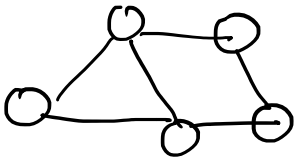
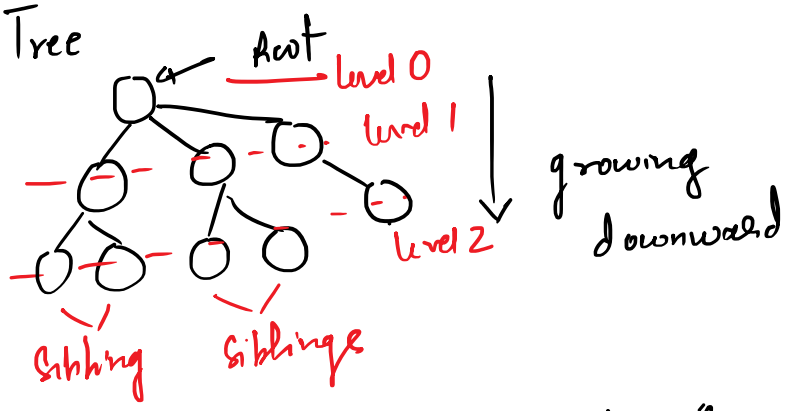
Trees →

Graph



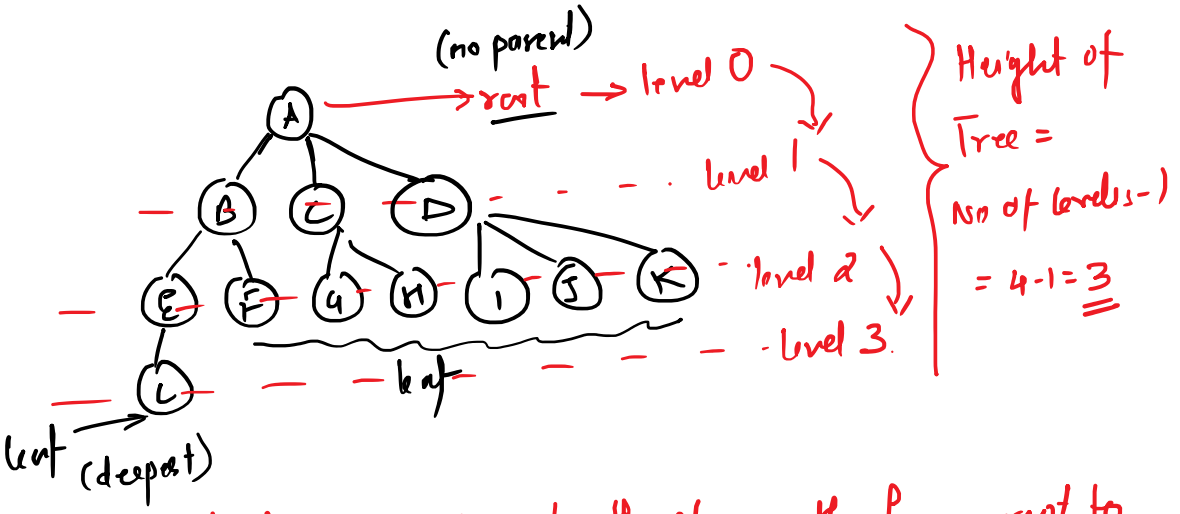
All nodes in same plane. (level)

Tree



A Tree is arrangement of nodes in hierarchical manner with the first node is known as root node from which the tree grows downward.

Consider →



Height of Tree := Height of Root Node = length of path from root to the deepest leaf

Leaf Node :- Node with no child.

Root Node :- Node with no parent.

Tree is n-ary Structure → n = max no. of child that every node in a Tree can have.

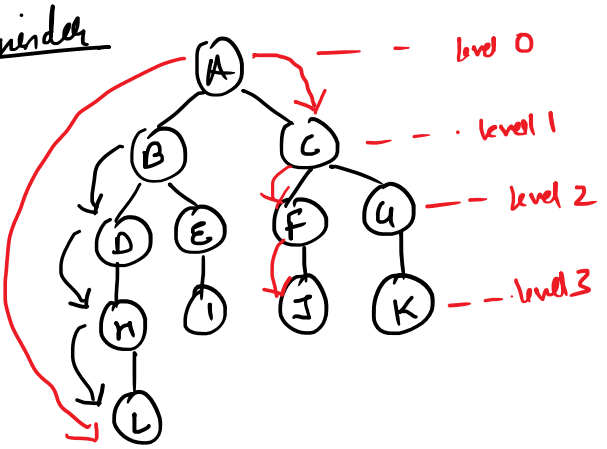
Tree is n-ary Structure → n = max no. of child that a node in a Tree can have.

if n=2 then it is known as Binary Tree → Every node in tree can have max 2 child.

Binary Tree :→ It is tree in which no node can have more than 2 child.

Terminologies →

Consider



① Path → A path from node A_1 to A_K is collection of nodes $A_1, A_2 \dots A_K$ such that every A_i is parent of A_{i+1}

Path from A → J ⇒ A → C → F → J

Path from B → L ⇒ B → D → H → L

Node is parent of Immediate next node.

② Degree of a Node → It is no of child that the node has.

Degree (B) = 2, Degree (F) = 1, Degree (L) = 0.

③ Degree of a Tree :→ It is degree of node having highest degree.

Degree (Tree) = 2.

(4) Depth of a Node → gt is the length of the path from root node to that node.
(distance from root)

Ex Depth of node (I) = Path from Root to I ⇒ A → B → E → I ⇒ 3

(5) Depth of Tree ⇒ gt is depth of the deepest leaf node.

Depth of Tree above :- depth of deepest leaf node ie h = 4

(6) Ancestors of a node → All the nodes that are on path from root node to that node are ancestors of that node.
(Predecessors).

Ex ancestors of (L) = A, B, D, H, L

(7) Descendents of a node: All the nodes that have path from given node are descendents of that node.
(Successors)

descendents of (B) = D, E, H, I, L

(8) Height of node → gt is no of edges on longest downward path from that node to the deepest leaf.
(distance from node to its deepest leaf).

Height of (B) = length of path from B to deepest leaf ie L = 3

(9) Height of tree = gt is height of root node.

In above tree

Height of root node = Height of (A) = length of path from A to deepest node ie L. = 4.

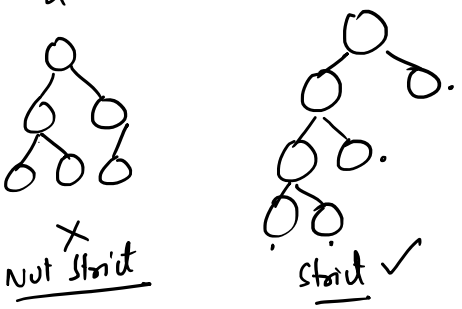
Note

Depth of a Node ⇒ Distance from root to that node.

Height of a Node ⇒ Distance from that node to its deepest leaf child

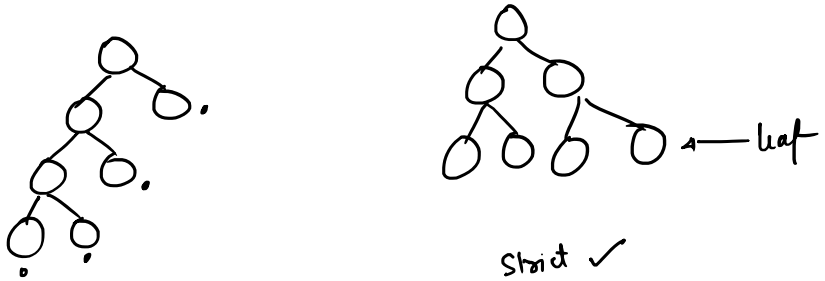
Types of Binary Tree →

① Strict Binary Tree ⇒ It is B.T where every node can have either a child or no child.



Not Strict ✗ Strict ✓

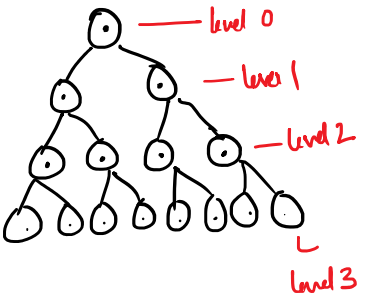
② Complete Binary Tree :⇒ It is Strict B.T where all the leaf nodes are at same level.



Strict ✓
Complete ✗
All leaf nodes are not at same level.

Strict ✓
Complete ✓
All the leaf nodes are at same level.

Kuch Baatein Wrt Complete Binary Tree !!! (C.B.T)



① No of nodes at any level $l \Rightarrow 2^l$

② No of leaf nodes $= 2^d$ (d = depth of tree)
$= 2^3 = 8$

③ No of Non leaf Nodes $= 2^d - 1$ (d = depth of tree)
$= 2^3 - 1 = 7$

④ Total No of nodes in C.B.T = No of leaf nodes + No of Non leaf Nodes
$= 2^d + 2^d - 1 \Rightarrow 2^{d+1} - 1$.

(4) Total No of nodes = No of ...
in C.B.T = $2^d + 2^d - 1 \Rightarrow 2^{d+1} - 1$.

Analysis → Let 'n' be total No of nodes in C.B.T and let 'd' be depth.

We know $n = 2^{d+1} - 1 \Rightarrow (n+1) = 2^{d+1}$

Take log on B.S $\log(n+1) = (d+1)\log 2$

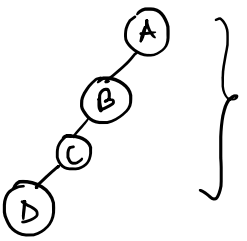
$$d+1 = \frac{\log(n+1)}{\log 2} = \log_2 n+1$$

$$\therefore d+1 = \log_2 n+1$$

$$\therefore \boxed{d = \log_2 n+1 - 1}$$

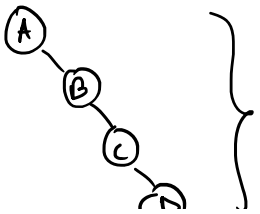
(5) Skewed Binary Tree → It is Binary Tree where every node can have Single and Same type of SubTree

(i) Left Skewed B.T → Here Every node will have only Left Subtree (child)



Left Skewed B.T.

(ii) Right Skewed B.T → Here Every node will have only right Subtree (child).

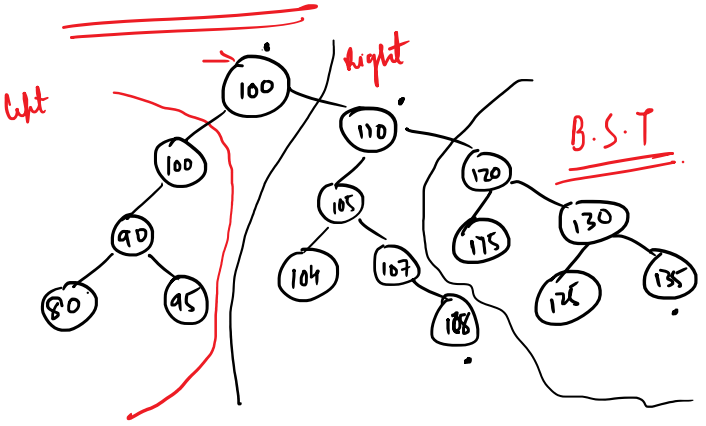


Right Skewed B.T.

(7) Binary Search Tree (BST) →

It is a Binary Tree in which all left descendents of the node are less than equal to that node and all right descendents of the node are greater than to that node.

[At Every node → left child ≤ node
right child > node]



* This B.S.T arrangements simplifies the search operation.

Tree Traversal →

(Vimp)

Traversal ⇒ Visiting Every node exactly once.

Tree Traversal ⇒ Visiting Every node of Tree Exactly once.

Binary Tree Traversal →

A Binary Tree can be traversed in any of the following fashion:→

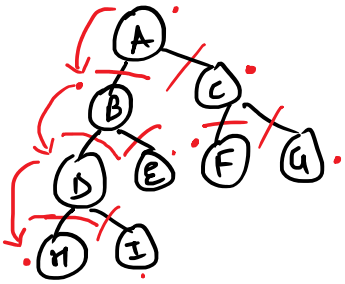
① In Order Traversal ⇒ At Every Node (L - V - R)

- L → Left child of node
- V → value of node
- R → Right child of Node

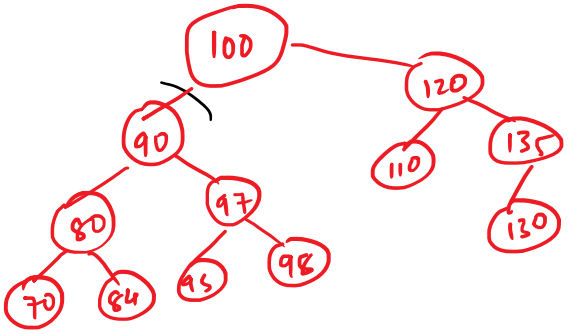
Here at Every node →

1. Visit left child ✓
2. Print Node Value.
3. Visit Right child.

Consider →



Inorder Traversal → H D I B E A F C G



⇒ Inorder traversal →

70 80 84 90 95 97 98 | 100 | 110 120 130 135

(70 80 84 90 95 97 98 : Left child; 100 : ↑; 110 120 130 135 : Right child)

BST

Left child ↑ Right child

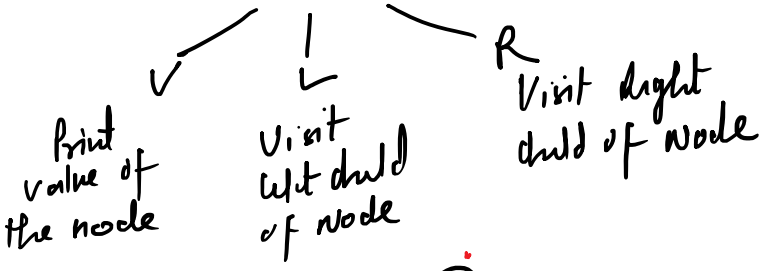
Sorted Order

* In Inorder Traversal

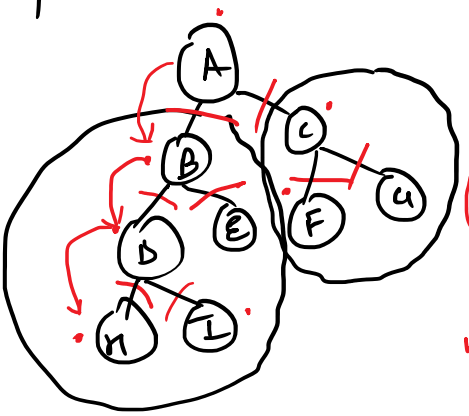
All the left child of nodes will appear to the left of node &
" " Right " " " " " " " right " "

* Inorder Traversal of BST will print the nodes in sorted (Ascending Order).

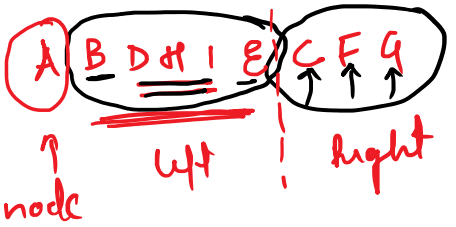
Pre Order Traversal → At Every Node



In Pre Order Traversal
At Every Node
1. Print value of node ✓
2. Visit its left child
3. Visit its right child



Preorder Traversal

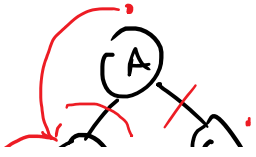


* Here the node will be displayed first followed by its left & right child.

Post Order Traversal → At Every Node

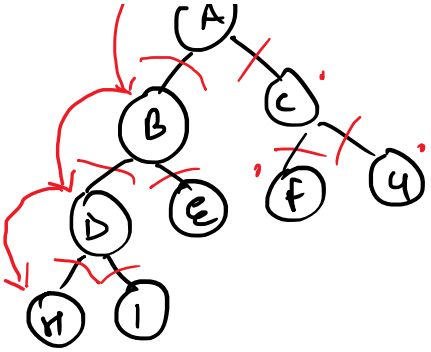
In Post Order Traversal At Every node Perform →

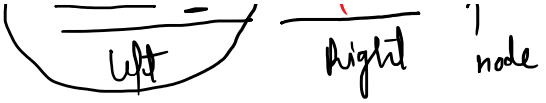
1. Visit its left child
2. Visit its Right child ✓
3. Print value of node.



Post Order Traversal → H I D E B F G C A

Left | Right | node





Here All the left subtree nodes will be displayed then Right subtree node and then the node is displayed.

Note: For Tree →

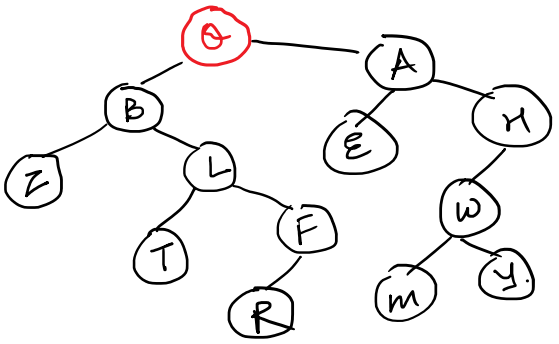
Inorder	Left	Root	Right
Preorder	Root	Left	Right
PostOrder	Left	Right	Root

Given Inorder Traversal: Z B T L R F Q E A M W Y H

Postorder Traversal: Z T R F L B E M Y W H A Q

Left | Root | Right | Root

Construct B.T =



Desired Binary Tree

Left SubTree of Q =

Inorder => Z B T L R F

Left | Root | Right

Postorder => Z T R F L B

Left | Right | Root

Right SubTree of Q =

Inorder= E A M W Y H

Left | Root | Right

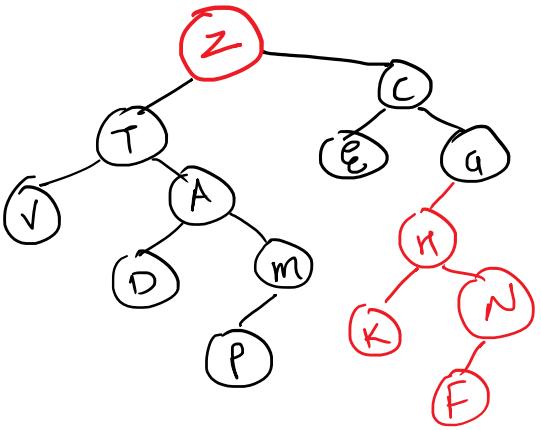
Postorder E M Y W H A

Left | Right | root

Consider

Inorder Traversal: V T D A P M Z E C K H F N G

Preorder Traversal: Z T V A D M P C E G H K N F



Left SubTree Construction.

Inorder: V T D A P M

Preorder: T V A D M P

Right SubTree Construction.

Inorder: E C K H F N G

Preorder: C E G H K N F

Application of Tree →

Huffmans Encoding :

Consider : A B A A C B B A C B A A A D C C B A A A A B C B A

Length = 25 characters

In C every character = 1 byte

∴ Total size of above txt = 25 × 1 = 25 byte

= 25 × 8 bits

= 200 bits

we want to reduce length of the Message. →

character	frequency.
A	11
B	8
C	5
D	1

Basic Working →

* Represent high frequency character with less number of bits
* Represent low frequency character with high number of bits.

Let Every character = 4 bits

Total size = 25 × 4 = 100 bits

Let A ⇒ 2 bits

B ⇒ 3 bits

C ⇒ 4 bit

D = 5 bit

∴ Total Size = 2 × 11 + 3 × 8 + 5 × 4 + 1 × 5

= 22 + 24 + 20 + 5

= 71 bits

Huffman Encoding

* Encoding is a process of converting symbols to code.
* The converted code must satisfy the following conditions →
  1. The code must be unambiguous i'e unique.
  2. The length of the code is inversely proportional to the frequency of the symbol

this can be done using Huffman's coding.

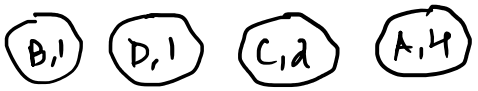
Consider :- Given Message :→ A C B A B A C A .

Generate Huffman Code :→

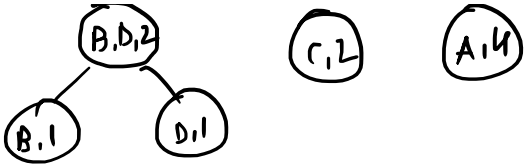
Step 1) Frequency Table →

Symbol	Frequency
A	4
B	1
C	2
D	1

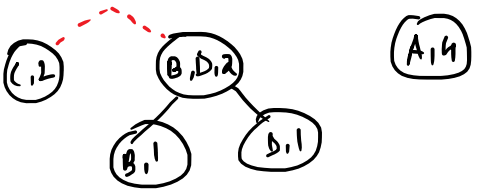
Step 2) Arrange Symbols in Asending order of frequency.



Step 3 Generate Huffman Code.

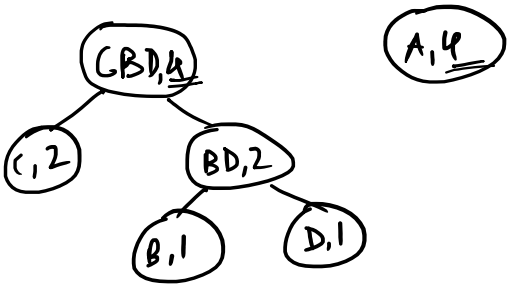


Rearrange =>

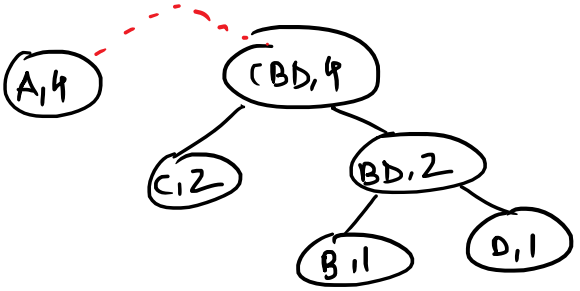


if weight of new created and existing is same then write existing first.

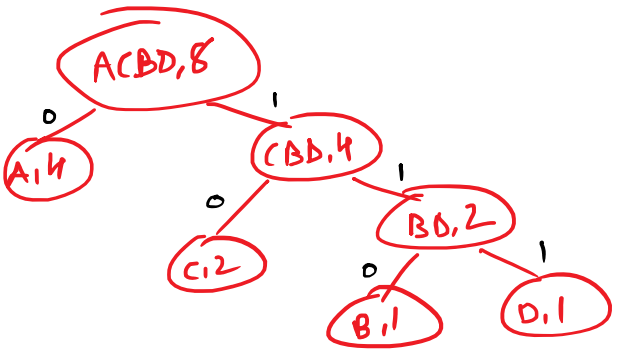
⇒



Rearrange =>



At Every Node
Assign left = 0
Right = 1



A = 0

B = 110

C = 10

D = 111

Ans Huffman Code for Each Symbol / Character.

If 4 bit Each then total Size = 8×4 = 32 bit

With Huffman Code = 4×1 + 3×1 + 2×2 + 1×3

$= 4+3+4+3 = 14$ bit

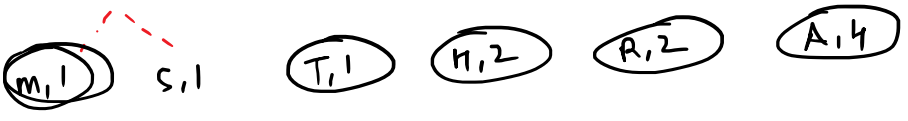
Q7

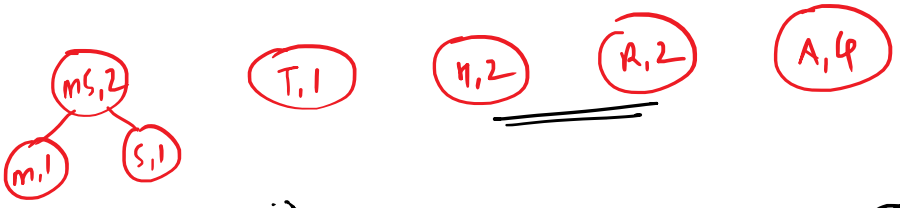
Generate Huffman Code for MAHARASHTRA

Solution => Step 1) Frequency table

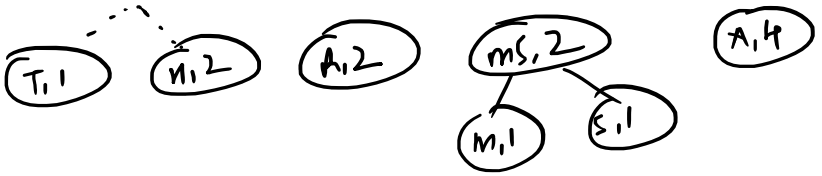
Symbol	Frequency
M	1
A	4
H	2
R	2
S	1
T	1

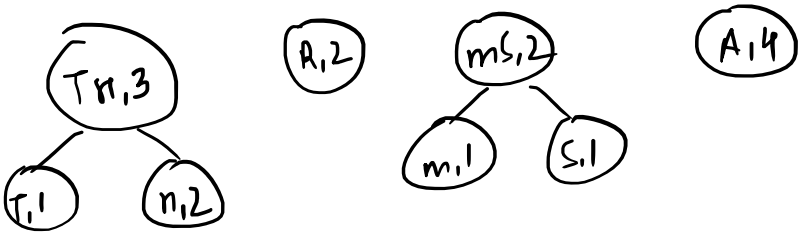
Step 2) Arrange Symbols in Ascending order of frequency.



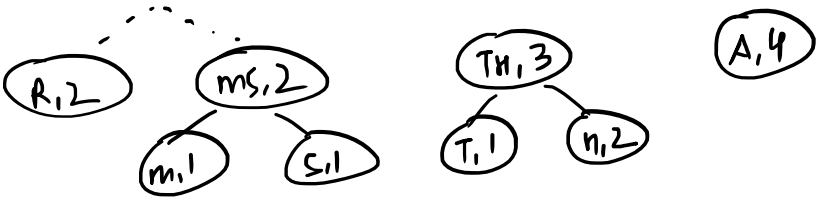


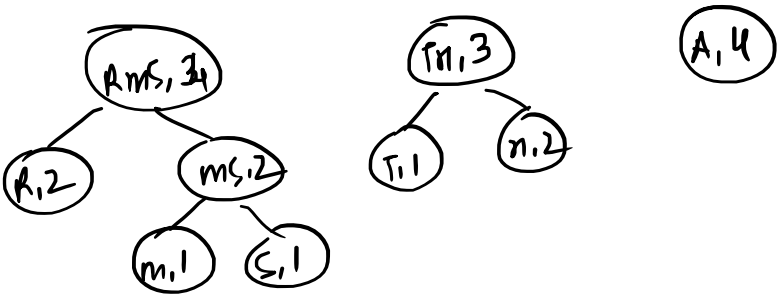
Rearrange =>



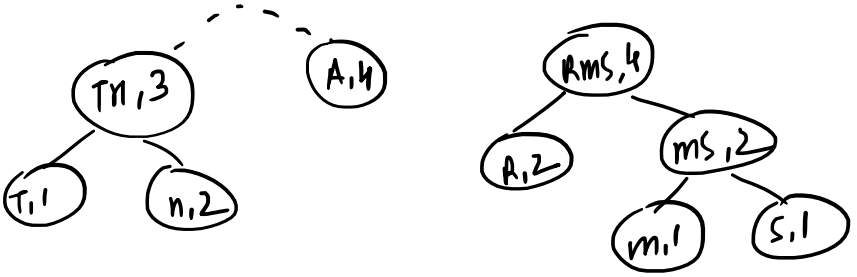


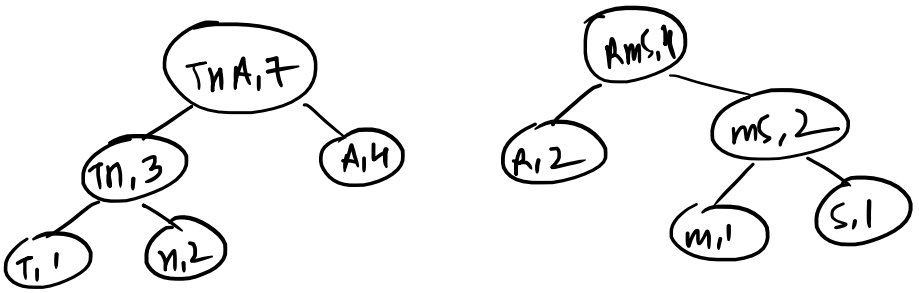
Rearrange =>



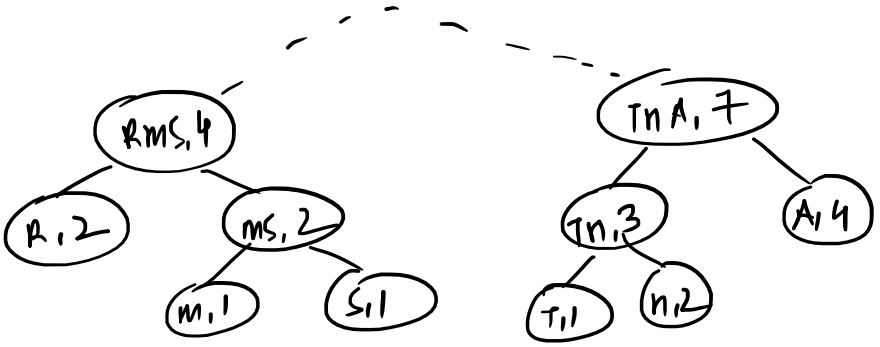


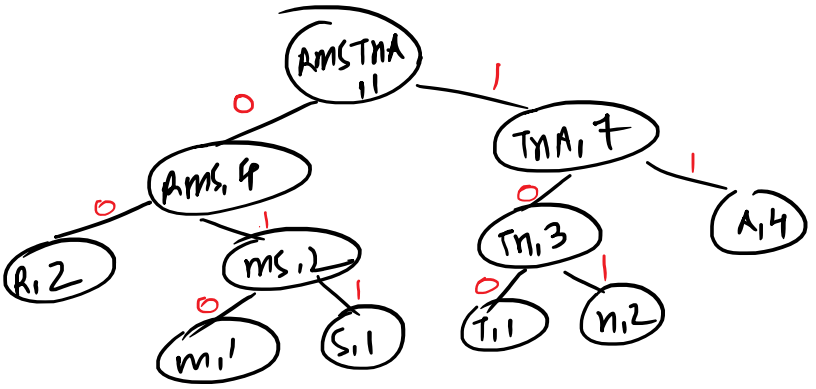
Rearrange





Rearrange





Huffman Code

M = 010
R = 00

T = 100
N = 101
A = 11

Huffman Code

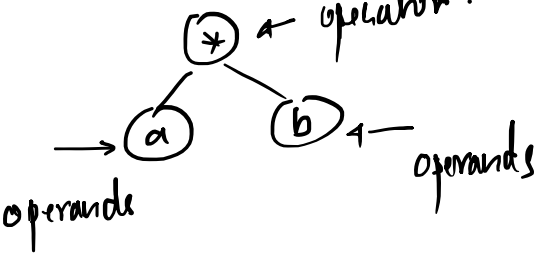
M = 010
R = 00
S = 011

H = 101
A = 11

Expression Tree →

* It is a tree constructed for representing an expression.
* Here the operands are child node of operators.

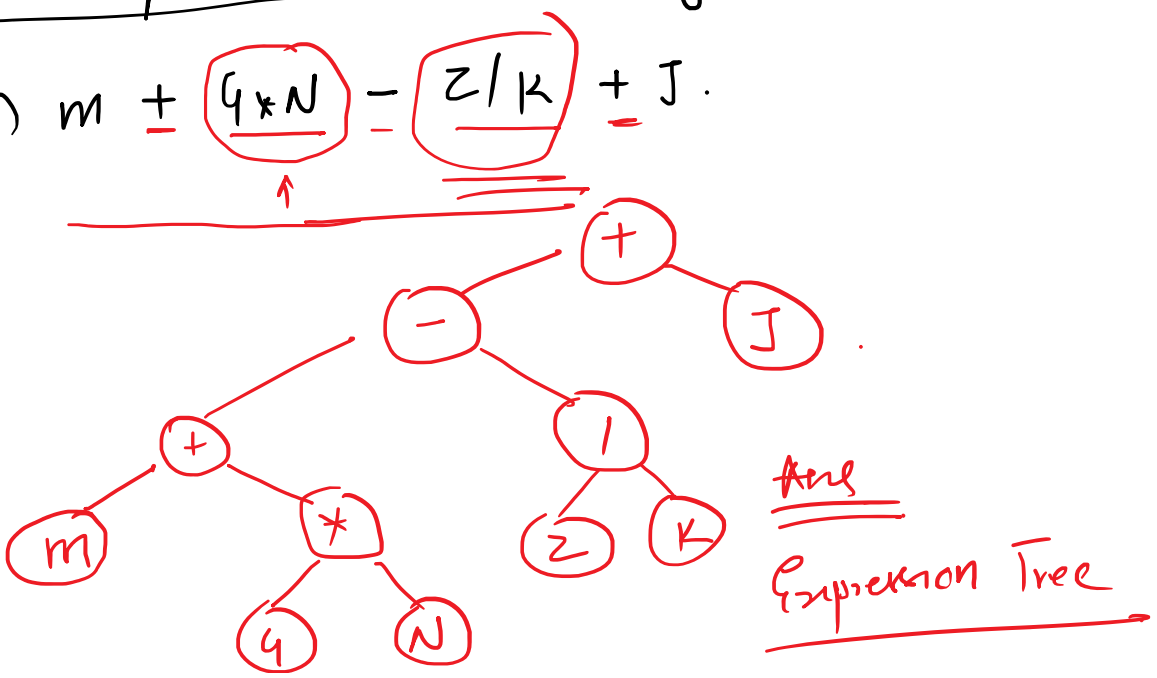
Ex a * b ⟶ Expression Tree



Traverse given Exp tree Inorder → a * b. (Infix Exp)
Preorder → * ab (Prefix Exp)
Postorder = a b * (Postfix Exp).

Q) Create Expression Tree for following Exp →.

(i) M + (Q * N) − (Z / K) + J.

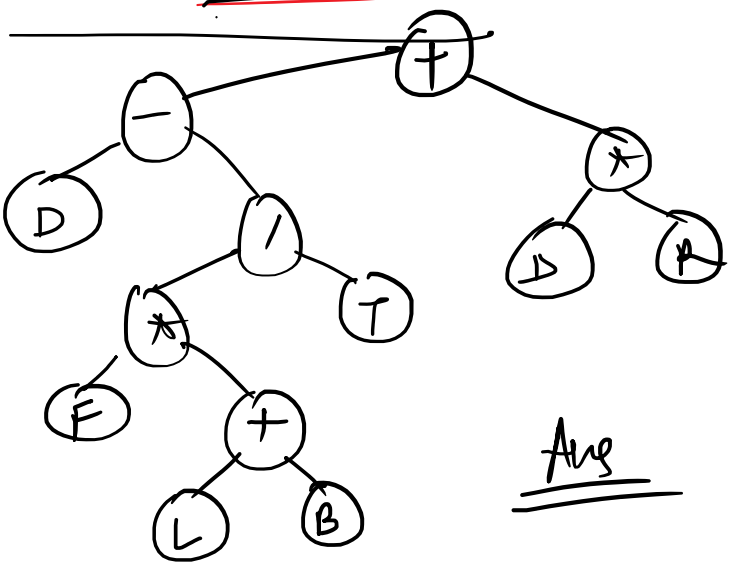


Ans
Expression Tree

Expression Tree

(4) (N)

(a) D − F * (L + B) / T + D * R



Ans

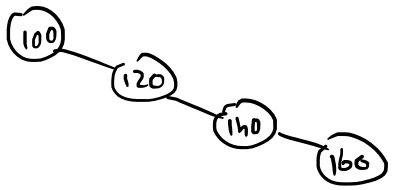
(Imp)

AVL Trees (Adelson, Velcki, Landis)

* It is BST
* It is Height Balanced Tree.

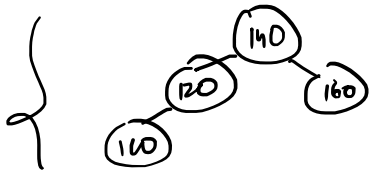
Why Needed?

Consider BST



To search 100 ⇒ 1 Comparison.
120 ⇒ 2 Comparison
140 ⇒ 3 Comparison
160 ⇒ 4 Comparison

But



⇒ To search 140 = 3 comp
120 = 2 comp
160 = 2 comp
140 ⇒ 1 compar

less the Height of Tree, less is the No of Comparison performed to Search an Element.

AVL Algo Ensures that the tree is Height Balanced.

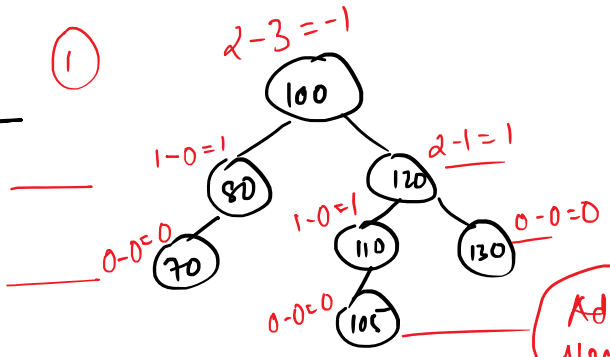
Balance of a Node → (How to decide, Node is balanced or not)

* gt is difference bet^n the Height of its (Node) Left Subtree & Right Subtree.

* In AVL, a node is said to be balanced if the difference is -1, 0, 1

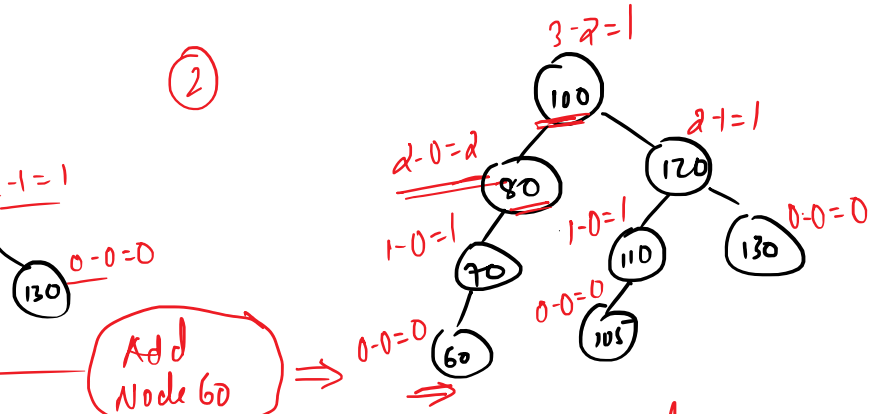
A Tree is said to be AVL Balanced, if Every node in tree is Balanced as per above conditions

Consider (1)



Balanced AVL

(2)

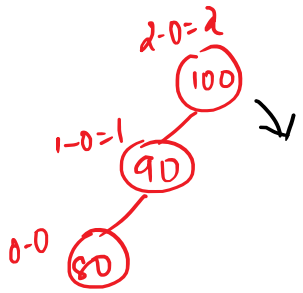


↑ Now Node 80 UnBalanced, so the tree is not Balanced AVL

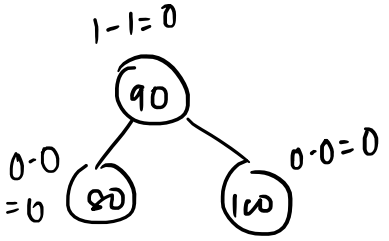
In a Balanced AVL, An Insertion may cause one/more nodes (ancestor) to become unbalanced, In such case appropriate rotations must be applied to the closest unbalanced ancestor node to the newly added node.

Types of Rotations to create balanced AVL Tree :-

① L L Case (Left Left Case).



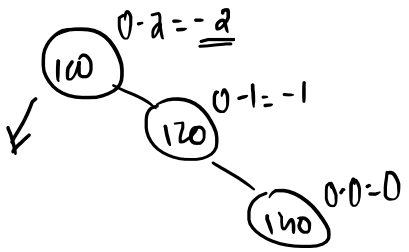
Rotate the unbalanced node to its right



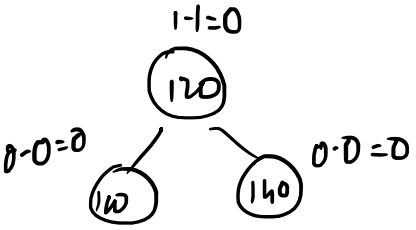
Balanced AVL

Node 100 is unbalanced due to LL Case

② Right Right (RR) Case →



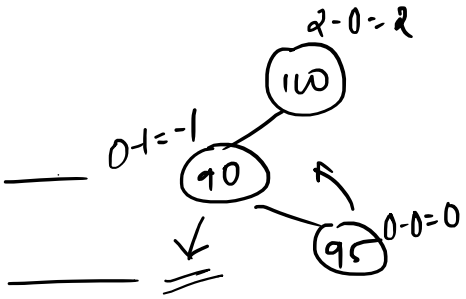
Rotate the unbalanced node to the left



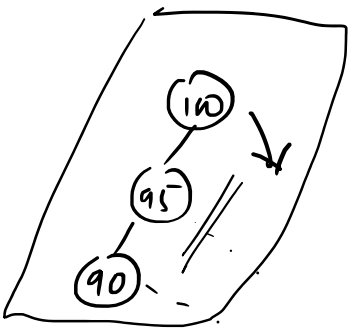
Balanced AVL.

Node 100 is unbalanced due to RR Case

③ Left - Right (L-R) Case



1. Rotate the child of unbalanced node to left



Now LL Case

Node 100 is unbalanced due to LR Case.

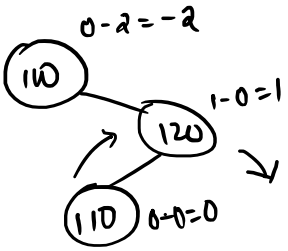
2. Rotate the unbalanced node

Node 100 is unbalanced due to L R Case.

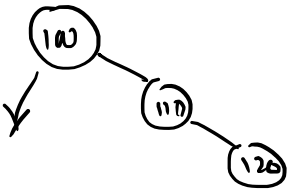
2. Rotate the unbalanced node to Right

Balanced AVL.

(4) Right left (R-L) Case →



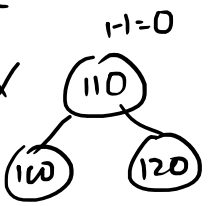
1. Rotate child of unbalanced node to Right



RR Case

Node 100 is unbalanced due to R-L Case

2. Rotate the unbalanced node to the left

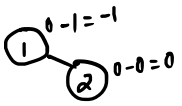


Balanced AVL.

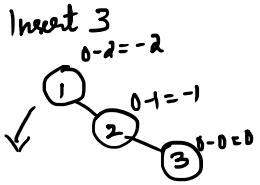
Construct AVL Tree Using 1, 2, 3, 4, 5, 6, 7

(1) Insert 1

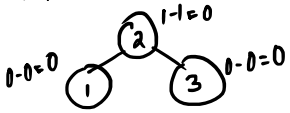
(2) Insert 2



(3) Insert 3

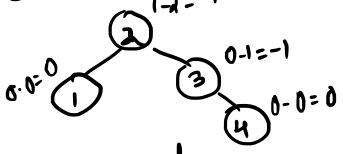


Node 1 is unbalanced
do RR Case
∴ Rotate 1 to the left



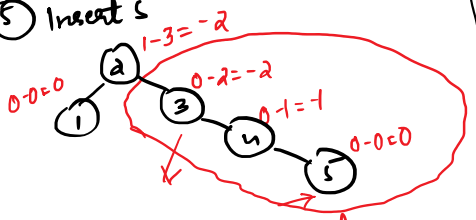
Balanced.

(4) Insert 4

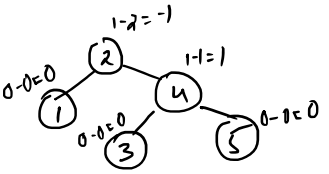


Balanced.

(5) Insert 5

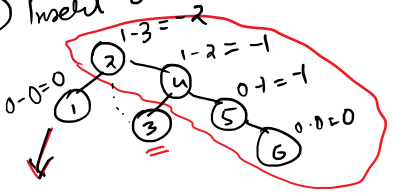


When node 5 is inserted we have two unbalanced ancestor of 5
node 2 and node 3
The nearest unbalanced ancestor of node 5 is node 3
So first make node 3 balance.
node 3 is unbalanced due to RR Case.
So Rotate Node 3 to the left

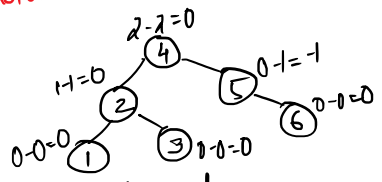


Balanced.

(6) Insert 6

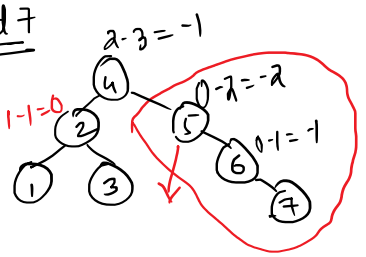


Node 2 is unbalanced due to RR Case.
Rotate Node 2 to its left

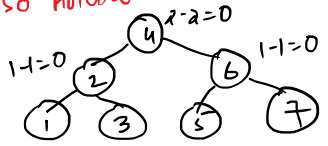


Balanced.

(7) Insert 7



Node 5 is unbalanced due to RR Case.
So Rotate 5 to its left



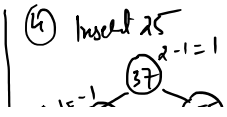
Balanced AVL Tree.

02) Construct Balanced AVL Tree Using

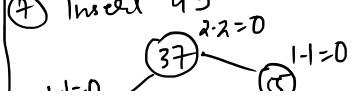
37 55 18 25 68 10 43 50 40 46 52 60.

(1) Insert 37

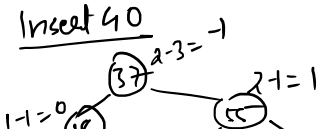
(4) Insert 25



(7) Insert 43

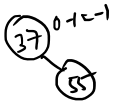


(9) Insert 40

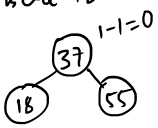


① Insert 37

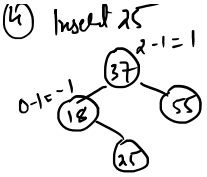
② Insert 55



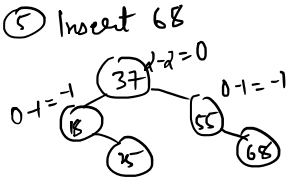
③ Insert 18



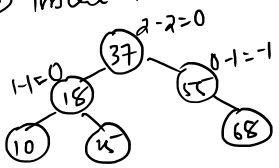
④ Insert 25



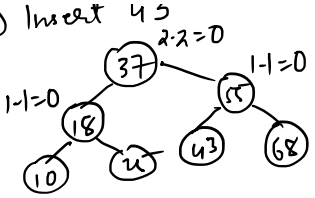
⑤ Insert 68



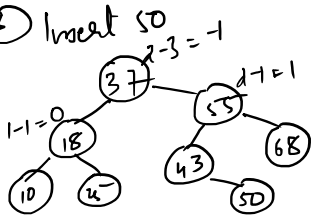
⑥ Insert 10



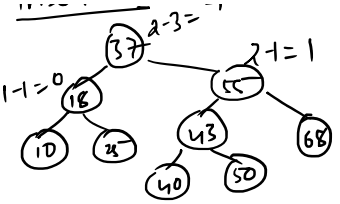
⑦ Insert 45



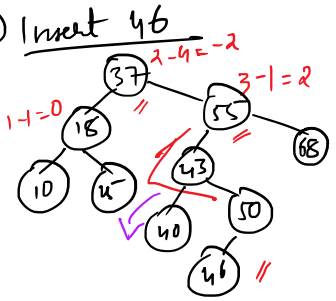
⑧ Insert 50



⑨

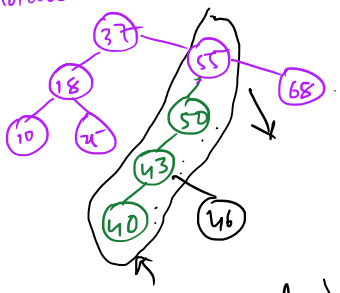


⑩ Insert 46



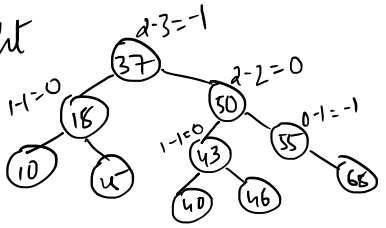
nearest unbalanced ancestor is 55 due to LR case.

Step 1 Rotate 43 to the left (Since 43 is child of unbalanced node 55)



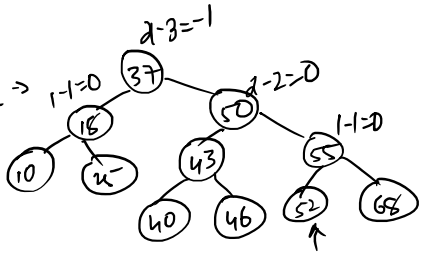
(Node 55 unbalanced to LL case)

Step 2 Rotate unbalanced node 55 to Right



Balanced

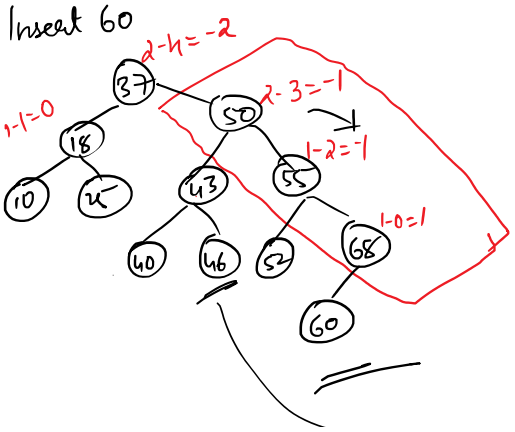
⑪ Insert 52 →



Balanced

(Prepared one)

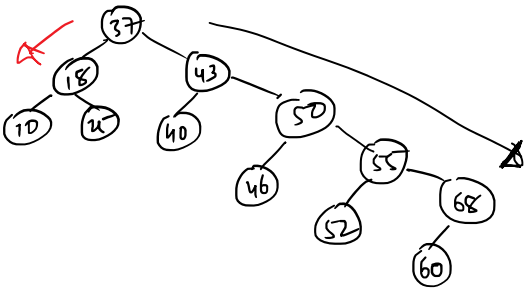
⑫ Insert 60



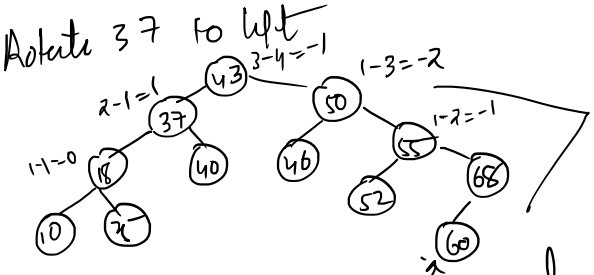
Consider

Case 1) Node 37 unbalanced due to RL Case.

Step 1) Rotate 50 to Right



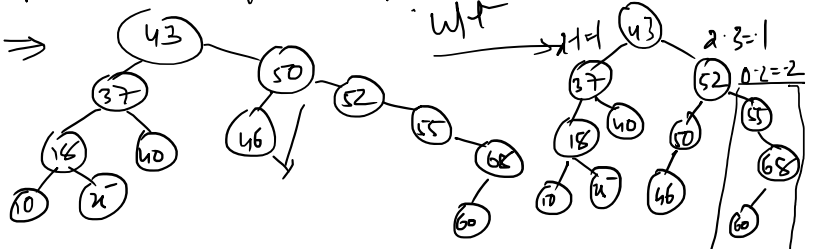
Rotate 37 to left



Node 50 is unbalanced due to RL case

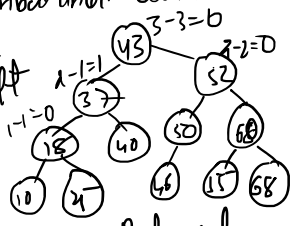
Rotate 55 to Right

Rotate 50 to left



55 is unbalanced due to RL

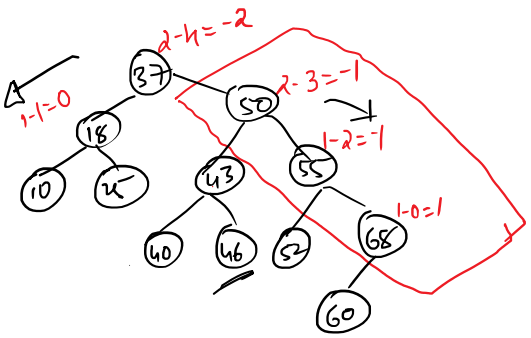
Rotate 68 to R
Then 55 to left

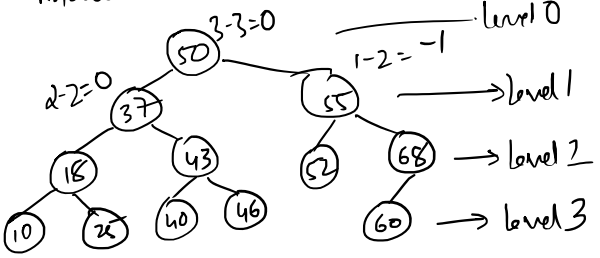


Balanced

Case 2) Consider 37 is unbalanced due to RR Case

Step 1 Rotate 37 to the left





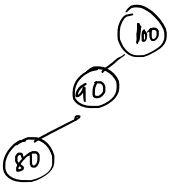
Balanced AVL

Height of Tree = 3

Max No of comp to search any node = 4

Note ⇒ (1) LL Case →

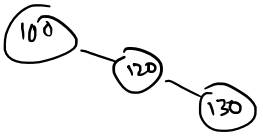
[Left of Left child]



⇒ LL Case

[New node added to left of left child of 100 so 100 unbalanced due to LL Case]

(2) RR Case

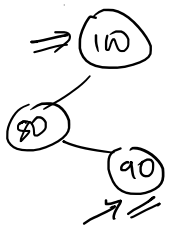


RR Case

New Node 130 added to Right of Right child of 100

So 100 is unbalanced due to RR case.

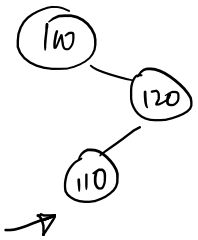
(3) L R Case



New Node 90 is added to Right of Left child of 100

∴ 100 is unbalanced due to L R Case

(4) R-L Case



New Node 110 is added to Left of Right child of 100

∴ 100 is unbalanced due to R L Case

Multiway Search Tree

(left ≤ parent
right > parent)

① B-Tree ⇒ It stands for balanced Tree

⇒ B tree grows upward.

⇒ All the leaf nodes are at same level

⇒ No Rotations are required.

⇒ In B Tree of order n (odd), Every node can have maximum n-1 values and can have max n children (key)

⇒ At each node in B Tree, except root, minimum n/2 (data/keys) is desired.

⇒ If node becomes full of keys/data then to balance the node, we have to perform split and propagate operation

[At Every node in B Tree of order n (odd) = { min = n/2 keys/values. ; max = n-1 keys/values }]

Q)

Construct B-Tree of order 5 using following keys.

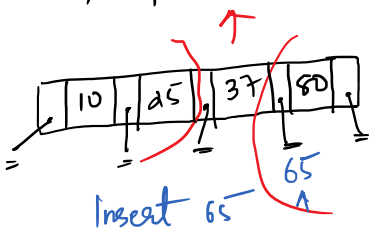
25, 37, 80, 10, 65, 50, 60, 90, 70, 85, 95, 15, 20, 30, 40, 45, 55

Sol→ Order n = 5

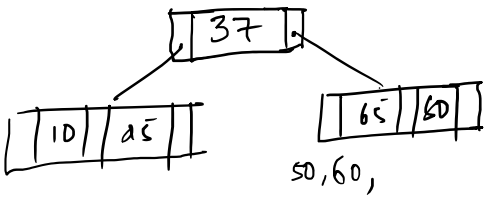
At Every node { min = n/2 = 2 value. ; max = n-1 = 4 values } After 4 values → split & propagate (up)

At Every node max = n-1 = 4 values } After 4 values → split

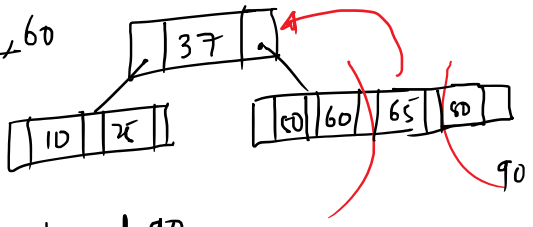
① Insert 25, 37, 80, 10 (Ensure Sorted)



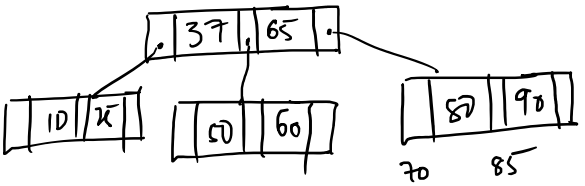
②



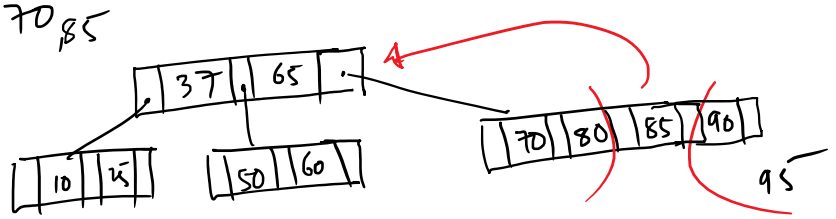
③ Insert 50, 60



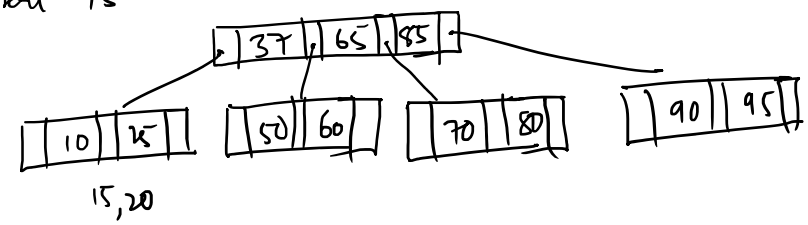
④



⑤ Insert 70, 85

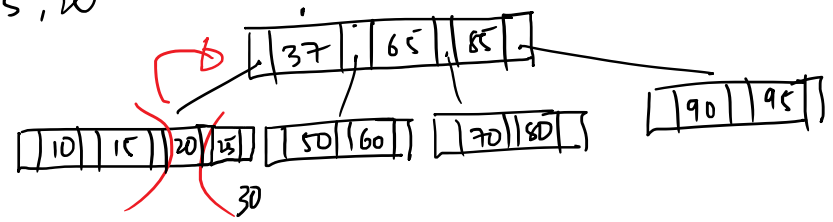


⑥ Insert 95



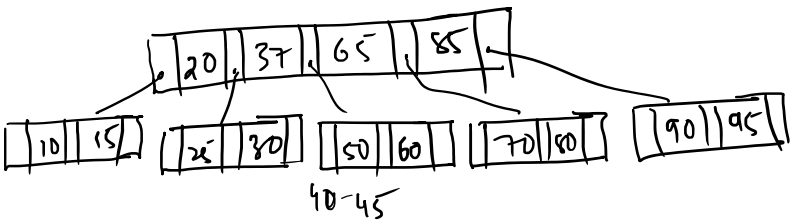
⑦

(7) Insert 15, 20

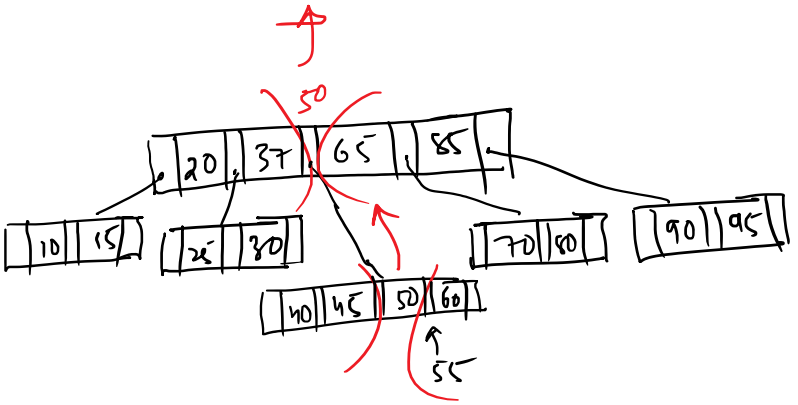


Insert 30

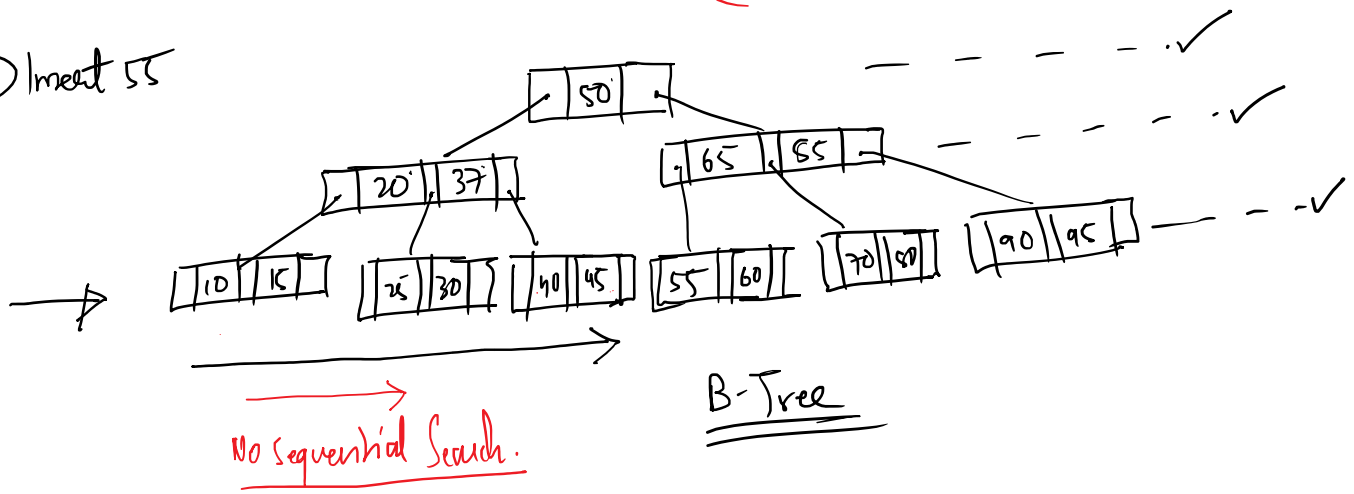
(8)



(9) Insert 40, 45



(10) Insert 55



Here the keys are records.

Hence the records are at all the levels.

B+ Tree :→

(1) In B-Tree, the elements (keys) are stored at all the levels.

(2) Here the records are kept at all levels.
Hence linear search for the leaf nodes is not possible.

(3) To achieve this, we must have all the elements at leaf node itself.

[(4) This can be achieved using following modification in B-Tree

(i) When leaf node is split and median node goes up, its (median) copy is stored in its left child.

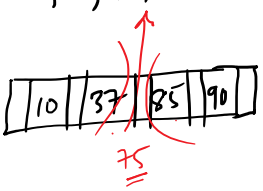
(ii) All the leaf nodes must be linearly connected.

The B-Tree with above modification is known as B+ Tree.]

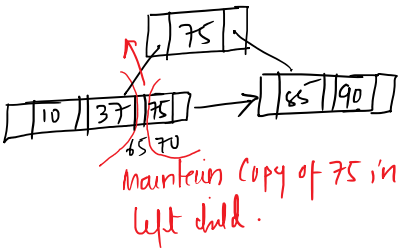
Construct B+ Tree using following (Order = 5) < min = $\lceil n/2 \rceil$ = 3 values, max = n-1 = 4 values

37, 85, 90, 10, 75, 65, 70, 55, 60, 45, 48, 35, 30

① Insert 37, 85, 90, 10

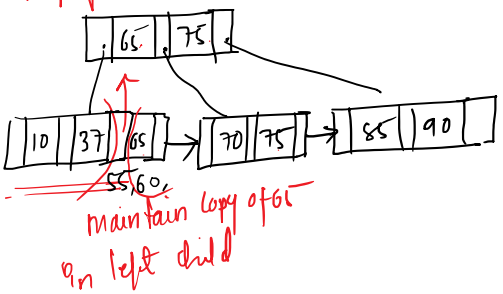


② Insert 75



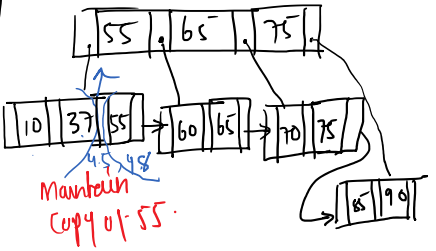
③ Insert 65, 70

propagate 65 to up.



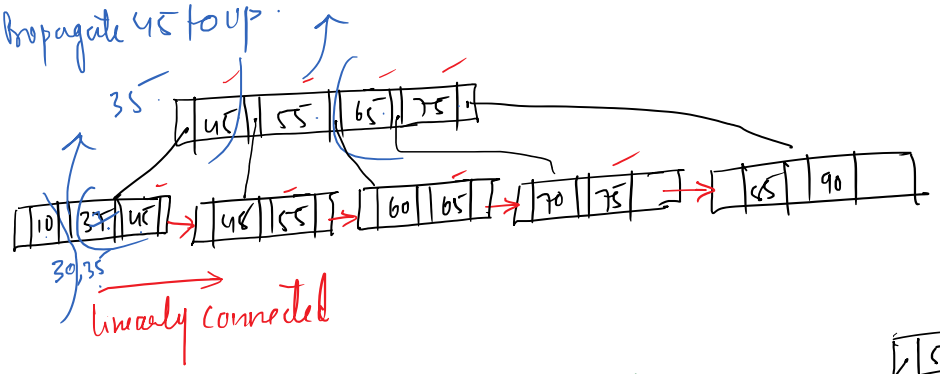
④ Insert 55, 60

propagate 55 to up.



④ Insert 45, 48

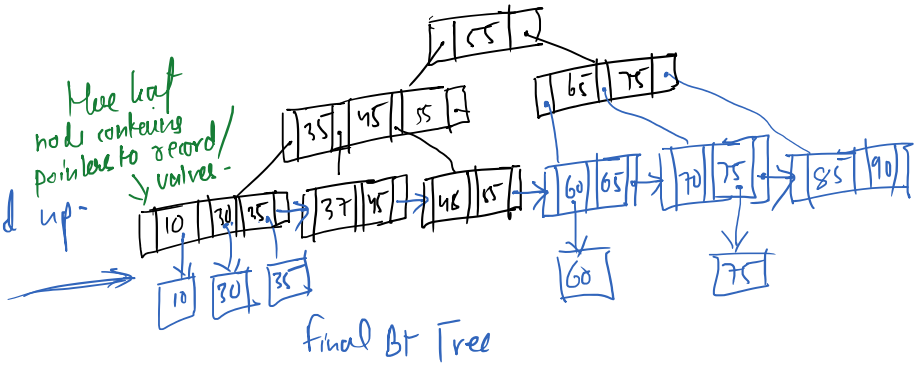
Propagate 45 to up.



⑤ Insert 35, 30

35 is propagated upward and now 55 is propagated up.

Here leaf node contains pointers to record/value.



Final B+ Tree

* All the elements are present in leaf.
* All leaf nodes are linearly connected
* Here the records are not stored in B+ Tree (So less memory is needed).
* Here at leaf we have pointers to the records stored in

Somewhere memory.

* Since the leaf nodes are linearly connected, we can perform linear search. in B+ Tree.

* In B-Tree, At every node we have records, and if we intend to store pointer then extra field is needed.

* In B+Tree, the leaf node has empty fields where we can store pointers.

* B+ Tree has its application in database Implement^{n}.

BST Construction →

Construct BST 100, 80, 85, 110, 105, 70, 86, 78, 116, 120, 123, 121