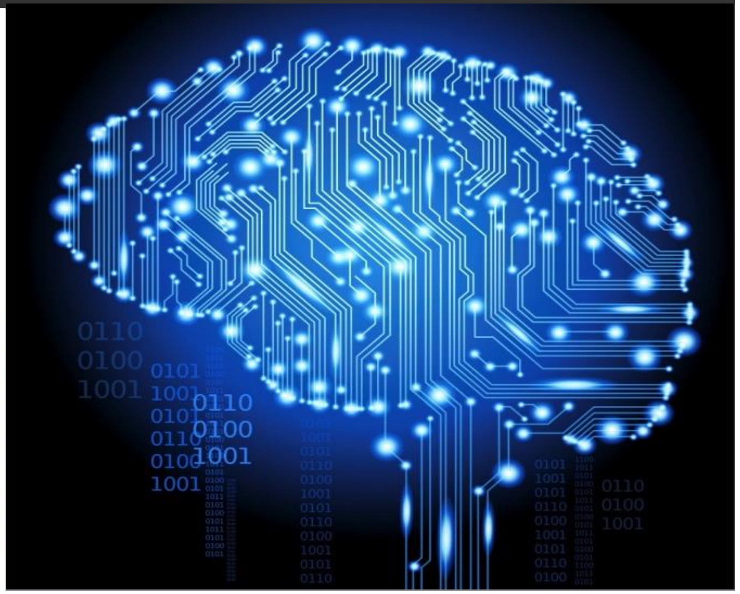
MONASH University

Information Technology

Heaps II

Prepared by Maria Garcia de la Banda
Updated by Brendon Taylor

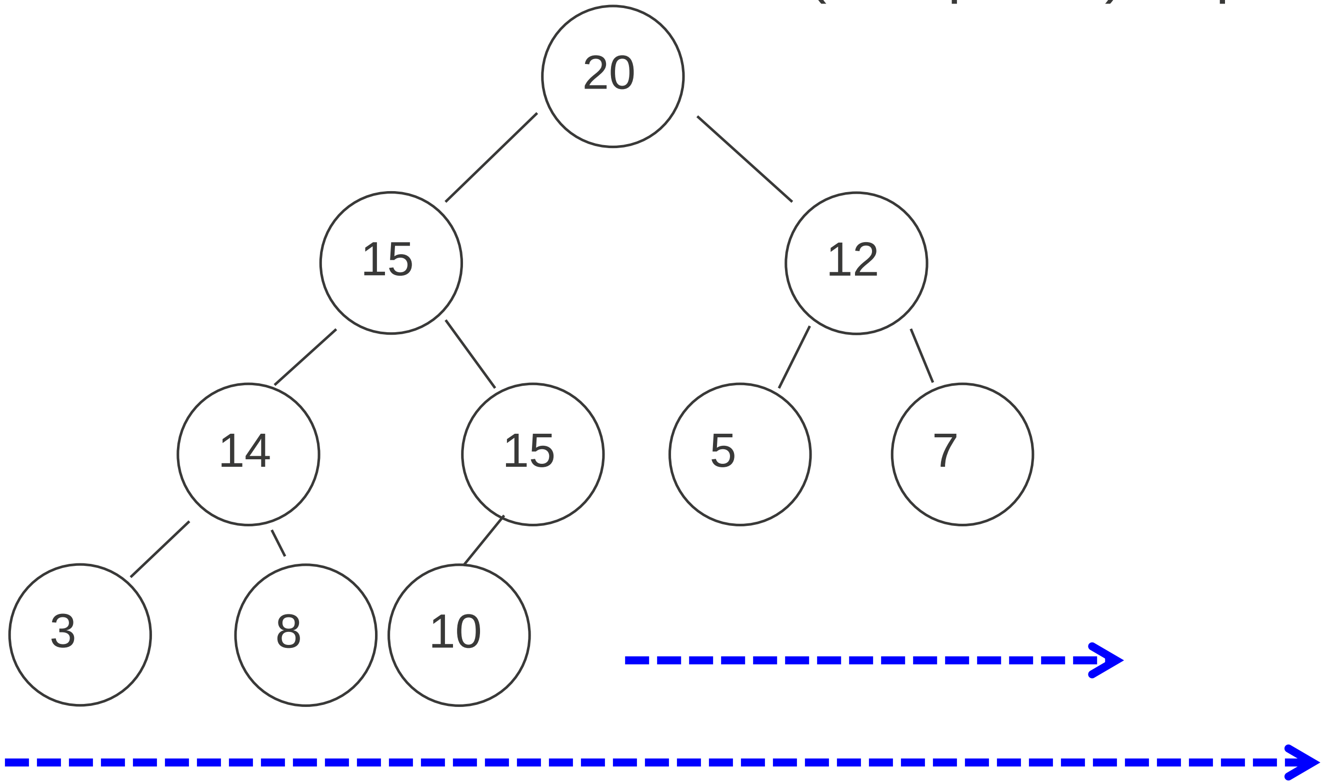


Objectives for these two lectures

- **To understand a simple implementation of Heaps**
  - To be able to reason about the complexity of its operations
  - To be able to implement them and modify them
- **To consider a particular application:**
  - Sorting a priority queue
- **To understand a “cleverer” construction method for Heaps**
  - Bottom-up heap construction
- **To understand heap-sort**
  - A fast, recursive alternative to mergesort and quicksort

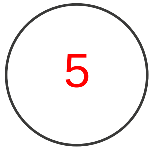
Recall: Basics of heaps

- **A heap is a binary tree that is :**
  - Complete: All levels but the last are full; the last is filled from the left
  - Heap-ordered: each child is smaller than (or equal to) its parent



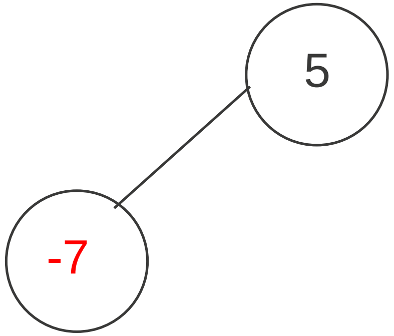
Heap vs BST

Heap vs Binary Search Tree



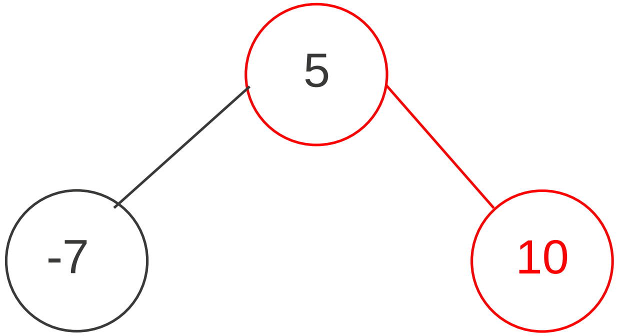
Lets insert the numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty heap, in that order

Heap vs Binary Search Tree



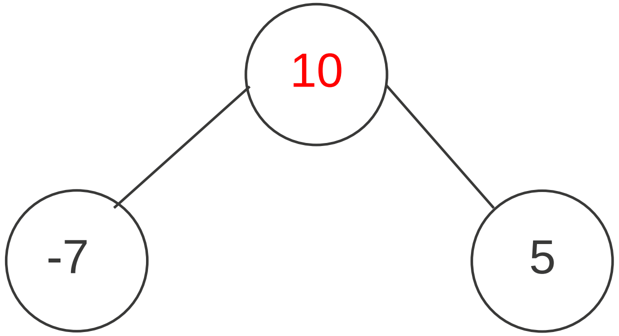
Lets insert the numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty heap, in that order

Heap vs Binary Search Tree



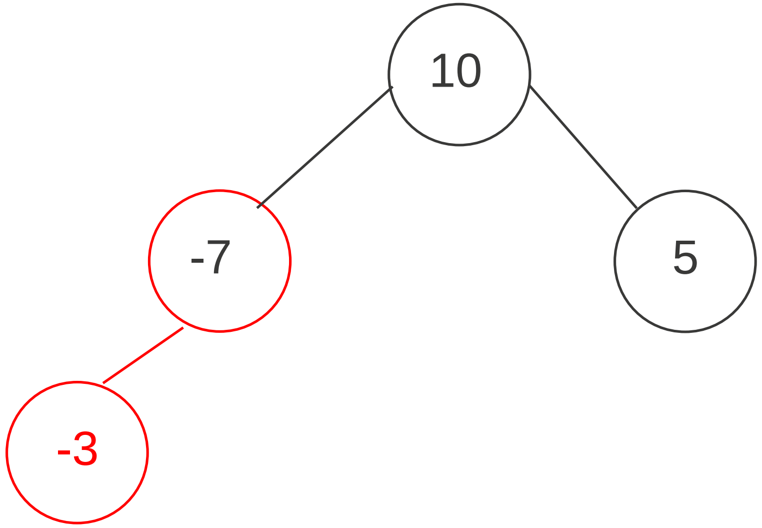
Lets insert the numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty heap, in that order

Heap vs Binary Search Tree



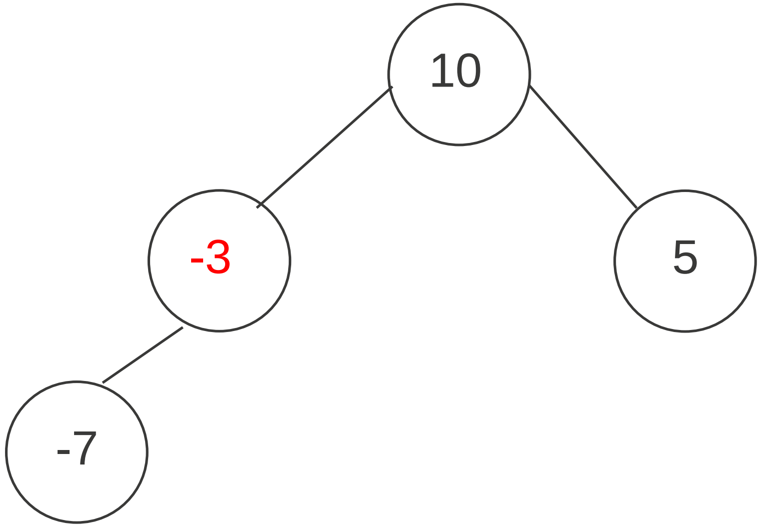
Lets insert the numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty heap, in that order

Heap vs Binary Search Tree



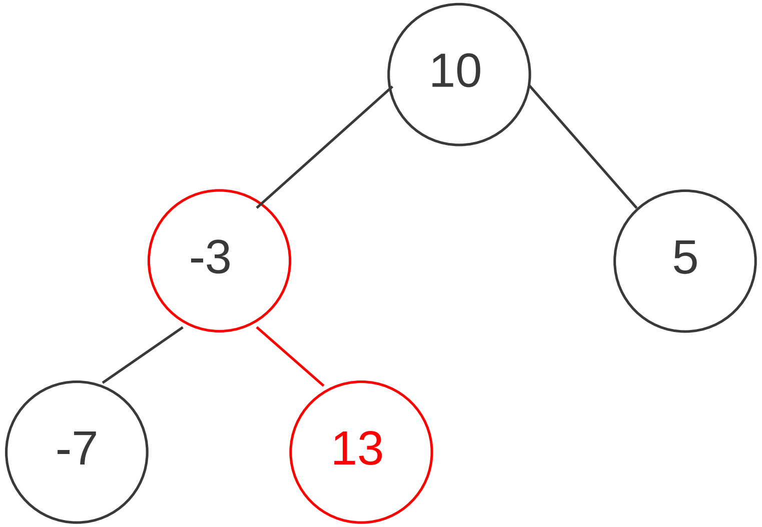
Lets insert the numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty heap, in that order

Heap vs Binary Search Tree



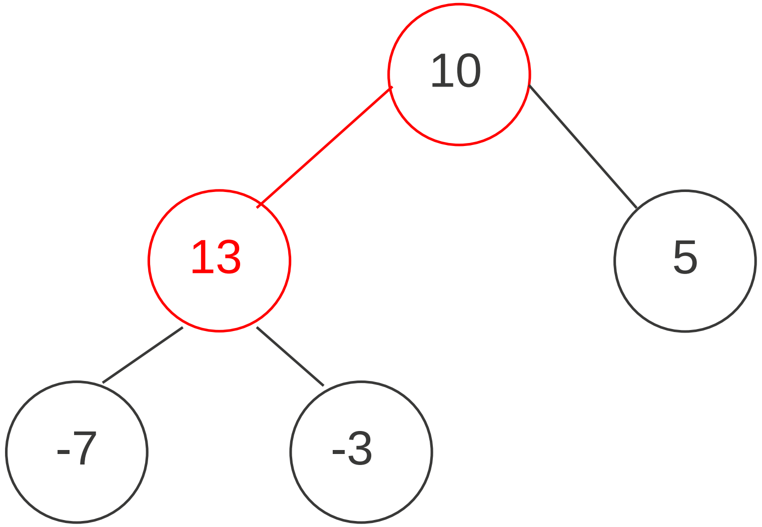
Lets insert the numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty heap, in that order

Heap vs Binary Search Tree



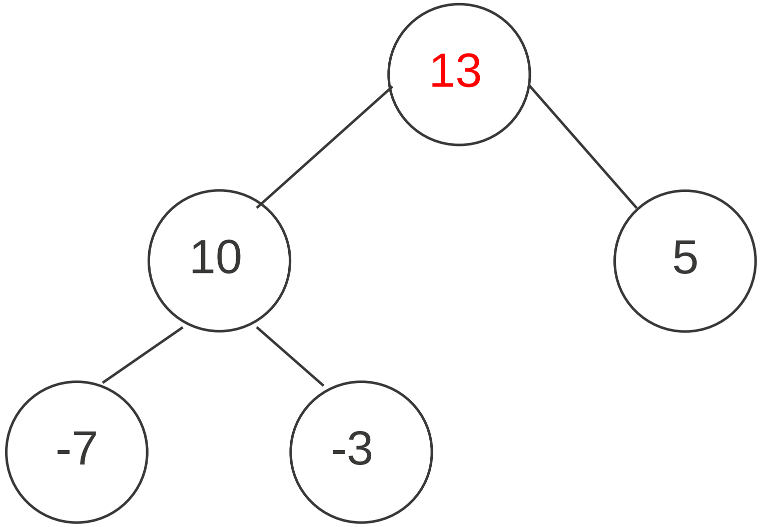
Lets insert the numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty heap, in that order

Heap vs Binary Search Tree



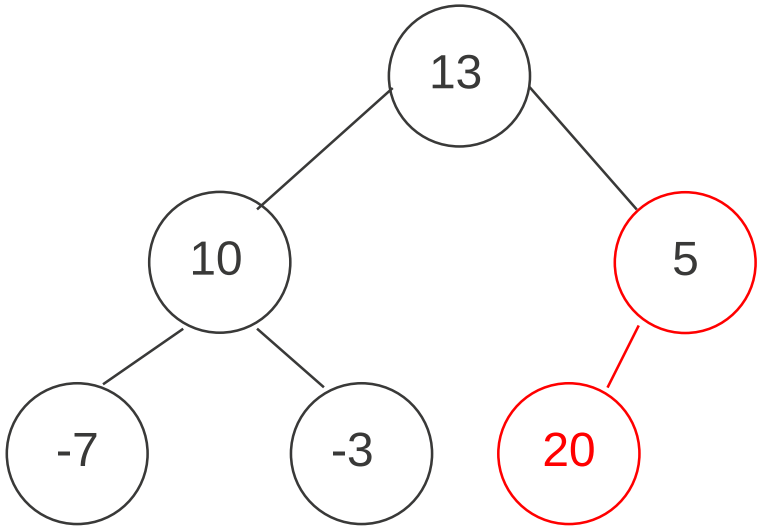
Lets insert the numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty heap, in that order

Heap vs Binary Search Tree



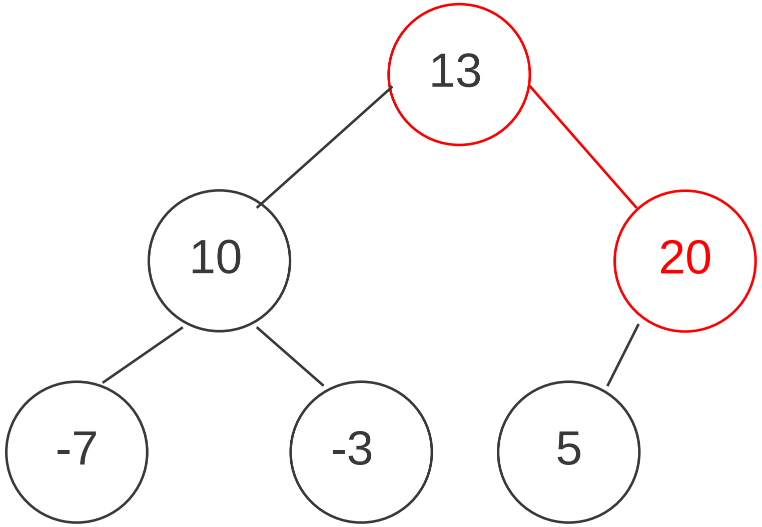
Lets insert the numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty heap, in that order

Heap vs Binary Search Tree



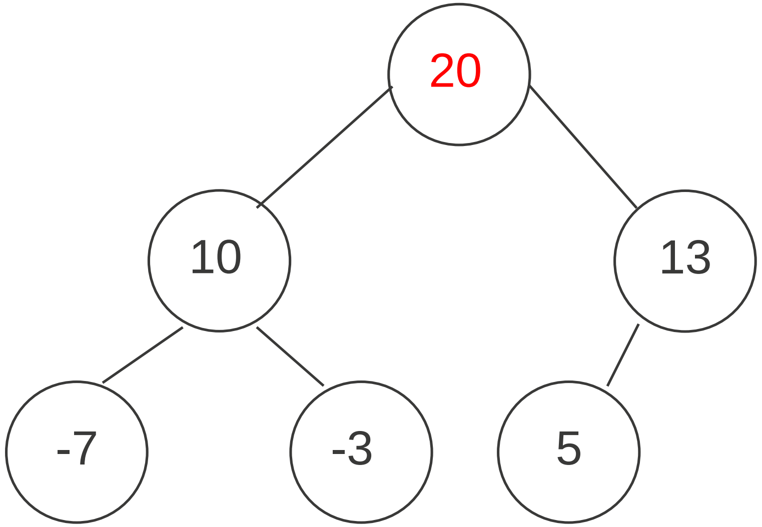
Lets insert the numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty heap, in that order

Heap vs Binary Search Tree



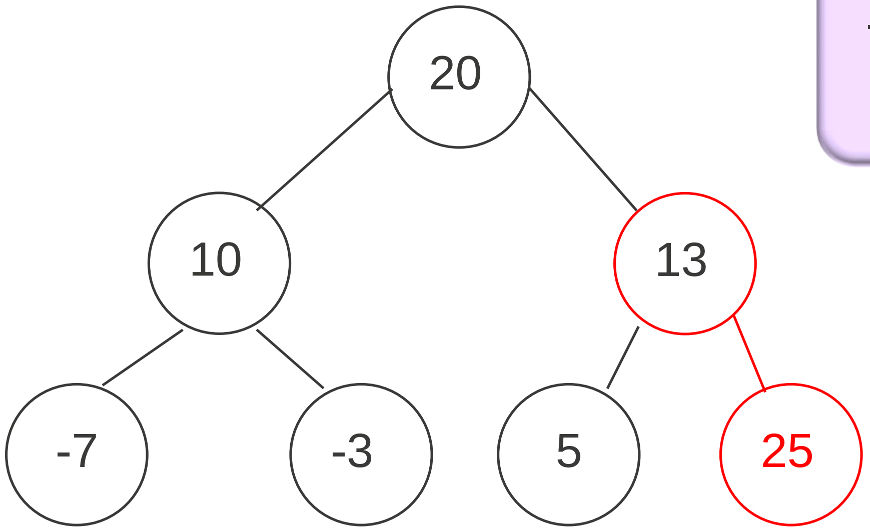
Lets insert the numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty heap, in that order

Heap vs Binary Search Tree



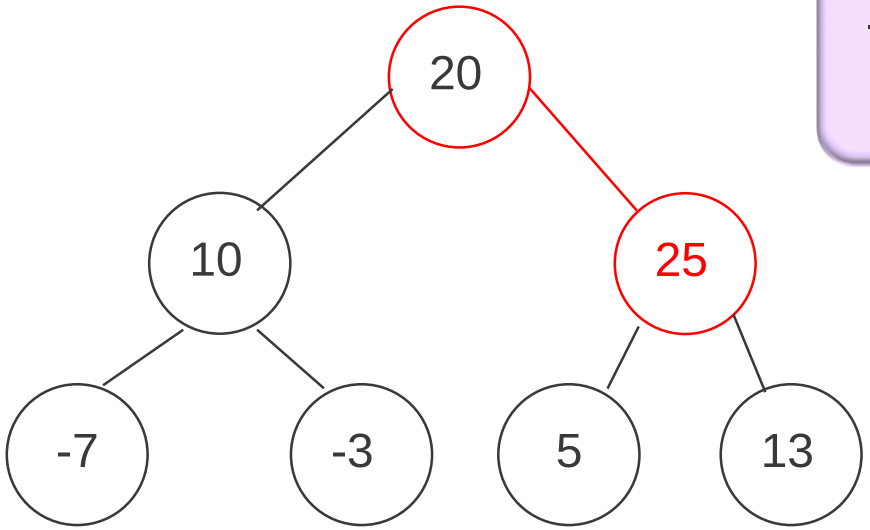
Lets insert the numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty heap, in that order

Heap vs Binary Search Tree



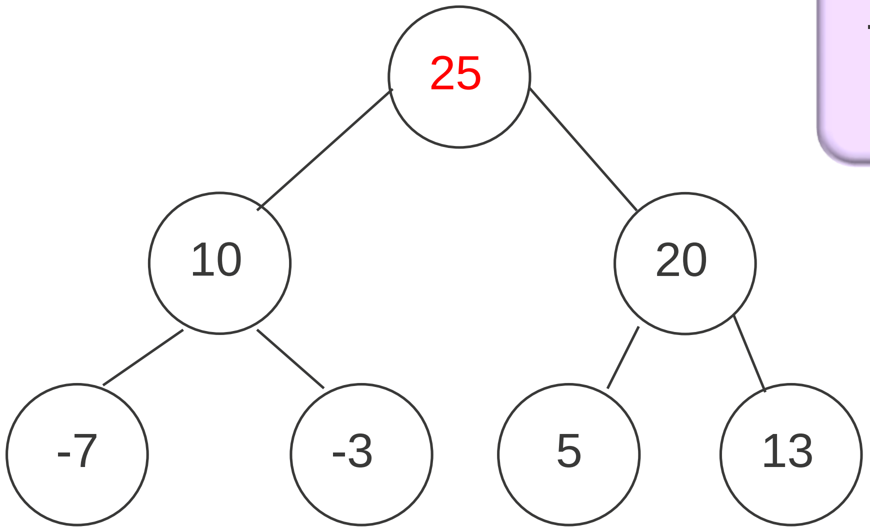
Lets insert the numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty heap, in that order

Heap vs Binary Search Tree



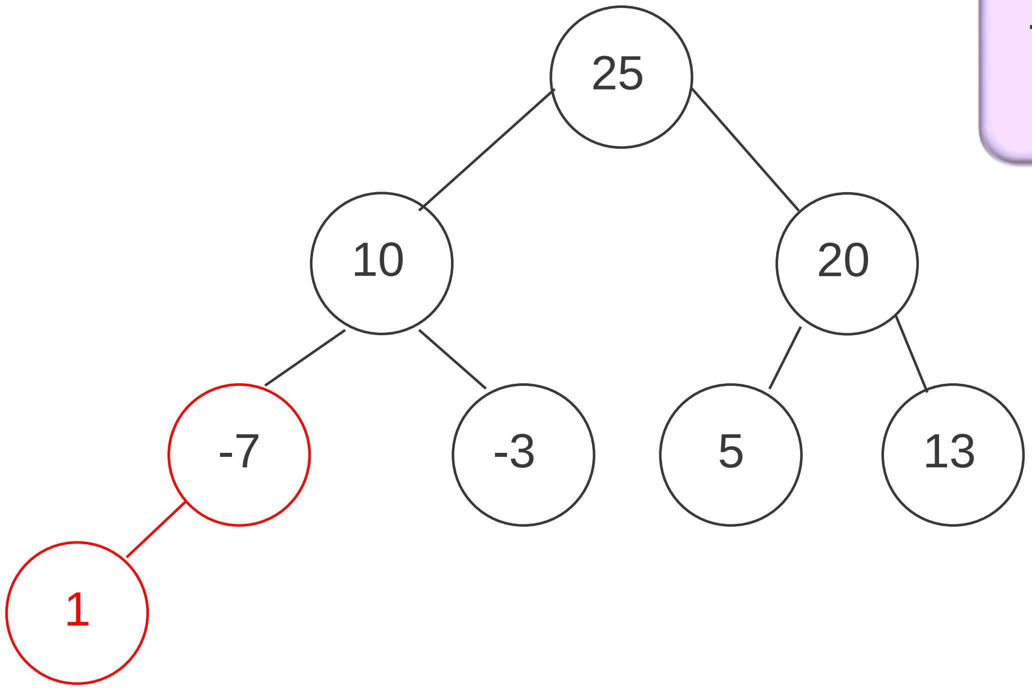
Lets insert the numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty heap, in that order

Heap vs Binary Search Tree



Lets insert the numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty heap, in that order

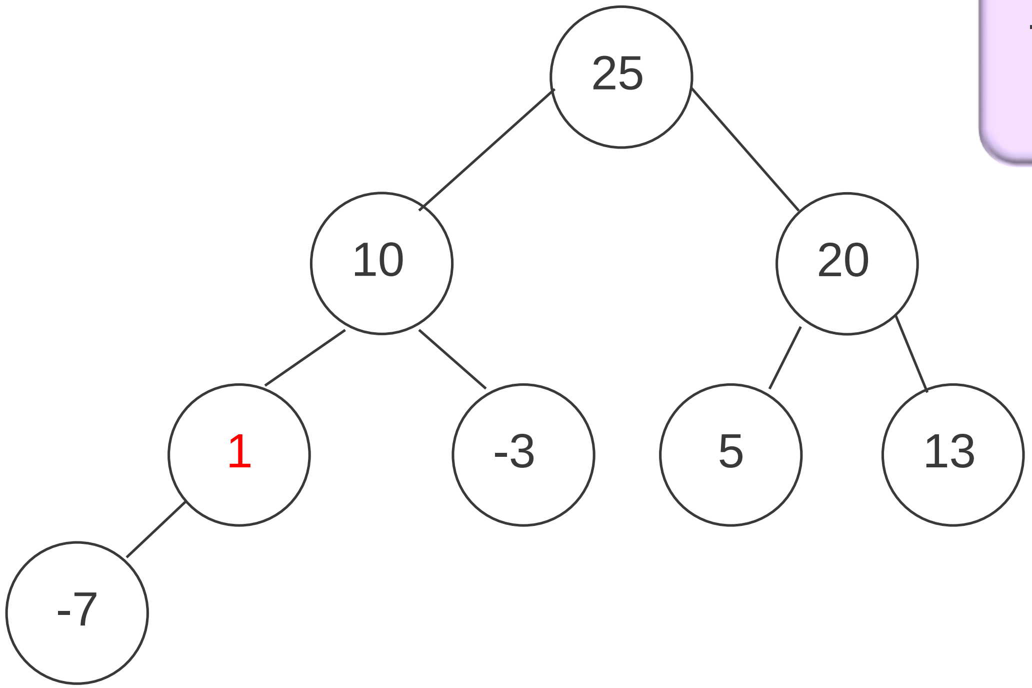
Heap vs Binary Search Tree



Lets insert the numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty heap, in that order

Heap vs Binary Search Tree

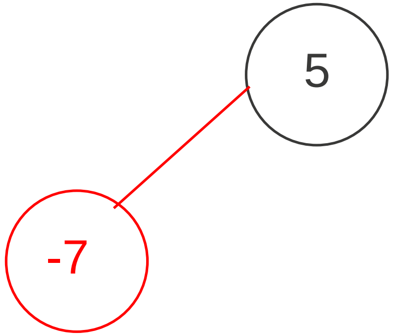
Lets insert the numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty heap, in that order



Heap vs Binary Search Tree

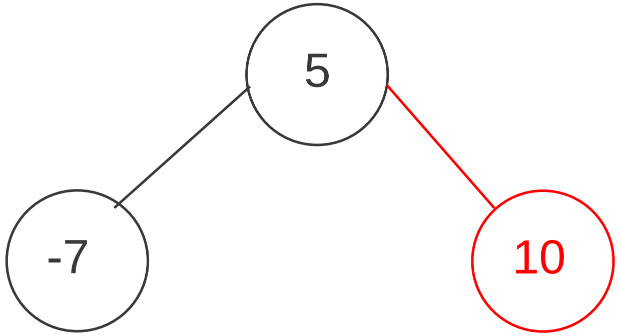
Lets insert now the same numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty binary search tree, in the same order

Heap vs Binary Search Tree



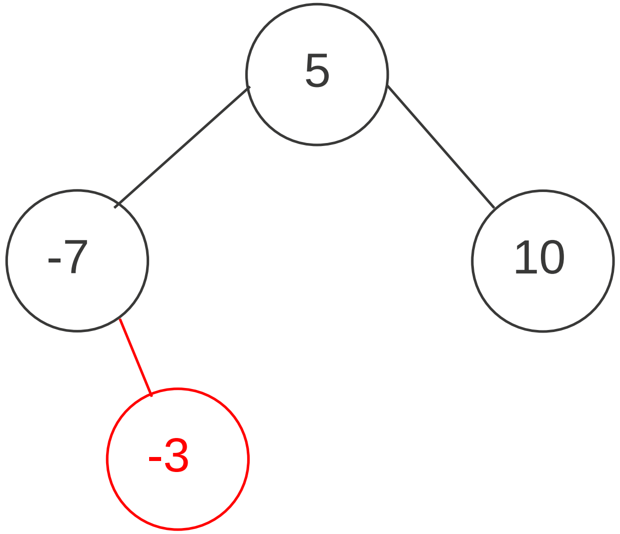
Lets insert now the same numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty binary search tree, in the same order

Heap vs Binary Search Tree



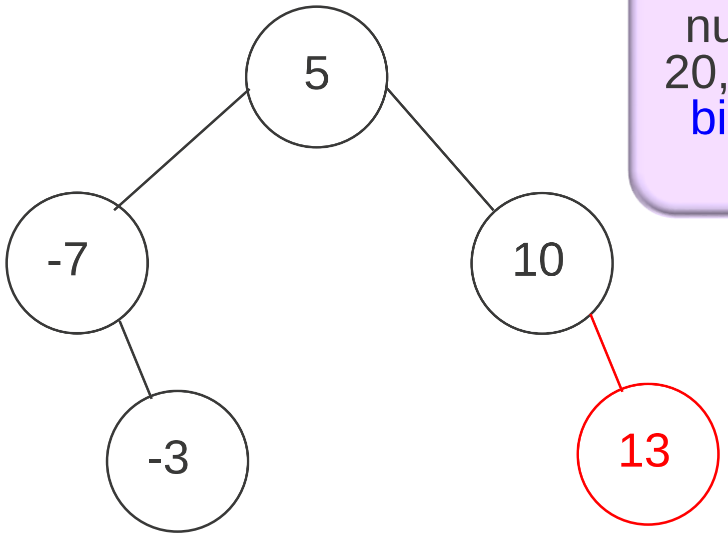
Lets insert now the same numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty binary search tree, in the same order

Heap vs Binary Search Tree



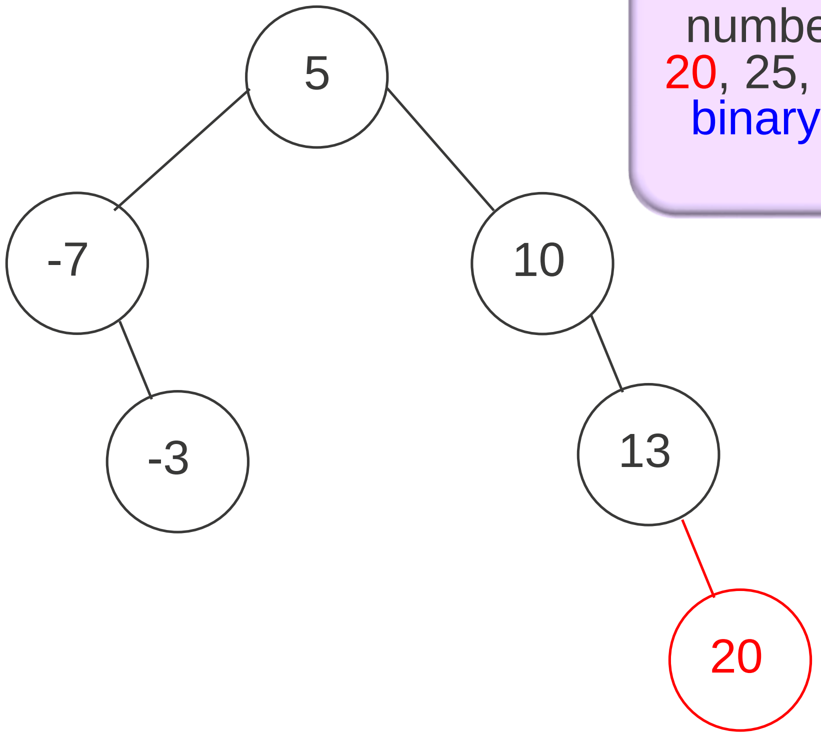
Lets insert now the same numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty binary search tree, in the same order

Heap vs Binary Search Tree



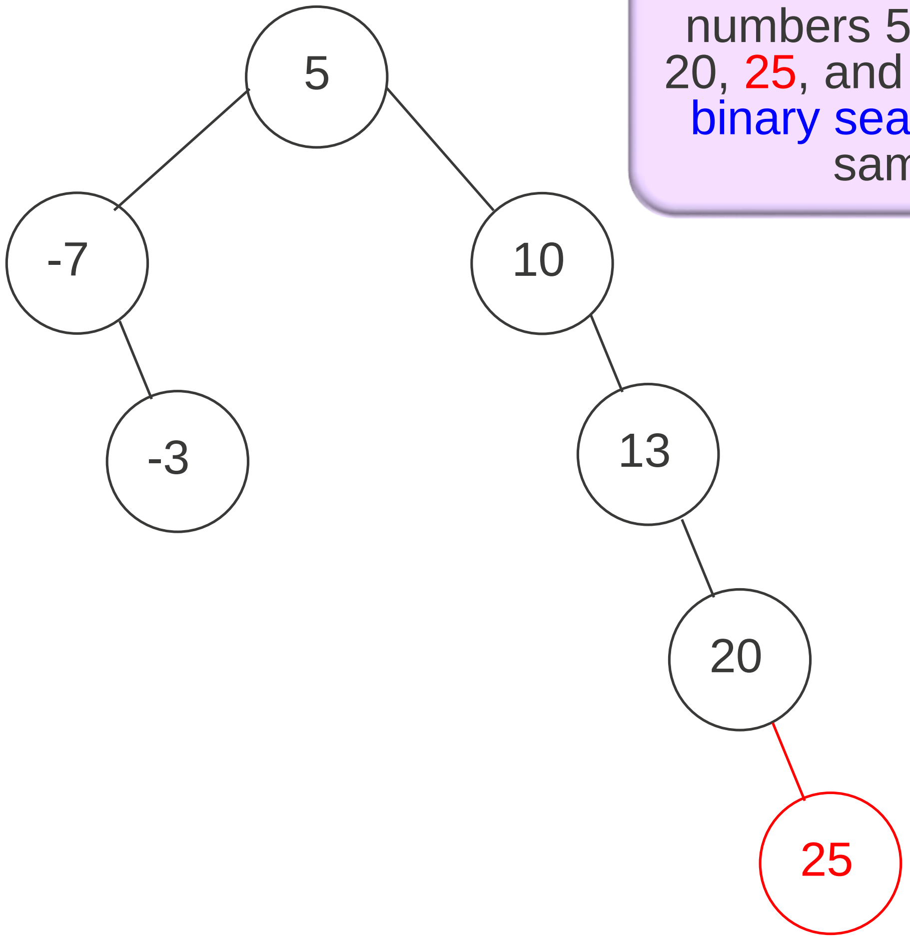
Lets insert now the same numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty binary search tree, in the same order

Heap vs Binary Search Tree



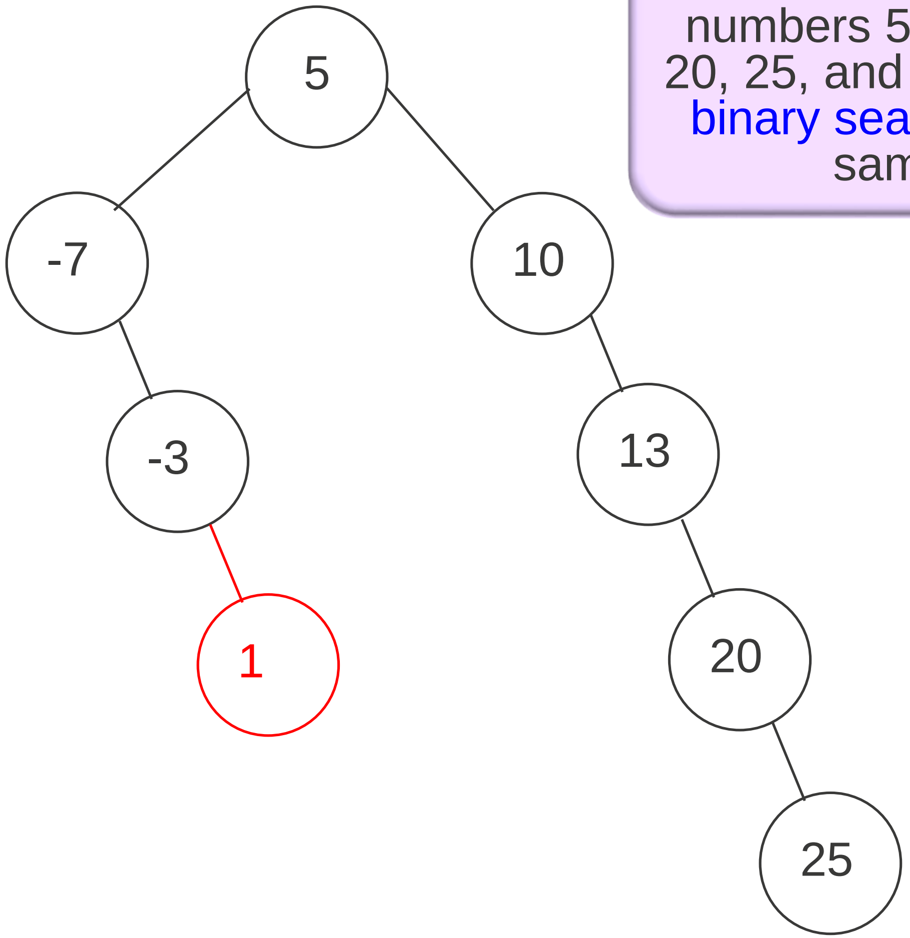
Lets insert now the same numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty binary search tree, in the same order

Heap vs Binary Search Tree



Lets insert now the same numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty binary search tree, in the same order

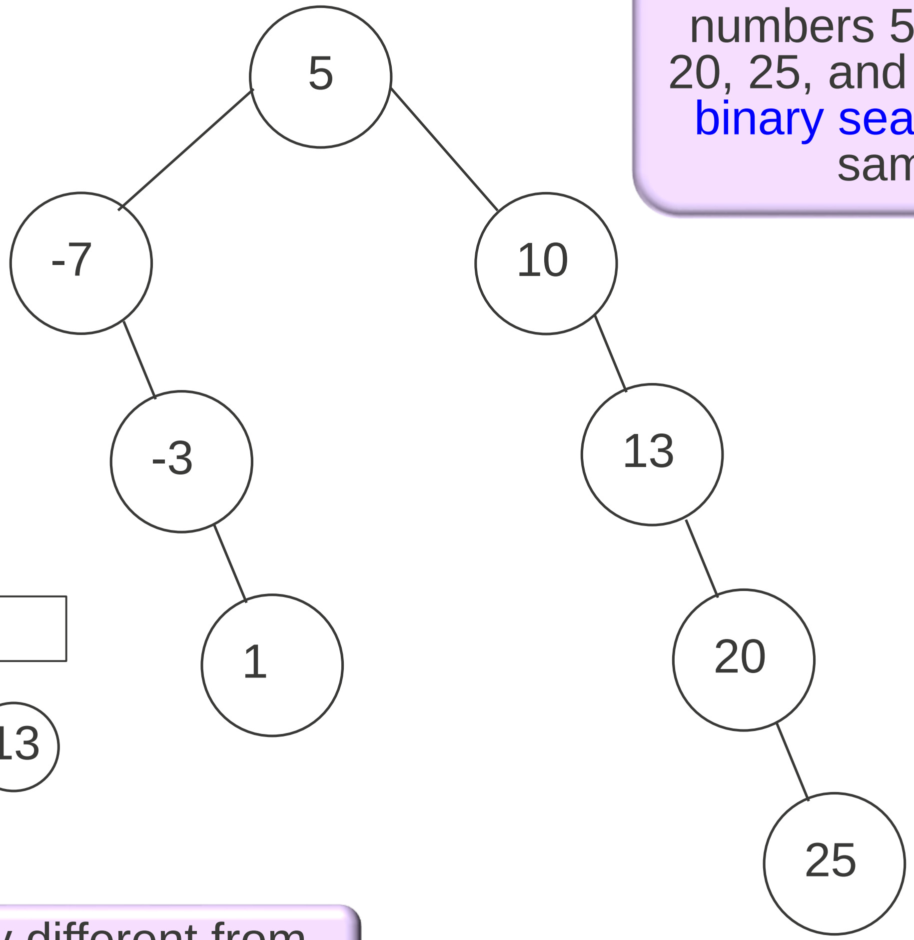
Heap vs Binary Search Tree

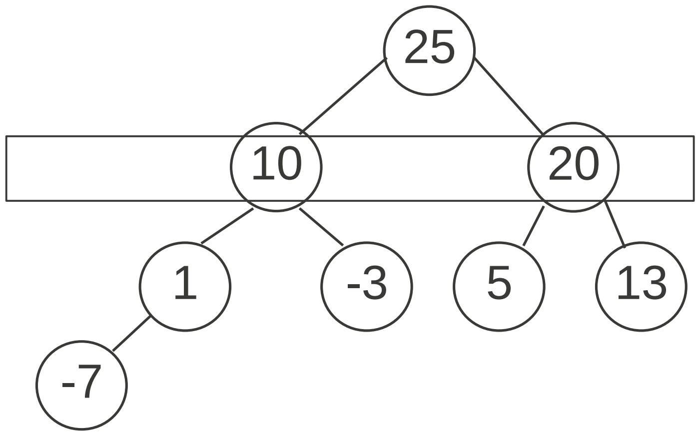


Lets insert now the same numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty binary search tree, in the same order

Heap vs Binary Search Tree

Lets insert now the same numbers 5, -7, 10, -3, 13, 20, 25, and 1 into an empty binary search tree, in the same order





Very different from the heap

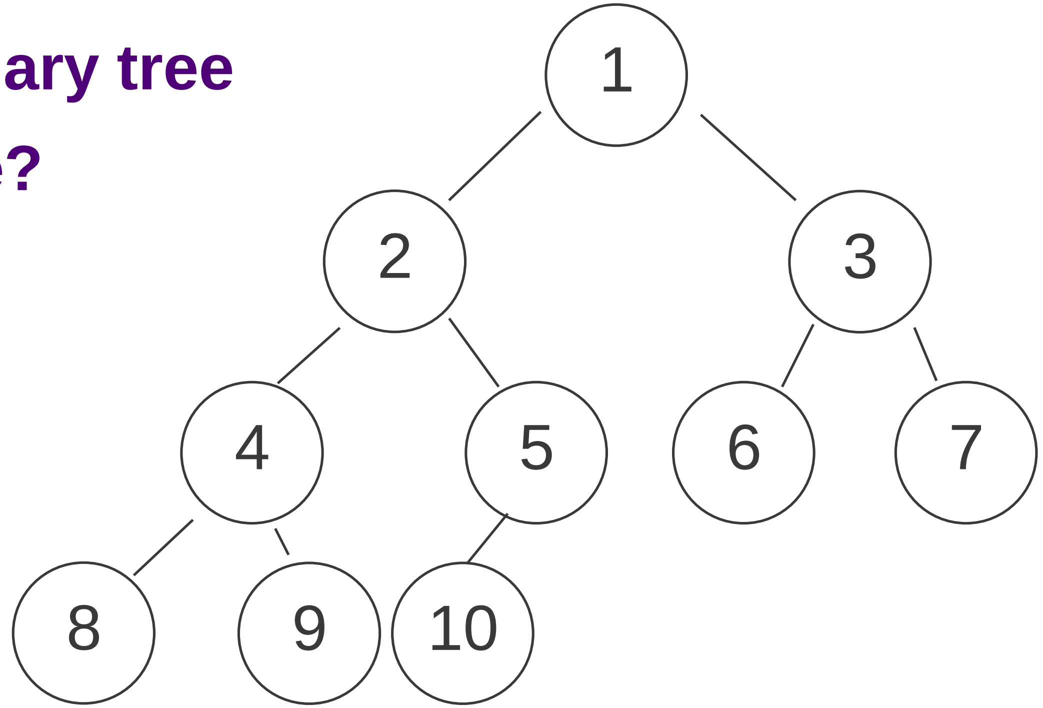
Priority Queue Implementation

Implementation

- **How can we implement this data type?**
- **With a binary tree made of linked nodes**
  - Downside: complex -- requires extra pointers to move up the tree (rise a node)
  - Which also means it requires a lot of extra memory
- **With an array**
  - Possible thanks to completeness
  - Very compact

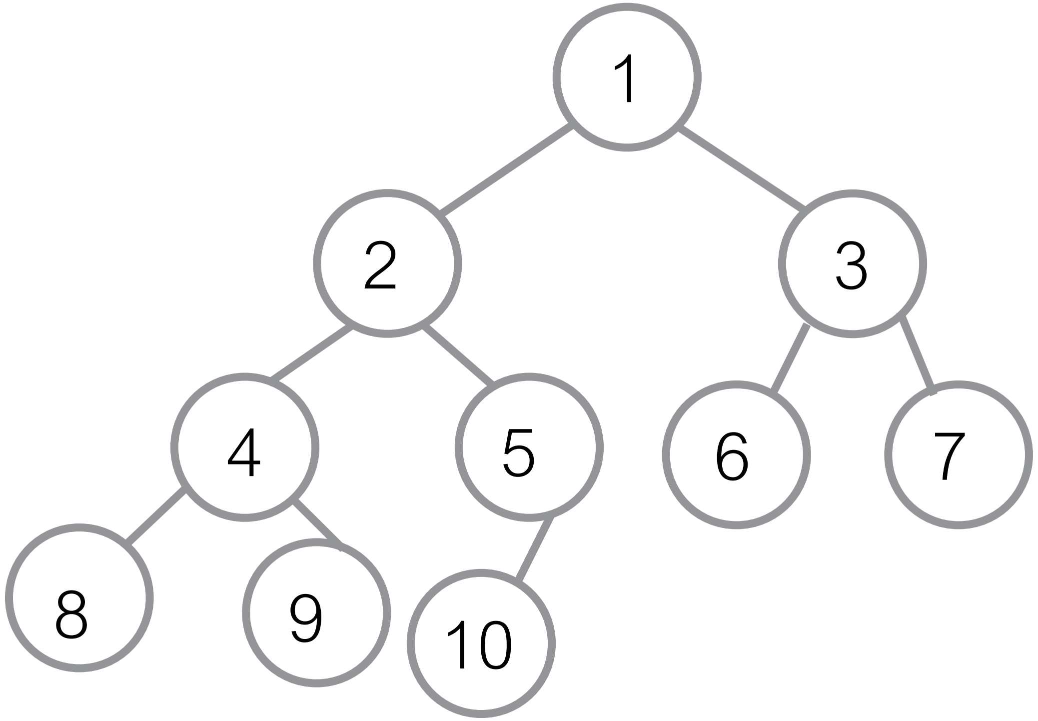
Implementing it with an array

- Consider the following complete binary tree
- Where are the children of each node?
- And the parent of each node?
- We will see in the next slides!



The figure shows both the binary tree and the array equivalent. We will highlight in grey each parent (so, inner nodes) with their children.

The table will show the array index of the grey nodes

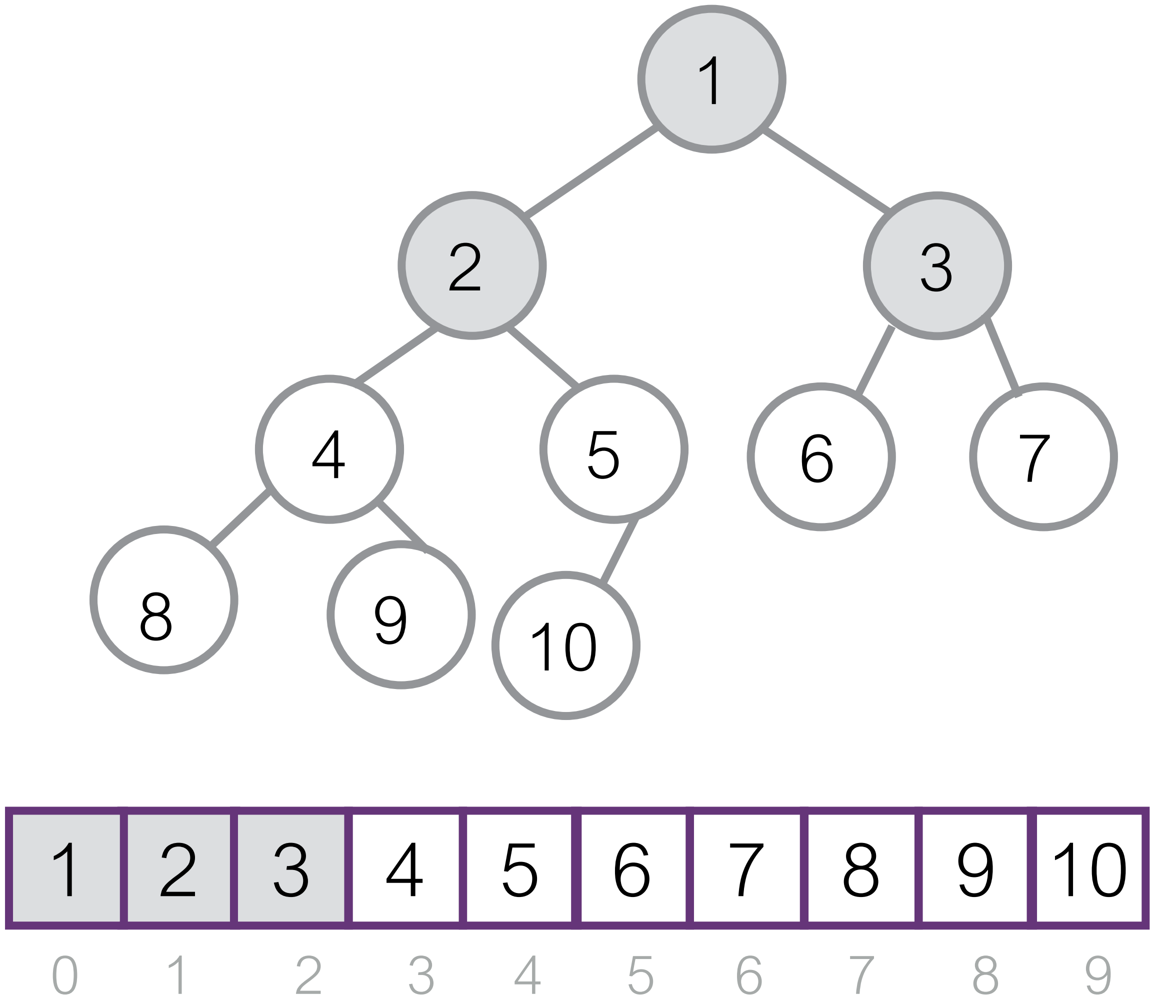


1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Parent Position	**Child Left**	**Child Right**

The figure shows both the binary tree and the array equivalent. We will highlight in grey each parent (so, inner nodes) with their children.

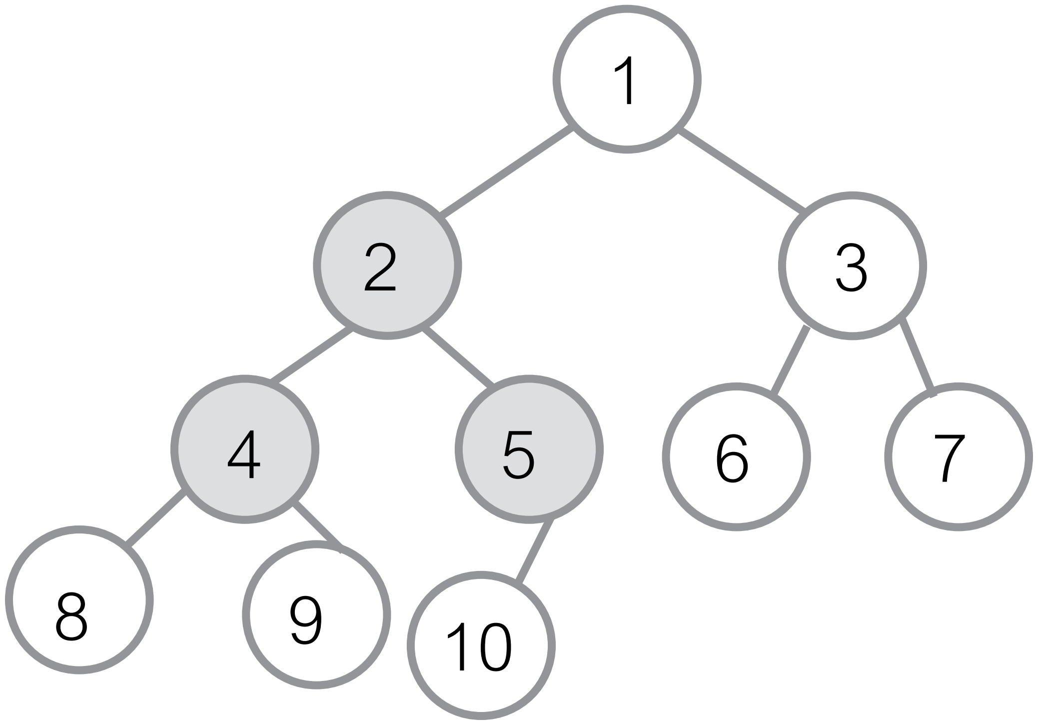
The table will show the array index of the grey nodes



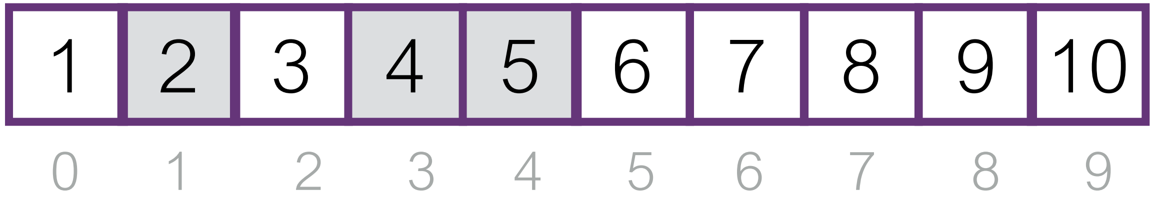
Parent Position	**Child Left**	**Child Right**
0	1	2

The figure shows both the binary tree and the array equivalent. We will highlight in grey each parent (so, inner nodes) with their children.

The table will show the array index of the grey nodes

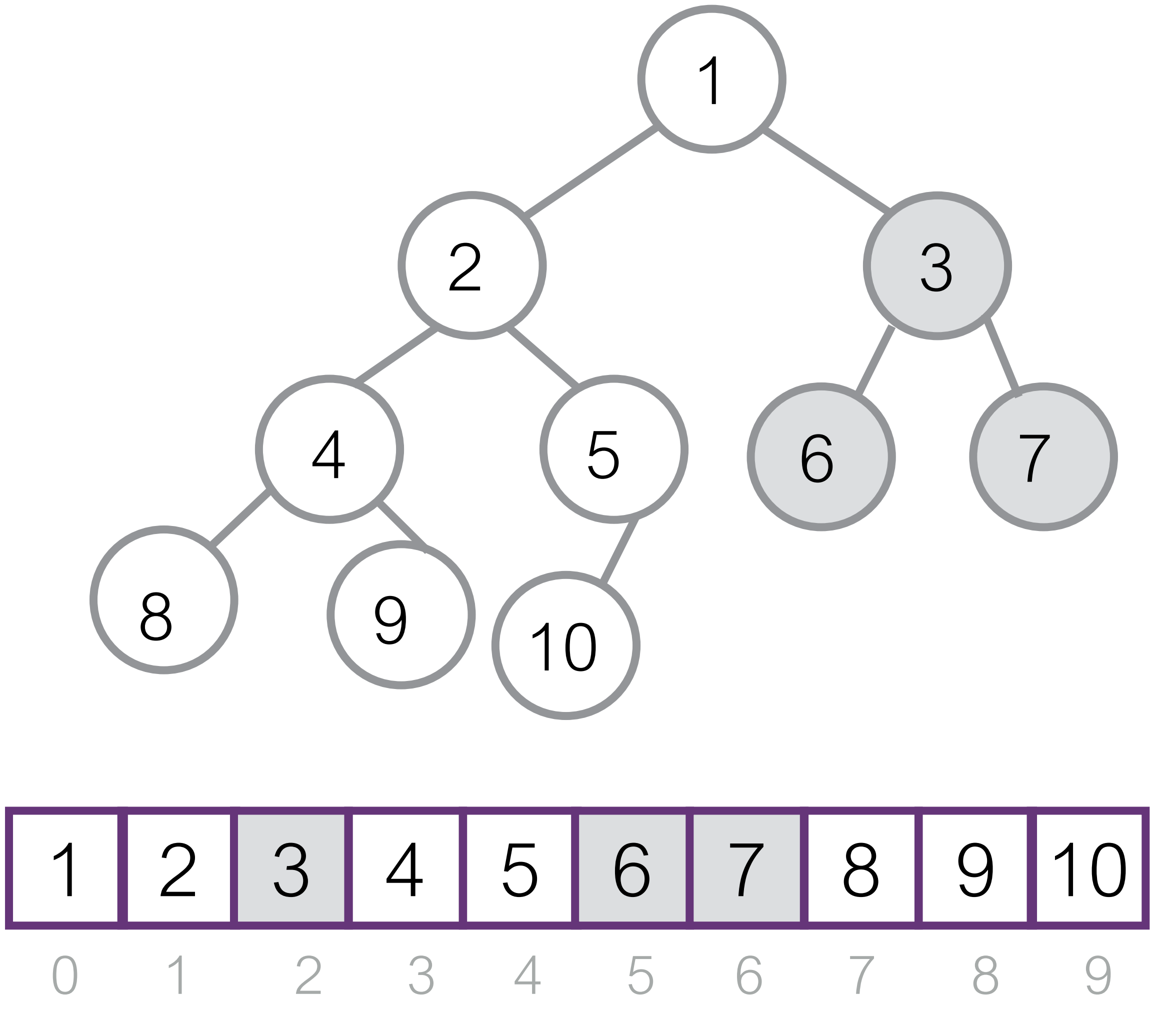


Parent Position	Child Left	Child Right
0	1	2
1	3	4



The figure shows both the binary tree and the array equivalent. We will highlight in grey each parent (so, inner nodes) with their children.

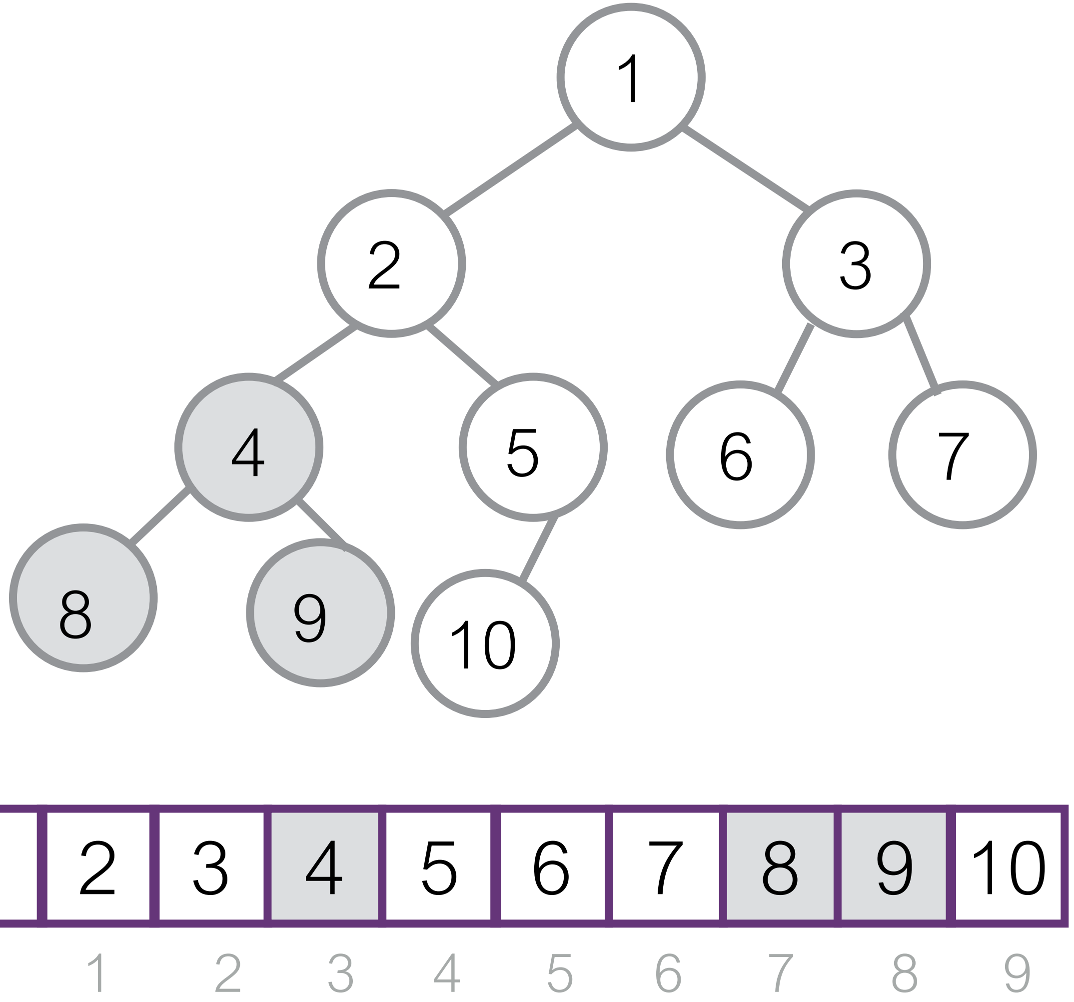
The table will show the array index of the grey nodes



Parent Position	Child Left	Child Right
0	1	2
1	3	4
2	5	6

The figure shows both the binary tree and the array equivalent. We will highlight in grey each parent (so, inner nodes) with their children.

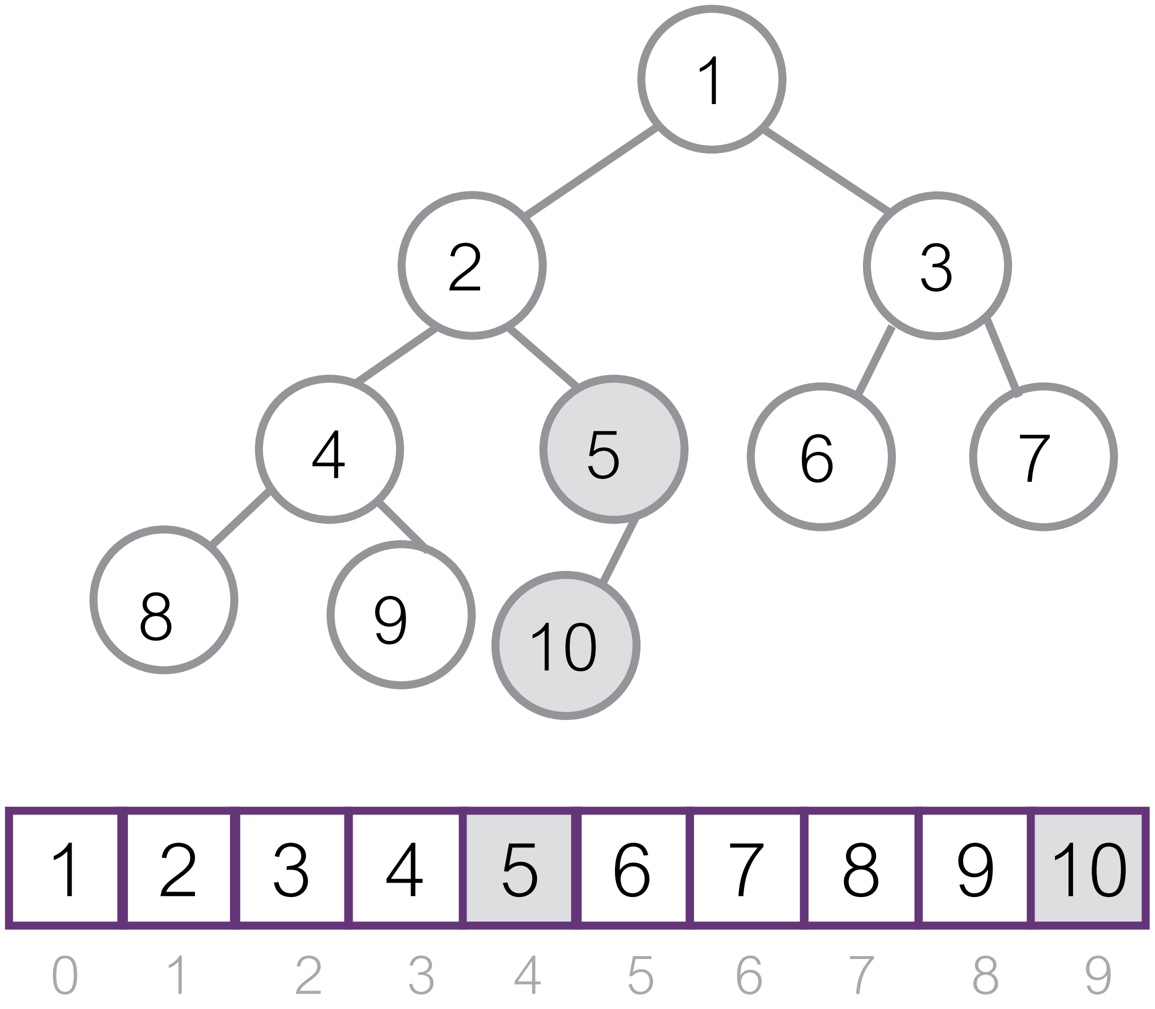
The table will show the array index of the grey nodes



Parent Position	Child Left	Child Right
0	1	2
1	3	4
2	5	6
3	7	8

The figure shows both the binary tree and the array equivalent. We will highlight in grey each parent (so, inner nodes) with their children.

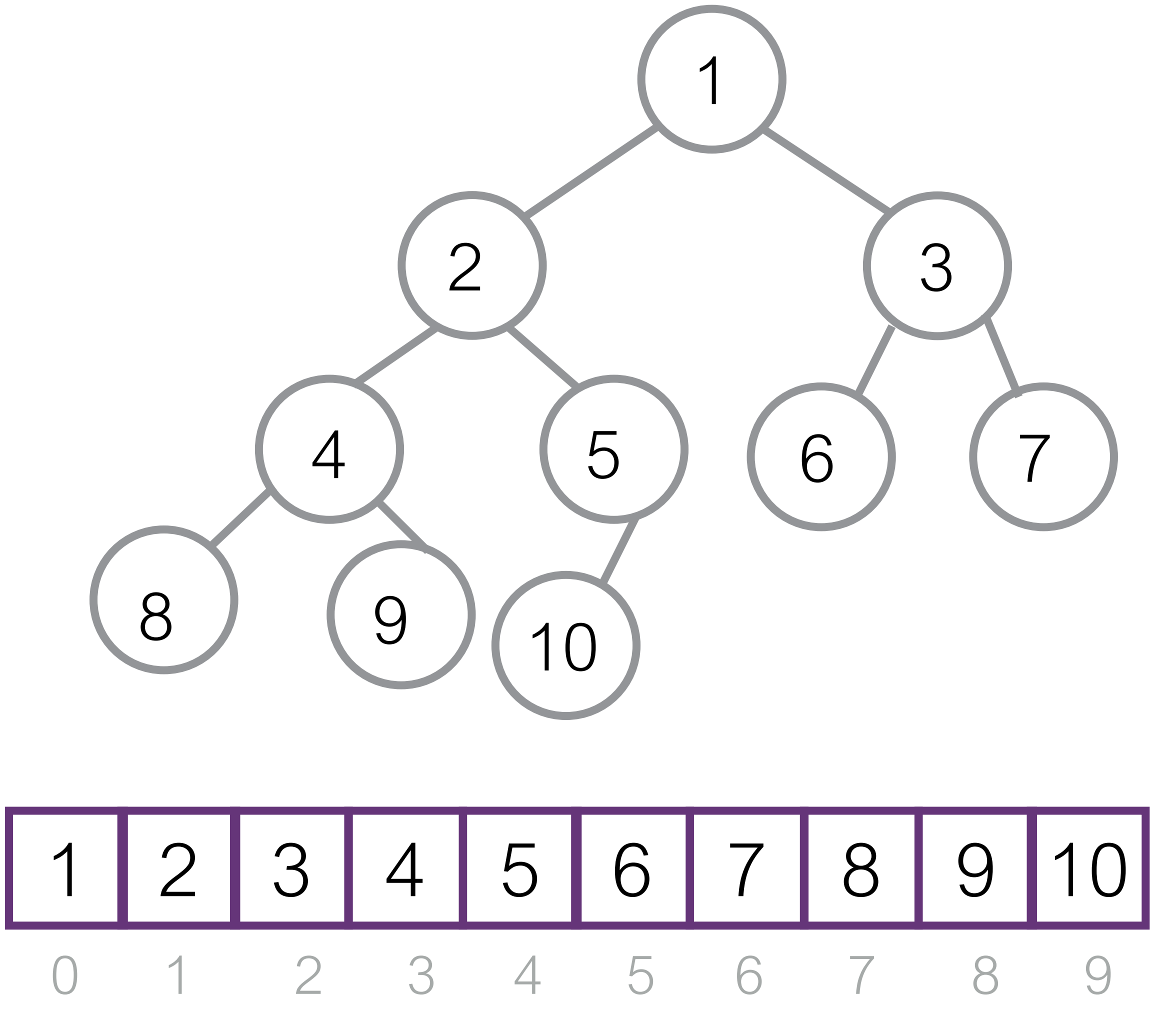
The table will show the array index of the grey nodes



Parent Position	**Child Left**	**Child Right**
0	1	2
1	3	4
2	5	6
3	7	8
4	9	

The figure shows both the binary tree and the array equivalent. We will highlight in grey each parent (so, inner nodes) with their children.

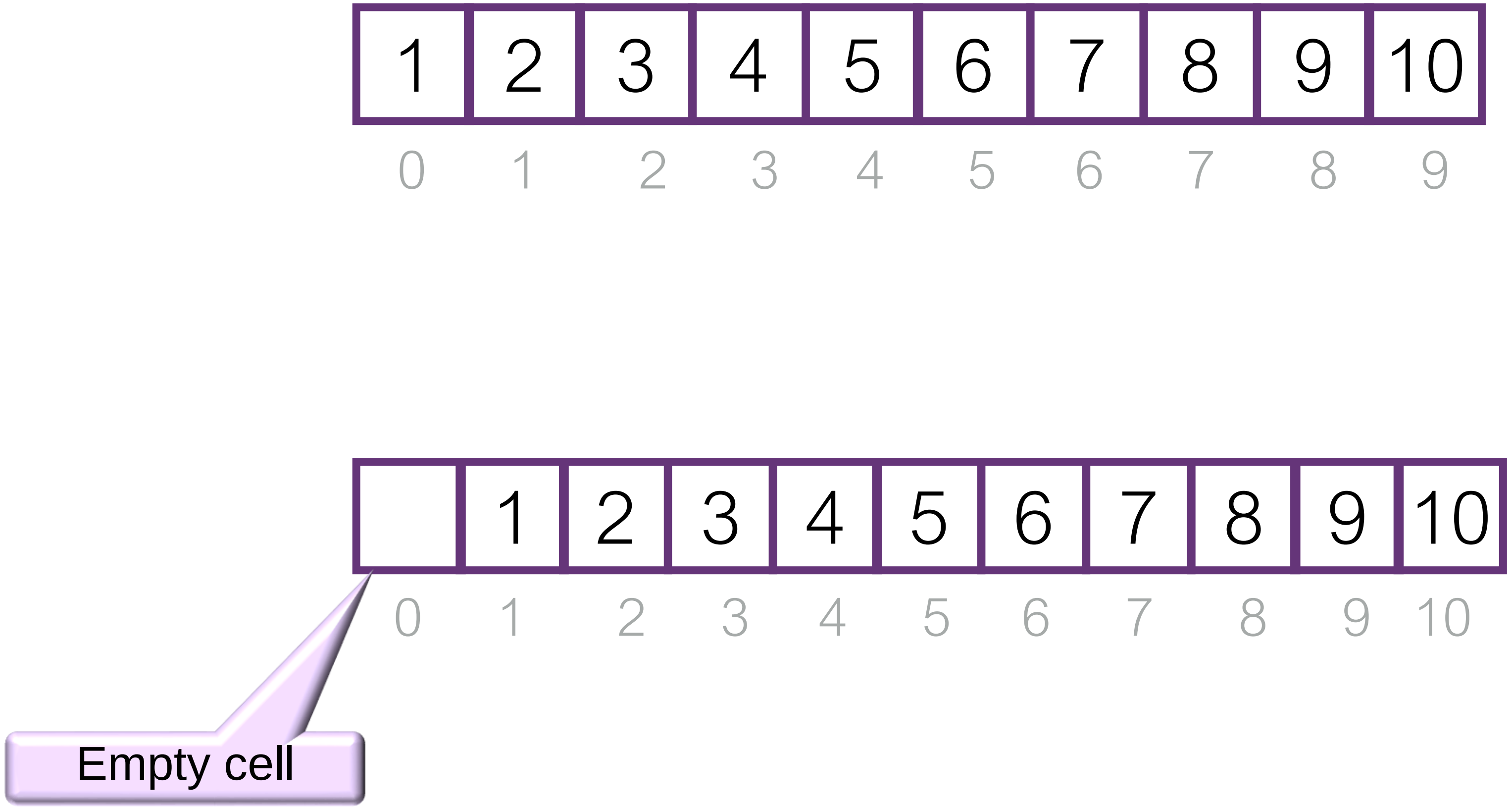
The table will show the array index of the grey nodes



Parent Position	**Child Left**	**Child Right**
0	1	2
1	3	4
2	5	6
3	7	8
4	9	
k	**?**	**?**

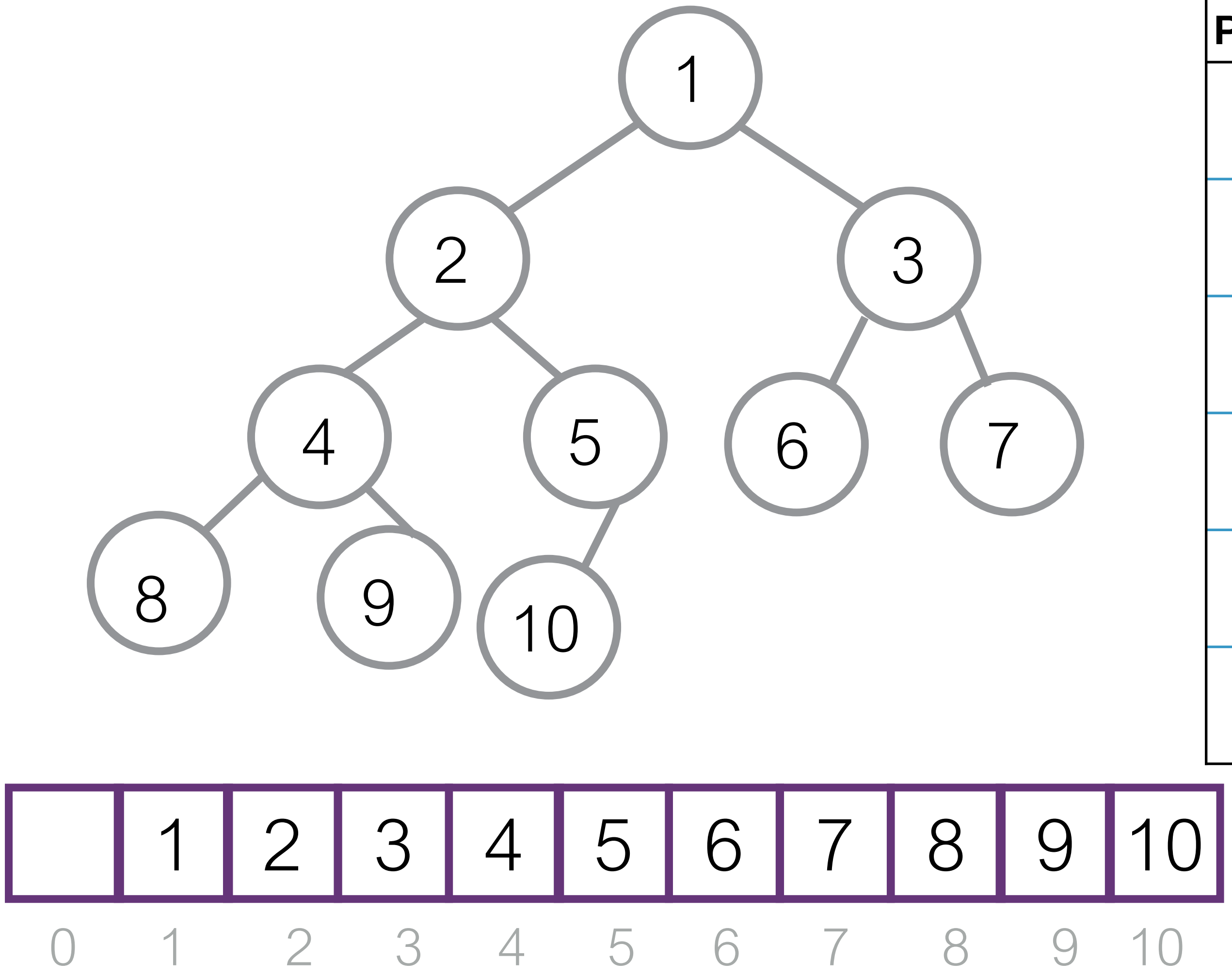
Not as clear as we would like

If we shift by 1, it will clarify positions



The figure shows both the binary tree and the array equivalent. We will highlight in grey each parent (so, inner nodes) with their children.

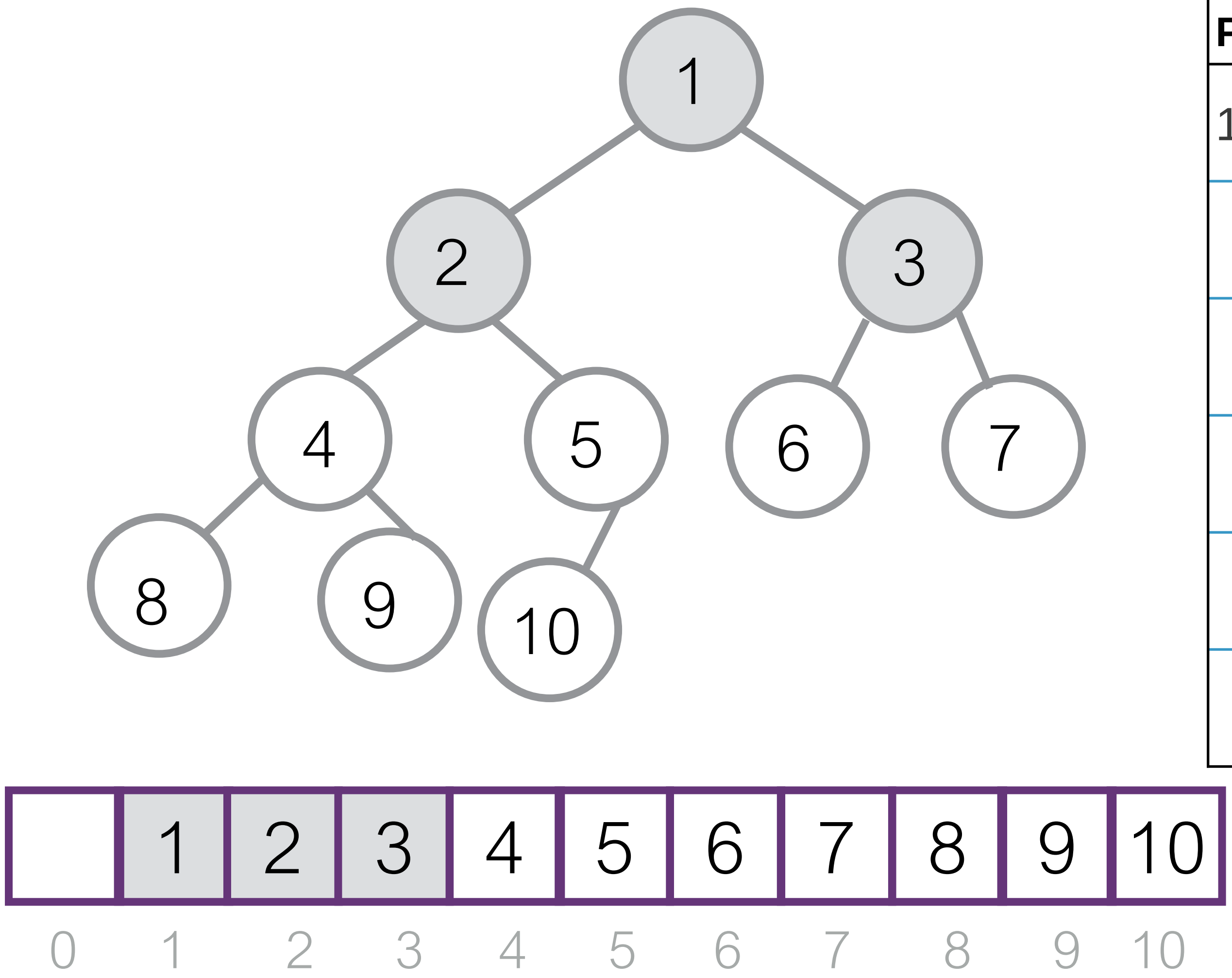
The table will show the array index of the grey nodes



Parent Position	**Child Left**	**Child Right**

The figure shows both the binary tree and the array equivalent. We will highlight in grey each parent (so, inner nodes) with their children.

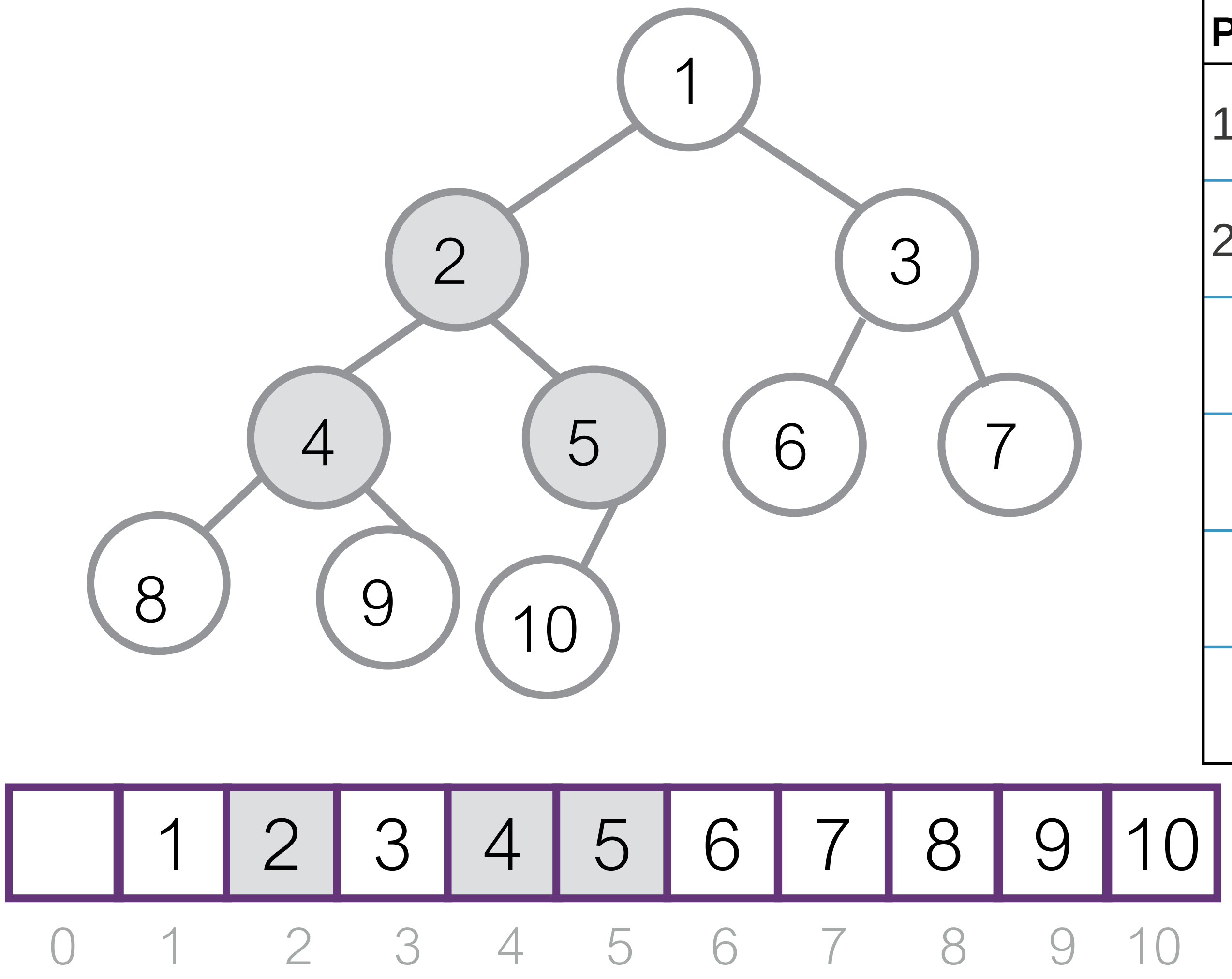
The table will show the array index of the grey nodes



Parent Position	**Child Left**	**Child Right**
1	2	3

The figure shows both the binary tree and the array equivalent. We will highlight in grey each parent (so, inner nodes) with their children.

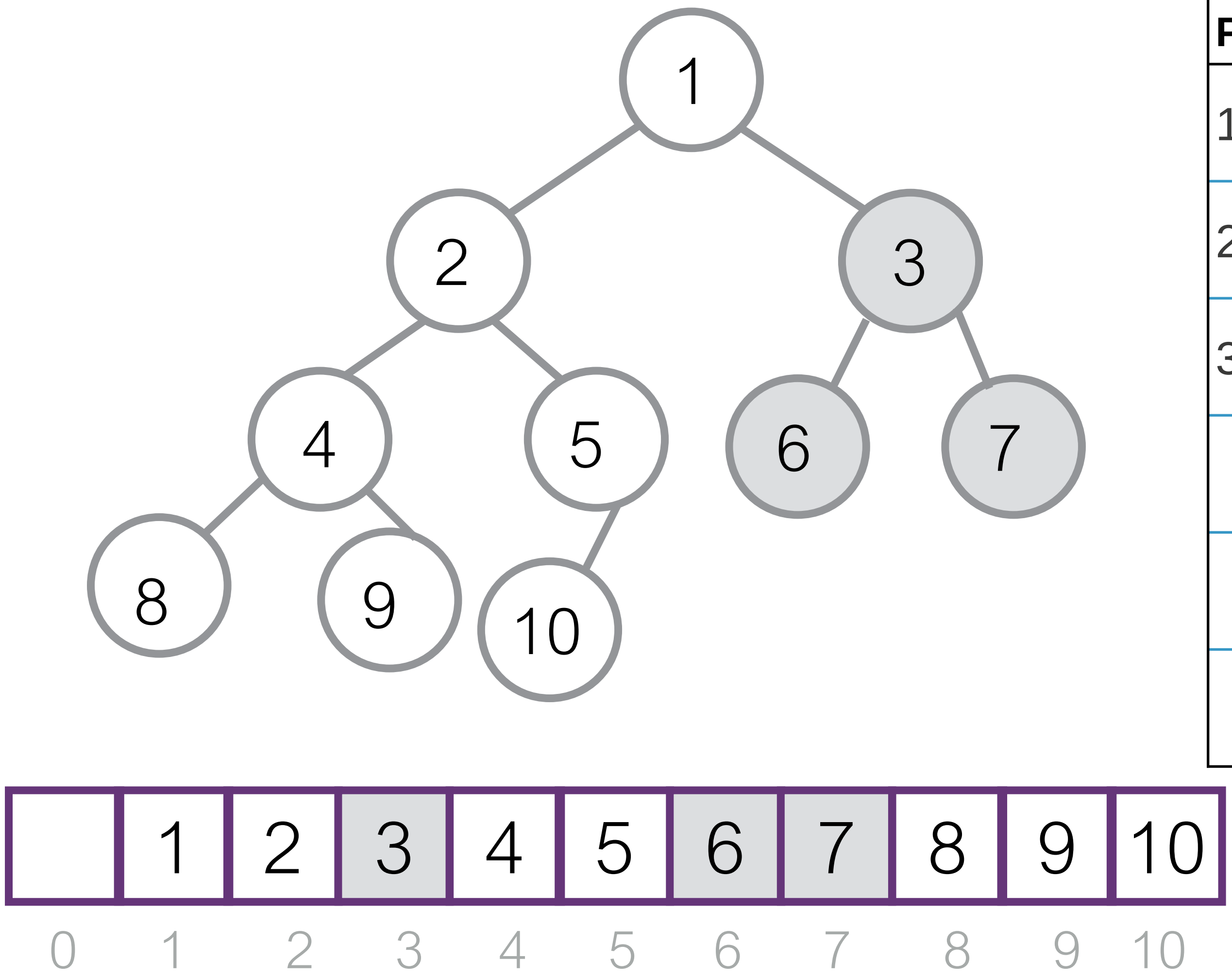
The table will show the array index of the grey nodes



Parent Position	**Child Left**	**Child Right**
1	2	3
2	4	5

The figure shows both the binary tree and the array equivalent. We will highlight in grey each parent (so, inner nodes) with their children.

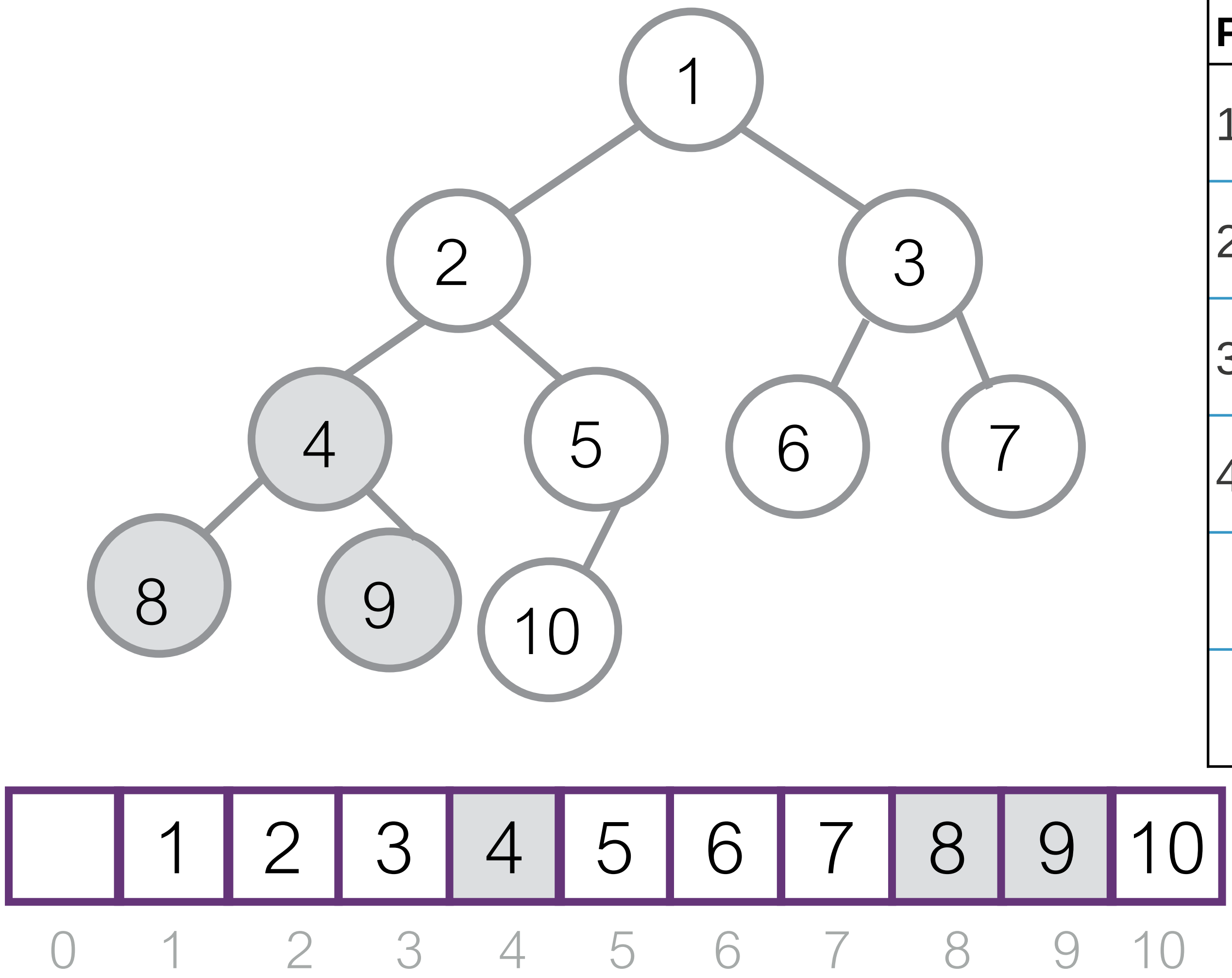
The table will show the array index of the grey nodes



Parent Position	Child Left	Child Right
1	2	3
2	4	5
3	6	7

The figure shows both the binary tree and the array equivalent. We will highlight in grey each parent (so, inner nodes) with their children.

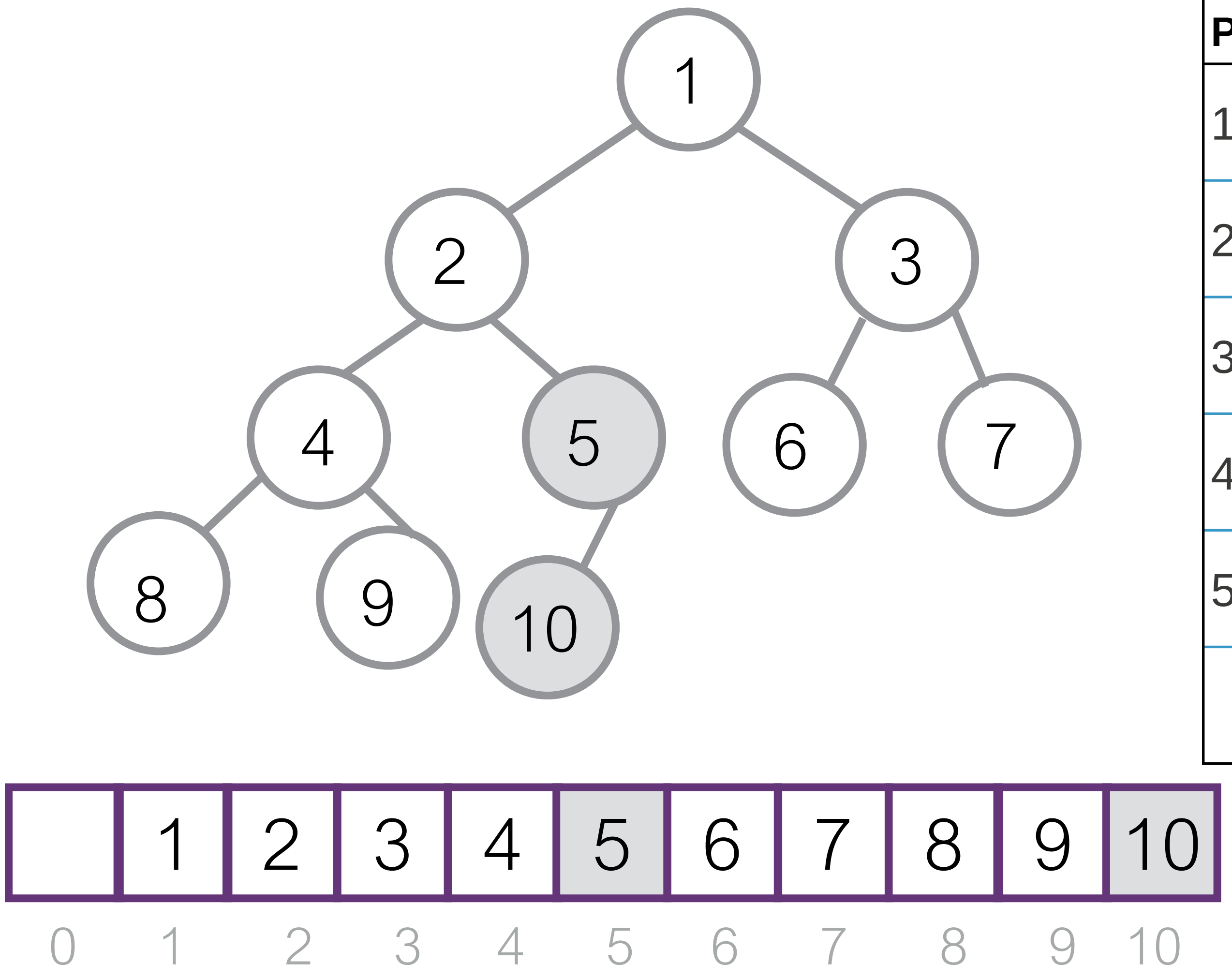
The table will show the array index of the grey nodes



Parent Position	**Child Left**	**Child Right**
1	2	3
2	4	5
3	6	7
4	8	9

The figure shows both the binary tree and the array equivalent. We will highlight in grey each parent (so, inner nodes) with their children.

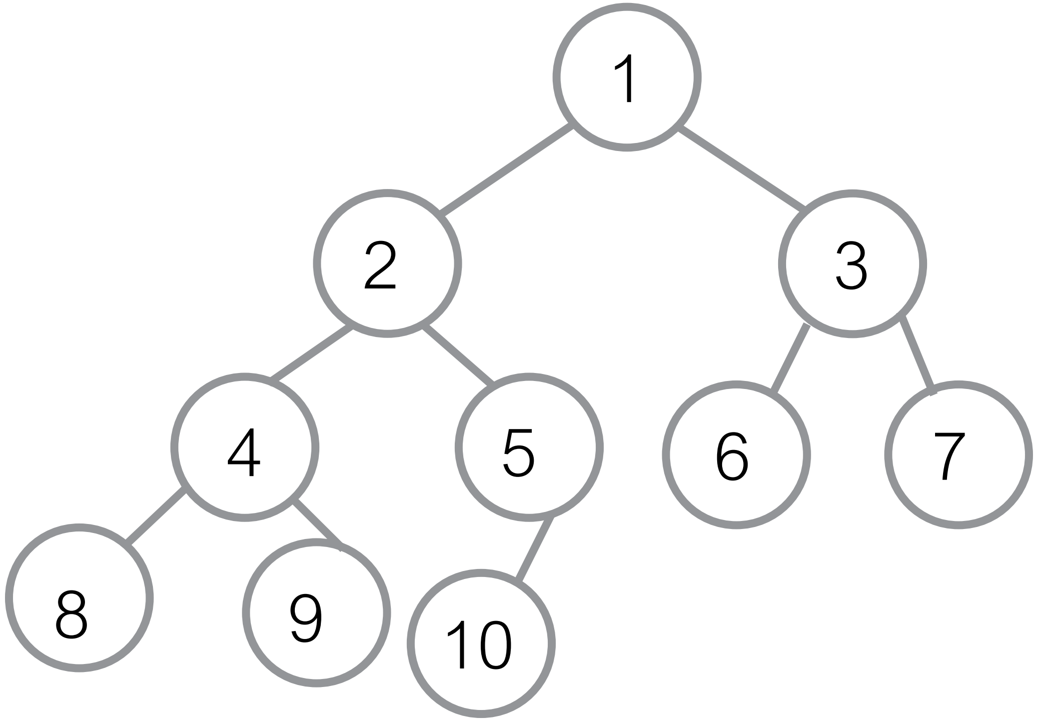
The table will show the array index of the grey nodes



Parent Position	Child Left	Child Right
1	2	3
2	4	5
3	6	7
4	8	9
5	10	

The figure shows both the binary tree and the array equivalent. We will highlight in grey each parent (so, inner nodes) with their children.

The table will show the array index of the grey nodes

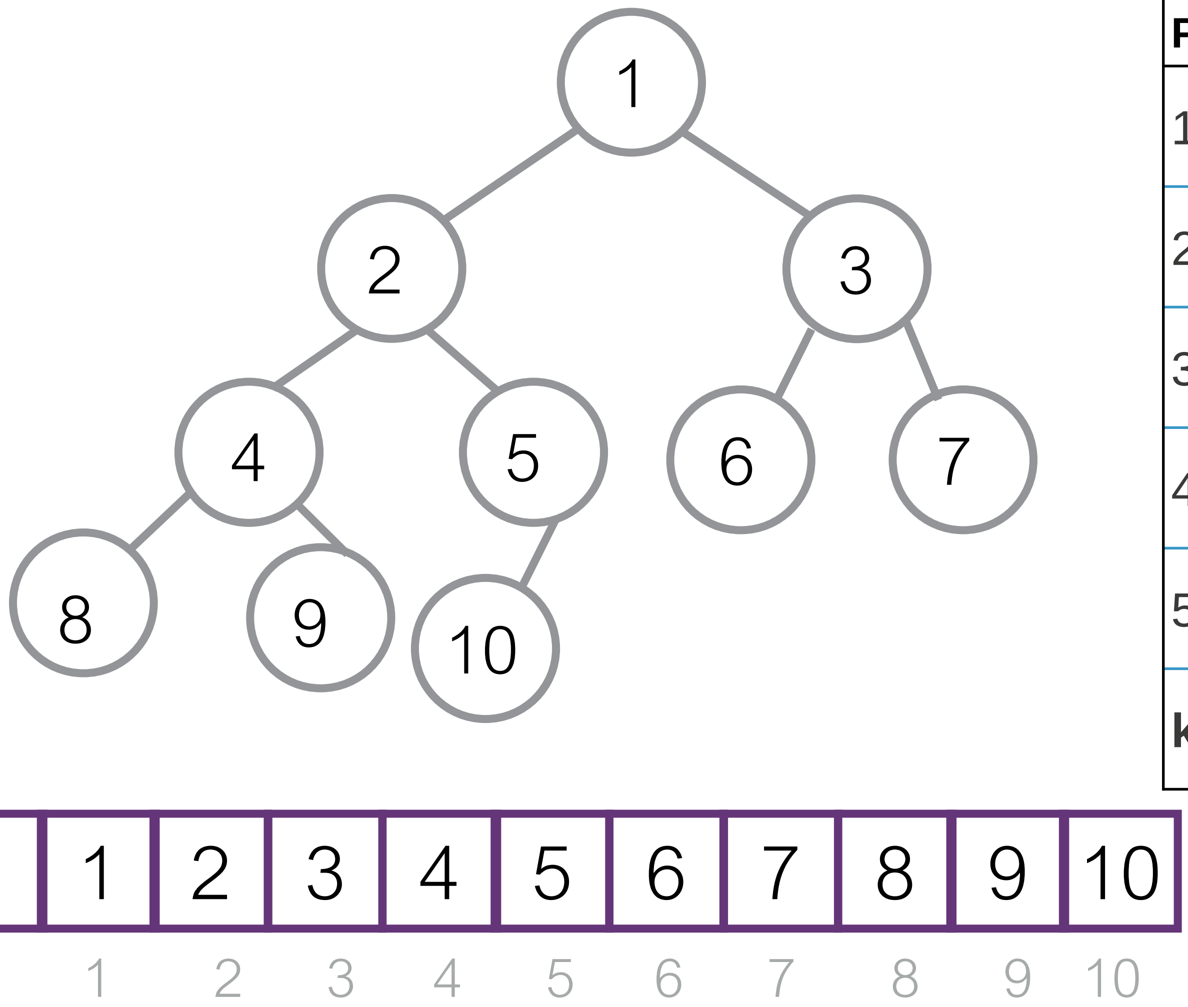


Parent Position	Child Left	Child Right
1	2	3
2	4	5
3	6	7
4	8	9
5	10	
k		

	1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9	10

The figure shows both the binary tree and the array equivalent. We will highlight in grey each parent (so, inner nodes) with their children.

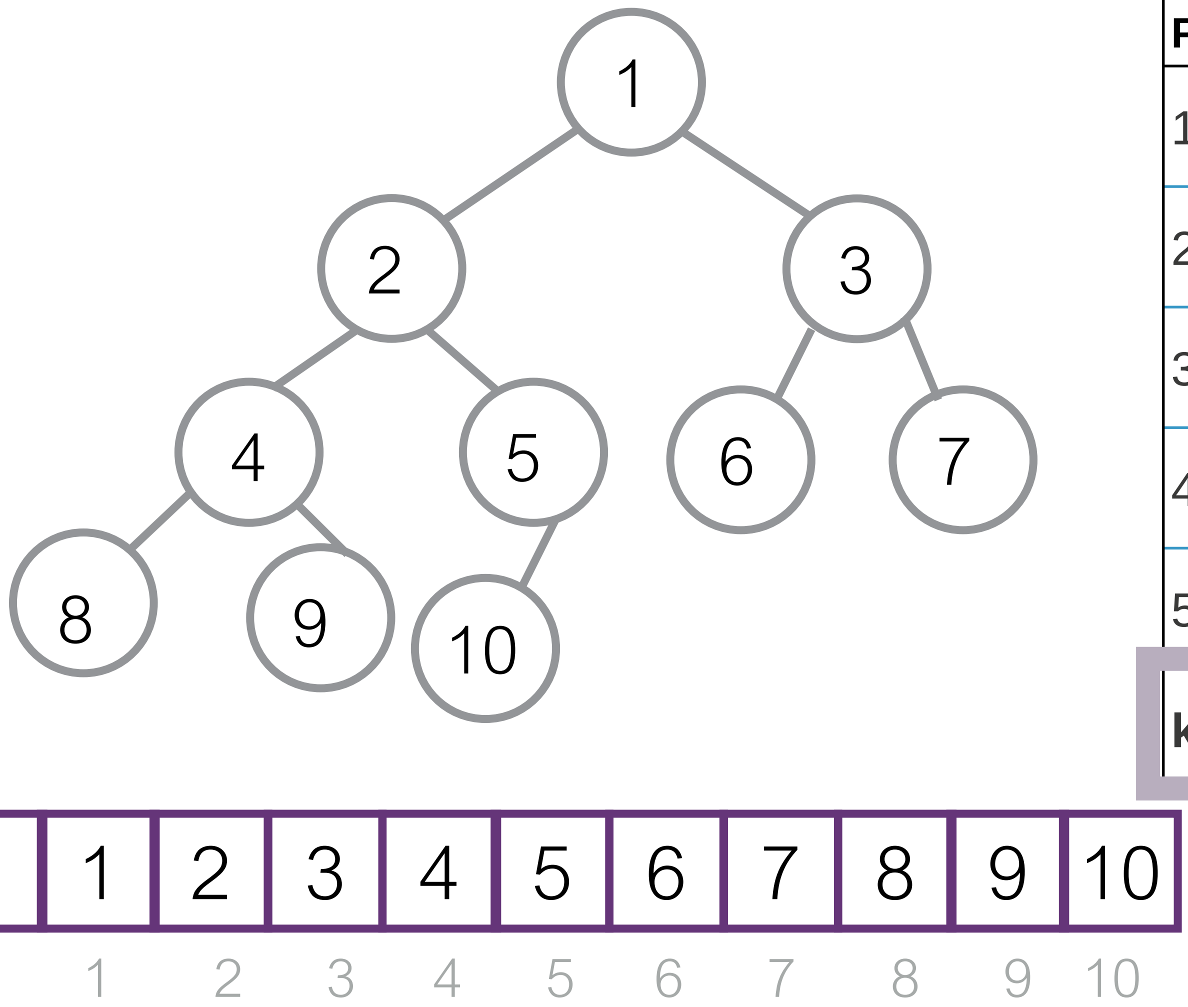
The table will show the array index of the grey nodes



Parent Position	**Child Left**	**Child Right**
1	2	3
2	4	5
3	6	7
4	8	9
5	10	
k	**2*k**	

The figure shows both the binary tree and the array equivalent. We will highlight in grey each parent (so, inner nodes) with their children.

The table will show the array index of the grey nodes



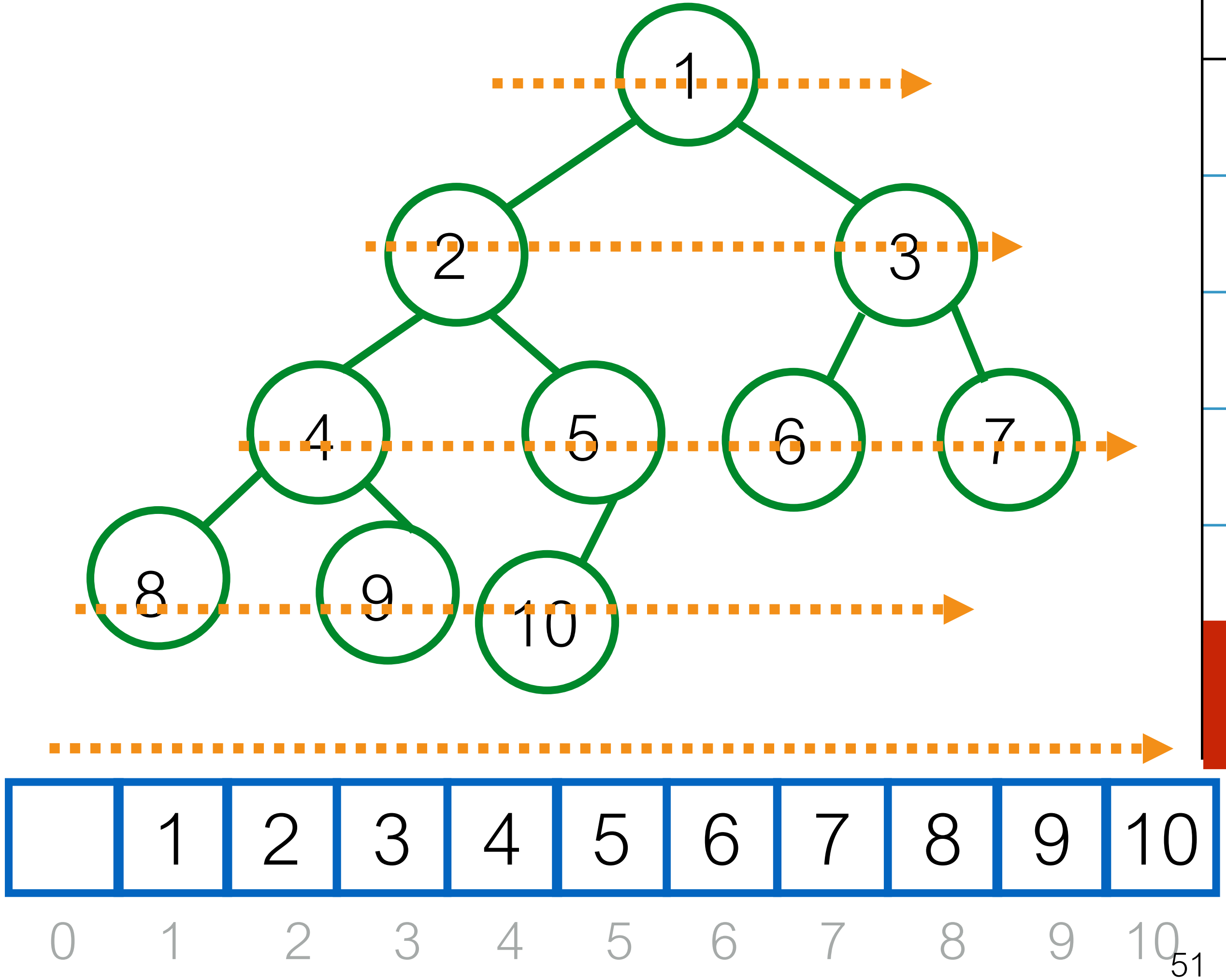
Parent Position	Child Left	Child Right
1	2	3
2	4	5
3	6	7
4	8	9
5	10	
k	**2*k**	**2*k+1**

Now is clear!

Root at position 1

Children of k: 2*k 2*k+1 (if they exist)

Parent of k: position k//2 (except for root)



Parent Position	Child Left	Child Right
1	2	3
2	4	5
3	6	7
4	8	9
5	10	
k	**2*k**	**2*k+1**

A concrete implementation

```
class Heap(Generic[T]):
    MIN_CAPACITY = 1

    def __init__(self, max_size: int) -> None:
        self.length = 0
        self.the_array = ArrayR(max(self.MIN_CAPACITY, max_capacity) + 1)

    def __len__(self) -> int:
        return self.length

    def is_full(self) -> bool:
        return self.length+1 == len(self.the_array)
```

How many items are stored. Points to the last item added (if any) – rather than to the first empty cell

Indices start at 1

- **Note that we need an extra cell in the array**
  - If our heap has 10 elements, we use indices 1..10 to store them
    - Thus, we need an array of 11 cells

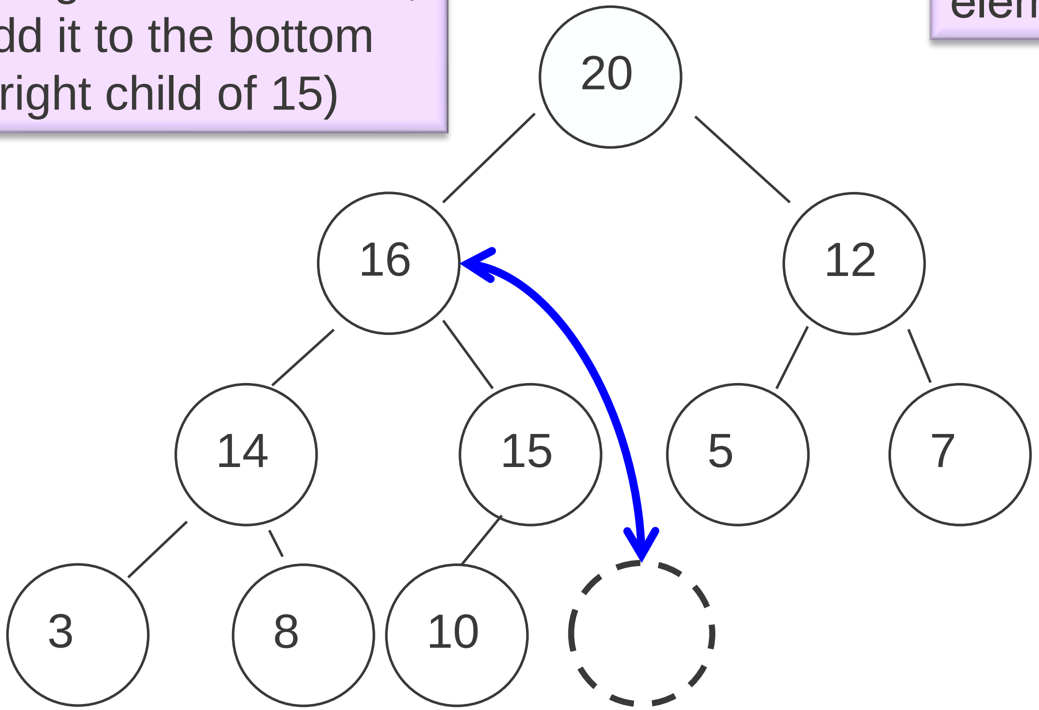
Heap Implementation
Add

Recall: adding a new element (say 18)

When adding a new element, we first add it to the bottom element (right child of 15)

We then need to make this element “rise” to the right place

To rise we need to compare with the parent, if any



Algorithm for the add operation

1. **If there is no space, return `False`**
2. **If there is space**

   II. Put new element at the bottom of heap

   III. While the heap-order is broken (element is smaller than its parent)

   - rise: swap new element with parent

   IV. Return `True`

We could also have decided to resize when full and returned nothing…

A concrete implementation for `add`

```
"""Rise element at index k to its correct position
:pre: 1<= k <= self.length"""
def rise(self, k: int)-> None:
    while k > 1 and self.the_array[k] > self.the_array[k//2]:
        self.swap(k, k//2)
        k = k//2

def add(self, element: T) -> bool:
    has_space_left = not self.is_full()

    if has_space_left:
        self.length += 1
        self.the_array[self.length] = element
        self.rise(self.length)

    return has_space_left
```

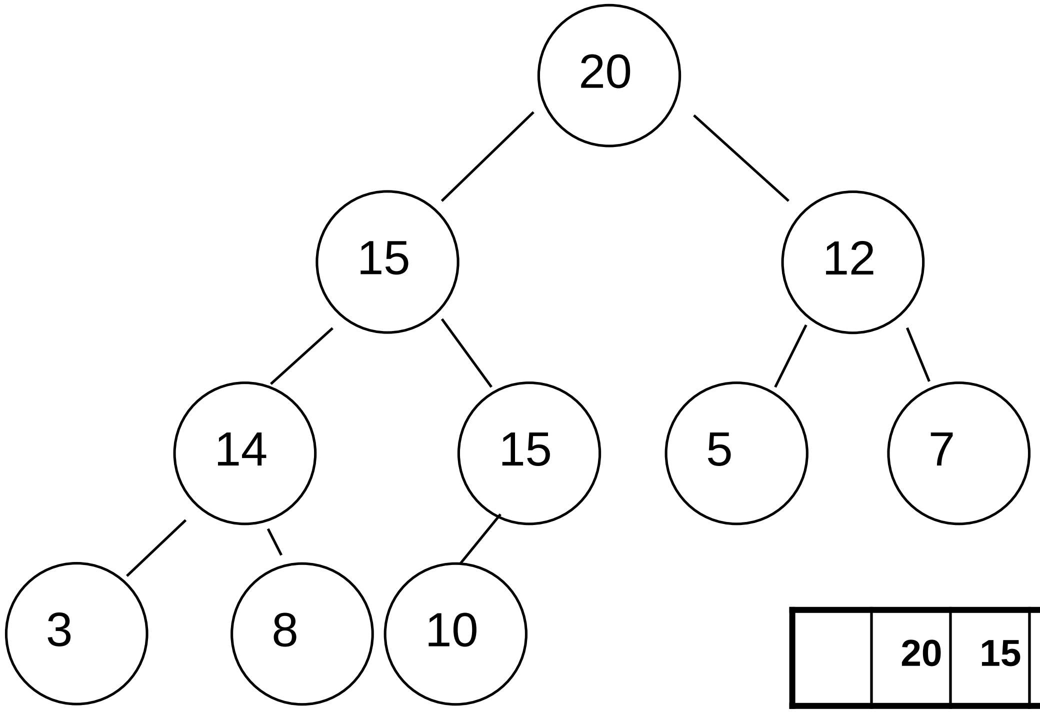
Not enforced

k has a parent

parent is smaller

Remember, if k has a parent, it is found at k//2

	20	15	12	14	15	5	7	3	8	10			
0	1	2	3	4	5	6	7	8	9	10	11	12	13



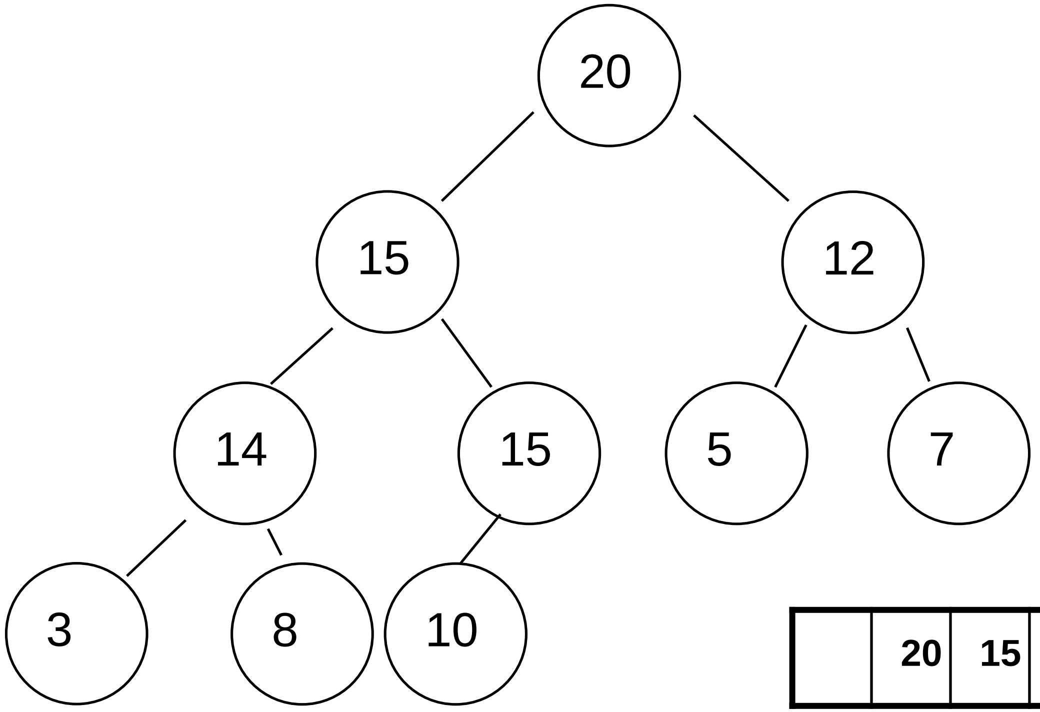
Assume we want to add element 18

```
self.length = 10
self.the_array =
```

	20	**15**	**12**	**14**	**15**	**5**	**7**	**3**	**8**	**10**			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def rise(self, k: int) -> None:
    while k > 1 and self.the_array[k] > self.the_array[k//2]:
        self.swap(k, k//2)
        k = k//2

def add(self, element: T) -> bool:
    has_space_left = not self.is_full()
    if has_space_left:
        self.length += 1
        self.the_array[self.length] = element
        self.rise(self.length)
    return has_space_left
```



Assume we want to add element 18

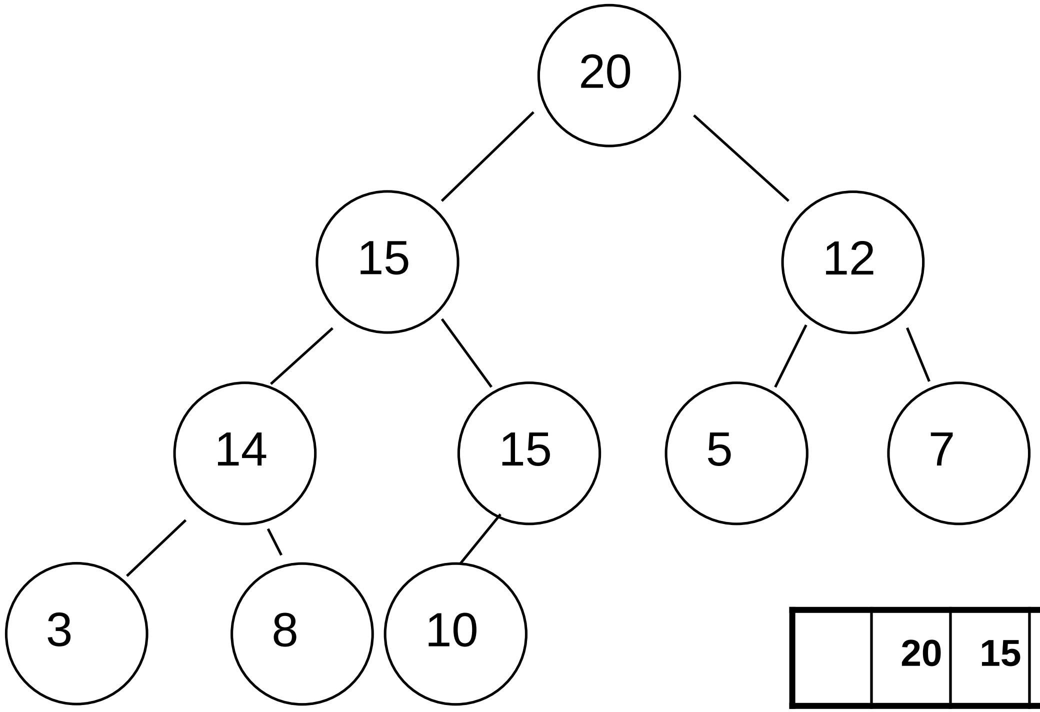
`self.length = 10`

`self.the_array =`

	20	15	12	14	15	5	7	3	8	10			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def rise(self, k: int) -> None:
    while k > 1 and self.the_array[k] > self.the_array[k//2]:
        self.swap(k, k//2)
        k = k//2

def add(self, element: T) -> bool:
    has_space_left = not self.is_full()
    if has_space_left:
        self.length += 1
        self.the_array[self.length] = element
        self.rise(self.length)
    return has_space_left
```



Assume we want to add element 18

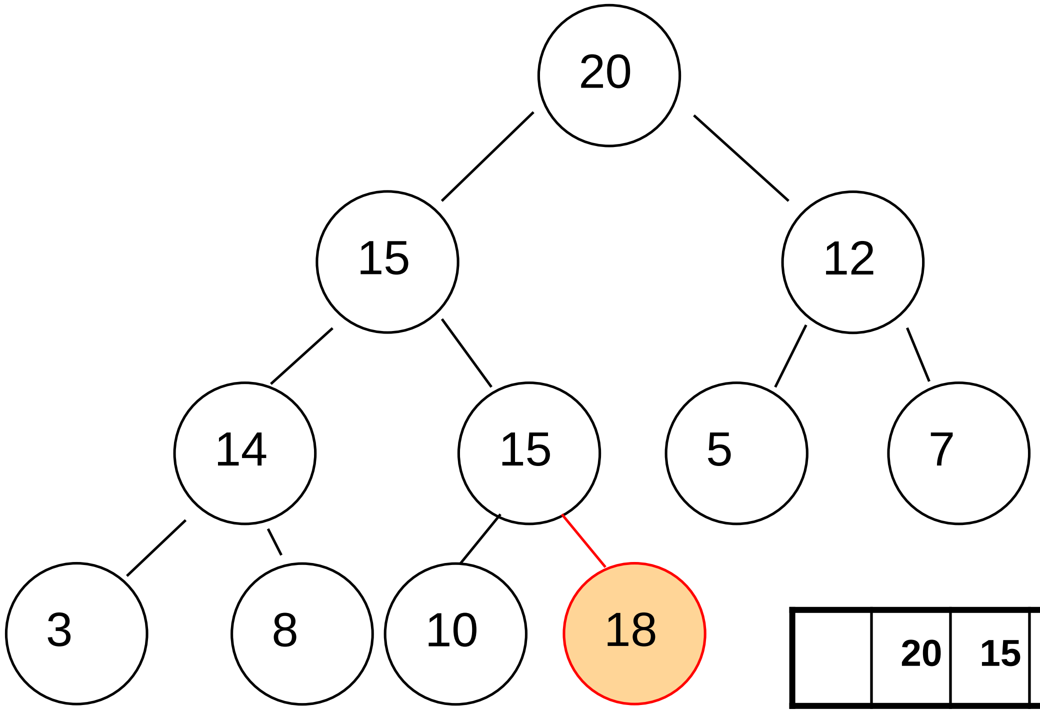
`self.length = 11`

`self.the_array =`

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	20	15	12	14	15	5	7	3	8	10			

```
def rise(self, k: int) -> None:
    while k > 1 and self.the_array[k] > self.the_array[k//2]:
        self.swap(k, k//2)
        k = k//2

def add(self, element: T) -> bool:
    has_space_left = not self.is_full()
    if has_space_left:
        self.length += 1
        self.the_array[self.length] = element
        self.rise(self.length)
    return has_space_left
```



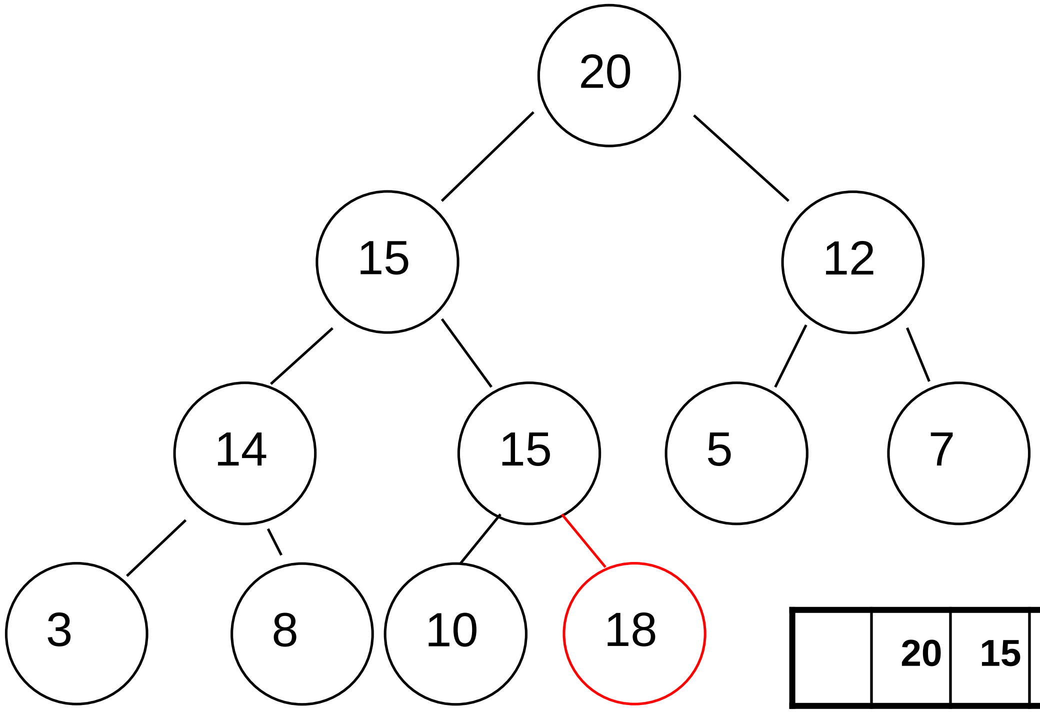
Assume we want to add element 18

```
self.length = 11
self.the_array =
```

	20	**15**	**12**	**14**	**15**	**5**	**7**	**3**	**8**	**10**	**18**		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def rise(self, k: int) -> None:
    while k > 1 and self.the_array[k] > self.the_array[k//2]:
        self.swap(k, k//2)
        k = k//2

def add(self, element: T) -> bool:
    has_space_left = not self.is_full()
    if has_space_left:
        self.length += 1
        self.the_array[self.length] = element
        self.rise(self.length)
    return has_space_left
```



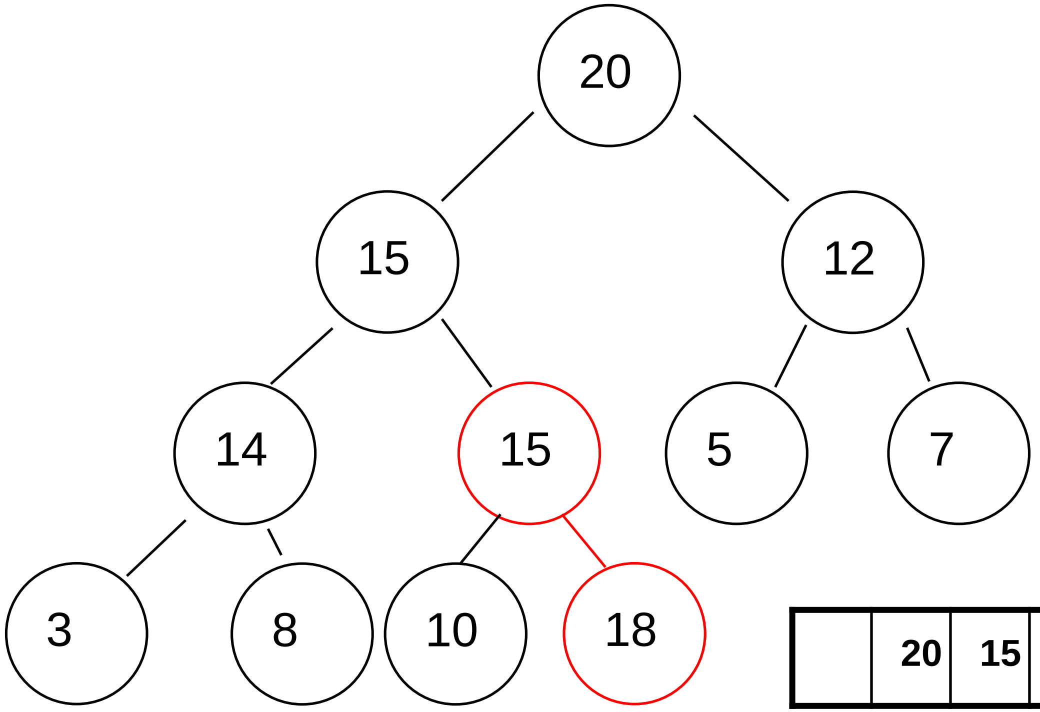
Assume we want to add element 18

```
self.length = 11
self.the_array =
```

	20	**15**	**12**	**14**	**15**	**5**	**7**	**3**	**8**	**10**	**18**		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def rise(self, k: int) -> None:
    while k > 1 and self.the_array[k] > self.the_array[k//2]:
        self.swap(k, k//2)
        k = k//2

def add(self, element: T) -> bool:
    has_space_left = not self.is_full()
    if has_space_left:
        self.length += 1
        self.the_array[self.length] = element
        self.rise(self.length)
    return has_space_left
```



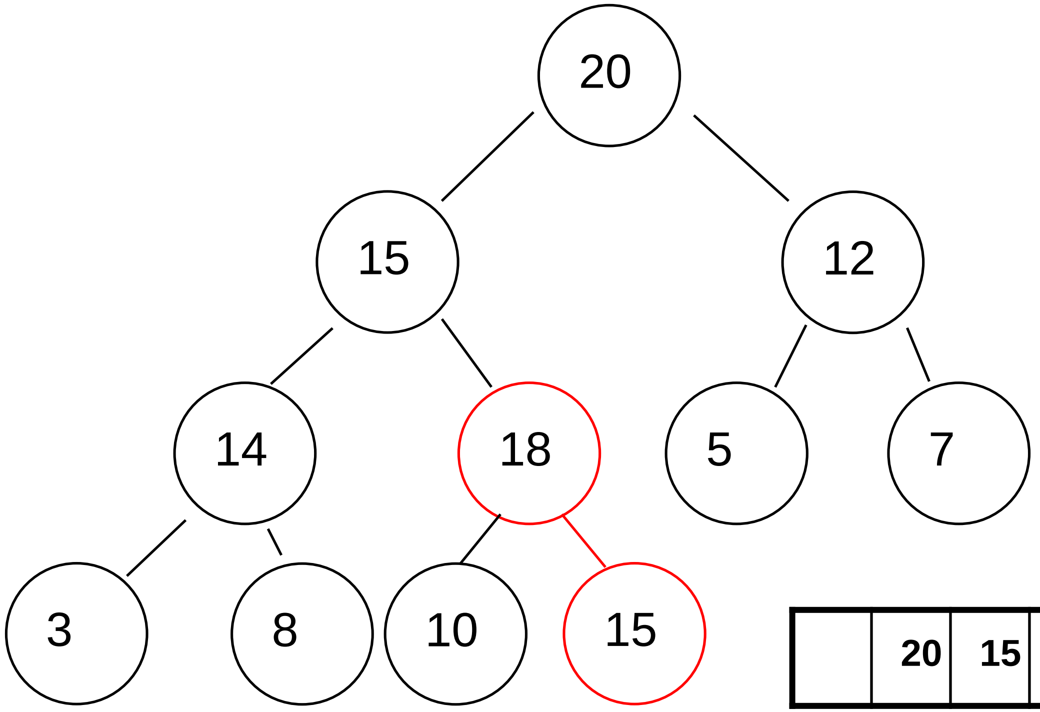
Assume we want to add element 18

```
self.length = 11
self.the_array =
```

	20	15	12	14	15	5	7	3	8	10	18		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def rise(self, k: int) -> None:
    while k > 1 and self.the_array[k] > self.the_array[k//2]:
        self.swap(k, k//2)
        k = k//2

def add(self, element: T) -> bool:
    has_space_left = not self.is_full()
    if has_space_left:
        self.length += 1
        self.the_array[self.length] = element
        self.rise(self.length)
    return has_space_left
```



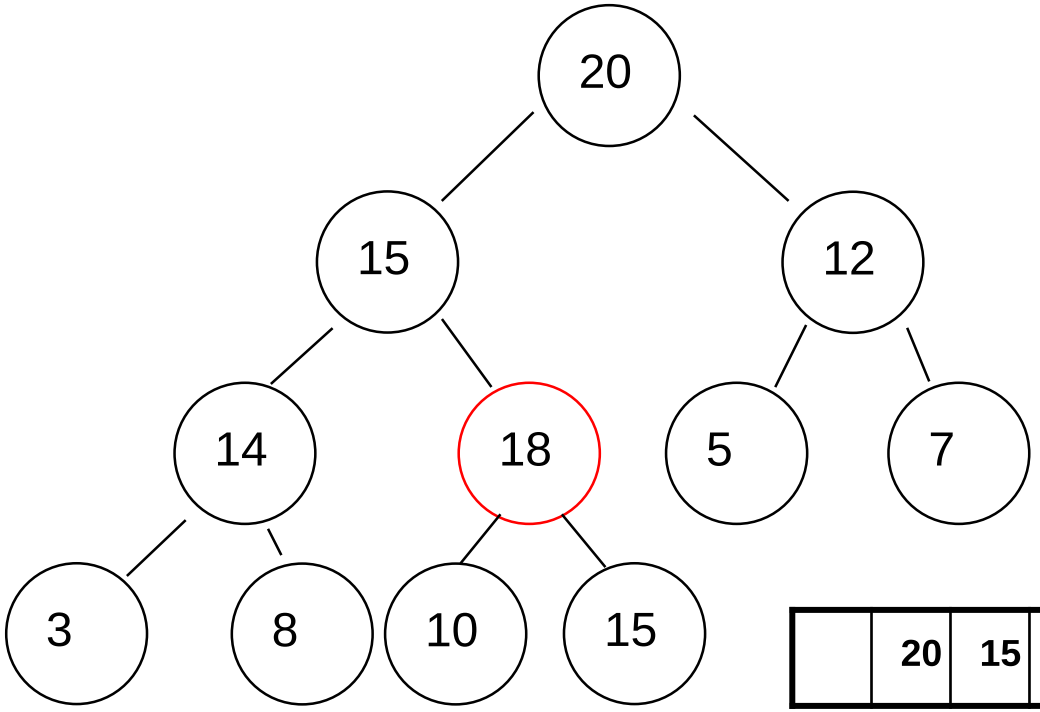
Assume we want to add element 18

```
self.length = 11
self.the_array =
```

	20	**15**	**12**	**14**	**18**	**5**	**7**	**3**	**8**	**10**	**15**		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def rise(self, k: int) -> None:
    while k > 1 and self.the_array[k] > self.the_array[k//2]:
        self.swap(k, k//2)
        k = k//2

def add(self, element: T) -> bool:
    has_space_left = not self.is_full()
    if has_space_left:
        self.length += 1
        self.the_array[self.length] = element
        self.rise(self.length)
    return has_space_left
```



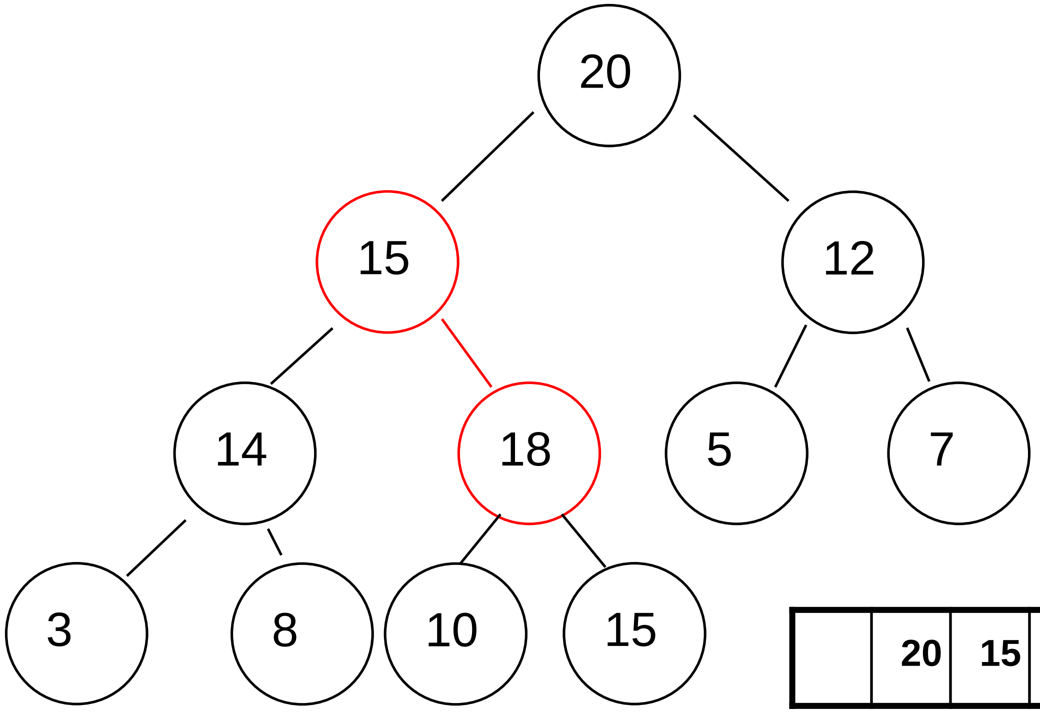
Assume we want to add element 18

```
self.length = 11
self.the_array =
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	20	15	12	14	18	5	7	3	8	10	15		

```
def rise(self, k: int) -> None:
    while k > 1 and self.the_array[k] > self.the_array[k//2]:
        self.swap(k, k//2)
        k = k//2

def add(self, element: T) -> bool:
    has_space_left = not self.is_full()
    if has_space_left:
        self.length += 1
        self.the_array[self.length] = element
        self.rise(self.length)
    return has_space_left
```



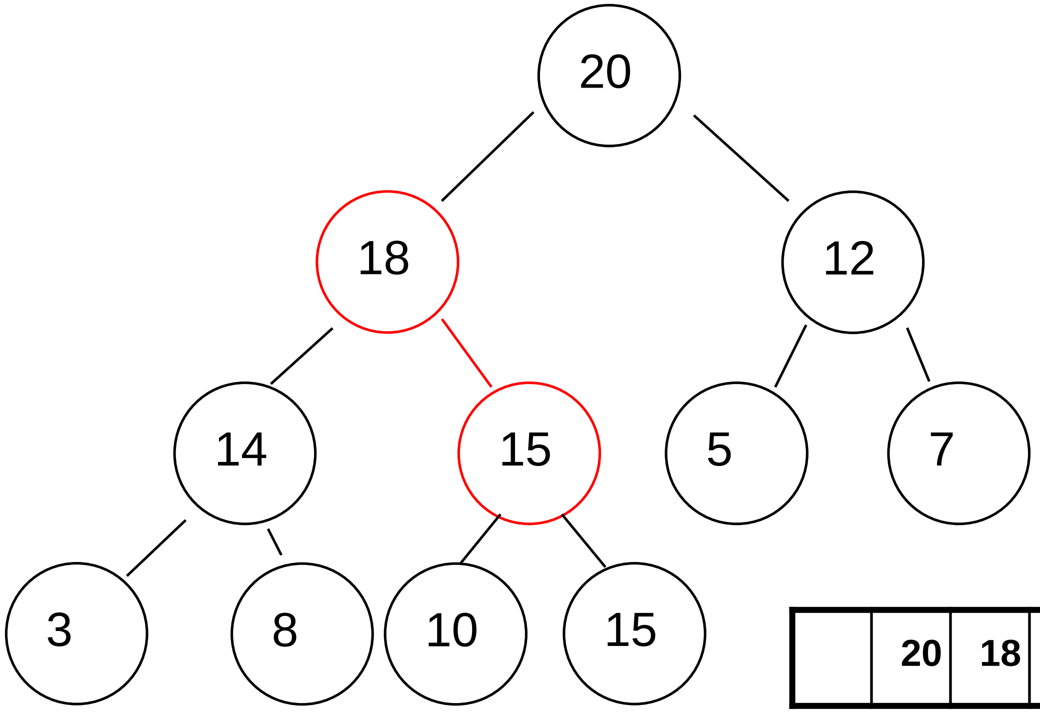
Assume we want to add element 18

```
self.length = 11
self.the_array =
```

	20	**15**	**12**	**14**	**18**	**5**	**7**	**3**	**8**	**10**	**15**		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def rise(self, k: int) -> None:
    while k > 1 and self.the_array[k] > self.the_array[k//2]:
        self.swap(k, k//2)
        k = k//2

def add(self, element: T) -> bool:
    has_space_left = not self.is_full()
    if has_space_left:
        self.length += 1
        self.the_array[self.length] = element
        self.rise(self.length)
    return has_space_left
```



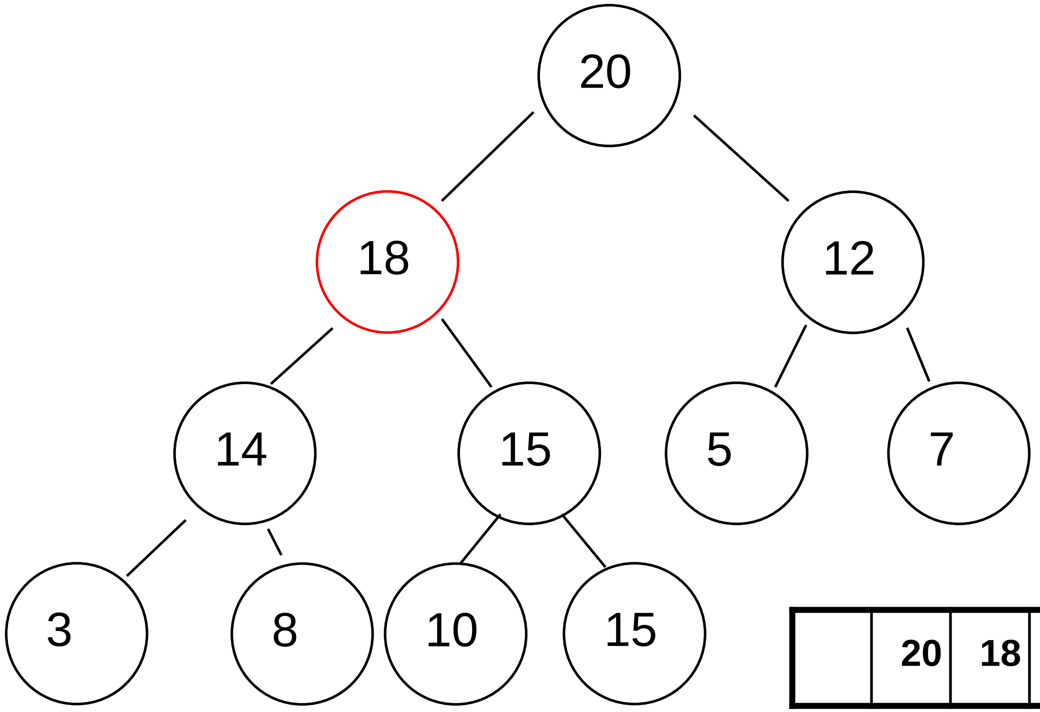
Assume we want to add element 18

```
self.length = 11
self.the_array =
```

	20	**18**	**12**	**14**	**15**	**5**	**7**	**3**	**8**	**10**	**15**		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def rise(self, k: int) -> None:
    while k > 1 and self.the_array[k] > self.the_array[k//2]:
        self.swap(k, k//2)
        k = k//2

def add(self, element: T) -> bool:
    has_space_left = not self.is_full()
    if has_space_left:
        self.length += 1
        self.the_array[self.length] = element
        self.rise(self.length)
    return has_space_left
```



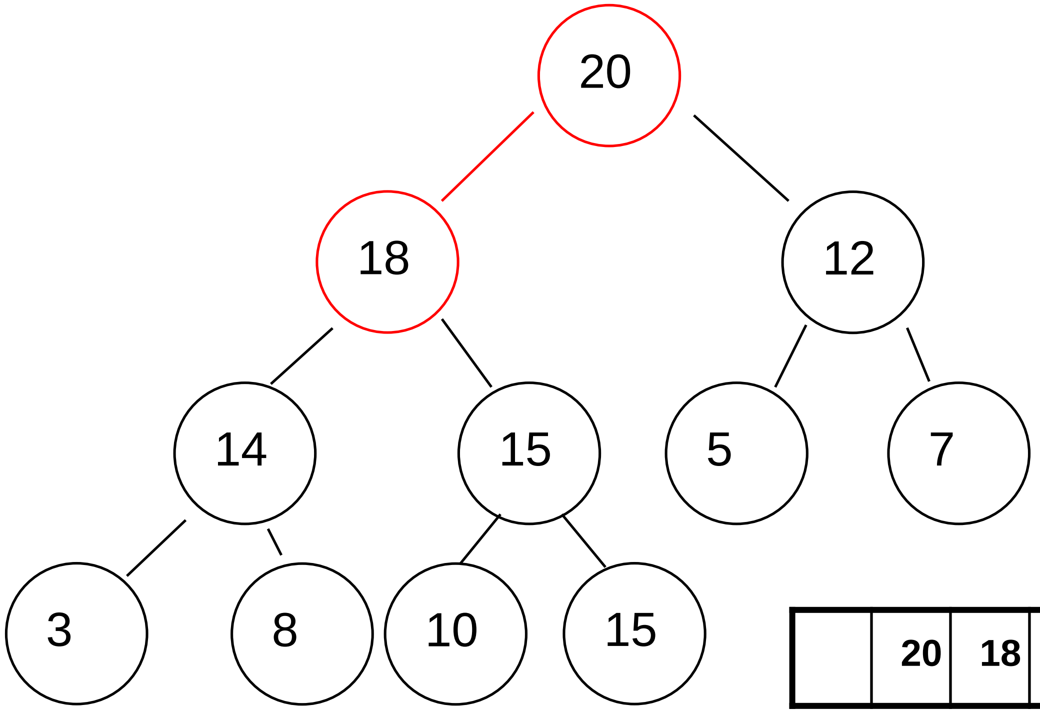
Assume we want to
add element 18

```
self.length = 11
self.the_array =
```

	20	18	12	14	15	5	7	3	8	10	15		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def rise(self, k: int) -> None:
    while k > 1 and self.the_array[k] > self.the_array[k//2]:
        self.swap(k, k//2)
        k = k//2

def add(self, element: T) -> bool:
    has_space_left = not self.is_full()
    if has_space_left:
        self.length += 1
        self.the_array[self.length] = element
        self.rise(self.length)
    return has_space_left
```



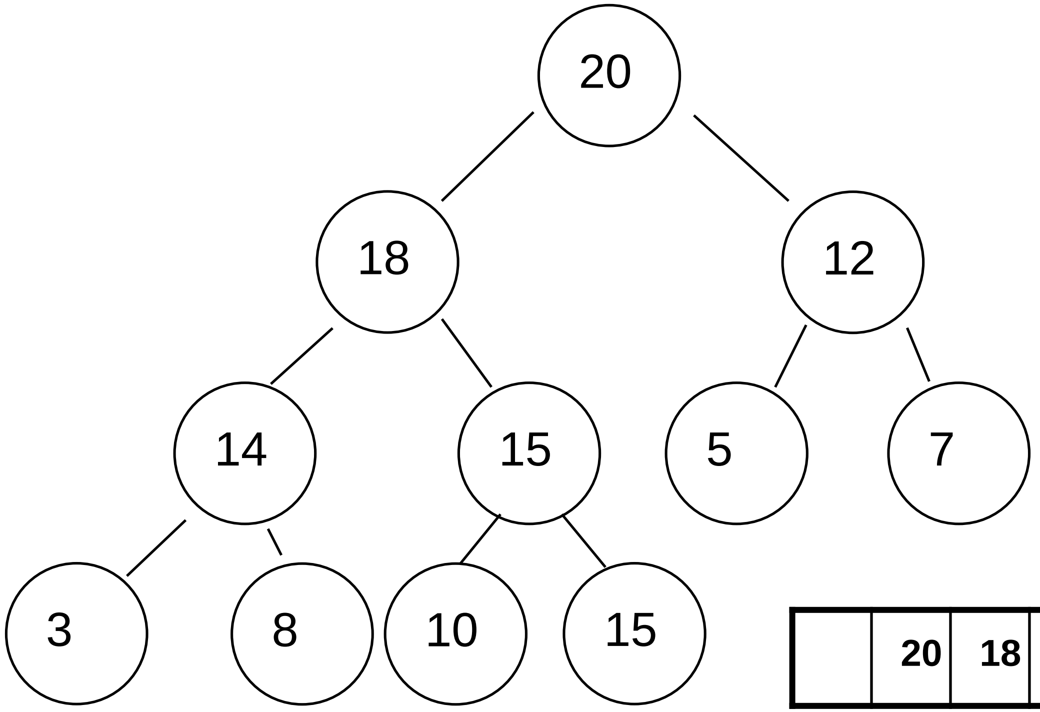
Assume we want to
add element 18

```
self.length = 11
self.the_array =
```

	20	**18**	**12**	**14**	**15**	**5**	**7**	**3**	**8**	**10**	**15**		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def rise(self, k: int) -> None:
    while k > 1 and self.the_array[k] > self.the_array[k//2]:
        self.swap(k, k//2)
        k = k//2

def add(self, element: T) -> bool:
    has_space_left = not self.is_full()
    if has_space_left:
        self.length += 1
        self.the_array[self.length] = element
        self.rise(self.length)
    return has_space_left
```



Assume we want to
add element 18

```
self.length = 11
self.the_array =
```

	20	**18**	**12**	**14**	**15**	**5**	**7**	**3**	**8**	**10**	**15**		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def rise(self, k: int) -> None:
    while k > 1 and self.the_array[k] > self.the_array[k//2]:
        self.swap(k, k//2)
        k = k//2

def add(self, element: T) -> bool:
    has_space_left = not self.is_full()
    if has_space_left:
        self.length += 1
        self.the_array[self.length] = element
        self.rise(self.length)
    return has_space_left
```

An alternative implementation for `add`

```
""" Rise element at index k to its correct position
:pre: 1<= k <= self.length """
def rise(self, k: int, element: T) -> int:
    while k > 1 and element > self.the_array[k//2]:
        self.the_array[k] = self.the_array[k//2]
        k = k//2
    return k

def add(self, element: T) -> bool:
    has_space_left = not self.is_full()
    if has_space_left:
        self.length += 1
        self.the_array[self.rise(self.length, element)] = element
    return has_space_left
```

Shuffles parents down

Returns the position of the “hole”

This is faster, but less clear

No swaps, just shuffles parents down to make a hole for the new element, and then put it in the hole

Another alternative for `add`

```
def add(self, element: T) -> bool:
    has_space_left = not self.is_full()

    if has_space_left:
        self.length += 1
        k = self.length
        while k > 1 and element > self.the_array[k//2]:
            self.the_array[k] = self.the_array[k//2]
            k = k//2

        self.the_array[k] = element

    return has_space_left
```

If we eliminate the modularisation (i.e., call to `rise`)

This version is the fastest, but least maintainable

Eliminating **`rise`** is not a good idea: what if the priority of a node has a sudden increase?

Complexity of `add`

- **Start with the first version and assume there is space**
- **Operations:**

1. put new element at bottom of heap
2. while the heap-order is broken
   a. swap new element with parent

O(1)

OCompare

O(1)

How many times
can this loop iterate?

Complexity of `add`

Is the depth of a heap always ≈ log N?

Yes! As it is complete

- **Loop 2 can iterate at most Depth times ≈ log N**
  - after Depth iterations, the new element is at the root
- **Best case:**
  - O(1) + OCompare when the element is smaller or equal than its parent, which means OCompare.
- **Worst case:**
  - O(1) + O(logN)*O(1)*OCompare when the element rises all the way to the top, which means O(logN)*OCompare

- **Same for all versions?**
  - Yes: some do less copies (less swaps), but have the same O(1)

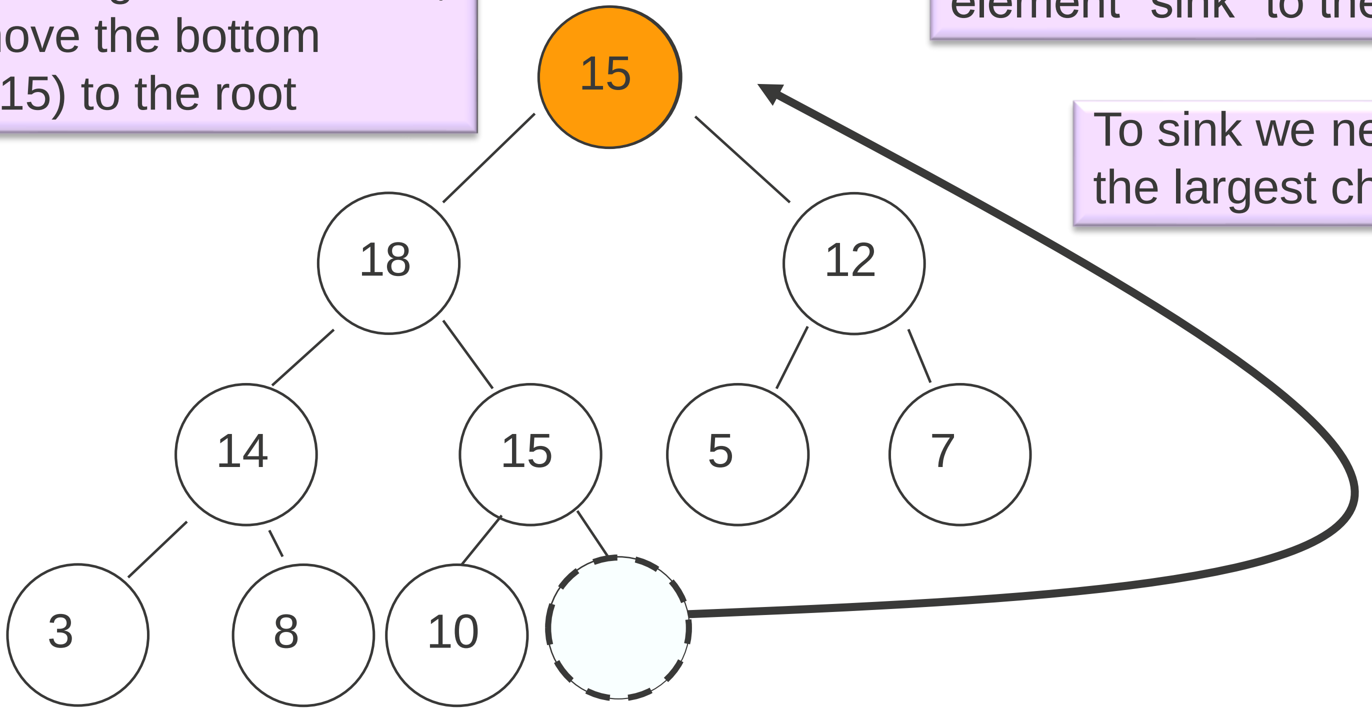
Heap Implementation
Get max

Recall: getting the max element

When removing the maximum, we first move the bottom element (15) to the root

We then need to make this element “sink” to the right place

To sink we need the largest child



Algorithm for `get_max`

1. **swap root element with bottom-right element**
2. **remove that element**
3. **while the heap-order is broken**
   - sink: swap the out-of-place element with its largest child

A concrete implementation

```
""" Returns the index of the largest child of k.
pre: 2*k <= self.length (at least one child) """
def largest_child(self, k: int) -> int:
    if self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

""" Make the element at index k sink to the correct position. """
def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```

Left greater than right

Somewhere in here lies a subtle error

k has at least one child

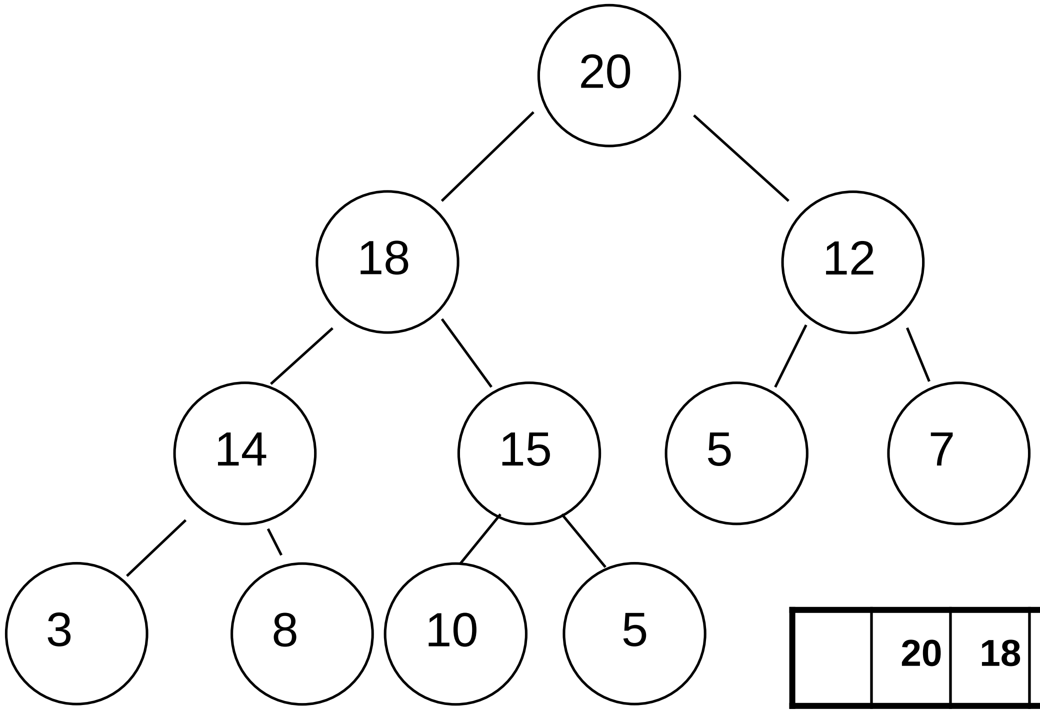
Element >= than its largest child

Correct version

```
def largest_child(self, k: int) -> int:
  """ Check also for k having only one child. """
  if  2*k == self.length or
      self.the_array[2*k]> self.the_array[2*k+1]:
    return 2*k
  else:
    return 2*k+1
```

Aside: subtle errors

- **Errors like that are very easy to make, and hard to spot.**
- **Your armour against them includes:**
  - Thorough testing
  - Code review
  - Proofs of correctness



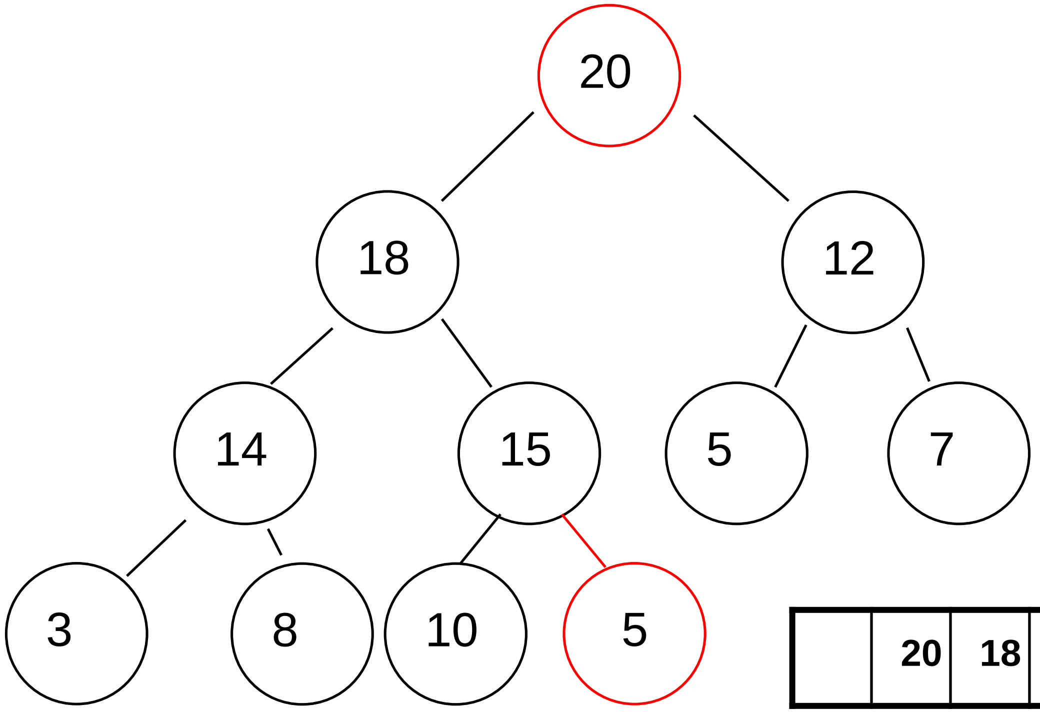
Assume we want to get the max element

```
self.length = 11
self.the_array =
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	20	**18**	**12**	**14**	**15**	**5**	**7**	**3**	**8**	**10**	**5**		

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

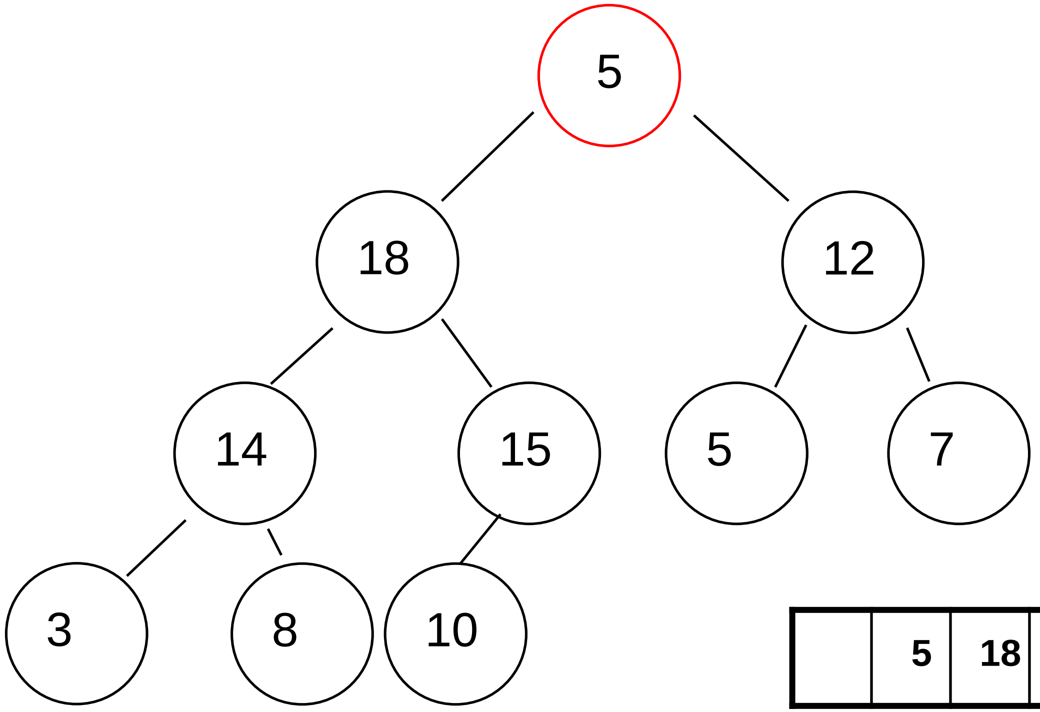
`self.length = 11`

`self.the_array =`

	20	**18**	**12**	**14**	**15**	**5**	**7**	**3**	**8**	**10**	**5**		
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

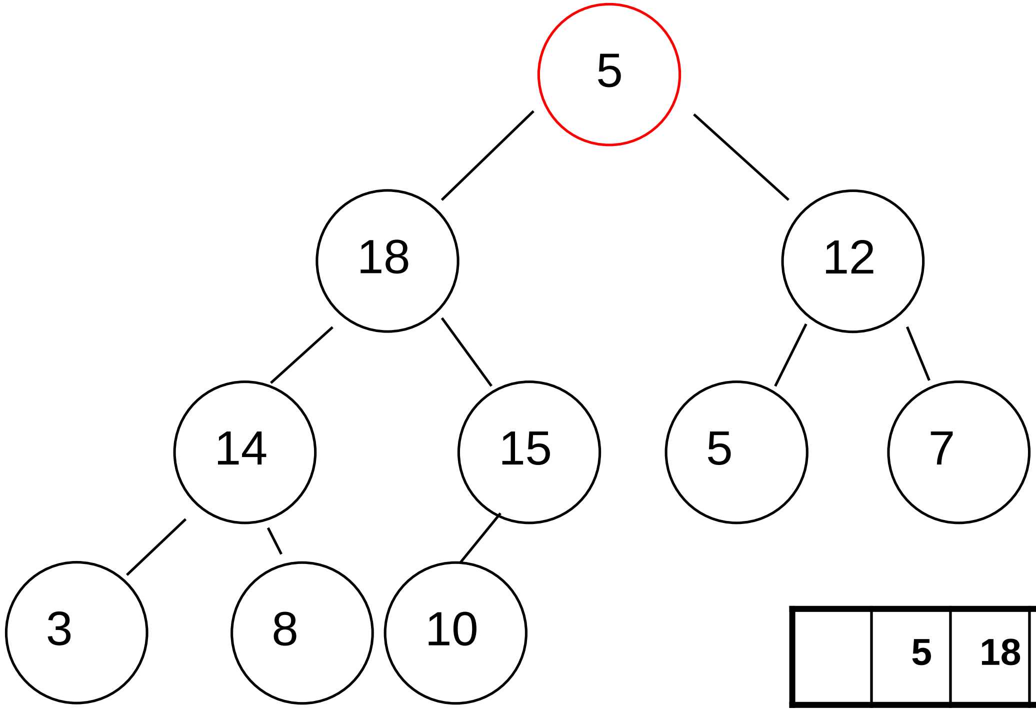
`self.length = 10`

`self.the_array =`

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	5	18	12	14	15	5	7	3	8	10			

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

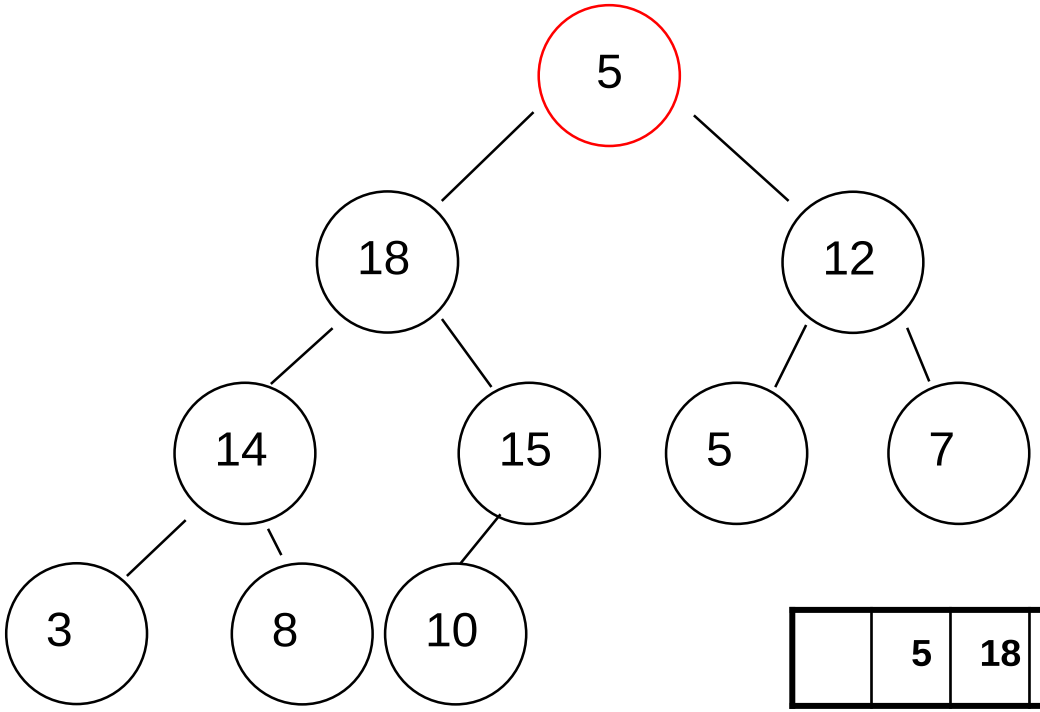
self.length = 10

self.the_array =

	5	18	12	14	15	5	7	3	8	10			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

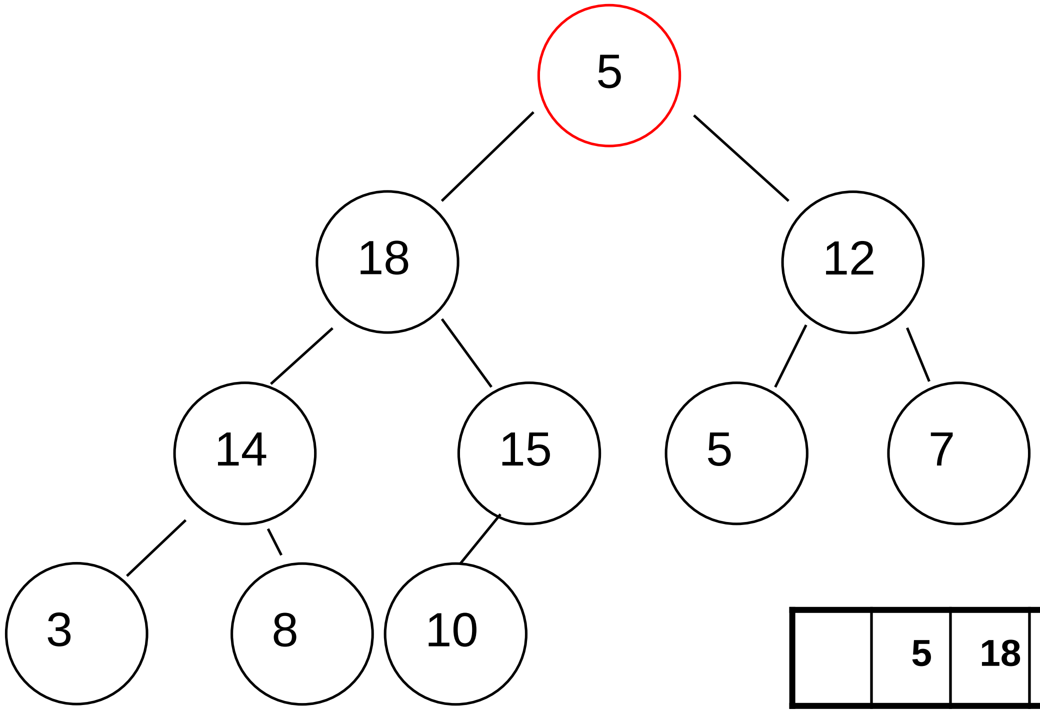
`self.length = 10`

`self.the_array =`

	5	18	12	14	15	5	7	3	8	10			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

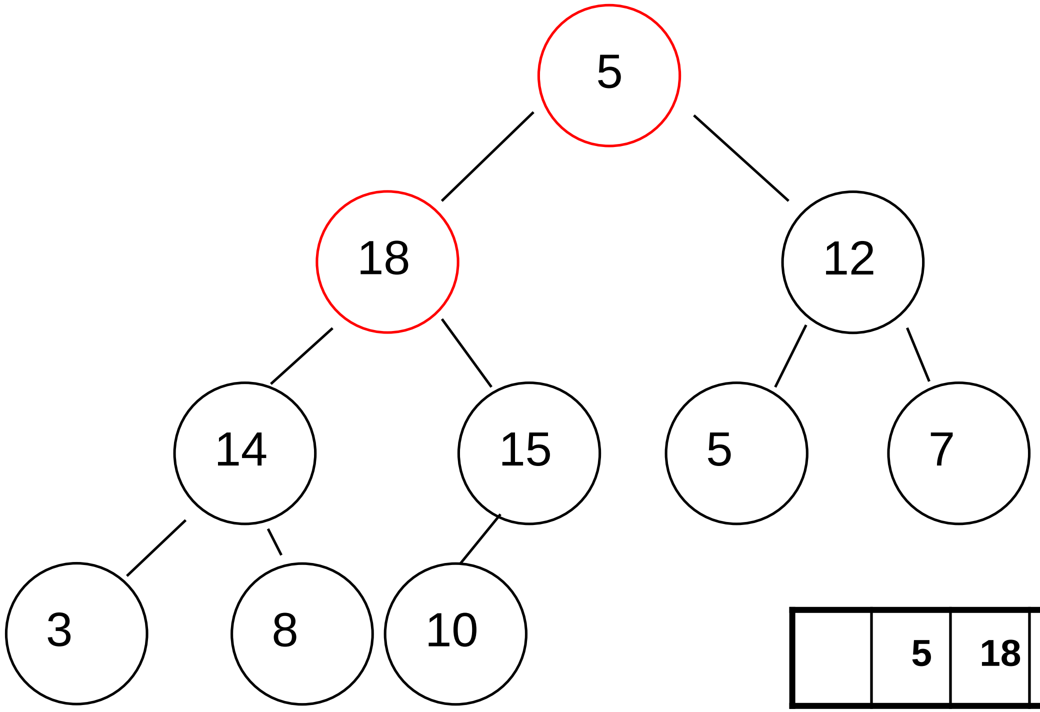
`self.length = 10`

`self.the_array =`

	5	18	12	14	15	5	7	3	8	10			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

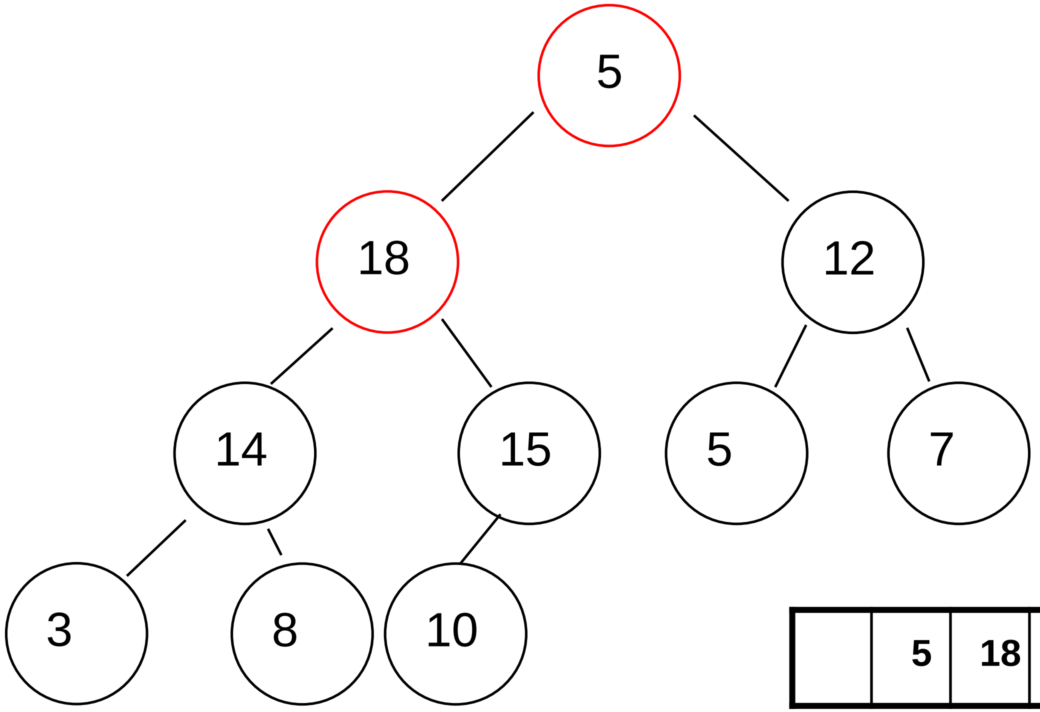
`self.length = 10`

`self.the_array =`

	5	18	12	14	15	5	7	3	8	10			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

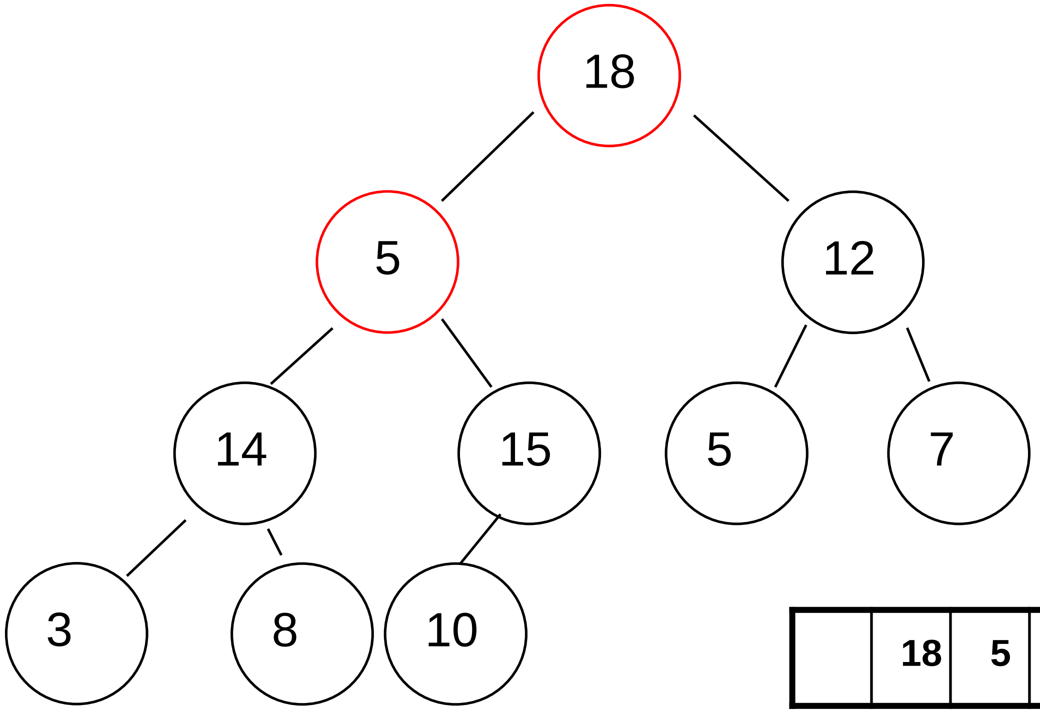
`self.length = 10`

`self.the_array =`

	5	18	12	14	15	5	7	3	8	10			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

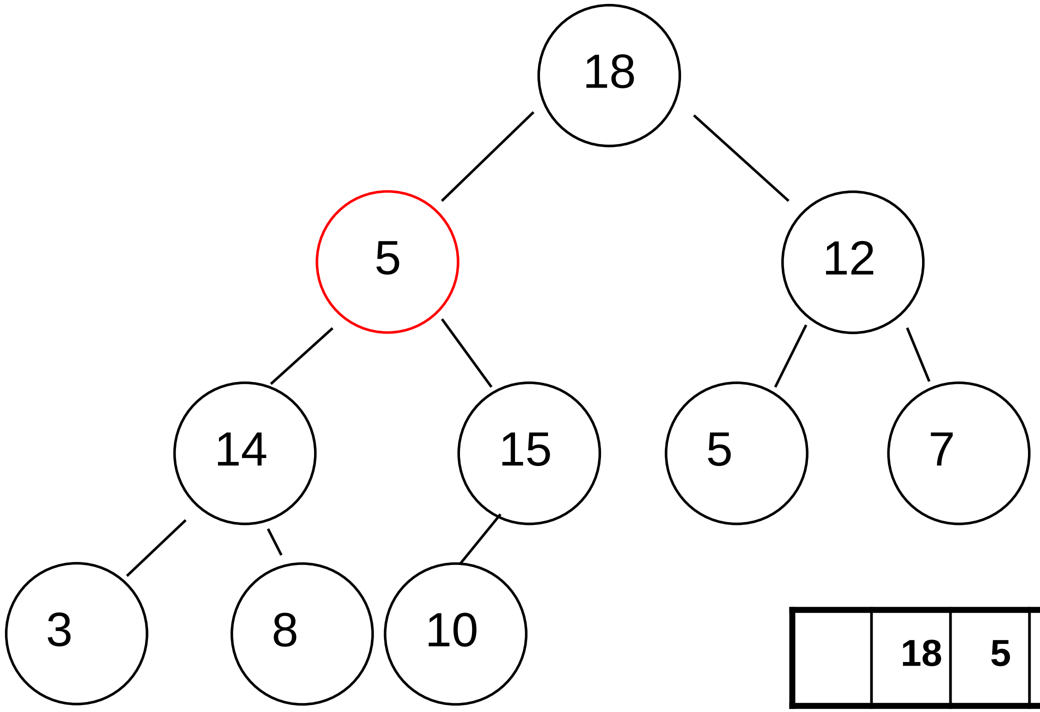
self.length = 10

self.the_array =

	18	5	12	14	15	5	7	3	8	10			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

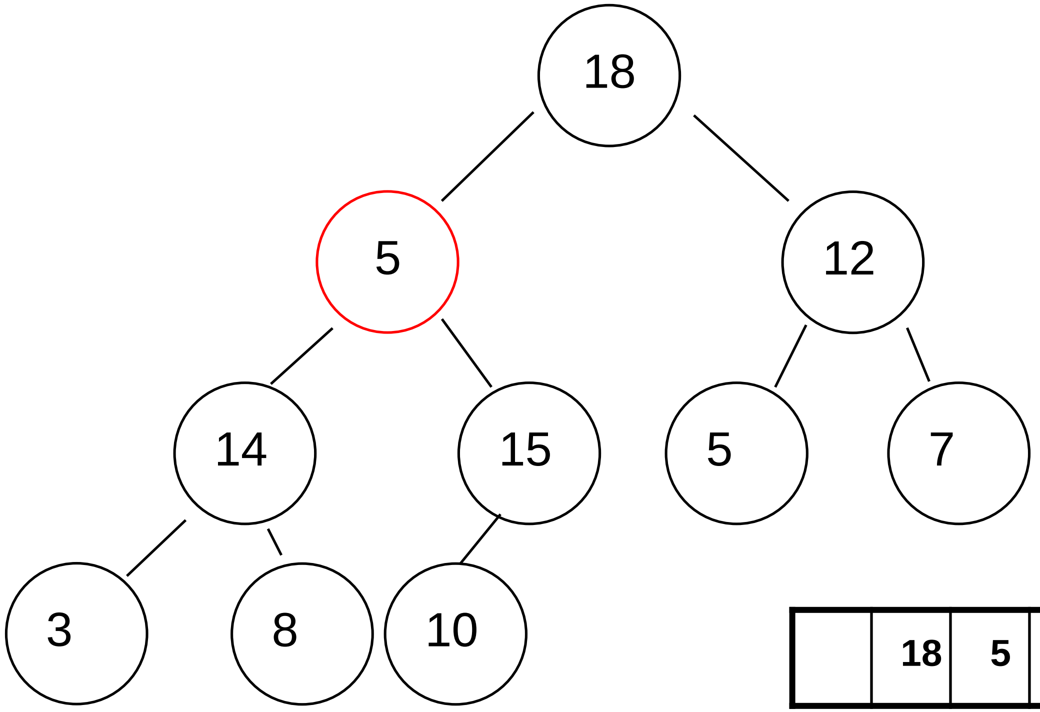
self.length = 10

self.the_array =

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	18	5	12	14	15	5	7	3	8	10			

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

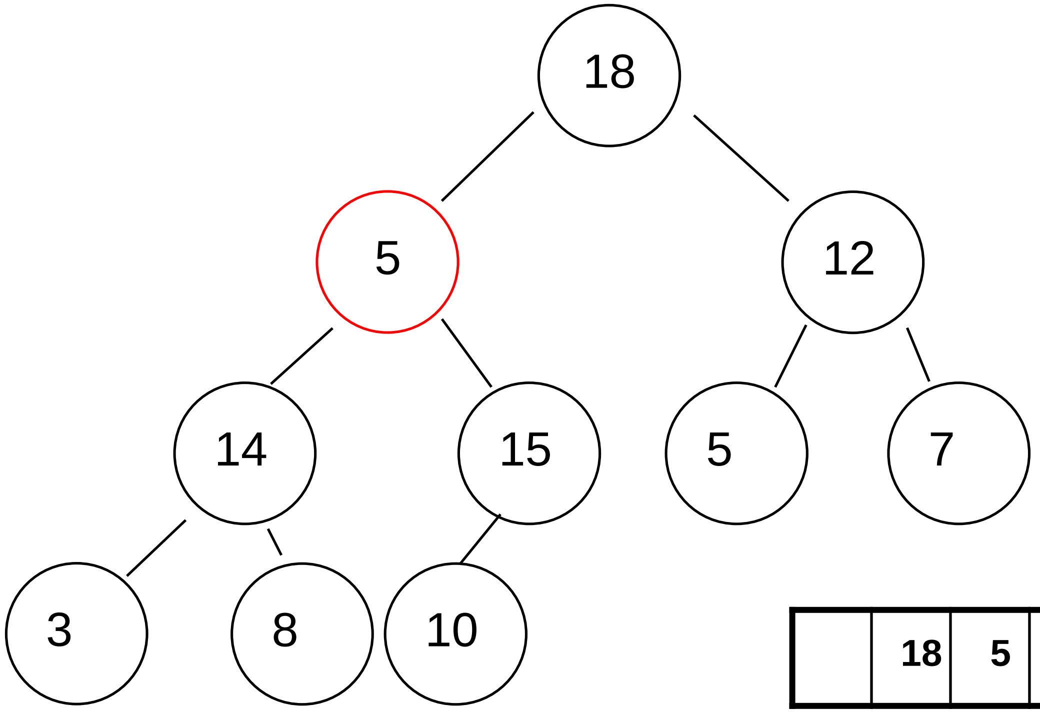
Then, start sinking from the root

```
self.length = 10
self.the_array =
```

	18	5	12	14	15	5	7	3	8	10			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

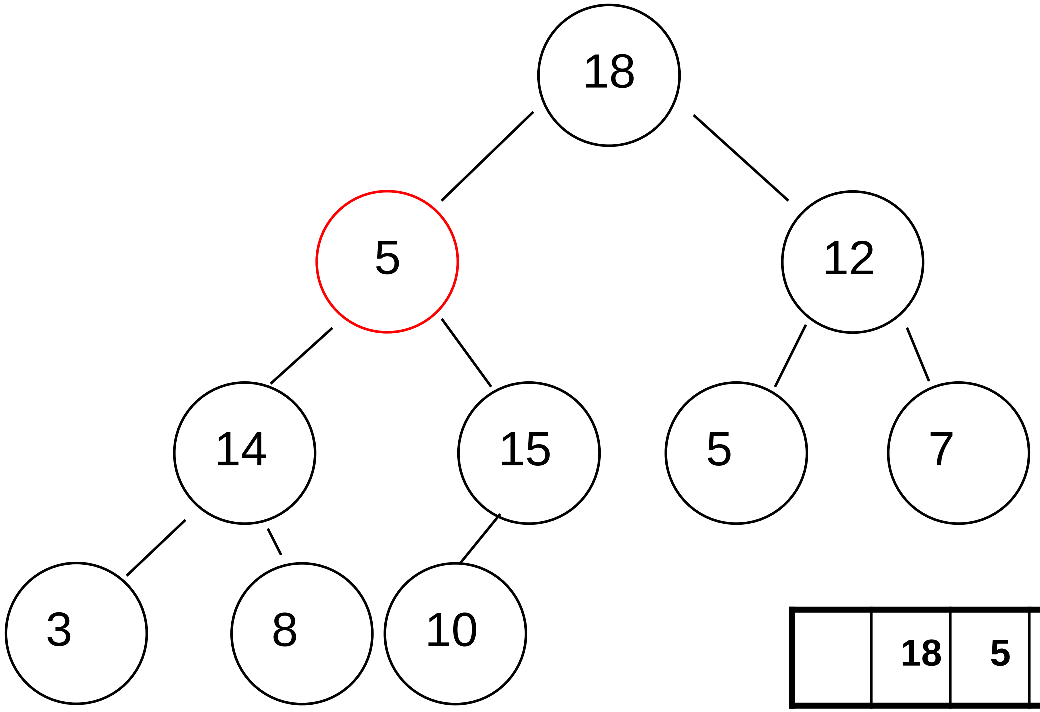
self.length = 10

self.the_array =

	18	5	12	14	15	5	7	3	8	10			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

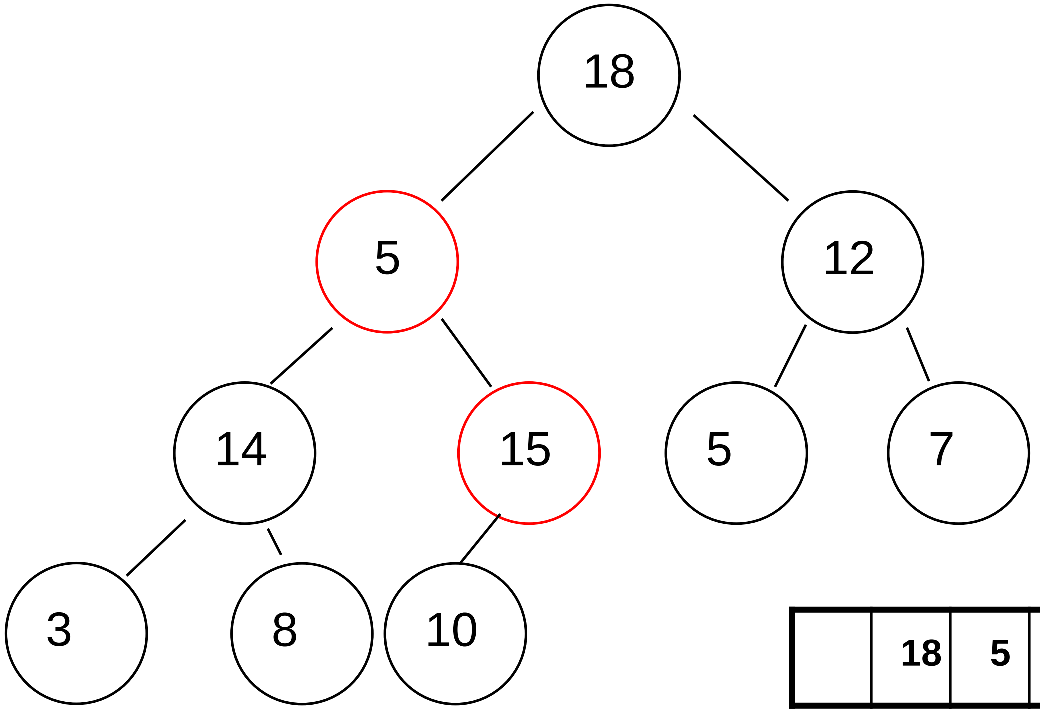
self.length = 10

self.the_array =

	18	5	12	14	15	5	7	3	8	10			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

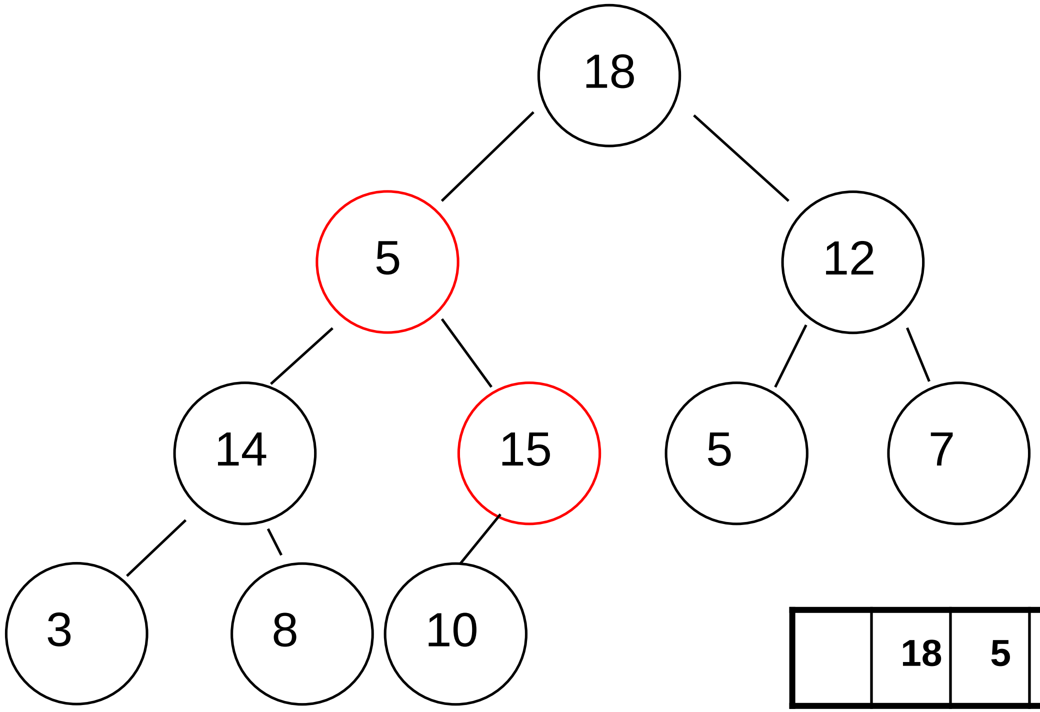
`self.length = 10`

`self.the_array =`

	18	5	12	14	15	5	7	3	8	10			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

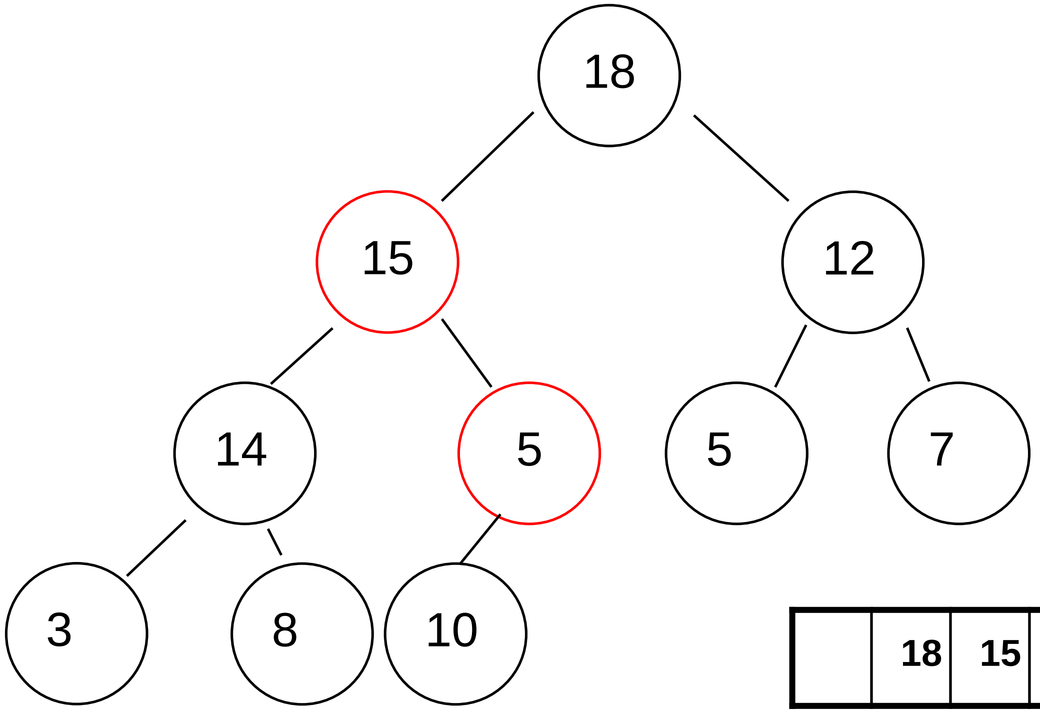
`self.length = 10`

`self.the_array =`

	18	5	12	14	15	5	7	3	8	10			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

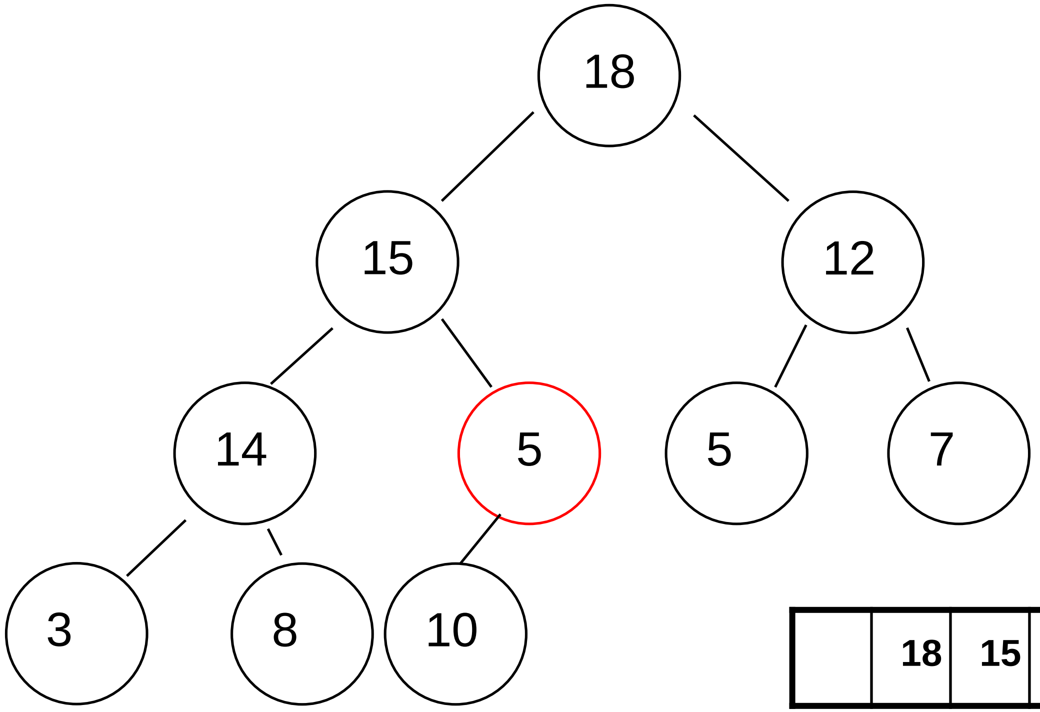
`self.length = 10`

`self.the_array =`

	18	15	12	14	5	5	7	3	8	10			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

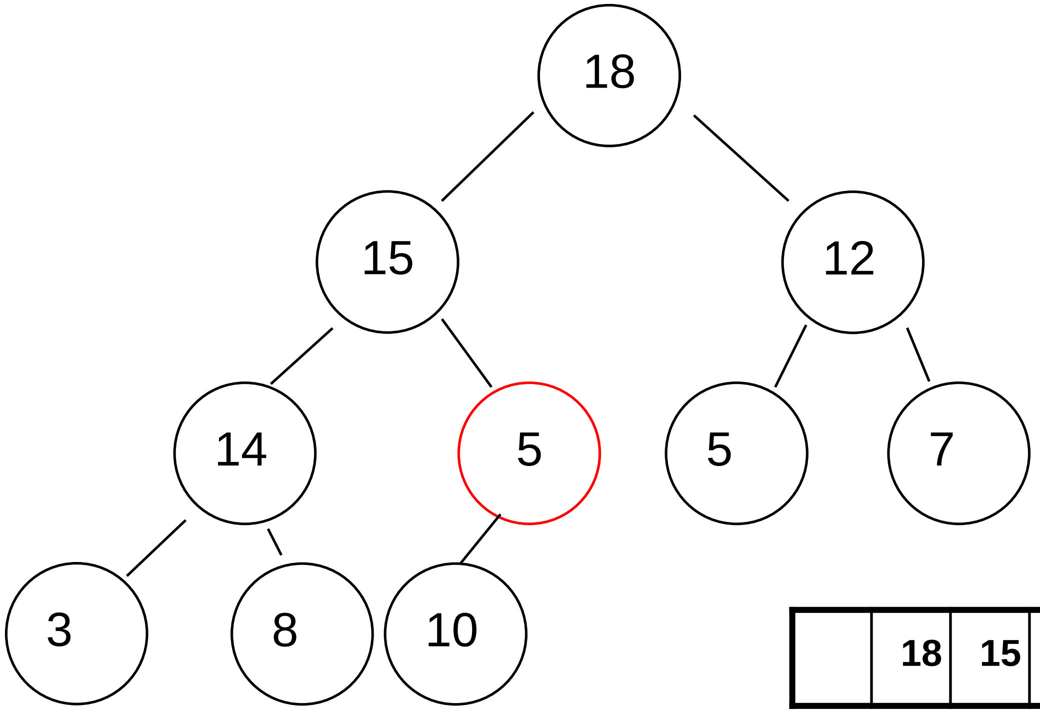
self.length = 10

self.the_array =

	18	15	12	14	5	5	7	3	8	10			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

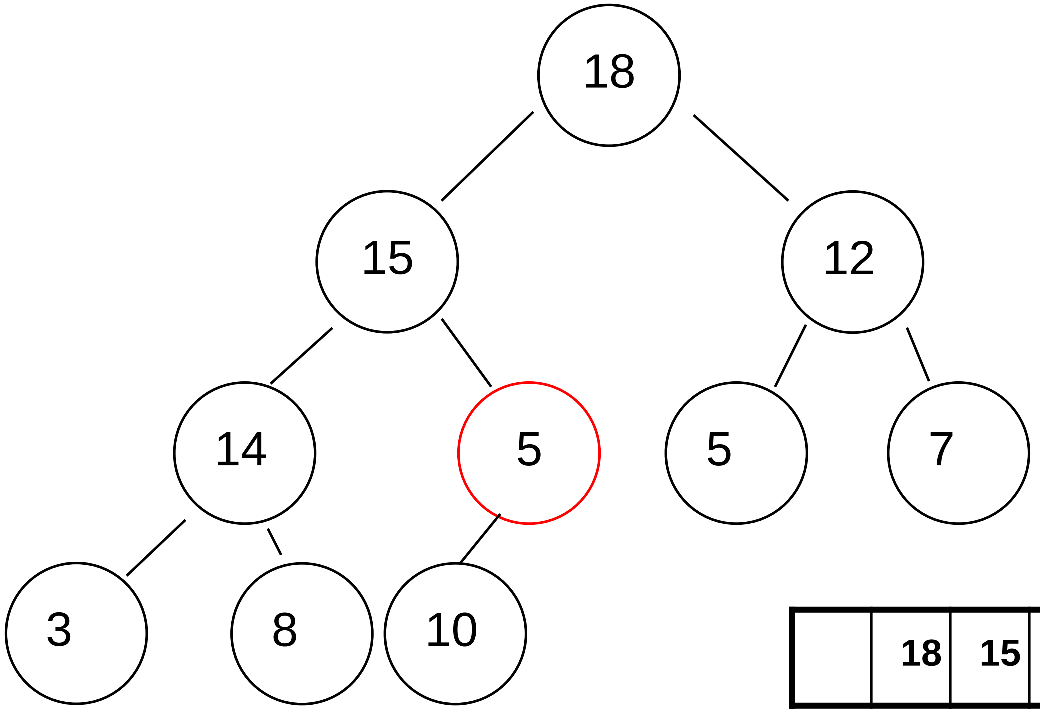
Then, start sinking from the root

```
self.length = 10
self.the_array =
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	18	15	12	14	5	5	7	3	8	10			

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

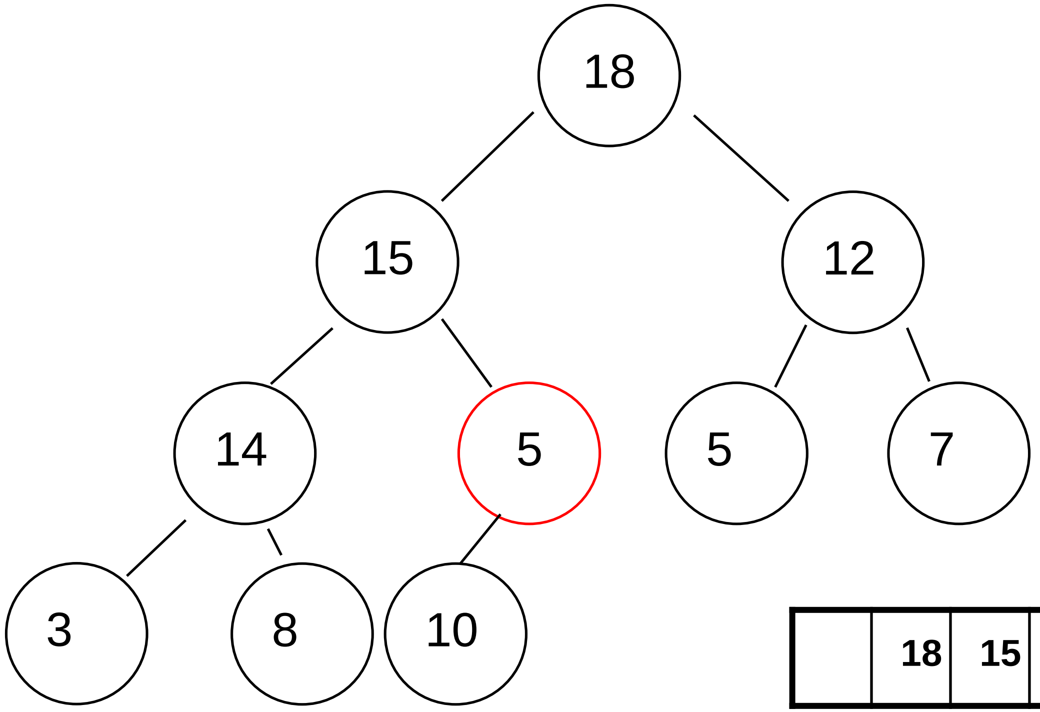
`self.length = 10`

`self.the_array =`

	18	15	12	14	5	5	7	3	8	10			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

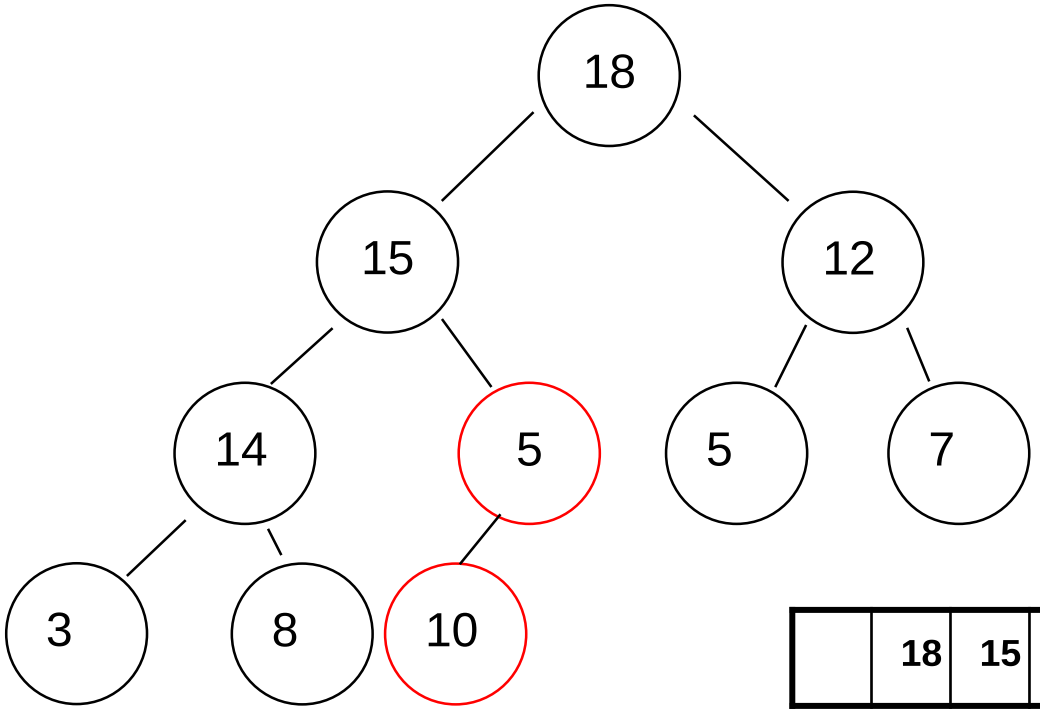
`self.length = 10`

`self.the_array =`

	18	15	12	14	5	5	7	3	8	10			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

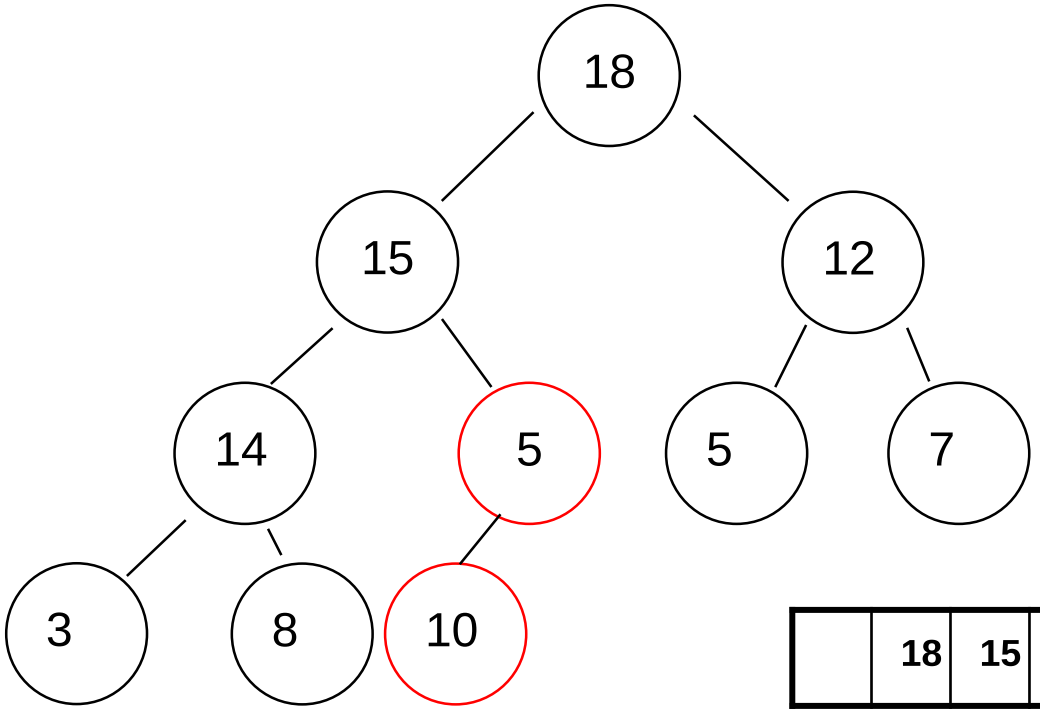
`self.length = 10`

`self.the_array =`

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	18	15	12	14	5	5	7	3	8	10			

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

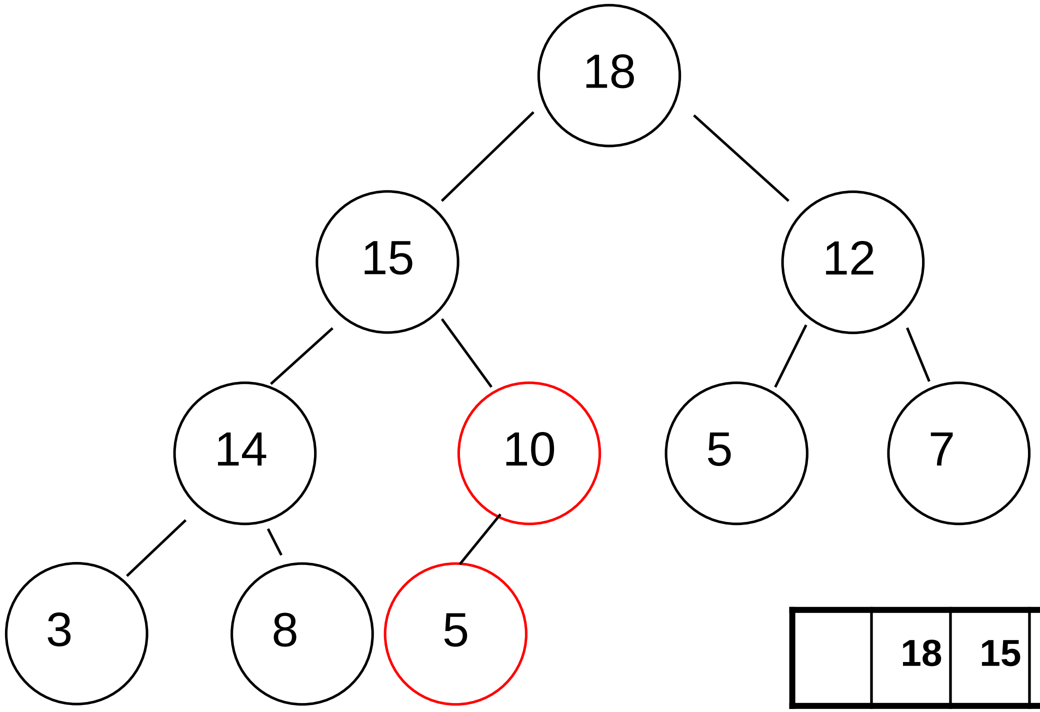
`self.length = 10`

`self.the_array =`

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	18	15	12	14	5	5	7	3	8	10			

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

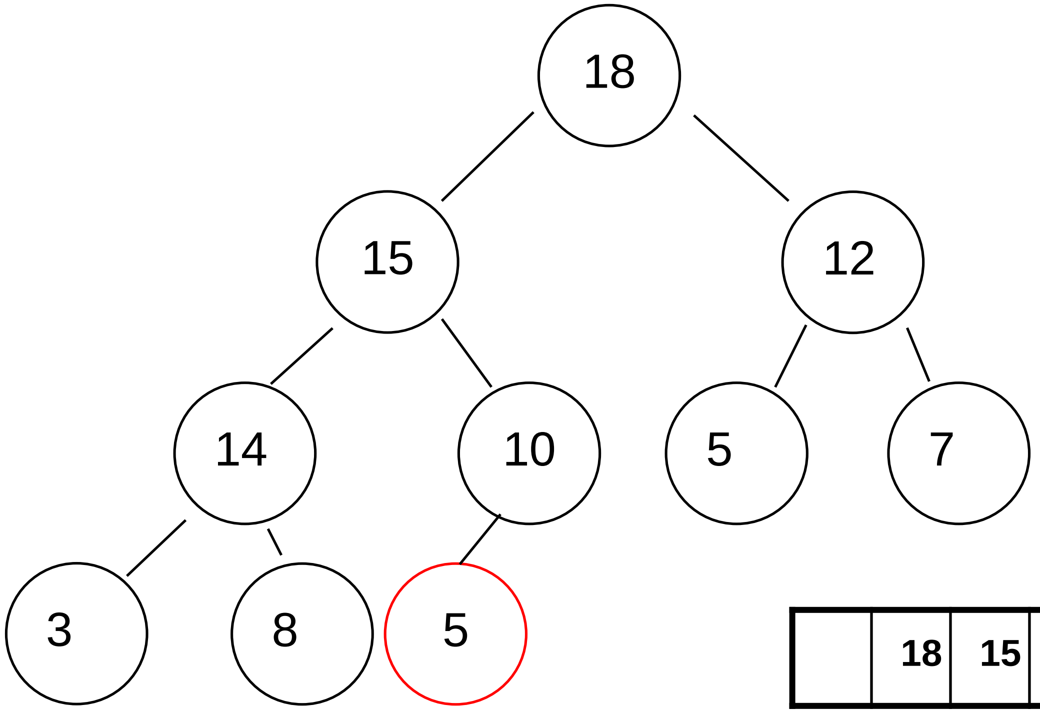
self.length = 10

self.the_array =

	18	15	12	14	10	5	7	3	8	5			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

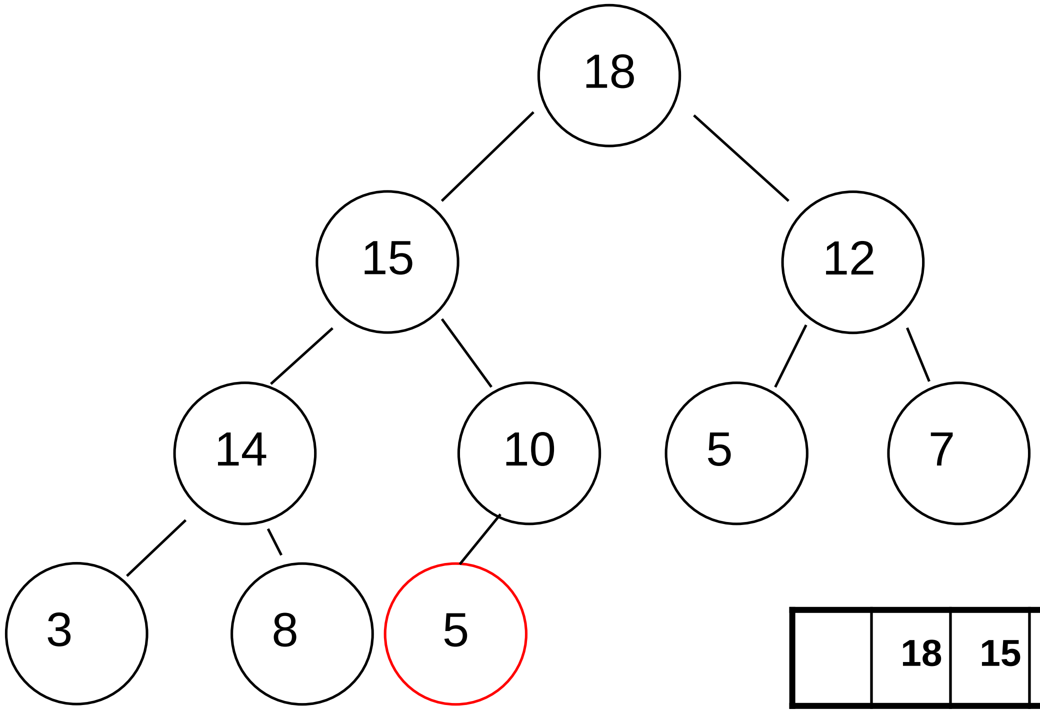
`self.length = 10`

`self.the_array =`

	18	15	12	14	10	5	7	3	8	5			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```



Assume we want to get the max element

First delete the last node and swap its item with the root

Then, start sinking from the root

self.length = 10

self.the_array =

	18	**15**	**12**	**14**	**10**	**5**	**7**	**3**	**8**	**5**			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
def largest_child(self, k: int) -> int:
    if 2*k == self.length or self.the_array[2*k] > self.the_array[2*k+1]:
        return 2*k
    else:
        return 2*k+1

def sink(self, k: int) -> None:
    while 2*k <= self.length:
        child = self.largest_child(k)
        if self.the_array[k] >= self.the_array[child]:
            break
        self.swap(child,k)
        k = child
```

Complexity of `getMax`

- **Operations:**

  O(1)

  1. swap root element with bottom-right element
  2. remove that element O(1)
  3. while the heap-order is broken OCompare
     a. swap the out-of-place element with its largest child

  O(1)

How many times
can this loop iterate?

Complexity of `get_max`

- **Loop 3 can iterate only Depth times ≈ log N**
  - After Depth iterations, the new element is at the bottom of the tree
- **Best case:**
  - O(1) + O(1) + OCompare when the element is greater or equal to one of its children (cannot be smaller), which means OCompare
- **Worst case:**
  - O(1)+O(1) +O(logN)*O(1)*OCompare when the element sinks all the way to the bottom, which means O(logN)*OCompare

Thus, both add and get_max are O(log N)*Ocompare.
Better than O(N) – which was the case for lists…

Priority Queue Sort

An application of Heaps: “PQueue-sort”

- **Remember:**
  - We are using Heaps to implement Priority Queues
- **With a fast `get_max` we can also sort any list quickly**
  1. Put all elements into a p-queue
  2. Call `get_max` *n* times
- **Gives all the elements in descending order**
- **Possible for any Priority Queue implementation**
  - BUT only worth it if it is fast

“PQueue-sort” properties for a heap impl.

- **Step 1: N `add` operations to the queue**
  - For a heap: O(N*logN) (*comparisons)
- **Step 2: N `get_max` operations**
  - For a heap: O(N*logN) (*comparisons)
- **Total for a heap implementation:**

O(N*logN + N*logN) = O(N*logN)

- **If putting them back into the list is O(N)**
  - We get the same complexity as quick_sort and merge_sort

Properties of "PQueue-sort"

- **Advantages**
  - Works with any implementation of priority queues
- **Disadvantages**
  - Requires O(N) space for the new queue
  - O(N*logN) heap construction

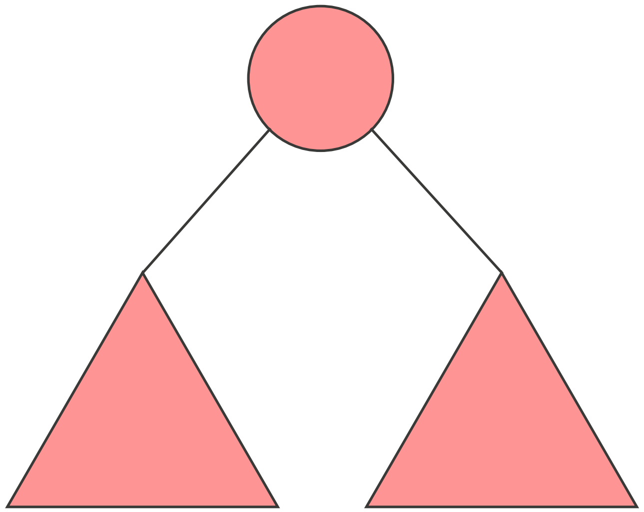
Heaps
Bottom up construction

Cleverer: Bottom-up heap construction

- **We've been constructing the heap one element at a time**
- **After each element is added, we "heapify" the entire heap**
  - O(logN) per element added
- **If we know many elements in advance, we can do it more quickly**

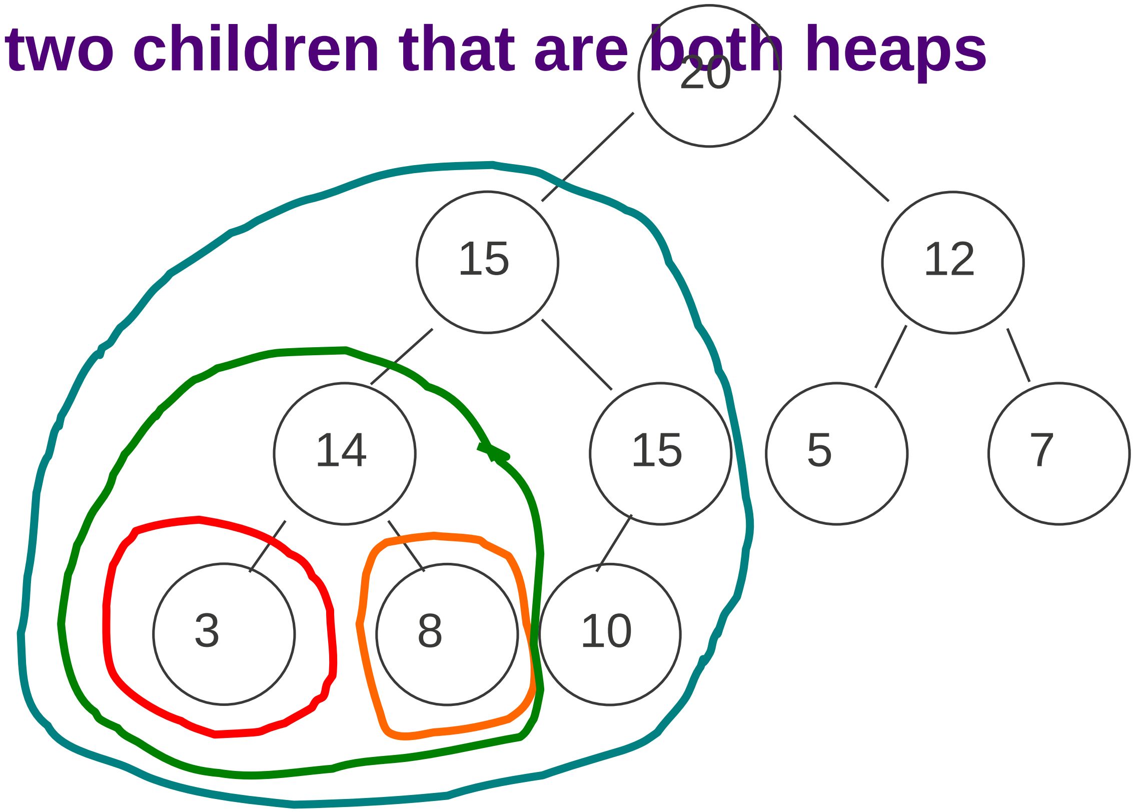
Cleverer: Bottom-up heap construction

- **Recall that a binary search tree is *recursive*: a BST is an element plus two children that are also BSTs**



Cleverer: Bottom-up heap construction

- Remember, heaps are also recursive
- A heap is an element with two children that are both heaps
- Every circle is a heap



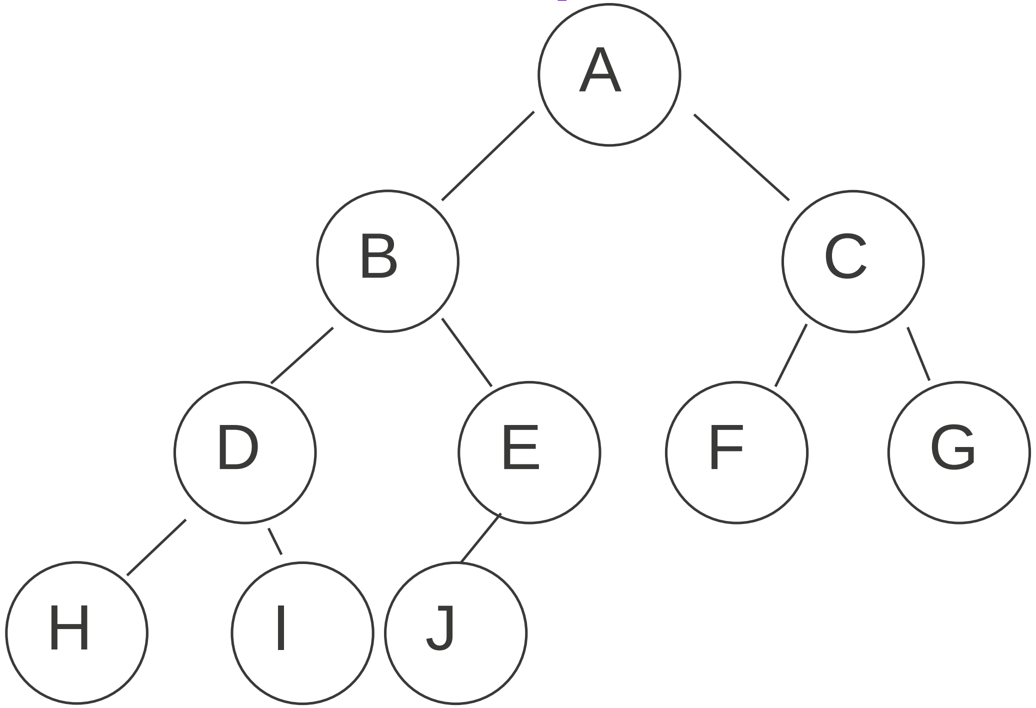
Bottom-up heap construction

- **So, to build a heap, we can start from the bottom and make little heaps**
- **By definition, every leaf node is a heap**
  - So leaves are already done: less work!

- **In a complete binary tree with N elements, how many leaves are there?**

How many leaves?

- **The first non-leave is the parent of the last element**
- **N//2**

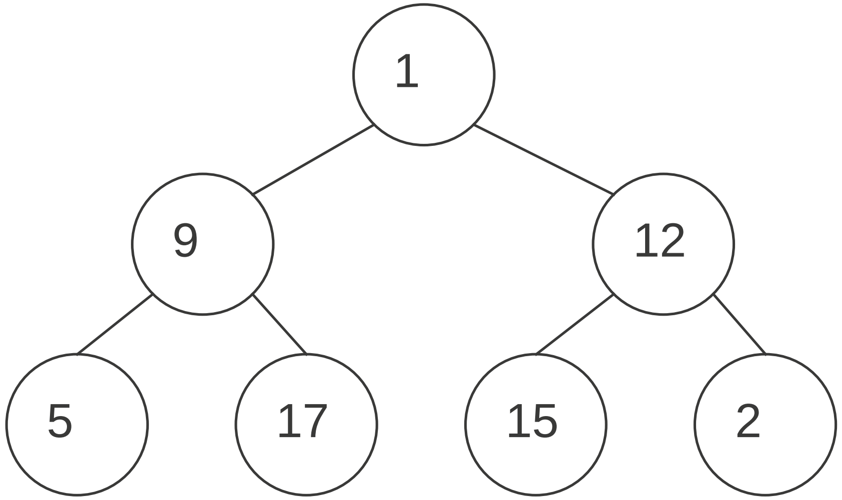


	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Bottom-up heap construction

- **Start with an array in arbitrary order**
- **Build sub-heaps, starting at the bottom right**
- **The leaves are already done, so we start at "12"**
  - `number_elements//2`

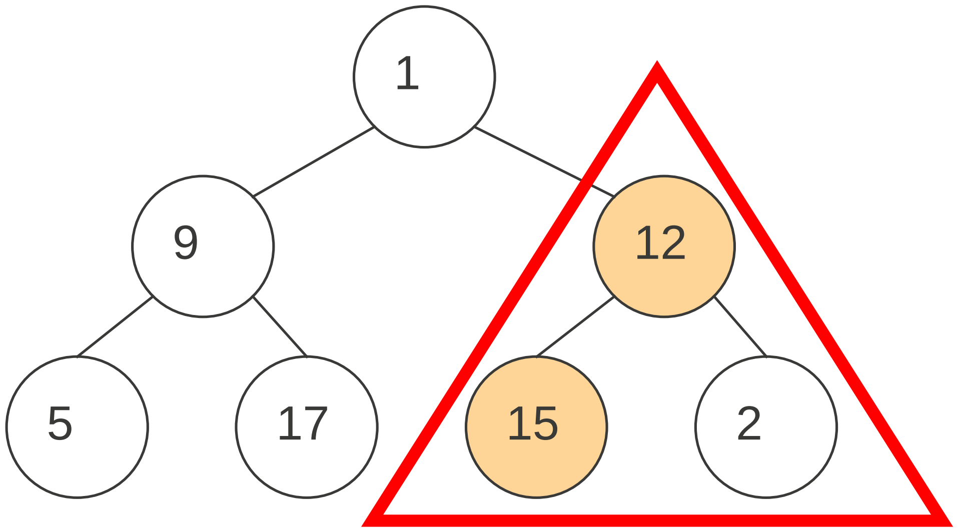
	1	9	12	5	17	15	2
0	1	2	3	4	5	6	7



Bottom-up heap construction

- **It's out of order, so swap with its largest child**

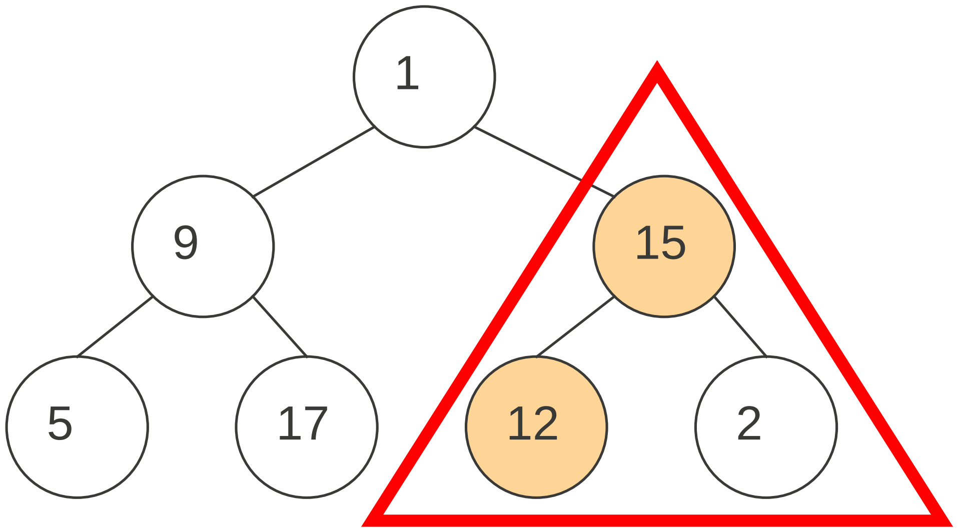
	1	9	12	5	17	15	2
0	1	2	3	4	5	6	7



Bottom-up heap construction

- It’s out of order, so push the root down until it’s correct
- Now it’s a heap

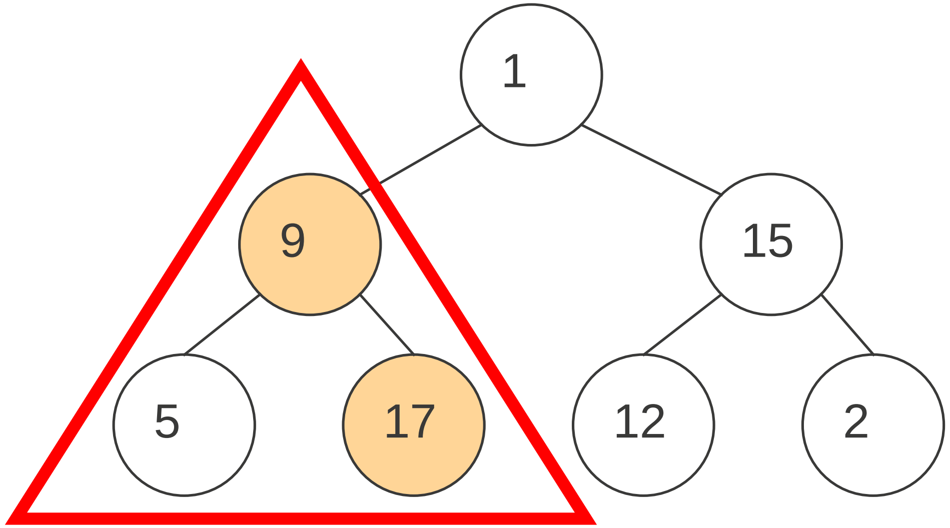
	1	9	15	5	17	12	2
0	1	2	3	4	5	6	7



Bottom-up heap construction

- **Keep going…**

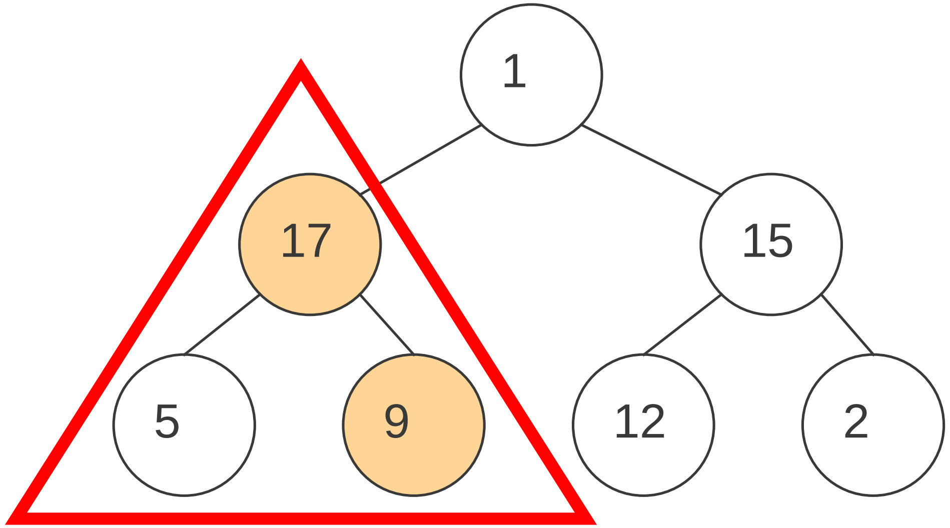
	1	**9**	**15**	**5**	**17**	**12**	**2**
0	1	2	3	4	5	6	7



Bottom-up heap construction

- **Keep going…**

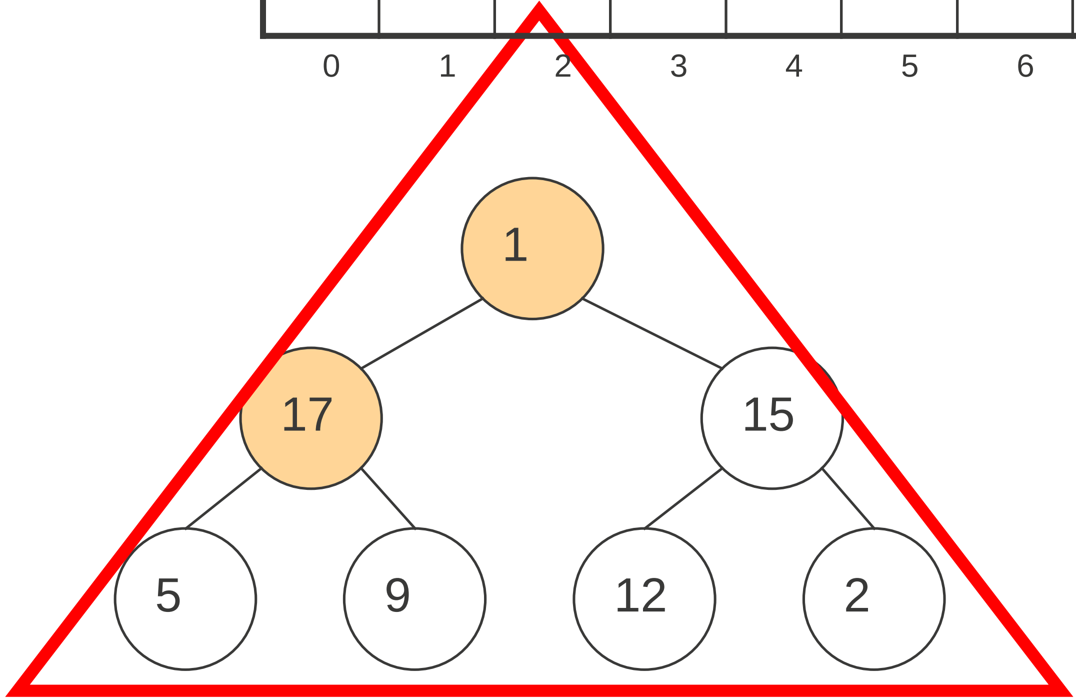
	1	**17**	**15**	**5**	**9**	**12**	**2**
0	1	2	3	4	5	6	7



Bottom-up heap construction

- **Keep going…**

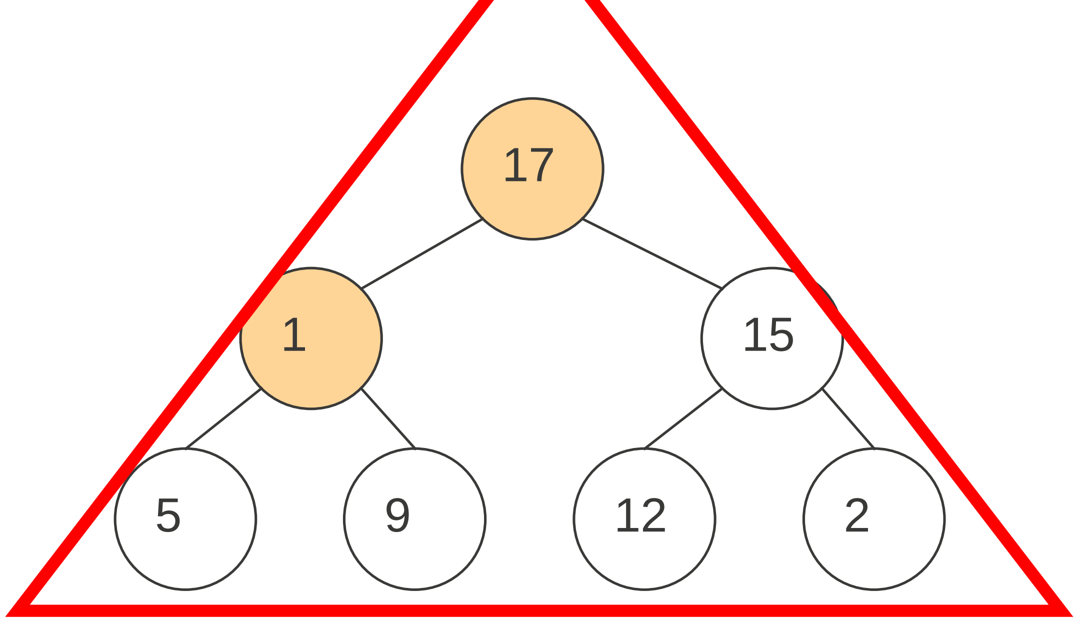
0	1	2	3	4	5	6	7
	1	17	15	5	9	12	2



Bottom-up heap construction

- **Keep going…**

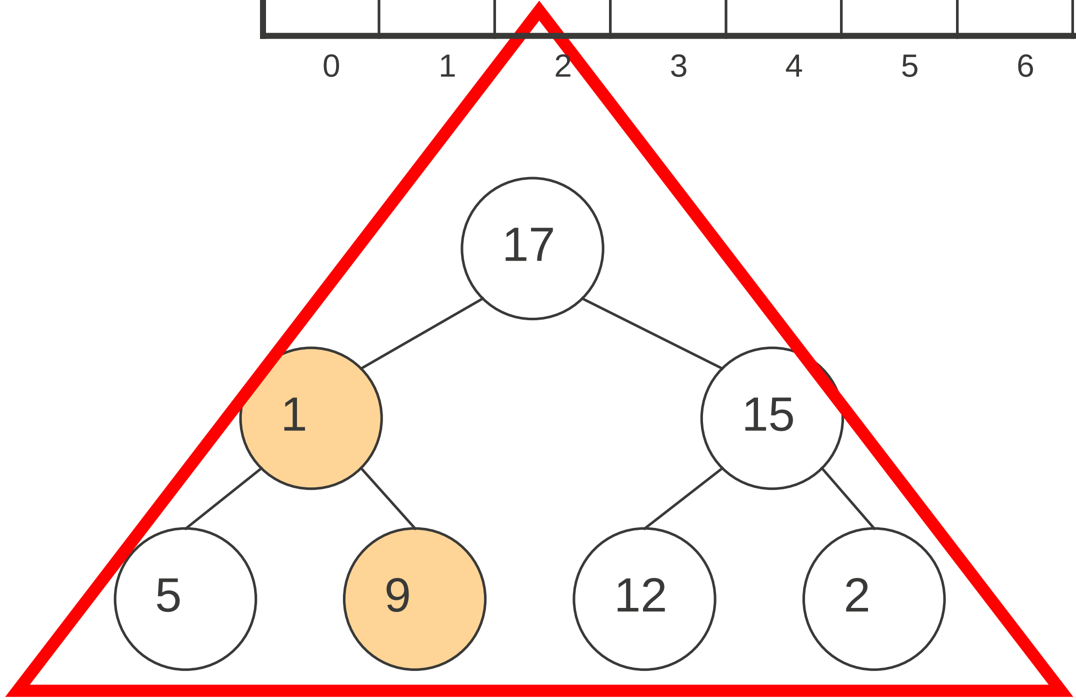
	17	**1**	**15**	**5**	**9**	**12**	**2**
0	1	2	3	4	5	6	7



Bottom-up heap construction

- **Keep going…**

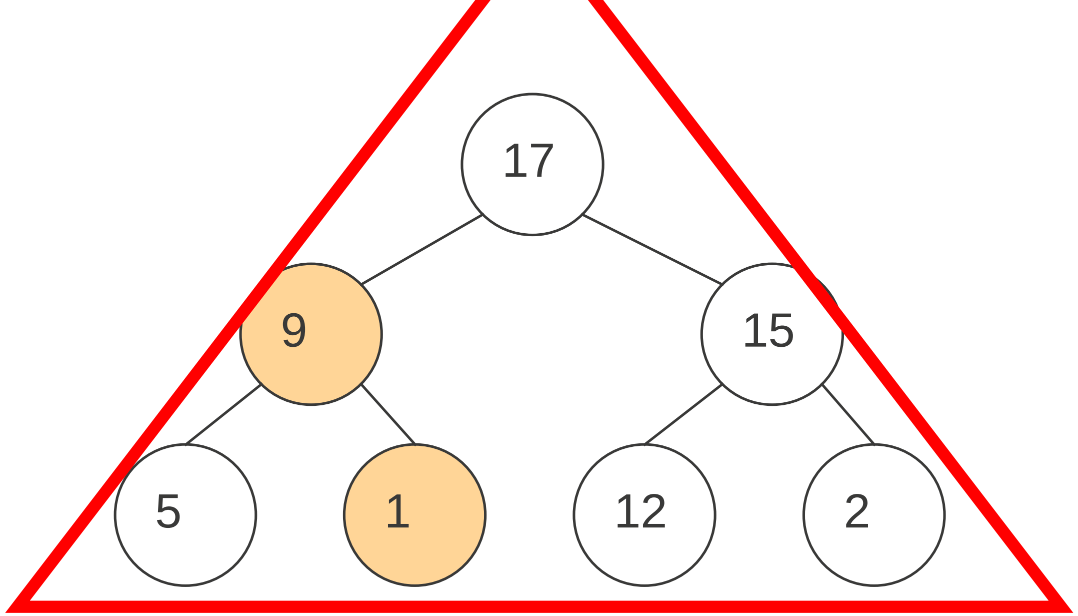
	17	1	15	5	9	12	2
0	1	2	3	4	5	6	7



Bottom-up heap construction

- **Done!**

	17	**9**	**15**	**5**	**1**	**12**	**2**
0	1	2	3	4	5	6	7



Alternative __init__ for known elements

```
""" If elements are known in advance, they are in an_array
Assume that max_size=len(an_array) if given """
def __init__(self, max_size: int, an_array = None: ArrayR[T]) -> None:
    self.the_array = ArrayR(max(self.MIN_CAPACITY, max_capacity) + 1)
    self.length = max_size

    if an_array is not None:
        # copy an_array to self.the_array (shift by 1)
        for i in range(self.length):
            self.the_array[i+1] = an_array[i]

        # heapify every parent
        for i in range(max_size//2,0,-1):
            self.sink(i)
```

Complexity of creating a heap of N elems

- **At worst, how far does each node need to be pushed down?**

- **Its height!**

Complexity of creating a heap of N elems

No. nodes	Height
2^0	h-1
2^1	h-2
2^2	h-3
2^{h-3}	2
2^{h-2}	1
2^{h-1}	0

- **Sum of all heights in heap:**
  - $S = 1(2^{h-2}) + 2(2^{h-3}) + \ldots + (h-3)2^2 + (h-2)2^1 + (h-1)2^0$
- **Simplify**

Complexity of creating a heap of N elems

- **Multiply by two:**
  - $S = 1(2^{h-2}) + 2(2^{h-3}) + \ldots + (h-3)2^2 + (h-2)2^1 + (h-1)2^0$
  - $2S = 1(2^{h-1}) + 2(2^{h-2}) + \ldots + (h-3)2^3 + (h-2)2^2 + (h-1)2^1$
- **Realign:**
  - $2S = 1(2^{h-1}) + 2(2^{h-2}) + \ldots + (h-3)2^3 + (h-2)2^2 + (h-1)2^1$
  - $S = \quad 1(2^{h-2}) + \ldots + (h-4)2^3 + (h-3)2^2 + (h-2)2^1 + (h-1)2^0$
- **Subtract (2S-S):**
  - $S = 1(2^{h-1}) + 1(2^{h-2}) + \ldots + (1)2^3 + (1)2^2 + (1)2^1 - (h-1)2^0$

$$= 2^{h-1} + 2^{h-2} + \ldots + 2^3 + 2^2 + 2^1 - (h-1)2^0$$

$$= 2^{h-1} + 2^{h-2} + \ldots + 2^3 + 2^2 + 2^1 + 2^0 - h$$

$$= 2^h - 1 - h$$

Complexity is O(N) where N is the number of nodes (since N is 2^h-1)

In-place Heap Sort

In-place heap sort

- **Our earlier queue sort requires O(N) extra space for the queue**
- **We can sort an array in-place (O(1) extra space) using heap principles**

In-place heap sort

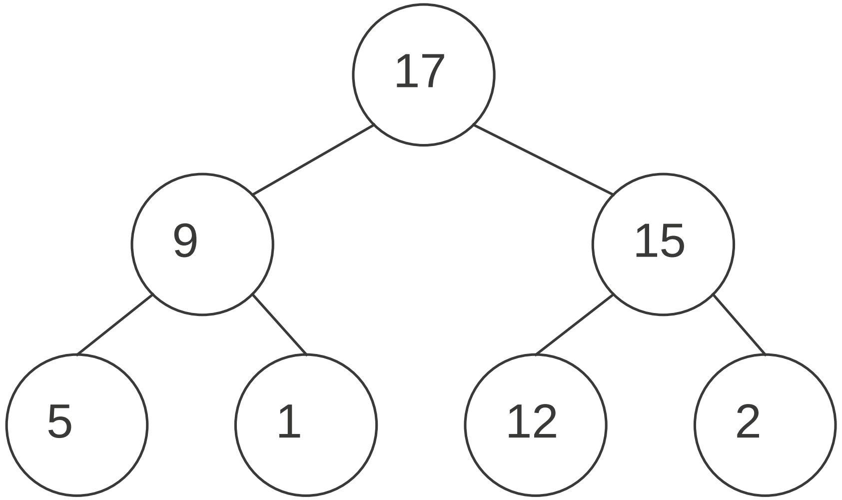
1. **Construct heap: O(N)**
2. **Do N times:**
    1. Get max element: O(logN)
    2. Put it in the “hole” made by previous step: O(1)

- **Summary: O(N*logN) time, O(1) space**

In-place heap sort

- **After making the heap…**

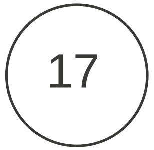
	17	**9**	**15**	**5**	**1**	**12**	**2**
0	1	2	3	4	5	6	7

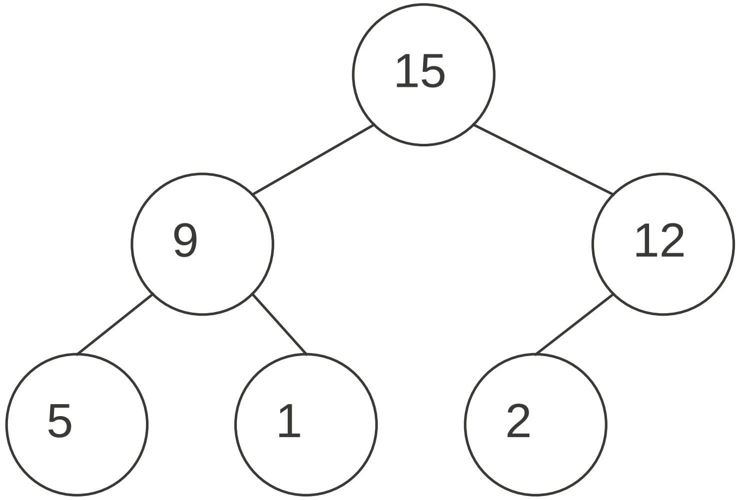


In-place heap sort

- **Remove the max element…**

	15	**9**	**12**	**5**	**1**	**2**	**?**
0	1	2	3	4	5	6	7

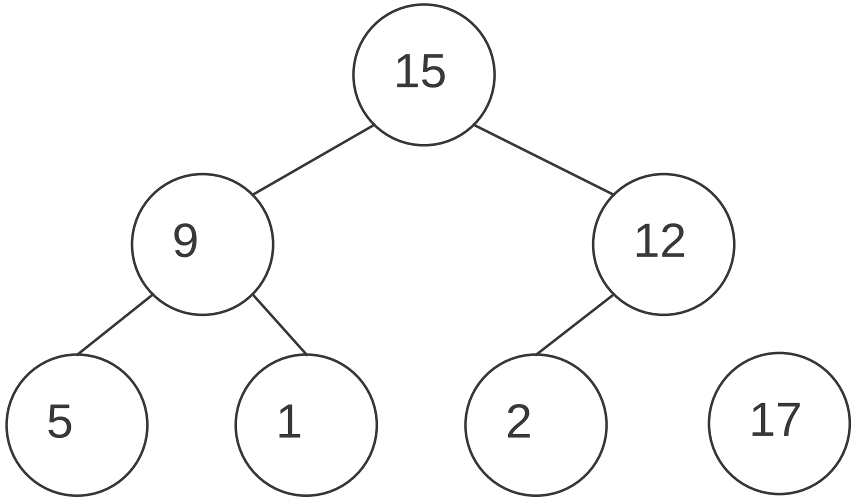




In-place heap sort

- **Put it in the hole.**

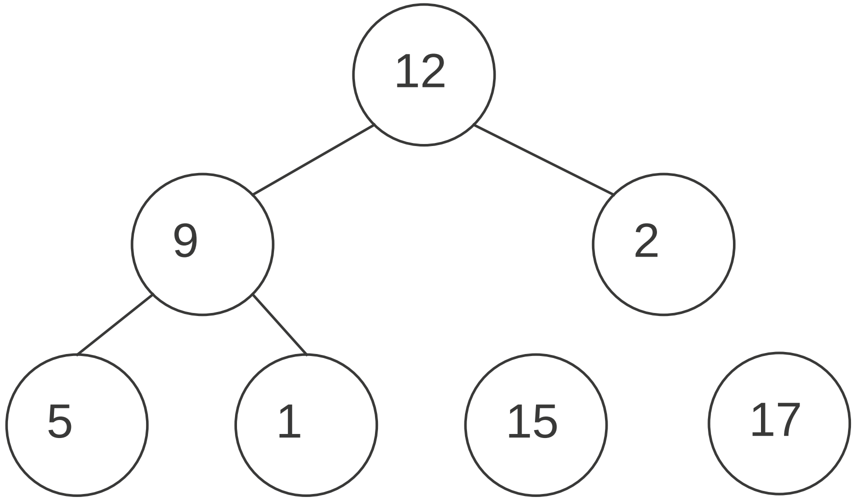
	15	**9**	**12**	**5**	**1**	**2**	**17**
0	1	2	3	4	5	6	7



In-place heap sort

- **Repeat**

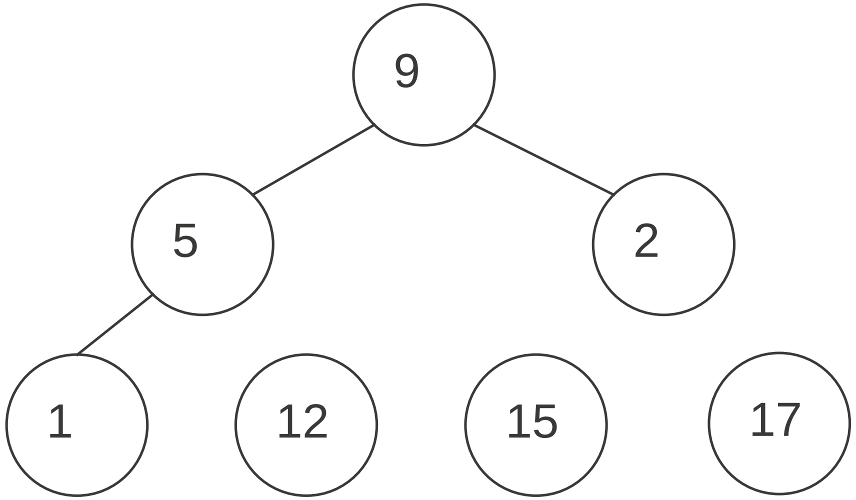
0	1	2	3	4	5	6	7
	12	**9**	**2**	**5**	**1**	**15**	**17**



In-place heap sort

- **Repeat**

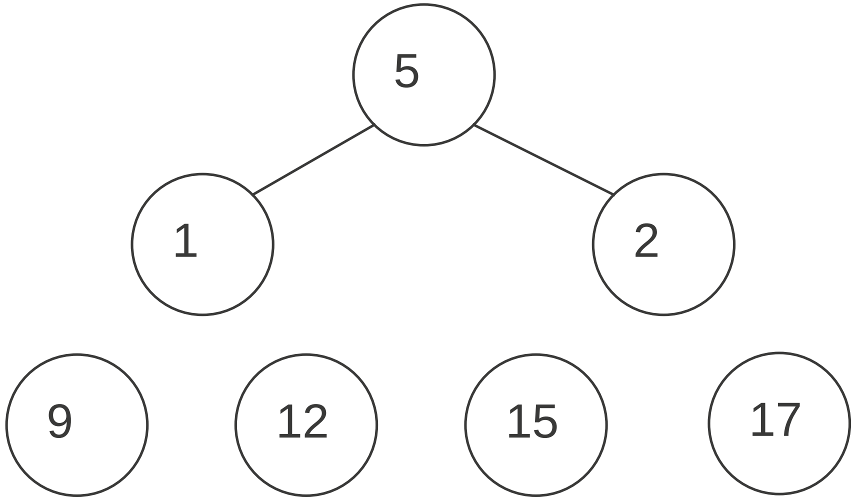
	9	**5**	**2**	**1**	**12**	**15**	**17**
0	1	2	3	4	5	6	7



In-place heap sort

- **Repeat**

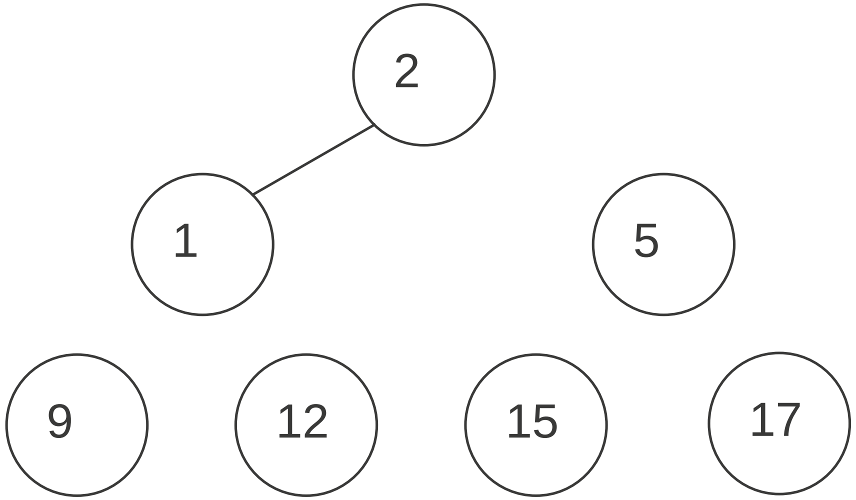
	5	**2**	**1**	**9**	**12**	**15**	**17**
0	1	2	3	4	5	6	7



In-place heap sort

- **Repeat**

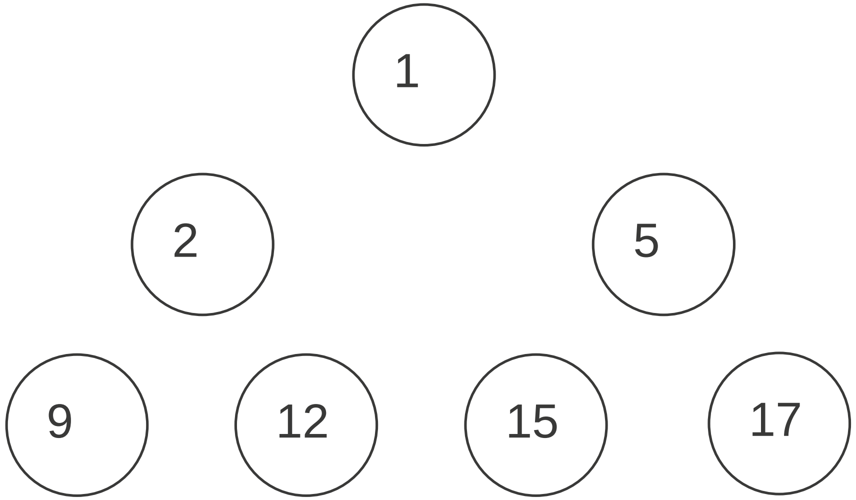
	2	**1**	**5**	**9**	**12**	**15**	**17**
0	1	2	3	4	5	6	7



In-place heap sort

- **Done**

	1	**2**	**5**	**9**	**12**	**15**	**17**
0	1	2	3	4	5	6	7



Summary

- **A simple Heap implementation**
  - rise
  - sink
  - largest_child
- **Queue-Sort**
- **Operations for a clever Heap implementation for Priority Queues**
  - Complexity and correctness