UNIT – IV

TREES

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

"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."

REPRESENTATION OF TREES

Root

An unique node in the tree to which subtrees are attached.

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 total number of subtrees of a node is called the degree of the node. The maximum degree in the tree is called degree of a tree.

Parent

The node having the sub-branches.

Children

The nodes branching from a particular node X are called children of X.

Siblings

Children of the same parent are said to be siblings.

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.

Forest

It is a collection of disjoint trees. It is obtained by removing root.

Non – Terminals

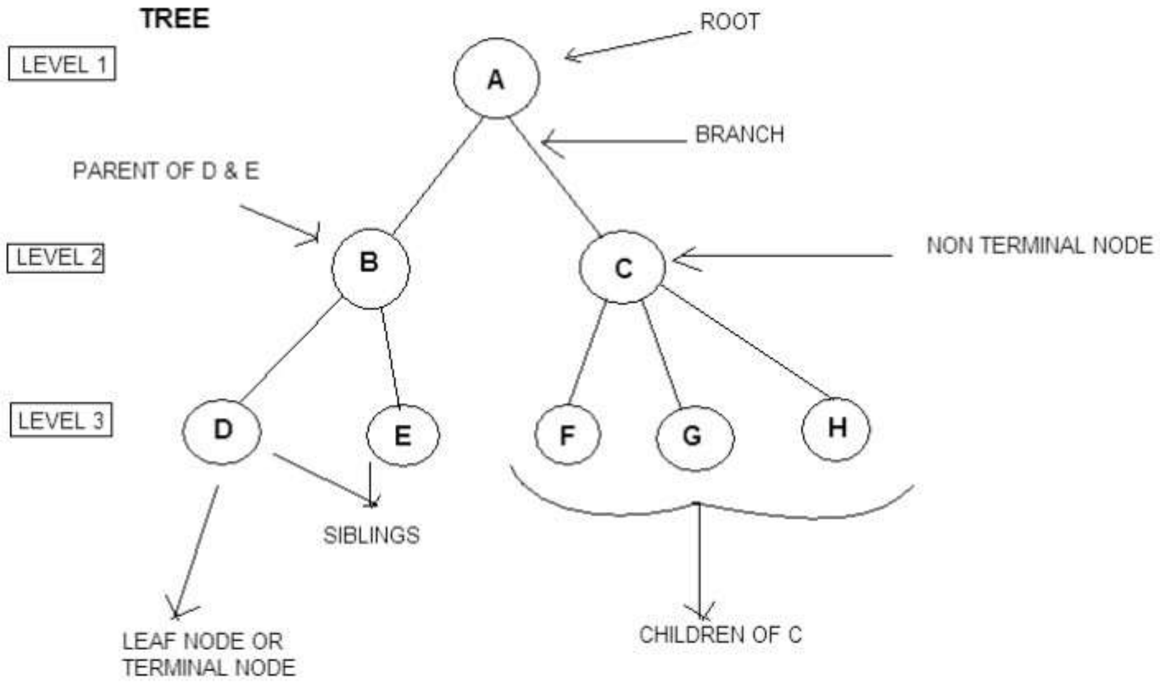
The nodes other than root node and leaf nodes.

Predecessor

Consider the node X, then the node previous to node X is called predecessor node.

Successor

Consider the node X, then the node that comes next to node X is called successor node.

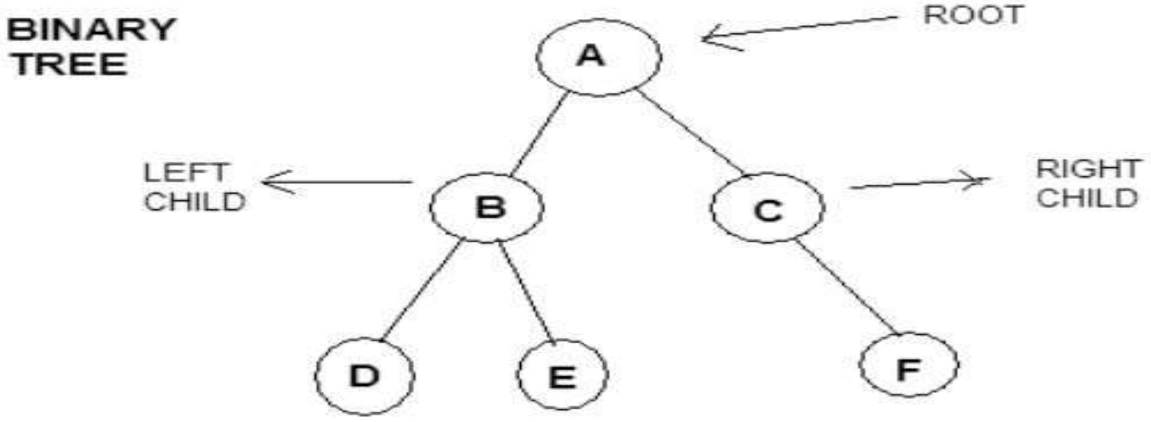


BINARY TREES

A binary tree is a finite set of nodes that is either 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.



TYPES OF BINARY TREES

Skewed Binary tree

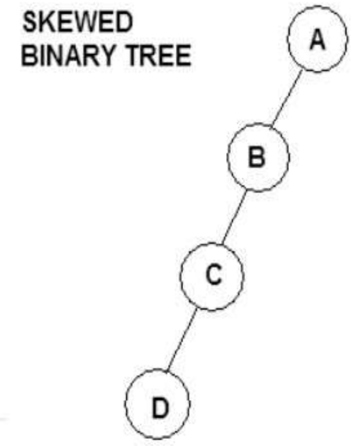
If the new nodes in the tree are added only to one side of the binary tree then it is a skewed binary tree.

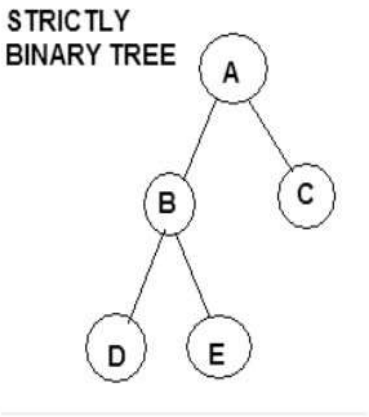
Strictly binary tree

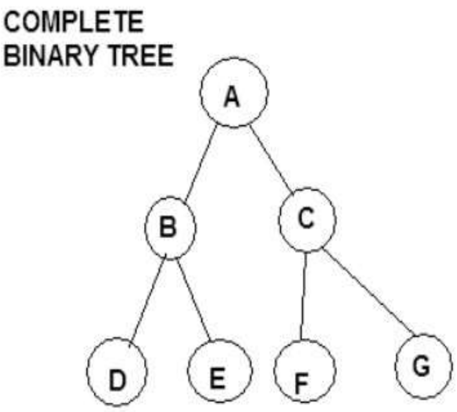
If the binary tree has each node consisting of either two nodes or no nodes at all, then it is called a strictly binary tree.

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.







THE ABSTRACT DATA TYPE OF BINARY TREES

```
Abstract datatype Binary_tree
{
   instances:
        a finite set of nodes either empty or consisting of a root node, left
        Binary_tree, right Binary_tree
  operations:
        for all bt,bt1,bt2 Є Bintree, item Є element
        Bintree create()            - creates an empty binary tree
```

Boolean Isempty(bt) - if(bt==empty) return true else return false

Bintree Makebt((bt1,item,bt2) - return binary tree whose left subtree is bt1 and whose right subtree is bt2and whose root node contains data item.

Bintree Lchild(bt) - if(Isempty(bt) return error else return the left subtree of bt.

Bintree Rchild(bt) - if(Isempty(bt) return error else return the right subtree of bt.

Bintree Data(bt) - if(Isempty(bt) return error else return the data in the root node of bt.

}

PROPERTIES OF BINARY TREES

Some of the important properties of a binary tree are as follows:

1. If *h* = height of a binary tree, then

    a. Maximum number of leaves = 2h

    b. Maximum number of nodes = 2h + 1 – 1

2. If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l + 1.

3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2l node at level l.

4. The total number of edges in a full binary tree with n node is n - 1.

BINARY TREE REPRESENTATION

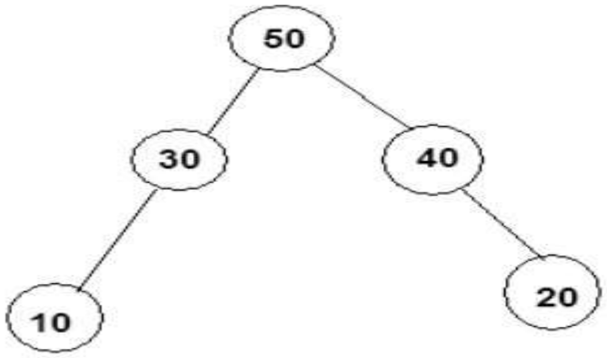
There are two ways in which a binary tree can be represented. They are:

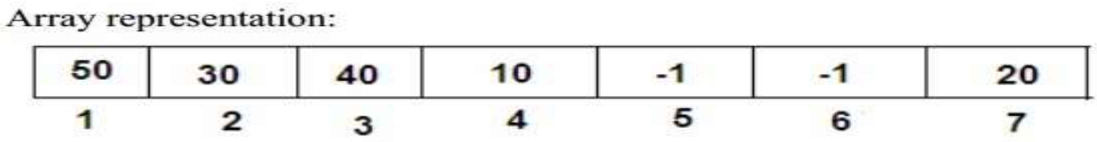
1. Array representation of binary trees.
2. Linked representation of binary trees.

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 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. The formulas for identifying the parent, left child and right child of a particular node.

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

Left child(i) = 2i, if 2i ≤ 2 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 ≤ 2 n. If 2i + 1 > n, then i has no right child.

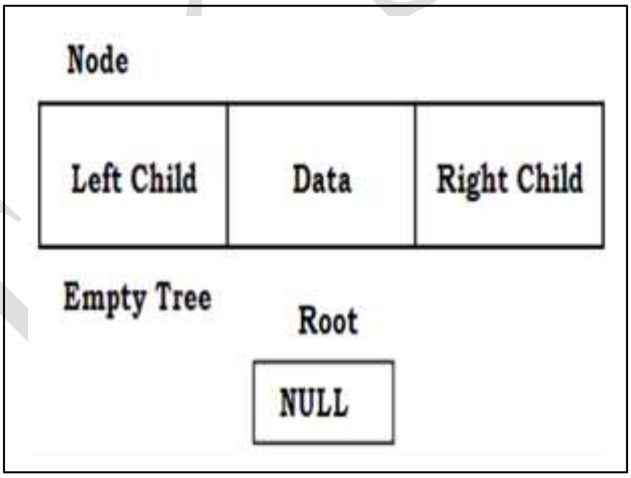
The empty positions in the tree where no node is connected are represented in the array using -1, indicating absence of a node. Using the formula, we can see that for a node 3, the parent is 3/2 is 1. Referring to the array locations, we find that 50 is the parent of 40. The left child of node 3 is

2*3 is 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 is 7. The position 7 in the array consists of 20. Hence, 20 is the right child of 40.

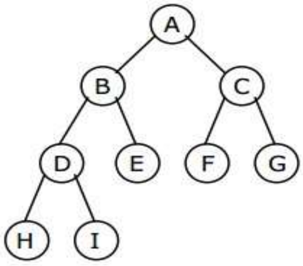
LINKED REPRESENTATION OF BINARY TREES

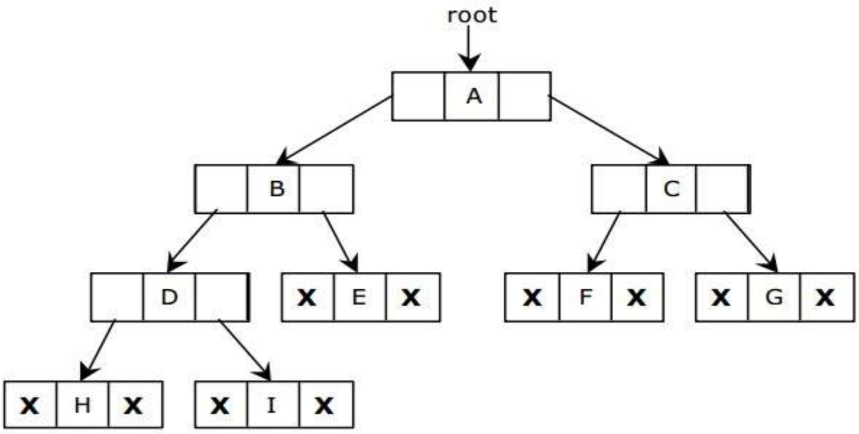
In linked representation of binary trees, instead of arrays, pointers are used to connect the various nodes of the tree. Hence each node of the binary tree consists of three parts namely, the data, left and right. The data part stores the data, left part stores the address of the left child and the right part stores the address of the right child.

```
struct binarytree
{
        struct binarytree *LeftChild;
        int data;
        struct binarytree *RightChild;
};
struct binarytree node;
node *root = NULL;
```



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

A tree traversal is a method of visiting every node in the tree. By visit, we mean that some type of operation is performed. For example, we may want to print the contents of the nodes. 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

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.

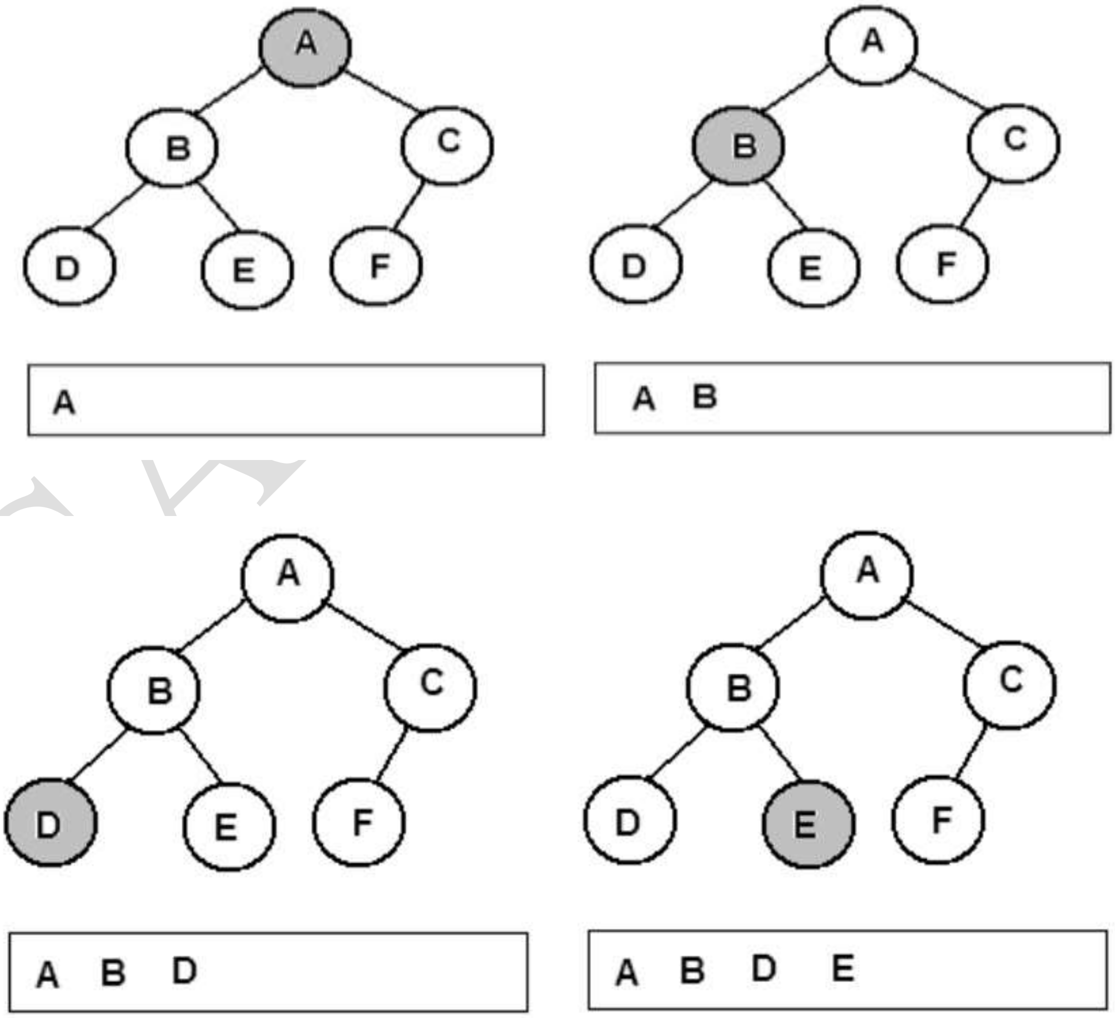
Postorder Traversal
(1) Traverse the left subtree of R in postorder.
(2) Traverse the right subtree of R in postorder.
(3) Process the root R.

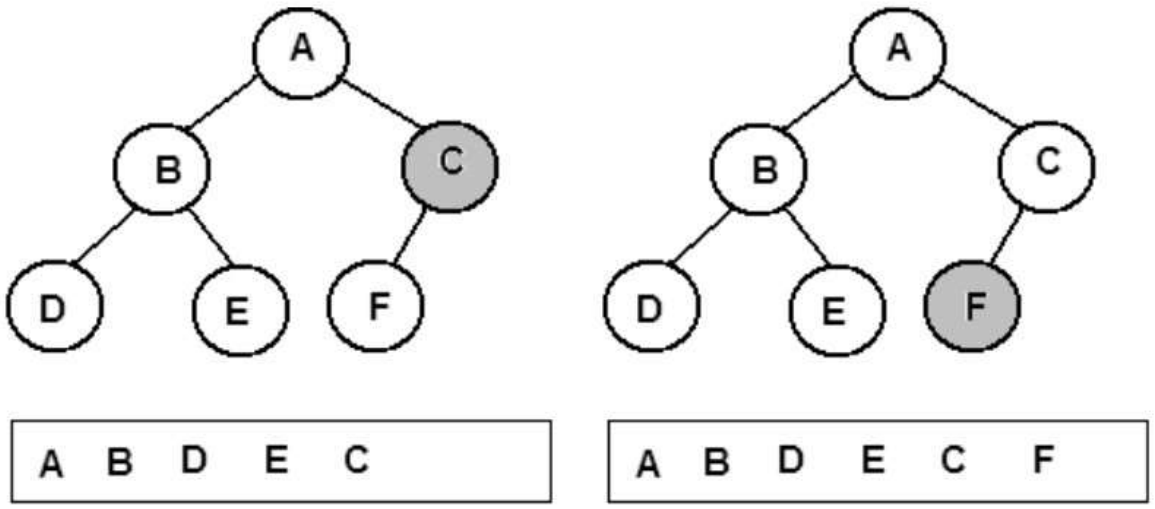
Observe that each algorithm contains the same three steps, and that the left subtree of R is always traversed before the right subtree. The difference between the algorithms is the time at which the root R is processed. The three algorithms are sometimes called the node-left-right (NLR) traversal, the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

Preorder Traversal

In the preorder traversal, the node element is visited first and then the left subtree of the node and then the right subtree of the node is visited. Consider 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.

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.





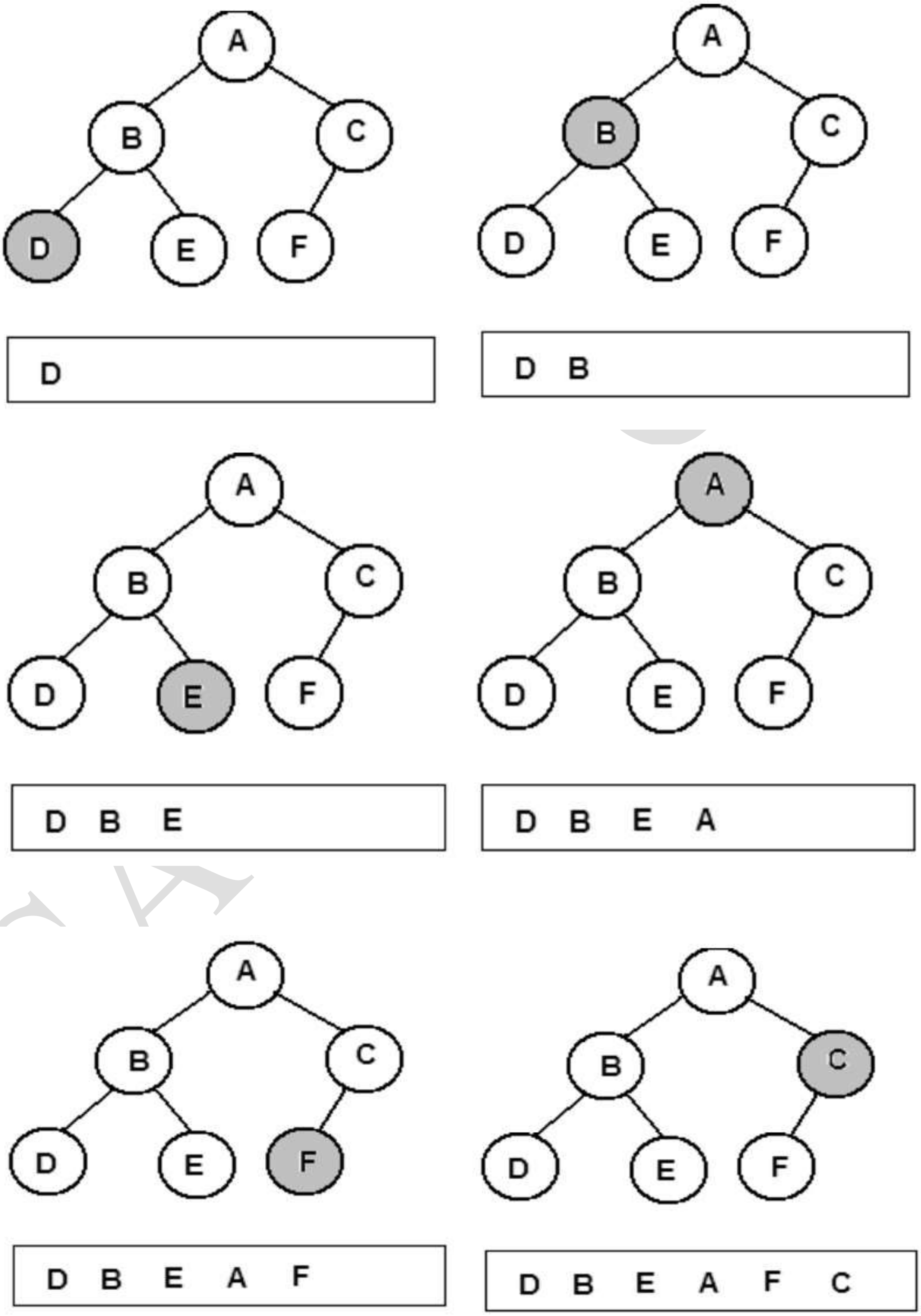
Algorithm for Preorder

```
PREORDER( ROOT )
Temp = ROOT
If temp = NULL
        return
display  temp -> data
If  temp - > left ≠ NULL
                PREORDER ( temp - > left )
If temp -> right  ≠ NULL
        PREORDER ( temp - > right )
```

Inorder Traversal

In the Inorder traversal method, the left subtree of the node element is visited first and then the node element is processed and at last the right subtree of the node element is visited. For 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 here 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.

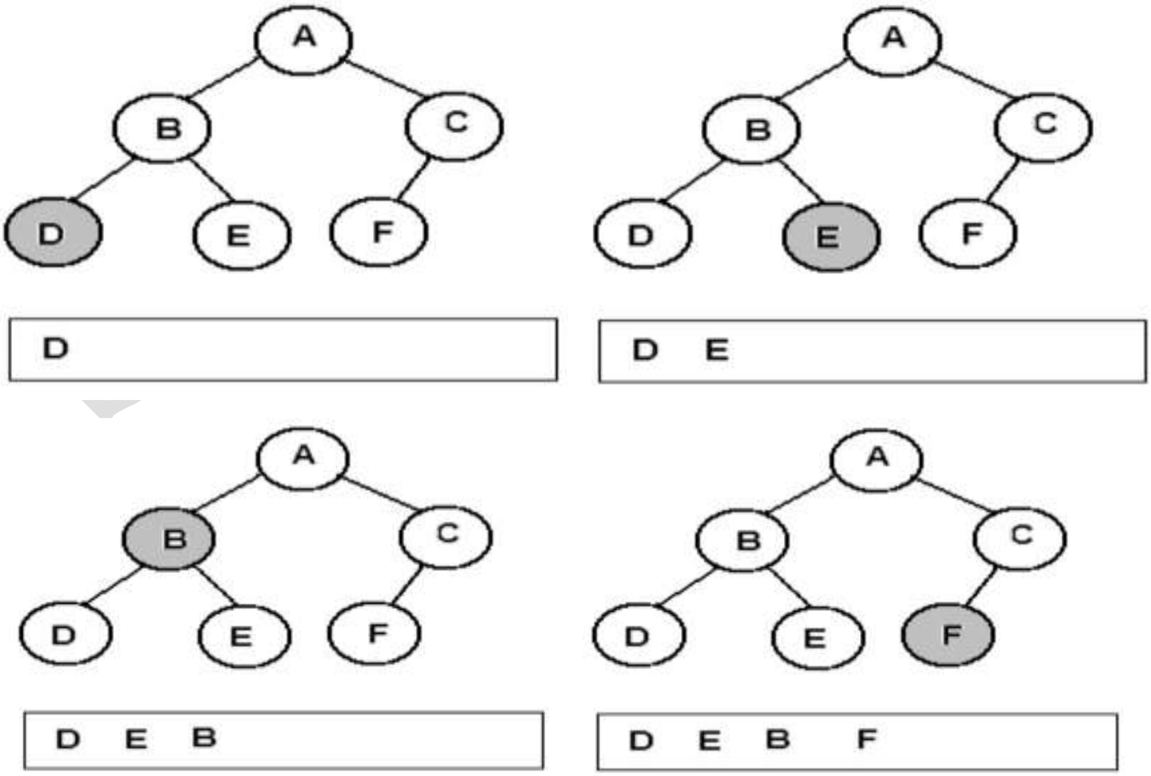


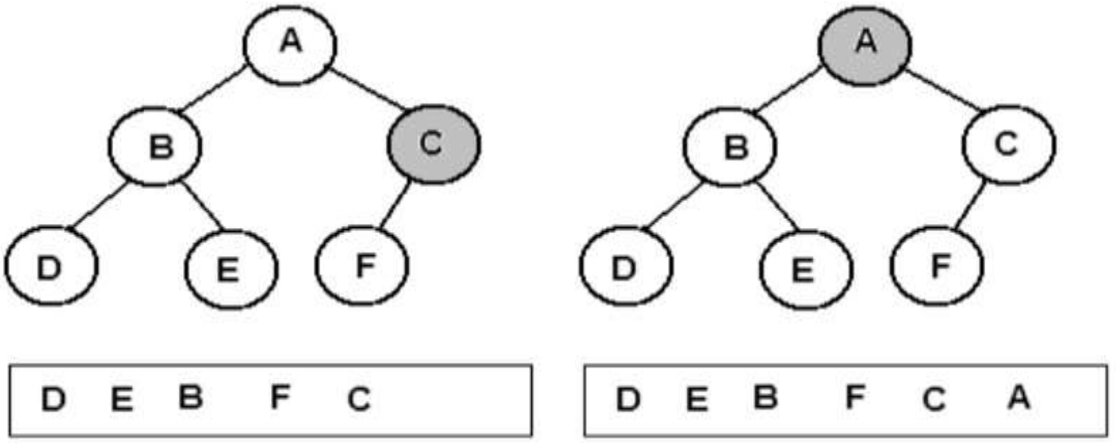
Algorithm for Inorder

```
INORDER( ROOT )
Temp = ROOT
If temp = NULL
        return
If  temp - > left ≠ NULL
                INORDER ( temp - > left )
display  temp -> data
If temp -> right  ≠ NULL
        INORDER ( temp - > right )
```

Postorder Traversal

In the postorder traversal method the left subtree is visited first, then the right subtree and at last the node element is processed. For 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 node element A is processed.





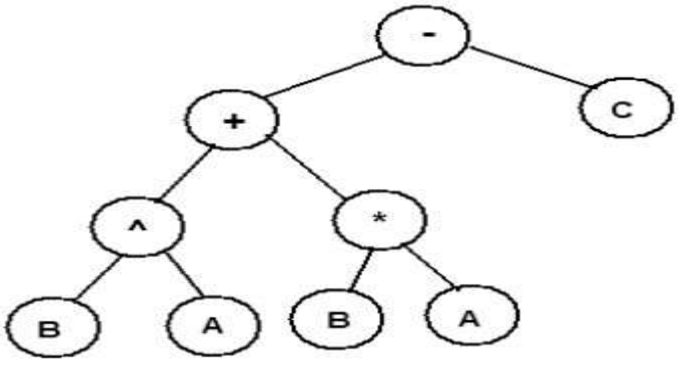
Algorithm for Postorder

```
POSTORDER( ROOT )
Temp = ROOT
If temp = NULL
        return
If  temp - > left ≠ NULL
                POSTORDER ( temp - > left )
If temp -> right  ≠ NULL
        POSTORDER ( temp - > right )
display  temp -> data
```

EXPRESSION TREES

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 –

THREADED BINARY TREES

In binary tree, the leaf nodes have no children. Therefore the left and right fields of the leaf nodes are made NULL. But NULL waste memory space so to avoid NULL in the node we will set threads.

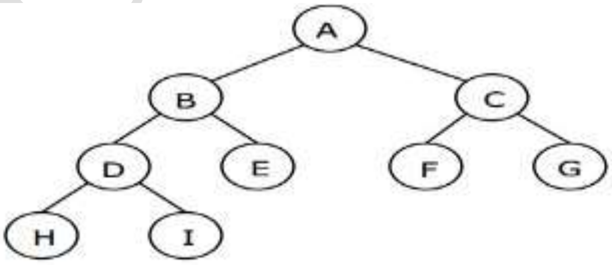
THREADS

Threads are links that point to its predecessor node and successor node. To construct threads we use the following rules.

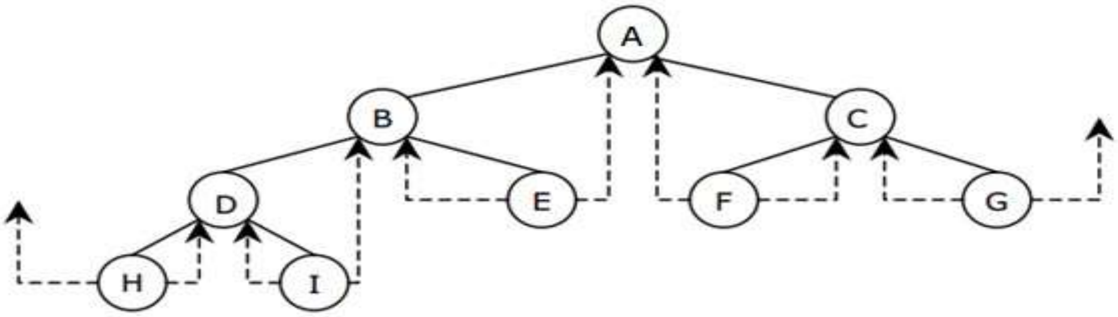
If ptr - > leftchild is NULL, replace ptr - >leftchild with a pointer to its inorder predecessor of ptr

If ptr - > rightchild is NULL, replace ptr - >rightchild with a pointer to its inorder successor of ptr

Let us consider the binary tree as follows



The corresponding threaded binary tree is as follows



The structure of a threaded binary tree is as follows

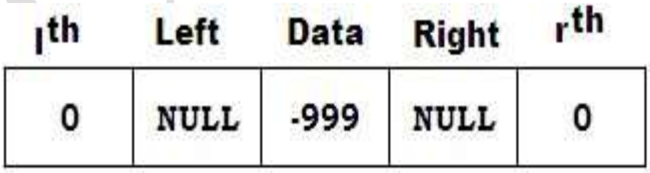
```
struct threadedbtree
{
        int leftthread, rightthread;
        int data;
        struct threadedbtree *leftchild;
        struct threadedbtree *rightchild;
};
```

INORDER TRAVERSAL OF A THREADED BINARY TREE

The basic idea in inorder threaded binary tree is that the left thread should point to the predecessor and the right thread points to inorder successor. The head node is the starting node and the root node of the tree s is attached to the left of the head node.

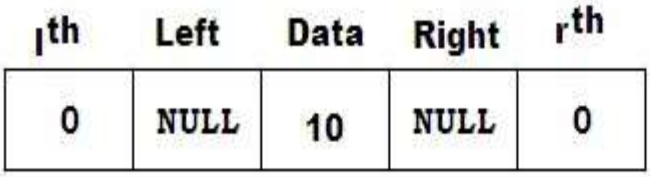
There are two additional fields in each node named as leftthread and rightthread set initially to 0. To explain about inorder thread traversing of a binary tree let us consider the values for creating a threaded binary tree 10, 8, 6, 12, 9, 11, 14

Initially, create a head node of the tree



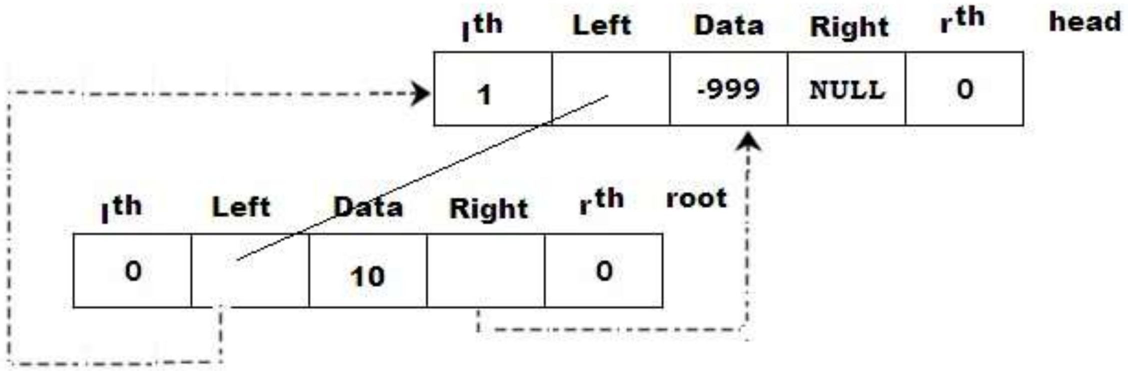
lth	Left	Data	Right	rth
0	NULL	-999	NULL	0

Now let us take the first value 10, this will be the root node and s attached to the left of head node as follows

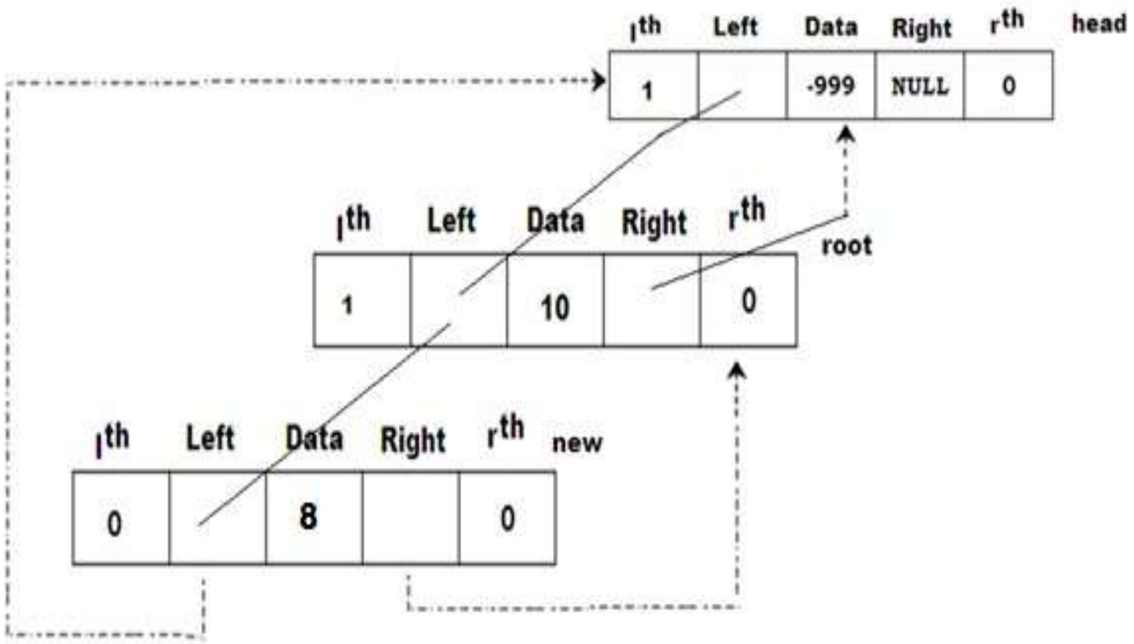


lth	Left	Data	Right	rth
0	NULL	10	NULL	0

The NULL links of the roots left and right will be pointed to the head node as follows



Next comes 8 now 8 is compared with root as it is less then attach 8 as the left child of the root 10.



new - > left = root- > left

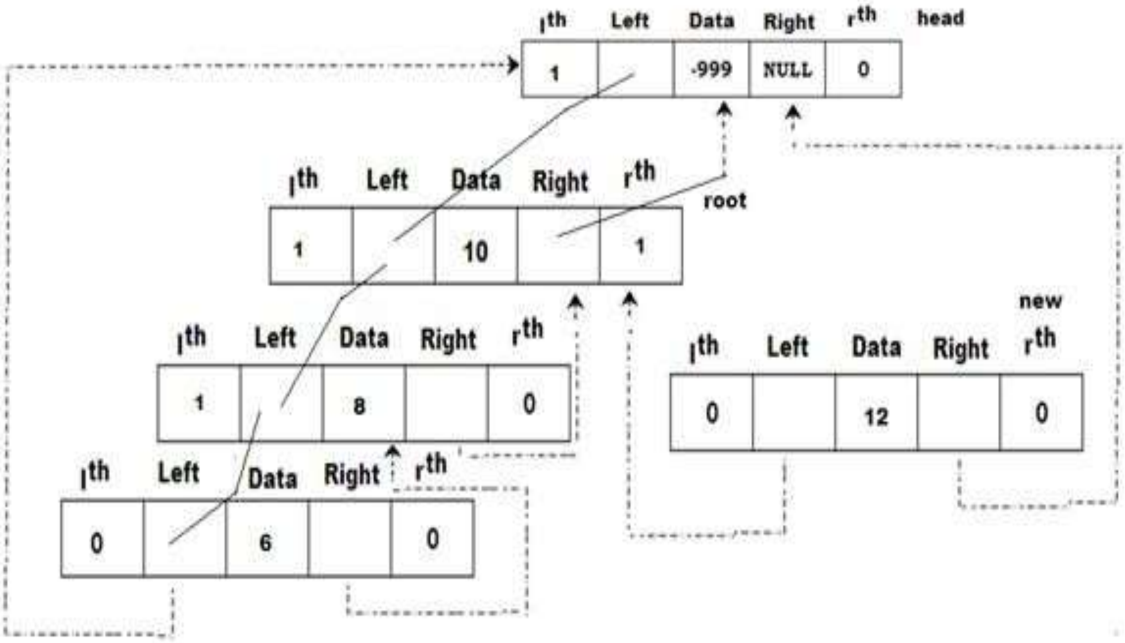
new - > right = root

root - > left = new

root - > lth = 1

The left link of node 8 points to its inorder predecessor and right link of the node 8 points to its inorder successor.

Similarly, the next node 6 is attached to the left of the node 8. The next node is 12 when compared with the root node 10 it is greater so we attach the node 12 to the right of the root node 10 which is as follows.



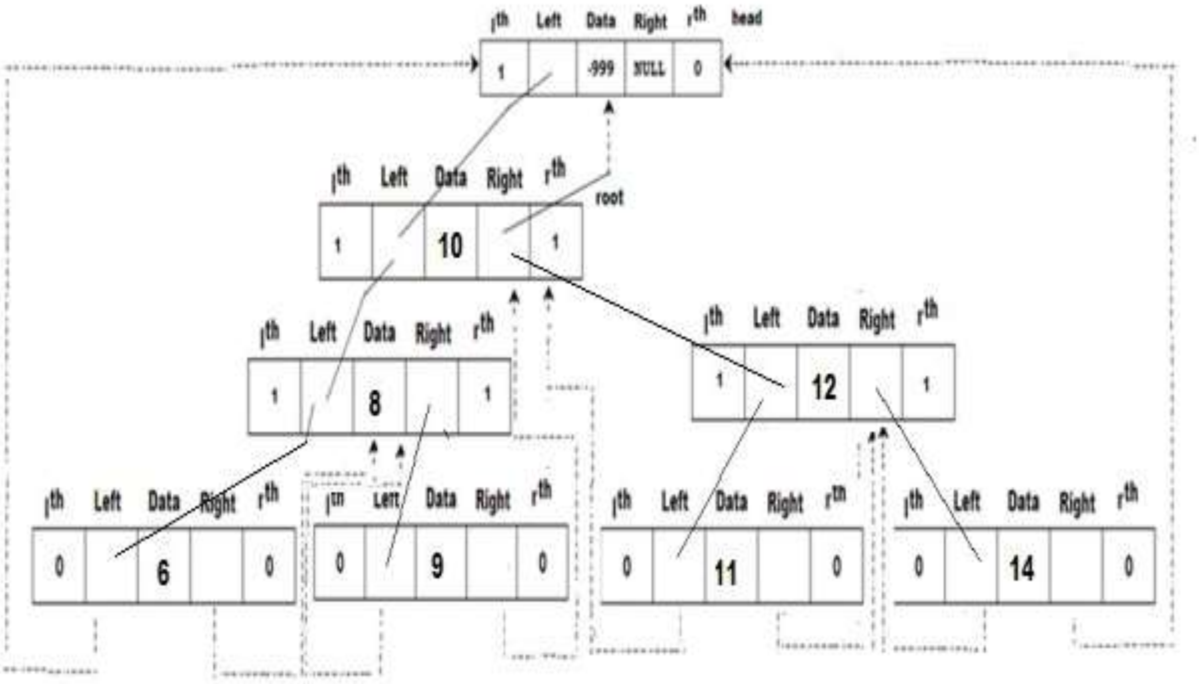
new - > right = root- > right

new - > left = root

root - > rth = 1

root - > right = new

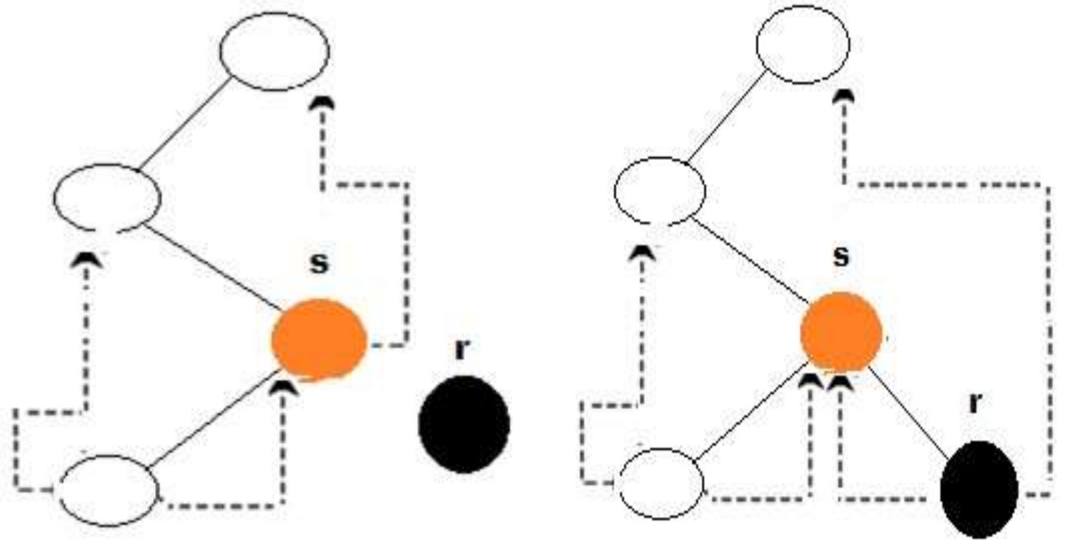
similarly we construct the remaining the nodes to the threaded binary tree by comparing with root node for the nodes 9, 11, 14 as follows



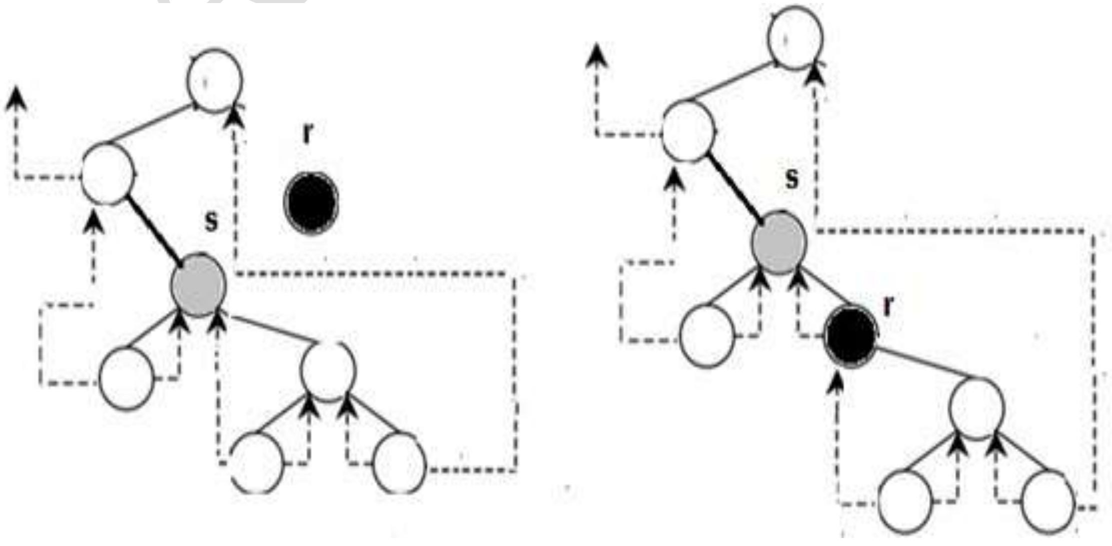
INSERTING A NODE INTO A THREADED BINARY TREE

Let us consider now how to insert the node into the threaded binary tree. The case we consider here is inserting the node **“r”** as the right child of a node **“s”.** The cases for insertion are

If **“s”** has an empty right subtree, then the insertion is simple as shown below



If the right subtree of **“s”** is not empty, then this right subtree is made the right subtree of **“r”** after insertion. Then **“r”** becomes inorder predecessor of a node that has leftthread = = true and consequently there is a thread which has to be updated to point to **“r”**. The node containing this thread was previously the inorder successor of **“s”**.



HEAPS

PRIORITY QUEUES

Heaps are used to implement priority queues. In this type of queues the element to be deleted is one with highest (lowest) priority. We can insert the element at arbitrary priority can be inserted into the queue. The ADT of max priority is as follows.

```
Abstract Datatype MaxPriorityQueue
{
        instances:
                A collection of n>0 elements, each element has a key
        operations:
                for all q Є MaxPriorityQueue, item Є Element, n Є integers
        MaxPriorityQueue create()       - creates an empty dictionay
        Boolean Isempty(q,n)            - if(n>0) return true else return false
        Element top(q,n)                - if(!isempty(q,n)) return an instance of
                                        the largest element in q else return error.
        Element pop(q,n)                - if(!isempty(q,n)) return an instance of
                                        the largest element in q and remove it
                                        from the heap else return error.
        MaxPriorityQueue push(q,item,n) - insert item into pq and return the
                                        resulting priority queue.
}
```

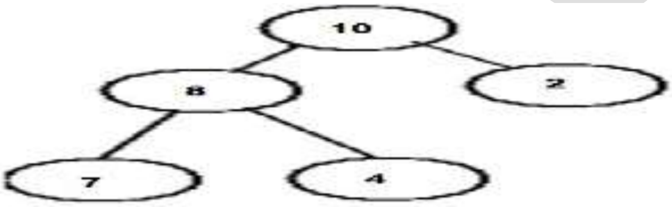
EXAMPLE OF PRIORITY QUEUES

Consider that we are selling the services of a machine. Each user pays a fixed amount per their use. But the time needed by the each user is different. Now we want to maximize the returns from the machine under the assumption that the machine is not idle. This can be maintained by using a priority queue of all persons waiting to use the machine. Whenever the machine becomes idle, the user with the smallest time requirement is selected. Hence a min priority queue is required.

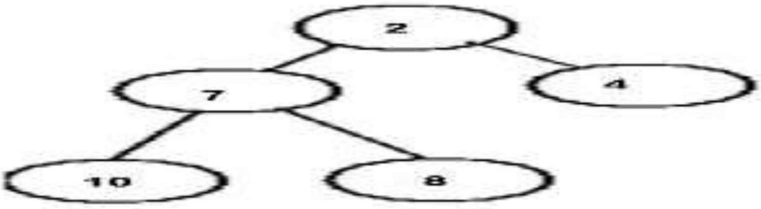
If each user needs the same amount of time on the machine but they are ready to pay different amounts for the service, then a priority queue based on the amount of payment can be maintained. Whenever the machine is idle then the user paying more amount will be selected. This requires a max priority queue.

DEFINITION OF A MAX HEAP

A max heap is a complete binary tree that is also a max tree. A max tree is a tree in which the key value in each node is larger than the key values of its children if any.

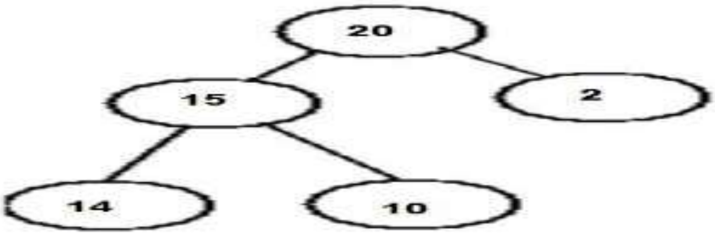


A min heap is a complete binary tree that is also a min tree. A min tree is a tree in which the key value in each node is smaller than the key values of its children if any.

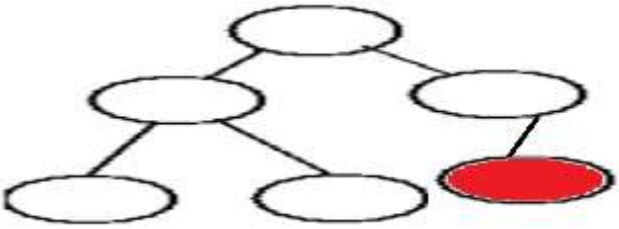


INSERTION INTO A MAX HEAP

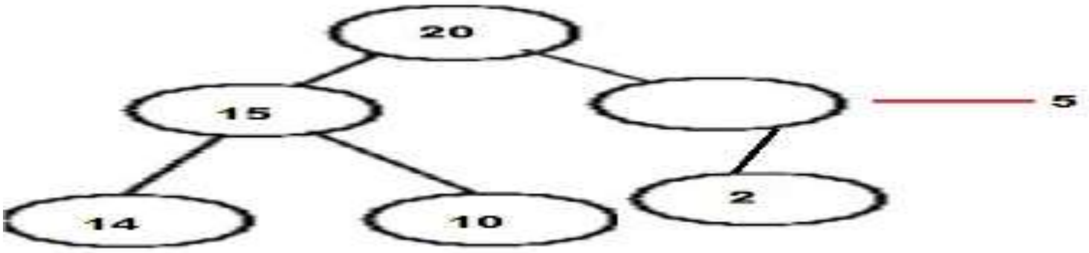
Let us consider a max heap of five elements.



When an element is added to this heap, the resulting is six element heap and it is a complete binary tree. To determine the correct place for the element to be inserted we use bubbling up process that begin at new node and move to the root. The node we want to insert bubbles up to ensure a max heap.

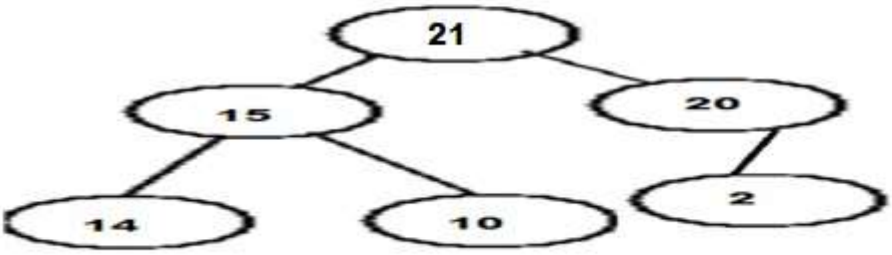


If the element we want to insert is with key value 1, it may be inserted as the left child of 2. But if the key value we want to insert is 5 then we cannot insert as left child of 2 because heap property fails. So 2 is moved down as left child and the place for 5 is the old place of 2.

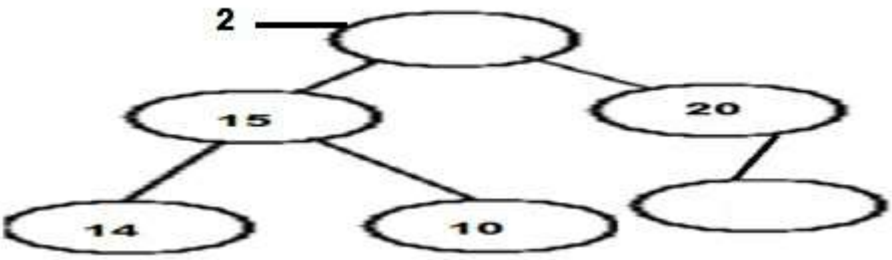


DELETION FROM A MAX HEAP

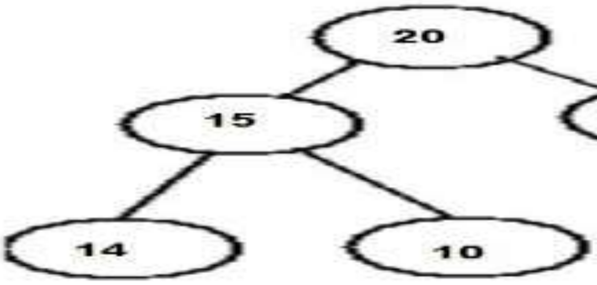
When an element is to be deleted from the max heap it is taken from the root of the heap. For example, a deletion from the heap results in removal of element 21 then the heap will have only five elements.



To do this we remove the element in position 6. Now we have right structure. But the root is vacant and the element 2 is not in the heap. If 2 is inserted into the root then the result binary tree is not max heap.



The element at the root should be largest in the tree apart from left and right child. This element is 20. It is moves to the root and create vacancy at position 3. Since it has no children we insert 2 at this place.



BINARY SEARCH TREES

A dictionary is a collection of pairs, each pair has a key and an associated item. We assume no two pairs have the same key. The ADT of a dictionary is shown below

```
Abstract Datatype dictionary
{
        instances:
                a collection of pairs where n>0 each pair has a key and an
                associated item

        operations:
                for all d Є dictionary, item Є Item, k Є key, n Є integers
        dictionary create()        - creates an empty dictionary
        Boolean Isempty(bt)        - if(n>0) return true else return false
        Element search(d,k)        - return item with key k otherwise return NULL
                                          if no such element.
        Element delete(d,k)        - delete and return item with key k.
        Void insert(d,item,k)      - insert item with key k into d.
}
```

A Binary Search Tree (BST) is a binary tree. It may be empty or it may if not empty than it satisfies the following properties.

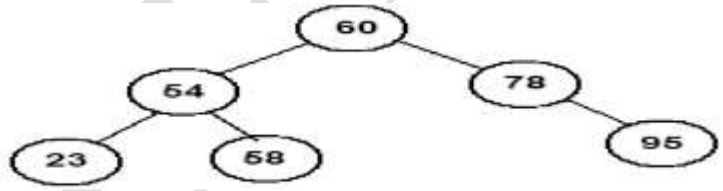
Each node has exactly one key and the keys in the tree are distinct

The keys if any in the left sub tree are smaller than the key in the root

The keys if any in the right sub tree are larger than the key in the root

The left and right sub trees are also binary search trees.

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).

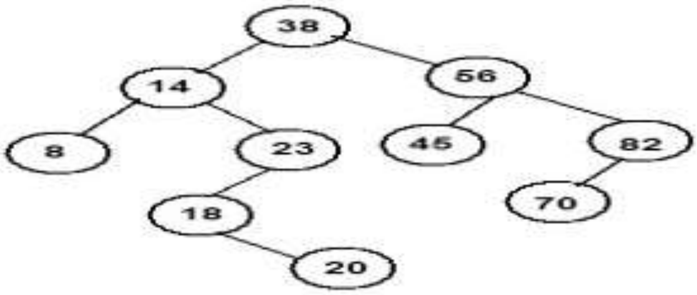


Consider the following list of numbers. A binary search tree can be constructed using this list of numbers, as shown.

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 than 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.

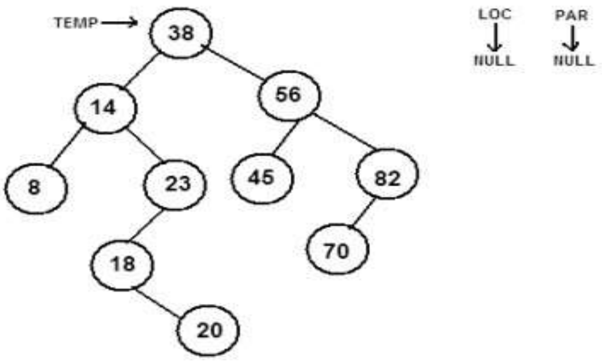


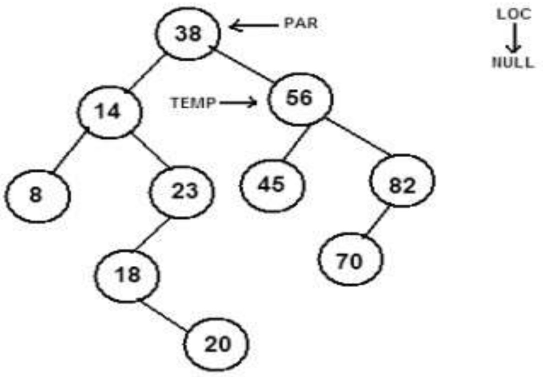
SEARCH OPERATION IN A BINARY SEARCH TREE

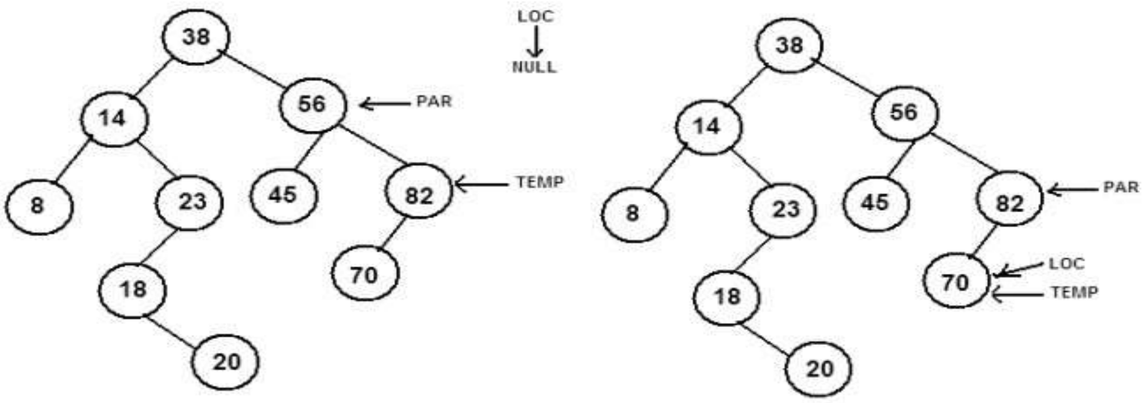
The search operation on a BST returns the address of the node where the element is found. The pointer LOC is used to store the address of the node where the element is found. The pointer PAR is used to point to the parent of LOC. Initially the pointer TEMP is made to point to the root node.

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 than 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.







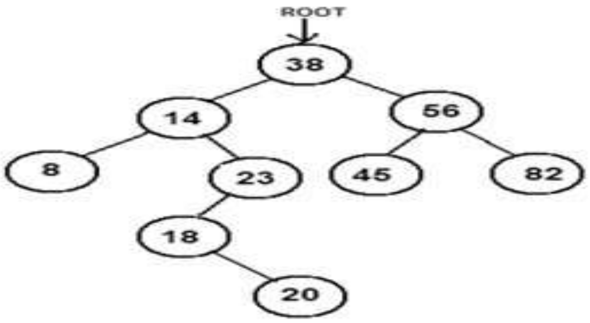
Algorithm SEARCH(ROOT, k)

```
temp = ROOT, par = NULL, loc = NULL
While temp ≠ NULL
        If k = temp - > data
                loc = temp
                break
        If k < temp - > data
                par = temp
                temp = temp - > left
        else
                par = temp
                temp = temp - > right
```

INSERT INTO IN A BINARY SEARCH TREE

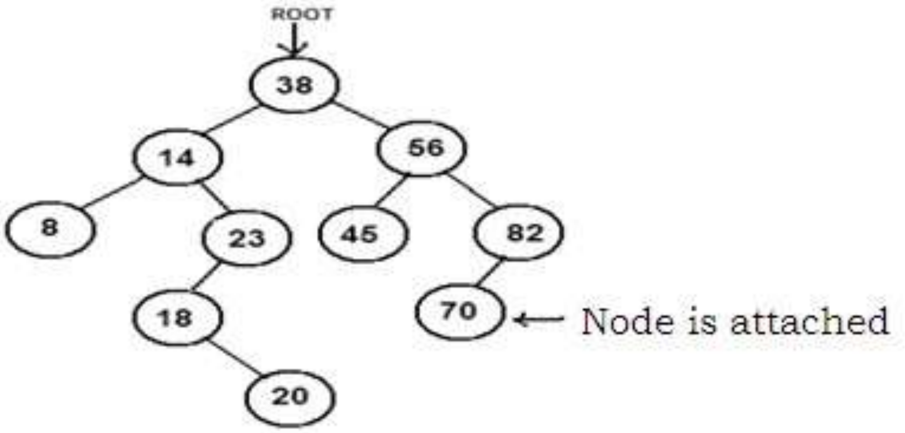
The BST itself is constructed using the insert operation described below. Consider the following list of numbers. A binary tree can be constructed using this list of numbers.

38, 14, 8, 23, 18, 20, 56, 45, 82.



For example we want to insert the element is 70. While inserting a node into the binary search tree first we have find the appropriate position in the binary search tree. We start comparing the node value 70 with the root if it is greater than the root then it is inserted on the right branch of the root else on the left branch of the root.

Now compare the node 70 with root node 38. As node 70 is greater than the root 38 we will move to the right subtree. Now compare node 70 with the node 56 as it greater then move to right and compare node 70 with node 82 as it less than the node 82 we attach 70 as left child of node 82. The diagram is shown below.



Algorithm INSERT(ROOT, k)

1. Read the value for the node which is to be created and store it in a node called new.

2. Initially if(root!=NULL) then root = new

3. Again read the next value of node created in new

4. If(new - > data < root - > data) then attach the new node as a left child of root otherwise attach the new node as a right child of root

5. Repeat step3 and 4 for constructing required binary search tree completely.

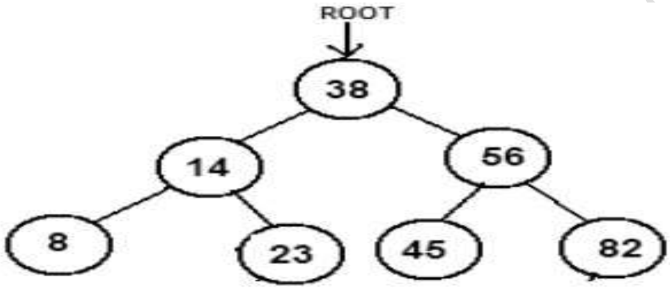
DELETION FROM A BINARY SEARCH TREE

The deletion of a node from a binary search tree occurs with three possibilities

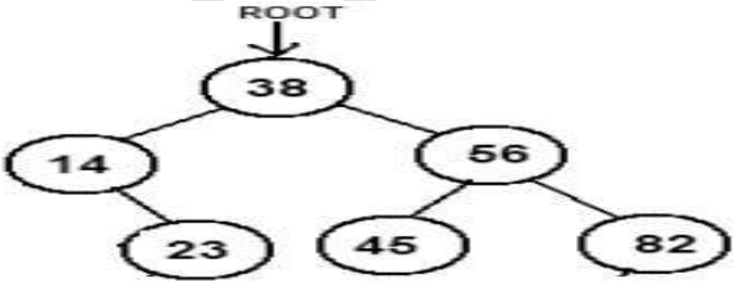
1. Deletion of a leaf node.
2. Deletion of a node having one child.
3. Deletion of a node having two children.

1. Deletion of a leaf node

This is the simplest deletion in which we set the left and right pointer of parent node as NULL. For example consider the binary search tree as follows.



From the above tree diagram the node we want to delete is the node 8, then we will set the left pointer of its parent (node 14) to NULL. Then after deletion the binary search tree is as follows.

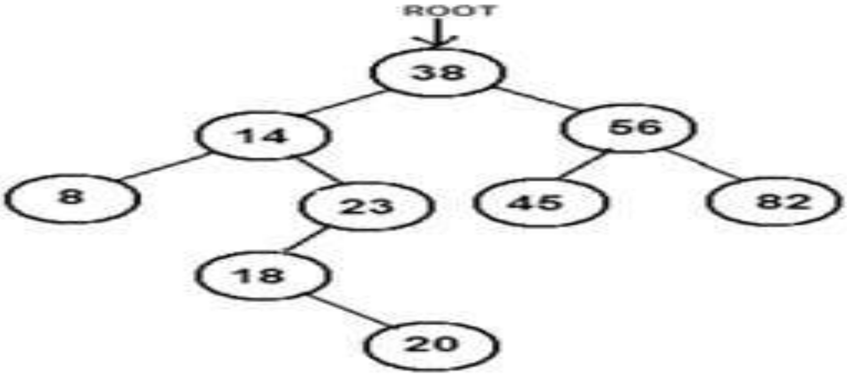


Algorithm

```
if(temp - > left == NULL && temp - > right == NULL)
    if(parent - > left == temp)
        parent - > left = NULL
    else
        parent - > right = NULL
```

2. Deletion of a node having one child

The node if we want to delete is having only one child (i.e., either left or right child). From the diagram the node we want to delete is having the value 23 then we simple copy node 18 at the place of 23 and set the node free.

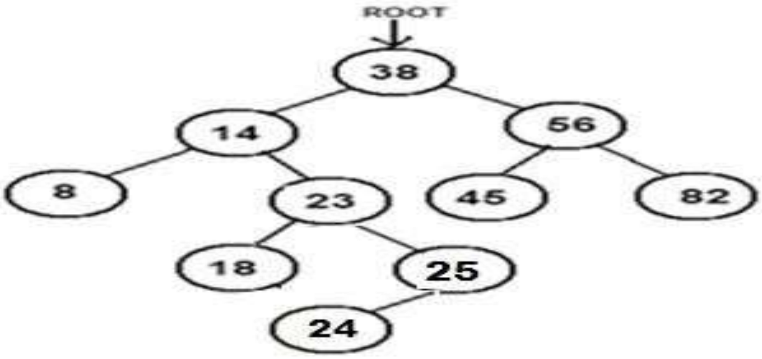


Algorithm

```
if(temp - > left !=NULL && temp - > right == NULL)
    if(parent - > left ==temp)
        parent - > left = temp - > left
    else
        parent - > right = temp - > left
    temp == NULL
    delete temp
```

3. Deletion of a node having two children

The node if we want to delete is having two children. From the diagram the node we want to delete is having the value 23 then we find the inorder successor of the node 23 and it is copied at the place of 23 and set the node 25 left pointer to NULL.



Algorithm

```
if(temp - > left != NULL && temp - > right != NULL)
    parent = temp
    temp_succ = temp - > right
```

```
while(temp_succ - > left != NULL)
        parent = temp_succ
        temp_succ = temp_succ - > left
        temp - > data = temp_succ - > data
        parent  - > right = NULL
```

HEIGHT OF A BINARY SEARCH TREE

The height of a binary search tree with “n” elements can become as large as “n”. For instance, when the values like 1, 2, --------n are inserted into the empty binary search tree. If insertions nd deletions are made at random then the height of the binary search tree is O(log n) on average.

Search trees with worst case height of O(log n) are called balanced search trees. These trees permit insertions, deletions and searches to be performed at time O(h). For example, AVL trees, Red / Black Trees, B-Trees, 2 – 3 Trees etc.