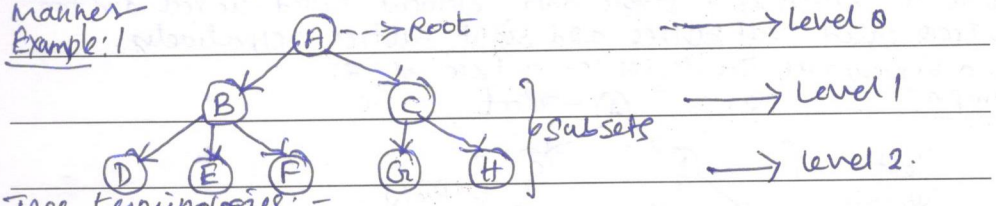
Syllabus: UNIT-4

Trees: Basic Tree terminologies, Different types of trees: Binary Tree, Binary Search Tree, AVL Tree; Tree operations on each of the tree and their algorithms with time complexities. B-Trees: Definition, operations

Tree: The hierarchical representation of data items is known as a "tree". A tree is a non-linear data structure. A tree has single element which is called as "root" and it has several subsets in hierarchical manner

Example:1



Tree terminologies:-

1) ROOT 2) Node 3) Terminal Node 4) Non-Terminal Node
5) Degree of a Node 6) Degree of a tree 7) Level 8) Edge
9) Depth of Tree 10) Siblings 11) Path 12) Forest

1) ROOT:- Root is a starting and special element of a tree. From the root the subsets gets derivation.

Ex:- In the above example-1 (A) is called as the 'root' of a tree.

2) NODE:- Each element in a tree is called a "node". A Node stores information of a tree and it links to other elements in a tree. Root itself is a node.

Ex:- In Example-1 A, B, C, D, E, F, G, H are the nodes of the tree.

3) Terminal Node:- A node which has no links to next elements is called as "Terminal Node". Terminal node is also called as "leaf nodes"

Ex:- In Example-1 D, E, F, G, H are termed as Terminal Nodes

4) Non-Terminal Node:- A node which has links to further elements is called as "Non-Terminal Node"

Ex:- In Example-1 B, C are Non-Terminal Nodes

5) Degree of a node:- The number of links of node is known as "degree" of a node

Ex:- The degree of node 'C' is 2 Node B is 3 Node D is 0.

6) Degree of a Tree:- The maximum degree of the nodes in a tree is called "degree of a Tree".

Ex:- The degree of a tree represented above is "3"

7) Level: The position of element in a tree is called as level. The level of root in a tree is "0".

8) Depth of the tree:- The maximum level of any node in a tree is known as depth of the tree.

Ex:- Depth of the tree in Example-1 is 2

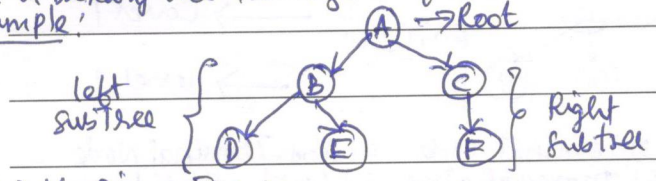
9) Edge:- The line which connects nodes in a tree is termed as "Edge"

10) Path:- It is a sequence of consequitive edges from some node to destination node in a tree

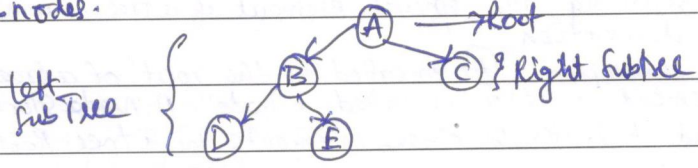
Ex:- A → B → D is a path.

11) Siblings:- The subnodes of a main node are called siblings
Ex:- In example-1 D, E, F are siblings of node-B.
12) Forest:- It is a collection of disjoint trees.

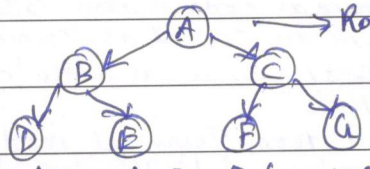
Binary Tree:- It is a finite set of data elements which is either empty or consists of a single data element called the root and two sub trees called left subtree and right subtree respectively.
In a binary tree the degree of a tree is '2'.
Example:



Strictly Binary Tree:- It is a binary tree in which all non-terminal nodes must be either have degree zero or consists 2 subtrees (or) sub nodes.



Complete Binary Tree:- It is a strictly binary tree in which all leaf nodes are at the same level
Example:-



All leafnodes like D, E, F, G are at level 2. Hence it is a complete binary tree.

Representation of a Binary Tree:- Static Representation (Using Arrays), Dynamic " " (Using Linked lists)

Static Representation:-
Syntax:- datatype array name[size];
Ex: char BTree[7];

BTree

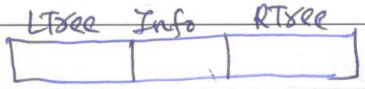
Disadvantages in Static Representation:
1) Not suitable for normal Binary tree. It is well suited for complete binary tree
2) Wastage of memory is more
3) Insertion and deletion is difficult.

Dynamic Representation of Binary Tree:-

Here a node consists of 3 fields

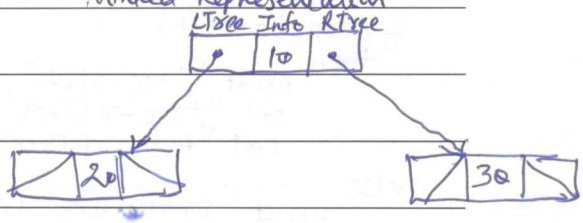
1) Info: Holds the actual data
2) LTree: It stores the address of left subTree
3) RTree: It stores the address of Right SubTree

Structure:-

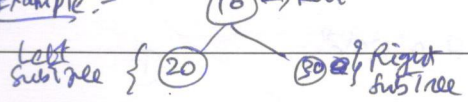



```
Struct node
{
  int info;
  struct node *LTree;
  struct node *RTree;
}; typedef struct node NODE;
```

Linked Representation:



Example:-



Primitive operations on Binary Tree:-

1) Create 2) Make Tree 3) Traversal 4) Insertion
5) Deletion 6) Searching

1) Create:- This creates an empty binary tree.
2) Make Tree:- This creates a binary tree with single node
3) Traversal:- It is the process of visiting each and every element of a tree exactly once.
4) Insertion:- It inserts an element to the binary tree
5) Deletion:- It deletes an element from the binary tree
6) Searching:- This searches for an element in the binary tree.

Traversal of Binary Tree:

It is an operation which we can visit each and every element of a tree exactly once.
There are three ways to traverse a binary tree:

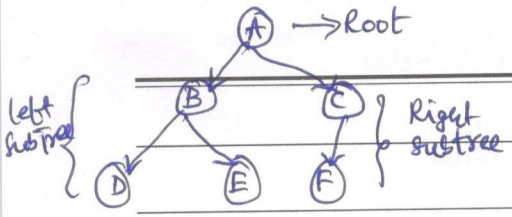
1) Preorder Traversal
2) Inorder Traversal
3) Postorder Traversal

1) Preorder Tree Traversal:-

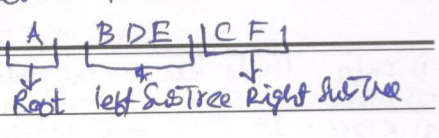
This traversal performs the following three steps:

a) Visit the root
b) Traverse the left subtree in preorder
c) Traverse the right subtree in preorder

Consider the binary tree



The preorder traversal of the tree



Algorithm for Preorder traversal is:

Step 1: start

2: [Check the root of the tree for emptyness]

if (tree != NULL)

then

print "tree->info"

call "tree->LTree" recursive procedure

call "tree->RTree" recursive procedure

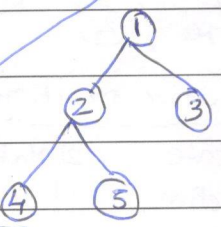
else

print "traversing is not possible"

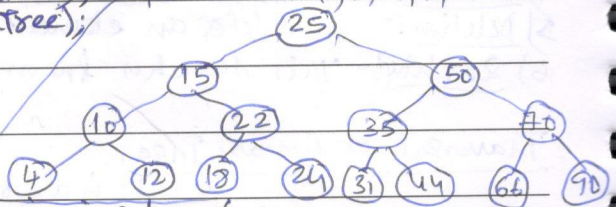
3: Stop

Pseudo code for PreOrder traversal:-

```
preorder (NODE *tree)
{
  if(tree == NULL)
  { printf("Traversing is not possible");
  }
  else
  { printf("%c", tree->info);
    preorder (tree->LTree);
    preorder (tree->RTree);
  }
}
```



Inorder: 4, 2, 5, 1, 3
Preorder: 1, 2, 4, 5, 3
Postorder: 4, 5, 2, 3, 1

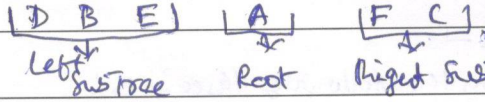


Inorder: 4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90
Preorder: 25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90
Postorder: 4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

Inorder Traversal :-

1) Traverse the left subtree in inorder
2) Visit the root
3) Traverse the Right subtree in inorder

The inoorder traversal of the tree



Algorithm:-

Step 1: start

2: [Check the emptyness of tree]

if (tree != NULL)

then

call "tree→LTree" recursively
print "tree→info"
call "tree→RTree" recursively

else
print "Tree is empty"

3: stop

Post order Traversal:

1. Traverse the left subtree in postorder
2. Traverse the right subtree in post order
3. Visit the root.

The postorder traversal of the tree is: D E B | F C | A
(D E B: left subtree; F C: Right subtree; A: Root)

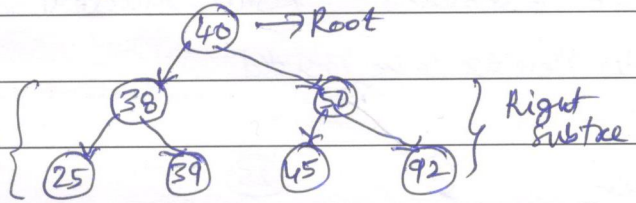
Algorithm for Postorder traversal:-

Step 1: start
2: [Check the emptyness of binary tree]
If (tree != NULL)
then
call "tree→LTree" recursively
call "tree→RTree" recursively
print "tree→info"
else
print "tree is empty"
3: stop

Binary Search Tree: (BST)

It is a tree in which the elements which are less than the root comes at the left side of the root and the elements which are greater than the root one comes at the right side of the root

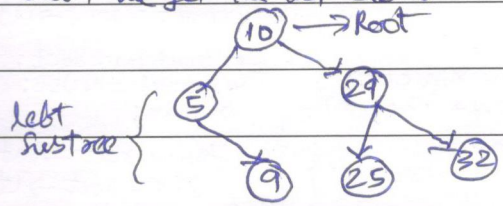
Example:-



Construct a binary search tree for the following list of elements
10, 29, 5, 32, 9, 25

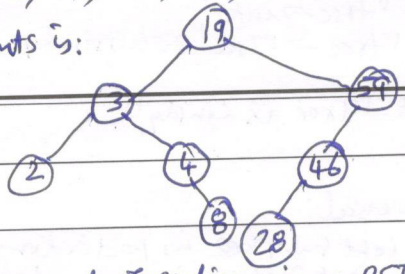
Procedure:- To construct BST, first we have to take the starting element and make it as a root element. Then take the second element compare with the root: if the element is less than root, then keep the element to the left side, other case keep the element to the right side. This process continues until we get the last element.

Example:



Example: 2: 19, 59, 3, 4, 46, 28, 2, 8

BST for the list of elements is:



Algorithm for searching and inserting in a BST:-

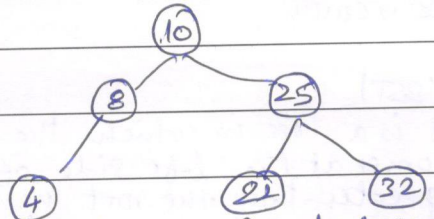
Step 1: Start

2: [Compare item with the root of the tree]
if item < Root then proceed to the LTree of the root
if item > Root, proceed to the RightTree of the root
if item == Root, Proceed to the either right/left side of the root.

3. [Repeat the step-2 until one of the following condition occur]
We meet a node such that the node is equal to 'item'. So search is successful
We meet an empty subtree there we can insert 'item'

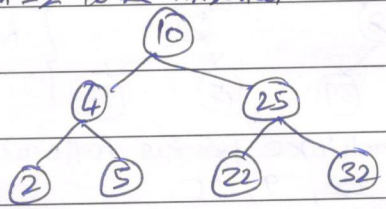
4: stop.

Example:-



item = 8 to Search Result: successful

Consider item = 2 to be inserted



Tree after inserting item = 2

//Program for implementation BST operations (Insert, delete, traversal, display)

```
#include<stdio.h>
#include<conio.h>
struct node
{ int info;
  struct node *ltree;
  struct node *rtree;
}*root;
void find(int item, struct node **parent,
          struct node **location)
{ struct node *ptr, *ptrsave;
  if(root == NULL)
  {
    *location = NULL;
    *parent = NULL;
    return;
  }
  if(item == root->info) /*item is not at root
  {
    *location = root;
    *parent = NULL;
    return;
  }
  if(item < root->info)
    ptr = root->ltree;
  else
```

```
    ptr = root->rtree;
  ptrsave = root;
while(ptr != NULL)
{
 if(item == ptr->info)
 {
   *location = ptr;
   *parent = ptrsave;
    return;
 }
 ptrsave = ptr;
 if(item < ptr->info)
     ptr = ptr->ltree;
 else
     ptr = ptr->rtree;
}
*location = NULL   /* item is not found
*parent = ptrsave;
}
void insert(int item)
{
 struct node *tmp, *parent, *location;
 find(item, &parent, &location);
 if(location != NULL)
 { printf("\n Item is already present");
   return;
 }
 tmp = (struct node*) malloc(sizeof(struct
                                node));
 tmp->info = item;
 tmp->ltree = NULL;
 tmp->rtree = NULL;
 if(parent == NULL)
     root = tmp;
 else if(item < parent->info)
     parent->ltree = tmp;
 else
     parent->rtree = tmp;
}
void case1(struct node *parent,
           struct node *location)
{ if(parent == NULL) /* item to be deleted
    root = NULL;          is a root node
 else
   if(location == parent->ltree)
     parent->ltree = NULL;
   else
     parent->rtree = NULL;
}
Void case2(struct node *parent,
           struct node *location)
{ struct node *child;
 if(location->ltree != NULL) /* item to
  child = location->ltree;   be deleted has ltree
 else child = location->rtree;   <- rtree
 if(parent == NULL)  /* item to be
    root = child        deleted is a root
 else
   if(location == parent->ltree)
     parent->ltree = child;
   else parent->rtree = child;
}
void case3(struct node *parent,
           struct node *location)
{ struct node *ptr, *ptrsave, *suc,
                 *parsuc;
 /* find inorder successor of its
                          parent */
 ptrsave = location;
 ptr = location->rtree;
 while(ptr->ltree != NULL)
 { ptrsave = ptr;
   ptr = ptr->ltree;
 }
 suc = ptr;
 parsuc = ptrsave;
 if(suc->ltree == NULL && suc->rtree
                          == NULL)
     case1(parsuc, suc);
 else
     case2(parsuc, suc);
 if(parent == NULL) /* item to be deleted
     root = suc;        is root node
 else
   if(location == parent->ltree)
     parent->ltree = suc;
   else
     parent->rtree = suc;
 suc->ltree = location->ltree;
 suc->rtree = location->rtree;
}
void del(int item)
{ struct node *parent, *location;
 if(root == NULL)
 { printf("\n Tree is empty");
   return;
 }
 find(item, &parent, &location);
 if(location == NULL)
 { printf("\n Item is not present");
   return;
 }
 if(location->ltree == NULL &&
    location->rtree == NULL)
     case1(parent, location);
 if(location->ltree != NULL &&
    location->rtree == NULL)
     case2(parent, location);
 if(location->ltree == NULL &&
    location->rtree != NULL)
     case2(parent, location);
 if(location->ltree != NULL &&
    location->rtree != NULL)
     case3(parent, location);
```

Deletion of a node from the Binary Search Tree:-

Step 1: Start

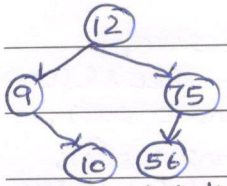
2: If the node-N (which we want to delete) has no children then node-N is deleted from tree by replacing the location of node-N in its parent node P(N) by NULL.

3 If the node-N has one child then node-N is deleted from the tree by simply replacing the location of node-N in its parent node P(N) by location of only child on node-N

4 If the node-N has two childerens then node-N is deleted from the tree by first inorder successor of N from tree by using step2 and step3 and then replace node-N in tree by the node S(N) (Successor node of N)

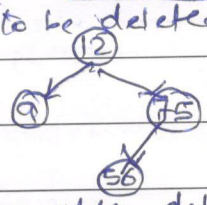
5. Stop.

Example:- Case 1: Deleting a node which has no children



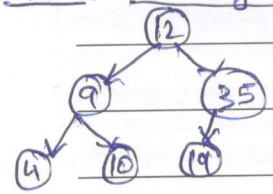
Tree before deleting

Item to be deleted is (10)



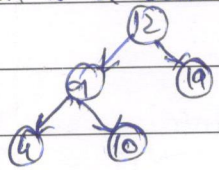
Tree after deletion of item (10)

Case 2: Deleting a node which has only one child



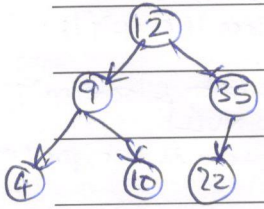
Tree before deletion

Item to be deleted is (35)



Tree after deletion of node (35)

Case 3: Deleting a node which has two childens



Tree before deletion

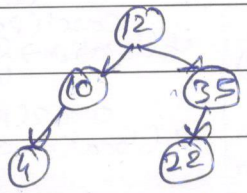
Item to be deleted is (9)

first inorder successor of (9) is

inorder: 4 9 10 12 22 35

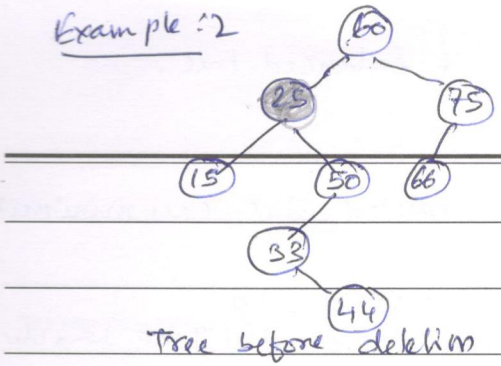
The first inorder successor of (9) is (10)

So replace (9) with (10)



Tree after deletion of node (9)

Example :2

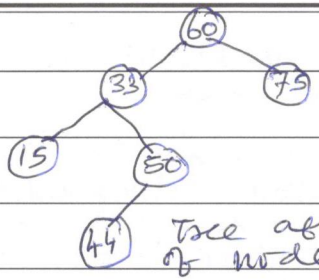


Tree before deletion

Node to be deleted is (25)

Inorder: 15 25 33 44 50 60 66 75

first Inorder successor of (25) is (33)

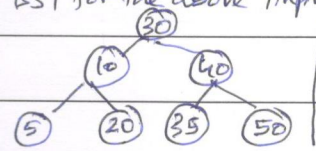


Tree after deletion of node (25)

Drawbacks of BST Trees:—

Consider Input 1: 30, 40, 10, 50, 20, 5, 35

The BST for the above input 1 is

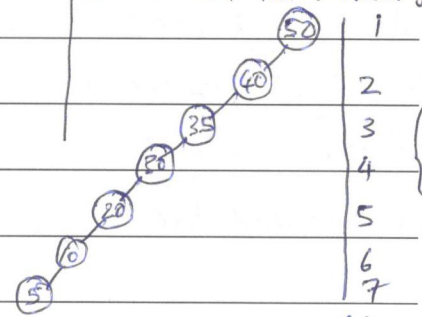


Tree height is 3

we treat this kind of trees as Balanced Tree

Input-2 : 50, 40, 30, 20, 10, 5

where input elements were not altered but their order gets changed



Tree Height : 7 which is equal to no of elements

So in general the height of BST in / Search takes

minimum of "log n"

maximum of "n"

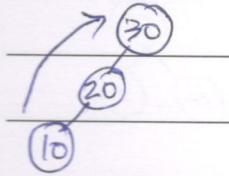
The height of the tree is not in our hands / control.

It depends on the order of elements and how elements are inserted

We can employ a method that makes imbalanced tree as a balanced tree.

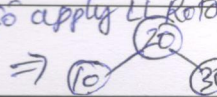
Consider the input : 30, 20, 10

order1 of input 30, 20, 10

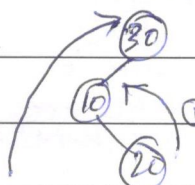


LL imbalanced tree

So apply LL Rotation

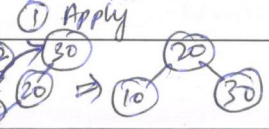


order2 of input 30, 10, 20

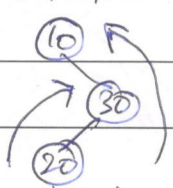


LR imbalanced

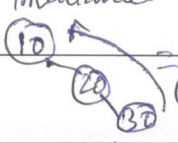
① Apply



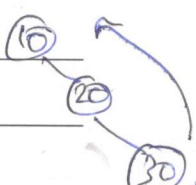
order3 10, 30, 20



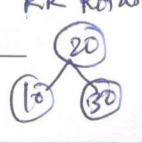
RL imbalanced



order4 10, 20, 30

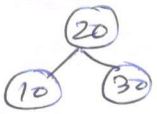


RR Rotation



other possibilities

20, 10, 30

20, 30, 10 ⇒ 

} balanced tree.

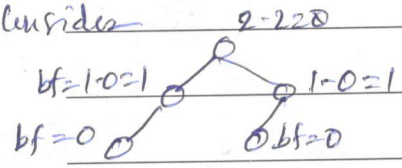
A minimum height in BST is expected that can minimize the time of search.

We use rotation operation which is developed by Adelson-Velsky and Landis called AVL Tree which is called as self balanced BST

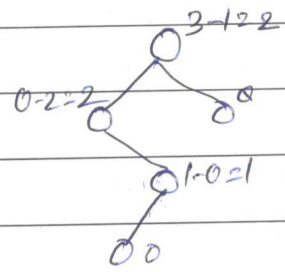
They introduce balance factor to identify whether a tree is balanced or not

Balance factor = Height of LSubTree - Height of RSubTree

$$bf = hl - hr = \{-1, 0, 1\}$$

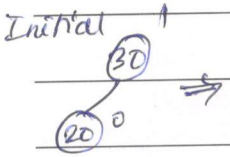
Consider 

is a Balanced tree.

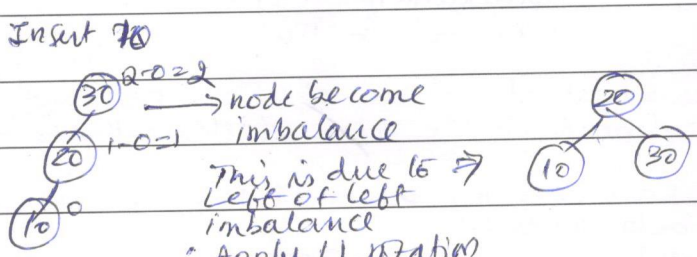


is a imbalanced tree

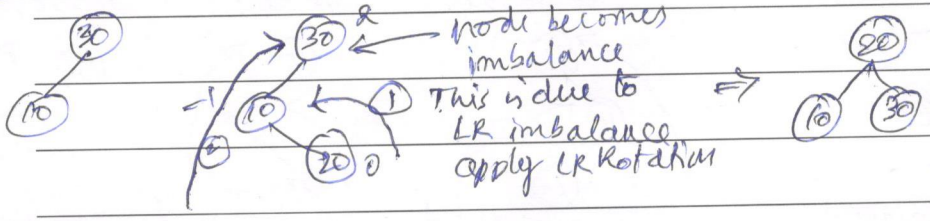
Example:-1

Initial 

Bst Tree

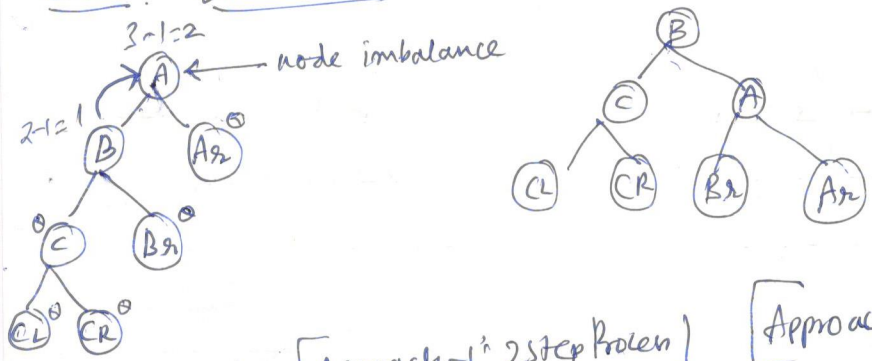
Insert 10 

Example 2

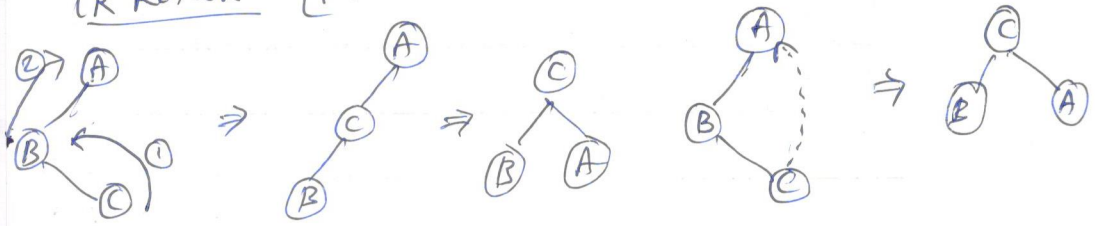
Insert 20 

Similarly apply RR, RL Rotations

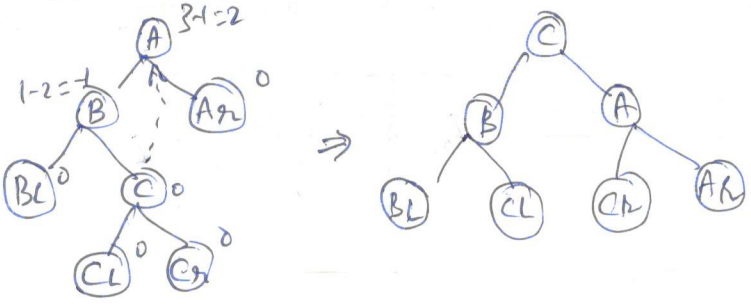
Case 1: LL Rotation:-



LR Rotation [Approach-1: 2 Step Process] [Approach-2 Single Step]

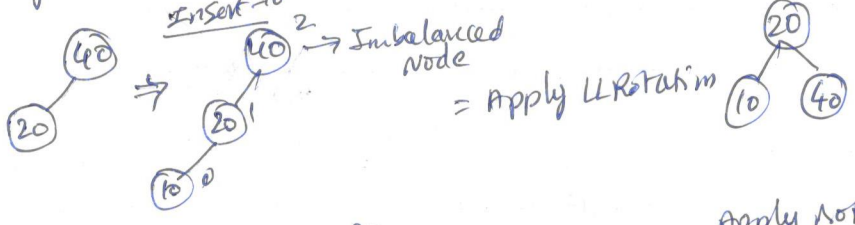


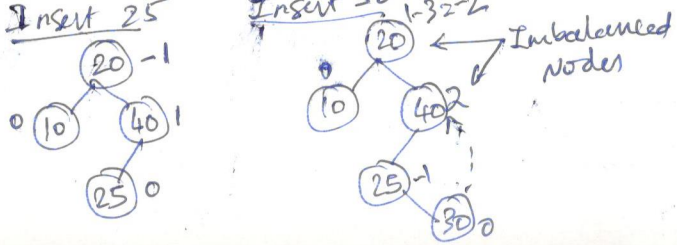
LR Rotation with Single Step:-



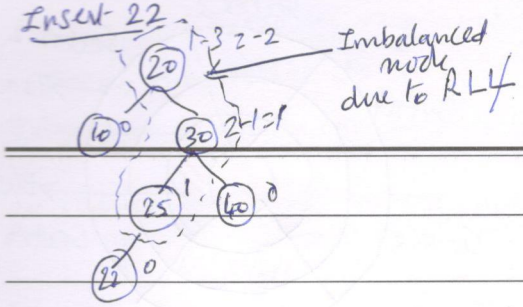
Example Illustration of Construction of AVL Tree:

Keys: 40, 20, 10, 25, 30, 22, 50

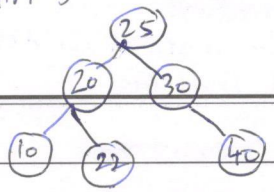




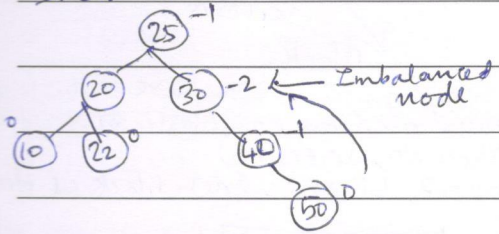
Insert 22



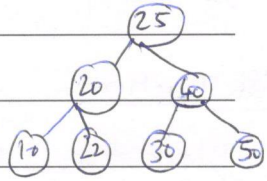
Apply RL Rotation



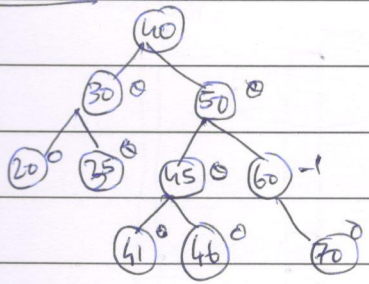
Insert 50



Apply RR Rotation



Consider 2-3=1



balanced Tree

Insert 42