Unit - IV

Bubble sort

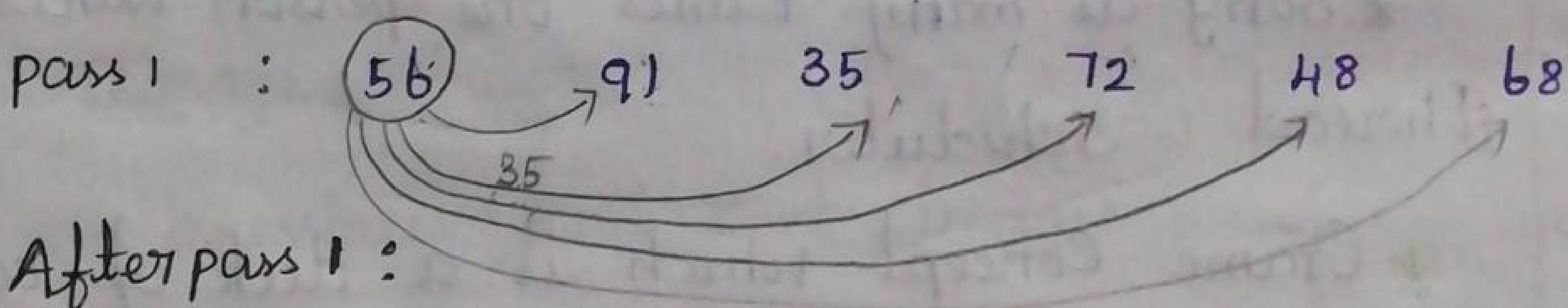
Bubble sort

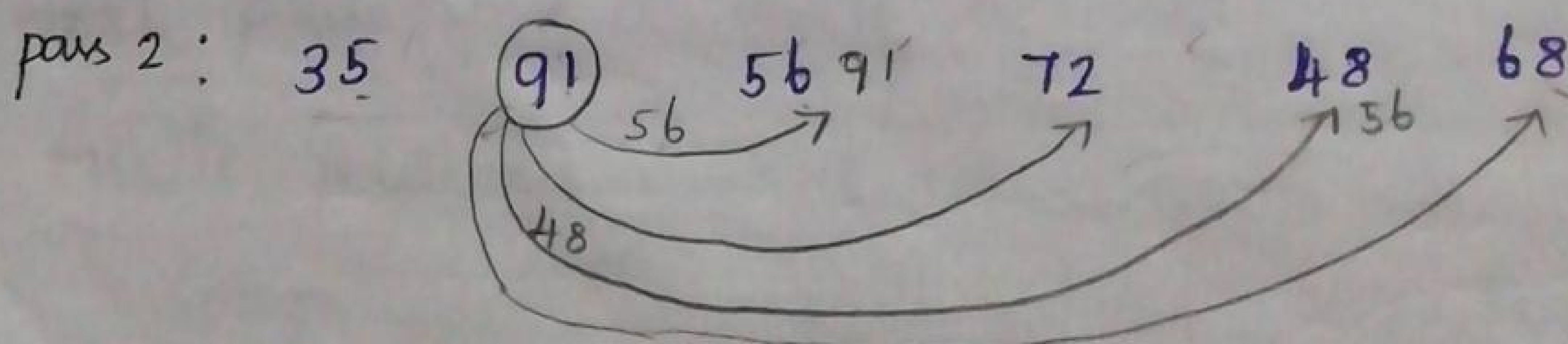
* It is the aldest sort
* It is also reffered as sinking sort.

procedure:

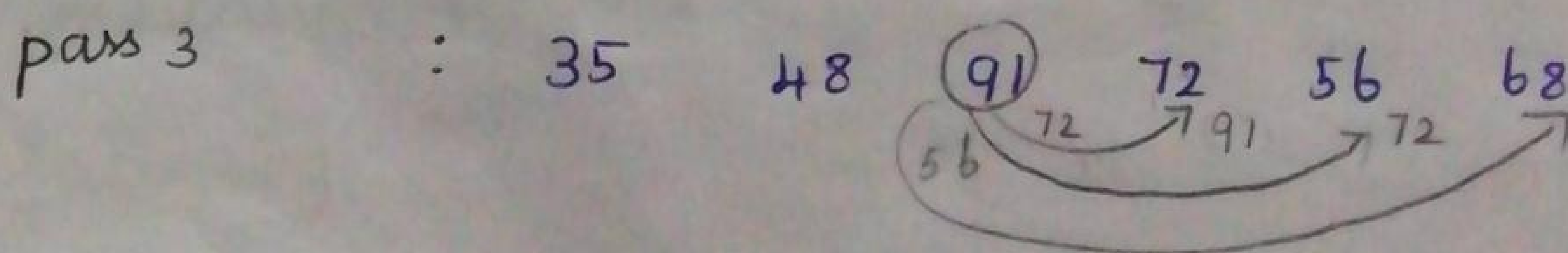
* It Repeatly move the smallest element to the lowest Index position in the list
* It Repeatly compares two consi ~~elem~~ elements and moves the smallest among them to the left.
* This process is repeated until all the elements are in the correct order.

Example: 56 91 35 72 48 68





After pass 2 : 35 48 91 72 56 68



Scanned by TapScanner

After pass 3 : 35 48 56 (91) 72 68

pass 4 : 35 48 56 68 (91) 72

After pass 4 : 35 48 56 68 72 91

Solution : 35 48 56 68 72 91

Write Explanation in Exam

Algorithm :

```
void bubblesort ( int a[ ], int n)
{
  int i, j, tmp;
  For( i=0; i<n; i++)
  {
   For( j= i+1; j<n; j++)
   {   if (a[i] >a[j])
       {
         tmp = a[i];
         a[i]=a[j];
         a[j] = tmp;
       }
   }
  }
```

Scanned by TapScanner

Insertion sort

Let's take this array

5	1	6	2	4	3

As we can see here, in insertion sort, we pick up a key and compares it with elements a head (ahead) of it, and puts the key in right place.

* File has nothing before it.

* One is compared to file and is inserted before file.

* six is greater than 5 and 1

* 2 is smaller than 6 and 5, but greater than 1, so it's is inserted after 1 and this goes on.

Example:

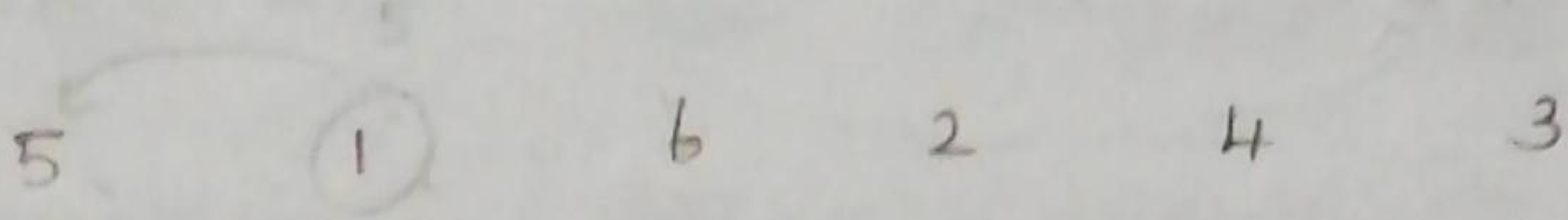
5	1	6	2	4	3

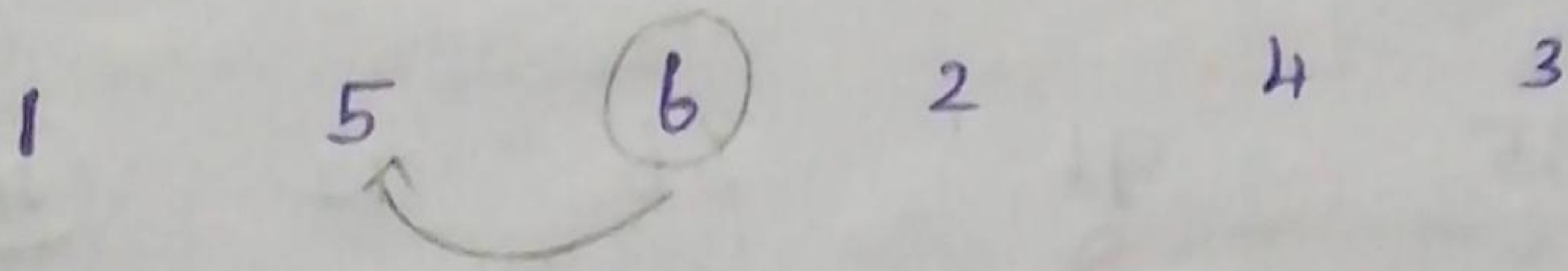
Always we start with the second element as key

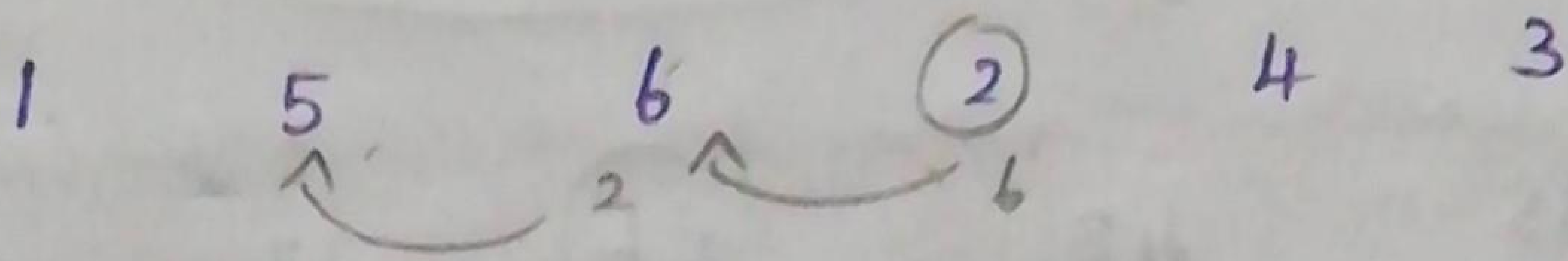
Scanned by TapScanner

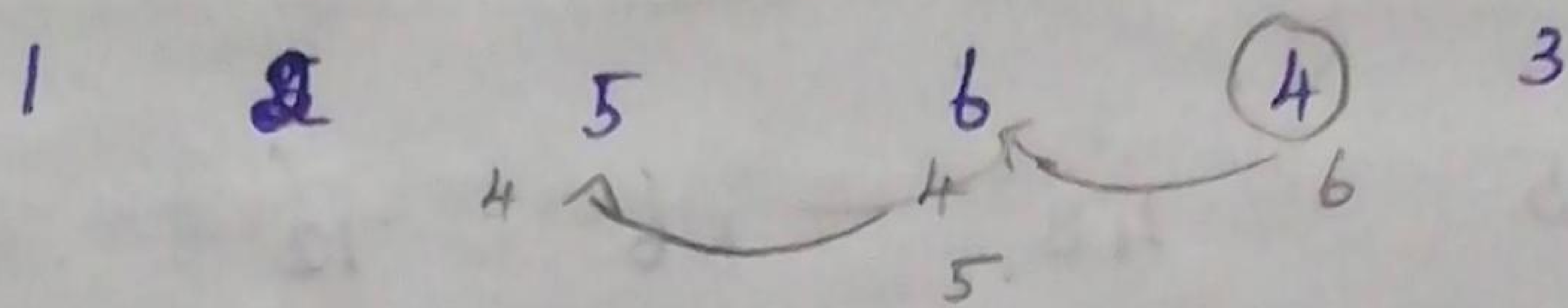
Example

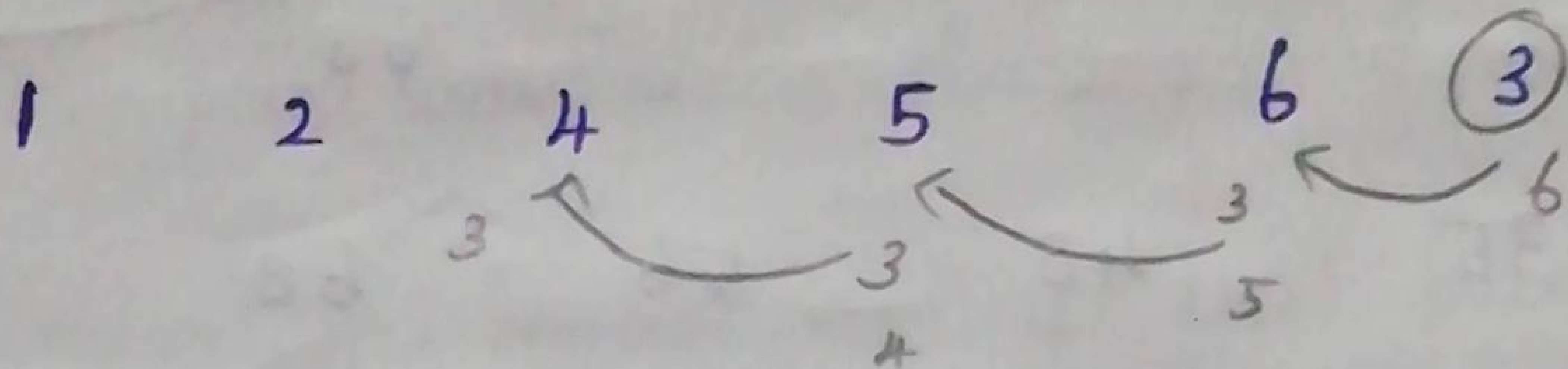
2nd element as key











solution: 1 2 3 4 5 6.

Scanned by TapScanner

Insertion sort algorithm:

```
Void insertion_ sort ( int a[ ], int n)
{
    int i, p, tmp;
    For (p=1; p<n; p++)
    {
        tmp = a[p];
        For ( j = p; j >0 && a [j-1] >tmp; j--)
            a [j] = a[ j-1];
        a [j] = tmp;
    }
```

Scanned by TapScanner

Selection sort :

It is one of the most basic sorting Techniques.

It works on the principals of identifying the smallest elements in the list and moving it to beginning of list this process is repeated untill all the elements in the list are sorted.

Example :

(Minium element)

Scanned by TapScanner

56 91 (35) 72 48 68

35 56

35 91 56 72 (48) 68

48 91

No change ← 35 48 (56) 72 91 68

35 48 56 72 91 (68)

68 72

35 48 56 68 91 (72)

72 91

35 48 56 68 72 91

Solution: Sorted Array / list is:

35 48 56 68 72 91

Algorithm:

```
Void Selection Sort ( int a [ ], int n)
{
  int i, j, k, min ;
  For (i= 0 ; i<n-1 ; i++)
  {
    min = a [i];
    For (j= i+1; j<n; j++)
    {
    if (a[j] < min)
    {
      min = a [j];
```

Scanned by TapScanner

```
        a[j] = a[i];
        a[i] = min;
      }
    }
}
```

Efficiency And selection sort

* The no. of. comparisons made during 1st pass} = n-1

No. of. comparisons made during 2nd pass} = n-2

No. of. comparisons made during last pass} = 1

Total No. of. comparisons = $(n-1) + (n-2) \cdots + 1$

$= n * (n-1)$

$= O(n^2)$

efficiency of selection sort = $O(n^2)$.

Scanned by TapScanner

Quick sort

* It's sort any list very Quickly

* It is not a stable search, but it is very fast and requires very less additional space.

* It is based on the rule of divide and conquire (partitian exchange sort).

* This algorithm divides the list in two Three main parts.

Scanned by TapScanner

1. elements less than the pivot element
2. pivot element (central element)
3. Elements greater than the pivot element.

Complaxity Analysis :

* worst case time complaxity = $O(n^2)$
* Best case time complaxity = $O(n \log n)$
* Space complaxity. order of $n \log n$.

Quick sort conditions :

1. i value → small
   j value → large
   } pivot
   else
   Swap(i, j)

2. i value → large
   j value → small
   (or) $i \geq j$
   Swap (p, j)

Example : pivot element

50 30 10 90 80 20 40 70

pivot element 50, comparing i & j value 30 and 70, condition is True. No needed swap

Step 1 : 50 30 10 90 80 20 40 70
P (50), i (30), j (70)

Step 2 : 50 30 10 90 80 20 40 70
P (50), i (10), j (70)

In step 2 p is pivot 50 i value is 10 j value is 70 Condition is True no neede

step 3 : 50 30 10 90 80 20 40 70
P i× j✓

pivot value (p) is 50, i condition is True
j condition false.

step 4 : 50 30 10 90 80 20 40 70
P i× 80 90 j×
(swap i, j)

pivot value is 50, Then comparing i and j value condition is false. so swap is needed.

step 5 : 50 30 10 40 80 20 90 70
P i (greater) j✓ move

i value should be increment. j value should not increment

step 6 : 50 30 10 40 80 20 90 70
P i not obey j Swap(i, j)
j

pivot value (p) is 50, i and j condition is false

step 7 : 50 30 10 40 20 80 90 70
P i j

Then ij value is 20

step 8 : 50 30 10 40 20 80 90 70
P j greater i × move

i is greater than j so we can implement 2nd condition and swapping pivot element and j element

step 9 : 50 30 10 40 20 80 90 70
P j i swap(p, j)
20 50

step 10 : 20 30 10 40 50 80 90 70
(i) (j) (i)

When the pivot element move to the center, divide the array left & right

Scanned by TapScanner

Step								
Step 11 :	20 (P)	30 (i)x	10	40 j	50	80 P	90 i	70 j
Step 12 :	20 P	30 ix	10 jx	40 Swap (i,j)		80 P	70 i	90 j
Step 13 :	20 P	10 i	30 j✓	40		80 P	70	90 (ij)
Step 14 :	20 P	10	30 (xi, j)	40		80 P	70 j	90 i
Step 15 :	20 P	10 jx	30 ix $i \geq j$ Swap (i,j)	40		70	80	90
Step 16 :	10	20	30	40	50	70	80	90

Algorithm :

i) if $l < r$

$s \rightarrow$ partition $(A(l \ldots r))$

Quicksort $(A(l \ldots s-1))$

Quicksort $(A(s+1 \ldots r))$

ii) $l \leftarrow A[l]$

$i \leftarrow l$, $j \leftarrow r+1$

repeat

repeat $i \leftarrow i+1$ until $A[i] \geq P$

repeat $j \leftarrow j-1$ until $A[j] \leq P$

Swap $[A[i], A[j]]$

until $i \geq j$

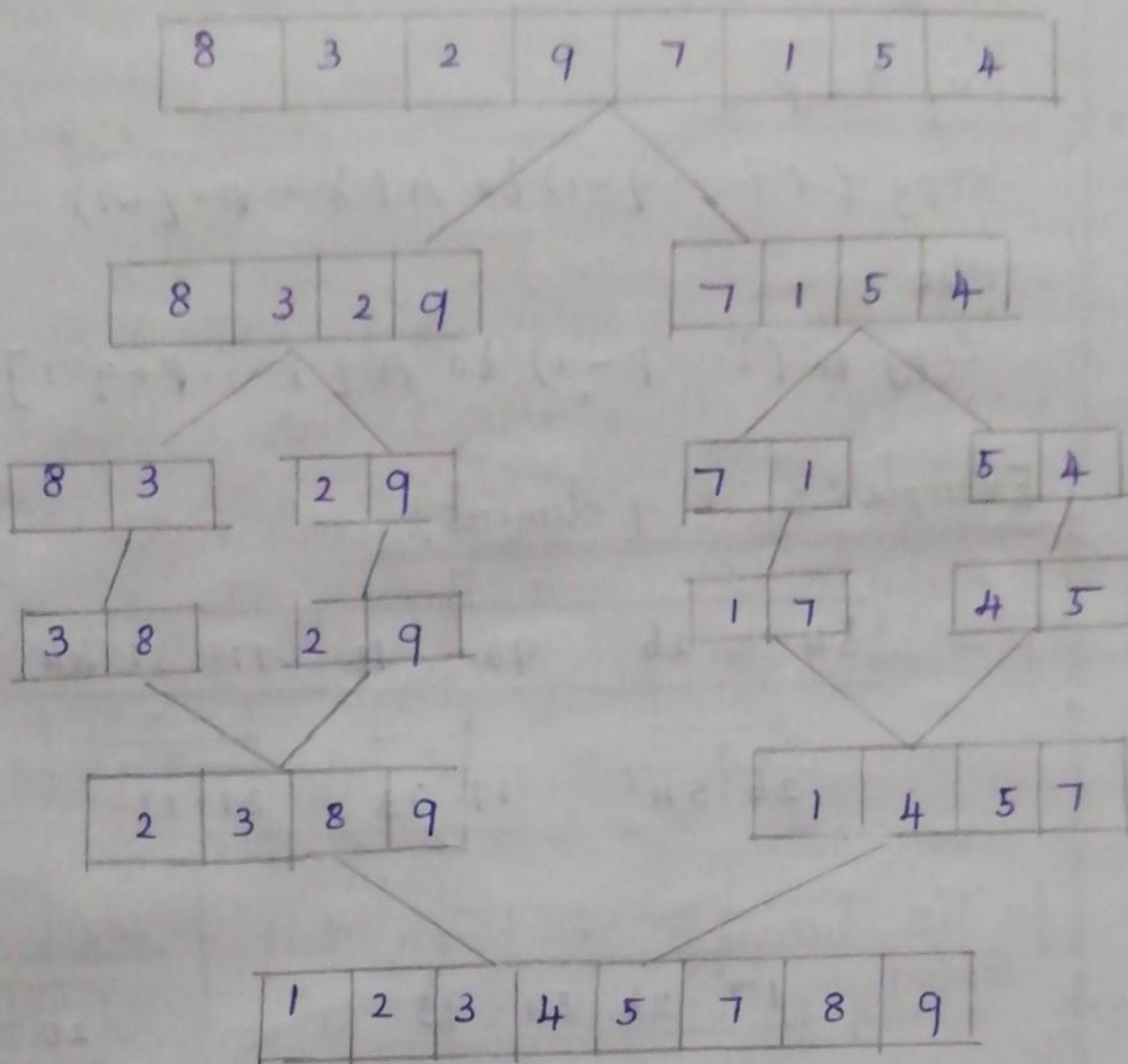
Swap $(A[i], A[j])$

Swap $(A[l], A[i])$

Scanned by TapScanner

Merge Sort

Example



Algorithm

i. if n > 1

copy A[(n/2) - 1] to B(0 ··· (n/2) - 1]

copy A[n/2 ··· n-1] to C[0 ··· (n/2) - 1]

mergesort (B[0 ··· [n/2] - 1)

mergesort (C[0 ··· [n/2] - 1)

merge (B, C, A)

ii. i ← 0, j ← 0, k ← 0

while i < p & j < Q do

if B[i] ≤ [j]

A[k] ← B[i];

i ← i+1

Scanned by TapScanner

A [k] ← [j];

j ← j + 1

k ← k + 1

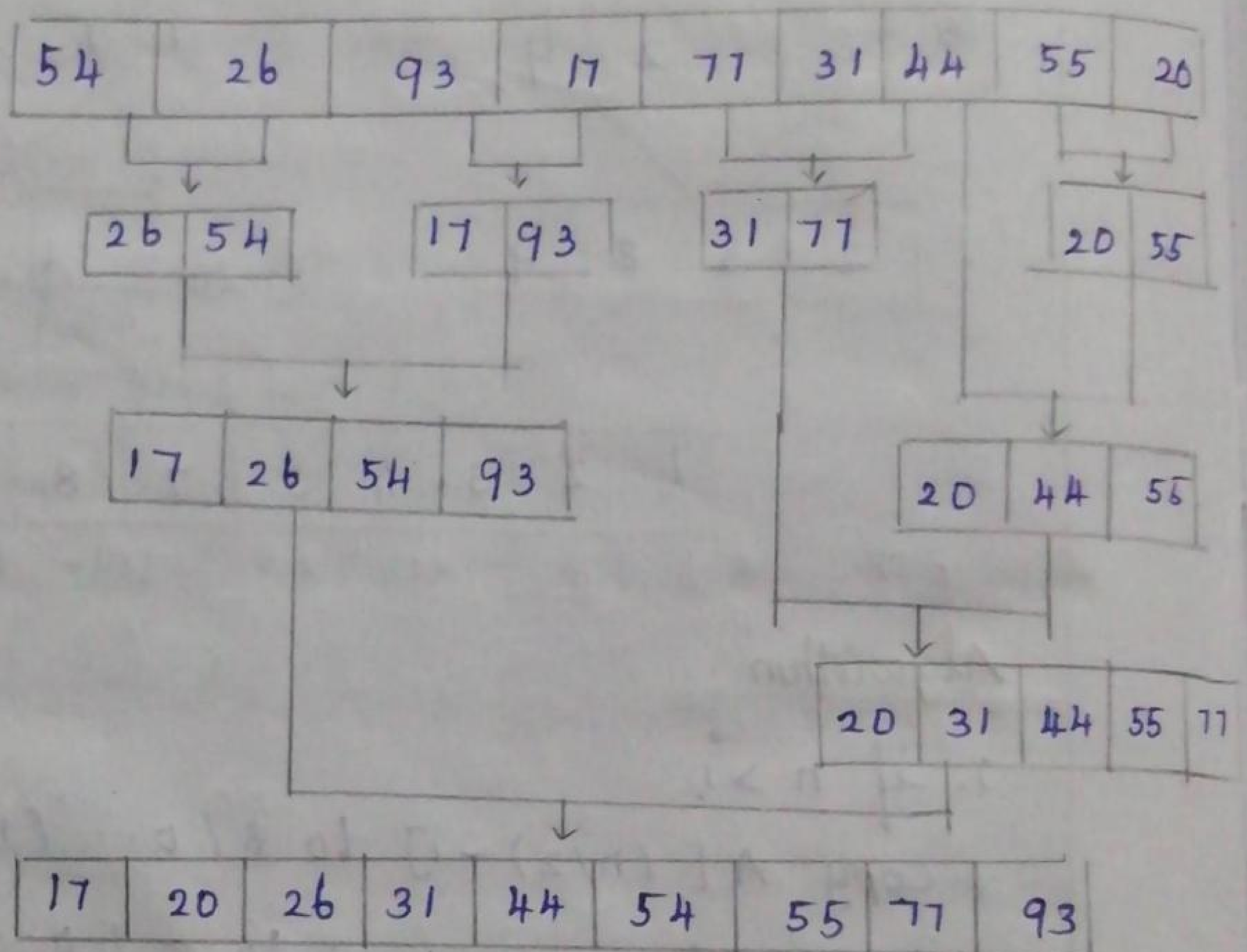
if i = p

copy ((j ⋯ q − 1) to A (k ⋯ p + q − 1)

else

copy B (i ⋯ p − 1) to A [k ⋯ p + q − 1]

Example : 2 (9 element)



Warst case time complexity (Big-O) : O(n * logn)

Best case time complexity (Big-omega) : O(n * log n)

Avgerage case time complexity (Big-Theta) : (n * logn)

Space case time complexity : O(n)

Scanned by TapScanner

Radix sort

It is a generalized from of bucket Sort. It distributes the list of elements across different buckets from 0 to 9

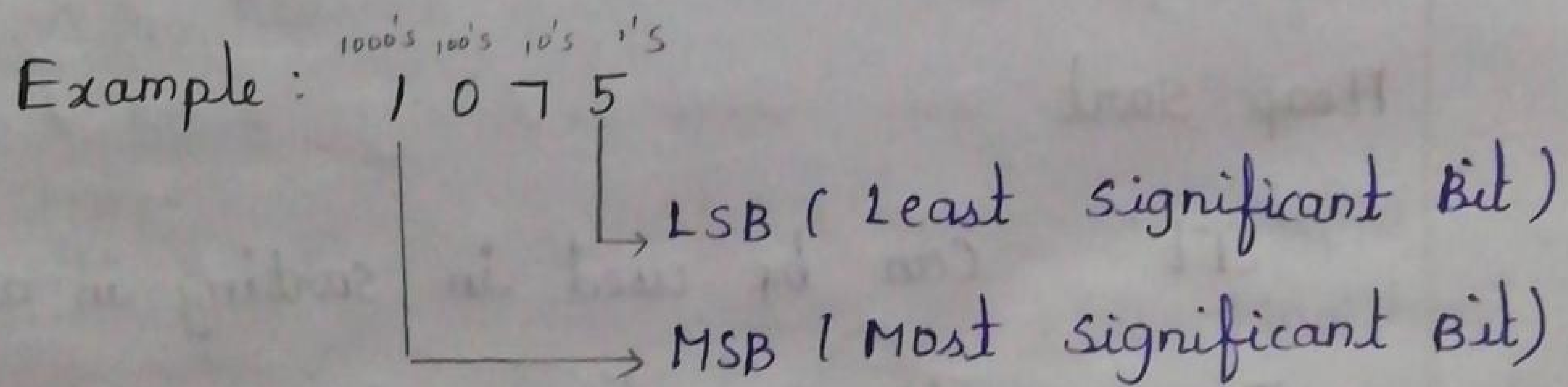
proceger: procedure

* In first pass, all elements are sorted accarding to the LSB (Least Significant Bit)

* In second pass, the numbers are arranged according to the second least significant bit

* This process is repeated until it reaches the most significant bit of all numbers in the number of passes depends upon number of digits

Example:



Example 1:

64, 8, 216, 512, 27, 729, 0, 1, 343, 125

pass 1: (LSB)

0	1	512	343	64	125	216	27	8	729
0	1	2	3	4	5	6	7	8	9

After 1 pass	00	01	512	343	64	125	216	27	08	729

pass 2 : (Second LSB)

0	1	2	3	4	5	6	7	8	9
0, 1, 8	512, 216	125, 27, 729		343		64			

After pass 2

008		729		
001	216	027		
000	512	125	343	064

pass 3 : (Third Least Significant Bit)

0	1	2	3	4	5	6	7	8	9
0, 1, 8, 27, 64	125	216	343		512		729		

0, 1, 8, 27, 64, 125, 216, 343, 512, 729

Scanned by TapScanner

Heap Sort

It can be used in sorting in array

It may be Ascending order or descending order.

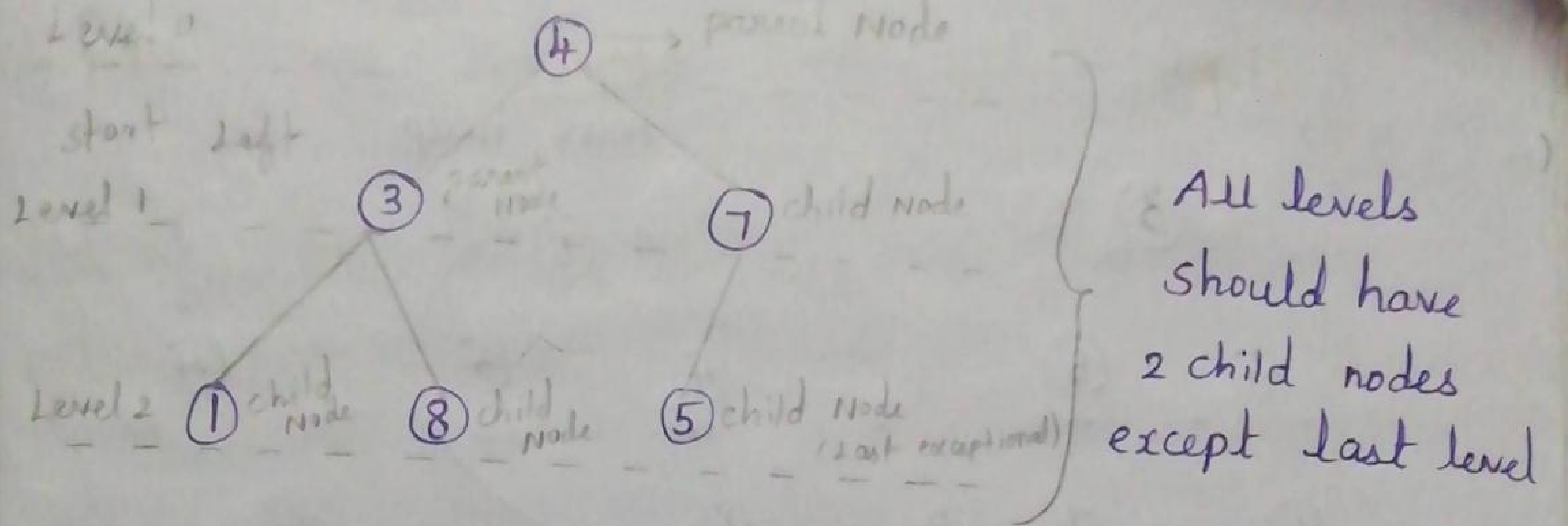
1. complete Binary Tree (CBT)

2. Max heap or min heap

CBT(complete Binary Tree)

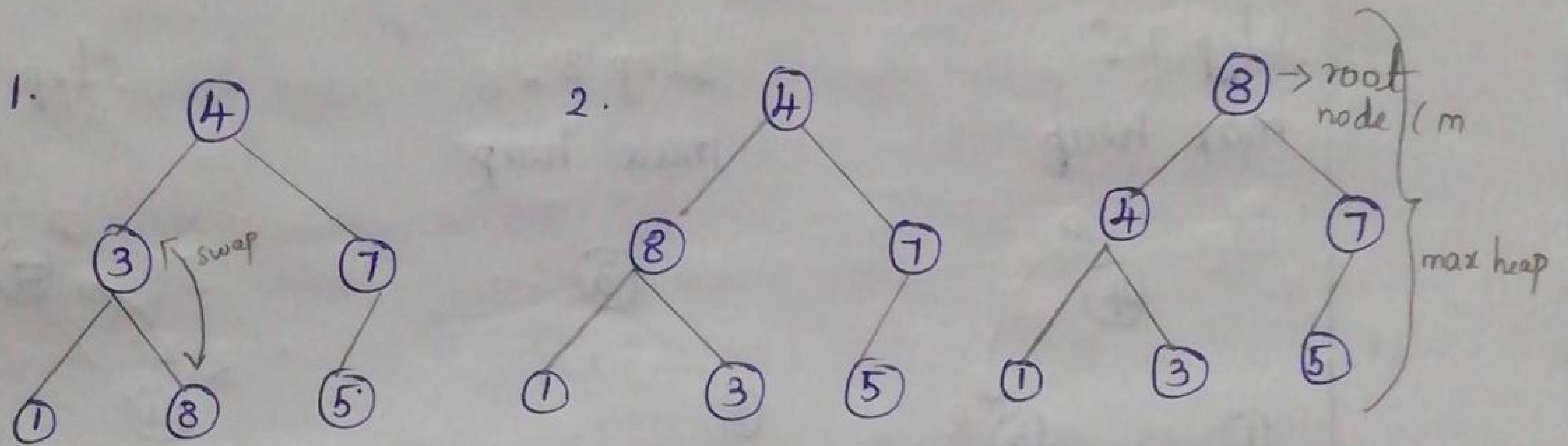
Example: 4, 3, 7, 1, 8, 5

Scanned by TapScanner



Max heap :

Example: 4, 3, 7, 1, 8, 5

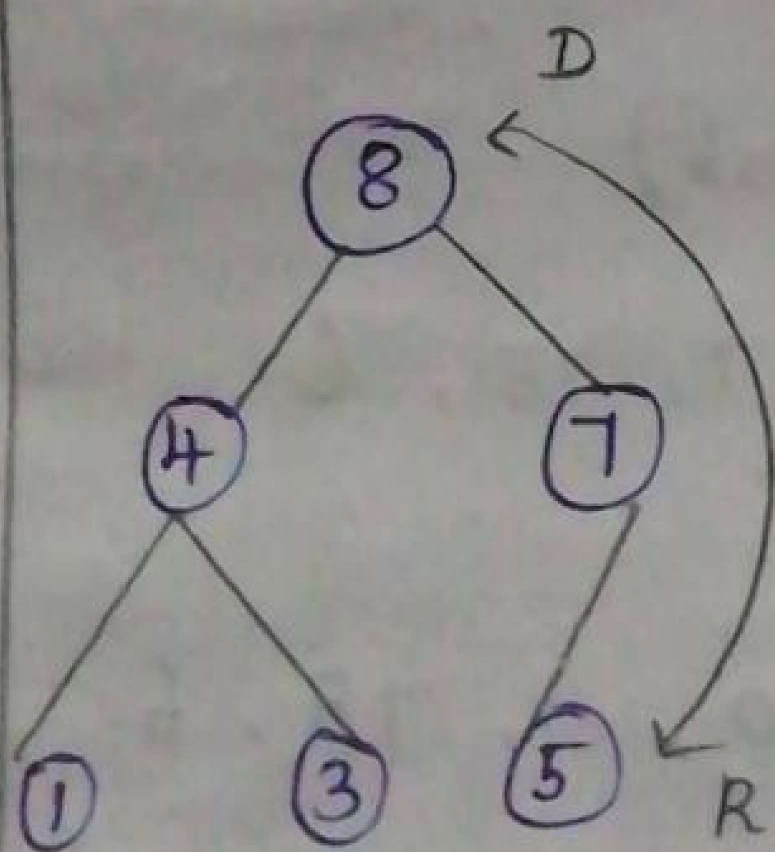


* In max heap, Maximum element will always be at the root

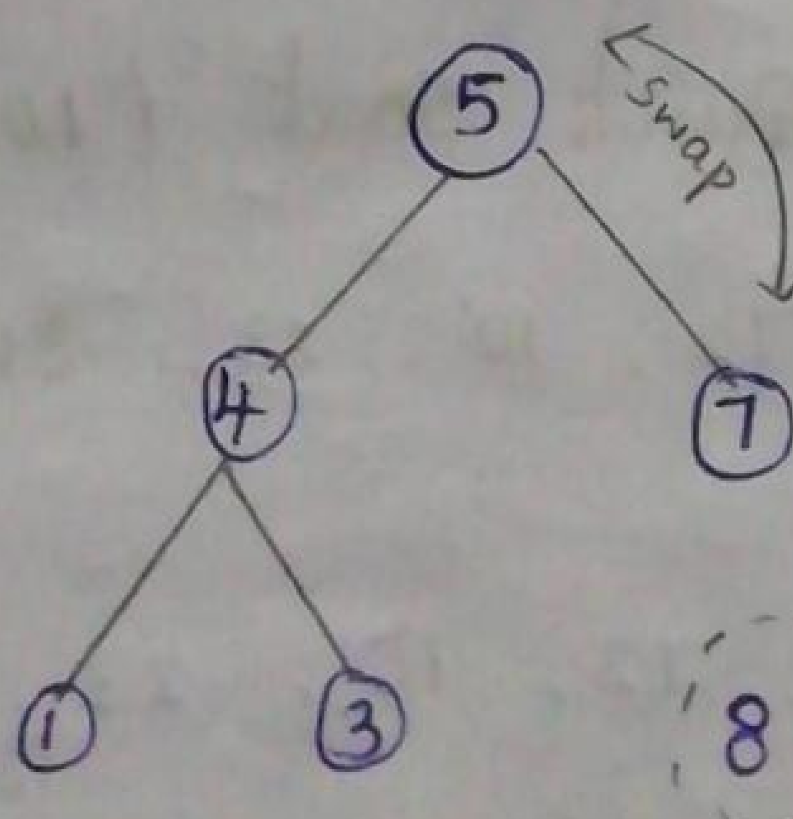
Max heap :

* Replace & delete the 1st node from the Tree.

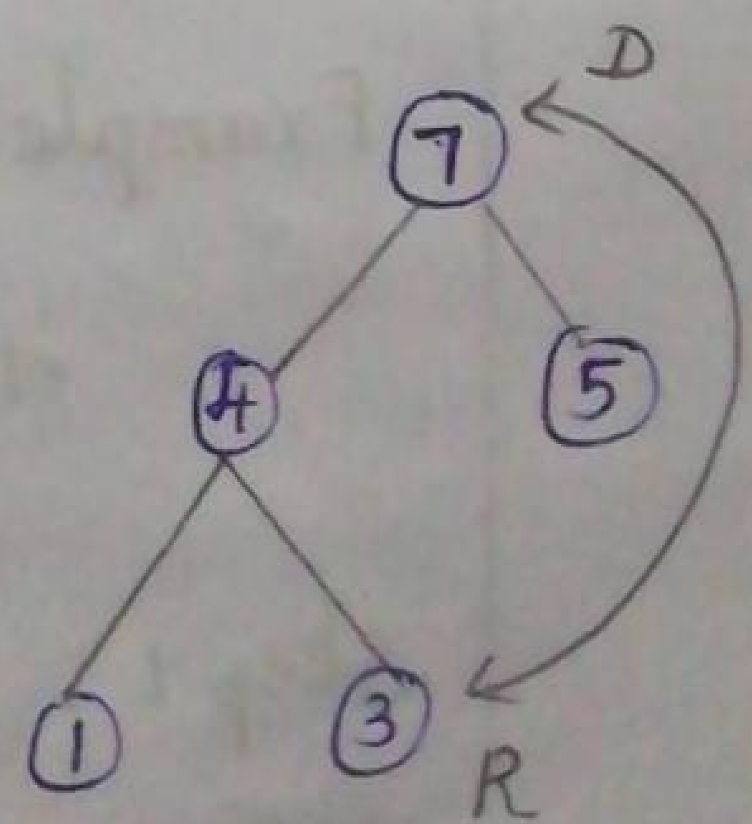
step 1:
Initial



step 2:



step 3:
Max heap

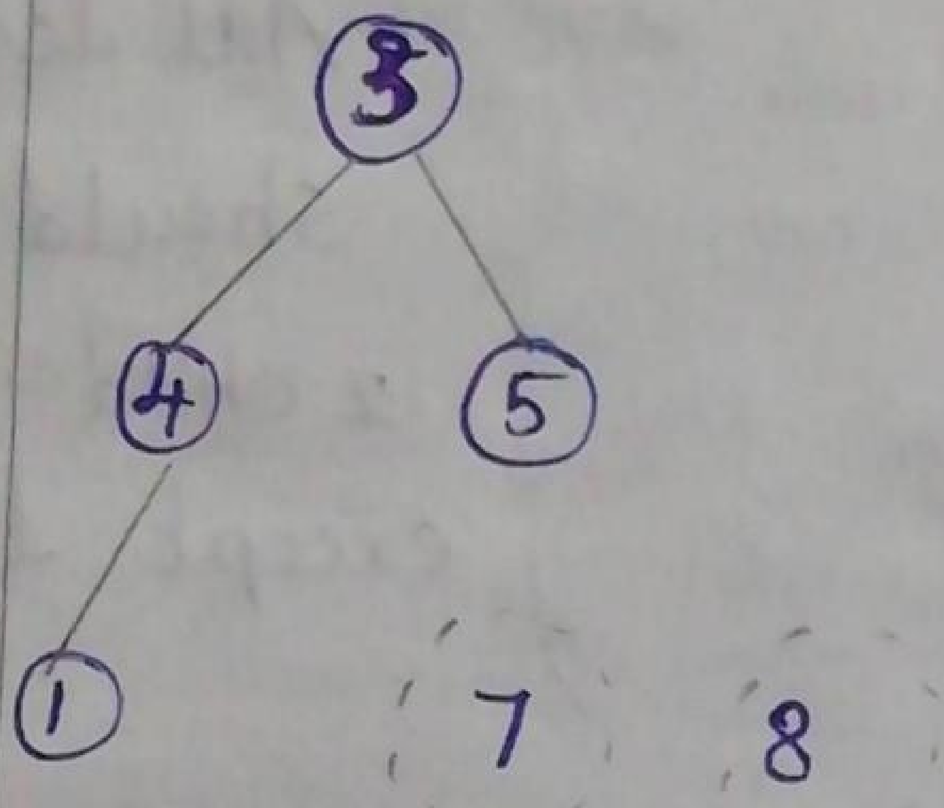


* R - Replace

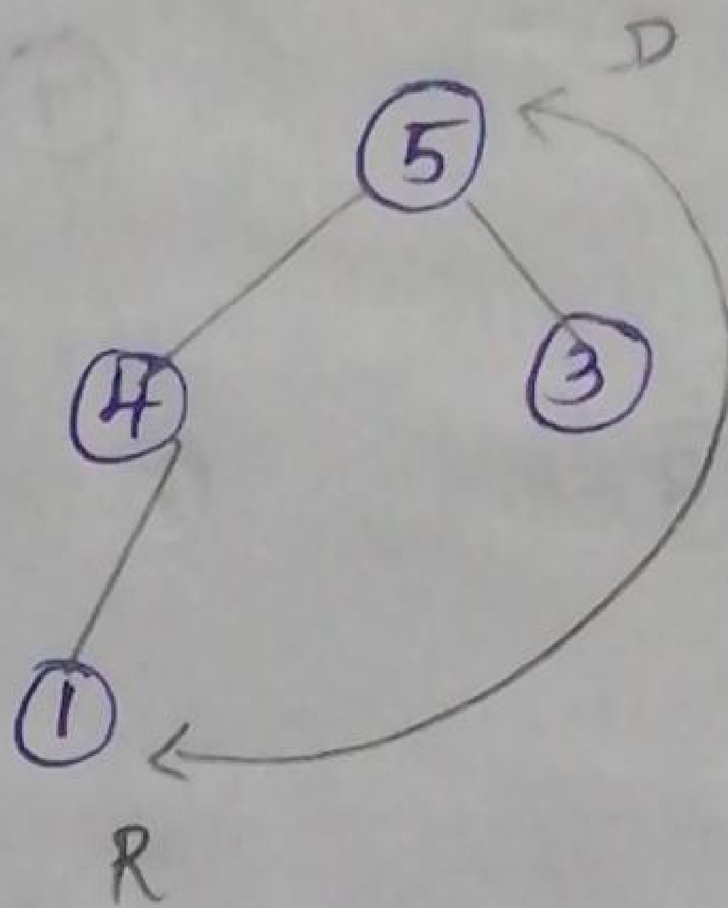
* D - Delete

Scanned by TapScanner

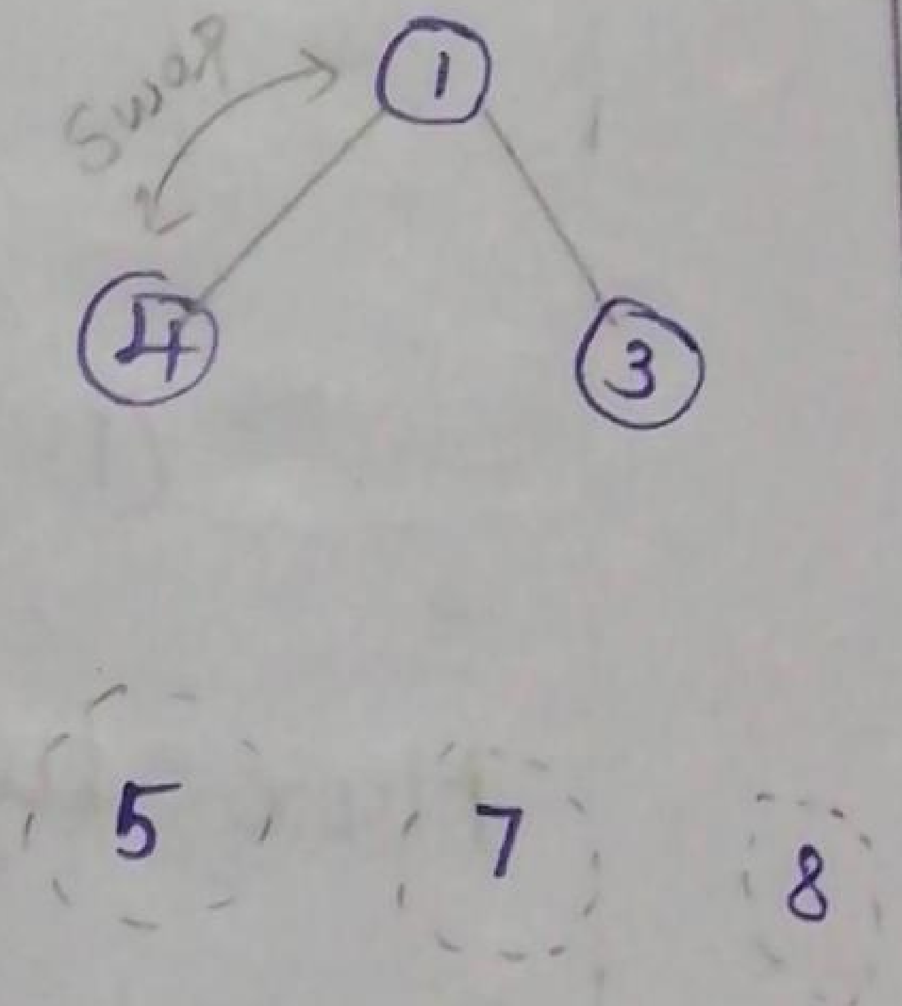
step 4:



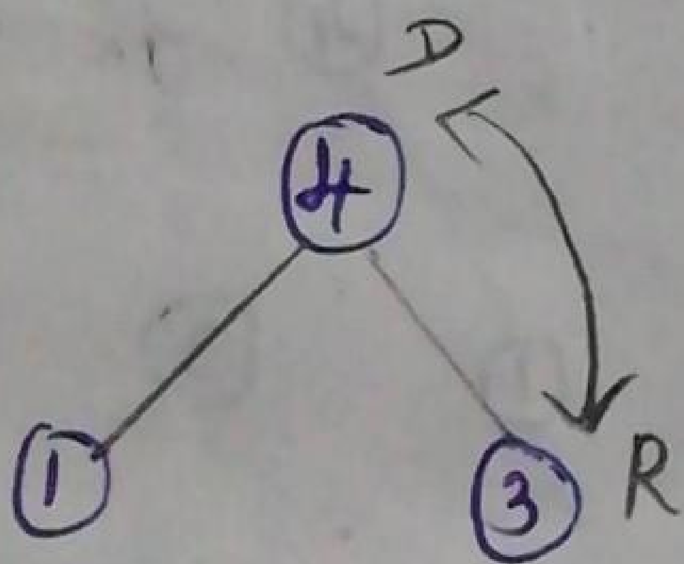
step 5:
max heap



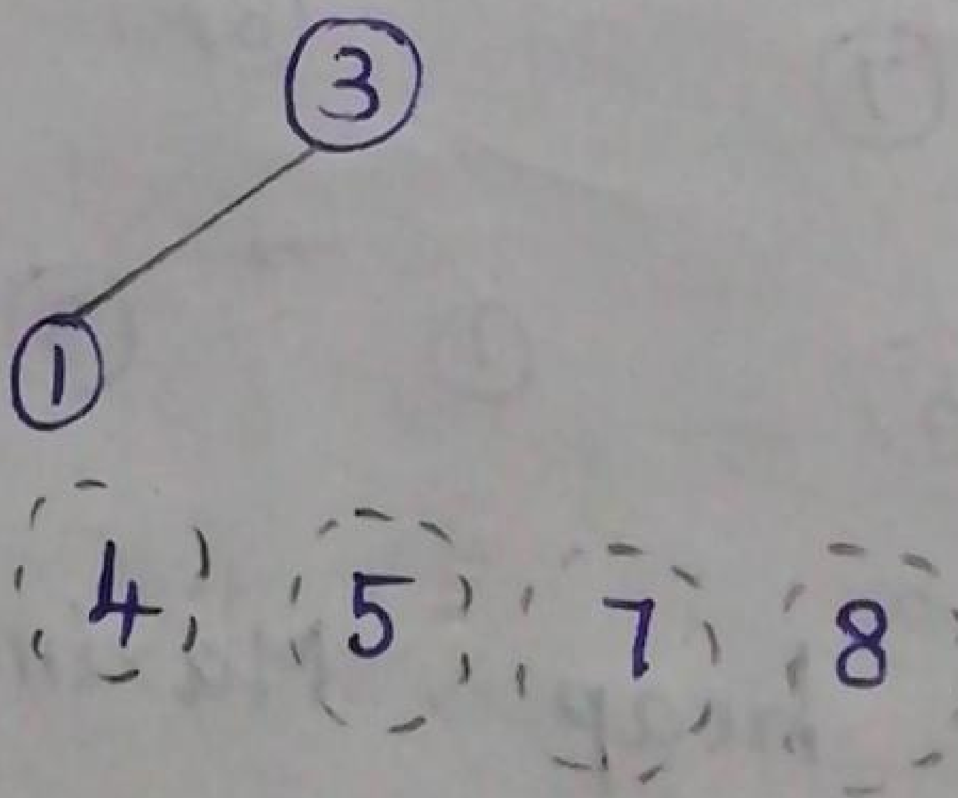
step 6:



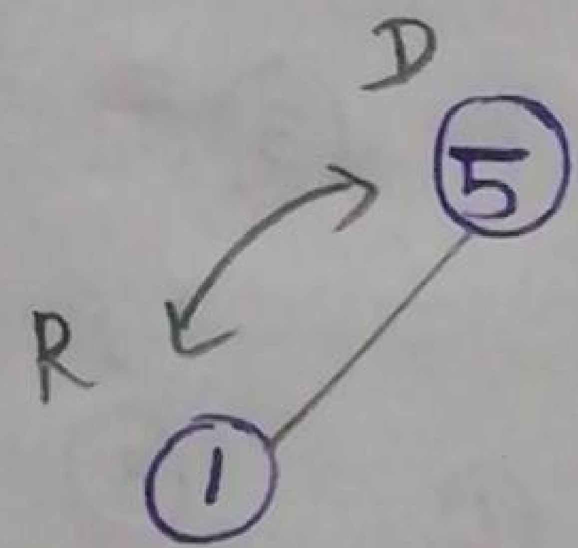
step 7:
max heap



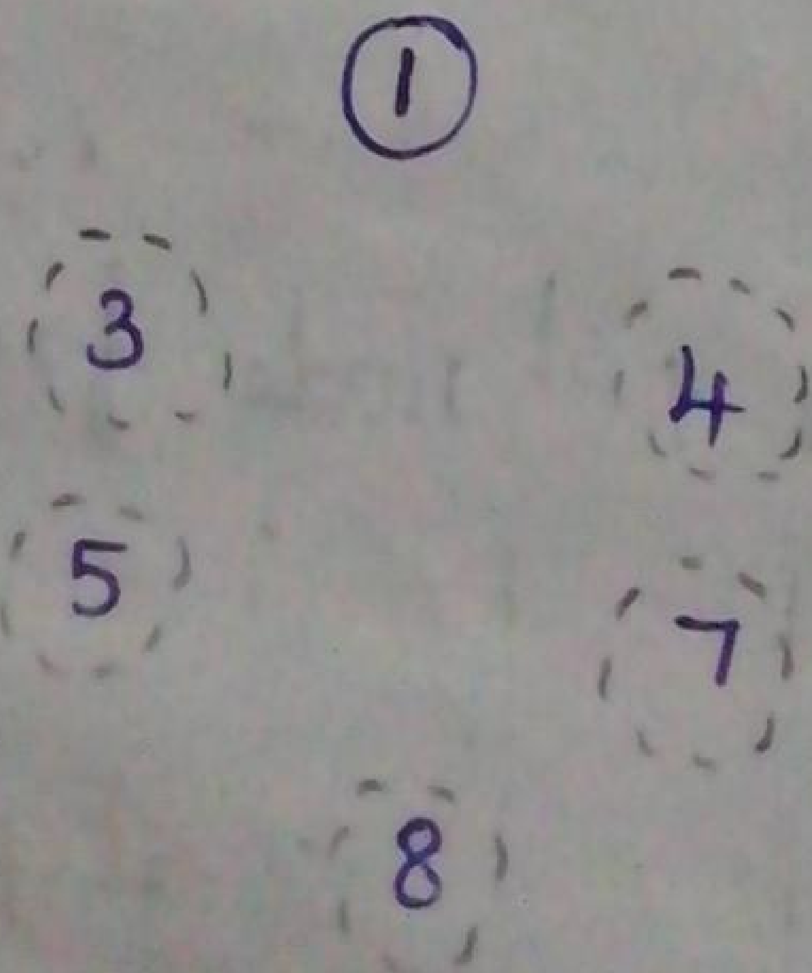
step 8:
max heap



step 9:



step 10:

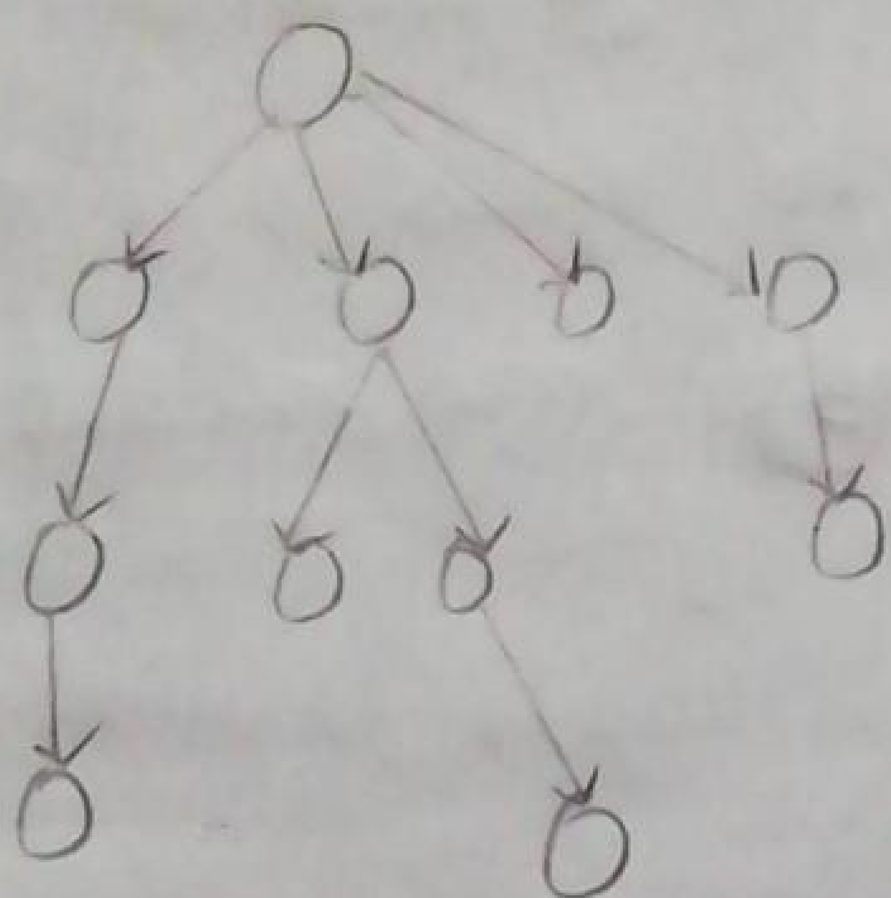


Sorted Array 1, 3, 4, 5, 7, 8

Scanned by TapScanner

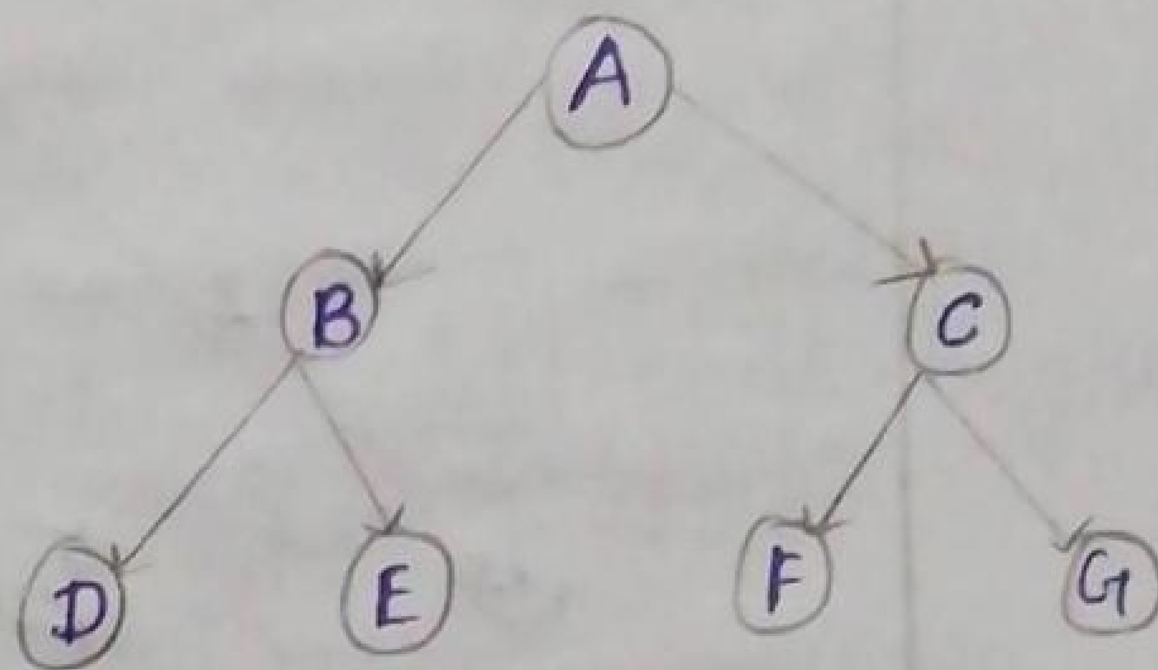
Binary Tree

* Non Linear
* maximum of 2 children



henerel Tree

VS



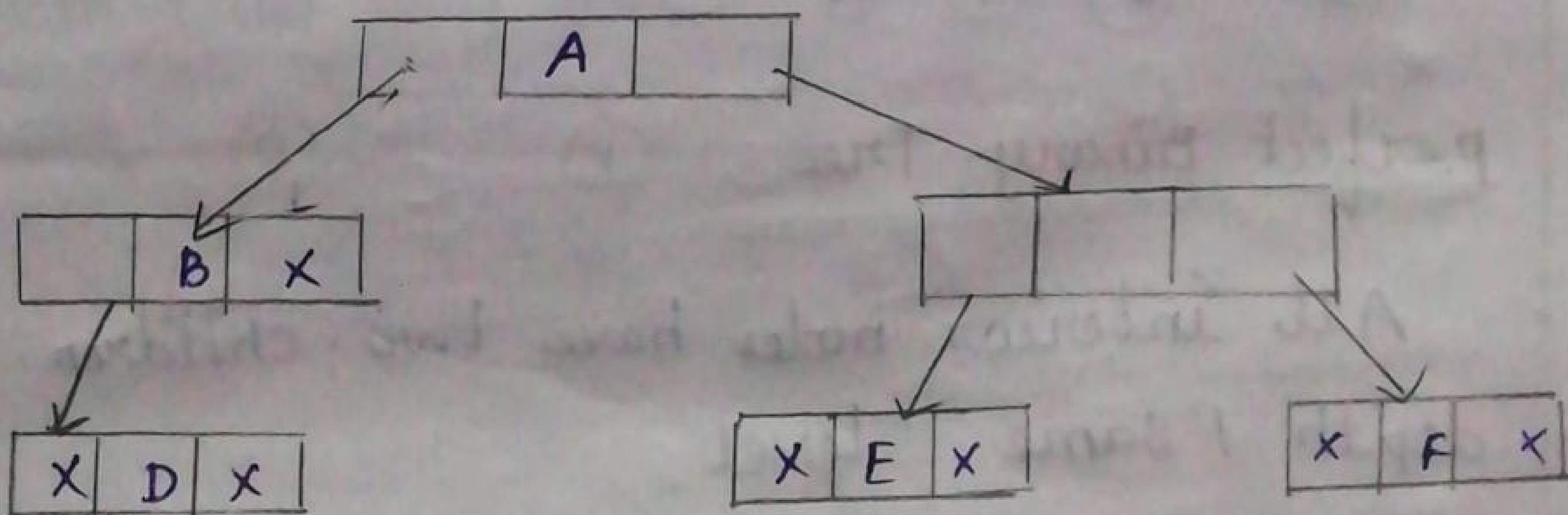
Binary Tree

* Left and Right

Binary Tree components

3 components

* Delete element
* painter to left sub tree
* painter to right sub tree



Types of Binary Trees

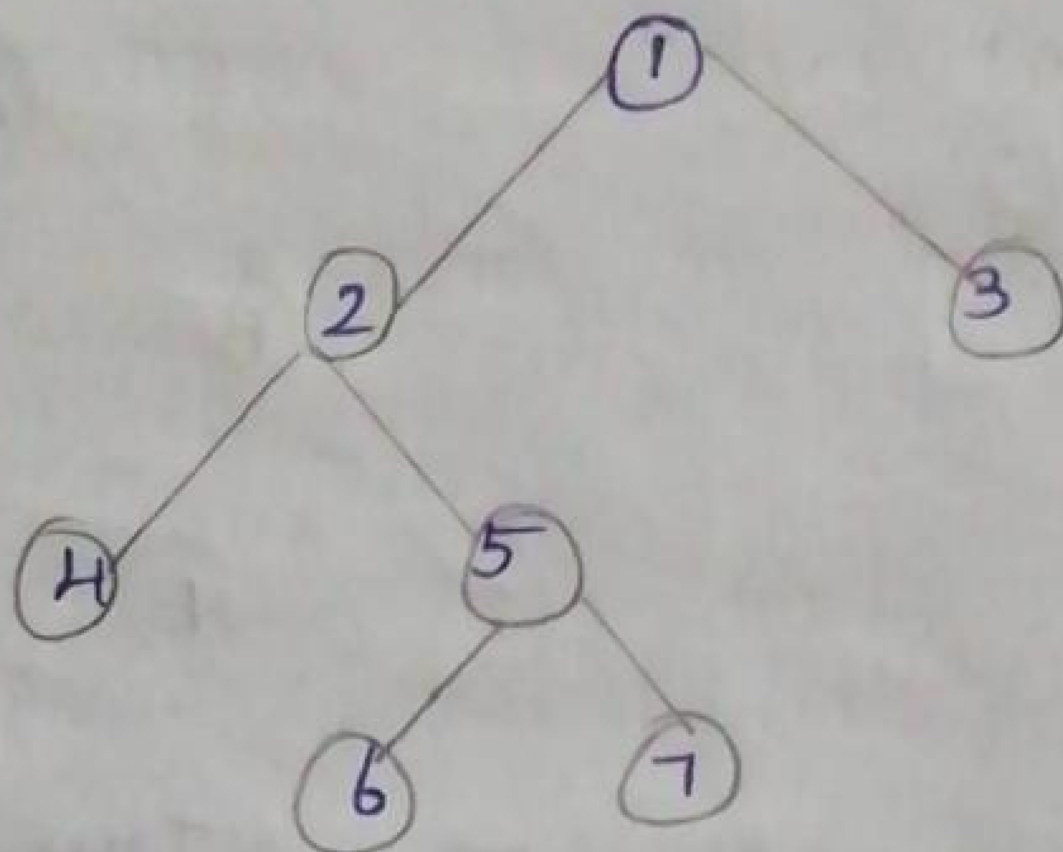
* Full Binary Tree
* Complete Binary Tree
* perfect Binary Tree
* Balanced Binary Tree
* Degenerated Binary Tree

Scanned by TapScanner

Full Binary Tree

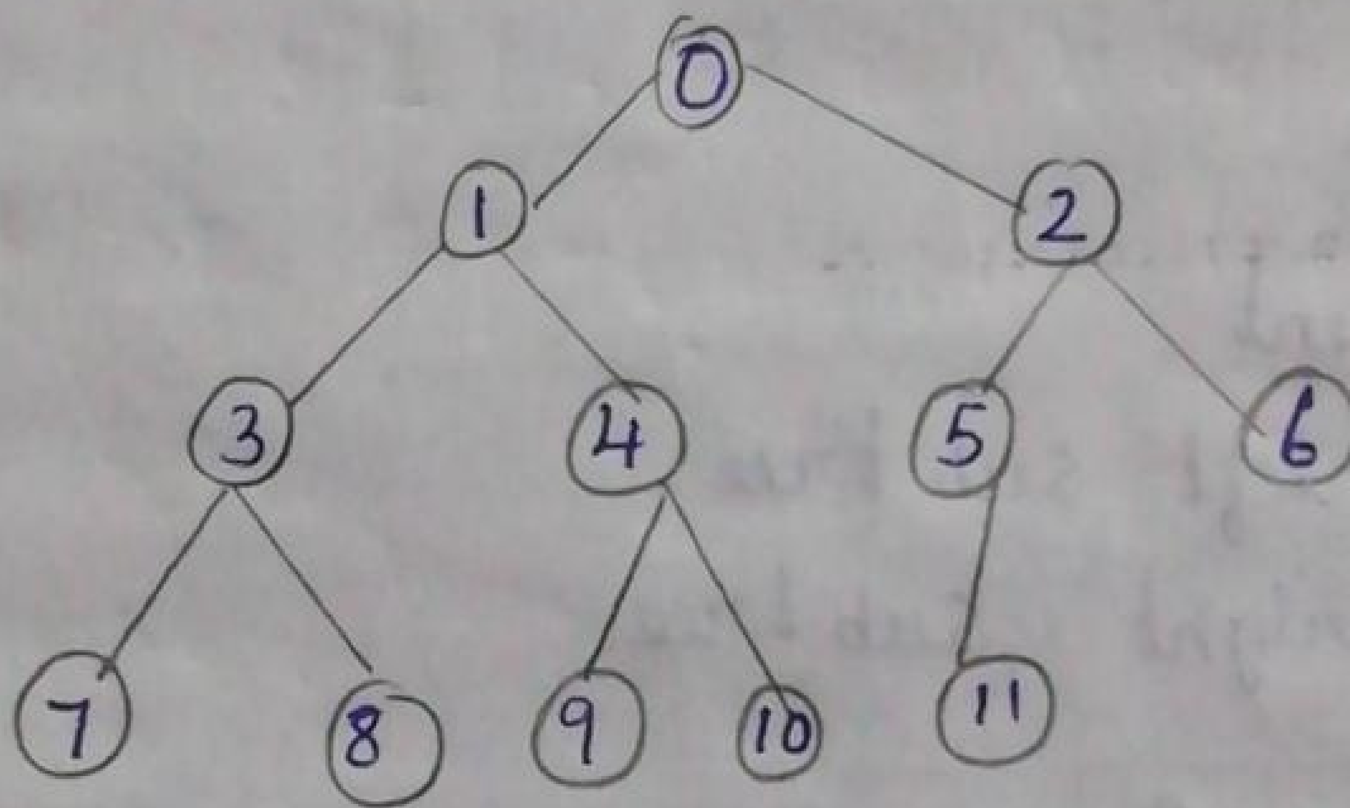
(Every parent node)

Internal node has either 2 / no child



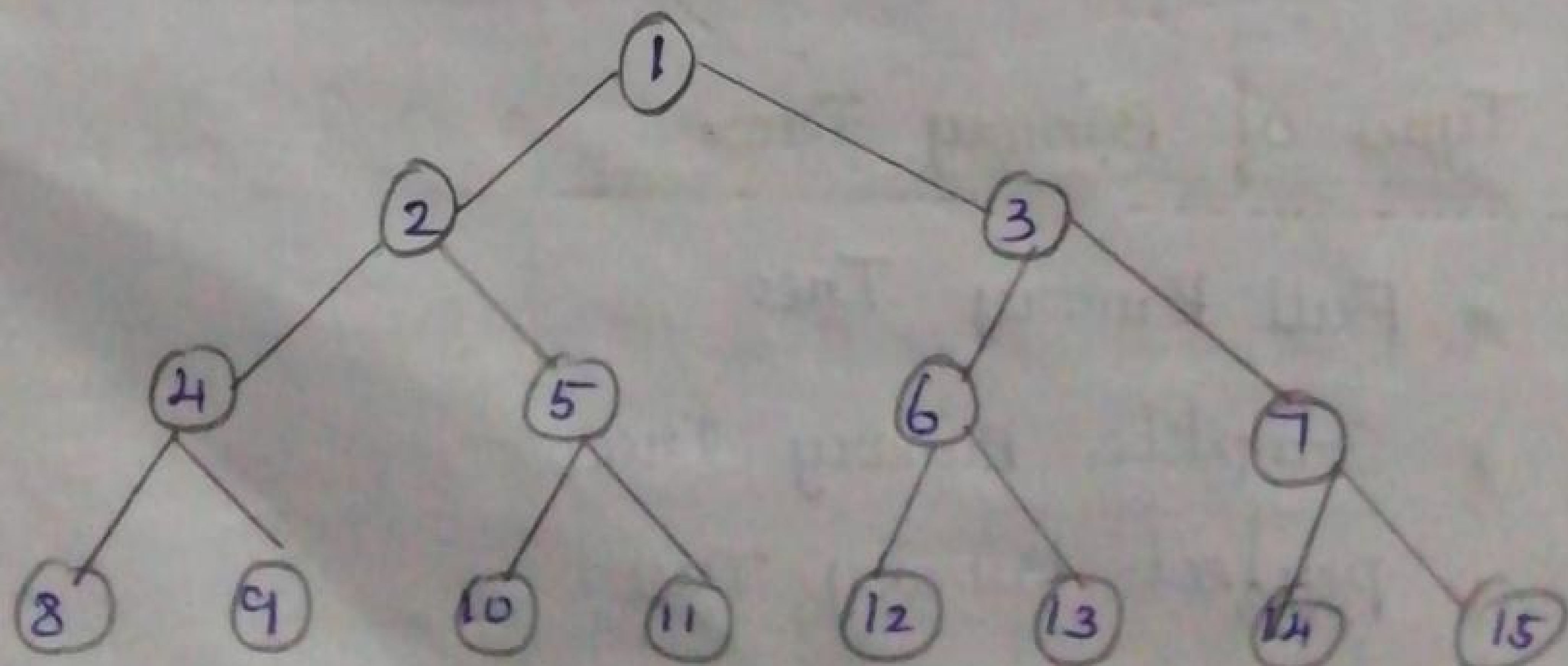
Complete Binary Tree

which all the levels are completely filled filled from the left.



perfect Binary Tree

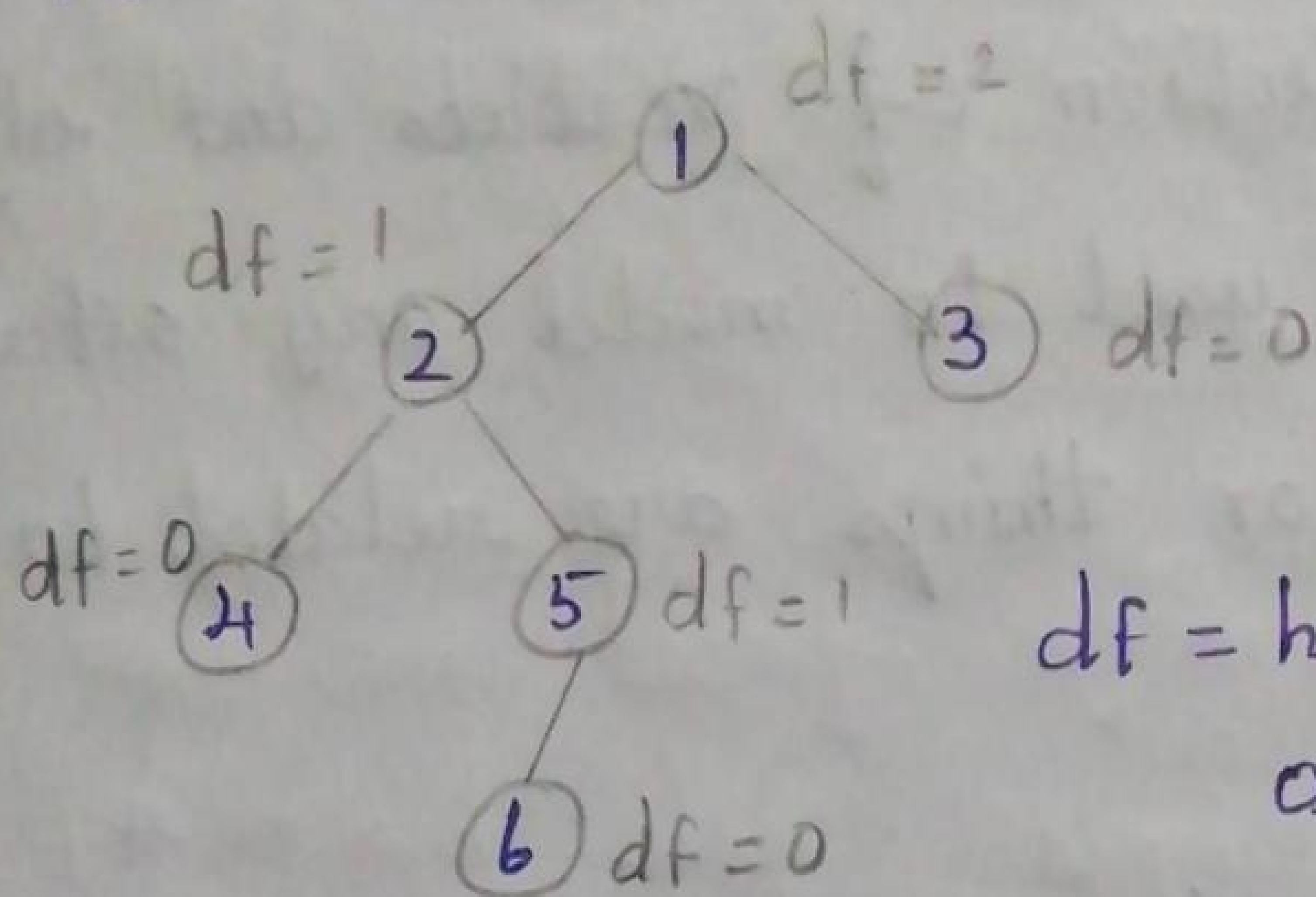
All interior nodes have two children, same depth / same level



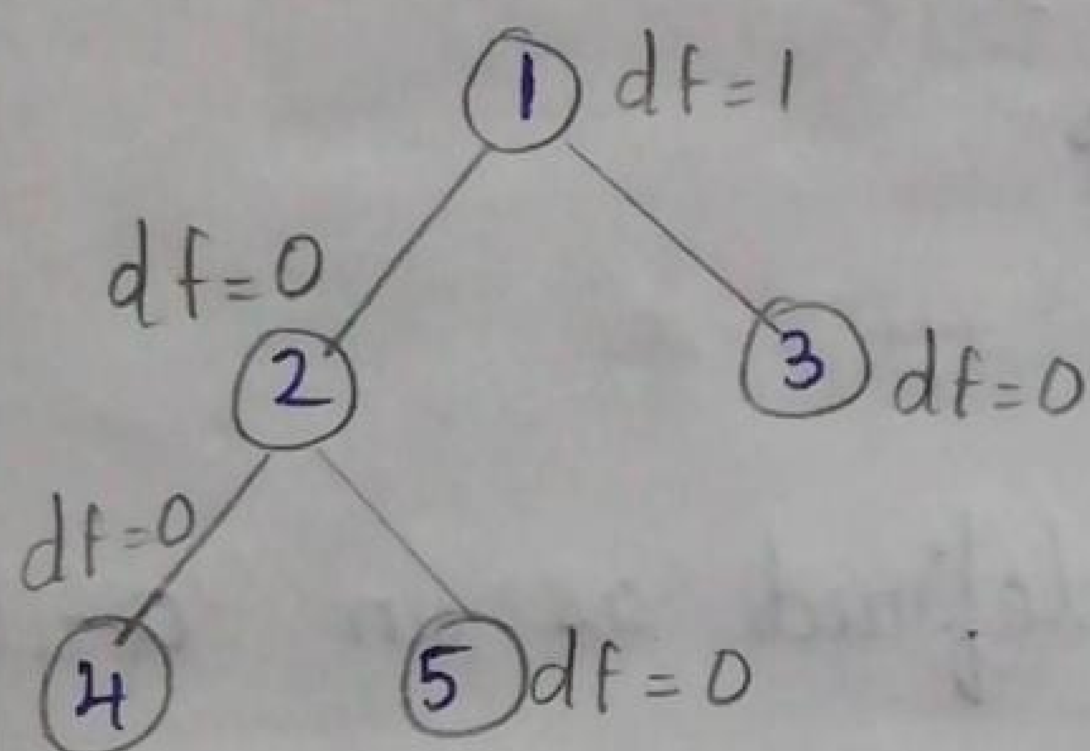
Scanned by TapScanner

Balanced Binary Tree

left and Right Sub trees - differ in height by no more than 1



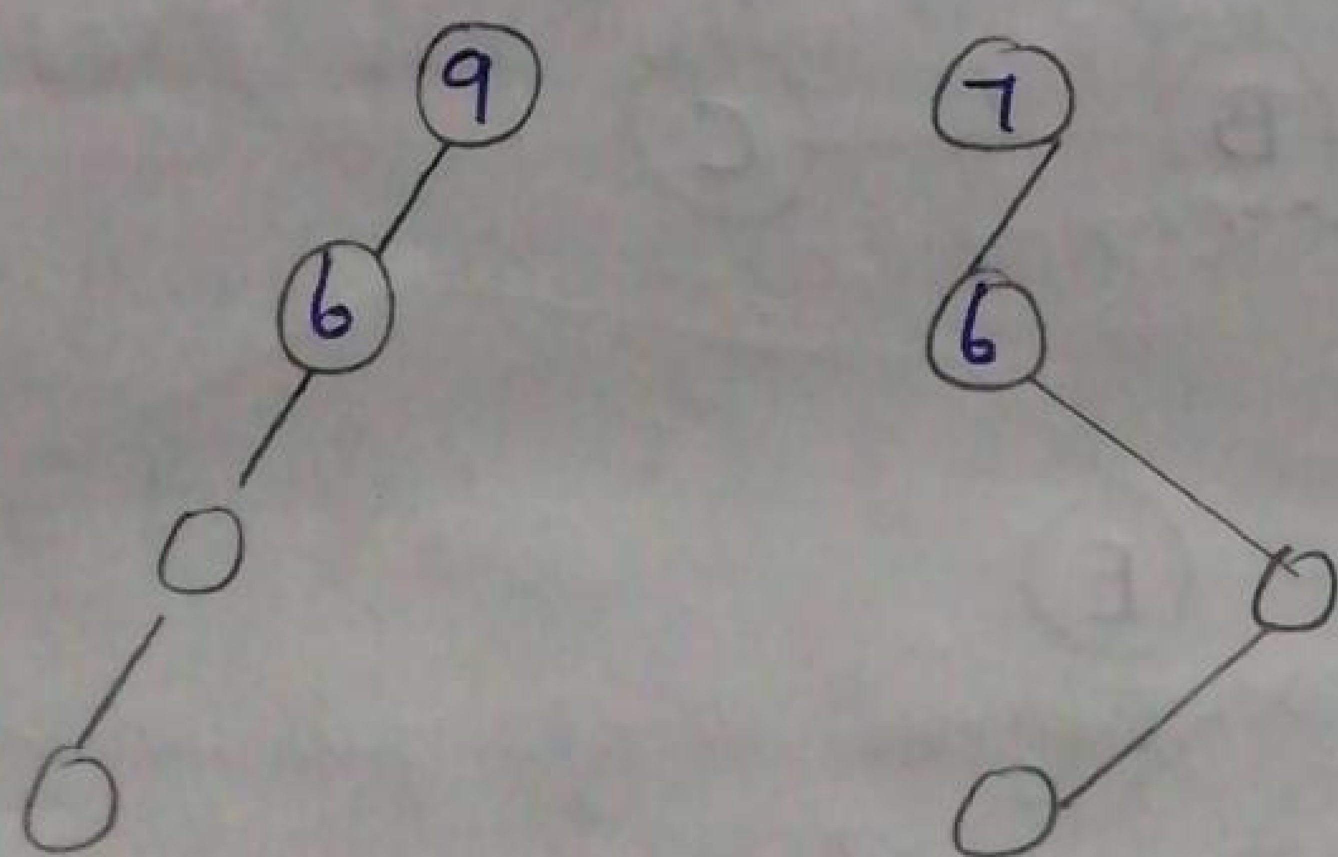
df = height of left child - height of right child.



Degenerate Binary Tree

Each parent node, there is only one associated child node linked list

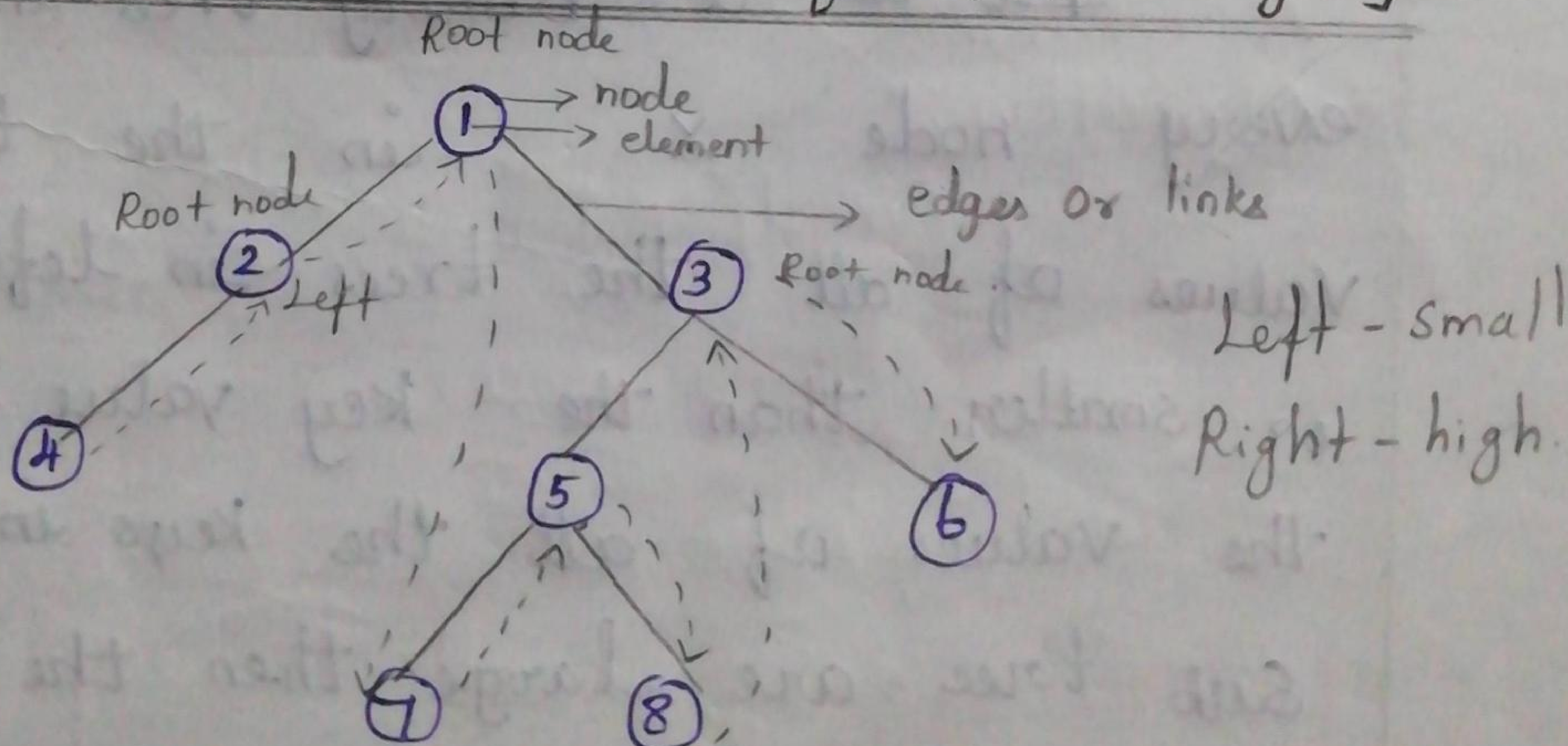
* Both trees only has one internal node



Tree Traversal : (Operation on Binary Tree)

* Visit all the nodes
* Connected Via edges (links)
* Three ways
    1. In Order Traversal
    2. pre order Traversal
    3. post order Traversal

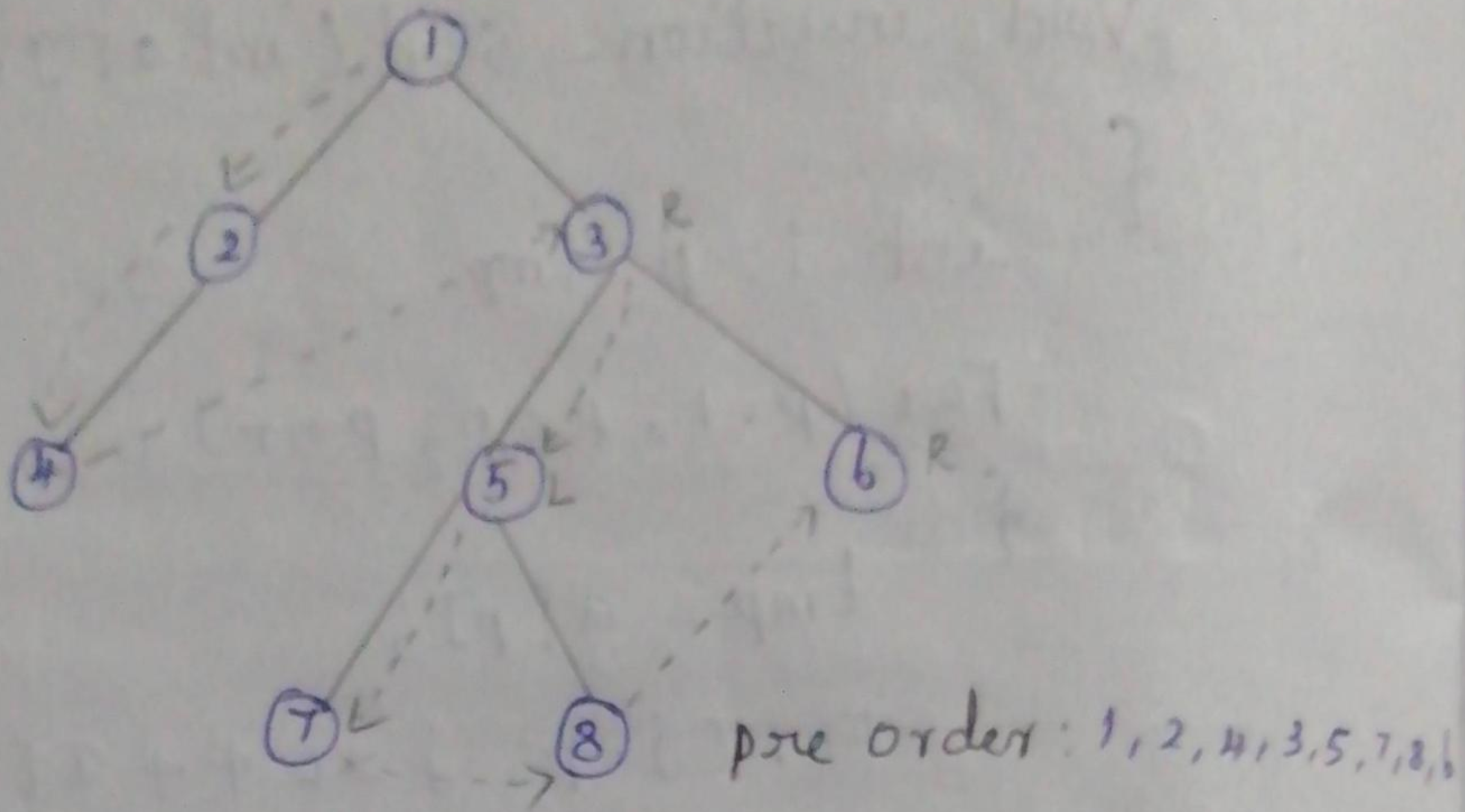
Inorder Traversal : [Left - Root - Right]



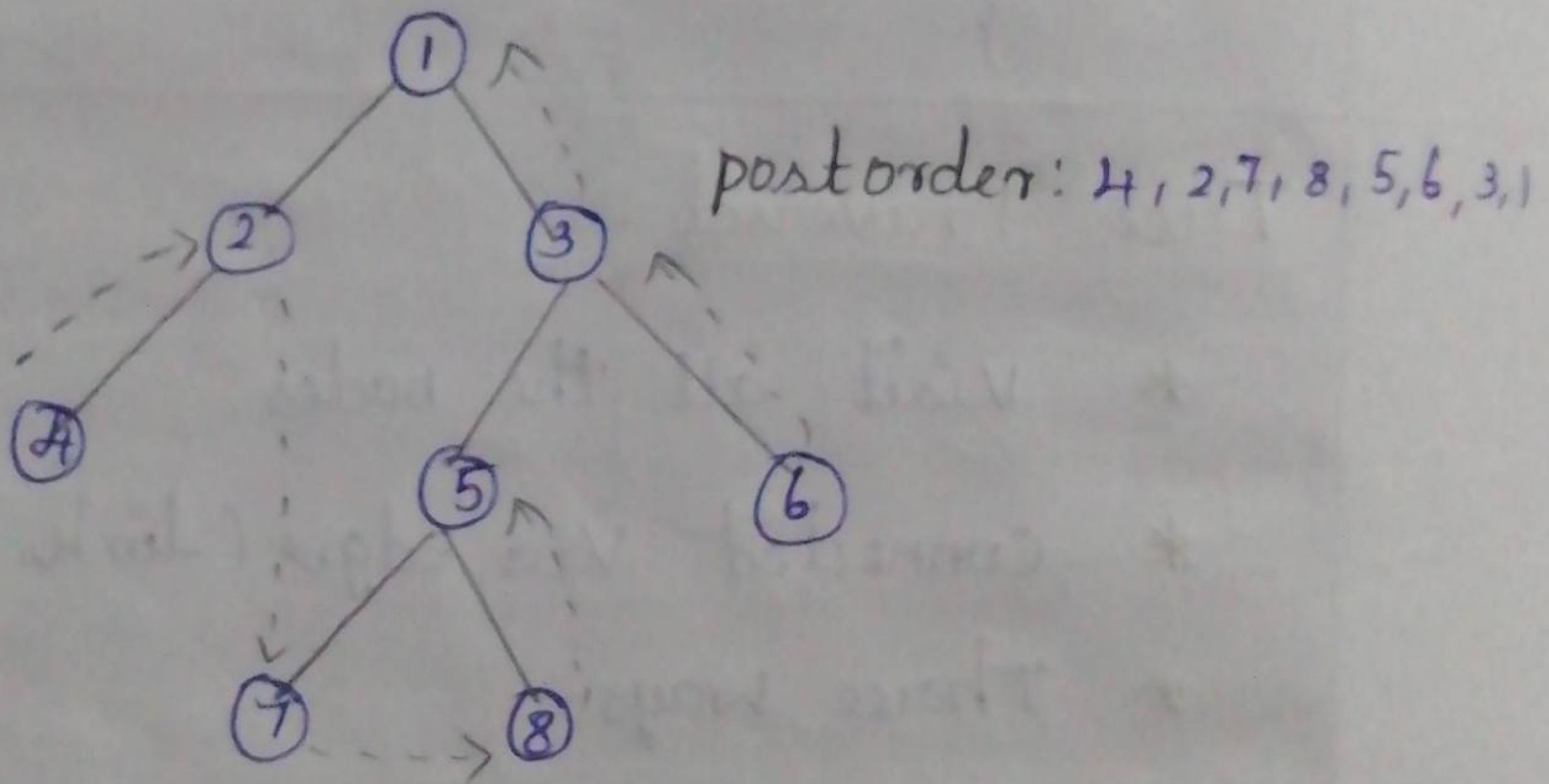
In Order : 4, 2, 1, 7, 5, 8, 3, 6

Scanned by TapScanner

pre - order Traversal : [Root - Left - Right]



post order Traversal : [Left - Right - Root]



Example : 10, 12, 5, 3, 4, 11, 2, 6, 7, 8 (Inorder, pre order, post order)

Scanned by TapScanner

Threaded Binary Tree

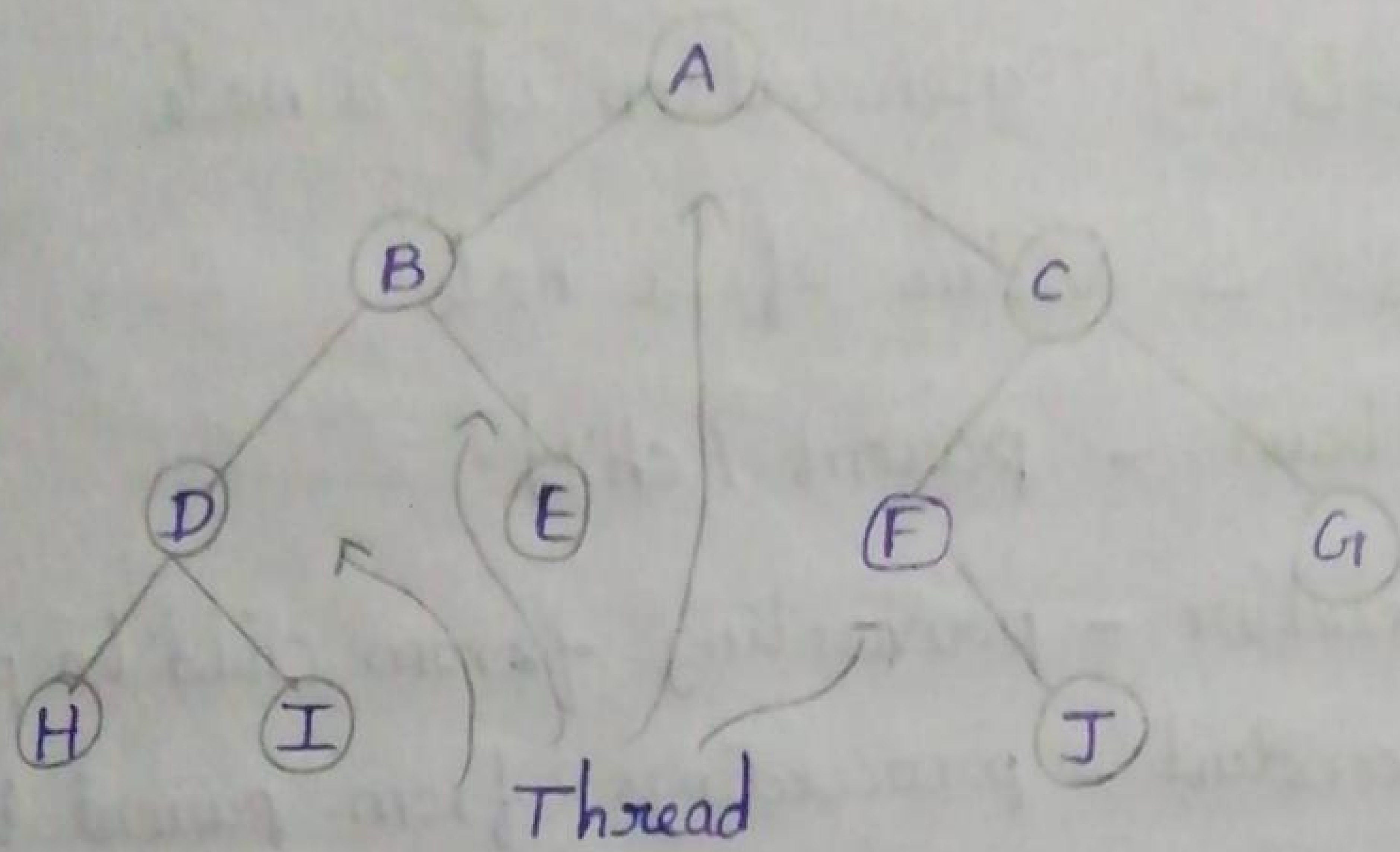
* It is also Binary Tree in which all left child ~~parents~~ pointer that are NULL (in LL representation) points to its in order predecessor, all right child ~~parents~~ pointer that are NULL (in LL rep) points to its in order successor.

* If there is no in order predecessor or in order successor, then it points to the root node.

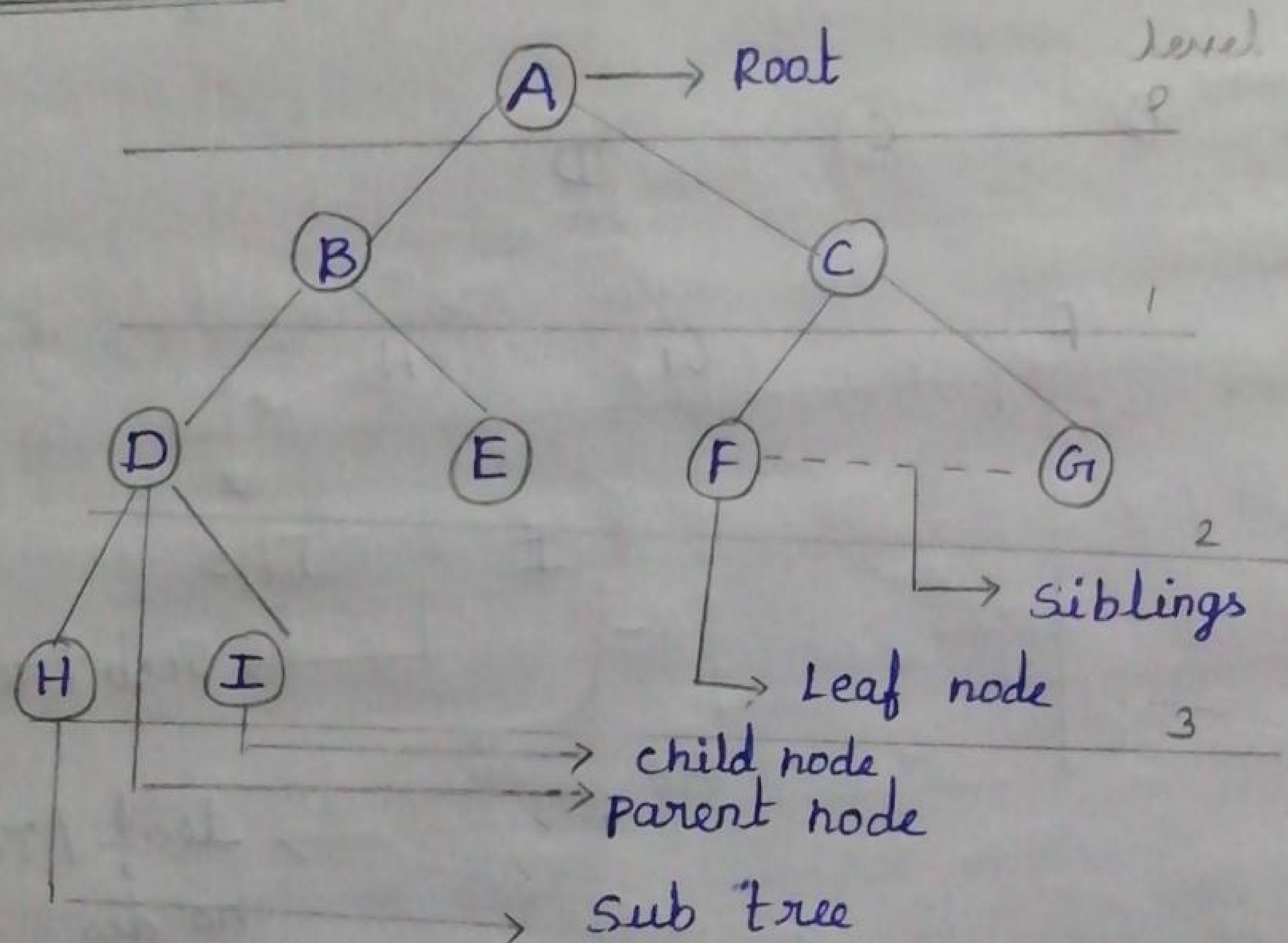
Example

H - D - I - B - E - A - F - J - C - G

Scanned by TapScanner



Trees



Important terms with respect to tree

1. path - along the edges
2. Root - only one root / tree, node @ the top
3. parent - edge upward
4. Child - edge downward
5. Leaf - does not have any child
6. Subtree - descendants of a node
7. Visiting - checking the value of a node
8. Traversing - passing through nodes

Scanned by TapScanner

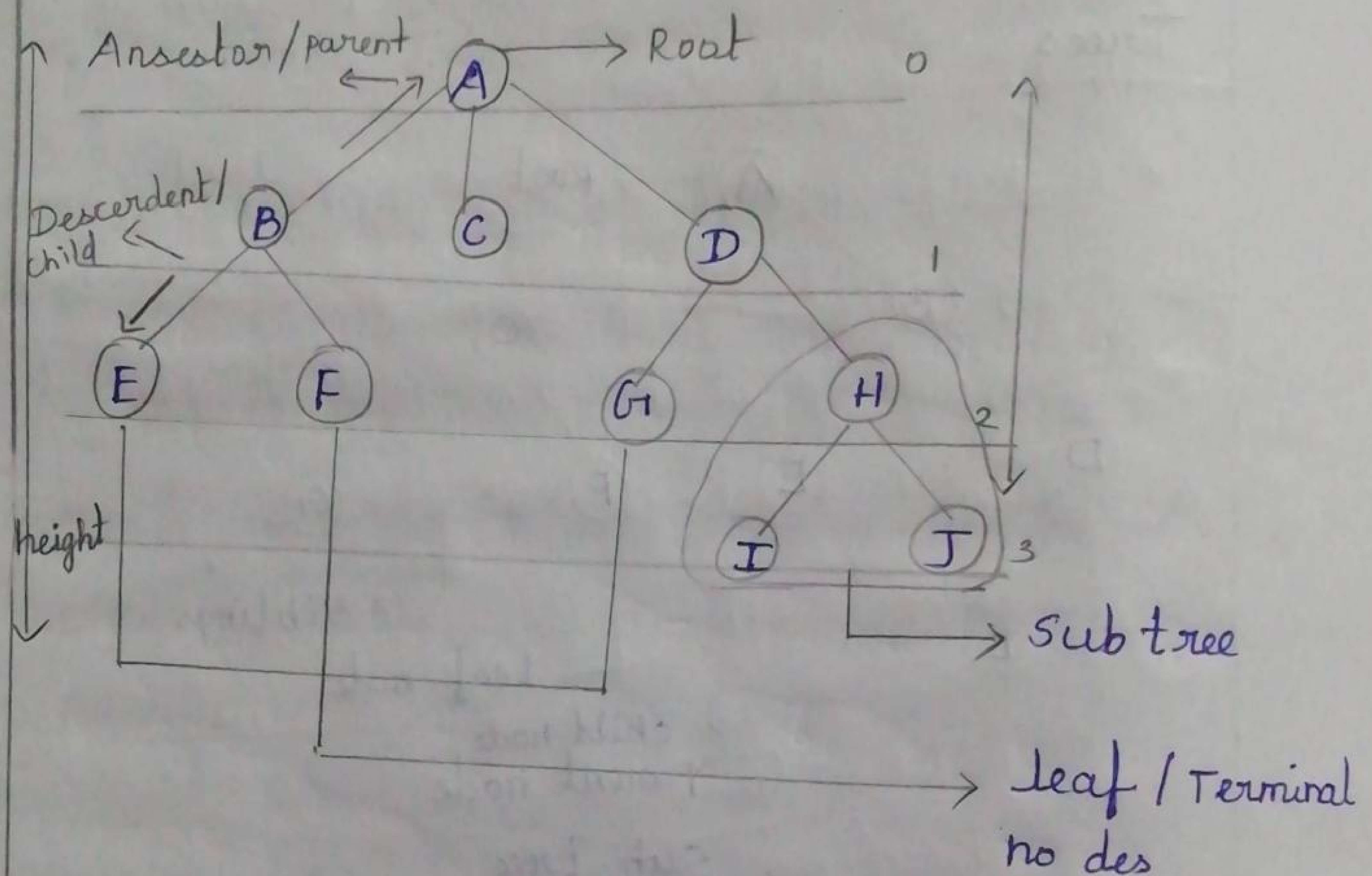
9. Levels - generation of a node

10. keys - Value of a node

11. Neighbour - parent / child

12. Ancestor - proceeding from child to parent

13. Descendent - proceedings from parent to child



14. Distance - Shortest path

15. width - No. of nodes in a level

16. Breedth - No. of leaves

17. Forest - Set of n ≥ o disjoint trees

18. ordered tree A rooted tree in which an ordering is specified for the children of each vertex

19. Size of a tree - No. of nodes in the tree

Scanned by TapScanner

Binary Search Tree (BST)

It is a Binary tree in which for every node x in the tree, the values of all the trees in left sub tree or smaller than the key value in x and the values of all the keys in it right sub tree are large then the key value in x.

Scanned by TapScanner

Operation perform in BST are:

1. Make empty
2. Insertion
3. deletion
4. Find
5. Find max
6. Find min.

Insertion:

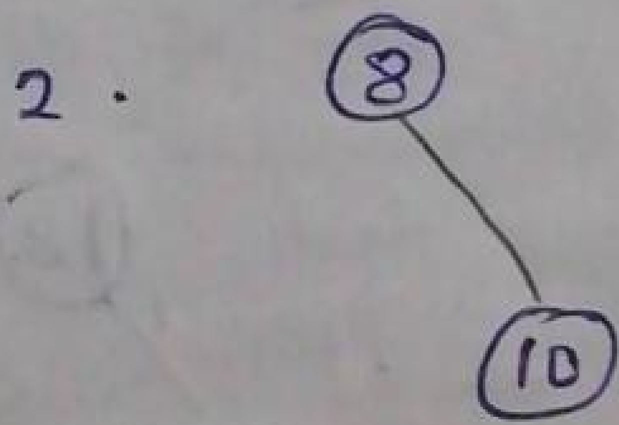
* To insert x into tree T, procede (proceed.) the Find Function

* It x is Found, do nothing otherwise insert x at the last spot on the path Travel. <u>Example:</u> 8, 10, 3, 2, 18, 6, 5, 13, 11

1.

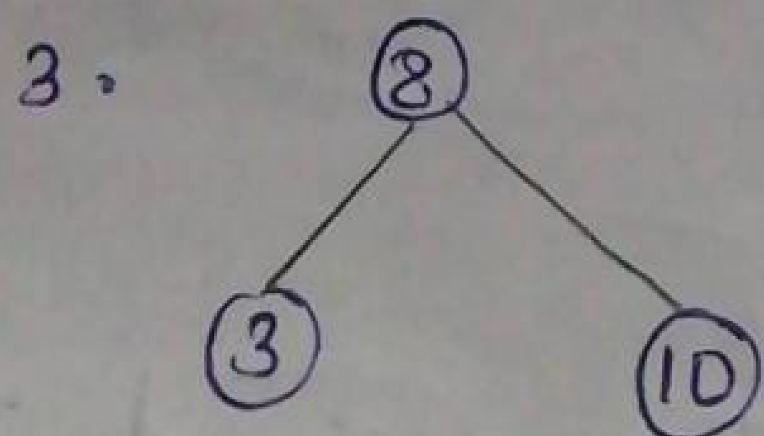
insert 8

2.



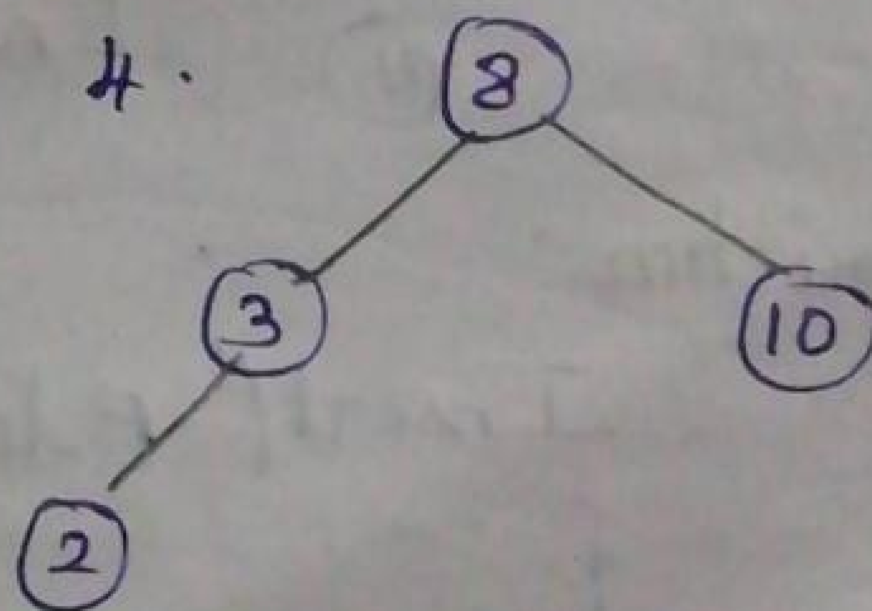
insert 10

3.



insert 3

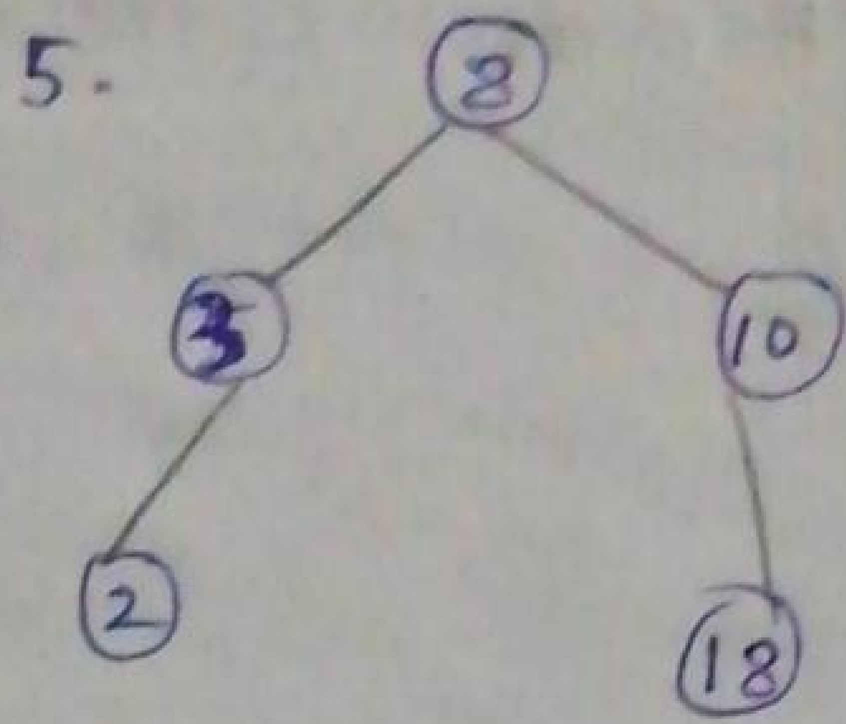
4.



insert 2

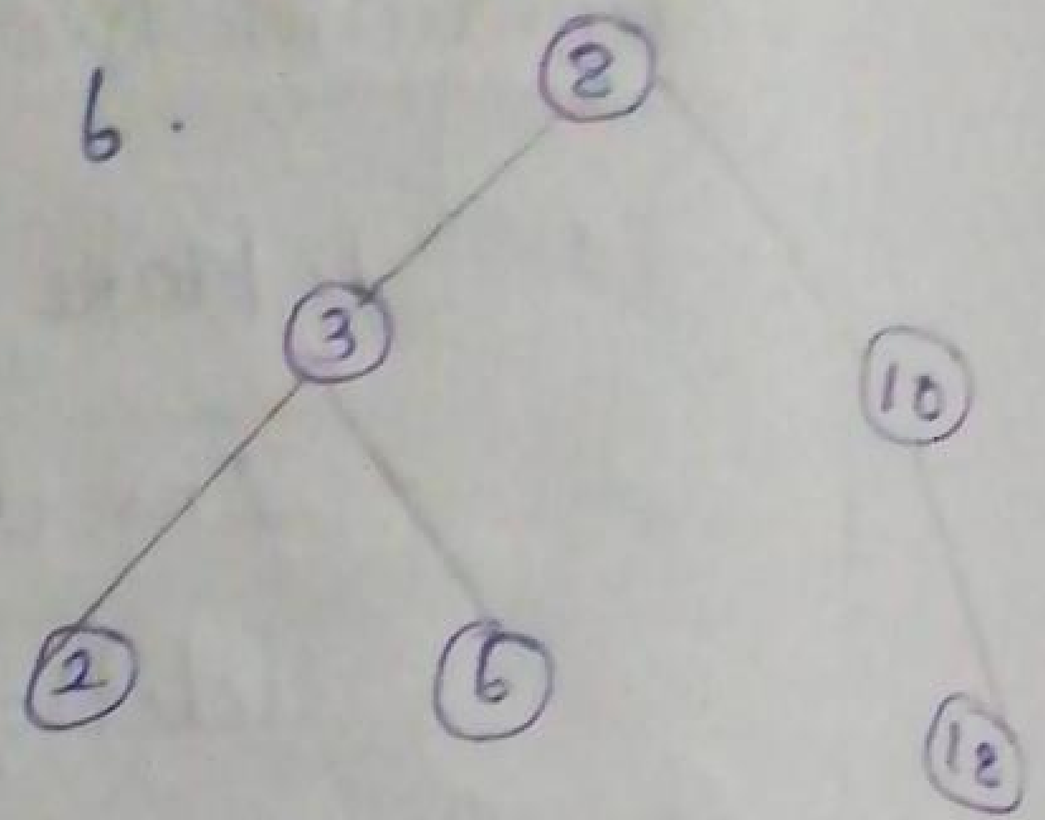
Scanned by TapScanner

5.



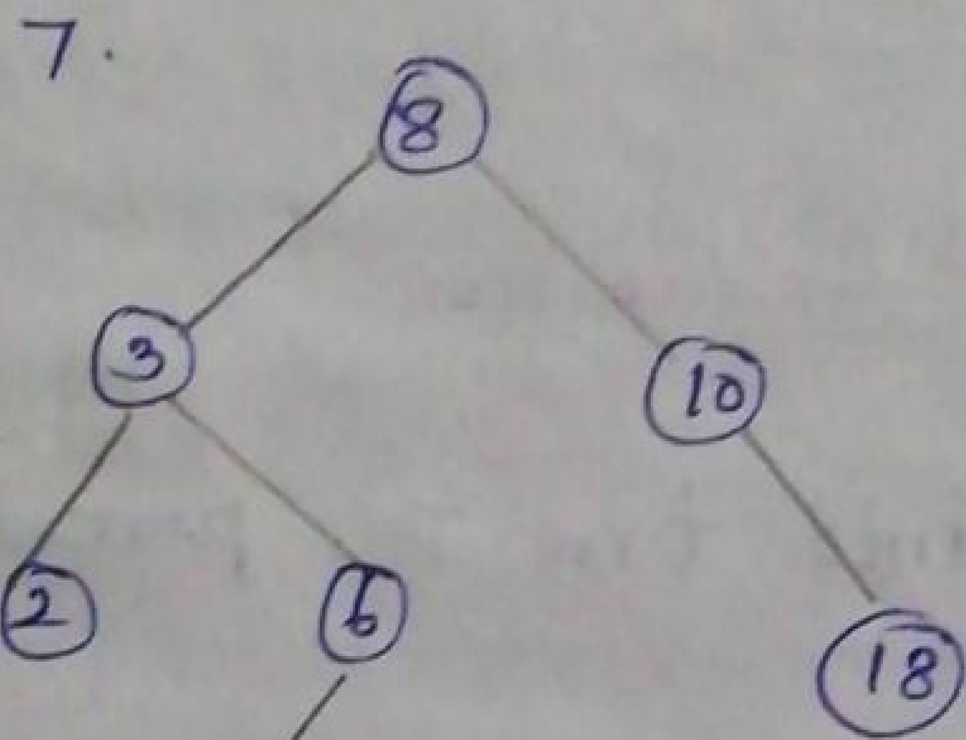
Insert 18

6.



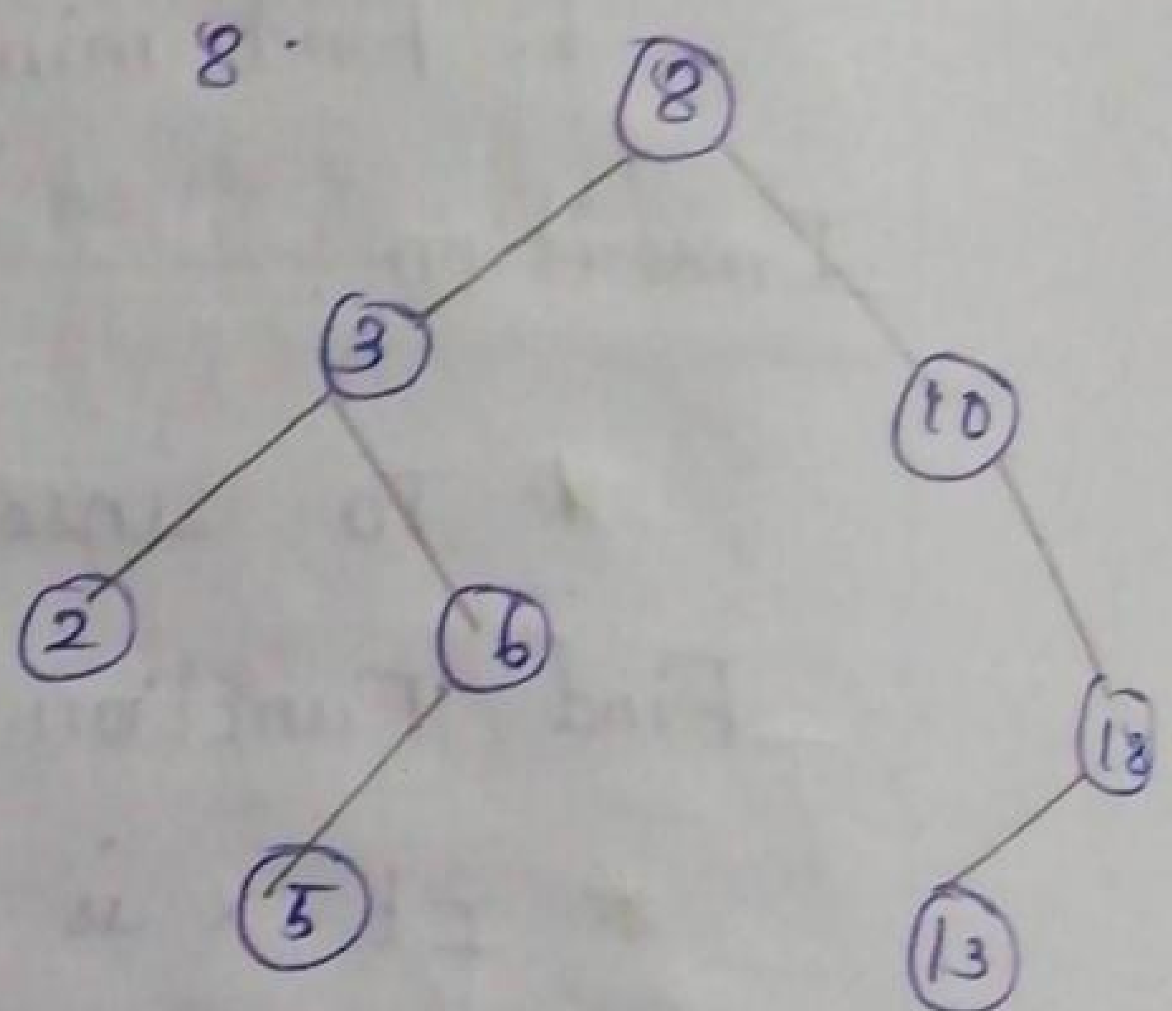
Insert 6

7.



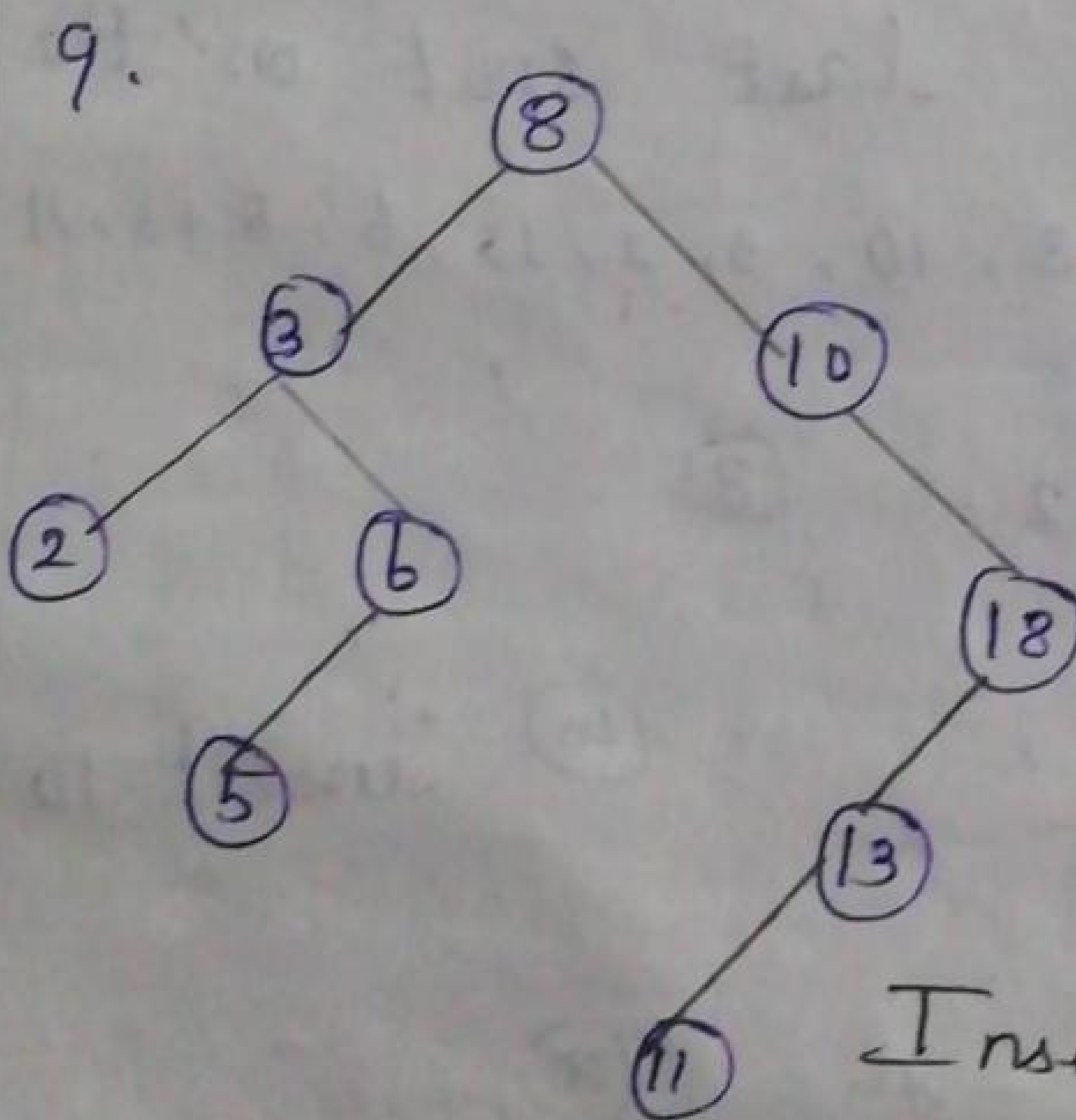
Insert 5

8.



Insert 13

9.



Insert 11

Algorithm:

```
Insert( Element Type x, Search Tree T)
{
 if (T == NULL)
 {
 T = malloc (size of (struct Tree node));
 if (T == NULL)
 Fatal Error (" Out of Space ))");
 else
 {
```

```
T⇒ left = NULL;
T → Right = NULL;
}
else
if (x > T→ Element)
T→ Right = Insert (x, T→ Right);
if (x < T→ Element
T→ left = Insert (x, T → left);
} return T;
```

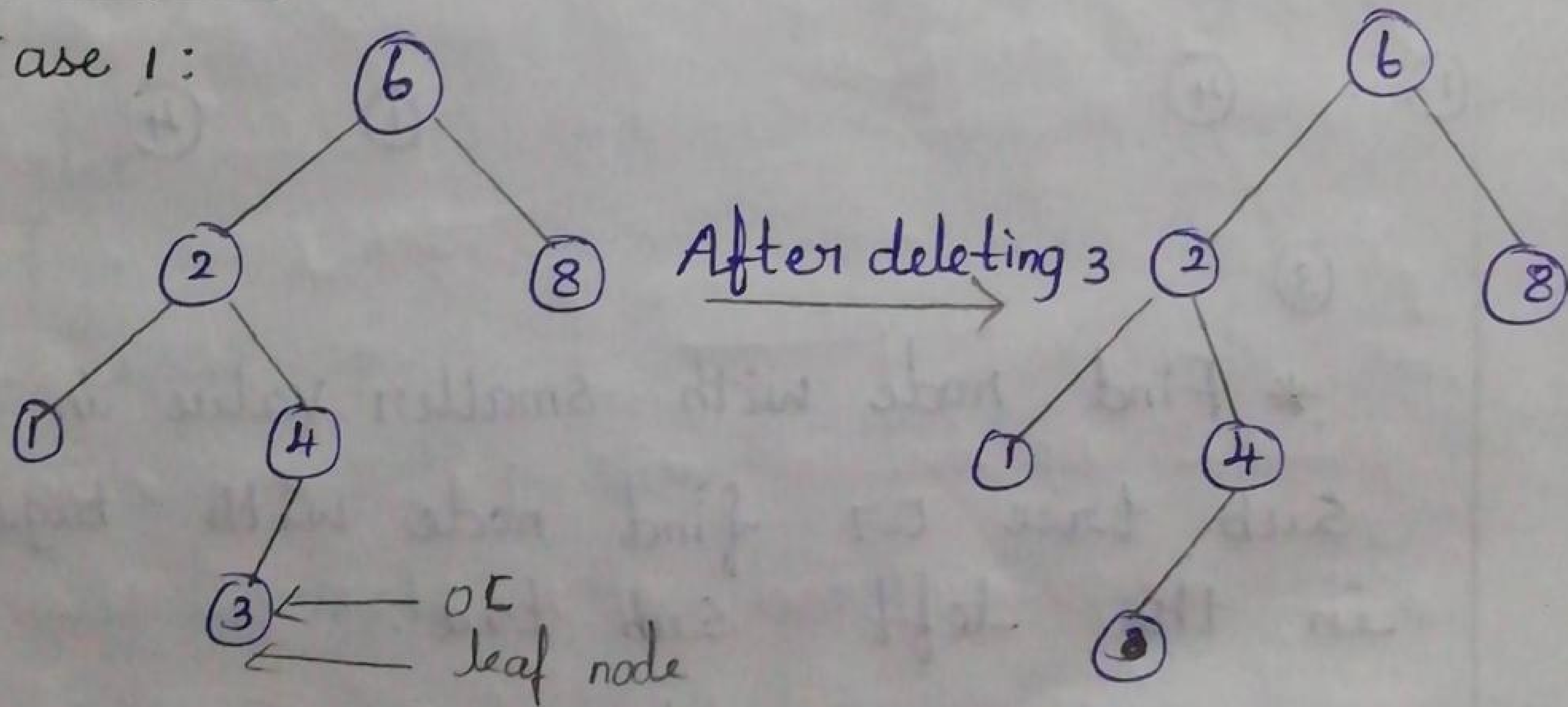
Deletion:

Case 1: Node with No children

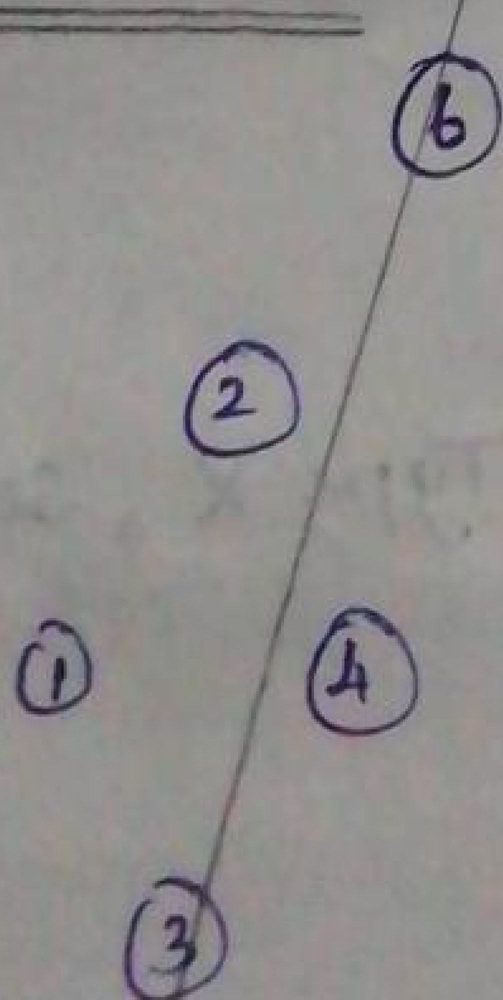
If the node is a leaf. It can be deleted immediately.

Example:

Case 1:



Case 2:

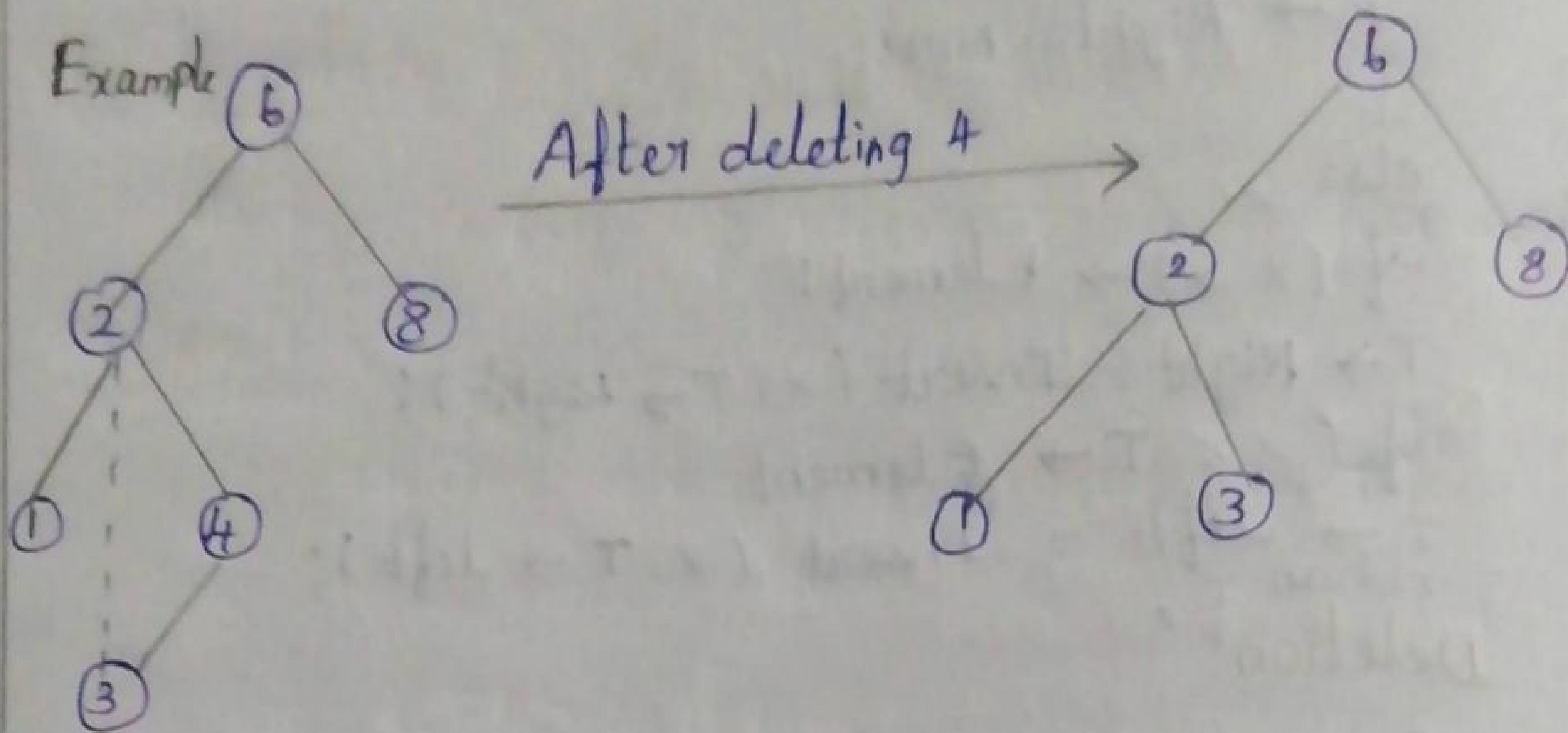


Case 2: node with one children.

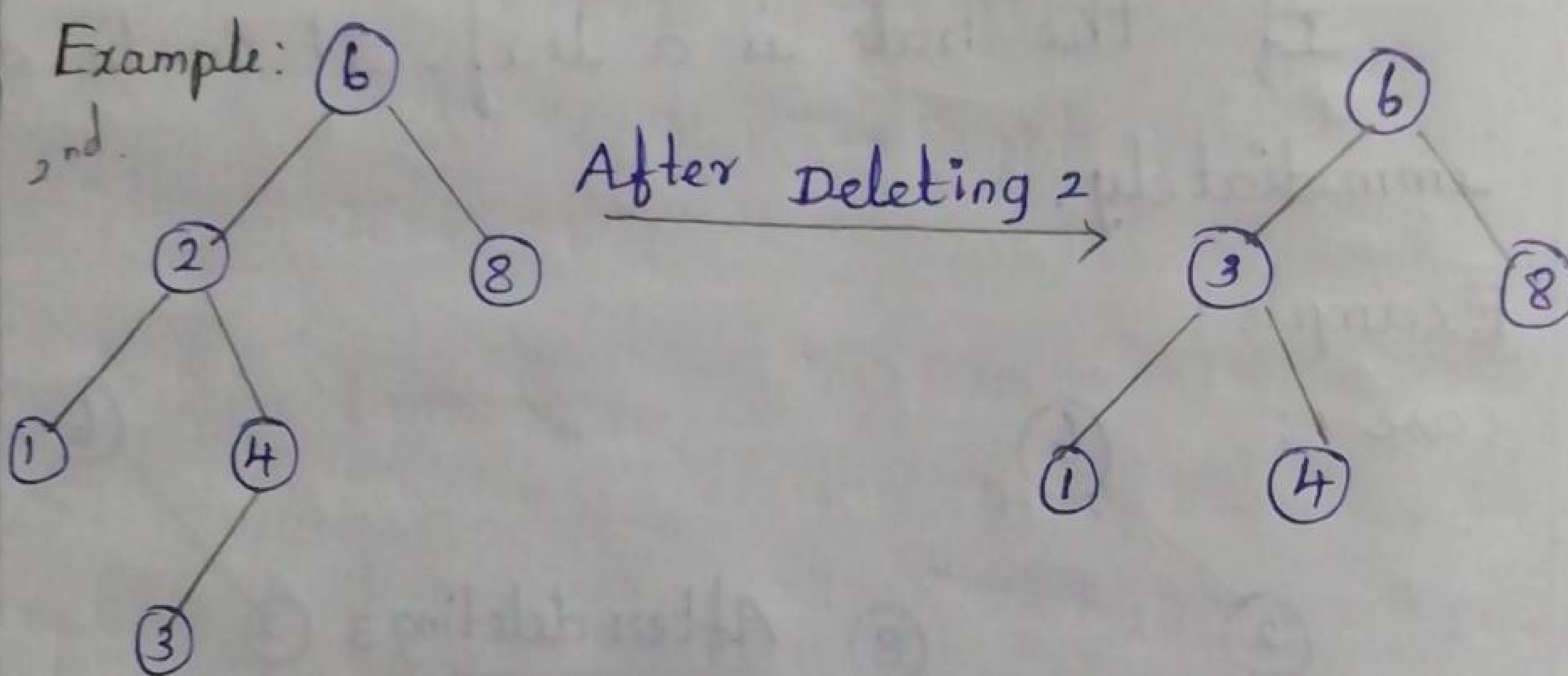
The node has one child, the node can be deleted after is parent, a pointed to by pass the node.

Scanned by TapScanner

Case 2: 1 children (Node with one children)



Case 3: 2 children (Node with 2 children)



1st * Find node with smaller value in the right sub tree or find node with biger value in the left sub tree.

* Replace the node in place of deleted node.

Algorithm:

```
Search Tree delete ( Element Type x , Search Tree T)
{
position Tmpcell;
if (T= = NULL)
Error ("Element not found");
else
```

Scanned by TapScanner

```
if ( x < T→ Element)
T → left = delete ( x , T → left)
else
if ( x > T → Element
T→ Right = Delete ( x , T → Right)
else
if (T→ left && T → Right
{
 Tmpcell = Fidmin
 ( T → Right);
 T → Element = Tmpcell → Element ;
 T → Right = Delete
 (T → Elements , T → Right);
}
else
{
 Tmpcell = T;
 if (T→ left == NULL)
 T = T→ Right;
 else
 if (T → Right == NULL)
 T = T → left ;
 Free ( Tmpcell);
}
return T;
```

The operation performed in BST

Make empty :

make empty operation is used for initialization

Algorithm

```
SearchTree makeEmpty (SearchTree T)
{
 if (T! = NULL)
 {
   makeEmpty (T→ Left);
   makeEmpty (T→ Right);
   Tree (T);
 }
 return NULL;
}
```

Find

It's operation will Find key x in the BST. If key is Found then ptr to that node is returned. If the key is not Found it will return NULL

Algorithm :

```
position Find (ElementType x, SearchTree T)
{
if (T== NULL)
 return NULL;
if (x < T → Element)
 return Find (x, T → left);
```

Scanned by TapScanner

```
else if ( x > T → Element )
return Find ( x , T → Right)
else
 return T;
}
```

Find min :

* It routine return the position of smallest element in the tree.

* To perform a Findmin, start at the root and go left as long as there is a left child. The stopping point in the smallest element

Algorithm :

```
position Findmin ( Search Tree T)
{
  if (T== NULL)
  return NULL;
  else if (T→ Left == NULL)
  return T;
  else
   return Findmin (T → left );
  }
```

Find Max :

* It routine return the position of largest element in the tree.

* To perform a Findmax, start at the root and go right as long as there is a right child. The stopping point is the largest element.

Scanned by TapScanner

Algorithm

```
position Findmax (SearchTree T)
{
  if (T== NULL)
  return NULL;
  else if
  (T→ right = = NULL)
  return T;
  else
  return Findmax (T→ Right);
}
```

Scanned by TapScanner