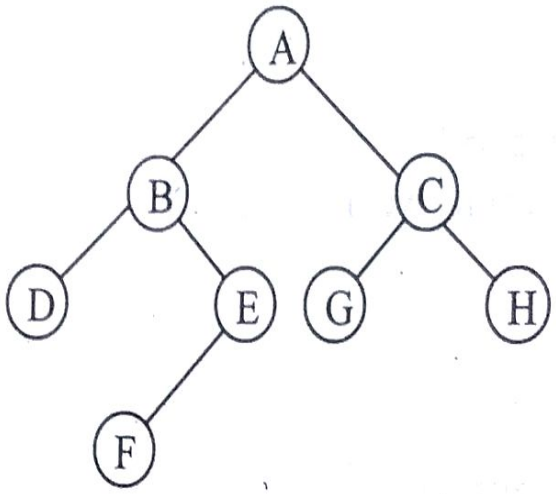
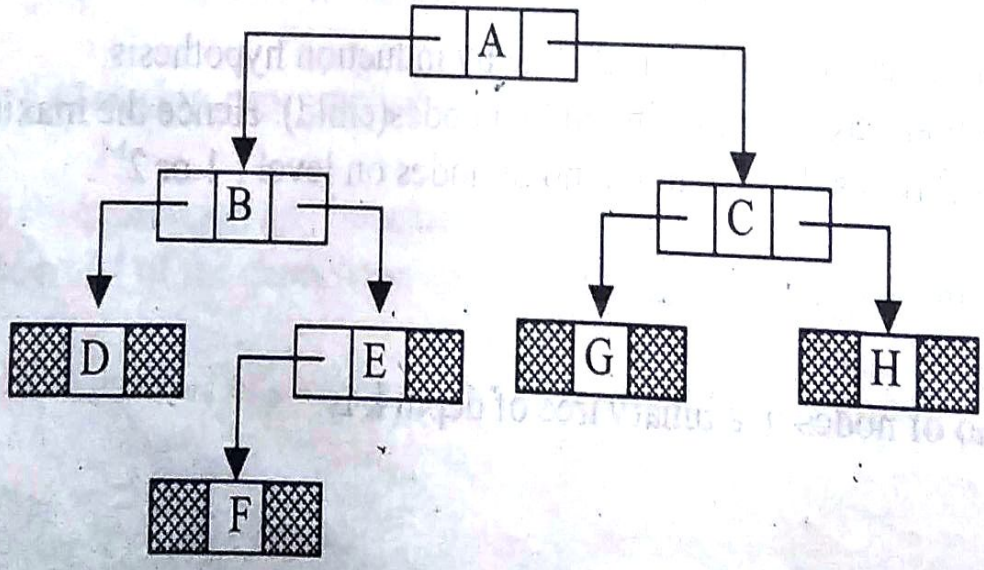
Linked Representation

As in the linked list we take the structure for tree. In which we take three members. First member for data (this can be whole record), second member for left child and third member is for right child. Second and third members are structure pointers which point to the same structure as for tree node.





First we should declare structure for tree node.

```
struct node {
        char data;
        node * lchild;
        node  *rchild;
      } ;
```

Here first member data is for information field of node, second member is for left child of node which points to the structure itself, it contains the address of left child. If node has no left child then it should be NULL.Third member is for right child of node which also points to the structure itself, it contains the address of right child. If node has no right child it should be NULL.

Traversing in Binary Tree

In tree creation we take three parameters node, left child and right child, so traversing of binary tree means traversing of node, left subtree and right subtree. If root is denoted as N, left subtree as L and right subtree as R, then there will be six combinations of traversals NRL,NLR,LNR,LRN,RNL,RLN. But only three are standard, NLR(node-left-right),LNR(left-node-rignt) and LRN(left-right-node) traversal. We can see left subtree is always traversed before right subtree. NLR is called preorder LNR is inorder NRL is postorder.

These traversals are as -

Preorder(NLR Traversal)
1.Visit the root.
2.Traverse the left subtree of root in preorder.
3.Traverse the right subtree of root in preorder.

Inorder Traversal(LNR Traversal)
1.Traverse the left subtree of root in preorder.
2.Visit the root.
3. Traverse the right subtree of root in preorder.

Postorder(LRN Traversal)
1.Traverse the left subtree of root in postorder.
2.Traverse the right subtree of root in postorder.
3.Visit the root.

We will create the function for preorder traversal as-

```
preorder(ptr)
struct node *ptr;
{
        if (ptr!=NULL)
        {
                printf("%c",ptr→data);
                preorder(ptr→lchild); /* calling with address of left child */
                preorder(ptr→rchild); /* calling with address of  right child */
        }
}
```

Here we are calling function recursively for left and right child.

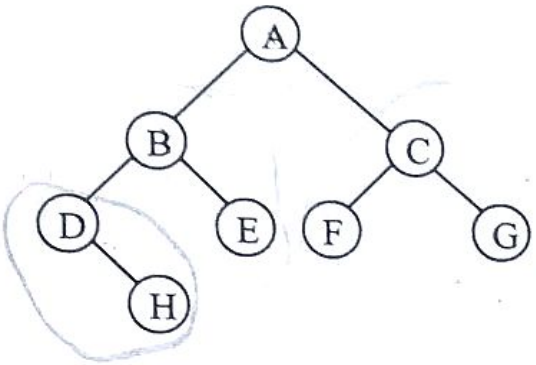
Similarly the function for inorder traversal will be as-

```
inorder(ptr)
struct  node *ptr;
{
        if (ptr!=NULL)
        {
                inorder(ptr→lchild);
                printf("%c",ptr→data)
                inorder(ptr→rchild)
        }
}
```

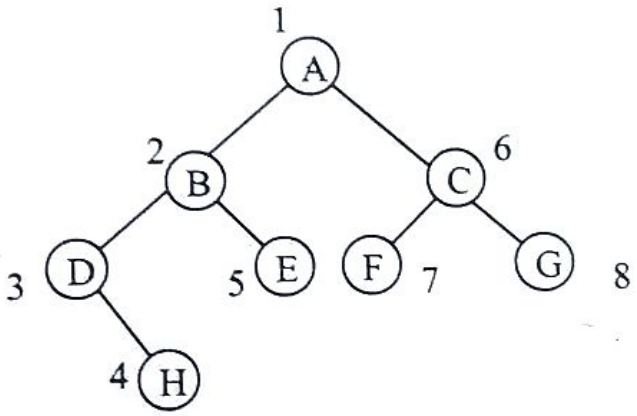
Function for postorder traversal is as-

```
postorder(ptr)
struct node *ptr;
{
        if (ptr!=NULL)
        {
                postorder(ptr→lchild);
                postorder(ptr→rchild);
                printf("%c",ptr→data);
        }
}
```

Let us take a binary tree and apply each traversal.

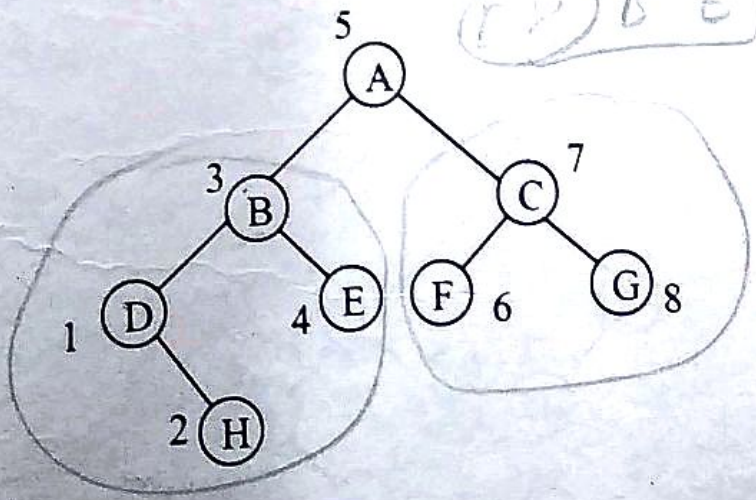


Preorder Traversal



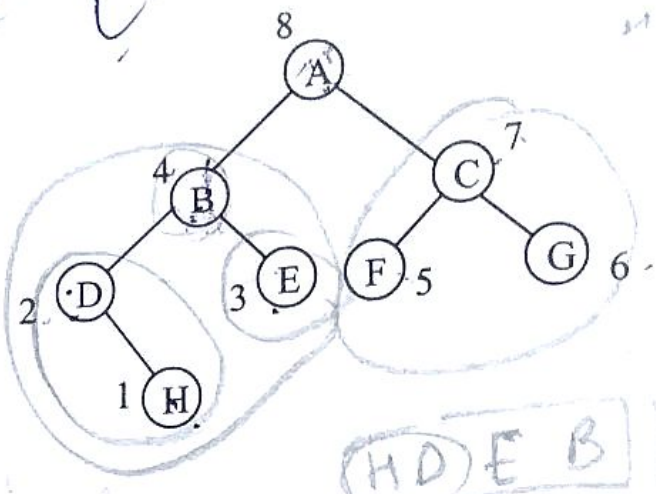
Here numbers represent the sequence of visited node. The nodes are visited in preorder as-
ABDHECFG

Inorder Traversal



The nodes are visited in inorder as-
DHBEAFCG

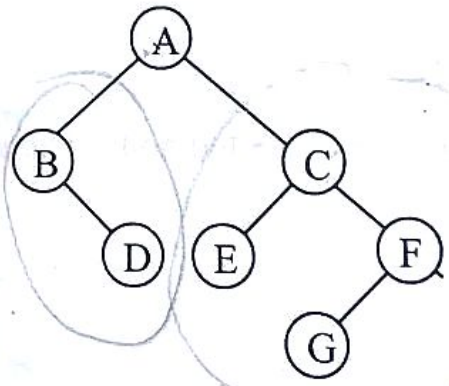
Postorder Traversal



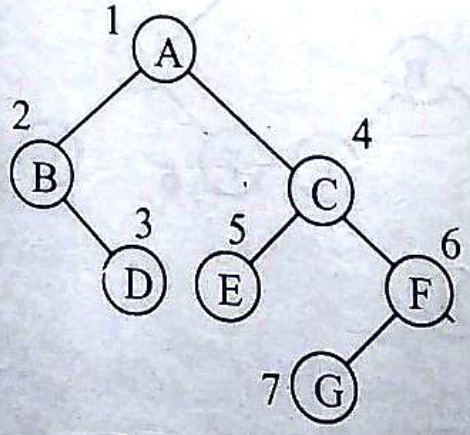
The nodes are visited in postorder as-
HDEBFGCA

Now we can see through recursion every node is traversed and it creates a copy of every call just as factorial program through recursion.

Let us take another binary tree and apply each traversal.

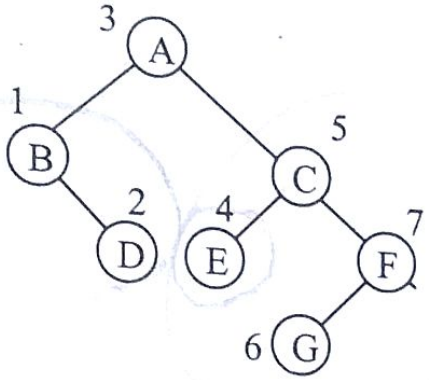


Preorder Traversal



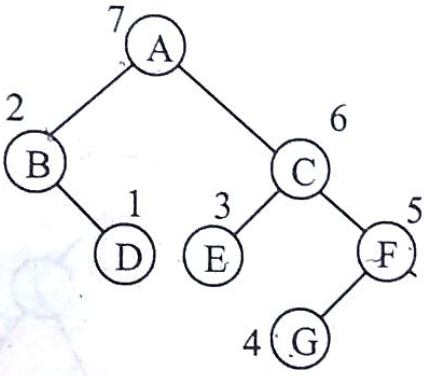
The nodes are visited in preorder as -
ABDCEFG

Inorder Traversal



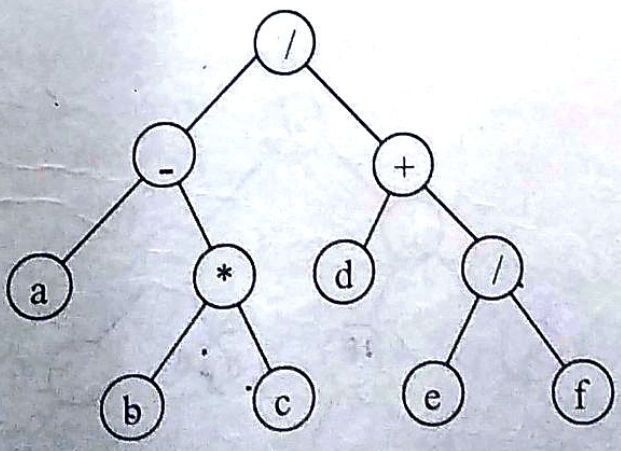
The nodes are visited in inorder as -
BDAECGF

Postorder traversal



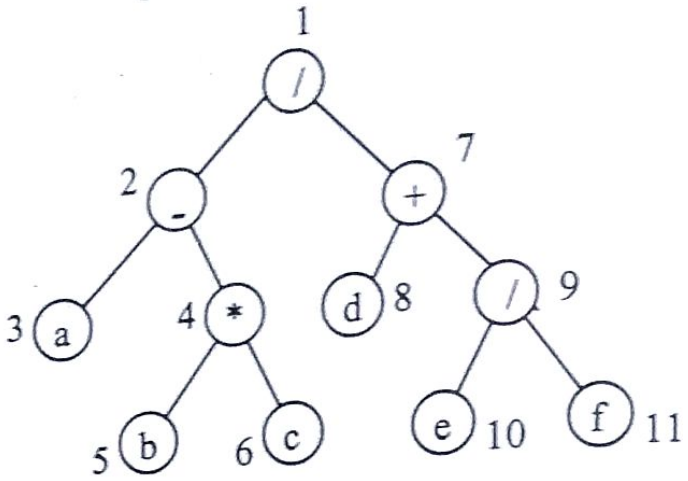
The nodes are visited in postorder as -
DBEGFCA

Let us take an algebraic expression in binary tree and appl each traversal.
Algebraic expression
(a – b * c) / (d + e / f)



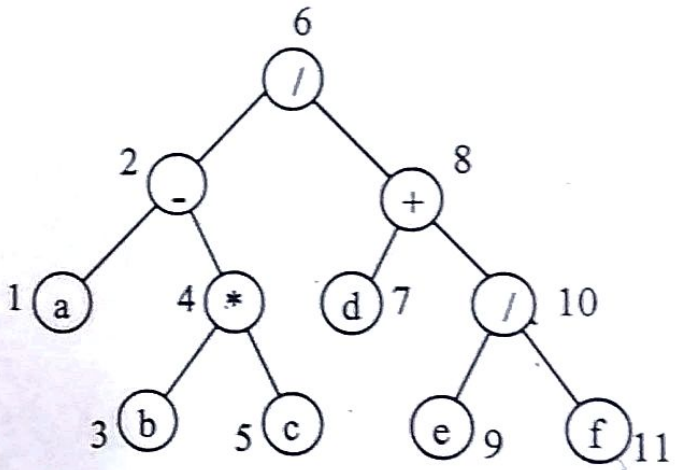
Binary tree representation of expression

Preorder Traversal



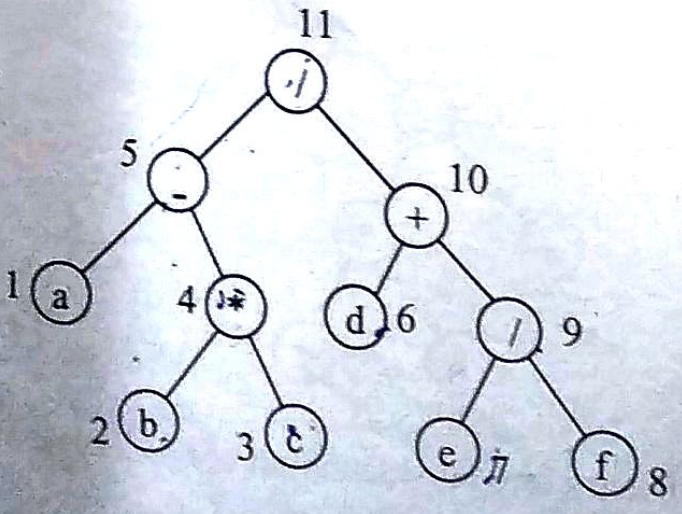
The nodes are visited in preorder as -
/-a*bc+d/ef
This is same as prefix of algebraic expression.

Inorder Traversal



The nodes are visited in inorder as-
a-b*c/d+e/f
This is same as infix of algebraic expression.

Postorder Traversal



The nodes are visited in postorder as-
abc*-def/+/
This is same as postfix of algebraic expression.

We can also make the tree if preorder and inorder traversals or postorder and inorder traversals are given.

...ions for traversals-

Creation of binary tree from preorder and inorder traversals -

Preorder ABDHECFG
Inorder DHBEAFCG

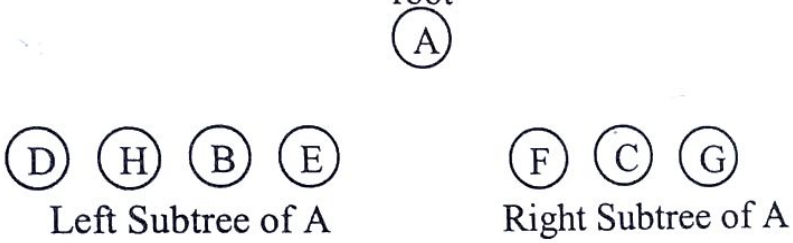
1. In preorder traversal root is the first node. Hence A is the root of the binary tree

root

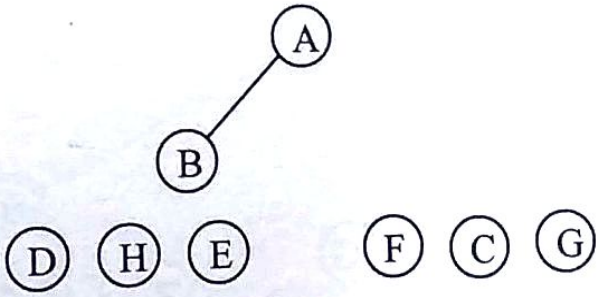
2. Now we can find the node of left subtree and right subtree with inorder sequence. Nodes which are in the left side of root in inorder are nodes of left subtree and nodes which are in the right side of root in inorder are nodes of right subtree.

Nodes of left subtree – DHBE
Nodes of right subtree – FCG

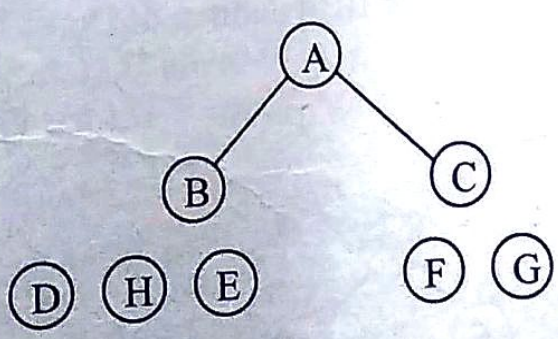
root



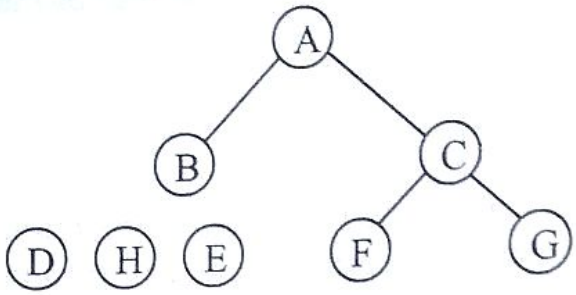
3. The left child of the root A will be the first node in preorder traversal after root A Hence B is the left child of A.



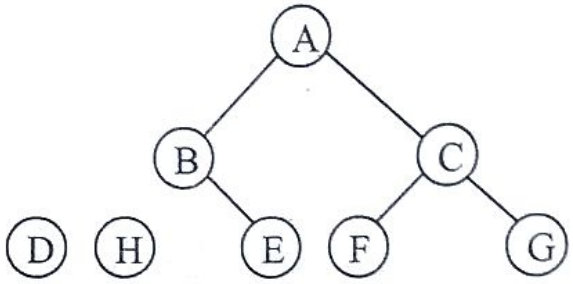
4. The right child of root A will be the first node after nodes of left subtree in preorder traversal. Hence C is the right child of A.



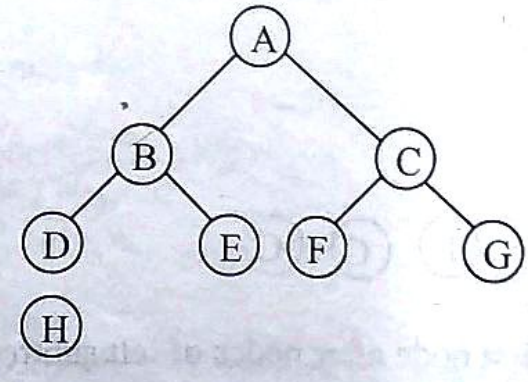
5. In inorder sequence, F is on the left of C and G is on the right side of C. Hence F will be in left subtree of C and G in right subtree of C.



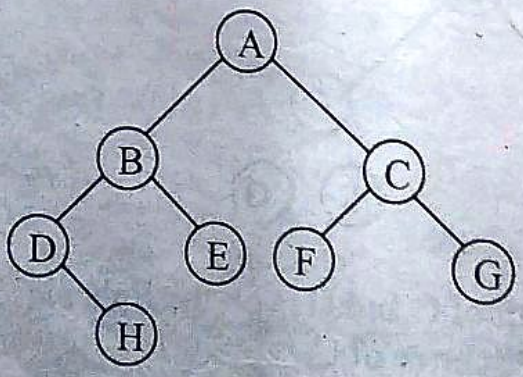
6. Now in inorder sequence, D and H are on the left side of B and E is on the right side of B. So D and H will form left subtree of B and E will be in right subtree of B.



7. In preorder sequence, root is traversed before left and right subtrees. In preorder traversal of this tree, D is coming before H, hence D is the root of left subtree of B and H can be either in left or right subtree of D.



8. To find out whether H is in left or right subtree of D, we look at inorder traversal. Since H is in right side of D, hence it will be in right subtree of D.



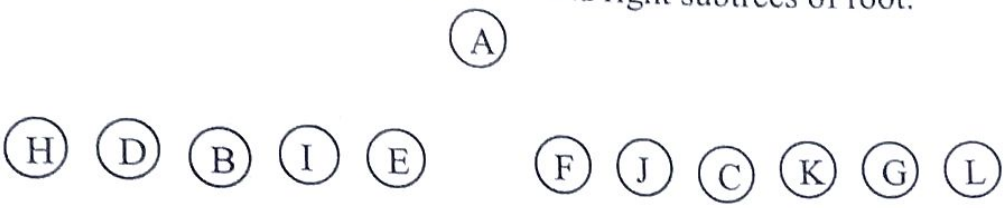
Creation of tree from postorder and inorder traversals

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

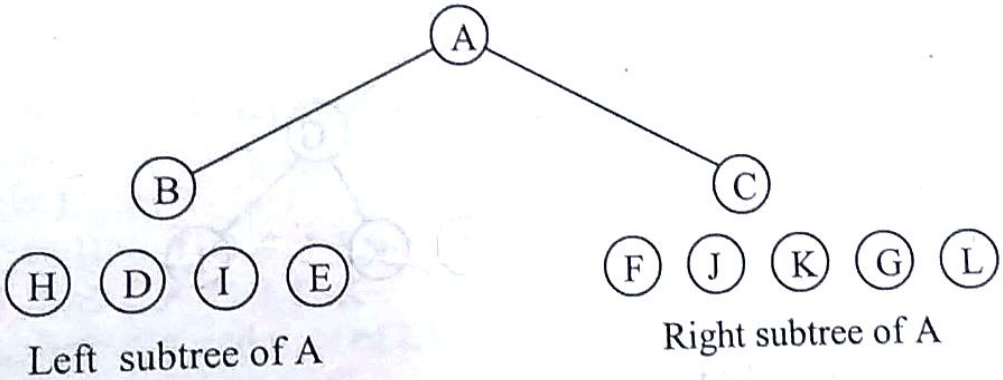
1. In postorder traversal root is the last node. Hence A is the root of the binary tree

root

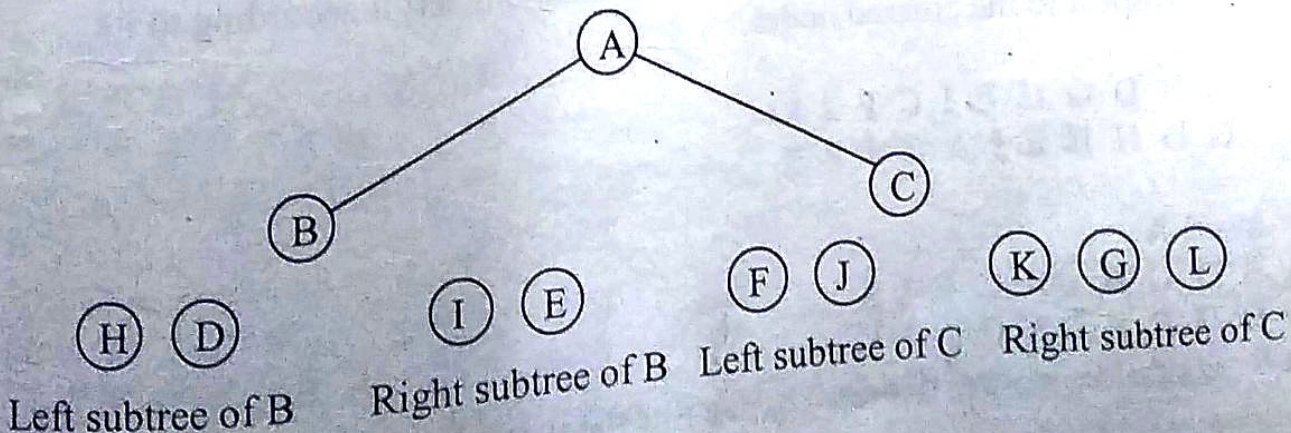
2. From inorder traversal we can find out left and right subtrees of root.



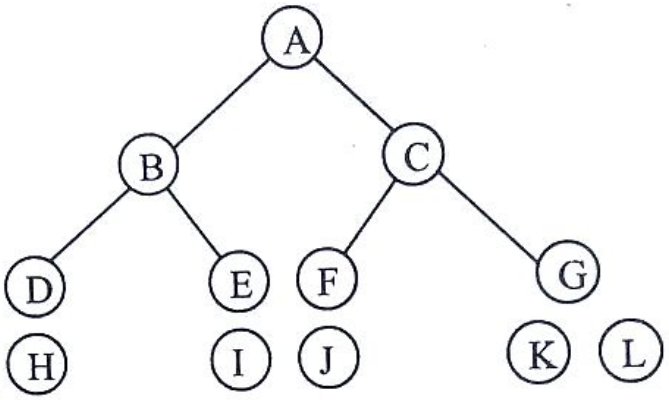
3. Now right child of A will be the node which comes just before node A. Hence C is right child of node A. Left child of A will be the first node before nodes of right subtree in postorder traversal. Hence B is left child of A.



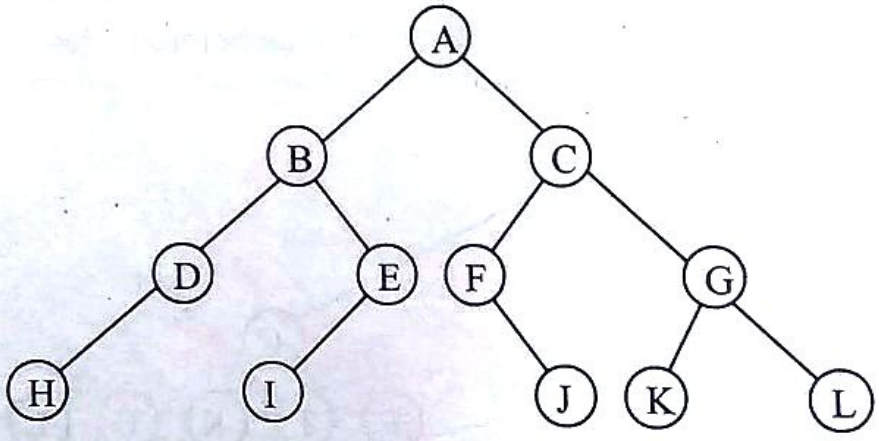
4. Now look at C in the inorder traversal. F and J are to the left of C and K,G,L are to the right of it, hence F,J form left subtree of C and K,G,L form right subtree of C. Now look at B in inorder traversal. H,D are to the left of B and I,E are to the right of B hence H,D form left subtree of B and I,E Form right subtree of B.



5. Now look at postorder traversal, node just before B is E, hence E is right child of B and node just before C is G, hence G is right child of C. D is the first node before the nodes of right subtree of B hence D is right child of B. F is the first node before nodes of right subtree of C. Hence F is left child of C.



6. Now look at inorder traversal, H is to the left of D hence it is left child of D, I is to the left of E, hence it is left child of E. J is to the right of F hence it is right child of F. K is to the left of G and L to the right of G hence K is left child of G and L is right child of G.



Shortcut method of creating the tree from preorder and inorder traversal

Creation of tree by this method is very simple and quick. In preoder traversal, scan the nodes one by one and keep them inserting in the tree. In inorder traversal, put a cross mark over the node which has been inserted. To insert a node in its proper position in the tree, we will look at that node in the inorder traversal and insert it according to its position with respect to the crossed nodes.

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

Step 1 Insert A
Pre **A** B D G H E I C F J K
In G D H B E I **A** C J F K
A is the first node in preorder traversal hence it is the root of the tree.

Step 2 Insert B

Pre A **B** D G H E I C F J K
In G D H **B** E I A C J F K

Since B is to left of A in inorder traversal hence it is left child of A.

Step 3 Insert D

Pre A B **D** G H E I C F J K
In G **D** H B E I A C J F K

Since D is to the left of B in inorder traversal hence D is left child of B.

Step 4 Insert G

Pre A B D **G** H E I C F J K
In **G** D H B E I A C J F K

Since G is to the left of D in inorder traversal hence is G is left child of D.

Step 5 Insert H

Pre A B D G **H** E I C F J K
In G D **H** B E I A C J F K

Since H is to the left of B and right of D in inorder traversal hence H is right child of D.

Step 6 Insert E

Pre A B D G H **E** I C F J K
In G D H B **E** I A C J F K

Since E is to the left of A and right of B in inorder traversal hence E is right child of B.

Step 7 Insert I

Pre A B D G H E **I** C F J K
In G D H B E **I** A C J F K

Since I is to the left of A and right of E in inorder traversal hence I is right child of E.

Step 8 Insert C

Pre A B D G H E I **C** F J K
In G D H B E I A **C** J F K

Since C is to the right of A in inorder traversal hence C is right child of A.

Step 9 Insert F

Pre A B D G H E I C **F** J K
In G D H B E I A C J **F** K

Since F is to the right of C in inorder traversal hence F is right child of C.

Step 10 Insert J

Pre A B D G H E I C F **J** K
In G D H B E I A C **J** F K

Since J is to the right of C and to left of F in inorder traversal hence J is left child of F.

Step 11 Insert K

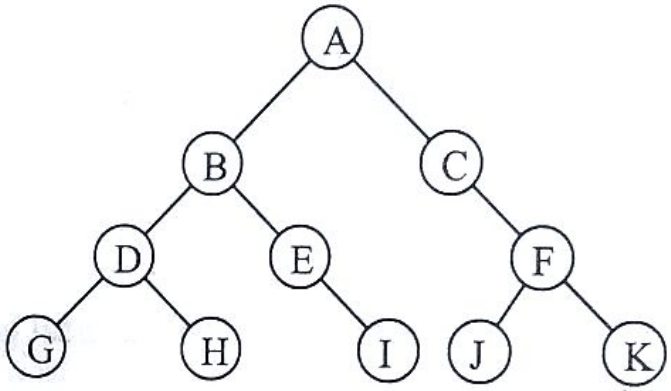
Pre A B D G H E I C F J **K**

In ~~G D H B E I A C J F~~ **K**

Since K is to the right of F in inorder traversal hence K is right child of F.

Draw the tree with each step and when all the nodes have been inserted, you will get the final tree. Here we have shown all the steps, but while creating tree on your own just write the preorder and inorder traversal once and keep on scanning the nodes in preorder and cross the inserted nodes in inorder and simultaneously construct your tree by inserting nodes.

The tree for the above preorder and inorder traversal will be-

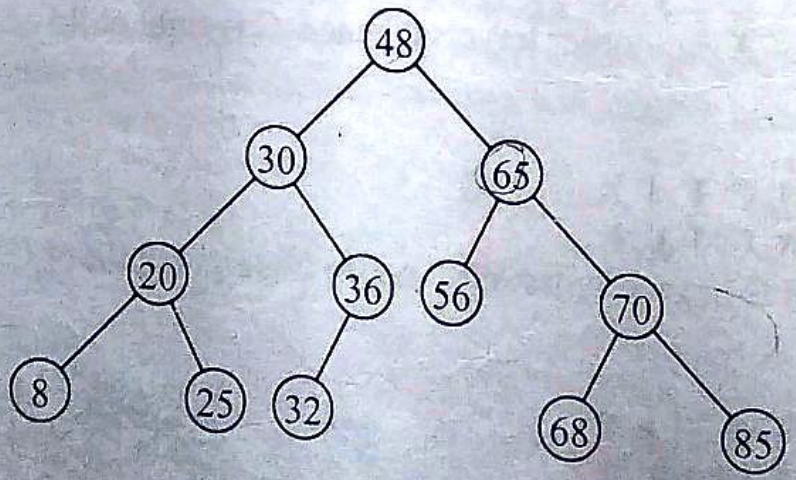


Creation of tree from postorder traversal and inorder traversal by shortcut method is same as creation of tree from inorder and preorder. The only difference is that here we will start scanning the nodes from the right side means the last node in the postorder traversal will be inserted first and first node will be inserted in the last.

Binary Search Tree

A binary search tree is a binary tree in which each node has value greater than every node of left subtree and less than every node of right subtree.

Binary search tree is very useful data structure in which item can be searched in $O(\log_2 N)$ where N is the number of nodes.



Binary Search Tree

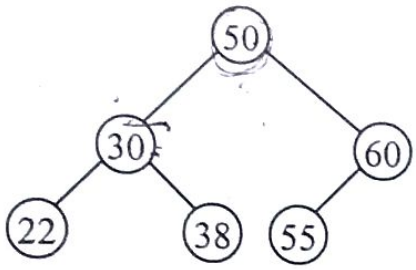
Search and Insertion Operations

Searching and insertion is one time operation in binary search tree because before insertion of any element we first search the exact place for inserting.

Suppose a data is given and we want to search and insert that data. This can be done as -
1.Compare data with data of root node
(a) If data<data of node then compare with data of left child node.
(b) if data >data of node then compare with data of right child node

At last, we find the node which has same value as data or we will reach the exact place where we will insert the node.

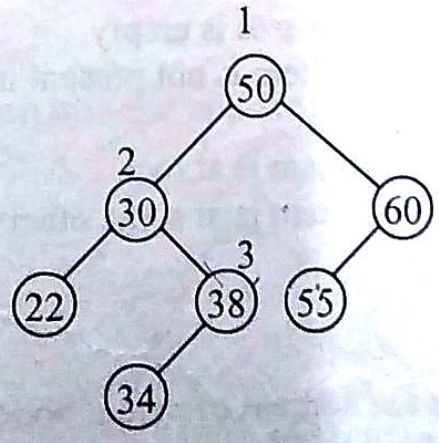
Let us take a binary search tree.



We want to insert a node which has value 34.

Steps-

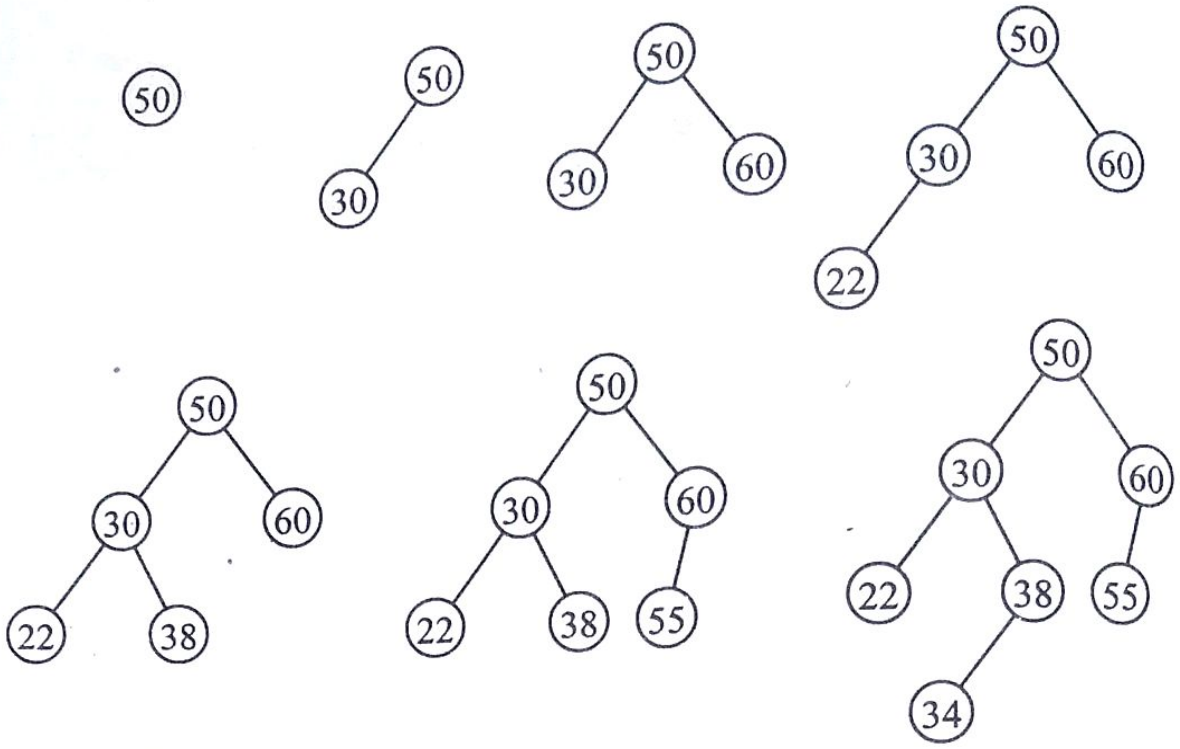
1. Compare 34 with 50. 34<50 so compare with left child of 50 which is 30.
2. Compare 34 with 30. 34>30 so compare with right child of 30 which is 38.
3. Compare 34 with 38. 34<38 so compare with left child of 38 but 38 has no left child so this is the exact place to insert 34. We will insert 34 as a left child of 38.



Here numbers show the path for insertion of new node.

Let us take seven numbers and insert them in binary tree which is initially empty.

50, 30, 60, 22, 38, 55, 34



Creation of find( )

This function find( ) is used both in deletion and insertion of nodes in binary search tree. The arguments to this function are the item to be found, address of location pointer and address of parent pointer. Note that we are passing here parent and location pointers by reference, why are we doing so. Generally, we pass parameters by reference when we want the function to return more than one value to the program. Here we want the find( ) function to return the address of item and address of parent also. So we are passing address of pointers to the find( ) function and hence the last 2 arguments to this function are a pointer to pointer. So arguments loc and par are pointer to pointer.

After calling find( ) function location and parent pointer will be assigned with some values and based on these values we can draw some conclusions.

(a) If *loc==NULL and *par==NULL means tree is empty.
(b) If *loc==NULL and *par!=NULL means item is not present in tree.
(c) If *loc!=NULL means item is present in tree.
If *loc!=NULL and *par==NULL item is at root.
If *loc!=NULL and *par!=NULL item is at some other place in the tree.

Insertion in Binary Search tree

Insertion in Binary search tree requires the address of parent node where we will insert the new node. So we call the function find( ) as-

find(item,&parent,&location);

After calling this function, we get the address of parent node and address of item in location pointer, if item already exists in tree. Since we are not allowing duplicate values,

so first we check this condition-

```
if(location!=NULL)
{
        printf("Item already present");
        return;
}
```

If location is NULL value means added item does not exist in tree. So we can add this in tree.

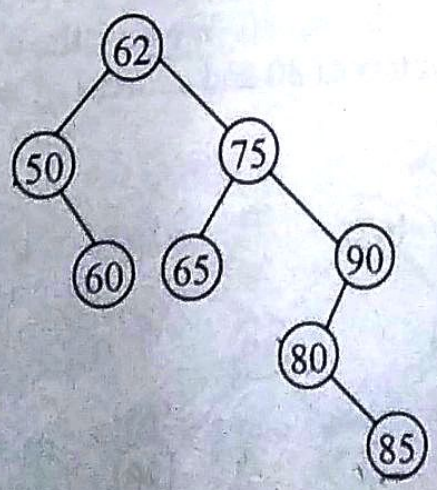
Now we will insert the value at proper position. First we will check that element will be added as a root if tree is empty, otherwise it will be added as left terminal node or right terminal node as-

```
tmp->info=item;
tmp->lchild=NULL;
tmp->rchild=NULL;
if(parent = = NULL)
        root=tmp;
else
        if(item<parent->info)
                parent->lchild=tmp;
        else
                parent->rchild=tmp;
```

Since it will be terminal node so left and right child of new node will be NULL. If new node will be added as a root then we assign the address of new node to the root. If new node will be added as left child then we assign the new node address to the lchild part of parent node. If new node will be added as right child then we assign the address of new node to the rchild part of parent node.

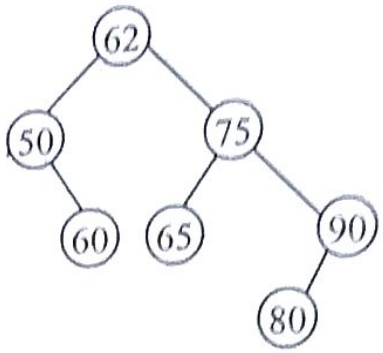
Deletion operation

Let us take binary search tree and apply delete operation



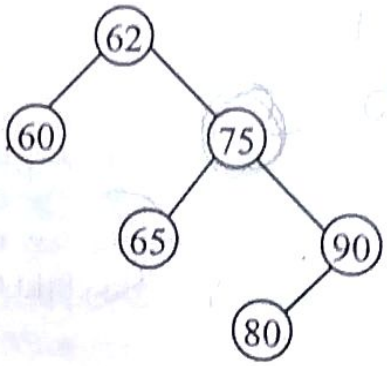
Now we want to delete the item 85. Since 85 has no children. So we can delete it simply by giving NULL value to its parent's right pointer. Here 80 is the parent of 85 and 85 is the right child of 80, so after deleting 85, right pointer of 80 will have NULL value.

Now the binary search tree will be as-

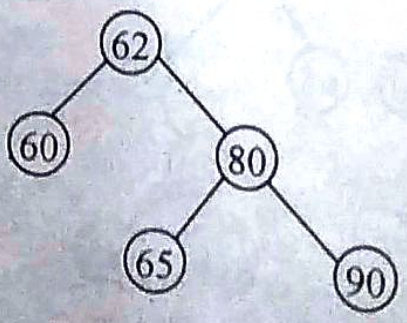


Now we want to delete the item 50. Since 50 has only one child. So we can delete it simply by giving the address of right child to its parent left pointer.
Here 62 is the parent of 50 and 60 is the right child of 50.

Now the binary search tree will be as –



Now we want to delete the item 75. Since 75 has two children, so first we delete the item which is inorder successor of 75. Here 80 is the inorder successor of 75. We delete 80 by simply giving the NULL value to its parent's left pointer. Here 90 is the parent and 80 is the left child of 90.
After that we replace the item 75 with item 80. We give the address of the left and right pointers of 75 to left and right pointers of 80 and address of 80 to the right pointer of parent 75 which is 62.
Now the binary search tree will be-



Deletion in Binary search tree requires the address of item to be deleted and address of parent node. Then only we can delete the item. So we call the function find( ) as-

find(item,&parent,&location);

After calling this function, we get the address of item in location pointer and address of parent node. If we get NULL value in location pointer then that item doesn't exist in tree.

There can be 5 possibilities while performing delete operation in binary search tree-

1. There are no nodes in the tree i.e the tree is empty.
2. Node to be deleted is not present in the tree.
3. Node to be deleted is leaf node i.e it has no children.
4. Node to be deleted has only one child.
5. Node to be deleted has two children.

We'll check for these possibilities in our del( ) function and take appropriate action.

1. First we check the condition for tree empty as-

```
if(root==NULL)
{
        printf("Tree is empty");
        return;
}
```

2. Now we have a need to get the address of item to be deleted and address of it's parent node. So we call find( ) function as-

find(item,&parent,&location);

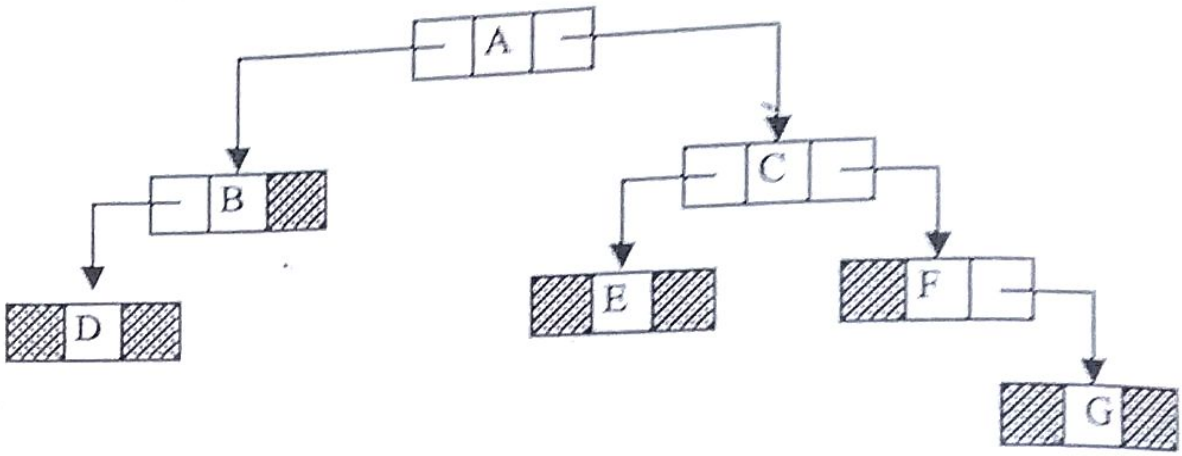
After calling this function, we get the address of item in location pointer and address of parent node in parent pointer. If we get NULL value in location pointer then that item doesn't exist in tree. So we check for item existence as-

```
if(location==NULL)
{
        printf("Item not present in tree");
        return;
}
```

Now we have address of node to be deleted and address of it's parent node. Now deletion can be for three cases-

1. Node to be deleted is leaf node i.e it has no children.
2. Node to be deleted has only one child.
3. Node to be deleted has two children.

We can see these cases below-



Tree T1

Case 1-

If the node is a leaf node i.e it has no children, then its left and right pointers will contain NULL. Here D, E and G are leaf nodes. This will be checked as-

```
if(location->lchild==NULL && location->rchild==NULL)
        case_a(parent,location);
```

Case 2-

If the node has only one child, that child can be either left child or right child. Here node B has only left child so its left pointer is not NULL and right pointer is NULL. So we have a need to check for both conditions as-

```
if(location->lchild!=NULL && location->rchild==NULL)
        case_b(parent,location);
```

Node F has only right child, so its right pointer is not NULL and left pointer is NULL.

```
if(location->lchild==NULL && location->rchild!=NULL)
        case_b(parent,location);
```

Case 3-

Now we have a need to consider the condition when the node has both left and right child. This can be checked as-

```
if(location->lchild!=NULL && location->rchild!=NULL)
        case_c(parent,location);
```

Now we will see the creation of function case_a( ), case_b( ) and case_c( ).

Creation of function case_a( )-

Deleting a leaf node is very simple. First we check if node to be deleted is root then we assign NULL value to root, otherwise if node is left child then NULL will be assigned to lchild part of parent node and if it's right child then NULL will be assigned to rchild part of parent node.

```
if(par==NULL) /*item to be deleted is root node*/
        root=NULL;
else
        if(loc==par->lchild)
                par->lchild=NULL;
        else
                par->rchild=NULL;
```

Suppose we want to delete the node G in tree T1 then we assign NULL to right pointer of its parent F, since G is right child of F. Similarly for deleting node D we assign NULL to left pointer of B since it is left child of B.

Creation of function case_b( )-

If item to be deleted has only one child, we can delete it by giving address of its child to its parent. First we check that node to be deleted has left child or right child. If it has only left child then we store the address of left child and if it has only right child then we store the address of it's right child in pointer variable child as-

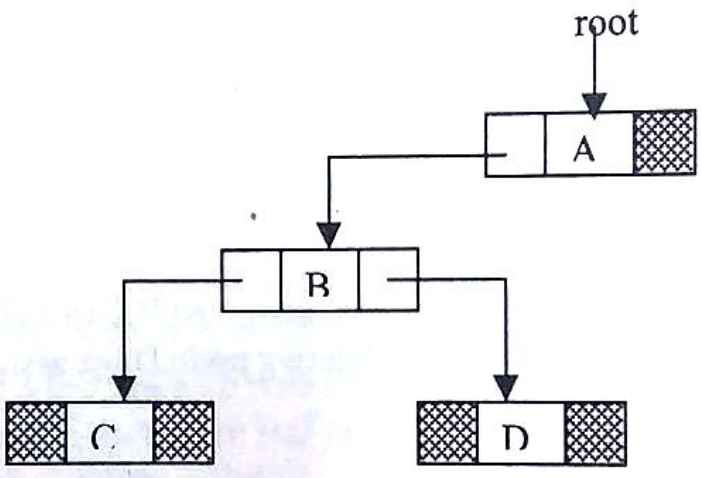
```
if(loc->lchild!=NULL)
        child=loc->lchild;
else
        child=loc->rchild;
```

Now we check that if node to be deleted is root node then we assign value of pointer variable child to root node. So child of deleted node(root) will become the root node. Otherwise, we check that node to be deleted is left child or right child of its parent. If it is left child then we assign the value of child pointer variable to lchild part of its parent. So child of node to be deleted will become the left child of it's parent. If it is right child then we assign the value of child pointer variable to rchild part of its parent. So child of node to be deleted will become the right child of its parent.

```
if(par = = NULL )
        root = child;
else
        if( loc = = par->lchild)
                par->lchild = child;
        else
                par->rchild = child;
```

In tree T1, for deleting B, put the address of its child G in the right pointer of its parent C since F is right child of C.

As in case_a( ) we had treated the condition of node to be deleted being the root node, here also we'll treat it separately, since the value of root will change.

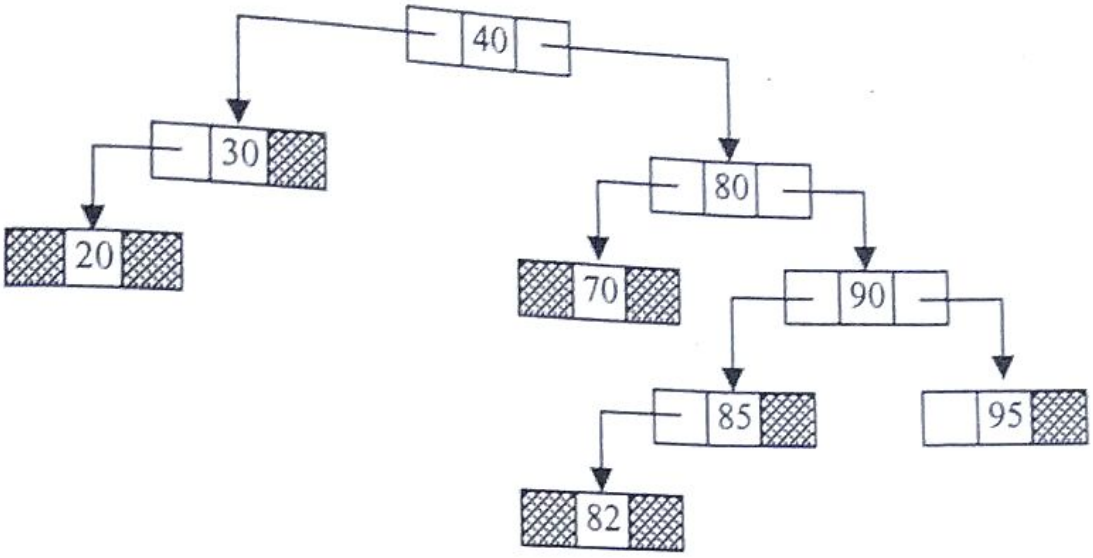


To delete A which is the root node we will assign the value of its child to the root pointer So now child of the deleted node will become the root node. Hence, after deleting A, root pointer will point to node B.

Creation of function case_c( )-

If the node to be deleted has two children then first delete the inorder successor of node and then replace the deleted node with the inorder successor.
Inorder successor of a node is the node which comes just after that node in the inorder traversal of the tree. Inorder succesor of any node will be the leftmost node of the right subtree of that node. Inorder successor can have either no child or only one right child. It can't have left child because then that left child will become the inorder successor Hence inorder successor can't have two children.

So to delete inorder successor we again have to call case_a( ) or case_b( ) depending on whether it has no child or one child. For deleting a node by case_a( ) or case_b( ) we need to send the address of that node and address of its parent. So to delete inorder successor we need to find out its address and address of its parent. Let us take a tree-



Inorder traversal of this tree is –

20 30 40 70 80 82 85 90 95

Suppose we have to delete node 80.Then we have to find address of its inorder successor and parent of inorder successor. From the inorder traversal we can see that inorder successor of 80 is 82.Now we'll see how we'll find out this in our program.
We know that inorder successor of a node is the leftmost node in the right subtree of that node. So initially we take two pointers ptr and ptrsave. Initialize ptrsave with the address of node to be deleted and ptr with the right child of node because we want to traverse right subtree of that node. To find out leftmost node of this right subtree, we will keep on traversing to left until we reach a node which has NULL in its left pointer, means there is no node left to it and we have reached the leftmost node.

```
ptrsave=loc;
ptr=loc->rchild;
while(ptr->lchild!=NULL)
{
        ptrsave=ptr;
        ptr=ptr->lchild;
}
```

When this loop will terminate, ptr will contain address of inorder successor and ptrsave will contain address of parent of inorder successor.

```
suc=ptr;
parsuc=ptrsave;
```

Now we have to delete successor from the tree so we'll call case_a( ) or case_b( ) depending on whether node has no child or one child.

```
if(suc->lchild==NULL && suc->rchild==NULL)
        case_a(parsuc,suc);
else
        case_b(parsuc,suc);
```

Now we have to replace the node to be deleted with the inorder successor. If node to be deleted is left child of its parent then put address of successor in left pointer of parent of that node otherwise put address of successor in right pointer of parent of that node. Here also like case_a( ) and case_b( ) we'll consider the condition of node to be deleted being the root node separately.

```
if(par==NULL)
        root=suc;
else
        if(loc==par->lchild)
                par->lchild=suc;
        else
                par->rchild=suc;
```

We have attached successor with the parent of the deleted node. Now we have to attach the successor with children of the deleted node so that the successor comes at the place of deleted node fully. So we will put the address of left child of node being deleted in the left pointer of successor and similarly for right child.

```
suc->lchild=loc->lchild;
suc->rchild=loc->rchild;
```

Now we see that successor has been removed from its original place in the tree and now it is at the place of deleted node. Note that replacement of node is done by exchanging pointers and not just by copying the contents.

```
case_c(struct node *par,struct node *loc)
{
        struct node *ptr,*ptrsave,*suc,*parsuc;

        ptrsave=loc;
        ptr=loc->rchild;
        while(ptr->lchild!=NULL)
        {
                ptrsave=ptr;
                ptr=ptr->lchild;
        }
        suc=ptr;
        parsuc=ptrsave;

        if(suc->lchild==NULL && suc->rchild==NULL)
                case_a(parsuc,suc);
```

```
        else
                case_b(parsuc,suc);

        if(par==NULL) /*if item to be deleted is root node */
                root=suc;
        else
                if(loc==par->lchild)
                        par->lchild=suc;
                else
                        par->rchild=suc;

        suc->lchild=loc->lchild;
        suc->rchild=loc->rchild;
}
```

Traversal in Binary Search Tree

Traversal in Binary Search Tree is same as preorder, inorder and postorder traversal of binary Tree.

Preorder Traversal

```
if(ptr!=NULL)
{
        printf("%d ",ptr->info);
        preorder(ptr->lchild);
        preorder(ptr->rchild);
}
```

Inorder Traversal

```
if(ptr!=NULL)
{
        inorder(ptr->lchild);
        printf("%d ",ptr->info);
        inorder(ptr->rchild);
}
```

Postorder Traversal

```
if(ptr!=NULL)
{
        postorder(ptr->lchild);
        postorder(ptr->rchild);
        printf("%d ",ptr->info);
}
```

```
/*Insertion,  Deletion and Traversal in Binary Search Tree*/
# include <stdio.h>
# include <malloc.h>

struct node
{
        int info;
        struct node *lchild;
        struct node *rchild;
}*root;

main( )
{
        int choice,num;
        root=NULL;
        while(1)
        {
                printf("\n");
                printf("1.Insert\n");
                printf("2.Delete\n");
                printf("3.Inorder Traversal\n");
                printf("4.Preorder Traversal\n");
                printf("5.Postorder Traversal\n");
                printf("6.Quit\n");
                printf("Enter your choice : ");
                scanf("%d",&choice);

                switch(choice)
                {
                case 1:
                        printf("Enter the number to be inserted : ");
                        scanf("%d",&num);
                        insert(num);
                        break;
                case 2:
                        printf("Enter the number to be deleted : ");
                        scanf("%d",&num);
                        del(num);
                        break;
                case 3:
                        inorder(root);
                        break;
                case 4:
                        preorder(root);
                        break;
                case 5:
                        postorder(root);
                        break;
```

```
                case 6:
                        exit( );
                default:
                        printf("Wrong choice\n");
                }/*End of switch */
        }/*End of while */
}/*End of main( )*/

find(int item,struct node **par,struct node **loc)
{
        struct node *ptr,*ptrsave;

        if(root==NULL)  /*tree empty*/
        {
                *loc=NULL;
                *par=NULL;
                return;
        }
        if(item==root->info) /*item is at root*/
        {
                *loc=root;
                *par=NULL;
                return;
        }
        /*Initialize ptr and ptrsave*/
        if(item<root->info)
                ptr=root->lchild;
        else
                ptr=root->rchild;
        ptrsave=root;

        while(ptr!=NULL)
        {
                if(item==ptr->info)
                {       *loc=ptr;
                        *par=ptrsave;
                        return;
                }
                ptrsave=ptr;
                if(item<ptr->info)
                        ptr=ptr->lchild;
                else
                        ptr=ptr->rchild;
        }/*End of while */
        *loc=NULL;   /*item not found*/
        *par=ptrsave;
}/*End of find( )*/
```

```
insert(int item)
{       struct node *tmp,*parent,*location;
          find(item,&parent,&location);
          if(location!=NULL)
          {
                  printf("Item already present");
                  return;
          }

          tmp=(struct node *)malloc(sizeof(struct node));
          tmp->info=item;
          tmp->lchild=NULL;
          tmp->rchild=NULL;

          if(parent==NULL)
                  root=tmp;
          else
                  if(item<parent->info)
                          parent->lchild=tmp;
                  else
                          parent->rchild=tmp;
}/*End of insert( )*/

del(int item)
{
        struct node *parent,*location;
        if(root==NULL)
        {
                printf("Tree empty");
                return;
        }

        find(item,&parent,&location);
        if(location==NULL)
        {
                printf("Item not present in tree");
                return;
        }

       if(location->lchild==NULL && location->rchild==NULL)
                case_a(parent,location);
       if(location->lchild!=NULL && location->rchild==NULL)
                case_b(parent,location);
       if(location->lchild==NULL && location->rchild!=NULL)
                case_b(parent,location);
       if(location->lchild!=NULL && location->rchild!=NULL)
                case_c(parent,location);
       free(location);
}/*End of del( )*/
```

```
case_a(struct node *par,struct node *loc )
{
        if(par==NULL) /*item to be deleted is root node*/
                root=NULL;
        else
                if(loc==par->lchild)
                        par->lchild=NULL;
                else
                        par->rchild=NULL;
}/*End of case_a( )*/

case_b(struct node *par,struct node *loc)
{
        struct node *child;

        /*Initialize child*/
        if(loc->lchild!=NULL) /*item to be deleted has lchild */
                child=loc->lchild;
        else            /*item to be deleted has rchild */
                child=loc->rchild;
        if(par==NULL )  /*Item to be deleted is root node*/
                root=child;
        else
                if( loc==par->lchild)  /*item is lchild of its parent*/
                        par->lchild=child;
                else            /*item is rchild of its parent*/
                        par->rchild=child;
}/*End of case_b( )*/

case_c(struct node *par,struct node *loc)
{
        struct node *ptr,*ptrsave,*suc,*parsuc;

        /*Find inorder successor and its parent*/
        ptrsave=loc;
        ptr=loc->rchild;
        while(ptr->lchild!=NULL)
        {
                ptrsave=ptr;
                ptr=ptr->lchild;
        }
        suc=ptr;
        parsuc=ptrsave;

        if(suc->lchild==NULL && suc->rchild==NULL)
                case_a(parsuc,suc);
        else
                case_b(parsuc,suc);
```

```
            if(par==NULL) /*if item to be deleted is root node */
                    root=suc;
            else
                    if(loc==par->lchild)
                            par->lchild=suc;
                    else
                            par->rchild=suc;

            suc->lchild=loc->lchild;
            suc->rchild=loc->rchild;
    }/*End of case_c( )*/

    preorder(struct node *ptr)
    {
            if(root==NULL)
            {
                    printf("Tree is empty");
                    return;
            }
            if(ptr!=NULL)
            {
                    printf("%d ",ptr->info);
                    preorder(ptr->lchild);
                    preorder(ptr->rchild);
            }
    }/*End of preorder( )*/

    inorder(struct node *ptr)
    {
            if(root==NULL)
            {
                    printf("Tree is empty");
                    return;
            }
            if(ptr!=NULL)
            {
                    inorder(ptr->lchild);
                    printf("%d ",ptr->info);
                    inorder(ptr->rchild);
            }
    }/*End of inorder( )*/

    postorder(struct node *ptr)
    {
            if(root==NULL)
            {
                    printf("Tree is empty");
                    return;
            }
```

```
        if(ptr!=NULL)
        {
                postorder(ptr->lchild);
                postorder(ptr->rchild);
                printf("%d  ",ptr->info);
        }
}/*End of postorder( )*/
```

Recursive Function for finding a node in Binary search tree

Initially value of ptr will be root.

```
struct node* search(struct node *ptr, int info)
{
        if(ptr!=NULL)
                if(info < ptr->info)
                        ptr=search(ptr->lchild,info);
                else if( info > ptr->info)
                        ptr=search(ptr->rchild,info);

        return(ptr);
}/*End of search( )*/
```