Date			
Page No.			

9 • live

1 meri ja

f(0) =

ADA

Insertion Sort

PSEUDOCODE

```
for j = 1 to A.length
      key = A[j]
      i = j - 1
        while i ≥ 0 and A[i] > key
           A[i+1] = A[i]
             i = i - 1
        A[i+1] = key
```

Worstcase	$O(n^2)$
Best case	$O(n)$
Average	$O(n^2)$

Best Case (Ascending order)

	Comparison	Swap
10 20 30 40 50 60	$(n-1)$	0

$\Rightarrow O(n)$

Worst case (Descending order)

	Comparison	Swap
60 50 40 30 20 10	0	0
50 60 40	1	1
40 50 60	2	2
	3	3
	⋮	⋮
	$n-1$	$n-1$

$1+2+3+\ldots(n-1)$

$= \frac{n^2-n}{2}$

$O(n^2)$

Date
Page No.

~~Bubble~~ SORT.

INSERTION SORT

Explanation of code,

Line What is does.

1. starts from 2nd element because 1st is already sorted.
2. Pick the element to insert.
3. Compare with previous element
4. shif elements oneposition right if they are larger
5. Inserts the key in its correct position.

Example.

A = [25, 10, 45, 20, 35]

	A[0]	A[1]	A[2]	A[3]	A[4]
K_1	(25)	10	45	20	35
K_2	25	(10)	45	20	35
K_3 =	10	25	(45)	20	35
K_4 =	10	25	45	(20)	35
K_5 =	10	20	25	45	(35)
K_6 =	10	20	25	35	45

Date
Page No.

Analysis

$T(n)$ = cost of outer loop + cost of inner loop.

The outer loop runs $(n-1)$ times.

The inner loop runs variable no. of times - depends on data order.

Line	operation	How many times it executes
1	for $j = 2$ to n.	$(n-1)$ times
2	Key = A[j]	$(n-1)$
3	$i = j - 1$	$(n-1)$
4-6	while loop	depends on data order ($\sum t_j$)
7	A[i+1] = Key.	$(n-1)$ times

If array is already sorted $t_j = 1$

If array is reverse " $t_j = j$

Every line has constant cost. Let's call it c_i

Total running time

$$T(n) = c_1 n + c_2 n + \cdots \sum_{j=2}^{n} c_4 t_j$$

Already sorted $(t_j = 1)$ $T(n) = O(n)$

Reverse " $(t_j = j)$ $T(n) = O(n^2)$

Best , Worst

↳ $T(n) = \sum_{j=2}^{n} \frac{n(n-1)}{2} = O(n^2)$

Quick Sort

Pseudocode.

```
Quick Sort (A, p, r)
1. if p<r
2.     q = partition(a, p, r)
3.     Quick Sort (a, p, q-1)
4.     Quick sort (a, q+1, r)

Partition ( A, p, r)
  x = A[r]          // pivot
  i = p-1
  for j = p to r-1
      if A[j] ≤ x
          i = i+1
          swap (A[i], A[j])
  swap (A[i+1] and A[r])
  return i+1
```

Date
Page No.

Example:

a = [10, 80, 30, 90, 20].

for p=0, r = 5.

Partition (a, 0, 5)

x = A[5] = 20

i = -1.

j → 0 to 4

j	A[j]	Compare	Action	Result	i
0	10	10 ≤ 20.	i=0 swap A[i] & [A[j]	[10,80,30,90,20]	0
1	80	80 ≤ 20	No.	[10,80,30,90,20]	0
2	30	30 ≤ 20	No.	[10, 80, 30, 90, 20]	0
3	90	90 ≤ 20	No	[10, 80, 30, 90, 20]	0
4	20	20 ≤ 20	i = 1 swap A[i] & A[j].	[10, 20, 30, 90, 30] ⇒ (10, 20, 80, 90, 30) 30, 90, 30	1

p+1 = (2)

[30, 90, 80]

Date
Page No.

Divide and conquer analysis

- For an array of size n.
  - Partition take O(n) time.
  - Then the array is divided into 2 subarrays of sizes k & n-k-1.

Hence the recurrence relation

$$T(n) = T(k) + T(n-k-1)$$

Deriving Recurrence Relation.

Let T(n) be running time of Quick sort for an array of size n.

Step 1. partition cost -

- The partition procedure scans through the array once - comparing each element with the pivot & possibly swapping it.
  hence cost of partition = O(n)

Step 2 Recursive Calls.

- After partitioning, suppose the pivot end up at index q.

That means:

- Let subarray size = $k = q - p$
- Right subarray size = $n - k - 1$

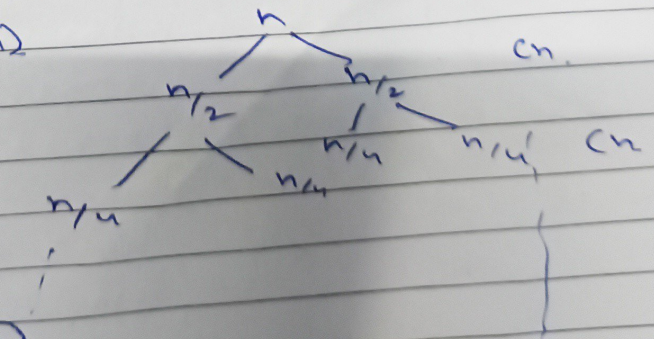
Total cost = cost of left sort + cost of right sort + cost of partition.

Best case (Balanced equally)

- If each partition divides the array almost equally than $k \approx n/2$.
- Substitute in recurrence

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

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



$n \to n/2$

$\frac{n}{2^i} = 1$

$i = \log_2 n$

①

$O(n \log n)$

Worst case (Highly Unbalanced partition).

- If the pivot is always the smallest or largest element.
  then one side gets almost all elements & otherside gets none.

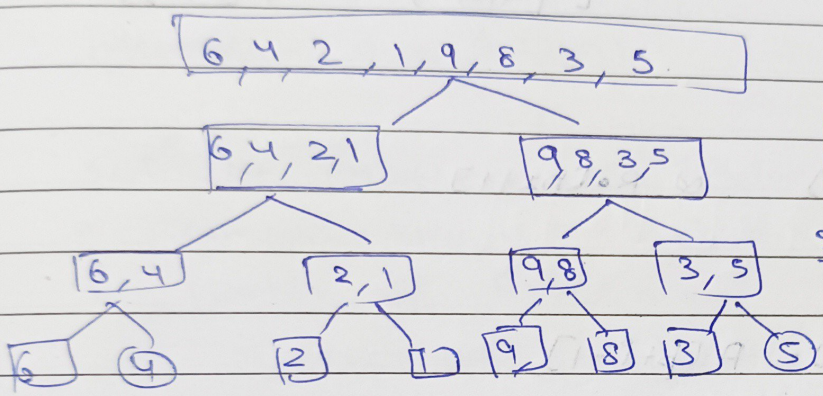
So, $k = 0$ or $k = n-1$.

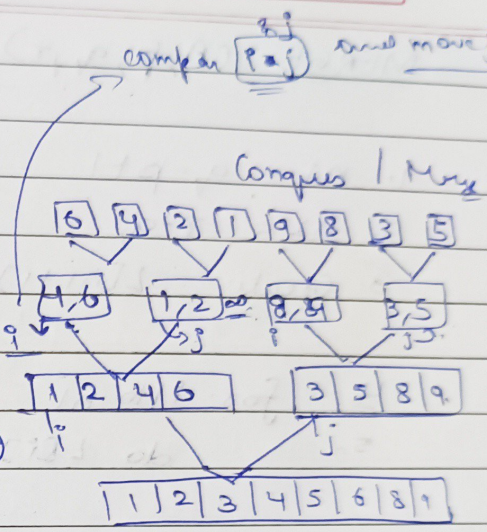
$$T(n) = (n-1) + (n-2) + \cdots + 1 = O(n^2)$$

Simple recurrence

$$T(n) = T(k) + T(n-k-1) + \Theta(n)$$

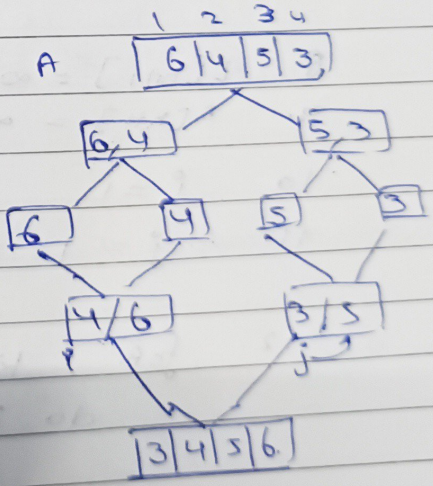
MERGE SORT





PSEUDO CODE

```
MERGE-SORT (A, p, r)
if  p < r
    q = ⌊(p+r)/2⌋
    MERGE-SORT (A, p, q)
    MERGE-SORT (A, q+1, r)
    MERGE (A, p, q, r)
```



MERGE (A, p, q, r)

L [4|6] R [3|3]

1. $n_1 = q - p + 1$
2. $n_2 = r - q$
3. Create $L[n_1+1]$ & $R[n_2+1]$
4. for i 1 to n_1
5. do $L[i] = A[p+i-1]$
6. for i 1 to n_2
7. do $R[i] = A[q+j]$
8. $L[n_1+1] = \infty$
9. $R[n_2+1] = \infty$
10. $i = 1$
11. $j = 1$
12. for k = p to r
    do if $L[i] \leq R[j]$
    then $A[k] = L[i]$
    $i++$
    else $A[k] = R[j]$
    $j++$.

Date | Page No.

Analysis of Mergesort.

- We analyse it in 3 steps:-

1. Divide the array into 2 halves + constant time. $O(1)$
2. Conquer: recursively sort each half :-> $2T(n/2)$
3. Combine all sorted array :-> $O(n)$

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

COUNTING SORT (Non comparison sort)

- Given input size -> n.
- Given range ~~n~~ K.

2 1 2 3 1 2 4

Occurrence array.

1	2
2	3
3	1
4	1
5	0

Make an array based on range

-> 1 1 2 2 2 3 4 5.

T.C = $O(n+k)$

- traverse to get count
- traverse to print sorted array

Eg 2 1 1 0 2 5 4 0 2 8 7 7 9 2 0 1 9

n = 17.

k = 9.

countarray C[0...k]

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

① store count

```
for (i=0; i<n; i++){
    C[A[i]] = C[A[i]]+1;
}
```

↳ n time

Now update this count array to get actual position of sorted array

countarr

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

updated count

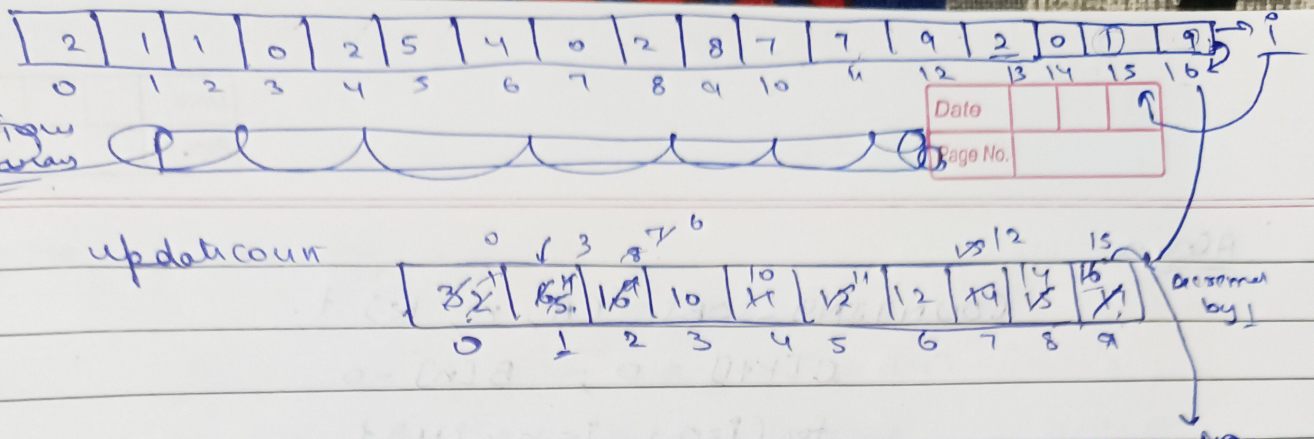
3	6	10	10	11	12	12	14	15	17
0	1	2	3	4	5	6	7	8	9

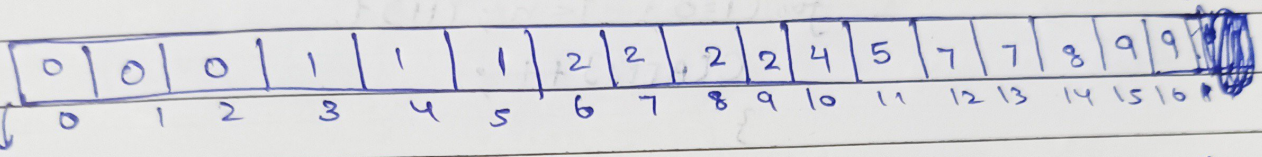
Update count.

②

```
for (i=1; i<=k; i++){
    C[i] = C[i] + C[i-1]
}
```

↳ k time





We started from back to maintain stability means order of elements

Building are

③

```
for ( i = n-1 ; i >=0 ; i--){
    B[--C[A[i]]] = A[i];
}
```

→ n times

④

```
for (i= 0 ; i<n ; i++){
    A[i] = B[i];
}
```

→ k times

T.C → O(n+k)

Date
Page No.

Algo:-

```
COUNTING-SORT (A, n, k) {
    C[k+1] = 0 ;  B[n] = 0.
    for (i=0 ; i<n ; i++) {
        C[A[i]]++ ;
    }
    for (i=1 ; i<=k ; i++) {
        C[i] = C[i] + C[i-1]
    }
    for (i=n-1 ; i>=0 ; i--) {
        B[--C[A[i]]] = A[i]
    }
    for (i=0 ; i<n ; i++) {
        A[i] = B[i]
    }
}
```

⇒ It is a non comparison based sorting algorithm.
⇒ Counting sort avoids comparison to beat O(n lgn) lower bound of comparison sort.

Time complexity analysis

Step 1 Initialize count array C[0 → k] → O(k)

→ we set all k+1 entries to zero

This taken

$T_1 \rightarrow O(k)$

Step 2 → count each element in A → O(n)

→ For each of n elements:

C[A[j]] = C[A[j]] + 1.

$T_2 \rightarrow O(n)$

Step 3 → Comput prefix sum.

For each $\substack{i=1 \text{ to } k \\ \Rightarrow \\ i}$ element.

C[i] = C[i] + C[i-1].

$T_3 \rightarrow O(k)$

Step 4. Build output array.

For i = n-1 to 0

B[--C[A[i]]] = A[i] $T_4 \rightarrow O(n)$

Total time comple.→ O(k) + O(n) + O(k) + O(n)

= O(k+n).

Date
Page No.

Space complexity

- Counting sort uses 3 arrays:
  - Input array A → size n.
  - Output B → size n
  - Count array C → size k+1

So total space.

space = O(n+k)

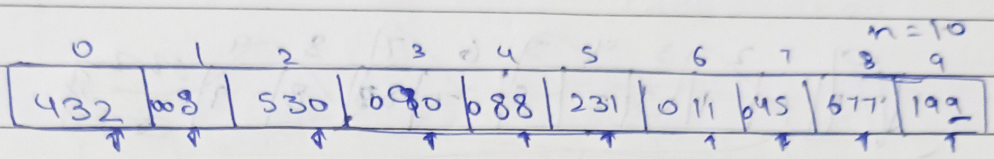
Counting sort does same amount of work, regradless of input order

So T.C is same in Best, Worst, average case.

Date
Page No.

RADIX SORT

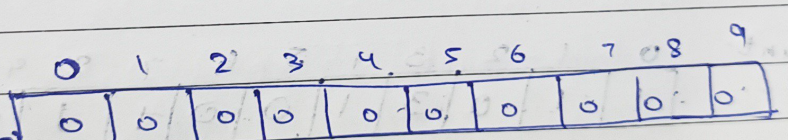
Eg



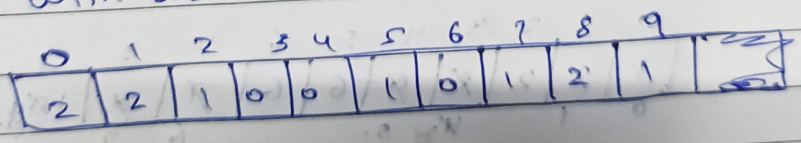
① First we have to check for max value count no of digits in it. Make all number accordingly to that like in above eg 677 → 3 digits so 8 becomes → 008

Pass 1 Apply count sort on L.S.B

Count-

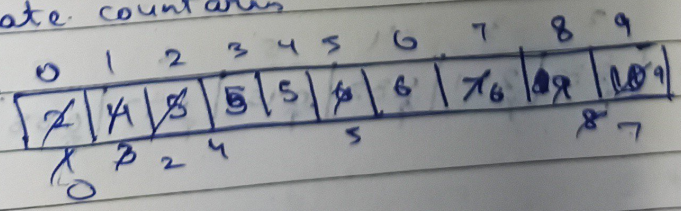


① Same as count sort build count array with LSB

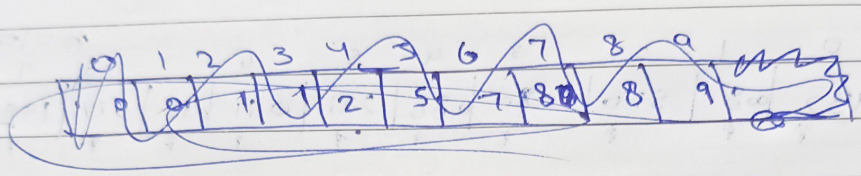


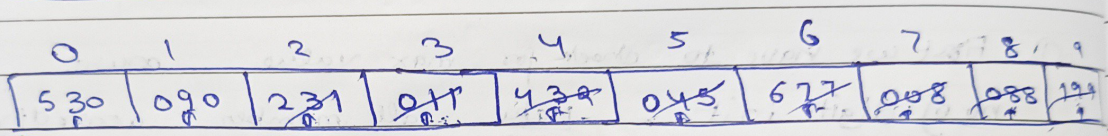
② update count array

Positional array



③ Build array B (acc. to count sort) of LSB

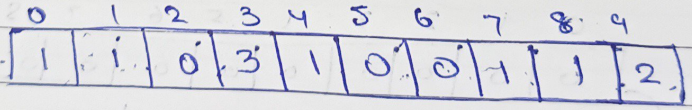




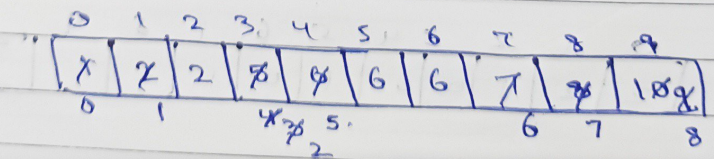
After pass 1

PASS 2. Now again apply count sort on next LSB i.e. tens place digit.

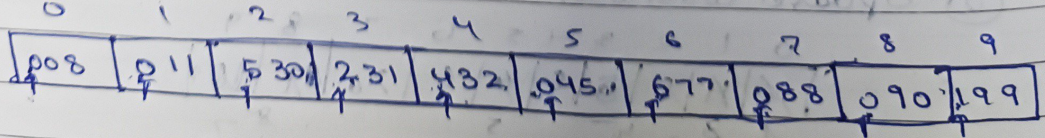
count:



Update count



Build B:



Date | Page No.

Pass 3 — Now again apply count sort for next LSB i.e. hundred's place digit

Count Sor

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

updat

0	1	2	3	4	5	6	7	8	9
5	6	7	7	8	9	10	10	10	10

4 → 5 → 6 → 0

Build B

0	1	2	3	4	5	6	7	8	9
008	011	045	088	090	199	231	432	530	677

CODE:

Main logic

```
RADIX_SORT( A, n) {
int m = get max (A, n)
for (pos = 1; max/pos > 0 ; pos * 10) {
        COUNTSORT (A, n, pos)
    }
}
```

// m = max. element

Max = 677

$\frac{677}{1} = 677$ ✓ (1) time

$\frac{677}{10} = 67$ ✓ (2) time

$\frac{677}{100} = 6$ ✓ (3) time

$\frac{677}{1000} = 0$ (X)

Date
Page No.

POP Radix

```
COUNT_SORT (int A[], int n, int pos) {
    int K = 9
    int count[K+1] = {0};
    for(i=0; i<n; i++){
        ++ count[(A[i]/pos) % 10];
    }
    for( i=1; i<=K; i++) {
        count[i] = count[i] + count[i-1];
    }
    int B[n] = {0}
    for ( i = n-1; i>=0; i++){
        B[--count[(A[i]/pos) % 10]] = A[i]
    }
    for (i=0, i<n; i++){
        A[i] = B[i]
    }
```

T.C = O[(n+K) * d]
d → no. of digits.

S.C = O(n+K)

Date
Page No.

THEORY.

- It sorts number digit by digit, starting from LSB.
- To sort each bit it uses counting sort.

Time Complexity Analysis

If n = no. of ~~digits~~ element
d = no. of digits
k = range of ~~each~~ digit

Then for each count sort T.C = O(n+k)
Count sort ~~runs~~ d times
T.C = O(d*(n+k))

S.C = O(n+k)

Date
Page No.

HEAP

MAX Heap

→ for every node i, the value of node is less than or equal to its parent value.

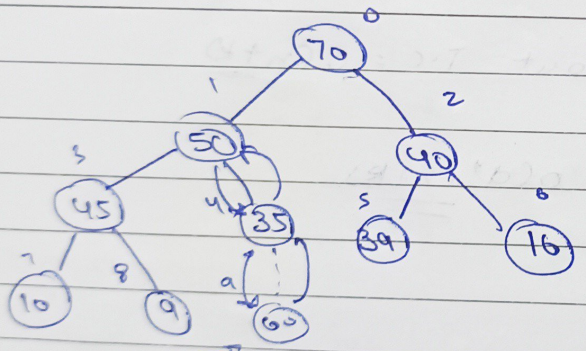
$A[Parent[i]] \geq A[i]$

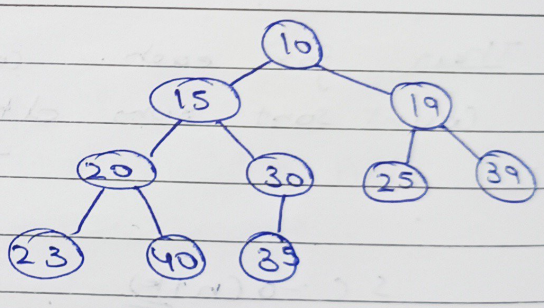
{except root node}

Min heap

for every node i, the value of node is greater than or equal to its parent value.

$A[Parent[i]] \leq A[i]$



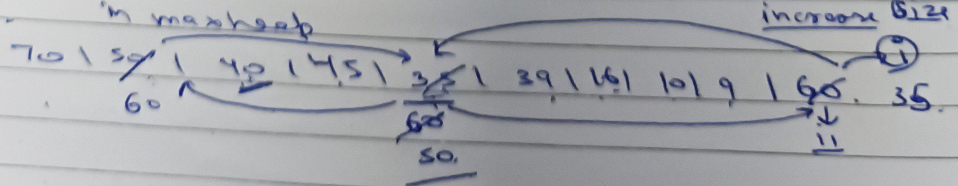


70 | 50 | 40 | 45 | 35 | 39 | 16 | 10 | 9

10 | 15 | 19 | 20 | 30 | 25 | 39 | 23 | 40 | 35

Formula to find parent = $(\text{floor})\left[\frac{i}{2}\right]$

insert 60 in maxheap



Date | Page No.

- During insertion we have to follow property of binary tree to complete it so we insert at leaf side as left as possible.

Max time required to insert an element & swap to its original position can be equal to height of a tree so,

$T(n) = O(\log n)$ { complexity for height of tree }

Insert Heap Func FOR MAx Heap.

```
insert_heap (A, n, value) {
    n = n+1;
    A[n-1] = value;
    i = n-1;
    while (i > 0) {
        Parent = floor((i-1)/2);
        if (A[parent] < A[i]) {
            swap (A[parent], A[i]);
            i = Parent;
        }
        else {
            return
        }
    }
```

Same for min heap insertion just change the case if [A(parent) > A(i)]

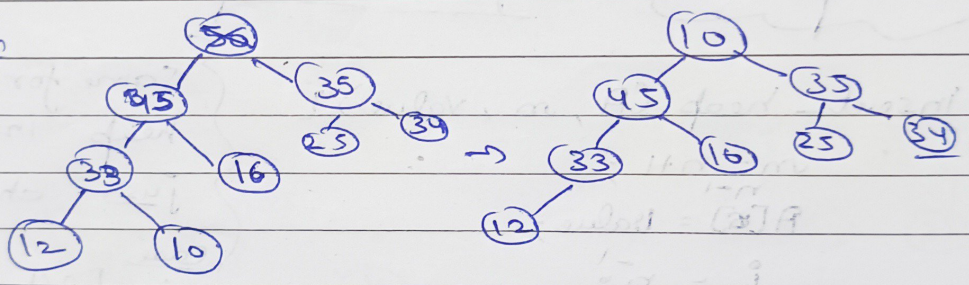
Date
Page No.

Delete in max heap

1. Delete Root node.
2. Fill it with the last element & delete last index
3. Now compare from top to bottom.

left child = $2 * i + 1$ { i = index of Parent }
right child = $2 * i + 2$

Ex



0	1	2	3	4	5	6	7	8
50	45	35	33	16	25	34	12	10

0	1	2	3	4	5	6	7	8
10	45	35	33	16	25	34	12	

(1)

L.C = 1 R.C = 2

compare value = 45

Compare with parent 45 > 10

swap.

45	10	35	33	16	25	34	12

Date | Page No.

L.C = 3

R.C = 4

(3, 4).

Compar = 33

Copare 33 > 10.

Swap

Node i

45	33	35	10	16	25	34	12
0	1	2	3	4	5	6	7

L.C = 7

R.C = X.

compar 12 > 10.

swap.

45	33	35	12	16	25	34	10

0	1	2	3	4 : i
4	1	3	2	16 9 ✱ ● 10 14 8 7

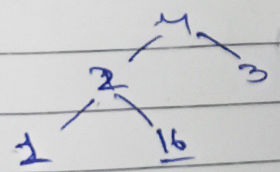
i=0

pos = $\lfloor i/2 \rfloor$

A[i] > A[pos]

swap. (2, 1)

i = pos



i=3 ① ①

Heap Sort → Build max heap. → delete all data one by one & you will get sorted heap.

HEAP SORT

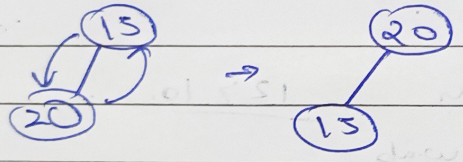
Build Max heap :-

	0	1	2	3	4
A =	15	20	7	9	30

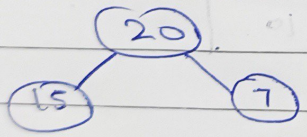
max-heap
use insert fn heap also

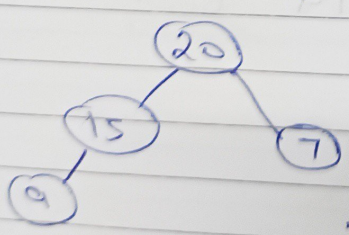
① (15)

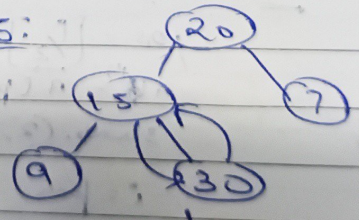
After every insertion we have to check property of max heap.

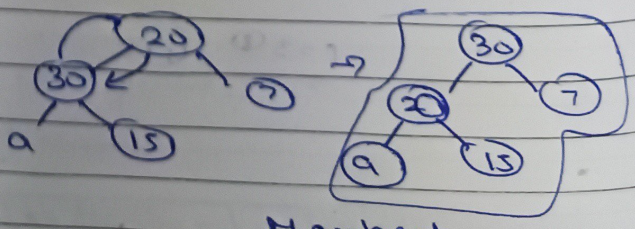
② 

every step must form complete binary tree

③ 

④ 

5: 

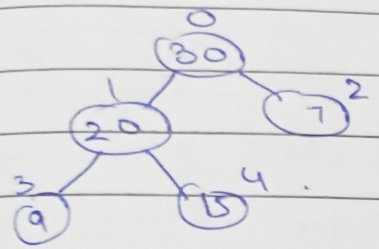


Maxheap

So now array = [30, 20, 7, 9, 15]

Now deletion in max heap

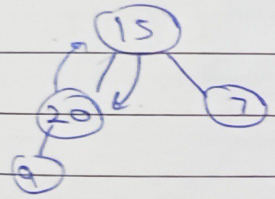
After we deletion we have to satisfy max heap (1)



0	1	2	3	4
30	20	7	9	15

Delete 30 & place 15 there

i→ 0	1	2	3	4
15	20	7	9	30



L.C = 2i+1 = 20

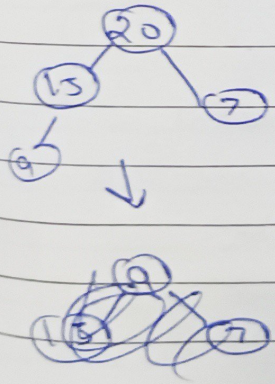
R.C = 2i+2 = 7.

20 > 7.

compare 20 > 15.

swap

0 (i)	1	2	3	4
20	15	7	9	30



~~Now delete 20 & replace root with 9~~

0	1	2	3	4
9	15	7	20	30

L.C = 15

L.C = 7

R.C = 9 9 > 7

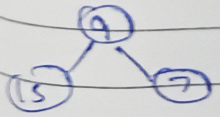
But 9 ≯ 15

No swap

(2)

Now delete 20 & replace 9.

i→ 0	1	2	3	4
9	15	7	20	30



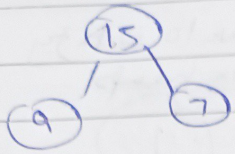
L.C = 15.

R.C = 7.

L.C > R.C

15 > 9 Swap

15	9	7	20	30



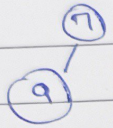
15	9	7	20	30

No swap

(3)

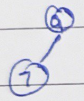
Delete 15 , replace 7

7	9	15	20	30



L.C = 9 ~~7~~ 9 > 7.

~~No Swap~~ swap 9 and 7



9	7	15	20	30

No swap

delete 9 & replace (7)

7	9	15	20	30

= sorted array

T.C

To insert an element

T.C = O(logn)

To insert n element

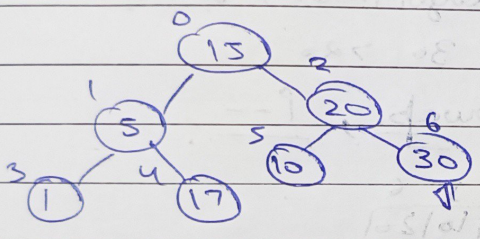
T.C = O(nlogn)

Date
Page No.

FOR Deletion of n element
$O = (n \log n)$

* As insertion takes $O(n \log n)$ time so we use heapify method to build max heap in T.C $= O(n)$.

HEAPIFY METHOD



	0	1	2	3	4	5	6
A =	15	5	20	1	17	10	30

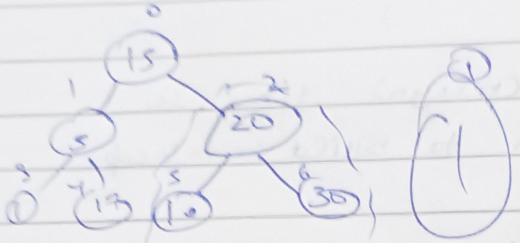
We start from leaf node and check for its child since till 1 there are leaf node so each one is its own max heap.

so we have to check from 20 to build a particular max heap

so we have to start from for (0 based index) $i = \text{floor}\left[\frac{n-1}{2}\right]$ down to 1

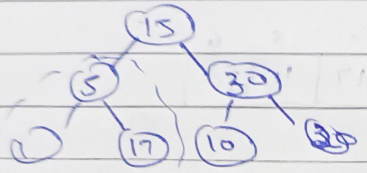
as $\left[\frac{i}{2}\right]$ to n are leaf node. →

Heapify method void start from $\frac{(n-1)}{2}$



i = 2

L.C → 2i+1
R.C → 2i+2
} compare 30 > 10



compare A[larger index] & A[i]
30 > 20
Swap, i--

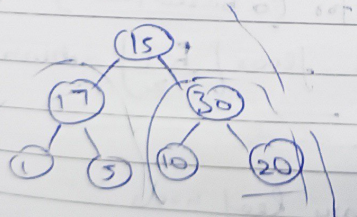
Array →

0	1	2	3	4	5	6
15	5	30	1	17	10	20

2 → i = 1

{ Also check again for left child & right child befor i --

L.C = 2i+1 = 1
R.C = 2i+2 = 17
} 17 > 1



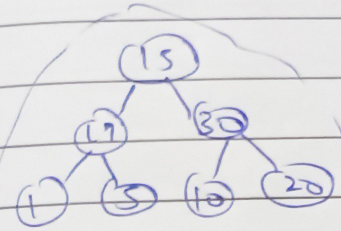
A[R.C] > A[i]
Swap, i--

0	1	2	3	4	5	6
15	17	30	1	5	10	20

Date
Page No.

(3)

$i = 0$



L.C = $2i+1 = 17$.

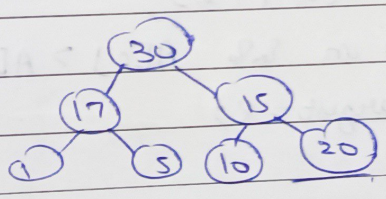
R.C = $2i+2 = 30$

$30 > 17$

$A[R.C] > A[i]$

$30 > 15$

swap.



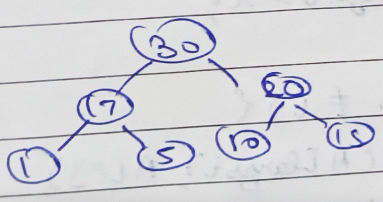
0	1	2	3	4	5	6
30	17	15	1	5	10	20

L.C = 10

R.C = 20

$20 > 15$

swap



T.C → O(h)

HEAP SORT

Code for MAX Heapify method

```
MAX-Heapify (A, n, i) {
    int largest = i;
    int l = (2*i + 1);
    int r = (2*i + 2);
    if (l <= n && A[l] > A[largest]) {
        largest = l;
    }
    if (r <= n && A[r] > A[largest]) {
        largest = r;
    }
    if (largest != i) {
        swap (A[largest], A[i])
        Max-heapify(A, n, largest)
    }
```

O(n)

```
HeapSort (A, n) {
    int i = floor [ (n-1)/2 ];
    while (i >= 0) {
        Max Heapify (A, i);
    }
```

//

```
Heap_Sort (A, n) {
    int j = floor((n-1)/2)

    Build maxheap:
    while (j >= 0) {
        Max_heapify(A, n, j);
        j--;
    }

    for deletion:
    for (i = n-1; i >= 0; i--) {
        swap(A[0], A[i]);
        MAX_heapify(A, i-1, 1);
    }
```

T.C -> O(n log n)