Traversal of a binary tree: (1)

There are three fundamental ways of traversal of a binary tree as follows:

a) Preorder (R, T_L, T_R)
b) Inorder (T_L, R, T_R)
c) Postorder (T_L, T_R, R)

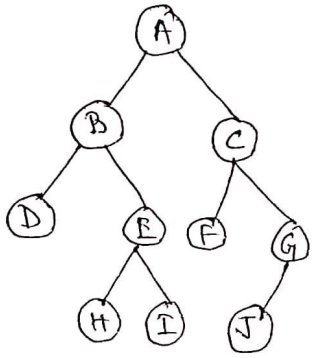
where R is the root node of the binary tree and T_L and T_R are the left and right subtree of the corresponding binary tree.

Let us consider the recursive implementation of the traversal algorithms, we assume that the binary tree already exists and is represented using linked representation where the 'pointer root' is the pointer to root node of the tree. We will take an integer value as the information in each node of the tree and visiting a node means printing the value. The recursive functions are given below:

```
void preorder (struct node *ptr)
{ if (ptr == null) /* base case */
      return;
   printf ("%d", ptr -> info);
   preorder (ptr -> lchild);
   preorder (ptr -> rchild);
}.
void inorder (struct node *ptr)
{  if (ptr == null) /* base case */
      return;
   inorder (ptr -> lchild);
   printf ("%d", ptr -> info);
   inorder (ptr -> rchild);
}
```

```
void postorder(struct node *ptr)
{
  if (ptr == null) /* base case */
    return;
  postorder(ptr->lchild);
  postorder(ptr->rchild);
  printf("%d", ptr->info);
}
```

These three cases can be illustrated with the help of the tree given below;

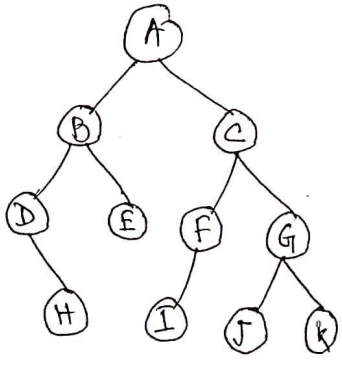


Preorder: A B D E H I C F G J

Inorder: D B H E I A F C J G

Postorder: D H I E B F J G C A

Let us take another example with the following tree and apply the traversal algorithm;



Preorder: A B D H E C F I G J K

Inorder: D H B E A I F C J G K

Postorder: H D E B I F J K G C A

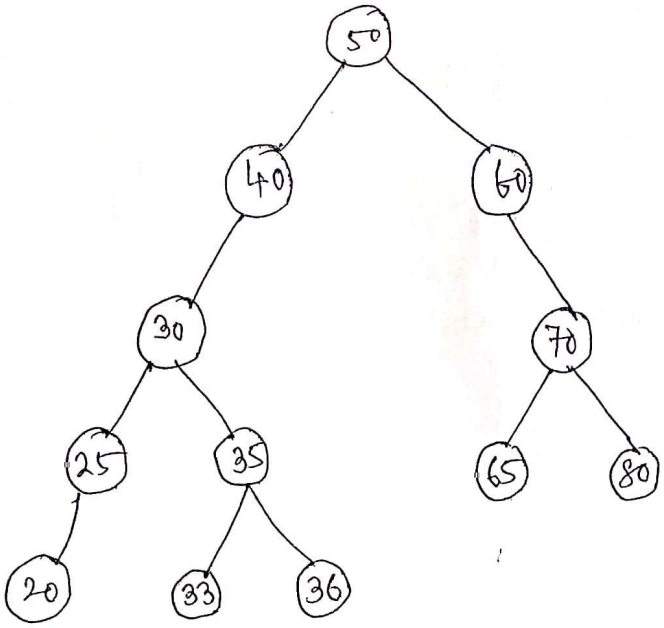
Recursive (Iterative) Traversal of Binary Tree:

If we want to write non recursive function for traversal of binary tree, we have to use an explicit stack. We assume that we have a stacks, implemented using array and the stack will store the address of the nodes. The function for implementing stack is well known push-stack() and *pop-stack(), which will increment and decrement the top of the stack pointer and also check the stack overflow and underflow conditions.

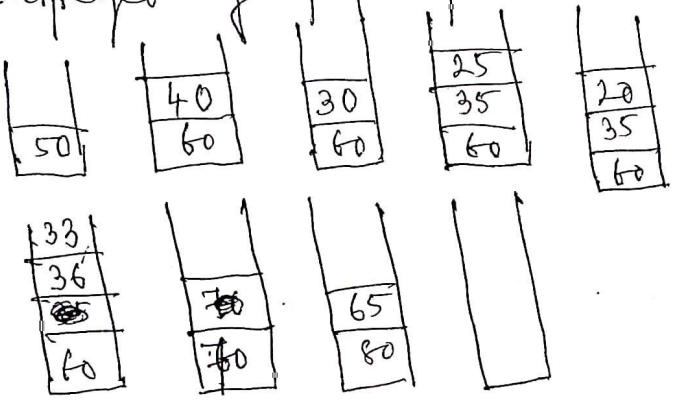
The function for non recursive preorder traversal is:

```
void nrec-pre(struct node root)
{
    struct node *ptr = root;
    if (ptr == null)
    {
        printf("Tree is empty\n");
        return;
    }
    push-stack(ptr);
    while (!stack-empty())
    {
        ptr = pop-stack();
        printf("%d", ptr->info);
        if (ptr->rchild != null)
            push-stack(ptr->rchild);
        if (ptr->lchild != null)
            push-stack(ptr->lchild);
    }
    printf("\n");
}
```

Let us consider the below tree for illustration:



Let us see the stack ~~content~~ content and the element displayed by pop after each iteration:



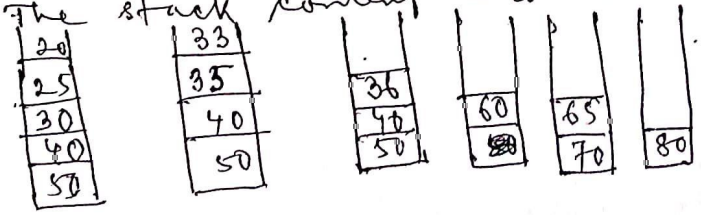
50, 40, 30, 25, 20, 35, 33, 36, 60, 70, 65, 80.

The function for non-recursive inorder traversal is:

```
void nrec_in (struct node *root)
{
  struct node *ptr = root;
  if (ptr == null)
  {
    printf ("Tree is empty \n");
    return;
  }
```

```
ile(1)
{
  while (ptr->lchild != null)
    {
      push_stack(ptr);
      ptr = ptr->lchild;
    }
  while (ptr->rchild == null)
    {
      printf("%d", ptr->info);
      if(stack.empty())
        return;
      ptr = pop_stack();
    }
  printf("%d", ptr->info);
  ptr = ptr->rchild;
}
printf("\n");
}
```

The stack content and the display is shown below:



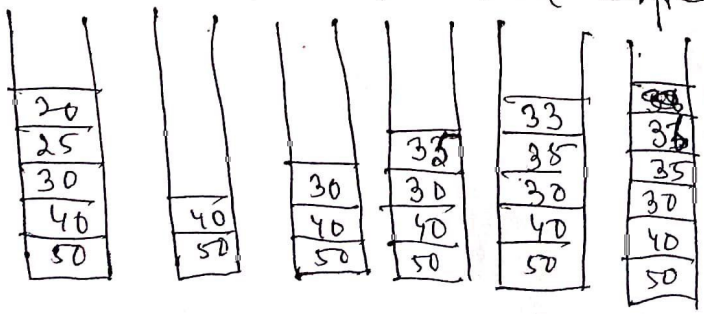
20, 25, 30, 33, 35, 36,
40, 50, 60, 65, 70, 80

Non Recursive version of postorder traversal can be given below:

```
void nrec_post(struct node *root)
{
  struct node *q, *ptr = root;
  if(ptr == null)
  { printf("Tree is empty\n");
    return;
  }
```

```
q = root;
while(1)
{
  while (ptr -> lchild != null)
  {
    push-stack(ptr);
    ptr = ptr -> lchild;
  }
  while (ptr -> rchild == null) || (ptr -> rchild == q)
  {
    printf("%d", ptr -> info);
    q = ptr;
    if (stack-empty())
        return;
    ptr = pop-stack();
  }
  push-stack(ptr);
  ptr = ptr -> rchild;
}
printf("\n");
}
```

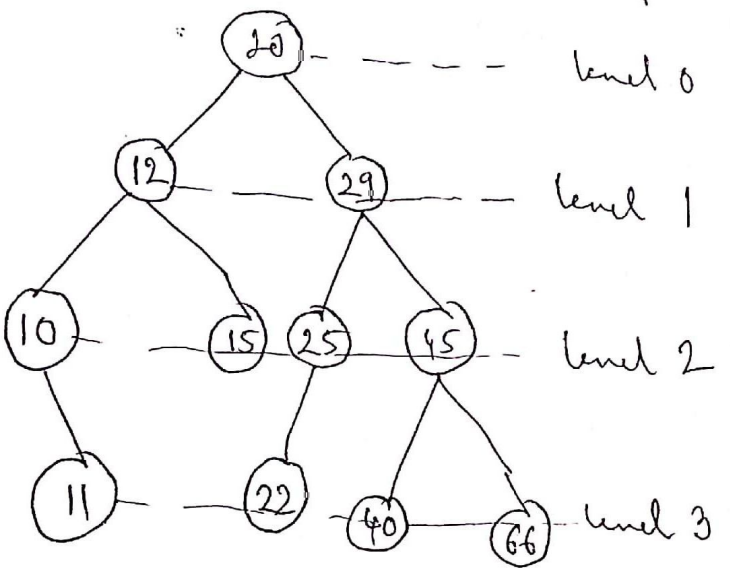
The stack content and the display of elements is shown below;



20, 25, 33, 36, 35, 30
40, 65, 80, 70, 60, 50

Level Order Traversal

In level order traversal, the nodes are visited from top to bottom and from left to right. Initially we visit the root node at level 0, then we visit all the nodes of level 1, then we visit all the nodes of level 2 and so on till last level.



Let us take the tree in the adjoining diagram and visit level order traversal. The nodes that are visited in the order

20, 12, 29, 10, 15, 25, 45, 11, 22, 40, 60.

This traversal can be implemented using a queue that will store the address of the nodes.

1. Insert root node in the queue.
2. Delete the first node from the front of the queue and visit it.
3. Insert all the successors of the node just visited in the back (rear) of the queue.
4. Continue step 2 and 3 till the queue is empty.

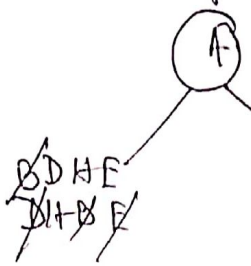
Creation of the binary tree from inorder and preorder traversal

Let us construct a binary tree from given preorder and inorder traversals:

Preorder; A B D H E C F I G J K

Inorder; D H B E A I F C J G K.

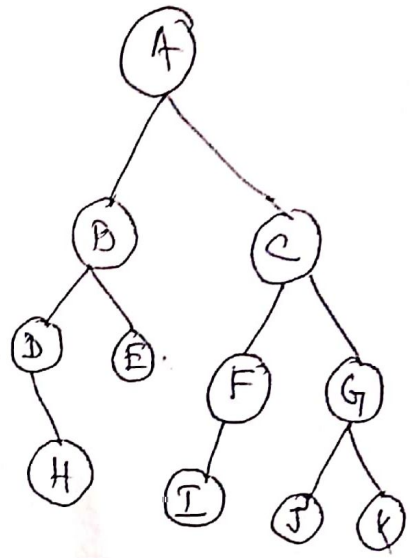
(1) A is the root of the binary tree.



(2) B and C are the left and right child of A. (3) D and E are left and right child of B, (4) H is the right child of D.

(5) I F are in the left subtree of C and J G K are in the right subtree of C. F is the left child of C and hence G is the right child of C

(6) J and K are the child of G.



Creation of binary tree from inorder and postorder traversals:

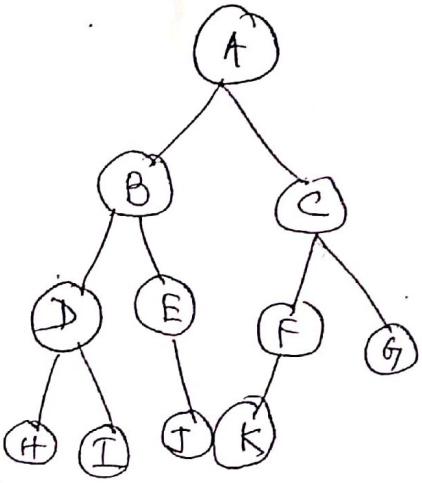
Postorder; H I D J E B K F G C A

Inorder; H D I B E J A K F C G

(1) A is root, H D I B E J are in the left subtree and K F C G is the right subtree. (2) B is the left and C is the right child of A (3) F is the left child and G is the right child of C.

(4) K is the left child of F. (5) E & J is the right subtree of B and H D I is the left subtree of B. (6) E is the right child of B. (7) J is the right child of E.

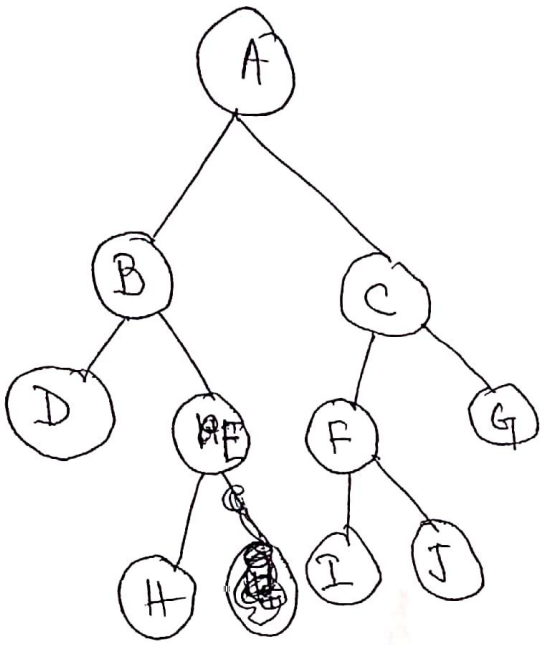
(8) D is the left child of B and H and I are the left and right child of D.

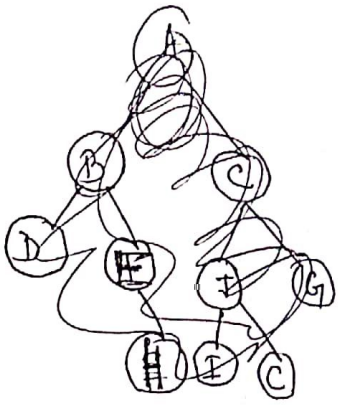


mple1. Form the binary tree from the following inorder and preorder
raversal:

inorder: D B H E A I F J C G

preorder: A B D E H C F I J G

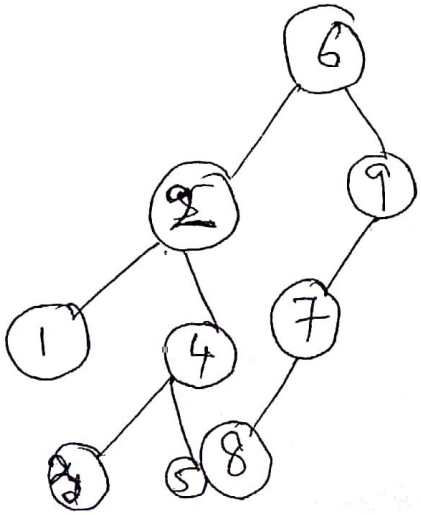




Example2. From the binary tree from the following inorder and postorder traversal:

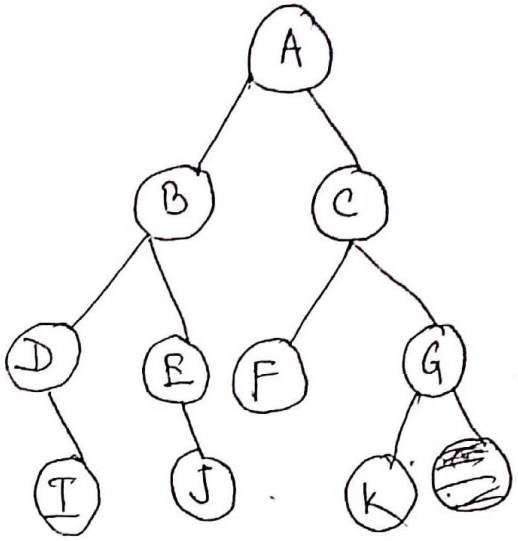
Inorder: 1, 2, 3, 4, 5, 6, 7, 8, 9

Postorder: 1, 3, 5, 4, 2, 8, 7, 9, 6.



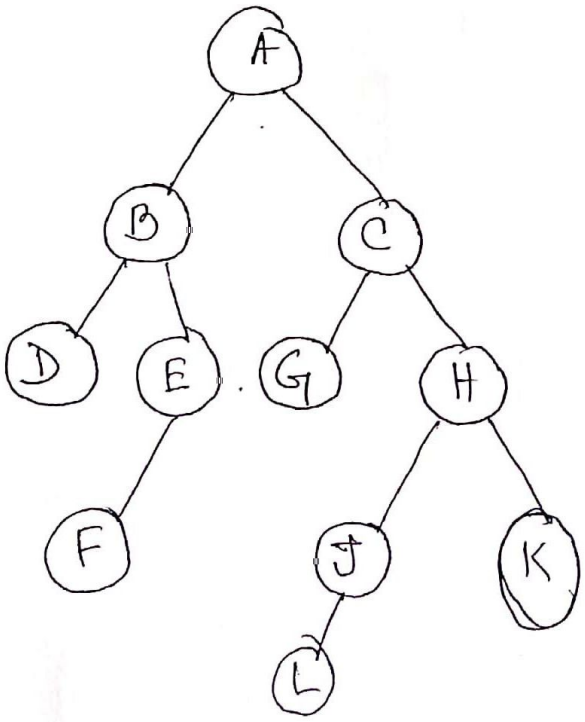
Example 3. Given the preorder and inorder traversal of a binary tree draw the actual tree representation.

Preorder; A B D I E J C F G K
Inorder; D I B E J A F C K G.



Example 4. Given the preorder and inorder sequences draw the resultant binary tree:

Preorder; A B D E F C G H J L K
Inorder; D B F E A G C L J H K



inary Search Tree:

The binary search tree is a binary tree that enables one to search for and find an element with an average running time $f(n) = O(\log_2 n)$. It also enables one to easily insert and delete elements. This structure contrast with the following structures:

a) Sorted linear array, where searching takes the same time but insertion and deletion is expensive.

b) Linked list, where insertion and deletion is easy, but the linear search takes a running time $f(n) = O(n)$.

Although, each node in a binary search tree may contain an entire record of data, the definition of the binary tree depends on a given field whose value are distinct and may be ordered.

Suppose T is a binary tree. Then T is called a binary search tree if each node N of T has the following property;

The value at N is greater than every value in the left subtree of N and is less than every value in the right subtree of N. It is not difficult to see that this property guarantees that the inorder traversal of T will yield a sorted listing of the elements of T.

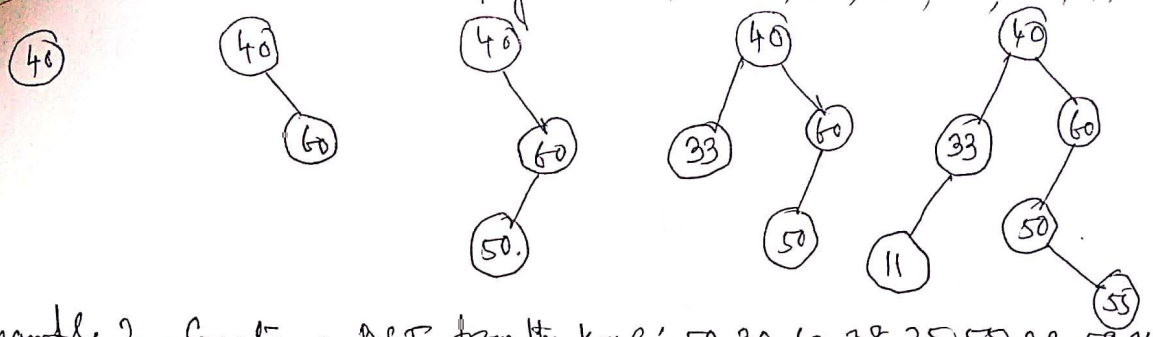
Searching and Inserting an 'item in BST

a) Compare 'item' with the root node N of the tree
   i) if item < N, proceed to the left child of N.
   ii) if item > N, proceed to the right child of N.

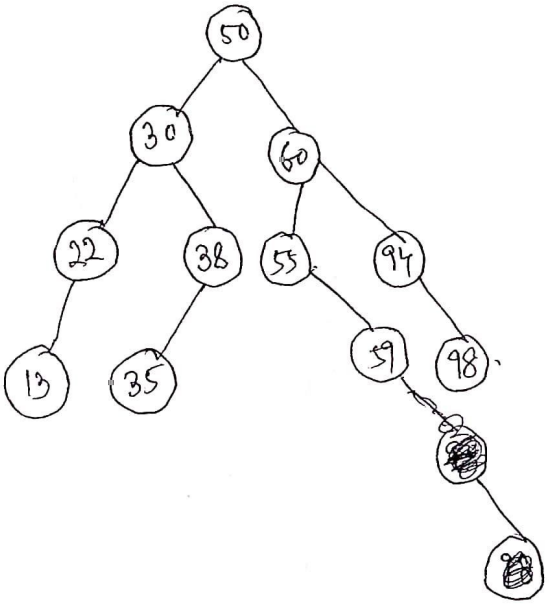
b) Repeat step (a) until one of the following cases occurs;
   i) We meet a node N, such that 'item = N, is the case for successful search
   ii) We meet an empty subtree, which means search is unsuccessful and we insert the 'item in the (D) in place of the empty subtree.

```
struct node *search_rec(struct node *ptr, int skey)
{
  while (ptr != null)
  {
    if (skey < ptr->info)
        ptr = ptr->lchild;
    else if (skey > ptr->info)
        ptr = ptr->rchild;
    else
        return ptr;
  }
}
struct node *insert_rec(struct node *root, int ikey)
{
  struct node *tmp, *par, *ptr;
  ptr = root;
  par = null;
  while (ptr != null)
  {
    par = ptr;
    if (ikey < ptr->info)
        ptr = ptr->lchild
    else if (ikey > ptr->info)
        ptr = ptr->rchild
    else { printf("duplicate key"); return root; }
  }
  tmp = (struct node *) malloc (sizeof (struct node));
  tmp->info = ikey; tmp->lchild = tmp->rchild = null;
  if (par == null)
      root = tmp;
  else if (ikey < par->info) par->lchild = tmp;
      else par->rchild = tmp;
  return root;
```

Example 1. Insert into an empty BST : 40, 60, 50, 33, 55, 11.



Example 2. Create a BST from the keys : 50, 30, 60, 38, 35, 55, 22, 59, 94, 13, 98



Deletion from a BST :

There could be three possible cases :

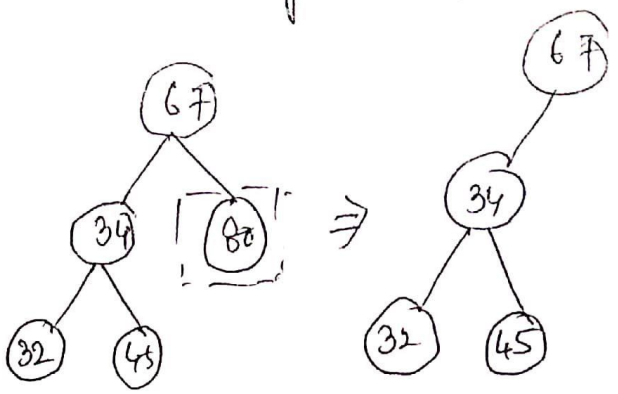
a) Deletion of a leaf node, the pointer from the parent is set to null.

b) Deletion of a node that has one successor, ~~delete the node, and~~ Exchange the location with its ~~parent~~ child and then delete the node using (a).

c) Deletion of a node that has both left and right child. Find the inorder successor of the node, Replace the node to be deleted with its inorder successor.

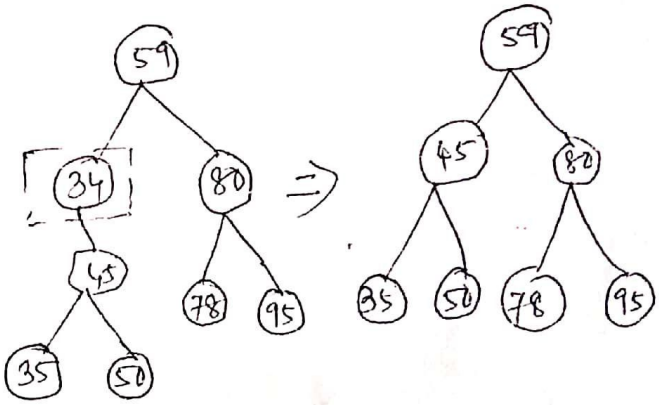
The algorithm for case (a)

```
if (par == Null)
    root = null;
else if (ptr == par->lchild)
    par->lchild = null
else par->rchild = null;
free (ptr);
return root;
```

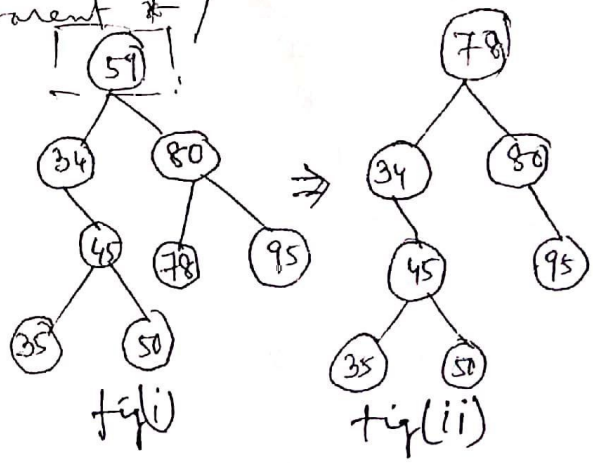


```
if (ptr->lchild != null)
    child = ptr->lchild;
else child = ptr->rchild;
```

Now follow algorithm (a)



```
struct node *succ, *parsucc;
/* find inorder successor and its parent */
parsucc = ptr;
succ = ptr->rchild;
while (succ->lchild != null)
{   parsucc = succ;
    succ = succ->lchild;
}
ptr->info = succ->info;
if (succ->lchild == null) && (succ->rchild == null)
    root = case_a (root, parsucc, succ)
else root = case_b (root, parsucc, succ)
return root;
```



fig(i) fig(ii)

Inorder. of fig(i) Tr R Tr

34, 35, 45, 50, 59, 78, 80, 90

↓ Successor.

Inorder successor will be the root after deletion