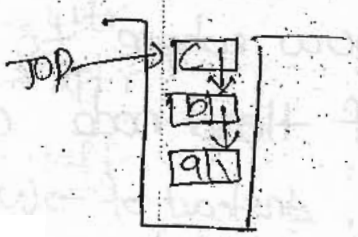
o/p: same as push(int x)



ie it inserts elements ahead (first) of list

Observation:

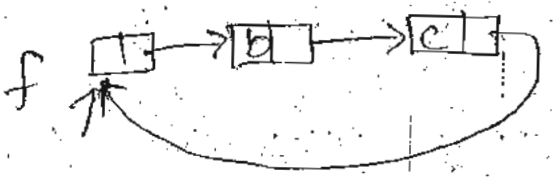
Linked stack push = insert front of SLL

Disadvantage of SLL:

deleted

1. The link part of last node is not utilised
2. stepping backwards is not possible (delete())

Hence SLL → circular LL



Advantage of circular LL:

1. the link part of last node is utilised by placing the address of first node
2. stepping backwards is possible by moving forward only in one round

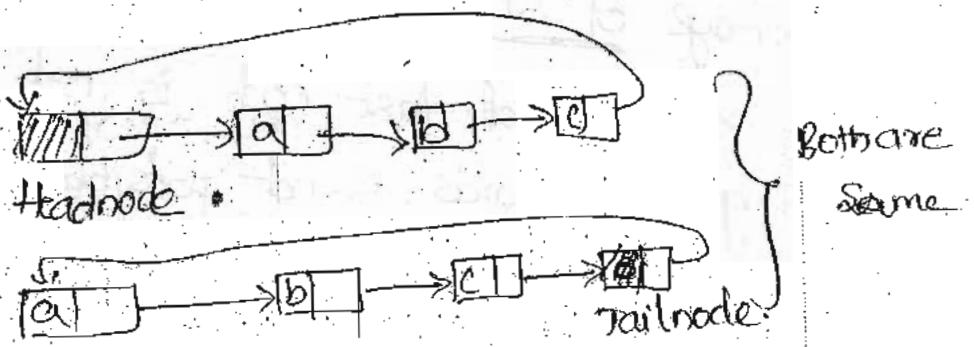
Disadvantage of circular LL:

- Danger of falling into infinite loop

Significance of Head node (or) tail node:

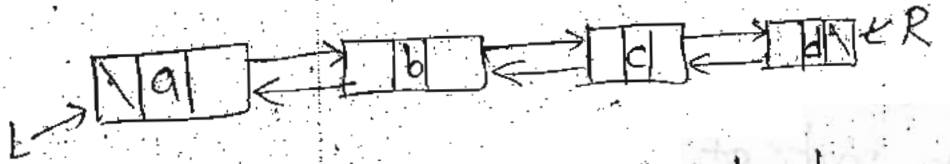
- No danger of falling into infinite loop

- The list can never be empty (atleast Head node is present to know where to begin)
- the information part of Head node can be utilised (as counter; instead of O(N), O(1))
- Certain operations are easy
  (i) concatenation()
  (ii) split()



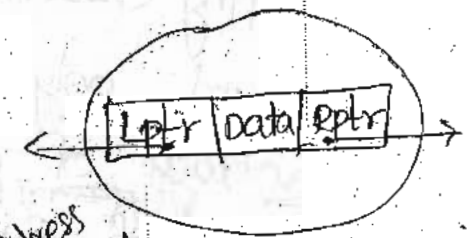
8/8/11

Doubly Linked list:



- It is a self referential structure

```
struct DNode
{
    struct DNode *LPtr;
    int Data;
    struct DNode *RPtr;
};
struct DNode *L = NULL;   // holds the address of leftmost node
struct DNode *R = NULL;   // holds the address of rightmost node
                          // Initialization
```



Access:

t → Data
t → Lptr
t → Rptr

Move left: $t = t \rightarrow Lptr$

Move right: $t = t \rightarrow Rptr$

(P) How many link fields are getting updated for performing the following operations

	Leftmost	Rightmost	else (middle)
Insert	3	3	4 ✓
Delete	1	1	2

→ L → NULL, R → NULL ← empty list

If (L == NULL)

(OR)

If (R == NULL)

↔ Single Node L → [ ] ← R

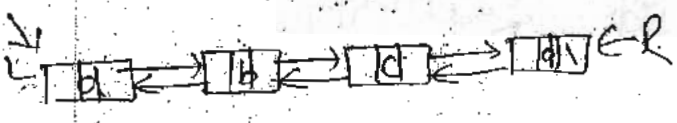
If (L == R)

A: Insert a node to the right of M, where M is the address of any middle node and the condition is that all the operations should be performed w.r.t. newly inserted node.

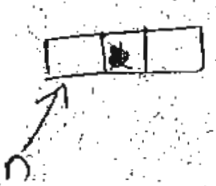
Inserting leftmost

```
void insertleftmost (int x)
{
1) struct DNode *n = GetDNode();
2)      n->data = x;
3)      n->Lptr = NULL; ①
4)      n->Rptr = L; ②
5)      L->Lptr = n; ③
6)      L = n;
}
```

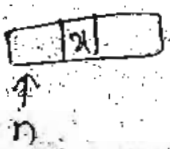
eg :



1)

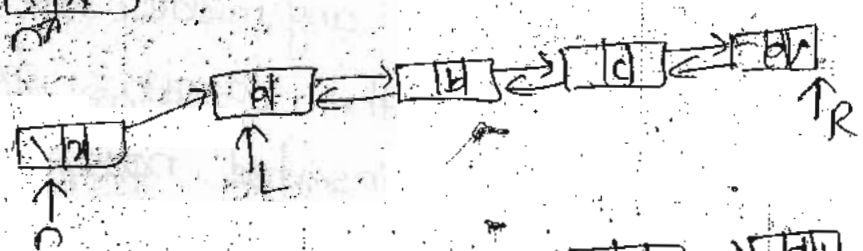


2)

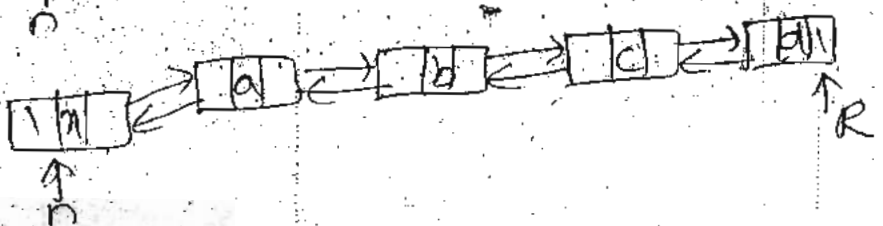


3)

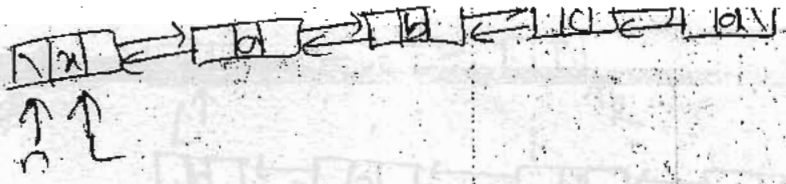
4)



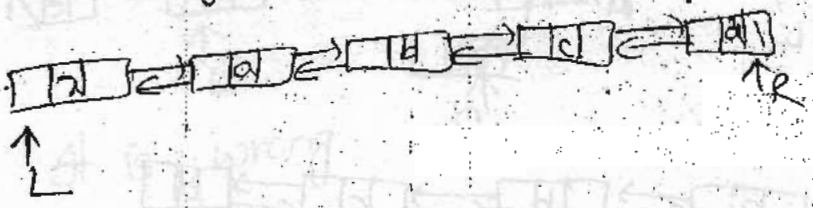
5)



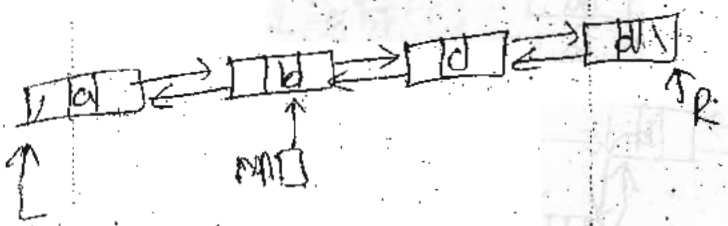
6)



After coming out



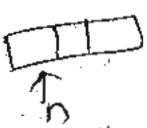
Inserting in middle:



```
void insert right of M (int x)
{
1) struct node *n = GetNode();
2) n -> Data = x;
3) n -> Lptr = M;                    (1)
4) n -> Rptr = M -> Rptr;            (2)
5) M -> Rptr
   n -> Lptr -> Rptr } = n;          (3)
6) n -> Rptr -> Lptr = n;            (4)
```

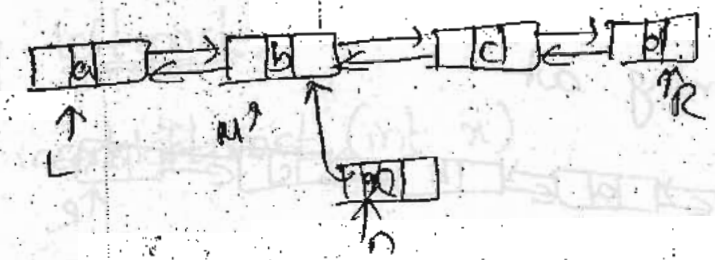
Eg:

1)

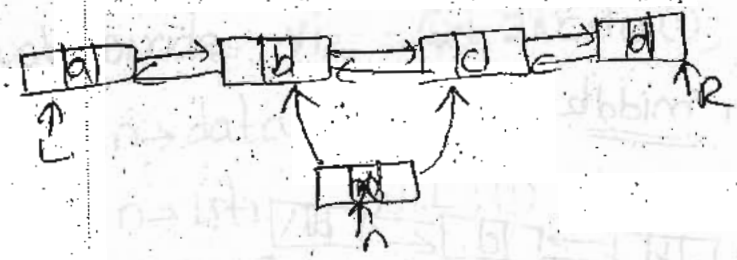


2)

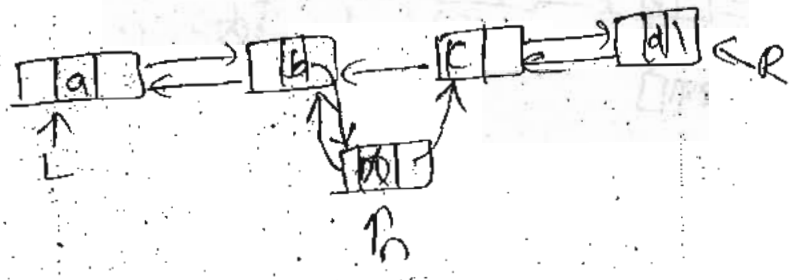
3)



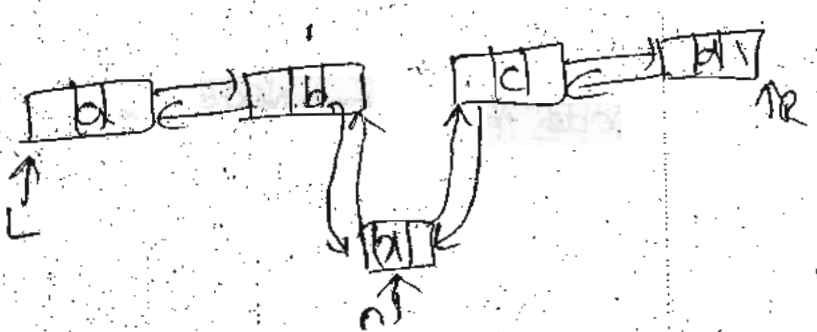
4)



5)



6)



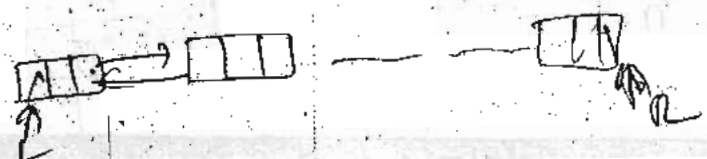
After coming out

p) Identify whether the following steps deletes the left most node

i) L = L → Rptr;

ii) L → Lptr = NULL;

Eg:



i)

ii)

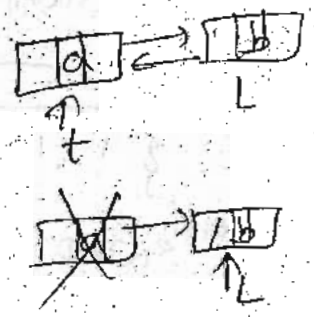
It is wrong.

So the correct steps are

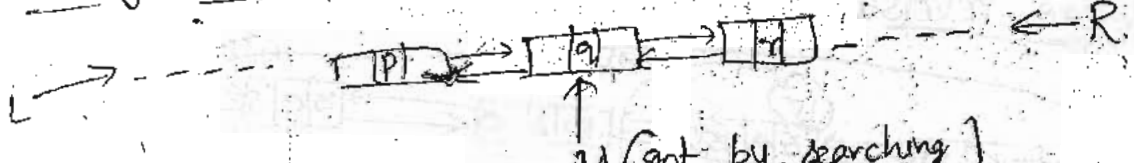
1) L = L → Rptr;
2) free (L → Lptr)
3) L → Lptr = NULL;

II method:

```
struct DNode *t=L;
L=L->Rptr;
L->Lptr = NULL;
free(t);
```



Deleting middle element:




```
void DeleteM()
{
  i) M->Lptr->Rptr = M->Rptr;
  ii) M->Rptr->Lptr = M->Lptr;
      free(M);
}
```

Circularly DLL

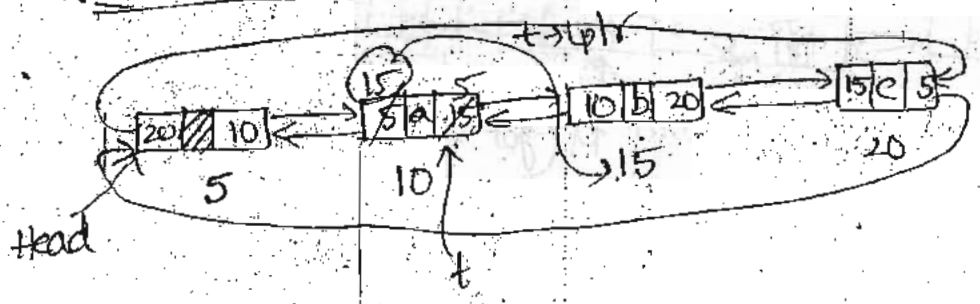
fill up the blank

a) t=t→Rptr b) t=t→Lptr c) t=NULL d) error

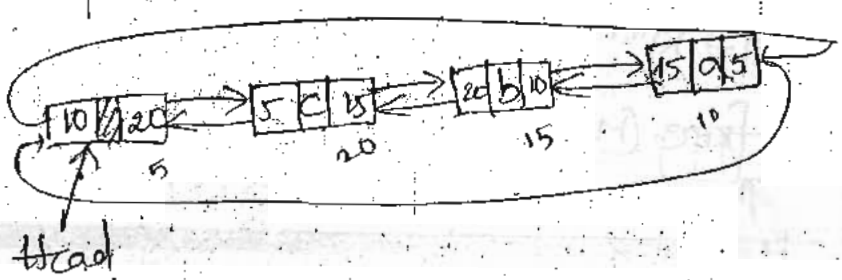
The following code reverses a given circularly DLL

```
void Reverse_CDLL()
{
    struct DNode *t = Head->Rptr;
    while (t!=Head)
    {
        swap (t->Lptr, t->Rptr);
        [          9          ];
    }
    swap (Head->Lptr, Head->Rptr);
}
```

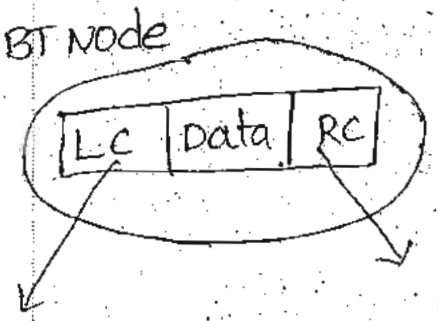
Before reverse:



After reverse



Trees:



Rajesh T.K.

LC = leftchild
RC = Rightchild

```
struct BTNode
{
    struct BTNode *LC;
    int Data;
    struct BTNode *RC;
};
```

Hence self referential structure

```
struct BTNode *t = NULL; //Initialisation
```

Root

Access:

t → Data

t → RC

t → LC

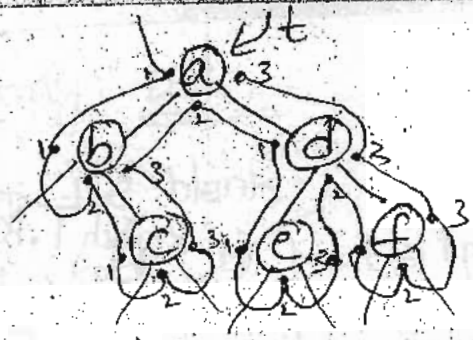
Descend Left:

t = t → LC

Descend Right:

t = t → RC

→



Traversals:

Inorder: Left, Root, Right
(1) (2) (3)

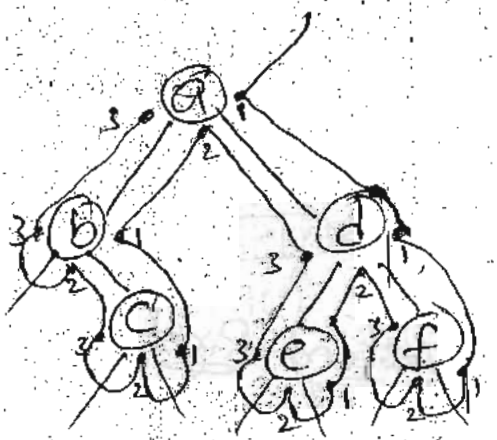
b c a e d f

Postorder: Left, Right, Root
(3)

c b e f d a

preorder: Root, Left, Right
(1)

a b c d e f



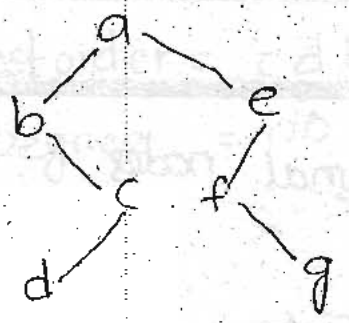
Converse Inorder: Right, Root, Left

f d e a c b

Converse postordor: f e d c b a Right, Left, Root

Converse preorder: Root, Right, Left → a d f e c b

→



Inorder: b d c a f g e converse Inorder: e g f a c d b

postorder: d c b g f e a converse postorder: g f e d c b a

preorder: a b c d e f g converse preorder: a e f g b c d

p) for a given unlabelled tree with 3 nodes, how many distinct BTs are therere?

∴ 5 BT

for 'n' nodes ⇒ $\frac{2nC_n}{n+1}$

$n=3 \Rightarrow \frac{6C_3}{4} = \frac{6\times5\times4}{4\times6} = 5$

$n=5 \Rightarrow \frac{10C_5}{6} = \frac{10\times9\times8\times7\times6}{120\times6} = 42$

p) for a given preorder: abc, how many BTs are there.

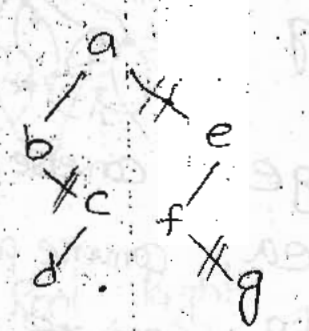
$\frac{2nC_n}{n+1}$

p) what is the post order for the given preorder:

Rptr
preorder = a b c d e f
TAG = - - 1 - 1 1

TAG indicates terminal nodes

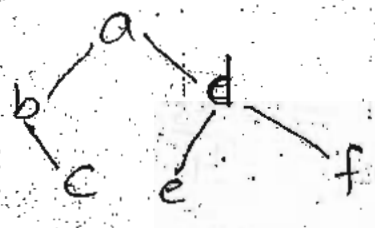
Eg.



Rptr
pre = a b c d e f g
TAG = - - - 1 - - 1

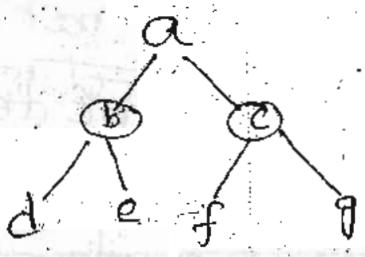
Given problem

Rptr
preorder = a b c d e f
TAG = - - 1 - 1 1



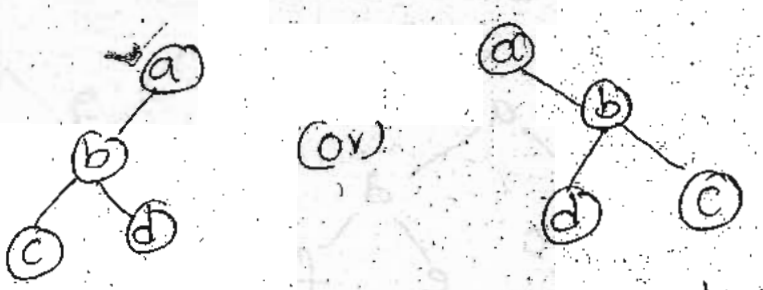
(P) Given post order = debfgca
degree = 0020022
↑
outdegree

debfgca



(P) postorder = c d b a

Degree = 0 0 2 1

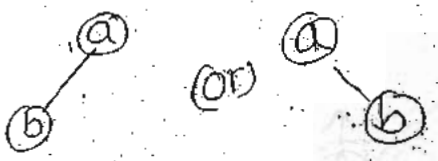


ob : when degree is '1' we cannot distinguish Left (or) Right child

12/8/11

P) preorder = @b

postorder = b@ Generate BT



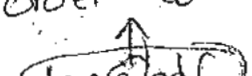
ob: To have unique pattern, we need Inorder + preorder (or) Inorder + postorder

(P) pre = abcdef

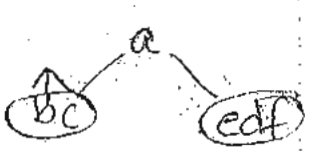
In = bcaedf

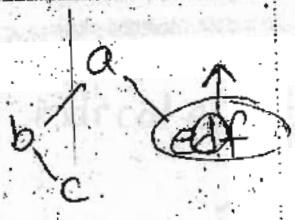
concept:

- consider Inorder
- Look @ preorder to find next node

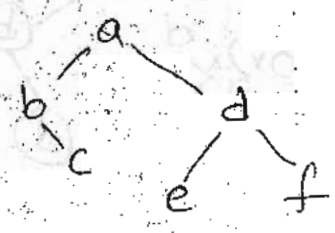


preorder: abcdef



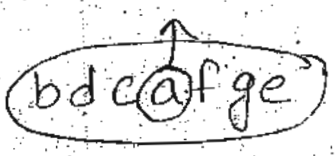


preorder: abcdef

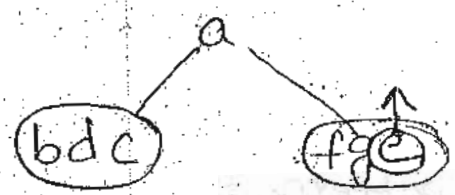


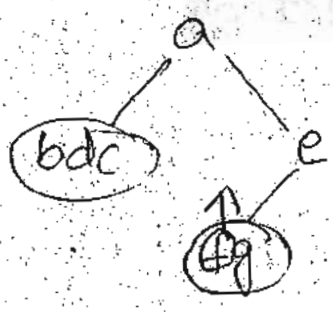
(P) postorder = dcbgfea

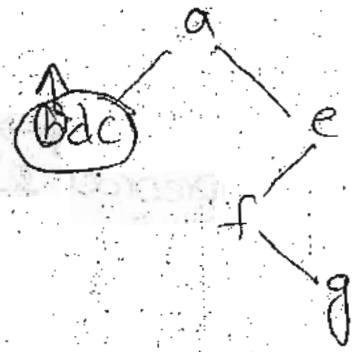
Inorder = bdcafge

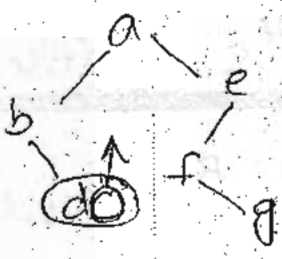


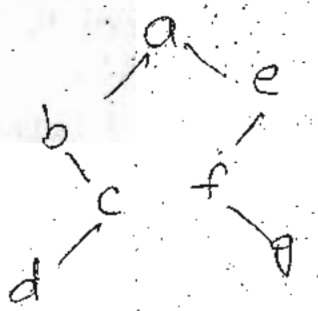
post: dcbgfea



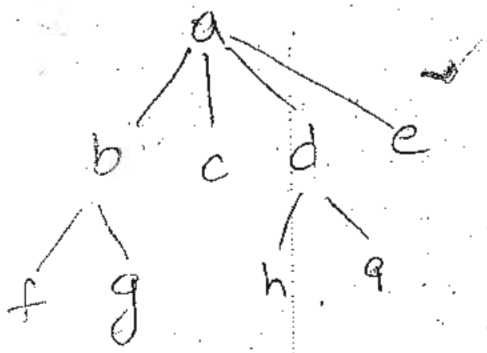








General tree representation: parenthesis representation



a(b(f,g), c, d(h,q), e)

a(bc(e(fgh))d)