VIT
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

REG.NO.:

SLOT: E1+ TE1

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
CONTINUOUS ASSESSMENT TEST - II
FALL SEMESTER 2025-2026

Programme Name & Branch : B.Tech. & CSE
Course Code and Course Name : BCSE202L and Data Structure and Algorithms
Faculty Name(s) : Common to ALL
Class Number(s) : Common to ALL
Date of Examination : 09 October 2025
Exam Duration : 90 minutes **Maximum Marks: 50**

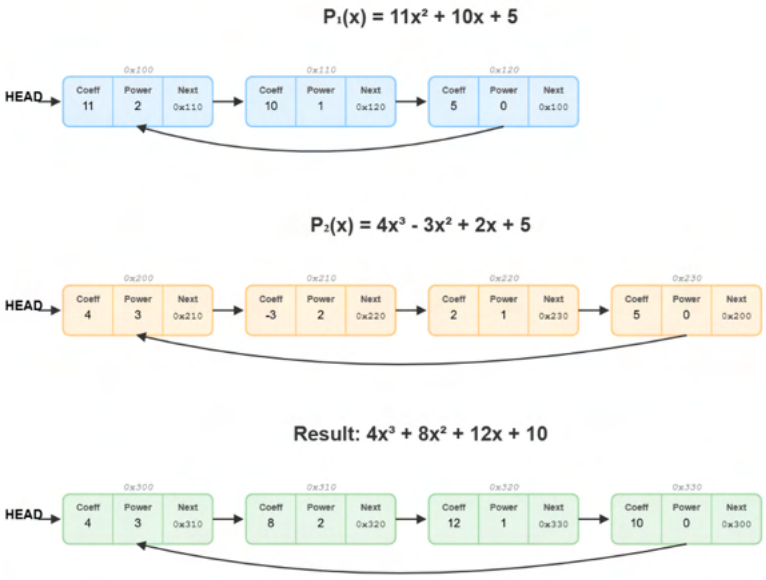
General instruction(s):

- Answer All Questions

Q. No	Question	Module	Marks	CO	BL

1.	A) Develop a pseudocode for a function **addPolynomials**(poly1, poly2) that takes two such polynomial lists and returns a new circular linked list representing their sum. **[5 M]**
	Illustrate the complete process by adding the following two polynomials. Show the initial lists and the final resulting list with diagrams.
	• $P_1(x) = 8x^2 + 3x^2 + 10x + 5$
	• $P_2(x) = 4x^3 - 3x^2 + 2x + 5$
	Solution:
	Note: Professors are requested to consider the student assumption in Polynomial1
	AddPolynomial(ploy1, poly 2):
	`result = init_circular_link_list( )`
	`p1 = poly1-> head`
	`p2 = ply2->head`
	`// Handling case when either of p1 or p2 is NULL`
	`if p1== NULL`
	`return p2`
	`else if p2 ==NULL`
	`return p1`
	`// Compare the coefficient and add to circular link list`
	`while True`
	`if p1.exp == p2.exp`
	`AddAtLast(result, p1.coeff + p2.coeff, p1.exp)`
	`p1 = p1->next`
	`p2 = p2->next`
	`else if p1.exp > p2.exp`
	`AddAtLast(result, p1.coeff, p1.exp)`
	`p1 = p1->next`
	`else`
	`AddAtLast(result, p2.coeff, p2.exp)`
	`p2 = p2->next`
	`// While Loop exit condition`
		M02	10	2	6

```
        if (p1 == poly1->head) && (p2== poly2->head)
            break

    while (p1->next !=NULL)
    // At this point, the main loop has ended because at least one polynomial has been fully traversed. Now, we copy the remaining terms from the  other polynomial.

  // 2. First remainder loop: Copy leftover terms from poly1, if any.
    while (p1 != poly1.head)
            AddAtLast(result, p1.coeff, p1.exp)
            p1 = p1->next

  //3. Second remainder loop: Copy leftover terms from poly2, if any.
    while (p2 != poly2.head)
            AddAtLast(result, p2.coeff, p2.exp)
            p2 = p2->next

     return result
```



B) Given a singly linked list $L: L_0 \rightarrow L_1 \rightarrow L_2 \ldots \rightarrow L_{n-1} \rightarrow L_n$ the task is to reorder it $L: L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \ldots$ without using any extra data structures for storage (i.e., O(1) space complexity). Develop a pseudocode for a function **reorderList(head)** that performs this operation. **[5 M]**

input_list: 10 → 20 → 30 → 40 → 50 → 60 → NULL

output_list: 10 → 60 → 20 → 50 → 30 → 40 → NULL

Solution:

```
ReorderList( LinkList L1)
        //Handle Edge case for L1 with 0, 1, 2 Node
        if L1->head ==NULL || L1->head->next =NULL ||
              L1->head-next->next == NULL
            return L1
        // A. Find the middle node of Link list and node prev to it
        prevNode, midNode = FindMid( L1)
        prevNode->next = NULL // Dividing L1 into two halves
        // 1B. Reverse the second half of the link list
        L2 = reverseLinkList(midNode)
        // 1C. Merge the two link list L1 and L2 alternatively
        L3  = Merge( L1, L2)
        Return L3

  A) FindMid(LinkList L1)
        // Handle the edge case for L1 with 0, 1, or 2 node
        if L1->head ==NULL || L1->head->next =NULL ||
              L1->head-next->next == NULL
            return NULL, L1
       // Initialization of pointers – fast, slow, prev
        slow = L1->head; fast = L1->head; prev = NULL

        // Moving slow & fast pointer one and two node at a time
      respectively
        while(fast!=NULL && fast->next!=NULL)
            prev = slow
            slow = slow->next
            fast = fast->next->next
       //For even number for it will return second middle node
       return prev, slow

  B)  ReverseLinkList( Link List L1)
        // Handle the edge case for L1 with o or 1 node
        if (L1->head ==NULL || L1->head->next =NULL)
            return L1
       // Reversing Link list using three pointer approach
      left = NULL; middle = L1->head; right = middle->next;
      while(middlet !=NULL)
            middle->next = left;
            left = middle;
            middle = right;
            if (right!=NULL)
                right = right->next;
      L1->head = left;
      return L1
```

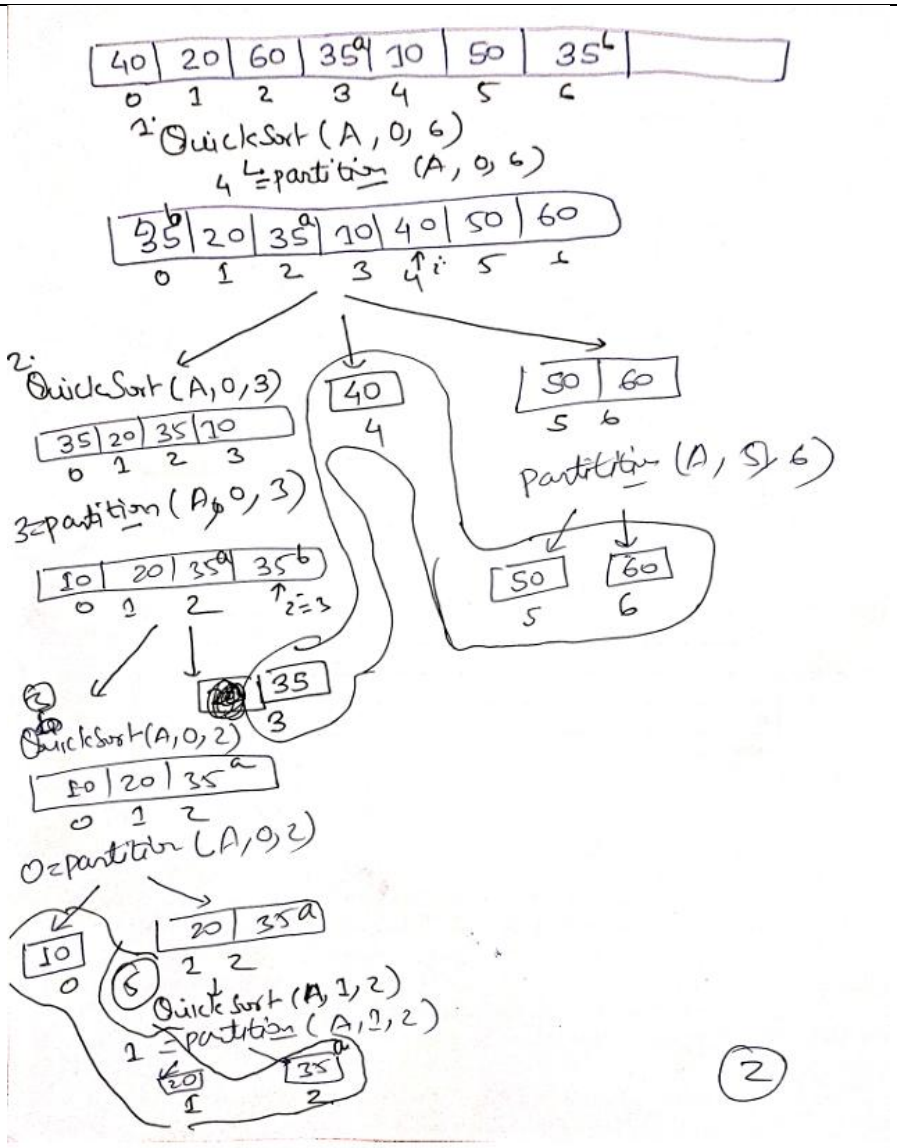
	C) MergeLinkList(LinkList L1, LinkList L2) **// To add alternative node to a new link list** **if (L1->head ==NULL)** **return L2** **else if (L2->head ==NULL)** **return L1** **temp1 = L1->head;** **temp2 = L2->head;** **while(temp1 !=NULL && temp2!=NULL)** **next1 = temp1->next** **next2 = temp2->next** **temp1->next = temp2** **if (next1 == NULL)** **break** **temp2->next = next1** **temp1 = next1** **temp2 = next2** **return L1**				
2.	Demonstrate the Quicksort on the array [45, 20, 60, 35, 10, 50, 35] using first element as pivot. Write the generic recurrence relation for the Quicksort and list its worst case, best case and average case time and space complexity. Justify whether Quicksort is in-place and stable algorithm or not? **Solution**:	M03	10	3	3

REG.NO.:

SLOT: E1+ TE1

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

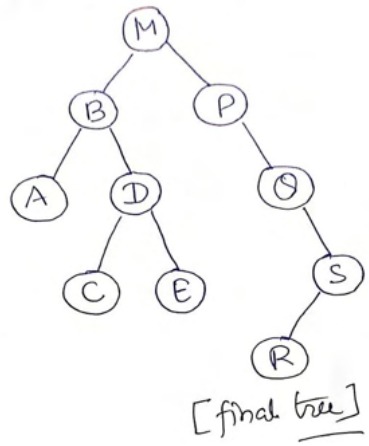
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
CONTINUOUS ASSESSMENT TEST - II
FALL SEMESTER 2025-2026



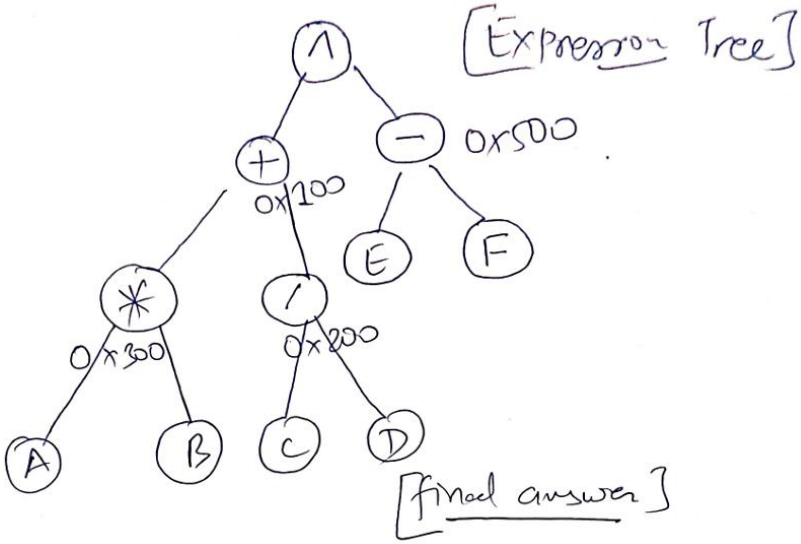
Note: Detailed solution is included at last

Generic Recurrence Relation of Quicksort:
If partitioning divides the array into subarrays of sizes k and n−k−1:

T(n)= T(k)+T(n−k−1) + Θ(n)
(where the Θ(n) term represents the cost of partitioning).

	<table><tr><th>Case</th><th>Time Complexity</th><th>Space Complexity</th><th>Remarks</th></tr><tr><td>Best Case</td><td>O(n log n)</td><td>O(log n)</td><td>Pivot divides array into nearly equal halves at each step</td></tr><tr><td>Average Case</td><td>O(n log n)</td><td>O(log n)</td><td>Random distribution of pivots gives balanced partitions on average</td></tr><tr><td>Worst Case</td><td>$O(n^2)$</td><td>O(n)</td><td>Pivot is always smallest or largest element (unbalanced partitions)</td></tr></table> • **In-place:** Yes (only uses recursion stack) • **Stable:** No (may change order of equal elements) as evident in this example				
3.	a) You are given the preorder and inorder traversals of a binary tree T. Construct the unique binary tree that corresponds to these traversals and draw it clearly. • **Preorder Traversal:** M, B, A, D, C, E, P, Q, S, R • **Inorder Traversal:** A, B, C, D, E, M, P, Q, R, S **Solution:**  **Note: Detail solution is present at the last**	M04	5	4	4
	b) Now, analyze the tree you constructed in Part (a). i. Is the constructed tree a Binary Search Tree (BST)? Justify your answer in one or two sentences. ii. Write an efficient **pseudocode** for a function **isBST**(root) that takes the root of a binary tree and returns **true** if the tree is a valid BST and **false** otherwise. The function should correctly handle the entire structure of the tree. **Solution:** **i.** **Yes** the above tree is BST.	M04	5		

	ii.
	// Function to check whether a binary tree is a valid BinarySearchTree (BST)
	FUNCTION **isBST**(root):
	# Call helper function with full initial range
	return **isBSTUtil**(root, -∞, +∞)
	
	//Helper function that verifies BST property within a given range
	FUNCTION **isBSTUtil**(node, minVal, maxVal):
	# Base case: An empty tree is always a valid BST
	if node == NULL:
	return True
	
	// Check if current node violates the BST property
	//The node's value must be strictly greater than minVal and less than maxVal
	if (node->data <= minVal **OR** node->data >= maxVal:
	return False
	
	// Recursively check the left and right subtrees
	//Left subtree must have values < node.data
	//Right subtree must have values > node.data
	return isBSTUtil(node->left, minVal, node->data) **AND**
	isBSTUtil(node->right, node->data, maxVal)				

4.	a) Construct the binary expression tree for the following infix expression. Then, perform a **preorder traversal** on your constructed tree to derive the corresponding Prefix (Polish Notation) expression. **Infix Expression:** ((A * B) + (C / D)) ^ (E - F) **Solution:**  **Prefix expression: ^ + * A B / C D - E F** **Note: Detail solution is present at the last**	M04	5	4	

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

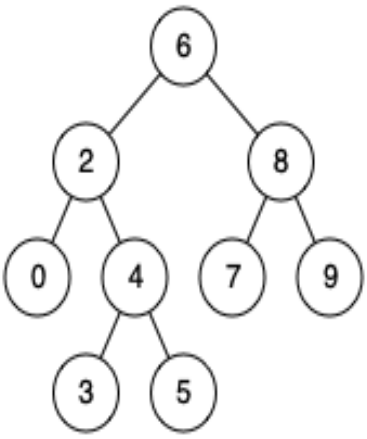
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
CONTINUOUS ASSESSMENT TEST - II
FALL SEMESTER 2025-2026

---

b) The **Lowest Common Ancestor (LCA)** between two nodes, **n1** and **n2**, in a tree is the lowest (i.e., deepest) node that has both **n1** and **n2** as descendants. Write an efficient **pseudocode** for a function **findLCA**(root, n1, n2) that finds the Lowest Common Ancestor of two given nodes in a **Binary Search Tree**. Your algorithm should leverage the properties of a BST to run faster than it would on a regular binary tree.

M04

5



LCA(2,8) = 6

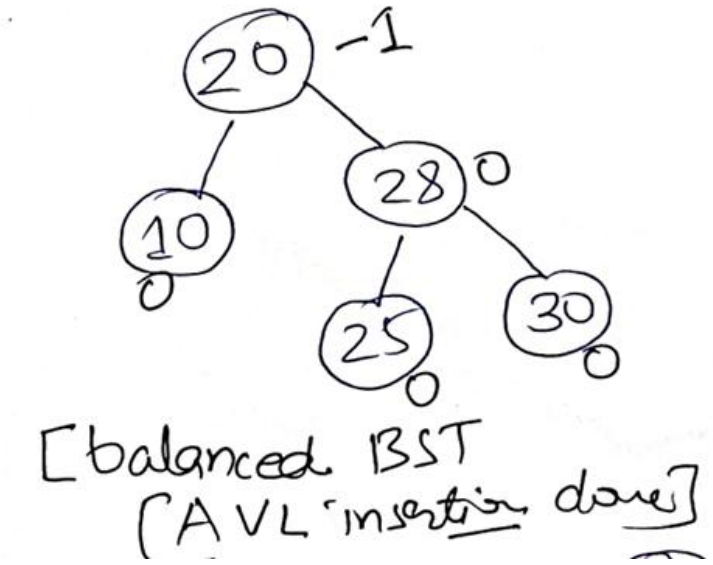
LCA(2,4) = 2

LCA(0,5) = 2

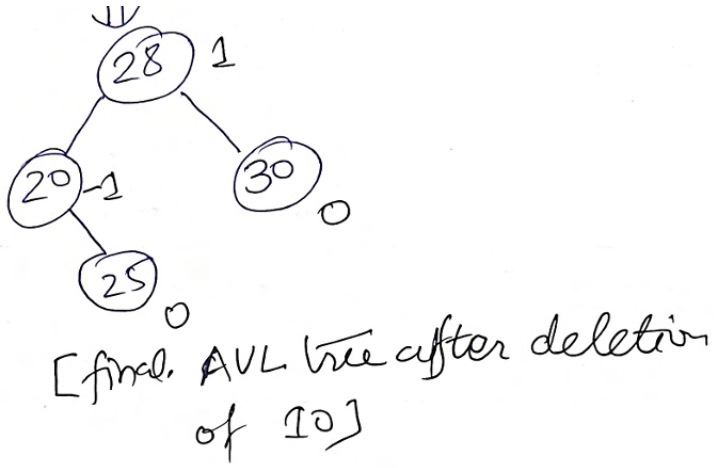
Solution:

```
FUNCTION searchBST(root, key):
  if root == NULL:
    return False
  if root->data == key:
    return True
  else if key < root->data:
    return searchBST(root->left, key)
  else:
    return searchBST(root->right, key)


FUNCTION findLCA(root, n1, n2):
  if root == NULL:
    return NULL

  // Check if both nodes exist
  If  !searchBST(root, n1) OR !searchBST(root, n2):
    return NULL

  while root != NULL:
    // Move left if both keys are smaller
    if n1 < root->data AND n2 < root->data:
      root = root->left
```

	// Move right if both keys are larger
	else if n1 > root->data AND n2 > root->data:
	root = root->right
	
	//Split point found → current node is LCA
	else:
	return root
	
	return NULL				

5.	Create a new AVL tree and insert the following keys in the given order: **Sequence:** 30, 20, 10, 25, 28. For each insertion, draw the resulting tree. If an insertion causes an imbalance, you must: **a)** Identify the unbalanced node. **b)** Name the type of imbalance (**LL, RR, LR, or RL**) and rotation required. **c)** Draw the final, rebalanced tree before proceeding to the next insertion. Take the final, balanced AVL tree you constructed after all insertions and delete the node 10. **Solution:** **Balanced AVL after all insertion**  	M07	10	5	4

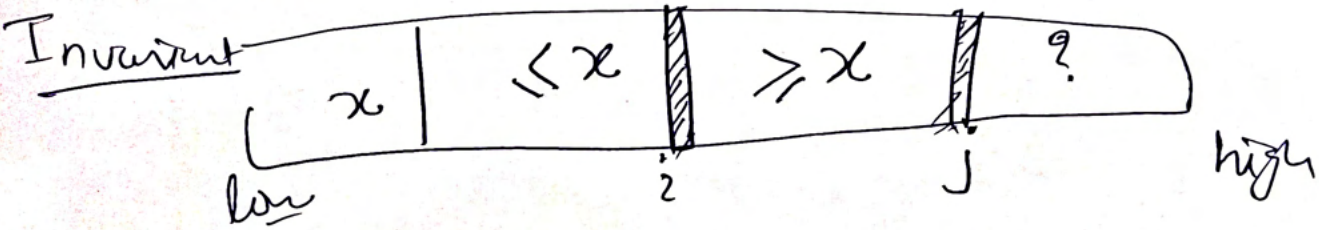
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
CONTINUOUS ASSESSMENT TEST - II
FALL SEMESTER 2025-2026

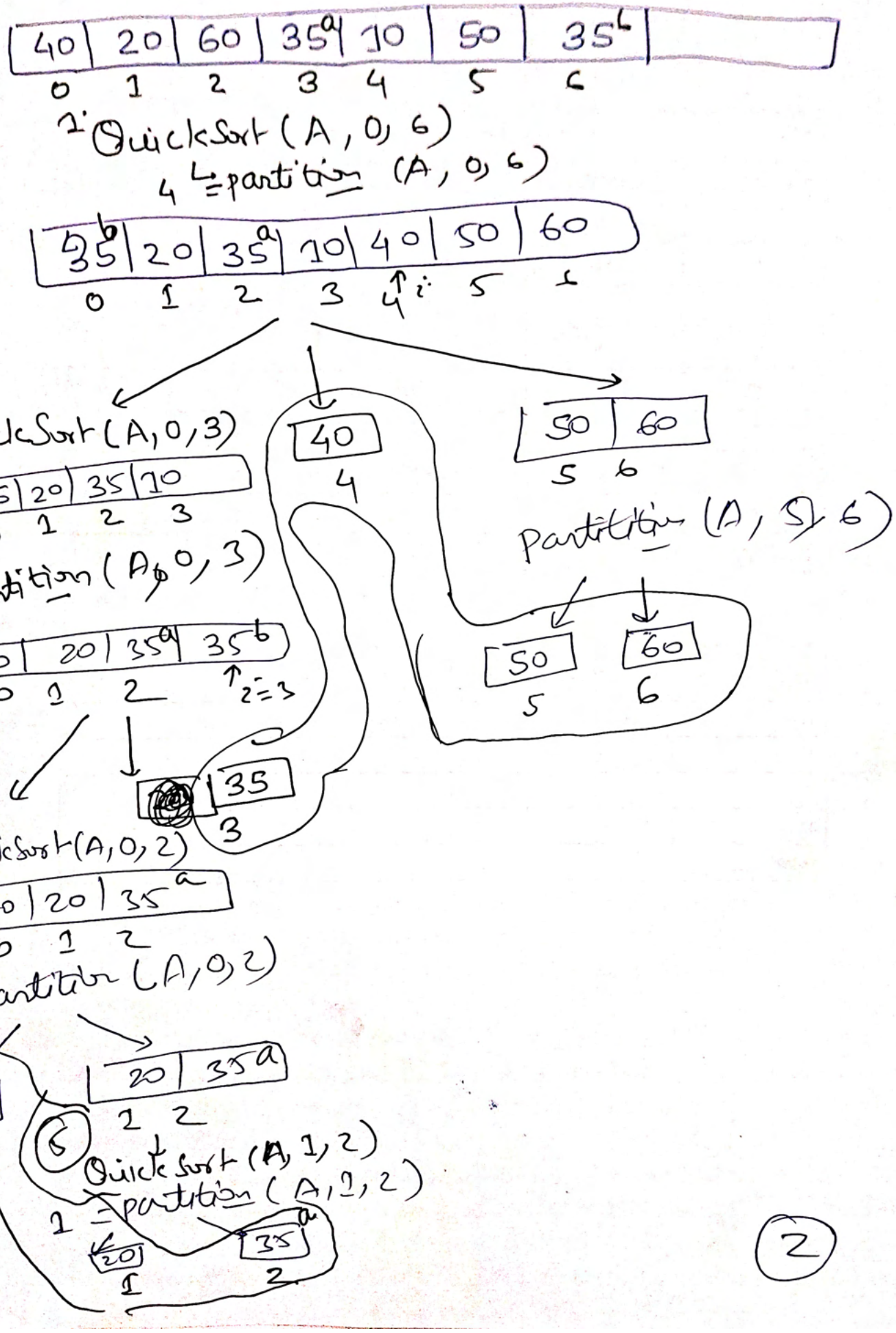
	Final AVL after deletion of node containing 10  [final AVL tree after deletion of 10] **Note: Detail solution is present at the last**				

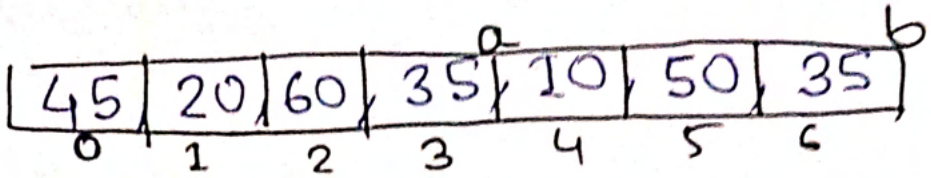
Quick Sort

Hoare Partition

```
QuickSort ( A, low, high)
    if (low > high)
        return
    i = partition ( A, low, high)
    QuickSort (A, low, i-1)
    QuickSort (A, i+1, high)

Partition ( A, low, high)
    x = A [low]  // pivot
    i = low
    for ( j = low; j <= high; j++)
        if (A[j] ≤ x)
            i = i+1
            SWAP( A[i], A[j] )
    SWAP( A [i], A [low] )
    return i
```



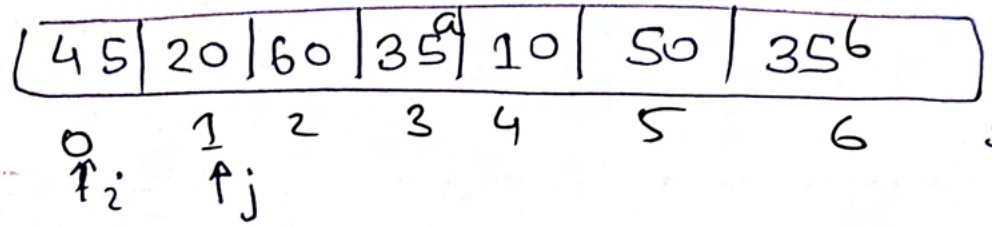

40 20 60 35a 10 50 35b
0 1 2 3 4 5 6
1: QuickSort (A, 0, 6)
4 ↳ partition (A, 0, 6)
35b 20 35a 10 40 50 60
0 1 2 3 4 i 5 6
2: QuickSort (A, 0, 3)
35 20 35 10
0 1 2 3
3 = partition (A, 0, 3)
10 20 35a 35b
0 1 2 i=3
40
4
50 60
5 6
partition (A, 5, 6)
50 60
5 6
35
3
5 QuickSort (A, 0, 2)
10 20 35a
0 1 2
0 = partition (A, 0, 2)
10
0
20 35a
1 2
6 QuickSort (A, 1, 2)
1 = partition (A, 1, 2)
20 35a
1 2



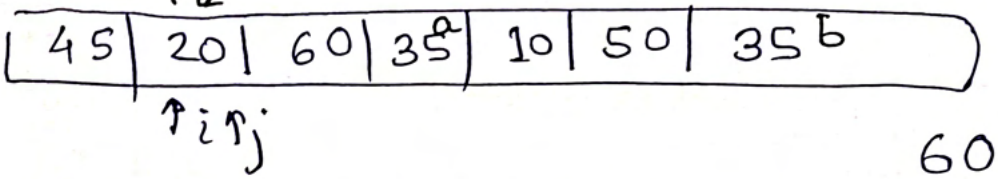
(1) QuickSort (A, 0, 6)

partition (A, 0, 6)

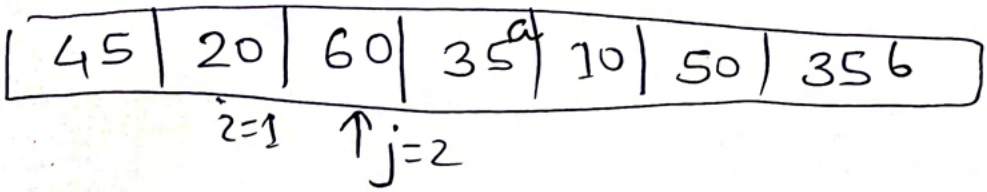
x = 45 // pivot



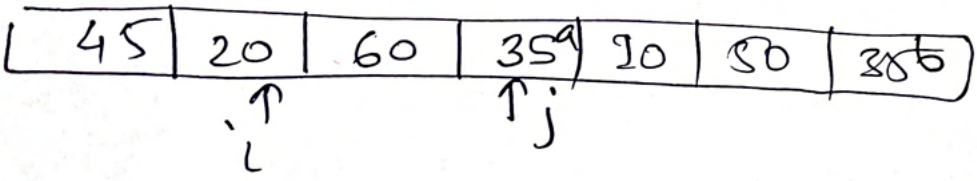
20 ≤ 45
i = 0; j = 1
i++
SWAP(A[i], A[j])



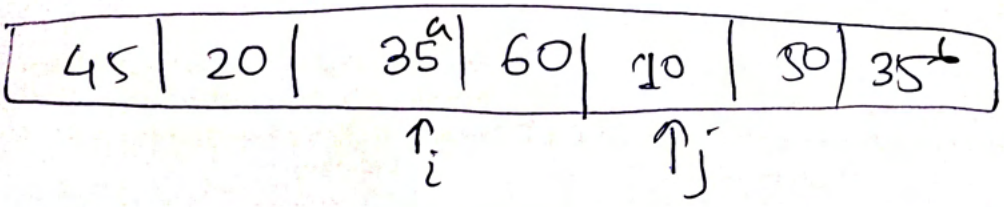
60 ✗ 45



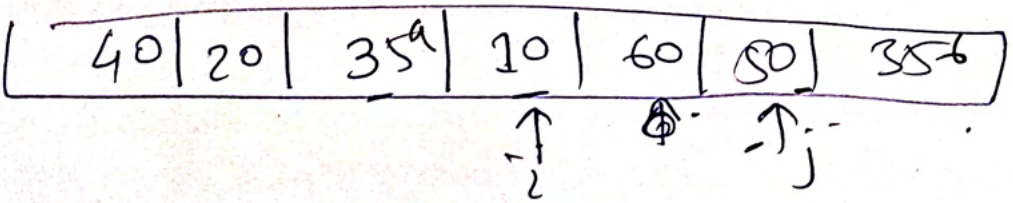
35 ≤ 45
i++
SWAP(A[i], A[j])



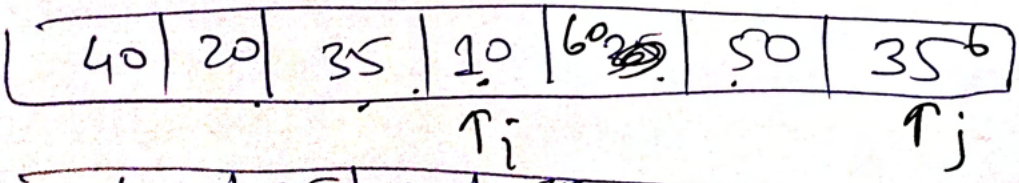
10 ≤ 45

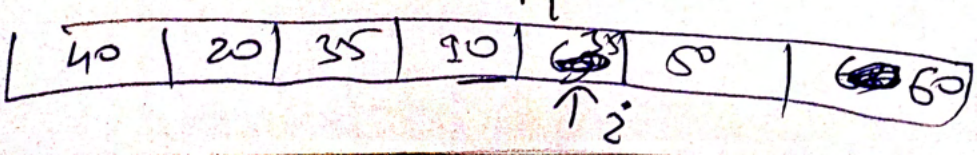


50 ✗ 40

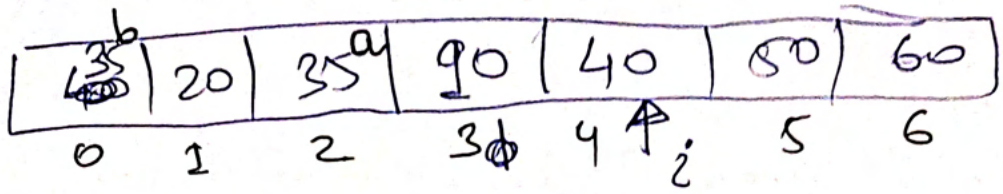


35 ≤ 45

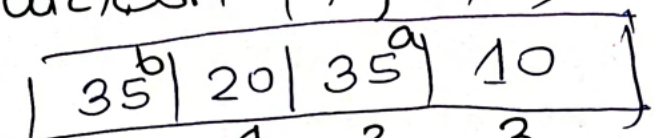




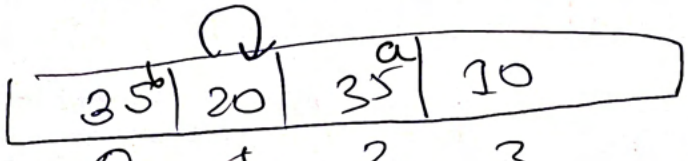
(3)



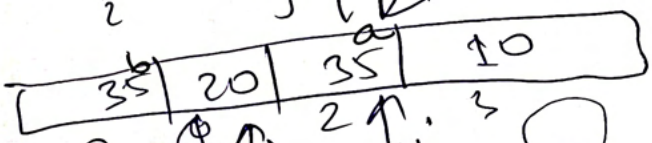
(2) QuickSort (A, 0, 3)



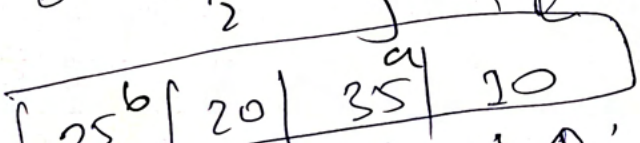
x = 35 // pivot



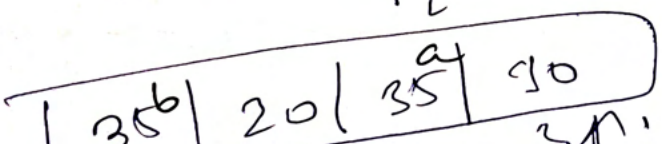
20 ≤ 35
i++
swap(A[i], A[j])



35 ≤ 35
i++
swap(A[i], A[j])

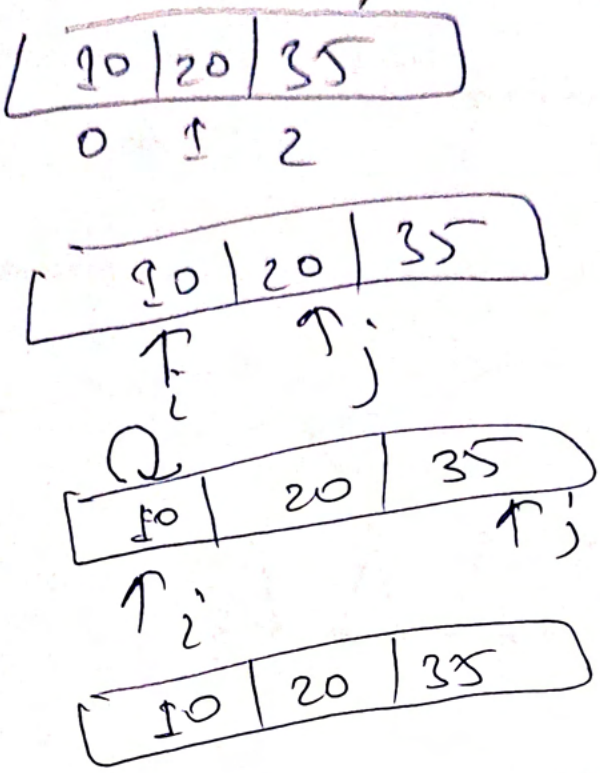


10 ≤ 35
i++
swap(A[i], A[j])

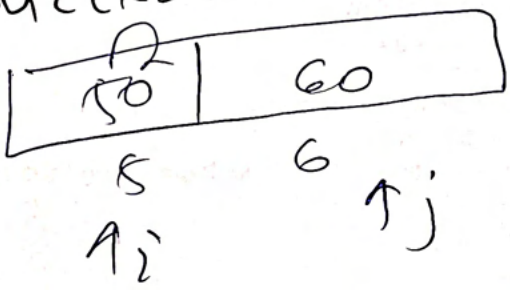


swap(A[i], A[hi])
return i

(4) Quicksort (A, 0, 2)



(5) Quicksort (A, 1, 2)



(Q3) a) solution

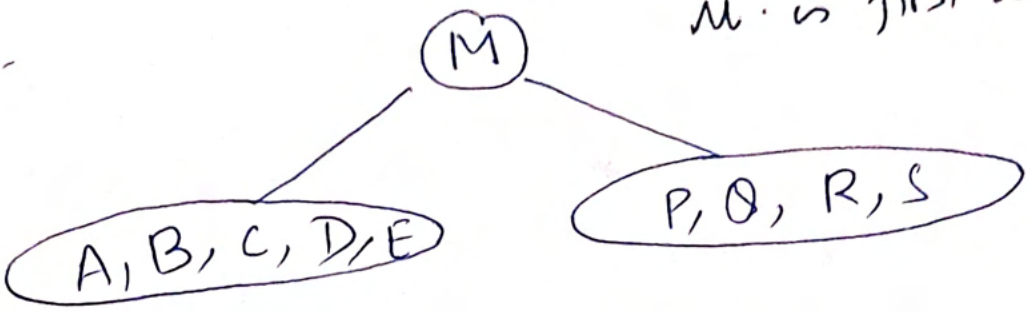
3

a) Preorder: M, B, A, D, C, E, P, Q, S, R

Inorder: A, B, C, D, E, M, P, Q, R, S

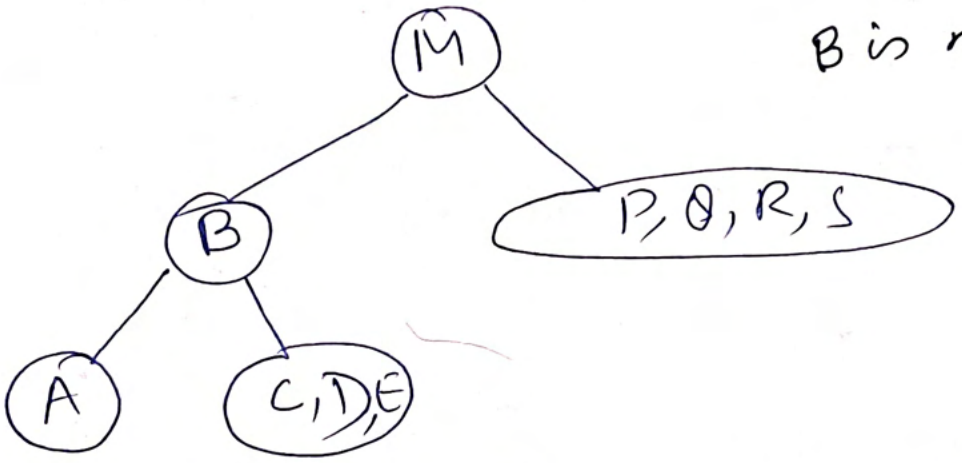
M is first root (Preorder)

Step1 :-



Step2:

B is next root



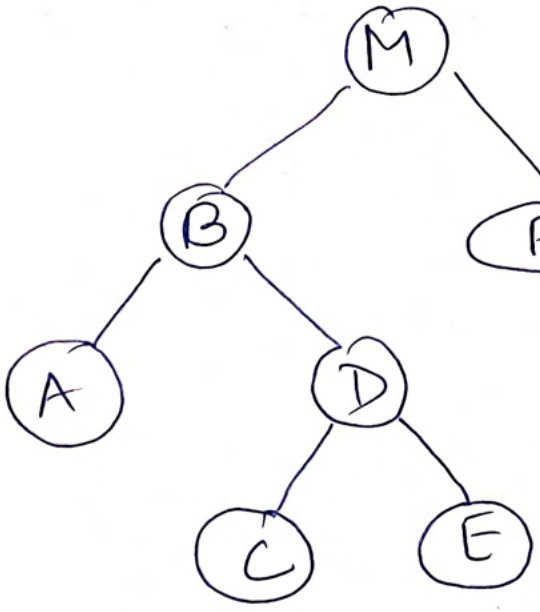
Step3 :-

A is next root.

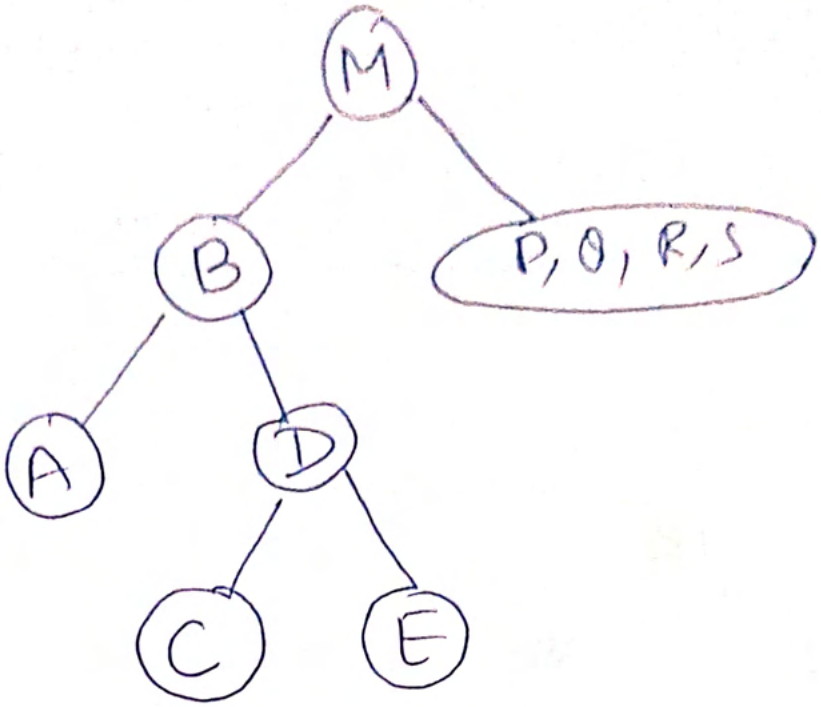
D is next root.

C is next root.

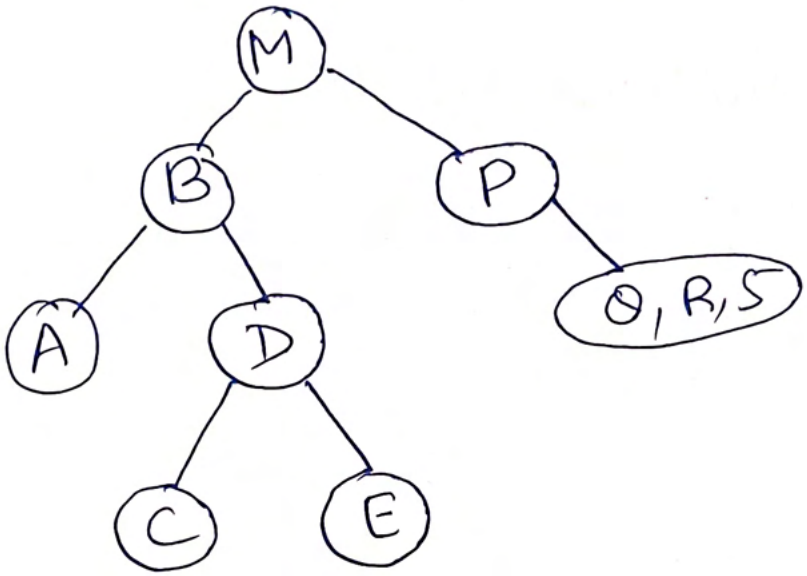
E is next root.



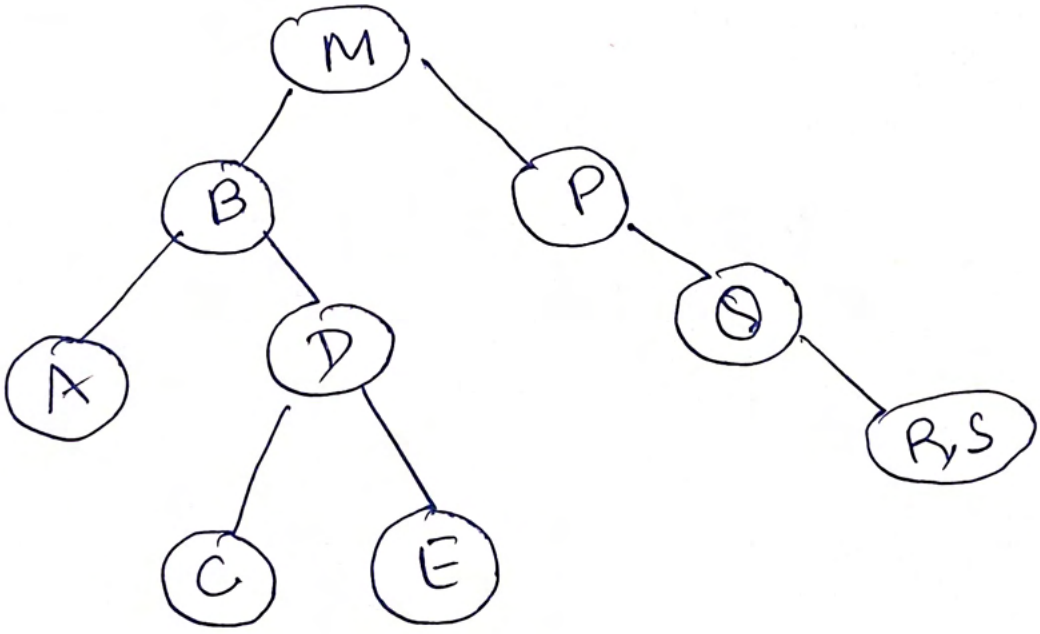
Step 4 :-



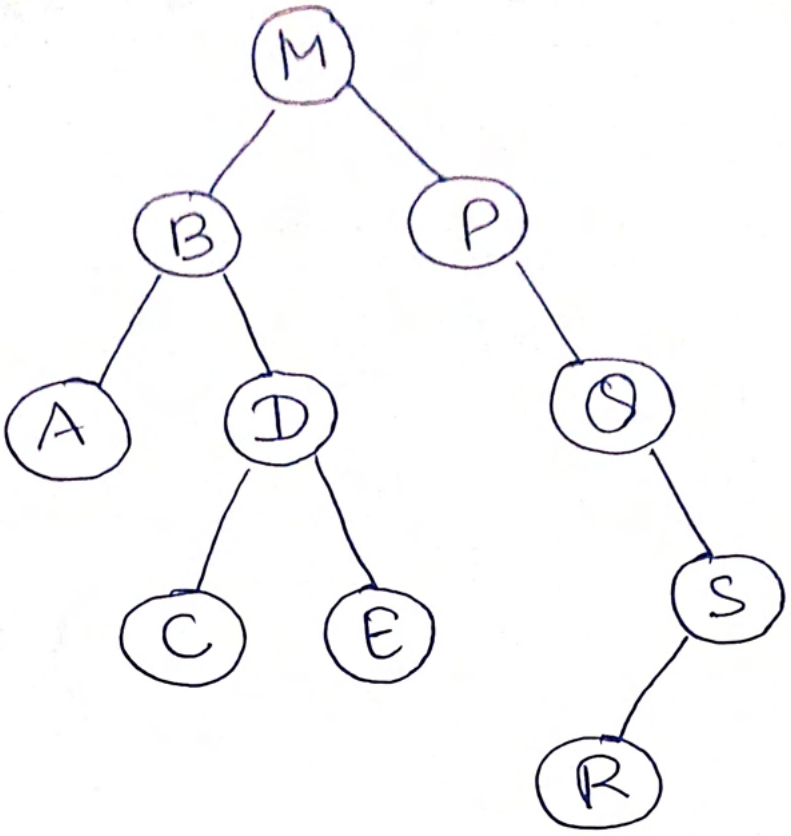
Step 5 :-



Step 6 :-



Step 7:-



[final tree]

Q4 a) solution

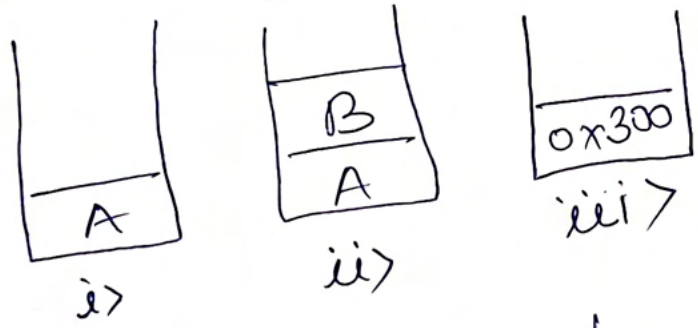
Infix Expression : $((A*B)+(C/D))\wedge(E-F)$

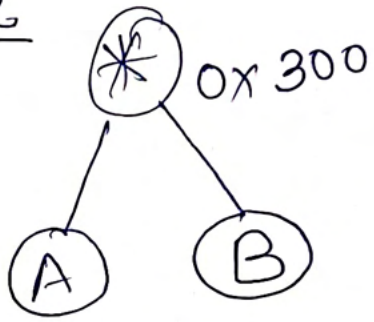
postfix Expression: $(AB*\ +\ CD/)\ \wedge\ EF-$

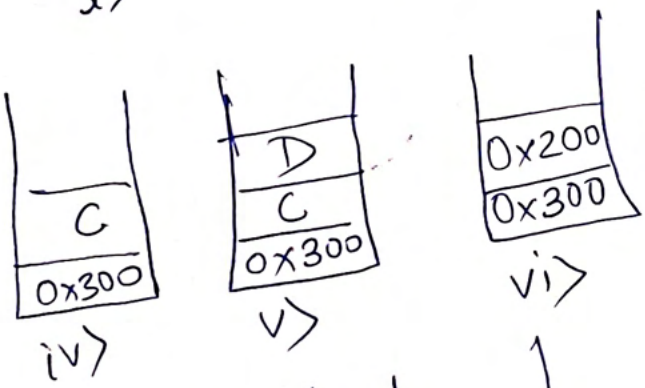
$AB*CD/+\ \wedge\ EF-$

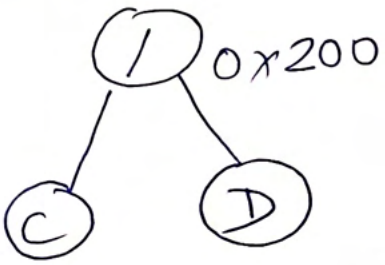
$AB*CD/+EF-\wedge$

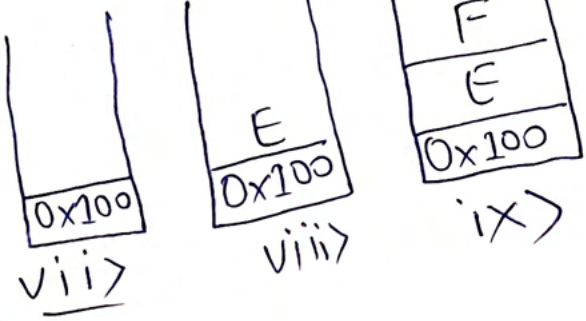
creation of Expression Tree

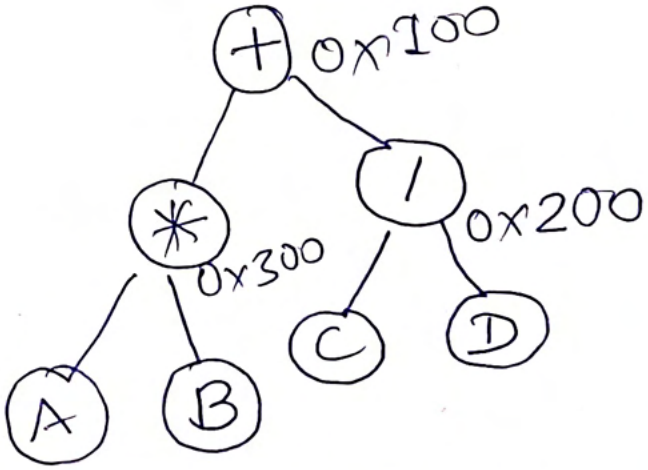


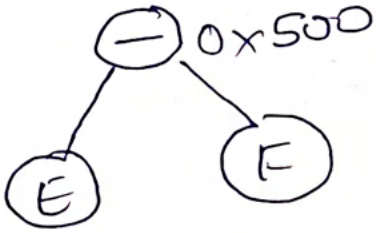




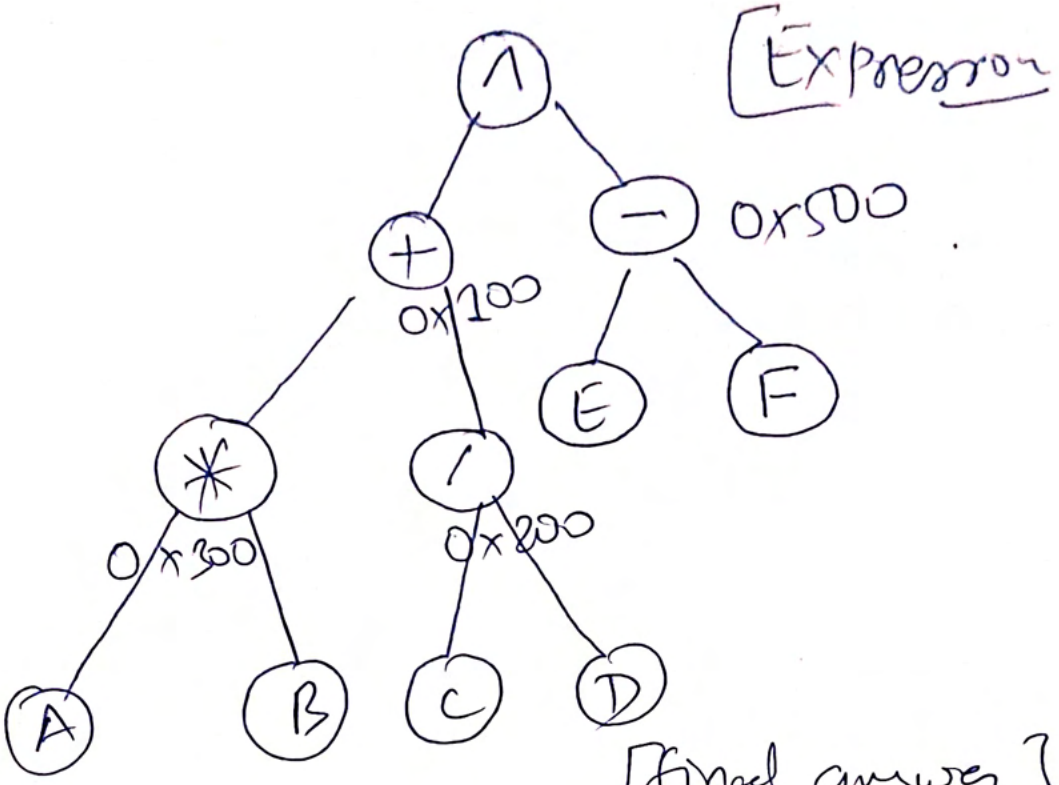








[Expression Tree]



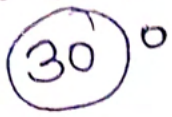
[final answer]

preorder traversal :

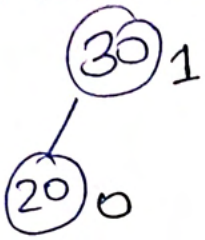
^ + * A B / C D - E F

Q5 solution

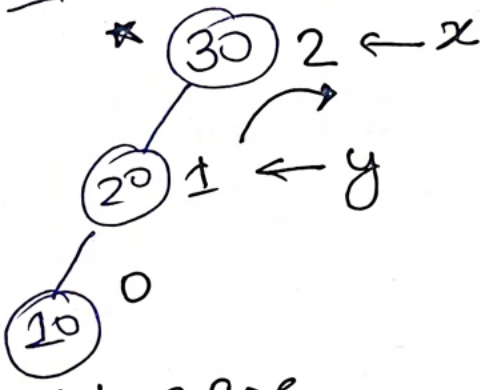
i) Insert 30



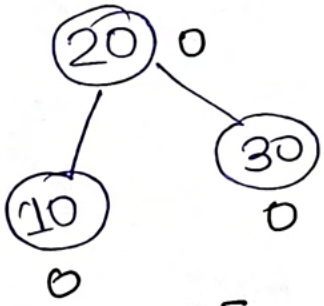
ii) Insert 20



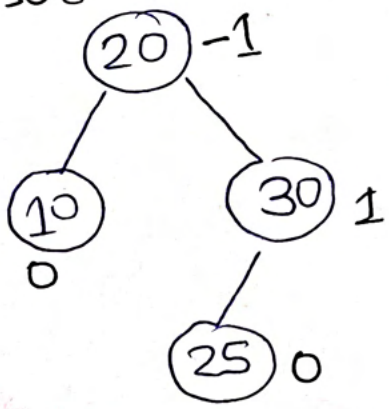
iii) Insert 10



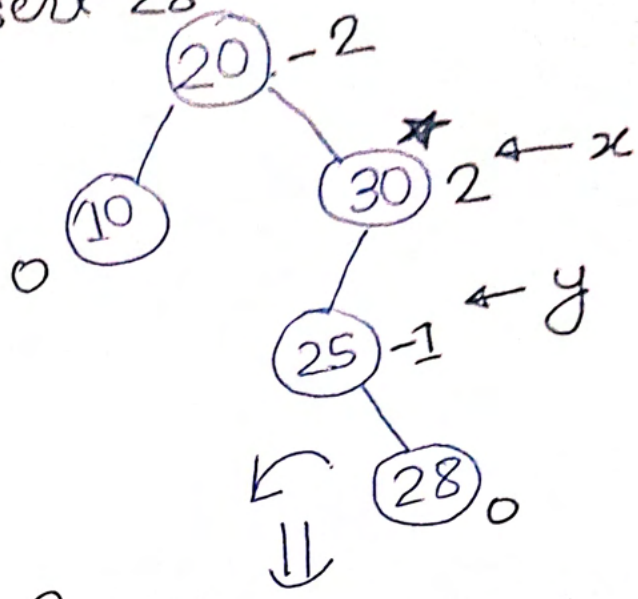
LL case
↳ rotate right at x



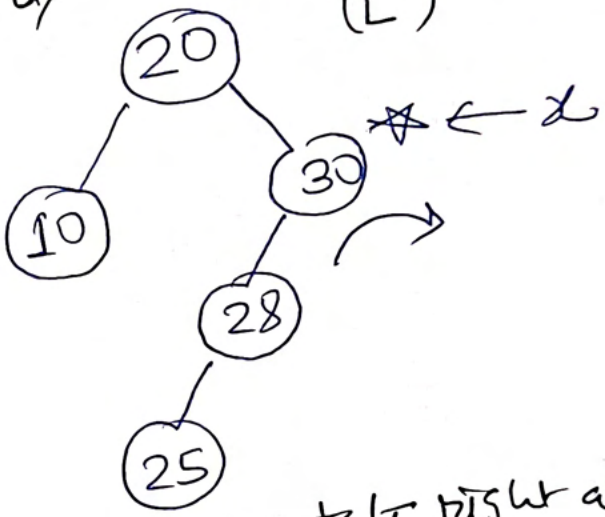
iv) Insert 25



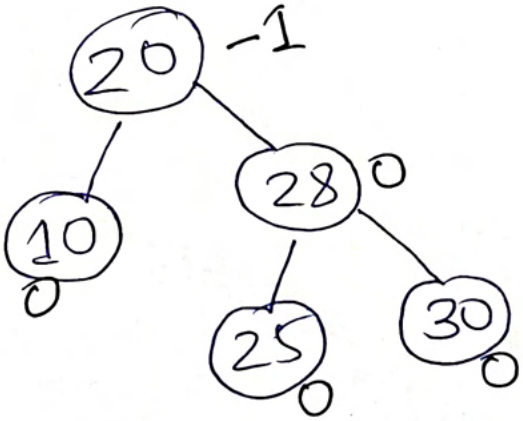
v) Insert 28



LR case
a) rotate left at y (L)

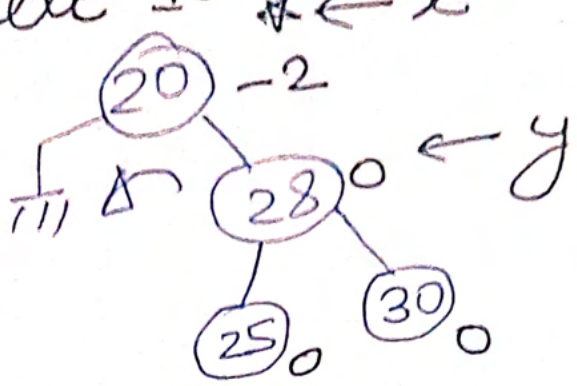


b) rotate right at x (R)



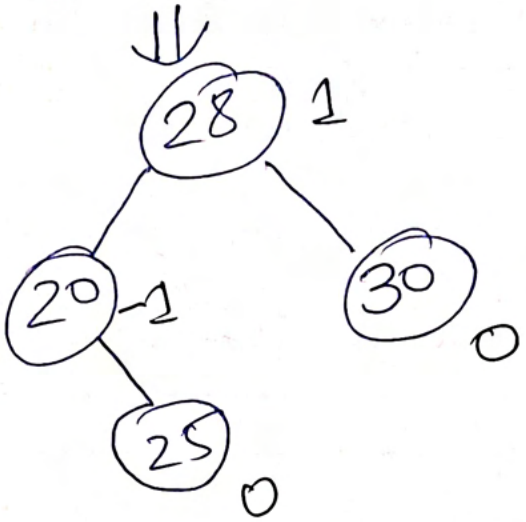
[balanced BST
[AVL insertion done]

VI) Delete 10



[R R case]

rotate left at x



[final AVL tree after deletion of 10]