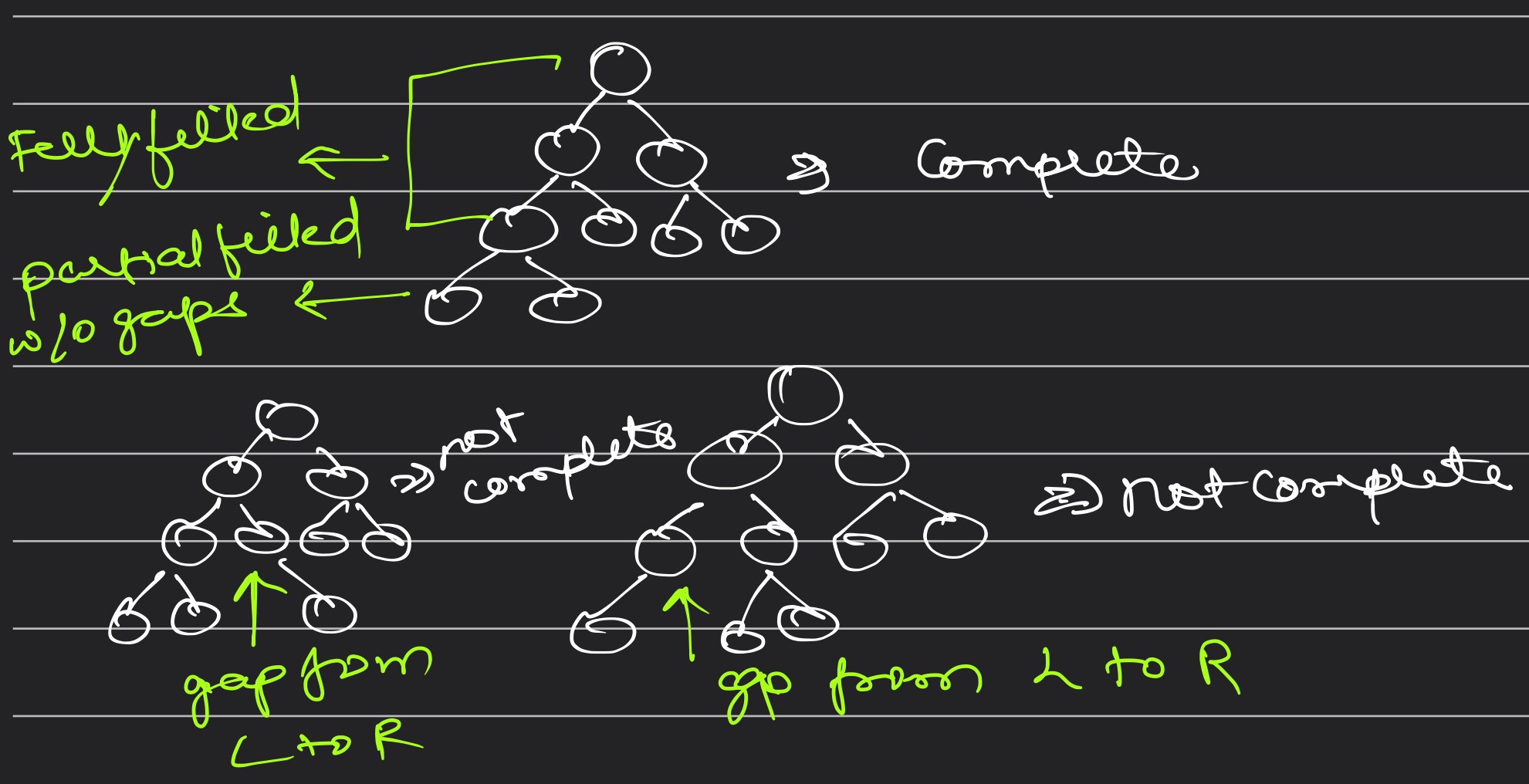
Binary Heap in Python

- used to implement Heap Sort
- Priority queues → Huffman Coding, Dijkstra Algo etc
- Two Types
  - Min Heap (Highest priority is assigned lowest value)
  - max (Highest priority assigned highest value)

Binary Heap is complete Binary Tree (stored as an array)

A Binary tree is called complete when all of its level are completly filled except possibly the last level and the last level should be filled from left to right, no gaps are allowed.



What advantage of Complete Binary tree ? why do we use such data structure.

An array constructed using complete Binary tree is useful for creating priority queues.
This array hold the parent child relationship.

Say

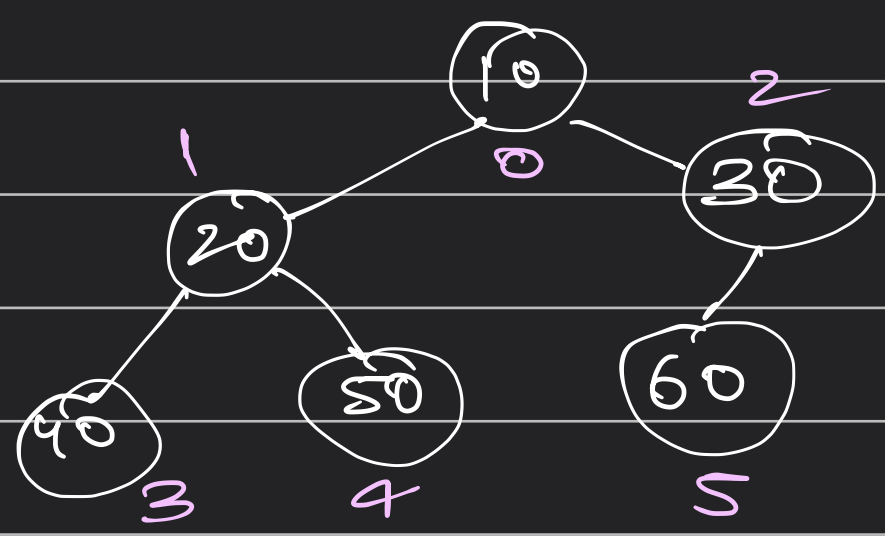
$$\text{Left}(i^{\circ}) = 2i^{\circ} + 1 \quad *$$

$$\text{Right}(i^{\circ}) = 2i^{\circ} + 2 \quad *$$

We can obtain Left or Right element at any index "i" in our array using the above formulas

Parent of Node at Index i

$$\text{Parent}(i^{\circ}) = \left\lfloor \frac{i^{\circ} - 1^{\circ}}{2} \right\rfloor$$



L To R

10	20	30	40	50	60
0	1	2	3	4	5

Left(1) = 2x1 + 1

Node20 Left = 3 = Node40

Right(0) = 2x0 + 2 = 2

Node10 Right = 2 = Node30

Node60
Parent(5) = $\lfloor \frac{5-1}{2} \rfloor$
Node30 = 2

Advantages

- Cache Friendly → every element is at contiguous location, one after another in space, terms following array property and maintaining parent child relationship

- Height of this tree is minimum possible, as only last level is allowed to be partially filled.

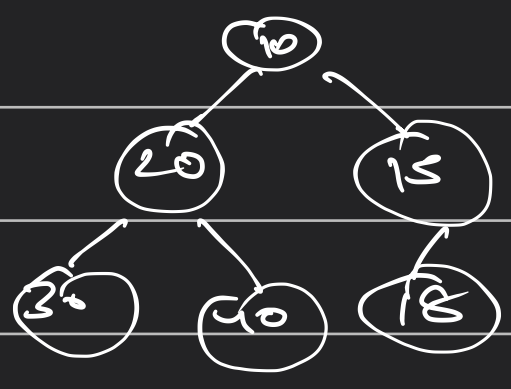
- No space wastage for keeping pointer information

- N items only require N size array allocation

Min Heap

→ Complete Binary tree

→ Every node has value smaller than its descendants.



$10 < 20, 15, 30, 40, 18$

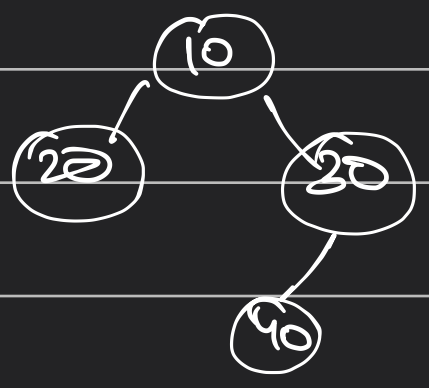
$20 < 30, 40$

$15 < 18$

we consider any Node as root everything under its subtree should be bigger than it.

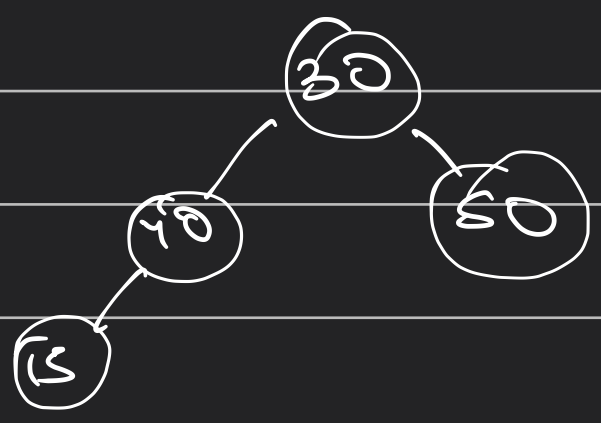
Question

(1)



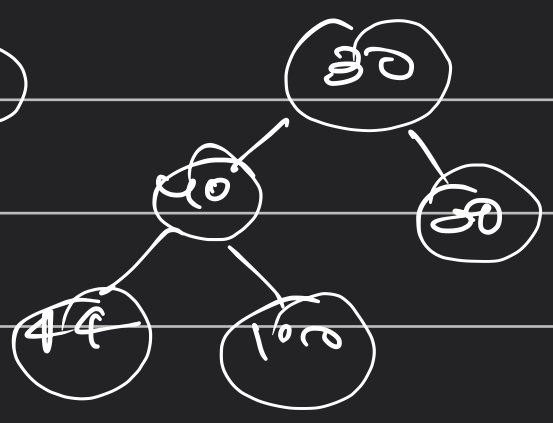
→ Not a min heap ∵ not a complete binary tree

(2)



→ Not a min heap, as 40 is greater than 15

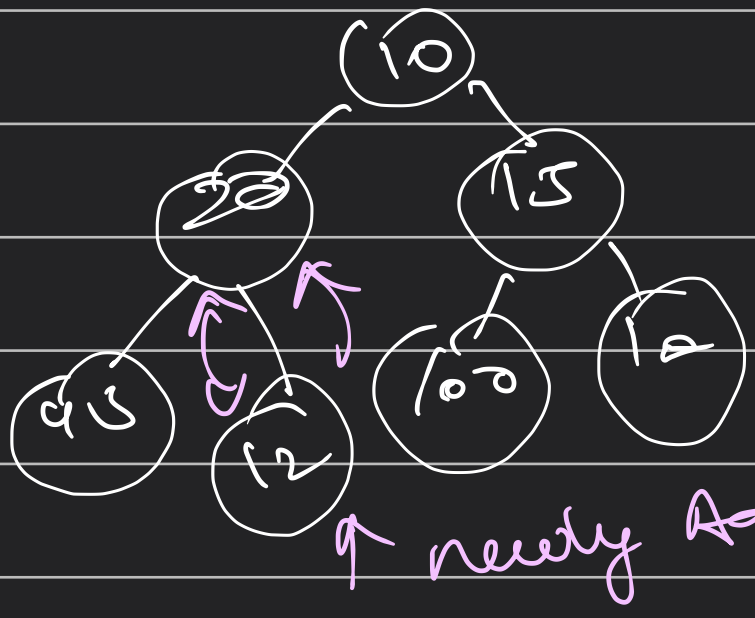
(3)



→ Min Heap yes, complete binary tree and each node is less than its descendent.

Insert in Heap (min Heap)

Append at last, compare with parent if parent is bigger than swap.



Stopping Condition : Until we reach the root or we find a parent which smaller than the item added.

Class MinHeap:

def insert(self, x):
    self.arr.append(x)
    i = len(arr) - 1
    while i>0 & self.parent(i) > arr[i]:
        p = self.parent(i)
        arr[i], arr[p] = arr[p], arr[i]
        i = p

function control

check Until we reach root or parent is smaller gets fixed.

Basic min heap class

class MyMinHeap:
    def __init__(self):
        self.arr = []
    def parent(self, i):
        return (i - 1)//2

    def leftChild(self, i):
        return (2*i + 1)

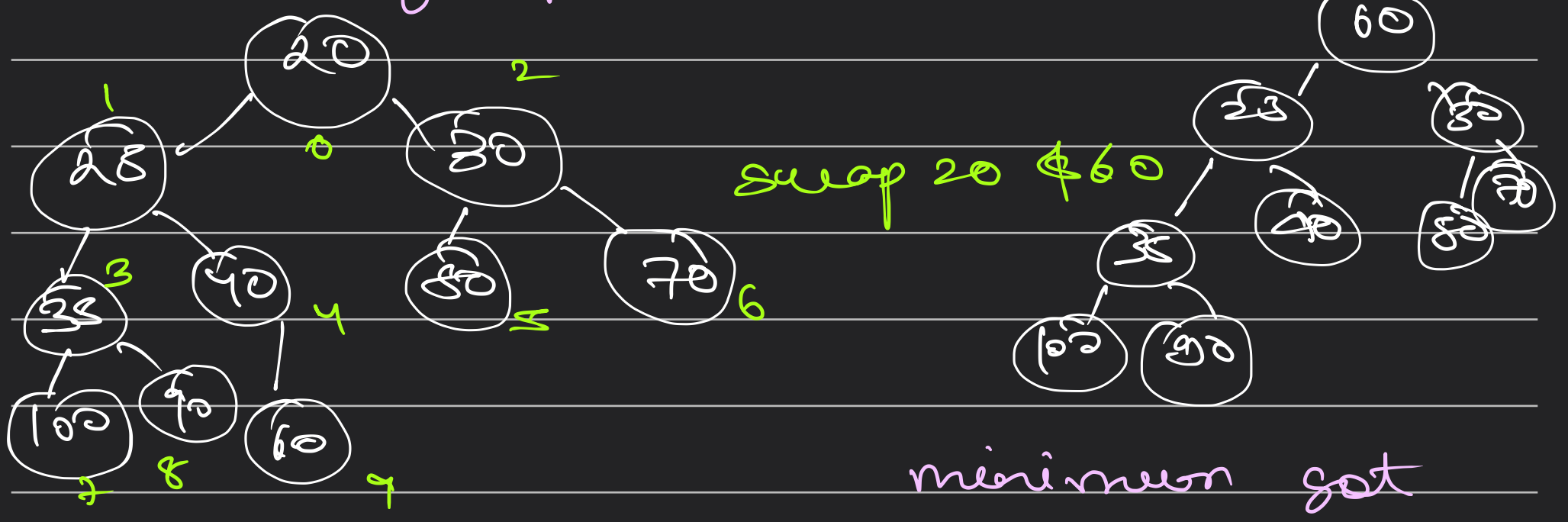
    def RightChild(self, i):
        return (2*i + 2)

Extract Min & Heapify

So in min heap the first element is considered to be minimum of the

heap
~~arr~~ from which heap array is developed
But after extracting the min, the
properties of "Min Heap" would get
disturb.
So to maintain it we swap min
i.e first element with the last one
and then heapify the array i.e
that swapped element will be sent
to its correct position in the Min
Heap after we have extracted the
min.

lets see how would be do, heapify
this works on the fact that in
the min heap only root is at
the wrong position



minimum got
extracted and is returned but our
heap is not min heap, or priority queue.

We use Heapify as besides root everyone
is at correct position.
We would recursively solve this problem
Now root should be min so we compare with left and right child

```
def minHeapify(self, i):
    arr = self.arr
    lt = self.leftChild(i)
    rt = self.rightChild(i)
    smallest = i
    n = len(arr)
    if lt < n and arr[lt] < arr[smallest]:
        smallest = lt
```

for indexing errors

index

```
    if rt < n and arr[rt] < arr[smallest]:
        smallest = rt
    if smallest != i:
        arr[smallest], arr[i] = arr[i], arr[smallest]
        self.minHeapify(smallest)
```

i.e until we update root

This we will recursively update the swapped root to its final position in min Heap.

The subtree of L, R and P would be changed recursively.

How would you extract min?

```
def extractMin(self):
    arr = self.arr
    n = len(arr)
    if n == 0:
        return -∞
```

```
res = arr[0]
arr[0] = arr[n-1]
arr.pop()
self.minheapify(0)
return res.
```

Decrease key ⇒ reduce a value

decrease(i, x)

↳ at ith index update value of x and x is < arr[i].

```
def decreasekey(self, i, x):
    arr = self.arr
    arr[i] = x
    while i != 0 & arr[parent(i)] > arr[i]:
        p = self.parent(i)
        arr[i], arr[p] = arr[p], arr[i]
        i = p
```

Need to make the min heap stable after the key decrease ⇒

Delete operation

Here the value gets replaced with "$-\infty$". Then what we do is we call decrease key with $-\infty$ this will update the min heap and $-\infty$ would go to the root. As being smallest and this min heap property is maintained

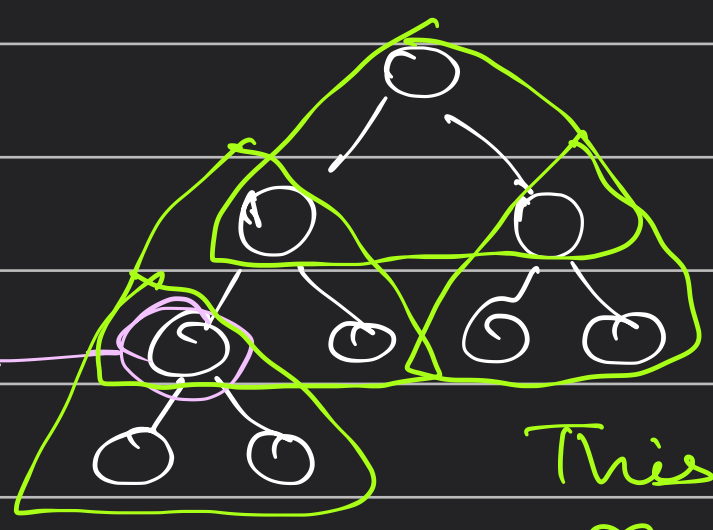
After this we call extractmin() this would remove the $-\infty$ and we will have a tree with the original value deleted

value to be deleted ⇒ Replace by $-\infty$ ⇒ $-\infty$ removed through extractmin()

```
def deletekey(self, i):
    n = len(self.arr)
    if i>=n
        return :
    else :
        decreasekey(i, -math.inf)
        extractmin()
```

- Building a minHeap from given array.

Here would follow a bottom up approach of fixing our subtree with a R and Parent ie three nodes. and we would move up recursively trough minHeap



Only problem we need this node as starting point

lastindex

$$index = (len(arr) - 1 - 1) // 2$$ → this gives index of parent for last child.

This is now minheapify will work

```
def __init__(self, arr = []):
    self.arr = arr
    i = (len(arr) - 2) // 2
    while i >= 0:
        self.minHeapify(i)
        i = i - 1
```