Q.1 Define Searching ?

Ans. Searching is an operation which finds the location of a given element in the list. The search is said to be successful or unsuccessful depending on whether the element that is to be searched is found or not. There are mainly two standard searching methods, which are commonly used

1. **Linear Search**
2. **Binary Search**

Q.2 Explain linear search?

Ans. This is the simplest method of searching. In this method, the element to be found is sequentially searched in the list. This method can be applied to a sorted or an unsorted list.

- Searching in case of a sorted list starts from 0^{th} element and continues until the element is found or an element whose value is greater (assuming the list is sorted in ascending order) than the value being searched is reached.
- Searching in case of unsorted list starts from the 0^{th} element and continues until the element is found or the end of the list is reached.

To understand this, consider the array shown in Fig. In which we are required to search a number 57 in an unsorted arrray.

11	2	9	13	57	25	17	1	90	3	**57**
11	**2**	9	13	57	25	17	1	90	3	**57**
11	2	**9**	13	57	25	17	1	90	3	**57**
11	2	9	**13**	57	25	17	1	90	3	**57**
11	2	9	13	**57**	25	17	1	90	3	**57**

Fig (1) : Linear search in an unsorted array

- The array shown in Fig consists of 10 numbers. Suppose the element that is to be searched is **57**. So **57** is compared with all the elements started with 0^{th} element and the searching process ends either when **57** is found or the lists ends.
- The performance of linear search algorithm can be measured by counting the comparisons done to find out an element. The number of comparisons is **O (n)**.

In case of sorted list, searching of element starts from the 0^{th} element. Searching ends when the element is found or any element of the list is found to be greater than the element to be searched. This is shown in Fig. (1).

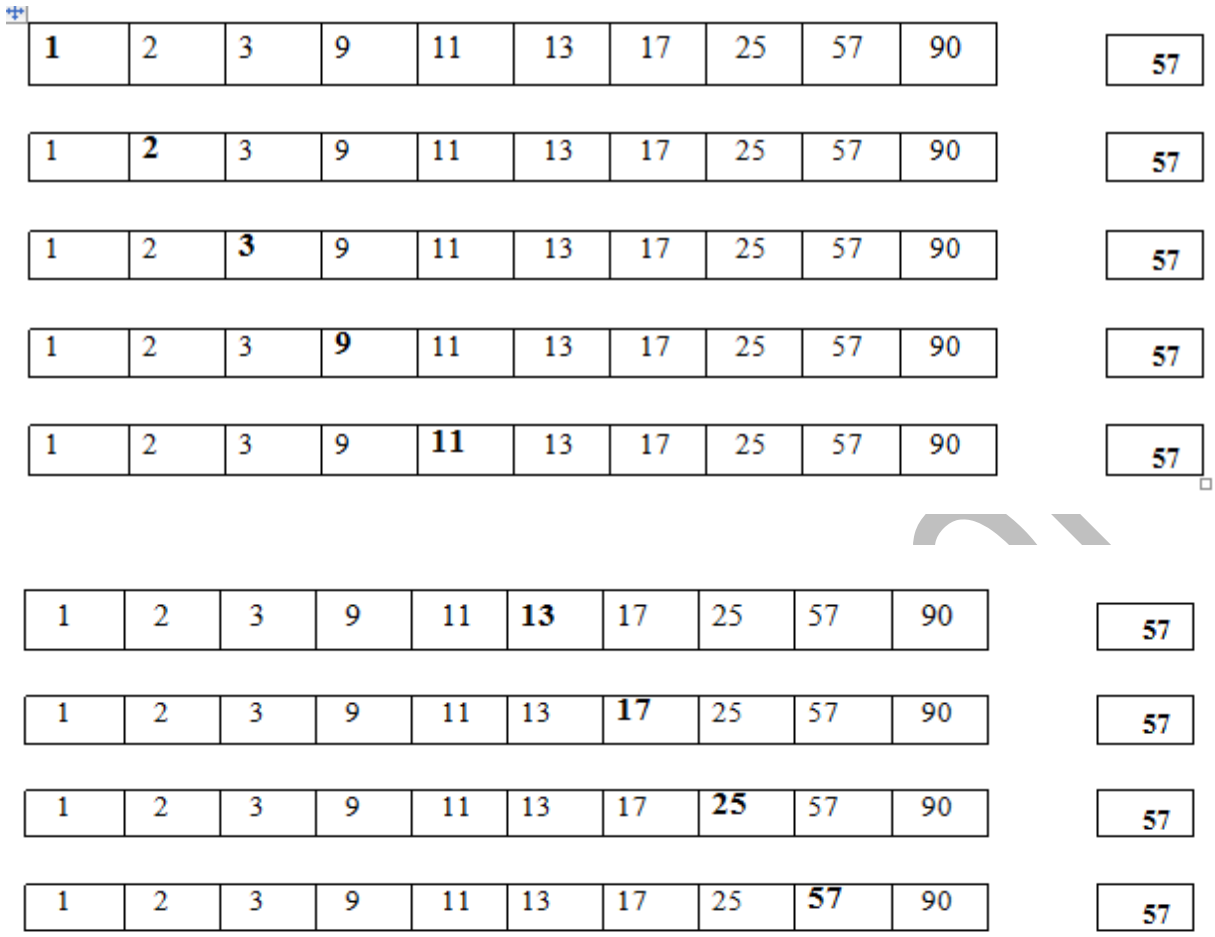


Fig (2) : Linear Search in a sorted array

Q.3 Write down the algorithm for linear search?

Ans. Let A[n] is an array of n elements and NUM is the number to be searched in array, I is the index number of each array element.

1. Set F = 0.
2. Read n elements of array A.
3. Read number NUM to be searched in Array A.
4. Repeat **Step** 4 for I = 0 to n :
   If NUM = A[I] then :
   Set F = 1
   BREAK;
   [End of If Structure]
   [End of **Step** 4 loop].
5. If F = 0 then :
   Write 'Number is not present in array'
   Else
   Write 'Number is present in array at location -',I+1
   [End of If Else Structure].
6. Exit

Q.4 Explain binary search?

Ans. **Binary search** method is very fast and efficient. This method requires that the list of elements be in sorted order.

- In this method, to search an element we compare it with the element present at the center of the list. If it matches then the search is successful. Otherwise, the list is divided into two halves: one from **0^{th}** element to the center element (first half), and another from center element to the last element (second half). As a result, all the elements in first half are smaller than the center element, whereas, all the elements in second half are greater than the center element.

- The searching will now proceed in either of the two halves depending upon whether the element is greater or smaller than the center element. If the element is smaller than the center element then the searching will be done in the first half, otherwise in the second half.

- Same process of comparing the required element with the center element and if not found then dividing the elements into two halves is repeated for the first half or second half. This procedure is repeated till the element is found or the division of half parts gives one element. Let us understand this with the help of Fig(3).

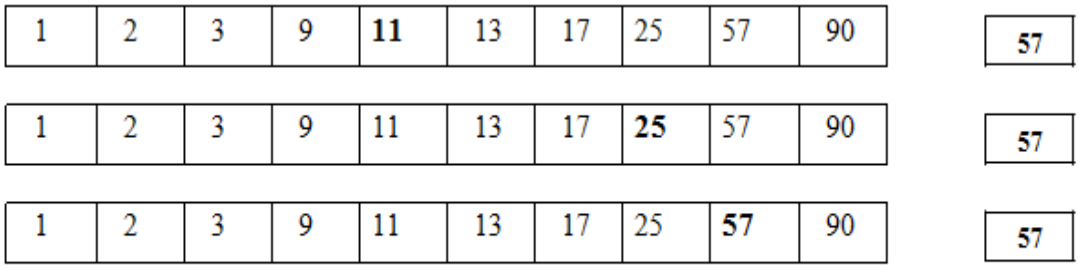


Fig. (3) : Binary Search

Suppose an array **arr** consists of **10** sorted numbers and **57** is element that is to be searched. The binary search method when applied to this array works as follows :

1. **57** is compared with the element present at the center of the list (i.e. **11**). Since **57** is greater than **11**, the searching is restricted only to the second half of the array.
2. Now **57** is compared with the center element of the second half of array (i.e. **25**). Here, again **57** is greater than **25** so the searching now proceed in the elements present between the **25** and the last element **90**.
3. This process is repeated till **57** is found or no further division of sub-array is possible.

The maximum number of comparisons in binary search is limited to **$\log_2 n$.**

Q.5 Write down the algorithm for binary search.

Ans. Let A[10] is an array of 10 elements and NUM is the number to be searched in array, L represents the lower bound, U represents the upper bound and M represents the index number of middle element, I represents the index number of each array element.

(1) Set F = 0, L = 0, U = 9.
(2) Read 10 elements of array A in ascending order.
(3) Read number NUM to be searched in Array A.
(4) Repeat **Step** 4 while L ≤ U
Set M = (L + U) / 2.
If NUM > A[M] : then
Set L = M +1
Else If NUM < A[M] : then
Set U = M – 1
Else
Set F = 1
BREAK.
[End of If Else If Structure]
[End of **Step** 4 loop].
5. If F = 0 : then
Write 'Number is not present in Array'.
Else
Write 'Number is present in Array at position–', M + 1.
[End of If Else Structure]
6. Exit

Q.6 Compare the linear search and binary search algorithm.

Ans. Consider the following set of elements : **1, 2, 3, 9, 11, 13, 17, 25, 57, 90**

Suppose, we want to search 25 in the above set of numbers. Table (1) shows number of comparisons required in both the methods.

Method	**Number of comparisons**
Linear Search	8
Binary Search	3

The table clearly shows that how fast a binary search algorithm works.

- The **advantage** of the binary search method is that, in each iteration, it reduces the number of elements to be searched from **n** to **n/2**. On the other hand, linear search method checks sequentially for every element, which makes it inefficient.

- The **disadvantage** of binary search is that it works only on sorted lists. So when searching is to be performed on unsorted list then linear search is the only option.

Q.7 Explain the term sorting?

Ans. **Sorting** means arranging a set of data in some order. There are different methods that are used to sort the data in ascending or descending order. These methods can be divided into two categories. They are as follows :

1. **External Sorting**
2. **Internal Sorting**

Q.8 Define the term external sorting?

Ans. When the data to be sorted is so large that some of the data is present in the memory and some is kept in auxiliary memory (hard disk, floopy, tape etc), then external sorting methods are used. External sorting is applied to the huge amount of data that cannot be accommodate in the memory all at a time. So data from the disk is loaded into memory part by part and each part that is loaded is sorted and the sorted data is stored into some intermediate file. Finally all the sorted parts present in different intermediate file. Finally all the sorted parts present in different intermediate files are merged into one single file.

Q.9 Define the term internal sorting?

Ans. If all the data that is to be sorted can be accommodated at a time in memory then internal sorting methods are used.

There are different types of internal sorting methods. The following methods sort the data in ascending order. With a minor change, we can also sort the data in descending order. Some standard methods are as given below :

1. **Selection Sort**
2. **Bubble Sort**
3. **Insertion Sort**
4. **Quick Sort**
5. **Merge Sort**
6. **Radix Sort**
7. **Heap Sort**

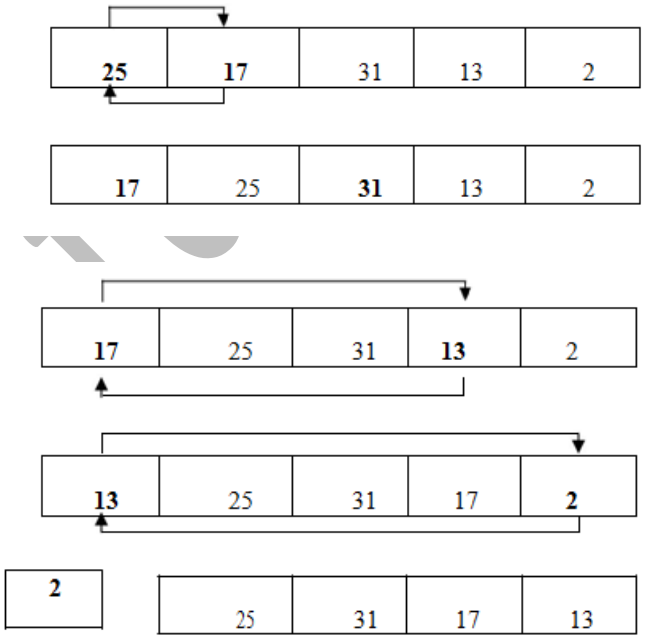
Q.10 Explain the concept of selection sort.

Ans. This is the simplest method of sorting. In this method, to sort the data in ascending order, the **0th** element is compared with all other elements. If the **0th** element is found to be greater than the compared element then they are interchanged. So after the first iteration, the smallest element is placed at 0th position. The same procedure is repeated for the 1st element and so on. This can be explained with the help of Fig.

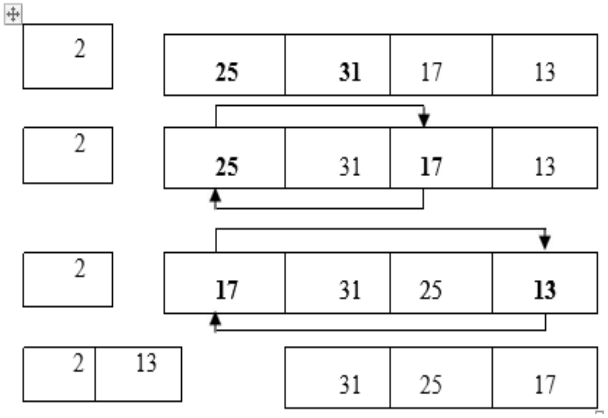
Suppose an array **arr** consists of 5 numbers. The selection sort algorithm work as follows :

- In the first iteration, the **0th** element 25 is compared with **1st** element 17 and since 25 is greater than 17, they are interchanged.
- Now the **0th** element 17 is compared with **2nd** element 31. But 17 being less than 31, hence they are not interchanged.
- This process is repeated till **0th** element is compared with rest of the elements. During the comparison if **0th** element is found to be greater than the compared element, then they are interchanged, otherwise not.
- At the end of the first iteration, the **0th** element holds the smallest number.
- Now the second iteration starts with the **1st** element 25. The above process of comparison and swapping is repeated.
- So if there are **n** elements, then after **(n – 1)** iterations, the array is sorted.

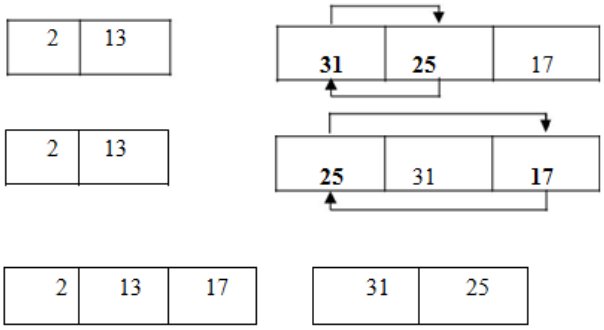
First Iteration



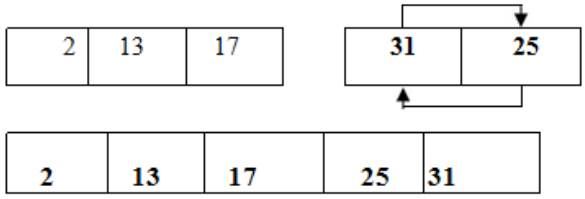
Second Iteration



Third Iteration



Fourth Iteration



Q.11 Write down the algorithm for selection sort.

Ans. Let A[5] is an array of 5 elements and TEMP is the variable used for swapping of array elements.

1. Read 5 elements of array A.
2. Write Original Array A.
3. Repeat **Step**s 3 to 4 for I = 0 to 3.
4. Repeat **Step** 4 for J = I+1 to 4.

IF A[I] > A[J] : then [Interchange A[I] & A[J]]
Set TEMP = A[J].
Set A[J] = A[I]

Set A[I] = TEMP
[End of IF structure].
[End of **Step** 4 loop].
[End of **Step** 3 loop].

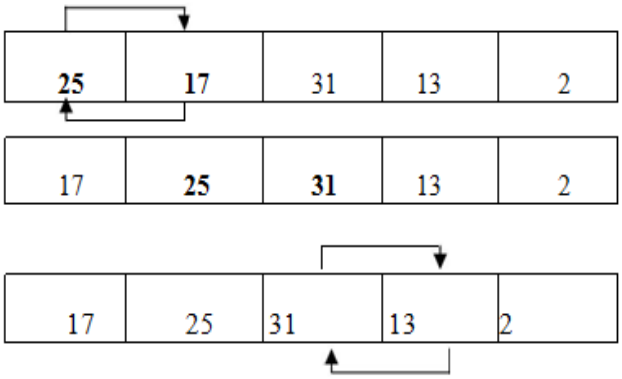
5. Write Sorted Array A.
6. Exit

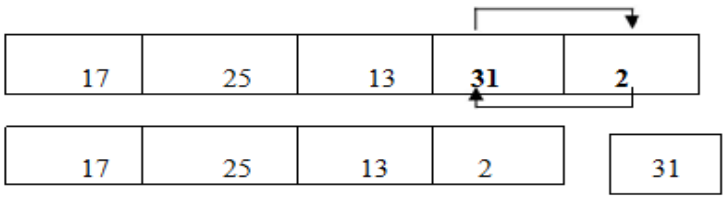
Q.12 Explain the concept of bubble sort.

Ans. In this method, to arrange elements in ascending order, to begin with the 0^{th} element is compared with the 1^{st} element.

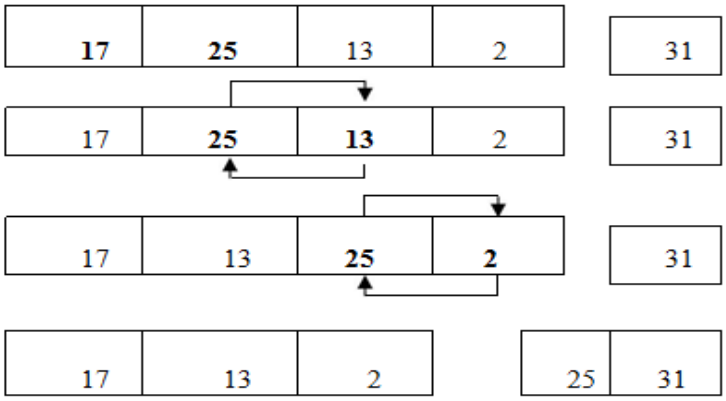
- If it is found to be greater than the 1^{st} element then they are interchanged. Then the 1^{st} element is compared with the 2^{nd} element, if it is found to be greater, then they are interchanged.
- In the same way all the elements(excluding list) are compared with their element and are interchanged if required.
- This is the first iteration and on completing this iteration the largest element gets placed at the last position. Similarly, in the second iteration the comparisons are made till the last but one element and this time the second largest element gets placed at the second last position in the list. As a result, after all the iterations the list becomes a sorted list.
- This can be explained with the help of Fig.

First Iteration

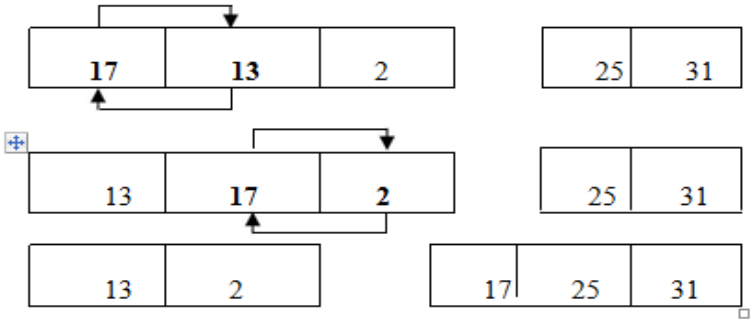




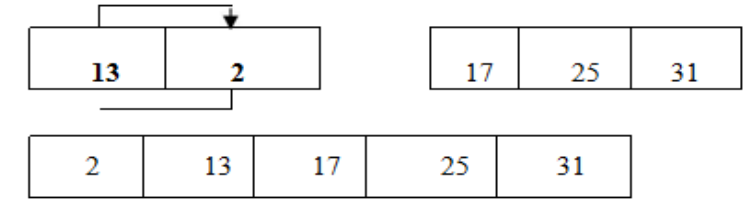
Second Iteration



Third Iteration



Fourth Iteration



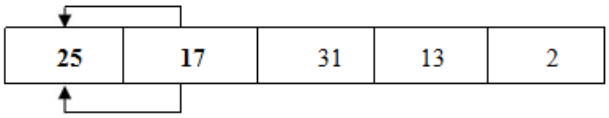
Q.13 Explain insertion sort?

Ans. Insertion sort is implemented by inserting a particular element at the appropriate position.

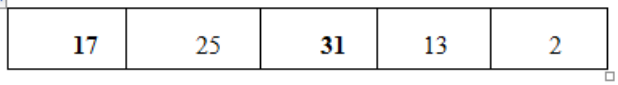
- In this method, the first iteration starts with comparison of 1^{st} eleme^{i}nt with the 0^{th} element. In the second iteration, 2^{nd} element is compared with the 0^{th} and 1^{st} element.
- In general, in every iteration an element is compared with all elements before it.

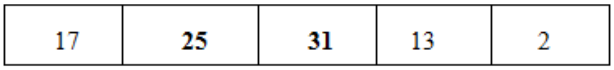
- During comparison, if is found that the element in question can be inserted at a suitable position then space is created for it by shifting the other element at the suitable position.
- This procedure is repeated for all the elements in the array. Let us understand this with the help of Fig.

First Iteration

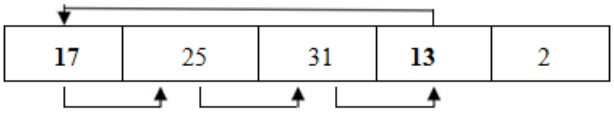


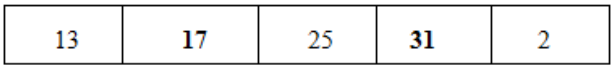
Second Iteration

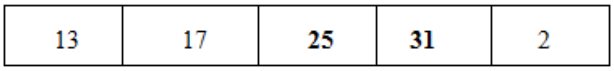




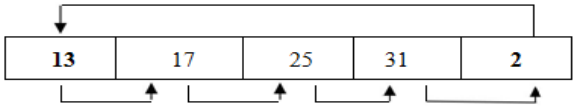
Third Iteration

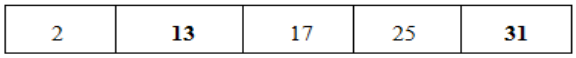


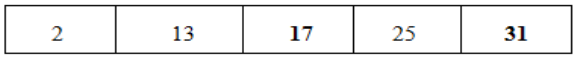


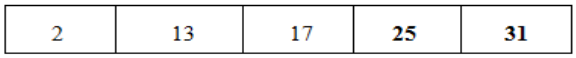


Fourth Iteration









Following **step**s explain the algorithm of insertion sort for an array **A** of **5** elements.

(a) In the first iteration, the **1st** element **17** is compared with the **0th** element **25**. Since **17** is smaller than **25**, **17** is inserted at **0th** place. The **0th** element **25** is shifted one position to the right.

(b) In the second iteration, the **2nd** element **31** and the **0th** element **17** are compared. Since **31** is greater than **17**, nothing is done. Then the **2nd** element **31** is compared with the **1st** element **25**. Again no action is taken as **25** is less than **31**.

(c) In the third iteration, the **3rd** element **13** is compared with the **0th** element **17**. Since, **13** is smaller than **17**, **13** is inserted at the **0th** place in the array and all the element from **0th** till **2nd** position are shifted to right by one position.

(d) In the fourth iteration, the **4th** element **2** is compared with the **0th** element **13**. Since, **2** is element smaller than **13**, the **4th** is inserted at the **0th** place in the array and all the are shifted right by elements from **0th** till **3rd** one position. As a result, the array now becomes a **sorted** Array.

Q.14 Write down the algorithm for insertion sort?

Ans. Let A [5] is an array of 5 elements and TEMP is the variable used for storing the number to be inserted at a particular position.

1. Read 5 elements of array A.
2. Write Original Array A.
3. Repeat **Step**s 3 to 5 for I = 1 to 4.
4. Repeat **Step** 4 to 5 for J = 0 to **I** – 1. IF A[J] > A[I] : then
   Set TEMP = A[I]
5. Repeat **Step** 5 for K = **I** to J + 1
   Set A[K] = A[K-1]

   [End of **Step** 5 loop].

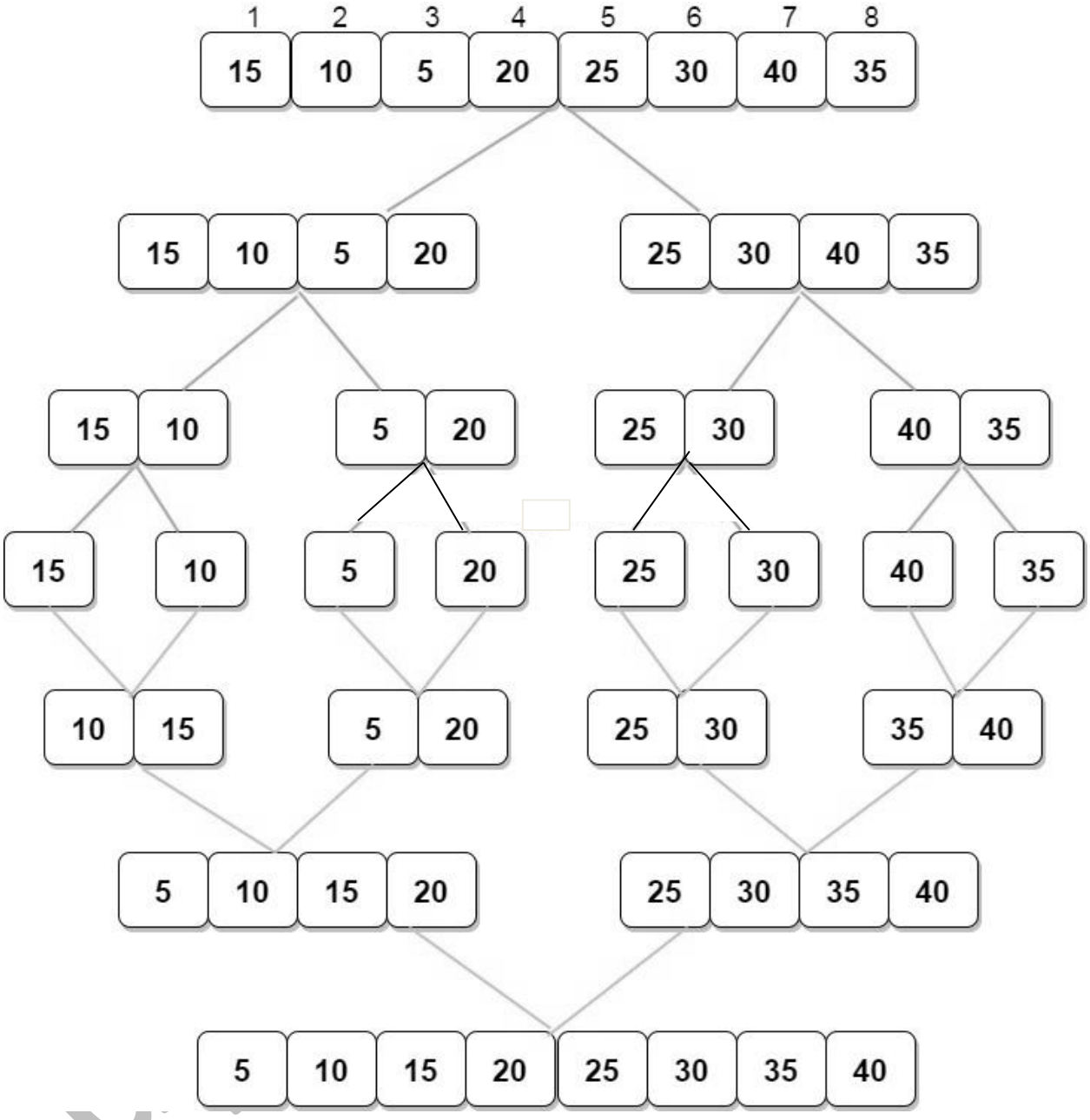
   [End of **Step** 4 IF structure].

   [End of **Step** 4 loop].

   [End of **Step** 3 loop].
6. Write Sorted Array A.
7. Exit

Q.15 Sort the given array with the help of merge sort (15,10,5,20,25,30,4,35) ?

Ans.



Q.16 Explain the concept of merge sort.

Ans. Merge sort is a sorting technique based on divide and conquer technique.

- With worst-case time complexity being O(n log n), it is one of the most respected algorithms.
- Merge sort first divides the array into equal halves and then combines them in a sorted manner

Q.17 Explain the concept of heap sort?

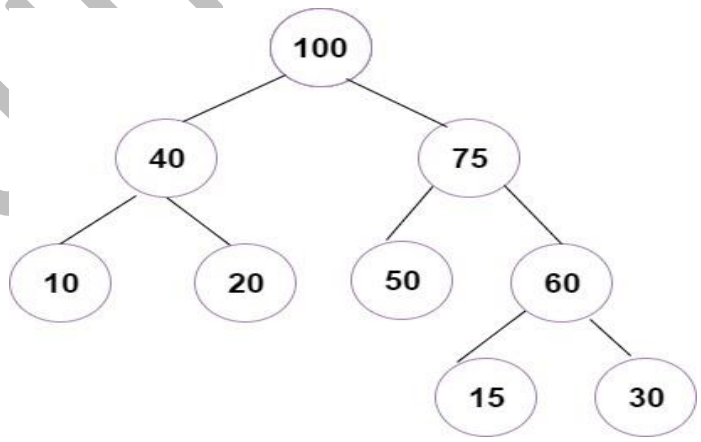
Ans. Heap sort is a comparison based sorting algorithm.

- It is a special tree-based data structure.
- Heap sort is similar to selection sort. The only difference is, it finds largest element and places the it at the end.
- This sort is not a stable sort. It requires a constant space for sorting a list.
- It is very fast and widely used for sorting.

The Time complexity of Heap Sort is O(nlogn).

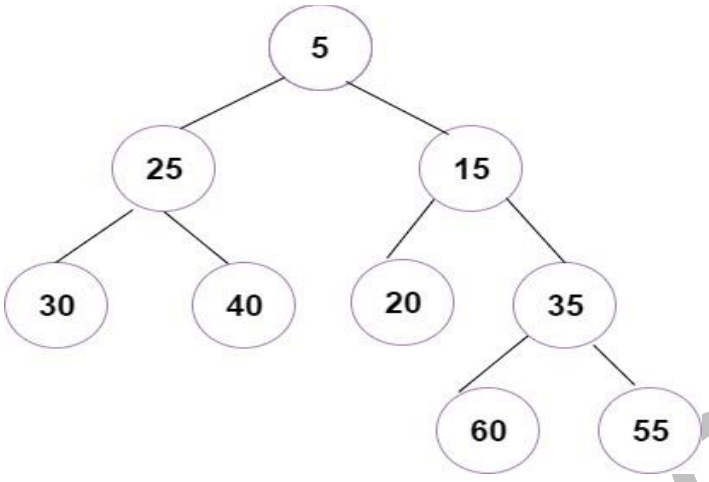
Max Heap

A Max heap is a complete binary tree 'H' with n elements, if each node 'N' of H is greater than equal to the children of 'N' is known as Max Heap. It is also known as Descending Heap.



Min Heap

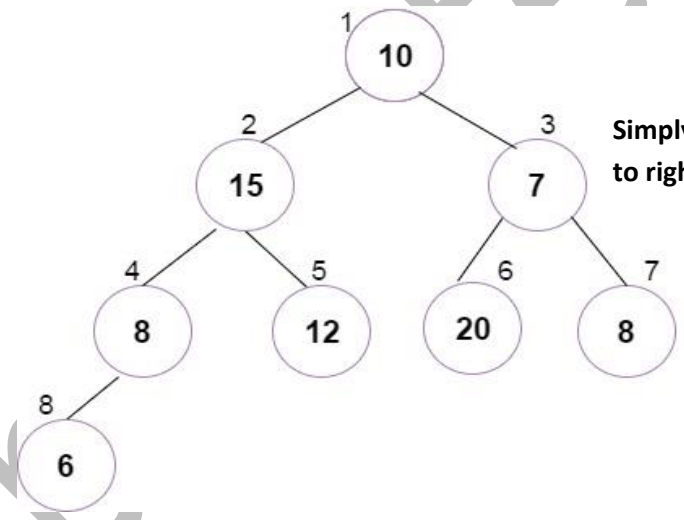
A Min heap is defined analogously i.e the value at 'N' is less than or equal to the value of children of 'N'. It is also known as Ascending Heap.



Example of Heap Sort:

Construct a heap from elements 10, 15, 7, 8, 12, 20, 8, 6.

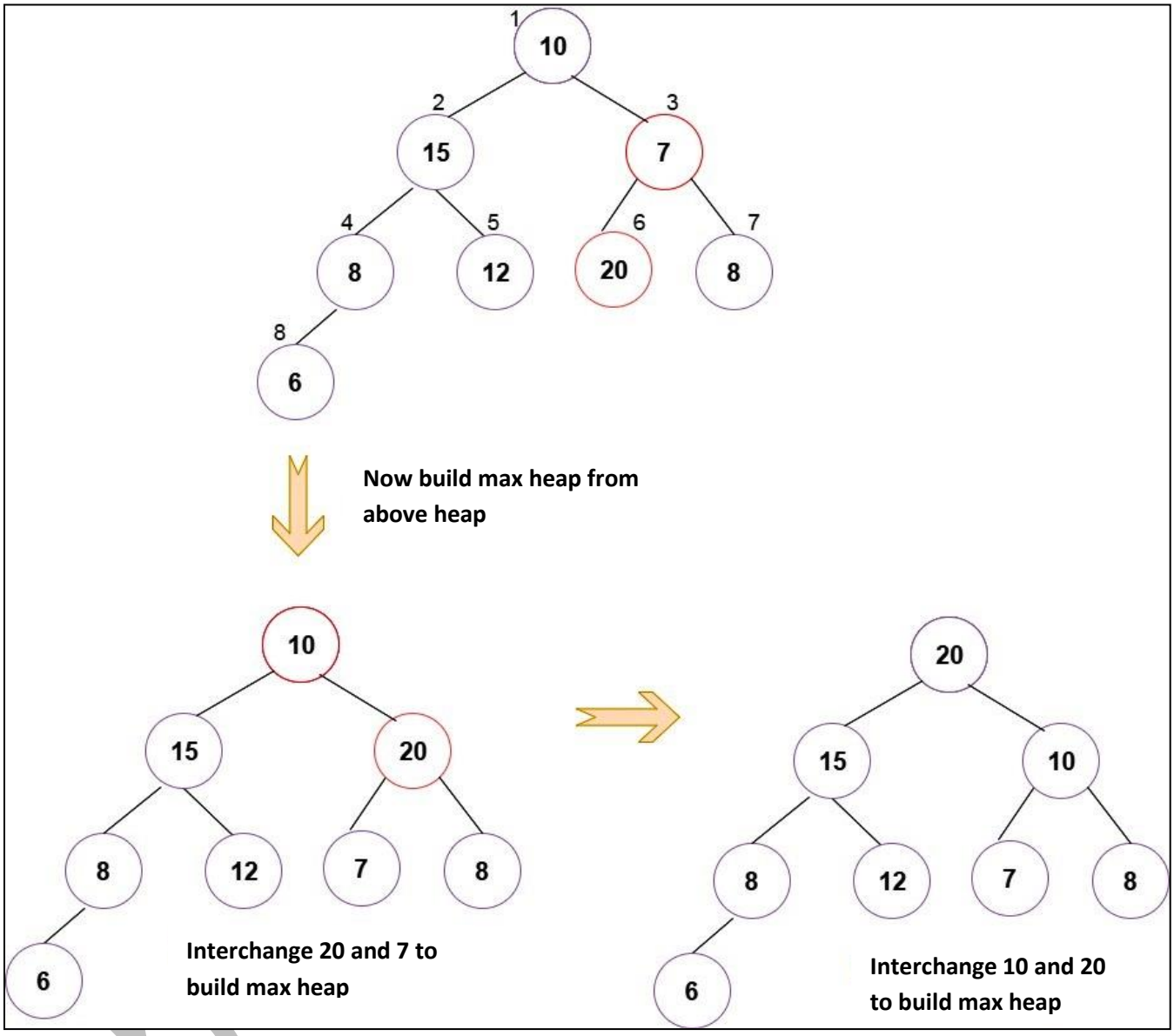
The first element is 10, make it as a Root. The next element 15 becomes left child of root and 3rd element becomes right child of root. Repeat this untill all elements are used.



Simply insert the element from left to right in order.

Build max heap from above heap

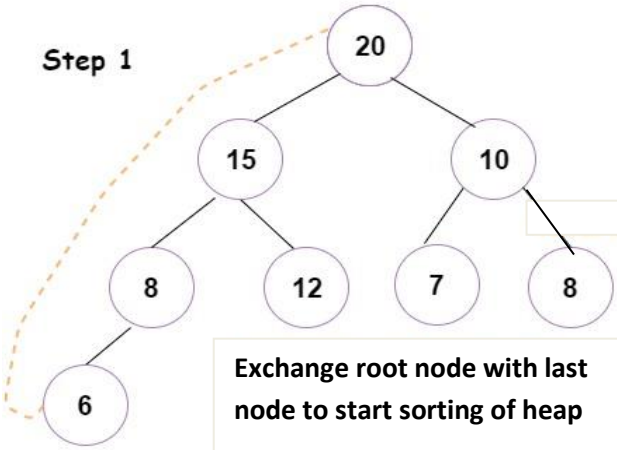
In right subtree, since 20 is greater than 7 and we are constructing a max heap so interchange them and make max heap. After interchanging, still the max heap property is not satisfied so interchange 20 and 10 also.



Sort above heap using heap sorting technique.

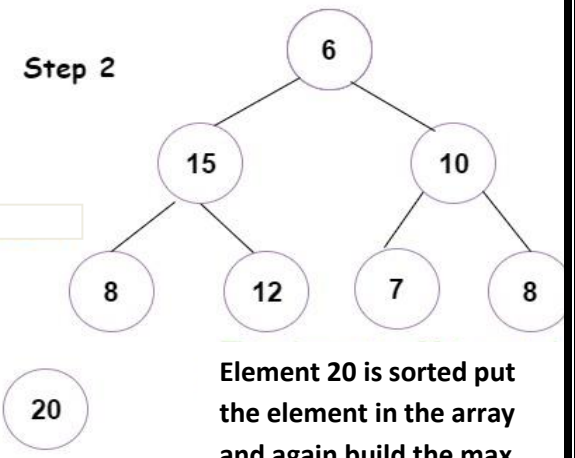
Start Sorting

Step 1



Exchange root node with last node to start sorting of heap

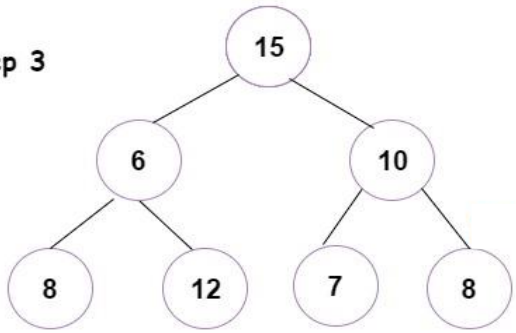
Step 2



20

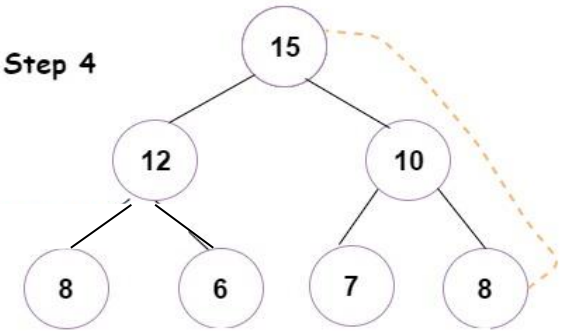
Element 20 is sorted put the element in the array and again build the max heap

Step 3



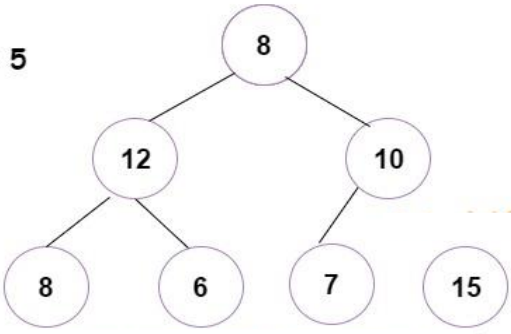
Interchange 12 and 6 to obtain max heap

Step 4



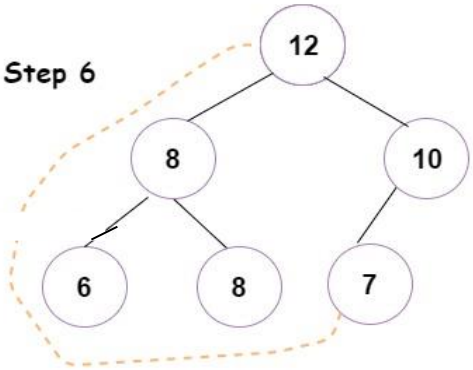
Exchange 15 with 8

Step 5



Put the element 15 in the array next to the previous element an again build the max heap.

Step 6



Exchange 12 with 7

Q.18 Write down the algorithm for heap sort.

Ans.

PARTITION (ARR, BEG, END, LOC)

```
Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG=0
Step 2: Repeat Steps 3 to 6 while FLAG =
Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND
            LOC != RIGHT SET RIGHT = RIGHT - 1
        [END OF LOOP]
Step 4: IF LOC = RIGHT
            SET FLAG = 1
        ELSE IF ARR[LOC] > ARR[RIGHT]
            SWAP ARR[LOC] withARR[RIGHT]
            SET LOC = RIGHT
        [END OF IF]
Step 5: IF FLAG = 0
            Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
            SET LEFT = LEFT + 1
            [END OF LOOP]
Step 6:     IF LOC = LEFT
                SET FLAG = 1
            ELSE IF ARR[LOC] < ARR[LEFT]
                SWAP ARR[LOC] withARR[LEFT]
                SET LOC = LEFT
            [END OF IF]
        [END OF IF]
Step 7: [END OF LOOP]
Step 8: END
```

QUICK_SORT (ARR, BEG, END)

```
Step 1: IF (BEG < END)
            CALL PARTITION (ARR, BEG, END, LOC)
            CALL QUICKSORT(ARR, BEG, LOC - 1)
            CALL QUICKSORT(ARR, LOC + 1, END)
        [END OF IF]
Step 2: END
```

Q.19 Explain the concept of radix sort?

Ans. Let 'n' be the number of elements and 'd' be the number of digit then it will require 'd' passes to be sorted completely. So in the first pass, the elements or number are sorted on the basis of last digit or letter in it. In the second pass, each number is arranged according to the second last letter in it and the pass continues until the first letter is considered from the number.

This algorithm is not suitable for large digit number as its worst case and average case complexity are O(n), where 'n' is the number of items.

In Pass 1 : Elements are sorted on the basis of unit digit.

In Pass 2 : The output of Pass 1 is used as input for Pass 2. In Pass 2 elements are sorted on the basis of ten's digit.

In Pass 3 : The output of Pass 2 is used as input for Pass 3. In Pass 3 elements are sorted on the basis of hundred digit. The final list is sorted.

Sort the numbers given below using radix sort. **345, 654, 924, 123, 567, 472, 555, 808, 911**

Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

Pass 2:

Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				

924			**924**							
345					**345**					
555						**555**				
567							**567**			
808	**808**									

Pass 3:

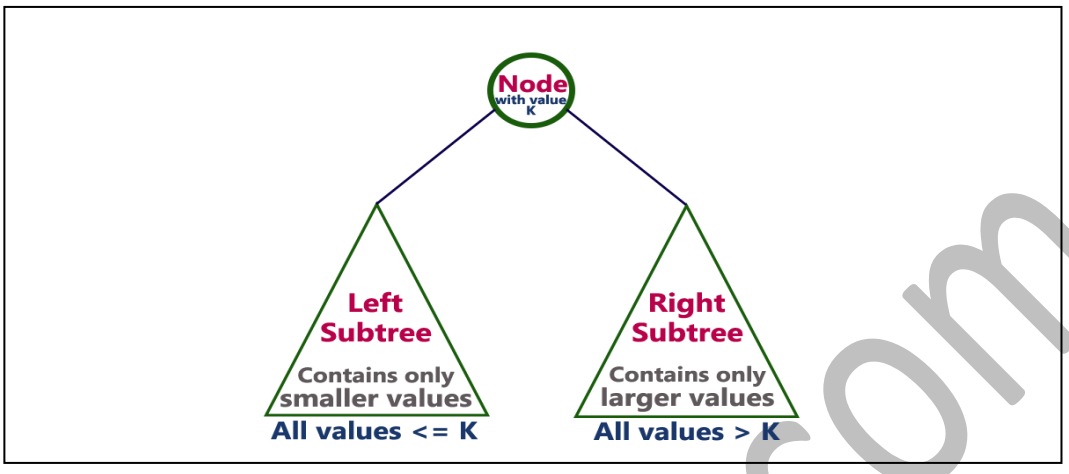
Number	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
808									**808**	
911										**911**
123		**123**								
924										**924**
345				**345**						
654							**654**			
555						**555**				
567						**567**				
472					**472**					

Q20. Explain binary search tree.

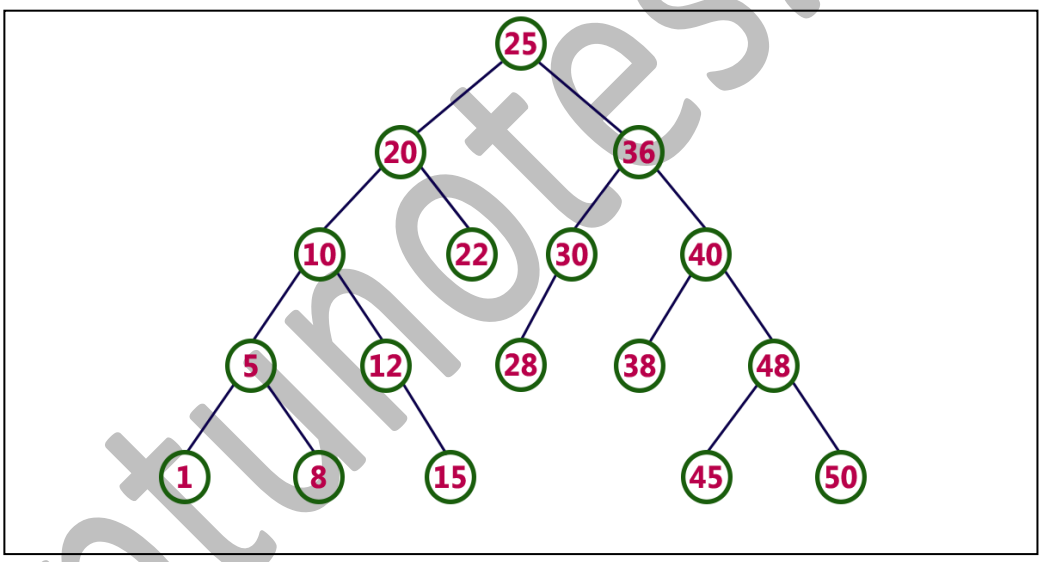
Ans A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

- The left sub-tree of a node has a key less than or equal to its parent node's key.

- The right sub-tree of a node has a key greater than to its parent node's key.



The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.



Q22. Write down the steps for searching the element in binary search tree?

Ans.

In a binary search tree, the search operation is performed with O(log n) time complexity. The search operation is performed as follows:

Step 1: Read the search element from the user

Step 2: Compare, the search element with the value of root node in the tree.

Step 3: If both are matching, then display **"Given node found!!!"** and terminate the function

Step 4: If both are not matching, then check whether search element is smaller or larger than that node value.

Step 5: If search element is smaller, then continue the search process in left subtree.

Step 6: If search element is larger, then continue the search process in right subtree.

Step 7: Repeat the same until we found exact element or we completed with a leaf node

Step 8: If we reach to the node with search value, then display "**Element is found**" and terminate the function.

Step 9: If we reach to a leaf node and it is also not matching, then display **"Element not found"** and terminate the function.

Q23. Write down the steps for inserting an element in binary search tree?

Ans

In a binary search tree, the insertion operation is performed with O(log n) time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows.

Step 1: Create a newNode with given value and set its left and right to NULL.

Step 2: Check whether tree is Empty.

Step 3: If the tree is Empty, then set set root to newNode.

Step 4: If the tree is Not Empty, then check whether value of newNode is smaller or larger than the node (here it is root node).

Step 5: If newNode is smaller than or equal to the node, then move to its left child. If newNode is larger than the node, then move to its right child.

Step 6: Repeat the above **step** until we reach to a leaf node (e.i., reach to NULL).

Step 7: After reaching a leaf node, then insert the newNode as left child if newNode is smaller or equal to that leaf else insert it as right child.

Q24. Write down the steps for deleting an element in binary search tree?

Ans In a binary search tree, the deletion operation is performed with O(log n) time complexity. Deleting a node from Binary search tree has follwing three cases...

Case 1: Deleting a Leaf node (A node with no children)

Case 2: Deleting a node with one child

Case 3: Deleting a node with two children

Case 1: Deleting a leaf node

We use the following **step**s to delete a leaf node from BST.

Step 1: Find the node to be deleted using search operation

Step 2: Delete the node using free function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

We use the following **step**s to delete a node with one child from BST.

Step 1: Find the node to be deleted using search operation

Step 2: If it has only one child, then create a link between its parent and child nodes.

Step 3: Delete the node using free function and terminate the function.

Case 3: Deleting a node with two children

We use the following **step**s to delete a node with two children from BST.

Step 1: Find the node to be deleted using search operation

Step 2: If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.

Step 3: Swap both deleting node and node which found in above **step**.

Step 4: Then, check whether deleting node came to case 1 or case 2 else goto **step**s 2

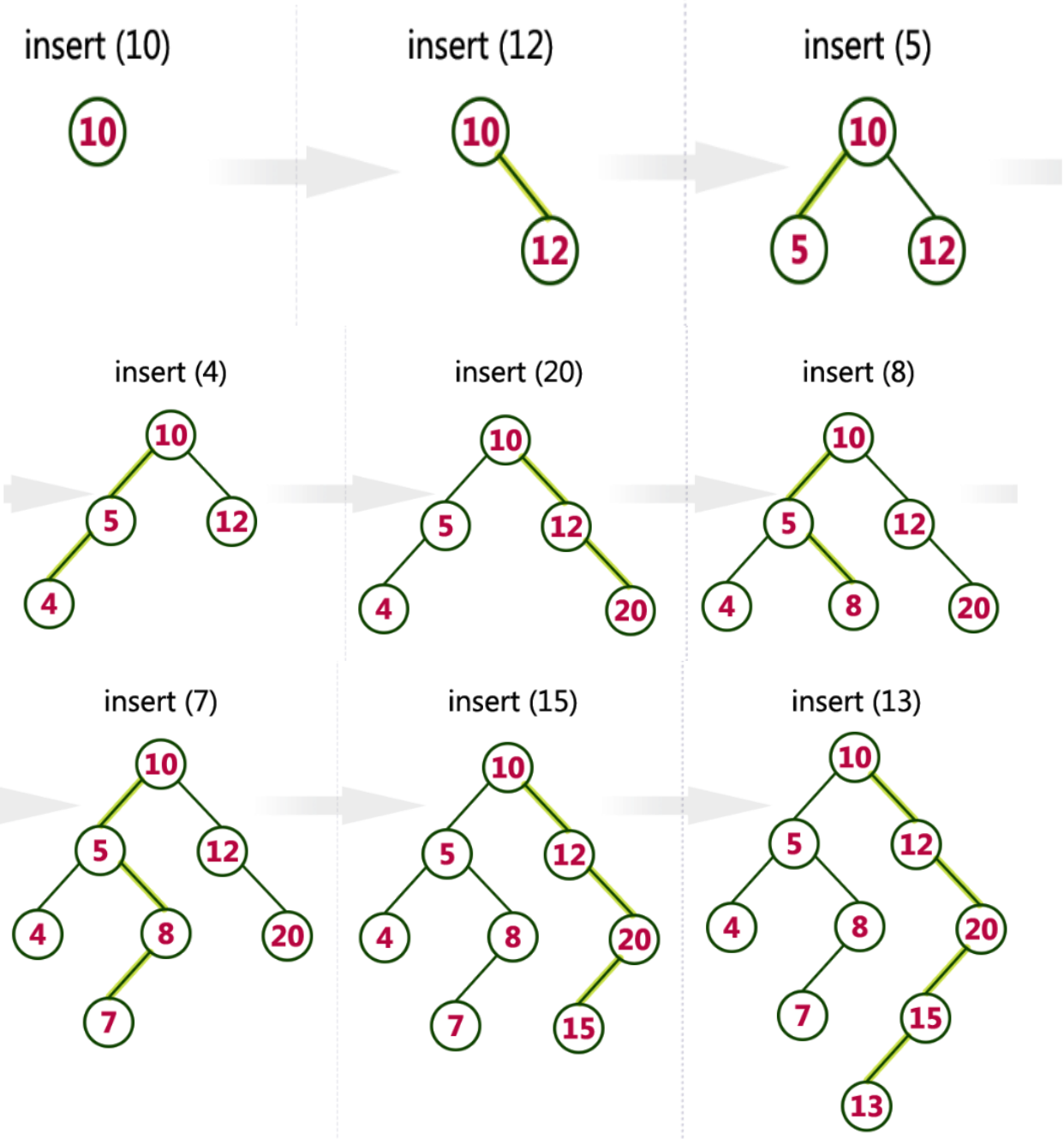
Step 5: If it comes to case 1, then delete using case 1 logic.

Step 6: If it comes to case 2, then delete using case 2 logic.

Step 7: Repeat the same process until node is deleted from the tree.

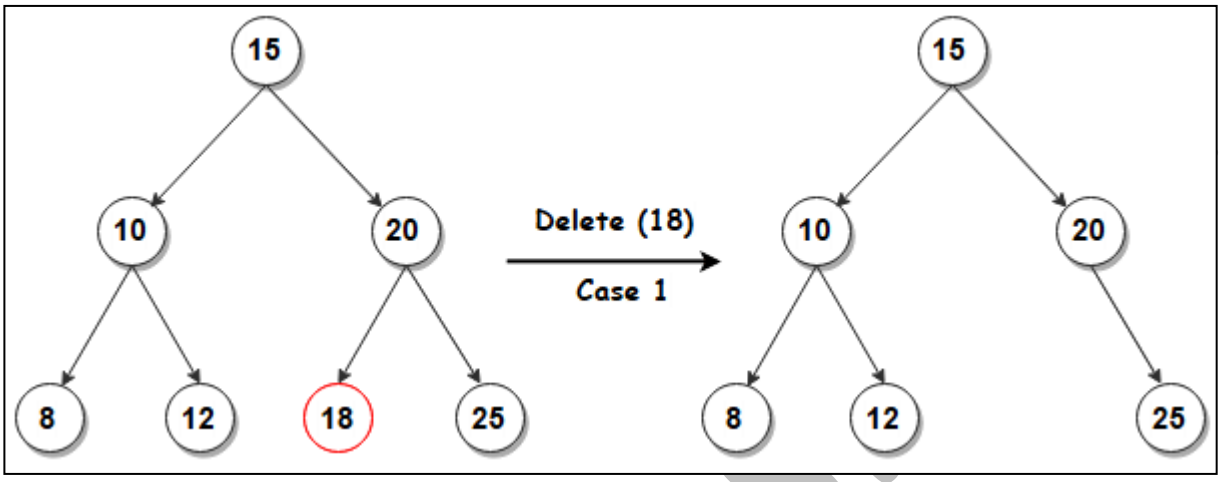
Q.25 Construct a Binary Search Tree by inserting the following sequence of numbers.
10, 12, 5, 4, 20, 8, 7, 15 and 13/

Ans.

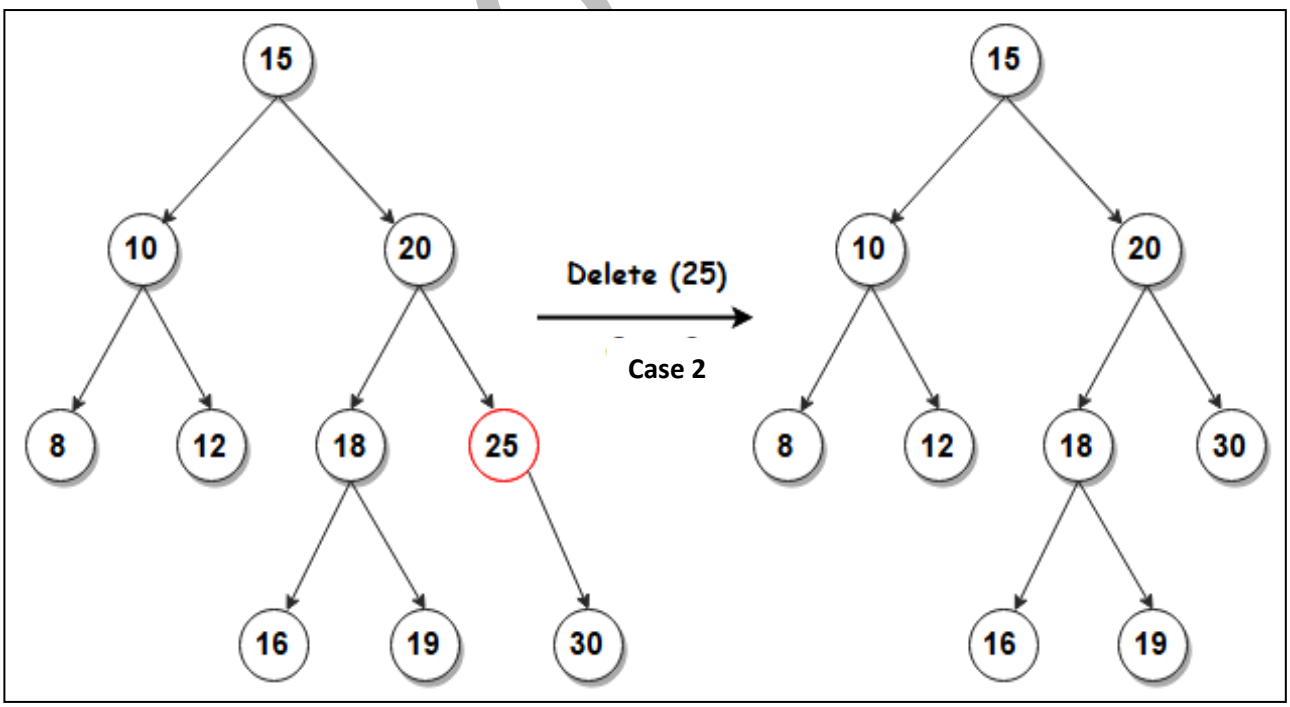


Q.26 Explain all case with suitable example for deleting the node in BST.

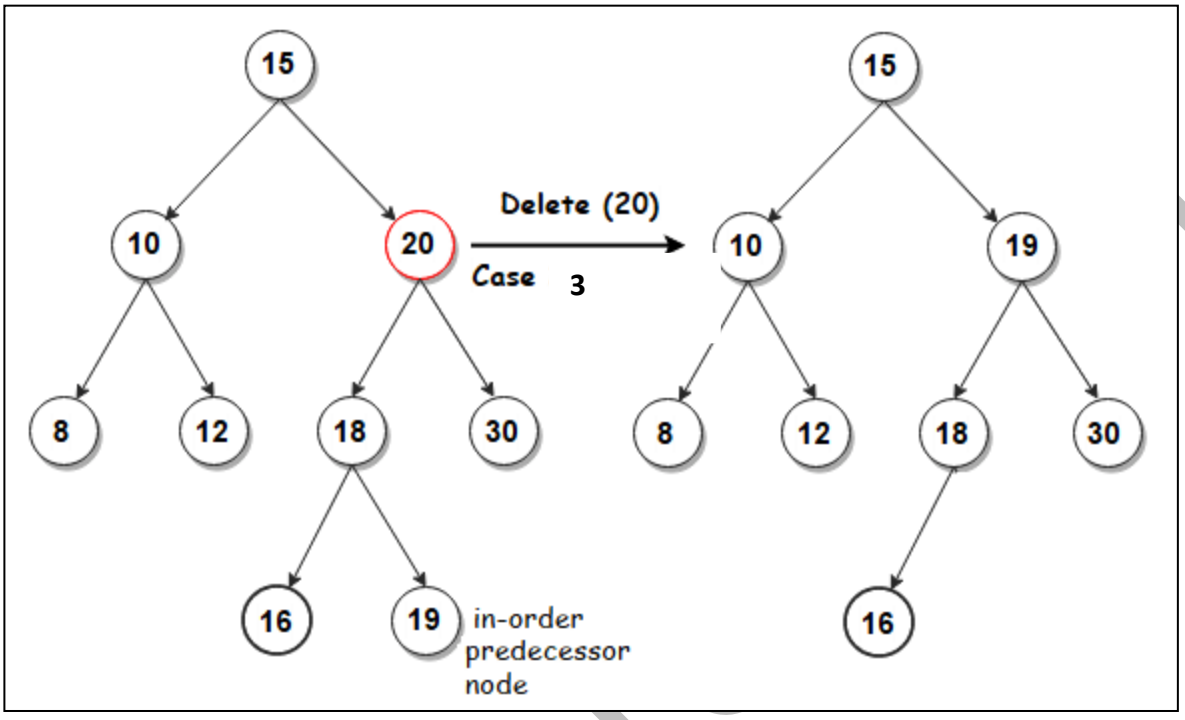
Ans. Case1 : Deleting a node with no child.



Case 2: Deleting a node with one child



Case 3: Deleting a node with two children



Q.27 Define AVL tree?

Ans. AVL tree is a self balanced binary search tree.

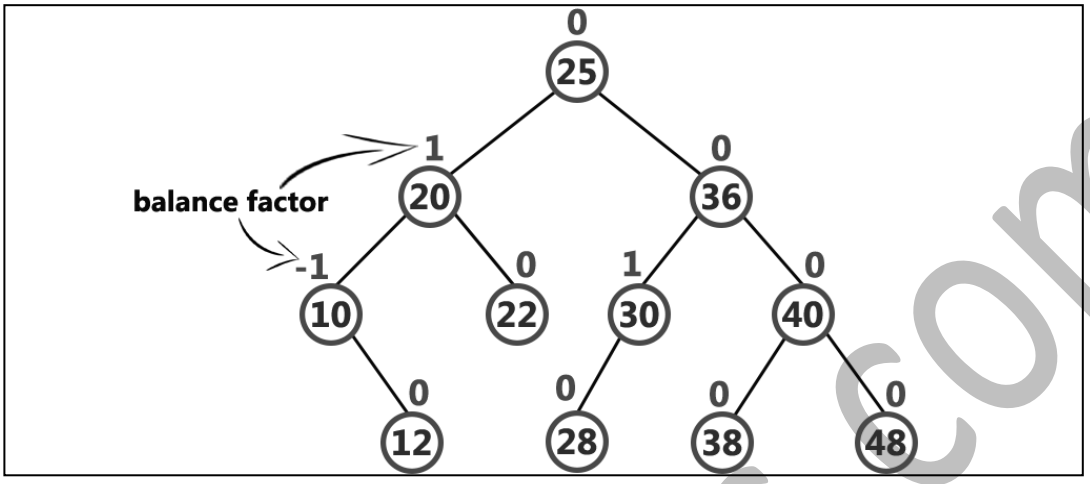
- That means, an AVL tree is also a binary search tree but it is a balanced tree.
- A binary tree is said to be balanced, if the difference between the height of left and right subtrees of every node in the tree is either -1, 0 or +1.
- In other words, a binary tree is said to be balanced if for every node, height of its children differ by at most one.
- In an AVL tree, every node maintains a extra information known as balance factor.
- The AVL tree was introduced in the year of 1962 by G.M. Adelson-Velsky and E.M. Landis.

- Balance factor of a node is the difference between the heights of left and right subtrees of that node.
- The balance factor of a node is calculated either height of left subtree - height of right subtree (OR) height of right subtree - height of left subtree.

- In the following explanation, we are calculating as follows...

Balance factor = heightOfLeftSubtree - heightOfRightSubtree

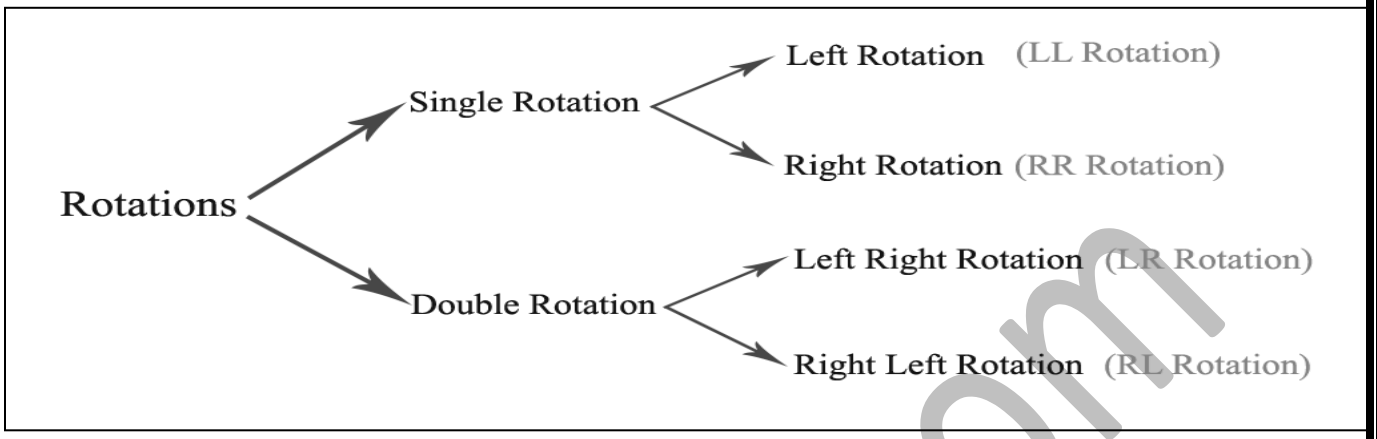
Every AVL Tree is a binary search tree but all the Binary Search Trees need not to be AVL trees.



Q.28 Explain AVL tree operation?

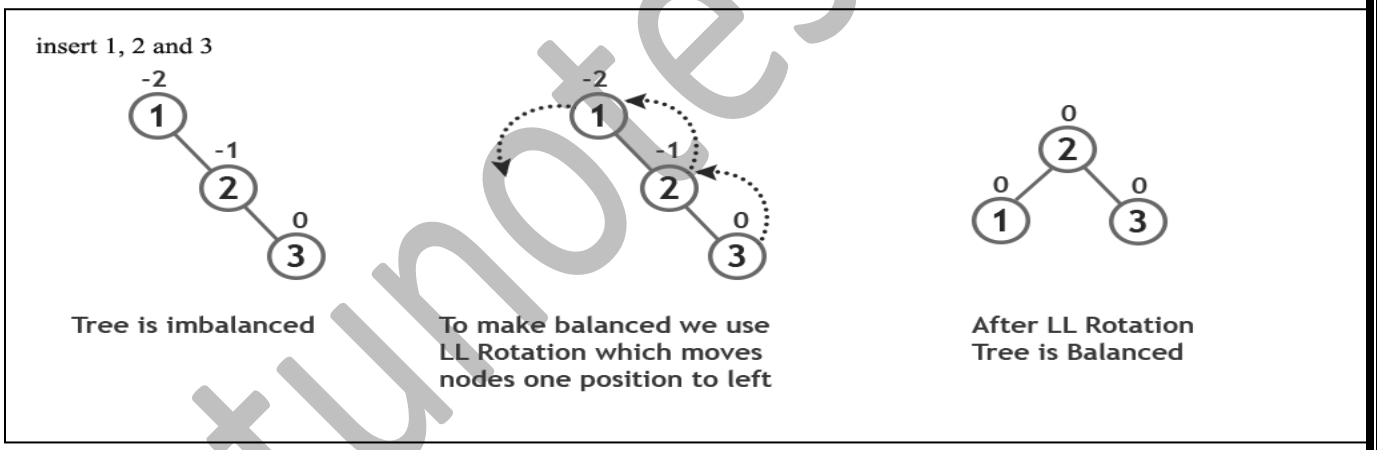
Ans. In AVL tree, after performing every operation like insertion and deletion we need to check the balance factor of every node in the tree.

- If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced.
- We use rotation operations to make the tree balanced whenever the tree is becoming imbalanced due to any operation.
- Rotation operations are used to make a tree balanced.
- Rotation is the process of moving the nodes to either left or right to make tree balanced.
- There are four rotations and they are classified into two types.



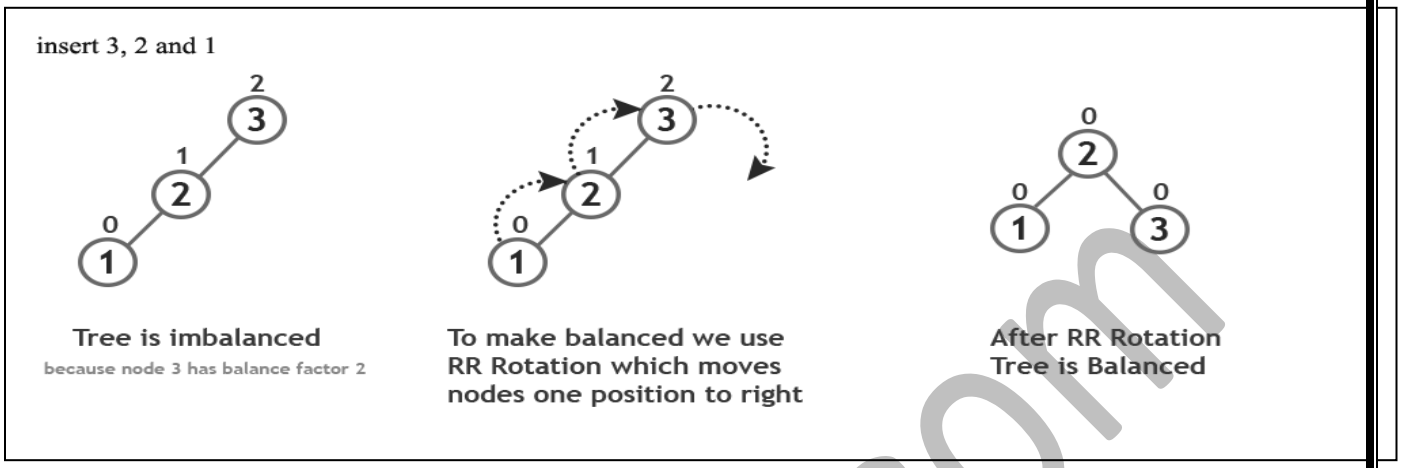
Single Left Rotation (LL Rotation)

In LL Rotation every node moves one position to left from the current position. To understand LL Rotation, let us consider following insertion operations into an AVL Tree.



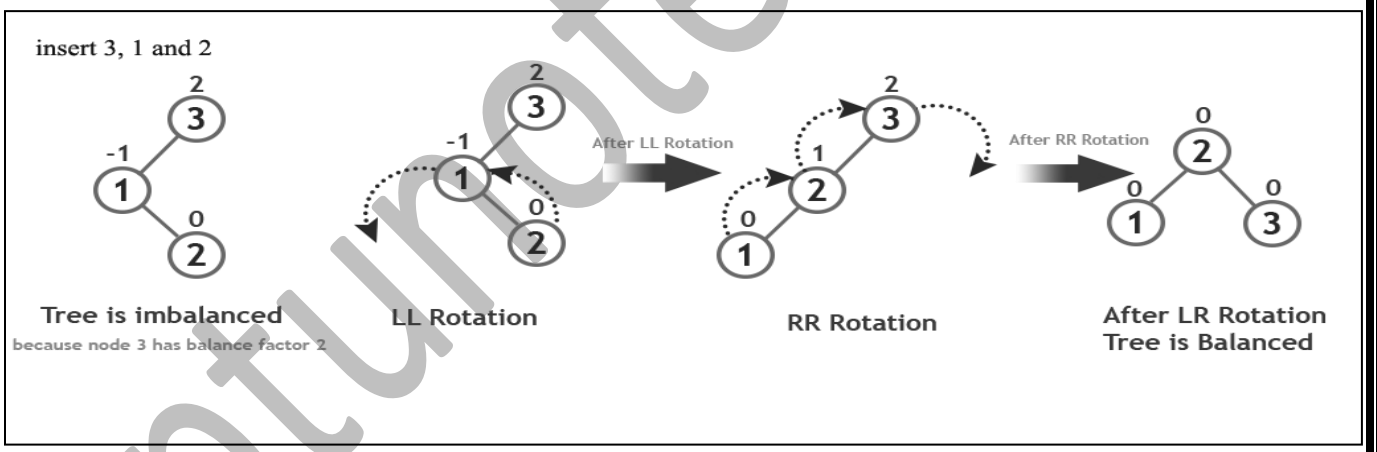
Single Right Rotation (RR Rotation)

In RR Rotation every node moves one position to right from the current position. To understand RR Rotation, let us consider following insertion operations into an AVL Tree.



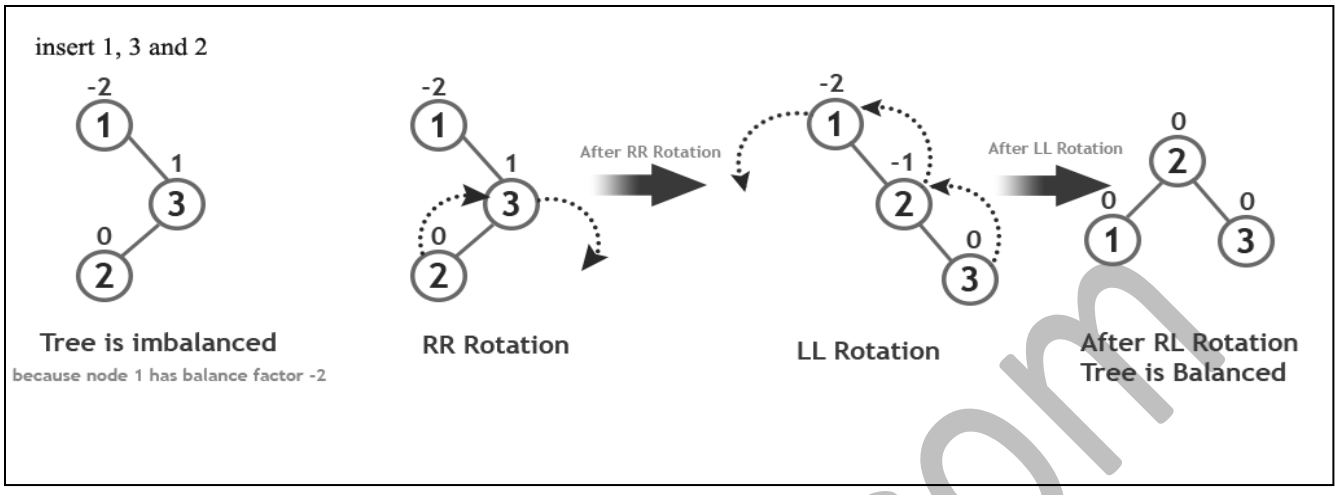
Left Right Rotation (LR Rotation)

The LR Rotation is combination of single left rotation followed by single right rotation. In LR Rotation, first every node moves one position to left then one position to right from the current position. To understand LR Rotation, let us consider following insertion operations into an AVL Tree.

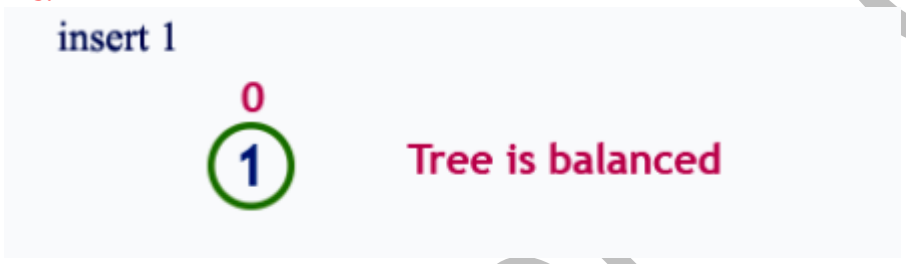


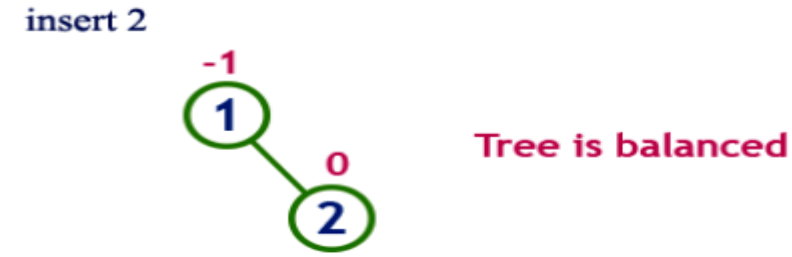
Right Left Rotation (RL Rotation)

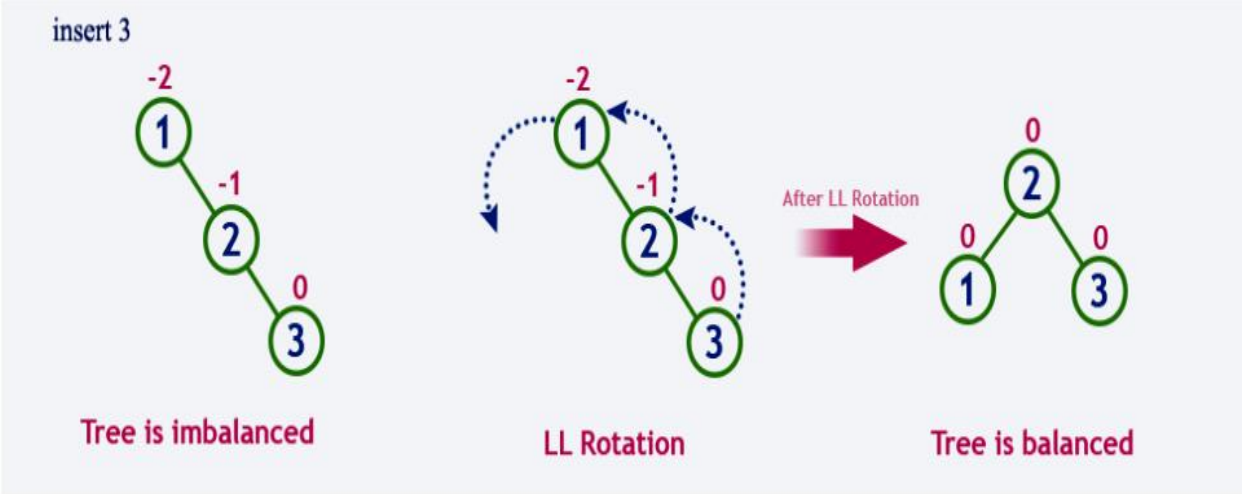
The RL Rotation is combination of single right rotation followed by single left rotation. In RL Roration, first every node moves one position to right then one position to left from the current position. To understand RL Rotation, let us consider following insertion operations into an AVL Tree.

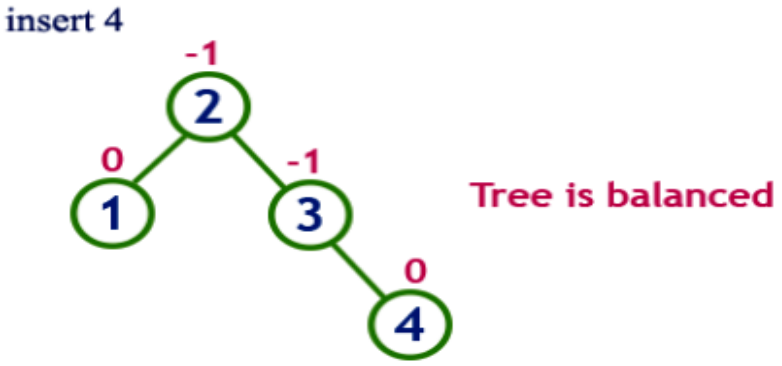


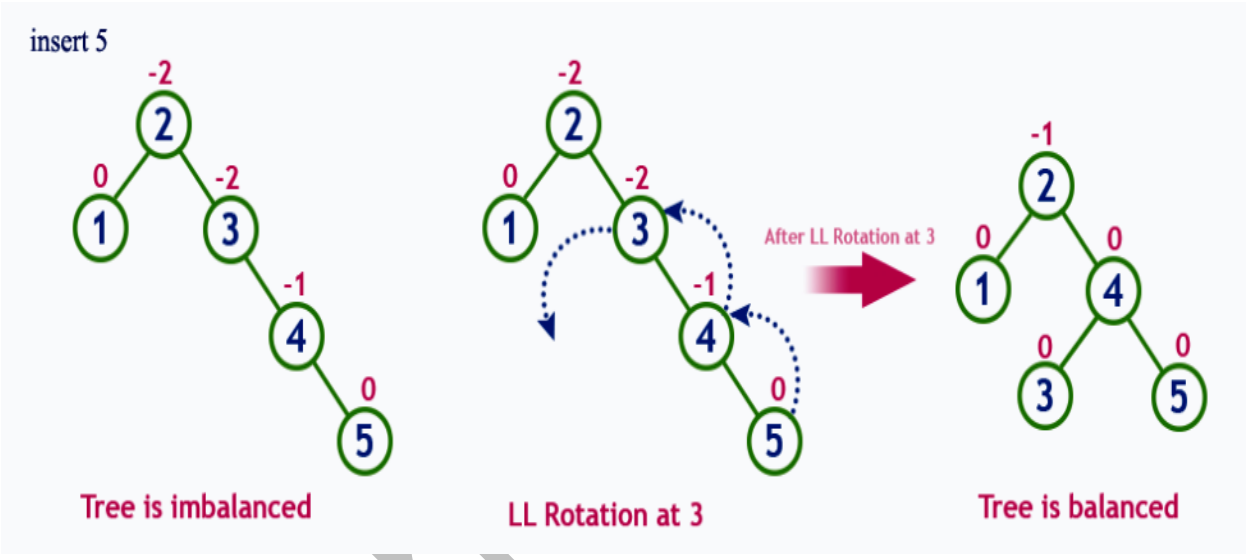
Q.29 Construct an AVL Tree by inserting numbers from 1 to 8.
Ans.

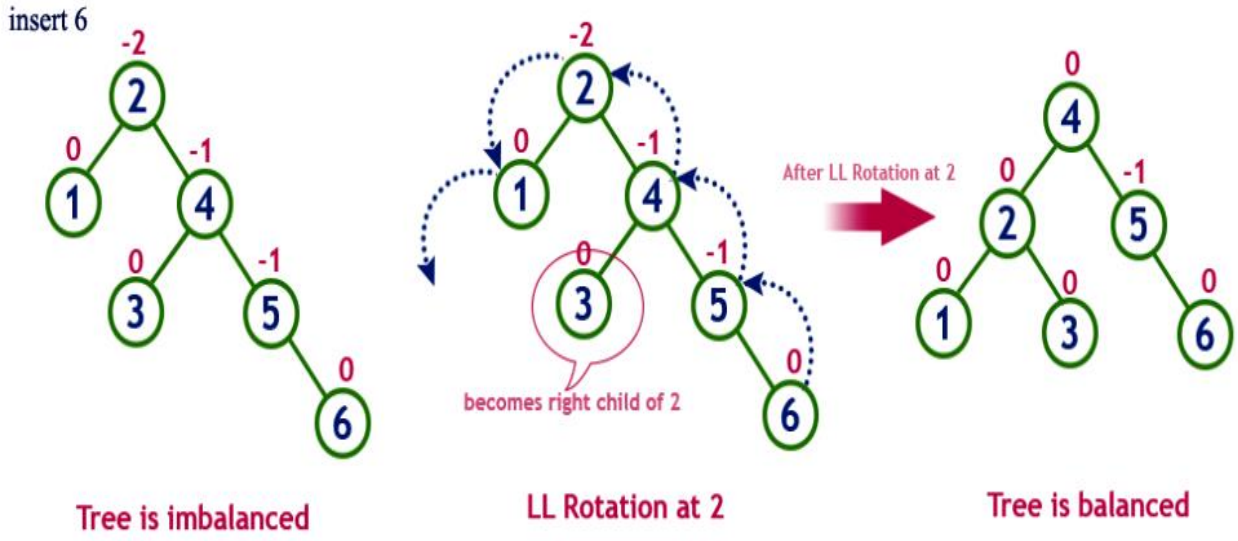


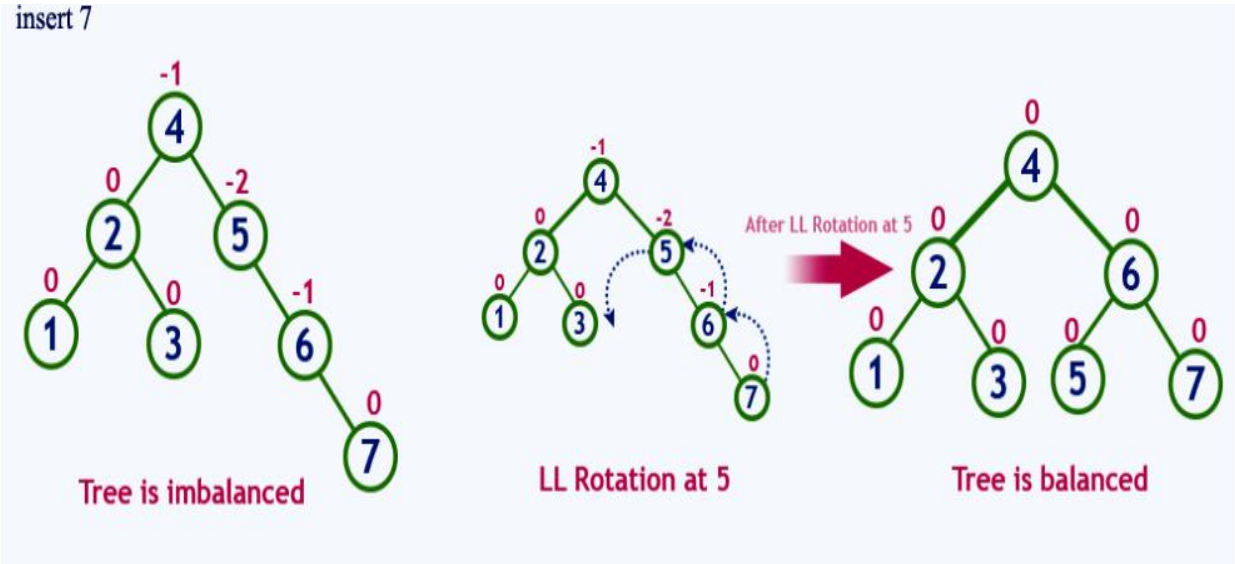


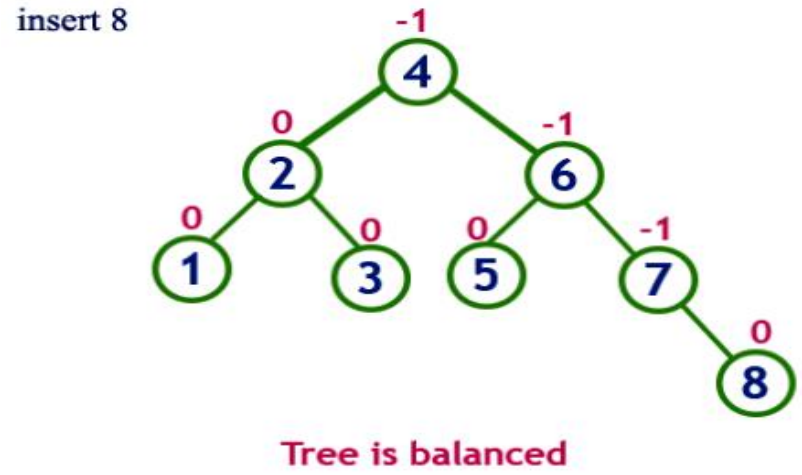



insert 4
-1
2
0
-1
1
3
0
4
Tree is balanced


insert 5
-2
2
0
-2
1
3
-1
4
0
5
Tree is imbalanced
-2
2
0
-2
1
3
-1
4
0
5
LL Rotation at 3
After LL Rotation at 3
-1
2
0
0
1
4
0
0
3
5
Tree is balanced


insert 6
-2
2
0
-1
1
4
0
-1
3
5
0
6
Tree is imbalanced
-2
2
0
-1
1
4
0
-1
3
5
0
6
becomes right child of 2
LL Rotation at 2
After LL Rotation at 2
0
4
0
-1
2
5
0
0
0
1
3
6
Tree is balanced





Q.29 Explain the term BTree?

Ans.B-Tree can be defined as follows.

- B-Tree is a self-balanced search tree with multiple keys in every node and more than two children for every node.
- Here, number of keys in a node and number of children for a node is depend on the order of the B-Tree. Every B-Tree has order.

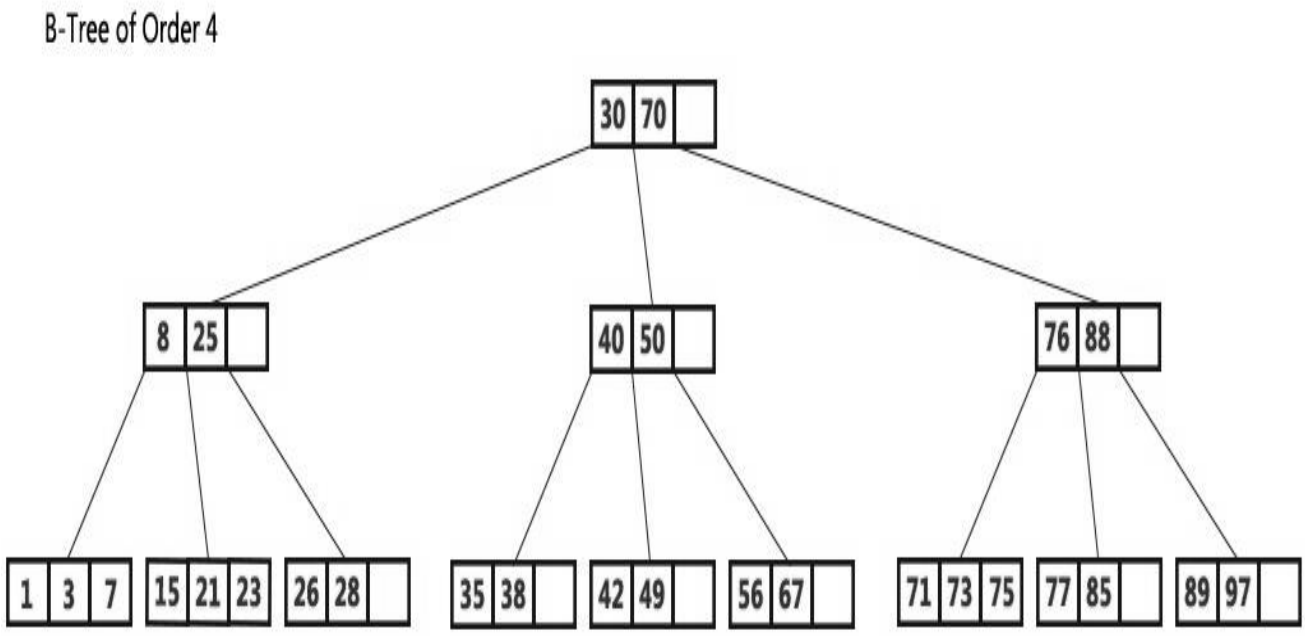
B-Tree of Order m has the following properties...

- **Property #1 -** All the leaf nodes must be at same level.

- **Property #2** - All nodes except root must have at least [m/2]-1 keys and maximum of m-1 keys.
- **Property #3** - All non leaf nodes except root (i.e. all internal nodes) must have at least m/2 children.
- **Property #4** - If the root node is a non leaf node, then it must have at least 2 children.
- **Property #5** - A non leaf node with n-1 keys must have n number of children.
- **Property #6** - All the key values within a node must be in Ascending Order.

For example, B-Tree of Order 4 contains maximum 3 key values in a node and maximum 4 children for a node.

Example:



Q.30 Explain the insertion operation in B Tree?

Ans. **I**n a B-Tree, the new element must be added only at leaf node. That means, always the new keyValue is attached to leaf node only. The insertion operation is performed as follows.

Step 1: Check whether tree is Empty.
Step 2: If tree is Empty, then create a new node with new key value and insert into the tree as a root node.
Step 3: If tree is Not Empty, then find a leaf node to which the new key value cab be added using Binary Search Tree logic.
Step 4: If that leaf node has an empty position, then add the new key value to that leaf node by maintaining ascending order of key value within the node.

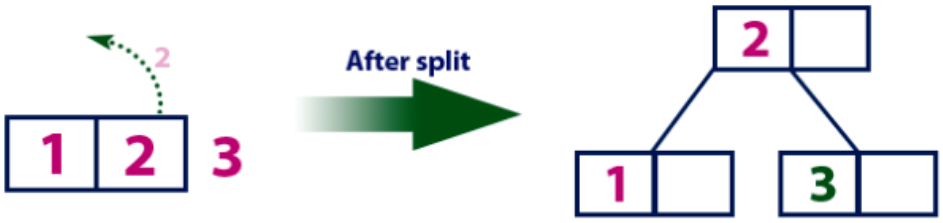
Step 5: If that leaf node is already full, then split that leaf node by sending middle value to its parent node. Repeat the same until sending value is fixed into a node.
Step 6: If the splitting is occurring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Q.31 Construct the B Tree of order 3 inserting an element number from 1 to 10.

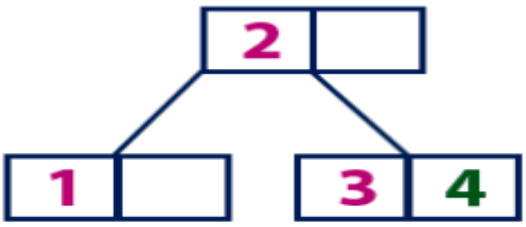
Ans. Step 1: Insert 1

Step 2: Insert 2

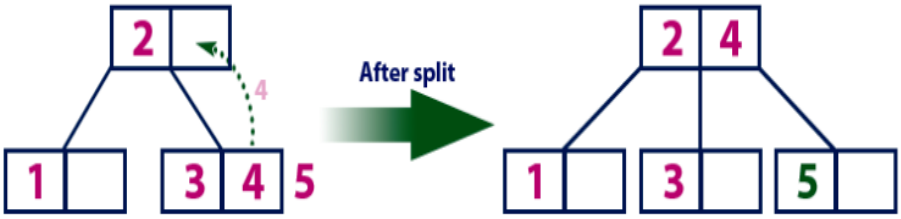
Step 3: Insert 3



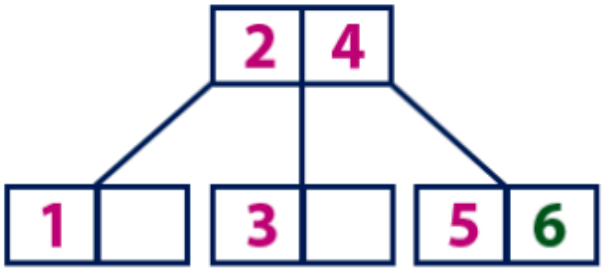
Step 4: Insert 4



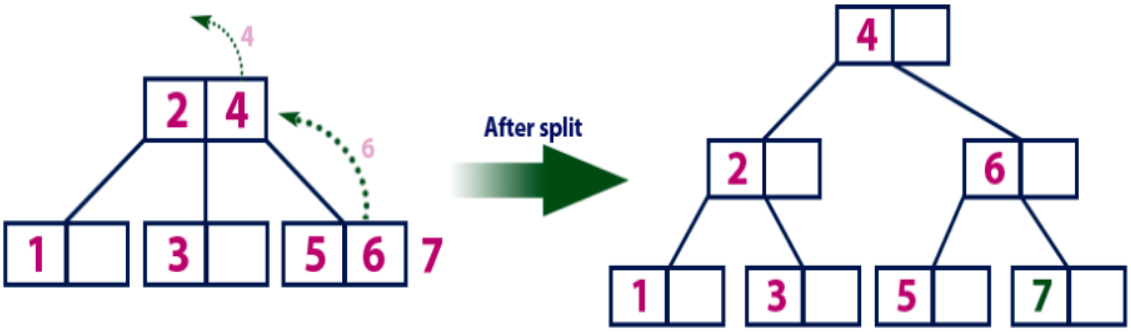
Step 5: Insert 5



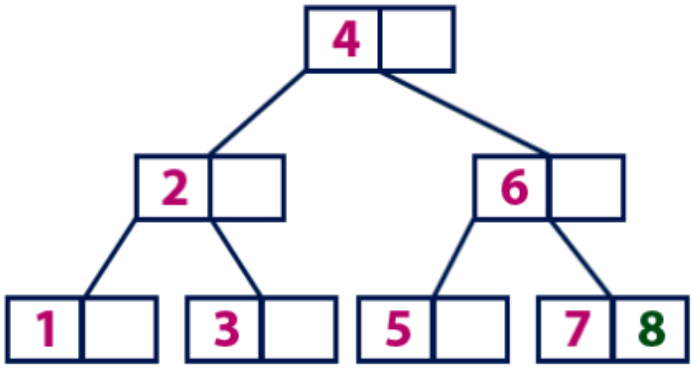
Step 6: Insert 6



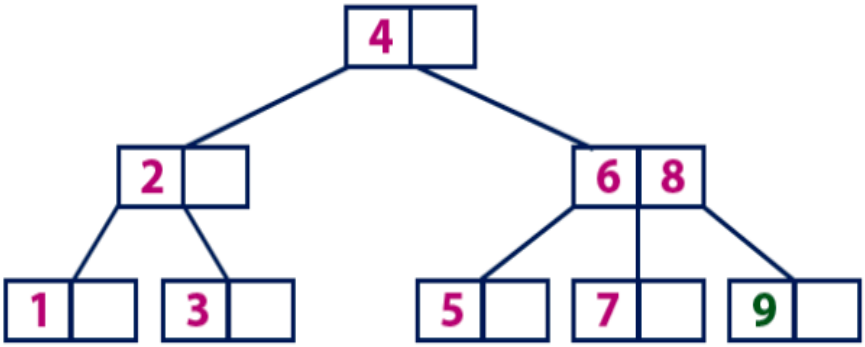
Step 7: Insert 7



Step 8: Insert 8



Step 9: Insert 9



Step 10: Insert 10