6. TREES

TREE

Link list usually provide greater flexibility than array, but they are linear structure and it is difficult to use them to organise a hierarchical representation of object. Although stacks and queues reflect some hierarchy, they are limited to only one dimension. To overcome this limit we create a new data structure called tree.

Tree is a data structure which allow us to associate a parent-child relationship between various pieces of data and thus allow us to arrange our records, data and file in hierarchy fashion. A tree is a non-linear data structure. Formally it is defined as a tree in a finite set of one or more nodes such that :-

(i) There is a special data item (node called the root of the tree)

(ii) And its remaining data item are partitioned into number of disjoint set of node each of which is a tree and they are called sub-tree.

Unlike natural trees, a tree data structure is depicted upside-down with the root at the top and the leaves at the bottom.

2

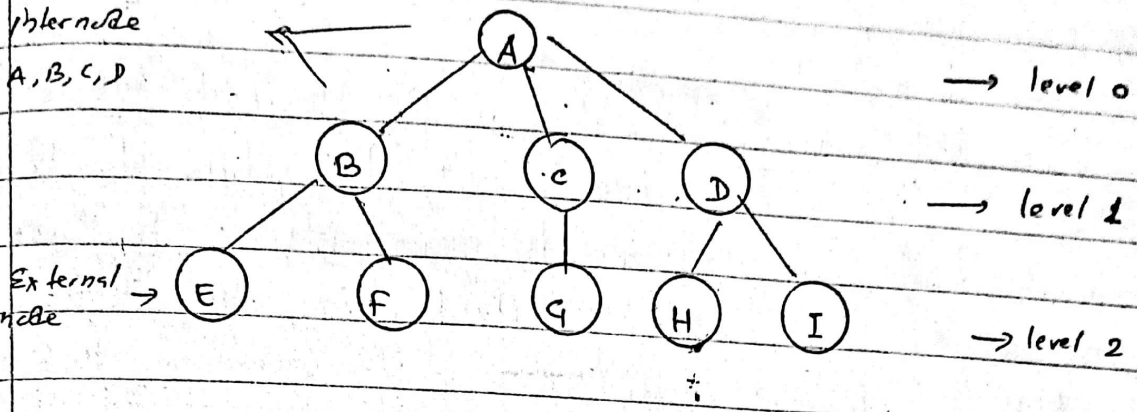


Fig tree

In this figure, A is root and remaining nod are partitioned into 3 disjoint set.

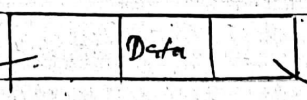
{ B, E, F } , { C, G } , { D, H, I }

BASIC TERMINOLOGIES

1. Node

We will use the term node rather than vertex with a binary tree. This is the main component of any tre structure. It stores the actual data along with link to other nodes.

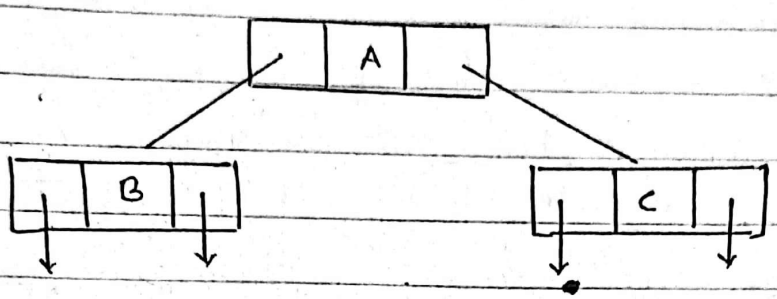
figure below shows the structure of nodes.



3

2. Parent

The parent of a node is the immediate predecessor of that node.



In the figure, A is the parent of nodes B and C

3. child.

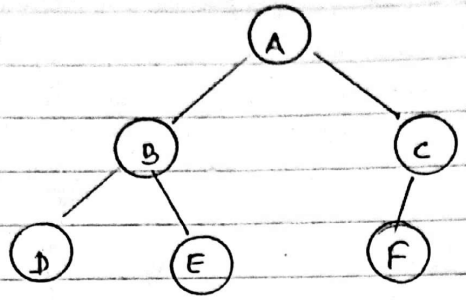
The immediate successor of the node are called child nodes. A child which is placed at the left side is called left child and the child which is placed at the right side is called right child.

In the above figure B and C are the child nodes of parent node A.

4. Root

It is a specially designated node in a tree. It is a node which has no parent. There can be only one root in a tree.

4



In the figure, the node A is the root of the tree.

5. Leaf node (External node)

A leaf node is which does not have any child node. In the above figure, nodes D, E and F are leaf nodes. It is also called terminal node.

6. Non-terminal node / internal node / non-leaf node

Any node except root node whose degree is not zero is called non-terminal node.

In the figure above, A, B and C are non-leaf nodes.

7. Siblings

The child nodes of a given parent node are called siblings. In the above figure D and E are siblings of A.

8. Branch or edge

It is a connecting line between two nodes. A node can have more than one edge. In the above figure, node A has two branches. And actually, a branch

a pointer to a node in the tree.

9. Path

Each node has to be reachable from the root through a unique sequence of edge called path. The no of edge in a path is called the length of a path.

10. Rooted tree

A rooted tree is one where we designate one node as the root. In computer science we normally reserve the term tree to refer to rooted tree. The more general structure is a free tree. In a rooted tree, any node is the root node of a subtree consisting of it and the node below it.

In the figure above, B and C are rooted tree.

11. Degree of node

It is the no of sub-tree of a node. And the degree of A is 2.

12. Level

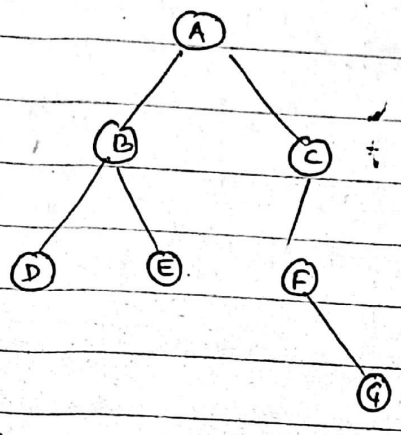
The entire tree structure is levelled in such a way that the root is always leveled as zero and increased it by 1 as we move from the root towards the subtree.

13. Depth

The depth of any node is the length of the path from the root to that node. In the fig. depth of node D is 2.

14. Height

The height of the tree is the longest path length from the node to a leaf node.



In the figure the height of the tree is 3.

BINARY TREE

A binary tree is a tree which is either empty or consists of a node called root node and two disjoint subtrees known as left sub-tree and right sub-tree which are again binary trees. So in binary tree, no node can have more than two children. A binary tree with an internal node has maximum of n+1 external node where root node considered as internal node.

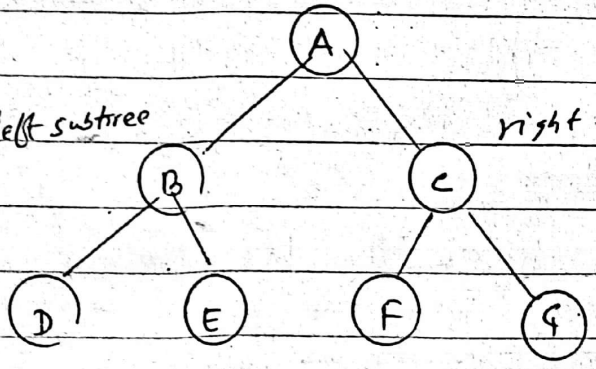


Fig. Binary tree

TRAVERSING IN BINARY TREE

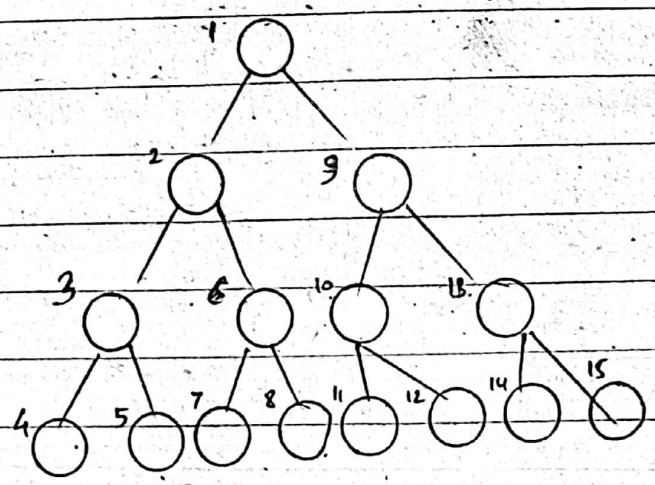
Traversal of binary tree means visiting every nodes of a binary tree at once. In many applications, it is necessary to move through all the nodes of a binary tree visiting each node only once. If there are n-nodes in a binary tree, then there are n different order in which they can be visited. There are different ways to traverse a binary tree and they differ in the order in which they visit the route (node). They are :-

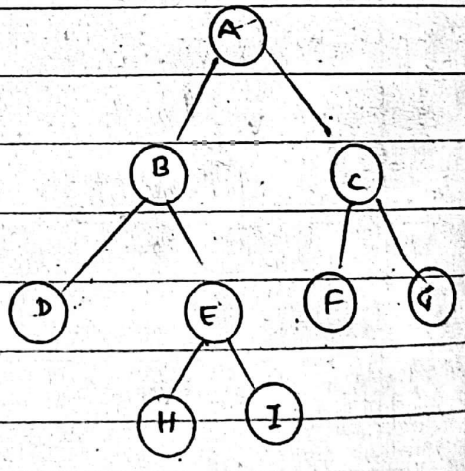
(i) Pre-order (Node - left - Right) NLR
(ii) In-order (Left - Node - Right) LNR
(iii) Post order (Left - Right - Node) LRN

(i) Pre-order (NLR)

To traverse a non-empty binary tree in pre-order, we perform the following three operations :-

(i) Visit the root or node (n).
(ii) Traverse the left sub-tree in pre-order
(iii) Traverse the right sub-tree in pre-order





→ ABDEHICFG

C-code for above algorithm

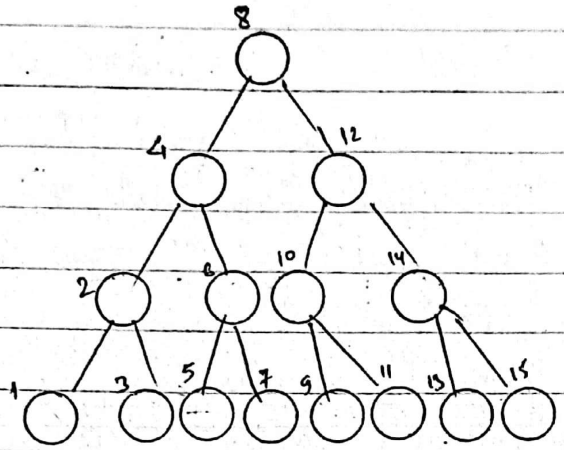
```
struct node
{
    int data;
    struct node * left;
    struct node * right;
};
void preorder (struct node * tree)
{
    if (tree != NULL)
    {
        printf ("%d", tree);
        preorder (tree -> left);
        preorder (tree -> right);
    }
}
```

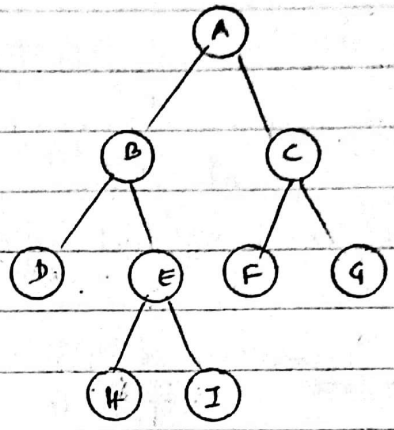
(ii) In order (LNR)

To traverse a non-empty binary tree in In-order we perform the following three operations:-

(a) traverse the left sub-tree in inorder (L)
(b) visit the root
(c) traverse the right sub-tree in inorder

1.5





→ D B H E I A F C G

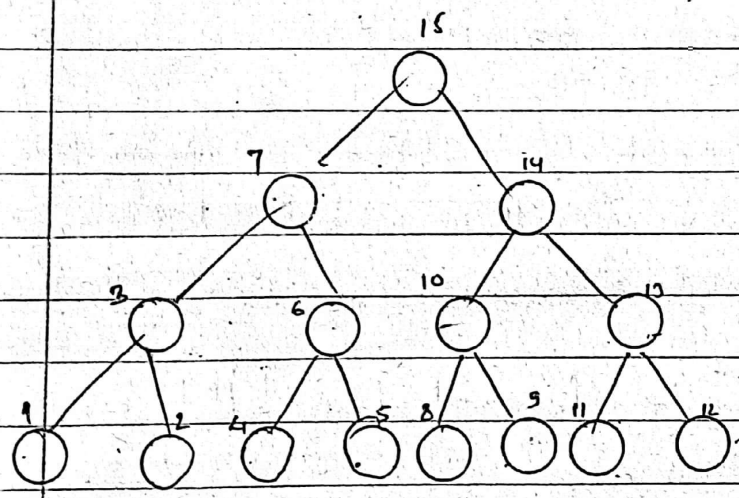
(b) Post order (L R N)

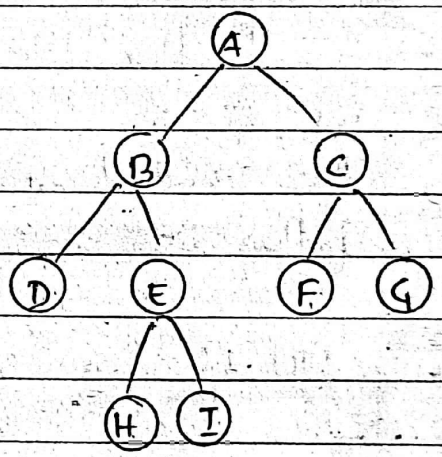
To traverse a non-empty binary tree in post-order, we perform the following three operations:-

(a) Visit the left sub-tree in post-order

(b) Visit the right sub-tree in post-order

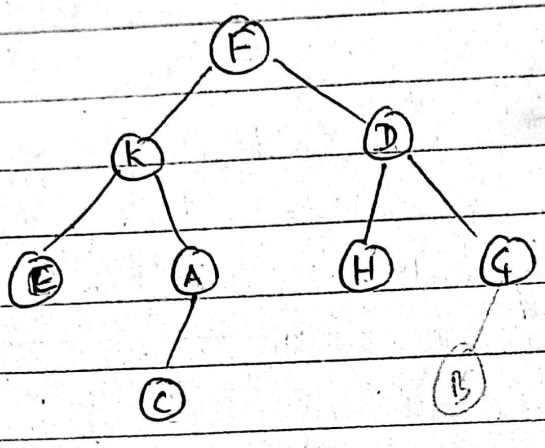
(c) Visit the node.





→ D H I E B F G C A

9. Similarly, G will be the right child of D.



checks.

NLR = F [KE(AC)] [DH(GB)]

LNR = [EK(CA)F] [HD(BG)]

= EKCAFHDBG

BINARY SEARCH TREE (BST)

A BST is an ordered binary tree such that either it is an empty tree or

(i) each data value in its right subtree is greater than the root value.

(ii) each data in its left sub-tree is less than the root value

(iii) left and right sub-tree are again binary search tree.

In BST, duplicate data values are not allowed.

For eg.

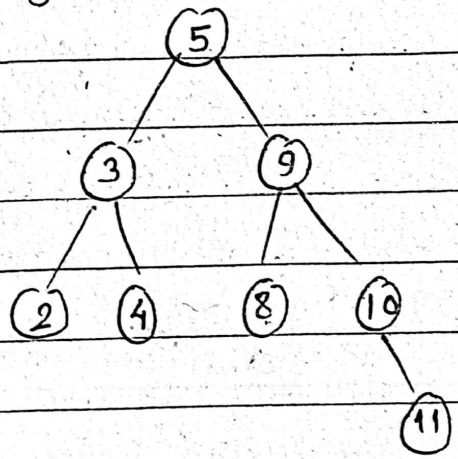
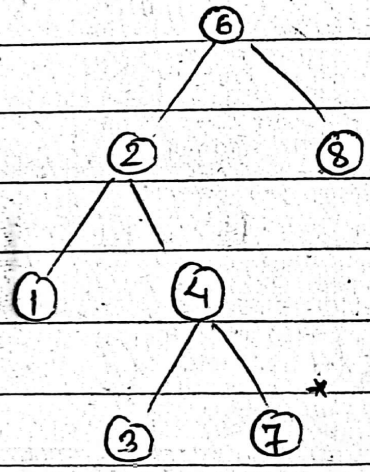


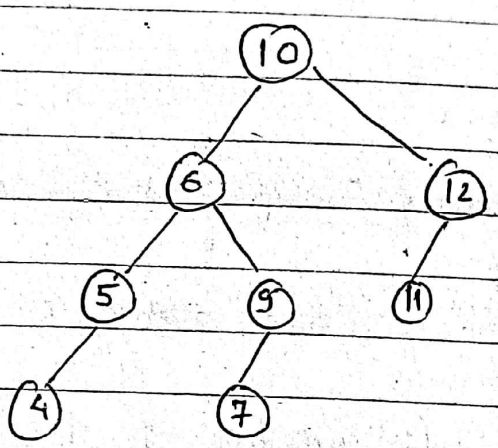
fig: BST



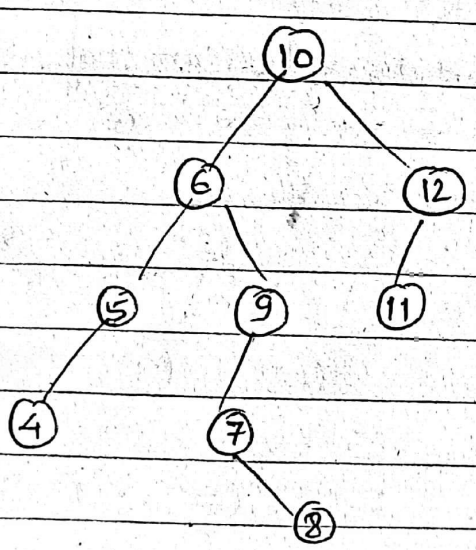
Here 7 is greater than root 6. So it is not a BST.

Inserting a node into binary search tree

To insert a node into BST, we must check whether the tree is already containing any node or not. If the tree is empty, then the new node is placed as the root node. If the tree is not empty, then proper location of new node is found and the new node becomes either left or right child of an existing node.



Insert node (8) in the above BST, then



Algorithm for inserting an item in BST.

Suppose T is BST and an item is given information then algorithm will be ;

1. Compare item with root node 'N' of the tree T

2. If item < N ,
   Then proceed to the left child of N
   else
   proceed to the right child of N.

3. Repeat step 1 and step 2 , until an empty sub-tree is meet.

4. Insert item in place of an empty subtree

5. Exit.

2)

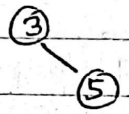
Create BST after inserting the following elements

3, 5, 11, 8, 4, 1, 12, 7, 2, 6, 10

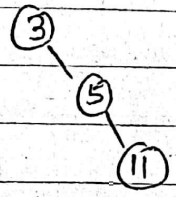
Sol^n: Since,

→ BST is empty, so (3) becomes the root node.

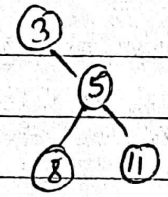
→ On inserting 5, since (5) is greater than (3), it becomes the right child.



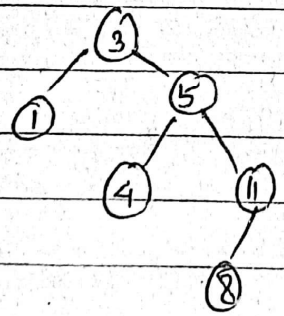
→ On inserting 11,



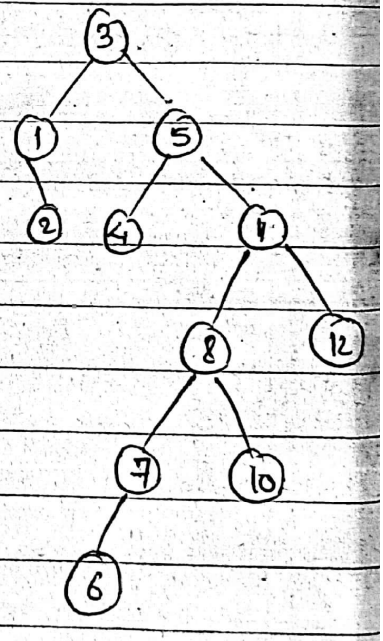
→ On inserting, 8,



→ On inserting 4, 1,



→ On inserting 12, 7, 2, 6, 10

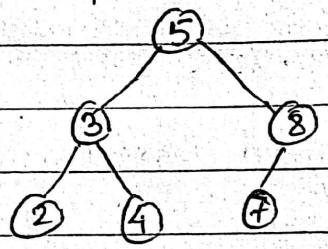


Deleting a node from BST

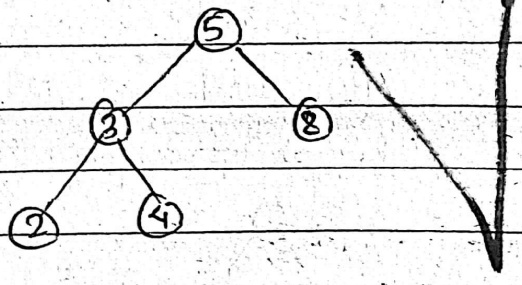
To delete a node from BST, first of all it is searched in BST. If it is found in the tree, then we have to consider the following possibilities otherwise a message is displayed indicating that the target node does not exists.

The following possibilities are :-

1. If the target node is a leaf node, then it can be deleted immediately by making its parent position pointing to null. For eg. Suppose we have a BST as :-

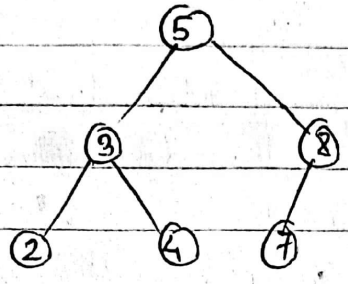


and we want to delete node (7), then after deleting node (7), the BST will be as :-

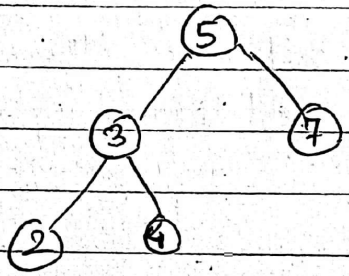


2. If the node has a single side, then node can be deleted after its parent. Adjust its pointer to point the child of the target node.

for eg.

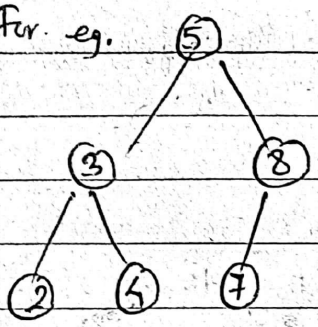


After deleting node (8), the BST will be :-

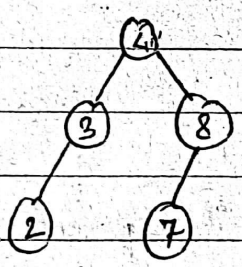


3. If the node to be deleted has two children, then the value of this node is replaced by the smallest value in the right sub-tree or largest value in its left sub-tr After that the empty node is recursively deleted by cons dering above two conditions.

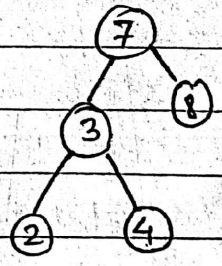
For eg.



After deleting node 5, the BST will have two solution.



'OR'



23 1 qst for exam.

Imp

AVL BALANCED TREE (named after, Adelson, Velskii, Landis)

In the name of Adelson, Verskey and Landies, it is called AVL tree. An AVL tree is a self balancing binary search tree. In an AVL tree, the height of two child subtrees of any node is differ by at most 1 ie height of left sub-tree and height of right sub-tree at most by 1. It is also called height balanced tree.

The balanced factor (BF) of a node is the height of its left sub-tree minus height of right sub-tree, ie,

ie: B.F = Height of Left-sub tree − Height of right sub tree

Therefore, in an AVL tree, the B.F of every node must be either −1, 0 or 1.

A node with any other than B.F is considered as unbalanced and requires rebalancing of the tree.

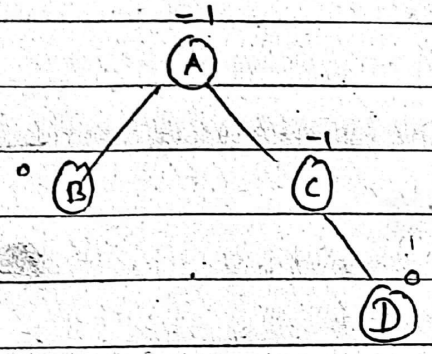
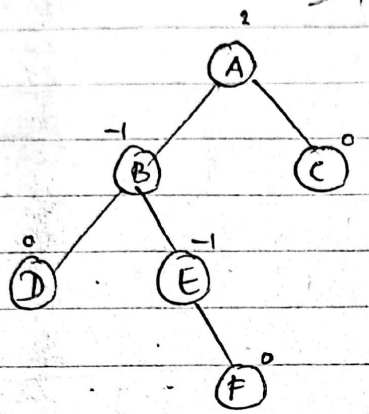


fig. It is a AVL balanced tree.



This is not a balanced tree

Building an AVL tree (Inserting a node into an AVL tree)

Inserting a new node into an AVL tree could violate the AVL tree property of height balance. Therefore it requires rotating the tree to satisfy the AVL property. The violation of AVL tree property may occur in four cases :-

(i) ~~Inserting~~ An insertion in left sub-tree of left ~~child~~ side

(ii) An insertion into right sub-tree of right ~~child~~. side

(iii) An insertion into left subtree of right child. side.

(iv) An insertion into right subtree of left side

To balance the tree we use rotation concept. For case I and case II, we require single rotation, and for case III and case IV, we require double rotation.

- FOR LR & RL rotation identify A, B & C then make A & B child of C

15

Algorithm to insert a node into an AVL tree

1. Insert the node in the same way as in BST.

2. Check the balance factor (BF) of each node after insertion

3. If we find a node with an unbalance (ie BF is not either 0, -1 or 1), stop ~~rotation~~ insertion.

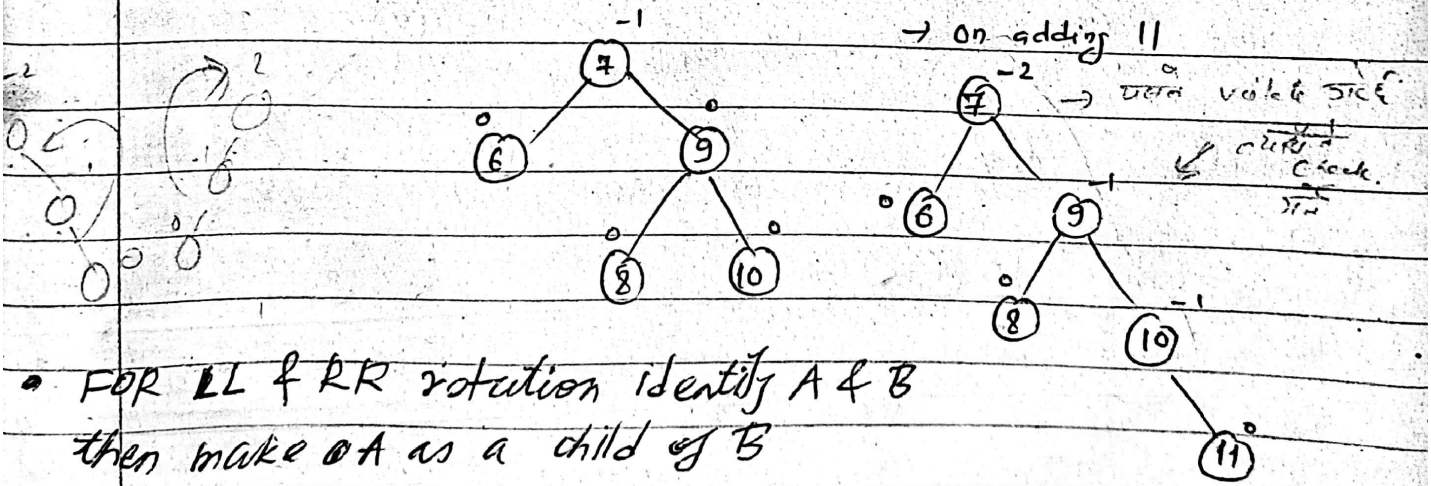
4. If these nodes are in a straight line, apply a single rotation to correct the unbalance.

5. If these nodes are in dog-leg pattern, apply a double rotation to correct the unbalance.

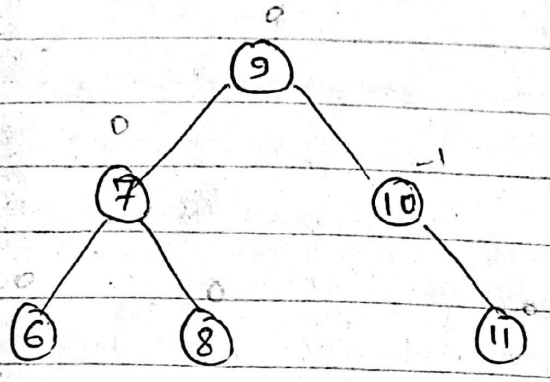
ROTATING TYPES

1. Inserting a node in the right sub-tree of right child.

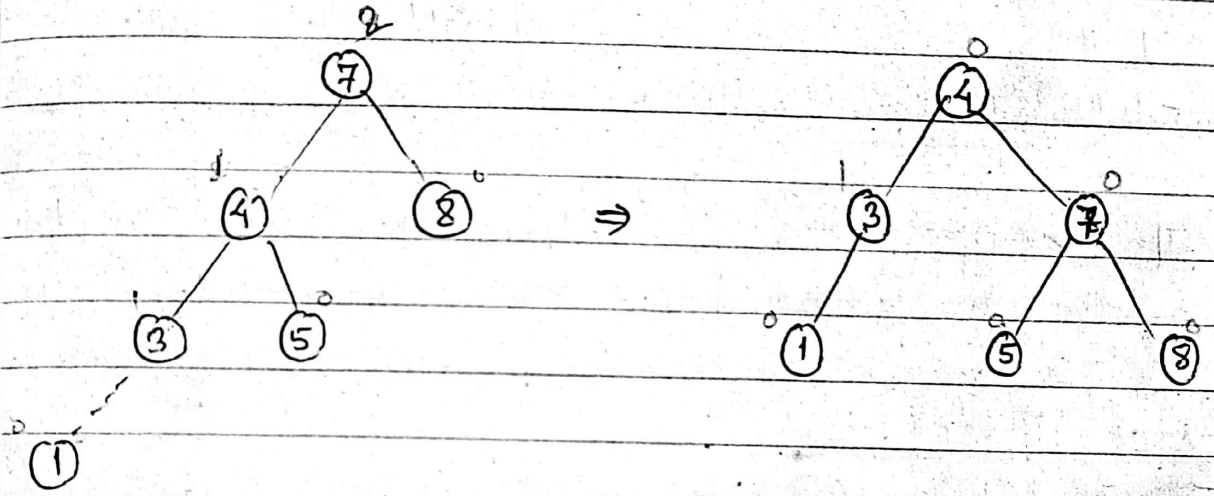
This involves SLR(single left rotation) at node. Suppose we have a BST tree as:-



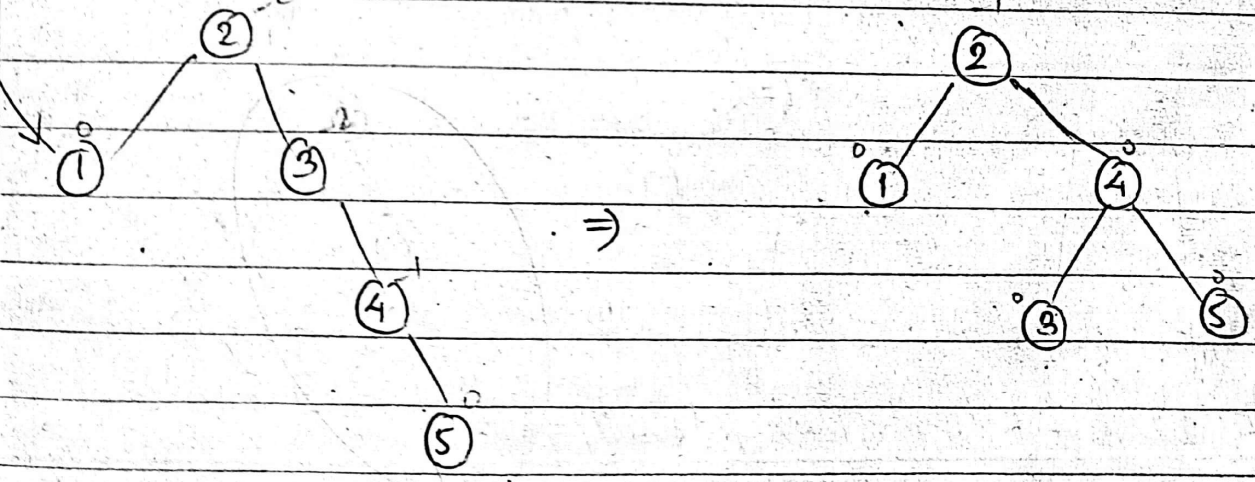
- FOR LL & RR rotation identify A & B then make A as a child of B



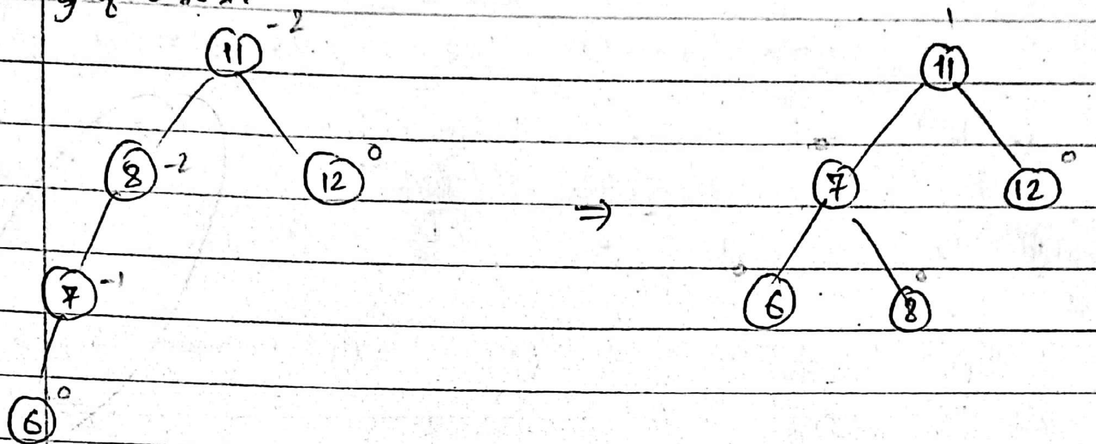
2. Inserting of a node into a left sub tree of left child involves single right rotation (SRR).



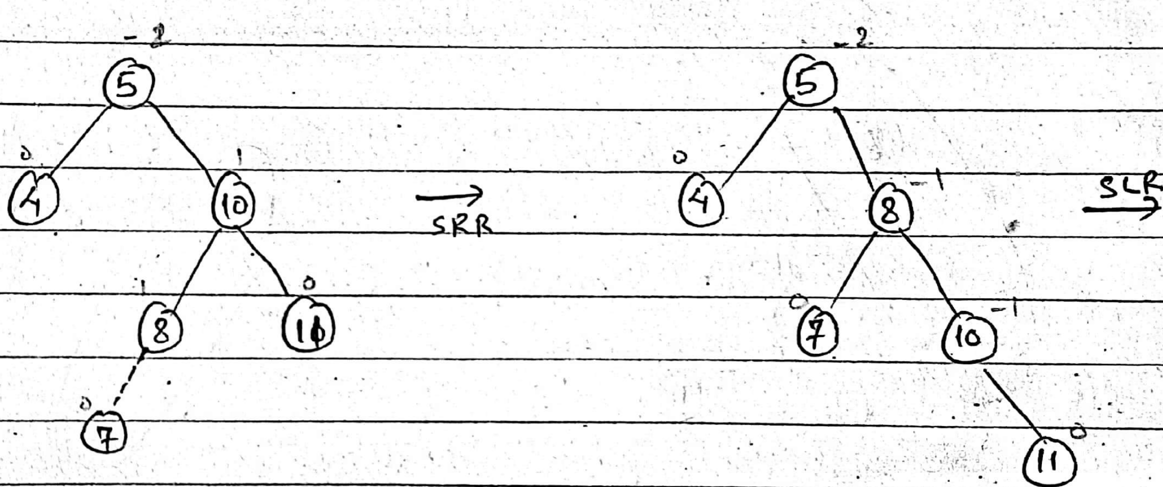
Eg. of case I:

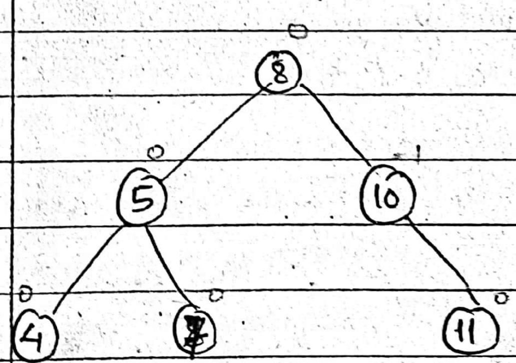


Eg of case II:



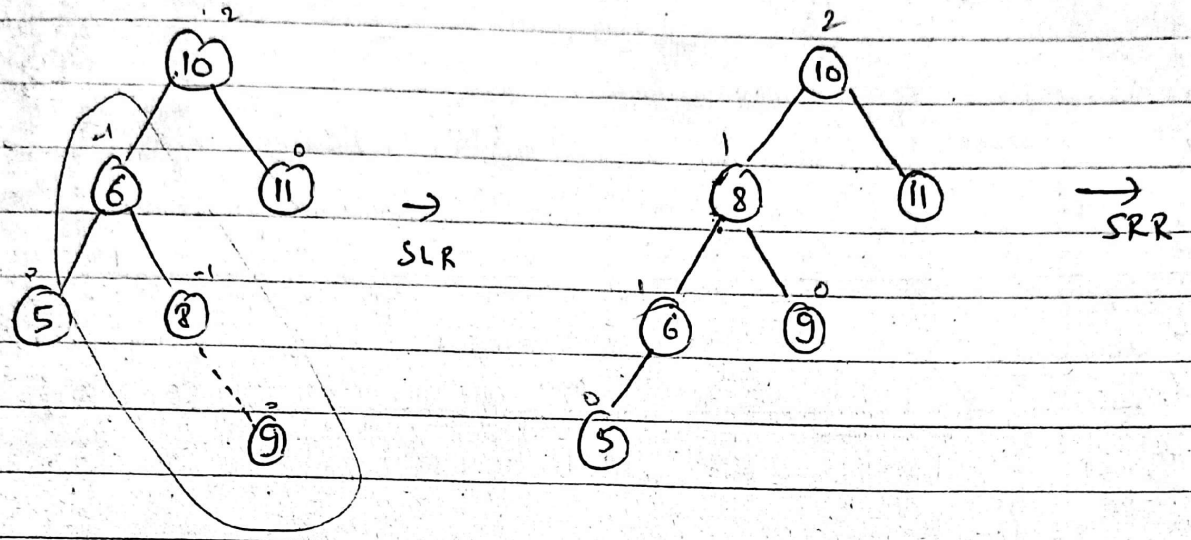
3. <u>Inserting of a node in the left sub-tree of a right child</u> involves double ~~right~~ rotation. (SRR + SLR) .

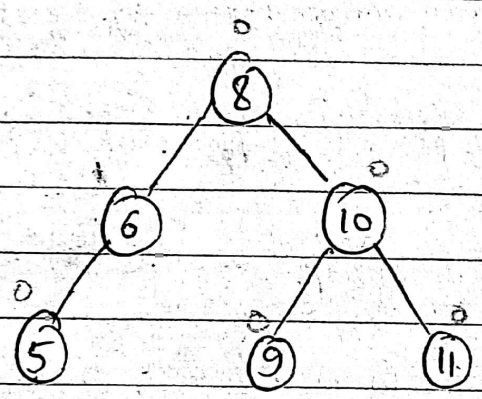




[illegible]

4. After inserting a node in the right sub tree of left side involves double rotation (ie. SLR + SRR)



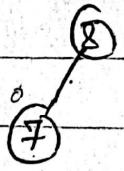


1M// Construct an AVL tree 8, 7, 10, 6, 5, 4, 3

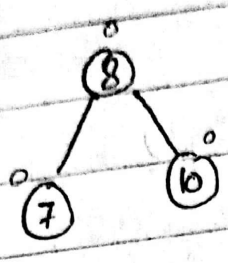
Soln:

Node (8) becomes the root node

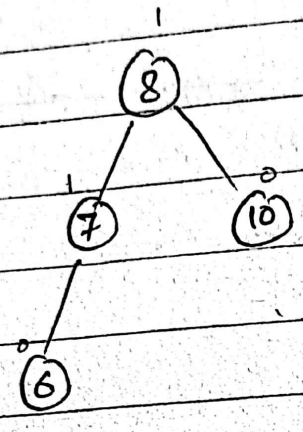
Add (7) in left



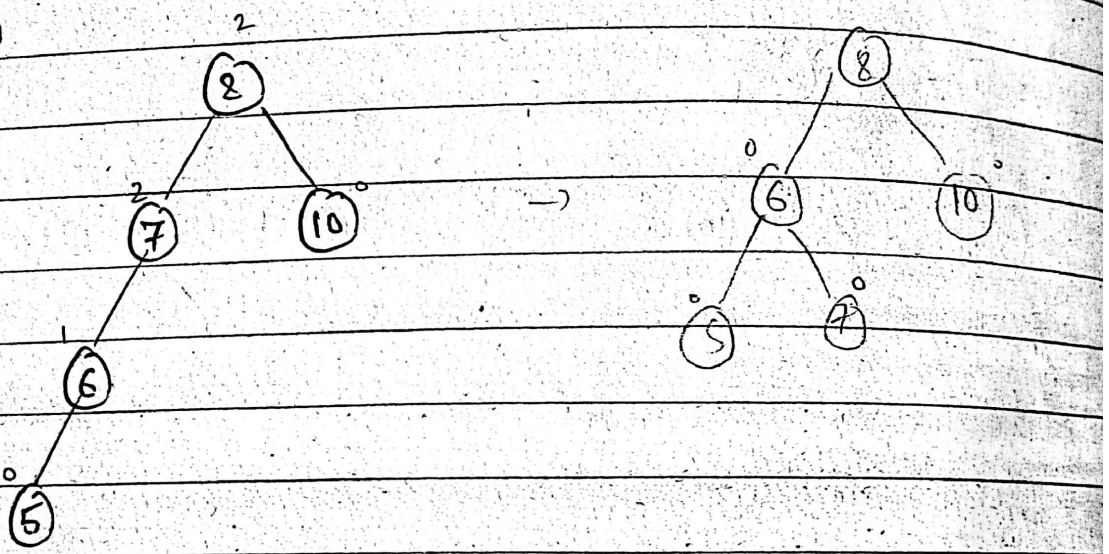
Add (10)



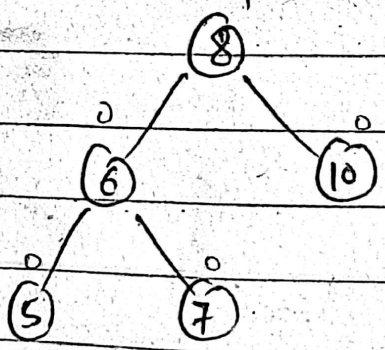
Add (6)



Add (5)

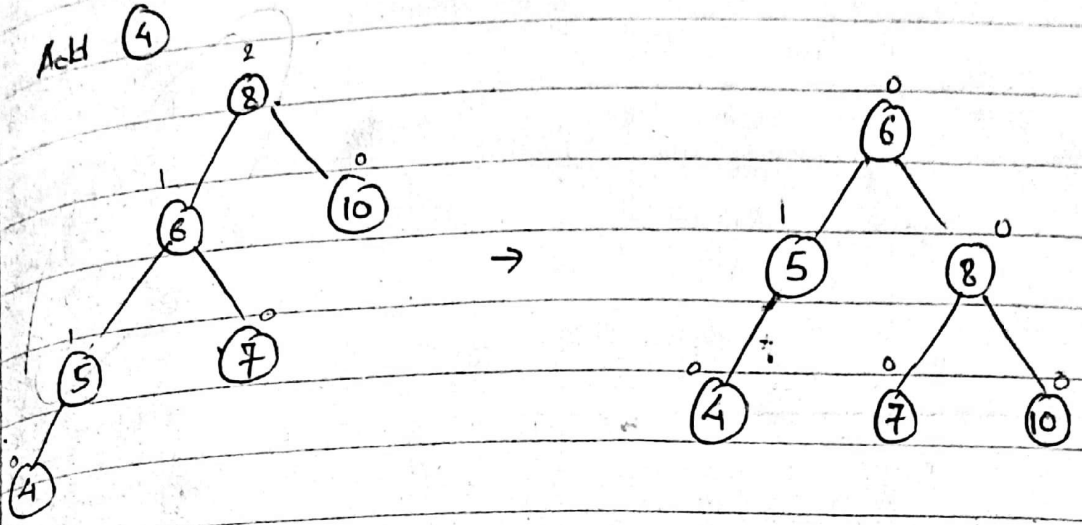


Now balancing the AVL tree.

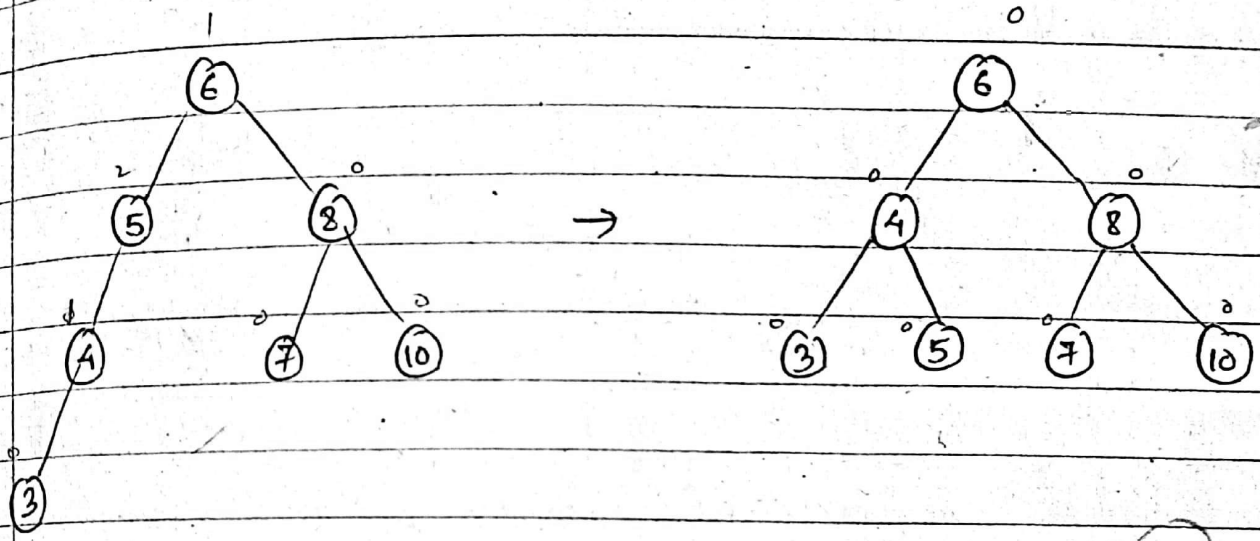


10

Add (4)



Add (3)



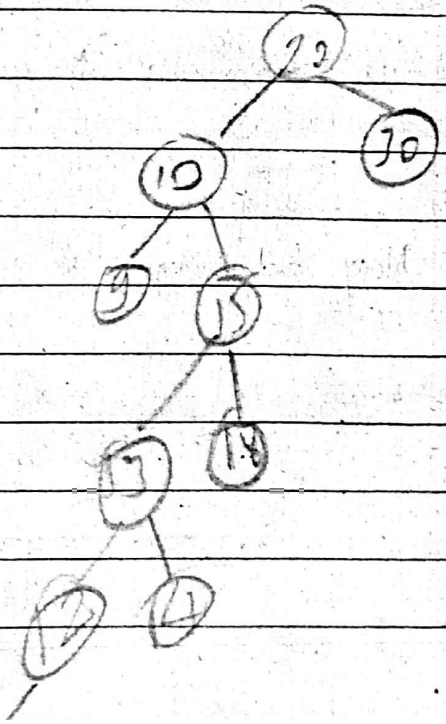
H/W

Construct an AVL tree :-

① 20, 10, 30, 9, 15, 18, 13, 12, 14, 11

② 1, 2, 3, 4, 5, 6, 7, 14, 13, 11

③ 3, 5, 11, 8, 4, 1, 12, 7, 2, 6, 10

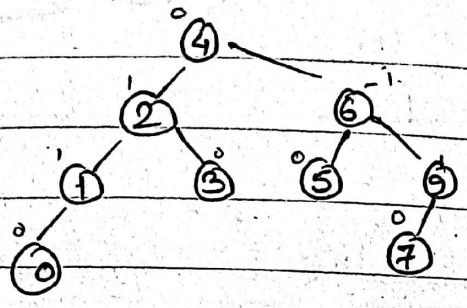


Deletion of a node in AVL tree

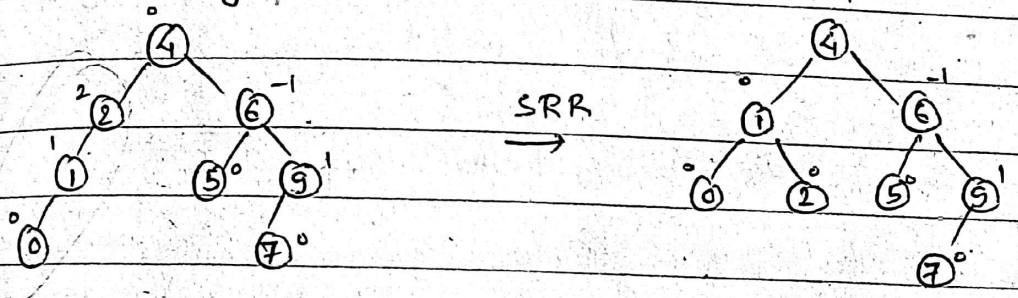
→ Delete a node in BST

→ check the balance. If the tree is not balance, then balance it through rotation.

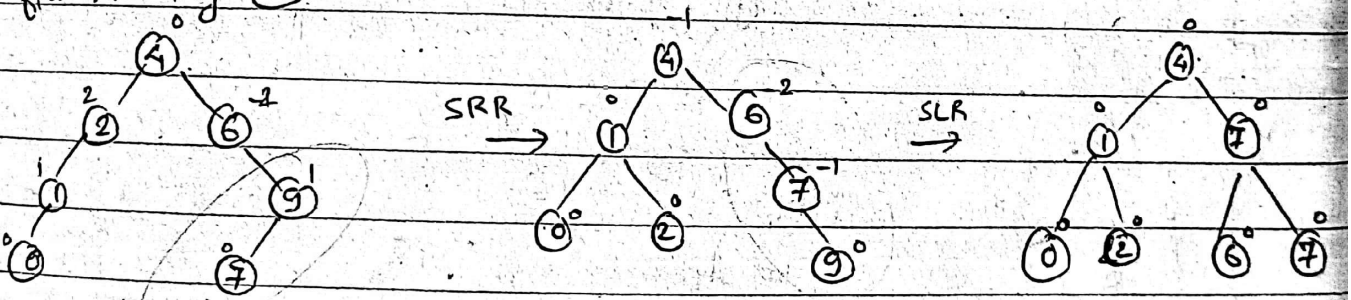
Consider the following AVL tree; then show ~~that~~ the deletion of 3, 5 and 4 in order.



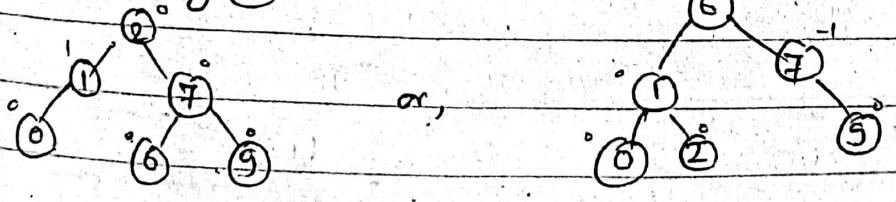
Sol^n: After deleting (3),



After deleting (5)



After deleting (4)



35

B - TREE

A B-tree of order n is a tree with the following structural properties:-

(i) The root is either leaf or is between 2 and m.

(ii) A non-leaf node except the root node must have children between $m/2$ and m.

(iii) All leaf nodes must be at same depth.

(iv) Each node has one fewer keys than the no of its children with a maximum m-1 key.

(v) Key are arranged in a defined order within a node (increasing order), all keys in the sub-tree to the left of the key are predecessors of the tree and keys on the right. are successor of the tree

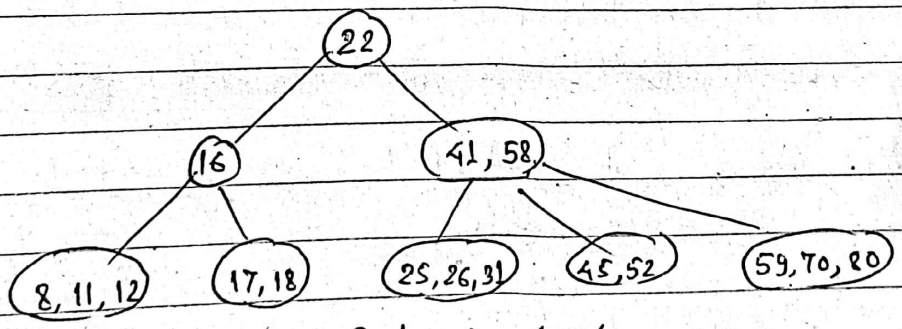


Fig. B-tree of order 4

→ B-tree of order - 3 is called 2-3 trees

→ B-tree of order - 4 is called 2-3-4 trees

→ B-tree of order - 5 is called 2-3-4-5 trees

Building of a B-tree (B-tree insertion)

To insert a new node in a B-tree, first of all the proper place for the new key is searched and that new key is placed on that position. If the node can accomodate the new key, then insertion is complete. If the node overflow (because of upper bound of the node), then the node is splitted. The node is splitted into three parts. The middle key is passed upward and inserted into the ~~upper~~ parent and two children are created on the left and right. The parent may also be overflow. Then the splitting process continue until the tree is adjusted.

Construct a B-tree of order 4 with the following key values.
1, 5, 6, 2, 8, 11, 13, 18, 20, 7, 9

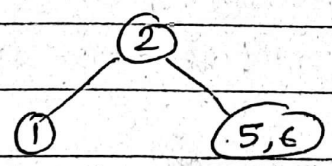
Sol^{n}: # Inserting (1) as root ~~tree~~ node.

(1)

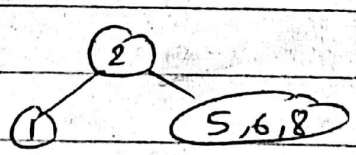
On inserting (5) and (6)

(1, 5, 6)

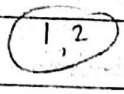
On inserting (2), it ~~voilet~~ voilates, and it must be splt



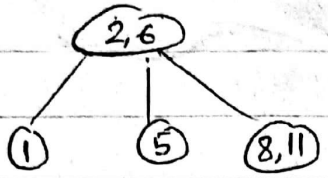
On inserting (8),



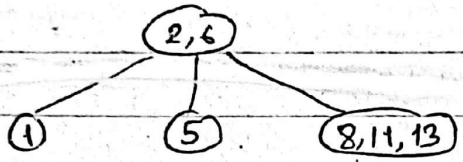
or



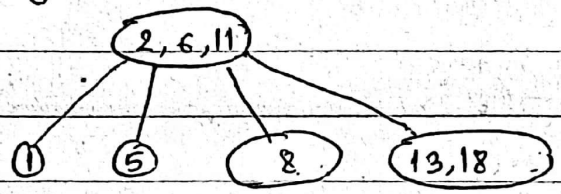
On inserting (11)

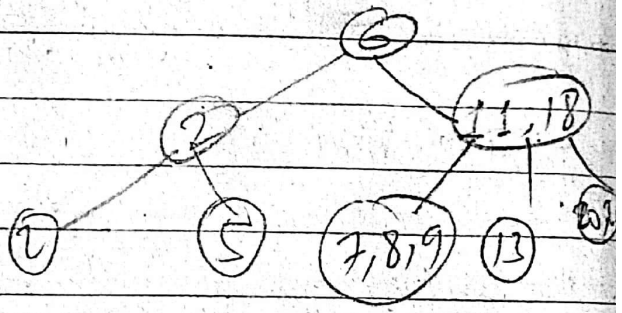


On inserting (13)

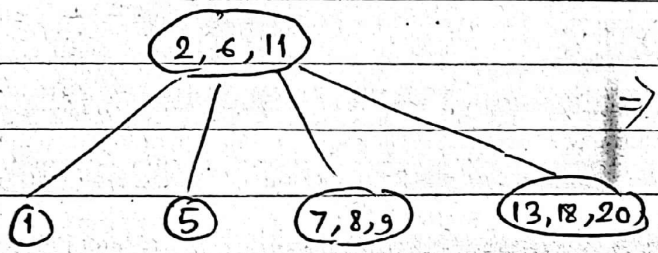


On inserting (18)





On inserting 20, 7, 9

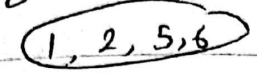


Construct of a B tree of order 5 with the following key values

1, 5, 6, 8, 11, 13, 18, 20, 7, 9, 12, 10

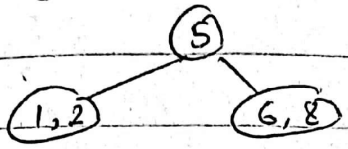
Inserting (1) as root node.

On inserting 5, 6, 2

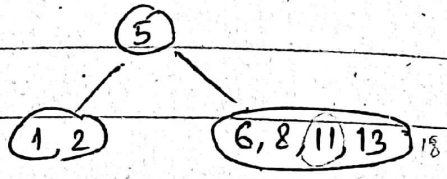


On inserting (8)

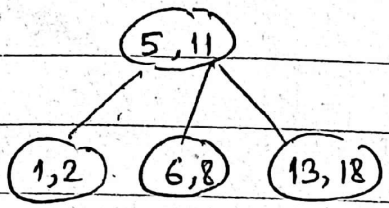
Mid value



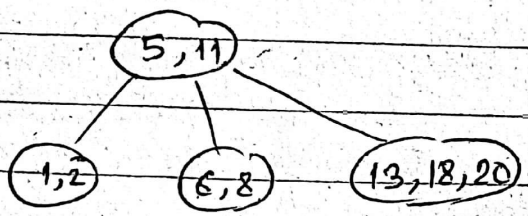
On inserting 11, 13



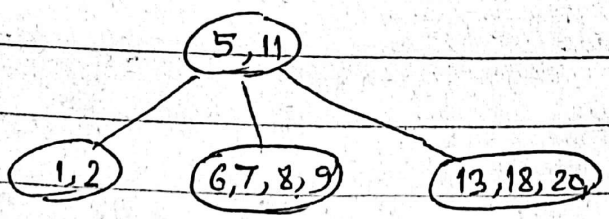
On inserting (18)



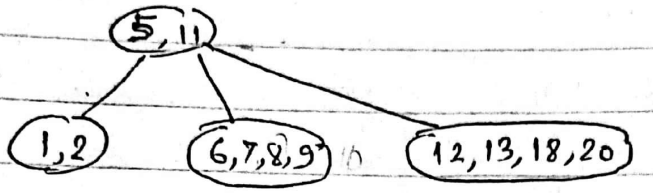
On inserting 20,



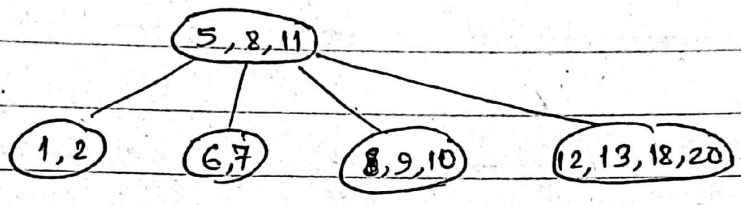
On inserting 7, 9



On inserting 12



On inserting 10,



HUFFMAN'S ALGORITHM

It is a method for building an extended binary tree with a weight path length from a set of given weight. Initially construct a forest of single tree, one associated with each weight. If there are at least two nodes, choose the two trees with the least weight associated with their roots and replace ~~that~~ them with a new node. Construct by ~~created~~ creating a root node whose weight is the sum of weight of the root of two trees. This process is repeated until the forest consists of one tree.

Algorithm to build Huffman tree

1. Get the string.

2. Get the frequency occurance of each symbol.

3. Select the symbol having the lowest probabilities of occurence

4. Combine these two into a single and again choose two symbols with smallest frequency and then combine them again.

5. Repeat it till final symbol occur.

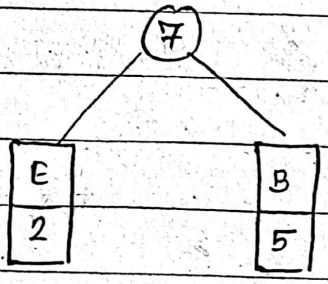
Example : Building a Huffman tree

Data item	A	B	C	D	E	F	G	H
Frequency	22	5	11	19	2	11	25	5

STEP 1: Sort the above in ascending order.

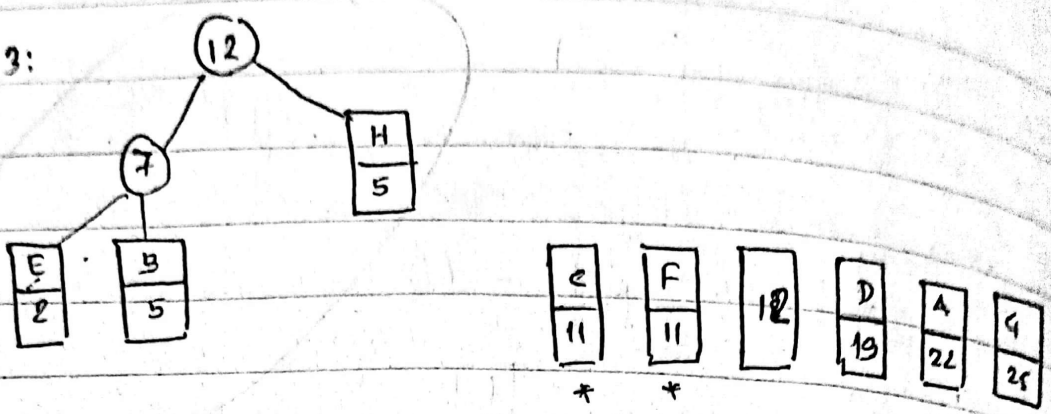
E	B	H	C	F	D	A	G
2	5	5	11	11	19	22	25
*	*						

STEP 2:

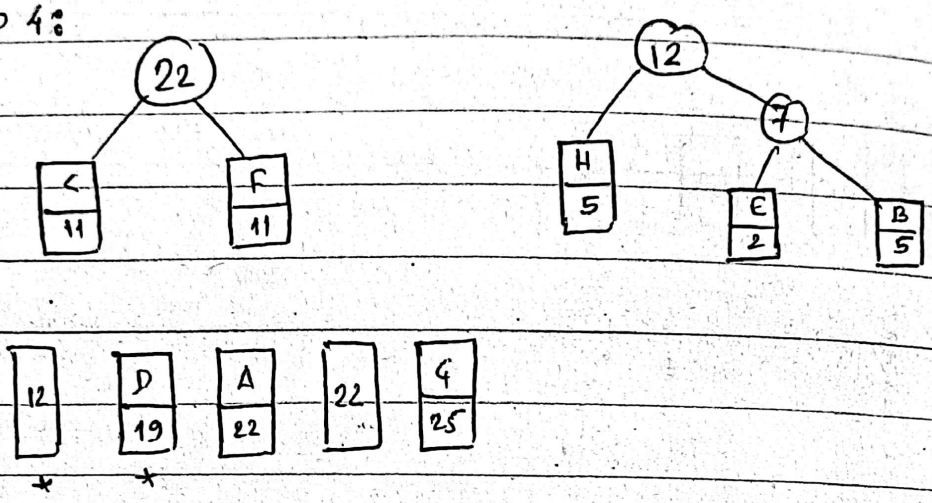


H	7	C	F	D	A	G
5		11	11	19	22	25
*	*					

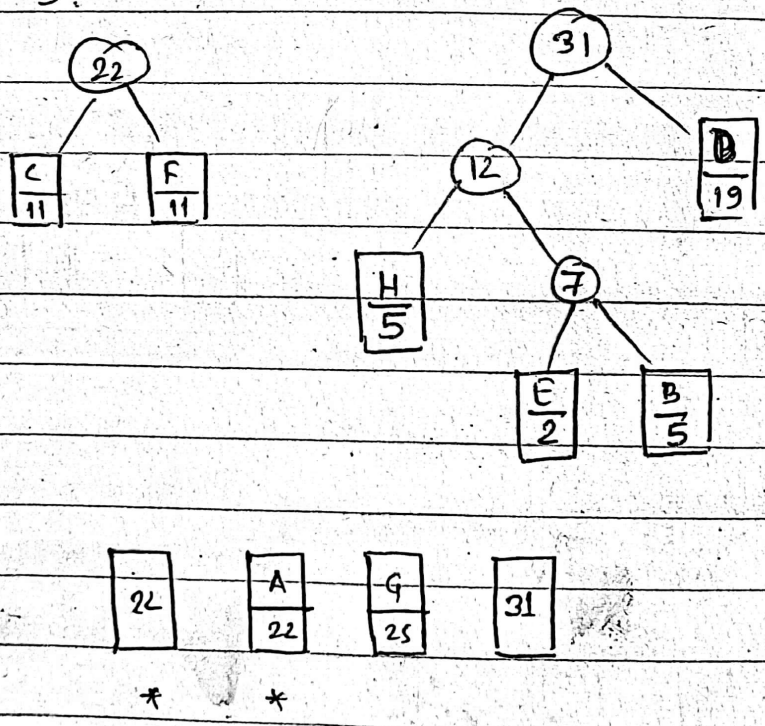
STEP 3:



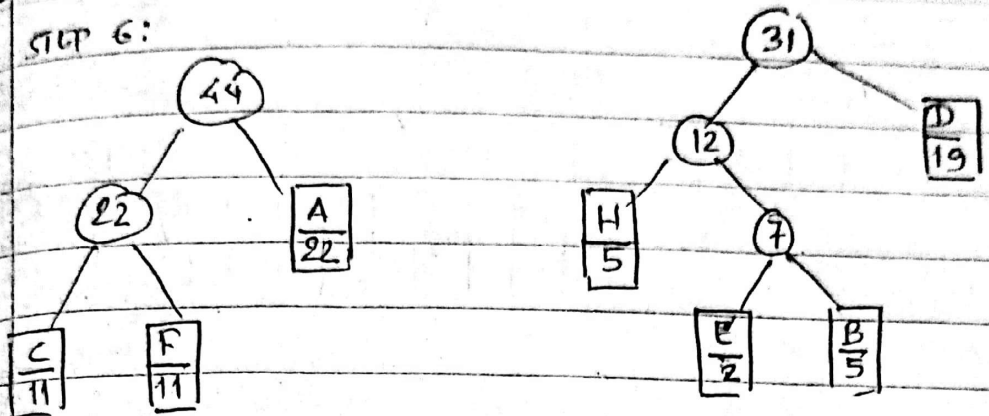
STEP 4:

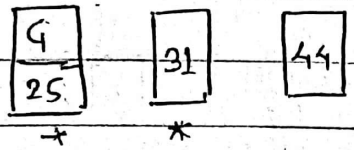


STEP 5:

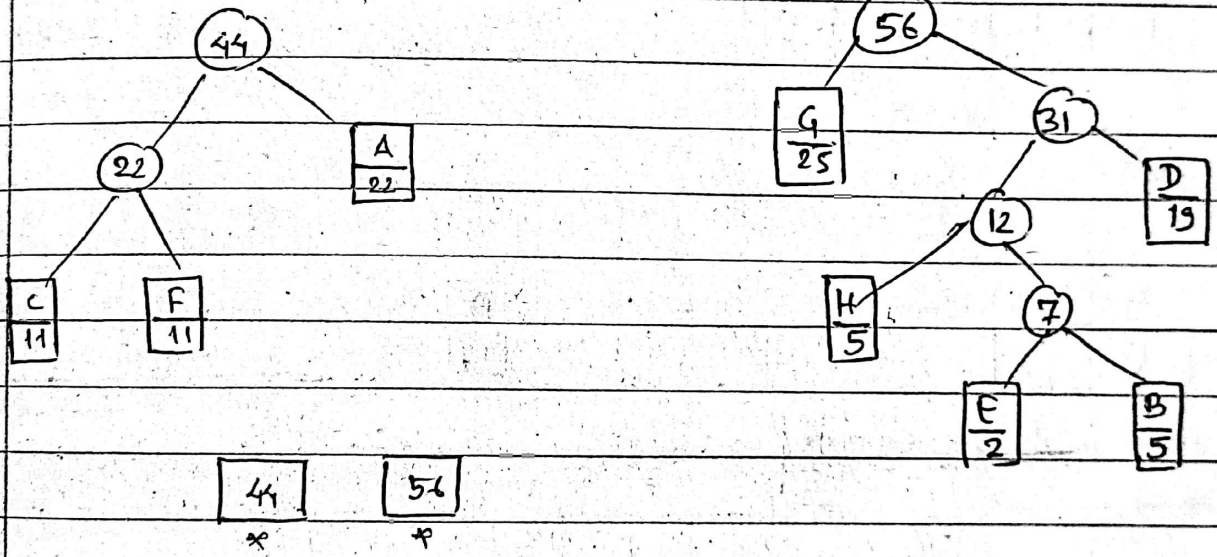


STEP 6:





STEP 7:



STEP 8: