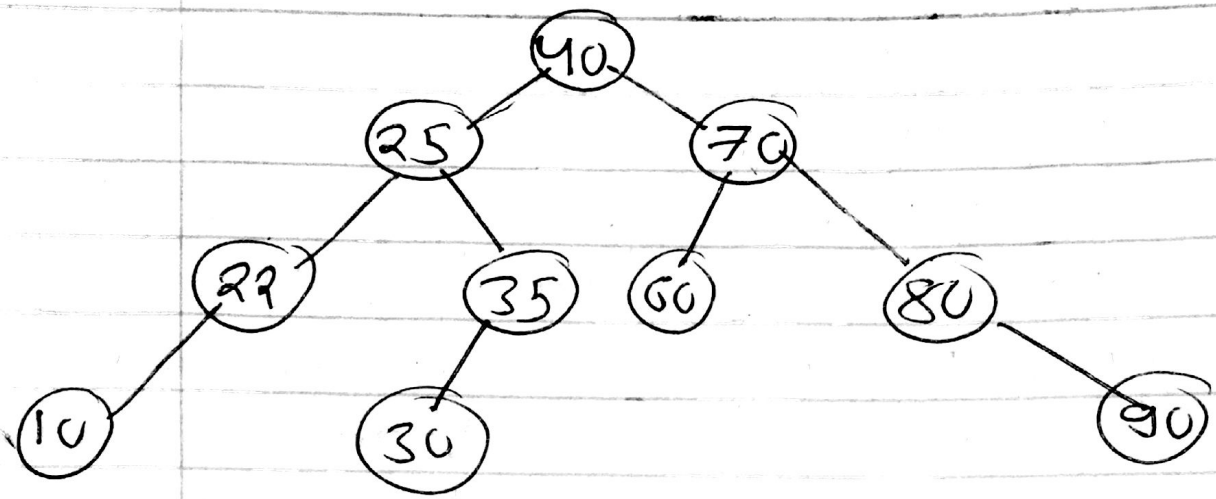


Fig: BST

iii) Deletion from BST : steps ↓

– To delete a node with particular key, first we need to perform a search. Once we find the node to be deleted, we need to consider the following possibilities

a) if the node is leaf, it can be deleted immediately
   – we just set the left and right pointers of parent to NULL

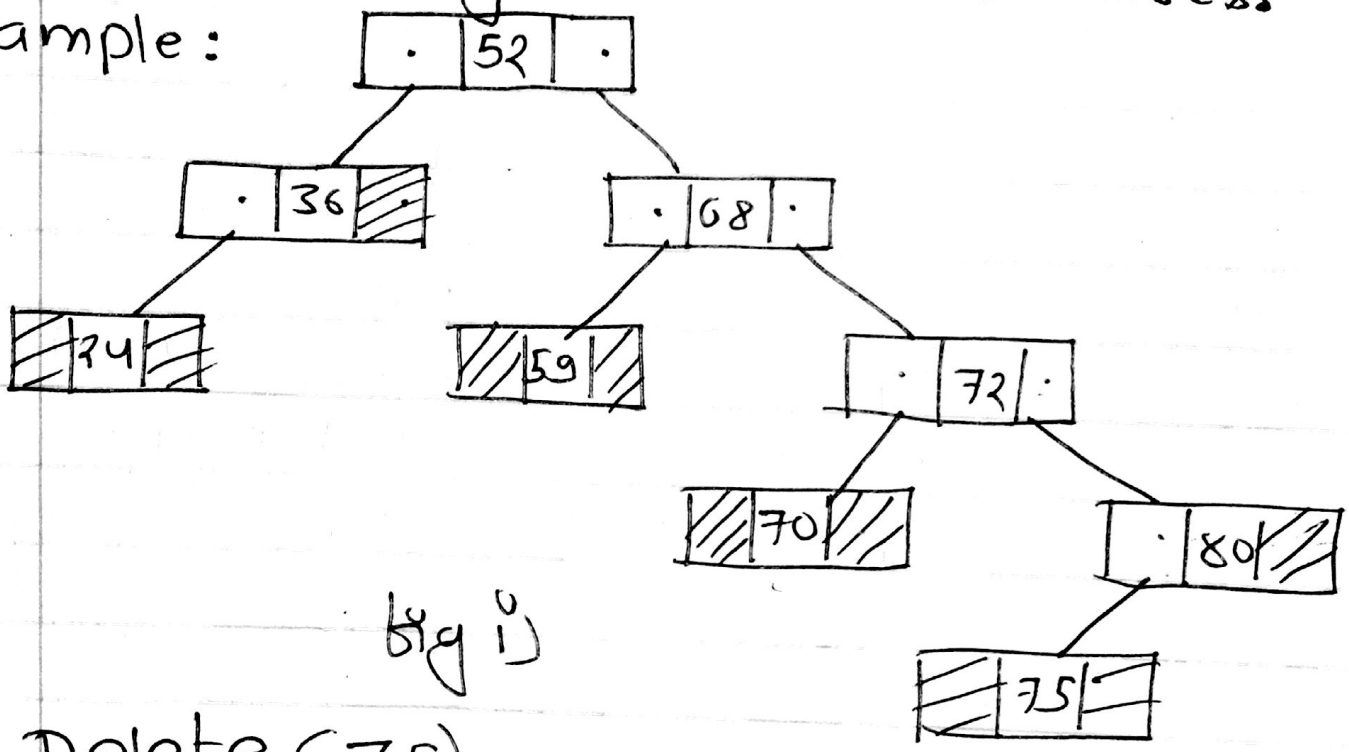
b) if the node has one child
   – In this case, we redirect its pointer to child to its pointer in parent.

imp

Inorder successor or Inorder precedor (92)

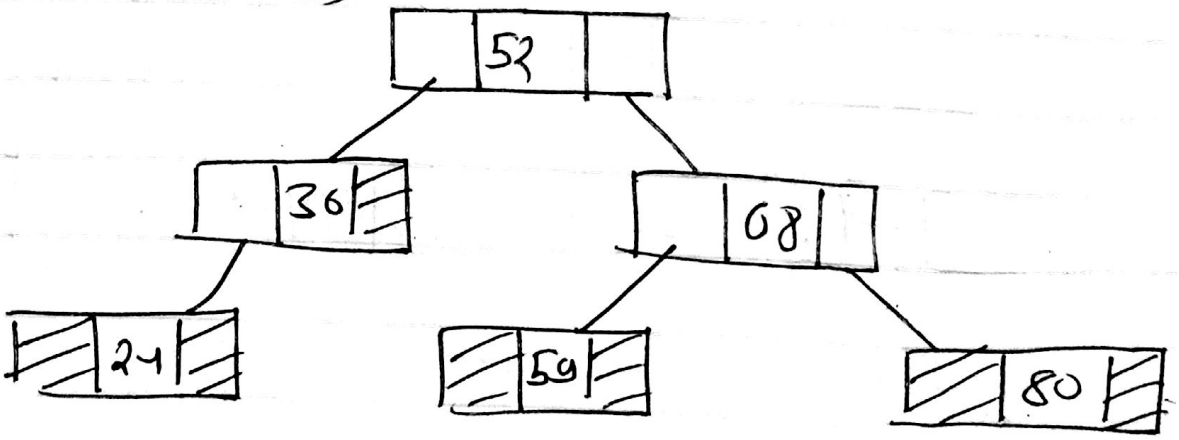
c) if the node has two children,
— in this case, we replace the node with the right most node of leftsubtree or leftmost node of right subtree and delete the node, using the above two cases.

example:

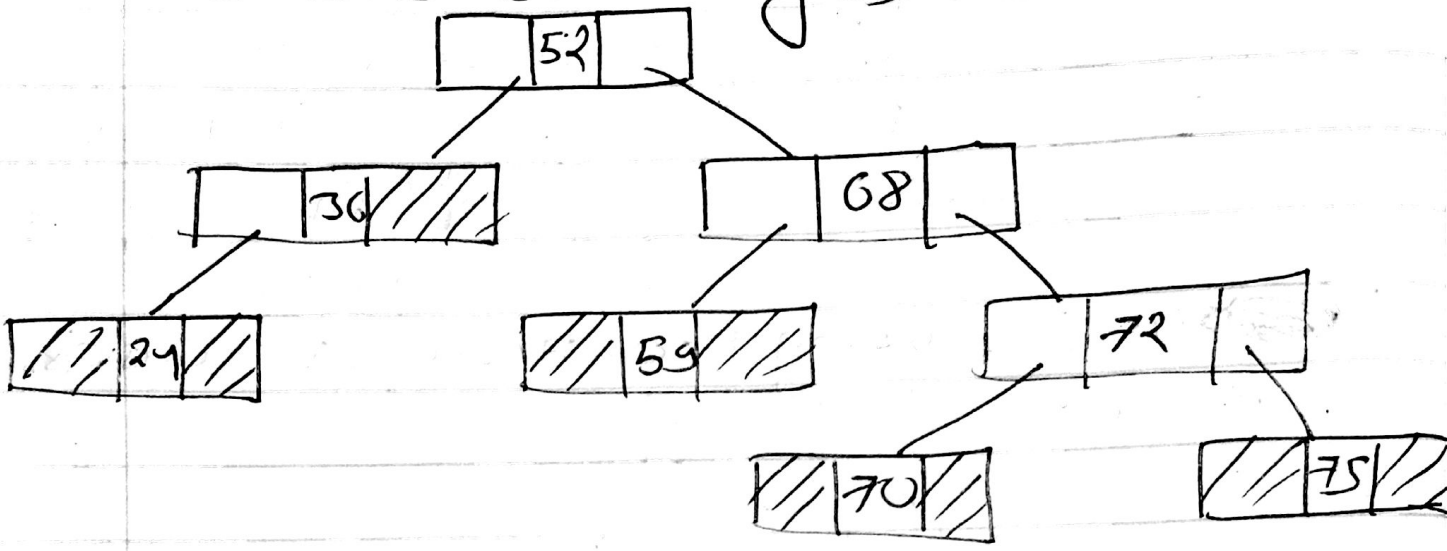


(or)

① Delete (75)

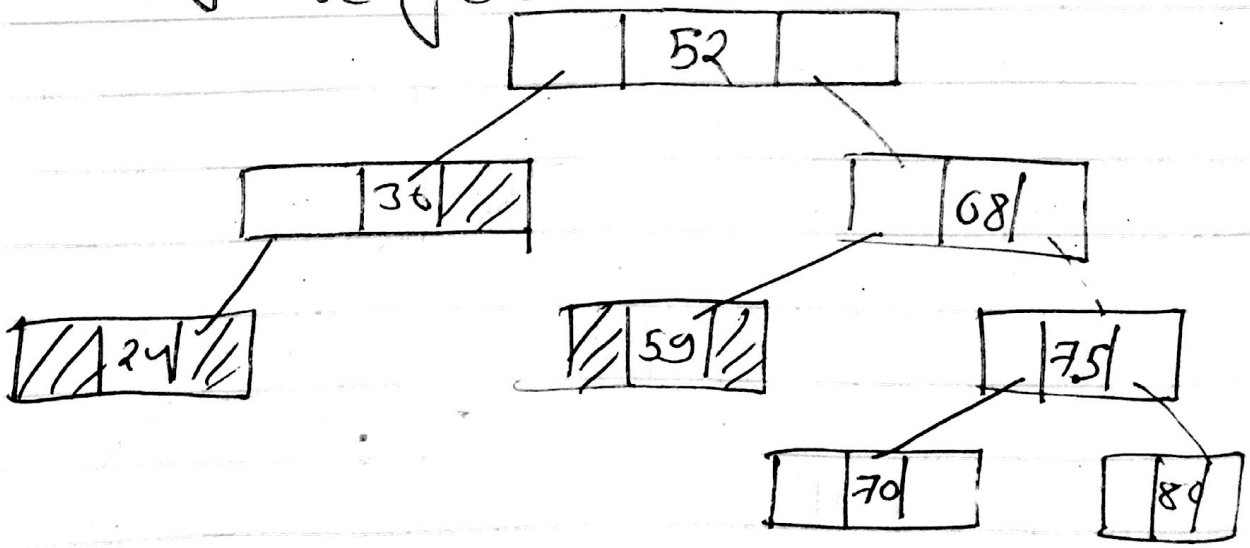


ii) Delete 80 in big (i)

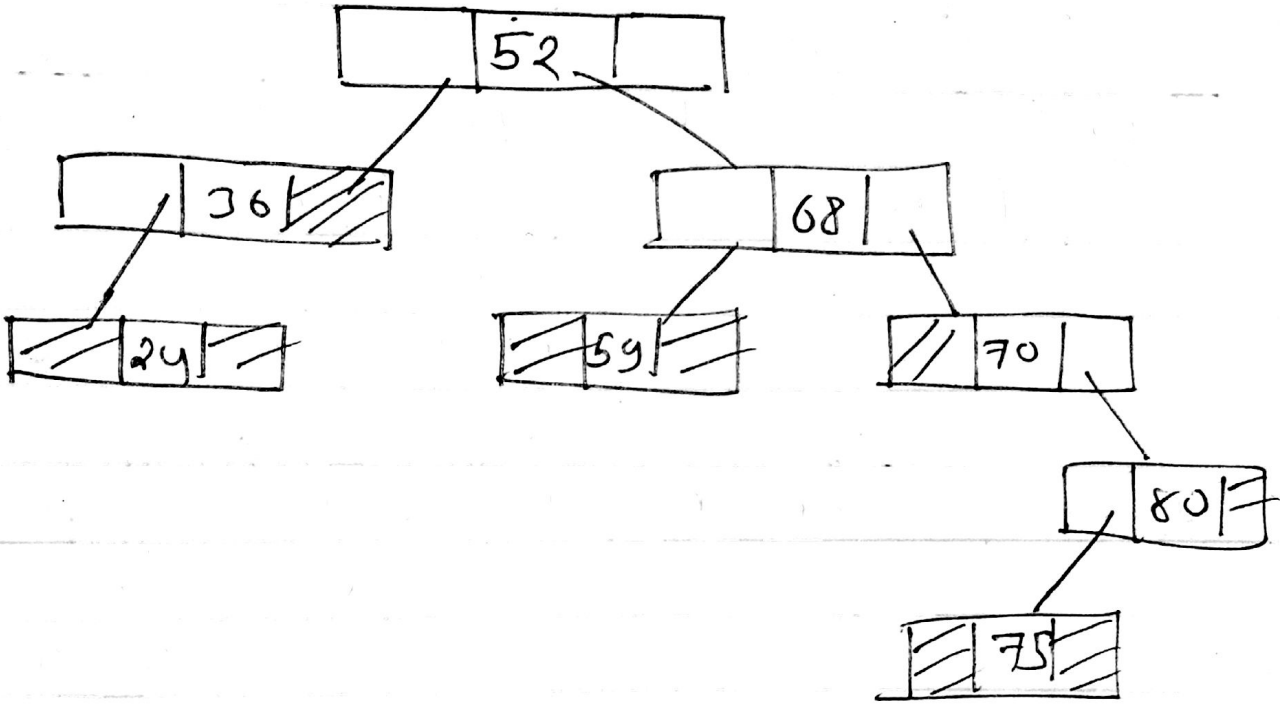


iii) Delete 72 in big i)

- Replacing 72 with 75 i.e 75 is the leftmost node of leaf right subtree, and deleting 75 we get.

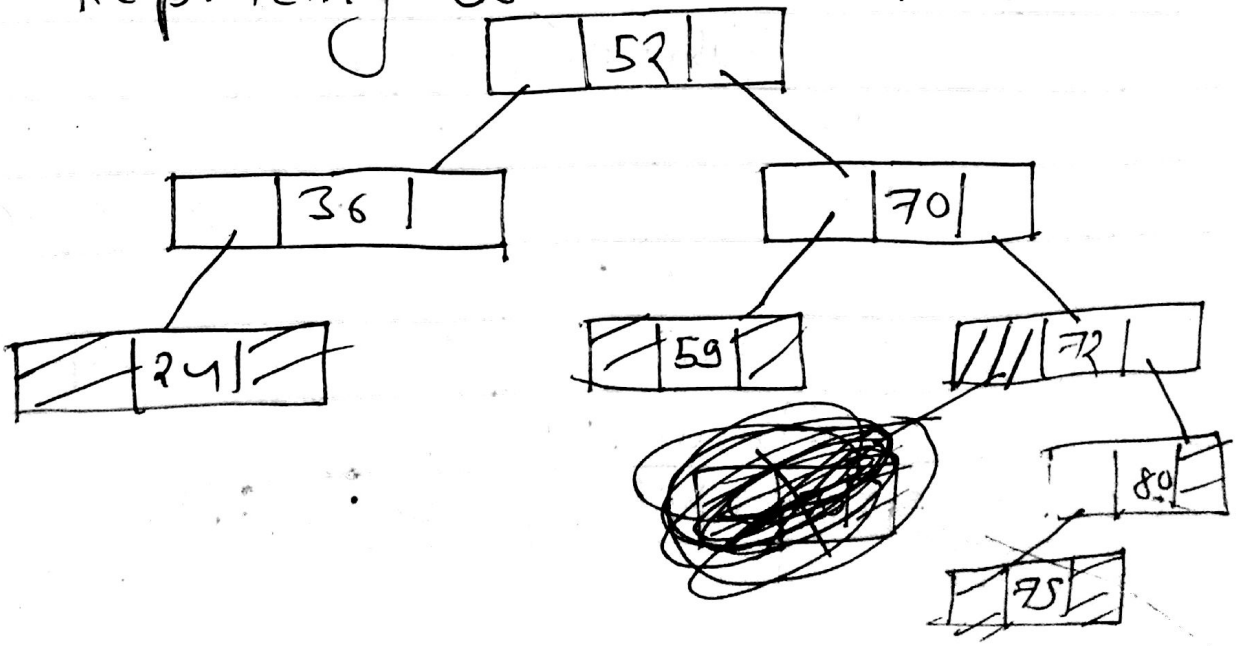


Also, we can delete it by replacing 72 with the right most node of left subtree i.e 70 and deleting leaf 70.



iv) Delete 68. in big i

— Replacing 68 with 70.



iii) Searching a Node in BST

The search operation in BST refers to the process of searching for a specific value in the tree.

Algorithm

1. Input the DATA to be searched.
2. Make current node point to root node.
3. If current node is null
   a) Display 'Not found'
   b) Exit
4. Compare the value to be searched with the value of current node.
   a) If value equal to current node value
      i) Display 'found'
      ii) Exit
   b) If value less than current node value
      i) Make current node point to left child
      ii) Go to step 3
   c) If the value is greater than current node value
      i) Make current node point to right child.
      ii) Go to step 3

C-implementation

```
Void Search(int data, Node *root)
{
 Node *current=root;
while(data != current->info)
  {
   if(data==current->info)
     {
     printf("Data found in tree");
      return;
     }
    if(current==NULL)
       {
       printf("Data Not found");
        }
   if(data> current->info)
        current=current->Rchild;
  else if(data< current->info)
        current=current->Lchild;
  }
}
```

Extended Binary Trees

A binary tree can be converted to an extended binary tree by adding a new node to its left node and to the nodes that have only one child. These new nodes are added in such a way that all the nodes in the resultant tree have zero or two children. The extended bree is also called as 2-tree. The nodes of the original tree are called internal nodes and the new nodes that are added to the binary tree, to make it an extended binary tree are called external nodes

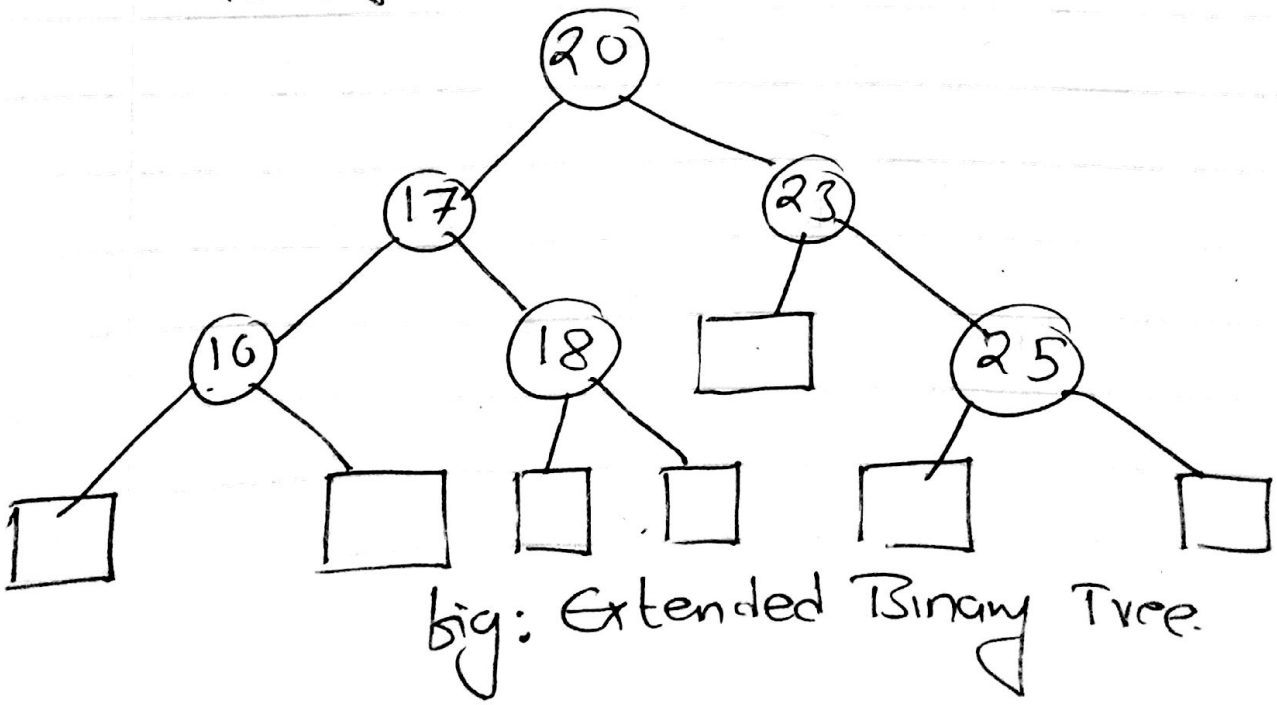


fig: Extended Binary Tree.

Threaded Binary Tree / Stackless DFS traversal

- In linked list representation of BST, there are many nodes that have an empty left child or empty right child or both. So the main objective of this tree is to make effective use of these null pointers.

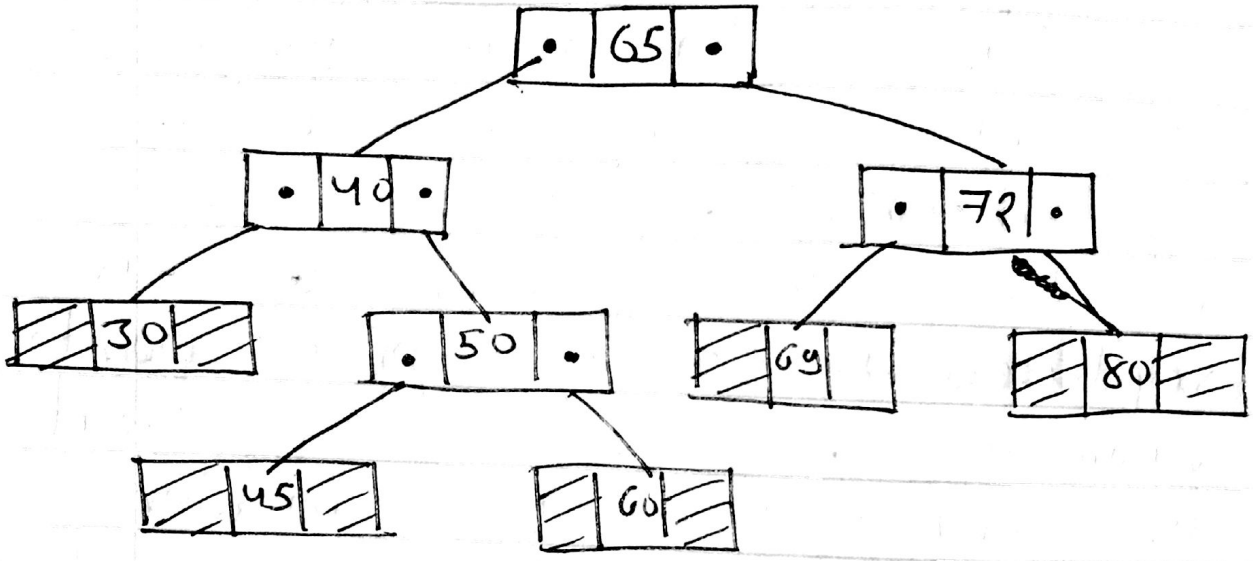
Here all the null pointers are replaced by the appropriate pointer value called thread. A field that holds the address of the inorder successor or inorder predecessor of a node is called Thread. A binary tree that implement thread is called threaded binary tree.

The NULL pointer replacement strategy is as below

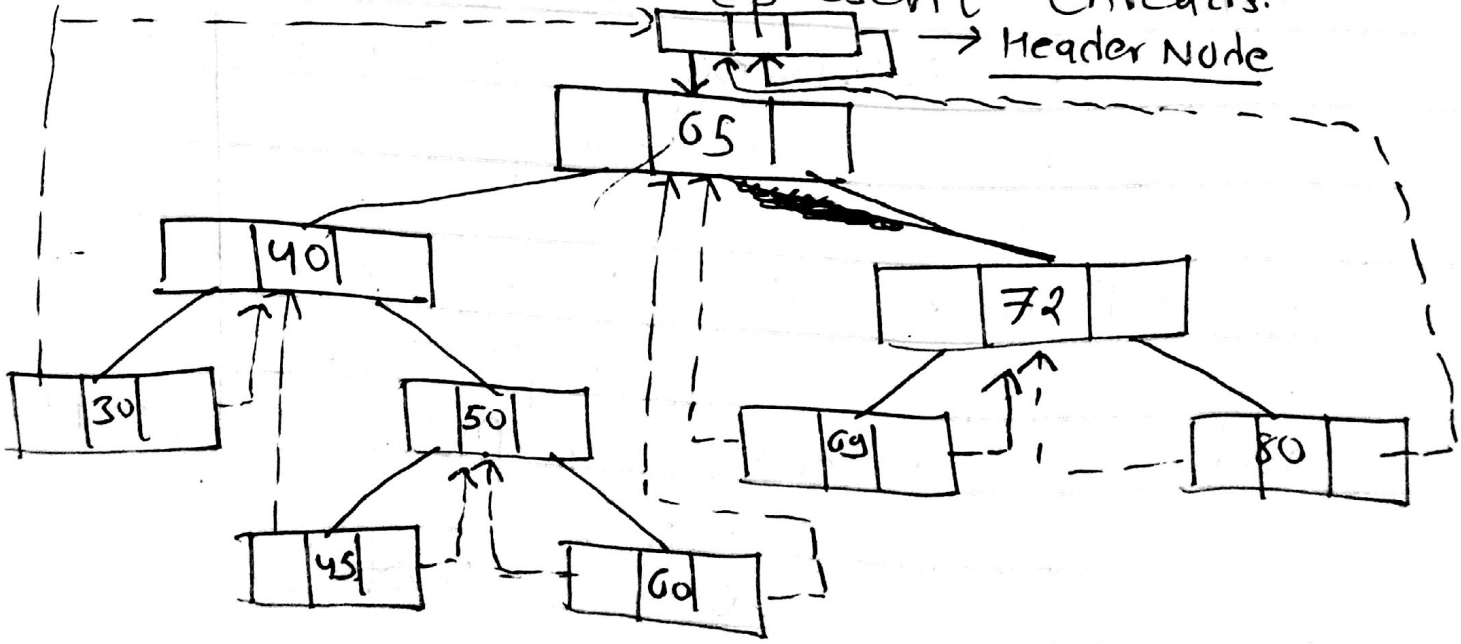
If the left child of a node P is equal to NULL, then this link is replaced by a pointer to the node which immediately precedes node P in inorder traversal. If right child of a node P is equal to NULL than this

link is replaced by a pointer to its inorder successor node.

eg: Consider the following BST



The below figure is threaded binary tree for above figure where dotted line represent threads.



Note: For header node, (QE)
- The Lchild points to its left subtree.
- The Rchild point to itself.

In above BST, the nodes are visited in inorder as

30 40 45 50 60 65 69 72 80

Now in case of threaded binary tree,

- The Lchild of 30 is Null so it contain pointer to inorder preceeder. 30 has no ~~preceeder~~ predecessor so it points to dummy node called header node.
- The Rchild of 80 is NULL so it should contain pointer to inorder successor 80 has no inorder successor so it also point to dummy node called header.
- The Lchild of 45 point to inorder ~~preceder~~ predecessor i.e 40 and Rchild of 45 point to inorder sucessor i-e 50.
- And similar for rest of the node having empty left and right child.

Advantage of threaded binary tree.

(1) Traversal in threaded binary tree is fast because it doesn't require to implement recursion to traverse the tree.

SBT : Size of leaf is zero. Size of internal node is size of sum of two children + 1. Based on the size, we define weight as weight[n] = size or size+1 #

② Since no recursion is implemented so it doesn't require to maintain stack hence it provides less memory usage.

Balanced Binary Tree

A balanced binary tree is one in which the largest path through the left subtree is the same length as the largest path of the right subtree. i.e from root to leaf. Searching time is very less in case of balanced binary tree.

Types

① Height balanced tree

In height balanced tree, balancing the height is important factor. One of the popular algorithm for constructing a height balanced tree is AVL tree.

② Weight balanced tree

In weight balanced tree there is an additional field called weight field or probability field in

addition to data element, left pointer, right pointer as that in other tree.

When the tree is created, the node with the highest probability of access are placed at the top. The root node contains highested weighted node. The tree is said to be weight balanced if the weight in the right and left subtrees are as equal as possible.

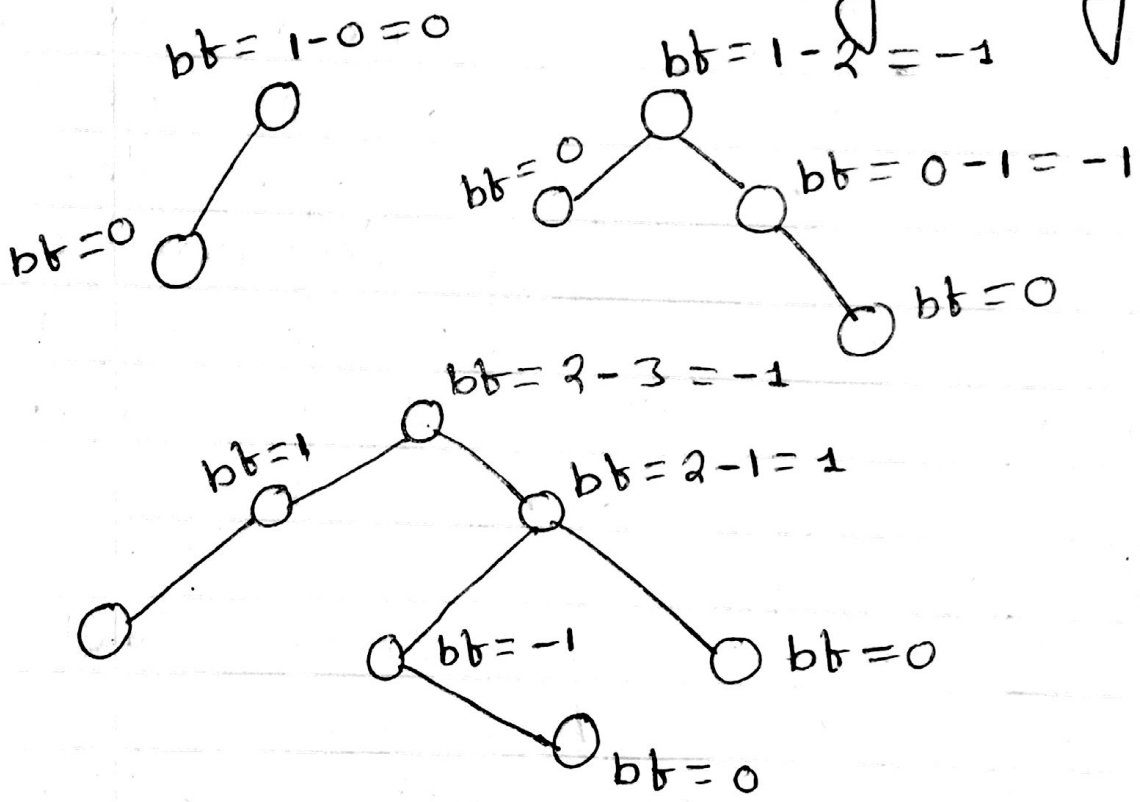
So the weight balanced tree provide lowest search time for the node that is most likely to be accessed.

AVL tree (Admissible tree) discoveries

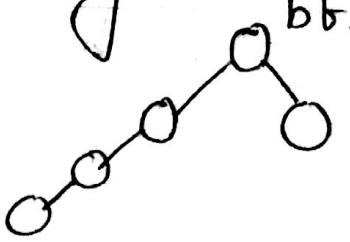
In the name of, Adelson Velskii and Landis, AVL, AVL tree is a self balancing binary search tree. In an AVL tree, the height of the two child subtrees of any node differs at most by one. i.e height of left subtree and height of right subtree differ at most by 1. It is also called as height balanced tree.

The balance factor (bf) of a node is the height of left subtree minus the height of its right subtree i.e

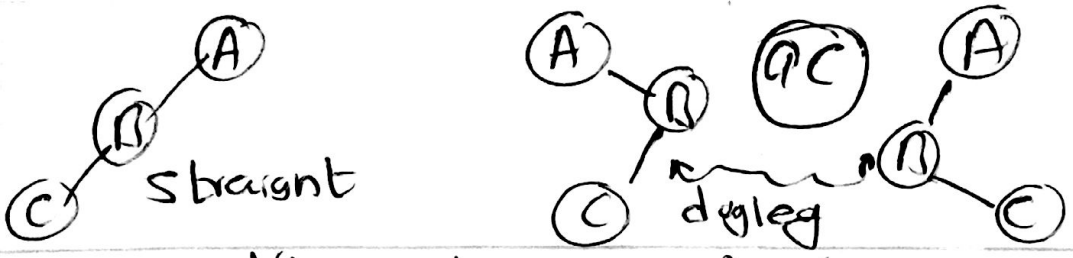
bf = height of left subtree − height of right subtree.



bf of node in an AVL tree should be either 0, 1 or -1. A node with any other bf is considered to be unbalanced and require rebalancing of the tree.



is not AVL tree.



Insertion of a Node in an AVL tree.

1) Insert the node in the same way as in BST.
2) Check the balance factor of each node after insertion.
3) If we find a node with imbalance ~~then~~ i.e bf is other then 0, 1 or -1 then stop insertion.
4) If these three nodes are in straight line then apply single rotation.
5) If these three nodes are in dog leg pattern then apply double rotation.

imp

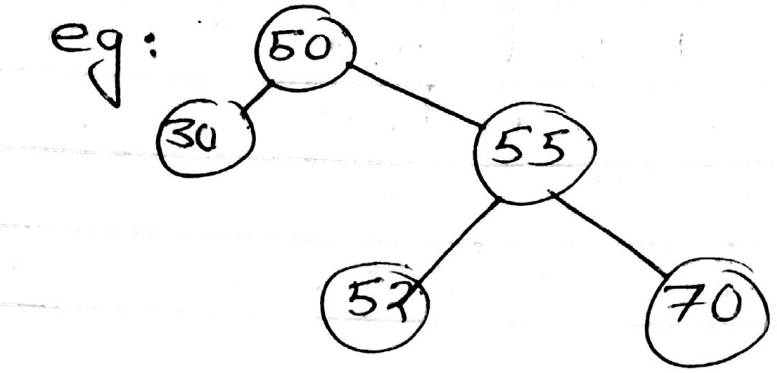
Rotation type

When an insertion of a node in an AVL tree create imbalances then in order to balance a tree, we have to perform rotations.

① Single left Rotation (SLR)

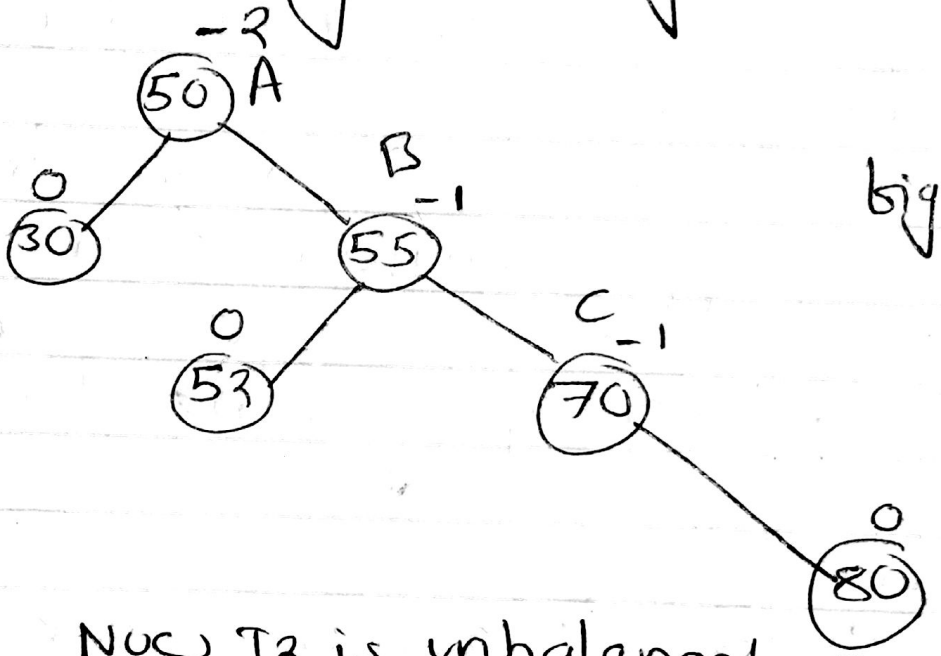
– Unbalanced is occured due to insertion in the right subtree of right child of the pivot node. Balancing can be done by applying SLR on three nodes.

eg:



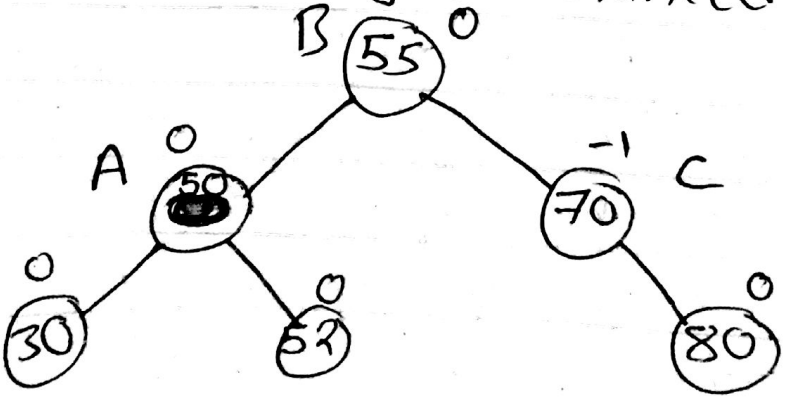
big : T1

Inserting 80 we get



big : T2

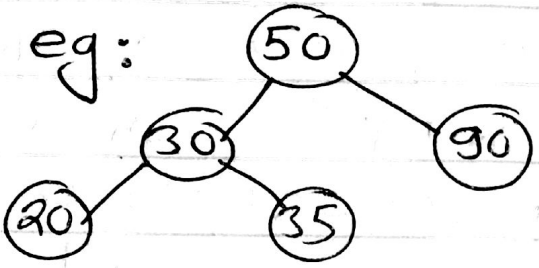
Now T2 is unbalanced applying single left rotation on 50, 55 and 70 (i.e three nodes in straight line) then adding 80 we get balanced AVL tree.



(2) Single Right rotation. (SRR)

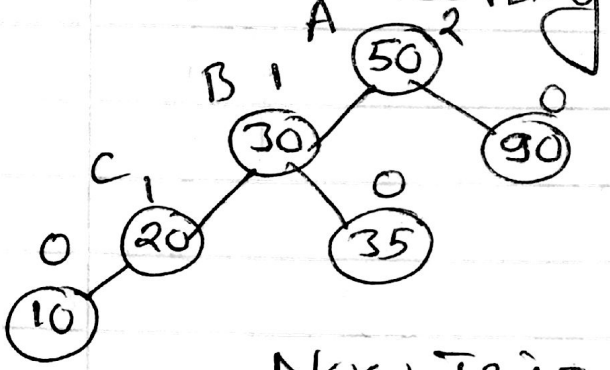
Unbalanced is occured due to insertion in left subtree of left child of the pivot node.

eg:



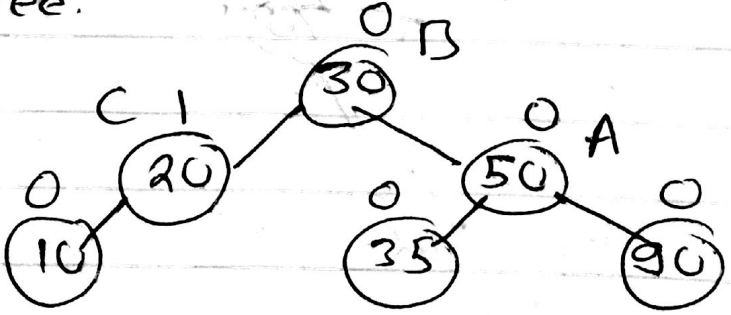
big: T1

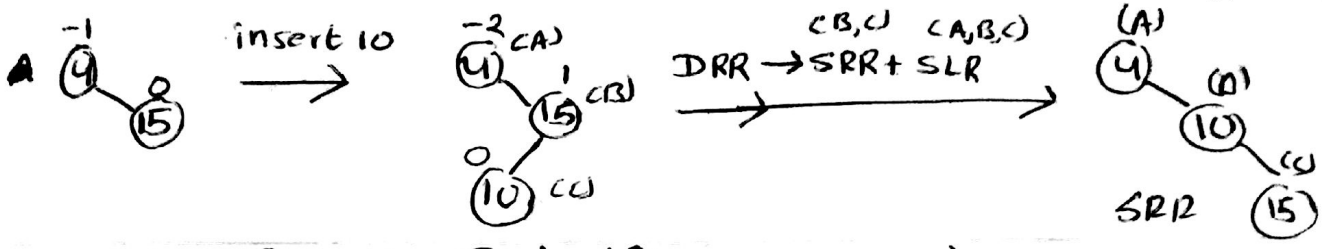
Now inserting 10, we get



big T2.

Now T2 is unbalanced, applying single right rotation on 50, 30 and 20 (i.e three nodes in straight line) then adding 10 we get balanced AVL tree.



eg: 

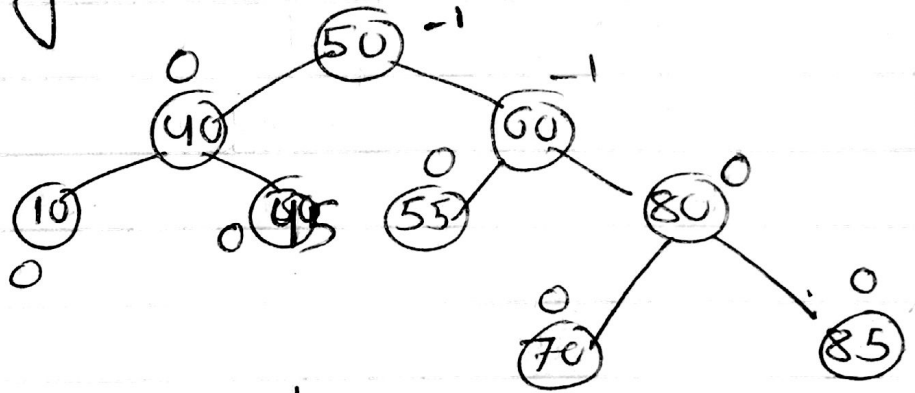
39

3

Double Right Rotation (DRR)

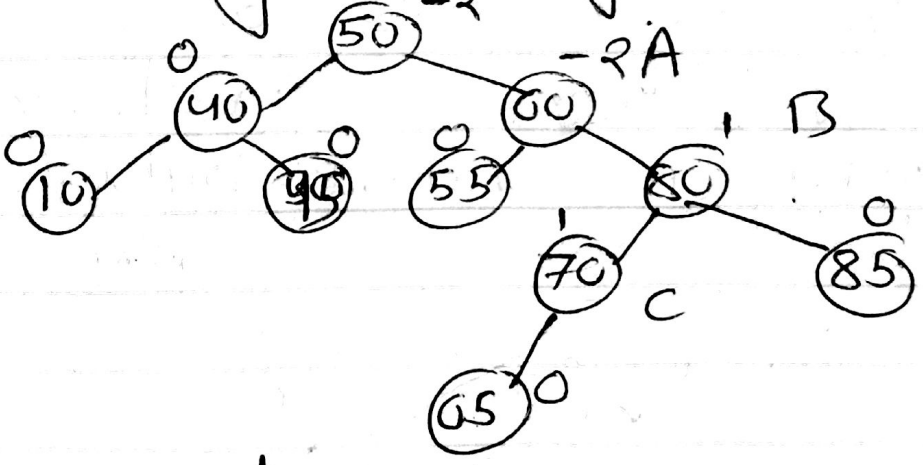
— Unbalanced is occured due to inserting a node in the left subtree of right child of pivot node. Balancing can be done by performing DRR i.e two single rotation which are in opposite direction: first SRR then SLR

eg: Consider the AVL tree as below.



big T1

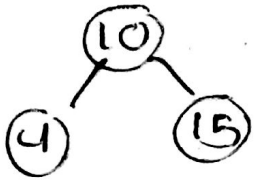
Inserting 65, we get



big T2

Now T2 is unbalanced, applying DRR on 60 80 and 70 C i.e three nodes in dog leg pattern).

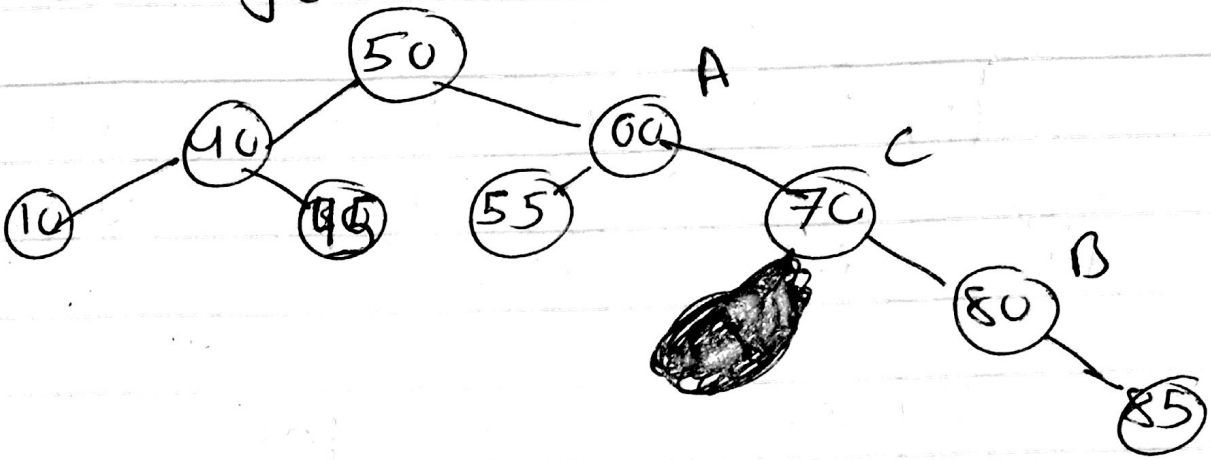
SLR →



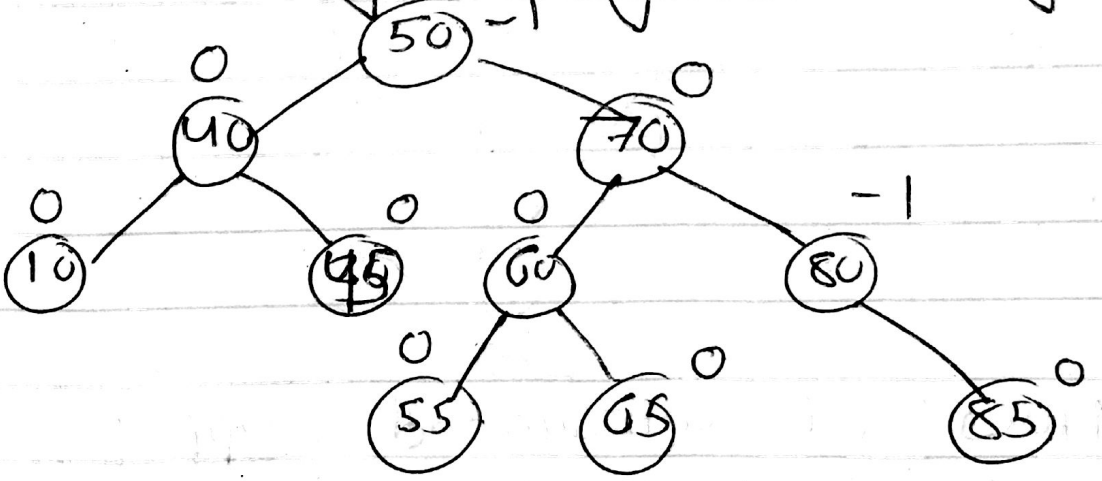
#

(B) (C)

First Applying SRR on 80 and 70 we get

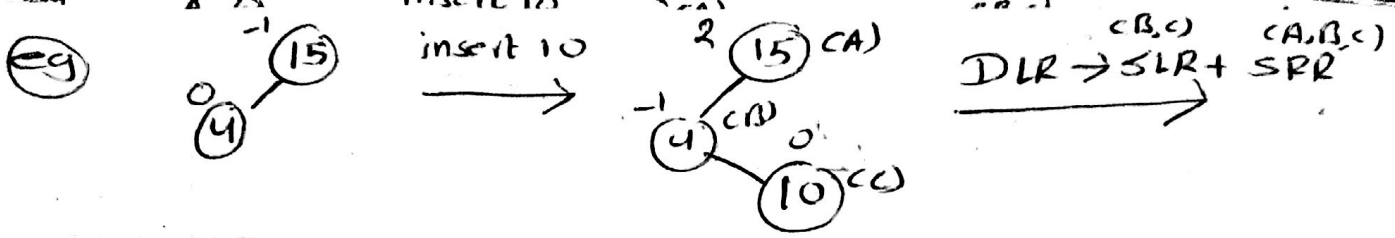


Then applying SLR on all three i.e (A) 60, (C) 70 and (B) 80 and finally adding 65 we get

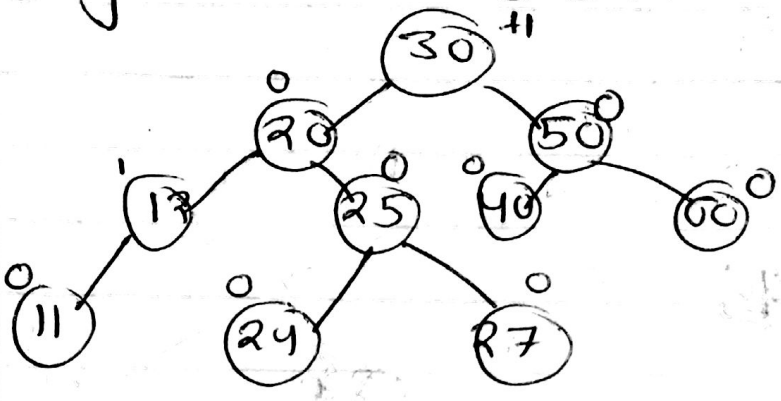


(4) Double left Rotation (DLR)

Unbalanced is occured due to inserting a node in the right subtree of the left child of pivot node. Balancing can be done by performing DLR. ie first SLR and then SRR

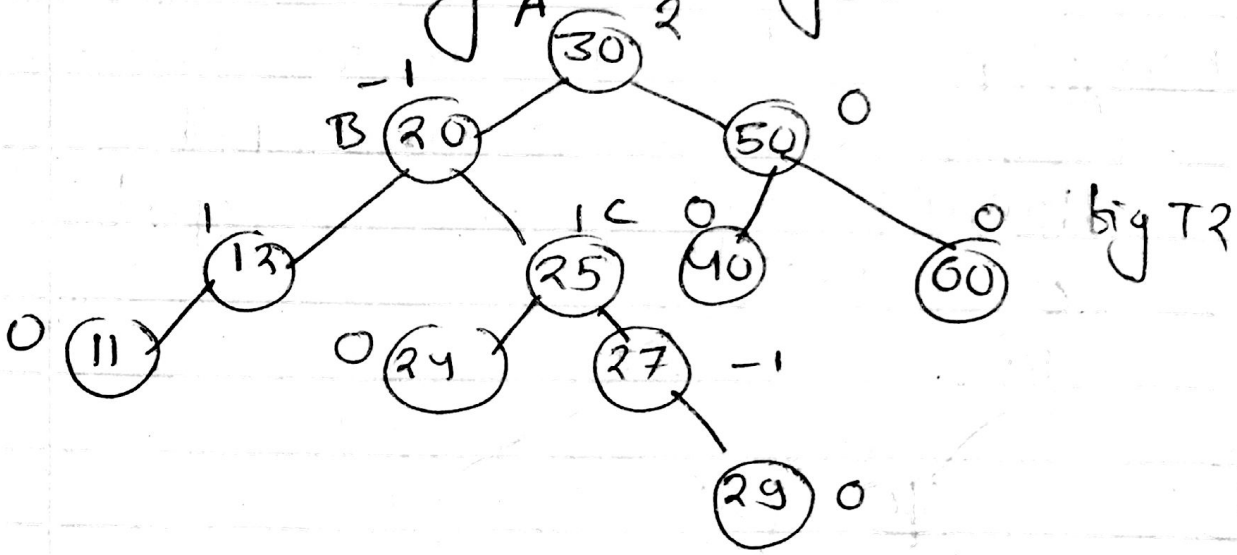


eg: Consider the AVL tree as below



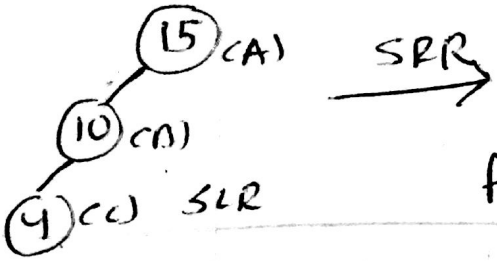
big T1

Inserting 29, we get

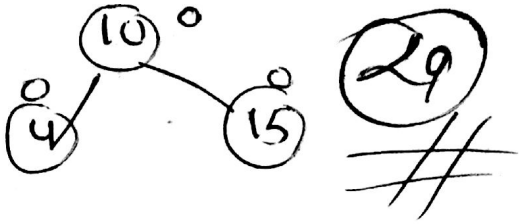


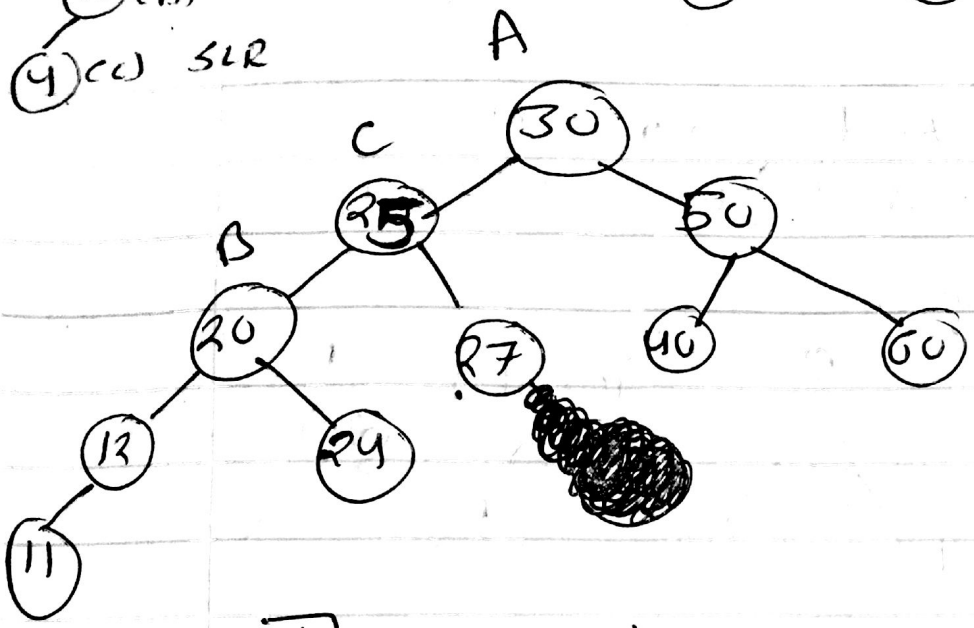
big T2

Now T2 is unbalanced, applying DLR on 30 20 and 25.
First applying SLR on 30 (B) and 25 (C) we get

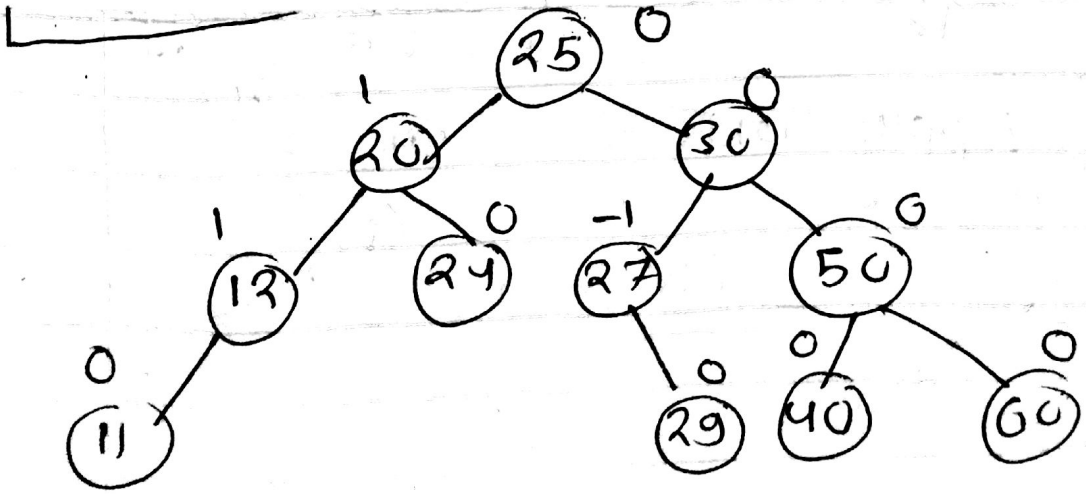


SRR →





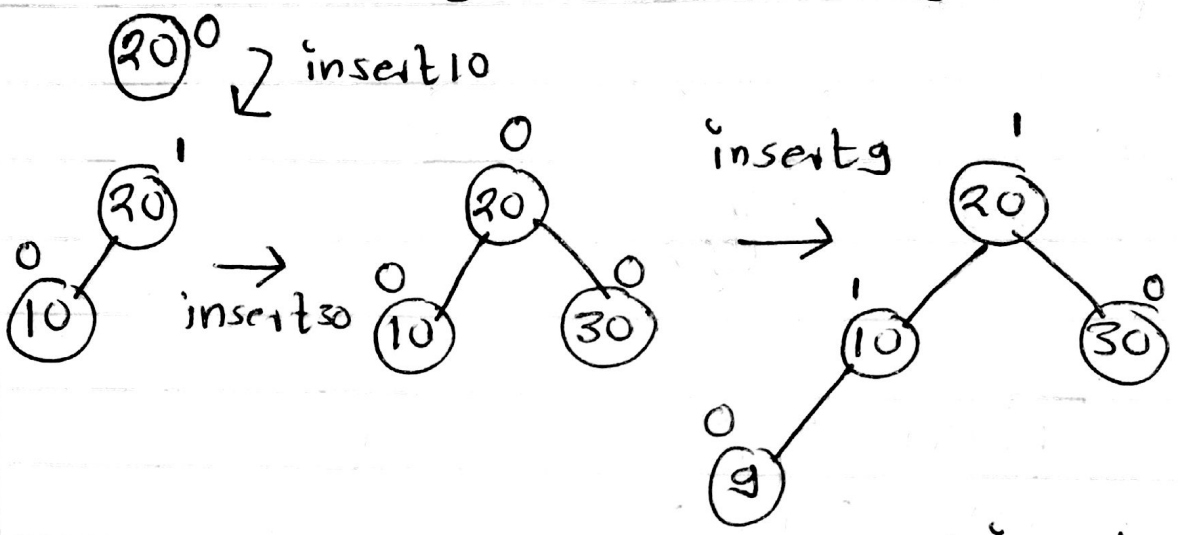
Then applying SRR on 30 25 and 20. and finally

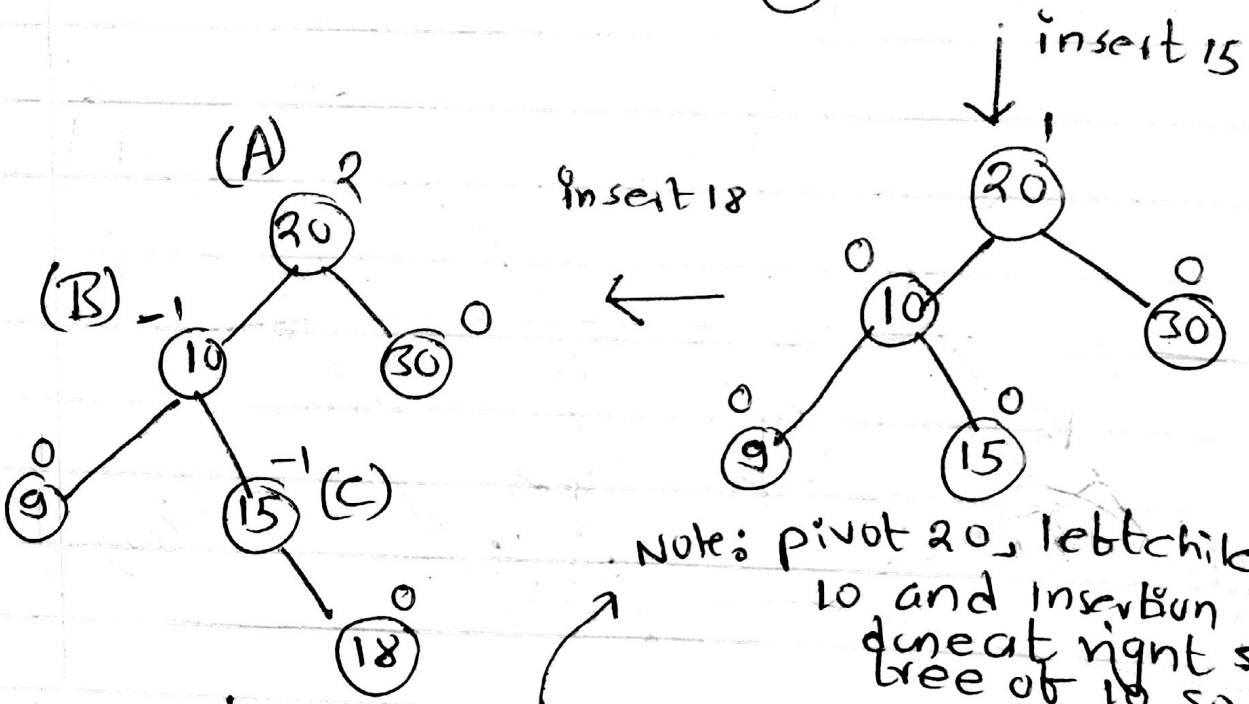


A is pivot

Construct AVL from
20, 10, 30, 9, 15, 18, 13, 13, 14, 11

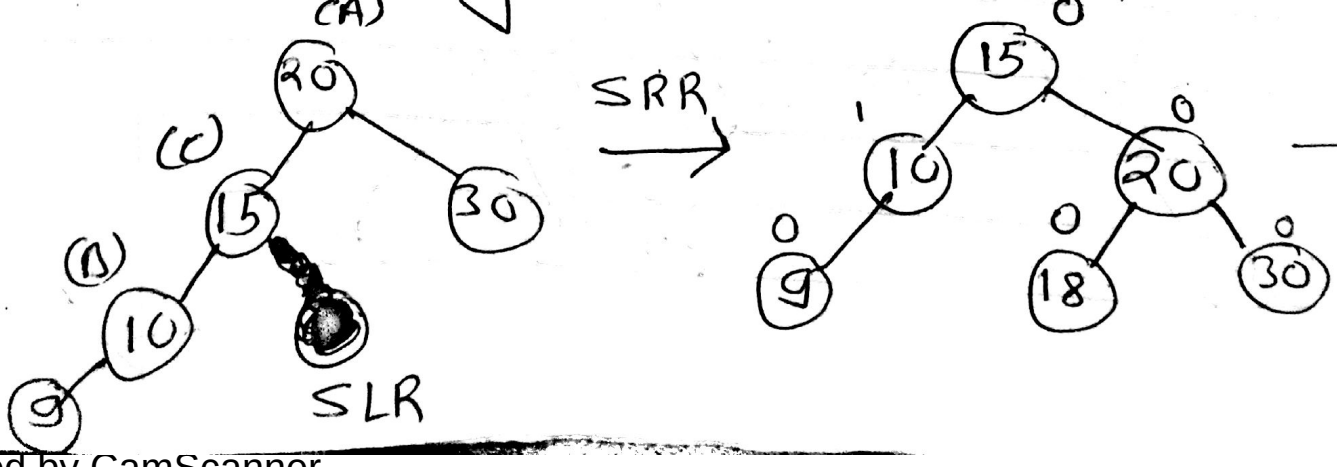
Soln:





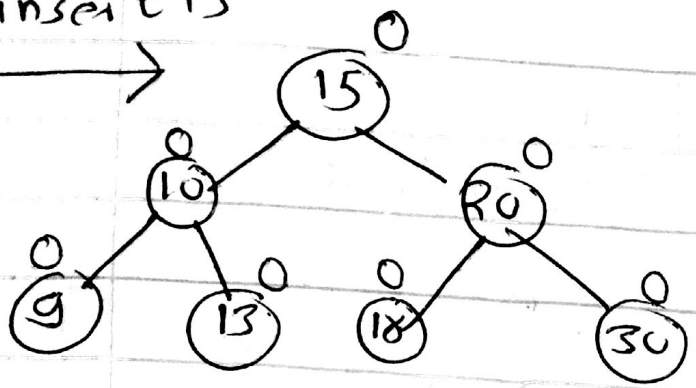
Note: pivot 20, leftchild 10 and insertion done at right sub-tree of 10 so DLR.

↓ Apply DLR → SLR + SRR

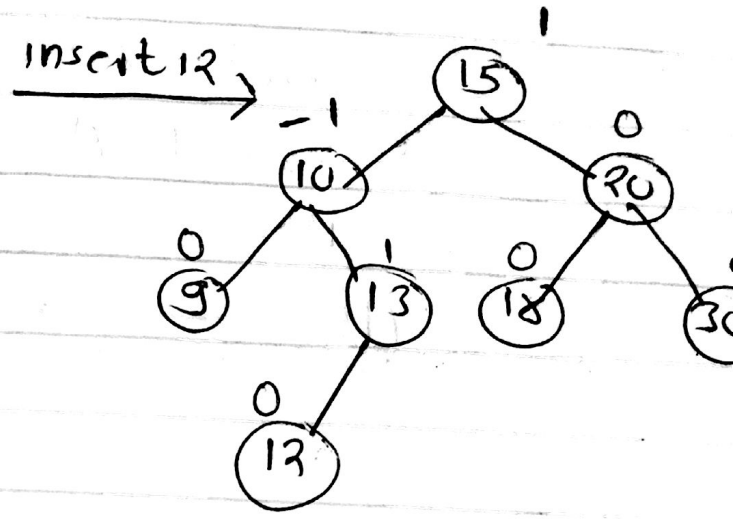


(22)

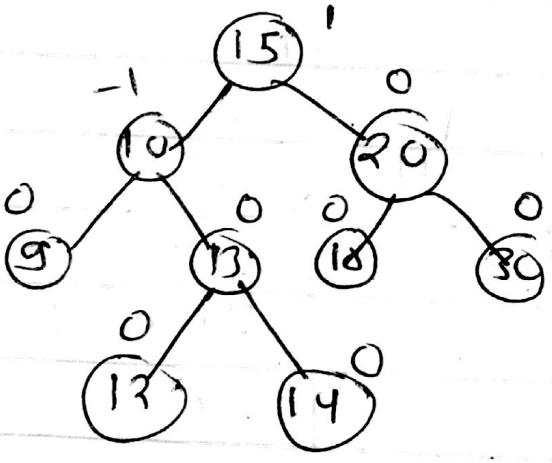
insert 13



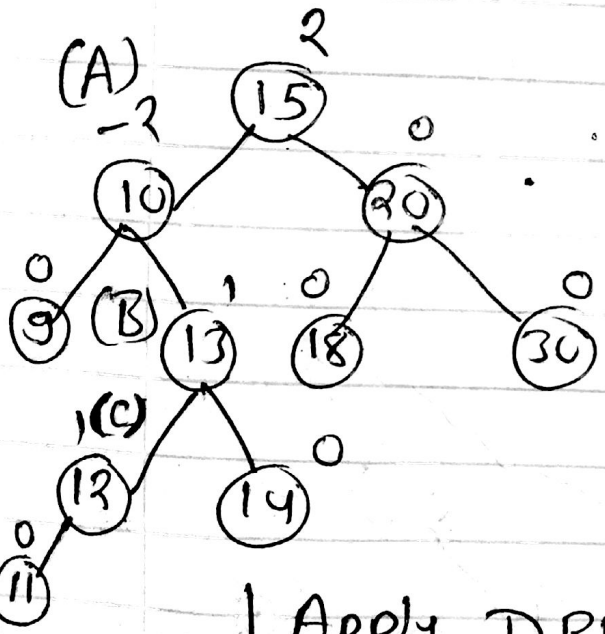
insert 12



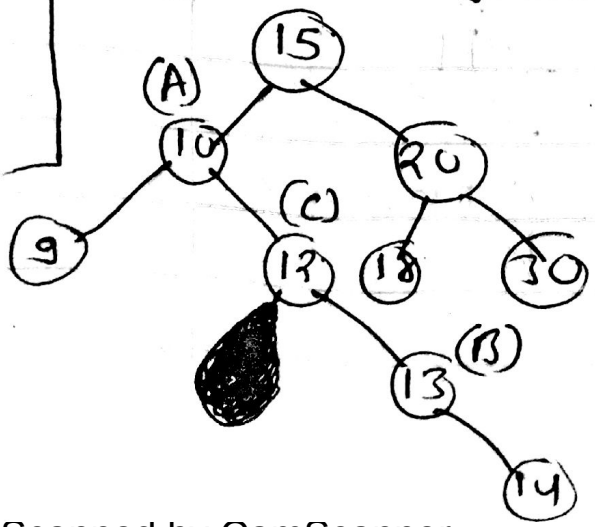
insert 14



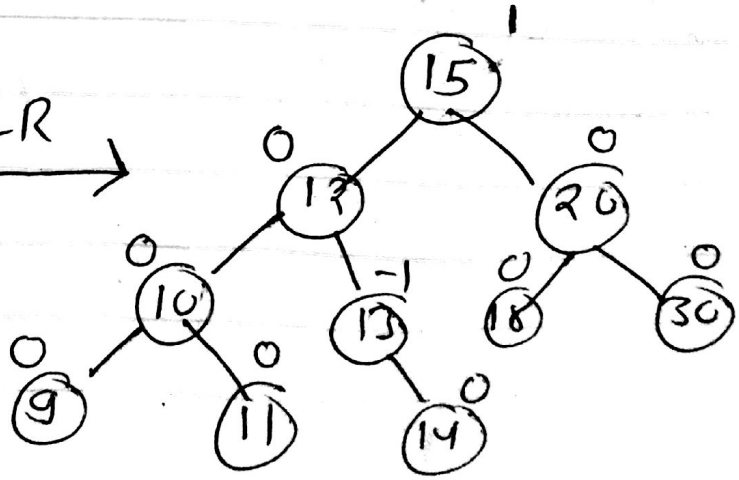
insert 11

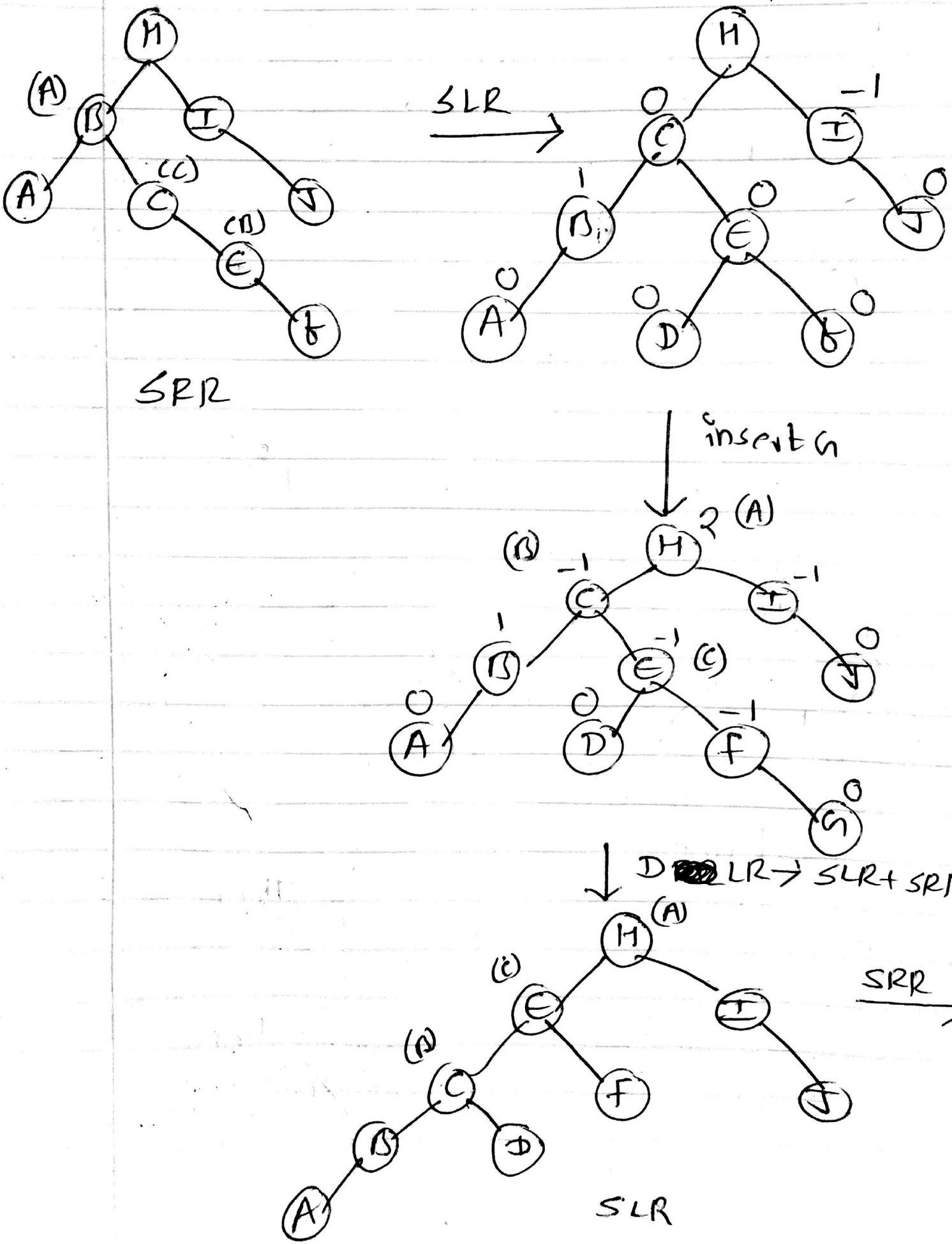


Apply DRR → SRR + SLR

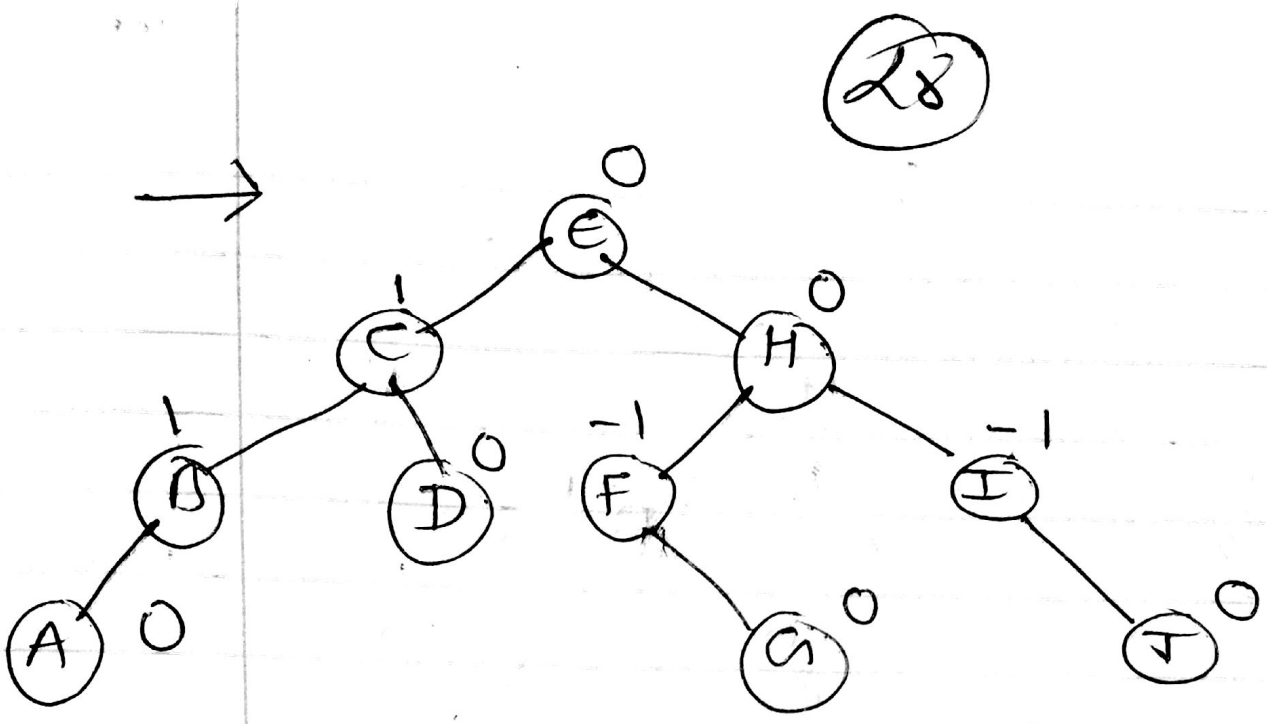


SLR

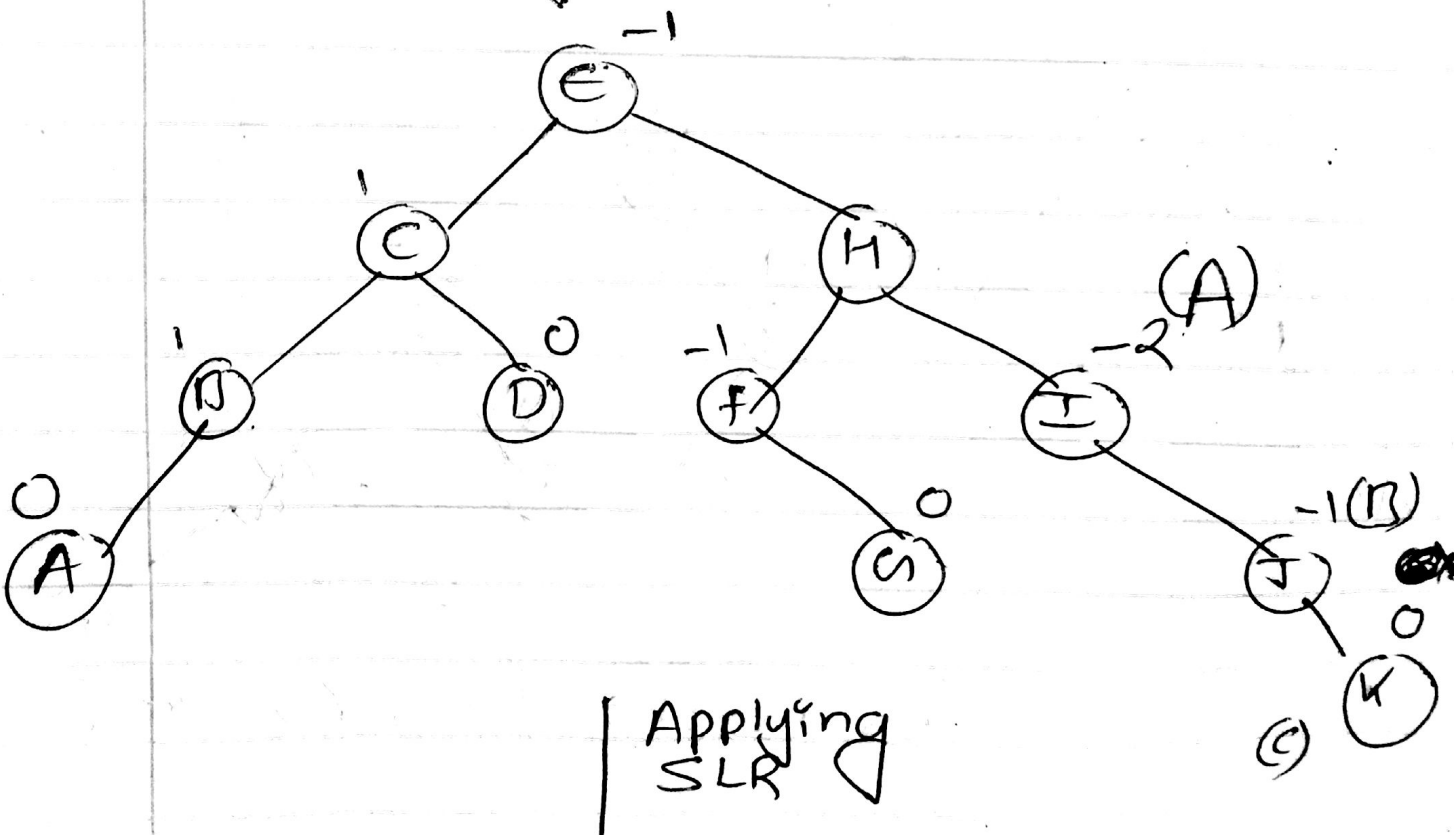



SLR
SRR
insertion
LR → SLR + SRR
SRR
SLR

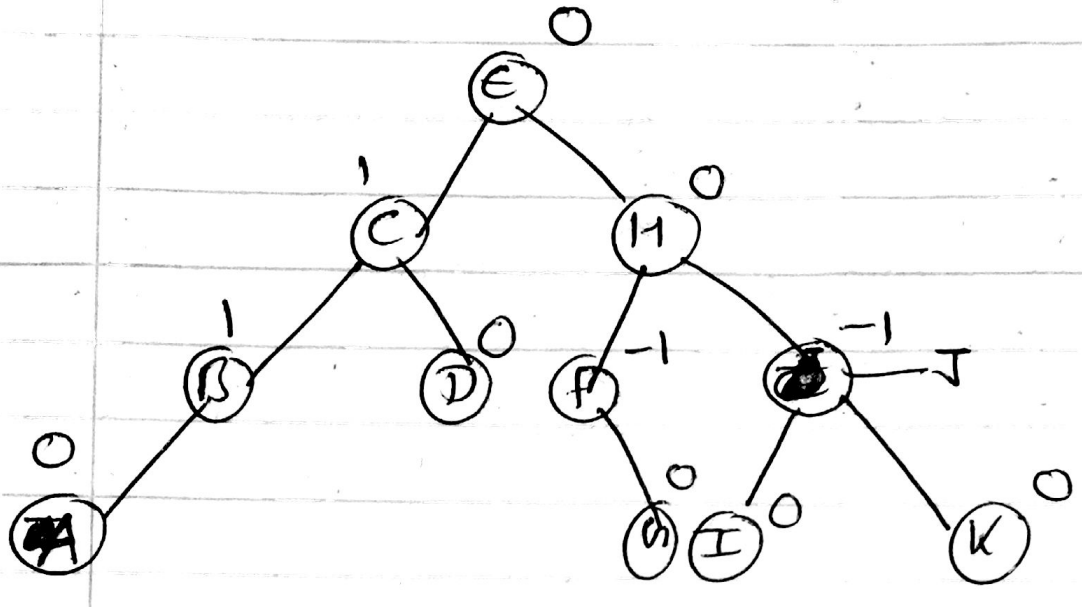
28



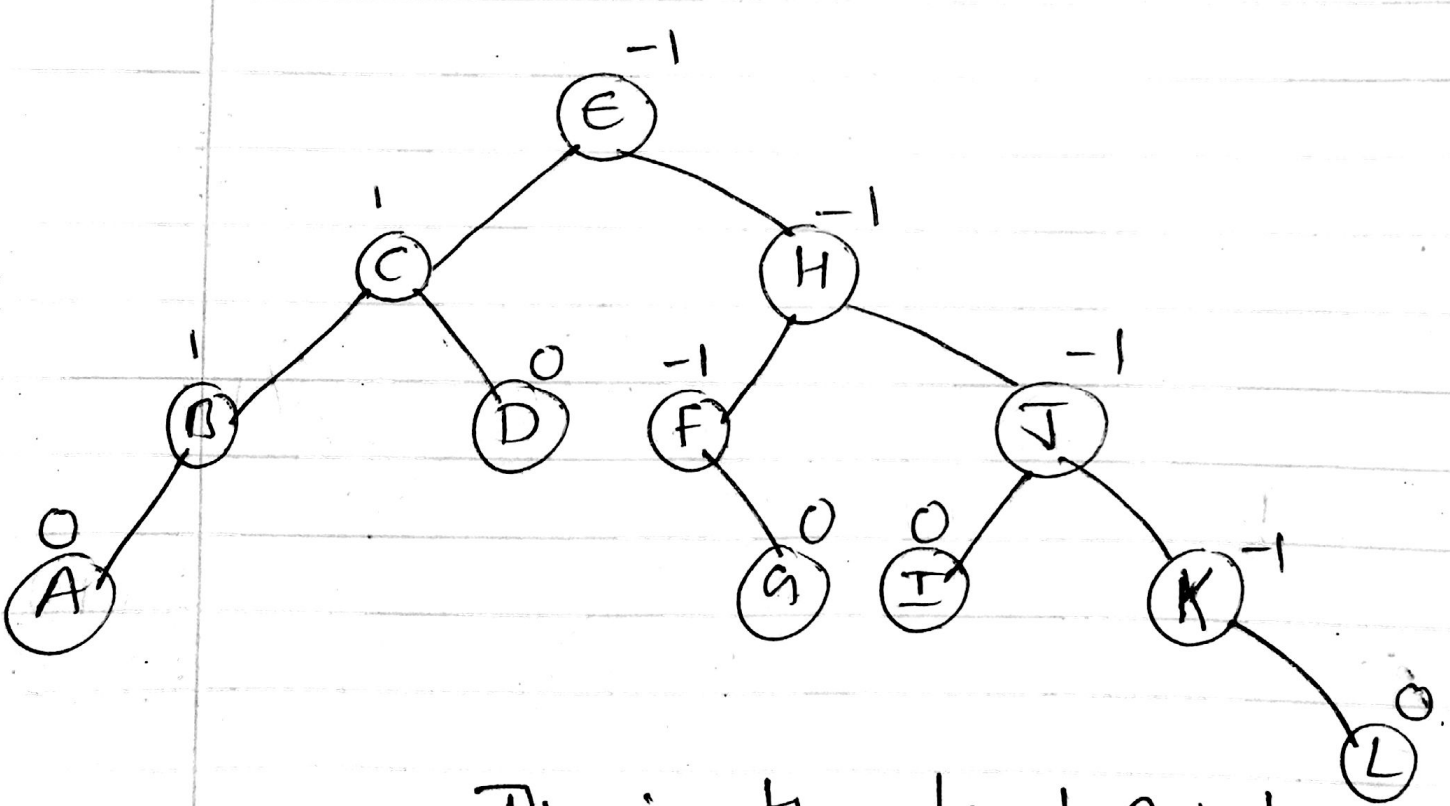
Insert K



Applying SLR



↓ insert L

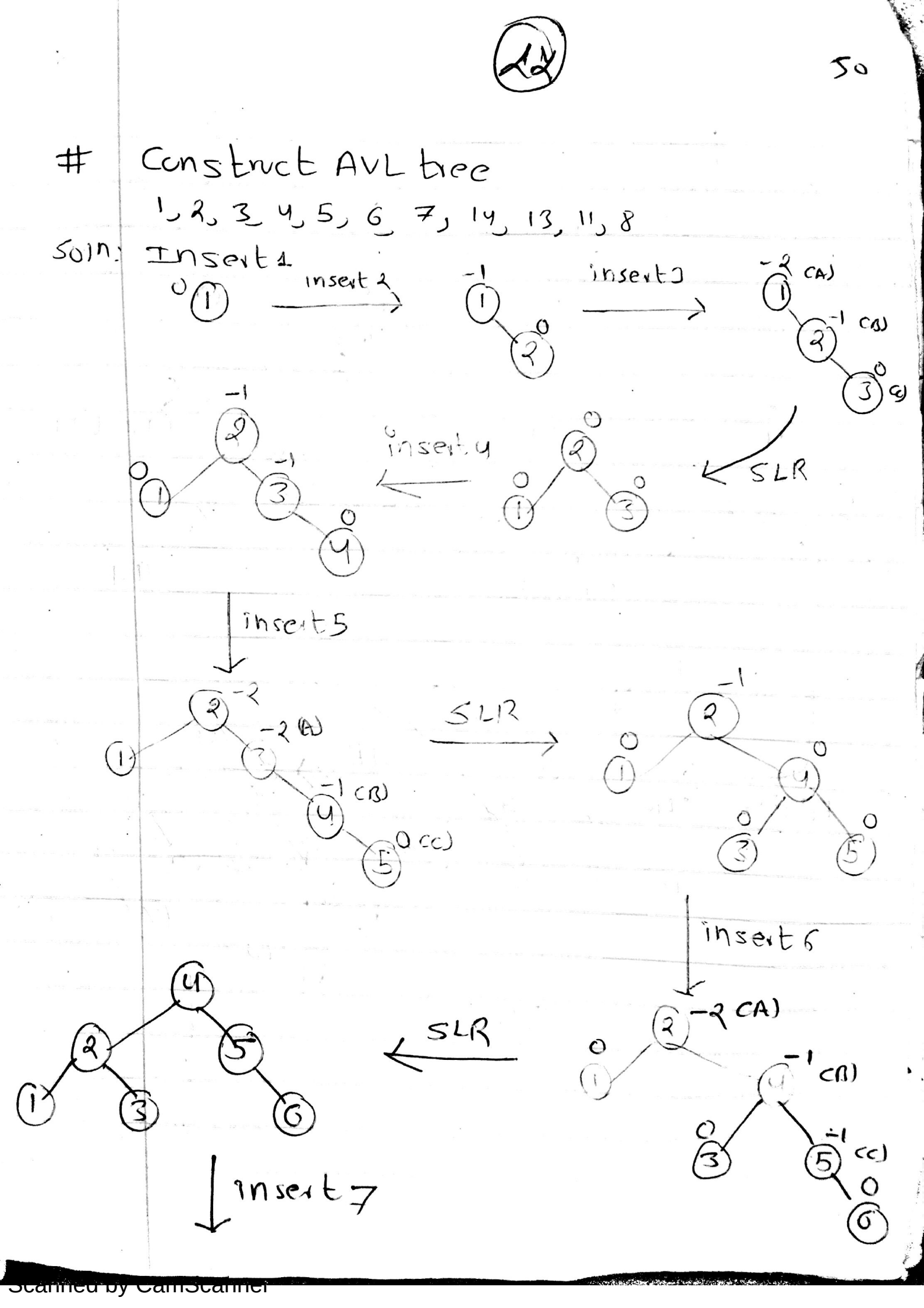


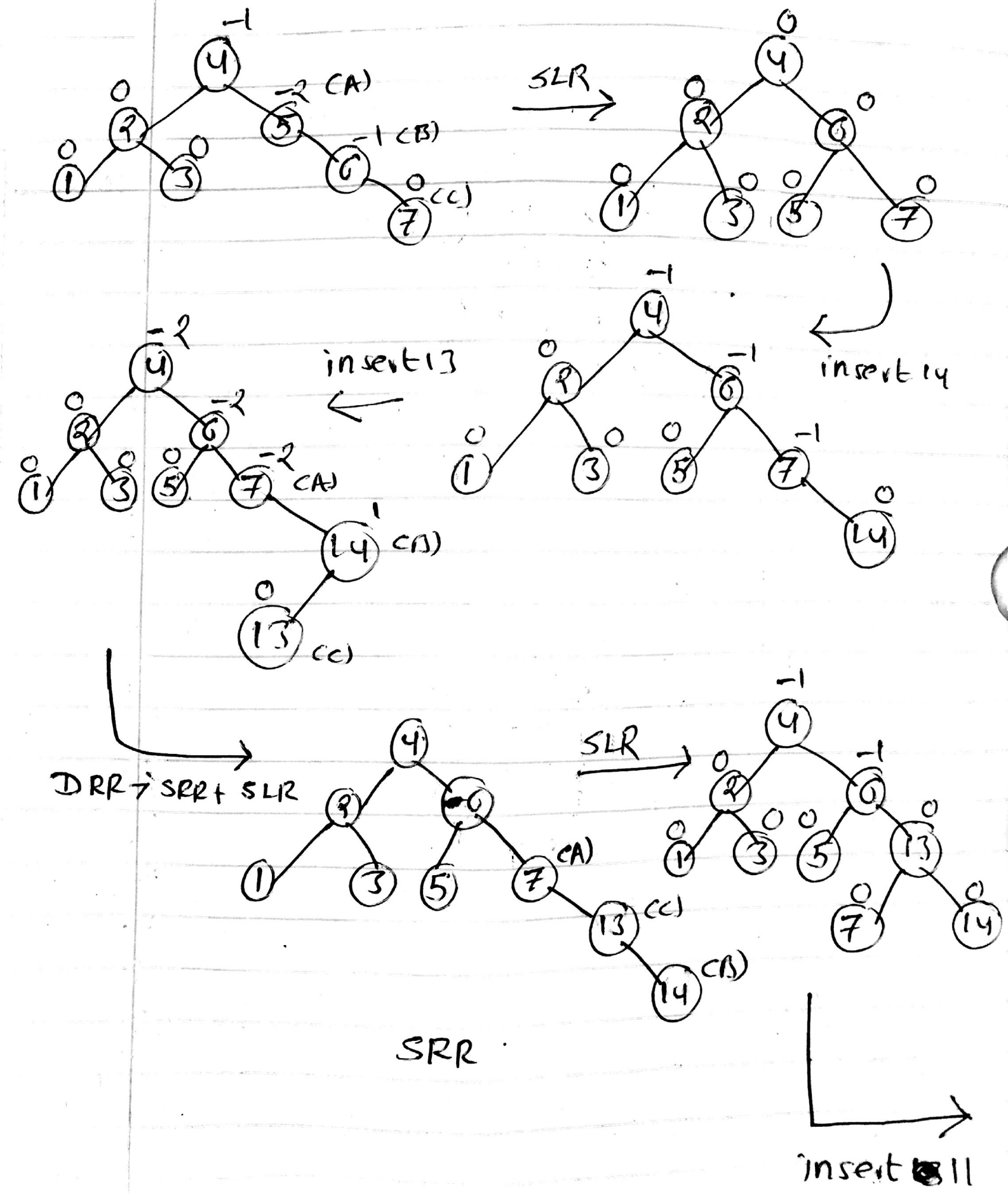
This is the final AVL tree.

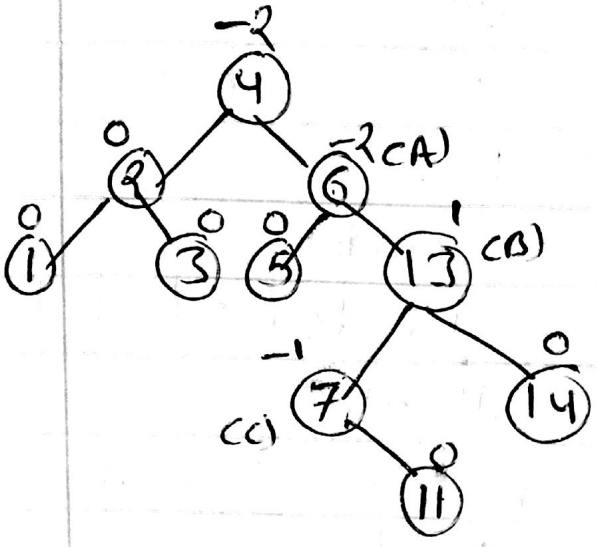
Construct AVL tree

1, 2, 3, 4, 5, 6, 7, 14, 13, 11, 8

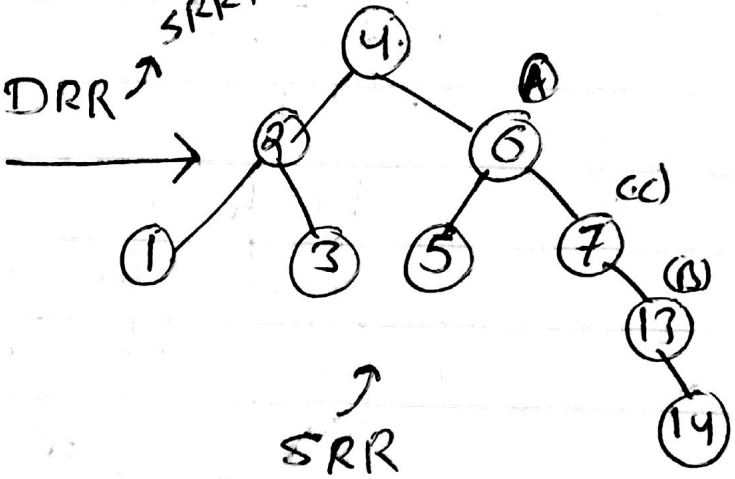
Soln: Insert 1





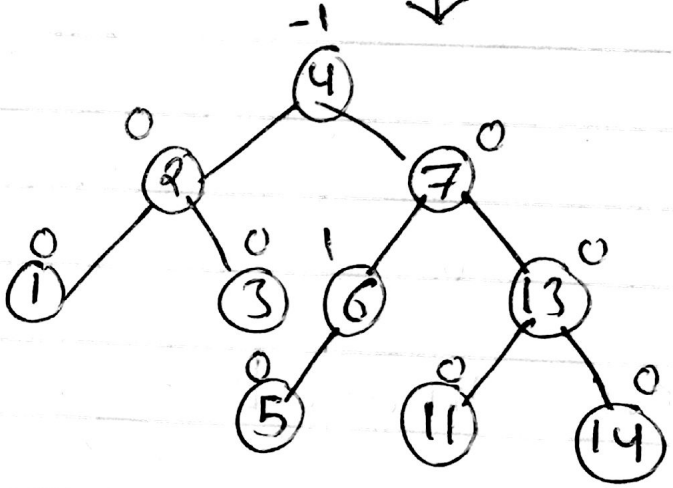


DRR → SRR+SLR

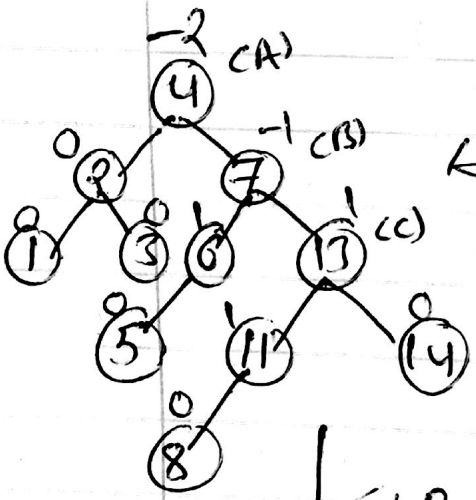


SRR

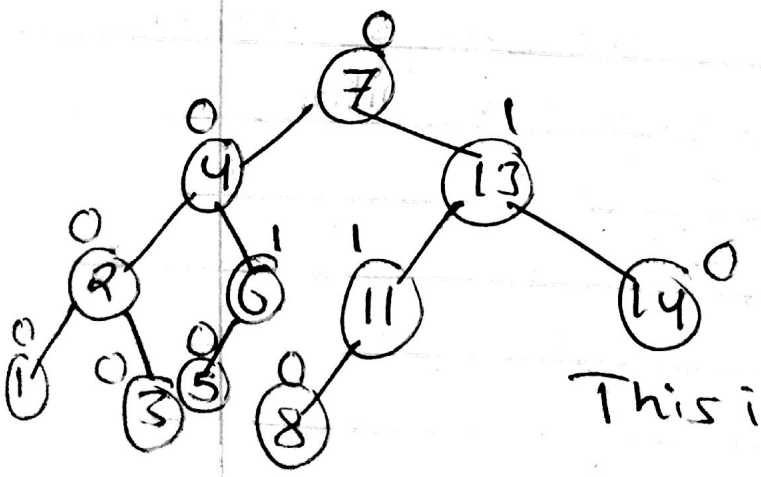
SLR



insert 8



SLR



This is final AVL tree.