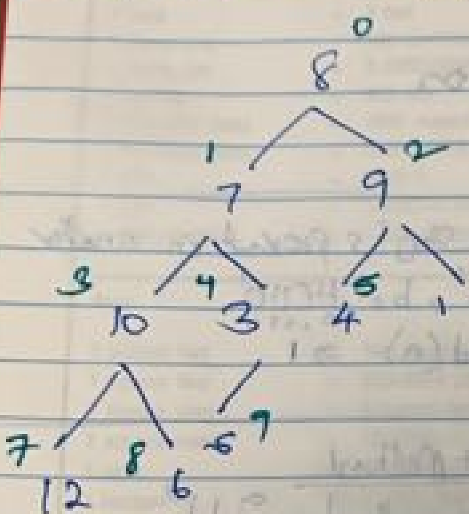
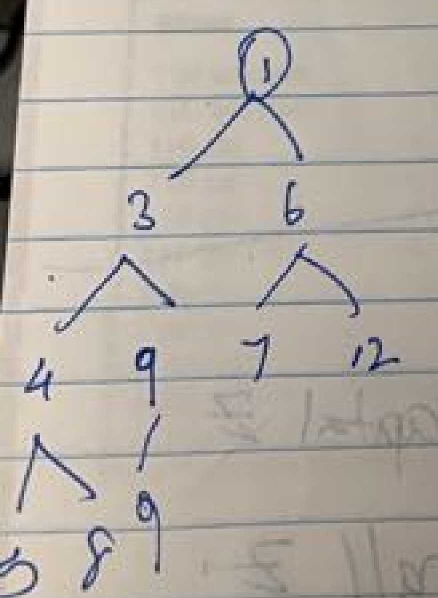
Heaps and Heap Sort

0	1	2	3	4	5	6	7	8	9	10
8	7	9	10	3	4	1	12	6	5	~~X~~



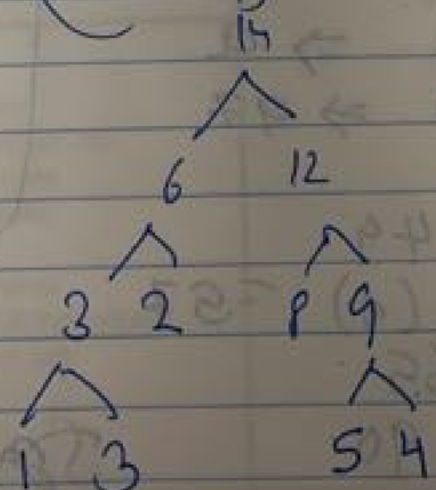
Min Heap

Value at each node is less or equal than its children

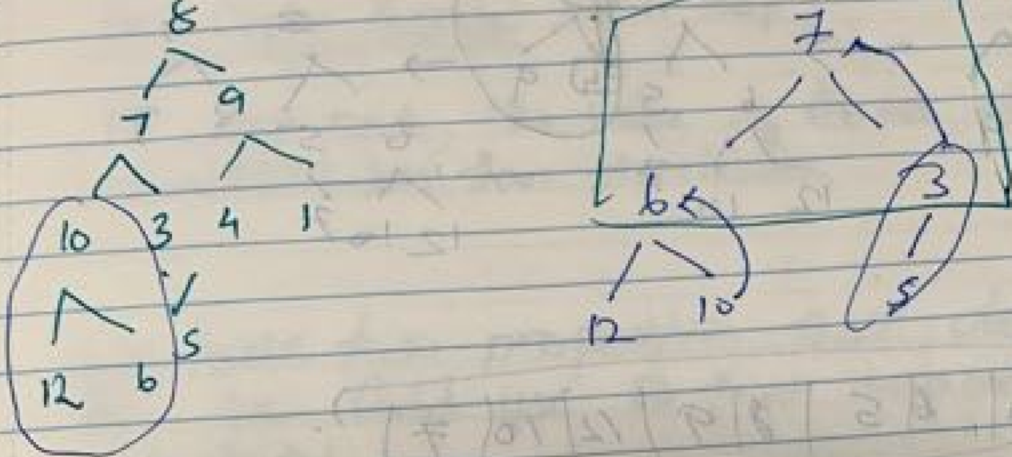


Max Heap

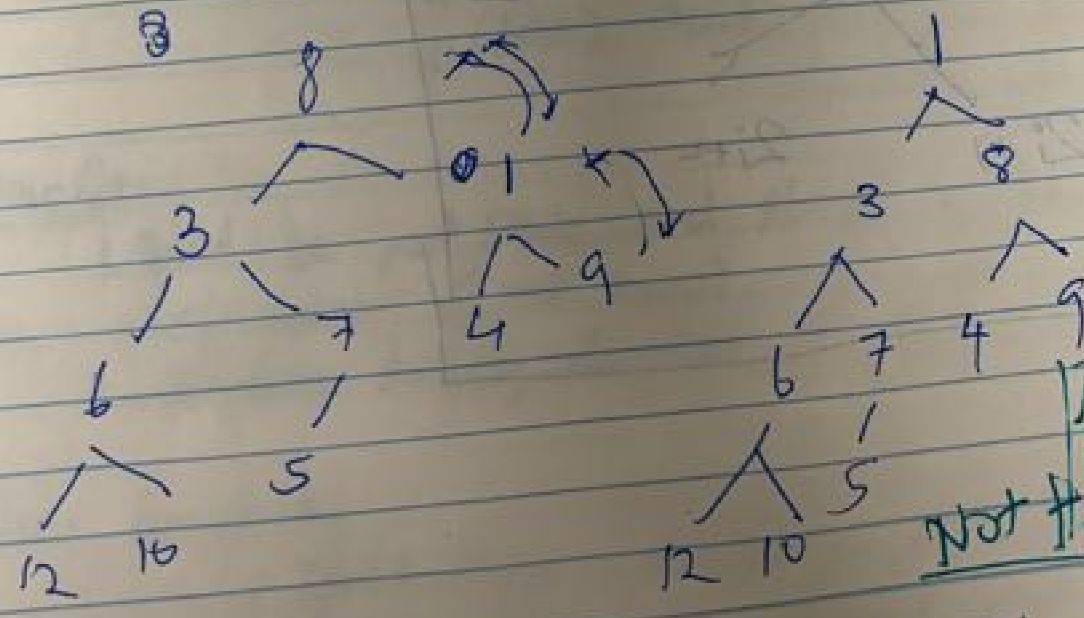
Value at each node is greater than its children (or equal to)



Heapify



Swap the lowest of the two nodes with its parent.

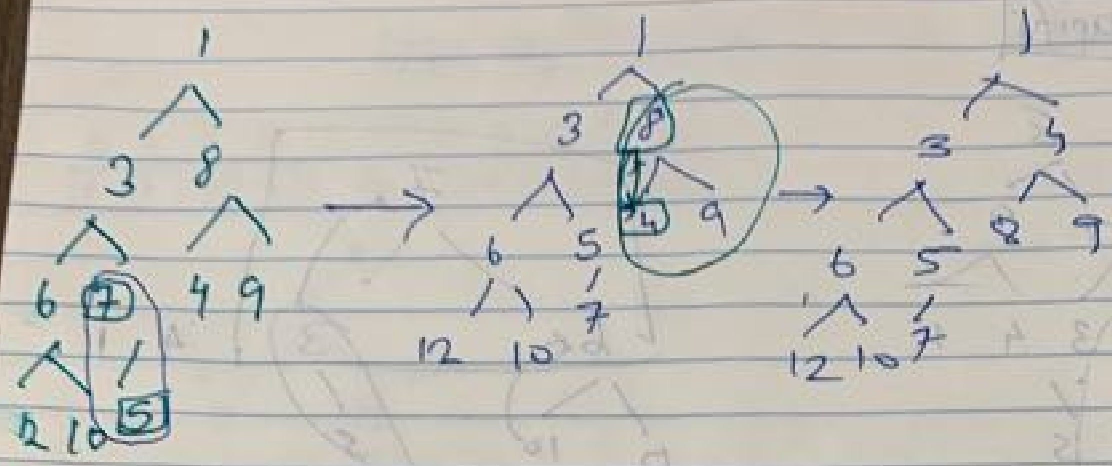


MIN

Not Heap!?

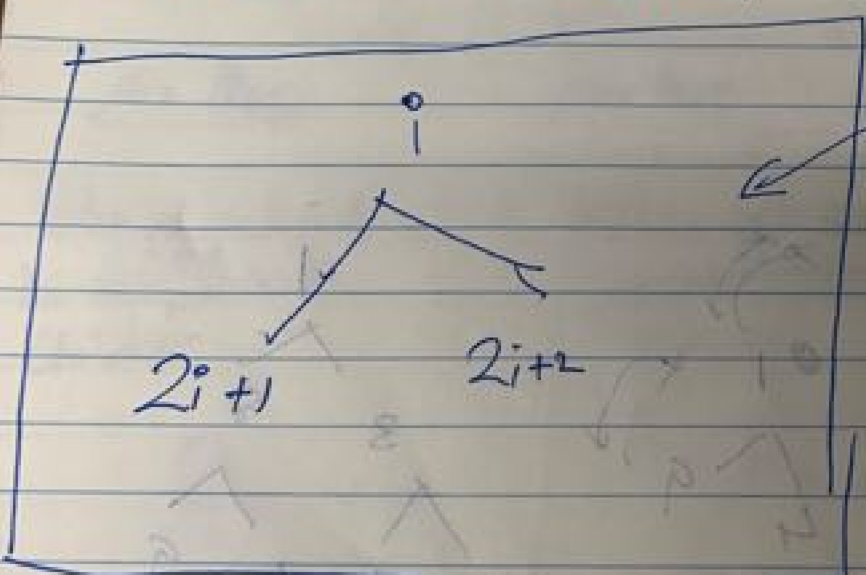
1	3	8	6	7	4	9	12	10	5

Min Heap



1	3	4	6	5	8	9	12	10	7

1 is the minimum element in this Heap



Heapsort

HeapSort   Binary Heap

Heap Order Property: (Min Heap)

For each node, The value of the root element should be less than the children.

We have two properties that ~~defines~~ ~~the~~ we could use so that heap order is maintained.

(1) Heapify Up
(2) Heapify down.

Time Complexities : O(1) findmin
O(logn) Insert
O(logn) delete

Heap Sort:
1. Delete the root Node and place it in the left corner (min heap)
2. Perform heapify_down()

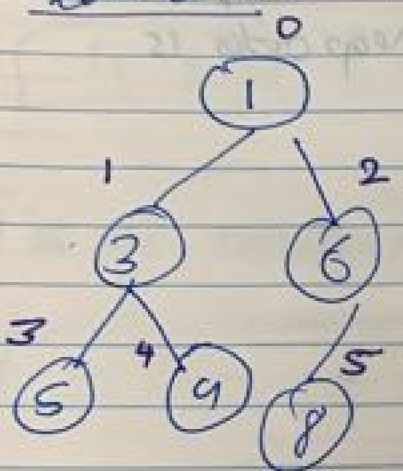
Time Complexity
1. delete_max O(1)
2. Heapify down (logn) * n times
nlogn

Binary Heap

1. Is a Complet tree.
2. Root element is at Arr [0]

Arr [i/2]	Returns parent node
Arr [2i+1]	Returns left child
Arr [2i+2]	Returns right child

Traversal method to Achieve representation is level order.



1 3 6 5 9 8

Binary Heap Applications:- Union Operation

- Heap Sort : O(nlogn)
- Priority Queue: Priority queue can be efficiently implemented using Binary heap

(*) Insert
(*) delete
(*) extractmax
(*) decreasekey:- reduces any value

O(nlogn)

Graph Algorithms - Dijikstra's Shortest Path.
- Prims Minimum Spanning Tree.

Many Problems can be efficiently solved with heaps.

1. K^{th} largest Element in an array
2. Sort an almost Sorted array.
3. Merge K Sorted arrays.

---

heapifying up

Usually this is called with an "add" Operation.

When we add the item to the end of the list and then move it upward.

We have to Continue moving it upward until it is necessary.

The Code is straightforward (→ Contd)

PseudoCode

```
def heapifyU():
    index = Size -1
    while [hasParent] and [parent > item]
        swap (parent, item)
            index = parent's index
```

Python

```
def heapifyUp():
    index = Size-1          #
    while has_Parent(index) and
            parent(index) > items[index]:

        Swap(parent_index[index], index)
        index = parent_index(index)
```

Implementation of ...

Heapify down():

As long as there are children we need to check which one ~~check for~~ is the lowest

```
def heapify-down():
    index=0
    while (has-left-index(index)
        smallerchild-index = left-child(index)
        if (has-right-child(index) &
            right-child(index) < left-child(index))
            smaller-child-index = right-child(index)

        if items[index] < items[smaller-child-index]
            break

        else
            swap(index, smaller-child-index)

        index = smaller-child-index
```

When to use these?!

~~kth largest element in the array~~

Heap Sort: ✓ Tail Recursion

In heap Sort we consider implementing a $\boxed{\text{max Heap}}$

Heapify Down →Deletion $\boxed{\text{percolate - down}}$

~~Ver~~

Initialize largest as root

```
def heapify (arr, n, i):
    largest = i
    l = 2*i+1
    r = 2*i+2
    if l<n and arr[l] > arr[largest]
        largest = l
    if r<n and arr[r] > arr[largest]
        largest = r

    if largest != i
        Swap (arr[i], arr[largest])
        heapify (arr, n, largest)
```

Percolate down

Please note:

when you remove an element, we swap the root element with the element to be removed (at index i) and then perform heapify down.

Heapify Op → Insertion.

The element is at the lowest position. Because we have just added it.

Addition of a new element always happens at the end.

def heapify-up (arr, n, i) (bubble up)

```
p = (i-1) // 2
if p < n and arr[p] > arr[i]:
    swap ( arr[i], arr[p]
    heapify-up ( arr, n, p).
```

Priority Queue

~~Every item has a priority associated with it~~

K^{th} largest item

There are 2 ways of solving this problem using Heaps

How to build a max heap?

Method 1:
Use max heap [straight forward way]

Add all items to max heap $\boxed{O(n)}$
(Build a max heap)

Extract Max K times → $O(K \log n)$

$O(n + K \log n)$

Method 2:

1. Build min heap of arr[0] to arr[K-1]
2. For each element compare it with root
~~Root is greater than root call Heapify~~
add
If the element is greater than root
make it root and call Heapify
else ignore the element
Step 2 time complexity
$O((n-K) \log K)$

MinHeap has K largest elements. Root of MH has the K^{th} largest element

How to Build a Max heap

```
for (int i = n/2 -1; i >= 0; i--)
    heap(arr, n, i)
```

$O(n)$: Complexity

~~Reasop~~ Reasonin
If you build the heap element at a time you will end up at a total of $n \log n$ operations. This would take a lot of time

The bottom up heap building at each level. In this case the Lowest Level (that has lots of nodes)

$\sim n/2$ nodes at zero
$\sim n/4$ nodes at height 1
$\sim n/8$ nodes at $h=2$.
$\sim n/16$ nodes at $h=3$
$n/2^{i+1}$ nodes for height i

$$\sum_{h=1}^{\log n} \lceil n/2^{h} \rceil \leq n \sum_{h=1}^{\infty} 2^{h} = n\left(1/2 + 1/4 + 1/8 + \dots\right)$$

$= n$

move zeros:

0	1	2	3	4	5
[1	0	2	0	3	0]

Curr = non = 0
non = 1, Curr = 1
non = 1 Curr = 2 Swap [1 2 0 0 3 0]
non = 2 Curr = 3
non 2 Curr = 3 X
non = 2 Curr = 4 Swap
3 5
Keep track of

Track the place where there is a zero.

Heap Vs Binary Search Tree

~~Here Order~~

Heap	Binary Search Tree
- good for PQ implementation	Search O(log n)
- find O(1)	print all items in sorted order O(n)
insert O(log n), delete = O(log n)	
Search O(n)	floor and ceil in O(log n)
print all items in sorted order O(n log n)	kth smallest / largest with additional data structure } O(log n)

↓ Curr
[1,3,2,0,3,0]

Here ORDER Has to be ~~Maintained~~ Maintained

non_zero	Curr	
0	0	Swap
1	1	Swap
2	2	Swap
3	3	—
3	4	Swap
4	5	

$\boxed{arr[Curr] \,!= 0}$ Increment

[1,3,2

~~non zero either stays at~~

(*) Non zero pointer moves to the next location only when there is a presence of non zero element

[1,3,2,0,3,0]

(*) Curr always moves to the next position.

```
def movezeros(arr):
    totsize = len(arr)
    non_zero = Curr = 0
    while Curr < tot_size
        if arr[Curr] != 0
            arr[non_zero], arr[Curr] = arr[Curr], arr[non_zero]
            non_zero += 1
        Curr += 1
    print(arr)
```