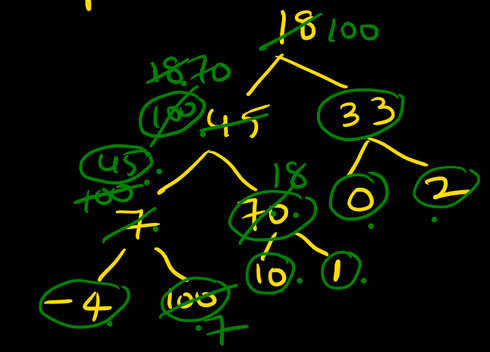
18	45	33	7	70	0	2	-4	100	10	1
1	2	3	4	5	6	7	8	9	10	11

⇒ Sort this using heap Sort. ✓

① Build max-heap (You can do ascending order) ✓

Build min-heap (You can do descending order)



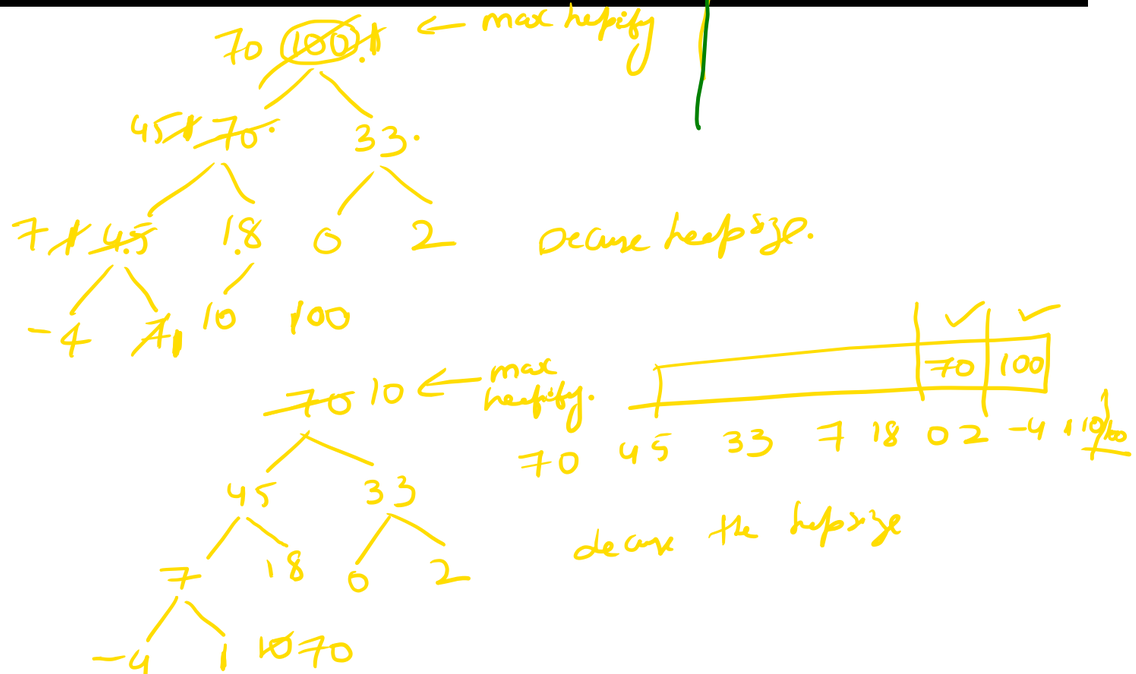
max heap. ✓

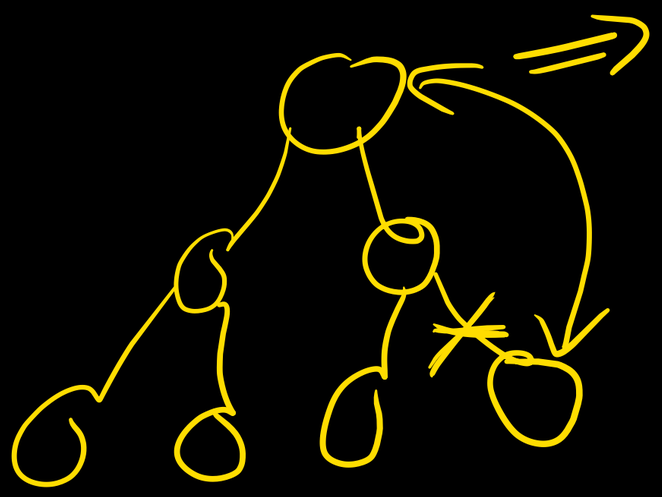
Every sorted array is a heap.

Every heap need not be sorted

100	70	33	45	18	0	2	-4	7	10	1

⇒ Sorted? ✗





we always exchange Root with the last element of the heap.

always highest element will be at the end of the array.

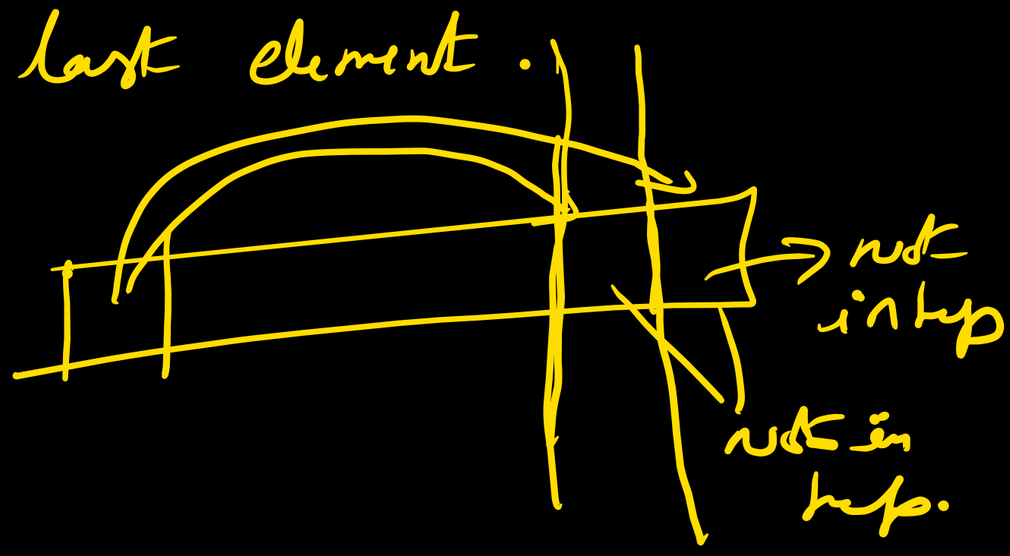
decrease the size of the heap.

apply max heapify.

① Exchange the root with last element.

② Decrease heap size.

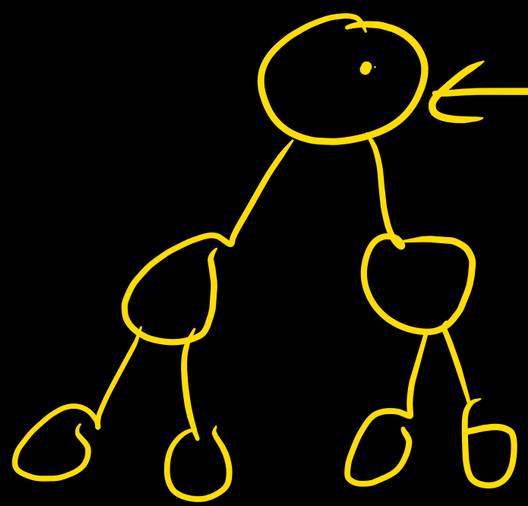
③ max heapify.

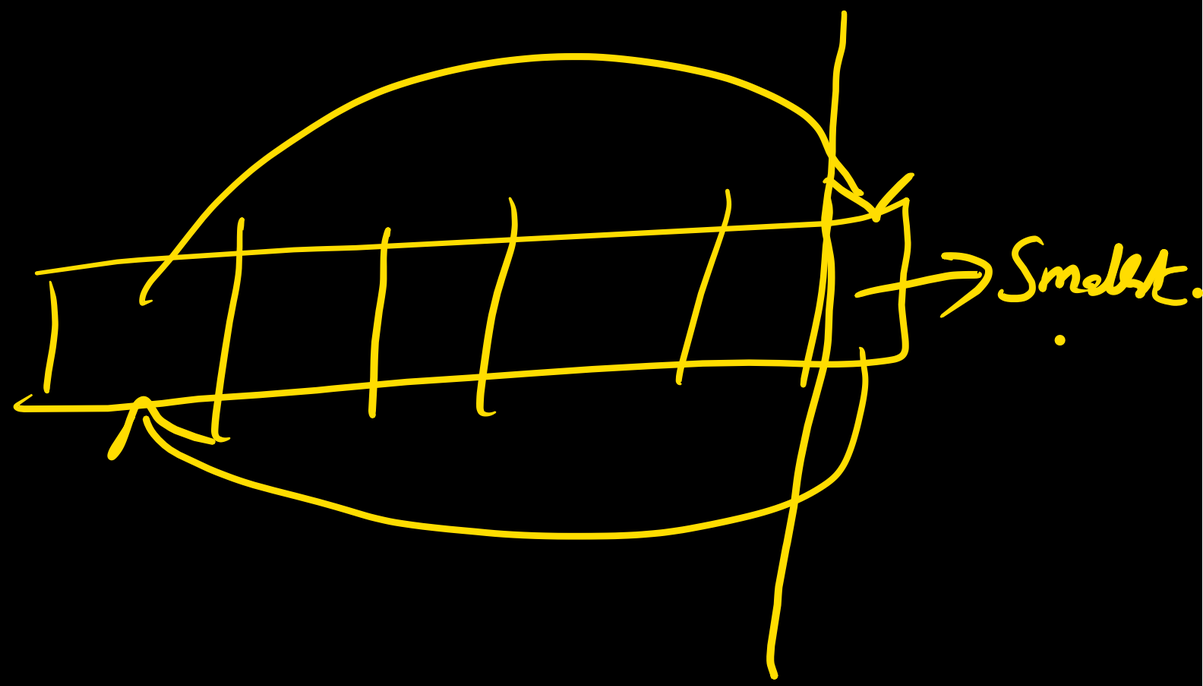


// ascending order.

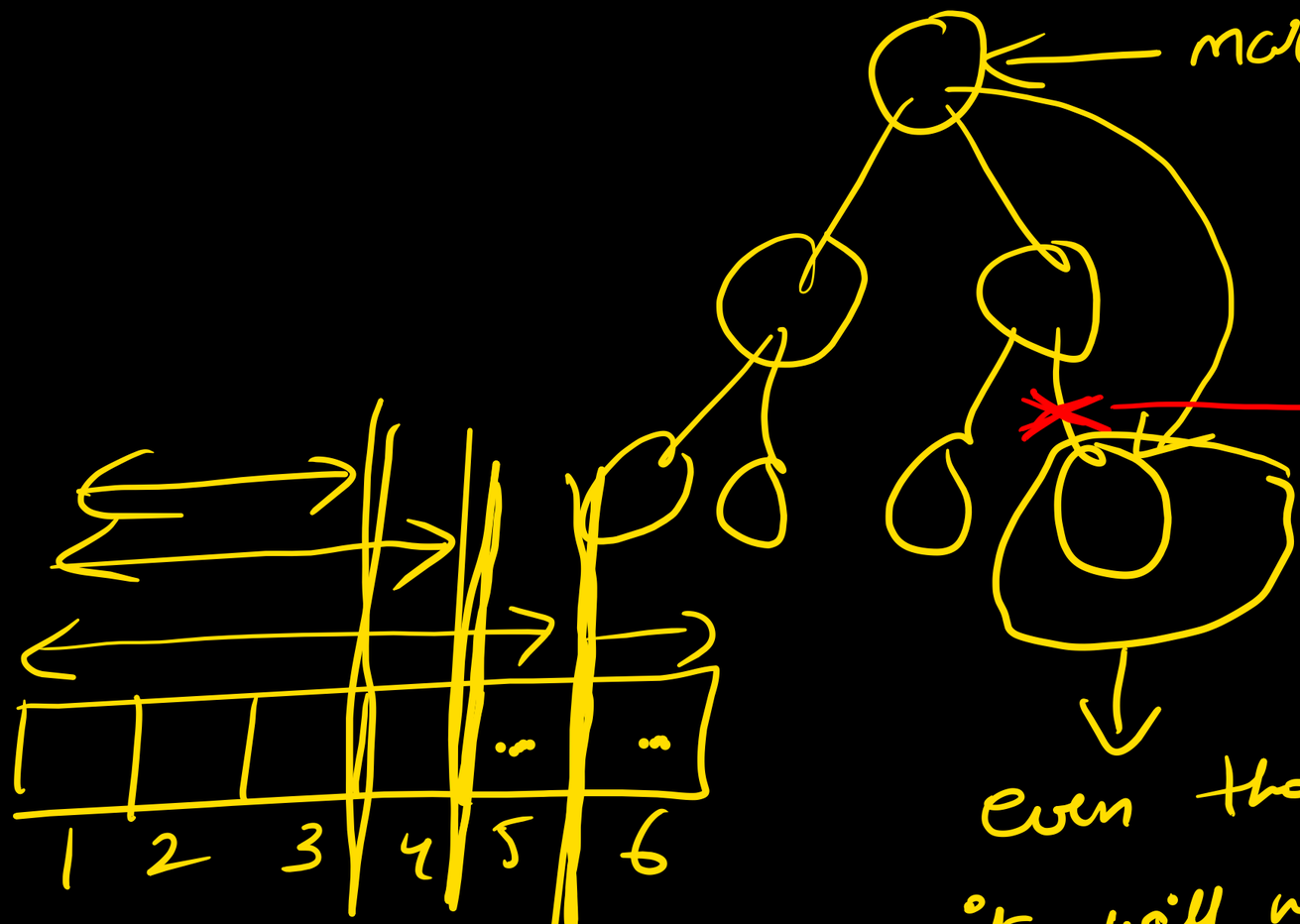
Descending order :-

① Build min heap.





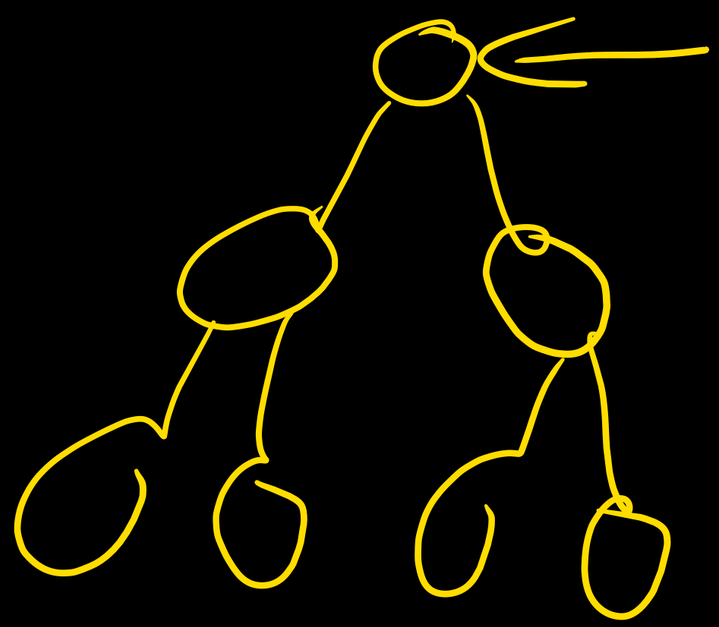
max heapify on the root.



heapsize = 6 ~~7~~ 5 3

heapsize = heapsize - 1

even though it is present in array,
it will not be considered
as a part of max heap



max heapify ⇒ { O(logn), O(log(n-1)), O(log(n-2)), ⋮, O(log1) }

nearly. (logn) are dominating most of the time.

$\leq O(n \log n)$

TC = heap sort = $O(n \log n)$ ⇒ in all case.

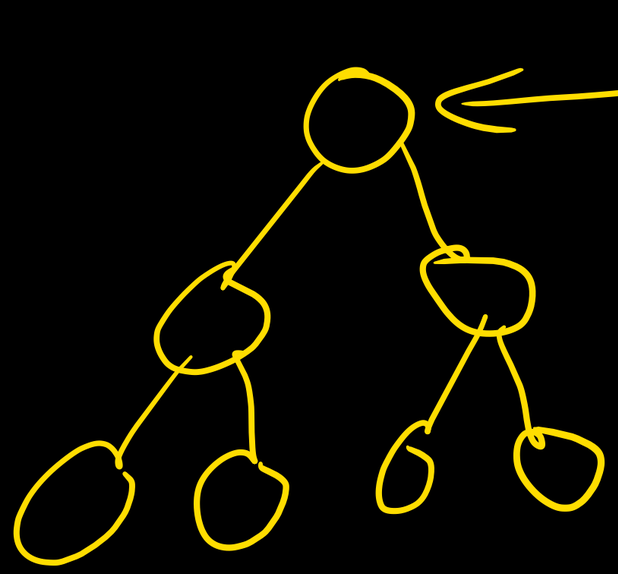
SC ⇒ max heapify ⇒ $O(\log n)$ ✓

One after the other.

In place algo:

In QS → WC → $O(n^2)$ ↓ BC → $O(n \log n)$

heap → WC BC → $O(n \log n)$.

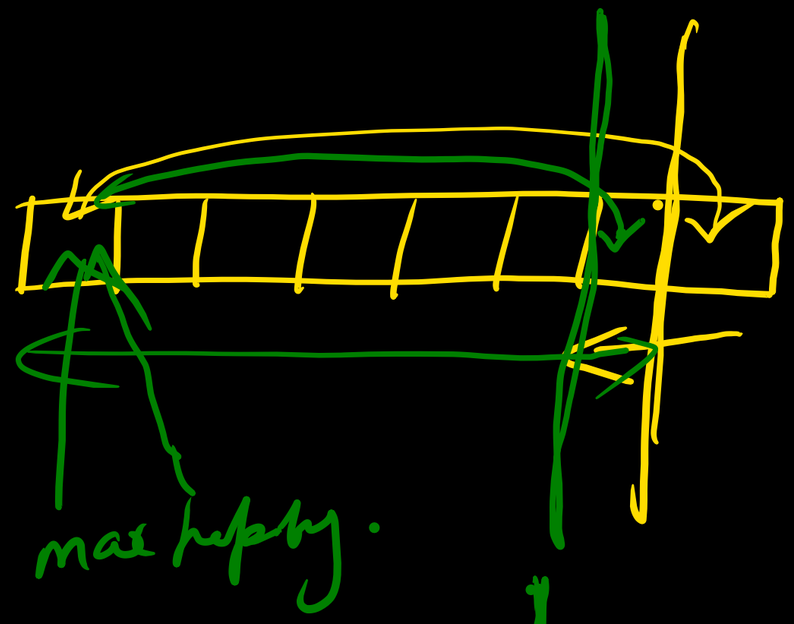


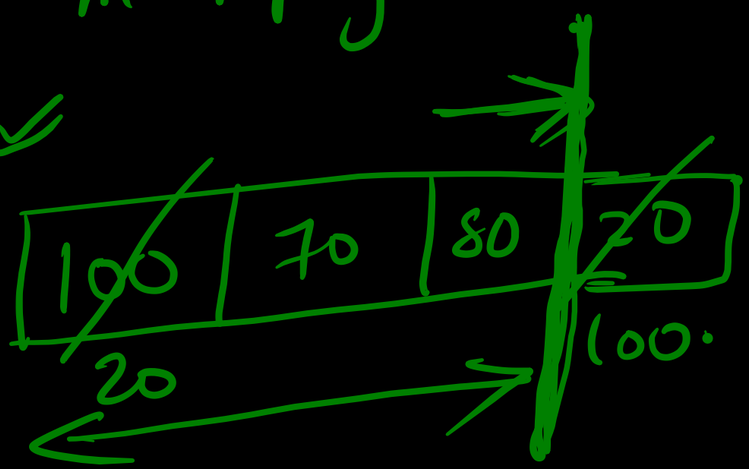
max heapify → n times.

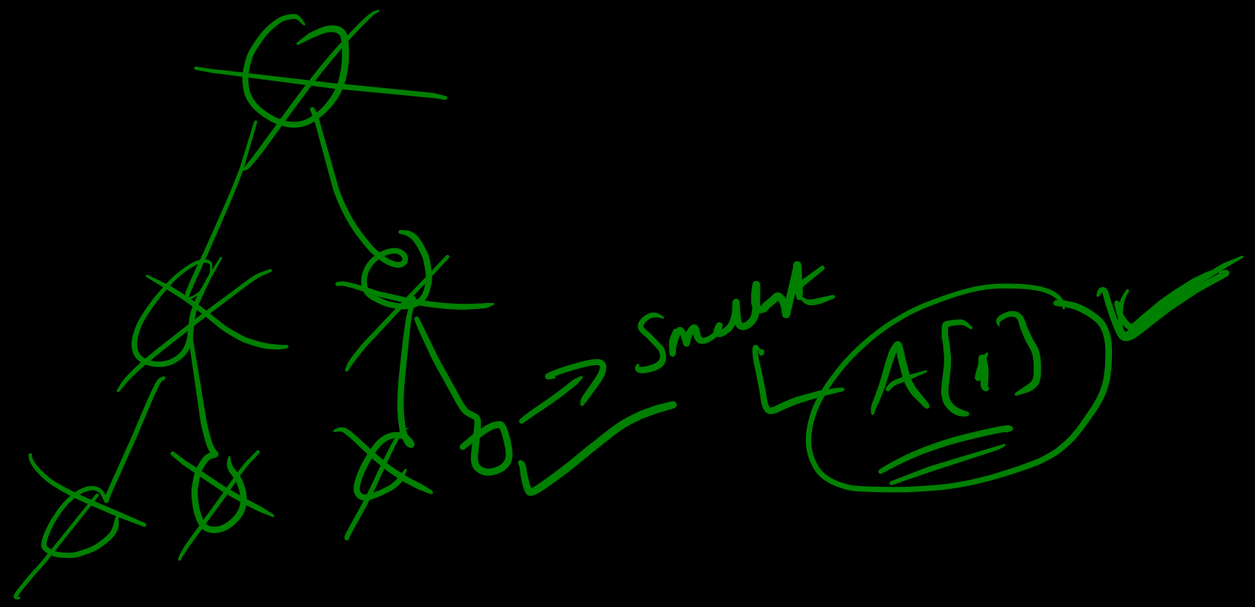
$O(n \log n) \times n$

$= O(n \log n)$

```
Heap sort (A)
{
    BUILD_MAX-HEAP(A);          (1)   Small
    for (i = A.length down to 2)
        exchange A[1] with A[i]
        A.heapsize = A.heapsize - 1
        MAX-HEAPIFY(A, 1)
}
```






Smallest
A[1]

Gate: In a heap of n elements, with the smallest element at the root, the 7th smallest element can be found in time

a) $\Theta(n \log n)$ b) $\Theta(n)$ c) $\Theta(\log n)$ d) $\Theta(1)$

min heap. → Find 7th min

↳ 6 time → delete min - $6 * \log n$

↳ see 7th element.

↳ 6 time → add them back - $6 * \log n$

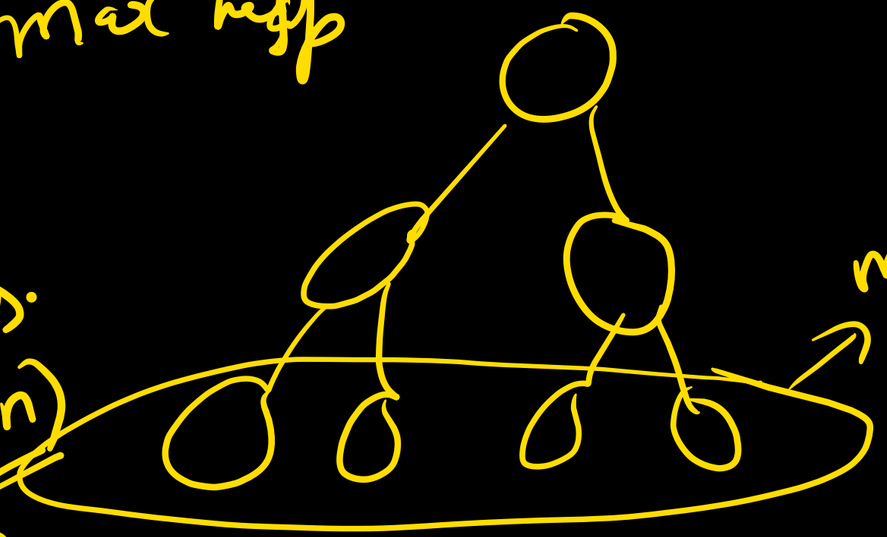
↓

insert

Gate: In a binary max heap containing n numbers, the smallest element can be found in time:

a) $\Theta(n)$ b) $\Theta(\log n)$ c) $\Theta(\log\log n)$ d) $\Theta(1)$

max heap



We have to do linear search on the leaves.

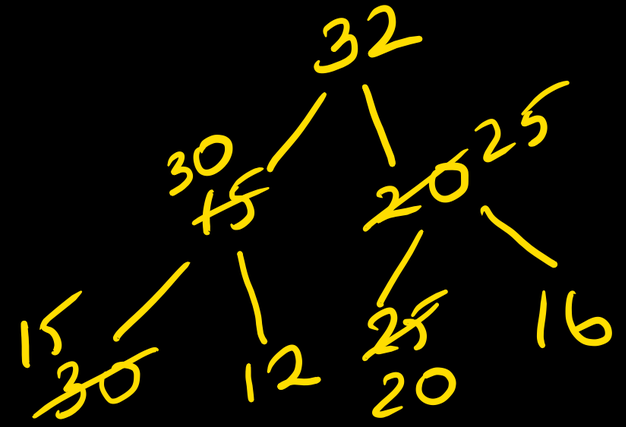
leaves $= O(n/2) = O(n)$

$TC = O(n)$

Gate → If the following elements are inserted into an max heap in that order 32, 15, 20, 30, 12, 25, 16 what is the max heap look like.

Build max heap → different answers.

Insertions → diff answers.



32 30 25 15 12 20 16.

O(nlogn)

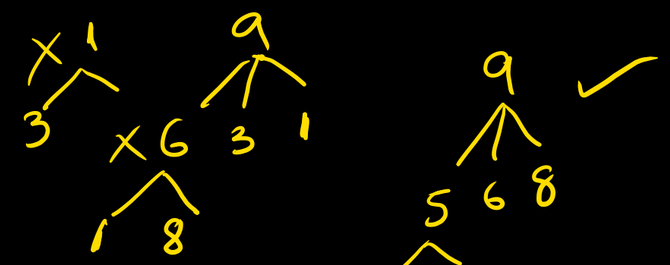
Q) which of the following is a ternary max heap?

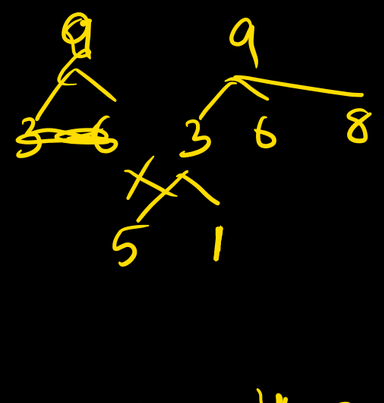
a) 1, 3, 5, 6, 8, 9

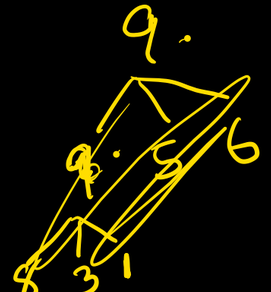
b) 9, 6, 3, 1, 8, 5

c) 9, 3, 6, 8, 5, 1

d) 9, 5, 6, 8, 3, 1







To the correct ans in above question 7, 2, 10, 4 are inserted

what is the resulting 3-ary max heap?

9 5 6 8 3 1

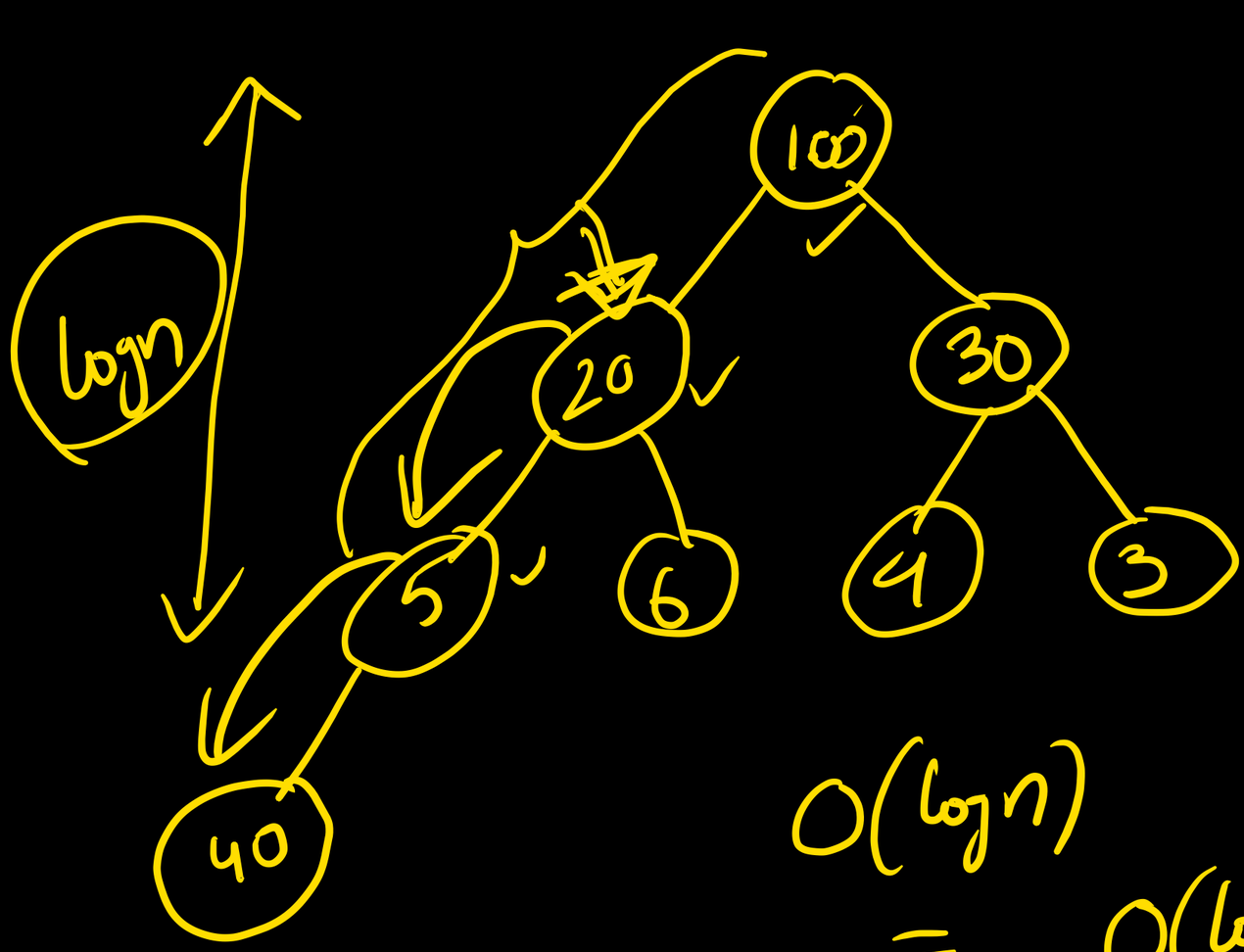
10 7 9 8 3 1 5 2 6 4

Q(log n)
O(log n)

Q: Consider the process of inserting an element in a max heap. If we can perform binary search on the path from new leaf to root to find the position of newly inserted element, the number of comparisons performed are

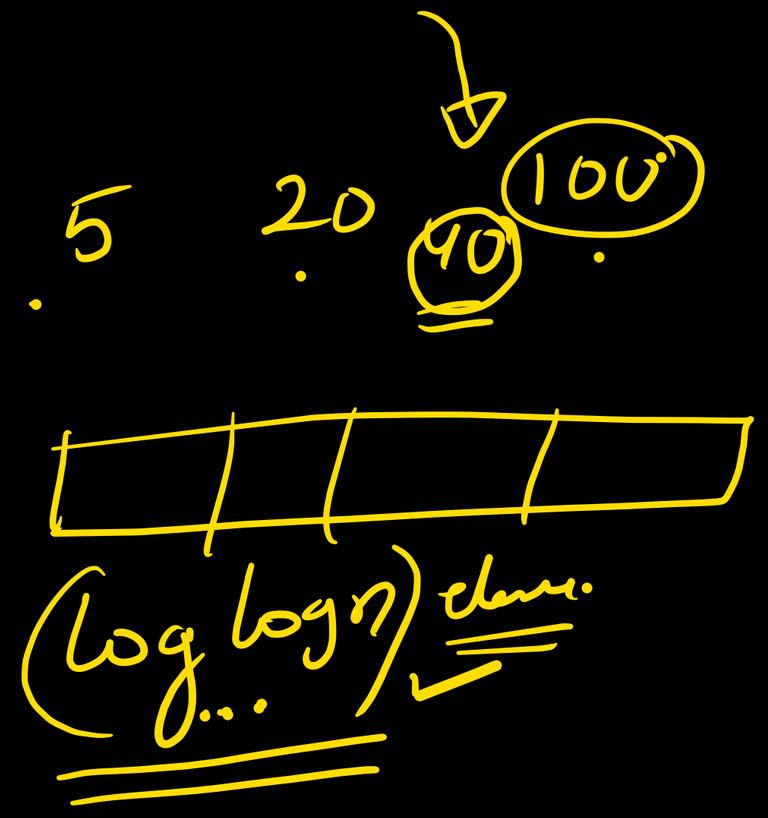
a) O(n) b) O(log n) c) O(log log n) d) O(1)

a newly inserted node, will be inserted if in the path from root to that leaf.



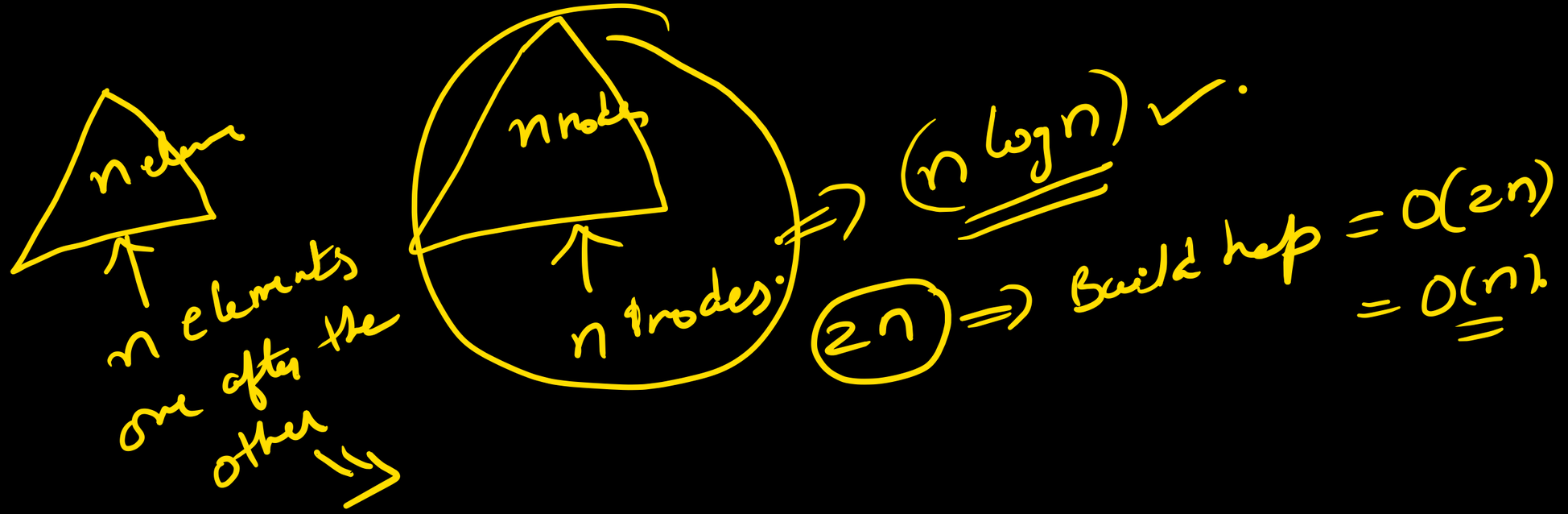
$O(\log n)$

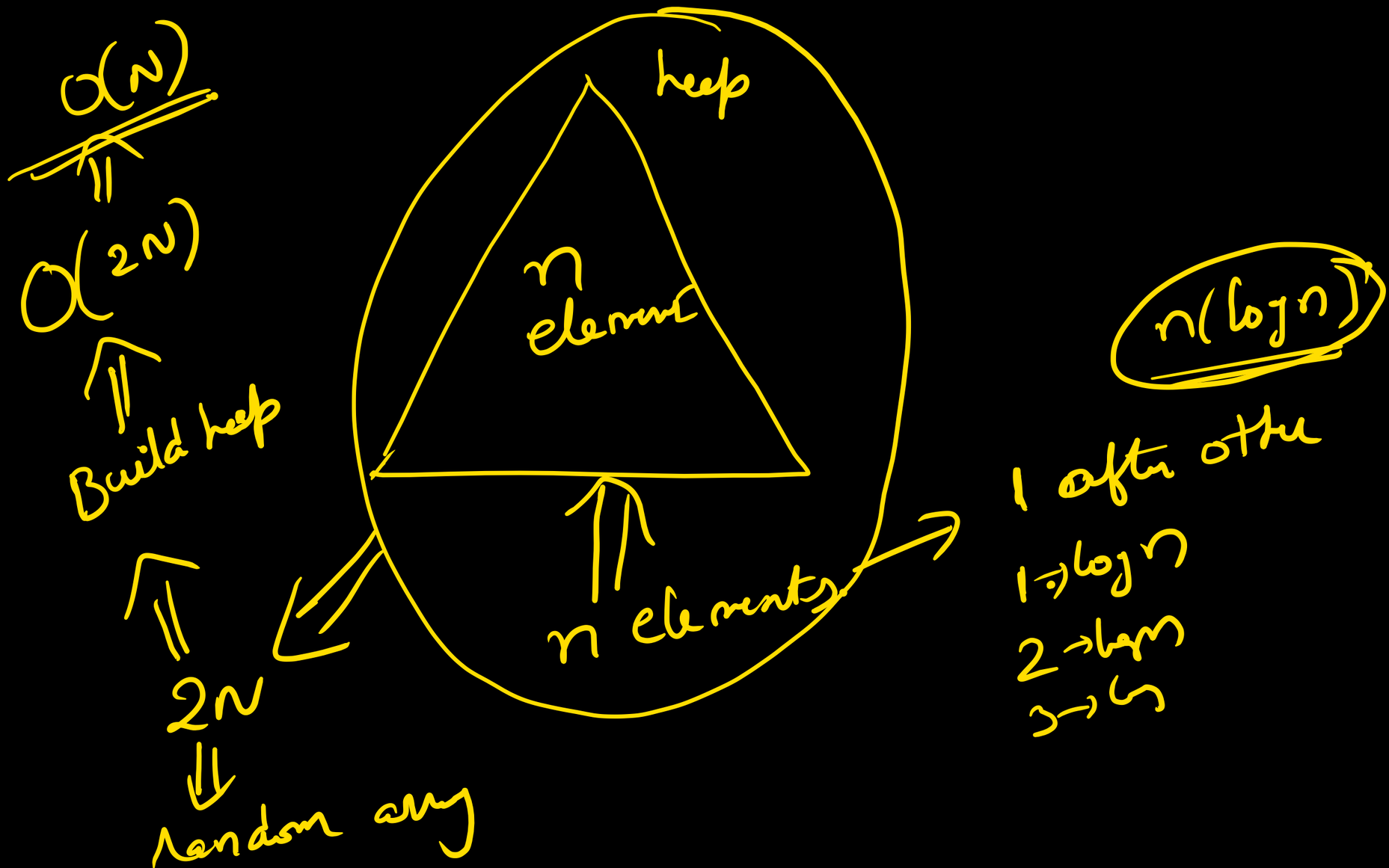
$= O(\log\log n) + O(\log n)$

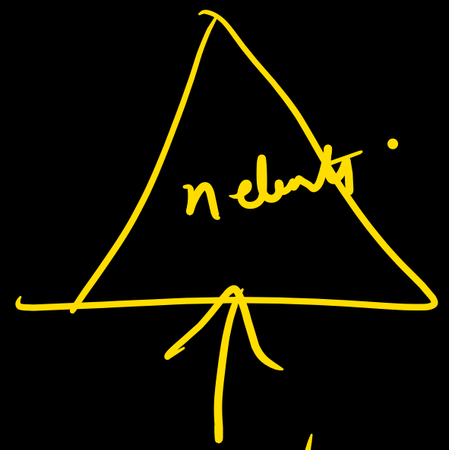


gate: We have a binary heap on n elements and wish to insert 'n' more elements (not necessarily one after another) into the heap. the total time required for this is

a) $\Theta(\log n)$ b) $\Theta(n)$ c) $\Theta(n \log n)$ d) $\Theta(n^2)$




O(N)
O(2N)
Build heap
heap
n elements
n elements
2N
random arr
n(log n)
1 after other
1 → log n
2 → log n
3 → log n



n elements

n elements → 1 after other

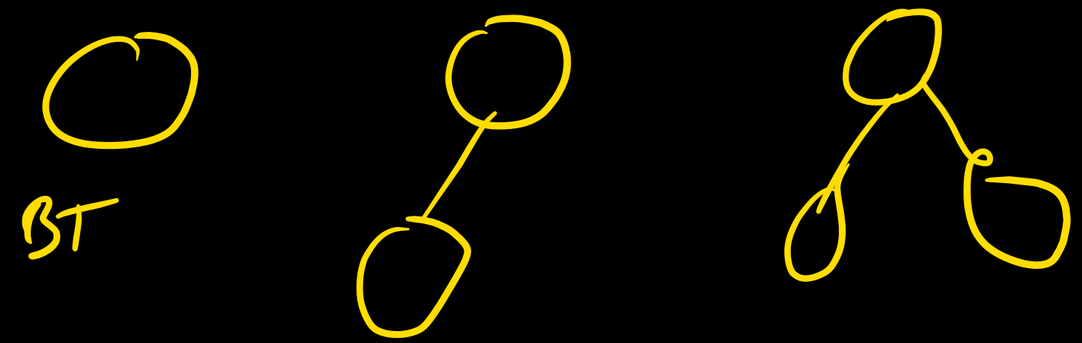
$(n \log n)$.

Break → 5 min

Heap is over.

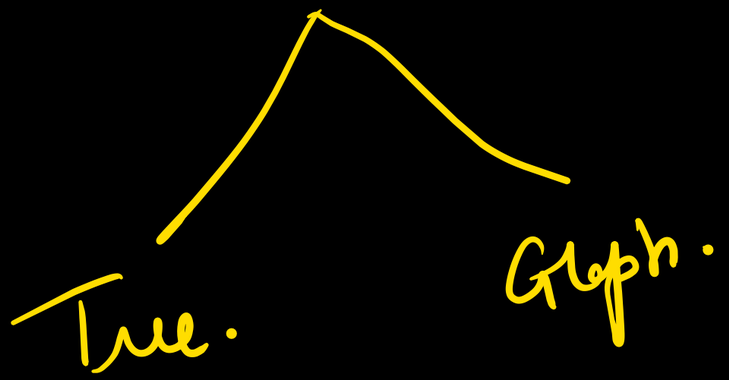
Binary trees

Binary tree : Binary tree is a tree with atmost 2 children.



Search : → generally finding one element.

Traversal : Finding all elements.

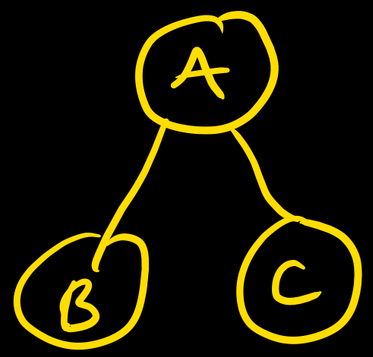


Tree traversals:

In order — left Root Right

pre order — Root left Right

post order — left Right Root.

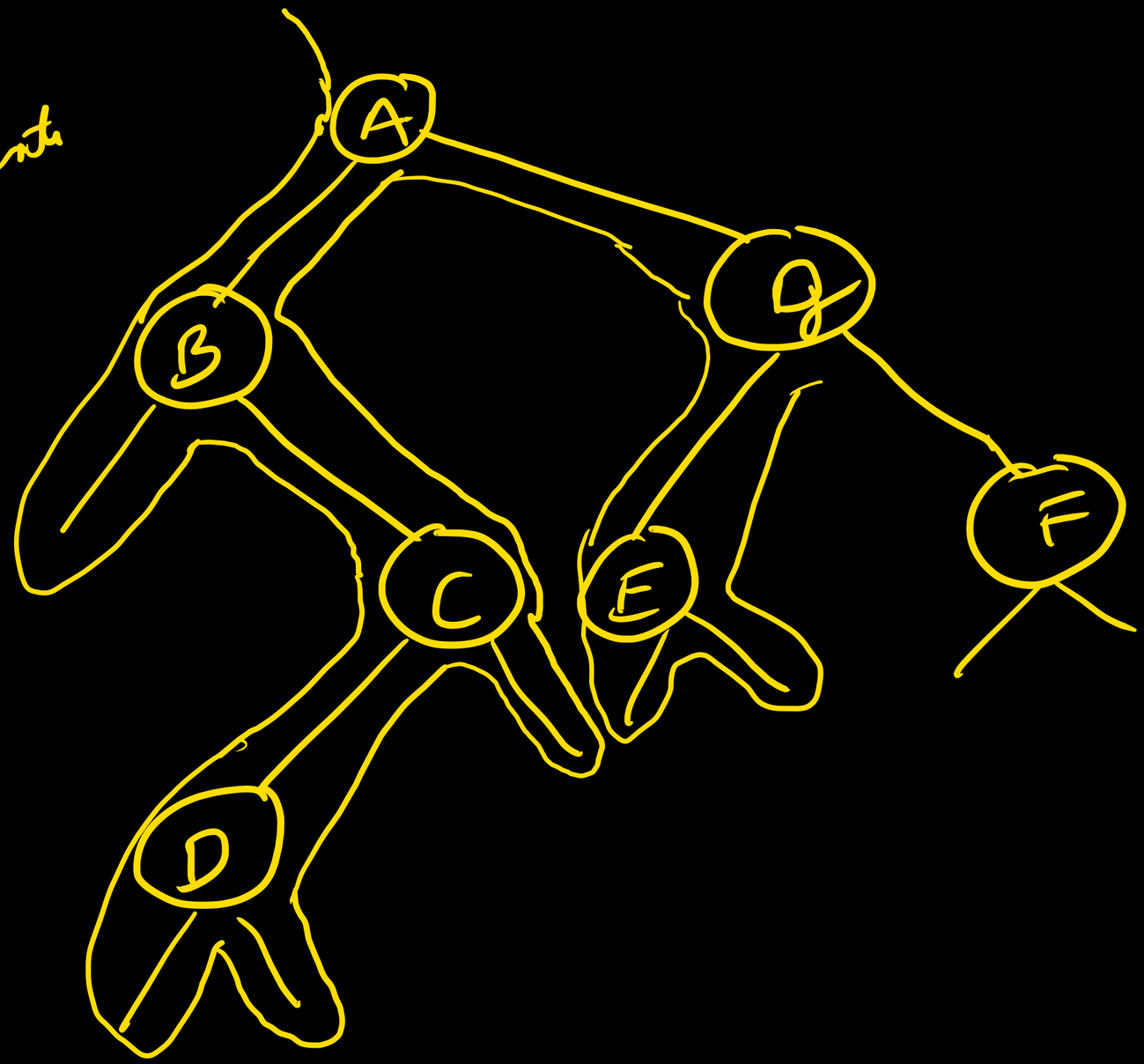


in order — B A C

pre order — A B C

post — B C A

Pre order
→ I time visit
A B C D G E F

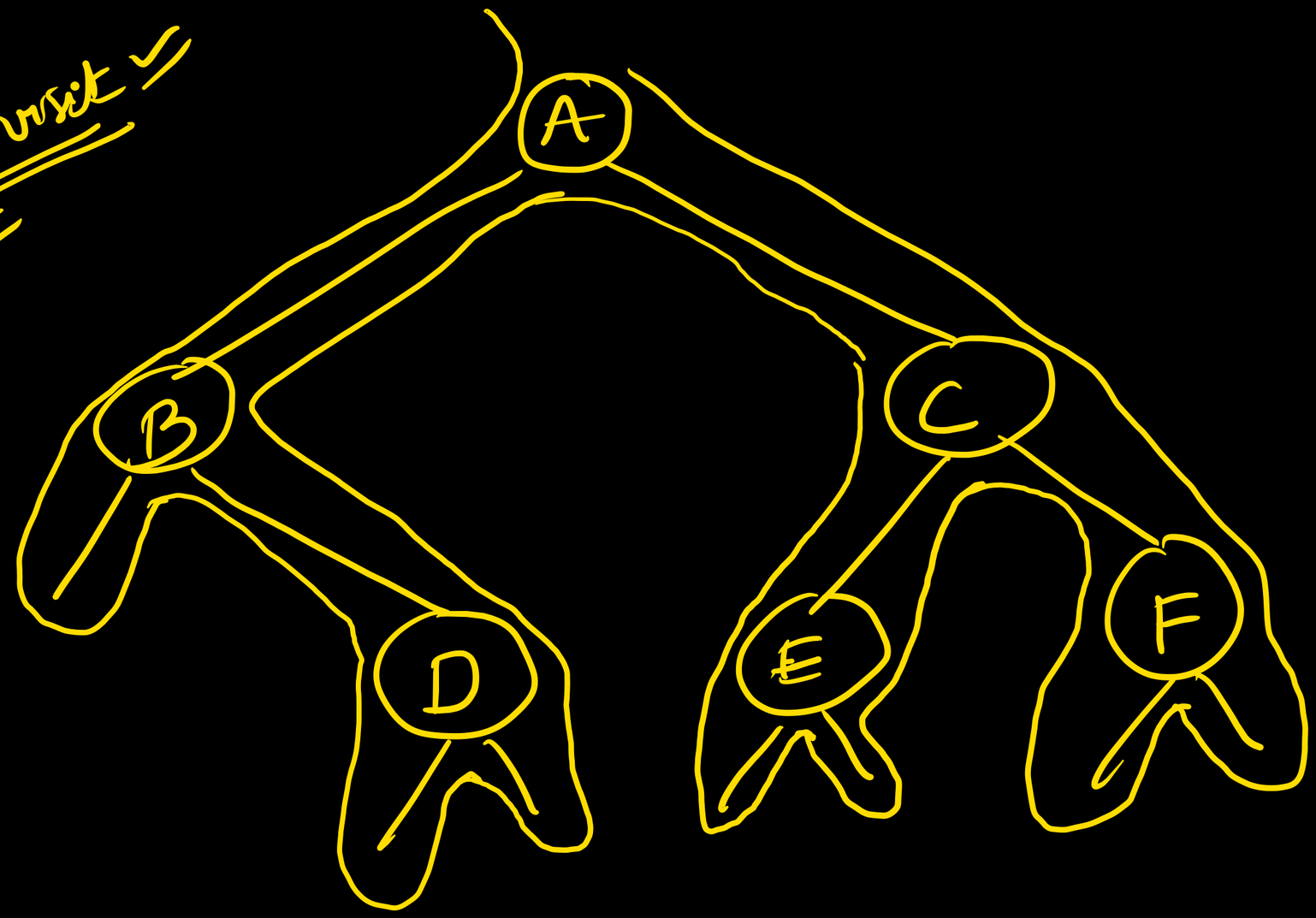


IN → L Root R
Pre → Root L R
post → L R Root.

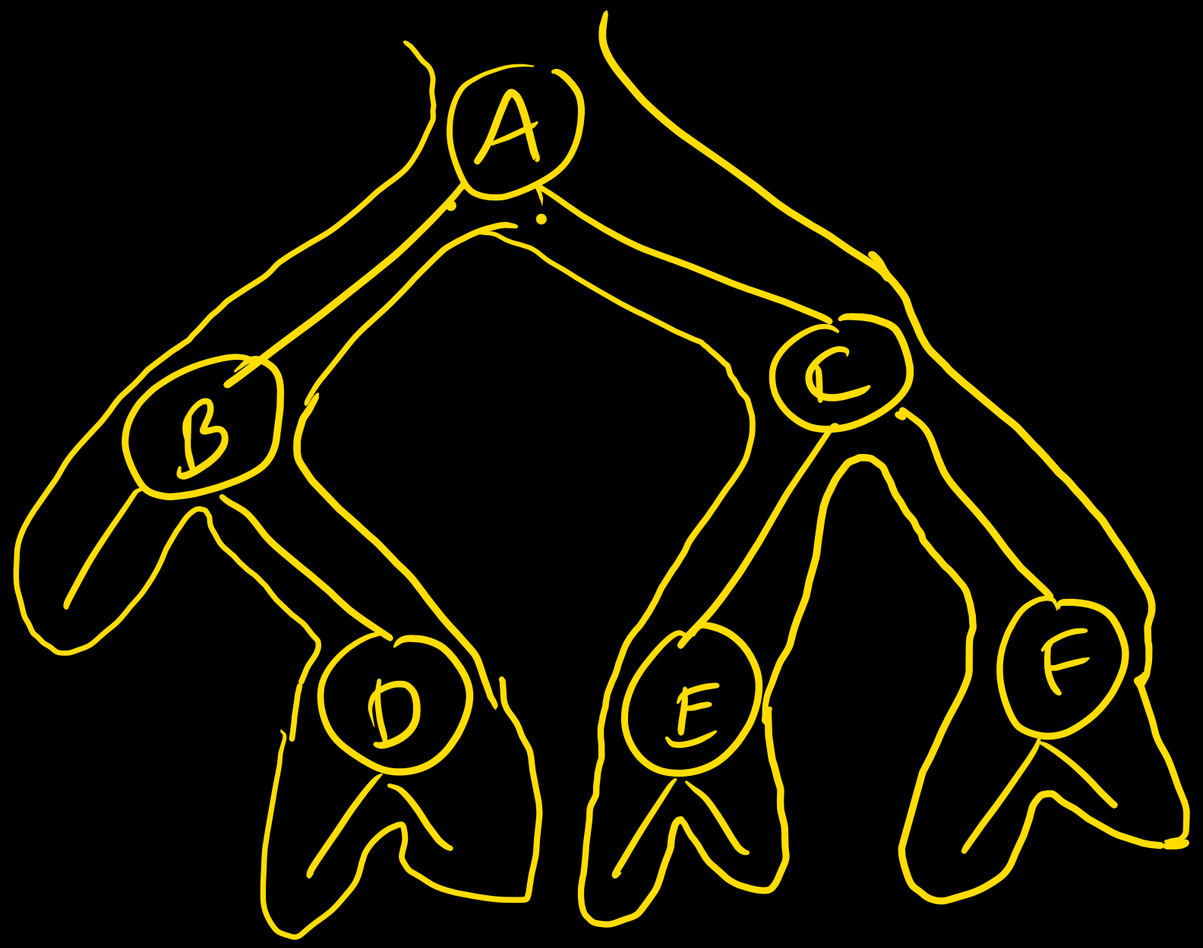
In order.

II time visit ✓

BDAECF

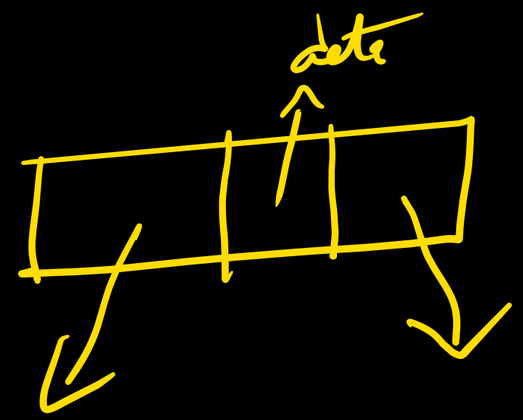


Post order:
III time

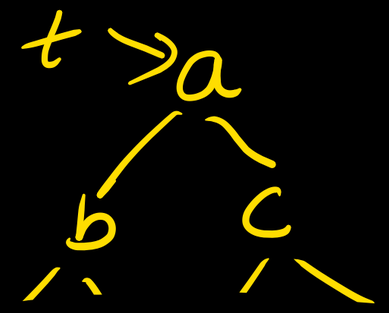


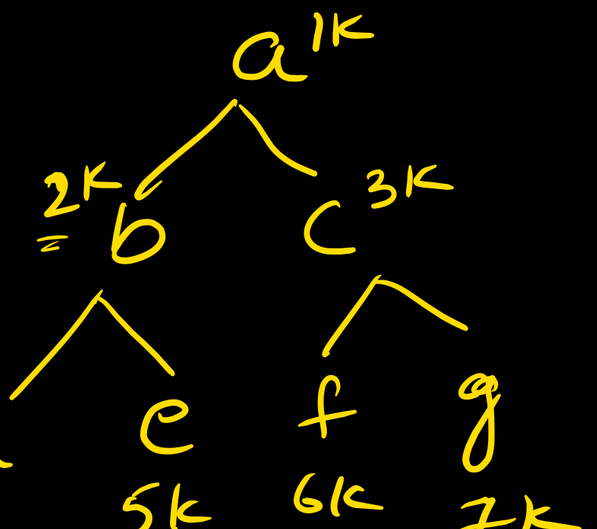
D B E F C A.

```
struct node
{ int data;
  struct node * left;
  struct node * right;
};
```

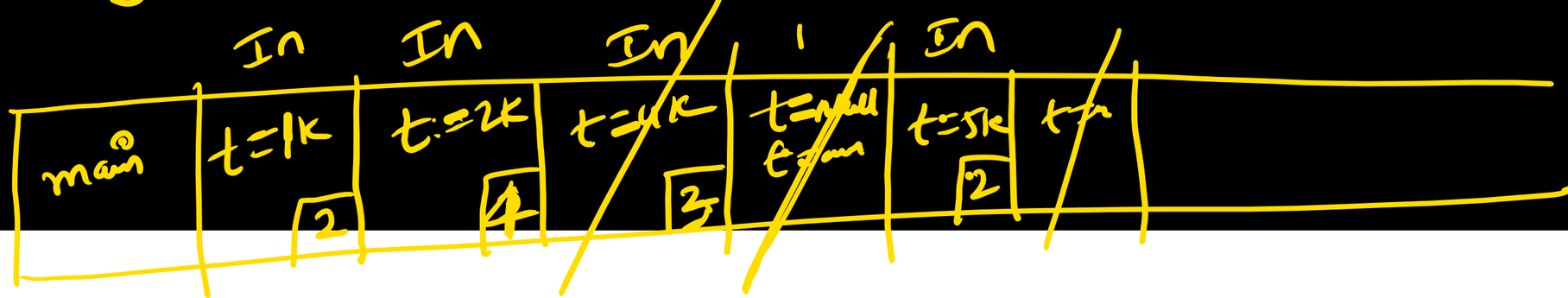


```
Void inOrder ( struct node * t)
{
  if ( t != NULL)
  {
  1 InOrder (t -> left);
    2 Printf ("%d", t -> data)
    3 InOrder (t -> right);
  }
}
```

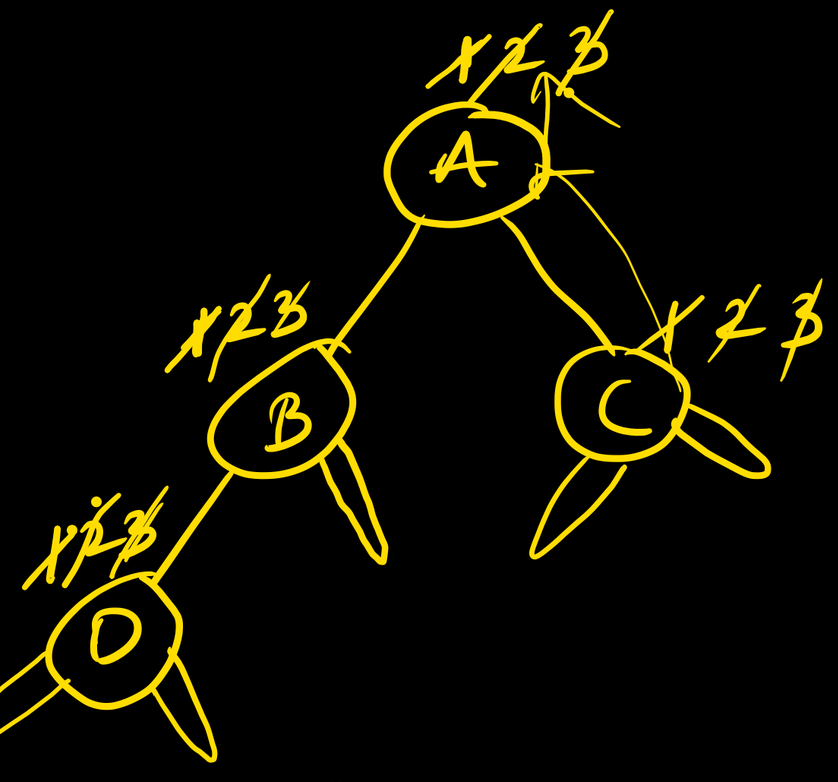




d b e



```
Inorder
{
   1) Inorder(L)
   2) Pf(data)
   3) Inorder(R)
}
```



Inorder. DBAC

```
Preorder
{
    Pf( )
    Preorder(L)
    Preorder(R)
}
```