Addition of two polynomials using linked list

BST Implementation

Splay, Red-Black Tree -

TREES I

Trees

Arrays, stacks, queues and linked lists are known as linear data structures where the elements are arranged in a linear fashion.

A tree is a non-linear data structure. Tree is used to model the hierarchical relationship among data. Example: We can represent a family hierarchy of various members as a tree which maintains the relationship among them.

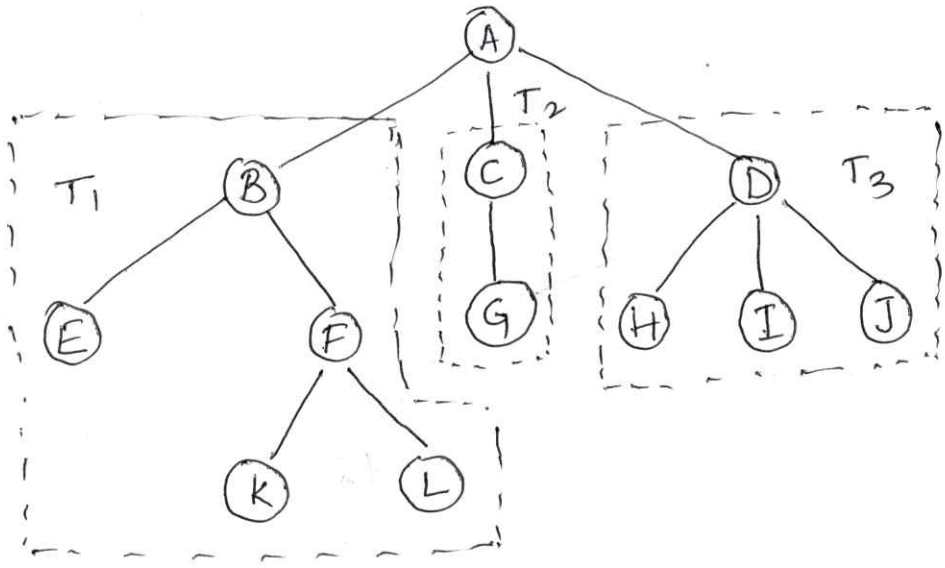
Defn:

A tree is a finite set of one or more nodes such that

i) There is a specially designated node called the root.

ii) The remaining nodes are partitioned into n ($n>0$) disjoint sets $T_1, T_2, \ldots, T_n$, where each T_i ($i=1,2,\ldots$) is a tree; $T_1, T_2, \ldots, T_n$ are called subtrees of the root.

Let us consider the sample tree shown in the figure below:



A Sample tree T.

In the sample tree T, there is a set of 12 nodes. Here, A is a special node being the root of the tree. The remaining nodes are partitioned into 3 sets T_1, T_2 and T_3; they are subtrees of the root node. By definition, each subtree is also a tree. Observe that a tree is defined recursively. The same tree can be expressed in a string notation as shown below:

$T \rightarrow (A(T_1, T_2, T_3))$.

$B(E, T_{12})$ $C(G)$ $D(H, I, J)$.

$F(K, L)$.

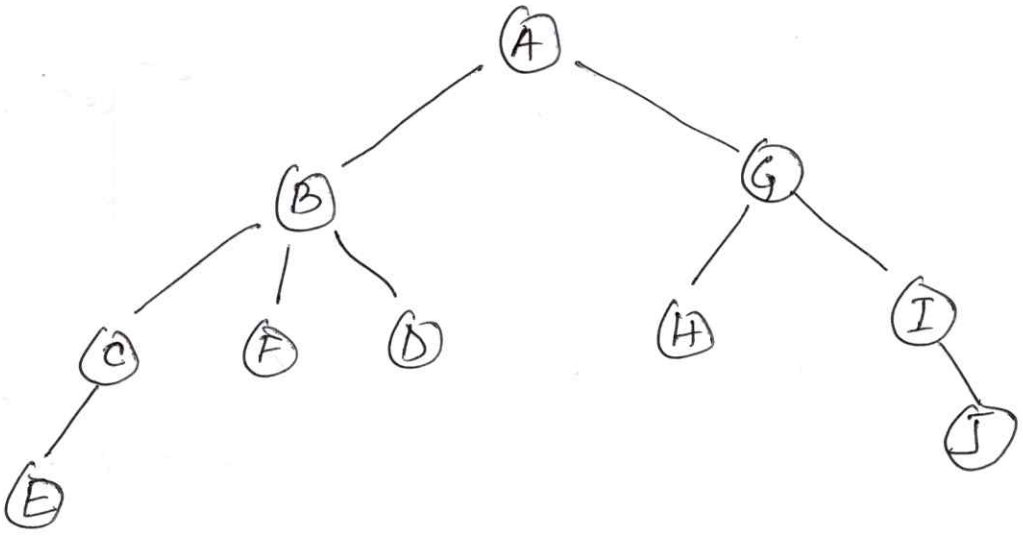
Or more precisely.

$T = (A(B(E, F(K, L))), C(G), D(H, I, J))$

Draw a tree with the given string notation:

$(A\,(B\,(C\,(E),\,F,\,D)),\,G\,(H,(I\,(J))))$.

T_3 (C(E), F, D) — T_1 (B(C(E), F, D)) — T_4 (I(J)) — T_2 (G(H, I(J)))



⊛ Basic Terminologies

1. Node: A node of a tree stores the actual data and links to the other node.
2. Parent: The immediate predecessor of a node.
3. Child: All immediate successors of a node.
4. Link or Edge: A pointer to a node in a tree. It is a connecting line of two nodes.
5. Root: A node without any parent. It is the first in the hierarchical arrangement of data items.
6. Leaf or terminal node: The node which is at the end and which does not have any child.

6. node with degree zero.

7. Non-terminal node: Any node (except the root node) whose degree is not zero.

8. Degree of a node: Maximum number of children that is possible for a node.

9. Level: Level is the rank in the hierarchy. Each node in a tree is assigned a level as follows:

   I. The root of the tree is at level 0.

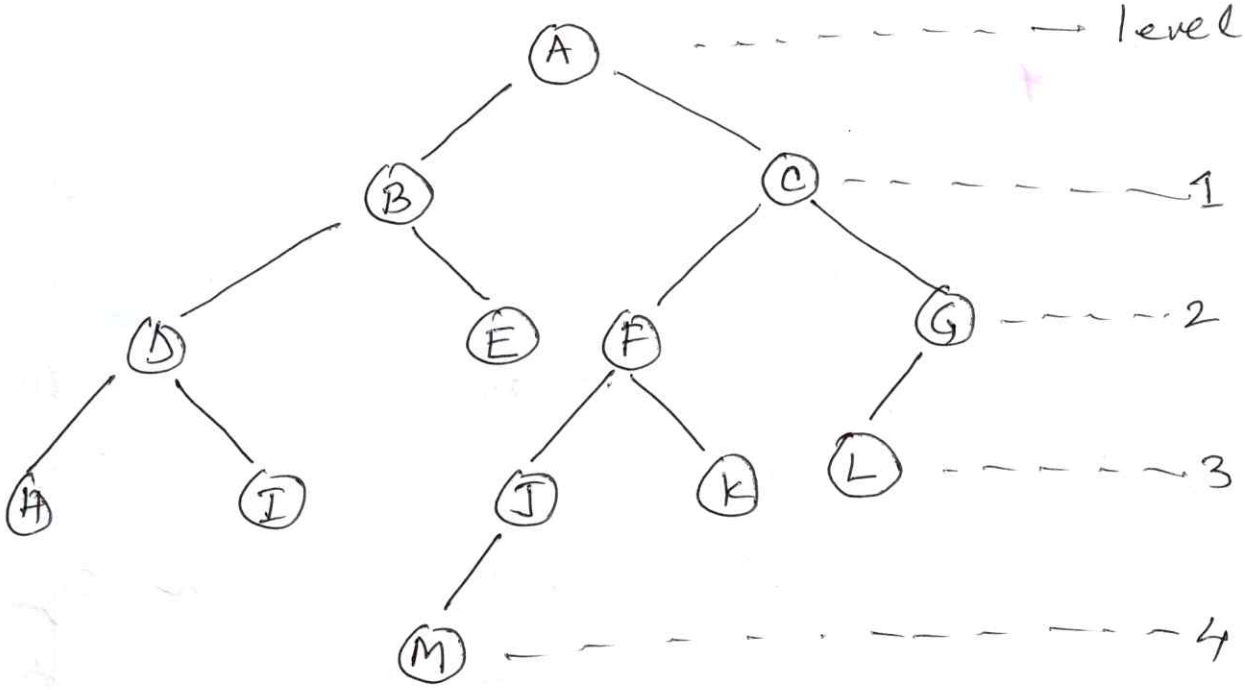
   II. Level of other node = level of its parent + 1.

10. Height: Maximum number of nodes that is possible in a path starting from root node to a leaf node.

11. Siblings: The nodes which have the same parent.

12. Path: Sequence of consecutive edges from the source node to the destination node.

13. Path length: The number of edges in a path.



A tree with 13 nodes.

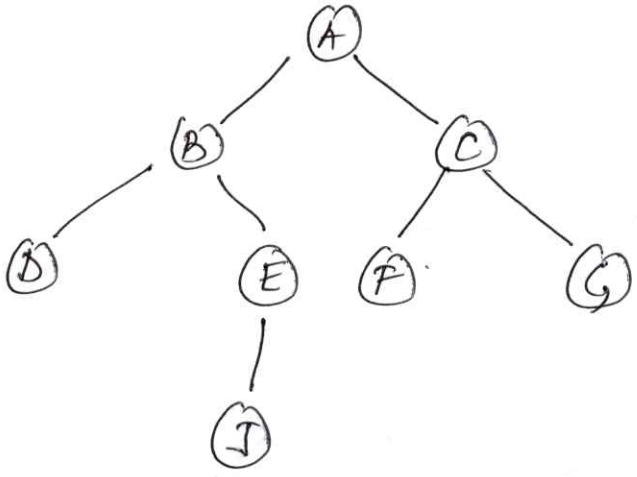
* H, I, E, M, K, L are the leafe nodes.
* The longest path is A - C - F + J - M and hence height is 5.
* Degree of the tree is 2.
* J & K are siblings.

Binary Tree.

A binary tree T is a finite set of nodes such that.

I. T is empty (called the empty binary tree) or

II) T contains a specially designated node called the root of T, and the remaining nodes of T form two disjoint binary trees T_1 and T_2 which are called the left subtree and the right subtree respectively.

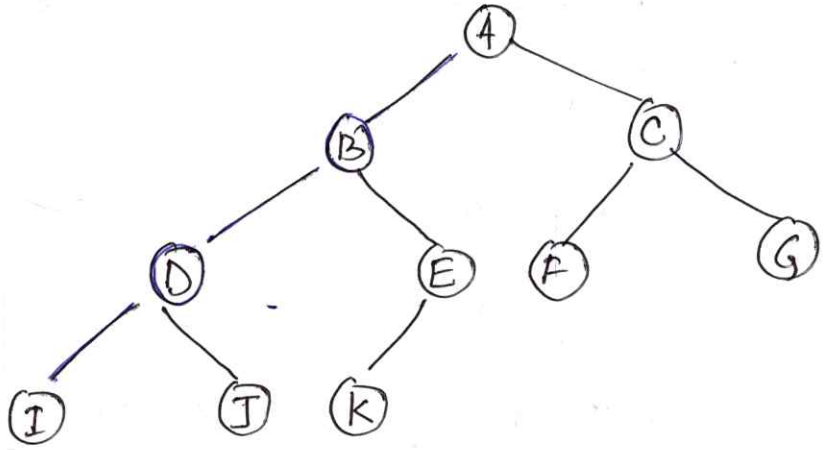


A node of a binary tree may have at most two children. (that is, a tree having degree = 2).

Types of Binary Tree

1. Complete Binary Tree: A binary tree is said to be complete if all its level

except possibly the last, have the maximum number of possible nodes, and if all the nodes at the last level appear as far left as possible.



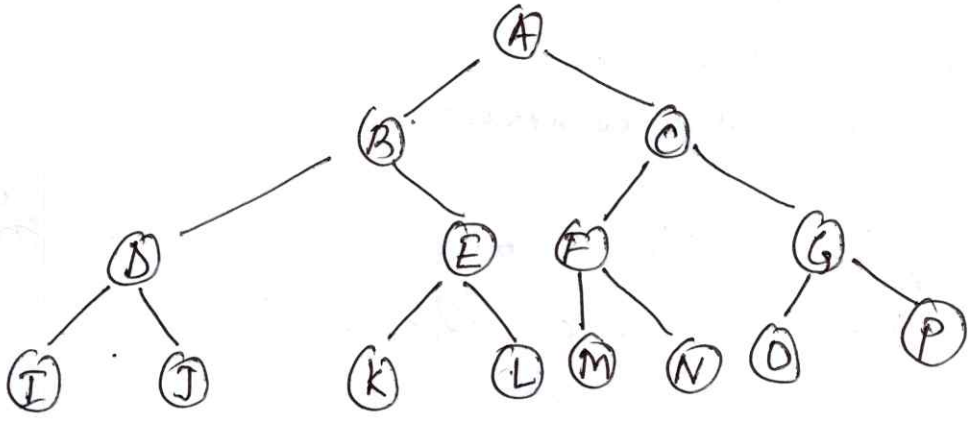
Complete Binary Tree.

2. Full Binary Tree: A binary tree is a full binary tree, if it contains maximum possible numbers of nodes in all level.

A binary tree is called a full binary tree if it satisfy the following two conditions:

i. Each node has either 0 or 2 children.

ii. All the leaf nodes are in the same level (and obviously this is the last level).



Full Binary Tree.

⊛ Properties of Binary Tree.

1. A tree with n nodes has exactly $(n-1)$ edges or branches.

2. The maximum number of nodes on level i is 2^i, where $i \geq 0$.

3. The maximum number of nodes in a binary tree with height h is $2^h - 1$.

4. For any non empty binary tree T, if n_0 is the number of leaf nodes (degree = 0) and n_2 is the number of internal nodes (degree = 2) then $n_0 = n_2 + 1$.

5. If n is the total number of nodes in a complete binary tree of height h, then $h = \lceil \log_2 (n+1) \rceil$

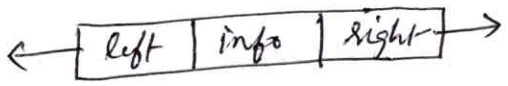
⊛ Representation of Binary Tree.

There are two ways to represent a binary tree in memory.

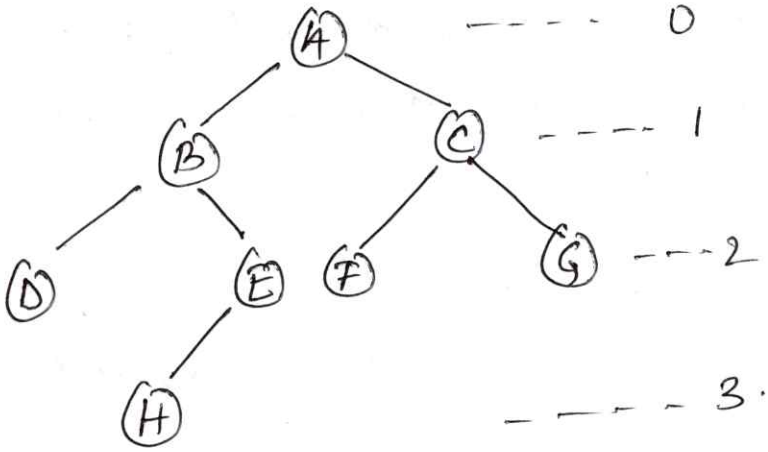
1. Linked representation
2. Sequential representation.

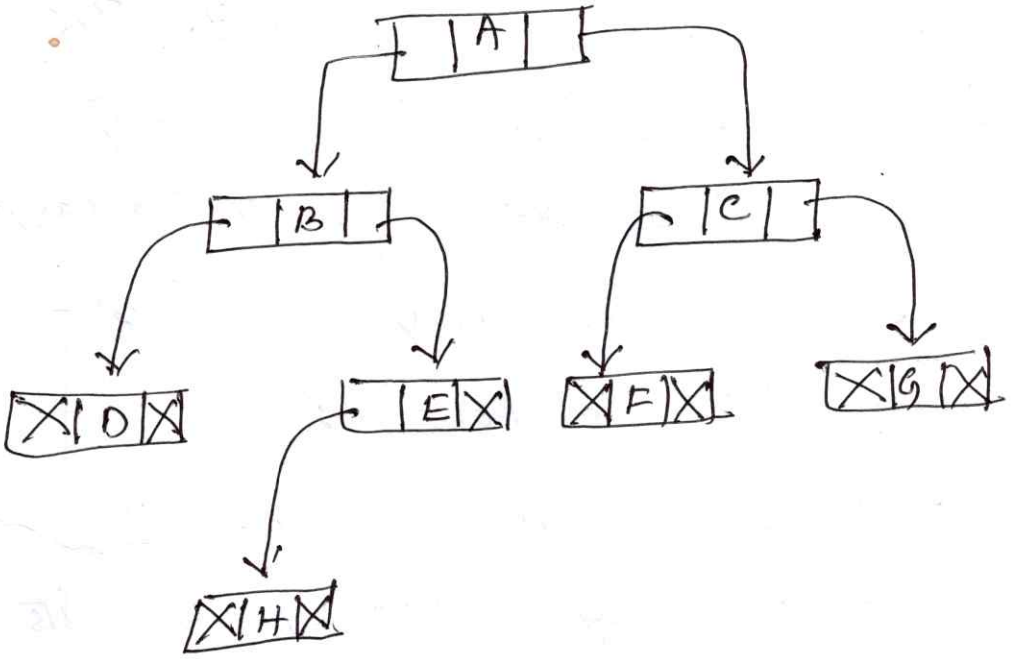
1. Linked representation: Here each node in a binary tree has three fields.

```
struct node
{   int info;
    struct node *left;
    struct node *right;
};
struct node *ptr,*root;
```



Structure of a node in linked representation





Tree using linked representation

→ ptr,root is a pointer to store the address of the root node in the tree.

→ left and right are the two link fields used to store the addresses of the left child and right child of a node; info is the information content of the node. With this representation, if one knows the address of the root node then from it any other node can be accessed.

→ The advantage of this representation is that it allows dynamic memory allocation; hence the size of the tree can be changed as and when the need arises.

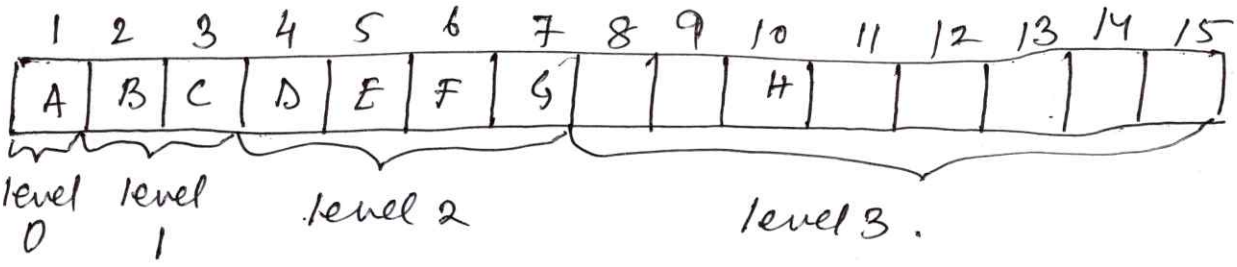
2. Sequential representation :

In this representation, the nodes are stored level by level, starting from the zero level where only the root node is present. The root node is stored in the first memory location (as the first element in the array).

This representation uses only a single linear array TREE as follows:

I. The root R of T is stored in TREE[1]

II. If a node N occupies TREE[K], then its left child is stored in TREE[2*K] and its right child is stored in TREE[2*K+1].



Sequential representation of the binary tree.

- For a binary tree with height h, $2^h - 1$ array space are required to store it.
- Useful for complete binary tree and most efficient for a full binary tree.

Traversals.

The traversal operation is used to visit each node in the tree exactly once. There are 3 standard ways of traversing a binary tree T with root R: Preorder, Inorder and Postorder.

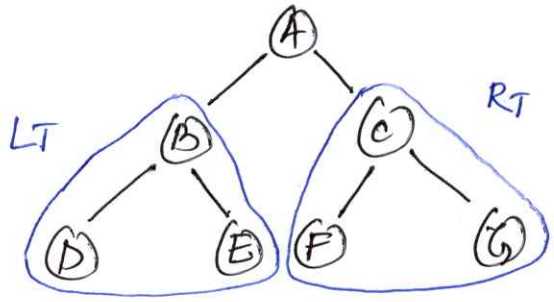
1) Preorder traversal of a binary tree.

1. Process the root R.
2. Traverse the left subtree of R in preorder.
3. Traverse the right subtree of R in preorder.

Recursive Preorder.

```
Preorder (ROOT)  // Root is a pointer to the root node of the binary tree.
1. ptr = ROOT
2. If (ptr ≠ NULL) then.
        1. Print ptr → info
        2. preorder (ptr → left)
        3. preorder (ptr → right)
```

3. End if.
4. Stop.



Preorder : A B D E C F G.

The preorder traversal of T processes A, traverses L_T and traverse R_T. However the preorder traversal of L_T processes the root B and then D and E. The preorder traversal of R_T processes root C and then F and G. Hence A B D E C F G is the preorder traversal of T.

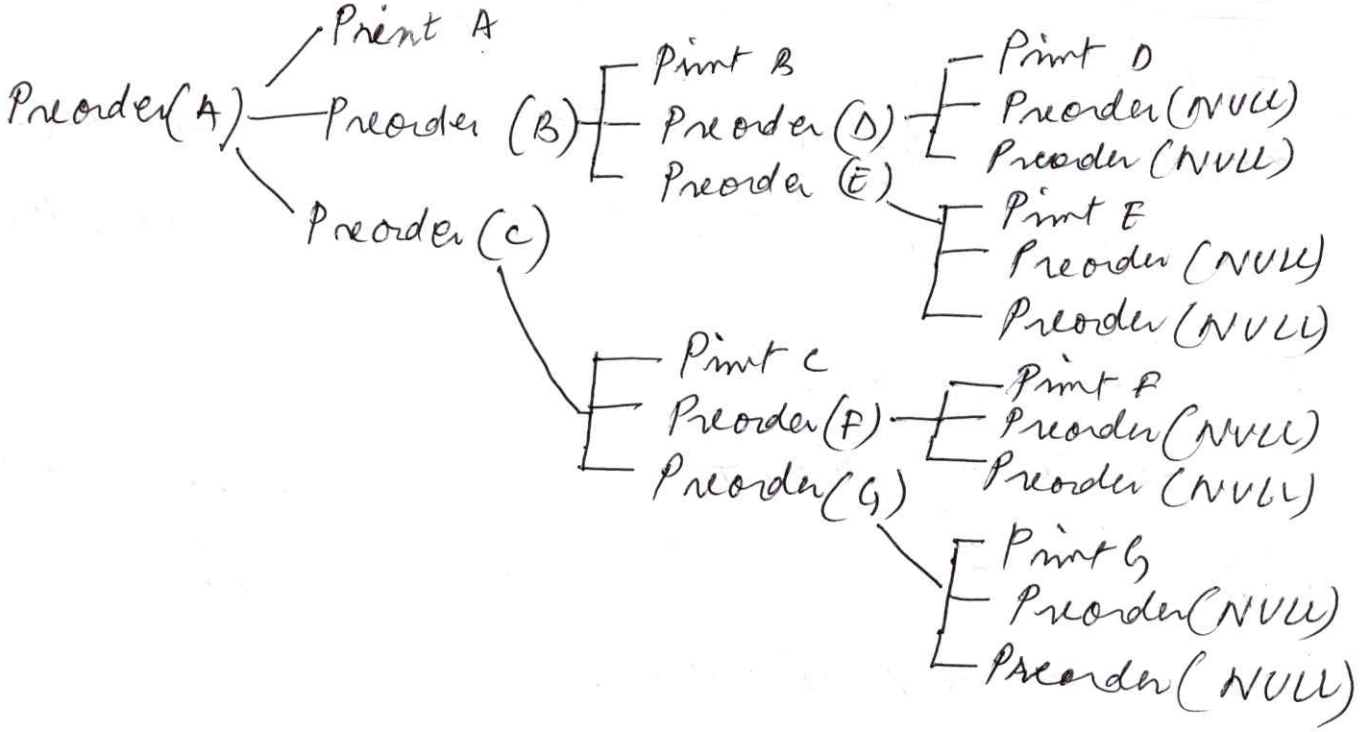


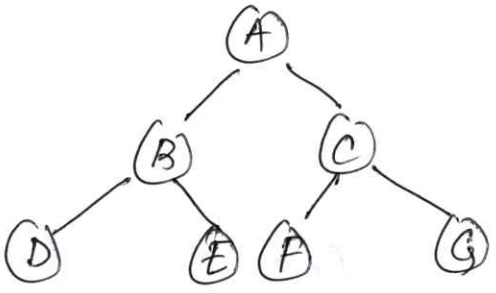
Fig : Tree of recursive calls for preorder Traversal of a Tree.

Inorder traversal of a binary tree.

1. Traverse the left subtree of the root R in inorder.
2. Process the root R.
3. Traverse the right subtree of the root R in inorder.

Inorder (ROOT)

1. ptr = ROOT
2. If (ptr ≠ NULL) then:
    1. Inorder (ptr → left)
    2. Print ptr → info
    3. Inorder (ptr → right)
3. Endif.
4. stop.



Inorder: DBEAFCG.

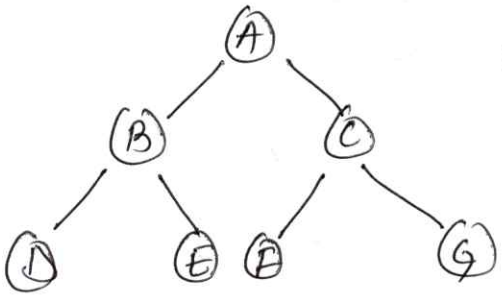
The inorder traversal of T traverses L_T, processes A and traverses R_T. However, the inorder traversal of L_T processes B, B and then E. And the inorder traversal of R_T processes F and C and then G. Hence DBEAFCG is the inorder traversal of T.

Postorder traversal of a binary tree.

1. Traverse the left subtree of the root R in postorder.
2. Traverse the right subtree of the root R in postorder.
3. Process the root R.

Postorder (ROOT)

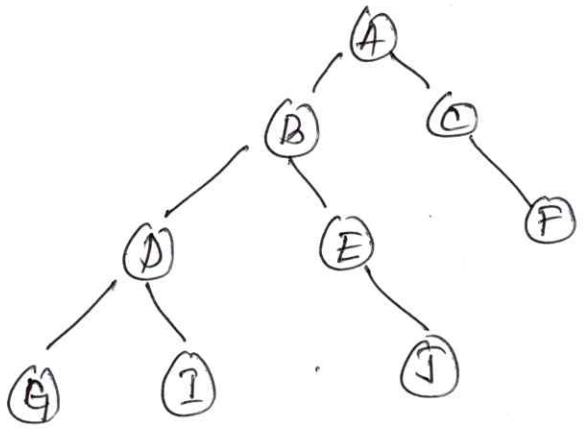
1. ptr = ROOT
2. If (ptr ≠ NULL) then.
    1. postorder (ptr → left)
    2. postorder (ptr → right)
    3. Print ptr → info.
3. End if.
4. Stop.



Postorder: D E B F G C A.

The postorder traversal of T traverse L_T, traverses R_T and processes A. However the postorder traversal of L_T process D, E and then B. The postorder traversal of R_T processes F,

G and then C. Finally root A is processed.



Preorder Traversal : A B D G I E J C F
Inorder Traversal : G D I B E J A C F
Postorder Traversal : G I D J E B F C A.

(*) Formation of Binary tree from its traversals.

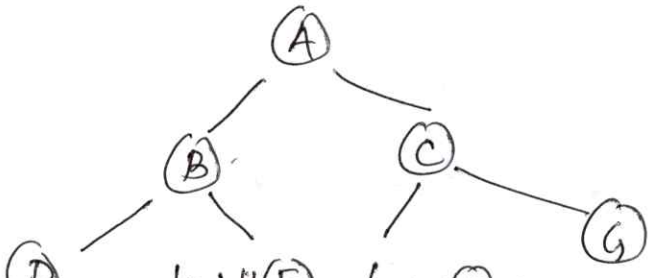
Creation of Binary tree from Preorder and Inorder traversal.

1. Inorder : D B H E A I F J C G
   Preorder : A B D E H C F I J G.

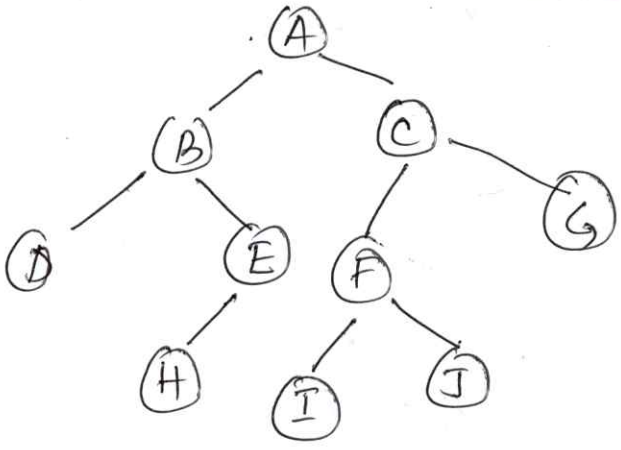
   1. From the preorder traversal, it is evident that A is the root node.
   2. In Inorder traversal, all the nodes which are on the left side of A belong to the left subtree and those which are on right side of A belong to the right subtree.
   3. Now, the problem reduces to form subtrees

and the same procedure can be applied recursively.

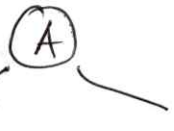
In: D B H E In: I F J C G
Pre: B D E H Pre: C F I J G



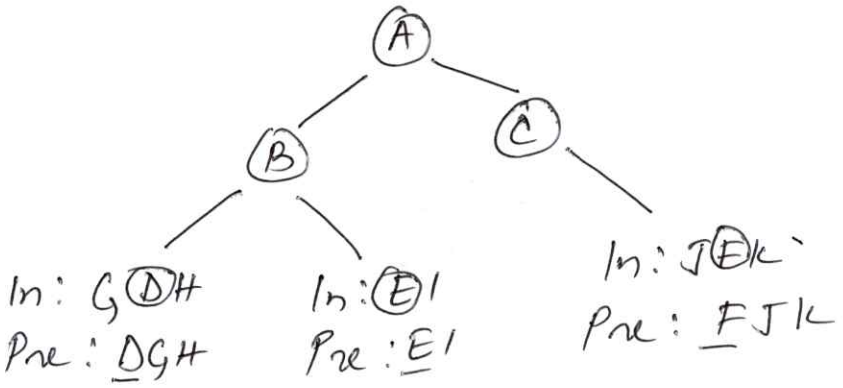
In: H E In: I F J
Pre: E H Pre: F I J

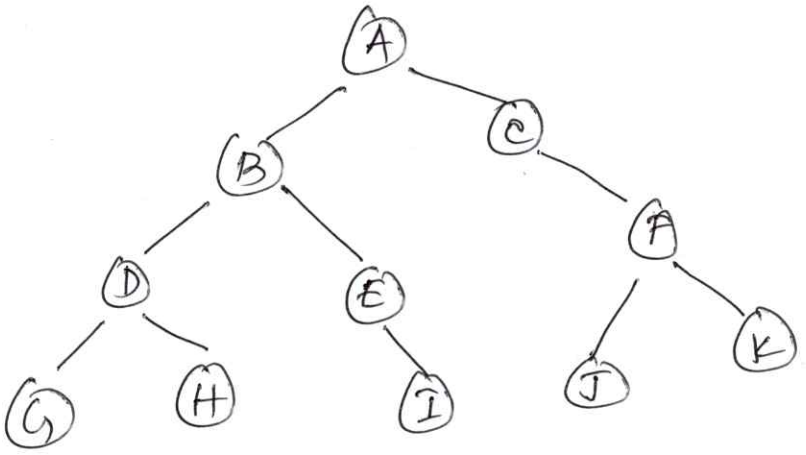


2. Inorder: G D H B E I A C J F K.
Preorder: A B D G H E I C F J K.



In: G D H B E I In: C J F K
Pre: B D G H E I Pre: C F J K.





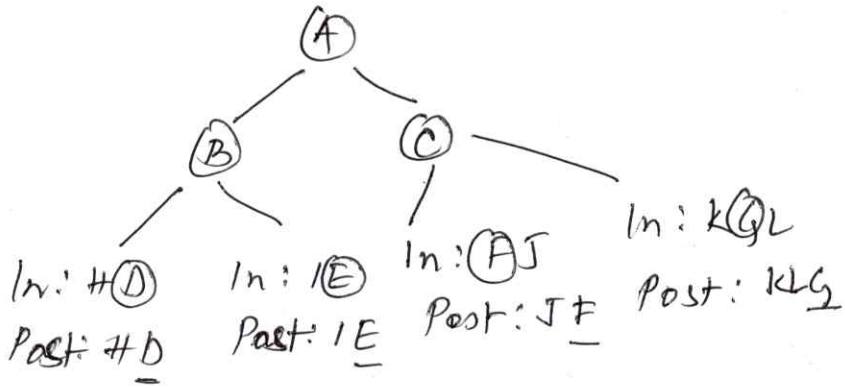
Creation of Binary tree from Postorder and Inorder traversal.

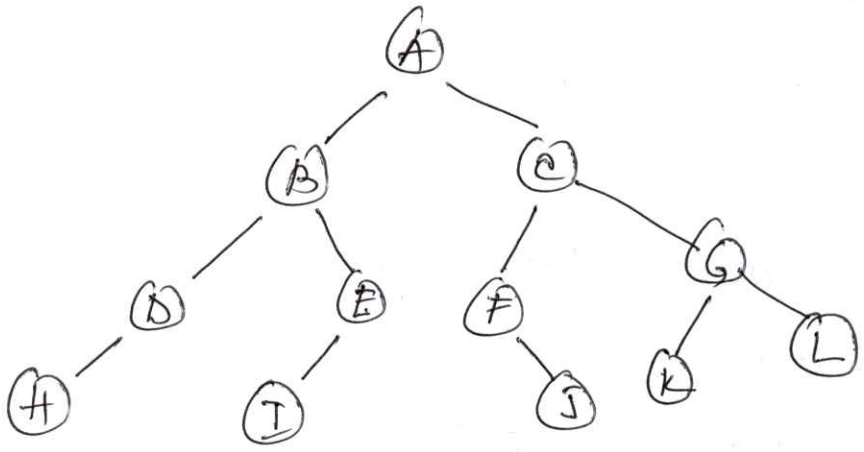
1. Inorder: H D B I E A F J C K G L
   PostOrder: H D I E B J F K L G C A

   In postorder traversal, the last nodes is the root node. In inorder traversal, all the nodes left of the root node belong to left subtree and all nodes right of the root node belong to the right subtree.

In: # D B I E
Post: # D I E B

In: F J C K G L
Post: J F K L G C



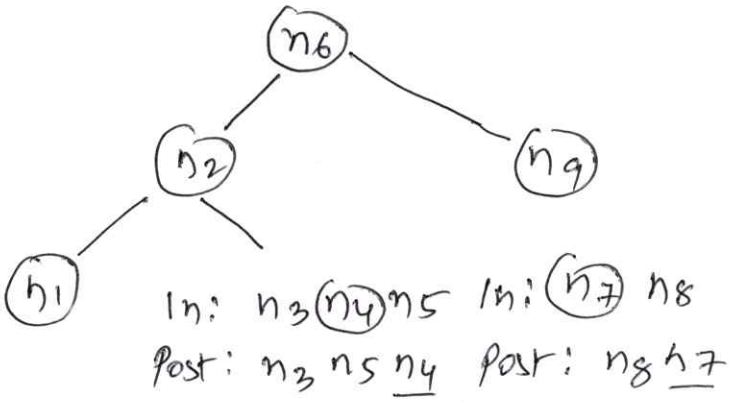


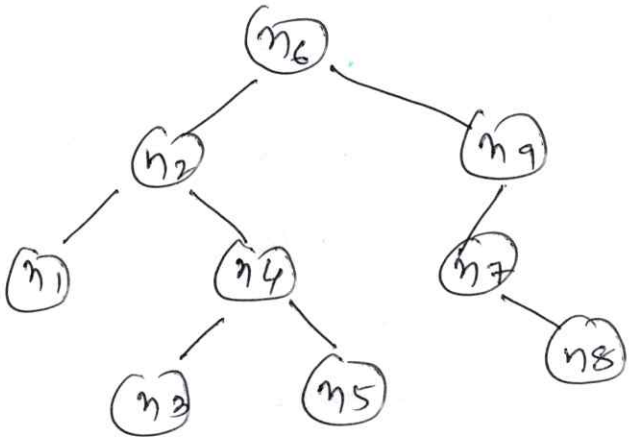
(2) Inorder : C D E B G H F K L P Q M N J A.
Postorder: E D C H G Q P N M L K J F B A.

(3) Inorder: n_1 n_2 n_3 n_4 n_5 n_6 n_7 n_8 n_9.
Postorder: n_1 n_3 n_5 n_4 n_2 n_8 n_7 n_9 n_6

In: n_1 n_2 n_3 n_4 n_5
Post: n_1 n_3 n_5 n_4 n_2

In: n_7 n_8 n_9
Post: n_8 n_7 n_9





Binary Search Tree (BST)

A binary tree T is called a binary search tree if each node N of T satisfies the following properties:

I. The value of N is greater than every value in the left subtree of N and.

II. The value of N is less than every value in the right subtree of N.

(*) Inserting into a BST

a) Compare ITEM with the root node N of the tree

   I. If ITEM < N, proceed to the left child of N.

   II. If ITEM > N, proceed to the right child of N.

b) Repeat step (a) until we meet an empty subtree and then we insert ITEM in place of the empty subtree.

INSERT_BST:

1. ptr = ROOT, flag = FALSE.
2. While (ptr ≠ NULL) and (flag = FALSE) do.
    1. Case: ITEM < ptr → INFO.
        a. ptr1 = ptr.
        b. ptr = ptr → LEFT
    2. Case: ITEM > ptr → INFO
        a. ptr1 = ptr.
        b. ptr = ptr → RIGHT
    3. Case: ptr → INFO = ITEM.
        a. flag = TRUE.
        b. Print "ITEM already exists".
        c. exit
    4. Endcase.
3. Endwhile.
4. If ptr = NULL then.
    1. new = GetNode (NODE) OR (NODE *) malloc (sizeof (N
    2. new → INFO = ITEM.
    3. new → LEFT = NULL
    4. new → RIGHT = NULL

5. If (ptr1 → INFO < ITEM) then.

1. ptr1 → RIGHT = new.

6. Else

1. ptr1 → LEFT = new.

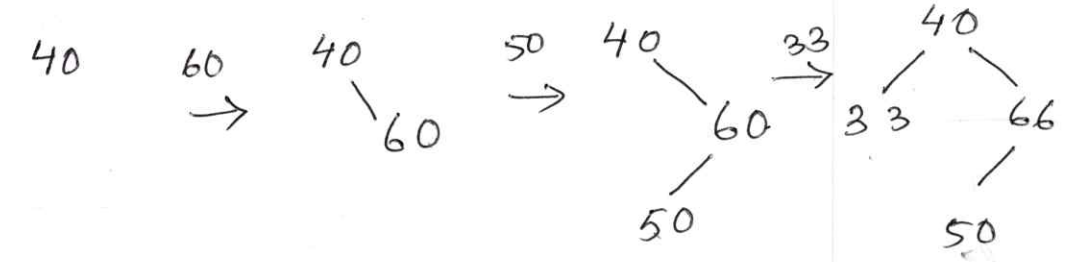
7. Endif.

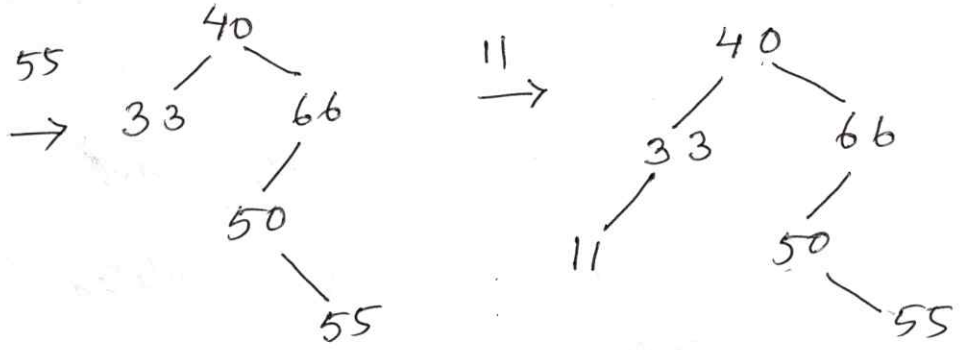
5. Endif

6. Stop.

1. Draw a BST with the following data:

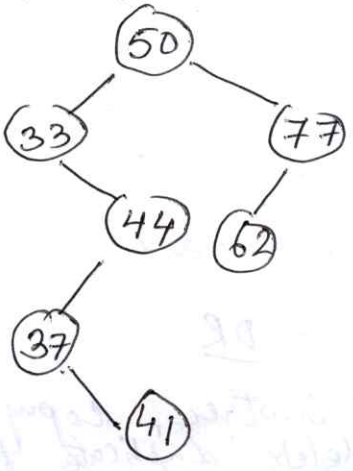
(40), 60, 50, 33, 55, 11.





2. (50), 33, 44, 77, 37, 62, 41

ODE))



→ The inorder traversal of a given search tree gives a list in ascending order.

⊛ Deleting in a BST

To delete N from a BST depends primarily on the number of children of node N. There are three cases:

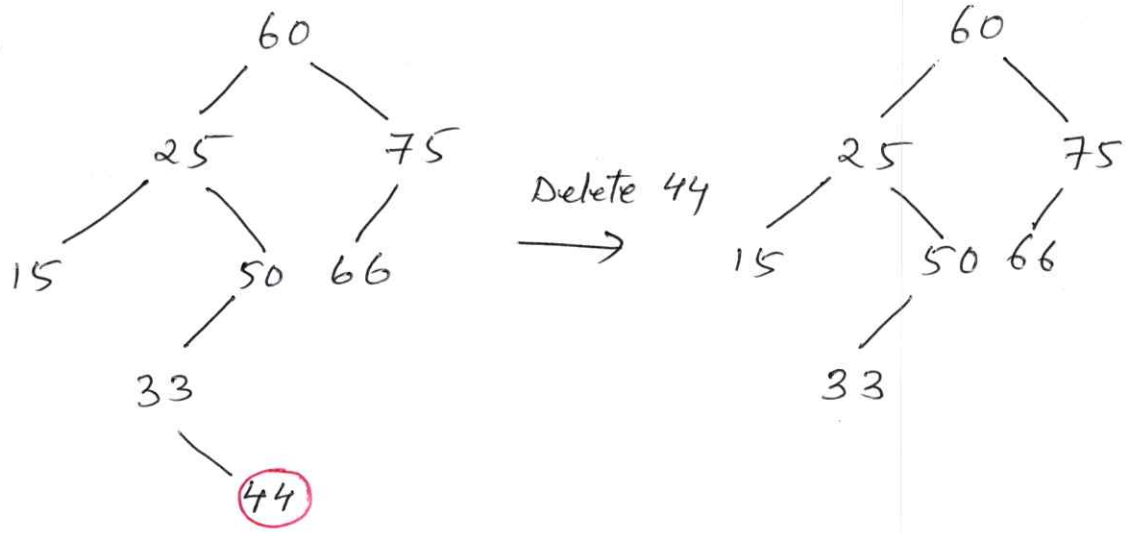
Case 1: N has no children. Then N is deleted from T by simply replacing the location of N in the parent node P(N) by the NULL pointer

Case 2: N has exactly one child. Then N is deleted from T by simply replacing the location of N in P(N) by the location of the only child of N.

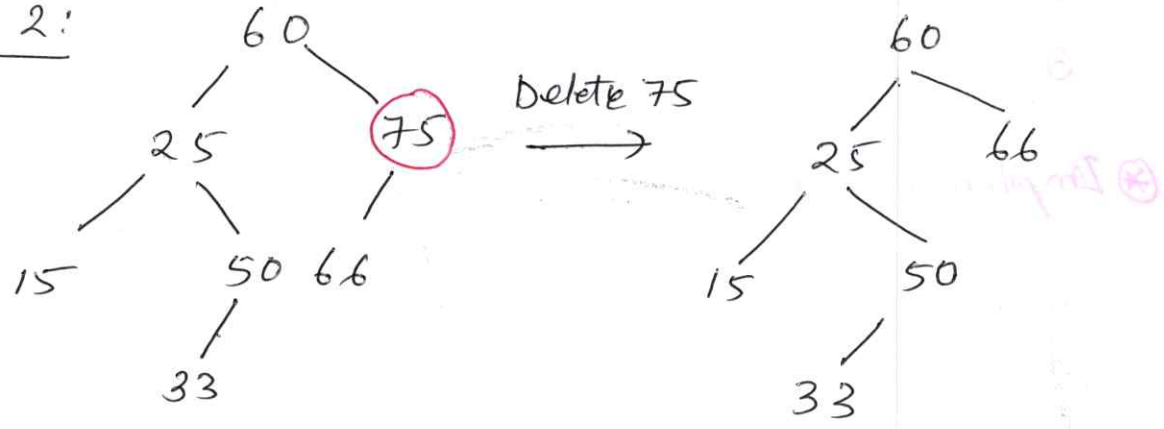
Case 3: N has two children. Let S(N) denote the inorder successor of N. Then N is deleted from T by first ~~deleting~~ deleting S(N) from T (by using case 1 or case 2) and then replacing node N in T by the node S(N).

- Find min in right subtree, copy the value in the targeted node and delete the duplicate from right subtree. OR
- Find max in left subtree, copy the value in the targeted node and delete duplicate from left subtree

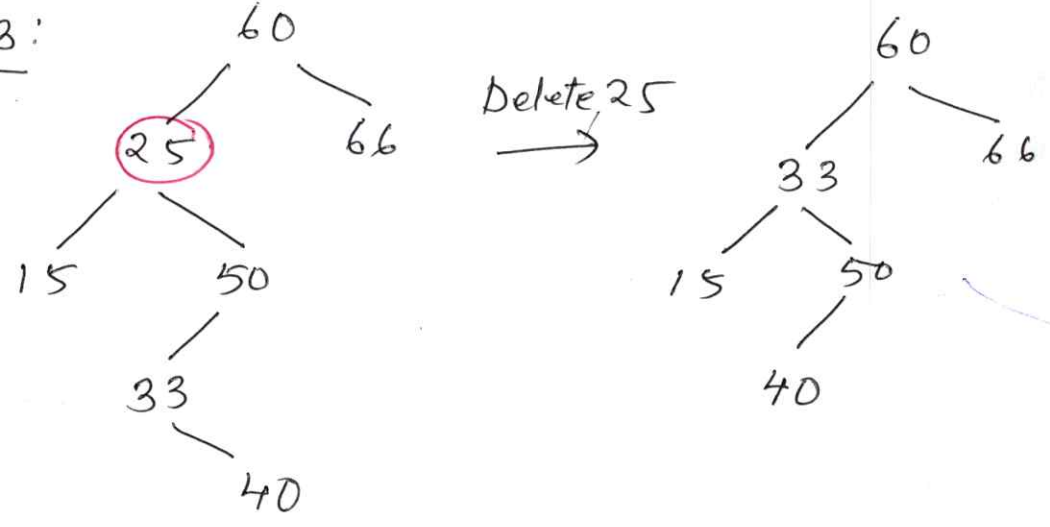
Case 1:



Case 2:



Case 3:



1. Find inorder successor (33) of N(25).
2. Delete inorder successor (33) using case 1 & case 2.
3. Replace 25 by 33.

Inorder : 15, 25, 33, 44, 50, 60, 66.

The inorder successor of N can be found by moving to the right child of N and then moving repeatedly to the left until meeting a node with an empty left subtree.

(*) Three operations of BST:

1. Traverse/Search
2. Insert.
3. Delete.

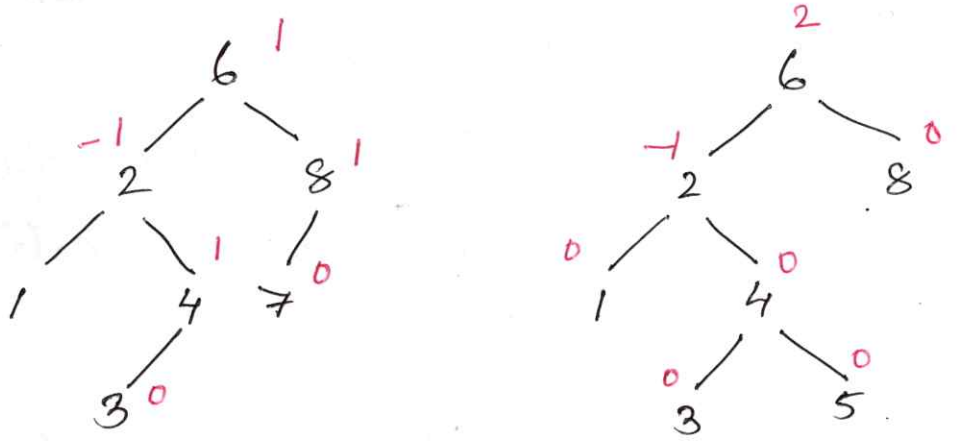
(*) Implementation

Height Balanced Tree or AVL tree.

A binary search tree is said to be height balanced binary search tree if all its nodes have a balance factor of 1, 0 or -1, where

balance factor = height of left subtree (h_L) − height of right subtree (h_R).

Consider the following 2 binary search tree, out of which one satisfies the properties of a height balanced binary search tree and the other does not.



AVL tree was developed by 2 Russian mathematicians G M Adelson Velskii and E. M. Landis.

Insertion into AVL tree.

1.) First insert the node into the AVL Tree according to the rules of insertion into binary search tree.

2) On the path starting from the root node to the node newly inserted, compute the balance factors of each node.

3) On the path traced in step 2, determine whether the absolute value of any node's balance factor is switched from 1 to 2. If so, the tree becomes unbalanced. The node which has its absolute value of balance factor switched from 1 to 2 is called the pivot node. (There may be more than one node which has its balance factor switched from 1 to 2, but the nearest node to the newly inserted node will be the pivot node).

4) Then perform one of the four AVL rotations on the unbalanced tree to make it balanced depending on some criteria.

(*) A height balanced tree is always a binary search tree and a complete binary search tree is always height balanced, but the reverse may not be true.

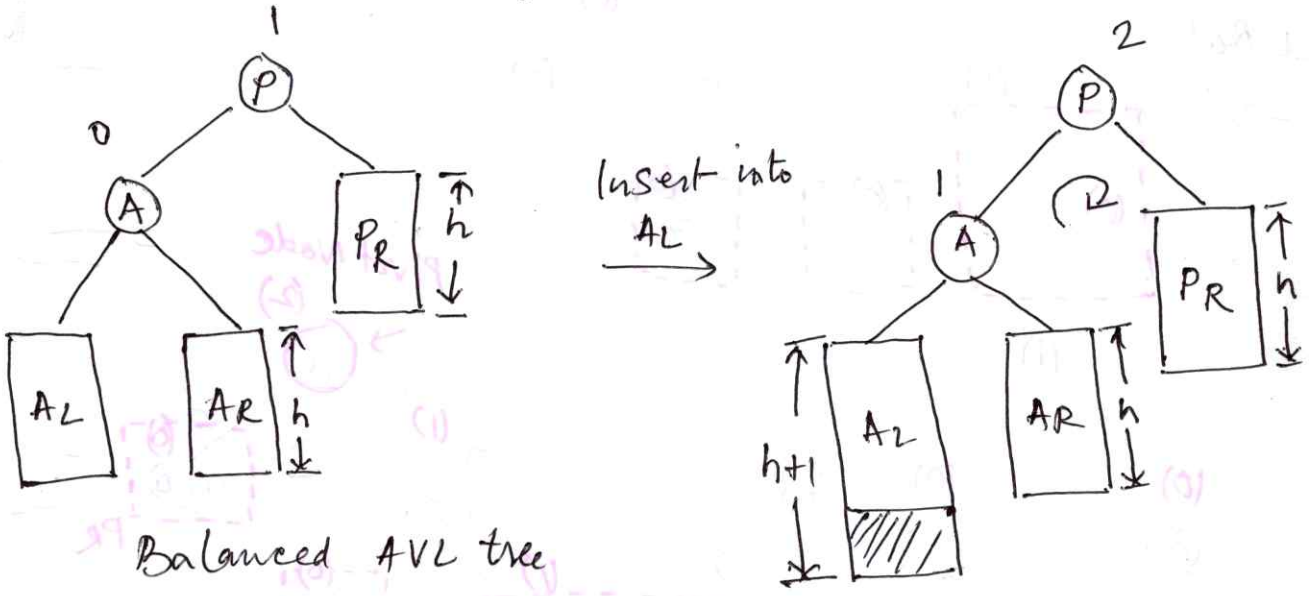
Case 1: Left to Left rotation (Insertion)

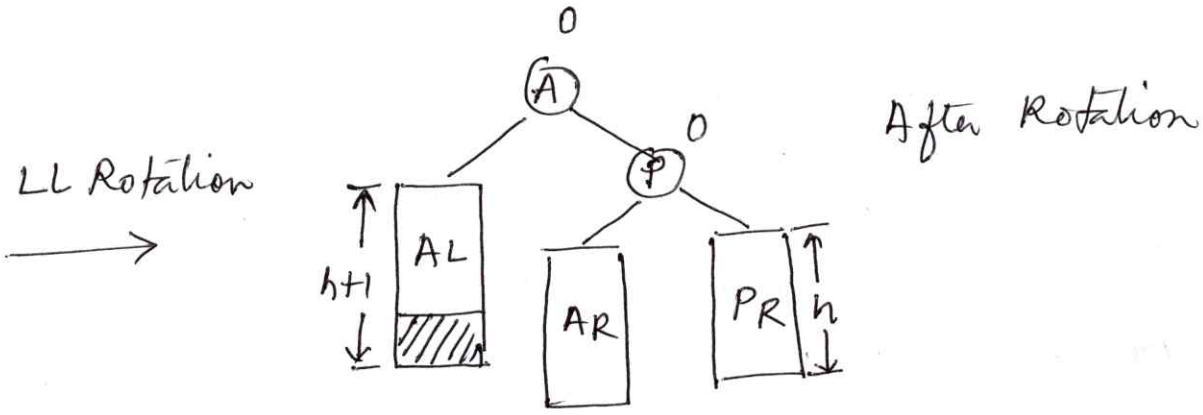
When the pivot node is left heavy and the new node is inserted in left subtree of the left child of pivot node then the rotation performed is left to left rotation.

In this case, the following manipulations in pointers take place:

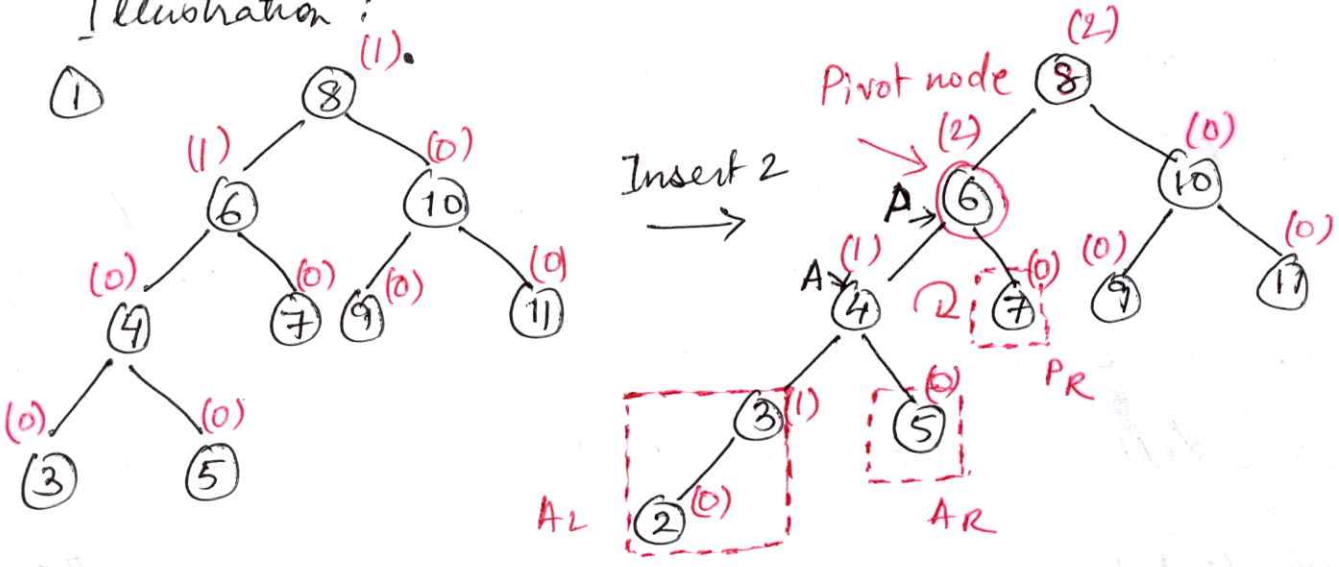
1. Right subtree (A_R) of left child (A) of pivot node (P) becomes the left subtree of P.
2. P becomes the right child of A.
3. Left subtree (A_L) of A remains the same.

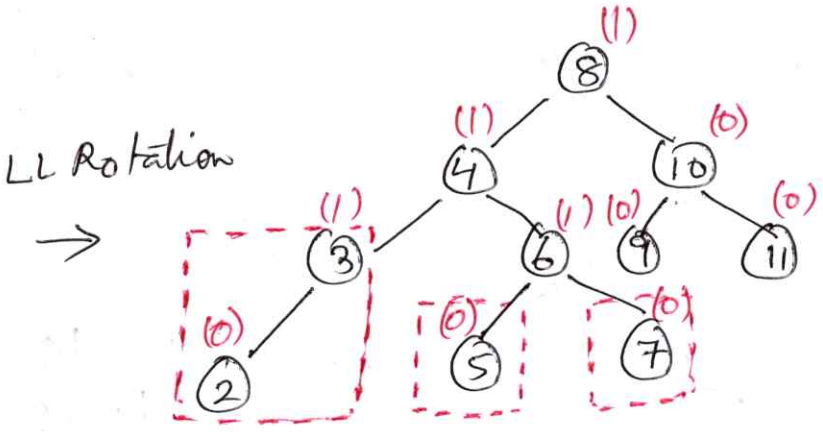
(*) Unbalance occurs due to insertion in the left subtree of the left child of the pivot node.

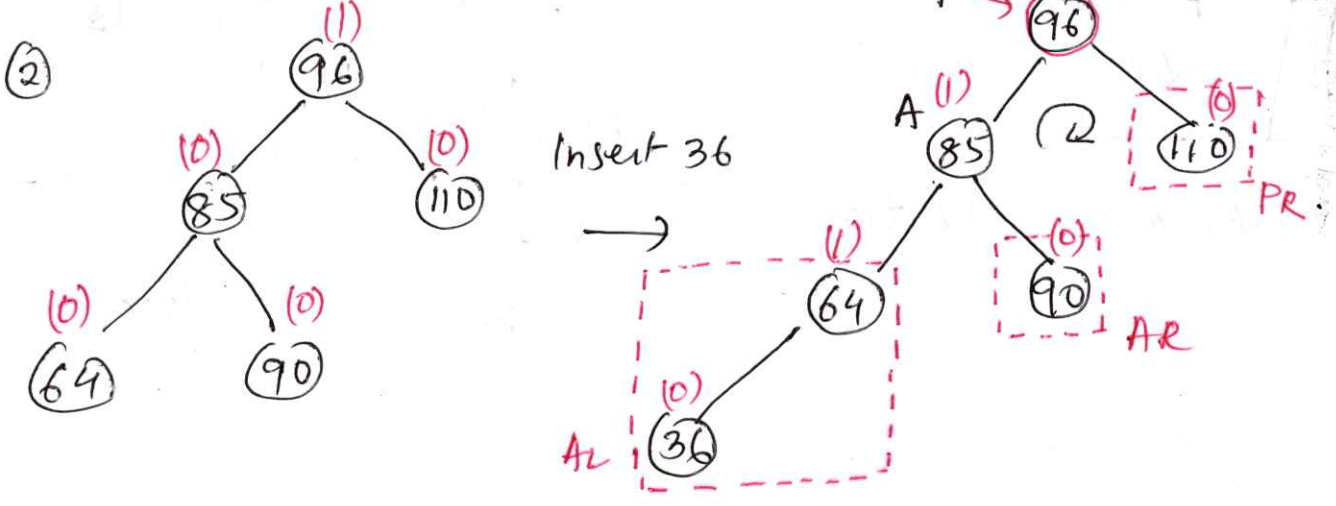




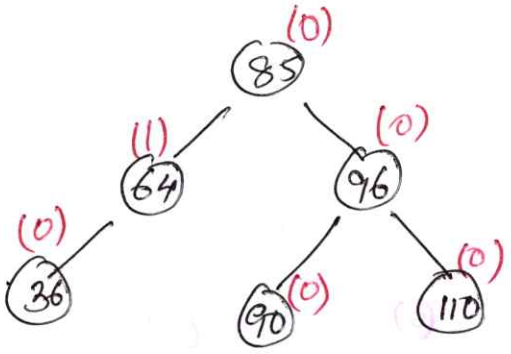
1 Illustration :







LL Rotation



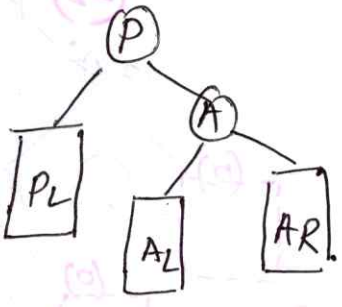
Case 2: Right to Right Rotation

Unbalance occurs due to insertion in the right subtree of the right child of the pivot node. In this case, the following manipulations in pointers take place:

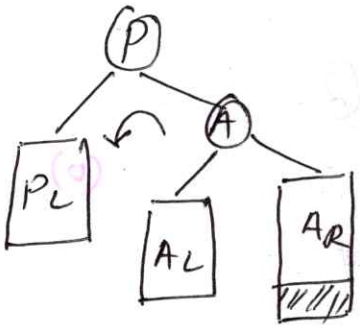
i. Left subtree (A_L) of right child (A) of pivot node (P) becomes the right subtree of P.

ii. P becomes the left child of A.

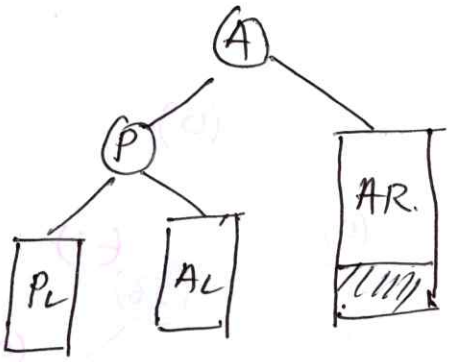
iii. Right subtree (A_R) of A remains same.



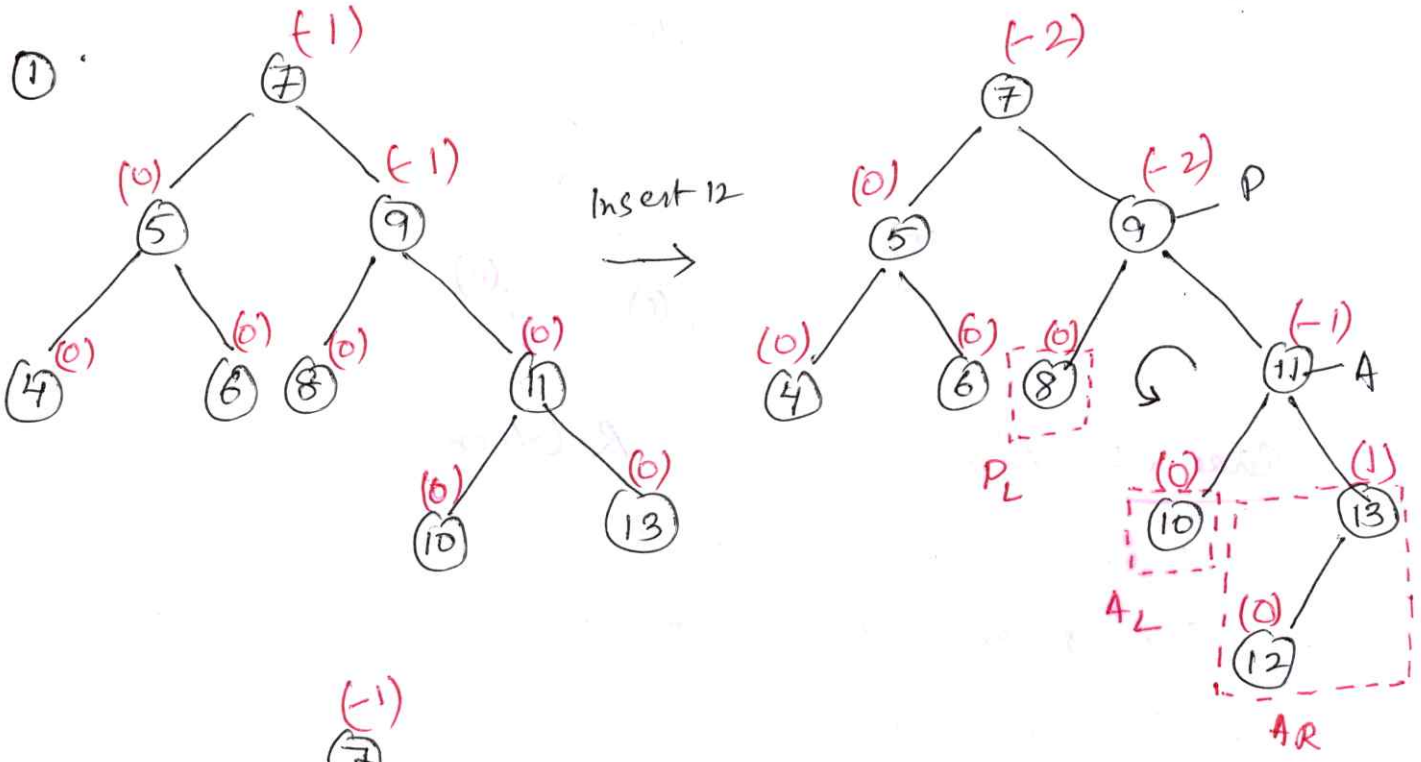
Insert into AR →



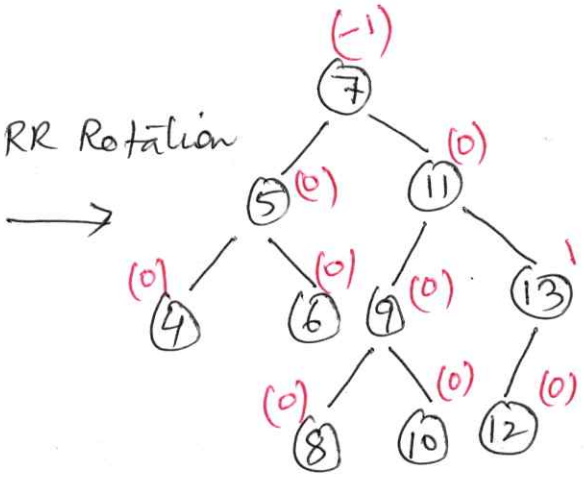
RR Rotation →



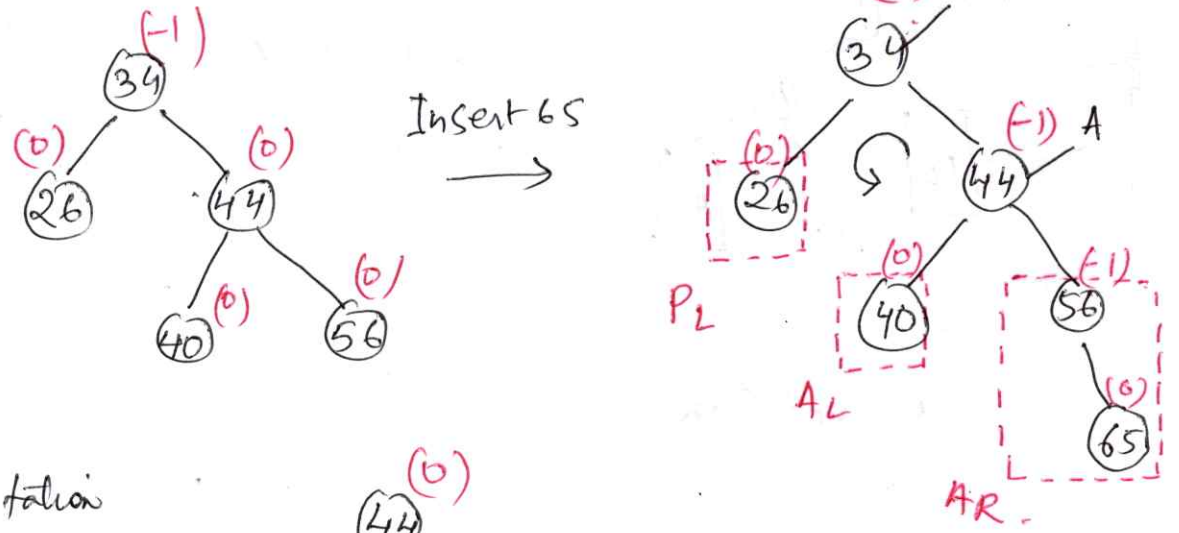
(1)



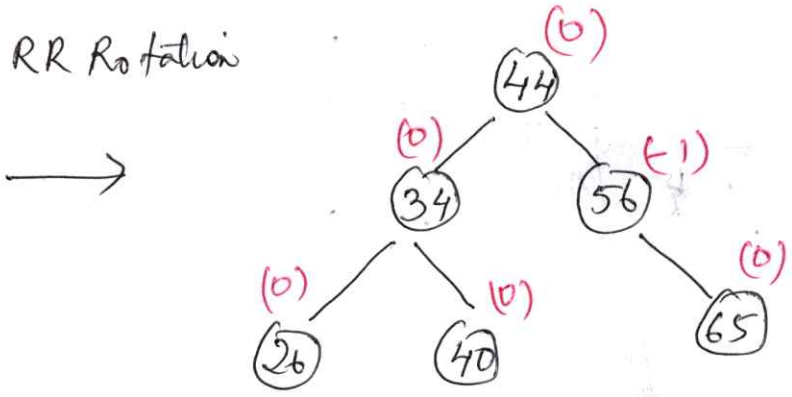
RR Rotation



(2)

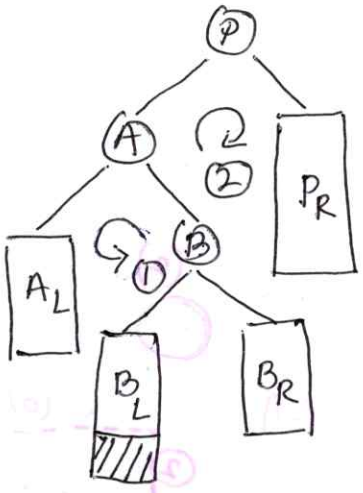


RR Rotation

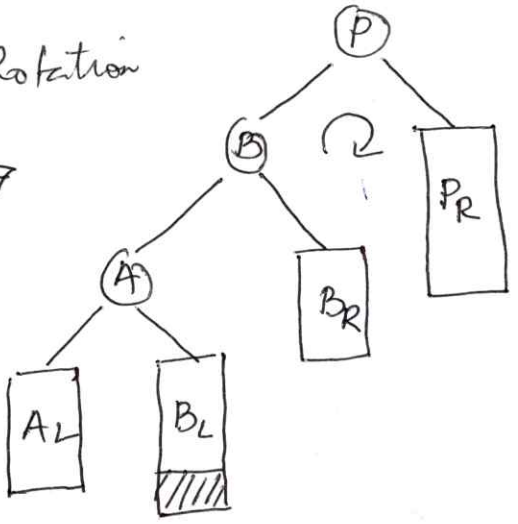


Case 3 : Left to Right Rotation

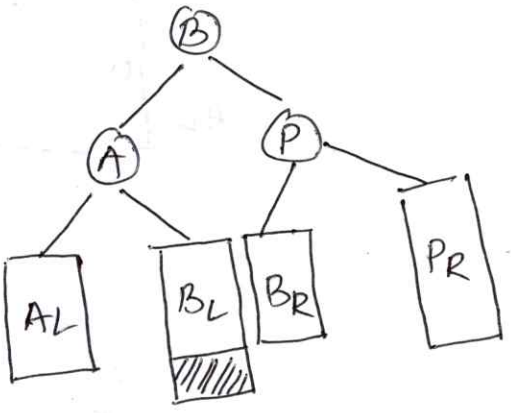
Unbalance occurs due to insertion in the right subtree of the left child of the pivot node.



RR Rotation →



LL Rotation →



Case 3 involves 2 rotations for the manipulation in pointers.

Rotation 1 :

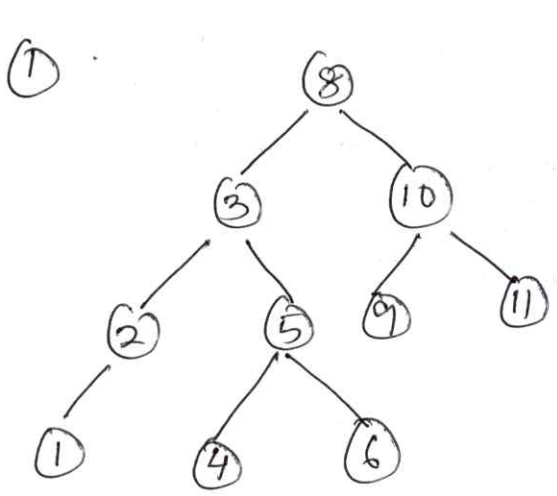
i. Left Subtree (B_L) of the right child (B) of the left child of the pivot node (P) becomes the right subtree of the left child (A).

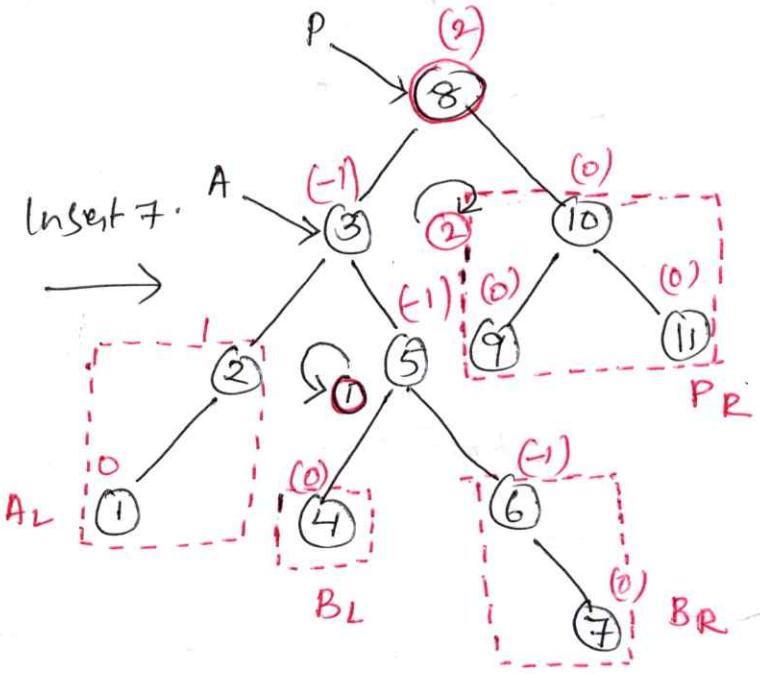
ii. Left child (A) of the pivot node (P) becomes the left child of (B).

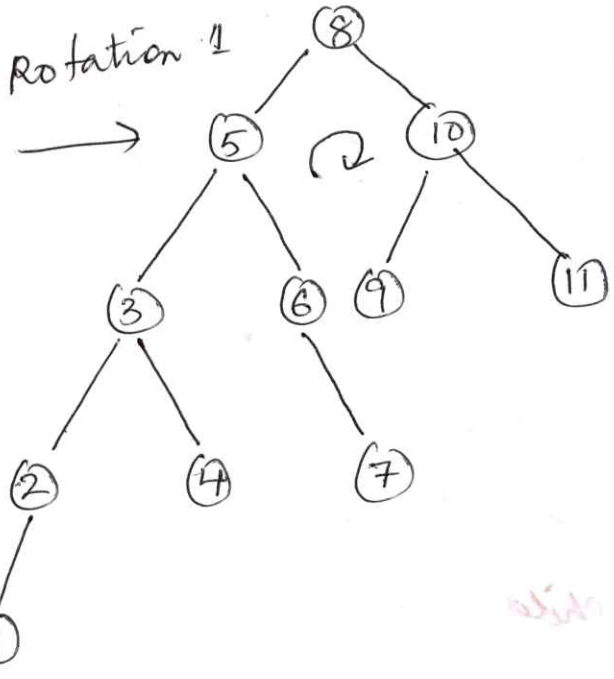
Rotation 2.

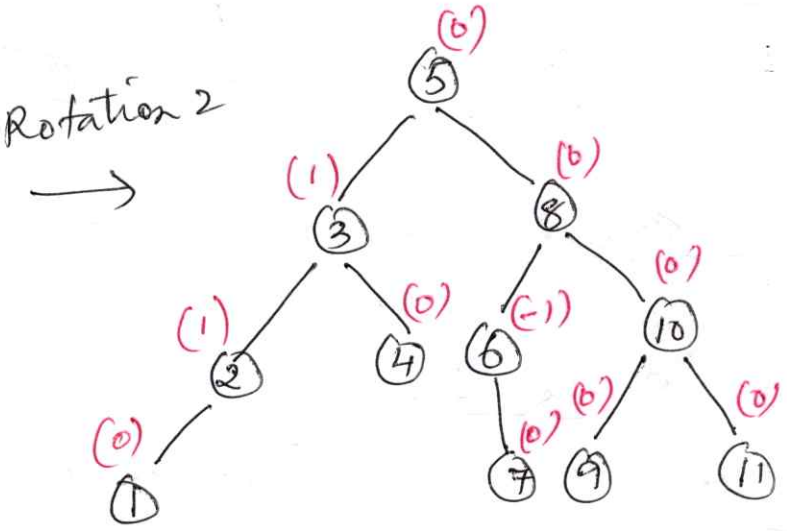
i. Right subtree (B_R) of the right child (B) of the left child (A) of the pivot node (P) becomes the left subtree of child (A) of the pivot node (P) becomes the left subtree of (P).

ii. (P) becomes the right child of (B).



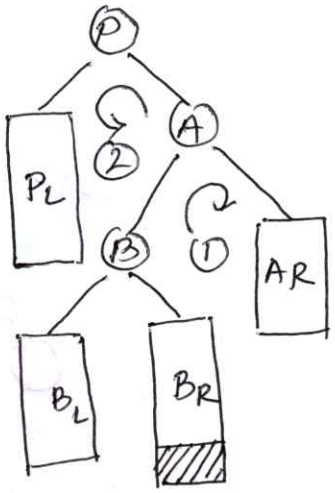




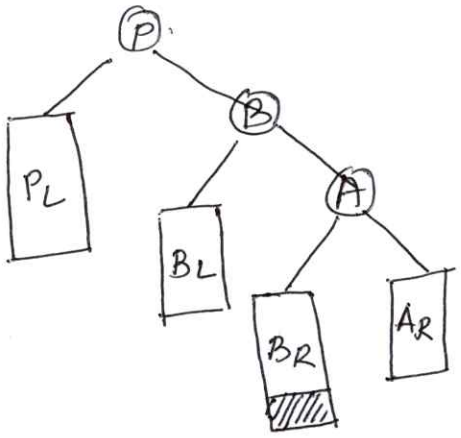


Case 4: Right to Left Rotation

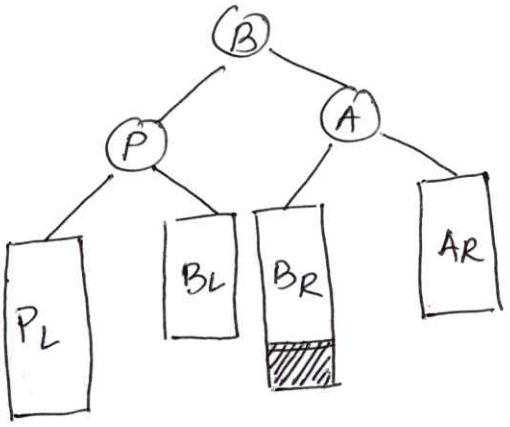
Unbalance occurs due to insertion in the left subtree of right child of the pivot node.



LL Rotation →



RR Rotation →



There are two rotations for the manipulations of pointers in this case. These are:

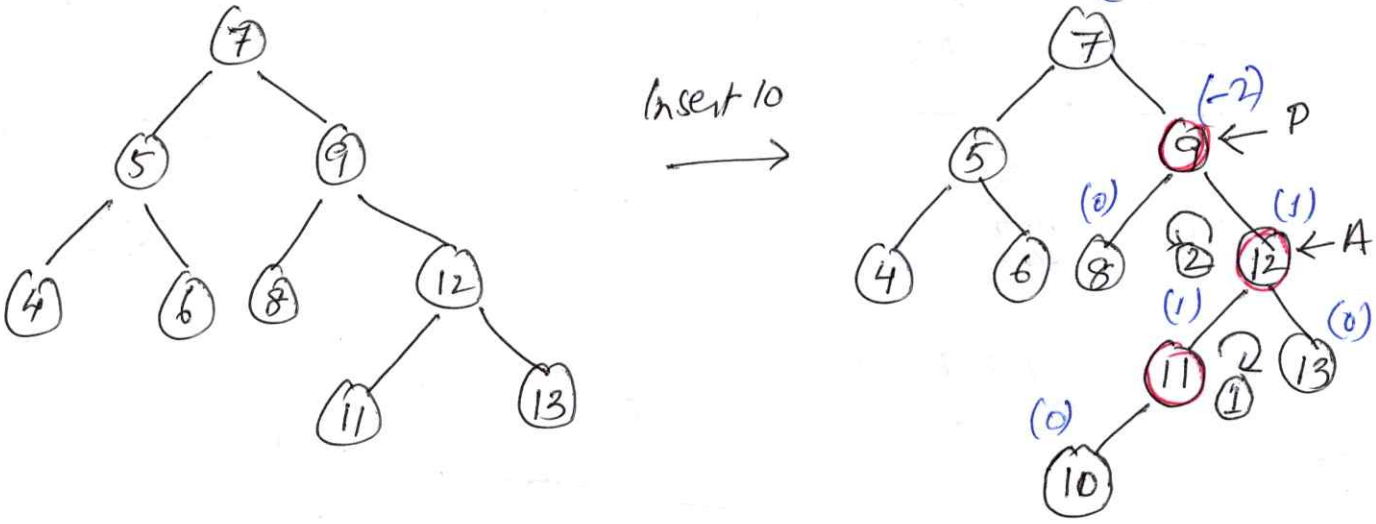
Rotation 1.

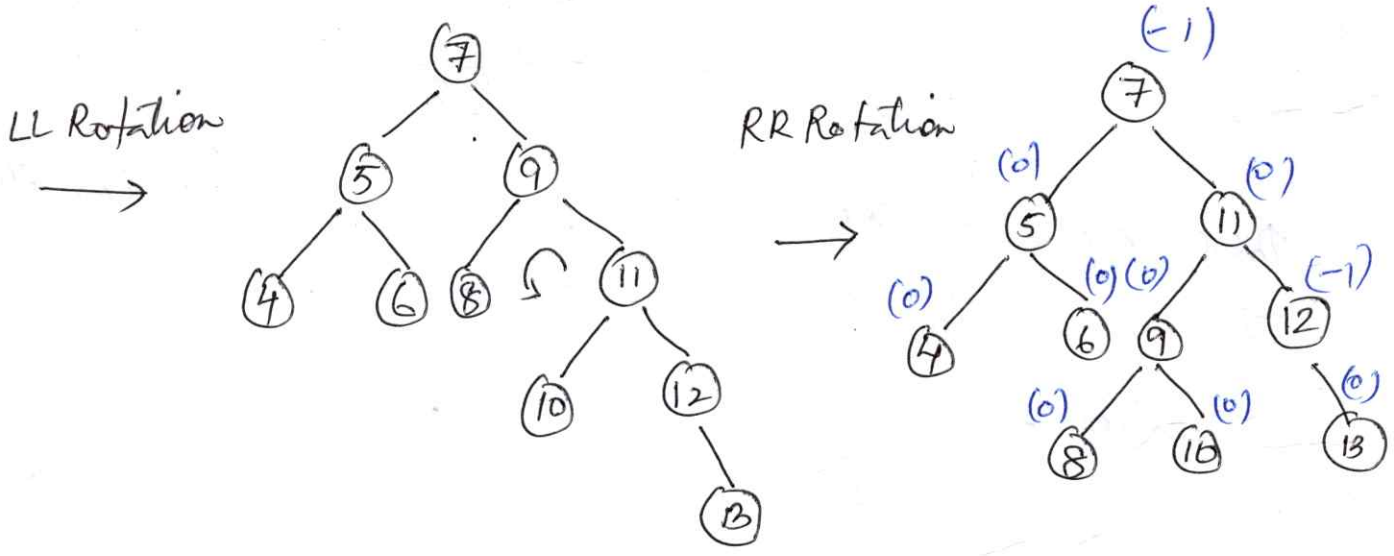
I. Right subtree (B_R) of the left child (B) of the right child (A) of the pivot node P becomes the left subtree of A.

II. Right child (A) of the pivot node (P) becomes the right child of B.

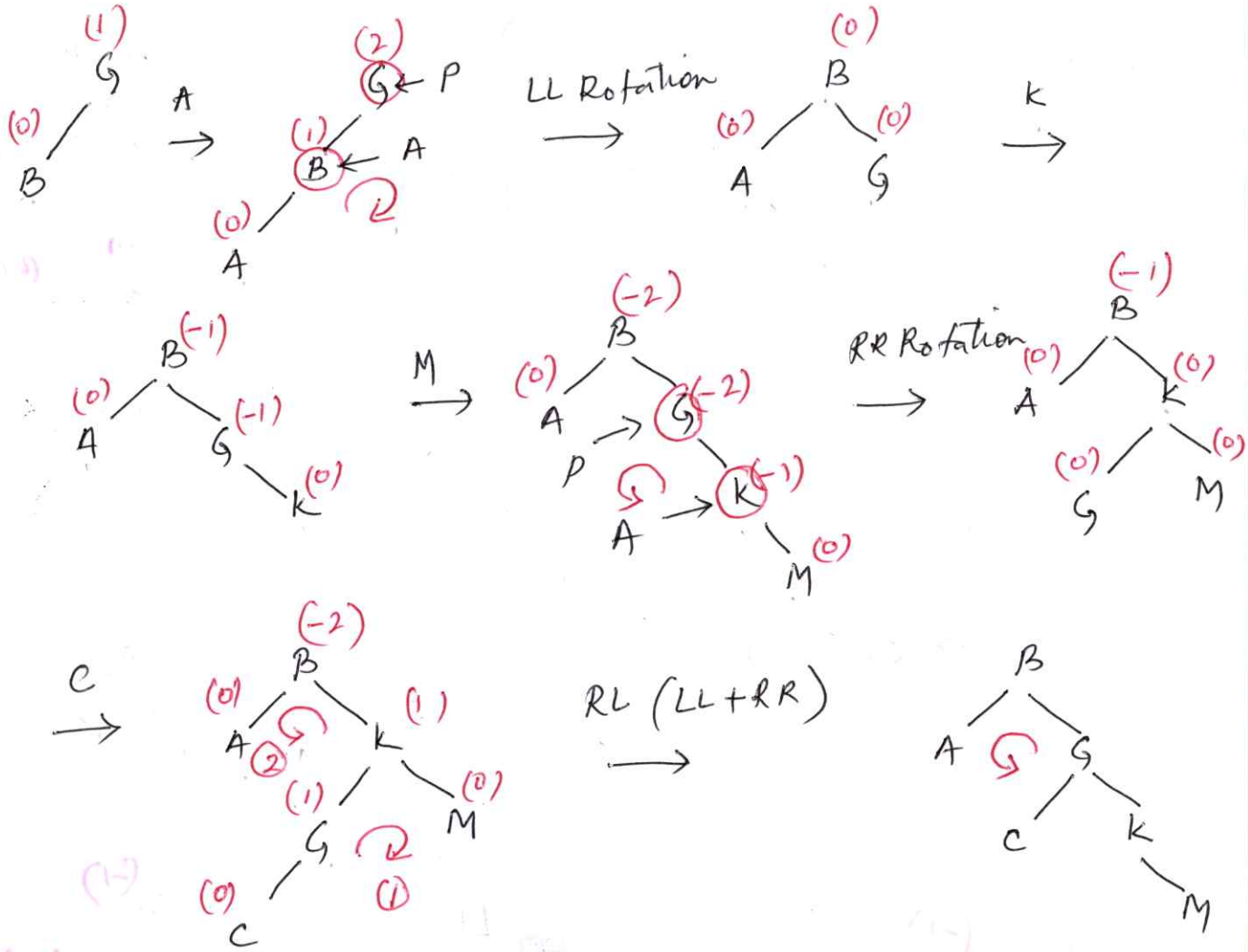
Rotation 2.

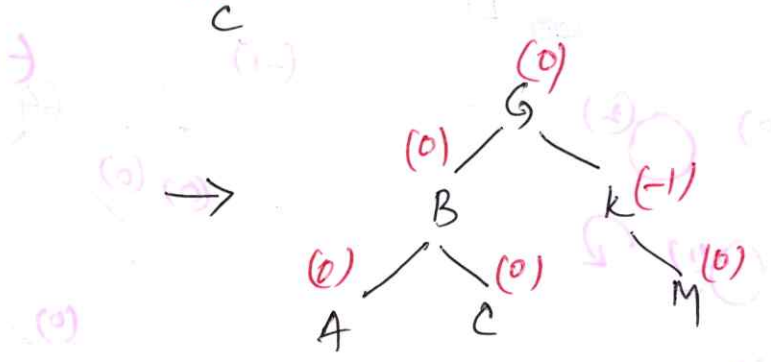
1. Left subtree (B_2) of the right child (B) of the right child (A) of the pivot node (P) becomes the right subtree of P.
2. P becomes the left child of B.





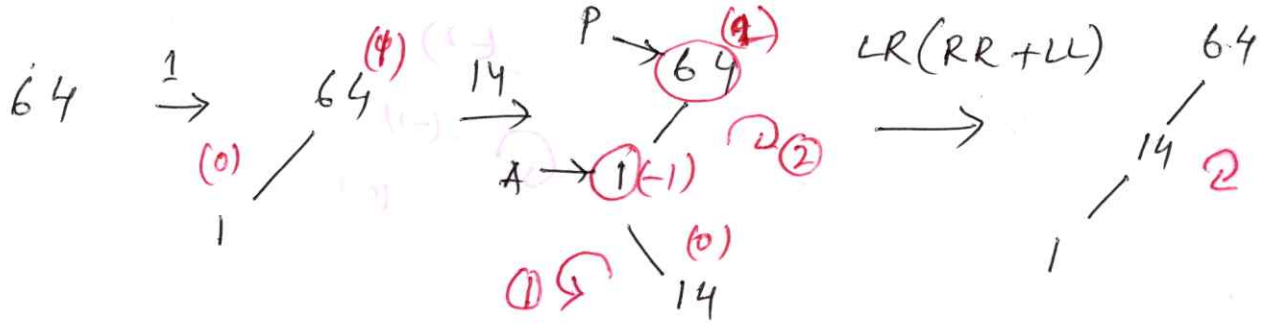
(*) Construct an AVL search tree with the following elements in the order of their occurrence:
G, B, 4, K, M, C.

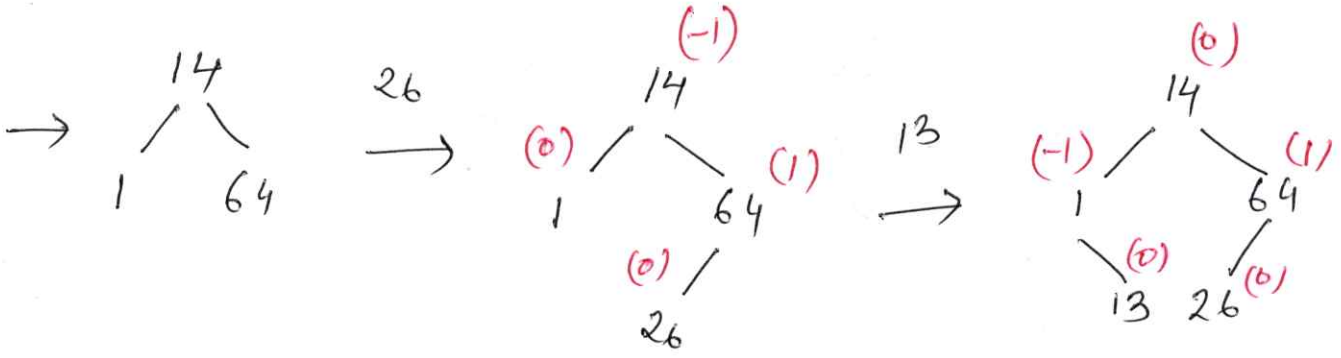


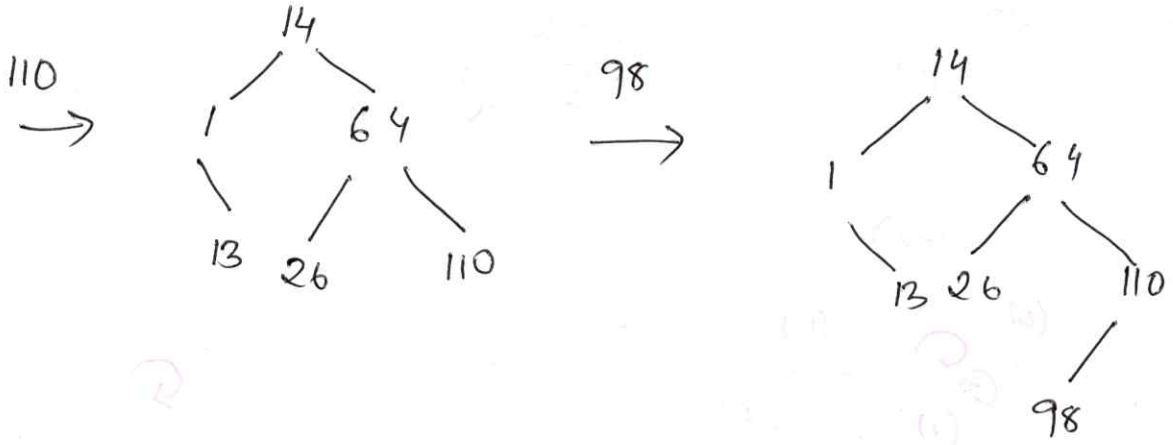


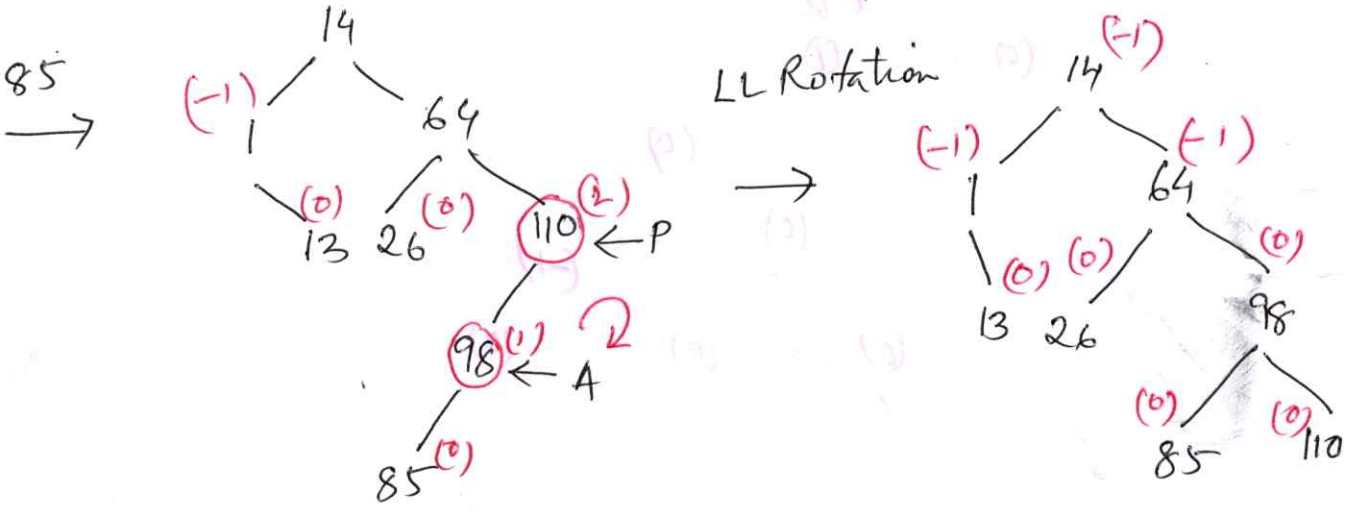
(*) Construct an AVL search tree by inserting the following elements in the order of their occurrence.

64, 1, 14, 26, 13, 110, 98, 85.

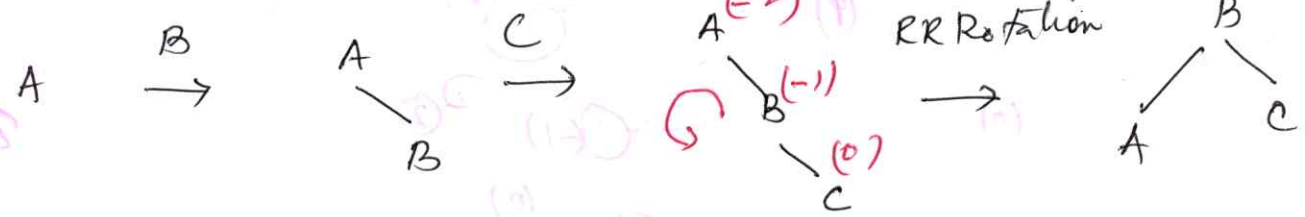


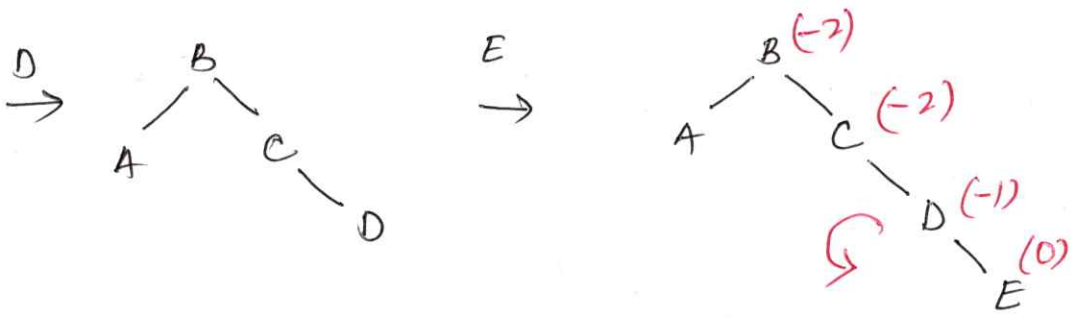




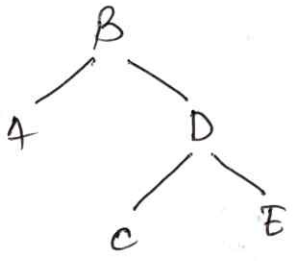


(*) Insert the following keys in the order shown to construct an AVL search tree: A, B, C, D, E.





RR Rotation →



m-Way Search Tree (Generalization of a BST).

If we relax the restriction that each node can have only one key, we can reduce the height of tree. An m-way search tree means a search tree with degree m. An m-way search tree T may be an empty tree. If T is non-empty, then it satisfies the following properties:

1. Each and every node of T has at least one key value and at most m-1 key values.

2. If in a node there are n keys $k_0, k_1, k_2, \ldots, k_{n-1}$ then $k_0 < k_1 < k_2 < \cdots < k_{n-1}$

3. All keys in left subtree of a key are less than it and all keys in right subtree of a key are greater than it.

4. Each of the subtrees of a node are also m-way search tree.

The following fig. shows an m-way search tree of order 3. The value N in the link part of every node denotes a NULL link.

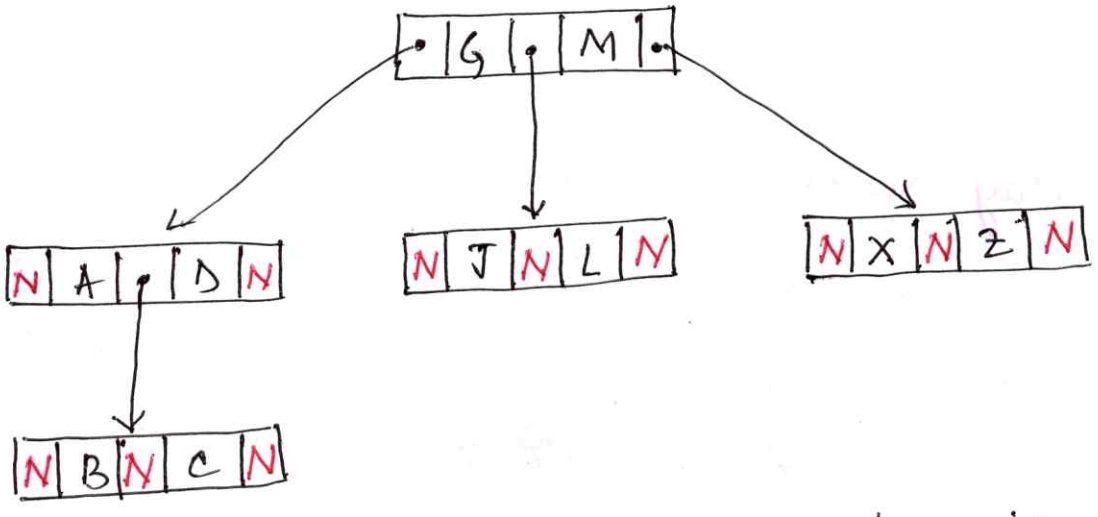


fig: An m-way search tree with $m=3$.

⇒ The height of an m-way search tree with n keys is $\log_m(n+1)$.

⇒ The maximum number of keys in an m-way search tree of height h is $m^h - 1$

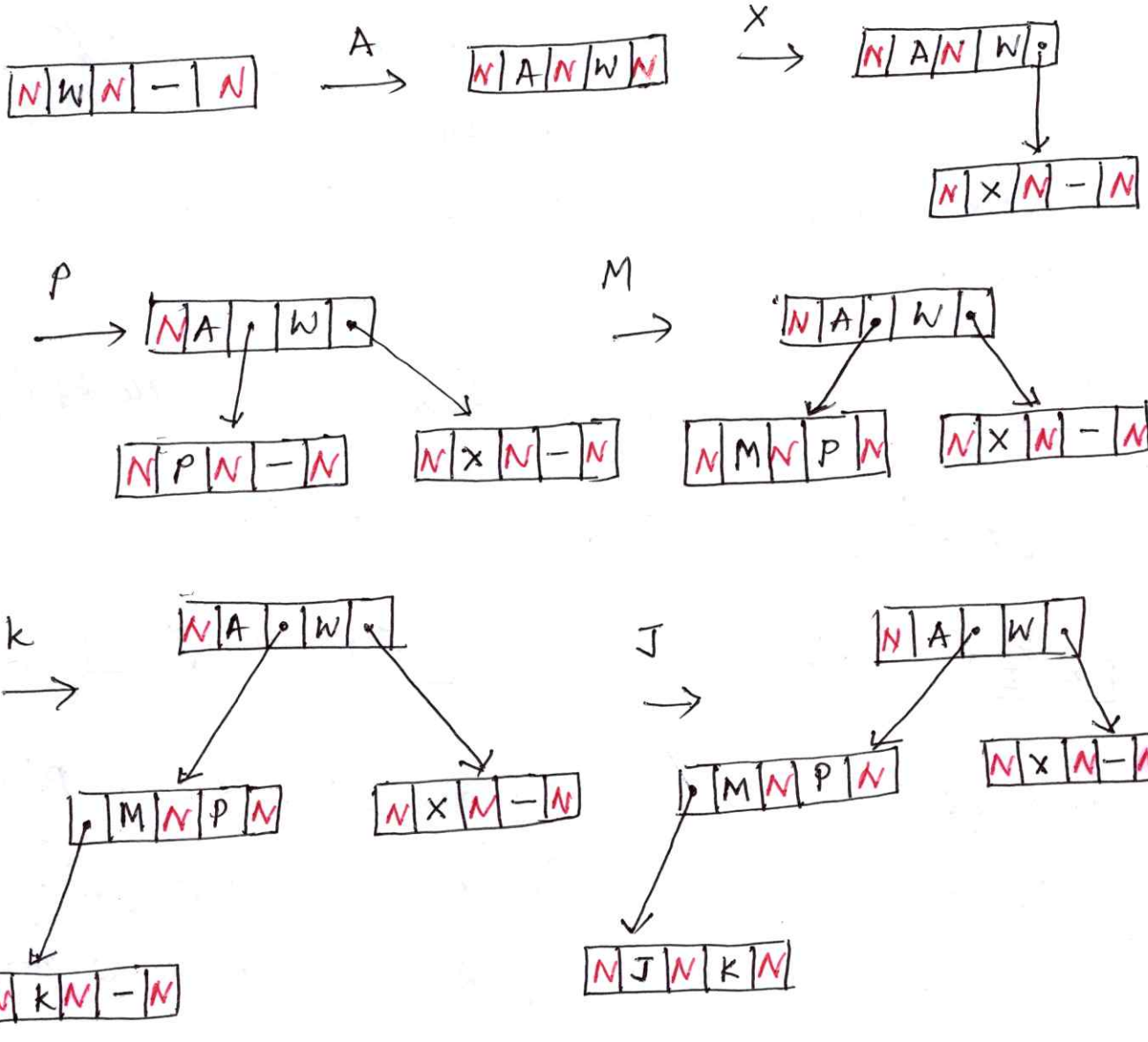
(*) Rules for insertion into m-way search tree.

First find the position of the node for the new key in the tree. If the node is not full then insert the key into appropriate position

in the node. For a full node, create a new node in proper place.

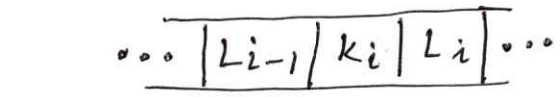
Example:

Insert the following keys into an m-way search tree of order 3. W, A, X, P, M, K, J.



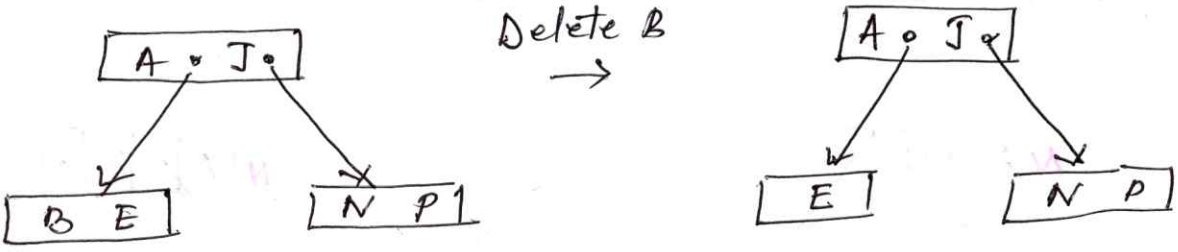
⊛ Rules for deletion from an m-way search tree.

Suppose key k_i is to be deleted from a node of an m-way search tree.

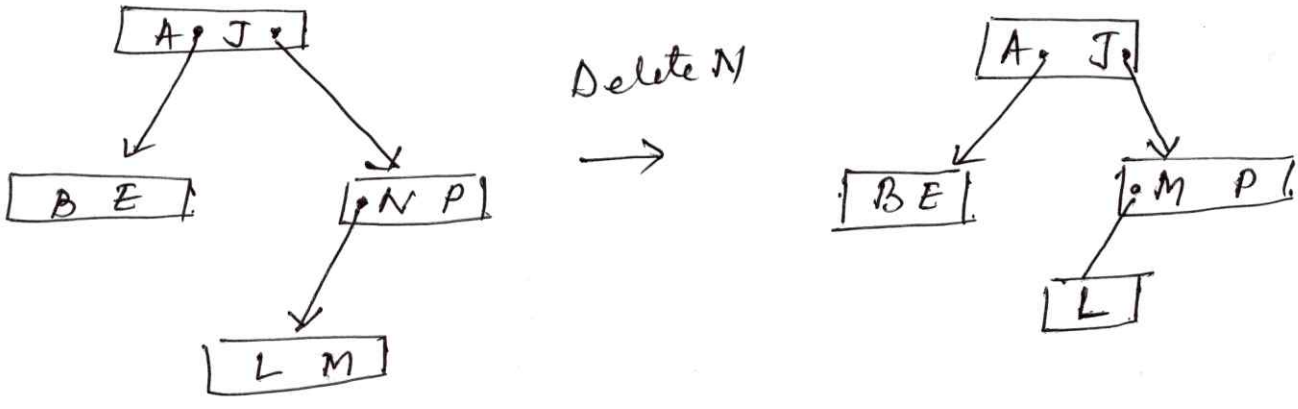


L_{i-1} and L_i are the addresses of the left child node and the right child node of k_i respectively. Now the following four conditions may come into consideration.

Case 1: L_{i-1} and L_i are both NULL. Delete k_i simply.



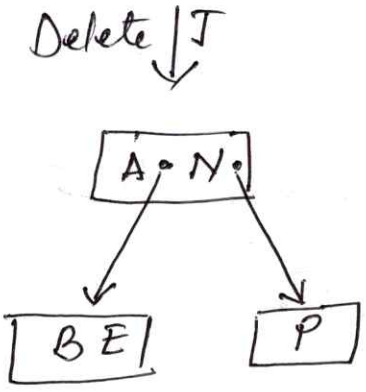
Case 2: L_{i-1} is not NULL and L_i is NULL. Pick the largest key value (k) from the left child node of k_i, delete key k and replace k_i with k.



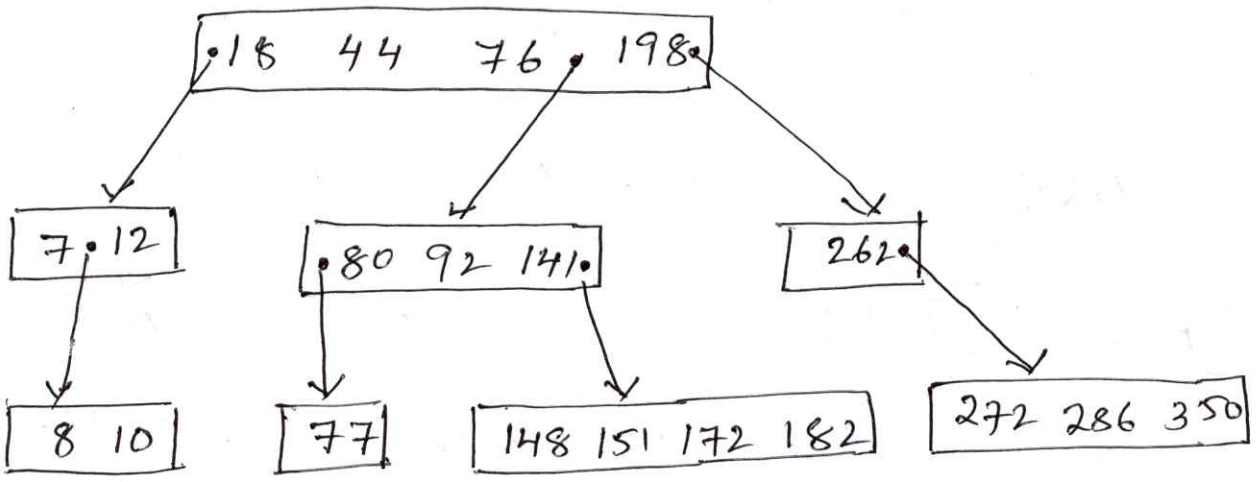
Case 3: L_{i-1} is NULL and L_i is not NULL. Pick the smallest key value (k) from the right child node of k_i, delete key k and replace k_i with k.

Case 4: L_{i-1} and L_i are both not NULL. Choose either the largest key value from the left child node of k_i or the smallest key value from the right child node of k_i, delete the key and replace k_i with the chosen key.

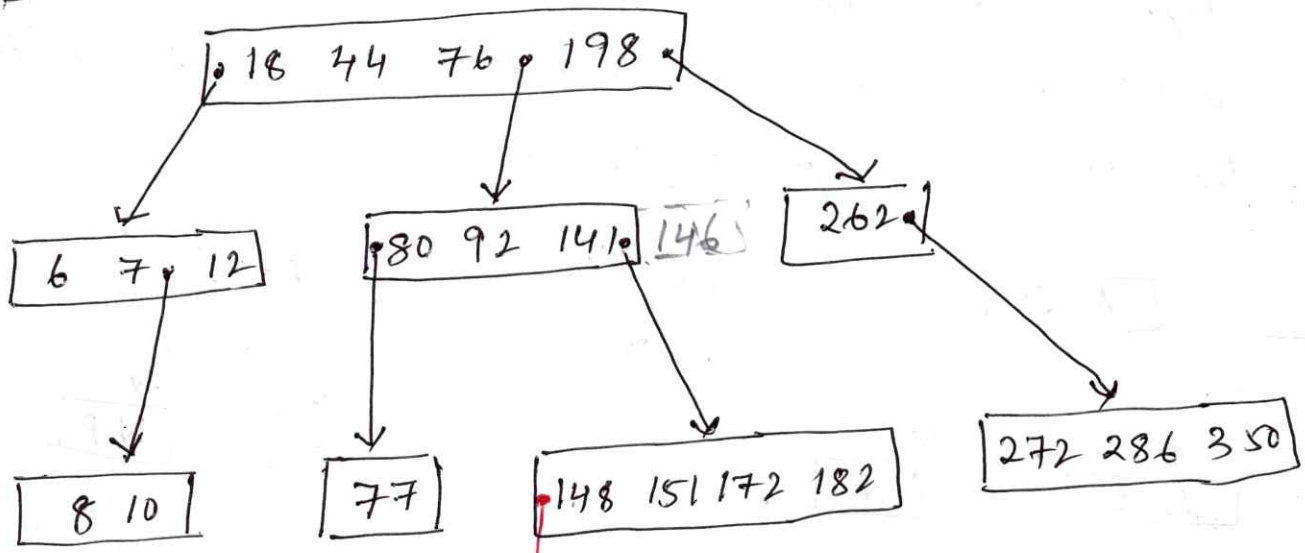
OR.



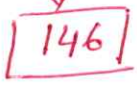
⊛ Insertion in an m-way search tree (5-way search tree).



Insert 6 :

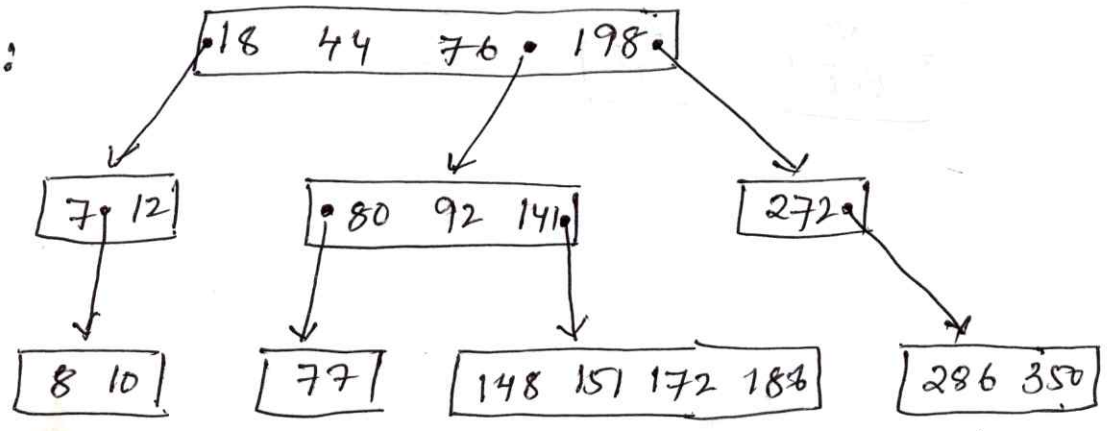


Insert 146 :

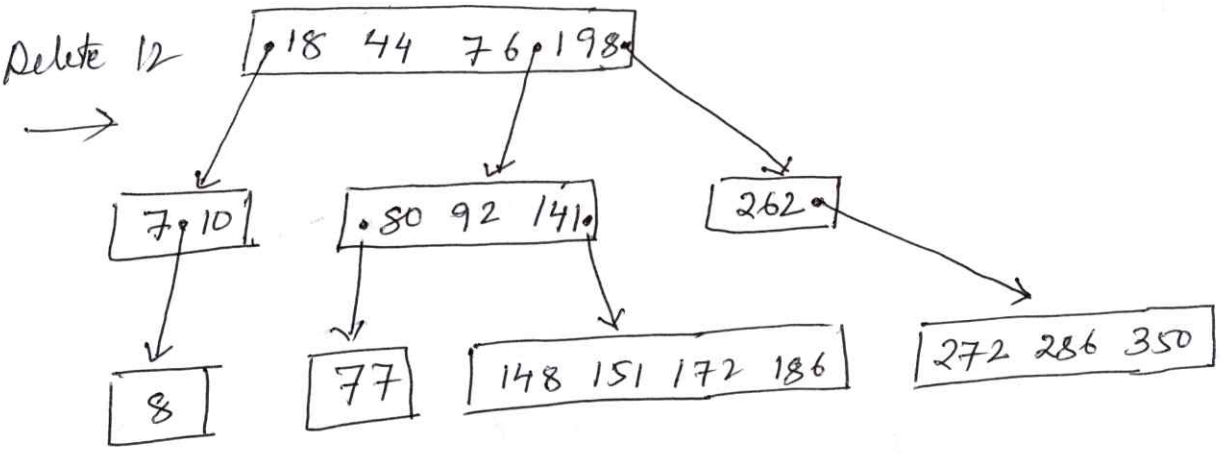


⊛ Deletion in an m-way search tree:

Delete 262 :

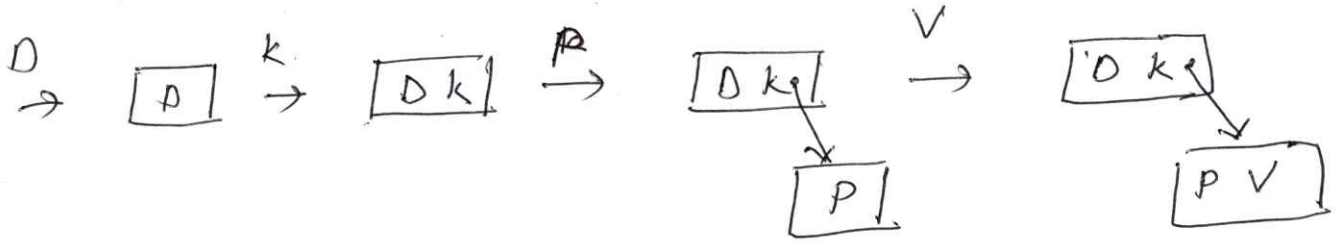


Delete 12



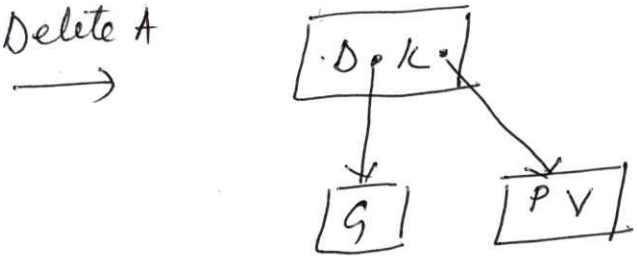
Example:

Construct a 3-way search tree with the following keys in the order shown: D, K, P, V, A, G.

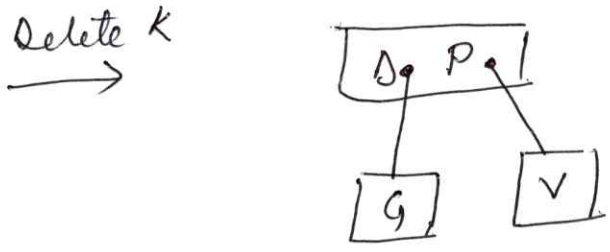


* Delete A and K from the above tree.

Delete A



Delete K



Q.

(*) BST Implementation

(*) Level wise printing of a tree.

(*) Construct an AVL Search Tree with the following elements in the order of their occurrence:
DEC, NOV, OCT, SEP, AUG, JUL, JUN, MAY, APR, MAR, FEB, JAN.

```
//Binary search tree implementation in C
//-----------------------------------------------------
#include<stdio.h>
#include<stdlib.h>

struct node
{
  int key;
  struct node *left, *right;
};

// A function to create a new BST node
struct node *newNode(int item)
{
  struct node *temp =  (struct node *)malloc(sizeof(struct node));
  temp->key = item;
  temp->left = temp->right = NULL;
  return temp;
}

// A function to do inorder traversal of BST
void inorder(struct node *root)
{
  if (root != NULL)
  {
    inorder(root->left);
    printf("%d ", root->key);
    inorder(root->right);
  }
}

/* A function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key)
{
  /* If the tree is empty, return a new node */
  if (node == NULL) return newNode(key);

  /* Otherwise, recur down the tree */
  if (key < node->key)
    node->left  = insert(node->left, key);
  else
    node->right = insert(node->right, key);

  /* return the (unchanged) node pointer */
  return node;
}

/* Given a non-empty binary search tree, return the node with minimum key value found in that
tree. Note that the entire tree does not  need to be searched. */
struct node * minValueNode(struct node* node)
{
```

```
  struct node* current = node;

  /* loop down to find the leftmost leaf */
  while (current->left != NULL)
    current = current->left;
   return current;
}

/* Given a binary search tree and a key, this function deletes the key and returns the new root */
struct node* deleteNode(struct node* root, int key)
{
  // base case
  if (root == NULL) return root;

  // If the key to be deleted is smaller than the root's key, then it lies in left subtree
  if (key < root->key)
    root->left = deleteNode(root->left, key);

  // If the key to be deleted is greater than the root's key, then it lies in right subtree
  else if (key > root->key)
    root->right = deleteNode(root->right, key);

  // if key is same as root's key, then This is the node to be deleted
  else
  {
    //Case I:No child
    if (root->left == NULL && root->right==NULL)
    {
      free(root);
      root=NULL;
    }
    // Case II: Node with only one child
    else if (root->left == NULL)
    {
      struct node *temp = root->right;
      free(root);
      return temp;
    }
    else if (root->right == NULL)
    {
      struct node *temp = root->left;
      free(root);
      return temp;
    }

    // Case III: Node with two children: Get the inorder successor (smallest in the right subtree)
    else
    {
      struct node* temp = minValueNode(root->right);
     // Copy the inorder successor's content to this node
     root->key = temp->key;
```

```
        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
        }
    }
  return root;
}

void main()
{
    struct node *root = NULL;
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    printf("Inorder traversal of the given tree \n");
    inorder(root);

    printf("\nDelete 20\n");
    root = deleteNode(root, 20);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 30\n");
    root = deleteNode(root, 30);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);

    printf("\nDelete 50\n");
    root = deleteNode(root, 50);
    printf("Inorder traversal of the modified tree \n");
    inorder(root);
    getch();

}
```