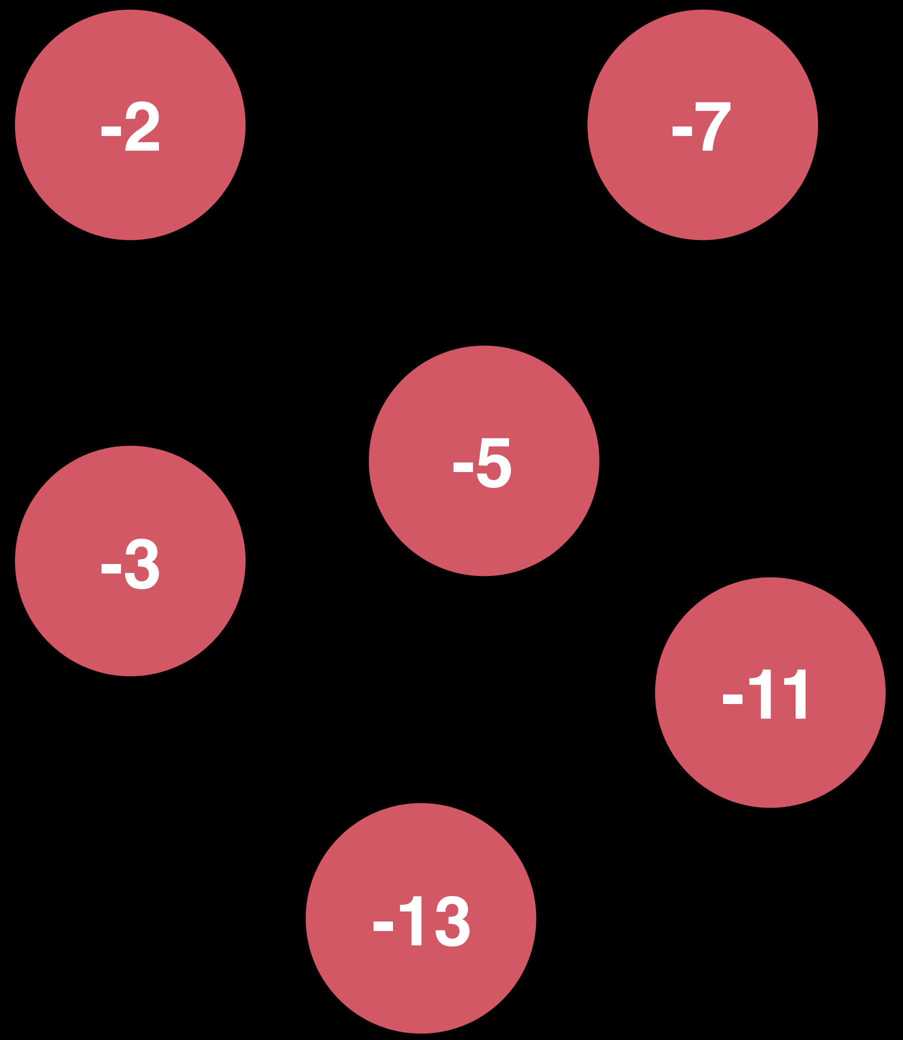
Turning Min PQ into Max PQ

An alternative method for numbers is to negate the numbers as you insert them into the PQ and negate them again when they are taken out. This has the same effect as negating the comparator.



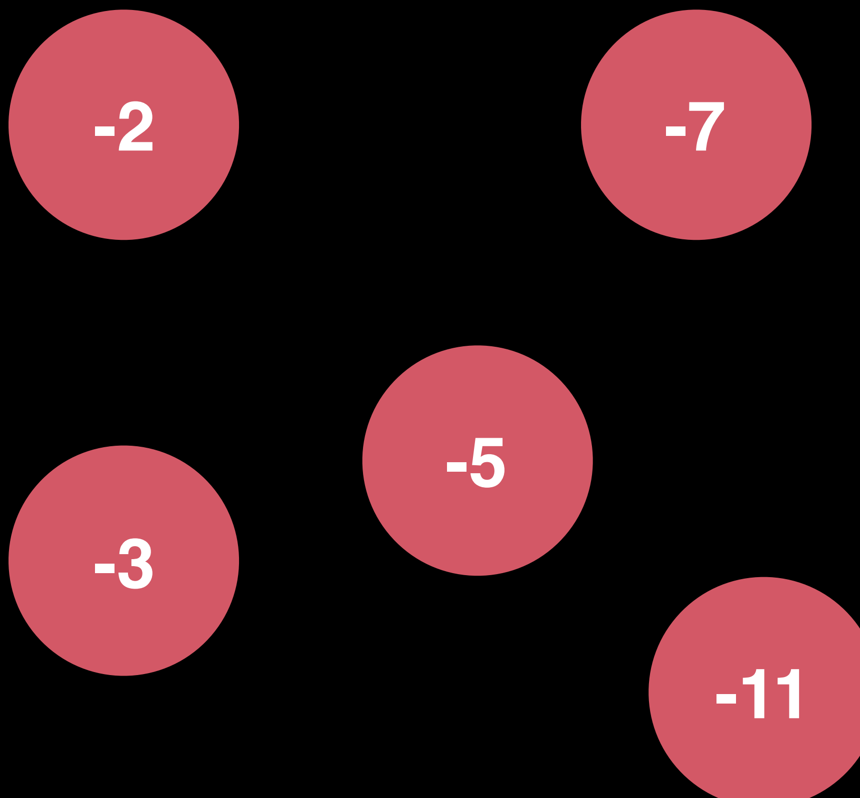
Turning Min PQ into Max PQ

An alternative method for numbers is to negate the numbers as you insert them into the PQ and negate them again when they are taken out. This has the same effect as negating the comparator.



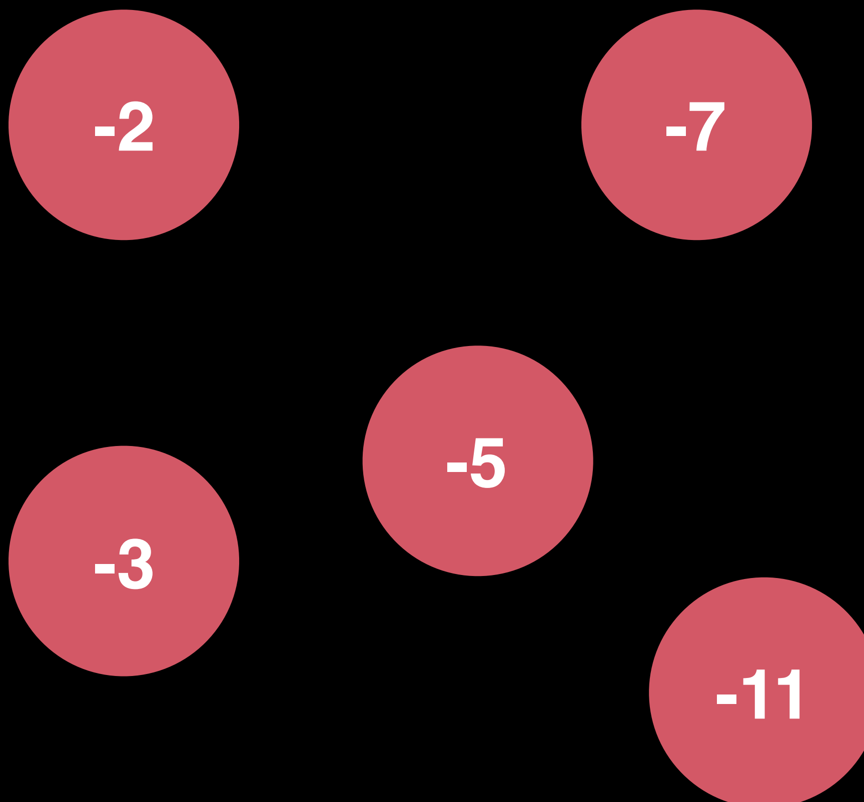
Turning Min PQ into Max PQ

An alternative method for numbers is to negate the numbers as you insert them into the PQ and negate them again when they are taken out. This has the same effect as negating the comparator.



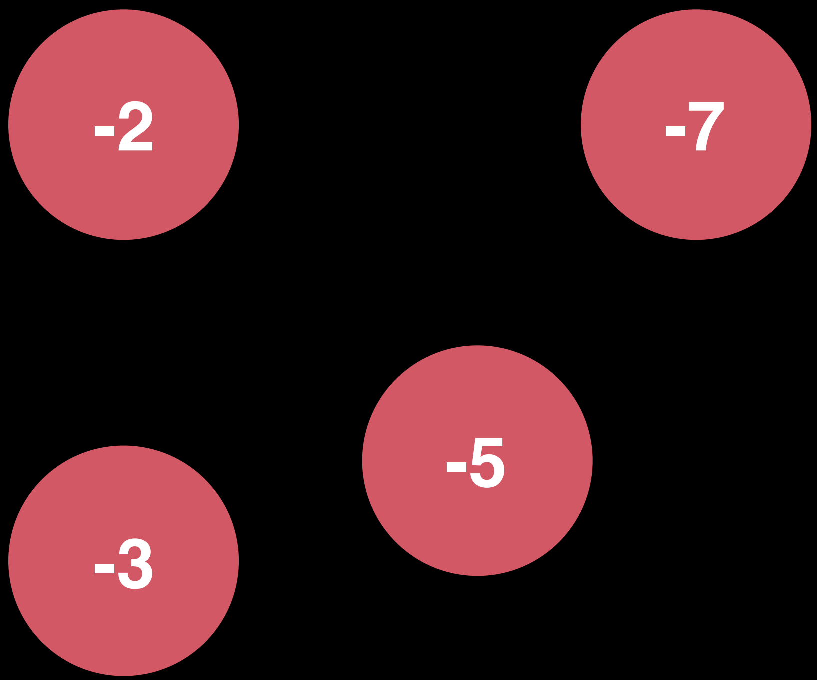
Turning Min PQ into Max PQ

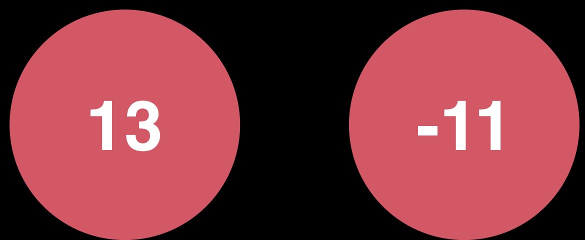
An alternative method for numbers is to negate the numbers as you insert them into the PQ and negate them again when they are taken out. This has the same effect as negating the comparator.



Turning Min PQ into Max PQ

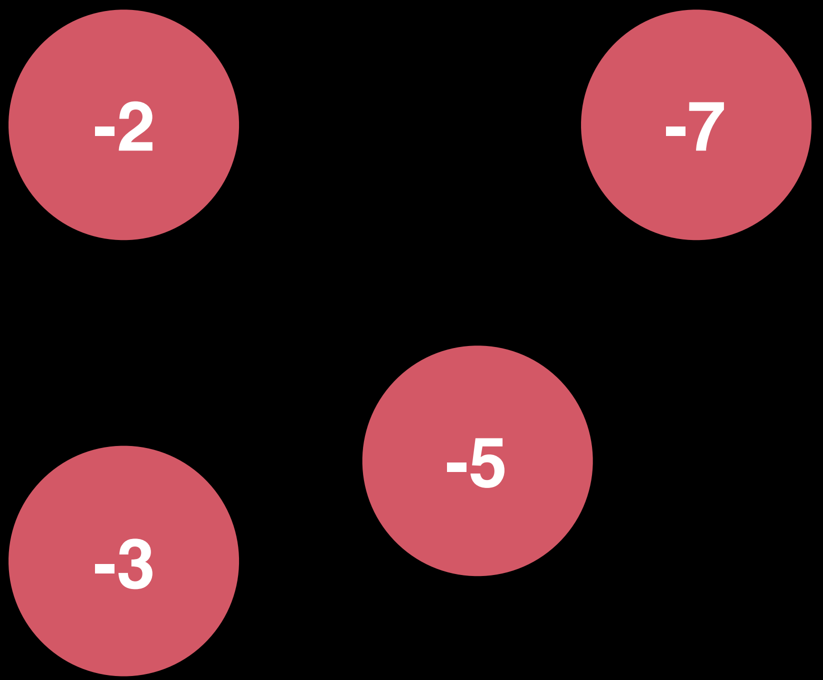
An alternative method for numbers is to negate the numbers as you insert them into the PQ and negate them again when they are taken out. This has the same effect as negating the comparator.

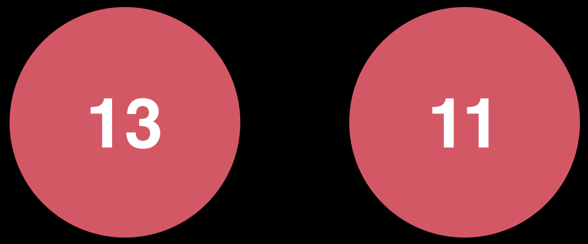




Turning Min PQ into Max PQ

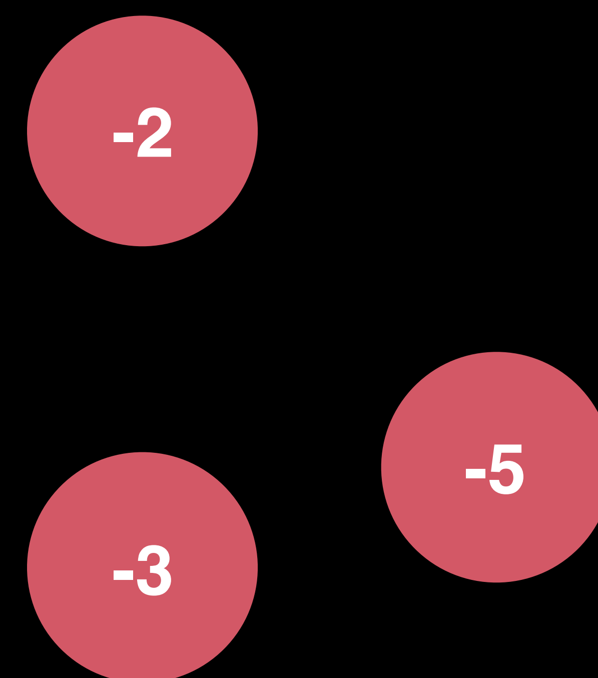
An alternative method for numbers is to negate the numbers as you insert them into the PQ and negate them again when they are taken out. This has the same effect as negating the comparator.





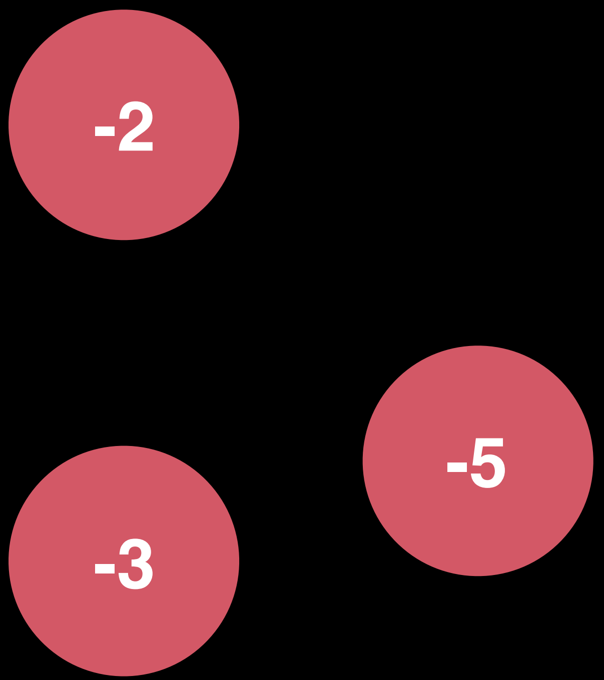
Turning Min PQ into Max PQ

An alternative method for numbers is to negate the numbers as you insert them into the PQ and negate them again when they are taken out. This has the same effect as negating the comparator.



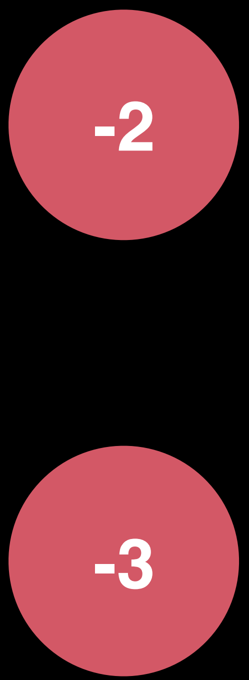
Turning Min PQ into Max PQ

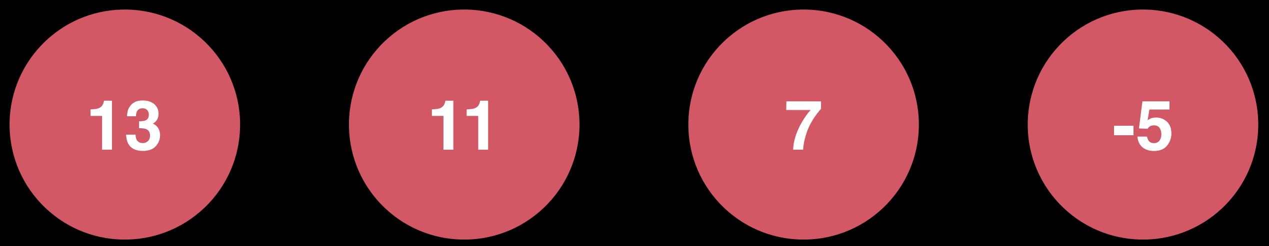
An alternative method for numbers is to negate the numbers as you insert them into the PQ and negate them again when they are taken out. This has the same effect as negating the comparator.



Turning Min PQ into Max PQ

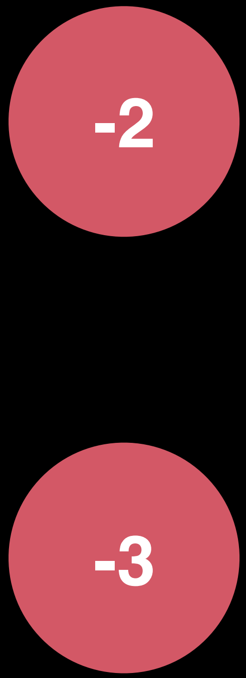
An alternative method for numbers is to negate the numbers as you insert them into the PQ and negate them again when they are taken out. This has the same effect as negating the comparator.

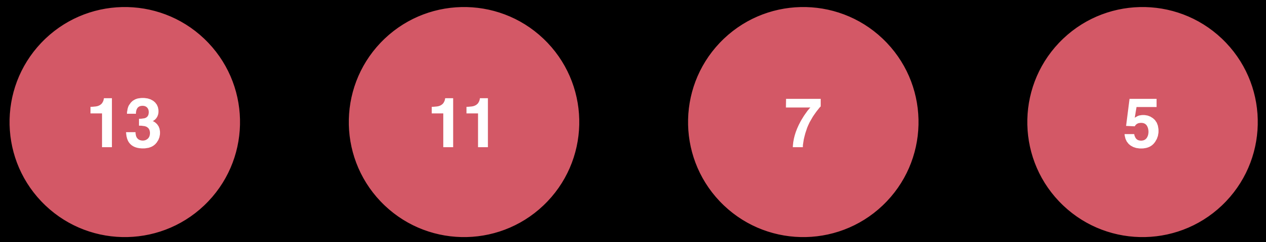




Turning Min PQ into Max PQ

An alternative method for numbers is to negate the numbers as you insert them into the PQ and negate them again when they are taken out. This has the same effect as negating the comparator.





Turning Min PQ into Max PQ

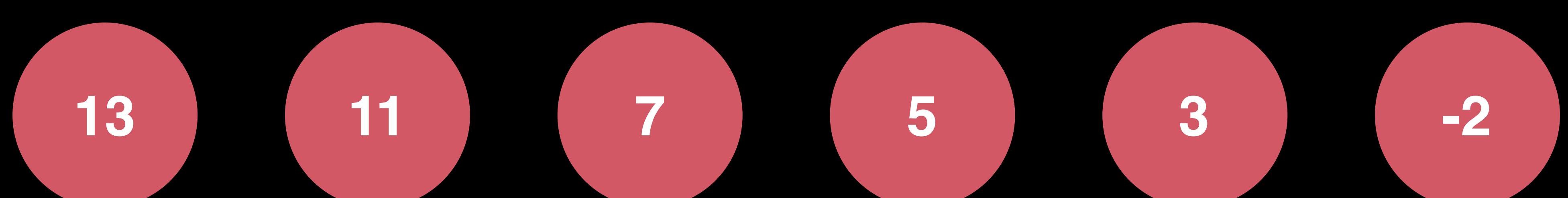
An alternative method for numbers is to negate the numbers as you insert them into the PQ and negate them again when they are taken out. This has the same effect as negating the comparator.

Turning Min PQ into Max PQ

An alternative method for numbers is to negate the numbers as you insert them into the PQ and negate them again when they are taken out. This has the same effect as negating the comparator.

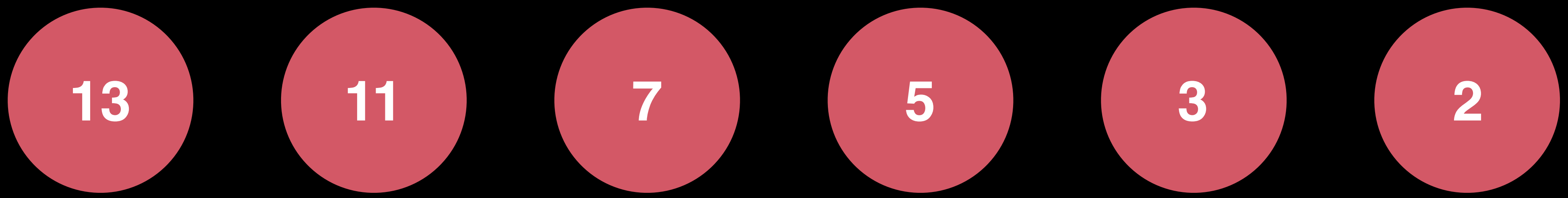
Turning Min PQ into Max PQ

An alternative method for numbers is to negate the numbers as you insert them into the PQ and negate them again when they are taken out. This has the same effect as negating the comparator.



Turning Min PQ into Max PQ

An alternative method for numbers is to negate the numbers as you insert them into the PQ and negate them again when they are taken out. This has the same effect as negating the comparator.



Turning Min PQ into Max PQ

Suppose ***lex*** is a comparator for strings which sorts strings in lexicographic order (the default in most programming languages). Then let ***nlex*** be the negation of ***lex***, and also let s_1, s_2 be strings

lex$(s_1, s_2) = -1$ if $s_1 < s_2$ lexicographically

lex$(s_1, s_2) = 0$ if $s_1 = s_2$ lexicographically

lex$(s_1, s_2) = +1$ if $s_1 > s_2$ lexicographically

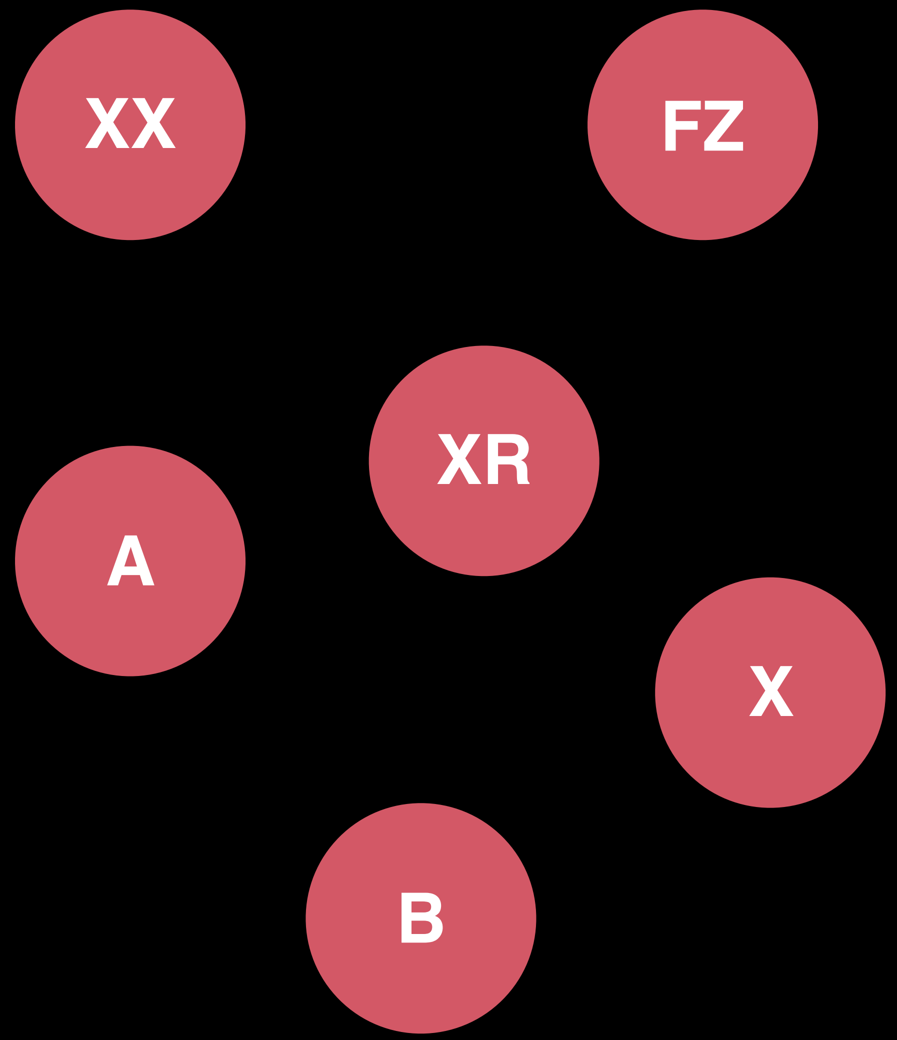
nlex$(s_1, s_2) = -(-1) = +1$ $s_1 < s_2$ lexicographically

nlex$(s_1, s_2) = -(0) = 0$ $s_1 = s_2$ lexicographically

nlex$(s_1, s_2) = -(+1) = -1$ $s_1 > s_2$ lexicographically

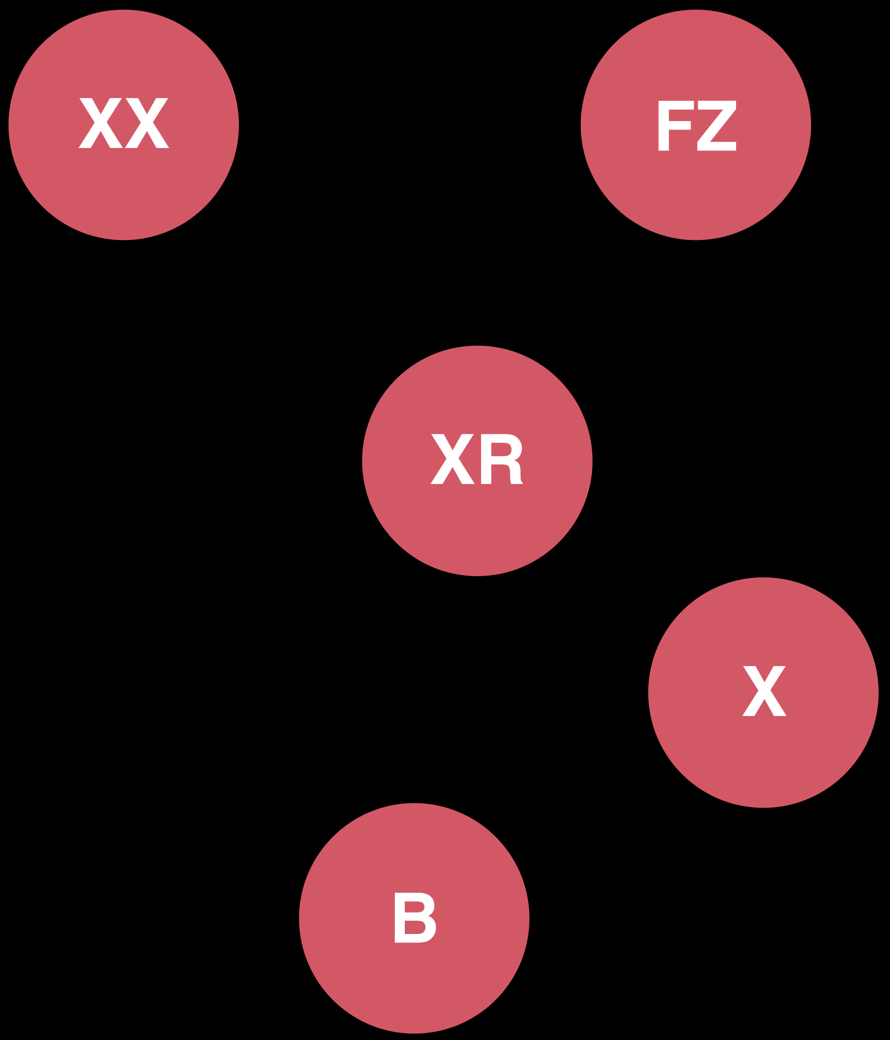
Turning Min PQ into Max PQ

By adding all these strings on the right to the PQ with the ***lex*** comparator, we obtain the following:

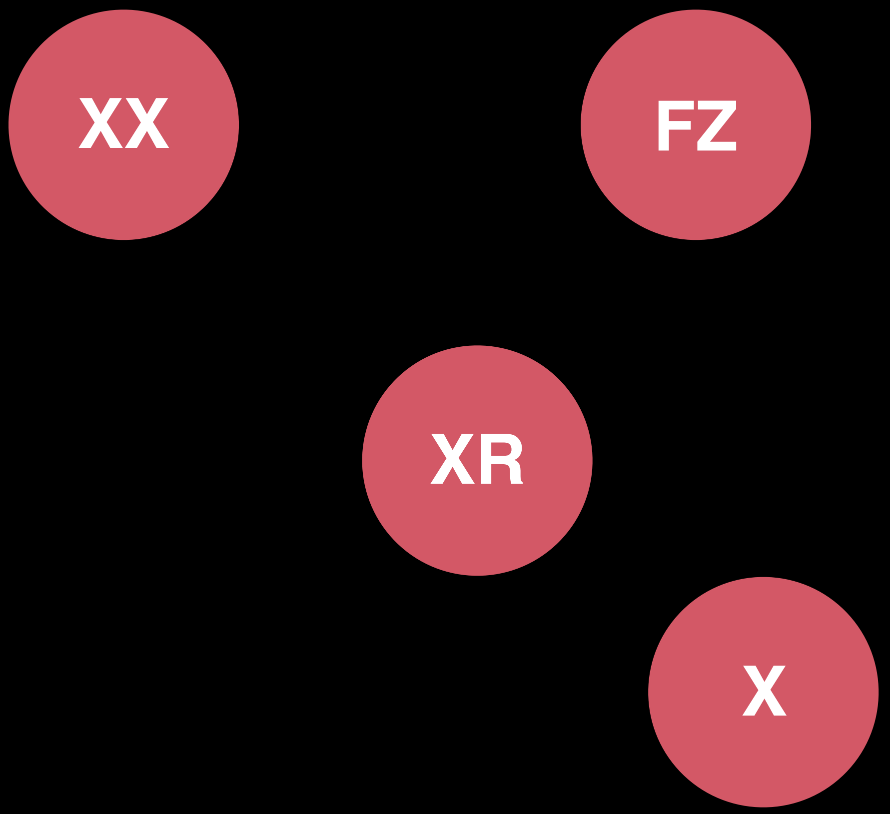


Turning Min PQ into Max PQ

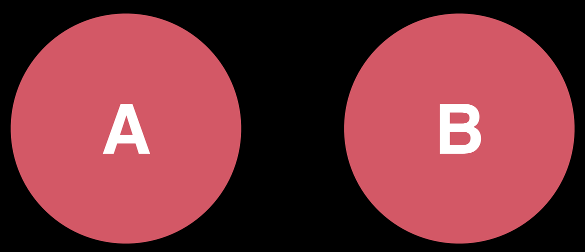
By adding all these strings on the right to the PQ with the ***lex*** comparator, we obtain the following:



Turning Min PQ into Max PQ



By adding all these strings on the right to the PQ with the ***lex*** comparator, we obtain the following:



Turning Min PQ into Max PQ

By adding all these strings on the right to the PQ with the ***lex*** comparator, we obtain the following:

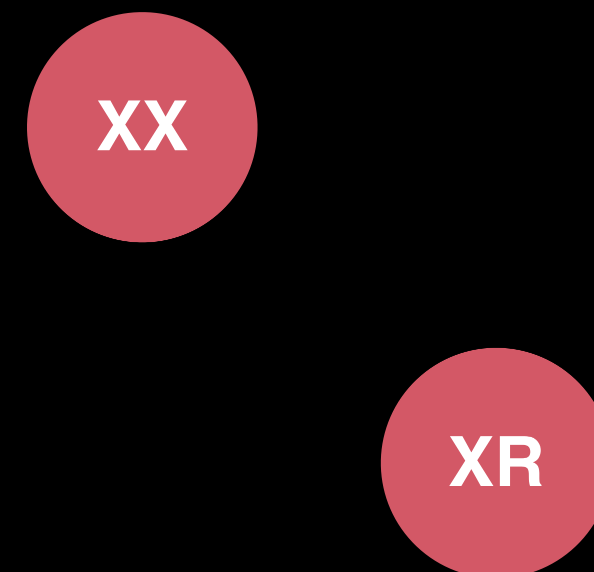
XX

XR

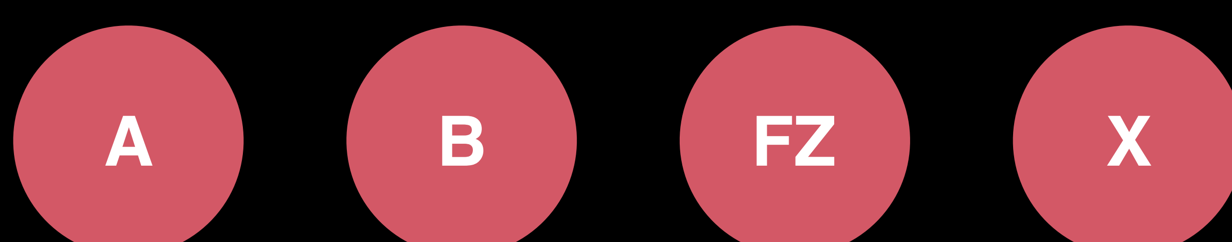
X

A B FZ

Turning Min PQ into Max PQ



By adding all these strings on the right to the PQ with the ***lex*** comparator, we obtain the following:



Turning Min PQ into Max PQ

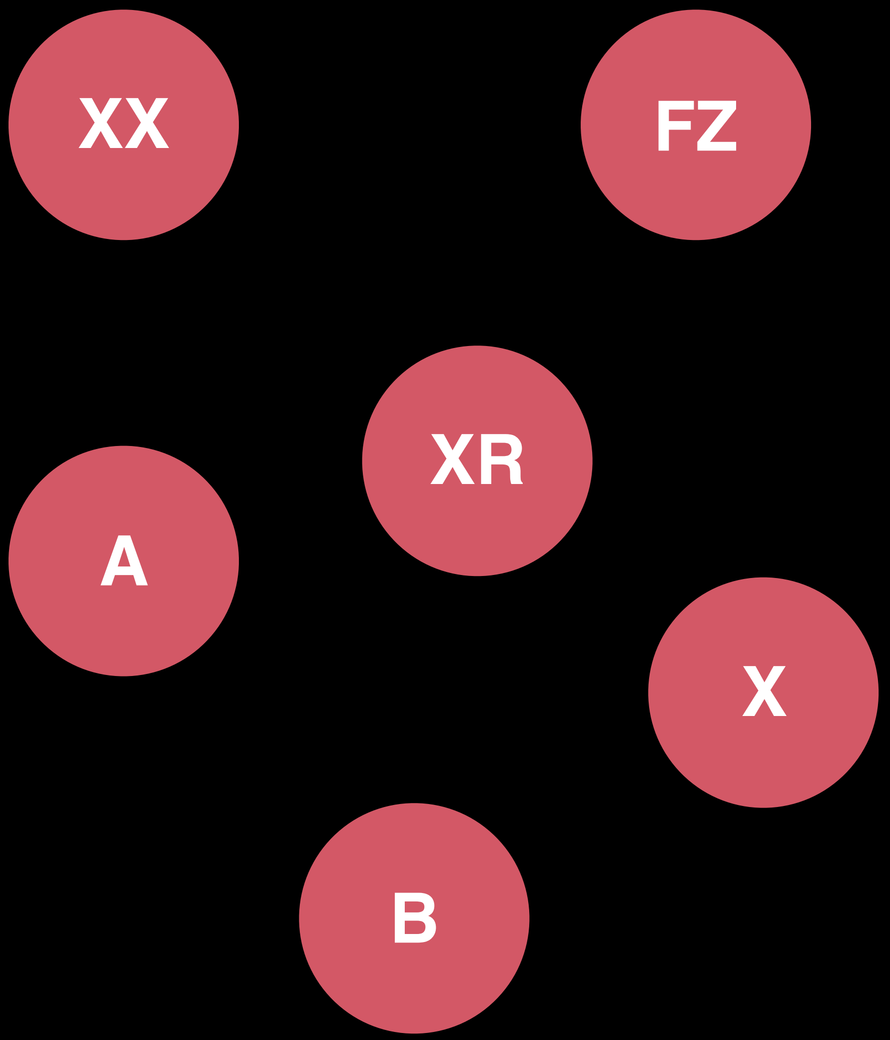
By adding all these strings on the right to the PQ with the ***lex*** comparator, we obtain the following:

Turning Min PQ into Max PQ

By adding all these strings on the right to the PQ with the ***lex*** comparator, we obtain the following:

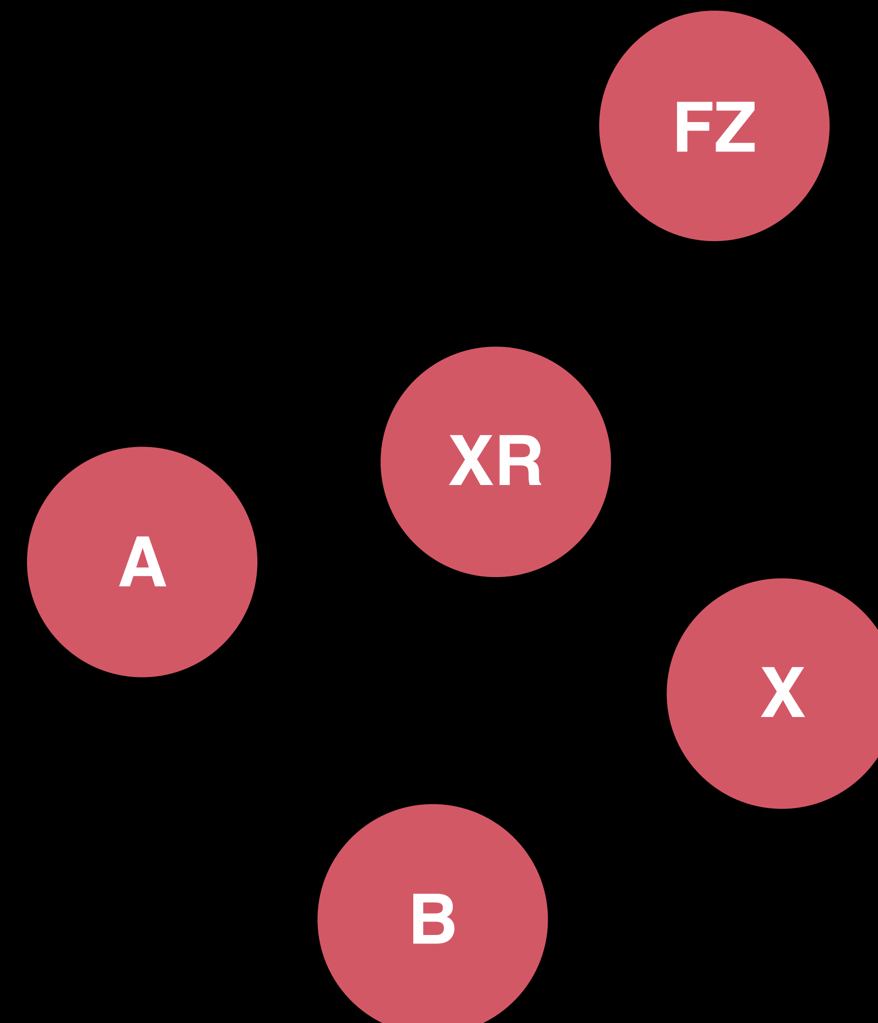
Turning Min PQ into Max PQ

By adding all these strings on the right to the PQ with the ***nlex*** comparator, we obtain the opposite:



Turning Min PQ into Max PQ

By adding all these strings on the right to the PQ with the ***nlex*** comparator, we obtain the opposite:



Turning Min PQ into Max PQ

By adding all these strings on the right to the PQ with the ***nlex*** comparator, we obtain the opposite:

FZ

A

X

B

XX

XR

Turning Min PQ into Max PQ

By adding all these strings on the right to the PQ with the ***nlex*** comparator, we obtain the opposite:

FZ

A

B

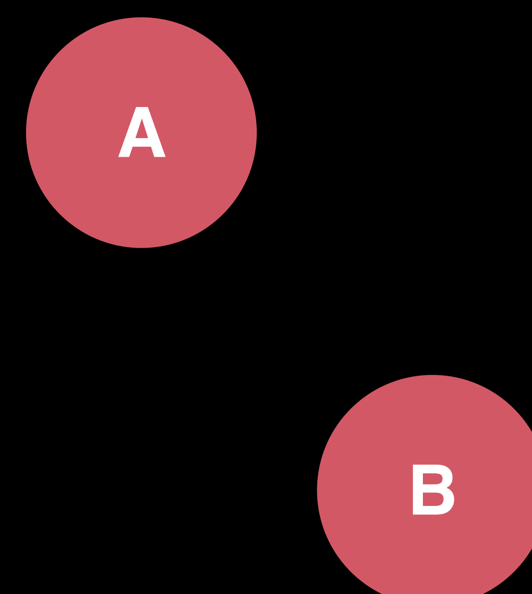
XX

XR

X

Turning Min PQ into Max PQ

By adding all these strings on the right to the PQ with the ***nlex*** comparator, we obtain the opposite:

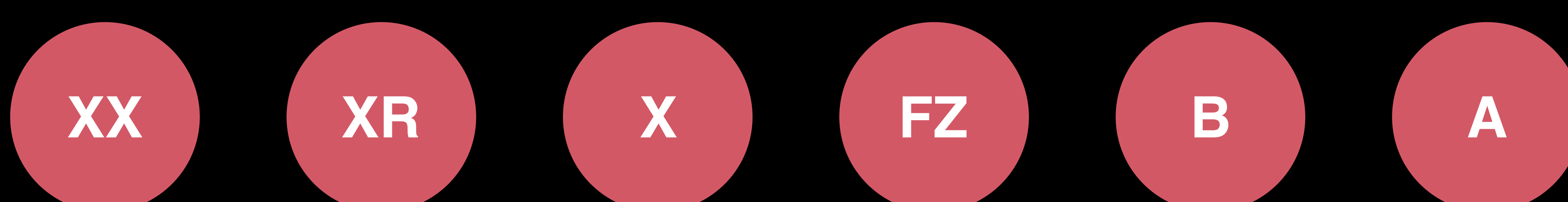


Turning Min PQ into Max PQ

By adding all these strings on the right to the PQ with the ***nlex*** comparator, we obtain the opposite:

Turning Min PQ into Max PQ

By adding all these strings on the right to the PQ with the ***nlex*** comparator, we obtain the opposite:



Adding Elements to Binary Heap

Ways of Implementing a Priority Queue

Priority queues are usually implemented with heaps since this gives them the best possible time complexity.

The Priority Queue (PQ) is an **Abstract Data Type (ADT)**, hence heaps are not the only way to implement PQs. As an example, we could use an unsorted list, but this would not give us the best possible time complexity.

Priority Queue With Binary Heap

There are many types of heaps we could use to implement a priority queue including:

Binary Heap
Fibonacci Heap
Binomial Heap
Pairing Heap
…

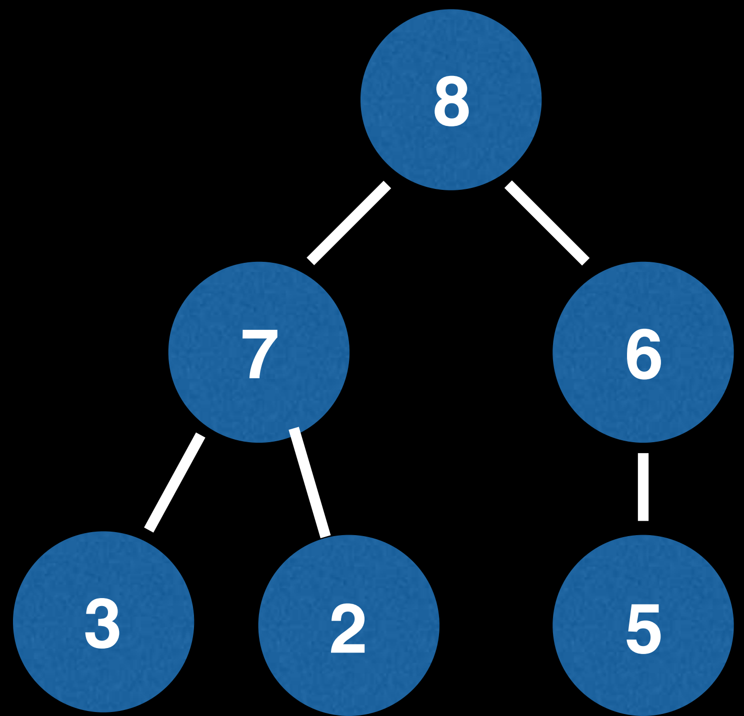
Priority Queue With Binary Heap

There are many types of heaps we could use to implement a priority queue including:

Binary Heap
Fibonacci Heap
Binomial Heap
Pairing Heap
…

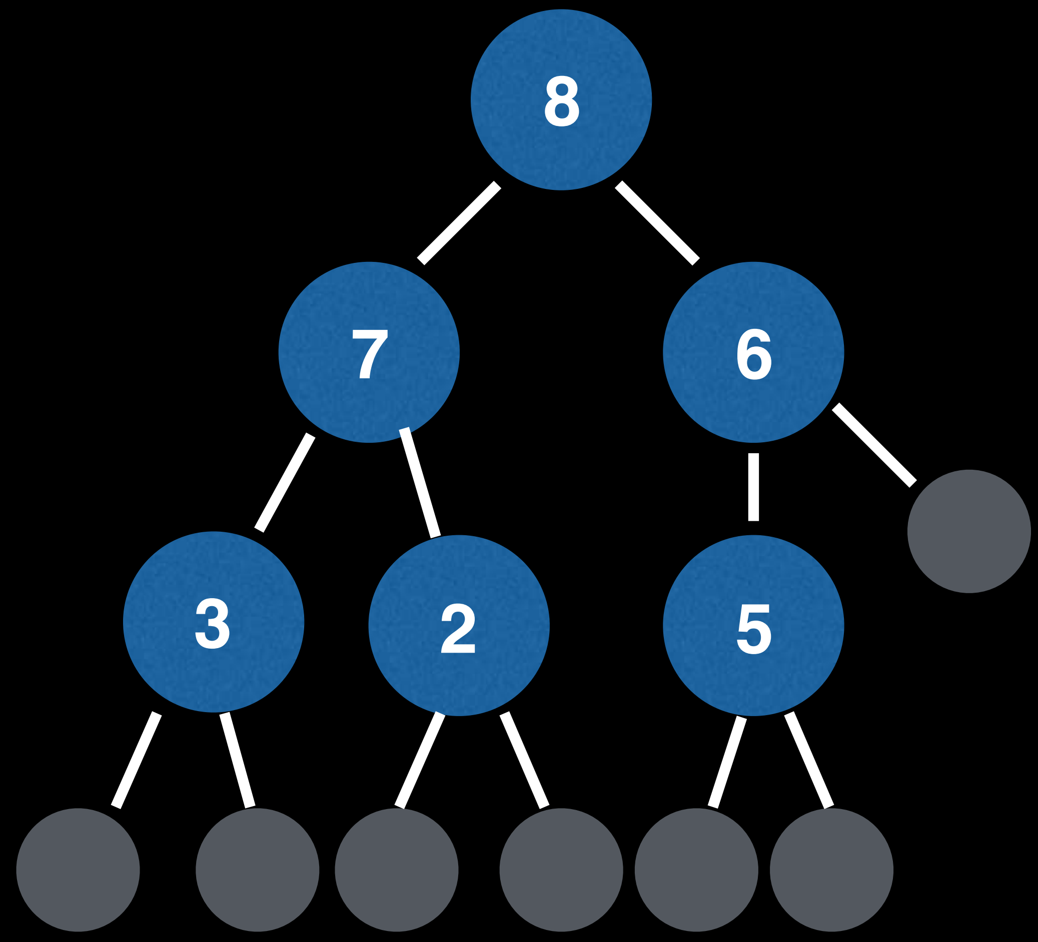
Priority Queue With Binary Heap

A **binary heap** is a **binary tree** that supports the **heap invariant**. In a binary tree every node has exactly two children.



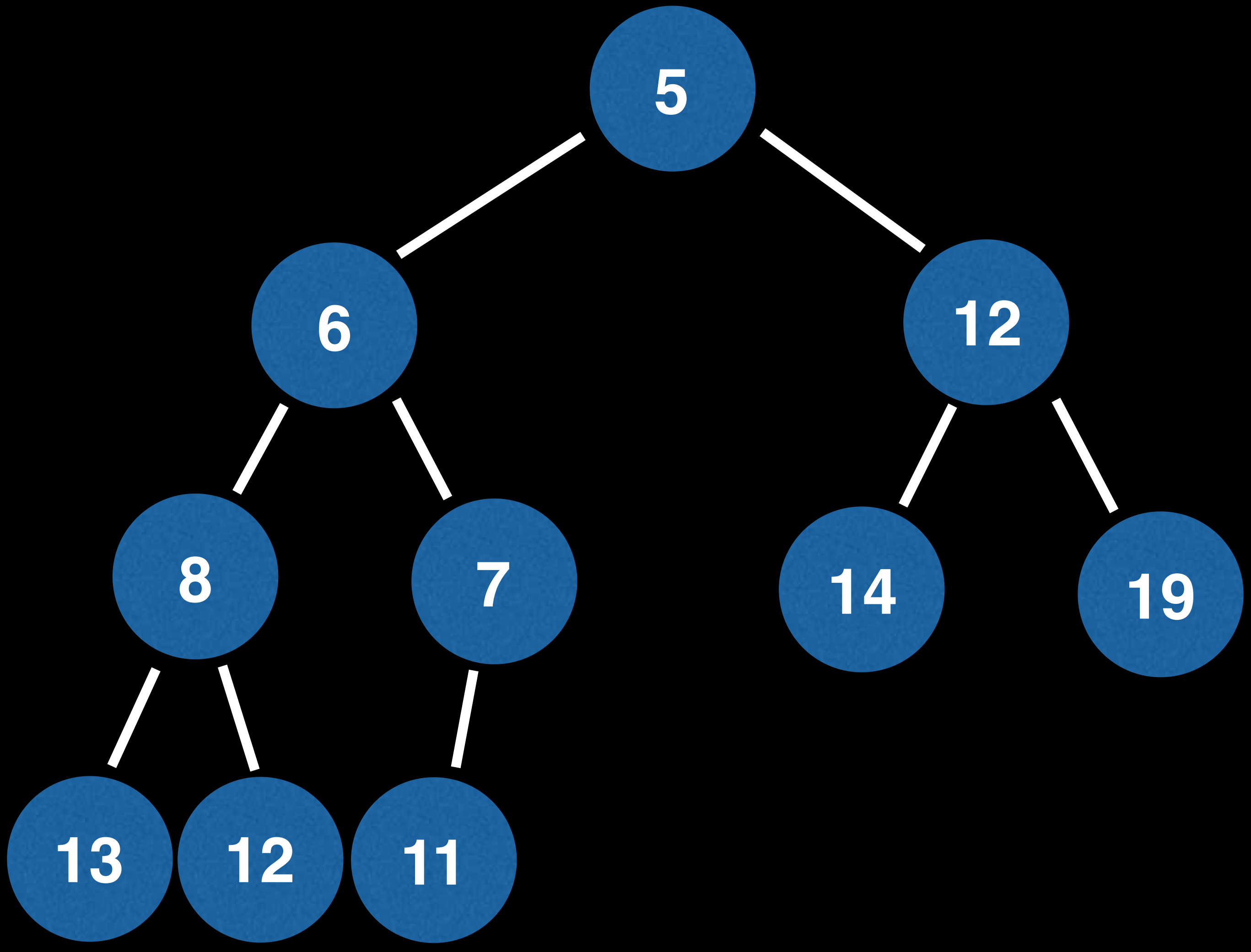
Priority Queue With Binary Heap

A **binary heap** is a heap where every node has exactly two children.



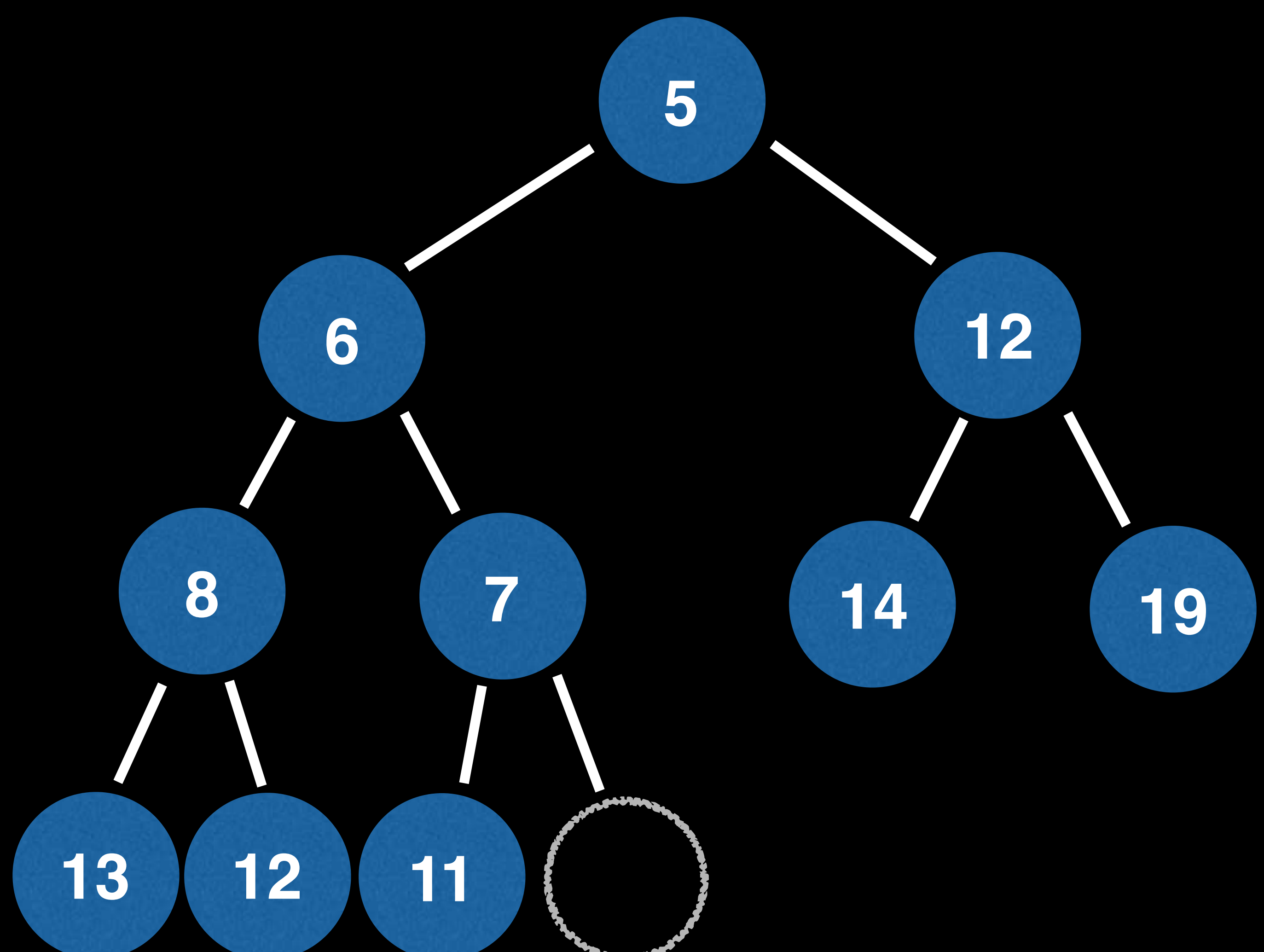
Priority Queue With Binary Heap

A **complete binary tree** is a tree in which at every level, except possibly the last is completely filled and and all the nodes are as far left as possible.



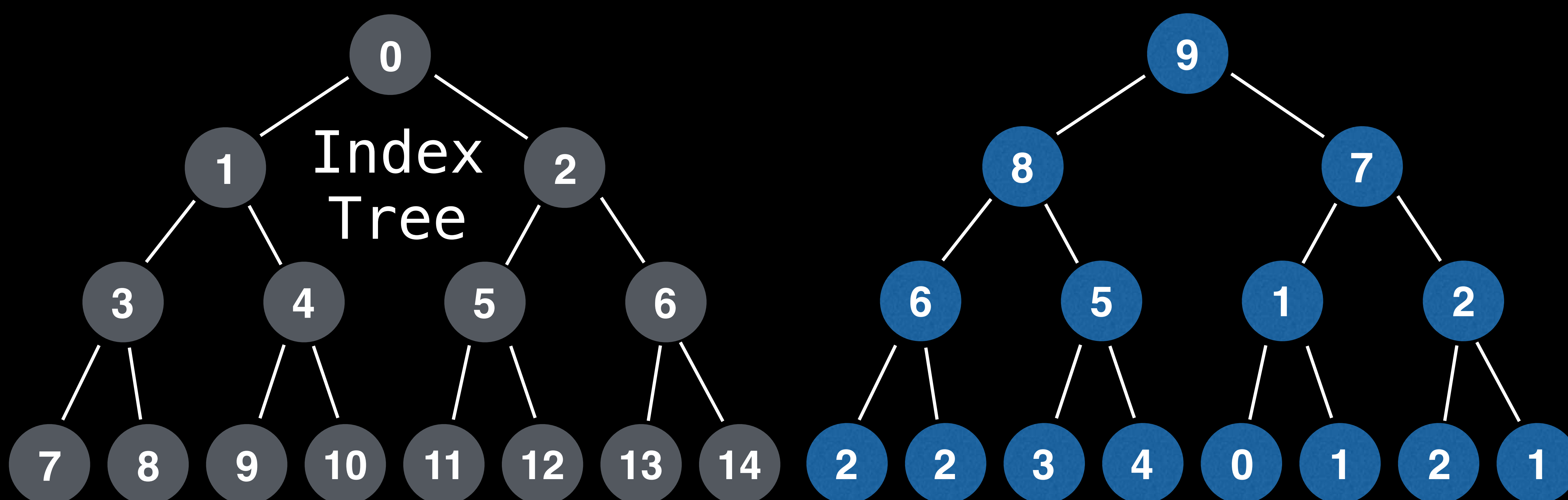
Priority Queue With Binary Heap

A **complete binary tree** is a tree in which at every level, except possibly the last is completely filled and and all the nodes are as far left as possible.



Binary Heap Representation

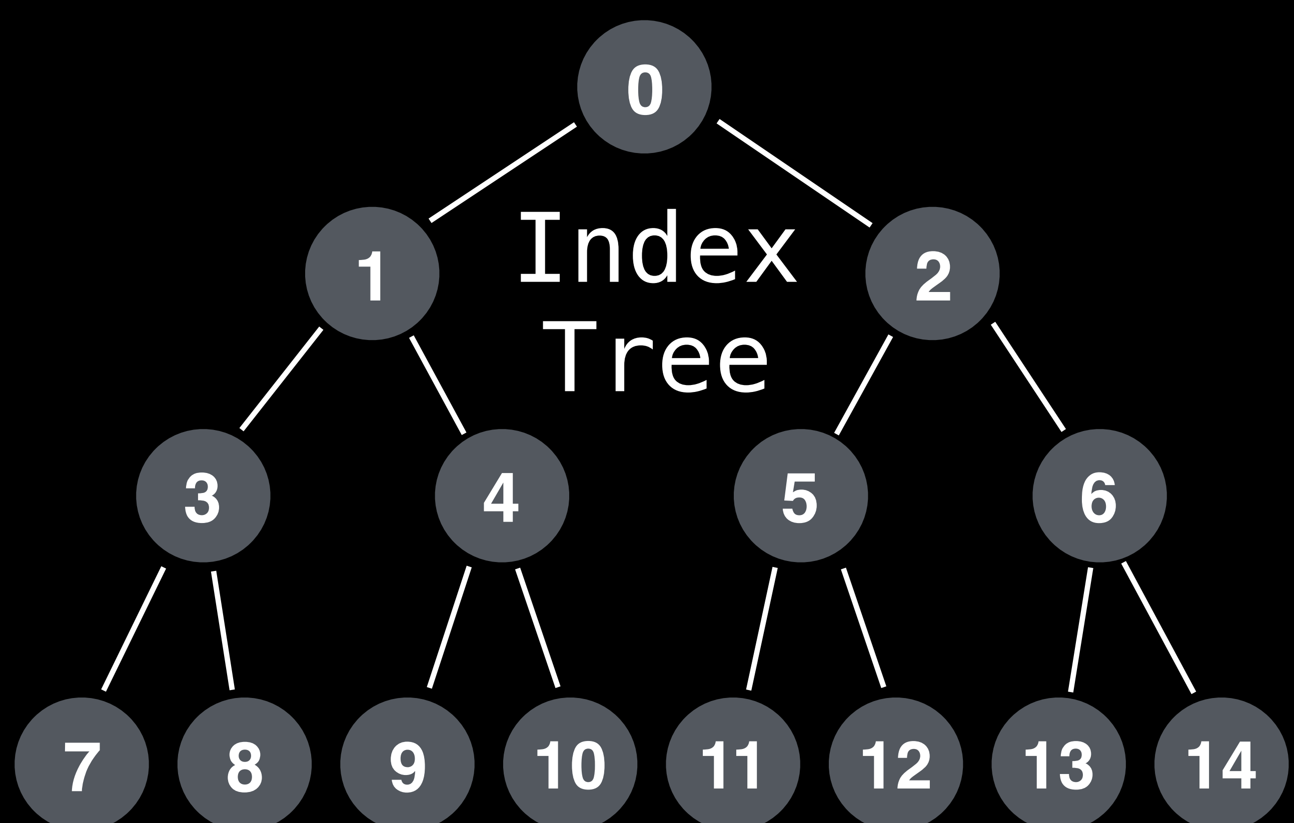
9	8	7	6	5	1	2	2	2	3	4	0	1	2	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

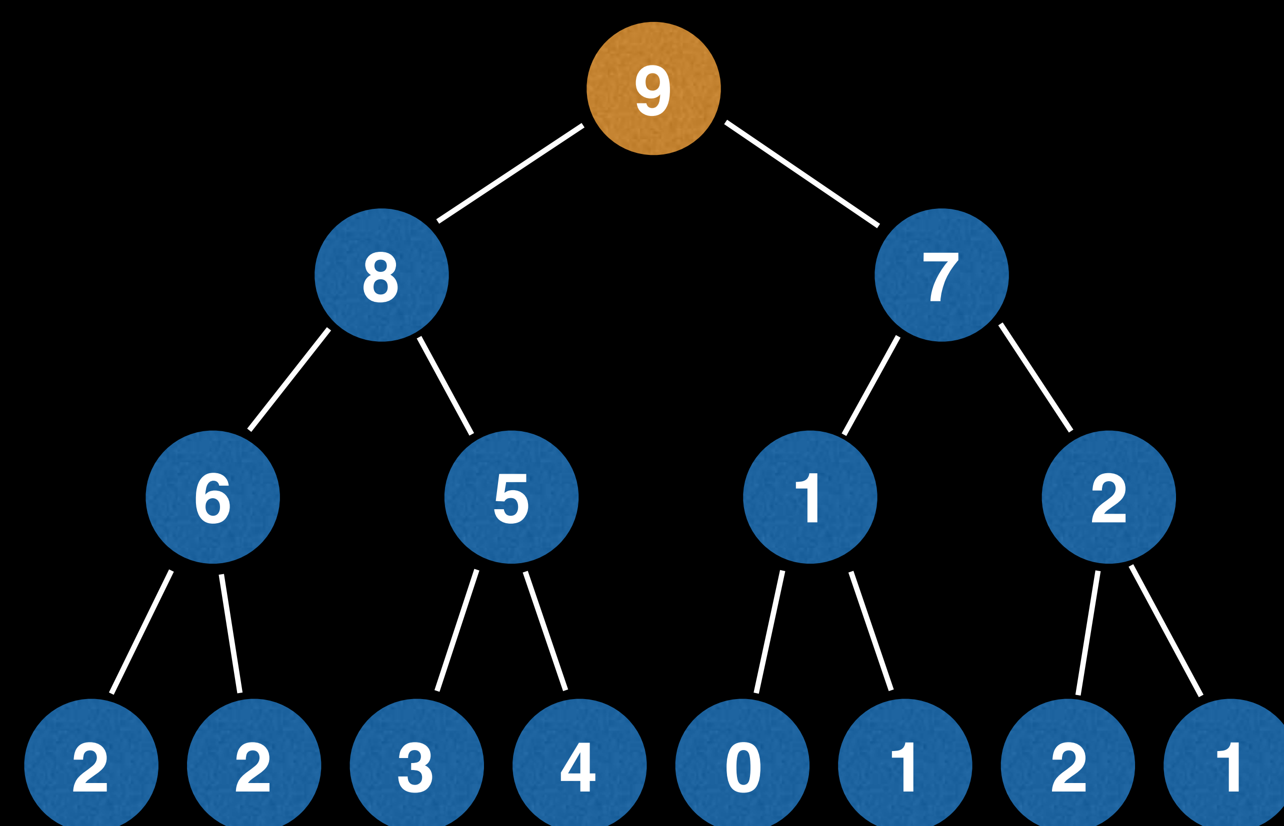


Binary Heap Representation

9	8	7	6	5	1	2	2	2	3	4	0	1	2	1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

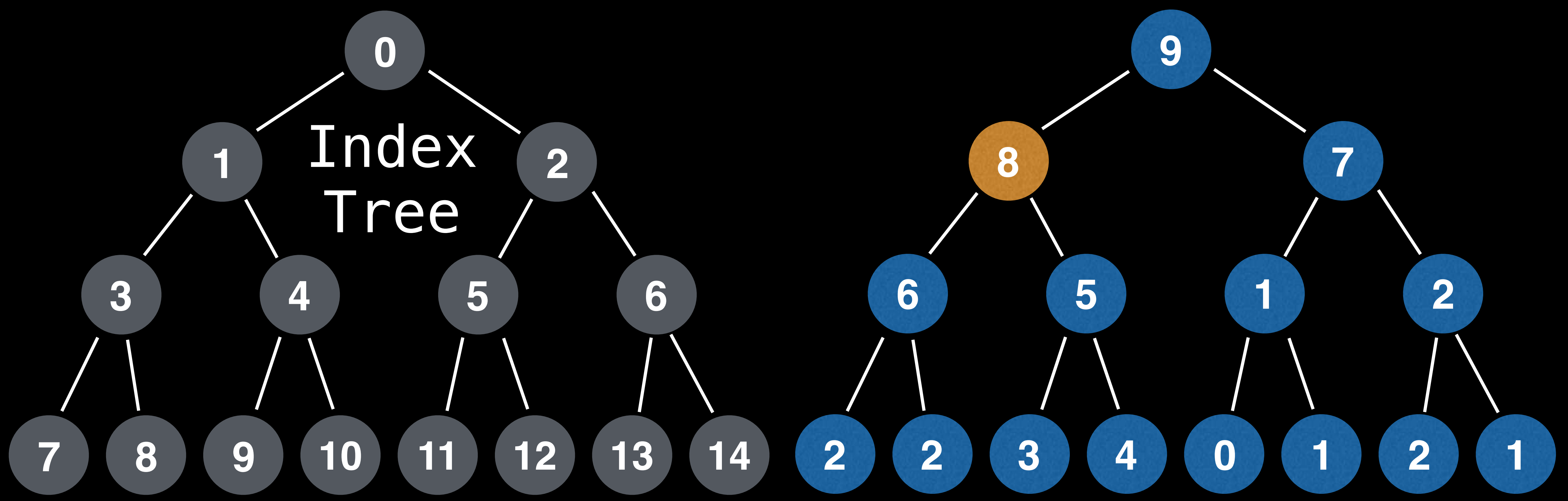




Binary Heap Representation

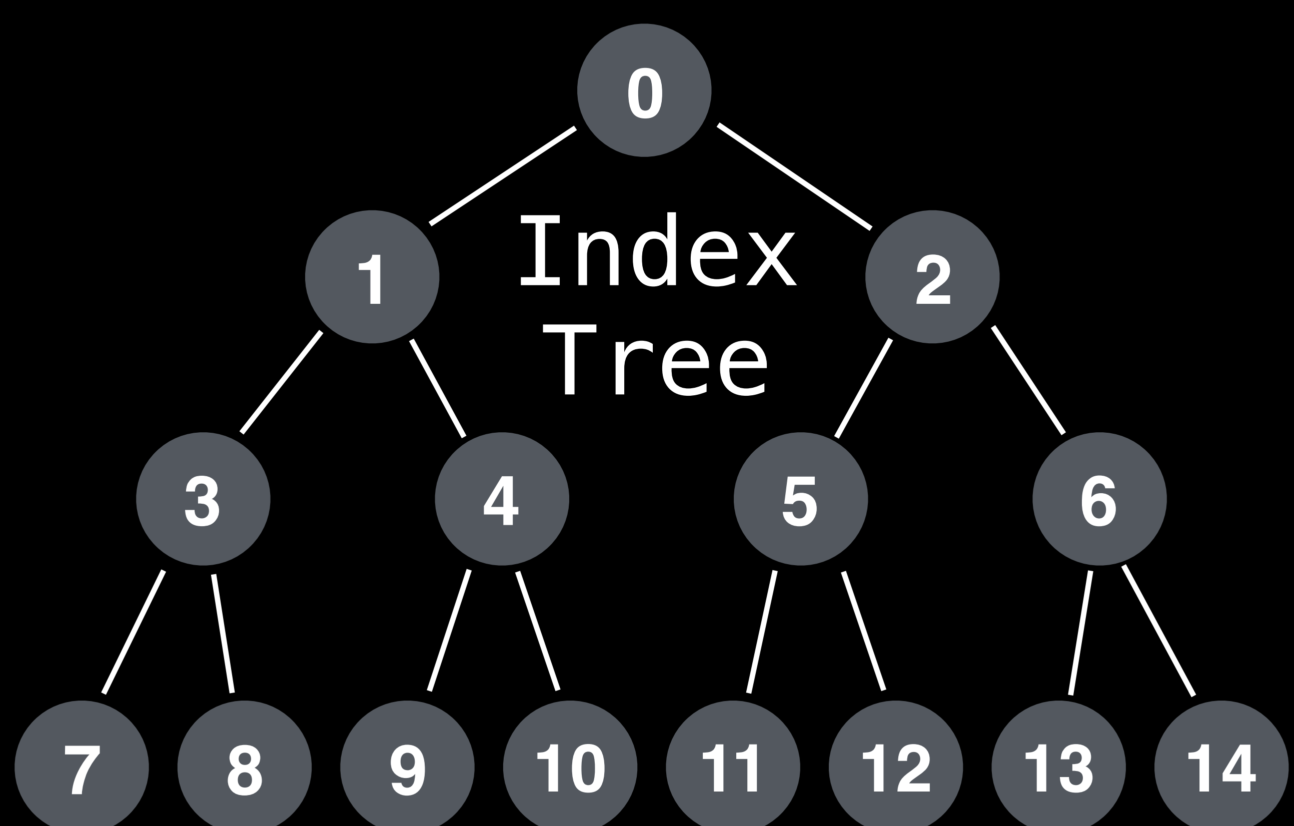
9	8	7	6	5	1	2	2	2	3	4	0	1	2	1

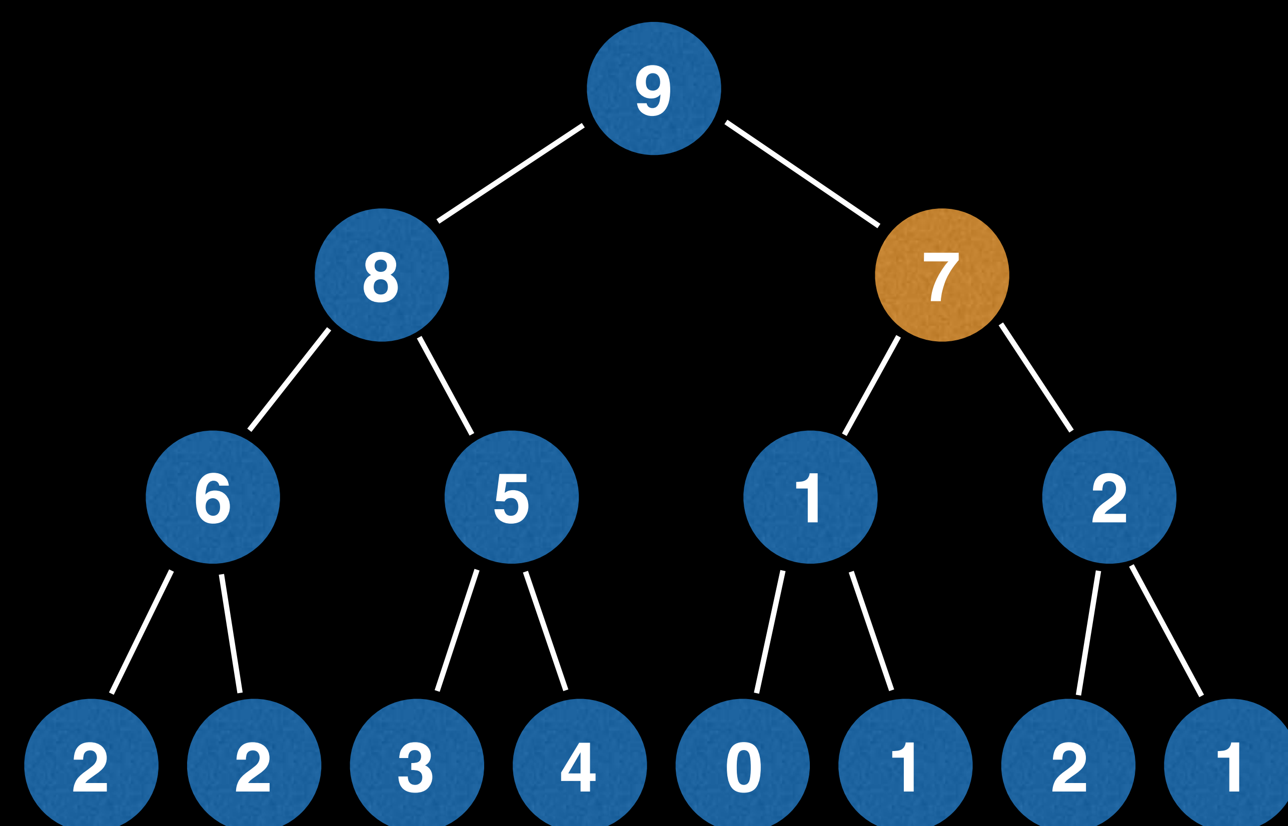
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14



Binary Heap Representation

9	8	7	6	5	1	2	2	2	3	4	0	1	2	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

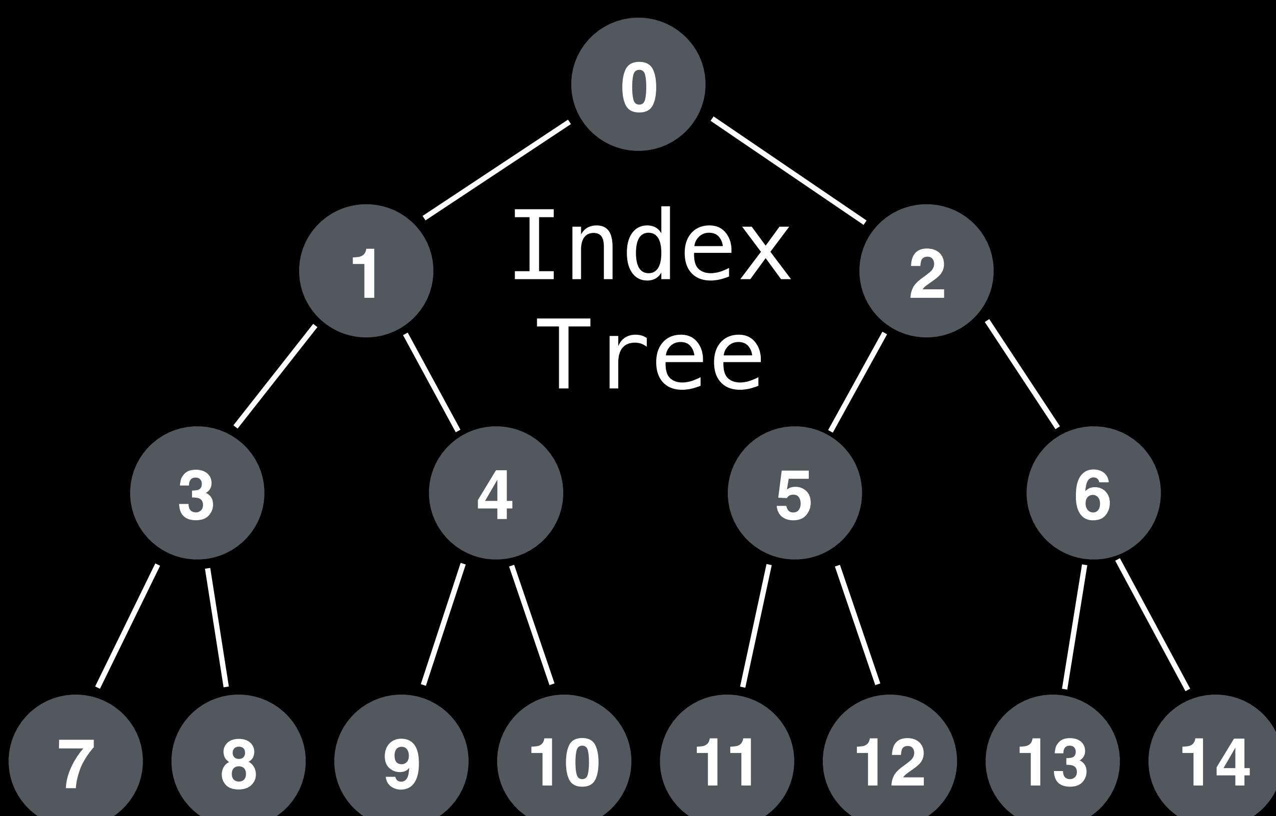


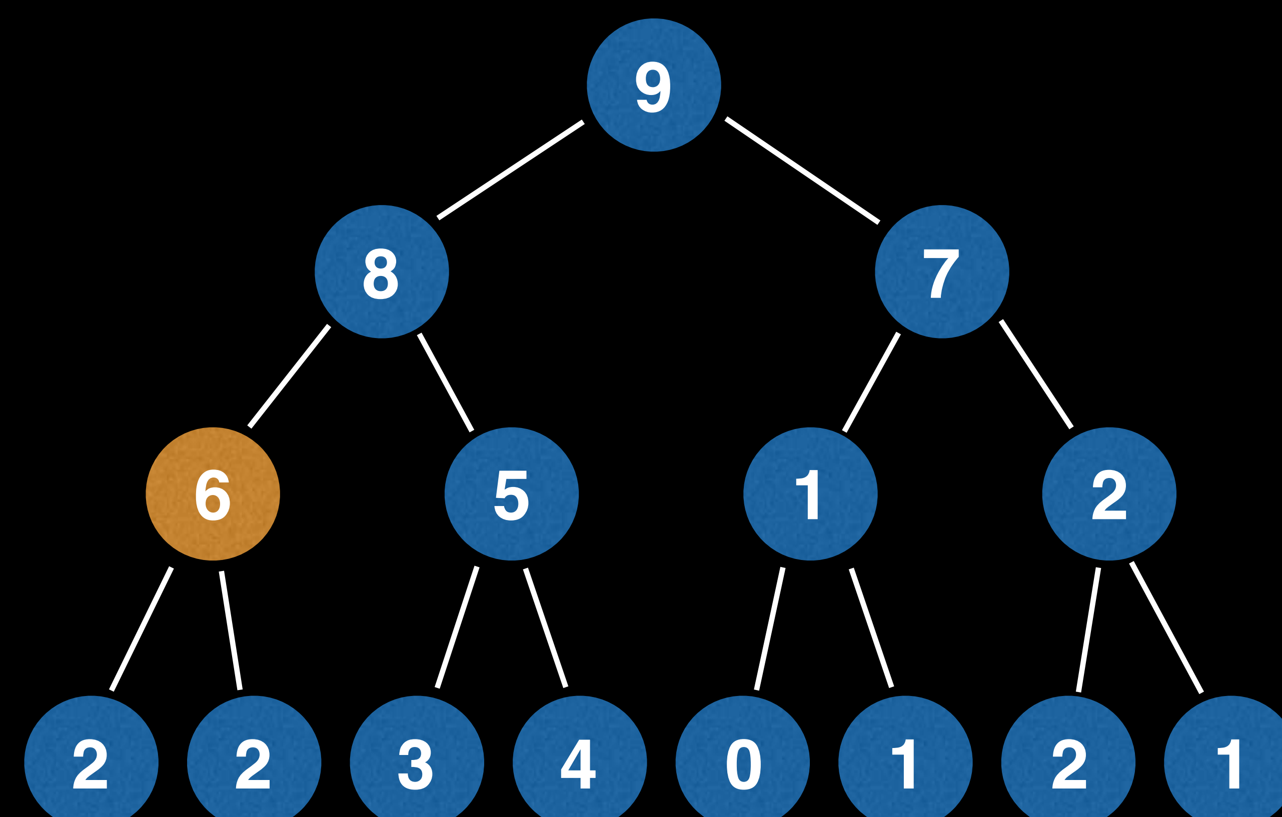


Binary Heap Representation

9	8	7	6	5	1	2	2	2	3	4	0	1	2	1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

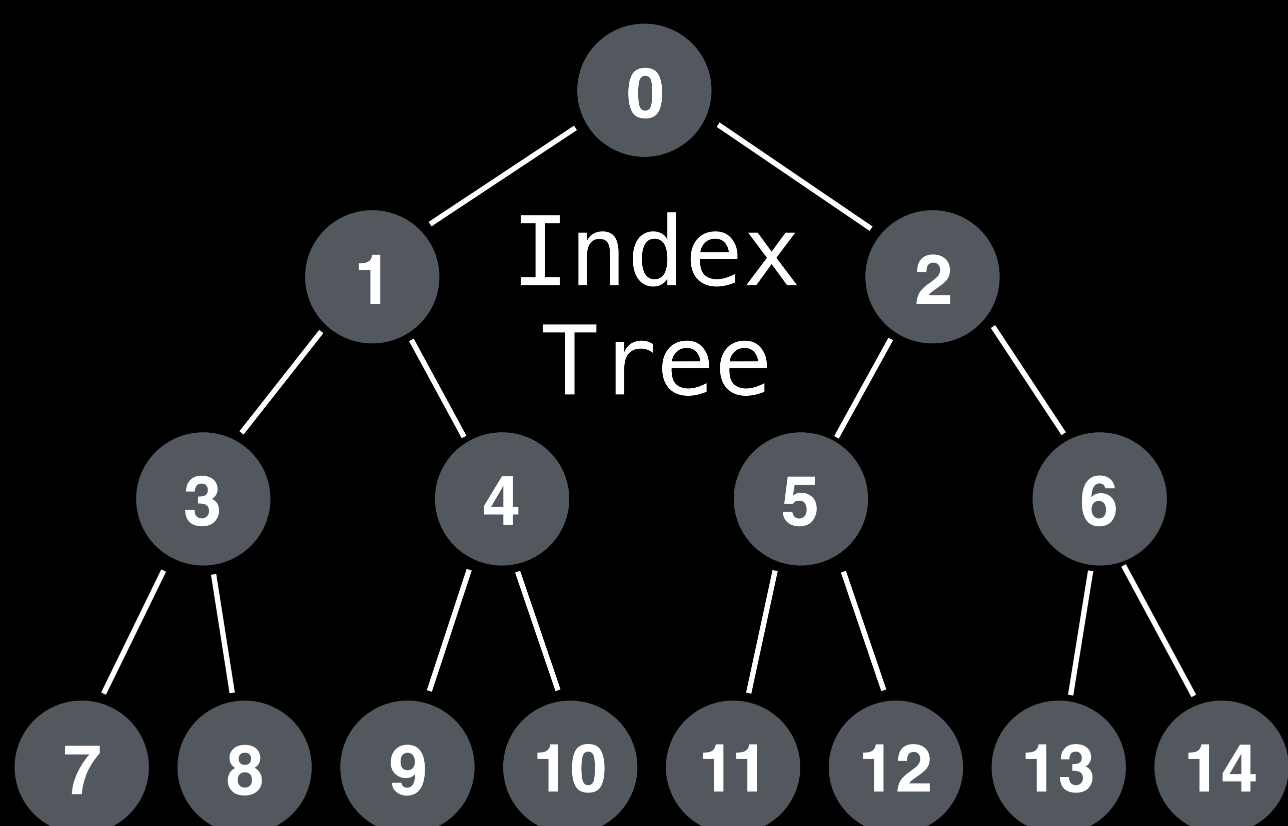


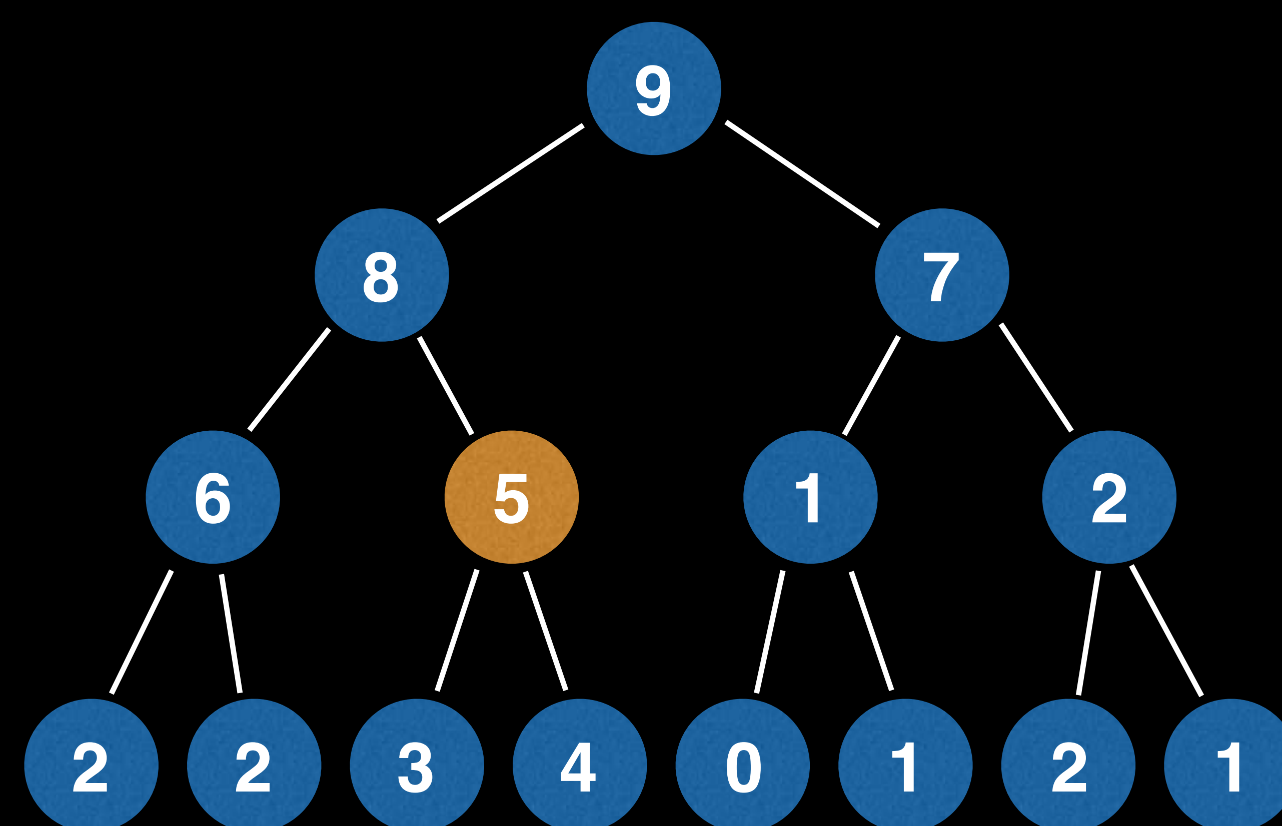


Binary Heap Representation

9	8	7	6	5	1	2	2	2	3	4	0	1	2	1

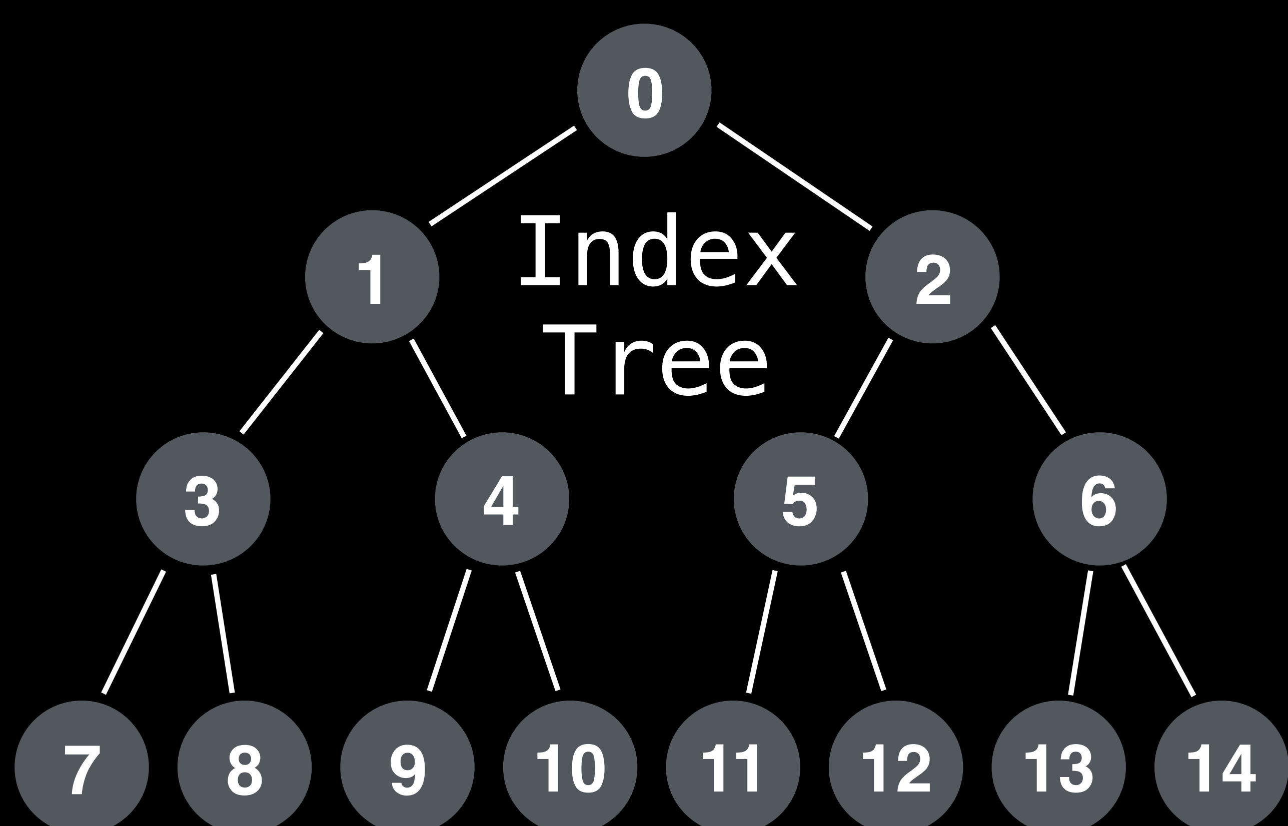
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

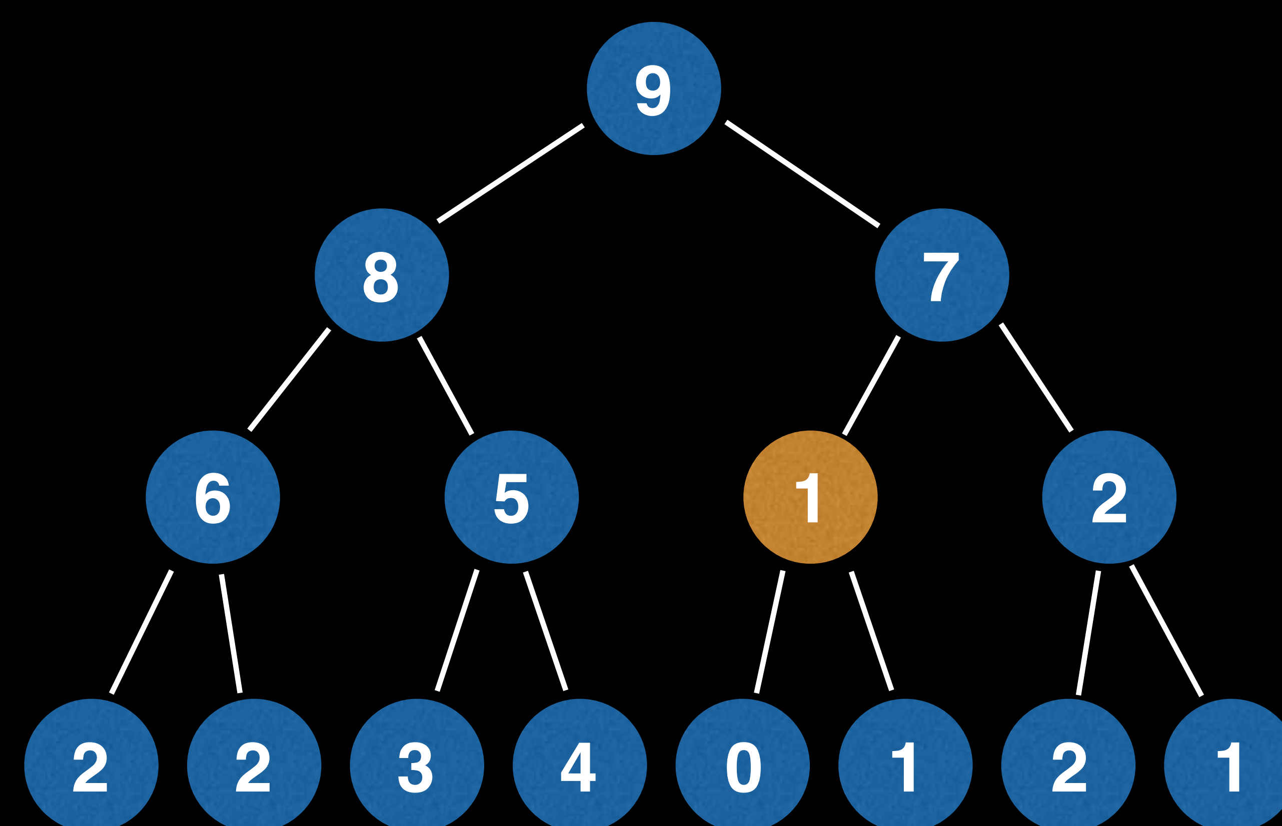




Binary Heap Representation

9	8	7	6	5	1	2	2	2	3	4	0	1	2	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

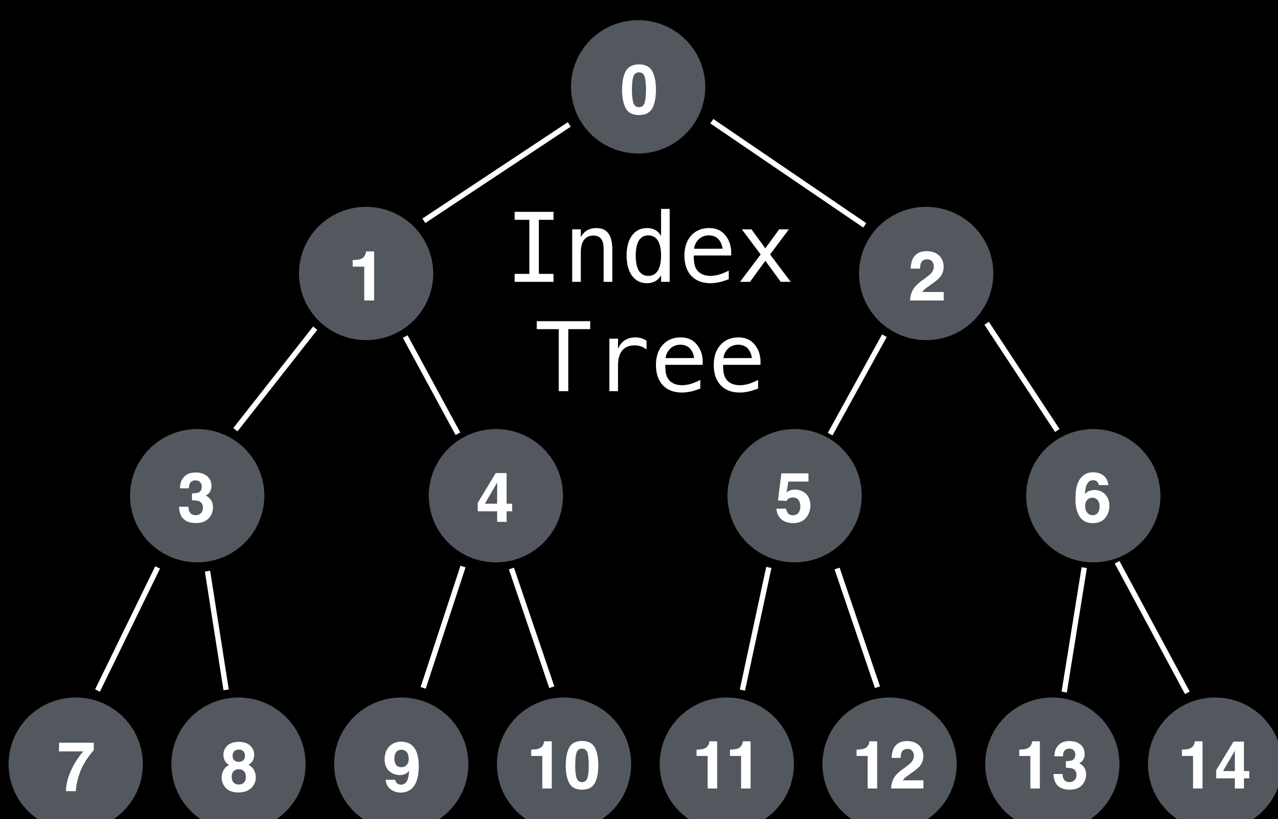


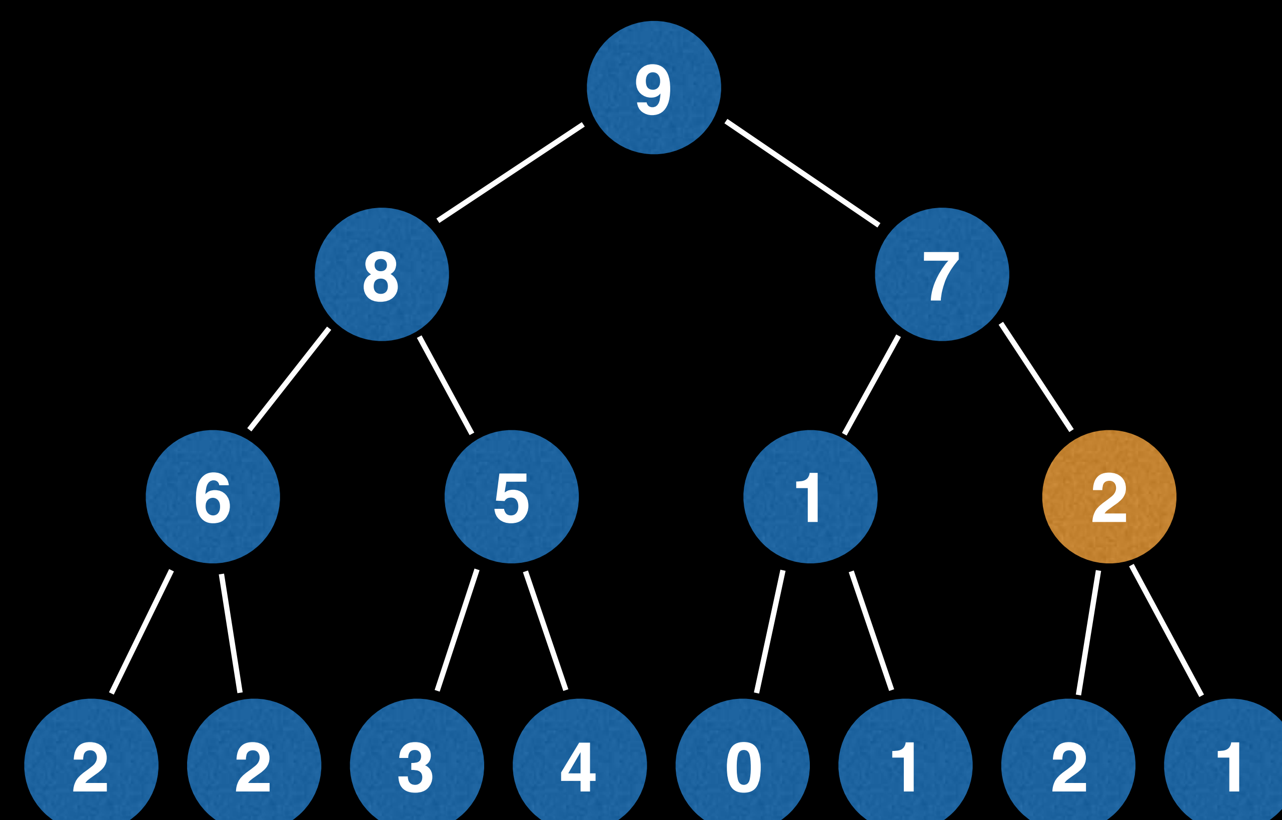


Binary Heap Representation

9	8	7	6	5	1	2	2	2	3	4	0	1	2	1

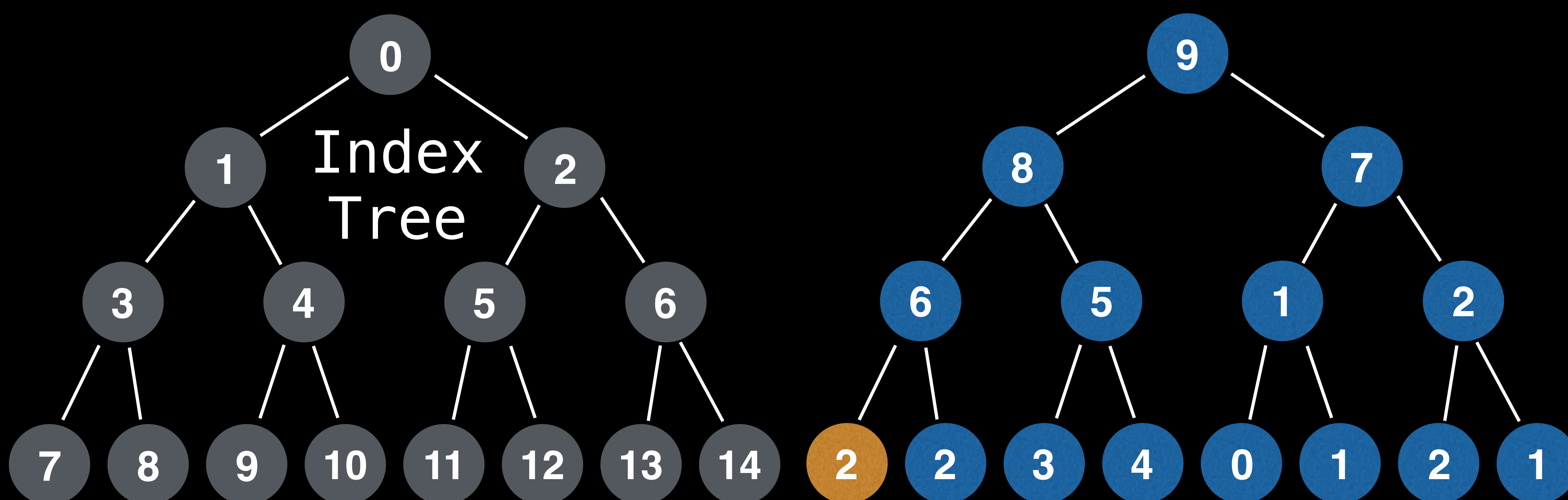
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14





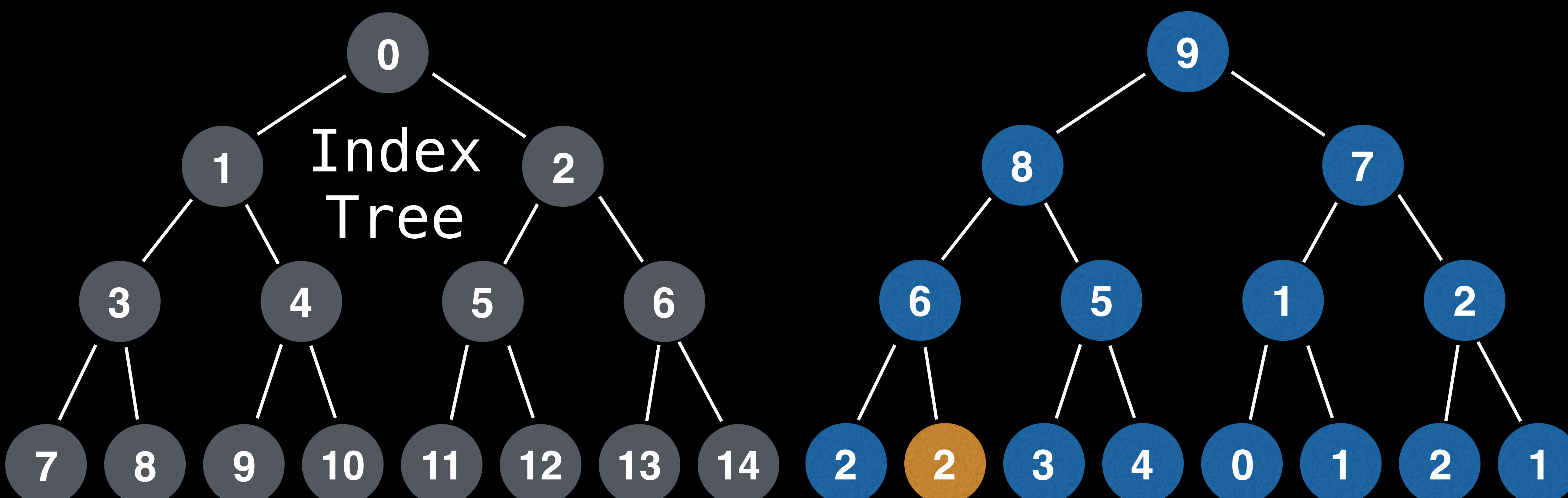
Binary Heap Representation

9	8	7	6	5	1	2	2	2	3	4	0	1	2	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14



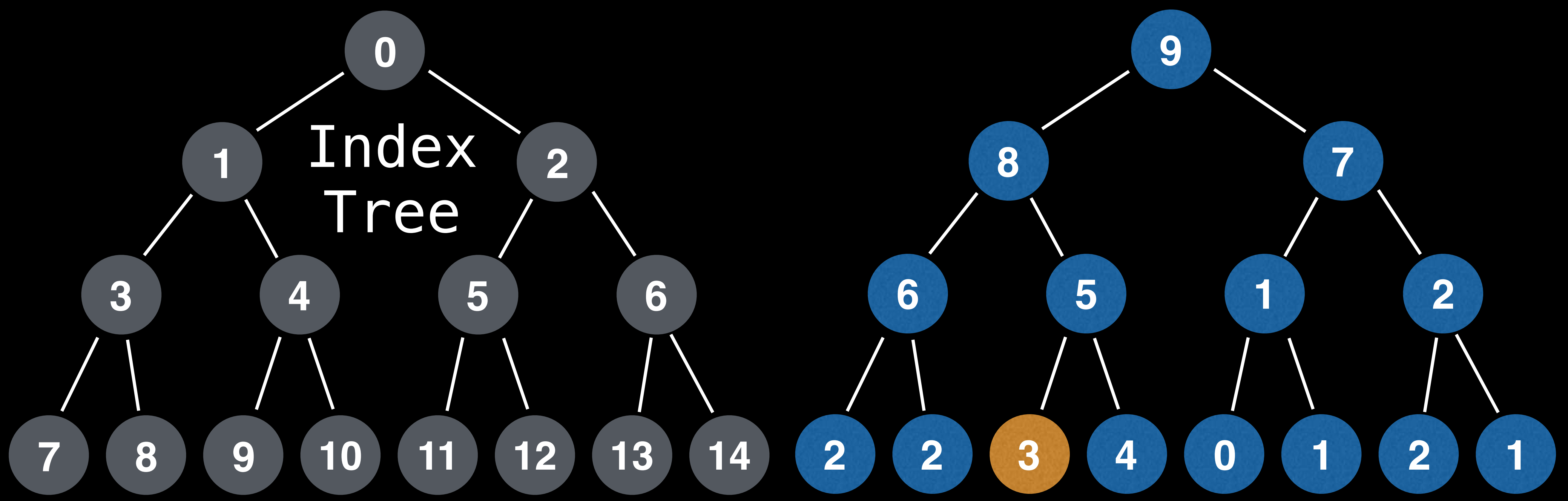
Binary Heap Representation

9	8	7	6	5	1	2	2	2	3	4	0	1	2	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14



Binary Heap Representation

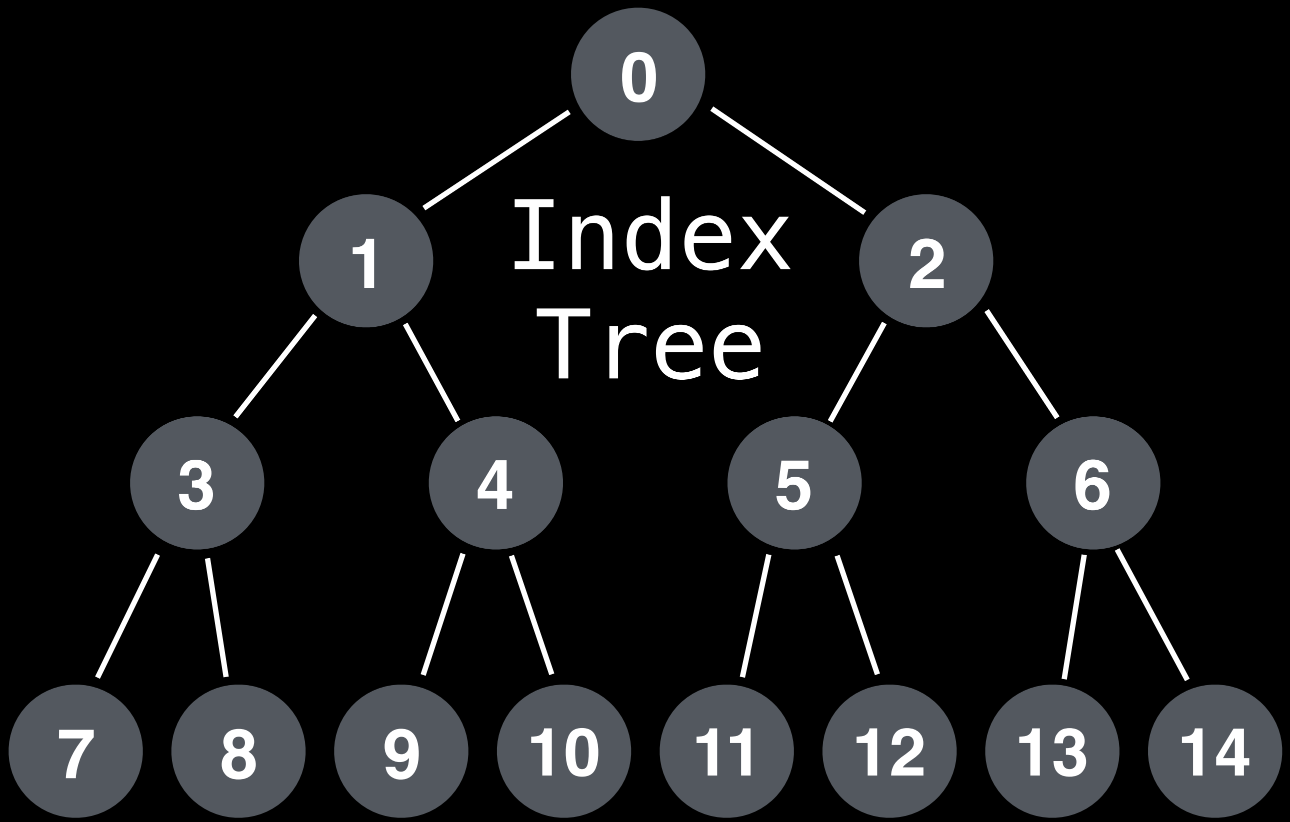
9	8	7	6	5	1	2	2	2	3	4	0	1	2	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

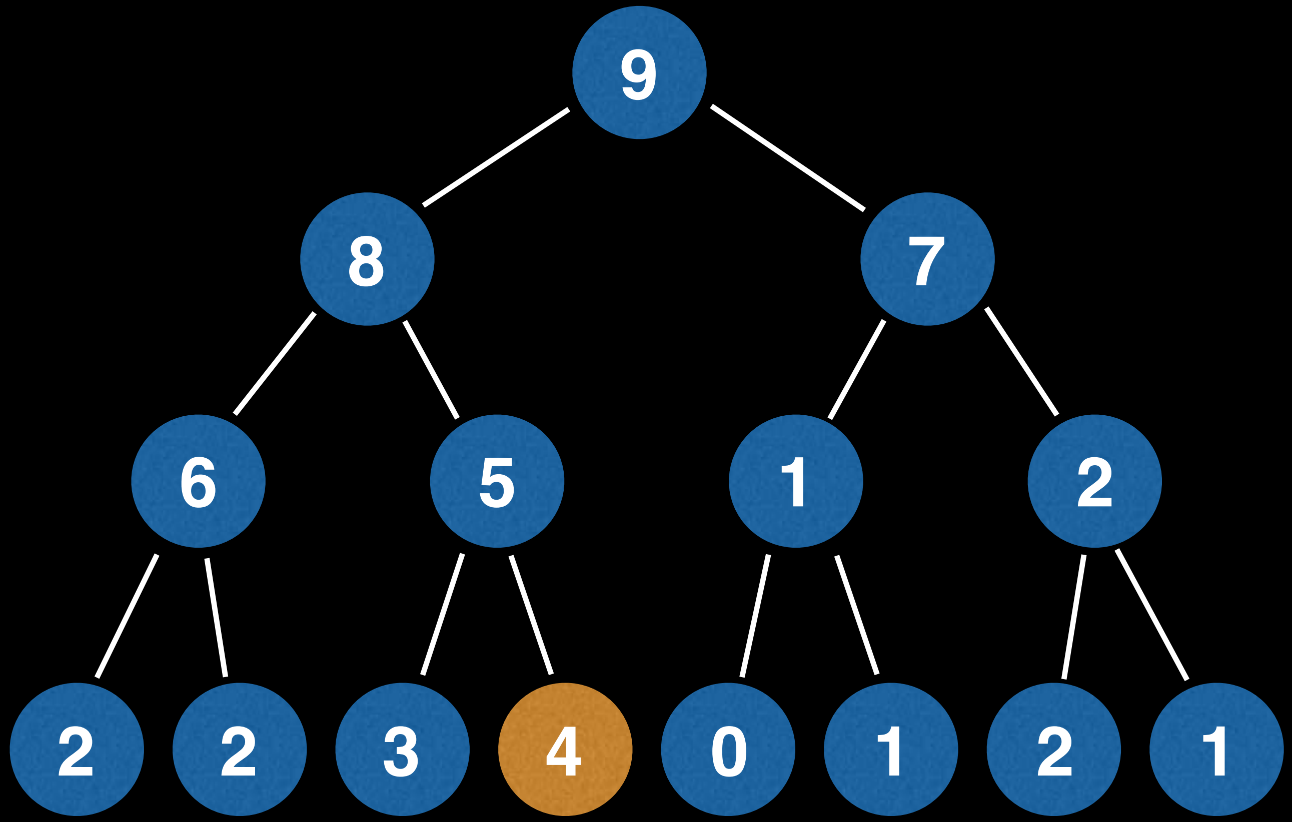


Binary Heap Representation

9	8	7	6	5	1	2	2	2	3	4	0	1	2	1

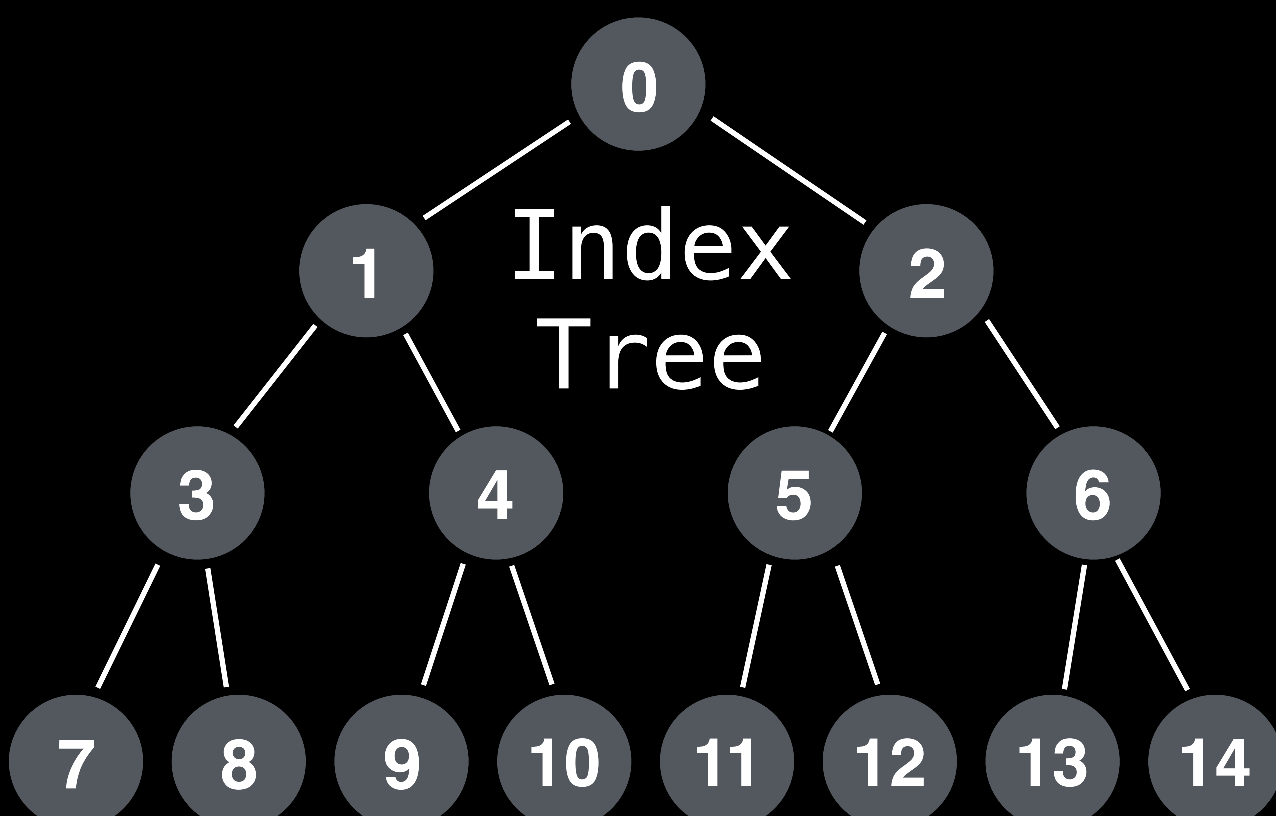
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

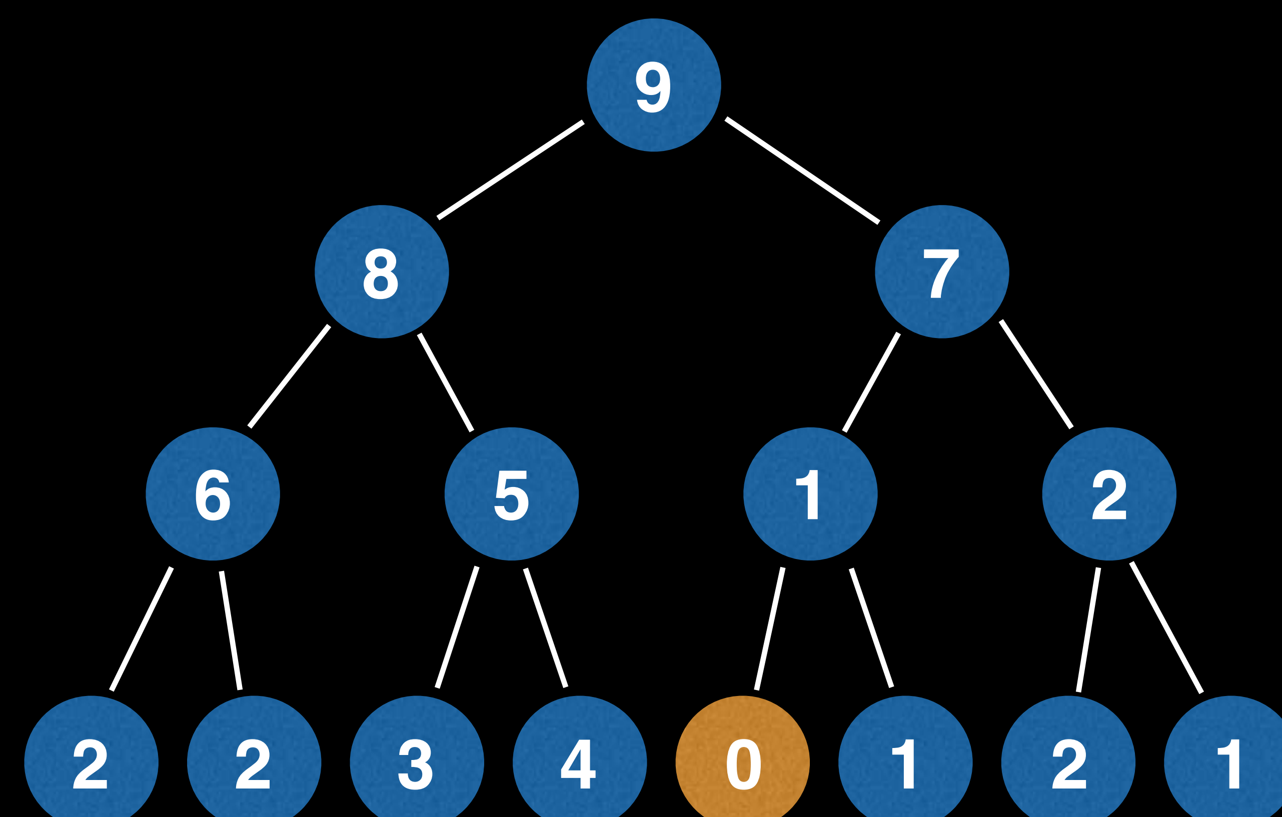




Binary Heap Representation

9	8	7	6	5	1	2	2	2	3	4	0	1	2	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

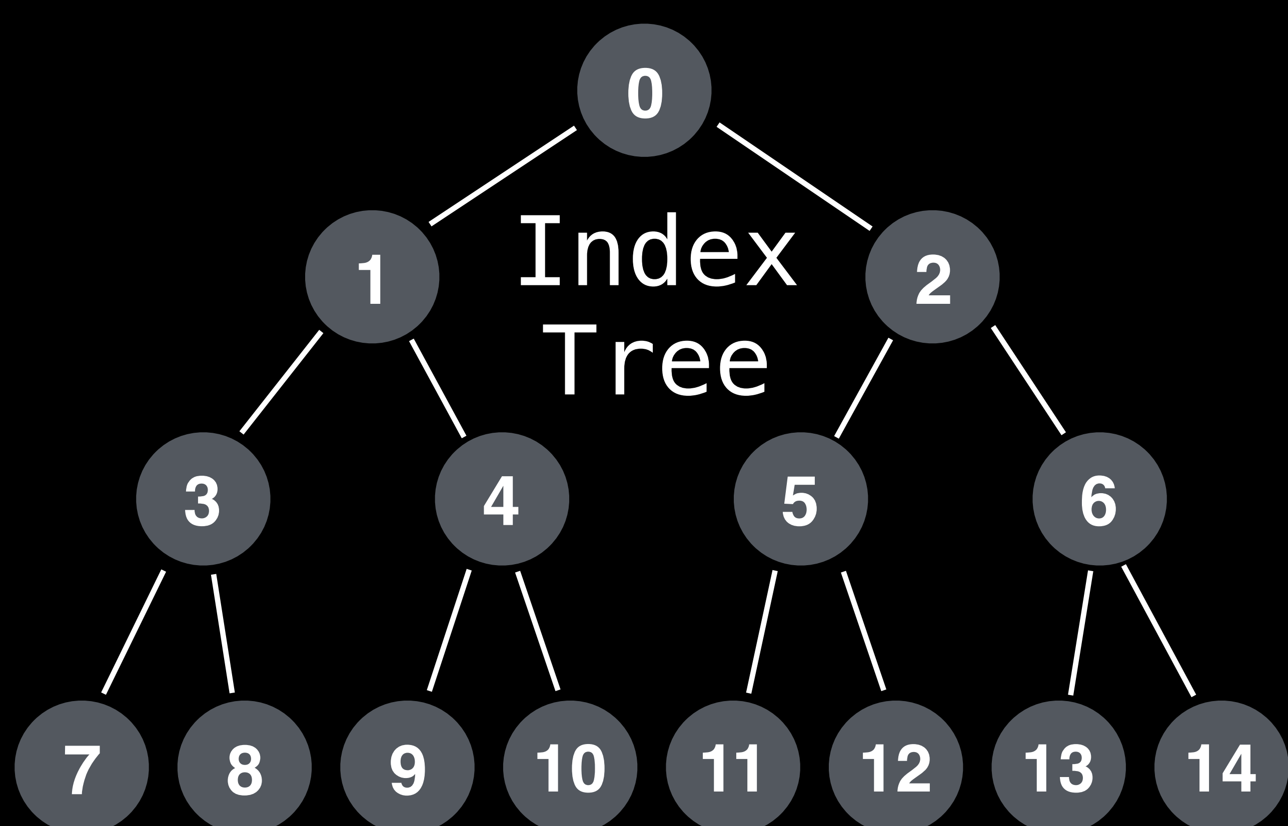


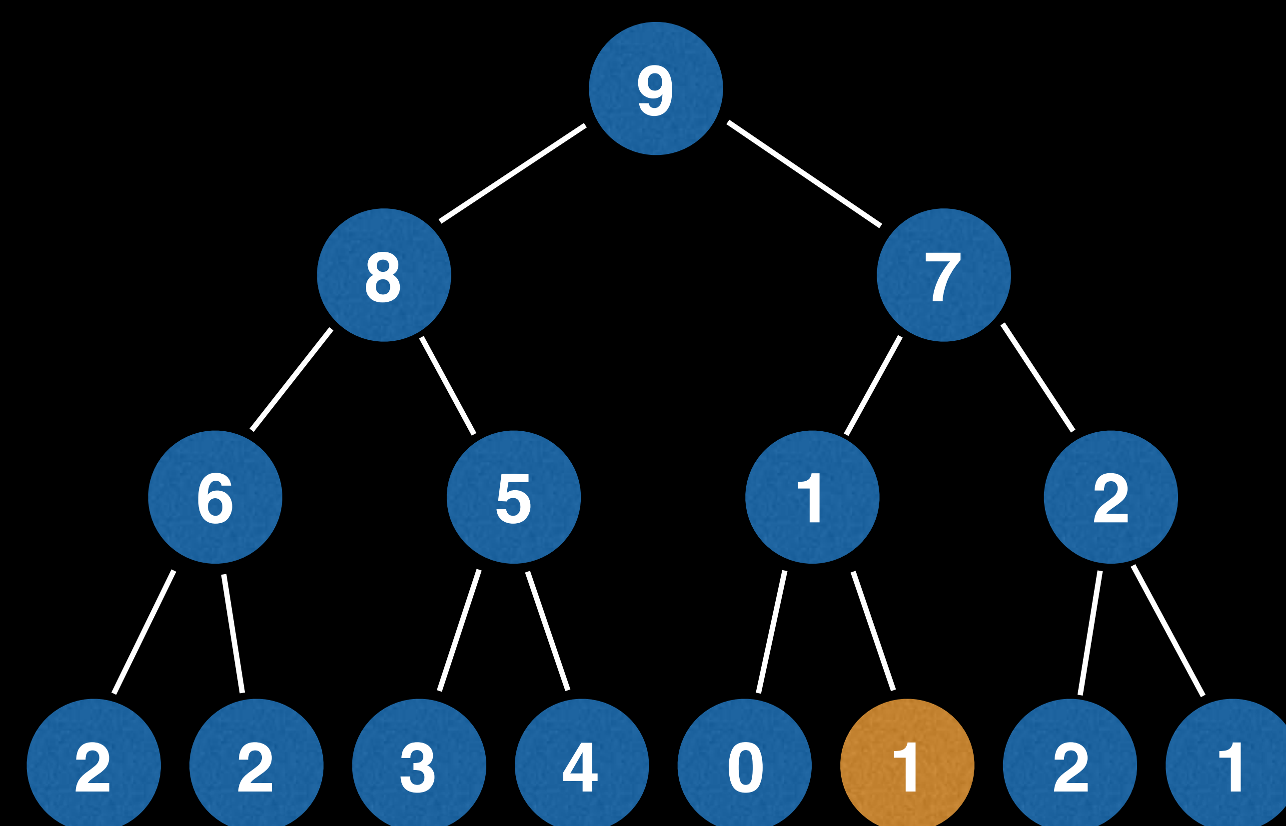


Binary Heap Representation

9	8	7	6	5	1	2	2	2	3	4	0	1	2	1

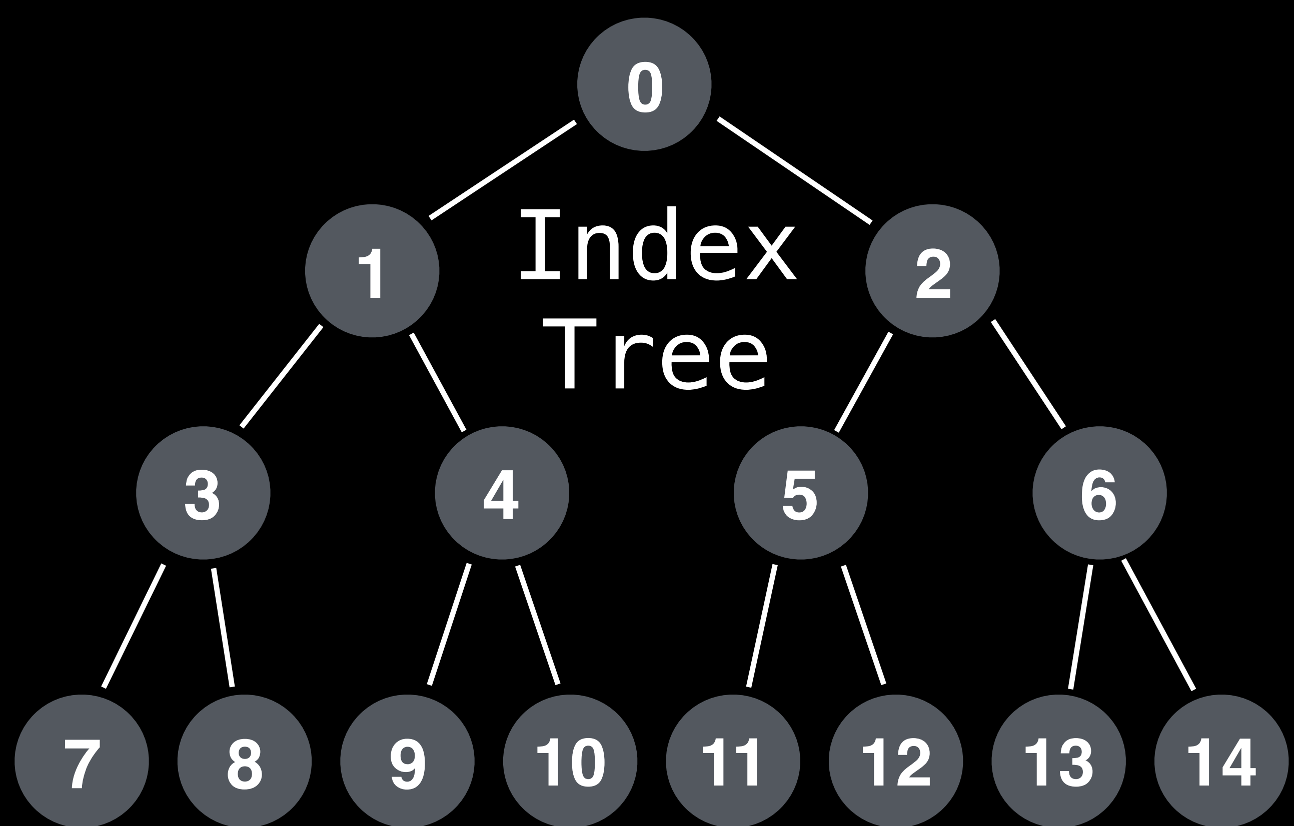
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

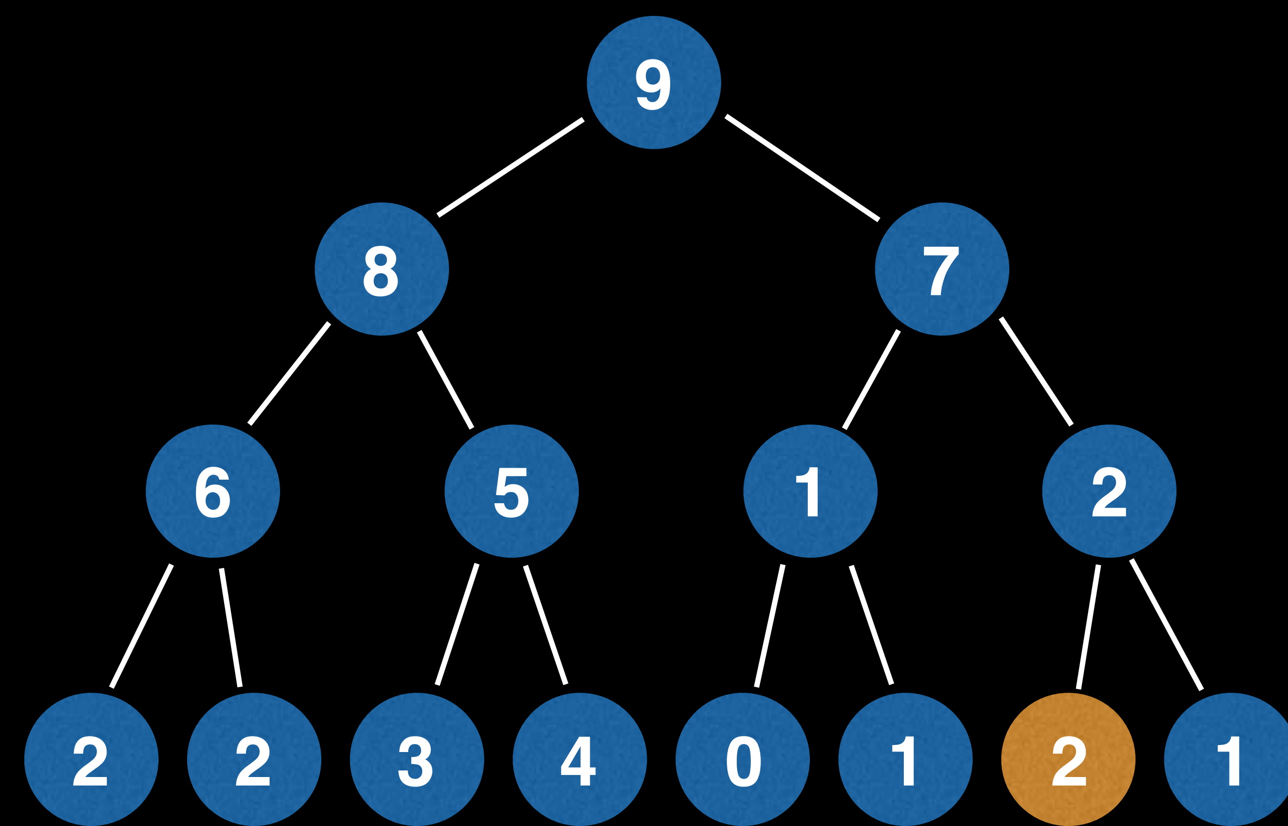




Binary Heap Representation

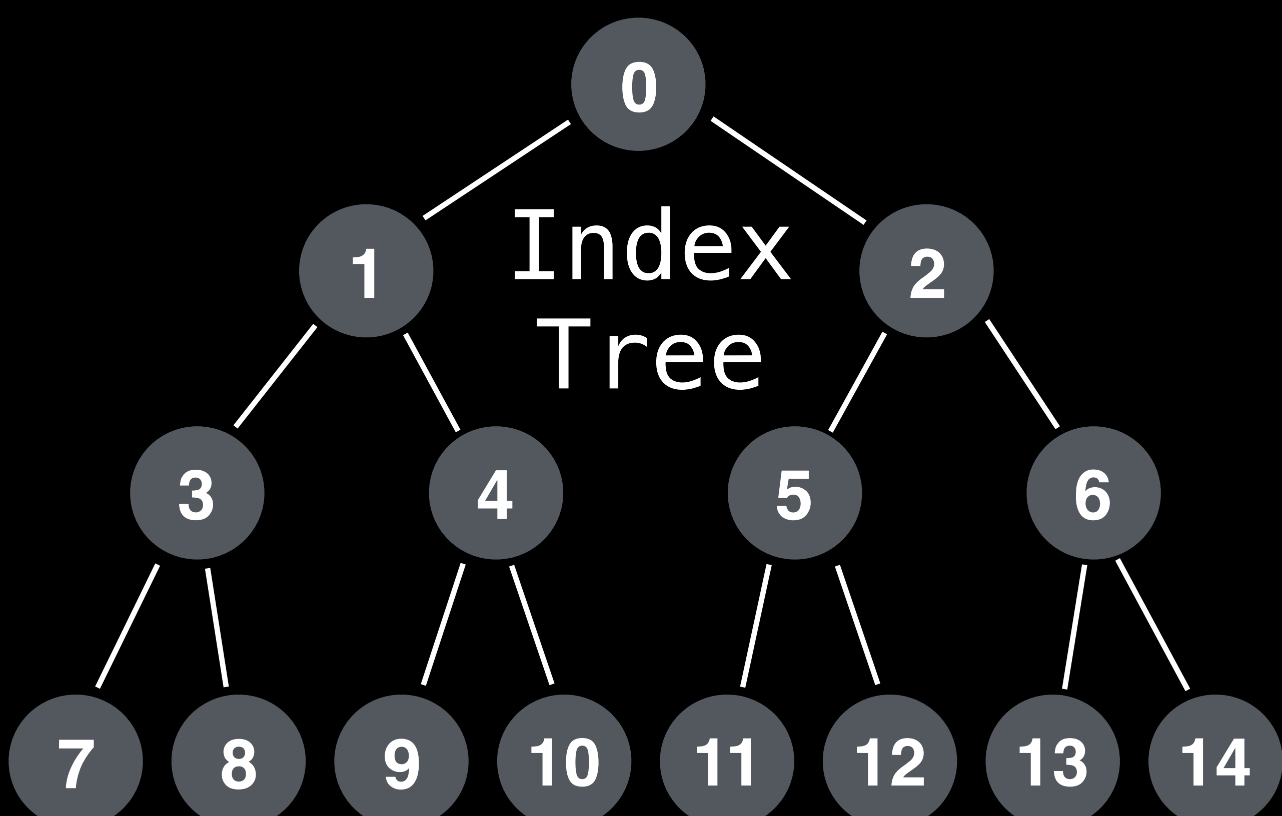
9	8	7	6	5	1	2	2	2	3	4	0	1	2	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

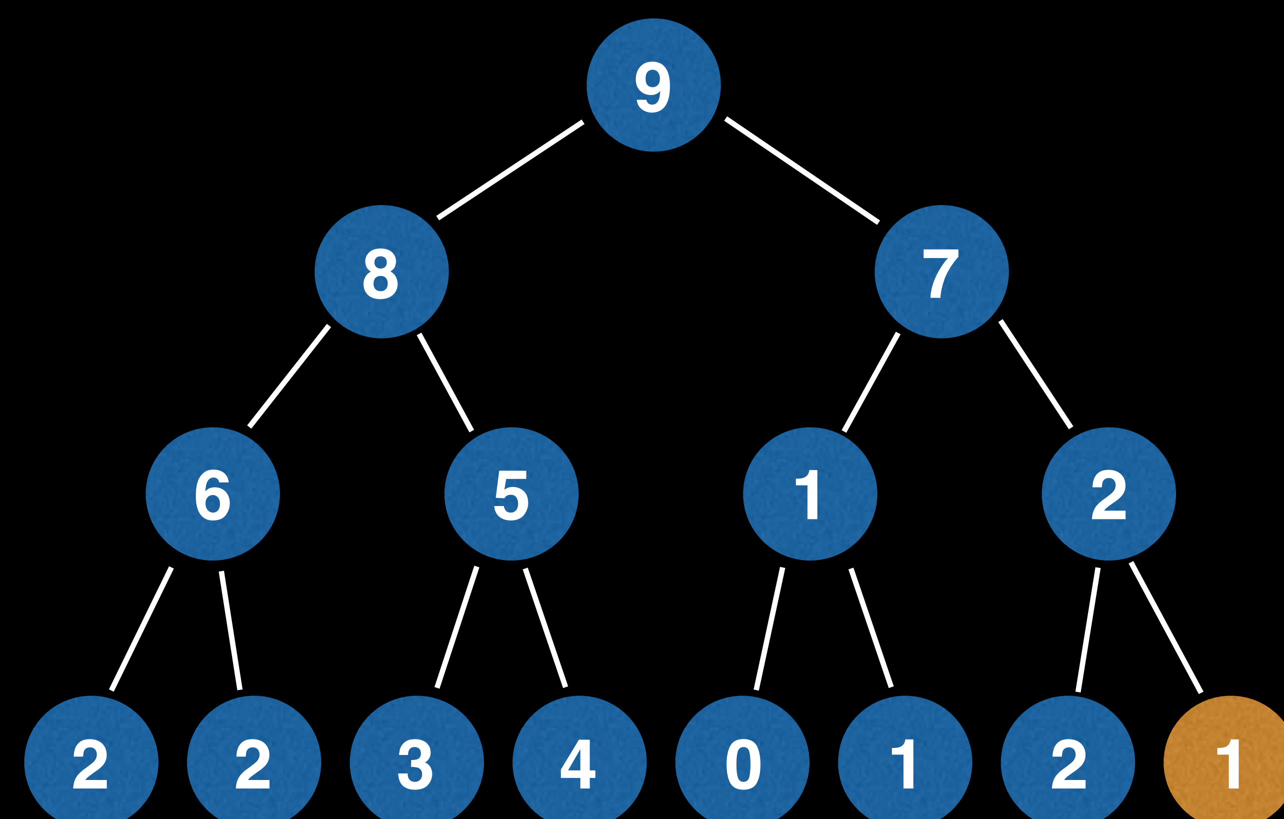




Binary Heap Representation

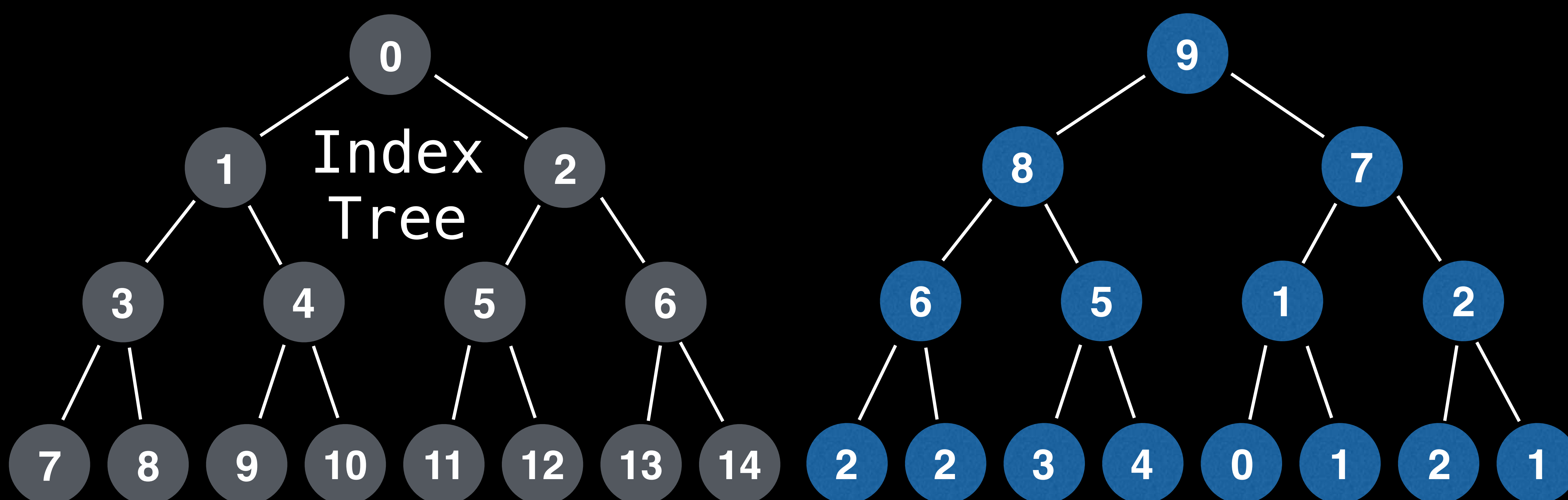
9	8	7	6	5	1	2	2	2	3	4	0	1	2	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14





Binary Heap Representation

9	8	7	6	5	1	2	2	2	3	4	0	1	2	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14



Binary Heap Representation

9	8	7	6	5	1	2	2	2	3	4	0	1	2	1

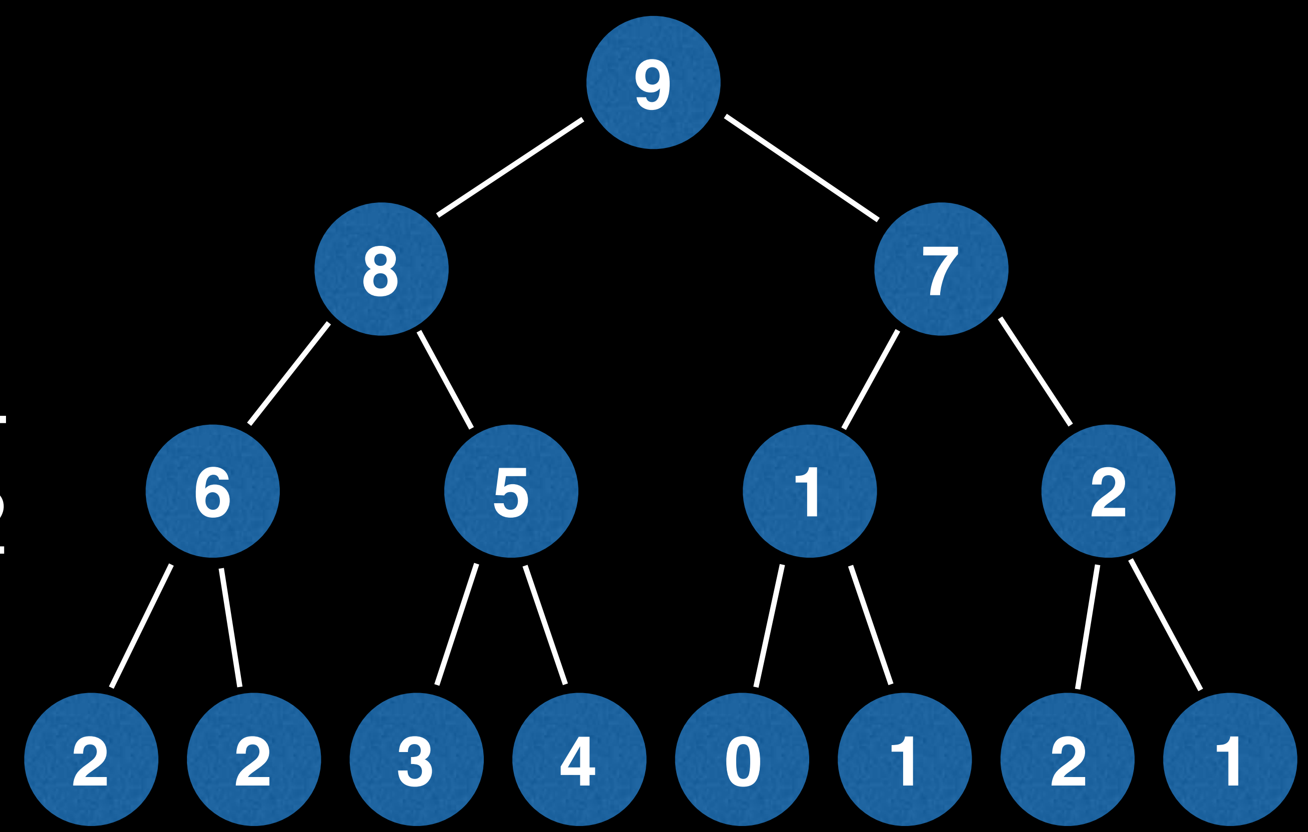
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Let *i* be the parent node index

Left child index: 2i + 1

Right child index: 2i + 2

(zero based)



Binary Heap Representation

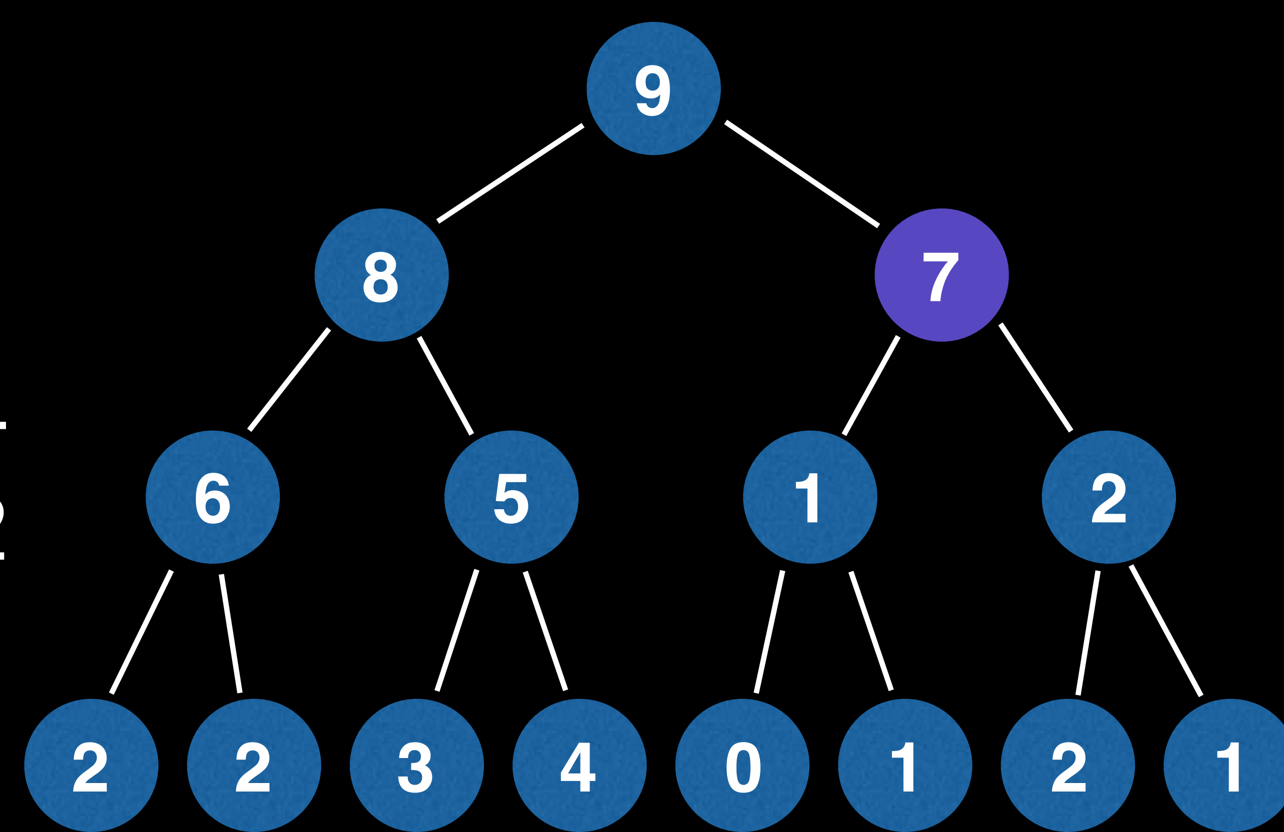
9	8	7	6	5	1	2	2	2	3	4	0	1	2	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Let *i* be the parent node index

Left child index: 2i + 1

Right child index: 2i + 2

(zero based)



Binary Heap Representation

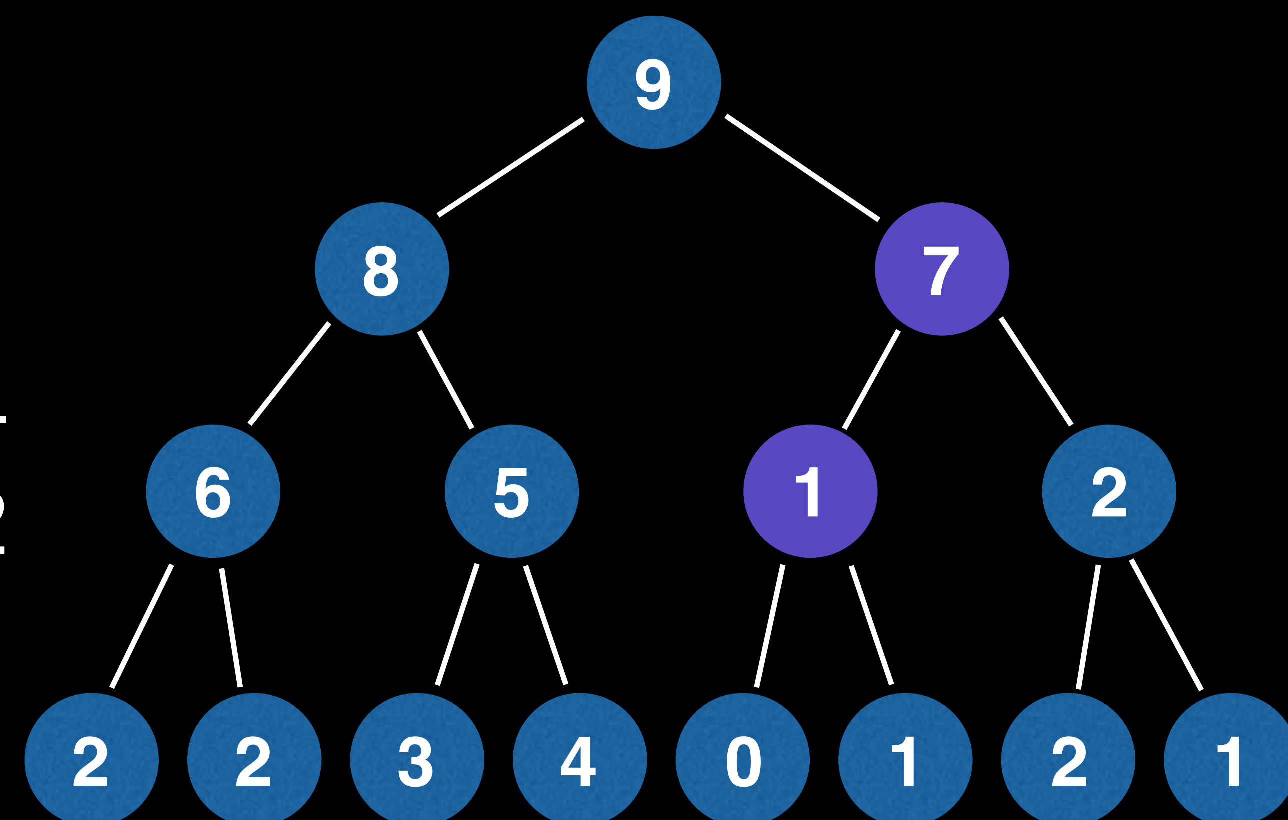
9	8	7	6	5	1	2	2	2	3	4	0	1	2	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Let i be the parent node index

Left child index: $2i + 1$

Right child index: $2i + 2$

(zero based)



Binary Heap Representation

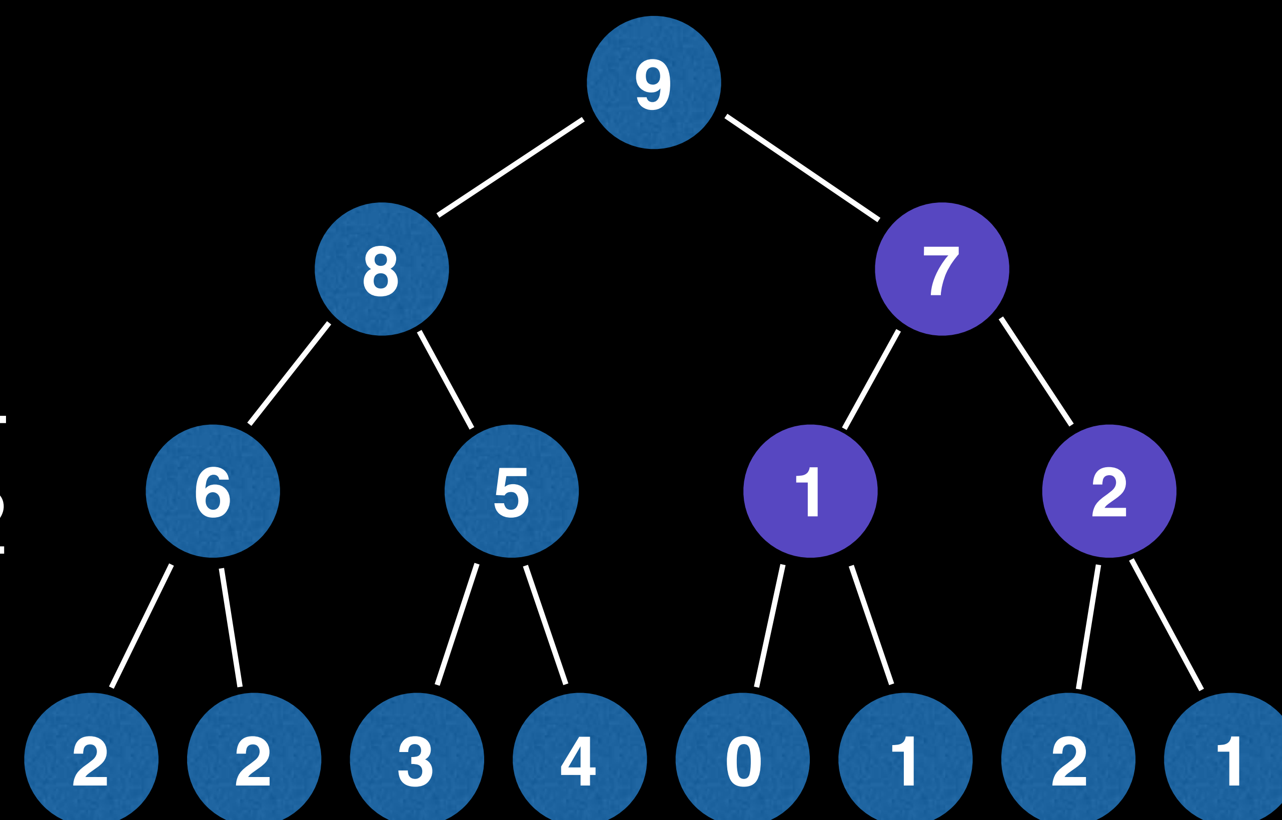
9	8	7	6	5	1	2	2	2	3	4	0	1	2	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Let *i* be the parent node index

Left child index: 2i + 1

Right child index: 2i + 2

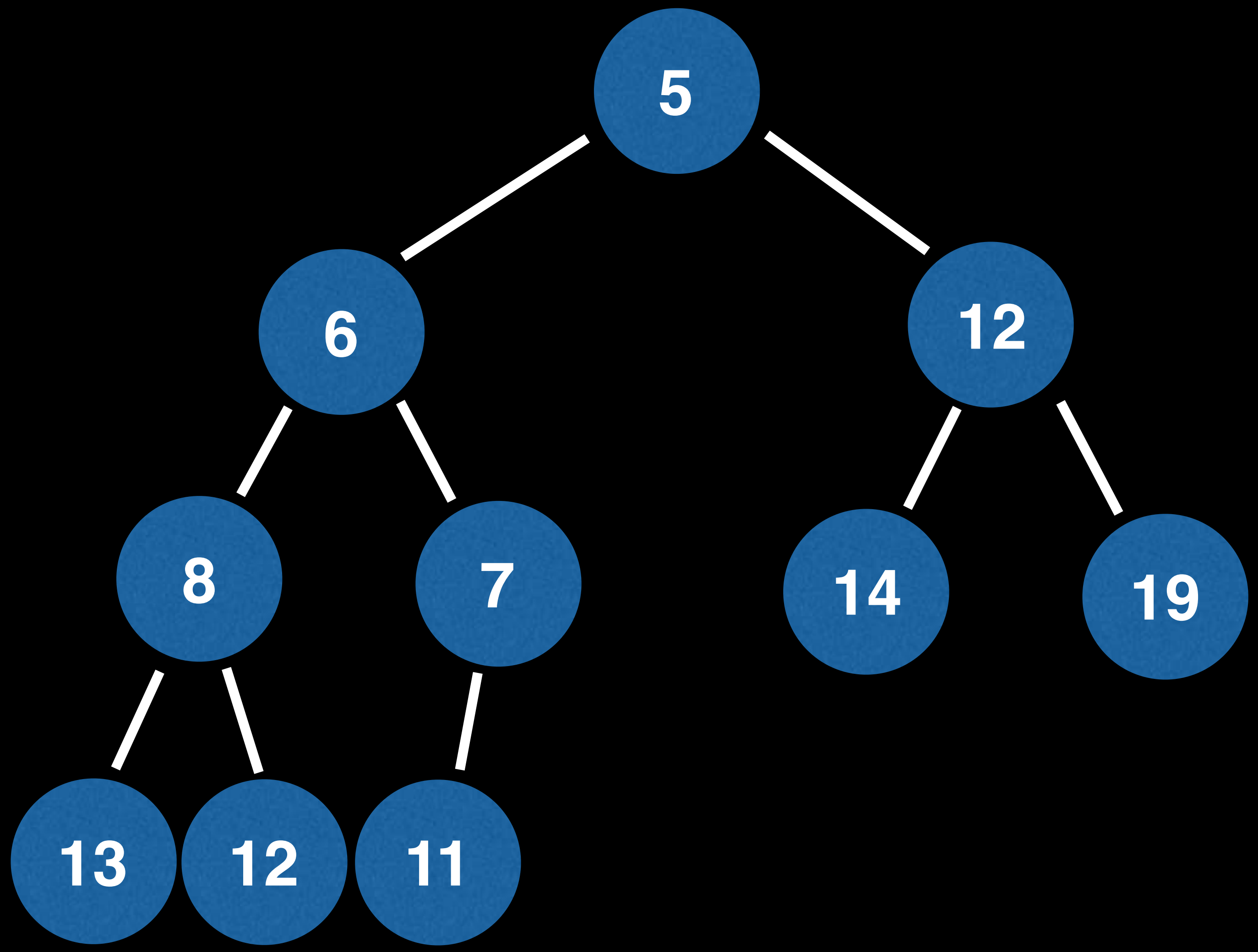
(zero based)



Adding Elements to Binary Heap

Instructions:

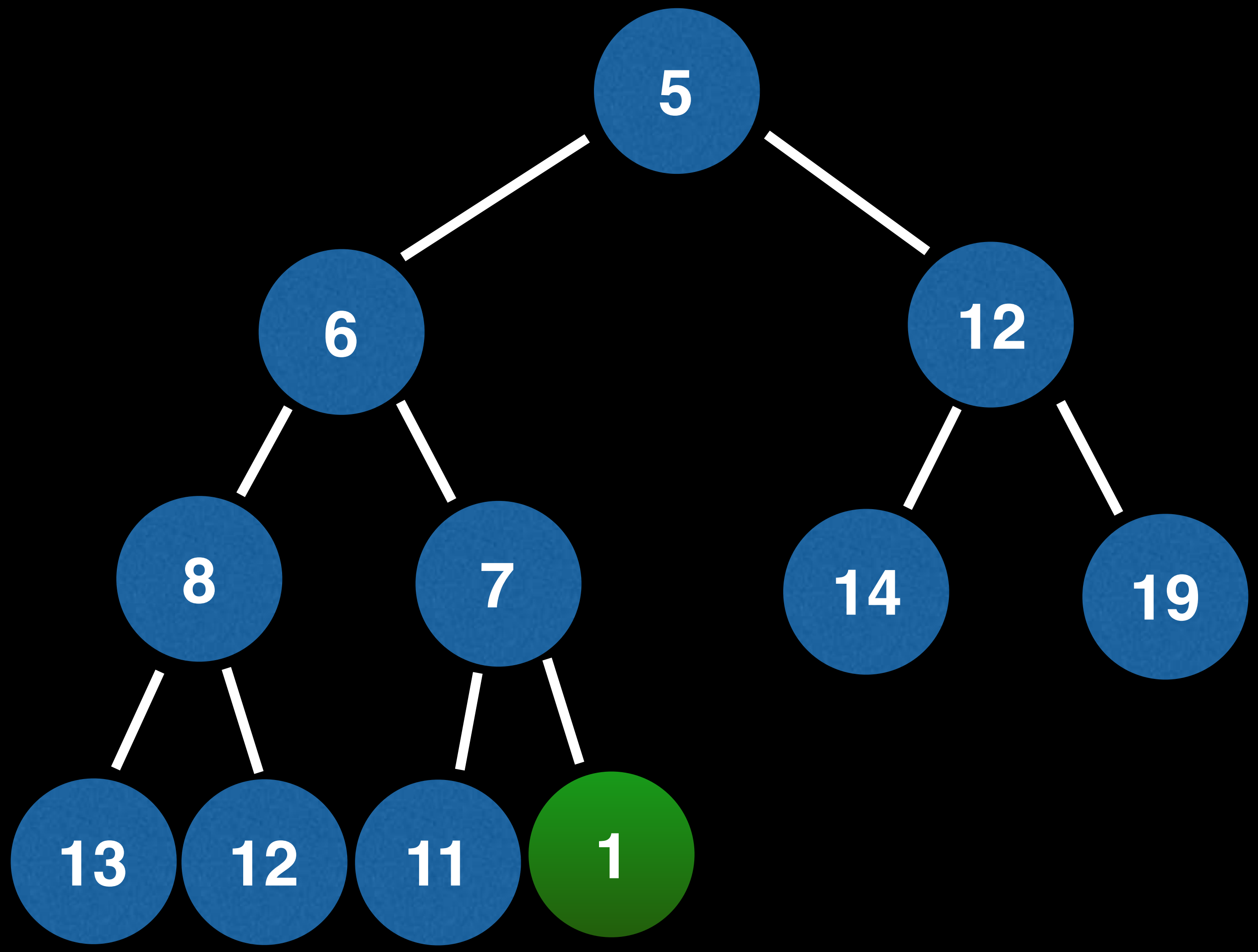
```
Insert(1)
Insert(13)
Insert(4)
Insert(0)
Insert(10)
```



Adding Elements to Binary Heap

Instructions:

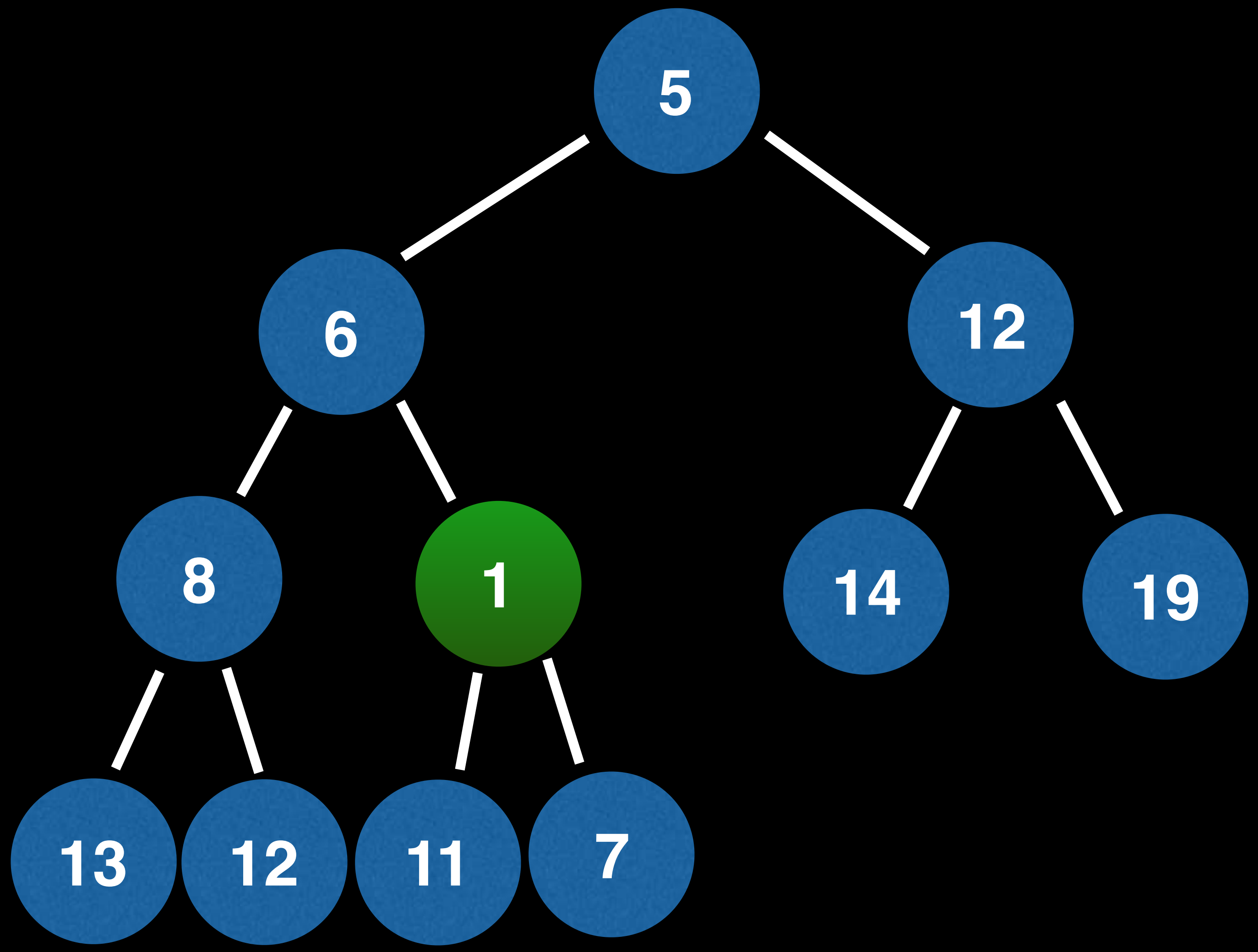
- Insert(1)
- Insert(13)
- Insert(4)
- Insert(0)
- Insert(10)



Adding Elements to Binary Heap

Instructions:

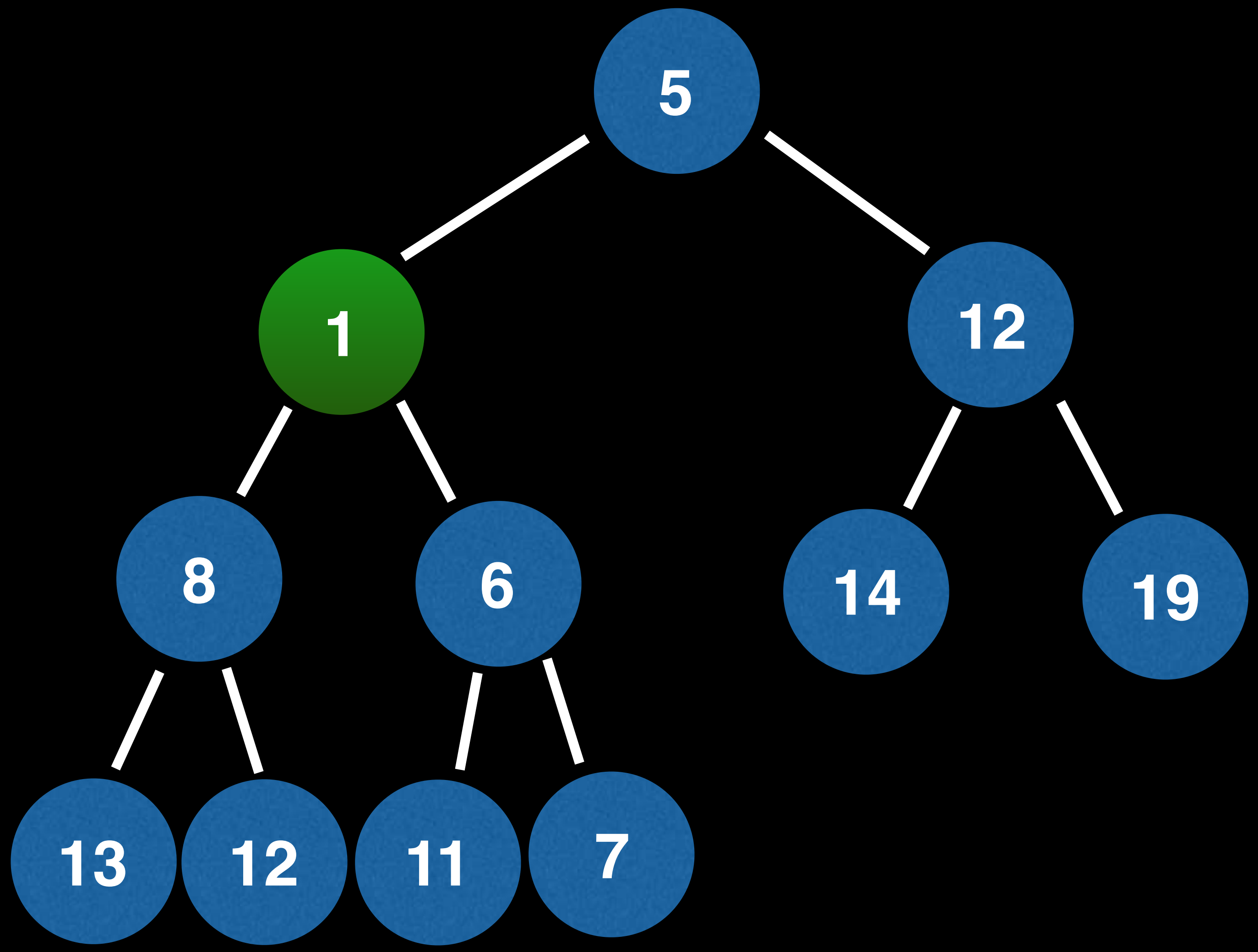
- Insert(1)
- Insert(13)
- Insert(4)
- Insert(0)
- Insert(10)



Adding Elements to Binary Heap

Instructions:

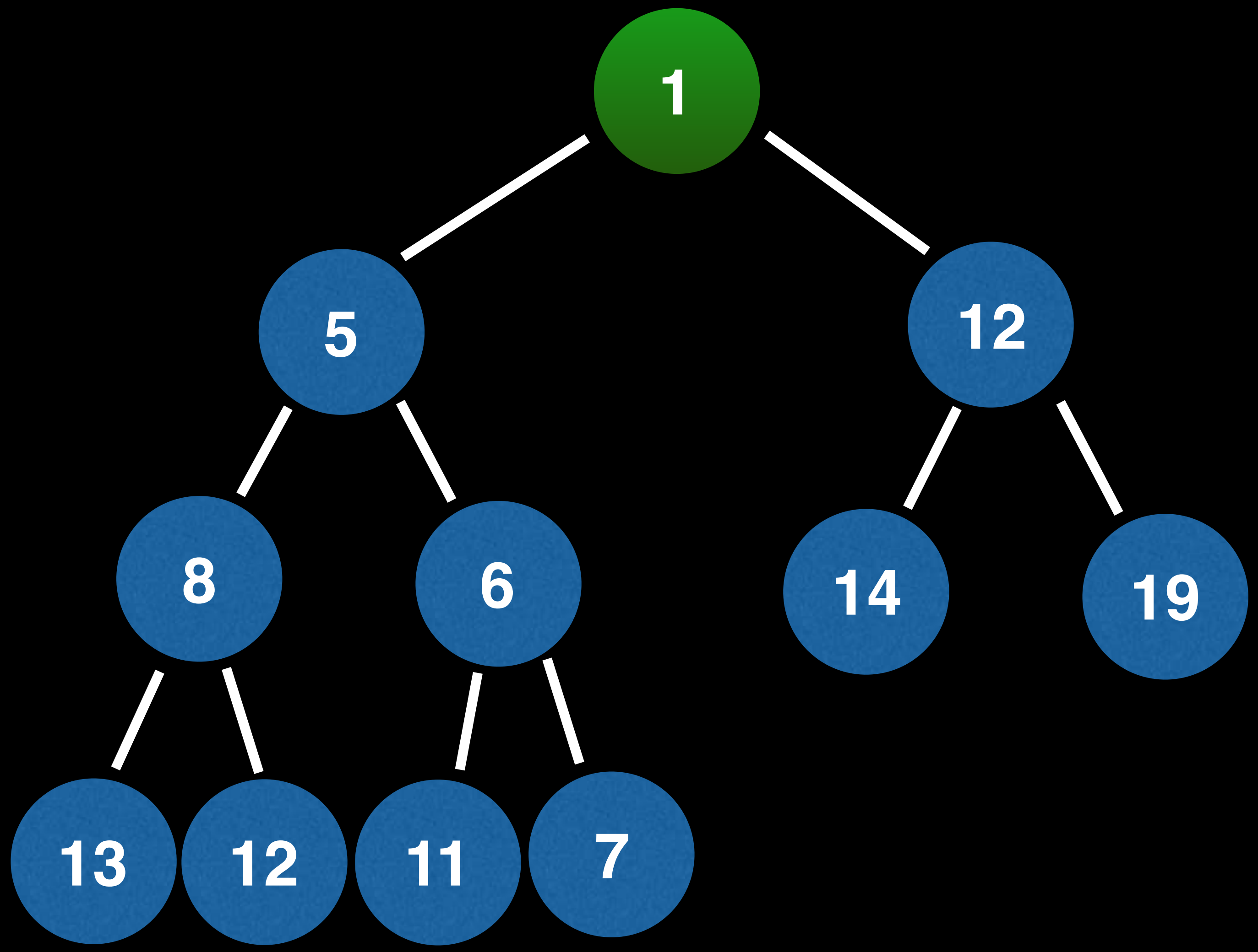
Insert(1)
Insert(13)
Insert(4)
Insert(0)
Insert(10)



Adding Elements to Binary Heap

Instructions:

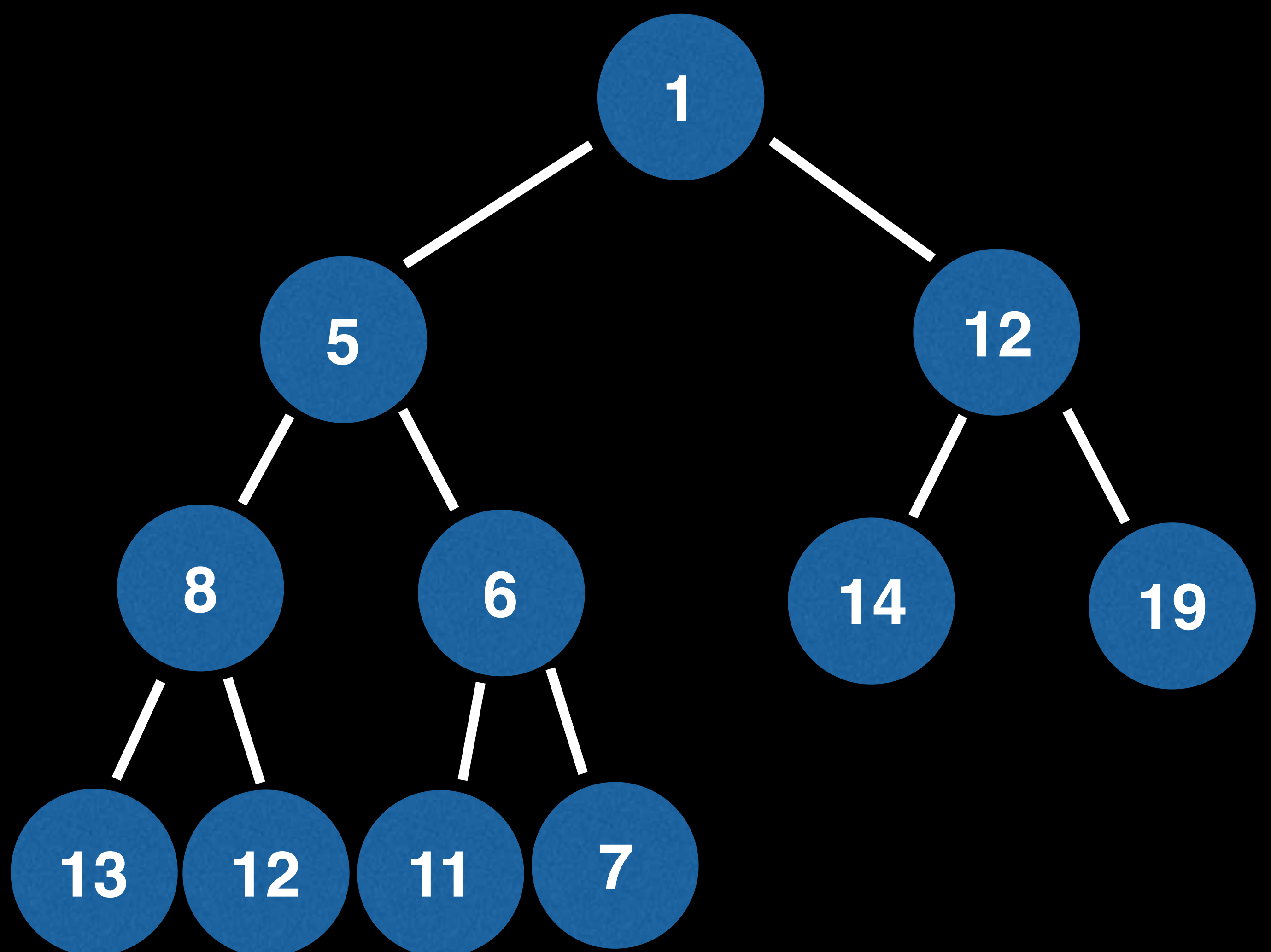
Insert(1)
Insert(13)
Insert(4)
Insert(0)
Insert(10)



Adding Elements to Binary Heap

Instructions:

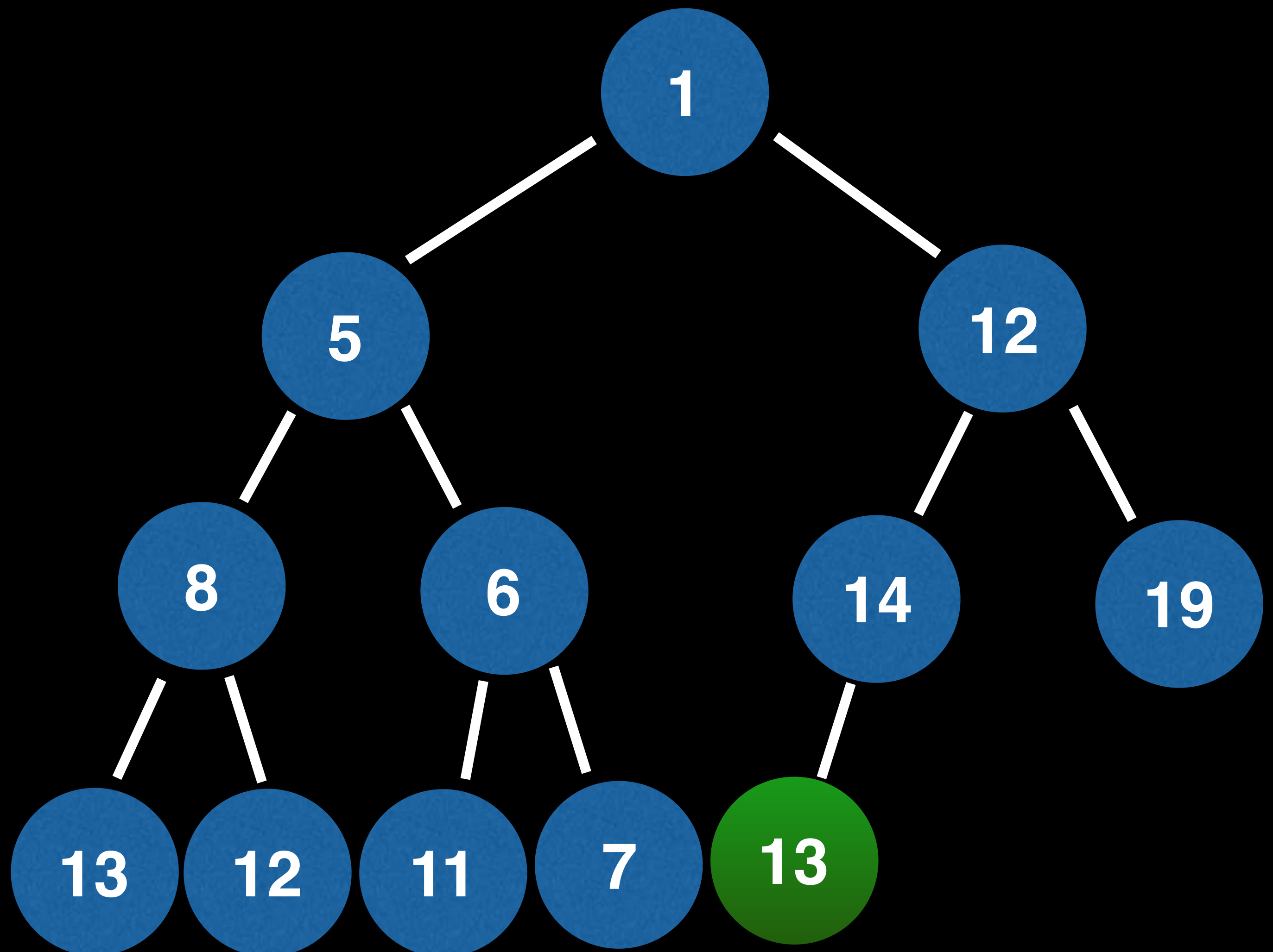
Insert(1)
Insert(13)
Insert(4)
Insert(0)
Insert(10)



Adding Elements to Binary Heap

Instructions:

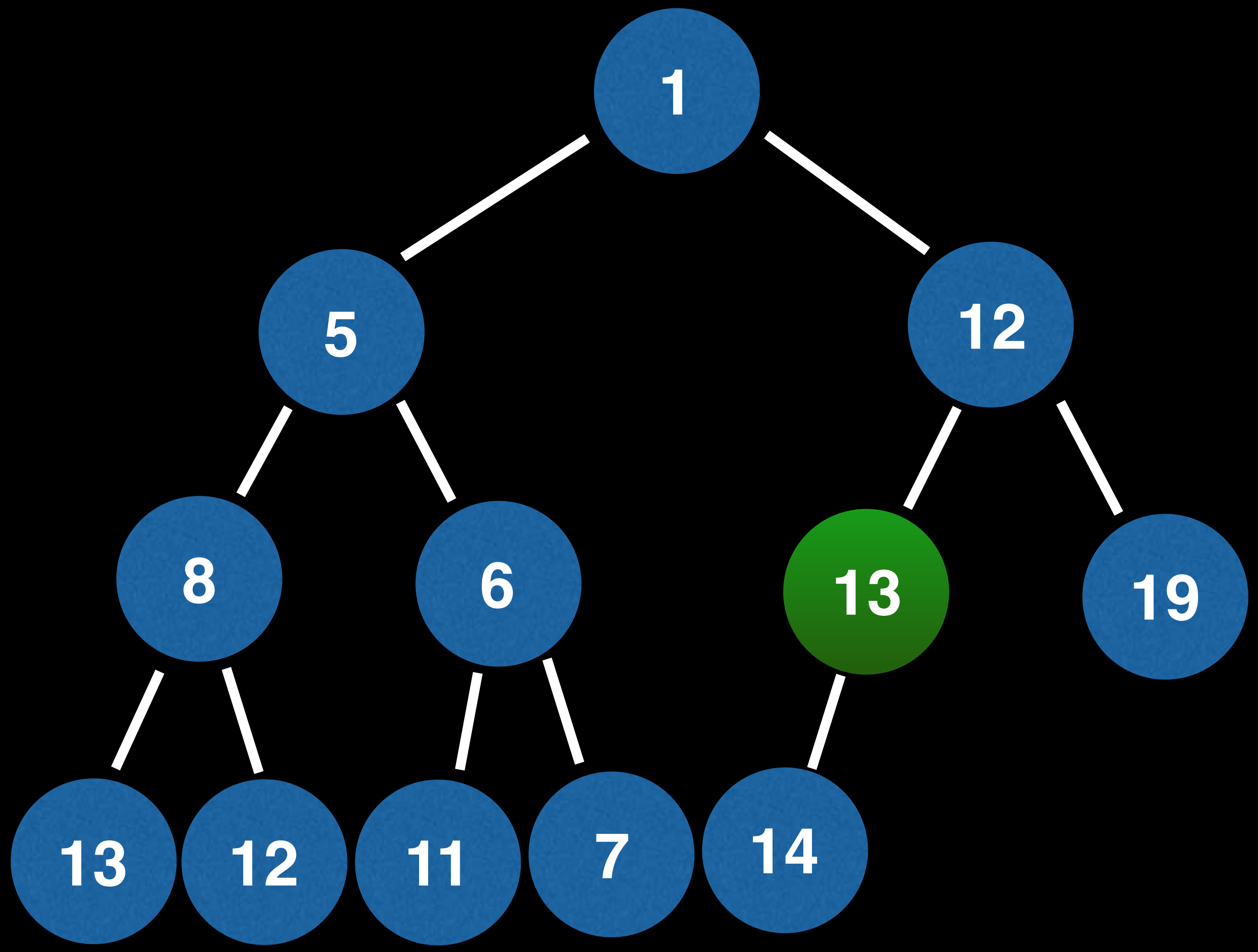
Insert(1)
Insert(13)
Insert(4)
Insert(0)
Insert(10)



Adding Elements to Binary Heap

Instructions:

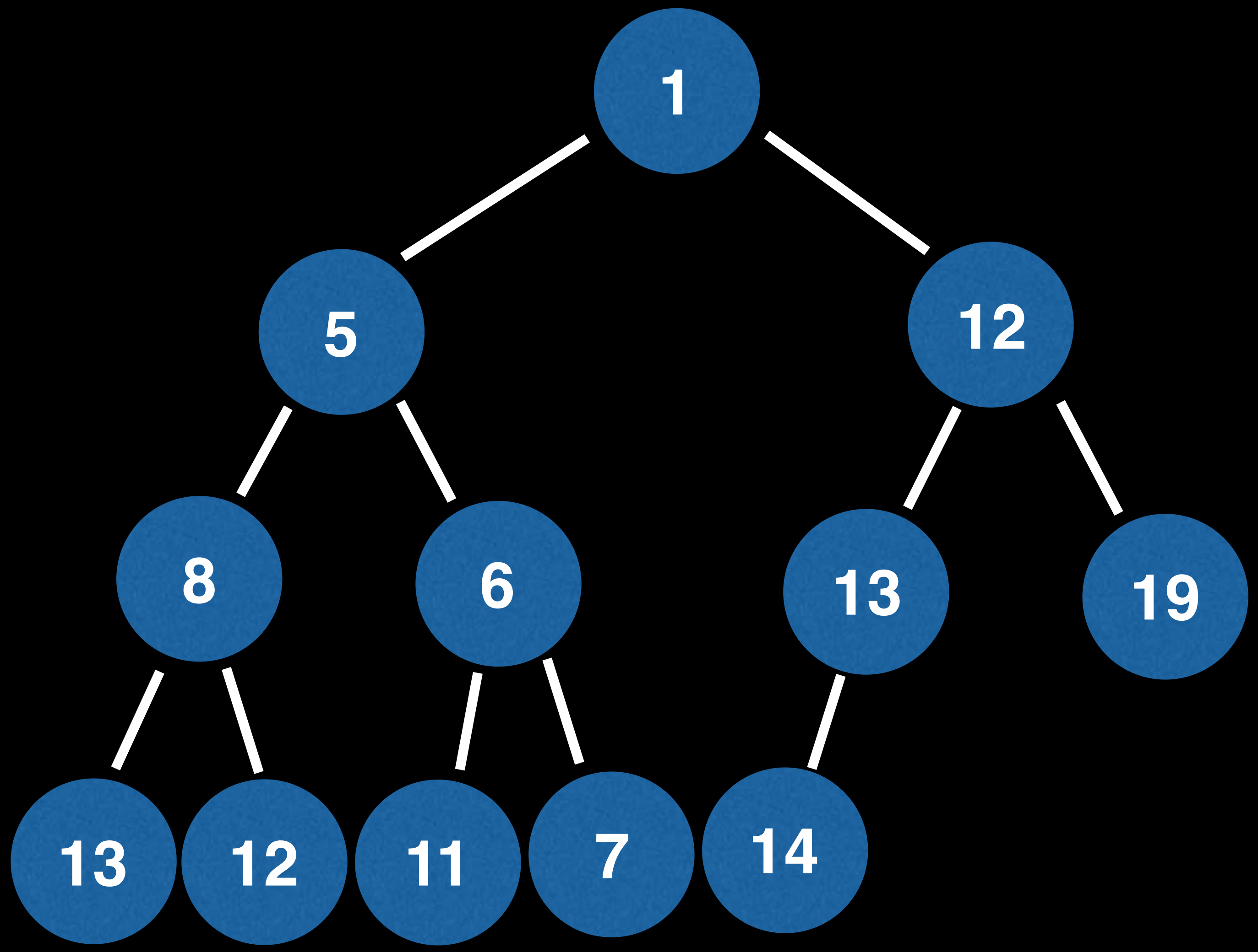
Insert(1)
Insert(13)
Insert(4)
Insert(0)
Insert(10)



Adding Elements to Binary Heap

Instructions:

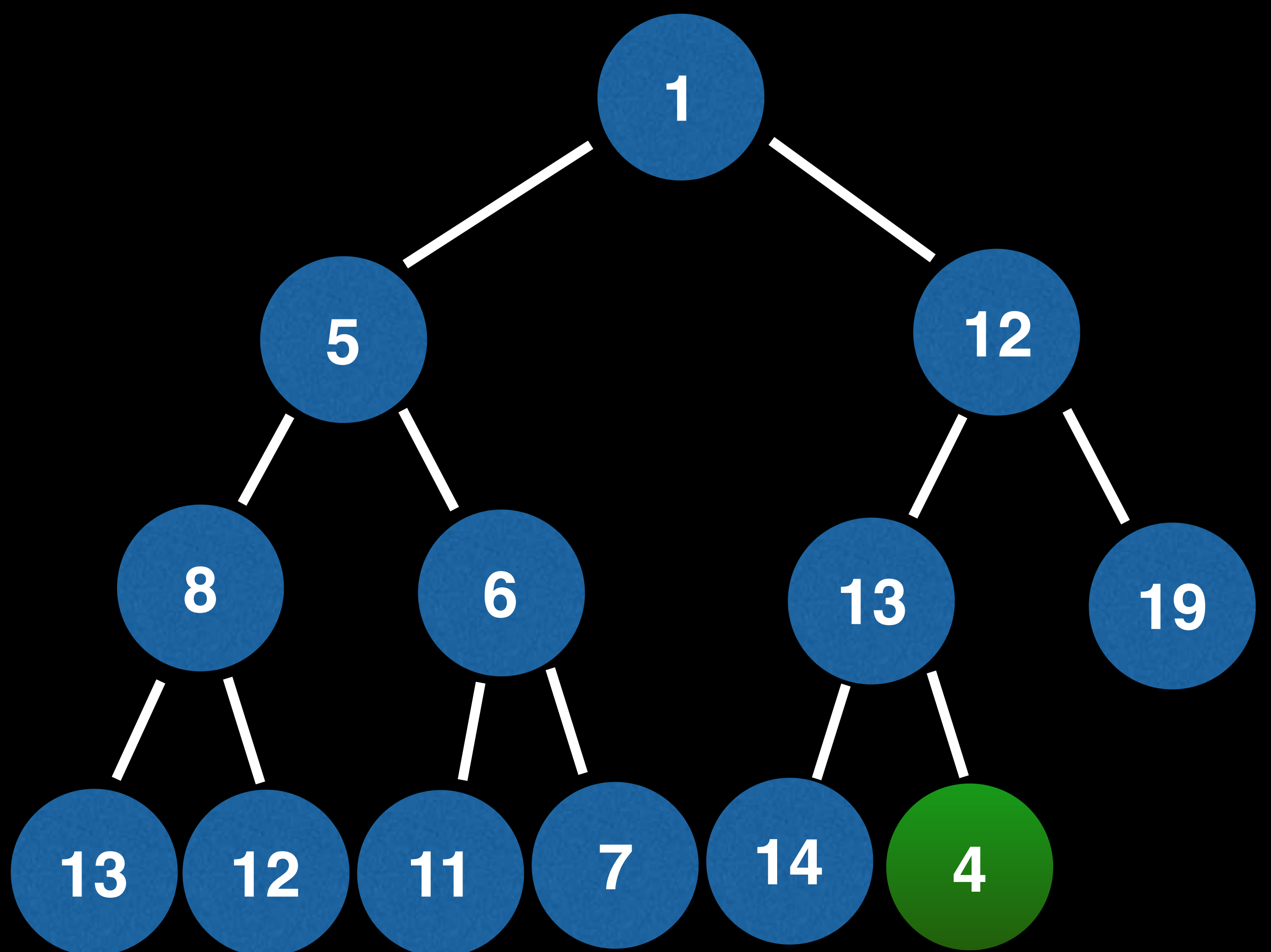
Insert(1)
Insert(13)
Insert(4)
Insert(0)
Insert(10)



Adding Elements to Binary Heap

Instructions:

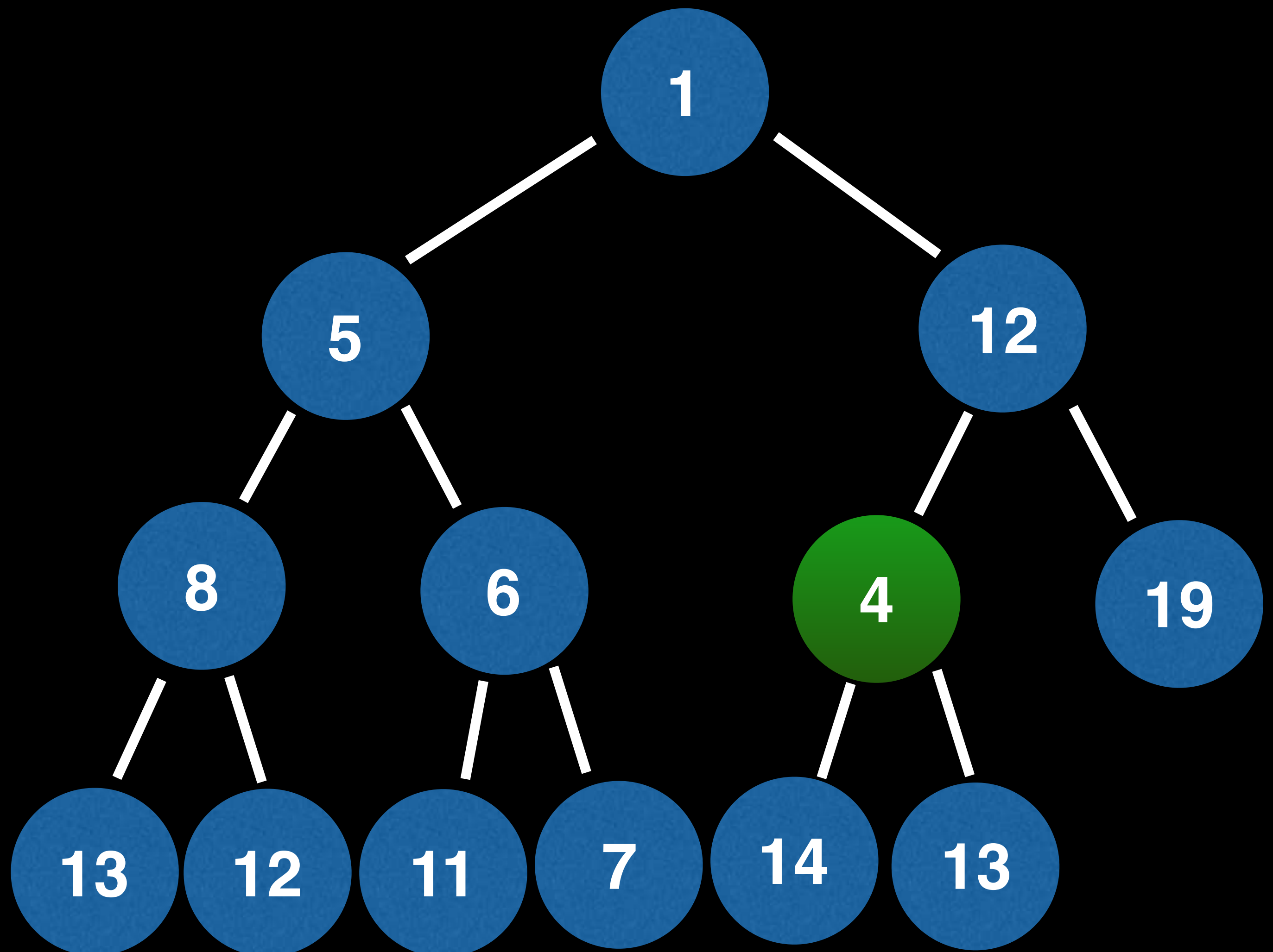
Insert(1)
Insert(13)
Insert(4)
Insert(0)
Insert(10)



Adding Elements to Binary Heap

Instructions:

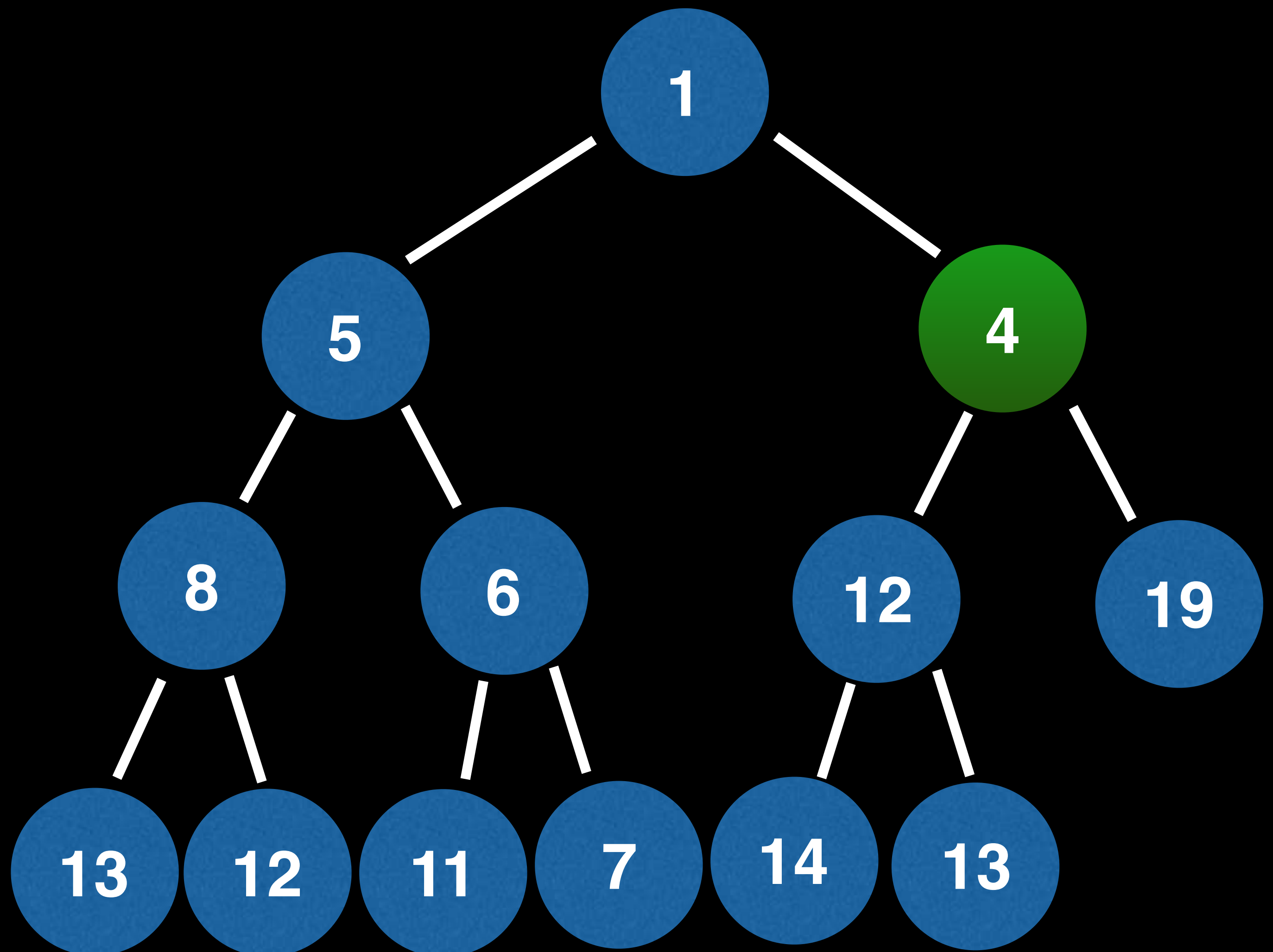
Insert(1)
Insert(13)
Insert(4)
Insert(0)
Insert(10)



Adding Elements to Binary Heap

Instructions:

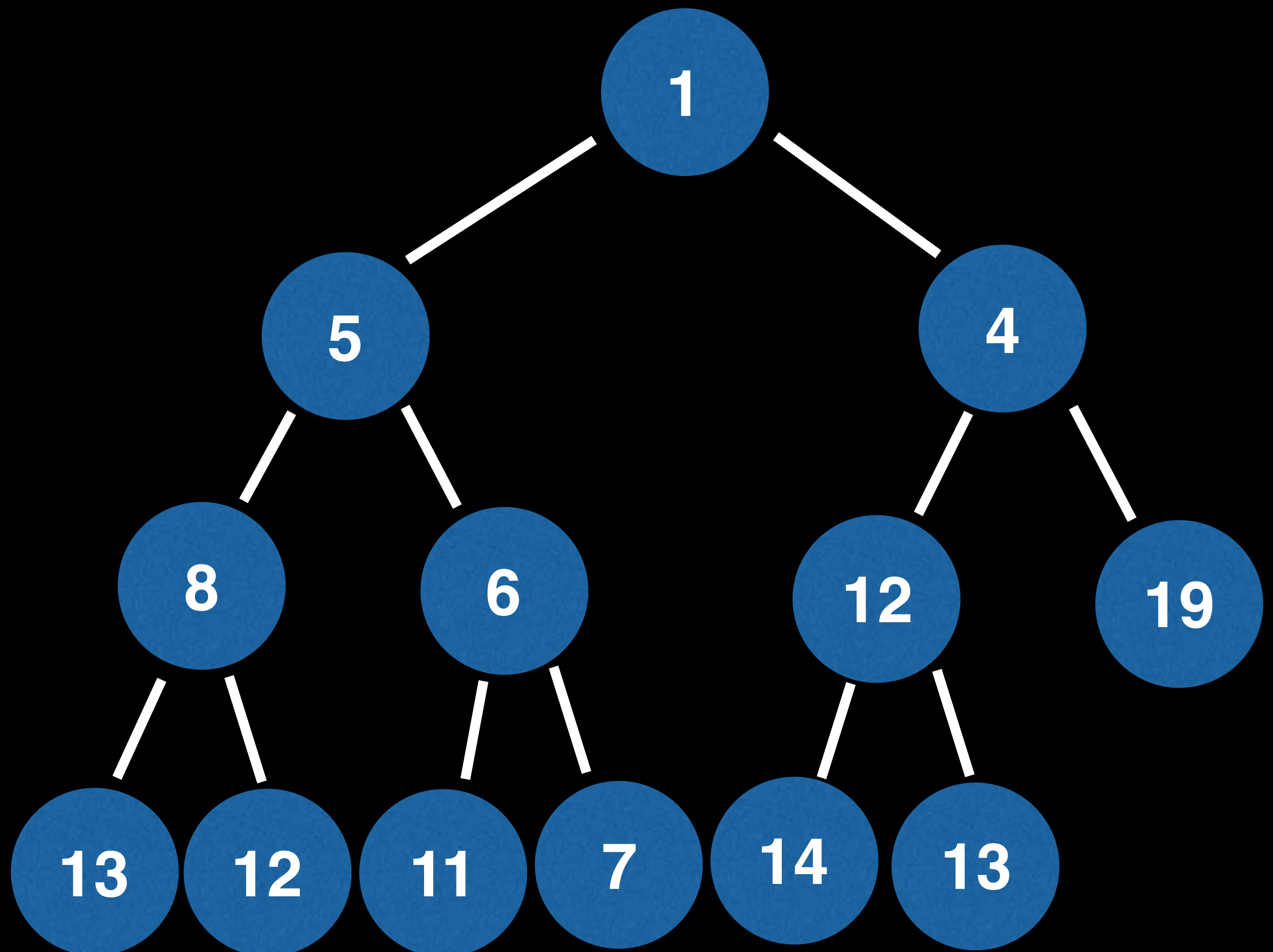
Insert(1)
Insert(13)
Insert(4)
Insert(0)
Insert(10)



Adding Elements to Binary Heap

Instructions:

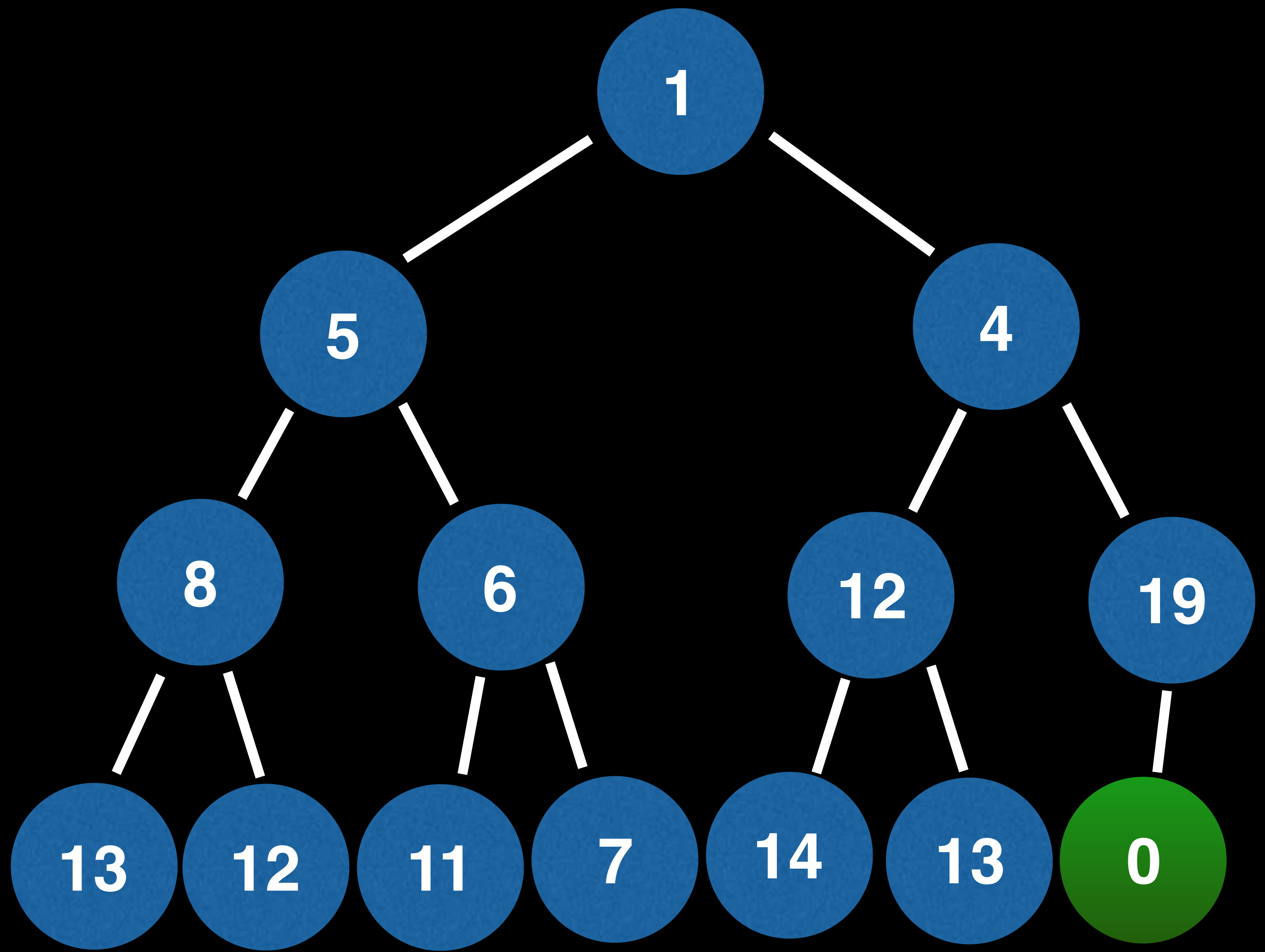
Insert(1)
Insert(13)
Insert(4)
Insert(0)
Insert(10)



Adding Elements to Binary Heap

Instructions:

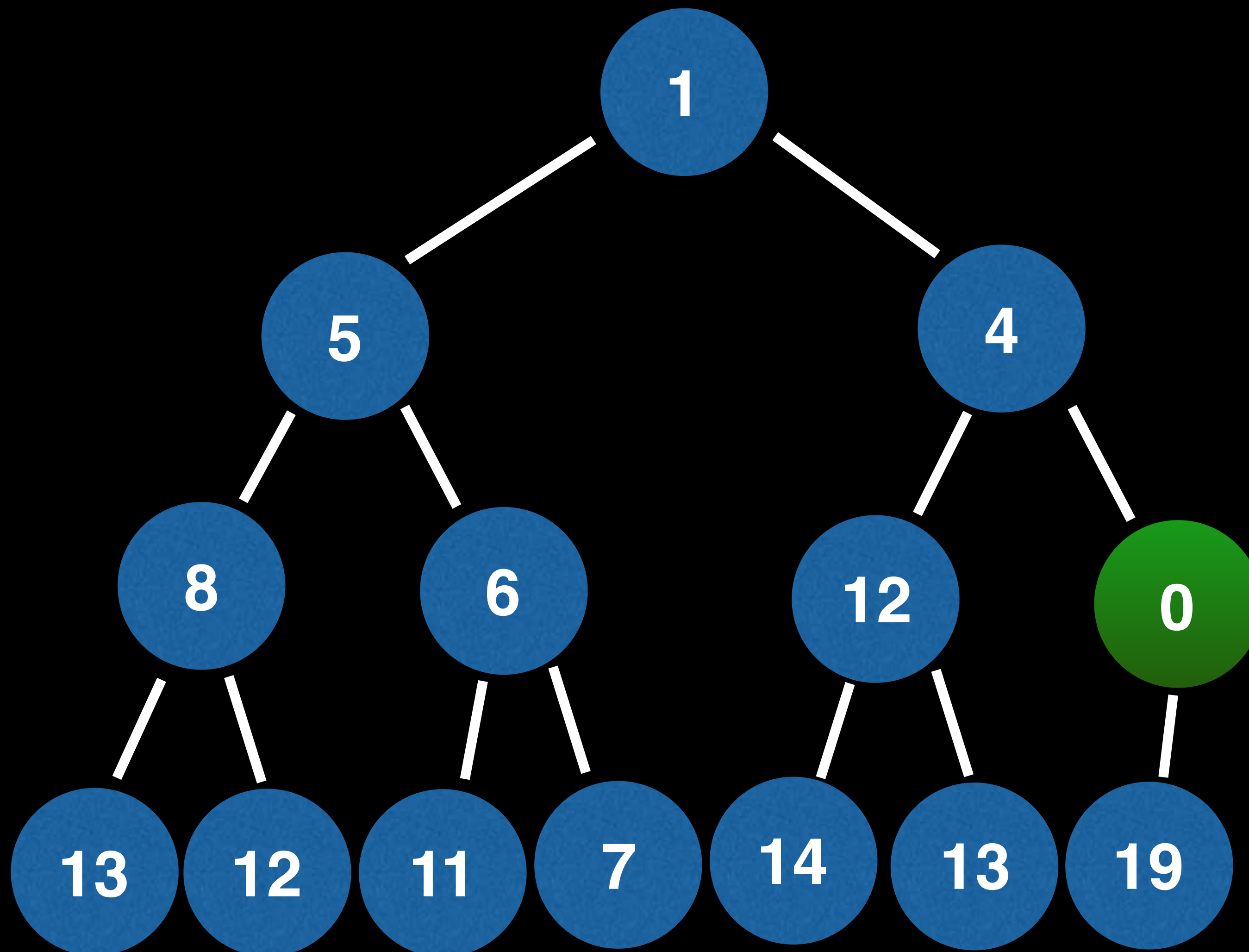
Insert(1)
Insert(13)
Insert(4)
Insert(0)
Insert(10)



Adding Elements to Binary Heap

Instructions:

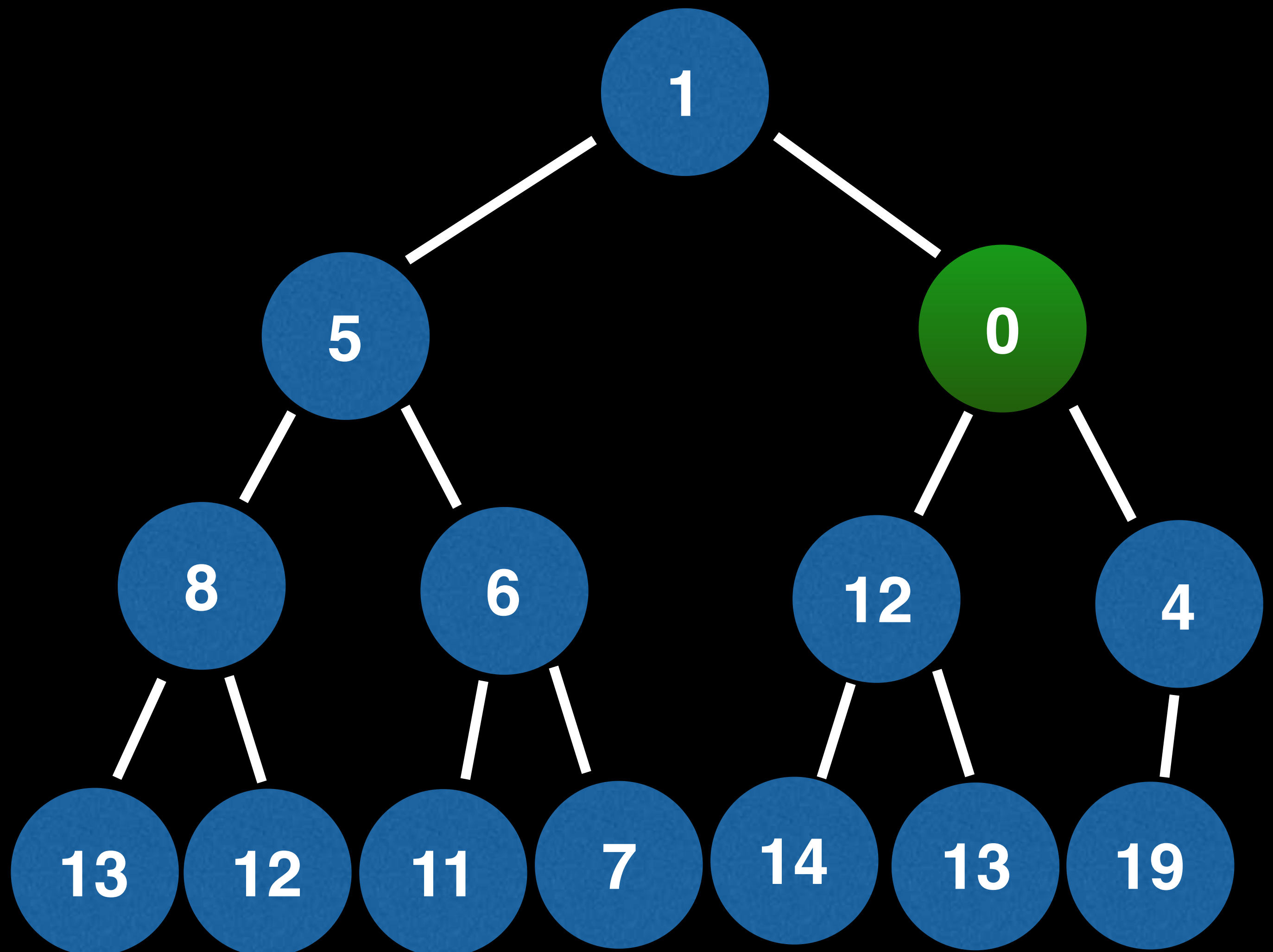
Insert(1)
Insert(13)
Insert(4)
Insert(0)
Insert(10)



Adding Elements to Binary Heap

Instructions:

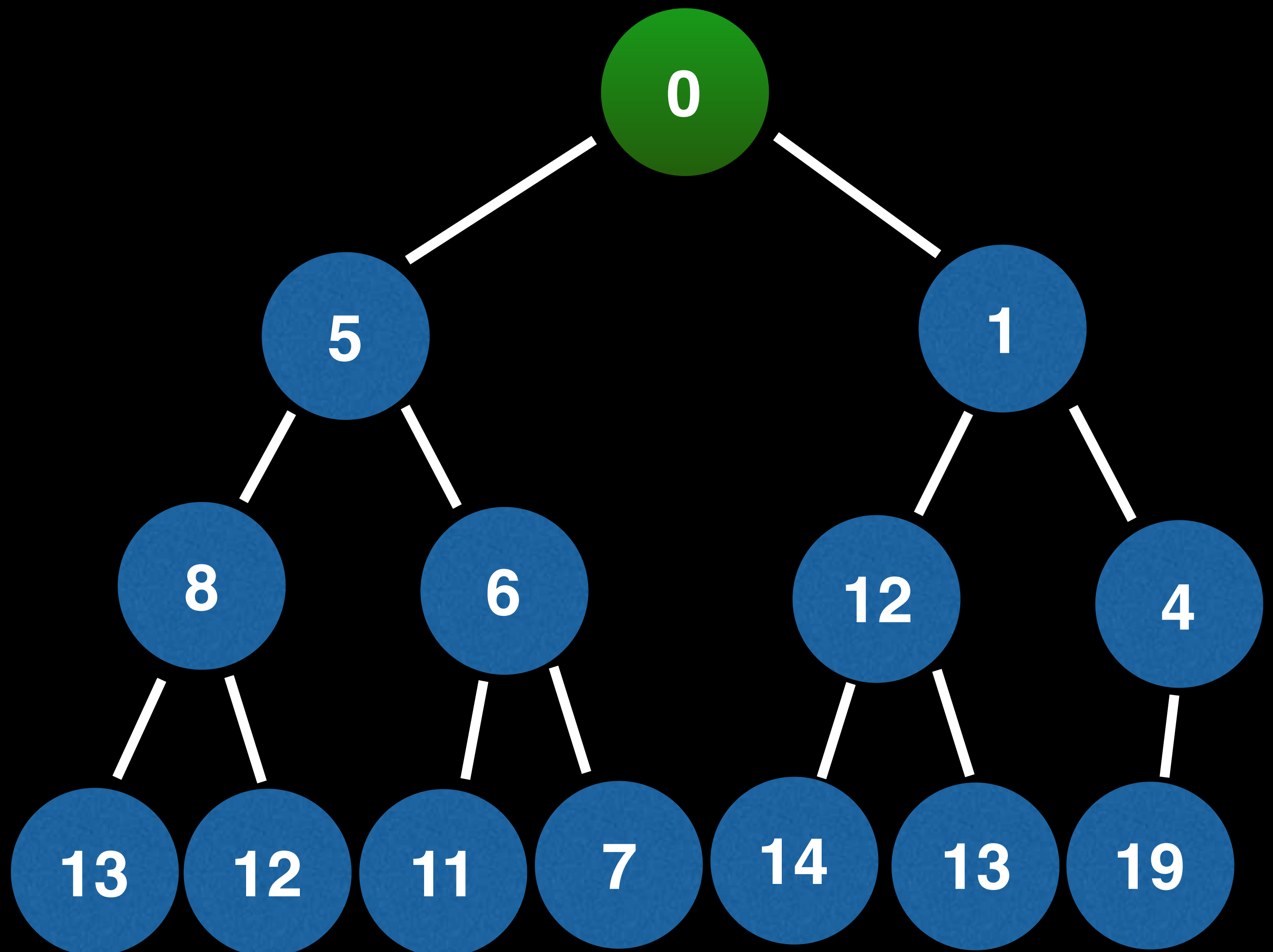
Insert(1)
Insert(13)
Insert(4)
Insert(0)
Insert(10)



Adding Elements to Binary Heap

Instructions:

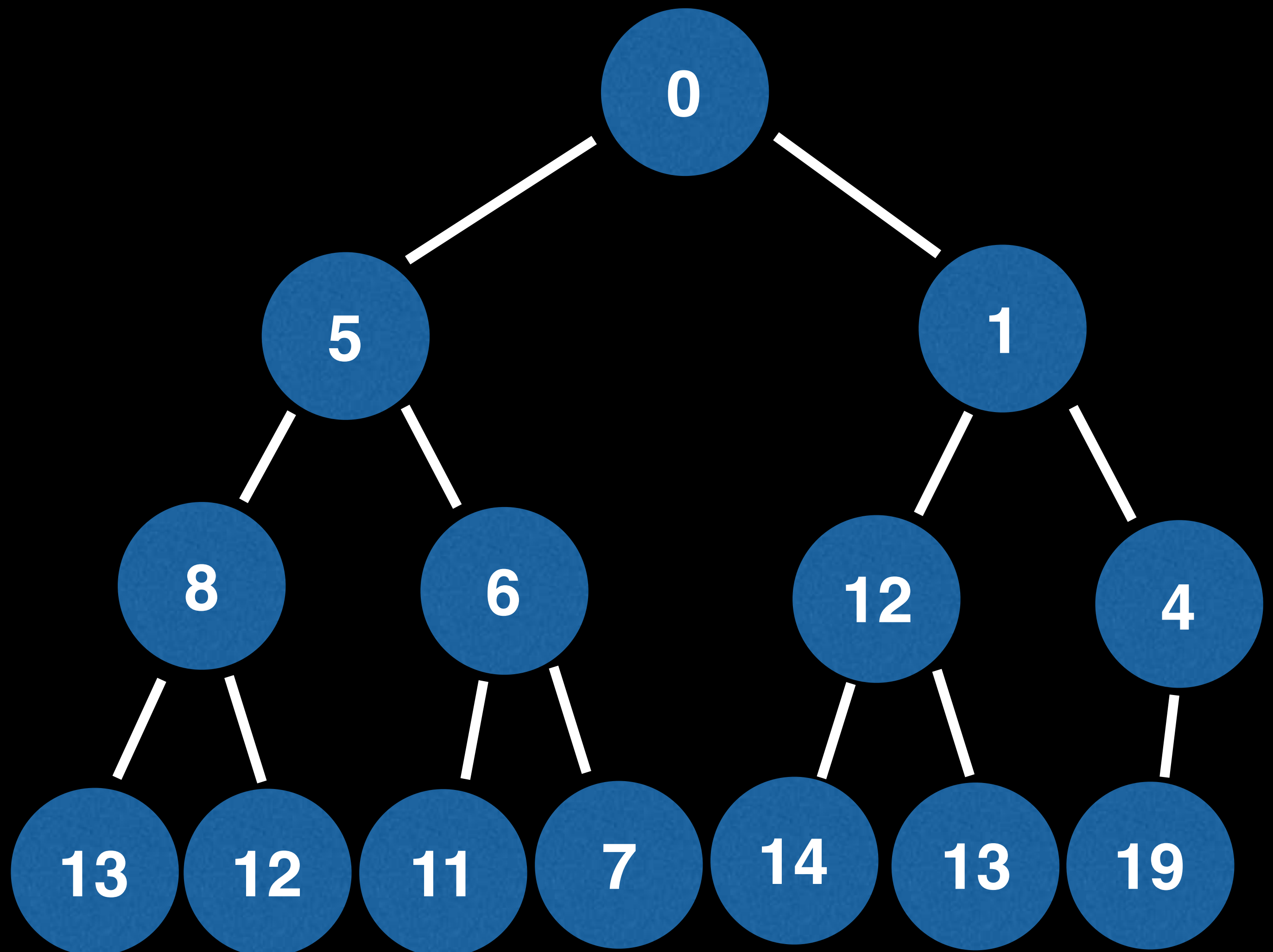
Insert(1)
Insert(13)
Insert(4)
Insert(0)
Insert(10)



Adding Elements to Binary Heap

Instructions:

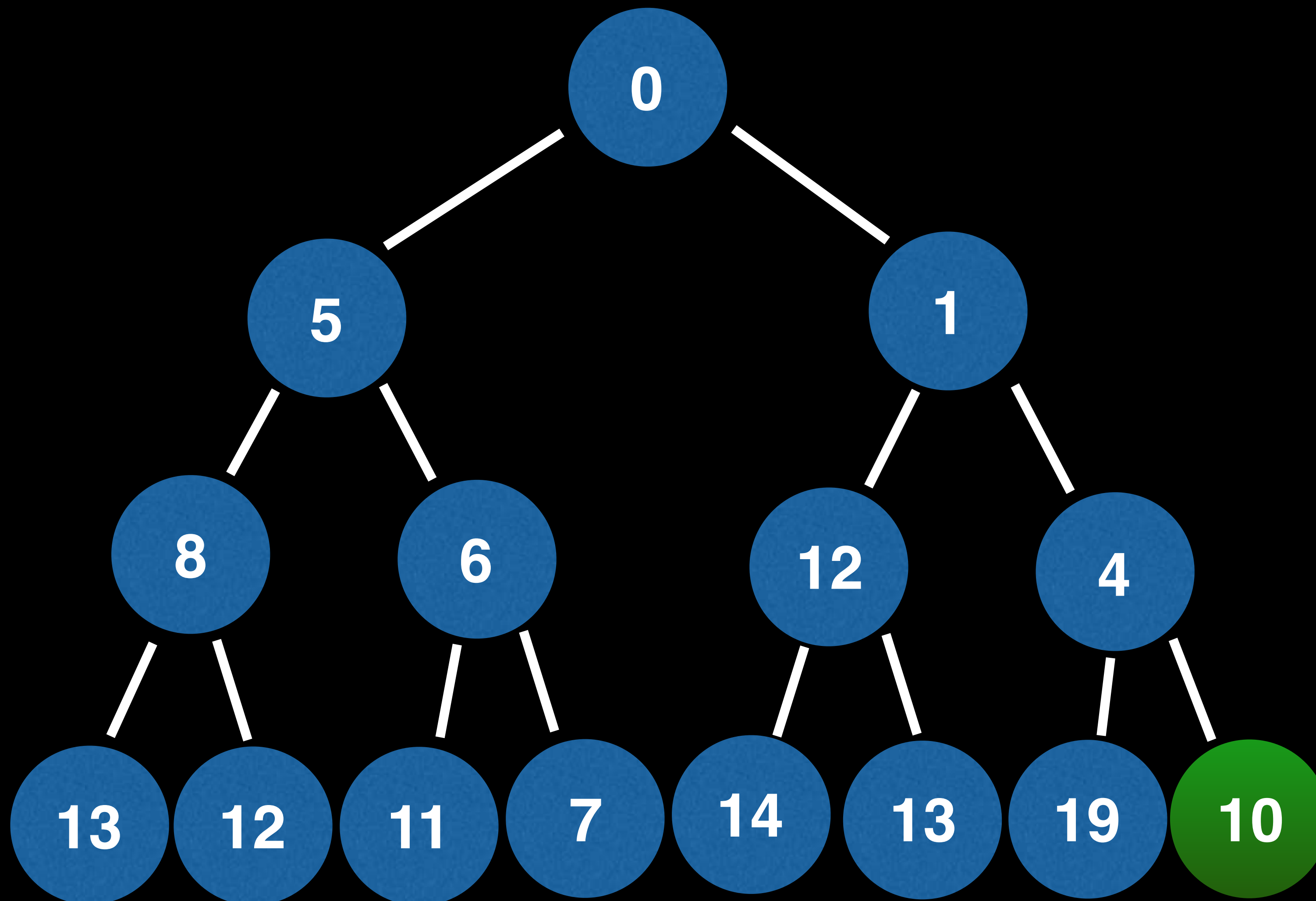
Insert(1)
Insert(13)
Insert(4)
Insert(0)
Insert(10)



Adding Elements to Binary Heap

Instructions:

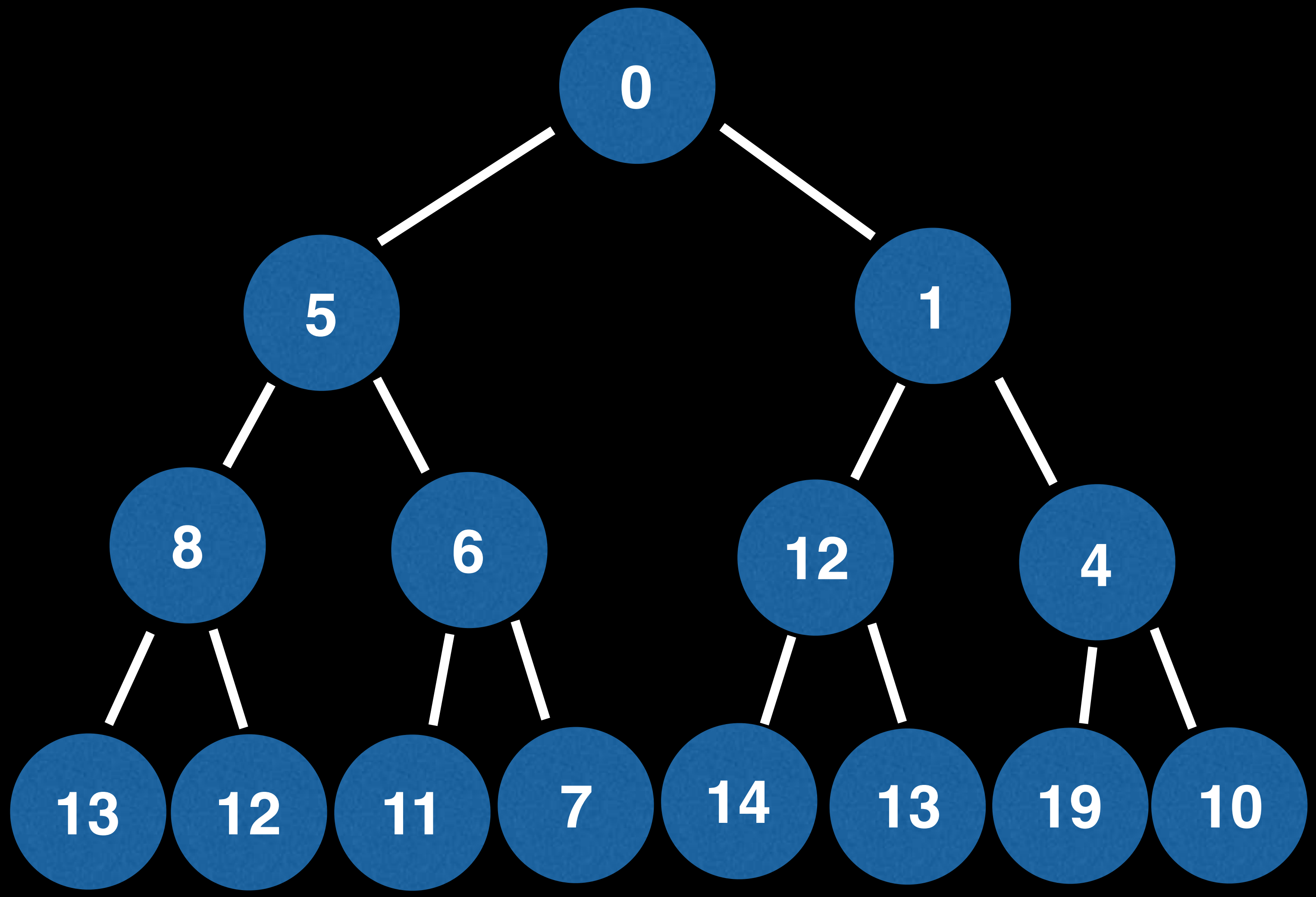
Insert(1)
Insert(13)
Insert(4)
Insert(0)
Insert(10)



Adding Elements to Binary Heap

Instructions:

Insert(1)
Insert(13)
Insert(4)
Insert(0)
Insert(10)

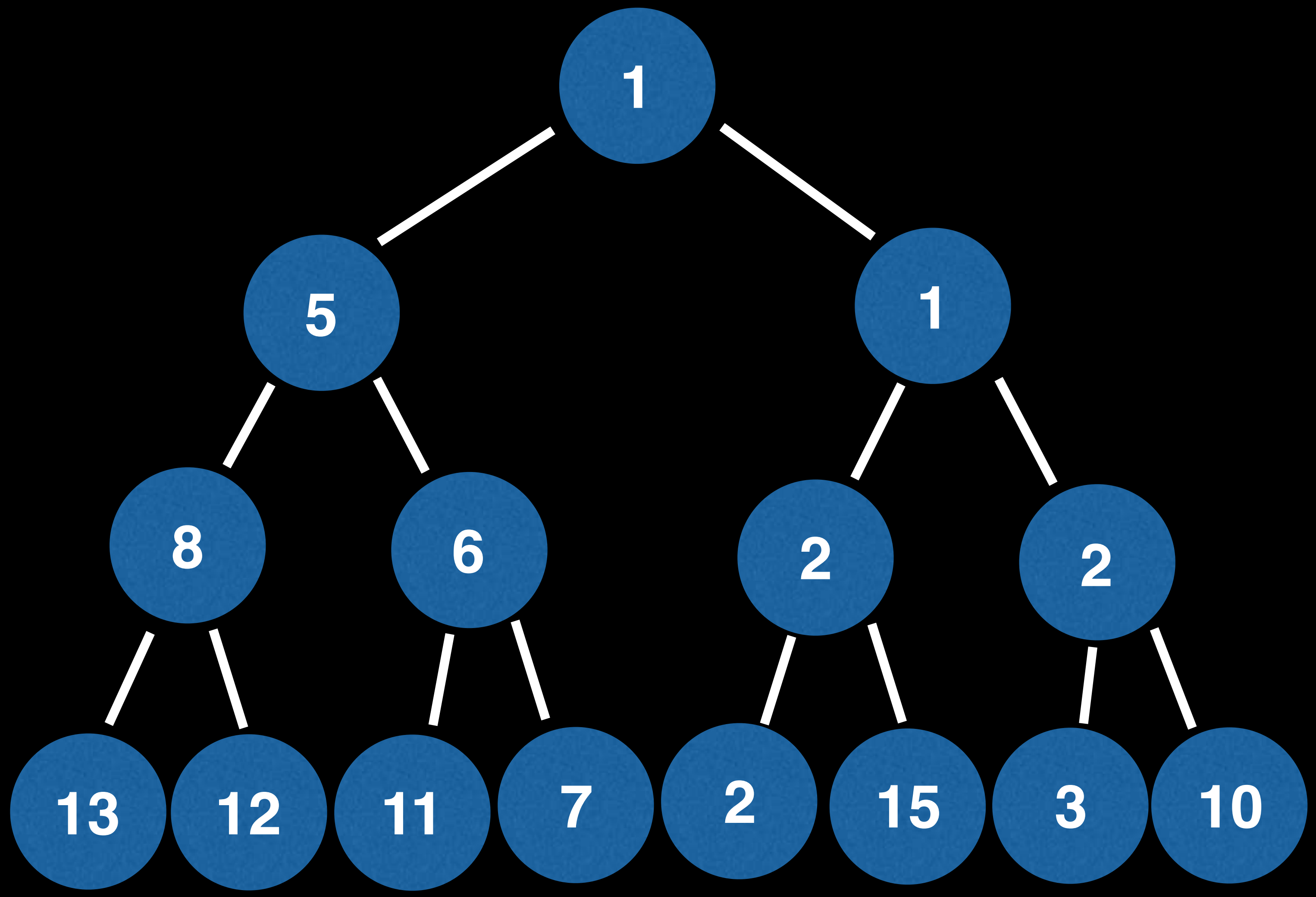


Removing Elements from Binary Heap

Removing Elements From a Binary Heap

Instructions:

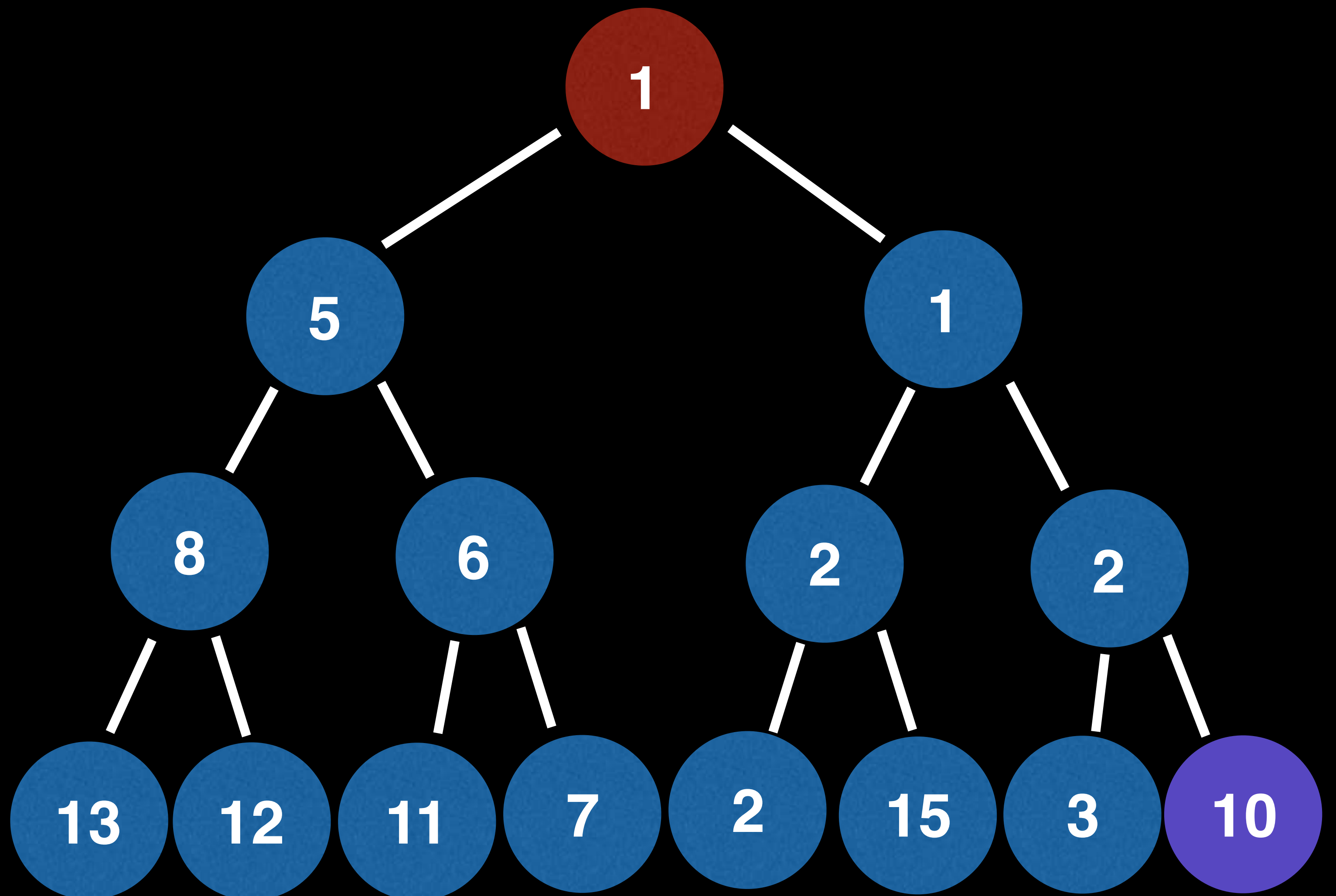
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

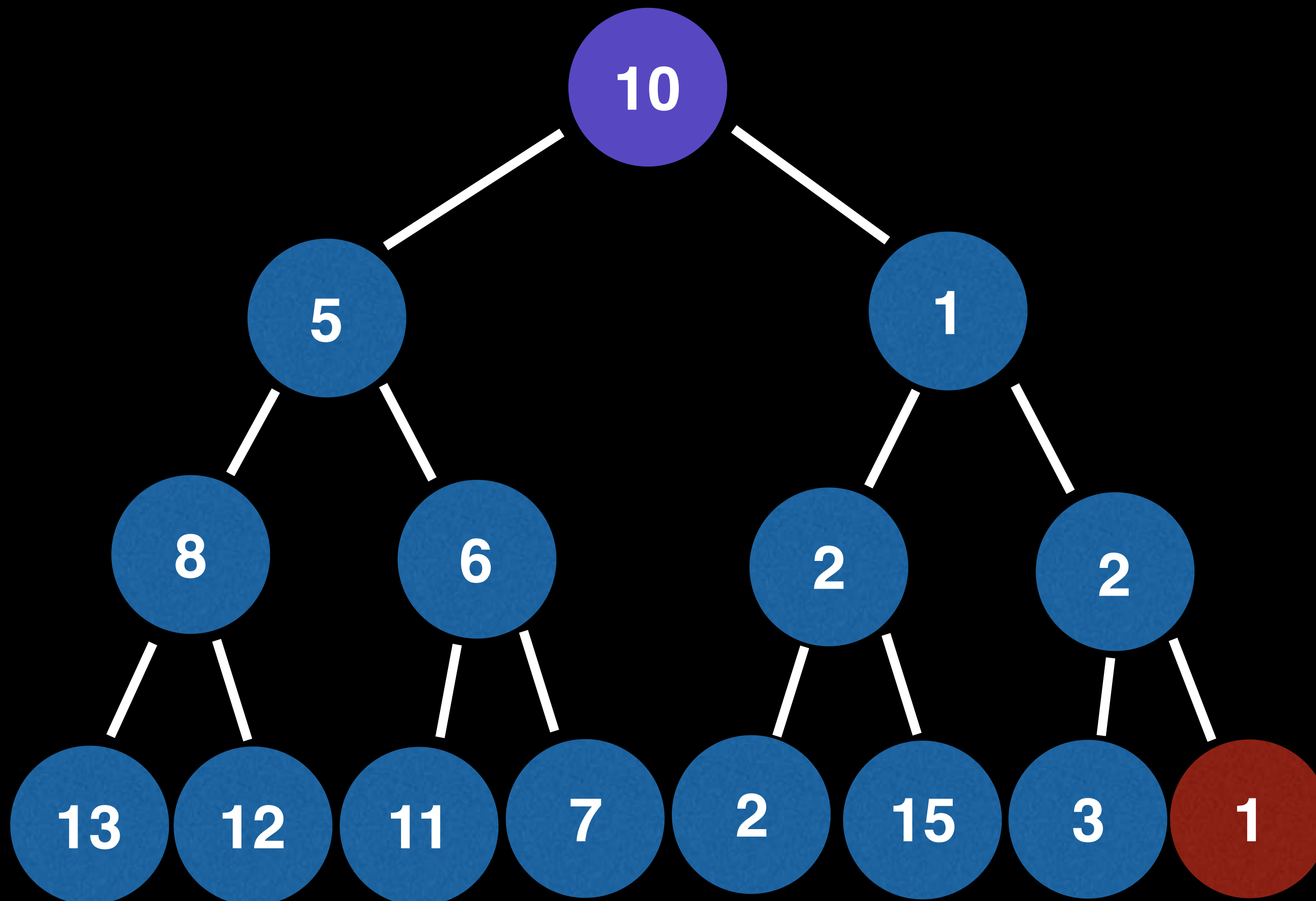
- Poll()
- Remove(12)
- Remove(3)
- Poll()
- Remove(6)



Removing Elements From a Binary Heap

Instructions:

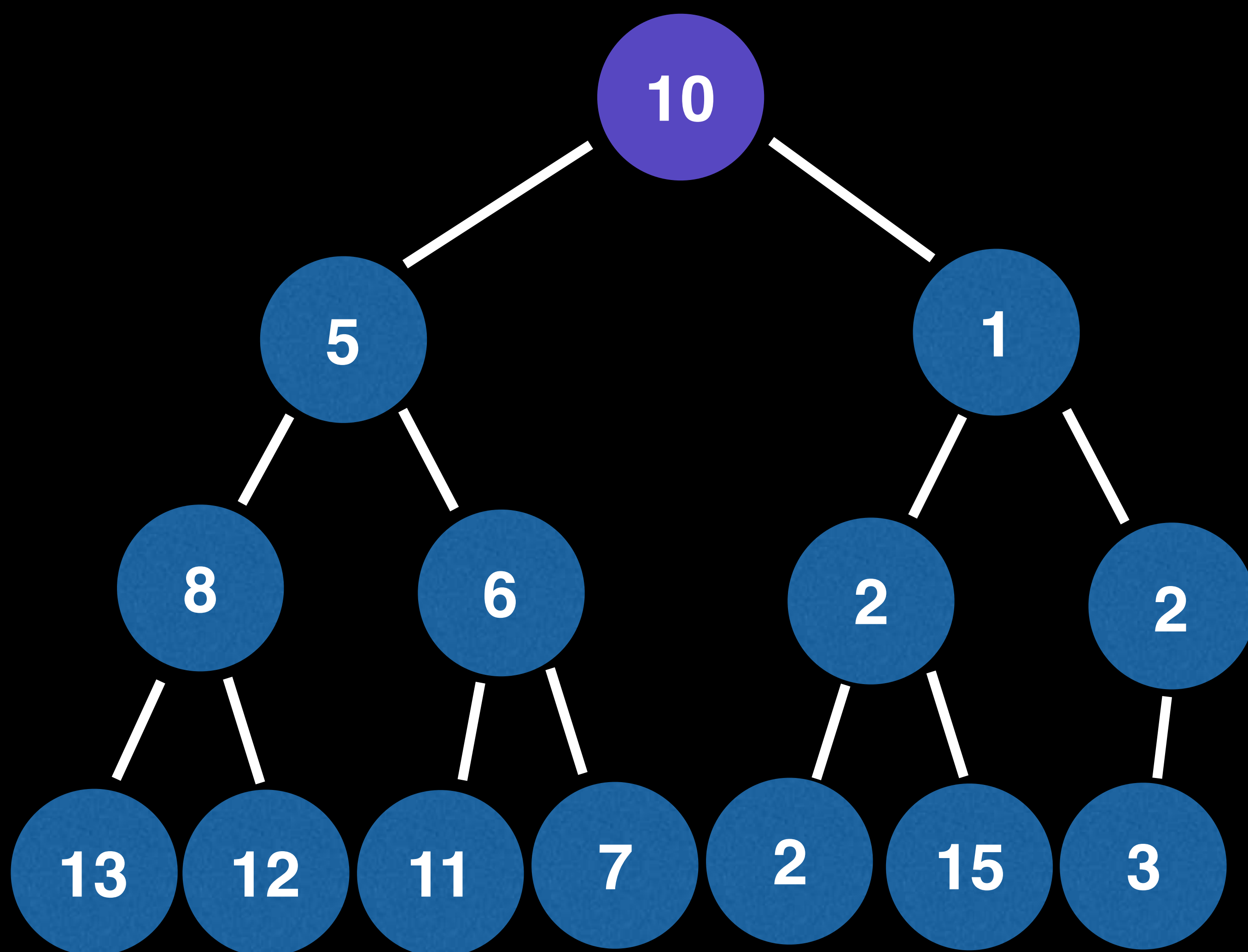
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

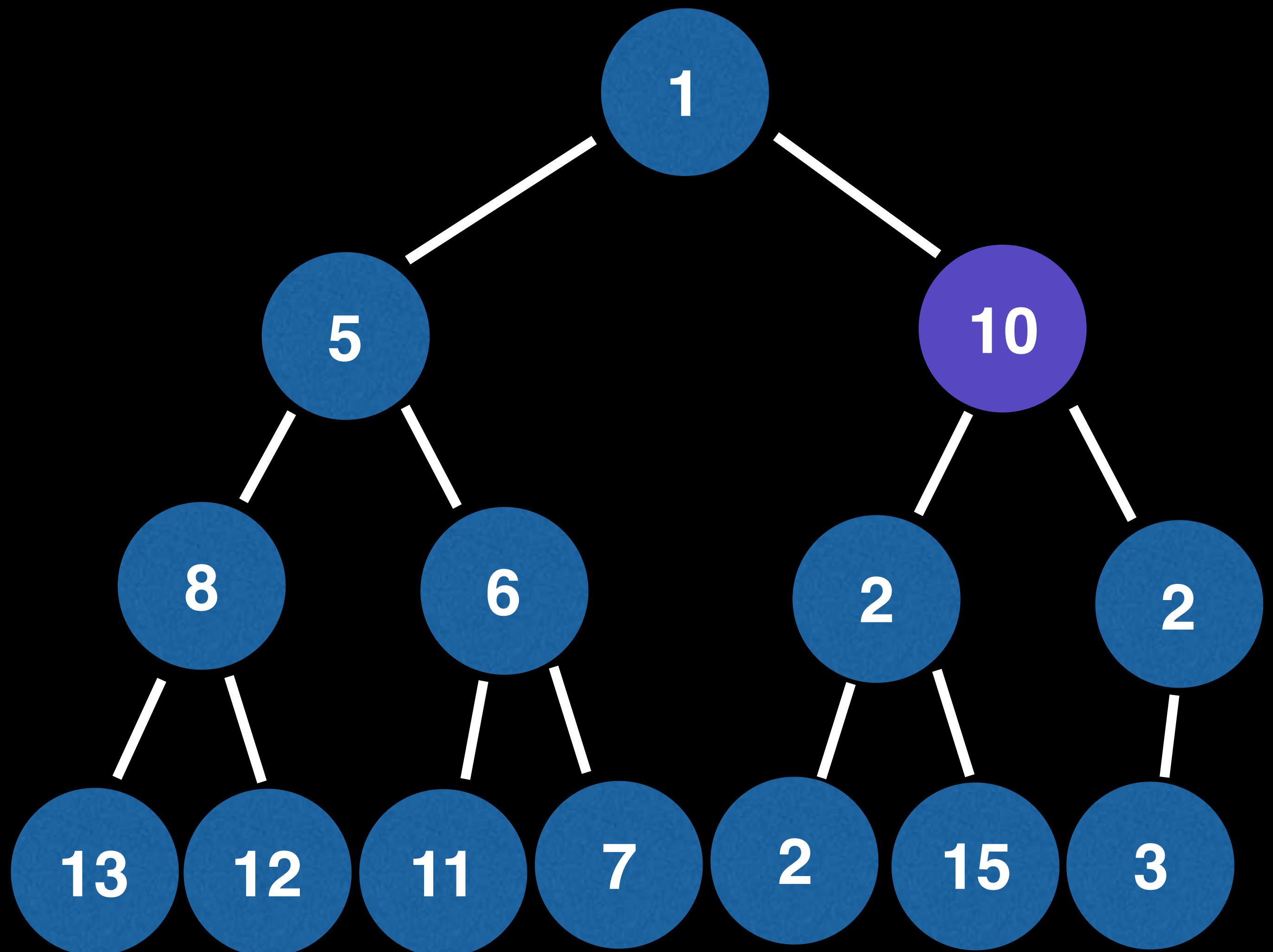
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

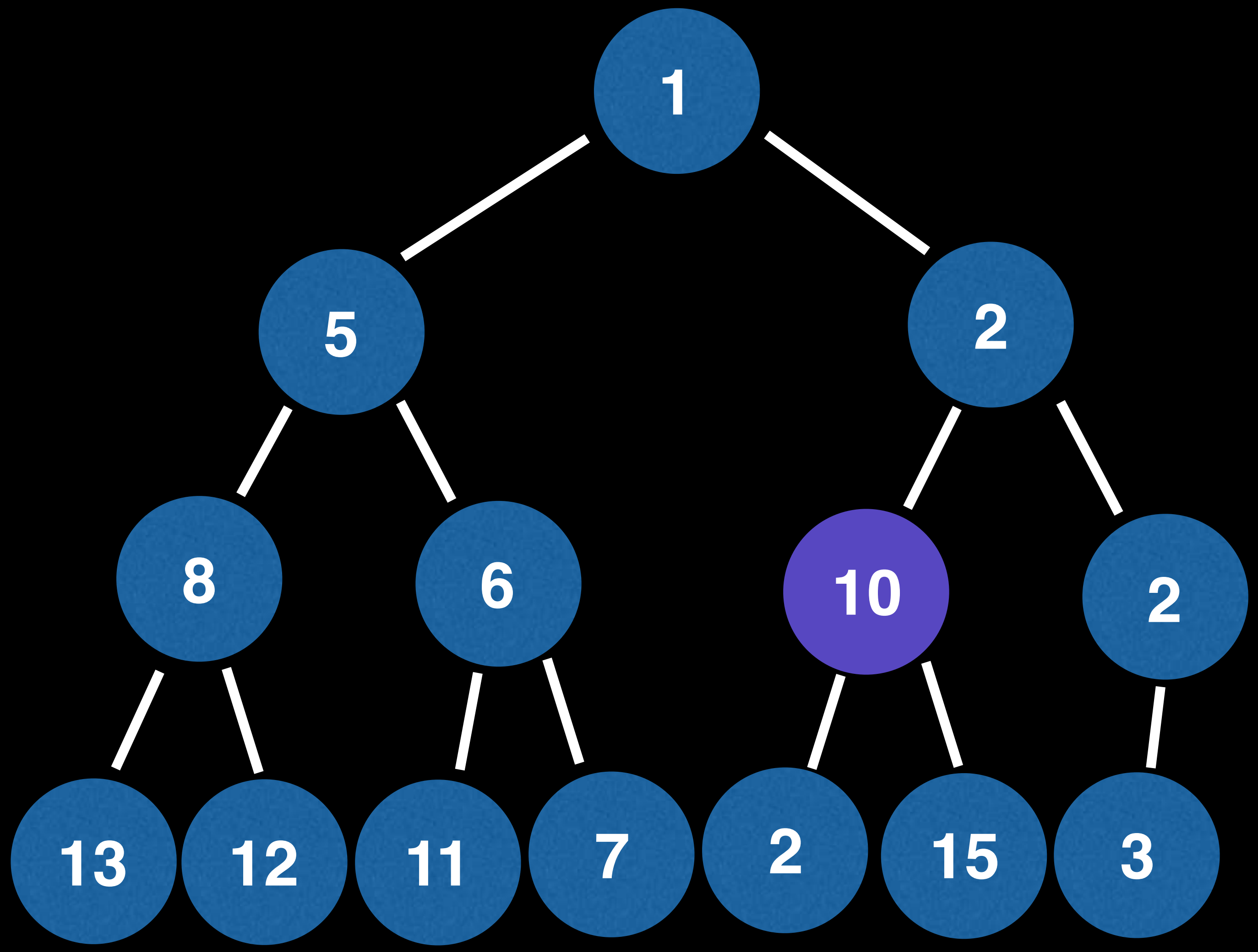
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

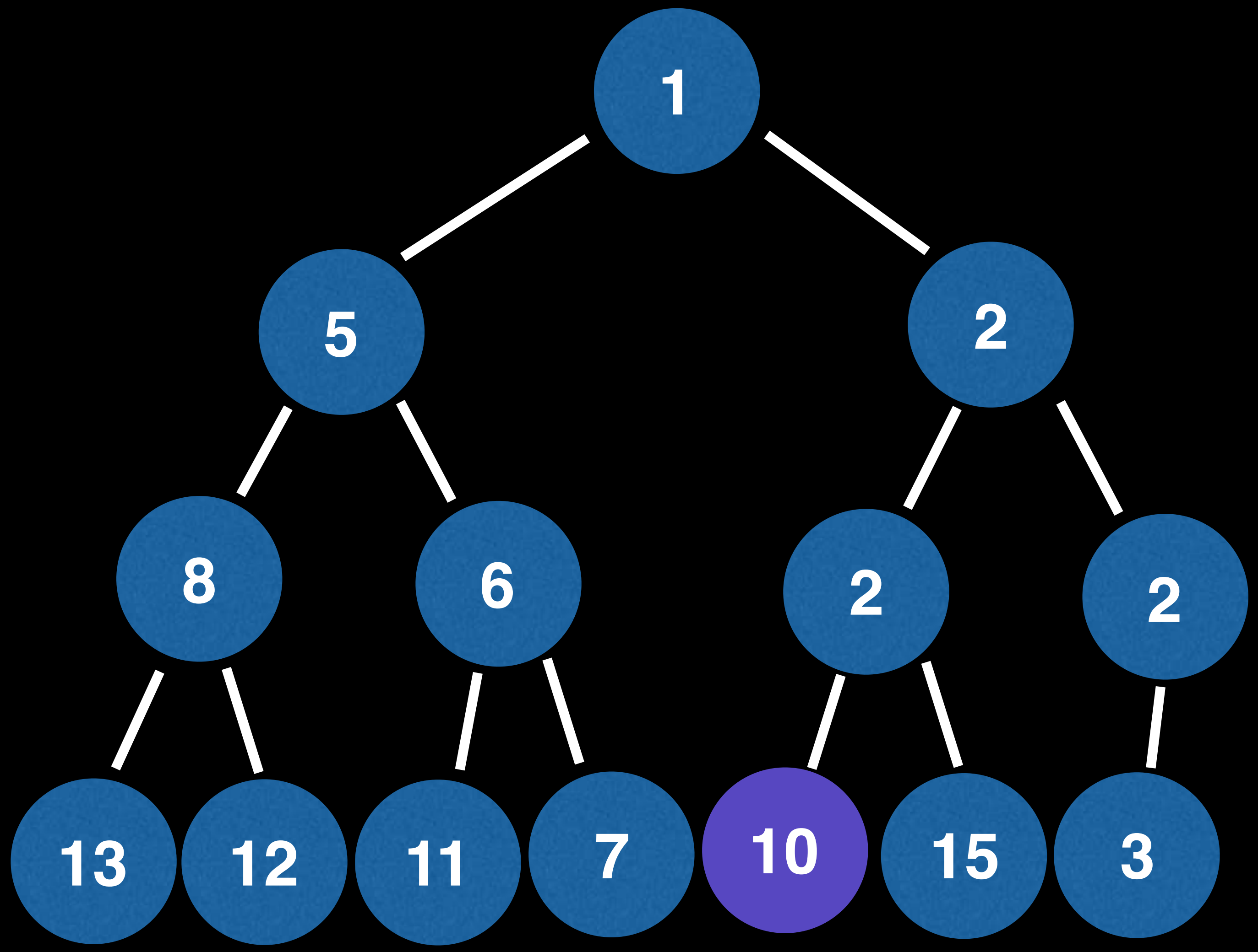
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

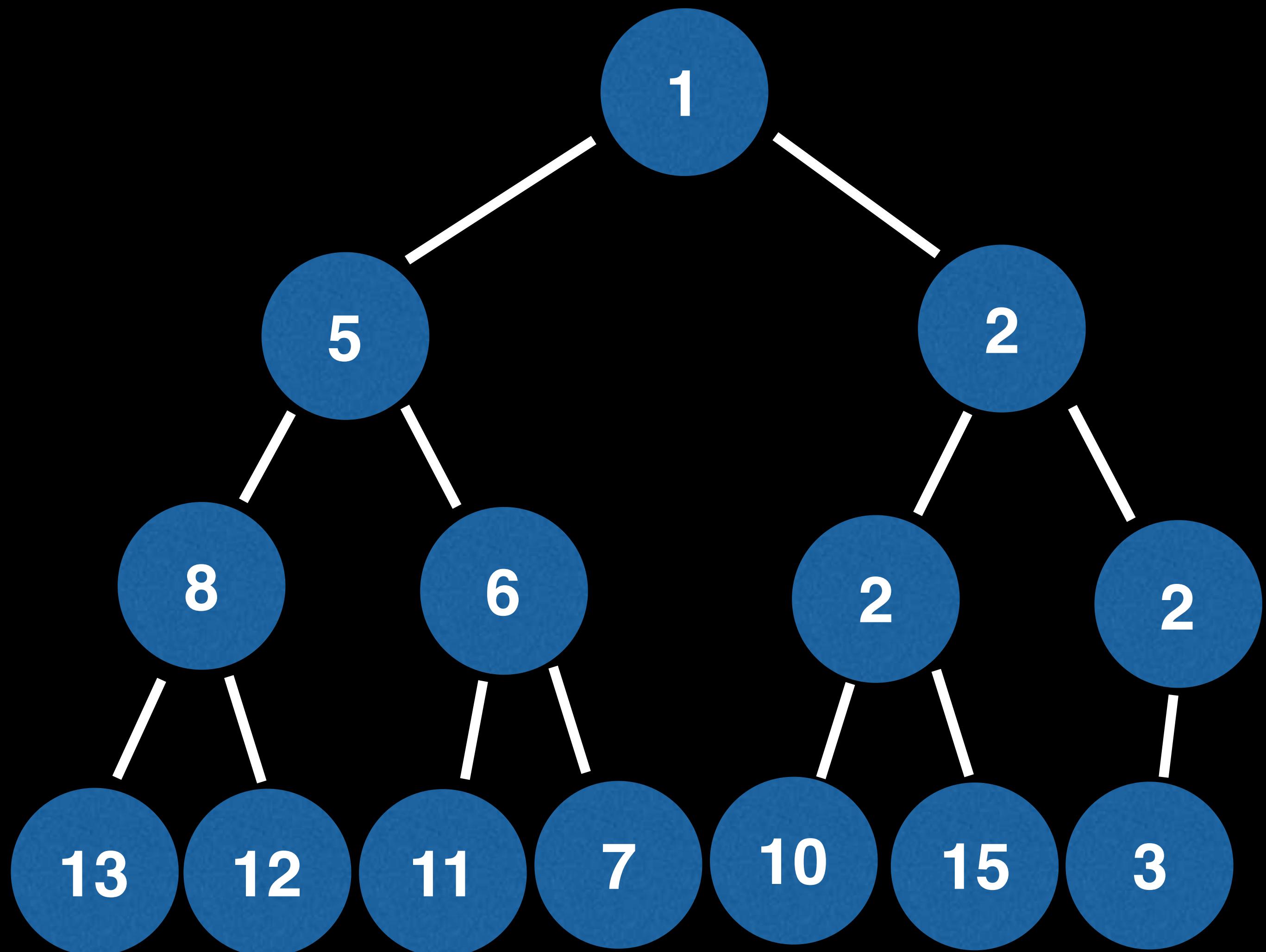
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

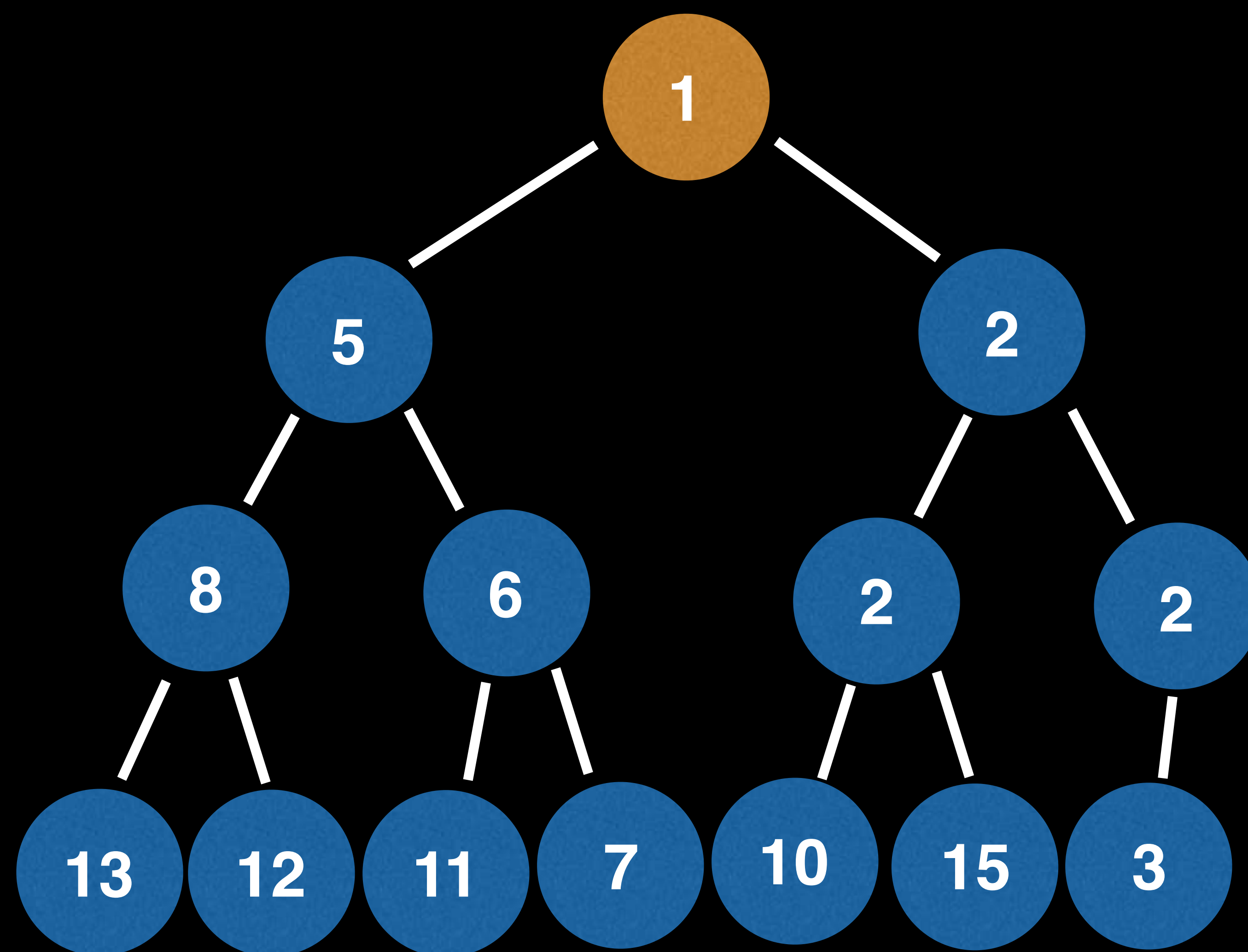
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

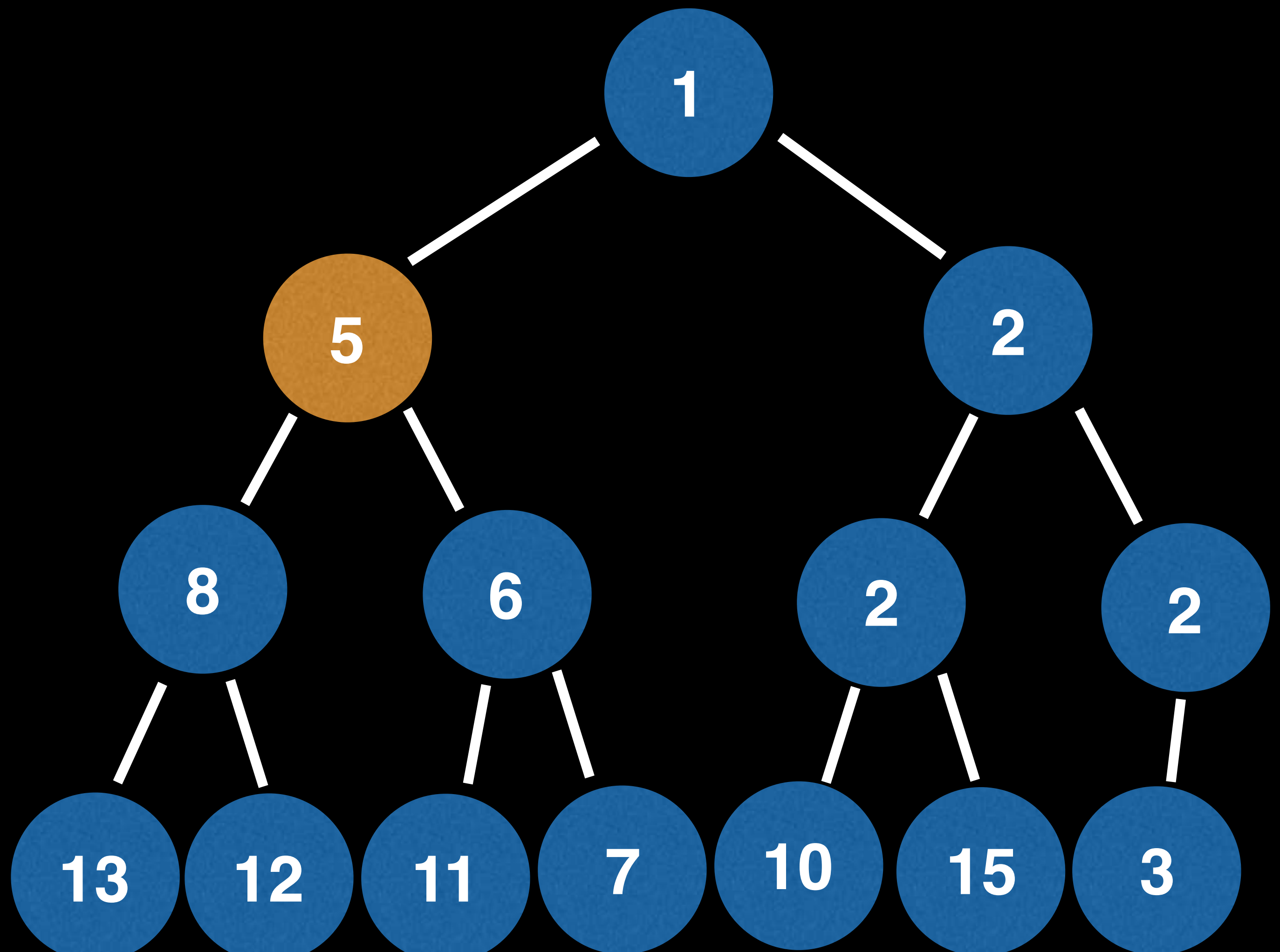
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

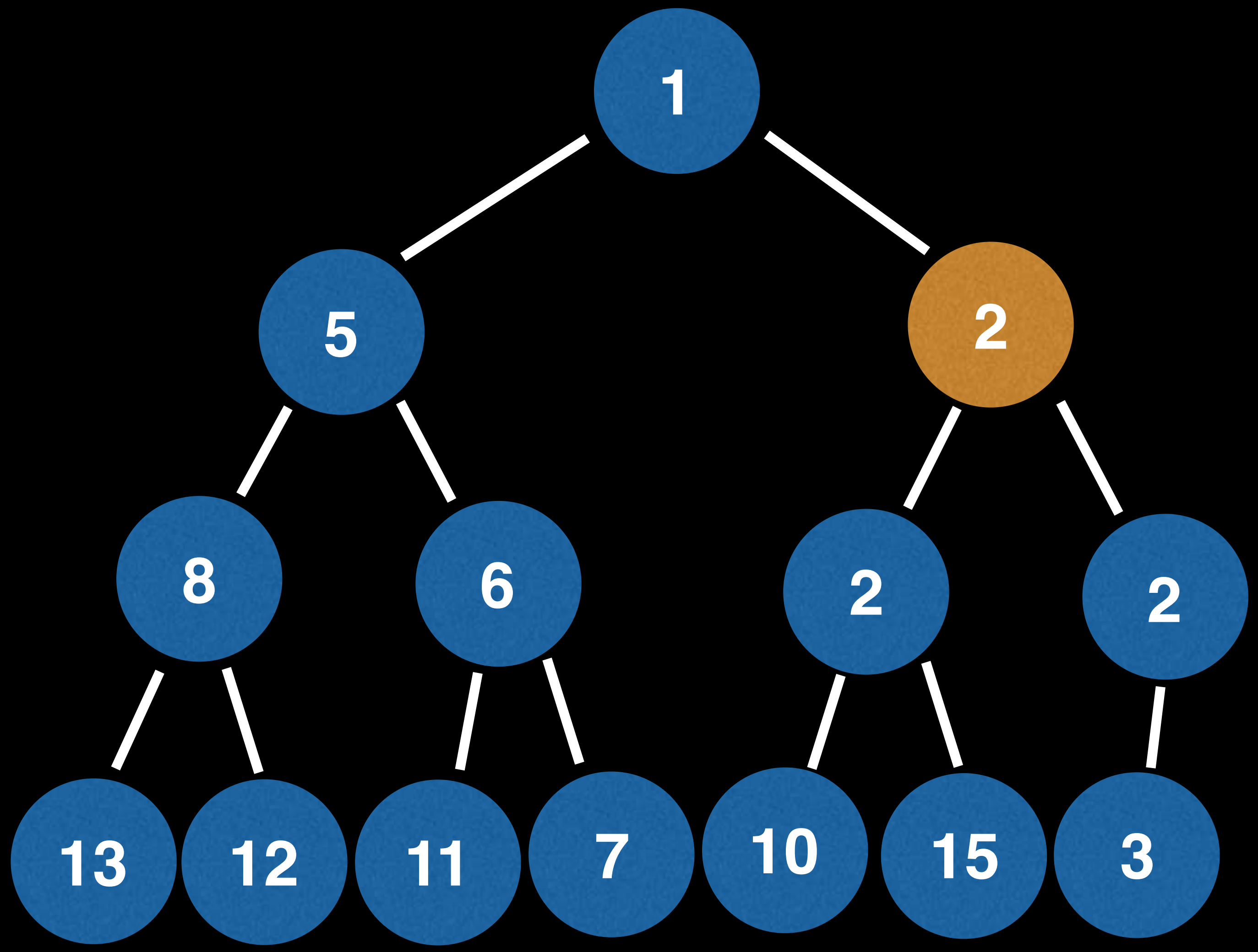
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

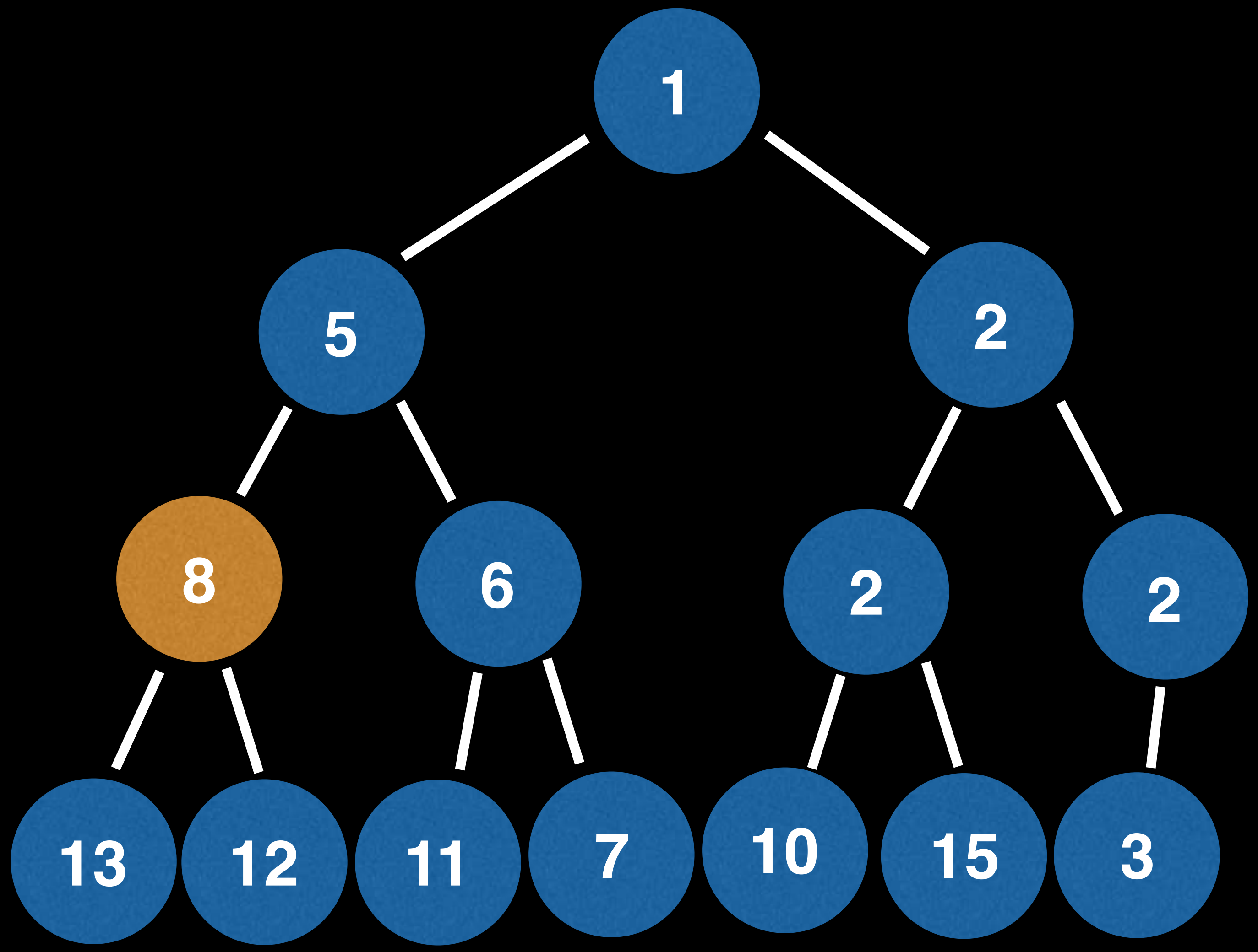
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

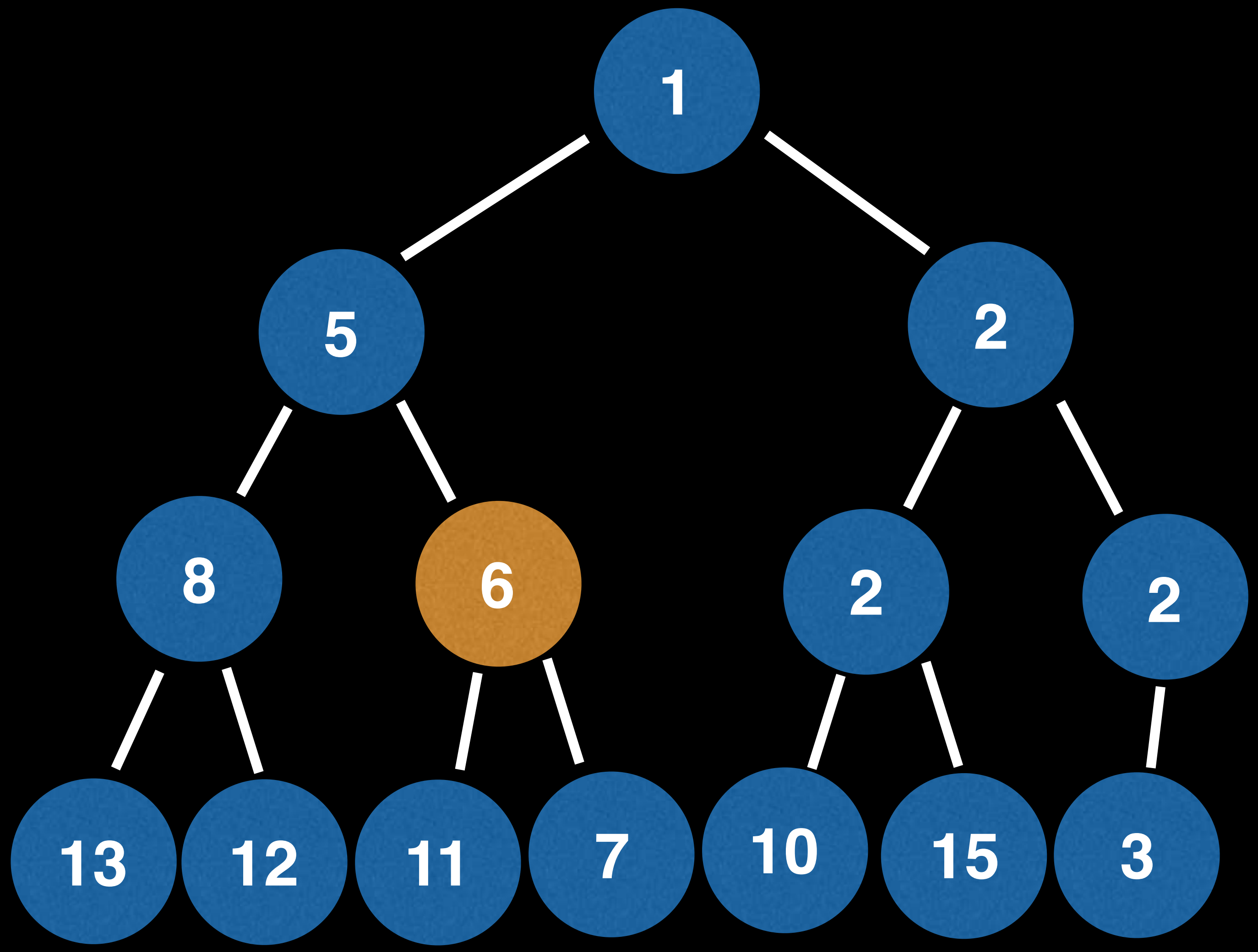
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

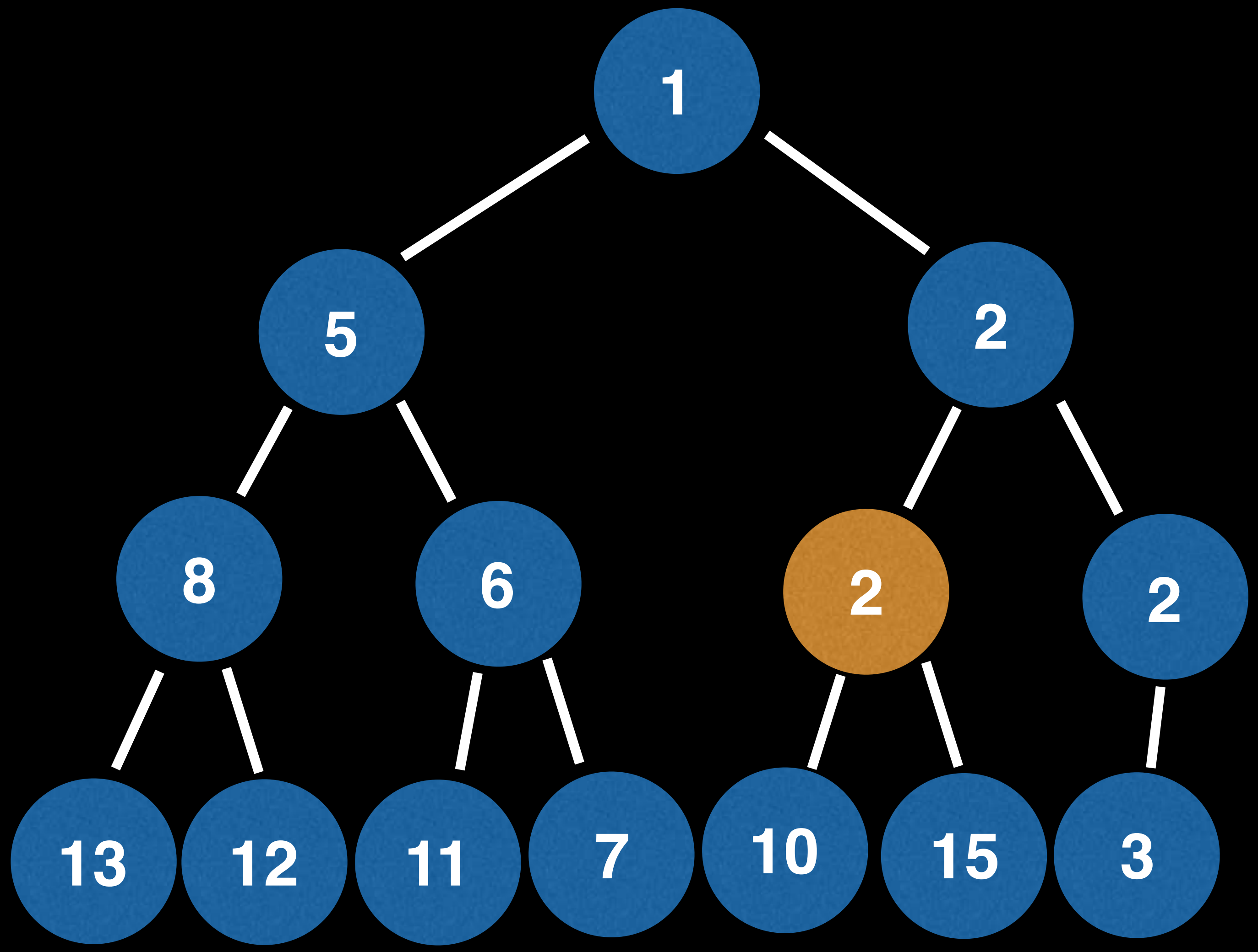
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

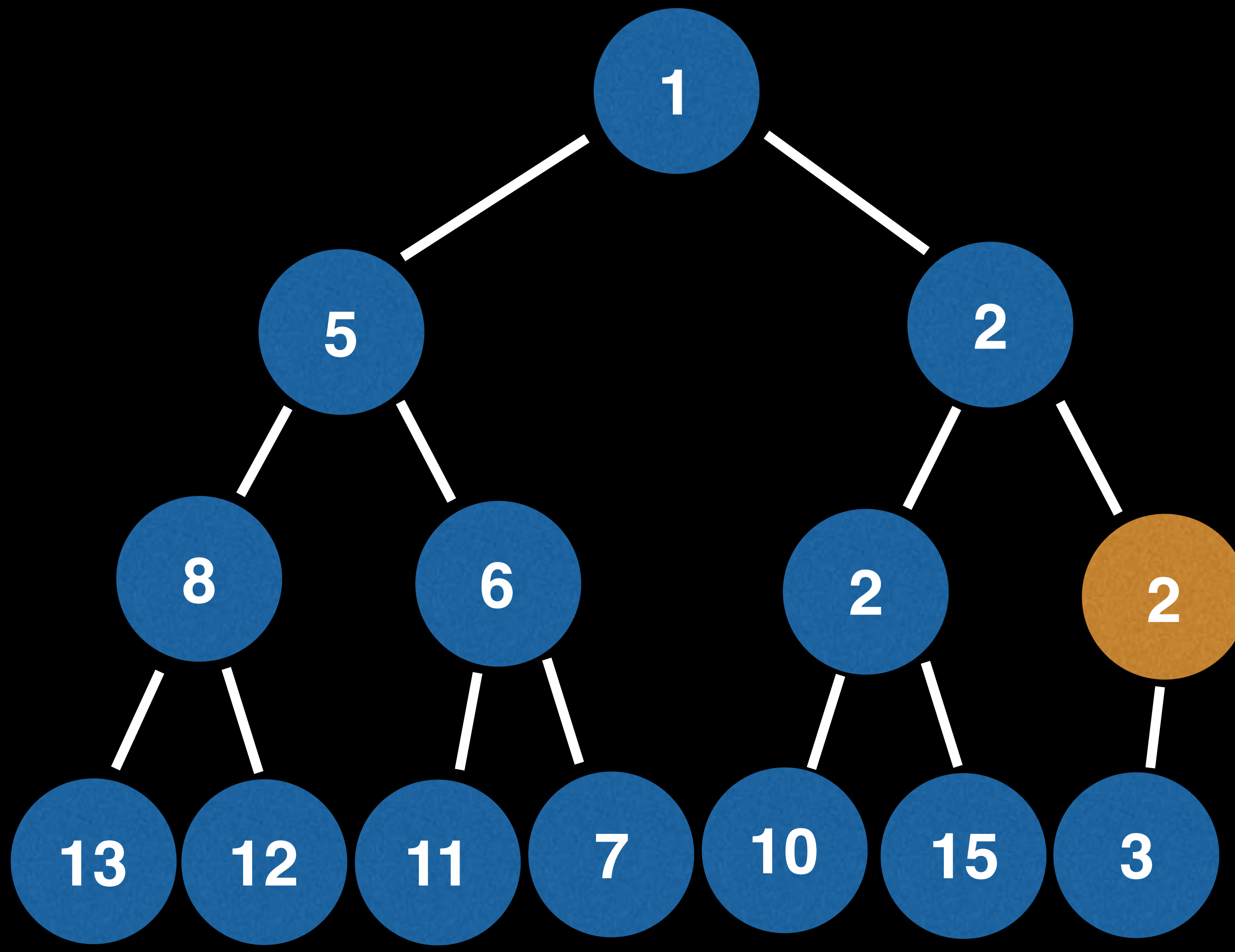
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

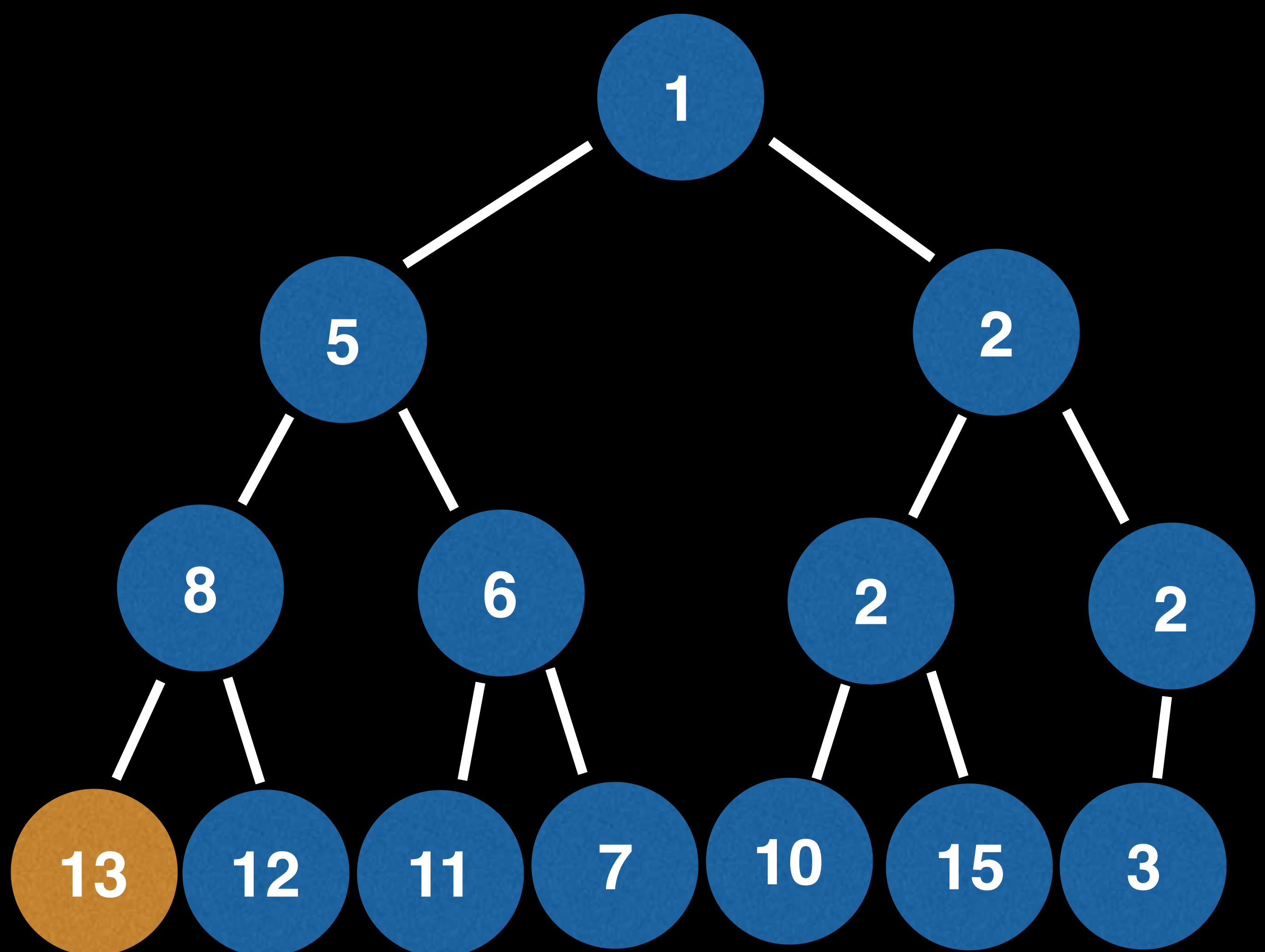
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

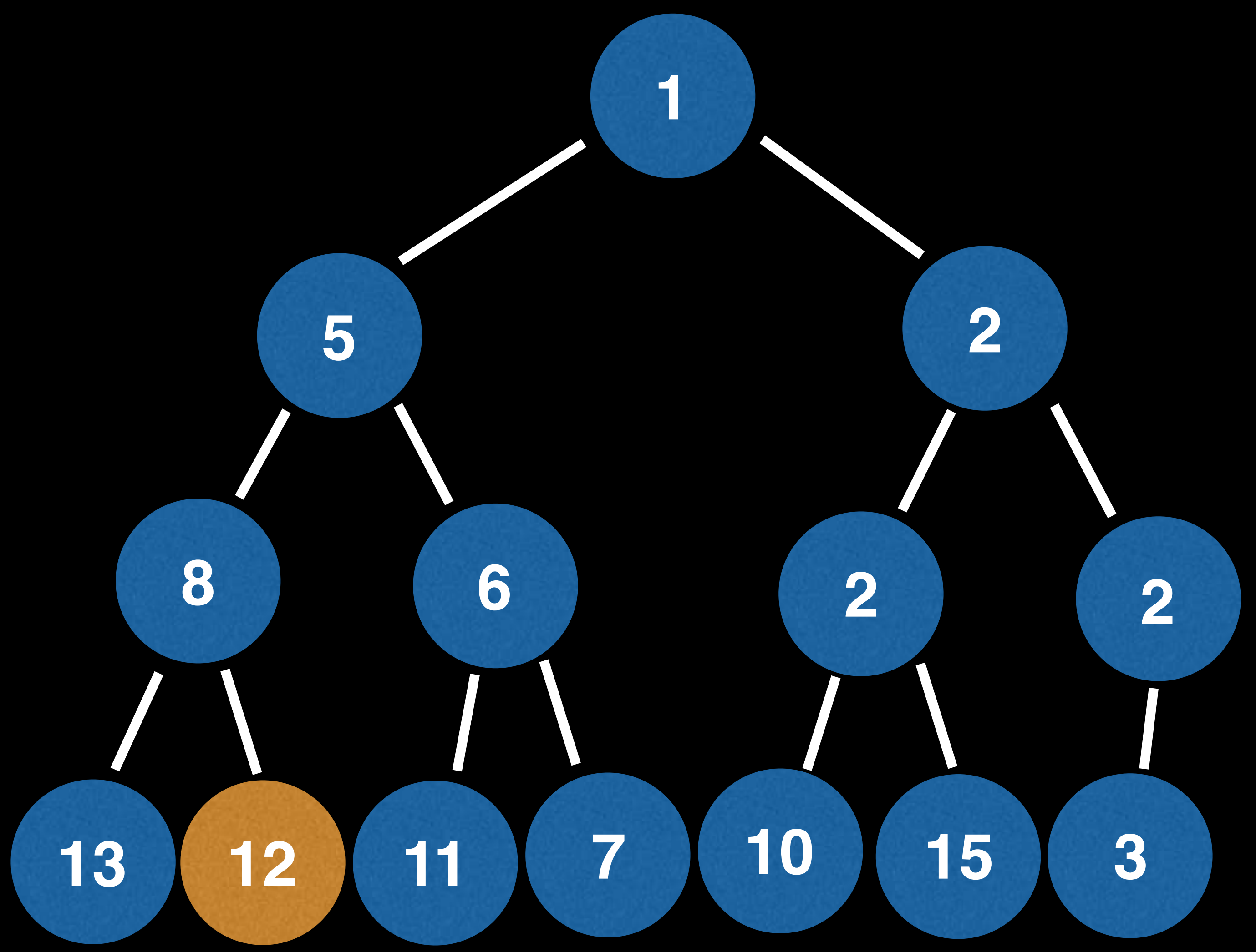
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

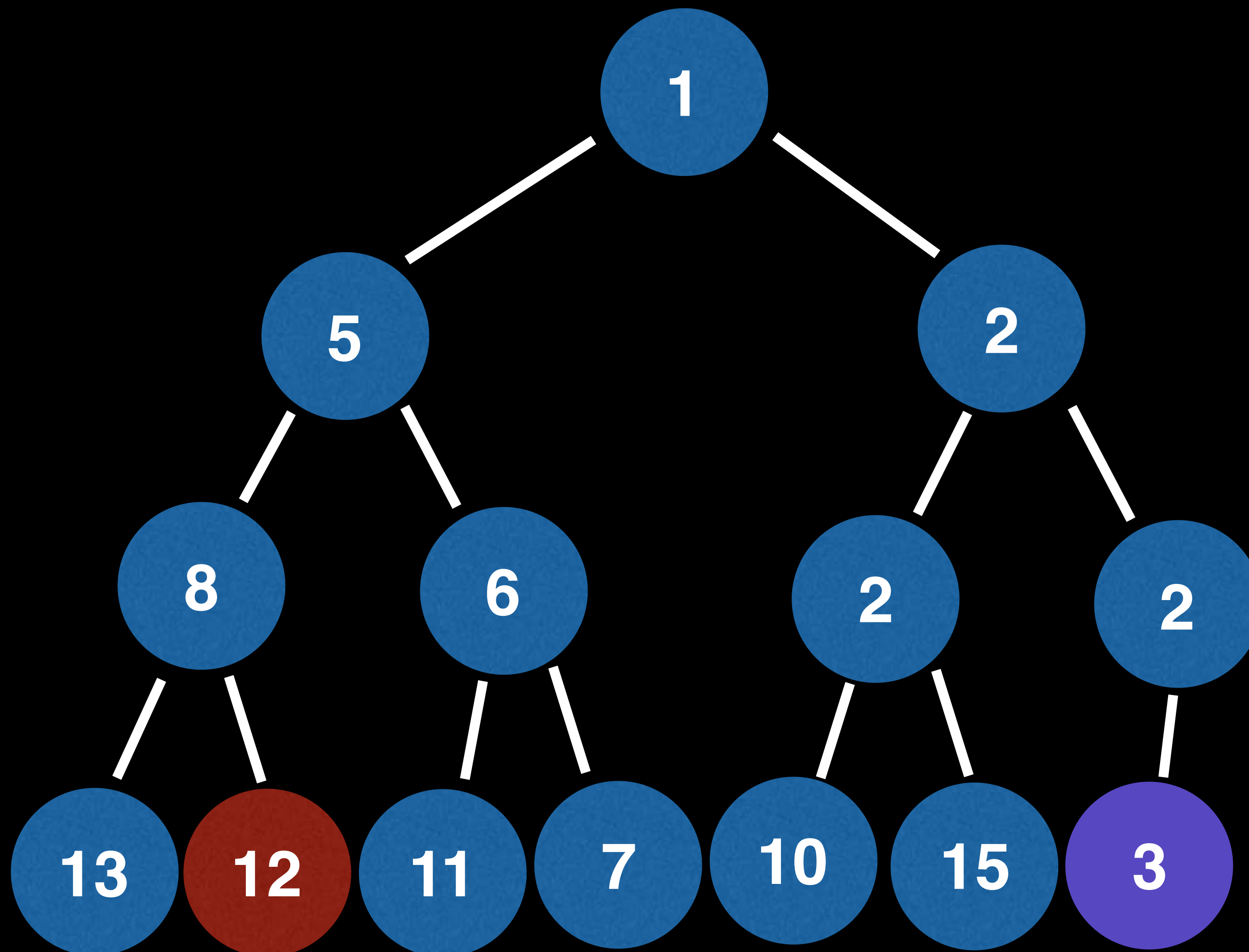
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

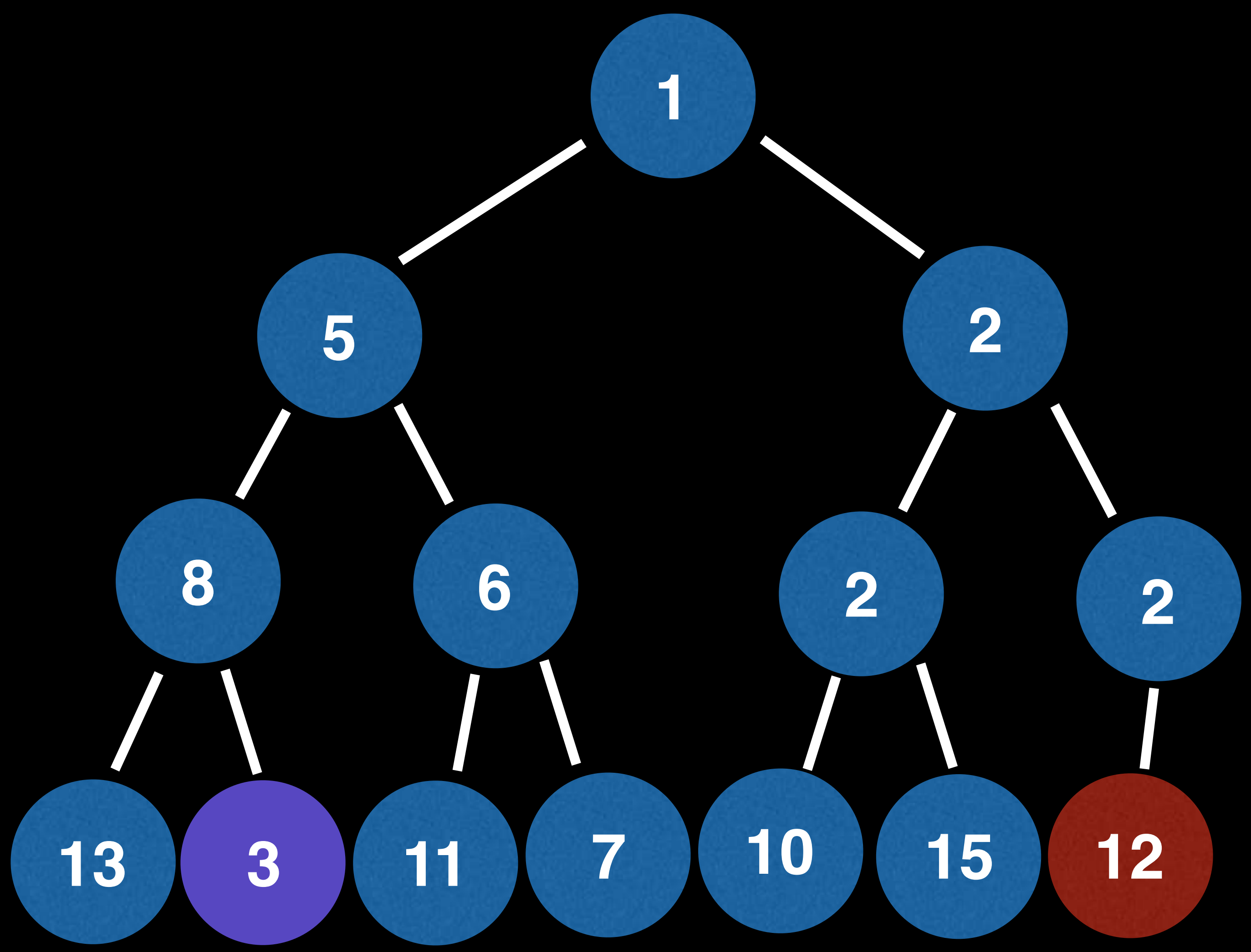
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

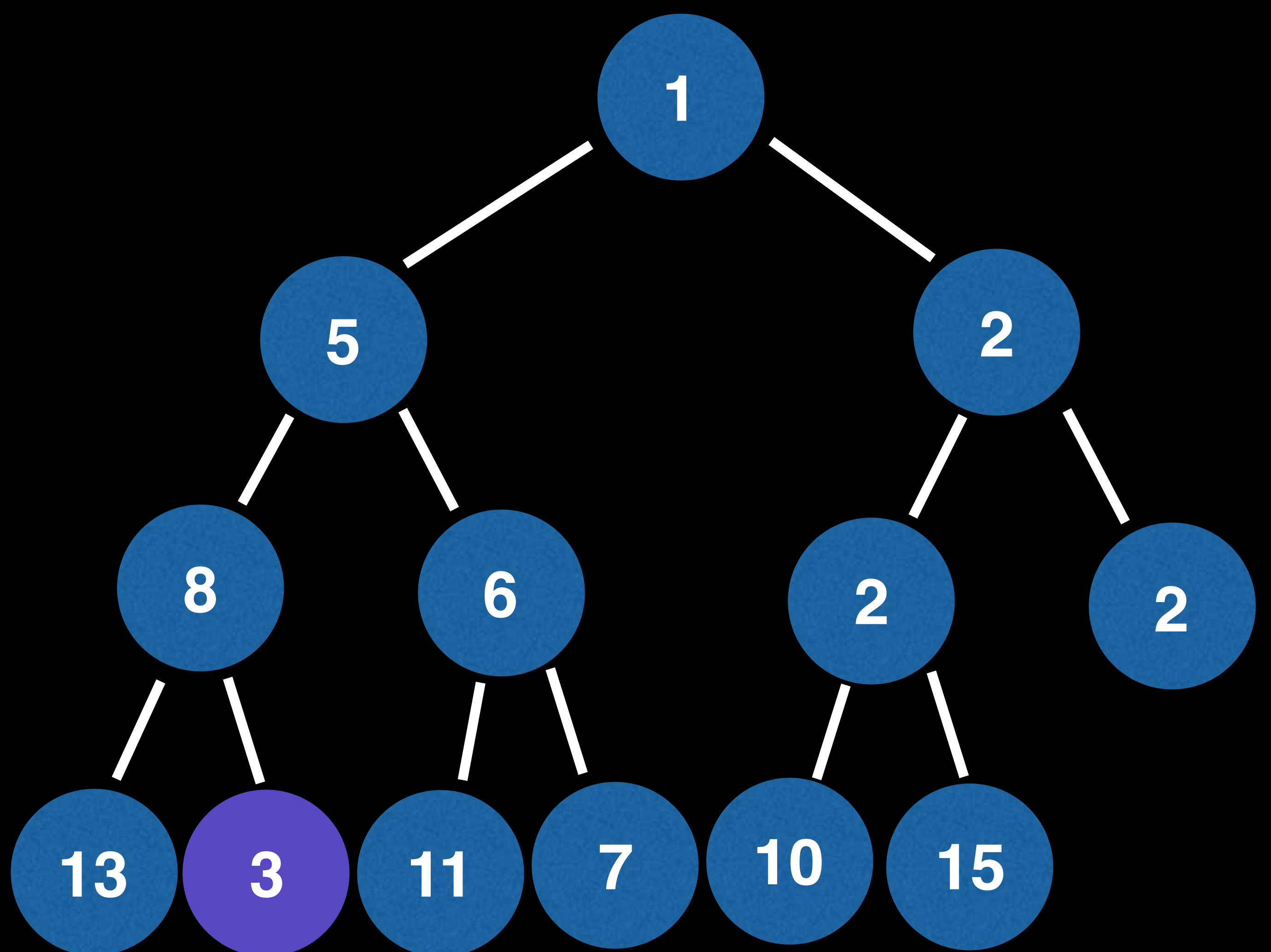
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

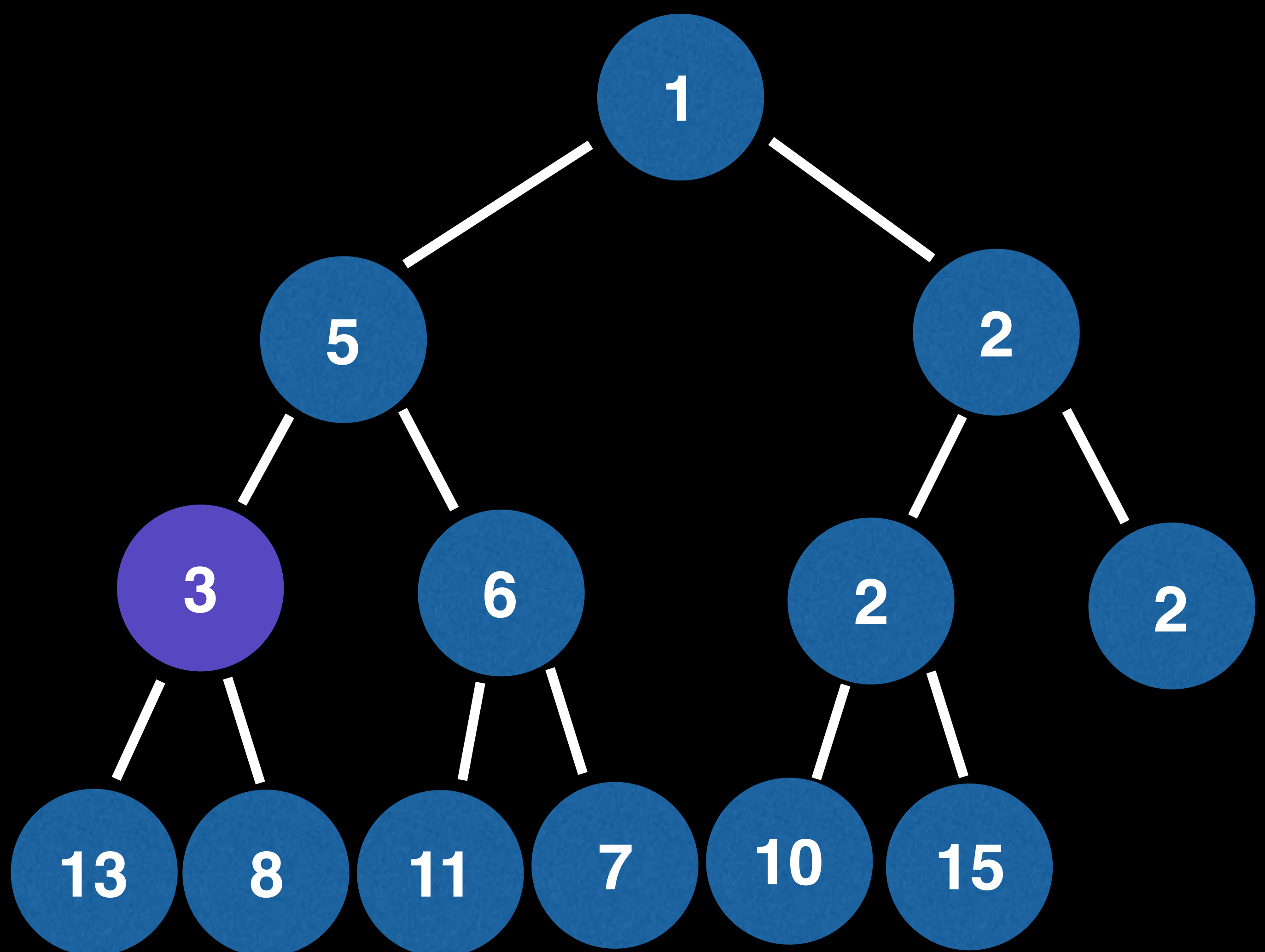
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

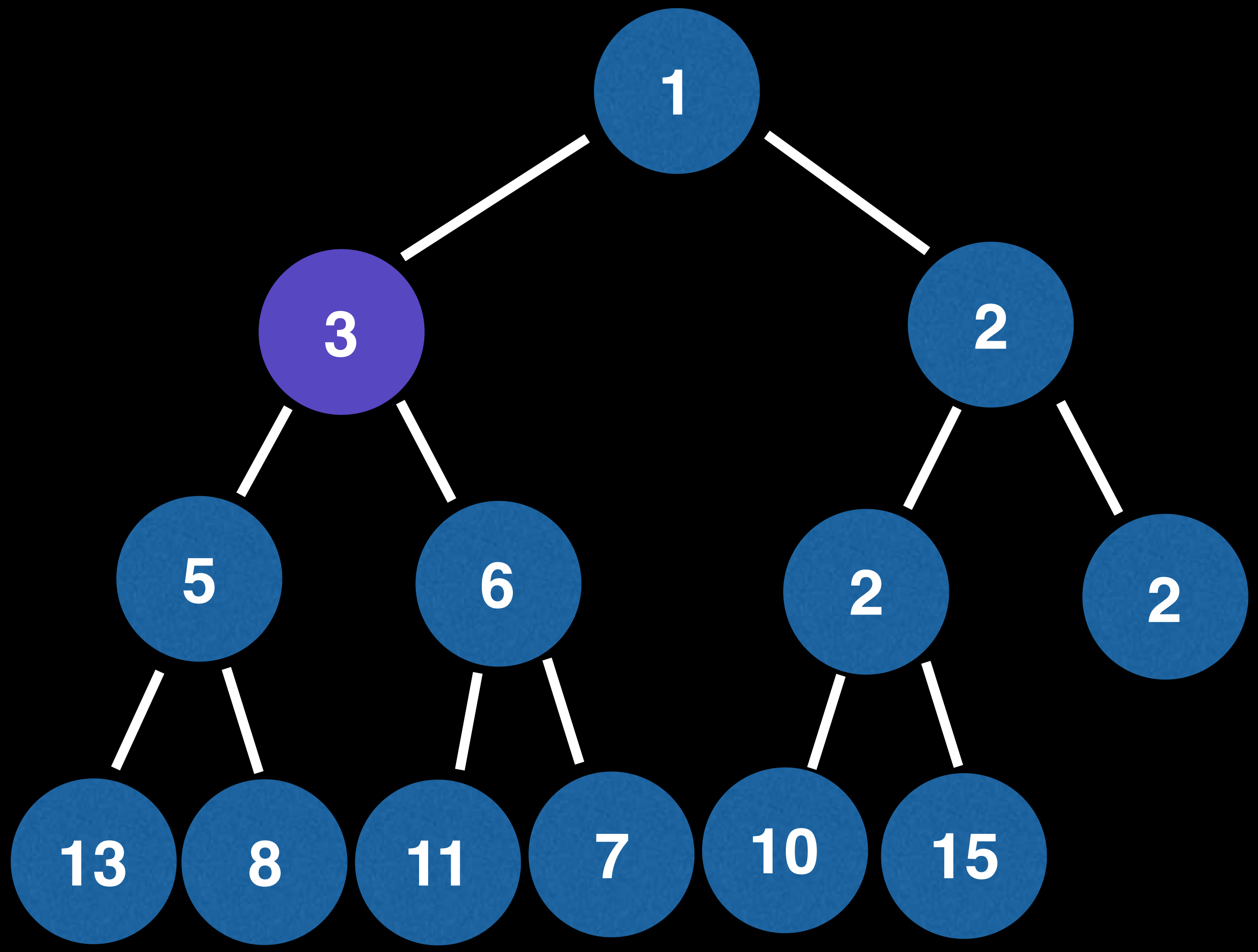
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

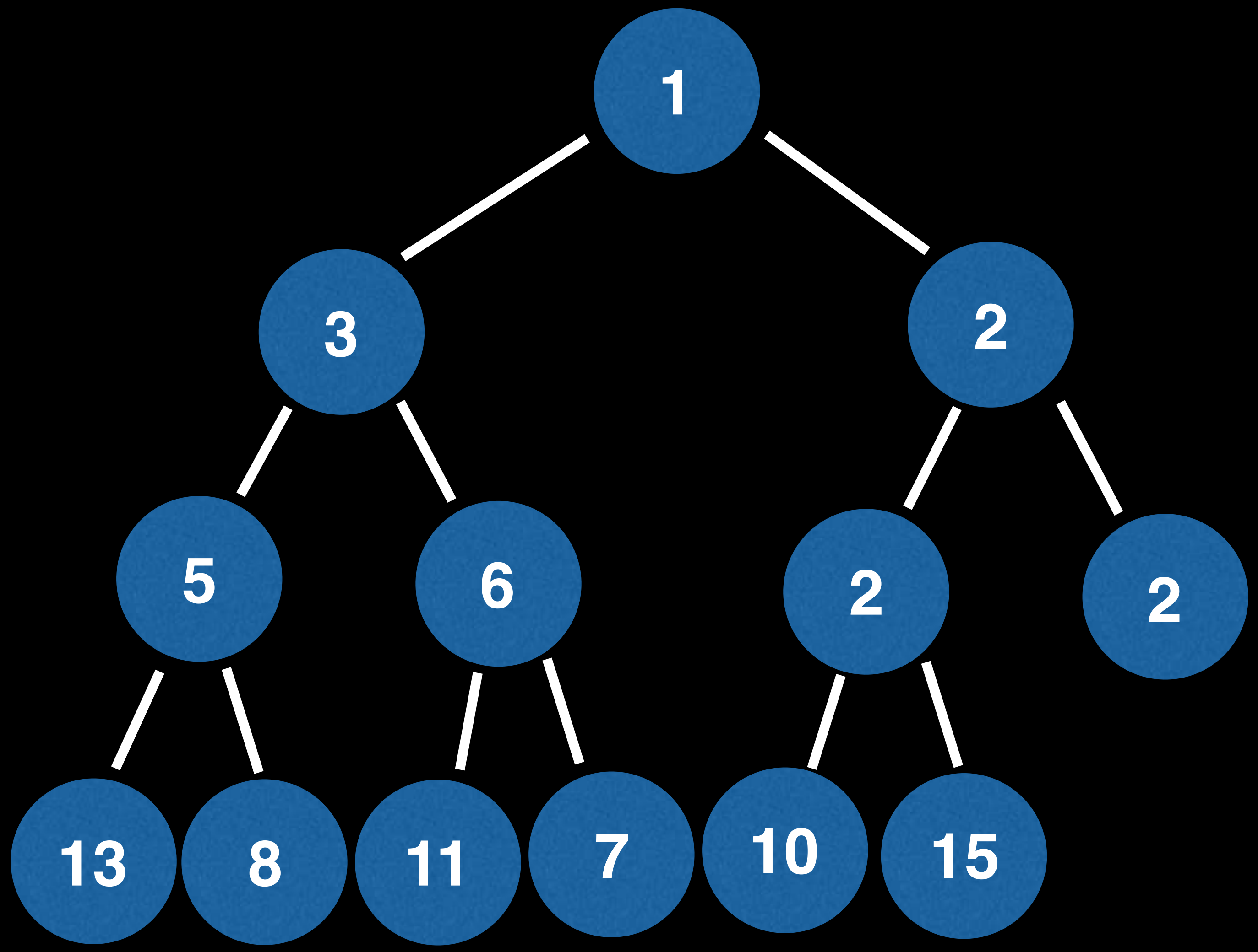
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

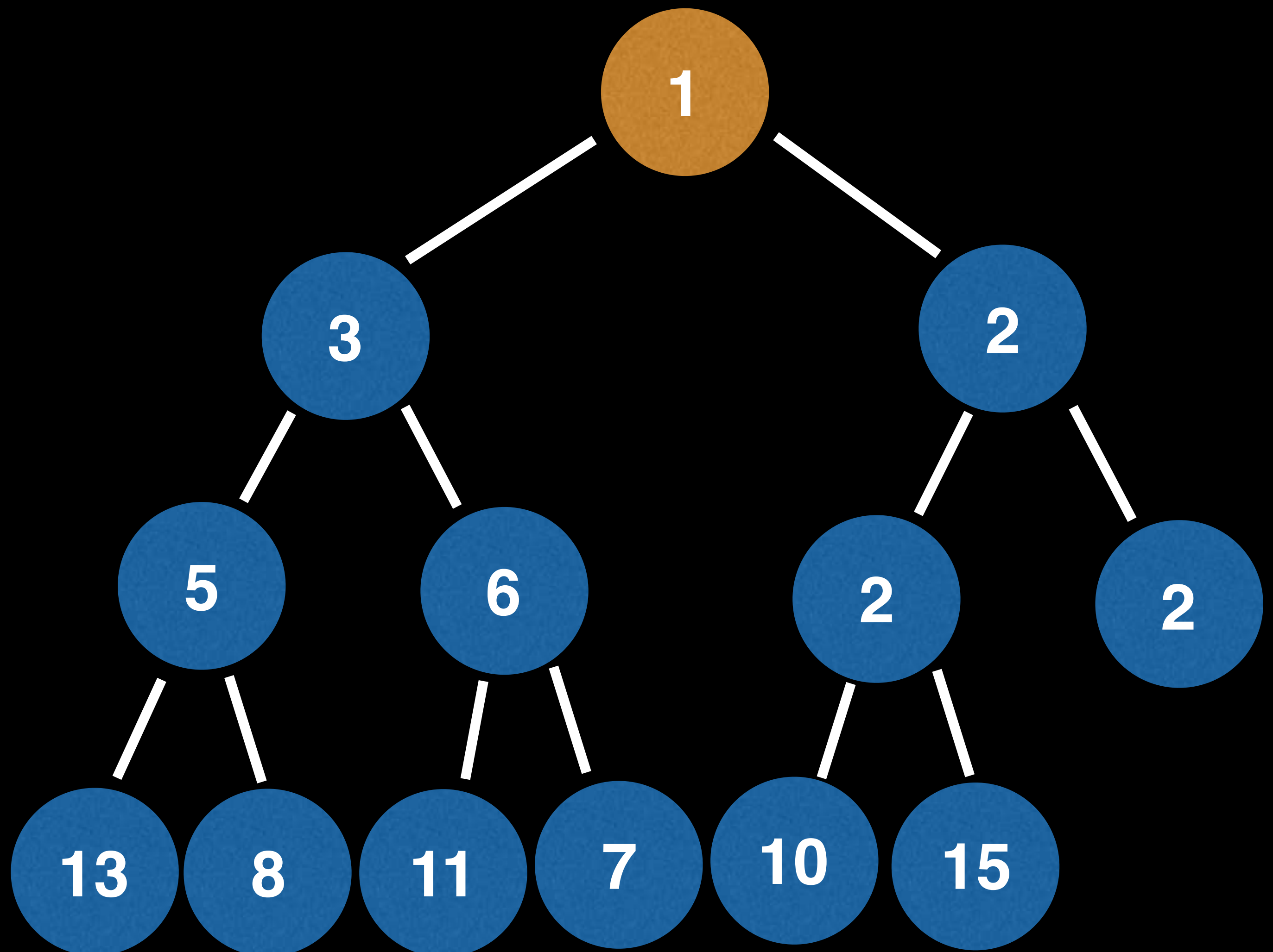
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

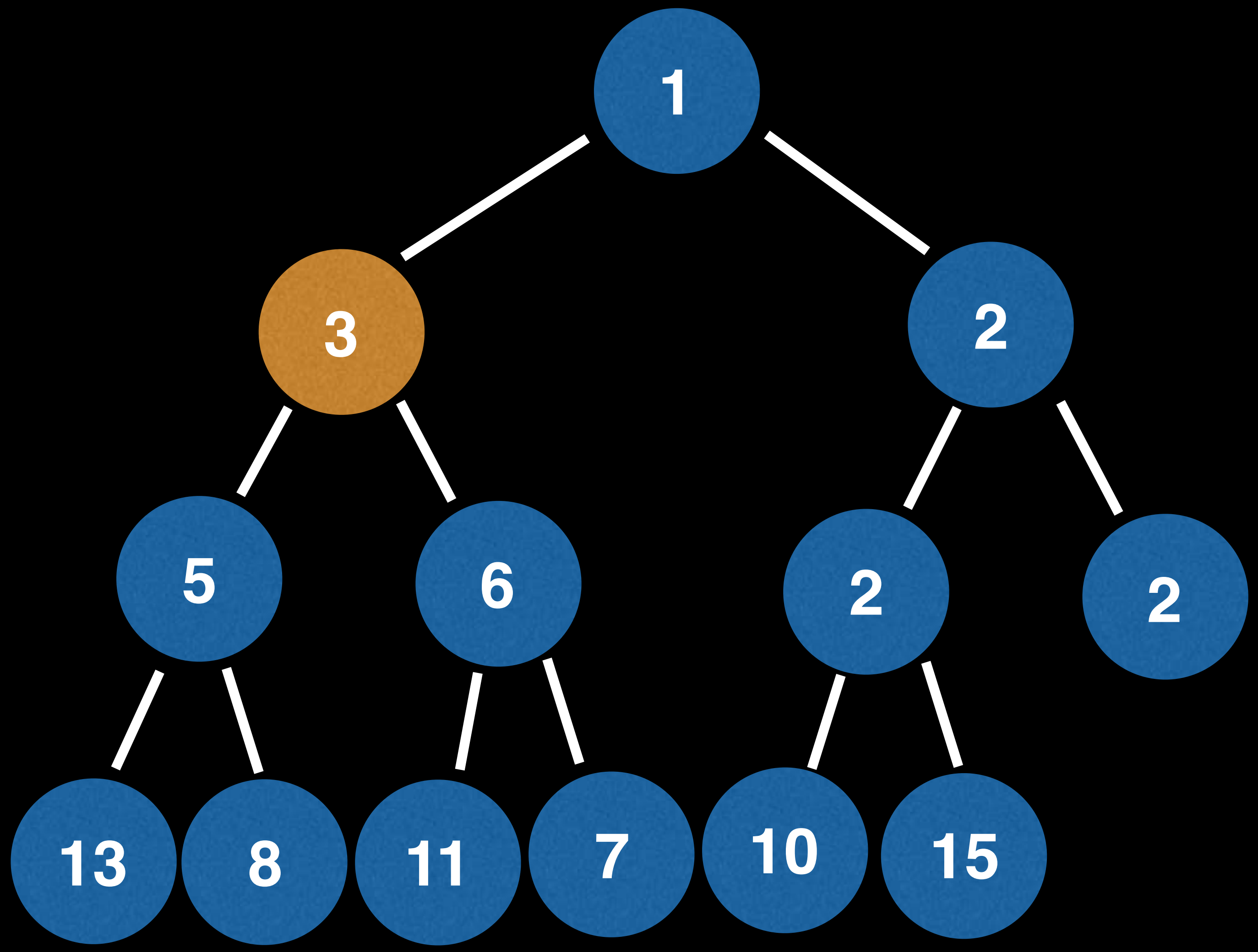
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

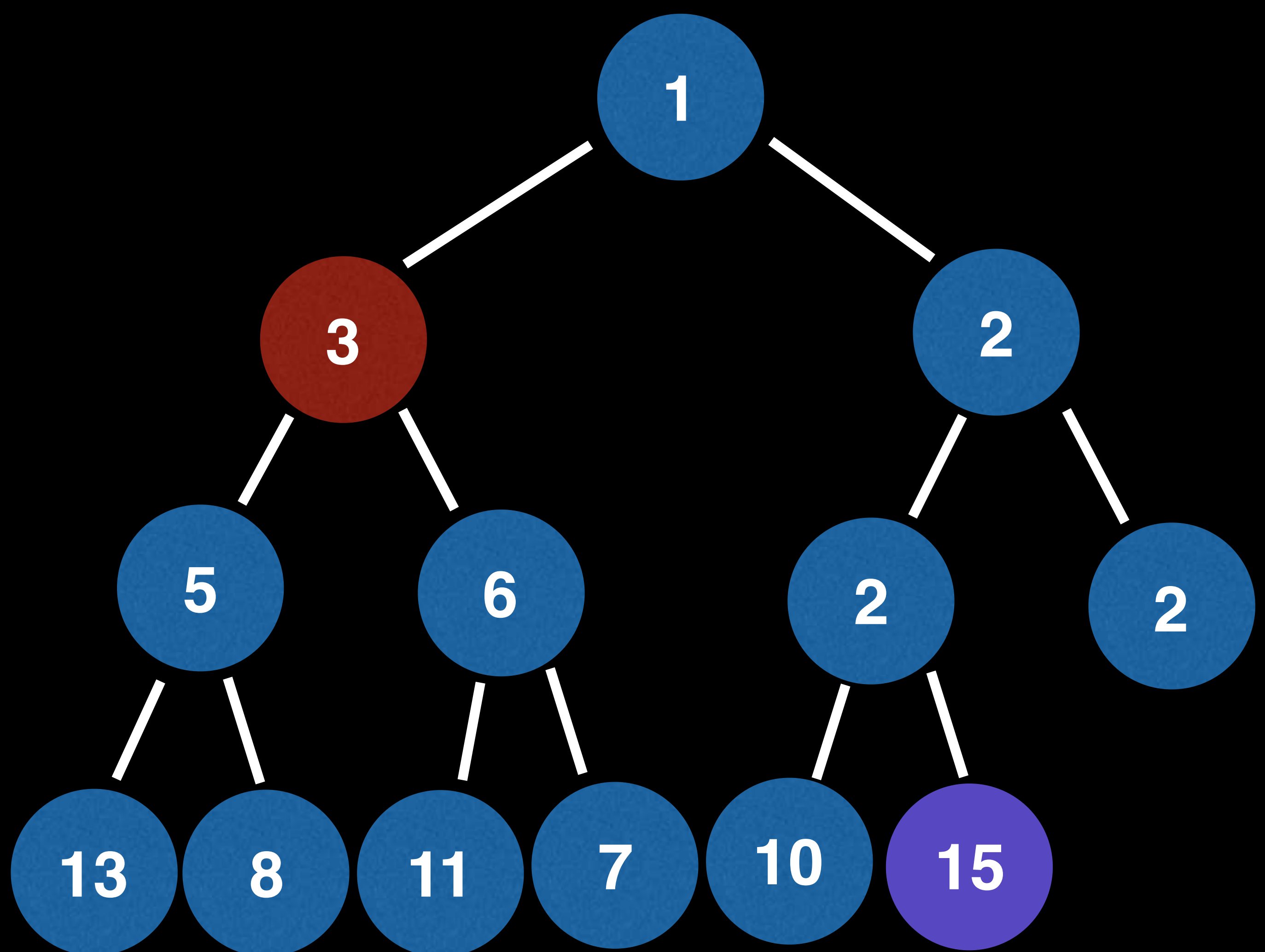
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

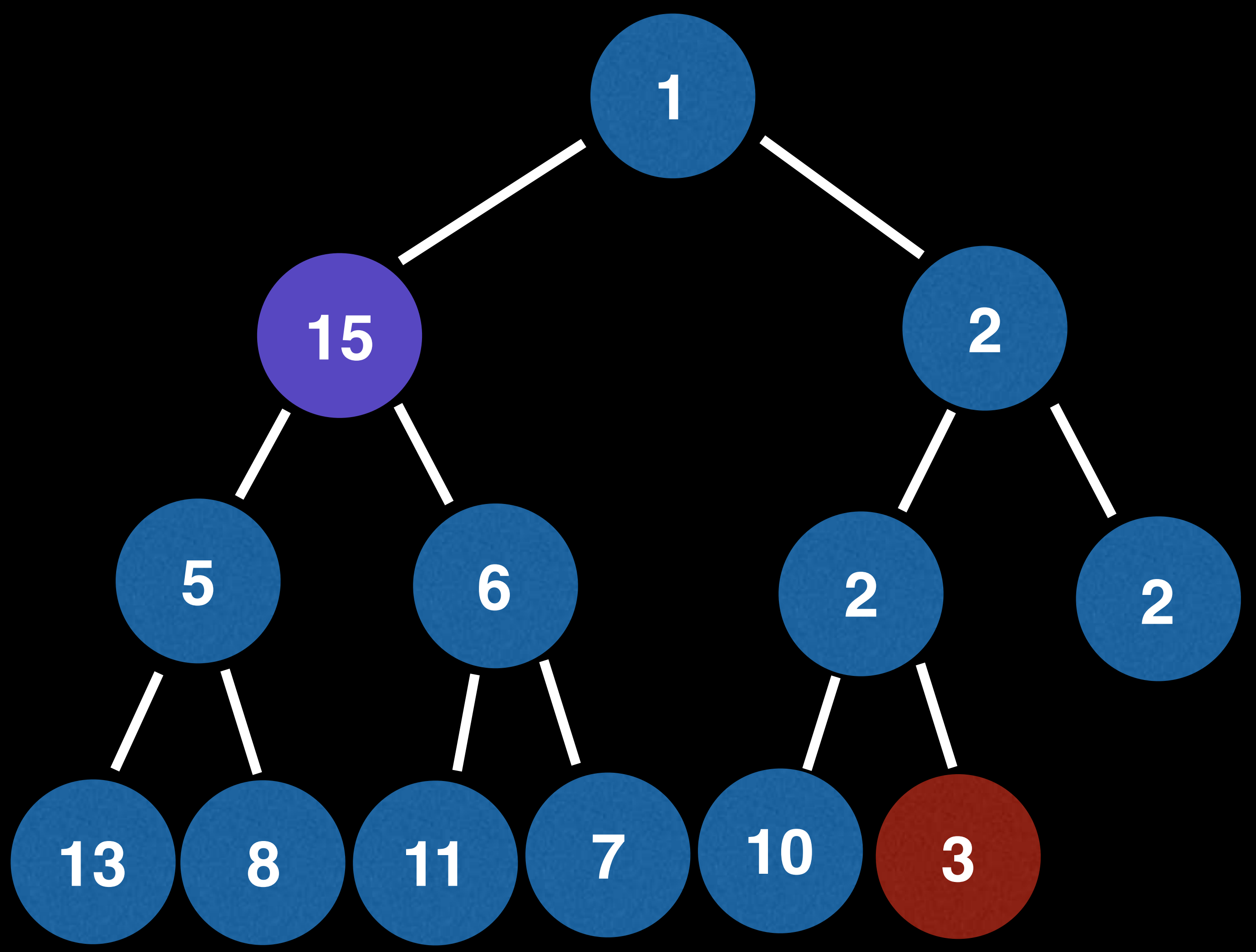
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

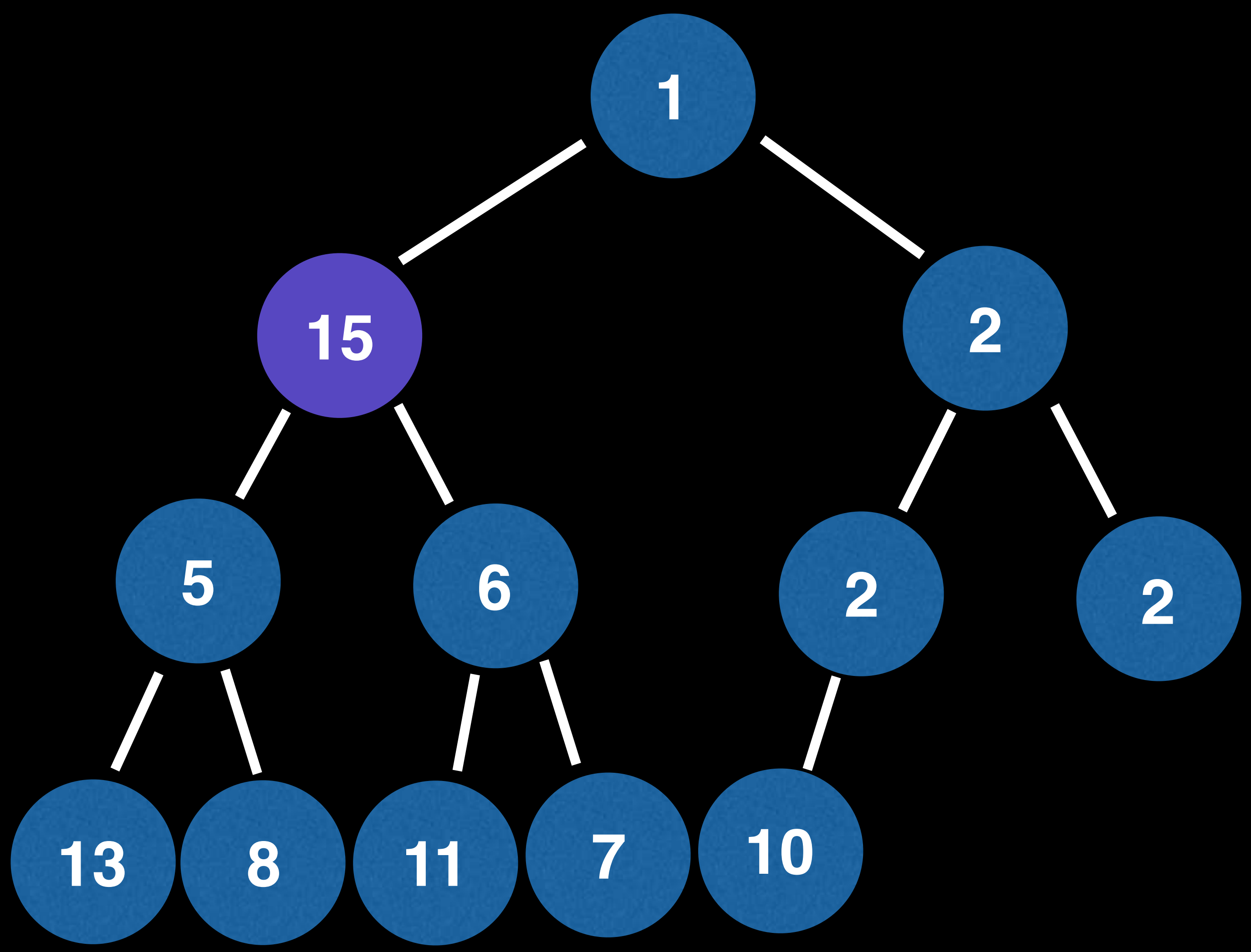
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

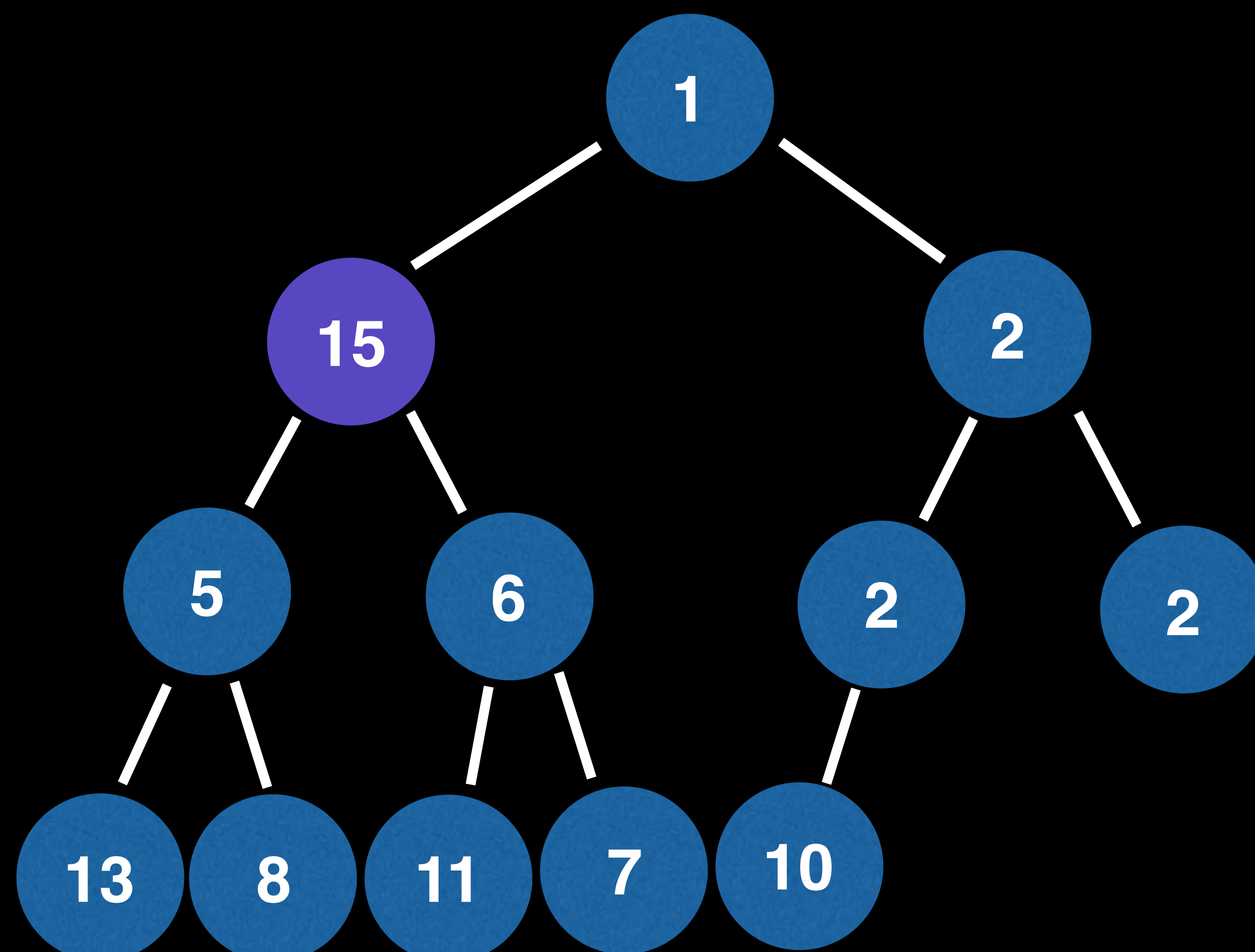
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)

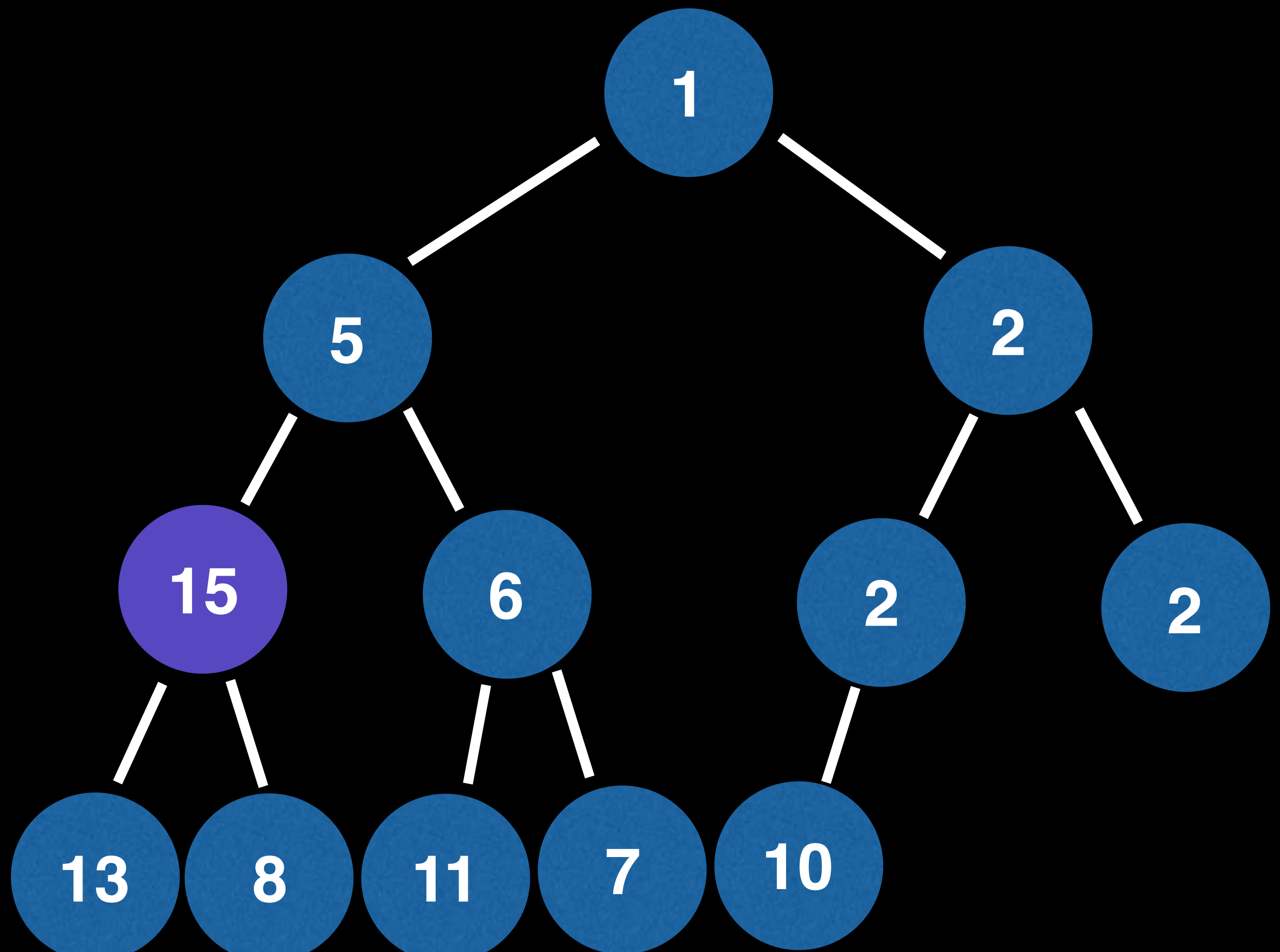


Do we bubble up or bubble down? We already satisfy the heap invariant from above, so bubble down it is!

Removing Elements From a Binary Heap

Instructions:

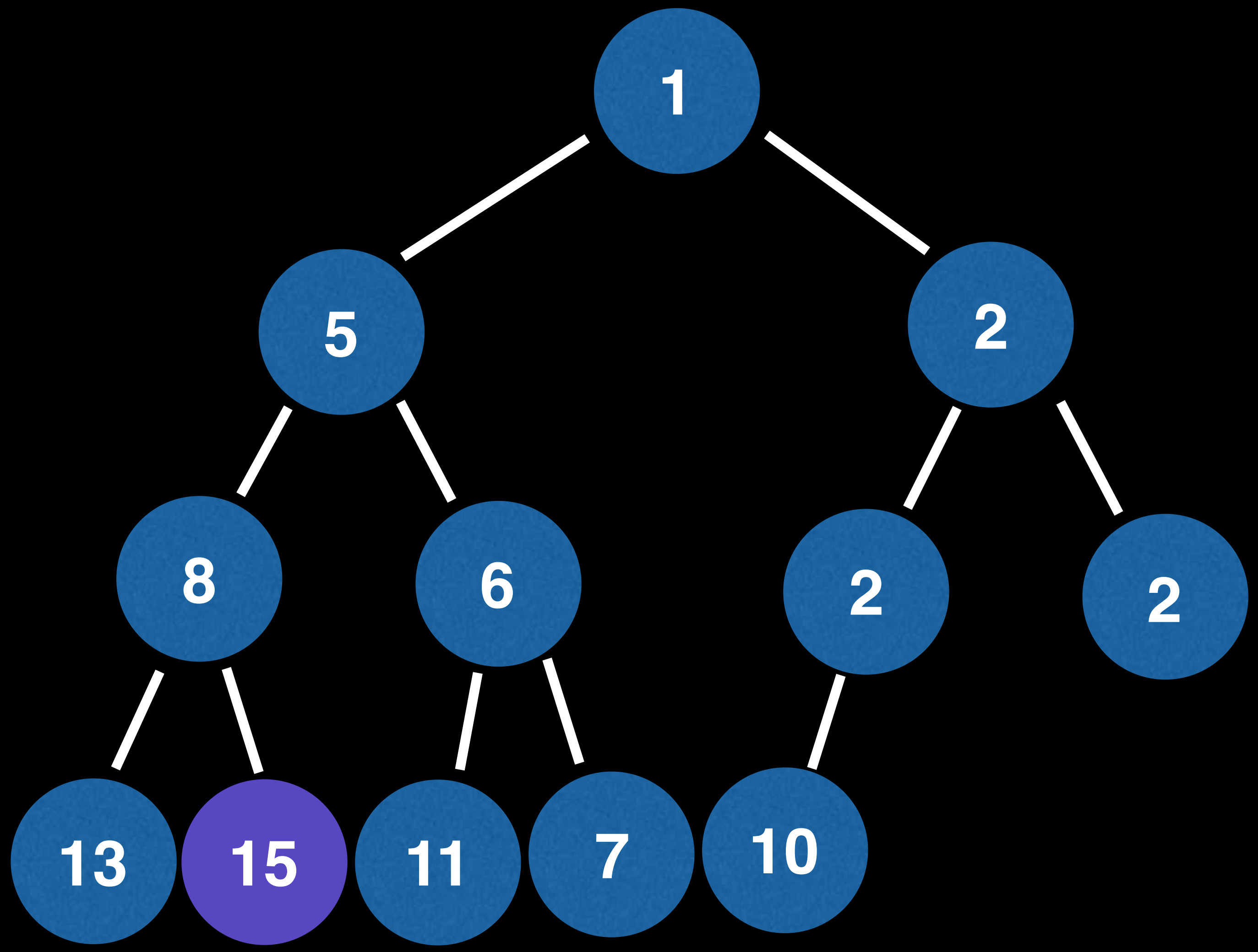
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

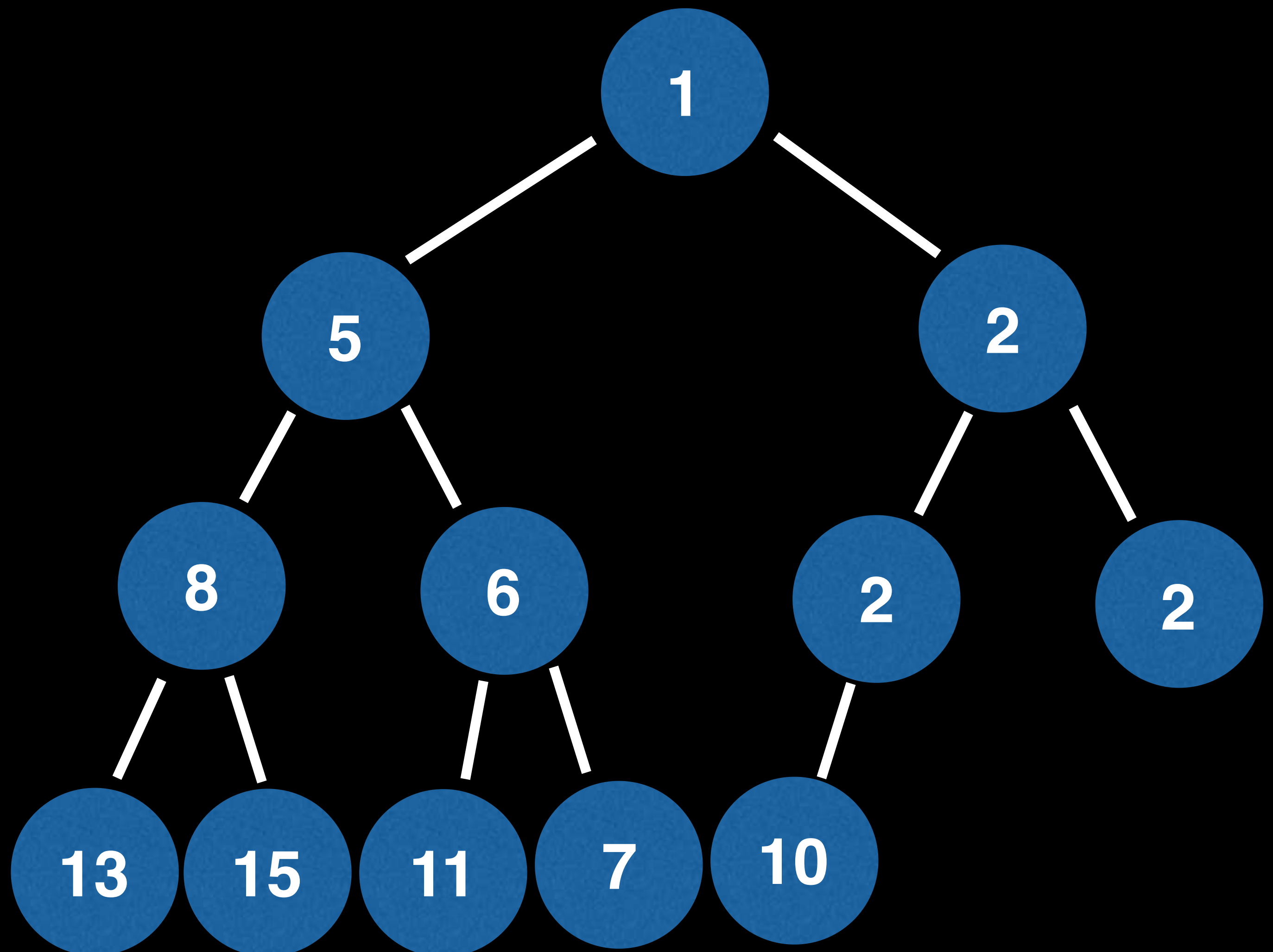
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

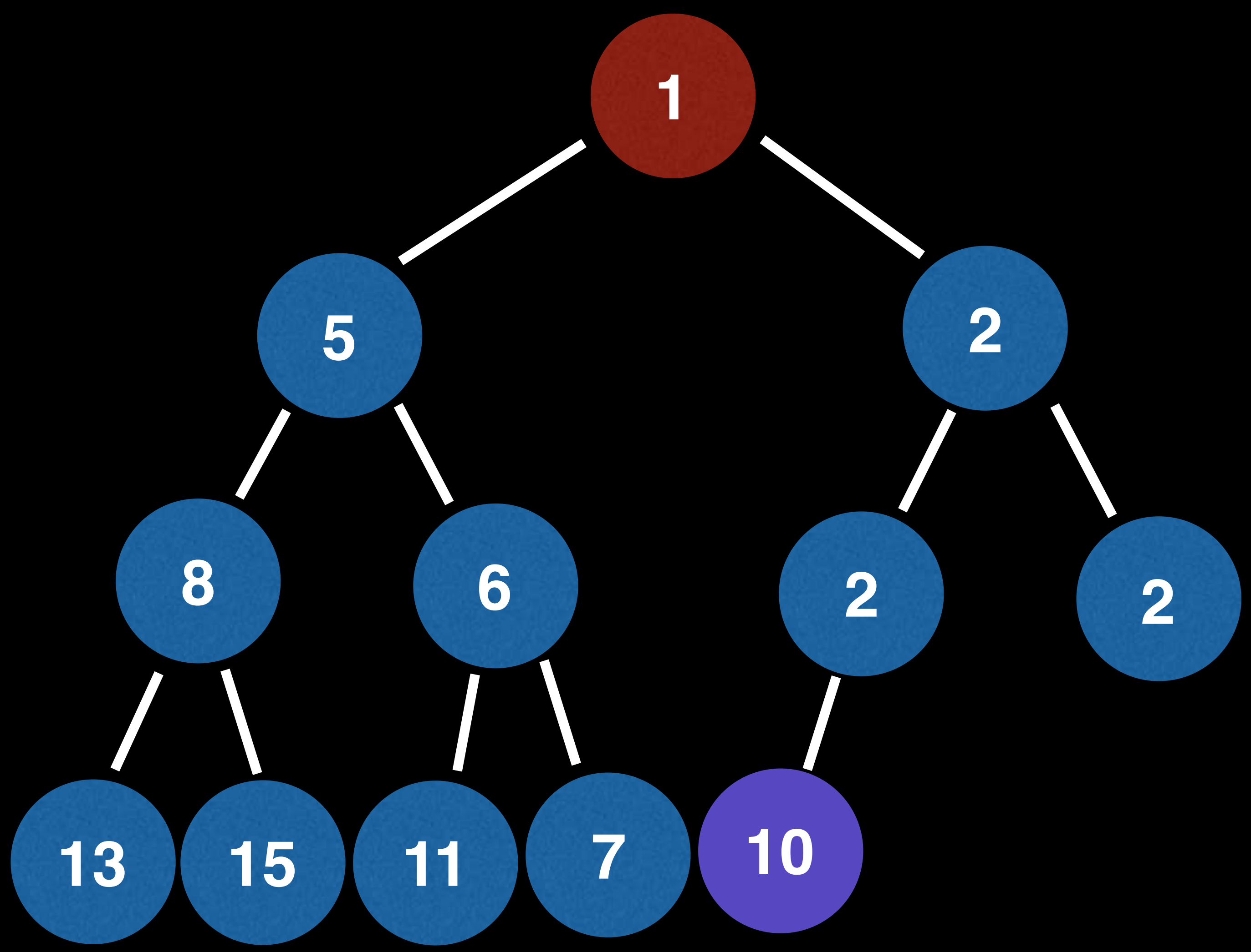
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

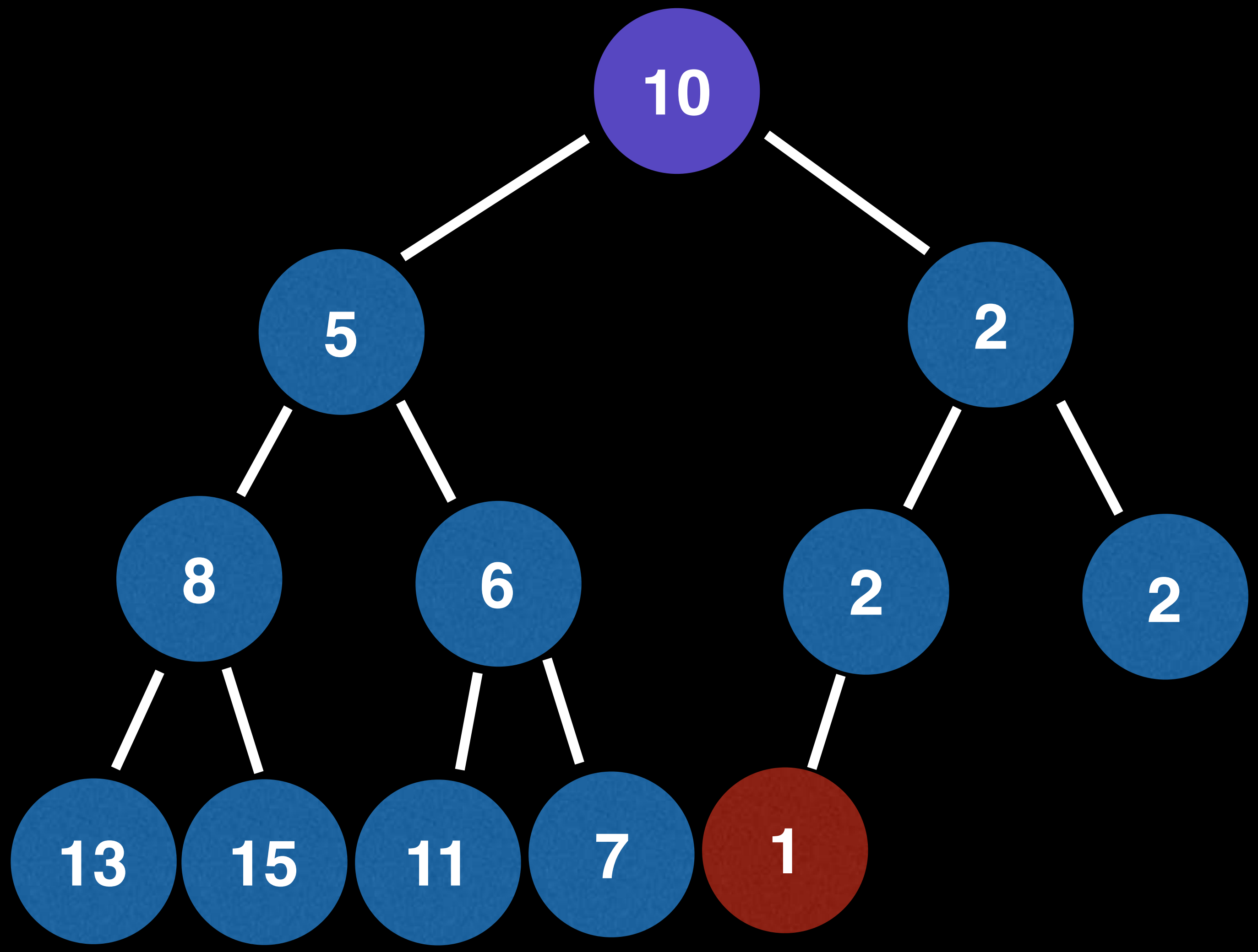
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

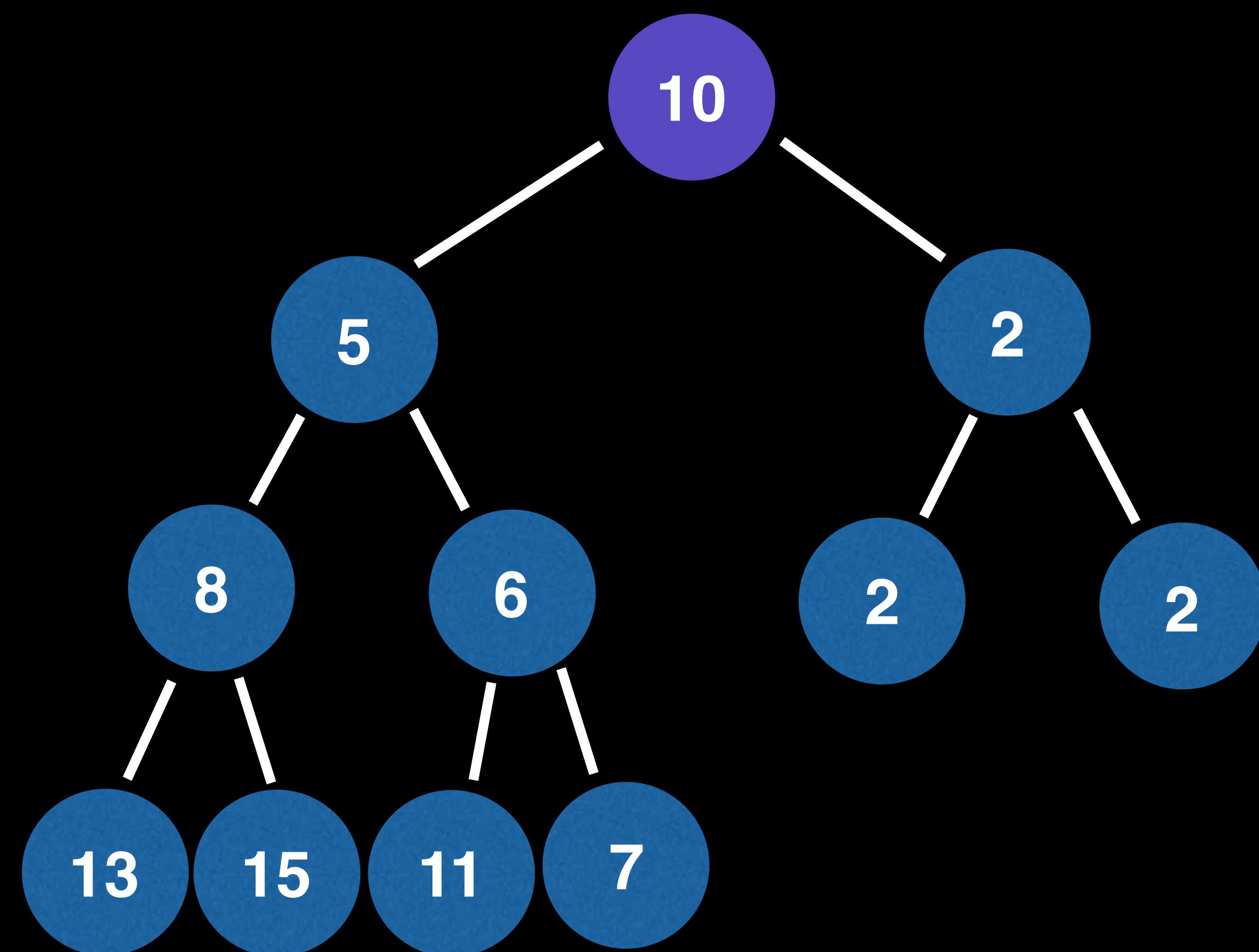
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

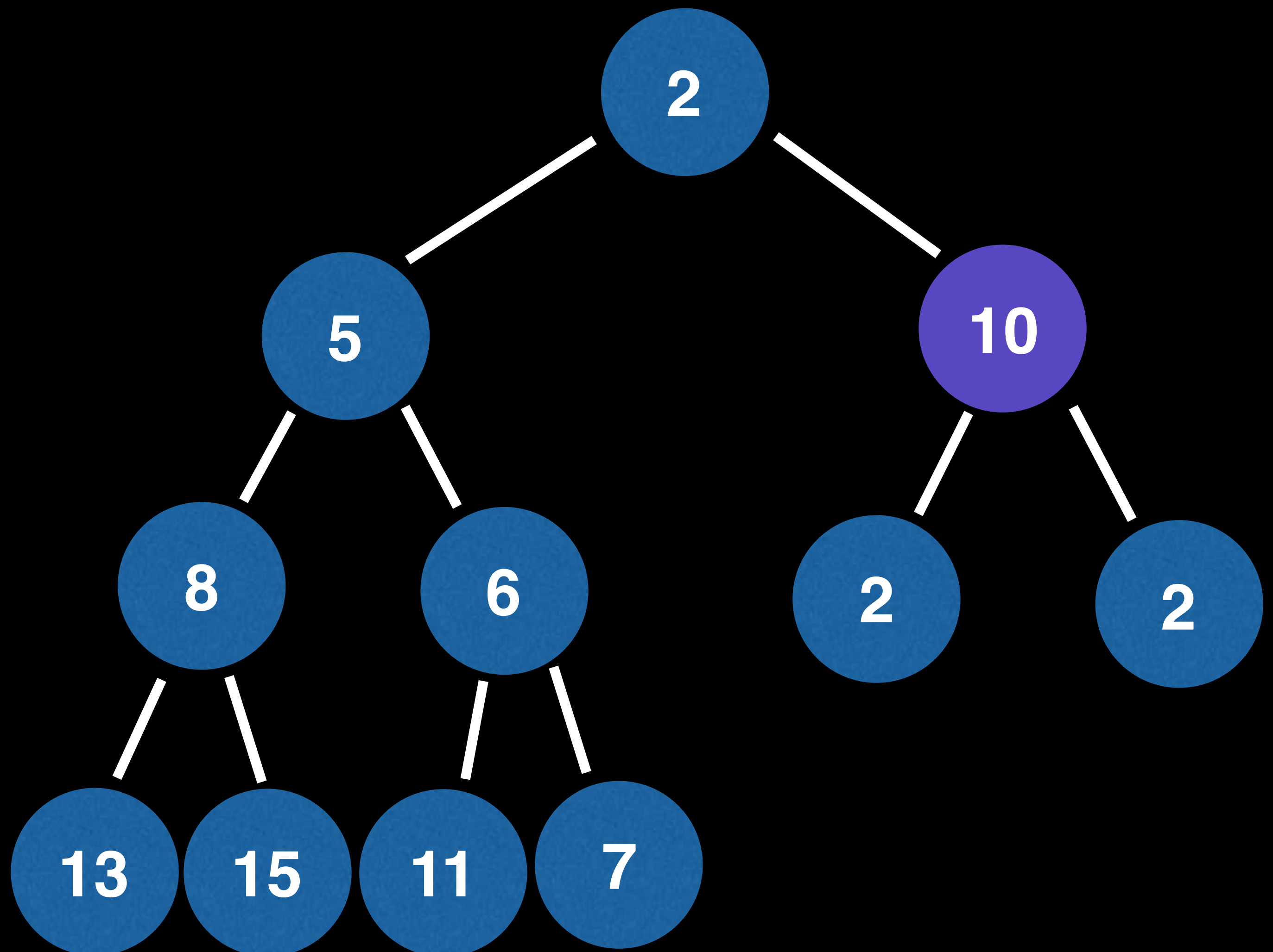
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

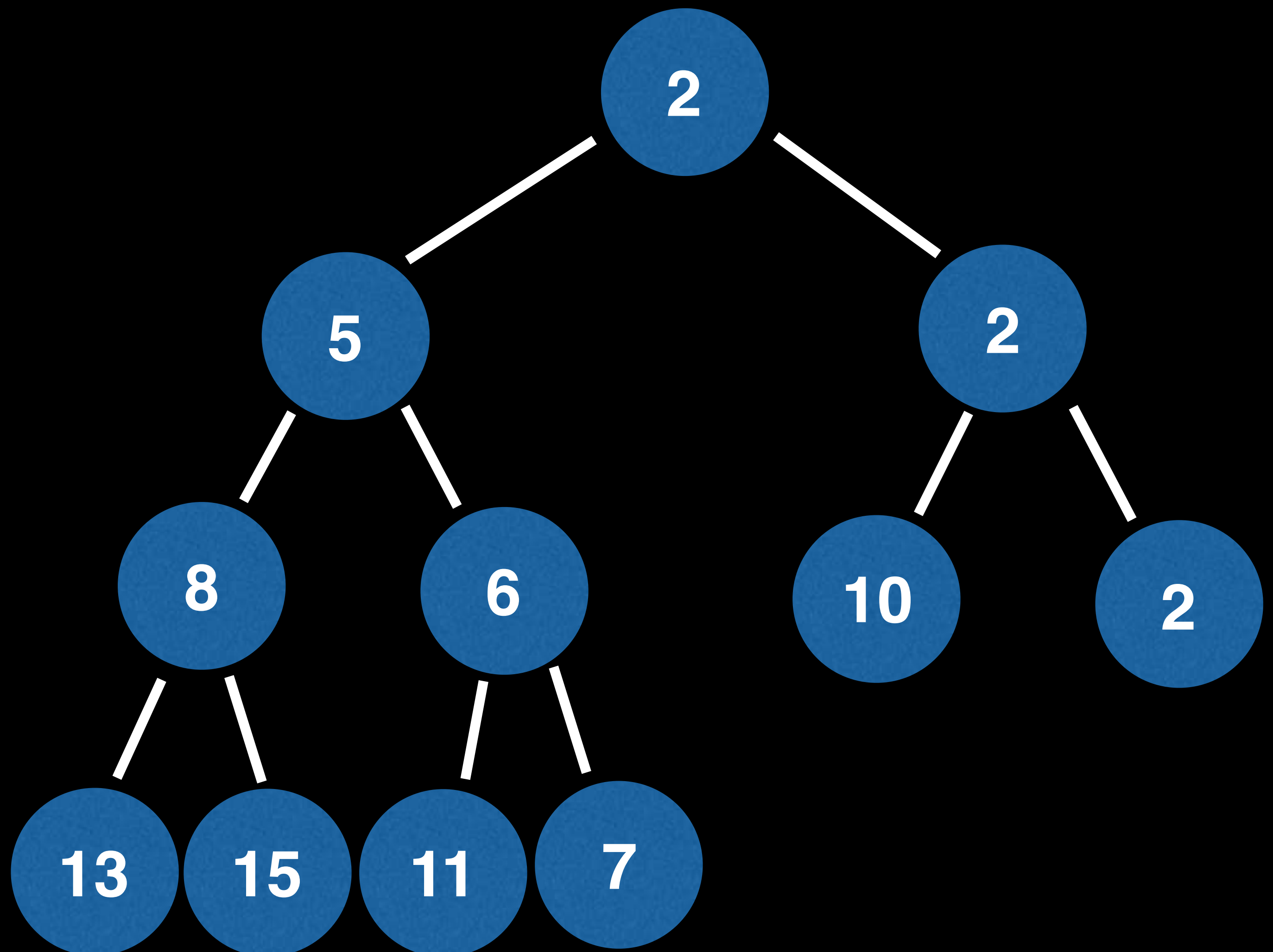
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

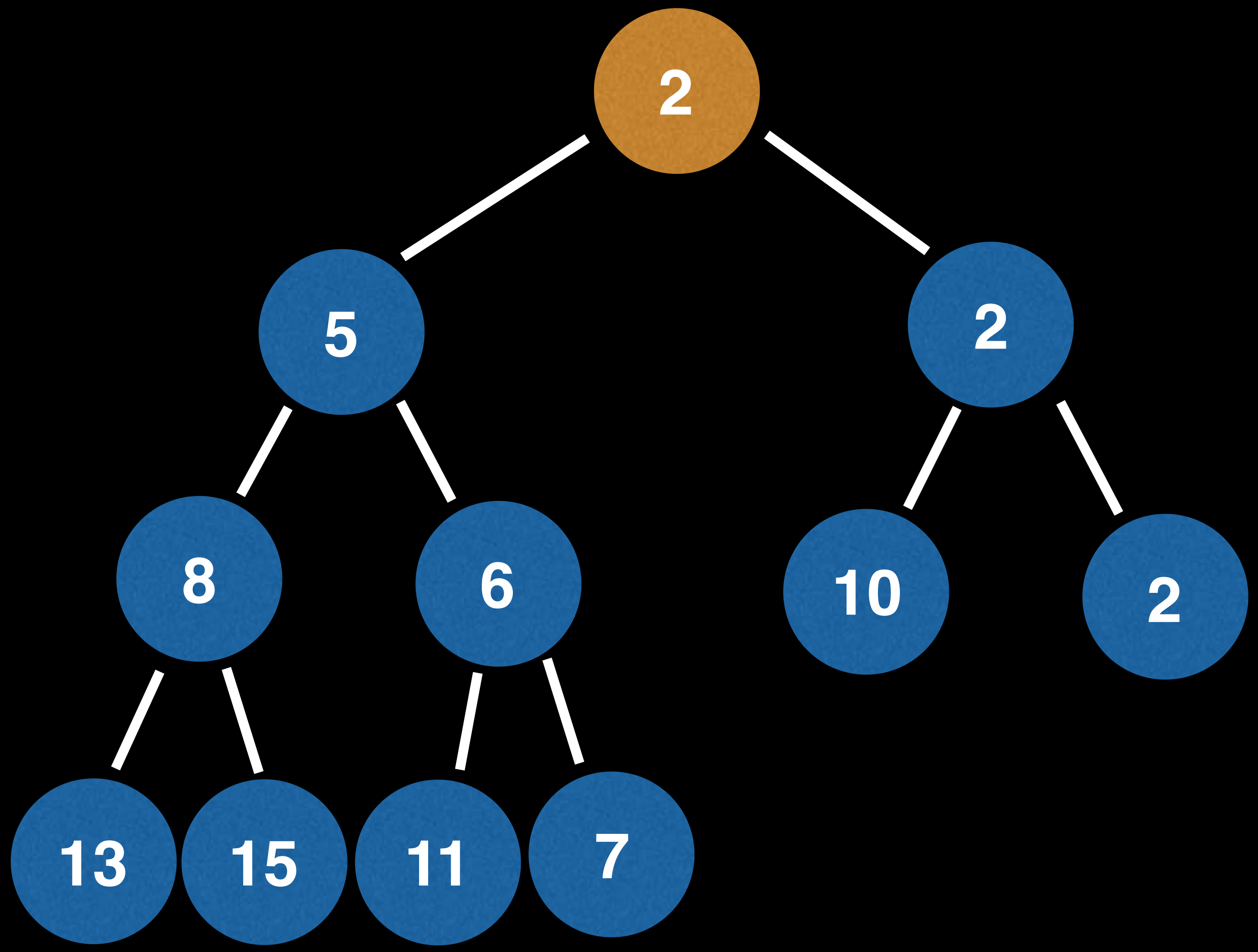
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

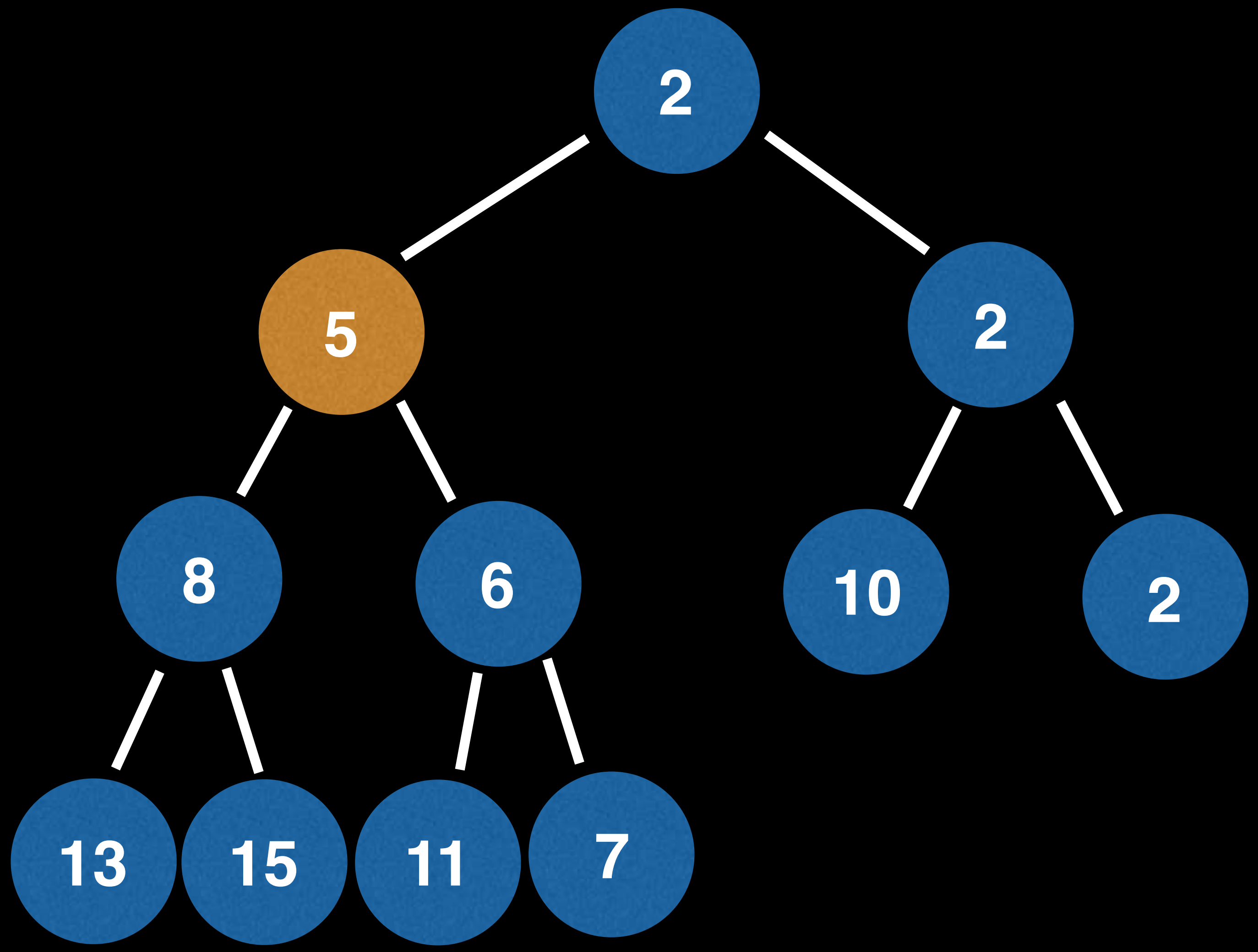
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

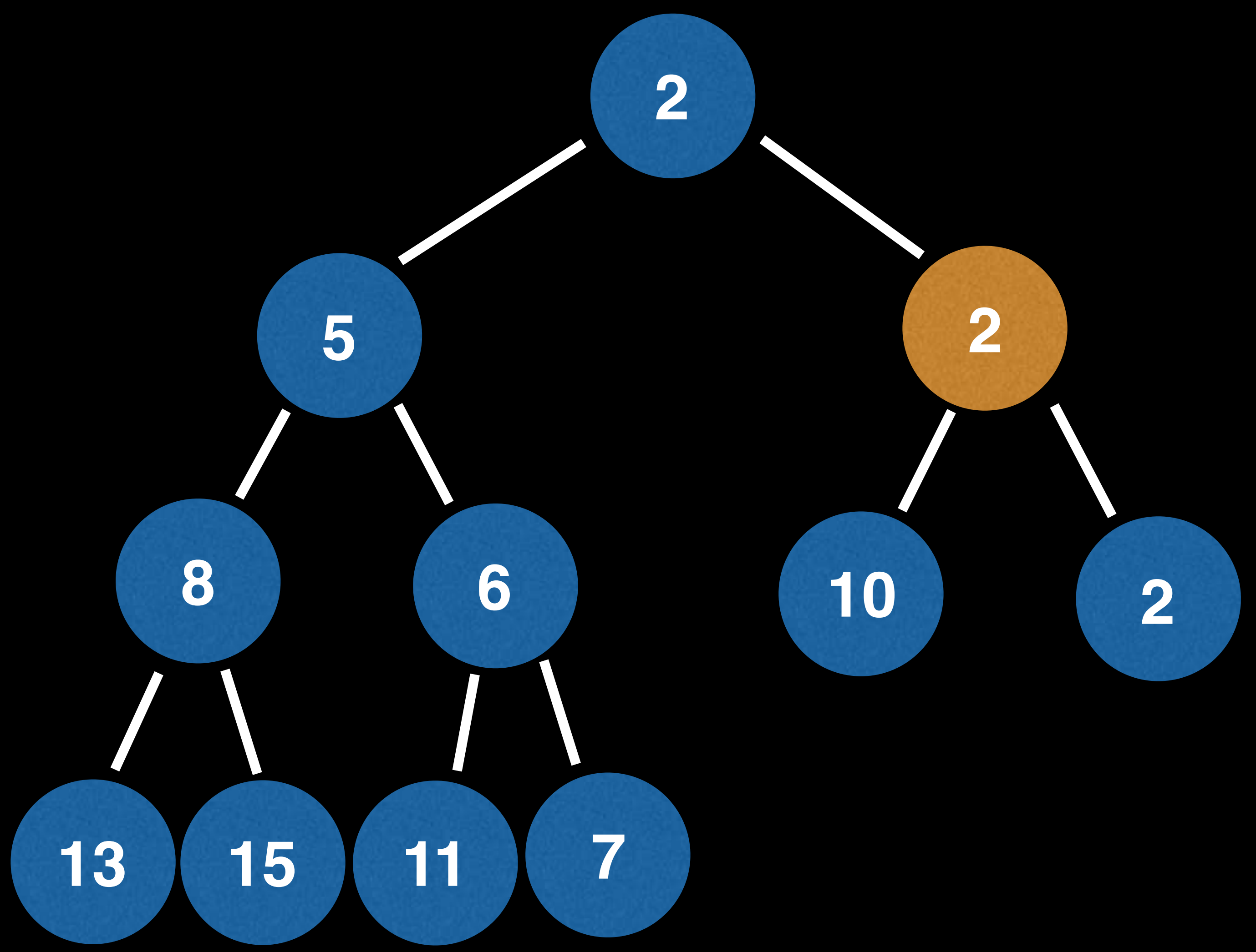
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

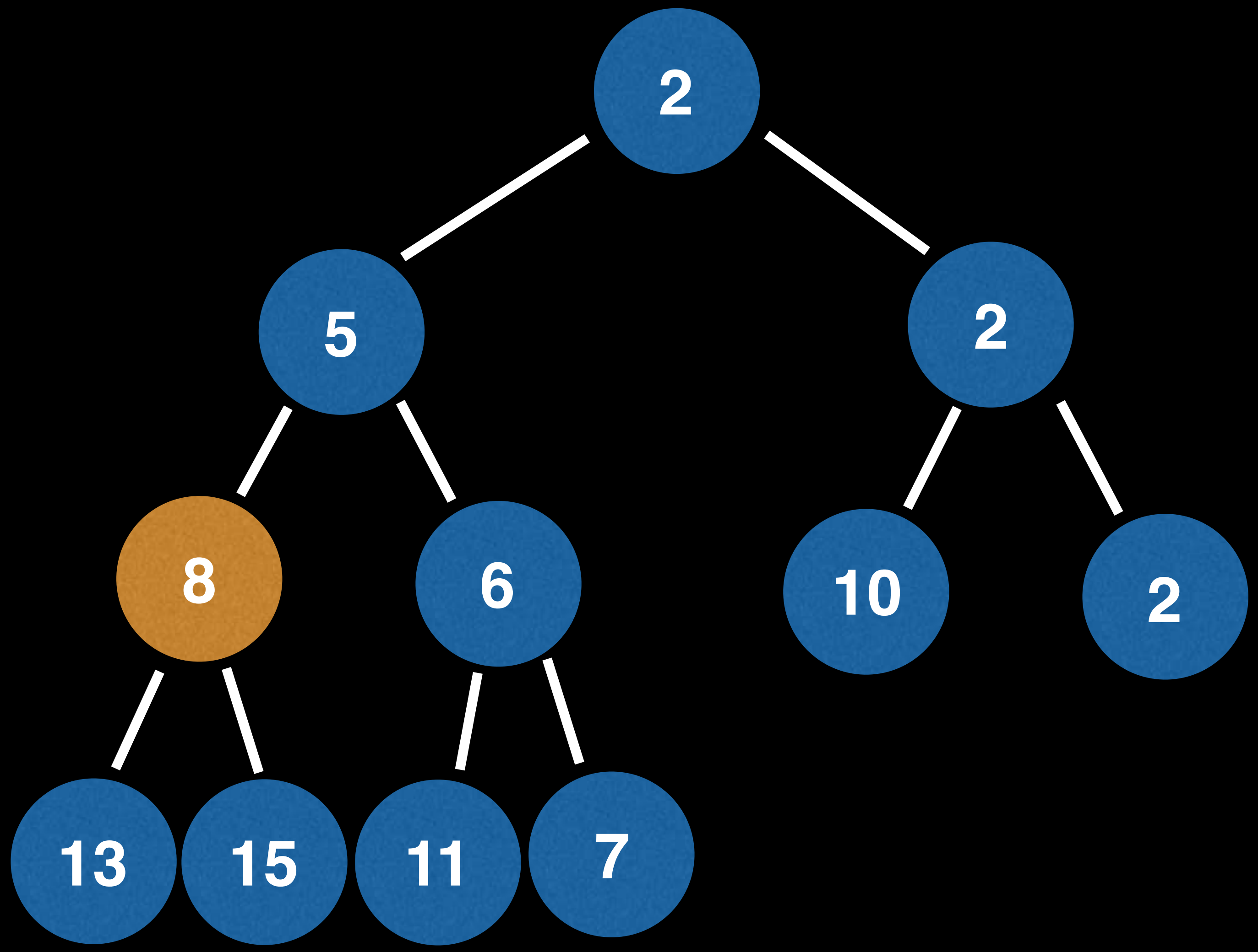
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

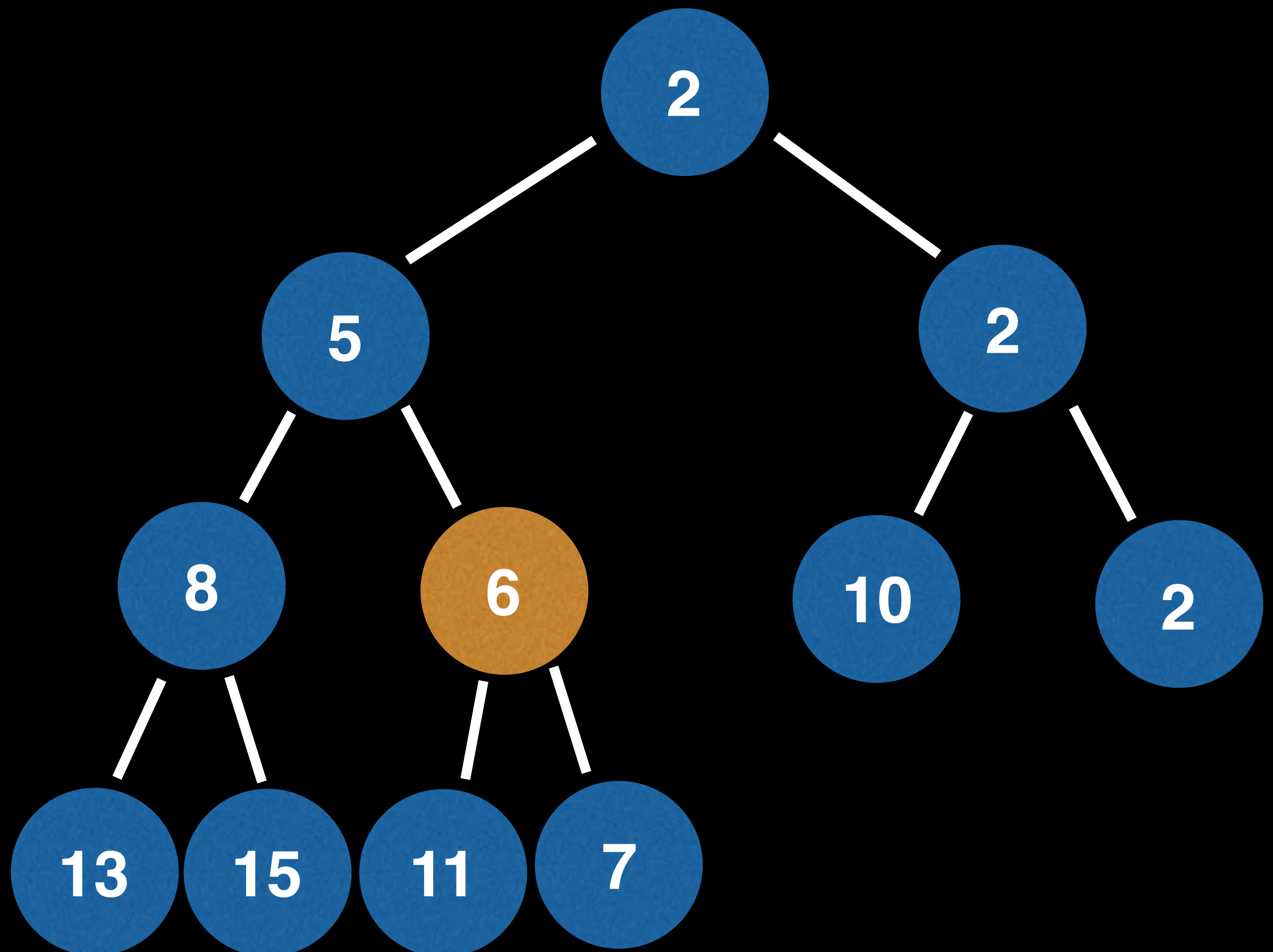
```
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)
```



Removing Elements From a Binary Heap

Instructions:

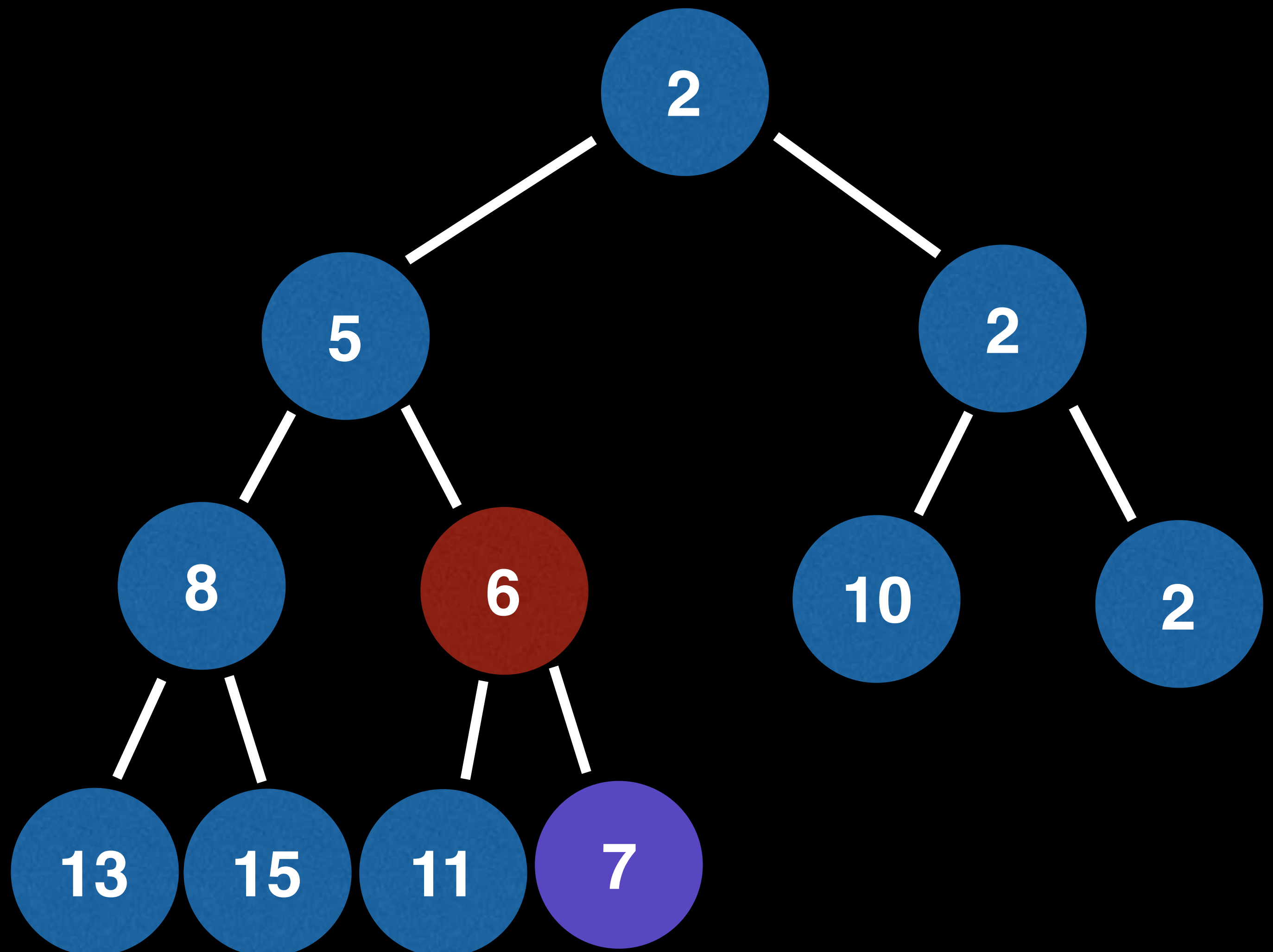
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

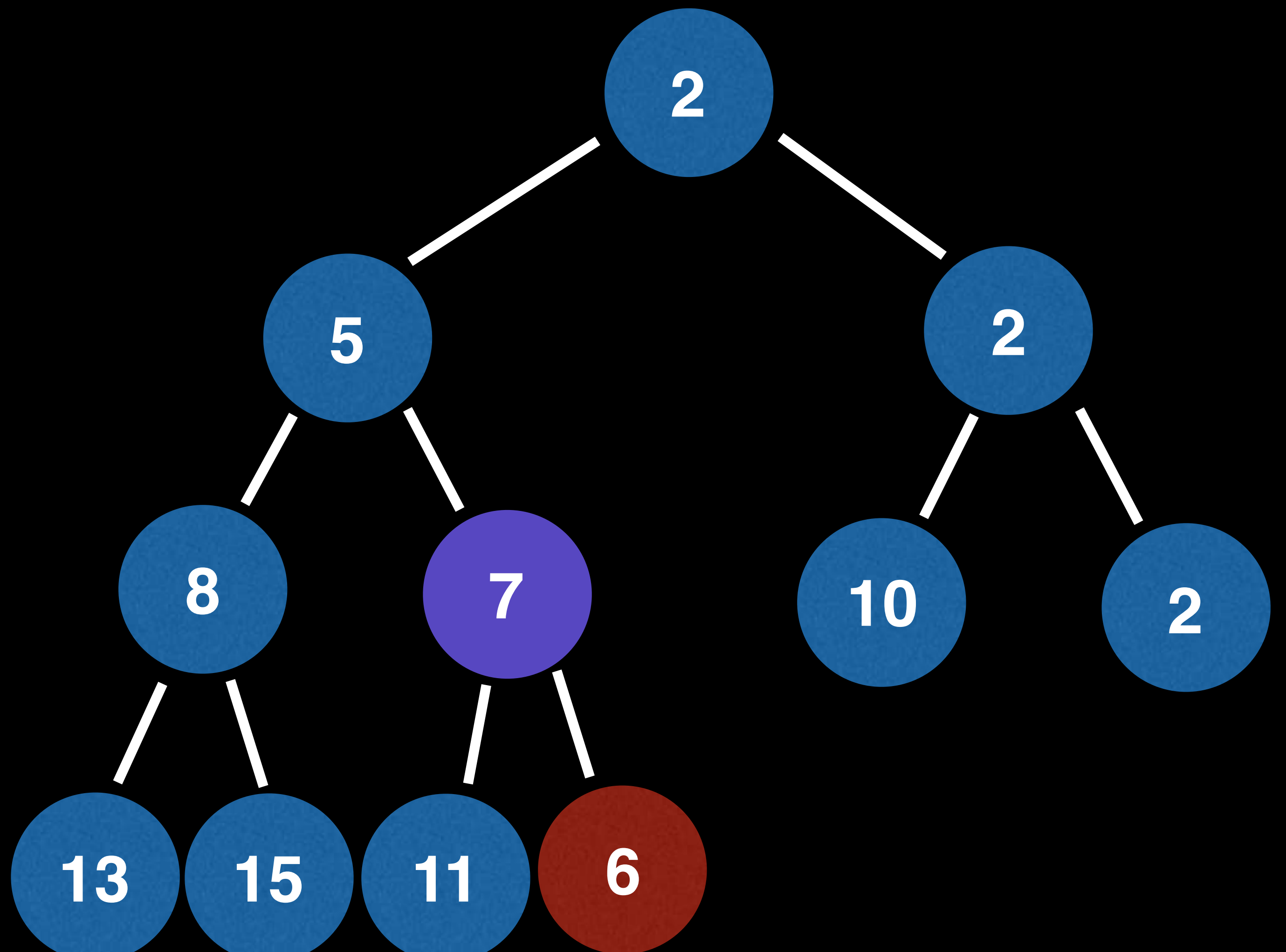
```
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)
```



Removing Elements From a Binary Heap

Instructions:

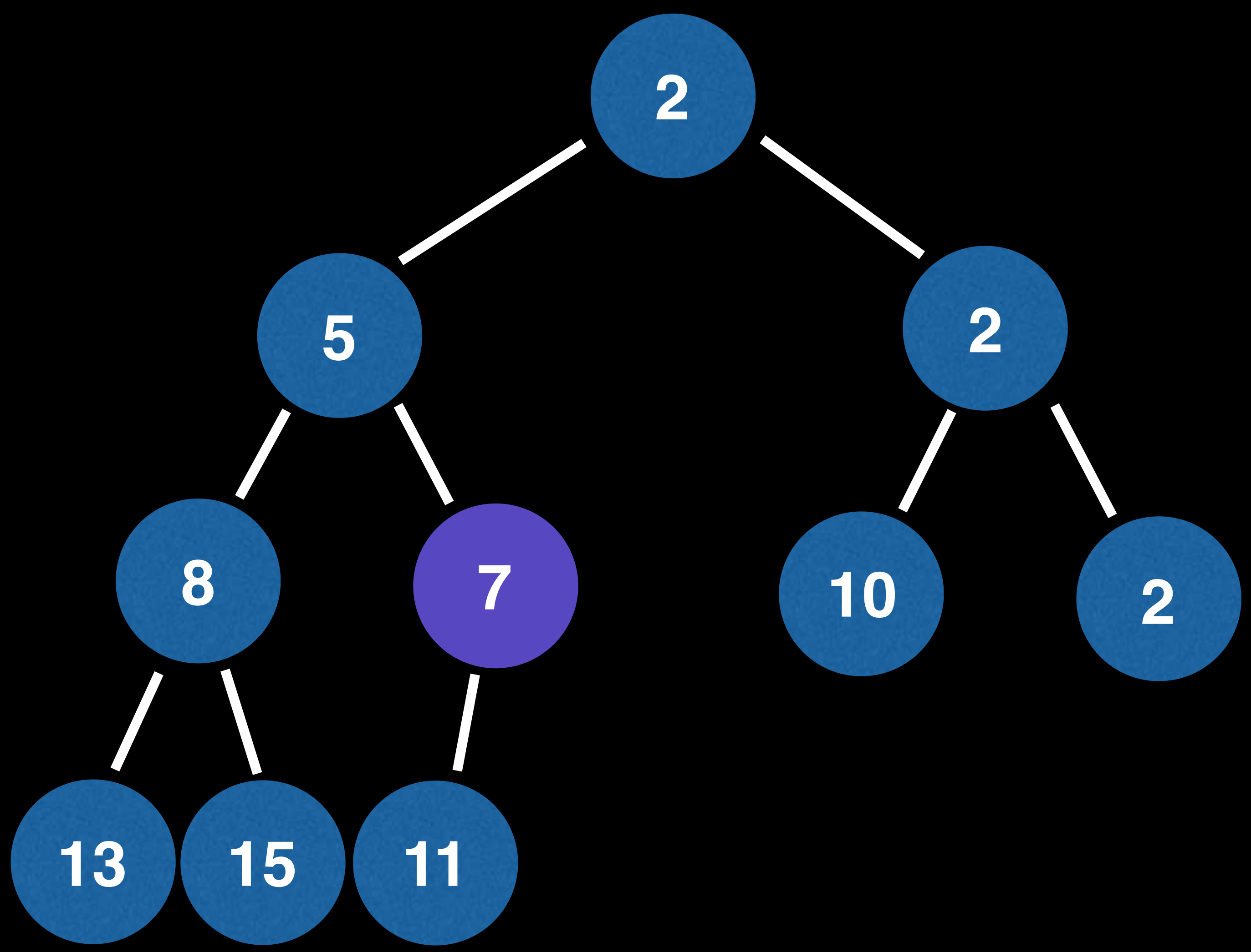
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)

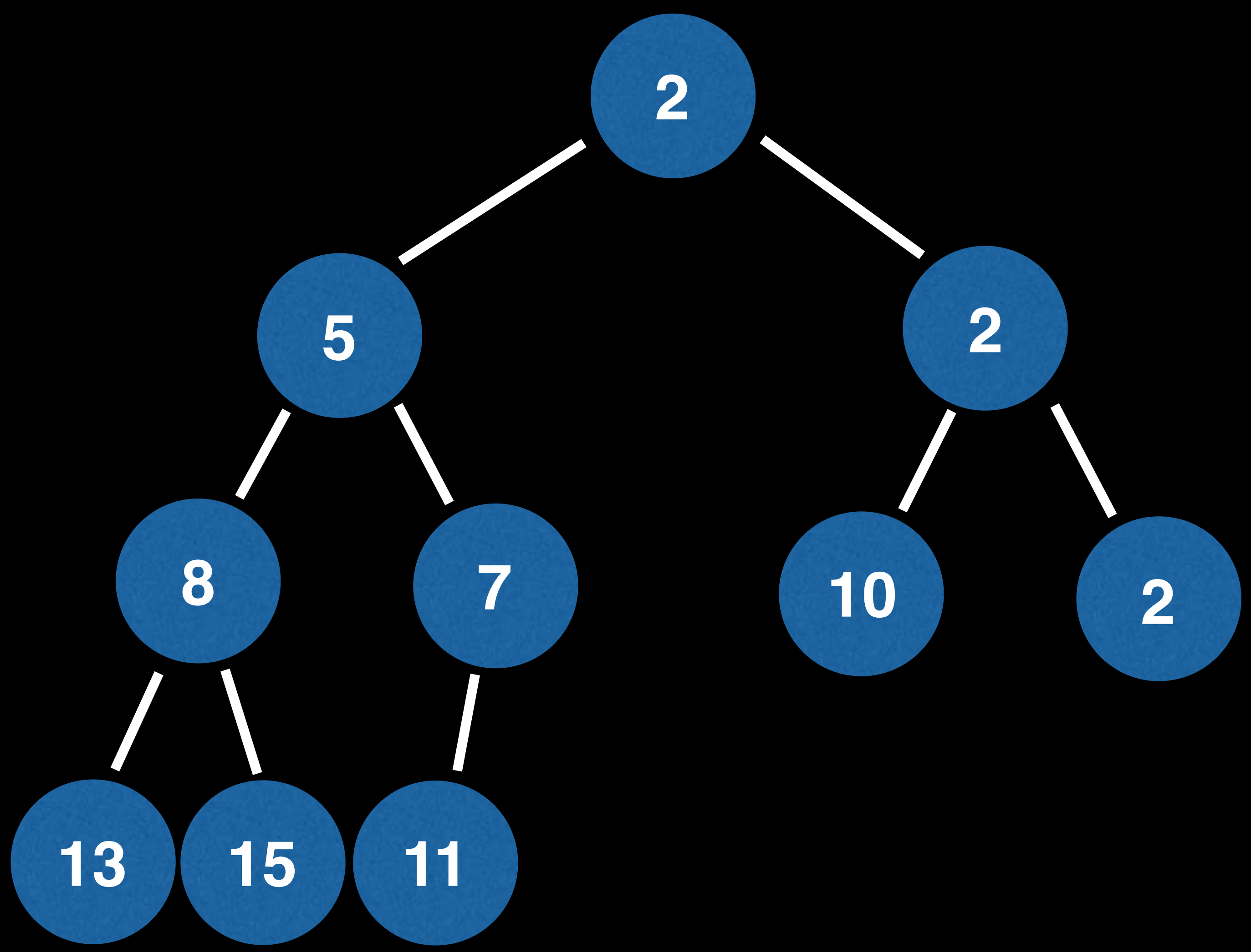


Do we bubble up or bubble down? Neither!
The heap invariant is already satisfied.

Removing Elements From a Binary Heap

Instructions:

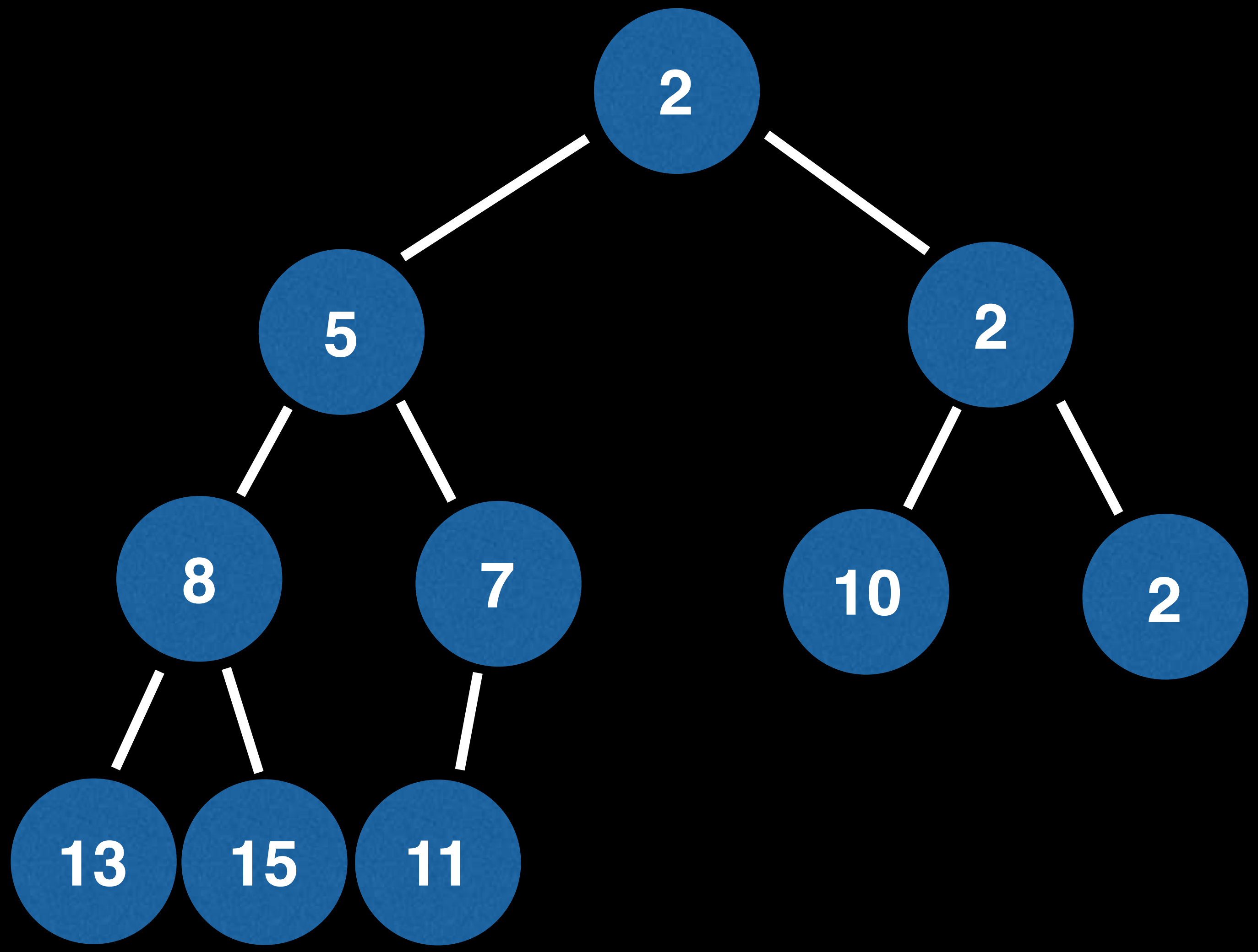
Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Removing Elements From a Binary Heap

Instructions:

Poll()
Remove(12)
Remove(3)
Poll()
Remove(6)



Polling – O(log(n))
Removing – O(n)

Removing Elements From Binary Heap in O(log(n))

The inefficiency of the removal algorithm comes from the fact that we have to perform a linear search to find out where an element is indexed at. What if instead we did a lookup using a **Hashtable** to find out where a node is indexed at?

A hashtable provides a constant time lookup and update for a mapping from a key (the node value) to a value (the index).

Removing Elements From Binary Heap in O(log(n))

Caveat: What if there are two or more nodes with the same value? What problems would that cause?

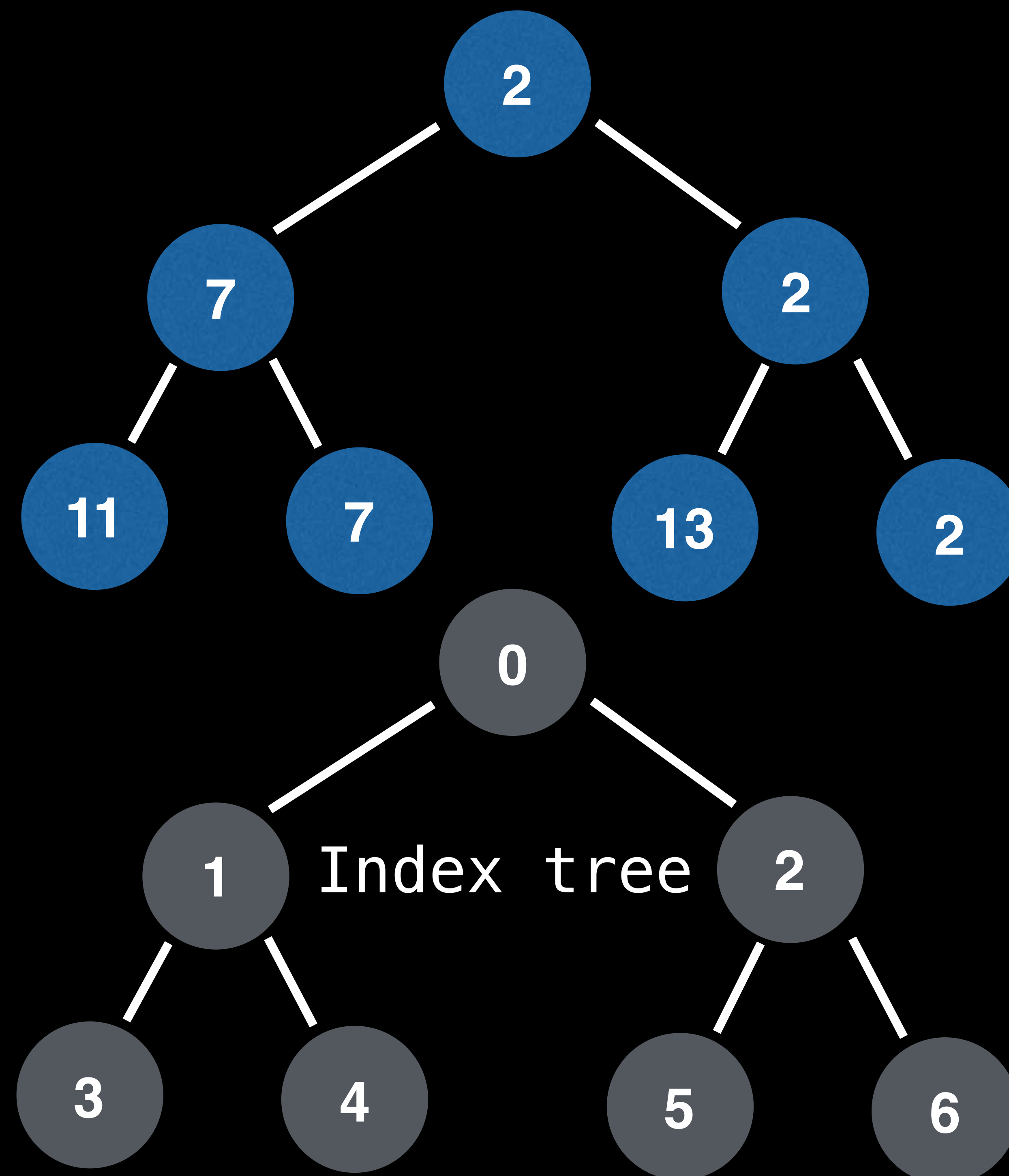
Removing Elements From Binary Heap in O(log(n))

Dealing with the multiple value problem:

Instead of mapping one value to one position we will map one value to multiple positions. We can maintain a **Set** or **Tree Set** of indexes for which a particular node value (key) maps to.

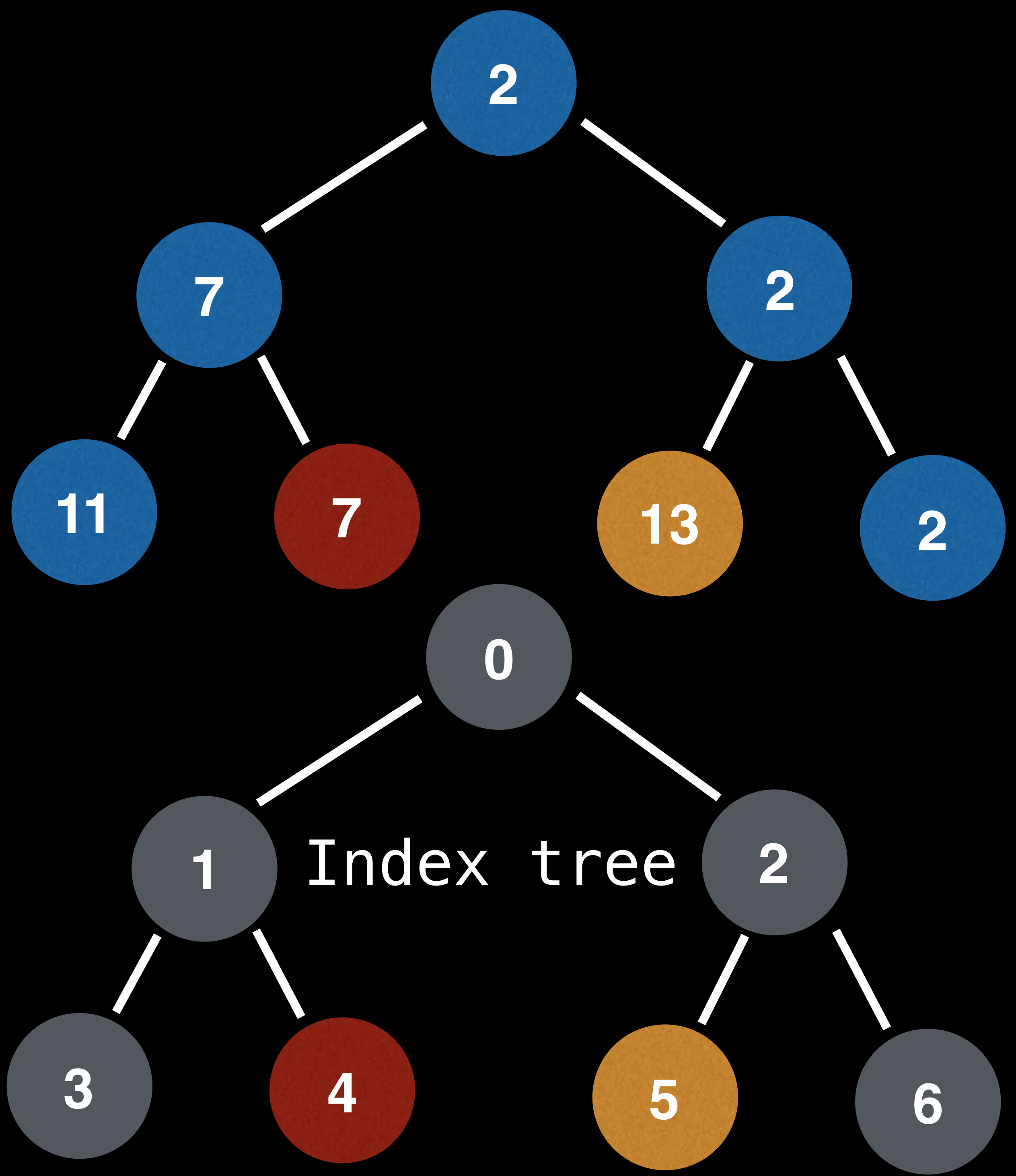
Removing Elements From Binary Heap in O(log(n))

Node Value (Key)	Postion(s) (Value)
2	0, 2, 6
7	1, 4
11	3
13	5



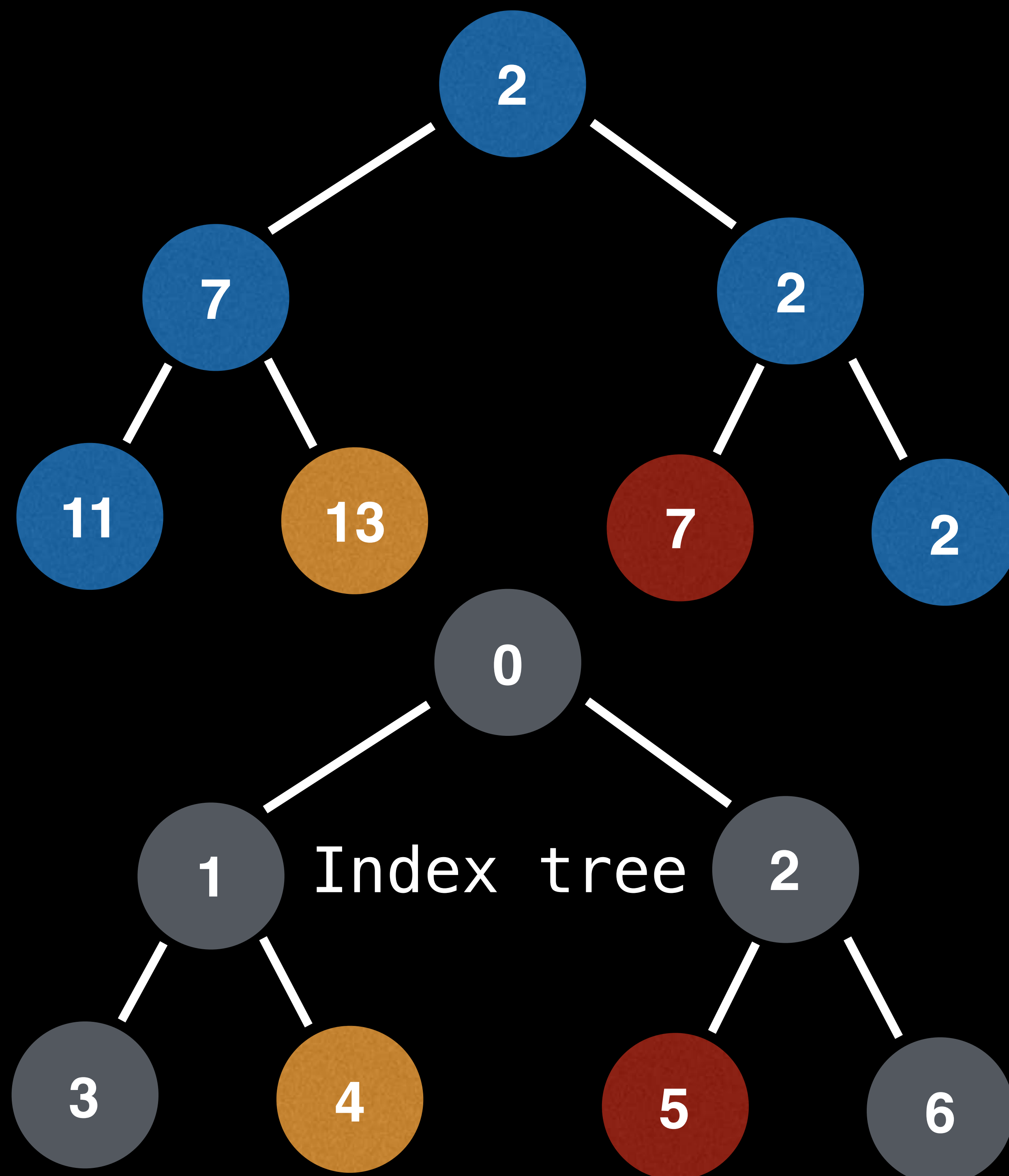
Removing Elements From Binary Heap in O(log(n))

Node Value (Key)	Postion(s) (Value)
2	0, 2, 6
7	1, 4
11	3
13	5



Removing Elements From Binary Heap in O(log(n))

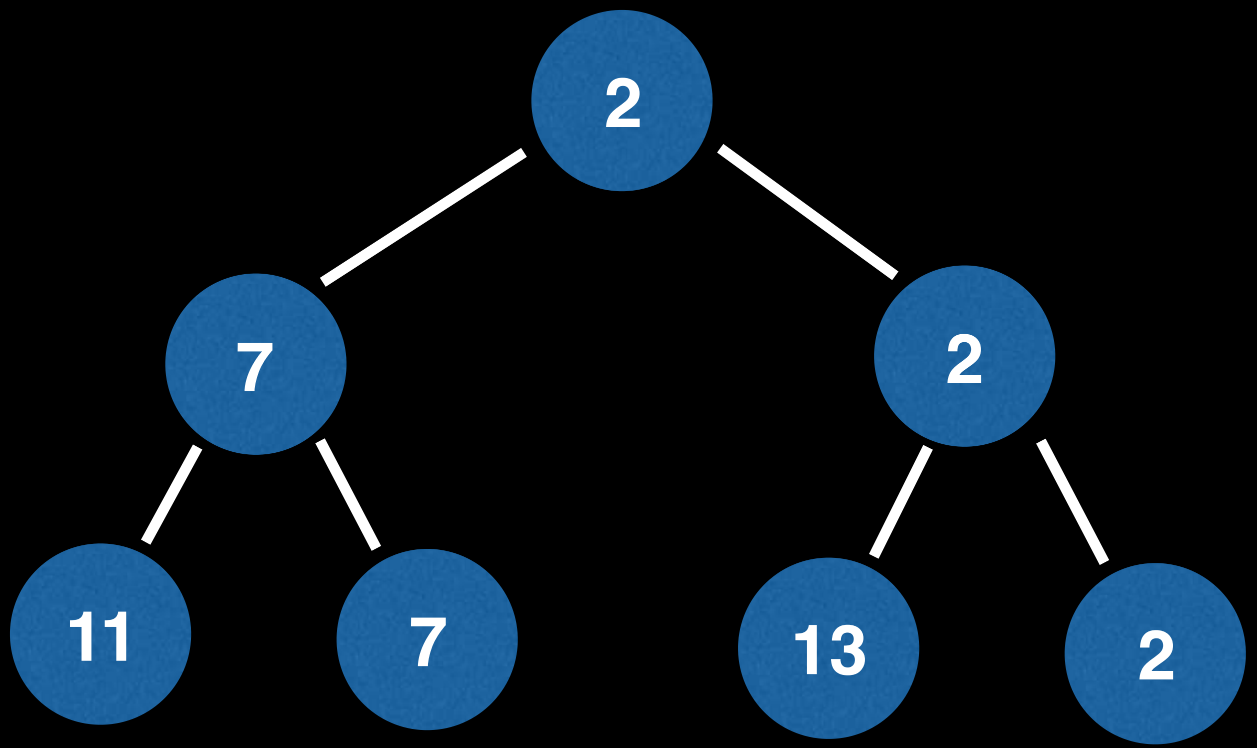
Node Value (Key)	Postion(s) (Value)
2	0, 2, 6
7	1, 5
11	3
13	4



Removing Elements From Binary Heap in O(log(n))

Question: If we want to remove a repeated node in our heap, which node do we remove and does it matter which one we pick?

Node Value	Postion(s)
2	0, 2, 6
7	1, 4
11	3
13	5

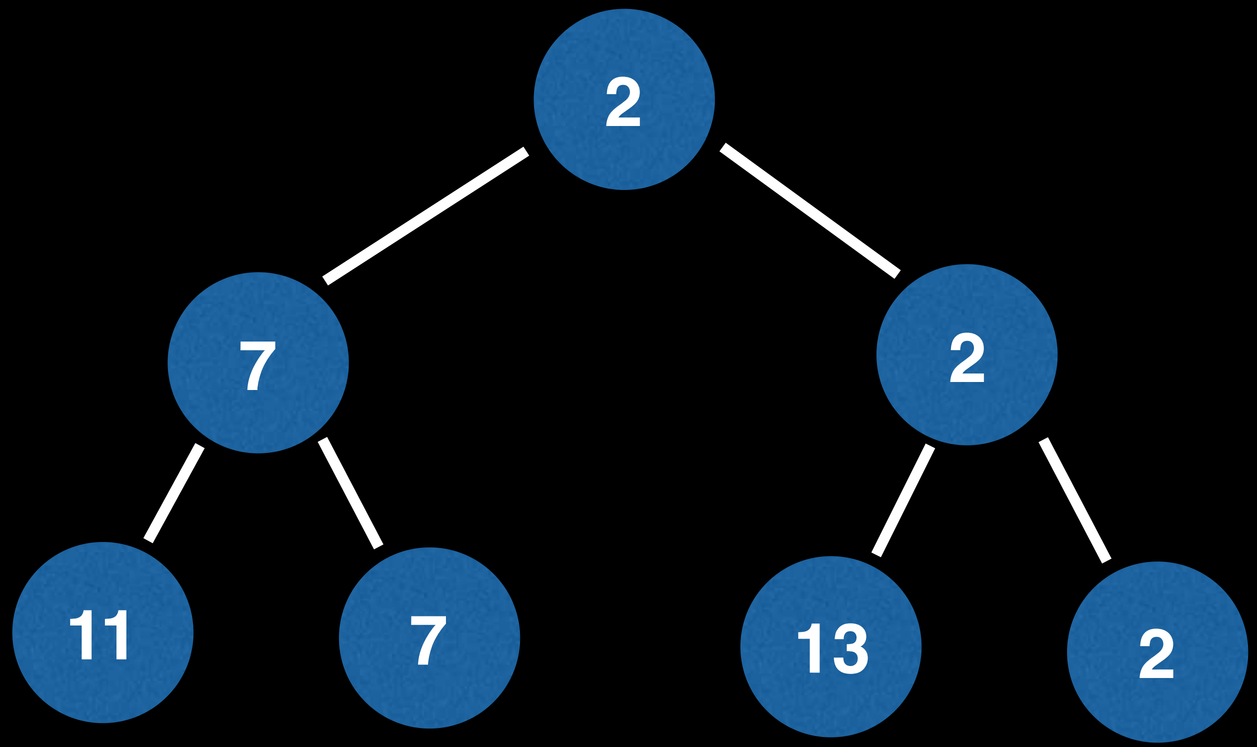


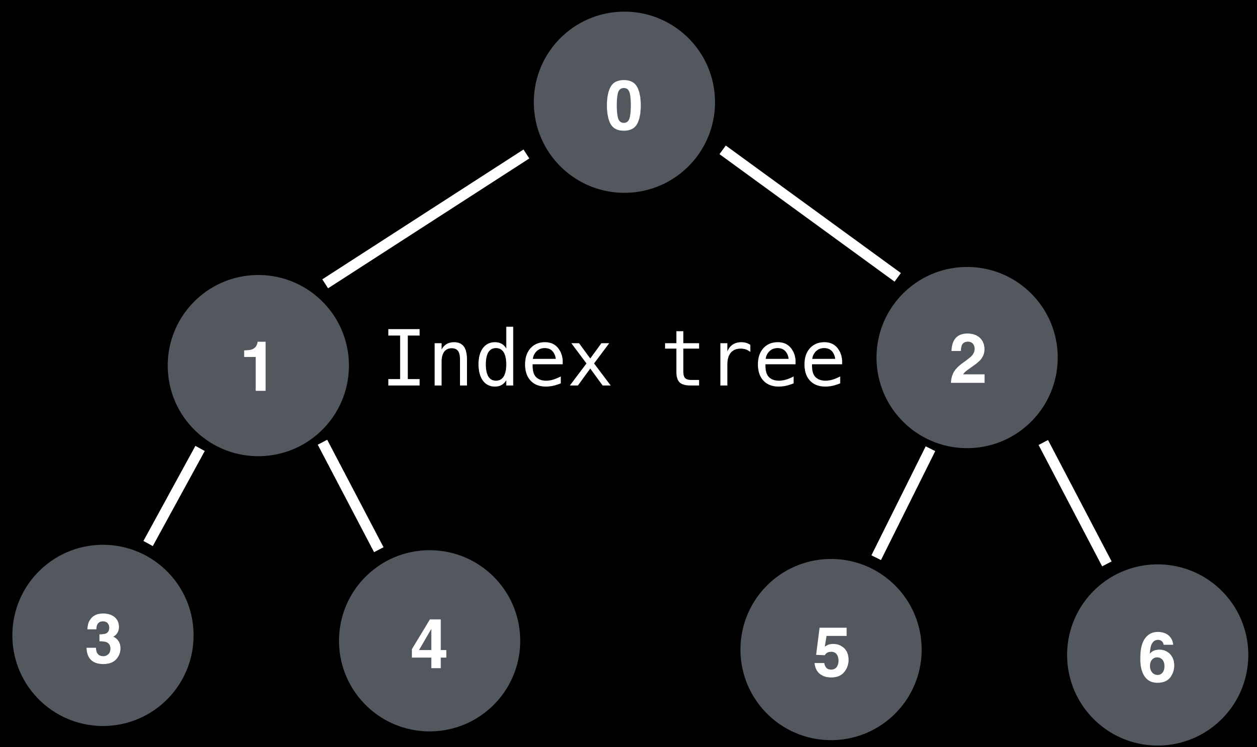
Removing Elements From Binary Heap in O(log(n))

Question: If we want to remove a repeated node in our heap, which node do we remove and does it matter which one we pick?

Answer: No it doesn’t matter which node we remove as long as we satisfy the heap invariant in the end.

Node Value	Postion(s)
2	0, 2, 6
7	1, 4
11	3
13	5

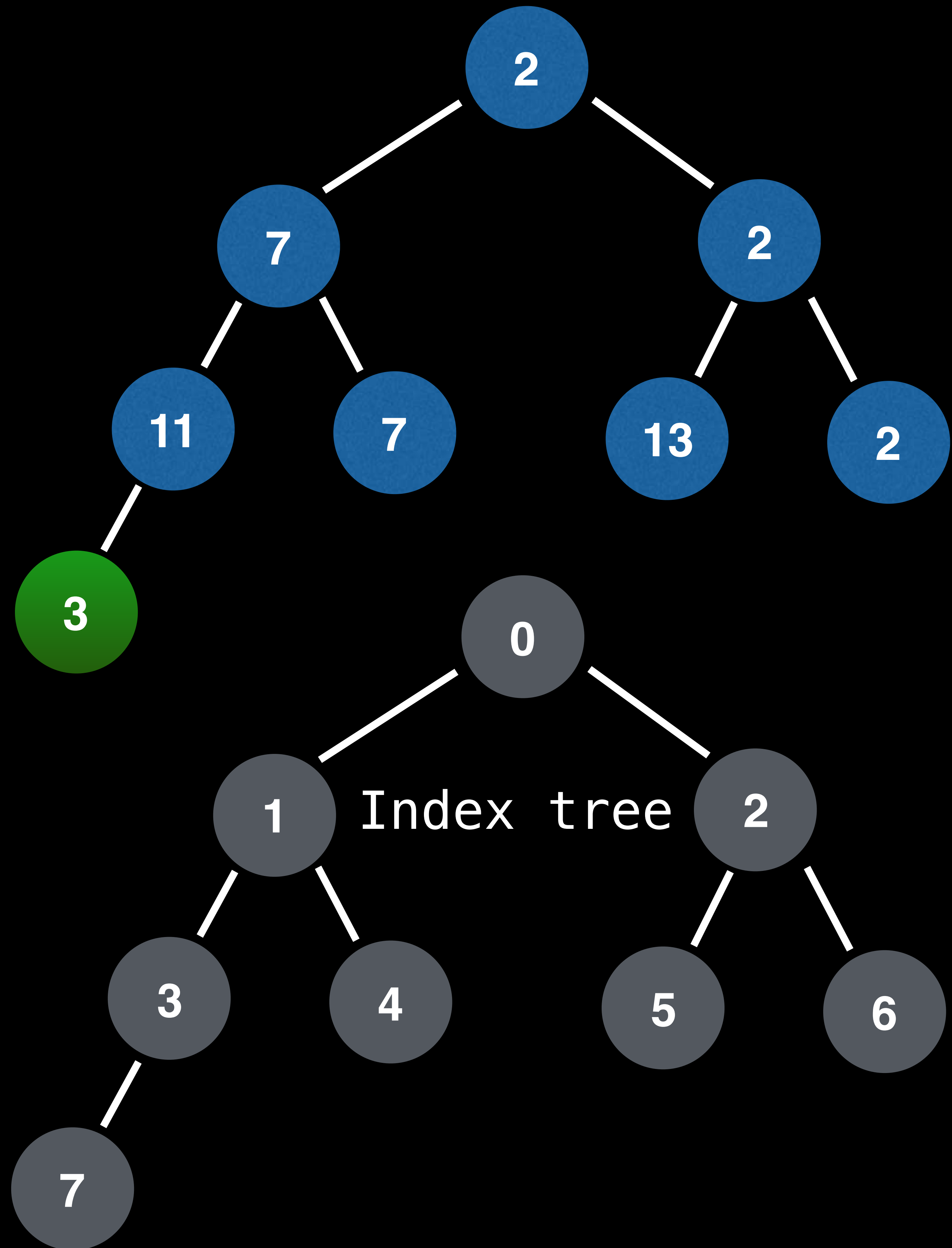




<u>Instructions</u>:

```
insert(3)
remove(2)
poll()
```

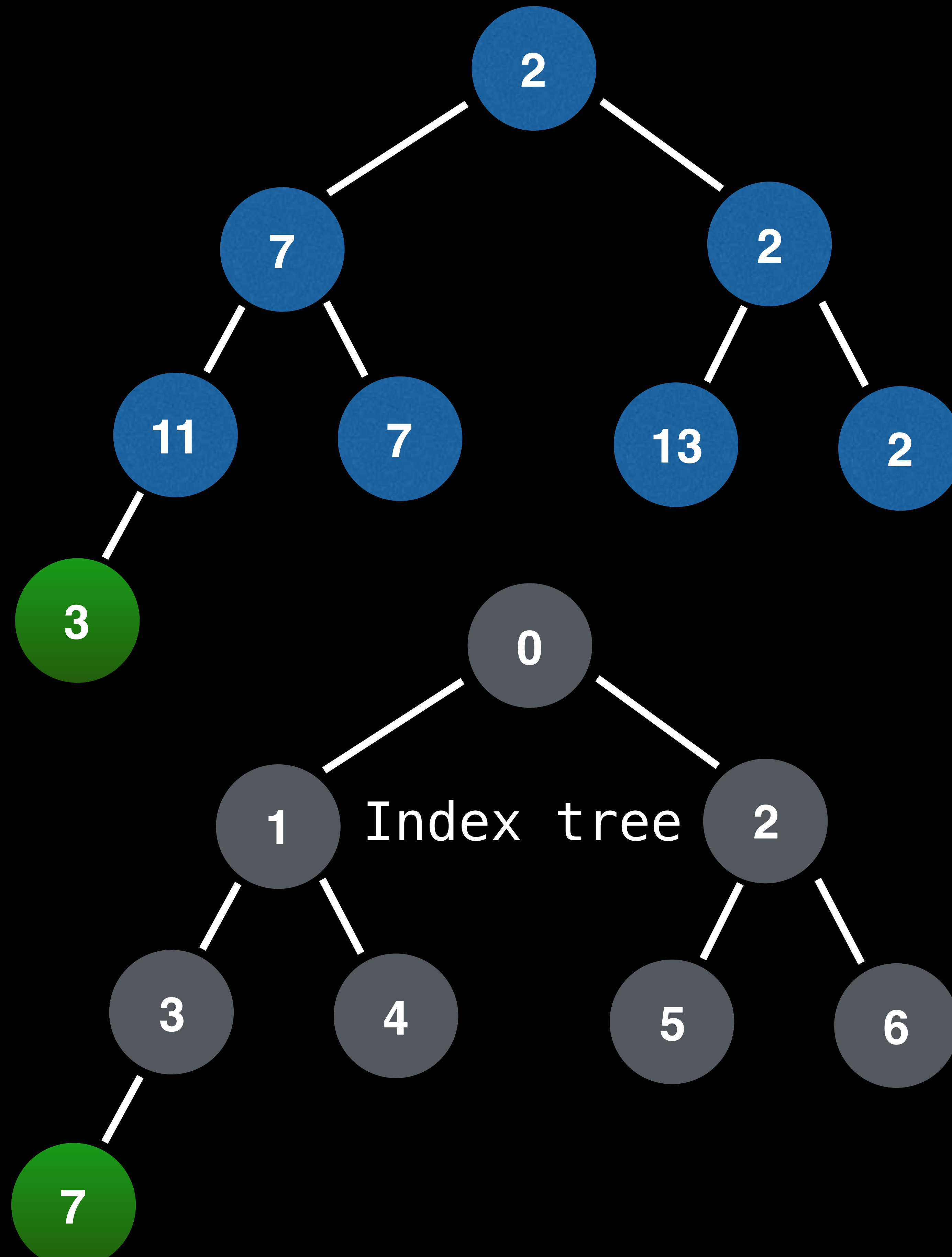
Node Value	Postion(s)
2	0, 2, 6
7	1, 4
11	3
13	5
3	7



Instructions:

```
insert(3)
remove(2)
poll()
```

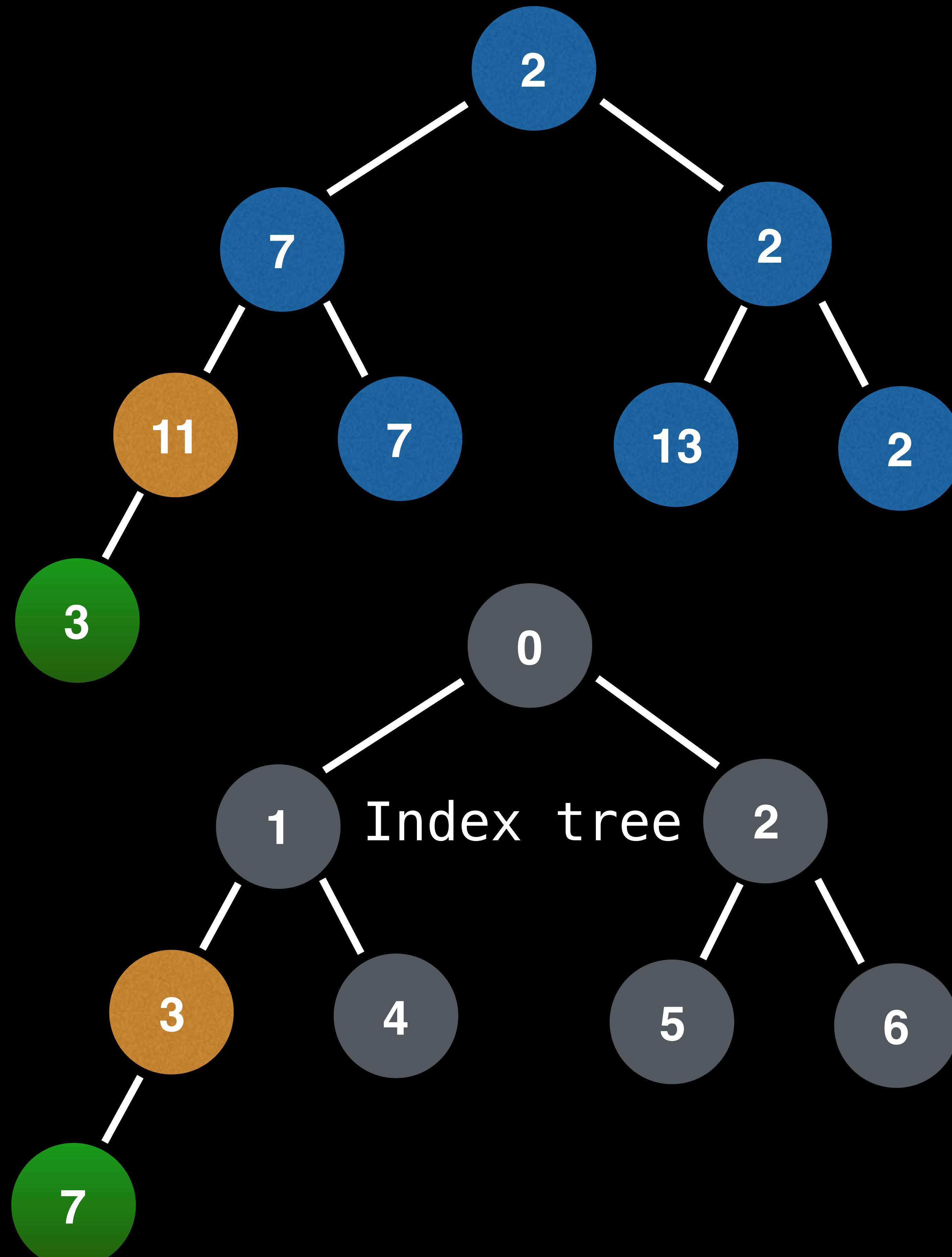
Node Value	Postion(s)
2	0, 2, 6
7	1, 4
11	3
13	5
3	7



Instructions:

- insert(3)
- remove(2)
- poll()

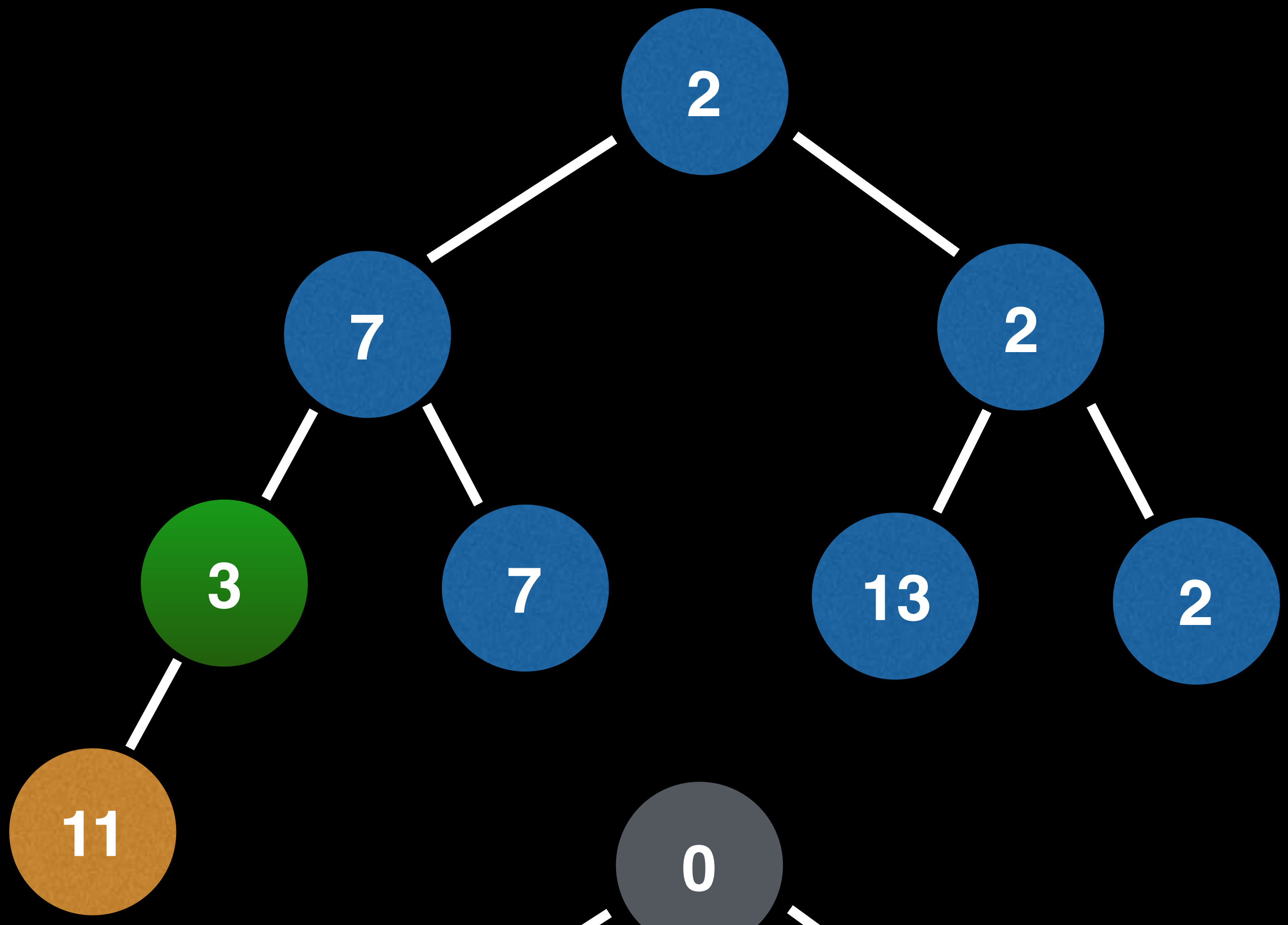
Node Value	Postion(s)
2	0, 2, 6
7	1, 4
11	3
13	5
3	7

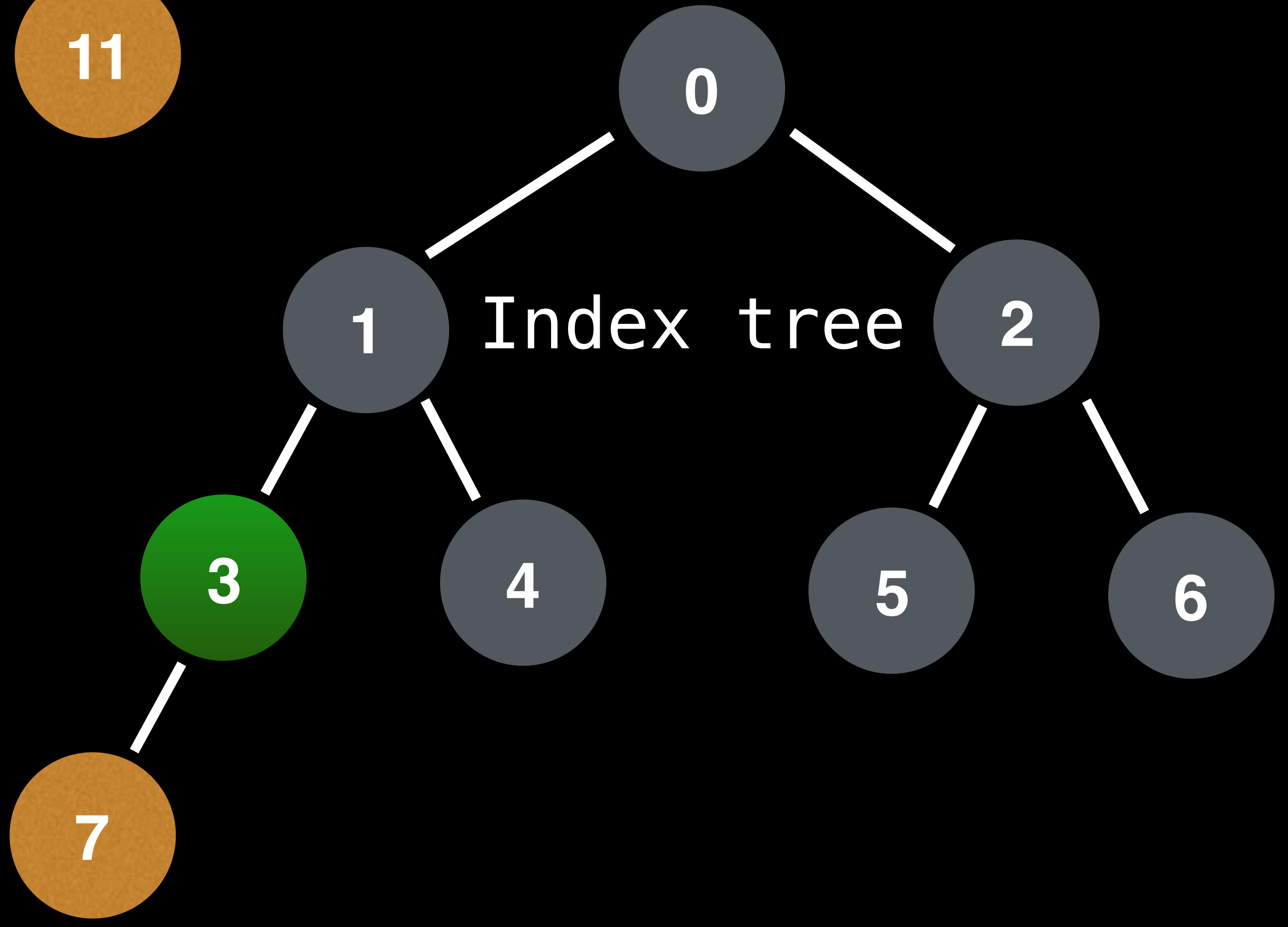


Instructions:

```
insert(3)
remove(2)
poll()
```

Node Value	Postion(s)
2	0, 2, 6
7	1, 4
11	7
13	5
3	3

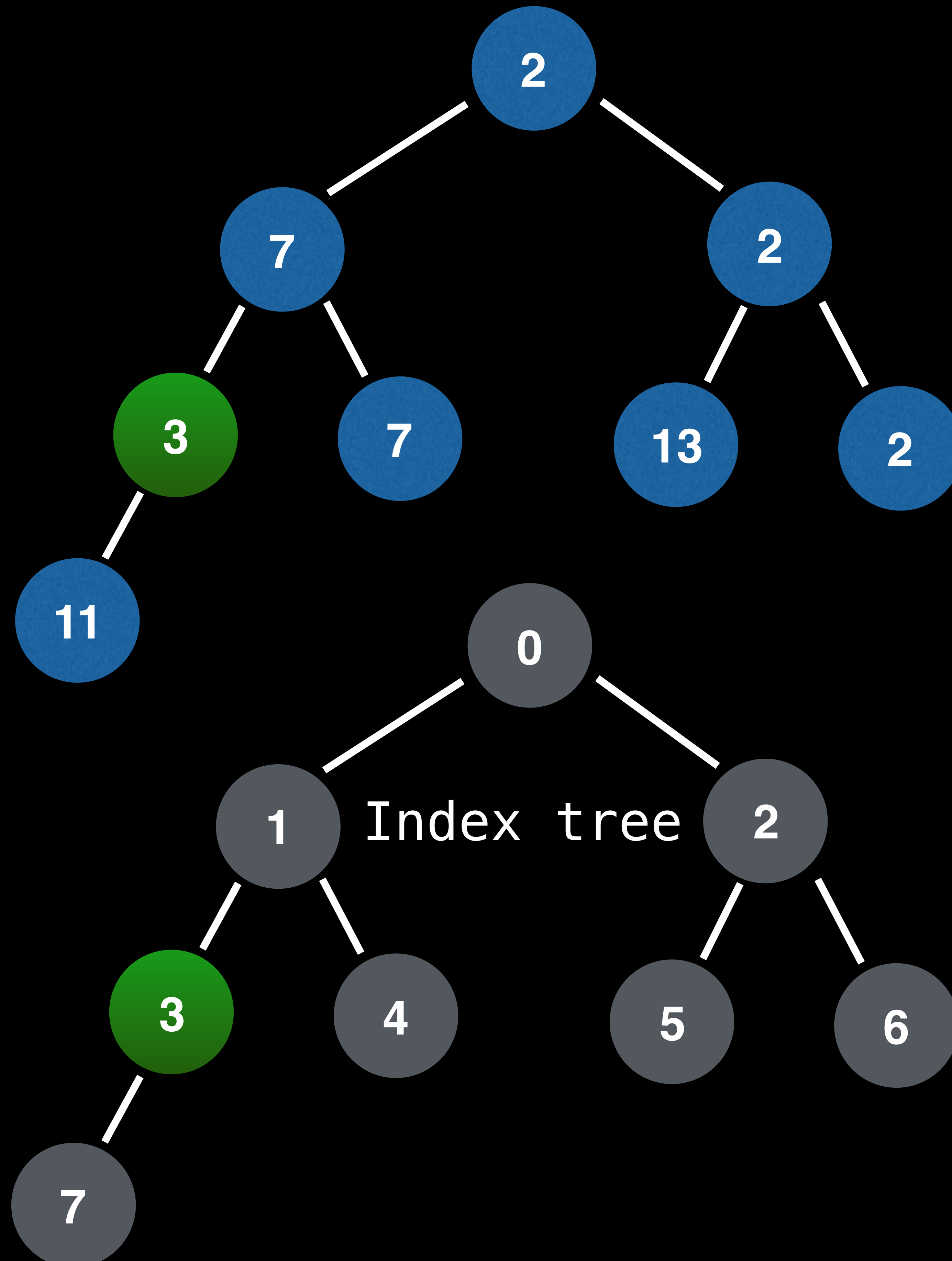




Instructions:

- insert(3)
- remove(2)
- poll()

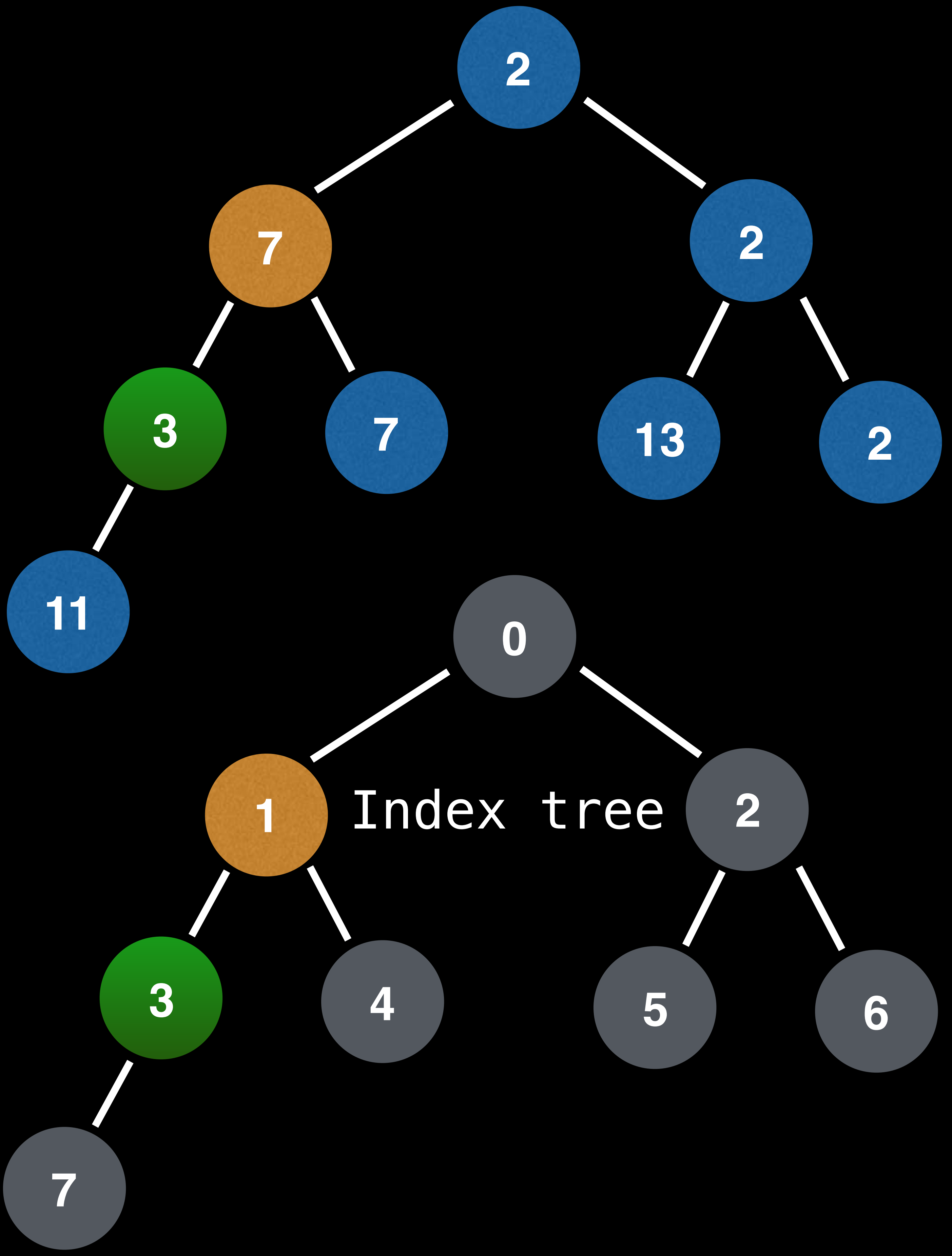
Node Value	Postion(s)
2	0, 2, 6
7	1, 4
11	7
13	5
3	3



Instructions:

```
insert(3)
remove(2)
poll()
```

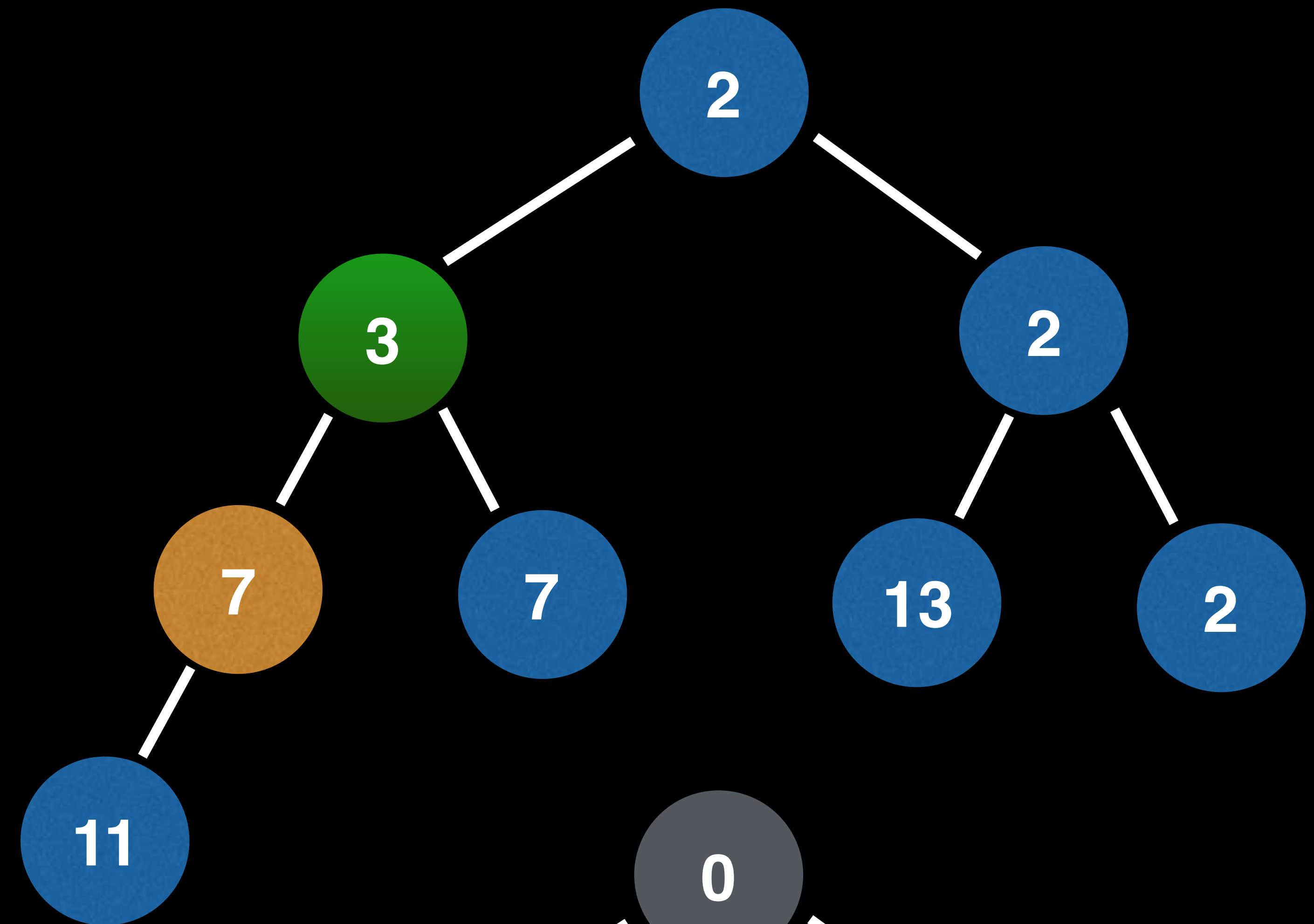
Node Value	Postion(s)
2	0, 2, 6
7	1, 4
11	7
13	5
3	3

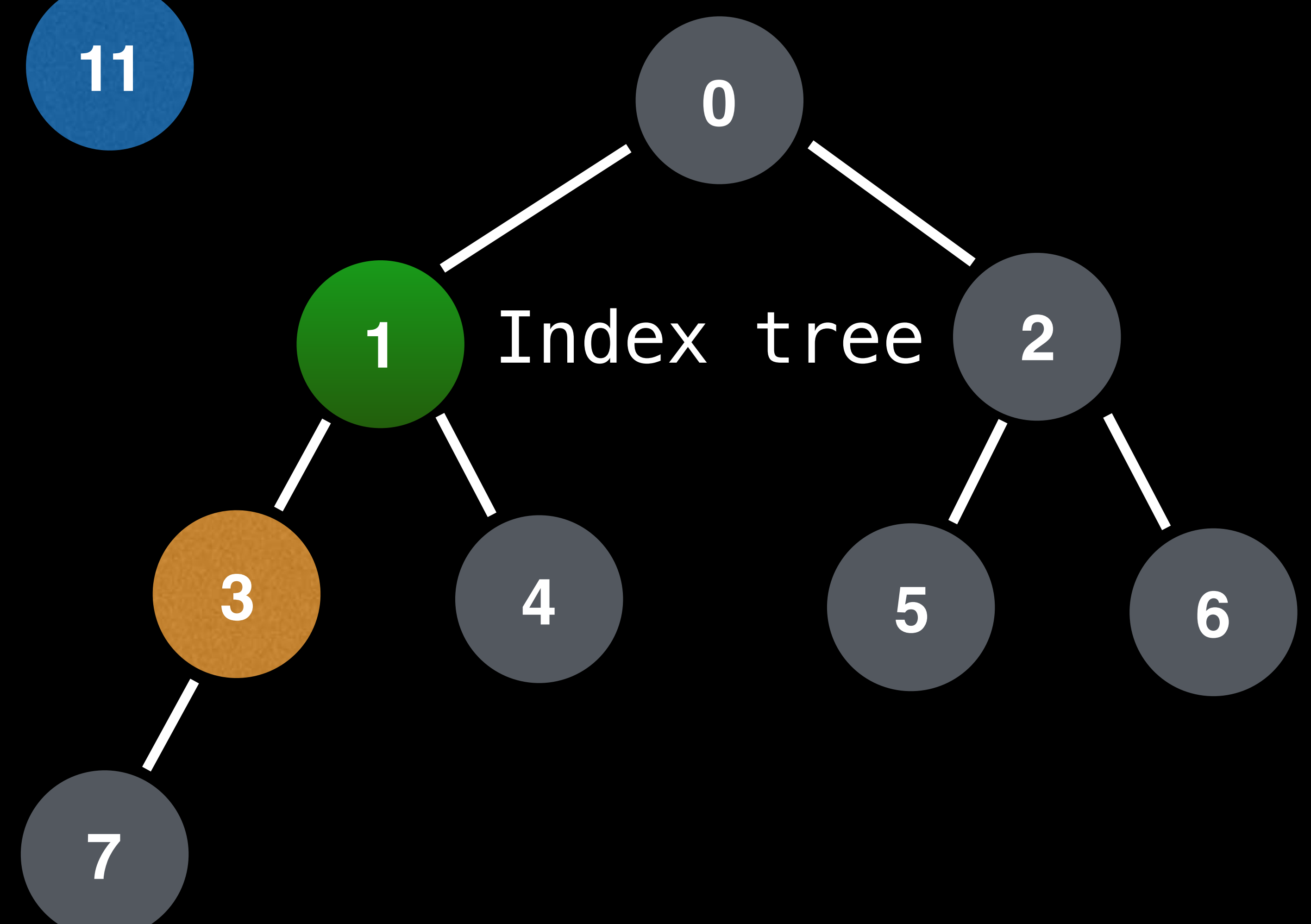


Instructions:

```
insert(3)
remove(2)
poll()
```

Node Value	Postion(s)
2	0, 2, 6
7	3, 4
11	7
13	5
3	1

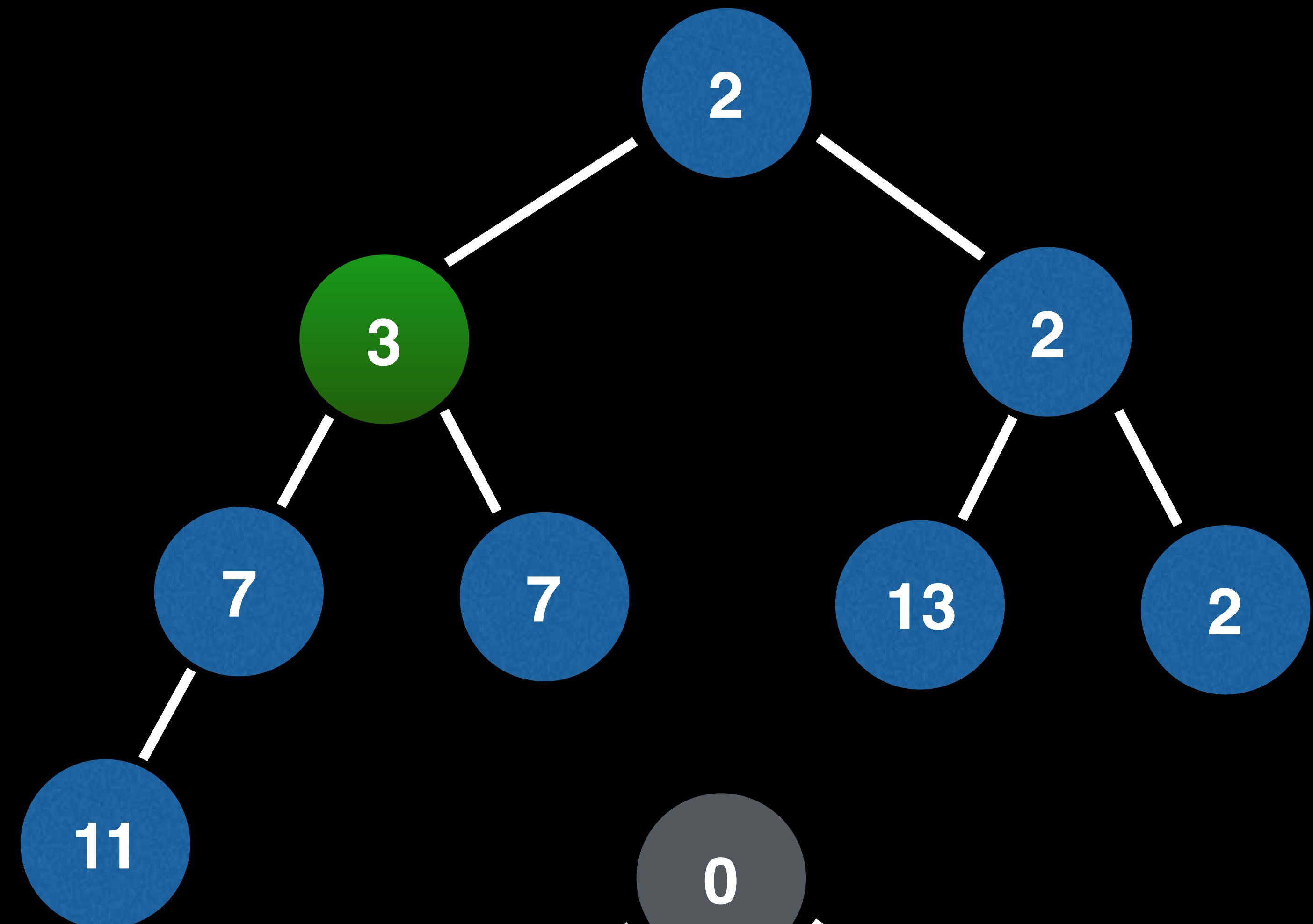




Instructions:

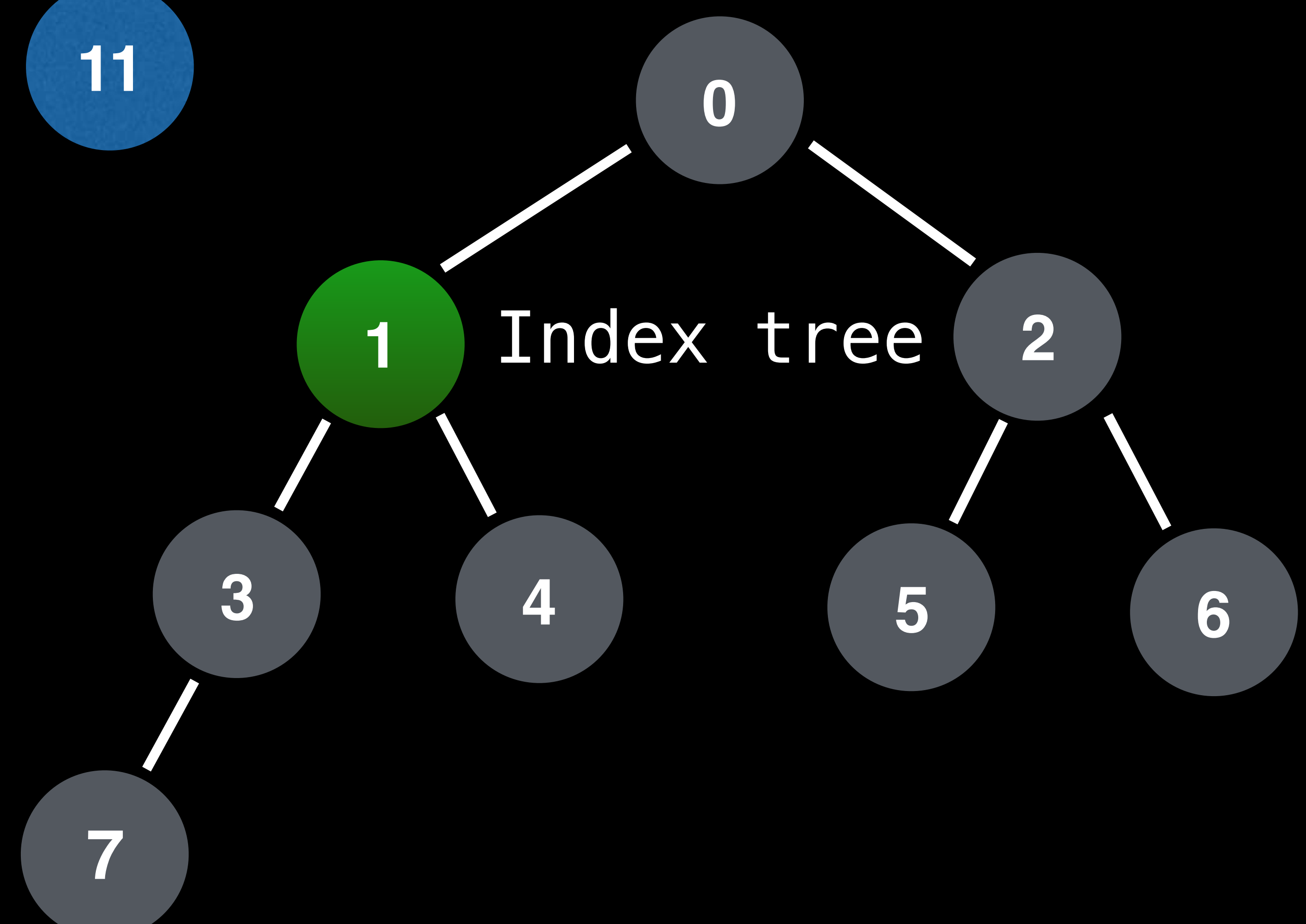
insert(3)
remove(2)
poll()

Node Value	Postion(s)
2	0, 2, 6
7	3, 4
11	7
13	5
3	1



Instructions:

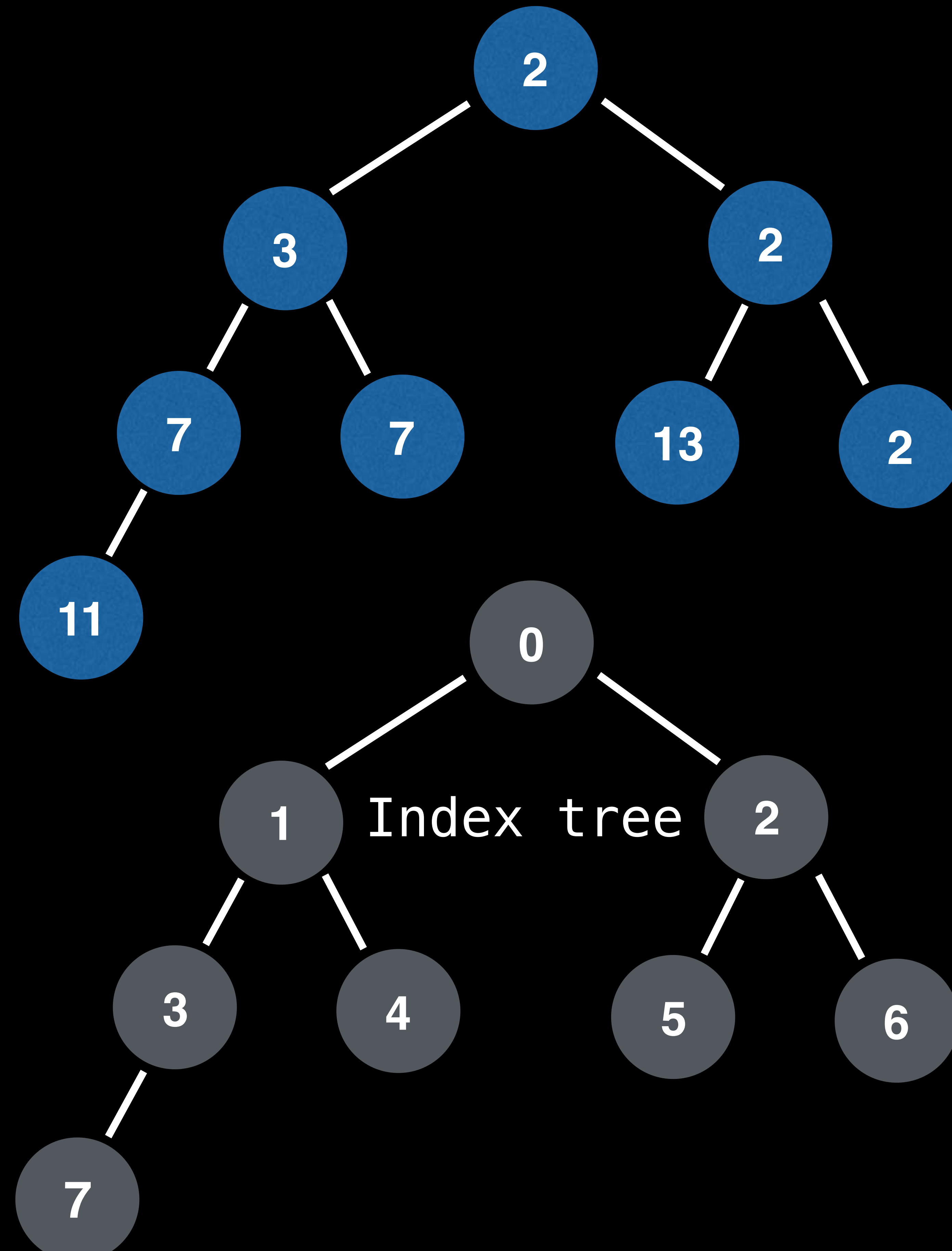
```
insert(3)
remove(2)
poll()
```



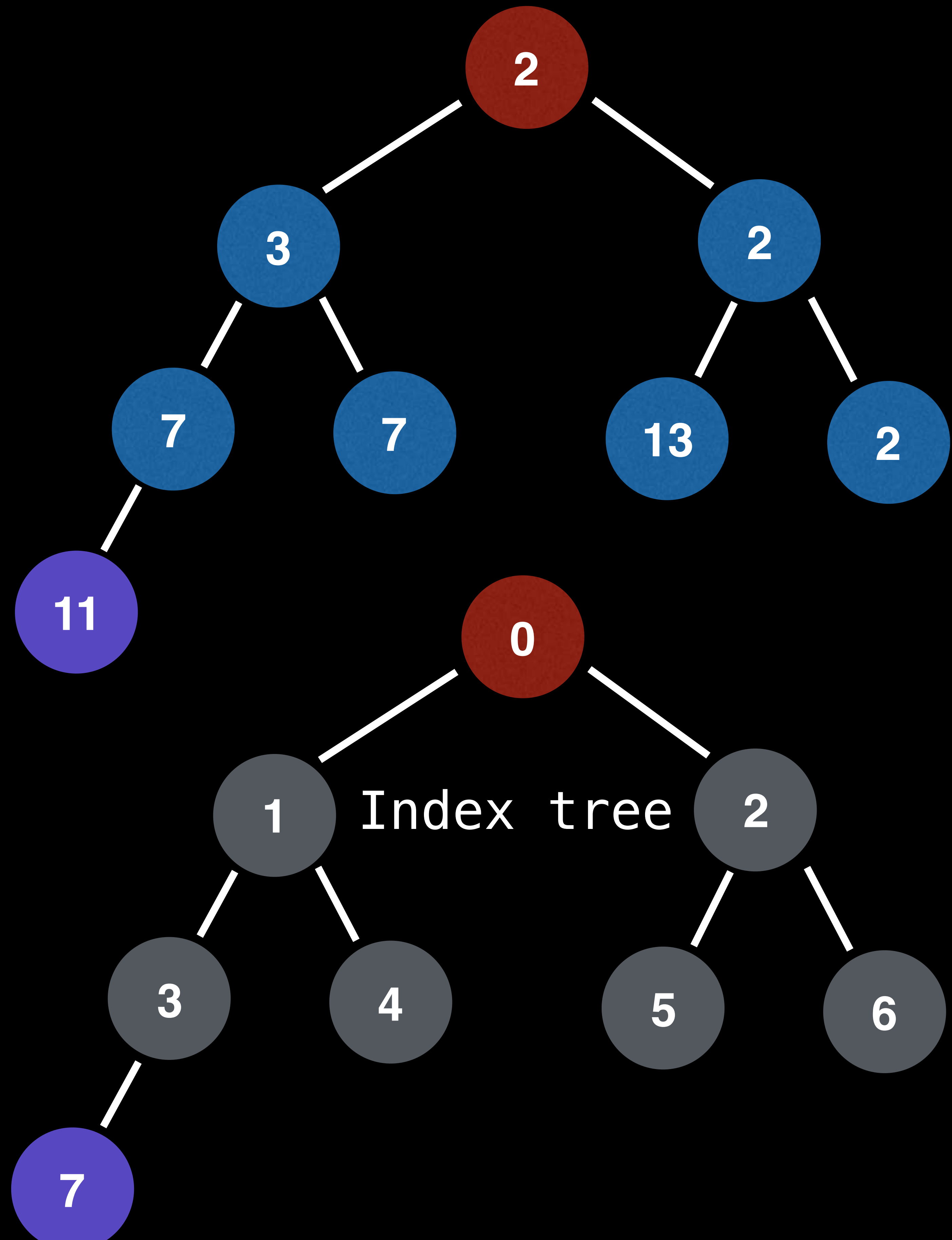
Node Value	Postion(s)
2	0, 2, 6
7	3, 4
11	7
13	5
3	1

Instructions:

```
insert(3)
remove(2)
poll()
```



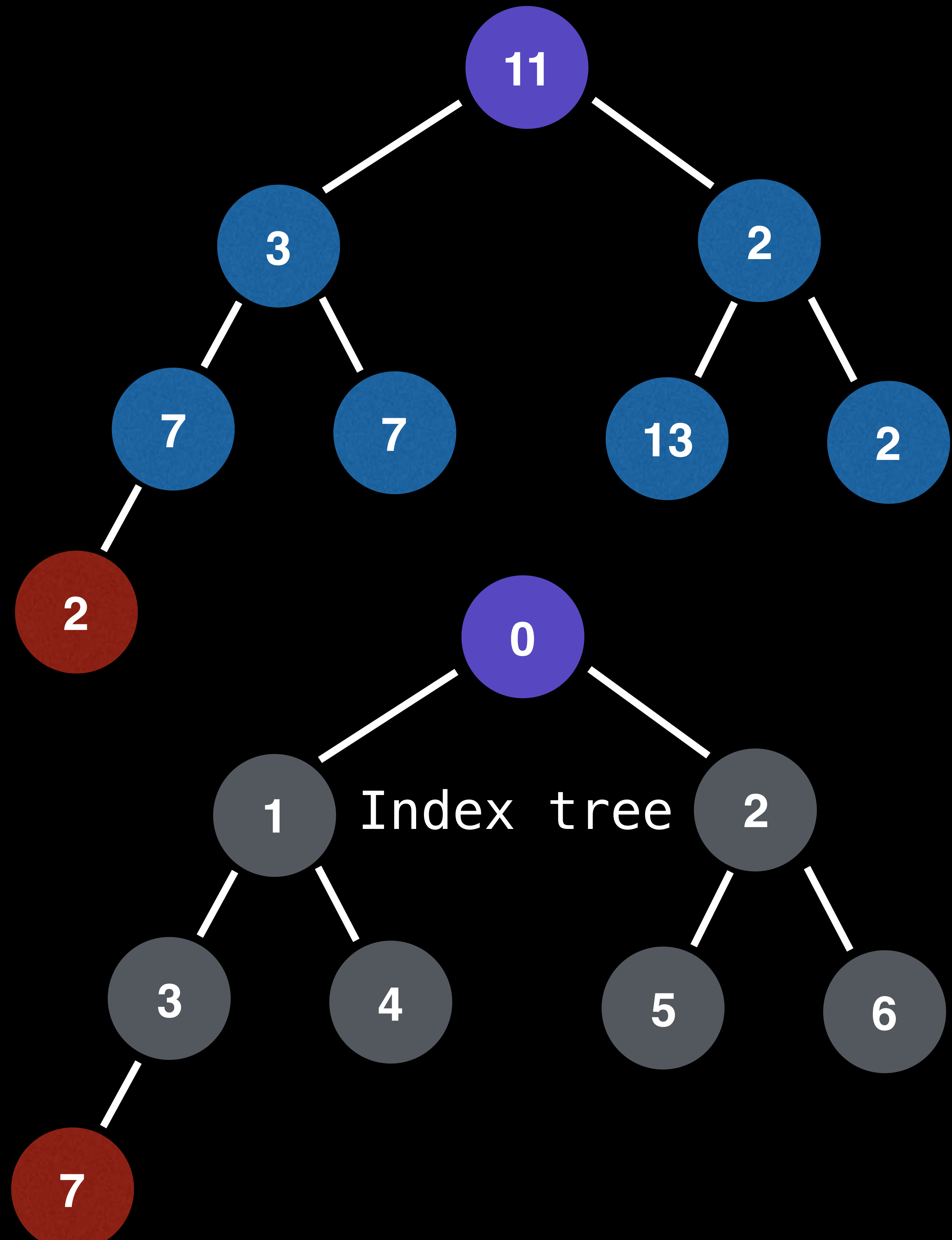
Node Value	Postion(s)
2	0, 2, 6
7	3, 4
11	7
13	5
3	1



Instructions:

```
  insert(3)
→ remove(2)
  poll()
```

Node Value	Postion(s)
2	7, 2, 6
7	3, 4
11	0
13	5
3	1



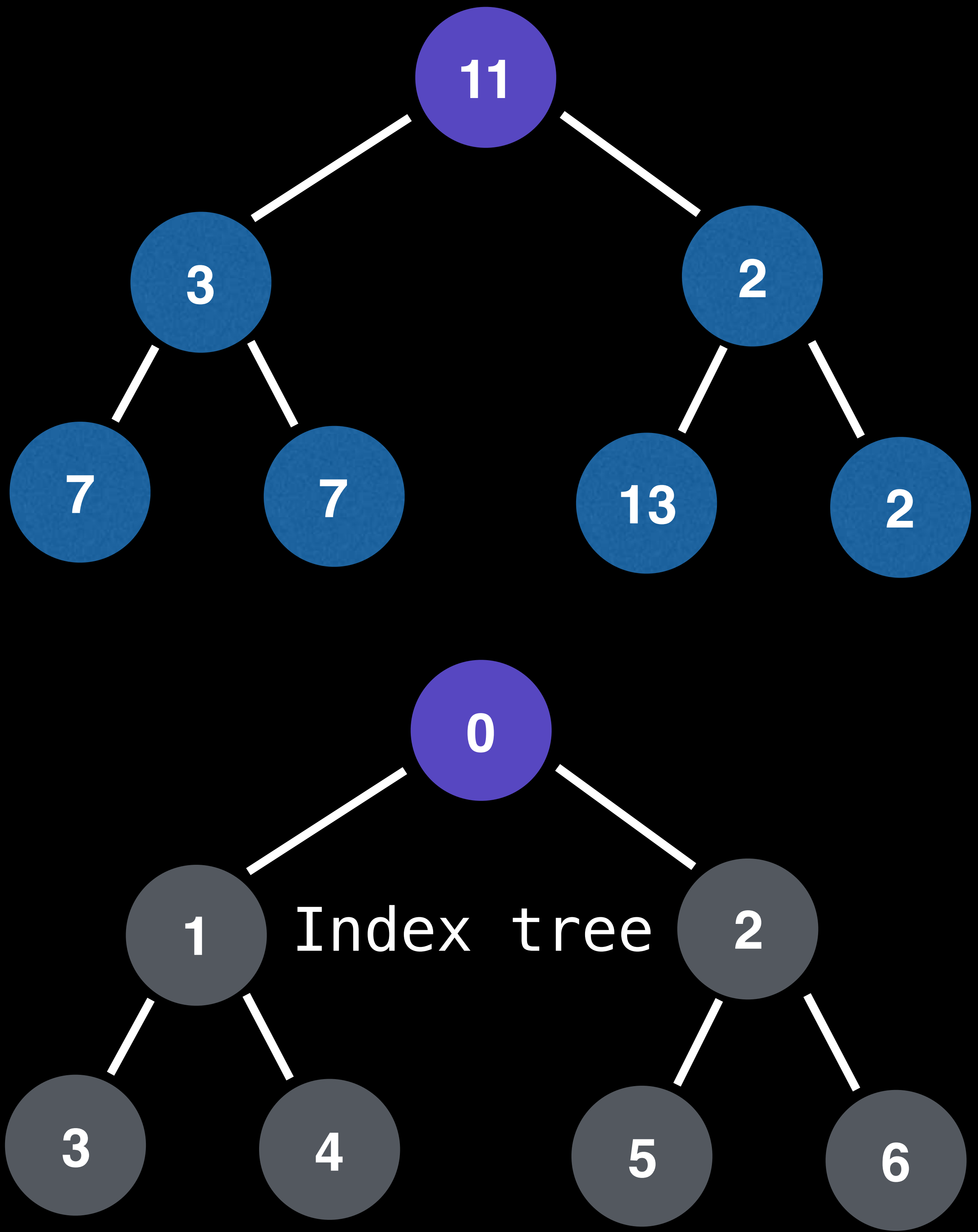
Instructions:

```
insert(3)
remove(2)
poll()
```

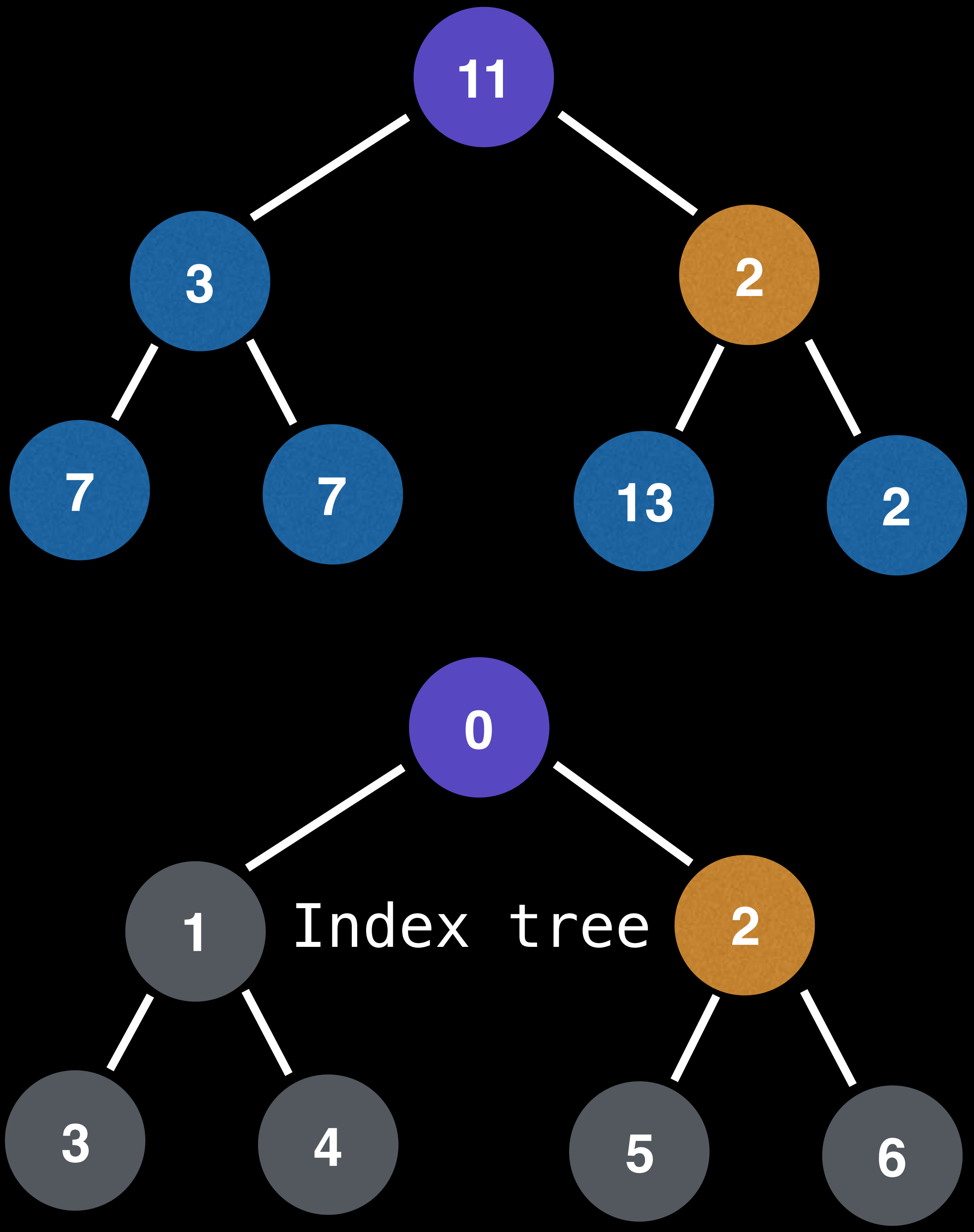
Node Value	Postion(s)
2	2, 6
7	3, 4
11	0
13	5
3	1

Instructions:

```
insert(3)
remove(2)
poll()
```



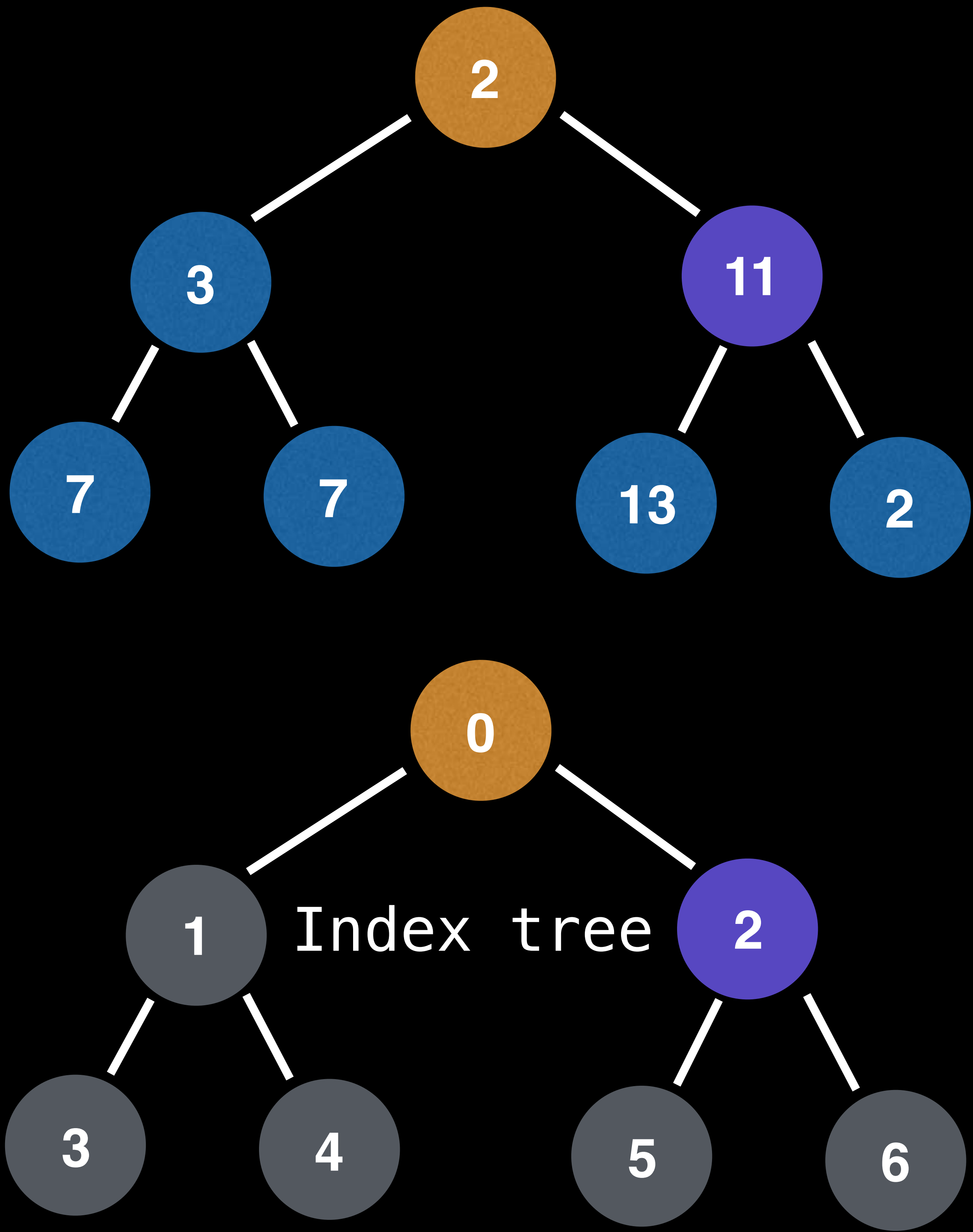
Node Value	Postion(s)
2	2, 6
7	3, 4
11	0
13	5
3	1



Instructions:

```
insert(3)
remove(2)
poll()
```

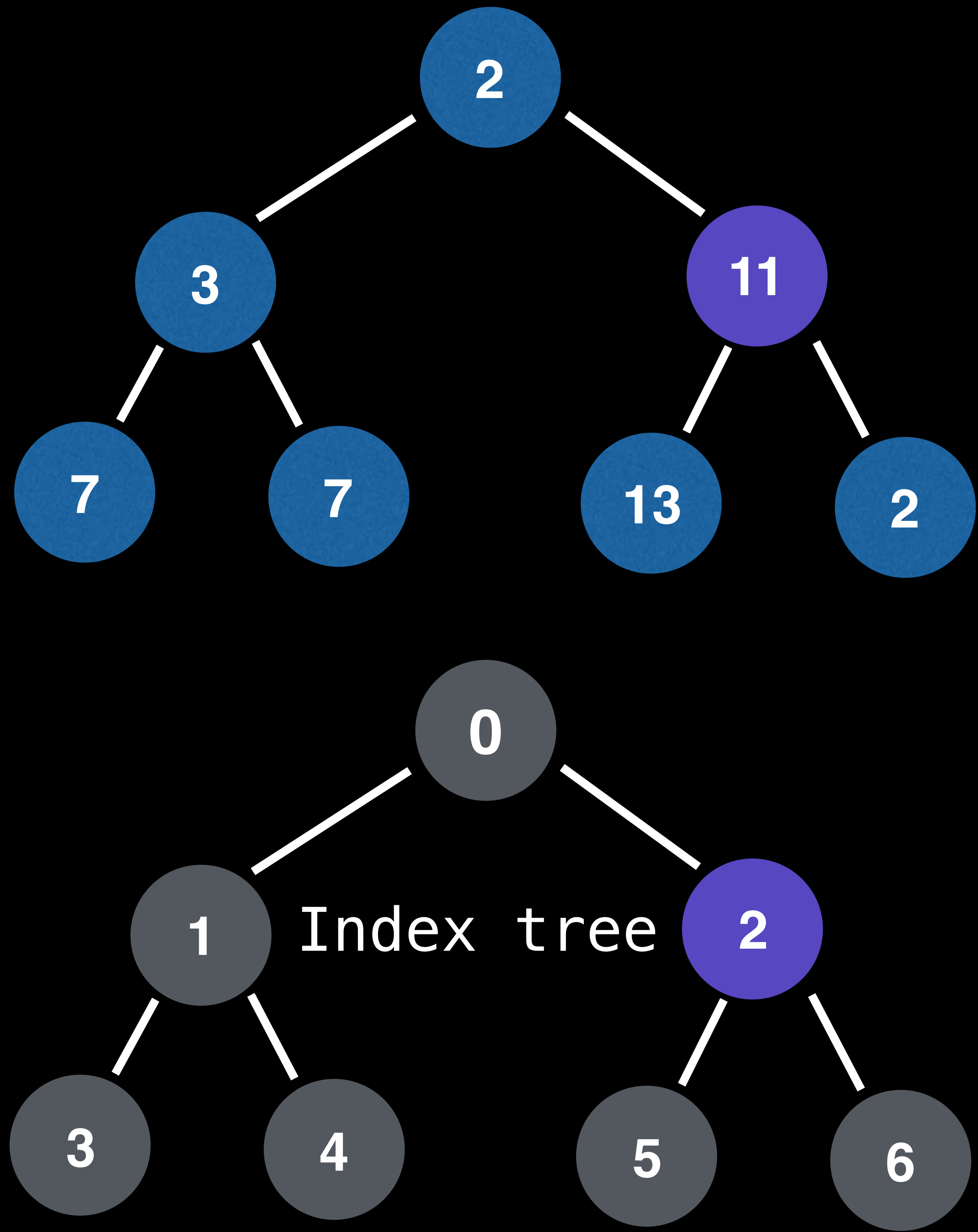
Node Value	Postion(s)
2	0, 6
7	3, 4
11	2
13	5
3	1



Instructions:

```
insert(3)
remove(2)
poll()
```

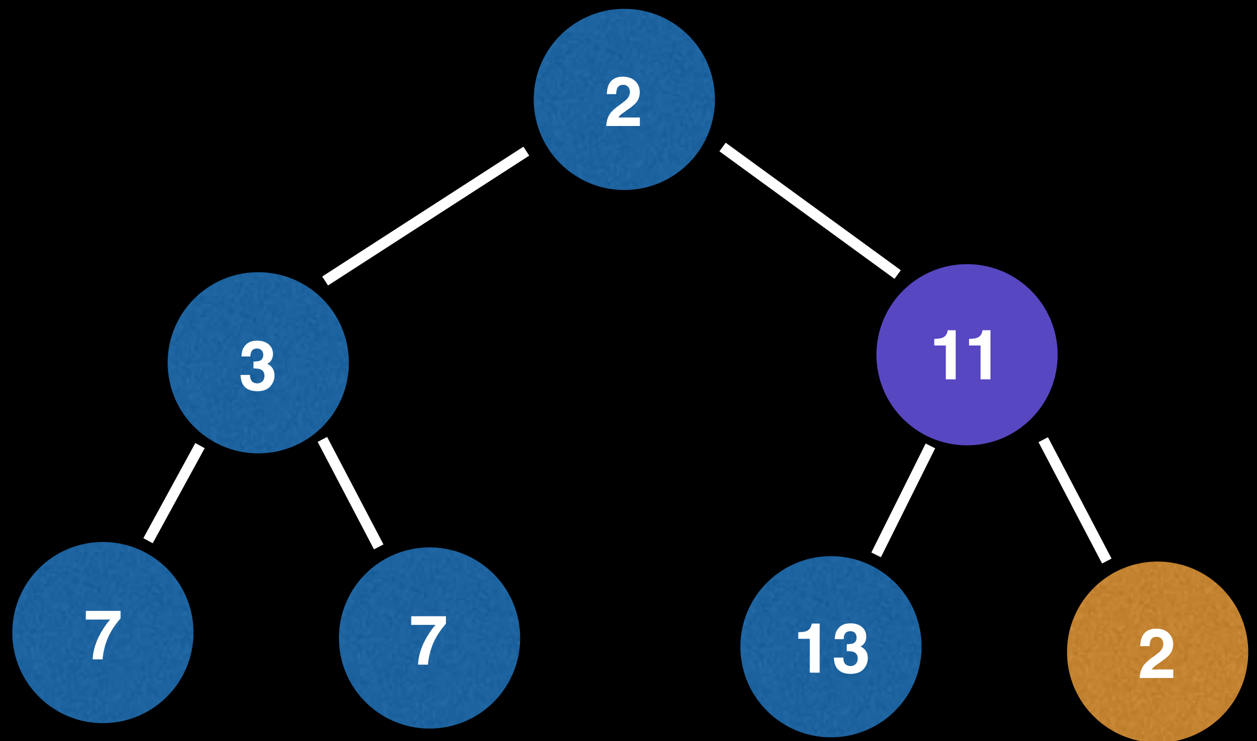
Node Value	Postion(s)
2	0, 6
7	3, 4
11	2
13	5
3	1



Instructions:

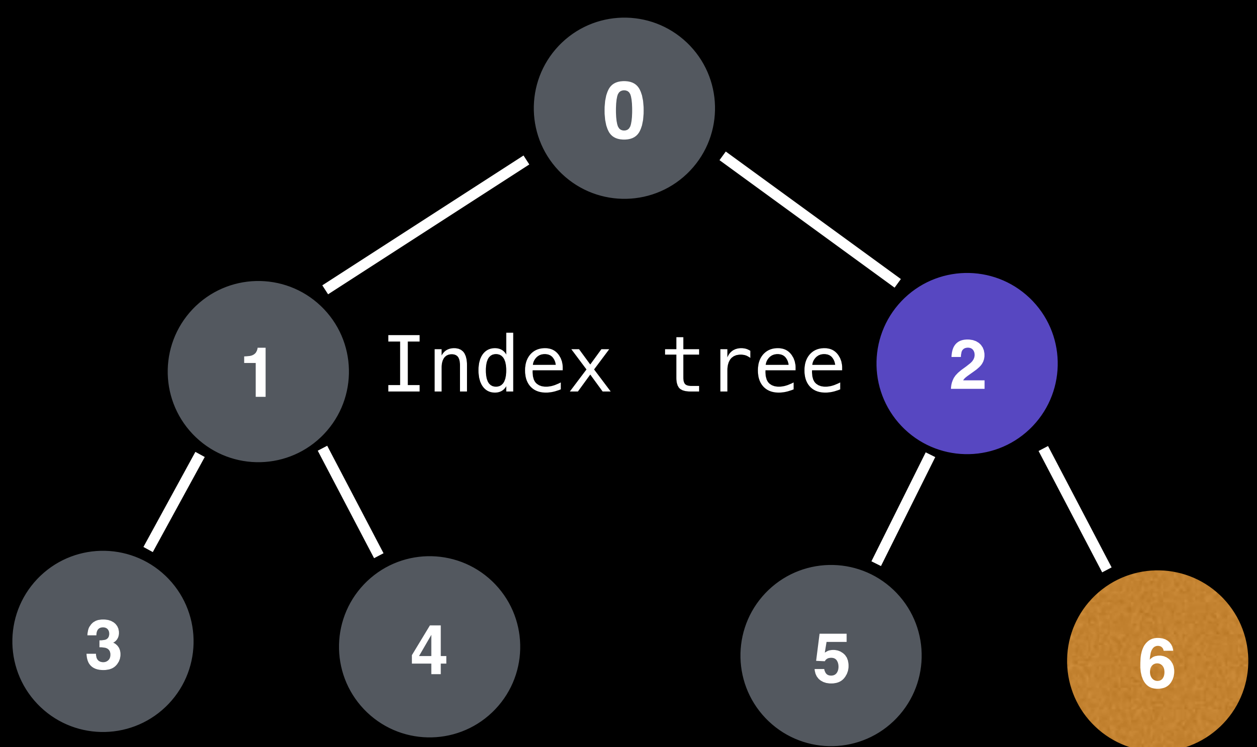
```
insert(3)
remove(2)
poll()
```

Node Value	Postion(s)
2	0, 6
7	3, 4
11	2
13	5
3	1

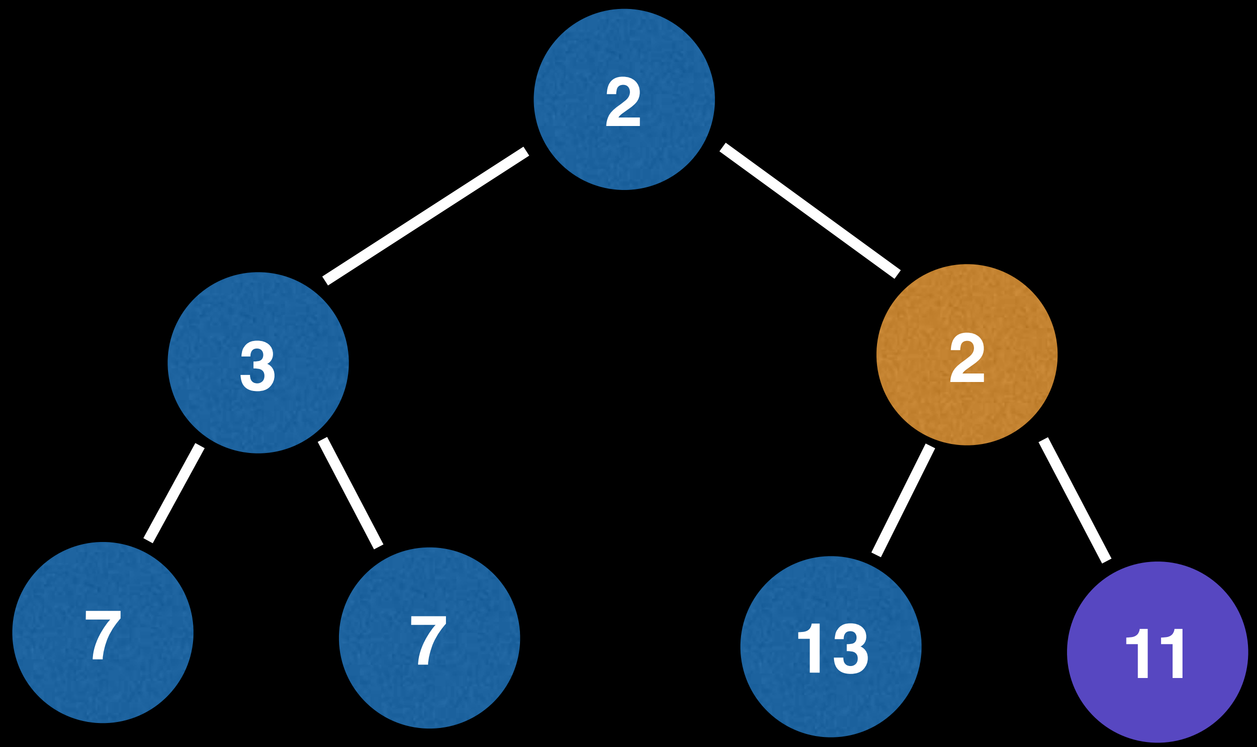


Instructions:

```
insert(3)
remove(2)
poll()
```

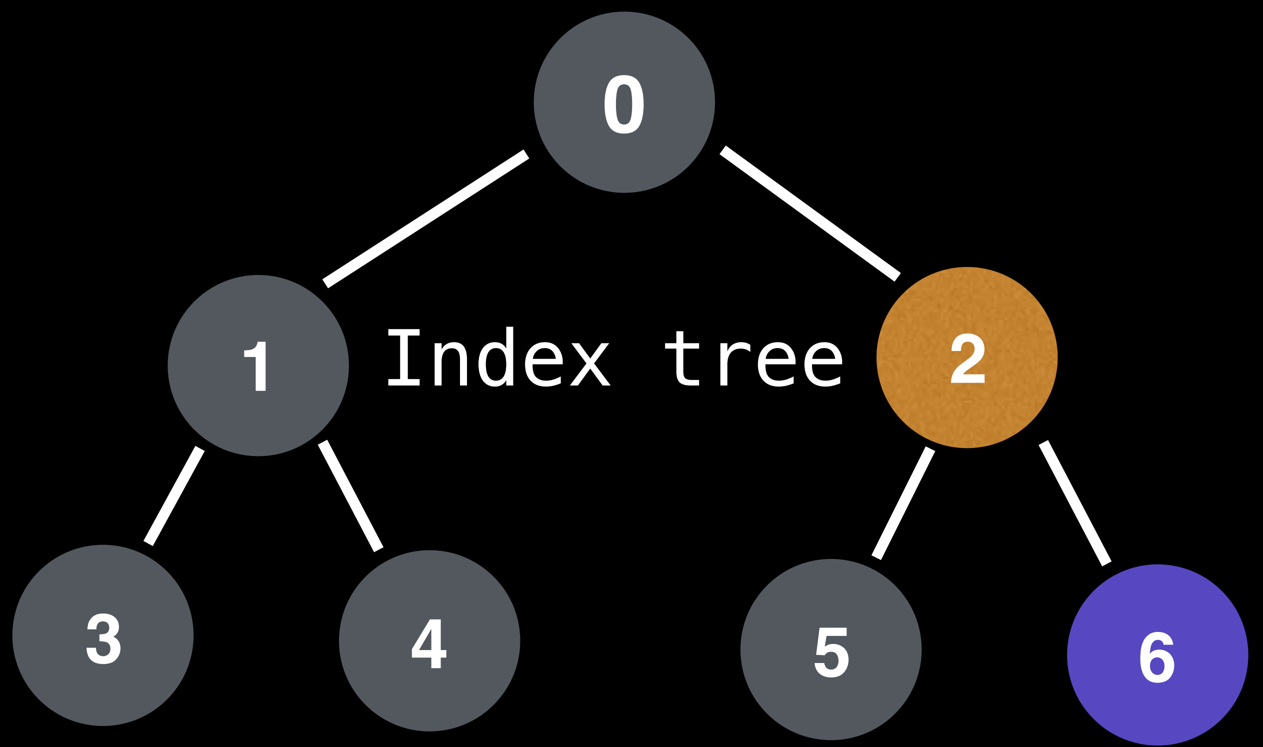


Node Value	Postion(s)
2	0, 2
7	3, 4
11	6
13	5
3	1

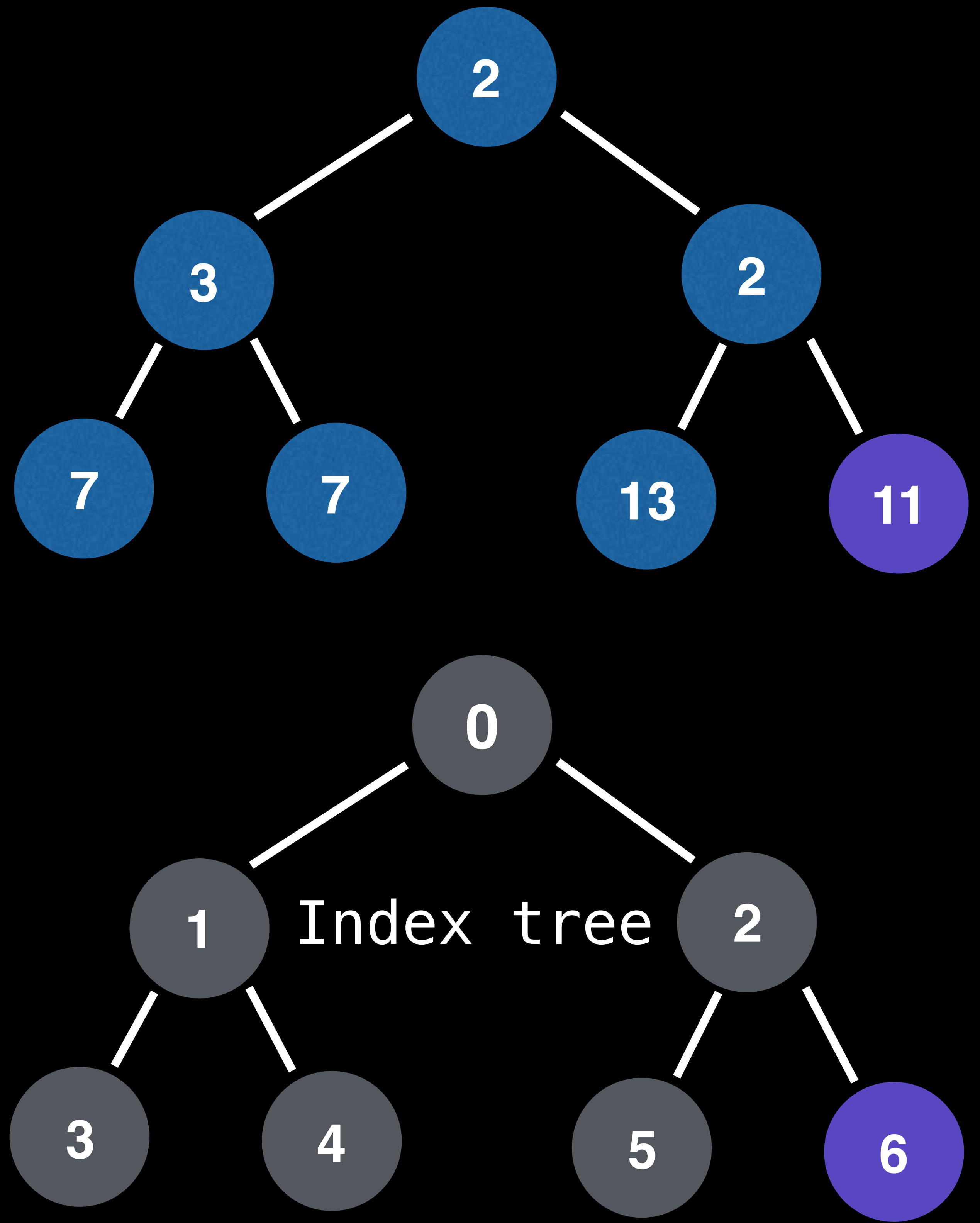


Instructions:

```
insert(3)
remove(2)
poll()
```



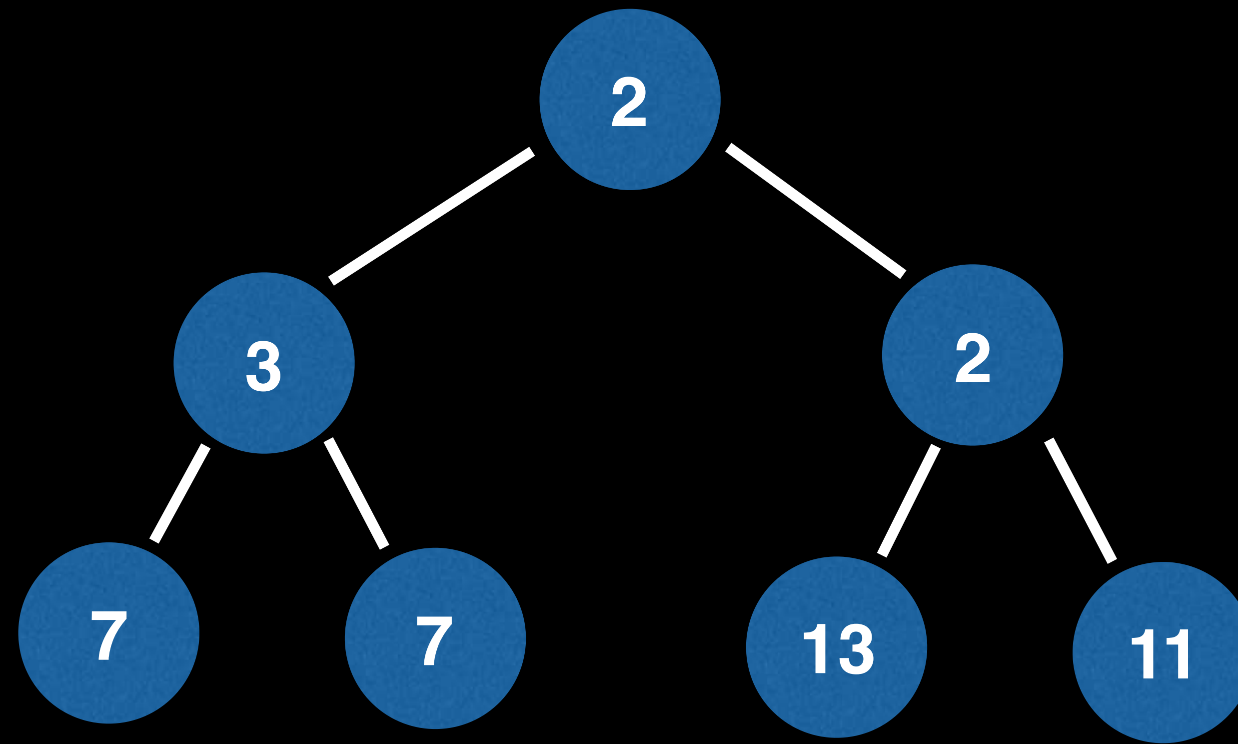
Node Value	Postion(s)
2	0, 2
7	3, 4
11	6
13	5
3	1

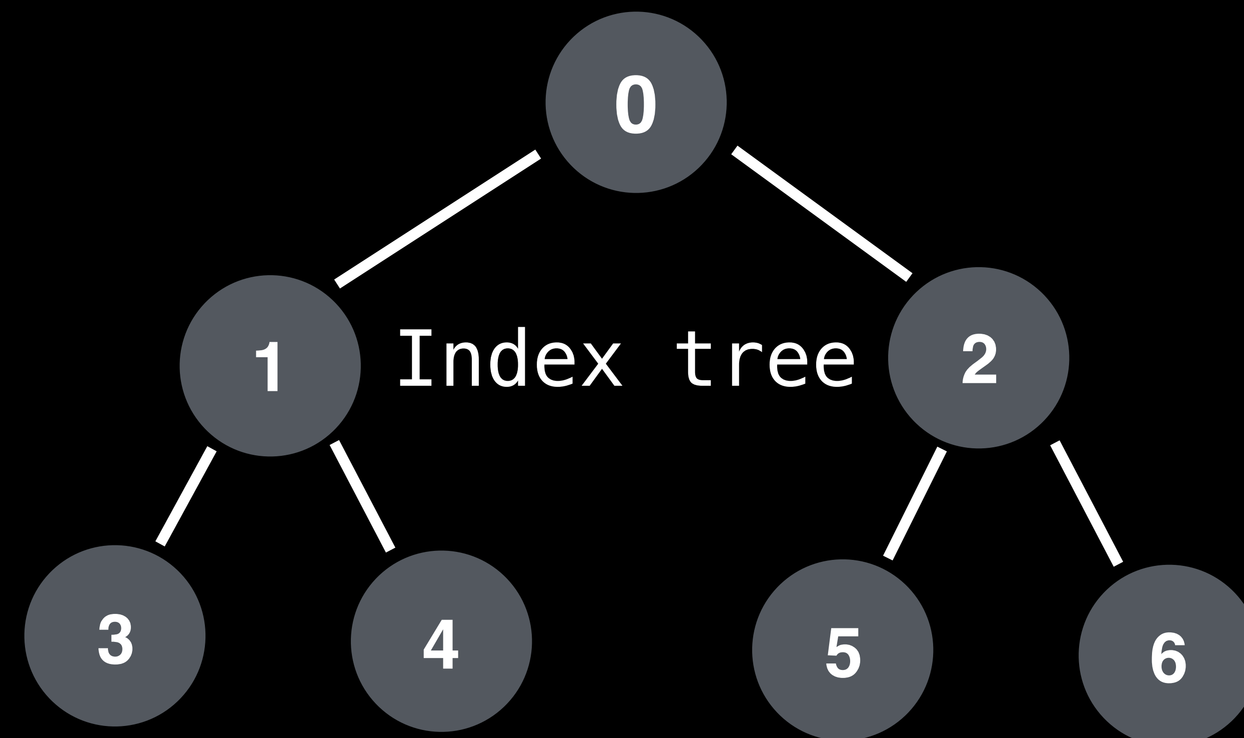


Instructions:

```
insert(3)
remove(2)
poll()
```

Node Value	Postion(s)
2	0, 2
7	3, 4
11	6
13	5
3	1





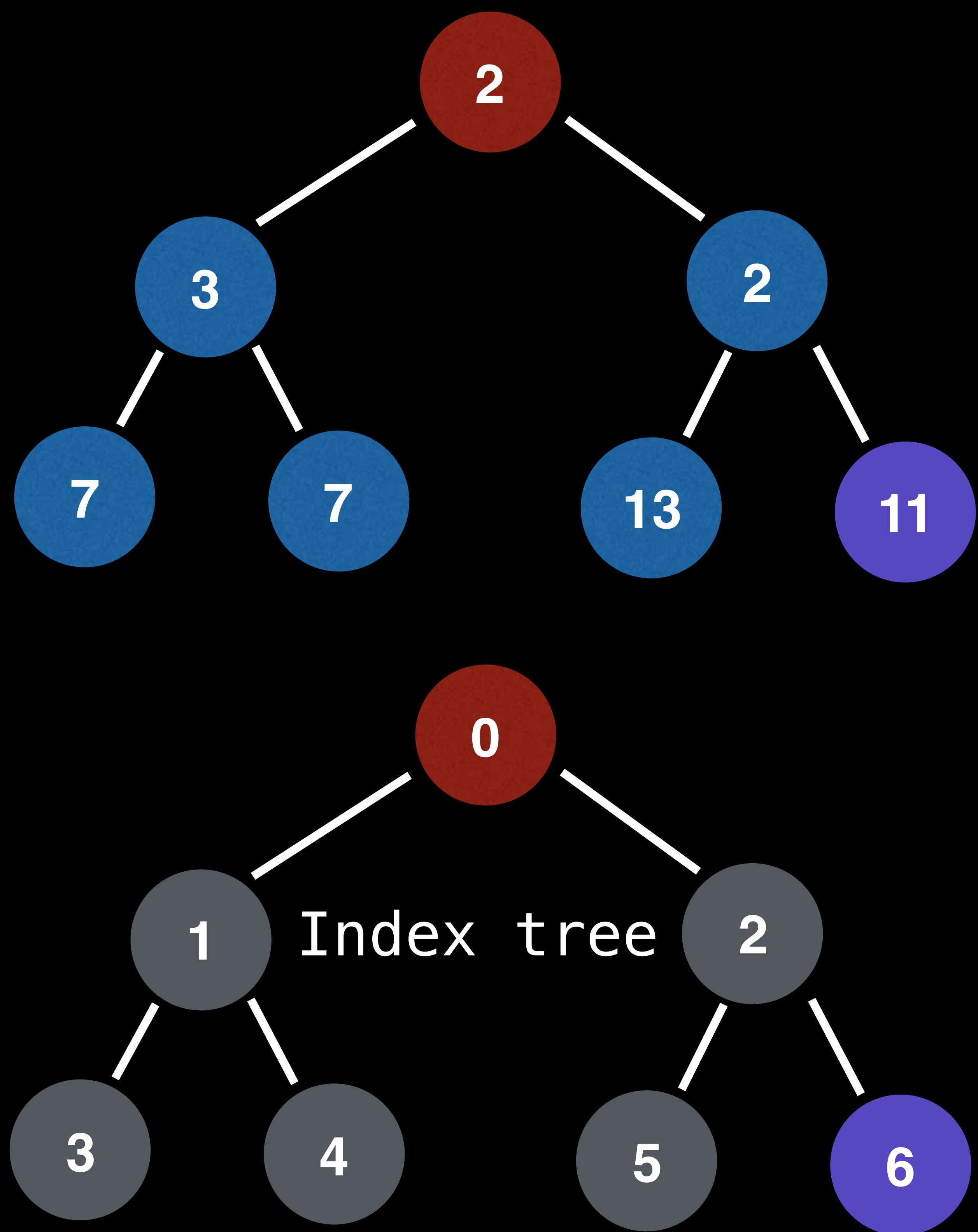
Instructions:

```
insert(3)
remove(2)
poll()
```

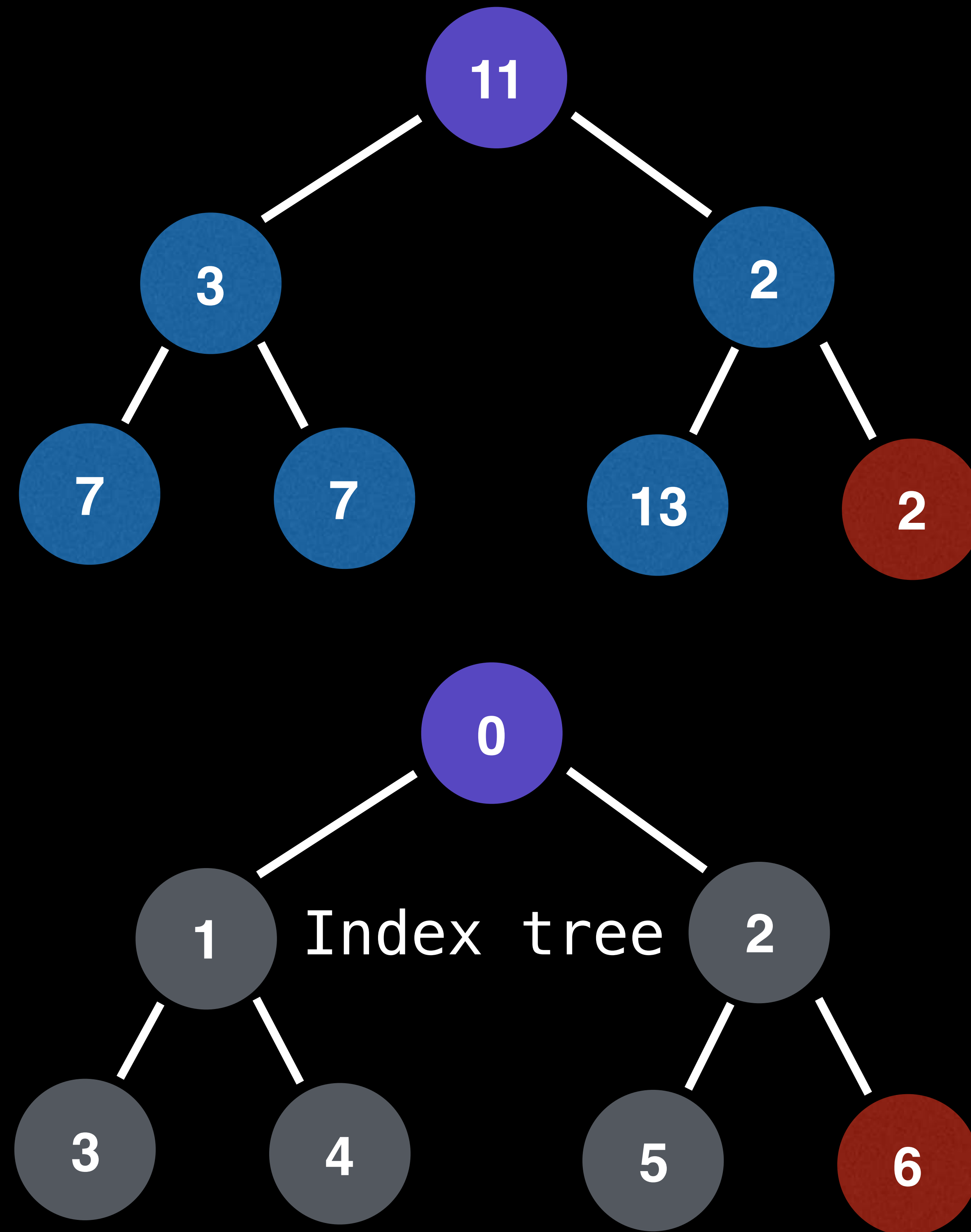
Node Value	Postion(s)
2	0, 2
7	3, 4
11	6
13	5
3	1

Instructions:

```
insert(3)
remove(2)
poll()
```



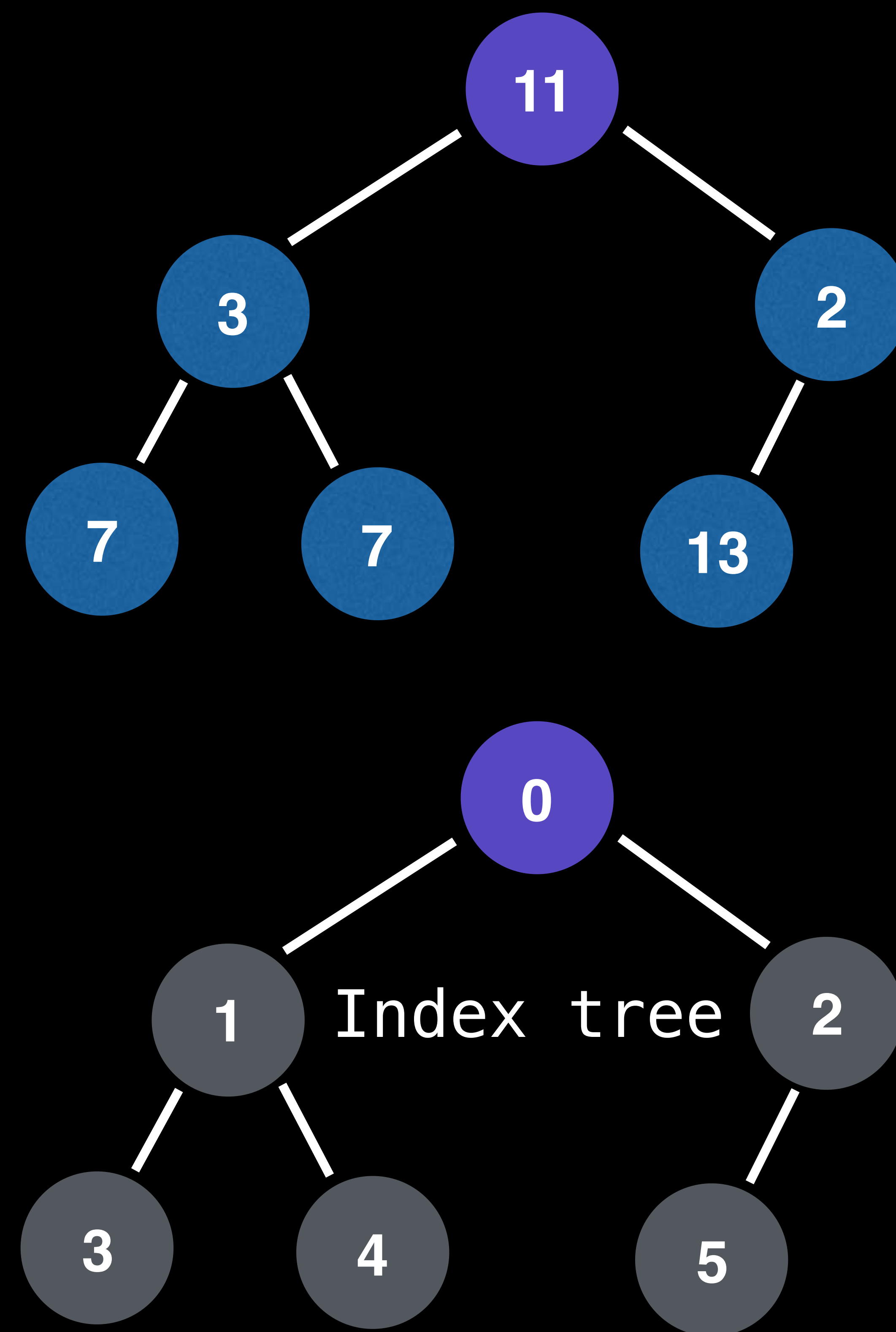
Node Value	Postion(s)
2	6, 2
7	3, 4
11	0
13	5
3	1



Instructions:

```
insert(3)
remove(2)
poll()
```

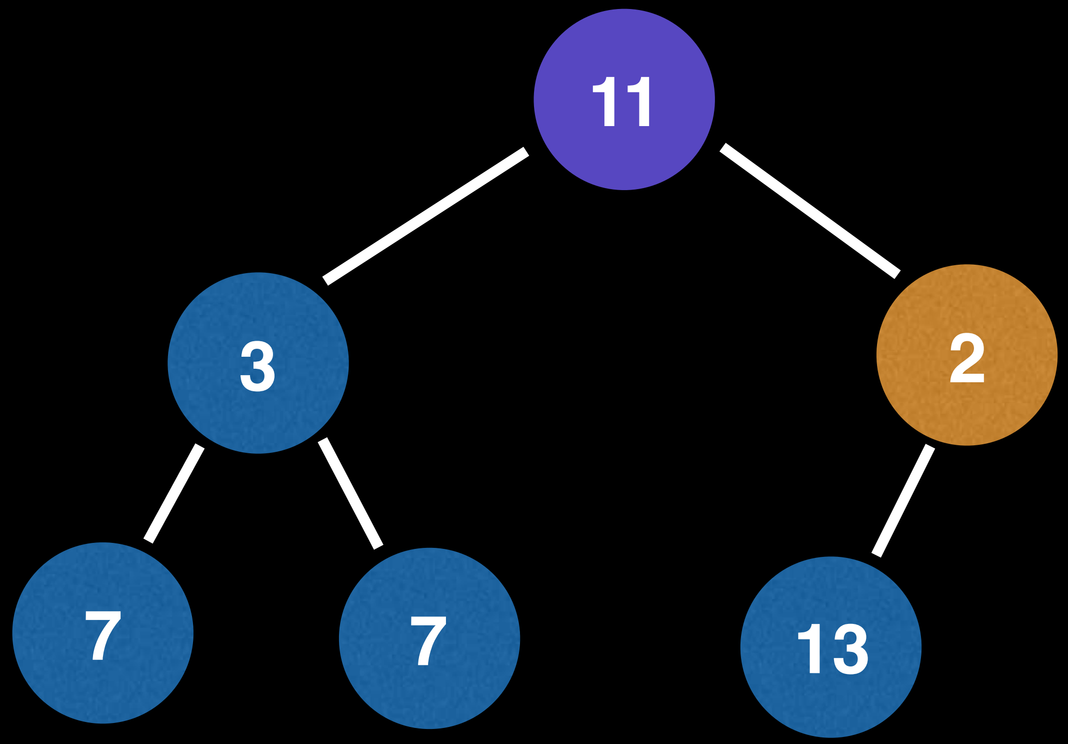
Node Value	Postion(s)
2	2
7	3, 4
11	0
13	5
3	1

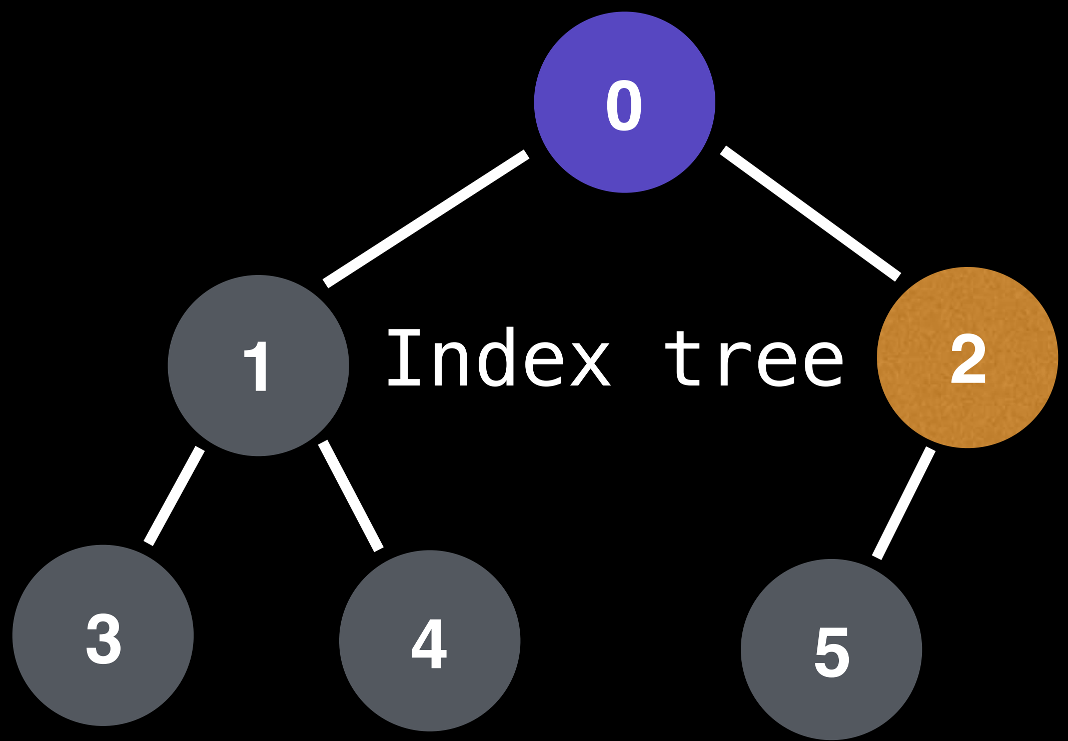


Instructions:

```
insert(3)
remove(2)
poll()
```

Node Value	Postion(s)
2	2
7	3, 4
11	0
13	5
3	1

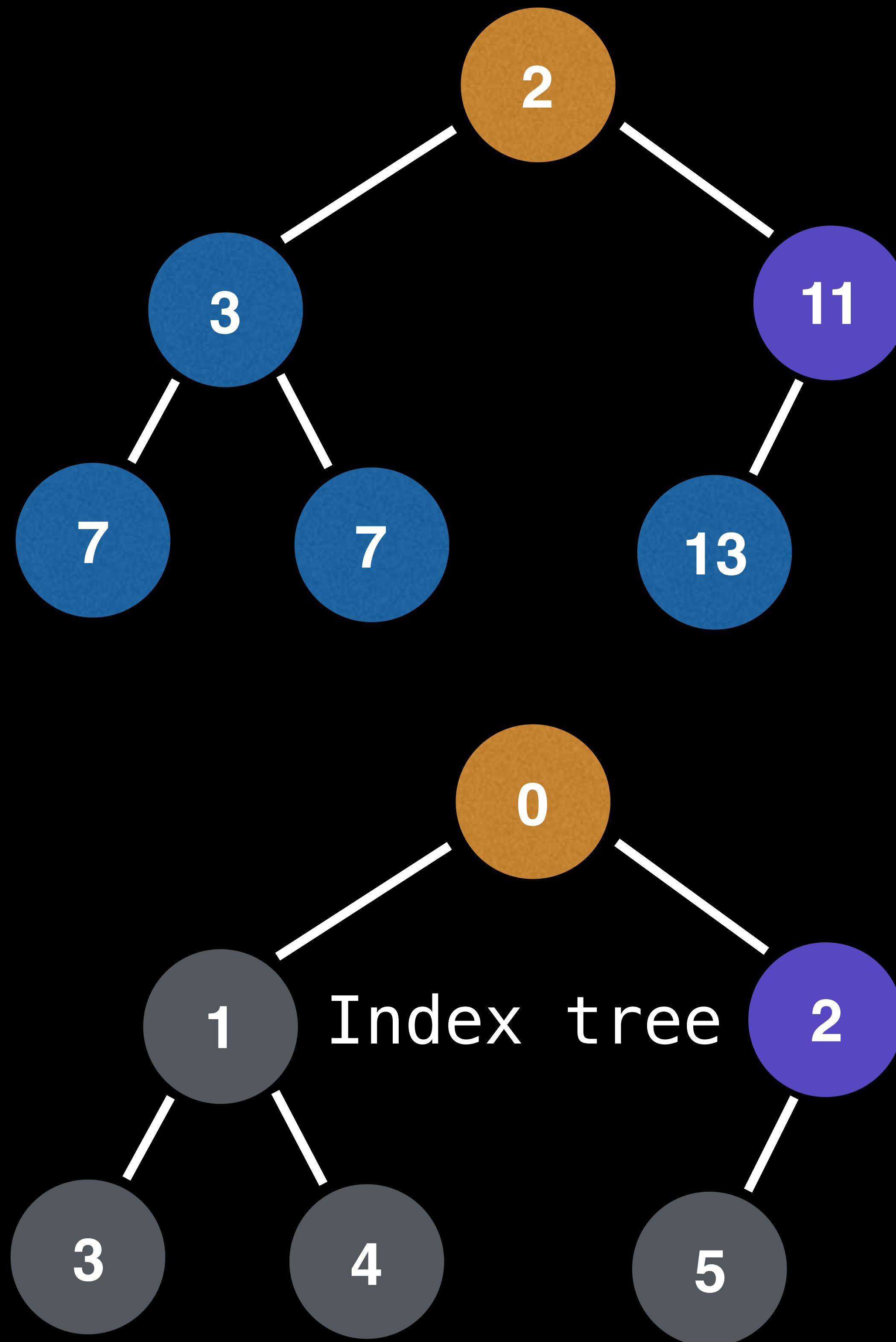




Instructions:

```
insert(3)
remove(2)
poll()
```

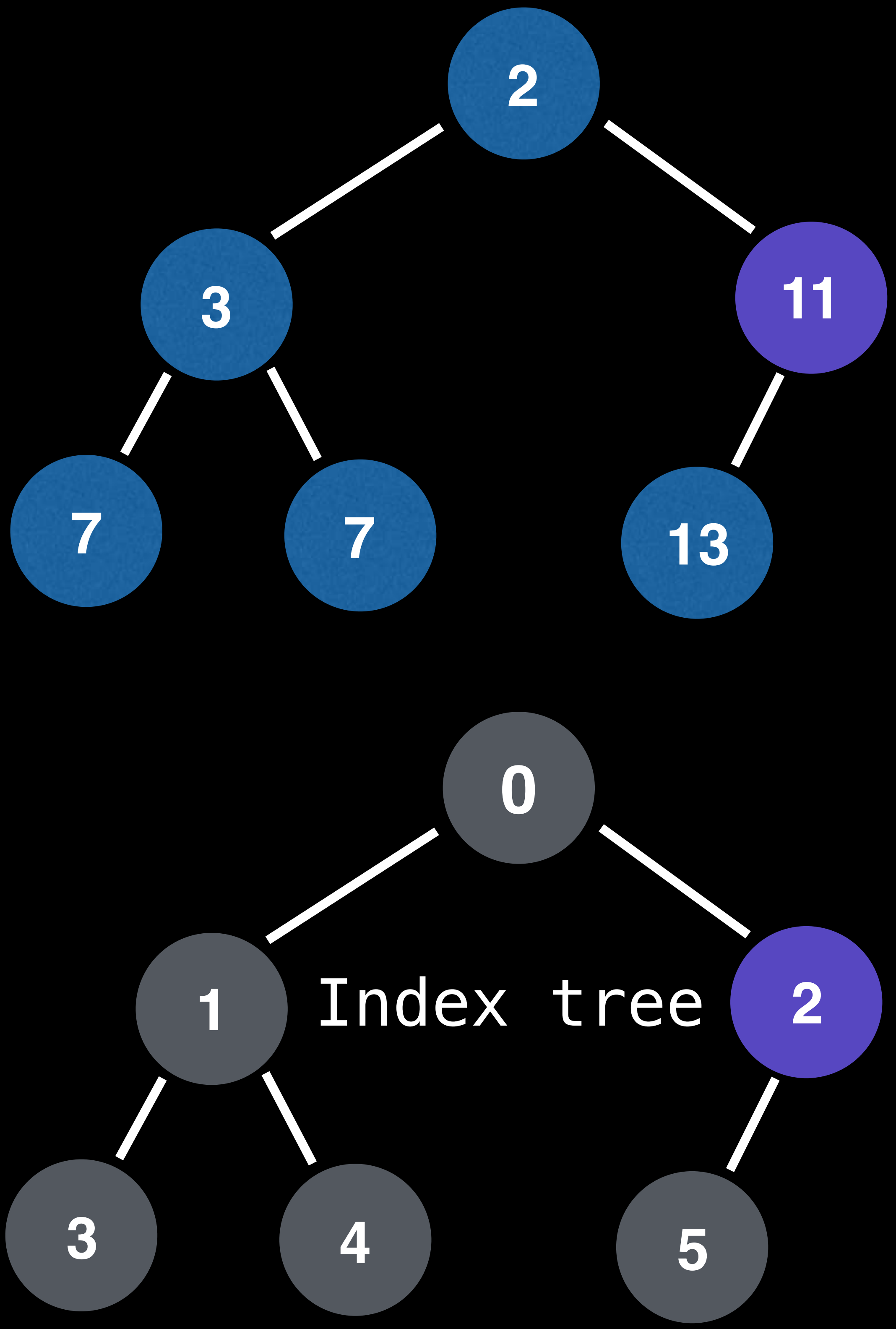
Node Value	Postion(s)
2	0
7	3, 4
11	2
13	5
3	1



Instructions:

```
insert(3)
remove(2)
poll()
```

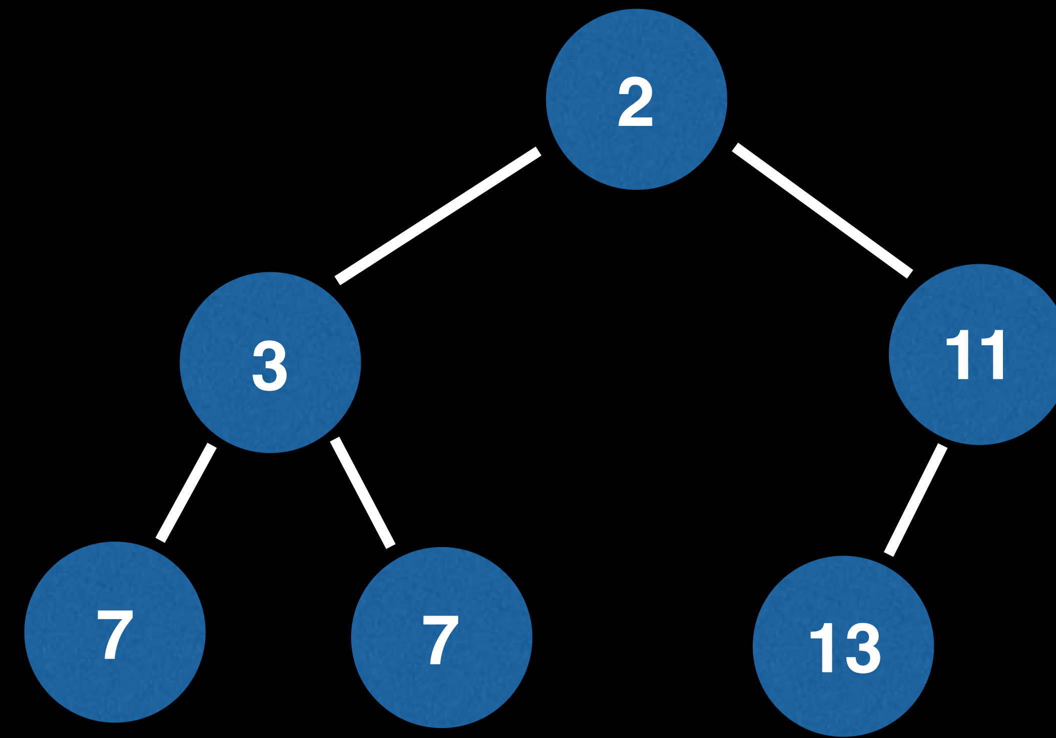
Node Value	Postion(s)
2	0
7	3, 4
11	2
13	5
3	1

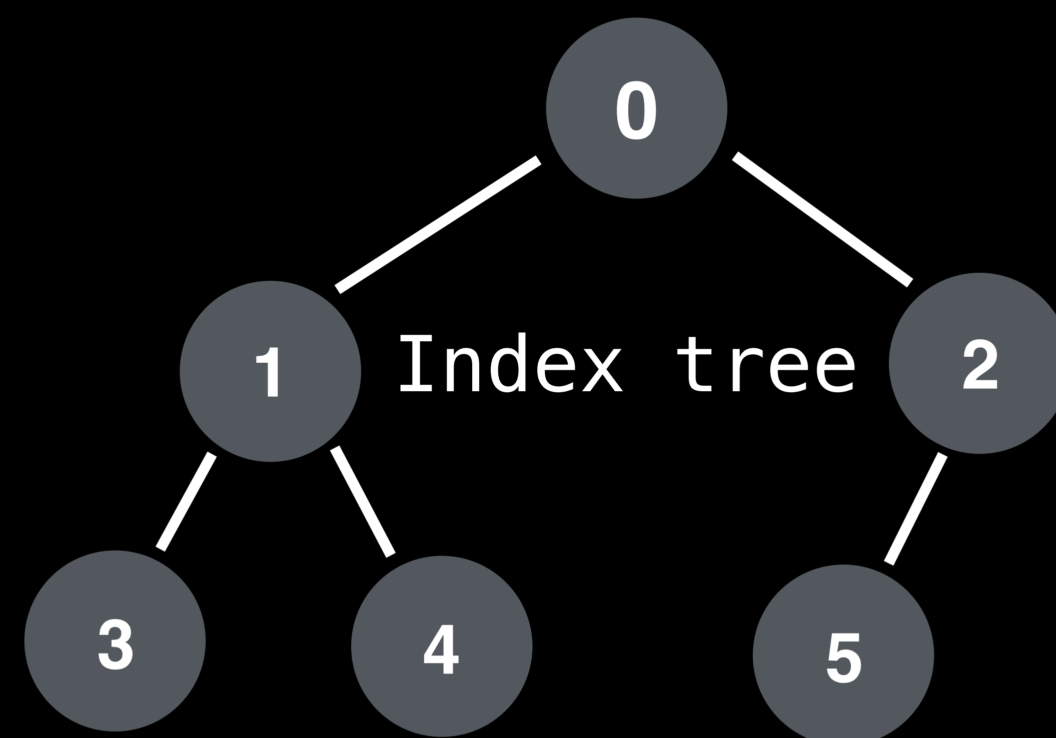


Instructions:

```
insert(3)
remove(2)
poll()
```

Node Value	Postion(s)
2	0
7	3, 4
11	2
13	5
3	1





<u>Instructions</u>:

```
insert(3)
remove(2)
poll()
```