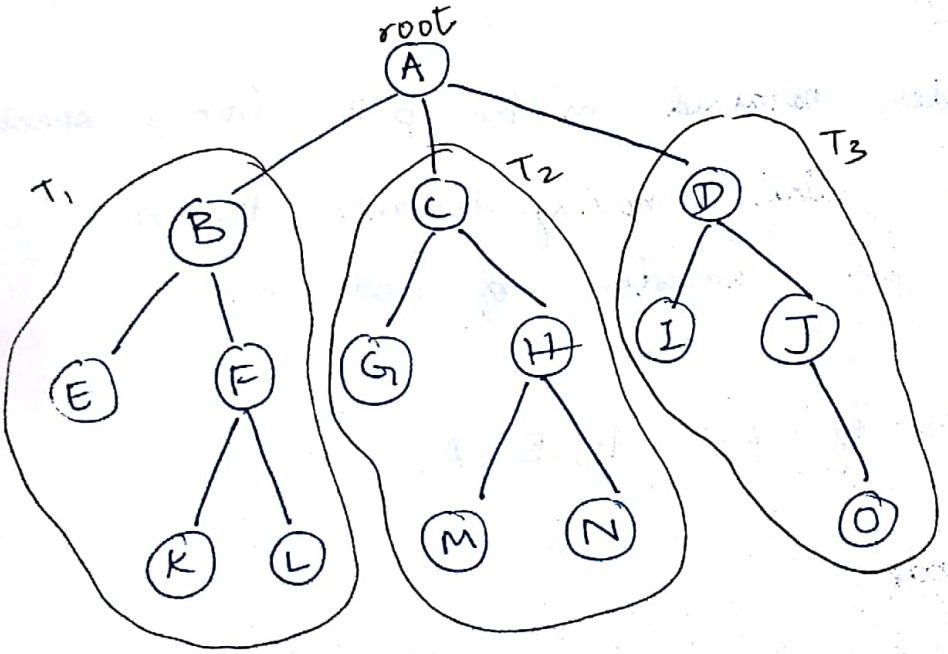
UNIT 4

Trees

A tree is a finite set of one or more nodes that exhibits the parent and the child relationship such that

- There is a special node called as root node
- The remaining nodes are partitioned into disjoint subsets denoted by $T_1 T_2 \ldots T_n$ which are termed as subtrees



Tree is non-linear non-primitive DS where data is stored in a heirarchial fashion.

Degree of a node

The number of subtrees of a given node is termed as the degree of a node.

degree of node A = 3

E = 0

Sibling

2 or more nodes having a common parent are called siblings.

Ancestors

The nodes obtained in the path from a specific node x while moving towards the root node are termed as ancestors of node x.

Ancestors of L = F, B, A.

Leaf node:

The nodes in a tree which have degree of 0.

E, K, L, G, M, N, I, O.

Level of a node

The distance of a node from the root node

the no of edges passed through root node to reach a node.

A = L0

BCD = L1

EFGHJI = L2

KLMNO = L3

Height of a tree

The no of edges on the longest downward path b/w the root and the leaf.

Last level + 1 height = 4 ✓

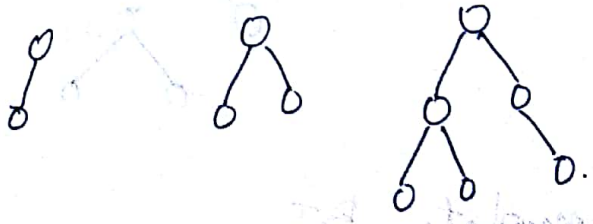
if ~~nodes~~ ~~are~~ no of nodes processed, he = 4 ✓

if no edges, he = 3 ✗

Binary tree

A tree in which each node has either 0, 1 or 2 subtrees.

Eg:-



(OR) (for marks):

A binary tree T is defined as a finite set of elements called nodes such that, ~~T is empty~~

i) T is empty (called empty tree or null tree)

ii) T contains a distinguished node 'R' called root of the node and the remaining nodes of T

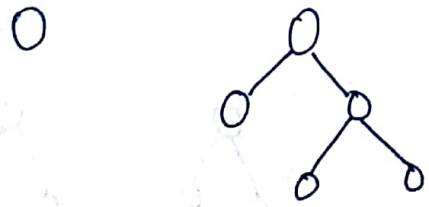
form an ordered pair of disjoint binary trees T_1 and T_2 respectively where T_1 is left subtree and T_2 is ~~left~~ right subtree.

Types of binary trees

1. Strictly binary tree
2. Complete BT
3. Almost complete BT
4. B search T
5. expression T
6. Skewed BT

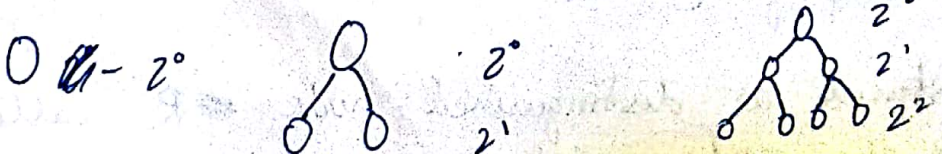
Strictly BT

If outdegree of every node in the tree is either 0 or 2 then it is Strictly BT.



Complete BT

A strictly BT having 2^i nodes at any given level i.

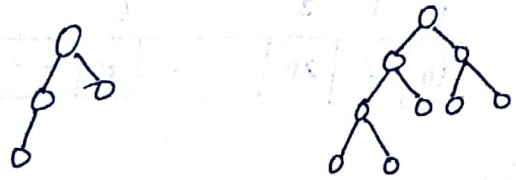


Almost complete BT (Heap) ☆☆☆.

It is CBT with the following properties

i) Apart from the last level, the other remaing levels should have 2^i nodes

ii) The last level the nodes should be left filled.



Skewed BT

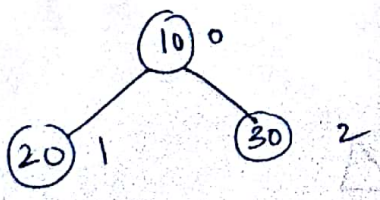
If ~~the~~ tree is built either on only one side, it's called skewed tree (left or right skBT)

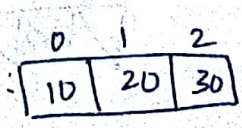
Repr of a BT

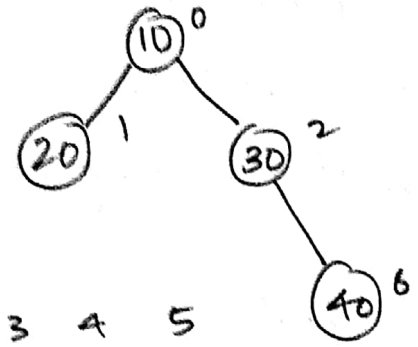
A binary tree can be represented in 2 ways

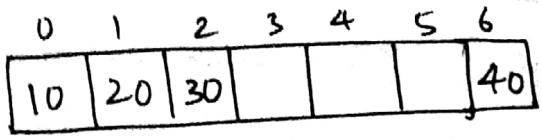
i) Array repr
ii) List repr

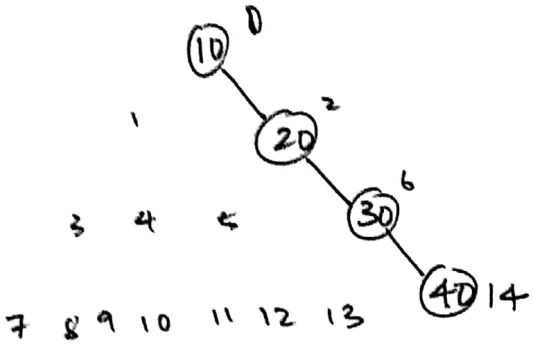
Array repr

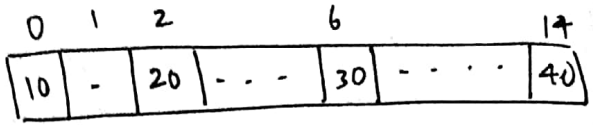










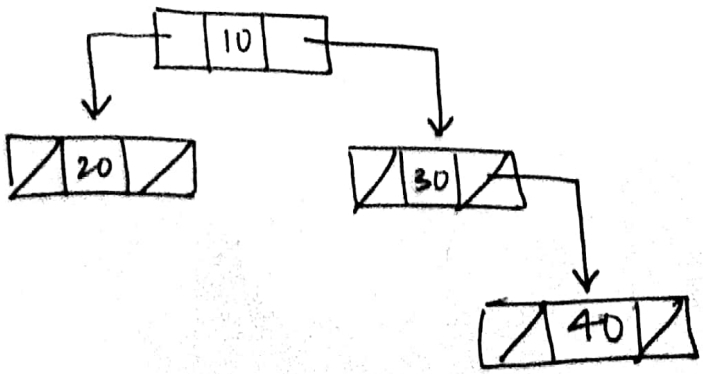


If tree is not complete, then wastage of memory.

List repr

In the list repr of a BT, we create a DLL where in each node has 3 fields.

1. info field - stores the info or key
2. llink - holds address of left subtree
3. rlink - holds address of right subtree.



The following struct def used to repr a node of BT

```
struct node
{
    int info;
    struct node *llink;
    struct node *rlink;
};
type def struct node * NODE;
```

root is a variable of type NODE which holds the address of root node

Operations on BTs

traverser opn ★☆★

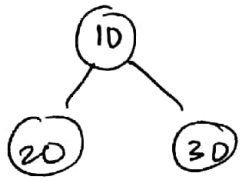
Traversal is the most common operation wherein we visit each node of a binary tree exactly once. There are 3 ways of traversal.

i) Preorder
ii) In order
iii) Postorder.

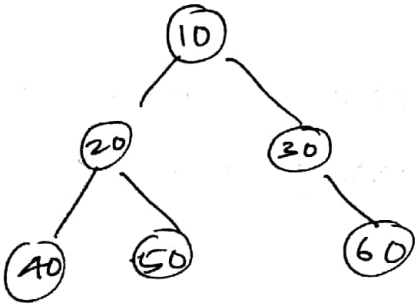
i) Preorder traverser.

The recursive definition to perform traversal on the binary tree is as follows.

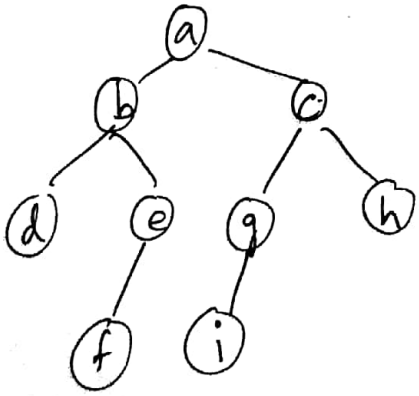
1. Visit the root node
2. recursively traverse the left subtree in the preorder.
3. recursively traverse the right subtree in the preorder.



i) 10 20 30
ii) 20 10 30
iii) 20 30 10



i) 10 20 40 50 30 60
ii) 40 20 50 10 30 60
iii) 40 50 20 | 60 30 10



i) a b d e f c g i h

~~ii) d b f e a i g c h~~

ii) d b f e a i g c h

iii) d f e b i g h c a.

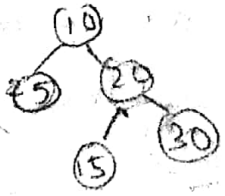
Recursive function to perform preorder traverser

```
void preorder (NODE root)
{
    if (root != NULL)
    {
      printf ("%d\t", root -> info);
      preorder (root -> llink);
      preorder (root -> rlink);
    }
}
```

(ii) Inorder traverser

The recursive definition to perform the inorder traversal on a binary tree is as follows

1. recursively visit the left subtree in inorder
2. visit the root node
3. recursively visit the right subtree in inorder




```
void inorder (NODE root)
{
    if (root != NULL)
    {
      inorder (root -> llink);
      pf("%d \t", root -> info);
      inorder (root -> rlink);
    }
}
```

iii) post order traverser

The recursive def^n to perform post order traverser is as follows ⇒

1. Recursively visit the left subtree in post order
2. Recursively visit the right subtree in post order
3. Visit the root node.

```
void postorder (NODE root)
{
    if (root != NULL)
    {
        postorder (root → llink);
        postorder (root → rlink);
        pf("%d\t", root → info);
    }
}
```

Assignment Q

Q. Write an iterative function to perform traversal on binary tree.

Construct a binary tree given the following traverser

i) preorder : a b d g h c e i f

inorder : g d h b a e i c f.

nder traverser

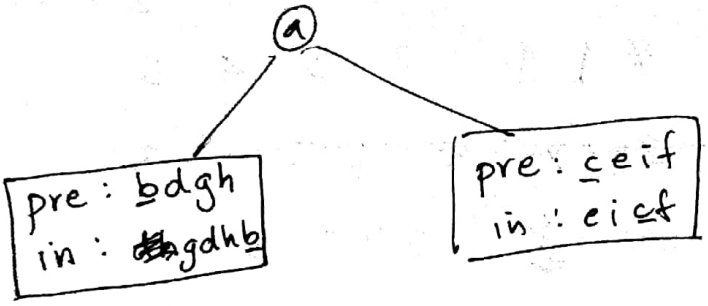
post order

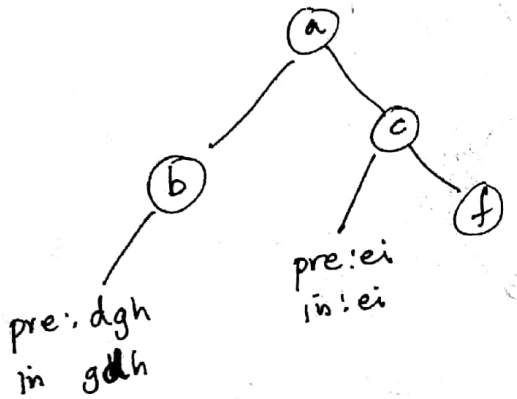
post order

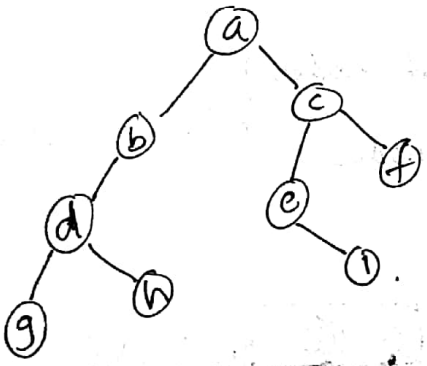
form traversal

lowing traverser.

i)



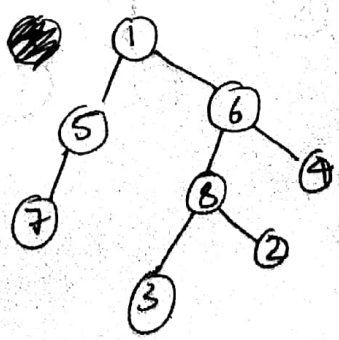




ii)

post : 7 5 3 2 8 4 6 1

in : 7 5 1 3 8 2 6 4



iii)

~~* $34~~ *$234

2 * 3 $4

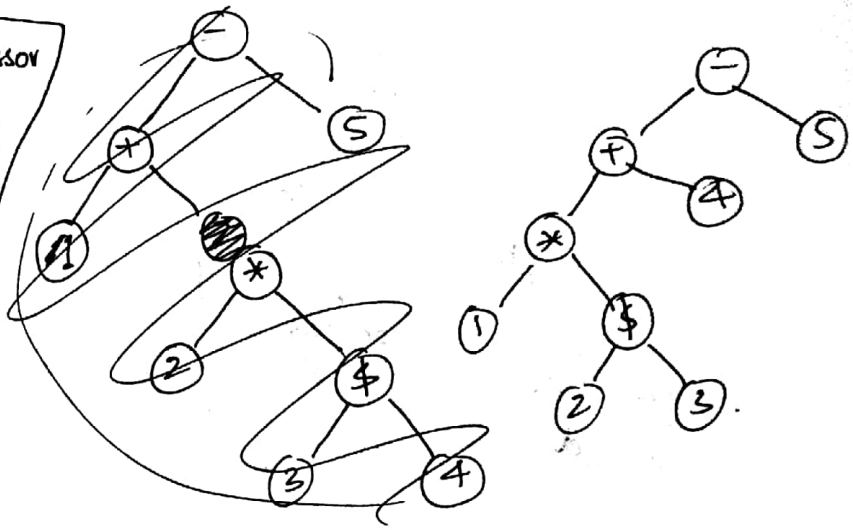
pre: − + * 1 $ 2 3 4 5

in : 1 + 2 * 3 $ 4 − 5

1 * 2 $ 3 + 4 − 5

preorder predecessor node of 2 is $

successor is 3

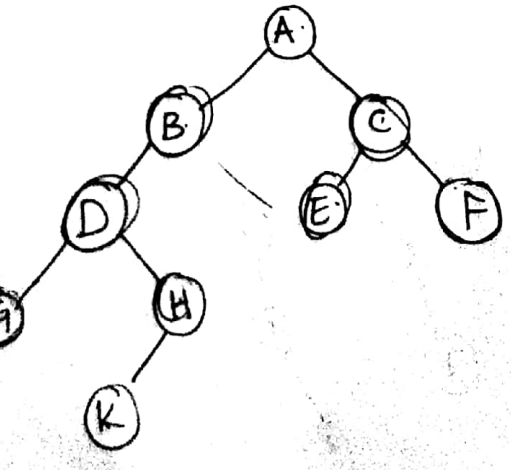


operators are non-leaf nodes
operands are leaf nodes
} AKA expression tree.

iv)

~~post~~ pre : A B D G H R C E F

post : G K H D B E F C A



A

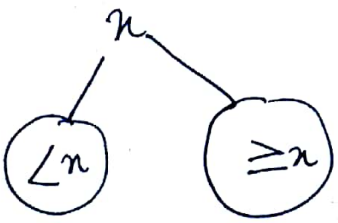
x = right child = predecessor of root node in post

y = left child = $ successor of root in pre.

if at any point x == y, tree is not unique.

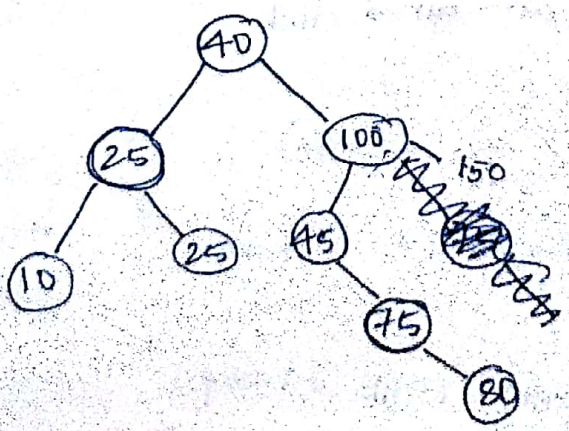
Binary search tree

is a binary tree in which for every node x, the left subtree of x will always be less than x and right subtree of x will always be greater than or equal to x.



Create a binary search tree for the given set of elements.

40 100 25 45 25 75 10 80



in: 10 25 25 40 45 75 80 100

A C func to create BST

```
NODE createBST(NODE root, int key)
{
    NODE temp, prev, cur;
    temp = (NODE) malloc(sizeof(struct node));
    temp->info = key;
    temp->llink = temp->rlink = NULL;
    if (root == NULL)
        return temp;
    prev = NULL;
    cur = root;
    while (cur != NULL)
    {
        prev = cur;
        if (key < cur->info)
            cur = cur->llink;
        else
            cur = cur->rlink;
    }
    if (key < prev->info)
        prev->llink = temp;
    else
        prev->rlink = temp;
    return root;
```

Program 7

To create BST and to perform the following operation
insert, delete, pre, in, post.

```
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int info;
    struct node *llink;
    struct node *rlink;
}; typedef struct node *NODE;

//create BST func

// pre, in, post

NODE delete (NODE root, int key)
{
  NODE temp

  if (root == NULL)
      return root;
  if (key < root->info)
      root->llink = delete (root->llink, key);
  else if (key > root->info)
      root->rlink = delete (root->rlink, key);
```

```
else
{
    if(root->llink == NULL)
    {
        temp = root->rlink;
        free(root);
        return temp;
    }
    else if(root->rlink == NULL)
    {
        temp = root->llink;
        free(root);
        return temp;
    }
    temp = inordersuccessor(root->rlink);
    root->info = temp->info;
    root->rlink = delete(root->rlink, temp->info)
}
return root;
}
```

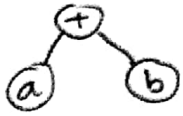
```
NODE inordersuccessor(NODE root)
{
    NODE cur = root;
    while(cur->llink != NULL)
        cur = cur->llink;
    return cur;
}
```

Expression tree

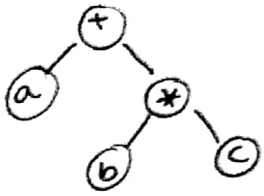
An expression tree is a binary tree wherein all the operands of the expression will be the leaf nodes and the operators in the expn will be non leaf nodes (internal nodes).

Ex:-

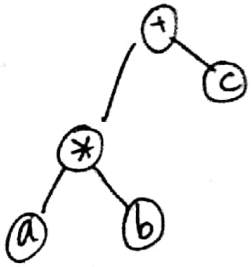
a+b



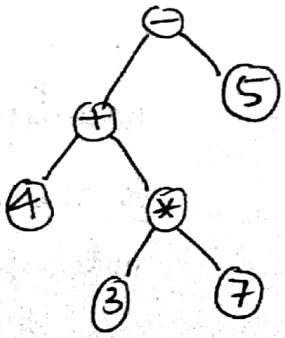
a+b*c



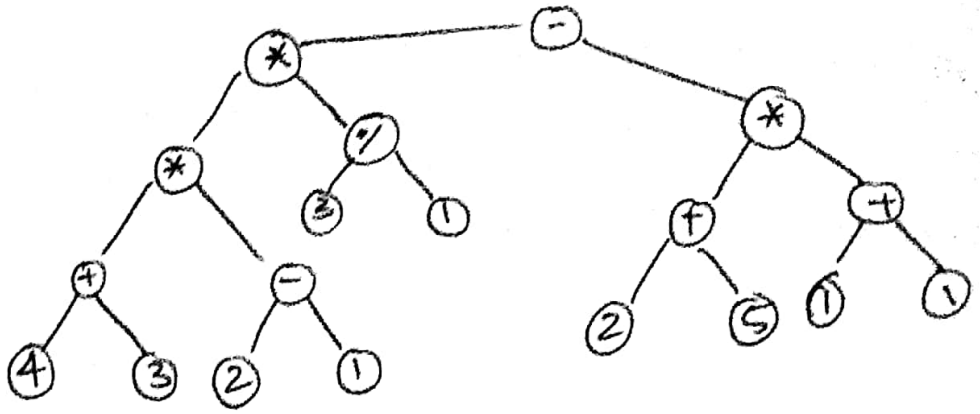
a*b+c



4+3*7-5



$((((4+3)*(2-1))*(3/1))-((2+5)*(1+1)))$



Algorithm to convert given infix expn to an ~~BST~~. Expression tree.

Step 1 : Scan the given infix expression from left to right

Step 2 : Create a node for each scanned symbol
Initialise, 2 stacks namely i) tree stack ii) op stack.

Step 3 : If the scanned symbol is

i) An operand - push it onto the tree stack

ii) An operator - <u>if</u> op stack is empty, push the op node onto op stack,

<u>else</u> - until top of op stack is having (precedence either <u>greater than or equal to</u> the scanned op, pop an op node from op stack and pop 2 nodes from tree stack and assign them as right and left child to the popped node of the op stack and push op node onto ~~tree~~ stack.

push the scanned node onto op stack.

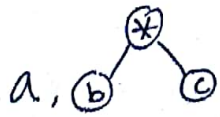
Step 4: Until the op stack becomes empty, pop a node from the op stack and pop 2 nodes from the tree stack., assign as right and left child and push onto tree stack.

Step 5: The root node of the tree stack gives the tree.

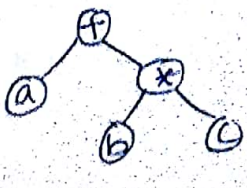
TRACE

Ex :- a+b.*c

symbol	tree stack	op stack
a	a	
+	a	+
b	a b	+
*	a b	+ *
c	a b c	+ *



+



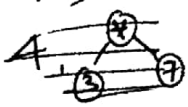
ex 4+3*7-5

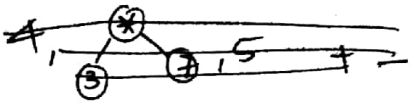
4 +

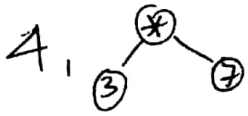
4 3 +

4 3 +*

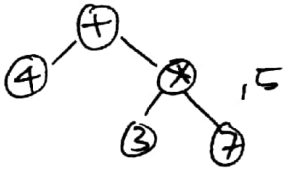
4 3 7 +*



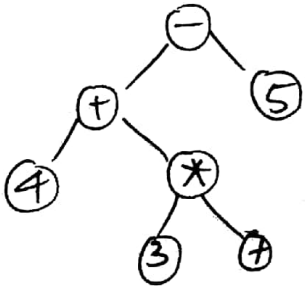




+



–



Program 8

To create an expn tree for given infix expn and traverse the tree in pre, in, post

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
struct node
{
   char info;
   struct node *llink;
   struct node *rlink;
}; typedef struct node *NODE;

// write preorder func - change %d to %c

// write in, post - change.

NODE create node (char item)
{
    NODE temp;
    temp = (NODE) malloc(sizeof(struct node));
    temp -> info = item
    temp -> llink = temp -> rlink = NULL;
    return temp;
}

int preced (char x)
{
    switch (x)
    {
        case ($) : return 3;
        case (*)
        case (/) : return 2;
        case (+) : case(-) : return 1;
```

```
NODE create exp tree (char infix [20])
{
    int i , t1=-1 , t2=-1 ;
    char sym ;
    NODE temp , treestack [15] , opstack [15] , temp1 , r, l;

    for (i=0; infix[i] != NULL; i++)
    {
        sym = infx [i] ;
        temp = create node (sym) ;
        if (isalphanum (sym))
            tree stack [++t1] = temp ;
        else
        {
            if (t2 == -1)
                opstack [++ t2] = temp;
            else
            {
                while ( t2 != -1 &&
                        ( preced ( opstack [t2] → info) ≥
                          preced (sym) ) )
                {
                    temp1 = opstack [t2--] ;
                    r = tree stack [t1--];
                    l = tree stack [t1--];
                    temp1 → rlink = r;
                    temp1 → llink = l;
                    tree stack [++t1] = temp1;
                }
                opstack [++t2] = temp;
            }
        }
    }
}
```

```
while (t2 != -1)
{
    temp1 = opstack [t2--];
    temp1 -> rlink = treestack [t1--];
    temp1 -> llink = treestack [t1--];
    treestack [++t1] = temp1;
}
return treestack [t1];


nt main ()
{
NODE root = NULL;
char infix [15];
printf ("\n Read infix exp\n");
scanf ("%s", infix);
root = createexptree (infix);
printf ("\n The preorder traversele is \n");
preorder (root);
pf (" In");
inorddder (root);
pf (" post");
postorder (root);
}
```

AVL Tree (Adelson Velskoi/ Lendis)

The worst case efficiency of any operation on a BST is O(n) where n is the no of nodes in the binary tree

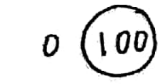
An AVL tree is a BST wherein, any operation in the worst case will take time of O (log n)

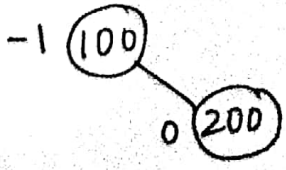
* An AVL tree is a BST wherein, the balance factor of each node in a tree should either be 0, +1 or −1 .

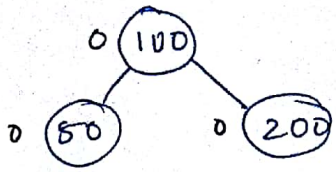
The balance factor of a node is calculated as

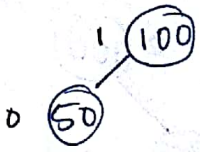
BF = ht of left subtree − ht of right subtree

Eg for AVLT :-







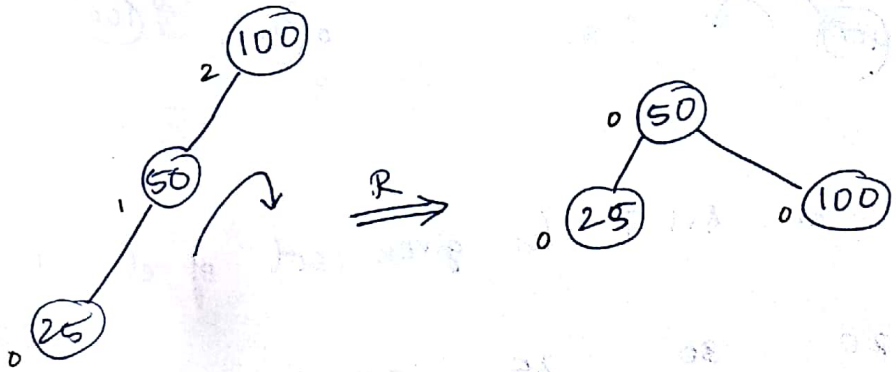


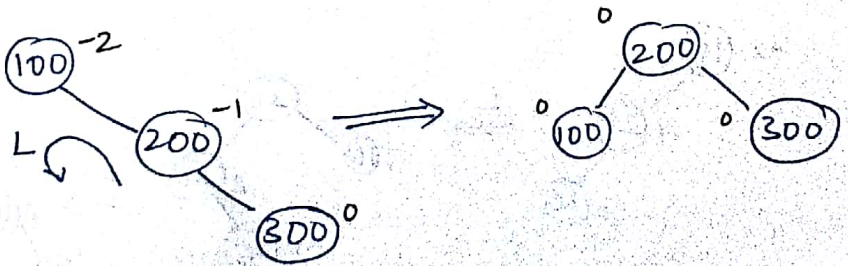
While constructing an AVL tree, to balance the tree, we come across 2 types of rotation opⁿ namely,

i) Single rotation

ii) Double rotation.

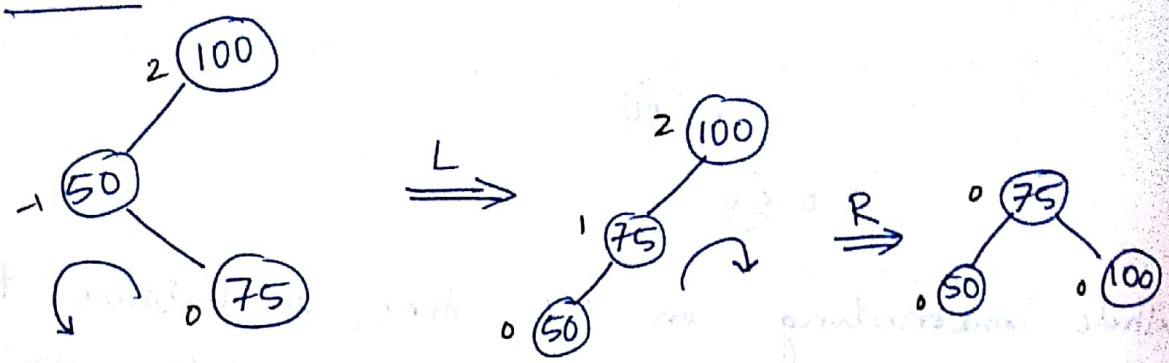
Single Rotation



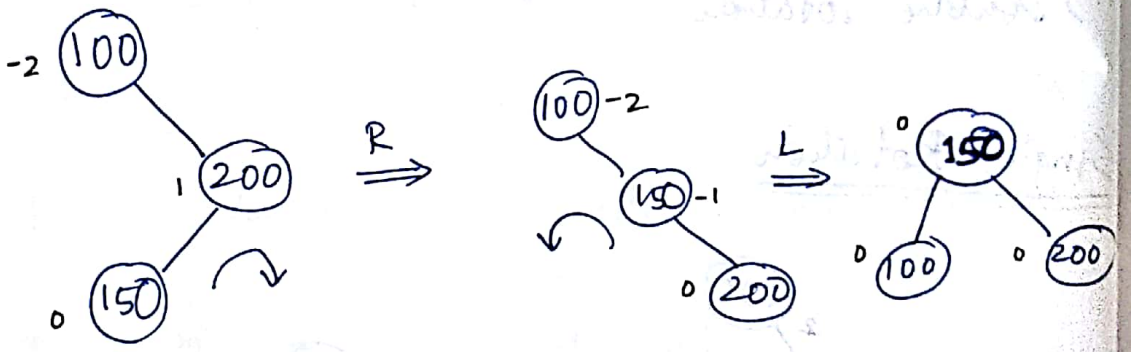


Double Rotation

i) LR rotn

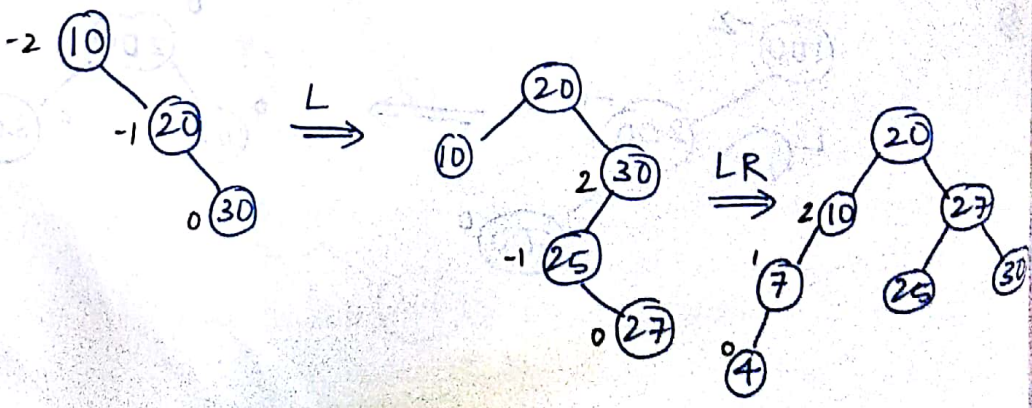


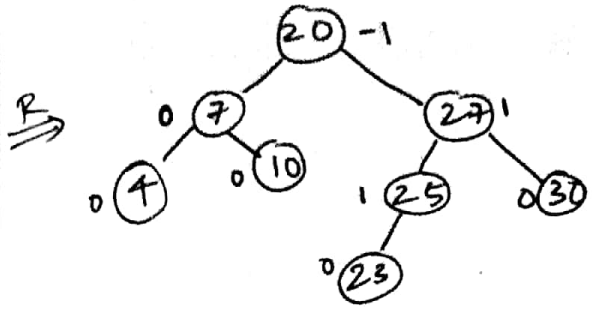
ii) RL rotn



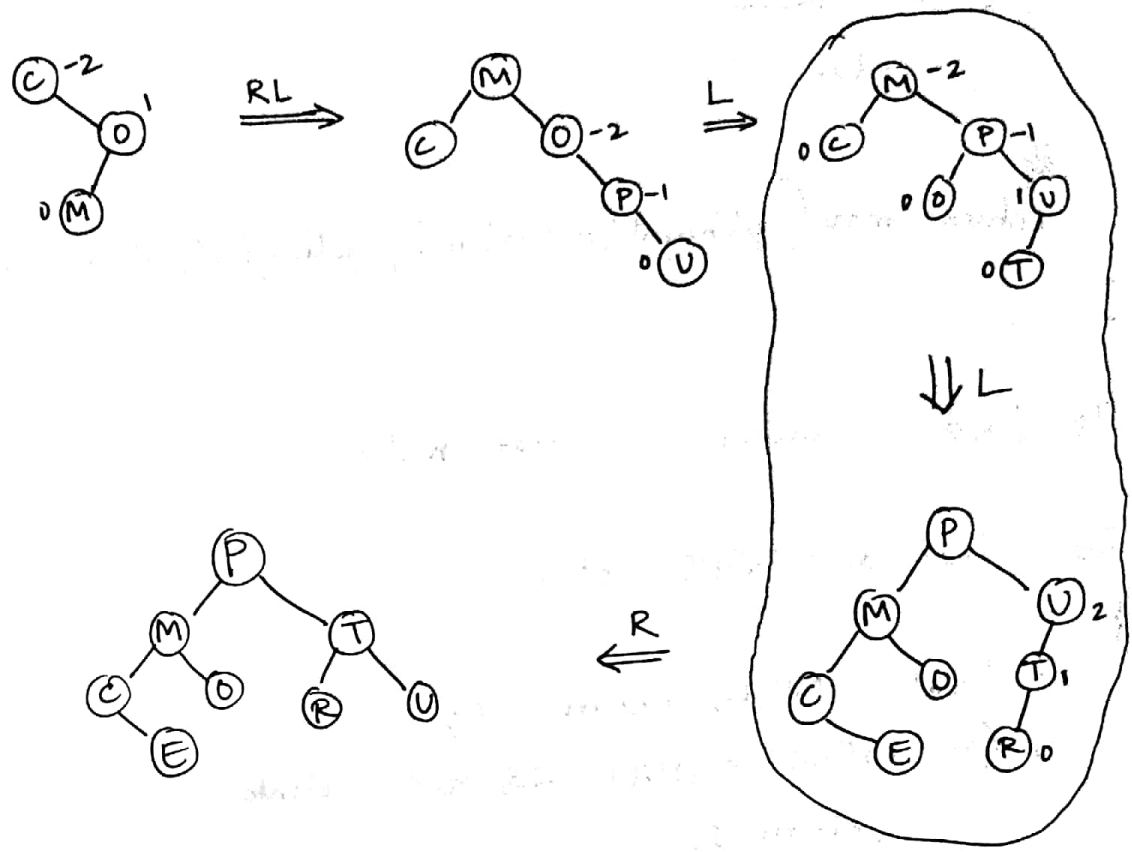
⑧ Construct an AVL T for given set of elements.

10 20 30 25 27 7 4 23





(B) The characters of the word "COMPUTER".



(S) "COMPUTING"

Write a C func to ~~write~~ find height of a tree.

```
int getheight(NODE root)
{
    if (root == NULL)
        return 0;
    return max(getheight(root->llink), getheight(root->rlink)) + 1;
}
```

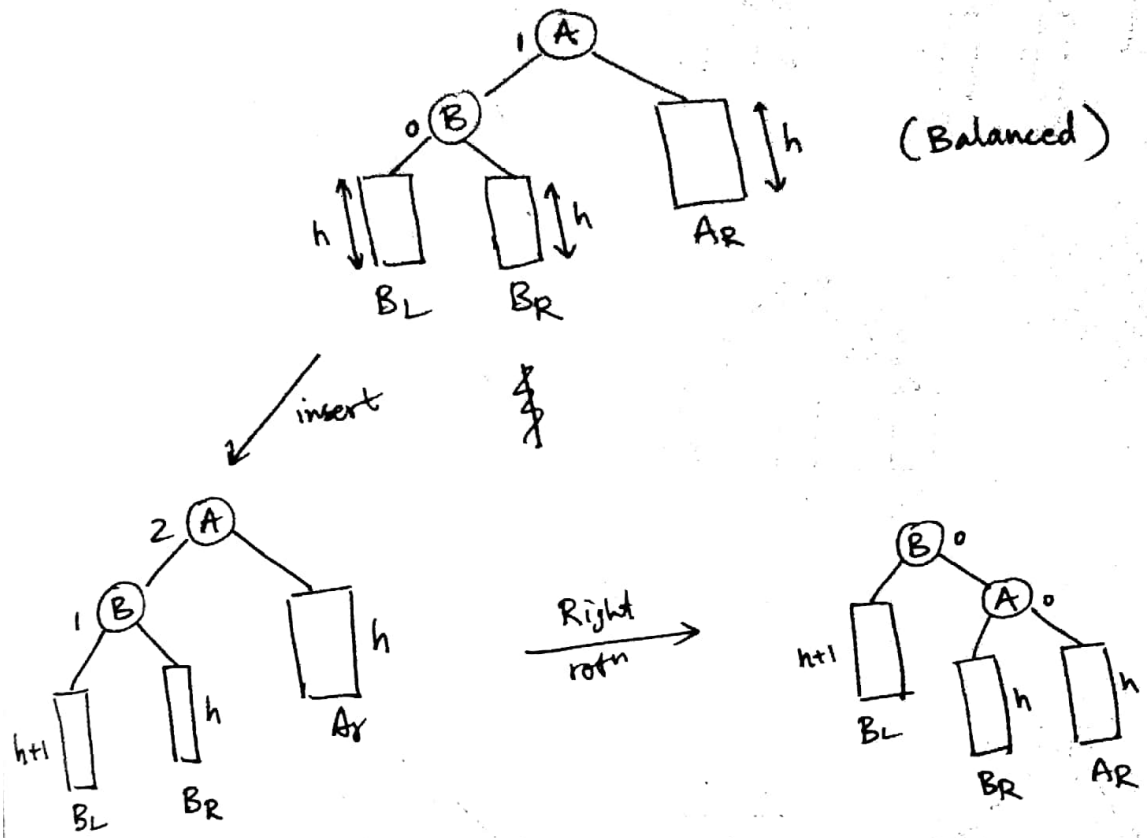
To find number of leaf nodes

```
int leafnode(NODE root)
{
    if (root == NULL) return 0;
    if (root->llink == NULL && root->rlink == NULL)
        return 1;
    return leafnode(root->llink) + leafnode(root->rlink);
```

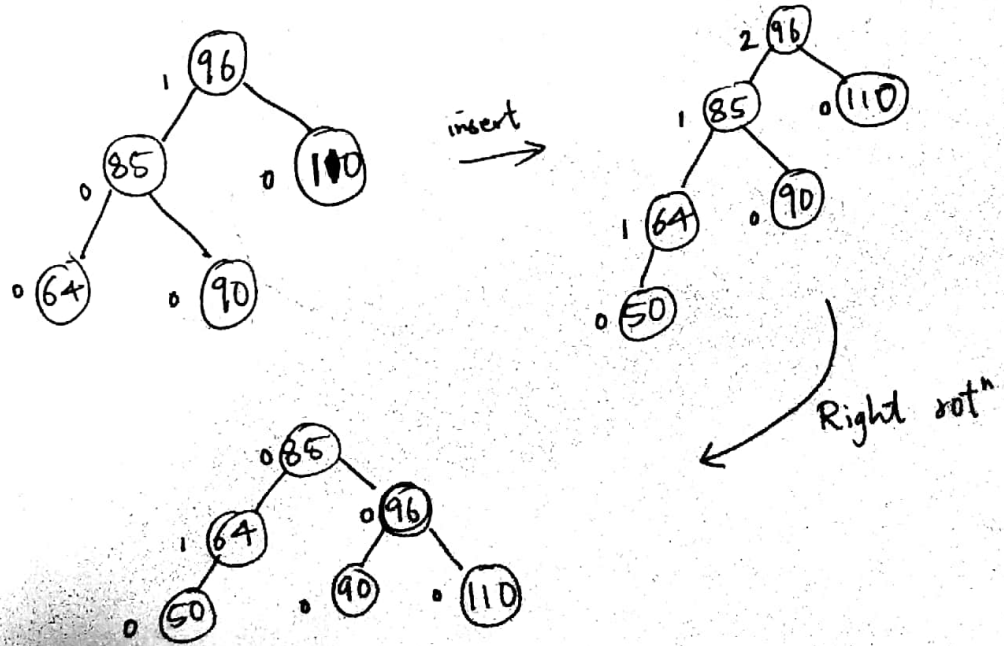
Single Rotation

Right Rotation (LL)

If a new node is inserted in the left subtree of the left subtree of the reference node

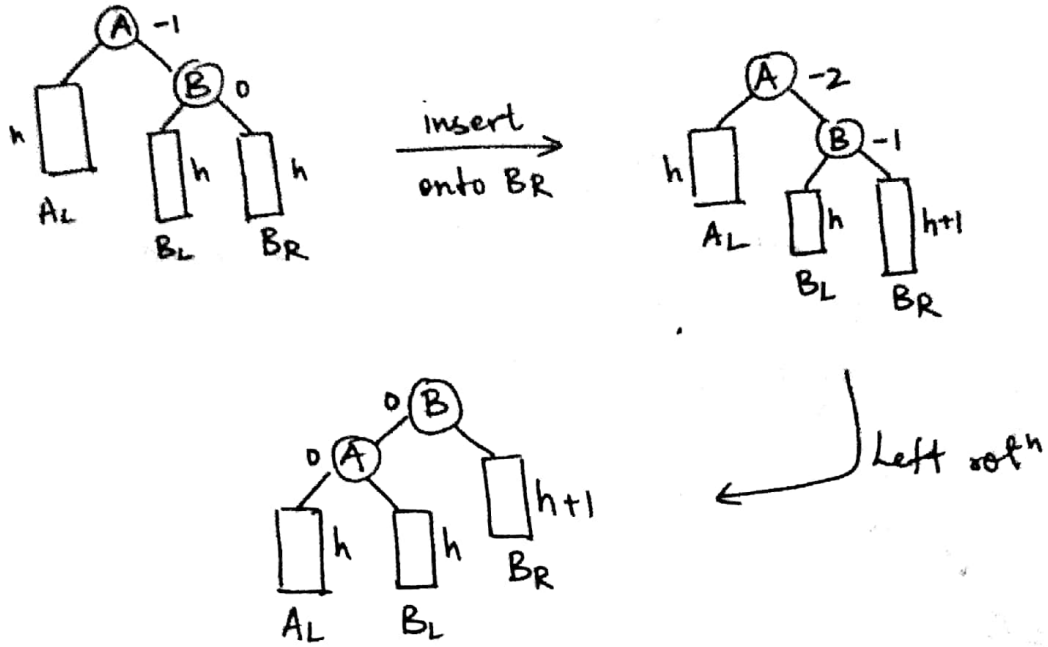


ex:-

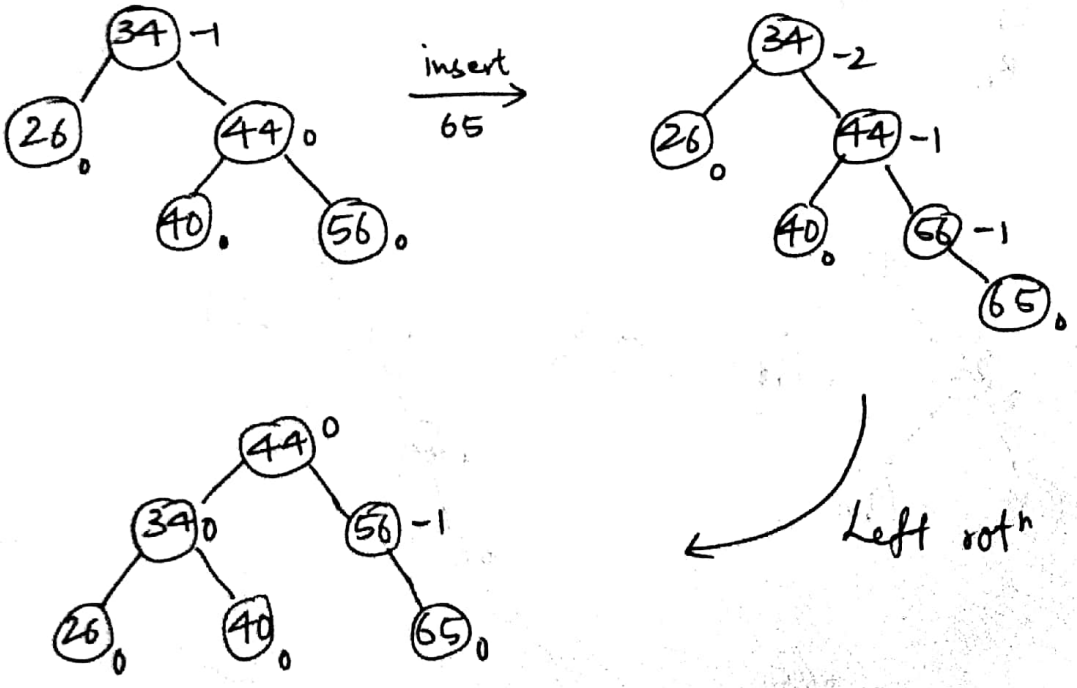


Left Rotation (RR)

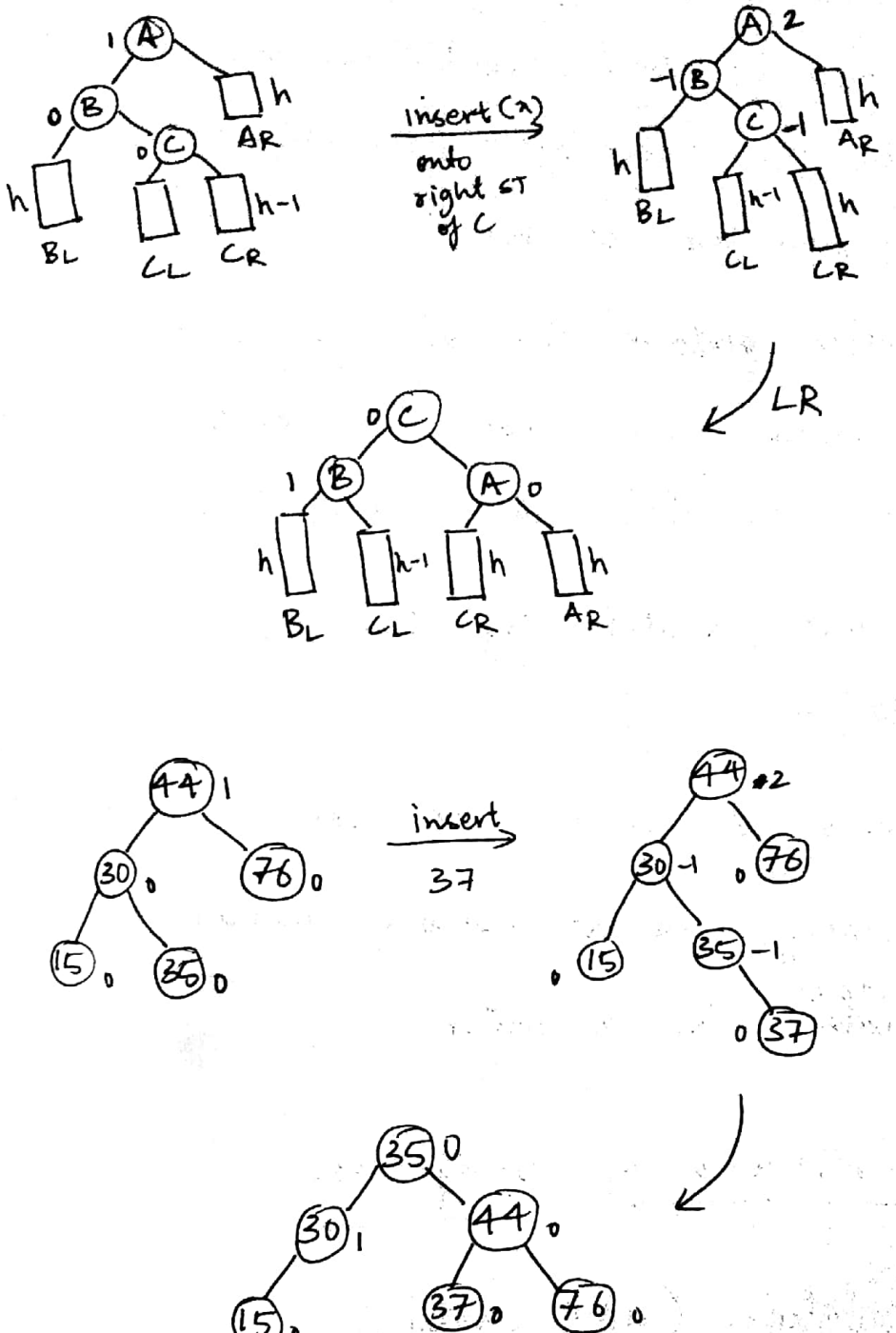
A new node is inserted in the right subtree of the right subtree of the reference node.



eg:-



LR rotation



Ex:-

Splay Tree

A splay tree is a self adjusting BST with the additional property that recently accessed elements are quick to access again (recently accessed elements should be promoted as root)

The splay tree performs the basic operation like insertion deletion and search in O(log n) time as compared to O(n) in a BST

Splay tree was invented by Daniel.D.Sleator, and Robert Tarjan

Applications of Splay tree can be found in Cache memory, virtual memory management, the routing tables in a router.

Types of rotations in Splay Tree

1. Zig rotation (right rotation)
2. Zag rotn (left rotn)
3. Zig Zig
4. Zag Zag
5. Zig Zag
6. Zag Zig.

The decision on what rotn to be performed depends on the following.

① Does the node we are trying to rotate have a grand parent?

② Is the node a left or a right child of the parent.

③ Is the parent the left or the right child of the GP.

④ If the node does not have a GP, we carry out the Left rotn if it is the right child of parent, otherwise we carry out right rotation.

⑤ If the node has a GP, then, the following 4 possibilities can happen

i) If the node is the left of parent and parent is left of GP. We need to do zig zig

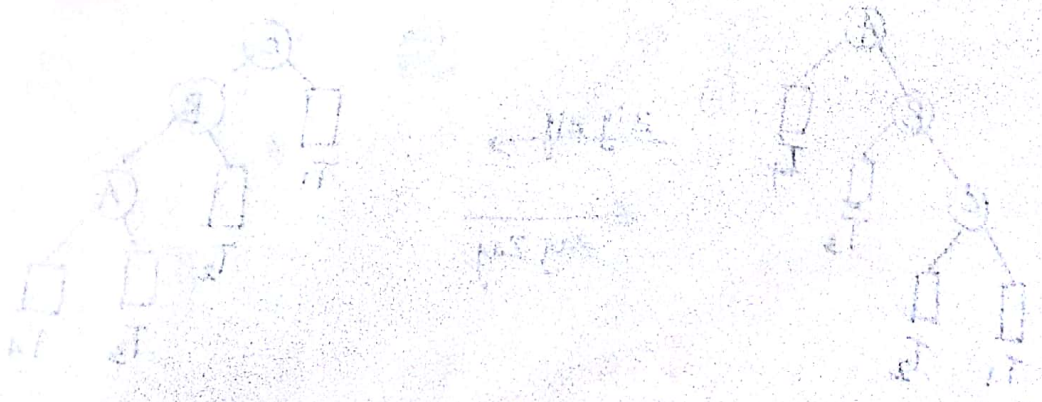
ii) If node is right of P and P is right of GP.

zag zag.

iii) If node is right of P but parent is left of GP. Zag Zig

iv) If node is left of P but P is right of GP

ZigZag



Procedure to perform insertion operation in splay tree

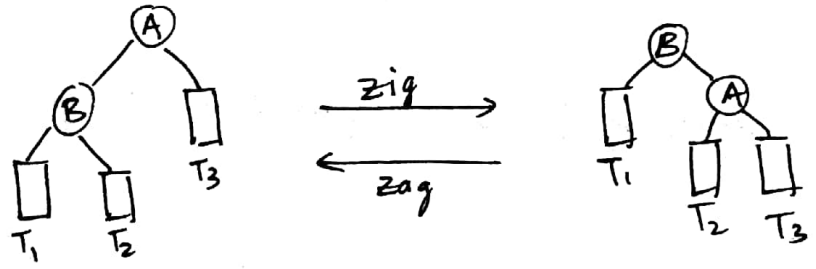
Step 1: Check whether tree is empty

2: If tree is empty, then insert the new node as root and exit.

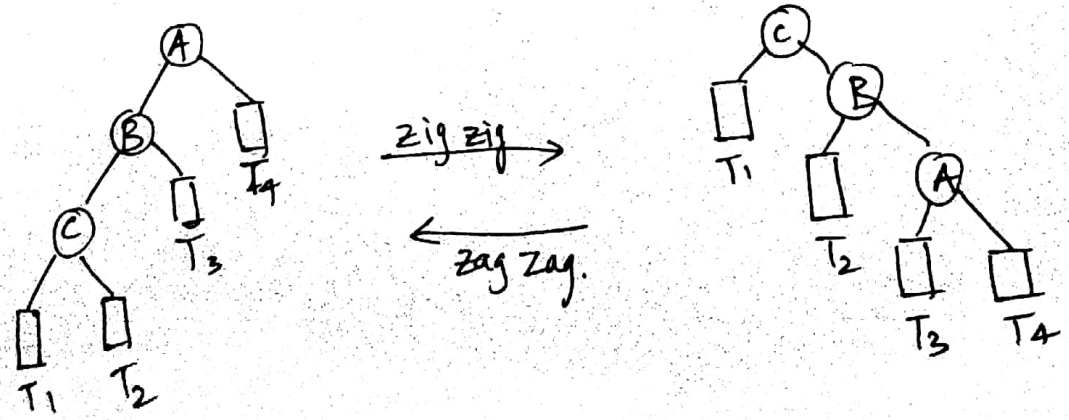
3: If tree is not empty then insert the new node as a leaf node based on the logic of BST

4: After insertion, splay the new node.

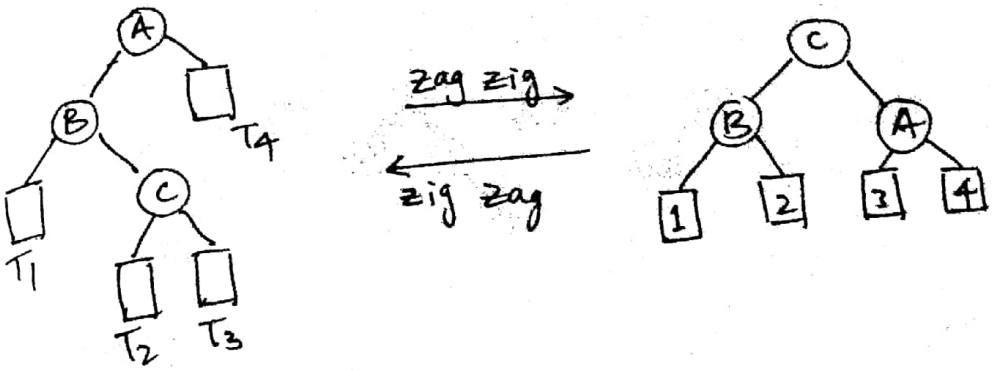
i) zig and zag.



ii) zig zig.

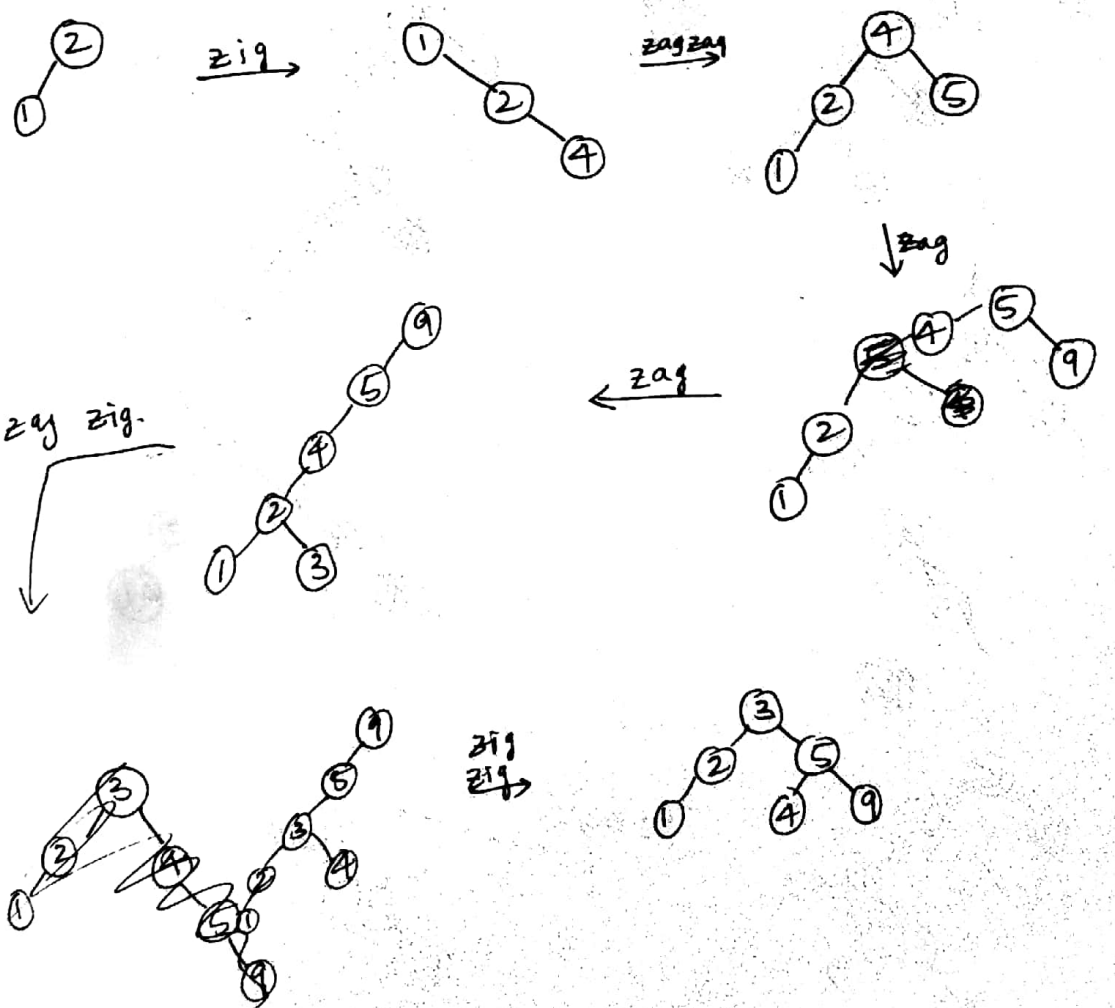


iii) zig zag.

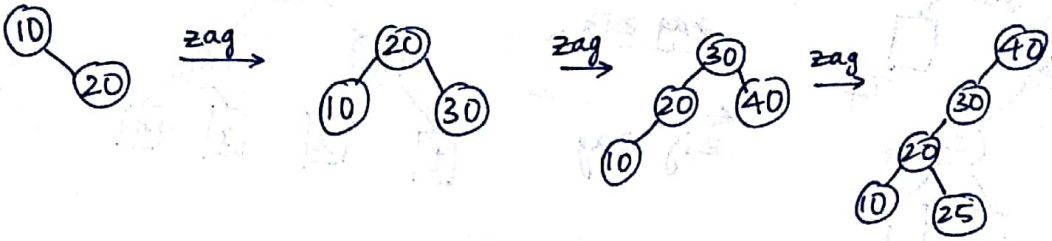


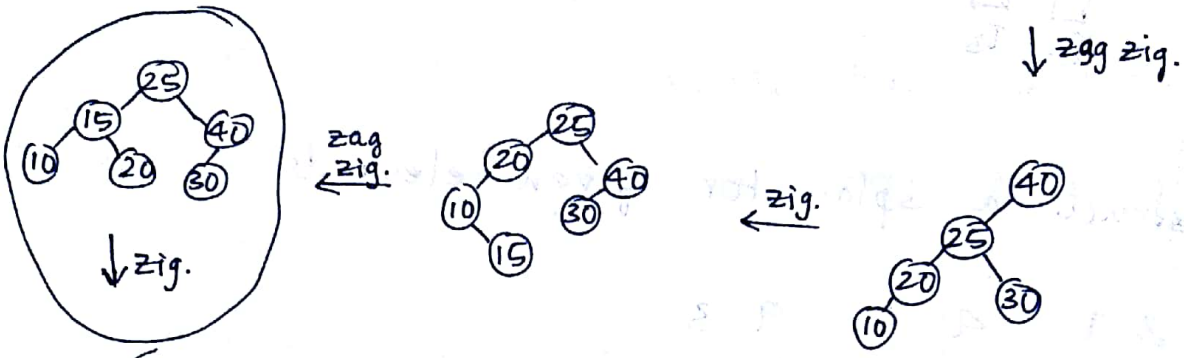
b) Construct a splay for given elements.

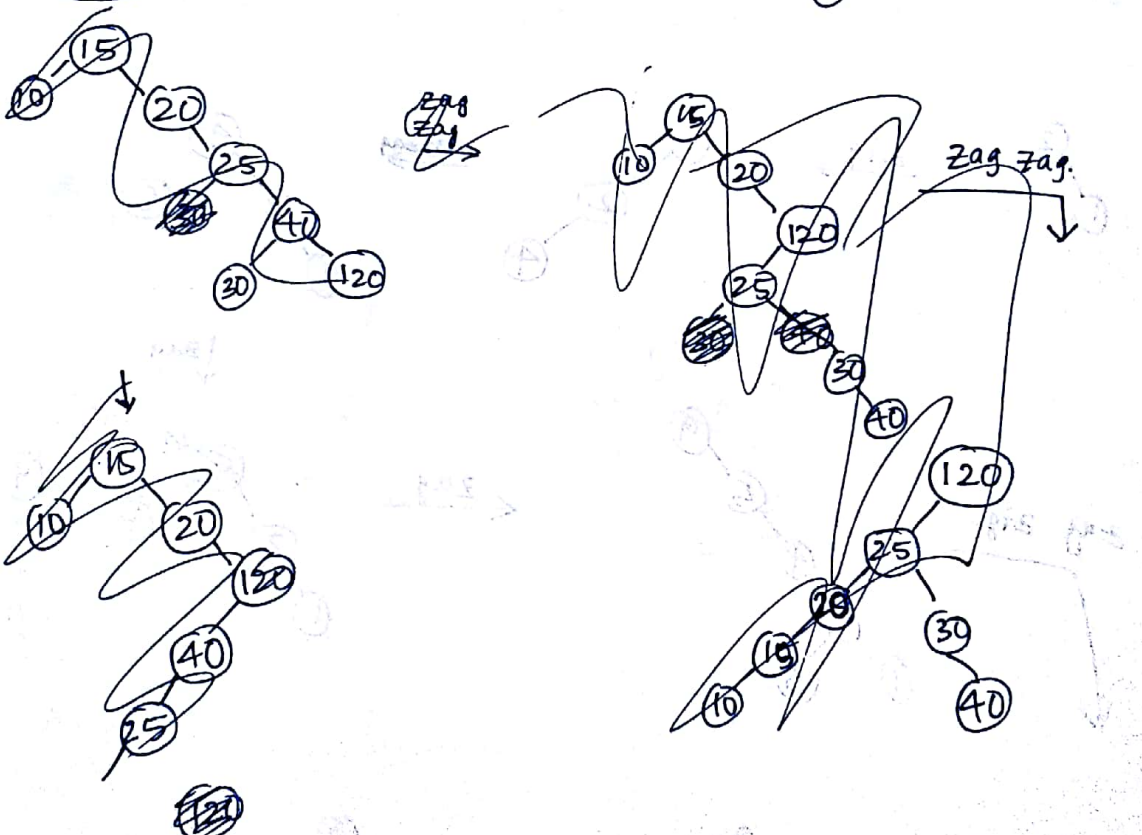
2 1 4 5 9 3

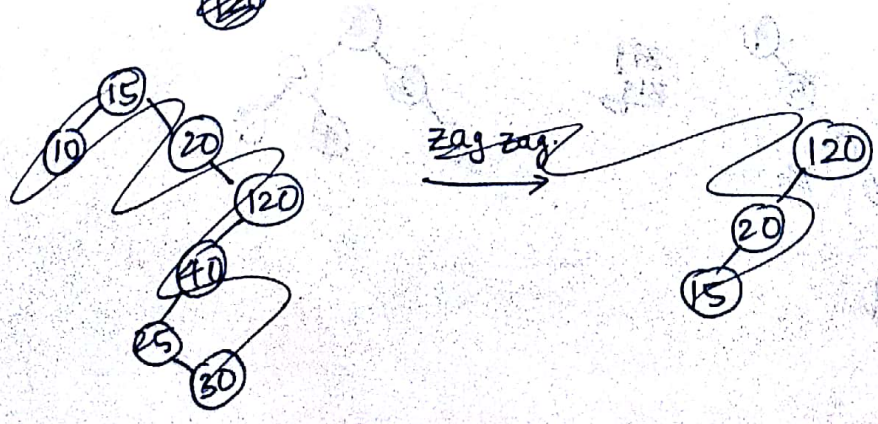


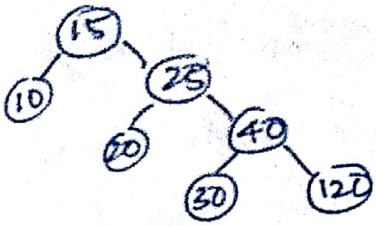
(8) 10 20 30 40 25 15 120



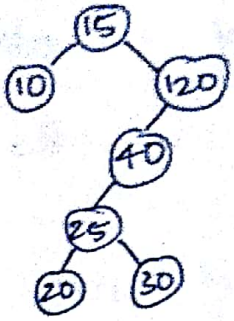




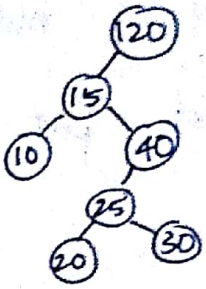


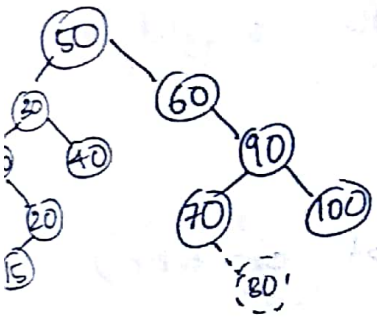


zag zag.

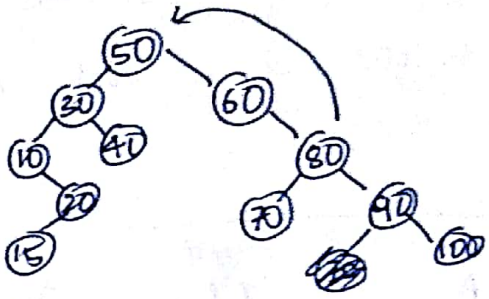


↓ zag.

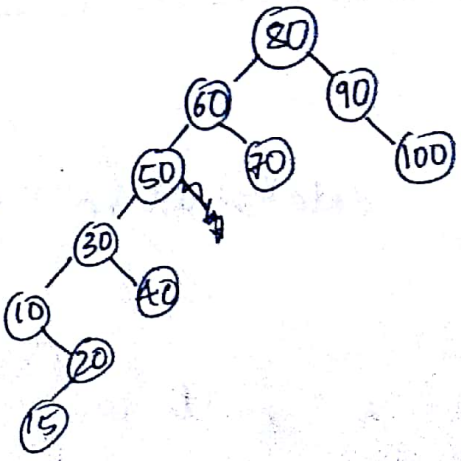




80
zag
zig



↓ zig zag.



The depth of a node is the no of edges from the node to the tree's root node.

The depth of root node is zero.

The height of a node is the number of edges on the longest path from the node to a leaf.

The height of leaf node is zero.
The height of a root node gives the height of the tree.

DMA , LLs , Trees (except exp tree)
for test 2

Trie

Its a data structure to maintain a set of strings.

Trie is a multi way search ~~search~~ tree.

Trie is derived from the word Information Retrieval.

edges from

.

of edges

le to a

height of the

exp tree)

a set of

rch tree.

Information

There are two types of tries

i) Standard trie
ii) Compressed trie.

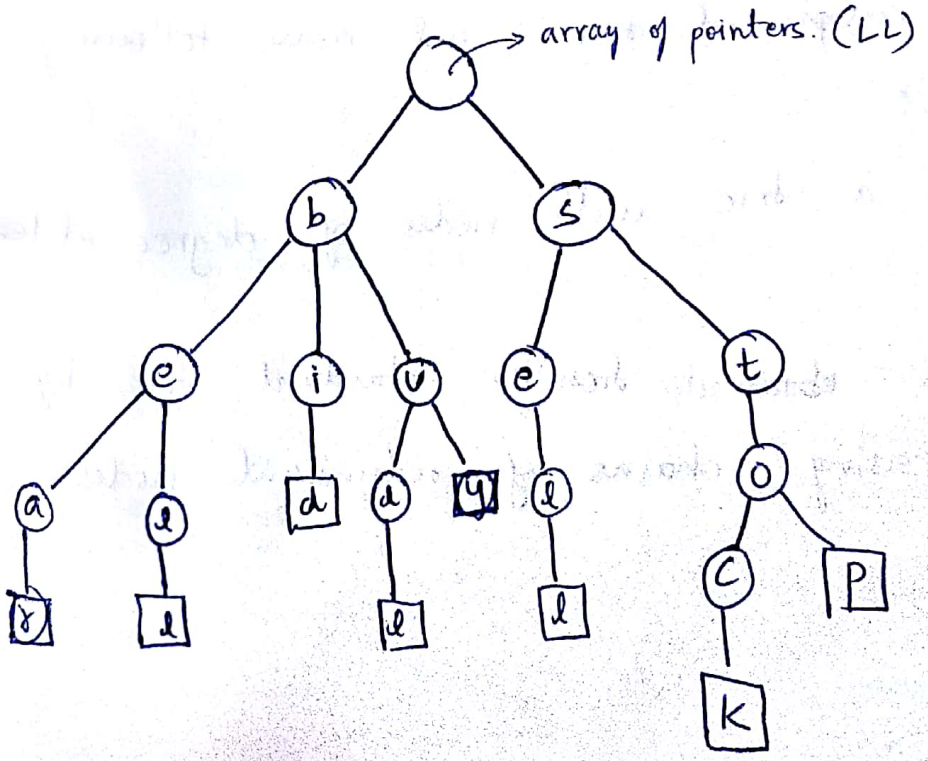
STANDARD

1) A std trie for a set of strings 's' is an ordered tree such that each node apart from root is labelled with a character.

2) The children of a node are alphabetically ordered.

3) The path from the external nodes to the root yeild the string of 's'

ex:- s = { bear, bell, bid, bull, buy, sell, stock, stop }



each node

↓

i) character

ii) array of pointers

iii) pointer pointing to parent.

Time taken to search a string in a trie is ~~order of~~ $O(ns)$

n = size of string

s = no of pointers

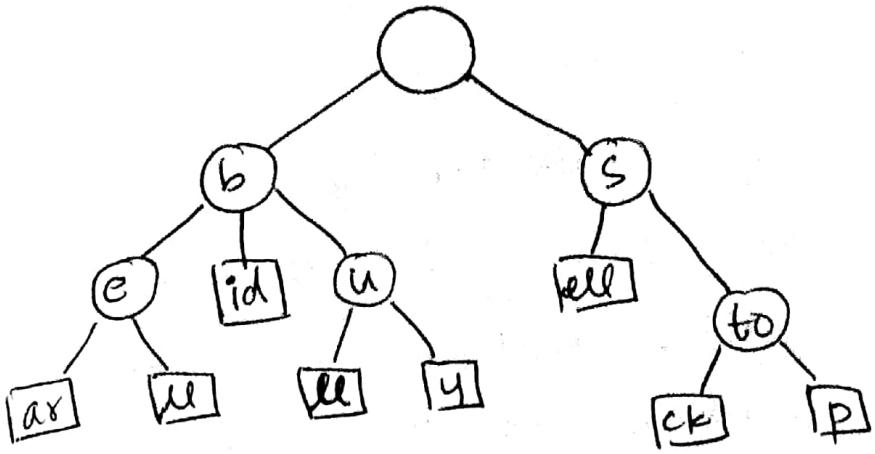
COMPRESSED TRIE

The compressed trie reduces the size of the standard trie.

In a compressed trie, we have following properties :-

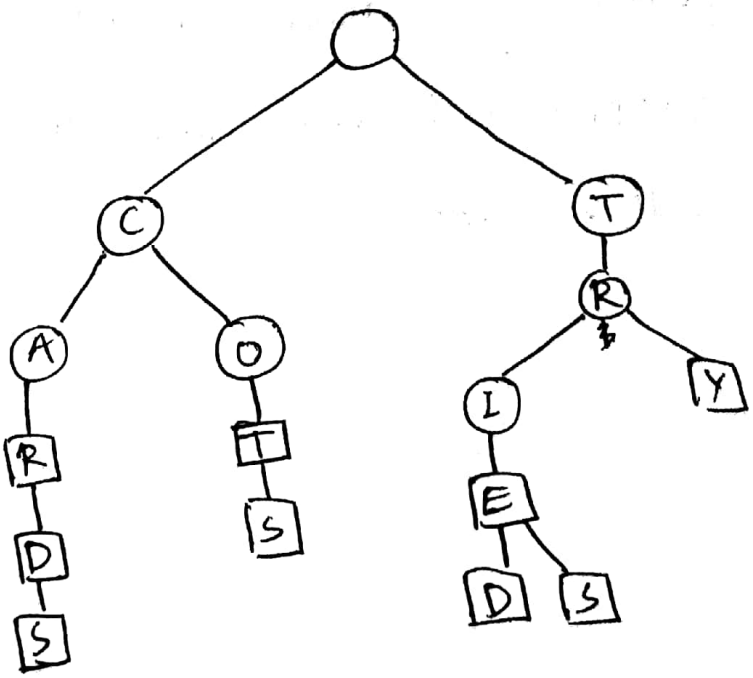
i) it is a trie with nodes of degree at least 2

ii) It is obtained from the standard trie by compressing chains of redundant nodes.

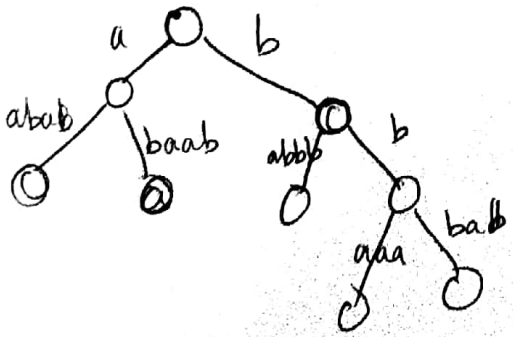


3) Contruct a trie for the given set of words

CAR
CARD
CARDS
COT
COTS
TRIE
TRIED
TRIES
TRY



Insertion operation on a trie



i) To implement a dictionary

ii) Used in the implementation of search engine.

iii) Used in pattern matching - generally in pattern matching algorithm, we do pre processing for the pattern. Using tries, we can preprocess the text itself while performing pattern matching and the preprocessed tree using trie is termed as suffix tree.