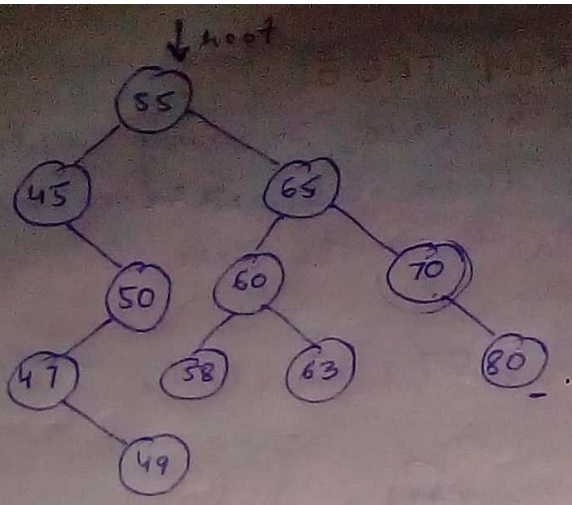


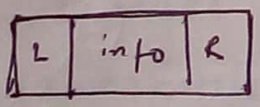
45, 47, 49, 50, 55, 58, 60, 63, 65, 70, 80

PRE-ORDER (NLR)

55, 45, 50, 47, 49, 65, 60, 58, 63, 70, 80

31 | 10 | 17

Binary Search Tree




```
typedef struct node
{   struct node *left;
    int info
    struct node * right;
} tree_type;
```

Using Recursion

Step 1. Begin

Step 2. ~~Allocate memory for a node tree_type~~

```
If root == NULL
    root → p
root → info = num
L + R == NULL
```

then root becomes = root→left and repeat the process

if no. > root→info

then root becomes = root→right and repeat the process

```
void insert ( tree_type** root, int n)
{
~~tree_type *p;~~

if (*root == NULL)
{
    *root = (tree_type *) malloc (sizeof( tree_type));
    (* root ) → left = NULL;
    (* root) → right = NULL;
    (* root)→info = n;
}
else if ( n < (*root)→info )
    insert ( &((*root)→left), n );
else if ( n >((* root)→info ))
    insert ( &(*root)→right )), n );
}
```

insert (55)
insert (65)
insert (70)

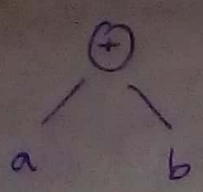
80

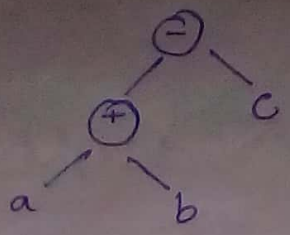
~~HIGHER KEY~~ HIERARCHY of OPERATIONS

1. Unary
2. Exponential
3. Mul, Div, Mod.
4. Add, Sub.
5. =

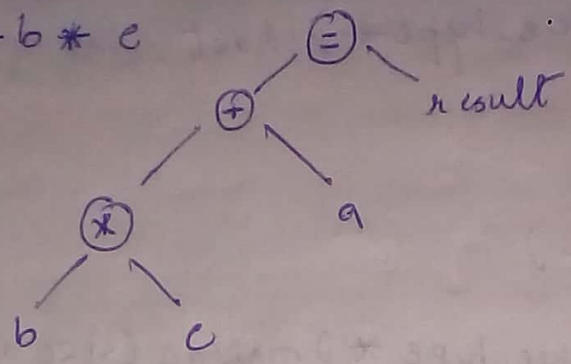
These priorities may be altered by using brackets.
In an expression tree, the operation which has the lowest priority comes at the top and the operation which has the highest priority comes at the bottom

a + b

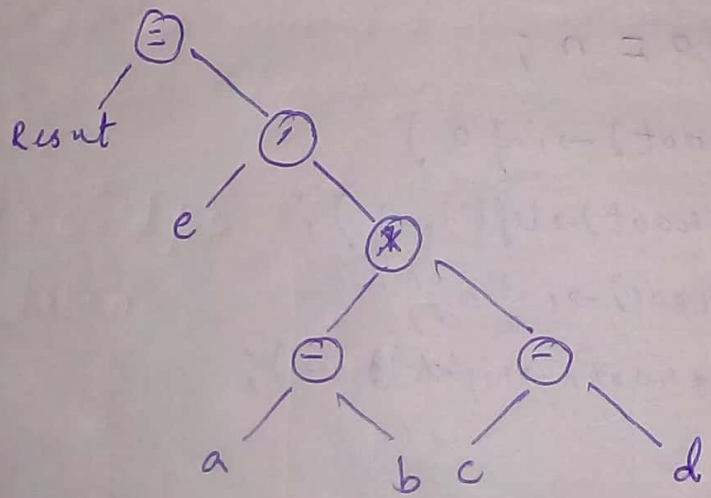




result = a + b * c

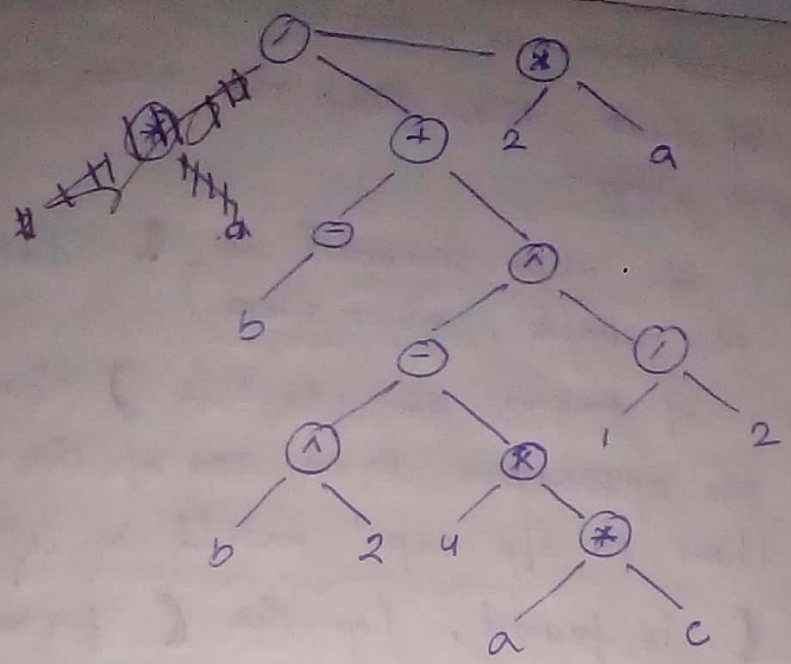


Result = (a - b) * (c - d) / e



Postfix: result ab - cd - * e / =

(-b + (b ^ 2 - 4 * a * c) ^ (1/2)) / (2 * a)



1/11/17

~~2a * b - b 2 ^ 4 a c * * - 1 2 / ^ + /~~ Postfix.

b - b 2 ^ 4 a c * * - 1 2 / ^ + 2 a * /

Converting an infix exp^n to postfix using a memory stack

~~Import~~ Input is an infix exp^n

Step 1 : Start

Step 2 : Read a token from input exp^n

Step 3 : If token is an operand directly place it to output exp^n, goto step 7

Step 4 : If token is an operator then compare its priority with priority of token which is in top of stack.

Step 4.1 : If priority of token is scanned is more than priority of operator at top of stack then push the token onto stack, goto step 7

Step 4.2 : If priority of token scanned is equal to or less than priority of operator at top of stack then kep on poping operators from top of stack and [illegible] to o/p exp^n unhill the priority of token

Scanned becomes more than priority of operator at top of stack. Now push the token onto stack and go to step 7

Step 5: If token scanned is (then push it onto the stack, go to step 7

Step 6: If token scanned is) then keep on poping the operators from top of the stack and include them to o/p exp^n until a left side bracket (is found. Pop the (from the stack but don't include into o/p exp^n

Step 7: ~~Is and of if no~~, ~~then~~ Is end of i/p exp^n. if no

Step 8: Pop the remaining entries from the stack and include them to o/p exp^n

Q. a * b + c - d / e

Token	Action	stack	o/p exp^n
a	include to o/p		a
*	push	*	a
b	include to o/p	*	ab
+	pop		ab*
	push	+	ab*
c	include to o/p	+	ab*c
-	pop		ab*c+
	push	-	ab*c+
d	include to o/p	-	ab*c+d
/	push	-/	ab*c+d
e	include to o/p	-/	ab*c+de
	pop	-	ab*c+de/
	pop		ab*c+de/-

Q. ((a+b)-(c+d))/2

Token	stack	o/p exp^n
(	(	
(	((	
a	((	a
+	((+	a
b	((+	ab
)	((	ab+
	(	ab+
-	(-	ab+
(	(-(	ab+
c	(-(	ab+c
+	(-(+	ab+c
d	(-(+	ab+cd
)	(-(	ab+cd+
	(-	ab+cd+
)	(	ab+cd+-
		ab+cd+-
/	/	ab+cd+-
2	/	ab+cd+-2
		ab+cd+-2/

H.W | Root of quad. f^n.

Q. Deleting a node from binary search tree

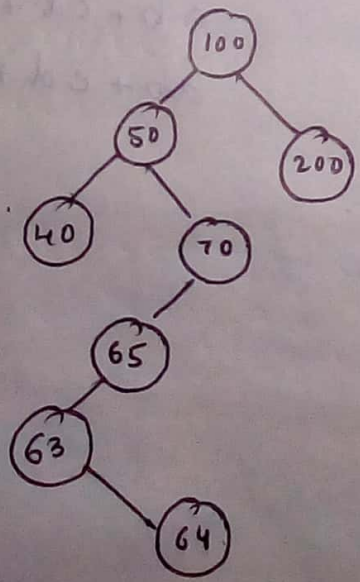
Step 1 : Begin

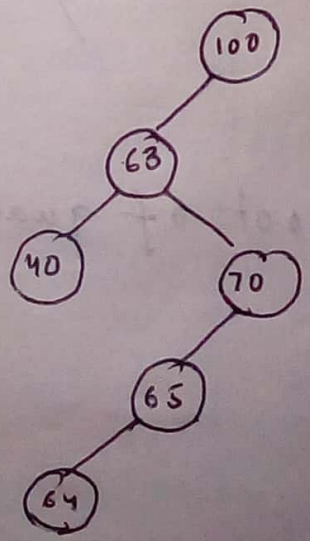
Case 1 : The node to be deleted does not have any child, then we'll directly delete the node and update the field of parent through will it was connected to NULL.

Case 2 : If the node to be deleted has one child then after deleting that node, the child of node deleted will be connected to parent of node deleted through same relationship through which node deleted was connected to its parent.

2/11/17

Case 3 : The node to be deleted has 2 children then we find inorder successor of that node and then copy data from inorder successor to the node which we want to delete and then we delete inorder successor





Evaluating a postfix exp^n using stack

Input is a postfix exp^n

step 1 : Start

step 2 : Scan a token from input exp^n

step 3 : If token is an operand then push it onto stack goto step 6

step 4 : If token is a unary operator then pop an entry from the stack, do operation specified by unary operator and push the result back to stack, goto step 6

step 5 : If token is a binary operator then pop two entries entry1 and entry two respectively from the stack. Perform operation specified by operator Op entry 2 by entry 1. Push the result back to stack.

step 6. Is end of i/p exp^n. if yes then goto step 7, otherwise goto step 2.

step 7. Pop the final entry from the stack which becomes result of the calculations.

step 8. Stop

4 * 3 + 6/2

4 3 * 6 2 / +

Token	Action	Stack	o/p exp^n
4	push	4	
3	push	4 3	
*	pop	4	
	pop		
	push	12	
6	push	12 6	
2	push	12 6 2	
/	pop	12 6	
	pop	12	

+	push	12 3
	pop	12
	pop	3
	o/p	

15 //

-2+3

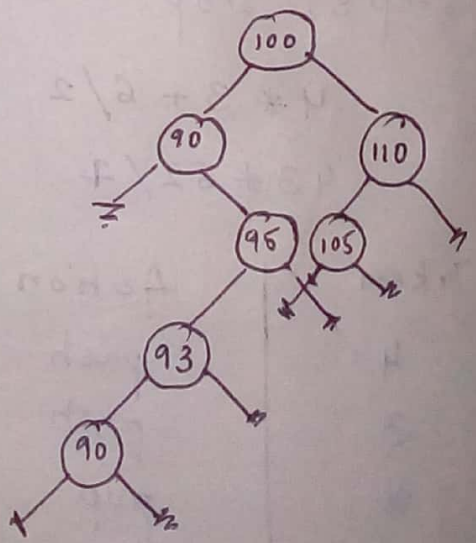
2-3+

①	2	push	2
②	-	pop	
		push	-2
③	3	push	-2 3
④	+	pop	-2
		pop	
		o/p	

1 //

Traversing a Binary Search Tree in Inorder.

```
void inorder (tree_type * rt)
{
    if (rt == NULL)
        return ;
    else
    {
        inorder (rt→left);
        printf (rt→info) ;
        inorder (rt→right);
    }
}
```



```
pre-order
void preorder ( tree_type *rt )
{   if (rt == NULL)
        return ;

    else
    {  printf ("%d", rt->info) ;
       preorder (rt->left) ;
       preorder (rt->right) ;
    }
}
```

6|10|17

Q. Assuming that we already have a BST with a root. Write a f^n to count total no. of nodes. No. of nodes should be printed from the main function

Q. Write a f^n to count total no. of leaf nodes.

Q. Count total no. of nodes which are connected to the parents through right relationship.

B-TREE

A multiway tree where in every node we can store more than one data items (primary keys)

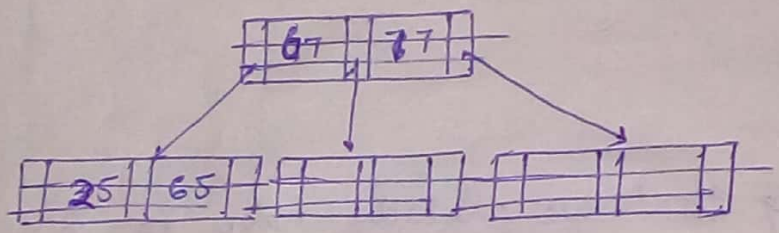
So a B-Tree of order n is a multiway tree where in every node n-1 data items may be stored and n addresses can be generated from every node.

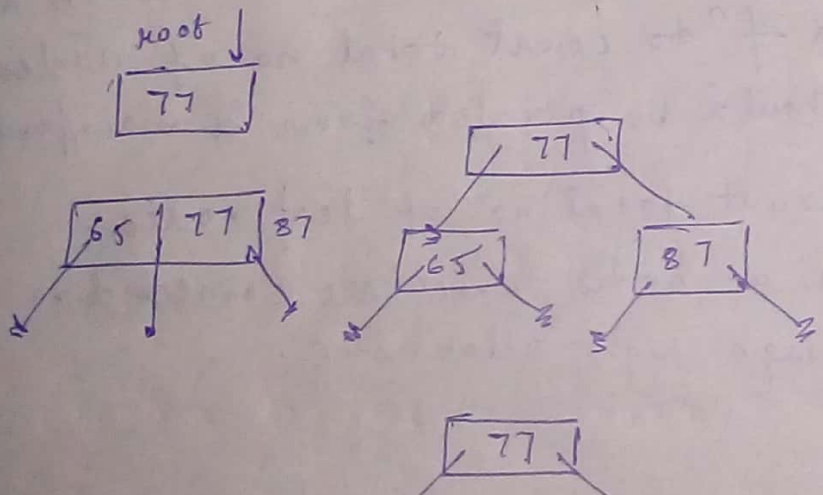
A B.S.T may be considered as a B-Tree of order 2. So in every node at the max. 1 data items is stored and 2 addresses are generated.

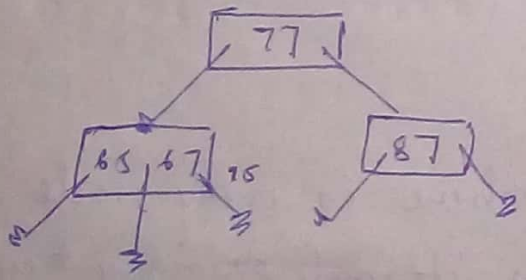
In a B-Tree, a new data item is, first of all, always included in bottom node. Once the bottom node is full, it gets splitted and the data moves in upward dirⁿ.

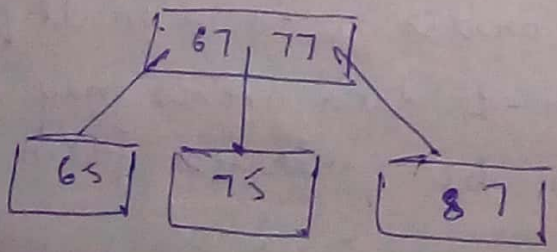
Q. Draw a B-Tree of order 3 with following data

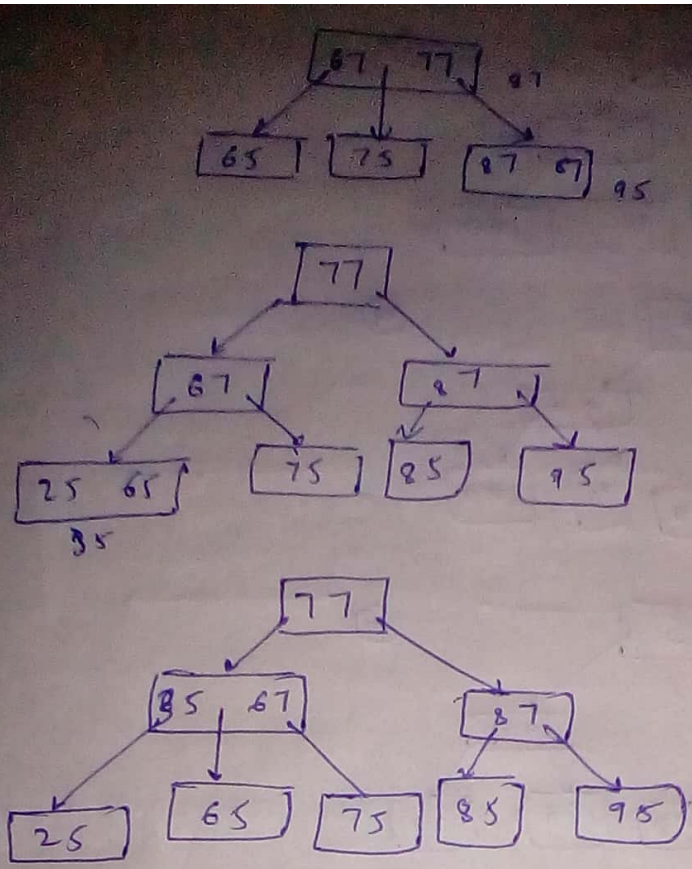
77, 67, 87, 65, 75, 85, 95, 25, 35





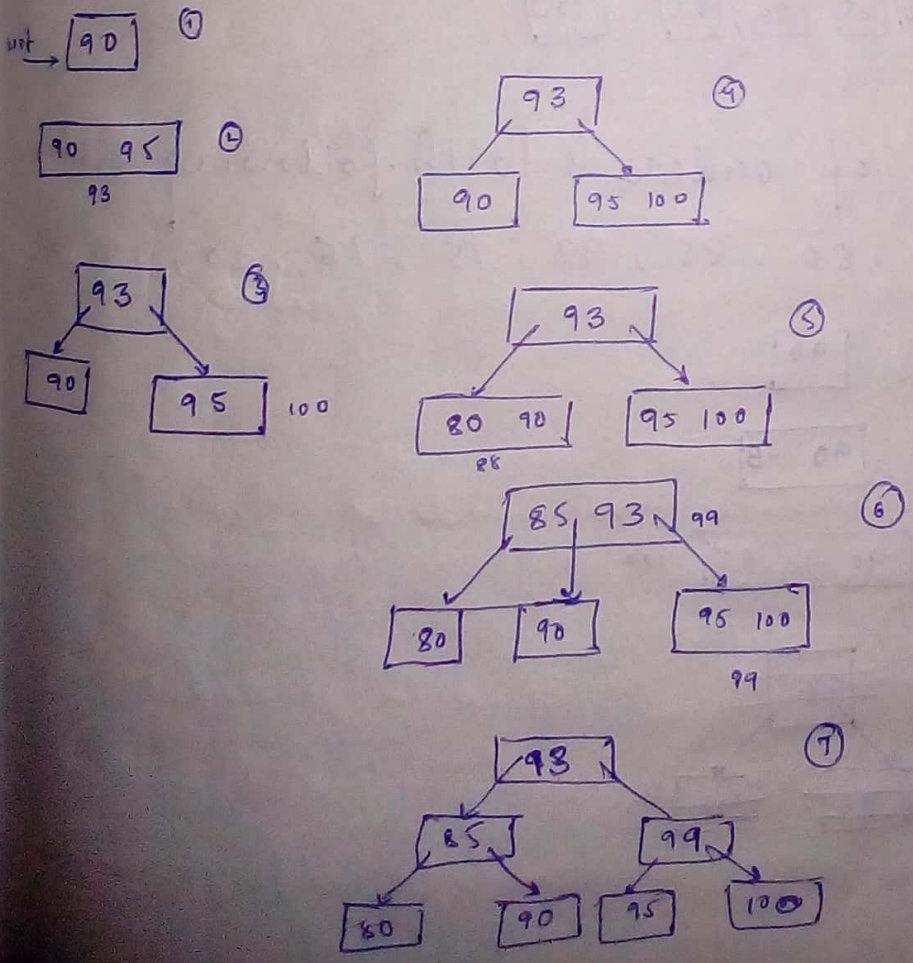


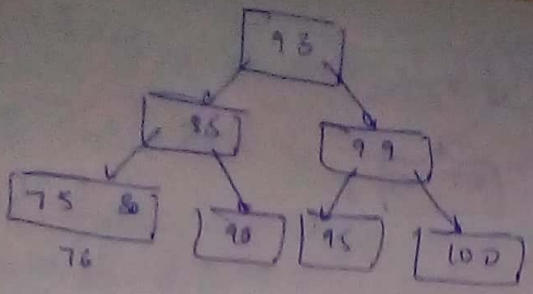


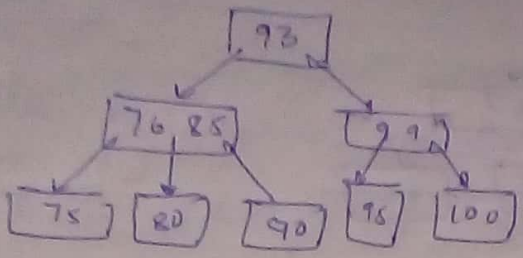


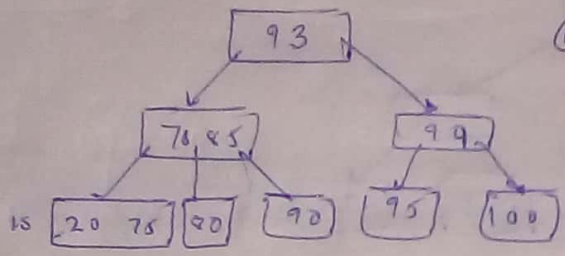
Q. Draw a B-Tree of order 3 with following data

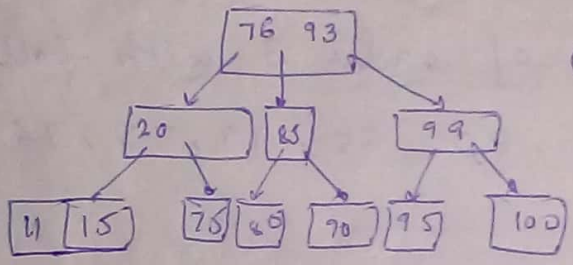
90, 95, 93, 100, 80, 85, 99, 75, 76, 20, 15, 11





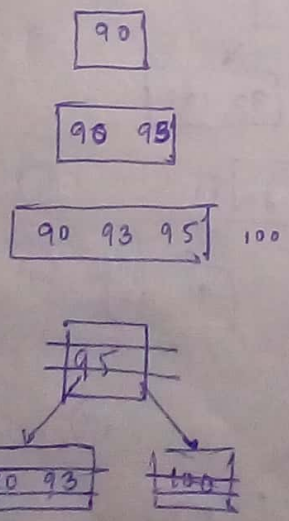


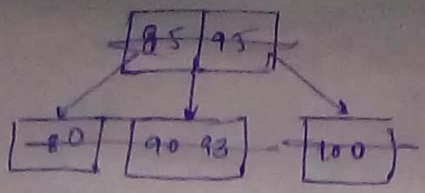


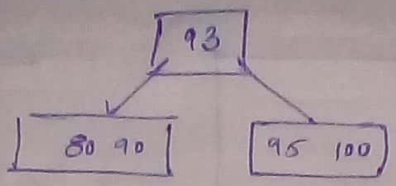


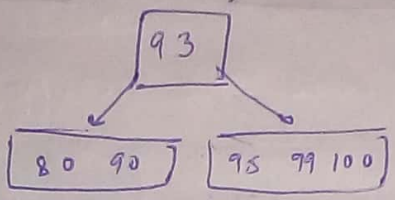
Create a B-Tree of order 4 with following data

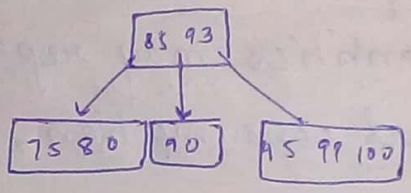
90, 95, 93, 100, 80, 85, 99, 75, 76, 20, 15, 11

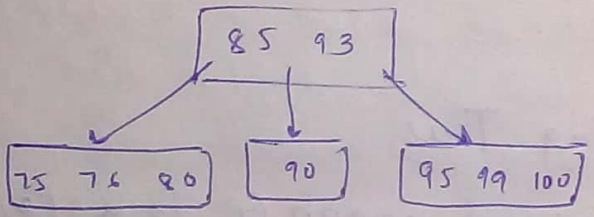


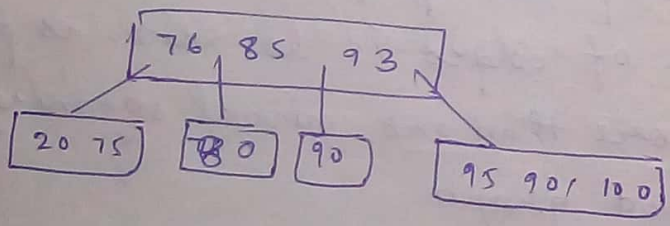


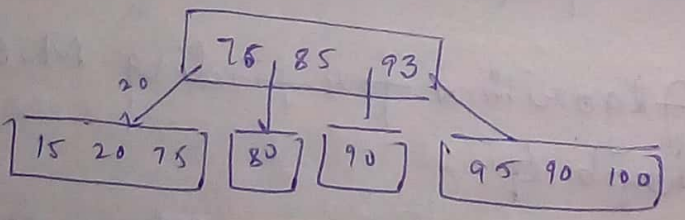


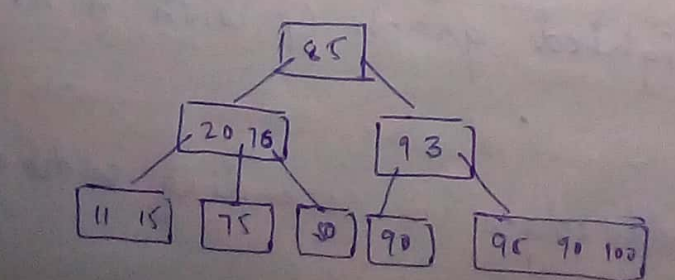








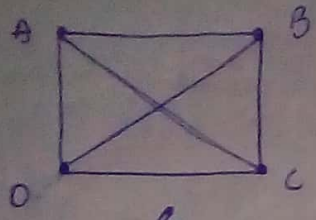




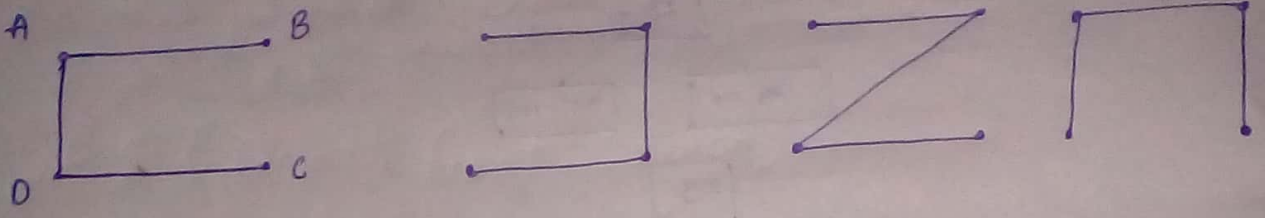
Scanned by CamScanner

7/11/17

Minimal Spanning Tree



G



Weighted Graph

- A graph where +ve quantities are assigned to all edges. These +ve quantities may represent something such as distance, fuel consumption, time taken b/w the nodes.

Minimal Spanning Tree

- Tree T of a weighted graph G is a tree so that sum of weight of edges is as min. as possible. There may be more than one minimal spanning trees for a given weighted graph

KRUSKAL's Algorithm for finding M.S.T for a weighted graph G.

Input is a weighted graph G with n nodes.

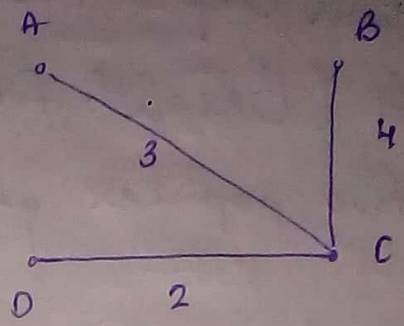
Step 1: Start

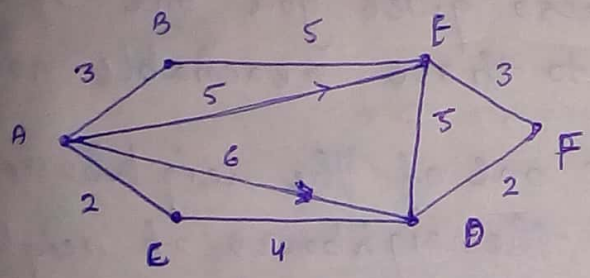
step 2: Arrange all edges in ascending order of weights.

Step 3: Starting with only vertices connect n-1 edges with min. weights so that no cycles produced

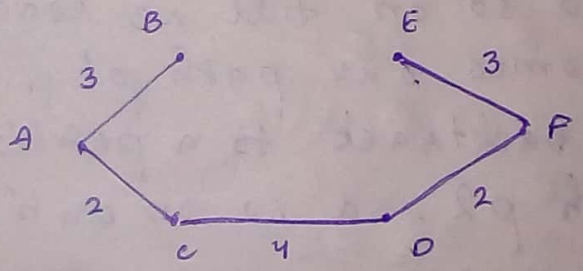
Step 4: Stop.

edge	weight
CO/	2
AC/	3
AD	4
BC/	4
AB	5
BD	6





AC	2
DF	2
AB	3
EF	3
CD	4
AE	5
BE	5
DE	5
AD	6



GRAPH TRAVERSAL Algorithms

DFS (Depth first search)
BFS (Breadth first search).
are two popular graph traversal algorithms.

DFS (Depth first search)

- Used to find what are the reachable nodes from a given source node. It also gives you one of the possible path to reach to those reachable nodes.

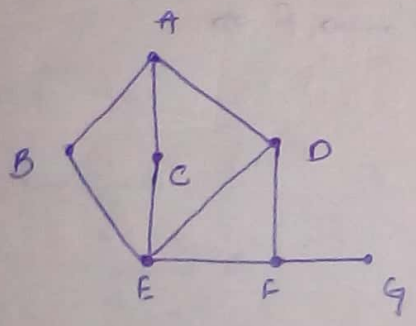
On DFS first we visit one of the neighbours of source node, then one of the neighbours of neighbour and so on till we reach to a dead end. This becomes our path p1. After that if require we backtrack to a particular node to find another path p2 and so on until all the reachable nodes from a source node are covered. Input is a connected graph G.

Step 1:
Step 2: set all the vertices of graph to ready state, i.e. status equal to 1.
Step 3: Push the source node onto stack and change its status to 2, that is waiting state.
Step 4: Repeat step 5 and 6 until stack is empty.

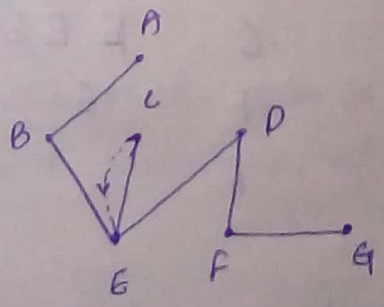
Step 5: Process a node from top of a stack and change its status to 3. i.e processed state.

Step 6: Examine all the neighbours of node processed
If status of neighbour is 1, then push it onto stack and change its status to 2.
Otherwise if status of neighbour is 2, then push it onto stack and delete its previous entry from the stack.
Otherwise if status of neighbour is 3, then just ignore it.

Step 7: stop

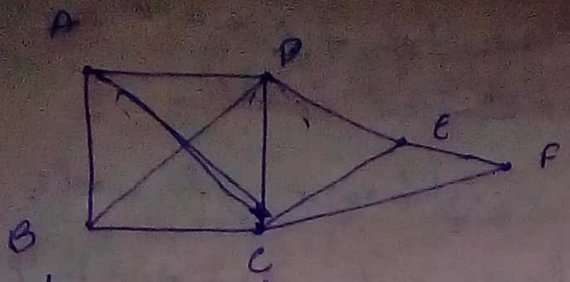


node processed	stack ←
	A
A	B C D
B	E C D
E	C D F ~~C~~ ~~D~~
C	D F
D	~~F~~ F
F	G
G	

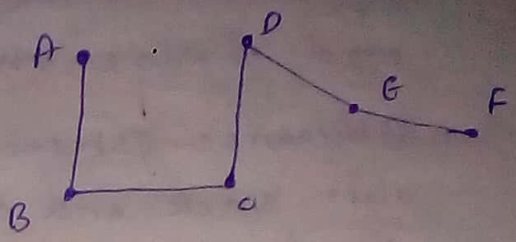


P_1 => A → B → E → C

P_2 => D → F → G



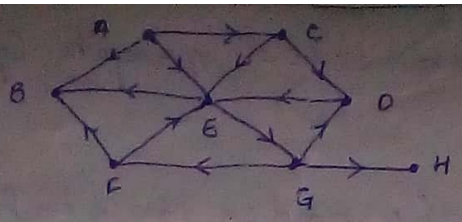
node.	stack ←
	A
A	B C D
B	C D ~~∅~~ ~~∅~~
C	~~D~~ ~~E~~ F ∅
D	E ~~E~~ ~~F~~
E	F ~~∅~~
F	



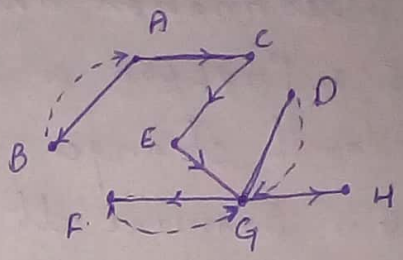
P1 ⇒ A → B → C → D → E → f ∞

node	stack ←
	A
A	D ~~C~~ B
D	~~B~~ C E ~~E~~ ~~∅~~
B	C ~~∅~~ E
C	~~E~~ E ~~E~~
F	E
F	

8/11/17



node	stack ←
	A
A	B C E
B	E E
E	E D ~~E~~
E	G D
G	D F H ~~D~~
D	F H
F	H
H	



l_1 ⇒ A → B

l_2 ⇒ C → E → G → D

l_3 ⇒ F

l_4 ⇒ H

BFS (Breadth first Search)

A graph traversal algorithm to find a path b/w a source node & destination node with min. no. of nodes in b/w.

In BFS first we process all the neighbours of source node, then we process all the neighbours of neighbours and so on till we reach to the destination node, then we backtrack from destination node to source node to find the path with minimum no. of nodes in b/w.

Input is a connected graph G

Step 1: Start

Step 2: Set all the nodes to status = 1 i.e ready state.

Step 3: Insert the source node into queue.

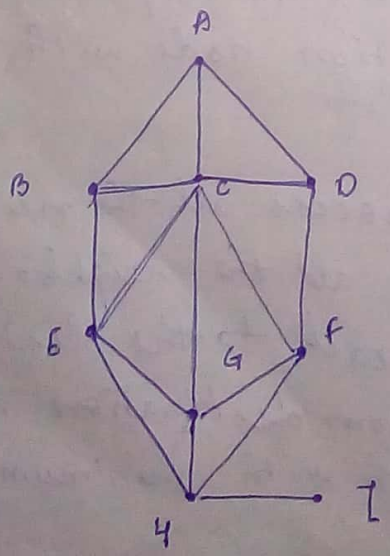
Step 4: Repeat step 5 and 6 untill we reach to destination node or queue becomes empty.

Step 5: Process a node from front of the queue and change its status to 3.

Step 6: Examine all the neighbours, if status of neighbour is 1, then insert it in queue, otherwise ignore it.

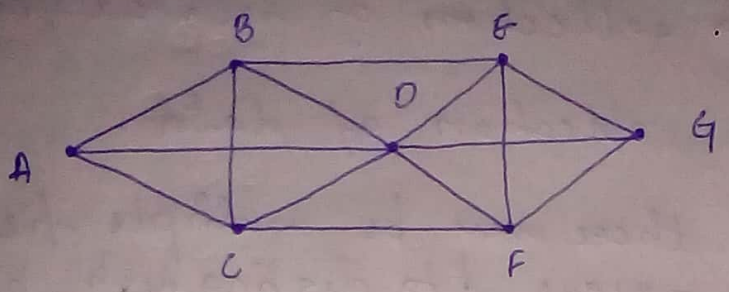
Step 7: If we have reached to destination node then backtrack to source node to find the resultant path with min no. of nodes in b/w, otherwise queue becomes empty without processing destination node means there is no path b/w source node and destination node.

Step 8: Stop.



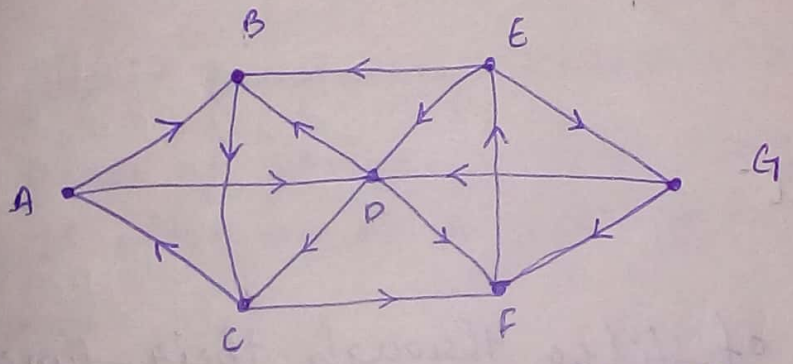
	A	B	C	D	E	G	F	H	I
origin	ϕ	A	A	A	B	C	C	E	H

A → B → E → H → I



queue	A	B	C	D	E	F	G
origin	ϕ	A	A	A	B	C	D

A → D → G



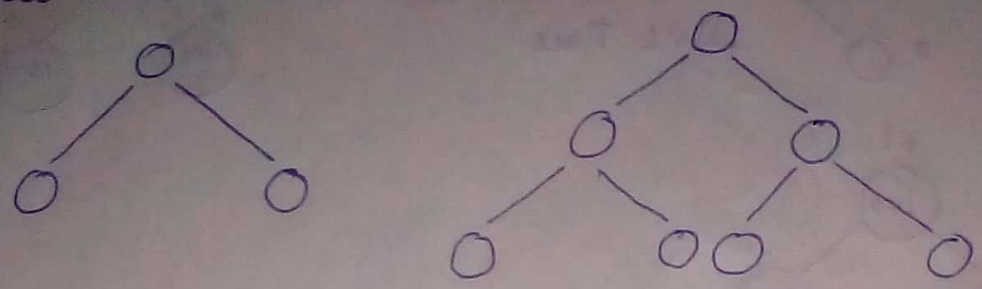
queue	A	B	D	C	F	E	G
origin	ϕ	A	A	B	D	F	E

A → D → F → E → G

A Completely Balanced Tree

for Every node height of left subtree and right subtree are same.

e.g



for having a completely balanced tree of level n. we require $2^n - 1$ node.

However it is difficult to acheive a completely balanced tree because we may not have desired no. of nodes.

Goal of balancing a tree is to minimize no. of levels or height of the tree so that there are less refrences to memory stack.

We always try to acheive nearly balanced trees.

*BALANCING FACTOR

Height of the right subtree - height of the left subtree for a particular node.

For a completely balanced tree, balancing factor should be 0 for all nodes.

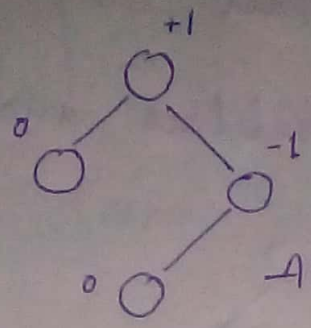
AVL Tree

(Adelson Velski Landis)

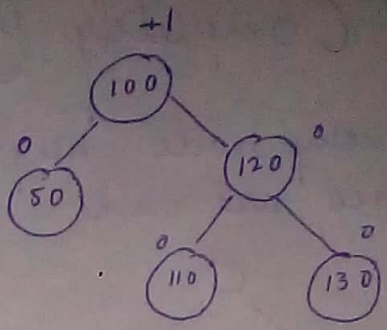
A self balanced BST where balancing factor for every node is either +1, 0 or -1. It means for every node the diff. b/w right subtree & left subtree should not be

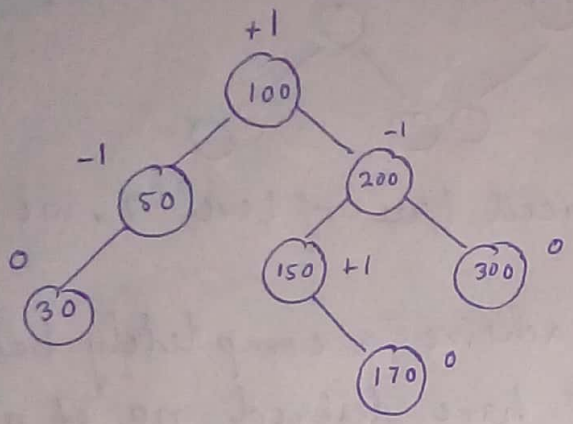
more than 1.

e.g.



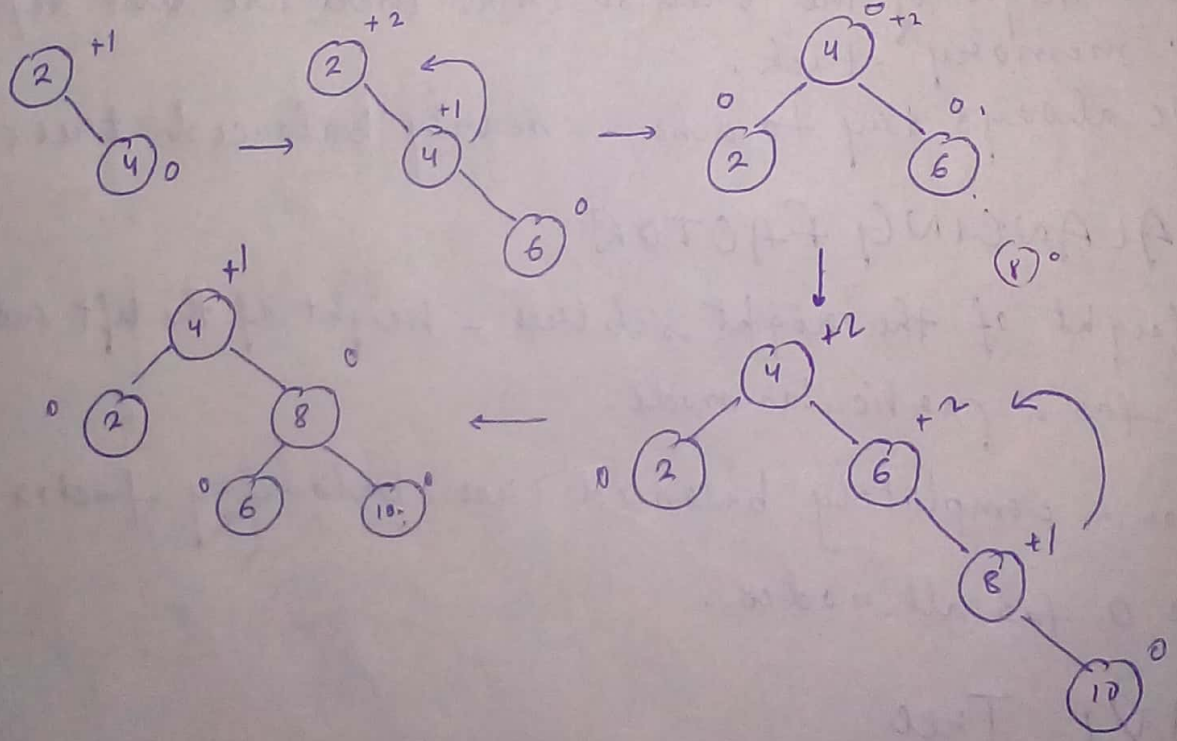
AVL Tree



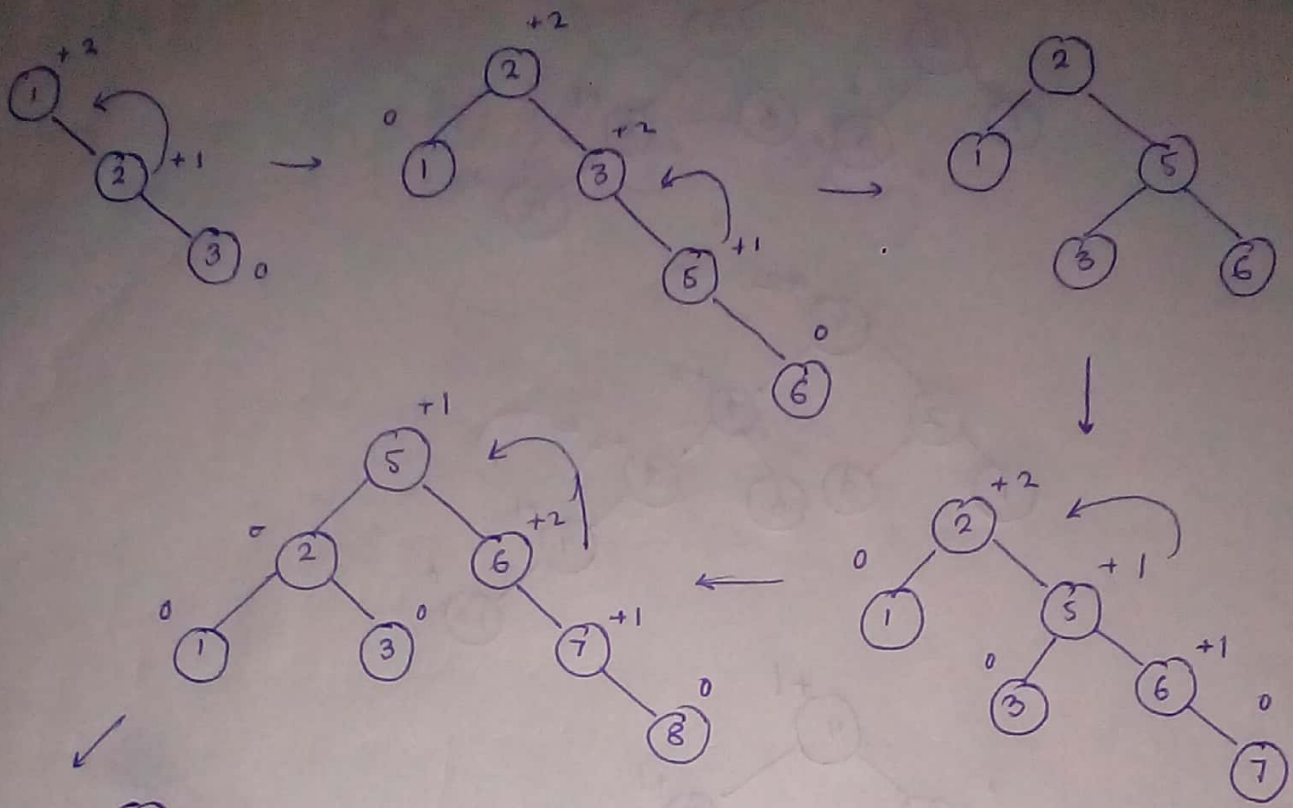


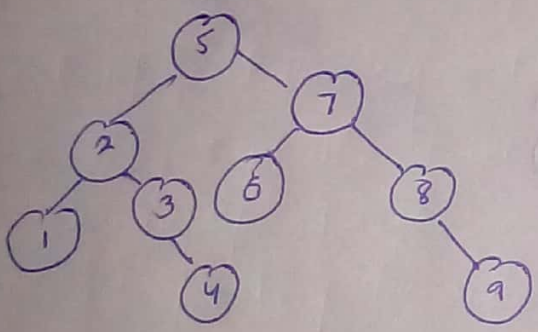
Q.1) Draw an AVL tree with following data.

2, 4, 6, 8, 10

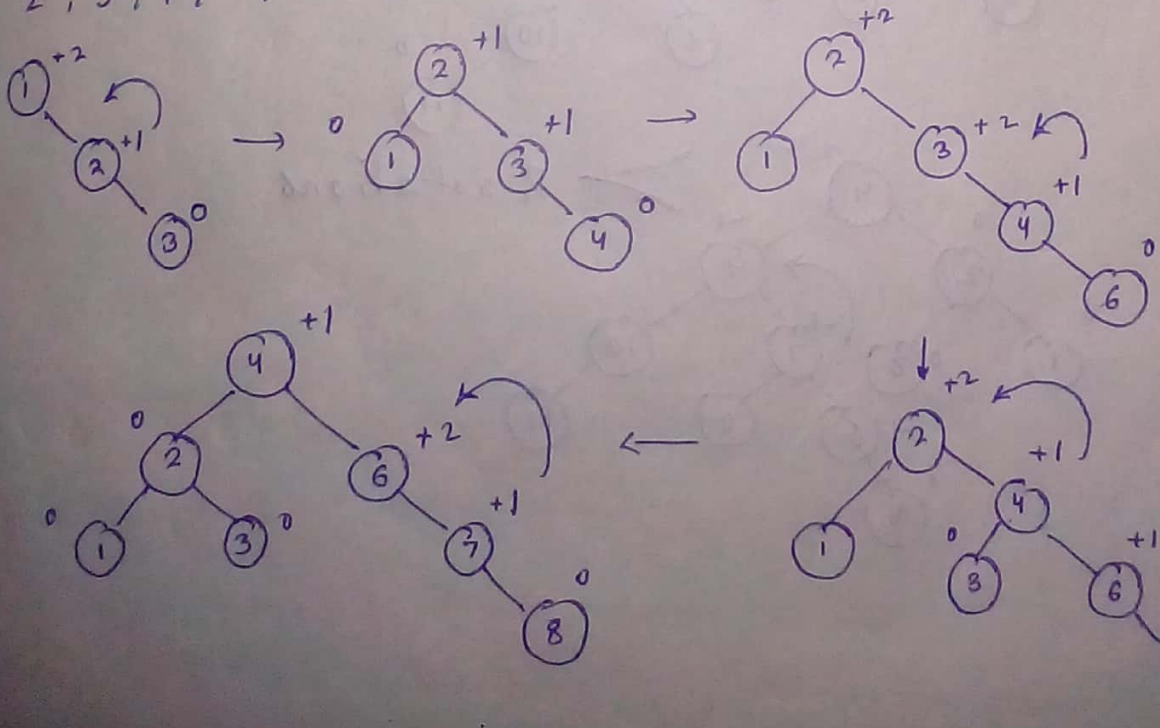


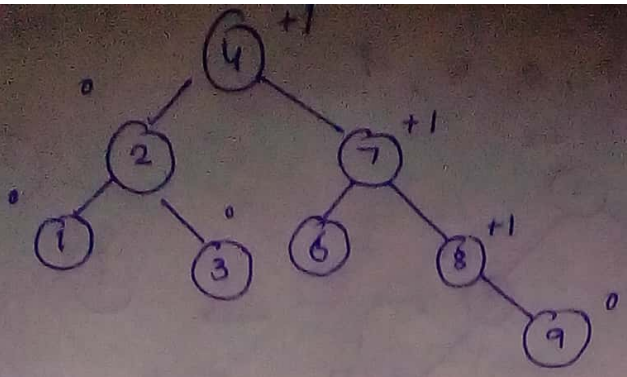
ü) 1, 2, 3, 5, 6, 7, 8, 4, 9

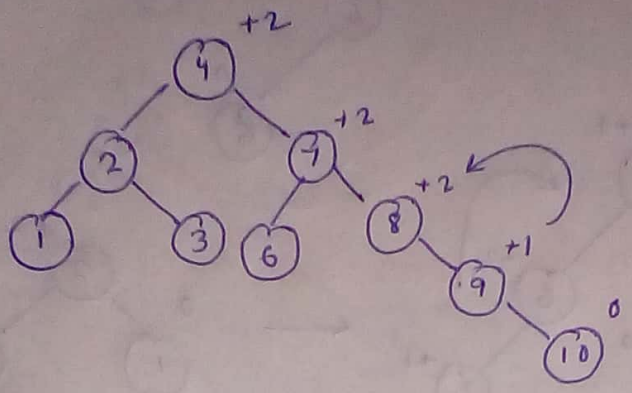


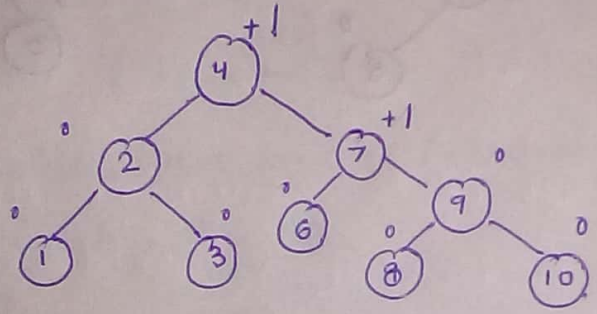


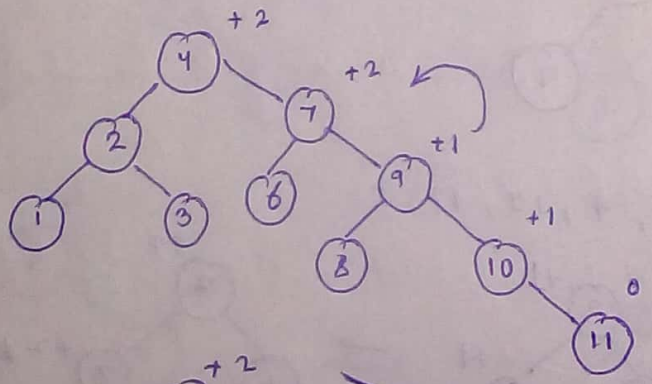
ü) 1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 5

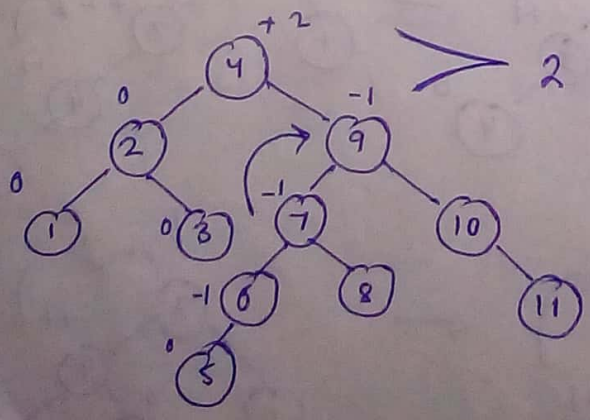




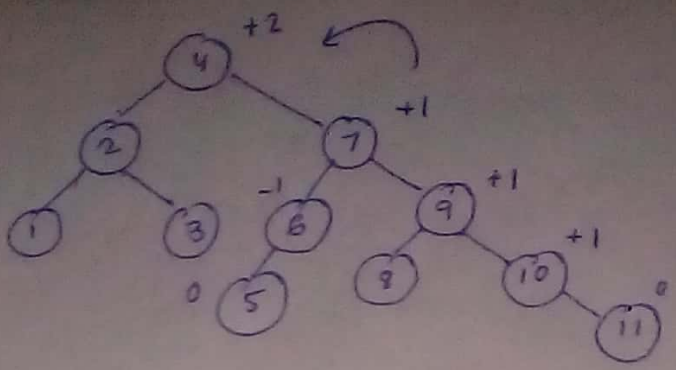


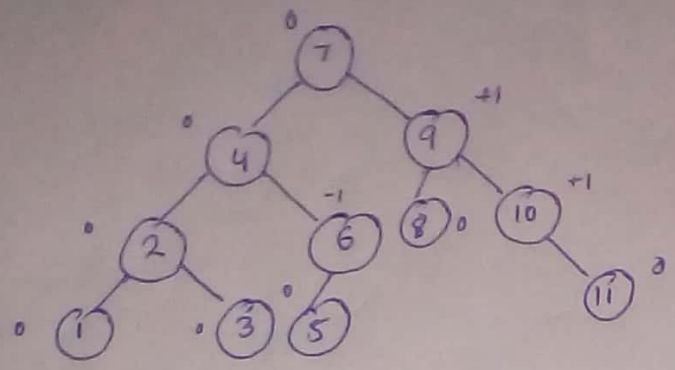






2 rotations





AVLTREE ISOK