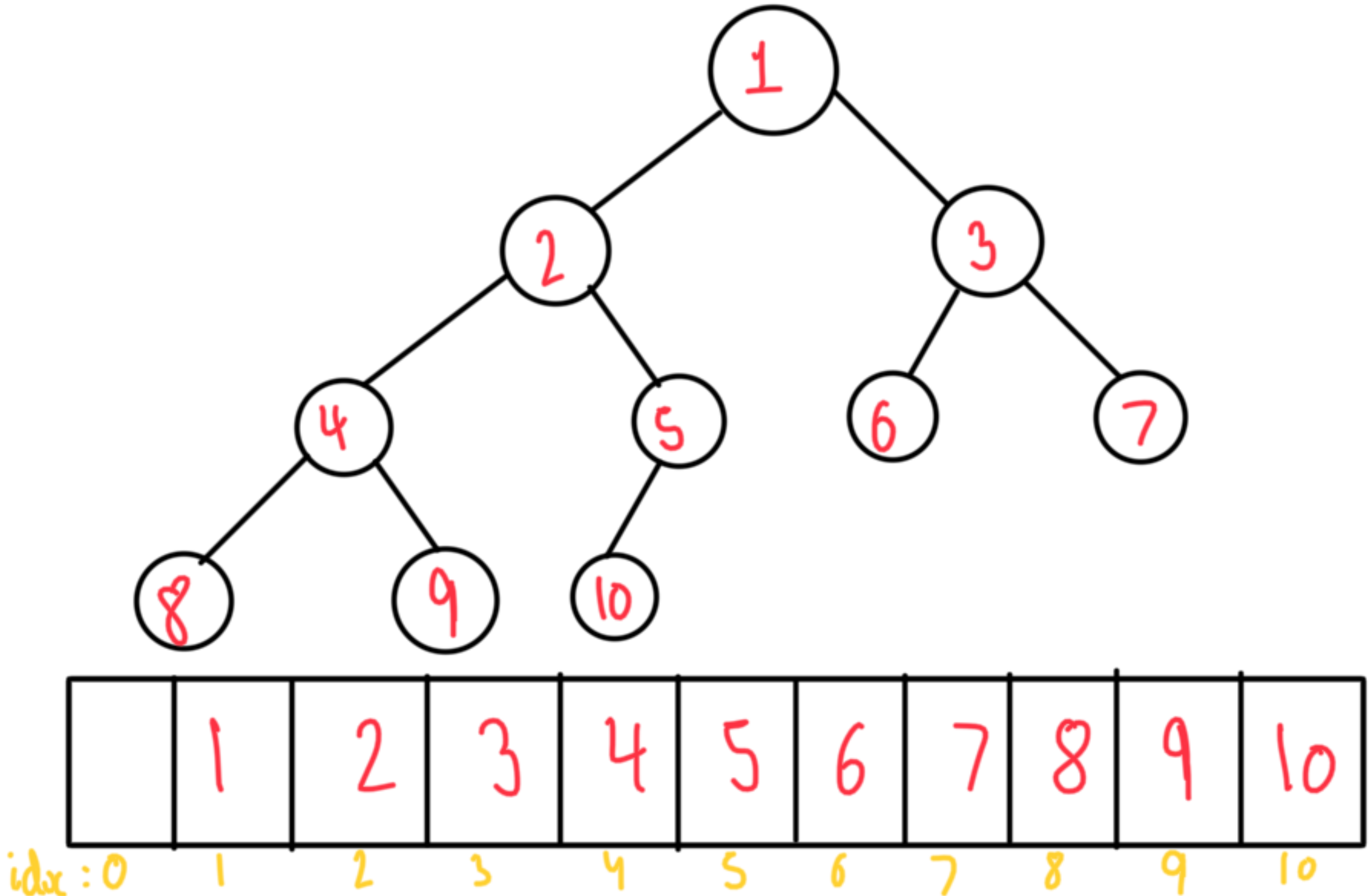
6 Heap Operaions



Parent Node	Corr. Node	Child left	Child right
-	1	2	3
1	2	4	5
1	3	6	7
2	4	8	9
2	5	10	-
K//2	K	2K	2K+1

Definition: Binary Heap Data Structure

The binary heap data structure can be described as follows.

- A binary heap is a complete binary tree (all levels except for the last are completely filled).
- Every element in a max heap is no smaller than its children (i.e. the maximum element is at the top).

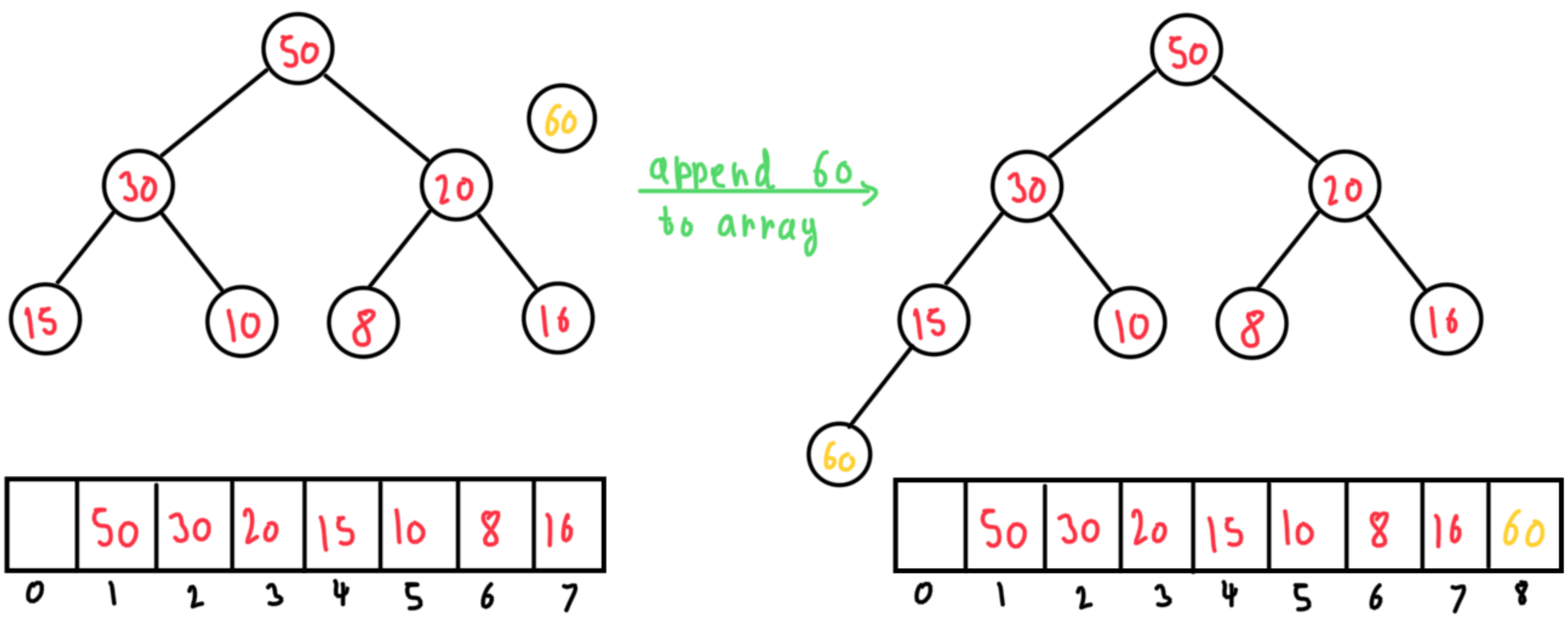
Property: Binary Heap Data Structure

The binary heap data structure has the following properties.

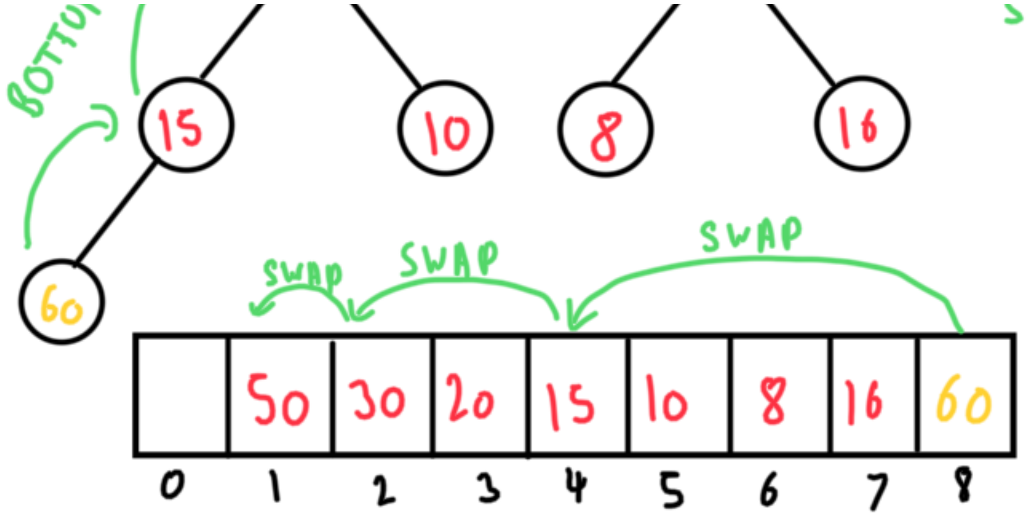
- Due to its structure, a binary heap can be represented as a flat array $array[1...n]$

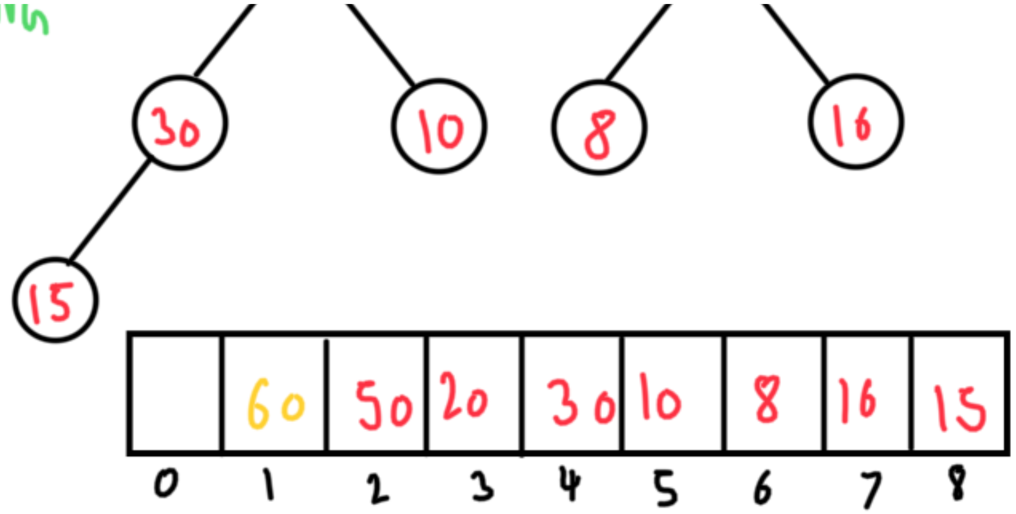
- Due to its structure, a binary heap can be represented as a flat array *array*[1..*n*] where the root node is array[1] and for each node *array*[*i*], its children (if they exist) are elements *array*[2*i*] and *array*[2*i* + 1].
- An existing array can be converted into a heap in place in $O(n)$ time. ← heapify
- A new item can be inserted into a binary heap in $O(\log(n))$ time.
- The maximum element can be removed from a binary heap in $O(\log(n))$ time.

Insert: Adds an element to the heap.



SWAPPING





- Check the parent nodes (K//2).
- If parent node < new node, SWAP.
- Continue SWAPPING till Parent node is $\geq$ new node.

Best : $O(1)$ — occurs when all nodes have same value

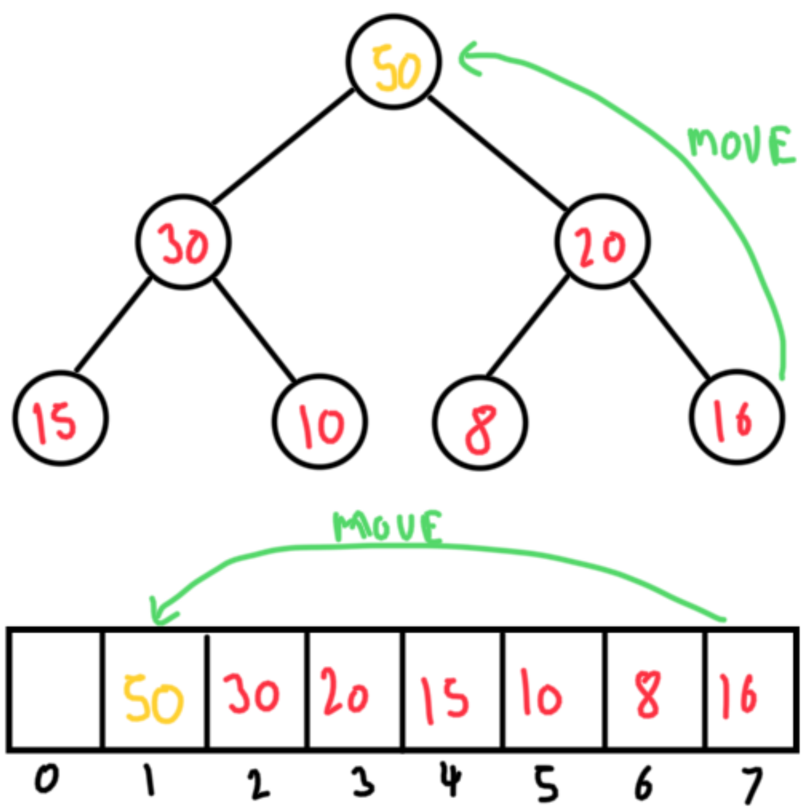
Worst : $O(\log n)$

```
def insert(arr[1...n], key):          — T(n) = log n + c = O(log n)

    arr.append(key)  } c
    n += 1
    rise(A[1... n], n)                 — log n


def rise(arr[1...n], i):              — O(log n)

    parent = ⌊i/2⌋
```

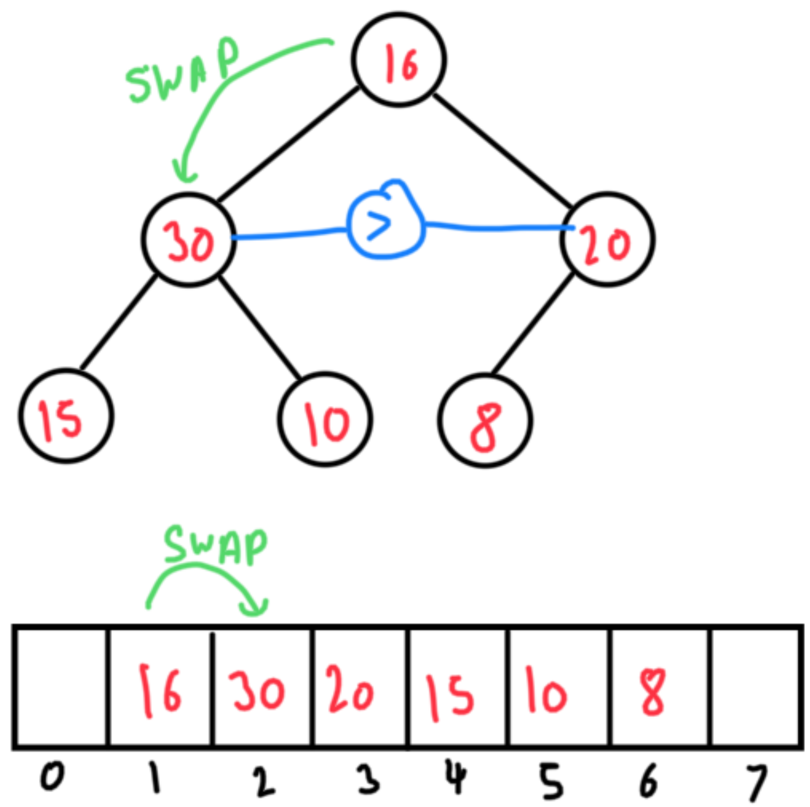
```
while parent >= 1:

    if arr[parent] < arr[i]:
        swap(arr[parent], arr[i])
        i = parent
        parent = ⌊i/2⌋
    else
        break
```

$O(\text{height}) = O(\log n)$

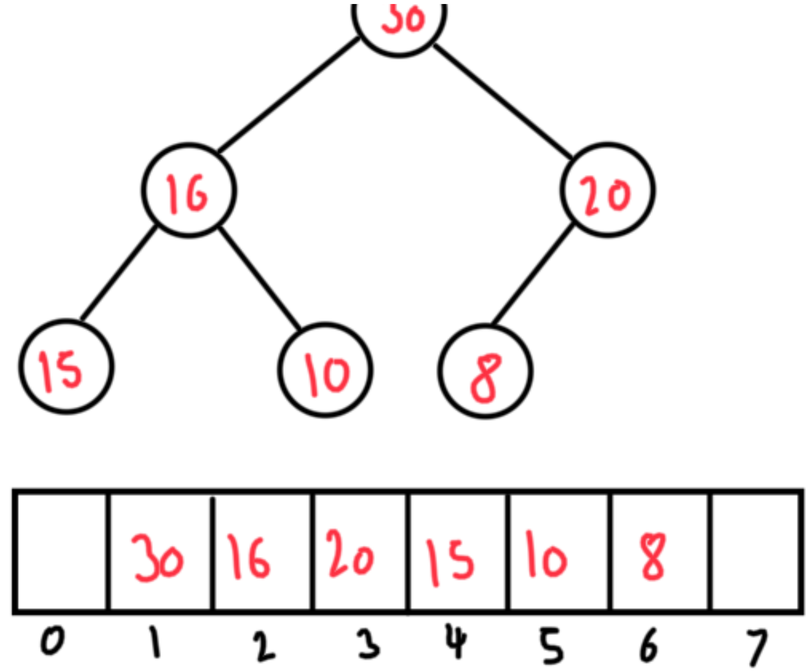
Delete: Removes/Returns the top node/first element from heap



MOVE the first elem in array with the last element.



- After deleting the root node and moving the last node to root, CHECK which child is greater.
- Swap the new root with the greatest child, if new root < greatest child.
- Repeat this till new node ≥ greatest child.




```
def extract_max(arr[1...n]):
    swap(arr[1], arr[n])
    n -= 1
    max_val = arr.POP()
    fall(arr[1...n], 1)

    return max_val
```

$T(n) = c + \log n = O(\log n)$

$\left.\begin{array}{l}\text{swap(arr[1], arr[n])}\\ \text{n -= 1}\\ \text{max\_val = arr.POP()}\end{array}\right\} c$

fall(arr[1...n], 1) — $\log n$

Time:

$\boxed{\begin{array}{l}\text{Best: } O(1)\\ \text{Worst: } O(\log n)\end{array}}$

when all items have same value.

```
def fall(arr[1...n], i):
    child = 2·i
```

$O(\log n)$

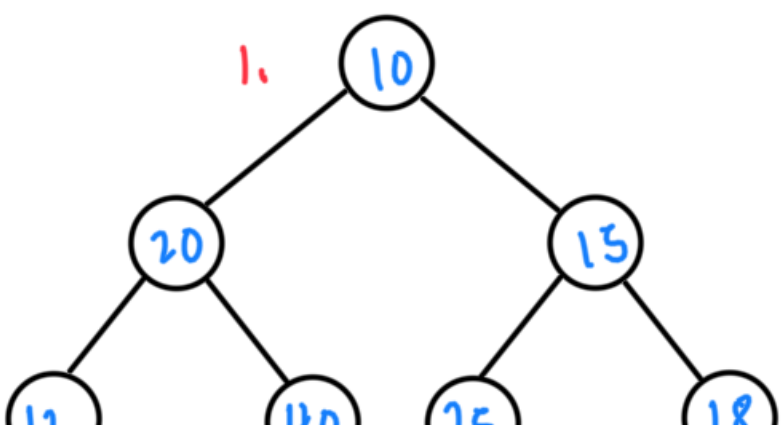
$O(h) = O(\log n)$

```
while child <= n:

    if (child < n ) and (arr[child+1] > arr[child]):
        child += 1

    if arr[i] < arr[child]:
        swap(arr[i], arr[child])
        i = child
        child = 2·i
    else:
        break
```

Heapify: Creating a heap from the given array in a TOP-BOTTOM manner. No use of the INSERT operation.

$\underbrace{\text{TOP-BOTTOM}}_{\text{Fall()}}$ INSERT (Bottom-up)

1.



- From the last node (len(arr)-1) to root node (0), check if they meet the heap invariable.

- If they do not, move that node in

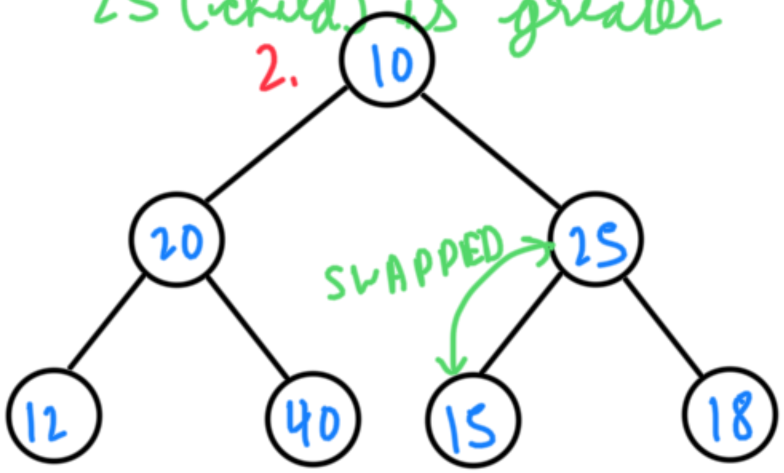
a TOP-DOWN manner like done in delete.

	10	20	15	12	40	25	18
0	1	2	3	4	5	6	7

1. 18 does not have any child, so its part of its heap (sub-heap). Same goes for 25, 40, 12.

2. Now, 15 at i=3 has 2 children (i=6,7). The largest child is 25 at i=6. 25 (child) is greater than 15 (parent), so SWAP.

2.

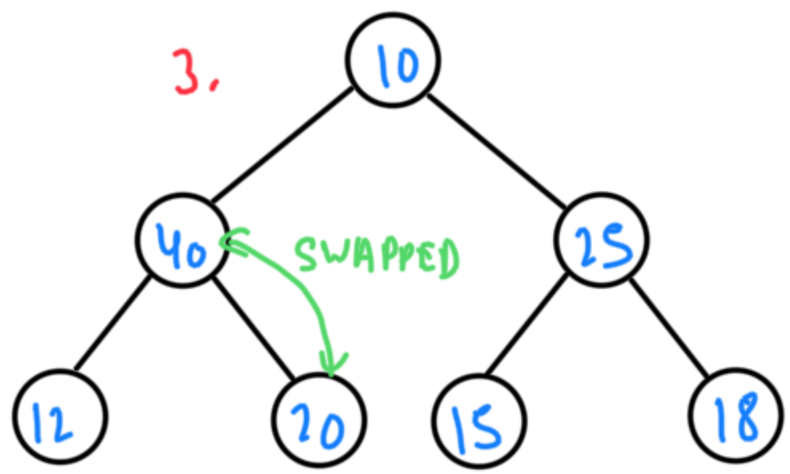


SWAPPED

	10	20	25	12	40	15	18
0	1	2	3	4	5	6	7

3. Now, 20 at i=2 has 2 children at i=4,5. The largest child is 40 at i=5. 40 > 20, so SWAP.

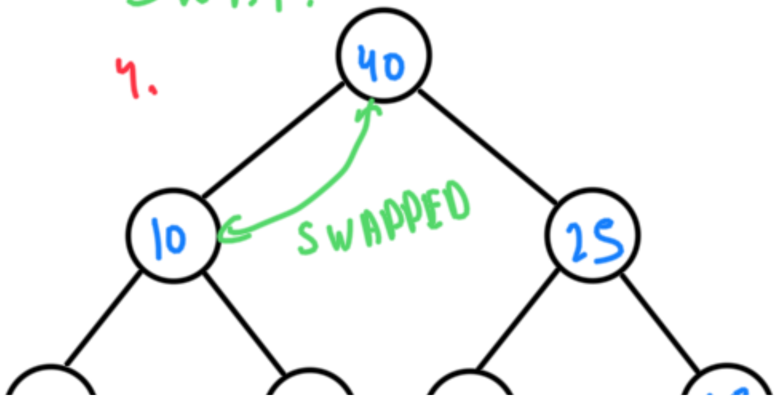
3.



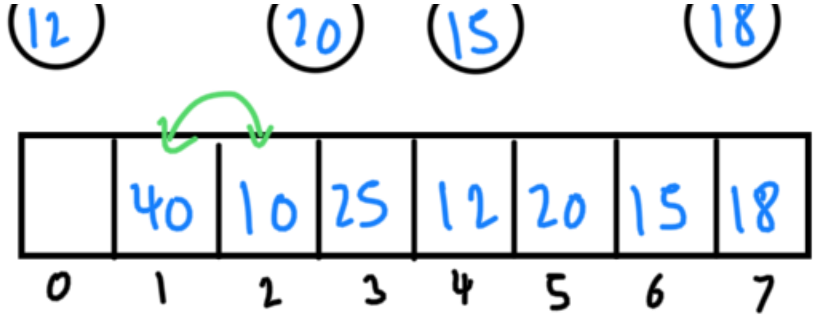
	10	40	25	12	20	15	18
0	1	2	3	4	5	6	7

4. Now, 10 at i=1 has childn at i=2,3. Largest child is 40 at i=2. 40 > 10, so SWAP.

4.

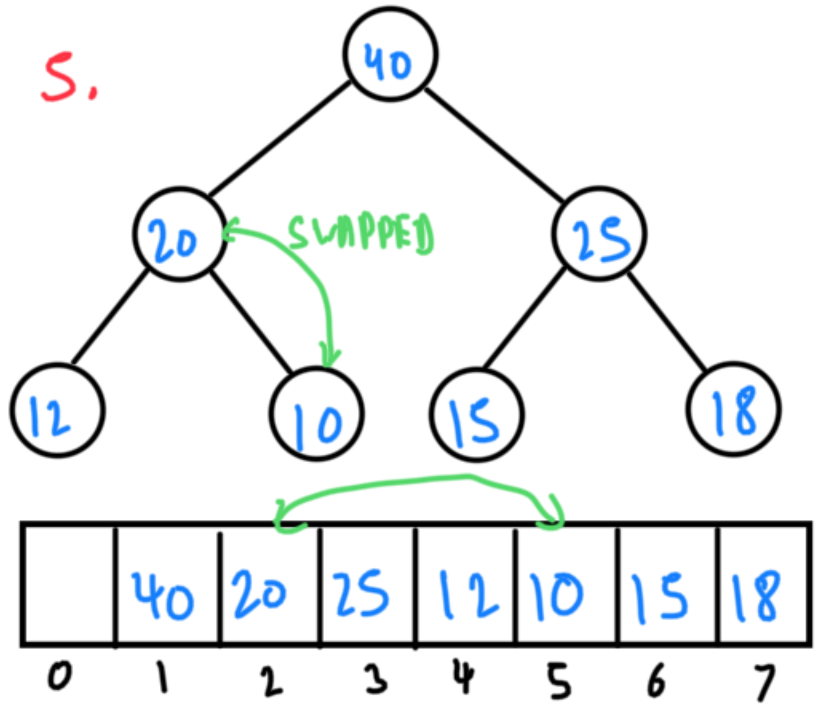


5. 40 & 10 are swapped, but 10 is not meeting the Heap invariable.



So, now sift down.
10 is at $i=2$, its largest child is 20 at $i=5$ & $20>10$, so SWAP.

5.



← Heap Created using Heapify

Heapify Time Complexity: $O(n)$

Creating a heap by inserting: $O(n \log n)$

Creating a heap using heapify- Call- shift down (TOP-DOWN):

Time taken by last level nodes:

Level = h nodes = $n/2$

Time = $(0 * n/2)$

Time taken by 2^{nd} last level nodes:

level = $h-1$ nodes = $n/4$

Time = $(1 * n/4)$

0 shifting down!

1 shift-down in worst case

Time taken by node at 1st level:

Level = 1 nodes : 1

Time = $(h * 1)$

Travelling down the tree's height

Total time = $(0 * n/2) + (1 * n/4) + (2 * n/8) + \ldots + (h * 1)$

$h = \log n$

Time = $0 + \frac{n}{4} + \frac{n}{8} + \ldots + \log n$

$\boxed{\text{Time} = O(n)}$

def heapify(arr[1...n]): —— $O(n)$

for (i = n/2 to n):

fall(arr[1...n], i)

Note: We can skip leaf nodes (last-level).

$\frac{n}{4} + \frac{n}{8} + \ldots + \log n = O(n)$

Code:

```
class MaxHeap:

    def __init__(self):
        self.heap = [None]
```

```
def heapify(self, array):
    self.heap += array
    n = len(self.heap) - 1
    for i in range(n // 2, 0, -1):
        self.fall(i)

def insert(self, x):
    self.heap.append(x)
    self.rise(len(self.heap) - 1)

def extract_max(self):
    if len(self.heap) <= 1:
        return None
    self.heap[1], self.heap[-1] = self.heap[-1], self.heap[1]
    max_value = self.heap.pop()
    self.fall(1)
    return max_value

def rise(self, i):
    parent = i // 2
    while parent >= 1:
        if self.heap[parent] < self.heap[i]:
            self.heap[parent], self.heap[i] = self.heap[i], self.heap[parent]
            i = parent
        else:
            break

def fall(self, i):
    n = len(self.heap) - 1
    child = 2 * i
    while child <= n:
        if child < n and self.heap[child + 1] > self.heap[child]:
            child += 1
        if self.heap[i] < self.heap[child]:
            self.heap[i], self.heap[child] = self.heap[child], self.heap[i]
            i = child
        else:
```

```
            break


    def __str__(self) -> str:
        return str(self.heap)
```