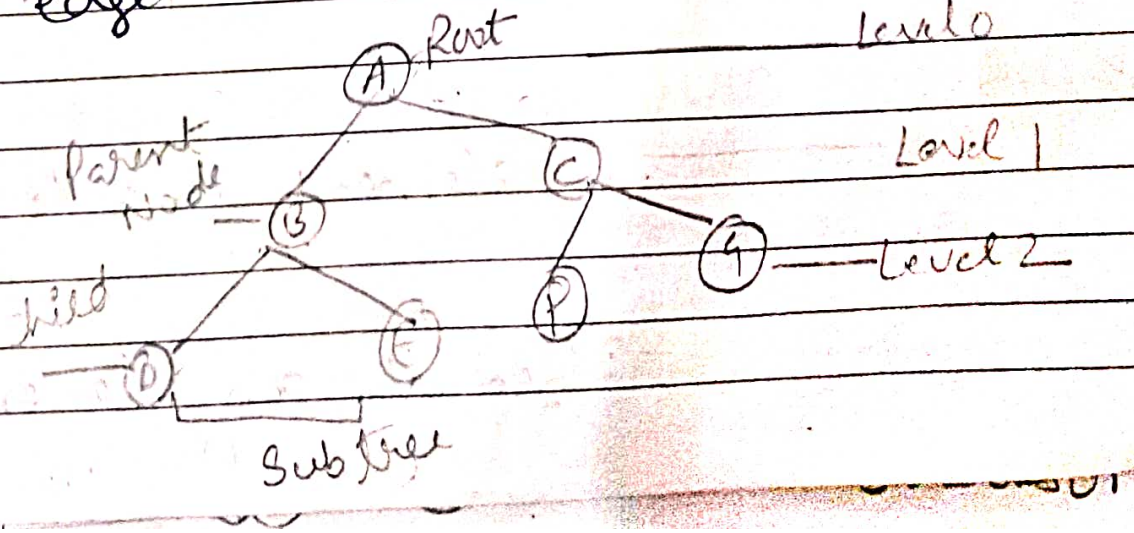
TREE's — A tree is a hierarchical data structure defined as a collection of Nodes. Nodes represent value and nodes are connected by edges.

- The tree has one node called root. The tree originates from this & hence it does not have any parent.
- Each node has one parent only but can have multiple childrens.
- Each node is connected to its children via edge.



PAGE NO. :
DATE : / /

Basic Terminologies

① **Root** - Special Node in a tree. The entire tree originate from It & It does not have any parent.

② **Parent Node** - It is an Immediate Predecessor of a Node.

③ **Child Node** :- All Immediate Successors of a Node are its Children.

④ **Leaf** :- Node which does not have any children.

⑤ **Edge** :- - It is a Connection b/w 2 Nodes. or a line b/w 2 Nodes or leaf.

⑥ **Siblings** - Node with the same parent

⑦ **Subtree** :- descendants of a Node represents Subtree.

⑧ **Path | Traversing** :- No. of Successive Edges from Source node to destination Node.

* **Basic Operations** :- ① **Insert** :- Insert element in a Tree / Create a tree.

② **Search** :- Search an Element in a tree

In Example Node A has become ...

PAGE NO. :
DATE : / /

(3) Traversing — Pass through the tree in a specific manner it contains (Pre, Post & In order Traversing).

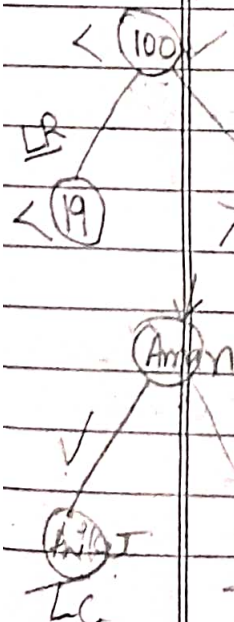
* Tree Node Code —


```
Struct Node {
    int data
    Struct Node *leftchild;
    Struct Node * RightChild;
};
```

1). Insert operation — 1st Insertion Creates the tree. Afterwards an element is to be Inserted, 1st locate its proper location. Start searching from the root node, then If the data < the key Value, Search for the empty location at left Side & if data is > the key Value Search empty location in Right Side.

Algo —

```
If root is NULL
    then Create root Node
    return
If root Exists then
    Compare the data with Node.data
While Until insertion position is located
If data is greather than node.data
    go to right Subtree
Else
    goto left Subtree.
end while
```



insert data
end If

② Search operation :- If Root.data = Searchdata
return root
else
while data not found
If data is greater than node.data
go to right Subtree
else
go to left Subtree
If data found
Return node
end while
return data not found
end If.

* TREE TRAVERSAL :- Traversal is a process to visit all the nodes of a tree and may print their values too. all Nodes are connected via edges. we always start from the Root node. We cannot Randomly access a Node in a tree. There are 3 ways to Traverse a Tree.

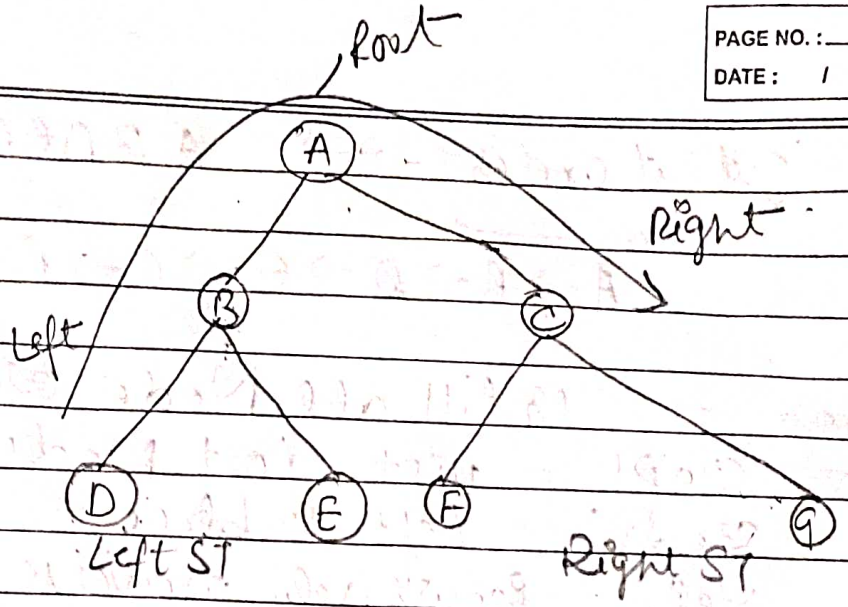
A+B

① In-Order :- In this Method, the left Subtree is visited first then the Root and later the Right

In Example "Node A has ..."

PAGE NO. :
DATE : / /

Eg :-



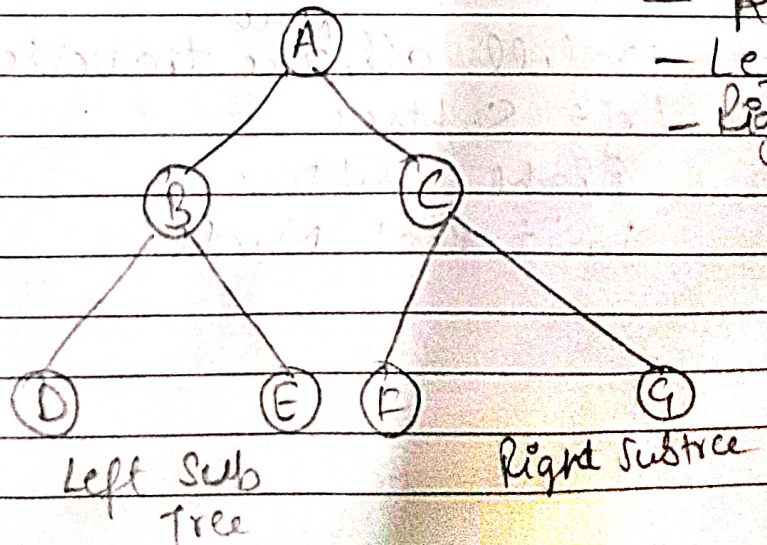
Output of Inorder Traversal is

D → B → E → A → F → C → G

Algorithm — Untill all nodes are Traversed —
Step 1 :— Recursively Traverse leftSubTree
Step 2 : — Visit Root Node
Step 3 : — Recursively Traverse Right Sub tree

2. Pre-Order :— In this, The Root Node is visited 1st after that Left Node then Right Node.

— Root
— LeftChild
— Right Child.



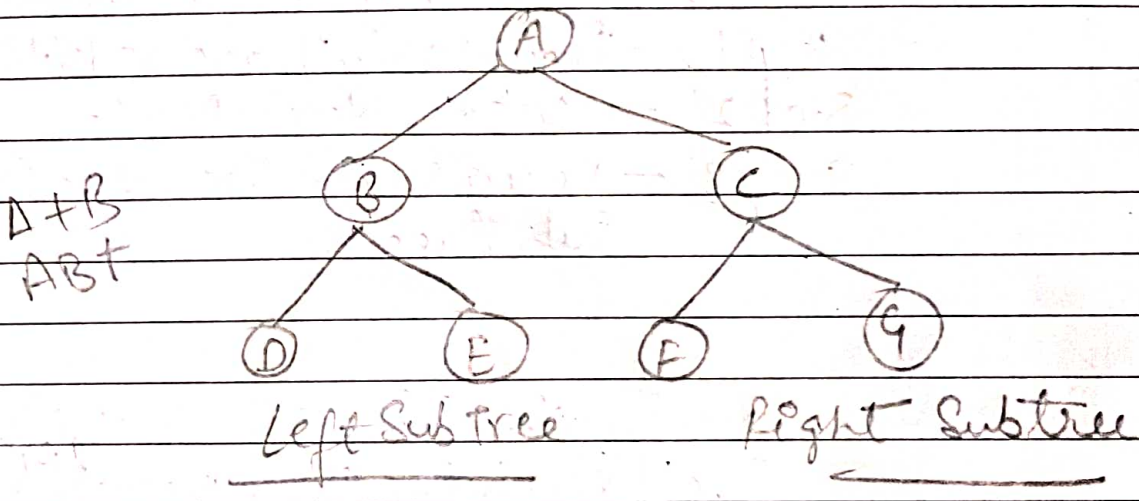
PAGE NO. :
DATE : / /

Output order — A B D E C F G

A → B → D → E → C → F → G

Algo — Until all Nodes are traversed
Step 1 :— Visit Root Node
Step 2 :— Left Subtree
Step 3 :— Recursively Traverse Right Subtree

3) Post order — In this Root Node visit last, 1st Traverse left Node & then Right.



The Output is = D → E → B → F → G → C → A

Algo — Until all Nodes are traversed
① Left Subtree
② Right Subtree
③ Visit Root Node

Sub Example: "Node A has become ..."

Types of Trees

PAGE NO.: _____
DATE: / /

There are Mainly four types of trees used in data structures.

1. Binary Tree
2. AVL Tree
3. Binary Search Tree
4. Threaded Binary Tree

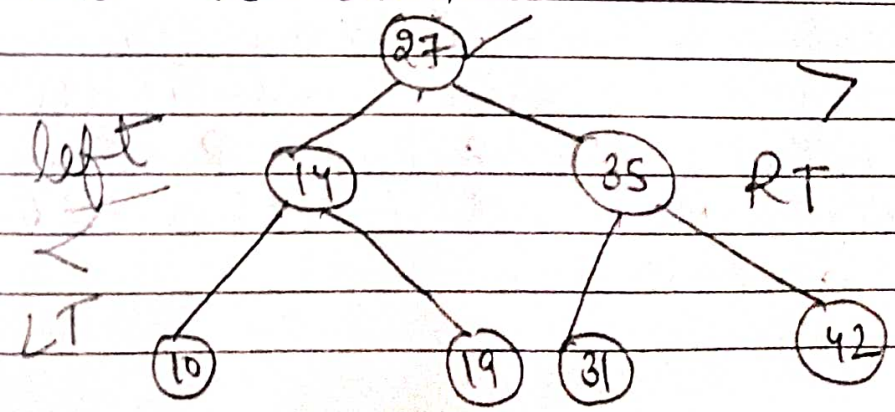
(1) **Binary Search Tree (BST)** :— In which,

- The left Sub tree of a Node has a key less or equal to (≤) its parent node
- The Right Sub tree of a Node has a key > to its parent Node's key.

Thus BST, divides all its Sub-Trees into 2 Segments, the Left & Right Subtree

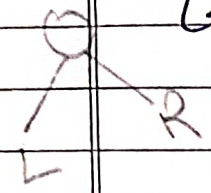
Left_Subtree (Keys) ≤ node (Key) ≤ Right_Subtree (Keys)

Representation — Collection of Nodes arranged in a way, while Searching the desired key is compared to the keys in BST & If found, the associated value is Retrieved.



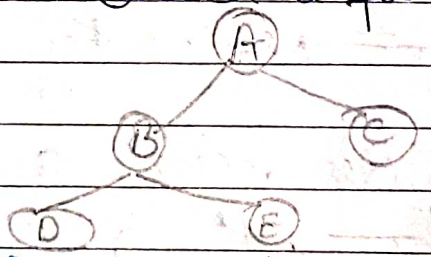
The Root Key (Q7) Contains all less Valued Keys on Left Side & More Valued Keys on Right Side.

(2) Binary Tree :— In this, Every node can Have at most 2 Children Left & Right.

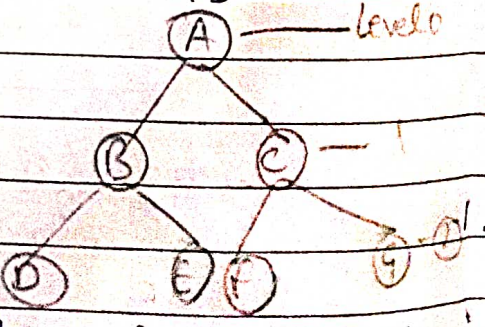


Fully BT Perfect BT Complete BT

* Fully BT :— If Every Node in a tree has either 0 or 2 Children then the tree is Called a full Tree.



* Perfect BT :— It is a binary tree in which all Interior Nodes have 2 Children and all leaves have the same depth and same level.

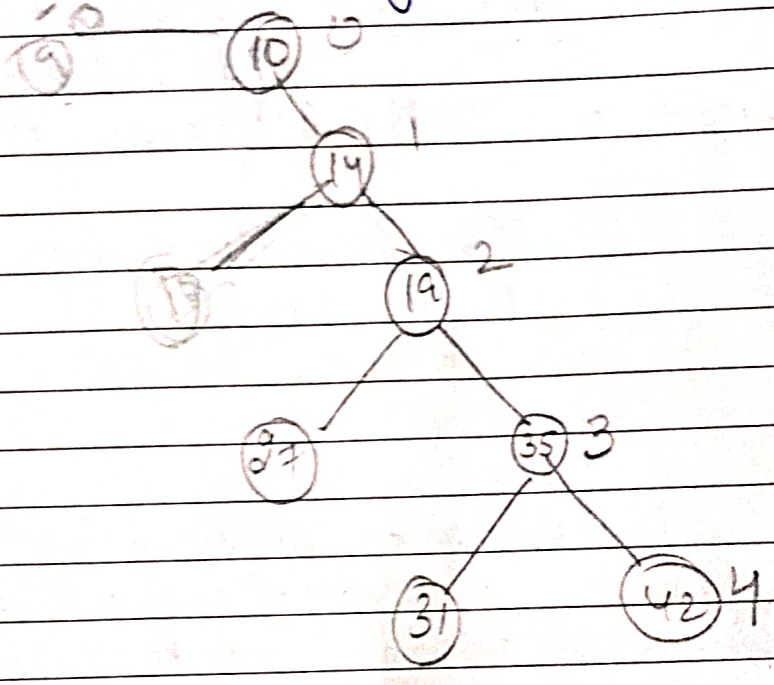


* Complete BT — A Binary tree is Complete Binary tree if all levels are Completly filled Except Possibly the last level and last level has all keys as left as possible.

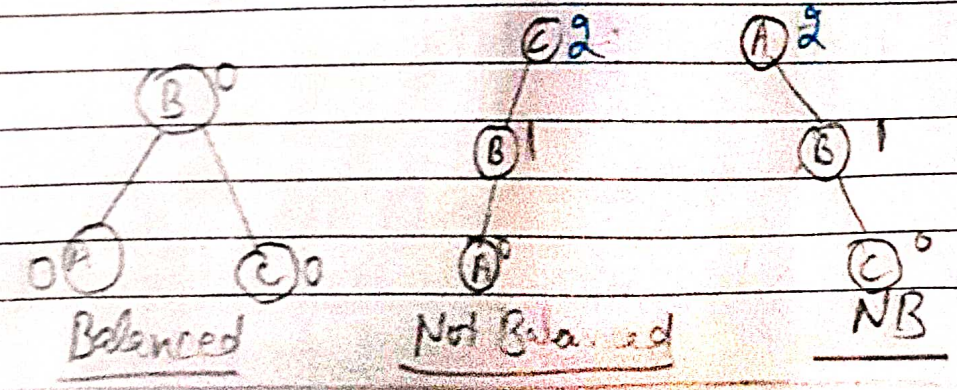
PAGE NO. :
DATE : / /

AVL Tree

- It is observed that BST, needs to arises Balance out the Existing BST.



- To Balance the BST's, AVL Trees used to Balance out.
- AVL Stands for — Adelson, Velski & Landis, AVL trees are height Balancing binary Search tree. AVL Checks the height of left & right Sub Trees and assures that the difference is not More than 1. This difference is Called Balance factor.



Here we see the 1st tree is balanced & rest of 2 is not balanced.

PAGE NO. :
DATE : / /

⇒ In Second Tree → the left Subtree of C has height 2 and the right Subtree C has height 0. So the difference is 2.

→ In 3rd Tree :— the height of A is 2 and left Subtree of A has height 0. So the difference is 2.

* The AVL Tree permits difference (Balance factor) to be only 1.

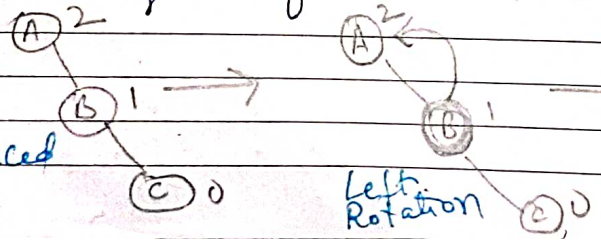
Balance factor = (height(left-Subtree) − H(Right-Subtree)

* If the difference in the height of left & Right Subtree is More than 1. the tree is balanced using Some Rotation techniques.

AVL Rotation

- Left Rotation
- Right Rot.
- L-R Rot.
- Right-left Rot.

① Left Rotation :— It is Single Rotation. If a tree becomes unbalanced when a Node is inserted into the Right Sub tree of the Right Subtree, then we perform a Single left Rotation.



Complete BT

ARRAYS

Array is a Container which can hold a fix no. of items & these items should be of the Same type. Most of data structure make use of Arrays to Implement their algo. (homogenous data)

- Element :— Each item Stored in an array is called an Element.
- Index :— Each location of an element in an array has a Numerical Index, which is used to Identify the Element.

Array Representation :— Arrays can be declared in various ways in different languages.

```
int array[10] = {35, 33, 42, 10, 14, 19, 27, 44, 26, 31};
```

(int: Type; array: Name; [10]: Size)

Index	0	1	2	3	4	5	6	7	8	9
Element	35	33	42	10	14	19	27	44	26	31

Size = 10

- Index starts with 0.
- Array length 10 Means It can Store 10 elements.

PAGE NO.:
DATE: / /

- Each Element can be accessed via its index for Eg. we can fetch Index 6 as 27.

Basic Operations — ① Traverse — Prints all array elements one by one

② Insertion & Deletion — Add & delete element at the Given Index.

③ Search — Search an element using Index or by the Value.

④ update — Updates an Elements at the Given Index.

Array Insertions — ① Insertion at the beigning of an Array.

A[2, 3, 4, 5] → A[0, 2, 3, 4, 5]
0 1 2 3 0 1 2 3 4

② Insertion at the Given Index of an Array.

A[1, 2, 4, 5] → A[1, 2, 3, 4, 5]
0 1 2 3 0 1 2 3 4

③ Insertion after the given Index of an Array.

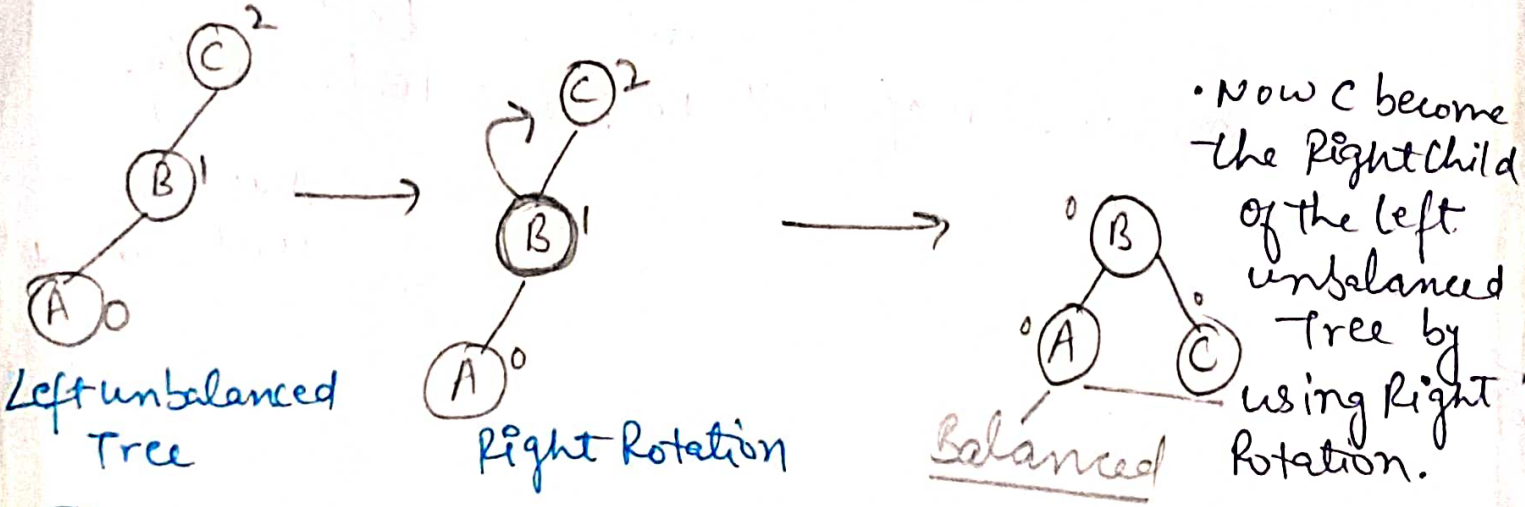
A[1, 2, 4, 5] → A[1, 2, 3, 4, 5]
0 1 2 3 0 1 2 3 4

④ Insertion before the Given Index of an Array.

A[1, 2, 4, 5] → A[1, 2, 4, 5, 3]
0 1 2 3 0 1 2 3 4
↑
3

In our Example, Node A has become Unbalanced as a Node is inserted in the Right Subtree of A's Right Subtree. We Perform the Left Rotation by making A the left Subtree of B.

② **Right Rotation** :— AVL Tree May become unbalanced, If a Node is inserted in the left Subtree of the left subtree. The tree then Needs a Right Rotation.

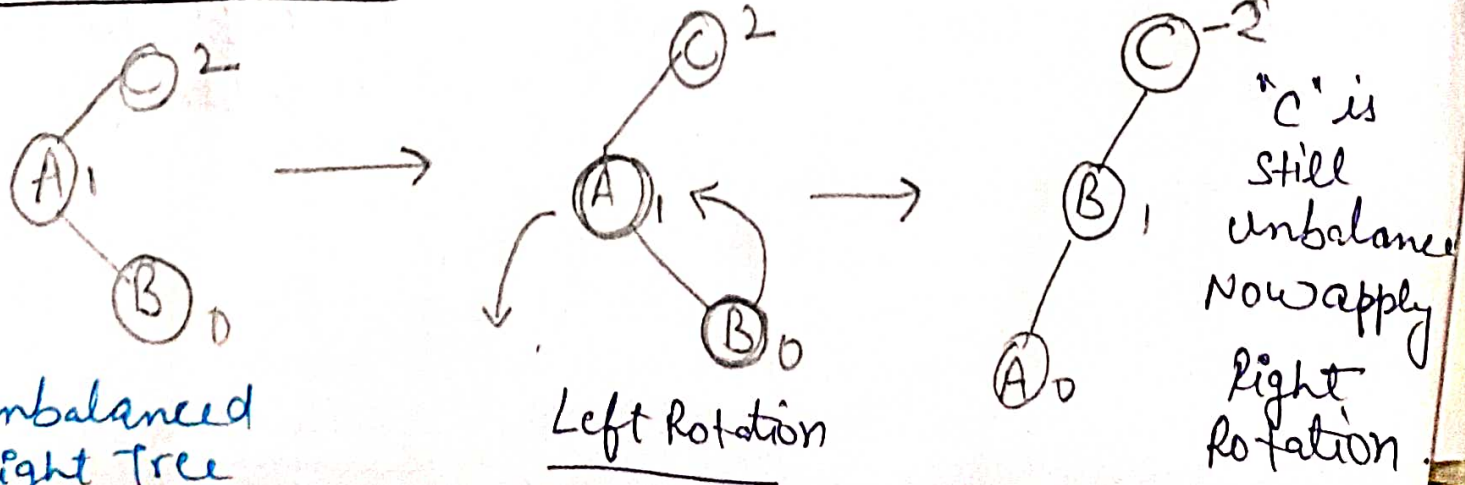


③ **Left - Right Rotation** :— It is Complex Version of upper Performed Rotations.

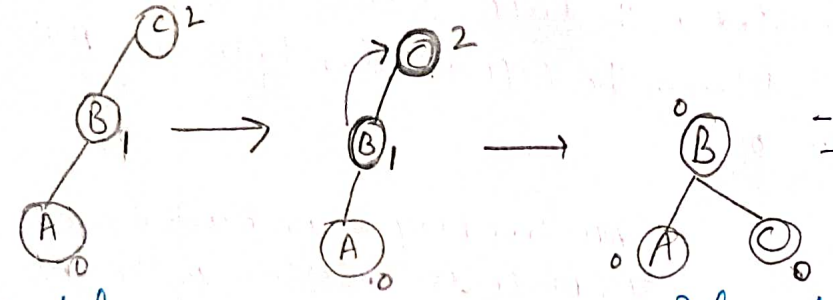
- a Left Right Rotation is Combination of L-R Rotation

1st Perform — Left Rotation ✓

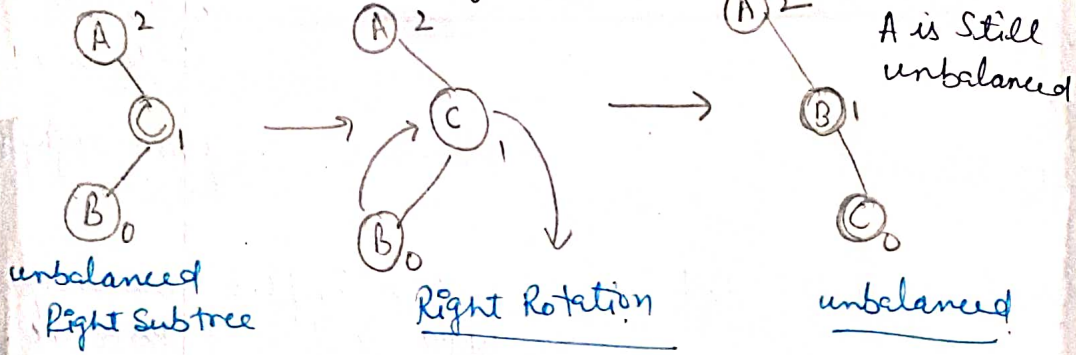
2nd — Right ✓



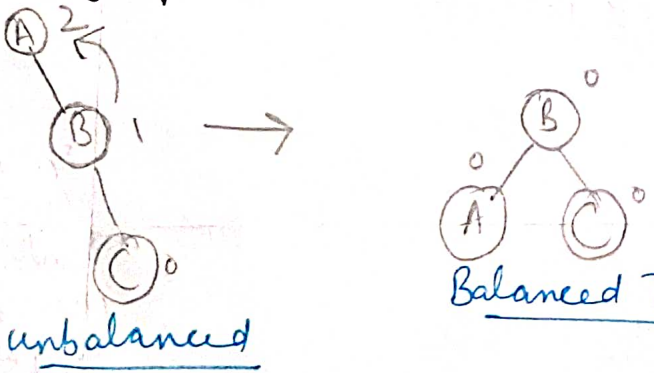
⇒ Apply Right Rotation

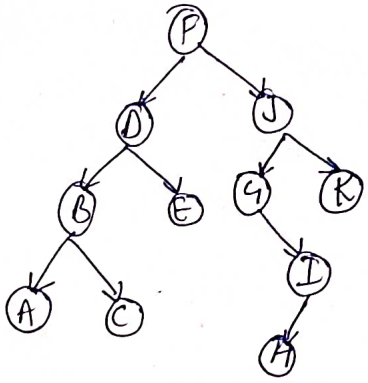


④ Right Left Rotation :— 1st Rotation — Right
2nd — Left
It is combination of both Rotations respectively.



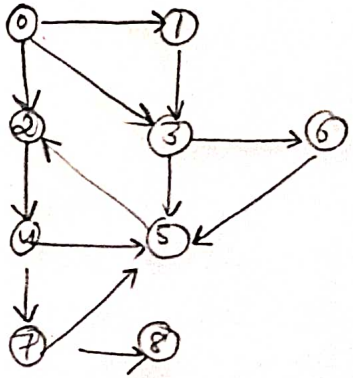
↳ Apply left Rotation :—



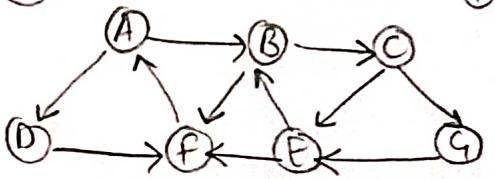


BFS :—

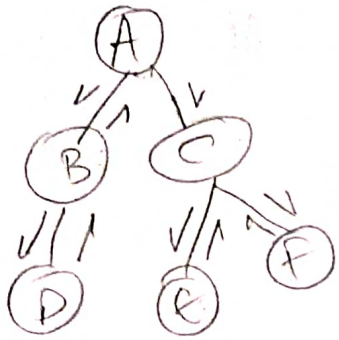
①



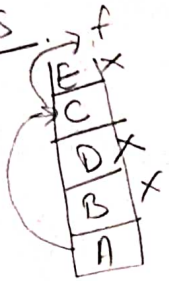
BFS ②



Q)



DFS



LIFO

BFS

① A | B | C |

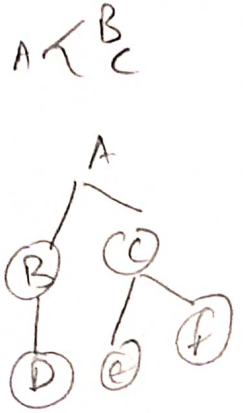
A → B | C | D

B → C | D

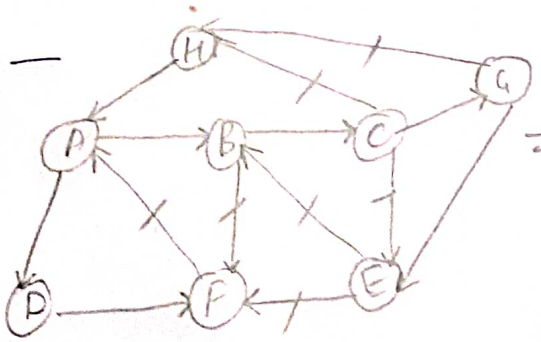
D → D | E | F

E | F

EX FX



DFS :—



⇒ 7 unused Path Removed

Threaded Binary Tree (TBT) :—

A binary Tree is Threaded by making all right child Pointers that would Normally be Null Point to the Inorder Successor of the Node (If it exists).

- All left child Pointers that would normally be Null point to the inorder predecessor of the Node.

* Thread :— The Pointers Refrence the Next Node in an Inorder Traversal Called Threads.

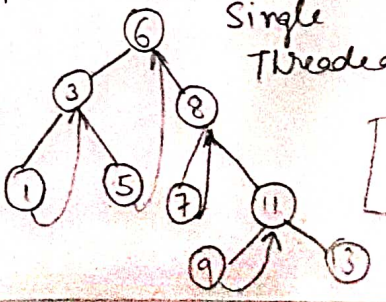
* Why do We Need Threaded Binary Tree :—

* Binary Tree have a lot of wasted space. the leaf Node each Have 2 Null Pointers. we Can Use these Pointers to help us Inorder Traversals.

* Threaded binary Tree Makes the tree Traversal faster.

* Types of TBT
  - Single either → Right / ← left
  - Double → R & L

① Single TBT :— Each Node is threaded towards Either the inorder Predecessor or Succesor (left or Right)

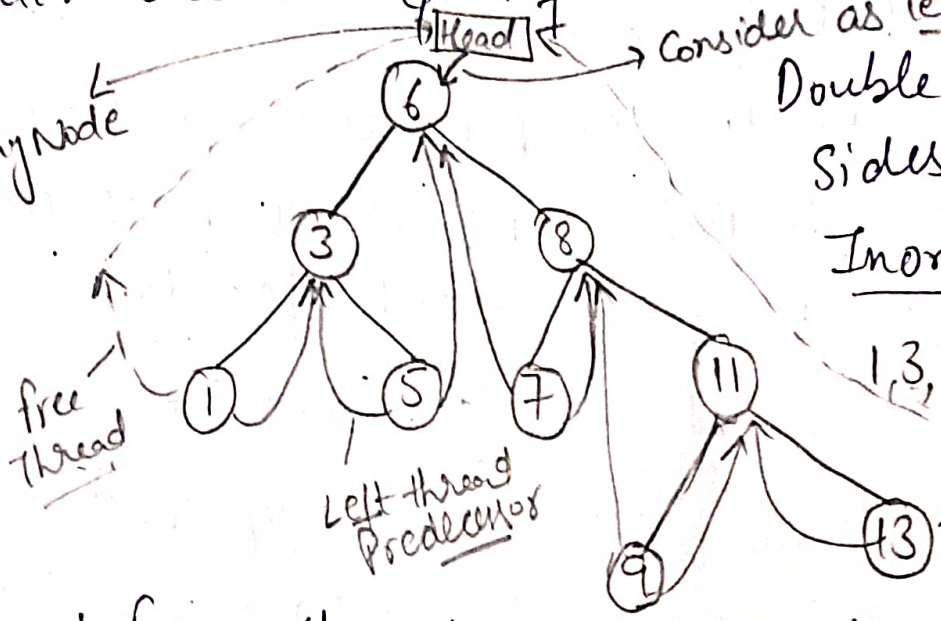


Single Threaded Rightside

1, 3, 5, 6, 7, 8, 9, 11, 13

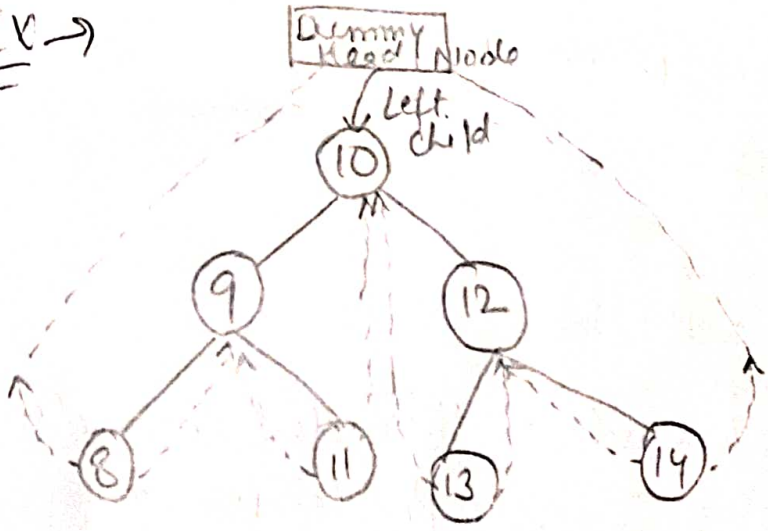
Inorder Successor

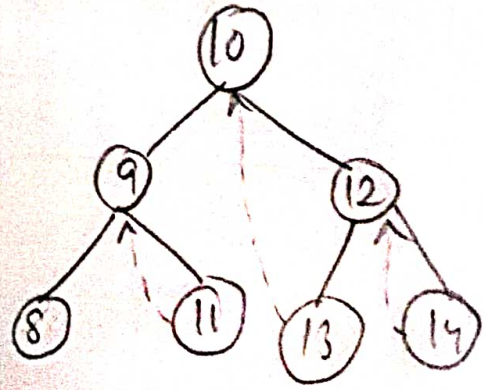
② Double TBT :— Each Node is threaded Towards both inorder Predecessor & Successor (left & Right). all Right Pointers will Point to inorder Successor & leftside → Predecessor.



+ free Threads are called "Dangling Pointers"

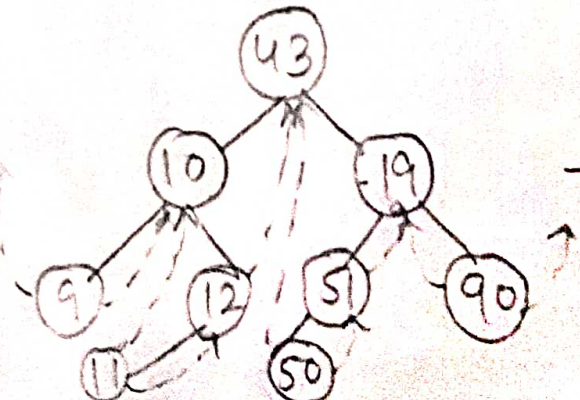
Ex →





Single Thread Left Thread

Inorder — 8,9,11,10,13,12,14

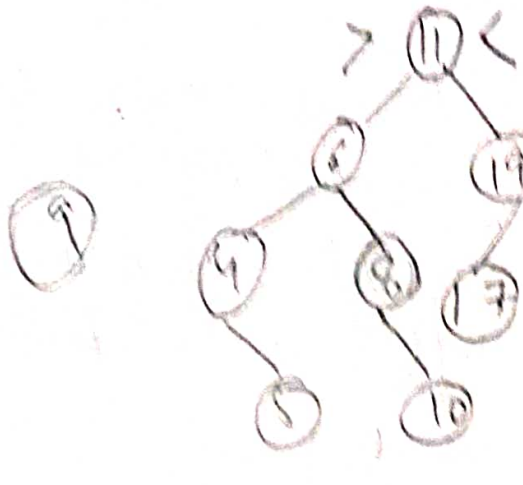


— 9,10,11,12,43,50,51,19,90

Operations on Binary Search Tree :—

Insertion | Deletion

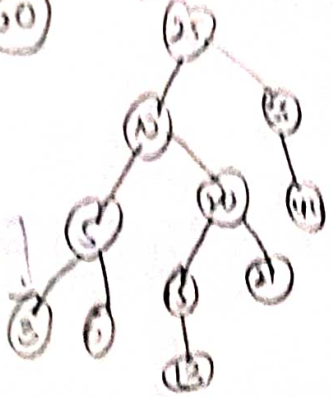
(1) **Insertion** :— 11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31



Insert - 60 .

- Compare each value on right side

Right Node

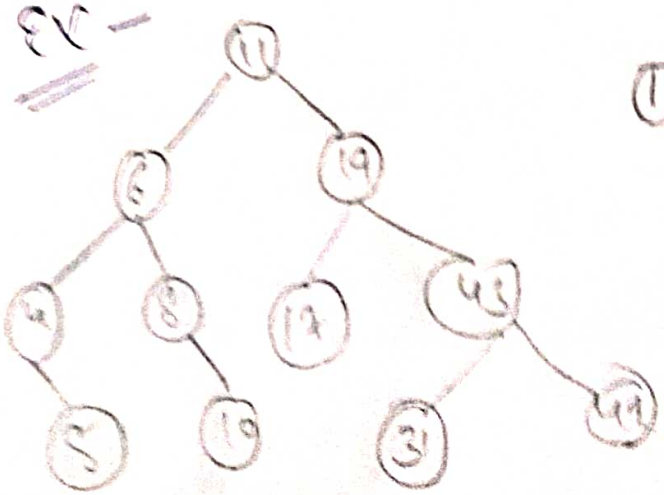


(2) **Deletion** :— having 3 conditions

(1) 0 child (2) 1 child

(3) 2 child

Ex —

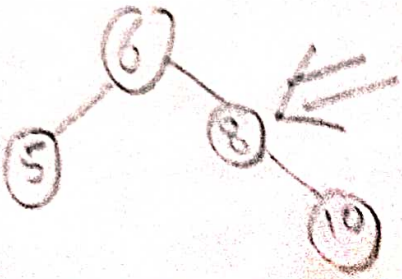


(1) **0 child** :— delete the Node having 0 child.

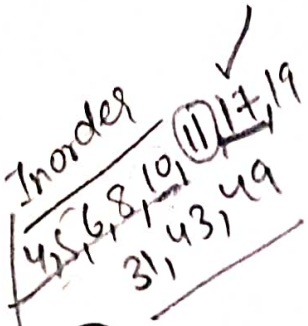
Ex — delete 31 because No child is there

(2) **1 child** :— delete Having 1 child value.

Ex → 5 only single child of 4.



③ 2 Child :- delete Node having 2 Childs but Having 2 Cases

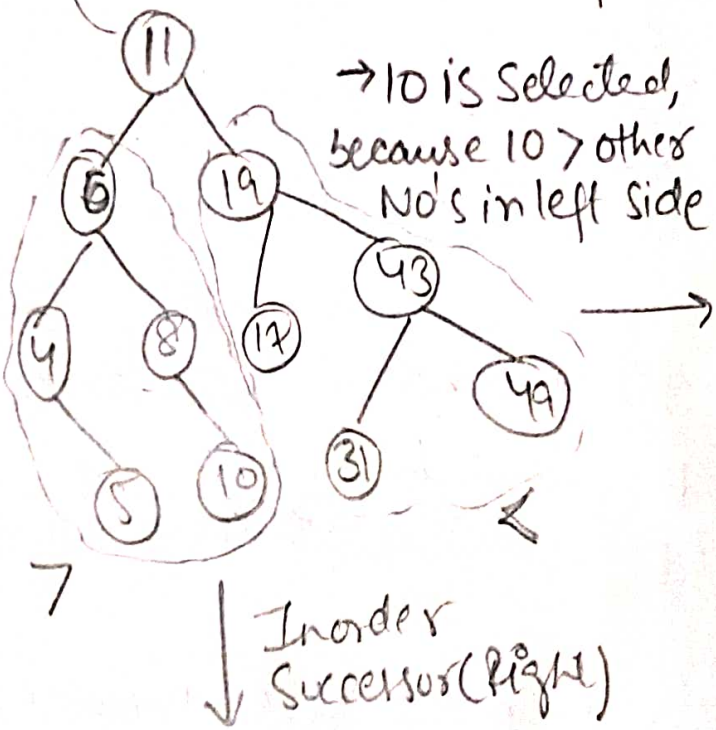


① Inorder Predecessor

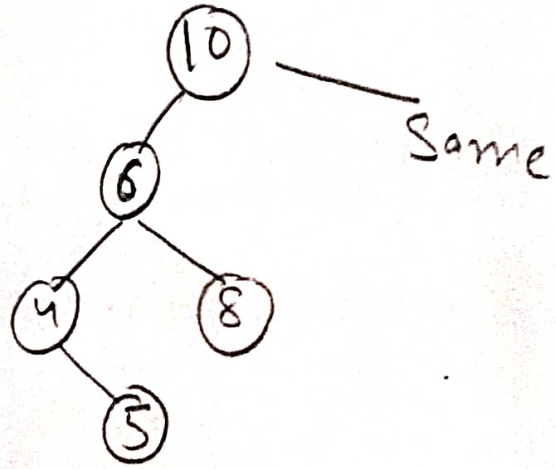
② Inorder Successor

① Inorder Predecessor :- (left) for Eg Remove 11 having 2 Childrens.

- So, select one No. from left side to Replace 11.



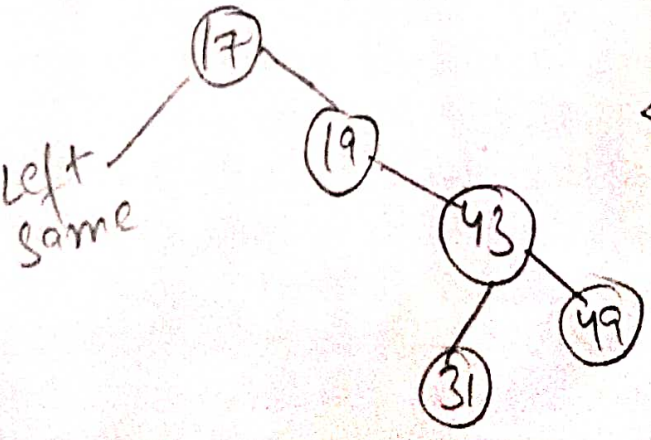
→ 10 is Selected, because 10 > other No's in left side



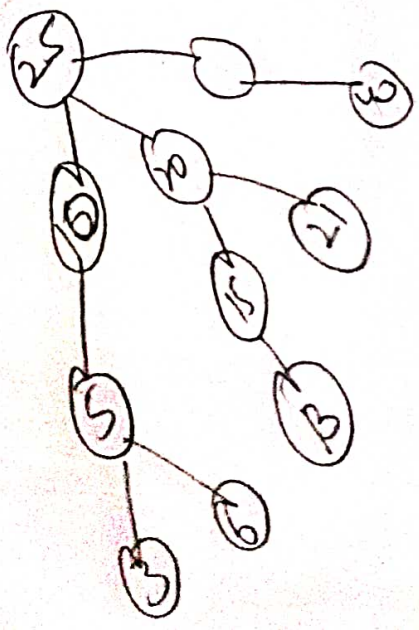
Same

Inorder Successor (Right)

- Select Min or < No. for Inorder Successor from Right Side



Left same



B-TREES

→ B-Tree is a specialised m-way tree that can be widely used for disk access. A B-tree of order 'm' can have at most 'm-1' keys & 'm' children. One of the main reason of using B-tree is its capability to store large no. of keys in a single node.

① Every node in a B-tree contains at most 'm' children

$m = 5$

② Every node in a B-tree except the root node & the leaf node contains at least $m/2$ children.

③ The root nodes must have at least 2 nodes.

④ All leaf nodes must be at the same level.

Note - It is not necessary that all nodes contains the same no. of children but each node must have $m/2$ no. of nodes.

$m = 5$

$\frac{m}{2} = \frac{5}{2} = 2.5$

(3) ceiling value.

keys $= m-1$
$5-1$
$= 4$

K_1 K_2 K_3 K_4 — ⑤ & 4 keys

root node

children

same level

- Keys — $m-1$
  $5-1=4$ ✓

Min Keys — Root Node — 1

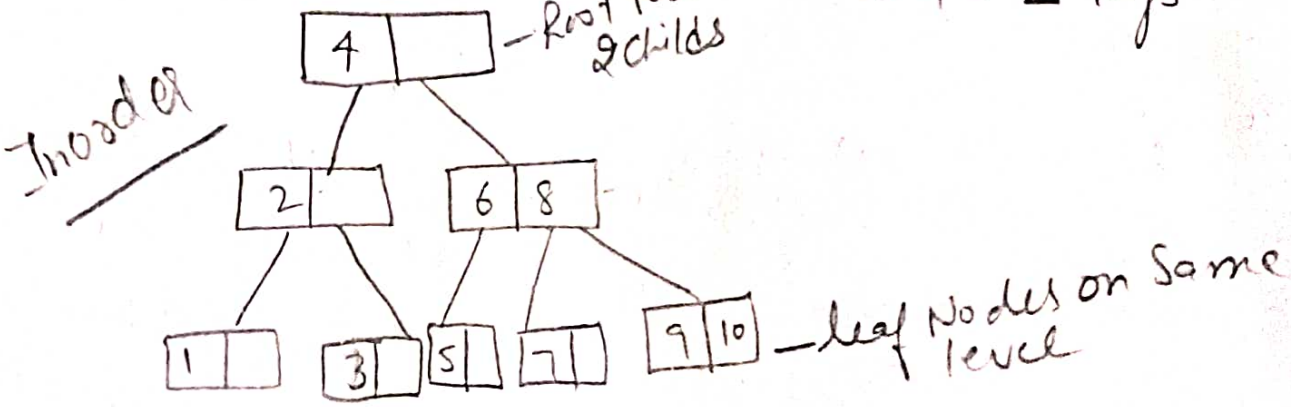
- all other Nodes $= \left\lceil \frac{m}{2} \right\rceil - 1$

$\frac{5}{2} = 2.5 \rightarrow 3$ $\quad 3-1=2$

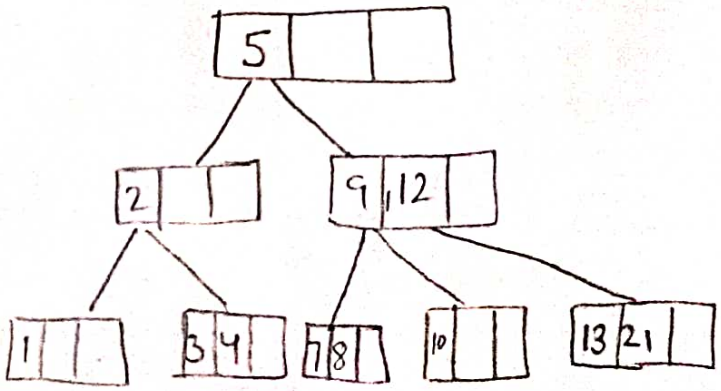
Ex → Order = 3
insert Values from 1–10.

$m=3$
$m-1=2$ Keys

Inorder



Ex → B Tree for order = 4 (m=4)
5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8



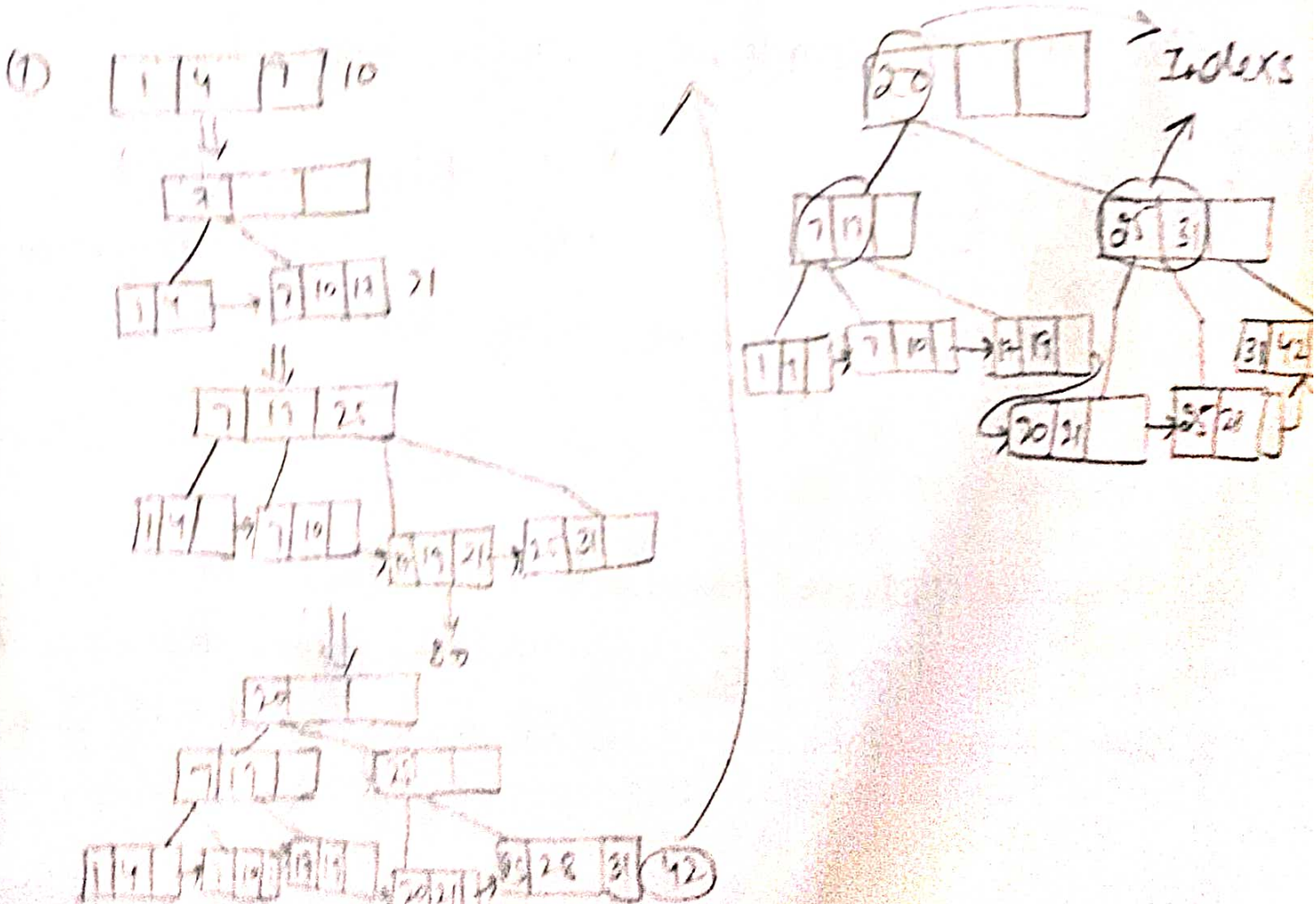
1, 2, 3, 4, 5, 7, 8, 9, 10, 12, 13, 21 — Sorted data

B+ Trees

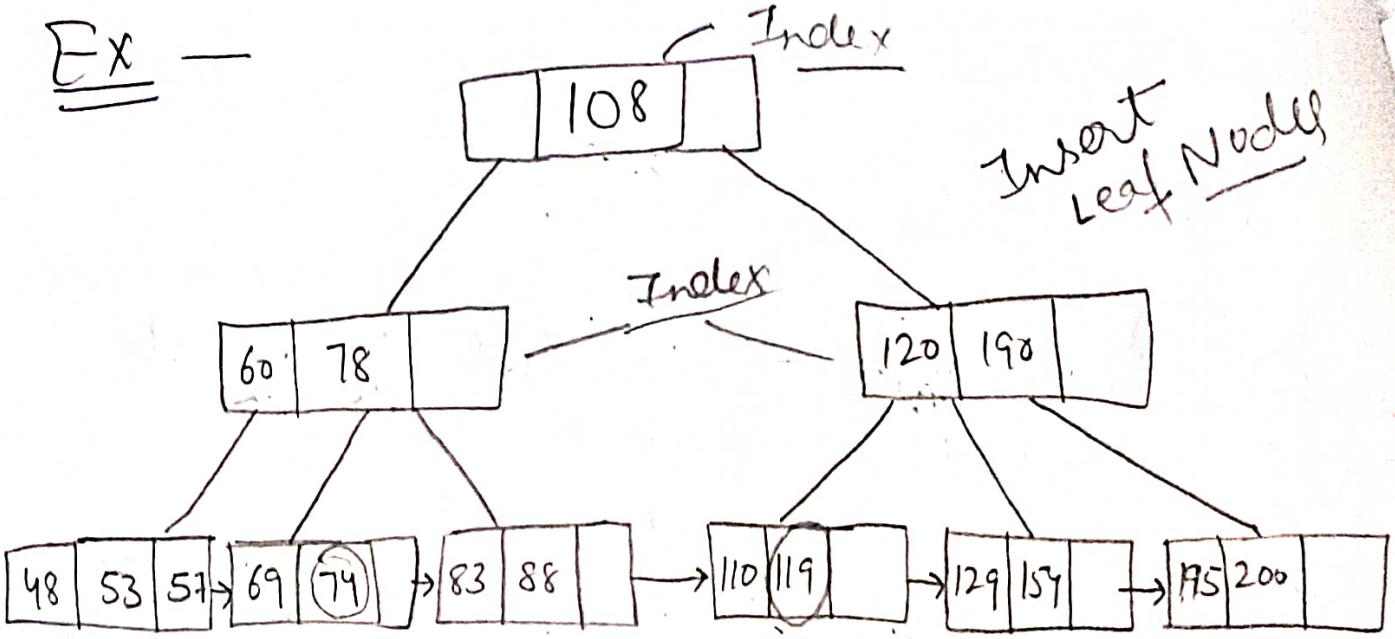
- It is an Extension of B Tree.
- The leaf nodes of a B+ Tree are linked Together in the form of Single linked lists to make the Search queries more Efficient.
- It stores large amount of data.
- The internal nodes of B+ Trees are often called Index Nodes. It stores data only in leaf Nodes not in Root Node.

Ex - Order = 4 ; min key = 1
max child = 4 = 3
min child = $\frac{m}{2}$ = 2

1, 4, 7, 10, 17, 21, 31, 25, 19, 20, 28, 42 [Index values does not repeat]



Ex —



Insert Leaf Nodes

Q) Difference b/w B & B+ Trees

B Tree	B+ Trees
* Search keys can not be repeatedly stored.	* Redundant search keys can be stored.
* data can be stored in leaf nodes as well as internal nodes.	* Data can only be stored on the leaf nodes.
* Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	* Searching is faster bcz data can only be found in leaf nodes.
* Deletion of Internal nodes are so complicated & Time Consuming.	* deletion will never be a complexed process bcz element will always be deleted from the leaf nodes.

leafs can not be linked together

the search operations more efficient ✓

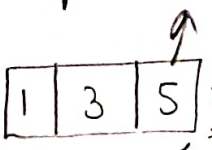
Q B+ TREE – Insert

1, 3, 5, 7, 9, 2, 4, 6, 8, 10

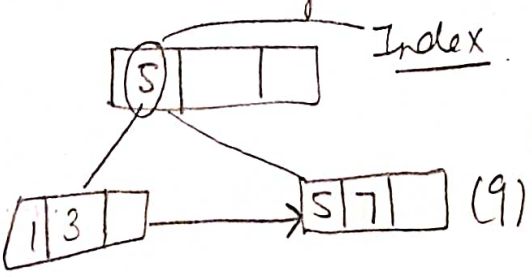
order = 4

keys = 3

① 1, 3, 5

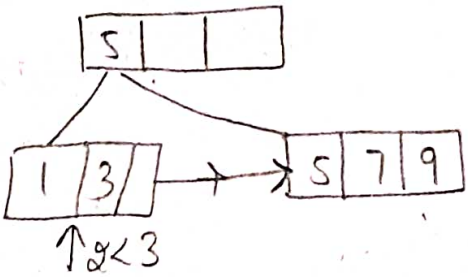


② Right Move Index



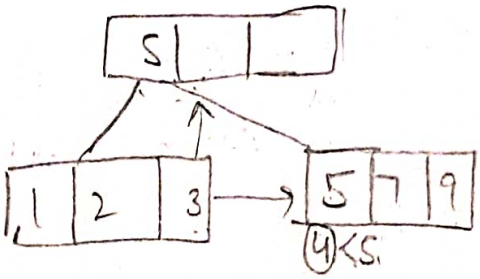
Add (9)

③



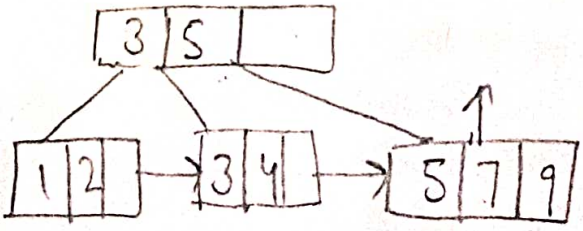
(Add 2)

④



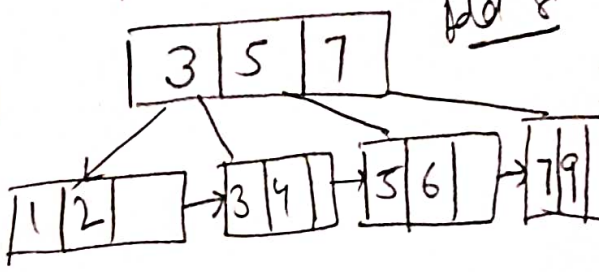
(Add 4)

⑤



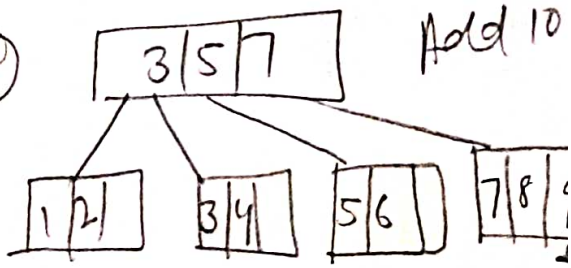
Add 6

⑥



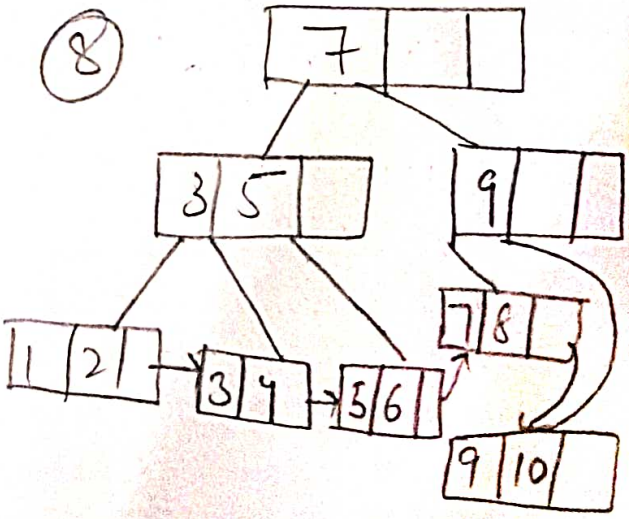
Add 8

⑦



Add 10

⑧

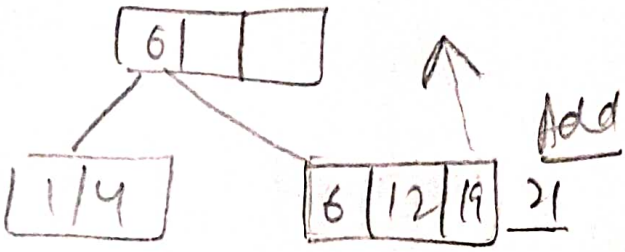


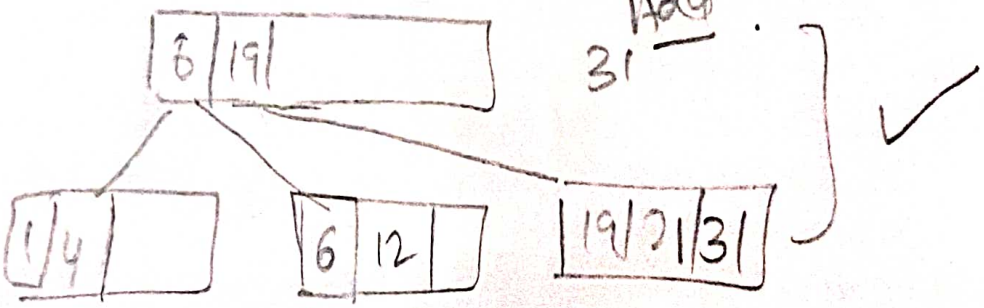
Q. 1, 4, 6, 12, 19, 21, 31 → B+ Tree

(m) order-4

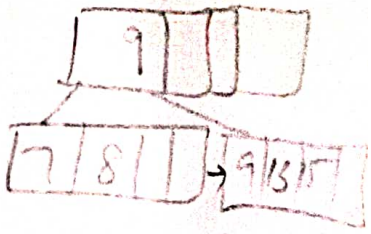
Keys = m − 1 = (3)

① 

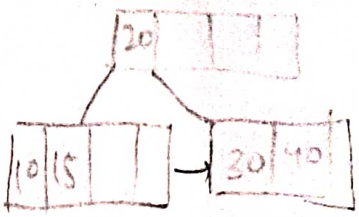
② 

③ 

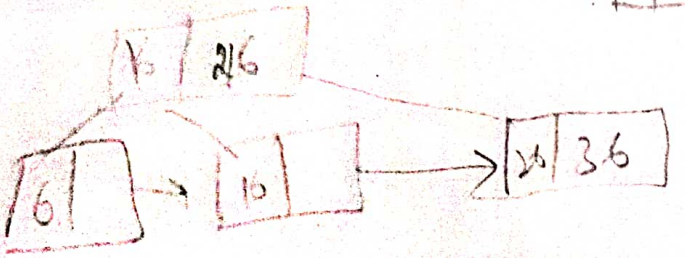
Q ① 7, 9, 13, 15, 8 order = 5



② 10, 20, 30, 40, 15 → order = 5



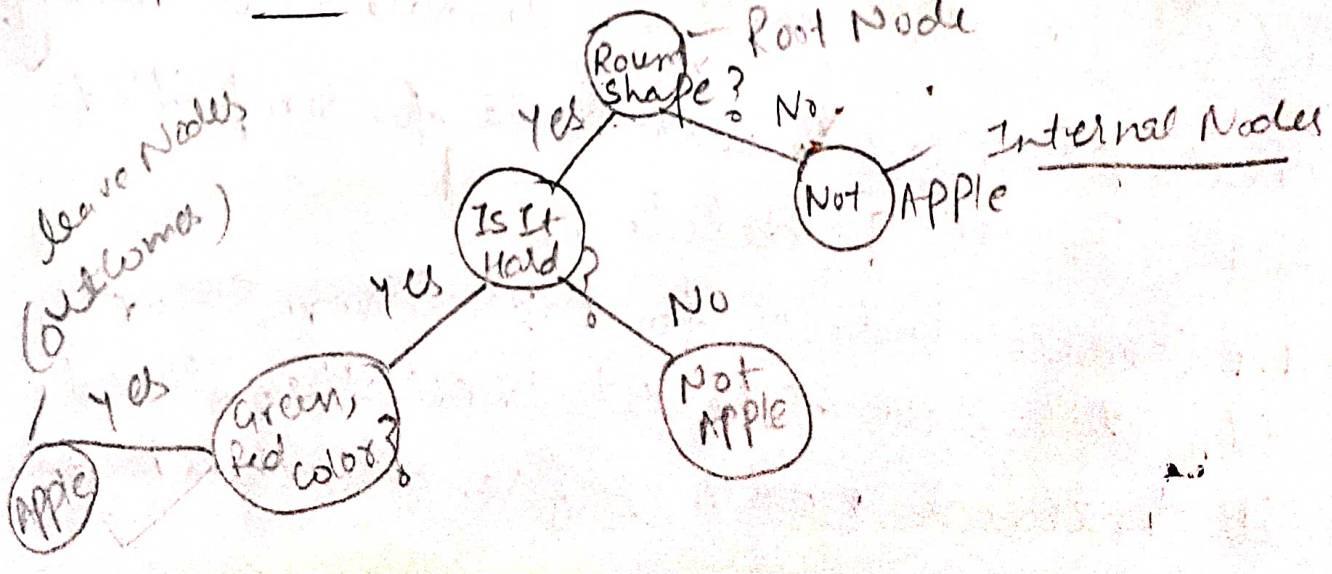
③ 6, 16, 26, 36, 46 order = 3



I → Applications Of Binary Trees :- Binary Trees are Mainly Used for Searching & Sorting as they Provide a Means to Store data Hierarchy. Some Common operations that can be Conducted on Binary Trees include Insertion, deletion & Traversal.

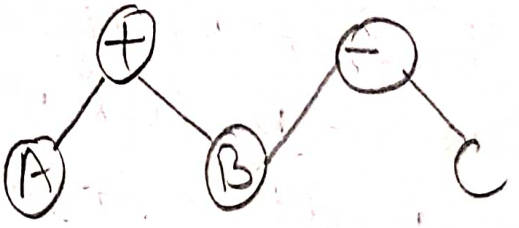
① Routing Tables :- The tree data structure will Store the location of routers based on their IP Addresses. Routers with similar Addresses are grouped under a single Subtree.

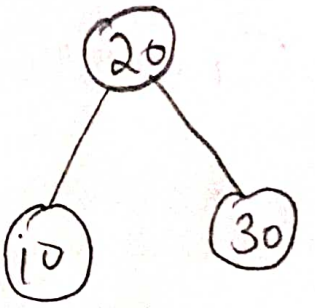
② Decision Trees :- It is used for classification Purposes. The Binary tree data structure is used here to emulate the decision making Process.
- A decision tree usually begins with a Root Node. The internal Nodes are conditions on data set features.
- Branches are decision rules while the leave Nodes are the outcomes of the decision.



(3) Expression Evaluation :— Used in Mathematical Expressions are Statements with operators and operands that evaluate a value. The leaves of the BT are the Operands while the internal Nodes are the operators. This

Ex :— A + B − C





Binary Search Tree

(4) Sorting :— A BT is simply an ordered or Sorted binary Tree such that the value in the left child is < than the value of Root Node. and Right child is ≥ than the Root Node.

- To complete a sorting procedure, the items to be sorted are first inserted into a Binary Search Tree.

(5) Indices for databases :— In databases indexing, B^+ Trees are used to sort data for simplified Search, Insert & delete.

- The DB Creates indices for each Given record in the database. The BT Trees then stored in its internal nodes, reference to data records

- With the Actual data records in its leaf nodes.
- This provide Sequential Access to data in the DB's.

(6) Data Compression :- In data Compression, it is used to Compress data by using fewer bits.
Or all about encoding the data.

- Encoding is related to occurance of the characters, frequently occuring Characters will Have a Shorter Path as Compared to less occuring Characters. This is done to reduce the No. of bits for frequent Characters & ensure Max. data Compression.