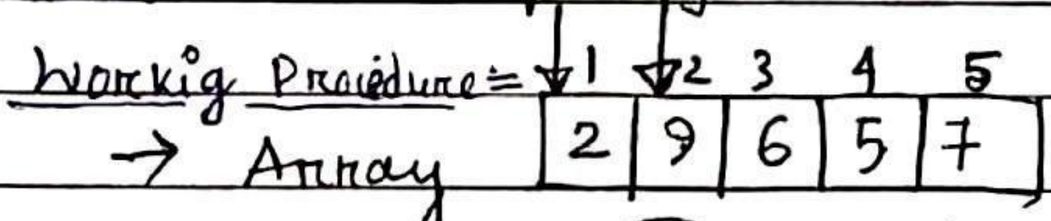
Sorting Techniques

- Insertion sort algorithm and analysis =

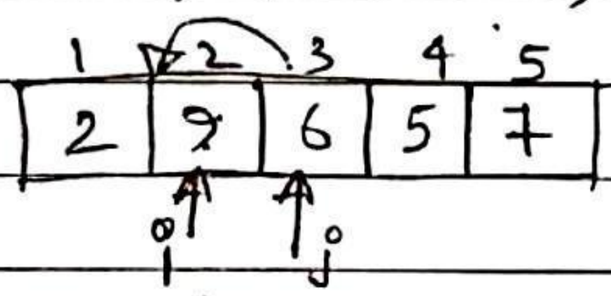
```
Insertion-sort (A)
{
for(j=2 to A.length)
{
    Key = A[j];   // insert A[j] into stored sequence
                     A[1 .... j-1]
    i = j-1
    while(i>0 and A[i] > Key)
      { A[i+1] = A[i].
        i = i-1 }
    A[i+1] = Key
}
```

Working Procedure =

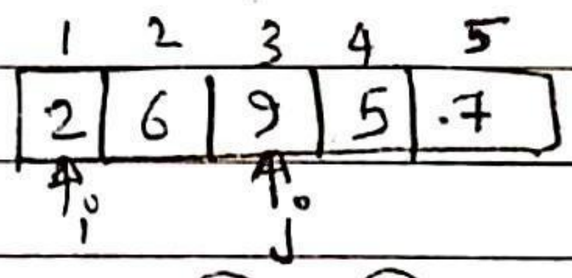
→ Array



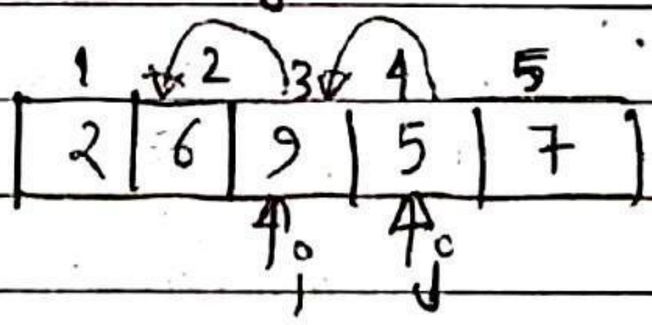
(1)



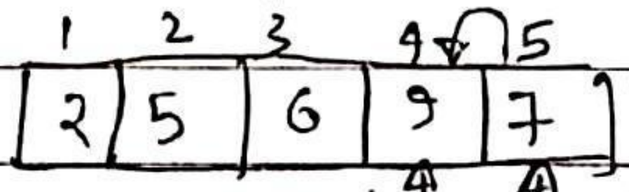
(2)



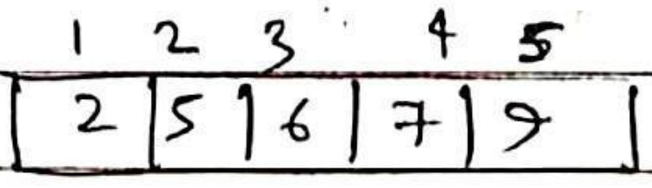
(3)



(4)



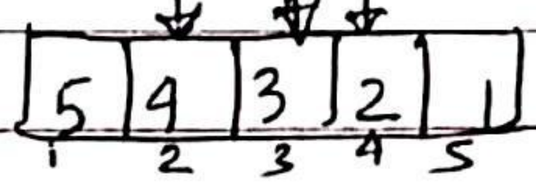
(5)



array (sorted)

atlantis
Date
Page

• Time Complexity in worst case = $O(n^2)$



→ When reversed array

	Comparisons	movement
Index		
2 —	1 +	1 = 2 = 2(1)
3 —	2 +	2 = 4 = 2(2)
4 —	3 +	3 = 6 = 2(3)
5 —	4 +	4 = 8 = 2(4)
:		
n —	(n-1) +	(n-1) = 2(n-1)

$T(n) = 2(1) + 2(2) + 2(3) + 2(4) + \cdots + 2(n-1)$

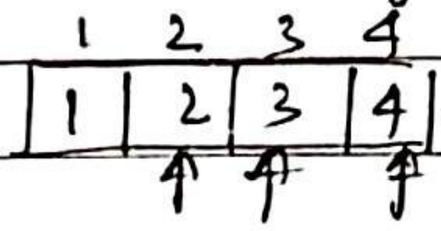
$= 2\left(1+2+3+4+\cdots+(n-1)\right)$ (A.P) $\quad \frac{n(n+1)}{2}$

$= 2\,\frac{(n-1)\,(n-1+1)}{2}$

$= n^2 - n$

$T(n) = O(n^2)$.

• Time Complexity In Best case = ~~$\Theta(n)$~~ $\Omega(n)$.



→ when array ~~is~~ already sorted.

Index		Comparisions		movement	
2	—	1	+	0	= 1
3	—	1	+	0	= 1
4	—	1	+	0	= 1
:					
n	—	1	+	0	= 1

$T(n) = 1+1+1+1+\cdots+1 = \Omega(n-1) = \Omega(n)$.

atlantis
Date
Page

- Space Complexity = O(1)

(key, i, j) =
(need only 3 variable)

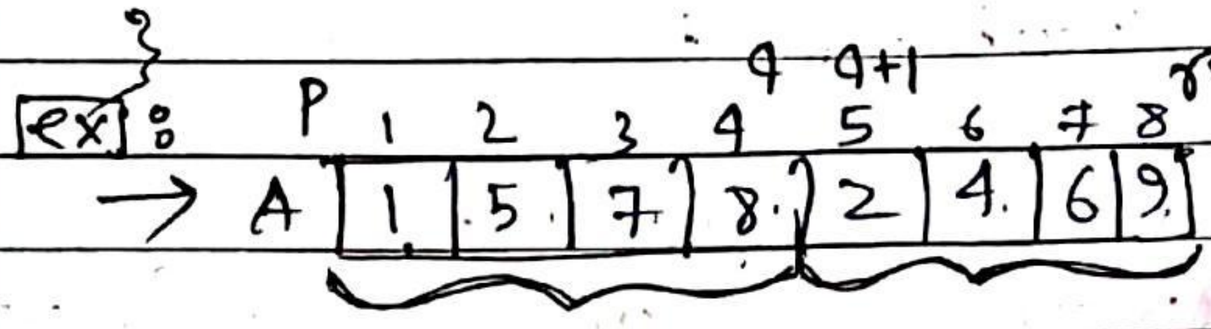
→ When we need constant space to sort any given list, such algo also call Inplace Algo.
So, Insertion sort also called Inplace Algo.

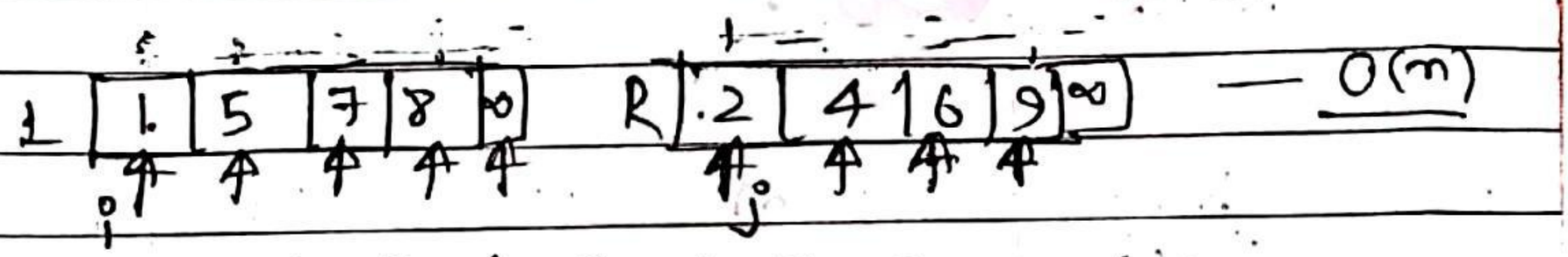
When used	Comparisons	movement	
Binary Search	O(log n)	n	= $O(n^2)$ (T.C)
double linked list	O(n)	O(1)	= O(n) (T.C)

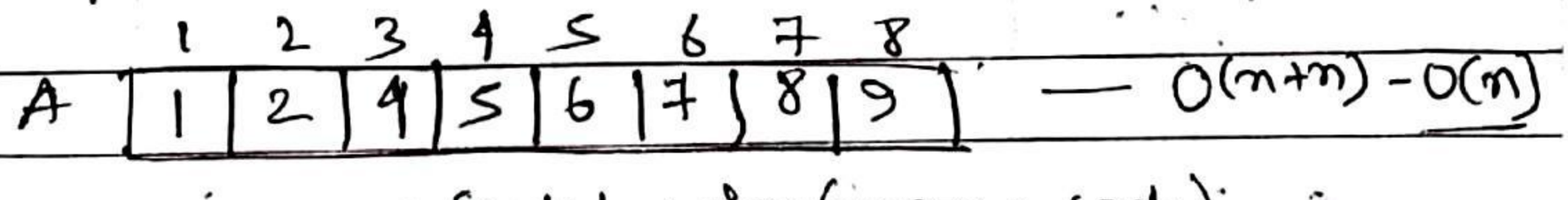
- Merge sort algorithm and analysis =

```
MERGE (A, P, q, r)  [Merge procedure]
{
  n1 = q - p + 1;
  n2 = r - pq;
  let L [1 ... n+1] and R [1 to n2+1] be new arrays
  for (i = 1 to n1)
     L[i] = A[p+i-1]

  for (j = 1 to n2)
     R[j] = A [q+j];

  L[n1+1) = ∞;
  R(n2+1) = ∞;
  i = 1, j = 1;
```

```
for (k=p to r)
    if (L[i] ≤ R[j])
        A[k] = L[i]
        i = i+1;
    else
        A[k] = R[j]
        j = j+1;
```







Sorted using (merge sort)

Total time taken by merge sort = O(n).
Space complexity = O(n).

⇒ ~~Merge program is also called~~
Merge program is also called out of place prob

→ Merge procedure is also called out of place procedure.

atlantis
Date
Page

[Merge-sort]

Merge-Sort(A,P,r) — T(n)

{

if P<r

q = ⌊(P+r)/2⌋

merge-sort(A,P,q) — T(n/2)

merge-sort(A,q+1,r) — T(n/2)

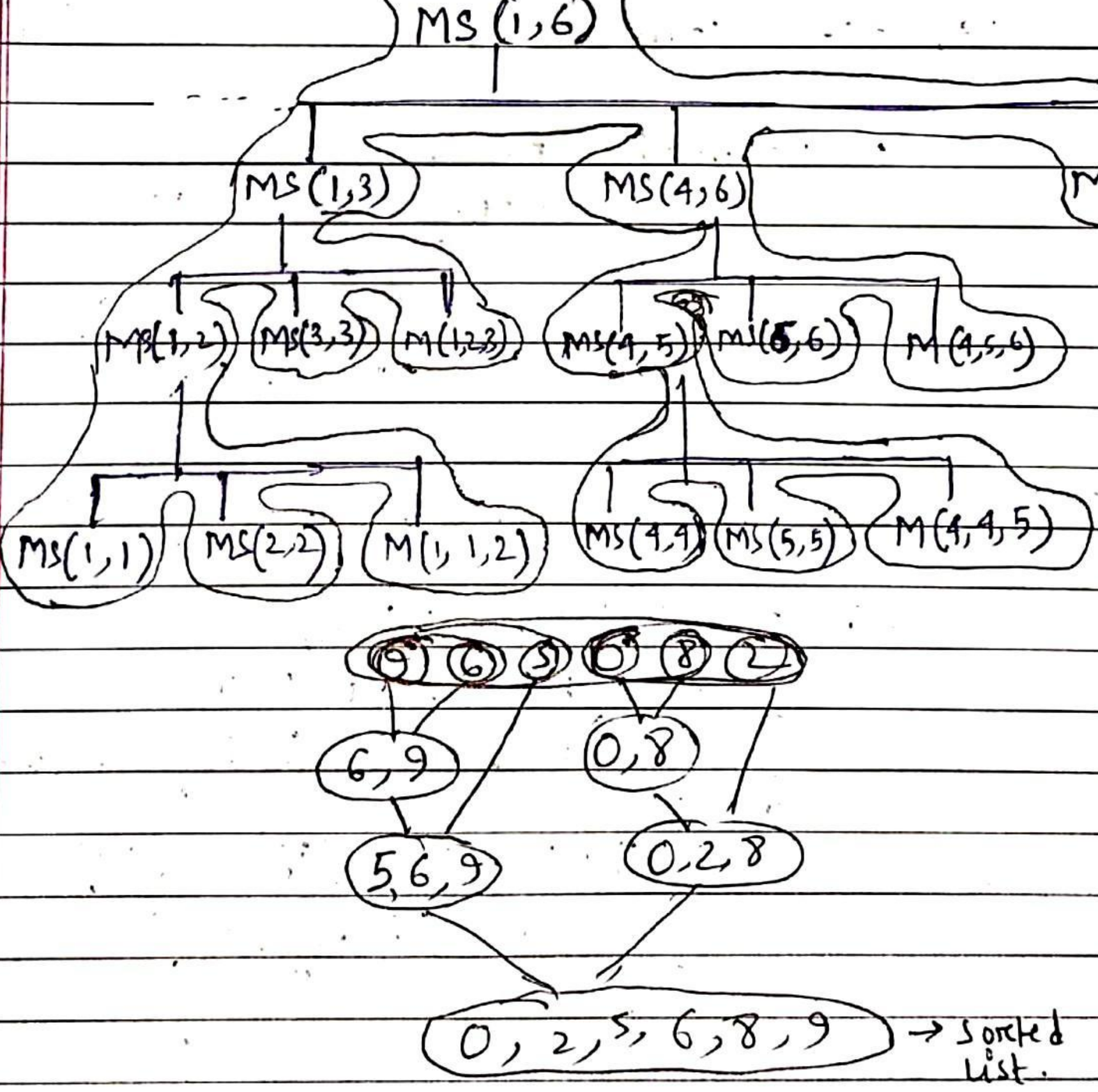
merge(A,P,q,r) — O(n)

}

Ex:

	1	2	3	4	5	6
A	9	6	5	0	8	2

P r



atlantis
Date
Page

✓ Space Complexity required by the merge sort - O(n)

4)	4	8 6 11 12 13
3)	3	7 8 10 14 15
2)	2	9 16
1)	1	

6 - 4 level

$n - (\lceil \log n \rceil + 1)$ levels.

size of stack = $(\lceil \log n \rceil + 1)\, K$

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

So, total space required to compare merge sort.

for merge procedure $= O(n)$

stack $= O(\log n)$

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

$= O(n)$.

✓ Time complexity required by the merge sort $= O(n \log n)$

$T(n) = 2 * T(n/2) + O(n)$

$a=2, b=2, K=1, P=0$

$a = b^K$ (Using masters theorem)

ii) a)

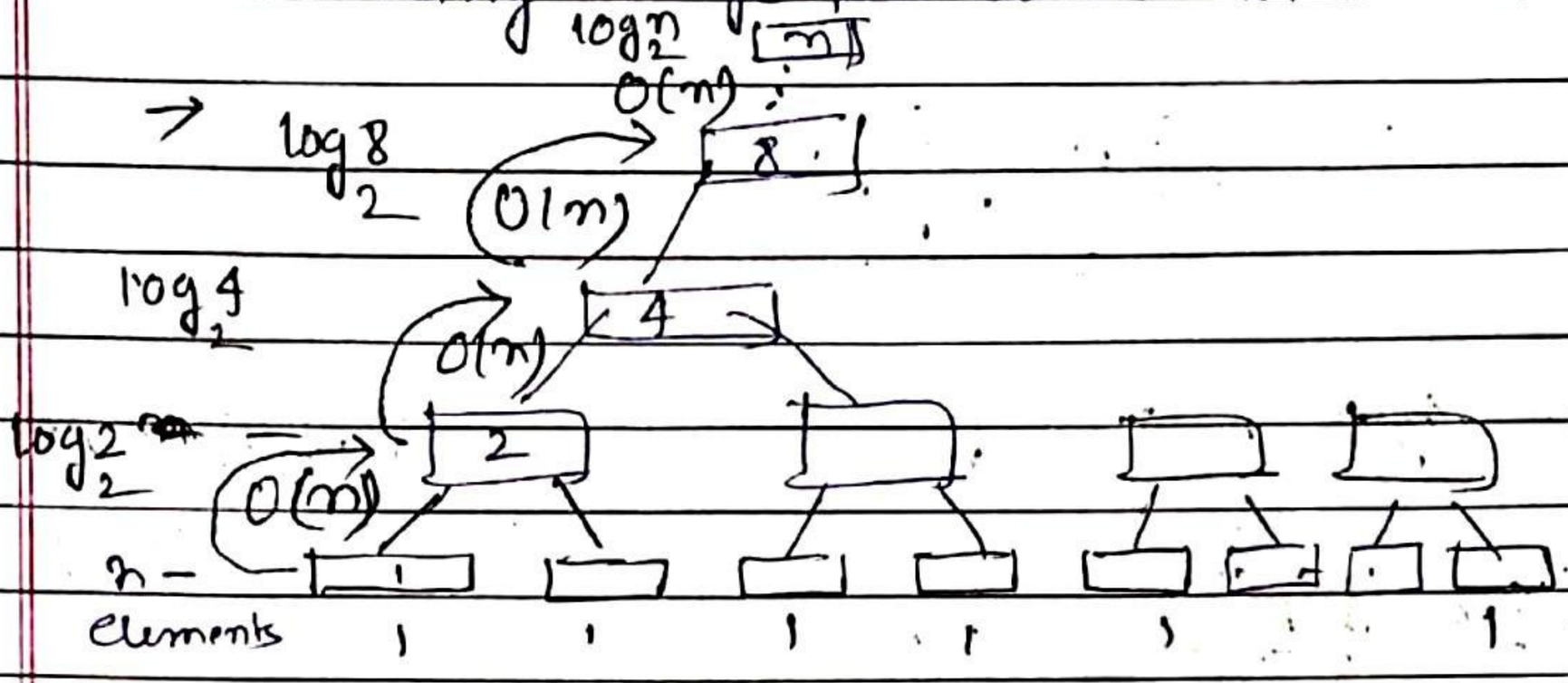
$T(n) = \theta(n^{\log_2^2} \log^{0+1} n)$

$= \theta(n \log n)$.

atlantis
Date
Page

Q-1 (2-Way merging)

Given 'n'-elements, merge them into one sorted list using merge procedure. What is time complexity -



Total time = $O(n) * O(\log n)$

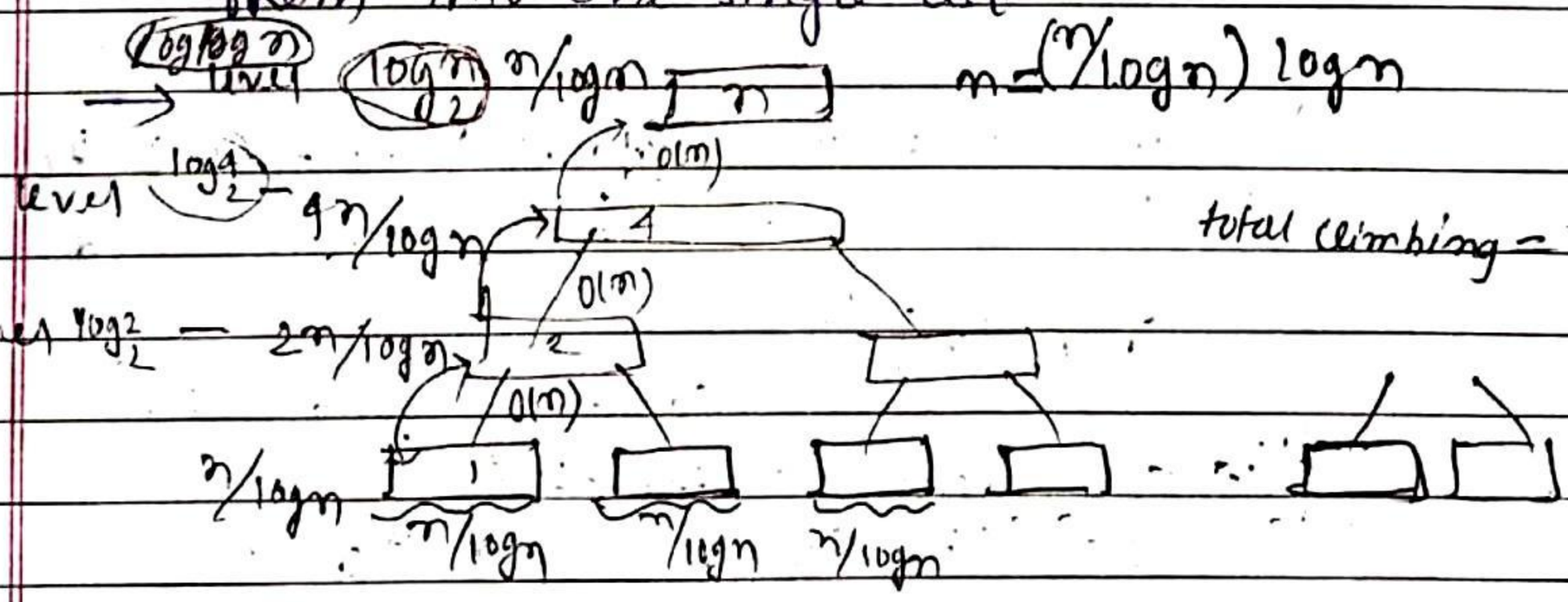
$= \boxed{O(n \log n)}$

$O(n)$ → Work done each level.

no. of levels → $O(\log n)$

Q-2

Given "$\log n$" sorted lists of size "$n/\log n$", What is the total time required to merge them into one single list -



~~total element~~

$\log n * n/\log n = n$ element each level.

~~time comp~~ Time complexity = $O(n \log \log n)$.

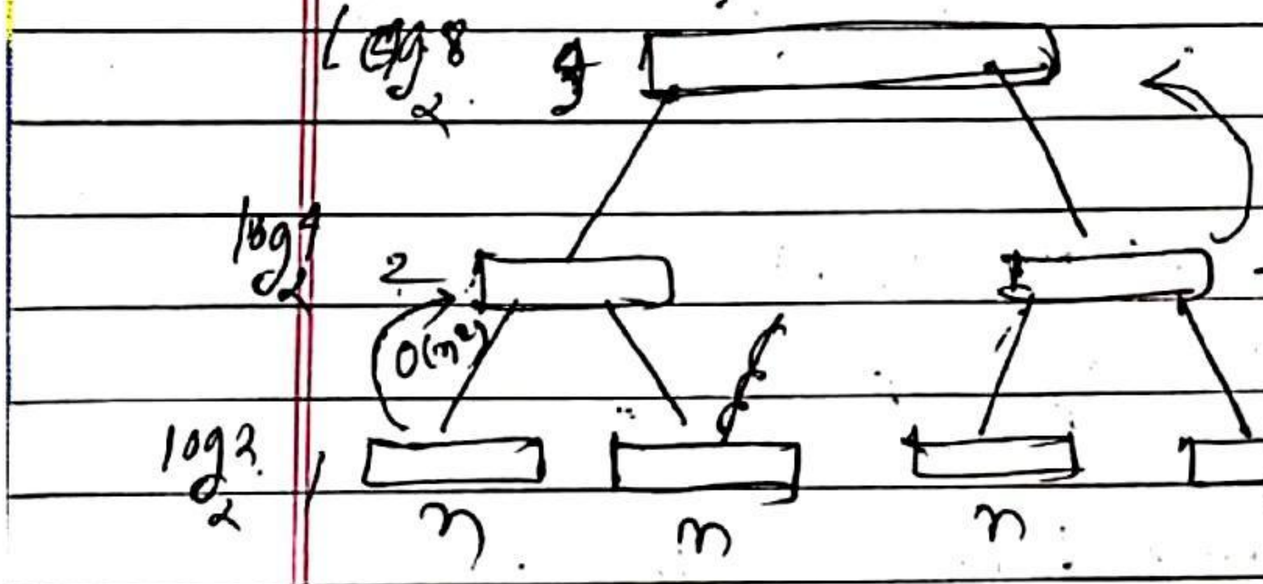
atlantis
Date
Page

Q-3.

"n" strings each of length "n" are given then what is the time taken to sort them

→

time = $(\log n)$ n



$= O(\log n) \times O(n^2)$

$= O(n^2 \log n)$

$= O(n) \times O(n) + O(n^2)$

$= O(n^2)$

$= O(n^2)$ efficient

Total time Complexity $= O(n^2 \log n)$.

Q-4

(1) Merge sort uses Divide and Conquer.

(2) m and n total time taken to merge m, n two sorted list —

$O(m+n)$

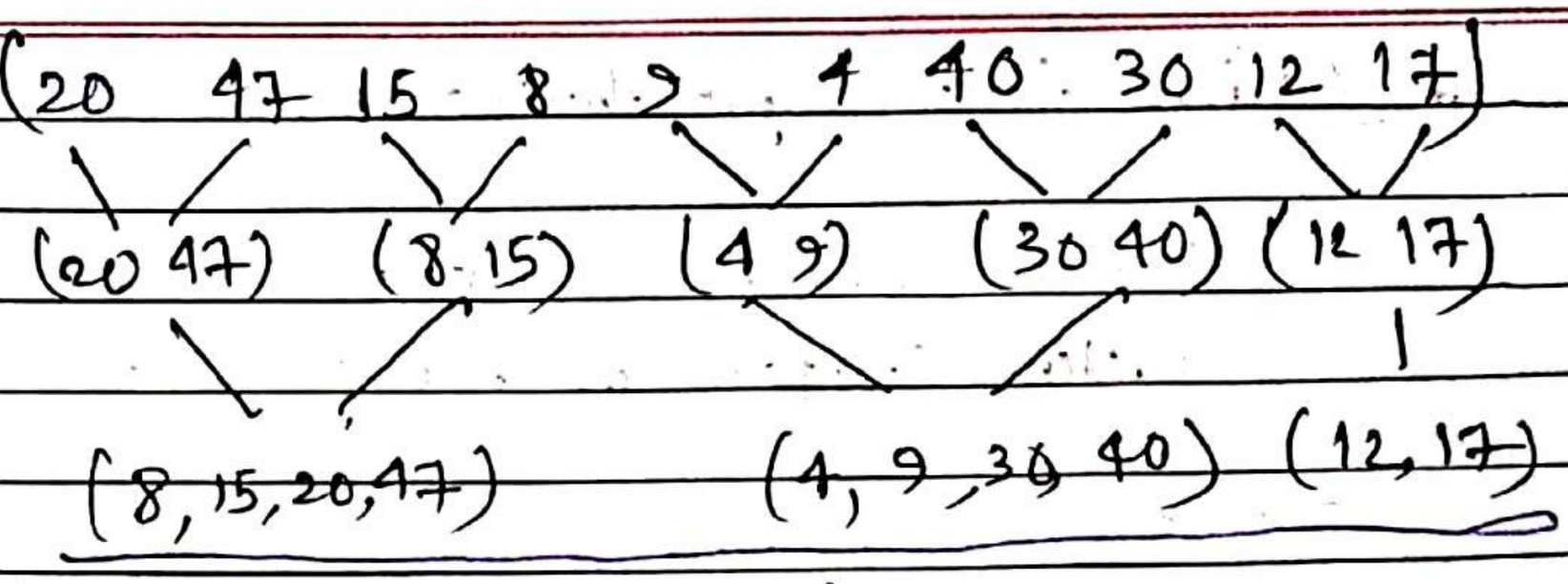
(Compare and copy)

(3) Algo (2 way merging) ~~sorted in~~

20 47 15 8 9 4 40 30 12 17

What will be the order of elements after 2nd pass —

→ (20 47 15 8 9 4 40 30 12 17)



↳ After 2nd pass this is the order we get.

• Quick Sort Algorithm =

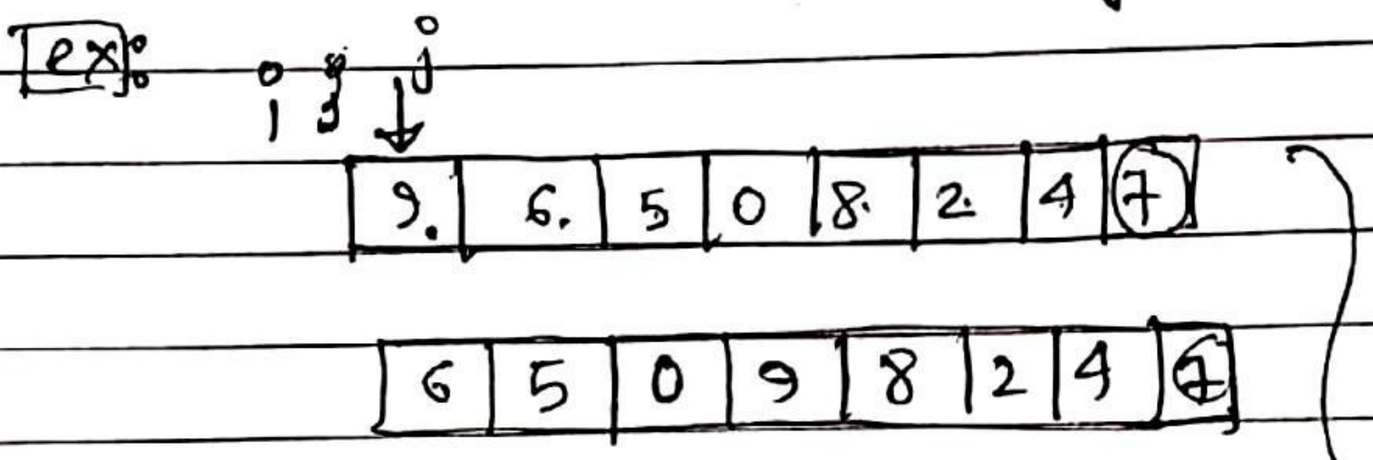
→ For Smaller ~~no. of~~ Input, Quick sort ~~o~~ run faster Compare to merge sort.

→ Quick Sort and merge sort both follow divide and conquer method.

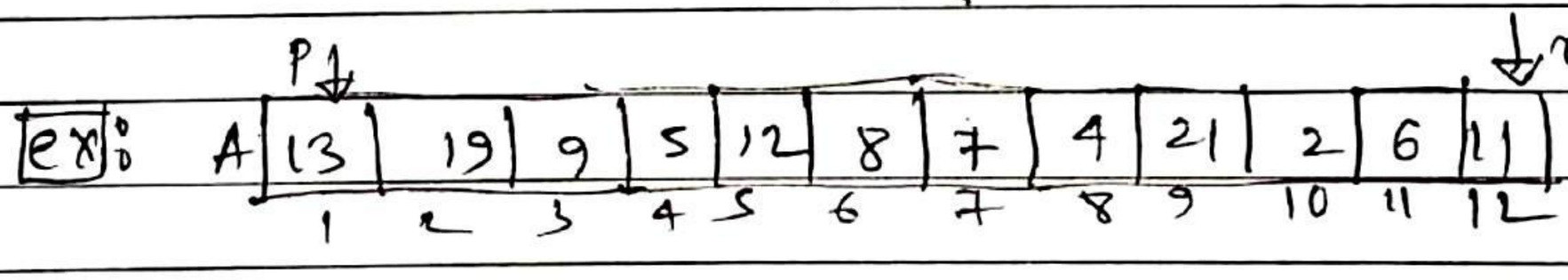
• Partition Algorithm =

Time taken by the partioning algorithm = O(n).

ex:



ex:



PARTITION (A, P, r)

{

x = A[r]

i = P-1;

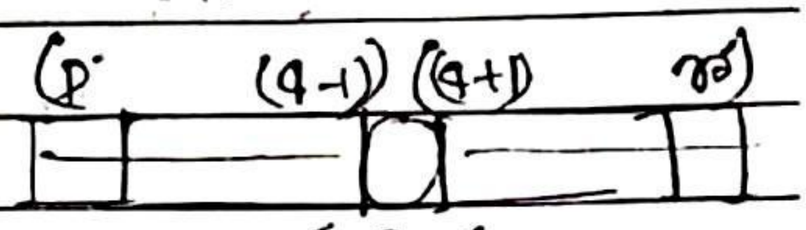
for (j=p to r-1)

{ i = i+1;

exchange A[i] with A[j] }

```
if (A[j] <x)
  { i = i+1;
    exchange A[i] with A[j]
  }
}
exchang A[i+1] with A[r]
return i+1;
}
```

QUICKSORT (A, P, r) — T(n)




```
{
  if ( P<r)
  {
    q = PARTITION (A, P, r); -O(n)
    QUICKSORT (A, P, q-1); — T(n/2)
    QUICKSORT (A; P, q+1); — T(n/2)
  }
}
}
```

ex:

	1	2	3	4	5	6	7
A	5	7	6	1	3	2	4

1	2	3	4	5	6	7

QS (1,7)

P(1,7) → q = 4

QS (1,3) QS(5,7)

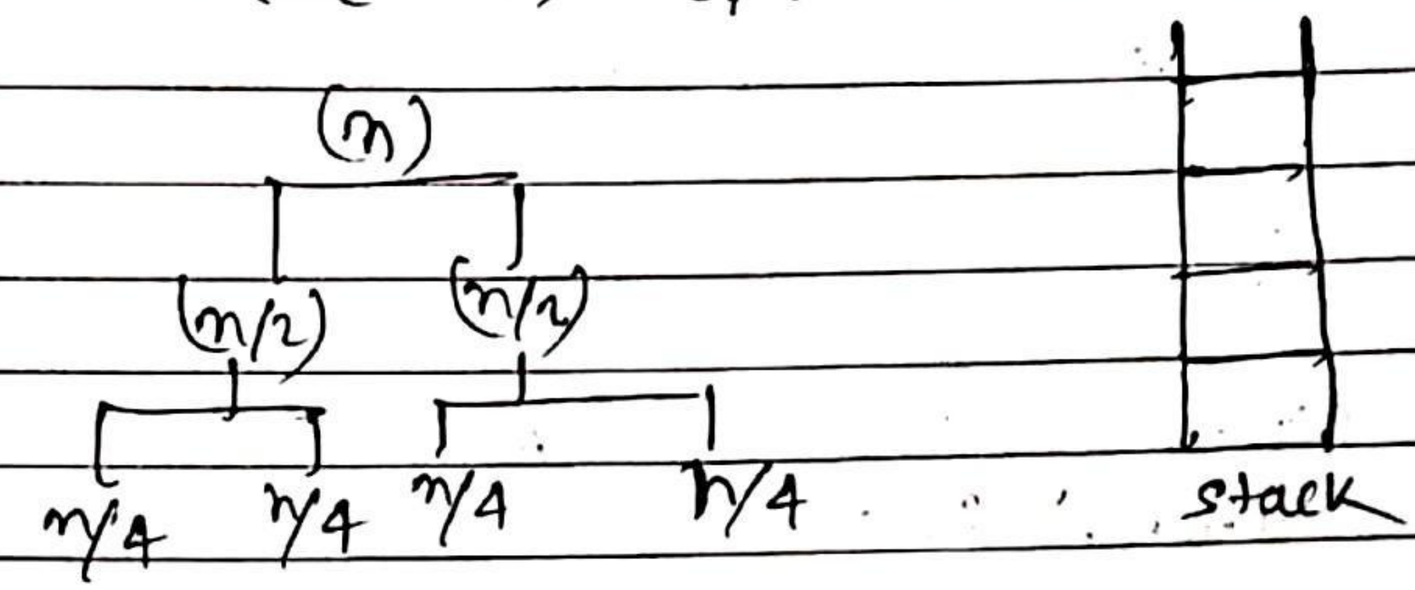
P(1,3) → q = 2 QS(1,1) QS(3,3)

P(5,7) → 5 QS(5,4) QS(6,7)

P(6,7) → 7 QS(6,6) QS(8,7)

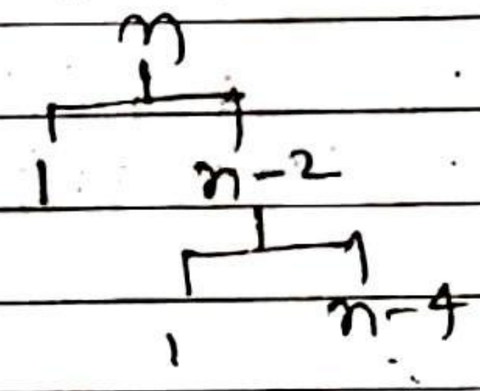
Total function call = 13

→ no. of levels in the tree equals to the no. of stack entries required.



In best case
✓ space complexity = O(log n)
(in case if it is balanced)

✓ In worst case space complexity = O(n)
(Unbalanced)



✓ In best case time complexity = θ(n log n)

$T(n) = 2 * T(n/2) + O(n)$

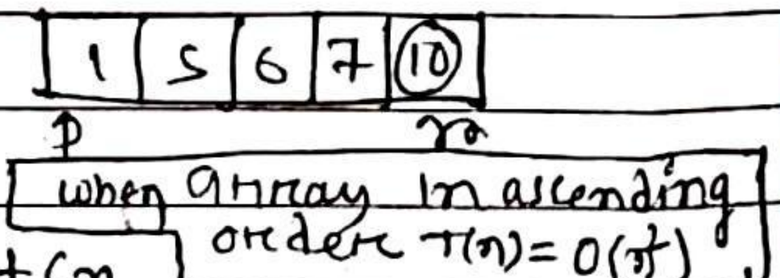
using masters theorem

$T(n) = \theta(n \log n)$.
$= \Omega(n \log n)$

✓ Worst case time complexity ≤ $O(n^2)$

back substitution -

$T(n) = T(n-1) + O(n)$
$= T(n-1) + Cn$
$= T(n-2) + C(n-1) + Cn$
$= T(n-3) + C(n-2) + C(n-1) + Cn$.
⋮



~~Example~~

$= c + c2 + c3 + \cdots + cn = c(1+2+\cdots n)$

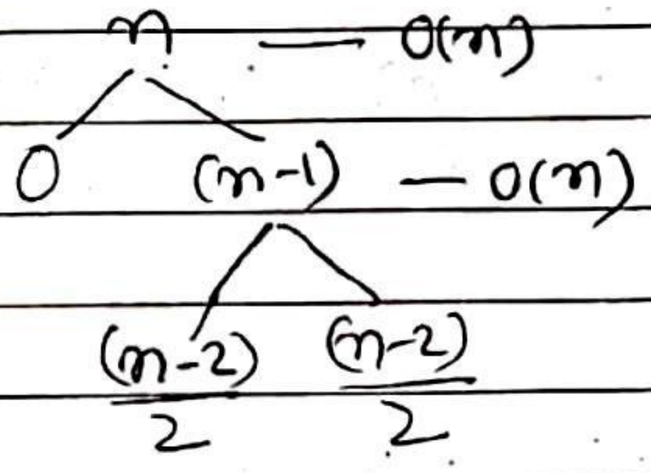
$= O(n^2)$ $= \frac{n(n+1)}{2}$

$T(n) = O(n^2)$.

→ Even input in assending order or desending order to, time complexity is $= O(n^2)$.
and also I/p all are same; then time complexity $= O(n^2)$

~~ex:~~

→ best and worst combination -



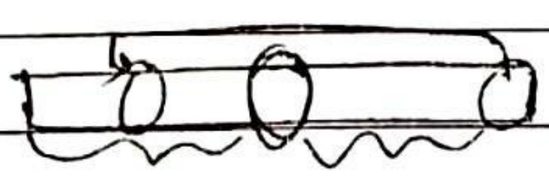
$T(n) = O(n) + O(n) + 2T\left(\frac{n-2}{2}\right)$

$\leq 2O(n) + T(n/2)$

$= \theta(n \log n)$.

Question-①

The median of 'n' elements can be found in O(n) time. which one of the following is correct about complexity of quick sort, in which median is selected as pivot ?

→ 

to found median $= O(n)$

replace $= O(1)$

partition $= O(n)$

$T(n) = O(n) + O(1) + O(n) + 2T(n/2)$ $- n/2 + n/2$

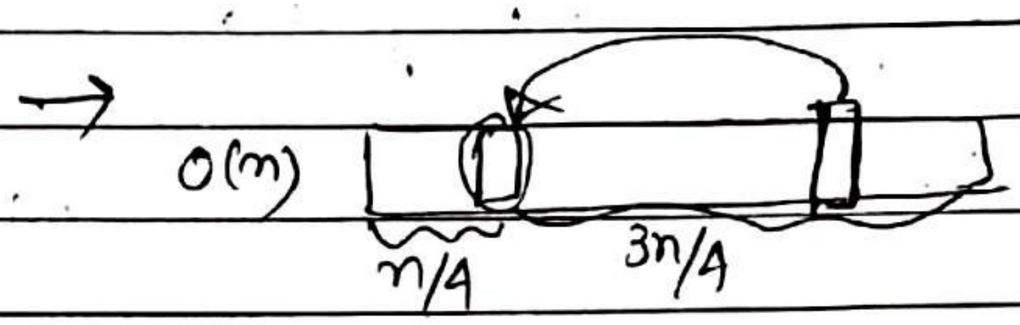
~~Using~~ $= O(n) + 2T(n/2)$

Using master theorem, $T(n) = \theta(n \log n)$

atlantis
Date
Page

Question - ②

In quick sort, for sorting 'n' elements, the $(n/4)^{th}$ smallest element is selected as pivot using $O(n)$ time algorithm. What is the worst space complexity of quick sort.



partition

$$T(n) = O(n) + O(1) + O(n) + T(n/4) + T(3n/4).$$

$$T(n) = O(n) + T(n/4) + T(3n/4)$$
$$= \Theta(n \log n).$$

1:3
1:9
1:27
1:999

$\Theta(n \log n)$

Question - ③

Using Quick sort On an algorithm, given I/P

1, 2, 3, · · · n — Time taken T_1

n, n-1, n-2, · · · 1 — " " T_2

What is the relationship between T_1 & T_2.

→ either elements are arranding order or desending order or all equal then all this case time taken $O(n^2)$.

$(T_1 = T_2)$.

atlantis
Date
Page

[Question] - ④

partition algo which take O(n) time,
we are spliting the problem into two part.
(1/5 n), (4/5 n), then what is time Complexity.

→ $T(n) = O(n) + T(n/5) + T(4n/5)$

$T(n) \leq O(n) + T(4n/5)$

- Introduction to - HEAPS :

	insert	search	Find min	Delete min
Unsorted array	O(1)	O(n)	O(n)	O(n)
Sorted array	O(n)	O(log n)	O(1)	O(n)
Unsorted Linked list	O(1)	O(n)	O(n)	O(n)
min heap	O(log n)		O(1)	O(log n)

→ heap is ~~a~~ datastructure which used optimise Time complexity some of the operation

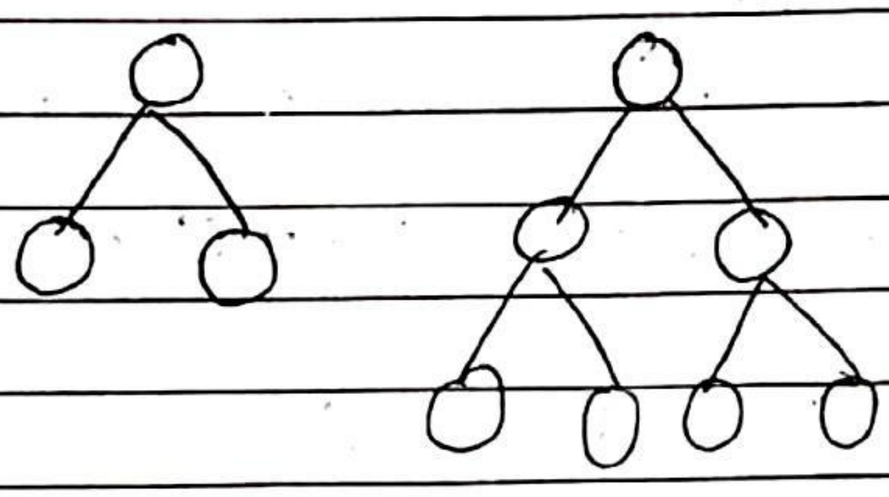
→

→ using heap we can implement heap sort algorithm.

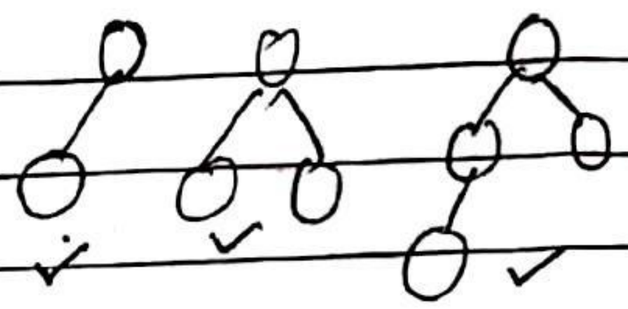
→ Heap could implement as a binary tree, or 3-ary ~~tree~~ tree ∴ n-ary tree

or

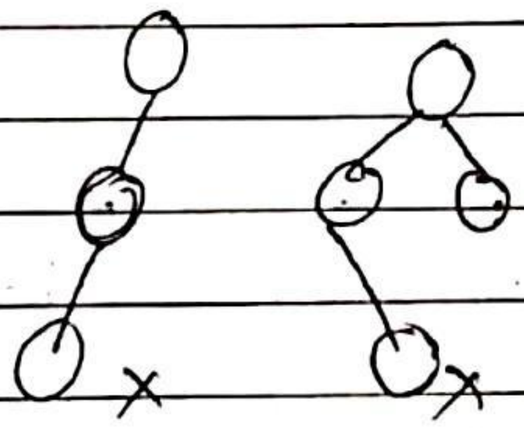
→ Every heap is almost Complete binary tree.



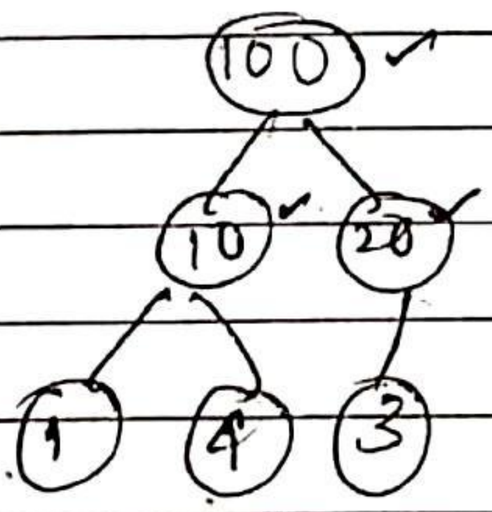
Complete B.T.



almost Complete B.T.

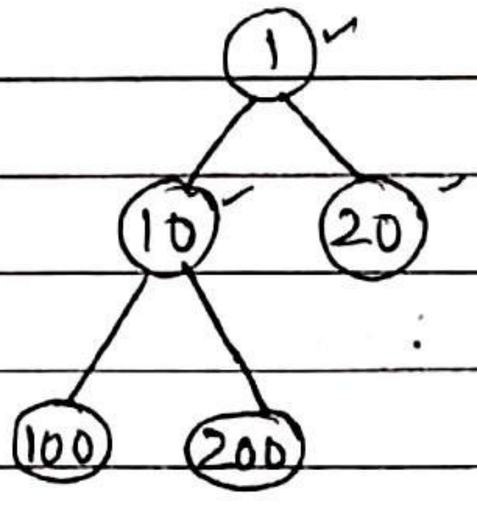


Max-heap :



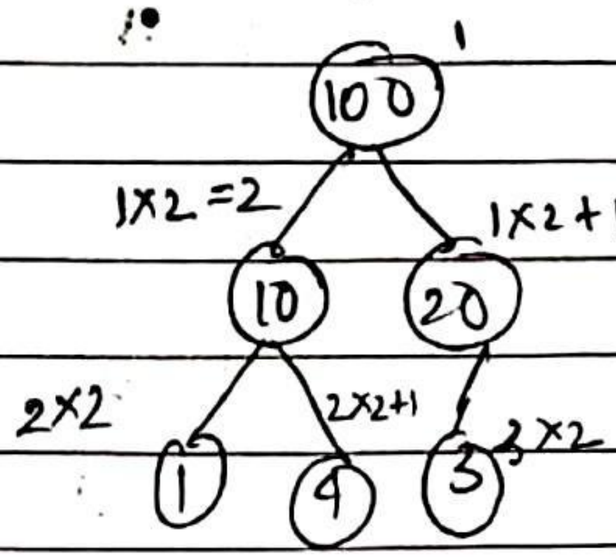
→ all the elements in the root should be greater than leaf.

Min-heap :



→ minimum element will present @ in the root.

max heap = Store complete binary tree in a array =



array	100	10	20	1	4	3
	1	2	3	4	5	6

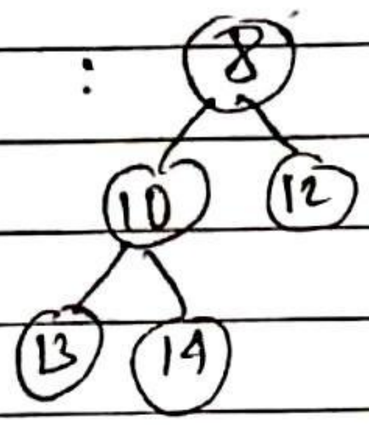
left child(i) = $2 \times i$

Right " (i) = $2 \times i + 1$.

parent(i) = $\lfloor i/2 \rfloor$

→ if the array in assending order - then it is already min heap.

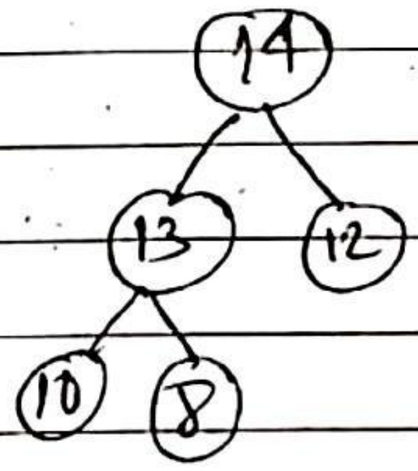
8	10	12	13	14



(root element always less than it child)

→ if the array in decending order - then it is already max heap.

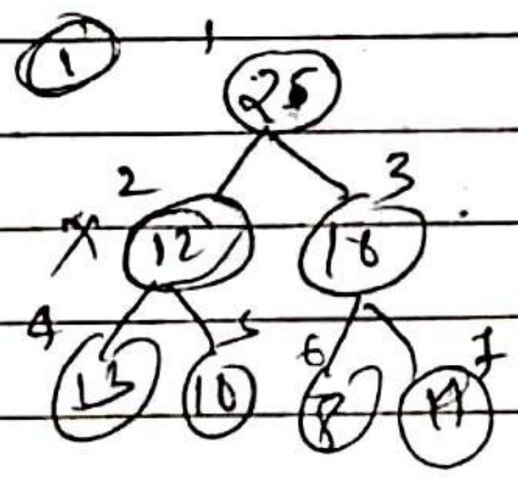
14	13	12	10	8

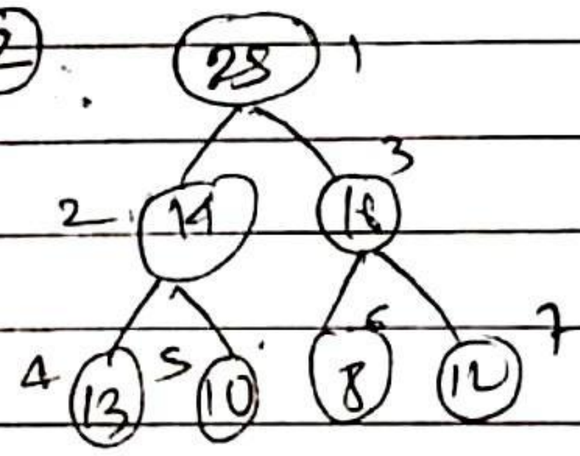


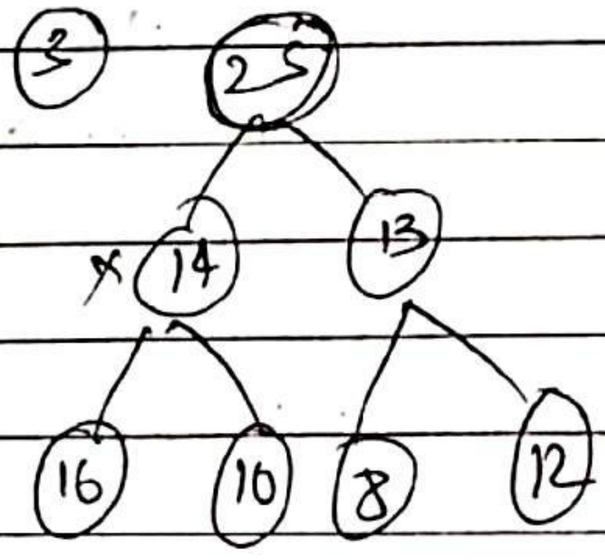
(root element always greater than its child)

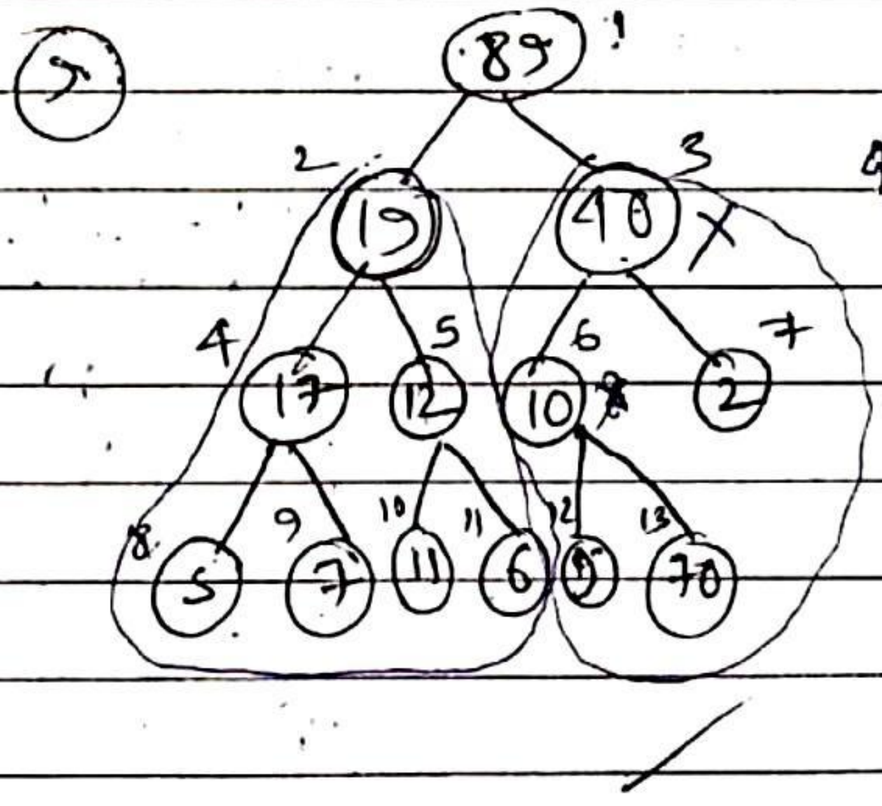
property follow heap ↑
Array heap

	1 2 3 4 5 6 7	Array length	Array heap size
①	25, 12, 16, 13, 10, 8, 14	7	1
②	25, 14, 16, 13, 10, 8, 12	7	7 (Max heap)
③	25, 14, 13, 16, 10, 8, 12	7	1
④	25, 14, 12, 13, 10, 8, 16	7	2
⑤	14, 13, 12, 10, 8	5	5 (max h)
⑥	14, 12, 13, 8, 10	5	5 (max h)
⑦	14, 13, 8, 12, 10	5	5 (max h)
⑧	14, 13, 12, 8, 10	5	5 (max heap)
⑨	8, 9, 19, 40, 17, 12, 10, 2, 5, 7, 11, 6, 9, 70	13	2





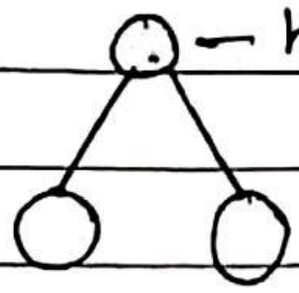




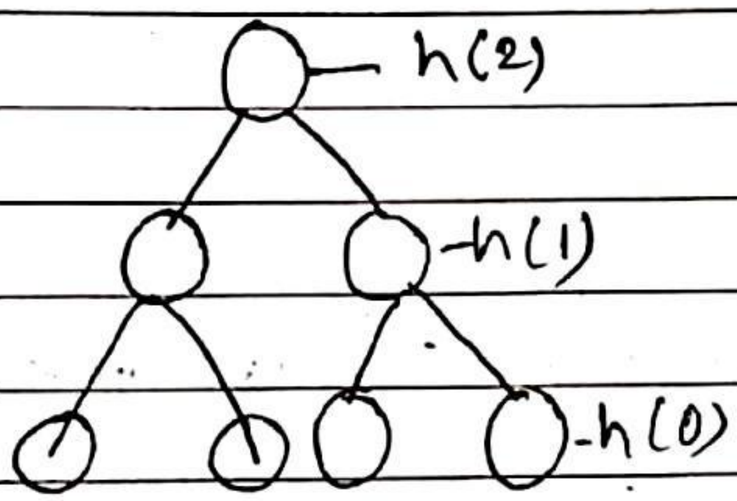
atlantis

Date
Page

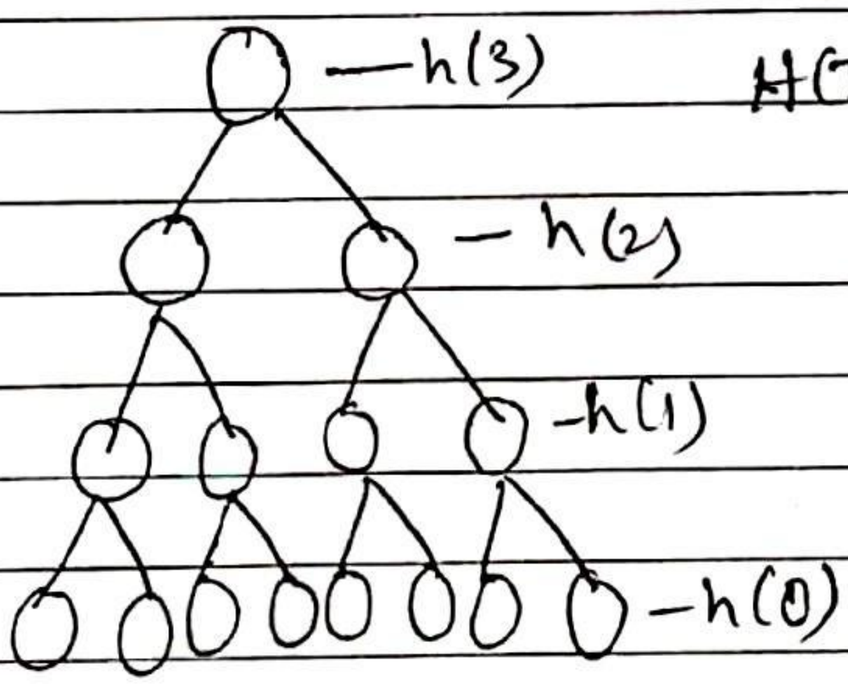
Some properties of complete binary tree =



hight of a tree = hight of root.



H(T) = 2



H(T) = 3

Hight	1	2	3	4	...	h	...
max node	3	7	15	31	...	$(2^{h+1}-1)$	

- max no. of node in complete binary tree = $(2^{h+1}-1)$

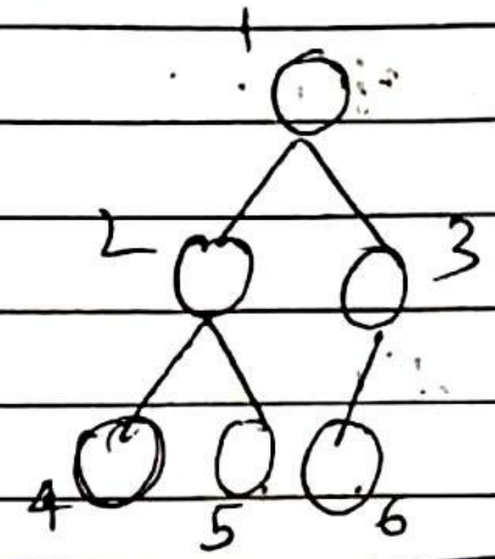
  (h → hight)

- 'n' nodes inside a complete or almost complete binary tree, ~~what is~~ then hight of tree = $\lfloor \log n \rfloor$

atlantis
Date
Page

→ hight of any binary heap is = $\lfloor \log n \rfloor$

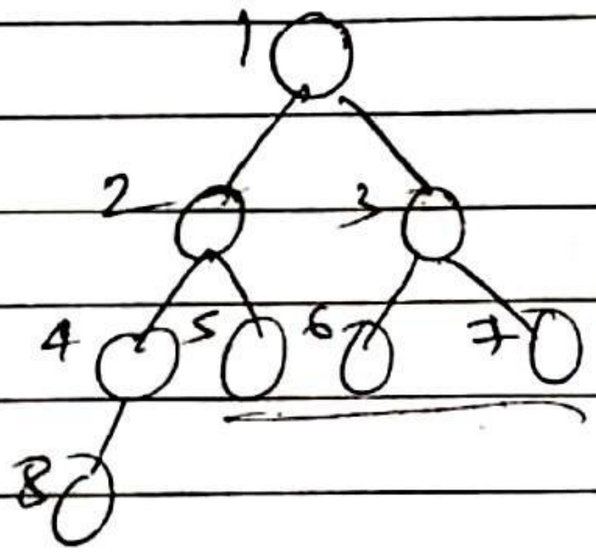
($n \rightarrow$ no. of node inside BT tree)



leafs = $\left(\lfloor \frac{n}{2} \rfloor + 1 \text{ to } n\right)$ — (follow ~~root~~ left in any leaf)

$= \left(\frac{6}{2} + 1 \text{ to } 6\right)$

$= (4 \text{ to } 6)$



leafs = $\left(\lfloor \frac{n}{2} \rfloor + 1 \text{ to } n\right)$

$= \left(\lfloor \frac{8}{2} \rfloor + 1 \text{ to } n\right)$

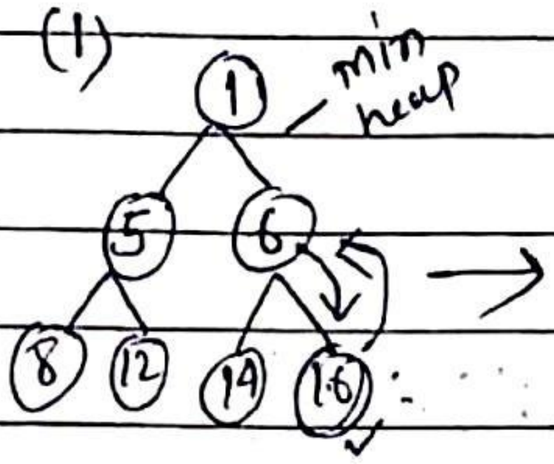
$= 5 \text{ to } 8$

MAX HEAP

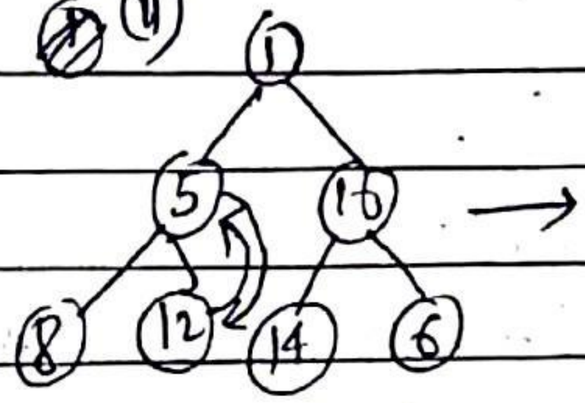
ex	0

1	5	6	8	12	14	16

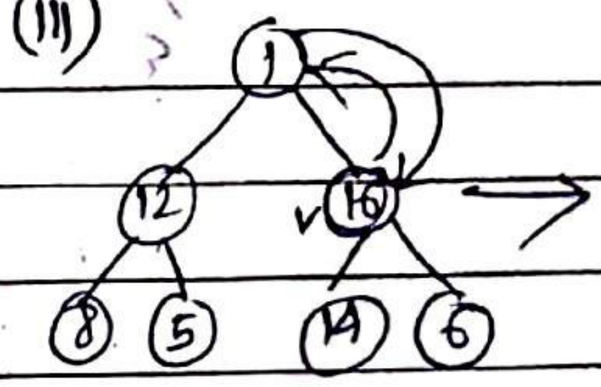
(I)



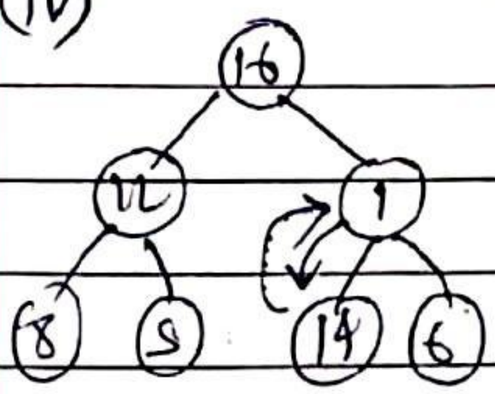
(II)



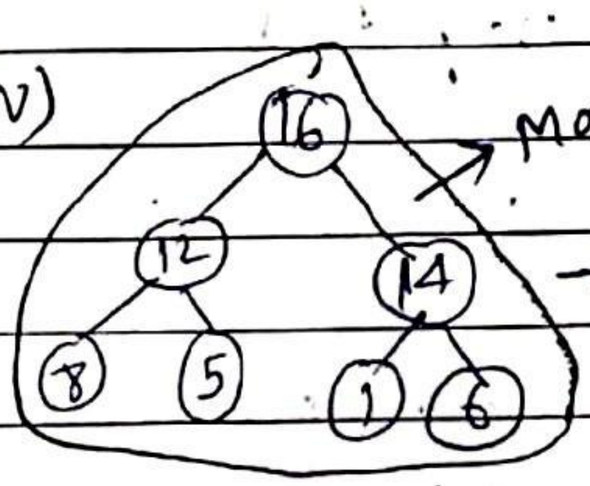
(III)



(IV)



(V)



max heap

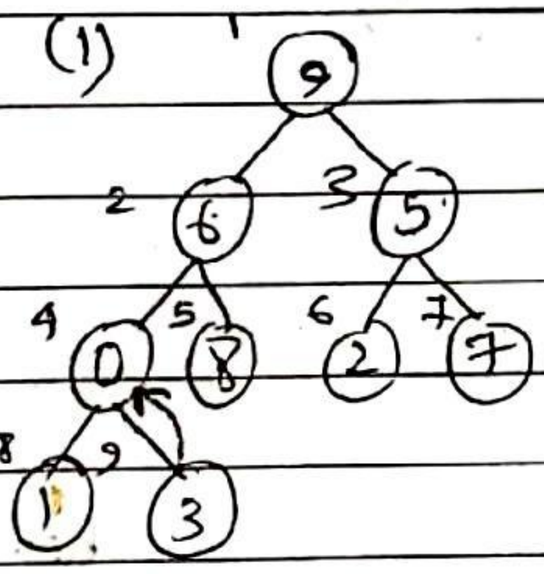
→ following max heap properties.

16	12	14	8	5	1	6

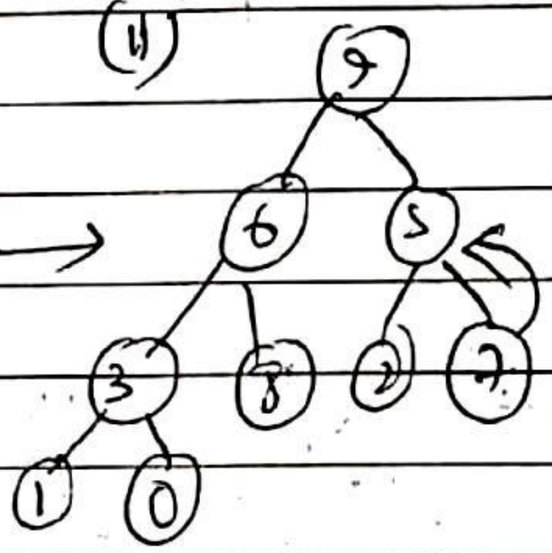
ex	0

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

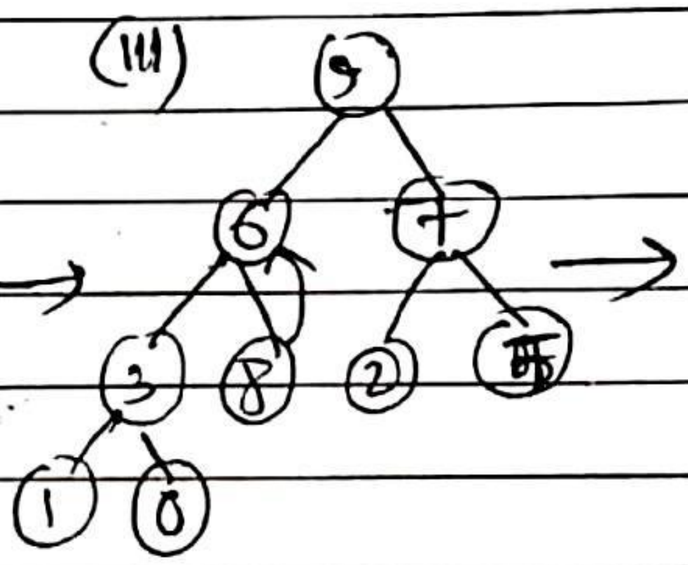
(I)



(II)



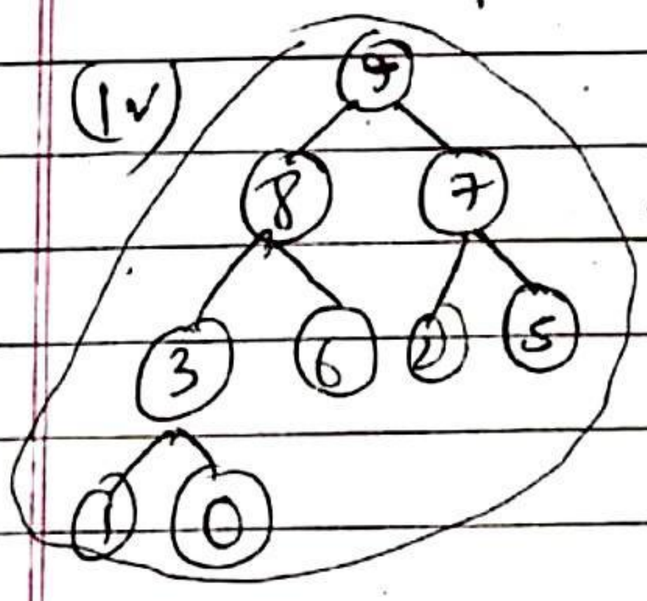
(III)



leaf = $\lfloor \frac{n}{2} \rfloor + 1$ to n

= (5 to 9)

(IV)



→ following Max heap Properties.

→ Every leaf is a Max heap.

(Max-heapify algorithm)

MAX-HEAPIFY (A, i)

{

l = 2i ;

r = 2i + 1 ;

if (l ≤ A.heap size and A[l] > A[i])

largest = l ;

else largest = i ;

if (l ≤ A.heap size and A[r] > A[largest])

largest = r ;

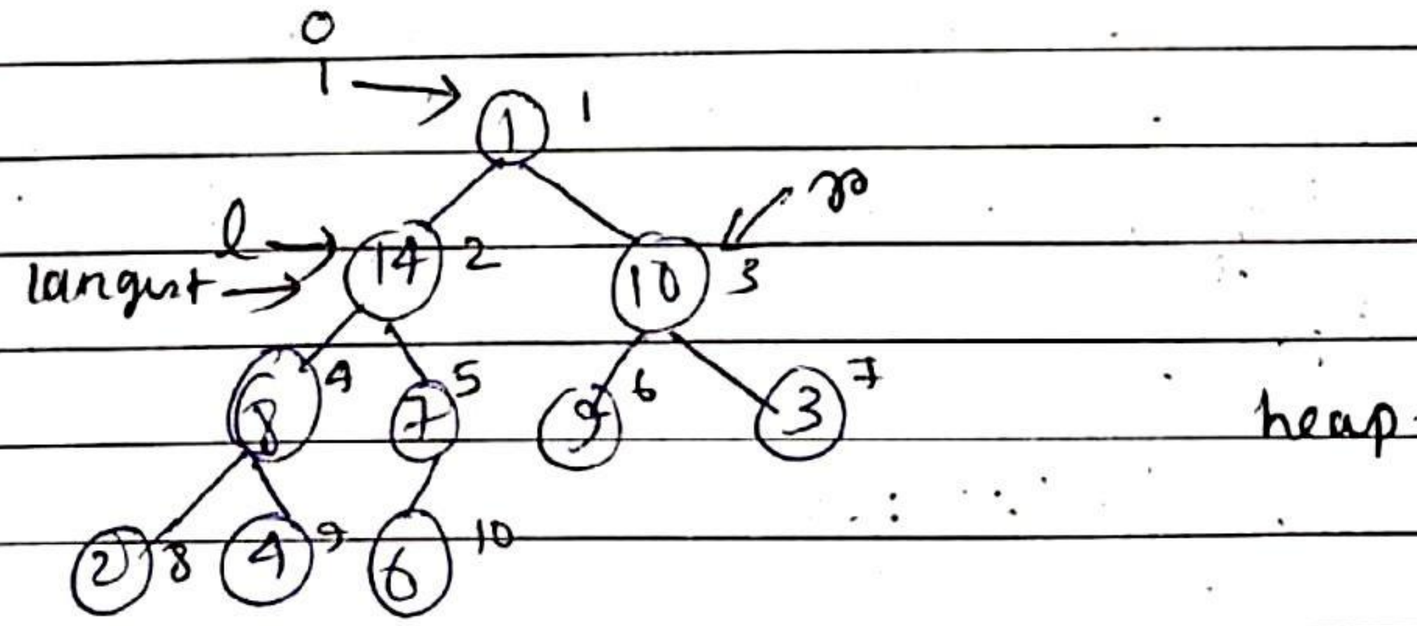
if (largest ≠ i)

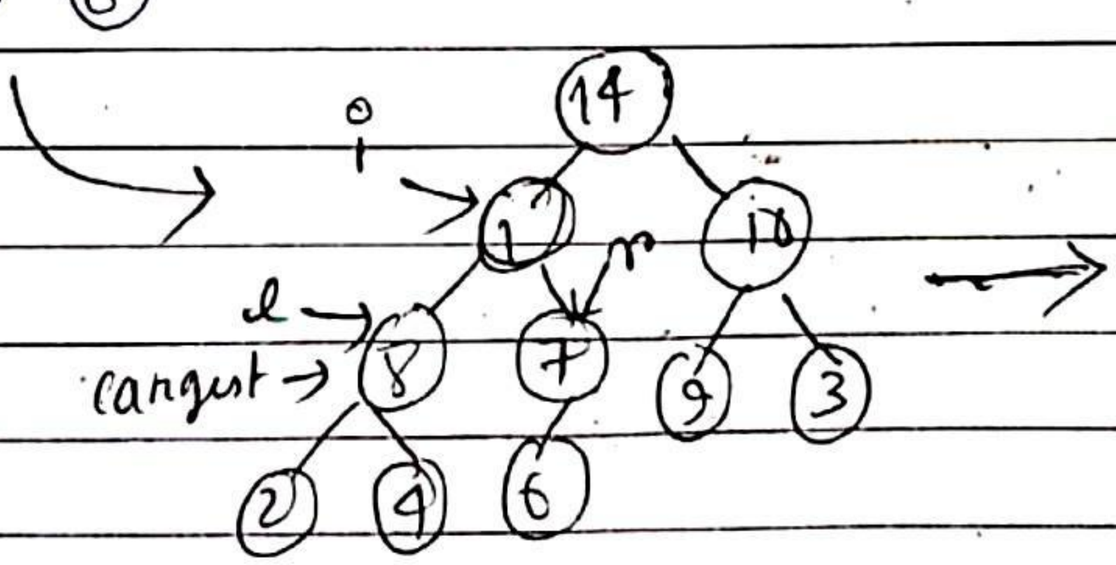
exchange A[i] with A[largest]

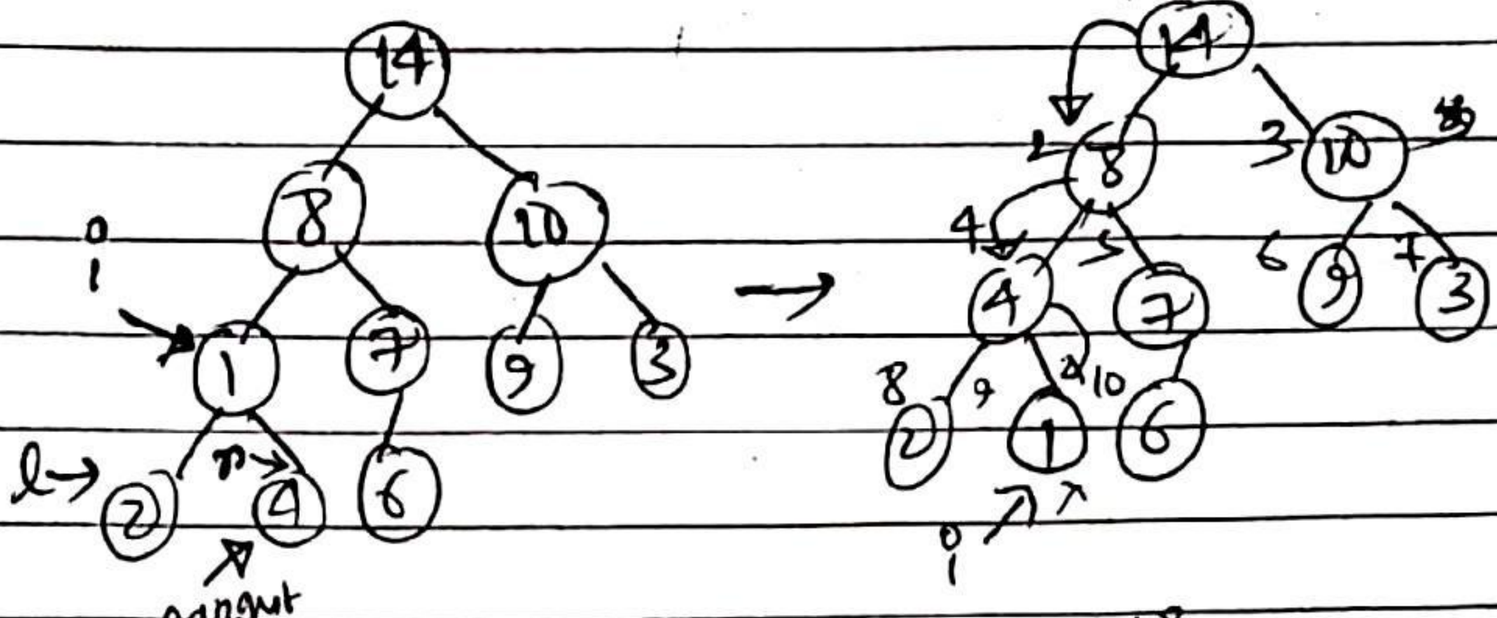
MAX-HEAPIFY (A, largest)

}

ex:







$l = 18$
$r = 19$

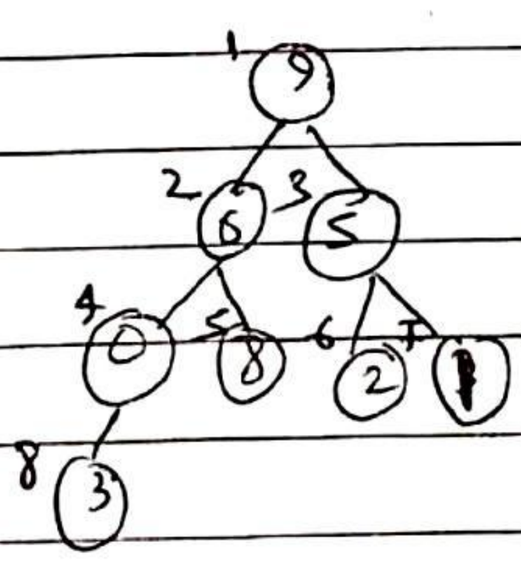
Total time complexity, $(2 \times \log n) = \boxed{O(\log n)}$

Space complexity, no. of level $= \boxed{O(\log n)}$

(Build max heap algorithm)

```
BUILD-MAX-HEAP(A)
{
    A.heapsize = A.length
    for (i = ⌊A.length/2⌋ downto 1)
        MAX-HEAPIFY(A, i)
}
```

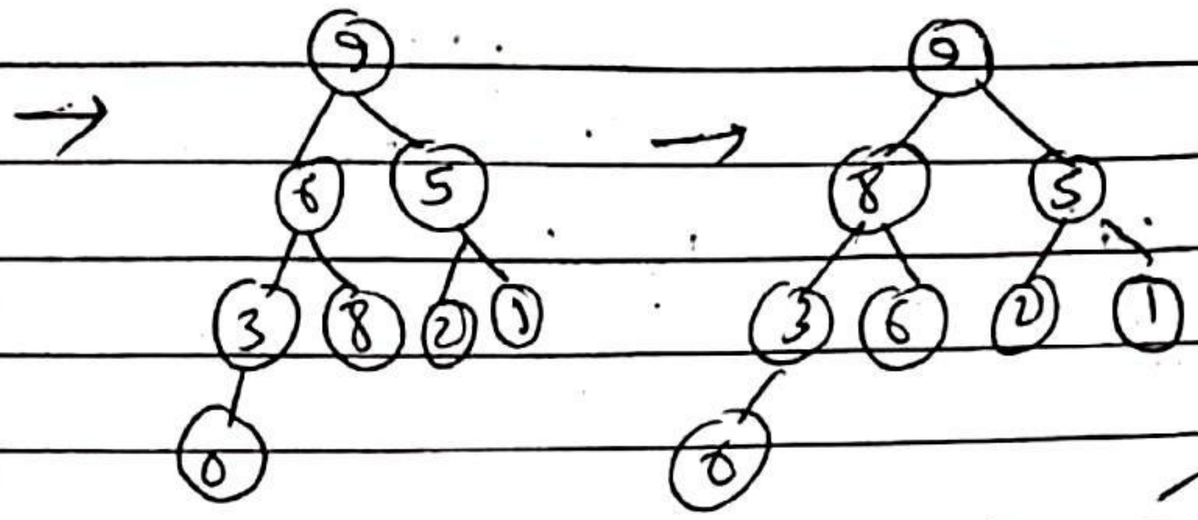
Ex: 9, 6, 5, 0, 8, 2, 1, 3

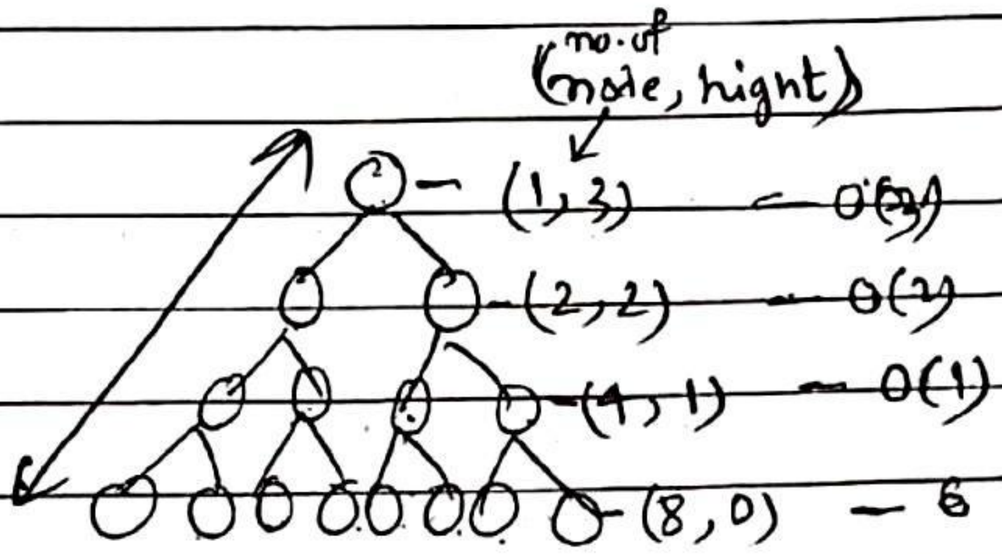


$(1 \text{ to } \lfloor n/2 \rfloor)$ – non leaf

$(\lfloor n/2 \rfloor + 1 \text{ to } n)$ – leaf.

$n \to 8$





Maximum no. of node present in ~~level~~ hight. "h" $= \left\lceil \frac{n}{2^{h+1}} \right\rceil$

$(h \to 0)$

$= \left\lceil \frac{15}{2^{0+1}} \right\rceil = 8$

$(h \to 1)$

$= \left\lceil \frac{15}{2^2} \right\rceil = 4$

$$\text{total time} = \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \text{ or } \epsilon h$$

$$= \frac{cn}{2} \sum_{h=0}^{\log n} \left(\frac{h}{2^h} \right)$$

$$= O\left(\frac{cn}{2} \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

In order build a max heap -

time complexity $= O(n)$

space complexity $= O(\log n)$

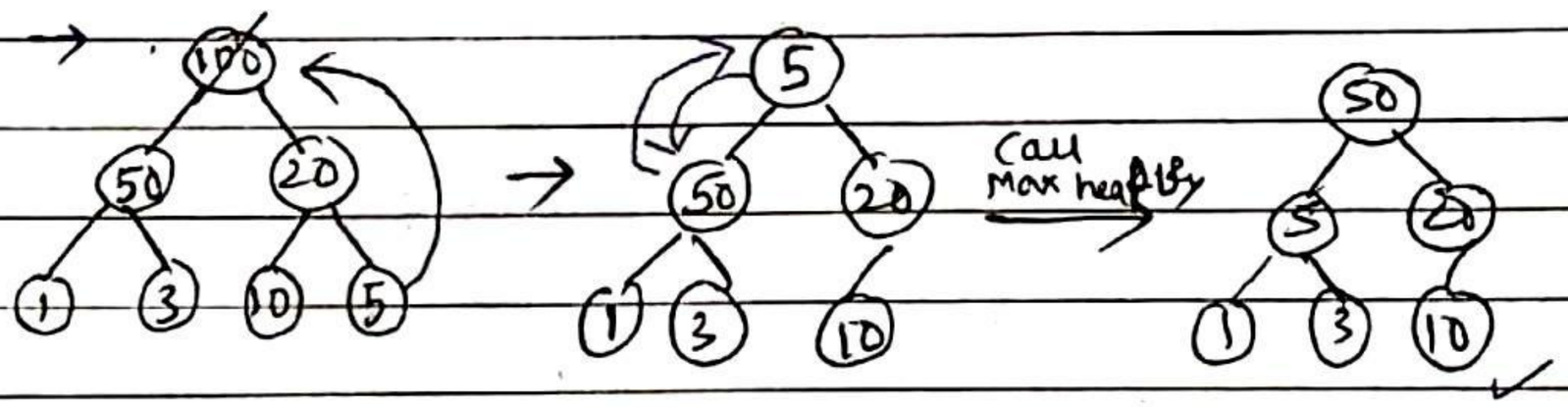
• Extract-max from ~~MAX-HEAP~~ MAX-HEAP :

```
HEAP-EXTRACT-MAX (A)
{
  if (A.heap-size < 1)
    error "heap underflow"
  max = A[1]
  A[1] = A[A.heap-size]
  A.heap_size = A.heap_size - 1
  MAX-HEAPIFY (A,1) }  O(log n) time.
  return max;
}
```

} Constant time

Ex:

100, 50, 20, 1, 3, 10, 5 , Delete-max-value (root value)



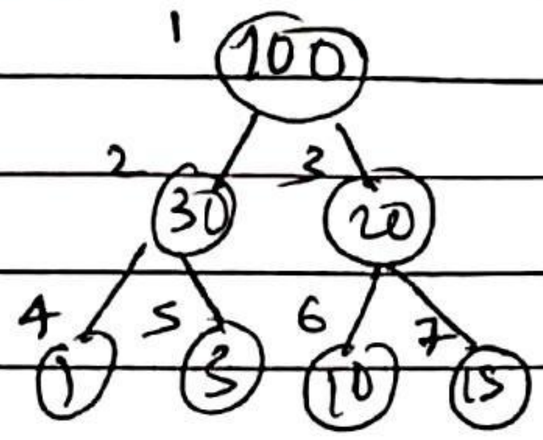
— Total time complexity = O(log n)
Space complexity = O(log n).

• HEAP - Increase Key (Max-heap)

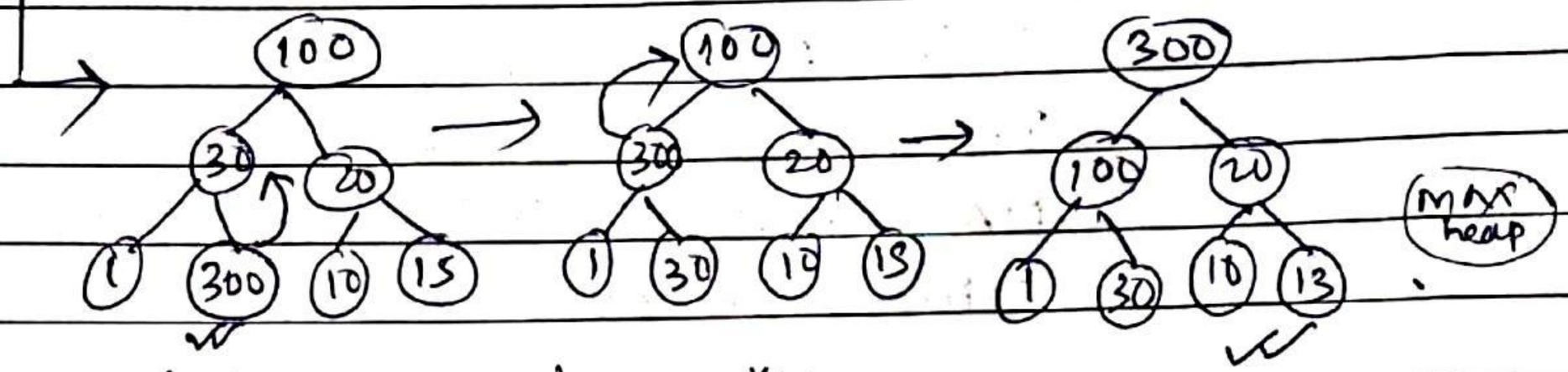
```
HEAP-increase-Key (A, i, Key)        i → Index number
{
  if (Key < A[i])
    error
  A[i] = Key
  while (i > 1 and A[i/2] < A[i])
  Change A[i] and A[i/2] ; i = i/2 ; }
```

atlantis
Date
Page

[ex]: Increase element of index 5 by 300 (key)



$\begin{cases} i = 5 \\ Key = 300 \end{cases}$

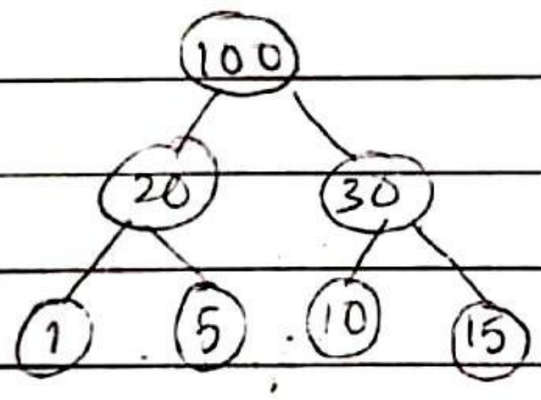


to Increase or decrease key =

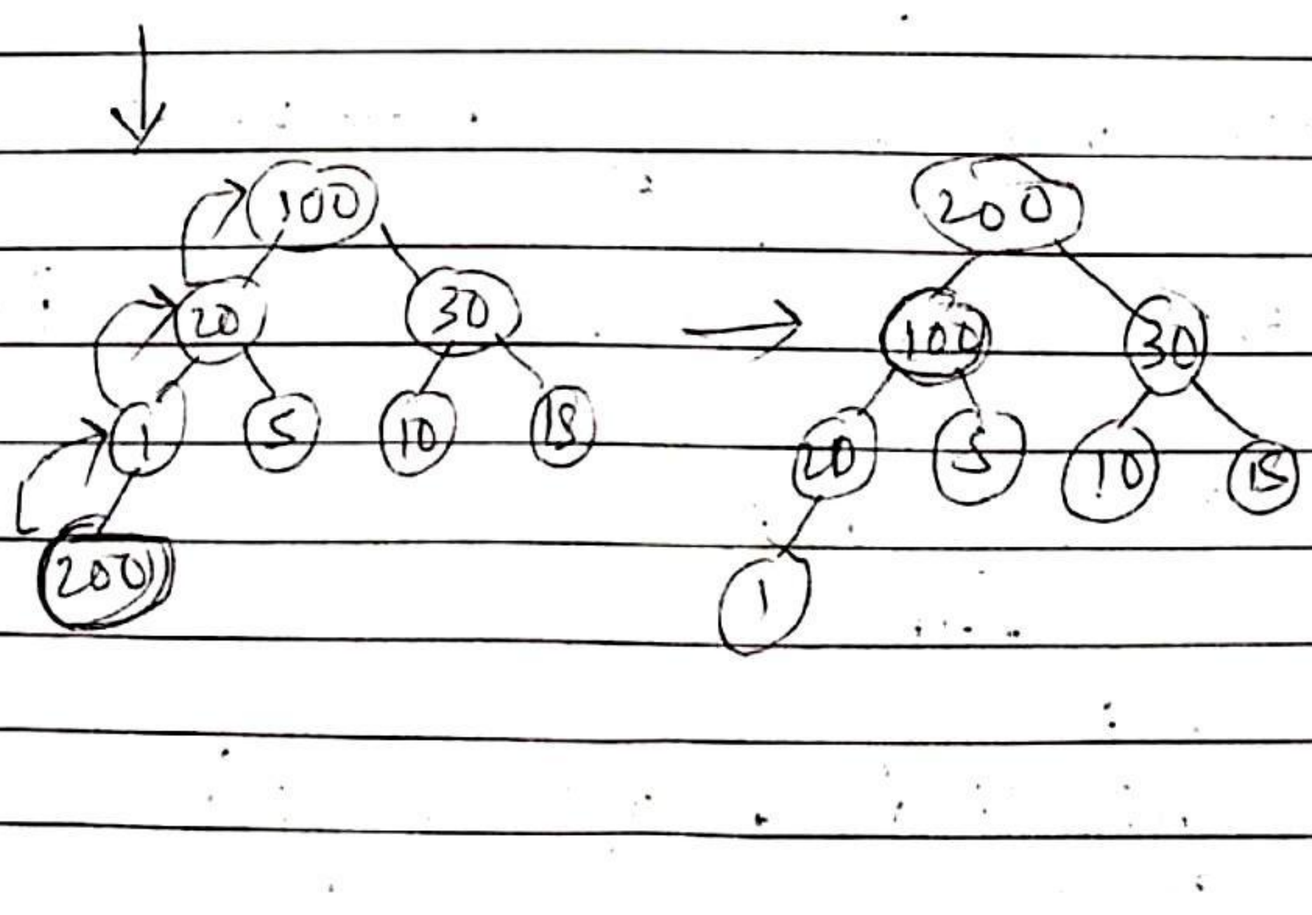
→ - Time Complexity = O(log n).

- Insert key into max-heap =

→ To Insert a element in max heap ;
Time complexity is = O(log n).



Insert - 200



(heap-operation)

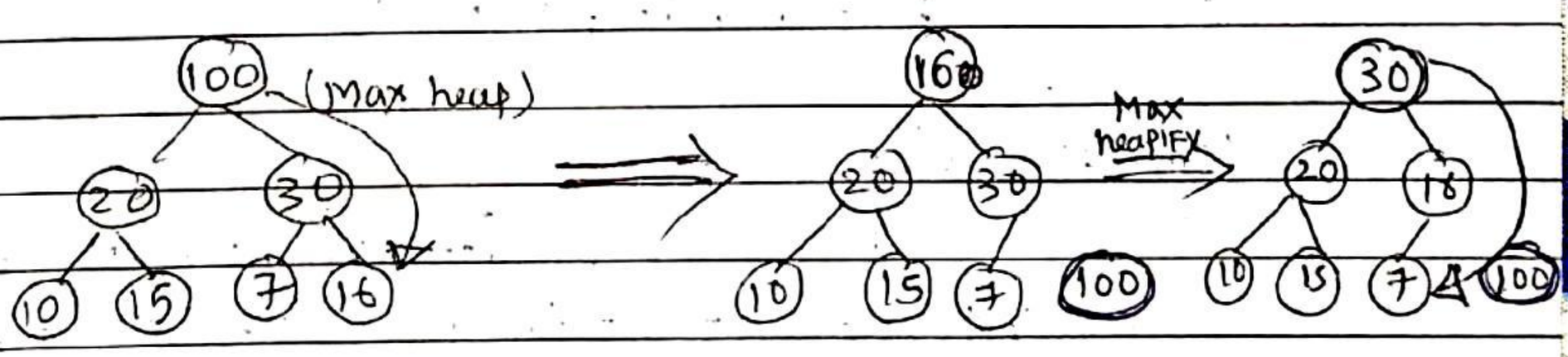
	Find max	Delete max	Insert	Key increase	Key decrease
Max heap	O(1)	O(logn)	O(logn)	O(logn)	O(logn)

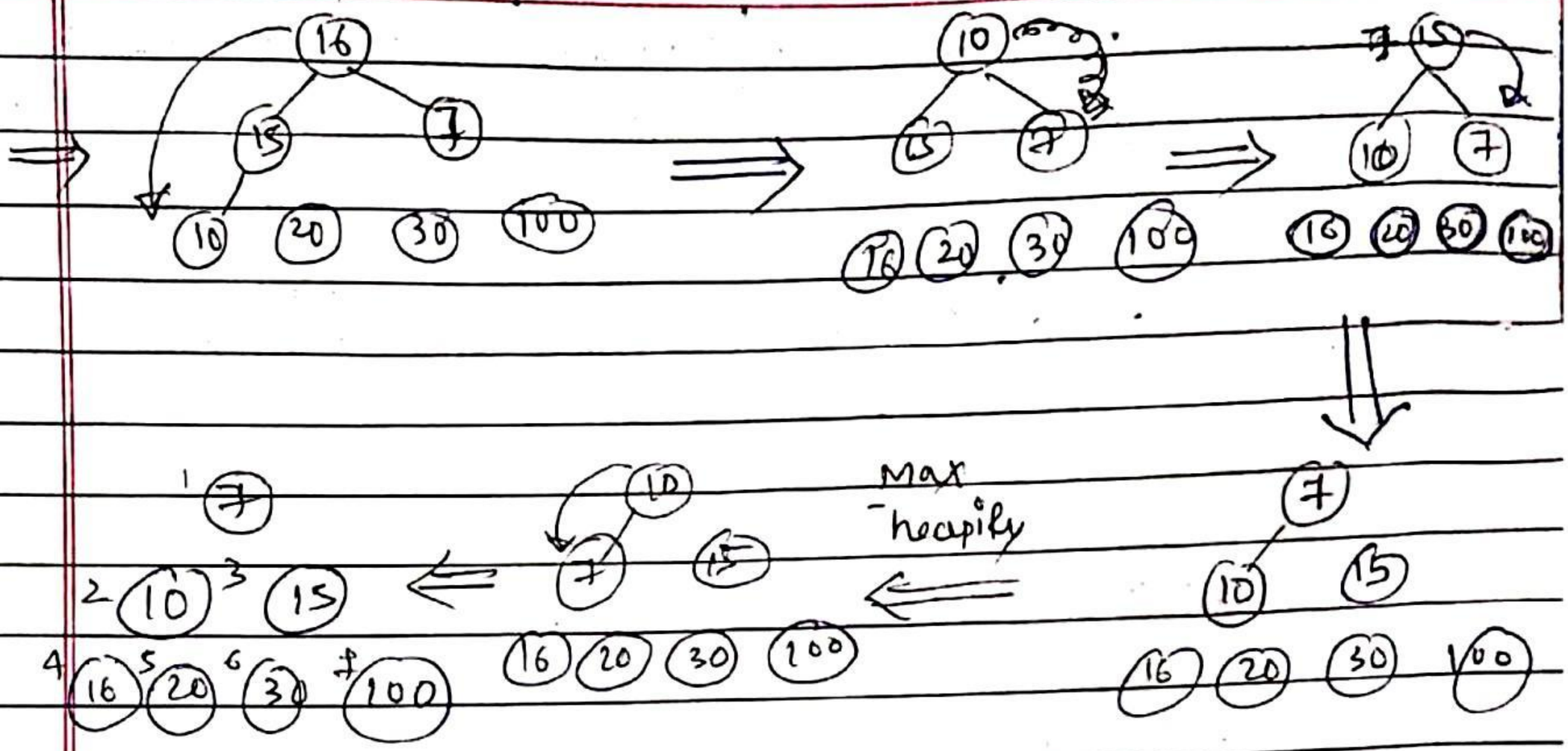
Find min	Search random element	delete any random element
O(n)	O(n)	O(n+n) = O(n)

- HEAP SORT and analysis =

```
HeapSort(A)
{
  BUILD-MAX-HEAP(A)
  for( i = A.length downto 2)
   exchange A[i] with A[1]
    A.heap size = A.heapsize - 1;
    Max-HEAPIFY(A,1)
}
```

→ | 100 | 20 | 30 | 10 | 15 | 7 | 16 |





→ Time complexity (Heap sort) = $O(n \log n)$

$T(\log n)$ → for height of heap.
$T(n)$ → for heapify.

Questions (heap and heap sort) :

Question-1

In a heap with '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) $\theta(1)$

→ delete 1st min $= O(\log n)$
" 2nd " $= $ "
" 3rd " $= $ "
" 4th " $= $ "
" 5th " $= $ "
" 6th min $= O(\log n)$

Find 7th $= O(1)$
Insert $= 6 * O(\log n)$
Time $= 6 * O(\log n) + O(1)$
$= O(\log n)$

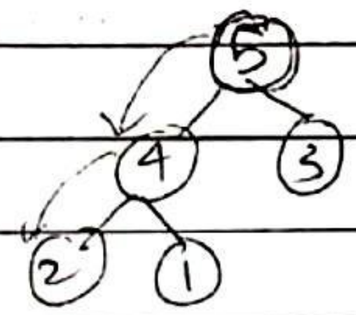
atlantis
Date
Page

Question ②

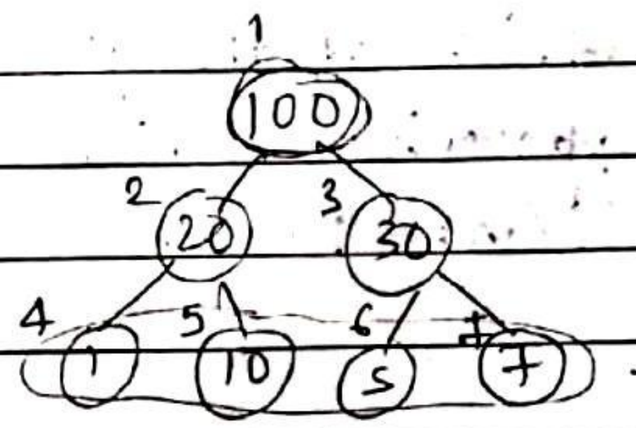
In a binary max heap containing 'n' numbers the smallest element can be found in time -

(a) $\theta(n)$ (b) $\theta(\log n)$ (c) $\theta(\log \log n)$ (d) $\theta(1)$.

→



$O(\log n) + O(1)$



- smallest element

$leaf = (\lfloor n/2 \rfloor + 1 \text{ to } n)$

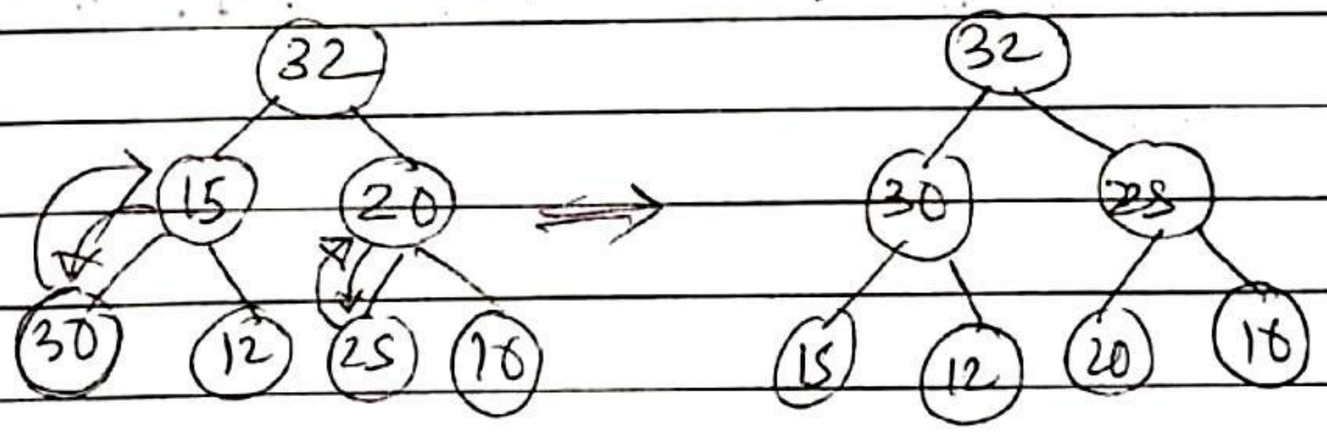
$= (4 \text{ to } 7)$

Time taken to find ~~mini~~ smallest no. $= O(n)$.

Question - ③

32, 15, 20, 30, 12, 25, 16 this element Inserted into a Max heap what is resulting heap look like -

→



resulting order (32, 30, 25, 15, 12, 20, 16)

atlantis
Date
Page

Question-9

3-ary Max heap,

① 1, 3, 5, 6, 8, 9

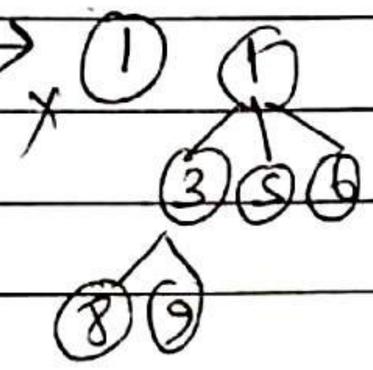
② 9, 6, 3, 1, 8, 5

③ 9, 3, 6, 8, 5, 1

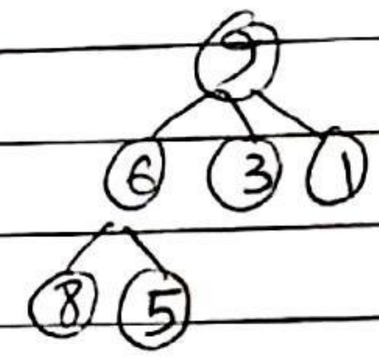
④ 9, 5, 6, 8, 3, 1

~~Which of the given sequence follow 3-ary Max heap property~~ — which of the following array represent 3-ary max heap -

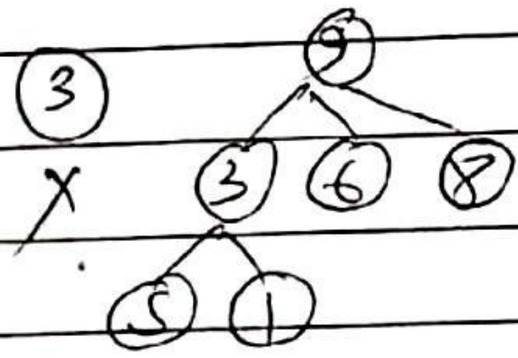
→ ① X

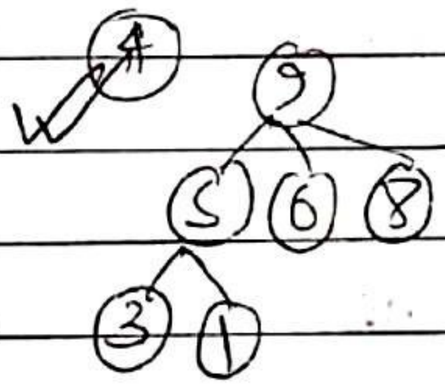


② X



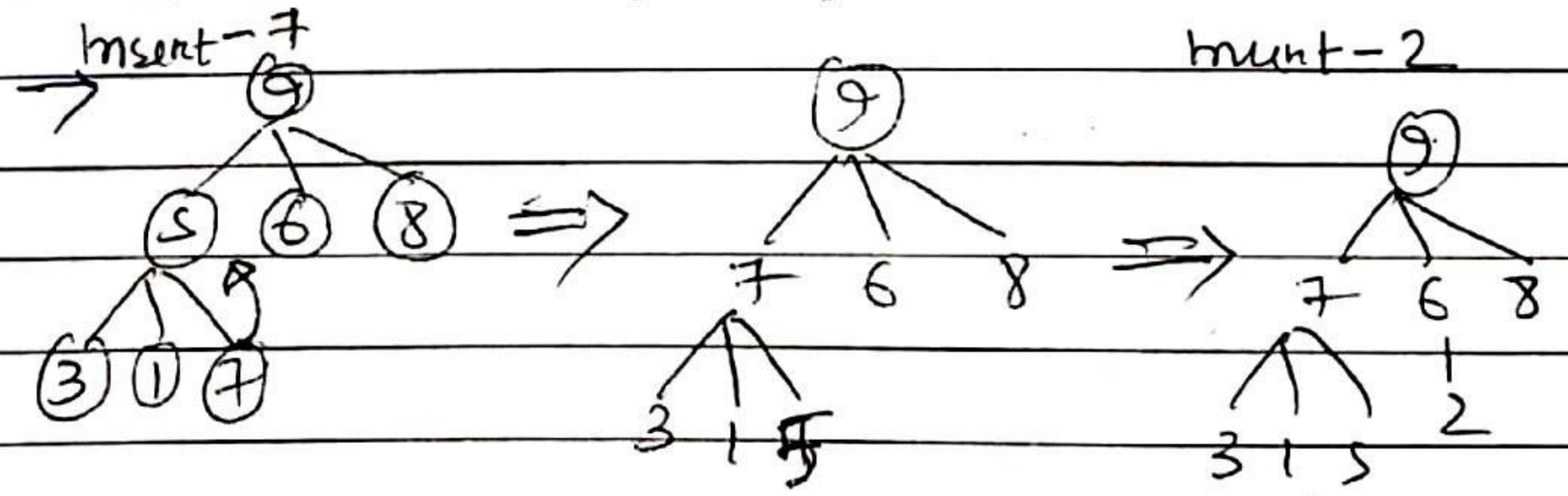
③ X

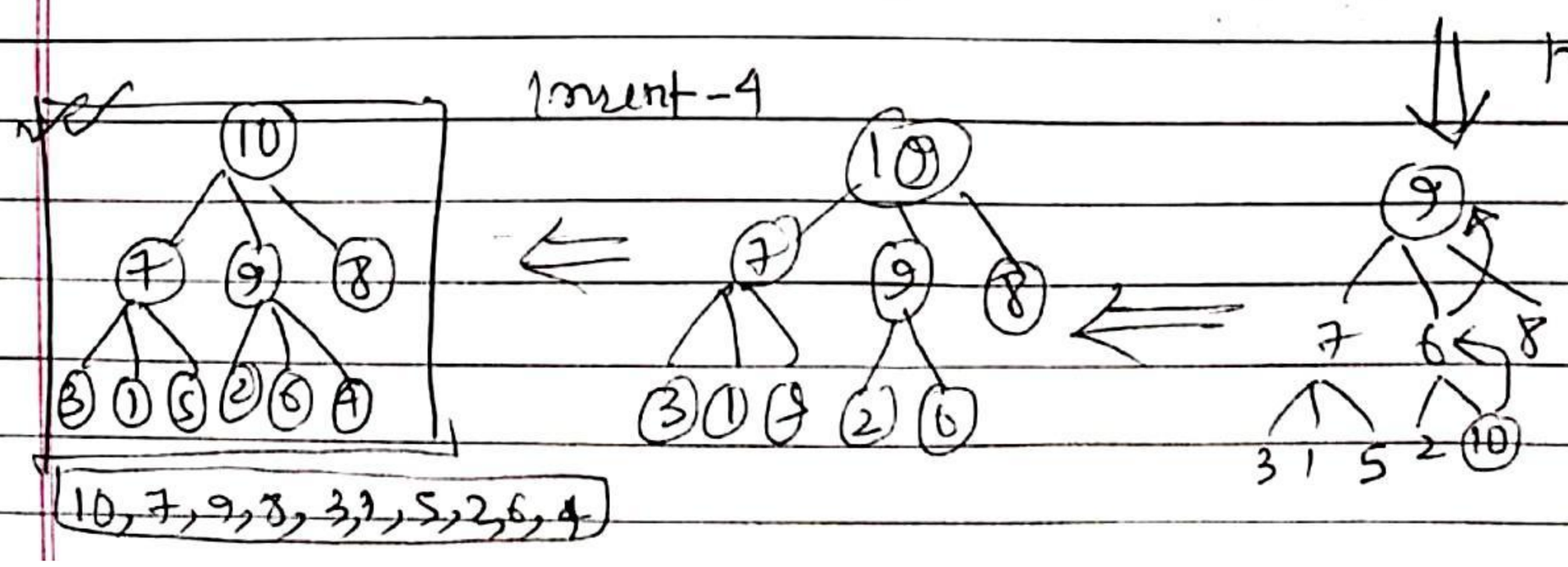




(always parent is greater than children)

then insert (7, 2, 10, 4) into ~~it~~ the result -



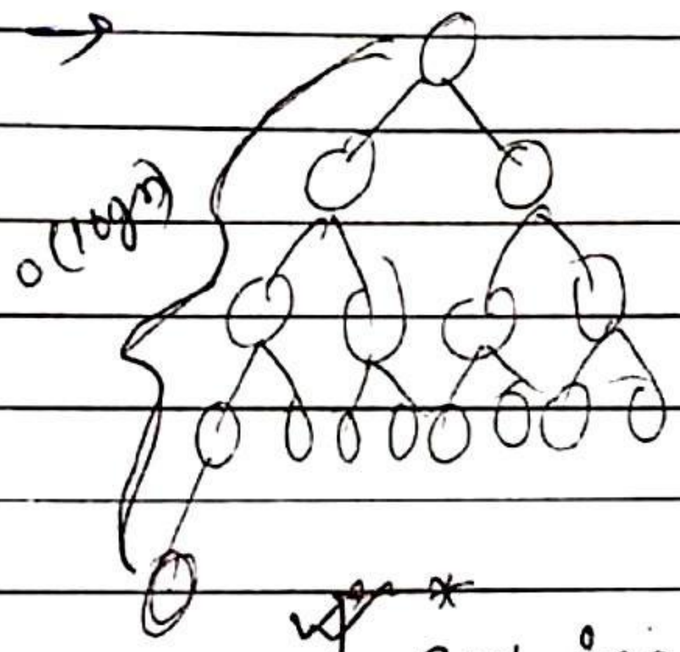


atlantis
Date
Page

Question - (5)

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

→



heap-hight (logn)
(When n element
Binary search O = logn
= O (log logn)

* applying Binary search on some problem which are have Time complexity of O(n) is not going to reduce the complexity to O(logn)

Question - (6)

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

→ 2n
↓ call (BUILD heap)

O(2n) = O(n)

→ put all 'n' element into array and then call build heap.

for 'n' element BT-C = O(n)
2n " Time-C = O(2n) = O(n).