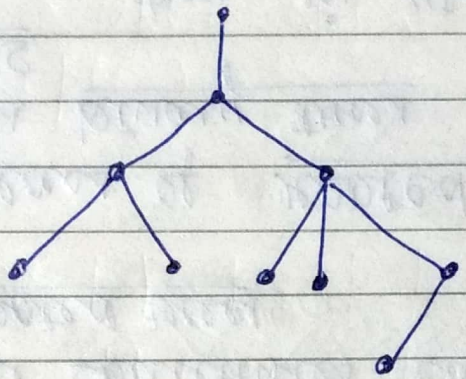
Trees

- Non linear Data structure
- used to rep data containing a hierarchical relationships b/w elements
- A tree is a connected graph w/o any circuits

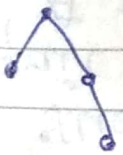


tree

- tree with one vertex

" " 2 vertices

" " 3 "



" " 4 "

Some properties of trees :-

1) There is one and only one path b/w every pair of vertices in a tree, T.
2) A tree with n vertices has n-1 edges
3) A graph is a tree if and only if it is minimally connected.
4) Any connected graph with n vertices and n-1 edges is a tree.

✓ A tree in which one vertex is distinguished from all the others is called a rooted tree.
✓ Generally, tree means tree w/o any root so they are sometimes called free trees or nonrooted trees.

A special class of rooted trees called binary rooted trees or binary trees

BINARY TREES.

- defined as a tree in which there is exactly one vertex of degree 2 and each of the remaining vertices is of degree one or three.

A vertex with degree 2 is known as the root. Therefore, every binary tree is a rooted tree.

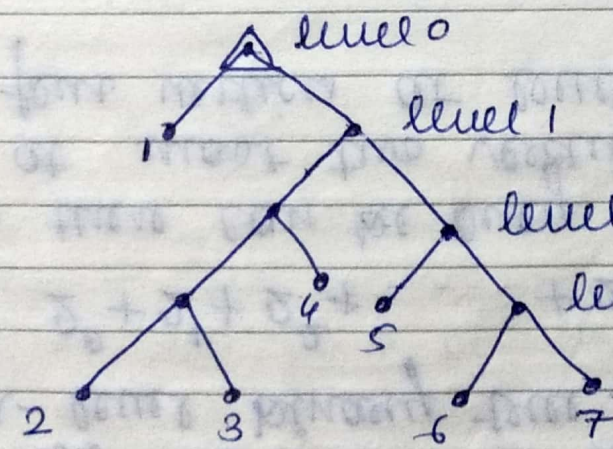
Properties of Binary trees :

1) The no. of vertices n in a binary tree is always odd. This is because
   a) there is exactly one vertex of even degree
   b) remaining $n-1$ vertices are of odd degrees.

2) Let p be the no. of pendant (leaf nodes) vertices in a binary tree T. Then, $n-p-1$ is the no. of vertices of degree 3.
   Therefore no. of edges in T equals

$$\frac{1}{2}[p + 3(n-p-1) + 2] = n-1$$

Hence, $p = \frac{n+1}{2}$



$n = 13$

$p = \frac{13+1}{2}$

$= 7$

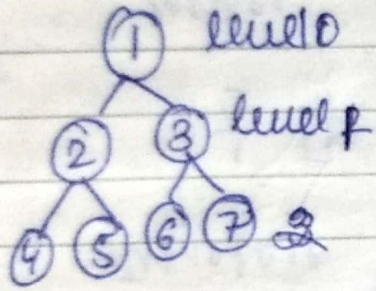
3) To find the maximum no. of vertices possible in a k-level binary tree,

$$2^0 + 2^1 + 2^2 + \dots + 2^k \geq n$$

because there can be only one vertex at level 0, at most two vertices at level 1, at most four vertices at level 2 and so on.

eg→ :-

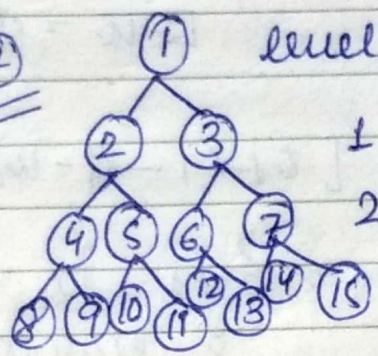
(1)



$2^0 + 2^1 + 2^2 = 7$

$\therefore 7 \geq 7$

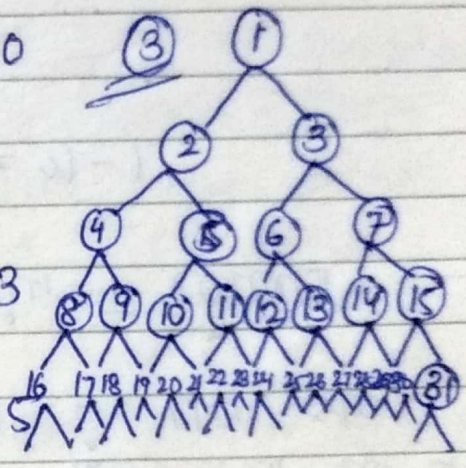
(2)



$2^0 + 2^1 + 2^2 + 2^3 = 15$

$\therefore 15 \geq 15$

(3)



$n = 63$

$2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 = 63$

$\therefore 63 \geq 63$

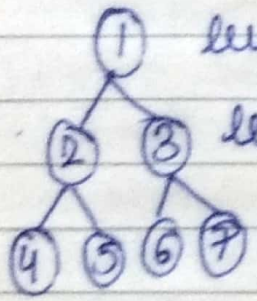
4) Maximum level, l_{max} of any vertex in a binary tree is called the height of the tree

The min^m possible height of the n-vertex binary tree is

$$min\ l_{max} = [\log_2(n+1) - 1]$$

eg → for $n = 7$

$$min\ l_{max} = [\log_2(7+1) - 1]$$



level 0
level 1
level 2

$= \log_2 8 - 1$

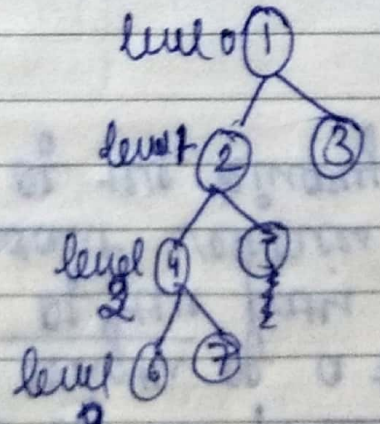
$= \log_2 2^3 - 1$

$= 3 - 1 = 2$

5) $max\ l_{max} = \frac{n-1}{2}$

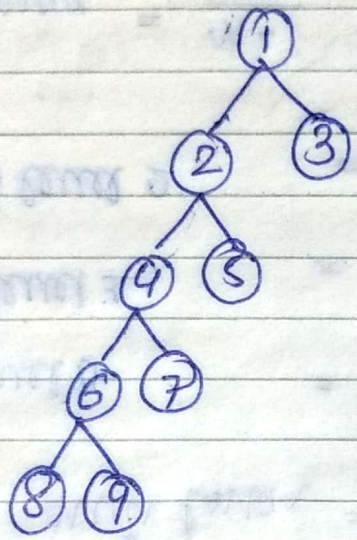
$= \frac{7-1}{2}$

$= 3$



6) The sum of the levels of all pendant vertices known as the path length of a tree, can be defined as the sum of the path lengths from the root to all pendant vertices.

The path length of the binary tree for



is

4+4+3+2+1 = 14. (Path lengths)

Traversing in a Binary Trees :-

3 types :-

1) Preorder → Root, left, right.
2) Inorder → left, Root, right
3) Postorder → left, right, Root

① PREORDER

ALGO :-

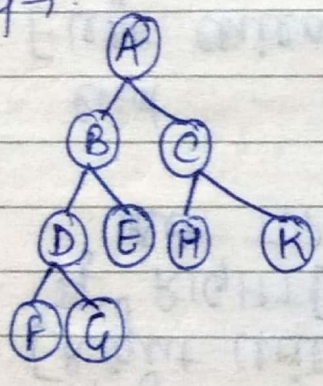
Initially push NULL onto stack and then set PTR := ROOT.

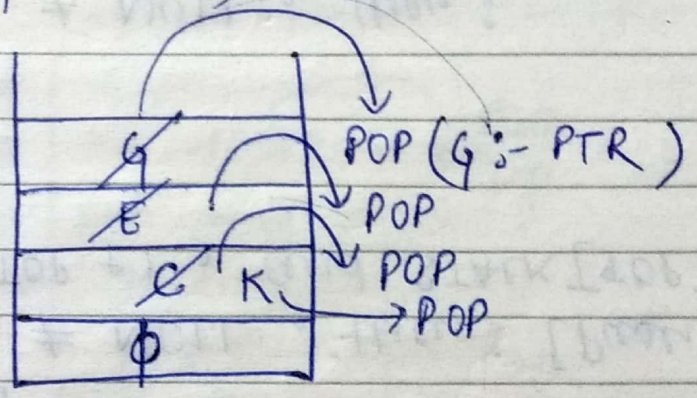
Then repeat the foll. steps until PTR = NULL.

a) Proceed down the left most path rooted at PTR, processing each node N on the path and pushing each right child R(N), if any, onto stack. The traversing ends after a node N with no left child L(N) is processed.

b) [Backtracking] Pop and assign to PTR the top element on stack. If PTR ≠ NULL then return to Step (a); otherwise exit.

eg →





PTR :- A

~~PTR~~ ~~PTR~~ PTR

PROCESS :- A, B, D, F, G, E, C, H, K

Initial : STACK : ϕ

Set PTR = A, root of tree T.

PREORD(INFO, LEFT, RIGHT, ROOT)

1) [Initially push NULL onto STACK & initialize PTR]
   Set TOP := 1, STACK[1] := NULL & PTR := ROOT.
2) Repeat Steps 3 to 5 while PTR ≠ NULL.
3) Apply PROCESS to INFO(PTR).
4) [Right child ?]
   If RIGHT[PTR] ≠ NULL, then: [Push on STACK]
   Set TOP := TOP + 1, and STACK[TOP] := RIGHT[PTR].
   end
5) [Left child ?]
   if LEFT[PTR] ≠ NULL, then:
   Set PTR := LEFT[PTR]
   else: [Pop from STACK]
   Set PTR := STACK[TOP] & TOP := TOP - 1
   end
   end of Step (2)
6) Exit.

② Inorder

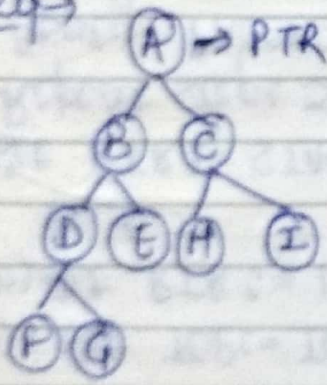
ALGO:

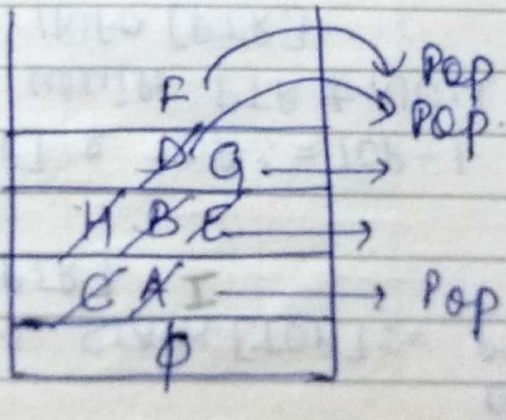
Initially push NULL onto STACK and then set PTR := ROOT. Then repeat the foll. steps until NULL is popped from STACK.

a) Proceed down the left most path rooted at PTR, pushed each node N onto STACK and stopping when a node N with no left child is pushed onto STACK.

b) [Backtracking] Pop & process the nodes on STACK. If NULL is popped, then EXIT. If a node N with a right child R(N) is processed, set PTR = R(N)

eg →





✓ PTR := A

F, D, G, B, E, A, H, C, I

FDGBEAHIC

INORD(INFO, LEFT, RIGHT, ROOT.

1) [Push NULL onto STACK & initialize PTR.]
   Set TOP := 1, STACK[] := NULL & PTR := ROOT
2) Repeat while PTR ≠ NULL: [Pushes left-most path onto STACK]
   a) Set TOP := TOP+1 & STACK[TOP] := PTR.
   b) Set PTR := LEFT[PTR].
   end
3) Set PTR := STACK[TOP] & TOP := TOP-1
4) Repeat steps 5 to 7 while PTR ≠ NULL: [Backtracking]
5) Apply PROCESS to INFO[PTR].
6) [Right child ?]
   if RIGHT[PTR] ≠ NULL, then:
   a) Set PTR := RIGHT[PTR].
   b) Go to Step 3.
   end of if.
7) Set PTR := STACK[TOP] and TOP := TOP - 1
   [end of step 4]
8) Exit.

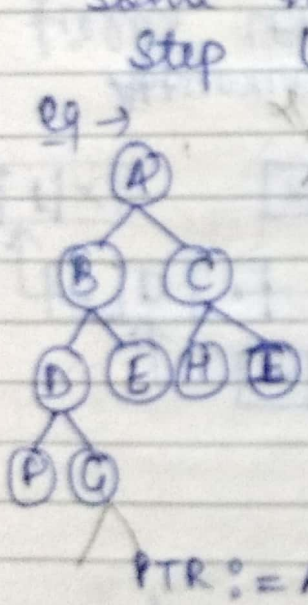
POSTORDER

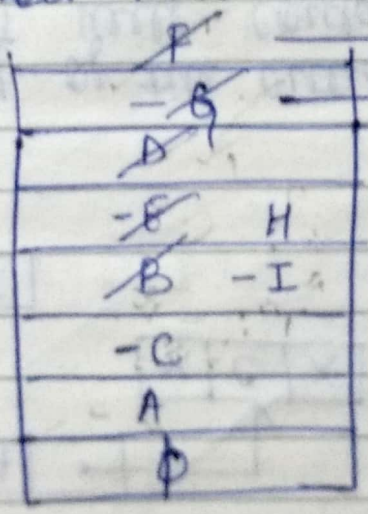
ALGO :-

Initially push NULL onto STACK & then set PTR := ROOT. Then repeat the following steps until NULL is popped from STACK

a) Proceed down the left-most path rooted at PTR. At each node N of the path, push N onto stack and, if N has a right child R(N), push -R(N) onto STACK.

b) [Backtracking]. Pop and process positive nodes on STACK. If NULL is popped, then EXIT. If a negative node is popped, i.e., PTR = -N for some node N, set PTR = N and return to Step (a).

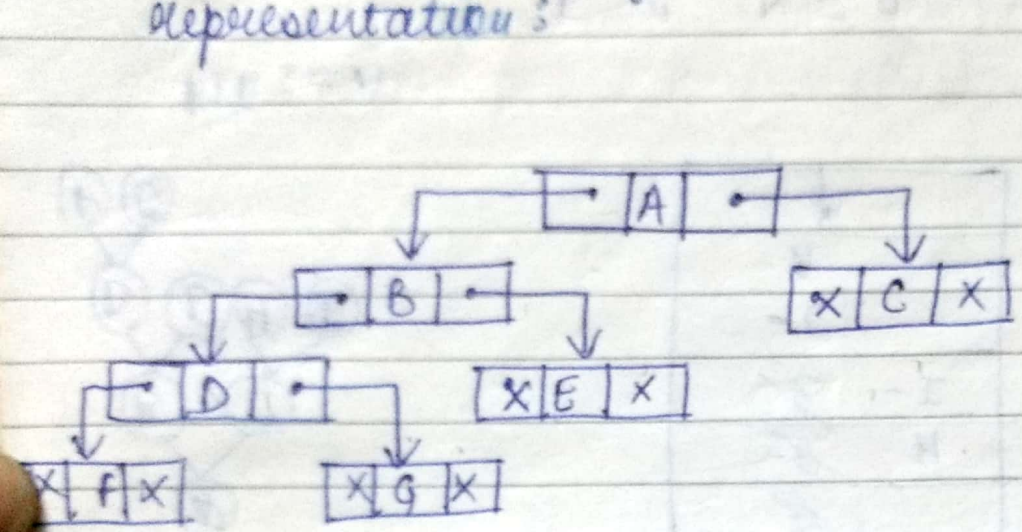


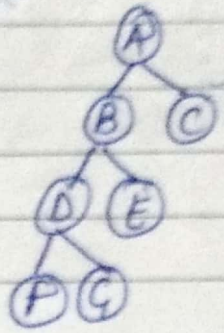


F, G, D, E, B, H, I, C, A

THREADED BINARY TREE

Consider a binary tree in the form of linked representation:





Approximately half of the entries in the ptr fields LEFT & RIGHT will contain null elements. This space may be more efficiently used by replacing the null entries by some other types of info. And null pointers are replaced by special pointers which point to nodes higher in the tree. These special pointers are called THREADS and binary tree with such pointers are called THREADED TREES.

There are two ways to thread a binary tree T.

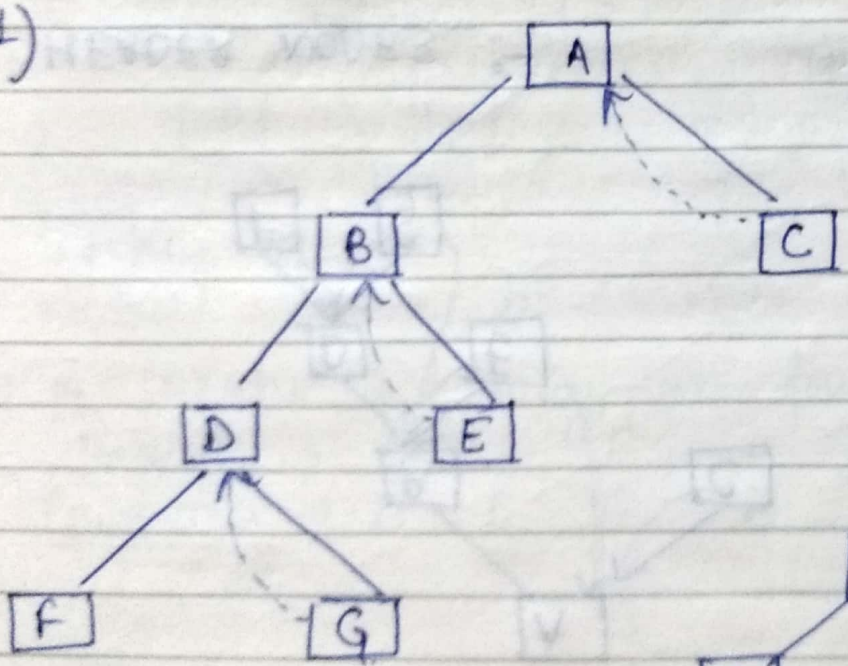
1) One-way threading 2) Two-way threading

Left threaded tree

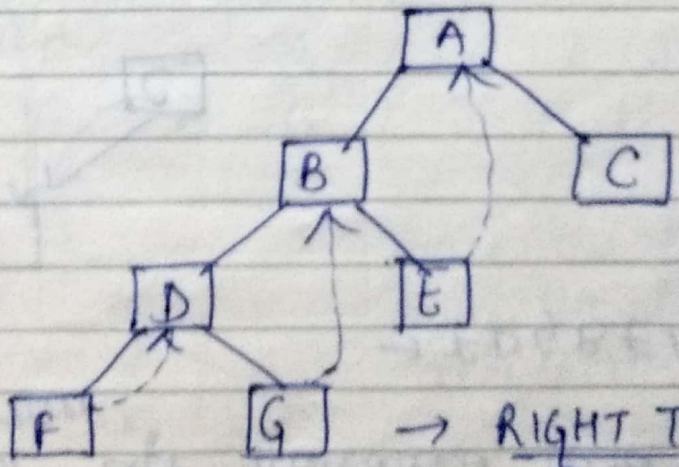
Right threaded tree

Take the Inorder :- F D G B E A C

1)



→ LEFT THREADED BINARY TREE

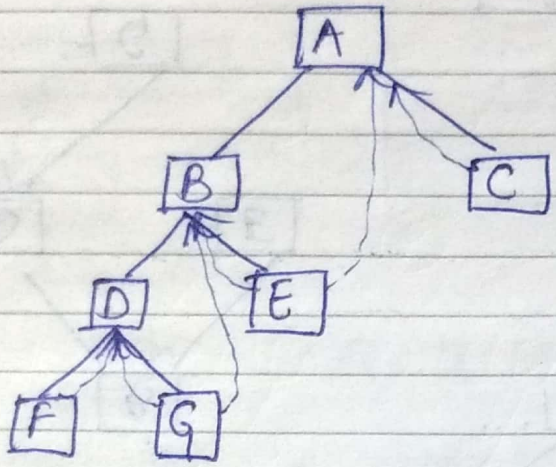


→ RIGHT THREADED BINARY TREE

2) TWO WAY THREADING

It contains both Left threaded and Right threaded Binary tree.

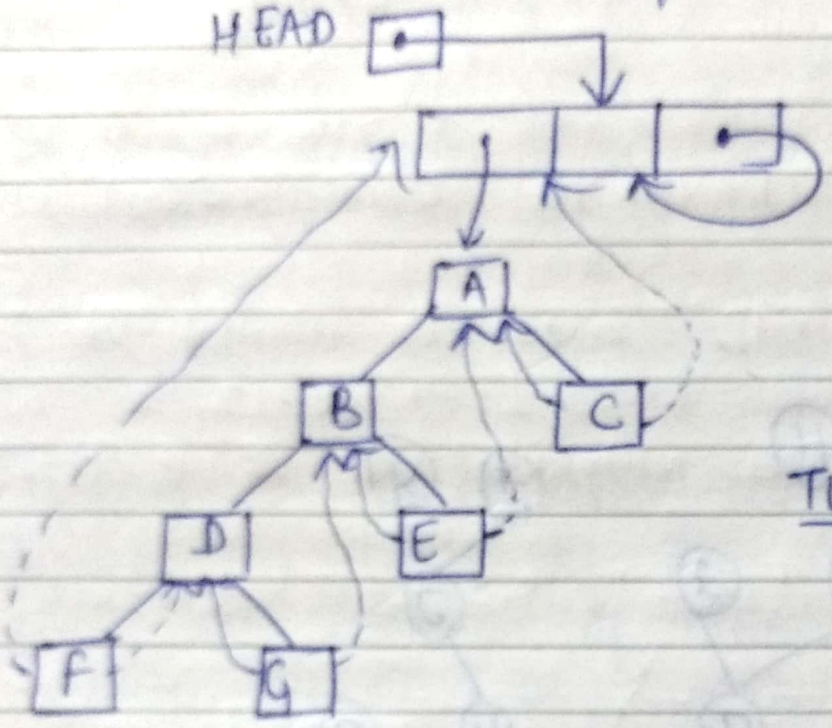
→ FDGBEAC



HEADER NODES :-

Suppose a binary tree T is maintained in mym by means of a linked representation. Sometimes an extra, special node called a header node is added to the beginning of T. When this extra node is used, the tree pointer variable HEAD will pt. to the header and left ptr of

the header node will pt. to the root of T.

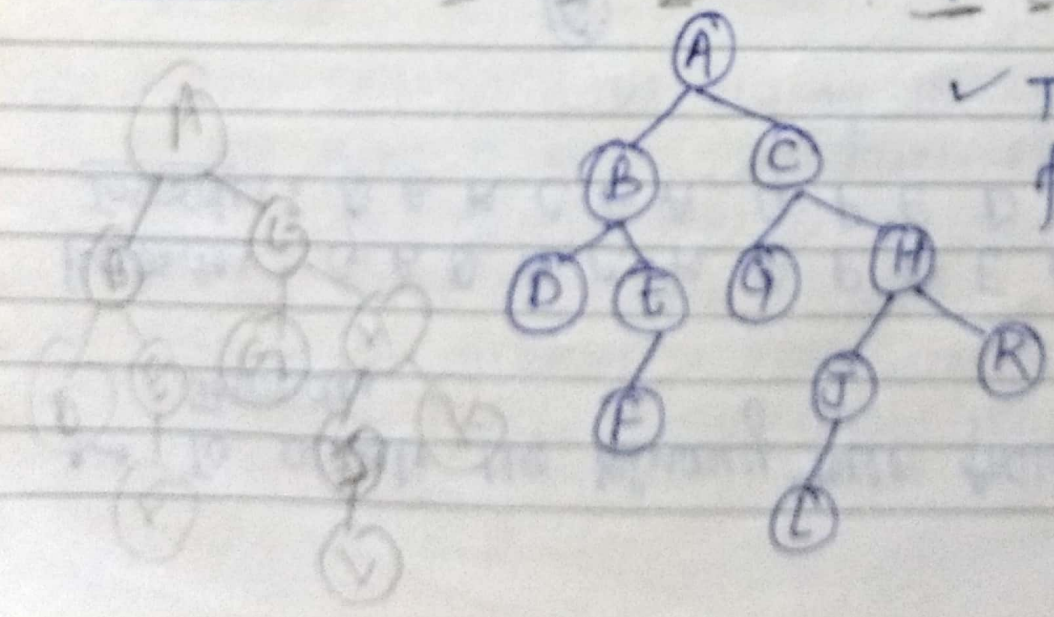


TWO WAY THREADING WITH HEADER NODE

:→ To create a binary tree from Postorder and Inorder

Inorder → D B F E A G C L J H K

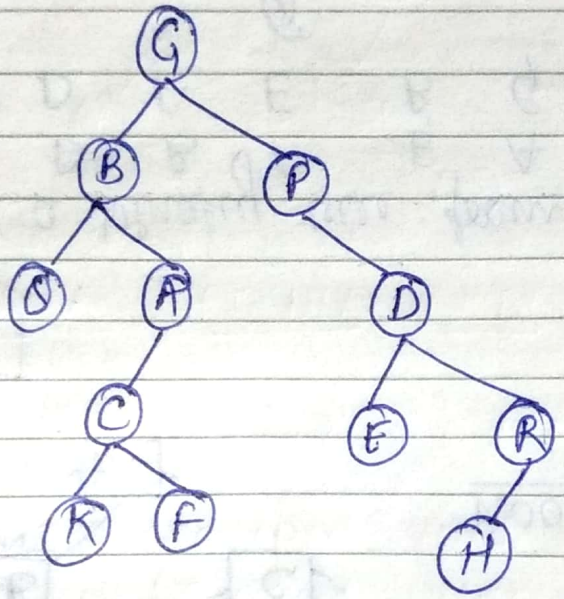
Postorder → D F E B G L J K H C A



✓ Trace the Postorder from the last and decide the left & right child acc to Inorder

:→ To create the binary tree from preorder and Inorder.

Preorder : G B Q A C K F P D E R H
Inorder : Q B K C F A G P E D H R



BINARY SEARCH TREES

A tree T is called a binary search tree if each node N of T has the foll. prop:

1) The value of N is > than every value in the left subtree of N and is less than every value in the right subtree of N.

(I) Searching and Inserting in BST.

~~Suppose an ITEM with the root node N of~~

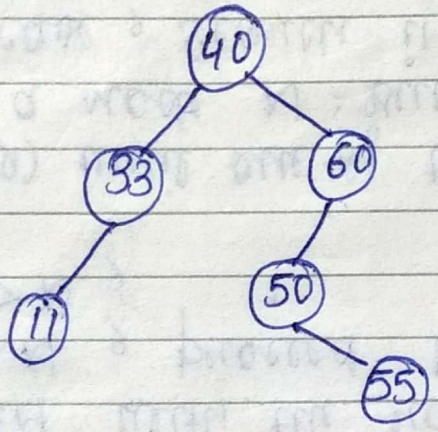
Suppose an ITEM of info given. The foll. algo finds the location of ITEM in the binary search tree T, or inserts ITEM as a new node in its appropriate place in tree.

a) Compare ITEM with the root node N of tree
   i) if ITEM < N, proceed to the left child of N
   ii) if ITEM > N, " " " right " " "

b) Repeat Step (a) until one of the foll. occurs:
   i) we meet a node N such that ITEM = N.
      In this case, search is successful.
   ii) we meet an empty subtree, which indicates that the search is unsuccessful and insert ITEM in place of the empty subtree.

CREATION OF BST :-

eg → Six numbers (40, 60, 50, 33, 55, 11) are to be inserted.



ALGO TO FIND A PARTICULAR NODE :

FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

ITEM is to be searched and to find the location LOC of ITEM in T and also the location PAR of the parent of ITEM. There are 3 special cases :-

i) LOC = NULL & PAR = NULL :⇒ tree is empty
ii) LOC ≠ NULL & PAR = NULL :⇒ ITEM is the root of T.
iii) LOC = NULL & PAR ≠ NULL :⇒ ITEM is not in T & can be added to T as a child of a node N with

loc. PAR.

1.) [Tree empty ?]
If ROOT = NULL then Set LOC = NULL & PAR = NULL & return

2.) [ITEM at root ?]
If ITEM = INFO[ROOT] then Set LOC = ROOT & PAR = NULL & return.

3.) [Initialize pointers PTR & SAVE.]
If ITEM < INFO[ROOT], then:
Set PTR = LEFT[ROOT] & SAVE := ROOT.
else:
Set PTR = RIGHT[ROOT] & SAVE := ROOT.
[end of if]

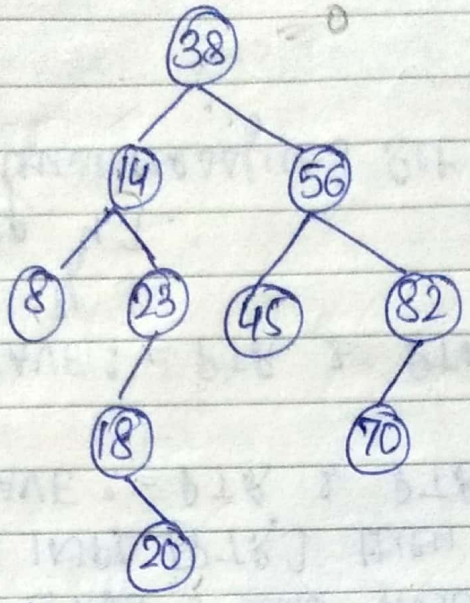
4.) Repeat steps 5 and 6 while PTR ≠ NULL.

5.) [ITEM found ?]
If ITEM = INFO[PTR] then Set LOC := PTR & PAR := SAVE, and Return.

6.) If ITEM < INFO[PTR] then
Set SAVE := PTR & PTR := LEFT[PTR].
else
Set SAVE := PTR & PTR := RIGHT[PTR].
[end of if].
[end of step 4].

7.) [Search unsuccessful] Set LOC := NULL & PAR := SAVE.

8.) Exit.



Let ITEM := 20

Step

3) if (20 < 38) then
PTR = 14 and SAVE := 0

(20 > 14)
Set SAVE := 14 and PTR = 23

(20 < 23)
Set SAVE := 23 and PTR = 18

(20 > 18)
Set SAVE := 18 and PTR = 20

(20 = 20)
Set
LOC = 20 and PAR = 18

To insert an item in BST :- (Algo)

INSBST (INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)

1.) Call FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

2.) If LOC ≠ NULL then Exit.

3.) [Copy ITEM into new node in AVAIL list]

a) If AVAIL = NULL then write OVERFLOW & EXIT.

b) Set NEW = AVAIL, AVAIL = ~~LEFT~~ RIGHT [AVAIL] & INFO [NEW] = ITEM

c) Set LOC := NEW, LEFT[NEW] = NULL & RIGHT [NEW] := NULL.

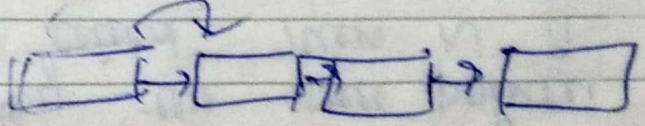
4.) [Add]

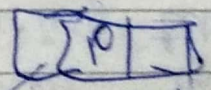
If PAR = NULL then
Set ROOT := NEW.
else if ITEM < INFO [PAR] then
Set LEFT [PAR] := NEW
else
Set RIGHT [PAR] := NEW
[end of if]

5.) Exit

~~29~~ Set ~~ITEM = 13~~.

3) ~~NEW = 13~~, ~~AVAIL =~~





14, 10, 17, 12, 10, 11, 20, 12, 18, 25, 20, 8, 22, 11, 23

51, 71, 16, 81, 9, 4, 12, 18, 9, 51

~~8, 14, 18, 38, 45, 5~~

7 2 9 0 5 6 8 1

Deleting in a Binary Search tree

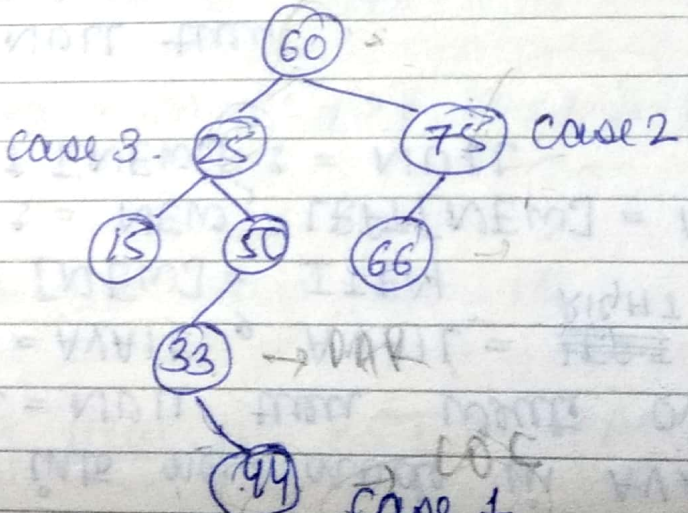
There are three cases:

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

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

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

Let



ALGO:-

DEL (INFO, LEFT, RIGHT, ROOT, ~~AVAIL~~, ITEM)
A binary search tree T is in memory and an ITEM of info is given. This algo deletes ITEM from the tree.

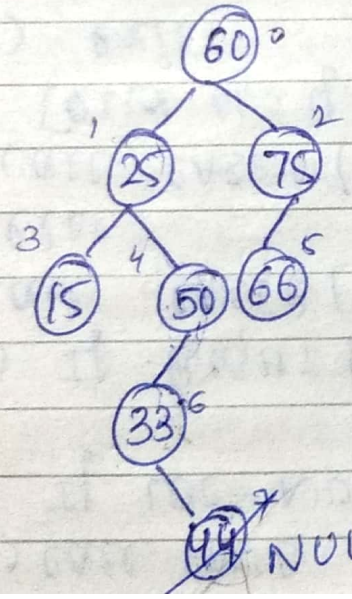
1.) Call FIND (INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
2.) If LOC = NULL then write: ITEM is not in tree and Exit.
3.) If RIGHT [LOC] ≠ Null & LEFT [LOC] ≠ NULL then call CASEB (INFO, LEFT, RIGHT, ROOT, LOC, PAR)
else:
call CASEA (INFO, LEFT, RIGHT, ROOT, LOC, PAR).
[end of if]
4.) exit.

CASEA (INFO, LEFT, RIGHT, ROOT, LOC, PAR) // for case 1 and case 2

1.) If LEFT [LOC] = NULL and RIGHT [LOC] = NULL then
set CHILD = NULL.
else if LEFT [LOC] ≠ NULL then
Set CHILD = LEFT [LOC]
else
Set CHILD = RIGHT [LOC]
[end of if]

2) If PAR ≠ NULL then
   If LOC = LEFT[PAR] then
      Set LEFT[PAR] = CHILD
   else
      Set RIGHT[PAR] = CHILD
   [end of if]
else
   Set ROOT = CHILD
[end of if]

3.) Return.



✓ ITEM to be deleted is 44
LOC = 7
PAR = 6
Left[LOC] = Right[LOC] = Null
Set CHILD = NULL.
✓ PAR ≠ NULL
and if (LOC = RIGHT[PAR]) True
Set RIGHT[PAR] := CHILD.
= NULL

CASEB (INFO, LEFT, RIGHT, ROOT, LOC, PAR)

This procedure will delete Node N at LOC. LOC where N has two children. The pointer PAR gives the location of the parent of N, or else PAR=NULL indicates that N is the root node. The pointer SUC gives the location of the inorder successor of N, and PARSUC gives the location of the parent of the inorder successor.

1) a) Set PTR := RIGHT[LOC] and SAVE := LOC
   b) Repeat while LEFT[PTR] ≠ NULL
      Set SAVE := PTR and PTR = LEFT[PTR]
   c) Set SUC := PTR and ~~PTR = LEFT[PTR].~~
      PARSUC = SAVE.

2) Call CASEA(INFO, LEFT, RIGHT, ROOT, SUC, PARSUC).

3) a) If PAR ≠ NULL, then:
      If LOC = LEFT[PAR], then
         Set LEFT[PAR] := SUC
      else
         Set RIGHT[PAR] := SUC.

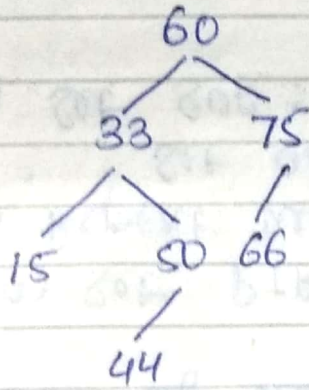
else:

Set ROOT := SUC

b) Set LEFT[SUC] := LEFT[LOC] and
RIGHT[SUC] := RIGHT[LOC]

4.) Return.

eg→ Consider a tree



Let "33" is to be deleted.

Inorder seq:

15 [33] 44 50 60 66 75

LOC SUC PARSUC

AVL Tree

The first balanced binary search tree was AVL tree (Adelson Velskii and Landis). These trees are binary search trees in which the height of two siblings are not permitted to differ by more than one

i.e.,

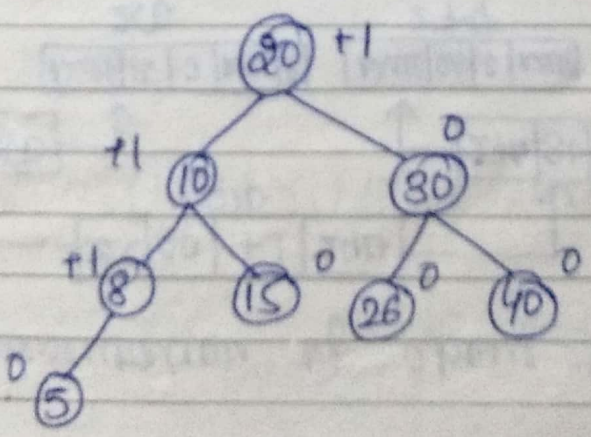
(Height of left subtree − height of right subtree) ≤ 1

$$BF = h(l) - h(r)$$

For an AVL tree, the value of the balance factor of any node is −1, 0, or 1. If it is other than these three values then the tree is not balanced and it is not an AVL tree and we have to rotate the tree to make it balanced tree.

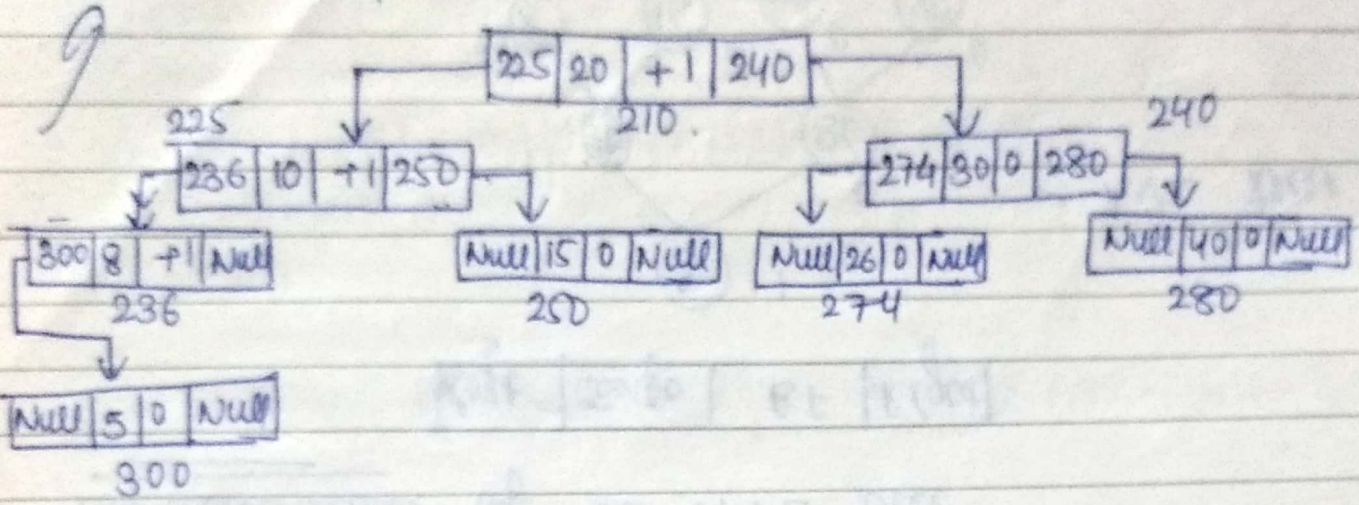
Representation of an AVL tree

Left	Info	BF	Right



✓ AVL tree

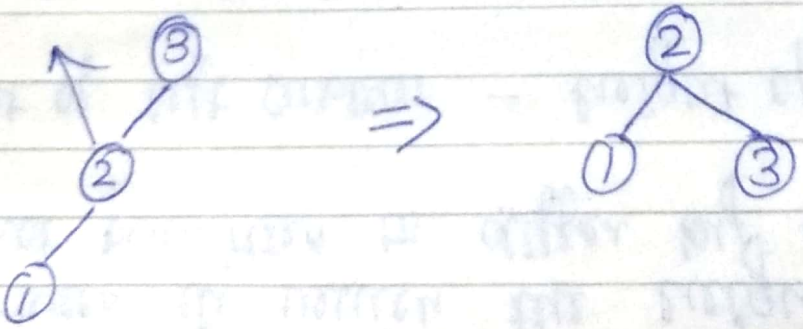
The linked representation of above tree is :-



Insertion in an AVL tree is same as the insertion in a Binary search tree but it affects the balanced factor. So, we need to rotate to maintain the BF.

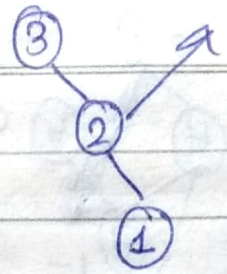
4 types of Rotation

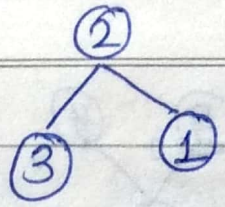
:- 1)



LL Rotation

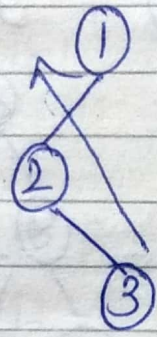
2:-)

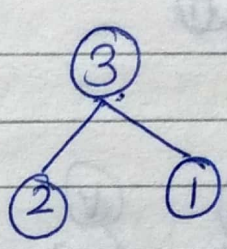




RR Rotation

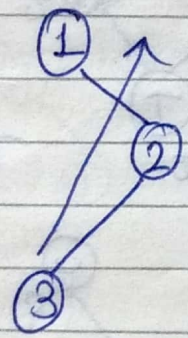
3:-)

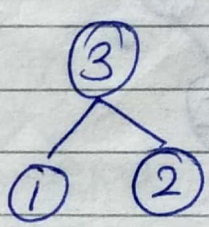




LR Rotation

4:-)





RL Rotation

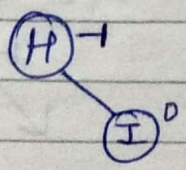
Q → Create an AVL search tree from given set of values :-

H, I, J, B, A, E, C, F, D, G, K, L.

A :- i) Insert H

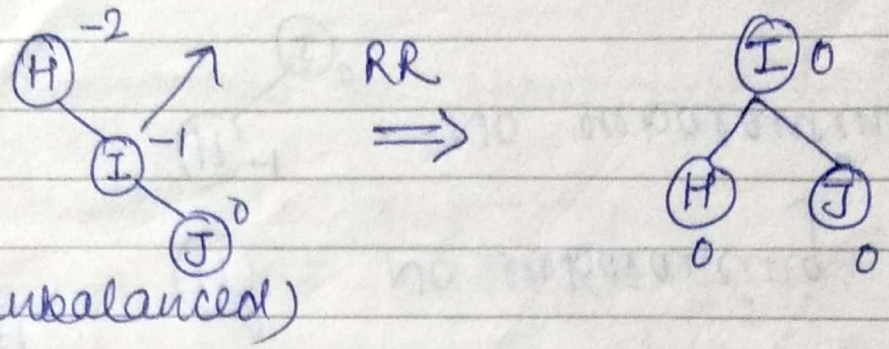
NO rebalancing

ii) Insert I



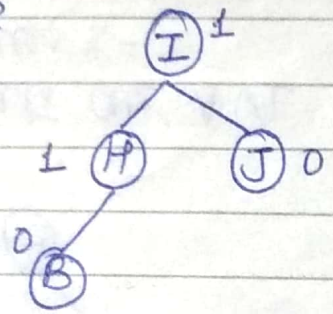
NO rebalancing

iii) Insert J



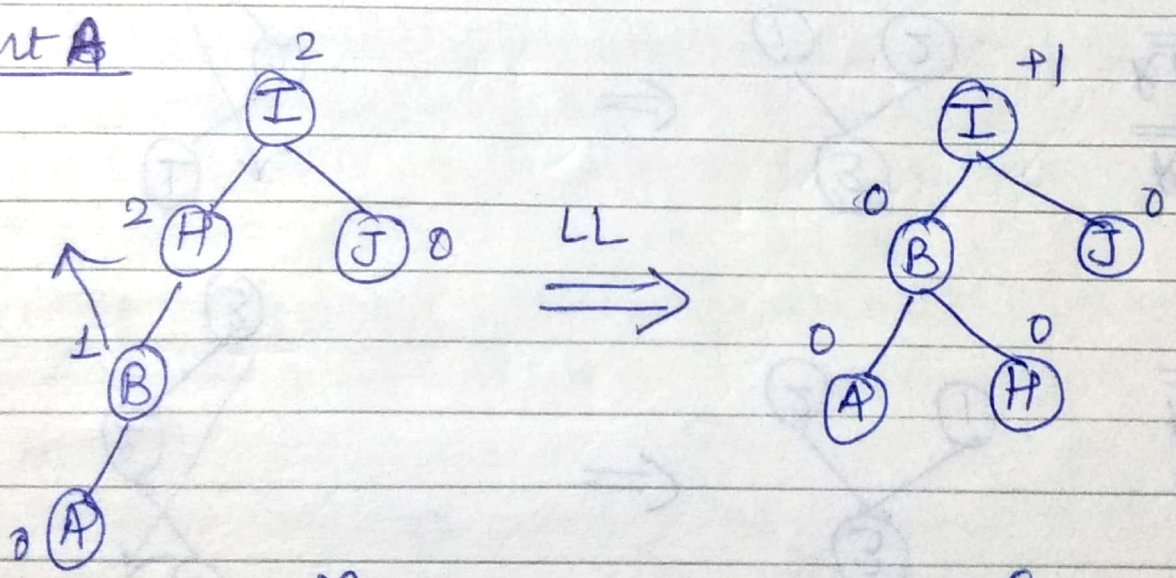
(unbalanced)

iv) Insert B

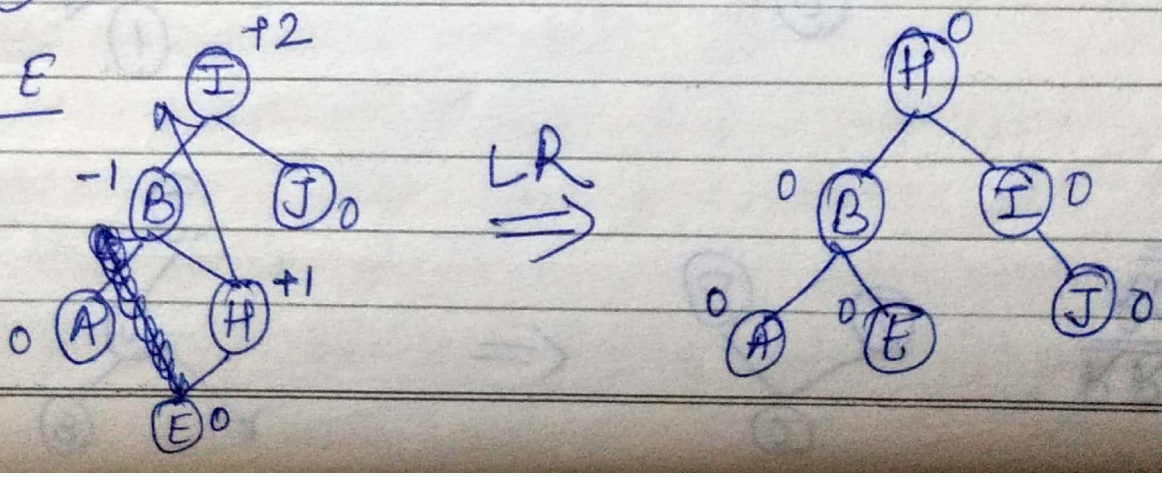


No rebalancing

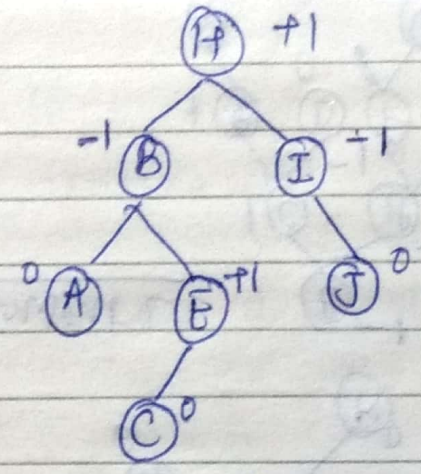
v) Insert A



vi) Insert E

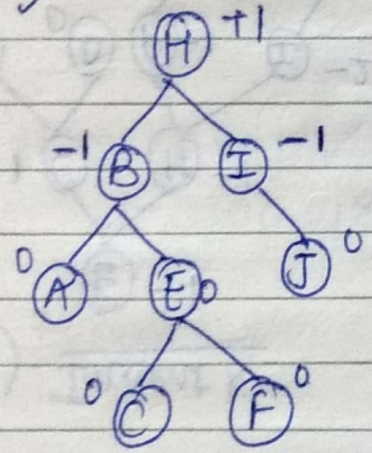


iii) Insert C



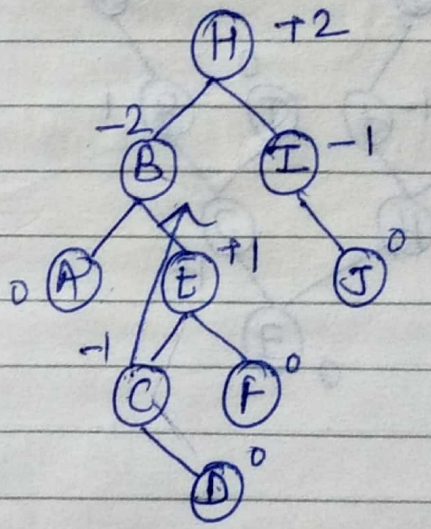
NO rebalancing

iiii) Insert F

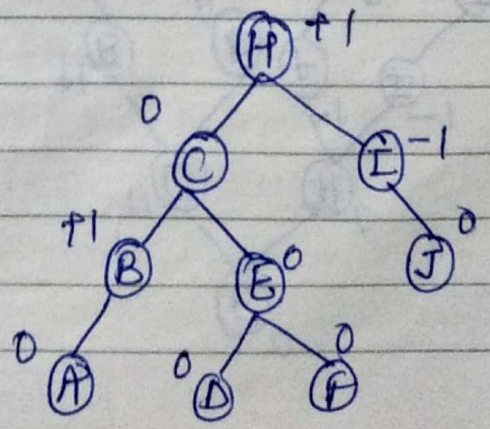


No rebalancing

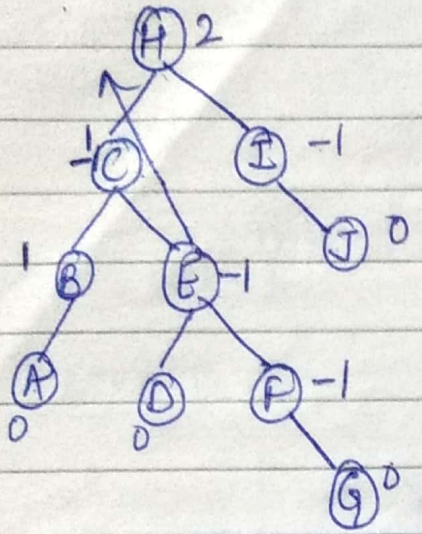
ix) Insert D



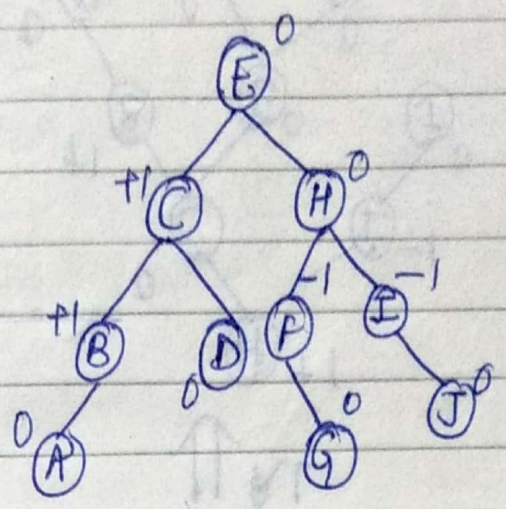
RL



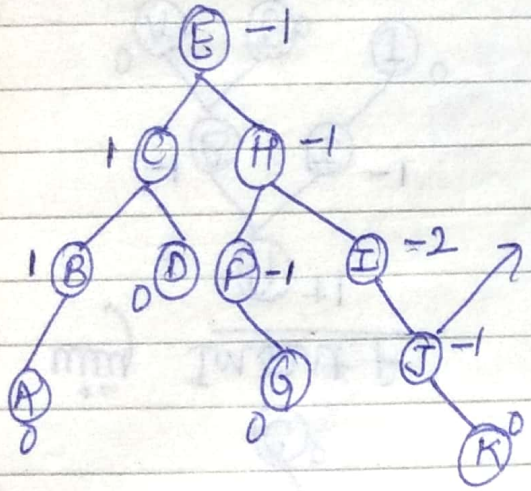
x) Insert G



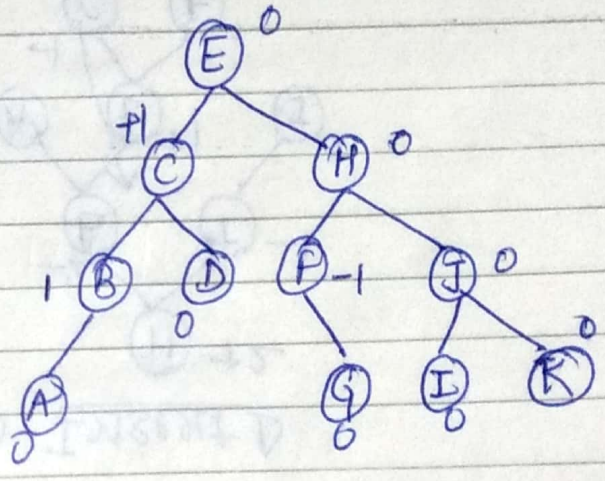
LR ⟹



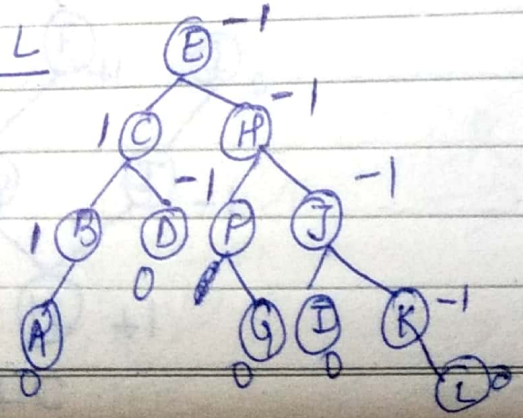
xi) Insert K



RR ⟹



(xii) Insert L



No rebalancing

Final AVL tree

Complexity of AVL tree = O(log N)

B-TREE

- It stands for Balanced-tree

eg→ Create a B-tree of order 5.

$m = 5$.

maximum no. of keys = $m - 1 = 4$

min " " " = $m/2 = 5/2 = 2$.

10, 20, 50, 60, 40, 80, 100, 70, 130, 90, 30, 120, 140, 25, 35, 160, 180.

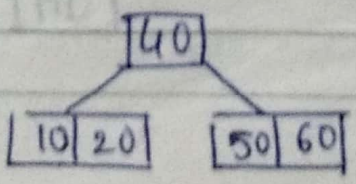
1) Insert 10 → [10]

2) Insert 20 → [10 | 20]

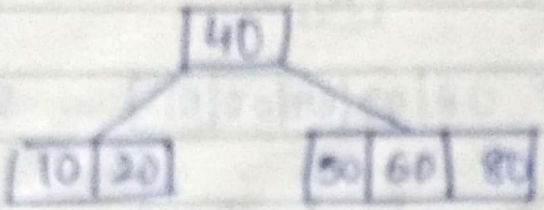
3) Insert 50 → [10 | 20 | 50]

4) Insert 60 → [10 | 20 | 50 | 60]

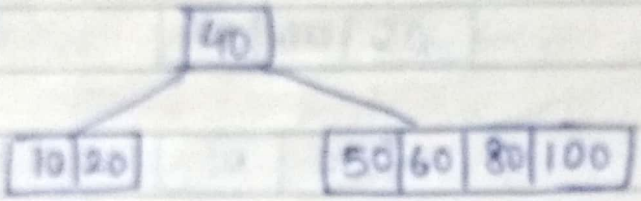
5) Insert 40 → [10 | 20 | 40 | 50 | 60] Break from the mid



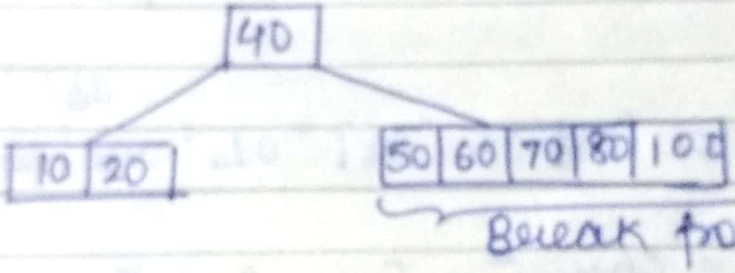
6) Insert 80

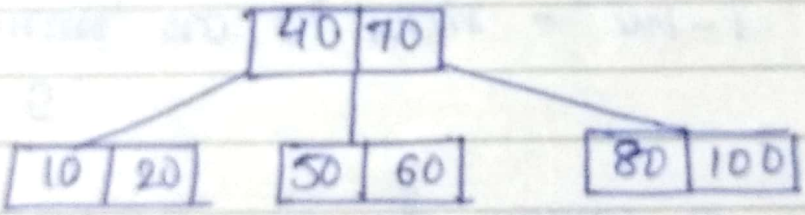


7) Insert 100

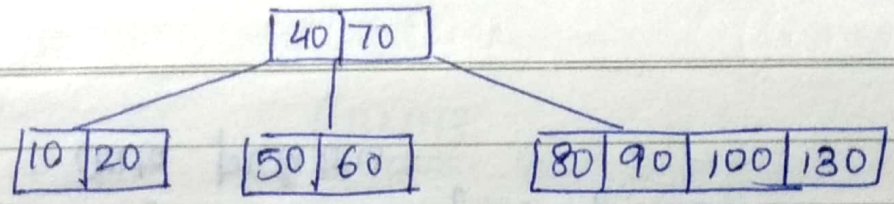


8) Insert 70

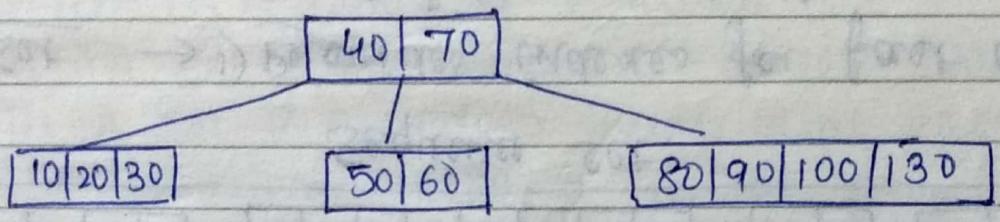




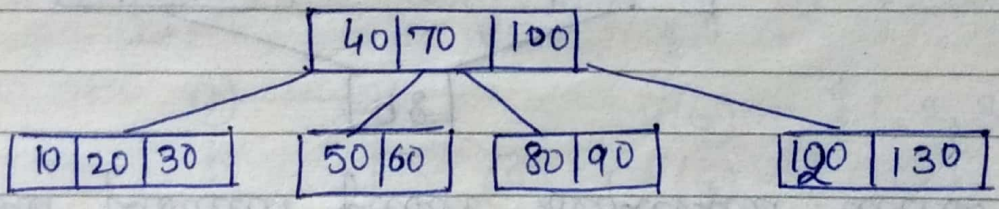
Insert 90 and 130



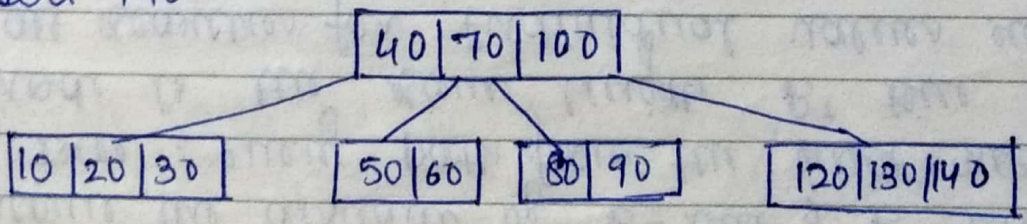
10) Insert 30



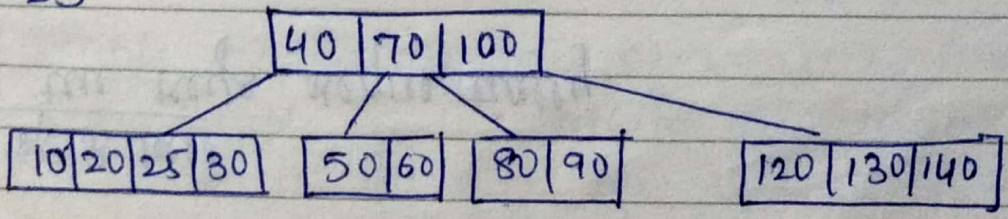
11) Insert 120



12) Insert 140



13) Insert 25

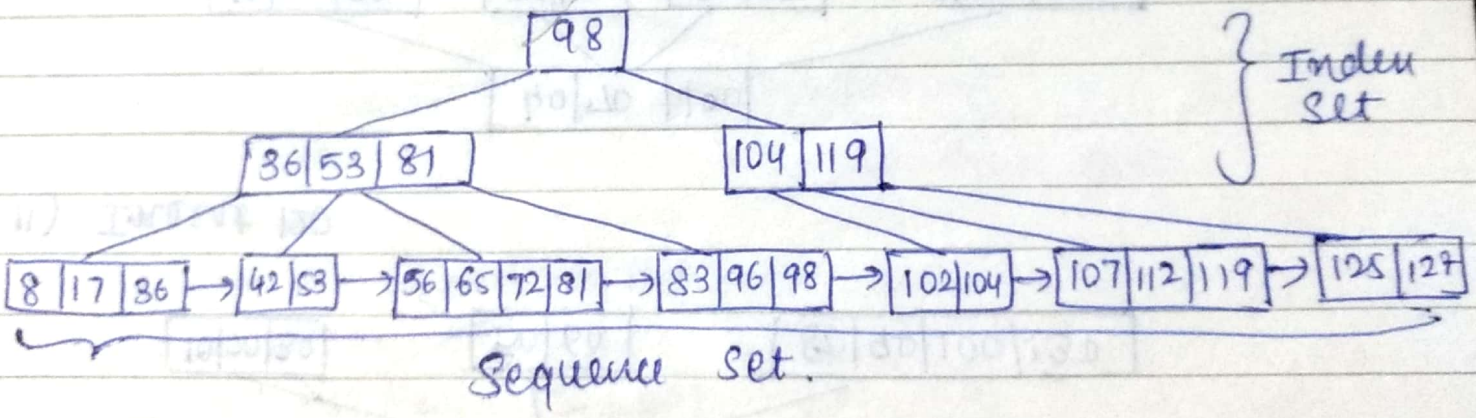


and so on.

Application of B Tree
- Organisation of a huge collection of records into a file structure.

Drawback of B tree
- Traverse the keys sequentially.

To overcome the disadv. of B-tree, B^+ tree is used. In B^+ tree, every path from the root node to a leaf node is the same length. B^+ tree means that all searches for individual values require the same no. of nodes to be read from the disk.
- B^+ tree provides faster sequential access of data



Index Set → 1) Provides indexes for fast access of data
2) Consists of internal nodes that store only key

Sequence Set → 1) Consists of leaf nodes that contain data pointers.