Chapter :- 6

Tree.

Non-Linear ~~data structure~~

Tree is a non-linear data structure consisting of data nodes connected to each other with a pointer in a same way as like linked list. Each node in a tree may be connected to two or ~~more~~ more nodes. The maximum number of nodes to which any single node is connected is called order of the tree.

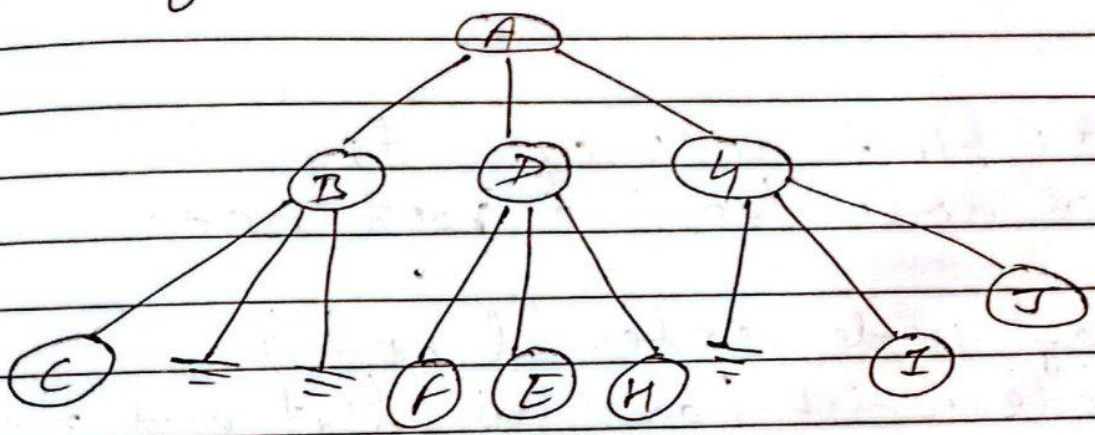
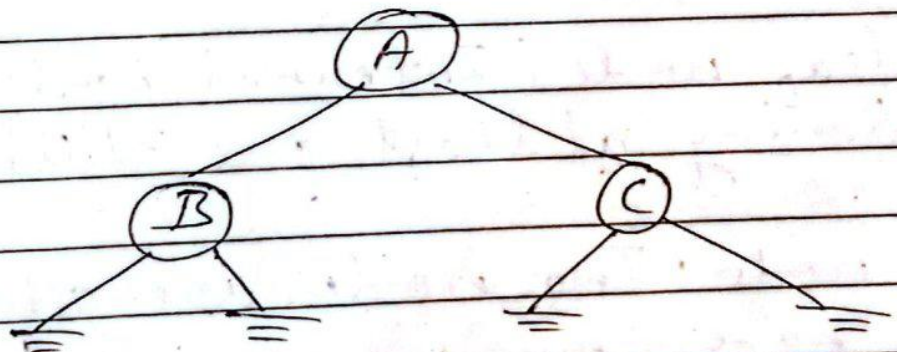
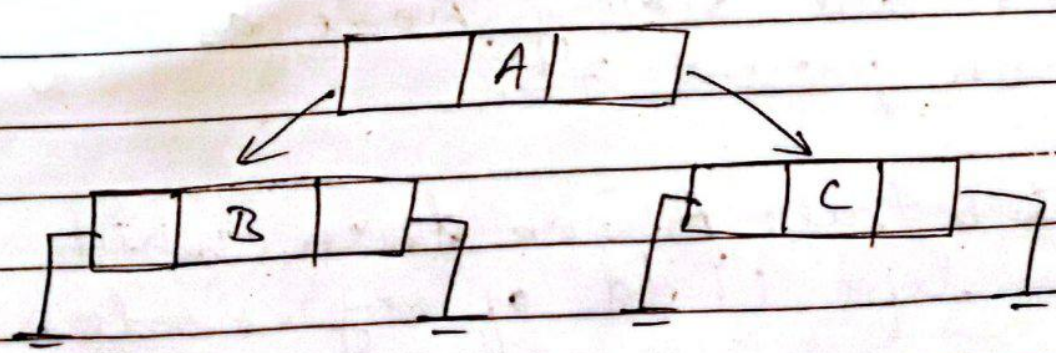
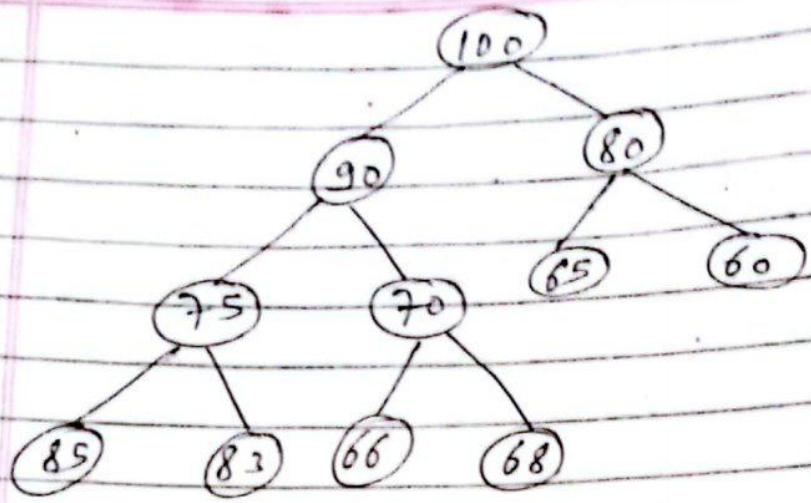


fig:- Tree of order 3







1) Root :- First node of a tree
-> here node 100 is a root node

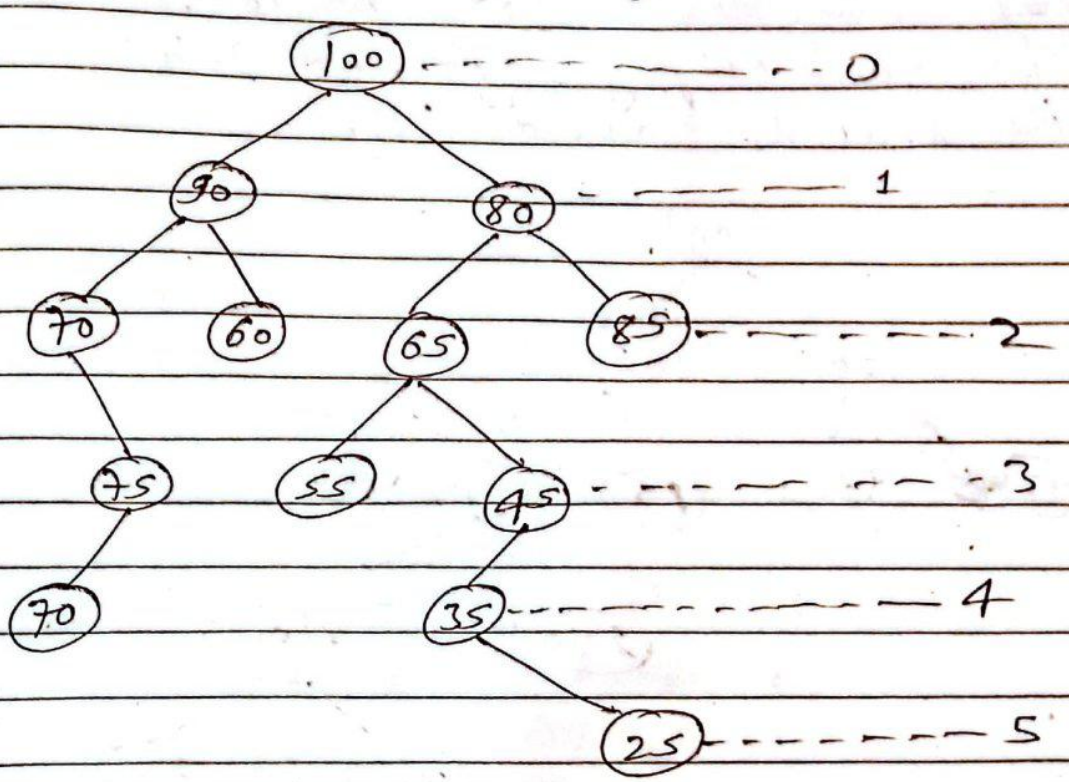
2. leaf node (external node)
-> Node which have no left and right child here node 85, 83, 66, 68, 65 and 60 are leaf node.

3. Non-leaf node (Internal node)
-> Node having at least one child

4. Sibling node : Two nodes are sibling nodes
-> if they have common parent here node 85, 83 are sibling node as they have common parent 75.

5. left sub tree: A sub tree which belongs to the left child of any node.

6. Right sub tree : A sub tree which belongs to the right child of any node.

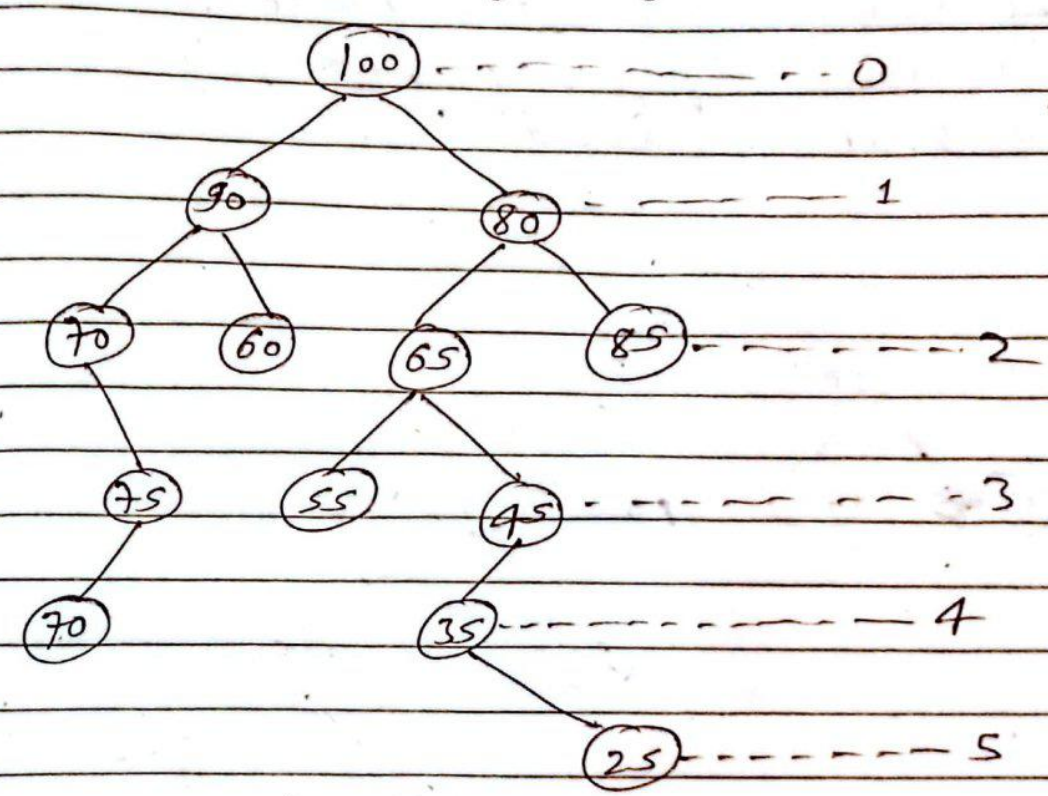


1. Depth : Depth of a tree is a maximum
of edges / nodes from root to leaf. He
depth is 5

2. Height : Height of a node ~~is~~ in a bin
tree is the number of edges in the lo
path from the node of interest to le
Height of 65 is 3.

3. Level
-> Level of root node is 0 and othe
have 1 level more than parent
or ~~predecessor~~ node.
predecessor

6. Right sub tree : A sub tree which belongs to the right child of any node



1. Depth : Depth of a tree is a maximum no of edges / nodes from root to leaf. Hence depth is 5

2. Height : Height of a node ~~in~~ in a binary tree is the number of edges in the longe path from the node of interest to leaf. Height of 65 is 3.

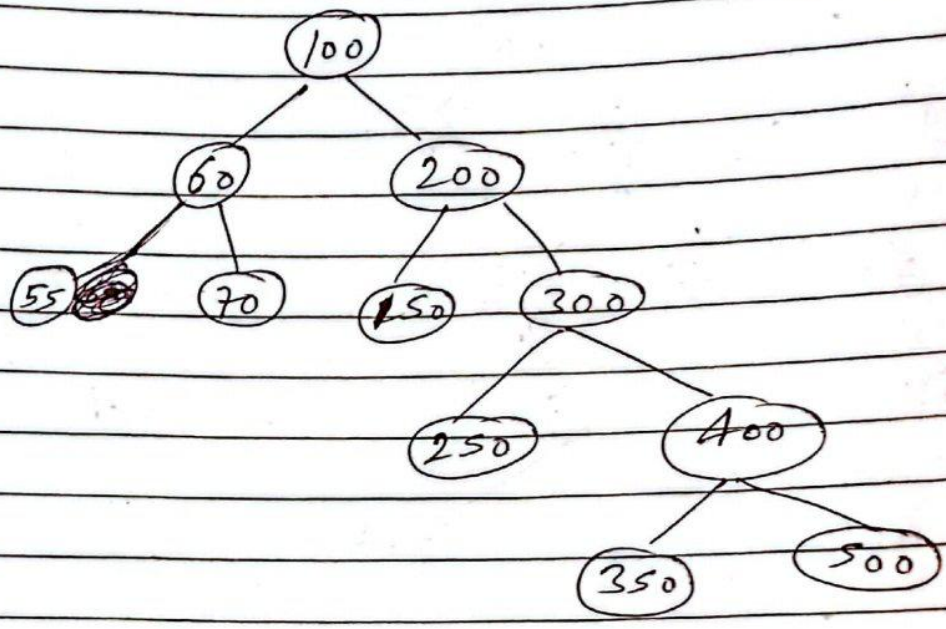
3. Level

-> Level of root node is 0 and other no have 1 level more than parent node or ~~predessor~~ predecessor node.

*Types of Binary Tree

1) Strictly Binary Tree (SBT)

-> If every non-leaf node in a binary tree has non-empty left and right sub tree, the tree is called S.B.T.



2) Completely Binary Tree (CBT)

-> A Strictly Binary tree is said to be completely binary tree if all the leaf nodes are in same level.

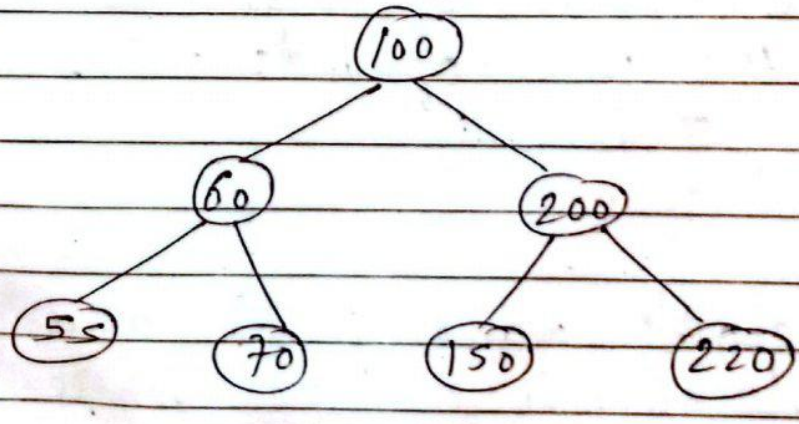


fig :- CBT

V -> Root
L -> Left
R -> right

VLR or VRL (same

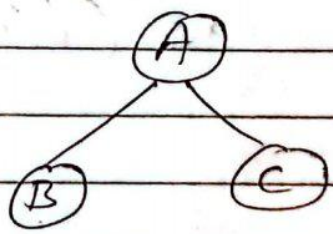
*Traversing in Binary Tree

Visiting all nodes present in a binary tree is called traversing. There are 3 types of traversing

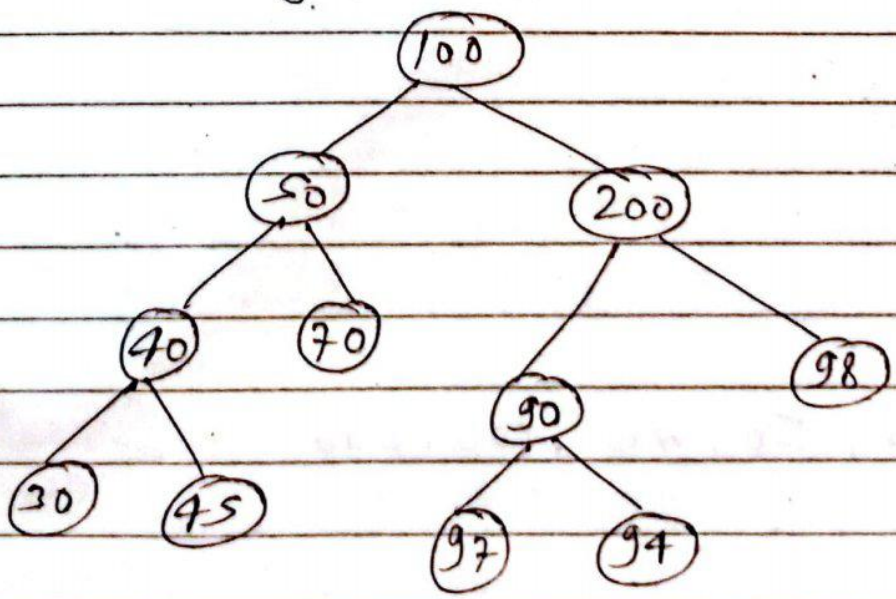
1) Pre-order Traversing (VLR) : A, B, C
-> Root node is visited first then visit its left and right child respectively.

2) Post-order (LRV) : B, C, A
-> Visit left child, right child and then v root node at last.

3) In-order (LVR) : B, A, C
-> Root node is visited in between left right child.

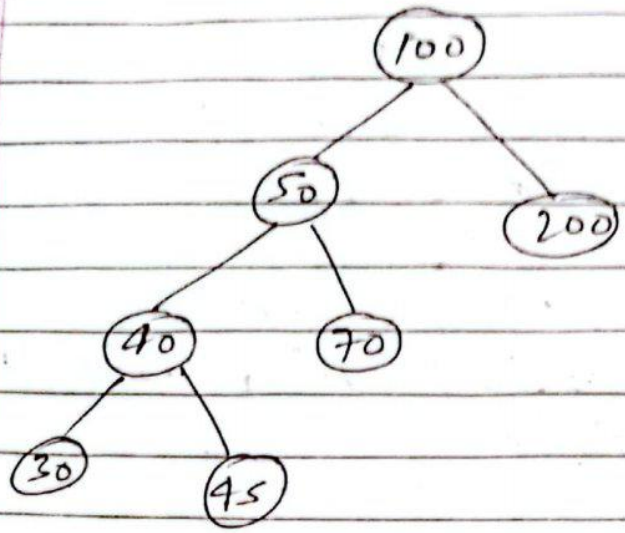


* Traversing / Pre-order traversal / (VL



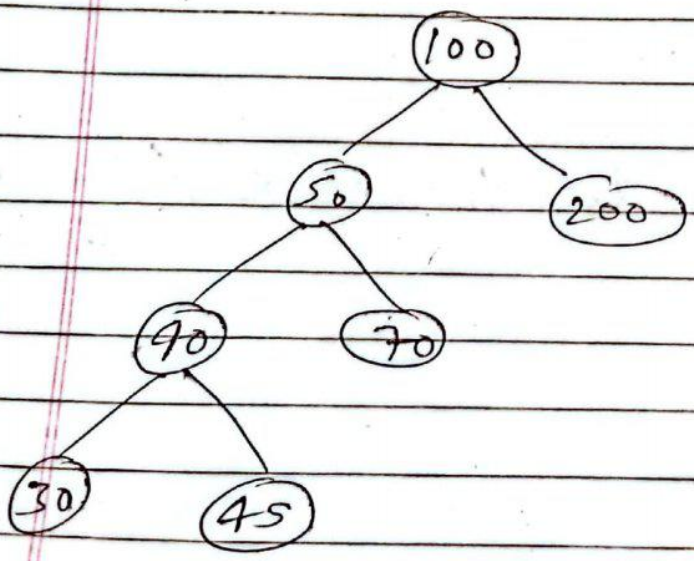
100, 50, 40, 30, 45, 70, 200, 90, 97, 97,
V L R

* Post-order traversal (L R V)

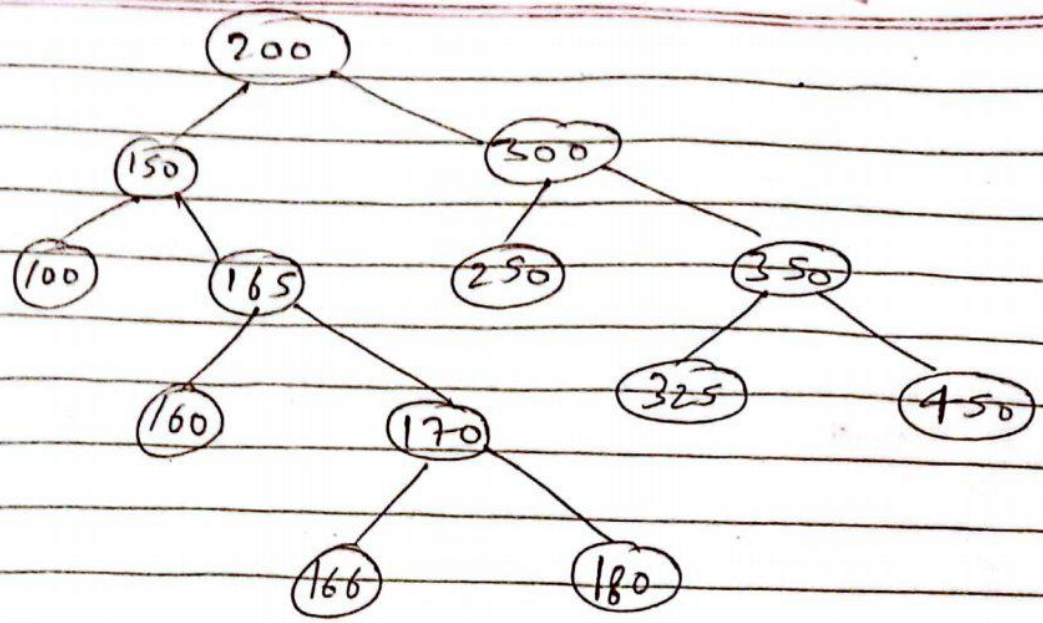


30, 45, 40, 70, 50, 200, 100

* In-order traversal (L V R)



30, 40, 45, 50, 70, 100, 200



Pre -> 200, 150, 100, 160, 166, 180, 170, 165, 300, 250, 325, 450, 350

Post -> ~~166, 160~~ 166, 180, 170, 160, 165, 160, 150, 325, 450, 350, 250, 300, 200

In -> 166, 170, 180, 160, 165, 100, 150, 325, ~~450~~, 350, ~~250~~, 300, 200, 450,

V

Pre -> (V-L-R): 200, 150, 100, 165, 160, ~~160~~, 170, 166, 180, 300, 250, 350, 325, 450

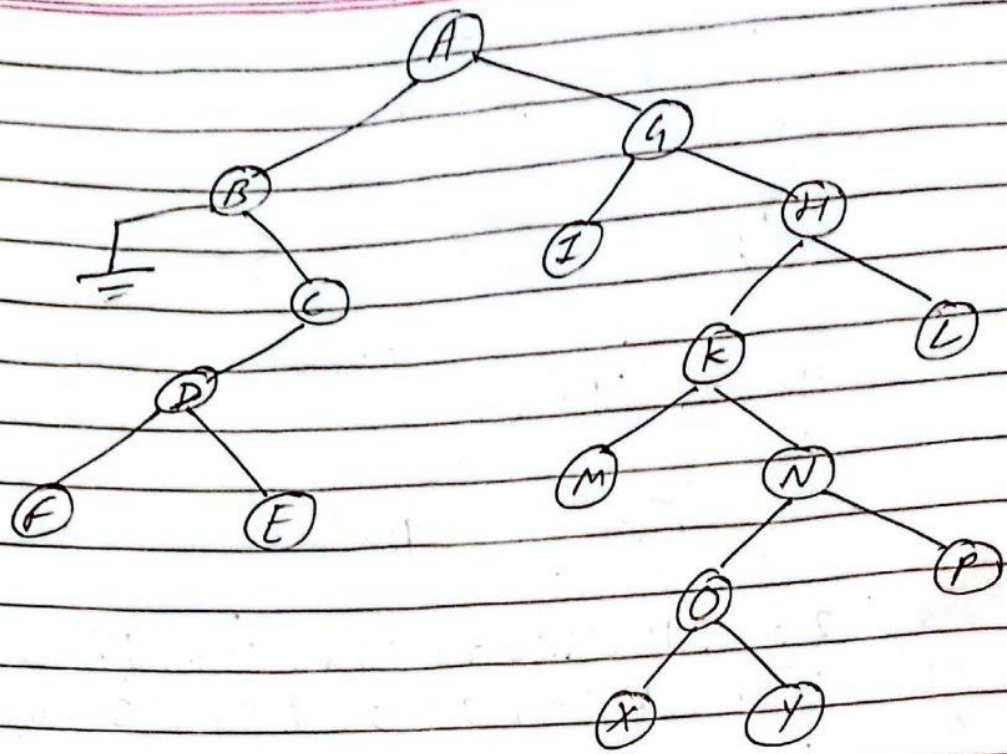
L R

post-order (L-R-V): 100, 160, 166, 180, 170, 165, 150, 250, 325, 450, 350, 300, 200

L R V

in-order (L-V-R): 100, 150, 160, 165, 166, 170, 180, 200, 250, 300, 325, 350, 450

L V R



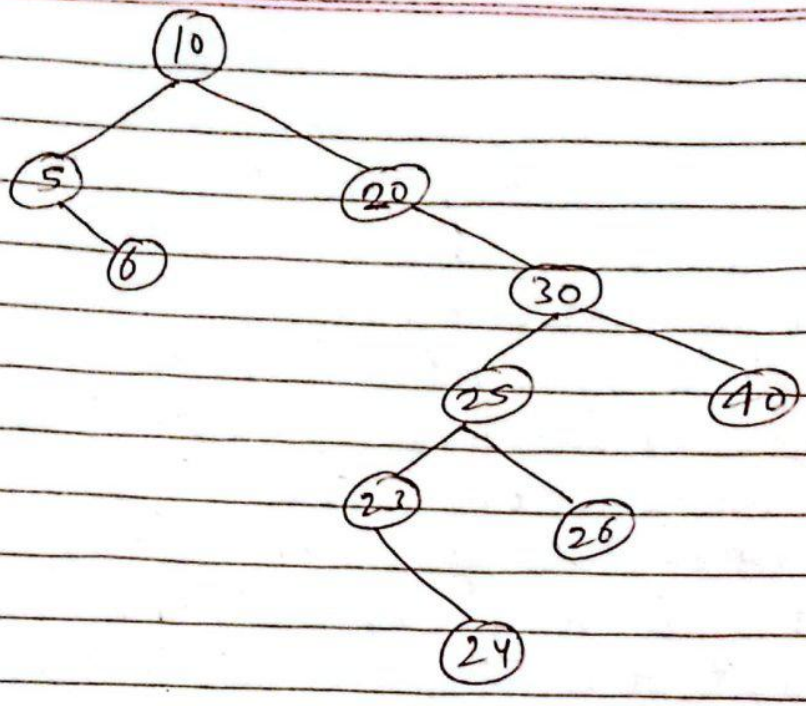
Pre-order (V-L-R): A, B, C, D, F, E, G, I, H, K, M, N, O, X, Y, P, L

Post-order (L-R-V): F, E, D, C, B, I, M, X, Y, O, P, N, K, L, H, G, A

(L-V-R) in-order: ~~F, D, E, C, B, A, I, G, M~~
~~B, F, D, E, C, A, I, G, M, K, X, O, Y, N, P, H~~
~~B, F, E, D, C~~

B, F, D, E, C, A, I, G, M, K, X, O, Y, N
L ✓ R

P, H, L



pre (V-L-R) : 10, 5, 6, 20, ~~30, 25,~~ 23, 24, 26,

post (L-R-V): 6, 5, ~~20~~, 24, 23, 26, 25, 40
30, 20, 10

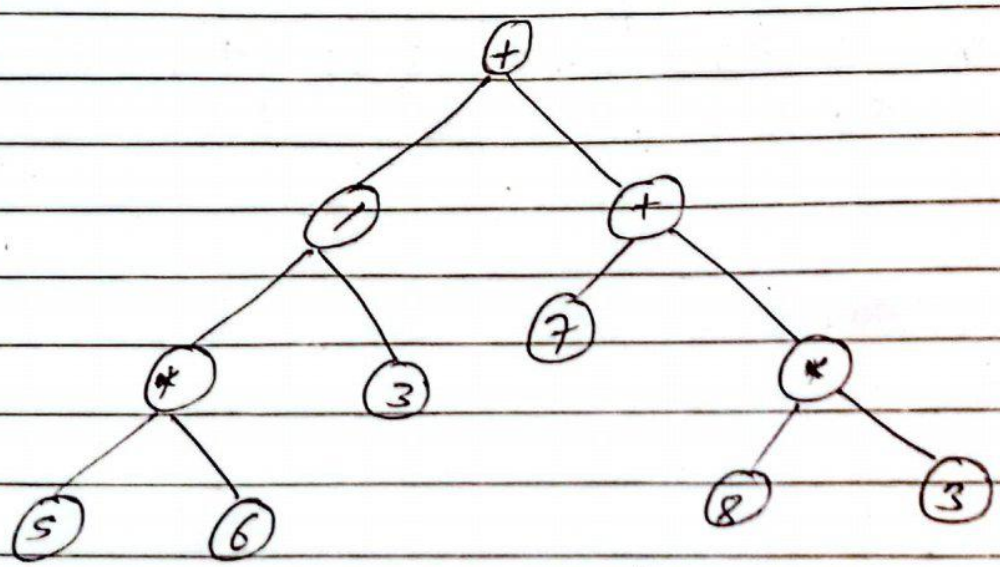
in (L-V-R): 5, 6, ~~10~~, 20, 23, 24, 25, 26,
40

* Binary expression Tree

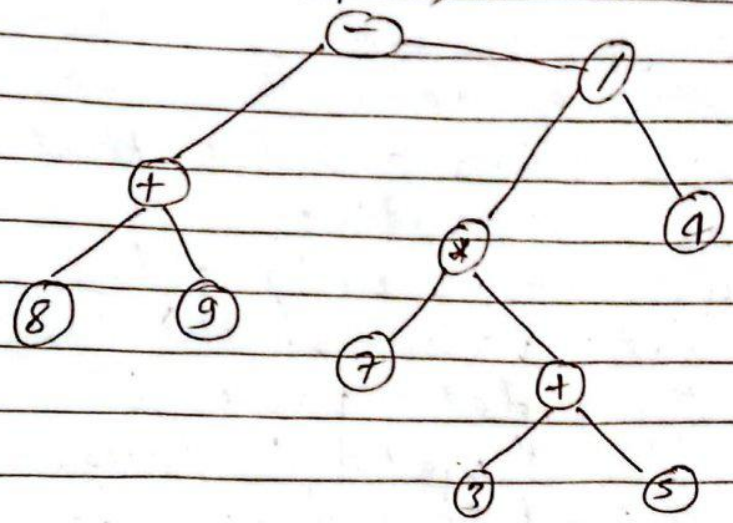
→ Expression tree are used to represent arithmetic expression especially the combinations of operators, operands and order of evaluation. Binary expression tree is a special kind of binary tree having following properties

1) Each leaf node contains a single operand
2) Each ~~node~~ ~~leaf~~ non-leaf node contains a single binary operator.
3) The left and right subtrees of an operator node represents sub-expression that must be evaluated before appointing the operator at the ~~root~~ root of sub-tree.

eg:- (5 * 6/3 + (7 + 8 * 3))

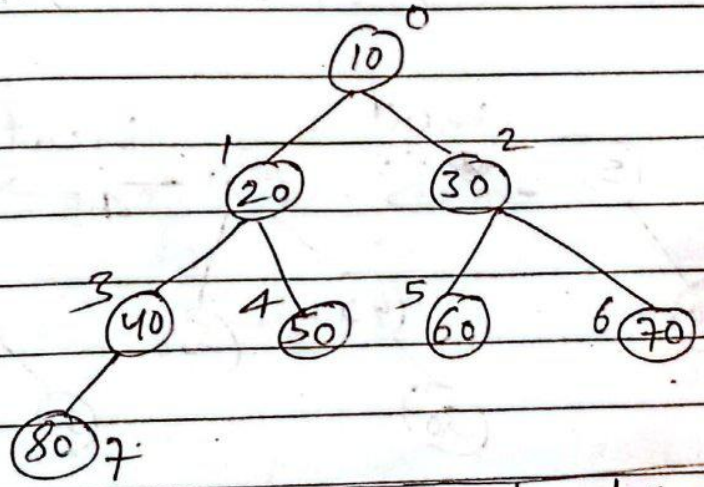


* 8 + 9 - (7 * (3 + 5) / 4)



* Construct a binary tree 10, 20, 30, 40, 50, 6
70, 80.

→



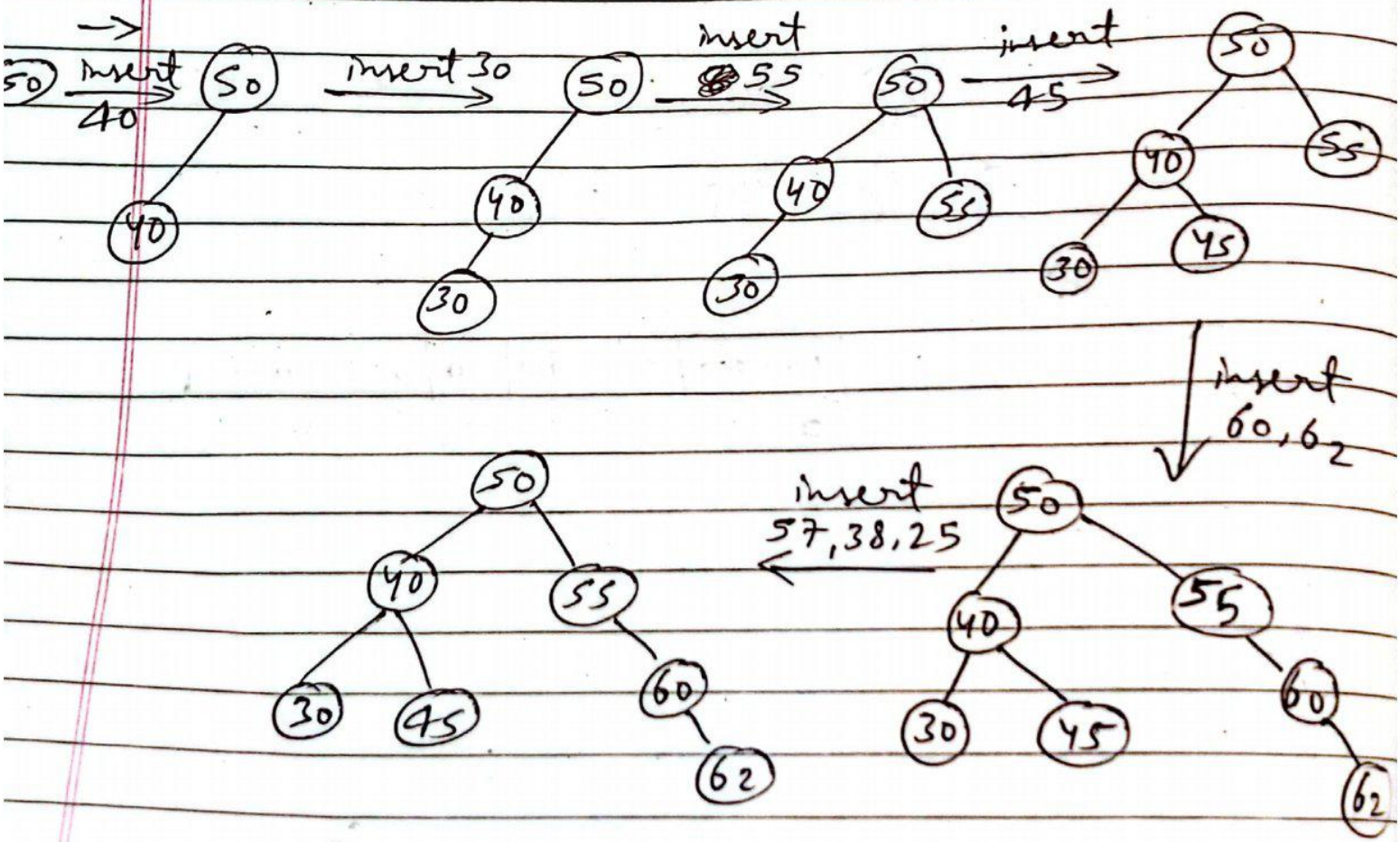
10	20	30	40	50	60	70	80
0	1	2	3	4	5	6	7

* Binary Search Tree (BST)

-> BST is a type of binary tree with special organization of data. BST is either empty or contains a value that satisfies following properties.

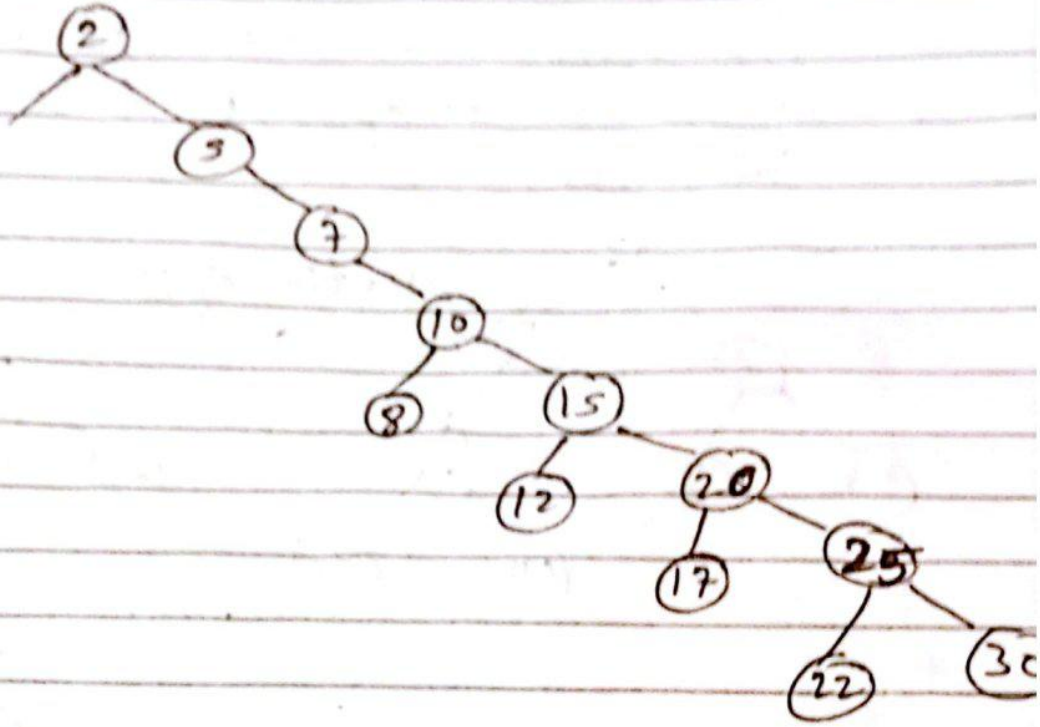
1) Every node has a unique key.
2) All the data value in the left sub-tree are smaller than data value in the root.
3) " " " " right " larger " " "
4) Left and right sub trees are also binary search tree.

* Create a BST :- 50, 40, 30, 55, 45, 60, 62, 57, 38, 25



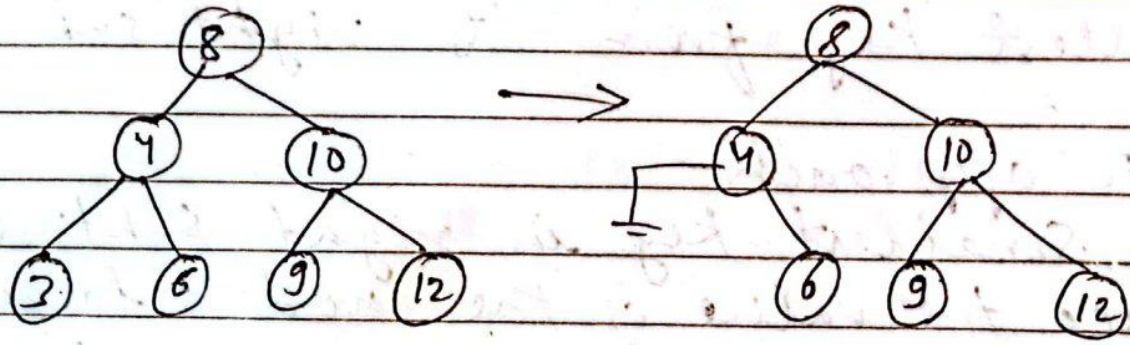
[1] [2]

* BST :- 2, 5, 7, 10, 8, 15, 20, 25, 12, 30, 19, 22

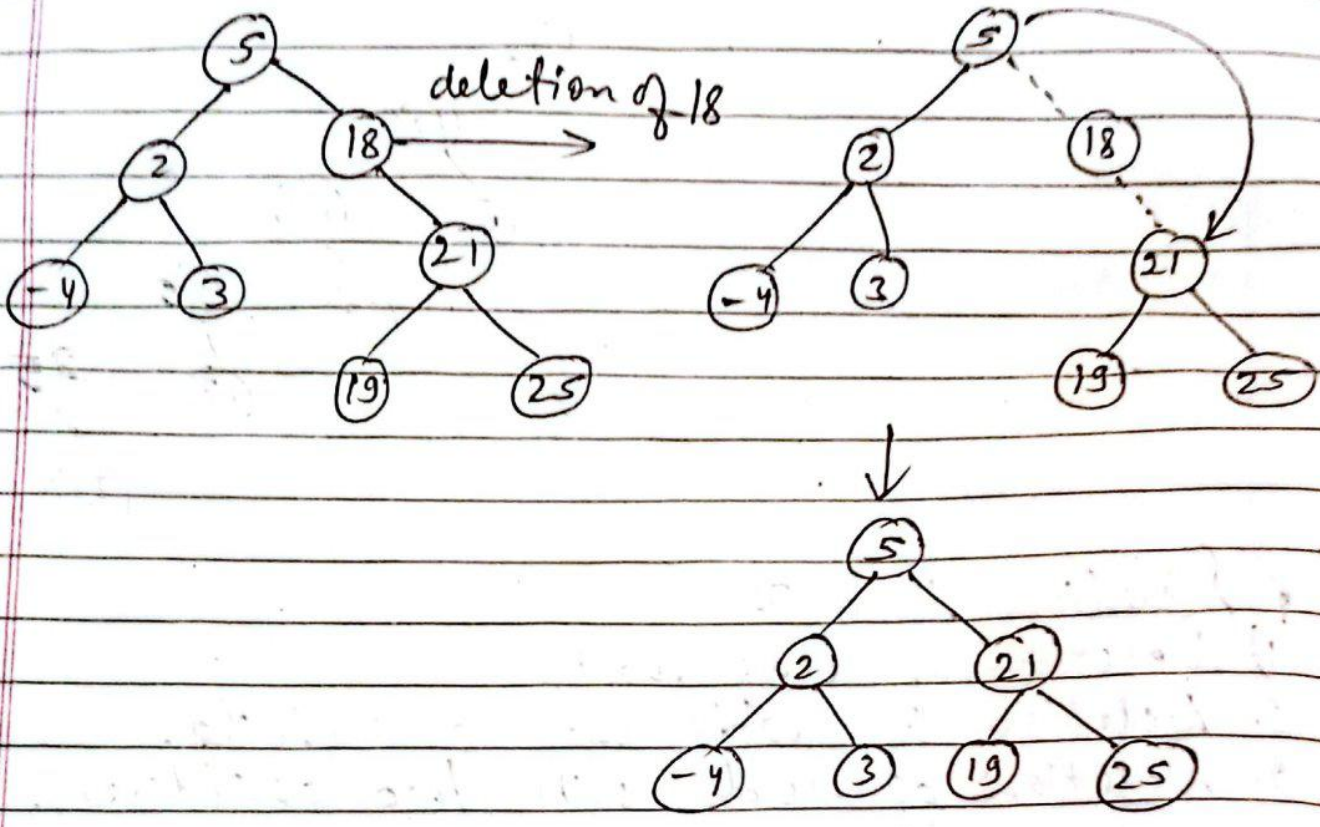


* Deletion in a BST
1) Deletion of leaf node
2. Deletion of node with one child.
3. Deletion of node with two children.

Case 1 :- Deletion of node with no child.
set corresponding left link of the parent node to NULL.



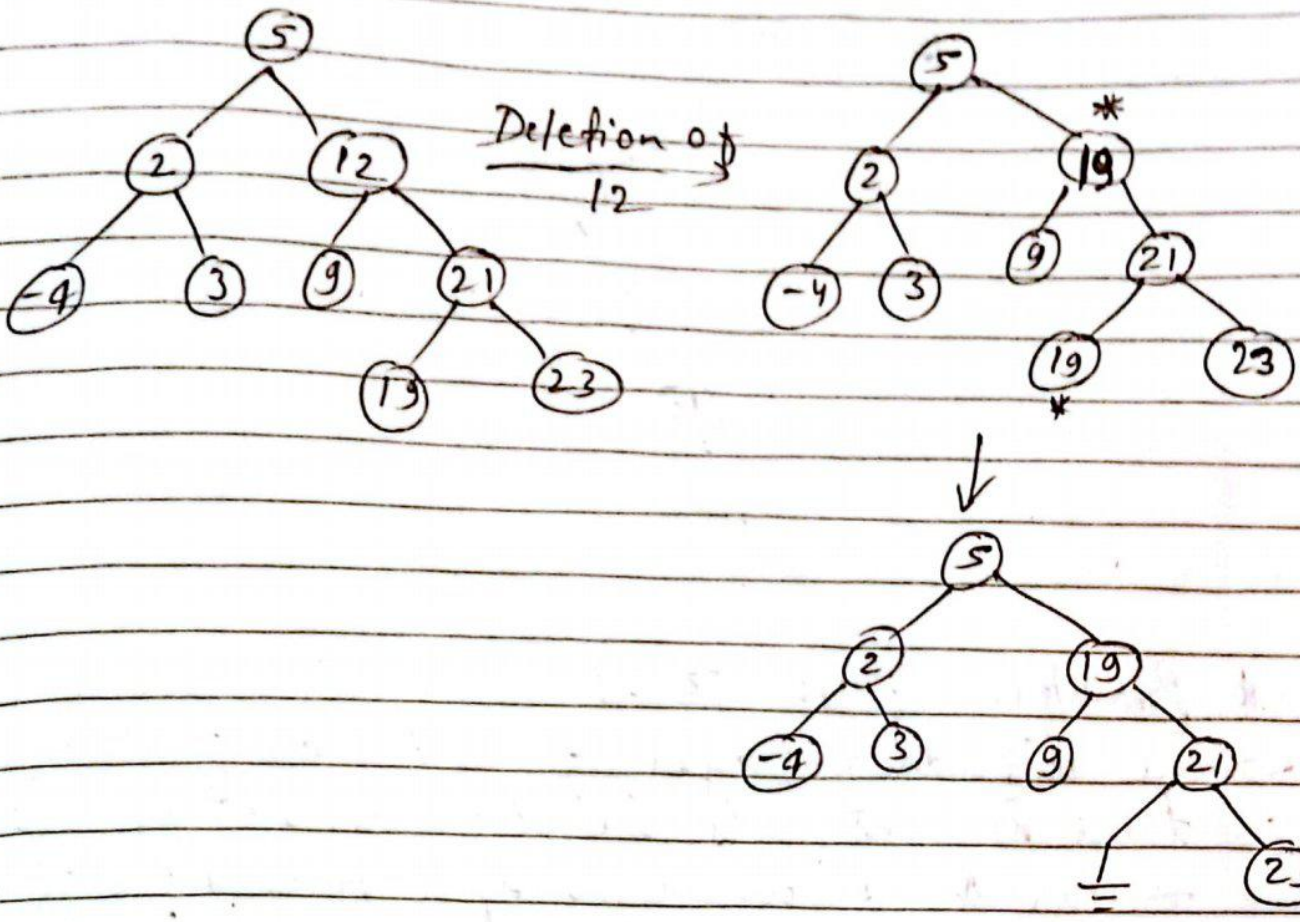
Case 2: Deletion of node with one child.
Cut the node to be removed from the tree, and then redirect the pointer from parent of the deleted node to its subtrees.



Case 3:- Node to be removed has two children.
Replace the node to be deleted with the smallest key from its right sub-tree.

* Basic approach
- Find smallest key in right subtree.
- Replace the value of the node to be removed with minimum. Now right subtree contains duplicate.

- Delete the minimum element from right sub tree.

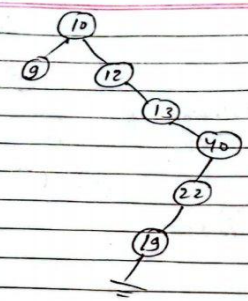


* Recursive algorithm :- Searching - BST (Root, item)

-> Input : Root is the first node of a tree & item is the key to be searched.
Output : Display message 'item found' or 'not found'.

```
1. If (root == NULL) Then
2.         return NULL
3. else if (root -> data == item)
4.         return item
5. else if (root -> data > item)
6. return Searching - BST (root -> left, item)
7. else
8. return Searching - BST (root -> right, item)
```

g:-



Problem with BST

→ The disadvantages of a Binary search tree is that its height can be as large as N-1 where N is a number of nodes. This means that the time required to perform insertion, deletion and many other operations can be O(N) in the worst case.

Example:- 25, 30, 35, 40, 45, 48, 50, 55

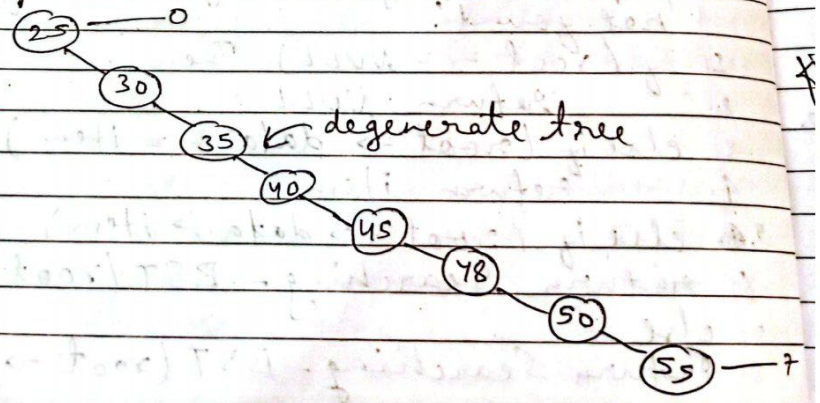


fig:- Height of tree is 7 having 8 nodes

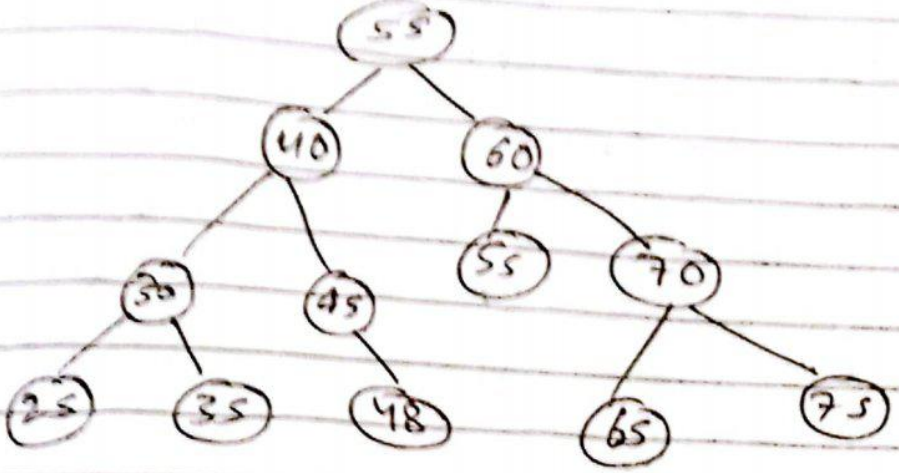


fig:- Height of tree is 3 for 12 nodes

->A binary search tree with N nodes has height at least $O(\log_2 N)$ in the best case and $O(N)$ in worst case. Thus our objective is to make height as small as possible i.e $O(\log_2 N)$ to increase the performance of searching, inserting & deleting & various other operations.

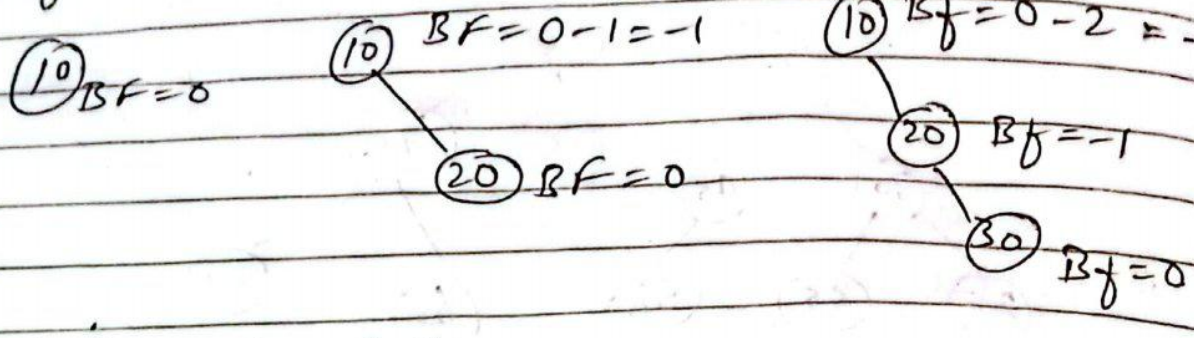
$O(\log_2 N) \rightarrow O(\log_2 8) = 3$

$O(N) \rightarrow O(8) = 8$

* AVL Tree : Height balanced BST

-> AVL tree is a height balanced binary search tree. AVL tree was introduced by Adelson-Velski and Landis. In an AVL tree every node maintains height, h height of left subtree and right subt

can differ by at most one.
Balance factor (BF) = height of left subtree - height of right subtree

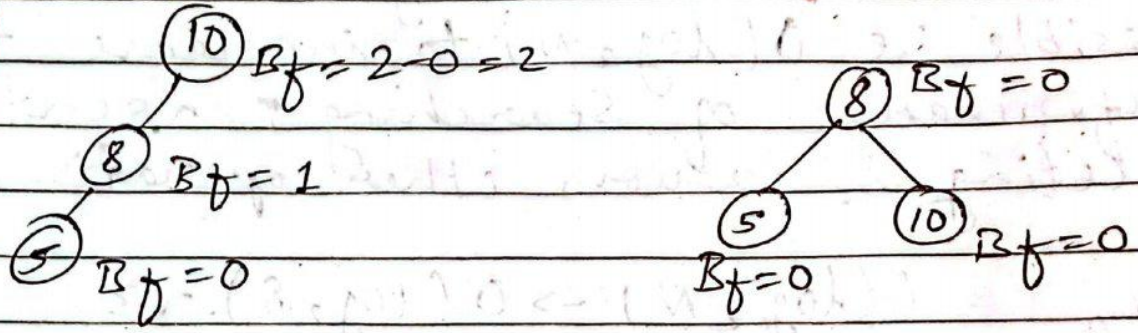


* Rotation in AVL

Case 1: If a node is inserted at the left subtree of left child of a root node.

eg, 10, 8, 5

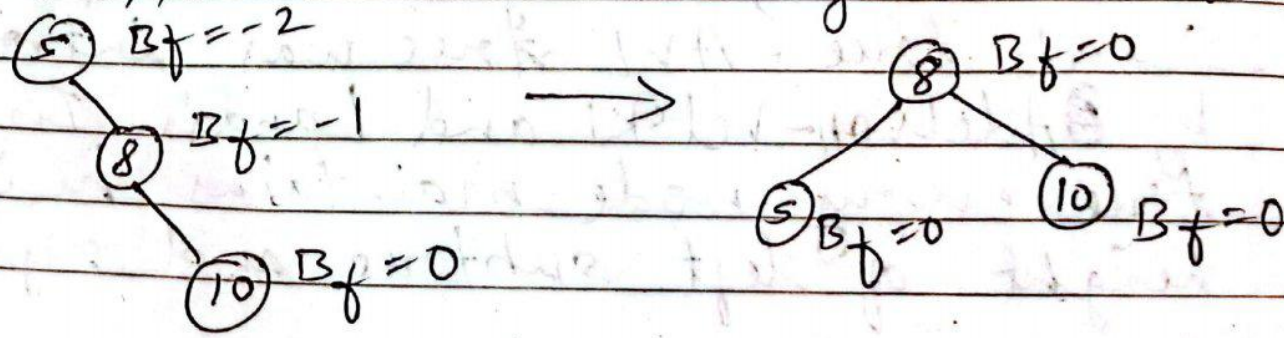
L-L -> Single R rotation



Case 2: If a node is added at the right subtree of right child of a root node

5, 8, 10

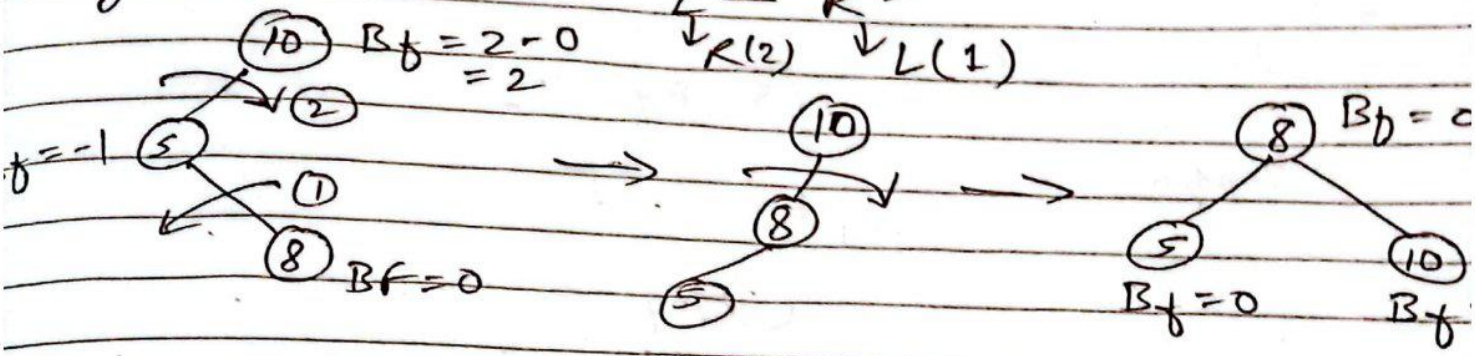
R-R -> Single L rotation



Case 3 :-

If a node is inserted at the right subtree of left child of root node.

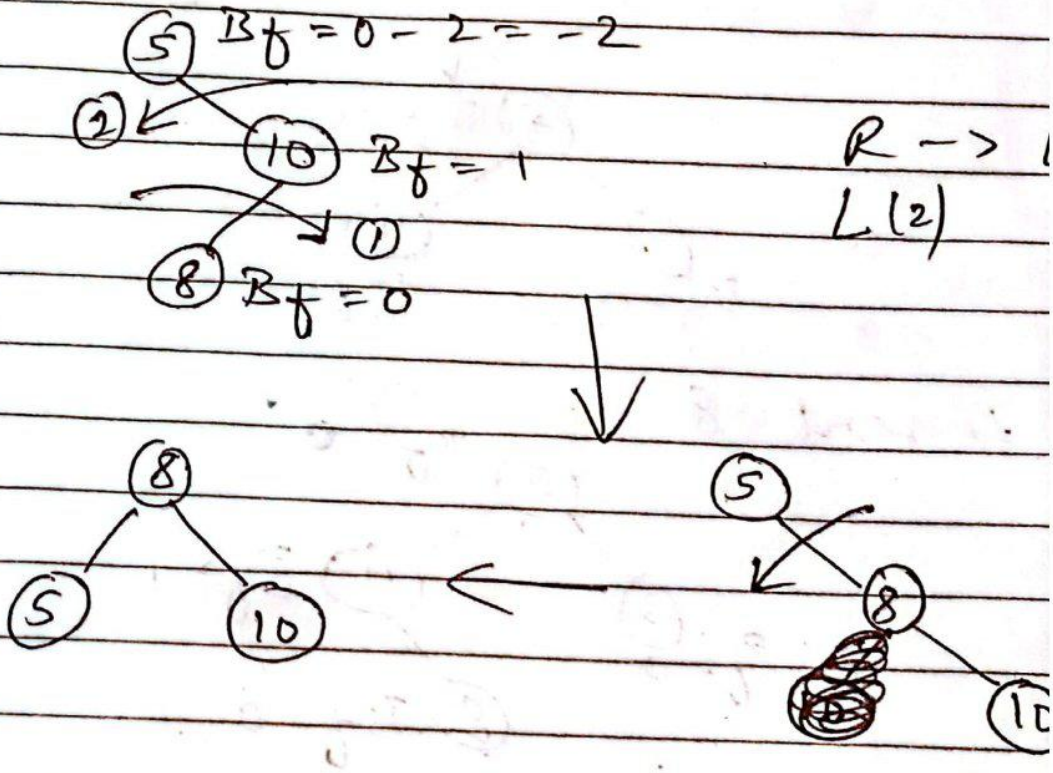
eg :- 10, 5, 8 L — R



Case 4 :

If a node is inserted at the left s subtree of right child of the root n

eg, 5, 10, 8

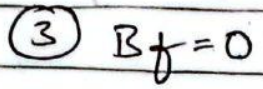


2011

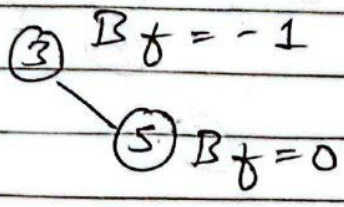
Construct an AVL tree from data given below.
3, 5, 11, 8, 4, 1, 12, 7, 2, 6, 10

->

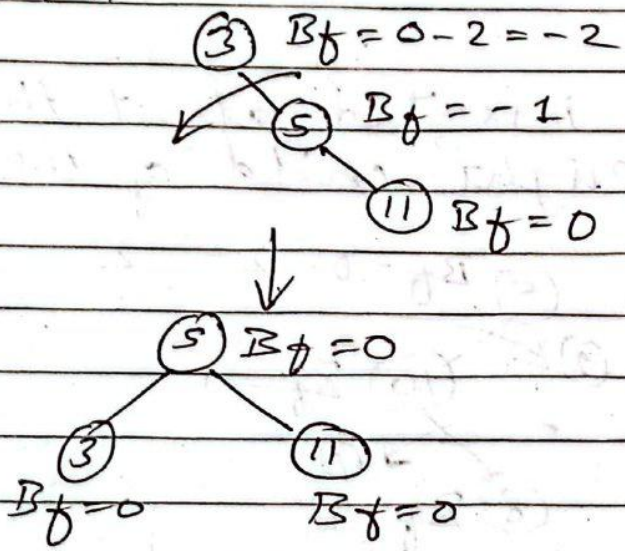
Insert 3



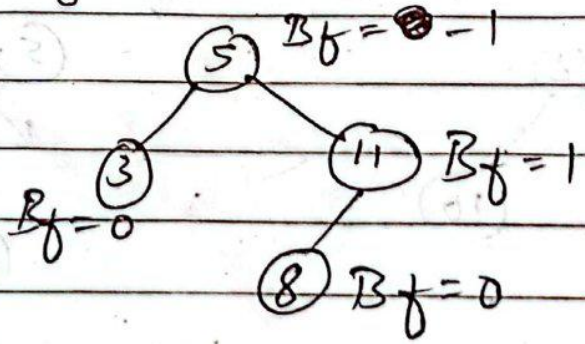
Insert 5



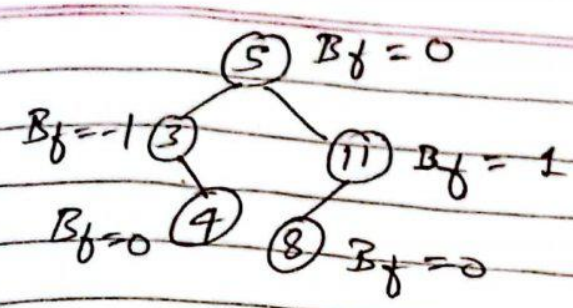
Insert 11



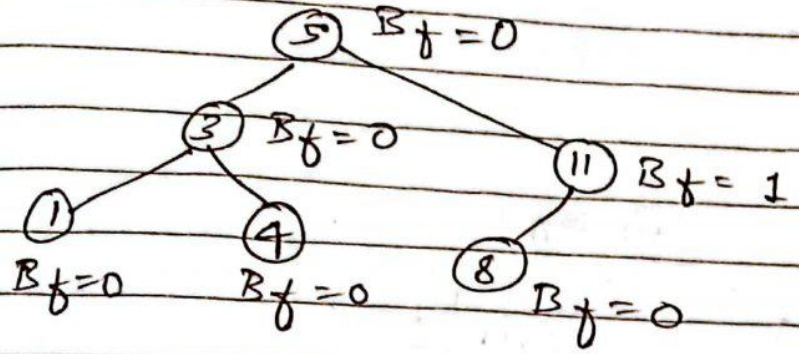
Insert 8



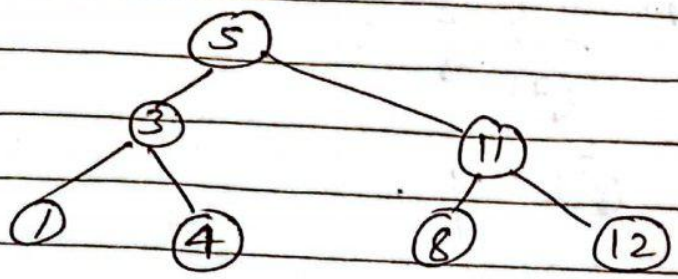
Insert 4



Insert 1



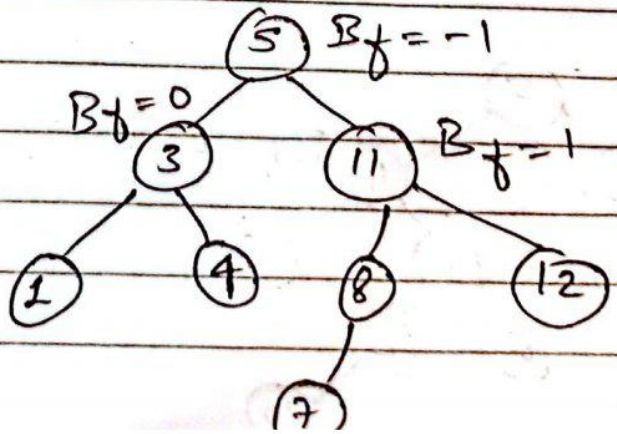
Insert 12



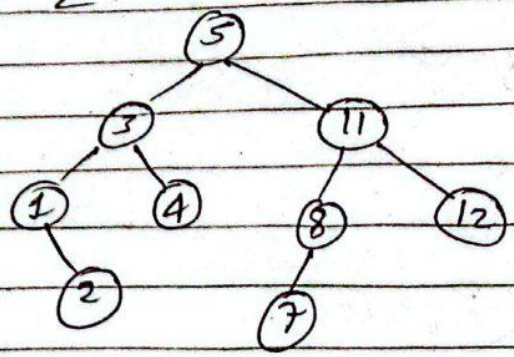
All
$B_f = 0$

Insert 7 Note:- Take two instant path fro imbalanced node to the current ly ad node, & so as to balance rotate in rev direction.

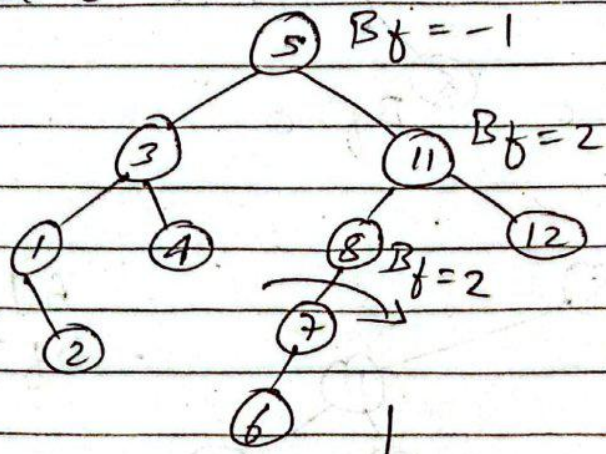
Insert 7

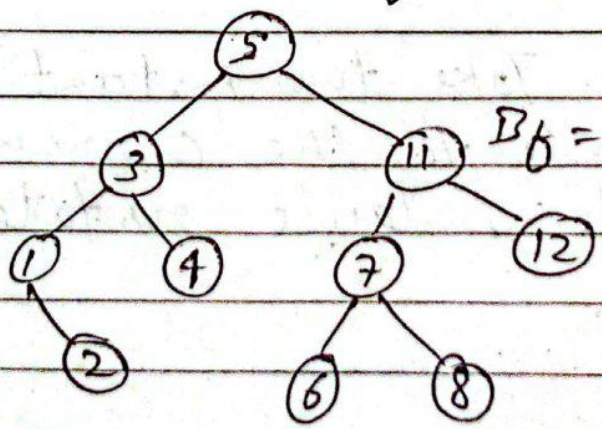


Insert 2

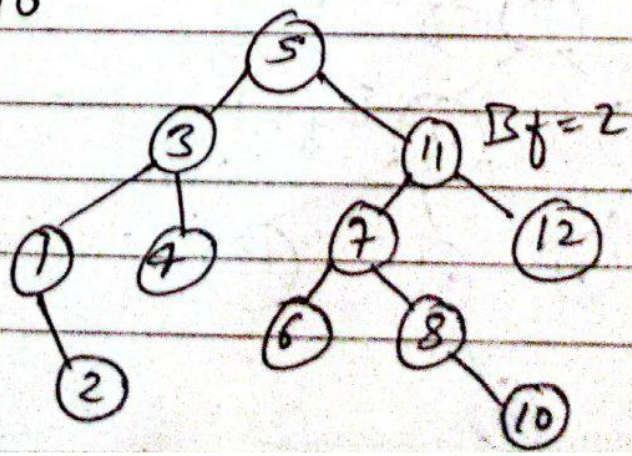


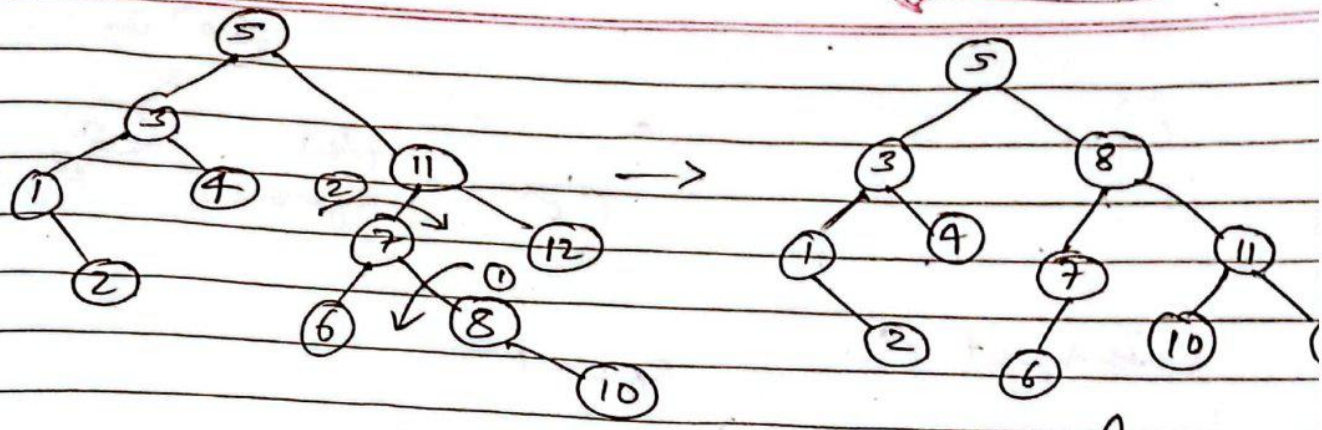
Insert 6





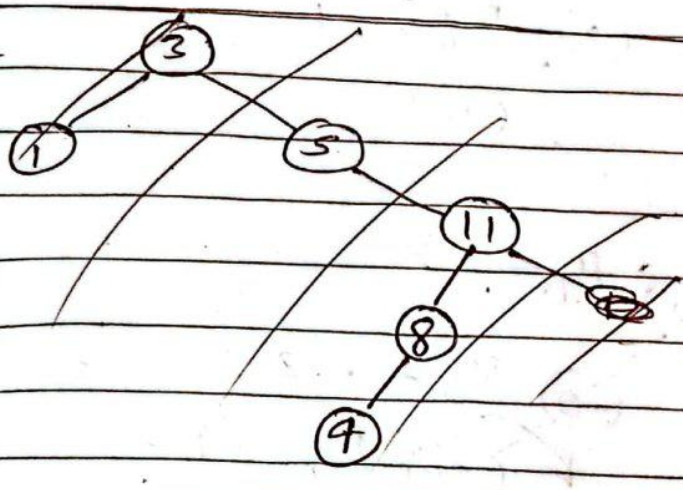
Insert 10





Ans.

practise..



* Create an AVL tree for following va
10, 20, 30, 40, 50, 60, 70

1) Insert 10

(10) Bf = 0

2) Insert 20

(10) Bf = -1

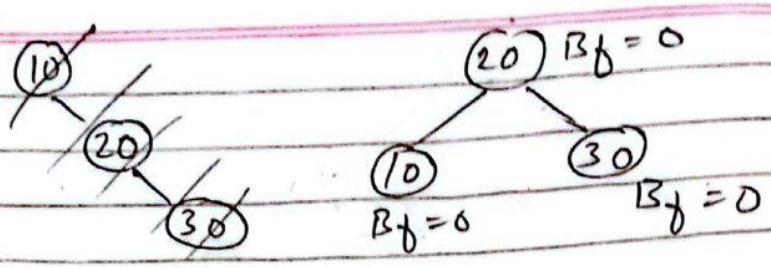
(20) Bf = 0

3) Insert 30.

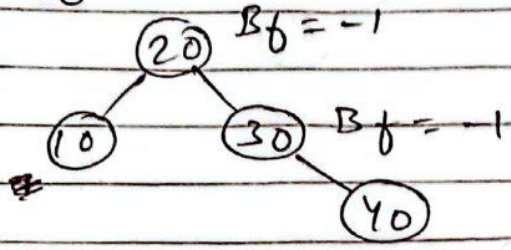
(10) Bf = -2

(20) bf = -1

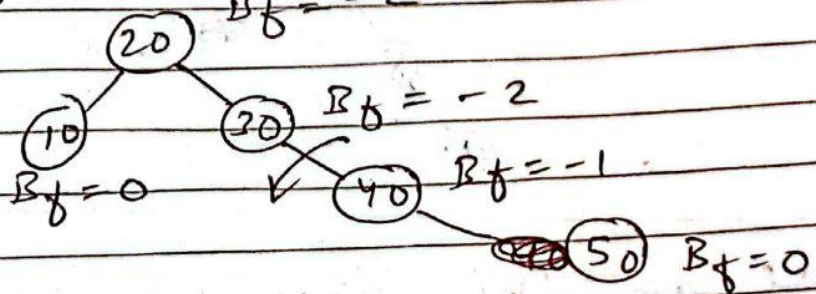
(30) Bf = 0



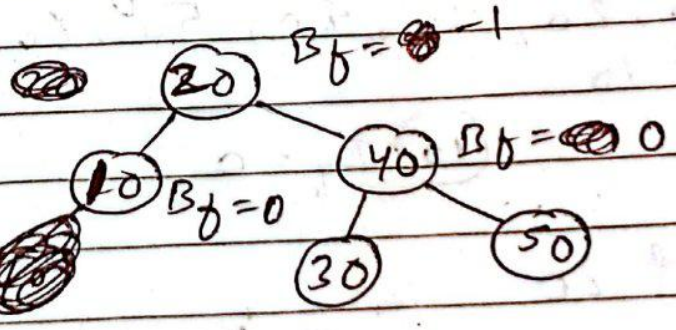
Insert 40



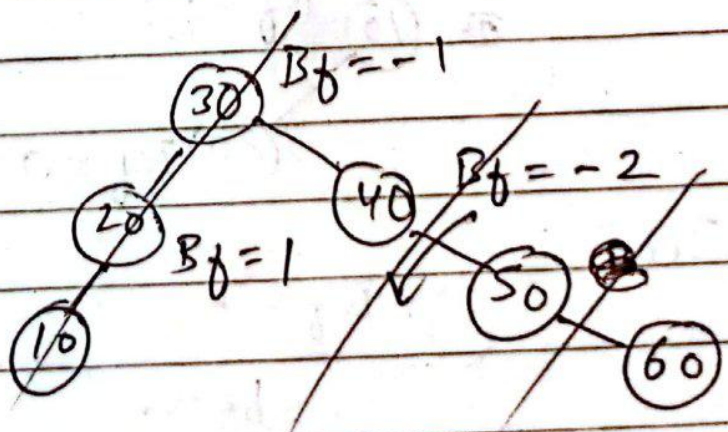
Insert 50

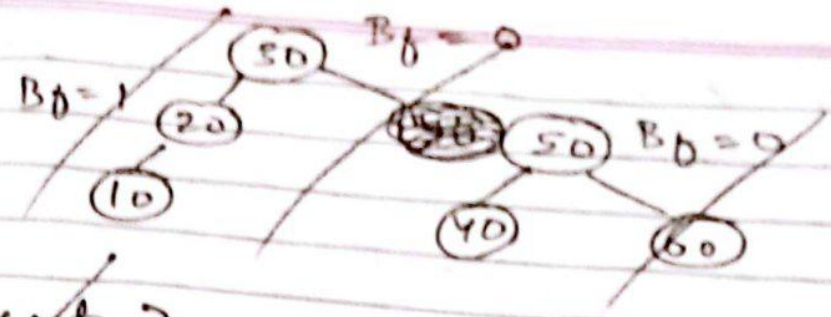


Insert

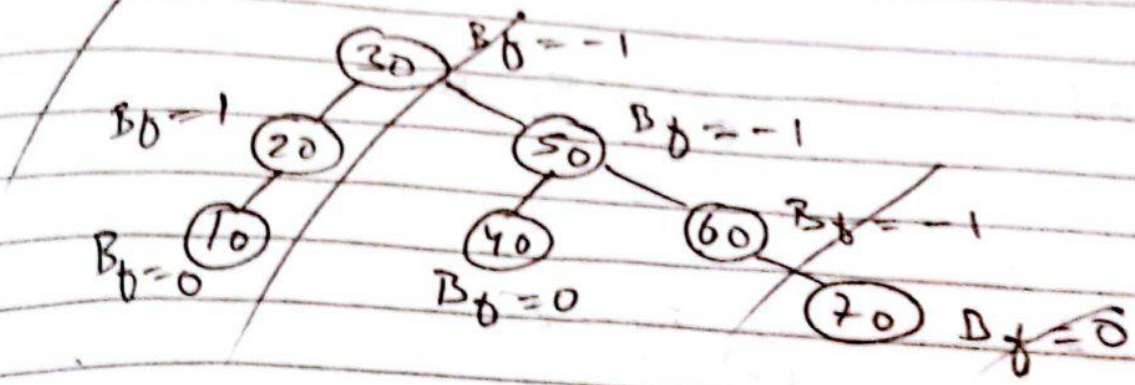


Insert 60

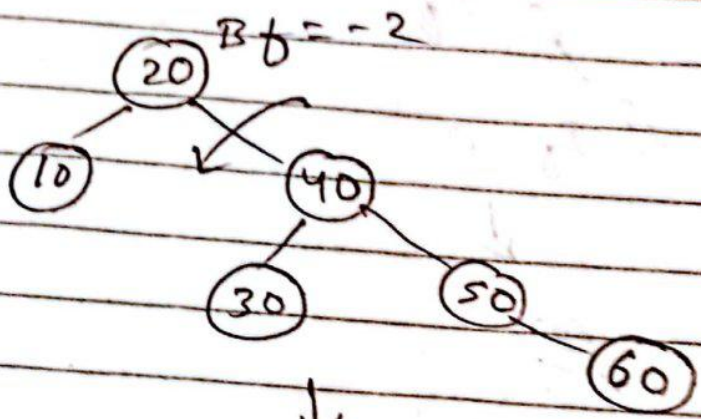




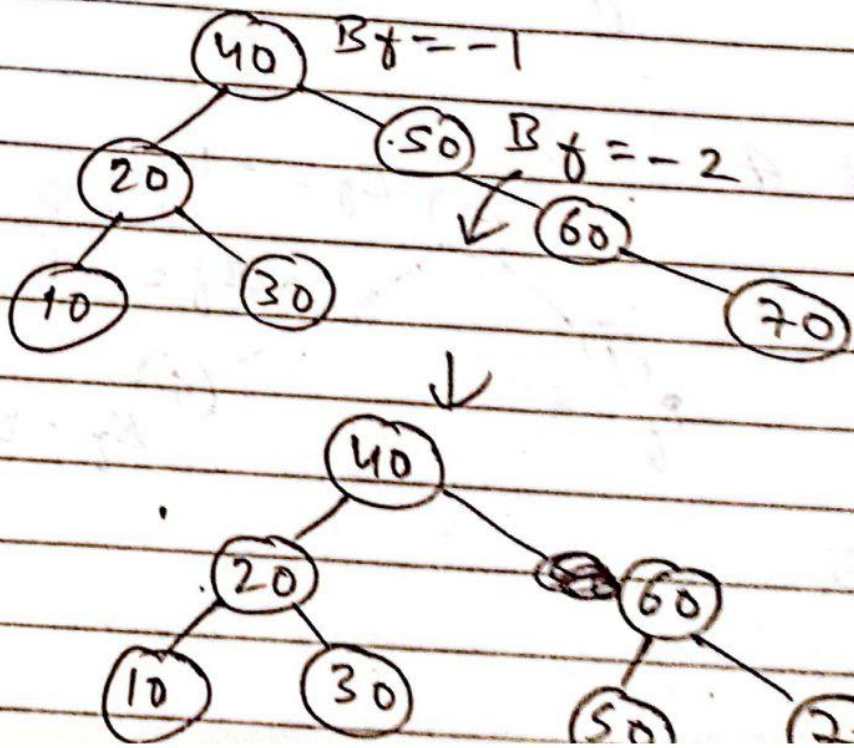
Insert 70



Insert 60

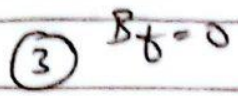


Insert 70

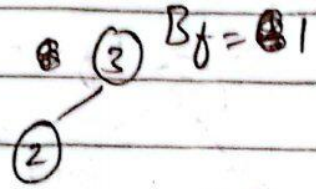


Create an AVL tree :- 3, 2, 1, 4, 5, 6, 16, 15, 14, 13, 12, 11

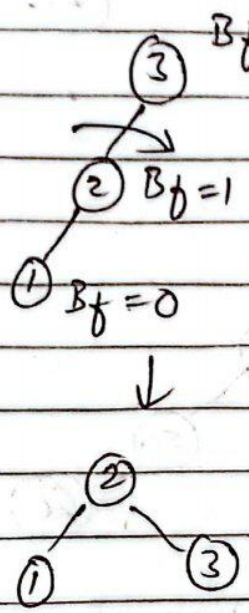
Insert 3



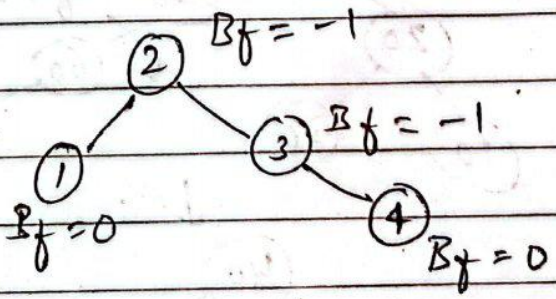
Insert 2



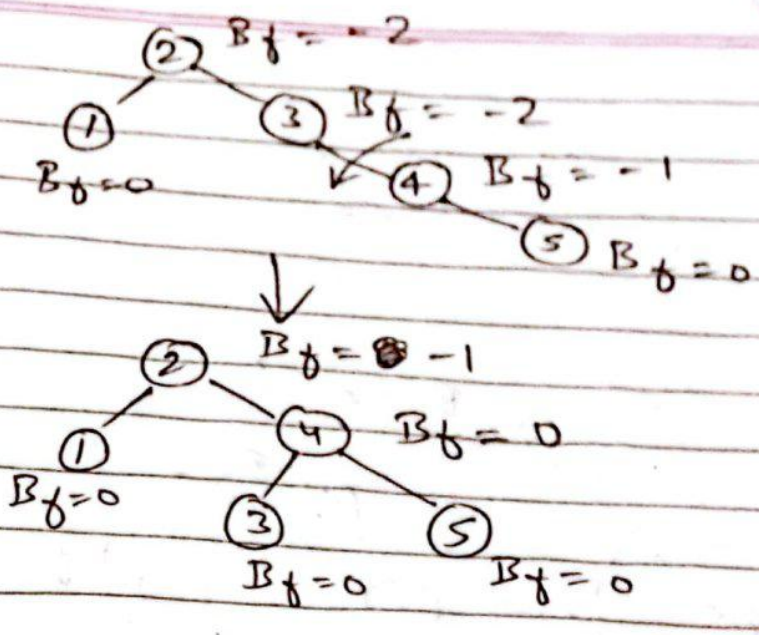
Insert 1



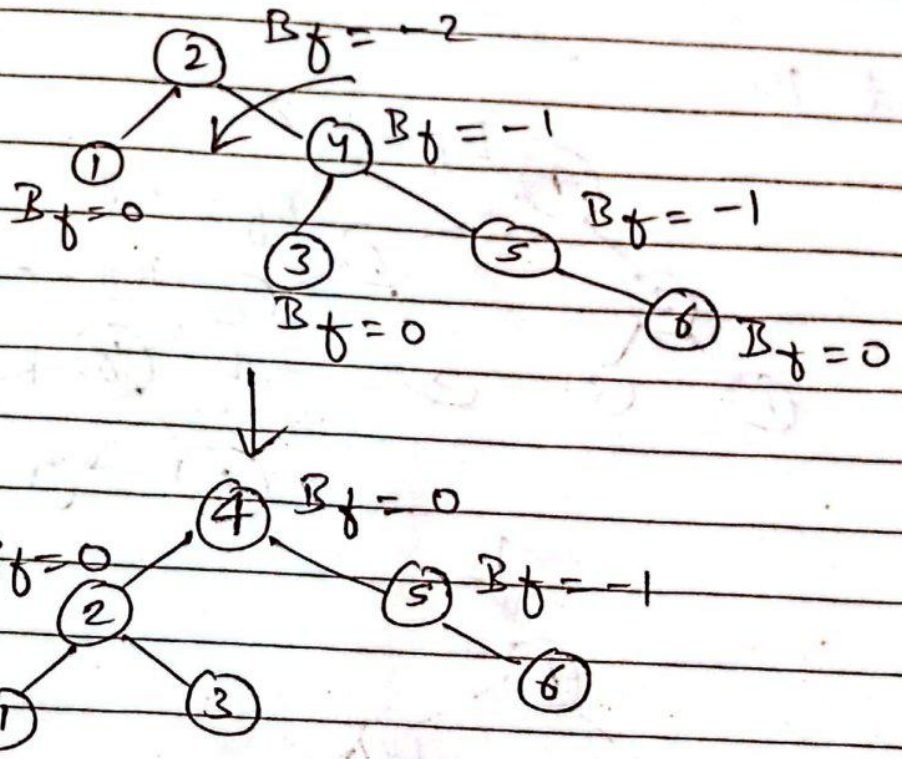
Insert 4



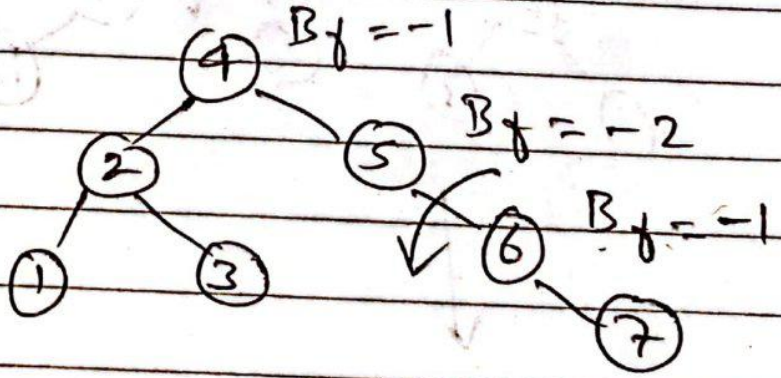
Insert 5

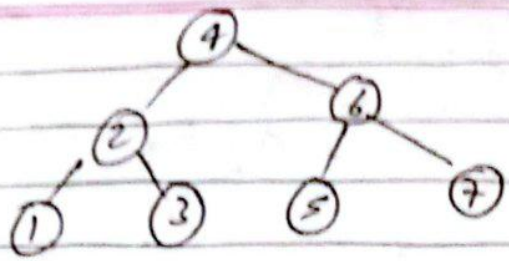


Insert 6

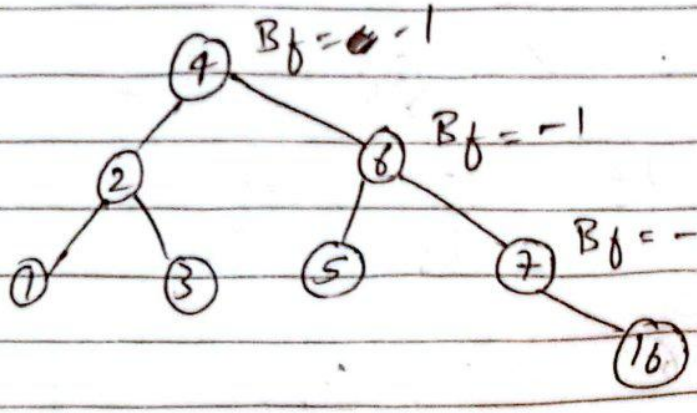


Insert 7

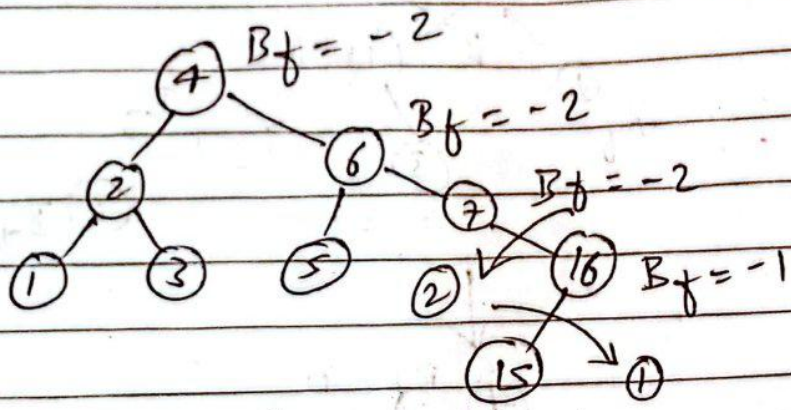


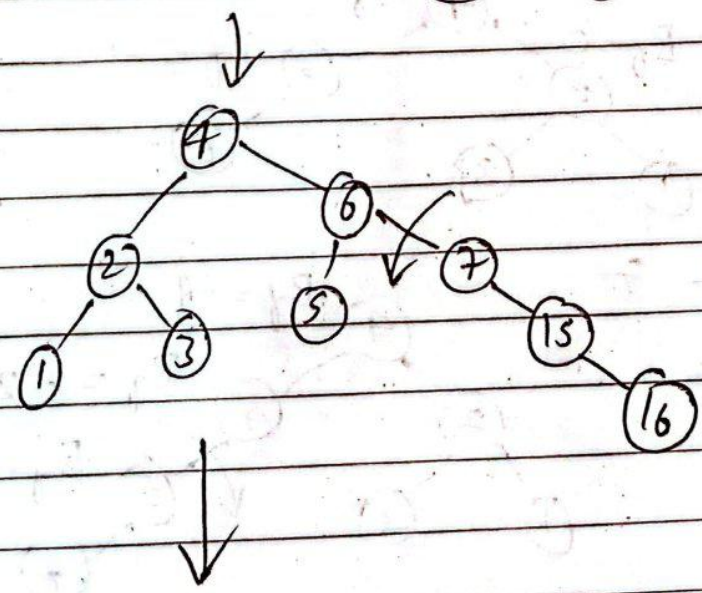


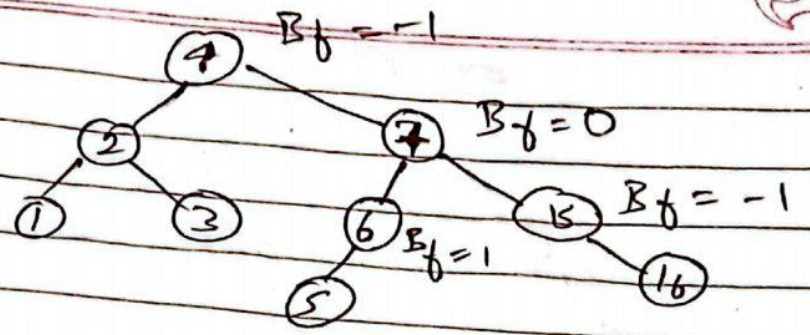
Insert 16



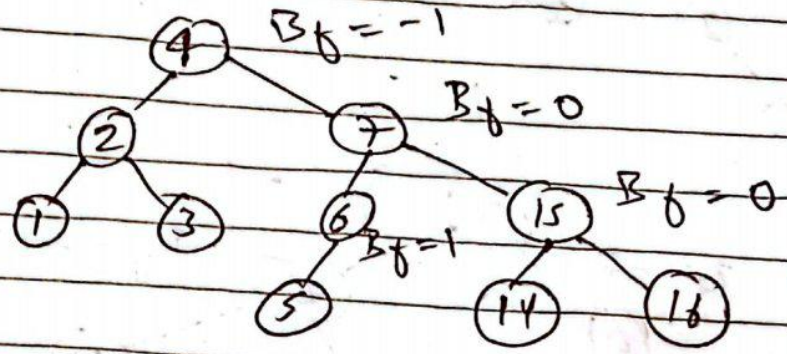
Insert 15



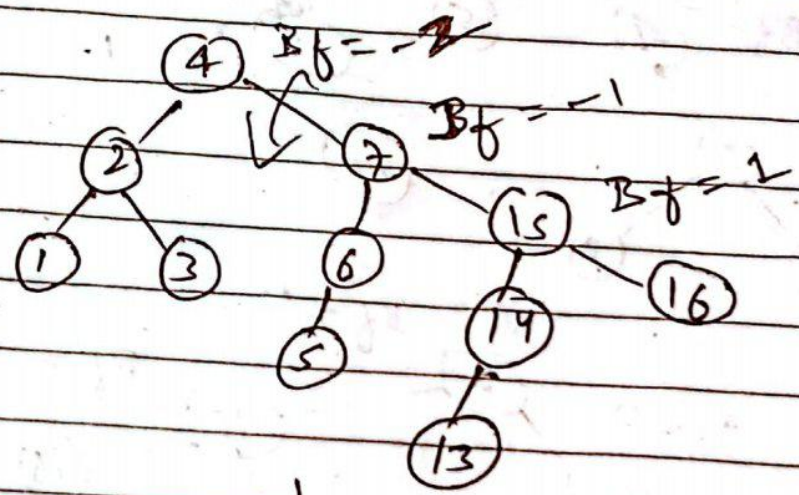


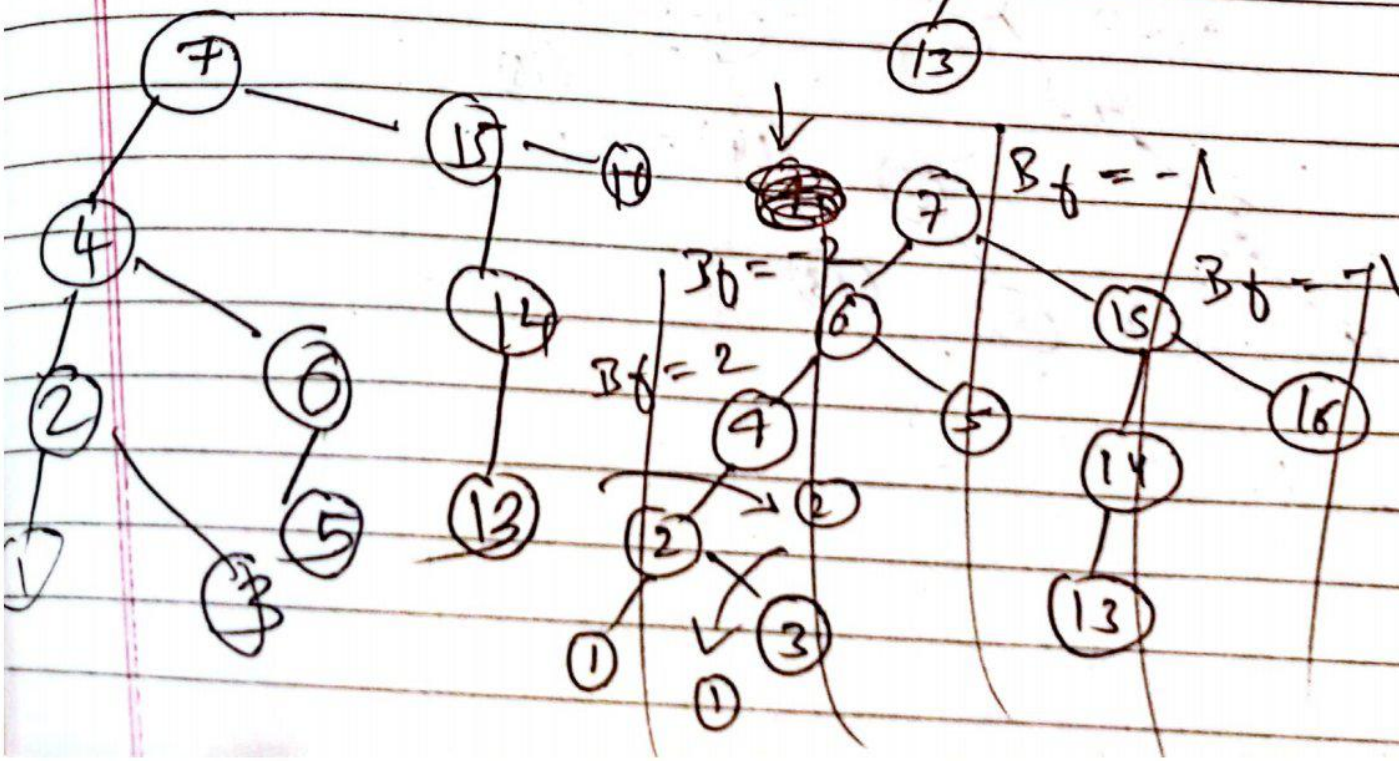


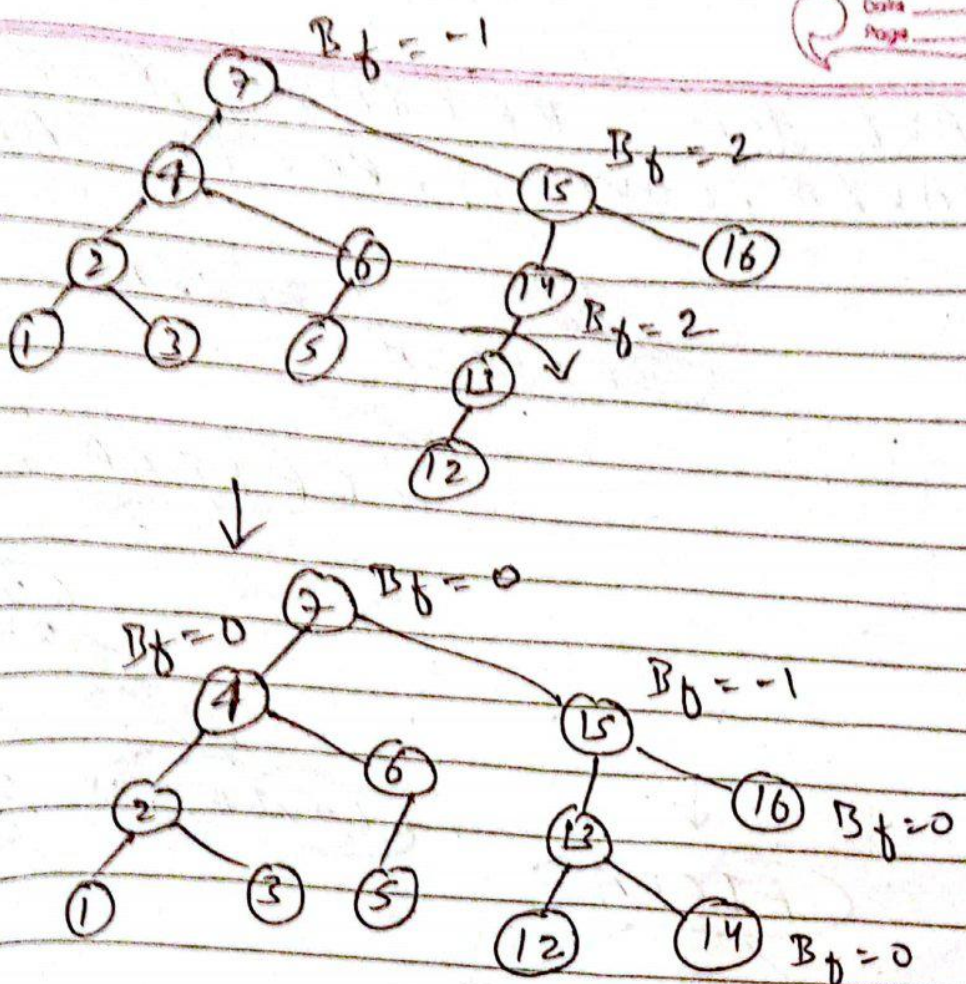
Insert 14



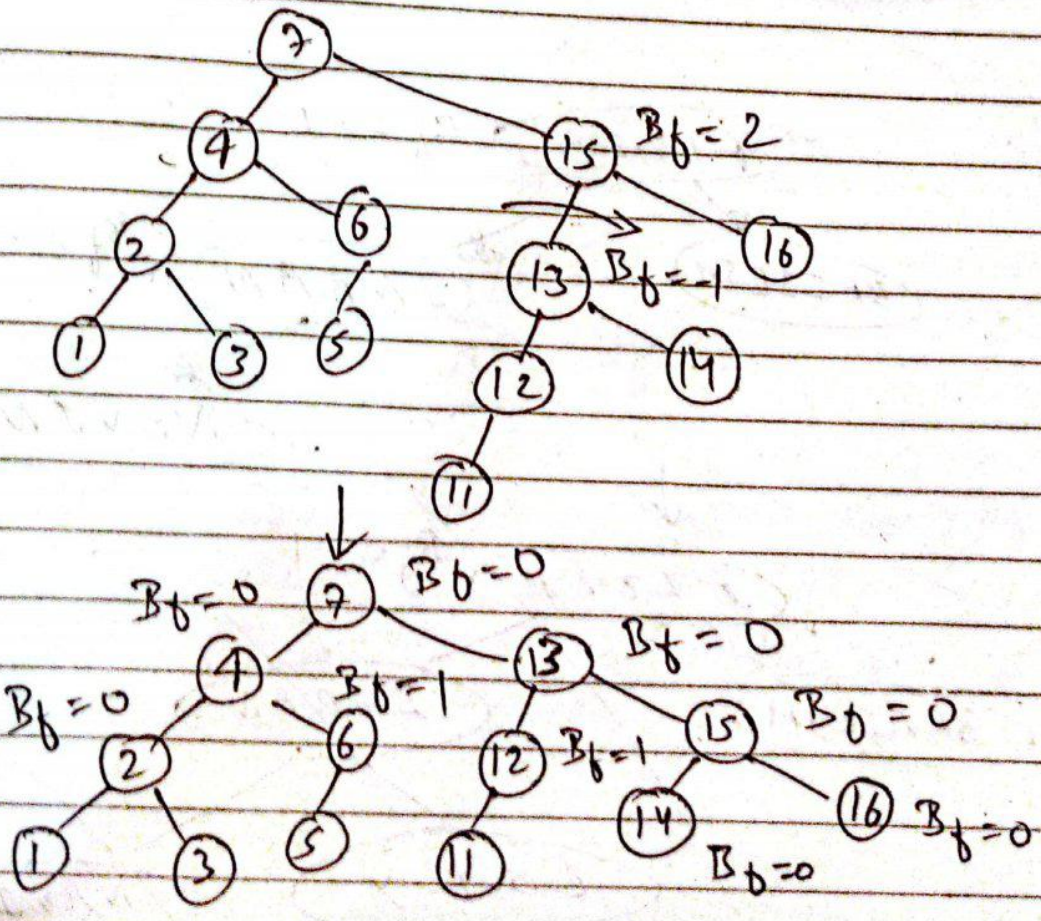
Insert 13





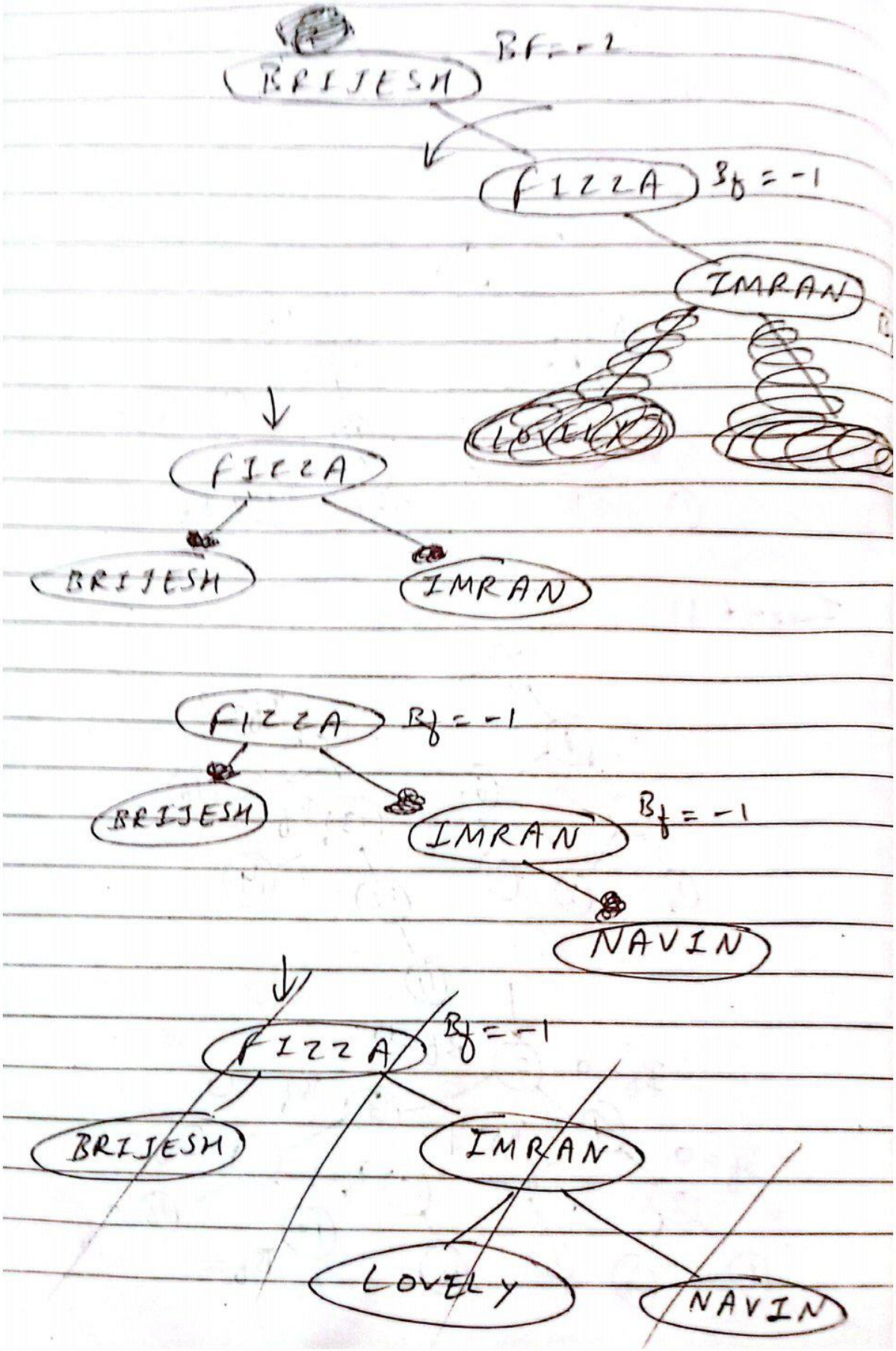

Bf = -1
Bf = 2
Bf = 2
Bf = 0
Bf = 0
Bf = -1
Bf = 0
Bf = 0

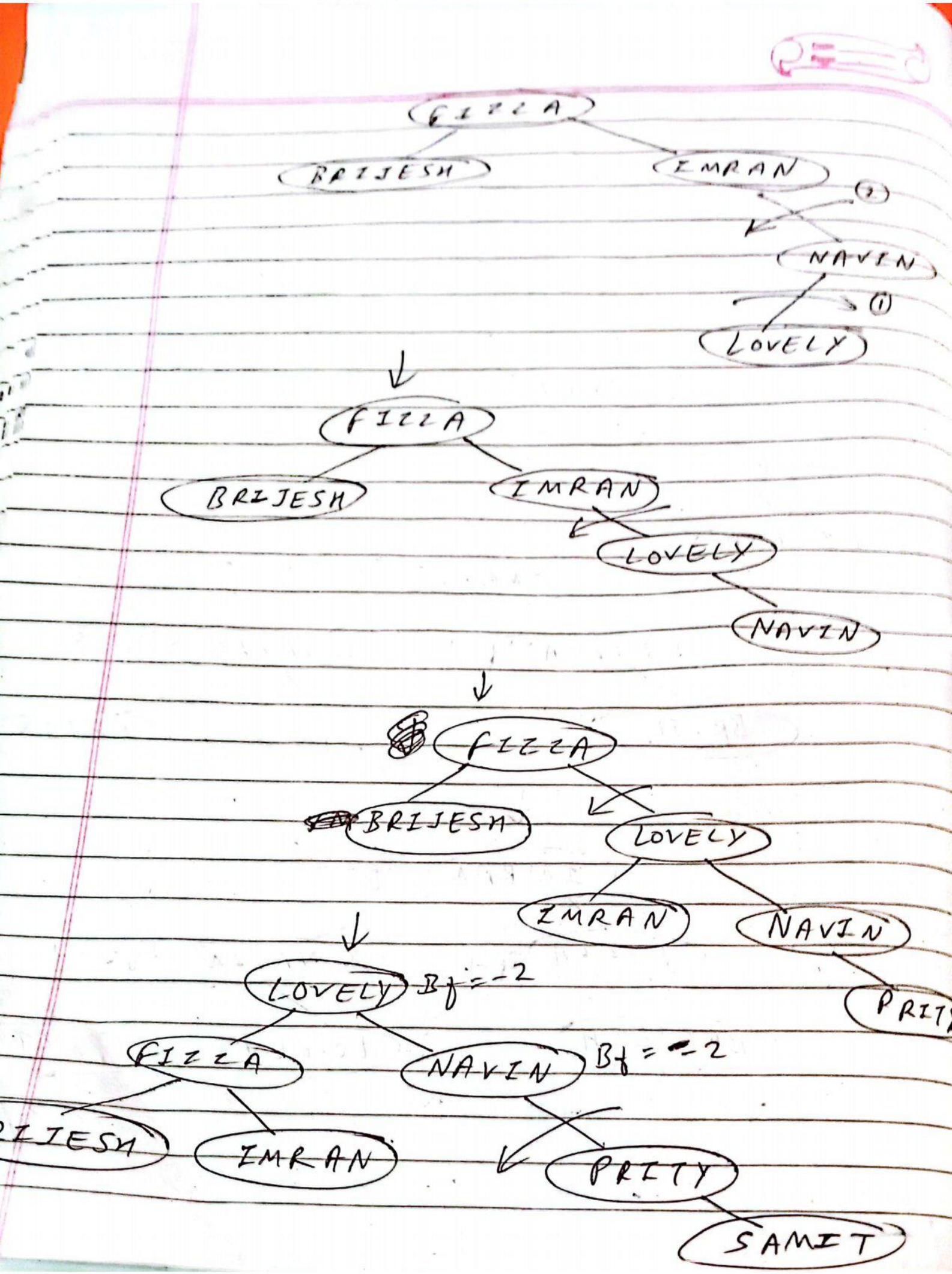
Insert 11

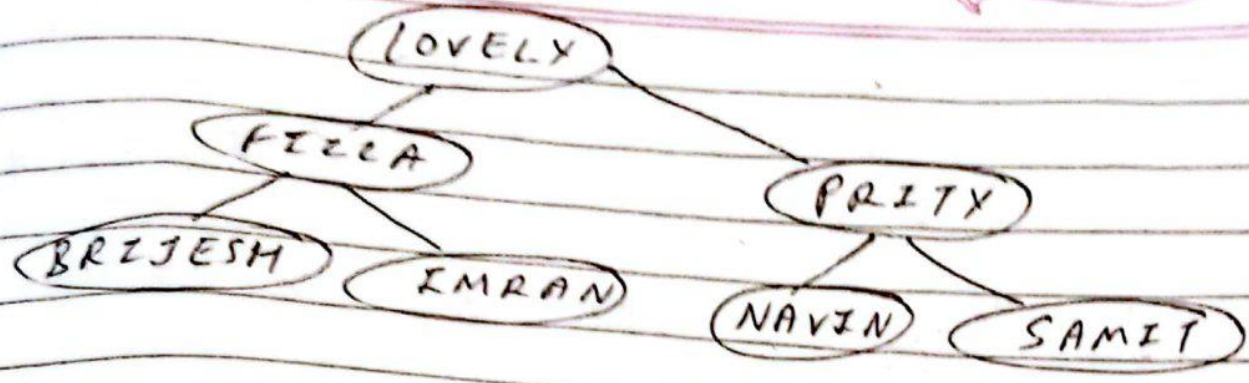

Bf = 2
Bf = -1
Bf = 0
Bf = 0
Bf = 0
Bf = 1
Bf = 0
Bf = 1
Bf = 0
Bf = 0

Q:- A

All name : BRIJESH, FIZZA, IMRAN, NAVIN, LOVELY, PRITY, SAMIT




FIZZA
BRIJESH
IMRAN
NAVIN
LOVELY
FIZZA
BRIJESH
IMRAN
LOVELY
NAVIN
FIZZA
BRIJESH
LOVELY
IMRAN
NAVIN
LOVELY
Bf = -2
FIZZA
NAVIN
Bf = -2
IMRAN
PRITY
SAMIT



* Huffman Coding : variable length encoding

-> It was proposed by Dr. David A Huffman in 1952. It is most efficient for representing numbers, letters, other symbols using binary code.

Application of Huffman code

* Both .mp3 and .jpg file formats use Huffman coding at one ~~Gra~~ stage of compression
* Used for Standard data compression.

Algorithm for Huffman Coding

1) Take the characters and ~~the~~ their frequencies and arrange in ascending order.
2) Add two lowest frequencies and form a tree assume root of the tree as new item.
3) Insert new item in the sorted list of vertices which ~~are~~ are not ~~have~~ not added in a tree.
4) Repeat step 2 to 3 untill a single tree is

formed containing all characters.
Assign a prefix code for binary tree
Assign '0' for left side & 1 for right side.
Get Huffman code for all characters.

* B^+ -tree -> All the leaf nodes are connec

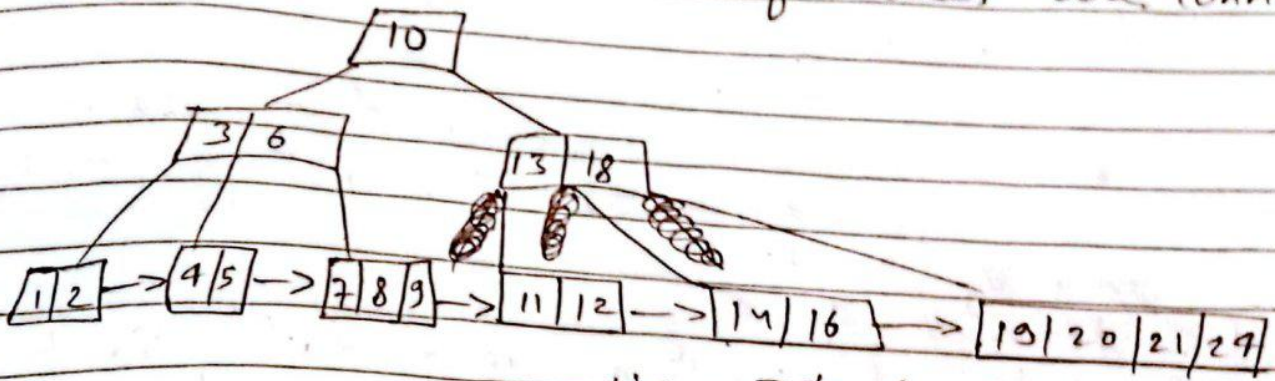


fig: B^+ tree

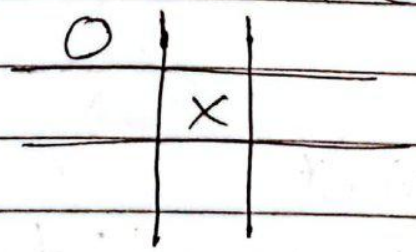
* Construct B-tree of order 5: 1, 12, 8, 2, 5, 14, 28, 17, 7, 52, 16, 48, 68, 3, 26, 29, 55, 45.

* Game tree

What is a game tree?

How this can help in formulating the g strategies where there are many poss

eg Tic-tac-toe



7. Sorting

Ascending & Descending
(Increasing) (Decreasing)

-> Sorting is arrangement of data into either ascending or descending order.

Types of Sorting

1) Internal Sorting

-> An internal sorting is a data sorting process that takes place entirely within the main memory of a computer; generally applied to small collection of data.

2) External Sorting

-> External sorting is a data sorting process where data are generally stored in a slower secondary storage; its used when data to be sorted is huge in number.

* Stable and Unstable Sort

-> Stable sorting algorithms maintains the relative order of records with equal keys (i.e values). That is, a sorting algorithm is stable if whenever there are two records R and S with the same key and with R appearing before S in the original list then R will appear before S in the sorted.

Ram	98
Hari	90
Sita	75
Geeta	98
Shyam	86
Kiran	90
Charan	86

Ram	98
Geeta	98
Hari	90
Kiran	90
Shyam	86
Charan	86
Sita	75

* Selection Sort

Main idea

* Find largest element in array.
* Exchange it with element in the last position.
* Find the second largest element and exchange it with element in second la position.
* Continue untill array is sorted.

Algorithm: SELECTION_SORT (A, n)

1) Start
2) loop $i = n-1$ to $i > 0$ $i = i-1$
   - 2.1 Max = A[0]
   - 2.2 index = 1
   - 2.3 loop 2 $J = 1$ to $J \leq 1$ $j = j+1$
     - 2.3.1 If A[J] Max Then
       - Max = A[J]
       - index = J
     - 2.3.2 END IF

$O(n^2) \rightarrow 5^2 => 25$ more steps Slower

2.4 End loop 2
2.5 A[index] = A[i]
2.6 A[i] = Max
3 ~~Stop~~ End loop
4. Stop

A = {35, 50, 75, 30, 60, 20}

Soln:

iteration	i=0	i=1	i=2	i=3	i=4	i=5	W
0	35	50	75	30	60	20	
1	35	50	20	30	60	75	
	(0→5) find max & swap with last ind						
2	35	50	20	30	60	75	0-
3	35	30	20	50	60	75	0-
4	20	30	35	50	60	75	0-
5	20	30	35	50	60	75	0-

Insertion Sort

-> It is a most common sorting technique used by card players. It is an in-place compar based sorting algorithm that is appropriate for small inputs. Here a sub-list is maintained which is always sorted. An element which is to be inserted in the sorted sub-list, has to find its appropri position while being inserted.

INSERTION - SORT (A, N)

1) loop j = 1 to N-1 step = 1
   1.1 key = A[j]
   1.2 i = j - 1
   1.3 while i ≥ 0 && A[i] > key
      1.3.1 A[i+1] ← A[i]
      1.3.2 i ← i-1
   1.4 End while
   1.5 A[i+1] ← key
2. End loop.

A = 15, 12, 14, 16, 11, 13

iteration	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	Wor... involv...
0	15	12	14	16	11	13	i=1 Key=1 key i... less t... A[0], ... insert... A[0]

A = 15, 12, 14, 16, 11, 13

insertion	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	Work involved
0	15	12	14	16	11	13	i=1 key=12 key is [l... than A[0], so insert... at A[0]
1	12	15	14	16	11	13	i=2 key=14 key is le... than A[1], so inser... at A[1]
3	12	14	15	16	11	13	i=3 key=16, since th... key is greater than... A[2], so no insert...
4	11	12	14	15	16	13	i=4 key=11 since k... is less than A[3], A... A[1] and A[0] so inse... at A[0].
5	11	12	13	14	15	16	i=5 key=13 since k...

is less than A[1], A[3] & A[2] so insert at A[2].

* Bubble Sort

→ Make repeated passes through a list of items exchanging adjacent items if necessary. At each pass, the largest (or smallest) unsorted item will be pushed (bubble out) to its proper place. Stop when no more exchange are done in last pass. Bubble sort is also known as exchange sort or comparison.

BUBBLE_SORT(A,N)

1) loop 1 i=0 to N-1
   1.1 loop 2 j=0 to N-i
      1.1.1 if A[j] > A[j+1]
         temp = A[j]
         A[j] = A[j+1]
         A[j+1] = temp
      1.1.2 End if
   1.2 End loop2
2 End loop 1

	A[0]	A[1]	A[2]	A[3]	A[4]
~~i=0~~					
i=0	75	95	35	20	60
J=0	75	95	35	20	60
J=1	75	35	95	20	60
J=2	75	35	20	95	60
J=3	75	35	20	60	95

i = 1	J = 0	35	75	20	60	95
	J = 1	35	20	75	60	95
	J = 2	35	20	60	75	95

8.

* RadixSort or Bucket Sort

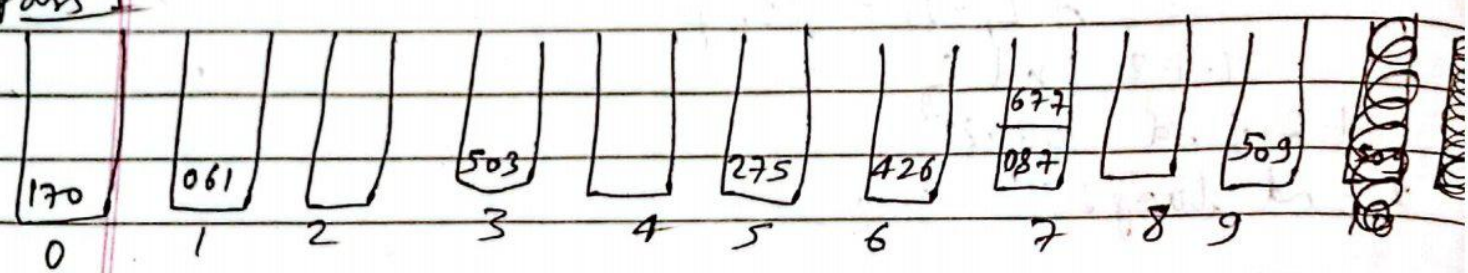
-> While sorting the decimal number we need to buckes labelled 0-9. The number of iteration required is the number of digits present in the largest number.

Sort the following numbers using radix sort.

275, 87, 426, 61, 509, 170, 677 and 503

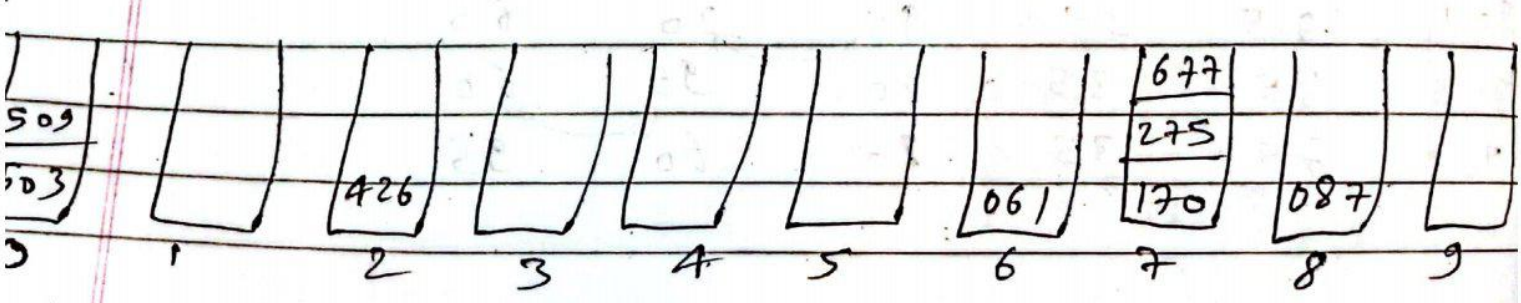
Input : 275, 087, 426, 061, 509, 170, 677, 503

Pass 1



Collect the values from left to right and bottom to top.

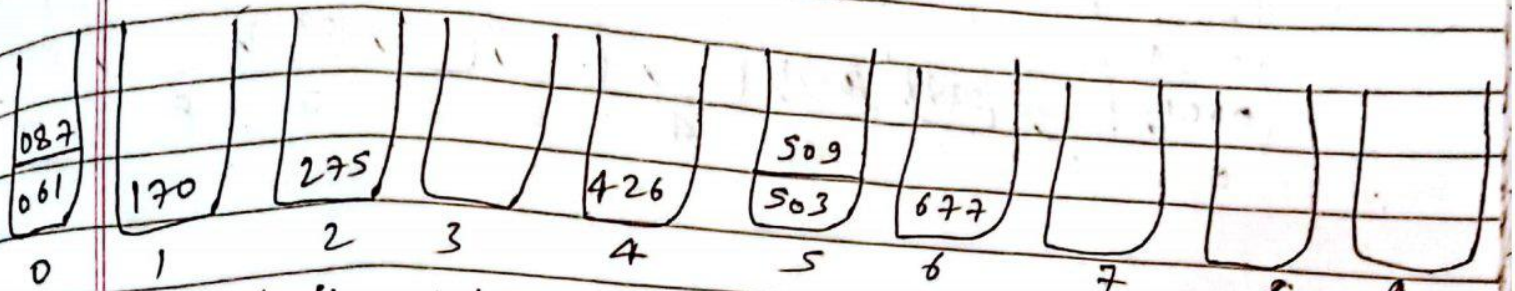
Pass 2 : 170, 061, 503, 275, 426, 087, 677, 509



collect the values from left to right & bottom to top.

pass 3 :

503, 509, 426, 061, [redacted] 170, 275, 677, 087



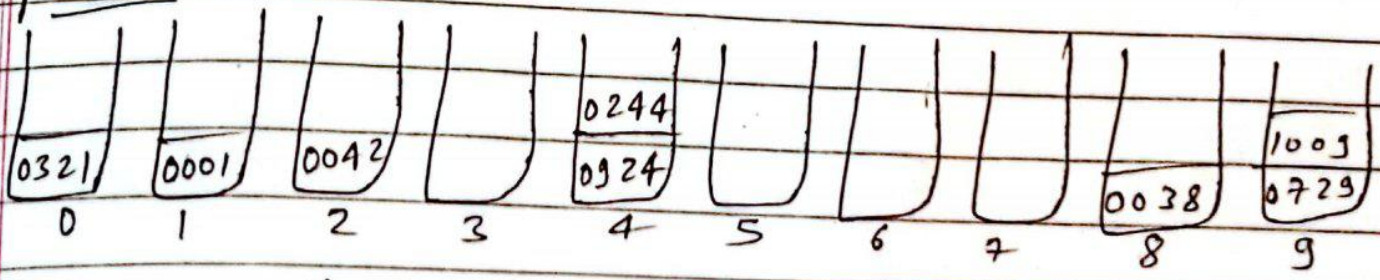
→ Collect the values from left to right & bottom to top

[redacted] 061, 087, 170, 275, 426, 503, 509, 677

* [redacted] 924, 1, 244, 729, 338, 42, 1009, 321

Input : 0924, 0001, 0244, 0729, 0338, 0042, 1009, 0321

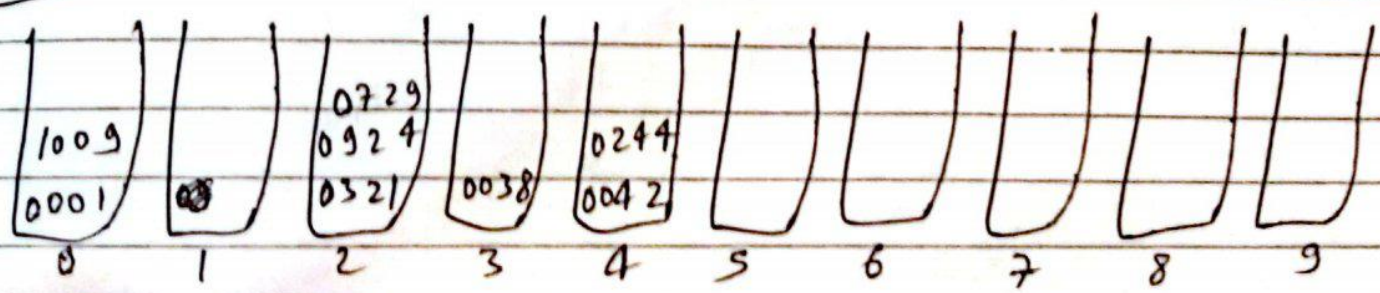
pass 1



→ " " " "

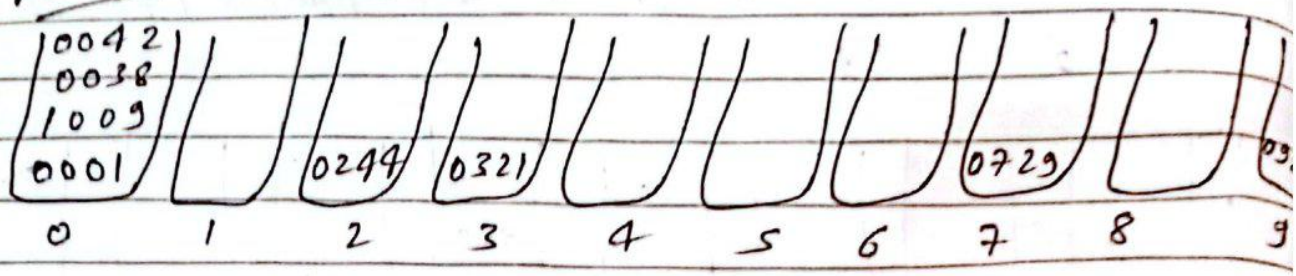
0321, 0001, 0042, 0924, 0244, 0038, 0729, 1009

pass 2



-> 400 0001, 1009, 0321, 0924, 0729, 0038, 02
0042

Pass 3



-> 0001, 1

*Quick Sort

Here we select any element from a list and find its right position where all the elements on its left is smaller than pivot and all the element on its right is greater than pivot. We repeat this process untill there is single element on its left and right of pivot.

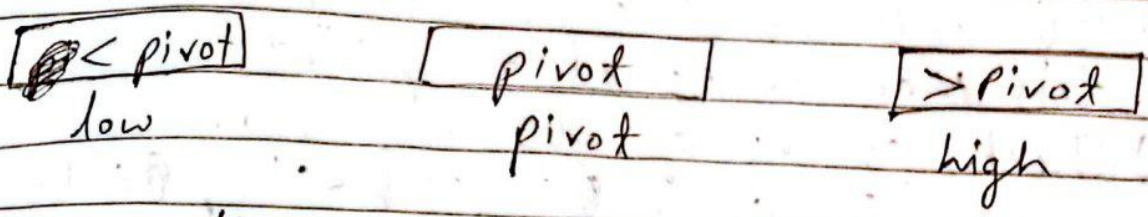


fig:- Partition of list in quict sort.

Or

9 ← (12) → (45) 13 22

9 12 13 22 (45)

9 (12) (13) 22 (45)

Steps

1) Pick an element, called a pivot; from the array
2) Partitioning: ~~reco~~ ~~rorder~~ reorder the array so that all elements with value less than pivot comes before the pivot, while all the elements with values greater than pivot comes after pivot.
3) ~~for~~ Recursively apply the above two steps to the sub array of elements with smaller

values and sub-array of elements with greater value.

* Algorithm

1) Move down towards right and up toward left.
2) Down will stop once it gets the value greater than pivot.
3) Up will stop once it gets the value less than or equal to pivot.
4) Check status of down & up.

If down index < up index
Swap (down ⟷ up) element
else
Swap (up ⟷ pivot)

eg.

```
      0   1   2   3   4
     15, 45, 35, 20, 60
     ↑d                ↓up
```

pivot = 15

①

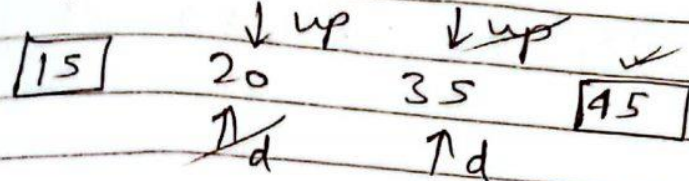
```
     ↓up  1   2   3   4↓ up
     15, 45, 35, 20, 60
     ↑d   ↑d
```

pivot = 15

②

```
     [15] 45  35  20↓up  60↓up
          ↑d          ↑d
```

pivot = 45



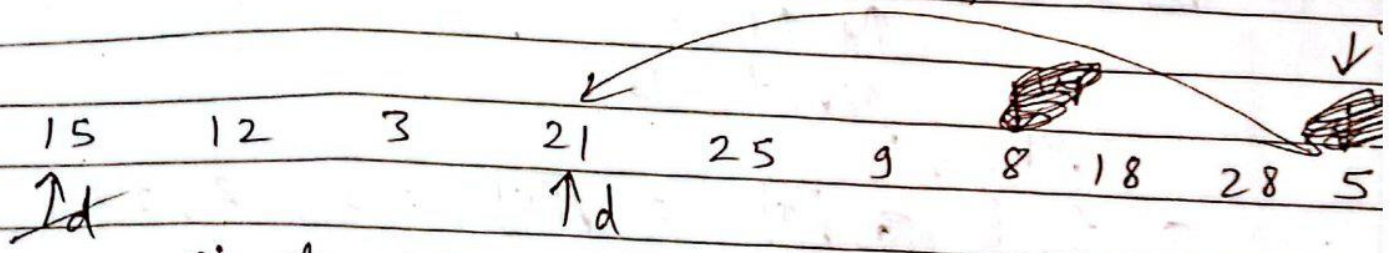
pivot = 20

15 | 20 | 35 | 45 | 60

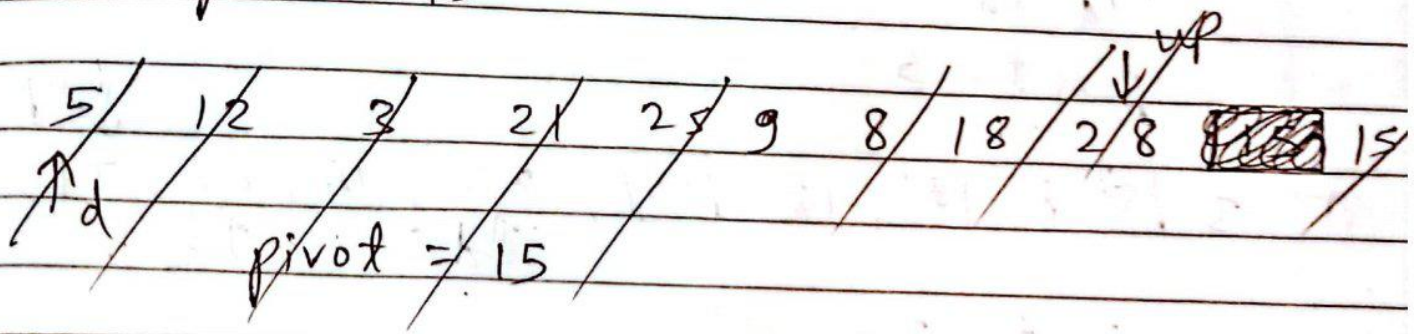
Q

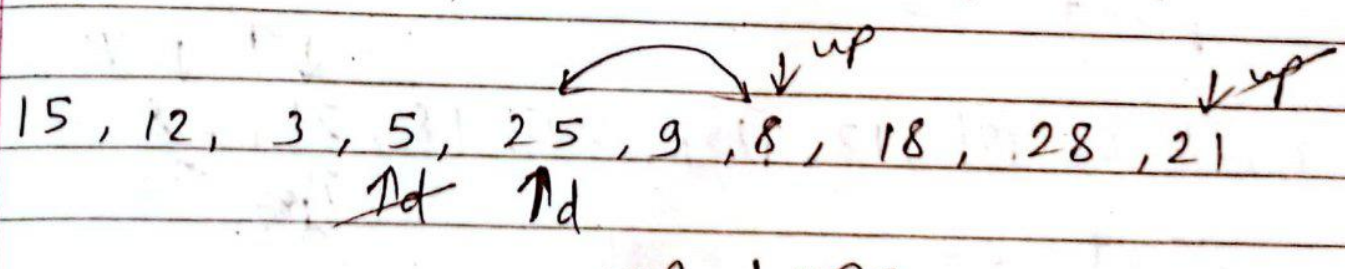
15, 12, 3, 21, 25, 9, 8, 18, 28, 5

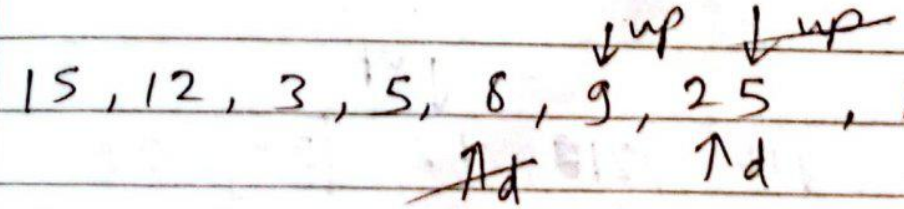
(1)



pivot = 15







9, 12, 3, 5, 8, [15], 25, 18, 28, 21

② 9, 12, 3, 5, 8, [15], 25, 18, 28, 21

pivot = 9

9, 8, 3, 5, 12, [15], 25, 18, 28,

pivot = 9

③ 5, 8, 3, [9], 12, [15], 25, 18, 28, 2

pivot = 5

5, 3, 8, [9], 12, [15], 25, 18, 28, 21

pivot = 5

④ 3, [5], 8, [9], 12, [15], 25, 18, 28, 21

pivot = 25

3, [5], 8, [9], 12, [15], 25, 18, 21, 28

pivot = 25

⑤ 3, [5], 8, [9], 12, [15], 21, 18, [25], 28

pivot = 21

3, [5], 8, [9], 12, [15], 18, 21, [25], 28

So,
Hence, Sorted
3, 5, 8, 9, 12, 15, 18, 21, 25, 28

Q. 7, 9, 45, 12, 56, 90, 3, 8, 50

① pivot = 7

7, 3, 45, 12, 56, 90, 9, 8, 50

pivot = 7

② 3, [7], 45, 12, 56, 90, 9, 8, 50

pivot = 45

3, 7, 45, 12, 8, 90, 9, 56, 50

pivot = 45

3, 7, 45, 12, 8, 9, 90, 56, 50

pivot = 45

3, 7, 9, 12, 8, [45], 90, 56, 50

(3) 3, [7], 9, 12, 8, [45], 90, 56, 50

pivot = 7

3, [7], 9, 8, 12, [45], 90, 56, 50

pivot = 9

3, [7], 8, [9], 12, [45], 90, 56, 50

pivot = 90

3, [7], 8, [9], 12, [45], 50, 56, [90]

pivot = 50

3, [7], 8, [9], 12, [45], [50], 56, [90]

13/2 13/4 13/8

*Shell Sort

-> Shell Sort developed by Donald L Shell, is a non-stable sort. shell sort is the modification of insertion sort. Shell sort improves on the efficiency of insertion sort by quickly shifting values to their destination. Average time is $O(n^{1.25})$ while worst case is $O(n^{1.5})$

~~Inserted~~

Instead of sorting an entire array at once, the array is first divided into smaller segments. eg k segments. Then these segments are sorted separately using the insertion sort. The shell sort is called diminishing increment sort because value of k is continuously decreased.

Example :- 45, 36, 75, 20, 05, 90, 80, 65, 30, 50, 10, 75, 85

=>

45	36	75	20	05	90	80	65	30	50	10	75	85
0	1	2	3	4	5	6	7	8	9	10	11	12

Sets assume the value of k = 3

Sub file 1 : a[0] a[3] a[6] a[9] a[12]

" " 2 : a[1] a[4] a[7] a[10]

" " 3 : a[2] a[5] a[8] a[11]

20 05 30 45 10 36 75 30 50 10 36 90 75 80 65 90

45	36	75	20	05	90	80	65	30	50	10	75	85
0	1	2	3	4	5	6	7	8	9	10	11	12

-> 20 05 30 45 10 75 50 36 75 80 65 90

Next iteration K = 2

10	20	36	30	36	75	65	75
20	05	30	45	10	75	50	36	75	80	65	90	85

K = 2

10 05 20 36 30 45 50 75 65 80 75 90 85

K = 1

05	10	30	36	65	75	75	80	85
10	05	20	36	30	45	50	75	65	80	75	90	85

K = 1

05, 10, 20, 30, 36, 45, 50, 65, 75, 75, 80, 85, 9

Or,

K -> n/2 -> n/4 -> n/8

6 3 1

6)

30	75	75	90
45	36	75	20	05	90	80	65	30	50	10	75	85

3)

20	10	36
05	45	36	50	10	80	65
45	36	30	20	05	75	80	65	75	50	10	90	85

)

20	30	45	50	36	65	75	75
10	10	36	36	75	65	80	85	90
05	20	30	45	50	75	75	65	80	85	90
20	05	30	45	10	75	50	36	75	80	65	90	85

05, 10, 20, 30, 36, 45, 50, 65, 75, 75, 80, 85, 90

* Merge Sort :- Divide and Conquer

→ A classical sequential sorting algorithm using divide and conquer approach. Unsorted list is first divided in to half. Each half is again divided into two continue until individual numbers are obtained. Then the two numbers are combined (merged) into sorted list of two numbers, pair of two will be merge to get list of four sorted numbers. continue this until all numbers are now combined into one sorted list.

* Divide : partition S in to sequences S_1 & S_2 about n/2 elements.
* Recur : recursively sort S_1 & S_2
* Conquer : Merge S_1 & S_2 into a single sorted sequence.

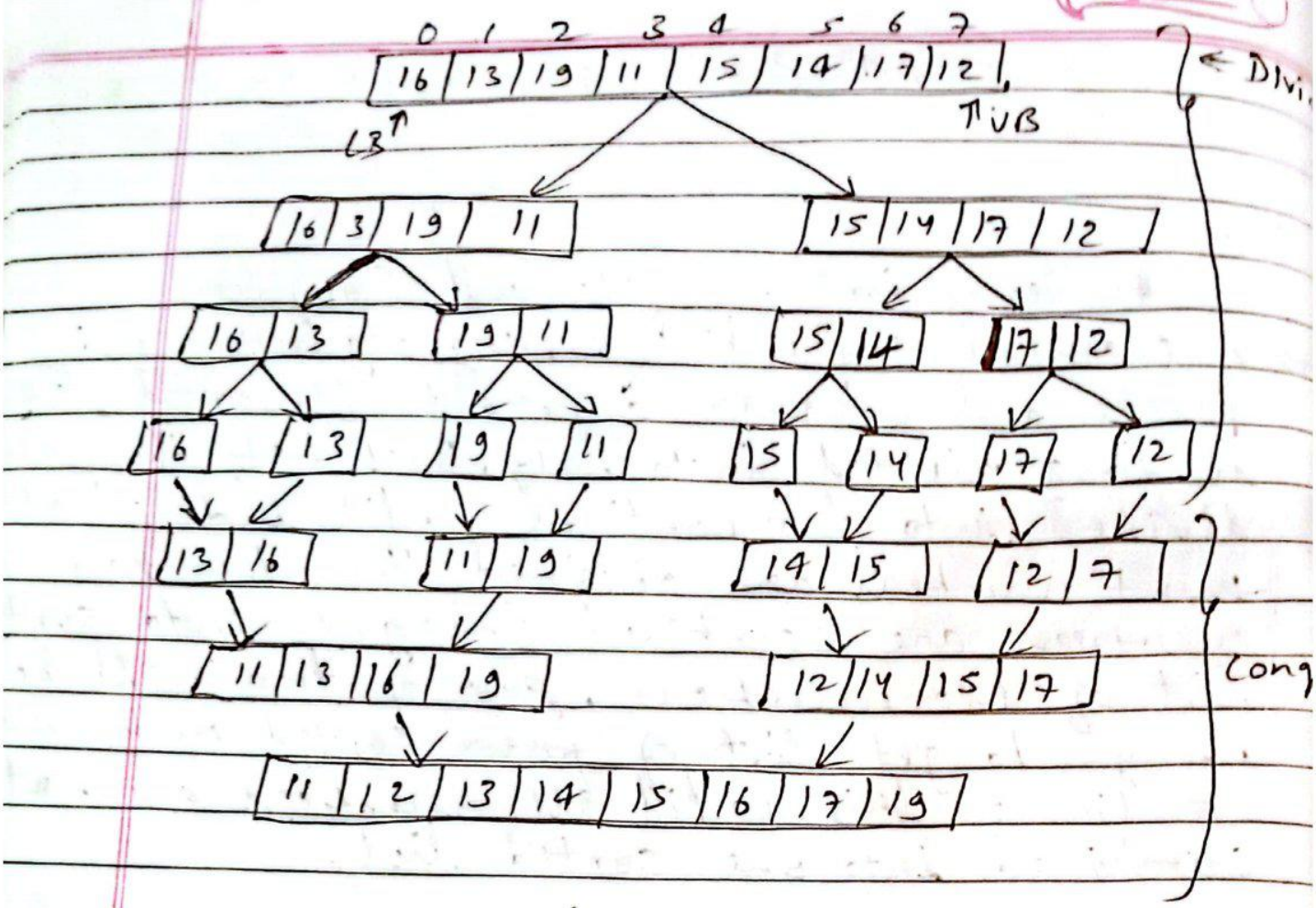
Sort following sequence : 16, 13, 19, 11, 15, 14, 17

mid ← n/2

mid ← 7/2 = 3

(LB, mid) (mid+1, UB)

(0, 3) (4, 7)




```
while (i<nL && j<nR)
 {
  if (L[i]<=R[j])
    {
    A[K] ← L[i]
     i ← i+1
}
else
    {
    A[K] ← R[j]
        J ← J+1.
}
```

```
        k ← k+1
    }
    while (i < nL)
    {
    A[k] ← L[i]
      i ← i+1
      k ← k+1,
    }
    while (j < nR)
      {
      A[k] ← R[j]
       J ← J+1
       k ← k+1
      }
```

for divide

```
Merge Sort (A)
  {
  n ← length (A)
  if (n < 2) return
      mid ← n/2
   left ← array (mid)
  right ← array (n-mid
```

Merge Sort (A, left, right)

1. If left < right. Then
   mid ← (left + right)/2
   Merge Sort (A, left, mid)
   Merge Sort (A, mid + 1, right)
   Merge (A, left, mid, right)
2. End IF

Merge (A, left, mid, right)

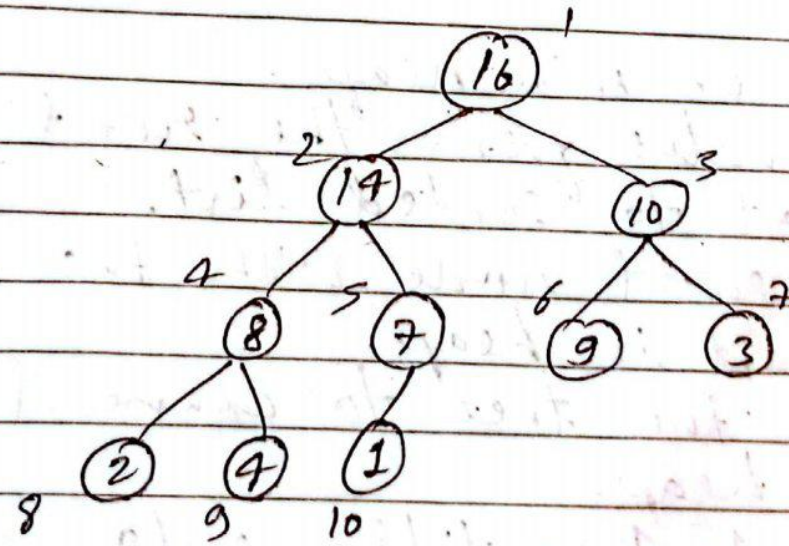
1. k = left, p = mid + 1
2. While left <= mid && p <= right Repe
   2.1 If A[left] <= A[p]
   T[i++] = A[left++]
   2.2 else
   T[i + 1] = A[p++]
3. End While
4. While left <= mid Repeat
   T[i + 1] = A[left ++]
5. End while
6. While p <= Right Repeat
   T[i++] = A[p++]
7. End While
8. loop i = k to Right
   A[i] = T[i - k]
9. Stop

* Heap &
* Heap Sort

-> Binary tree satisfying following properties
1. All leaves are on two adjacent levels
2. All leaves on the deepest level occur at left of the ~~tree~~ tree.
3. All leaves above the deepest level are completely filled.
4. Both children of any nodes are again heap.
5. The value stored at any node is at least as largest (or smallest) compared to its children.

Array representation of Heap

0	1	2	3	4	5	6	7	8	9	10
	16	14	10	8	7	9	3	2	4	1

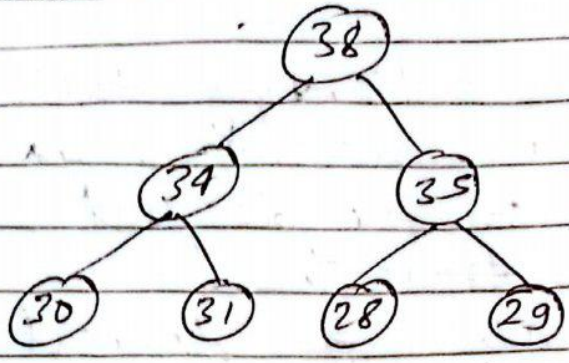


i → Parent A[i/2]

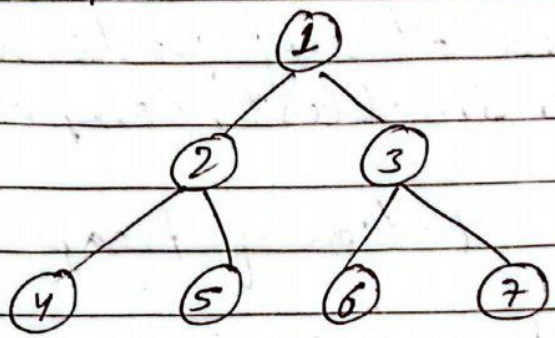
i → left child 2i

i → right child 2i+

Max - Heap



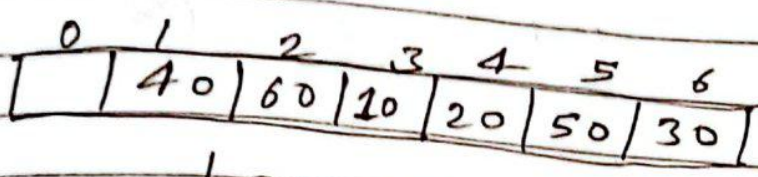
Min - heap

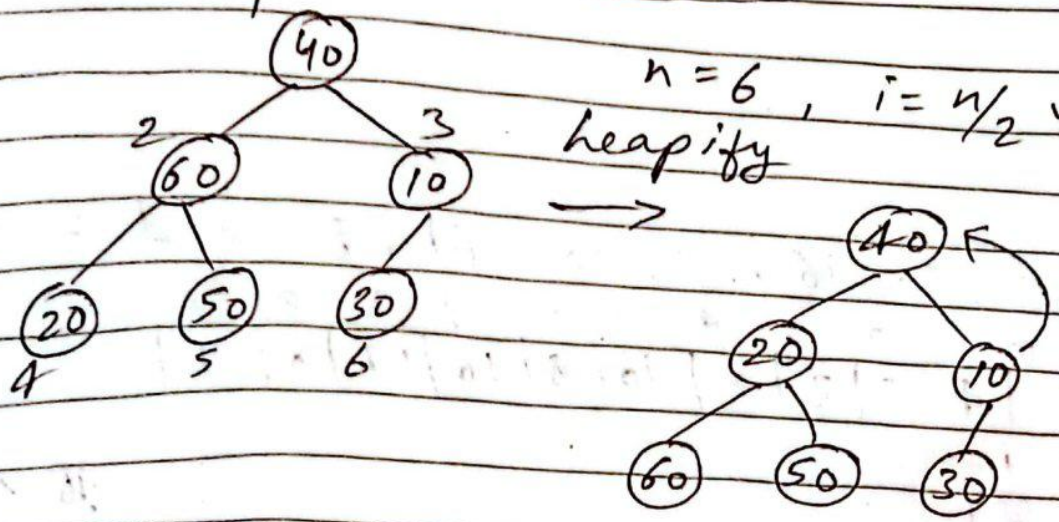


* Heap Sort

(1) Create max or min heap
(2) Extract the value from the root and transfer that to sorted list.
(3) Replace the root node with the value arriving last in heap.
(4) Re-arrange the tree to convert it to max or min heap.
(5) Repeat Step 1 - 4 untill we do not add all the elements to sorted list.

Sort in ascending order : 40, 60, 10, 20, 50, 30





n = 6, i = n/2 ~ 3, 2, 1

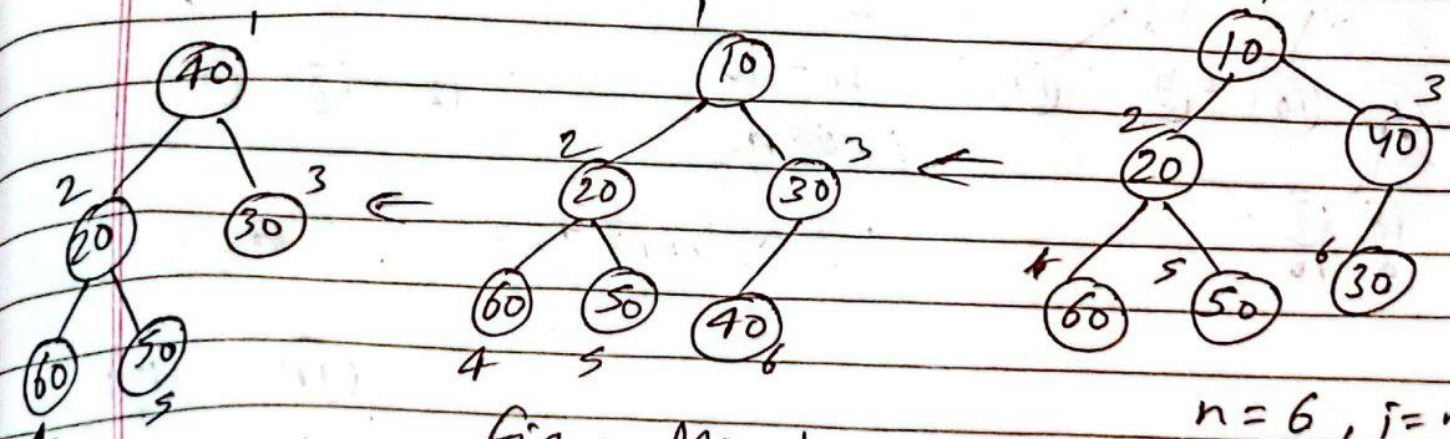


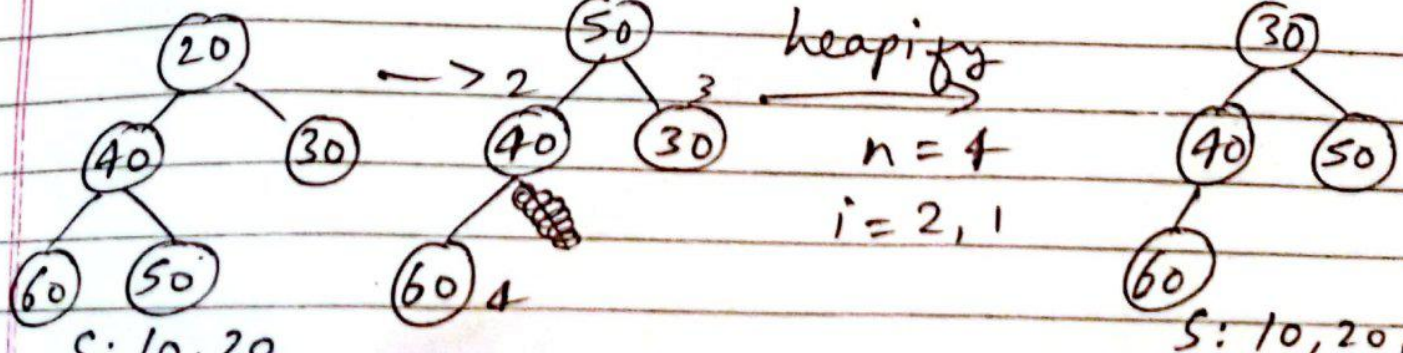
Fig :- Min heap

n = 6, i = n/2 ~ 3, 1

S: 10,

n = 5, i -> n/2 ~ 2

i = 2, 1



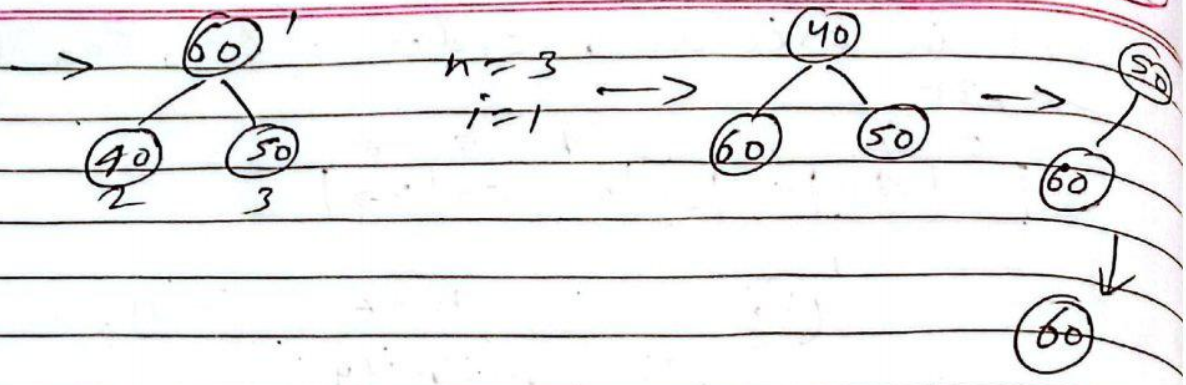
n = 4

i = 2, 1

S: 10, 20

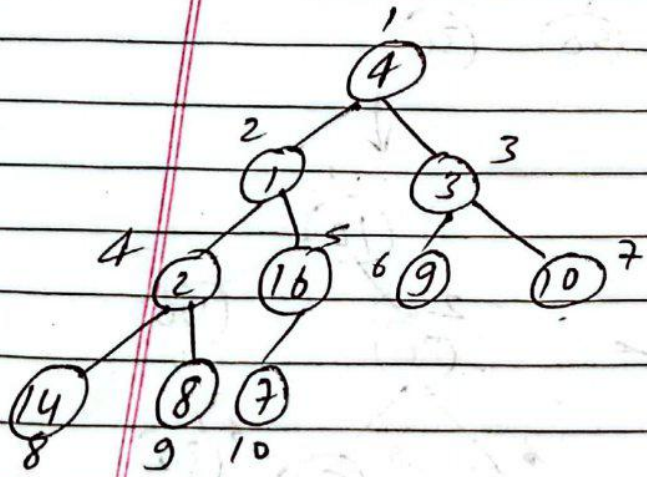
S: 10, 20, 30

S: 10, 20, 30, 40, 50, 60



* 4, 1, 3, 2, 16, 9, 10, 14, 8, and 7

0	1	2	3	4	5	6	7	8	9	10
	4	1	3	2	16	9	10	14	8	7

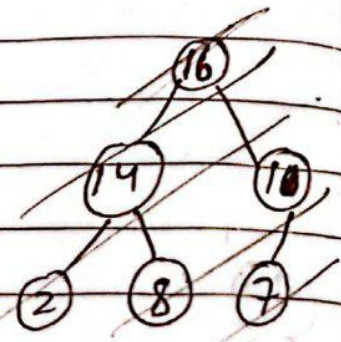


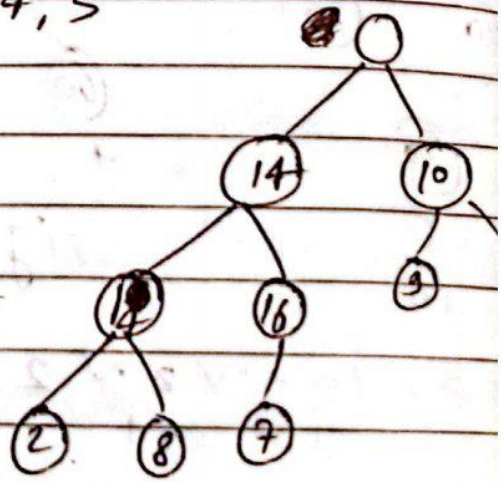
heapify →

n = 10

i = ~~n~~ n/2

= 5 ~ 1, 2, 3, 4, 5





Sorting Algorithm	Best Case	Average Case	Worst Case
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log$

Chapter :- 8
Searching

Information retrieval process with help of some keys.

* External search: Searching in Secondary Storage
* Internal search: " " primary "

* Types of Searching
* Sequential Search: Array Search
* Binary Search:
* Binary Search tree (BST)

* Sequential Searching

```
Sequential-Search (A, n, key)
1. Flag = False
2. loop i=0 to n-1 step = 1
   2.1 If A[i] = = key
           Flag = True
   2.2 End IF
3. End loop
4. Return Flag
```

* Analysis

Best Case = O(1)

Worst Case = O(n)

$$\text{Average} = \frac{(n+1)}{2}$$

$$= 0.5 + n/2$$

$$= O(n/2)$$

Binary Search

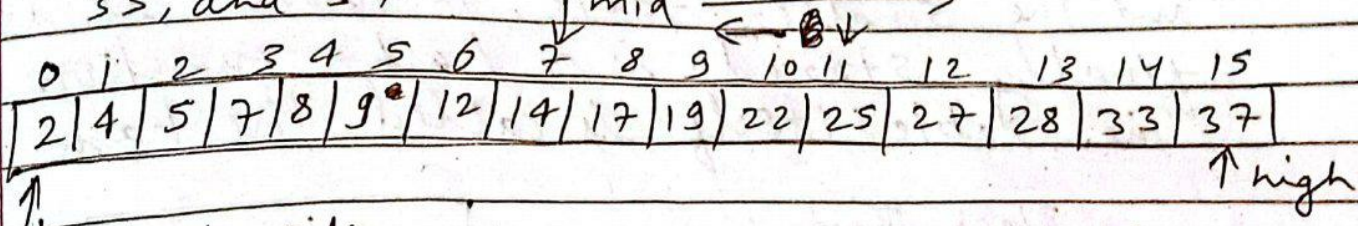
-> A binary search looks for a key in a list using divide and conquer strategy it assumes that the items in the array being search are stored. The algorithm begins at the middle of the array in a binary search. If the key which is found in middle then we stop if not we move either towards the left or right as the array is sorted. We repeatedly divide the problem in to almost half by searchi in the middle.

Binary - Search (A, key, low, high)

1. Start
2. If low > high Then
   return -1
3. End If
4. Else
   4.1 set mid = (low + high)/2
   4.2 If A[mid] == key Then
   return mid
   4.3 End If A[M,d] > key Then
   4.4 Else If
   return Binary - Search (A, key, low, mid - 1)
   4.5 End Else if
   4.6 Else
   return Binary - Search (A, key, mid+1, high)

4.7 End Else

Search 22 in following sequence of data :-
2, 4, 5, 7, 8, 9, 12, 14, 17, 19, 22, 25, 27, 28, 33, and 37



iteration : 1

low = 0

high = 15 , mid = (low + high)/2 = 15/2 = 7

A[7] = 14 , key = 22

(14 == 22) False

Binary - Search (A, key, 8, 15)

mid ← 23/2 = 11

A[11] = 25

key = 22

key < A[mid]

Binary - Search (A, 22, 8, 10) [A, key, low

key = 22, low = 8, high = 10

mid = (low + high)/2 = 9 , ~~key~~ k

Binary - Search (A, 22, 10, 10)

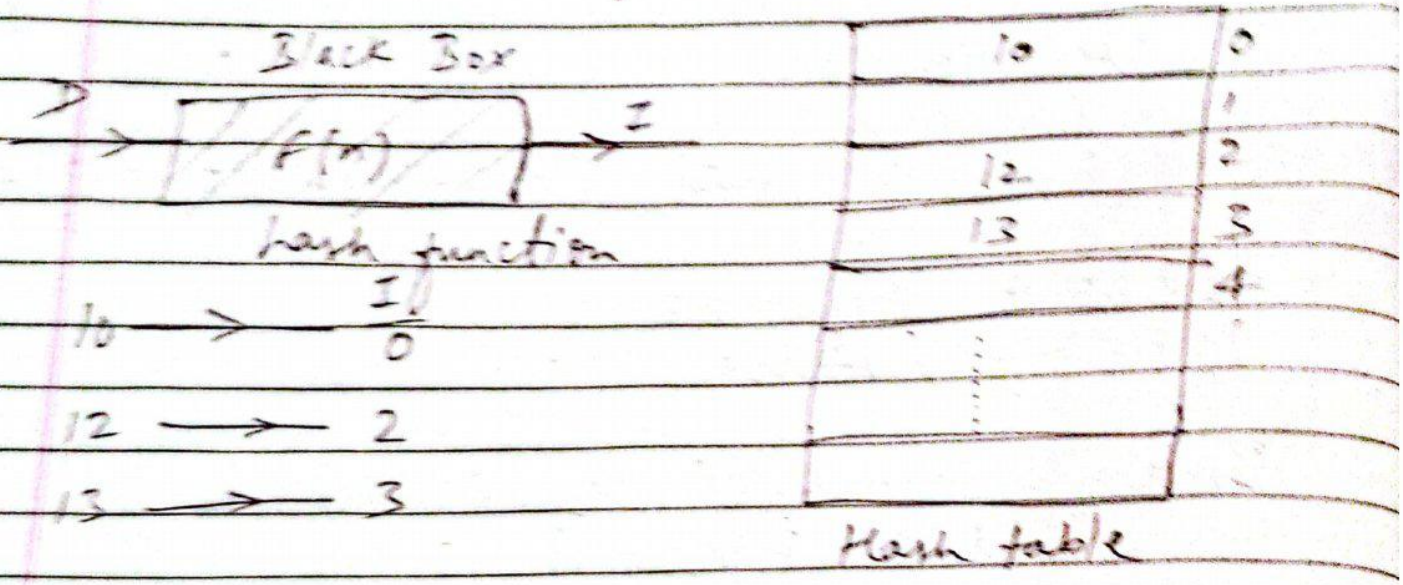
mid = (10 + 10)/2 = 10

A[10] = 22

True

Hashing

→ The purpose of mapping large number or data into a position in a smaller table is called hashing. The function which provides the map between the original data and the smaller table, in which it is finally stored is a hash function. The table itself is called the hash table.



Hash function maps data D to index I.

→ A good hash function should have
* Quick & easy to compute
* It should minimize the number of collisions
* The hash function should spread the data as evenly as possible ~~over~~ over the hash table.

A hash procedure must be ~~deleter~~ deterministic i.e data D will always generate Index I.

* Examples of Hash function.

1) Direct Hashing
* Data represents the index no.
* Suitable for small table
* Collision is high

2) Modular Arithmetic

D → [D % size of hash-table] → I

3) Folding Method

9841 42 43 11

42 + 43 + 44 = 129

4) Truncation

98510 66 8 2 4

5th ↑ 4th ↑ 1st ↑

I = 664

* Hash Collision

-> Two keys mapping to the same location in hash table is called "collision"
Collision can be reduced by using good hash function but cannot be prevented.
During insertion, the goal of collision resolution is to find a free slot in the table.

Collision resolution technique

1) Open Addressing
   1.1) Linear probing
   1.2) Quadratic probing
   1.3) Double hashing
2) Chaining

* Open Addressing

In open addressing we must deal with a collision by finding another unoccupied location else where in array. All items are stored in the hash table itself.

1) Linear probing

If collision occurs when we are inserting a new item into the table, we simply probe forward in the array, one step at a time, untill we find an empty slot where we can store new data.

Probe sequences

$h_i(key) = [h(key) + i]\%$. table-size

$i = 1, 2, \ldots\ldots, \text{table-size} - 1$

* Disadvantages

=> Data tends to cluster about certain point in the table, whereas other parts ~~having~~ being unused, hence searching is lengthy

Insert keys 18, 41, 22, 44, 59, 32, 31, 73.
If collision occurs resolve using linear probing where size of hash table is 13.

Ans:-

- $h'(X) = X \bmod 13$
- $h(18) = 18 \% 13 = 5$
- $h(41) = 41 \% 13 = 2$
- $h(22) = 22 \% 13 = 9$
- $h(44) = 44 \% 13 = 5$ (collision)

  When $i = 1$

  $h'(44) = [44 + i] \% 13 = 6$
- $h(59) = 59 \% 13 = 7$
- $h(32) = 32 \% 13 = 6$ (collision)

  when $i = 1$

  $h'(32) = (32 + 1) \% 13 = 7$ (collision)

  $i = 2$

  $h'(32) = (32 + 2) \% 13 = 8$
- $h(31) = 31 \% 13 = 5$ (collision)

  $i = 1$

  $h'(31) = (31 + 1) \% 13 = 6$ (collision)

  $i = 2$

  $h'(31) = (31 + 2) \% 13 = 7$ (collision)

  $i = 3$

  $h'(31) = (31 + 3) \% 13 = 8$ (collision)

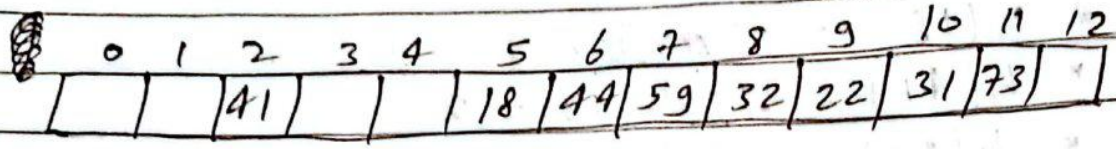
  $i = 4$

  $h'(31) = (31 + 4) \% 13 = 9$ (collision)

  $i = 5$

  $h'(31) = (31 + 5) \% 13 = 10$
- $h(73) = 73 \% 13 = 8$ (collision)

i = 1 (collision)
i = 2 (collision)
1 = 3
$h'(73) = (73 + 3) \% 13 = 11$



* Quadratic probing

-> Quadratic probing eliminates primary clustering problem of linear probing. Collision resolution f^n is quadratic, we try cells $h + 1^2$, $h + 2^2$, ... and so on are tested until an empty slot is found.

Probe sequence

$h + a^2$ where $a = 1, 2, 3, \ldots\ldots$

Problem

* Its not sure that we will probe all location in the table (i.e no gurantee to find an empty cell if table is more than half ful
* If hash table size is not prime, then more no of collisions.
* Although it eliminates the problem of primacy clustering it may result in Secondary clustering.

$h'(key) = (h(key) + i^2) \% size$
value

* Load the keys 23, 13, 21, 14, 7, 8, & 15 in this order in a hash table of size 7 using quadratic probing with $(i) = i^2$ and the hash fn $h(key) = key \% 7$

→

* $h(23) = 23 \% 7 = 2$
* $h(13) = 13 \% 7 = 6$
* $h(21) = 21 \% 7 = 0$
* $h(14) = 14 \% 7 = 0$ (collision)
  Next possible index for 14
  $h(14) = (0 + 1^2) \% 7 = 1$
* $h(7) = 7 \% 7 = 0$ (collision)
  Next possible
  $h'(7) = (0 + 1^2) \% 7 = 1$ (collision)
  $h'(7) = (0 + 2^2) \% 7 = 4$
* $h(8) = 8 \% 7 = 1$ (collision)
  $h'(8) = (1 + 1^2) \% 7 = 2$ (collision)
  $h'(8) = (1 + 2^2) \% 7 = 5$
* $h(15) = 15 \% 7 = 1$ (collision)
  $h'(8) = (1 + 1^2) \% 7 = 2$ (collision)
  $h'(8) = (1 + 2^2) \% 7 = 5$ (collision)
  $h'(8) = (1 + 3^2) \% 7 = 3$ (collision)

0	21
1	14
2	23
3	15
4	7
5	8
6	13

* Double Hashing

→ * Double hashing aim aims to avoid both primary and secondary clustering and guranteed to find ~~the~~ a free position in a hash table as ~~for~~ long as it's not full.

* The secondary hash f^n Hash2(key) cannot have zero values.
* If table size is a primary number, this guarantees that successive probes will (eventually) try every index in the hash table before an index is repeated.

- Double hashing algorithm
* When an item is inserted use Hash1(key) to determine location in table.
* If collision occurs ~~we~~ use hash2(key) to determine how far to move forward in the array looking for vaccant slot.

Probe sequence

$h_i(key) = [h(key) + i * hp(key)] \%$ table size

for $i = 0, 1, 2, \ldots, \text{table size} - 1$.

where h(key): first hash function and

h(p) key: second hash f^n.

Common way of finding hp(key)

$hp(key) = 1 + key \% (\text{table size} - 1)$

2) $hp(key) = q - (key \% q)$
Where q is a prime number less than table size

3) $hp(key) = q * (key \% q)$
Where q is a prime number less than table size.

* Load the keys 18, 26, 35, 9, 64, 47, 96, 36 & 70 in a hash table of size 13.

-> size = 13

$h_i(key) = [h(key) + i * hp(key)] \% \text{table size}$

$hp(key) = 1 + key \% (\text{table size} - 1)$

$h(key) = key \% \text{table size}$

Solution

* $h(18) = 18 \% 13 = 5$
* $h(26) = 26 \% 13 = 0$
* $h(35) = 35 \% 13 = 9$
* $h(9) = 9 \% 13 = 9$ (collision)
  $hp(9) = 1 + 9 \% 12 = 10$
  $h_1(9) = [9 + 1 * 10] \% 13 = 6$
* $h(64) = 64 \% 13 = 12$
* $h(47) = 47 \% 13 = 8$
* $h(96) = 96 \% 13 = 5$ (collision)
  $hp(96) = [1 + 96 \% 12] = 1.$
  $h_1(key) = [5 + 1 * 1) \% 13 = 6$ (collision)
* $h_2(key) = (5 + 2 * 1) \% 13 = 7$
* $h(36) = 36 \% 13 = 10$
* $h(70) = 70 \% 13 = 5$ (collision)

$h_p(70) = 1 + 70 \% 12 = 11$

$h_1(70) = [5 + 1 \times 11] \% 13 = 3$

0	26
1	
2	
3	70
4	
5	18
6	9
7	96
8	47
9	35
10	36
11	
12	64

* Chaining

-> It uses linked list as and dynamic data allocation. Hash table and associated hash functions are defined as usual manner except that the array in an array of pointers to linked list, one list for each index.

If an item/data to be inserted, the hash f^n will find the index and if there is a collision will have chain of nodes of linked list for every collisions.

* Demerits

Compared to array takes more space.

10, 12, 13, 55, 60, 70, 80, 32, 42, 21, 31

table size = 10

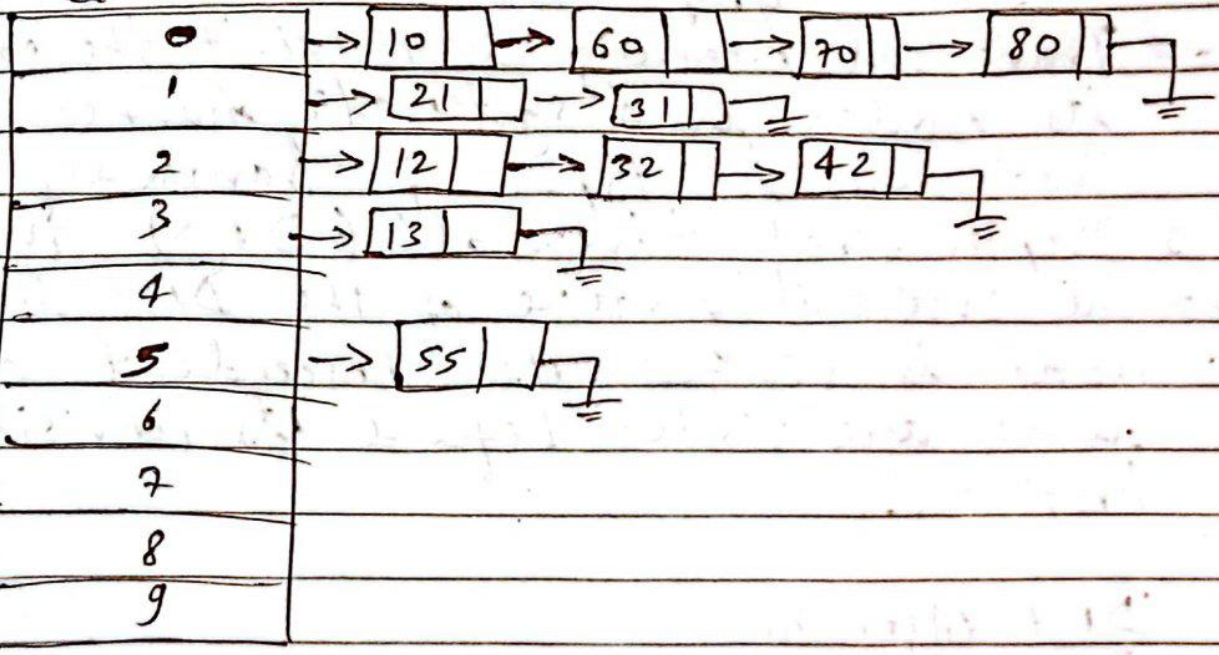
h(key) = key % tablesize

h(10) = 0 h(70) = 0 h(21) = 1

h(12) = 2 h(80) = 0 h(31) = 1

h(55) = 5 h(32) = 2

h(60) = 0 h(42) = 2



* Sentinel node

-> A sentinal node is a specifically designat node used with linked list and tree as a traversal path terminator. A sentinel nod does not hold or reference any data managed by the data structure.

* Graph Theory

* Adjacency list
* Adjacency matrix
* Minimum Spanning tree
  1. Kruskal
  2. Prims
* Shortest Path
  Dijkstra's

* Graph Traversal

-> Traversal meaning is to visit the each of its nodes in a systematic manner.
There are two types of traversal.

1. Depth first traversal (DFS) {use stack}

-> The essential feature of the DFT is that a node is visited. All descendent of the node are visited before its unvisited brothers.

DFT Algorithm

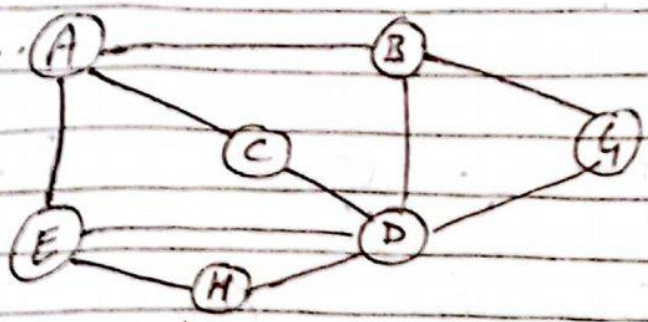
1. Set the starting vertex and push on to stack.
2. While stack is not empty Repeat
   2.1 Pop a vertex from the stack, call it v.
   2.2 If v is not already visited, add v to list of visited vertices
   2.3 find adjacent vertices of v (from adjacency matrix)
   2.4 push adjacent vertices on to stack if they are not already visited and not

in the stack.

3. End while

4. Stop

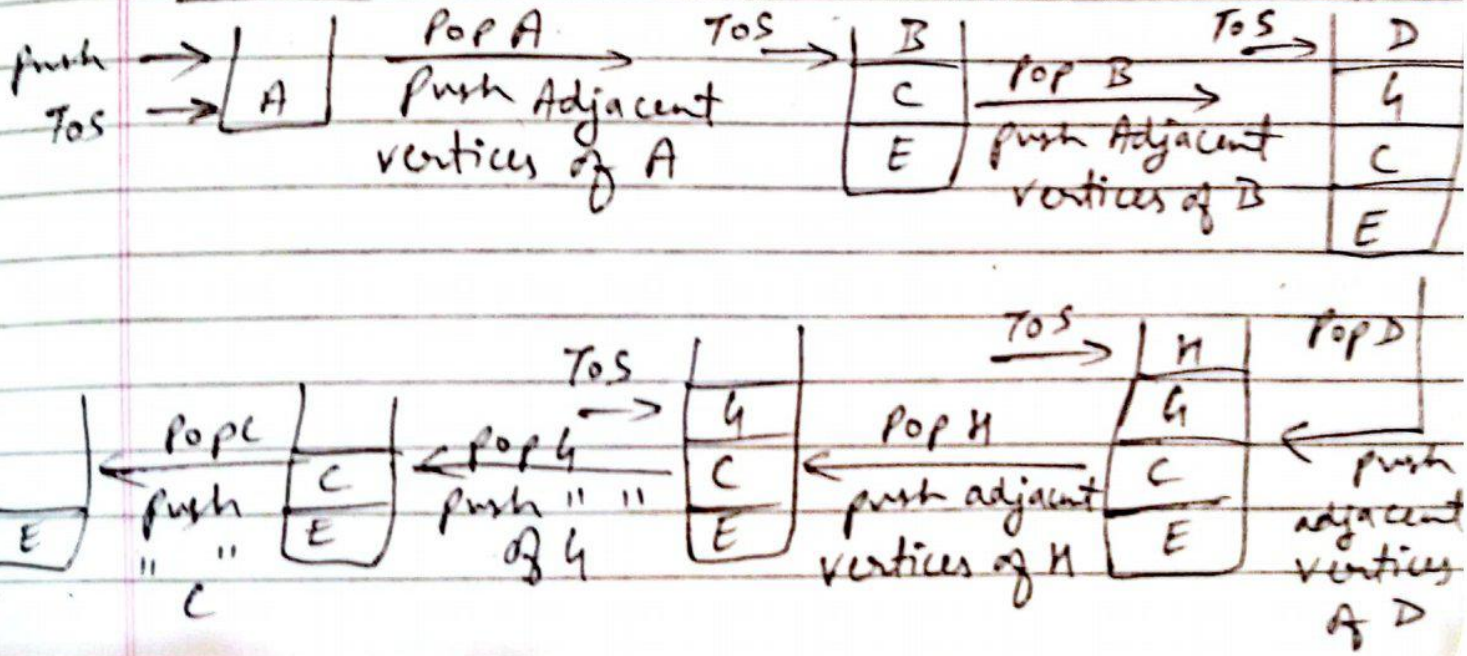
ex:-



Edge vertex	table Successor
A	B, C, E
B	A, D, G
C	A, D
D	B, C, E, G, H
E	A, D, H
G	B, D
H	D, E

Stack

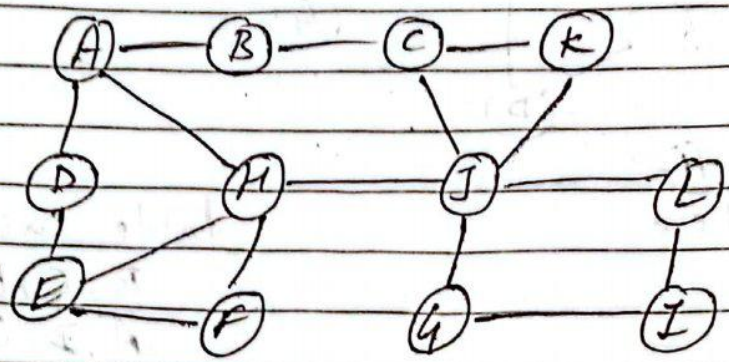
Visited Nodes : A, B, D, H, G, C, E



Pop E →
push "
" of E

Tos = -1

∴ Visited Nodes :- A, B, D, H, G, C, E

* 

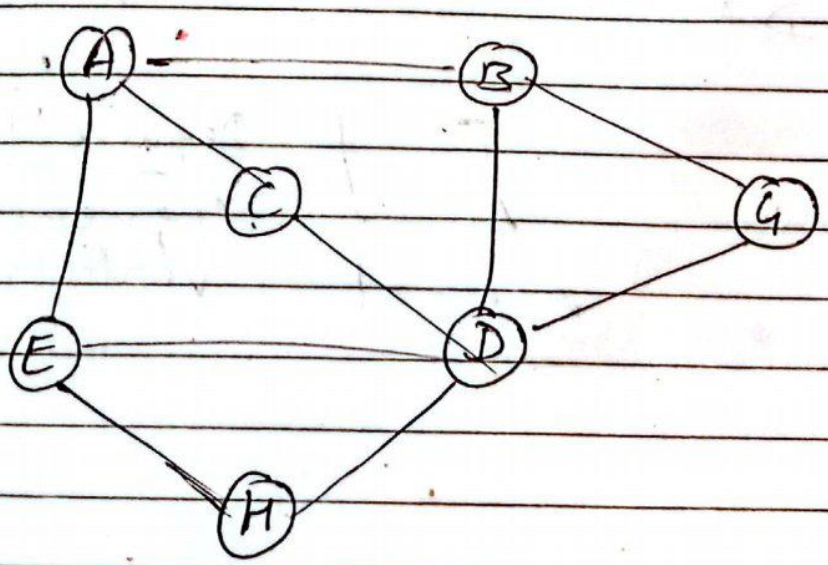
DFS = ?

Breadth First Traversal (BFT)

-> It is also called breath breadth first Search (BFS). It is implemented using queue. In BFT, visit all successors of visited node before visiting any successo of those successors.

Algorithm

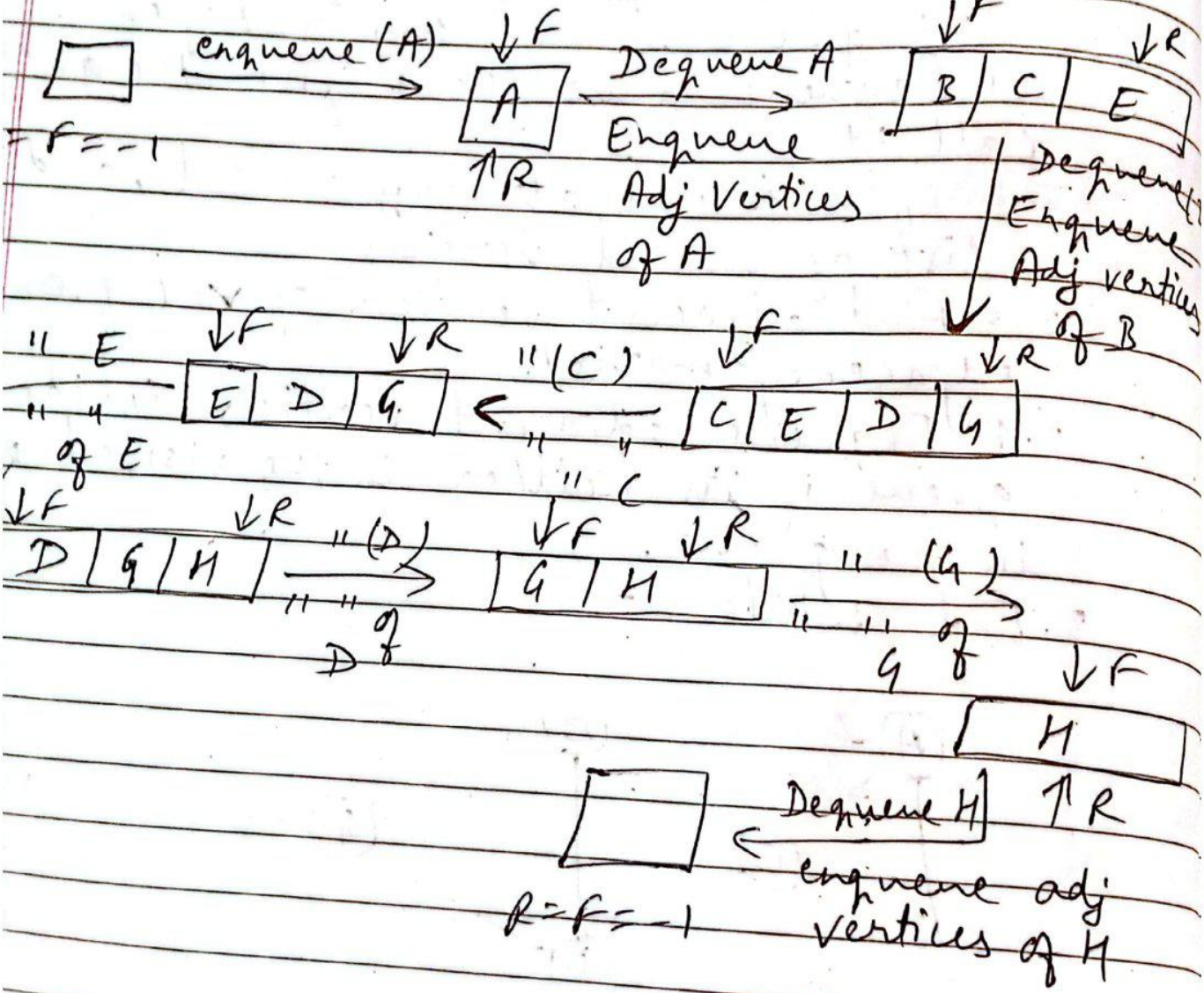
1. Set the starting vertex and enqueue it
2. While queue is not empty Repeat.
   2.1 Dequeue a vertex V from the queue call it v.
   2-2 If v is not already visited, add v list of visited vertices.
   2-3 Find adjacent vertices of v (from adjacency matrix).
   2.4 Enqueue adjacent vertices inside th queue if its neither in the visited no in the queue.
3. End While

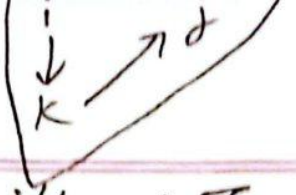


Answer is not unique

vertex	Adj vertex
A	B, C, E
B	A, D, G
C	A, D
D	B, C, E, G, H
E	A, D, H
G	B, D
H	D, F

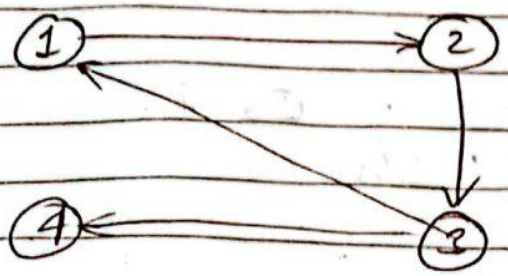
Visited list : A, B, C, E, D, G, H





1

* Warshall Algorithm: Transitive closure
-> Warshall algorithm determines whether there is a path between any two nodes in the graph. It does not gives the number of path between any two nod



Pre-requisite
* can only be applie to directed cyclic graph
* Remove self-bac loop & parallel ed

$r_{ij}^{(k)} = r_{ij}^{(k-1)}$ or $[r_{ik}^{(k-1)}$ && $r_{kj}^{(k}$

$k = 1, 2, 3, \ldots\ldots, n$

where n = number of nodes.

$r^{(0)}$

	1	2	3	4
1	0	1	0	0
2	0	0	1	0
3	1	0	0	1
4	0	0	0	0

for k=1

$r_{ij}^{(1)} = r_{ij}^{(0)}$ or $[r_{11}^{(0)}$ && r

$i=1, j=1, k=1$ | $r_{11}^{(1)} = r_{11}^{(0)}$ or $[r_{11}^{(0)}$ & $r_{11}^{(0)}]$

$i=1, j=2, k=1$

⋮

Similarly, if 1 write 1.
Or if and write 1 else 0

R(1)

	1	2	3	4
1	0	1	0	0
2	0	0	1	0
3	1	1*	0	1
4	0	0	0	0

↓ $i=4, k=1, J=2$

for $k=2$

R(2)

$r_{ij}^{(2)} = r_{ij}^{(1)}$ or $[r_{i_2}^{(1)}$ && $r_{2j}^{(1)}]$

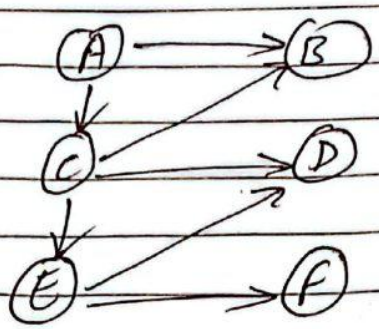
	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	1	1	1	1
4	0	0	0	0

$k=3$

R(3)

Topological Sort

→ Topological sort of a directed acyclic graph $G = (V, E)$ is a linear ordering of $u, u \in V$ such that $(u, v) \in E$ then u appears before v in the this ordering. If G is a cyclic graph then no such ordering exists.



Adjacency list

A	→	B, C
B	→	
C	→	B, D, E
D	→	
E	→	D, F
F	→	

Sorted list: A, C, B, E, D, F

① Initialize empty

Queue = ϕ $\quad R = F = -1$

Indegree of nodes

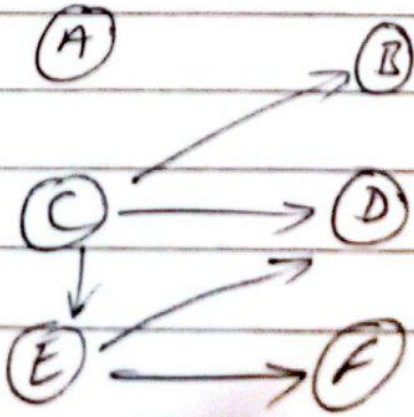
A: 0 $\quad$ B: 2 $\quad$ C: 1 $\quad$ D: 2

E: 1 $\quad$ F: 1

Enqueue node having indegree zero.

Queue: A $\quad R = F = 0$

Dequeue the node from Q and the edges going out of the node A.



② Queue : ϕ $R = F = -1$

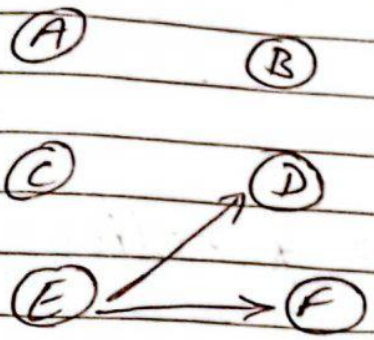
Indegree of nodes:

B : 1 C : 0 D : 2 E : 1 F : 1

Enqueue C

Queue : C $R = F = 0$

Delete node C and edges going out of C.



③ Queue = ϕ $R = F = -1$

Indegree of nodes

B : 0 D : 1 E : 0 F : 1

Enqueue B and E

Queue : E, B $R = 1$ $F = 0$

$\uparrow R$ $\uparrow F$

Dequeue B & edges goint out of B

No change in indegree.

Delete E & edges going out of E.

④ Queue : ϕ $R = F = -1$

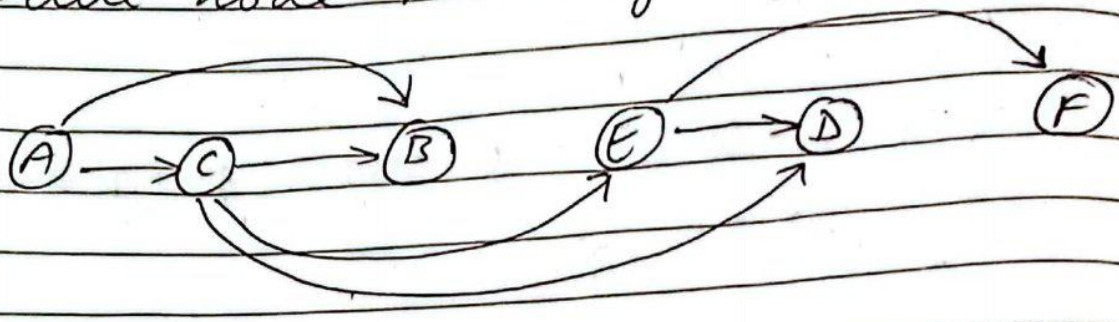
In degree of nodes

D : 0 F : 0

Queue : F, D $F = 0$, $R = 1$

$\uparrow R$ $\uparrow F$

Delete node D & edges going out of D.
No change in indegree.
Delete node F & edges going out of F.

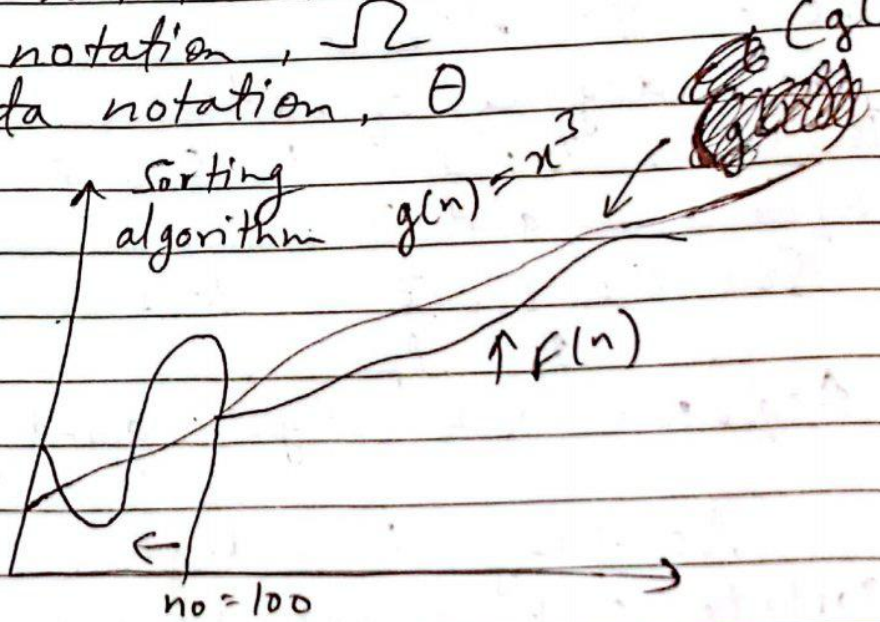


~~* Growth~~

* Growth function

Asymptotic notations

1) Big-oh Notation, O
2) Omega notation, Ω
3) Theta notation, Θ



$F(n) = O(g(n))$

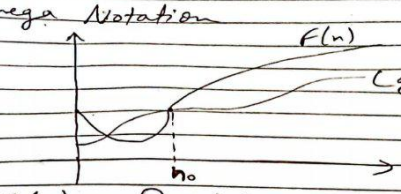
1) Big-oh Notation, O

$F(n) = O(g(n))$ Iff there are two positive constant 'c' and 'n₀' such that

$|F(n)| \leq C|g(n)|$ for all $n \geq n_0$

Thus we say $F(n)$ is the Big-oh notation of $g(n)$.

2) Omega Notation

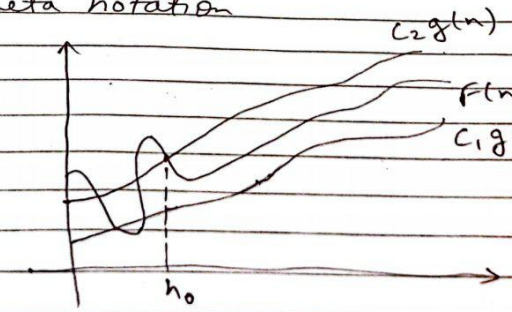


$F(n) = \Omega\, g(n)$

Iff there are two positive constants C & n_0 such that

$|F(n)| \geq C|g(n)|$ for all $n \geq n_0$

3) Theta notation



$F(n) = \Theta(g(n))$

Iff there are three positive constants C_1, C_2 & n_0 such that

$C_1 g(n) \leq F(n) \leq C_2 g(n)$ for all n:

$F(n)$ is theta notation of $g(n)$.