5008CEM Programming For Developers

Search Algorithms

Beate Grawemeyer | John Halloran

Intended Learning

- Understand how a range of search algorithms work:

  linear search | binary search | interpolation search

- Be able to implement these
- Ahead of moving further into relationships between algorithms and data structures, understand:

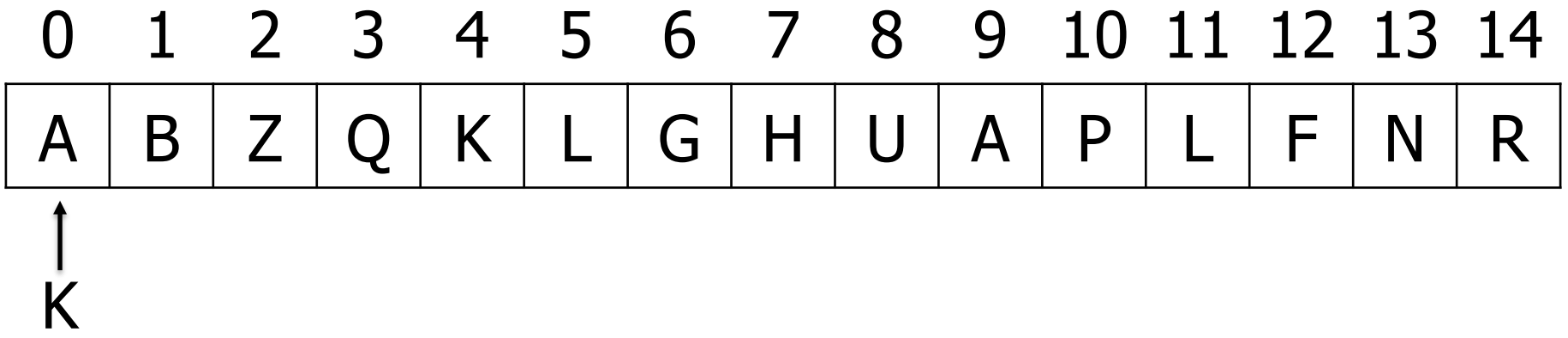
  heap sort

Linear Search

- Also called sequential search
- Iterate over elements until found or sequence ends

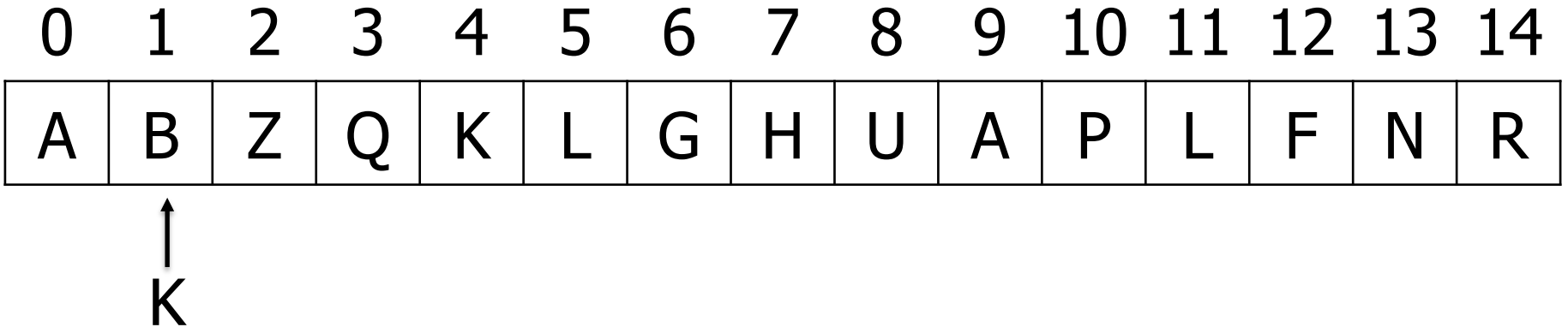
How Linear Search Works

Search value: K



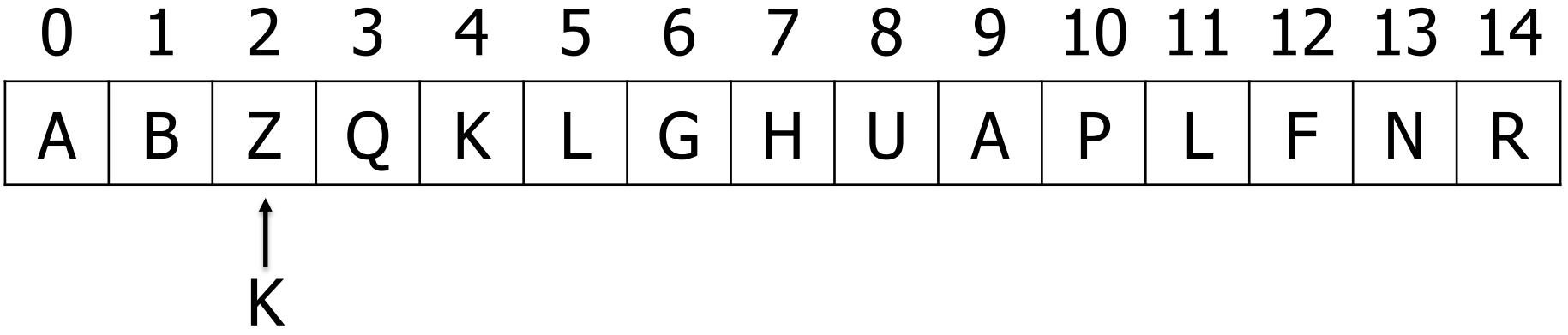
How Linear Search Works

Search value: K



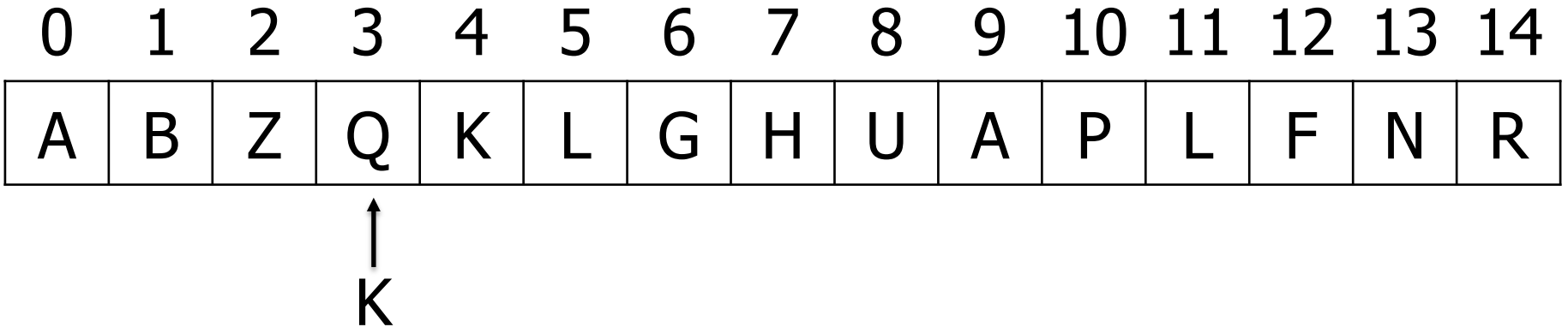
How Linear Search Works

Search value: K



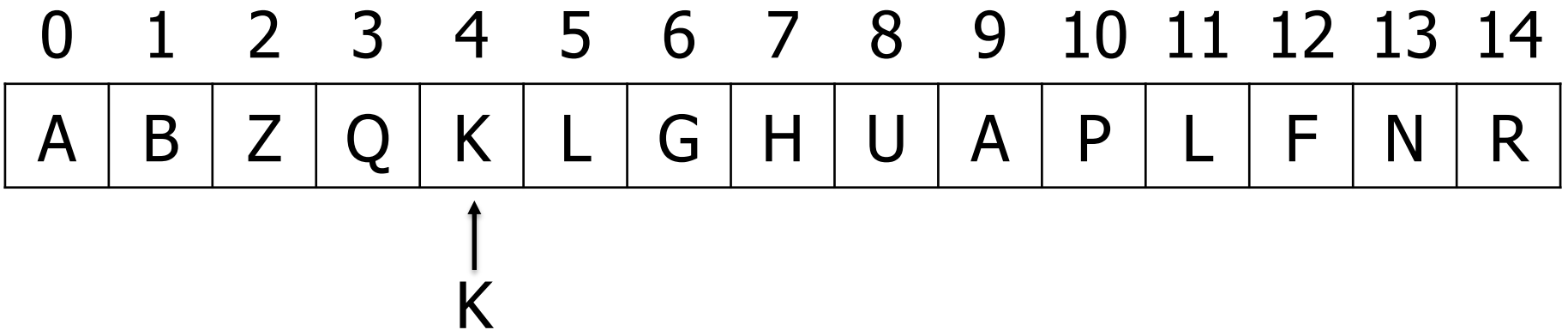
How Linear Search Works

Search value: K



How Linear Search Works

Search value: K



Linear Search

- Not a very good algorithm
- We have to check every single item in sequence
- Inefficient

Linear Search Pseudocode

- Not provided!
- Linear Search is very simple
- Please work out and implement the pseudocode yourself

Binary Search

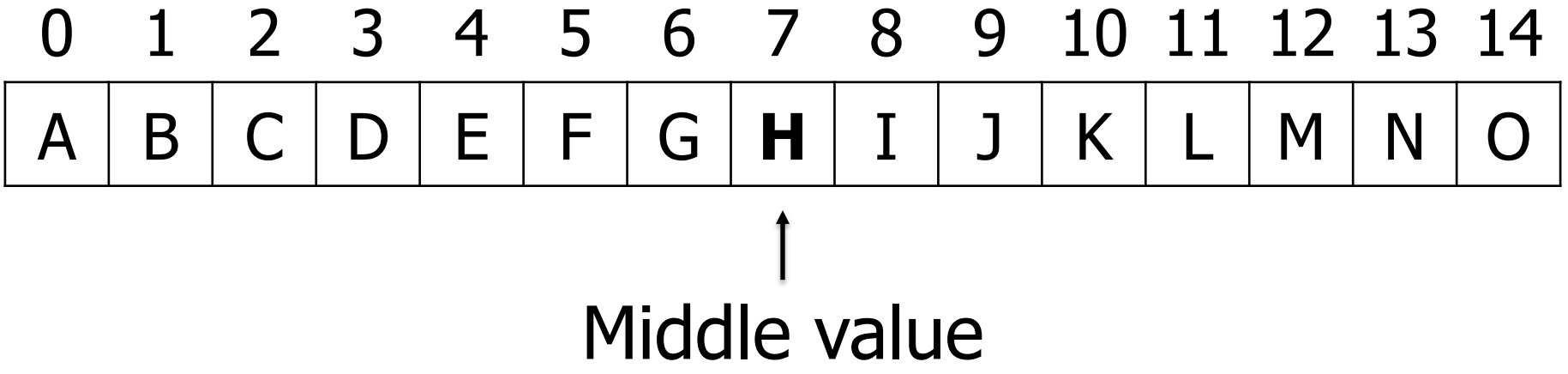
- Much faster than linear search
- A ‘Divide and Conquer’ algorithm
- Only works on sorted sequences

How It Works: Algorithm

1. Find middle value of sequence
2. If search value is the middle value, then success
3. If search value is less than the middle value, discard the top half of the sequence
4. If search value is greater than the middle value, discard the bottom half of the sequence
5. Repeat from (1) until value is found or length of the sequence is zero (i.e. value is not found)

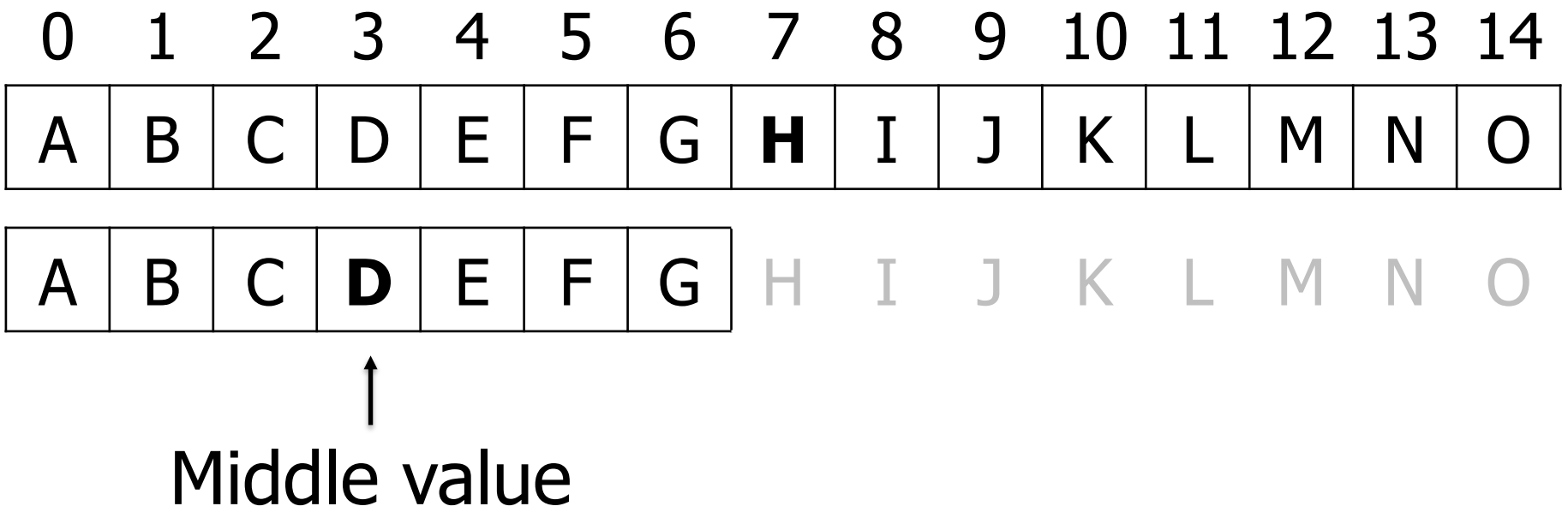
How Binary Search Works

Search value: **E**



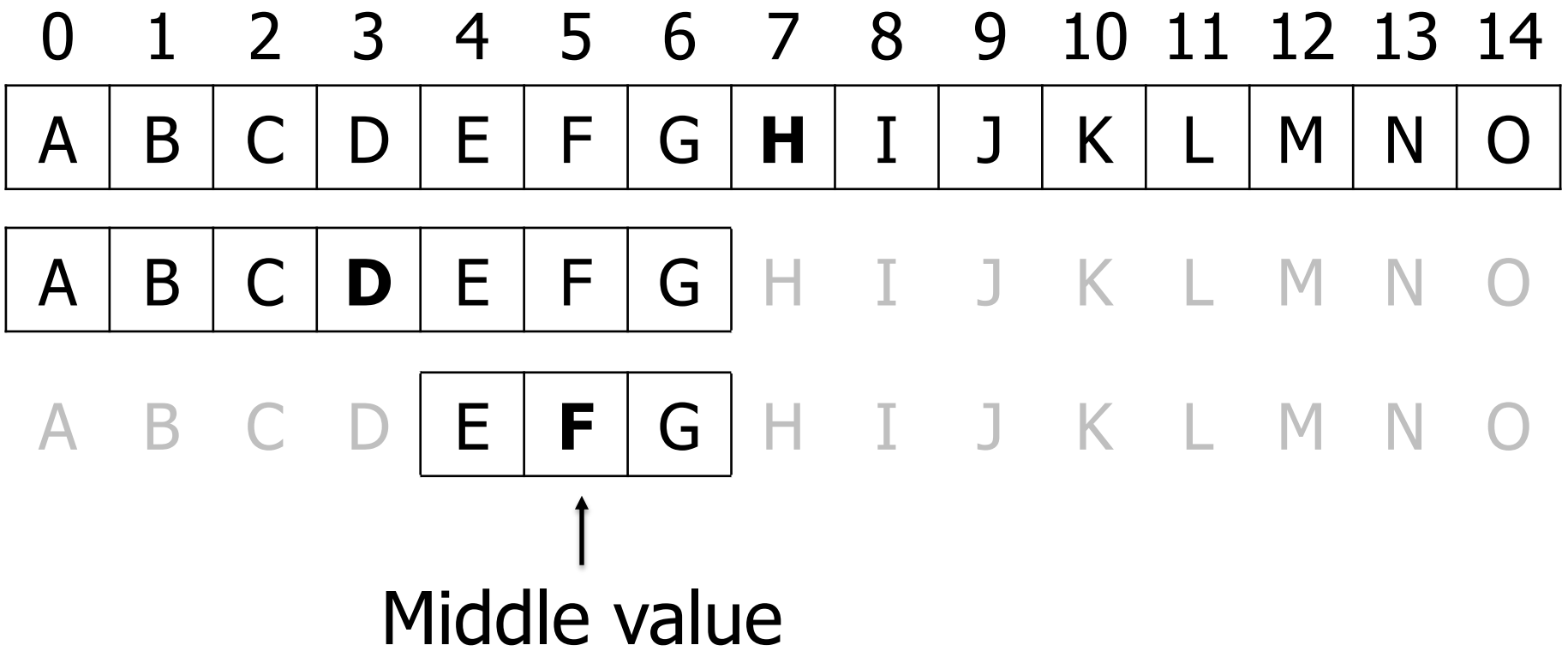
How Binary Search Works

Search value: **E**



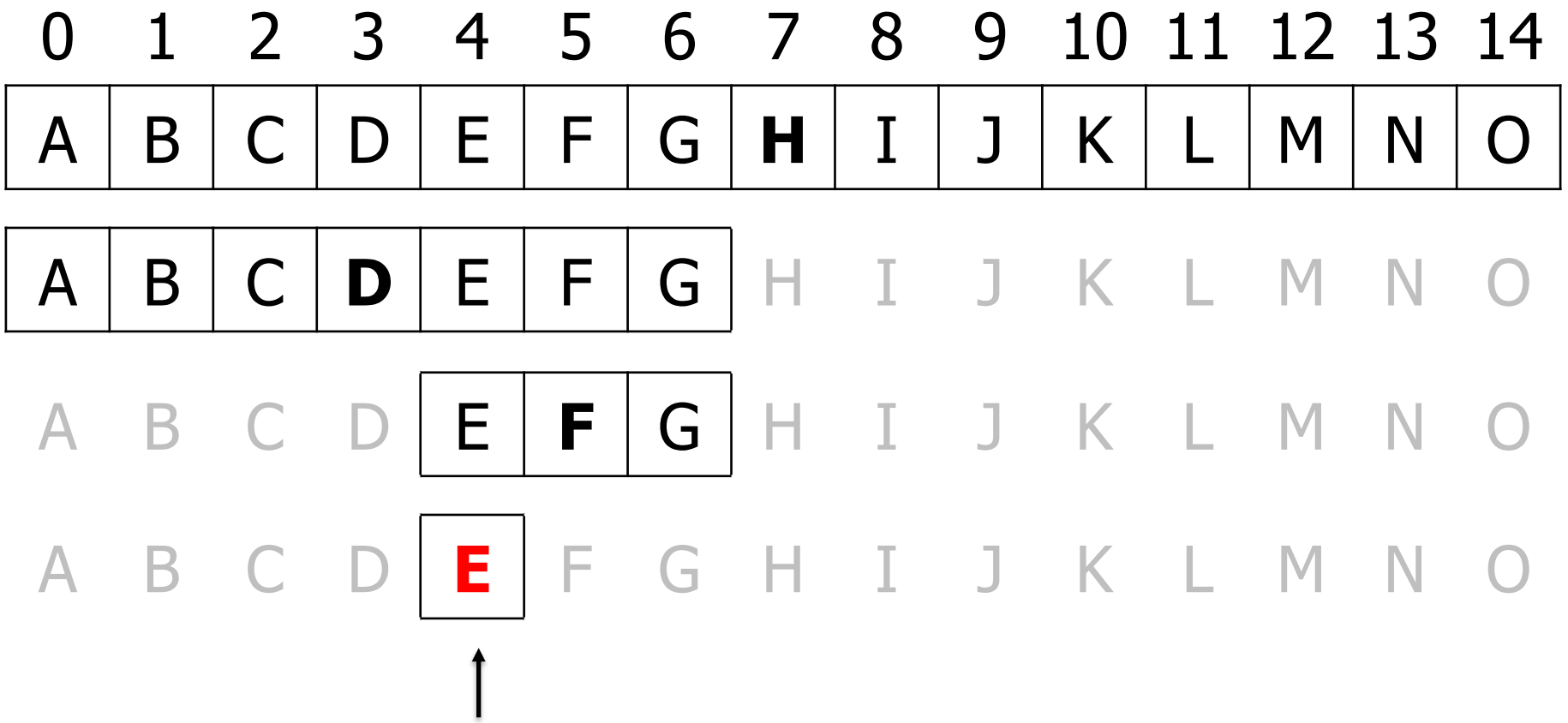
How Binary Search Works

Search value: **E**



How Binary Search Works

Search value: **E**



Middle value == Search value

Linear Search vs Binary Search

- Binary Search much faster
- To search a million elements linearly could worst case mean a million comparisons
- For Binary Search the worst case is 20
- This makes Binary Search more efficient
- In complexity terms, Binary Search is O(log n); Linear is O(n)
  - (We will start to cover complexity analysis in the next lecture)

Binary Search Iterative Pseudocode

```
BINARY_SEARCH(sequence, value)
    middle ← GET_MIDDLE(sequence)
    WHILE value != middle
        IF length(sequence) > 1
            IF value > sequence[middle]
                sequence ← sequence[middle .. length(sequence)]
            ELSE
                sequence ← sequence[0 .. middle]
        ELSE
            IF value != sequence[middle]
                RETURN FALSE
    RETURN TRUE

//NOTE: This pseudocode includes a call to a function GET_MIDDLE(sequence)
//This function is not defined by the pseudocode: you'd need to define it
//Note that the sequence length can be even or odd, so you'd need to work out
//how to decide the middle for either
//You might prefer to do this in the function BINARY_SEARCH
```

Binary Search Recursive Pseudocode

```
BINARY_SEARCH(sequence, value)
    ans ← True
    IF length(sequence) = 1 and value != sequence[0]
        ans = False
    ELSE
        middle = length(sequence) // 2          //find middle
        IF value != sequence[middle]:           //if value is not middle
            IF value > sequence[middle]:        //if greater
                sequence = sequence[middle .. length(sequence)]
                                                //recurse on right half
                ans = BINARY_SEARCH (sequence, value)
            ELSE                                //if less
                sequence = sequence[0 .. middle]
                                                //recurse on left half
                ans = BINARY_SEARCH (sequence, value)
    RETURN ans
```

Implementing Binary Search

- Binary Search can be implemented iteratively or recursively
- It appeared at the end of 4000CEM: you may already have done one or both
- If not, this is a non-viva task in this week's labs
- Remember that it requires a sorted sequence

Interpolation Search

- Interpolation Search is an improved Binary Search
- It checks locations based on the key being searched

  Key = the value we are looking for

- The data needs to be sorted
- The data is ideally uniformly distributed

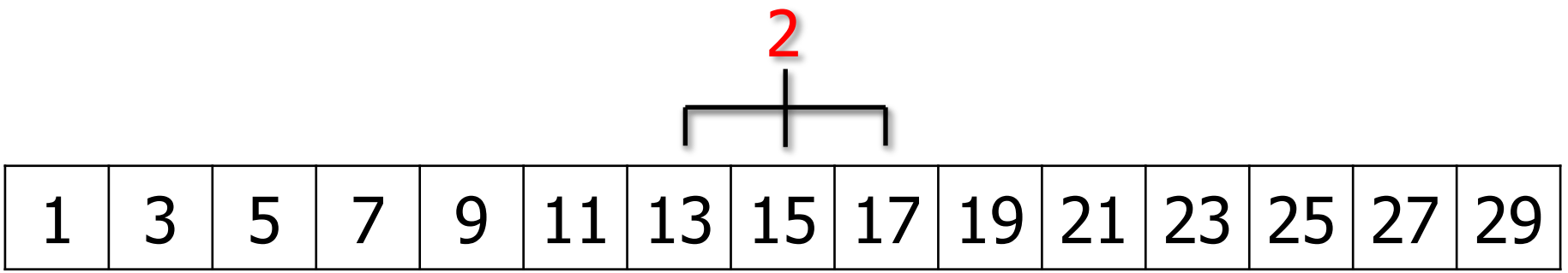
Uniform Distribution

1	3	5	7	9	11	13	15	17	19	21	23	25	27	29

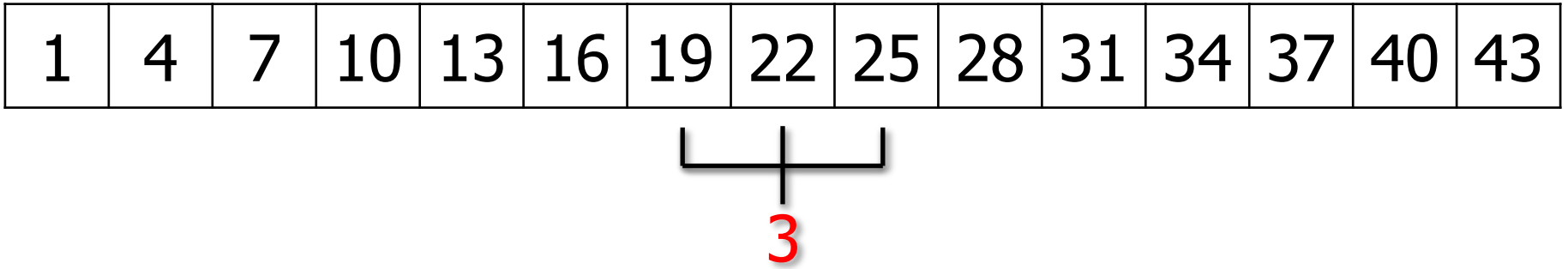
Same interval between elements

1	4	7	10	13	16	19	22	25	28	31	34	37	40	43

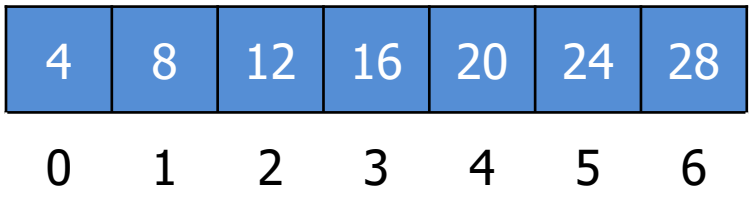
Uniform Distribution



Same interval between elements

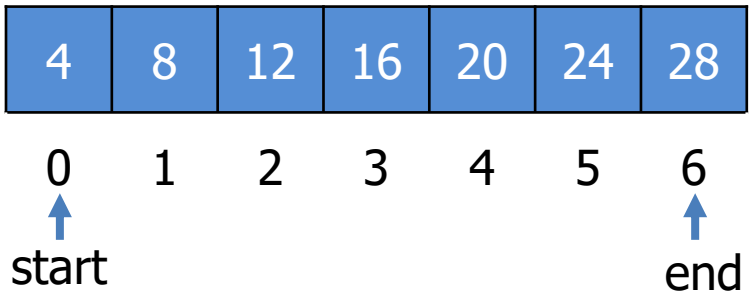


How Interpolation Search Works



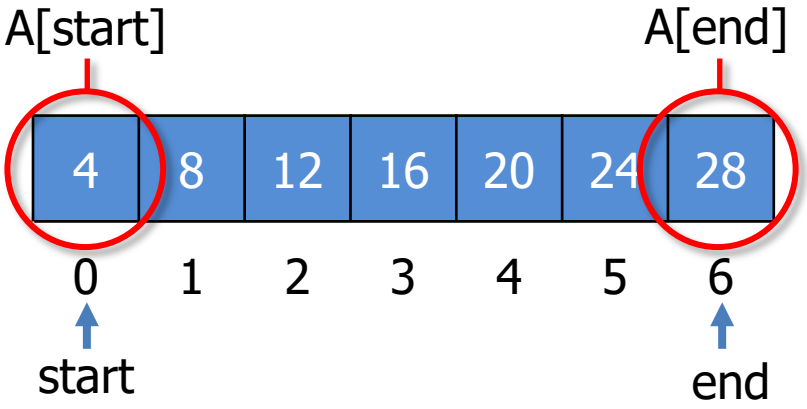
1 Sorted and uniformly distributed sequence A

How Interpolation Search Works



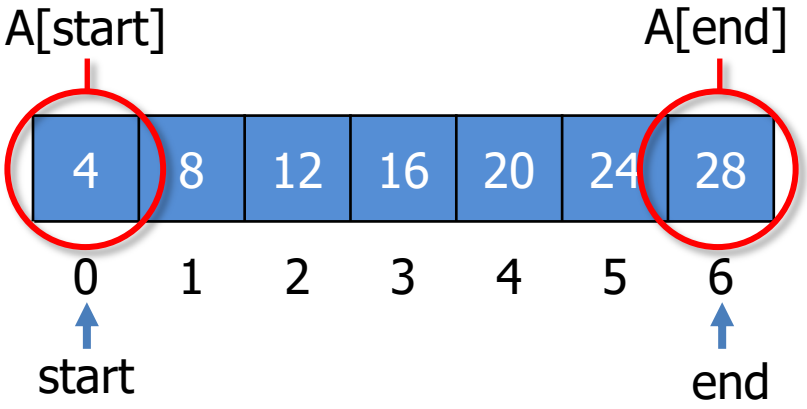
1. Sorted and uniformly distributed sequence A
2. start = 0; end = n-1

How Interpolation Search Works



1. Sorted and uniformly distributed sequence A
2. start = 0; end = n-1
3. $$pos = start + \frac{(key - A[start]) * (end - start)}{A[end] - A[start]}$$

How Interpolation Search Works

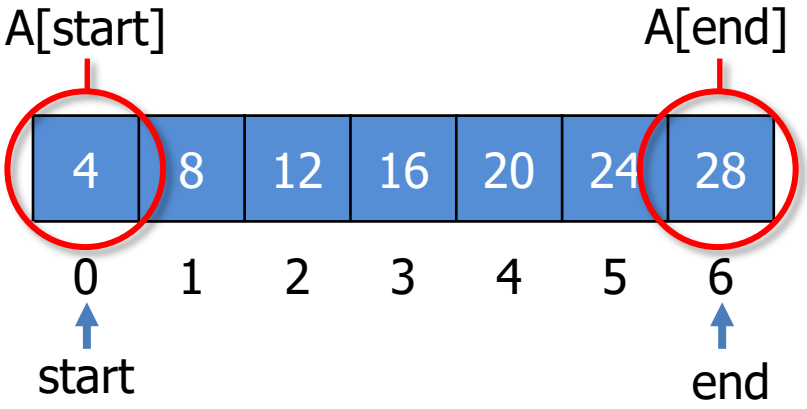


1. Sorted and uniformly distributed sequence A
2. start = 0; end = n-1

Key=12

3. $$pos = start + \frac{(key - A[start]) * (end - start)}{A[end] - A[start]}$$

How Interpolation Search Works



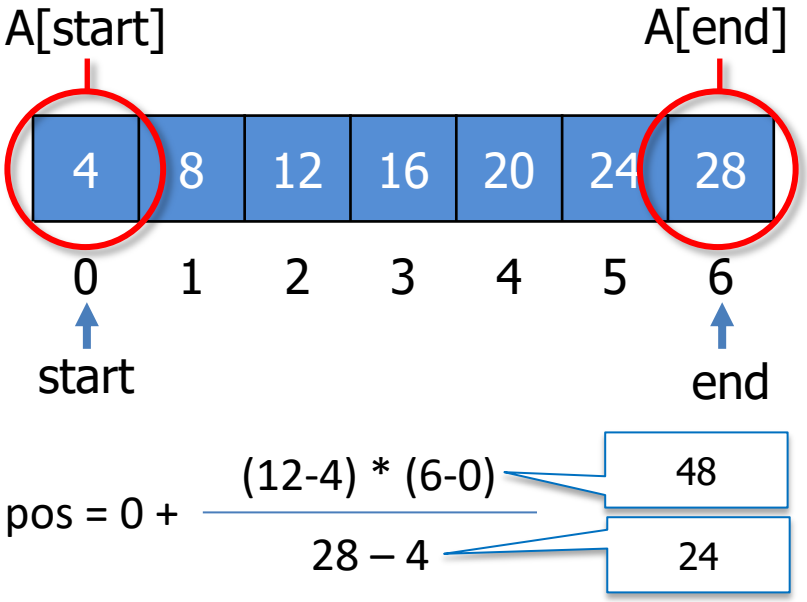
$$pos = 0 + \frac{(12-4) * (6-0)}{28 - 4}$$

1. Sorted and uniformly distributed sequence A
2. start = 0; end = n-1

Key=12

3. $$pos = start + \frac{(key-A[start]) * (end-start)}{A[end] - A[start]}$$

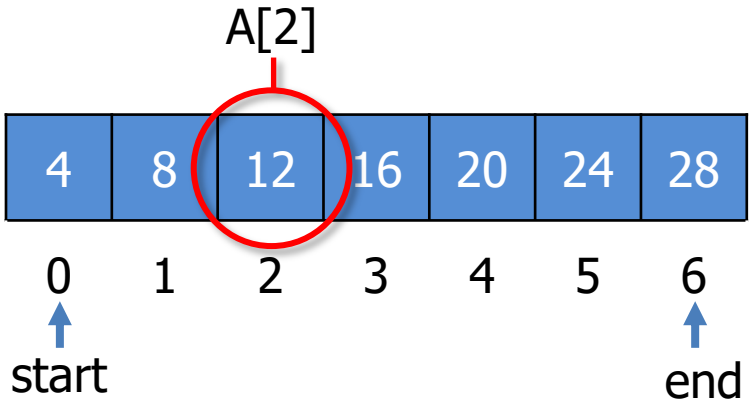
How Interpolation Search Works



1. Sorted and uniformly distributed sequence A
2. start = 0; end = n-1
3. $pos = start + \frac{(key - A[start]) * (end - start)}{A[end] - A[start]}$

Key=12

How Interpolation Search Works



$$pos = 0 + \frac{(12-4) * (6-0)}{28 - 4} = 2$$

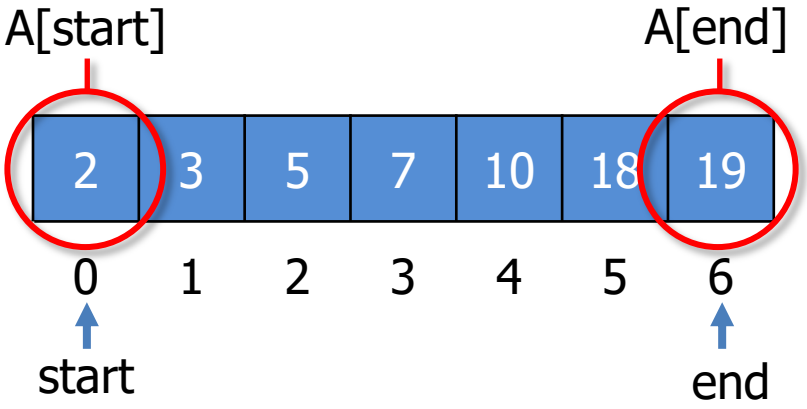
A[2] = 12; position found

1. Sorted and uniformly distributed sequence A
2. start = 0; end = n-1
3. $pos = start + \frac{(key-A[start]) * (end-start)}{A[end] - A[start]}$
4. if A[pos]==key, key is found at position pos

So…

- Interpolation Search on a sorted, uniformly distributed sequence is powerful!
- But what happens if the sequence is sorted, but NOT uniformly distributed?

Non uniform distribution

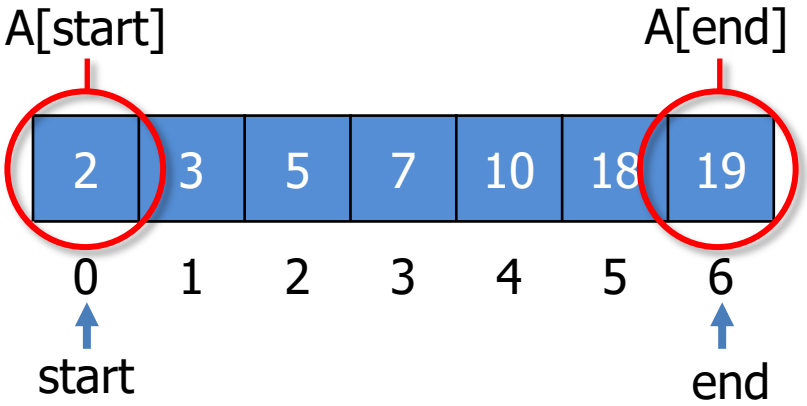


1. Sorted and non-uniformly distributed sequence A
2. start = 0; end = n-1
3. $pos = start + \frac{(key - A[start]) * (end - start)}{A[end] - A[start]}$

Key=10

4. if A[pos]==key, key is found at position pos
5. else if key > A[pos] start = pos+1
6. else if key < A[pos] end = pos-1

Non uniform distribution



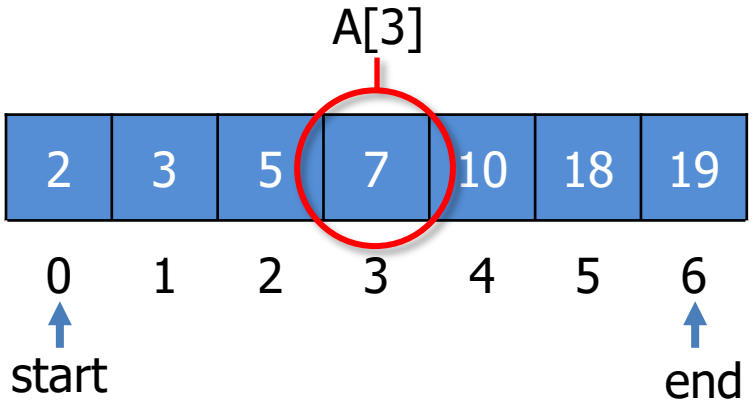
$$pos = 0 + \frac{(10-2) * (6-0)}{19 - 2} = 3 \text{ (2.82 rounded)}$$

1. Sorted and non-uniformly distributed sequence A
2. start = 0; end = n-1
3. $pos = start + \frac{(key-A[start]) * (end-start)}{A[end] - A[start]}$

Key=10

4. if A[pos]==key, key is found at position pos
5. else if key > A[pos] start = pos+1
6. else if key < A[pos] end = pos-1

Non uniform distribution

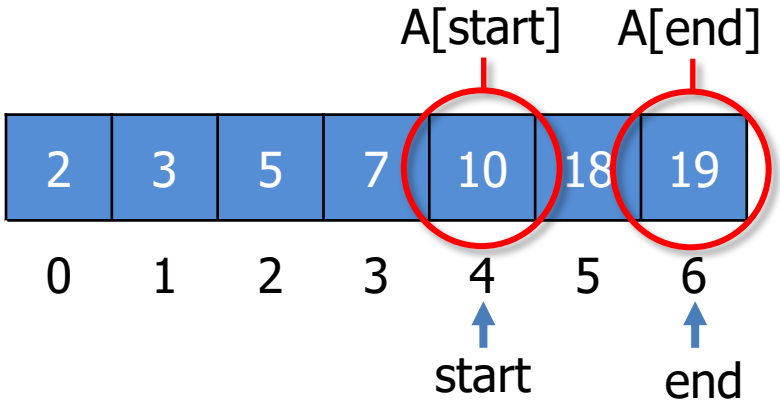


$$pos = 0 + \frac{(10-2) * (6-0)}{19 - 2} = 3 \text{ (2.82 rounded)}$$

A[3] = 7
Key 10 > A[3]
So start = 3+1 = 4

1. Sorted and non-uniformly distributed sequence A
2. start = 0; end = n-1
3. $pos = start + \frac{(key-A[start]) * (end-start)}{A[end] - A[start]}$ (Key=10)
4. if A[pos]==key, key is found at position pos
5. else if key > A[pos] start = pos+1
6. else if key < A[pos] end = pos-1

Non uniform distribution



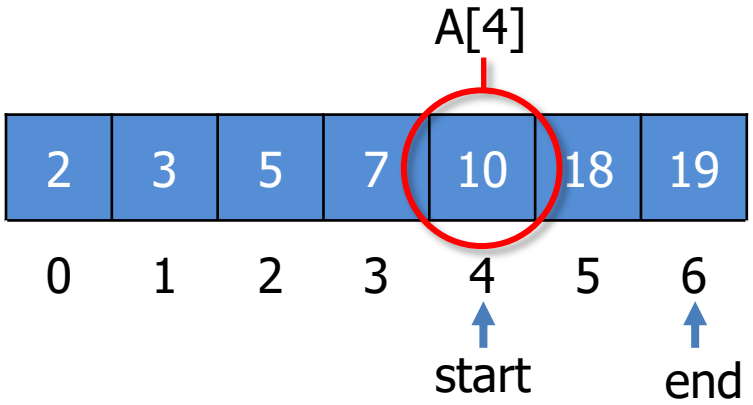
$$pos = 4 + \frac{(10-10) * (6-4)}{19 - 10} = 4$$

1. Sorted and non-uniformly distributed sequence A
2. start = 0; end = n-1
3. $pos = start + \frac{(key-A[start]) * (end-start)}{A[end] - A[start]}$

Key=10

4. if A[pos]==key, key is found at position pos
5. else if key > A[pos] start = pos+1
6. else if key < A[pos] end = pos-1

Non uniform distribution



$$pos = 4 + \frac{(10-10) * (6-4)}{19 - 10} = 4$$

A[4] = 10

1. Sorted and non-uniformly distributed sequence A
2. start = 0; end = n-1
3. $pos = start + \frac{(key-A[start]) * (end-start)}{A[end] - A[start]}$ (Key=10)
4. if A[pos]==key, key is found at position pos
5. else if key > A[pos] start = pos+1
6. else if key < A[pos] end = pos-1

Iterative Interpolation Search Pseudocode

```
INTERPOLATION_SEARCH(A, key):
    start ← 0
    end ← len(A) – 1
    WHILE A[end] != A[start] AND key >= A[start] AND key <= A[end]
        pos ← ROUND(start + ((key – A[start]) * (end – start) / (A[end] – A[start])))
        IF A[pos] == key
            RETURN pos
        ELSE IF (key < A[pos])
            start ← pos + 1
        ELSE
            end ← pos – 1
    RETURN FALSE                          //if the value is not in the sequence
```

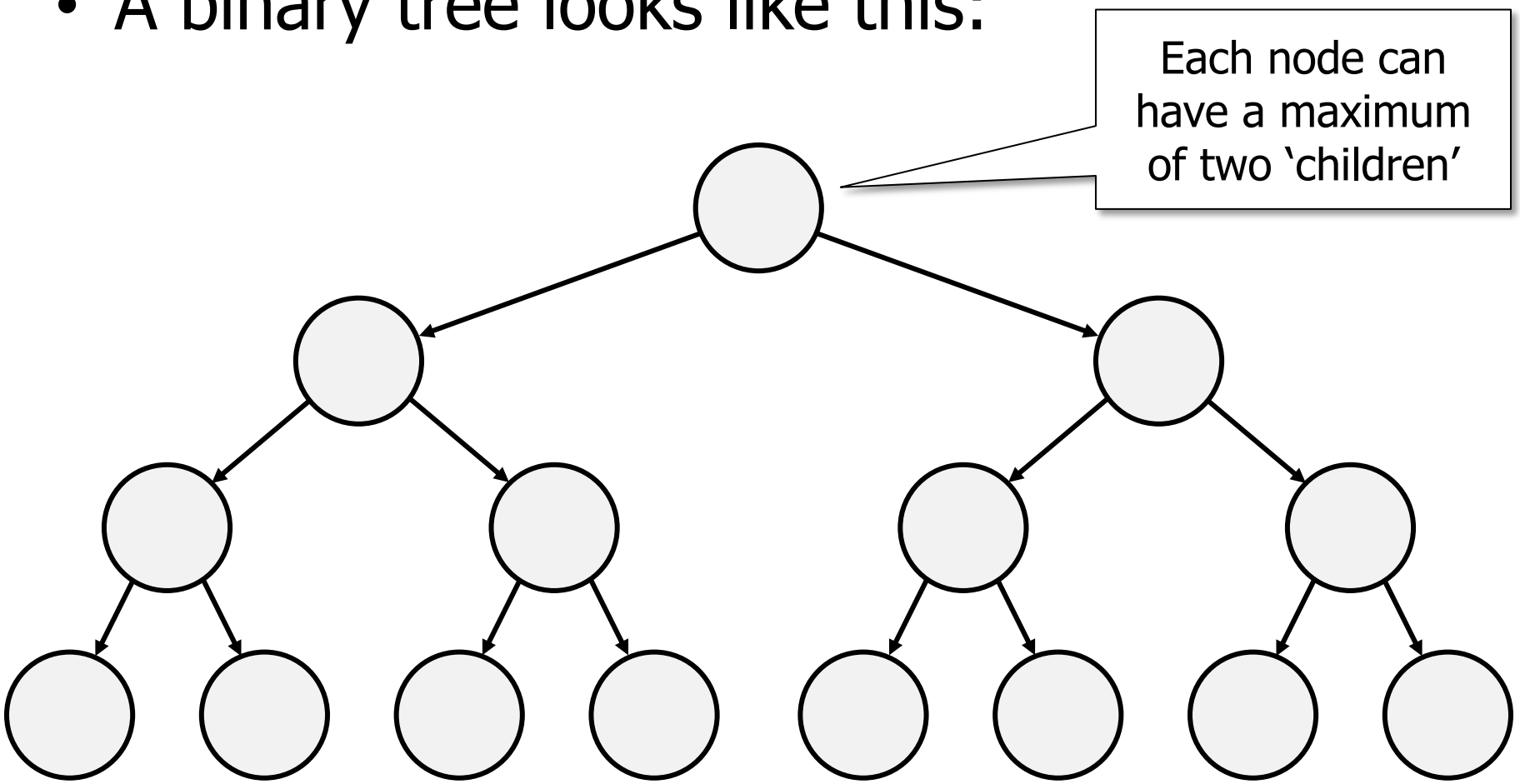
- See if you can work out a recursive version!

Data Structures

- We've seen that both Binary Search and Interpolation Search need a sorted list
- Other search algorithms need data structures that are organised in more complex ways
- To make a start on this, we will now look at **heap sort**
- Which in turn requires a quick intro to **binary trees**

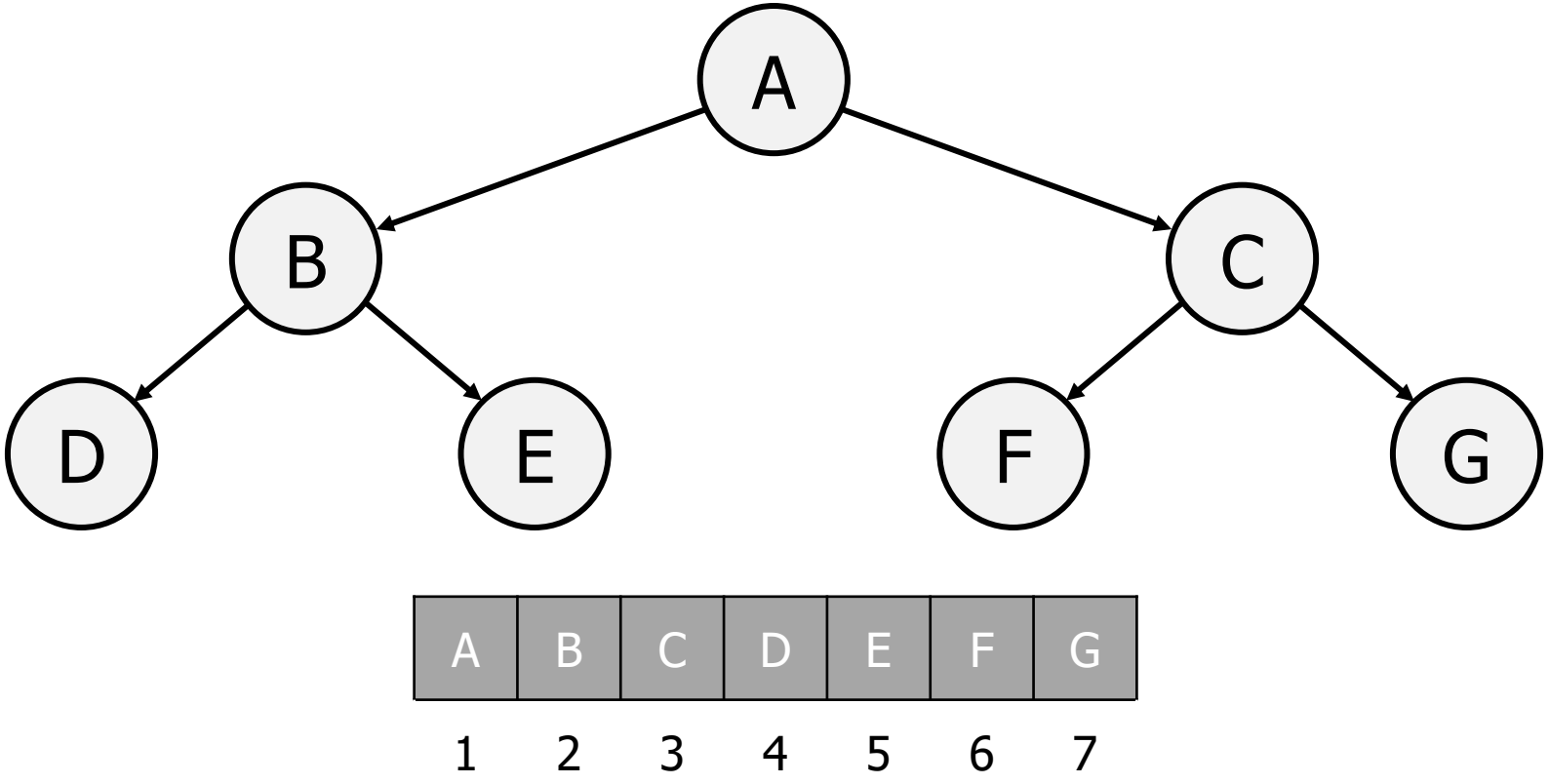
Binary Trees

- A binary tree looks like this:

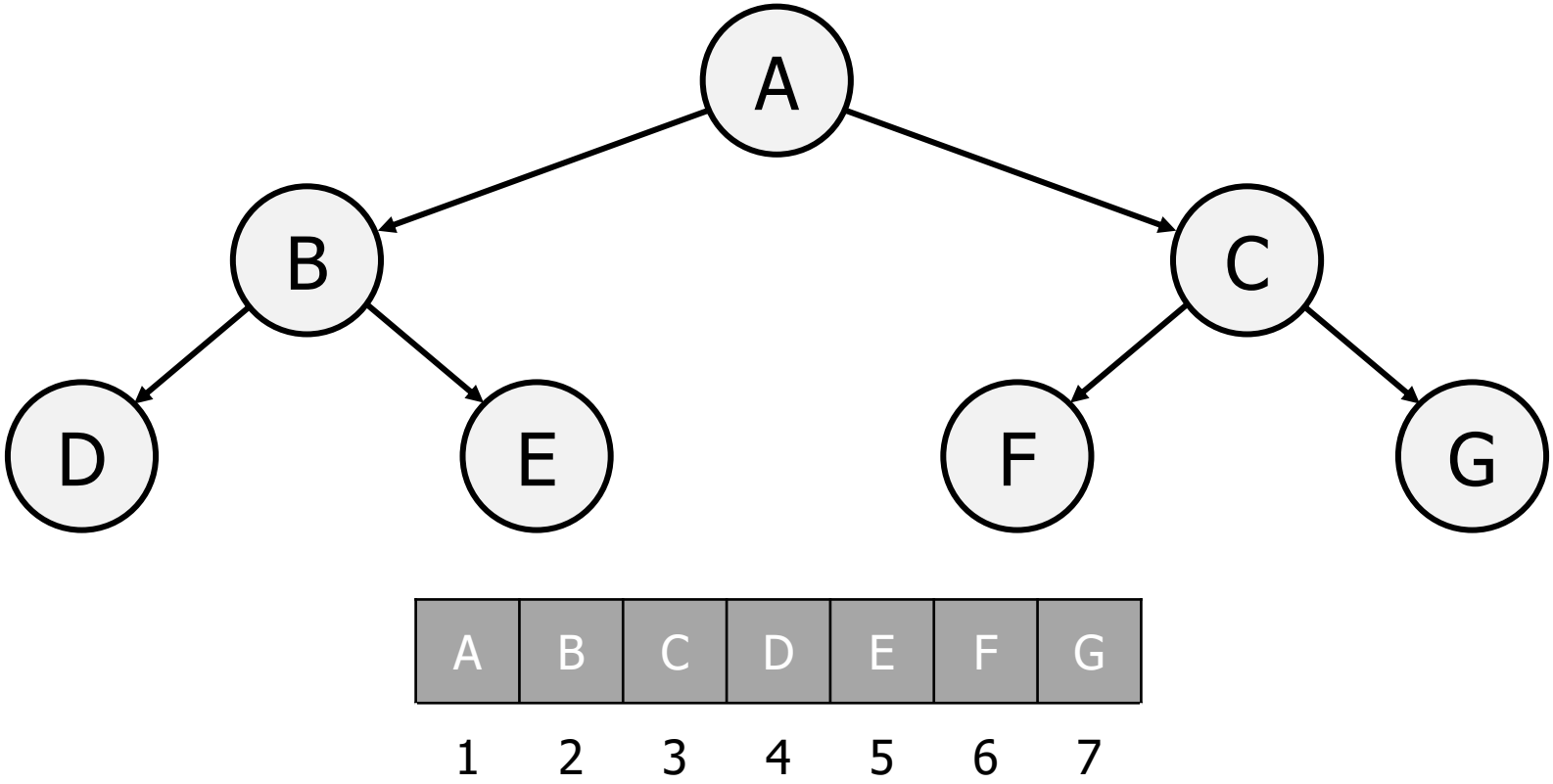


Binary Trees and Lists

- Trees relate to lists like this:



Some rules



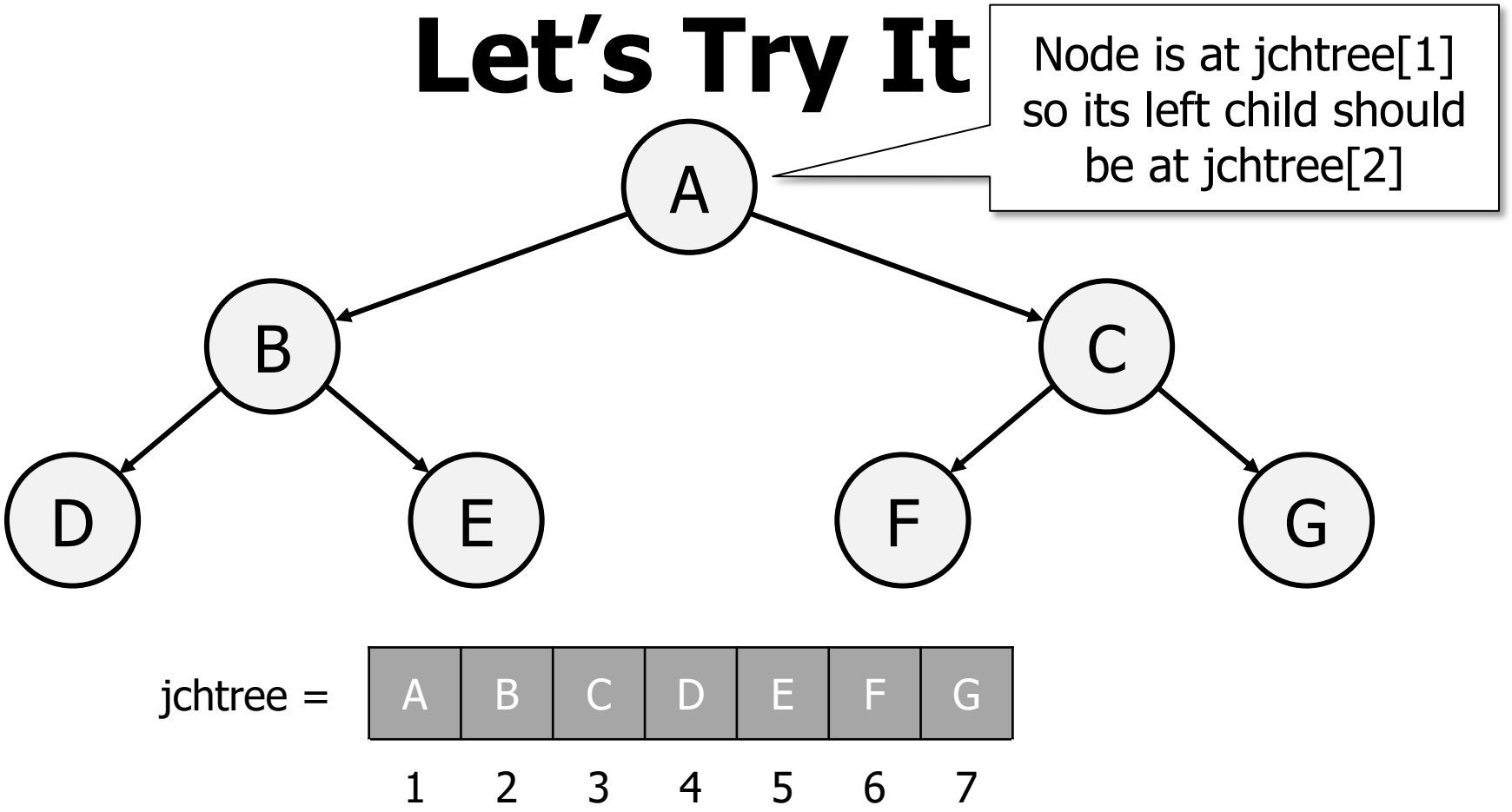
If a node is at index i:

its left child is at index 2 * i

its right child is at index 2 * i + 1

its parent is at i // 2

Let's Try It



If a node is at index i:

its left child is at index 2 * i

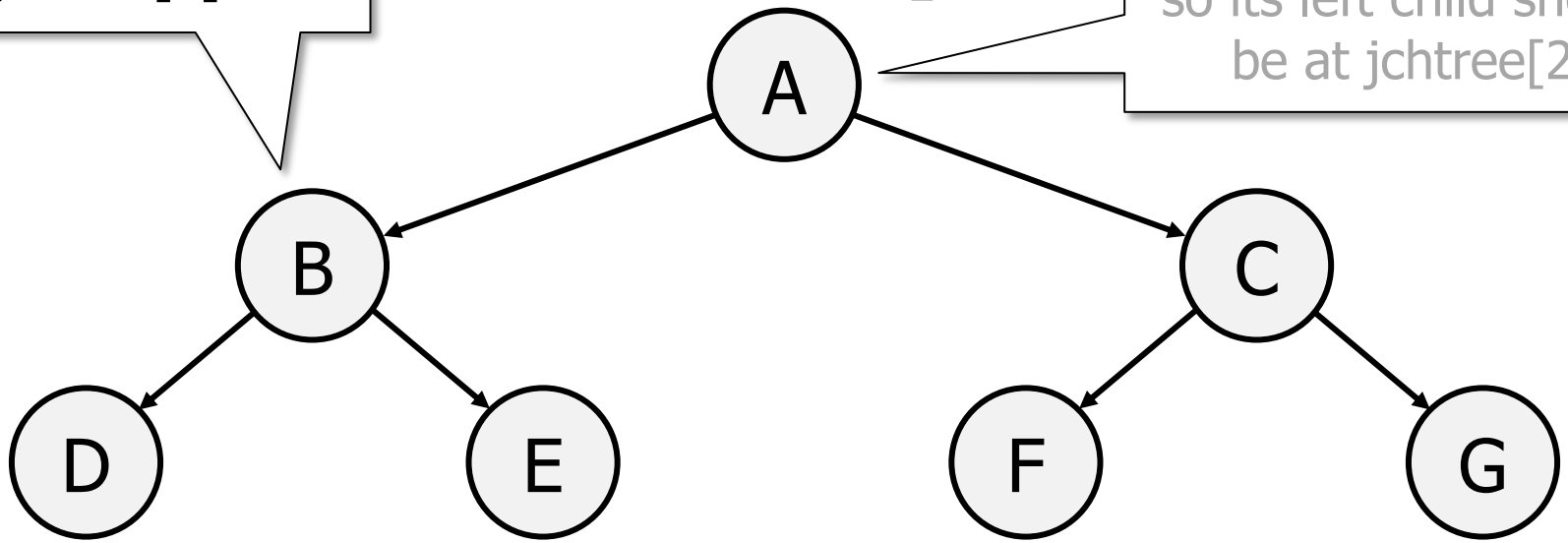
its right child is at index 2 * i + 1

its parent is at i // 2

Let's Try It

jchtree[1]==A
jchtree[2]==B

Node is at jchtree[1] so its left child should be at jchtree[2]



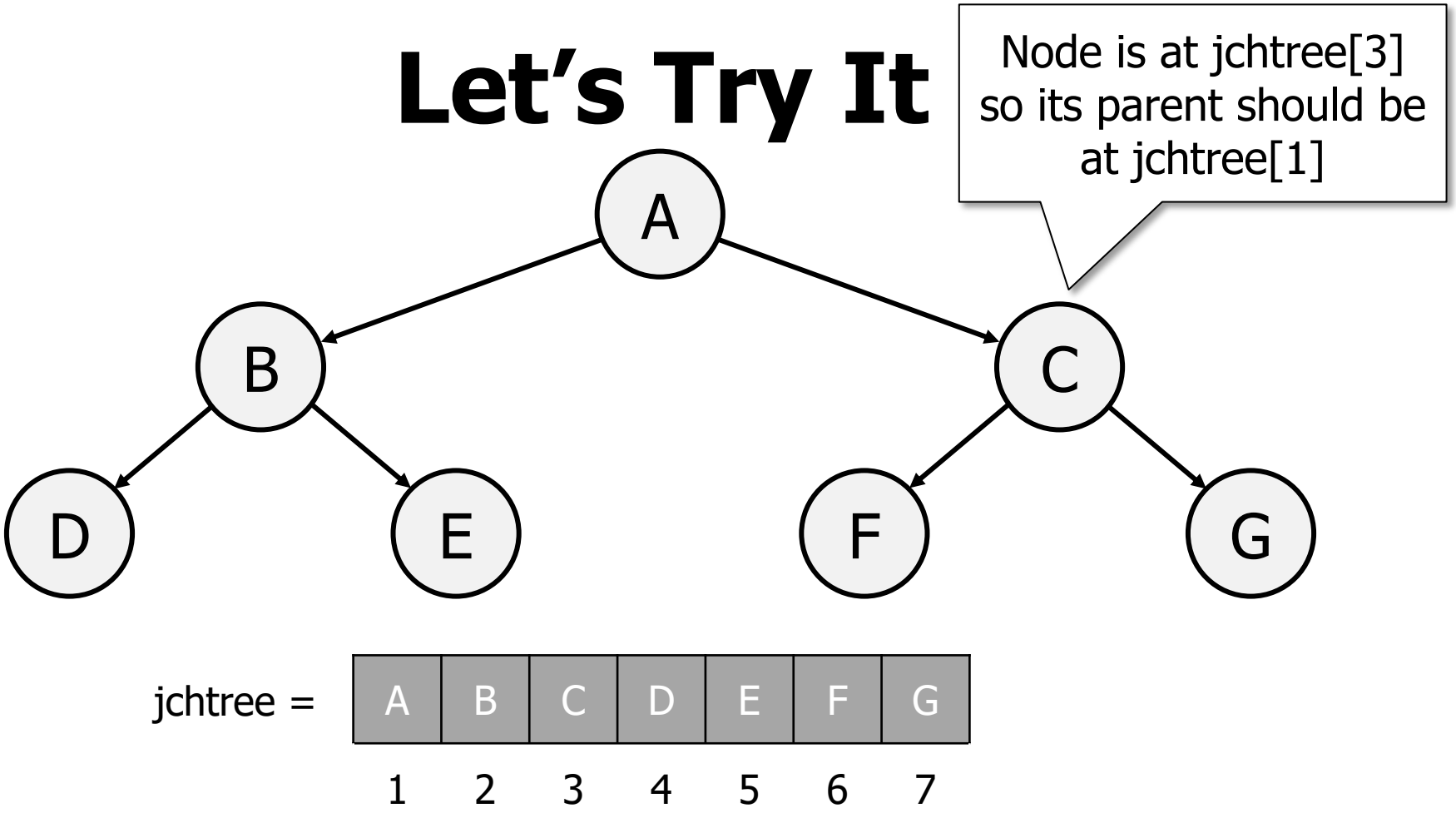
jchtree =

A	B	C	D	E	F	G
1	2	3	4	5	6	7

If a node is at index i:

its left child is at index 2 * i
its right child is at index 2 * i + 1
its parent is at i // 2

Let's Try It



If a node is at index i:

its left child is at index 2 * i

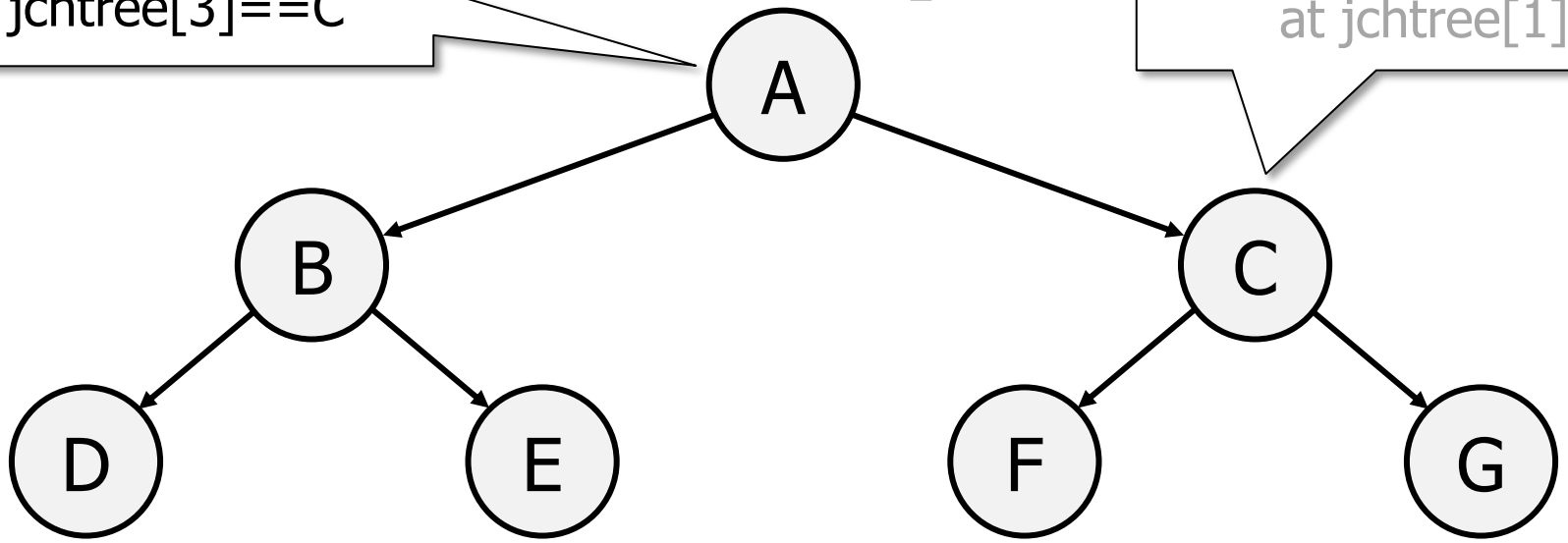
its right child is at index 2 * i + 1

its parent is at i // 2

Let's Try It

jchtree[1]==A, parent of jchtree[3]==C

Node is at jchtree[3] so its parent should be at jchtree[1]



jchtree =

1	2	3	4	5	6	7
A	B	C	D	E	F	G

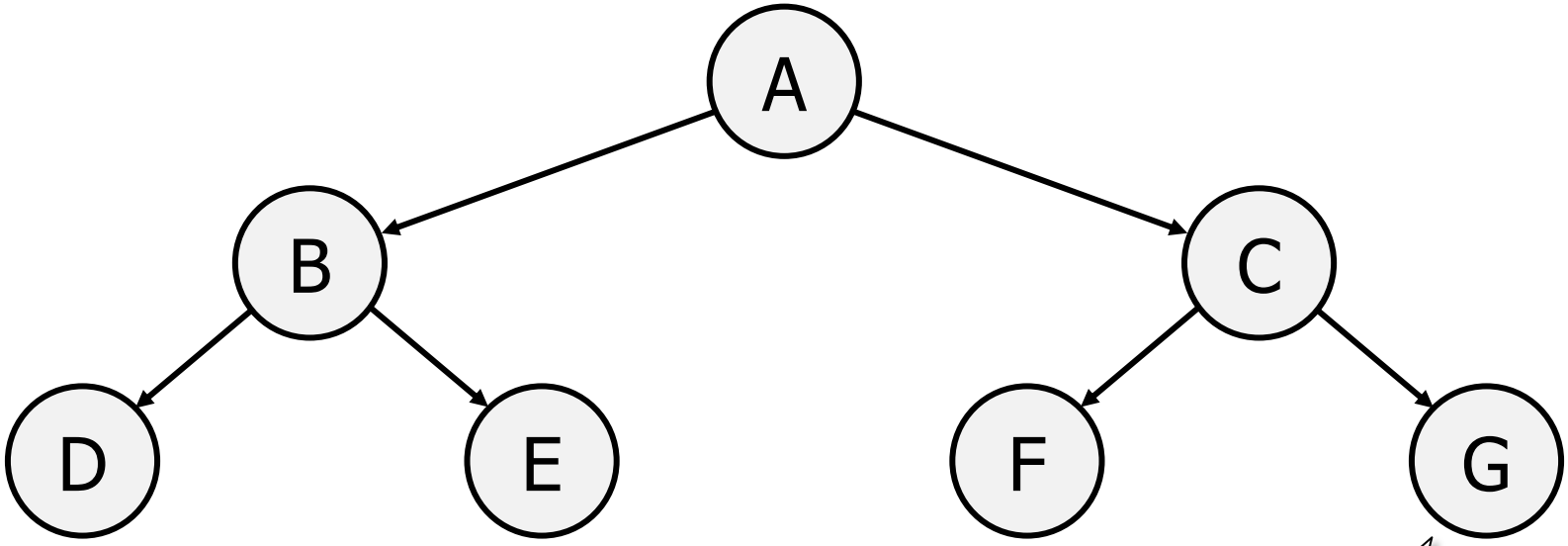
If a node is at index i:

its left child is at index 2 * i

its right child is at index 2 * i + 1

its parent is at i // 2

Let's Try It



jchtree =

A	B	C	D	E	F	G
1	2	3	4	5	6	7

Node is at jchtree[7] so its right child should be at jchtree[15]

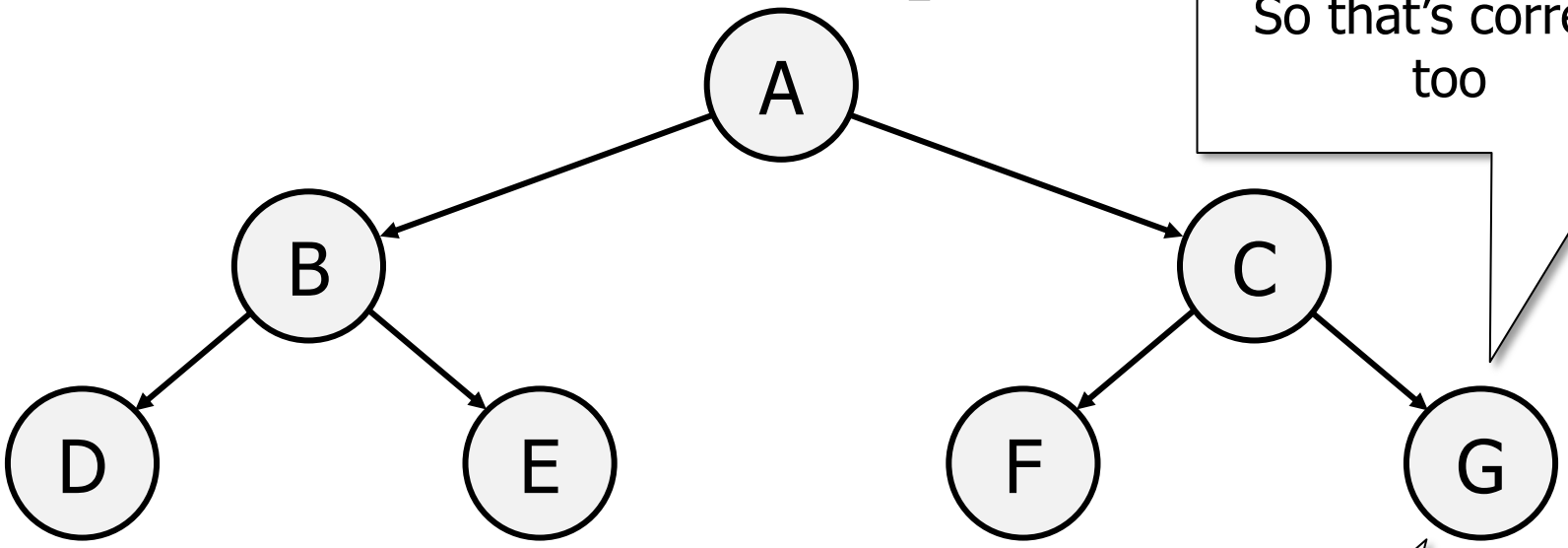
If a node is at index i:

its left child is at index 2 * i

its right child is at index 2 * i + 1

its parent is at i // 2

Let's Try It



But there is no jchtree[15]! So that's correct, too

jchtree =

A	B	C	D	E	F	G
1	2	3	4	5	6	7

Node is at jchtree[7] so its right child should be at jchtree[15]

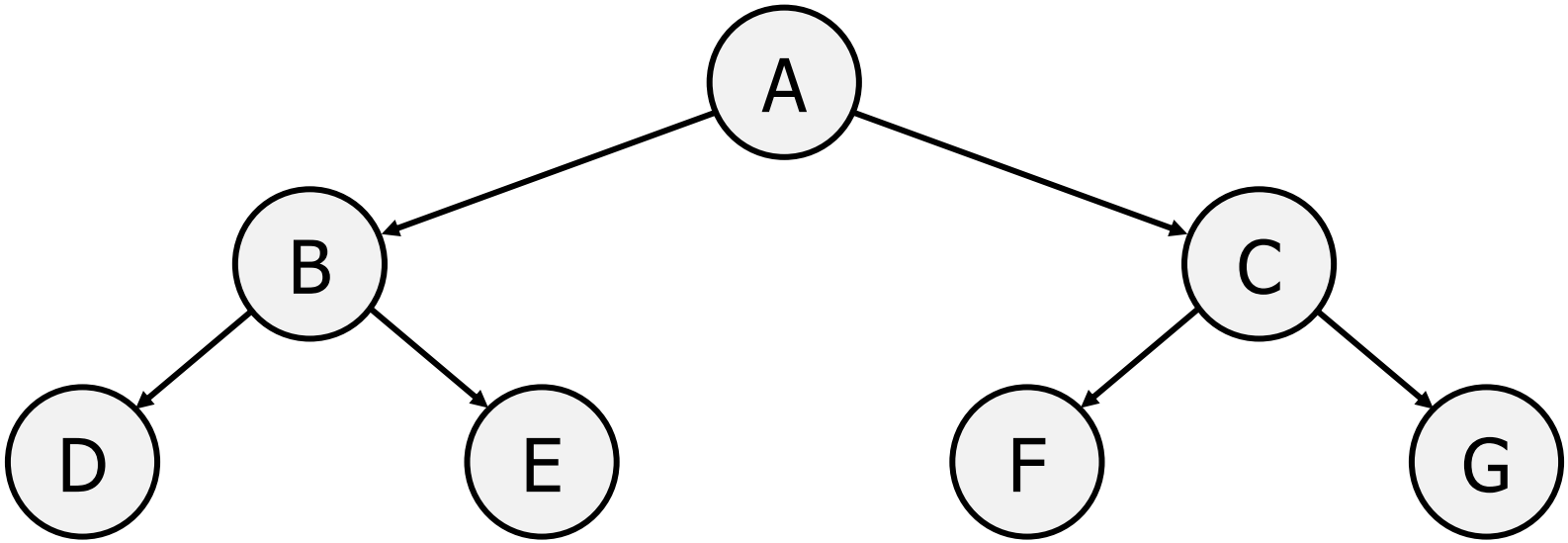
If a node is at index i:

its left child is at index 2 * i

its right child is at index 2 * i + 1

its parent is at i // 2

You can try out other for examples yourself…



jchtree =

A	B	C	D	E	F	G
1	2	3	4	5	6	7

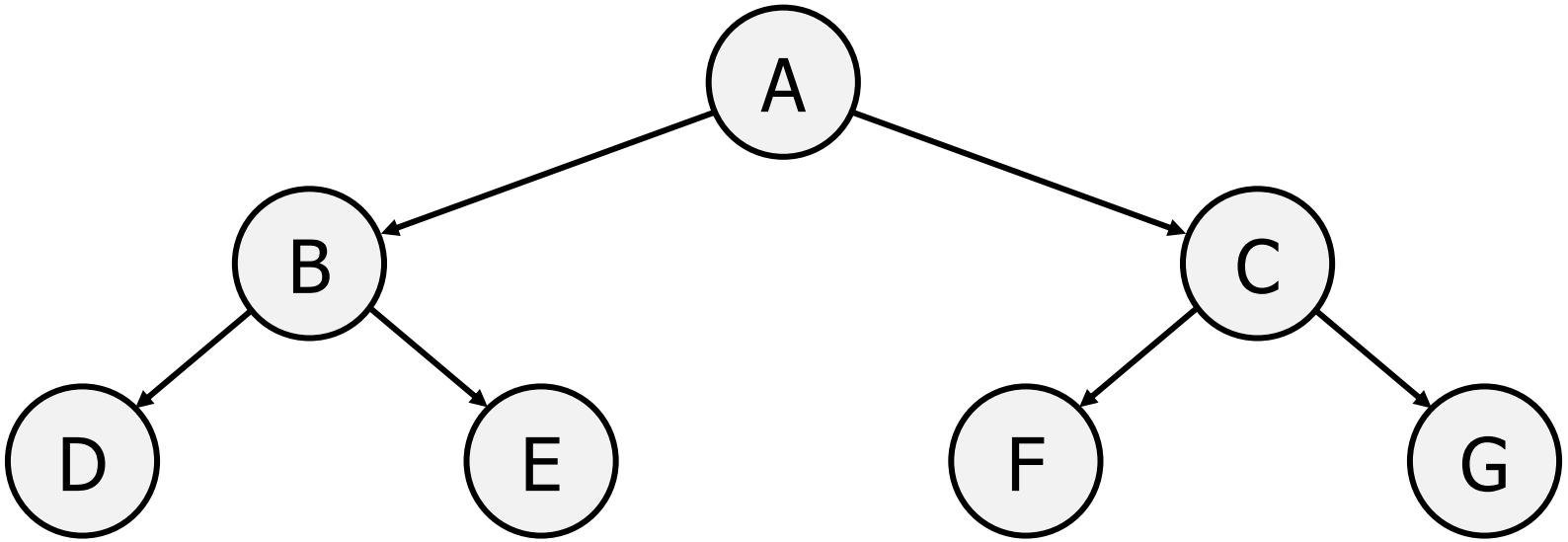
If a node is at index i:

its left child is at index 2 * i

its right child is at index 2 * i + 1

its parent is at i // 2

The rules relating binary trees to lists are important for understanding Heap Sort…



jchtree =

A	B	C	D	E	F	G
1	2	3	4	5	6	7

If a node is at index i:

its left child is at index 2 * i

its right child is at index 2 * i + 1

its parent is at i // 2

Heaps

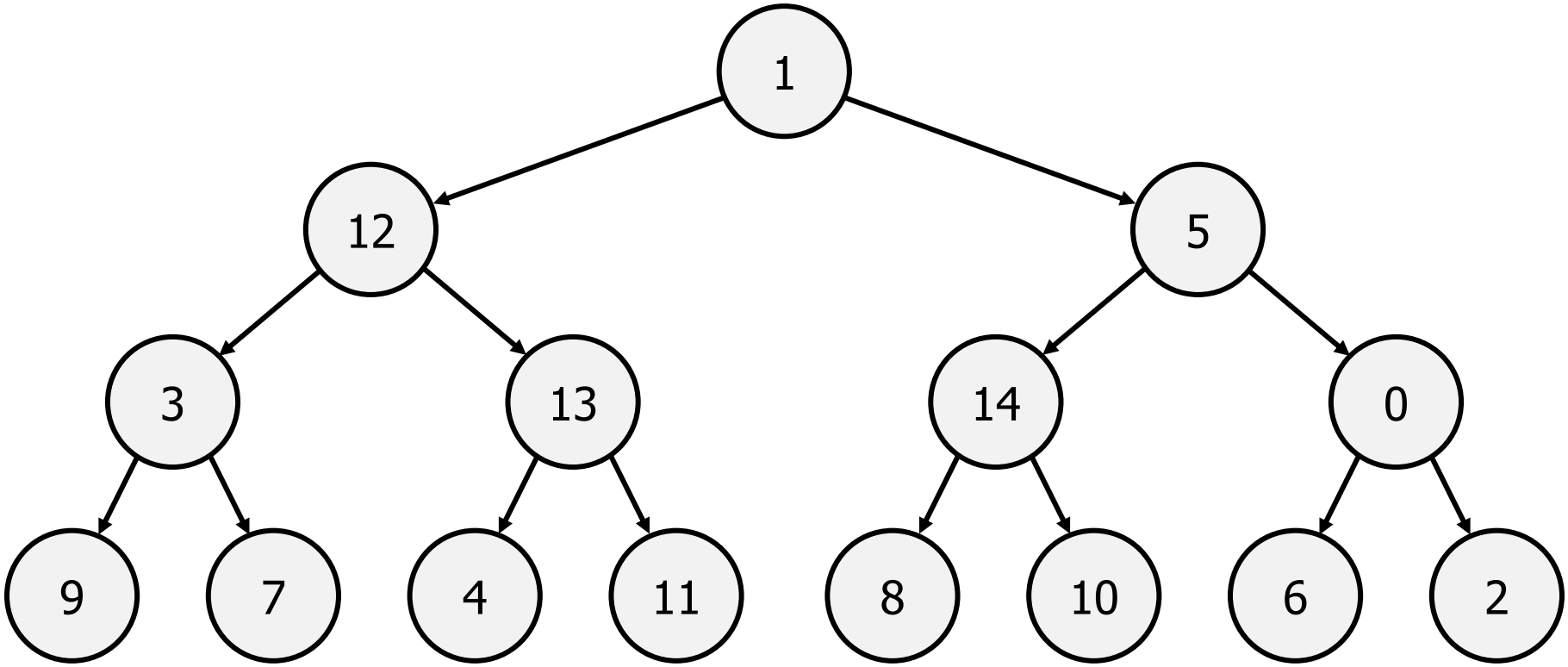
- A heap is a binary tree where each node is greater than or equal to each of its children *
- Like any binary tree, each parent can have 0, 1 or 2 children – but no more

* This is true of max heap, which we are looking at here. Min heap is the opposite: each child is equal to or greater than the its parent

Heap Sort

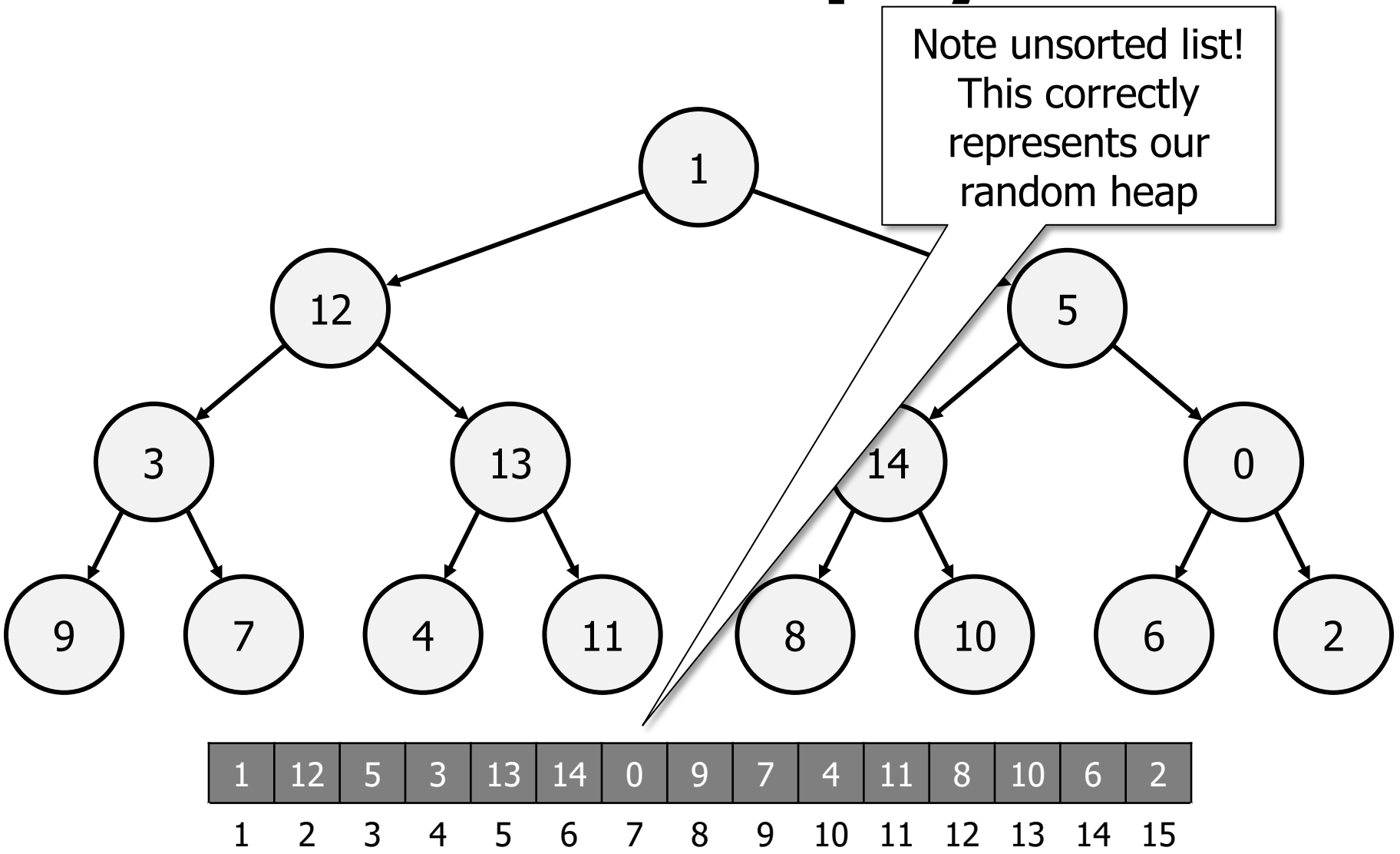
- Now we will look at how heap sort works
- It uses a binary tree, which is 'heapified'
- But heapification **does not sort**!
- So there are two processes:
    1. Heapify
    2. Sort
- We start with (1) Heapify…
- … using a random set of values

Let's Heapify

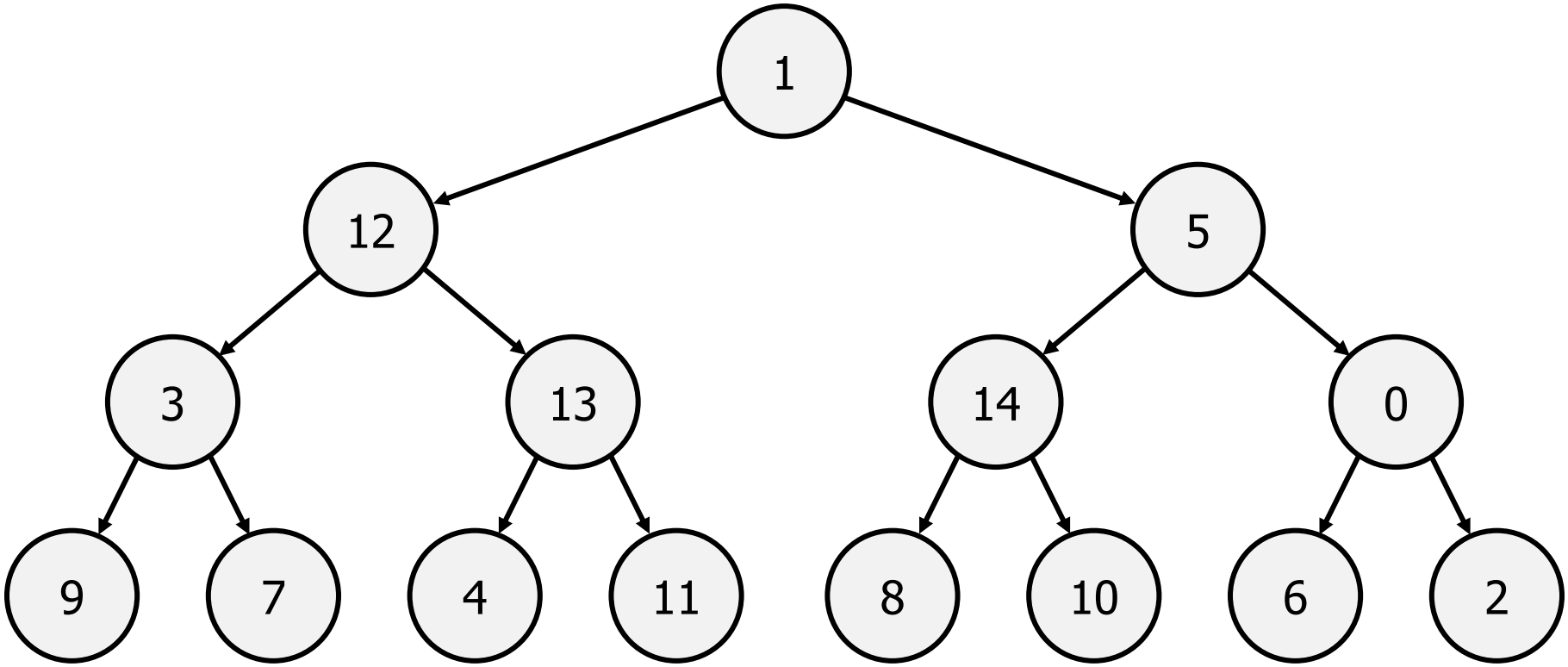


1	12	5	3	13	14	0	9	7	4	11	8	10	6	2
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Let's Heapify



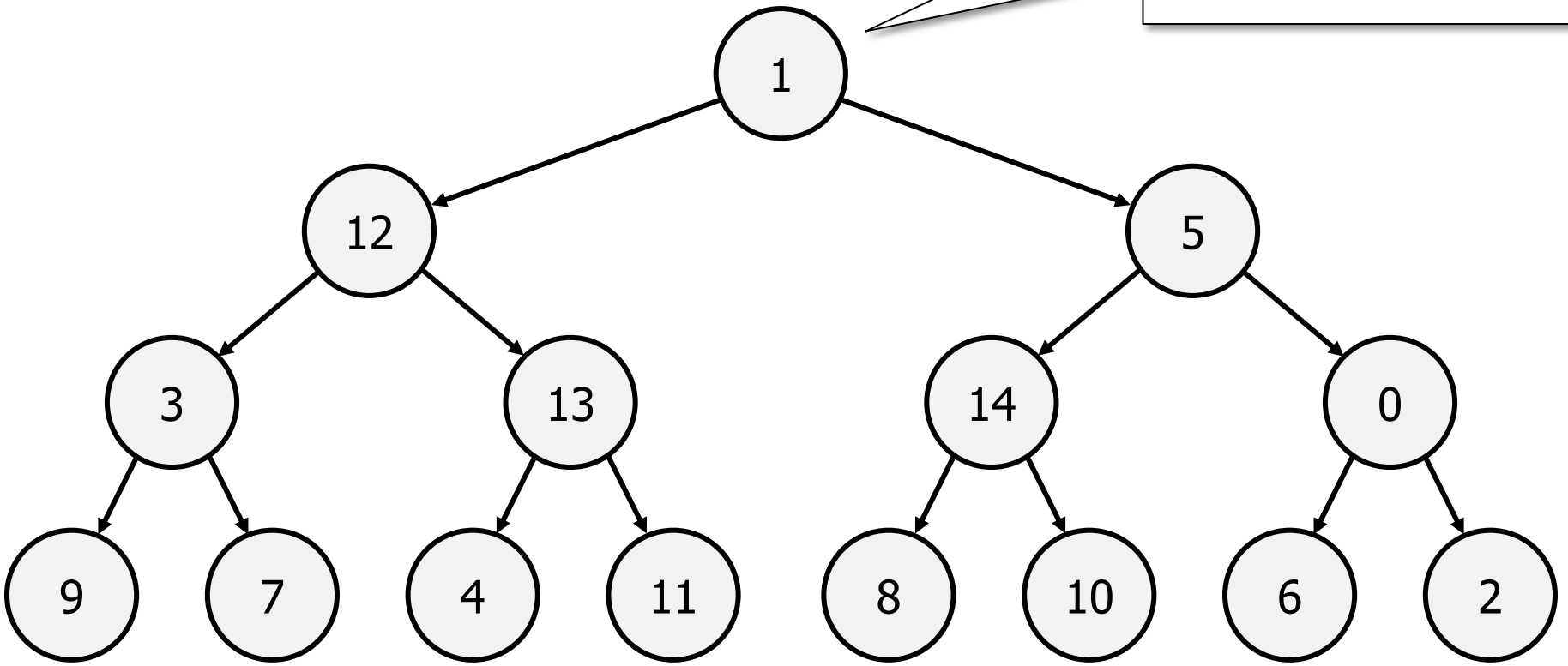
How it works



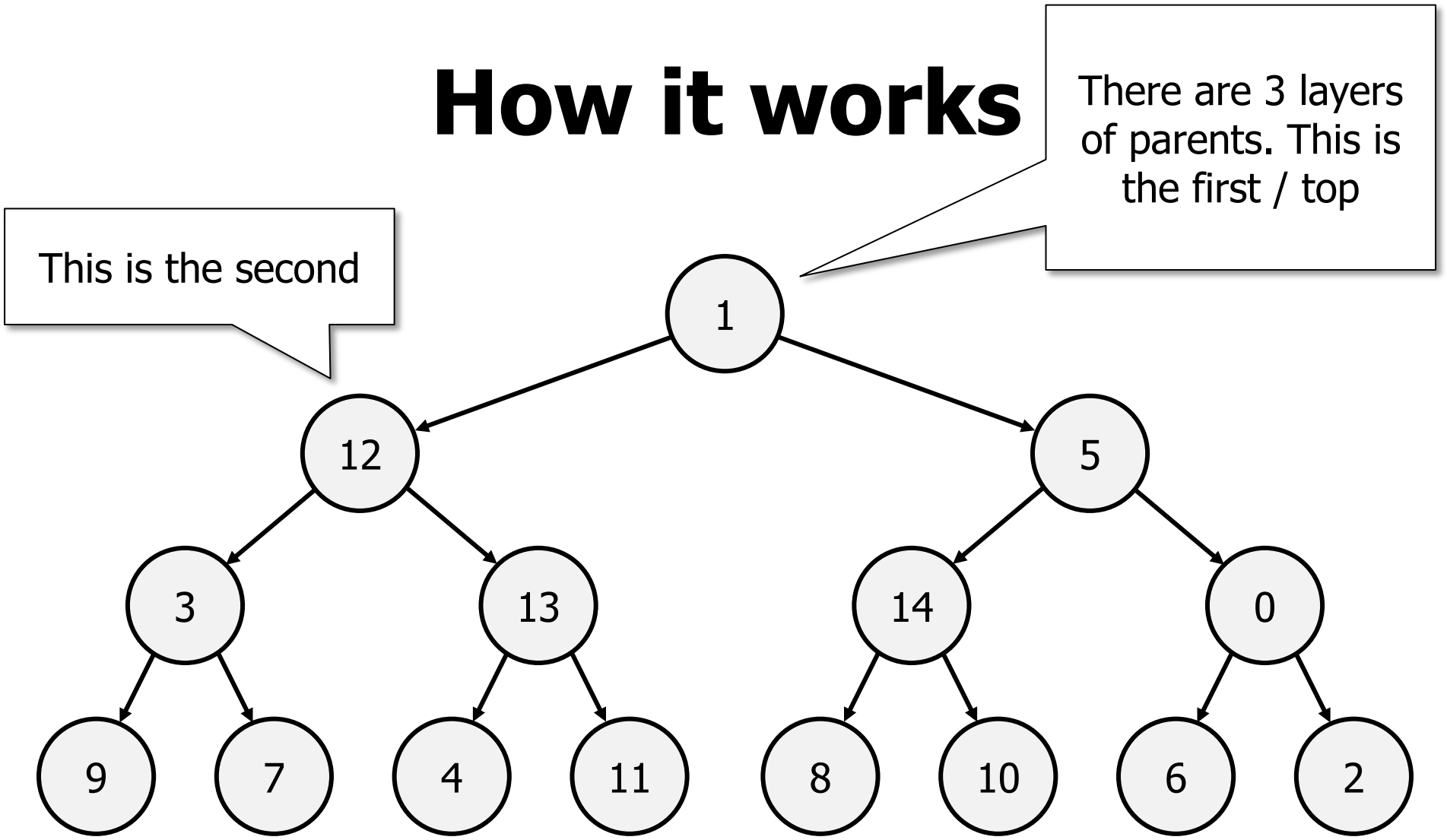
As we go through Heapify we are not going to worry about what happens to our list; we'll come back to it at the end

How it works

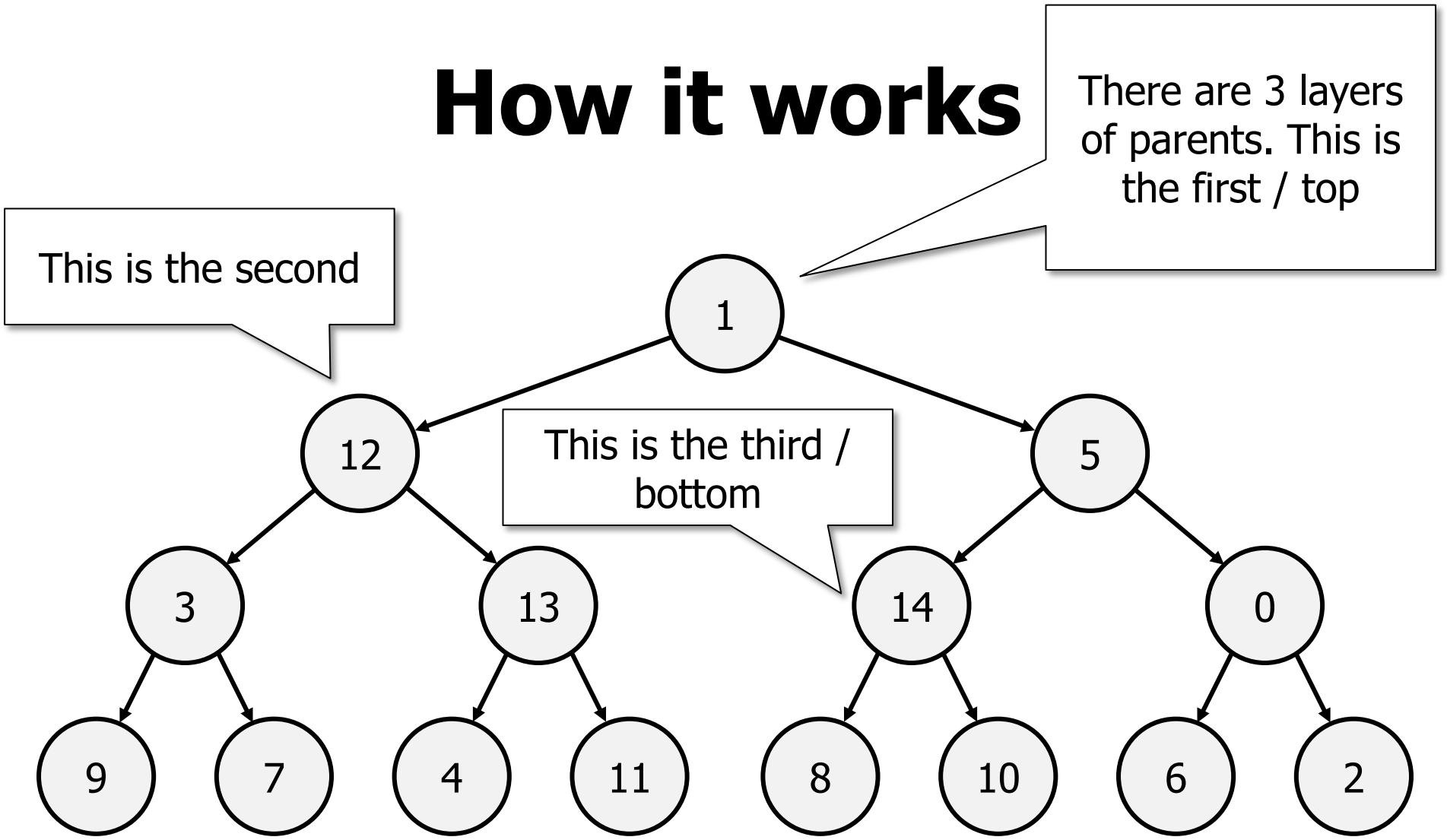
There are 3 layers of parents. This is the first / top



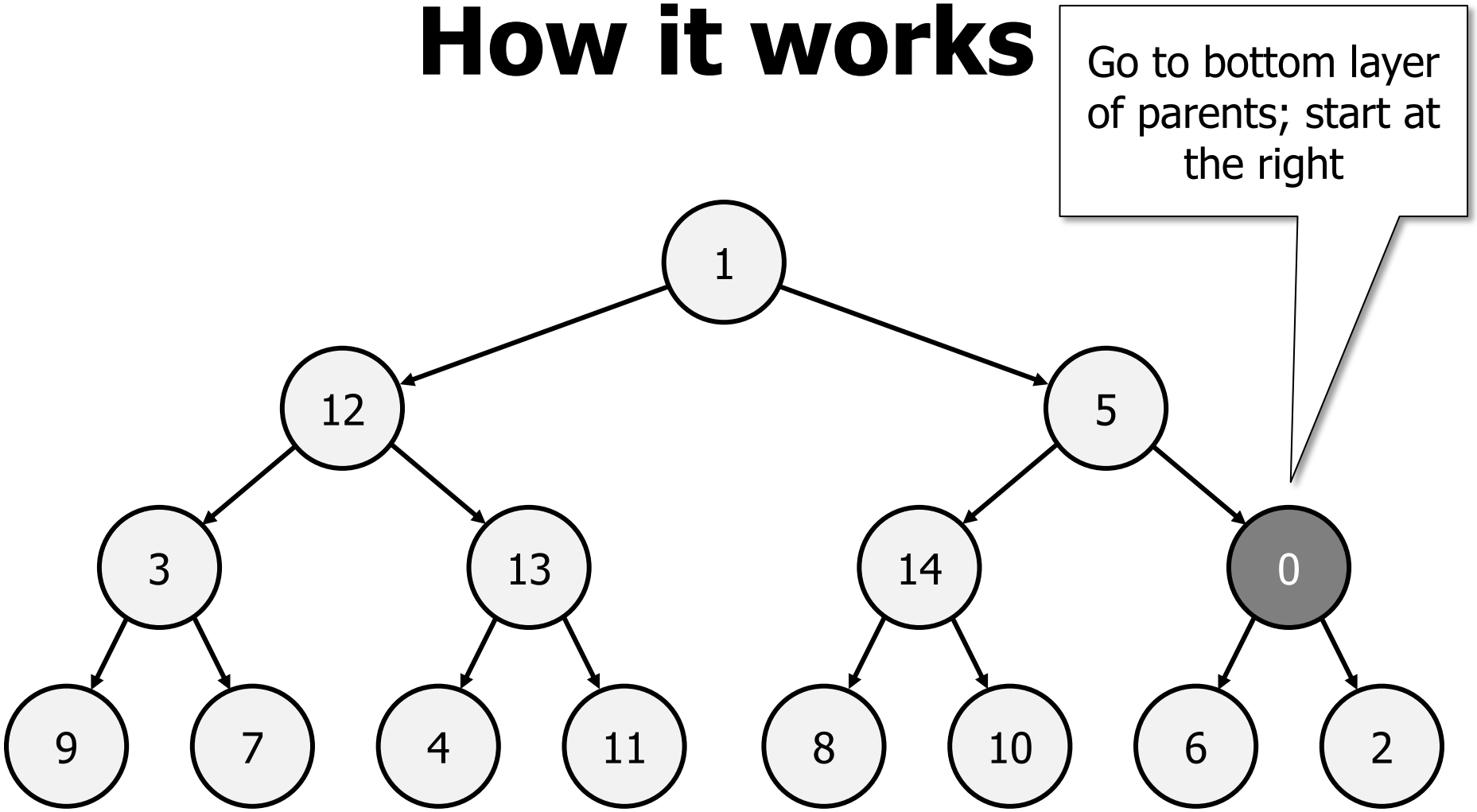
How it works


There are 3 layers of parents. This is the first / top
This is the second
1
12
5
3
13
14
0
9
7
4
11
8
10
6
2

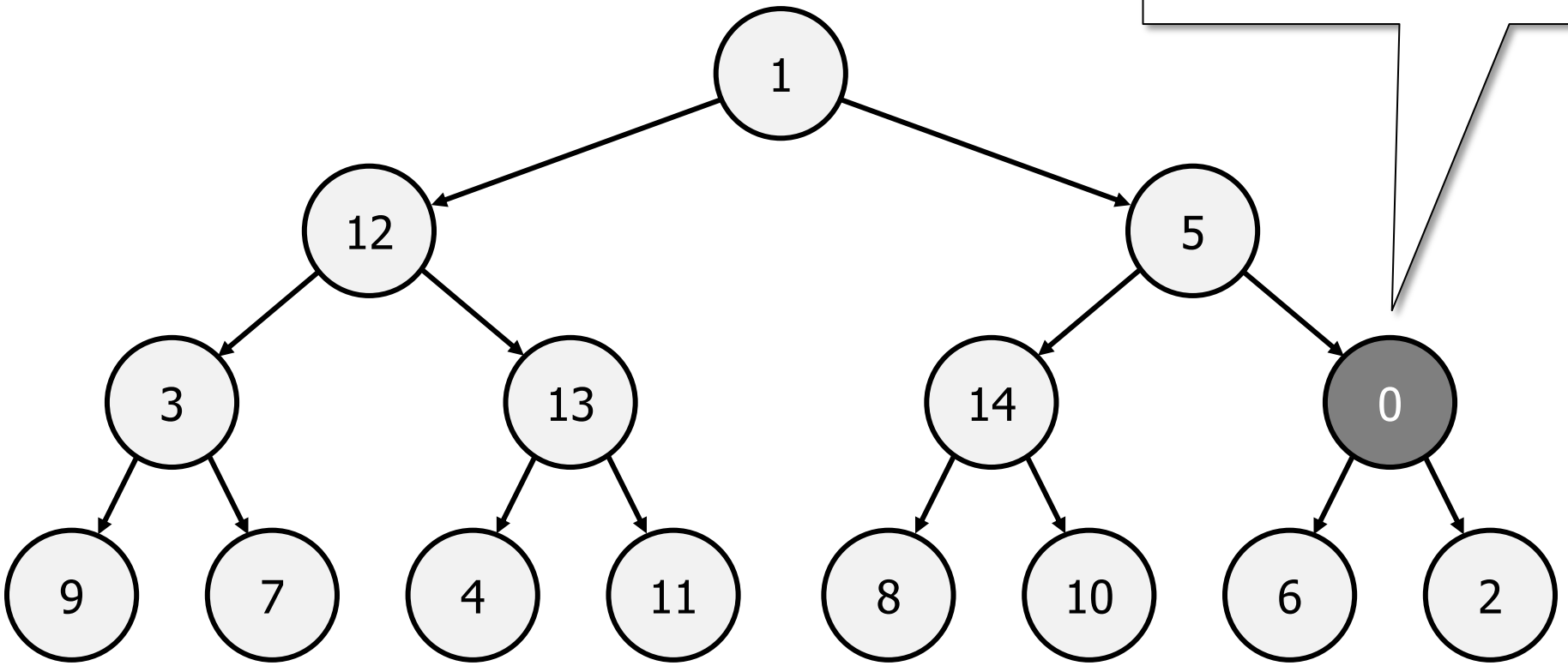
How it works


There are 3 layers of parents. This is the first / top
This is the second
This is the third / bottom
1
12
5
3
13
14
0
9
7
4
11
8
10
6
2

How it works


Go to bottom layer of parents; start at the right
1
12
5
3
13
14
0
9
7
4
11
8
10
6
2

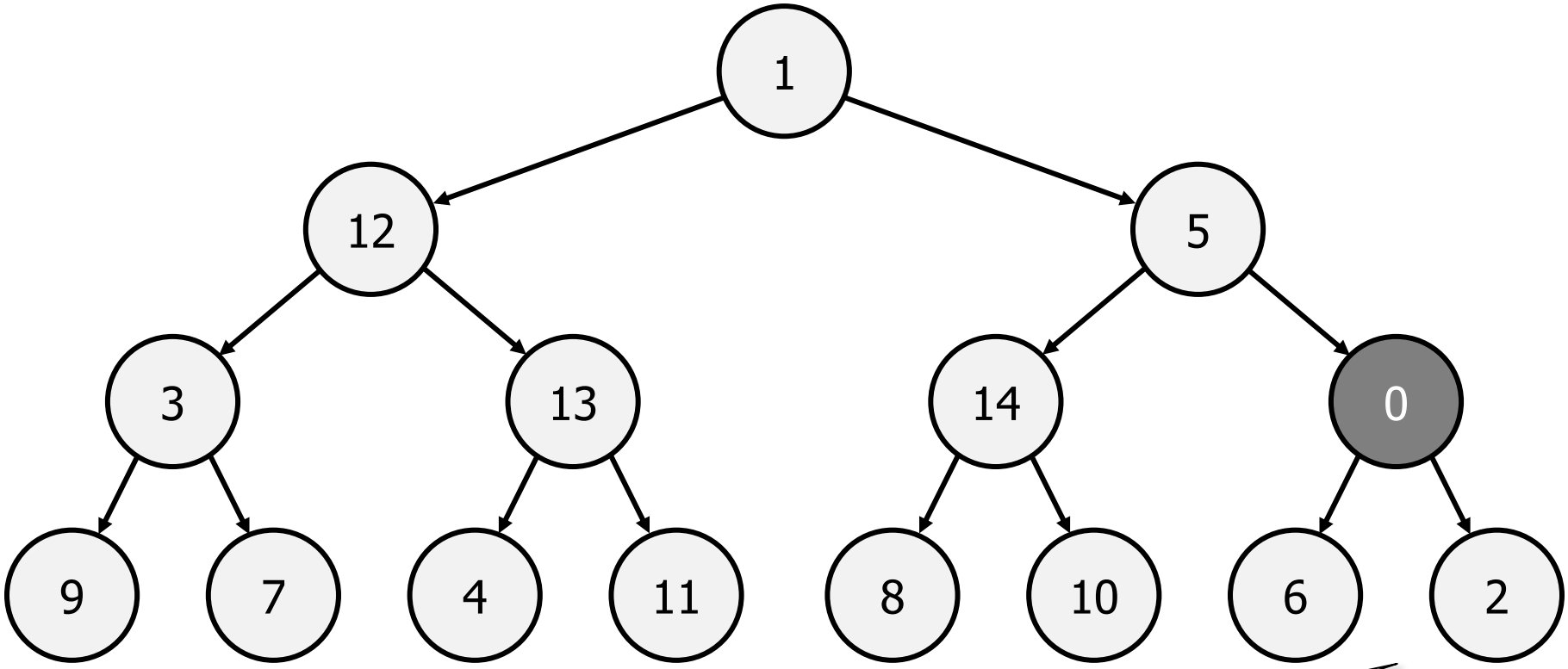
How it works



Go to bottom layer of parents; start at the right

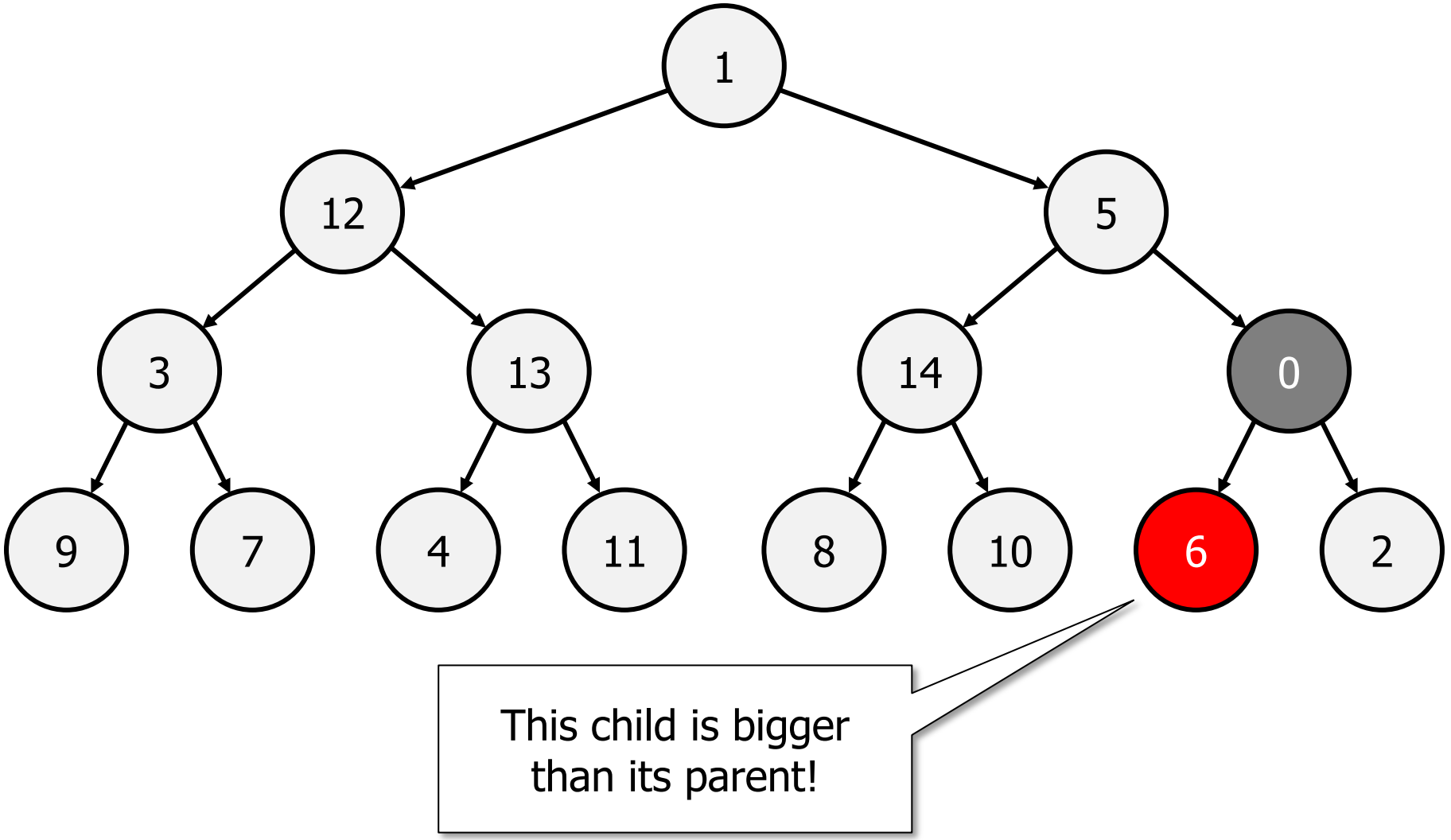
Rule is: right to left, bottom to top

How it works

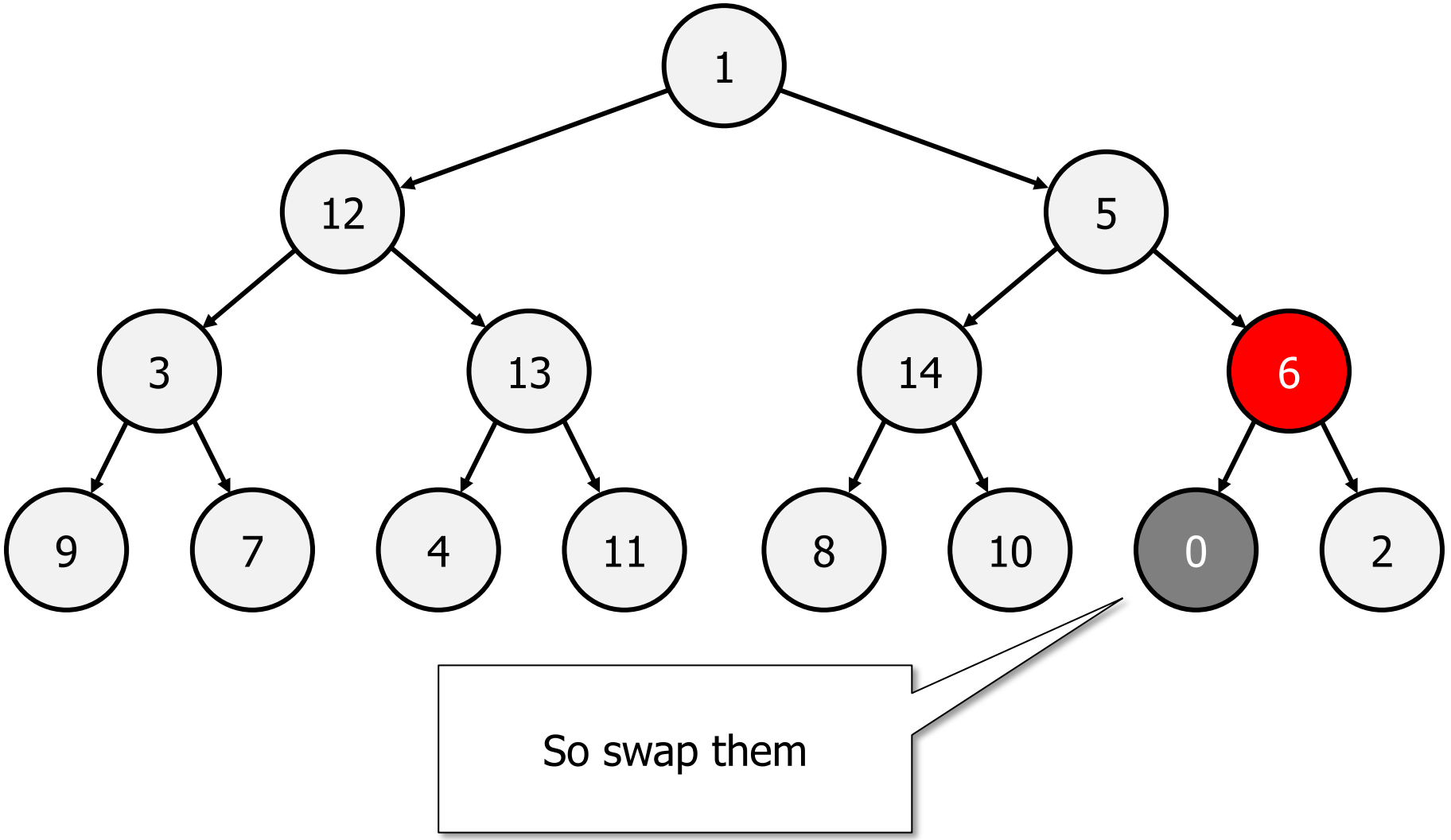


Check which child is bigger

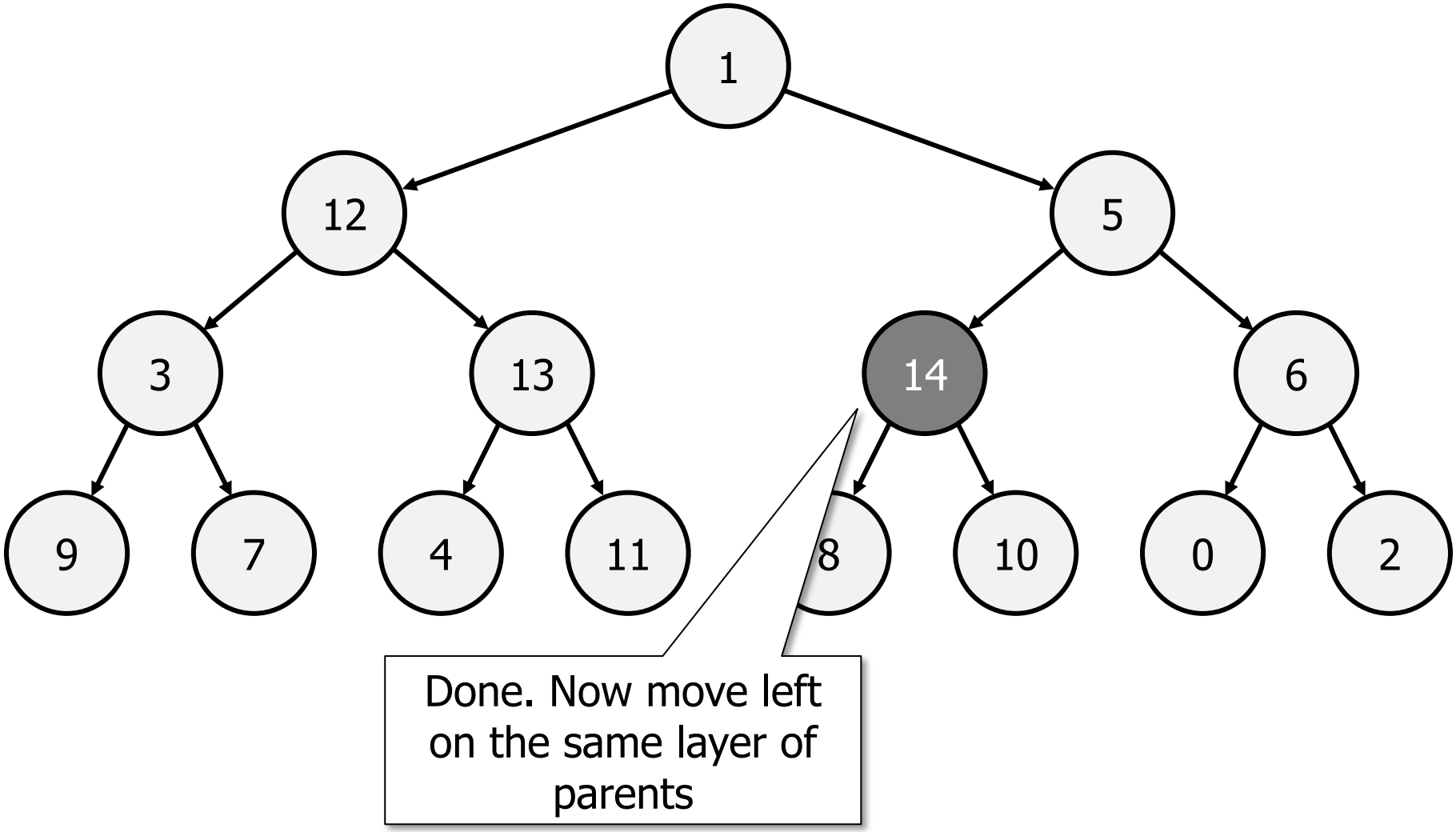
How it works


1
12
5
3
13
14
0
9
7
4
11
8
10
6
2
This child is bigger
than its parent!

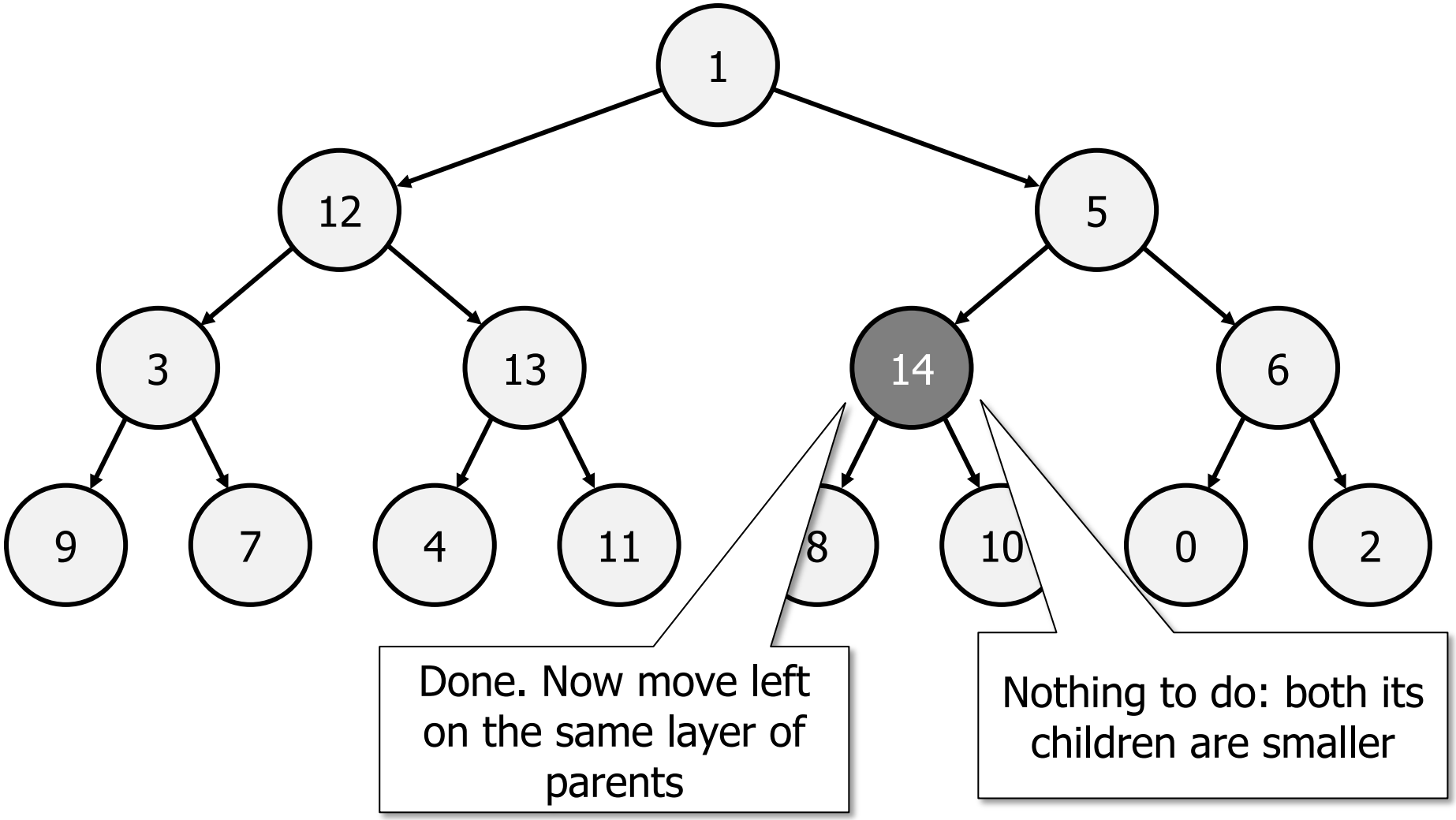
How it works



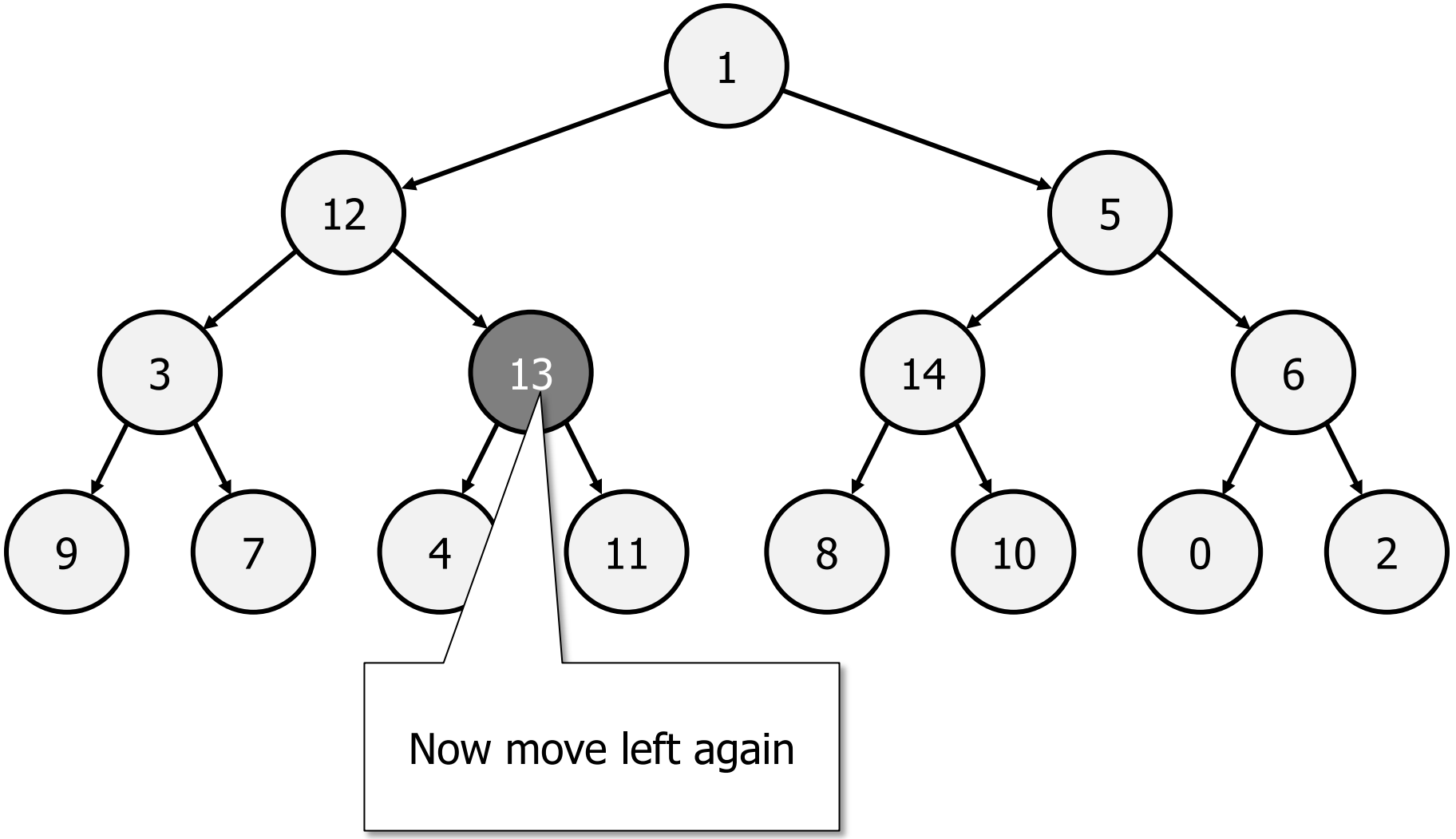
How it works


1
12
5
3
13
14
6
9
7
4
11
8
10
0
2
Done. Now move left on the same layer of parents

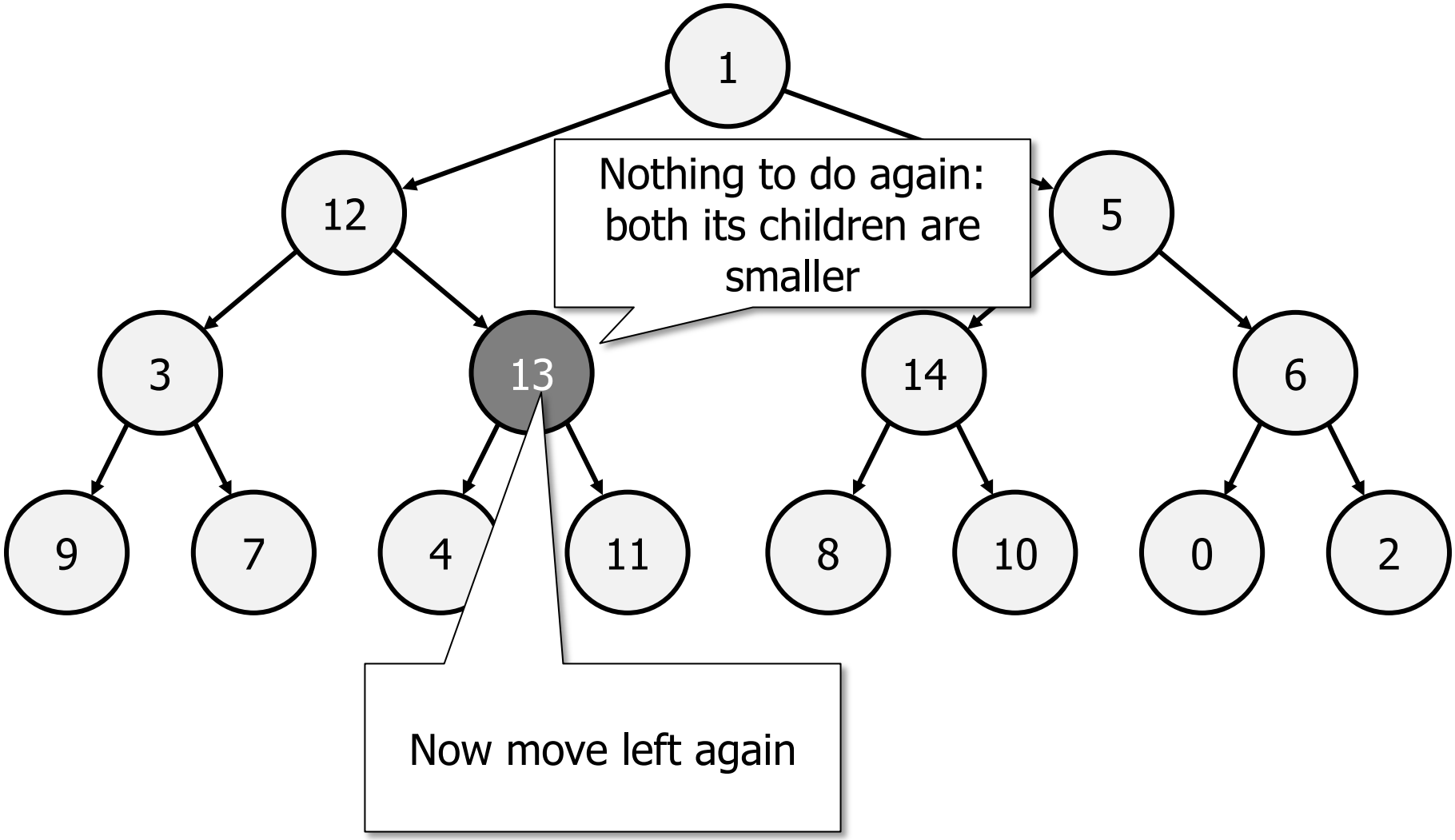
How it works


1
12
5
3
13
14
6
9
7
4
11
8
10
0
2
Done. Now move left on the same layer of parents
Nothing to do: both its children are smaller

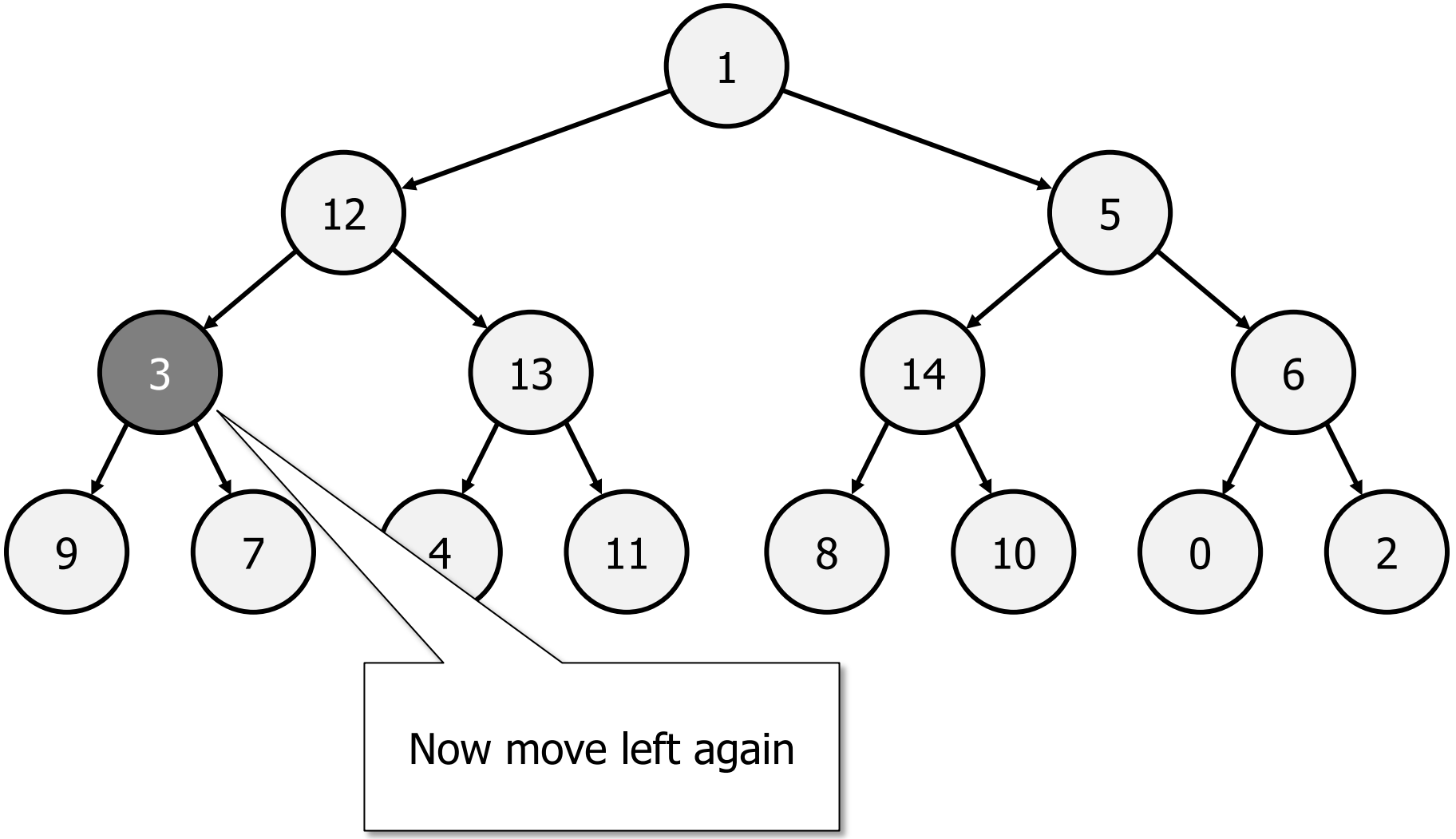
How it works



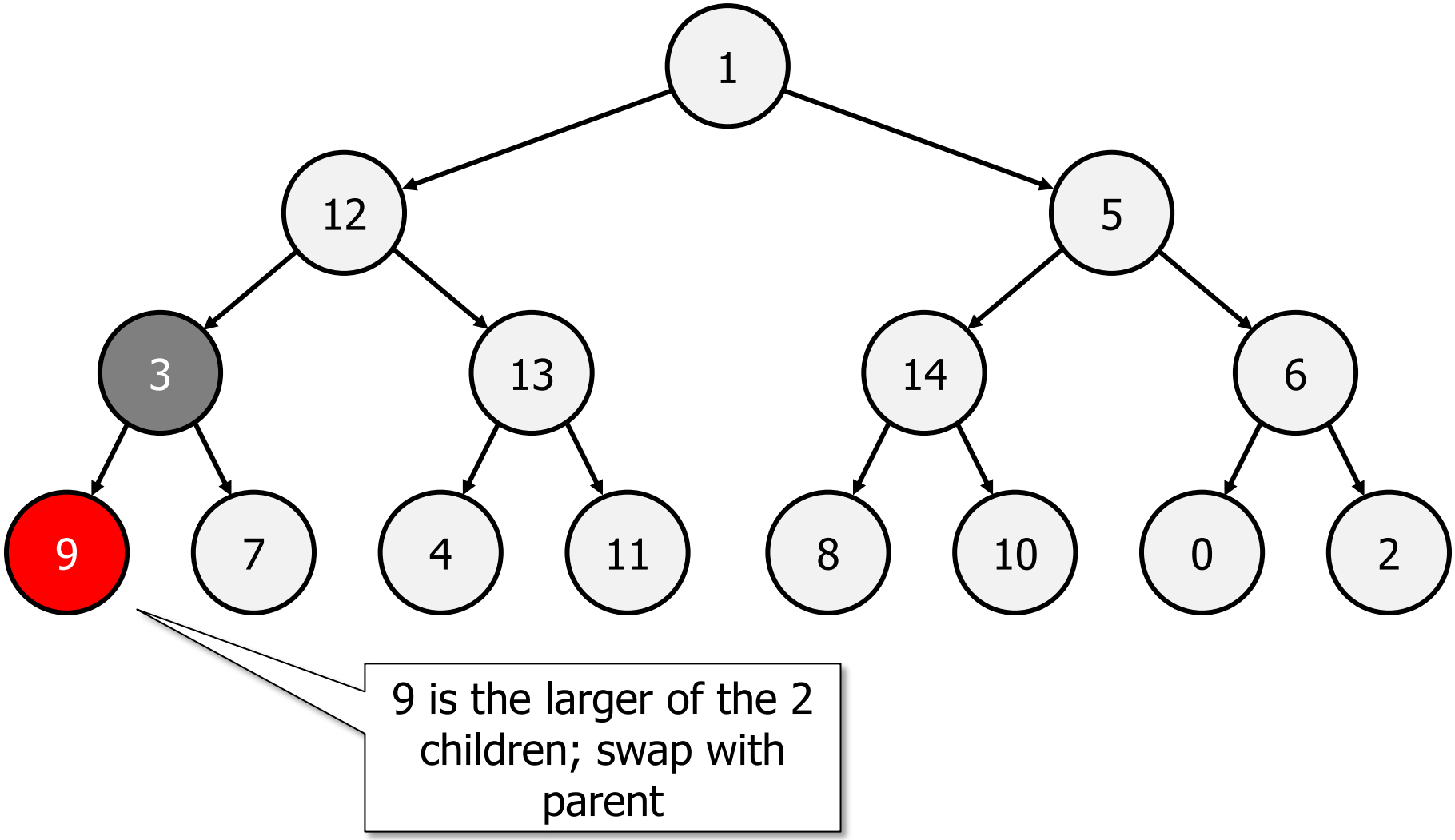
How it works


1
12
5
3
13
14
6
9
7
4
11
8
10
0
2
Nothing to do again: both its children are smaller
Now move left again

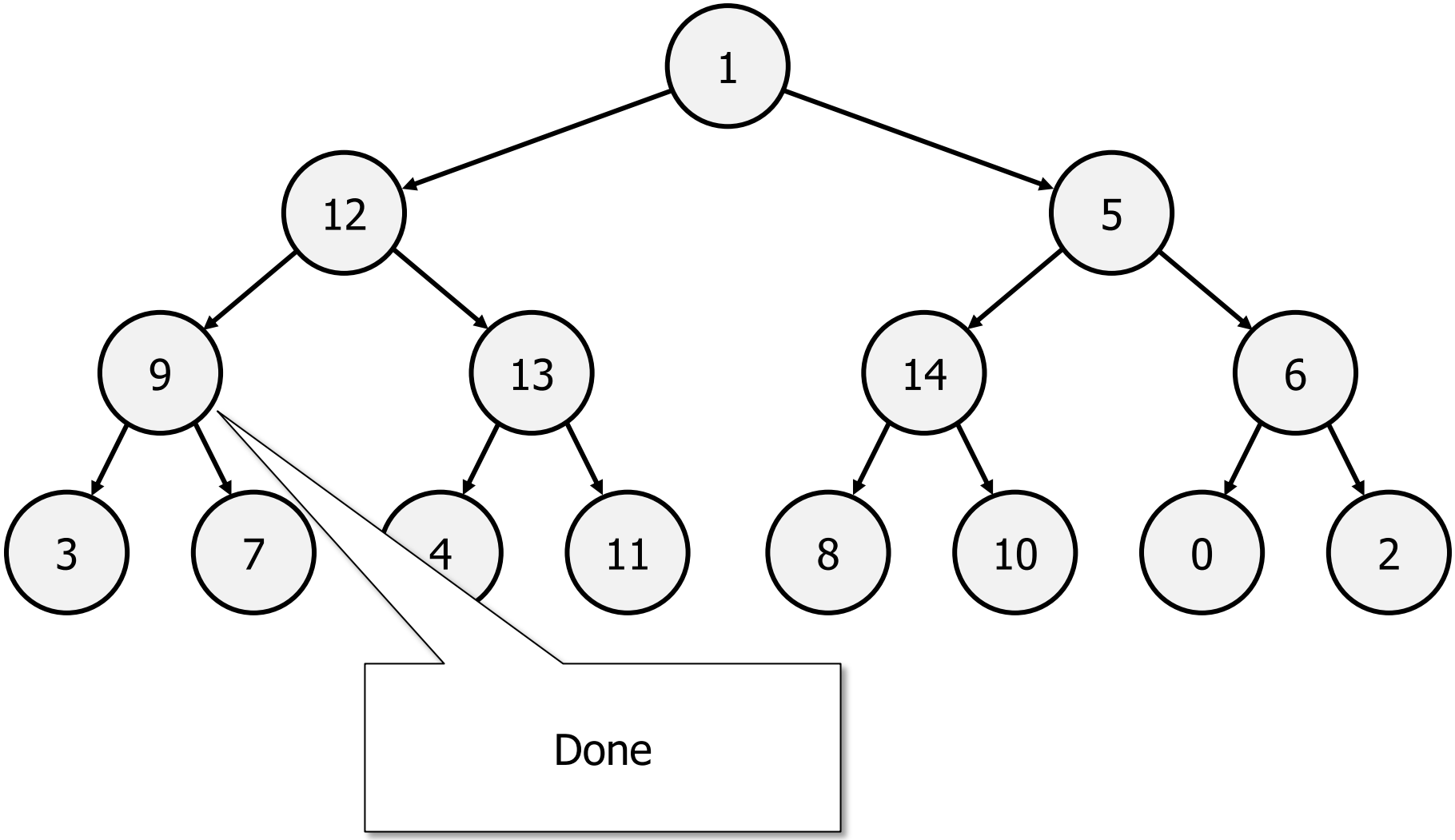
How it works



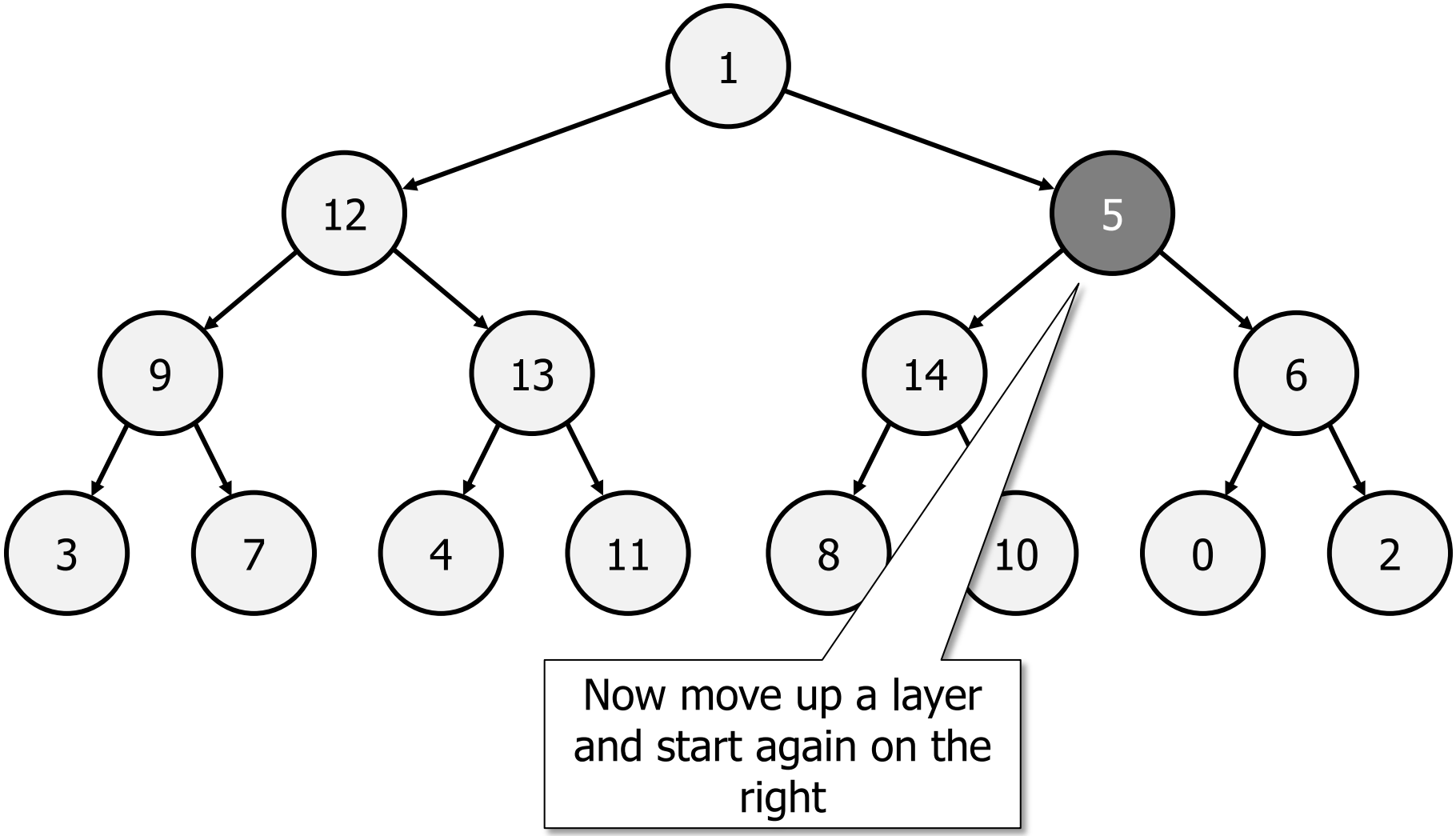
How it works


1
12
5
3
13
14
6
9
7
4
11
8
10
0
2
9 is the larger of the 2 children; swap with parent

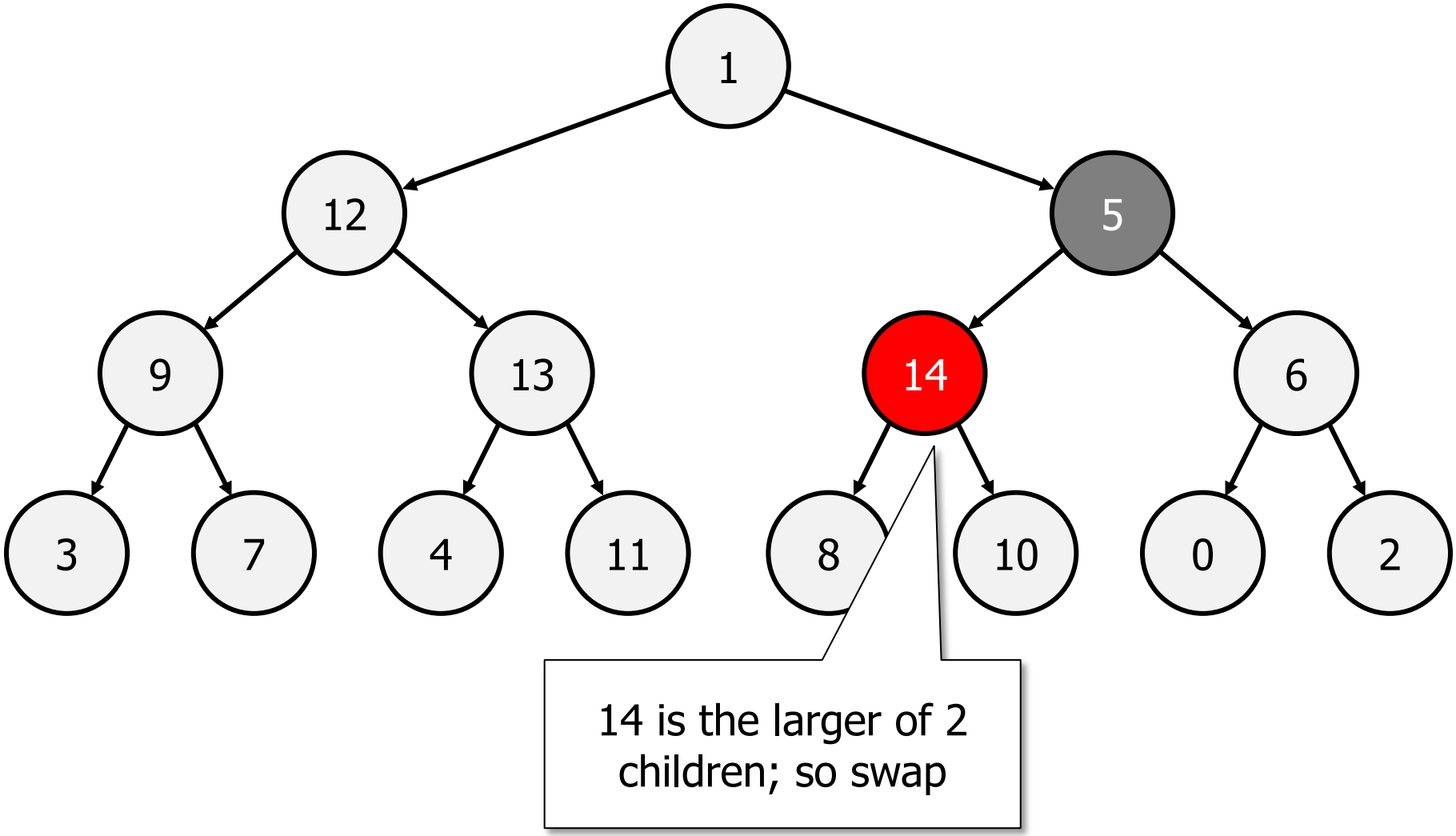
How it works



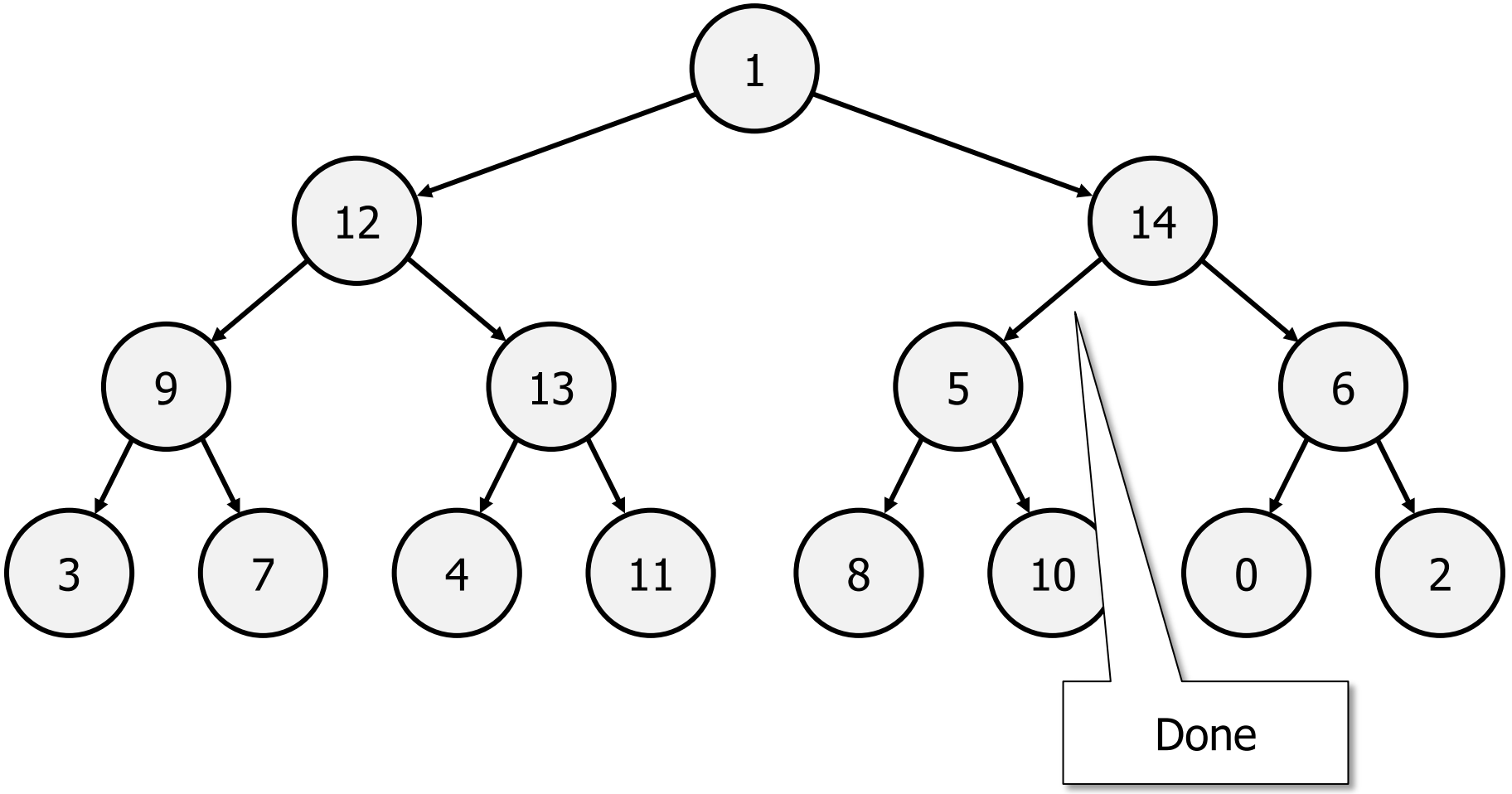
How it works


1
12
5
9
13
14
6
3
7
4
11
8
10
0
2
Now move up a layer and start again on the right

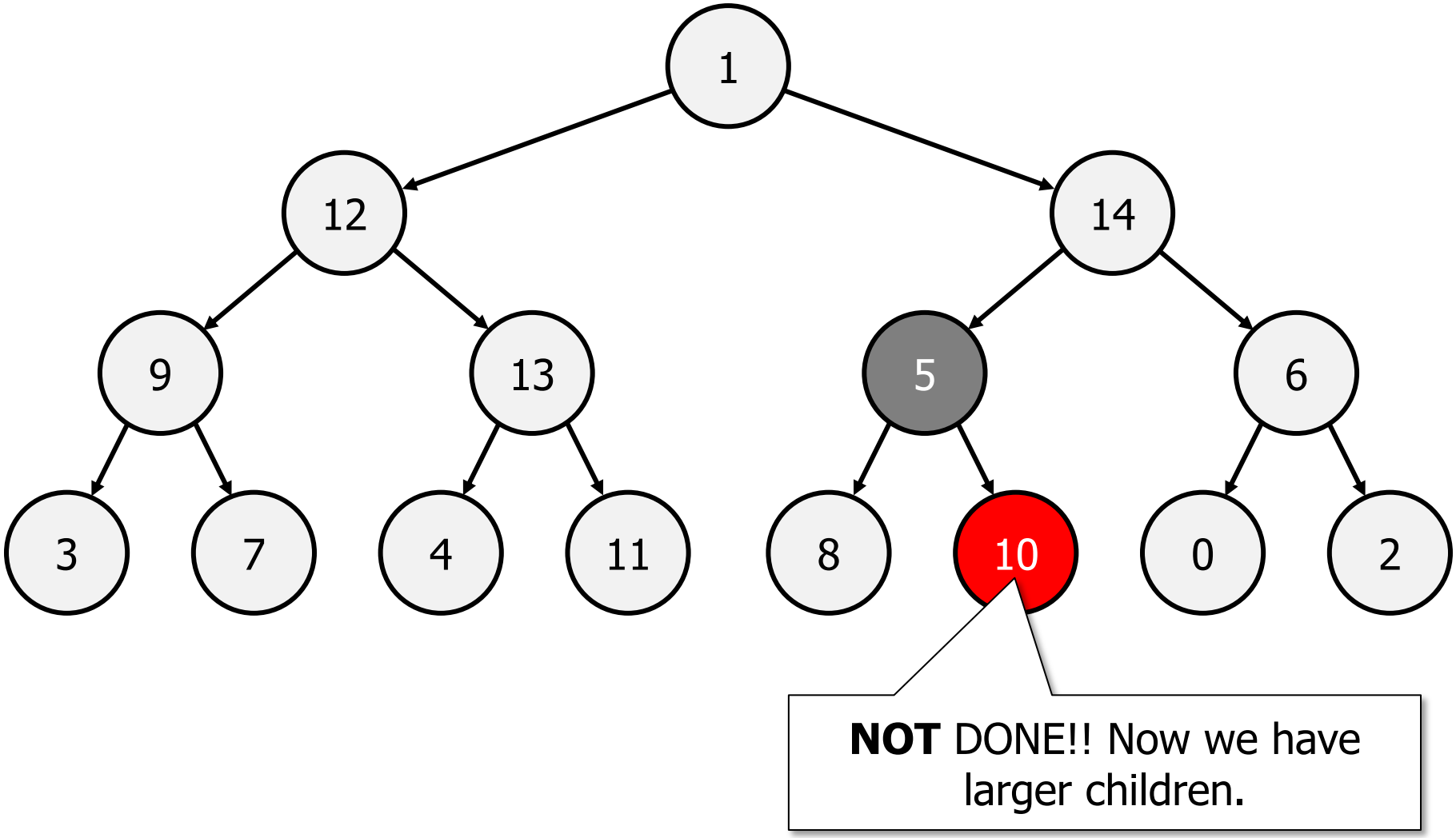
How it works


1
12
5
9
13
14
6
3
7
4
11
8
10
0
2
14 is the larger of 2 children; so swap

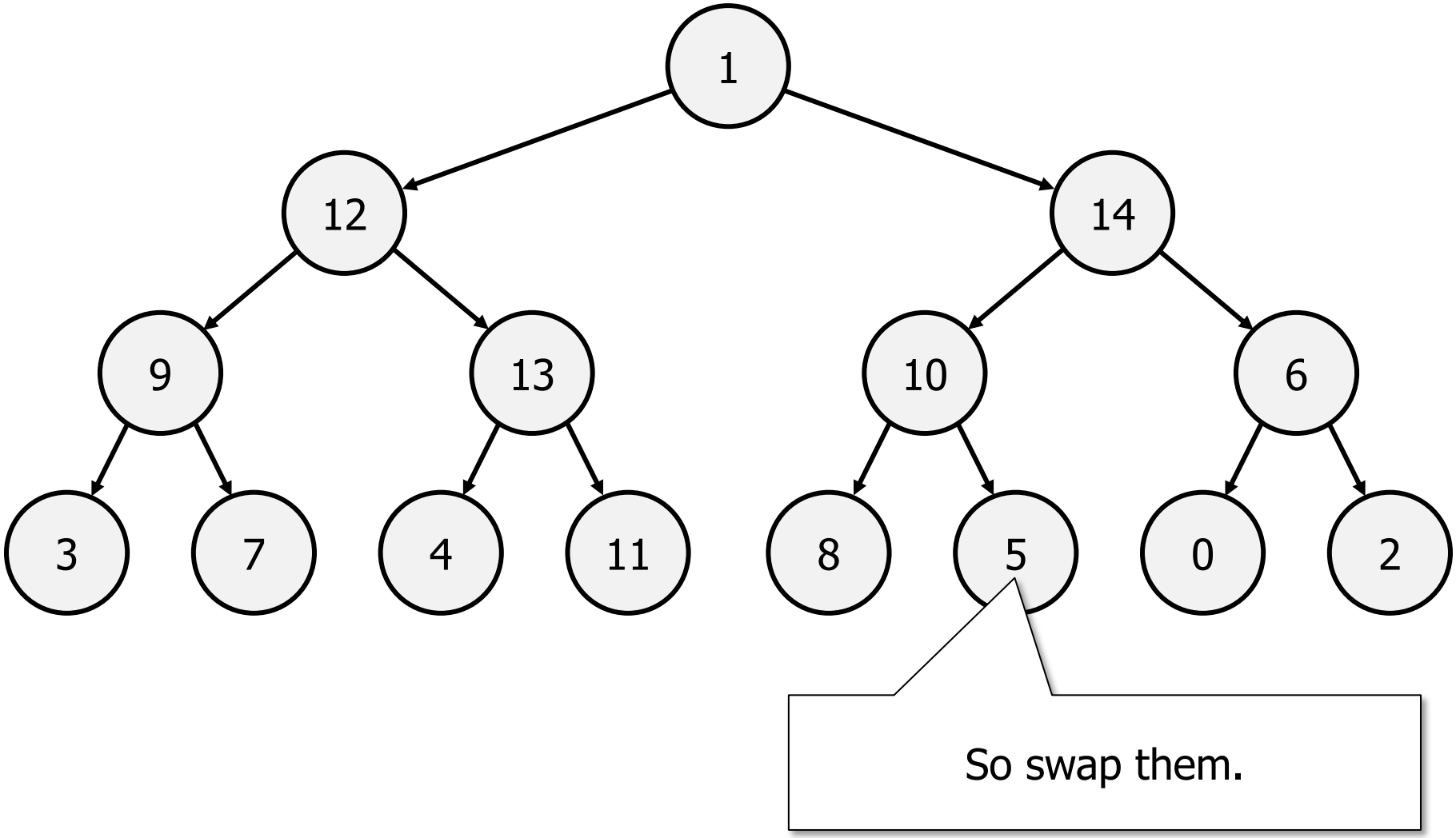
How it works



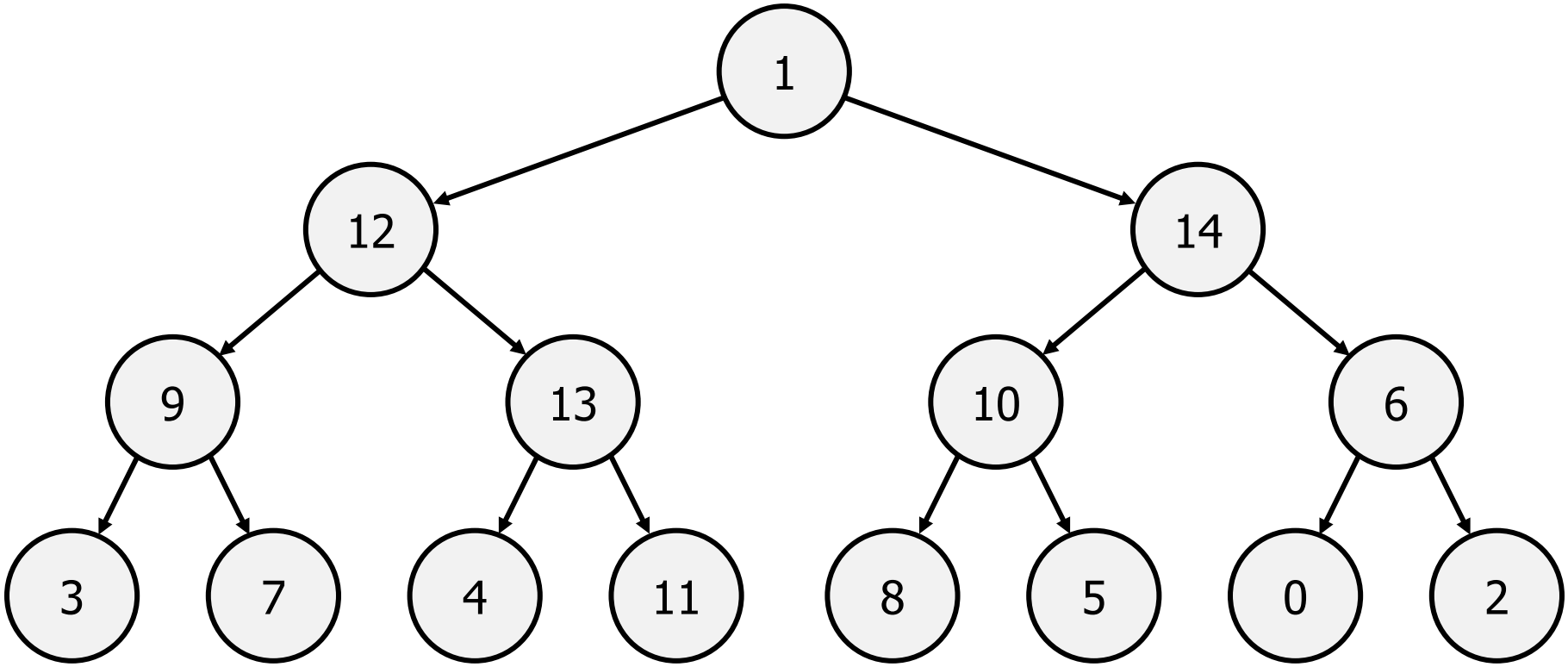
How it works


1
12
14
9
13
5
6
3
7
4
11
8
10
0
2
NOT DONE!! Now we have larger children.

How it works

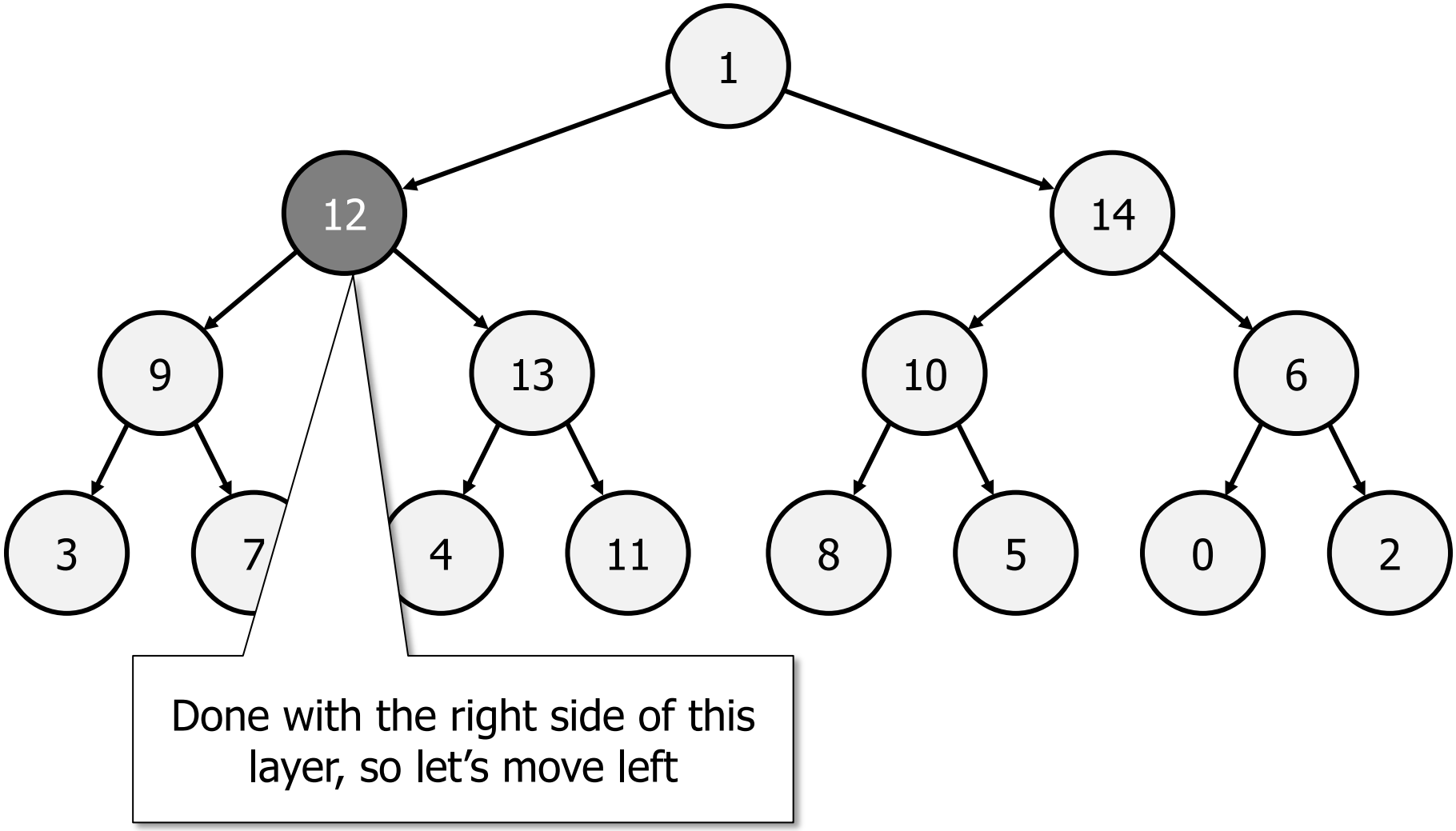


Sifting

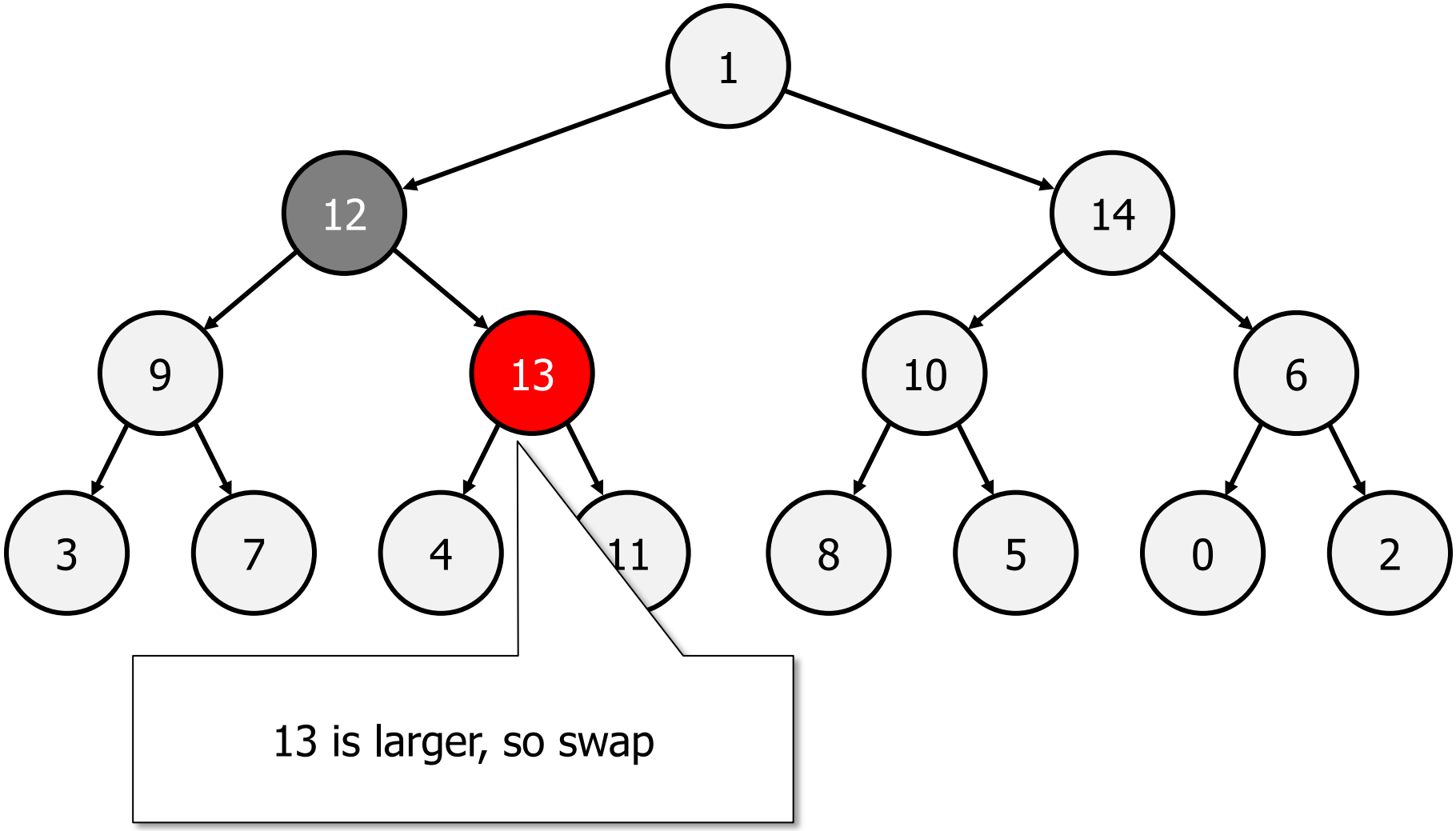


This process of swapping smaller and larger values so that every parent is larger than its children is called **sifting**. We sift downwards, so if we have more than one layer below, we need to continue sifting

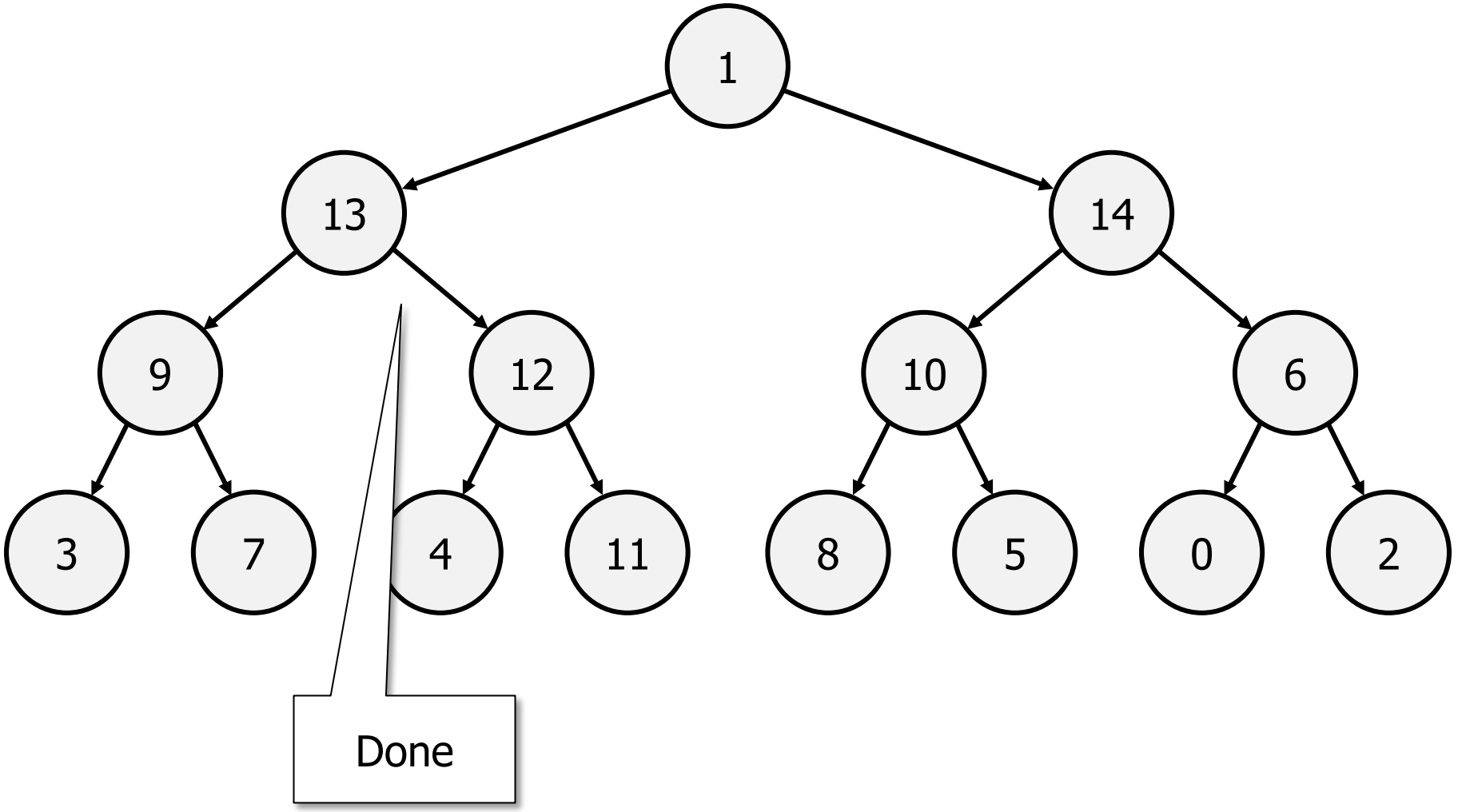
How it works


1
12
14
9
13
10
6
3
7
4
11
8
5
0
2
Done with the right side of this layer, so let's move left

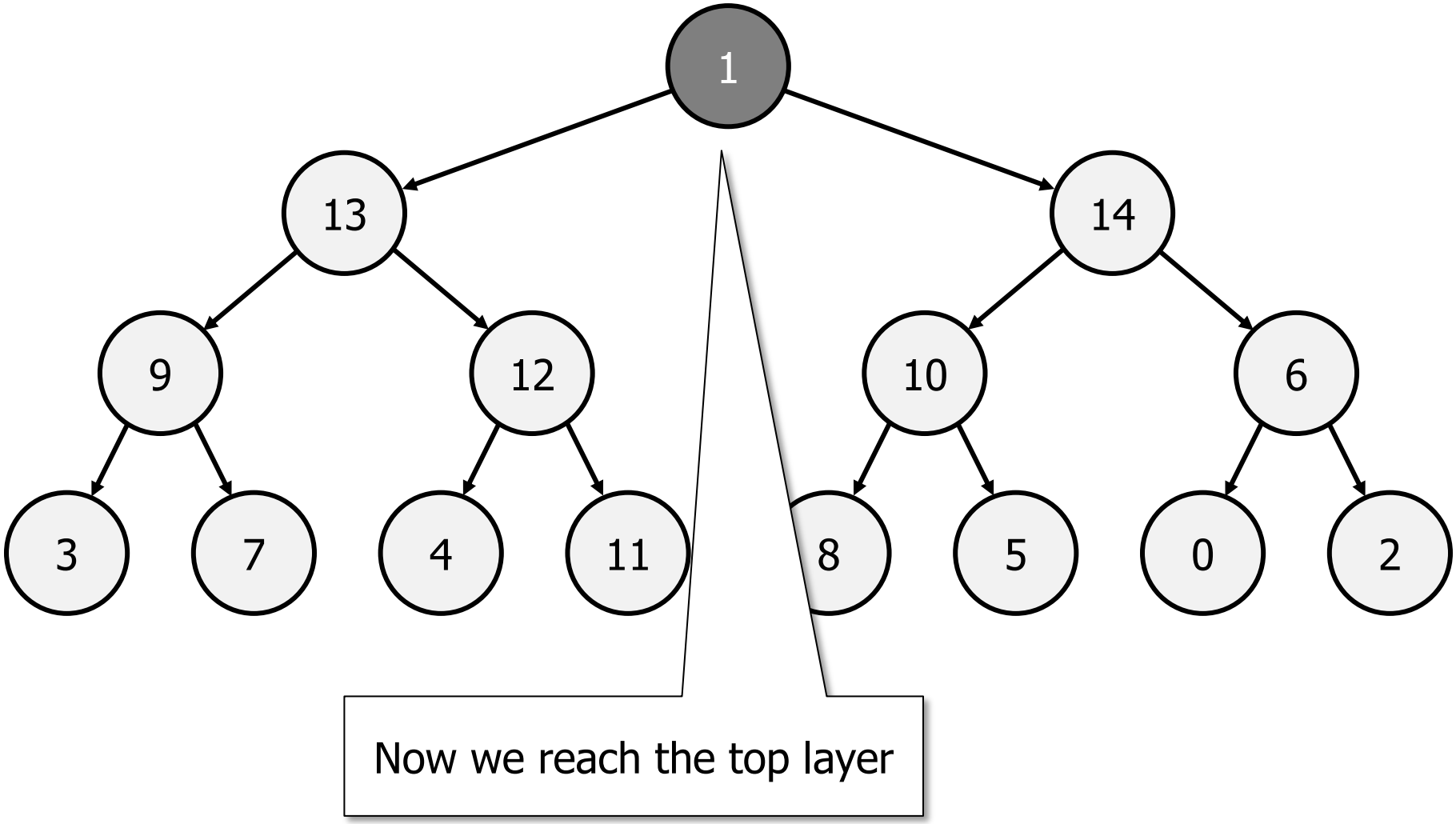
How it works



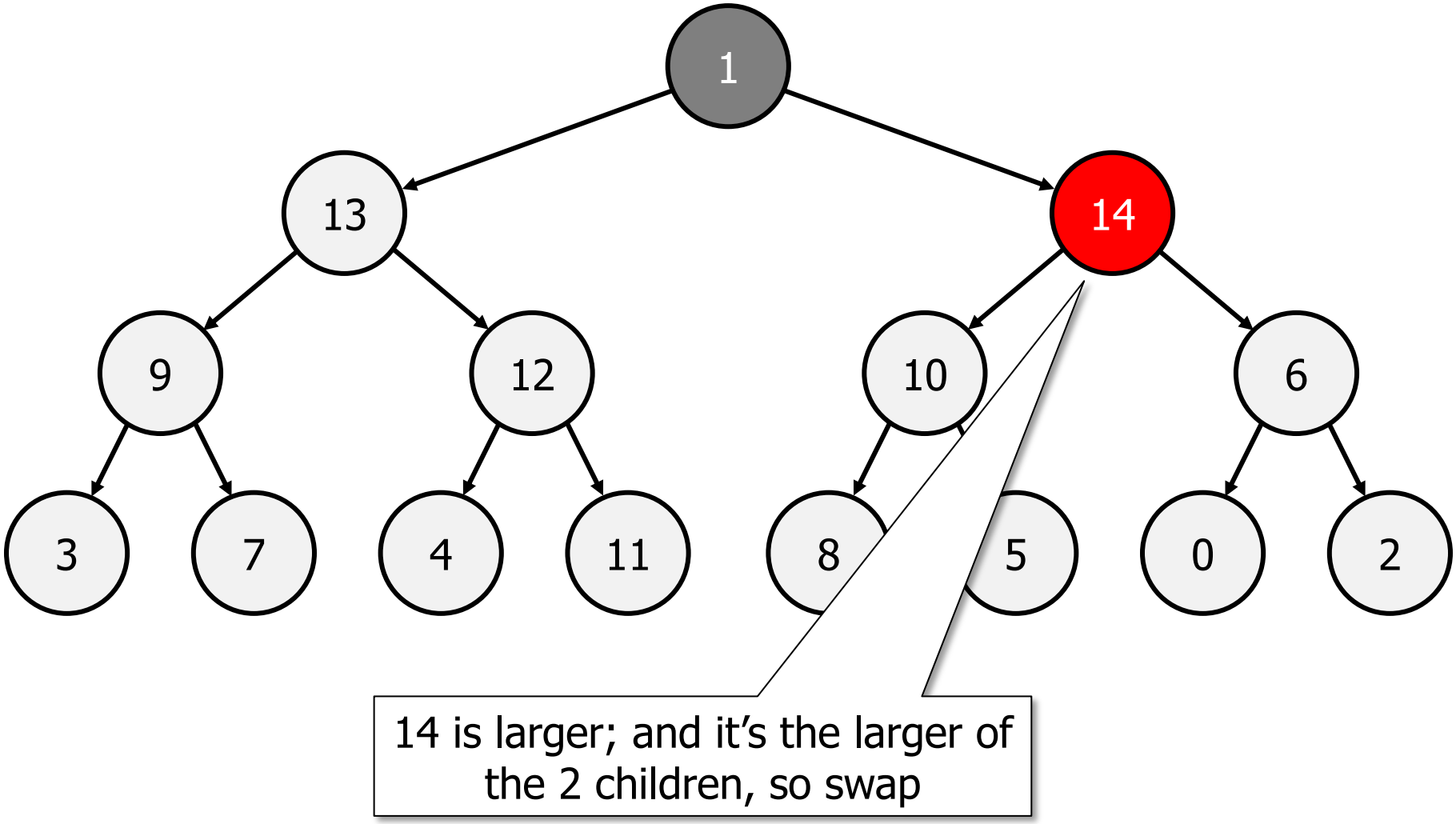
How it works



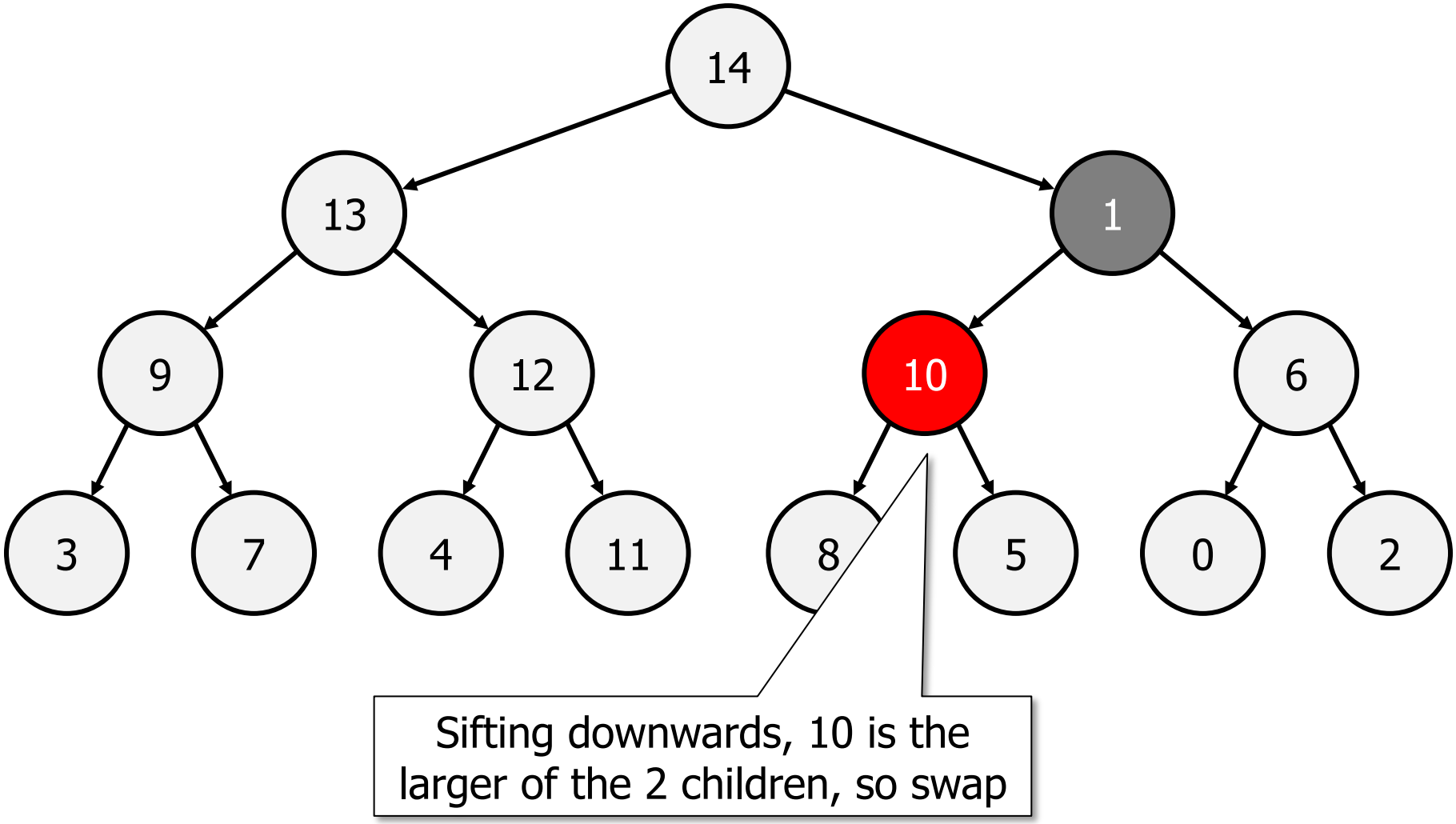
How it works



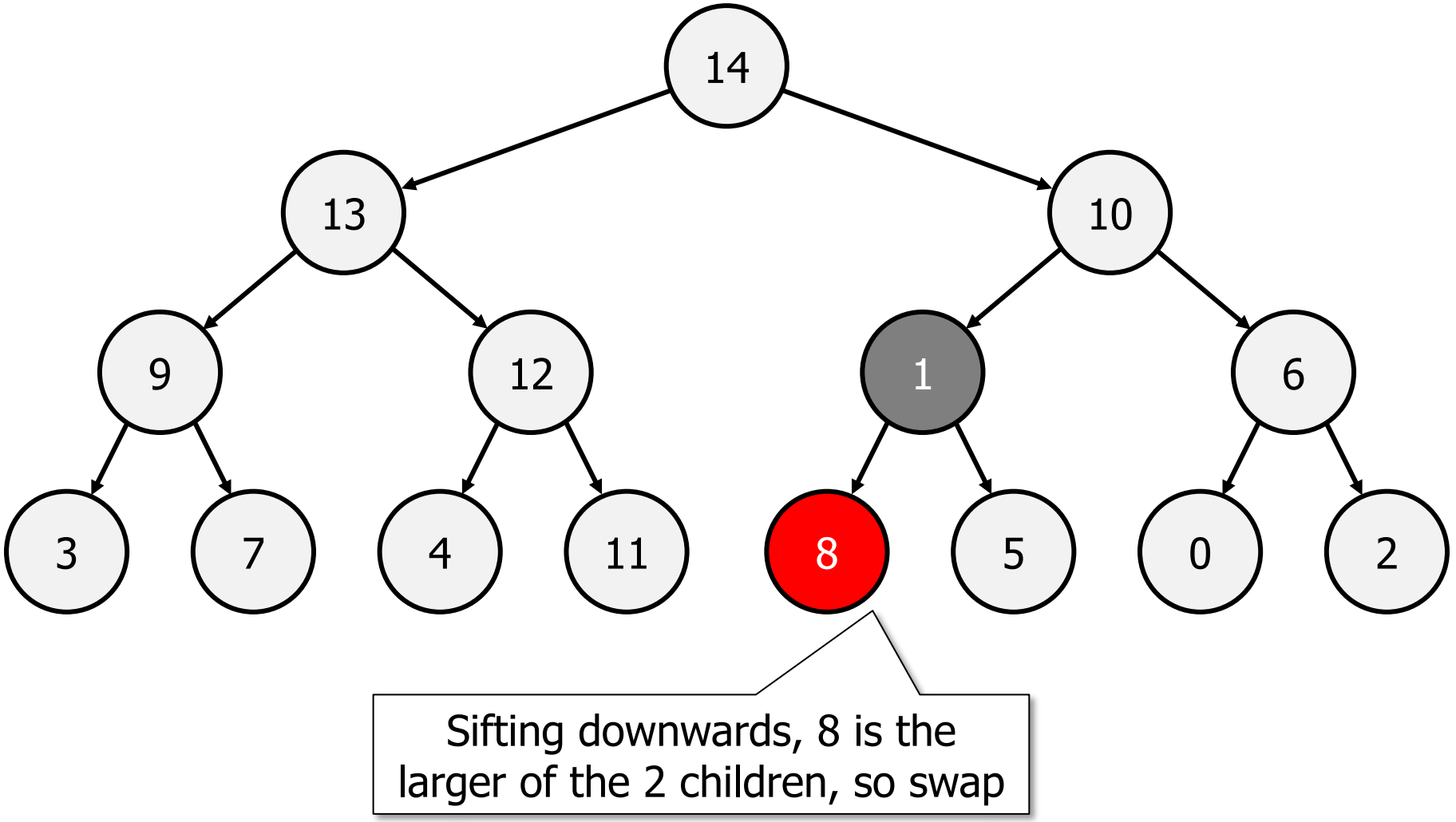
How it works


1
13
14
9
12
10
6
3
7
4
11
8
5
0
2
14 is larger; and it's the larger of the 2 children, so swap

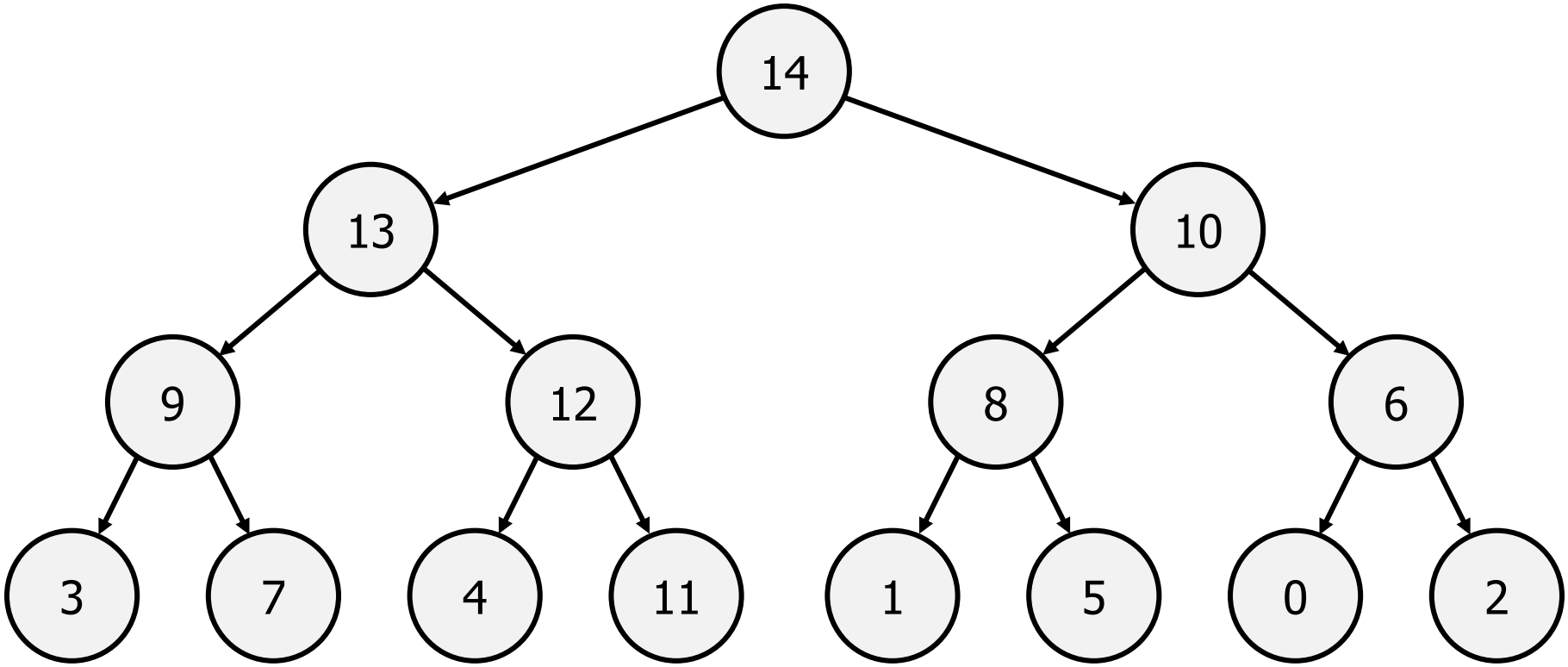
How it works


14
13
1
9
12
10
6
3
7
4
11
8
5
0
2
Sifting downwards, 10 is the larger of the 2 children, so swap

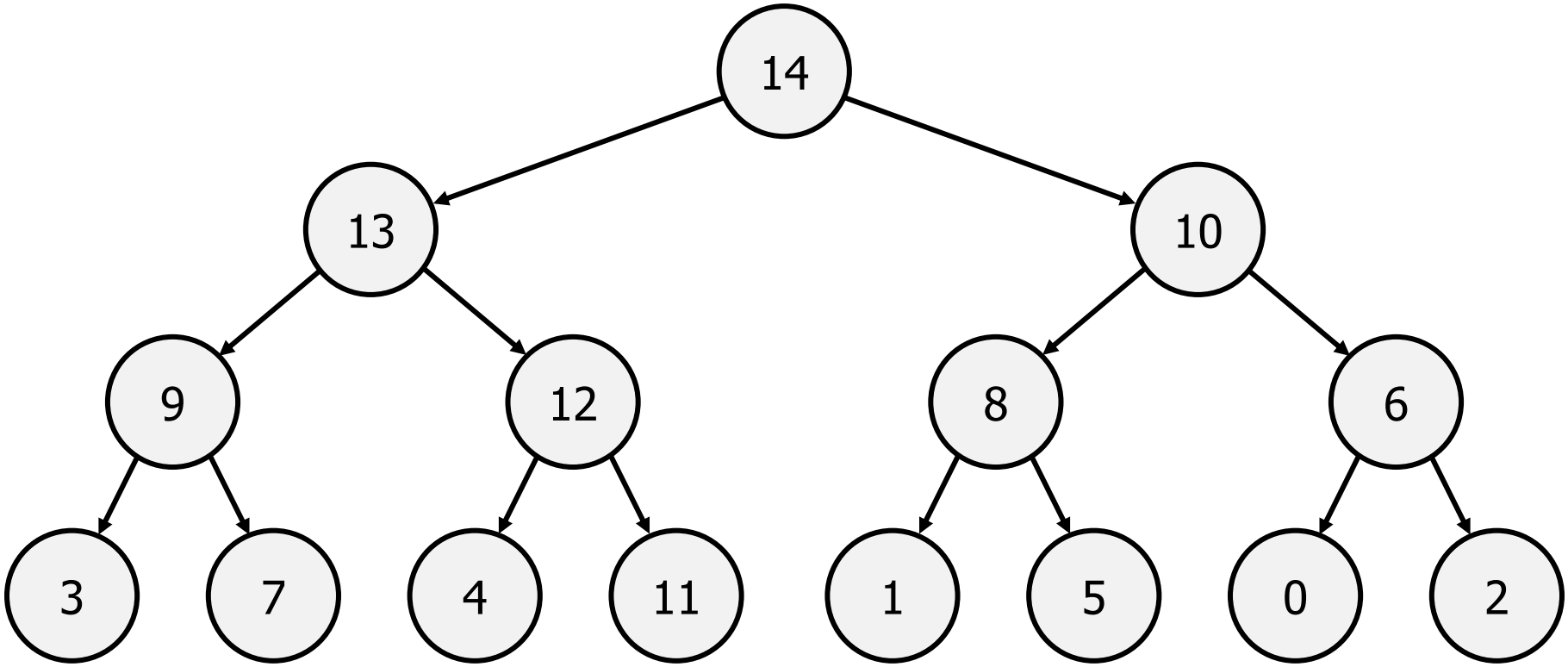
How it works


14
13
10
9
12
1
6
3
7
4
11
8
5
0
2
Sifting downwards, 8 is the larger of the 2 children, so swap

Done. We're Heapified

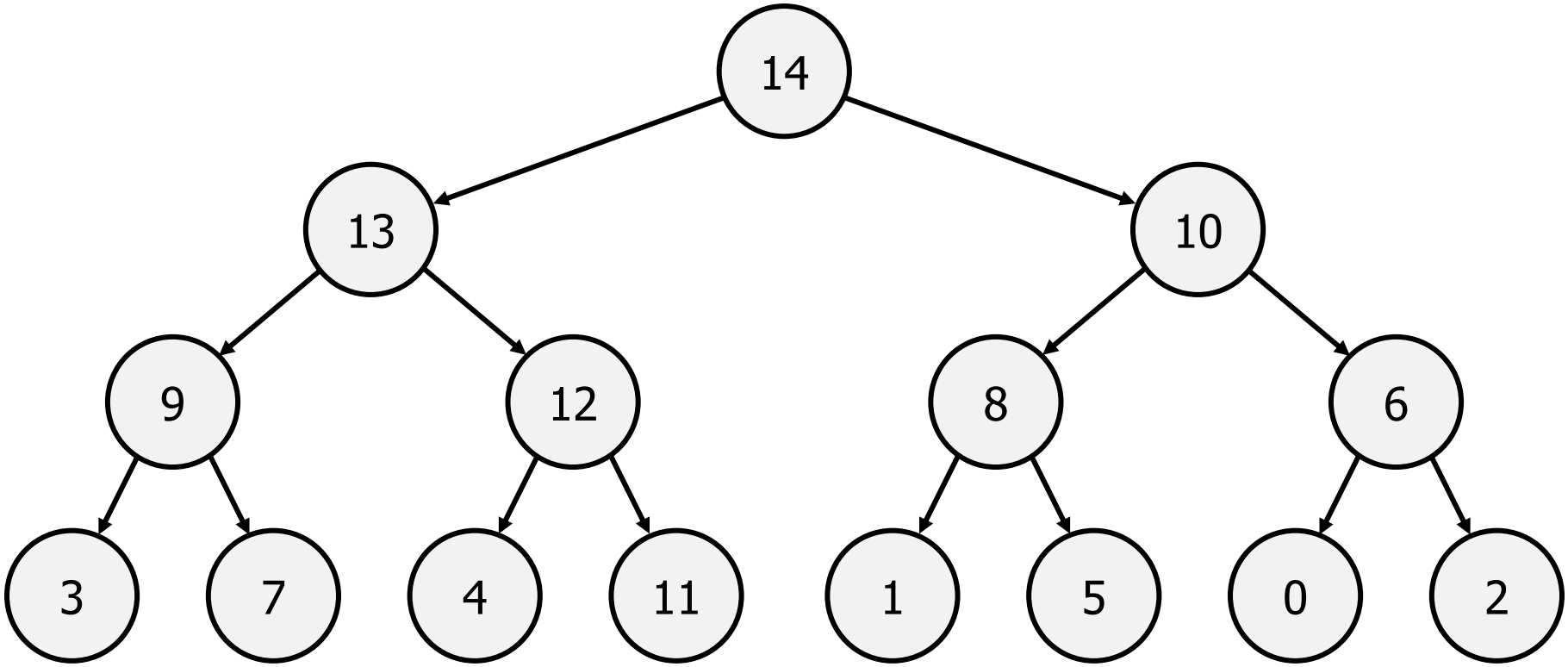


Let's Check The List



14	13	10	9	12	8	6	3	7	4	11	1	5	0	2
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

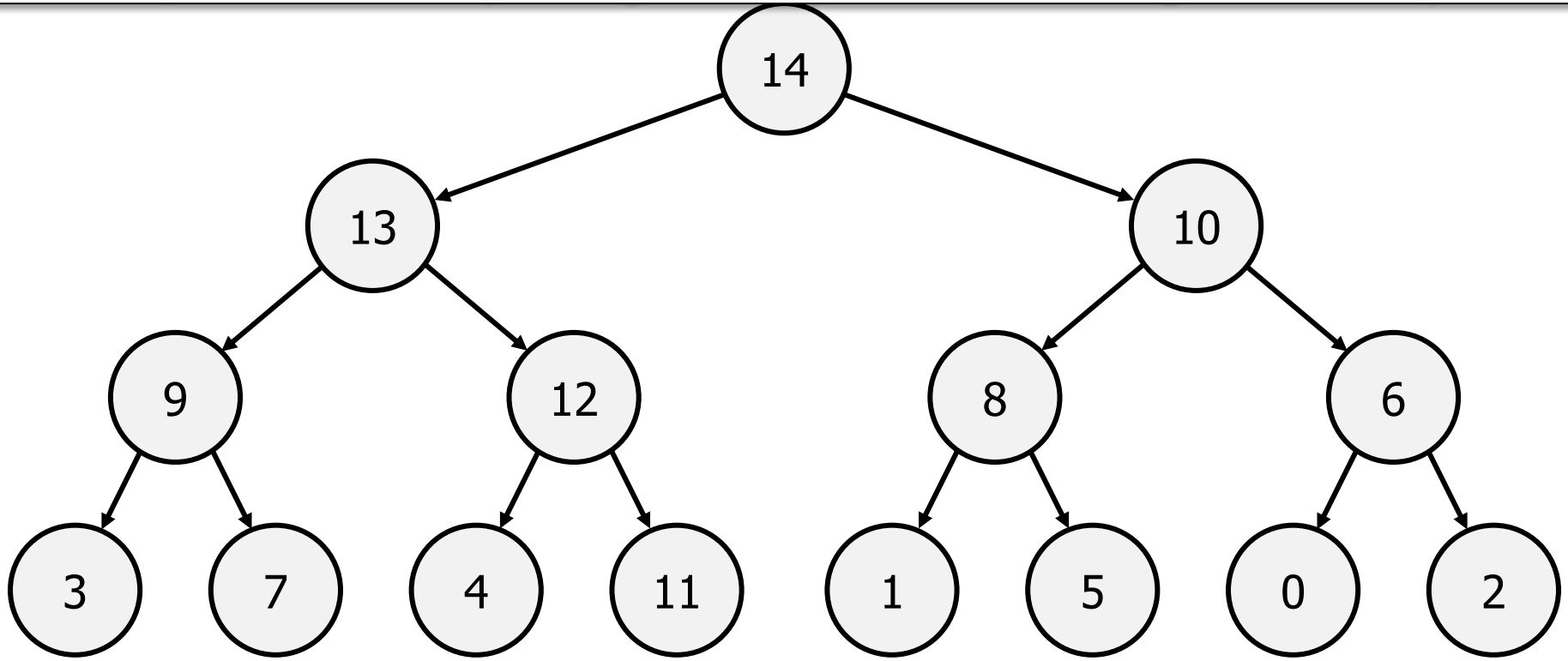
List Is Not Sorted!



14	13	10	9	12	8	6	3	7	4	11	1	5	0	2
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Heapify Does Not Sort!

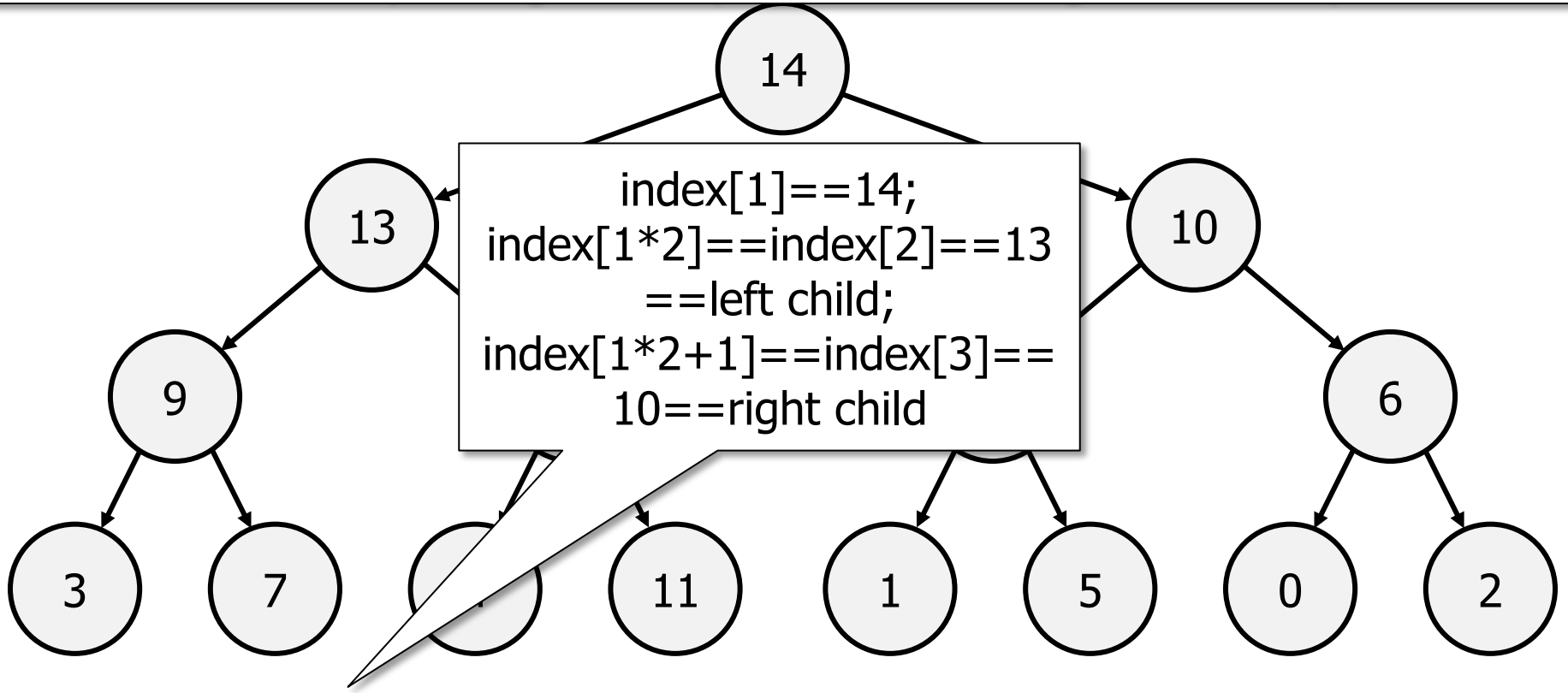
But the heap satisfies the rules: every parent is greater than each of its children; and the corresponding list obeys the rules linking children to parents



14	13	10	9	12	8	6	3	7	4	11	1	5	0	2
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Heapify Does Not Sort!

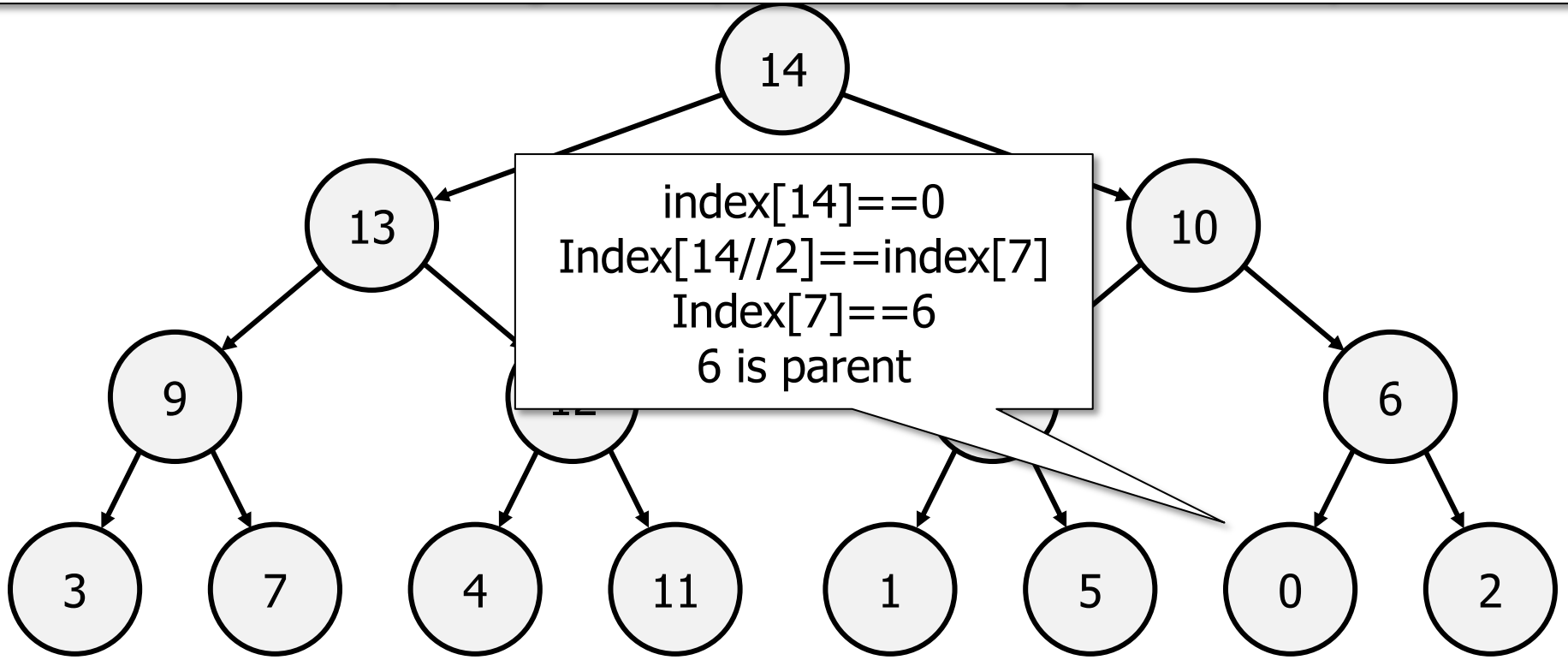
But a heap satisfies the rules: every parent is greater than each of its children; and the corresponding list obeys the rules linking children to parents



14	13	10	9	12	8	6	3	7	4	11	1	5	0	2
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Heapify Does Not Sort!

But a heap satisfies the rules: every parent is greater than each of its children; and the corresponding list obeys the rules linking children to parents

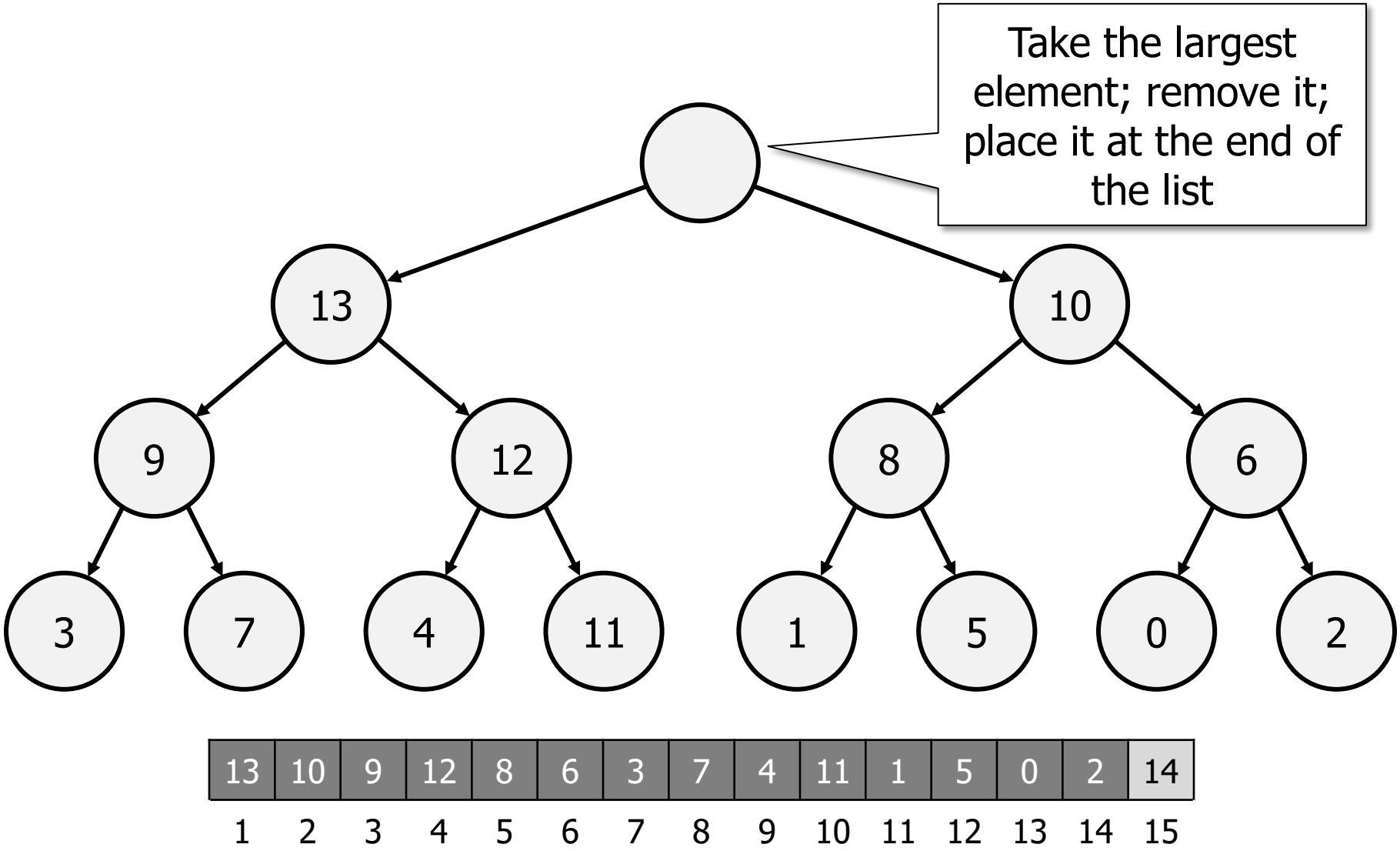


14	13	10	9	12	8	6	3	7	4	11	1	5	0	2
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Heap Sort

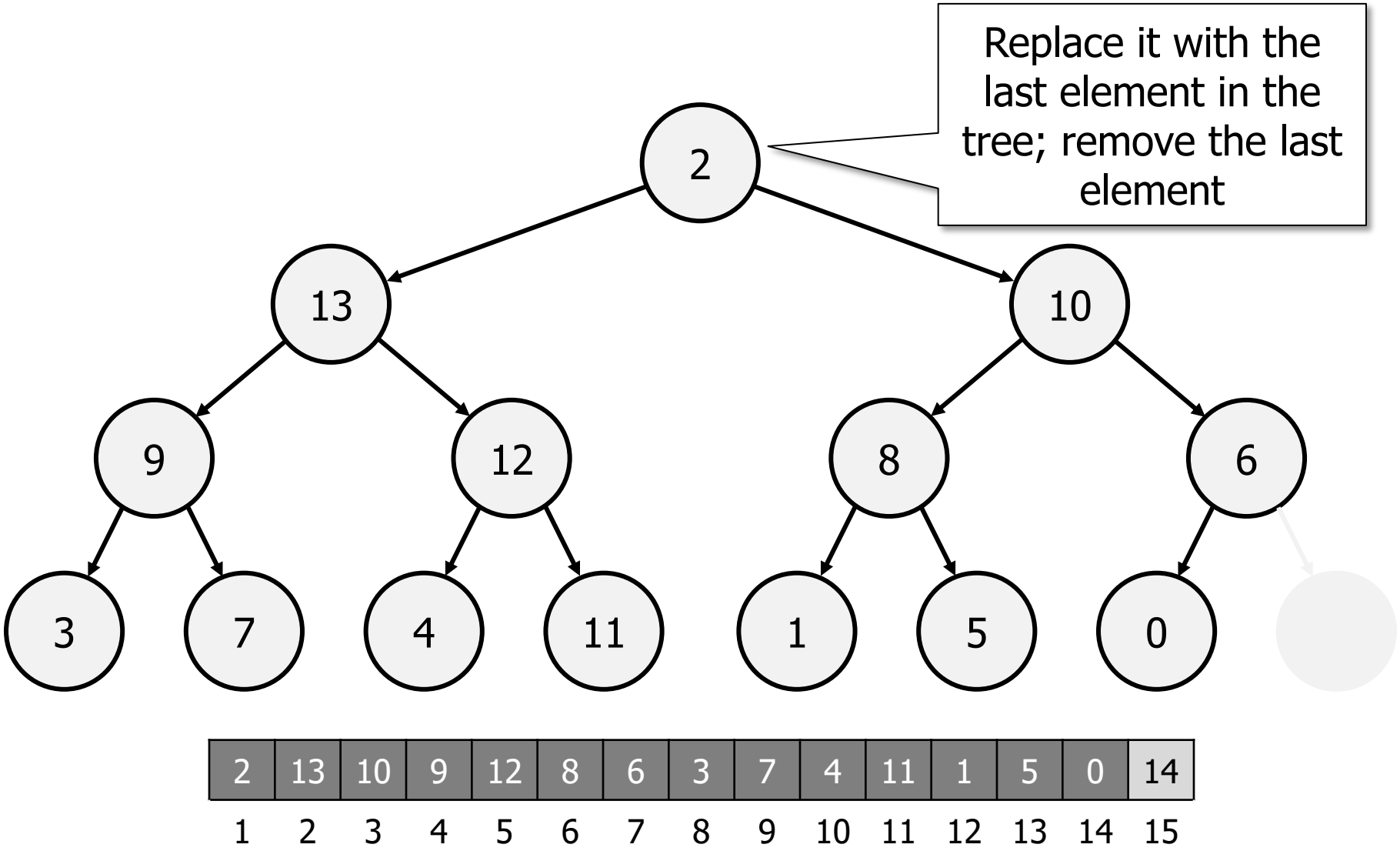
- So: Heapification **does not sort**!
- There are two processes in Heap Sort:
    1. Heapify
    2. Sort
- We've Heapified
- So now let's sort

Sorting A Heap

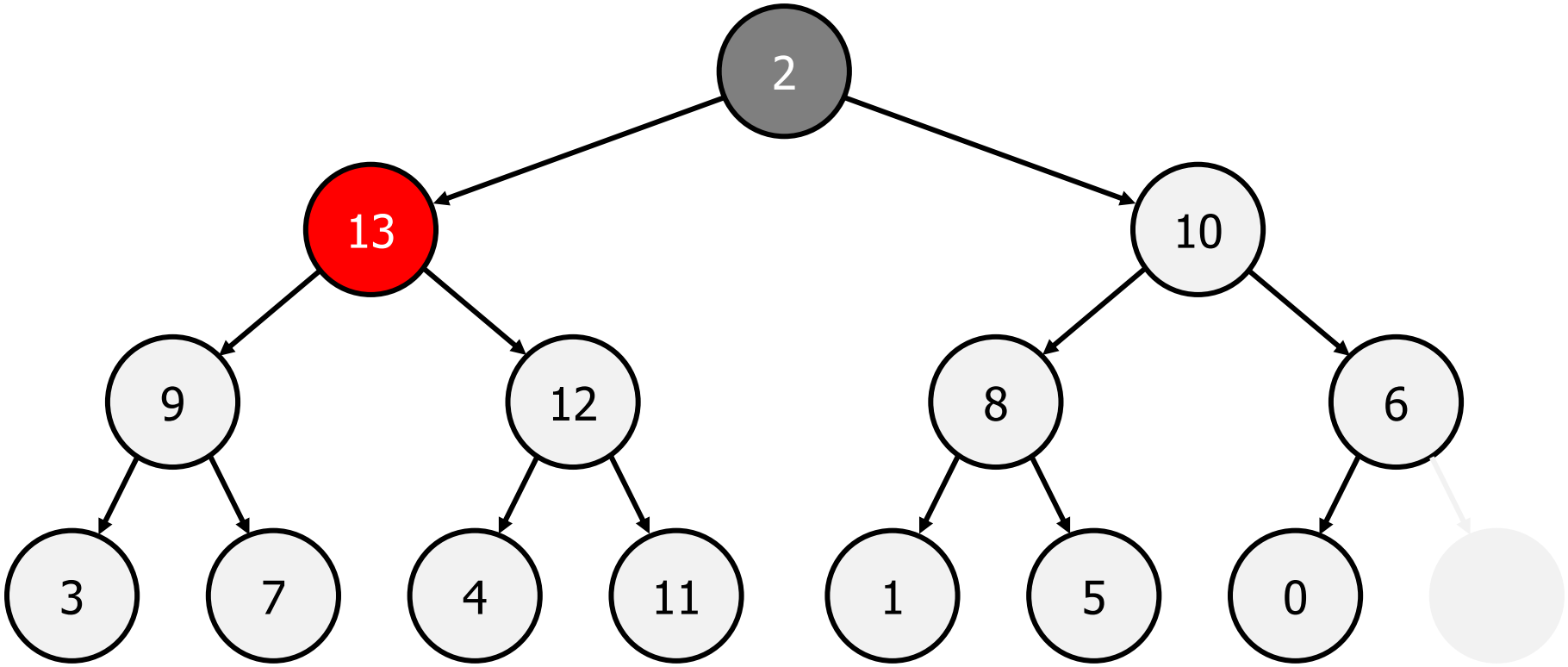


13	10	9	12	8	6	3	7	4	11	1	5	0	2	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sorting A Heap

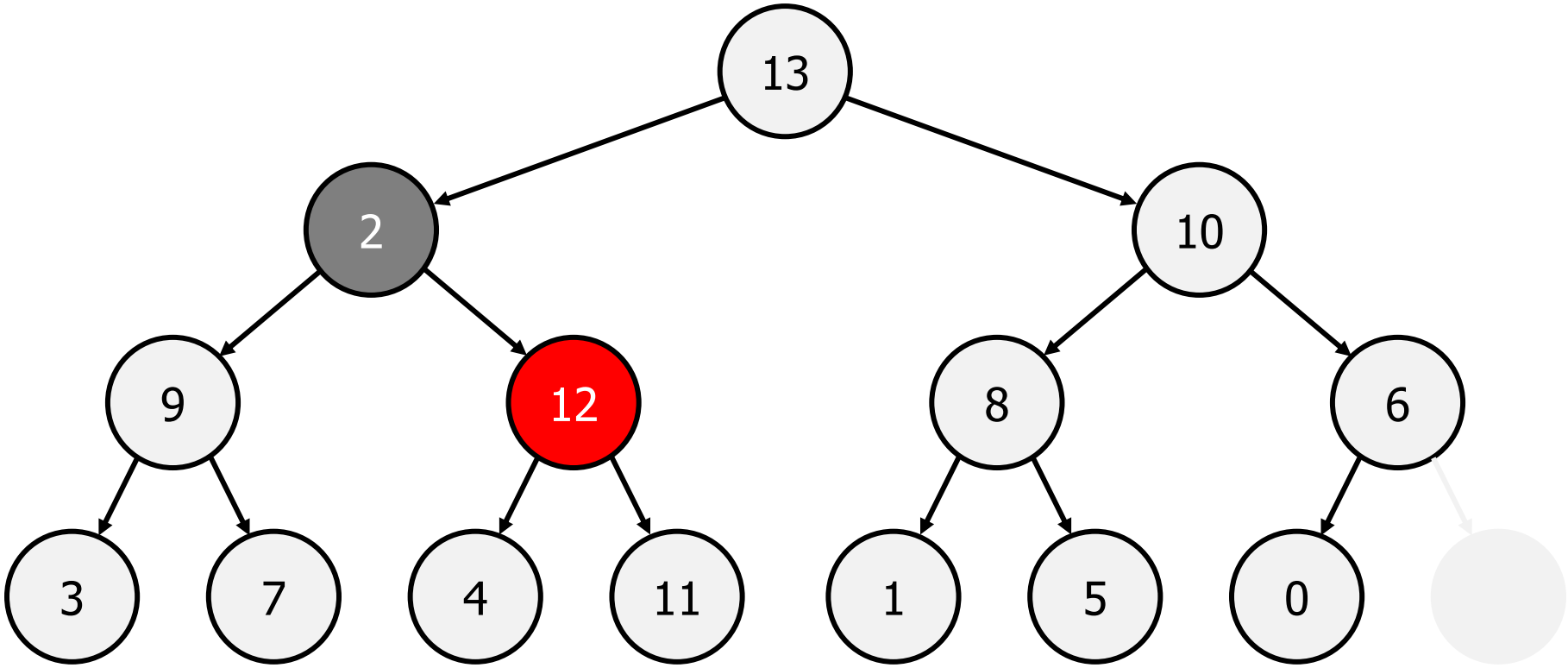


Sift Down (as in Heapify)



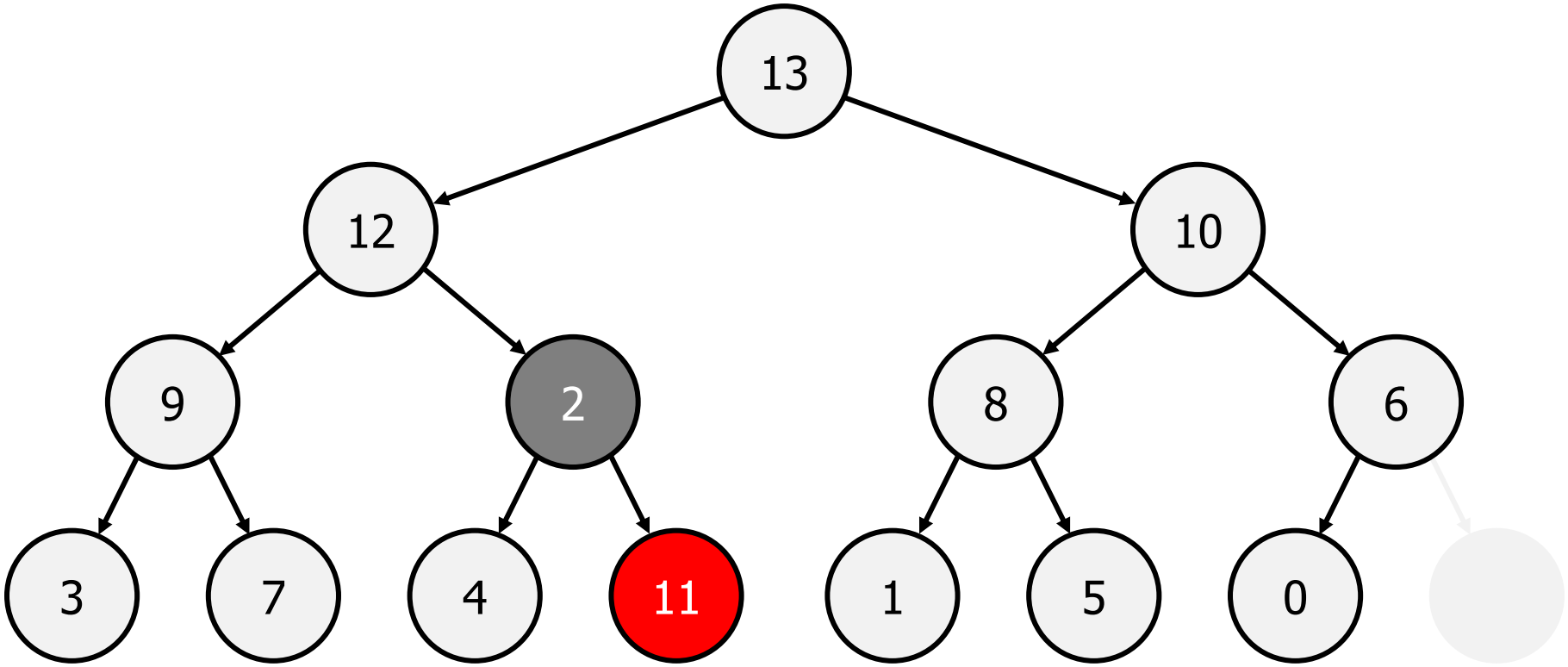
2	13	10	9	12	8	6	3	7	4	11	1	5	0	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



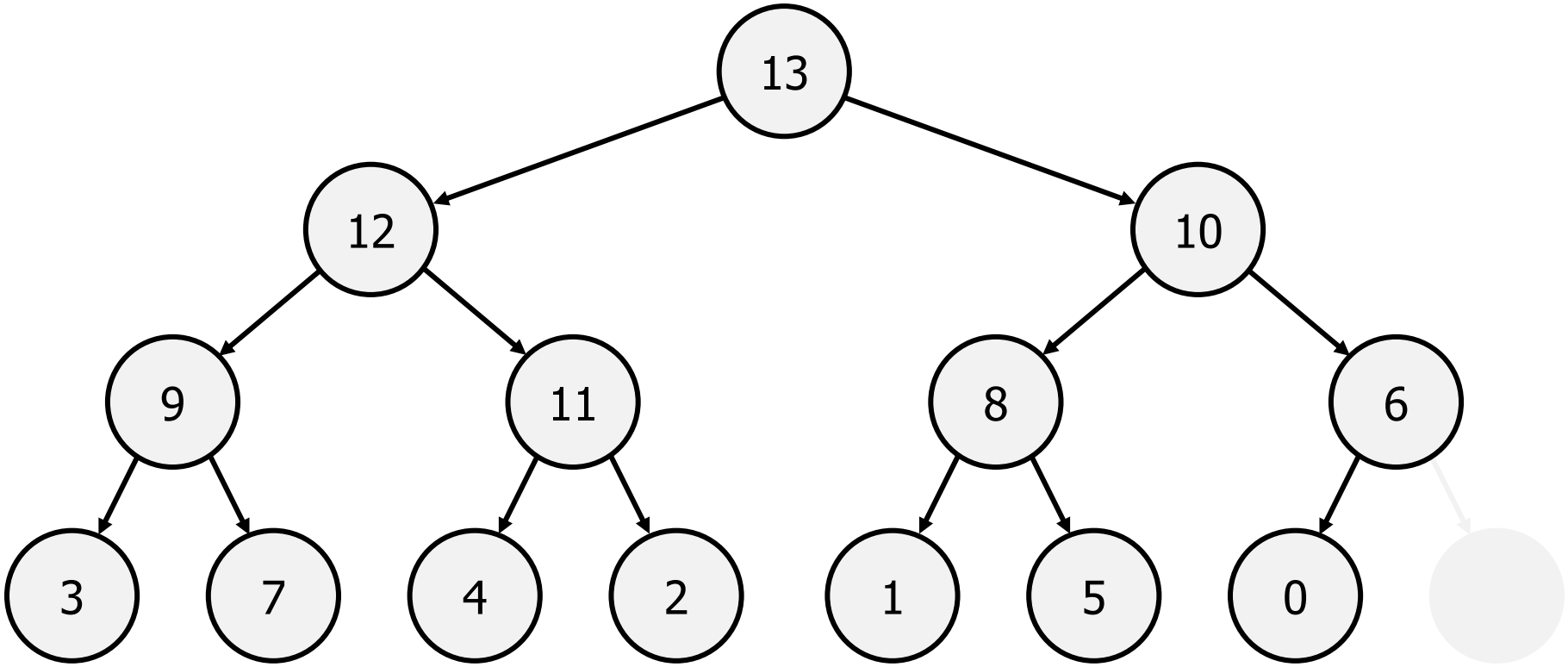
13	2	10	9	12	8	6	3	7	4	11	1	5	0	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



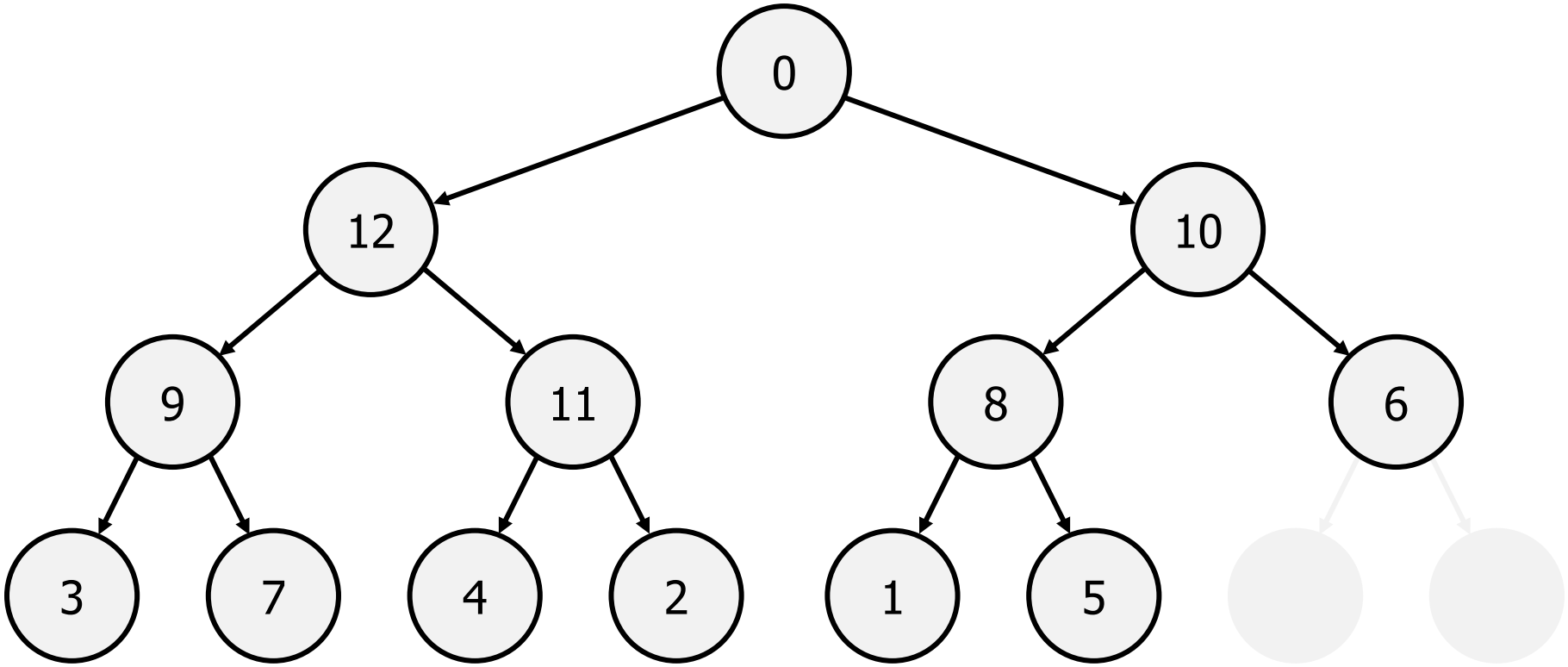
13	12	10	9	2	8	6	3	7	4	11	1	5	0	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down Complete



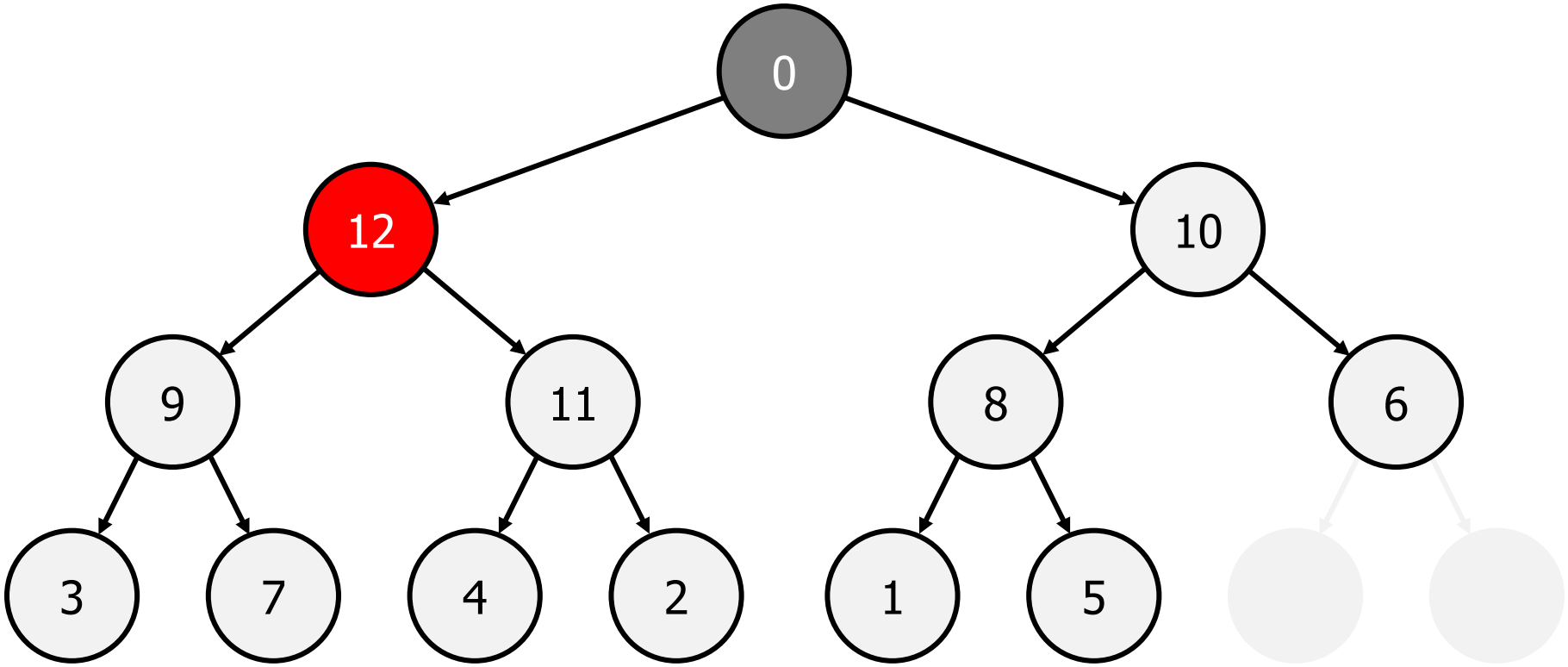
13	12	10	9	11	8	6	3	7	4	2	1	5	0	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Remove Largest; Replace With Last Element



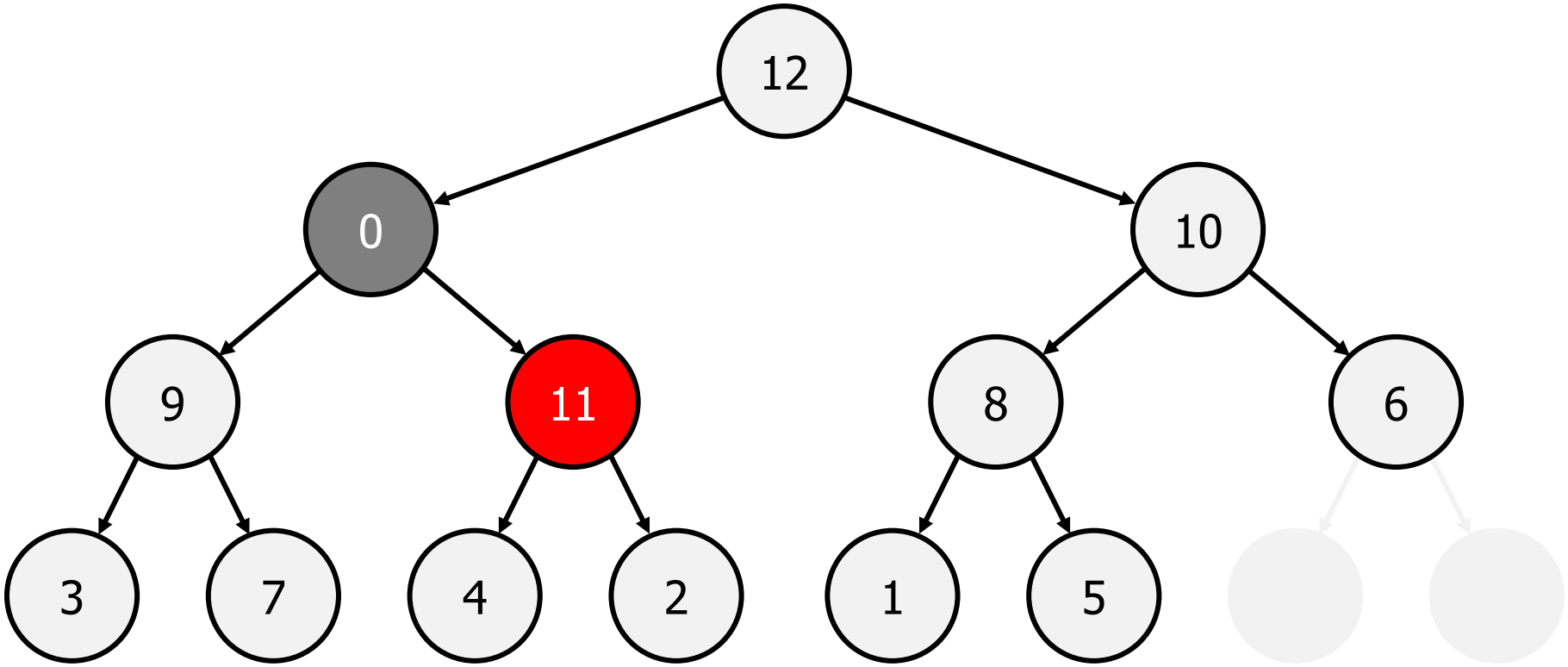
0	12	10	9	11	8	6	3	7	4	2	1	5	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



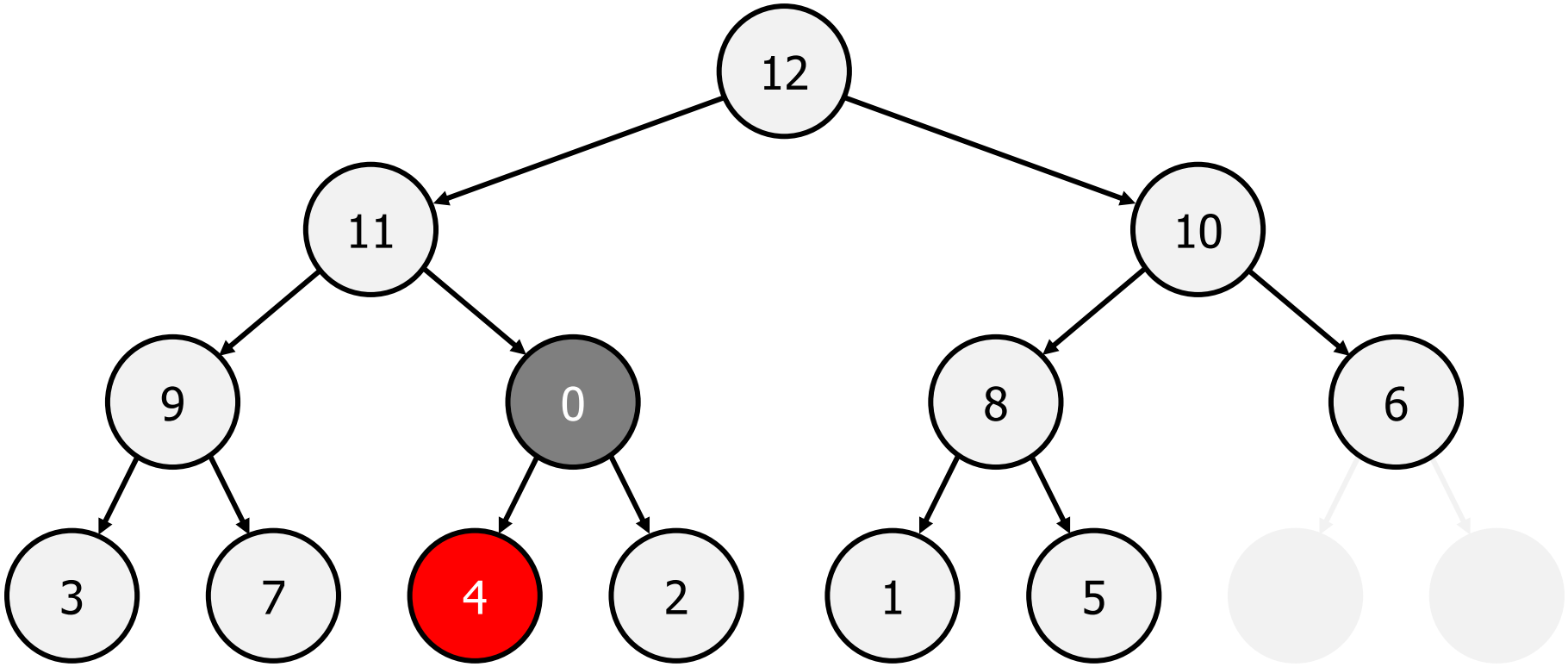
0	12	10	9	11	8	6	3	7	4	2	1	5	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



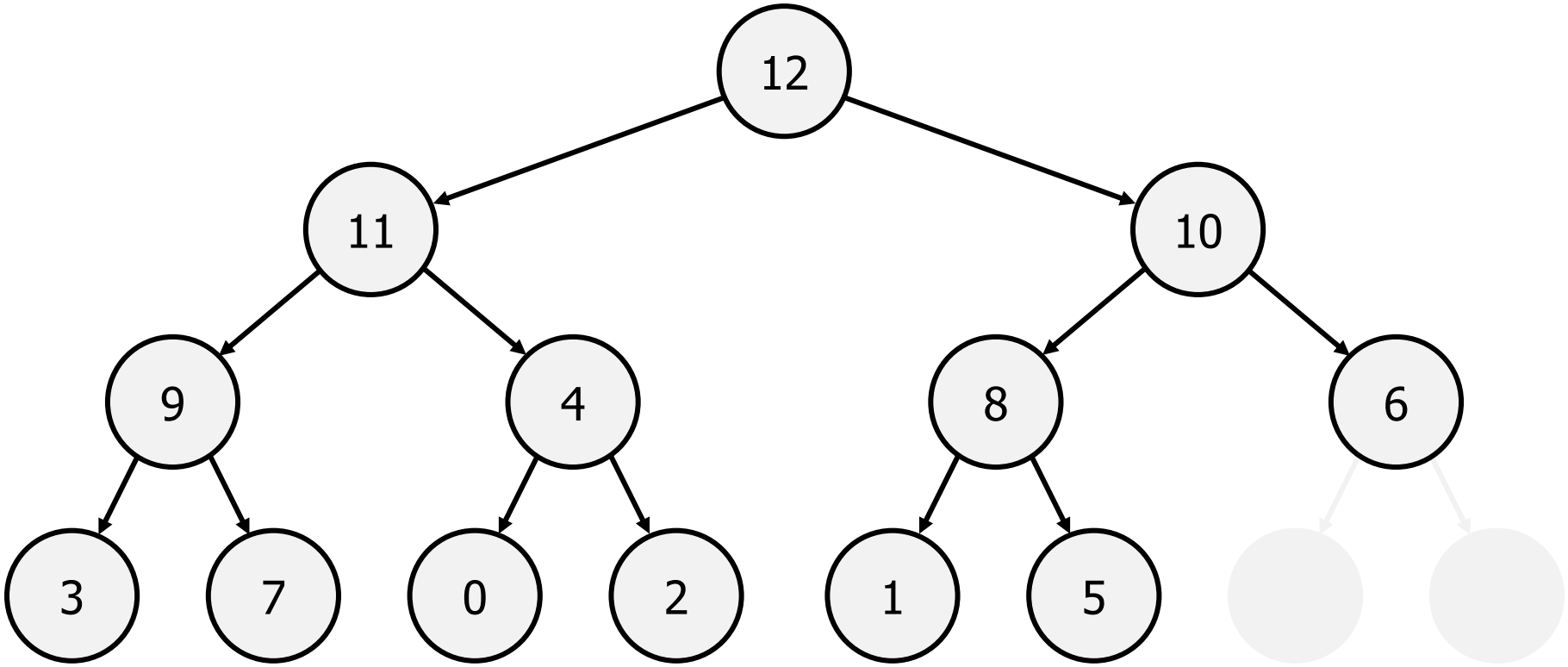
12	0	10	9	11	8	6	3	7	4	2	1	5	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



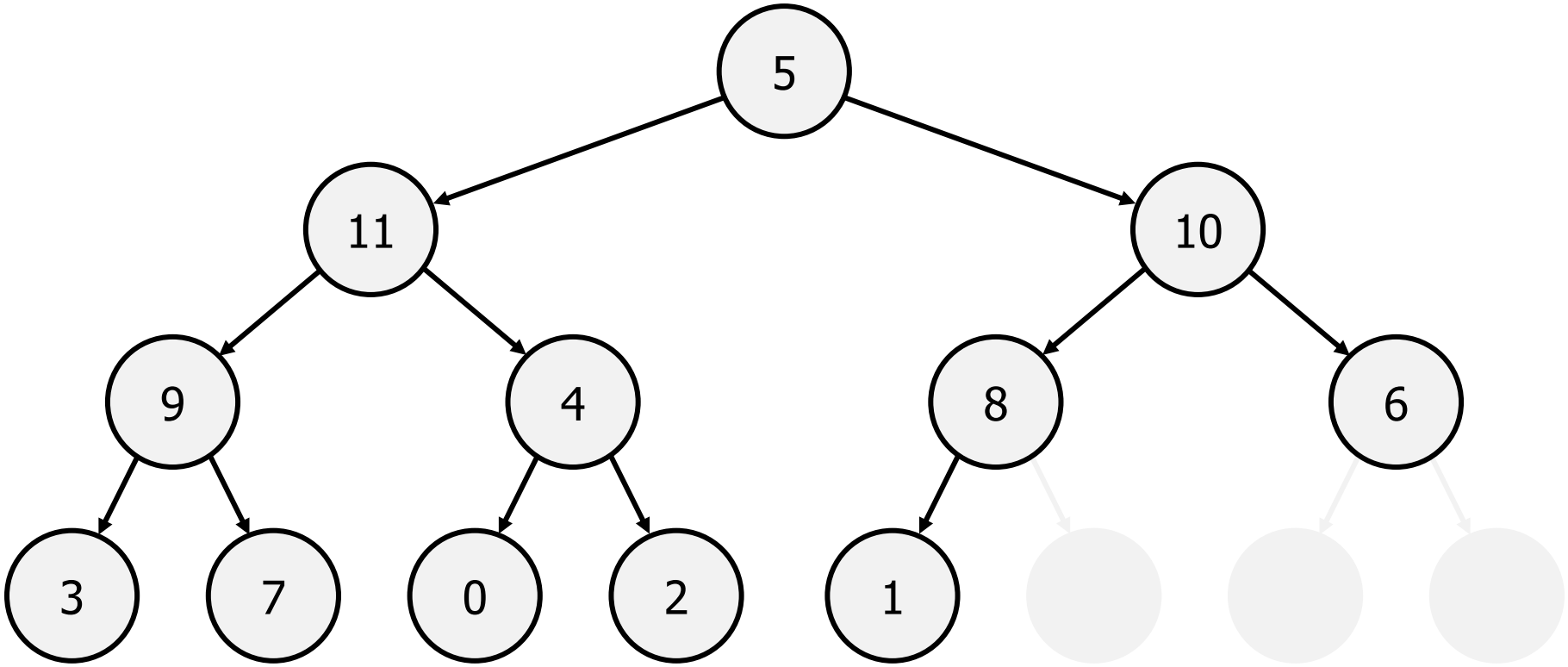
12	11	10	9	0	8	6	3	7	4	2	1	5	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down Complete



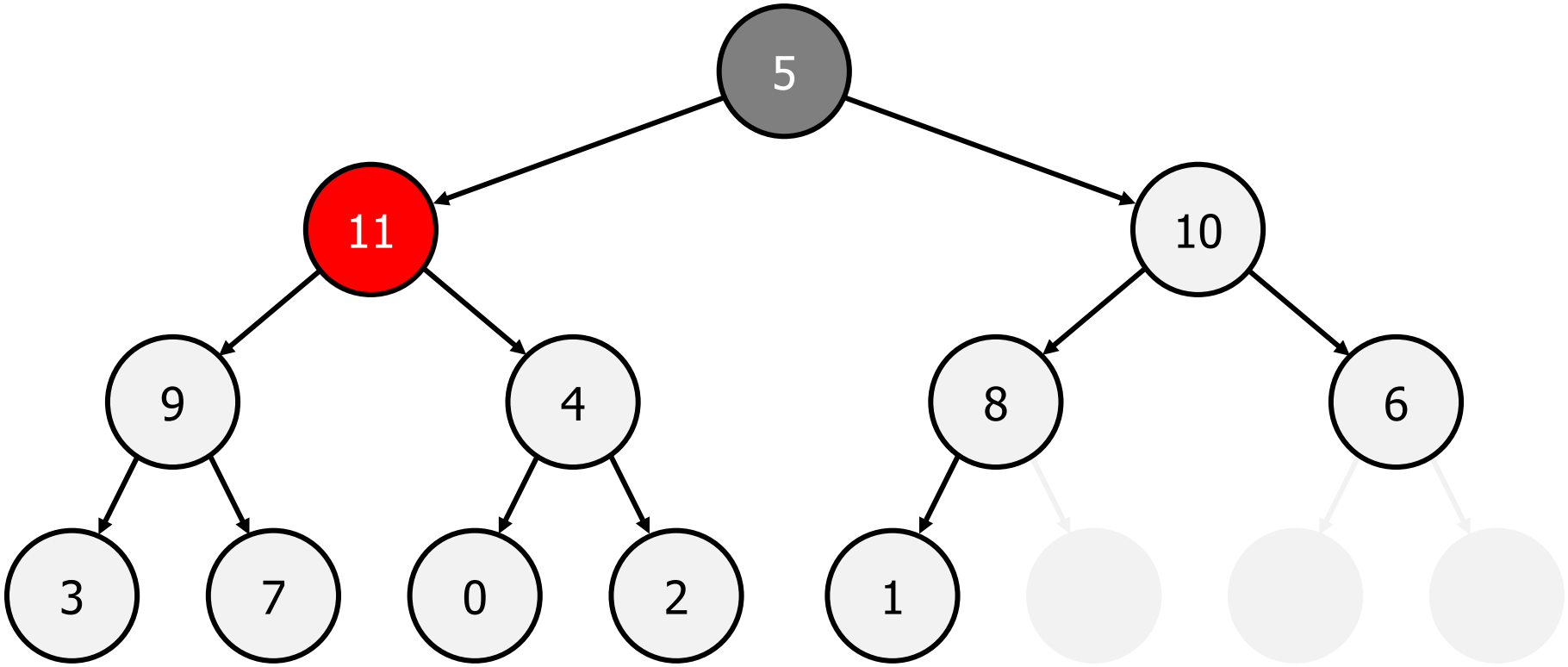
12	11	10	9	4	8	6	3	7	0	2	1	5	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Remove Largest; Replace With Last Element



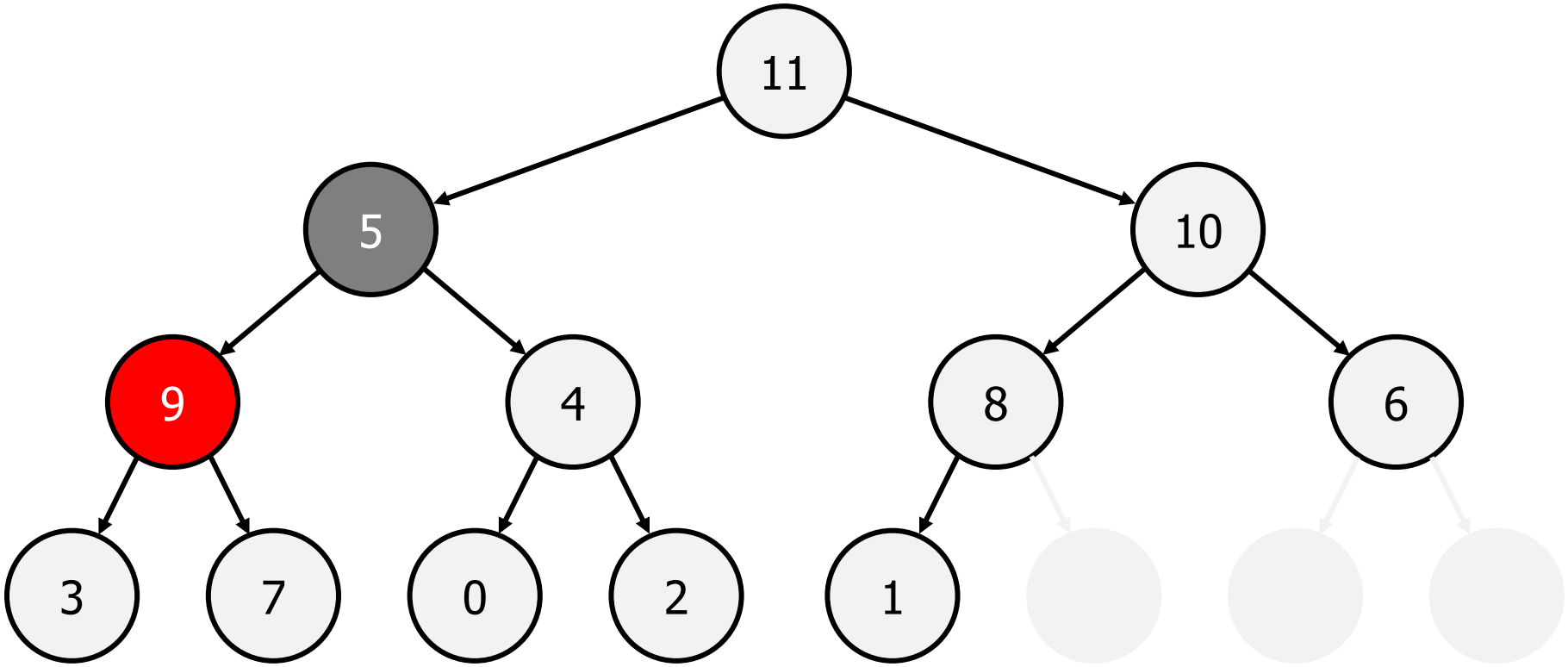
5	11	10	9	4	8	6	3	7	0	2	1	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



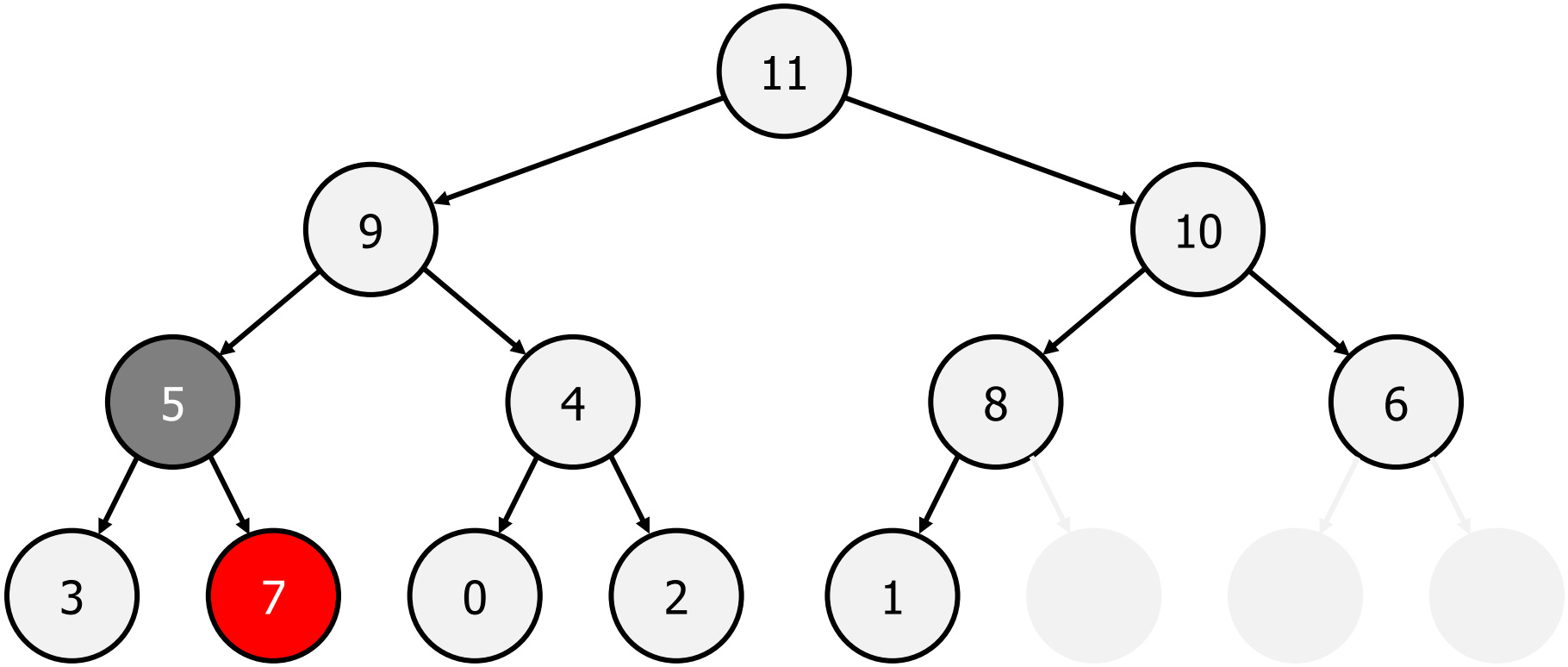
5	11	10	9	4	8	6	3	7	0	2	1	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



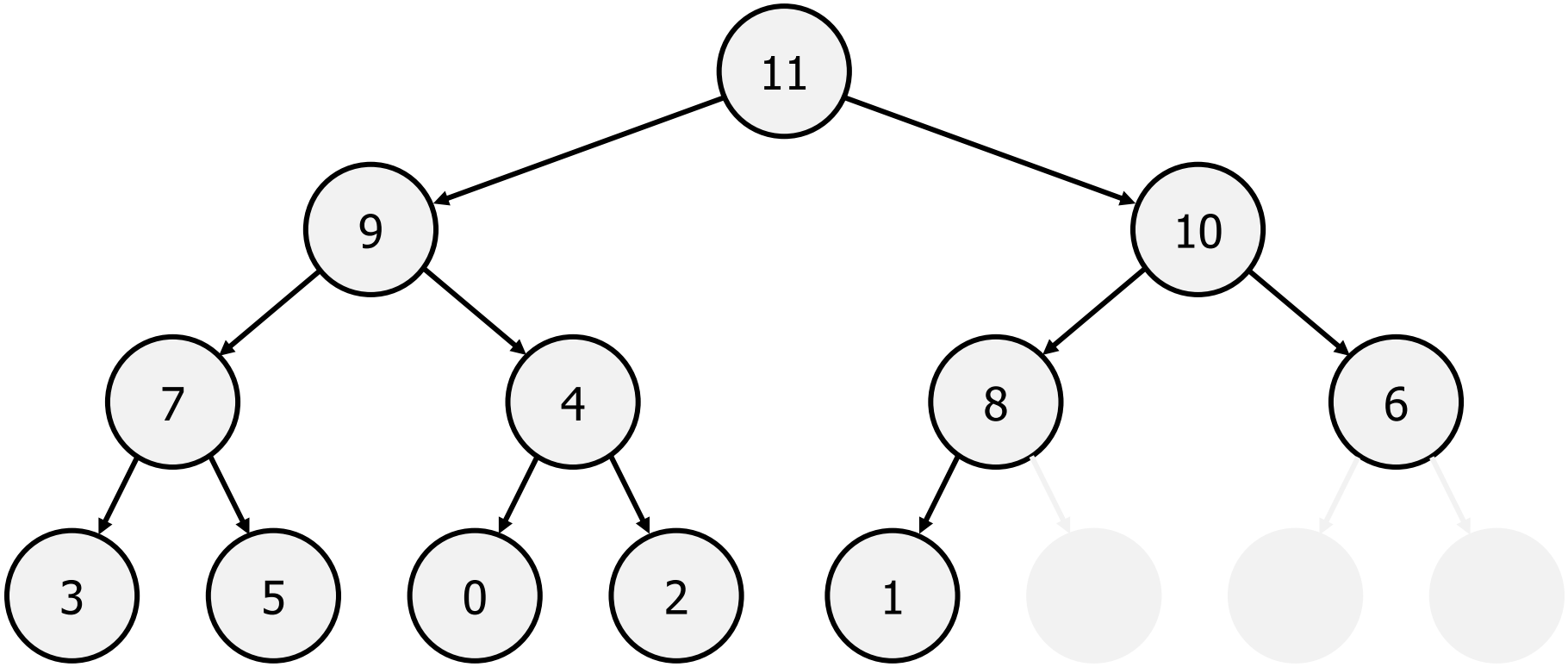
11	5	10	9	4	8	6	3	7	0	2	1	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



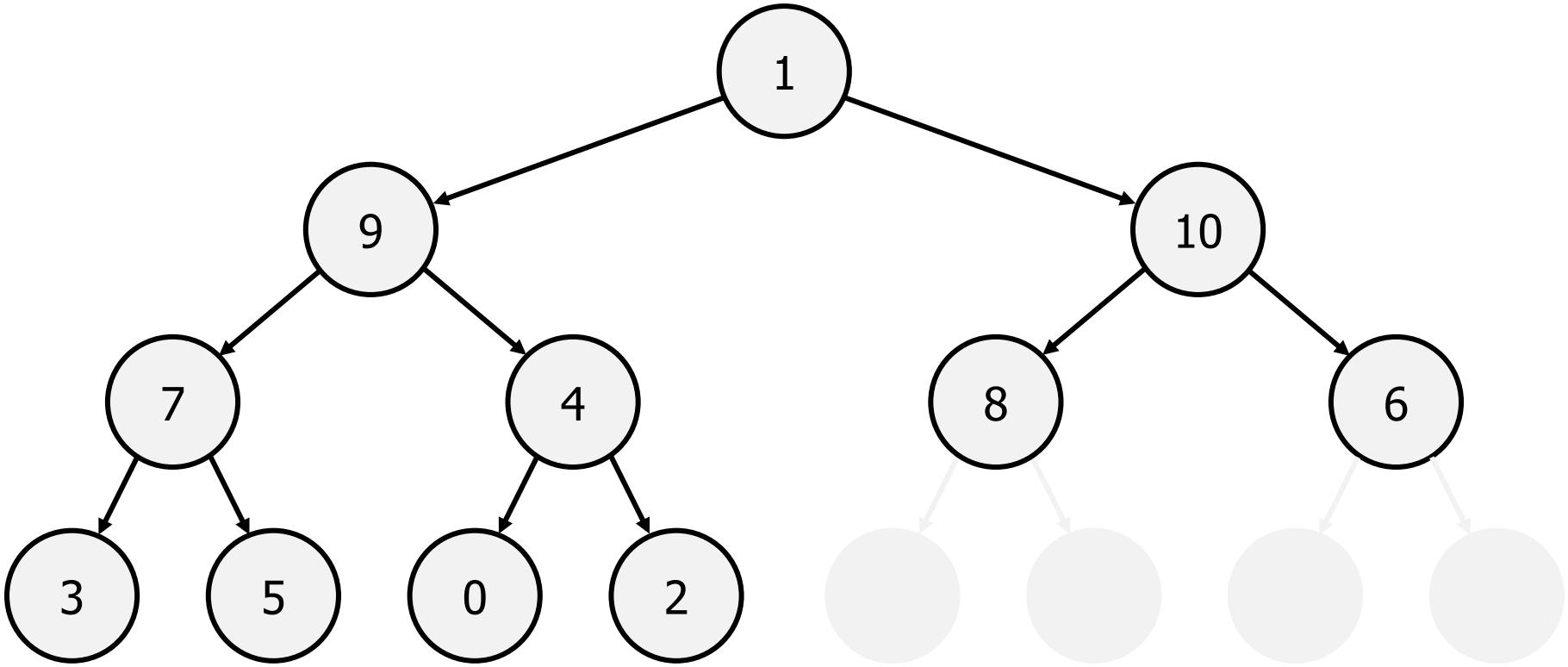
11	9	10	5	4	8	6	3	7	0	2	1	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down Complete



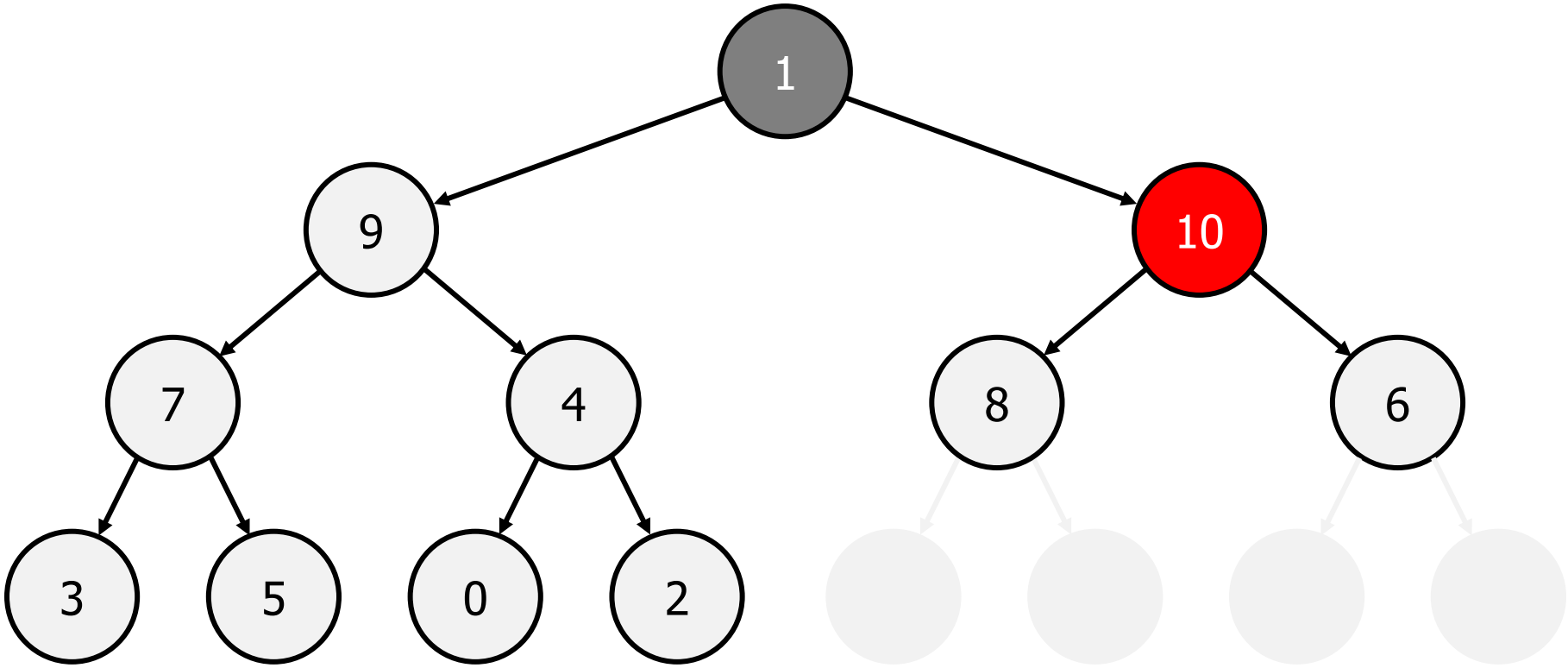
11	9	10	7	4	8	6	3	5	0	2	1	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Remove Largest; Replace With Last Element



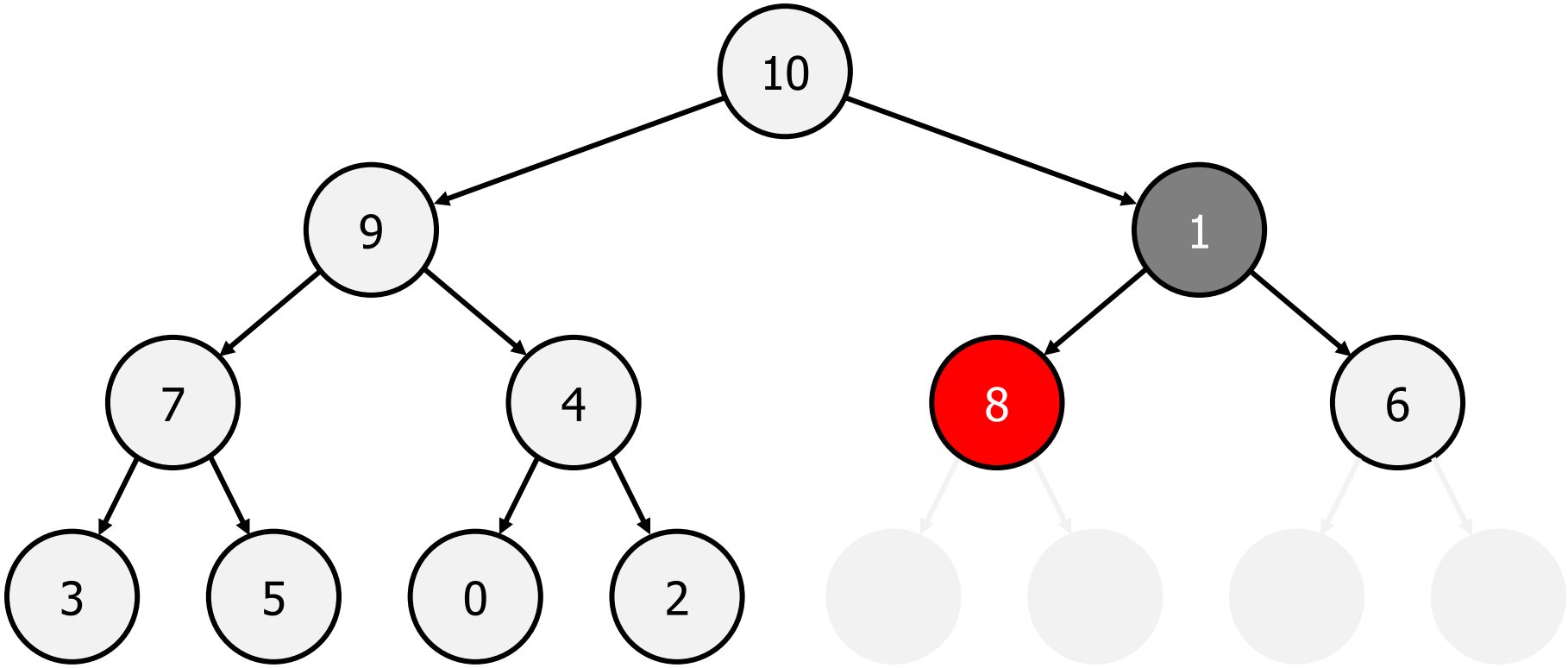
1	9	10	7	4	8	6	3	5	0	2	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



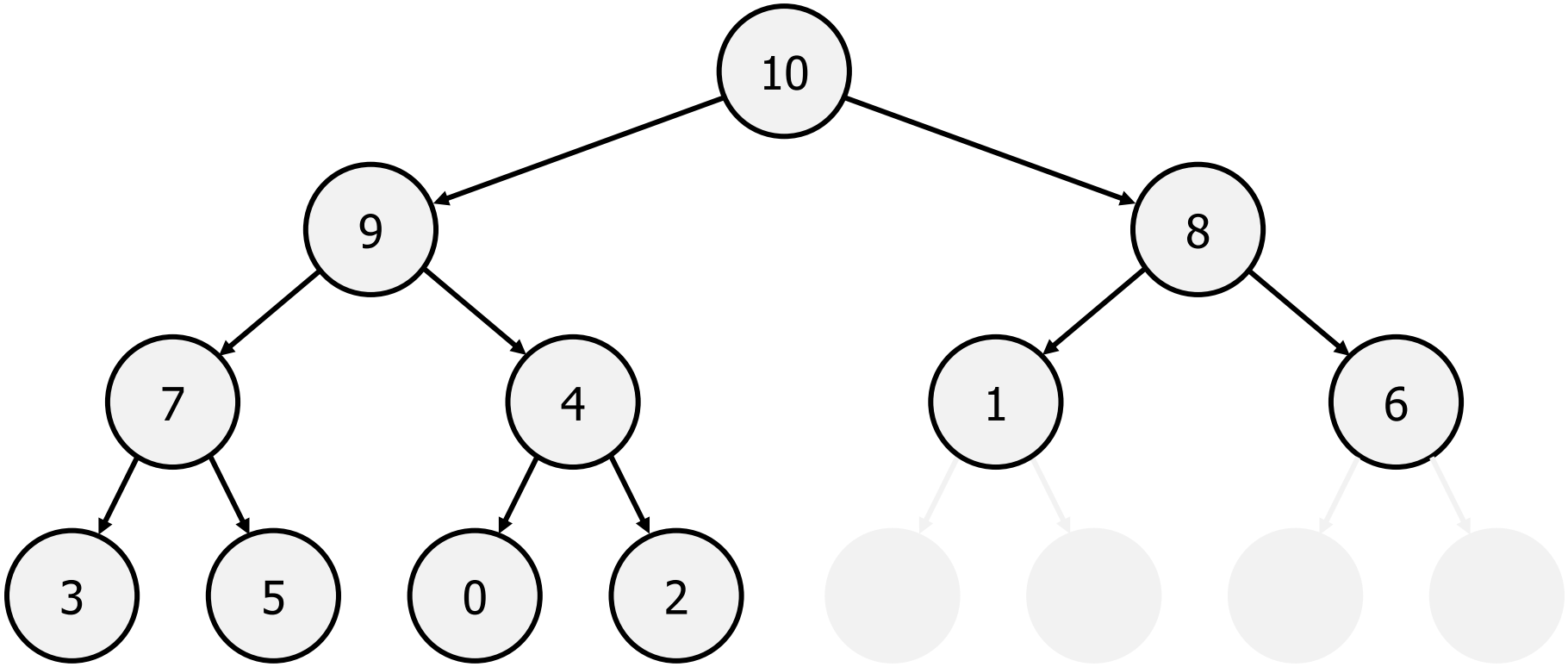
1	9	10	7	4	8	6	3	5	0	2	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



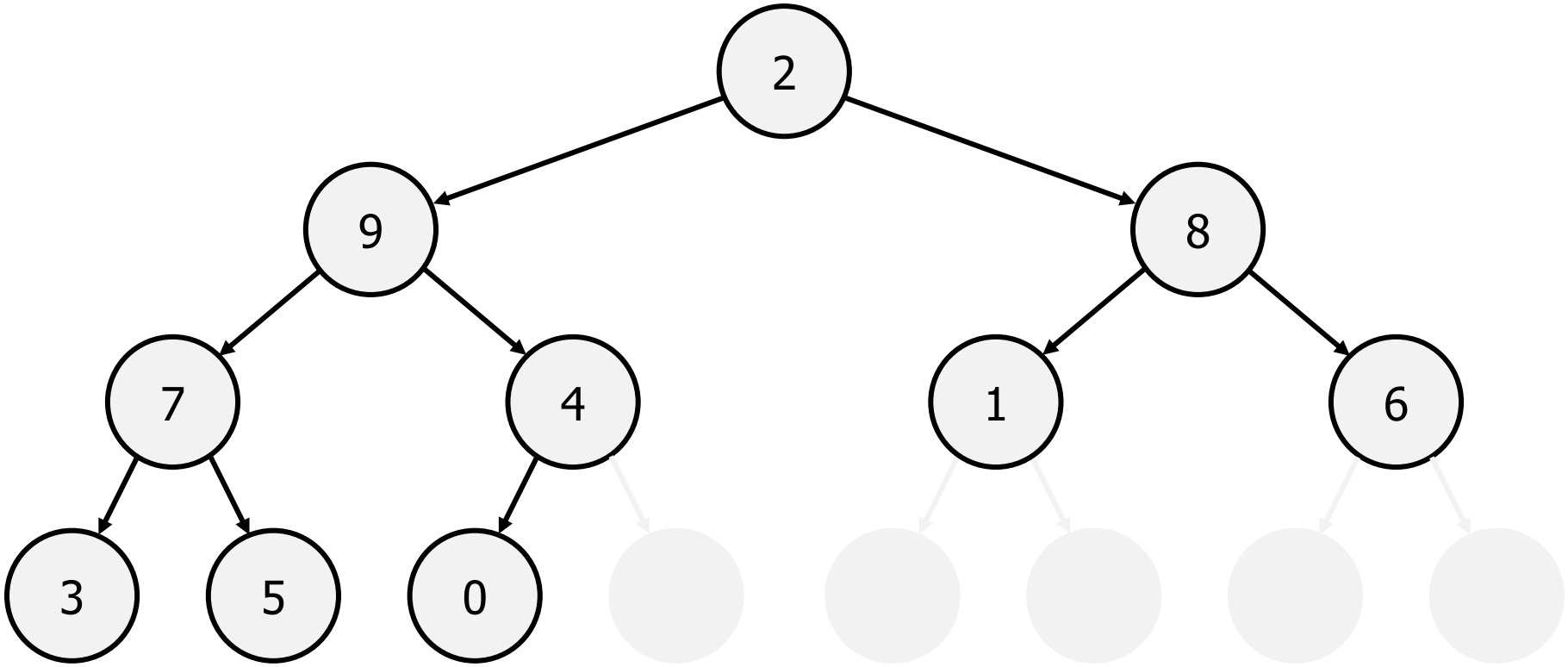
10	9	1	7	4	8	6	3	5	0	2	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down Complete



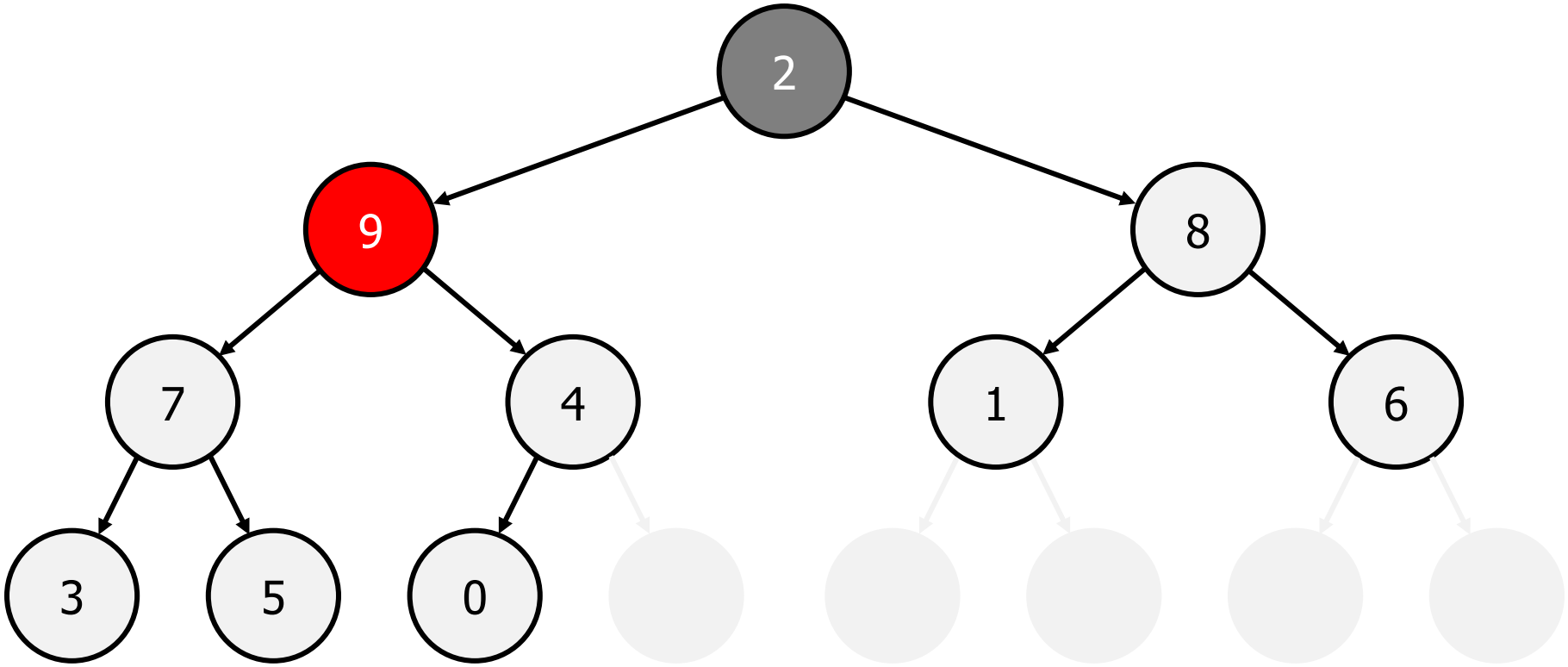
10	9	8	7	4	1	6	3	5	0	2	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Remove Largest; Replace With Last Element



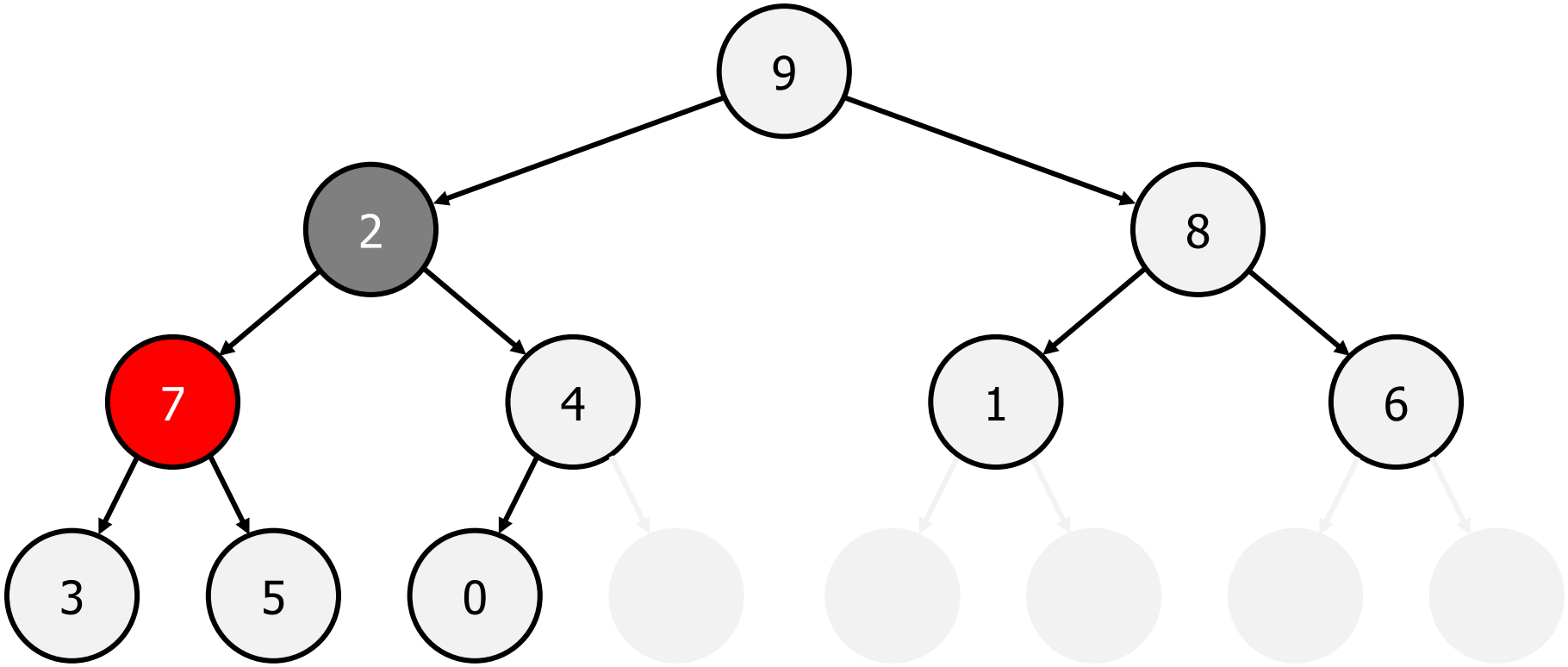
2	9	8	7	4	1	6	3	5	0	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



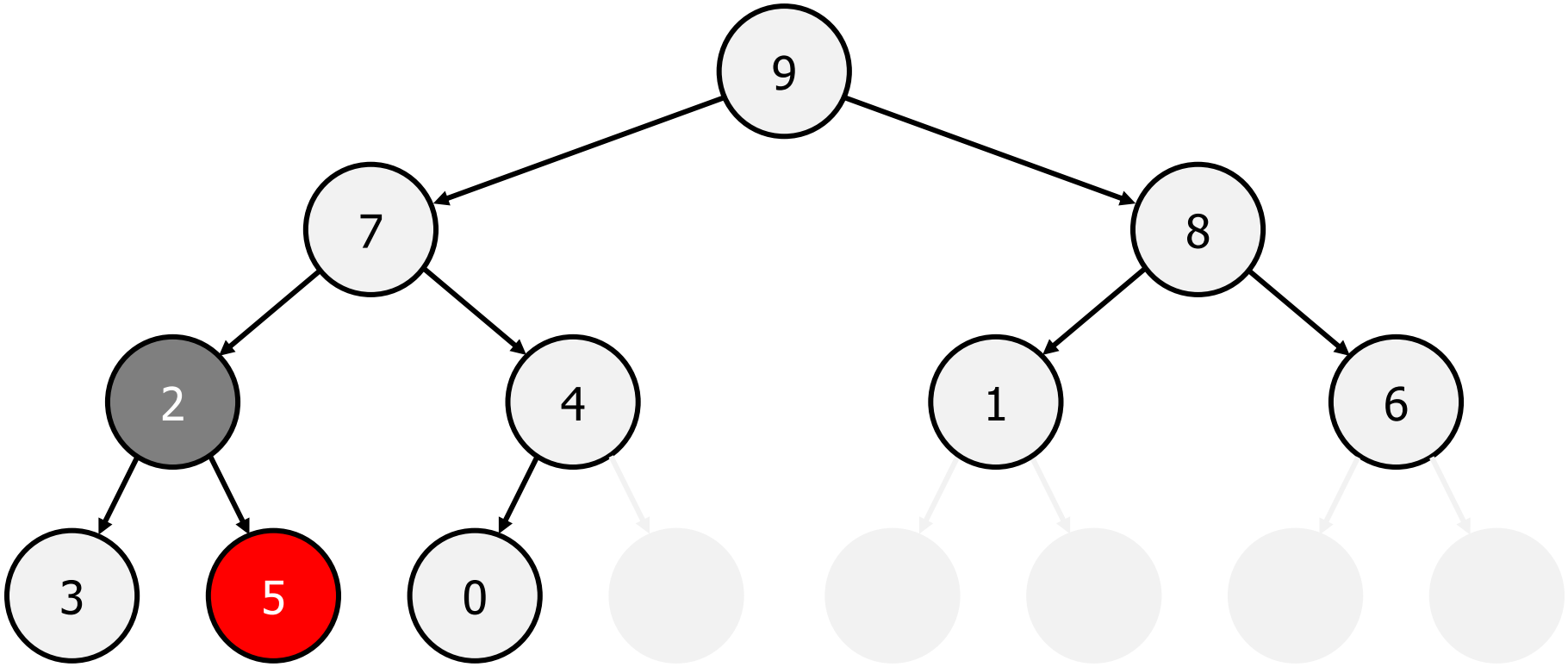
2	9	8	7	4	1	6	3	5	0	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



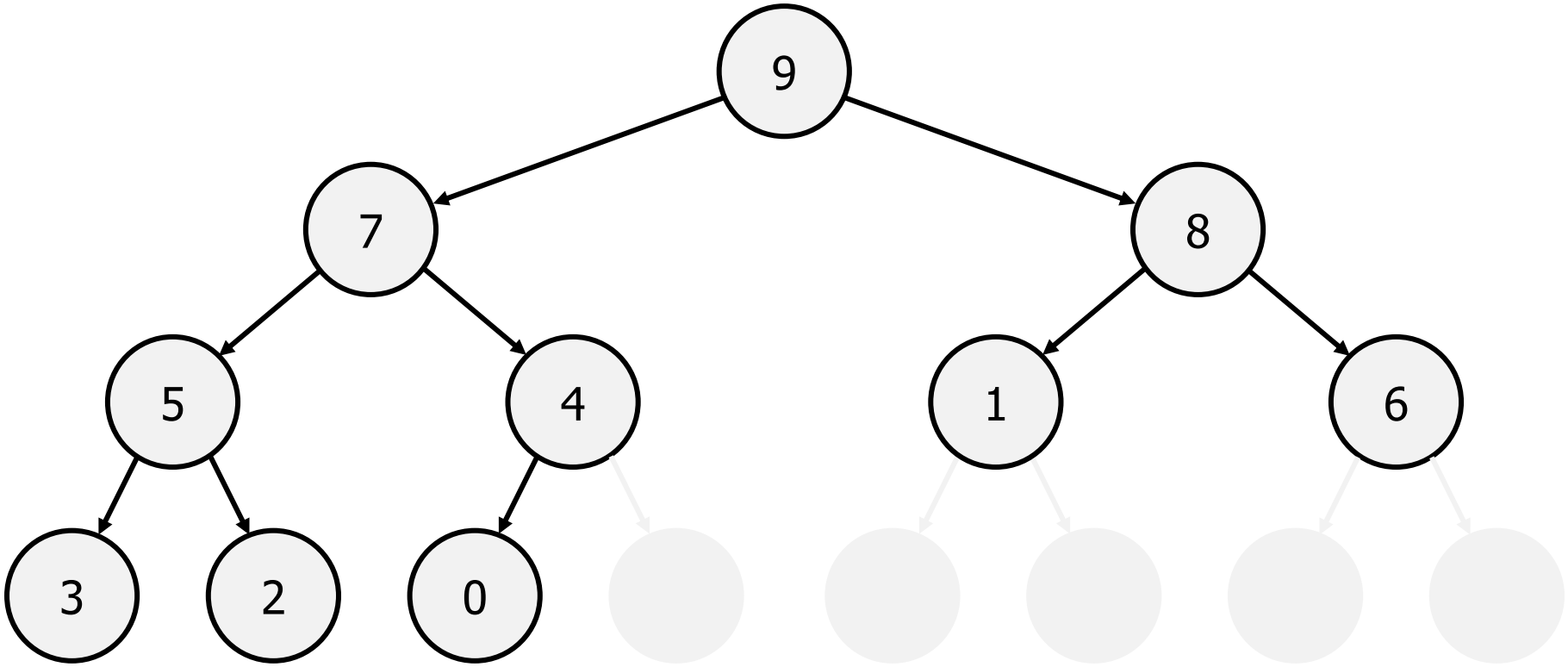
9	2	8	7	4	1	6	3	5	0	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



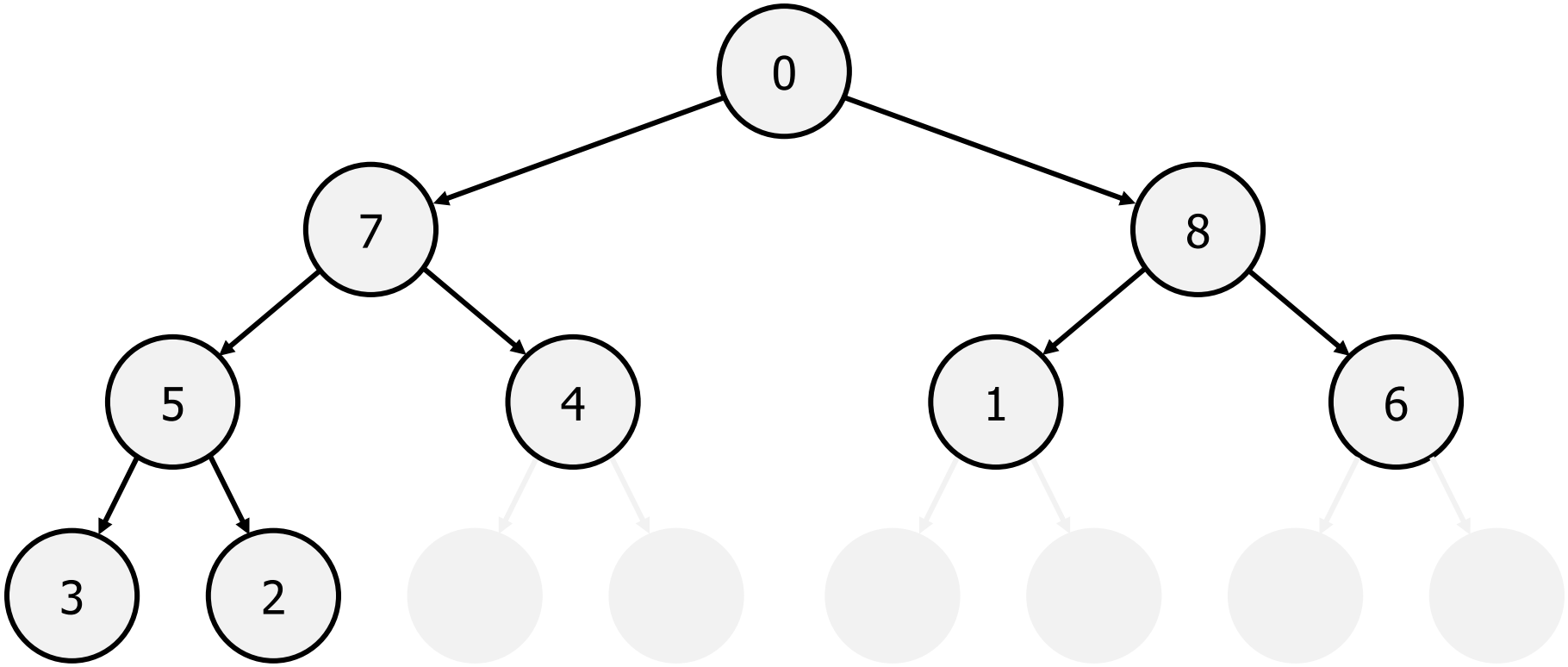
9	7	8	2	4	1	6	3	5	0	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down Complete



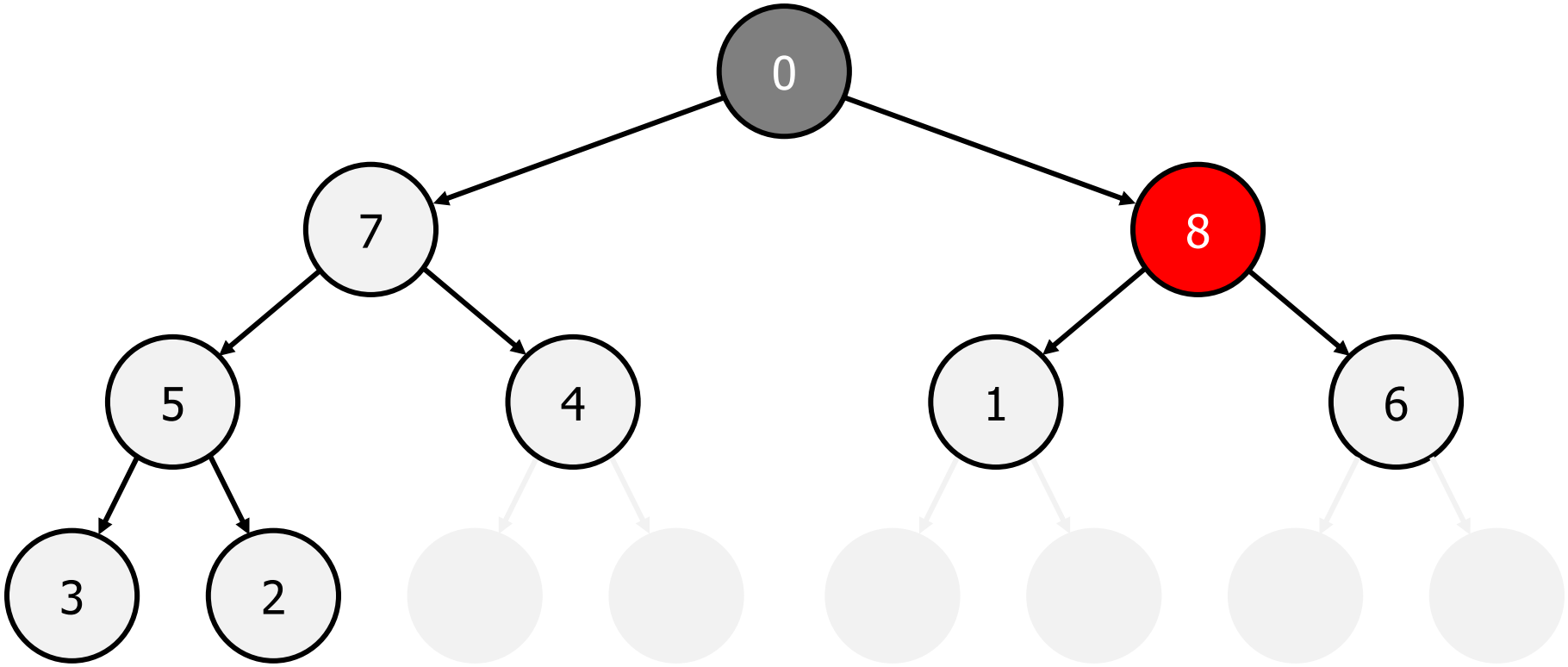
9	7	8	5	4	1	6	3	2	0	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Remove Largest; Replace With Last Element



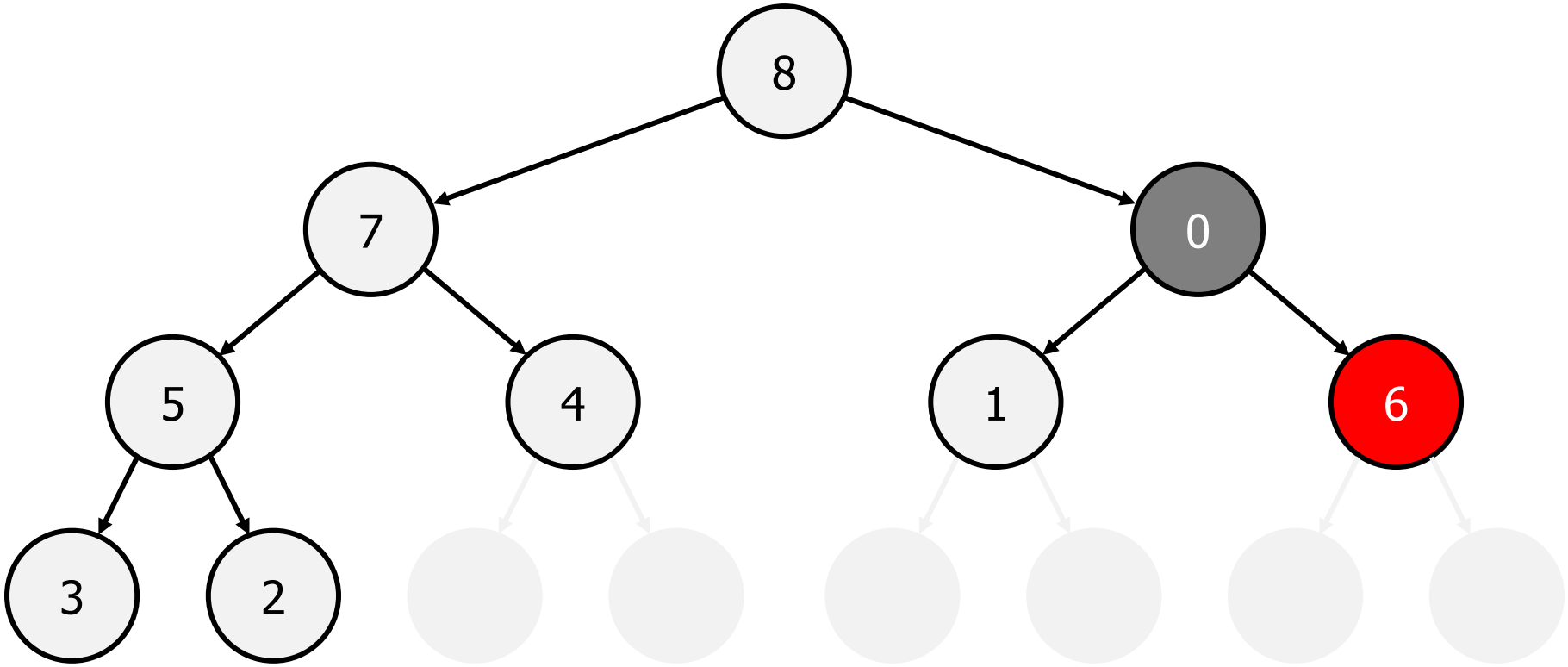
0	7	8	5	4	1	6	3	2	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



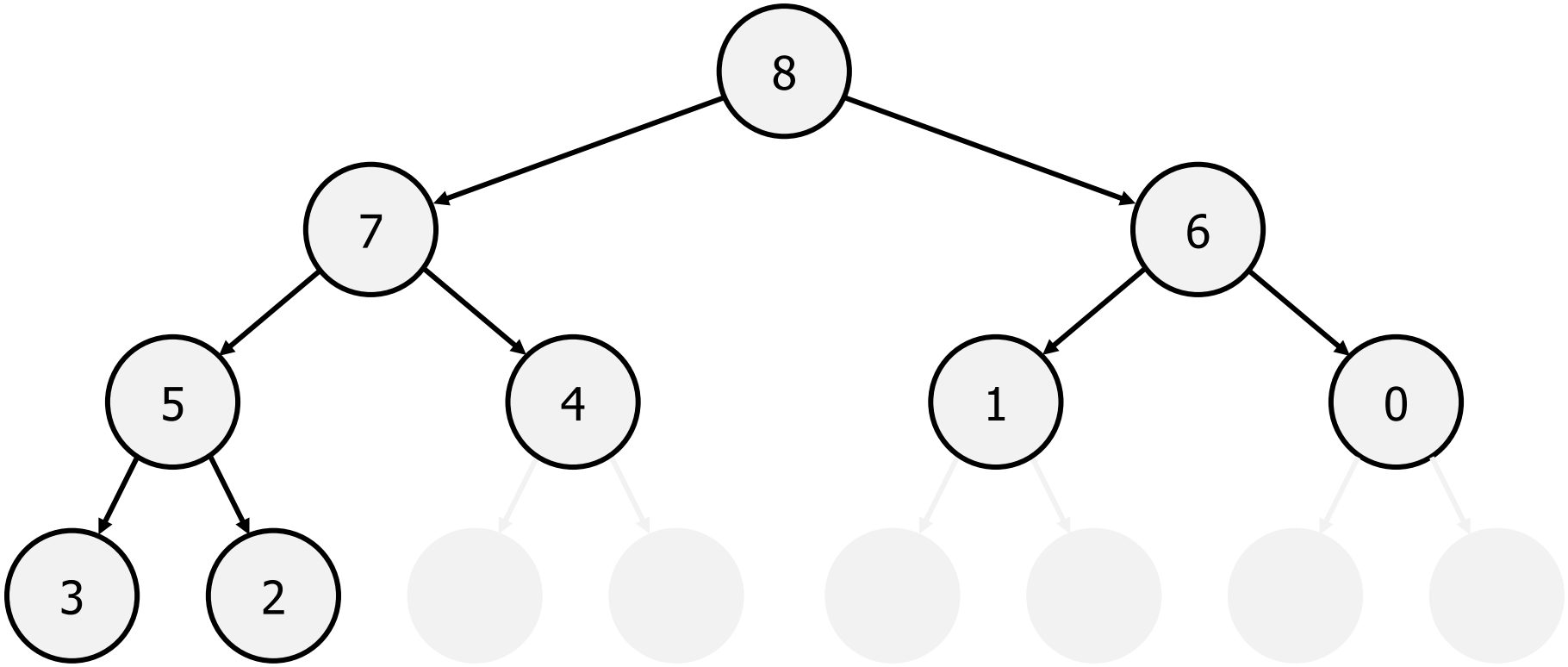
0	7	8	5	4	1	6	3	2	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



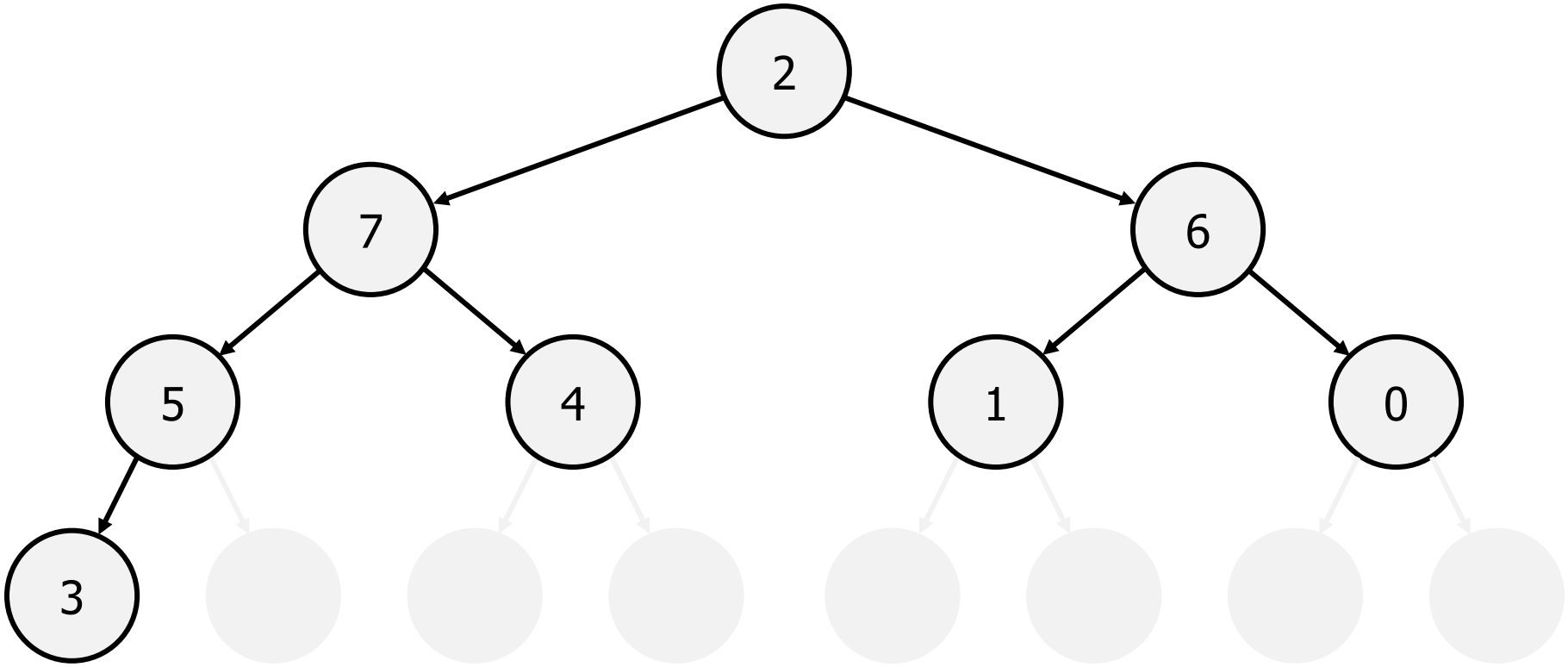
8	7	0	5	4	1	6	3	2	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down Complete



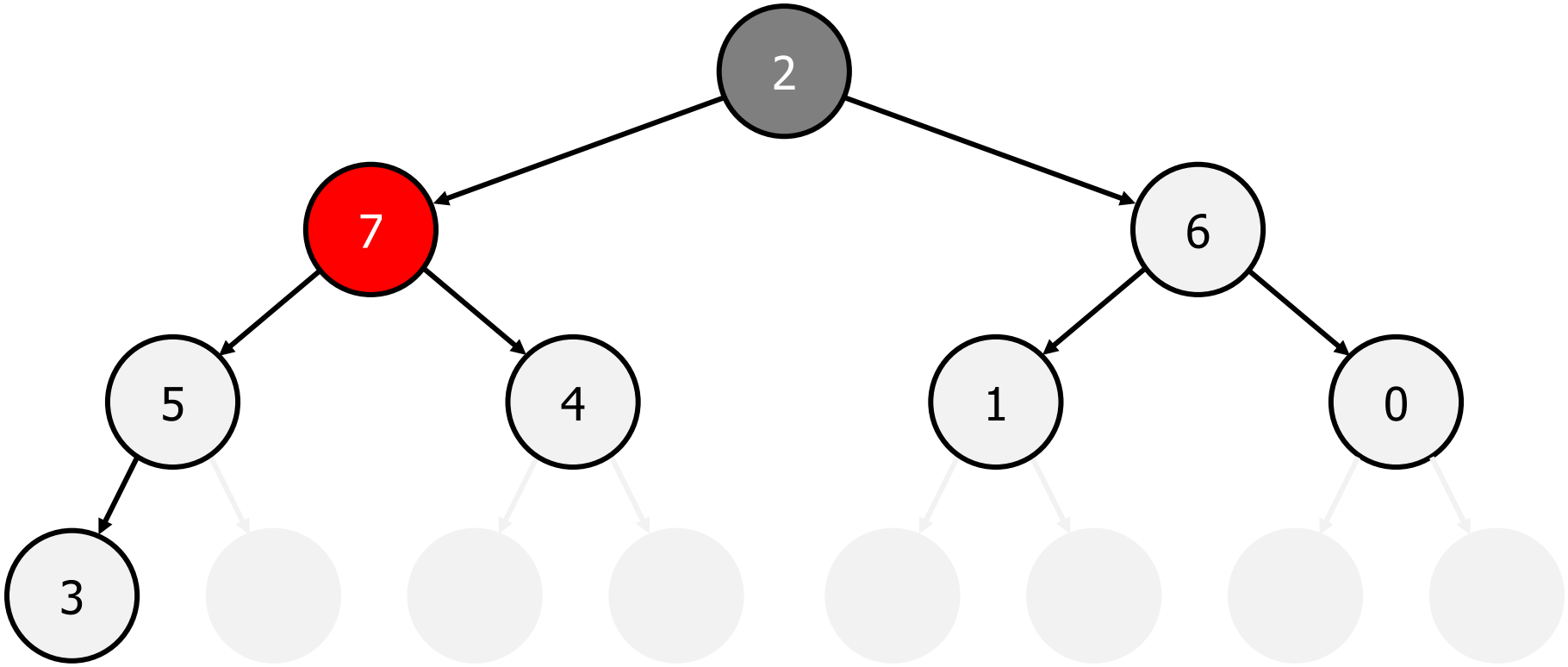
8	7	6	5	4	1	0	3	2	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Remove Largest; Replace With Last Element



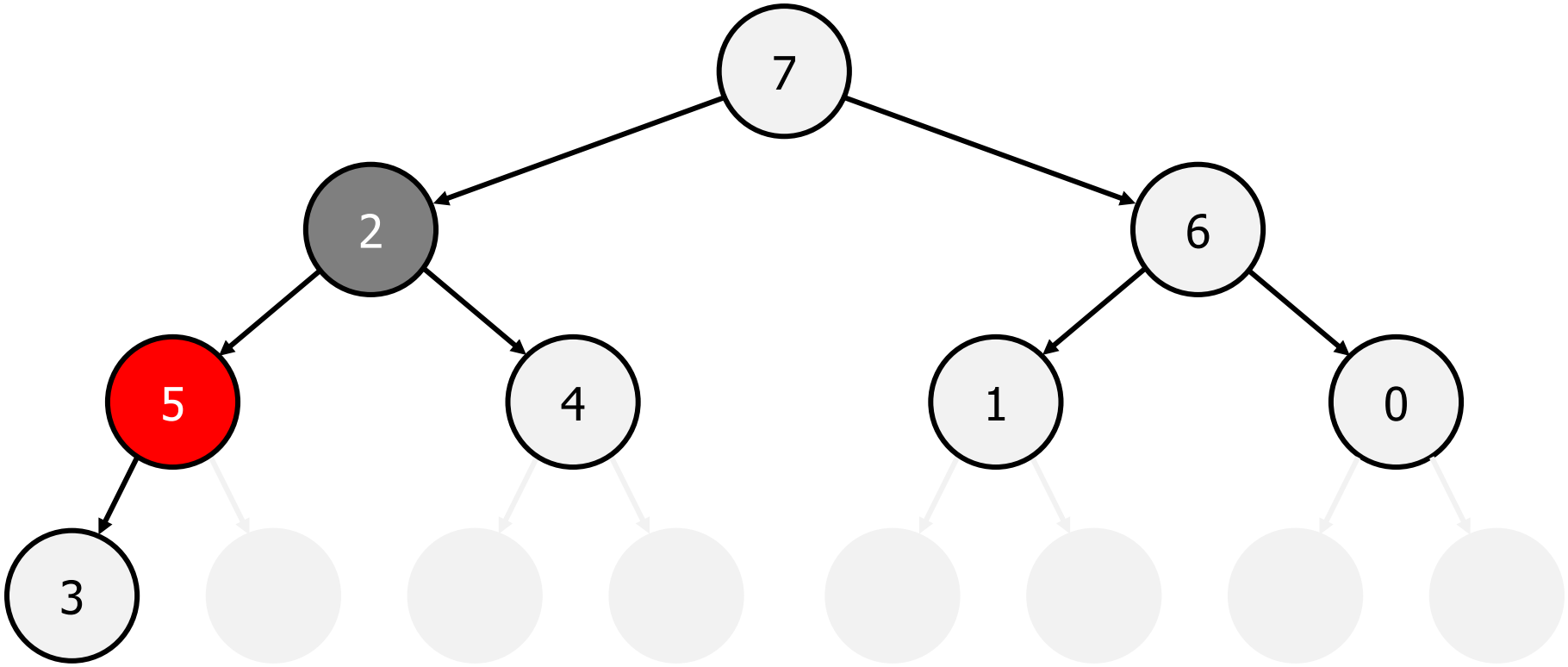
2	7	6	5	4	1	0	3	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



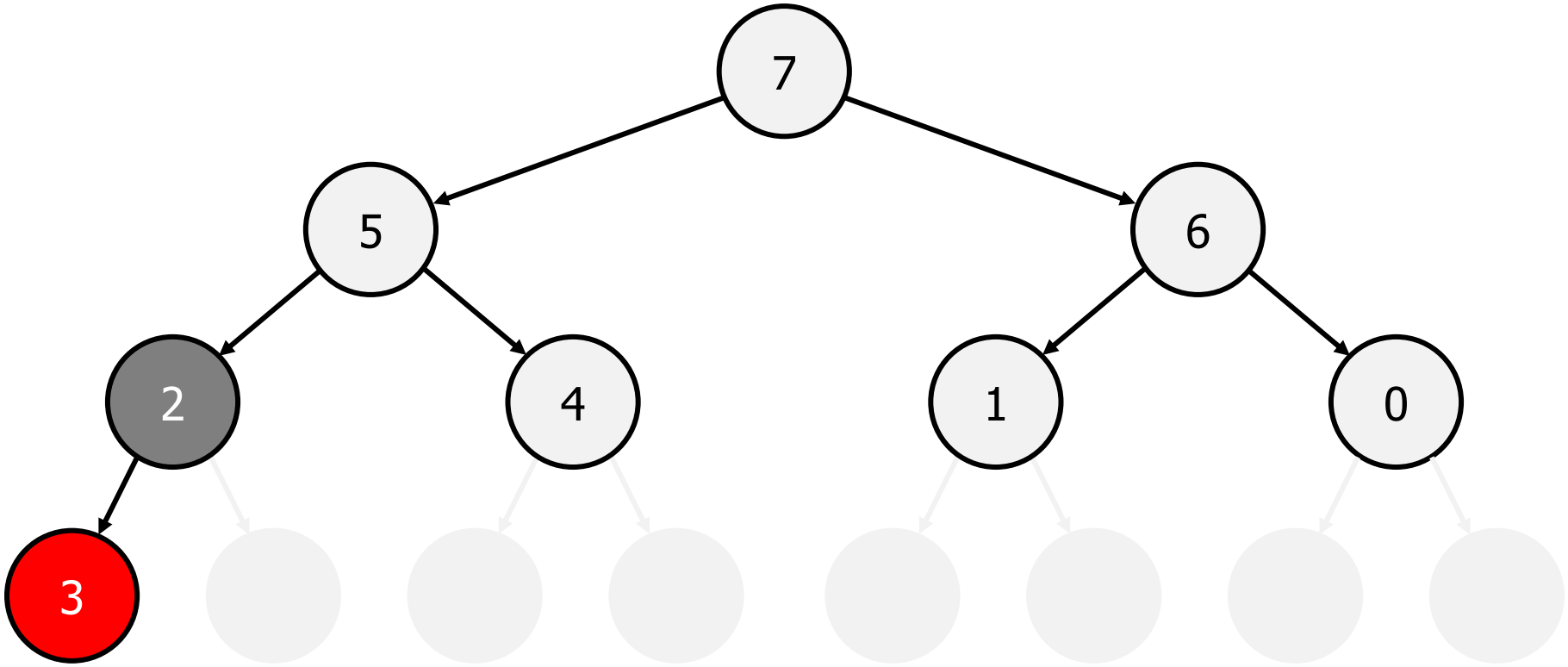
2	7	6	5	4	1	0	3	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



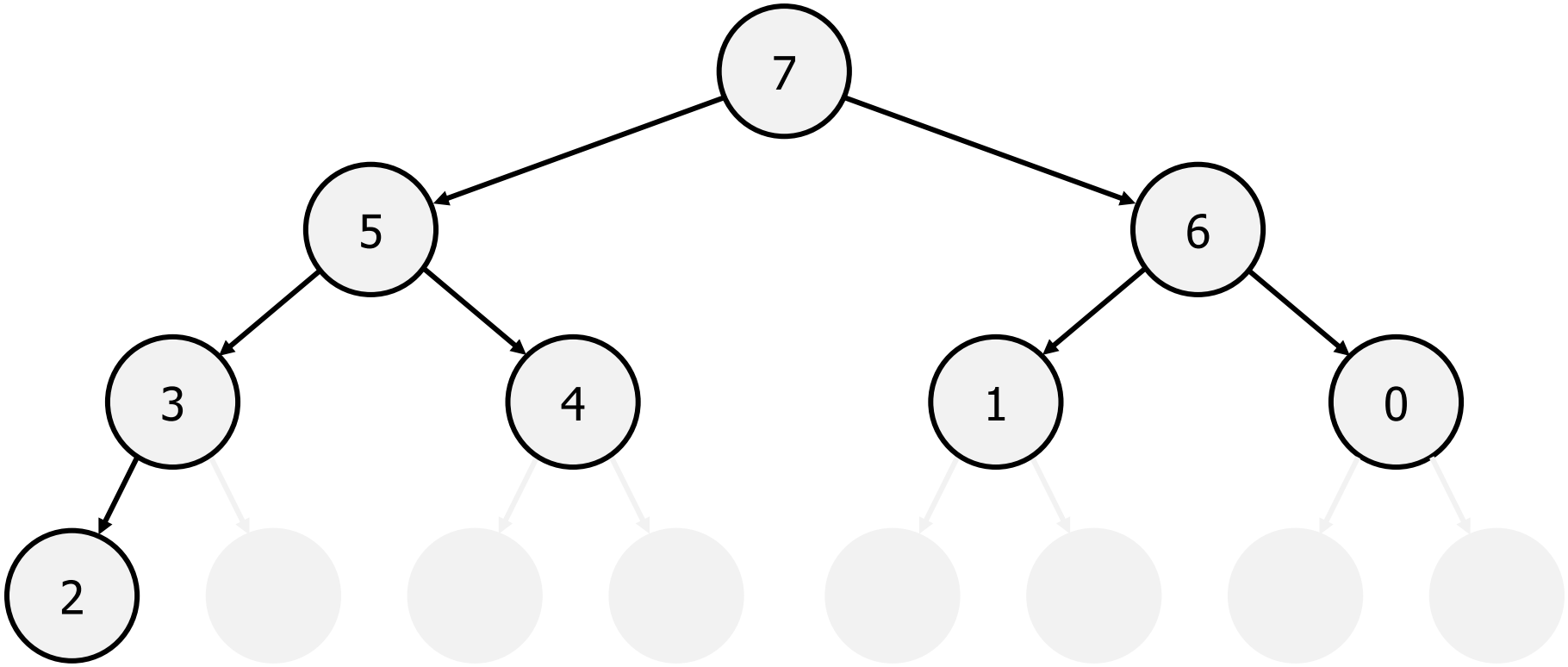
7	2	6	5	4	1	0	3	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



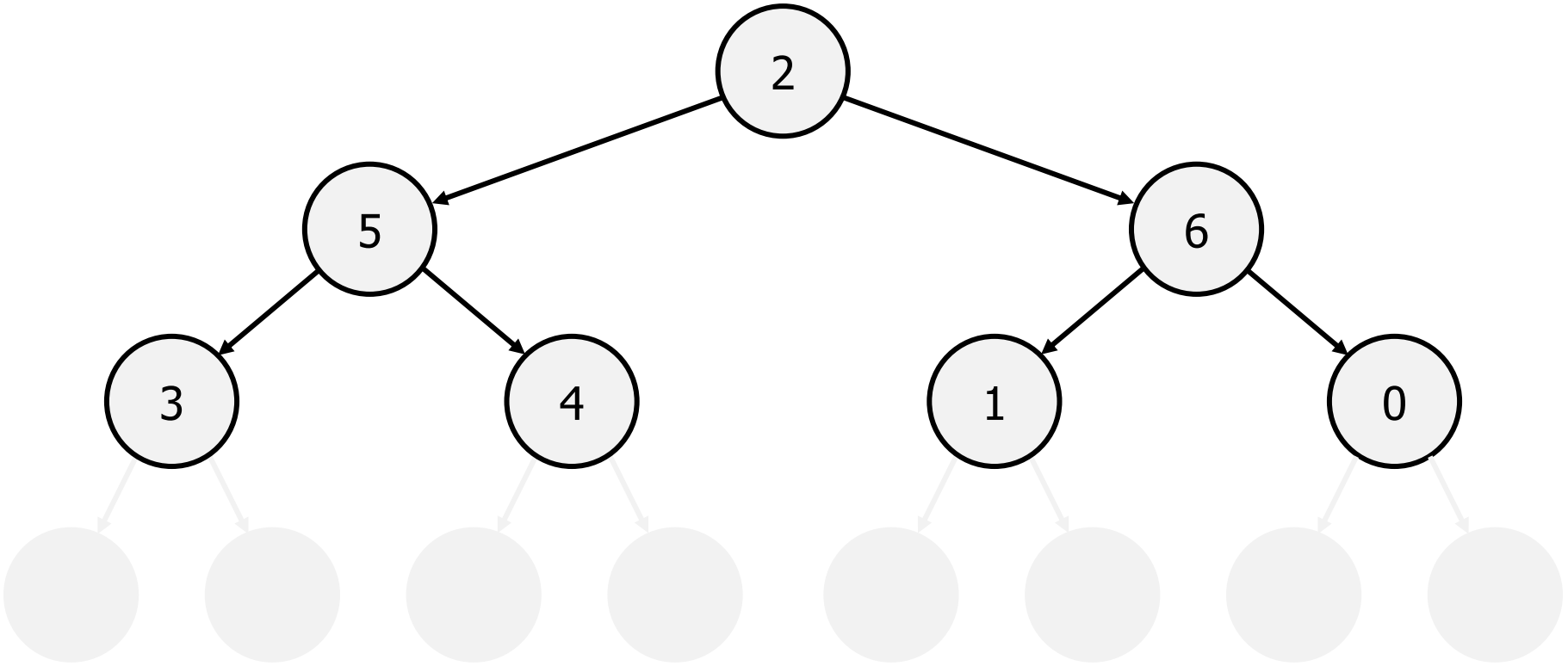
7	5	6	2	4	1	0	3	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down Complete



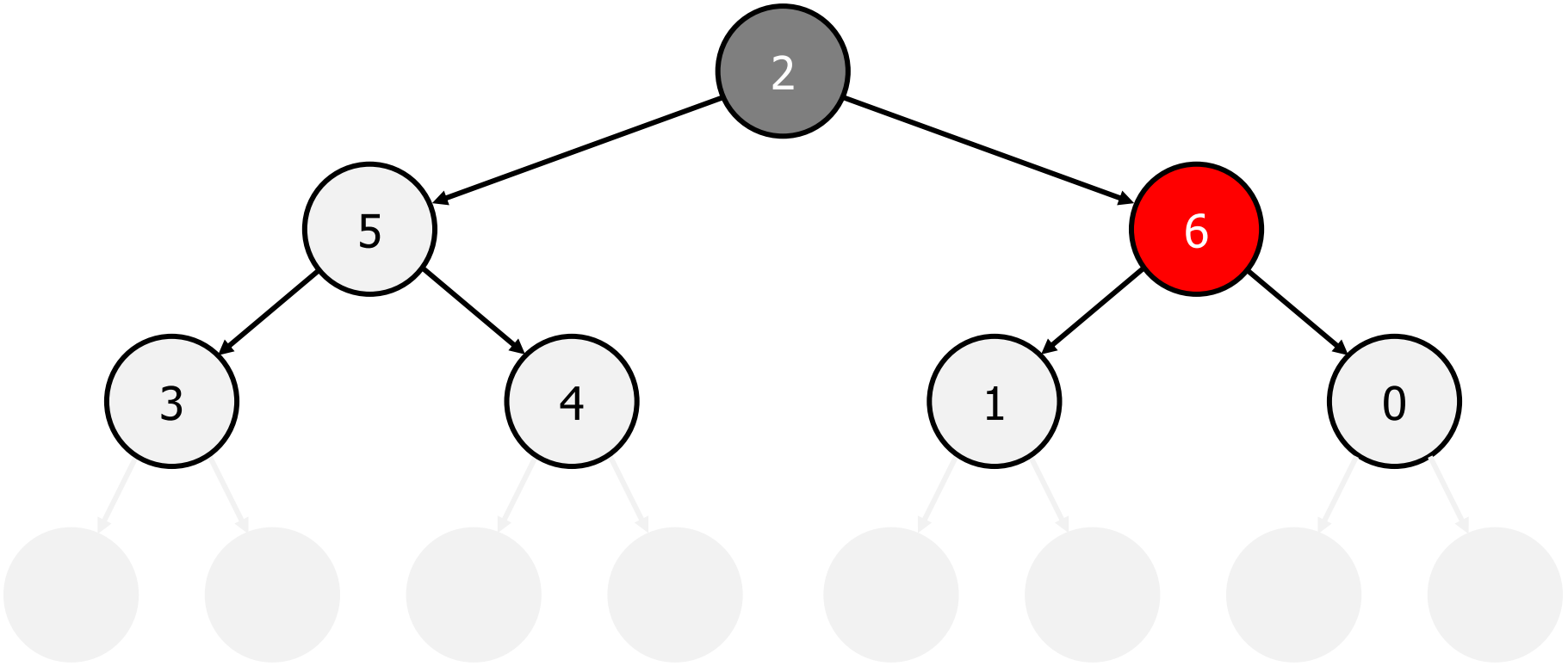
7	5	6	3	4	1	0	2	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Remove Largest; Replace With Last Element



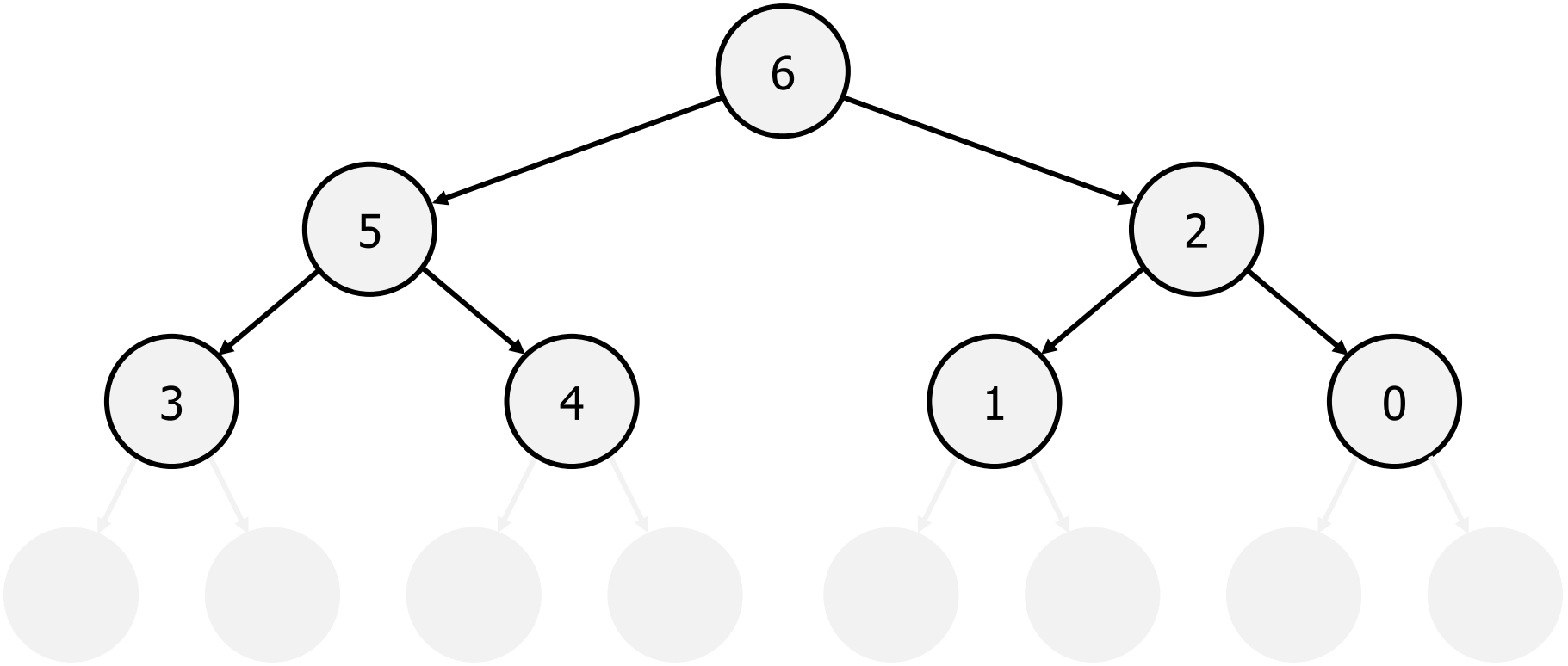
2	5	6	3	4	1	0	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



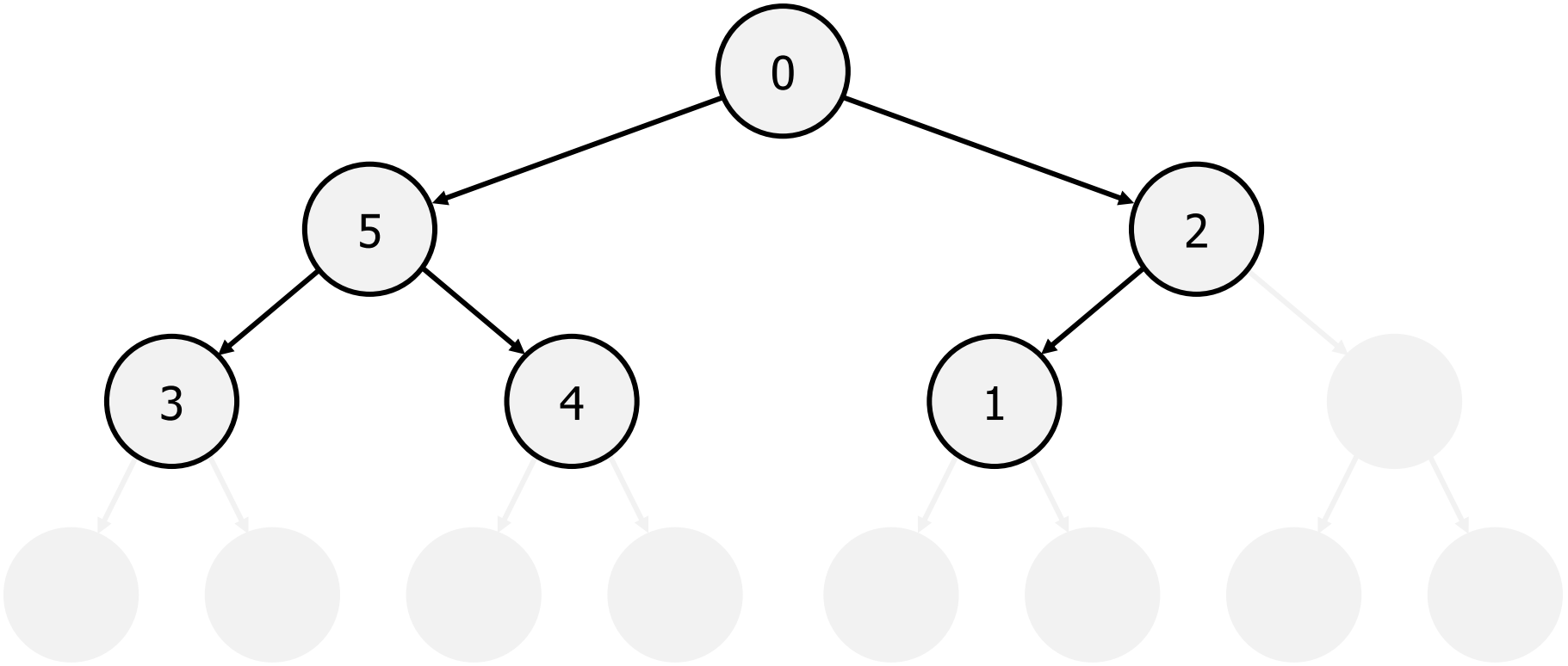
2	5	6	3	4	1	0	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Complete



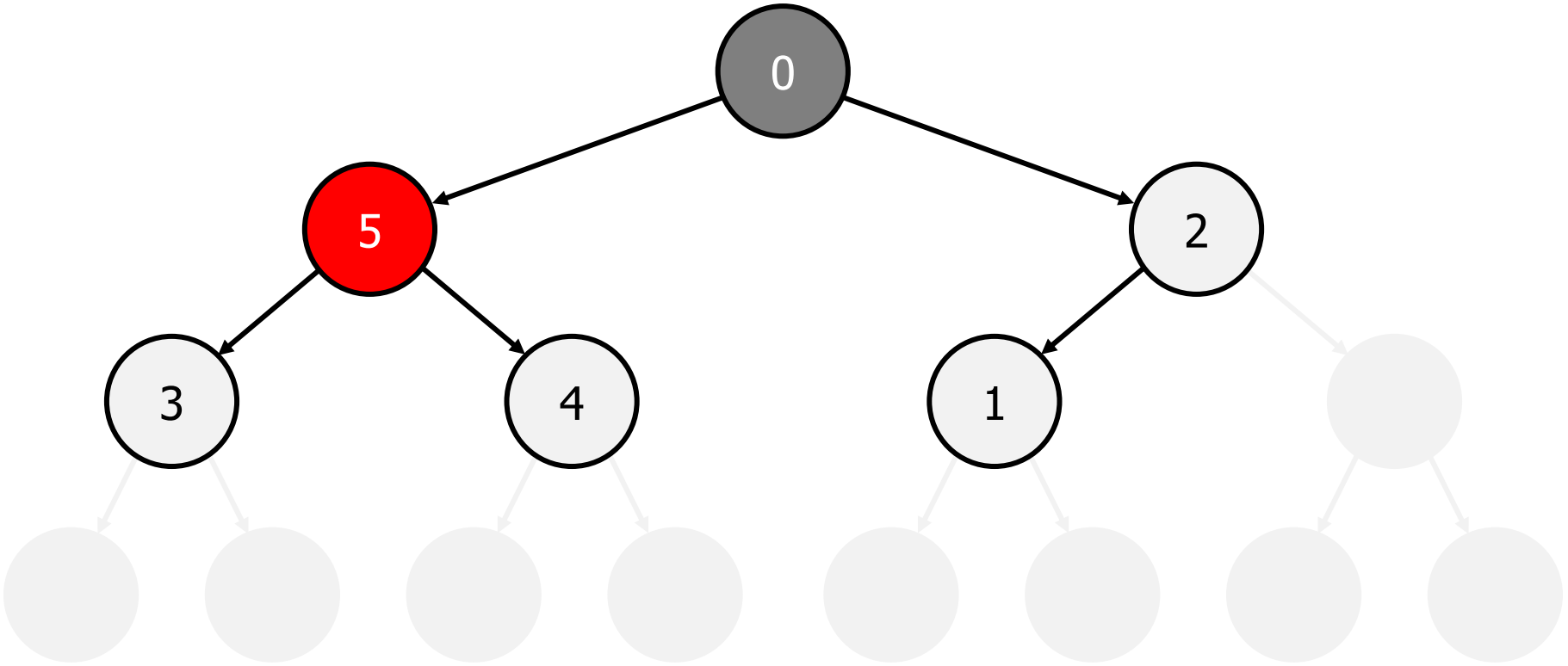
6	5	2	3	4	1	0	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Remove Largest; Replace With Final Element



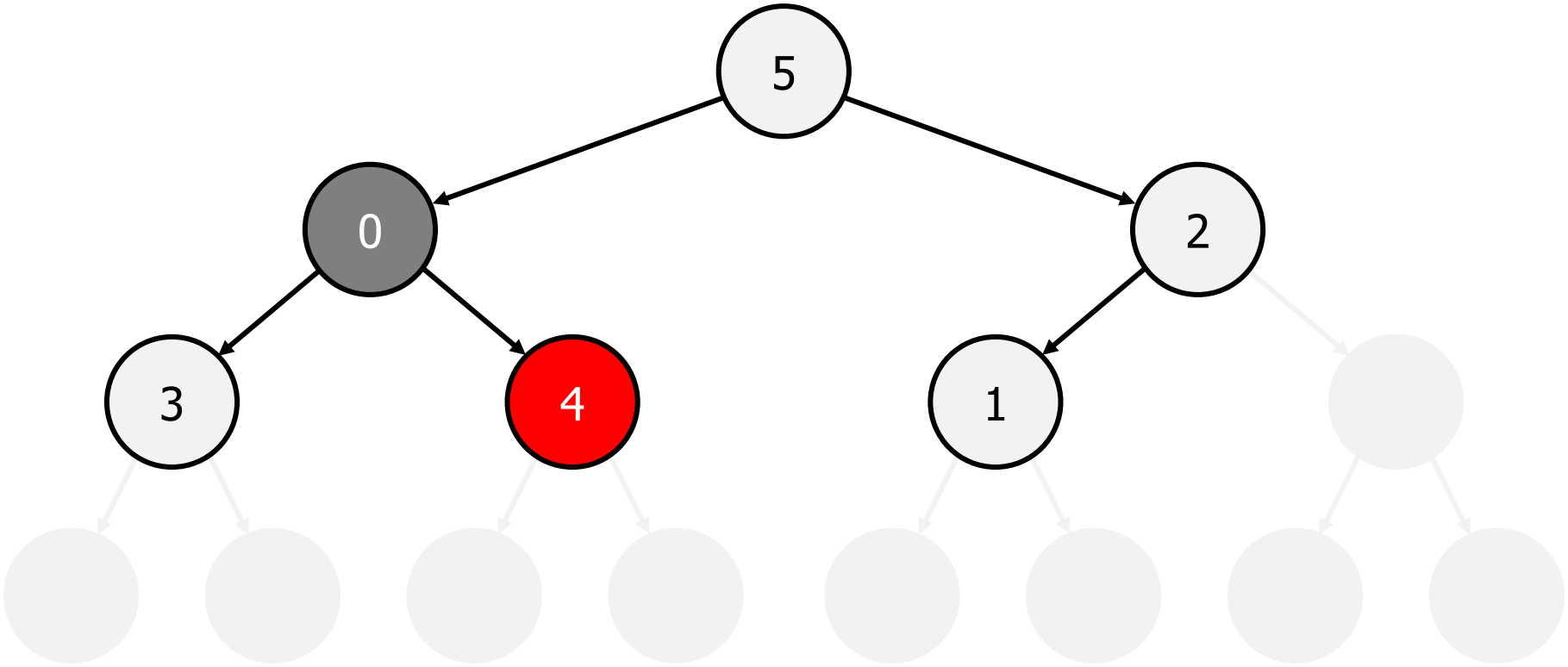
0	5	2	3	4	1	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



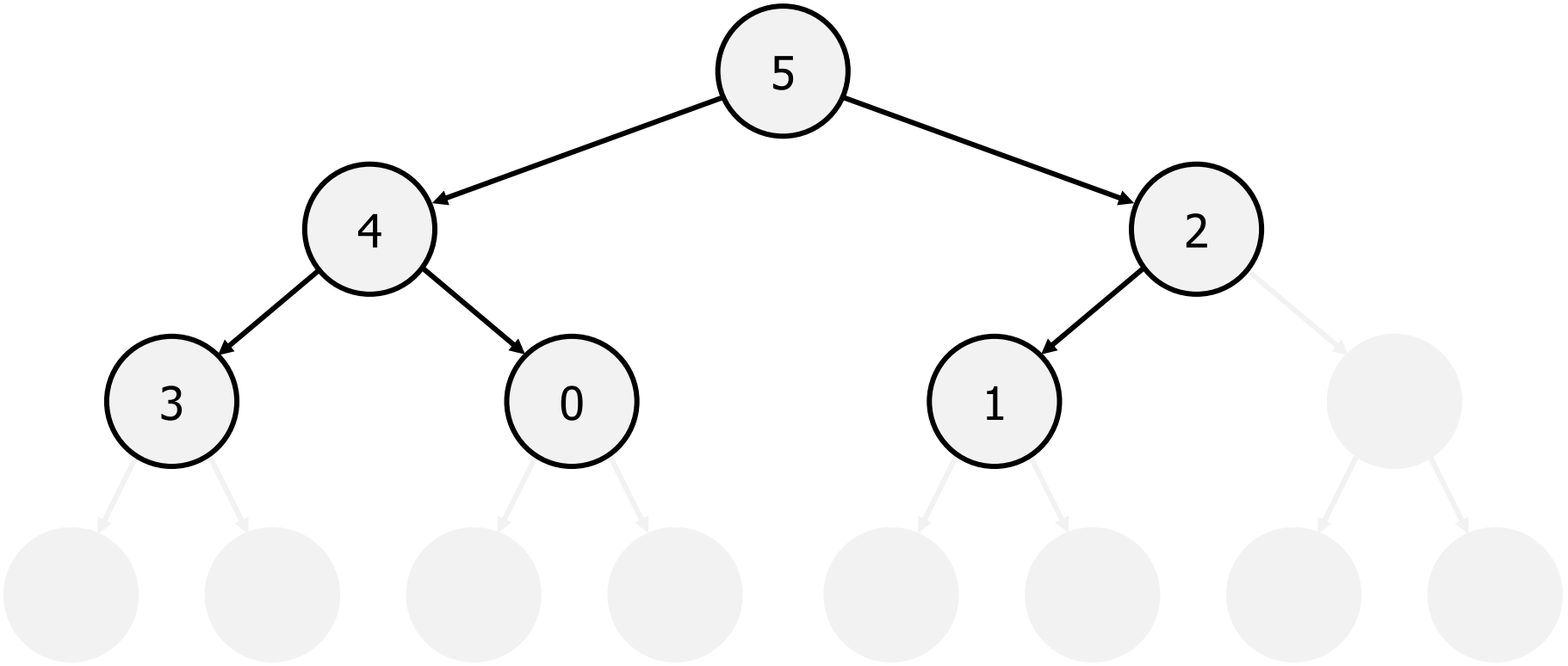
0	5	2	3	4	1	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



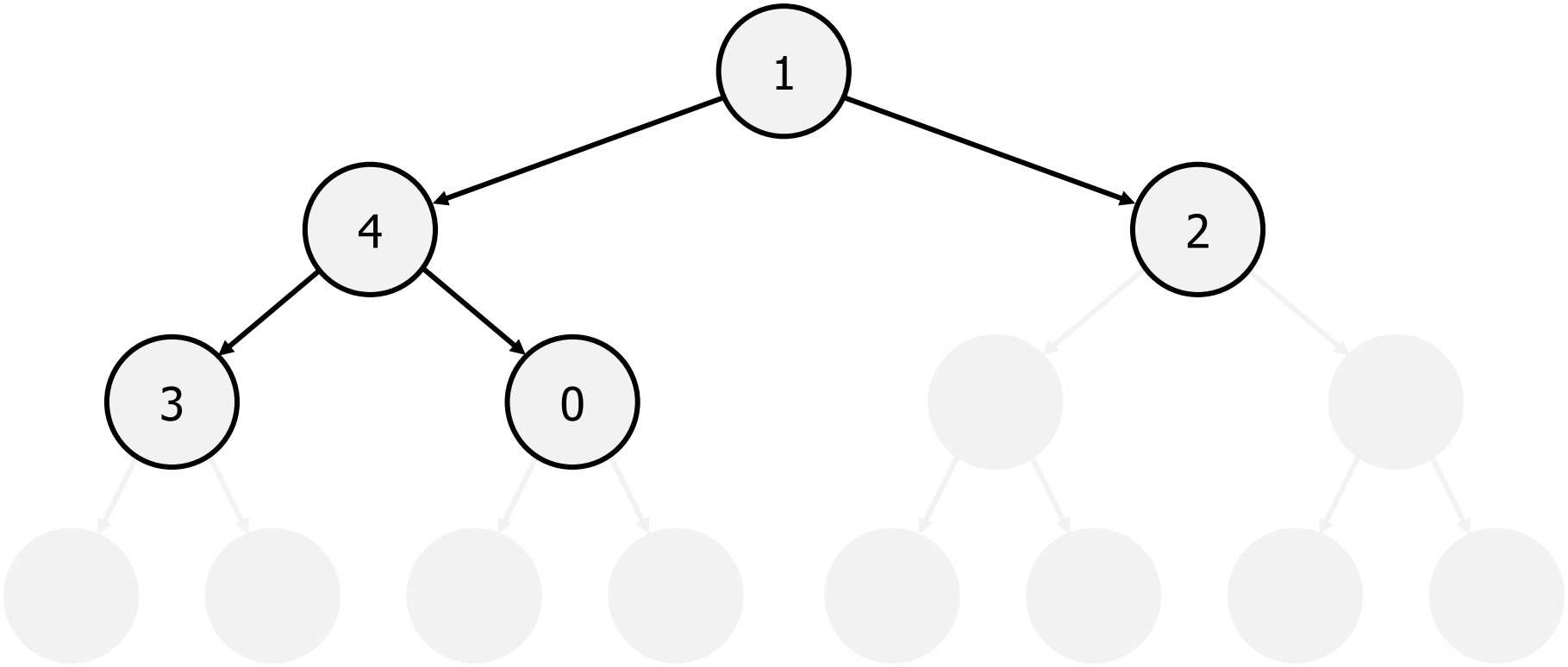
5	0	2	3	4	1	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down Complete



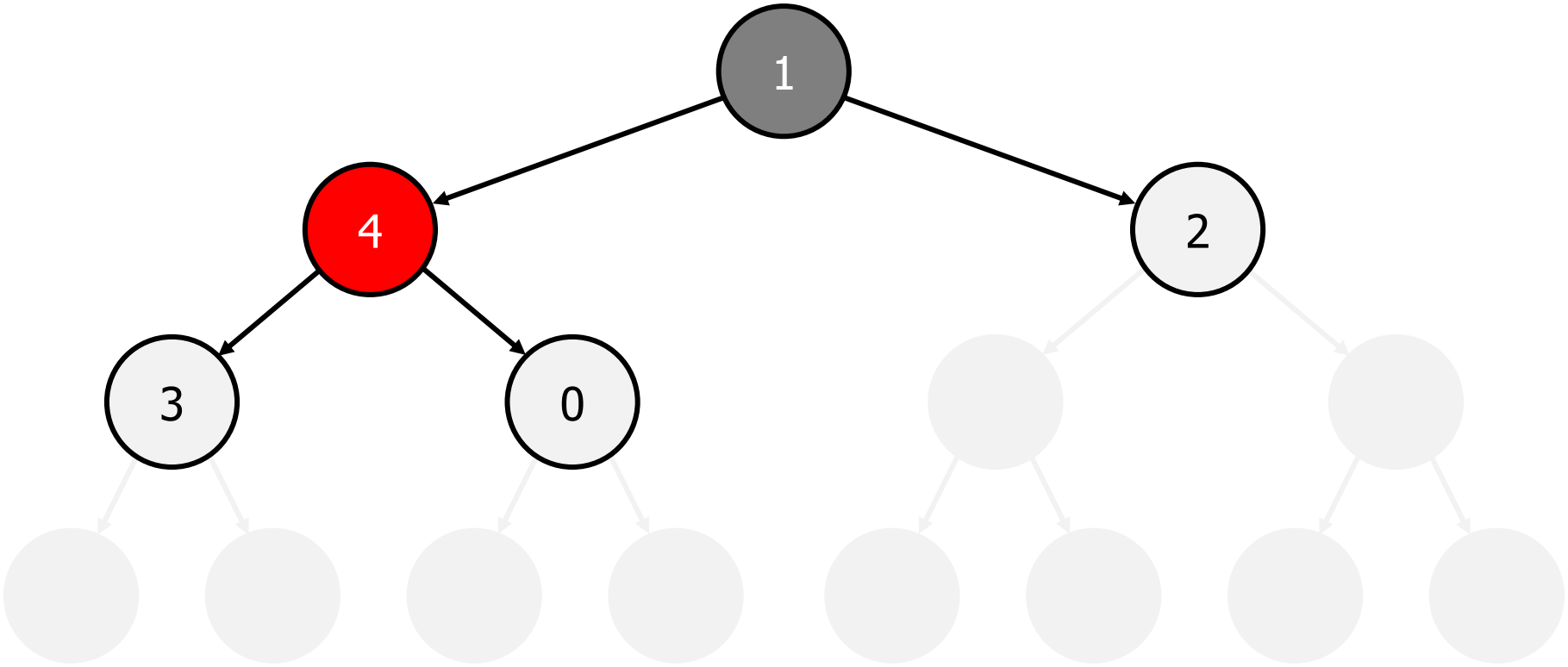
5	4	2	3	0	1	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Remove Largest; Replace With Last Element



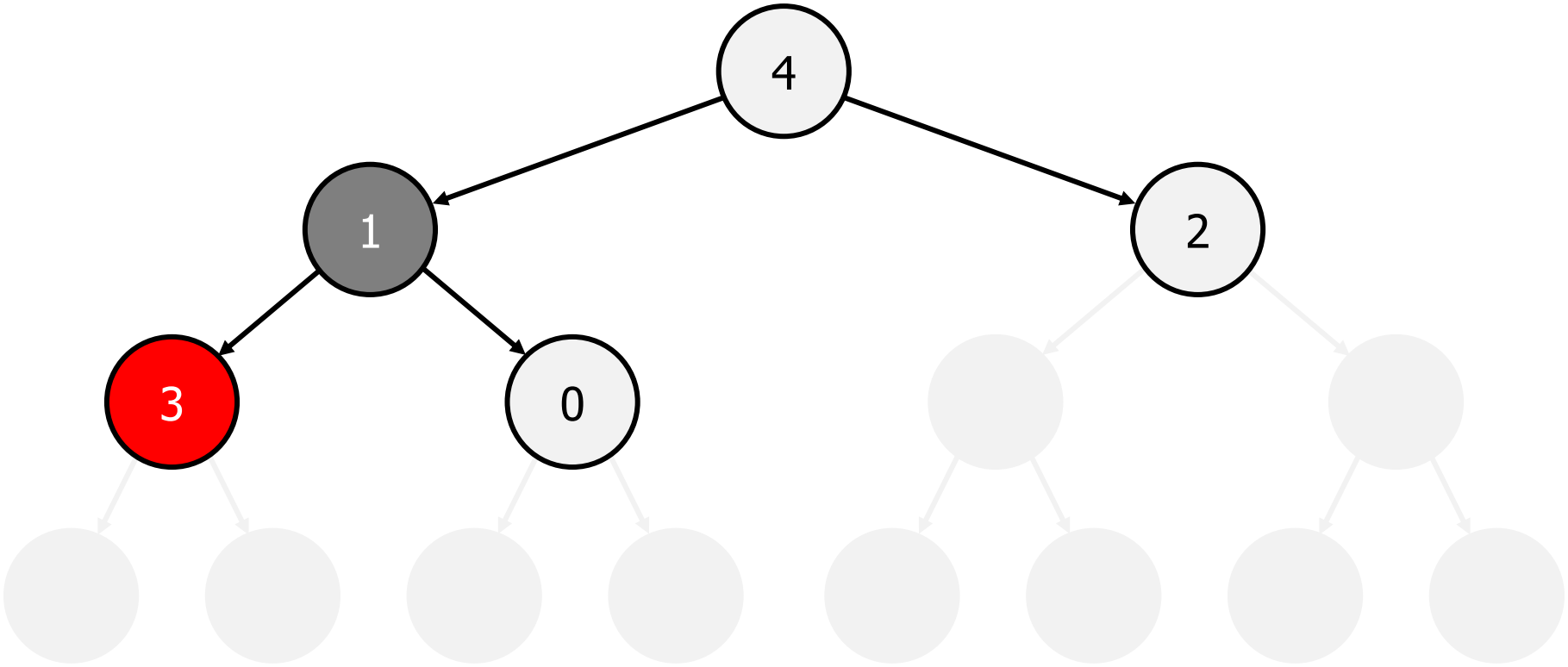
1	4	2	3	0	5	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



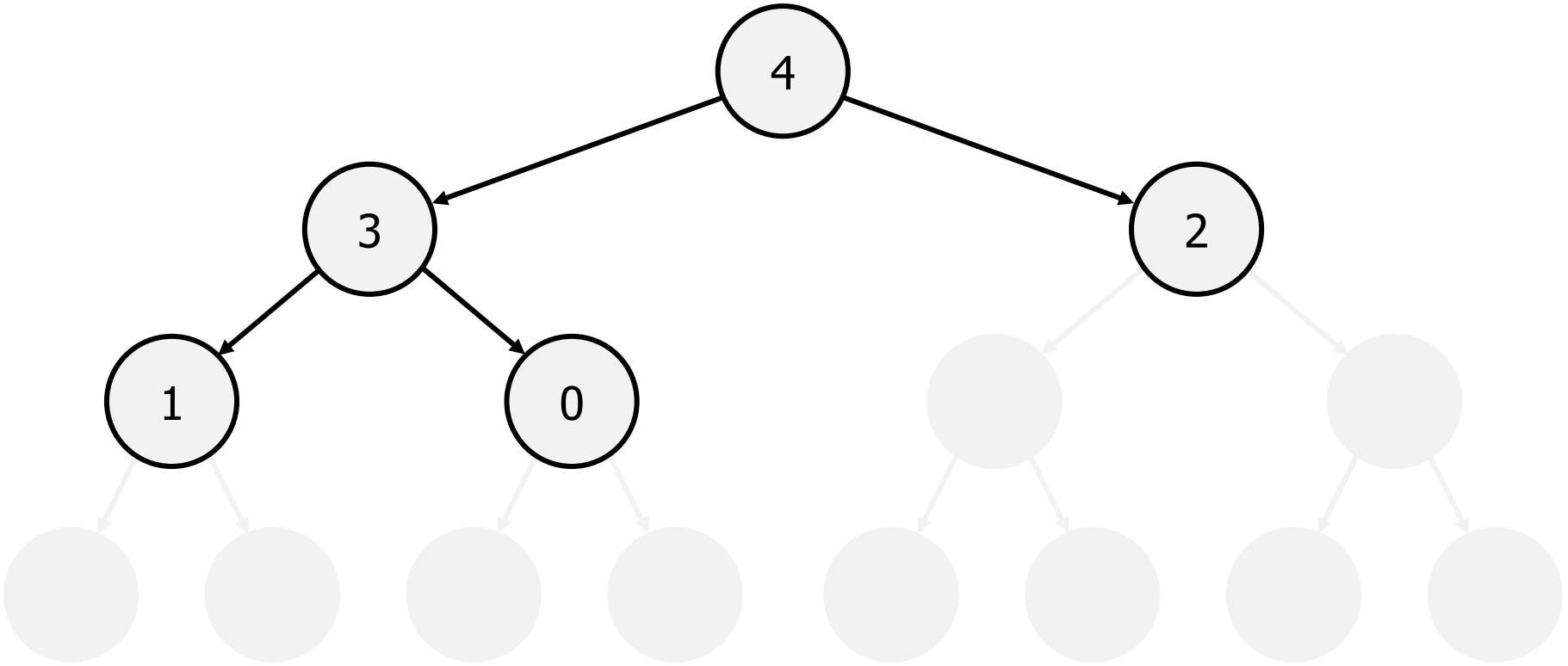
1	4	2	3	0	5	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



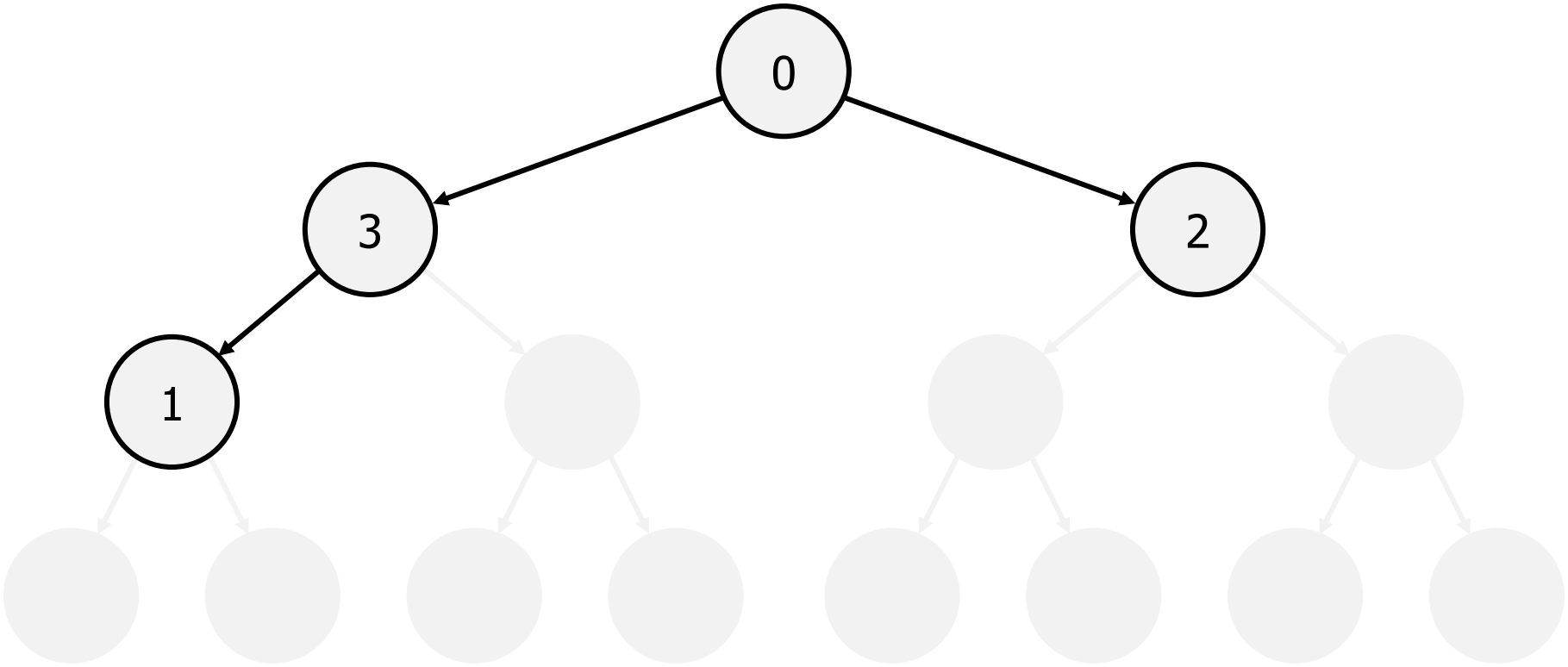
4	1	2	3	0	5	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down Complete



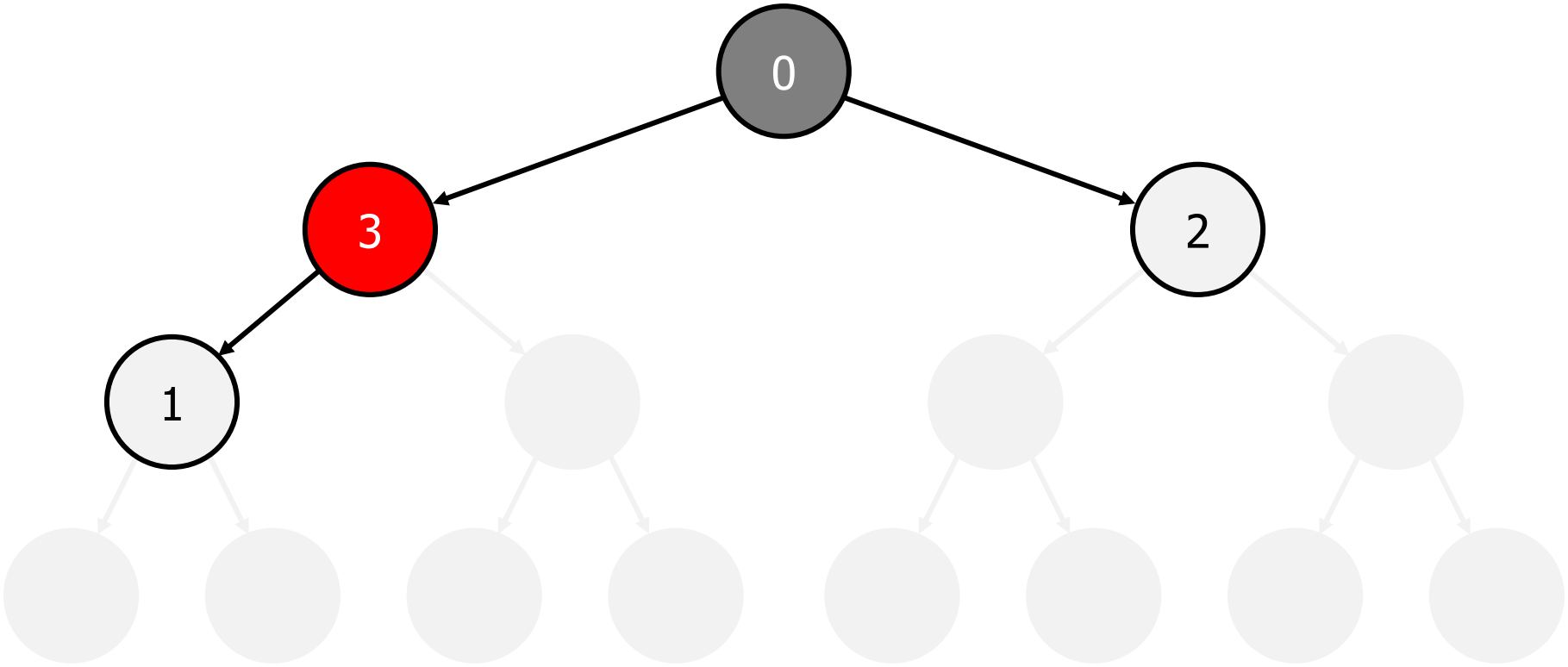
4	3	2	1	0	5	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Remove Largest; Replace With Last Element



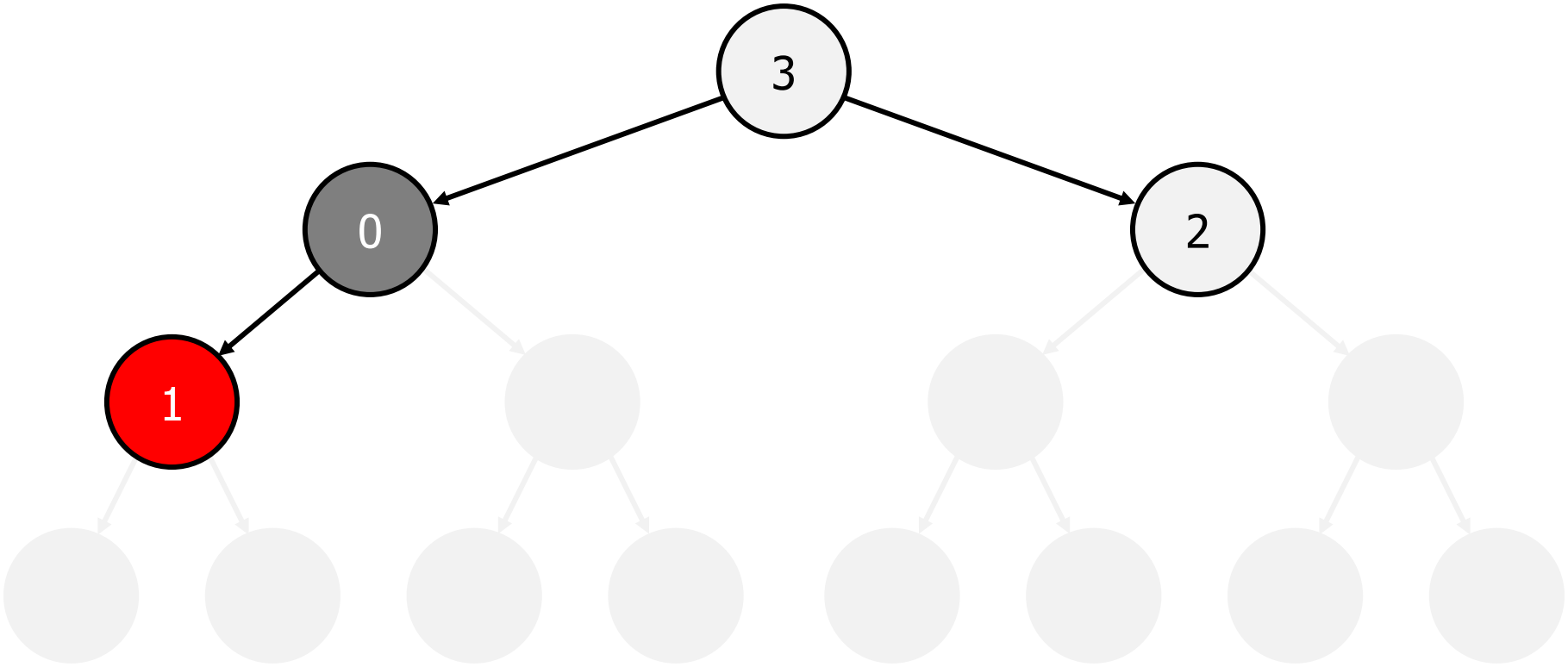
0	3	2	1	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



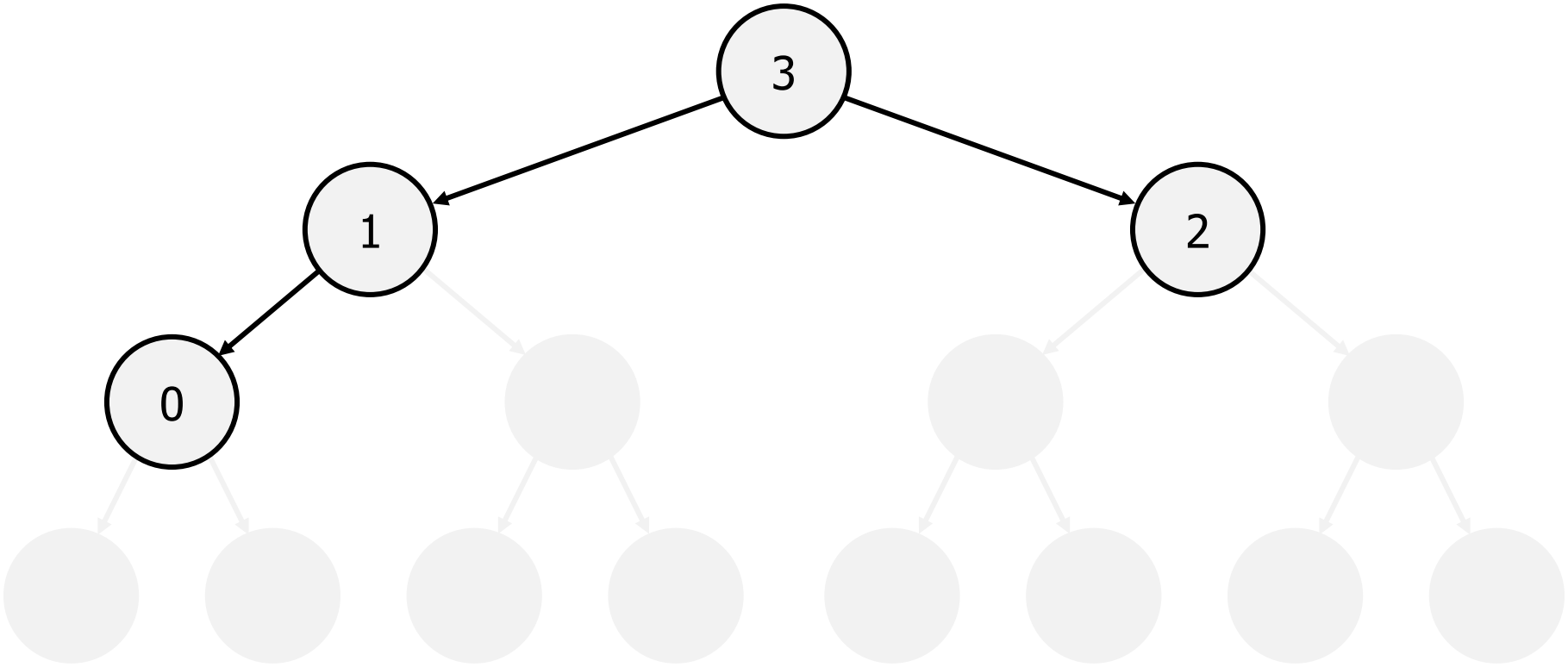
0	3	2	1	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



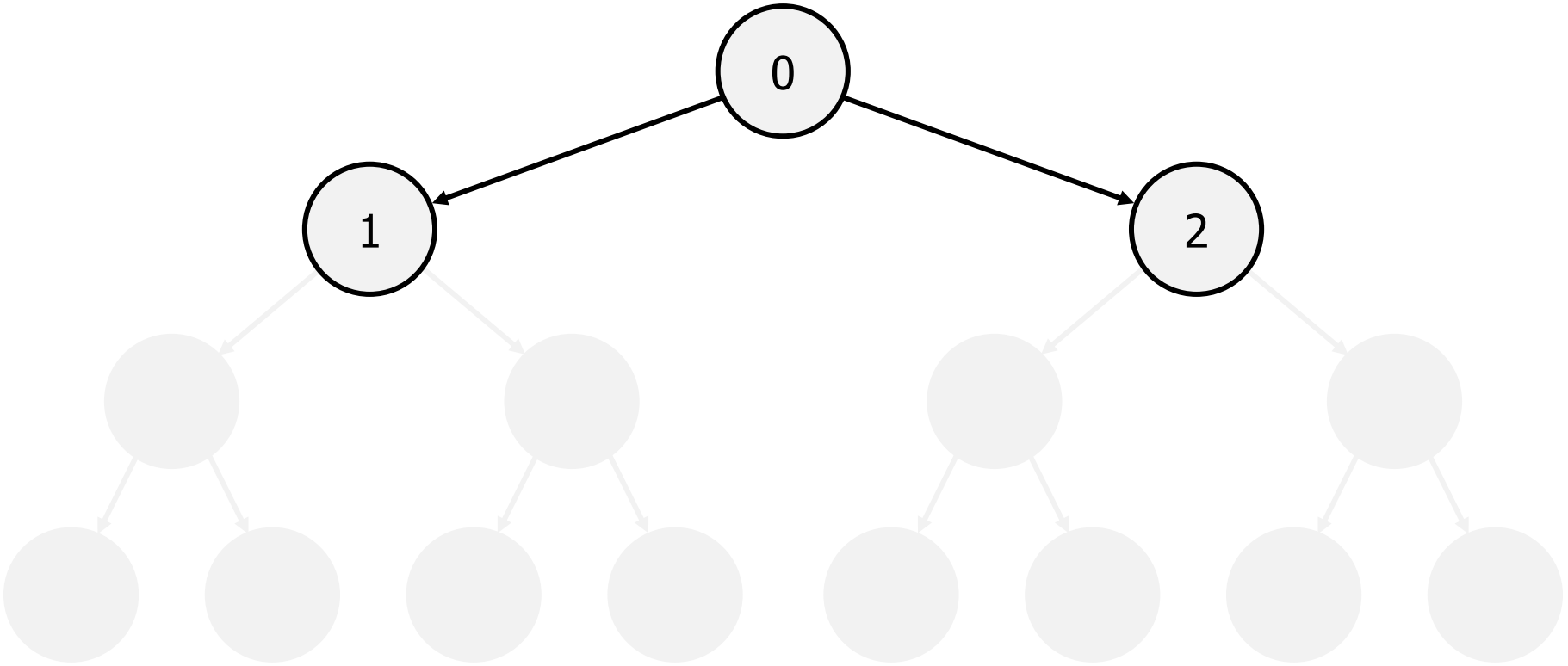
3	0	2	1	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down Complete



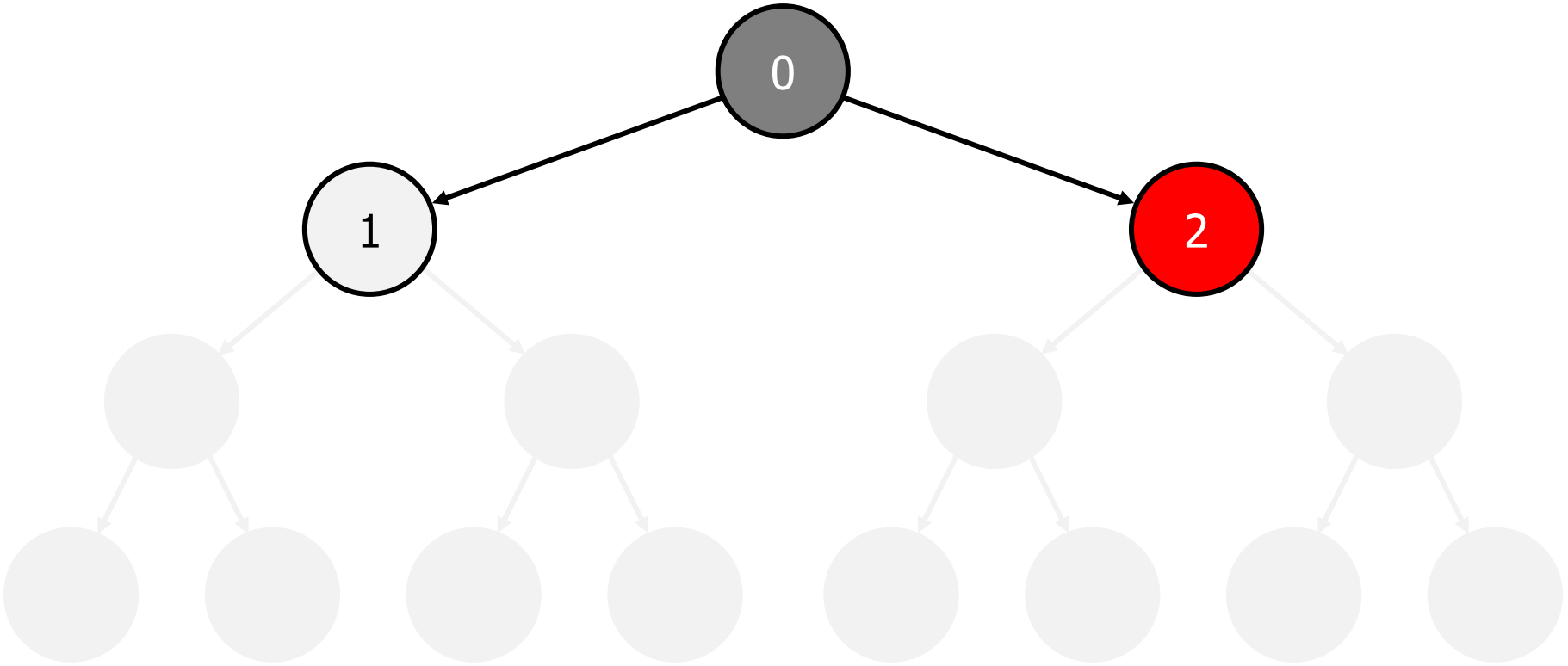
3	1	2	0	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Remove Largest; Replace With Last Element



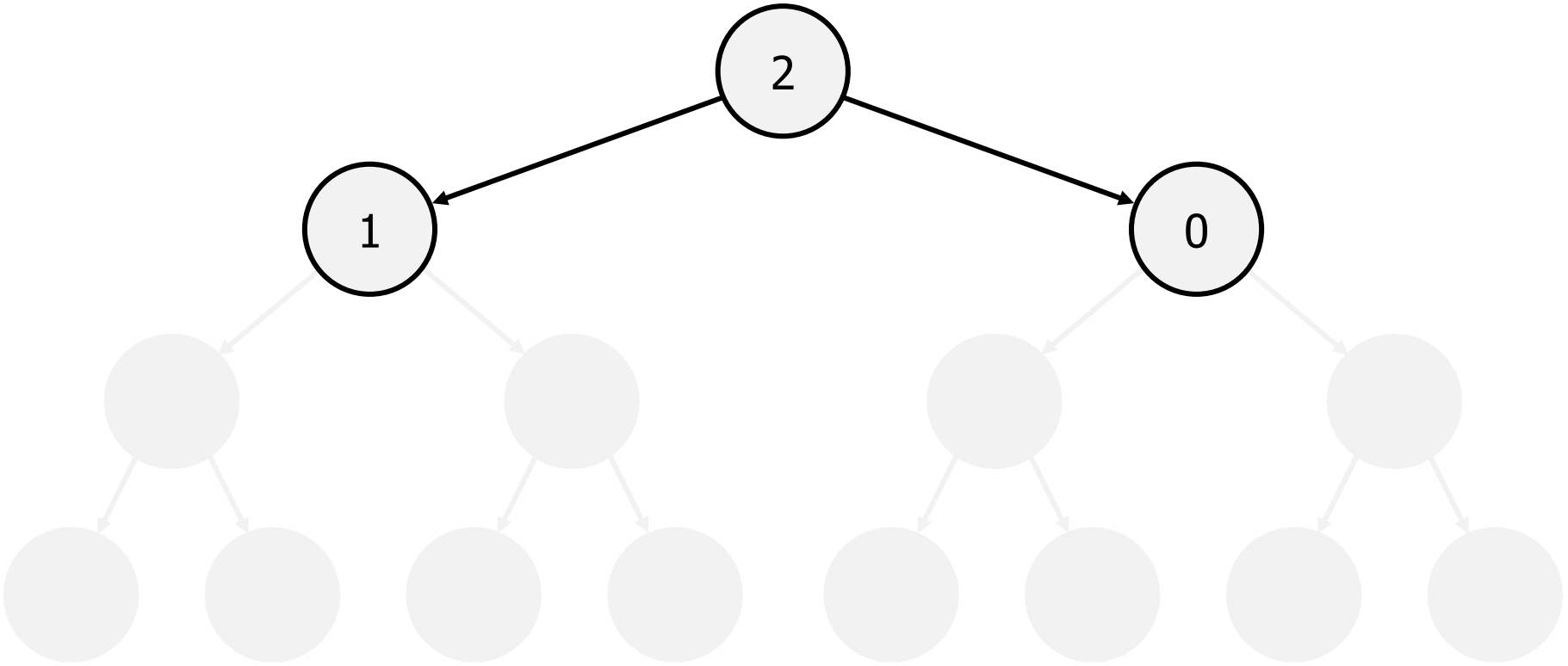
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



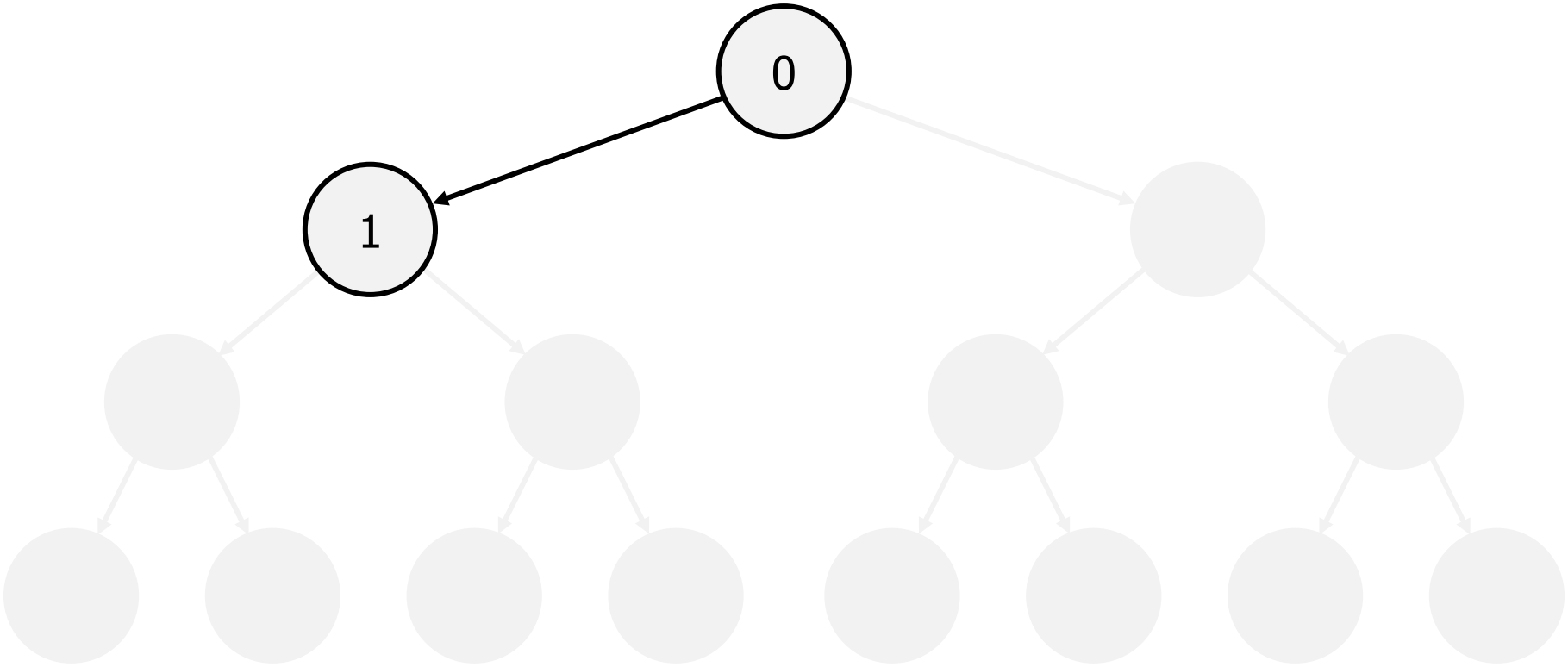
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down Complete



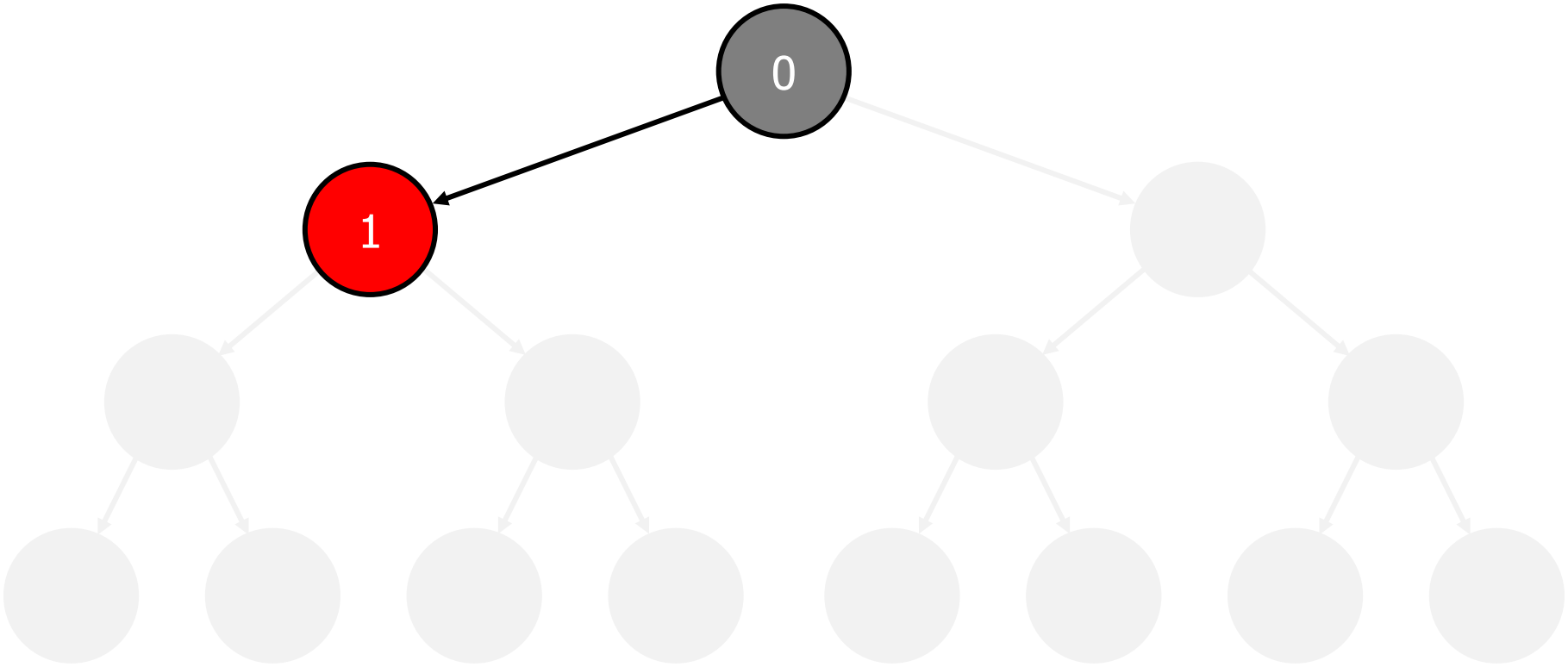
2	1	0	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Remove Largest; Replace With Last Element



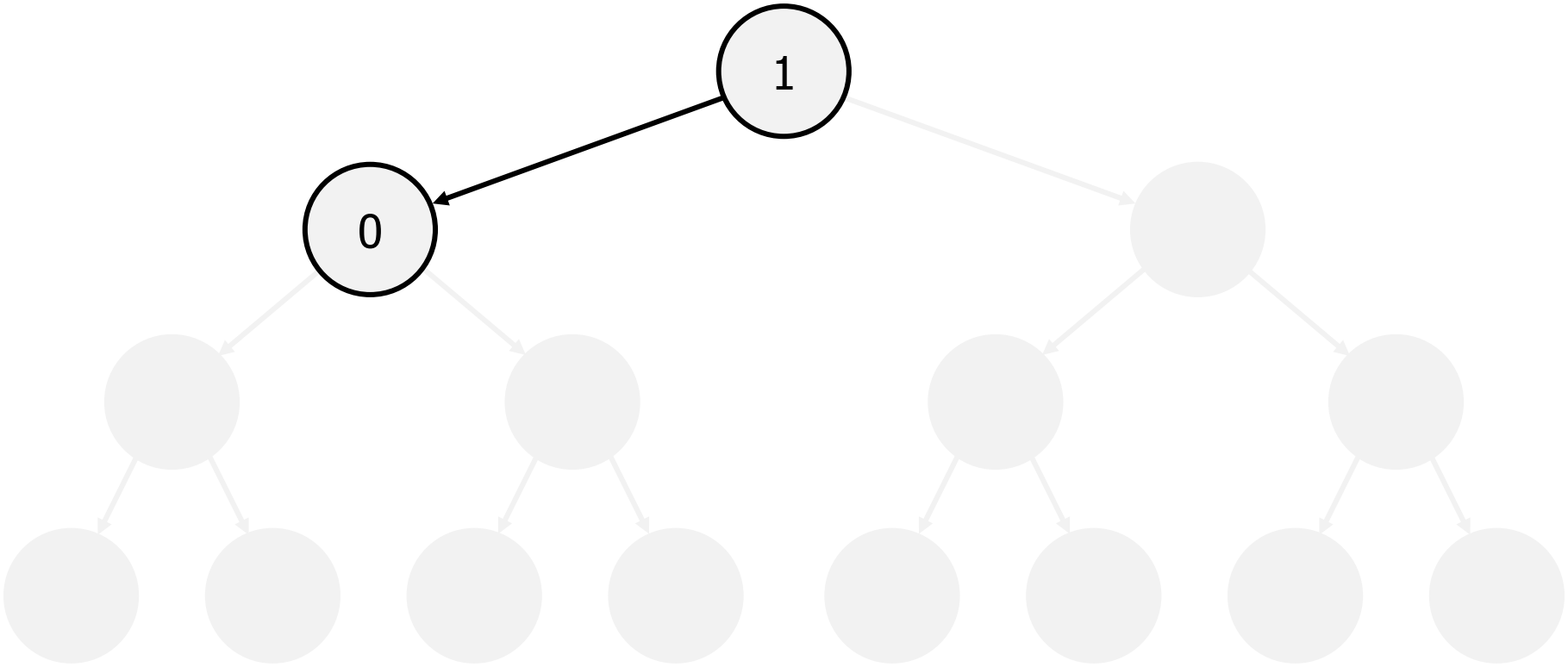
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down



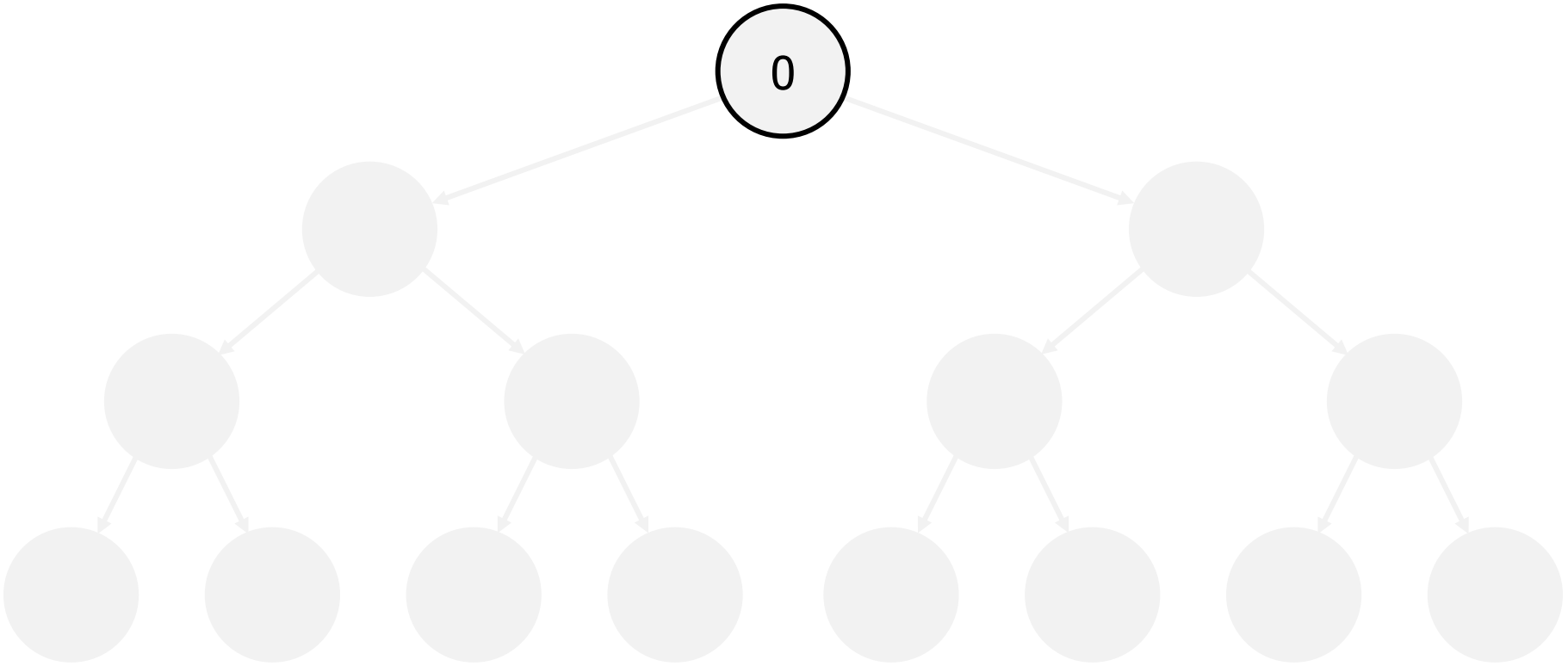
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sift Down Complete



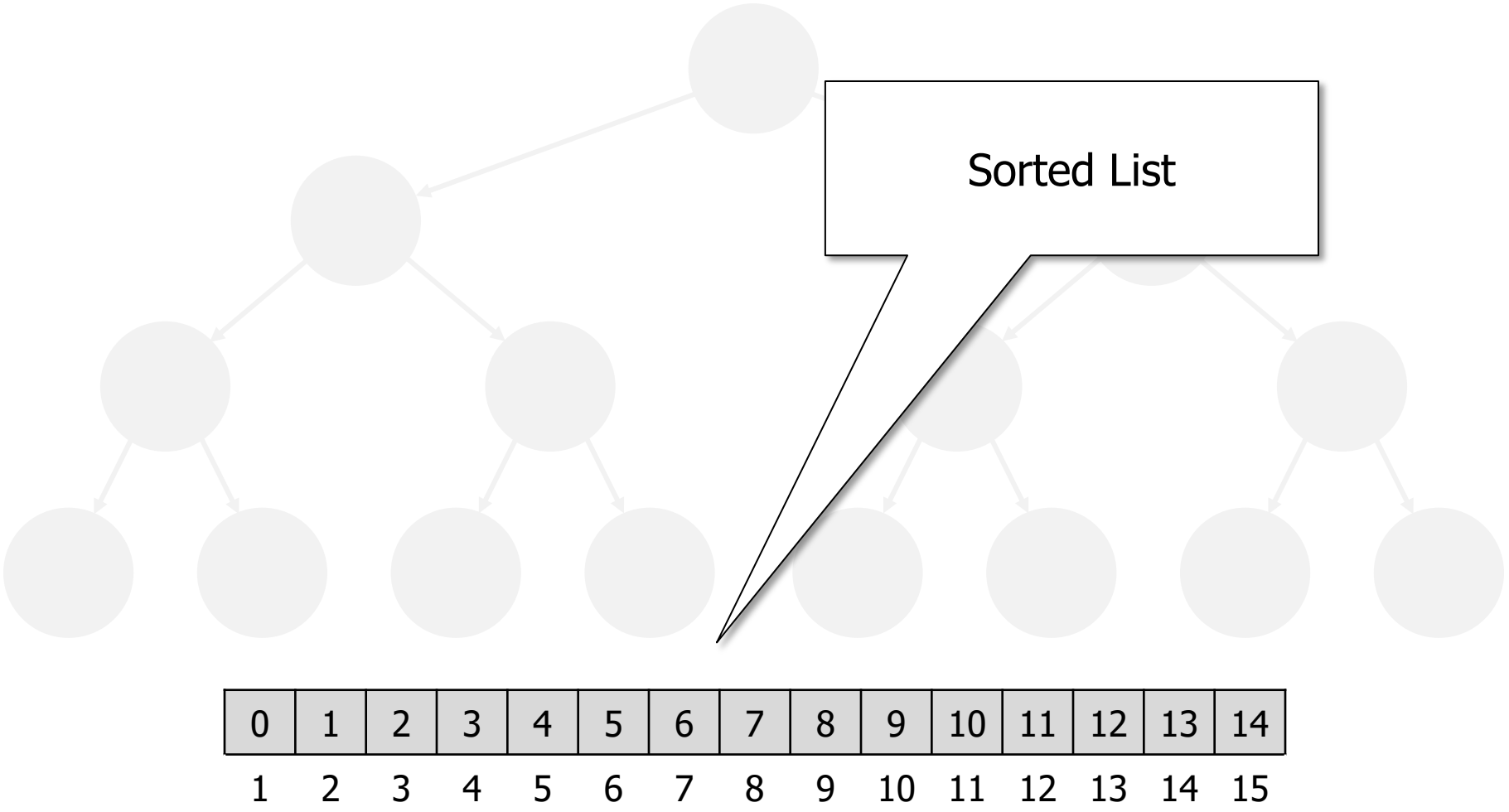
1	0	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Nothing to Sift



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Remove Largest; Heap Sort Complete



Heap Sort Iterative Pseudocode 1/2

```
HEAP_SORT(A, count)                    //A is the list; count is the length of the list
    HEAPIFY(A, count)
    end ← length(A) - 1
    WHILE end > 0                      //We know the largest is at A[0]
        swap(A[end], A[0])
        end ← end – 1                  //End is in the right place, so move down 1.
        SIFT_DOWN(A, 0, end)           //Fix the heap

HEAPIFY(A, count)
    start ← floor((count - 2) / 2)     //Start from the last parent
    WHILE start >= 0 DO
        SIFT_DOWN(A, start, count-1)   //Sift the value at start down
        start ← start – 1              //Go to the next parent node.
```

Heap Sort Iterative Pseudocode 2/2

```
SIFT_DOWN(A, start, end)
    root ← start
    WHILE root * 2 + 1 <= end                          //if there are children
        swap ← root                                    //keep track of largest
        child ← root * 2 + 1                           //is left child larger?
        IF A[swap] < A[child]
            swap ← child
        IF child+1 <= end AND A[swap] < A[child + 1]   //is right child larger?
            swap ← child + 1
        IF swap != root
            swap(A[root], A[swap])
            root ← swap
        ELSE
            RETURN                                     //no changes, so must be done
```

Recursive Heap Sort

- It is possible to implement Heap Sort recursively
- That means that instead of 2 functions Heapify and Sift_Down, we only need a Heapify function, which implements sifting
- The Heapify function calls itself
- See if you can work this out
- The bigger question is when / whether to use recursion or not
- This is a question we will return to

Summary (1/2)

- We looked at 3 search algorithms:
  1. Linear search
  2. Binary Search
  3. Interpolation search
- We saw that (2) and (3) required sorted lists; and (3) ideally needs uniform distribution
- This leads us into the issue of sorts needing particular data structures…

Summary (2/2)

- Then we looked at Heap Sort (possibly the most complex sort we've looked at)
- Heap Sort uses a heapified binary tree
- A tree implements parent-child relationships
- Heap Sort leverages those relationships to sort a list where the data structure is a heap (i.e. a heapified tree)
- But how / why is Heap Sort better or worse than other sorts?
- This will be coming when we get to complexity analysis (which we start in the next lecture)