SCS1201 – ADVANCED DATA STRUCTURES

UNIT – I

TREES:

A tree is a non-linear data structure that is used to represents hierarchical relationships between individual data items.

A tree is an ideal data structure for representing hierarchical data. A tree can be theoretically defined as a finite set of one or more data items (or nodes) such that:

1. There is a special node called the root of the tree.
2. Removing nodes (or data item) are partitioned into number of mutually exclusive (i.e., Disjoined) subsets each of which is itself a tree are called sub tree.

Ordinary and Binary Trees Terminology:

Tree: *A tree is a finite set of one or more nodes such that, there is a specially designated node called root. The remaining nodes are partitioned into n>=0 disjoint sets T1, T2,..Tn, where each of these set is a tree T1,…Tn are called the subtrees of the root.*

Branch: *Branch is the link between the parent and its child.*

Leaf: *A node with no children is called a leaf.*

Subtree: *A Subtree is a subset of a tree that is itself a tree.*

Degree: *The number of subtrees of a node is called the degree of the node. Hence nodes that have degree zero are called leaf or terminal nodes. The other nodes are referred as non-terminal nodes.*

Children: *The nodes branching from a particular node X are called children of X and X is called its parent.*

Siblings: *Children of the same parent are said to be siblings.*

Degree of tree: *Degree of the tree is the maximum of the degree of the nodes in the tree.*

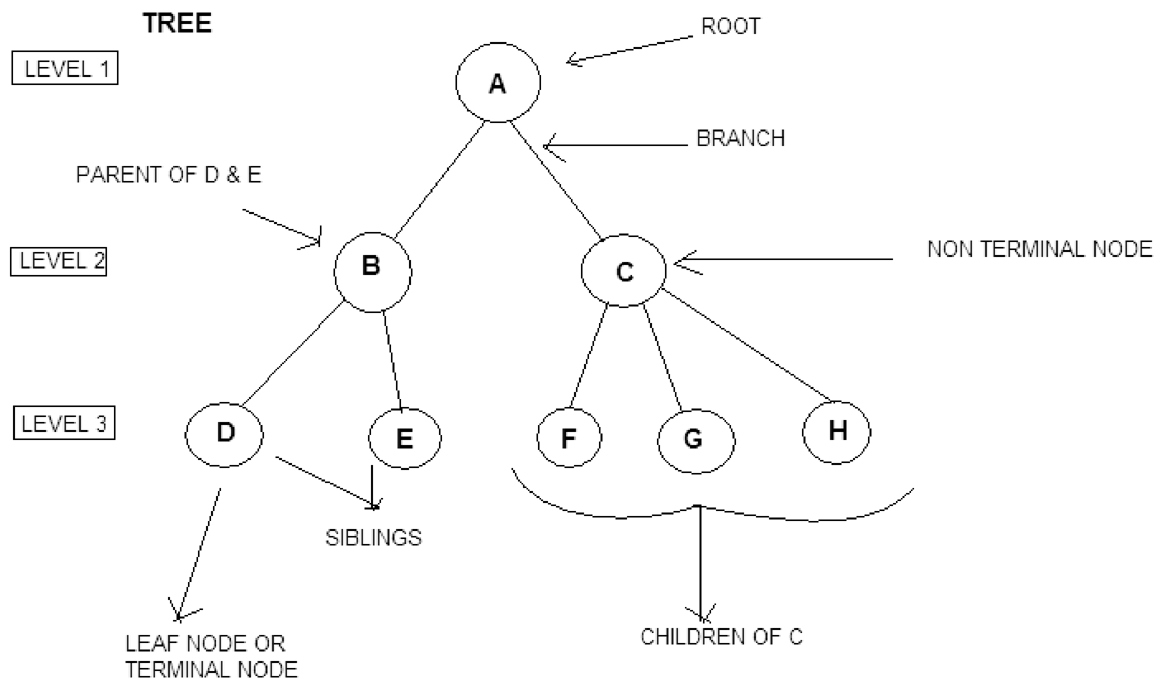
Ancestors: *Ancestors of a node are all the nodes along the path from root to that node. Hence root is ancestor of all the nodes in the tree.*

Level: *Level of a node is defined by letting root at level one. If a node is at level L, then its children are at level L + 1.*

Height or depth: *The height or depth of a tree is defined to be the maximum level of any node in the tree.*

Climbing: *The process of traversing the tree from the leaf to the root is called climbing the tree.*

Descending: *The process of traversing the tree from the root to the leaf is called descending the tree.*

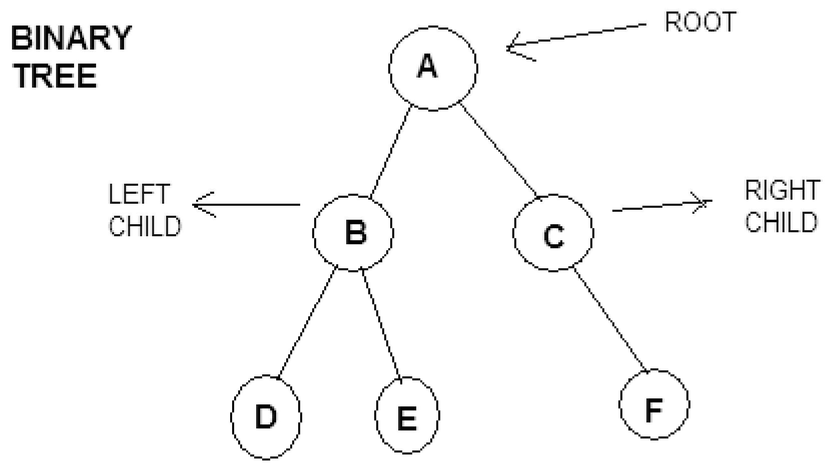


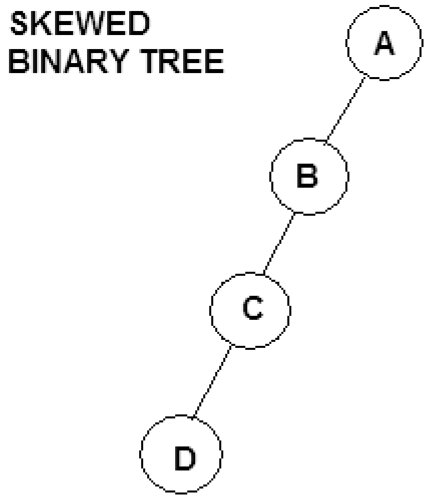
BINARY TREE

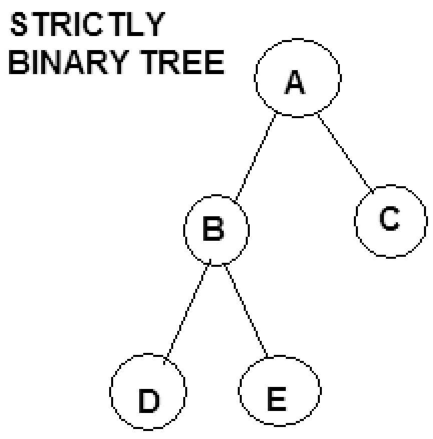
Binary tree has nodes each of which has no more than two child nodes.

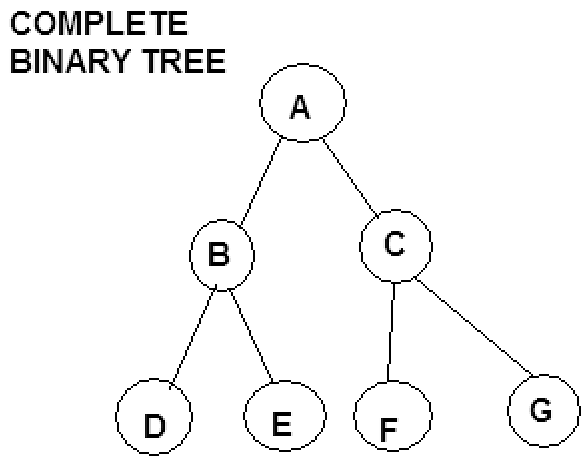
Binary tree: *A binary tree is a finite set of nodes that either is empty or consists of a root and two disjoint binary trees called the left subtree and right subtree.*

Left child: *The node present to the left of the parent node is called the left child.*
Right child: *The node present to the right of the parent node is called the right child.*


BINARY
TREE
ROOT
A
LEFT
CHILD
B
C
RIGHT
CHILD
D
E
F


SKEWED
BINARY TREE
A
B
C
D


STRICTLY
BINARY TREE
A
B
C
D
E


COMPLETE
BINARY TREE
A
B
C
D
E
F
G

Skewed Binary tree: *If the new nodes in the tree are added only to one side of the binary tree then it is a skewed binary tree.*

Strictly binary tree: *If the binary tree has each node consisting of either two nodes or no nodes at all, then it is called a strictly binary tree.*

Complete binary tree: *If all the nodes of a binary tree consist of two nodes each and the nodes at the last level does not consist any nodes, then that type of binary tree is called a complete binary tree.*

It can be observed that the maximum number of nodes on level i of a binary tree is $\mathbf{2^{i-1}}$, where $i \geq 1$. The maximum number of nodes in a binary tree of depth k is $\mathbf{2^k - 1}$, where $k \geq 1$.

PROPERTIES OF BINARY TREES

- The number of nodes n in a full binary tree, is at least $n = 2h+1$ and at most $n = 2^{h+1}-1$, where h is the height of the tree.
- A binary tree of n elements has n-1 edges.
- A binary tree of height h has at least h and at most 2h - 1 elements
- A tree consisting of only a root node has a height of 0.
- The number of leaf nodes in a perfect binary tree, is $l = (n+1)/2$. This means that a perfect binary tree with l leaves has $n=2l-1$ nodes.
- The number of internal nodes in a **complete** binary tree of n nodes is $n/2$.

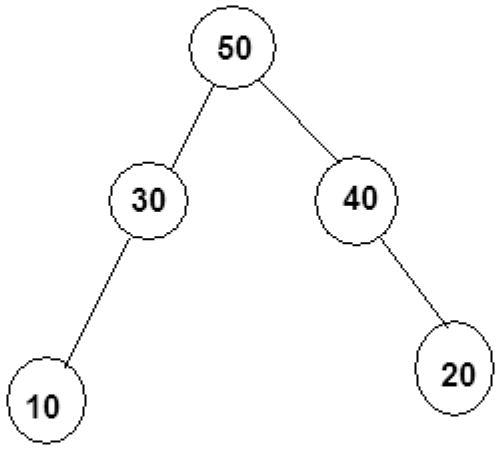
REPRESENTATION OF BINARY TREES

There are two ways in which a binary tree can be represented. They are:

(i) *Array representation of binary trees.*
(ii) *Linked representation of binary trees.*

ARRAY REPRESENTATION OF BINARY TREES

When arrays are used to represent the binary trees, then an array of size 2^k is declared where, k is the depth of the tree. For example if the depth of the binary tree is 3, then maximum $2^3 - 1 = 7$ elements will be present in the node and hence the array size will be 8. This is because the elements are stored from position one leaving the position 0 vacant. But generally an array of bigger size is declared so that later new nodes can be added to the existing tree. The following binary tree can be represented using arrays as shown.



Array representation:

50	30	40	10	-1	-1	20
1	2	3	4	5	6	7

The root element is always stored in position 1. The left child of node i is stored in position 2i and right child of node is stored in position 2i + 1. Hence the following formulae can be used to identify the parent, left child and right child of a particular node.

Parent(i) = i / 2, if $i \neq 1$. If i = 1 then i is the root node and root does not has parent.

Left child(i) = 2i, if $2i \leq n$, where n is the maximum number of elements in the tree. If 2i > n, then i has no left child.

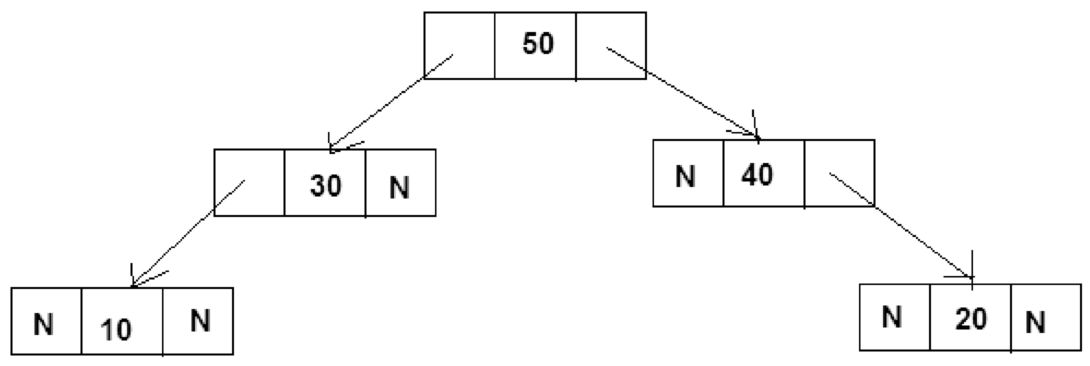
Right child(i) = 2i + 1, if $2i + 1 \leq n$. If 2i + 1 > n, then i has no right child.

The empty positions in the tree where no node is connected are represented in the array using -1, indicating absence of a node.

Using the formula, we can see that for a node 3, the parent is 3/ 2 →1. Referring to the array locations, we find that 50 is the parent of 40. The left child of node 3 is 2*3 → 6. But the position 6 consists of -1 indicating that the left child does not exist for the node 3. Hence 50 does not have a left child. The right child of node 3 is 2*3 + 1 → 7. The position 7 in the array consists of 20. Hence, 20 is the right child of 40.

LINKED REPRESENTATION OF BINARY TREES

In linked representation of binary trees, instead of arrays, pointers are used to connect the various nodes of the tree. Hence each node of the binary tree consists of three parts namely, the info, left and right. The info part stores the data, left part stores the address of the left child and the right part stores the address of the right child. Logically the binary tree in linked form can be represented as shown.



The pointers storing NULL value indicates that there is no node attached to it. Traversing through this type of representation is very easy. The left child of a particular node can be accessed by following the left link of that node and the right child of a particular node can be accessed by following the right link of that node.

BINARY TREE ADT REPRESENTATIONS:

Abstract Data Type (ADT): *An Abstract Data Type is a representation in which we provide a specification of the instances as well as of the operations that are to be performed. More precisely,*

An Abstract Data Type is a data type that is organized in such a way that the specification of the object and the specification of the operation on the object are separated from the representation of the object and the implementation of the operation.

Abstract Datatype BinT(node *root)
{
Instances: Binarytree is a non linear data structure which contains every node except the leaf nodes at most two child nodes.

Operations:
1.Insertion:
This operations is used to insert the nodes in the binary tree.
2.Deletion:
This operations is used to delete any node from the binary tree.Note that if root node is removed the tree becomes empty.
}

BINARY TREE TRAVERSALS

There are three standard ways of traversing a binary tree T with root R. They are:

(i) Preorder Traversal
(ii) Inorder Traversal
(iii) Postorder Traversal

General outline of these three traversal methods can be given as follows:

Preorder Traversal:

(1) Process the root R.
(2) Traverse the left subtree of R in preorder.
(3) Traverse the right subtree of R in preorder.

Inorder Traversal:

(1) Traverse the left subtree of R in inorder.
(2) Process the root R.
(3) Traverse the right subtree of R in inorder.

Postorder Traversal:

(1) Traverse the left subtree of R in postorder.
(2) Traverse the right subtree of R in postorder.
(3) Process the root R.

Observe that each algorithm contains the same three steps, and that the left subtree of R is always traversed before the right subtree. The difference between the algorithms is the time at which the root R is processed. The three algorithms are sometimes called, respectively, the node-left-right (NLR) traversal, the left-node-right (LNR) traversal and the left-right-node (LRN) traversal.

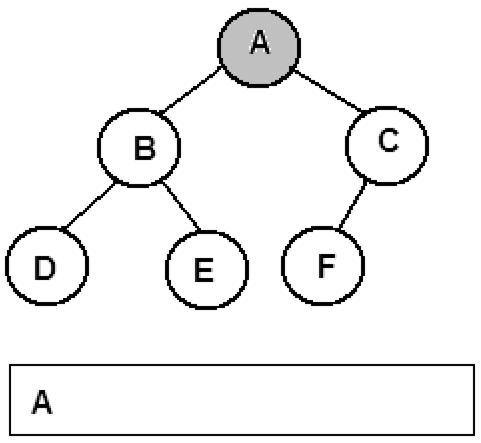
Now we can present the detailed algorithm for these traversal methods in both recursive method and iterative method.

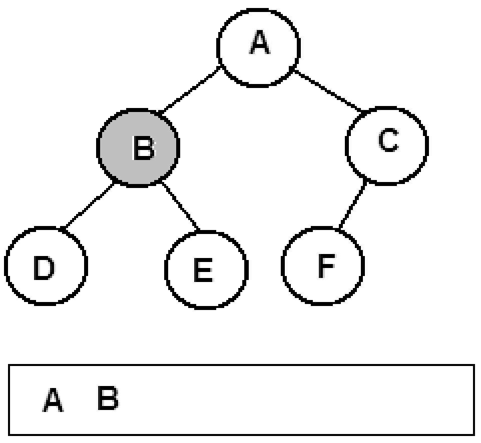
Traversal algorithms using recursive approach

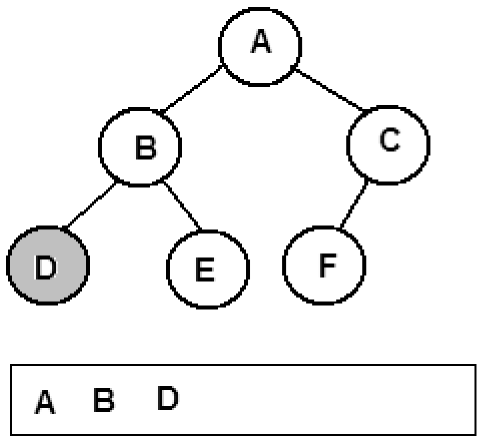
Preorder Traversal

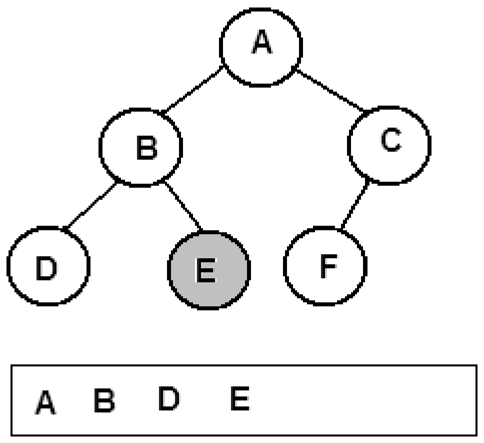
In the preorder traversal the node element is visited first and then the right subtree of the node and then the right subtree of the node is visited. Consider the following case where we have 6 nodes in the tree A, B, C, D, E, F. The traversal always starts from the root of the tree. The node A is the root and hence it is visited first. The value at this node is processed. The processing can be doing some computation over it or just printing its value. Now we check if there exists any left child for this node if so apply the preorder procedure on the left subtree. Now check if there is any right subtree for the node A, the preorder procedure is applied on the right subtree.

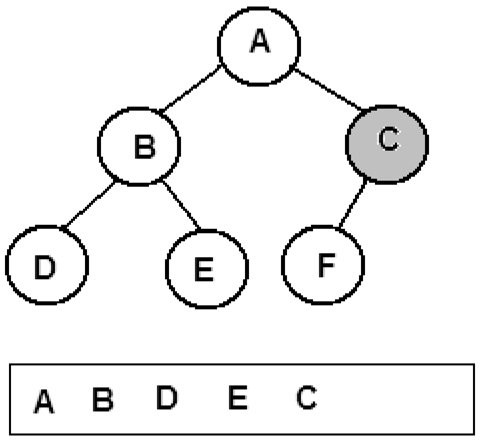
Since there exists a left subtree for node A, B is now considered as the root of the left subtree of A and preorder procedure is applied. Hence we find that B is processed next and then it is checked if B has a left subtree. This recursive method is continued until all the nodes are visited.

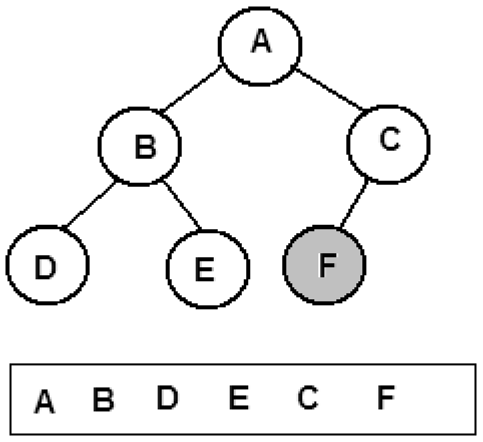












The algorithm for the above method is presented in the pseudo-code form below:

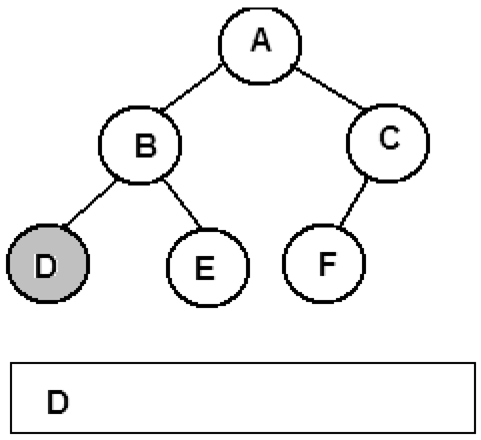
Algorithm

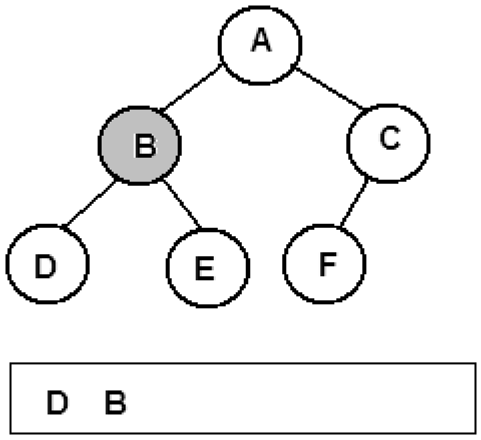
```
PREORDER( ROOT )

Temp = ROOT
If temp = NULL
        Return
End if
```

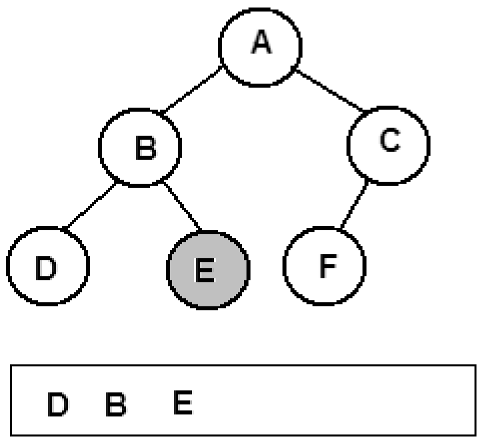
```
Print info(temp)
If left(temp) $\neq$ NULL
        PREORDER( left(temp))
End if
If right(temp) $\neq$ NULL
        PREORDER(right(temp))
End if
End PREORDER
```

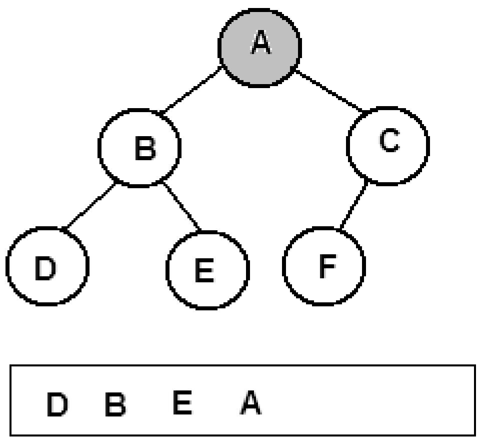
Inorder Traversal

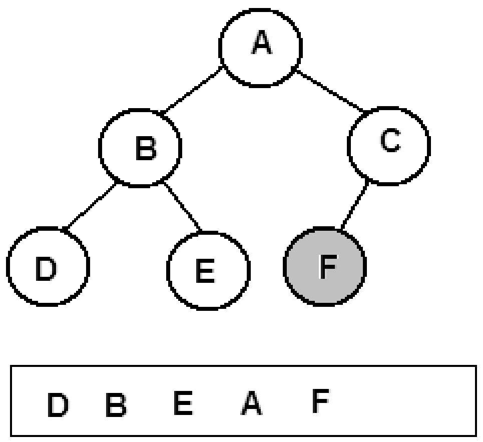
In the Inorder traversal method, the left subtree of the current node is visited first and then the current node is processed and at last the right subtree of the current node is visited. In the following example, the traversal starts with the root of the binary tree. The node A is the root and it is checked if it has the left subtree. Then the inorder traversal procedure is applied on the left subtree of the node A. Now we find that node D does not have left subtree. Hence the node D is processed and then it is checked if there is a right subtree for node D. Since there is no right subtree, the control returns back to the previous function which was applied on B. Since left of B is already visited, now B is processed. It is checked if B has the right subtree. If so apply the inroder traversal method on the right subtree of the node B. This recursive procedure is followed till all the nodes are visited.

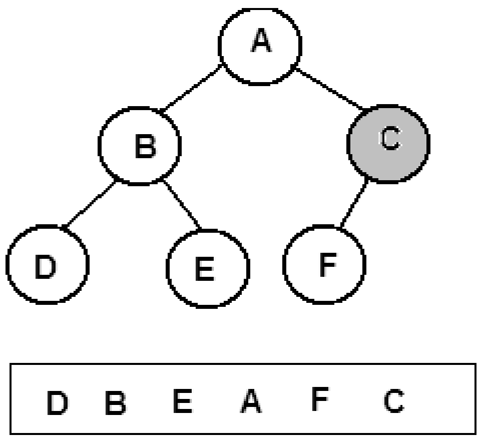










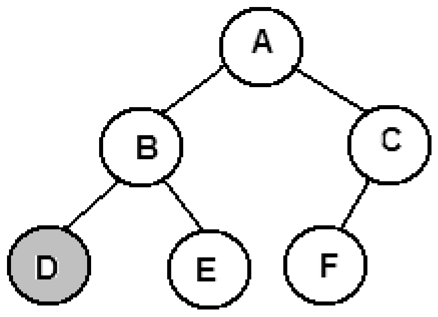


Algorithm

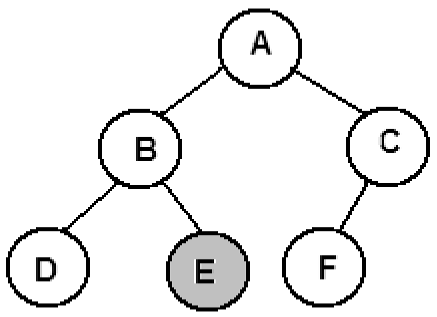
```
INORDER( ROOT )

Temp = ROOT
If temp = NULL
        Return
End if
If left(temp) ≠ NULL
        INORDER(left(temp))
End if
Print info(temp)
If right(temp) ≠ NULL
        INORDER(right(temp))
End if
End INORDER
```

Postorder Traversal

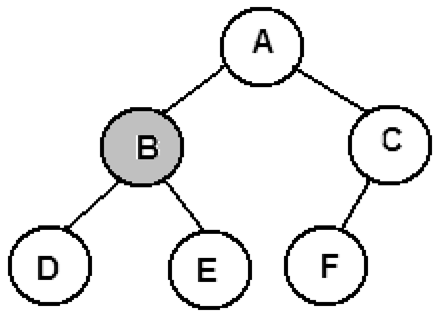
In the postorder traversal method the left subtree is visited first, then the right subtree and at last the current node is processed. In the following example, A is the root node. Since A has the left subtree the postorder traversal method is applied recursively on the left subtree of A. Then when left subtree of A is completely is processed, the postorder traversal method is recursively applied on the right subtree of the node A. If right subtree is completely processed, then the current node A is processed.



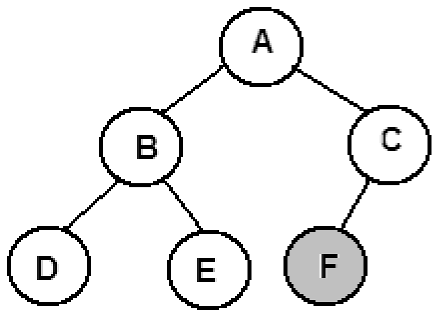
D



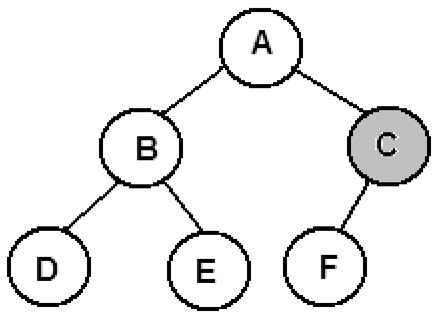
D E



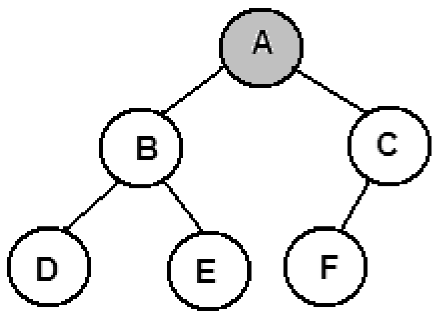
D E B



D E B F



D E B F C



D E B F C A

Algorithm

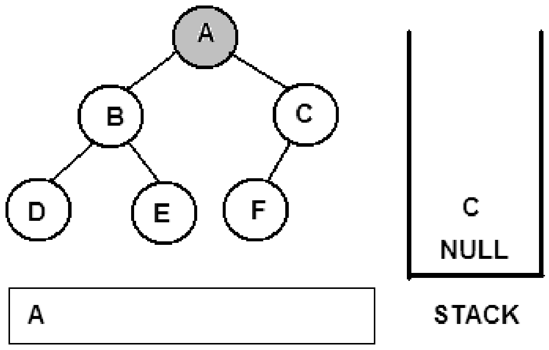
```
POSTORDER( ROOT )

Temp = ROOT
If temp = NULL
        Return
End if
If left(temp) ≠ NULL
        POSTORDER(left(temp))
```

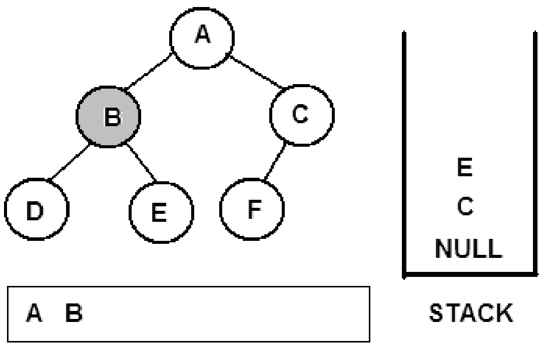
```
End if
If right(temp) ≠ NULL
        POSTORDER(right(temp))
End if
Print info(temp)
End POSTORDER
```

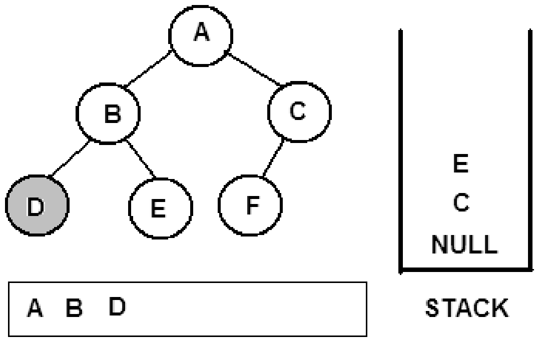
Binary Tree Traversal Using Iterative Approach

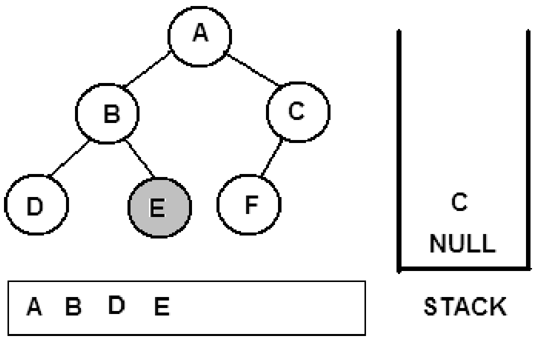
Preorder Traversal

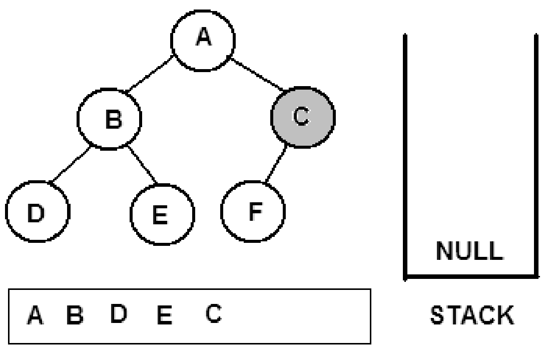
In the iterative method a stack is used to implement the traversal methods. Initially the stack is stored with a NULL value. The root node is taken for processing first. A pointer temp is made to point to this root node. If there exists a right node for the current node, then push that node into the stack. If there exists a left subtree for the current node then temp is made to the left child of the current node. If the left child does not exist, then a value is popped from the stack and temp is made to point to that node which is popped and the same process is repeated. This is done till the NULL value is popped from the stack.

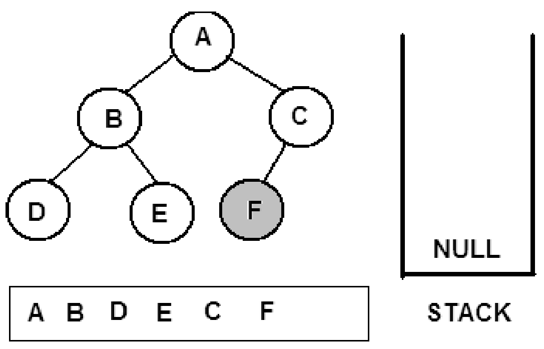












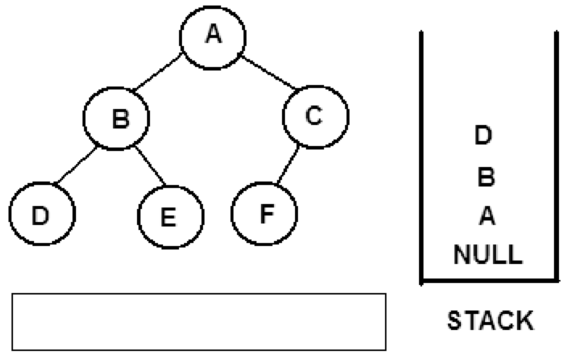
Algorithm

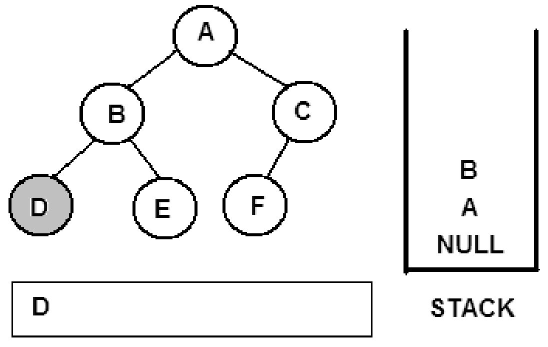
```
PREORDER( ROOT )

Temp = ROOT,  push(NULL)
While temp ≠ NULL
    Print info(temp)
    If right(temp) ≠ NULL
        Push(right(temp))
    End if
    If left(temp) ≠ NULL
        Temp = left(temp)
    Else
        Temp = pop( )
    End if
End while
End PREORDER
```

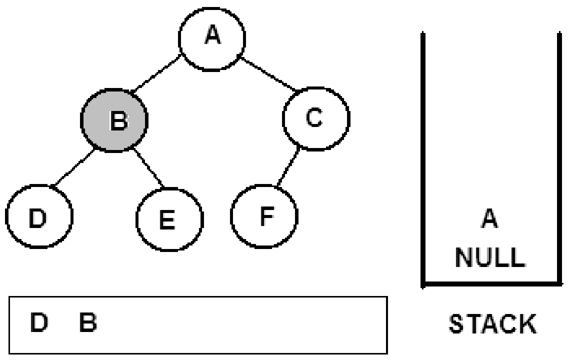
Inorder Traversal

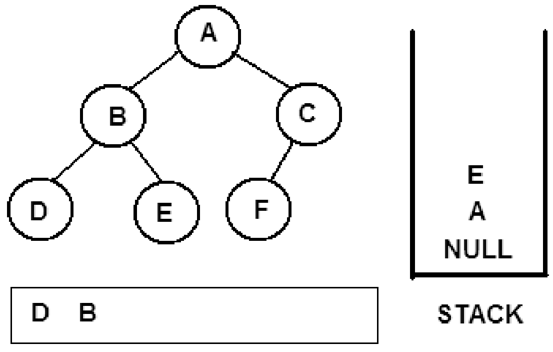
In the Inorder traversal method, the traversal starts at the root node. A pointer Temp is made to point to root node. Initially, the stack is stored with a NULL value and a flag RIGHTEXISTS is made equal to 1. Now for the current node, if the flag RIGHTEXISTS = 1, then immediately it is made 0, and the node pointed by temp is pushed to the stack. The temp pointer is moved to the left child of the node if the left child exists. Every time the temp is moved to a new node, the node is pushed into the stack and temp is moved to its left child. This is continued till temp reaches a NULL value.

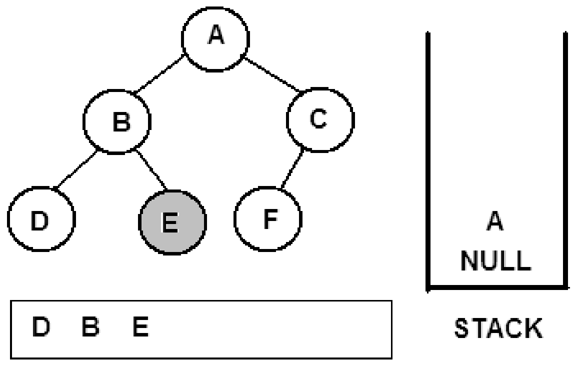
After this one by one, the nodes in the stack are popped and are pointed by temp. The node is processed and if the node has right child, then the flag RIGHTEXISTS is set to 1 and the process describe above starts from the beginning. Thus the process stops when the NULL value from the stack is popped.

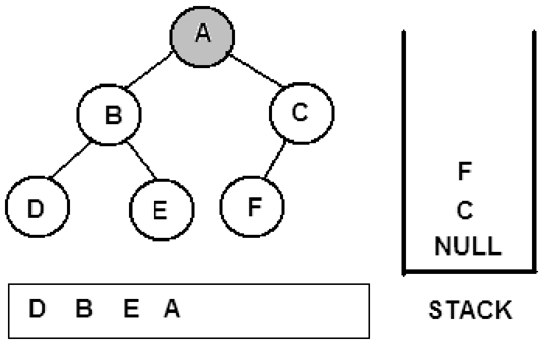


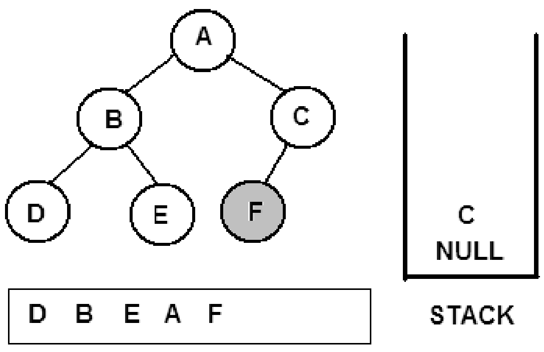


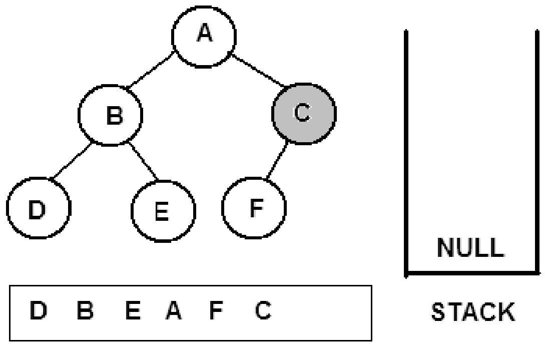












Algorithm

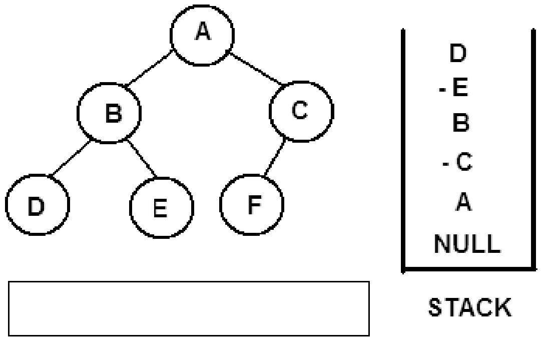
INORDER(ROOT)

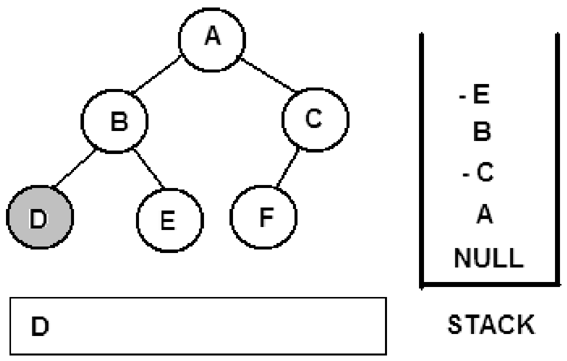
```
Temp = ROOT,  push(NULL),  RIGHTEXISTS = 1
While RIGHTEXISTS = 1
    RIGHTEXISTS = 0
    While temp $\neq$ NULL
        Push(temp)
        Temp = left(temp)
    End while

    While (TRUE)
        Temp = pop( )
        If temp = NULL
            Break
        End if
        Print info(temp)
        If right(temp) $\neq$ NULL
            Temp = right(temp)
            RIGHTEXISTS = 1
            Break
        End if
    End while
End while
End INORDER
```

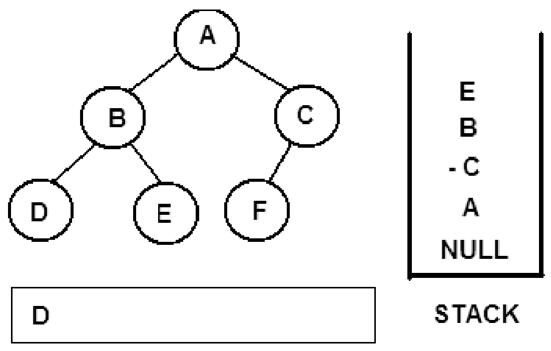
Postorder Traversal

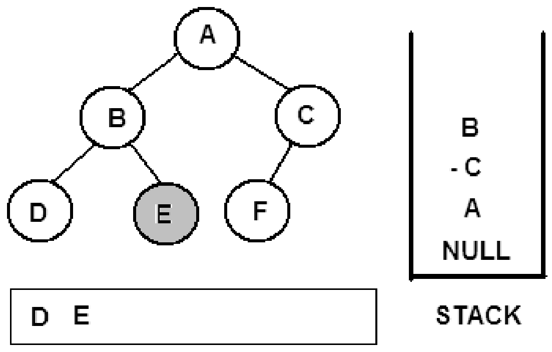
In the postorder traversal method, a stack is initially stored with a NULL value. A pointer temp is made to point to the root node. A flag RIGHTEXISTS is set to 1. A loop is started and continued until this flag is 1. The current node is pushed into the stack and it is checked if it has a right child. If so, the negative of value of that node is pushed into the stack and the temp is moved to its left child if it exists. This process is repeated till the temp reached a NULL value.

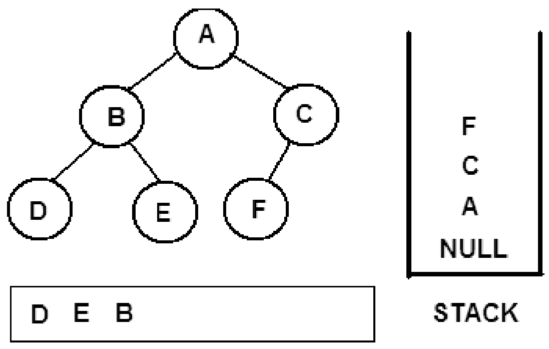
Now the values in the stack are popped one by one and are pointed by temp. If the value popped is positive then that node is processed. If the value popped is negative, then the value is negated and pointed by temp. The flag RIGHTEXISTS is set to 1 and the same above process repeats. This continues till the NULL value from the stack is popped.

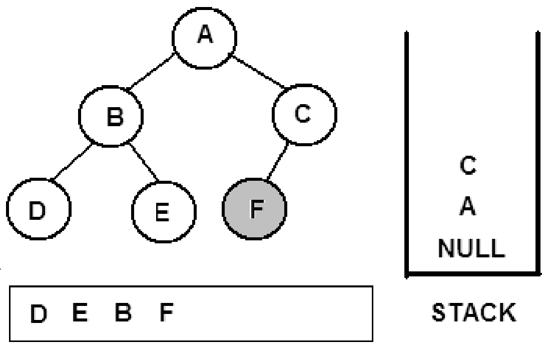


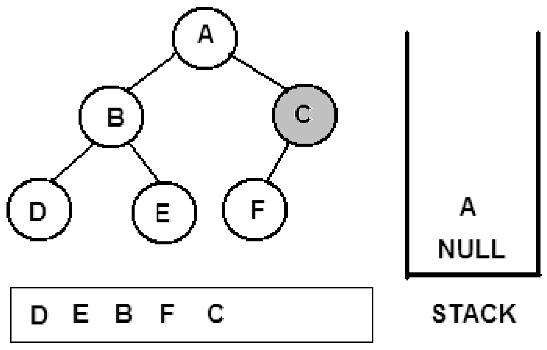


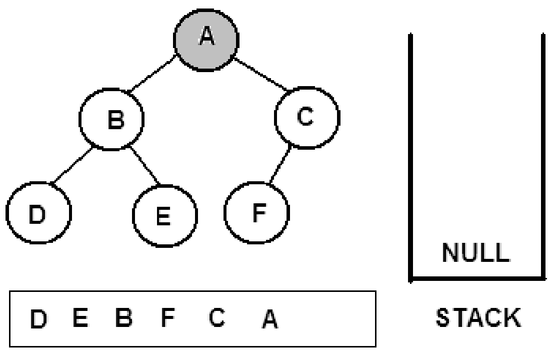












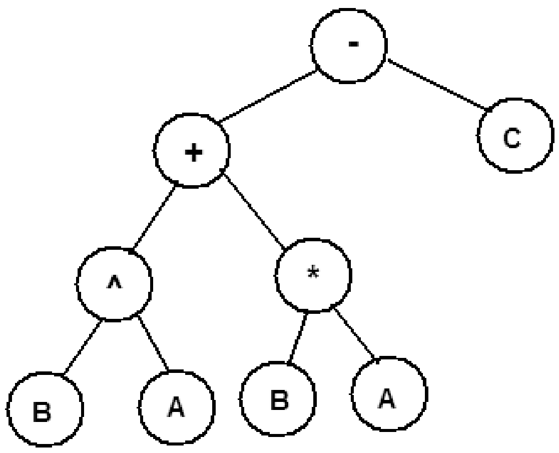
Algorithm

```
POSTORDER( ROOT )

Temp = ROOT,  push(NULL),  RIGHTEXISTS = 1
While RIGHTEXISTS = 1
        RIGHTEXISTS = 0
        While temp ≠ NULL
                Push (temp)
                If right(temp) ≠ NULL
                        Push( - right(temp))
                End if
                Temp =left(temp)
        End while
        Do
                Temp = pop( )
                If temp > 0
                        Print info(temp)
                End if
                If temp < 0
                        Temp = -temp
                        RIGHTEXISTS = 1
                        Break
                End if
        While temp ≠ NULL
End while
End POSTORDER
```

Expression Tree

The trees are many times used to represent an expression and if done so, those types of trees are called expression trees. The following expression is represented using the binary tree, where the leaves represent the operands and the internal nodes represent the operators.

B ^ A + B * A - C



If the expression tree is traversed using preorder, inorder and postorder traversal methods, then we get the expressions in prefix, infix and postfix forms as shown.

- + ^ B A * B A - C

B ^ A + B * A - C

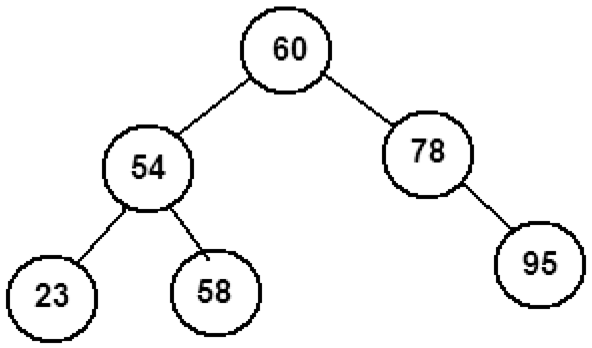
B A ^ B A * C –

BINARY SEARCH TREES

Binary Search Tree: *A Binary tree T is a Binary Search Tree (BST), 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.*

Consider the following tree. The root node 60 is greater than all the elements (54, 23, 58) in its left subtree and is less than all elements in its right subtree (78, 95). Similarly, 54 is greater than its left child 23 and lesser than its right child 58. Hence each and every node in a binary search tree satisfies this property.

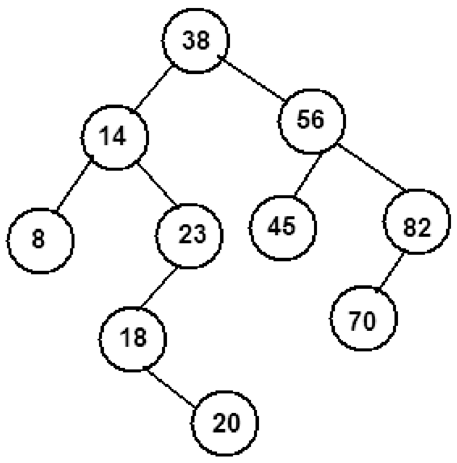
The reason why we go for a Binary Search tree is to improve the searching efficiency. The average case time complexity of the search operation in a binary search tree is O(log n).



Consider the following list of numbers. A binary tree can be constructed using this list of numbers, as shown.

38 14 8 23 18 20 56 45 82 70

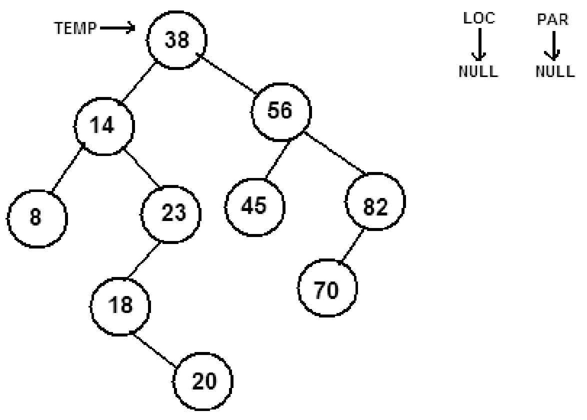
Initially 38 is taken and placed as the root node. The next number 14 is taken and compared with 38. As 14 is lesser than 38, it is placed as the left child of 38. Now the third number 8 is taken and compared starting from the root node 38. Since is 8 is less than 38 move towards left of 38. Now 8 is compared with 14, and as it is less that 14 and also 14 does not have any child, 8 is attached as the left child of 14. This process is repeated until all the numbers are inserted into the tree. Remember that if a number to be inserted is greater than a particular node element, then we move towards the right of the node and start comparing again.

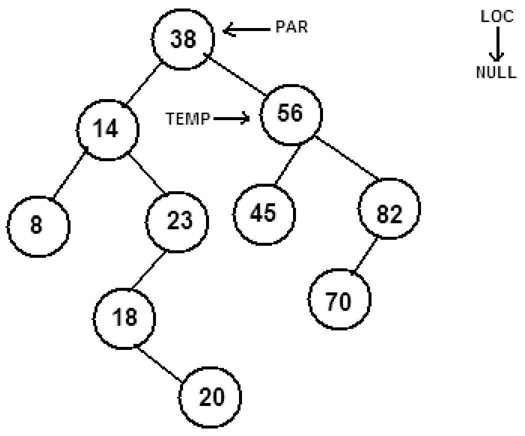


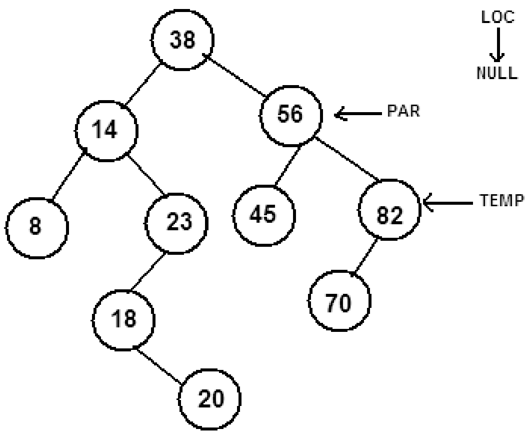
Search Operation in a Binary Search Tree

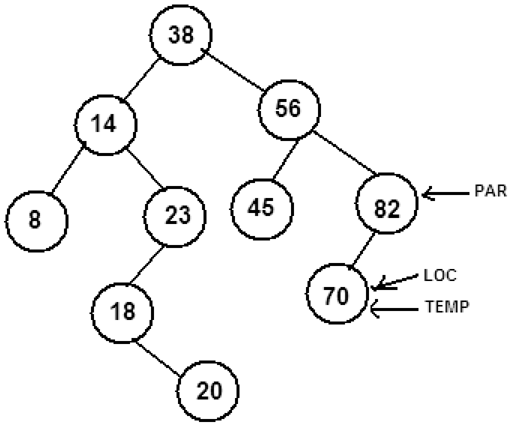
The search operation on a BST returns the address of the node where the element is found. The pointer LOC is used to store the address of the node where the element is found. The pointer PAR is used to point to the parent of LOC. Initially the pointer TEMP is made to point to the root node. Let us search for a value 70 in the following BST. Let k = 70. The k value is compared with 38. As k is greater that 38, move to the right child of 38, i.e., 56. k is greater than 56 and hence we move to the right child of 56, which is 82. Now since k is lesser than 82, temp is moved to the left child of 82. The k value matches here and hence the address of this node is stored in the pointer LOC.

Every time the temp pointer is moved to the next node, the current node is made pointed by PAR. Hence we get the address of that node where the k value is found, and also the address of its parent node though PAR.









Algorithm

```
SEARCH( ROOT, k )

Temp = ROOT,  par = NULL, loc = NULL
```

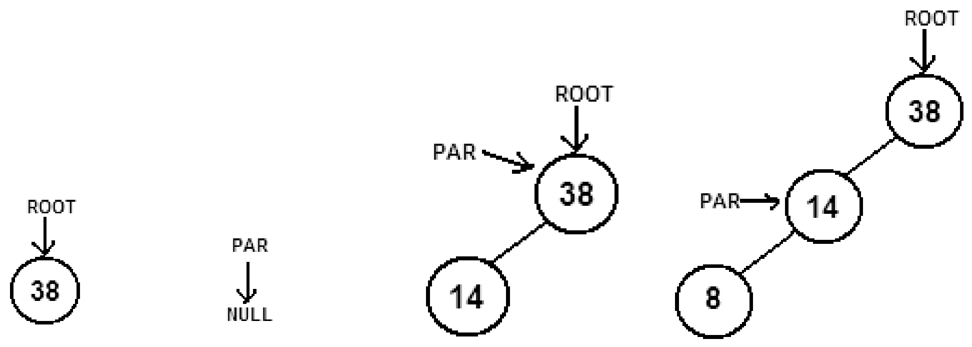
```
While temp ≠ NULL
        If k = info(temp)
                Loc = temp
                Break
        End if
        If k < info(temp)
                Par = temp
                Temp = left(temp)
        Else
                Par = temp
                Temp = right(temp)
        End if
End while
End SEARCH
```

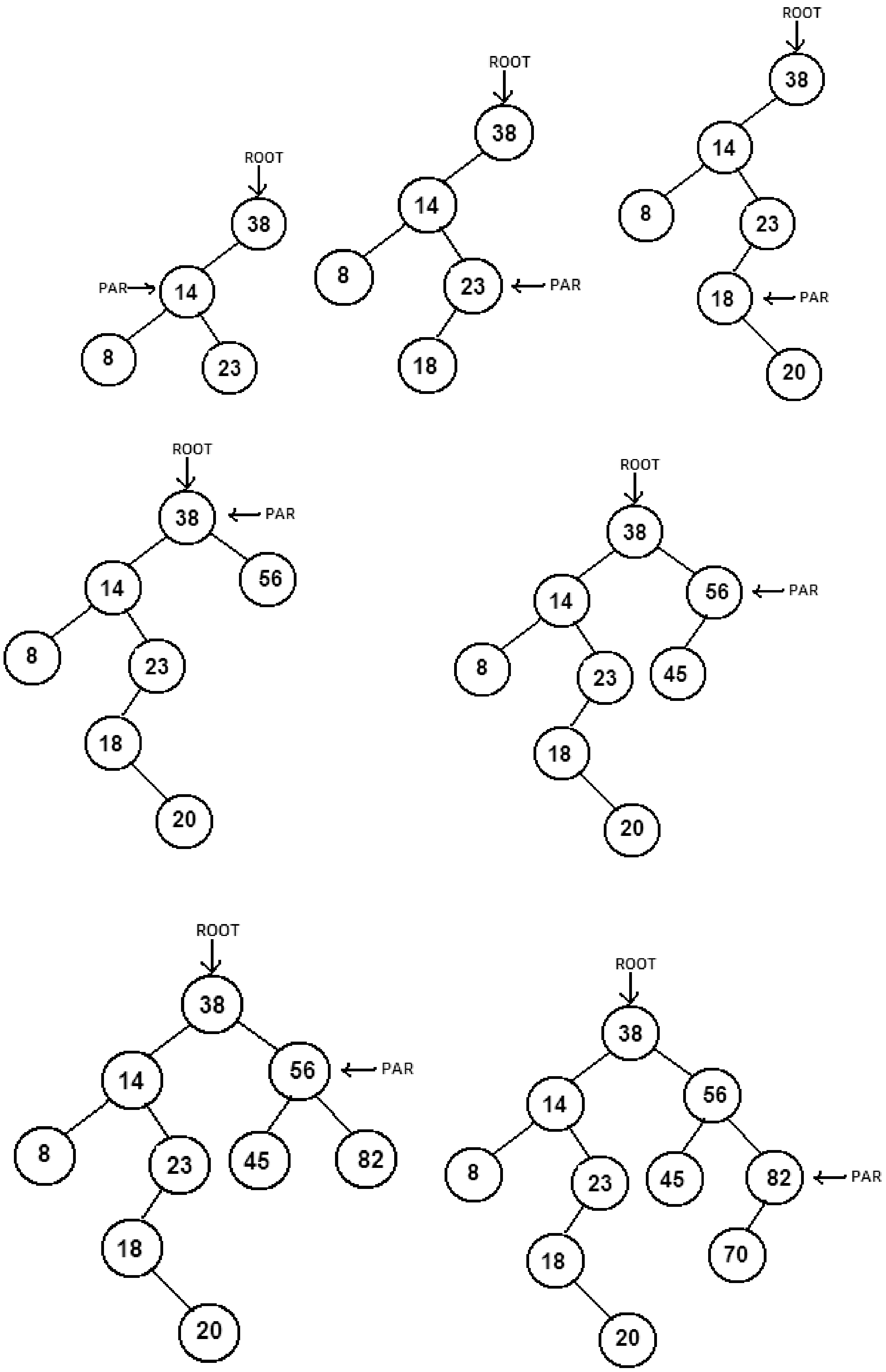
Insert Operation in a Binary Search Tree

The BST itself is constructed using the insert operation described below. Consider the following list of numbers. A binary tree can be constructed using this list of numbers using the insert operation, as shown.

39 14 8 23 18 20 56 45 82 70

Initially 38 is taken and placed as the root node. The next number 14 is taken and compared with 38. As 14 is lesser than 38, it is placed as the left child of 38. Now the third number 8 is taken and compared starting from the root node 38. Since is 8 is less than 38 move towards left of 38. Now 8 is compared with 14, and as it is less that 14 and also 14 does not have any child, 8 is attached as the left child of 14. This process is repeated until all the numbers are inserted into the tree. Remember that if a number to be inserted is greater than a particular node element, then we move towards the right of the node and start comparing again.





Algorithm

```
INSERT( ROOT, k )

Temp = ROOT,  par = NULL
While temp ≠ NULL
    If k = info(temp)
        Print "Item already exists!"
        Return
    End if
    If k < info(temp)
        Par = temp
        Temp = left(temp)
    Else
        Par = temp
        Temp = right(temp)
    End if
End while

Info(R) = k, left(R) = NULL, right(R) = NULL
If par = NULL
    ROOT = R
End if
If k < info(par)
    Left(par) = R
Else
    Right(par) = R
End if
End INSERT
```

Delete Operation in a Binary Search Tree

The delete operation in a Binary search tree follows two cases. In case A, the node to be deleted has no children or it has only one child. In case B, the node to be deleted has both left child and the right child. It is taken care that, even after deletion the binary search tree property holds for all the nodes in the BST.

Algorithm

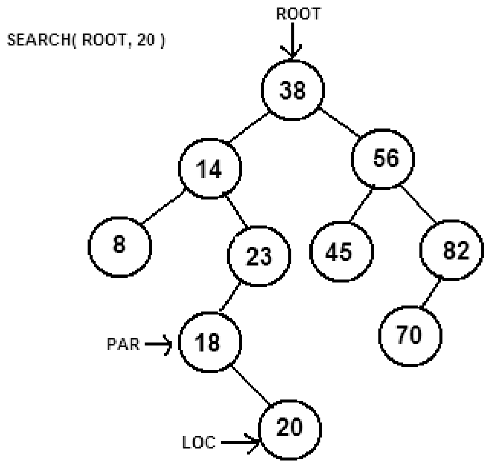
```
DELETE( ROOT, k )

SEARCH( ROOT, k )
If Loc  = NULL
    Print "Item not found"
    Return
End if

If right(Loc) ≠ NULL and left(Loc) ≠ NULL
    CASEB(Loc, par)
Else
```

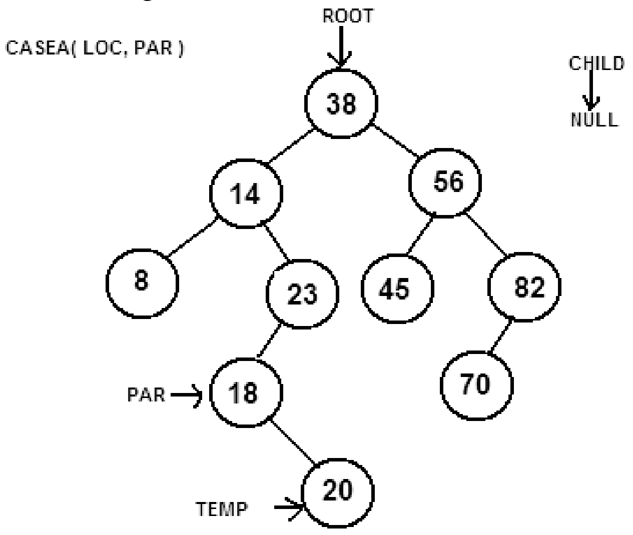
```
        CASEA(Loc, par)
End if

End DELETE
```

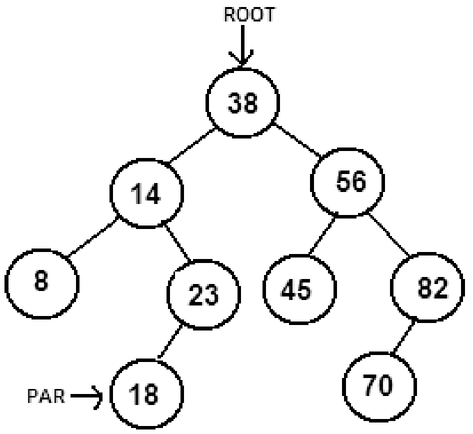
Case A:

The search is operation is performed for the key value that has to be deleted. The search operation, returns the address of the node to be deleted in the pointer LOC and its parents' address is returned in a pointer PAR. If the node to be deleted has no children then, it is checked whether the node pointed by LOC is left child of PAR or is it the right child of PAR. If it is the left child of PAR, then left of PAR is made NULL else right of PAR is made NULL.

The sequence of steps followed for deleting 20 from the tree is as shown.

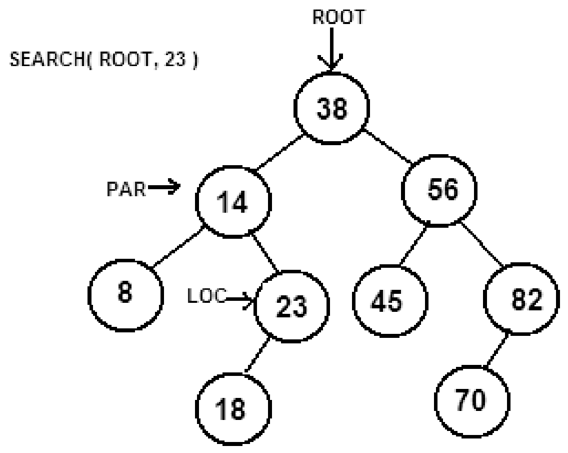


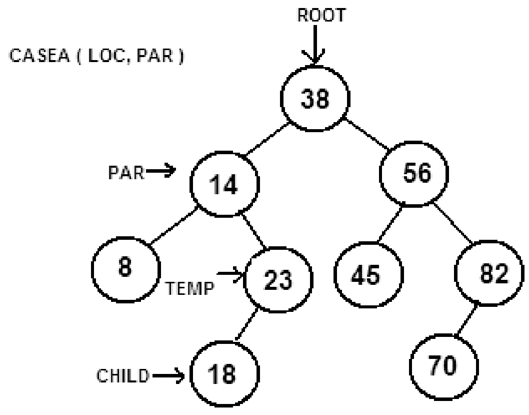


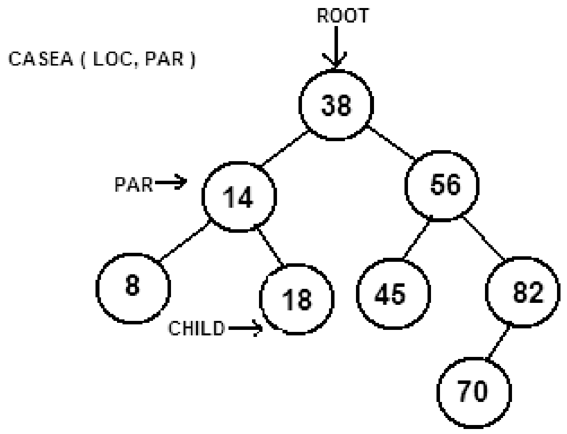


If the node to be deleted has one child, then a new pointer CHILD is made to point to the child of LOC. If LOC is left child of PAR then left of PAR is pointed to CHILD. If LOC is right child of PAR then right of PAR is pointed to CHILD.

The sequence of steps for deleting the node 23 is shown.

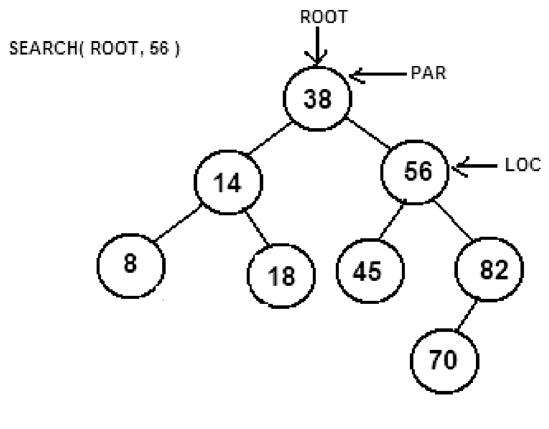


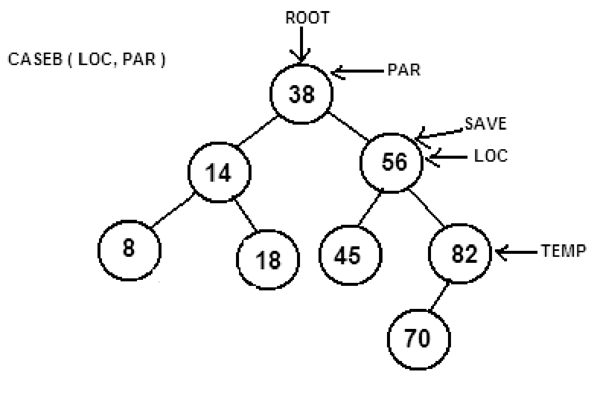




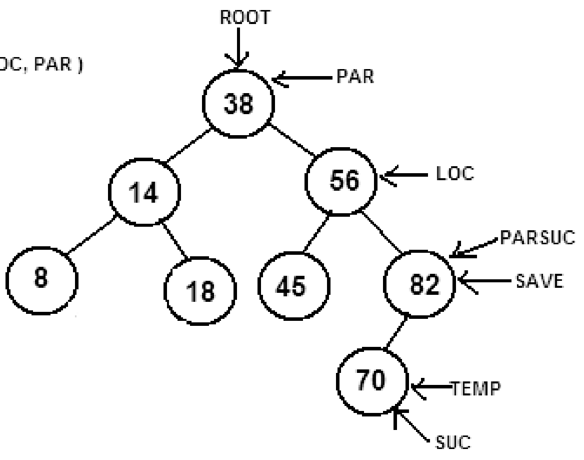
Case B:

In this case, the node to be deleted has both the left child and the right child. Here we introduce two new pointers SUC and PARSUC. The inorder successor of the node to be deleted is found out and is pointed by SUC and its parent node is pointed by the PARSUC. In the following example the node to be deleted is 56 which has both the left child and the right child. The inorder successor of 56 is 70 and hence it is pointed by SUC. Now the SUC replaces 56 as shown in the following sequence of steps.



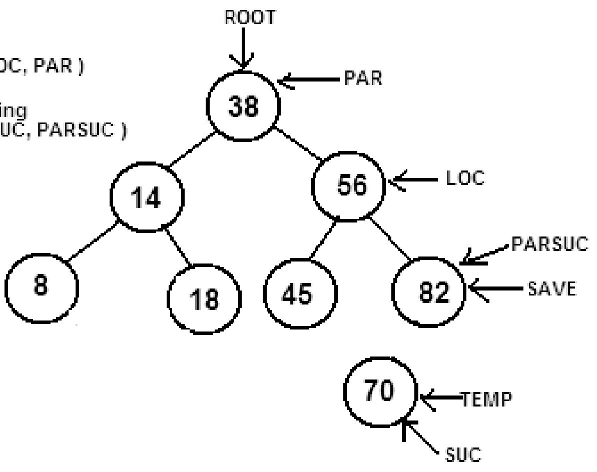


CASEB (LOC, PAR)

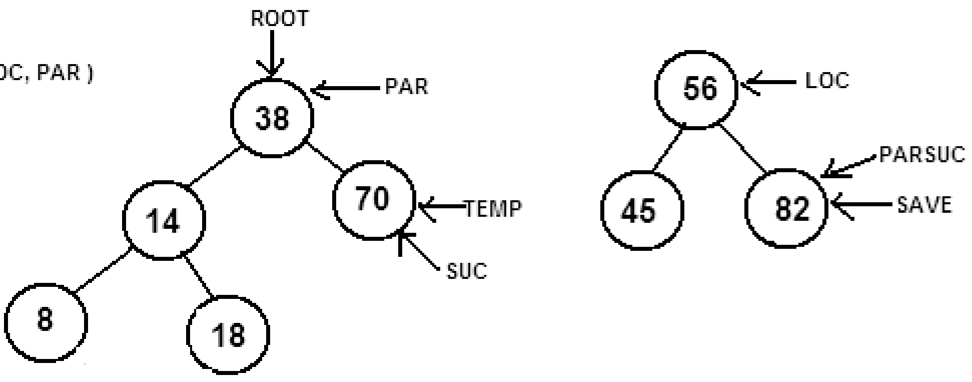


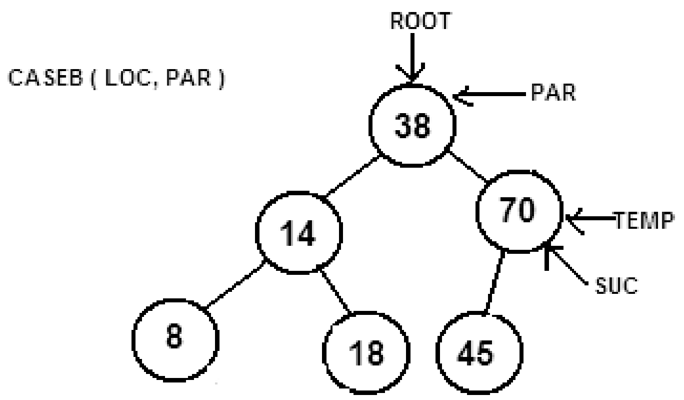
CASEB (LOC, PAR)

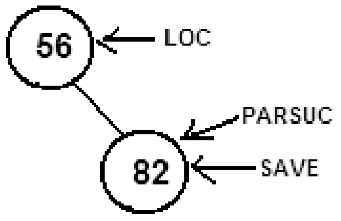
After executing
CASEA (SUC, PARSUC)

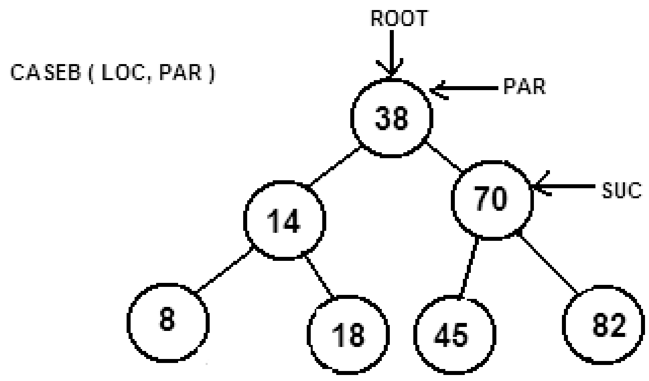


CASEB (LOC, PAR)









Algorithm

```
CASEA( Loc, par )

Temp = Loc
If left(temp) = NULL and right(temp) = NULL
    Child = NULL
Else
    If left(temp) ≠ NULL
        Child = left(temp)
    Else
        Child = right(temp)
    End if
End if

If par ≠ NULL
    If temp = left(par)
        Left(par) = child
    Else
        Right(par) = child
    End if
Else
    ROOT = child
End if
End CASEA
```

CASEB(Loc, par)

```
Temp = right(Loc)
Save = Loc
While left(temp) ≠ NULL
    Save = temp
    Temp = left(temp)
End while

Suc = temp
Parsuc = save
CASEA( suc, parsuc)
If par ≠ NULL
    If Loc = left(par)
        Left(par) = suc
    Else
        Right(par) = suc
    End if
Else
    ROOT = suc
End if
Left(suc) = left(Loc)
Right(suc) = right(Loc)

End CASEB
```