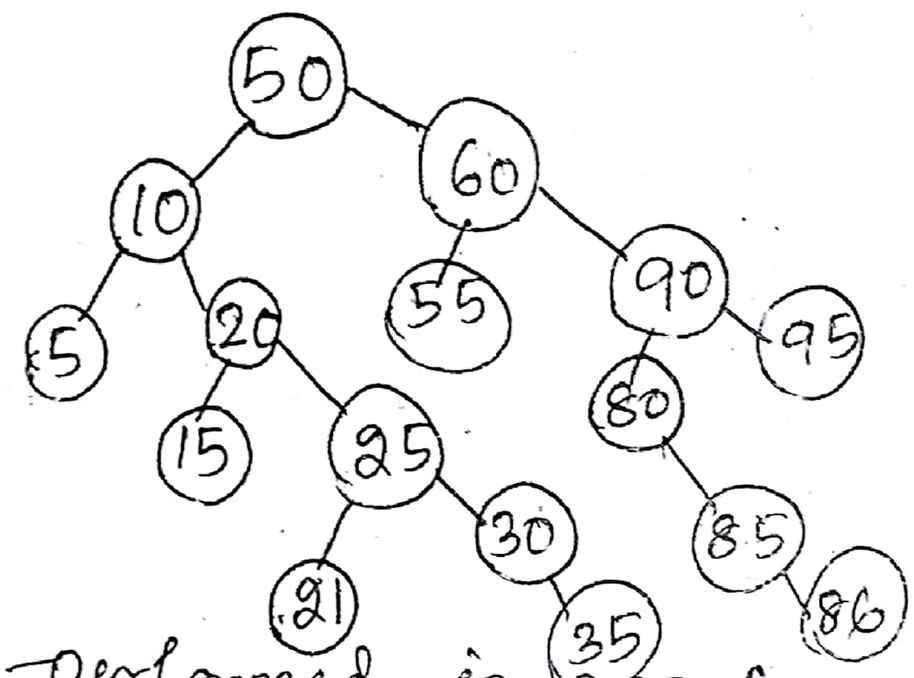
Binary Search tree

A binary search tree is a binary tree that may be empty and if it is not empty then it satisfy following properties.

1) All keys (values present at node) in the left subtree of root are less than the key in the root.
2) All the keys in the right subtree of root are greater than the key in the root.
3) Left & right subtree of root are also binary search tree.

Construction of binary Search tree :-

50, 10, 20, 5, 15, 25, 30, 60, 55, 90, 95, 80, 21, 85, 86, 35



Operations Performed in BST :-

(1) Traversing :-

Preorder : 50 10 5 20 15 25 21 30 35 60 55 90 80 85 86 95

inorder : 5 10 15 20 21 25 30 35 50 55 60 80 85 86 90 95

postorder : 5 15 21 35 30 25 20 10 55 86 85 80 95 90 60 50

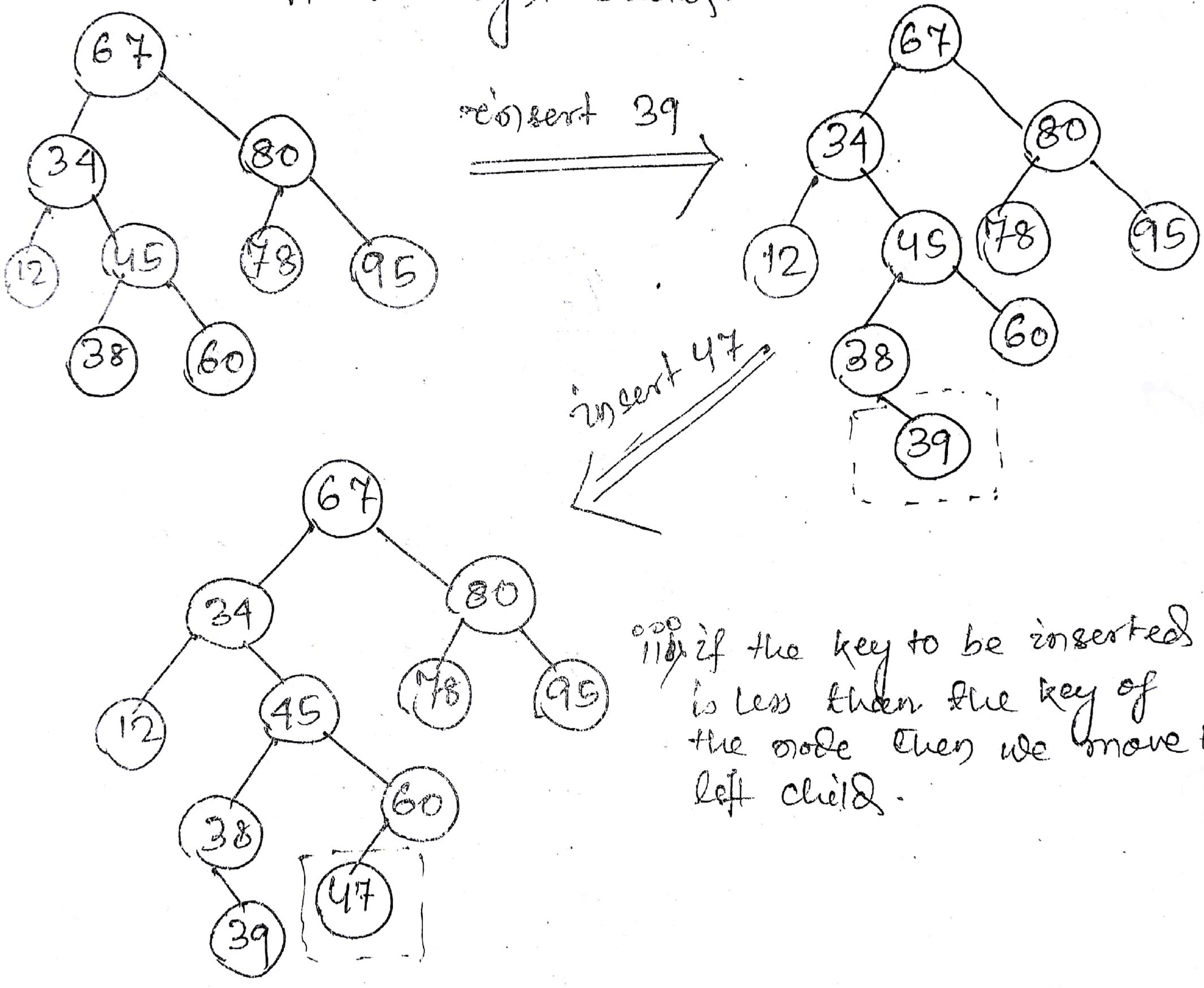
(2) Searching :-

```
void search (node *ptr, int skey);
{
while(ptr != NULL)
  {
  if (skey == ptr -> info)
    printf("element found");
  else if (skey < ptr -> info)
      ptr = ptr -> lchild;
  else ptr = ptr -> rchild;
  }
}
```

③ Insertion: To insert any node in to binary search tree, then initially data item that is to be inserted is ~~equal to the key~~ compared with the data of the root node.

i) if the key to be inserted is equal to the key in the node then there is nothing to be done as duplicate keys are not allowed.

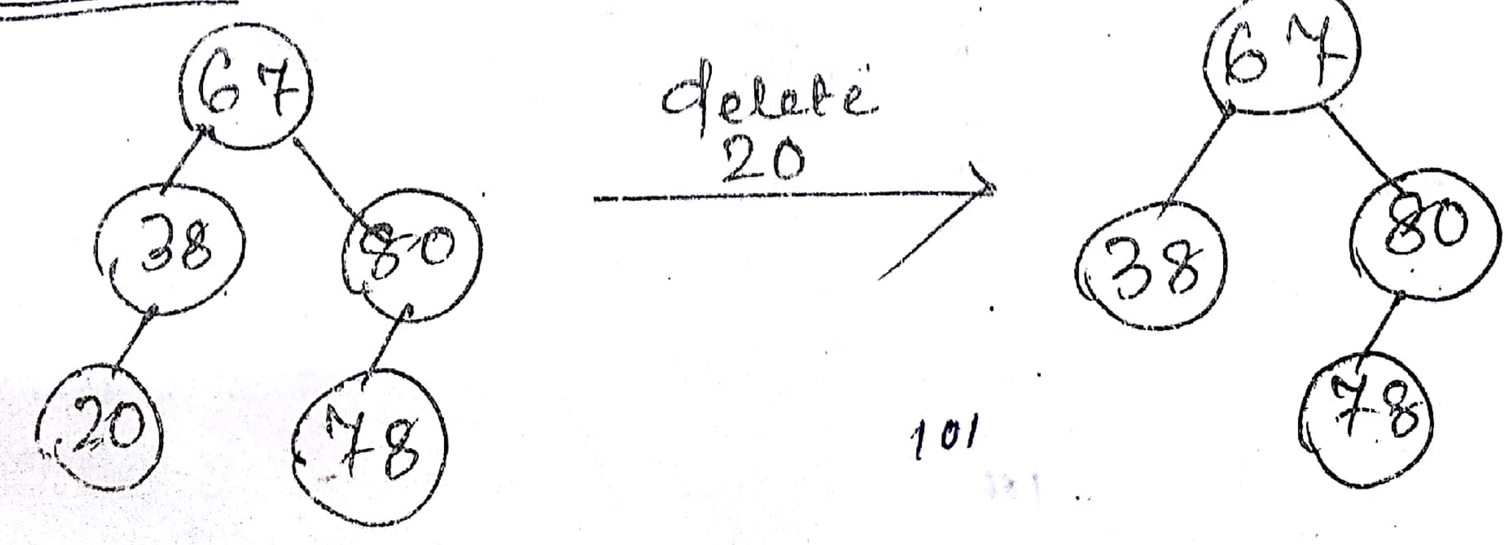
ii) if the key to be inserted is ~~less~~ greater than the key of the node then we move to right child. We insert the new key when we reach a NULL left or right child.



ii) if the key to be inserted is less than the key of the node then we move to left child.

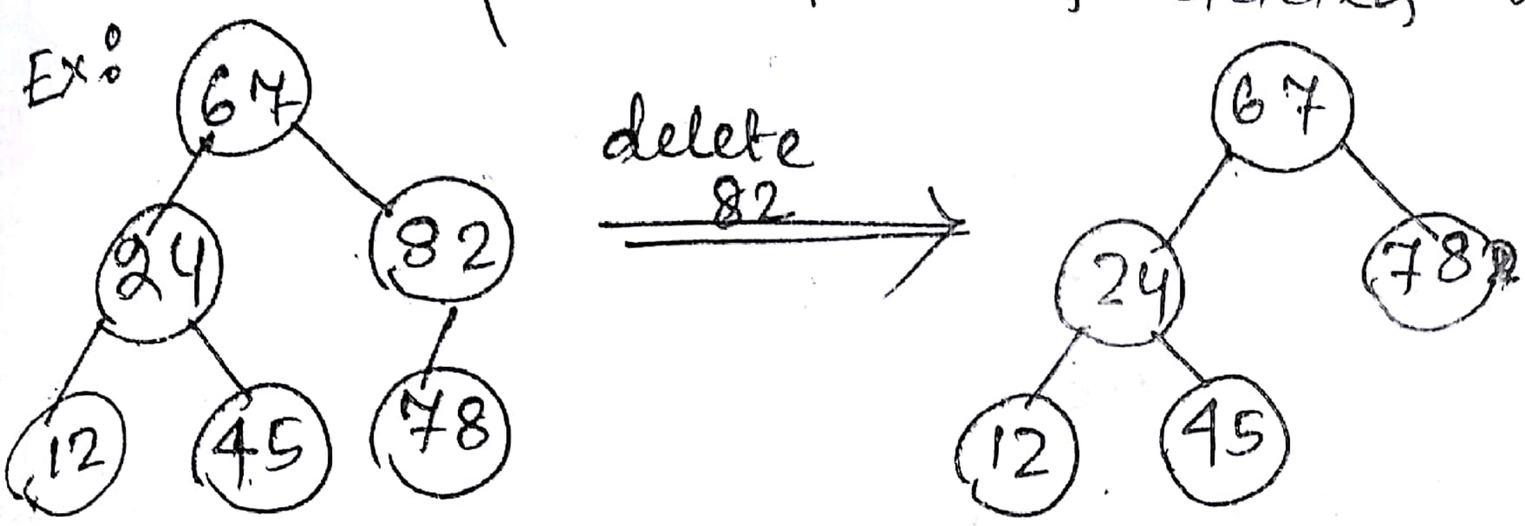
4) Deletion:- There are 3 cases.

case 1: To delete a leaf node

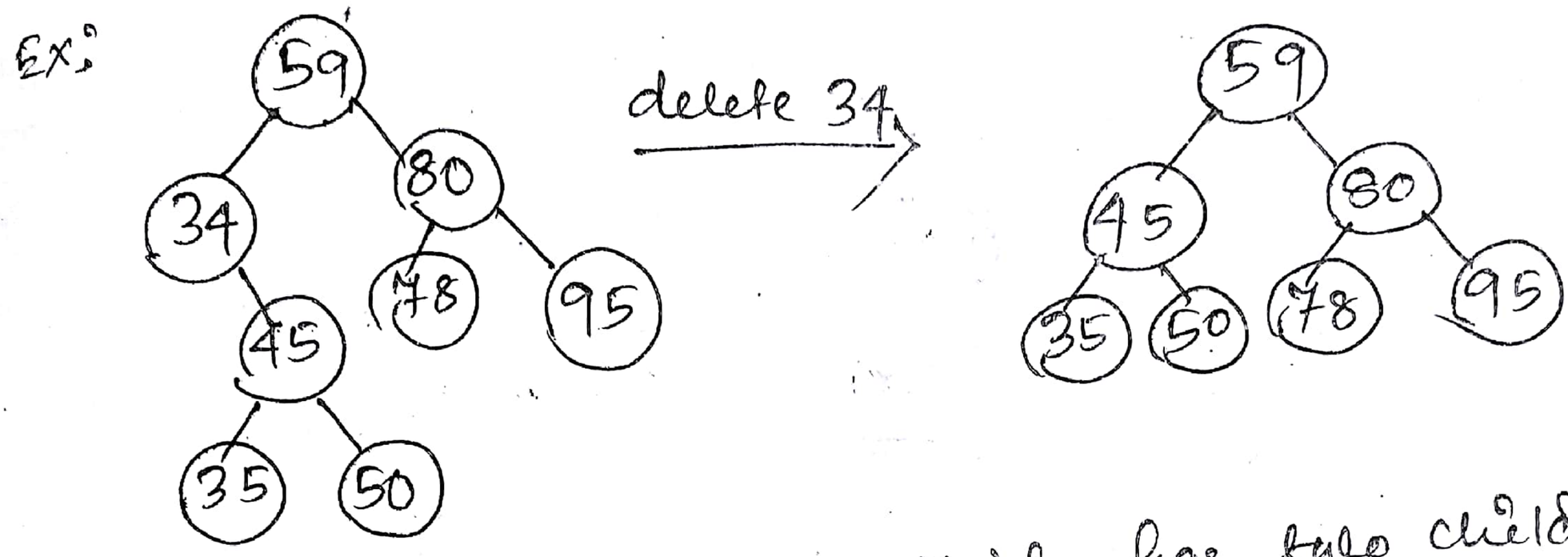


Case 2: To delete a node which has only one child. After deletion of that node the child of that node takes the place of deleted node.

Ex:

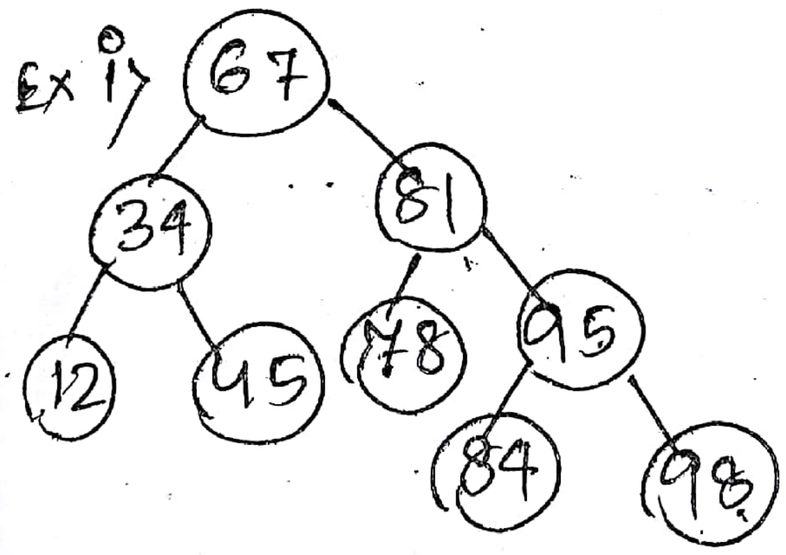


Ex:



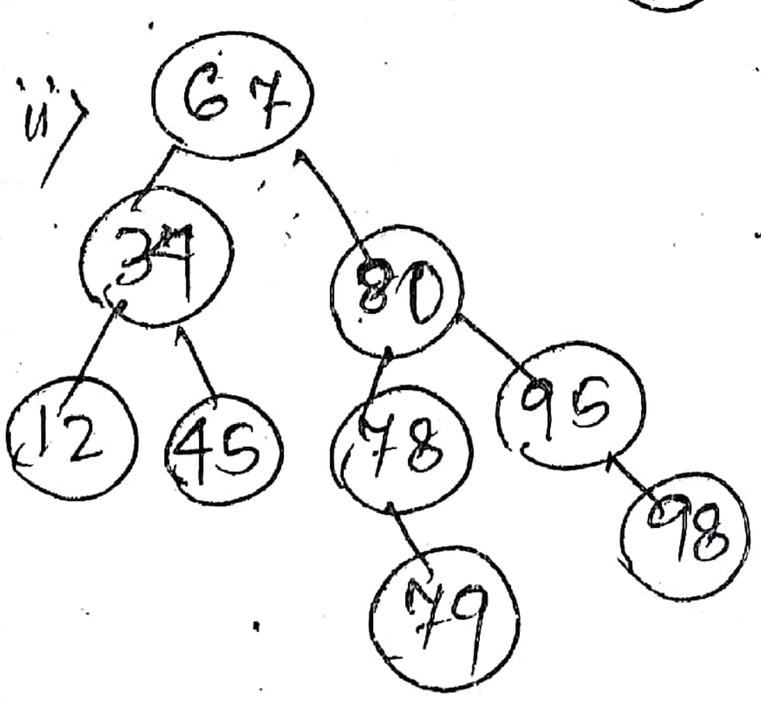
Case 3: To delete a node which has two children
→ Here we have to find the inorder successor of that node, then the inorder successor takes the place of deleted node & arrange all the nodes attached to the nodes according to BST rules.

Ex i)



inorder: 12, 34, 45, 67, 78, 81, 84, 95, 98

ii)



inorder: 12, 34, 45, 67, 78, 79, 80, 95, 98

Height Balanced Tree :- (AVL)

It was described in 1962 by two Russian Mathmetics G. M. ADE'SON-VEL'SKII & E. M. LANDIS.

> A height balanced tree (AVL) is a binary tree in which the difference in heights between the left & right subtree is not more than one for every node.

> The height of a tree is the no. of nodes in the longest path from the root to any leaf.

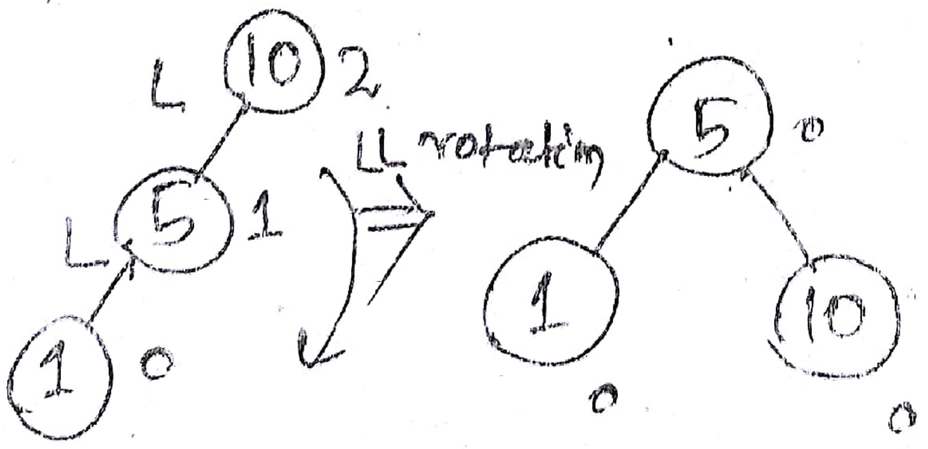
> While insertion of any node to the tree we have to find out the balancing factor which is the differen between the left height & right height or difference bet no. of node in left subtree & longest path and no. nodes in right subtre
Insertion with an AVL tree :- in longest path.

→ Insert a new node into an AVL tree by first using the usual binary search tree.

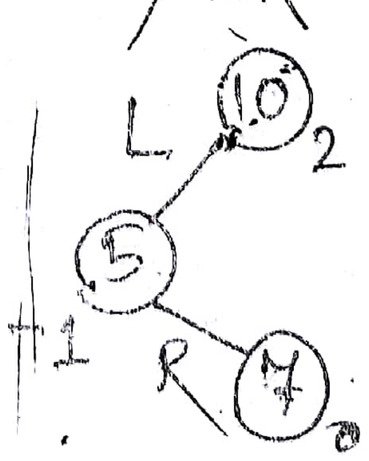
⇒ Comparing the key of new node with that on root and inserting the new node into either left or right subtree.

⇒ Check the balancing factor of each and every node if the balancing factor value is −1, 0, 1 then there is not any modification required if it is other than −1, 0, or 1 then do the following rotation to balance the tree.

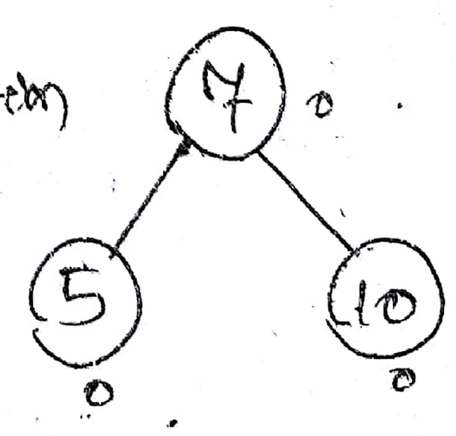
i) LL rotation :-



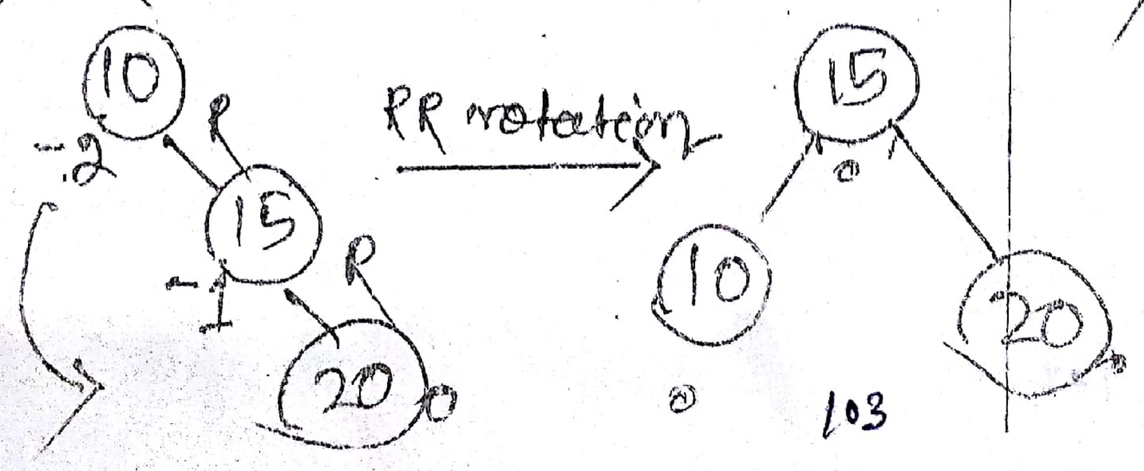
ii) LR rotation :-



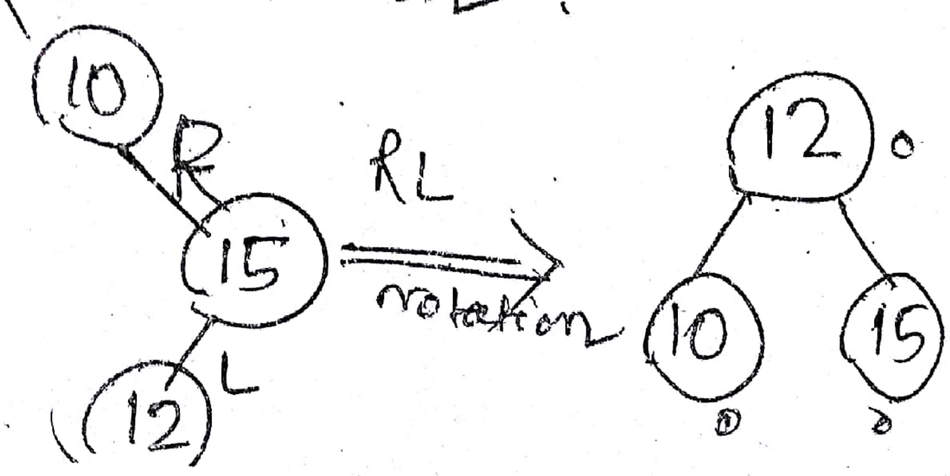
LR rotation ⟹



iii) RR rotation →



iv) RL rotation :-



Q)

Q) construct an AVL from the following sequence

55 66 77 15 11 33 22 35 25 44 ~~88~~ 99

i) 55 (0)

ii) (insert 55)

55 (-1)
66 (0)

iii) (insert 77)

55 (-2) R
66 (-1) R
77 (0)

RR rotation

66 (0)
55 (0)
77 (0)

iv) insert 15

66 (1)
55 (1)
77 (0)
15 (0)

v) insert 11

66 (1)
55 (2)
77 (0)
15 (1)
11 (0)

LL rotation

66 (1)
15 (0)
55 (0)
77 (0)
11 (0)

vi) insert 22

55 (1)
15 (-1)
66 (-1)
11 (0)
33 (1)
77 (0)
22 (0)

vii) insert 35

55 (1)
15 (-1)
66 (-1)
11 (0)
33 (0)
77 (0)
22 (0)
35 (0)

viii) insert 25

55
15 (-2)
66 (-1)
11 (0)
33 (1)
77 (0)
22 (-1)
35 (0)
25 (0)

RL rotation

55 (0)
22 (0)
66 (-1)
15 (1)
33 (0)
77 (0)
11 (0)
25 (0)
35 (0)

ix) insert 44

55 (2)
22 (-1) R
66 (-1)
15 (1)
33 (1)
77 (0)
11 (0)
25 (0)
35 (-1)
44 (0)

LR rotation

33 (0)
22 (1)
55 (0)
15 (1)
25 (0)
35 (-1)
66 (-1)
11 (0)
44 (0)
77 (0)

x) 33 insert

66 (2)
15 (-1) R
77 (0)
11 (0)
55 (1)
33 (0)

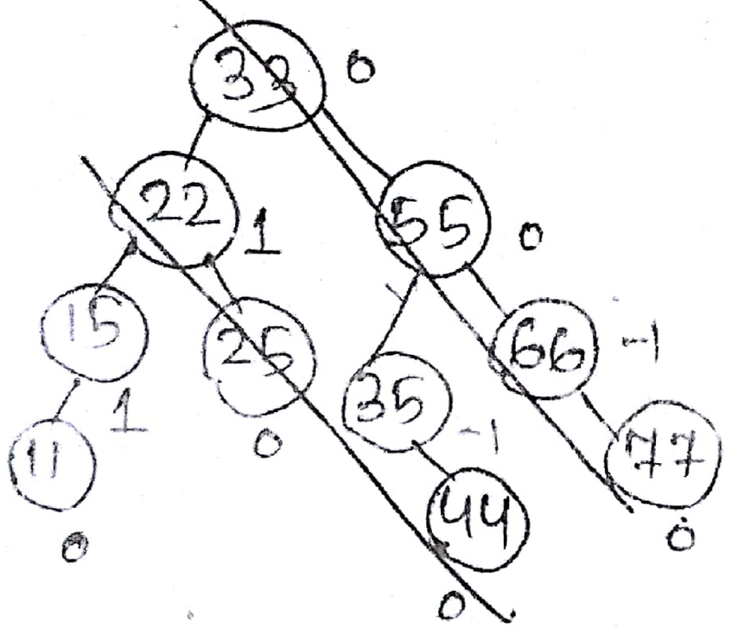
LR rotation

55 (0)
15 (0)
66 (-1)
11 (0)
33 (0)
77 (0)

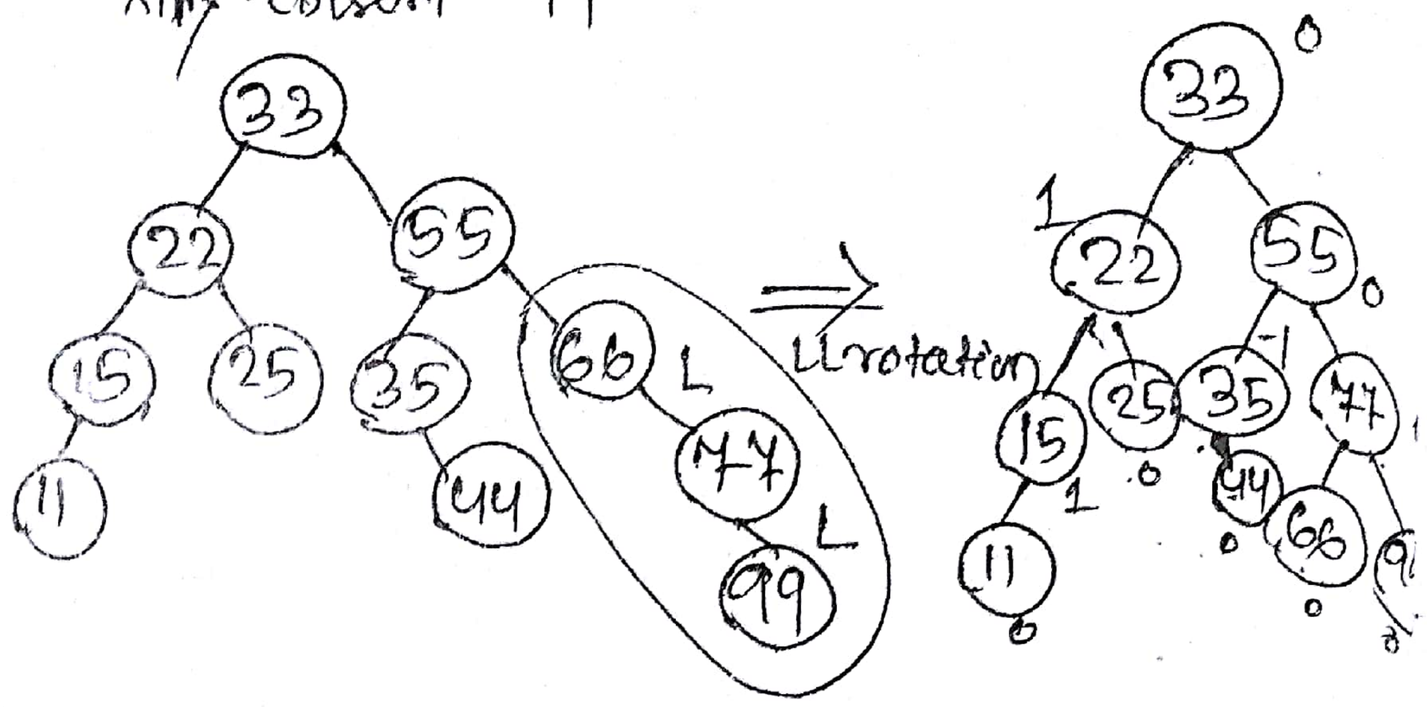
xi) 99 insert

55 (0)
15 (0)
66 (1)
11 (0)
33 (0)
77 (1)
99 (0)

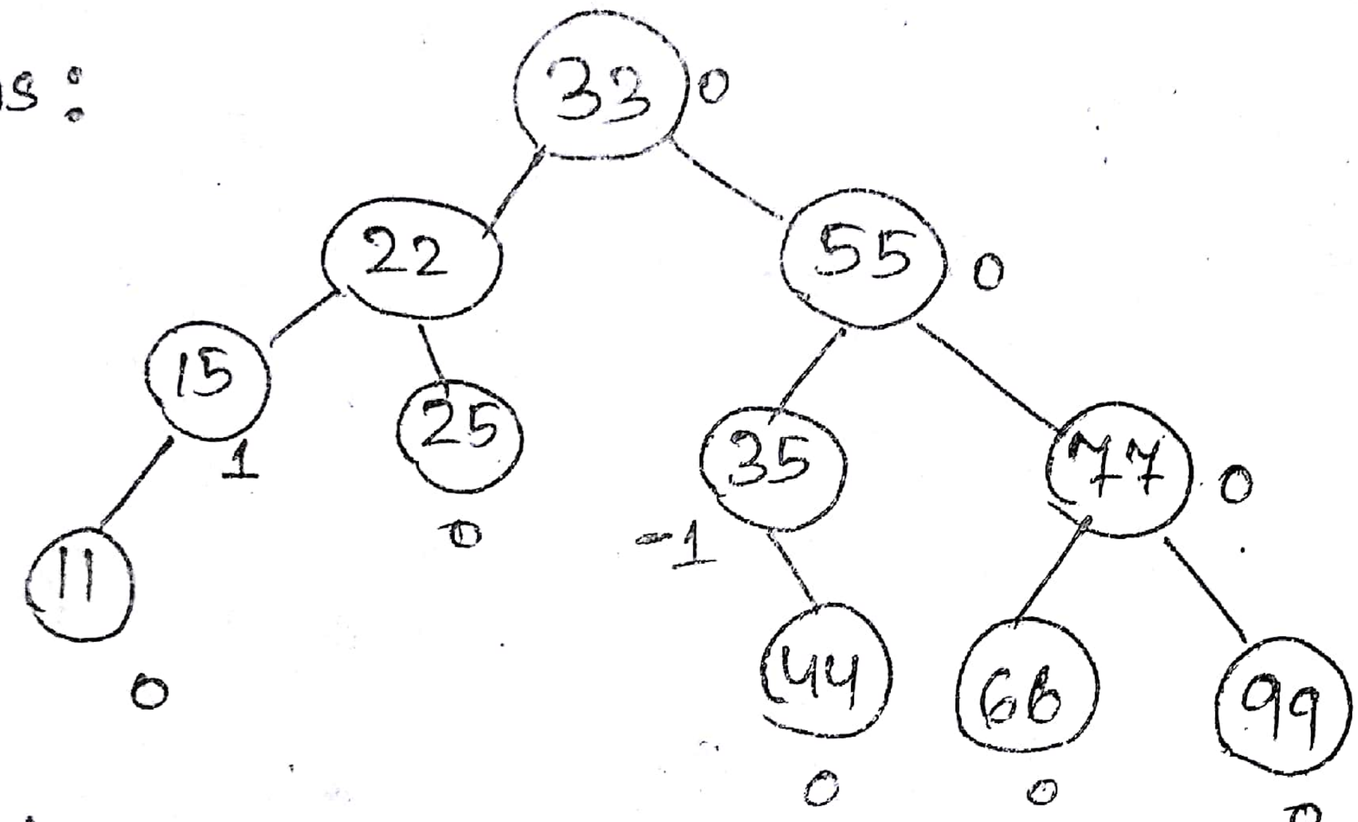
x) Insert 88



xi) Insert 99

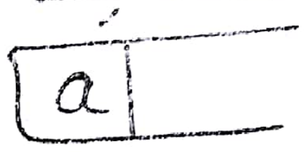
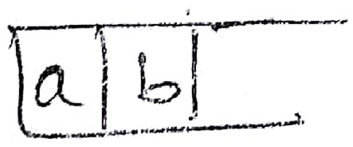
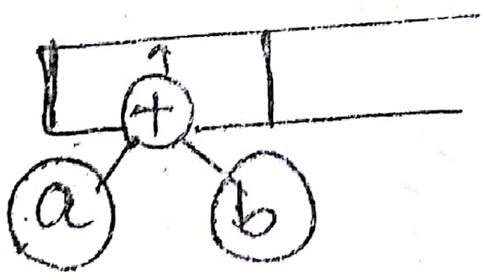
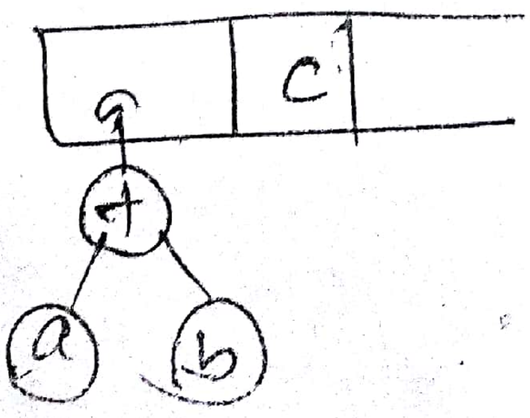


Ans:

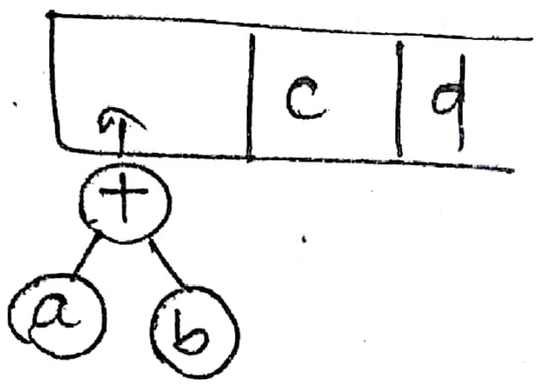


Construction of an expression tree from a postfix / postorder expression —:

ab+cde+** scan from left to right.

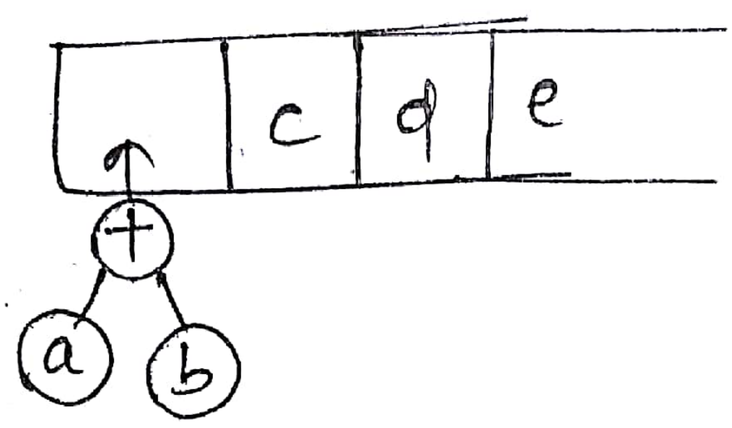
Symbol scanned	Stack	operation
a		push(a)
b		push(b)
+		when any operator arises pop topmost 2 element & construct tree i.e. topmost element will be right child & 2nd topmost element will be left child & push tree again to stack.
c		push(c)
e		

d



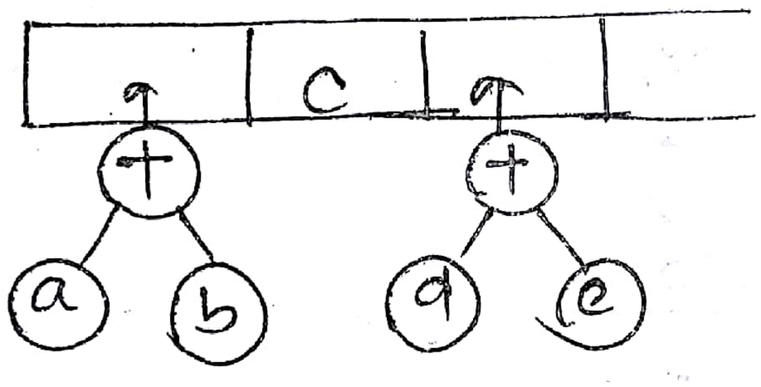
push(d)

e



push(e)

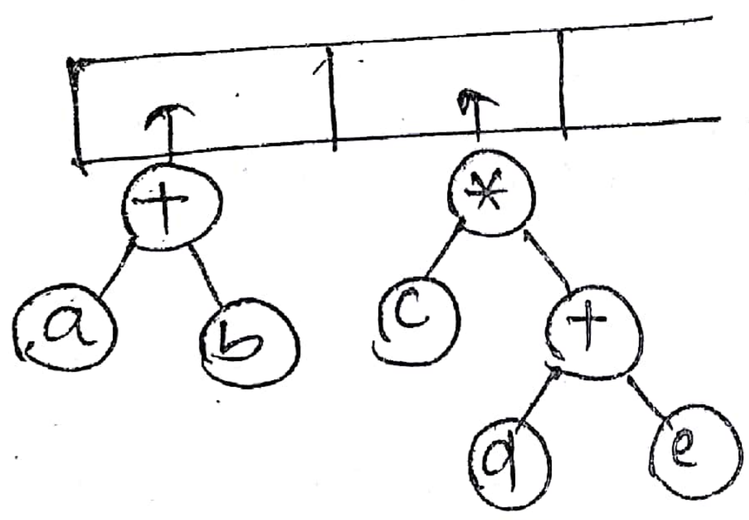
+



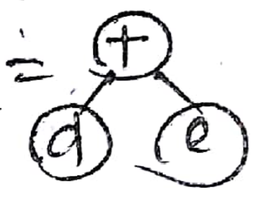
pop(e) = e

pop(d) = d
construct tree & push tree to stack

*

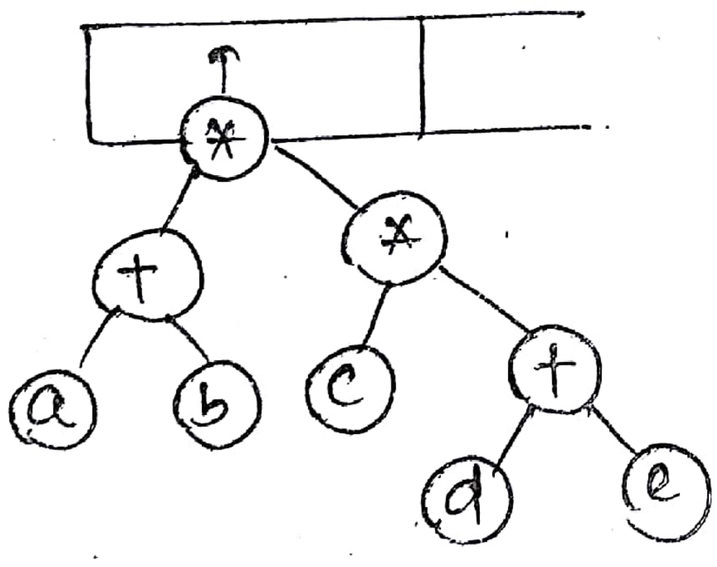


pop() =

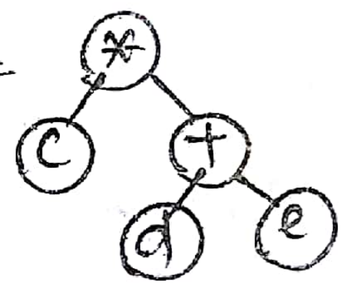


pop() = c
construct tree & push tree to stack

*



pop() =

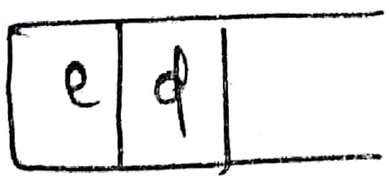


pop() =

construct tree & push tree to stack

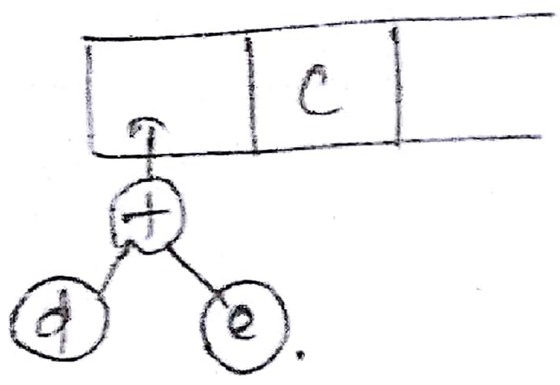
Construction of an expression tree from a prefix / preorder expression —

* + ab * c + de ⇒ scan from right to left

symbol scanned	stack	operation
e		push(e)
d		push(d)
+		pop() → d pop() → e construct tree topmost element will be left child & 2nd topmost element will

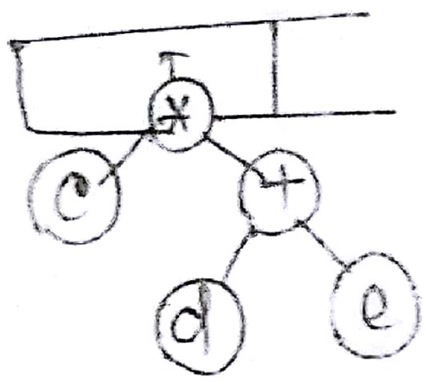
be right child,
push tree to stack

c

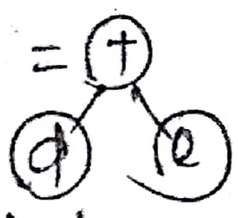


push(c)

*

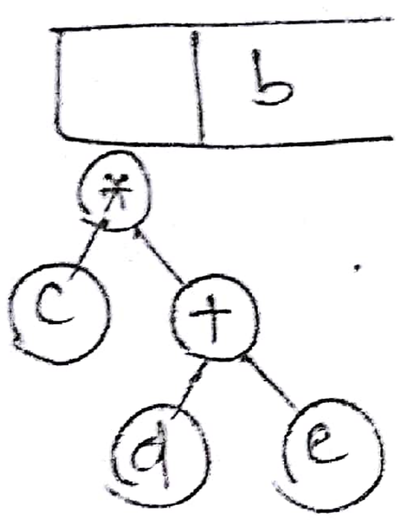


pop() = c

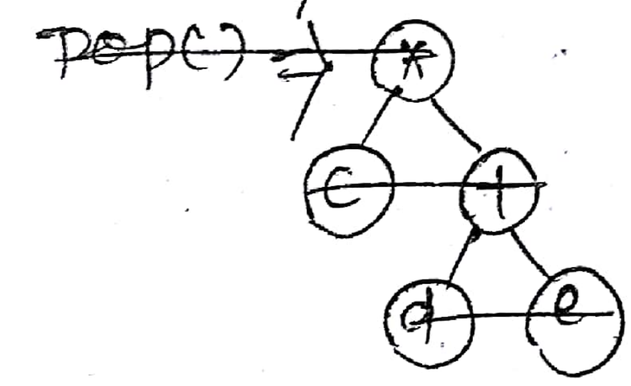
pop() = 

> construct tree
> push tree to stack

b



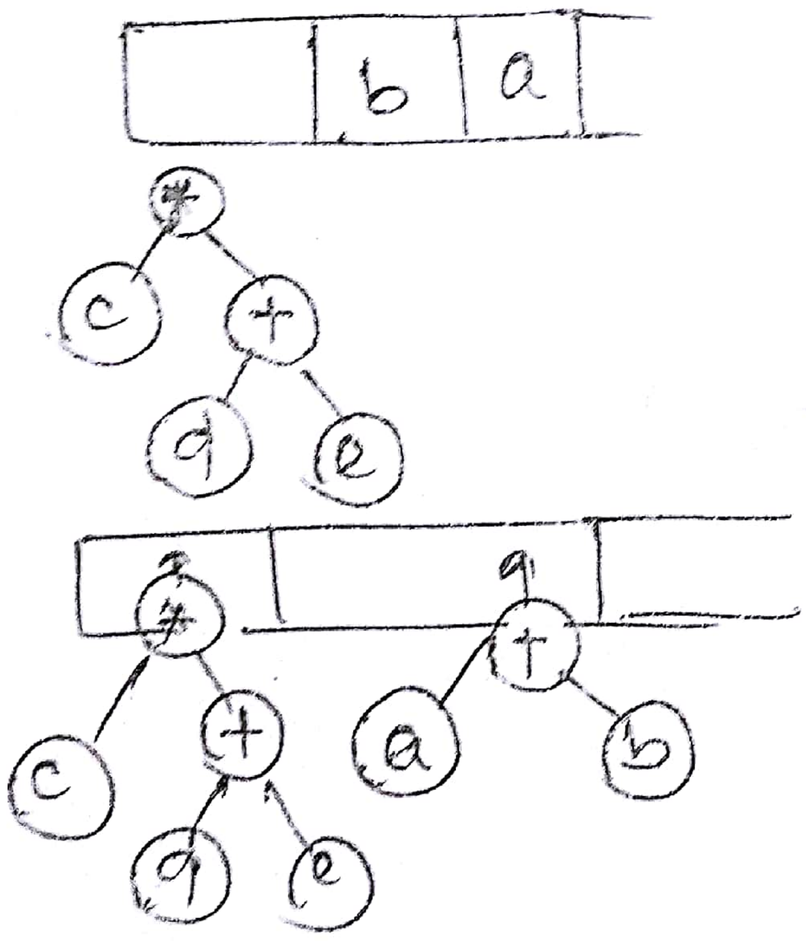
pop() → b push(b)



a

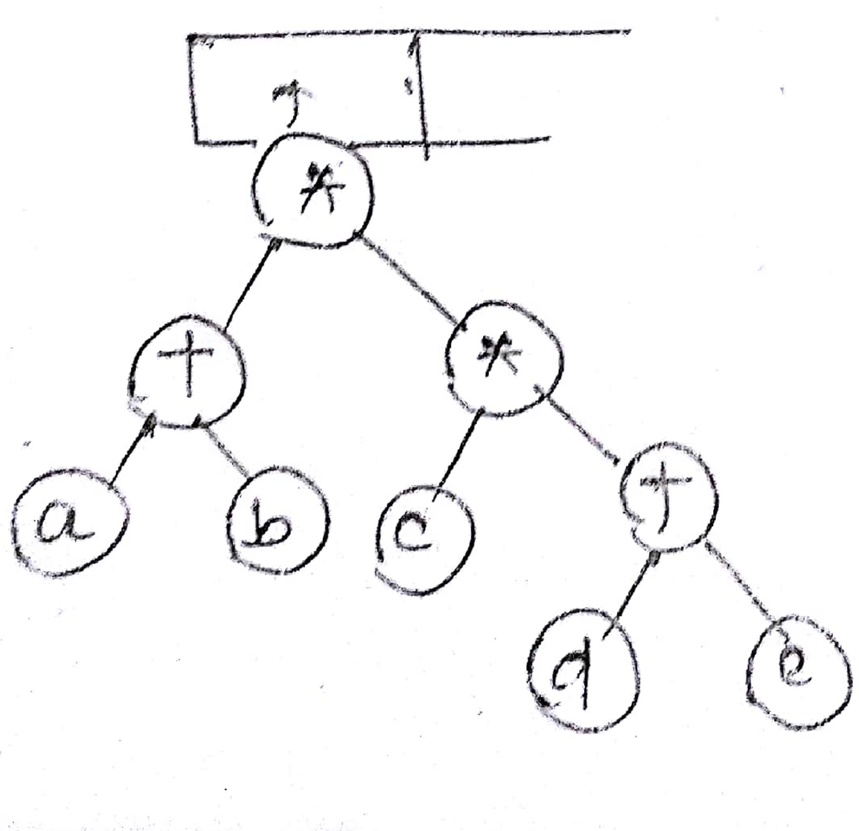
push(a)

\+



pop() → a
pop() → b
construct tree, push tree (a) to stack.

*



pop()
pop()
construct tree, push tree to stack.