ADVANCED DATA STRUCTURES

UNIT – II

TREES

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

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.*

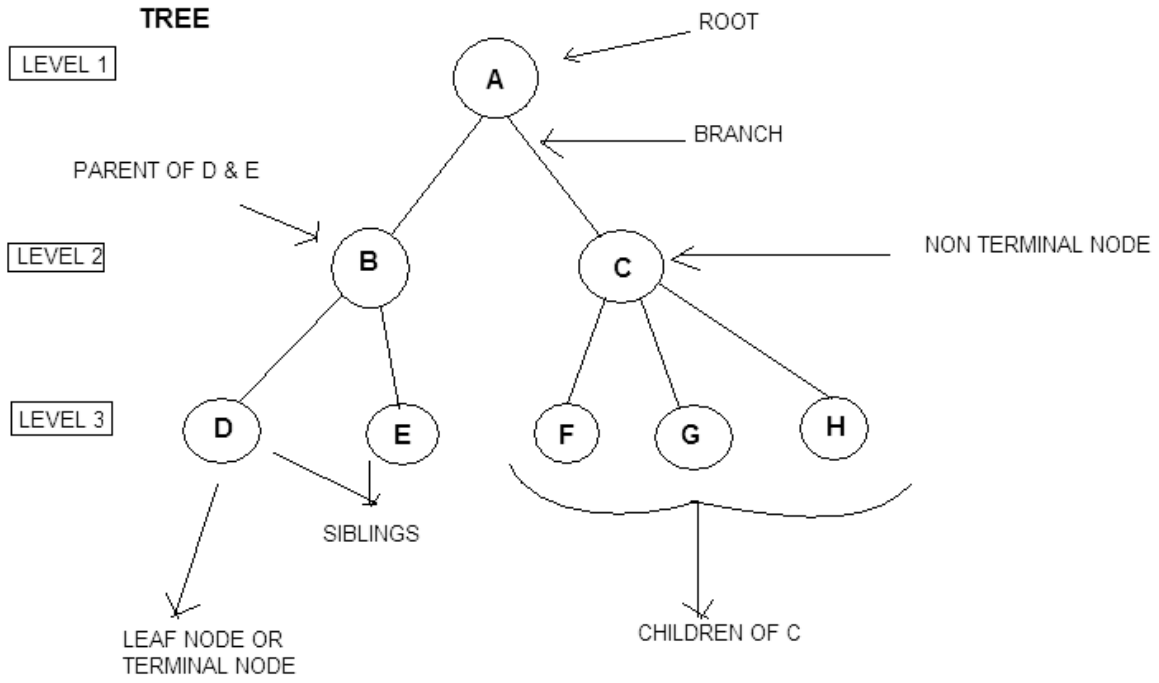
Ancestors: *Ancestors of a node are all the nodes along the path from root to that node. Hence root is ancestor of all the nodes in the tree.*

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.*

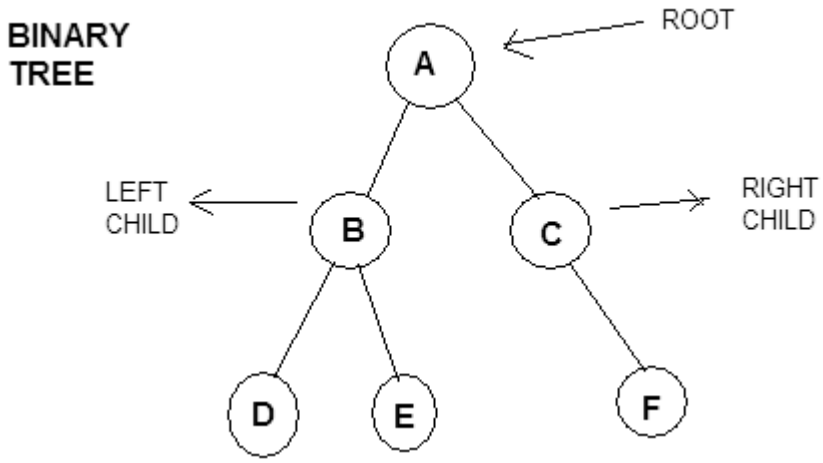


BINARY TREE

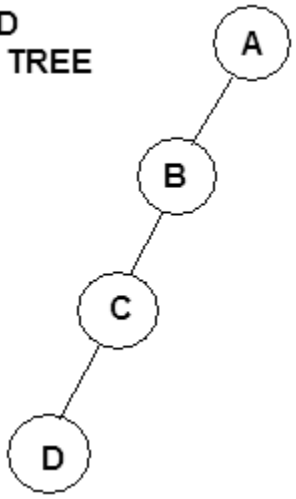
Binary tree has nodes each of which has no more than two child nodes.

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.*

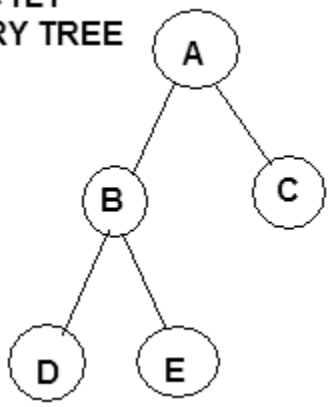
Left child: *The node present to the left of the parent node is called the left child.*
Right child: *The node present to the right of the parent node is called the right child.*



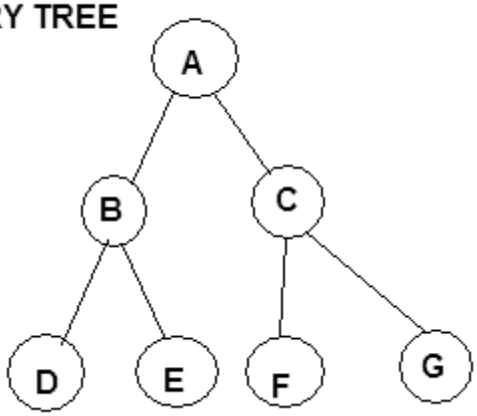
SKEWED BINARY TREE



STRICTLY BINARY TREE



COMPLETE BINARY TREE



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 $k \geq 1$.

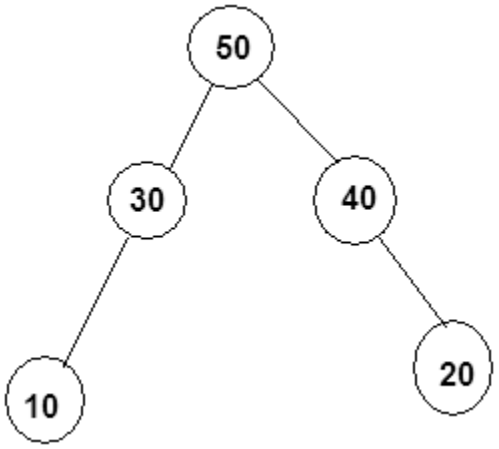
REPRESENTATION OF BINARY TREES

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.

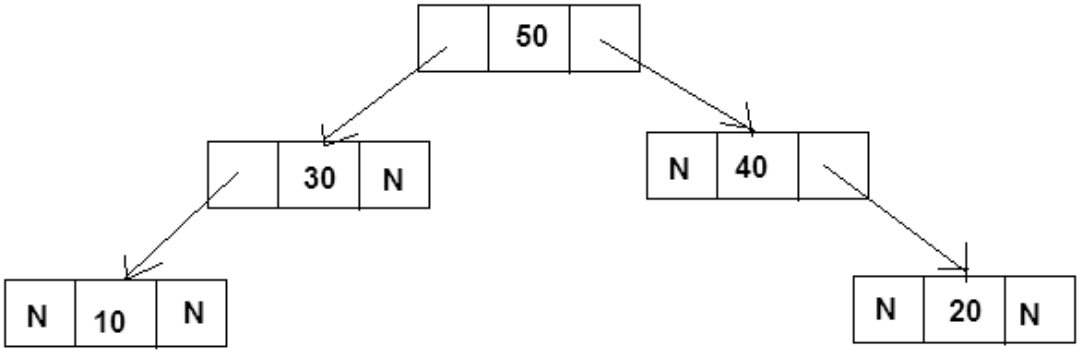
Right child(i) = 2i + 1, if $2i + 1 \leq n$. If 2i + 1 > n, then i has no right child.

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 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.

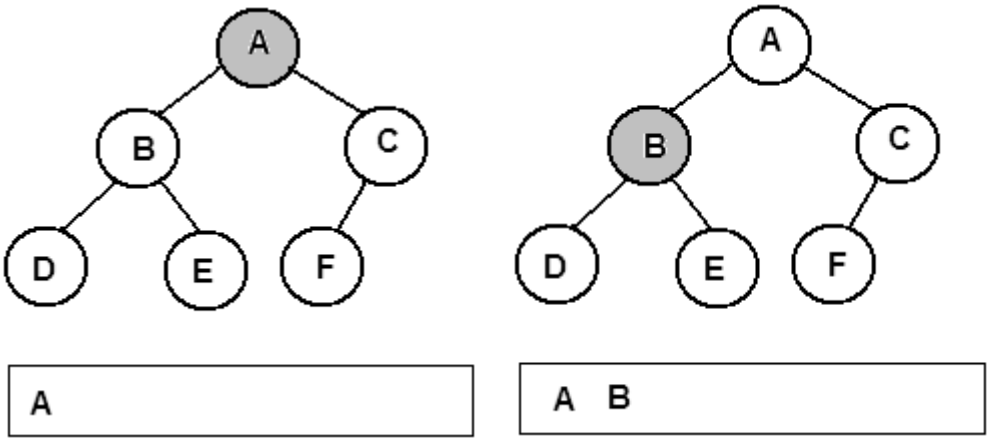
Now we can present the detailed algorithm for these traversal methods in both recursive method and iterative method.

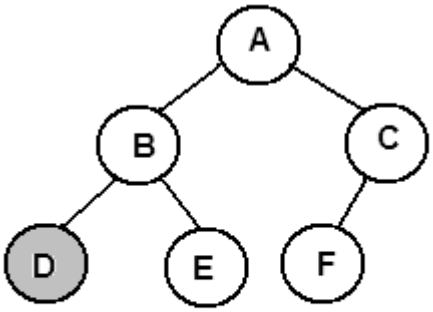
Traversal algorithms using recursive approach

Preorder Traversal

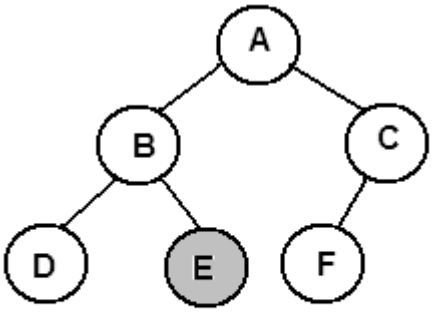
In the preorder traversal the node element is visited first and then the right subtree of the node and then the right subtree of the node is visited. Consider the following case where we have 6 nodes in the tree A, B, C, D, E, F. The traversal always starts from the root of the tree. The node A is the 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.

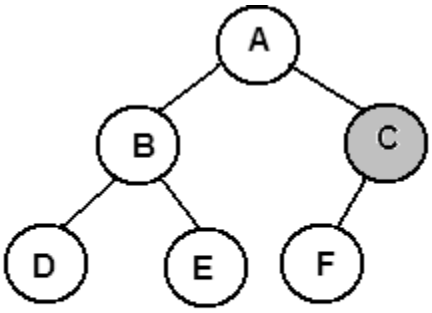




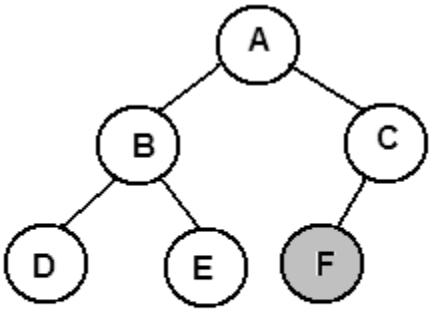
A B D



A B D E



A B D E C



A B D E C F

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

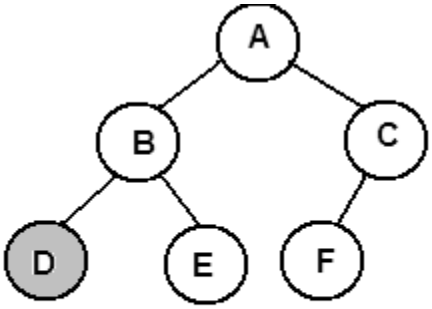
Algorithm

```
PREORDER( ROOT )

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

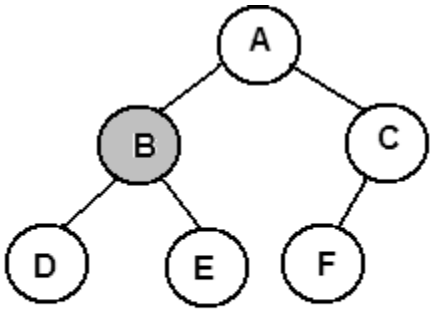
Inorder Traversal

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

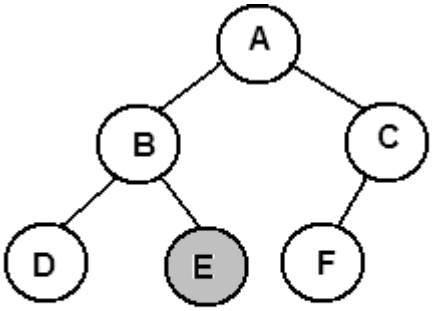
visited. In the following example, the traversal starts with the root of the binary tree. The node A is the root and it is checked if it has the left subtree. Then the inorder traversal procedure is applied on the left subtree of the node A. Now we find that node D does not have left subtree. Hence the 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.



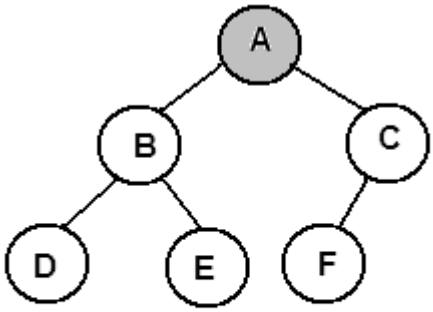
D



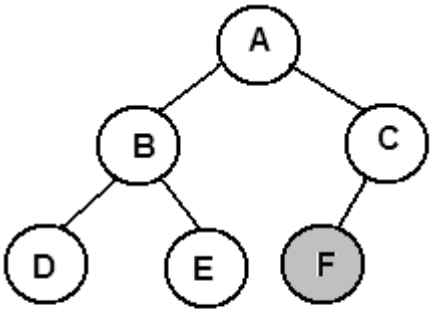
D B



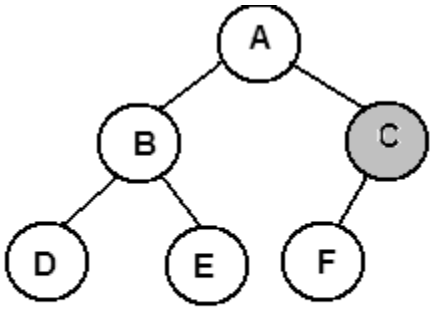
D B E



D B E A



D B E A F



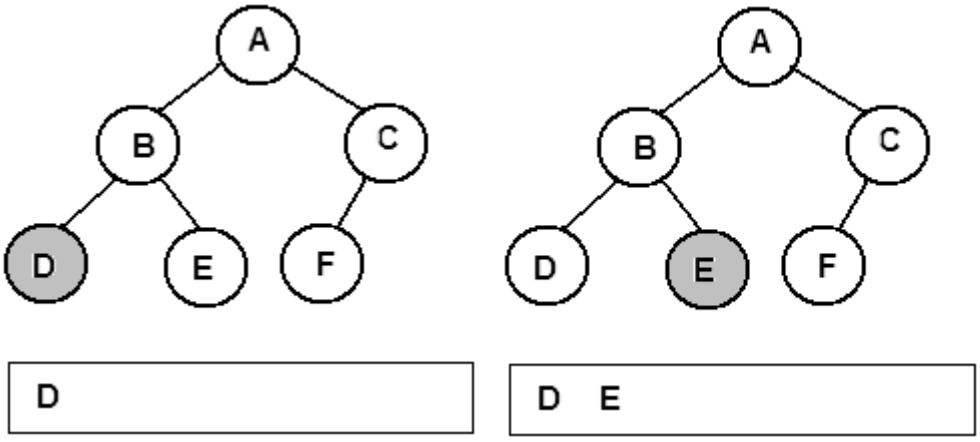
D B E A F C

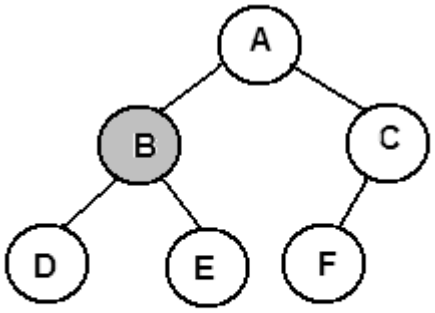
Algorithm

```
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
```

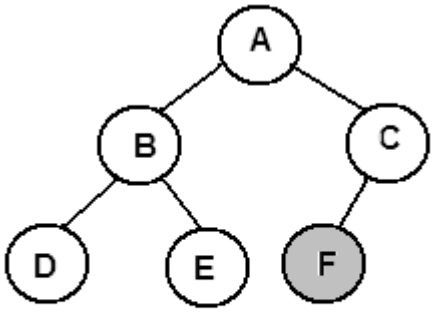
Postorder Traversal

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.

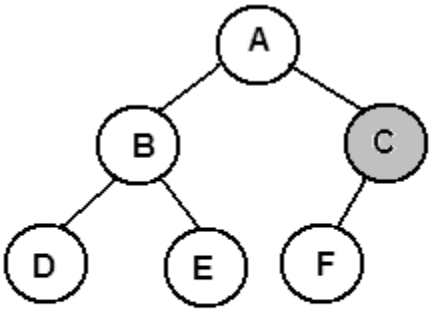




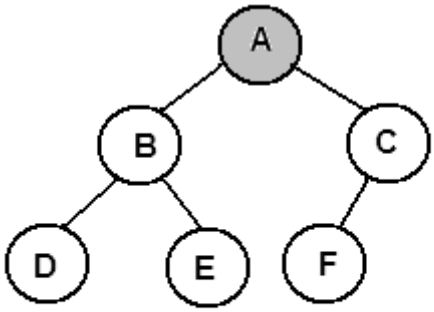
D E B



D E B F



D E B F C



D E B F C A

Algorithm

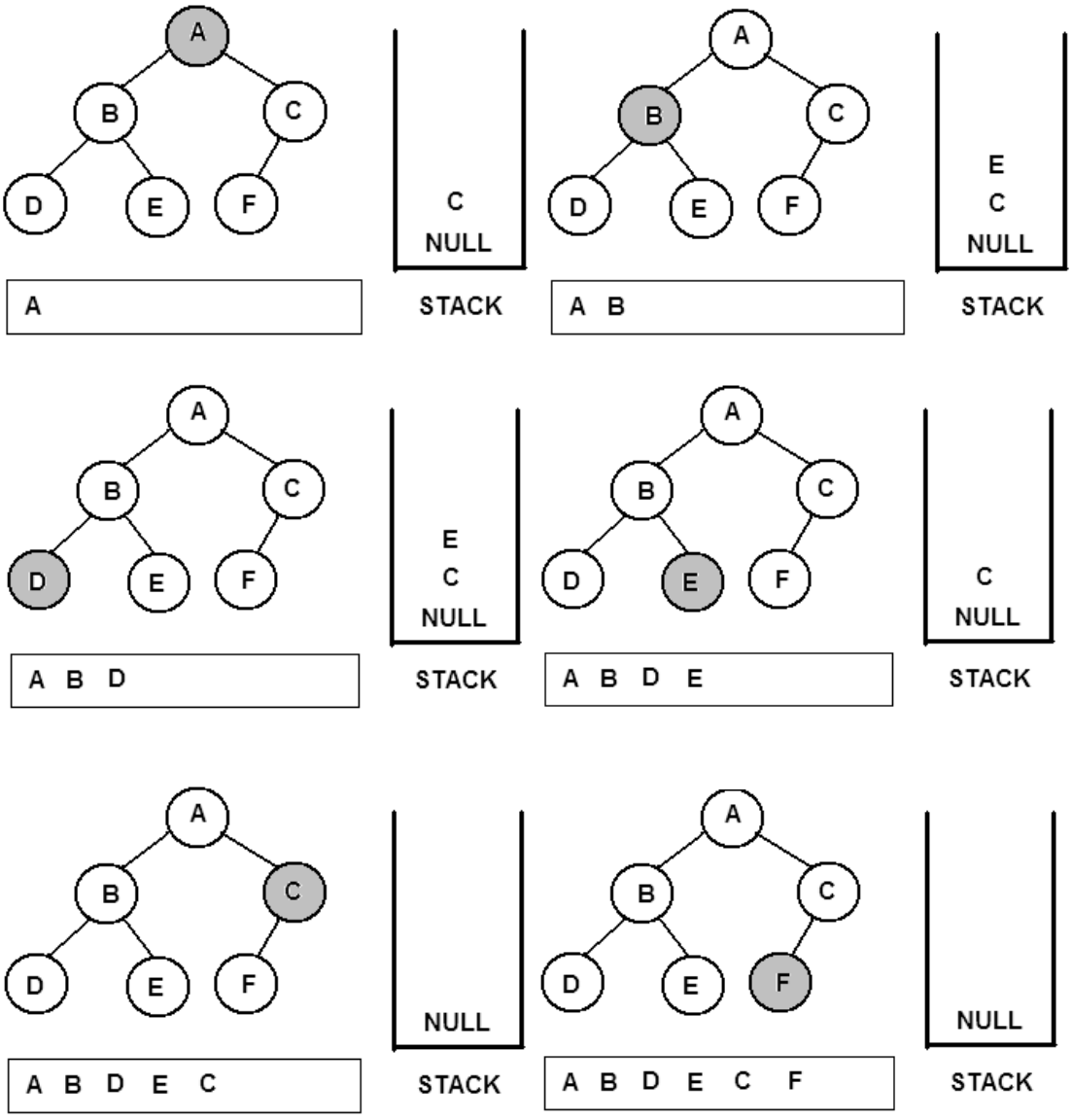
```
POSTORDER( ROOT )

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)

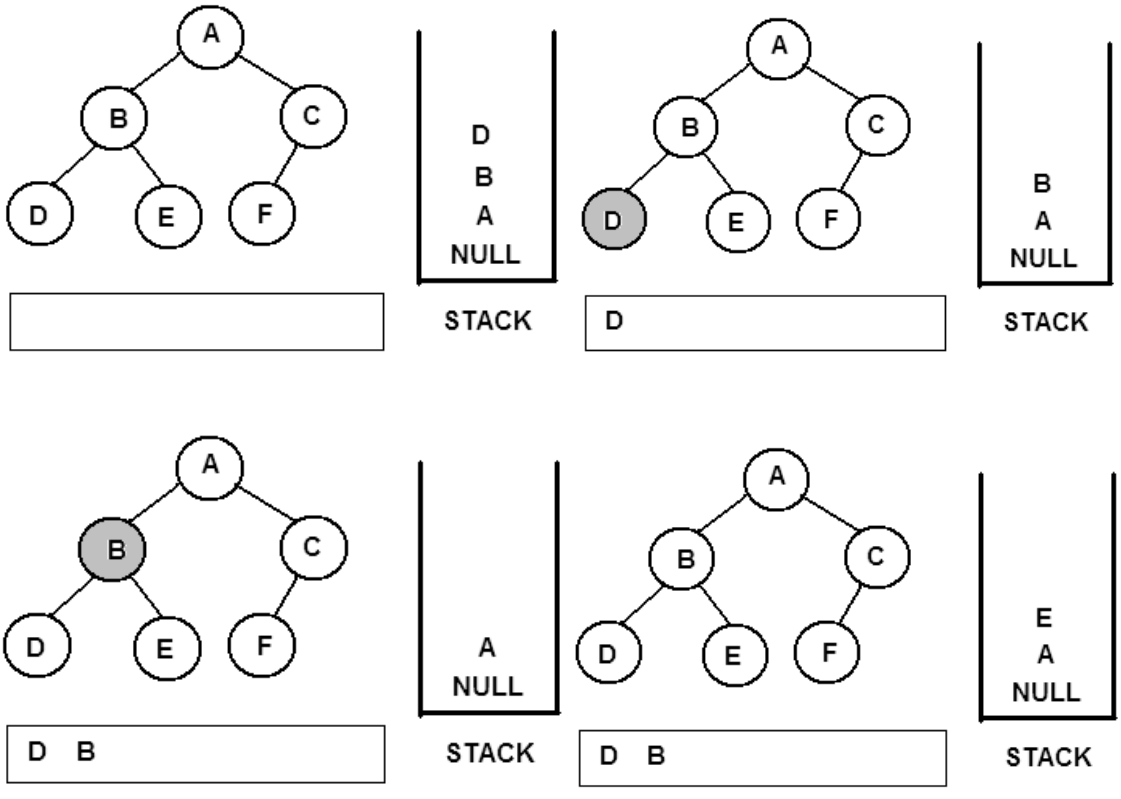
```
Temp = ROOT,  push(NULL)
While temp ≠ NULL
        Print info(temp)
        If right(temp) ≠ NULL
                Push(right(temp))
        End if
```

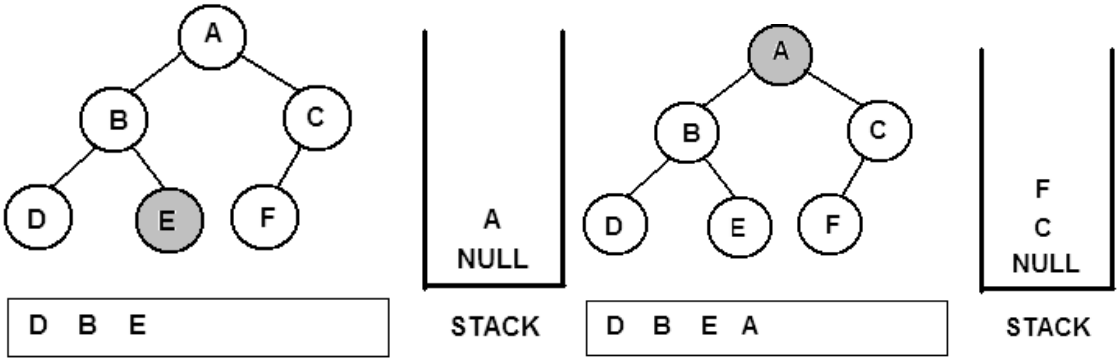
```
        If left(temp) ≠ NULL
                Temp = left(temp)
        Else
                Temp = pop( )
        End if
End while
End PREORDER
```

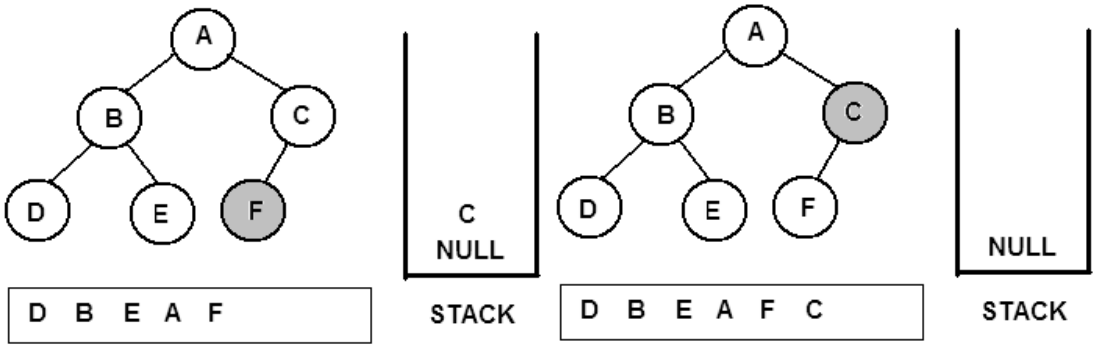
Inorder Traversal

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

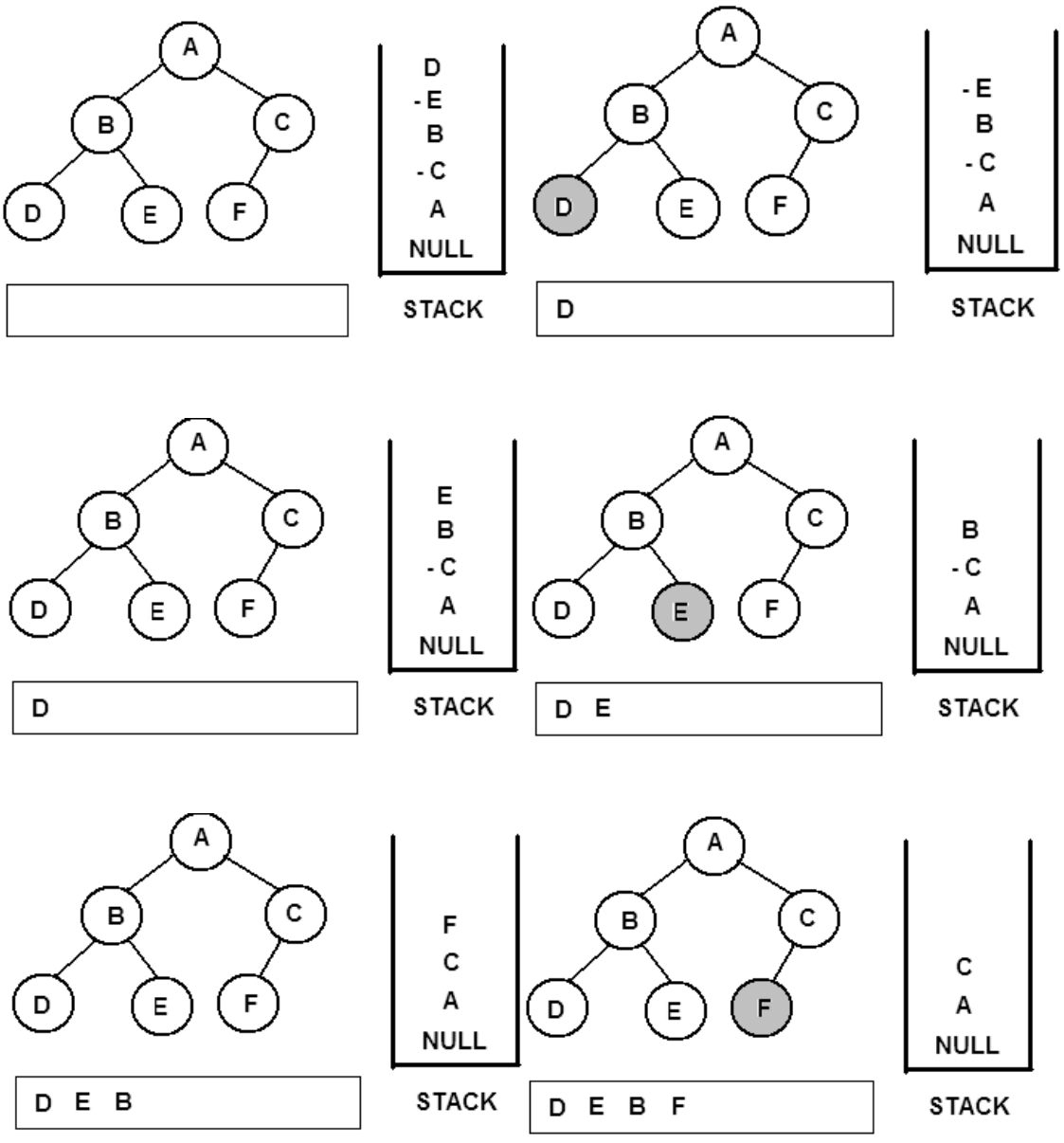
INORDER(ROOT)

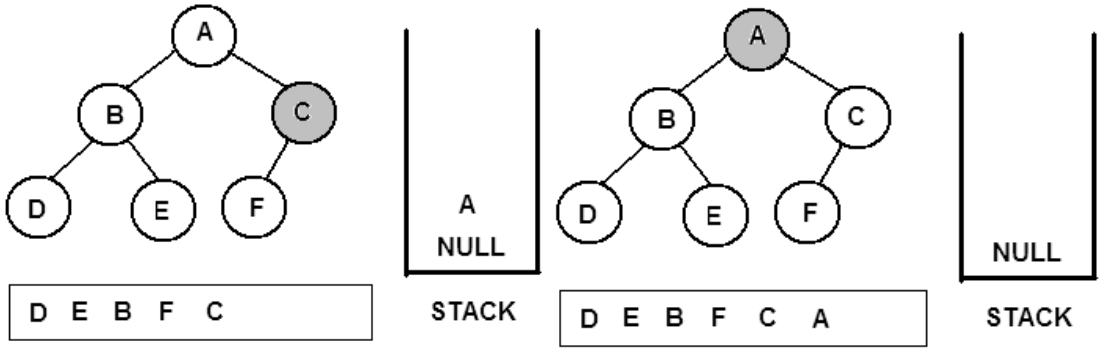
```
Temp = ROOT,  push(NULL),  RIGHTEXISTS = 1
While RIGHTEXISTS = 1
        RIGHTEXISTS = 0
        While temp ≠ NULL
                Push(temp)
                Temp = left(temp)
        End while

        While (TRUE)
                Temp = pop( )
                If temp = NULL
                        Break
                End if
                Print info(temp)
                If right(temp) ≠ 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.





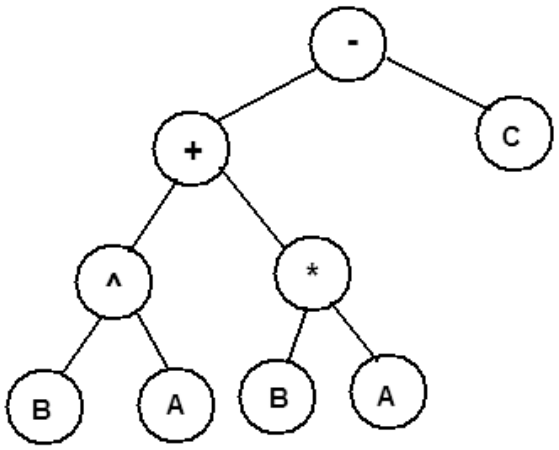
Algorithm

```
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

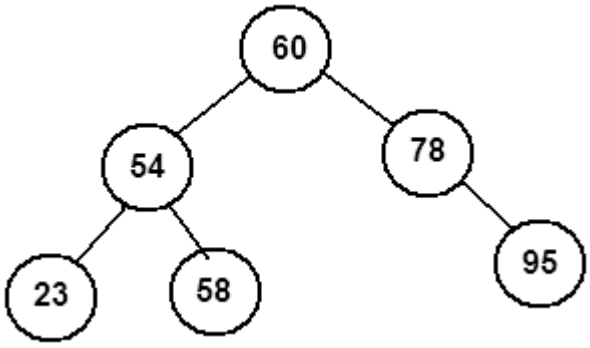
B A ^ B A * C –

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.

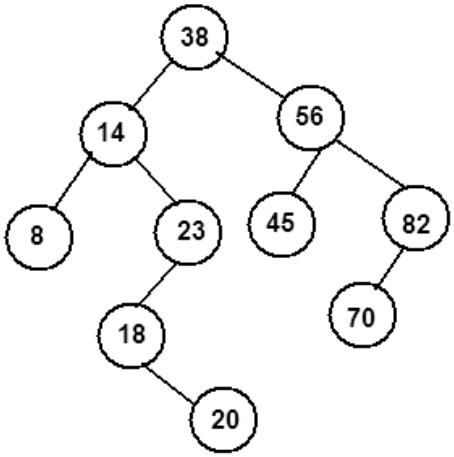
The reason why we go for a Binary Search tree is to improve the searching efficiency. The average case time complexity of the search operation in a binary search tree is O(log n).



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

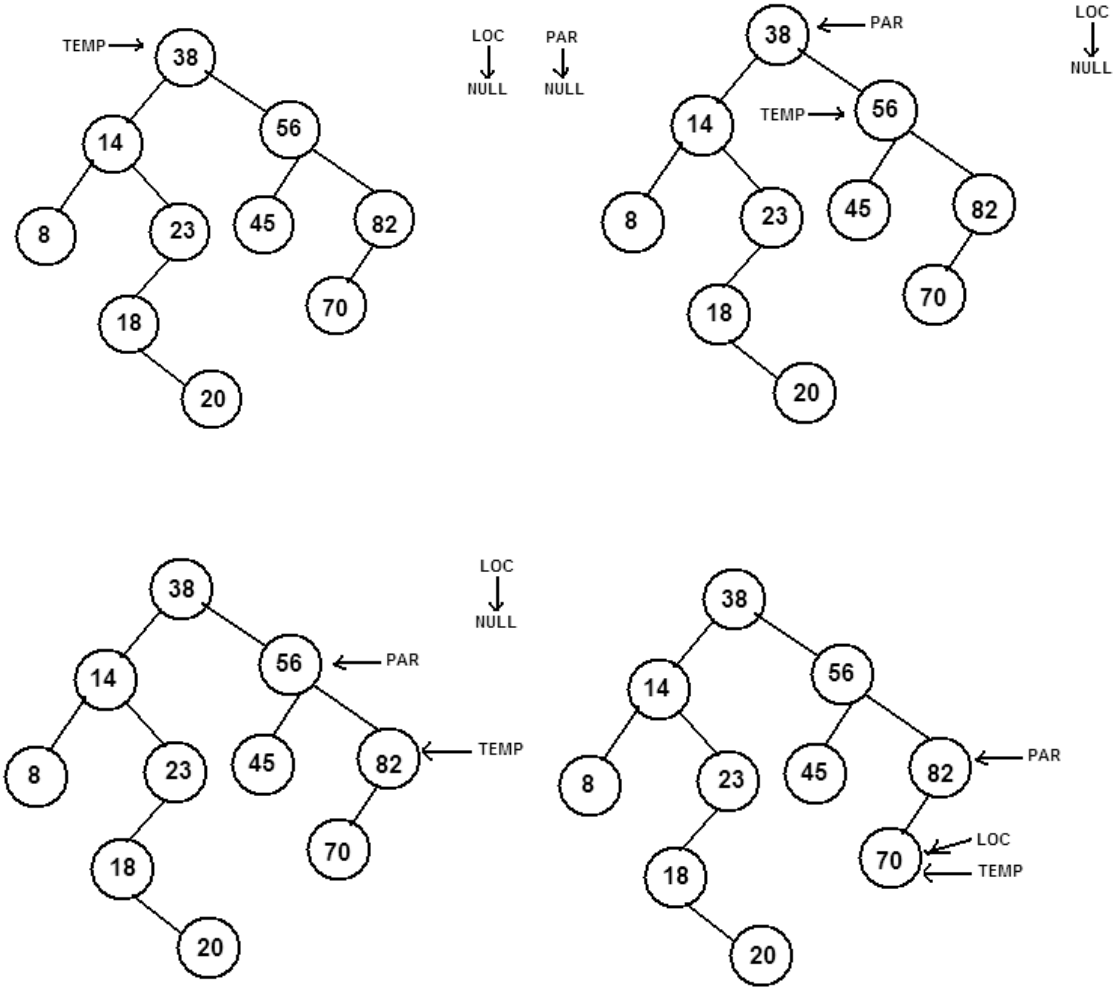
Initially 38 is taken and placed as the root node. The next number 14 is taken and compared with 38. As 14 is lesser than 38, it is placed as the left child of 38. Now the third number 8 is taken and compared starting from the root node 38. Since is 8 is less than 38 move towards left of 38. Now 8 is 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
```

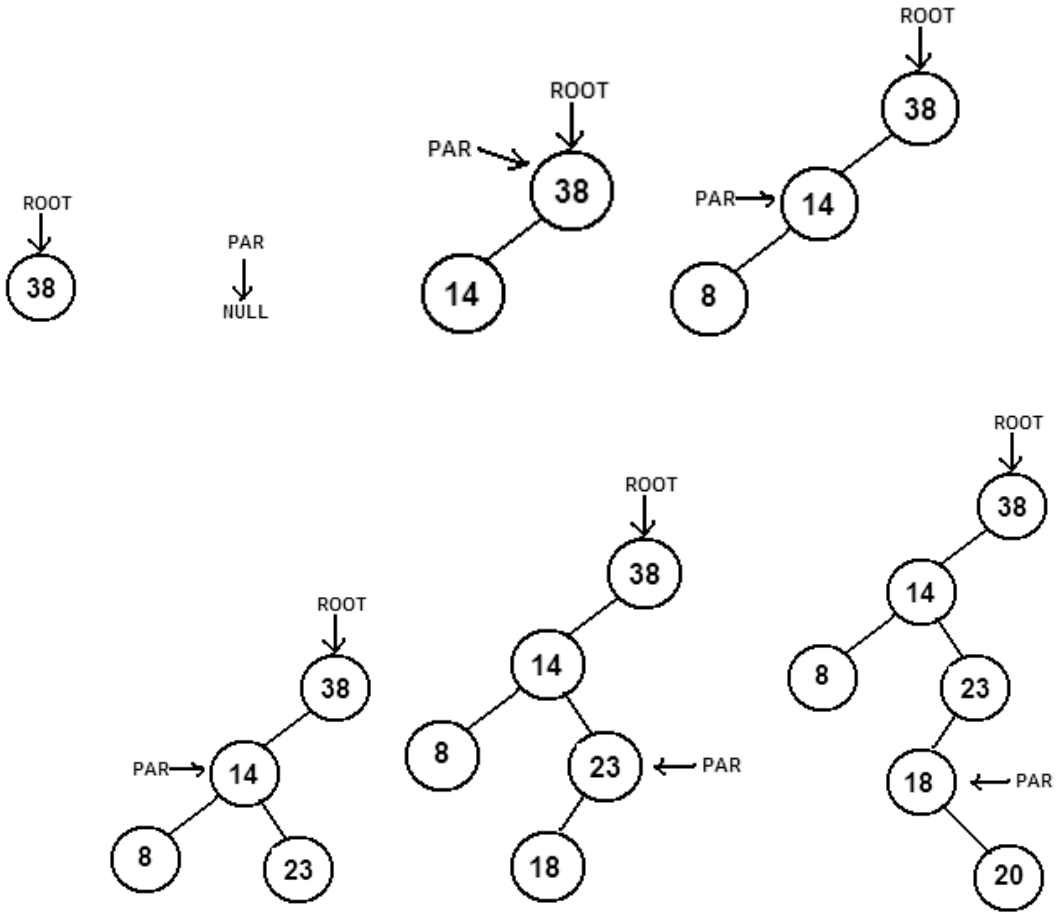
End while
End SEARCH

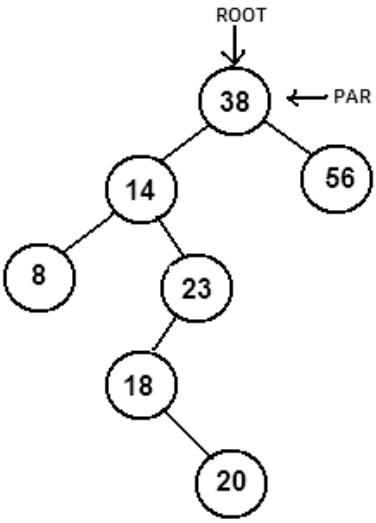
Insert Operation in a Binary Search Tree

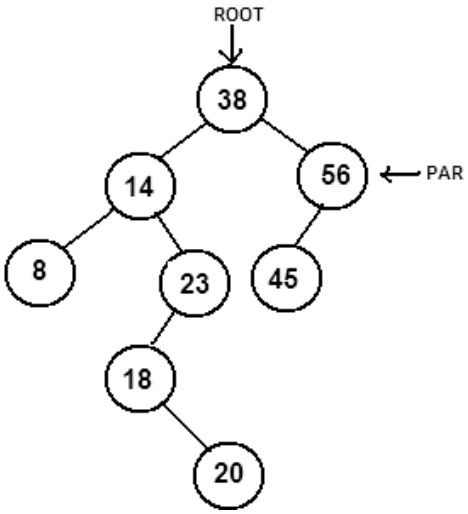
The BST itself is constructed using the insert operation described below. Consider the following list of numbers. A binary tree can be constructed using this list of numbers using the insert operation, as shown.

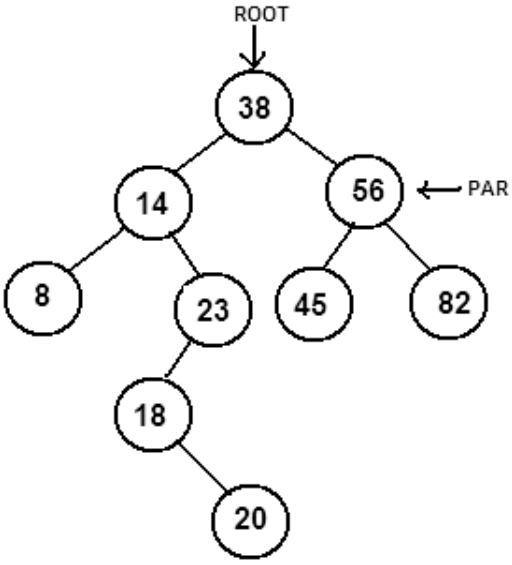
39 14 8 23 18 20 56 45 82 70

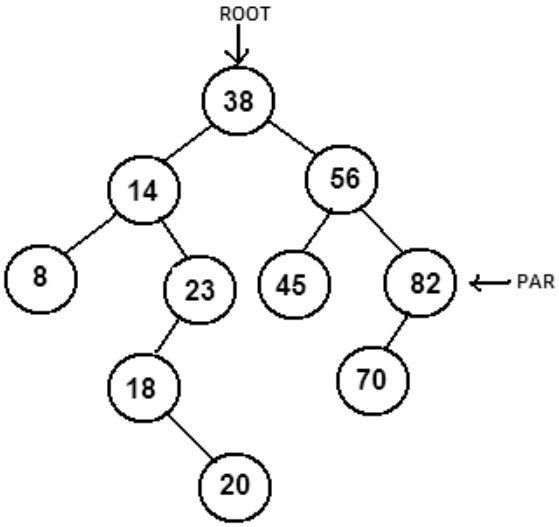
Initially 38 is taken and placed as the root node. The next number 14 is taken and compared with 38. As 14 is lesser than 38, it is placed as the left child of 38. Now the third number 8 is taken and compared starting from the root node 38. Since is 8 is less than 38 move towards left of 38. Now 8 is 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.

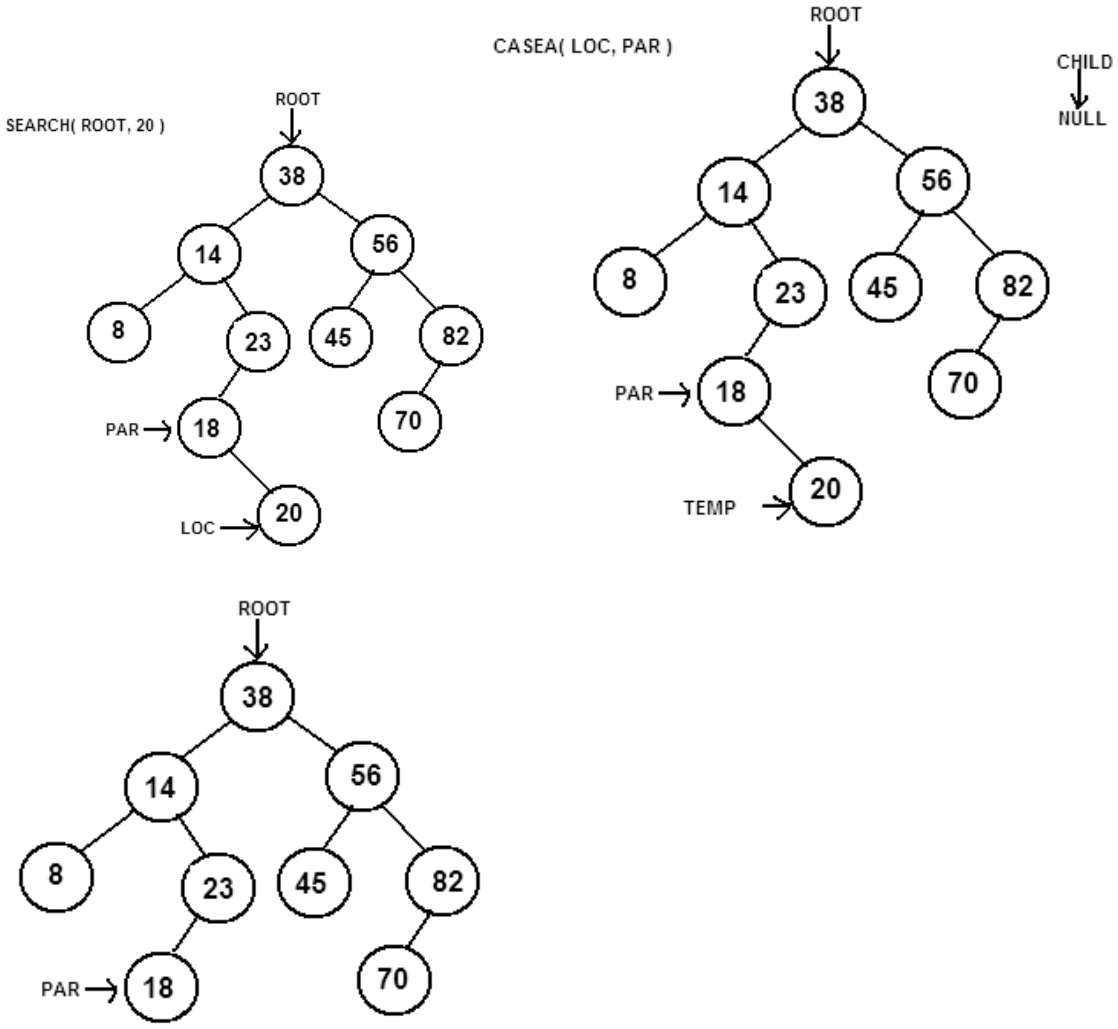
Algorithm

```
DELETE( ROOT, k )

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

If right(Loc) ≠ NULL and left(Loc) ≠ NULL
        CASEA(Loc, par)
Else
        CASEB(Loc, par)
End if

End DELETE
```

Case A:

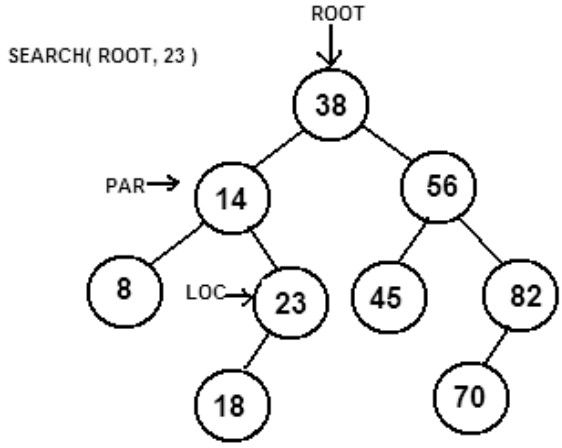
The search is operation is performed for the key value that has to be deleted. The search operation, returns the address of the node to be deleted in the pointer LOC and its parents' address is returned in a pointer PAR. If the node to be deleted has no children then, it is checked whether the node pointed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

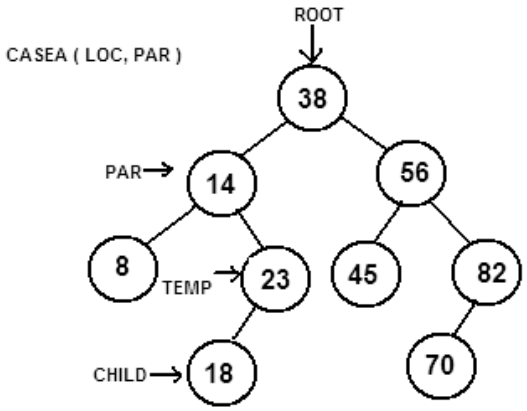
The sequence of steps followed for deleting 20 from the tree is as shown.

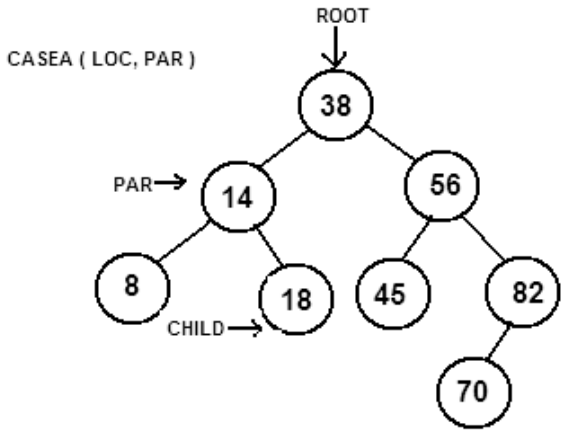


If the node to be deleted has one child, then a new pointer CHILD is made to point to the child of LOC. If LOC is left child of PAR then left of PAR is pointed to CHILD. If LOC is right child of PAR then right of PAR is pointed to CHILD.

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

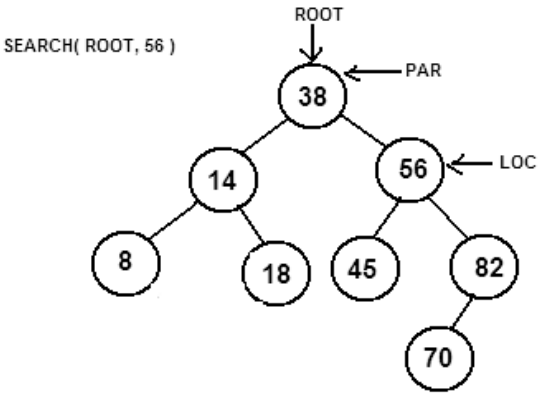


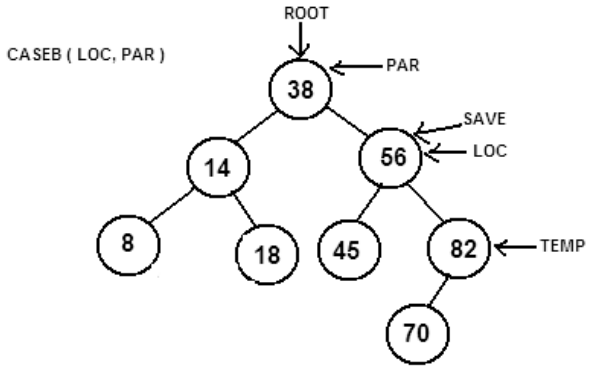


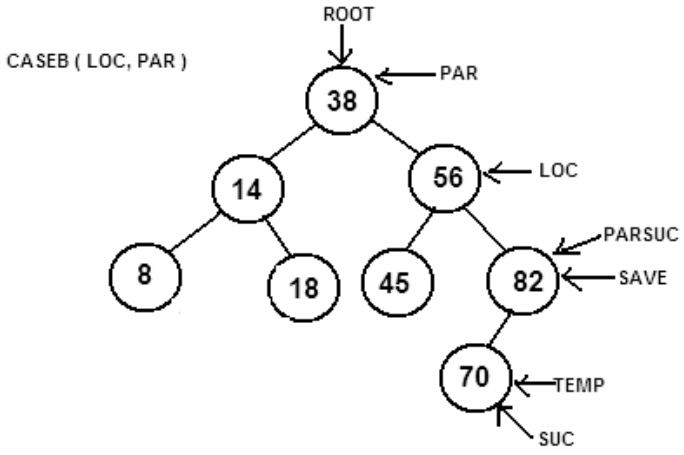


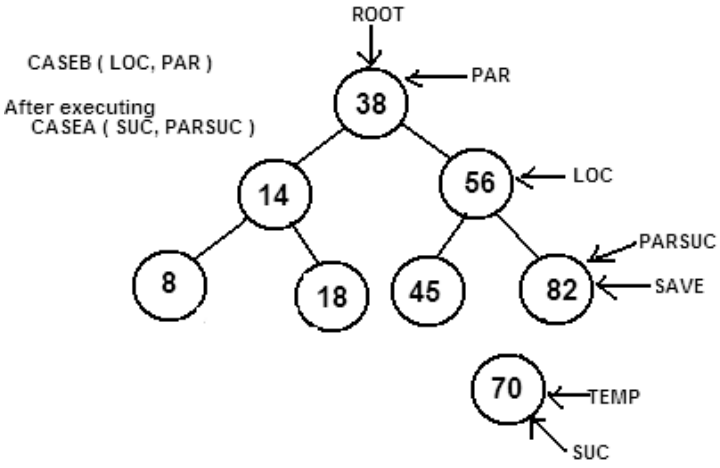
Case B:

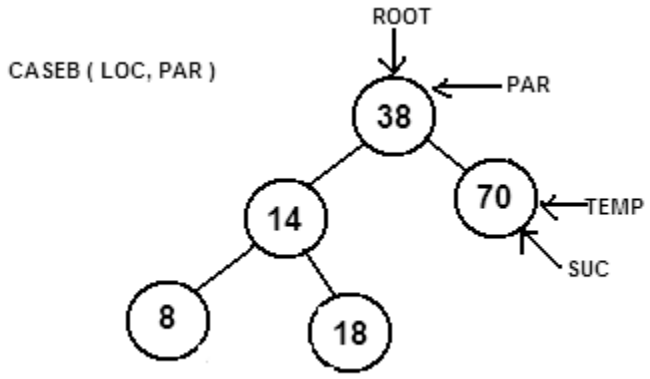
In this case, the node to be deleted has both the left child and the right child. Here we introduce two new pointers SUC and PARSUC. The inorder successor of the node to be deleted is found out and is pointed by SUC and its parent node is pointed by the PARSUC. In the following example the node to be 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.

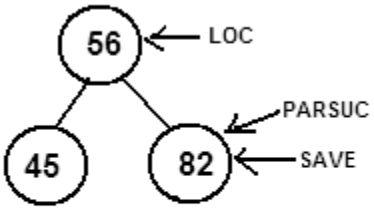


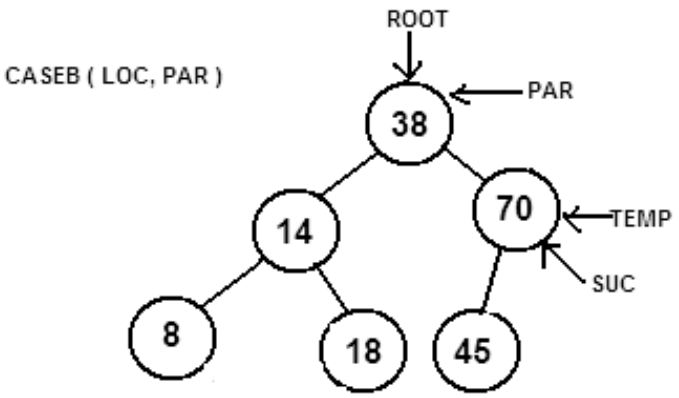


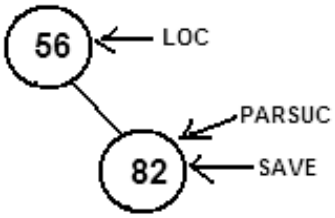


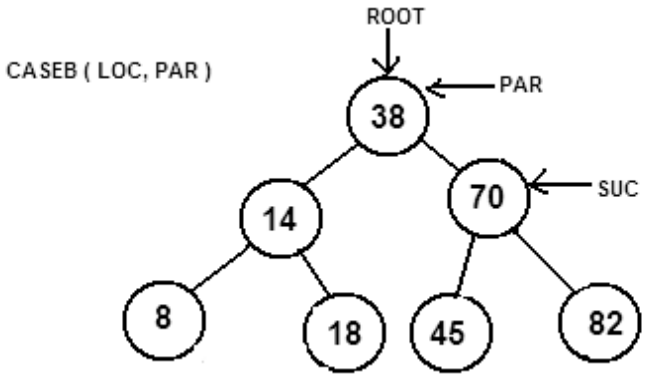

ROOT
CASEB (LOC, PAR)
After executing
CASEA (SUC, PARSUC)
PAR
38
14
56
LOC
8
18
45
82
PARSUC
SAVE
70
TEMP
SUC


ROOT
CASEB (LOC, PAR)
PAR
38
14
70
TEMP
8
18
SUC


56
LOC
PARSUC
45
82
SAVE


ROOT
CASEB (LOC, PAR)
PAR
38
14
70
TEMP
8
18
45
SUC


56
LOC
PARSUC
82
SAVE



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(temp)
                Left(par) = child
        Else
                Right(par) = child
        End if
Else
        ROOT  =  child
End if
End CASEA
```

CASEB(Loc, par)

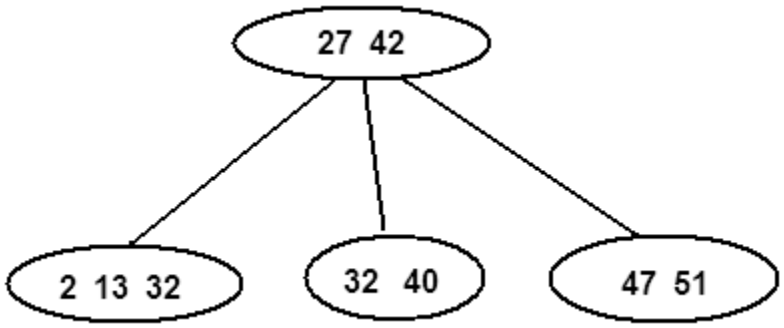
```
Temp = right(Loc)
Save = Loc
While left(temp) ≠ NULL
        Save = temp
        Temp = left(temp)
End while

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
```

B – TREES

Multiway search tree (m-way search tree): *Multiway search tree of order n is a tree in which any node may contain maximum n-1 values and can have maximum n children.*

Consider the following tree. Every node in the tree has one or more than one values stored in it. The tree shown is of order 3. Hence this tree can have maximum 3 children and each node can have maximum 2 values. Hence it is an m-way search tree.

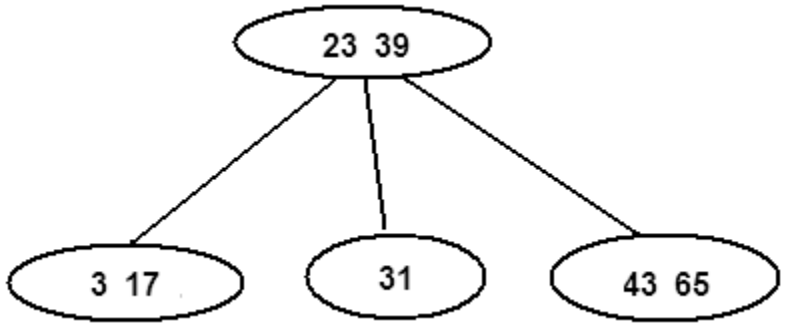


B – Tree: *B – tree is a m-way search tree of order n that satisfies the following conditions.*

(ii) *All non-leaf nodes (except root node) have at least n/2 children and maximum n children.*
(iii) *The non-leaf root node may have at least 2 children and maximum n children.*
(iv) *B-Tree can exist with only one node. i.e. the root node containing no child.*
(v) *If a node has n children then it must have n-1 values.*
(vi) *All the values that appear on the left most child of a node are smaller than the first value of that node. All values that appears on the right most child of a node are greater that the last values of that node.*

(vii)	*If x and y are any two i^{th} and $(i+1)^{th}$ values of a node, where x < y, then all the values appearing on the $(i+1)^{th}$ sub-tree of that node are greater than x and less than y.*
(viii)	*All the leaf nodes should appear on the same level. All the nodes except root node should have minimum n/2 values.*

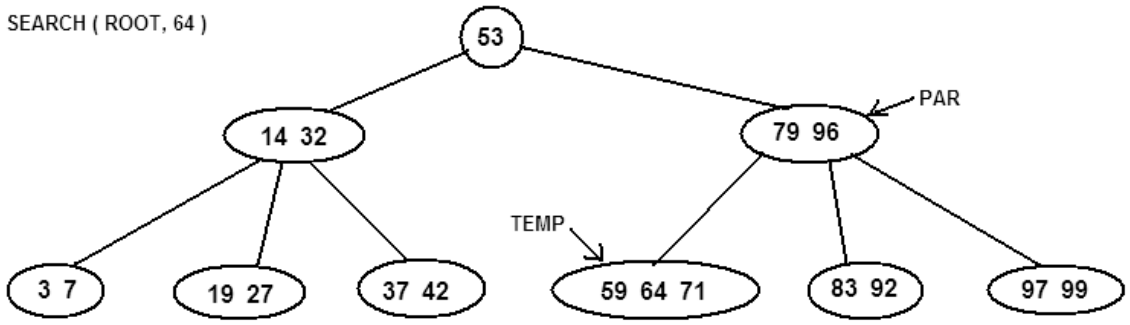
Consider the following tree. Clearly, it is a m-way search tree of order 3. Let us check whether the above conditions are satisfied. It can be seen that root node has 3 children and therefore has only 2 values stored in it. Also it is seen that the elements in the first child (3, 17) are lesser than the value of the first element (23) of the root node. The value of the elements in the second child (31) is greater than the value of the first element of the root node (23) and less than the value of the second element (39) in the root node. The value of the elements in the rightmost child (43, 65) is greater than the value of the rightmost element in the root node. All the three leaf nodes are at the same level (level 2). Hence all the conditions specified above is found to be satisfied by the given m-way search tree. Therefore it is a B-Tee.



Search Operation in a B-Tree

Let us say the number to be searched is k = 64. A temporary pointer temp is made to initially point to the root node. The value k = 64 is now compared with each element in the node pointed by temp. If the value is found then the address of the node where it is found is returned using the temp pointer. If the value k is greater than i^{th} element of the node, then the temp is moved to the $i+1^{th}$ node and the search process is repeated. If the k value is lesser than the first value in the node, then the temp is moved to the first child. If the k value is greater than the last value of the node, then temp is moved to the rightmost child of the node and the search process is repeated.

After the particular node where the value is found is located (now pointed by temp), then a variable LOC is initialized to 0, indicating the position of the value to be searched within that node. The value k is compared with each and every element of the node. When the value of the k is found within the node, then the search comes to an end position where it is found is stored in LOC. If not found the value of LOC is zero indicating that the value is not found.



Algorithm

SEARCH(ROOT, k)

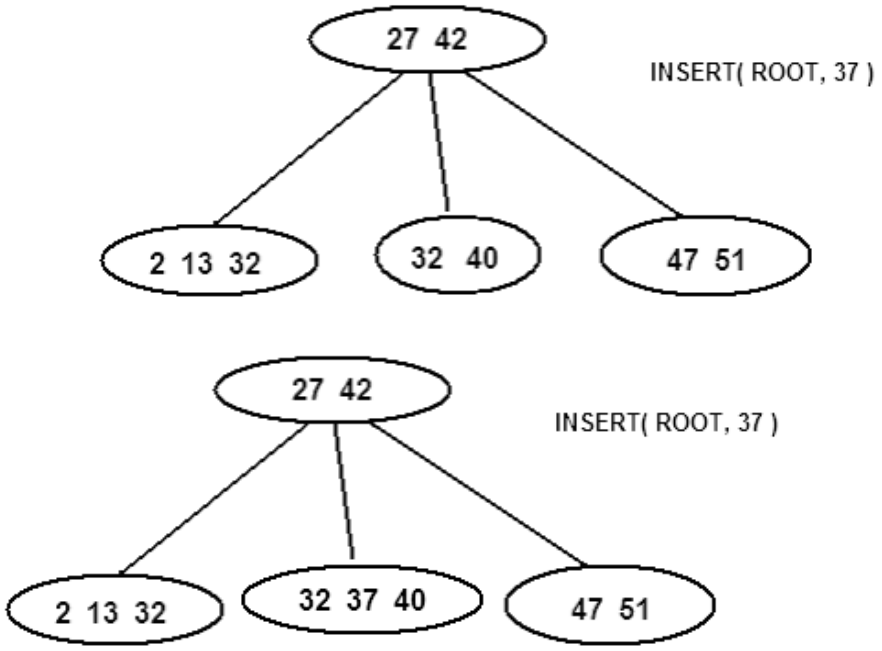
```
Temp = ROOT, i = 1, pos = 0
While i ≤ count(temp) and child[i](temp) ≠ NULL
        If k = info(temp[i])
                Pos = i
                Return temp
        Else
                If k < info(temp[i])
                        SEARCH(child[i](temp), k)
                Else
                        If i = count(temp)
                                Par = temp
                                Temp = child[i+1](temp)
                        Else
                                i = i + 1
                        End if
                End if
        End if
End While

While i ≤ count(temp)
        If k = info(temp[i])
                Pos = i
                Return temp
        End if
End while

End SEARCH
```

Insert Operation in a B-Tree

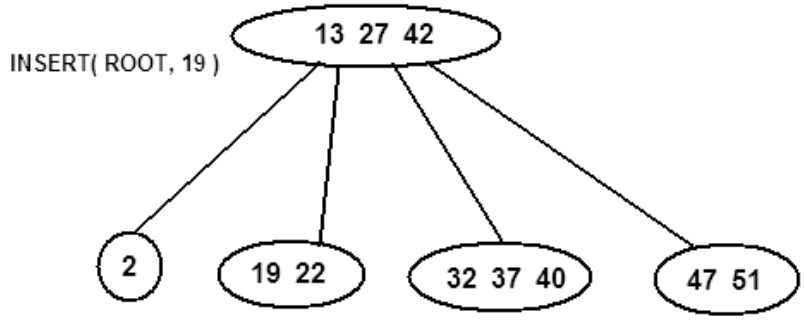
One of the conditions in the B-Tree is that, the maximum number of values that can be present in the node of a tree is n – 1, where n is the order of the tree. Hence, it should be taken care that, even after insertion, this condition is satisfied. There are two cases: In the first case, the element is inserted into a node which already had less than n- 1 values, and the in the second case, the element is inserted into a node which already

had exactly n-1 values. The first case is a simple one. The insertion into the node does not violate any condition of the tree. But in the second case, if the insertion is done, then after insertion, the number values exceeds the limit in that node.

Let us take the first case. In both the cases, the insertion is done by searching for that element in the tree which will give the node where it is to be inserted. While searching, if the value is already found, then no insertion is done as B-Tree is used for storing the key values and keys do not have duplicates. Now the value given is inserted into the node. Consider the figure which shows how value 37 is inserted into correct place.



In the second case, insertion is done as explained above. But now, it is found that, the number of values in the node after insertion exceeds the maximum limit. Consider the same tree shown above. Let us insert a value 19 into it. After insertion of the value 19, the number of values (2, 13, 19, 22) in that node has become 4. But it is a B-Tree of order 4 in which there should be only maximum 3 values per node. Hence the node is split into two nodes, the first node containing the numbers starting from the first value to the value just before the middle value (first node: 2). The second node will contain the numbers starting just after the mid value till the last value (second node: 19, 22). The mid value 13 is pushed into the parent. Now the adjusted B-Tree appears as shown.



Algorithm

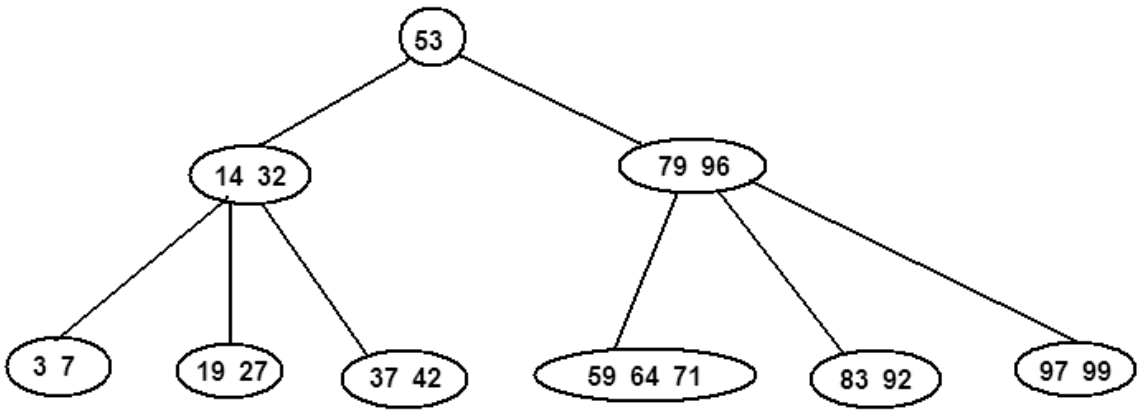
```
INSERT( ROOT, k )

Temp = SEARCH( ROOT, k )
If count(temp) < n-1
        Ins(temp, k)
        Return
Else
        Repeat for i = n/2 +1 to n-1
                Info(R[i-n/2]) = info(temp[i])
                Count(R) = count(R) + 1
        End repeat
        Count(temp) = n/2 – 1
        Ins(par, info(temp[m/2])
End if
INSERT( ROOT, k )

End INSERT
```

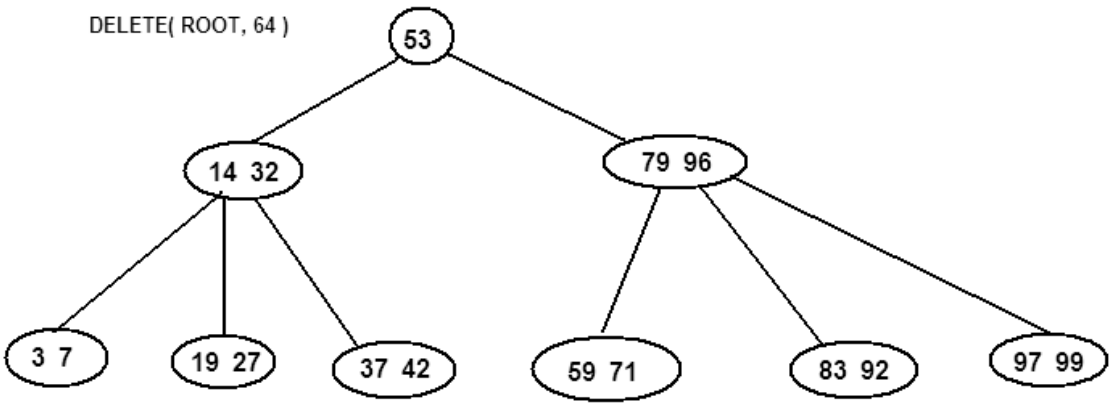
Delete Operation in B-Tree

When the delete operation is performed, we should take care that even after deletion, the node has minimum n/2 value in it, where n is the order of the tree.

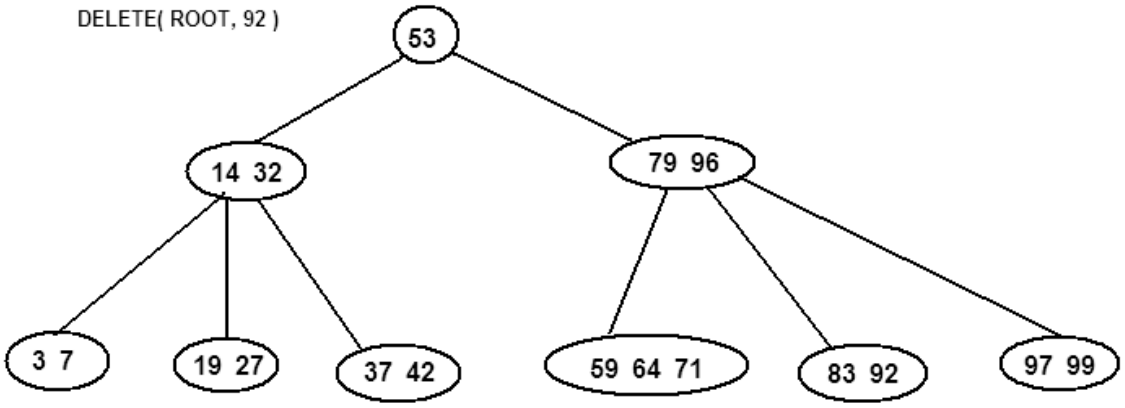
There are three cases:

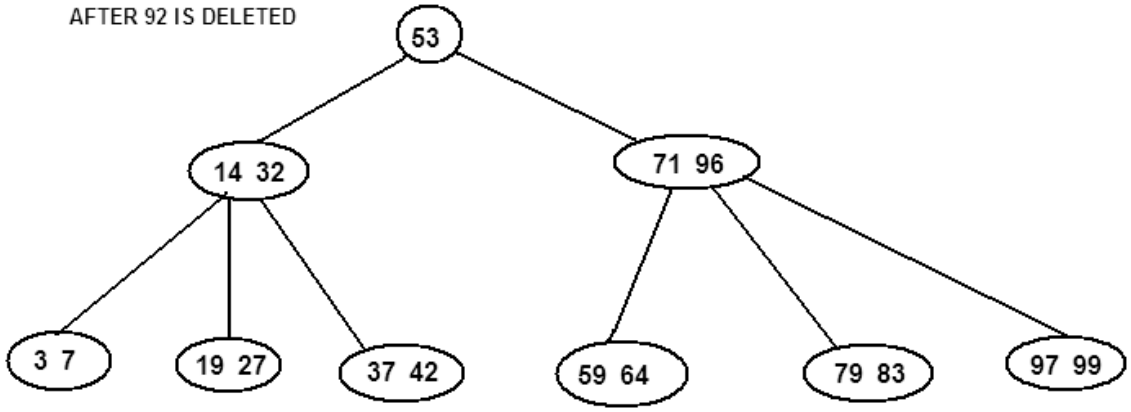
Case 1: The node from which the value is deleted has minimum n/2 values even after deletion. Let us consider the following B-Tree of order 5. A value 64 is to be deleted. Even after the deletion of the value 64, the node has minimum n/2 values (i.e., 2 values). Hence the rules of the B-Tree are not violated.





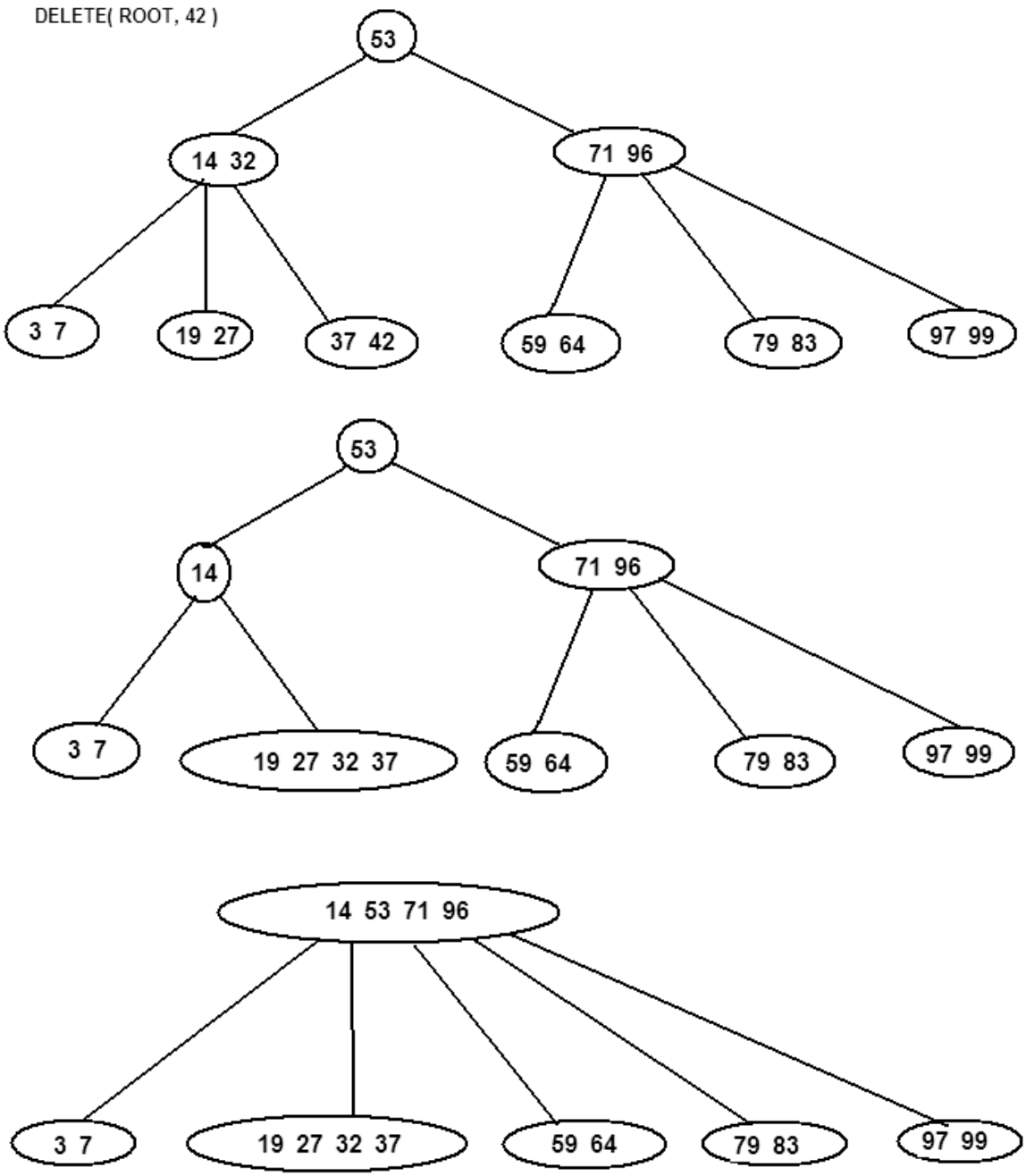
Case 2: In the second case, after the deletion the node has less than minimum n/2 values. Let us say we delete 92 from the tree. After 92 is deleted, the node has only one value 83. But a node adjacent to it consist 3 values (i.e., there are extra values in the adjacent node). Then the last value in that node 71 is pushed to its parent and the first value in the parent namely 79 is pushed into the node which has values less than minimum limit. Now the node has obtained the minimum required values.





Case 3: In the previous case, there was an adjacent node with extra elements and hence the adjustment was made easily. But if all the nodes have exactly the minimum required, and if now a value is deleted from a node in this, then no value can be borrowed from any of the adjacent nodes. Hence as before a value from the parent is pushed into the node

(in this case 32 is pushed down). Then the nodes are merged together. But we see that the parent node has insufficient number of values. Hence same process of merging takes place recursively till the entire tree is adjusted.



Algorithm

DELETE(ROOT, K)

```
Temp = SEARCH( ROOT, k ), DELETED  = 0, i = 1
While i < = count(temp)
        If (k = info(temp[i])
                DELETED = 1
```

```
                Delete temp[i]
        End if
End while

If DELETED = 0
        Print "Item not found"
        Return
Else
        If count(temp) < n / 2
                i = 1
                While i <= count(par)
                        If count(child[i](par)) > n/2
                                s = child[i](par)
                                break
                        Else
                                i = i + 1
                        End if
                End while
                If info(temp[1]) > info(s[count(s)])
                        Ins(temp, info(par[1]))
                        Ins(par, info(s[count(s)]))
                Else
                        Ins(temp, info(par[count(par)]))
                        Ins(par, info(s[1]))
                End if
        End if
End if
End DELETE
```

HEIGHT BALANCED TREES (AVL TREES)

The Height balanced trees were developed by researchers Adelson-Velskii and Landis. Hence these trees are also called AVL trees. Height balancing attempts to maintain the balance factor of the nodes within limit.

Height of the tree: Height of a tree is the number of nodes visited in traversing a branch that leads to a leaf node at the deepest level of the tree.

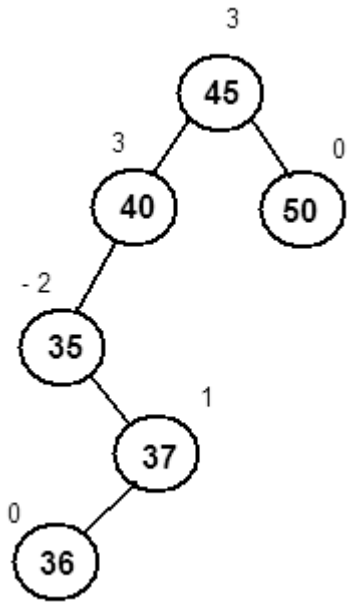
Balance factor: The balance factor of a node is defined to be the difference between the height of the node's left subtree and the height of the node's right subtree.

Consider the following tree. The left height of the tree is 5, because there are 5 nodes (45, 40, 35, 37 and 36) visited in traversing the branch that leads to a leaf node at the deepest level of this tree.

Balance factor = height of left subtree – height of the right subtree

In the following tree the balance factor for each and every node is calculated and shown. For example, the balance factor of node 35 is (0 – 2) = - 2.

The tree which is shown below is a binary search tree. The purpose of going for a binary search tree is to make the searching efficient. But when the elements are added to the binary search tree in such a way that one side of the tree becomes heavier, then the searching becomes inefficient. The very purpose of going for a binary search tree is not served. Hence we try to adjust this unbalanced tree to have nodes equally distributed on both sides. This is achieved by rotating the tree using standard algorithms called the AVL rotations. After applying AVL rotation, the tree becomes balanced and is called the AVL tree or the height balanced tree.

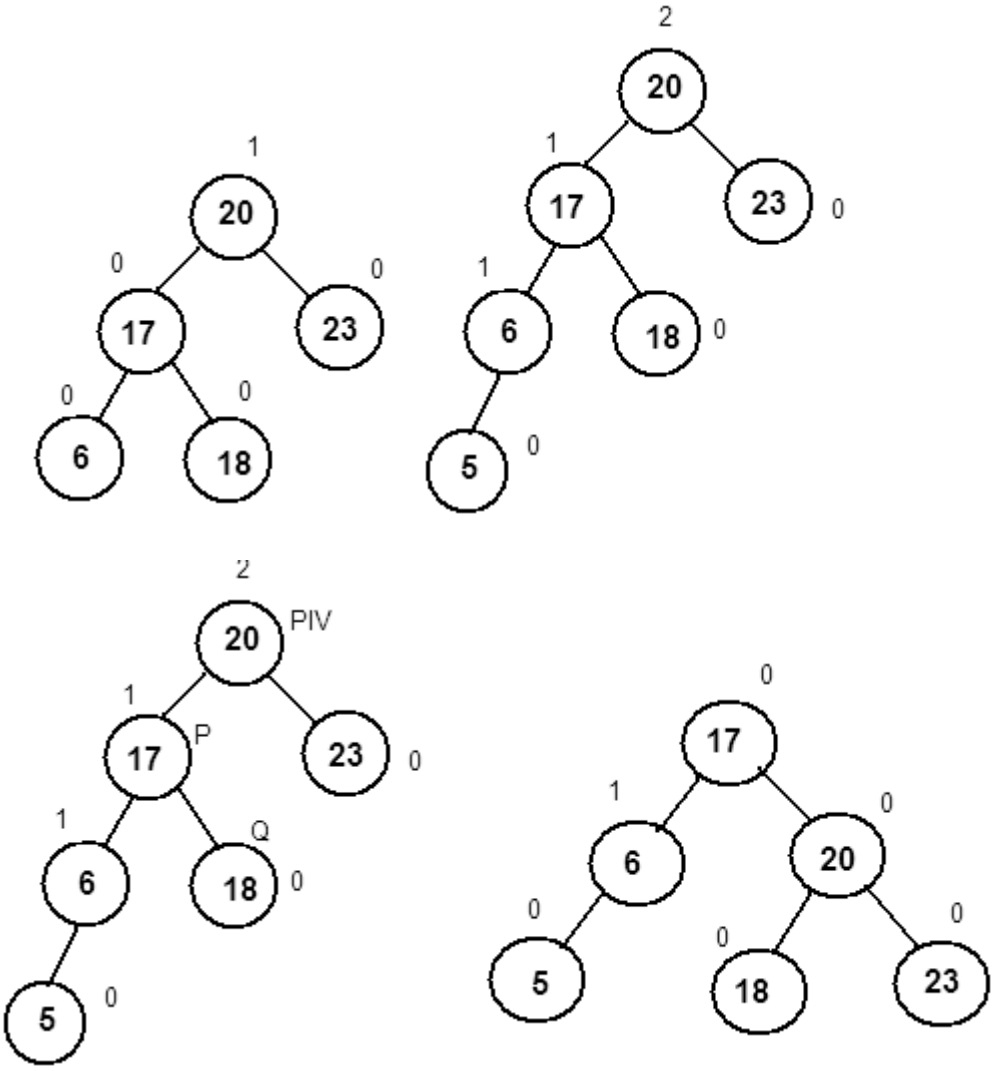


The tree is said to be balanced if each node consists of a balance factor either -1 or 0 or 1. If even one node has a balance factor deviated from these values, then the tree is said to be unbalanced. There are four types of rotations. They are:

1. *Left-of-Left rotation.*
2. *Right-of-Right rotation.*
3. *Right-of-Left rotation.*
4. *Left-of-Right rotation.*

Left-of-Left Rotation

Consider the following tree. Initially the tree is balanced. Now a new node 5 is added. This addition of the new node makes the tree unbalanced as the root node has a balance factor 2. Since this is the node which is disturbing the balance, it is called the pivot node for our rotation. It is observed that the new node was added as the left child to the left subtree of the pivot node. The pointers P and Q are created and made to point to the proper nodes as described by the algorithm. Then the next two steps rotate the tree. The last two steps in the algorithm calculates the new balance factors for the nodes and is seen that the tree has become a balanced tree.



Algorithm

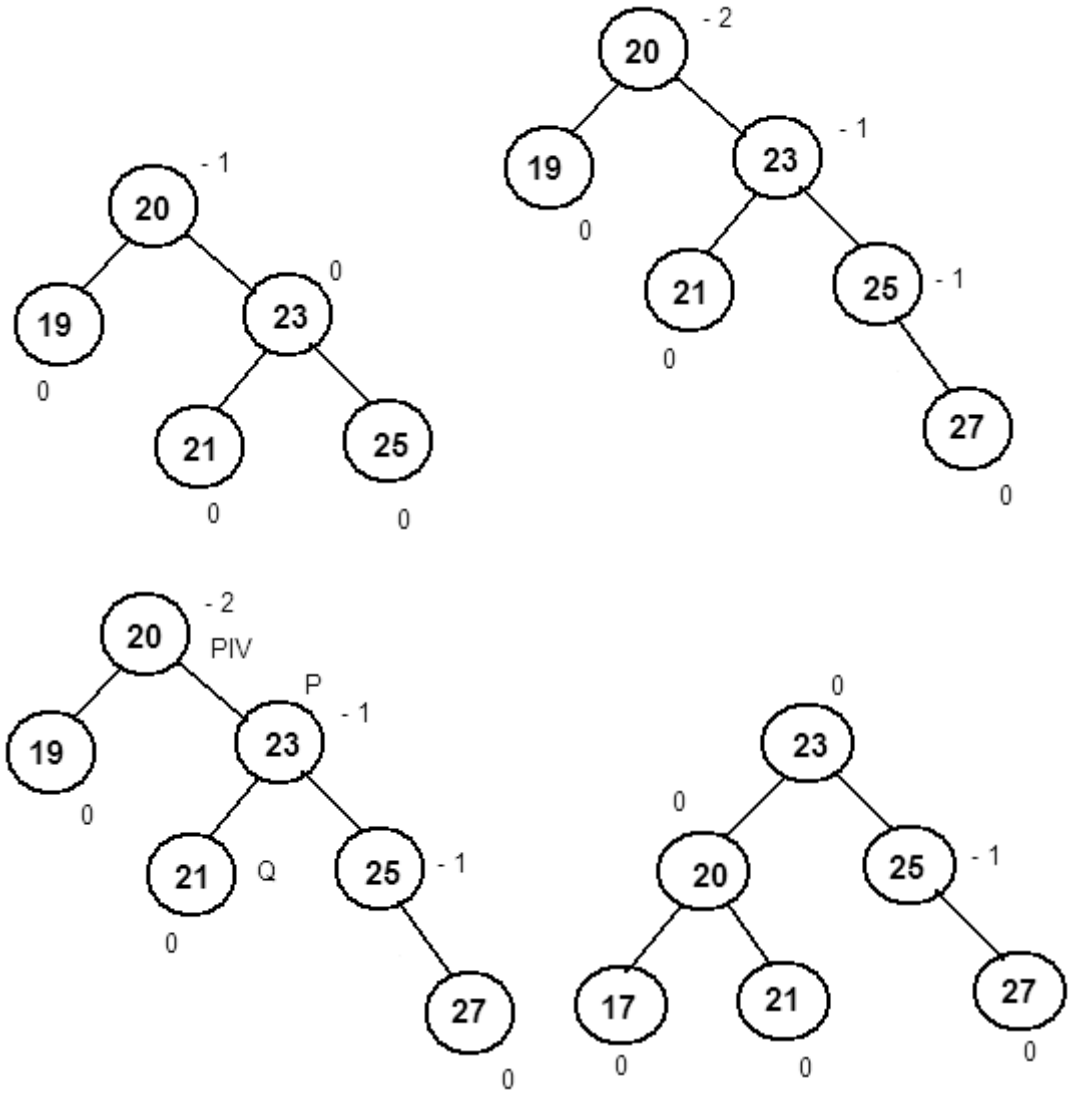
LEFT-OF-LEFT(pivot)

P = left(pivot)
Q = right(P)
Right(P) = pivot
Left(pivot) = Q
Pivot = P
Bal(pivot) = 0
Bal(right(pivot)) = 0

End LEFT-OF-LEFT

Right-of-Right Rotation

In this case, the pivot element is fixed as before. The new node is found to be added as the right child to the right subtree of the pivot element. The first two steps in the algorithm sets the pointer P and Q to the correct positions. The next two steps rotate the tree to balance it. The last two steps calculate the new balance factor of the nodes.



Algorithm

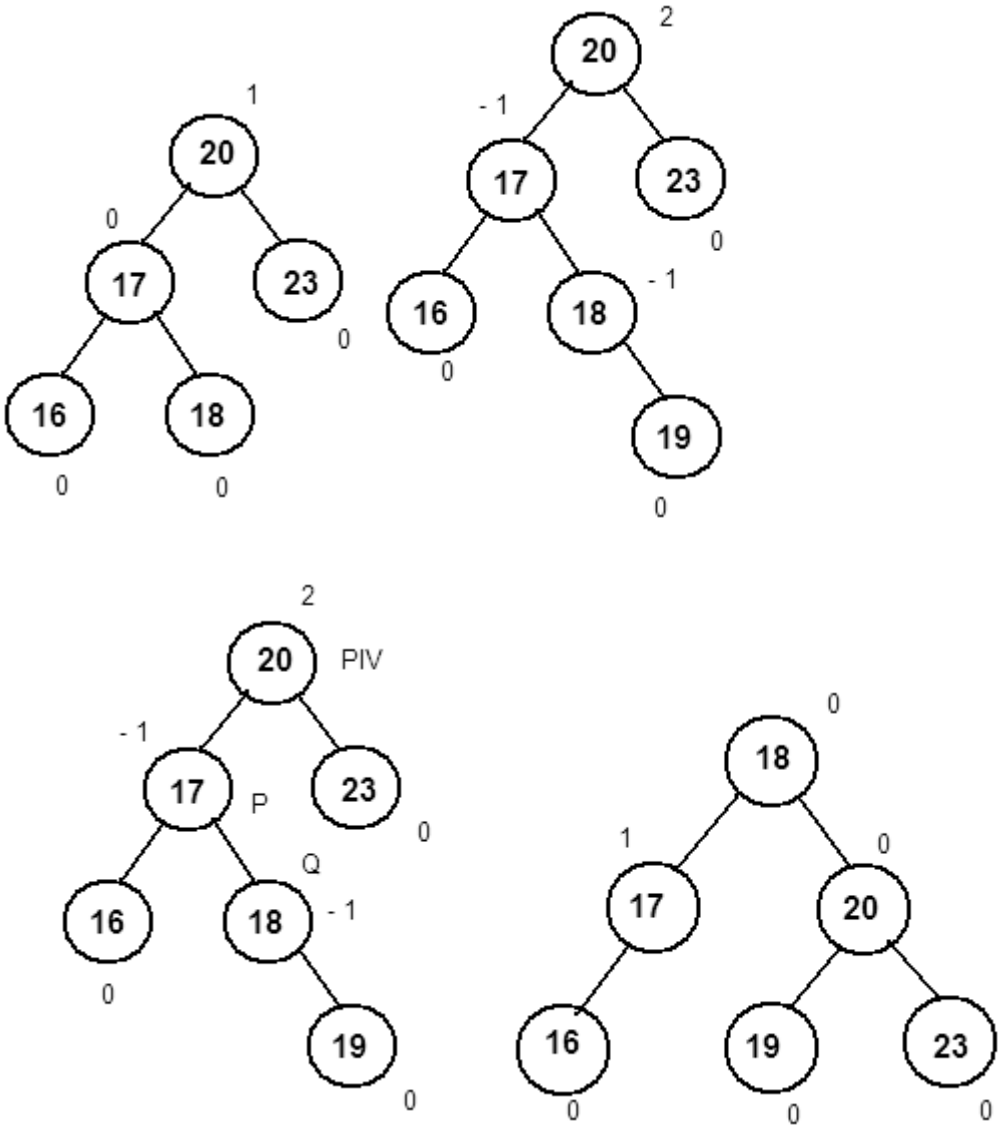
RIGHT-OF-RIGHT(pivot)

P = right(pivot)
Q = left(P)
Left(P) = pivot
Right(pivot) = Q
Pivot = P
Bal(pivot) = 0
Bal(left(pivot)) = 0

End RIGHT-OF-RIGHT

Right-of-Left Rotation

In this following tree, a new node 19 is added. This is added as the right child to the left subtree of the pivot node. The node 20 fixed as the pivot node, as it disturbs the balance of the tree. In the first two steps the pointers P and Q are positioned. In the next four steps, tree is rotated. In the remaining steps, the new balance factors are calculated.



Algorithm

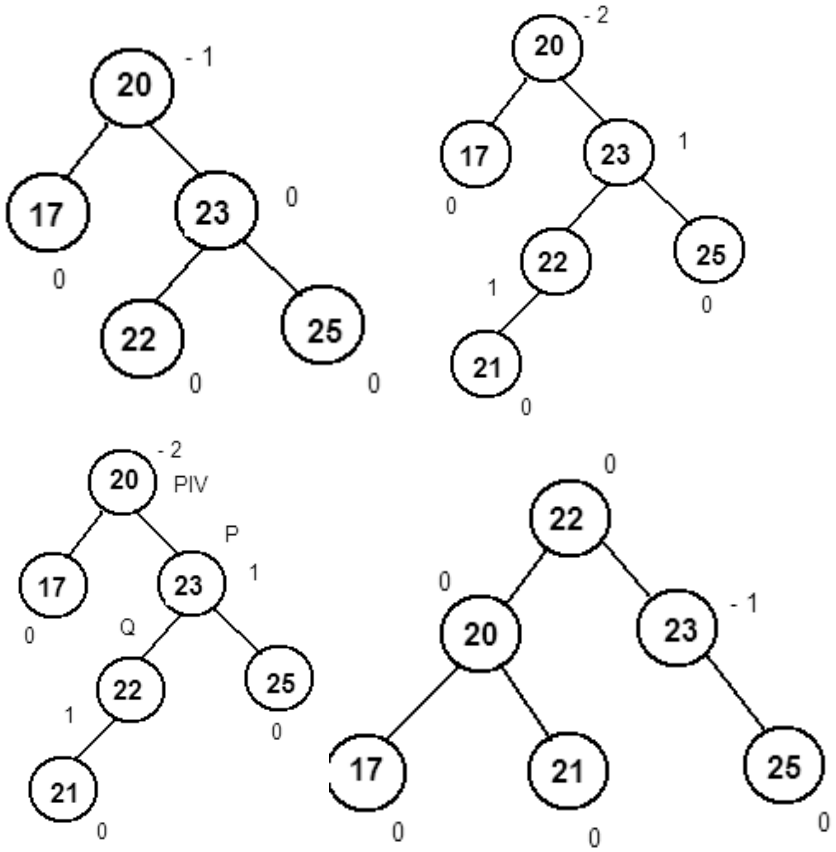
RIGHT-OF-LEFT(pivot)

P = left(pivot)
Q = right(P)
Left(pivot) = right(Q)
Right(P) = left(Q)
Left(Q) = P
Right(Q) = pivot

```
Pivot = Q
If Bal(pivot) = 0
        Bal(left(pivot)) = 0
        Bal(right(pivot)) = 0
Else
        If Bal(pivot) = 1
                Bal(pivot) = 0
                Bal(left(pivot)) = 0
                Bal(right(pivot)) = -1
        Else
                Bal(pivot) = 0
                Bal(left(pivot)) = 1
                Bal(right(pivot)) = 0
        End if
End if

End RIGHT-OF-LEFT
```

Left-of-Right

In the following tree, a new node 21 is added. The tree becomes unbalanced and the node 20 is the node which has a deviated balance factor and hence fixed as the pivot node. In the first two steps of the algorithm, the pointers P and Q are positioned. In the next 4 steps the tree is rotated to make it balanced. The remaining steps calculate the new balance factors for the nodes in the tree.



Algorithm

```
LEFT-OF-RIGHT(pivot)

P = right(pivot)
Q = left(P)
Right(pivot) = left(Q)
Left(P) = right(Q)
Right(Q) = P
Left(Q) = pivot
Pivot = Q
If Bal(pivot) = 0
        Bal(right(pivot)) = 0
        Bal(left(pivot)) = 0
Else
        If Bal(pivot) = 1
                Bal(pivot) = 0
                Bal(right(pivot)) = 0
                Bal(left(pivot)) = -1
        Else
                Bal(pivot) = 0
                Bal(right(pivot)) = 1
                Bal(left(pivot)) = 0
        End if
End if

End LEFT-OF-RIGHT
```

HEAPS

Heaps : *Suppose H is a complete a binary tree with n elements, then H is called a heap, or* ***a maxheap****, if each node N of H has the following property : The value at N is greater than or equal to the value at each of the children of N. A* ***minheap*** *is a heap, where the value at N is lesser than or equal to the value at each of the children of N.*

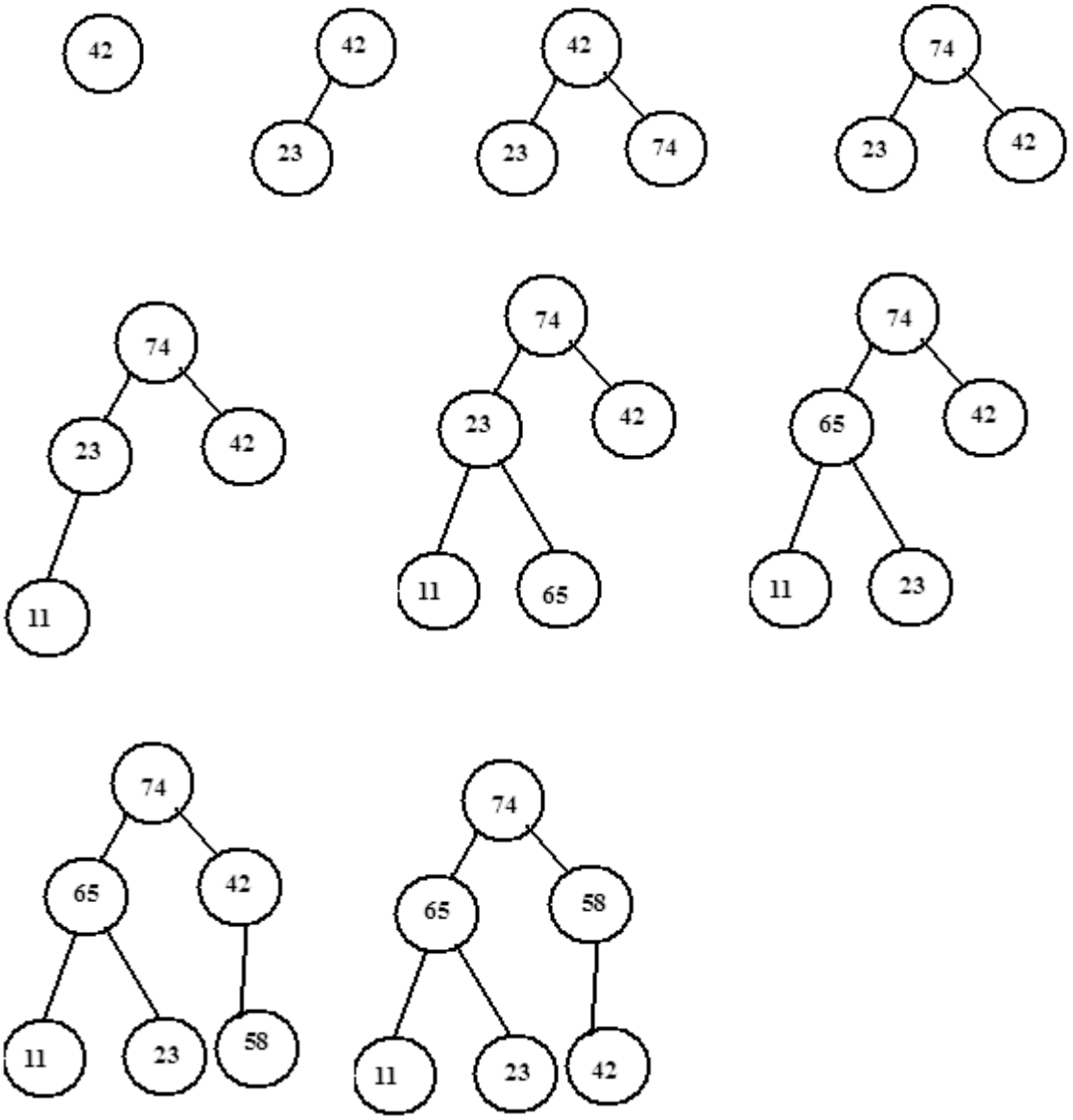
Insert operation in a Heap

The Insert operation is used to insert an element into the heap as well as to build the heap from the scratch. Consider the following list of numbers. If a max heap is to be constructed using the list of numbers then that is done as shown.

42	**23**	**74**	**11**	**65**	**58**

The first element is placed as the root of the tree. The next element is placed at the end of the present tree. Whenever new element is added at the end of the tree, it is compared with its parent and if it is greater than its parent, then it is exchanged with its

parent. The same process is continued till the root or till the new node finds its correct position.



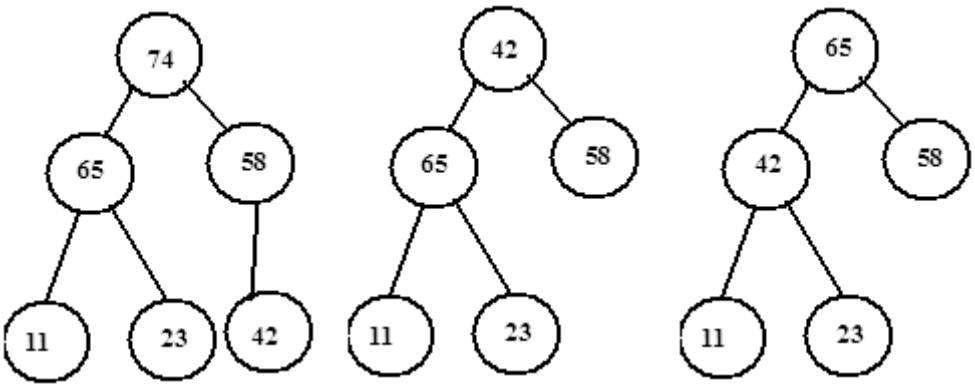
Algorithm

```
INSERT( A[ ], T, k )

N = T
N = N + 1
A[N] = k

While N! = 1
        If A[N] > A[N/2]
                N = N/2
        Else
                Break
        End if
End while
```

```
T = T + 1
End INSERT
```

Delete operation in a Heap

The Delete operation always deletes the root element from the heap. The last element of the heap is overwritten in the root node. Now the root node does not satisfy the heap property, hence we need to heapify the entire tree. Hence a procedure called walkdown is applied on the root node, which walks down the root node to its correct position, thus making the tree again a heap.



Algorithm

```
DELETE( A[ ], T, k )

A[1] = A[T]
T = T – 1
WALKDOWN( A[ ], 1, T )

End DELETE
```

```
WALKDOWN(A[ ] , I, N)

// A is the array used for implementing the heap
// N is the number of elements in the heap
// I is the position of the node where the walkdown procedure is to be applied.

While I ≤ N/2
        L ← 2I, R ← 2I + 1
If A[L] > A[I]
Then
        M ← L
Else
```

```
        M ← I
End If
If A[R] > A[M] and R ≤ N
Then
        M ← R
End If
If M ≠ I
Then
        A[I] ↔ A[M]
        I ← M
Else
        Return
End If
End While

End WALKDOWN
```

----------End of 2nd unit----------