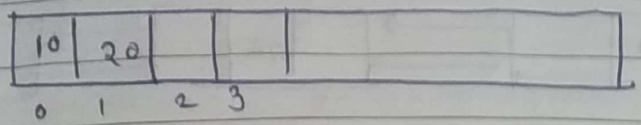
* LINKED LIST *

PAGE No.
DATE / /

In array time complexity is less as

A[10]



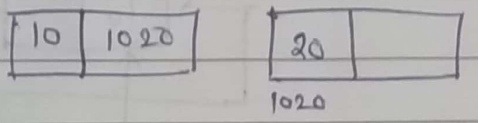
(1000 + (3 X 4)) → To find a element

↓ Base address ↓ index └→ no. of bytes

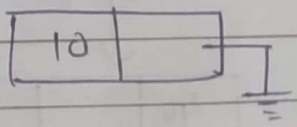
T.C. = O(1)

* Declaration for Linked List *

```
struct Node
{
int data;
struct Node *next;
};
```

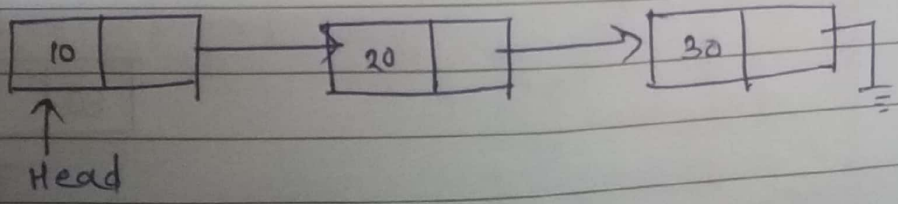



```
struct Node *newnode
newnode = (Node *) malloc (size of (struct Node)

newnode → data = 10;
newnode → next = Null;
```

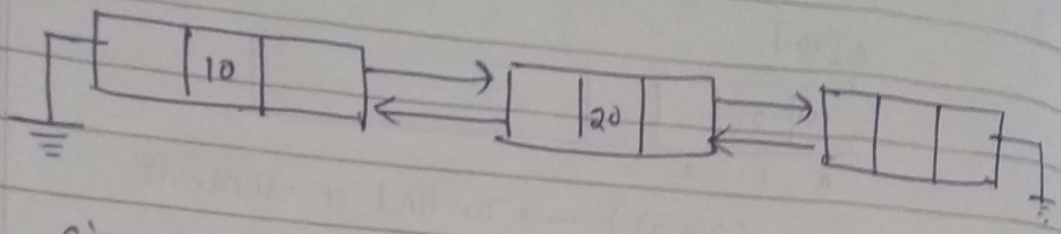


* Types of Linked List :-

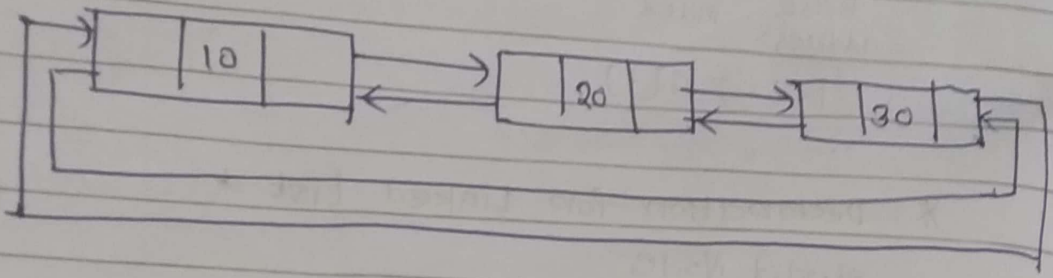
(1) Singly Linked List :-

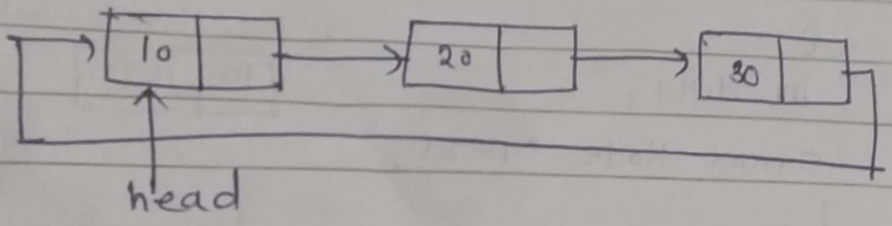


② Doubly Linked list

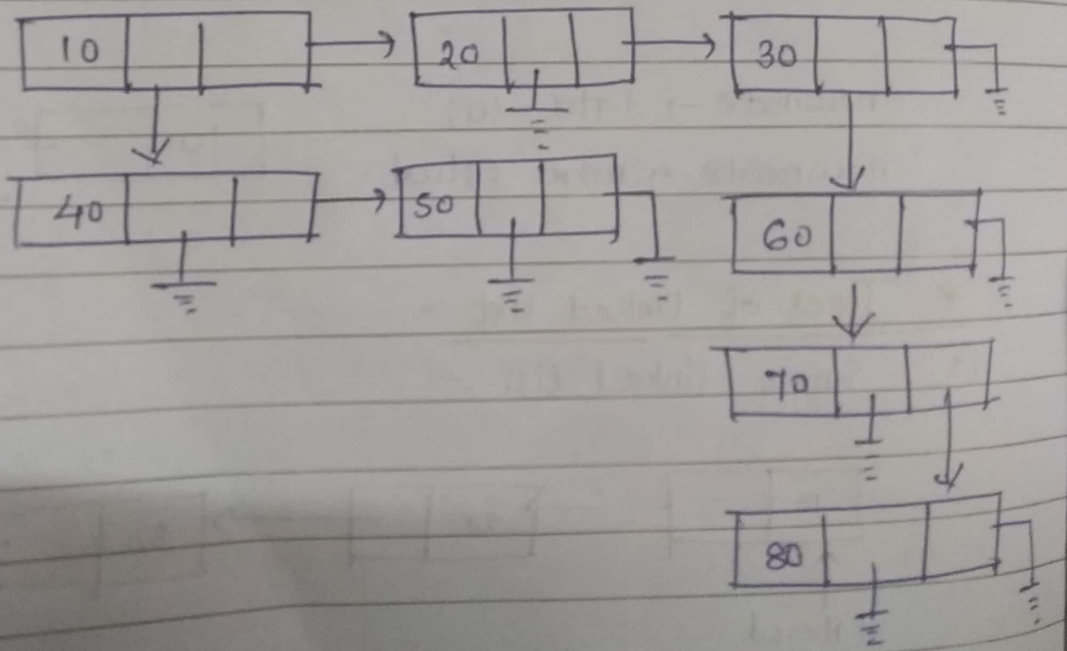


③ circular Linked list.





④ Generalized Linked list



* Singly linked list :-

→ created a node
→ Insertion
→ Deletion
→ Display

```
main ()
{
struct Node *head;
insert (&head)      // insert (head)
}
void insert (struct node **head)
{
 struct node *new;
 new = *head;       // newhead
}
```

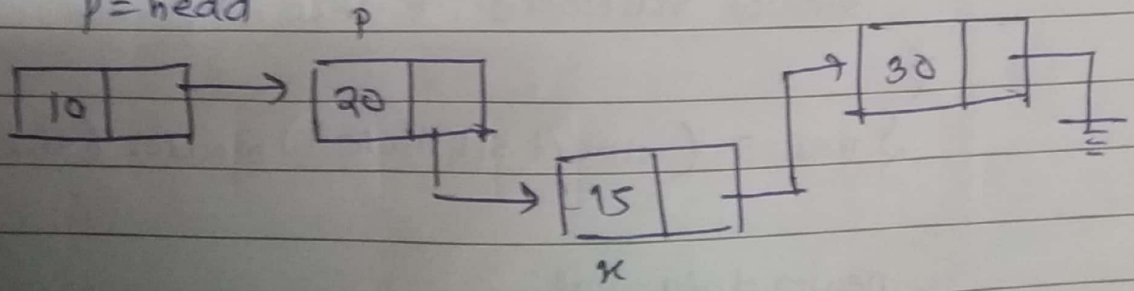
① Insertion

(a) begin

```
new → data = 5;
new → next = head;
head = new;
```

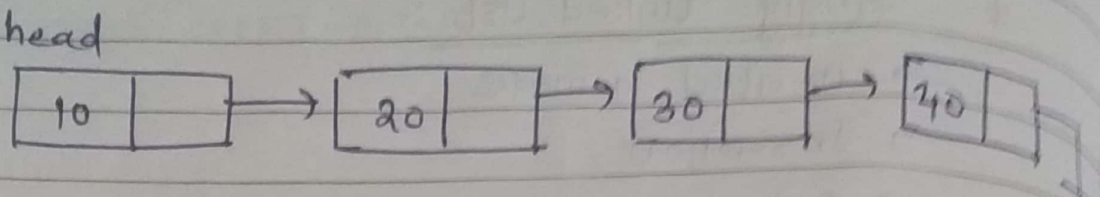
(b) between

p = head

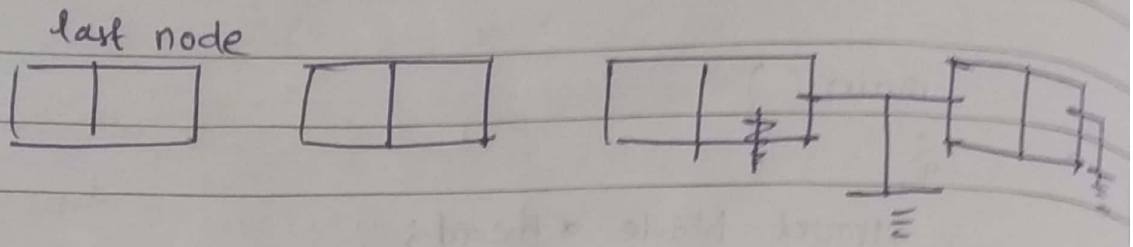



```
x → next = p → next
p → next = x;
```

- Deleting the 1st node :-

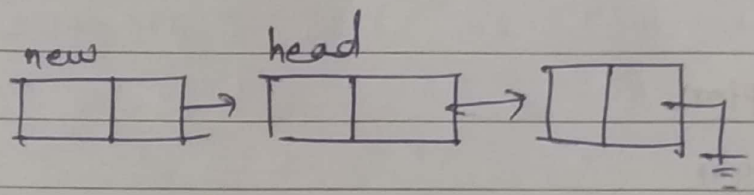



```
q -> next = p -> next
free (p);
```



* Insertion code :-

```
struct SLLNode
{
int data;
struct SLLnode *next;
};
```



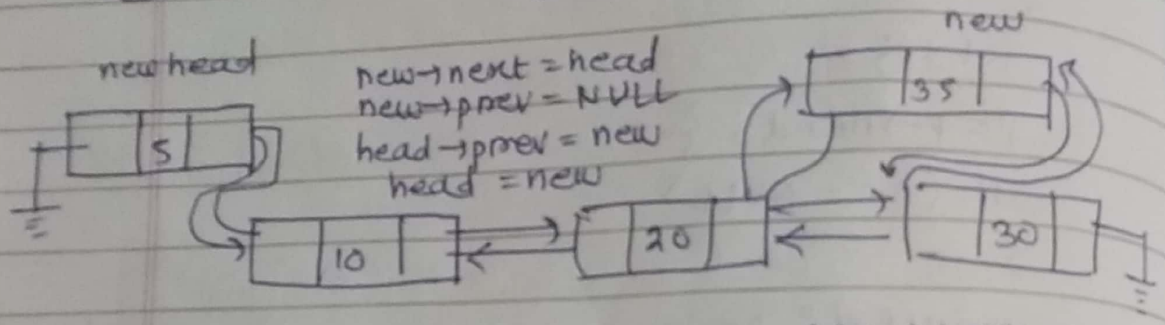

```
void insert ( struct SLLNode **head, int val,
                                    int pos)
{ int k = 1;
  struct SLLNode *new, *p;
if
{ new = (struct SLLnode*) malloc (sizeof (struct
                                    SLLnode))
new -> data = val;
new -> next = NULL;
```

```
if (pos == 1)
{
new -> next = *head;
*head = new;
return;
}
p = *head;
while ( p -> next != NULL) && i < (pos - 1)
{
 k++;
 p = p -> next; }
 if (p == NULL)
{
printf (" Desired address not found");
return;
}
new -> next = p -> next;
p -> next = new;
}
main ()
{ struct SLLNode * head;
 head = (struct SLLNode *) malloc( - - - )
 head -> data = 1;
 head -> next = NULL;
 insert (&head, 202);
}
```

* DLL :- (Doubly Linked list)

```
struct DLLNode
{
int data;
struct DLLNode *prev, *next;
};
```

① Insertion ← Begin, Between, End



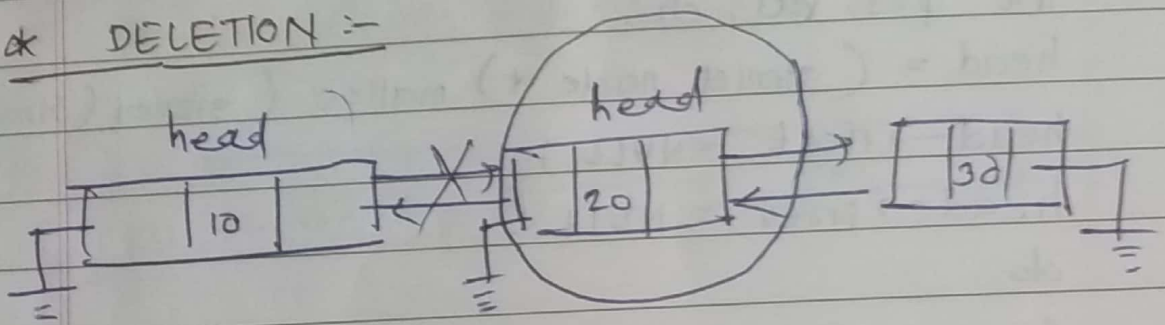

```
new→prev = p;
new→next = p→next
p→next→prev = new;
p→next = new;
```

```
void DLL insert (struct Node DLL ** head, int val,
                 int pos)
{ int k = 1;
Struct DLLNode *new, *temp;
 new = malloc (      );
new→next = new→prev = NULL;
new→data = val;
if (pos == 1)
{new→next = *head;
(*head)→prev = new;
*head = new;
return;
}
```

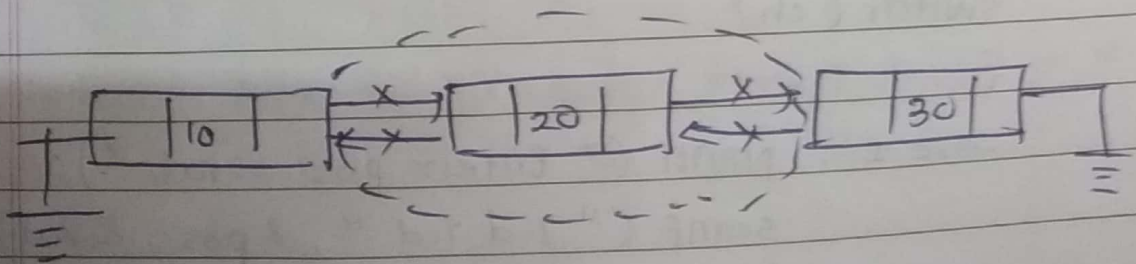
at begin

```
p = *head;
while (p->next != NULL) & k < (pos-1))
{
k++;
p = p->next;
}
new->prev = p;
new->next = p->next;
if (p->next != NULL)
    p->next->prev = new;
    p->next = new;
}
```

* DELETION :-




```
p = head;
head = head->next
head->prev = NULL;
free(p);
```



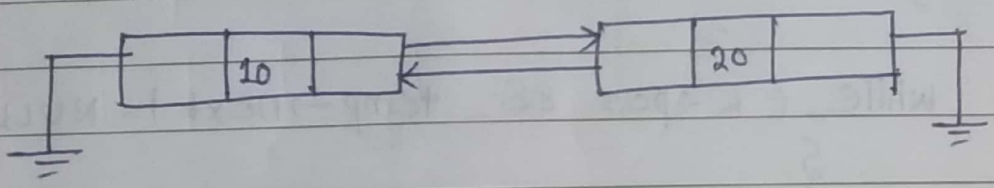
* Insertion of DLL using single pointer *

```
struct DLLnode
{
int data;
struct DLLnode *next, * prev;
};
struct DLLnode *insert( struct DLLnode*,
                        int, int);

main()
{
struct node * head;
int pos, val, ch;
head = (struct node *) malloc ( sizeof (structnode));
head→next = NULL;
head→prev = NULL;
do
{
printf ("1. Insert 2. Delete 3. trans/display
         4. Exit");
scanf ("%d", &ch);
switch (ch)
{
case 1 : printf (" Enter pos & val ");
         scanf ("%d %d ", &pos, &val);
         head = insert (head, pos, val);
         break;
```

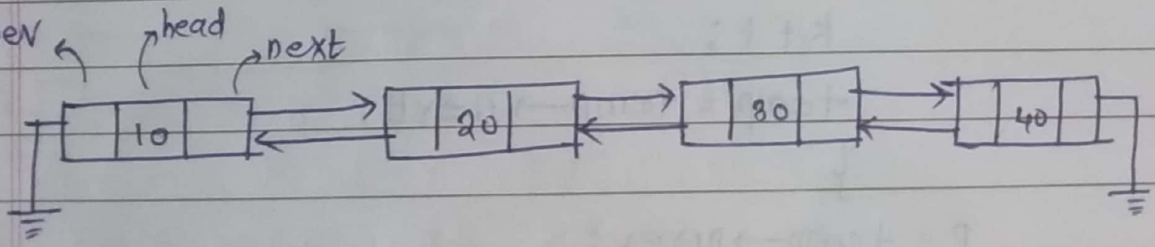
```
case 2: printf ("Enter node no. to be
                        deleted");
        scanf ("%d", &pos);
        head = delete (head, pos);
        break;

case 3 : display (head);
         break;
        }
      }
while (ch != 4);
}
```

* Deletion from DLL *

(1)



(2)




```
struct DLLnode * delete ( struct DLL node *head,
                          int pos)
{
int k = 1;
struct node *p, *temp ;
temp = head;
```

list empty

```
if (head == NULL)
{
printf ("\n List is empty");
return (head);
}
```

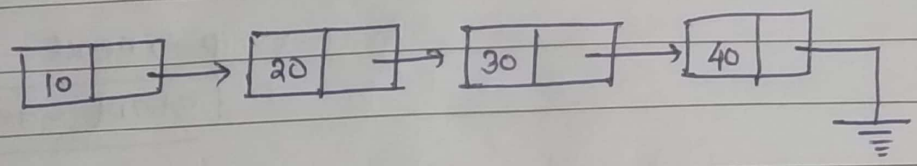
1st node deletion

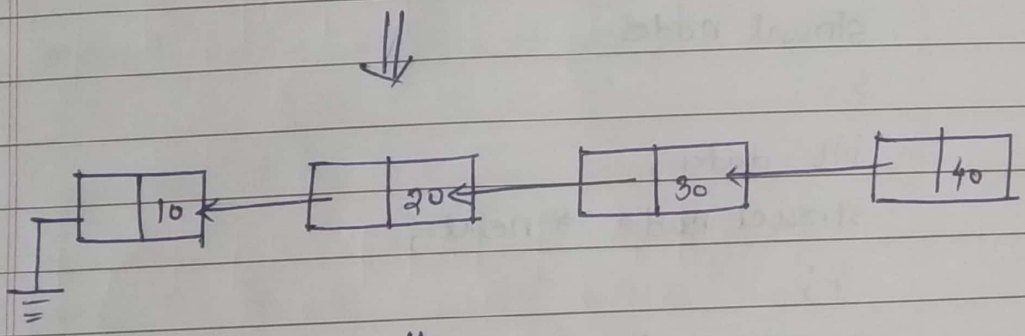
```
if (pos == 1)
{
head = head→next;
if (head != NULL)
    head → prev = NULL;
    free (temp);
    return (head);
}

while (k<pos && temp→next != NULL)
    {
     k++;
     temp = temp→next;
    }
p = temp→prev;
p→ next = temp→next;
if (temp→next != NULL)
   temp→ next→prev = p;
   free (temp);
   return (head);
}
```

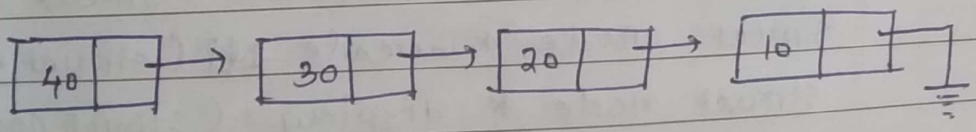
```
void display (struct DLLnode *head)
{
struct DLLnode *p;
p = head;
while (p != NULL)
   {
   printf("%d ⟶ " p→data);
   p = p→next;
   }
}
```

* Task

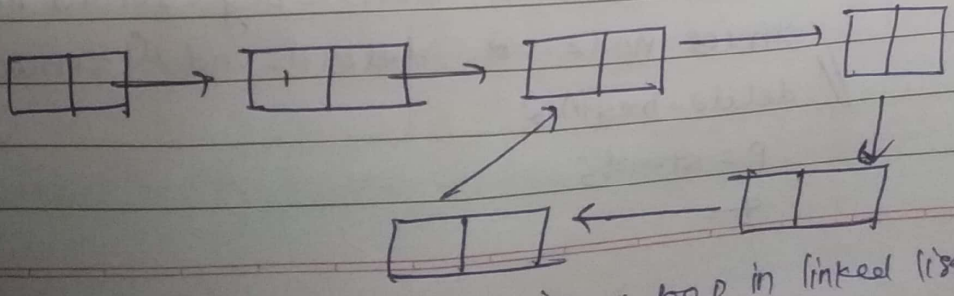
(a)







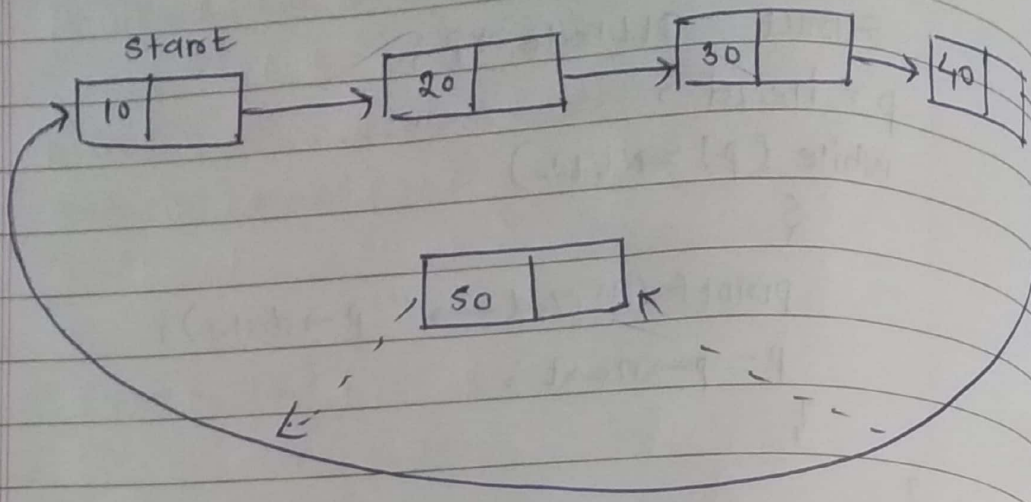
(1)



To detect whether there is a loop in linked list.

DATE / /

* CIRCULAR LINKED LIST *



insert → end / between / begin

```
while (p->next != start)
    p = p->next
temp->next = start
p->next = temp;
start = temp
```

```
struct node
{
int data;
struct node *next;
};
struct node *insert_beg (struct node*);
struct node *insert_end (struct node*);
struct node *create (struct node*);
struct node *display (struct node*);
struct node *delete_beg (struct node*);
struct node *delete_end (struct node*)
// delete-begin {
p = start;
q =
```

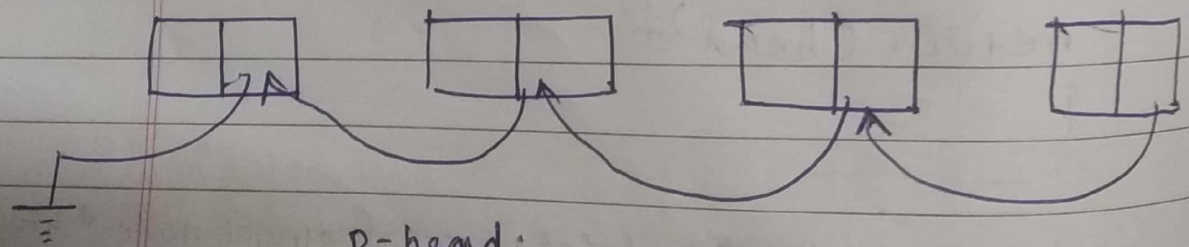
```
while (p->next != start)
  p = p->next;
  q = start;
  start = start->next;
  p->next = start;
  free (q);
// delete last
  p = start;
  while (p->next != start)
{
  q = p;
  p = p->next;
}
q->next = p->next;
free (p);

main ()
{
switch (ch)
{
case 1: head = insert (head);
        break;
case 2:
        head = insert_end (head);
        break;

struct node * insert_beg (struct node * head)
{
struct node *new_node, *p;
```

```
int num;
printf ("Enter data");
scanf ("%d", &num);
new_node = malloc (    );
new-node → data = num;
if (head == NULL)
{
new-node → next = new_node;
head = new_node;
return (head);
}
else
{
p=head;
while( p→next != head)
p = p→next;
new_node → next = head;
p → next = new_node;
head = new_node;
return (head);
}
}

// create_cll
struct node * create_cll (struct node * head)
{
struct node * new-node, *p;
int num;
printf (" Enter data");
```

```
scanf ("%d", &num);
while (num != -1)
{
new_node = (struct node *) malloc (. . . )
new_node → data = num;
if (head = = NULL)
{
new_node → next = new_node;
head = new_node;
}
else
{
p = head;
while (p → next != head)
    p = p → next;
  new_node → next = head;
  p → next = new_node;
printf
  printf ("enter data");
  scanf ("%d", &num);
 }
return (head);
}

struct node * delete_beg (struct node * head
{
struct node * new_node, *p;
p = head;
```

```
if (head == NULL)
{
printf ("linked is Empty");
return (head);
}
if (p→next == head)
{
free(p);
return ();
}
while (p→next != head)
    p = p→next;
    q = head;
    head = head→next;
    p→next = head;
    free(q);
    return (head);
}
```

* Reverse

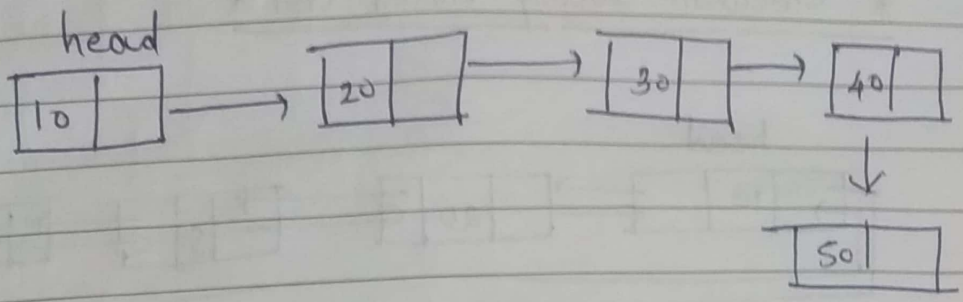


```
p=head;
q= p→next;
r = q→next;
p→next=NULL;
```

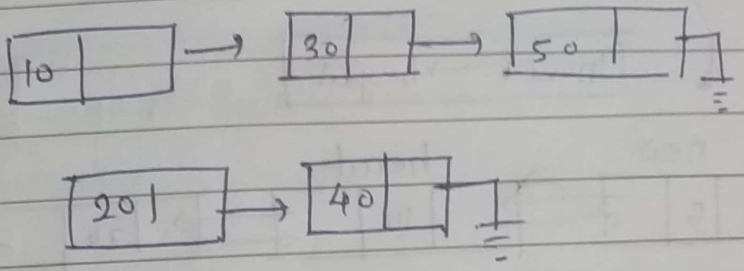
```
while (q != NULL)
{ q->next = p;
  p = q;
  q = r;
  r = r->next;
}
```

Q)

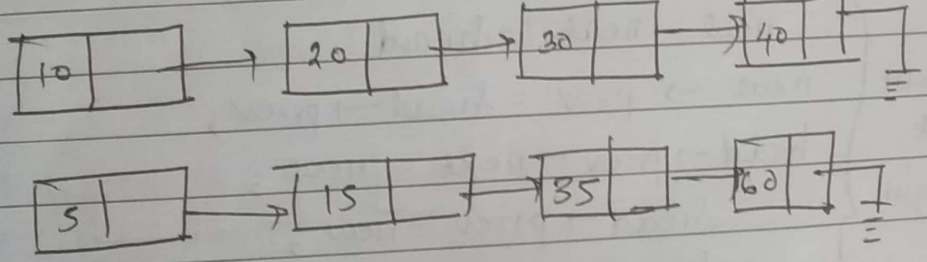
①



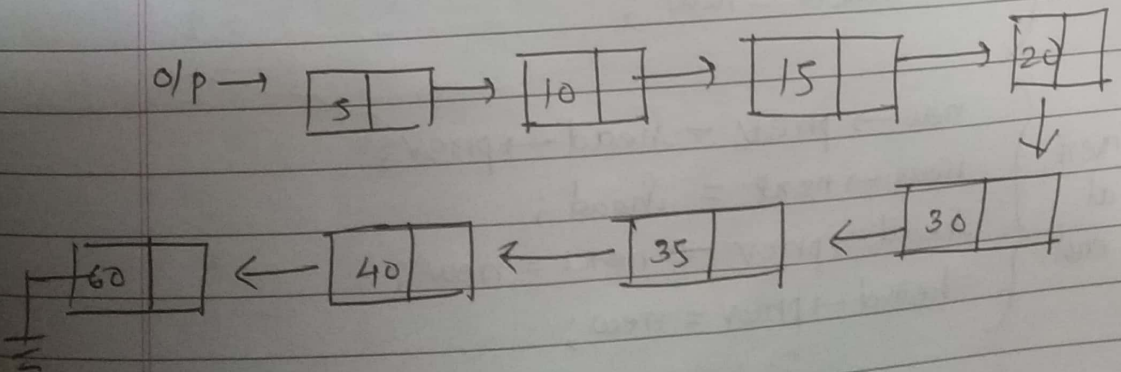
o/p →

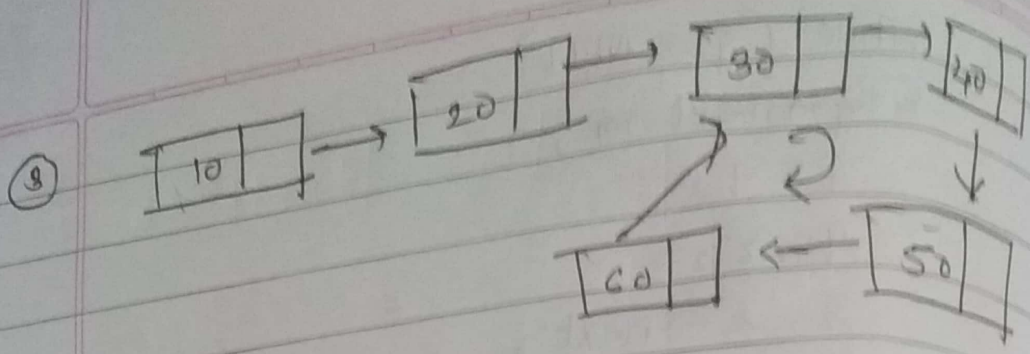


②

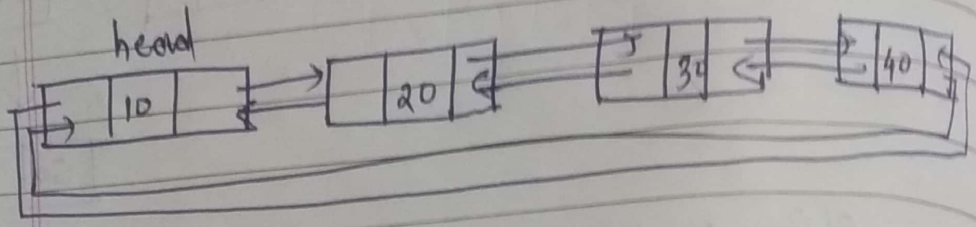


o/p →

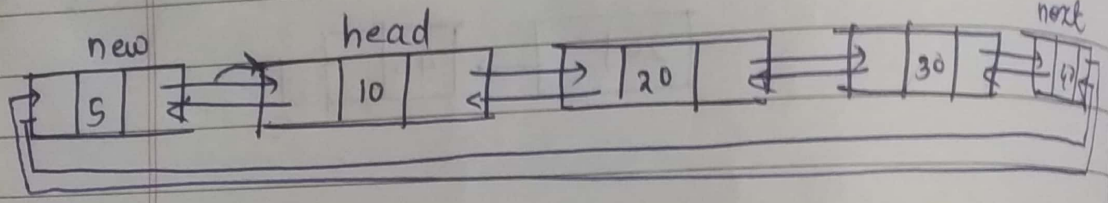




* Circular Doubly Linked List *



Insert → begin



insert at begin

```
new → next = head;
new → prev = head → prev;
head → prev → next = new;
head → prev = new;
head = new;
```

insert at end.

```
new → prev = head → prev;
new → next = head;
head → prev → next = new;
head → prev = new;
```

Deleting at front

1st node deletion

```
head = head → next ;
head → prev = p → prev ;
p → prev → next = head ;
free (p) ;
```

Deleting at end

last node deletion.

```
p = head → prev ;
p → prev → next = head ;
h → prev = p → prev ;
free (p) ;
```

* Generalized linked list (GLL) *

A = ( )

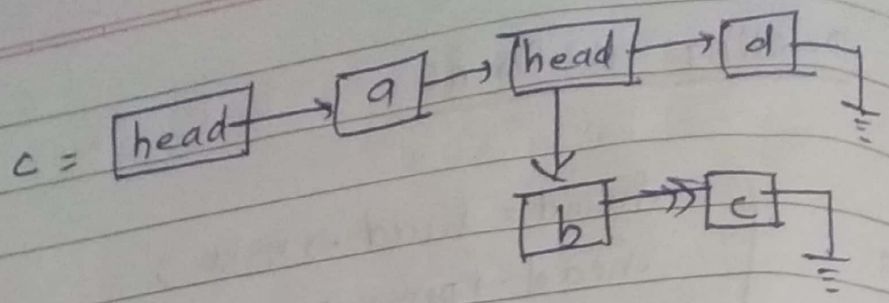
B = (a, b, c)

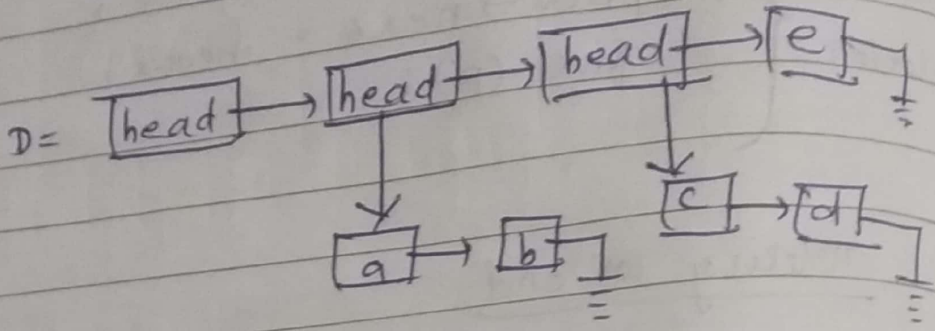
C = (a, (b, c), d)

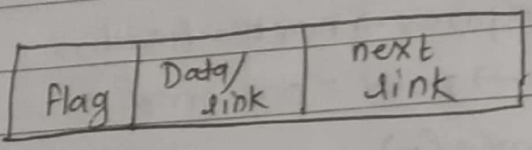
D = ((a, b), (c, d), e)

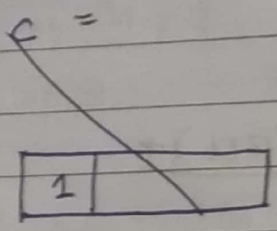
A = [head]

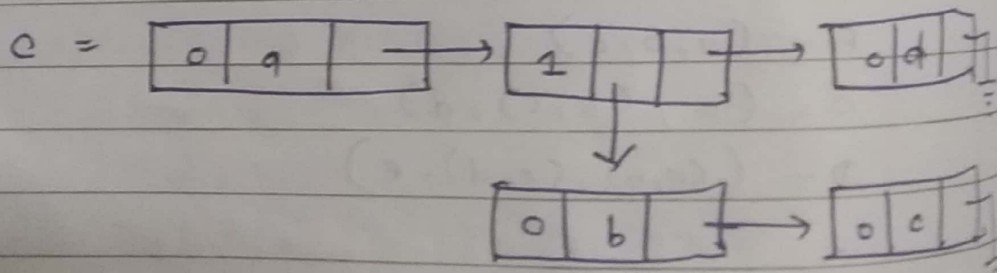
B = [head] → [a] → [b] → [c] ⏚











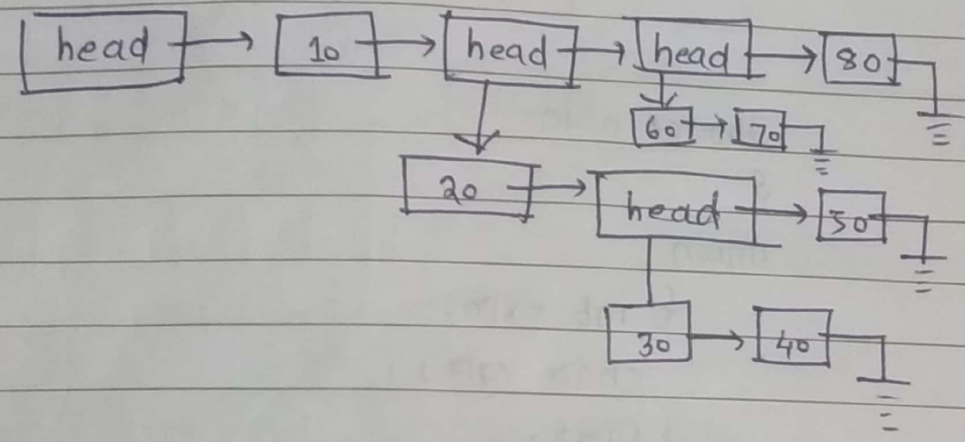

```
Struct node
{ int flag;
union {
        int data;
        struct node * down;
}un;
```

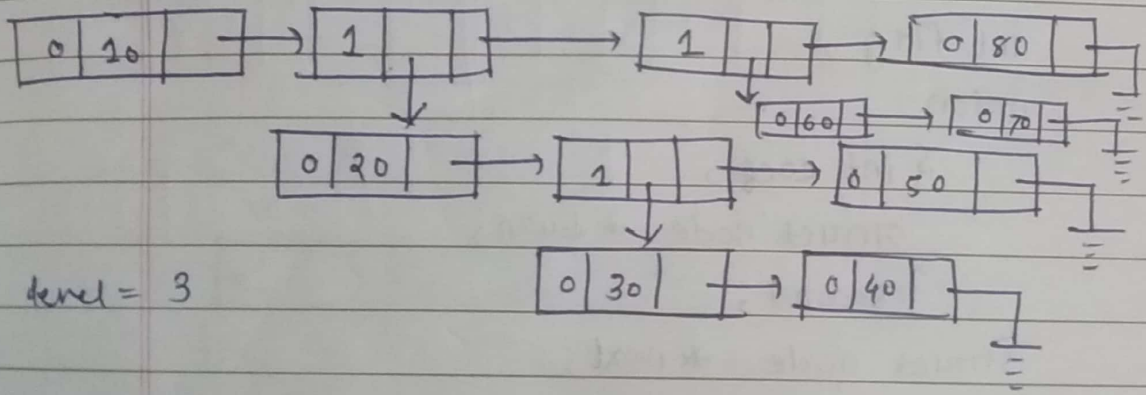
struct node *next;
};

* A = { 10, (20, (30, 40), 50), (60, 70), 80 }

B = { X, (A, B), ( ), C, (D, E) }

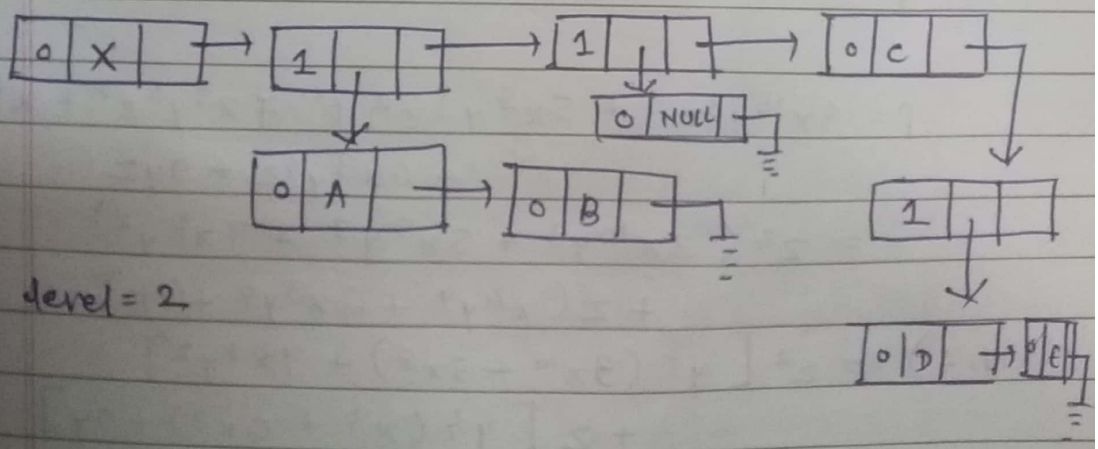
A=





level = 3

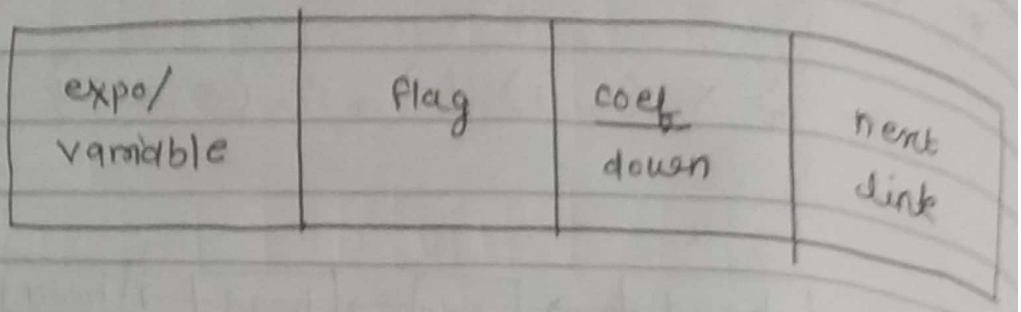
B =



level = 2

level = no. of down links + 1.

* $3x^{10}y^3z^2 + 5x^8y^3z^2 + 7x^8y^2z^2 + x^4y^4z + 6x^3y^4z + 9yz$

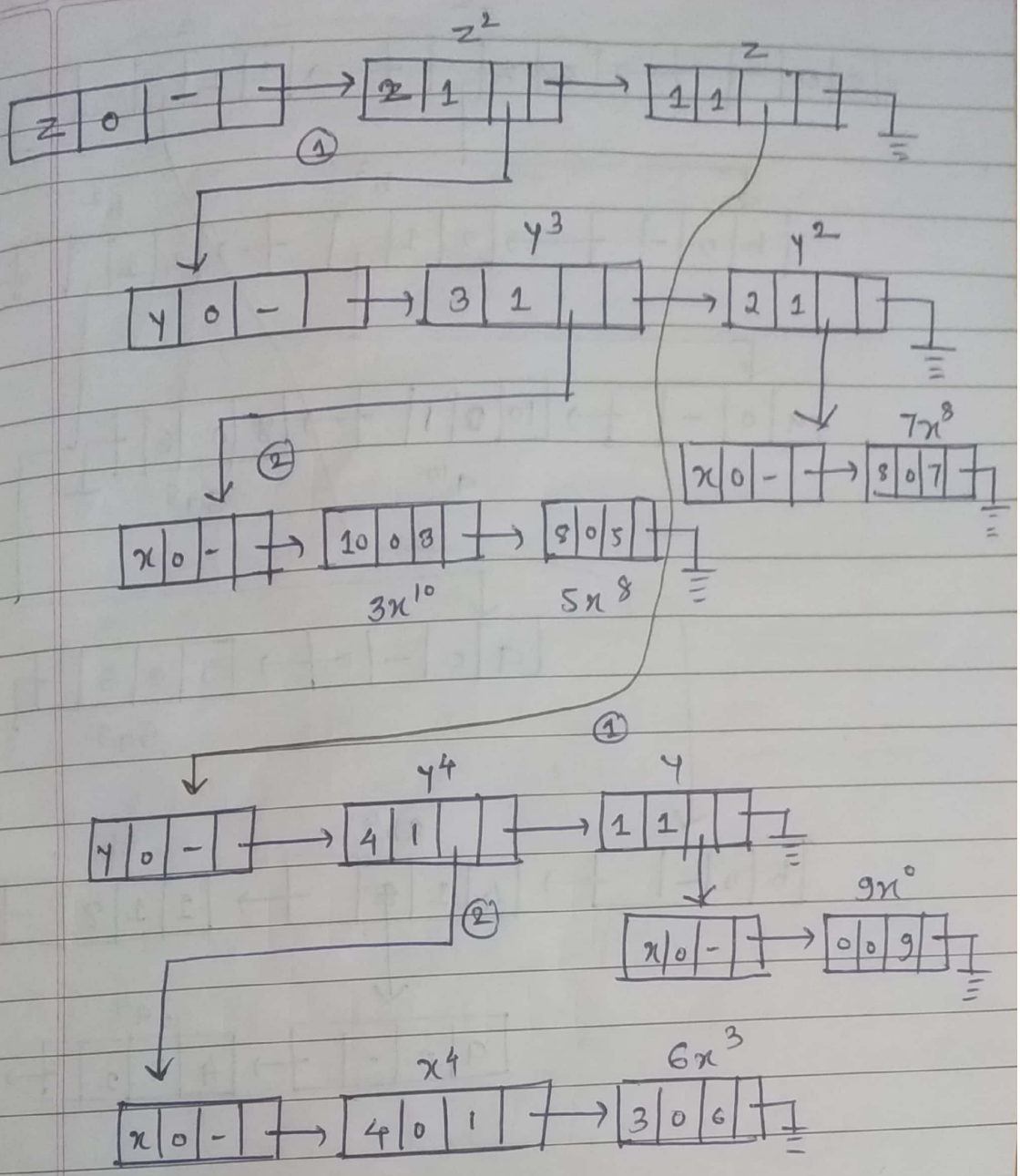



```
struct node
{
union
    { int expo;
      char var;
    } first;
int flag;
union
    { int coef;
      struct node *down;
    } second;
struct node *next;
}
```

$P = 3x^{10}y^3z^2 + 5x^8y^3z^2 + 7x^8y^2z^2 + x^4y^4z + 6x^3y^4z + 9yz$

$= z^2(3x^{10}y^3 + 5x^8y^3 + 7x^8y^2) + z(x^4y^4 + 6x^3y^4 + 9y)$

$P(x,y,z) = z^2[y^3(3x^{10} + 5x^8) + 7x^8y^2] + z[y^4(x^4 + 6x^3) + 9y]$

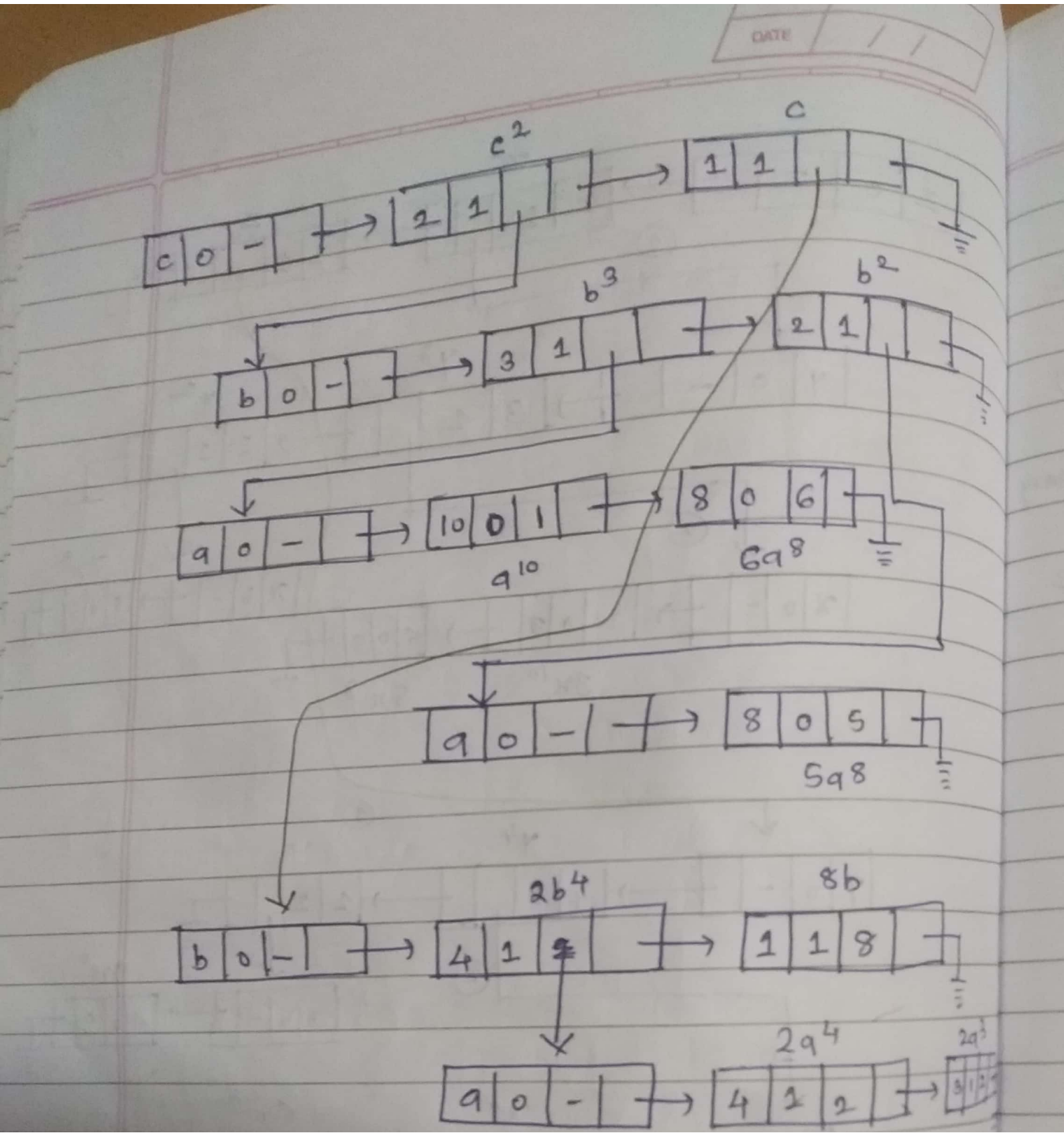


depth = level = 2 + 1 = 3

* $P(a,b,c) = a^{10}b^3c^2 + 6a^8b^3c^2 + 5a^8b^2c^2 + 2a^4b^4c + 2a^3b^4c + 8bc$

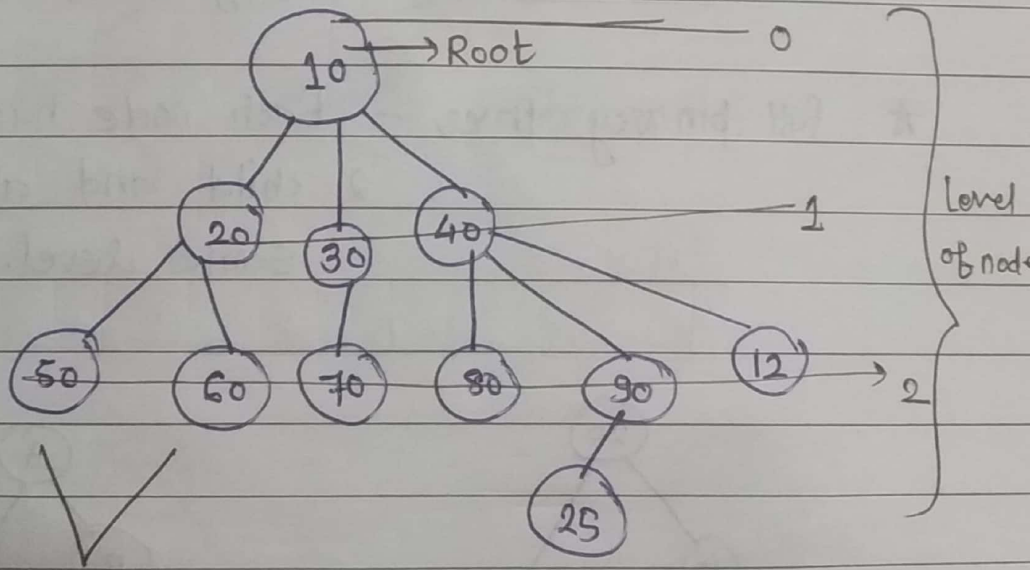
$= c^2(a^{10}b^3 + 6a^8b^3 + 5a^8b^2) + c(2a^4b^4 + 2a^3b^4 + 8b)$

$= c^2[b^3[a^{10} + 6a^8] + 5a^8b^2] + c[2b^4(2a^4 + 2a^3) + 8b]$



* TREES *

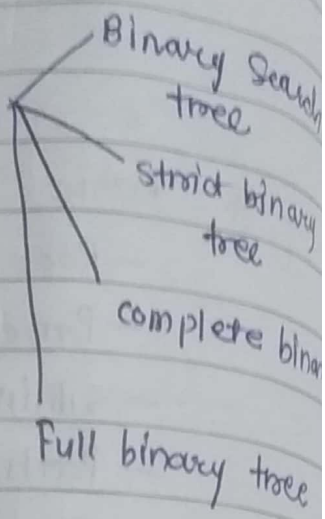
* Basic Terminologies
  - Root
  - Parent
  - siblings
  - Path
  - Degree of Node
  - Height
  - Ancestors
  - Descedents



siblings → common parent
leaf → Node having no child
degree = deg of ancestors + deg. of descendents

* Types of Trees

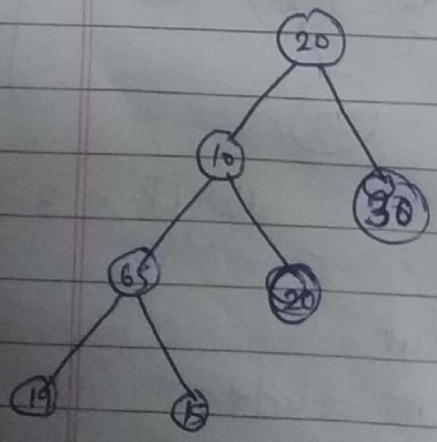
(1) Binary Tree —— 2 child or less
(2) m-way tree
—— node m childs



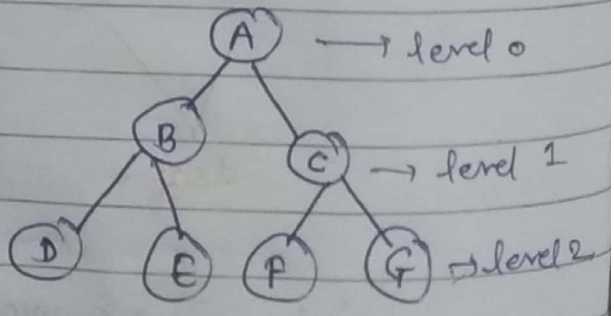
* strict binary tree – each node has exactly two or no child.

* complete binary tree – leaf nodes are at height h / h-1.

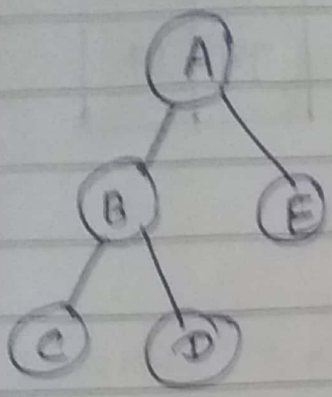
* full binary tree – Each node has exactly 2 child and all are at same level.



Strict binary tree

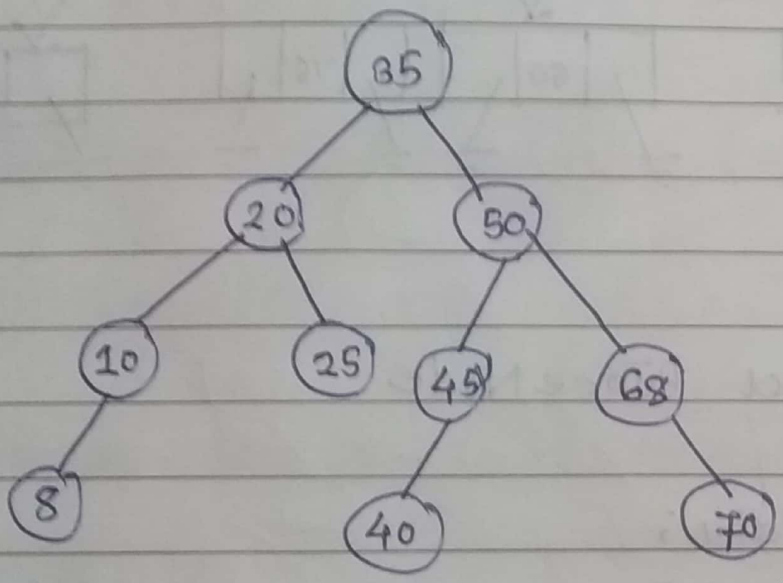


Full binary tree

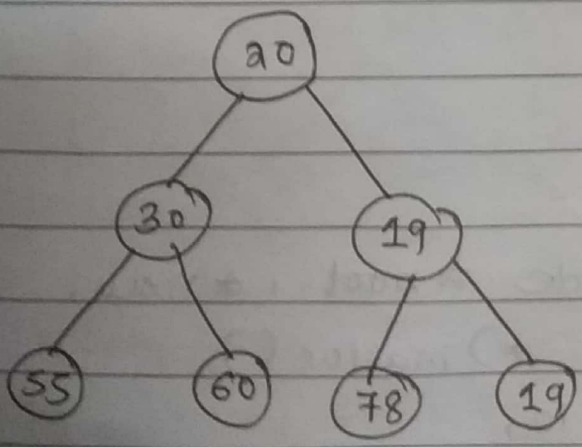


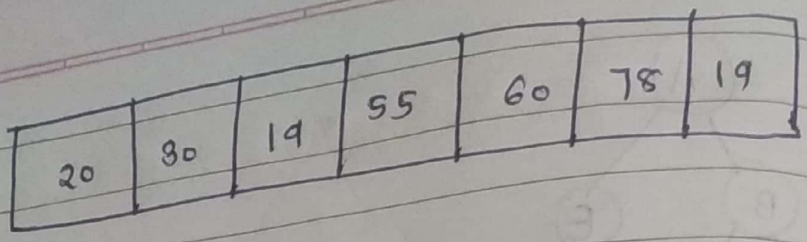
complete binary tree

* binary search tree — left child < parent < right child.

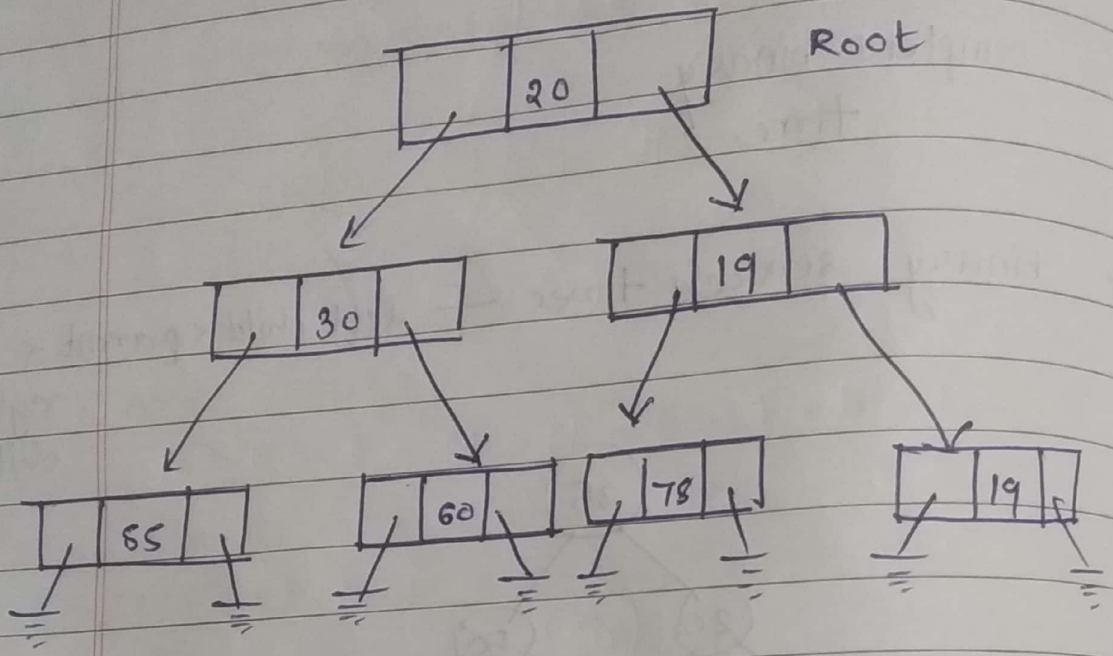


*





Array



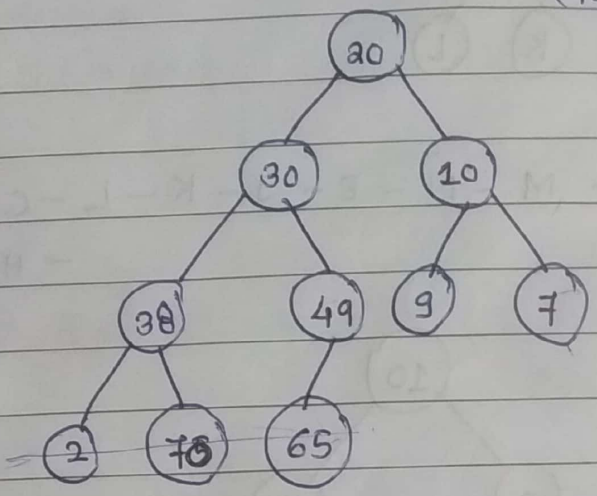

```
struct BTreeNode
{
int data;
struct BTreeNode * left, * right;
};

main()
{
Struct BTreeNode * root, * new;
root = (struct *) malloc()
rooot → data = 10;
root → left
```

PAGE No.
DATE / /

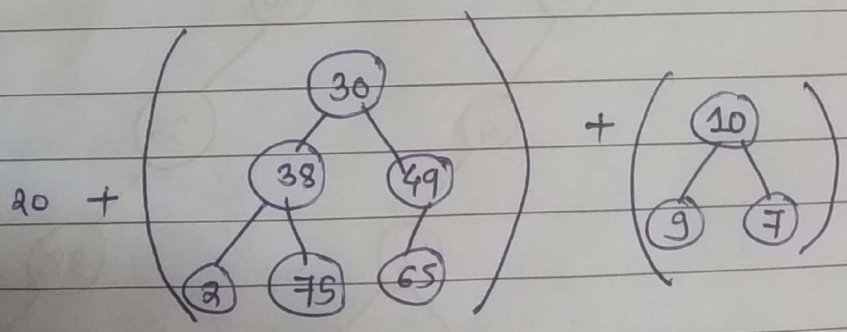
* Binary Tree traversals *

(1) Preorder traversals (P - L - R)

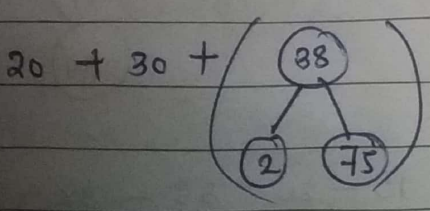
(Parent - left child - Right child)



⇒ 20 - 30 - 38 - 2 - 75 - 49 - 65 - 10 - 9 - 7



P L R

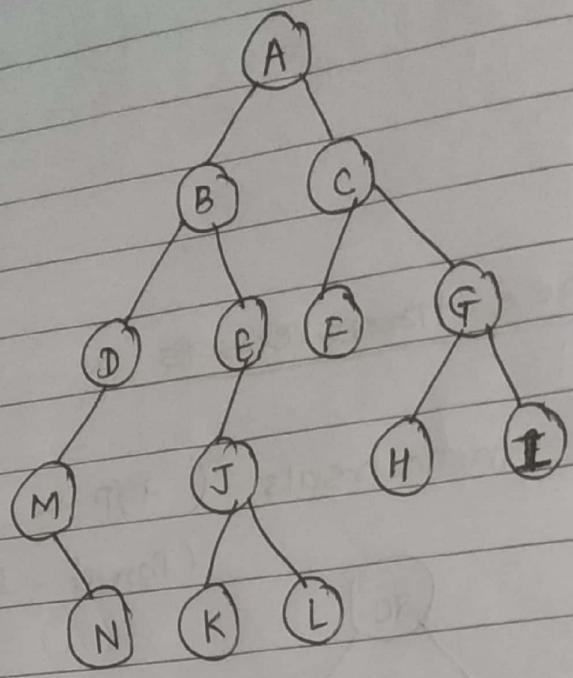


+ 10 + 9 + 7

20 + 30 + 38 + 2 + 75 + 49 + 65 + 10 + 9 + 7

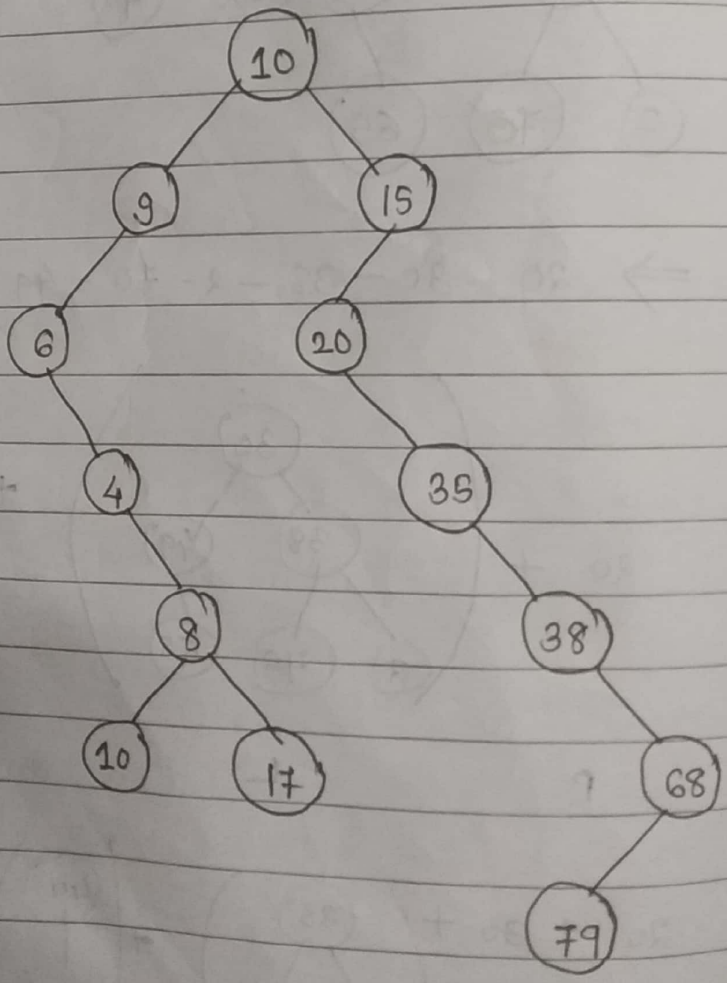
⇒ 20 - 30 - 38 - 2 - 75 - 49 - 65 - 10 - 9 - 7

*



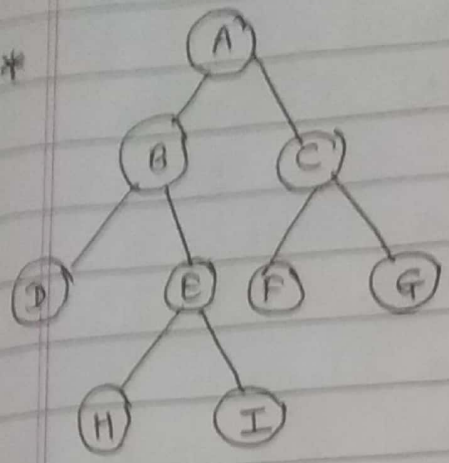
A - B - D - M - N - E - J - K - L - C - F - G - H - I

*

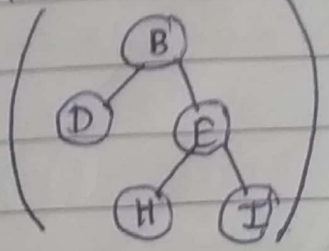


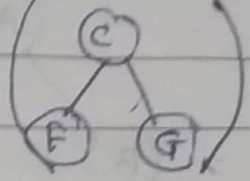
10 - 9 - 6 - 4 - 8 - 10 - 17 - 15 - 20 - 35 - 38 - 68 - 79

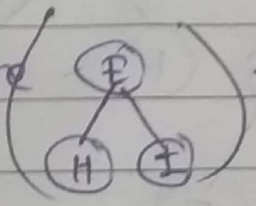
*



display
⇒ A + pre (root→left) + pre (root→right)





⇒ A + B + pre(D) + pre  + C + Pre(F) + pre(G)

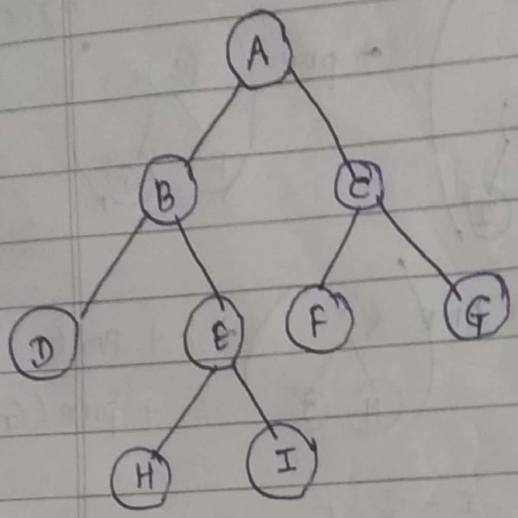
⇒ A - B - D - E - H - I - C - F - G

* Recursive Approach *

```
struct node
{
int data;
struct node, * left, * right }

void preorder (struct node *root)
{
 If (root)
```

```
{printf("%d", root->data);
preorder(root->left);
preorder(root->right);
}
}
```

* Non - Recursive

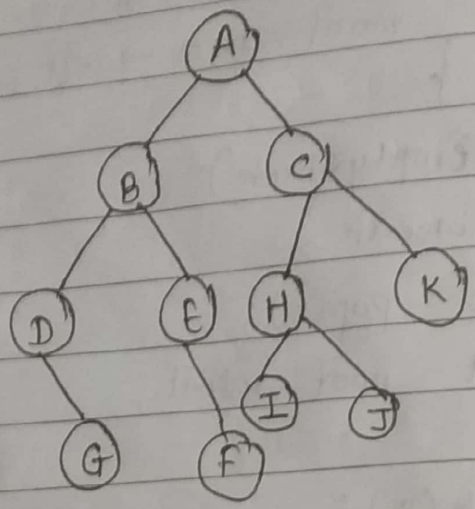


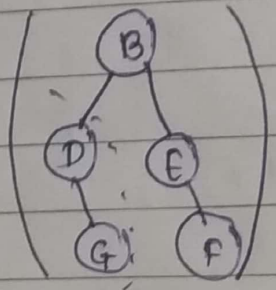
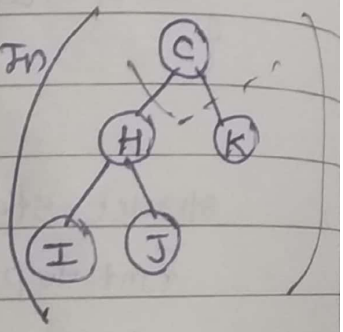

```
While (1)
{
 while (root)
     { printf (root);
       push (root);
       root = root->left;
     }
if (stack empty)
    break;
root = pop(s);
root = root->right;
}
```

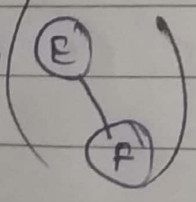
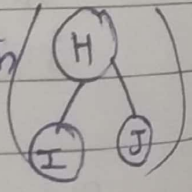
```
void preorder(struct node *root)
{
 while(1)
    {
     while(root)
          { printf("%d", root->data);
            push(s, root);
            root = root->left;
          }
     if(Emptystack)
        break;
     root = pop(s);
     root = root->right;
    }
 deleteStack(s);
}

struct stack
 {int top;
 struct node *data[30];
 };

struct node
 { int data;
  struct node *left, *right;
 };
struct stack *s = NULL;
struct node *pop(struct stack *s)
 {
  struct node *temp;
```

```
temp = s→data [s→top];
s→top = (s→top) - 1;
return (temp);
}
```

(2) Inorder Traversal (L-P-R)



⇒ Inorder  + A + In 

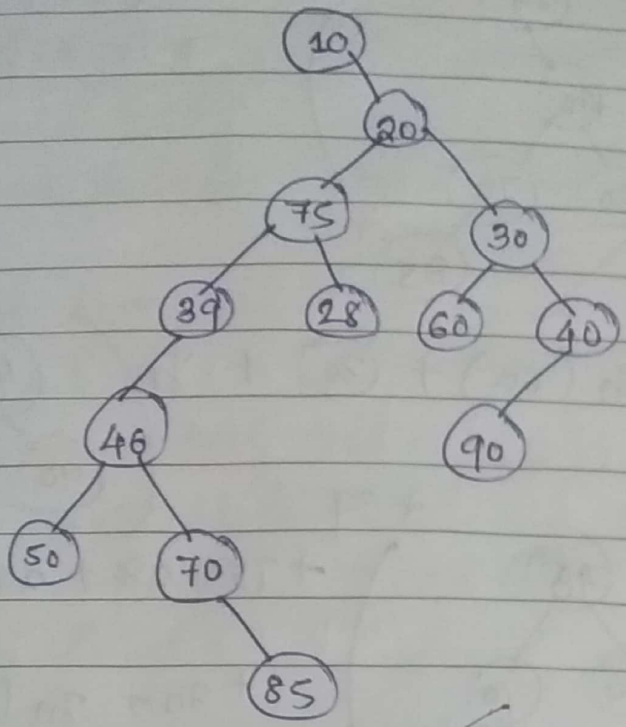
⇒ In  + B + In  + A + In 

+ C + K

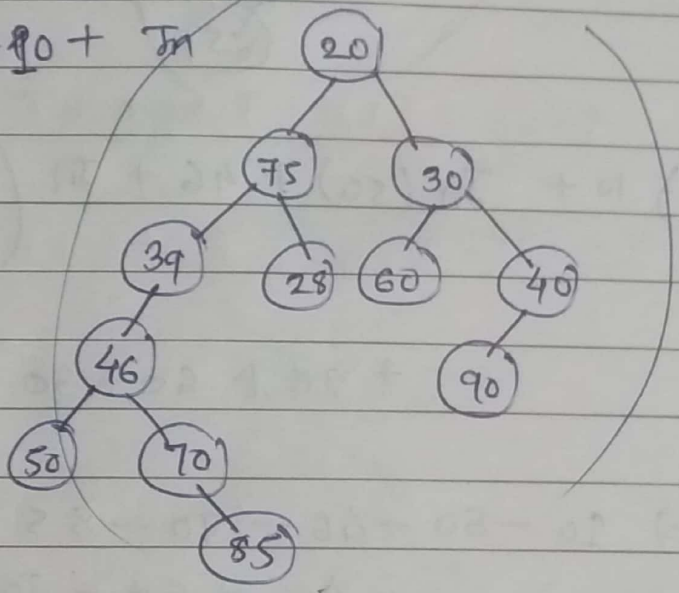
⇒ D + G + B + E + F + A + I + H + J + C + K

⇒ D - G - B - E - F - A - I - H - J - C - K

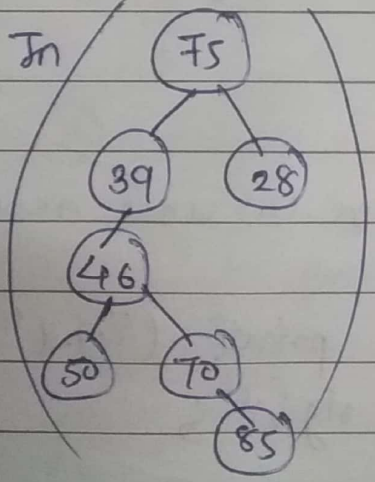
*



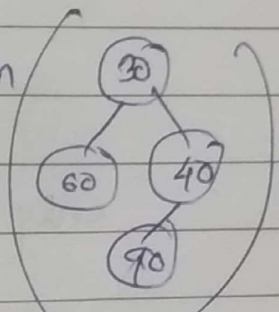
⇒ In(null) + 10 + In

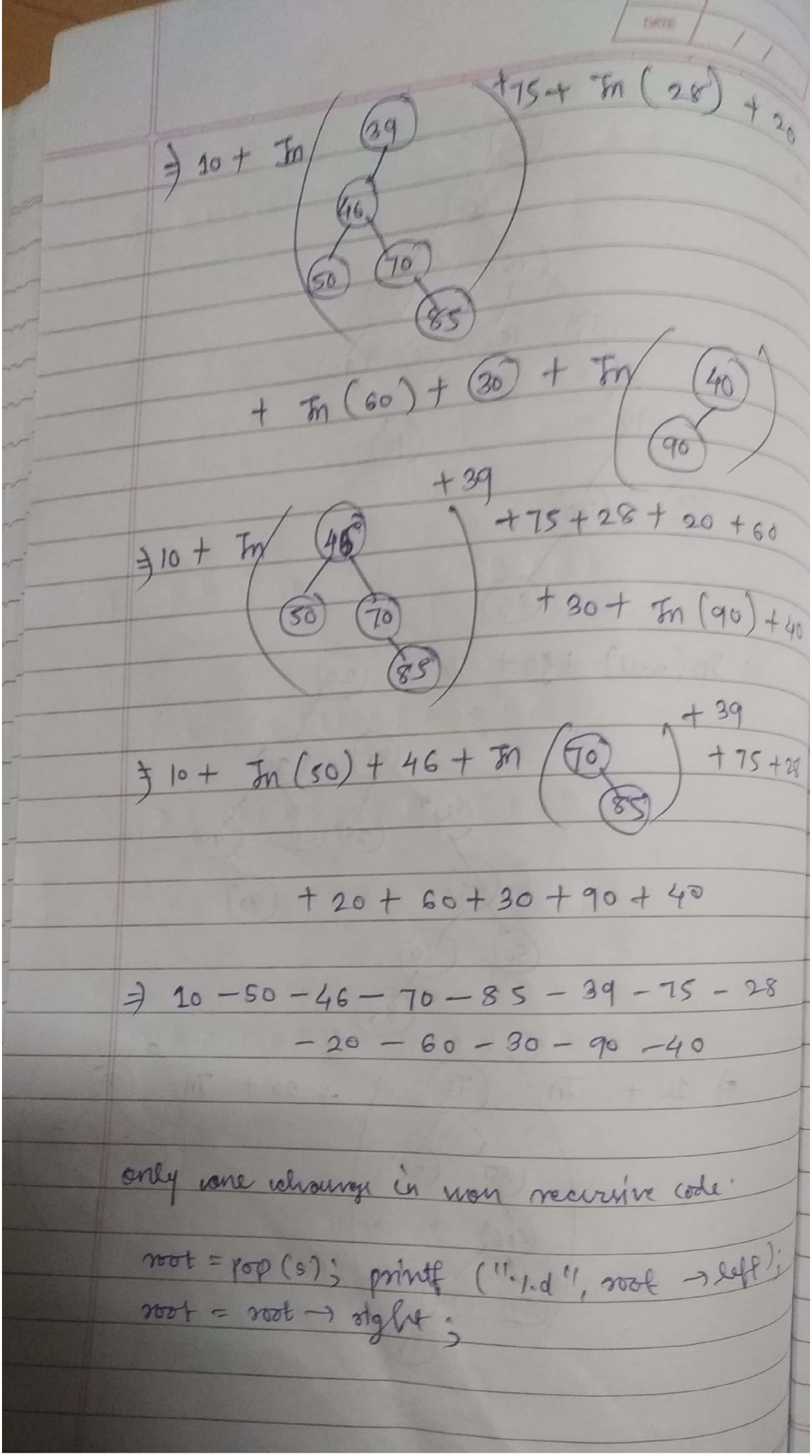


⇒ 10 + In



+ 20 + In

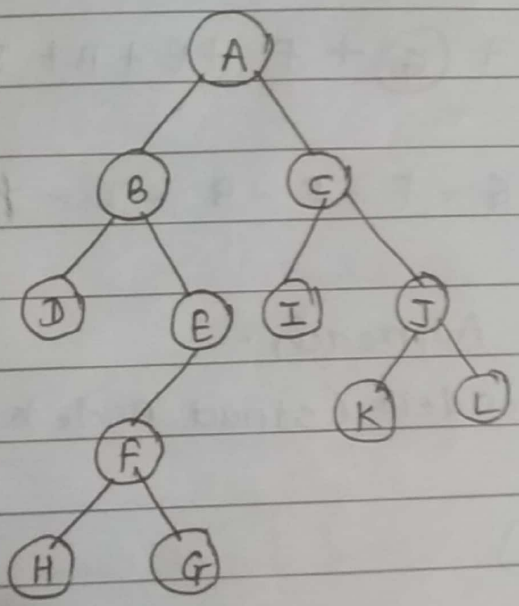


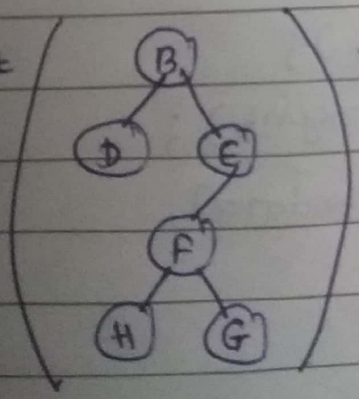
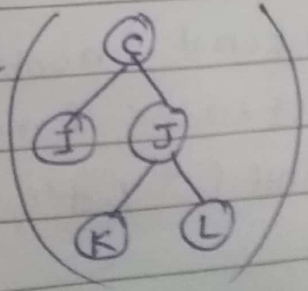


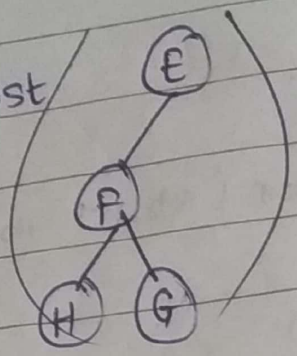
Recursive

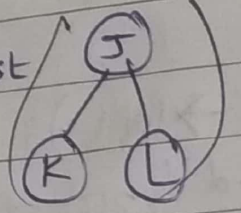
```
void inorder (struct node *root)
{
 if (root)
    {
    inorder( root->left );
    printf (" %d ", root -> data);
     inorder ( root -> right);
    }
}
```

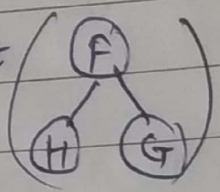
* Postorder Traversal * (L-R-P)



=) post  + post  + A

⇒ ~~D+~~ post(D) + post  + B + post(I)

 + C + A

⇒ D + post  + E + B + I + post(~~I~~)

+ post(L) + J + C + A

⇒ D + (H) + (G) + F + E + B + I + ~~K~~ + E L + ^(C+A) (+J)

⇒ D - H - G - F - E - B - I - ~~K~~ - L - ~~J~~ - ~~C~~ - A

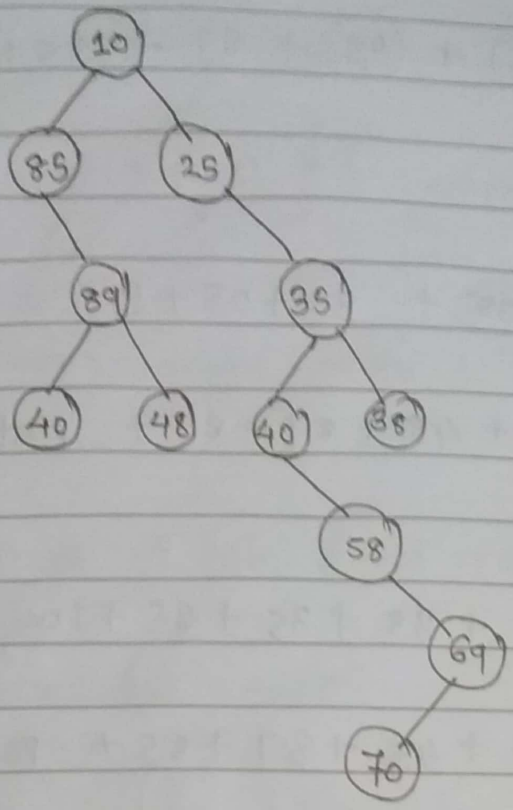
⇒ Recursive Approach.

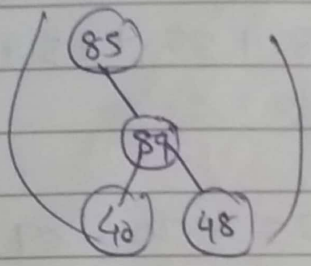
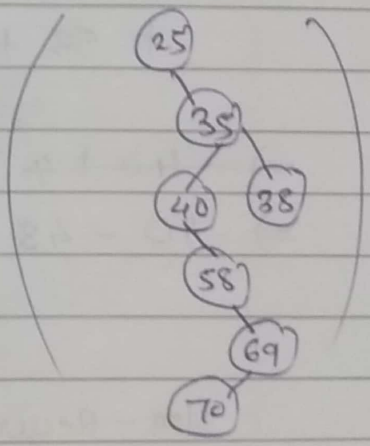
```
void postorder (struct node * root)
{
 if (root)
 {
 postord (root→left);
postorder (root→right);
printf ("%d", root→data);
 }
}
```

40 - 48 - 89 - 85 - ~~10~~ - 70 - 69 - 58 - 40 - 38 - 35 - 25 - 10

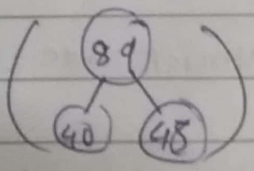
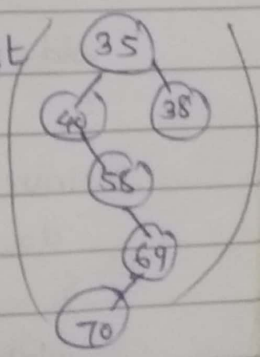
DATE

*

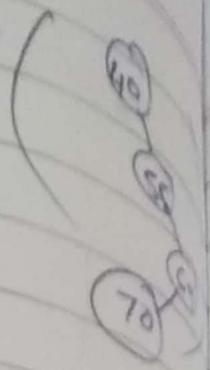


⇒ post  + post 

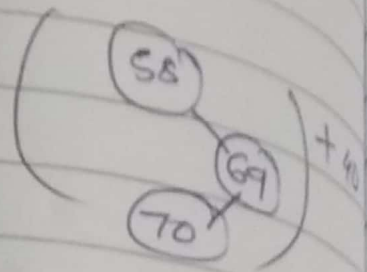
+ 10

⇒ ~~85~~ post  + 85 + post 

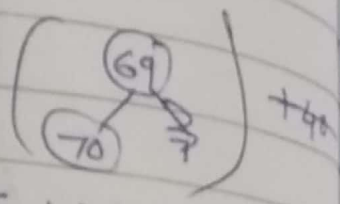
+ 25 + 10

⇒ (40) + (48) + 89 + 85 + post 

+ (38) + 35 + 25 + 10

⇒ 40 + 48 + 89 + 85 + post 

+ 38 + 35 + 25 + 10

⇒ 40 + 48 + 89 + 85 + post 

+ 58 + 40 + 38 + 35 + 25 + 10

~~⇒ 40 + 4~~

⇒ 40 - 48 - 89 - 85 - 70 - 69 - 58 - 40
- 38 - 35 - 25 - 10 .

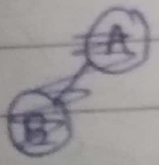
Non - Recursive Approach

```
void postorder (struct node *root)
{
struct node *previous = NULL;
  do
{
while (root ! NULL)
    {
```

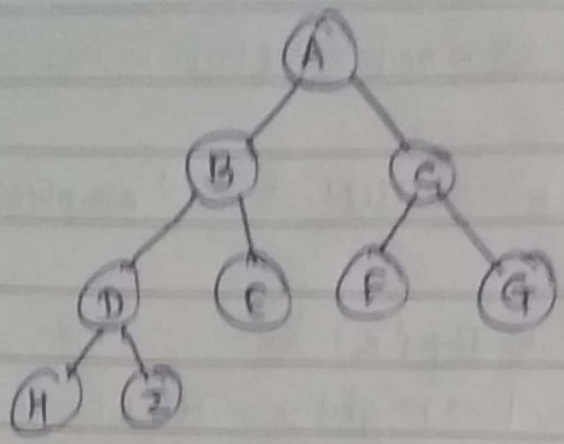
```
        push (s, root);
        root = root->left;
      }
while (root == NULL && !emptystack(s))
      {
       root = top(s);
       if (root->right == NULL || root->right
                                  == previous)
        {
        printf ("%d", root->data);
        pop (s);
        previous = root;
        root = NULL;
        }
       else
          root = root->right;
      }
      }
      while (!emptystack(s));
delete (s);
}
```

* Construct binary trees

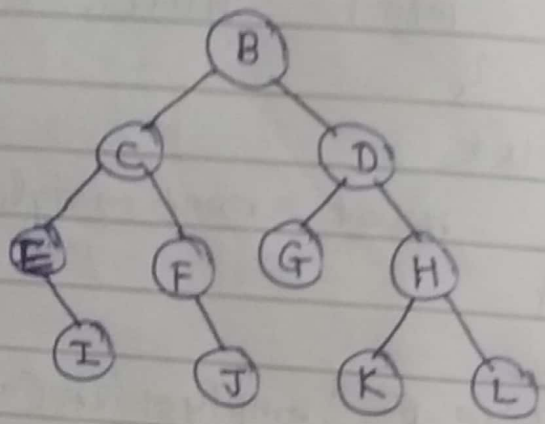
(1) PDE - A B D H I E C F G



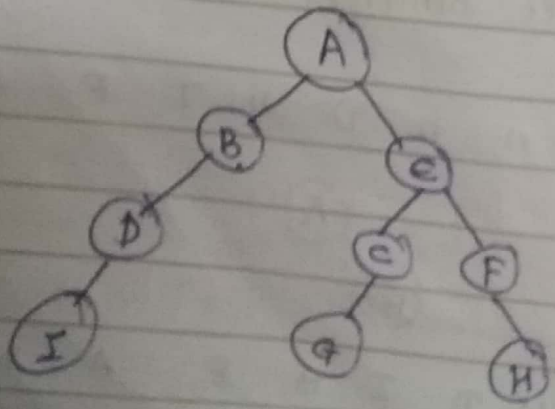
In - H D I B E A F C G



(2) post → I E J F C G K L H D B
In → E I C F J B G D K H L



(3) pre - A B D I E C G F H
post - I D B G C H F E A

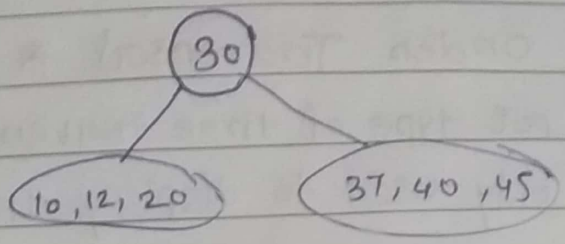


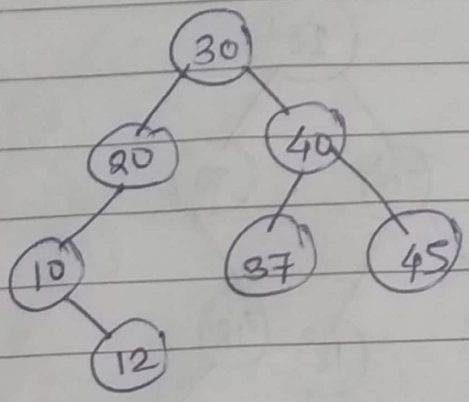
* In preorder & postorder, trees are not unique

PAGE NO.
DATE / /

LPR * Inorder - 10, 12, 20, 30, 37, 40, 45

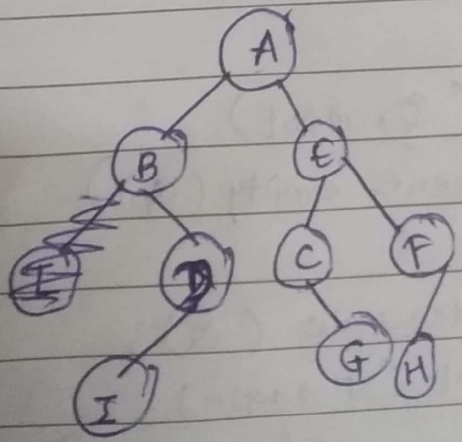
PLR preorder - 30, 20, 10, 12, 40, 37, 45





LPR * Inorder - B, I, D, A, C, G, E, H, F

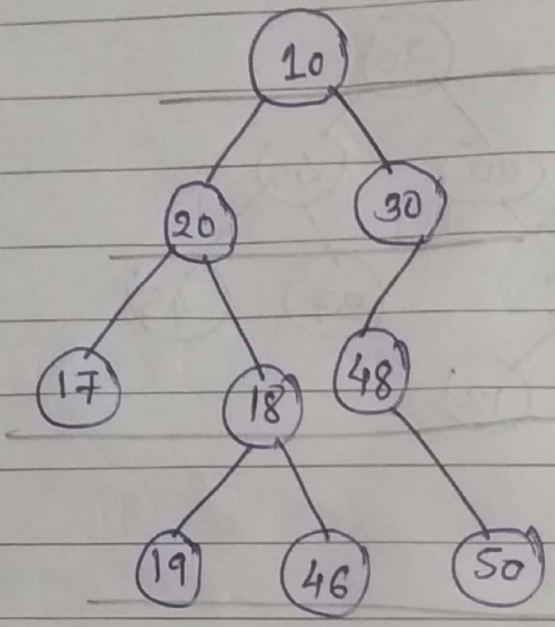
LRP postorder - I, D, B, G, C, H, F, E, A



~~10, 18, 30, 4, 16, 19, 18, 17~~

* Level order Traversal *

(It is not type of tree traversal.
It is only used to display tree level wise)



⇒ 10, 20, 30, 17, 18, 48, 19, 46, 50

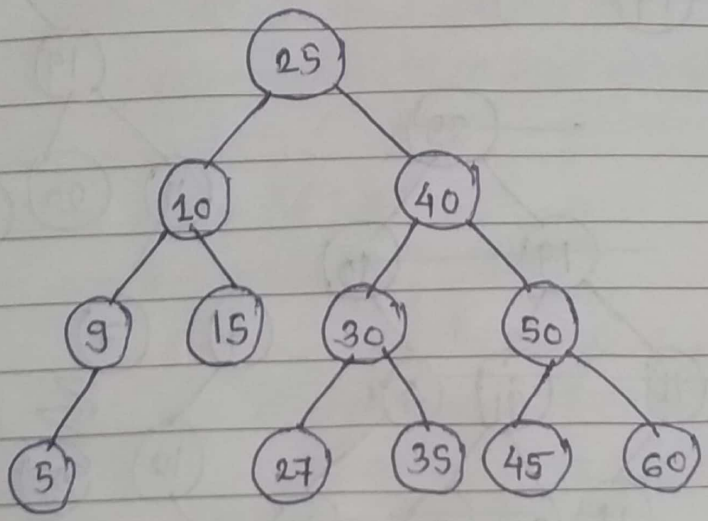
~~Insert~~

```
enqueue (Q, root)
while (!Queue empty (Q))
    {
      p = dequeue (Q);
      print (p→data);
      if (p→left != NULL)
         Enqueue (Q, p→left)
    if (p→right != NULL)
         Enqueue (Q, p→right);
}
```

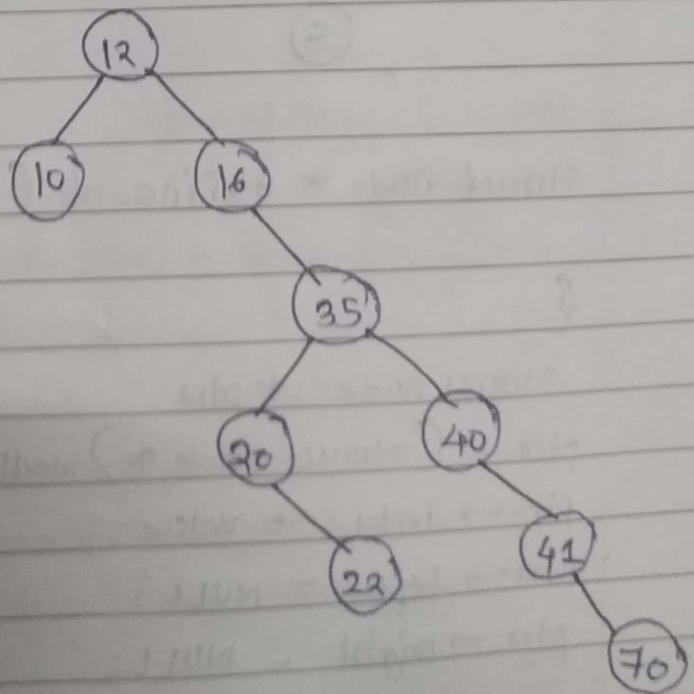
PAGE No.
DATE / /

* BST (Binary Search Tree) *

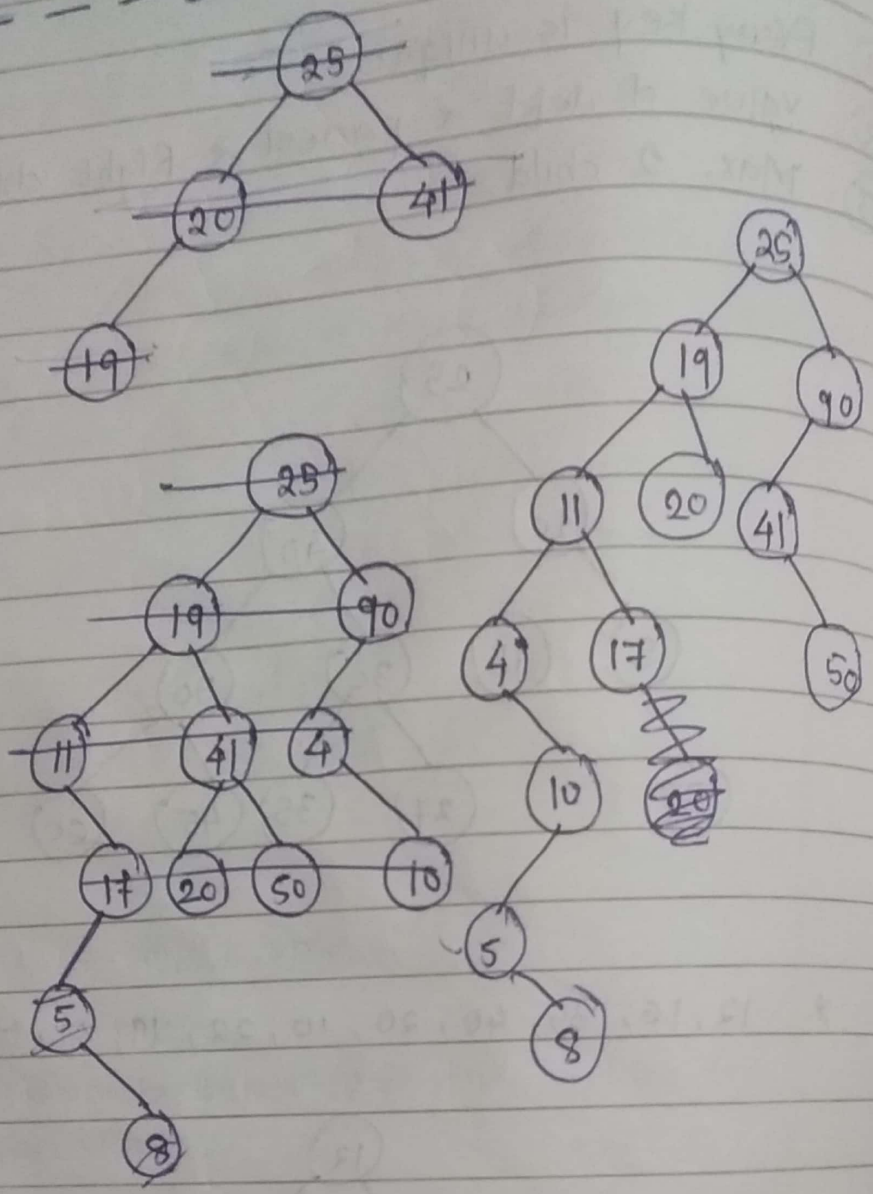
1. Every key is unique.
2. value of left < parent < Right child
3. Max. 2 child



* 12, 16, 35, 40, 20, 10, 22, 19, 41, 70



* 25, 90, 19, 11, 41, 4, 17, 20, 10, 50, 5, 8

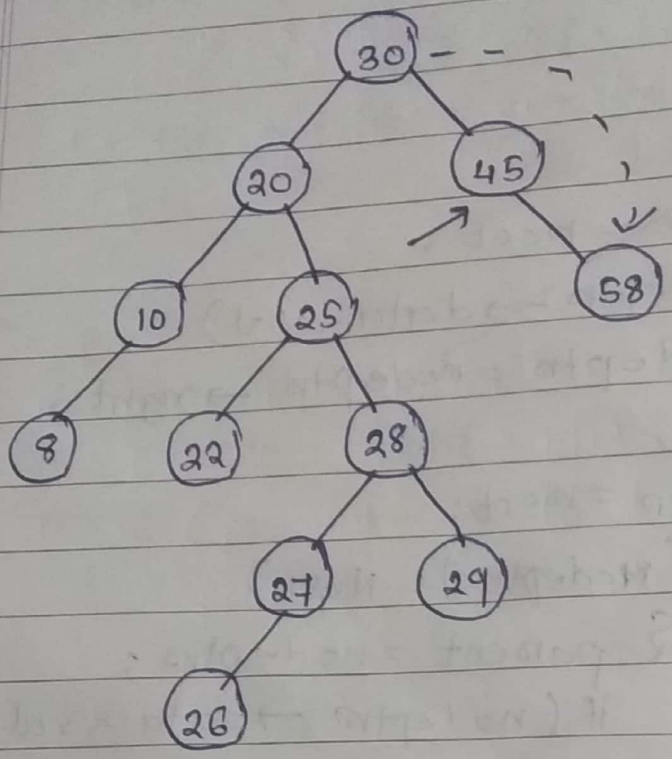



```
struct node * BSTinsert (struct node * root,
                         int val)
{
 struct node *ptr
ptr = (struct node *) malloc (size of (struct node));
ptr → data  = value;
ptr → left  = NULL;
ptr → right = NULL;
```

Program to find max value from BST.

```
if (root == NULL)
  {
   root = ptr;
  }
else
   {
    nodeptr = root;
    if (nodeptr -> data < val)
        nodeptr = nodeptr -> right;
else
  { nodeptr = root;
    While (nodeptr != NULL)
        { parent = nodeptr;
          if (nodeptr -> data < val)
              nodeptr = nodeptr -> right;
          else
              nodeptr = nodeptr -> left;
         if ( val < parent -> data)
              parent -> left = ptr;
         else
              parent -> right = ptr;
        }
      return (root);
  }
```

* Deletion from BST *



1. Tree is empty or not.
2. Seach value
3. check node has one child / two child. (if present)
4. if one child [- - - -]
5. if two child [- - - -]

Non-Recursive Approach.

```
struct node * delete BST( struct node *root,
                          int val)
{
struct node * cur, * parent;
if (root == NULL)
{
```

```
printf ("Tree is empty");
return (root);
}
cur = root;
while (cur->data != val && cur != NULL)
{
 parent = cur;
 if (val < cur->data)
      cur = cur->left;
 else if (val > cur->data)
      cur = cur->right;
}
if (cur == NULL)
 {
  printf ("value is not present");
  return;
 }
if (cur->left == NULL)
  ptr = cur->right;
else if (cur->right = NULL)
      ptr = cur->left;
else
  {
   suc = cur->right;
  psuc = cur;
  while (suc->left != NULL)
       {
        psuc = suc;
        suc = suc->left;
       }
```

psuc → parent of successor.

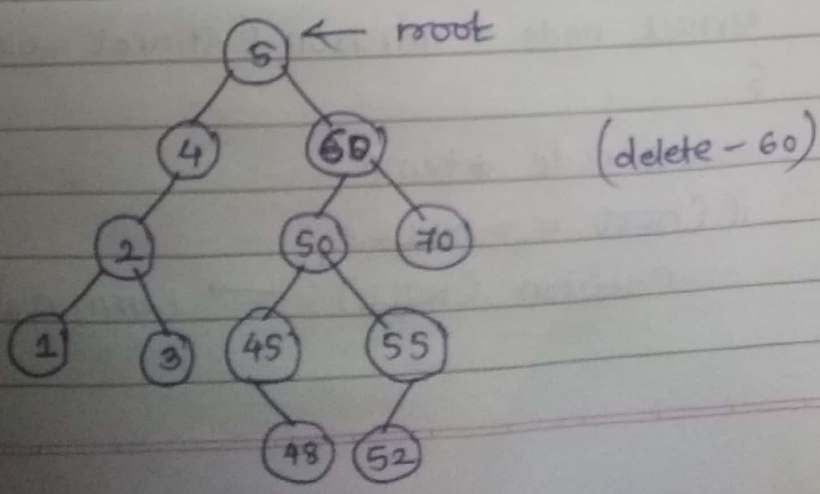
```
else
  {
   suc → left = curr → left;
  psuc → left = suc → right;
   suc → right = curr → right;
  }
  ptr = suc;
}
if (parent → right == curr)
    parent → right = ptr;
else if (parent → left == curr)
        parent → left = ptr;
        return
return (root);
}
```

* Recursive Approach *

```
struct node *delete (struct node *root,
                     int val)
{
 struct node *temp;
 if (root == NULL)
    printf ("No element");
 if (val < root → data)
    root → left = delete (root → left, val);
else if (val > root → data)
     root → right = delete (root → right, val);
```

```
else{
    if (root → left != NULL && root→right!=NULL)
      {
        temp= findmax (root → left);
        root→data = temp→ data;
        root→left = delete (root→left,
                            temp→data);
      }
else else
      {
        temp = root;
        if (root → left == = NULL)
            root = root→right;
        if (root→right = =NULL)
            root = root → left;
        free (temp);
      }
    }
return (root);
}
```

Explanation



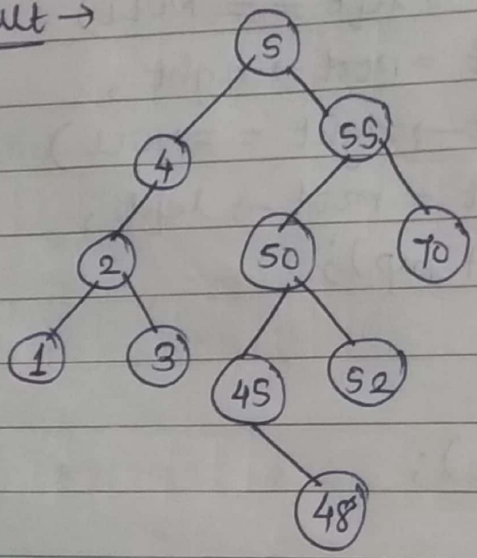
① root = 5 , val = 60
5 → right = delete (60, 60)

② root = 60, val = 60
temp = findmax (60 → left)
temp = 55
55 → left = delete (50, 55)

③ root = 50, val = 55 ⇒ 50 → right = delete (55,55)

④ root = 55, val = 55
root = 52.

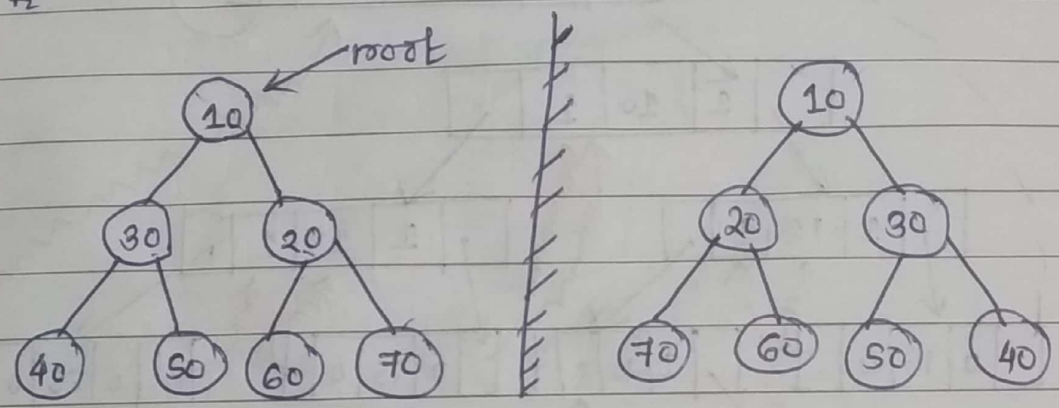
Result →



☆ Recursive code for mirrors Image of tree ☆

```
struct node * mirror (struct node * root)
{
struct node *temp;
if (root == NULL)
    return (NULL); ⟶ termination case
```

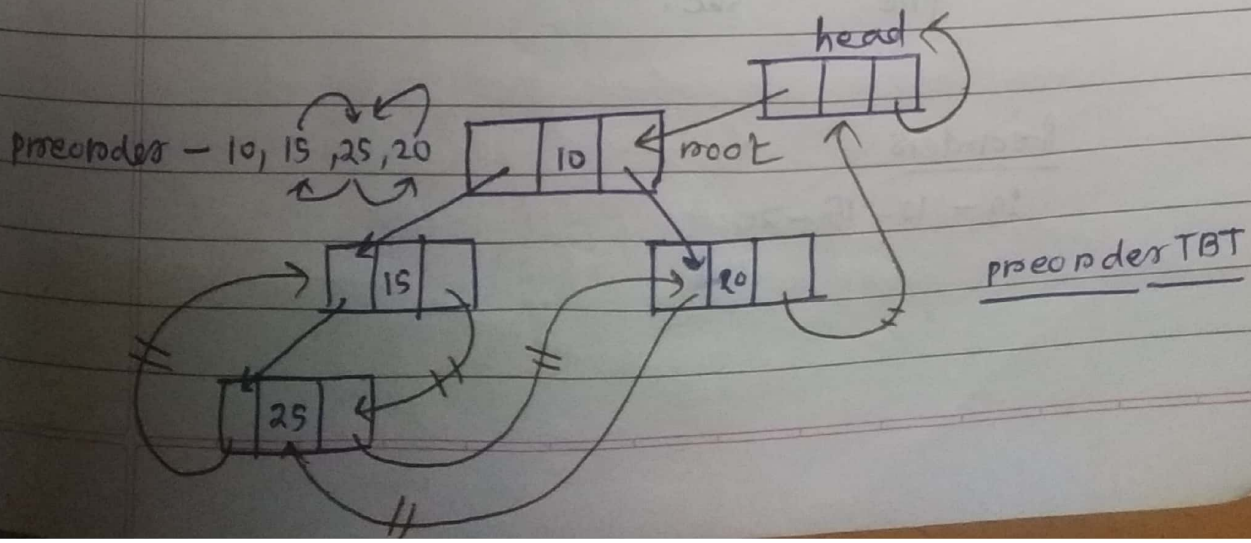
```
else {
        temp = root;
reaching to leaf node  [ mirror (root → left);
                       [ mirror (root → right);
swapping left and right [ temp = root → left;
                        [ root → left = root → right;
                        [ root → right = temp;
        }
    return (root);
}
```

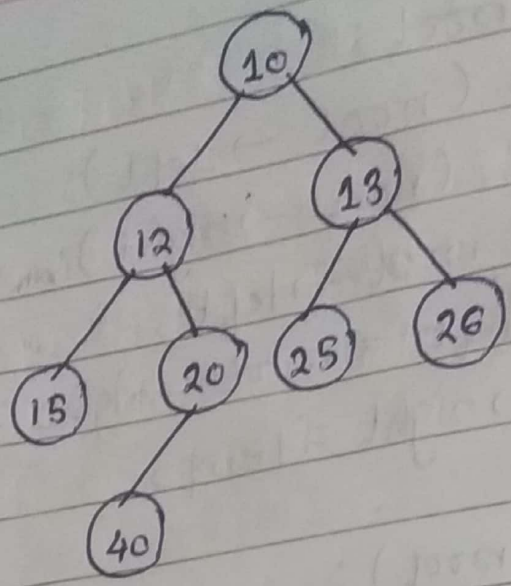


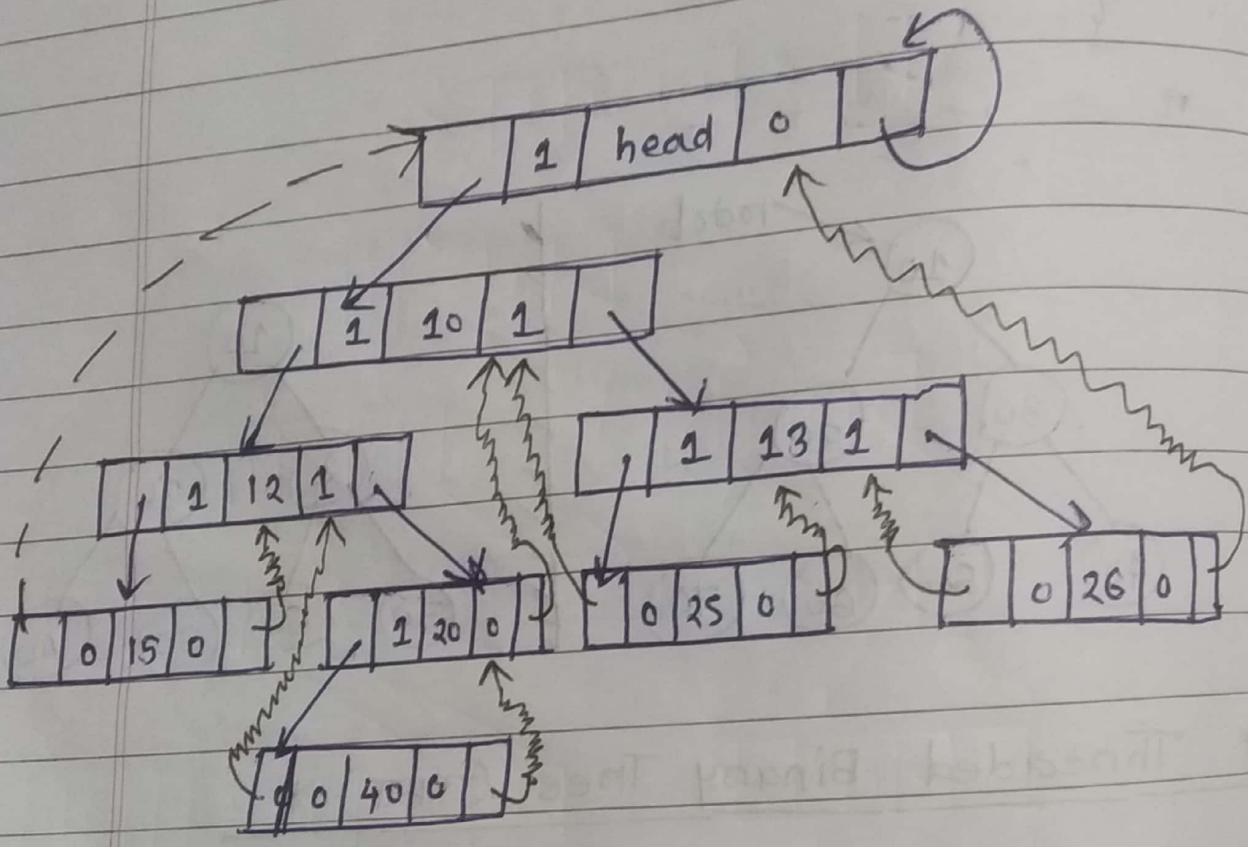
* Threaded Binary Tree (TBT) *

* A tree of 'n' nodes have 'n+1' null links.

* 'n+1' links are untilized links.







Inorder TBT

$-15-12-40-20-10-25-13-26-$

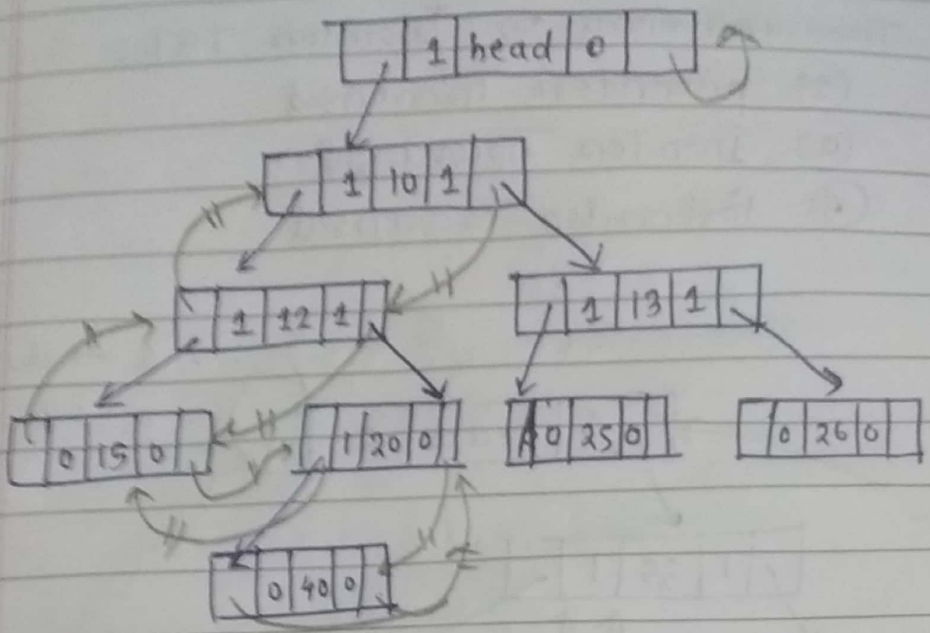
pre suc.

Preorder

10 - 12 - 15 - 20

10 – 12 – 15 – 20 – 40 – 13 – 25 – 26

Preorder TBT



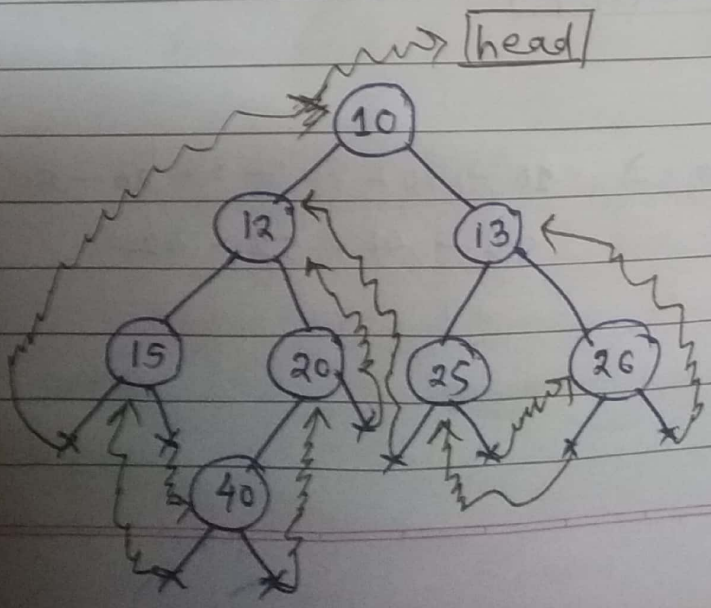
Link/Thread	L bit	Data	R bit	Link/Thread

L bit / R bit ⇒ 1 ⇒ link to child

L bit / R bit ⇒ 0 ⇒ Threading.

Postorder

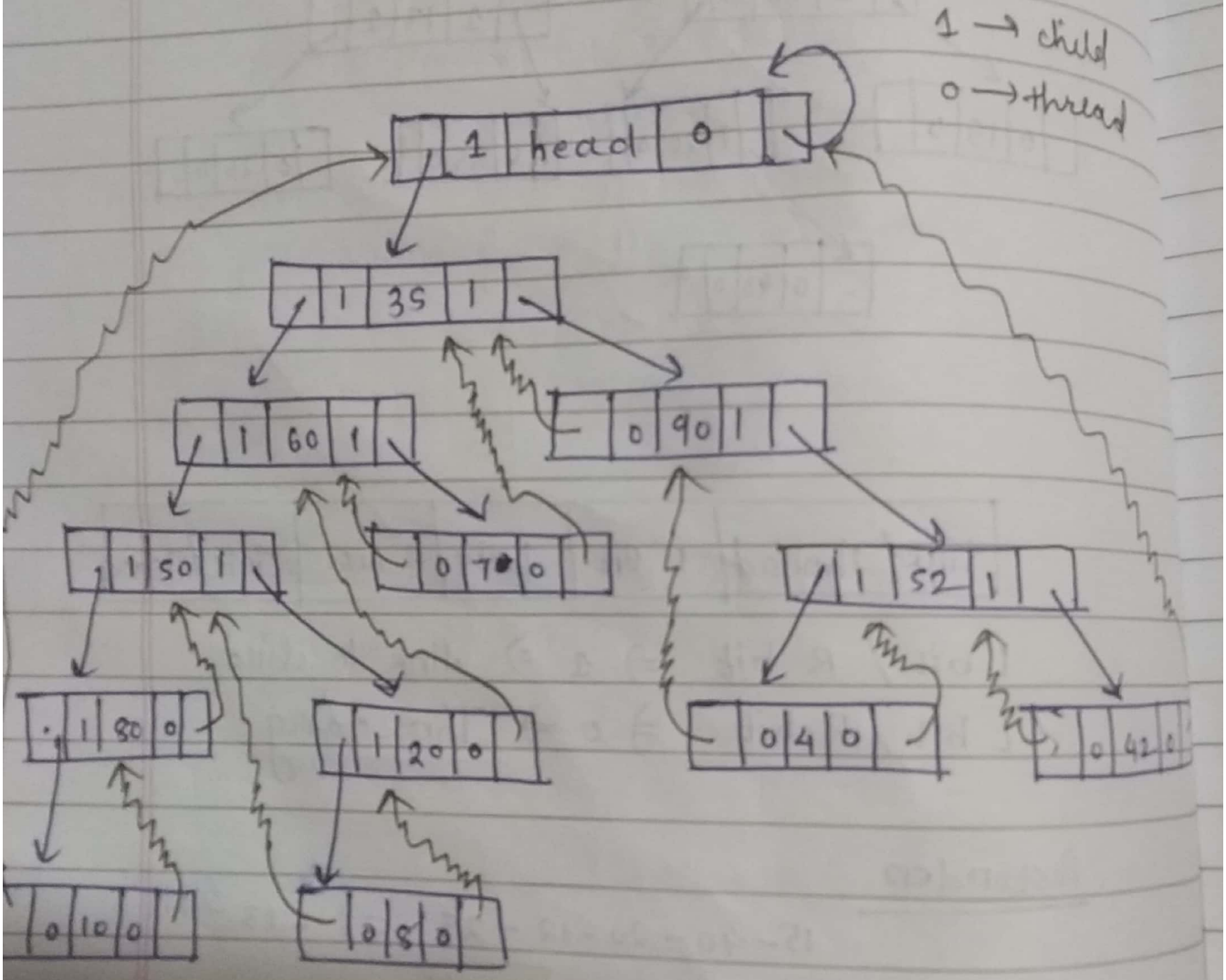
15 – 40 – 20 – 12 – 25 – 26 – 13 – 10



* Inorder TBT *

Tree traversal on Inorder TBT :-

1. preorder Traversal
2. Inorder Traversal
3. Postorder Traversal

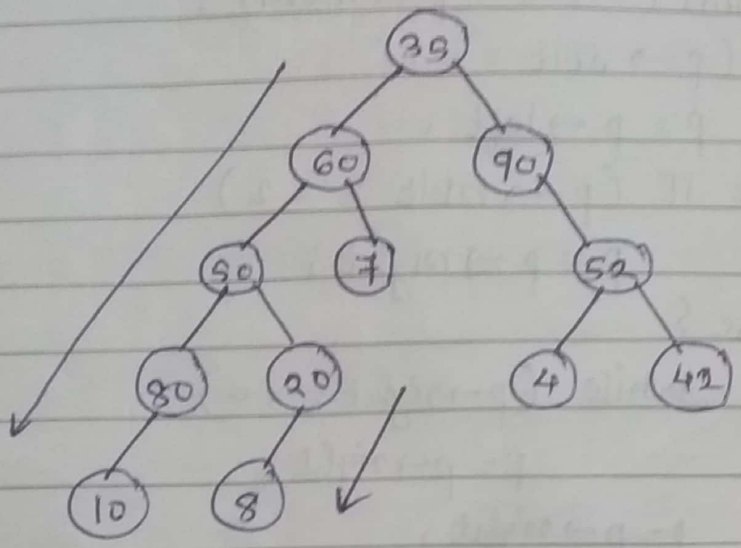


Inorder ⇒ 10 - 80 - 50 - 8 - 20 - 60 - 7 - 35 - 90 - 41 - 52 - 42

(1) Preorder Traversal on Inorder TBT

Parent → Left - Right

35 - 60 - 90 - 80 - 10 - 20 - 8 - 7 - 90 - 52 - 41 - 42



~~Algorithm~~

1) ~~if 1 → child if 0 → thread & bad~~

Algorithm

```
while ( node is not head)
  {
    display node
    if ( left child parent)
        goto left node
    else if ( right child prevent)
          goto right
    else ( till not getting parent child)
        goto right
  }
```

```
void preorder (struct node *head)
{ struct node *p;
  p=head→left;
  while (p!=head)
  { printf("%d", p→data);
    if (p→lbit==1)
        p=p→left;
    else if (p→rbit==1)
        p=p→right;
    else {
        while (p→right==0)
            p=p→right;
        p=p→right;
    }
```

② Inorder Traversal on Inorder TBT

Left - Parent - Right

10 - 80 - 50 - 8 - 20 - 60 - 7 - 35 - 90 - 4 - 52 - 42

~~① if child=1~~

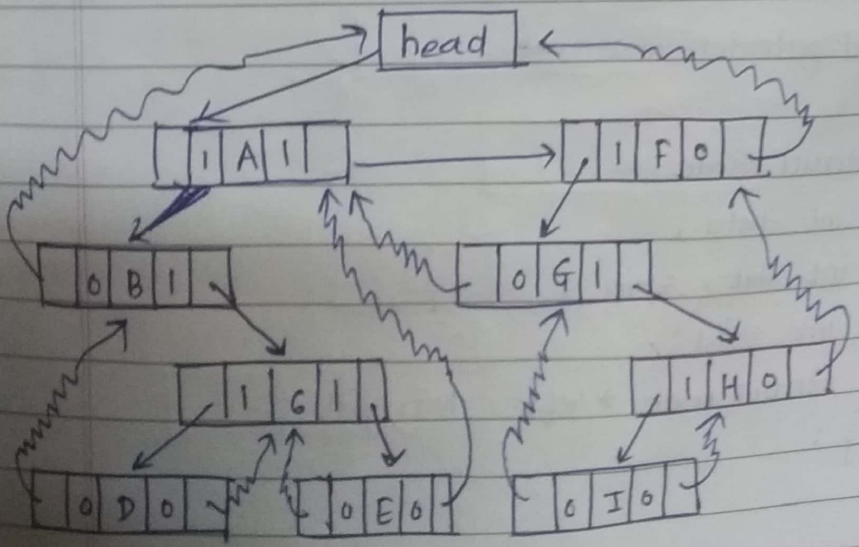
~~if many people are present in the queue then you go to shop~~

```
void inorder (struct node *head)
{
  struct node *T;
  T=head→left;
```

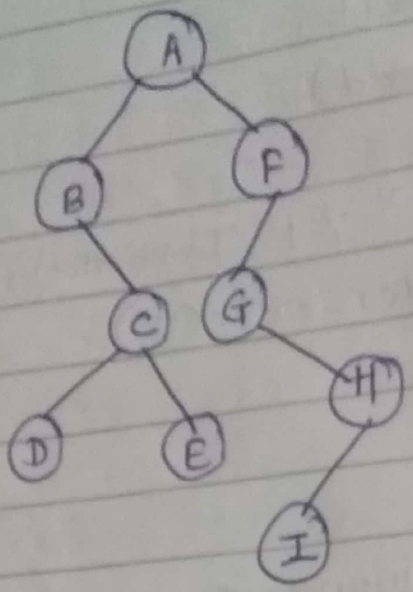
```
while (T->lbit == 1)
    T = T->left;
while (T != head)
    {
    printf ("%d", T->data);
    T= inorder_succ(T);
    }
}
```

```
struct node * inorder_succ (struct node *T)
    {
    if (T->rbit == 0)
        return (T->right);
    T= T->right;
    while (T->lbit == 1)
        T= T->left;
    return (T);
```

* Postorder Traversal on Inorder TBT *

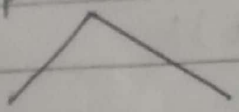


L - R - P



D - E - C - B - I - H - G - F - A.

① Find out 1st node to display.
└ postorder succ.



child = 1 → right
child = 0 → left.

deft turned parent
- parent have right child.
- parent don't have right child.

Right turned parent
- child = 1

~~void postorder (struct~~

```
struct node
{ int data;
  int lbit, rbit;
  int child;
  struct node *left, * right;
};
```

```
void post_traversal (struct node *head)
{ struct node *T = head -> left;
  while (T->lbit == 1 || T->rbit == 1)
    {
      if (T->lbit == 1)
        T = T->left;
      else
        T = T->right;
    }
  while (T != head)
    { printf ("%d", T->data);
      T = post_succ (T);
    }
}

struct node * post_succ (struct Node *T)
{
  if (T->child == 1)
    {
      while (T->lbit == 1)
        T = T->left;
      return (T->left);
    }
  else
  { if (T->rbit == 1)
      T = T->right;
    T = T->right;
    if (T->rbit == 0)
      return (T);
  else { T = T->right;
```

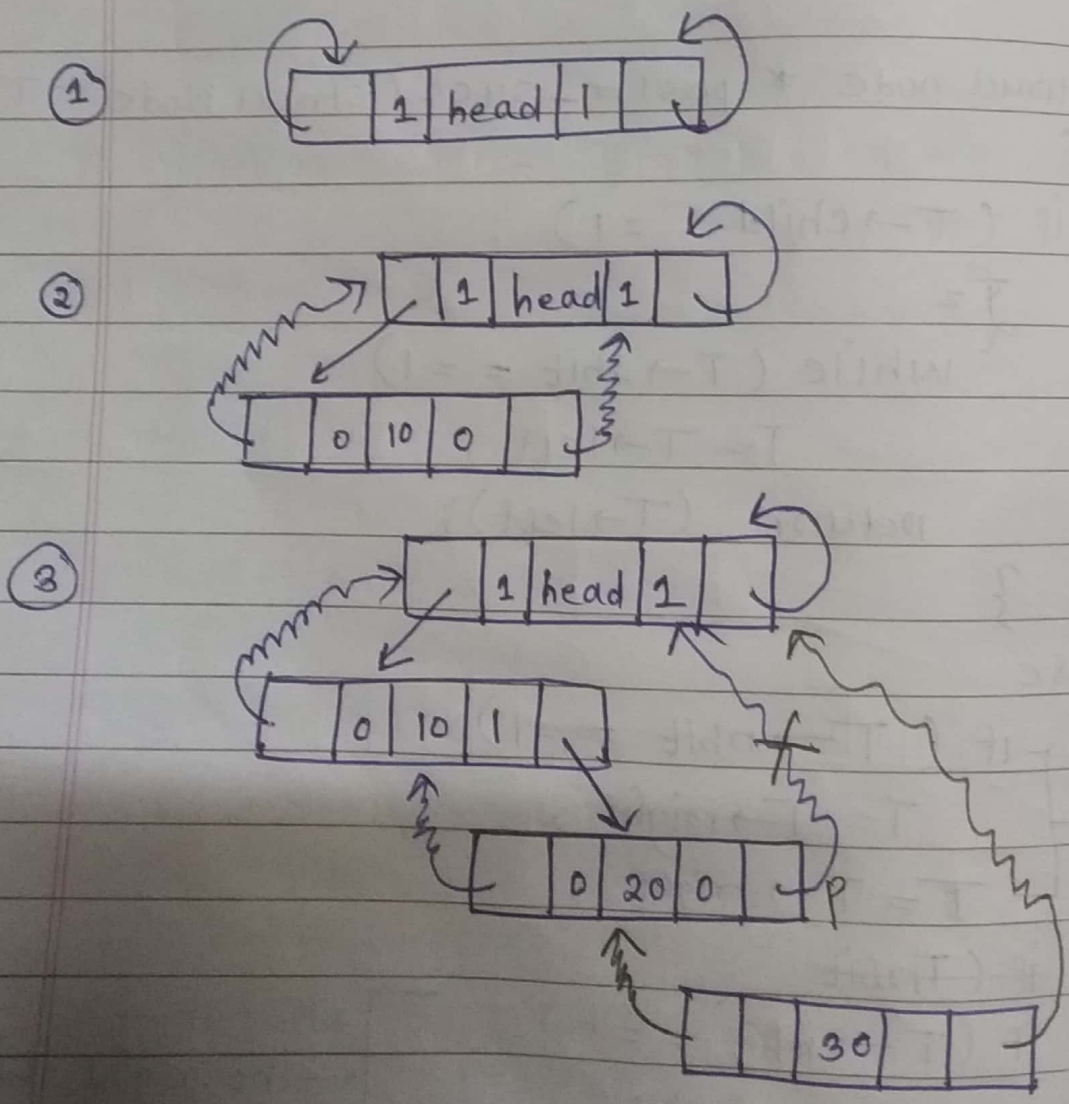
Reach to parent

no right child.

```
while ( T→lbit == 1 || T→rbit == 1)
    {
      if ( T→lbit == 1)
          T = T→left;
      else
          T = T→right;
    }
return (T);
}}
```

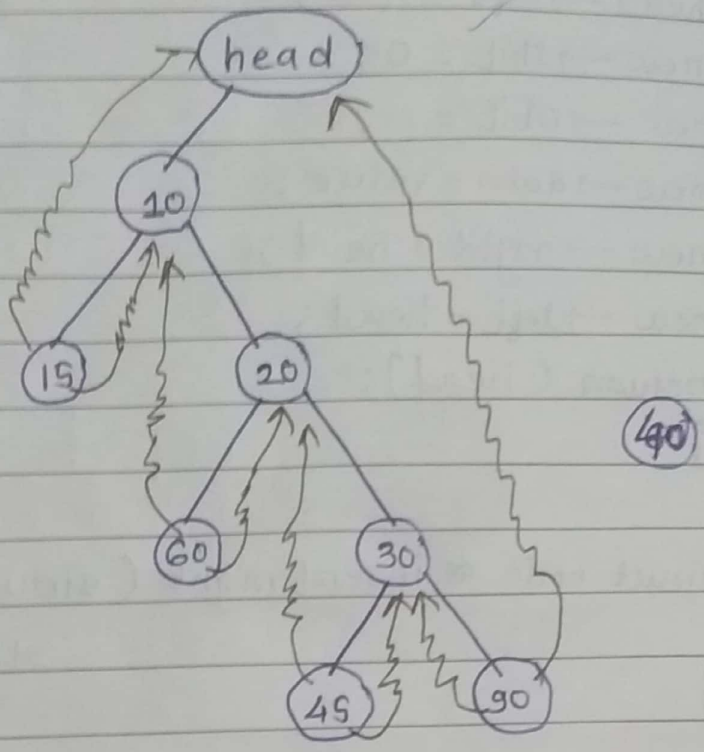
Return 1st node Right subtree.

* Creating Inorder TBT and Inserting Node in Inorder TBT *

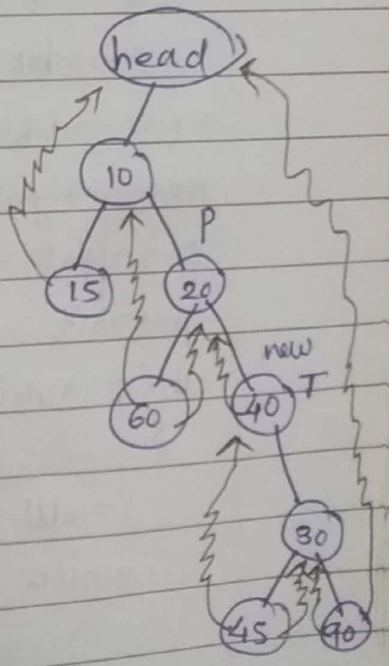


```
struct Node
{ int data;
  int lbit, rbit;
struct node * left, * right;
}
```

4

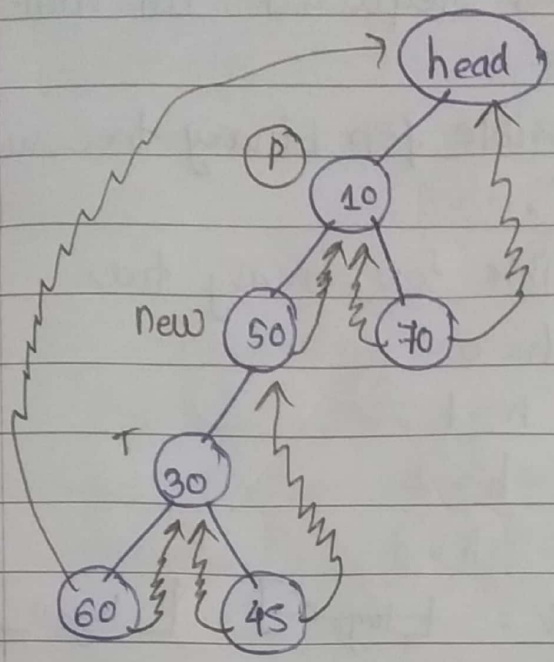



```
main ()
{
struct node *head;
head → lbit = 1;
head → rbit = 1;
head → left = head;
head → right = head;
head = create(head);
```



```
struct node * create ( struct node * head)
{
int val;
struct node *new;
new = () malloc ( ) ;
head → left = new;
new → lbit = 0;
new → rbit = 0;
new → data = value;
new → right = head;
new → left = head;
return (head);
}

struct node * insert right ( struct node *p,
                             struct node *ne
{
new → lbit = 0;
new → rbit = p → rbit;
new → left = p;
new → right = p → right;
p → right = new;
p → rbit = 1;
if (p → rbit = = 1)
{
 T = new → right
 while (T → lbit = = 1)
        T = T → left;
 T → left = new;
}
```

```
p->rbit = 1;
}
```




```
void insertleft (struct node *p, struct node *new)
{
struct node *T;
new->rbit = 0;
new->lbit = p->lbit;
 new->right = p;
new->left = p->left;
p->left = new;

if (p->lbit == =1)
{ T= new->left;
   while (T->rbit == =1)
          T=T->right;
          T->right = new;
}}
```

→ Recursive approach is always best for traversal
→ Time complexity for non-recursive approach over threaded binary tree is $O(n \cdot \log n)$
→ more memory is required in non-recursive than recursive
→ Max. height possible for binary tree with n nodes is 'n-1'.
→ Min height possible for binary tree

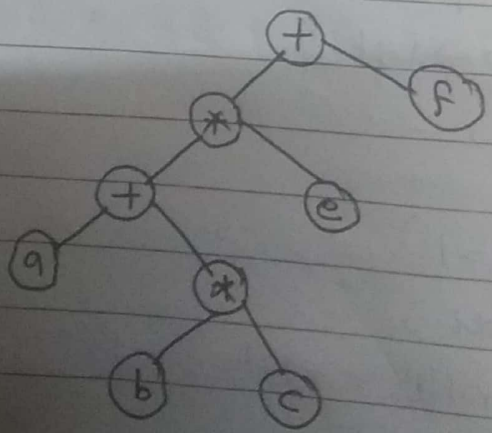
$n=1$	$h=0$
$n=2,3$	$h=1$
$n=4,5,6,7$	$h=2$
$n=8,9,10,\ldots 15$	$h=3$

min height = ~~$\lfloor \log_2 n \rfloor$~~ $\lfloor \log_2 n \rfloor$

* Application of Trees *

(1) Expression Tree :-

$(a+b*c)*e+f$



PLR preorder - + * + a * b c e f

LPR inorder - a + b * c * e + f

LRP postorder - a b c * + e * f +

(a + b * c) * e + f

prefix = + * + a * b c e f (preorder)

postfix = a b c * + e * f + (postorder)

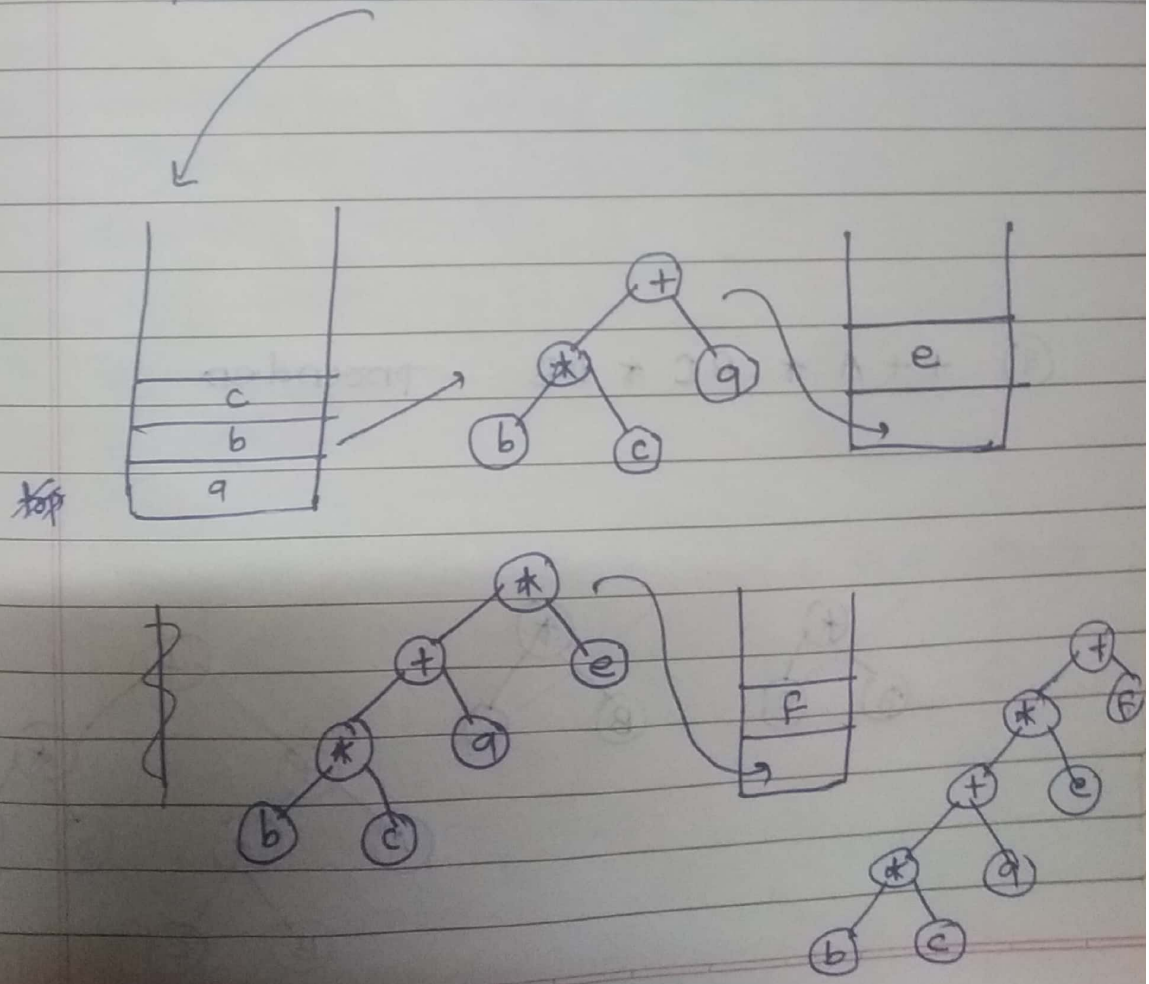
* if preorder & postorder is given, find expression tree.

→ preorder - + * + a * b c e f PLR

postorder - a b c * + e * f + LRP

preorder - + * + a * b c e f

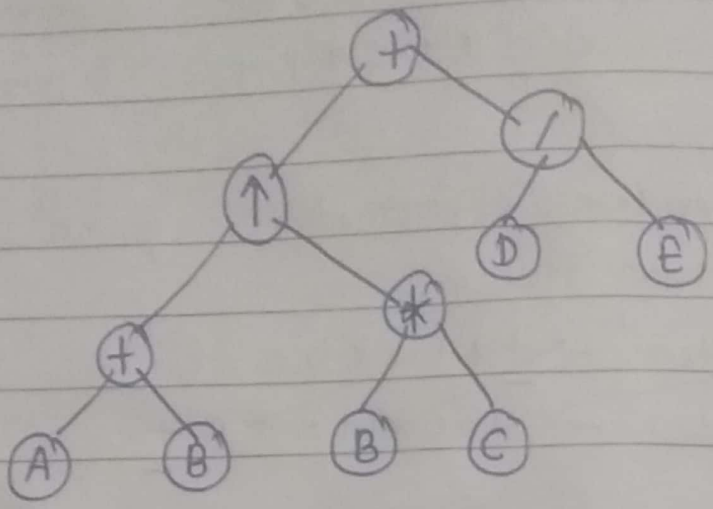
postorder - a b c * + e * f +



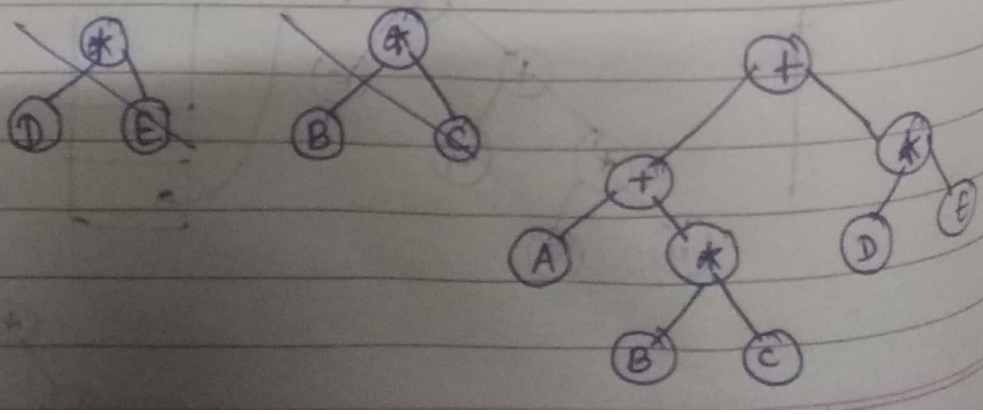
↑ ⟶ highest preference

① (A+B)↑(B*C) + D/E

Inorder

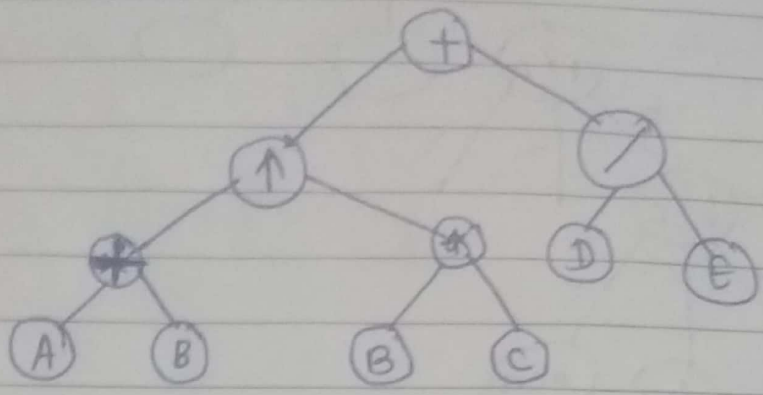


② ++A * BC * DE preorder

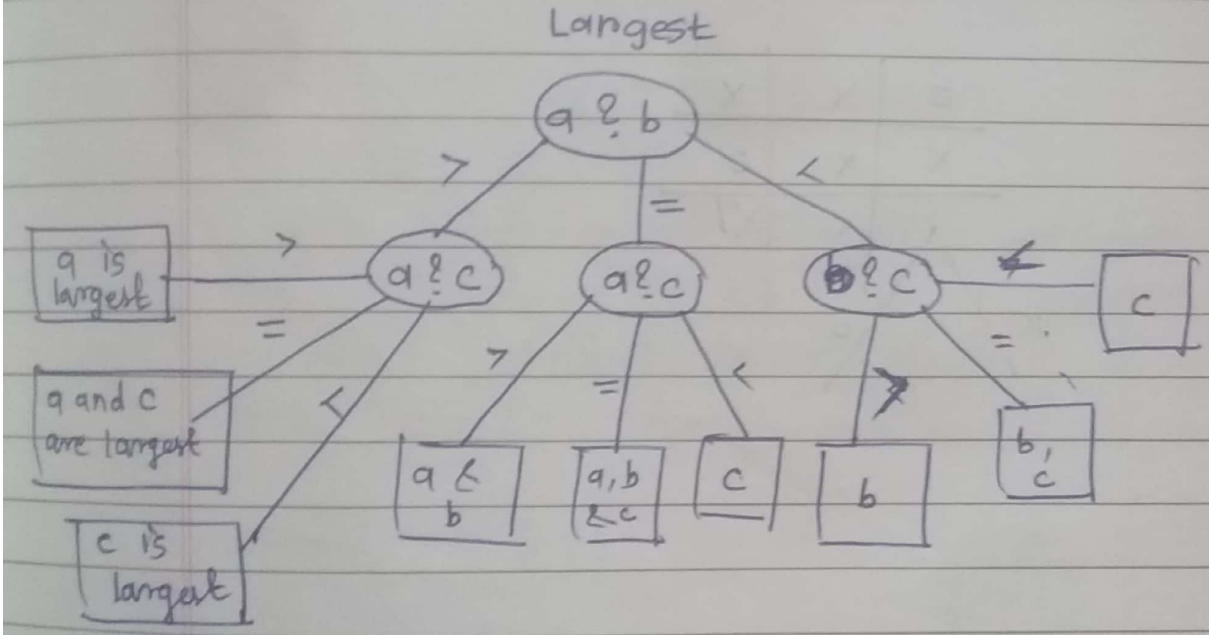


(8) AB + BC * ↑ DE / +

postorder

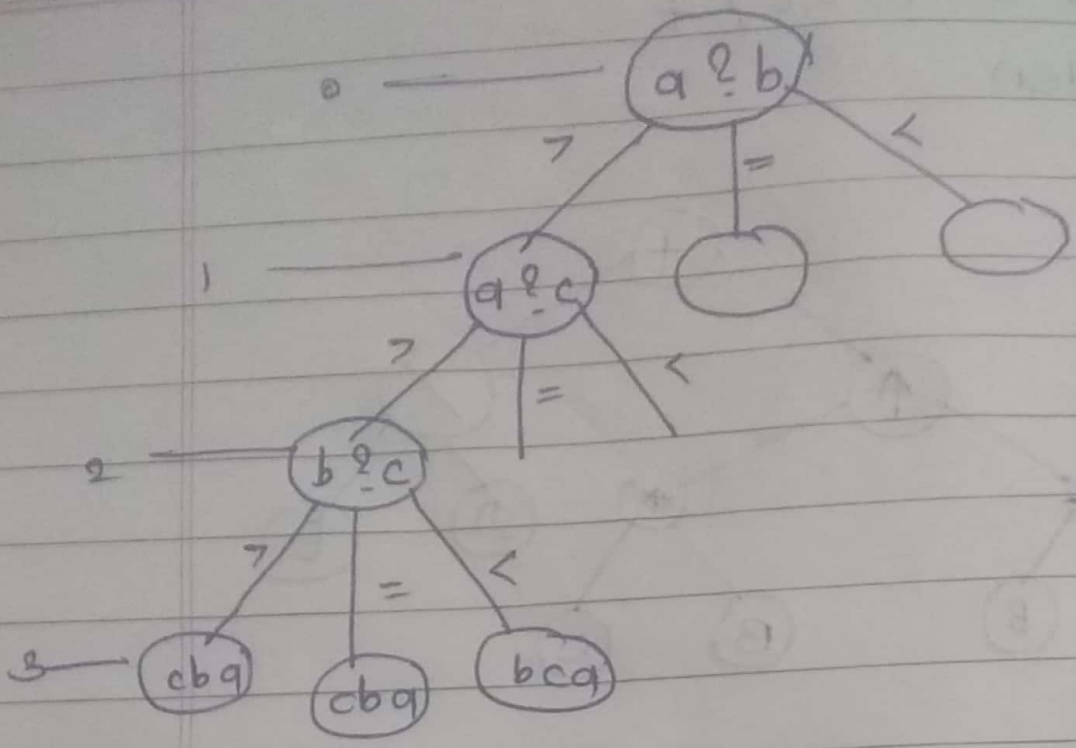


* Decision Tree *



Decision tree to sort a, b, c in ascending order.

a b c
a c b
b c a
b a c
c a b
c b a



* Game Tree *